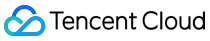
Tencent Cloud Observability Platform

Terminal Performance Monitoring Pro

Product Documentation

Copyright Notice

Trademark Notice



Service Statement

This document is intended to provide users with general information about Tencent Cloud's products and services only and does not form part of Tencent Cloud's terms and conditions. Tencent Cloud's products or services are subject to change. Specific products and services and the standards applicable to them are exclusively provided for in Tencent Cloud's applicable terms and conditions.

Contents

Terminal Performance Monitoring Pro
RUM Pro Introduction
Overview
Strengths
Scenarios
Integration Guide
Quick Integration
Android SDK Integration Guide
SDK Integration Overview
SDK Integration
SDK Initialization
Data Reporting Verification
API Description
iOS SDK Integration Guide
SDK Integration Overview
SDK Integration
SDK Initialization
Data Reporting Verification
API Description
Flutter SDK Integration Guide
SDK Integration Overview
SDK Integration
SDK Initialization
Data Reporting Verification
API Description
React Native SDK Integration Guide
SDK Integration Overview
SDK Integration
SDK Initialization
Data Reporting Verification
Log SDK Integration Guide (Android)
SDK Integration Overview
SDK Integration
SDK Initialization
Data Reporting Verification

API Description
Log SDK Integration Guide (iOS)
SDK Integration Overview
SDK Integration
SDK Initialization
Data Reporting Verification
API Description
Console Operation Guide
Metrics Description
Crash
ANR
FOOM/OOM
Error
Lag
Memory
Launch
Network
Traffic
Page Performance
Page View
Custom Event
Custom Speed
Log
Setting
SDK configuration
Symbol Table
Overview
Upload the symbol table
Symbol table classification
Resource Management
Data Analysis
Overview
business drill-down
Query
Custom Data
Attachment
FAQs

Terminal Performance Monitoring Pro
RUM Pro Introduction
Overview

Last updated：2026-05-25 18:01:54

Terminal Performance Monitoring Pro (RUM Pro) is a one-stop quality monitoring and management platform for client applications. It provides comprehensive crash analysis, performance monitoring, and exception alarm capabilities for your client applications (covering platforms such as Android, iOS, Windows, and Flutter). It aims to ensure application stability and smoothness, significantly reduce the risk of user churn, and enhance user experience and brand reputation.

Key Features of RUM Pro

Multi-Dimensional Crash Diagnosis

RUM Pro captures critical issues in mobile applications, including native crashes, Application Not Responding (ANR) errors, and exceptions in real time. It automatically collects stack traces, thread states, device information, and memory snapshots.

Intelligent Lag Analysis

RUM Pro performs in-depth tracing for main thread blocking issues and identifies time-consuming functions and operation paths to pinpoint the root causes of user interface lag and unresponsive interactions.

Network Performance Analysis

RUM Pro monitors network request details, including request duration, success rate, data size, DNS lookup time, and connection establishment time, to assist in optimizing network interaction experience.

Intelligent Root Cause Analysis

RUM Pro utilizes massive data to intelligently aggregate crash points and stack call paths, precisely locating the code files, line numbers, and specific methods where crashes occur, while correlating relevant system logs, application versions, device models, and network environment information.

Trend Analysis and Version Comparison

RUM Pro clearly displays real-time trends and changes in crash rates, supporting comparison of crash metrics across different application versions to quickly evaluate the stability of new releases.

Comprehensive Log Collection

RUM Pro supports custom log recording and upload. Contextual information is collected when a crash occurs to significantly improve troubleshooting efficiency.

Core Performance Metric Monitoring

RUM Pro monitors key performance indicators (KPIs), including application startup time, page loading time, frames per second (FPS), memory usage, data consumption, network request duration and success rate, and battery consumption.

Automatic Symbol Table Management

RUM Pro integrates the feature to upload and parse symbol files to ensure accurate symbolization of crash stacks, clearly restoring readable code information.

Application Version Management

RUM Pro allows you to easily manage the operational data and issue feedback for various application versions.

Mainstream Platform Support

RUM Pro maintains consistent compatibility with mainstream development platforms and application frameworks such as Android, iOS, Windows, Unity, Flutter, and React Native.

Fine-Grained Data Filtering

RUM Pro supports multi-dimensional filtering and analysis of crash, lag and abnormal performance data by app version, device model, operating system, region, network environment, user identifier, and so on.

Real-Time Alarms

RUM Pro allows you to configure threshold alarm rules for events such as sudden crash rate spikes, ANR rate threshold breaches, and abnormal key performance indicators. It sends immediate push notifications via email, SMS, WeChat, WeCom, and other channels to ensure relevant teams can respond and handle the issues in a timely manner.

Strengths

Last updated：2026-05-25 18:01:54

Mobile Quality Expertise

With years of experience in the mobile application domain, Terminal Performance Monitoring Pro (RUM Pro) can understand mobile-specific performance issues, such as lag and ANR, and stability issues, such as crashes and Out of Memory (OOM) errors. It offers professional capabilities including intelligent crash stack analysis, lag monitoring, ANR detection, and resource monitoring, accurately pinpointing the root causes of issues specific to mobile applications.

Comprehensive Crash Monitoring and Analysis

RUM Pro offers industry-leading real-time crash monitoring and analysis capabilities, supporting the capture of crash information from engines such as Java, native C or C++, and Unity. Through powerful error aggregation, stack trace deobfuscation, and intelligent root cause diagnosis, it helps developers quickly and accurately locate and fix crash issues, significantly reducing the crash rate.

In-Depth Performance Profiling

RUM Pro monitors key performance metrics such as application startup time, page rendering time, network request performance, lag frame rate, and ANR in real time. It offers time consumption analysis, multi-dimensional performance aggregation, and user session tracking, clearly showing performance bottlenecks to ensure application smoothness and enhance user experience.

Real-Time Alarms and Closed-Loop Management

RUM Pro supports intelligent threshold alarms based on key metrics such as crash rate and ANR rate, as well as issue-level alarm tracking for new issues and top issues. It notifies relevant personnel in real time through multiple channels (including email, SMS, WeChat, WeCom, DingTalk, and webhook), and provides closed-loop management features such as issue assignment, status tracking, and resolution verification, enhancing team collaboration and resolution efficiency.

Low-Cost Integration and Stable Services

Built on Tencent's extensive experience with massive service operations, the underlying SDK is powered by TDS-Bugly with over a decade of continuous operation. The SDK is lightweight, highly available, and easy to integrate, with minimal impact on application performance. Leveraging Tencent Cloud's robust infrastructure, RUM Pro delivers highly available and high-throughput data processing services, enabling developers to enjoy a stable and reliable service experience without operational complexity. Flexible pricing models (usage-based packages and monthly active user packages) are available, offering outstanding cost-effectiveness.

Scenarios

Last updated：2026-05-25 18:01:55

Crash Management and Stability Assurance

- Terminal Performance Monitoring Pro (RUM Pro) accurately captures all crash scenarios on mobile devices, including native crashes, ANR, and FOOM. By optimizing the exception capture workflow and moving exception reporting earlier in the process, it addresses the issue of unreported exceptions.
- RUM Pro supports multi-dimensional crash analysis (memory allocation, VMMap, and process status) for iOS and Android applications. Combined with the on-device data and app state restoration technology of minidump and tombstone, it can efficiently identify hidden issues such as app crashes caused by sudden spikes in image memory usage.

Memory Troubleshooting

For complex scenarios such as sudden OOM rate increases, memory leaks, and dangling pointers, RUM Pro provides advanced analysis capabilities including memory detail analysis.

- It automatically captures Java heap dump files and analyzes reference chains of leaking objects.
- It supports the correlation between FOOM and memory leak monitoring, precisely identifying OOM degradation issues caused by factors such as Hippy framework upgrades.

Minute-Level Alarms and Responses

Based on a cloud-native real-time streaming processing architecture, RUM Pro enables:

- Minute-level alarms for critical metrics such as sudden crash rate spikes and ANR threshold breaches.
- Support for complex alarm scenarios such as the emergence of new issues and degradation of top issues.
- Multi-dimensional filtering (by version, device model, and operating system) and false positive prevention mechanisms.
- Mainstream alarm channels such as phone calls, SMS, WeChat, WeCom, DingTalk, and Feishu, along with webhooks. This enables seamless integration with customers' internal platforms, facilitating the establishment of an integrated corporate response system.

End-to-End Performance Optimization

- Lag mitigation: monitors lag in the main and worker threads, and helps pinpoint the root cause of thread blocking with ANR trace and garbage collection (GC) details.
- Resource monitoring: provides real-time alerts for abnormal data traffic and power consumption to prevent resource abuse.
- Startup speed optimization: analyzes page loading performance to identify rendering bottlenecks.

Multi-Platform Release and Verification

- Version quality comparison: supports A/B test drill-down analysis (over 20 custom fields) to accurately evaluate the stability of new versions.
- Cross-platform support: covers platforms such as Android, iOS, Flutter, and Windows, ensuring consistent release quality across platforms.

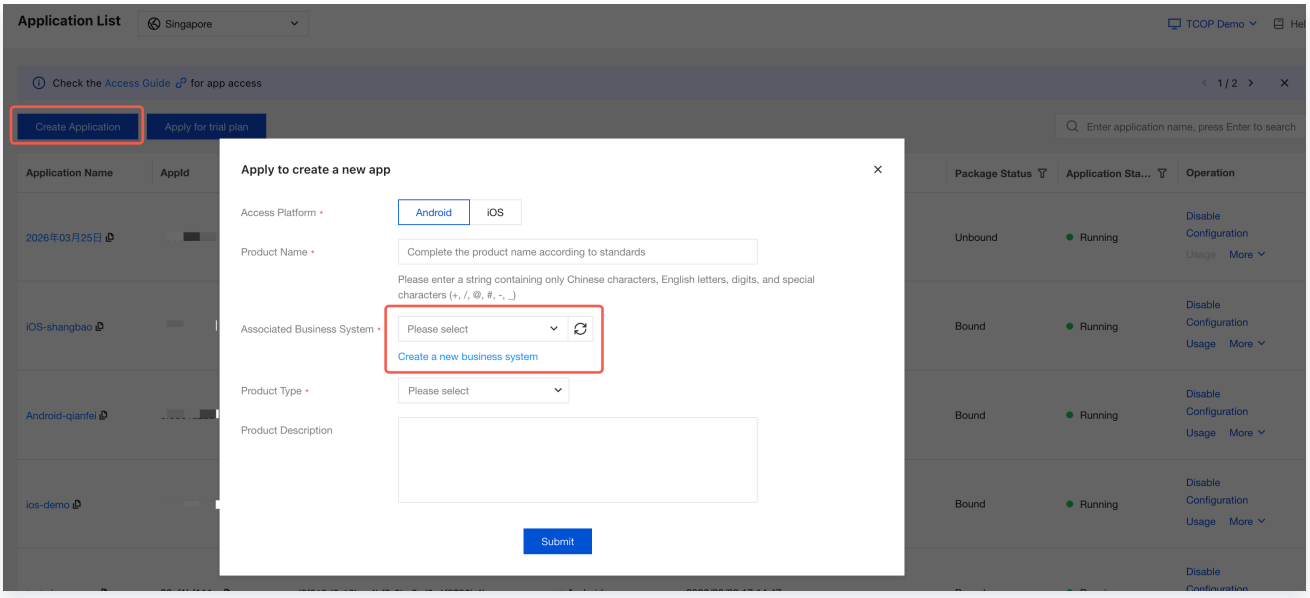
Integration Guide
Quick Integration

Last updated：2026-05-26 10:14:06

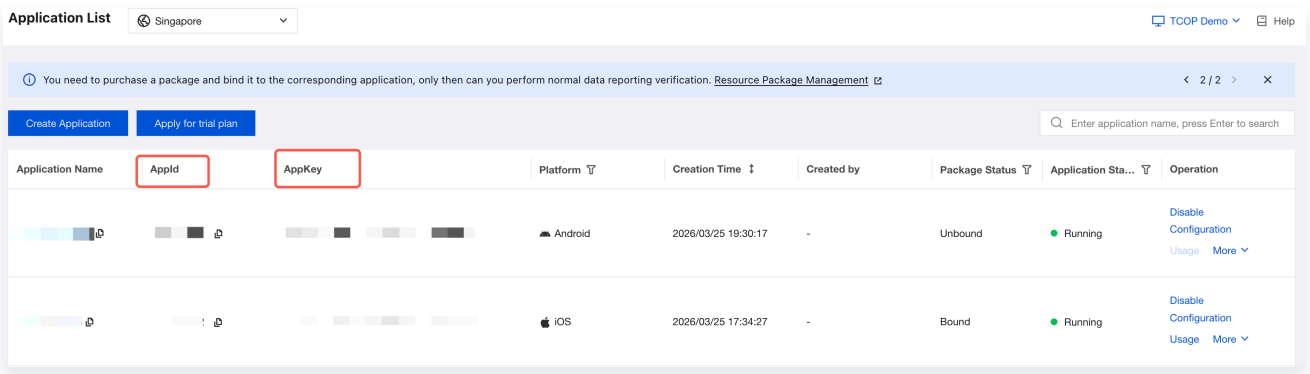
This article helps you quickly integrate the Terminal Performance Monitoring Pro (RUM Pro) SDK and use and verify the data reporting feature.

Creating an Application

1. Log in to the RUM Pro console and go to the **Application List** page.
2. On the Application List page, click **Create Application** and select the business system as required. If you do not have a business system, click **Create a new business system**, enter the information, and create one on the Business System List page.



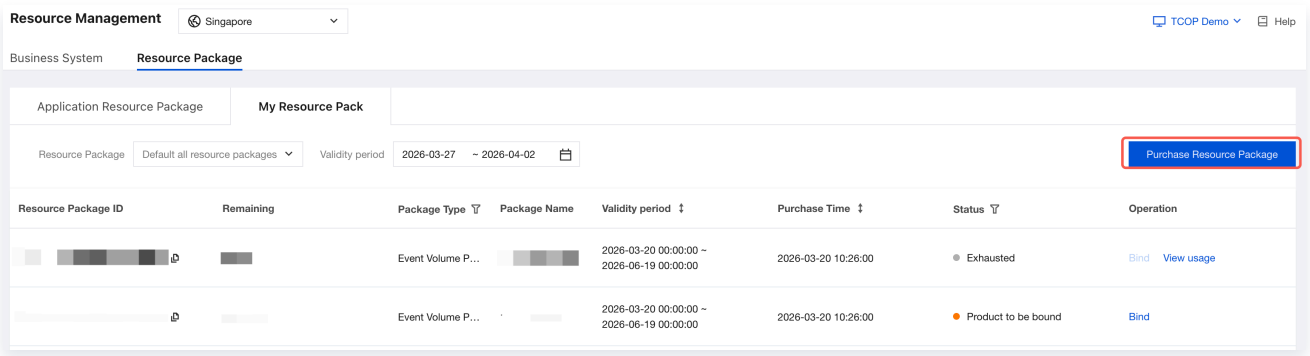
3. After the application is created successfully, you can see the AppID and AppKey assigned by the platform, which serve as the product credentials on the platform.



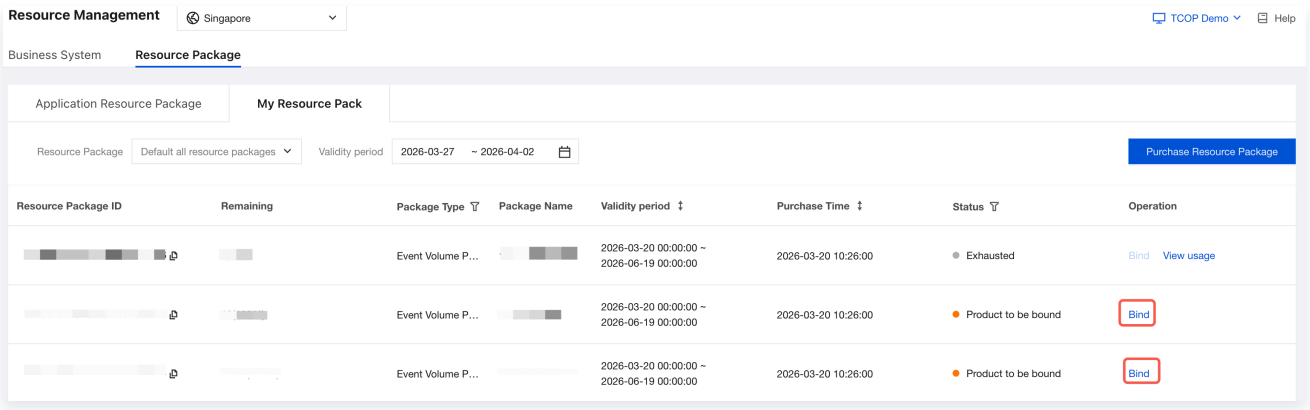
Purchasing a Resource Package

After you create an application in RUM Pro, you need to purchase and bind a resource package to use the application to report data. For more information about resource packages, see Billing Overview. If the application does not have a bound and active resource package, or if the bound resource package has been used up, the application cannot report data.

1. Go to the Resource Management page, choose Resource Package, and click **Purchase Resource Package**. For more information about resource packages, see Billing Overview.



2. After the purchase is complete, click **Bind** in the Operation column to bind the application.



Integrating the SDK

After the application is created successfully, you can refer to the Android SDK Integration Guide, iOS SDK Integration Guide, Flutter SDK Integration Guide, and React Native SDK Integration Guide to integrate the RUM Pro SDK into your application.

Checking the Reporting Feature

After the RUM Pro SDK is integrated into the application according to the guide, you can simulate or trigger relevant events to check the reporting process. RUM Pro supports features such as crash monitoring, ANR monitoring, OOM monitoring, error monitoring, and performance monitoring (startup monitoring, lag monitoring, and memory monitoring).

Different monitoring features are different in their data reporting logic. For abnormal termination monitoring, reporting is enabled globally by default. After the SDK captures an exception, it is reported immediately or upon the next application startup. For performance monitoring, sampling is enabled by default. You can create configuration tasks and select appropriate sampling rates based on your needs.

Crash Monitoring

Crash monitoring is enabled globally by default and does not support sampling. You can simulate a crash, restart the application after the crash occurs, and check whether the Crash/Issue List contains the reported data.

- Android sample code:

```
public void testNPECrash() {
    String value = null;
    System.out.println("Length of str is: " + value.length());
}
```

- iOS sample code:

```
- (void)testNPECrash {
    NSString *str = nil;
    NSInteger length = [str length];
    NSLog(@"Length of str is: %ld", length);
}
```

ANR Monitoring

Android ANR monitoring is enabled globally by default and does not support sampling. iOS ANR monitoring is enabled globally by default and supports sampling.
The following sample code simulates an ANR by executing a long-running task (such as sleeping for 20 seconds) on the UI thread. After the application is killed by the system due to unresponsiveness, restart the application and check whether the ANR/Issue List contains the reported data.

- Android sample code:

```
public void testANR() {
    try {
        Thread.sleep(20000); // Simulates a time-consuming operation.
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
```

- iOS sample code:

```
- (void)startLongRunningTask {
    dispatch_async(dispatch_get_main_queue(), ^{
        // Simulates a long-running task.
        [NSThread sleepForTimeInterval:20.0];
    });
}
```

OOM Monitoring

Android OOM monitoring is enabled globally by default and does not support sampling. iOS OOM monitoring is enabled globally by default and supports sampling.
The following sample code simulates an OOM scenario by continuously allocating memory without releasing it, ultimately causing the application to be killed by the system due to OOM. Restart the application and check whether the OOM/Issue List contains the reported data.

- Android sample code:

```
public void simulateOOM() {
    List<byte[]> list = new ArrayList<>();
    while (true) {
        byte[] bytes = new byte[1024 * 1024]; // Allocates 1 MB of
memory.
        list.add(bytes);
    }
}
```

- iOS sample code:

```
- (void)simulateFOOM {
    NSMutableArray *array = [NSMutableArray array];
    while (true) {
        @autoreleasepool {
            NSString *string = [NSString stringWithFormat:@"%d",
arc4random_uniform(1000000)];
            [array addObject:string];
        }
    }
}
```

Error Monitoring

Error monitoring is enabled globally by default and does not support sampling. The RUM Pro SDK provides a series of APIs that allow you to report custom exceptions and caught exceptions. These exceptions are referred to as errors.
After the application calls the error reporting API to report data, you can directly check whether the Error/Issue List contains the reported data without restarting the application.

Performance Monitoring

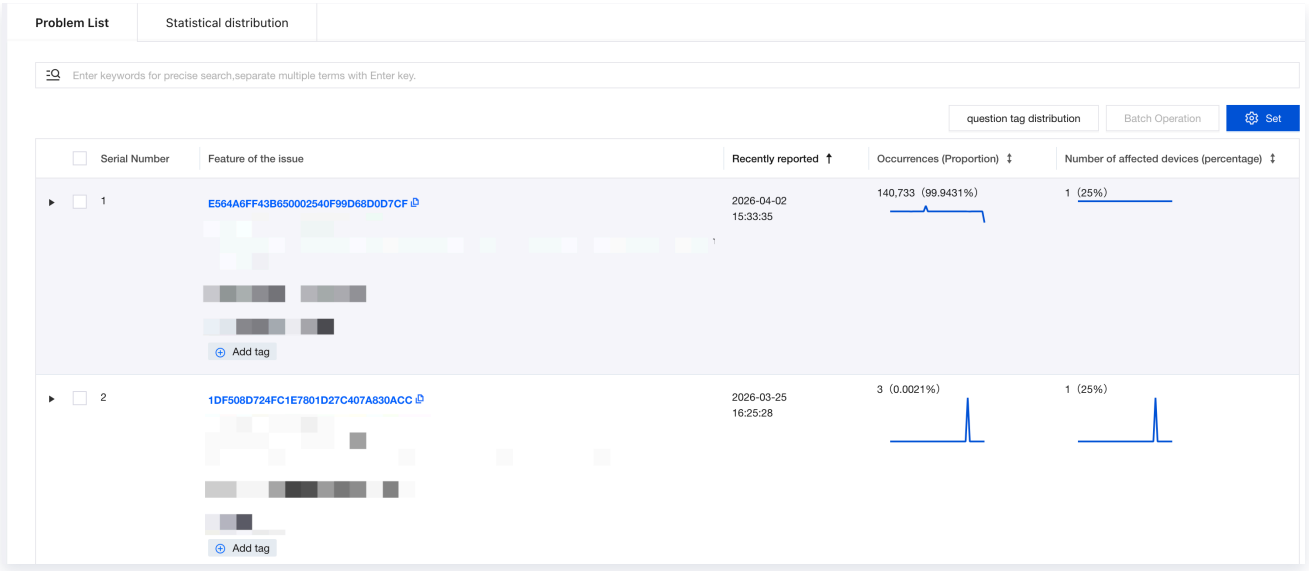
RUM Pro supports lag monitoring, startup monitoring, and memory monitoring. All these performance monitoring features support sampling, and sampling is disabled by default. You need to create configuration tasks in Settings/SDK Configuration to specify the sampling rate for each performance monitoring item.

1. For easier testing during the testing phase, it is recommended to create a dedicated allowlist testing task with all monitoring features set to 100% sampling, which means to set "sample_ratio" and "event_sample_ratio" to 1.
2. Configuration changes generally take 10 minutes to take effect. To ensure the configurations are applied, it is recommended to start the application 10 minutes after the configurations are modified, and then restart the application after a few minutes.
3. After application startup monitoring is enabled, the startup monitoring feature is automatically activated when the application starts. Startup data is reported after SDK initialization.
4. Lag and memory metrics are collected during runtime after monitoring is enabled, and the data is reported upon the next startup.
5. For lag and memory issue monitoring, after monitoring is enabled, when abnormal cases are detected, data is reported immediately if the device is connected to Wi-Fi. Otherwise, data is cached locally and reported upon the next application startup.

Viewing the Data

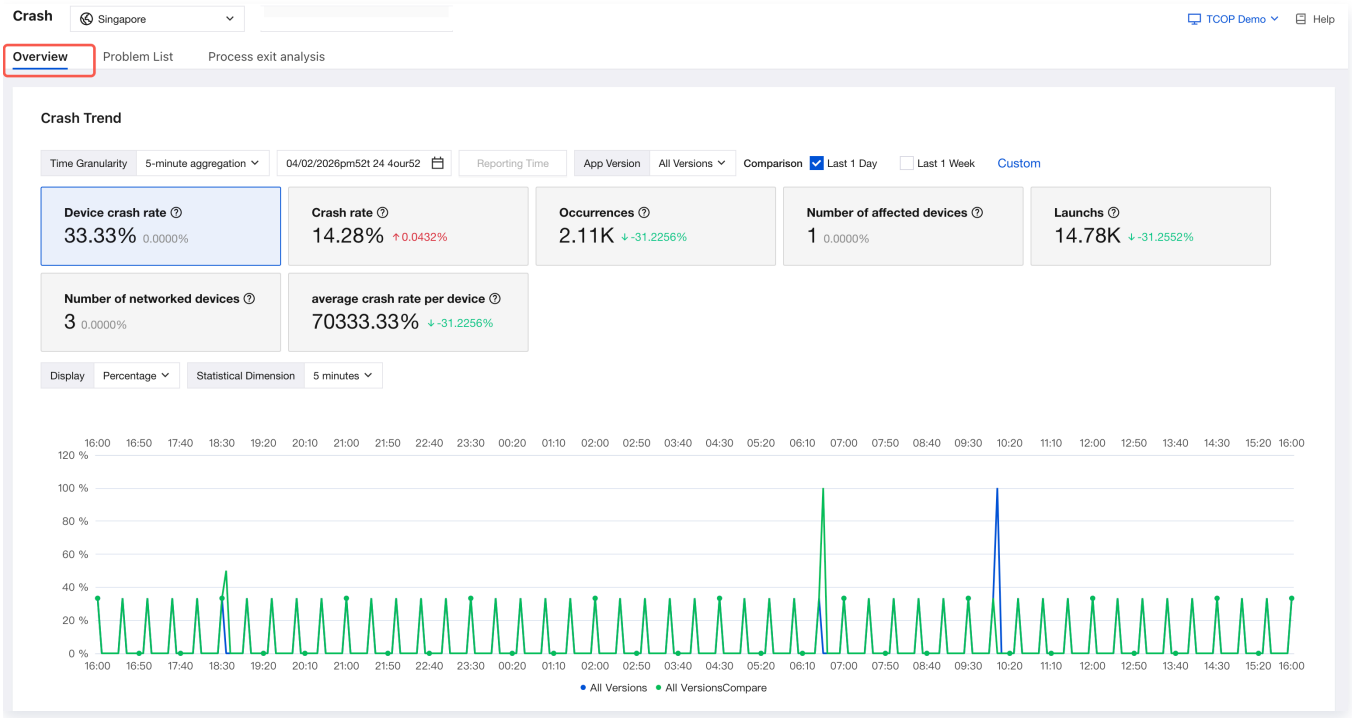
After the data is successfully reported, you can view the data in the RUM Pro console, as shown below:

1. The left sidebar contains the entry points for various monitoring features.
2. Each monitoring feature includes an overview, metric analysis, issue list, and issue details.



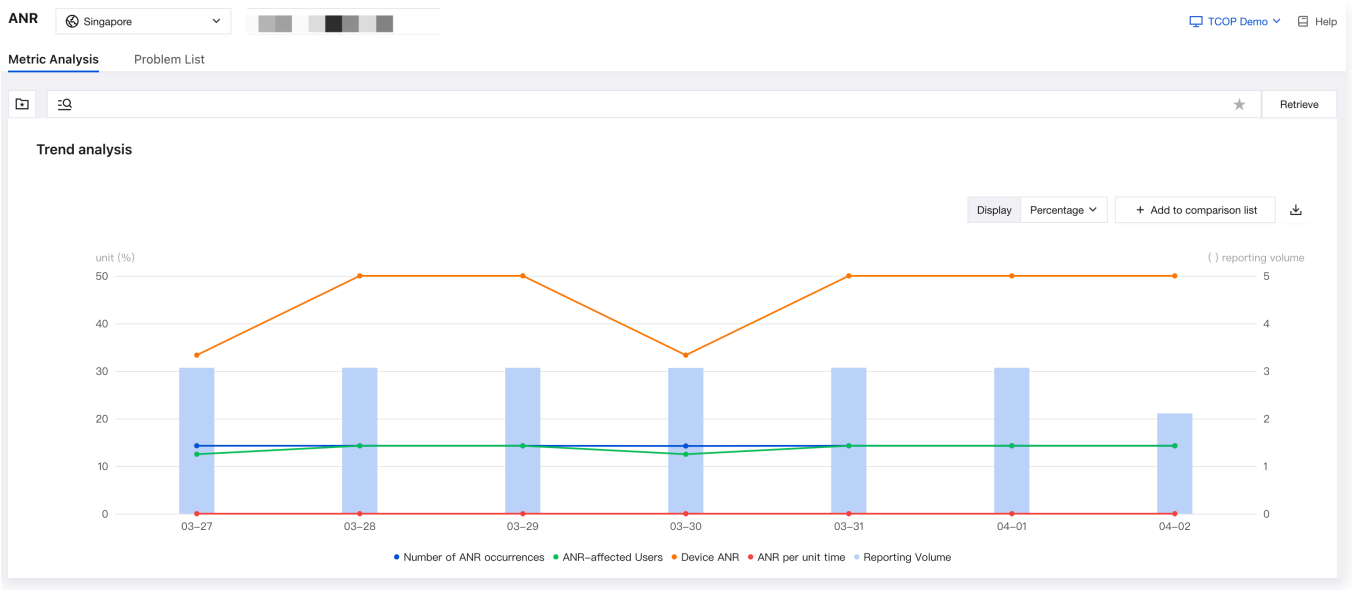
Exception Overview

The exception overview includes metric analysis data for quality monitoring such as crash, error, ANR, and OOM monitoring. It supports real-time data analysis, trend analysis, and exception distribution.



Metric Analysis

Metric analysis includes lag, memory, and startup analysis data. It supports trend analysis, comparative analysis, and multidimensional analysis for metrics.

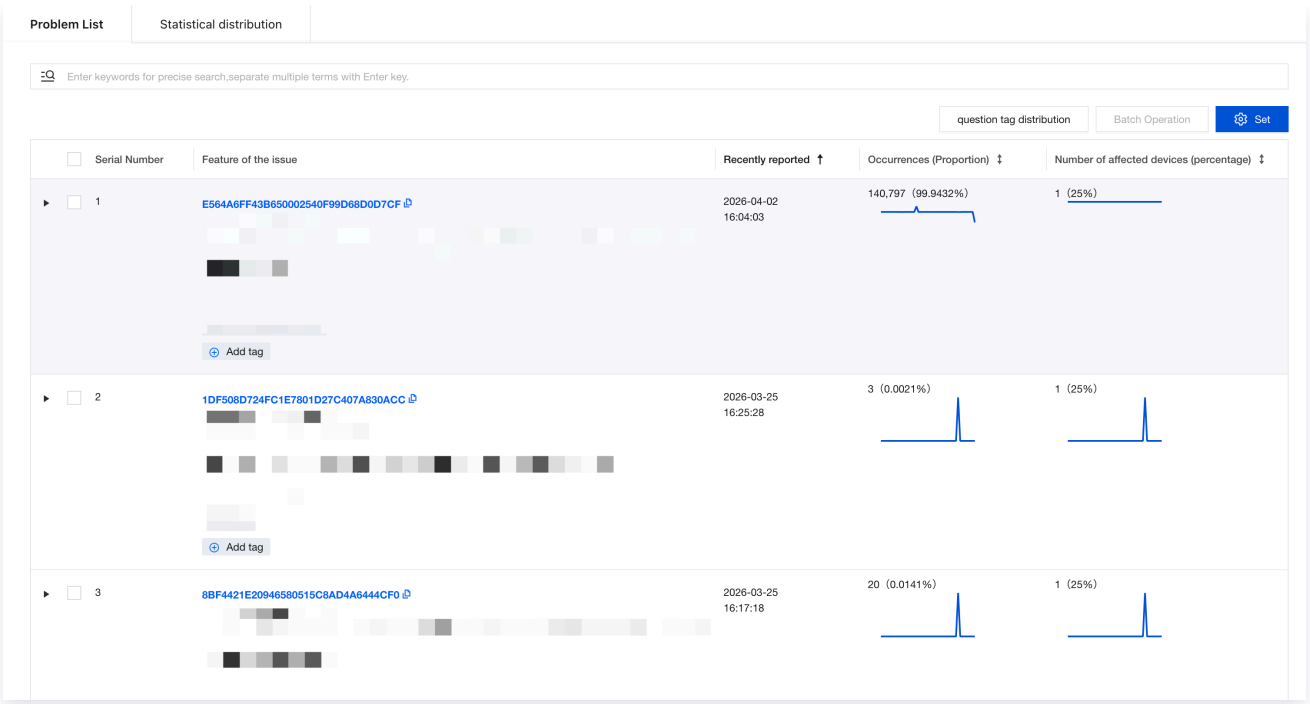


Issue List

Both quality monitoring (such as crash, error, ANR, and OOM monitoring) and performance monitoring (such as lag, memory, and startup monitoring) have issue lists. After an exception is identified, the SDK collects on-device information and reports it to the server. After the server receives the data, the processing pipeline extracts the characteristics of each reported case, and aggregates cases with identical characteristics into issues. The issue list displays these issues and provides rich search capabilities to facilitate user analysis and queries.

Common steps for using the issue list:

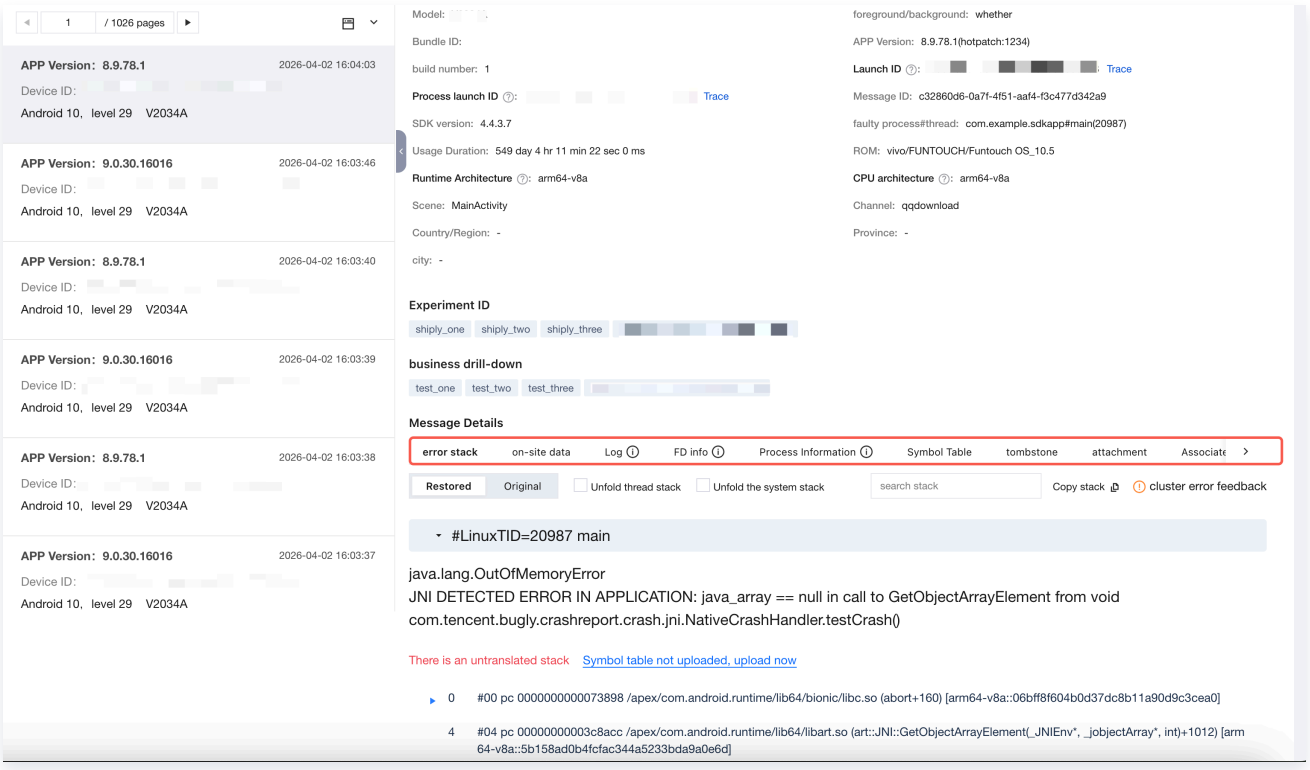
1. Edit the search conditions based on analysis needs. You can also choose to display only frequently used search conditions and hide less frequent ones.
2. After you edit the search conditions, click Query to submit the search task.
3. After the search is complete, the page automatically refreshes to update the search results.
4. Click to view the details of an issue.



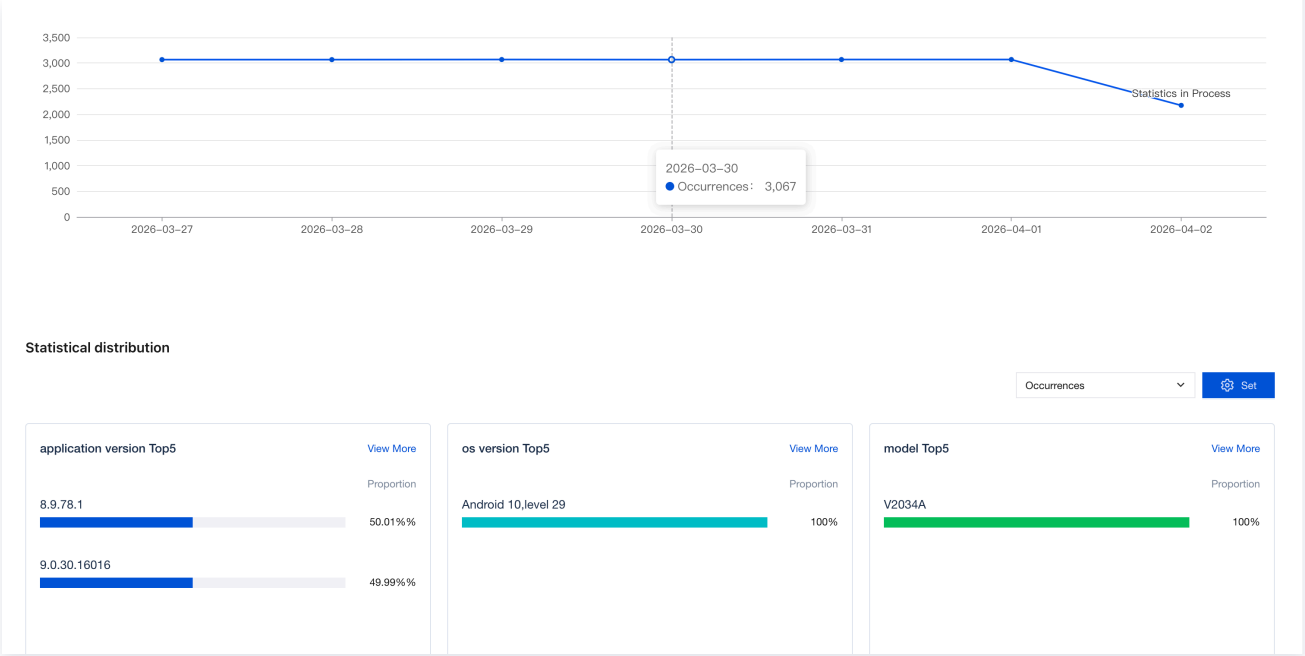
Issue Details

Issue details display the details of a specific issue. Typically, you use issue details to identify the root cause of the exception.

1. First, analyze the error stack. In the case details, select a specific case and analyze its error stack.
2. If the error stack is insufficient to define the issue, you may need to use logs, FD information, process infomation, or even information in attachment.



3. If you are still uncertain about why the issue occurs, you can use the drill-down analysis feature to analyze its reporting trends and distribution.



Android SDK Integration Guide
SDK Integration Overview

Last updated：2026-05-25 18:01:55

The Terminal Performance Monitoring Pro (RUM Pro) SDK is powered by the Tencent Bugly team. This article describes how to integrate the RUM Pro SDK in Android. After the SDK is integrated and data reporting is verified, you can use the analysis features in the console.

SDK Overview

- **SDK version**: 4.4.5.6.
- **SDK features**: A professional application quality monitoring tool that provides collection and analysis services for abnormal data, helping developers identify and resolve issues in a timely manner to develop high-quality apps.
- **Service provider**: Tencent Cloud Computing (Beijing) Co., Ltd.
- RUM Pro SDK Compliance Guide.
- RUM Pro SDK Personal Information Protection Rules.

Integration Steps

1. Before you integrate the SDK, please carefully read the RUM Pro SDK Compliance Guide.
2. Complete the SDK integration.
3. Complete the SDK initialization. During initialization, the SDK may collect certain user information. Please initialize the SDK only after the user has agreed to the RUM Pro SDK Personal Information Protection Rules. No information is collected before the SDK is initialized.
4. Verify data reporting.
5. For more information on other SDK features, see the API Description.

SDK Integration

Last updated：2026-05-25 18:01:55

The Terminal Performance Monitoring Pro (RUM Pro) SDK is powered by the Tencent Bugly team and uses the same SDK package as Bugly Pro. It supports both automatic and manual integration.

Prerequisites

RUM Pro requires Bugly SDK 4.4.5.6 or later.

Automatic Integration (Recommended)

1. Add the Maven repository address in the project-level build.gradle file.

```
buildscript {
    repositories {
        maven { url 'https://repo1.maven.org/maven2/' }
    }
}

allprojects {
    repositories {
        maven { url 'https://repo1.maven.org/maven2/' }
    }
}
```

2. Add dependencies and attribute configurations in the module's build.gradle file.

```
android {
    defaultConfig {
        ndk {
            abiFilters 'armeabi-v7a', 'arm64-v8a'
        }
    }
}

dependencies {
    implementation "com.tencent.bugly:bugly-pro:4.4.5.6"
}
```

Manual Integration (Not Recommended)

If you cannot use automatic integration due to special business requirements, security requirements, version control, or other reasons, you can also manually integrate the SDK by following the steps below.

1. Download the AAR file of the RUM Pro Android SDK.

   Download address for the RUM Pro Android SDK: Central Repository: com/tencent/bugly/bugly-pro. Select the appropriate version. The bugly-pro-4.4.5.6.aar SDK is used as an example for the following steps.
2. Add dependencies in the Android Studio project.

   Copy the AAR file to the libs directory of the module, and add dependencies in the Gradle file of the module.

```
dependencies {
    // Bugly SDK dependencies
    implementation 'org.jetbrains.kotlin:kotlin-android-extensions-
runtime:1.3.41'
    implementation 'org.jetbrains.kotlin:kotlin-stdlib-jdk7:1.3.41'
    implementation 'com.squareup.leakcanary:shark:2.7'
    implementation 'androidx.annotation:annotation:1.2.0'
    // Manual integration
    implementation files('libs/bugly-pro-4.4.5.6.aar')
}
```

SDK Initialization

Last updated：2026-05-25 18:01:55

This article describes how to initialize the SDK.

Example Code

Refer to the following code to initialize the SDK. We recommend that you initialize the SDK as early as possible to catch exceptions early on.

```
public static void initBugly(Context context) {
    // 1. Pre-build initialization parameters. These initialization
parameters are required.
    String appID = "a278f01047"; // [Required] Product APPID.
    String appKey = "1e5ab6b3-b6fa-4f9b-a3c2-743d31dffe86"; //
[Required] Product APPKEY.
    BuglyBuilder builder = new BuglyBuilder(appID, appKey);

    // 2. [Required] Configure the domain for reporting. Because
Terminal Performance Monitoring Pro (RUM Pro) and Bugly use the same
SDK, the domain for reporting is used to distinguish them.

buglyBuilder.setServerHostType(BuglyBuilder.ServerHostTypeBuglyCloud);

    // 3. Basic initialization parameters. We recommend that you
configure these initialization parameters.
    builder.uniqueId = "unique_id"; // [Recommended] Configure the
device ID, which must be unique. If this is not configured, RUM Pro will
generate a unique ID. This affects the statistics of device exception
rates and connected devices. It is recommended to store and reuse it via
SharedPreferences (SP).
    builder.userId = "user_id"; // [Recommended] Configure the user ID,
which affects the statistics of user exception rates. It is recommended
to store and reuse it via SP. Multiple settings within the same process
lifecycle are not supported.
    builder.deviceModel = Build.MODEL; // [Recommended] Configure the
device type. After the device model is configured, RUM Pro SDK no longer
reads the system's device model.
```

```
    builder.appVersion = "1.0.0"; // [Recommended] Configure the app
version number. If this is not configured, the value is read from
packageManager. It is recommended to configure this manually according
to the application's specifications. The value must be consistent with
the application version parameter used for uploading the symbol table.
    builder.buildNumber = "builderNum"; // [Recommended] Configure the
app build number. This is used for Java stack trace deobfuscation and
must be consistent with the build number parameter used for uploading
the symbol table.
    builder.appVersionType = BuglyAppVersionMode.Debug; // [Recommended]
Configure the version type.

    // 4. Additional initialization parameters. Configure these
initialization parameters as needed.
    builder.appChannel = "appChannel"; // Configure the app channel.
    builder.logLevel = BuglyLogLevel.LEVEL_DEBUG; // Configure the log
printing level. Levels can be obtained from Bugly LogLevel.
    builder.enableAllThreadStackCrash = true; // Configure whether to
capture all thread stacks on a crash. This is enabled by default.
    builder.enableAllThreadStackAnr = true; // Configure whether to
capture all thread stacks on an Application Not Responding (ANR) error.
This is enabled by default.
    builder.enableCrashProtect = true; // Configure whether to enable
crash protection mode during performance monitoring. This is enabled by
default.
    builder.debugMode = false; // Configure the debug mode, which can be
enabled during the debugging phase.
    builder.initAppState = BuglyBuilder.APP_STATE_FOREGROUND; //
Supported since version 4.4.3.7. This parameter is optional and can be
used to specify the foreground and background state of the application
during Bugly SDK initialization. If this is not specified, the SDK will
determine the application's foreground and background state during
initialization via getRunningAppProcesses. If this is specified, the SDK
will use the assigned state without calling getRunningAppProcesses.

    // 5. Configure the callback methods. Configure the initialization
parameters as needed.
    builder.setCrashHandleListener(crashHandleListener); // Configure
the crash handling callback. For details, see the callback API.
```

```
    builder.setUploadHandleListener(uploadhandleListener); // Configure
the crash reporting callback. For details, see the callback API.

    // 6. Initialization. This must be called.
    Bugly.init(context, builder);
}
```

Note:

- Context must be ApplicationContext.
- The device ID is crucial. RUM Pro uses the device ID to calculate the device exception rate. We strongly recommend that you specify the correct device ID to ensure device uniqueness.
- BuglyBuilder must be created before the init method, and repeated calls to the init method should be avoided.
- The Bugly.init API must be called for initialization. After initialization, you can call other APIs for customized settings. Otherwise, the settings will not take effect.
- The sampling of performance monitoring items can be modified in Application Configuration > SDK Configuration. You can modify the device sampling rate to enable or disable performance monitoring items.
- We recommend that you initialize the RUM Pro SDK after the user agrees to the RUM Pro SDK Personal Information Protection Rules.
- The domain for reporting must be configured via BuglyBuilder.setServerHostType during initialization.

Data Reporting Verification

Last updated：2026-05-25 18:01:55

Prerequisites

- Crash, ANR, and OOM data is 100% reported, is enabled by default, and does not support sampling. All other monitoring items are disabled by default and support sampling. Generally, performance metric monitoring is enabled by default, while abnormal performance monitoring is disabled by default. You can adjust these settings through SDK configuration. During integration, please make sure the enabled features meet your expectations.
- You need to create a configuration task on the Terminal Performance Monitoring Pro > Setting > SDK configuration page. For more information on the configuration feature, see SDK Configuration.
- During integration, it is recommended to set the sampling rate of all monitoring items to 1.0 (sample_ratio and event_sample_ratio) for easier data reporting verification. After the integration is complete, you can change the sampling rate as needed. Alternatively, you can create multiple configuration tasks based on the version type (corresponding to the initialized BuglyBuilder#appVersionType) to configure development tasks, grayscale release tasks, and production tasks. You can enable all monitoring items for development tasks, apply sampling as needed during the grayscale phase, and reduce sampling for the official release as required.
- Check whether you have an active resource package for the integrated product as described in the Purchase Instructions. If the product is not bound to an active resource package, data cannot be reported.

Crash Monitoring

Crash monitoring captures runtime exceptions and errors (such as Java exceptions or native crashes) and collects crash logs and stack traces to help developers locate and fix issues, enhancing application stability.

Simulating an Exception

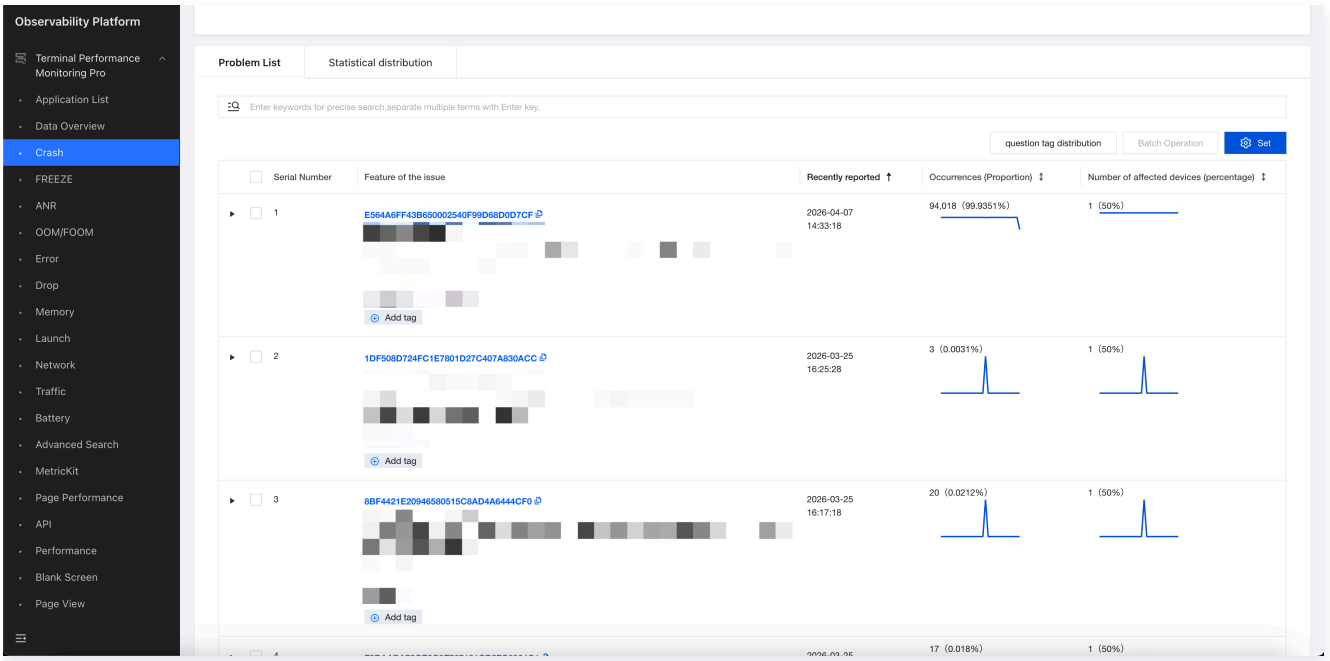
After initialization, you can simulate Java exceptions or native exceptions to verify the reporting feature of the SDK.
After a crash occurs, certain exceptions are reported after a restart. To ensure the current crash handling is completed, it is recommended to wait for more than 10 seconds after the crash to initiate the restart. Reporting may be delayed. During testing, it is recommended to manually trigger the simulation multiple times before you refresh the page.

```
Bugly.testCrash(Bugly.JAVA_CRASH); // Simulates a Java exception.
```

```
Bugly.testCrash(Bugly.NATIVE_CRASH); // Simulates a native exception.
```

Checking Data Reporting

After a crash is reported, you can view the reported issue in Crash > Problem List.



ANR Monitoring

Android ANR monitoring detects when the application's main thread becomes unresponsive for an extended period (such as failing to process input events within 5 seconds). It records and reports the relevant stack trace information to help developers locate and optimize lag issues.

Simulating an Exception

Similar to crashes, you can simulate an ANR either by using the test API provided by Terminal Performance Monitoring Pro (RUM Pro), or you can manually execute a time-consuming task on the UI thread.

- Simulate an ANR with the API.

```
Bugly.testCrash(Bugly.ANR_CRASH); // Simulates an ANR.
```

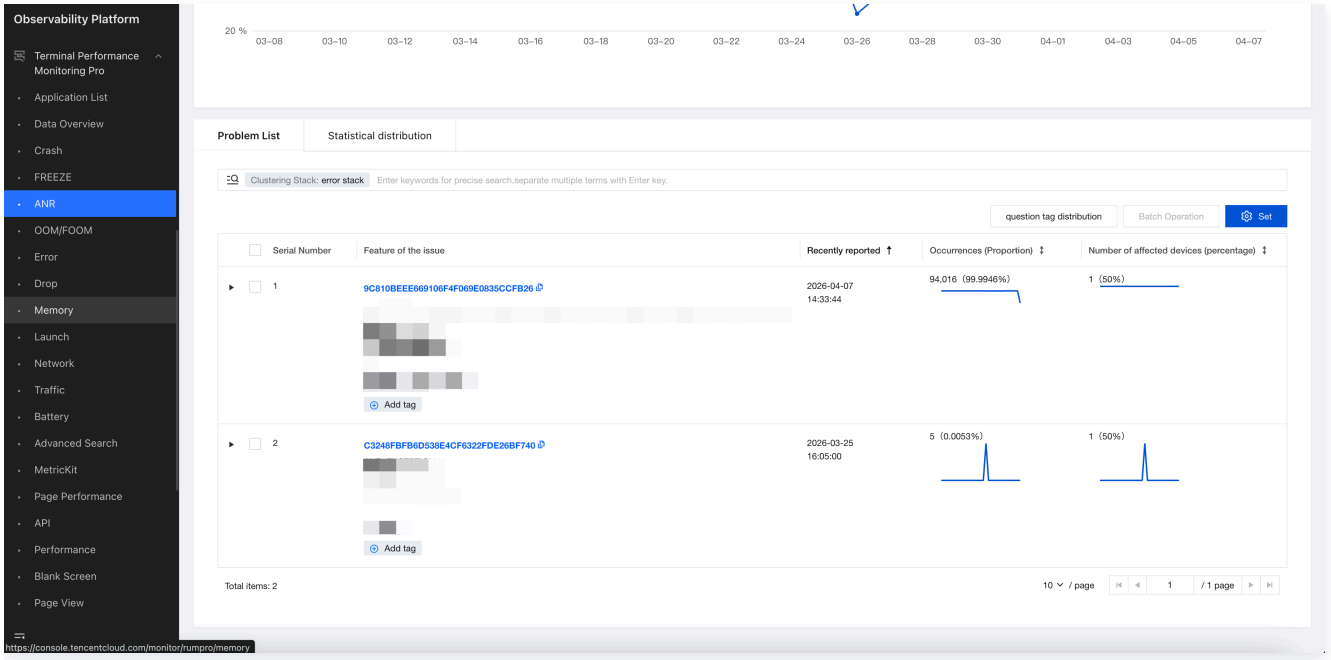
- Execute the following time-consuming logic on the UI thread to trigger an ANR.

```
private void testANR(){
    try {
        Thread.sleep(30000);
    } catch (Exception e){
        // do nothing
```

```
        }
    }
```

Checking Data Reporting

After an ANR is reported, you can view the reported issue in ANR > Problem List.



Error Monitoring

Errors generally refer to user-defined exceptions, such as caught Java exceptions, C# exceptions, JavaScript exceptions, and Lua exceptions. They are typically reported via handleCatchException or postException in RUM Pro (for details, see the Other APIs, Reporting a Custom Exception, or Reporting a Caught Java Exception section).

Checking the SDK Configuration

Before you verify the data reporting feature, please check the error sampling settings in Setting > SDK Configuration.

SDK Integration

The application can report custom errors through the handleCatchException and postException APIs provided by RUM Pro.

- To report a caught Java exception, see the following sample code.

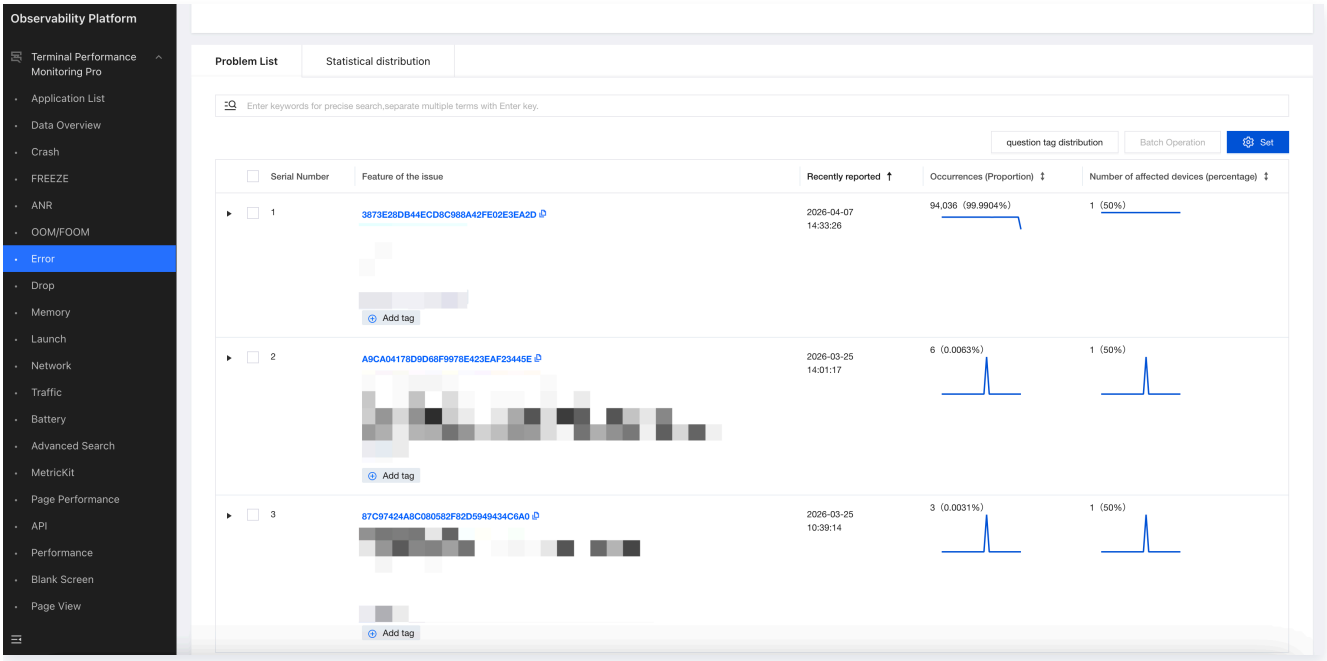
```
private void testJavaCatchError(){
      String content = "a illegal string.";
      try {
          JSONObject jsonObject = new JSONObject(content);
          double price  = jsonObject.getDouble("price");
      } catch (Throwable t) {
          Bugly.handleCatchException(Thread.currentThread(), t,
                              "in test code for json parse fail.",
content.getBytes(), true);
      }
}
```

- To report a custom exception, see the following sample code.

```
Bugly.postException(4, "testErrorType", "testErrorMsg", "testStack",
null);
```

Checking Data Reporting

After an error is reported, you can view the reported issue in Error > Problem List.

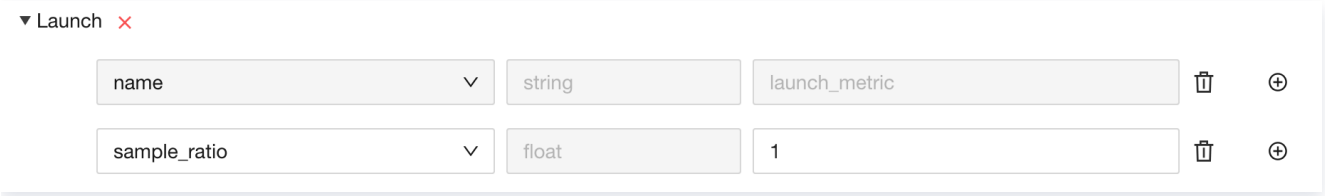


Startup Monitoring

When startup monitoring is enabled, the startup module automatically installs the monitoring feature and reports startup data after SDK initialization. You can customize the startup completion point via the Bugly.reportAppFullLaunch API. Startup monitoring also supports custom tags and tracking APIs to monitor time-consuming tasks during the startup process.

Checking the SDK Configuration

Before you verify the data reporting feature, please check the startup sampling settings in Setting > SDK Configuration.



SDK Integration

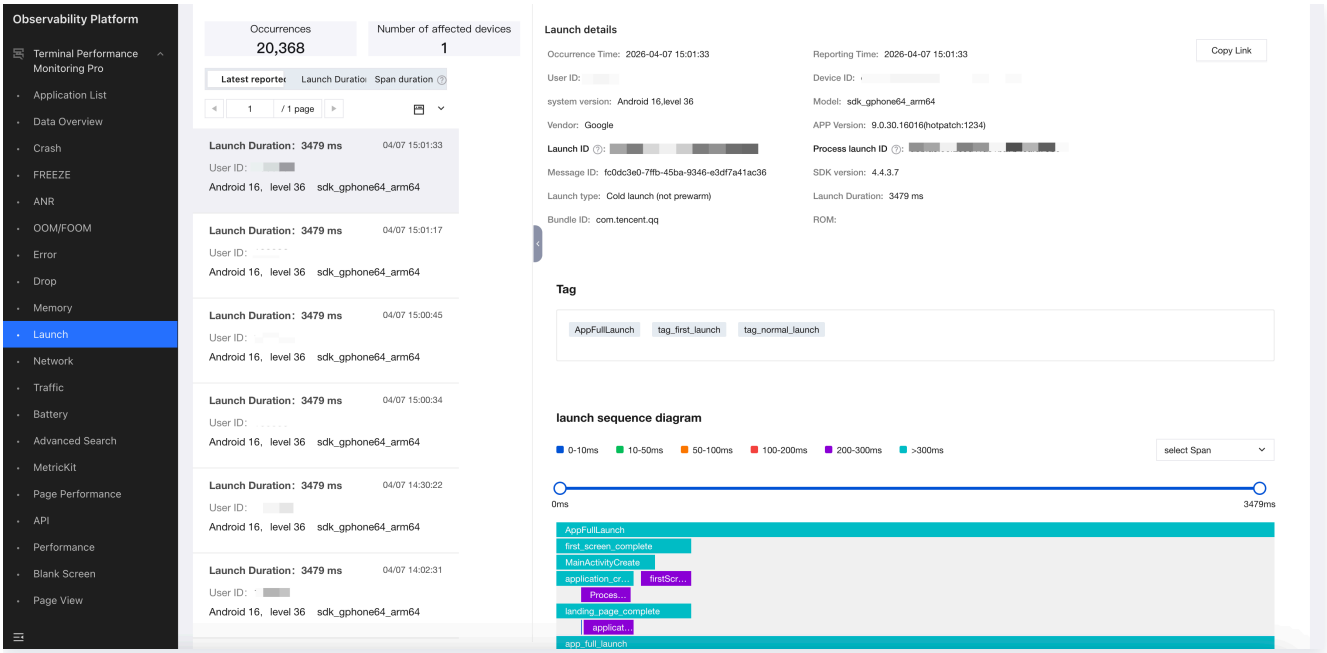
Please read the API Description – Startup Monitoring section to learn more about the startup APIs.

```
private void costJobOne() {
    AppLaunchProxy.getAppLaunch().addTag("has_cost_job_one");
    AppLaunchProxy.getAppLaunch().spanStart("costJobOne", null);
    try {
        Thread.sleep(300);
    } catch (Throwable t) {
        Bugly.handleCatchException(Thread.currentThread(), t,
"costJobOne sleep fail", null, true);
    }
    AppLaunchProxy.getAppLaunch().spanEnd("costJobOne");
}

private void finalJob() {
    AppLaunchProxy.getAppLaunch().reportAppFullLaunch();
}
```

Checking Data Reporting

After the data is reported, you can view the report details in Launch > Launch instance.



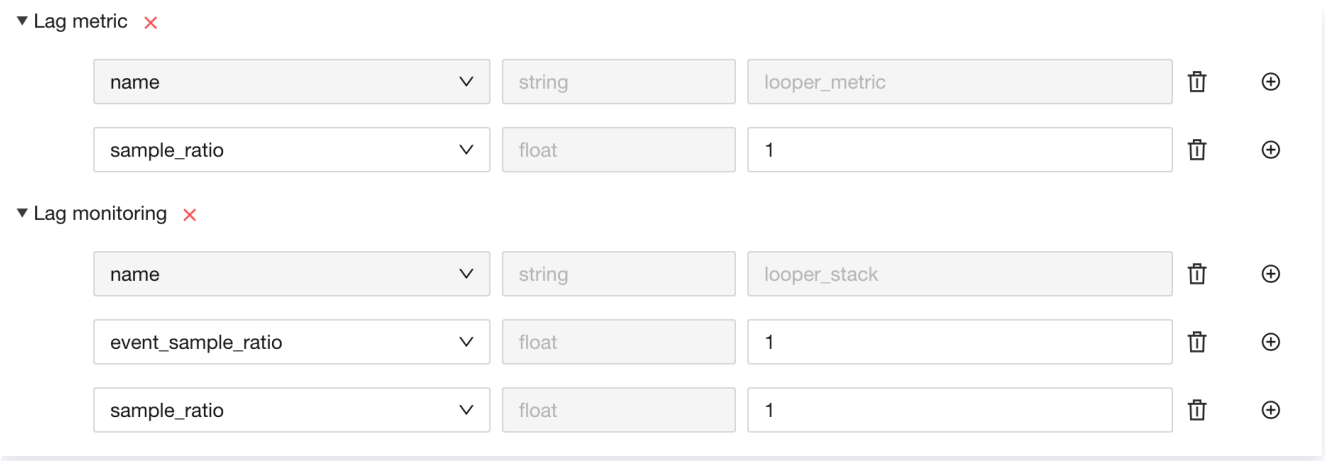
Lag Monitoring

Application lag is a major cause of user churn. The lag monitoring module is designed to help you address lag issues. A key approach to lag mitigation involves using effective metrics to assess the current state of the application and capturing sufficient information to provide optimization guidance. The module evaluates application smoothness through two metrics: frames per second (FPS) and suspension rate. By monitoring lag issues and directly capturing lag stacks, it helps pinpoint the root causes of lag issues and provides clear optimization guidance.

- Lag metric uses stable and lightweight collection techniques to collect smoothness data throughout the entire application runtime.
- Lag issue monitoring utilizes high-frequency stack trace capturing to provide rich contextual information for accurate cause analysis.
- The self-developed fast stack trace capturing technology for the Android platform achieves a 6 times performance improvement compared to traditional stack trace capturing methods.
- Both lag metrics and lag issues support custom sampling rates and provide robust data analysis capabilities.

Checking the SDK Configuration

Before you verify the data reporting feature, please check the lag configuration in Setting > SDK Configuration.



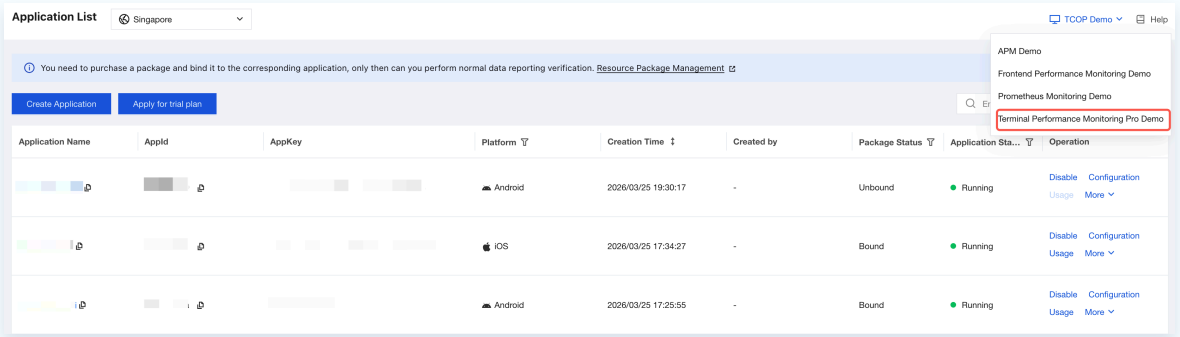
- **Lag**

  When the lag metric ("looper_metric") is enabled, the SDK monitors application smoothness after initialization. During a single application run, data is first collected and stored locally, then aggregated and reported during the next startup. To avoid impacting application startup performance, the SDK aggregates and reports the cached data to the backend 5 minutes after initialization.

  - FPS: frame rate. It refers to the number of images the GPU and CPU can produce during application runtime, measured in frames per second (FPS). It is typically used as a metric to evaluate hardware performance and application smoothness.
  - Suspension rate: If the refresh delay between two application frames exceeds 200 ms, it is considered that the application has poor responsiveness to user interactions, and this delay is added to the suspension time. The suspension rate of a device is its total suspension time divided by the total foreground duration in a day, measured in seconds per hour.

  Note:

  You can test list scrolling lag in the demo to observe the FPS and suspension rate. To use the demo, choose **Terminal Performance Monitoring Pro Demo** in the upper-right corner of the Terminal Performance Monitoring Pro > Application List page.



- **Lag**

  After lag monitoring ("looper_stack") is enabled, if time-consuming logic executed on the UI thread exceeds 500 ms, lag reporting is triggered. Lag monitoring detects UI thread lag by monitoring message

execution on the UI thread.
During the verification phase, it is recommended to set both the "event_sample_ratio" (message sampling rate) and "sample_ratio" (device sampling rate) for lag monitoring to 1. This ensures that lag reporting is triggered when the drolagp threshold is met.

Simulating an Exception

To simulate lag reporting, see the following sample code.

```
    private void testLongLag() {
        showToast("Long lag testing");
        sleep(200);
        callFunctionA();
        callFunctionB();
    }

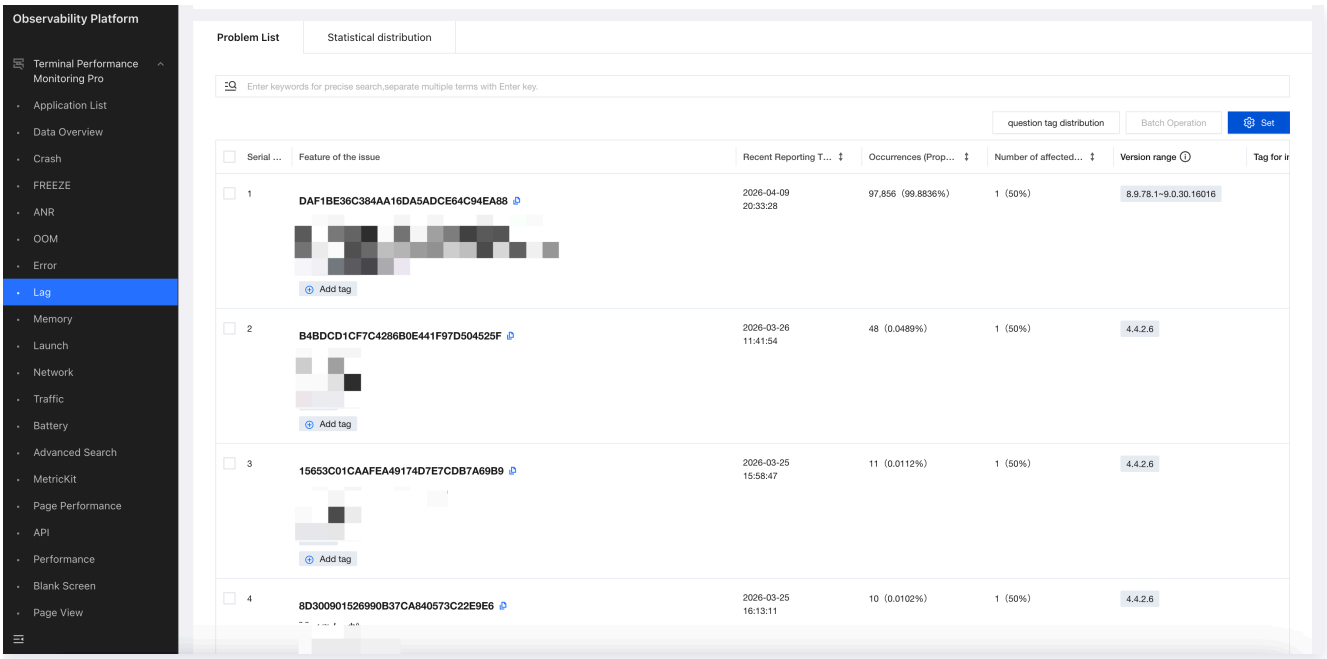
    private void callFunctionA() {
        sleep(400);
        callFunctionB();
    }

    private void callFunctionB() {
        sleep(300);
    }

    private static void sleep(long timeInMs) {
        try {
            Thread.sleep(timeInMs);
        } catch (Throwable t) {
            Bugly.handleCatchException(Thread.currentThread(), t,
"sleep", null, false);
        }
    }
```

Checking Data Reporting

After a lag is reported, you can view the reported issue in Lag > Problem List.



Memory Monitoring

Memory monitoring supports the memory peak and memory details features, with more memory monitoring features under development.

Memory Peak

Memory peak refers to the highest memory usage of the application during its entire runtime. RUM Pro queries the application's memory usage through scheduled tasks, records the maximum usage throughout the runtime, and reports it after the next application startup.

- Physical memory peak: peak physical memory used by the application during a single run, also known as peak physical set size (PSS).
- Virtual memory peak: peak virtual memory used by the application during a single run, also known as peak virtual set size (VSS).
- Java heap memory peak: peak Java heap memory used by the application during a single run.

Memory Details

During SDK initialization, you need to configure the monitoring item BuglyMonitorName.MEMORY_JAVA_CEILING in BuglyBuilder.

```
public static void initBugly(Context context) {
    // 1. Pre-build initialization parameters. Initialization parameters
are required.
    String appID = "xxxxxxx"; // [Required] appID of the product created
in RUM Pro.
```

```
    String appKey = "xxxxxxxxxxx"; // [Required] appKey of the product
created in RUM Pro.
    BuglyBuilder builder = new BuglyBuilder(appID, appKey);

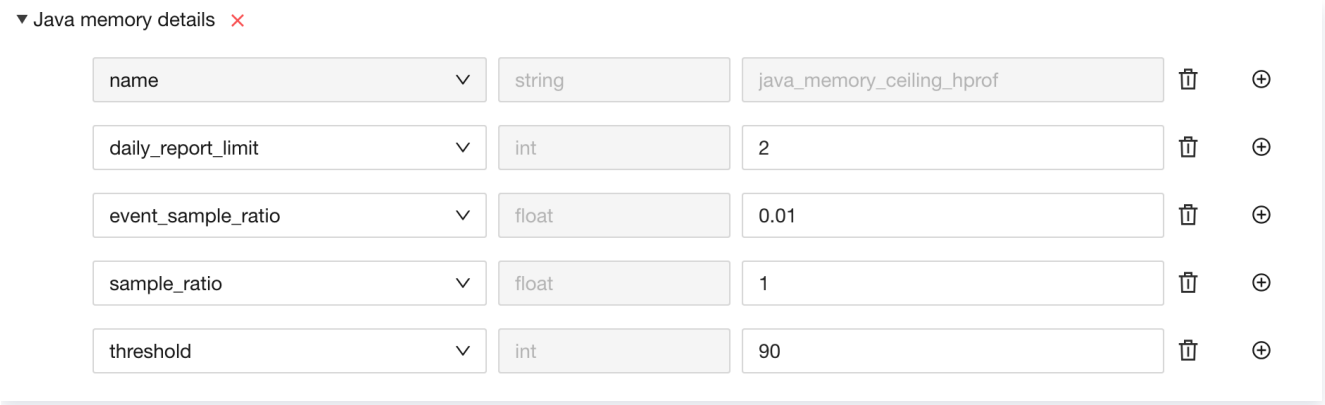
    ......

    // 2. Enable Java memory details.
    build.addMonitor(BuglyMonitorName.MEMORY_JAVA_CEILING);

    // 3. Initialization. This must be called.
    Bugly.init(context, builder);
}
```

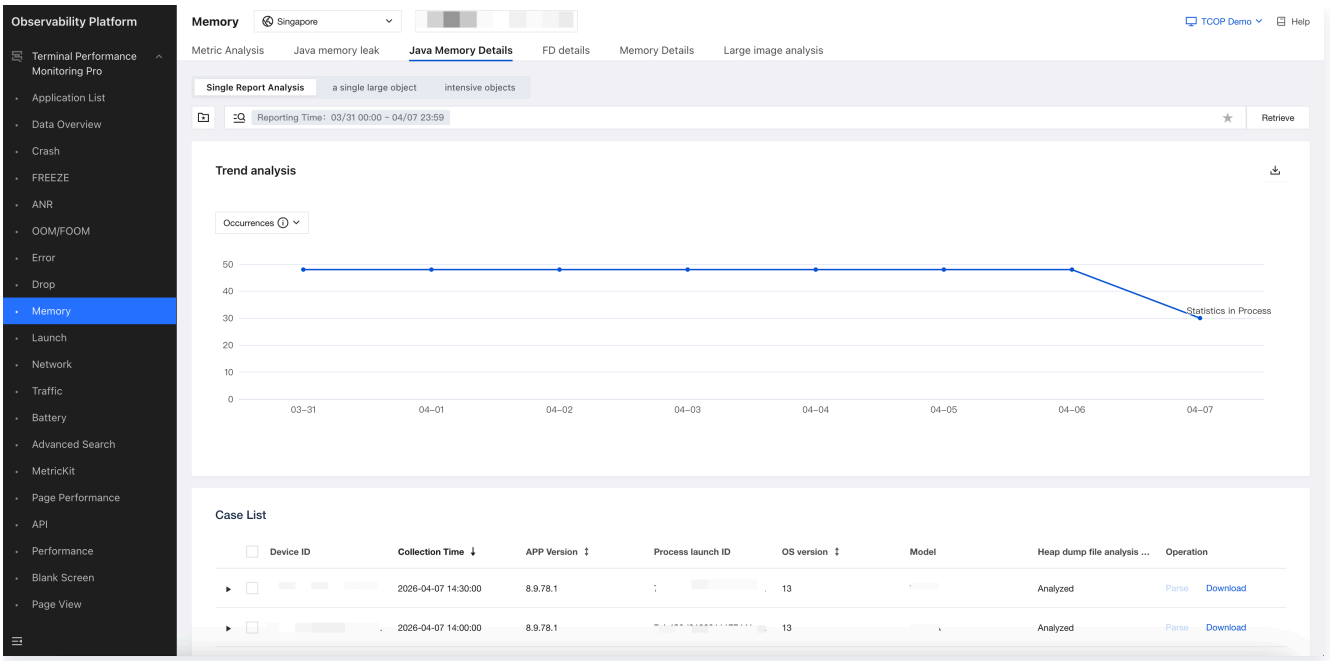
In Setting > SDK Configuration, modify the SDK configuration task to adjust the configuration properties for **Java memory details**.

- sample_ratio: controls the user sampling rate, which is the percentage of devices to enable this feature.
- event_sample_ratio: controls the event sampling rate, which determines whether an issue is reported when the memory limit is reached.
- threshold: sets the trigger for dumping heap dump files. 90 means dumping starts when 90% of the maximum value is reached. The maximum heap memory corresponds to Runtime.getRuntime().maxMemory().



Checking Data Reporting

After reporting is triggered, you can view the reported issue in Memory > Java Memory Details.



Network Monitoring

Network monitoring uses the EventListener of OkHttp3 to monitor the quality of HTTP requests, including request duration, success rate, data transfer volume, and key phase duration of the request. The following capabilities are implemented through an instrumentation-free approach:

1. You can follow the SDK integration guide and use the OkHttpClient created by using BuglyListenerFactory to monitor the quality of HTTP and HTTPS requests.
2. BuglyURLStreamHandlerFactory is provided as a proxy for the system's native HttpURLConnection to monitor HTTP and HTTPS requests implemented using the system's native APIs.

> **Note:**
> The following scenarios are not supported:
> - HTTP requests not implemented using native APIs or OkHttp3.
> - The component uses the OkHttp3 library to implement HTTP requests but the OkHttpClient process is not modified, meaning that BuglyListenerFactory is not used as the EventListener.Factory for OkHttpClient.

Checking the SDK Configuration

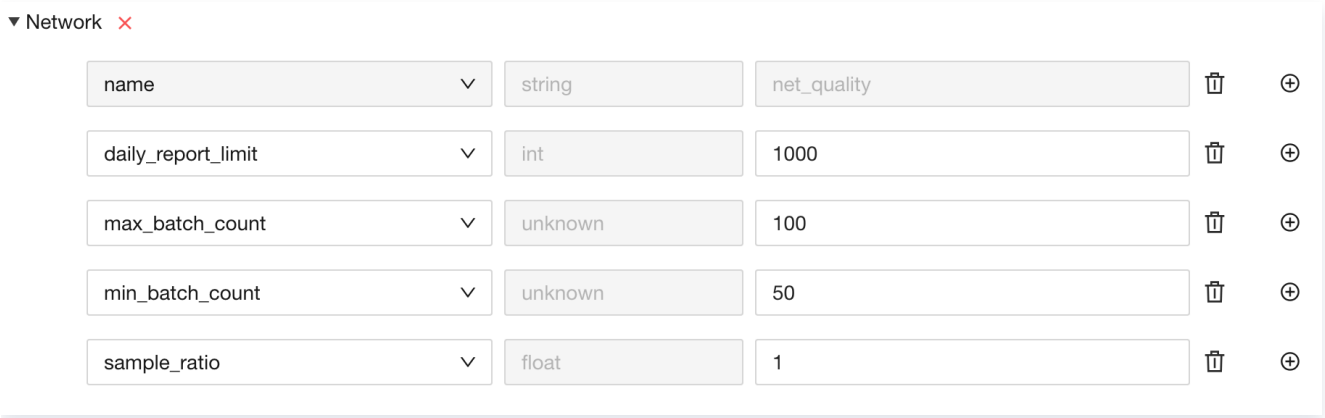
Before you verify the data reporting feature, please check the network monitoring configuration in Setting > SDK Configuration.

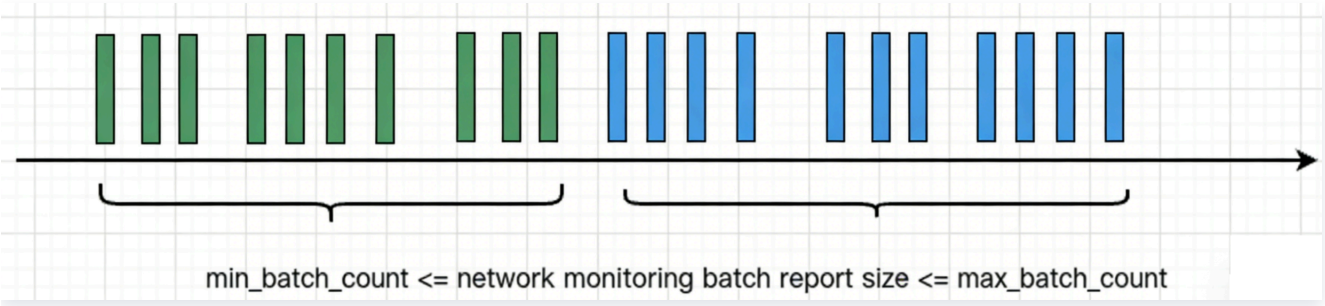
Network monitoring includes the following configuration properties:

- sample_ratio: device sampling rate, which is the percentage of devices to enable network monitoring.
- daily_report_limit: specifies the maximum number of times network monitoring data can be reported per day. Network monitoring data is reported in batches. The specific batching logic is determined by

max_batch_count and min_batch_count.

- max_batch_count: specifies the maximum number of HTTP request quality data entries contained in a single network monitoring data report. The default value is 100.
- min_batch_count: specifies the minimum number of HTTP request quality data entries contained in a single network monitoring data report. The default value is 50.





Note:

- During the testing phase, it is recommended to set sample_ratio to 1. After testing, you can change the device sampling rate based on actual needs.
- During the testing phase, it is recommended to set min_batch_count to 5 or 10 for faster data reporting to the backend. After testing, you can change the value based on your business needs or restore it to the default value.

SDK Integration

1. Enable proxy for the system's native HttpURLConnection. If you do not want to enable proxy for HTTP requests **using the system's native APIs**, you can skip this step.

   If you want to enable proxy for HTTP requests that use the system's native APIs, it is recommended to enable proxy as early as possible in Application#onCreate.

```
// Enables the proxy.
BuglyURLStreamHandlerFactory.init(okHttpClient);
```

After the proxy is enabled, you can use the following code to disable it.

```
// Disables the proxy.
BuglyURLStreamHandlerFactory.reset();
```

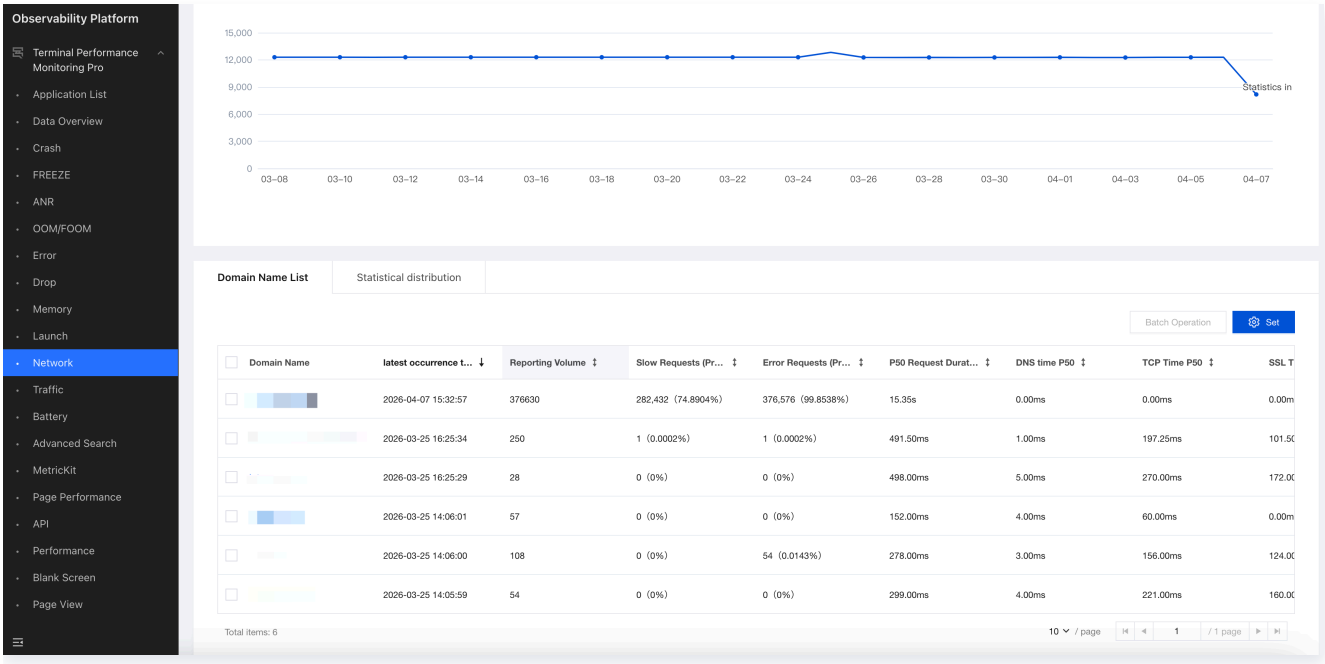
2. Use BuglyListenerFactory to create an OkHttpClient.

```
OkHttpClient client = new OkHttpClient.Builder()

.eventListenerFactory(BuglyListenerFactory.getInstance())
                .build();
```

Checking Data Reporting

After integration, you can view the reported data in Network > HTTP or Network > Network Error.

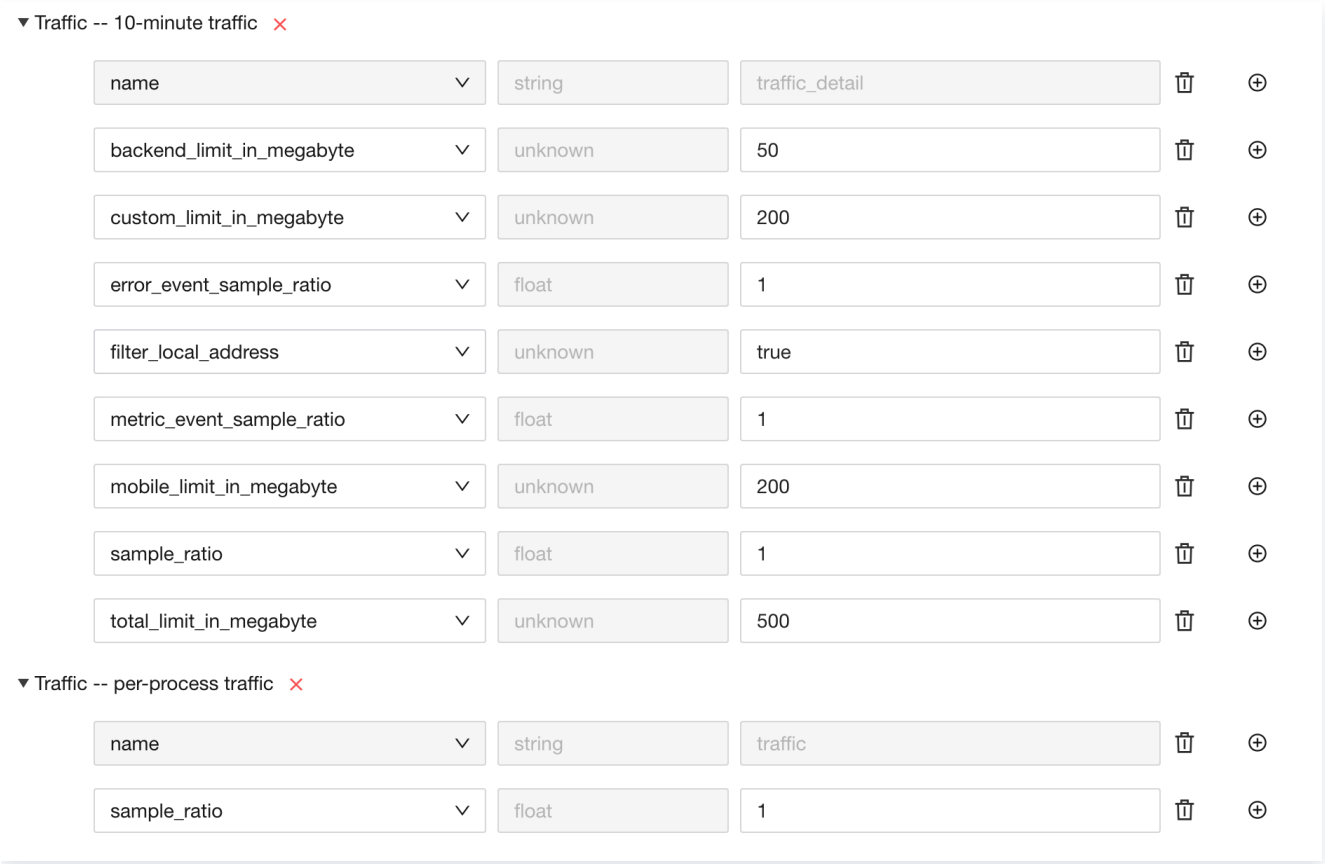


Traffic Monitoring

The core feature of traffic monitoring is its ability to track application data usage in real time. Through real-time monitoring, you can stay informed about data consumption patterns and promptly identify issues such as unexpected traffic usage. This helps you avoid exceeding data plans or incurring additional charges. Additionally, traffic monitoring assists in optimizing application power consumption and reducing unnecessary data requests to deliver a more efficient user experience and extend the battery life. This feature also enables drilling down to the domain level, allowing you to identify which domains or services consume a large amount of traffic, facilitating targeted measures to optimize data transmission.

Checking the SDK Configuration

Before you verify the data reporting feature, please check the traffic monitoring configuration in Setting > SDK Configuration. Traffic monitoring includes two configuration items: **Traffic--10-minute traffic** and **Traffic--per-process traffic**.



SDK Integration

To enable the traffic monitoring feature, you need to execute the following statements during SDK initialization, and enable traffic monitoring in the backend.

```
...
buglyBuilder.addMonitor(BuglyMonitorName.TRAFFIC);
buglyBuilder.addMonitor(BuglyMonitorName.TRAFFIC_DETAIL);
...
// Initializes the Bugly SDK.
Bugly.init(application, buglyBuilder);
```

The SDK also provides APIs for reporting custom scenario traffic.

```
CustomTrafficStatistic.getInstance().addHttpToQueue(SocketInfo
```

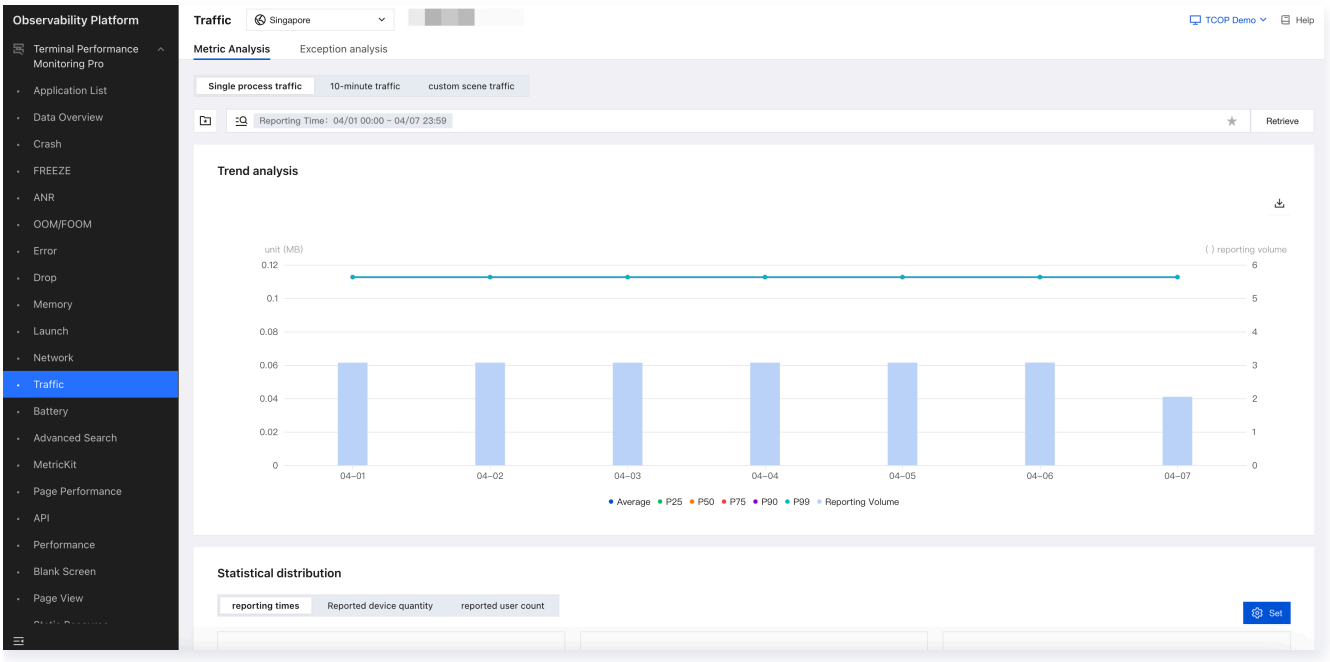
```
socketInfo);
```

SocketInfo contains many other fields. To collect traffic statistics, you need to assign values only to the following fields. Other member variables can be ignored.

```
class SocketInfo {
    receivedBytes;    // Bytes received.
    sendBytes;        // Bytes sent.
    networkType;      // Network type. Set to 1 for Wi-Fi traffic, 2 for
5G traffic, or 3 for no network.
    frontState;       // Foreground and background state. Set to 1 for
foreground, or 2 for background.
    host;             // Domain name, such as www.baidu.com.
    startTimeStamp;   // Timestamp when the network request starts, in
milliseconds.
    ......
}
```

Checking Data Reporting

After the data is reported, you can view the data in Traffic > Metric Analysis or Traffic > Exception analysis.



Page Performance

User experience (UX) is a critical factor in mobile application development. With the advancement in smartphone hardware performance and the continuous increase in user demands, users have higher

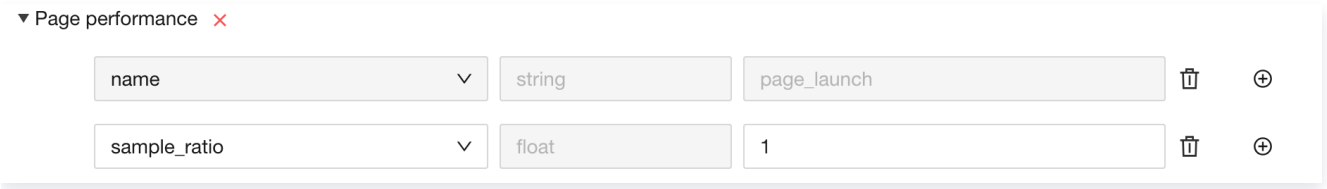
expectations for application responsiveness and smoothness. Page startup time is a key metric for user experience, especially for mobile applications. An extended page startup time can frustrate users and may even result in user churn.

RUM Pro monitors the cold start time of Activities by default. The start time is divided into page rendering time and page loading time.

- Page rendering time: This is the duration from the call of Activity's onCreate to the completion of the first frame rendering.
- Page loading time: This is the same as the page rendering time by default. You can customize the loading end time of the Activity through the reportActivityFullLaunch API.

Checking the SDK Configuration

Before you verify the data reporting feature, please check the configuration of **Page performance** in Setting > SDK Configuration.
sample_ratio: controls the user sampling rate, which is the percentage of devices to enable this feature. 1 indicates the feature is enabled for all devices. 0 indicates the feature is disabled for all devices.



SDK Integration

Before you call the Bugly.init API, add the following code in the client.

```
...
buglyBuilder.addMonitor(BuglyMonitorName.PAGE_LAUNCH); // Add this
statement.
...
Bugly.init(context, builder);
```

In addition to the overall page startup time, you can also call the following APIs to monitor the time consumption of each span during the page startup process. A span is a sub-stage of the page startup process. The following are the APIs of the com.tencent.rmonitor.pagelaunch.PageLaunchMonitor class, which is a singleton class. You can obtain the singleton object via PageLaunchMonitor.getInstance().

```
/**
  * Marks the start of a span during Activity startup.
  * @param activity
```

```
   * @param name Span name.
   * @param parentName Name of the parent span.
   */
public void startSpan(Activity activity, String name, String
parentName);


/**
 * Marks the end of a span during Activity startup.
 * @param activity
 * @param name Span name, which must match the parameter of startSpan.
 */
public void endSpan(Activity activity, String name);


/**
 * The time when this API is called is the custom end time of the
Activity startup.
 * @param activity
 */
public void reportActivityFullLaunch(Activity activity);
```

> **Note:**
> The startSpan and endSpan methods must be called in pairs. Only one record is kept for spans with the same name. A later span with the same name will overwrite the previous one.

See the following sample code:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    PageLaunchMonitor.getInstance().startSpan(this, "MemoryTracerTest",
"");
    PageLaunchMonitor.getInstance().startSpan(this, "onCreate",
"MemoryTracerTest");
    startMemoryTrace();
    PageLaunchMonitor.getInstance().endSpan(this, "onCreate");
}
```

```
@Override
public void onResume() {
    super.onResume();

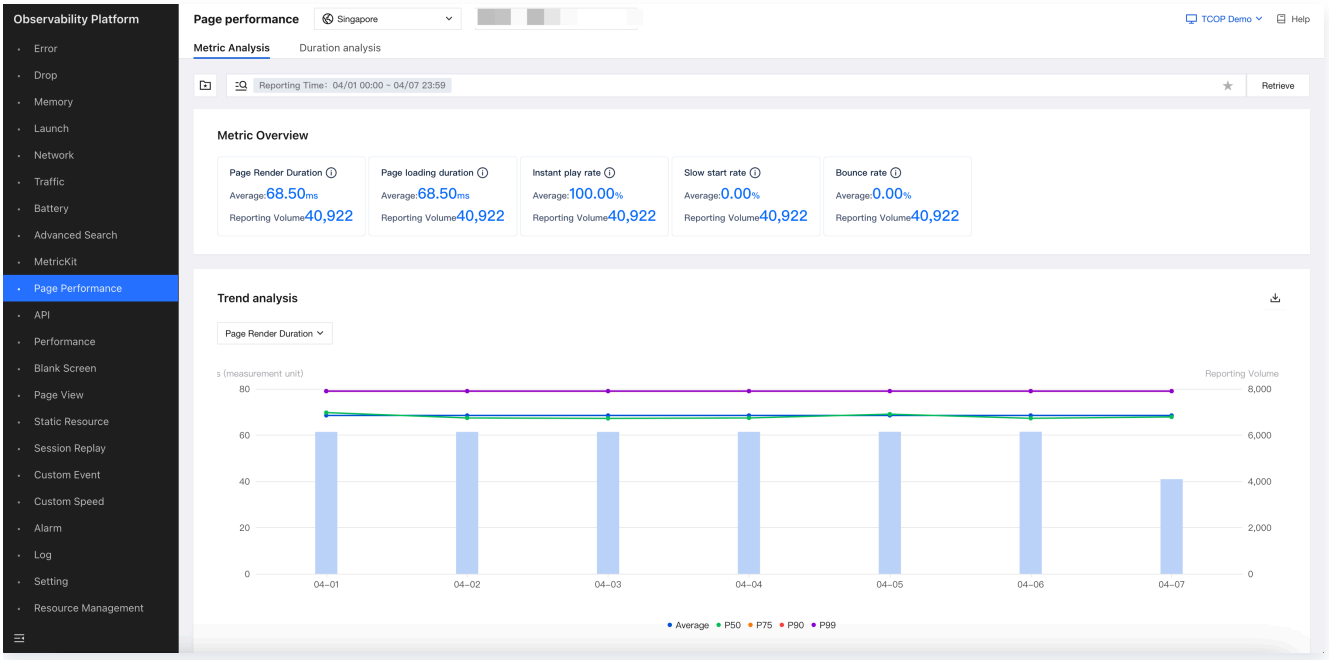
    PageLaunchMonitor.getInstance().startSpan(this, "onResume",
"MemoryTracerTest");
    reStartMemoryTrace();
    PageLaunchMonitor.getInstance().endSpan(this, "onResume");
    PageLaunchMonitor.getInstance().endSpan(this, "MemoryTracerTest");

    // After the page has finished loading, call reportLaunchFinished to
define the end of the Activity startup.
    new Thread (new Runnable() {
        reportLaunchFinished();
    }).start();
}

private void reportLaunchFinished() {
    PageLaunchMonitor.getInstance().reportActivityFullLaunch(this);
}
```

Checking Data Reporting

After the data is reported, you can view the reported data in Page Performance > Metric Analysis.



API Description

Last updated：2026-05-25 18:01:55

This article describes the feature APIs of the Terminal Performance Monitoring Pro (RUM Pro) SDK to facilitate flexible and in-depth integration.

Callback APIs

Configuring the Crash Handling Callback

RUM Pro can trigger a callback to your business logic when handling a crash. You can add custom logic in the callback or provide additional information to be reported along with the exception.

```
public interface CrashHandleListener {

    /**
     * During the crash handling callback, the data returned by this API
is reported as an attachment named userExtraByteData.
     * @param isNativeCrashed Indicates whether the crash is a native
exception.
     * @param crashType Exception type.
     * @param crashStack Exception stack.
     * @param nativeSiCode SI_CODE for the native exception. This data
is invalid for non-native exceptions.
     * @param crashTime Time when the native exception occurs.
     * @return Byte stream of the reported attachment.
     */
    byte[] getCrashExtraData(boolean isNativeCrashed, String crashType,
String crashAddress,
            String crashStack, int nativeSiCode, long crashTime);

    /**
     * During the crash handling callback, the data returned by this API
is displayed in the attachment extraMessage.txt.
     * @param isNativeCrashed Indicates whether the crash is a native
exception.
     * @param crashType Exception type.
     * @param crashStack Exception stack.
     * @param nativeSiCode SI_CODE for the native exception. This data
is invalid for non-native exceptions.
```

```
     * @param crashTime Time when the native exception occurs.
     * @return Reported data.
     */
    String getCrashExtraMessage(boolean isNativeCrashed, String
crashType, String crashAddress,
            String crashStack, int nativeSiCode, long crashTime);

    /**
     * This API is executed before the crash handling callback.
     * @param isNativeCrashed Indicates whether the crash is a native
exception.
     */
    void onCrashHandleStart(boolean isNativeCrashed);

    /**
     * This API is executed after the crash handling callback.
     * @param isNativeCrashed Indicates whether the crash is a native
exception.
     * @return The return value has no practical meaning. It does not
affect the normal use of this method and can be ignored.
     */
    boolean onCrashHandleEnd(boolean isNativeCrashed);

    /**
     * This API is executed during the crash handling callback.
     * @param isNativeCrashed Indicates whether the crash is a native
crash.
     * @param crashType Crash type.
     * @param crashMsg Crash message. For example, "Attempt to invoke
virtual method 'int java.lang.String.length()' on a null object
reference" (added in version 4.4.1.2).
     * @param crashAddress Crash address.
     * @param crashStack Crash stack.
     * @param nativeSiCode SI_CODE. This is valid for native exceptions.
     * @param crashTime Crash time.
     * @param userId User ID at the time of the crash.
     * @param deviceId Device ID at the time of the crash.
     * @param crashUuid Unique identifier for this exception.
     * @return The return value has no practical meaning. It does not
affect the normal use of this method and can be ignored.
```

```
     */
    boolean onCrashSaving(boolean isNativeCrashed, String crashType,
String crashMsg, String crashAddress,
        String crashStack, int nativeSiCode, long crashTime, String
userId, String deviceId,
        String crashUuid, String processName);
}
```

During the initialization phase, configure the callback via BuglyBuilder as shown below:

```
CrashHandleListener crashHandleListener = xxx;
buglyBuilder.setCrashHandleListener(crashHandleListener);
```

Configuring the Crash Reporting Callback

RUM Pro can trigger a callback to your business logic when reporting a crash. You can add custom logic in the callback.

```
public interface UploadHandleListener {

        /**
         * Callback triggered when the reporting starts.
         * int TYPE_JAVA_CRASH = 0;  // Java crash.
         * int TYPE_NATIVE_CRASH = 1; // Native crash.
         * int TYPE_JAVA_CATCHED = 2; // Java error reporting.
         * int TYPE_ANR_CRASH = 3; // Application Not Responding (ANR).
         * int TYPE_U3D_CRASH = 4; // U3D error reporting.
         * @param requestKey Reporting keyword (starting from version
4.4.3.4, this field indicates the type of the reported exception case;
valid values are shown above).
         */
        public void onUploadStart(int requestKey);

        /**
         * Callback triggered after the reporting ends.
         * int TYPE_JAVA_CRASH = 0;  // Java crash.
         * int TYPE_NATIVE_CRASH = 1; // Native crash.
         * int TYPE_JAVA_CATCHED = 2; // Java error reporting.
         * int TYPE_ANR_CRASH = 3; // ANR.
```

```
         * int TYPE_U3D_CRASH = 4; // U3D error reporting.
         * @param requestKey Reporting keyword (starting from version
4.4.3.4, this field indicates the type of the reported exception case;
valid values are shown above).
         * @param responseKey Keyword of the response packet.
         * @param sended Byte stream sent.
         * @param recevied Byte stream received.
         * @param result True indicates successful reporting. False
indicates failure.
         * @param exMsg Additional information.
         */
        public void onUploadEnd(int requestKey, int responseKey, long
sended, long recevied, boolean result, String exMsg);
}
```

During the initialization phase, configure the callback via BuglyBuilder as shown below:

```
UploadHandleListener uploadHandleListener = null;
buglyBuilder.setUploadHandleListener(uploadHandleListener);
```

Note:

Starting from version 4.4.3.4, the requestKey parameter of UploadHandleListener is used to describe the type of the reported exception case. Its valid values are as follows:

- int TYPE_JAVA_CRASH = 0; // Java crash
- int TYPE_NATIVE_CRASH = 1; // Native crash
- int TYPE_JAVA_CATCHED = 2; // Java error reporting
- int TYPE_ANR_CRASH = 3; // ANR
- int TYPE_U3D_CRASH = 4; // U3D error reporting

Other APIs

You need to call other APIs after SDK initialization. Calls made before the initialization are invalid. The SDK provides the following additional APIs.

Updating the Device ID

```
/**
 * Updates the device ID (after initialization).
```

```
 * @param context Context.
 * @param deviceId Device ID.
 */
public static void updateUniqueId(Context context, String deviceId);
```

Updating the User ID

```
/**
 * Updates the user ID (after initialization).
 * @param context Context.
 * @param userId User ID.
 */
public static void updateUserId(Context context, String userId);
```

Updating the Device Model

The SDK does not collect the phone model by default. You can use this API to update the phone model to the RUM Pro SDK after the model is obtained in a compliant manner.

```
/**
 * Updates the device model (after initialization).
 * @param context Context.
 * @param model Device model.
 */
public static void updateDeviceModel(Context context, String model);
```

Updating the Log Level

```
/**
 * Updates the log level (after initialization).
 * @param logLevel Log level, which can be obtained from BuglyLogLevel.
 */
public static void updateLogLevel(int logLevel);
```

Reporting a Custom Exception

If you need to report errors other than Java crashes, native crashes, and ANR, such as C# exceptions, you can use the custom exception API to report them to RUM Pro. The reported data is displayed on the Terminal Performance Monitoring Pro > Error page.

```
/**
 * Reports a custom exception.
 * @param thread Thread where the error occurs. The default value is the
current thread.
 * @param category Exception type. U3D C# : 4 | JavaScript : 8 | Cocos2d
Lua : 6.
 * @param errorType Error type.
 * @param errorMsg Error message.
 * @param stack Error stack.
 * @param extraInfo Additional information.
 */
public static void postException(Thread thread, int category, String
errorType, String errorMsg,
                                 String stack, Map<String, String>
extraInfo);
public static void postException(int category, String errorType, String
errorMsg,
                                 String stack, Map<String, String>
extraInfo);
```

Reporting a Caught Java Exception

If you want to report an exception caught in a Java try-catch block to RUM Pro, you can use the caught Java exception reporting API. The reported data is displayed on the Terminal Performance Monitoring Pro > Error page.

```
/**
 * Handles and reports a caught exception.
 * @param thread Thread where the error occurs.
 * @param exception Exception.
 * @param extraMsg Additional information.
 * @param extraData Additional data.
 * @param enableAllThreadStack Specifies whether to enable stack trace
capturing for all threads. This is enabled by default.
 * @return Reporting result.
 */
public static boolean handleCatchException(Thread thread, Throwable
exception, String extraMsg,
```

```
                                                byte[] extraData, boolean
enableAllThreadStack);
public static boolean handleCatchException(Thread thread, Throwable
exception, String extraMsg,
                                                byte[] extraData);
```

Configuring a Case Tag

```
/**
 * This is not thread-safe. It is recommended to switch to the same
thread for this operation.
 * Configures case tags using the tag IDs in the RUM Pro console.
 * 1. First, add a tag in the RUM Pro console to obtain the tag ID.
 * 2. Configure the tag ID using setCaseLabels. To configure multiple
tags, separate them with a vertical bar (|).
 * 3. These tag IDs are reported along with the exception case.
 * Example: "123|456|789".
 * SDK version: 4.3.2.9+
 * @param labels Case tags.
 */
public static void setCaseLabels(String labels);
```

Configuring Business Drill-Down

```
/**
 * This is not thread-safe. It is recommended to switch to the same
thread for this operation.
 * Configures business drill-down tags. To configure multiple tags,
separate them with a vertical bar (|).
 * A maximum of 30 tags can be configured. If more than 30 tags are
specified, only the first 30 tags are used.
 * Each tag cannot exceed 1024 characters in length. Tags longer than
this limit cannot be added successfully.
 * Example: "test_one|test_two|test_three".
 * SDK version: 4.4.1+
 * @param labels Business drill-down tags.
 */
public static void setTestLabels(String labels);
```

Configuring a Suspected Object for Java Memory Leaks (Performance Monitoring)

After you enable Java memory leak monitoring in RUM Pro, you can use the following API to configure suspected leak objects for the SDK to monitor. If this API is not configured, the SDK only monitors memory leaks for Activities and Fragments.

```
/**
 * Configures a suspected object for Java memory leaks.
 * @param leakObj Suspected object.
 */
public static void startInspectLeakObj(Object leakObj);
```

Test API

The SDK provides an API to test crashes. During debugging, you can use this API to test crash collection and reporting.

```
/**
 * Crash testing.
 * @param crashType Crash type. Valid values: Bugly.JAVA_CRASH,
Bugly.NATIVE_CRASH, and Bugly.ANR_CRASH.
 */
public static void testCrash(@CrashTypeKey int crashType);
```

Custom Log APIs

The SDK provides the capability to attach custom logs in the event of an exception. Before a crash occurs (excluding callbacks), you can call the following APIs to write custom logs.

```
public class BuglyCustomLog {
    public static void v(String tag, String msg) {
        BuglyLog.v(tag, msg);
    }

    public static void d(String tag, String msg) {
        BuglyLog.d(tag, msg);
    }

    public static void i(String tag, String msg) {
        BuglyLog.i(tag, msg);
```

```
    }

    public static void w(String tag, String msg) {
        BuglyLog.w(tag, msg);
    }

    public static void e(String tag, String msg) {
        BuglyLog.e(tag, msg);
    }

    public static void e(String tag, String msg, Throwable throwable) {
        BuglyLog.e(tag, msg, throwable);
    }

    /**
     * Configures the log cache size. Logs are written in a loop based
on the configured cache size.
     * @param size Log cache size.
     */
    public static void setCache(int size) {
        BuglyLog.setCache(size);
    }
}
```

After custom logs are written, if an exception occurs in the process, the custom logs will be reported and displayed as `BuglyLog.zip` in the **`attachment`**.

Custom Data

Adding and Removing Custom Data (Crash Monitoring)

The SDK allows you to add custom data to save custom environment information at the time of a crash or before it occurs. The custom data is reported along with the exception and displayed in the **Custom Field** section on the exception case details page. You can add a maximum of 50 key-value pairs as custom data.

```
/**
 * Adds custom data.
 * @param context context
 * @param key key
 * @param value value
```

```
 */
public static void putUserData(Context context, String key, String
value);
```

If you need to remove custom data, use the following API.

```
/**
 * Removes custom data.
 * @param key key
 * @return If the data exists, the value for the key is returned.
 */
public static String removeUserData(Context context, String key);
```

Configuring and Updating a Custom File Upload Path (Crash Monitoring)

The SDK allows you to upload custom large files and associate them with exception cases. Custom file uploads use a separate channel from exception reporting, ensuring no interference with the exception handling process. Multiple files can be configured. You can configure and update the array of file paths via the following API at any time before an exception occurs. After the process restarts due to an exception, the custom files will be reported and displayed as custom_log.zip in the **attachment** of the exception case.

```
/**
 * Configures or updates the custom file path array.
 * @param files File path array. A maximum of 10 files is allowed. The
compressed size cannot exceed 10 MB.
 * @return Configuration result.
 */
public static boolean setAdditionalAttachmentPaths(String[] files);
```

Important:

- This API allows a maximum of 10 file paths.
- The compressed size limit for the uploaded files is 10 MB. Files exceeding this limit are not uploaded.
- This API can be called multiple times, and the updated file paths will overwrite the previous values.
- The custom files are uploaded during the second startup of the process, introducing a delay relative to exception reporting.
- Frequent crashes may trigger the file cleanup logic, which may lead to missed file uploads.

- The sampling rate for custom file uploads can be configured.

Activating and Deactivating a Custom Scenario (Performance Monitoring)

In performance monitoring, you can customize scenarios and execute specific monitoring or callback operations based on these scenarios. You can use the following APIs to activate and deactivate custom scenarios.

```
/**
 * Activates a custom scenario.
 * @param sceneName Scenario name.
 */
public static void enterScene(String sceneName);

/**
 * Deactivates a custom scenario.
 * @param sceneName Scenario name.
 */
public static void exitScene(String sceneName);
```

Note:

- When a scenario is active (between enterScene and exitScene), if an issue is detected, the reporting scenario is the [custom scenario].
- After exitScene is called, if an issue is detected, the reporting scenario is [simpleName Of lastResumedActivity].
- It is recommended to use enterScene and exitScene in pairs, such as: enterScene(A) -> exitScene(A) -> enterScene(B) -> exitScene(B). The following example shows the scenario if the APIs are not called in pairs:
  - enterScene(A) --> The custom scenario is A.
  - enterScene(B) --> The custom scenario is B.
  - enterScene(C) --> The custom scenario is C.
  - exitScene(B) --> The custom scenario is C.
  - exitScene(C) --> The custom scenario is empty.
  - exitScene(A) --> The custom scenario is empty.
- If sceneName is empty, the API returns immediately.

Adding and Removing a Custom Data Collector (Performance Monitoring)

The configuration method for the performance monitoring module is different from that for the crash monitoring module. The custom data for the performance monitoring module includes **custom dimensions** and **custom fields**.

- **Custom dimensions**: These are three sets of keys (for more information, see ICustomDataEditor) specified by the SDK, with 10 keys per set. The application can select appropriate keys to report data. In the console, you can set aliases for these keys for viewing and analysis. For custom dimensions, the server stores them as separate fields. This enables rich query and analysis capabilities. Currently, global settings or callback settings before data reporting are supported. You can obtain the global API via `Bugly.getGlobalCustomDataEditor` to set custom dimensions, or use metric callbacks (`ICustomDataCollector`) or issue callbacks (`ICustomDataCollectorForIssue`) to configure or update custom dimensions.
- **Custom fields**: These involve keys and values freely configured by the application. The SDK does not interpret the field types and stores them as strings. In the console, you can query reports containing specific content via custom fields in the issue list and issue details. You can only configure custom fields via the issue callback `ICustomDataCollectorForIssue` .

Global Settings

Global settings mean the application can call the global setting API at any time to configure custom dimensions. For the same key, repeated settings update its existing value. During performance reporting, the globally cached custom dimension data is obtained.
You can use the following API to obtain the default global custom dimension collection callback and configure global custom dimensions.

```
/**
 * Obtains the global custom dimension setting API.
 * @return Global custom dimension setting class.
 */
public static ICustomDataEditor getGlobalCustomDataEditor();

// Configuration example
ICustomDataEditor customDataEditor = Bugly.getGlobalCustomDataEditor();
customDataEditor.putStringParam(ICustomDataEditor.STRING_PARAM_1,
getAppStateDesc());
customDataEditor.putNumberParam(ICustomDataEditor.NUMBER_PARAM_1,
Debug.getPss());
```

Callback Settings

Callback settings mean the application configures callbacks. Before detected issues are reported, the API callback of the application is triggered to collect additional contextual information based on the specific issue and scenario. Callback settings are categorized into metric callbacks and issue callbacks.

```
// Adds a callback. Metric callbacks and issue callbacks need to be
configured separately.
Bugly.addCustomDataCollector(customDataEditor);

// Removes a callback. Metric callbacks and issue callbacks need to be
removed separately.
Bugly.removeCustomDataCollector(customDataDditor);
```

- **Metric callbacks**: `ICustomDataCollector`. For example, lag metric and memory peak callbacks are metric callbacks. These callbacks are triggered before information collection and data caching, allowing the application to add additional information if needed.

```
/**
 * Configures a metric data callback. Multiple callbacks can be
configured for different monitoring items and scenarios.
 * @param dataCollector Global custom data callback class.
 */
public static void addCustomDataCollector(ICustomDataCollector
dataCollector);
/**
 * Removes a global custom data callback.
 * @param dataCollector Global custom data callback class.
 */
public static void removeCustomDataCollector(ICustomDataCollector
dataCollector);

// Configuration example
ICustomDataEditor customDataEditor = new ICustomDataCollector() {
    @Override
    public void collectCustomData(String pluginName, String scene,
ICustomDataEditor customDataEditor) {

customDataEditor.putStringParam(ICustomDataEditor.STRING_PARAM_1,
getAppStateDesc());
```

```
customDataEditor.putNumberParam(ICustomDataEditor.NUMBER_PARAM_1,
Debug.getPss());
    }
};

// Configures the metric callback.
Bugly.addCustomDataCollector(customDataEditor);
// Removes the metric callback.
Bugly.removeCustomDataCollector(customDataDditor);
```

- **Issue callbacks**: `ICustomDataCollectorForIssue`. For example, lag monitoring, Java memory leak, and large image analysis callbacks are issue callbacks. These callbacks are triggered when abnormal issues are detected, allowing you to add additional information if needed.

```
/**
 * Configures a single-metric issue custom data callback. Multiple
callbacks can be configured for different monitoring items and
scenarios.
 * @param issueDataCollector Single-metric issue custom data callback
class.
 */
public static void addCustomDataCollector(ICustomDataCollectorForIssue
issueDataCollector);
/**
 * Removes a single-metric issue custom data callback.
 * @param issueDataCollector Single-metric issue custom data callback
class.
 */
public static void
removeCustomDataCollector(ICustomDataCollectorForIssue
issueDataCollector);

// Configuration example
ICustomDataCollectorForIssue issueDataCollector = new
ICustomDataCollectorForIssue() {
    @Override
    public void collectCustomData(String pluginName, String scene,
ICustomDataEditorForIssue customDataEditor) {
```

```
        // Allows overriding global custom dimension settings.

customDataEditor.putStringParam(ICustomDataEditor.STRING_PARAM_1,
getAppStateDesc());

customDataEditor.putNumberParam(ICustomDataEditor.NUMBER_PARAM_1,
Debug.getPss());
    }
}
// Configures the issue callback.
Bugly.addCustomDataCollector(issueDataCollector);
// Removes the issue callback.
Bugly.removeCustomDataCollector(issueDataCollector);
```

Dynamic Switches

If you need to dynamically enable or disable certain crash monitoring or performance monitoring items of RUM Pro in specific scenarios, you can use the following APIs. However, the dynamic crash switches should be used only when necessary, as they may result in missed exception reports.

```
/**
 * Dynamic switch for crash monitoring.
 * @param crashType Crash type. Valid values: Bugly.JAVA_CRASH,
Bugly.NATIVE_CRASH, and Bugly.ANR_CRASH.
 * @param isAble Set to true to enable monitoring, or false to disable
monitoring.
 */
public static void setCrashMonitorAble(@CrashTypeKey int crashType,
boolean isAble);

/**
 * Dynamic switch for performance monitoring (enable or disable a group
of monitoring items).
 * @param monitorList List of monitoring items, which can be obtained
from BuglyMonitorName.
 * @param isAble Set to true to enable monitoring, or false to disable
monitoring.
 */
public static void setPerformanceMonitorsAble(List<String> monitorList,
boolean isAble);
```

```
/**
 * Dynamic switch for performance monitoring (enable or disable a single
monitoring item).
 * @param monitorName Monitoring item.
 * @param isAble Set to true to enable monitoring, or false to disable
monitoring.
 */
public static void setPerformanceMonitorAble(String monitorName, boolean
isAble);

/**
 * Disables all performance monitoring items.
 */
public static void abolishPerformanceMonitors();
```

Startup Monitoring

By default, startup monitoring considers the end of the first frame rendering of the first Activity as the startup completion point. You can also use the `reportAppFullLaunch` API to specify a custom startup completion point in the application. Additionally, AppLaunch provides APIs for adding custom tags and custom tracking spans.

```
AppLaunch appLaunch = AppLaunchProxy.getAppLaunch();
appLaunch.reportAppFullLaunch();
```

- **Custom tags**: You can add custom tags to the startup data for subsequent drill-down analysis, as shown below:

  ```
  AppLaunch appLaunch = AppLaunchProxy.getAppLaunch();
  appLaunch.addTag("show_splash");
  ```

- **Custom tracking spans**: You can use spans to monitor time-consuming tasks during the startup process.

  ```
  AppLaunch appLaunch = AppLaunchProxy.getAppLaunch();
  appLaunch.spanStart("login", null); // Login task begins.
  ....
  appLaunch.spanStart("verification_code", "login"); // Verification
  code task begins.
  ```

```
....
appLaunch.spanEnd("verification_code"); // Verification code task
ends.
....
appLaunch.spanEnd("login"); // Login task ends.
```

Detailed description of the AppLaunch API:

```
public interface AppLaunch {
    /**
     * Starts startup monitoring.
     * @param context Context
     */
    void install(Context context);

    /**
     * Adds a custom tag.
     * @param tag Custom tag.
     */
    void addTag(String tag);

    /**
     * Starts a tracking span.
     * @param spanName Name of the tracking span to be started.
     * @param parentSpanName Name of the parent span for the current
tracking span.
     */
    void spanStart(String spanName, String parentSpanName);

    /**
     * Ends a tracking span.
     * @param spanName Name of the tracking span to be ended.
     */
    void spanEnd(String spanName);

    /**
     * Marks startup completion.
     */
    void reportAppFullLaunch();
```

```
}
```

iOS SDK Integration Guide
SDK Integration Overview

Last updated：2026-05-25 18:01:55

The Terminal Performance Monitoring Pro (RUM Pro) SDK is powered by the Tencent Bugly team. This article describes how to integrate the RUM Pro SDK in iOS. After the SDK is integrated and data reporting is verified, you can use the analysis features in the console.

SDK Overview

- **SDK version**: 2.8.1.5.
- **SDK features**: A professional application quality monitoring tool that provides collection and analysis services for abnormal data, helping developers identify and resolve issues in a timely manner to develop high-quality apps.
- **Service provider**: Tencent Cloud Computing (Beijing) Co., Ltd.
- RUM Pro SDK Compliance Guide.
- RUM Pro SDK Personal Information Protection Rules.

Integration Preparations

1. Before you integrate the SDK, please carefully read the RUM Pro SDK Compliance Guide.
2. Complete the SDK integration.
3. Complete the SDK initialization. During initialization, the SDK may collect certain user information. Please initialize the SDK only after the user has agreed to the RUM Pro SDK Personal Information Protection Rules. No information is collected before the SDK is initialized.
4. Verify data reporting.
5. For more information on other SDK features, see the API Description.

SDK Integration

Last updated：2026-05-25 18:01:55

The Terminal Performance Monitoring Pro (RUM Pro) SDK is powered by the Tencent Bugly team and uses the same SDK package as Bugly Pro. It supports both automatic and manual integration.

Version Upgrade Notice

Because the SDK contains cache data, older versions may not be fully compatible with the cache data from newer versions. **Therefore, downgrading to an earlier SDK version after an upgrade is not recommended** and may cause issues with reading cache data. If a downgrade is necessary, please contact us.

Prerequisites

RUM Pro requires Bugly iOS SDK 2.8.1.5 or later.

Automatic Integration (Recommended)

1. Add the following code to the Podfile of your project.

```
pod 'BuglyPro','2.8.1.5' // This version is an example. Refer to the
SDK update log to obtain the latest version.
```

2. Save the file and run `pod install`. Then, open the project using the file with the .xcworkspace extension.

```
pod install
```

Manual Integration (Not Recommended)

If you cannot use automatic integration due to special business requirements, security requirements, version control, or other reasons, you can also manually integrate the SDK.

1. Download the xcframework file of the RUM Pro iOS SDK.

   Download address for the RUM Pro iOS SDK: Central Repository: com/tencent/bugly
2. Drag the BuglyPro.xcframework file into the Xcode project (please select the **Copy items if needed** option).
3. Add the dependencies.
   - SystemConfiguration.framework
   - Security.framework
   - Network.framework

4. Check the project configuration.

   In the Build Settings of the Xcode project, check whether **-ObjC** is added under Other Linker Flags. If it is not present, add it manually.

SDK Initialization

Last updated：2026-05-25 18:01:56

This article describes how to initialize the SDK.

Operation Steps

Refer to the following code to initialize the Terminal Performance Monitoring Pro (RUM Pro) SDK after the app launches. It is generally recommended to perform initialization in `application:didFinishLaunchingWithOptions:delegate`.

1. Import the corresponding header file in the AppDelegate implementation file.

```
#import <BuglyPro/Bugly.h>
#import <BuglyPro/BuglyConfig.h>
#import <BuglyPro/BuglyDefine.h>
```

2. Start the framework in didFinishLaunchingWithOptions.

```
// Creates a config object with the product's APP_ID and APP_KEY.
BuglyConfig *config = [[BuglyConfig alloc] initWithAppId:APP_ID
appKey:APP_KEY];
// Required. Specifies the reporting domain type.
config.serverHostType = BuglyServerHostTypeCloud;
// Recommended. Specifies the build type. This field will be used in
later configurations.
config.buildConfig = BuglyBuildConfigGray;
// Optional. Sets Bugly callbacks.
config.delegate = self;
// Sets device ID and user ID as early as possible.
config.deviceIdentifier = @"device_id";
config.userIdentifier = @"user_id";

NSArray *modules = @[BUGLY_MODULE_CRASH, RM_MODULE_LOOPER,
RM_MODULE_MEMORY];
// NSArray *modules = RM_MODULE_ALL;
[Bugly start:modules config:config completeHandler:^{
   // SDK initialization completion callback
}];
```

Note:

- The device ID is critical because RUM Pro uses it to calculate the device exception rate. It is strongly recommended to set a correct device ID to ensure device uniqueness.
- Performance monitoring items can be configured in Application Configuration > SDK Configuration by adjusting the device sampling rate to enable or disable specific performance monitoring features.
- It is recommended to obtain user consent for the RUM Pro SDK Personal Information Protection Rules before initializing the SDK.

Data Reporting Verification

Last updated：2026-05-25 18:01:56

Prerequisites

- Crash data is reported at 100% and does not support sampling. All other monitoring capabilities support sampling, but they are disabled by default.
- You need to create a configuration task on the Terminal Performance Monitoring Pro > Setting > SDK configuration page. For more information on the configuration feature, see SDK Configuration.
- During integration, it is recommended to set up a **test number allowlist** as described in SDK Configuration. This involves setting all sampling rates to 1.0 to facilitate data reporting verification. After integration, you can adjust the sampling rates as needed. Alternatively, you can create multiple configuration tasks for development, canary, and production based on the version type (corresponding to the buildConfig in the initialized `BuglyConfig`). For the development task, enable all monitoring items; for the canary phase, sample as needed; for production, reduce the sampling rate according to requirements.
- Check whether the product to be integrated has purchased and bound an active resource package. If not, the product will be unable to report data.

Crash Monitoring

> **Note:**
> Crash capture relies on the system's signal listener interface. If your app integrates other SDKs with similar capabilities or contains logic that listens for signals, it may affect the crash capture success rate of this SDK. To mitigate this impact, ensure the SDK is initialized after all other listener logic has been registered.

Simulating an Exception

After initialization, you can simulate OC exceptions, C++ exceptions, and signal exceptions to verify data reporting. For details, see the Demo.
After a crash occurs, some exceptions are only reported on the next app launch. Reporting may also be delayed. For testing purposes, it is recommended to trigger an exception, restart the app, and then refresh the display page.

```
// Simulates an OC exception.
id data = [NSArray arrayWithObject:@"Hello World"];
[(NSDictionary *)data objectForKey:@""];
```

```
// Simulates a signal exception.
abort();

// Simulates a C++ exception.
throw "Something went wrong";
```

Checking Data Reporting

After a crash is reported, you can view the issue in Crash > Problem List. Click the issue to view its details.

ANR Monitoring

Similar to crash monitoring, you can simulate ANR using the test interface below or by performing a time-consuming task on the UI thread.

```
// Simulates an ANR.
while(YES)
{
    ;
}
```

After an ANR is reported, you can view the issue in ANR > Problem List. Click the issue to view its details.

FOOM Monitoring

FOOM Rate: FOOM (Foreground Out of Memory) generally refers to scenarios where an app is unexpectedly killed by the system due to excessive memory usage. Due to how FOOM is detected, this may also include cases where the app is unexpectedly killed for other reasons. You can simulate FOOM using the test interface below:

```
- (void)foomButtonClicked {
    self.timer = [NSTimer scheduledTimerWithTimeInterval:0.01
target:self selector:@selector(timerFired) userInfo:nil repeats:YES];
}

- (void)timerFired {
    CGSize size = CGSizeMake(1024 , 1024);
    const size_t bitsPerComponent = 8;
    const size_t bytesPerRow = size.width * 4;
    CGContextRef ctx = CGBitmapContextCreate(calloc(sizeof(unsigned
char), bytesPerRow * size.height), size.width, size.height,
```

```
                                                bitsPerComponent,
bytesPerRow,

CGColorSpaceCreateDeviceRGB(),

kCGImageAlphaPremultipliedLast);
    CGContextSetRGBFillColor(ctx, 1.0, 1.0, 1.0, 1.0);
    CGContextFillRect(ctx, CGRectMake(0, 0, size.width, size.height));
}
```

Checking Data Reporting

After an FOOM is reported, you can view the issue in FOOM > Problem List. Click the issue to view its details.

Error Monitoring

Errors generally refer to user-defined exceptions, such as caught OC/C++ exceptions, C# exceptions, JavaScript exceptions, and lua exceptions. These are typically reported using `reportException:`, `reportError:`, or `reportExceptionWithCategory:name:reason:callStack:extraInfo:terminateApp:` in the `BuglyCrashMonitorPlugin.h` file. For details, see Other Interfaces/Reporting Custom Exceptions.

Simulating an Exception

- To report a custom error, see the following sample code:

```
- (void)userDefinedCrash {
    NSError *error = [NSError errorWithDomain:@"Error Test" code:-2
userInfo:nil];
    [BuglyCrashMonitorPlugin reportError:error];
}
```

- To report OC/C++ exceptions, refer to the following sample code:

```
- (void)userDefinedCrash {
    @try {
        NSString *value = @"This is a string";
        [NSException raise:@"TurboEncabulatorException"
                    format:@"Spurving bearing failure: Barescent skor
motion non-sinusoidal for %p", value];
```

```
    } @catch (NSException *exception) {
        [BuglyCrashMonitorPlugin reportException:exception];
    }
}
```

- To report a custom exception, refer to the following sample code:

```
    NSArray * arr = [NSThread callStackSymbols];
    [BuglyCrashMonitorPlugin reportExceptionWithCategory:3
name:@"reportTest" reason:@"test" callStack:arr extraInfo:
[NSDictionary dictionary] terminateApp:false];
```

Checking Data Reporting

After an error is reported, view the issue in Error > Problem List. Click the issue to view its details.

> **Note:**
> In `reportExceptionWithCategory:name:reason:callStack:extraInfo:terminateApp:` method, callStack specifies the error stack, and extraInfo is displayed in the **Attachments** section.

Startup Monitoring

When startup monitoring is enabled, the startup module automatically installs the monitoring feature and reports startup data after SDK initialization. You can customize the startup completion point via the `[BuglyLaunchMonitorPlugin endColdLaunch]` API. Startup monitoring also supports custom tags and trace APIs to monitor time-consuming tasks during the startup process For more details on startup APIs, see API Description > Startup Monitoring.

```
[BuglyLaunchMonitorPlugin startSpan:@"parentSpan" parentSpanName:nil];
[BuglyLaunchMonitorPlugin startSpan:@"spanTest"
parentSpanName:@"parentSpan"];

dispatch_after(dispatch_time(DISPATCH_TIME_NOW, (int64_t)(3.0 *
NSEC_PER_SEC)), dispatch_get_main_queue(), ^{
    [BuglyLaunchMonitorPlugin endSpan:@"spanTest"];
    [BuglyLaunchMonitorPlugin endSpanFromLaunch:@"spanFromLaunch"];
});
```

```
dispatch_after(dispatch_time(DISPATCH_TIME_NOW, (int64_t)(4.0 *
NSEC_PER_SEC)), dispatch_get_main_queue(), ^{
    [BuglyLaunchMonitorPlugin endSpan:@"parentSpan"];
});

[BuglyLaunchMonitorPlugin addTag:@"tagTest1"];
[BuglyLaunchMonitorPlugin addTag:@"tagTest2"];
```

Checking Data Reporting

You can view reported issues in Launch > Launch instance. Click an issue to view its details.

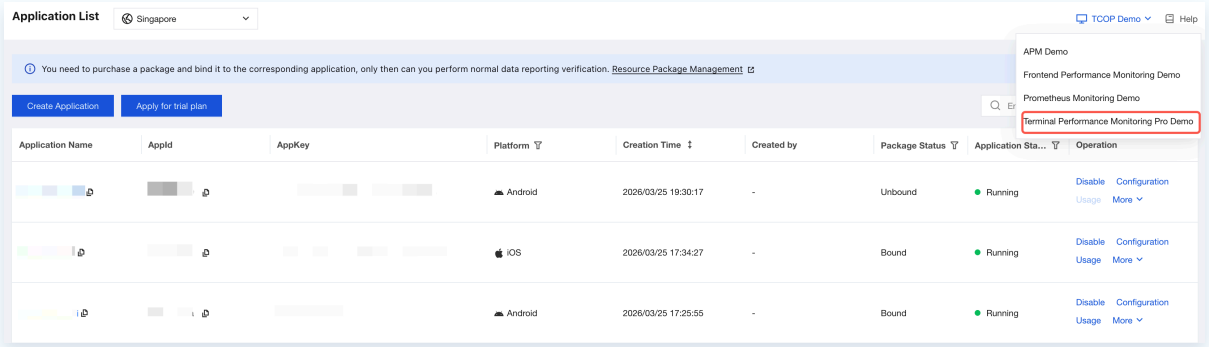
Lag Monitoring

- **Lag Metrics: FPS and Suspension Rate**

  When the lag metric (`"looper.fluency"`) is enabled, the SDK monitors application smoothness after initialization. During a single application run, data is first collected and stored locally, then aggregated and reported on the next startup. To avoid impacting application startup performance, the SDK aggregates and reports cached data to the backend 5 minutes after initialization.

  - FPS (Frames Per Second): The number of refreshes per unit time. Due to the introduction of high-refresh-rate screens (e.g., iPhone 13), FPS uses a normalized calculation method, where any value exceeding 60 FPS is normalized to 60.
  - Suspension Rate: The suspension rate is defined as the total time when the callback interval exceeds 200ms within a unit of time, divided by that unit time, multiplied by 3600. The unit is seconds per hour (s/h).

> **Note:**
> You can test list scrolling lag in the demo to observe FPS and suspension rate. Choose **Terminal Performance Monitoring Pro Demo** in the upper-right corner of the Terminal Performance Monitoring Pro > Application List page.



- **Lag Monitoring**

After lag monitoring (`"looper.hitches"`) is enabled, if time-consuming logic executed on the UI thread exceeds 500 ms, lag reporting is triggered. Lag monitoring detects UI thread lag by monitoring message execution on the UI thread.
During the verification phase, it is recommended to set both lag monitoring parameters `"event_sampl e_ratio"` (message sampling rate) and `"sample_ratio"` (device sampling rate) to 1. This ensures that lag reporting is triggered whenever the lag threshold is met.

Simulating an Exception

You can add a time-consuming operation to the tableView callback method. Refer to the following sample code to simulate lag reporting:

```
- (UITableViewCell *)tableView:(UITableView *)tableView
cellForRowAtIndexPath:(NSIndexPath *)indexPath {
    UITableViewCell *cell = [[UITableViewCell alloc]
initWithStyle:UITableViewCellStyleDefault
reuseIdentifier:@"defaultcell"];
    cell.textLabel.text = [self.dataList objectAtIndex:indexPath.row];
    cell.textLabel.font = [UIFont systemFontOfSize:80];

    BOOL hitch = arc4random() % 5;
    if (hitch) {
        [NSThread sleepForTimeInterval:0.02];
    }

    return cell;
}
```

Checking Data Reporting

After lag is reported, you can view the issue in Lag > Problem List. Click the issue to view its details.

Memory Monitoring

Peak Memory

Peak memory usage refers to the maximum phys_footprint value during the entire lifecycle of the app. A higher peak memory usage indicates that the app consumes more memory in worst-case scenarios.
After data is reported, you can view the issues in Memory > Metric Analysis.

VC Memory Leaks

VC memory leaks refer to the monitoring of leaks in UIViewController within the iOS app development framework. You can verify this using the following sample code:

```
    UIStoryboard *storyboard = [UIStoryboard
storyboardWithName:@"RMVCLeakExample"
                                                        bundle:
[NSBundle mainBundle]];

    UIViewController * subVC = [storyboard
instantiateViewControllerWithIdentifier:@"rm.vcleak.subvc"];

    [self.navigationController pushViewController:subVC animated:YES];

    self.holdVC = subVC;
```

After a leak occurs and is reported, you can view the issue in Memory > VC memory leak > VC memory leak issue List. Click the issue to view its details.

Large Memory Allocation Monitoring

Large memory allocation monitoring tracks scenarios where an app allocates substantial memory blocks, helping you identify memory-intensive business logic. When an app triggers a memory allocation that exceeds a certain threshold, this monitoring collects and reports data including the allocation stack trace, current business context, and memory usage. After data is reported, you can view the issues in Memory > Large memory allocations.

Large memory allocations do not necessarily indicate a defect, but they highlight business logic with significant memory consumption that requires attention and optimization. Addressing these can help prevent rapid memory depletion that could lead to FOOM situations.

API Description

Last updated：2026-05-25 18:01:56

This article describes the feature APIs of the Terminal Performance Monitoring Pro (RUM Pro) SDK to facilitate flexible and in-depth integration.

Initialization APIs

During SDK initialization, in addition to setting appId and appKey, you can also configure various user parameters in BuglyConfig.

```
/// Sets a custom version number.
@property (nonatomic, copy) NSString *appVersion;
/// Sets a custom device unique identifier.
@property (nonatomic, copy) NSString *deviceIdentifier;
/// Sets a custom user ID.
@property (nonatomic, copy) NSString *userIdentifier;
/// Sets build config to fetch different configurations.
@property (nonatomic, assign) BuglyBuildConfig buildConfig;

/// Bugly Delegate
@property (nonatomic, assign) id<BuglyDelegate> delegate;
```

Callback APIs

Setting the Crash Attachment Callback

SDK trigger a callback to your business logic when handling a crash. You can add custom logic in the callback or provide additional information to be reported along with the exception.

```
/// Allows you to attach data to case reports.
/// param eventType Case event type
/// @param customData Business custom fields
/// @param eventInfo Case information (currently empty)
- (NSDictionary<NSString *, NSString *> *)attachmentForEvent:(NSString
*)eventType
                                                    customData:
(BuglyCustomData *)customData
```

```
                                                    eventInfo:
(NSDictionary *)eventInfo;

#### 2. Callback for setting crash information

Bugly trigger a callback to your business logic when reporting a crash.
You can add custom logic in the callback.

``` objective-c
/**
 *  Callback when an exception occurs
 *  @param exception Exception information
 *  @return The record to be reported along with the exception
 */
- (NSString * __nullable)attachmentForException:(NSException *
__nullable)exception;
```

Setting the Log Callback

1. Implement a function callback method.

```
/// Custom log function
static void logger_func(RMLoggerLevel level, const char *log) {
    //handle sdk log.
}
```

2. Then register the method.

```
// Registers Bugly log output.
[Bugly registerLogCallback:logger_func];
```

Other APIs

You need to call other APIs after SDK initialization. Calls made before the initialization are invalid. The SDK provides the following additional APIs.

Updating the Device ID

```
/**
* Updates device ID.
```
```

```
* @param deviceId Device ID
*/
+ (void)updateDeviceIdentifier:(NSString *)deviceId;
```

Updating the User ID

```
/**
* Updates userId.
* @param userId User ID
*/
+ (void)updateUserIdentifier:(NSString *)userId;
```

Obtaining the SDK Version Number

```
/**
 * Gets SDK version information.
 * @return SDK version number
 */
+ (NSString *)sdkVersion;
```

Reporting a Custom Exception

If you need to report errors other than OC/C++ exceptions, such as C# or JavaScript exceptions, you can use the custom exception API to report them to RUM Pro. The reported data is displayed on the Error page.

```
/**
 *    @brief Reports a custom error.
 *
 *    @param category    Type (Cocoa=3, CSharp=4, JS=5, Lua=6)
 *    @param aName       Name
 *    @param aReason     Error reason
 *    @param aStackArray Stack trace
 *    @param info        Additional data
 *    @param terminate   Whether to terminate the app process after
reporting
 */
+ (void)reportExceptionWithCategory:(NSUInteger)category
                               name:(NSString *)aName
                             reason:(NSString *)aReason
```

```
                          callStack:(NSArray *)aStackArray
                          extraInfo:(NSDictionary *)info
                       terminateApp:(BOOL)terminate;
```

Reporting a Custom Error

If you want to report NSError information to RUM Pro, you can use the error reporting API. The reported data is displayed on the Error page.

```
/**
 * Reports an error.
 *
 *  @param error Error information
 */
+ (void)reportError:(NSError *)error;
```

Reporting an OC Exception

If you catches an exception using try-catch and wants to report it to RUM Pro, you can use the exception reporting API. The reported data is displayed on the Error page.

```
/**
 * Reports a custom Objective-C exception.
 *
 *  @param exception Exception information
 */
+ (void)reportException:(NSException *)exception;
```

Setting a Case Tag

```
/**
 * Updates case tags. This must be called after Bugly SDK initialization
(in the Bugly setup completeHandler callback); otherwise, data loss may
occur.
 * SDK version: 2.7.53.3+
 * @params tagArr String array (max length: 30, each string max length:
1024 bytes)
 */
+ (void)updateCaseTags:(NSArray<NSString *> *)tagArr;
```

Setting Business Drill-Down

```
/**
 * Update business drill-down. This must be called after Bugly SDK
initialization (the Bugly setup completeHandler callback); otherwise,
data loss may occur.
 * SDK version: 2.7.53.3+
 * @params tagArr String array (max length: 30, each string max length:
1024 bytes)
 */
+ (void)updateTestTags:(NSArray<NSString *> *)tagArr;
```

Custom Data

Adding Custom Data (Crash Monitoring)

RUM Pro allows you to add custom data to save custom environment information at the time of a crash or before it occurs. The custom data is reported along with the exception and displayed in the **Custom Fields** section on the exception case details page. You can add a maximum of 50 key-value pairs as custom data.

```
/**
 * Sets key data to be reported along with crash information.
 *
 *  @param value KEY
 *  @param key VALUE
 */
+ (void)setUserValue:(NSString *)value forKey:(NSString *)key;
```

To retrieve the custom data you have set, use the following API.

```
/**
 * Gets custom data.
 *
 * @return Key data
 */
+ (NSDictionary * _Nullable)allUserValues;
```

Setting and Updating a Custom File Upload Path (Crash Monitoring)

RUM Pro allows you to upload custom large files and associate them with exception cases. Custom file uploads use a separate channel from exception reporting, ensuring no interference with the exception handling process. Multiple files can be configured. You can configure and update the array of file paths via the following API at any time before an exception occurs. After the process restarts due to an exception, the custom files will be reported and displayed as custom_log.zip in the **attachments** to the exception case.

```
/**
 * Sets a collection of absolute paths for custom attachments.
 * The compressed file must not exceed 10 MB and will be uploaded on the
next startup.
 */
+ (void)setAdditionalAttachmentPaths:(NSArray<NSString *>*)pathArray;
```

Note:

- This API allows a maximum of 10 file paths.
- The compressed file must not exceed 10 MB. Files exceeding this limit will not be uploaded.
- This API can be called multiple times. Updated file paths will overwrite the previous values.
- Custom files are uploaded on the next startup, which introduces a delay relative to exception reporting.
- Frequent crashes may trigger file cleanup logic, potentially causing missed file uploads.
- You can configure the sampling rate for custom file uploads. For details, see SDK Configuration.
- **Calling this API when a crash occurs will not take effect**. You need to set the file paths in advance.

Activating and Deactivating a Custom Scenario (Performance Monitoring)

In performance monitoring, you can customize scenarios and execute specific monitoring or callback operations based on these scenarios. You can use the following APIs to activate and reset custom scenarios.

- **Set a Custom Scenario**

```
/// Updates scenario information.
/// @param key Scenario ID. Features such as lag, memory, and resource
monitoring will cluster data based on this value.
+ (void)setScene:(NSString *)key;
```

- **Reset a Custom Scenario**

```
/// Ends the custom scenario and resumes using the scenario
information automatically captured by RMonitor.
+ (void)resetScene;
```

Adding and Removing a Custom Data Collector (Performance Monitoring)

The custom settings for the performance monitoring module in RUM Pro differ from those in the crash monitoring module. The performance monitoring module supports two types of custom data: custom dimensions and custom fields.

- **Custom dimensions**: Three sets of keys are predefined by the SDK (see `BuglyCustomData.h` for details), with 10 keys per set. You can select appropriate keys to report data as needed. In the console, you can assign aliases to these keys for easier viewing and analysis. For custom dimensions, the server stores each field separately, enabling advanced query and analysis capabilities in the future. Currently, both global settings and pre-reporting callbacks are supported.
- **Custom fields**: Keys and values are freely defined by you. The SDK does not interpret the field types and stores all values as strings. In the console, you can search for reports containing specific content using these custom fields in the issue list and issue details pages.

> **Note:**
> **Global settings** mean you can call the global setting API at any time to configure custom dimensions. For the same key, repeated calls update the existing value. When performance data is ready to be reported, the globally cached custom dimension data is retrieved.

Use the following API to get the default callback for collecting global custom dimensions, and then set the dimensions through this callback.

```
/**
 * Adds custom tags. Calling this before SDK initialization has no
effect.
 * @param data Custom fields to be updated
 */
+ (void)updateCustomData:(BuglyCustomData *)data;

/**
 * Gets a copy of the current custom tags. Returns empty if not set.
 * @return Configured custom fields
 */
+ (nullable BuglyCustomData *)currentCustomData;
```

```
/**
 * Adds custom data for specific event reporting.
 * @param data Custom fields to be updated
 * @param eventType The corresponding event type
 */
+ (void)updateCustomData:(BuglyCustomData *)data forEvent:
(BuglyEventTypeName)eventType;

/**
 * Gets custom data for a specific event.
 * @param eventType The event type to get
 * @return The corresponding configured custom fields
*/
+ (nullable BuglyCustomData *)currentCustomDataForEvent:
(BuglyEventTypeName)eventType;

/**
 * Updates this custom data with new custom fields.
 * @param dict Dictionary information conforming to the BuglyCustomData
storage format
 * @return The updated data
 */
- (BuglyCustomData *)customDataByUpdateDict:(NSDictionary *)dict;

/**
  *Gets numeric custom data.
  * @param key If the key is not defined in NumberParamKey, returns 0.0
immediately.
  * @return The current value if set; otherwise, 0.0.
  */
- (nullable NSNumber *)getNumberParam:(BuglyCustomNumberDataKey)key;

/**
 * Sets numeric custom data.
 * @param key If the key is not defined in NumberParamKey, adding fails.
 * @param param NUMBER VALUE
 */
- (BOOL)putNumberParam:(nullable NSNumber *)param forKey:
(BuglyCustomNumberDataKey)key;
```

```
/**
  * Gets string custom data.
  * @param key If the key is not defined in StringParamKey, returns ""
immediately.
  * @return The current value if set; otherwise, "".
  */
- (nullable NSString *)getStringParam:(BuglyCustomStringDataKey)key;

/**
 * Sets string custom data.
 * @param key If the key is not defined in StringParamKey, adding fails.
 * @param param The length must not exceed
BUGLY_CUSTOM_DATA_MAX_STRING_VALUE_LENGTH.
 *     If value = null, it will be set to an empty string.
 *     If value exceeds the length limit, it will be truncated to the
first BUGLY_CUSTOM_DATA_MAX_STRING_VALUE_LENGTH characters.
 */
- (BOOL)putStringParam:(nullable NSString *)param forKey:
(BuglyCustomStringDataKey)key;

/**
 * Gets string array custom data.
 * @param key If the key is not defined in StringArrayParamKey, returns
an empty list immediately.
 * @return The current value if set; otherwise, an empty list. The
returned list cannot be modified.
 */
- (nullable NSArray<NSString *> *)getStringArrayParam:
(BuglyCustomArrayDataKey)key;

/**
 * The addition sequence is: A1 -> A2 -> A2 -> A3 -> A4. This API
produces a result similar to: [A1, A2, A3, A4].
 * Adds VALUE to string array custom data.
 * @param key If the key is not defined in StringArrayParamKey, adding
fails.
 * @param param A non-null, non-empty string. If the value does not
exist, it will be added; if it exists, returns immediately. The length
must not exceed BUGLY_CUSTOM_DATA_MAX_STRING_VALUE_LENGTH.
 */
```

```
- (BOOL)addStringToStringArrayParam:(NSString *)param forKey:
(BuglyCustomArrayDataKey)key;

/**
 * Removes VALUE from string array custom data.
 * @param key If the key is not defined in StringArrayParamKey, removal
fails.
 * @param param If the value exists, it will be removed; if it does not
exist, returns immediately.
 */
- (BOOL)removeStringFromStringArrayParam:(NSString *)param forKey:
(BuglyCustomArrayDataKey)key;

/**
 * The addition sequence is: A1 -> A2 -> A2 -> A3 -> A4. This API
produces a result similar to: [A1, A2, A2, A3, A4].
 * Adds VALUE to string array custom data.
 * @param key If the key is not defined in StringArrayParamKey, adding
fails.
 * @param param The length must not exceed
BUGLY_CUSTOM_DATA_MAX_STRING_VALUE_LENGTH. The value is added regardless
of whether it exists, allowing duplicates.
 * @return Returns failure if the value is null, empty, exceeds the
length limit, or the array has reached its maximum size; otherwise,
returns success.
 */
- (BOOL)addStringToSequence:(NSString *)param forKey:
(BuglyCustomArrayDataKey)key;
```

Startup Monitoring

Definition

The **cold start duration** calculation differs between iOS versions before 15 and iOS 15 and later.

- iOS versions prior to 15: **Cold start duration = First frame rendering time − App process creation time**
- iOS 15 and later:

  Apple introduced Prewarming in iOS 15, which can pre-launch processes. This makes the period from "App process initialization end time" to "`main` function execution time" highly unpredictable. Therefore, this uncertain interval is excluded from the cold start duration calculation, which is adjusted as

follows: **Cold start duration = First frame rendering time − main function execution time + App process initialization end time − App process creation time**

API Overview

Custom Scenario Duration Statistics

- Call `+ (void)startSpan:(NSString *)spanName parentSpanName:(nullable NSString *)parentSpanName` of `BuglyLaunchMonitorPlugin` to record the start timestamp of a custom scenario.

```
[BuglyLaunchMonitorPlugin startSpan:@"testSpan" parentSpanName:nil];
```

- Call `+ (void)endSpan:(NSString *)spanName` of `BuglyLaunchMonitorPlugin` to mark the end of a custom scenario interval. Ensure the spanName matches the one used when starting the span; otherwise, the data will be considered invalid.

```
[BuglyLaunchMonitorPlugin endSpan:@"testSpan"];
```

- Call `+ (void)endSpanFromLaunch:(NSString *)spanName` of `BuglyLaunchMonitorPlugin` to record the duration of a span measured from process creation.

```
[BuglyLaunchMonitorPlugin endSpanFromLaunch:@"testSpan"];
```

Setting Startup Tags

- Call `- (void)addTag:(NSString *)tagName` of `BuglyLaunchMonitorPlugin` to add a tag.
- Call `- (void)removeTag:(NSString *)tagName` of `BuglyLaunchMonitorPlugin` to remove a tag.

```
[BuglyLaunchMonitorPlugin addTag:@"tagTest1"];
[BuglyLaunchMonitorPlugin removeTag:@"tagTest1"];
```

Customizing Launch End Time

For business-specific reasons, the point at which startup is considered complete may not align with the end of `CA::Transaction::commit`. Therefore, the `- (void)endColdLaunch` API is added for you to actively call and declare that the cold start has ended. Calling this API will trigger startup data reporting, and subsequent updates to tags or span information will not take effect.

If the API is not called, startup data reporting will still be triggered when the app is backgrounded, crashes, or after a certain timeout period (default: 3 minutes).

```
[BuglyLaunchMonitorPlugin endColdLaunch];
```

Flutter SDK Integration Guide
SDK Integration Overview

Last updated：2026-05-25 18:01:56

The Terminal Performance Monitoring Pro (RUM Pro) SDK is powered by the Tencent Bugly team, offering error monitoring, lag metric collection, and launch monitoring. This article describes how to integrate the RUM Pro SDK in Flutter. After the SDK is integrated and data reporting is verified, you can use the analysis features in the console.

SDK Overview

- **SDK version**: 0.4.19.
- **SDK features**: A professional application quality monitoring tool that provides collection and analysis services for abnormal data, helping developers identify and resolve issues in a timely manner to develop high-quality apps.
- **Service provider**: Tencent Cloud Computing (Beijing) Co., Ltd.
- RUM Pro SDK Compliance Guide.
- RUM Pro SDK Personal Information Protection Rules.

Integration Steps

1. Before you integrate the SDK, please carefully read the RUM Pro SDK Compliance Guide.
2. Complete the SDK integration.
3. Complete the SDK initialization. During initialization, the SDK may collect certain user information. Please initialize the SDK only after the user has agreed to the RUM Pro SDK Personal Information Protection Rules. No information is collected before the SDK is initialized.
4. Verify data reporting.
5. For more information on other SDK features, see the API Description.

SDK Integration

Last updated：2026-05-25 18:01:56

This article describes how to integrate the Flutter SDK using automatic integration.

Prerequisites

- Terminal Performance Monitoring Pro (RUM Pro) requires Bugly Flutter SDK version 0.4.19 or later.
- Before integration, ensure that you have Created an Application in RUM Pro and obtained the corresponding AppKey and AppID.

Automatic Integration

1. Add the `bugly_pro_flutter` dependency to the `pubspec.yaml` of your project.

```
bugly_pro_flutter:
  version: 0.4.19
```

2. Run `flutter pub get` to update dependencies.

```
flutter pub get
```

3. Configure the obfuscation allowlist for Android.

   If obfuscation is enabled for your Android project, you need to add the following allowlist rule.

```
-keep class java.com.tencent.bugly.**{*;}
```

SDK Initialization

Last updated：2026-05-25 18:01:56

This article describes how to initialize the Terminal Performance Monitoring Pro (RUM Pro) Bugly Flutter SDK.

Sample Code

Add the initialization code to the `main` function in your project's `main.dart` file:

```
import 'package:bugly_pro_flutter/bugly.dart';

void main() {
  Bugly.setSdkRegion(SdkRegion.BUGLY_CLOUD);  // Sets the reporting
domain.
  BuglyOptions options = BuglyOptions(appId: '', appKey: '', bundleId:
'');
  options.monitorTypes = [
    MonitorType.launchMetric,
    MonitorType.looperMetric,
    MonitorType.looperStack,
    MonitorType.exception
  ];
  options.userId = 'pro_tester';

  Bugly.init(options, appRunner: (){
    runApp(const MyApp());
  },
  beforeInitRunner: (options){
    options.appId = 'afxxxxb01';          // appid of the registered
product
    options.appKey = 'aef434ba-xxxx-xxxx-xxxx-030b10aae88b'; // appkey
of the registered product
    options.bundleId = 'com.xxx.bundleid';
  });
}
```

Note:

- The first line of initialization code must call Bugly.setSdkRegion(SdkRegion.BUGLY_CLOUD) to set the reporting domain to Tencent Cloud.
- It is recommended to initialize the RUM Pro SDK only after obtaining user consent to the RUM Pro SDK Personal Information Protection Rules.

Integration FAQs

Resolving "Zone mismatch" Error During Flutter Startup

- Error description: A Dart error "Zone mismatch" occurs during startup.
- Cause analysis: This error is typically caused by WidgetsFlutterBinding.ensureInitialized() and runApp() being executed in different zones.
- Solution: Ensure both methods are executed in the same zone.

Correct code organization:

```
import 'package:bugly_pro_flutter/bugly.dart';

void main() {
  BuglyOptions options = BuglyOptions(appId: '', appKey: '', bundleId:
'');
  options.monitorTypes = [
    MonitorType.launchMetric,
    MonitorType.looperMetric,
    MonitorType.looperStack,
    MonitorType.exception
  ];
  options.userId = 'pro_tester';

  WidgetsFlutterBinding.ensureInitialized();
  final zone = Zone.current;    // !!!!!! Capture the zone!!!!!!

  Bugly.init(options, appRunner: (){
    // runApp(const MyApp());
    zone.run(() => runApp(const MyApp())); // !!!!!!Change to run runApp
within the zone here!!!!!!
  },
  beforeInitRunner: (options){
    options.appId = 'afxxxxb01';          // appid of the registered
product
```

```
    options.appKey = 'aef434ba-xxxx-xxxx-xxxx-030b10aae88b'; // appkey
of the registered product
    options.bundleId = 'com.xxx.bundleid';
  });
}
```

Note:

- Execute WidgetsFlutterBinding.ensureInitialized() first and save the Zone where this API is executed.
- Use the saved zone to execute runApp() in Bugly.Init to ensure WidgetsFlutterBinding.ensureInitialized() and runApp() are executed in the same zone.
- For asynchronous operations, use async/await properly.

Solving Android Platform Compilation Issues

After integration, if the app crashes with the error message `Abort message: 'JNI DETECTED ERROR IN APPLICATION: java_class == null in call to GetStaticMethodID`, and code obfuscation is enabled in the project, you need to add Bugly-related classes to the keep allowlist.

```
-keep class java.com.tencent.bugly.**{*;}
```

Solving iOS Platform Compilation Issues

When integrating the Flutter APM SDK into an iOS native project (host project), you may encounter C++ linking errors such as Undefined symbols for architecture arm64. This error indicates missing C++ symbols. The causes are as follows:

- Parts of the Bugly Flutter SDK are implemented in C++, so the host project must support C++ runtime libraries.
- By default, iOS projects created with Xcode do not automatically link against the C++ standard library (libc++), so you need to add it manually.

Solution steps:
In Xcode, explicitly add the libc++.tbd (or libc++.dylib) dependency:

1. Open the Xcode project.

   Double-click your project's ios/Runner.xcworkspace (for Flutter project) or the native project file.
2. Navigate to the build configuration. In the project navigator, perform the following operations:

   2.1 Select your target (usually Runner).

2.2 Switch to the "Build Phases" Tab.

3. Add the C++ library.
   3.1 Locate the "Link Binary With Libraries" section.
   3.2 Click the "+" button, then search for libc++.
   3.3 Select **libc++.tbd** (recommended) or **libc++.dylib** (both have the same functionality; .tbd is the newer lightweight version).
4. Verify the configuration. Perform the following operations in the "Build Settings" Tab:
   4.1 Search for Other Linker Flags.
   4.2 Ensure that –lc++ is included (if manually added earlier, you can remove it to avoid duplication).
5. Recompile the project.
   Perform `Clean Build` (Cmd + Shift + K) and then rerun.

Resolving Dependency Library Conflicts

If running `flutter pub get` after integrating the Flutter SDK reports dependency conflicts, try adjusting the conflicting library version to match your project's requirements. For example, if the business depends on uuid library 4.4.2 but the Flutter SDK depends on 3.0.6, running `flutter pub get` will result in an error. To resolve this, add the following content in pubspec.yaml. If the issue persists, please submit a ticket to contact us.

```
# Force overwrite the uuid dependency to version 4.4.2.
dependency_overrides:
  uuid: ^4.4.2
```

Data Reporting Verification

Last updated：2026-05-25 18:01:56

Prerequisites

- Dart exceptions, startup performance metrics, lag metrics, and network reporting all need to be enabled and support sampling. During integration, ensure that the enabled features meet your expectations.
- You need to create a configuration task on the Terminal Performance Monitoring Pro > Setting > SDK configuration page. For more information on the configuration feature, see SDK configuration.
- During integration, it is recommended to set the sampling rate of all monitoring items to 1.0 (sample_ratio and event_sample_ratio) for easier data reporting verification. After the integration is complete, you can change the sampling rate as needed. You can enable all monitoring items for development tasks, apply sampling as needed during the grayscale phase, and reduce sampling for the official release as required.
- Check whether the product to be integrated has purchased and bound an active resource package. If not, the product will be unable to report data.

Error Monitoring

Simulating an Exception

Refer to the following sample code to simulate various types of exceptions:

```
class MyCrashTestApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(title: const Text('APM Exception Capture Test')),
        body: Center(
          child: Column(
            mainAxisAlignment: MainAxisAlignment.center,
            children: [
              // 1. Synchronous exception test
              ElevatedButton(
                child: const Text('Trigger Synchronous Exception'),
                onPressed: () => _throwSyncException(),
              ),
```

```
              const SizedBox(height: 20),

              // 2. Asynchronous exception test
              ElevatedButton(
                child: const Text('Trigger Asynchronous Exception'),
                onPressed: () => _throwAsyncException(),
              ),

              const SizedBox(height: 20),

              // 3. Widget build exception
              ElevatedButton(
                child: const Text('Trigger Widget Exception'),
                onPressed: () => Navigator.push(
                  context,
                  MaterialPageRoute(builder: (_) => FaultyWidget())
                ),
              ),
            ],
          ),
        ),
      ),
    );
  }

  // Synchronous exception example
  void _throwSyncException() {
      // Simulates a business logic error.
      final list = [];
      print(list[0]); // Out-of-bounds access
  }

  // Asynchronous exception example
  Future<void> _throwAsyncException() async {
      await Future.delayed(Duration(milliseconds: 500));
      // Simulates an asynchronous error.
      dynamic nullObject = null;
      print(nullObject.length); // NoSuchMethodError
  }
}
```

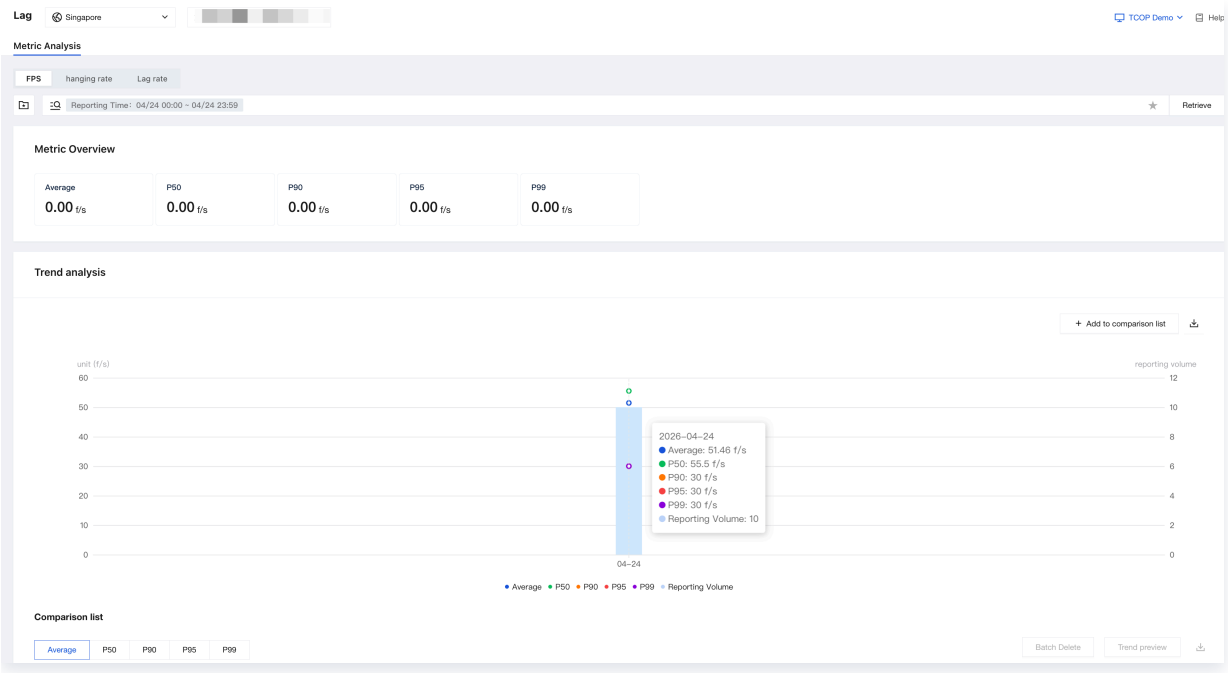
```
// Exception-throwing Widget
class FaultyWidget extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    // Intentionally throws an exception during build.
    if (DateTime.now().millisecond % 2 == 0) {
      return Container(color: Colors.blue);
    }
    throw Exception('Random exception during Widget build');
  }
}
```

Checking Data Reporting

After an error is reported, view the issue in Error > Problem List. Click the issue to view its details.

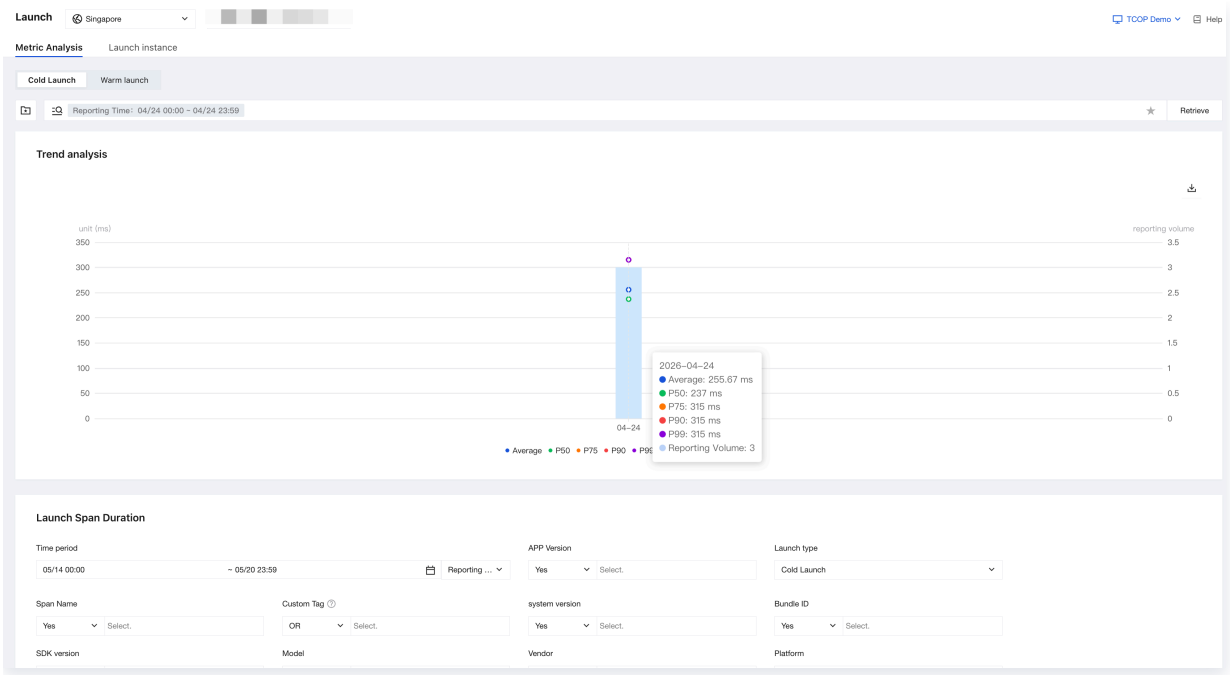
Lag Metrics

As long as the lifecycle listeners are properly configured, lag data will be captured. For detailed instructions, see API Description > Lifecycle Listening. You can view the data on the Lag page.



Launch Monitoring

As long as the launch monitoring instrumentation is properly configured, launch data will be captured. For detailed instructions, see API Description > Configuring Launch Monitoring Instrumentation. You can view the data on the Launch page.



API Description

Last updated：2026-05-25 18:01:56

This article describes the feature APIs of the Terminal Performance Monitoring Pro (RUM Pro) SDK to facilitate flexible and in-depth integration.

Adding a Page Switching Behavior

You can add page switching behavior based on the framework you use. Rum Pro stores up to 256 records. They are automatically reported when an error occurs. You can view them in **Context Data** > **Page Tracking**.

```
Bugly.addPageAction('pageAction');
```

Lifecycle Listening

Set up `BuglyNavigatorObserver` in the app.

```
class _MyAppState extends State<MyApp> with WidgetsBindingObserver {


  @override
  void initState() {
    super.initState();
  }


  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(
          title: const Text('APM Example'),
        ),
        body: Center(
          child: MyHome(),
        ),
      ),
      // Adds BuglyNavigatorObserver.
      navigatorObservers: [BuglyNavigatorObserver()],
```

```
      routes: {
        'MainPage': (context) => MainPage(),
        'FirstPage': (context) => FirstPage(),
        'SecondPage': (context) => SecondPage(),
      },
    );
  }
}
```

If you have integrated a hybrid stack framework such as TRouter that takes over the native Navigator, you need to add `BuglyNavigatorObserver` to the hybrid stack framework, as shown below.

```
runApp(MaterialApp(
    home: TRouteContainer(navigatorObservers:
BuglyNavigatorObserver(),),
));
```

> **Note:**
> TRouter version 2.5.7 or later is required; otherwise, some lifecycle events may be lost.

Configuring Launch Monitoring Instrumentation

To track launch performance, you need to set instrumentation points when opening Flutter pages from native code.

- Android:

```
Intent intent = new Intent(MainActivity.this,
TestFlutterActivity.class);
MainActivity.this.startActivity(intent);
FlutterLaunchMonitor.onEnterFlutter();
```

- iOS:

```
[[TFlutterLaunchMonitor sharedInstance] onEnterFlutter];
```

Setting the User ID

If your app cannot set the user ID during SDK initialization, you can call `Bugly#setUserId` to set it.

```
Bugly.setUserId('your_user_id');
```

Setting Log

By default, internal SDK logs are printed using print and are not persisted. You can redirect logs to your custom logging system by setting BuglyOptions.logger.

```
BuglyOptions.logger = MyLogger
```

React Native SDK Integration Guide
SDK Integration Overview

Last updated：2026-05-25 18:01:56

The RUM Pro SDK is technically supported by the Tencent Bugly team and provides monitoring for errors, page visits, and more. This document describes how to integrate the RUM Pro SDK on the React Native platform. After you integrate the SDK and verify that data is reported successfully, you can use the related analysis features in the console.

SDK Overview

- **SDK Version**: 1.0.0-beta.5
- **SDK features**: A professional application quality monitoring tool that provides collection and analysis services for abnormal data, helping developers identify and resolve issues in a timely manner to develop high-quality apps.
- **Service provider**: Tencent Cloud Computing (Beijing) Co., Ltd.
- RUM Pro SDK Compliance Guide.
- RUM Pro SDK Personal Information Protection Rules.

Integration Steps

1. Before you integrate the SDK, please carefully read the RUM Pro SDK Compliance Guide.
2. Complete the SDK Integration.
3. Complete the SDK Initialization. During initialization, the SDK may collect certain user information. Please initialize the SDK only after the user has agreed to the RUM Pro SDK Personal Information Protection Rules. No information is collected before the SDK is initialized.
4. Verify data reporting.

SDK Integration

Last updated：2026-05-25 18:01:57

This article introduces how to integrate the React Native SDK through automatic integration.

Prerequisites

RUM Pro requires React Native SDK version 1.0.0-beta.5 or later.

Automatic Integration

Step 1: Downloading the SDK

Run the following command to download the SDK.

```
npm install bugly-rn-sdk@1.0.0-beta.5
```

Note:

- The React Native SDK is currently in a limited release for validation. It is recommended that you integrate and test it on a small scale first. Proceed to a full rollout only after its quality has been verified to meet expectations.
- The Bugly React Native SDK is integrated into the JS layer of React Native and collects runtime data from that layer.

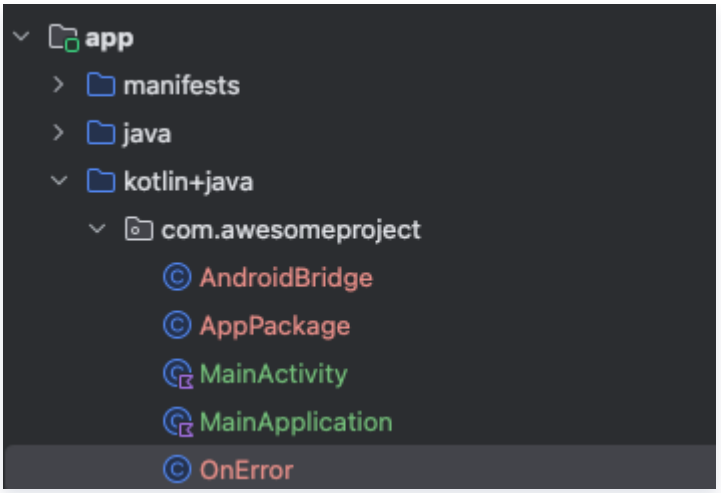
Step 2: Reporting JS Exceptions Using the Native SDK (Optional)

Note:
To report JavaScript exceptions using the native SDK, follow these steps.

Bugly supports cross-platform development frameworks in reporting JS-layer data by calling the Native-layer SDK. React Native itself provides a bridging mechanism for calling Native-layer functions. Based on the official React Native documentation, Bugly has also implemented the corresponding bridging capability. For details, see the Android Native Module Bridging Documentation and the iOS Native Module Bridging Documentation.
The following section uses React Native as an example to describe how to call the iOS and Android SDKs respectively via bridging and report JavaScript exceptions.

Android Integration Steps

1. Create Java files in app/kotlin+java/com.awesomeproject. The new files are AppPackage, AndroidBridge, and OnError.



2. Call the Android SDK in OnError to report error data.

```
package com.awesomeproject;
import android.util.Log;
public class OnError {
    public void sendMsg(String errMsg) {
        // Report error data using the Android SDK
        Map<String, String> extraInfo = new HashMap<>();
        String[] components = errMsg.split("\n");
        String msg = components[0].replace("Error: ", "");
        String[] secondPart = Arrays.copyOfRange(components, 1,
components.length);
        String stack = String.join("\n", secondPart);
        Bugly.postException(8, "test_error", msg , stack, extraInfo);
    }
}
```

3. Declare AndroidBridge in AndroidBridge, and call the OnError class to report error information.

```
package com.awesomeproject;

import androidx.annotation.NonNull;

import com.facebook.react.bridge.ReactApplicationContext;
import com.facebook.react.bridge.ReactContextBaseJavaModule;
import com.facebook.react.bridge.ReactMethod;
```

```
public class AndroidBridge extends ReactContextBaseJavaModule{
    AndroidBridge(ReactApplicationContext context) {
        super(context);
    }
    private OnError onError = new OnError();
    @NonNull
    @Override
    public String getName() {
        return "AndroidBridge";
    }
    @ReactMethod
    public void sendJSError (String errMsg) {
        onError.sendMsg(errMsg);
    }
}
```

4. Register AndroidBridge in React Native.

```
package com.awesomeproject;
import com.facebook.react.ReactPackage;
import com.facebook.react.bridge.NativeModule;
import com.facebook.react.bridge.ReactApplicationContext;
import com.facebook.react.uimanager.ViewManager;

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
public class AppPackage implements ReactPackage{
    @Override
    public List<ViewManager>
createViewManagers(ReactApplicationContext context) {
        return Collections.emptyList();
    }
    @Override
    public List<NativeModule> createNativeModules
(ReactApplicationContext context) {
        List<NativeModule> modules = new ArrayList<>();
        modules.add(new AndroidBridge(context));
```

```
        return modules;
    }
}
```

5. Add AppPackage in MainApplication.

```
package com.awesomeproject
import android.app.Application
import com.facebook.react.PackageList
import com.facebook.react.ReactApplication
import com.facebook.react.ReactHost
import
com.facebook.react.ReactNativeApplicationEntryPoint.loadReactNative
import com.facebook.react.ReactNativeHost
import com.facebook.react.ReactPackage
import
com.facebook.react.defaults.DefaultReactHost.getDefaultReactHost
import com.facebook.react.defaults.DefaultReactNativeHost

class MainApplication : Application(), ReactApplication {

  override val reactNativeHost: ReactNativeHost =
      object : DefaultReactNativeHost(this) {
        override fun getPackages(): List<ReactPackage> =
            PackageList(this).packages.apply {
              add(AppPackage())
            }

        override fun getJSMainModuleName(): String = "index"

        override fun getUseDeveloperSupport(): Boolean =
BuildConfig.DEBUG

        override val isNewArchEnabled: Boolean =
BuildConfig.IS_NEW_ARCHITECTURE_ENABLED
        override val isHermesEnabled: Boolean =
BuildConfig.IS_HERMES_ENABLED
      }
```

```
  override val reactHost: ReactHost
    get() = getDefaultReactHost(applicationContext, reactNativeHost)

  override fun onCreate() {
    super.onCreate()
    loadReactNative(this)
  }
}
```

6. Call the Android SDK from the JS layer to report error data.

```
import { NativeModules } from 'react-native';
const { AndroidBridge } = NativeModules;
ErrorUtils.setGlobalHandler((error, isFatal) => {
    AndroidBridge.sendJSError(error.stack || error);
});
```

iOS Integration Steps

1. Create the OnError.swift, iOSBridge.swift, and IOSBridge.m files in the project root directory.

```
// OnError.swift
class OnError {
  func tran2BuglyError(errMsg: String) -> (category: UInt,
name:String, reason: String, callStack: Array<String>) {
    let components = errMsg.components(separatedBy: "\n")
    let firstPart = components[0].replacingOccurrences(of: "Error: ",
with: "")
    let secondPart = Array(components.dropFirst())
    return (5, "JS Error", firstPart, secondPart)
  }

  public func sendMsg(errMsg: String) -> Void {
    let BuglyError = tran2BuglyError(errMsg: errMsg)
    // Call Bugly IOS to report errors
    BuglyCrashMonitorPlugin.reportException(withCategory:
BuglyError.category, name: BuglyError.name, reason: BuglyError.reason,
callStack: BuglyError.callStack, extraInfo: [:], terminateApp: false)
  }
```

```
}

// IOSBridge.swift
@objc(IOSBridge)
class IOSBridge: NSObject {
  @objc(sendJSError:)
  func sendJSError(errMsg: String) -> Void {
    let onError = OnError()
    onError.sendMsg(errMsg: errMsg)
  }
}

// IOSBridge.m
#import <React/RCTBridgeModule.h>

@interface RCT_EXTERN_MODULE(IOSBridge, NSObject)

RCT_EXTERN_METHOD(sendJSError:(NSString *)errMsg)

@end
```

2. Create a bridging file.

```
// AwesomeProject_Bridging_Header.h
#ifndef AwesomeProject_Bridging_Header_h
#define AwesomeProject_Bridging_Header_h
#import <React/RCTBridgeModule.h>

#endif /* AwesomeProject_Bridging_Header_h */
```

3. Finally, listen for errors in the JS layer and call the iOS SDK to report them.

```
import { NativeModules } from 'react-native';
const { IOSBridge } = NativeModules
ErrorUtils.setGlobalHandler((error, isFatal) => {
    IOSBridge.sendJSError(error.stack || error);
});
```

SDK Initialization

Last updated：2026-05-25 18:01:57

This article describes how to initialize the RUM Pro React Native SDK.

Sample Code

Before initializing, ensure that you have created a product in RUM Pro Creating an Application and obtained the corresponding AppKey and AppID. Then, initialize the SDK using the following code.

```
import Aegis, { BUGLY_SERVER_HOST } from 'bugly-rn-sdk';
// Optional. Used to obtain the device's network information.
import netInfo from '@react-native-community/netinfo';

const aegis = new Aegis({
  id: 'xxxx', // The app id of the integrated product. Required.
  appKey: 'xxxxxxx', // The app key of the integrated product. Required.
  aid: "deviceID", // The device ID. It should be unique for different
devices. Recommended.
  env: "debug", // The log level for sdk console output. "debug" outputs
all logs. The default value is "production", which outputs only error
logs during sdk runtime. Optional.
  serverHost: BUGLY_SERVER_HOST.OA, // The reporting domain. Required.
  netInfo: netInfo, // Optional. Network information can be obtained
after it is passed in.
  plugin:{
    error: true, // Enable error monitoring.
  }
});
```

Note:

- The SDK uses @react-native-community/netinfo to obtain network information. If this third-party library is not available, the returned network category is unknown.
- When aid is not set during aegis initialization, the SDK randomly generates a device ID. The SDK uses @react-native-async-storage/async-storage to persistently store the device ID. If this third-party library is not available, the SDK generates a different device ID each time the application starts.

React Native primarily uses the following third-party libraries to implement page navigation: React Navigation and React Native Navigation. You can enable the SDK's page visit feature by following the instructions below.

React Navigation

Set up the page navigation feature for your React Native application.

```
import { NavigationContainer, useNavigationContainerRef } from '@react-
navigation/native';
import { createNativeStackNavigator } from '@react-navigation/native-
stack';
function App() {
  const Stack = createNativeStackNavigator();
  const containerRef = useNavigationContainerRef();
  return (<NavigationContainer
            ref = {containerRef}
            onReady={
              () => {
                // Enable the page visit feature using
wrapNavigationRef.
                aegis.wrapNavigationRef(containerRef);
              }
            }
            >
  <Stack.Navigator initialRouteName="Home">
      <Stack.Screen name="Home" component={HomeScreen} />
    </Stack.Navigator>
  </NavigationContainer>
  );
}
```

React Native Navigation

Pass in the Navigation from the React Native Navigation library during aegis initialization.

```
import { Navigation } from 'react-native-navigation';
const aegis = new Aegis({
  // The configuration items are the same as those for SDK
initialization.
```

```
  navigation: Navigation,
});
```

Data Reporting Verification

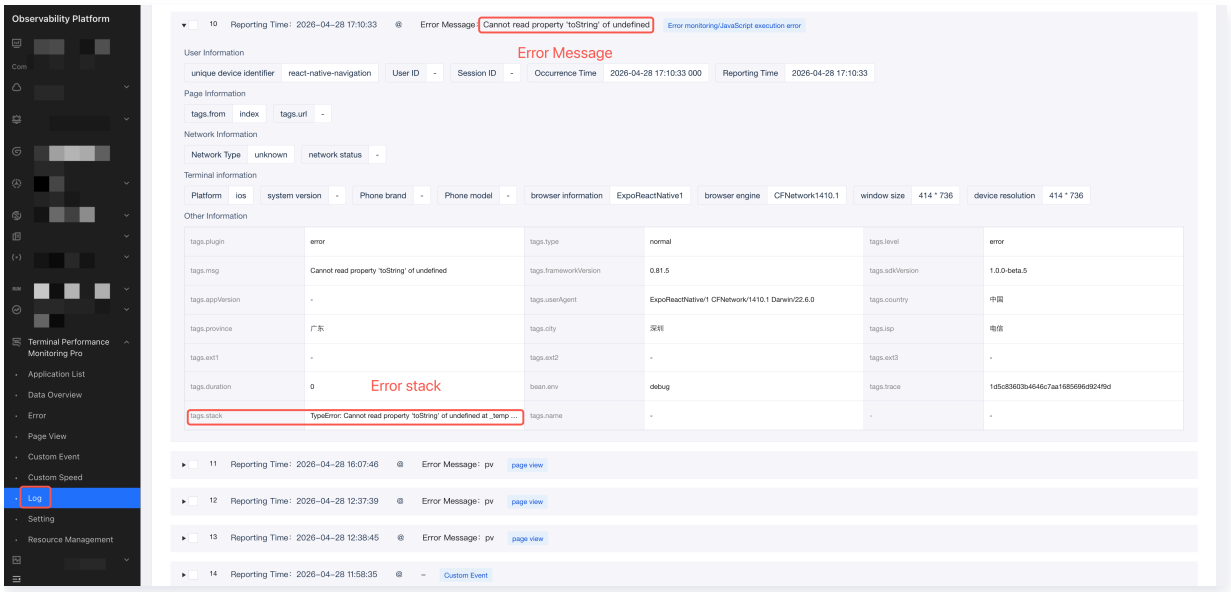
Last updated：2026-05-25 18:01:57

Error Monitoring

The React Native SDK collects JavaScript errors and uncaught Promise exceptions.

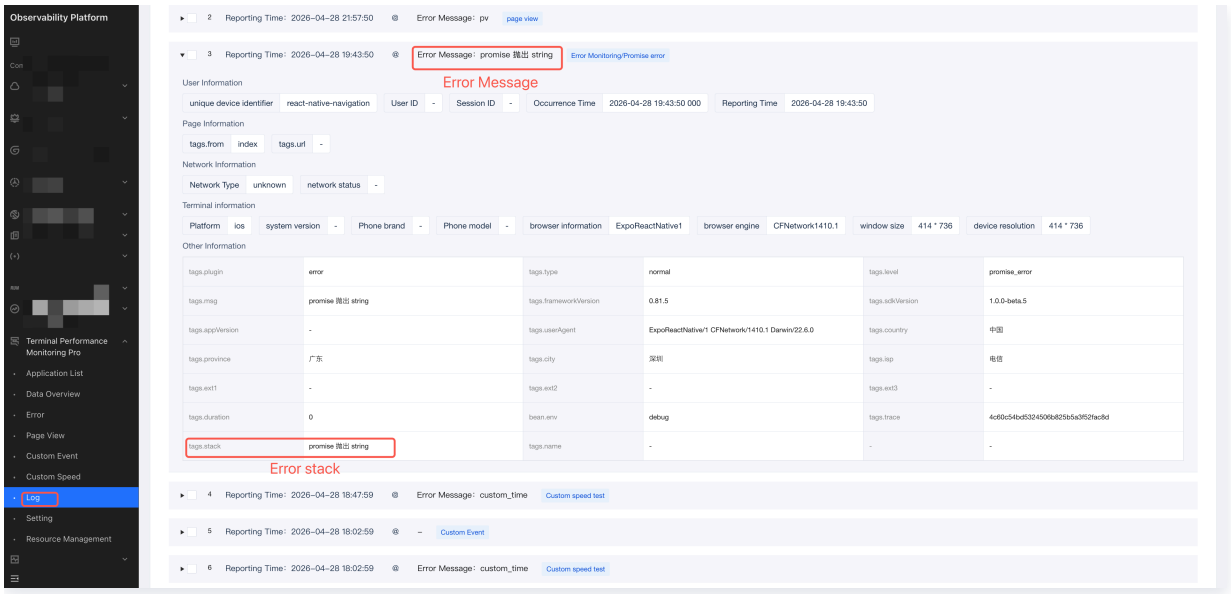
JS Errors

The SDK uses ErrorUtils.setGlobalHandler to listen for JavaScript errors. You can view error information and error stacks on the RUM Pro > Log page.



Promise Exceptions

The SDK uses onUnhandled to listen for uncaught Promise exceptions. You can view the exception information and stack traces on the RUM Pro > Log page.



Page view

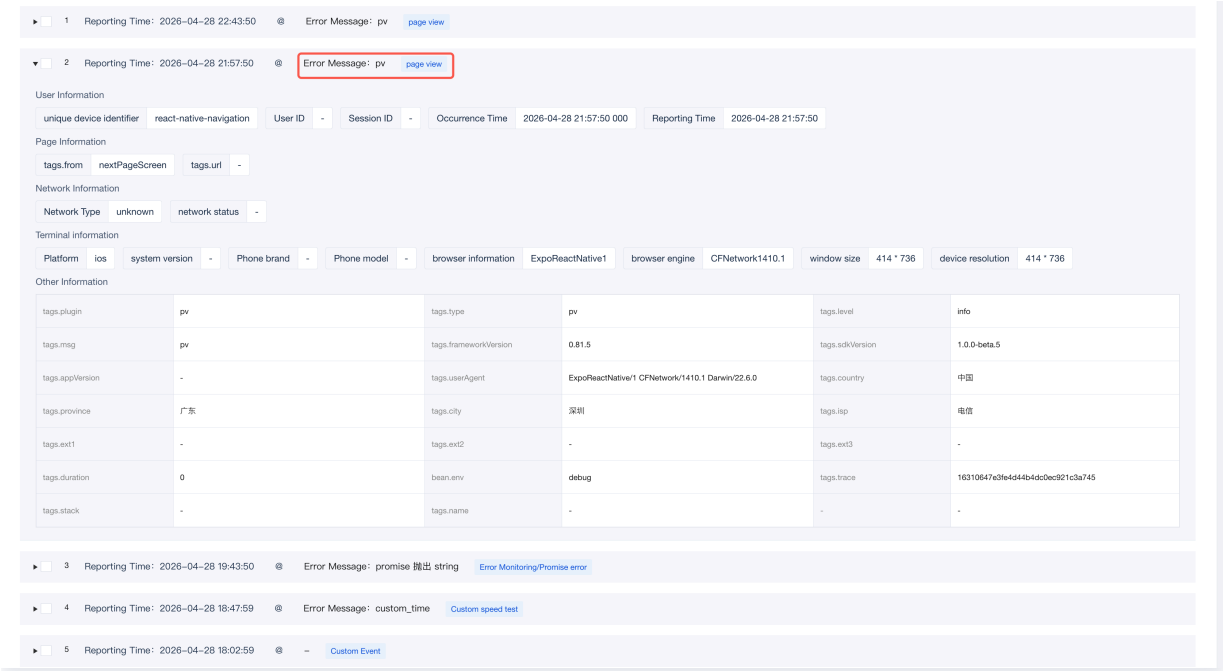
The React Native SDK for RUM Pro implements the page visit feature based on the React Navigation and React Native Navigation libraries. You can view exception information and stack traces on the RUM Pro > Log page.

React Navigation

Use containerRef.addListener("state") to listen for route changes and obtain the page name based on getCurrentRoute().name.

React Native Navigation

Use Navigation.events().registerComponentDidAppearListener to listen for route changes and obtain the page name based on componentName.



Custom Reporting

The React Native SDK provides a rich set of custom reporting APIs for users.

Custom Log Reporting

info,error

These two methods are the primary means provided by the SDK for custom log reporting. Their usage is as follows:

```
Aegis.info('Report a log');
Aegis.error('Report an error log');

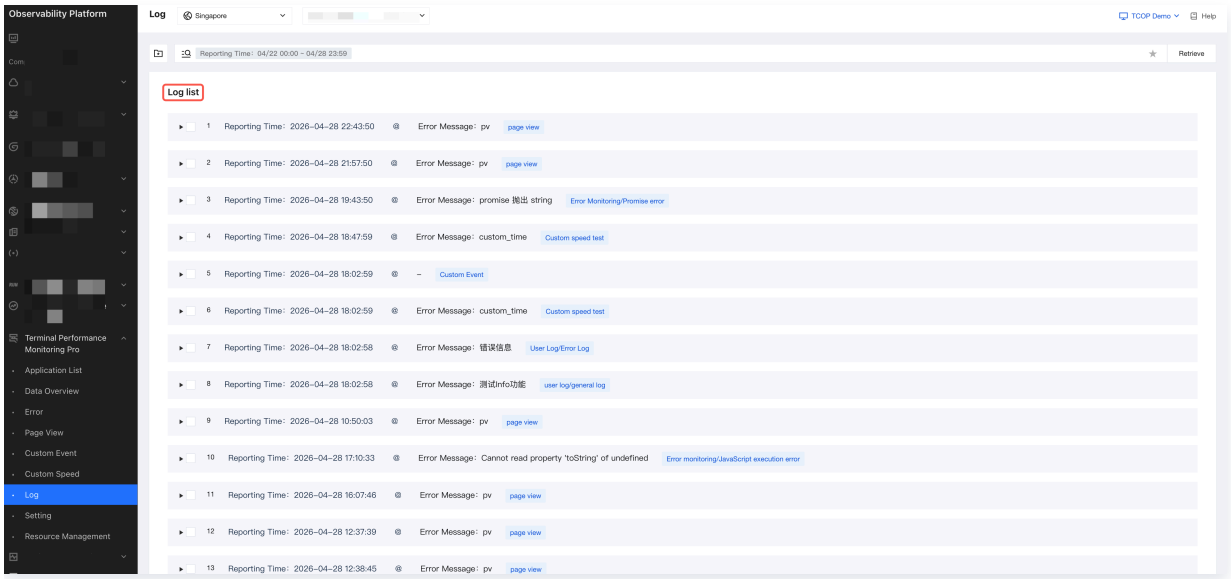
// You can also report logs using the following format.

Aegis.info({
 msg: 'xxx', // Recommended. If not set, the reported message will be
empty.
 [key: string]: any,
});

// Reports an error log (error level: custom_error).
Aegis.error({
 msg: 'Error log message', // Recommended. If not set, the reported
message will be empty.
```

```
  [key: string]: any,
});
```

You can view exception information and exception stacks on the RUM Pro > Log page.



Custom Event Reporting

reportEvent

This method can be used to report custom events. The platform automatically aggregates metrics for reported events, such as PV and platform distribution. The reportEvent method supports two types of reporting parameters:

- String type.

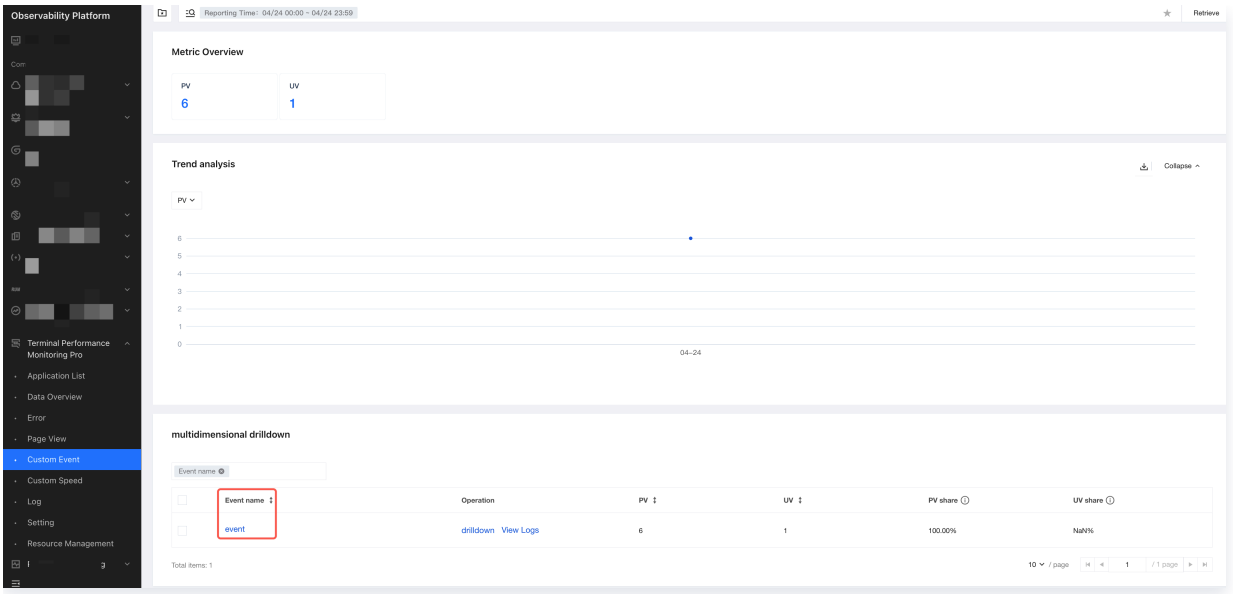
```
aegis.reportEvent('XXX request successful');
```

- Object type. Users can upload custom fields.

```
Aegis.reportEvent({
  name: 'XXX request successful', // Required
  [key: string]: any,
})
```

> **Note:**
> For custom event reporting, the default log type is {level: "info", type: "custom_event"}.

You can view reported information on the Terminal Performance Monitoring Pro > Custom Event page.



Custom Speed Test Reporting

- Custom speed test allows you to report arbitrary values, which are then statistically processed and calculated on the server. Since the server cannot handle dirty data, it is recommended to validate and restrict values at the reporting end to prevent dirty data from affecting the overall results.
- Currently, Aegis only supports numerical values in the range 0 to 60000. If your values exceed this range, it is recommended to transform them appropriately before reporting.
- For high-frequency custom speed test reporting, use reportTime whenever possible. Using time and timeEnd for reporting can lead to reported value overwriting issues. For example, if aegis.time(aaa) is called, and then aegis.time(aaa) is called again before aegis.timeEnd(aaa) is invoked, the reported time will be the timeEnd timestamp minus the timestamp of the second time call.

reportTime

This method can be used to report custom speed tests. Example:

```
// If the 'onload' time is 1s
aegis.reportTime('onload', 1000);
```

Alternatively, if you need to use additional parameters, you can pass custom fields:

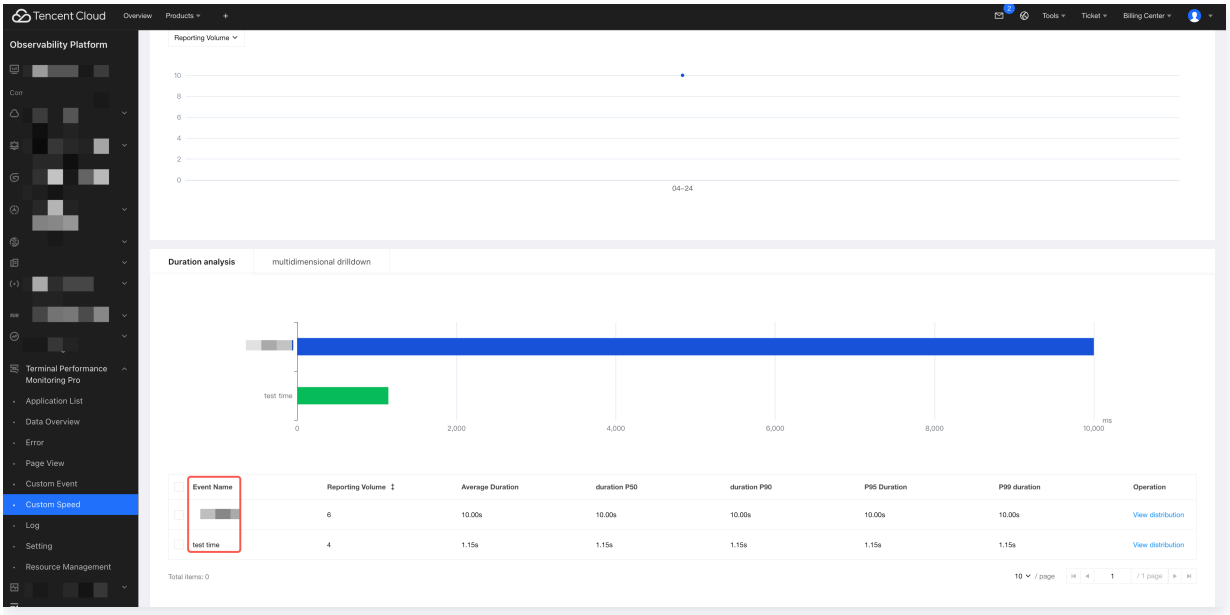
```
aegis.reportTime({
    name: 'onload', // Custom speed test name
    duration: 1000, // Custom speed test duration (0 - 60000)
    [key: string]: any,
});
```

time,timeEnd

These methods are also used for custom speed tests, suitable for measuring and reporting the duration between two points in time. Example:

```
aegis.time('complexOperation');
/**
 * .
 * .
 * After performing complex operations for a long time...
 * .
 * .
 */
aegis.timeEnd('complexOperation'); /* The log has been reported at this point. */
```

You can view reported information on the Terminal Performance Monitoring Pro > Custom Speed page.



Log SDK Integration Guide (Android)
SDK Integration Overview

Last updated：2026-05-25 18:01:57

The Terminal Performance Monitoring Pro (RUM Pro) SDK is powered by the Tencent Bugly team. This article describes how to integrate the RUM Pro log SDK on your platform. After the log SDK is integrated and data reporting is verified, you can use the analysis features in the console.

SDK Overview

- **SDK version**: 0.4.12.1.
- **SDK features**: A professional application quality monitoring tool that provides collection and analysis services for abnormal data, helping developers identify and resolve issues in a timely manner to develop high-quality apps.
- **Service provider**: Tencent Cloud Computing (Beijing) Co., Ltd.
- RUM Pro SDK Compliance Guide.
- RUM Pro SDK Personal Information Protection Rules.

Integration Steps

1. Before you integrate the SDK, please carefully read the RUM Pro SDK Compliance Guide.
2. Complete the SDK integration.
3. Complete the SDK initialization. During initialization, the SDK may collect certain user information. Please initialize the SDK only after the user has agreed to the RUM Pro SDK Personal Information Protection Rules. No information is collected before the SDK is initialized.
4. Verify data reporting.
5. For more information on other SDK features, see the API Description.

SDK Integration

Last updated：2026-05-25 18:01:57

This article describes how to integrate the log SDK using automatic integration.

Prerequisites

Before integration, ensure that you have created an application in Terminal Performance Monitoring Pro (RUM Pro) and obtained the corresponding AppKey and AppID.

Automatic Integration

1. Add the Maven repository address in the project-level build.gradle file.

```
repositories {
    maven { url 'https://repo1.maven.org/maven2/' }
}
```

2. Add dependencies and attribute configurations in the module's build.gradle file.

```
dependencies {
    // Logger (optional)
    // If you need the STL static linking version, replace "logger"
with "logger-static".
    implementation 'com.tencent.tdos-diagnose:logger:0.4.11'
    // Diagnose
    implementation 'com.tencent.tdos-diagnose:diagnose:0.4.11'
}
```

> **Note:**
> The STL dynamic linking version of the Logger module uses libc++_shared. If your project already includes this library, the dynamic linking version will result in a smaller APK size. If your project does not include libc++_shared or has other conflicting STL implementations, you can choose the static linking version instead.

Verifying the Release Build (Obfuscation)

After completing the above steps, verify that the obfuscated APK works properly (such as log printing, command issuing, and active reporting).

Generally, no additional configuration is required, as the platform log AAR already includes the obfuscation rules in proguard.txt. However, if you find that the obfuscated APK does not work properly, your build tools may not support automatic ProGuard configuration. In this case, you can try manually configuring the ProGuard rules to resolve the issue.

```
// Logger
-keep class com.tencent.mars.xlog.** { *; }
// Diagnose
-keep class com.tencent.tddiag.protocol.* { public *; }
-keep class com.tencent.tddiag.upload.UploadTask { public *; }
```

SDK Initialization

Last updated：2026-05-25 18:01:57

This article describes how to initialize the Terminal Performance Monitoring Pro (RUM Pro) log SDK.

Sample Code

Use the Builder pattern to configure and initialize the log system (TDLog) and diagnostic system (TDDiag). Set parameters such as log path, log level, size limits, and diagnostic configurations including AppId, environment, and adapters.

```
// Logger (optional)
TDLogConfig logConfig = new TDLogConfig.Builder(context)
    .setLogPath(logPath) // Log output directory. Write permission must
be obtained in advance.
    .setLogLevel(LogLevel.DEBUG) // Default: VERBOSE
    .setConsoleLog(true) // Default: true
    .setMaxFileSize(byte) // Default: 50 MB
    .setMaxAliveFileSize(byte) // Default: unlimited, minimum: 200
    .setMaxAliveDay(day) // Default: 7 days, minimum: 1 day
    .setPubKey(pubKey) // Public key for log encryption (optional).
Contact the RUM Pro assistant to configure the key pair on the backend.
    .build();
TDLog.initialize(context, logConfig);

// Diagnose
TDDiagConfig diagConfig = new TDDiagConfig.Builder()
    .setAppId(appId) // [Important] App ID obtained from RUM Pro
    .setAppKey(appKey) // [Important] App Key obtained from RUM Pro
    .setEnvironment(TDDiagConfig.ENV_CLOUD) // [Important] Sets the
environment. For Tencent Cloud, use TDDiagConfig.ENV_CLOUD.
    .setLoggerAdapter(TDLog.getLogImpl()) // Integrate with your own log
tools by implementing LoggerAdapter.
    .setDeviceInfoAdapter(deviceInfoAdapter) // Inject privacy
information as required by privacy compliance. Optional; defaults to
reading from android.os.Build.
    .setAppVersion("1.0.0") // Custom app version. Optional; defaults to
reading from PackageInfo.
```

```
    .setTrafficQuota(total, metered) // Sets daily traffic quota.
Optional; unlimited by default. Recommended to set.
    .setUploadCountLimit(limit, period, timeUnit) // Sets automatic
reporting frequency limit. Optional; unlimited by default. Recommended
to set.
    .setImportantLabels(...label) // Sets an allowlist of tags that are
exempt from TrafficQuota and UploadCountLimit.
    .build();
TDDiag.initialize(context, config);
```

After initialization is complete, you can start logging.

```
TDLog.i("tag", "xxxxxx");
TDLog.e("tag", "xxxxxx", e);
```

For more usage of TDLog and TDDiag, see the API Description.

Setting the Device Unique ID

In the Android log SDK, userId is used as an identifier for log retrieval and coloring (named userId for historical reasons). It is recommended to use a unique device ID. You can call this method to set the userId synchronously after initialization or asynchronously after the app obtains the unique identifier.

```
TDDiag.setUserId("uid");
```

Data Reporting Verification

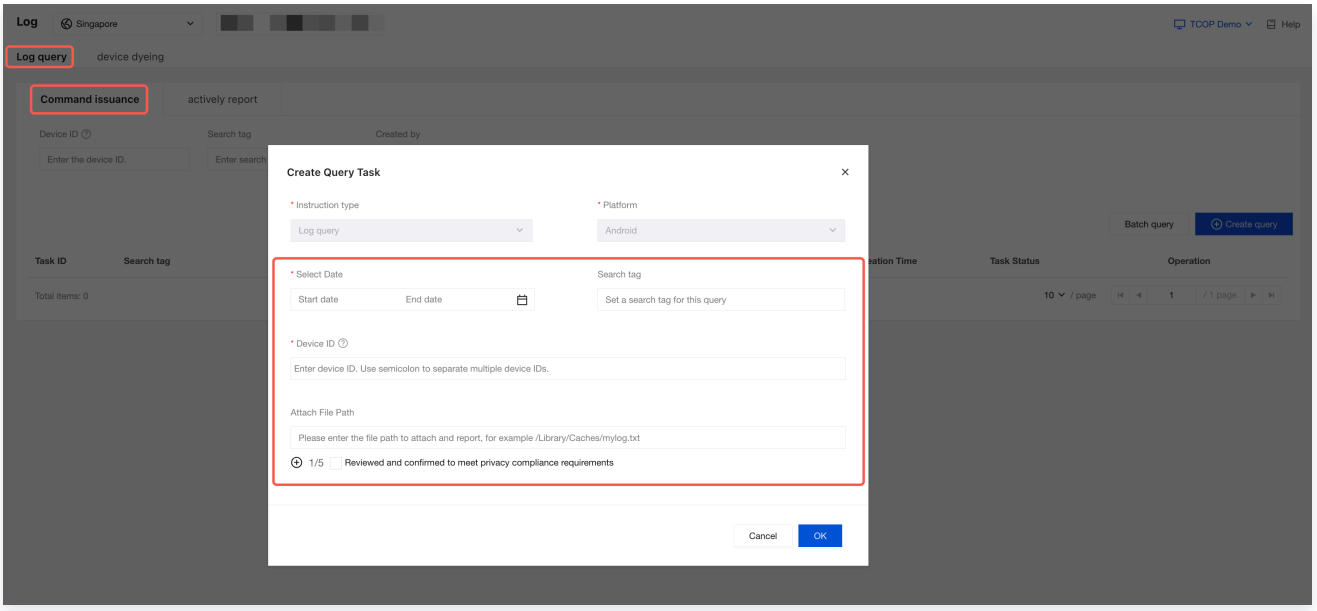
Last updated：2026-05-25 18:01:57

The SDK allows you to call the reporting APIs to upload custom logs, files, and other information. You can use this feature to promptly report local log information such as user feedback or exceptions, helping preserve the issue context.

Issuing a Command

The platform provides the feature to issue log retrieval commands. You can issue commands and query reporting results on the Log query > Command issuance page in the console.

1. On the Issue Command page, click Create query to create a log retrieval command. You need to select the **Select Date** and **Device ID** for the logs to be retrieved. You can add tags and attachment file paths for the retrieval.



2. After the command is issued, wait for the SDK to pull the configurations and upload the log files.

Actively reporting

Limits

To avoid wasting device resources and mobile data due to excessive use, the following proactive reporting limits apply:

- **File size limit**: limits the file size before compression. If this limit is exceeded, only the most recently modified files are retained. Default value: 500 MB for Wi-Fi or 200 MB for 4G. The limit can be modified via (Kotlin) TDDiag.uploadLogs(..., sizeLimit = byte) or (Java) UploadHelper.setSizeLimit(sizeLimit).
- **Frequency limit**: configured via the initialization parameter TDDiagConfig.Builder#setUploadCountLimit.

- **Traffic limit**: configured via the initialization parameter TDDiagConfig.Builder.setTrafficQuota(total, metered).
- **Retry limit**: At startup, up to 5 tasks can be retried. If the retry limit is exceeded, tasks are discarded based on priority. The priority order is **retrieval** > **allowlist tag** > **other proactive reporting**.
- **Circuit breaker**: After 10 consecutive upload failures, proactive reporting is disabled for 6 hours.

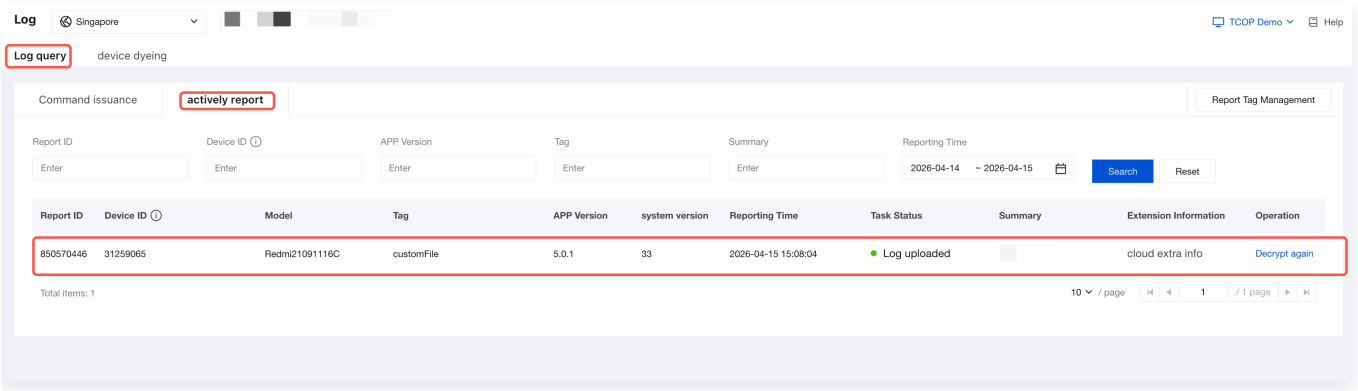
Allowlist

For critical reports, you can configure a tag allowlist via the initialization parameter TDDiagConfig.Builder.setImportantLabels(...labels) to bypass the proactive reporting limits.

> **Note:**
> You can add tags triggered only by user behavior to the allowlist, such as for user feedback, to prevent excessive usage. It is recommended to enforce code review (CR) for the code that configures the tag allowlist.

Checking Data Reporting

After the logs are uploaded, you can view the reported logs on the Log > Log query > actively reporting page in the console.



Example

- **Example 1**: uploading logs from the last half hour.

```
val now = System.currentTimeMillis() / 1000
TDDiag.upload(new UploadParam("recentLog")
              .setTimestamp(now - 1800, now)
              .setListener(uploadListener, true));
```

- **Example 2**: uploading a custom file.

```
TDDiag.upload(new UploadParam("customFile")
              .setExtraPaths(listOf(path))
              .setExtraInfo(msg, null));
```

- **Example 3**: creating a task to upload logs from the last 5 minutes when a crash occurs, which are uploaded upon the next startup.

```
long now = System.currentTimeMillis() / 1000;
TDDiag.upload(new UploadParam("crash")
              .setTimestamp(now - 300, now)
              .setSaveSync(true));
```

Note:
For more information on how to use the log SDK, see the API Description.

API Description

Last updated：2026-05-25 18:01:58

This article describes the feature APIs of the Terminal Performance Monitoring Pro (RUM Pro) log SDK to facilitate flexible and in-depth integration.

Diagnose

Diagnose is the diagnostic module of the application. It is used to manage the configuration, collection, and reporting of diagnostic data. Below are the detailed API descriptions.

TDDiag

The TDDiag API provides features, such as initialization configuration, user ID configuration, receiving push requests, proactive log reporting, and refreshing configuration, for managing and reporting diagnostic logs of the application.

```
object TDDiag {

    /**
     * Initialization.
     *
     * @param context
     * @param config
     */
    fun initialize(context: Context, config: TDDiagConfig)

    /**
     * Initialization.
     *
     * @param context
     * @param config
     * @param host When multiple processes are initialized, this should
be set to true for only one process. By default, this process is the
main process and is used to pull configurations and upload logs.
     */
    fun initialize(context: Context, config: TDDiagConfig, host:
Boolean)

    /**
```

```
     * Configures userId.
     */
    fun setUserId(id: String)

    /**
     * This is used to receive push requests.
     */
    fun onPush(data: String)

    /**
     * Proactively reports logs.
     *
     * @param label Tag. This is required and cannot exceed 64 bytes.
     * @param fileList List of file paths.
     * @param summary Summary. This cannot exceed 256 bytes and can be
searched.
     * @param extraInfo Additional information.
     * @param listener UI callback.
     * @param saveSync Indicates whether to only synchronously save the
task and upload data on the next startup.
     * @see upload
     */
    @Deprecated("Use TDDiag.upload(param) instead")
    @JvmStatic
    fun uploadFiles(
        label: String,
        fileList: List<File>,
        summary: String? = null,
        extraInfo: String? = null,
        listener: UploadListener? = null,
        saveSync: Boolean = false
    )

    /**
     * Proactively reports logs.
     *
     * @param label Tag. This is required and cannot exceed 64 bytes.
     * @param pathList List of file paths.
     * @param summary Summary. This cannot exceed 256 bytes and can be
searched.
```

```
     * @param extraInfo Additional information.
     * @param listener UI callback.
     * @param saveSync Indicates whether to only synchronously save the
task and upload data on the next startup.
     * @see upload
     */
    @Deprecated("Use TDDiag.upload(param) instead")
    @JvmStatic
    fun uploadPaths(
        label: String,
        pathList: List<String>,
        summary: String? = null,
        extraInfo: String? = null,
        listener: UploadListener? = null,
        saveSync: Boolean = false
    )

    /**
     * Proactively reports logs.
     *
     * @param label Tag. This is required and cannot exceed 64 bytes.
     * @param startTimestamp Starting timestamp (unit: seconds). This
value is included.
     * @param endTimestamp Ending timestamp (unit: seconds). This value
is not included.
     * @param extraPathList List of file paths.
     * @param summary Summary. This cannot exceed 256 bytes and can be
searched.
     * @param extraInfo Additional information.
     * @param listener UI callback.
     * @param saveSync Indicates whether to only synchronously save the
task and upload data on the next startup.
     * @param sizeLimit File size limit (before compression). This is
effective when the value is less than the [limits](README.md). No limit
is applied by default.
     * @see upload
     */
    @Deprecated("Use TDDiag.upload(param) instead")
    @JvmStatic
    fun uploadLogs(
```

```
        label: String,
        startTimestamp: Long,
        endTimestamp: Long,
        extraPathList: List<String>? = null,
        summary: String? = null,
        extraInfo: String? = null,
        listener: UploadListener? = null,
        saveSync: Boolean = false,
        sizeLimit: Long = Long.MAX_VALUE
    )

    /**
     * Proactively reports logs.
     */
    @JvmStatic
    fun upload(param: UploadParam)

    /**
     * Proactively refreshes the configuration.
     *
     * @param force Indicates whether to force refresh without checking
the pull interval.
     */
    fun syncConfig(force: Boolean = false): Boolean
}
```

TDDiagConfig.Builder

The TDDiagConfig.Builder API is used to configure the parameters required by TDDiag (such as appId, appKey, log adapter, and traffic quota) through chained methods, ultimately building a TDDiagConfig instance.

```
class TDDiagConfig.Builder {
    /**
     * Configures the appId applied on the [TDS - Debugging and
Diagnostics Platform](https://diagnose.woa.com/).
     */
    fun setAppId(appId: String): Builder

    /**
```

```
     * Configures the appKey applied on the [TDS - Debugging and
Diagnostics Platform](https://diagnose.woa.com/).
     */
    fun setAppKey(appKey: String): Builder

    /**
     * Configures LoggerAdapter for the logging module.
     */
    fun setLoggerAdapter(loggerAdapter: LoggerAdapter): Builder

    /**
     * Inject compliant privacy information (optional; this information
is read from android.os.Build by default).
     */
    fun setDeviceInfoAdapter(deviceInfoAdapter: DeviceInfoAdapter):
Builder

    /**
     * Custom app version (optional; this information is read from
PackageInfo by default).
     */
    fun setAppVersion(appVersion: String): Builder

    /**
     * Configures the tag allowlist.
     */
    fun setImportantLabels(vararg labels: String): Builder

    /**
     * Configures the daily traffic quota (optional; unlimited by
default).
     *
     * @param total Total traffic (unit: bytes). The value must be
greater than or equal to [metered].
     * @param metered 4G traffic (unit: bytes). The value must be
greater than 0.
     */
    fun setTrafficQuota(total: Long, metered: Long): Builder

    /**
```

```
     * Configures the automatic reporting frequency limit (optional;
unlimited by default).
     *
     * @param limit Frequency limit.
     * @param period Period.
     * @param timeUnit Period unit.
     */
    fun setUploadCountLimit(limit: Long, period: Long, timeUnit:
TimeUnit): Builder

    /**
     * Configures the environment (optional).
     *
     * @param env Default value: [TDDiagConfig.ENV_NORMAL]. Value for
global edition: [TDDiagConfig.ENV_OVERSEAS].
     */
    fun setEnvironment(env: String): Builder

    /**
     * Generates [TDDiagConfig].
     */
    fun build(): TDDiagConfig

}
```

UploadParam

The UploadParam class is a parameter configuration class for proactive log uploads. You can use its constructor and a series of methods to configure upload-related parameters such as tags, log time range, attached files, additional information, and callbacks.

```
/**
 * Proactive upload parameter class.
 */
class UploadParam {
    /**
     * @param label Tag. This is required and cannot exceed 64 bytes.
     */
    constructor(label: String)
```

```
    /**
     * Configures the log range.
     *
     * @param startTimestamp Starting timestamp (unit: seconds). This
value is included.
     * @param endTimestamp Ending timestamp (unit: seconds). This value
is not included.
     */
    fun setTimestamp(startTimestamp: Long, endTimestamp: Long):
UploadParam

    /**
     * Configures the list of attached files (or folders).
     */
    fun setExtraPaths(pathList: List<String>): UploadParam

    /**
     * Configures the list of attached files (or folders).
     */
    fun setExtraFiles(fileList: List<File>): UploadParam

    /**
     * Configures additional information.
     *
     * @param summary Summary. This cannot exceed 256 bytes and can be
searched.
     * @param extraInfo Additional information.
     */
    fun setExtraInfo(summary: String?, extraInfo: String?): UploadParam

    /**
     * Configures the UI callback.
     *
     * @param listener UI callback.
     * @param disableAsyncRetry Indicates whether to disable retry. The
default value is false. Set it to true to customize the retry method at
the upper layer.
     */
    fun setListener(listener: UploadListener?, disableAsyncRetry:
Boolean): UploadParam
```

```
    /**
     * Configures whether to only synchronously save the task and upload
data on the next startup.
     */
    fun setSaveSync(saveSync: Boolean = true): UploadParam

    /**
     * Configures the file size limit (before compression). This is
effective when the value is less than the [limits](README.md).
     */
    fun setSizeLimit(sizeLimit: Long): UploadParam

    /**
     * Configures whether to collect log cache files. Cache files are
not collected by default.
     */
    fun setIncludeCache(includeCache: Boolean = true): UploadParam

    /**
     * Additional search information for platform interoperability
(cross-system search).
     *
     * @param queryKey Key used to search logs. The value cannot exceed
128 bytes.
     * @param source Caller source. The value cannot exceed 128 bytes.
     */
    fun appendExtQueryInfo(queryKey: String, source: String):
UploadParam
}
```

LoggerAdapter

LoggerAdapter is a log component adapter API that provides features to configure the logging level, obtain log files within a specified time range, force log writes, print the SDK logs, and obtain the public key for log encryption, for adapting log-related operations.

```
/**
 * Log component adapter API.
 */
```

```
public interface LoggerAdapter {
    /**
     * Configures the color level.
     */
    void setColorLevel(@LogLevel int level);

    /**
     * Obtains the log files.
     *
     * @param startTimestamp Starting timestamp (unit: seconds). This
value is included.
     * @param endTimestamp Ending timestamp (unit: seconds). This value
is not included.
     * @return List of log files.
     */
    @Nullable
    List<File> getLogFiles(long startTimestamp, long endTimestamp);

    /**
     * Obtains the log files.
     *
     * @param startTimestamp Starting timestamp (unit: seconds). This
value is included.
     * @param endTimestamp Ending timestamp (unit: seconds). This value
is not included.
     * @param includeCache Indicates whether to collect cache files.
     * @return List of log files.
     */
    @Nullable
    default List<File> getLogFiles(long startTimestamp, long
endTimestamp, boolean includeCache) {
        return getLogFiles(startTimestamp, endTimestamp);
    }

    /**
     * Forces to write logs to files.
     */
    void flushLog();

    /**
```

```
     * Prints the SDK logs.
     */
    void printDiagnoseLog(@NotNull String tag, @NotNull String msg,
@Nullable Throwable tr);

    /**
     * Obtains the log encryption public key (optional; the default
value is null, which indicates that encryption is not enabled).
     */
    @Nullable
    default String getPubKey() {
        return null;
    }

}
```

DeviceInfoAdapter

DeviceInfoAdapter is a privacy information adapter API used to provide device brand (corresponding to Build.BRAND) and device model (corresponding to Build.MODEL) information required for privacy compliance.

```
/**
 * Privacy information adapter API.
 */
interface DeviceInfoAdapter {
    /**
     * {@link android.os.Build#BRAND}
     */
    fun getBrand(): String

    /**
     * {@link android.os.Build#MODEL}
     */
    fun getModel(): String
}
```

UploadListener

UploadListener is a callback interface for the log upload process, which triggers the corresponding callback when the upload starts, progress updates, upload succeeds, or upload fails (returning the specific failure reason) to provide information about the upload status.

```
interface UploadListener {
    /**
     * Upload starts.
     */
    fun onStart()

    /**
     * Progress updates. This callback may trigger 0 or multiple times.
     *
     * @param percent 0-100.
     */
    fun onProgress(percent: Int)

    /**
     * Upload succeeds.
     */
    fun onSuccess()

    /**
     * Upload fails.
     *
     * @param reason Failure reason:
     * * -4: Local restriction policy, including frequency limit,
traffic limit, and circuit breaker.
     * * -3: Failure to synchronize the upload URL to the backend,
usually due to network fluctuations.
     * * -2: Tag sampling limit configured on the console.
     * * -1: Repetitive task.
     * * 1: File upload network error.
     * * 2: Compressed log file size exceeds the threshold.
     * * 3: Compression IO error, usually due to insufficient storage.
     * * 4: During file upload, the Cloud Object Storage (COS) backend
returns an HTTP error.
     * * 5: No local logs.
     */
```

```
    fun onFailure(@UploadLogFailReasonType reason: Int)
```

Logger

Logger is the log component, which is responsible for standardized output, storage, and management of logs. Below are the detailed API descriptions.

TDLog

The TDLog class provides features, such as log system initialization, sub-instance creation, printing logs at different levels (such as VERBOSE and DEBUG), log flushing, and closing and obtaining the default log adapter instance, for log management and output.

```
public class TDLog {

    /**
     * Initialization.
     */
    public static void initialize(Context context, TDLogConfig config)

    /**
     * Creates a sub-instance.
     */
    public static ILogInstance getSubInstance(String category, boolean
consoleLog, int maxAliveDay, long maxAliveFileSize)

    /**
     * Obtains the default {@link LoggerAdapter} instance.
     */
    public static ITDLog getLogImpl()

    /**
     * Prints {@link LogLevel#VERBOSE} logs.
     */
    public static void v(String tag, String message)

    /**
     * Prints {@link LogLevel#VERBOSE} logs.
     */
```

```
    public static void v(String tag, String message, Throwable
throwable)

    /**
     * Prints {@link LogLevel#DEBUG} logs.
     */
    public static void d(String tag, String message)

    /**
     * Prints {@link LogLevel#DEBUG} logs.
     */
    public static void d(String tag, String message, Throwable
throwable)

    /**
     * Prints {@link LogLevel#INFO} logs.
     */
    public static void i(String tag, String message)

    /**
     * Prints {@link LogLevel#INFO} logs.
     */
    public static void i(String tag, String message, Throwable
throwable)

    /**
     * Prints {@link LogLevel#WARN} logs.
     */
    public static void w(String tag, String message)

    /**
     * Prints {@link LogLevel#WARN} logs.
     */
    public static void w(String tag, String message, Throwable
throwable)

    /**
     * Prints {@link LogLevel#ERROR} logs.
     */
    public static void e(String tag, String message)
```

```
    /**
     * Prints {@link LogLevel#ERROR} logs.
     */
    public static void e(String tag, String message, Throwable
throwable)

    /**
     * Prints {@link TDLogInfo} logs.
     */
    public static void log(TDLogInfo info)

    /**
     * Forces to write buffer logs to files.
     */
    public static void flushLog()

    /**
     * Closes the log instance.
     */
    public static void closeLog()
}
```

TDLogConfig.Build

TDLogConfig.Builder is the configuration builder for the logging component. You can use chained methods to configure parameters such as the log level, storage path, encryption public key, file size, retention days, and whether to output to Logcat, ultimately generating a TDLogConfig instance.

```
/**
 * Log component configuration builder.
 */
public static class TDLogConfig.Build {

    /**
     * constructor
     */
    public Builder(Context context)

    /**
```

```
     * Configures the lowest log level.
     */
    public Builder setLogLevel(@LogLevel int logLevel)

    /**
     * Configures whether to output to Logcat.
     */
    public Builder setConsoleLog(boolean consoleLog)

    /**
     * Configures the log file storage path.
     */
    public Builder setLogPath(String logPath)

    /**
     * Configures the public key for log file encryption.
     */
    public Builder setPubKey(String key)

    /**
     * Configures the maximum size of a single log file.
     */
    public Builder setMaxFileSize(long maxFileSize)

    /**
     * Configures the retention days for log files.
     */
    public Builder setMaxAliveDay(int maxAliveDay)

    /**
     * Configures the maximum total size of log files.
     */
    public Builder setMaxAliveFileSize(long maxAliveFileSize)

    /**
     * Creates a {@link TDLogConfig} instance.
     */
    public TDLogConfig build()
}
```

TDLogInfo

The TDLogInfo class is a class that holds log information. It is used to encapsulate detailed log data such as tags (including primary, secondary, and tertiary tags), level, thread information, content, exception object, and timestamp.

```
public class TDLogInfo {
    /**
     * Tag
     */
    public String tag;
    /**
     * Secondary tag.
     */
    public String subTag;
    /**
     * Tertiary tag.
     */
    public String thirdTag;
    /**
     * Log level.
     */
    @LogLevel
    public int level;
    /**
     * Thread ID.
     */
    public long tid;
    /**
     * Thread name.
     */
    public String tName;
    /**
     * Log content.
     */
    public String message;
    /**
     * Throwable object.
     */
    public Throwable throwable;
```

```
    /**
     * Time, in milliseconds. The default value is 0, which indicates
unspecified.
     */
    public long timeMillis;
}
```

ILogInstance

ILogInstance is a log instance API that provides features to print logs (including regular logs and logs with custom information), force writing logs to files, and close the current log instance.

```
/**
 * Log instance API.
 */
public interface ILogInstance {

    /**
     * Prints logs.
     */
    void log(String tag, @LogLevel int level, String message, @Nullable
Throwable throwable);

    /**
     * Prints logs with custom information.
     */
    void log(TDLogInfo info);

    /**
     * Forces to write logs to files.
     */
    void flushLog();

    /**
     * Closes the current log instance.
     */
    void closeLog();
}
```

Push Support (Optional)

The log SDK pulls configurations during initialization by default and checks for log retrieval commands. With push features integrated, the SDK can receive log retrieval commands faster, reducing waiting time.
You need to integrate your own push channels with the platform (the platform does not provide push capabilities). After a log retrieval or log coloring push request is received, you can call the following API to complete log retrieval or coloring.

```
TDDiag.onPush("{}");
```

Proactive Command Issuance Checking (Optional)

Under normal circumstances, setting the userId during startup and integrating push requests can ensure excellent real-time performance for the issuance of retrieval command configurations. If the app requires additional opportunities to check for command configuration issuance (such as when the app comes to the foreground), you can call this API at the appropriate time.

```
TDDiag.syncConfig(false); // False indicates a non-forced refresh.
```

The SDK has a frequency limit for the proactive command configuration issuance checks, and the limitation policy is dynamically determined by the backend based on load.

> **Note:**
> It is not recommended to perform forced refresh in any automatically triggered scenario without user interaction, to avoid imposing uncontrollable load pressure on the backend.

Log SDK Integration Guide (iOS)
SDK Integration Overview

Last updated：2026-05-25 18:01:58

The Terminal Performance Monitoring Pro (RUM Pro) SDK is powered by the Tencent Bugly team. This article describes how to integrate the RUM Pro log SDK on your platform. After the log SDK is integrated and data reporting is verified, you can use the analysis features in the console.

SDK Overview

- **SDK version**: 2.8.1.9.
- **SDK features**: A professional application quality monitoring tool that provides collection and analysis services for abnormal data, helping developers identify and resolve issues in a timely manner to develop high-quality apps.
- **Service provider**: Tencent Cloud Computing (Beijing) Co., Ltd.
- RUM Pro SDK Compliance Guide.
- RUM Pro SDK Personal Information Protection Rules.

Integration Steps

1. Before you integrate the SDK, please carefully read the RUM Pro SDK Compliance Guide.
2. Complete the SDK integration.
3. Complete the SDK initialization. During initialization, the SDK may collect certain user information. Please initialize the SDK only after the user has agreed to the RUM Pro SDK Personal Information Protection Rules. No information is collected before the SDK is initialized.
4. Verify data reporting.
5. For more information on other SDK features, see the API Description.

SDK Integration

Last updated：2026-05-25 18:01:58

This article describes how to integrate the iOS log SDK.

Prerequisites

- Before integration, ensure that you have created an application in Terminal Performance Monitoring Pro (RUM Pro) and obtained the corresponding AppKey and AppID.
- iOS Requirements: iOS 9.0 or later.

Importing TDOS_Diagnose.framework

The TDOS_Diagnose SDK supports integration via CocoaPods. Simply add the following line to the Podfile.

```
// Adds Pod dependency.
pod 'TDOS_Diagnose', '0.8.2.1'
```

Importing TDOS_Diagnose will include all of its modules by default. The main modules are:

- TDOS_Diagnose/Core: Core module, which provides log upload and command polling capabilities.
- TDOS_Diagnose/DefaultLogger: Default log printing module implementation, which depends on mars.framework.
- TDOS_Diagnose/DefaultStorageImpl: Default KV storage implementation, which depends on MMKV.
- TDOS_Diagnose/DefaultFilePacker: Default file compression implementation, which depends on SSZipArchive.

Note:
Except for the Core module, DefaultLogger, DefaultStorageImpl, and DefaultFilePacker can be replaced with other modules based on your needs. For details, see SDK Initialization.

SDK Initialization

Last updated：2026-05-25 18:01:58

This article describes how to initialize the Terminal Performance Monitoring Pro (RUM Pro) log SDK.

Configuring and Starting the SDK

Since the log printing, KV storage, and file compression modules in the SDK are replaceable, you need to configure the corresponding dependencies before initializing the SDK.

1. The SDK manages module dependencies through TDIAGDepends. A TDIAGDepends instance requires instances that implement the following protocols: id<TDLoggingIMPProtocol>, id<RAFTKVStorageFactoryProtocol>, and id<TDLogFilePackerProtocol>. For example:

```
#import <TDOS_Diagnose/TDOS_Diagnose.h>  // Log SDK module

#import <TDOS_Diagnose/TDMMKVFactoryImpl.h> // MMKV module
#import <TDOS_Diagnose/TDOSLoggerProxy.h> // Log printing module
#import <TDOS_Diagnose/TDLogFilePackerImp.h> // File compression
module



// Uses the SDK's default modules.
TDOSLoggerProxy *loggerProxy = [TDOSLoggerProxy defaultProxy];
TDMMKVFactory *kvFactory = [TDMMKVFactoryImpl sharedInstance];
TDLogFilePackerImp *filePacker = [[TDLogFilePackerImp alloc] init];

// Creates a TDIAGDepends instance.
TDIAGDepends *depends = [TDIAGDepends dependsWithLogImp:loggerProxy
                                            kvFactoryImp:kvFactory
                                          andFilePackerImp:filePacker];
```

   In the above example, TDOSLoggerProxy, TDMMKVFactoryImpl, and TDLogFilePackerImp are default instances provided by the SDK that implement the corresponding protocols. After creating TDIAGDepends, you can proceed with SDK initialization.
2. Provide the AppID and AppKey. Here, use the AppID and AppKey provided during platform product registration.

```
// appid & appKey, generated by the platform
NSString *appID = BUGLY_APPID;
```

```
NSString *appKey = BUGLY_APPKEY;
```

3. You need to provide a dataSource object that conforms to the TDLogSDKDataSource protocol. This object supplies necessary data to the SDK. For details, refer to the definition in TDLogSDKDataSource. The `- (NSString *)guidForTDLog;` method must be implemented. This method returns a unique user ID, which is used when the platform issues commands.

```
// Creates a log retrieval module configuration instance and
initializes it.
TDLogSDKConfig *config = [TDLogSDKConfig configWithAppId:appID
                                                  appKey:appKey
                                              dataSource:self
                                                 depends:depends];
```

> **Note:**
> By default, the log module uses an internal domain. For RumPro users, you need to specify the corresponding domain type in TDLogSDKConfig:
>
> ```
> /// Sets the service domain type.
> @property (nonatomic,assign)TDLogServerHostType serverHostType;
>
> /// Sets the reporting domain.
> https://cloud.bugly.qq.com
>
> /// Customizes the server domain.
> @property(nonatomic,strong)NSString *customServerHost;
> ```

4. Based on your business requirements, you can configure frequency and traffic control policies to prevent excessive network resource consumption from frequent log reporting.

```
// Enables frequency control (default unlimited). Pass empty to use
the SDK's default policy: 2 times/5min, or pass a custom policy. When
restricted, the automatic reporting API callback will indicate
failure.
// Example: Allow at most 3 reports within 10 minutes (using token
bucket algorithm).
```

```
//    TDLogFrequencyControlStrategy *strategy =
[TDLogFrequencyControlStrategy new];
//    strategy.times = 3;
//    strategy.timeInterval = 10 * 60;
// Note: This API requires TDLogSDKDataSource to implement the
whitelistForAutoUploadTags protocol, which provides an allowlist of
tags for automatic reporting. This ensures that essential reports are
not blocked.
[config setFrequencyLimitStatusForAutoUpload:YES
                  withCustomControlStrategy:nil]; // Recommended
setting to prevent frequent automatic reporting
// Enables traffic control (default: unlimited). Pass 0 to disable
reporting. Pass a negative value for no limit. When limited, the
automatic reporting API callback will indicate failure.
// Note: This API requires TDLogSDKDataSource to implement the
whitelistForAutoUploadTags protocol, which provides an allowlist of
tags for automatic reporting. This ensures that essential reports are
not blocked.
[config setTrafficQuota24hLimitForAutoUpload:(200 * 1024 * 1024)
                                    xgQuota:(50 * 1024 * 1024)]; //
Optional. Sets the traffic limit for automatic reporting.
```

5. Finally, call the corresponding method to start the SDK and complete the initialization process.

```
[[TDLogSDK sharedInstance] startWithConfig:config];
```

Configuring the Local Log Printing Module

TDOSLoggerProxy is a wrapper that implements the TDLoggingIMPProtocol interface provided by the TDOSLogger module in TDOS_Diagnose. To use the TDOSLogger module, you need to initialize it. The steps are as follows:

1. Initialize the log printing module.

```
// Initializes the log printing module.
TDOSLoggerConfig *loggerConfig = [TDOSLoggerConfig defaultConfig];
TDOSLogger *logger = [[TDOSLogger alloc] initWithConfig:loggerConfig];
```

2. Set the instantiated logger object to the TDOSLoggerProxy instance created earlier, so that the TDOSLogger module can function properly.

```
TDOSLoggerProxy *loggerProxy = [TDOSLoggerProxy defaultProxy];
[loggerProxy setLogger:logger];
```

Note:

For Extension project integration: This SDK supports iOS Extensions. If you are using the default MMKV dependency module, add the following script to your project's Podfile:

```
post_install do |installer|
  installer.pods_project.targets.each do |target|
    if target.name == "MMKV"
      target.build_configurations.each do |config|
        config.build_settings['GCC_PREPROCESSOR_DEFINITIONS'] ||=
['$(inherited)', 'MMKV_IOS_EXTENSION']
      end
    end
  end
end
```

For more features provided by TDOSLogger, see the API Description.

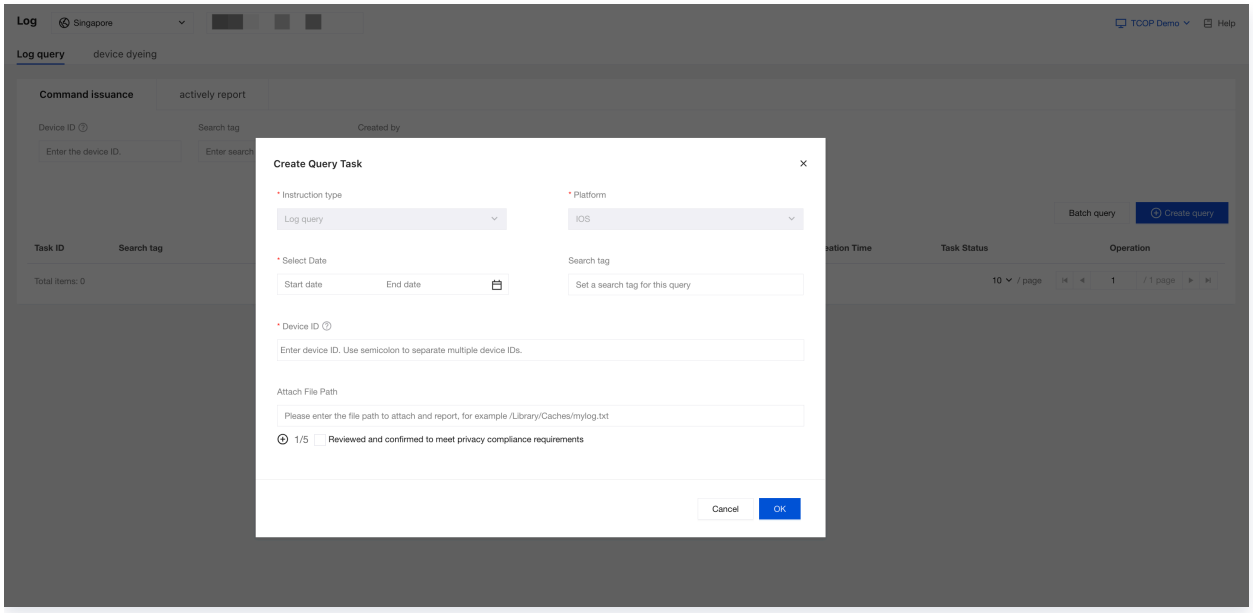
Data Reporting Verification

Last updated：2026-05-25 18:01:58

The SDK allows you to call the reporting APIs to upload custom logs, files, and other information. You can use this feature to promptly report local log information such as user feedback or exceptions, helping preserve the issue context.

Issuing a Command

The platform provides the feature to issue log retrieval commands. You can issue commands and query reporting results on the Terminal Performance Monitoring Pro > Log > Log query page in the console. Create a new log retrieval command in Create query:



You need to enter a valid **Device ID** value. This ID should match the value returned by the – (NSString *)guidForTDLog; method implemented in TDLogSDKDataSource during SDK initialization.
The TDOS_DiagnoseSDK proactively checks for pending log retrieval commands at regular intervals (e.g., on app startup or when returning to the foreground). If a command is found for the device, the SDK will report the corresponding logs.

Actively reporting

Since active reporting is automatically triggered by the SDK, its frequency and timing are not directly controllable. To ensure safe and efficient reporting, and to prevent issues such as excessive data usage caused by high-frequency uploads, the TDOS_Diagnose SDK provides the following 4 security mechanisms:

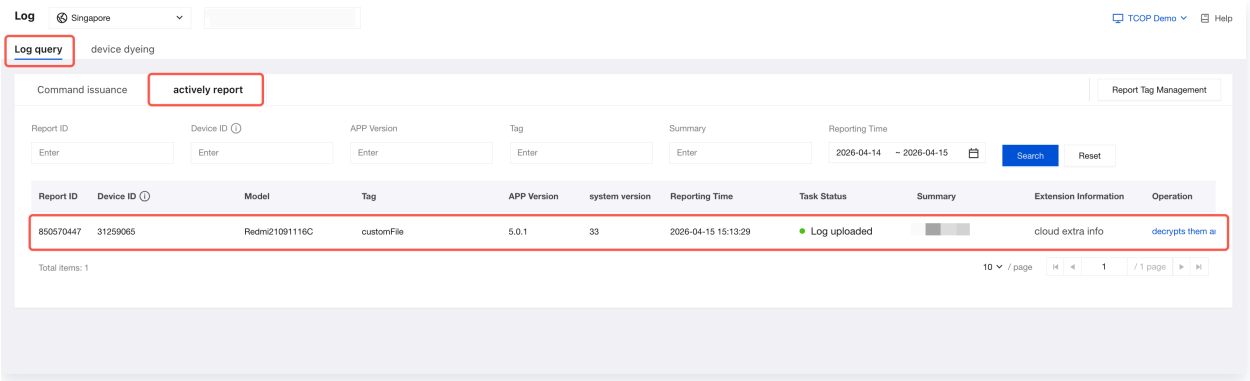
- Size limit: Restricts the total file size before compression (default: 500 MB). If the limit is exceeded, only the most recently modified files are retained.

- Frequency limit: Allows you to set a limit on how often proactive uploads can occur , such as 3 times within 10 minutes. Disabled by default. See TDLogSDKConfig for configuration options.
- Traffic limit: Allows you to set a daily quota for total reporting traffic and XG traffic. Disabled by default. Once the quota is exceeded, reporting is blocked. See TDLogSDKConfig for configuration options.
- Error circuit breaker: After 10 consecutive upload failures, automatic reporting is suspended for 6 hours. This is an internal safety mechanism and is not configurable.

The TDOS_Diagnose SDK allows the app to call the API to upload custom logs, files, and other information. Logs reported proactively can be queried under the actively reporting tab on the **Terminal Performance Monitoring Pro > Log > Log query** page.
You can use this feature to promptly report local log information such as user feedback or exception, helping preserve the issue context. The API description is as follows:

```
/// Proactively reports logs.
/// @param files List of log files
/// @param tag Tag (required, max 64 bytes)
/// @param summary Summary information (searchable, max 256 bytes)
/// @param extendInfoDict Extended information (max 1 KB)
/// @param completionBlock Callback for upload result
- (void)uploadFiles:(nonnull NSArray <NSString *> *)files
            withTag:(NSString *)tag
            summary:(nullable NSString *)summary
      andExtendInfo:(nullable NSDictionary *)extendInfoDict
         completion:(void (^)(BOOL result, NSString *_Nullable
errMsg))completionBlock;
```

After successful reporting, you can view the reporting records and content under the **actively report** tab on the Terminal Performance Monitoring Pro > Log > Log query page.



Note:

> Be mindful of the content size and call frequency when triggering proactive reporting to avoid excessive data usage or performance issues.

API Description

Last updated：2026-05-25 18:01:58

This article describes the feature APIs of the Terminal Performance Monitoring Pro (RUM Pro) log SDK to facilitate flexible and in-depth integration.

Log Printing Module

The SDK's built-in log printing capability is based on the WeChat Mars-xlog solution. The public interfaces are defined in TDLoggingIMPProtocol (TDOS_Diagnose/Classes/LogReport/TDLoggingIMPProtocol.h), which includes features such as local log printing, log coloring, and log file retrieval. The usage is as follows:

```
// Header file
#import <TDOS_Diagnose/TDOSLoggerProxy.h>

// Initialization method:
@interface TDOSLogger : NSObject <TDLoggingIMPProtocol>

/// Initializes the log SDK based on the configuration.
/// @param config configuration
- (instancetype)initWithConfig:(TDOSLoggerConfig *)config;

@end

// TDOSLoggerConfig includes the following optional settings:

/// Log directory
@property (nonatomic, readonly) NSString *logPath;
/// Default log level. Default: RAFTLogLevel_Error
@property (nonatomic, readonly) RAFTLogLevel defaultLogLevel;
/// Encryption Key
@property (nonatomic, readonly) NSString *publicKey;
/// Whether to output logs to the console. Default: NO (disabled).
@property (nonatomic, assign) BOOL enableConsoleLog;
/// Log file name prefix (optional)
@property (nonatomic, copy) NSString *logFilePrefix;
```

For detailed capability descriptions, see the sections below.

Basic Capabilities

Includes printing logs, setting log levels, forcing log flush, and so on. Log levels follow the RAFT log definition, RAFTLogDef(Log/RAFTLogDef.h), which provides 7 levels.

```
RAFTLogLevelVerbose = 0,
RAFTLogLevelDebug = 1,
RAFTLogLevelInfo = 2,
RAFTLogLevelWarn = 3,
RAFTLogLevelError = 4,
RAFTLogLevelFatal = 5,
RAFTLogLevelNone = 6,
```

On this basis, TDOS_DiagnoseSDK provides 5 macro definitions in TDOSLoggerProxy for quick integration:

```
#define TDLogDebug(tag, format, ...)
#define TDLogInfo(tag, format, ...)
#define TDLogWarn(tag, format, ...)
#define TDLogError(tag, format, ...)
#define TDLogFatal(tag, format, ...)
```

The underlying APIs are as follows:

```
/// log printing API (without format string)
- (void)log:(RAFTLogLevel)level
        tag:(NSString *)tag
       file:(const char *)file
       func:(const char *)func
       line:(int)line
        msg:(NSString *)msg;

/// log printing API (with format string)
- (void)log:(RAFTLogLevel)level
        tag:(NSString *)tag
       file:(const char *)file
       func:(const char *)func
       line:(int)line
     format:(NSString *)format, ... __attribute__((format(__NSString__,
6, 7))) NS_REQUIRES_NIL_TERMINATION;
```

```
/// Checks if coloring is currently enabled.
- (BOOL)isColorState;

/// Resets the cloloring level.
- (void)resetColorLevel;

/// Sets the log level.
/// @param level Log level
- (void)setLogLevel:(RAFTLogLevel)level;

/// Sets the special level for printing log and the expiration time for
the special level.
/// @param level Log level
/// @param endtime End time
- (void)setLogLevel:(RAFTLogLevel)level endTimestamp:(time_t)endtime;

/// Forces writes logs to file.
/// @param isSync Whether to write synchronously
- (void)flushLog:(BOOL)isSync;
```

Additional Log Settings

These include setting the maximum retention time for local logs (default: 10 days), setting the maximum total size of locally retained logs, setting console log output, and setting the timing for log persistence, and so on.

```
/// Sets the maximum log retention time.
/// @param time Unit: seconds. Default: 10 days (10 * 24 * 60 * 60)
- (void)setMaxLogAliveTime:(long)time;

/// Sets the total log file size. When exceeded, old logs are
automatically cleaned up (minimum limit: 50 MB).
/// @param maxSize Total file size. Default: 0 (unlimited).
- (void)setMaxTotalLogFileSize:(int64_t)maxSize;

/// Sets console log output.
/// @param enabled Whether to enable
- (void)setConsoleLogEnabled:(BOOL)enabled;
```

Log Information Retrieval

Get a list of log files (grouped by hour) within a specified time range (including the size limit feature), the log folder path, and the log encryption public key.

```
/// Gets a list of log files (hourly printed).
/// @param startTime Start timestamp
/// @param endTime End timestamp
- (nullable NSArray<NSString *> *)getPeroidLogFilesWithStartTime:
(NSTimeInterval)startTime
                                                         endTime:
(NSTimeInterval)endTime;

/// Gets the log list, supporting limiting the maximum log size.
/// (Discards old files first when the size limit is exceeded).
///
/// @param startTime Start timestamp
/// @param endTime End timestamp
/// @param sizeLimit Maximum log package
- (nullable NSArray<NSString *> *)getPeroidLogFilesWithStartTime:
(NSTimeInterval)startTime
                                                         endTime:
(NSTimeInterval)endTime
                                                       sizeLimit:
(unsigned long long)sizeLimit;

/// Gets the log folder path.
- (NSString *)getLogFolderPath;

/// Gets the log encryption public key.
- (nullable NSString *)getLogEncryptPublicKey;
```

Push-triggered Proactive Log Retrieval (Optional)

You can integrate your app's Push channel to trigger proactive log retrieval. Using your own push capability enables more timely log retrieval, reducing wait times.

> ⚠ Note:

> Your push delivery capability needs to be integrated with the RUM Pro platform, or you can proactively trigger the corresponding command after creating a new query.

When the client receives a push command, it must proactively call the following API of TDOS_Diagnose SDK to request the log retrieval configuration promptly.

```
/// Proactively pulls the log retrieval command.
///
/// Call this when a push message is received.
- (void)requestLogConfigFromServer;

// Calling example:
#import <TDOS_Diagnose/TDOS_Diagnose.h>
[TDLogSDK.sharedInstance requestLogConfigFromServer];
```

Console Operation Guide
Metrics Description

Last updated：2026-05-25 18:53:26

Overview

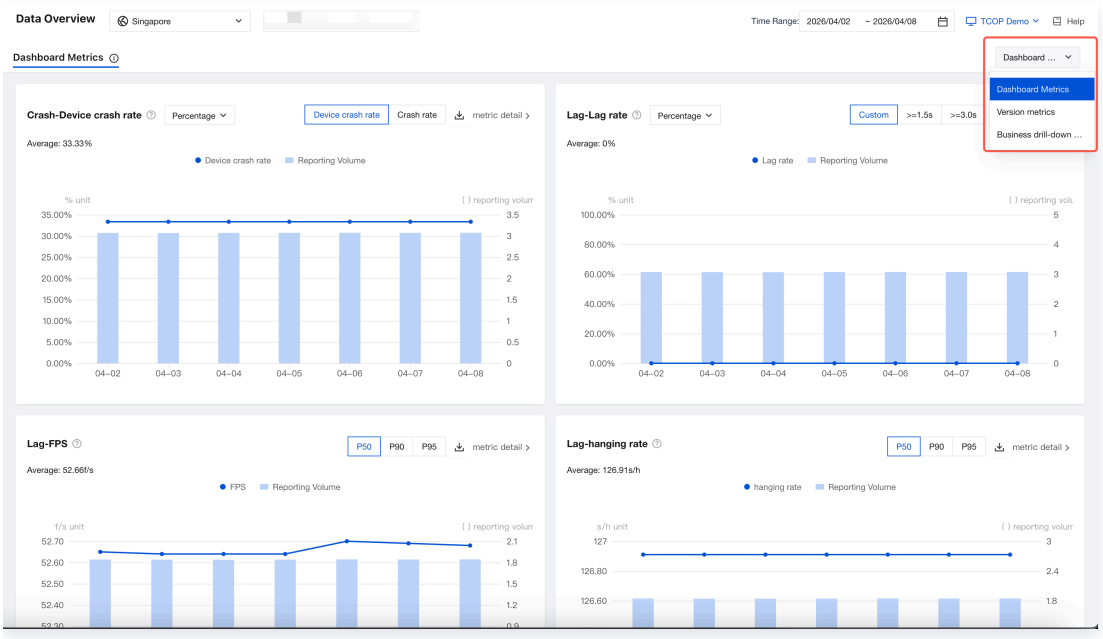
RUM Pro currently supports metrics such as crashes, ANR, OOM/FOOM, error rates, FPS, suspension rate, memory peaks, app launch time, and more. These metrics vary in their statistical approaches and are primarily categorized into two types:

- The metrics for calculating abnormality rates based on reported individual cases include crashes, ANR, and OOM/FOOM. For such metrics, the numerator typically represents the count of reported abnormal cases, while the denominator is the number of connected devices (deduplicated by device ID).
- Metrics such as FPS, suspension rate, memory peak, and app launch time are based on independently collected data statistics and are not directly correlated with the number of reported issues.

After confirming that the configuration is properly enabled, you can view relevant metric data on the Data Overview, Abnormal Overview, Metrics Analysis, and Issue List pages.

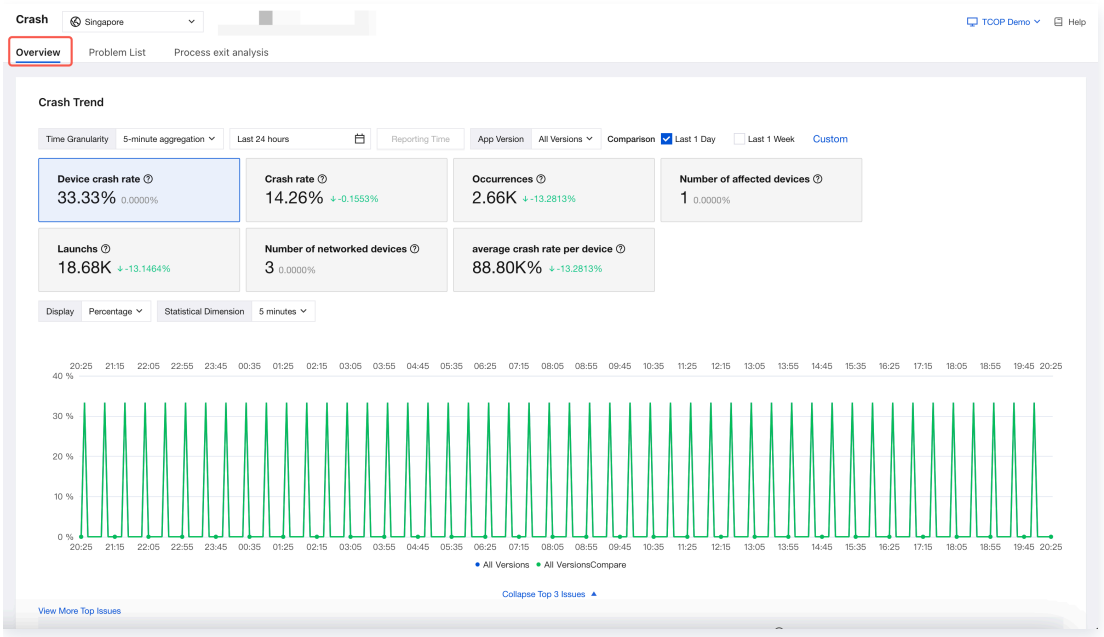
Data Overview

Data Overview page primarily displays the core metrics related to RUM Pro, visualizing various monitoring metrics to facilitate monitoring and analysis, including **Dashboard Metrics**, **Version metrics**, and **Business drill-down metrics**.



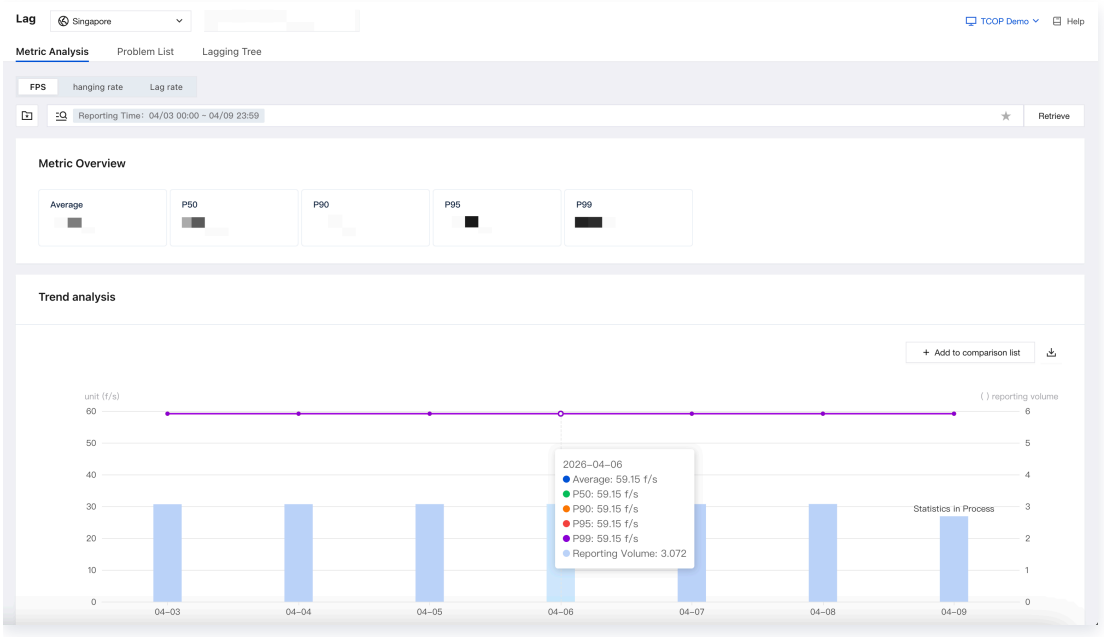
Exception Overview

Crashes, errors, as well as ANR and OOM metrics on the Android platform, can all be viewed through their respective overview modules, as shown in the figure below.



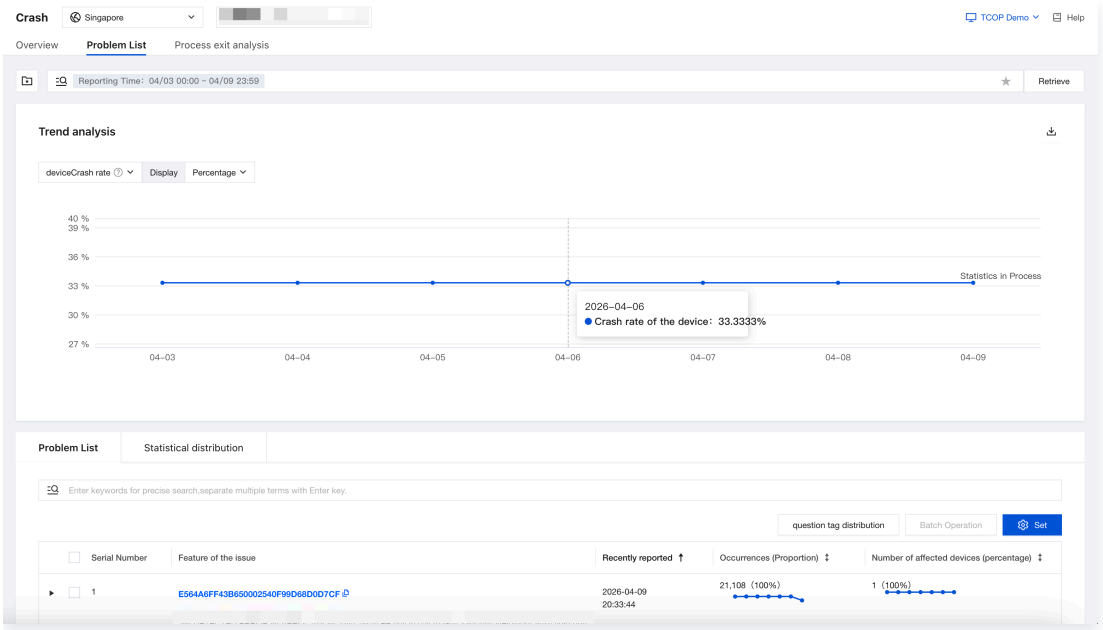
Metric Analysis

iOS metrics such as ANR, FOOM, lag, memory peaks, and app launch time can be viewed on the Metrics Analysis page, as shown in the figure below.



Problem List

In addition to analyzing metric data through the Data Overview, Abnormal Overview, and Metrics Analysis pages, metrics such as crash, ANR, OOM/FOOM, and errors can also be flexibly analyzed in the Problem List.



Metrics Introduction

Crash

Crash refers to an application stopping normal operation and exiting. Crash metrics include Device crash rate; Crash rate; and User Crash rate.

Definition of Abnormality Rate:

- Device Crash rate = Affected Device Count / Connected Device Count.
- Frequency-based Crash rate = Occurrence Count / Launch Count.
- User Crash rate = Affected User Count / Connected Device Count.

Definition of Statistical Fields:

- Number of Affected Devices: The count of devices experiencing crashes, calculated as a distinct count by **Device ID**.
- Number of Affected Users: The count of users experiencing crashes, calculated as a distinct count by **User ID**.
- Connected Device Count: The count of internet-connected devices, calculated as a distinct count by **Device ID**.

ANR

ANR: When an application becomes unresponsive during operation and is force closed by the system, it is recorded as an ANR.

Definition of Abnormality Rate:

- Device ANR Rate = Number of Affected Devices / Connected Device Count.
- Frequency-based ANR Rate = Occurrence Count / Launch Count.
- User ANR Rate = Number of Affected Users / Connected Device Count.

Definition of Statistical Fields:

- Number of Affected Devices: The count of devices experiencing ANR, calculated as a distinct count by **Device ID**.
- Number of Affected Users: The count of users experiencing ANR, calculated as a distinct count by **User ID**.
- Connected Device Count: The count of internet-connected devices, calculated as a distinct count by **Device ID**.

> **Note:**
> - Android ANR monitoring only requires the SDK to be properly initialized and the product to contain an active resource package to report normally. Android ANR monitoring currently does not support user toggle and is enabled by default.
> - iOS ANR monitoring supports user toggle. After SDK initialization and purchasing a resource package for the product, you need to create a configuration task to enable iOS ANR monitoring.

OOM

On the Android platform, the OOM rate includes the following three types:

- Java OOM Rate: The probability of crashes caused by Java memory usage exceeding limits in the process.
- Native OOM Rate: The probability of crashes caused by Native memory usage exceeding limits in the process.
- FD OOM Rate: The probability of crashes caused by FD resource usage exceeding limits in the process.

Definition of Abnormality Rate:

- Device OOM Rate = Number of Affected Devices / Connected Device Count.
- Count-based OOM Rate = Occurrence Count / Launch Count.
- User OOM Rate = Affected Users / Connected Devices.

Definition of Statistical Fields:

- Affected Devices: The number of devices that experienced OOM, counted by deduplication based on **Device ID**.
- Affected Users: The number of users affected by OOM, counted by deduplication based on **User ID**.
- Connected Device Count: The count of internet-connected devices, calculated as a distinct count by **Device ID**.

> **Note:**

> Android OOM, similar to crashes, can be reported normally as long as the SDK is properly initialized and the product has an active resource package. Android OOM monitoring currently does not support user toggles and is enabled by default.

FOOM

FOOM (Foreground Out Of Memory) on the iOS platform refers to the system force-quitting an application due to excessive memory consumption while running in the foreground.

Definition of Abnormality Rate:

- Device FOOM Rate = Affected Devices / Connected Devices.
- FOOM Rate = Occurrences / Launches.
- FOOM Rate = Affected Users / Connected Devices.

Definition of Statistical Fields:

- Affected Devices: The number of devices affected by FOOM, counted by deduplication based on **Device ID**.
- Affected Users: The number of users affected by FOOM, counted by deduplication based on **User ID**.
- Connected Device Count: The count of internet-connected devices, calculated as a distinct count by **Device ID**.

> **Note:**
> FOOM on the iOS platform is not enabled by default. Users need to go to Settings / SDK Configuration to create a configuration task or modify an existing configuration task to enable FOOM monitoring.

Error

Errors refer to user-defined exceptions reported to the platform through the APIs provided by the RUM Pro SDK.

Definition of Abnormality Rate:

- Device Error Rate = Affected Devices / Connected Devices.
- Error Rate = Occurrence Count / Launch Count.
- User Error Rate = Affected Users / Connected Devices.

Definition of Statistical Fields:

- Affected Devices: The number of devices that reported errors, counted by deduplication based on **Device ID**.
- Affected Users: users who reported errors, counted by deduplication based on **User ID**.
- Connected Device Count: The count of internet-connected devices, calculated as a distinct count by **Device ID**.

> **Note:**
> Similar to crash, as long as the SDK is properly initialized and the product has an active resource package, errors can be reported normally.

Lag Metrics

Lag metrics include FPS and suspension rate, which are disabled by default (i.e., not collected). Users need to actively enable them by creating a configuration task.

- FPS: Frame rate, the number of images the GPU and CPU can generate during application runtime, measured in frames/second (Frames Per Second, abbreviated as FPS). It is typically used as a metric to evaluate hardware performance and application smoothness.
- Suspension rate: If the refresh delay between two application frames exceeds 200 ms, it is considered that the application has poor responsiveness to user interactions, and this delay is added to the suspension time. The suspension rate of a device is its total suspension time divided by the total foreground duration in a day, measured in seconds/hour.

Memory metrics

Memory peak: the maximum memory used by the application during a single run.

- Physical memory peak: peak physical memory used by the application during a single run, also known as peak physical set size (PSS).
- Virtual memory peak: peak virtual memory used by the application during a single run, also known as peak virtual set size (VSS).
- Java heap memory peak: peak Java heap memory used by the application during a single run.
- Foreground memory peak: peak memory used by the application in the foreground during a single run.
- Background memory peak: peak memory used by the application in the background during a single run.

Launch time

The metrics for launch monitoring are launch time, which includes cold launch time and warm launch time.

- Default cold launch: from process creation to the first frame drawn of the first Activity.
- Cold launch: from the creation of the application process to the completion of the first rendering of the launch page. You can customize the launch completion point.
- Warm launch: from the opening of an Activity to the completion of the first rendering of the page, when the process has already been created and there is no Active Activity.
- Context switch: switching to the foreground after the application is in a suspended state in the background.

> **Note:**

> Android platform default cold launch. The Android platform provides the AppLaunchMonitor#ReportAppFullLaunch API, allowing users to customize the launch completion point. Since the definition of the launch completion point varies across applications, direct comparison between different apps is not feasible. However, the default cold launch uses the first frame rendering of the initial Activity as the statistical endpoint for launch completion, enabling direct comparison of launch time between applications.

Crash

Last updated：2026-05-25 18:53:26

RUM Pro provides comprehensive crash analysis capabilities, supporting the collection and reporting of various crash exceptions on both Android and iOS platforms to help users better address application quality issues. This article details the features and usage of each module within the crash analysis functionality of RUM Pro.

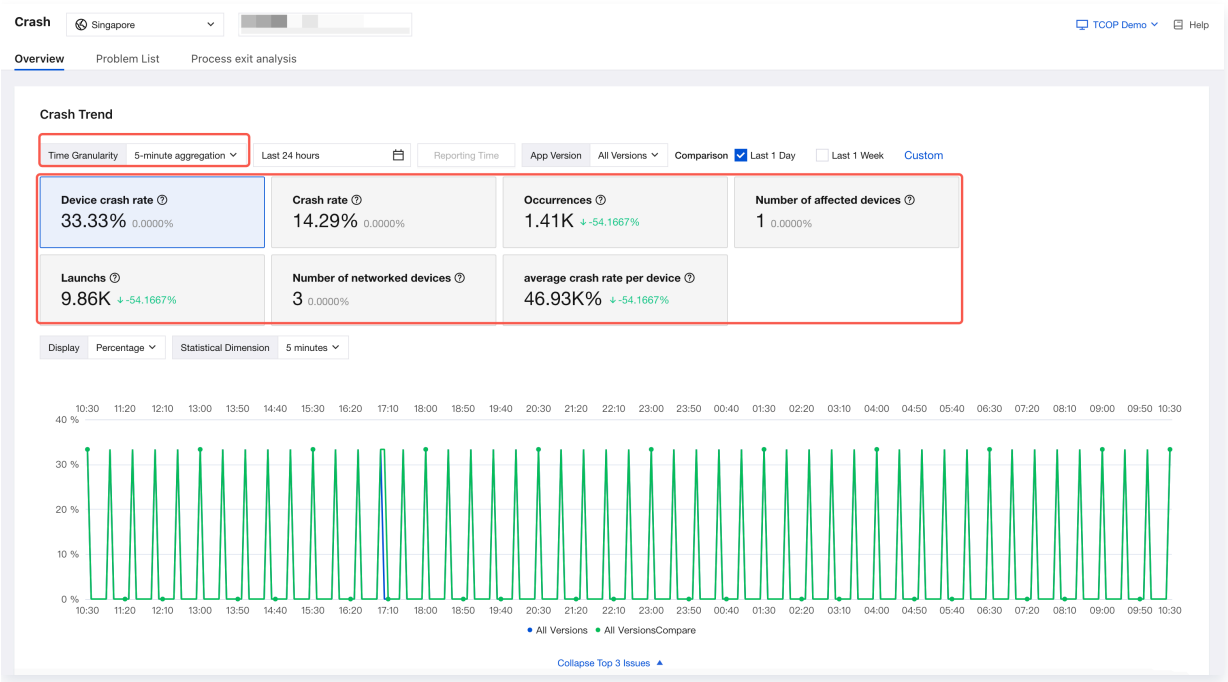
Crash Overview

On the **Crash Overview** page, RUM Pro provides comprehensive analysis of crash rates and crash trends, primarily supporting the following analytical capabilities.

Metrics Overview

Supports the display of today's **Device crash rate**, **Crash rate**, **Occurrences**, **Number of affected devices**, and supports day-over-day comparison relative to the previous day.

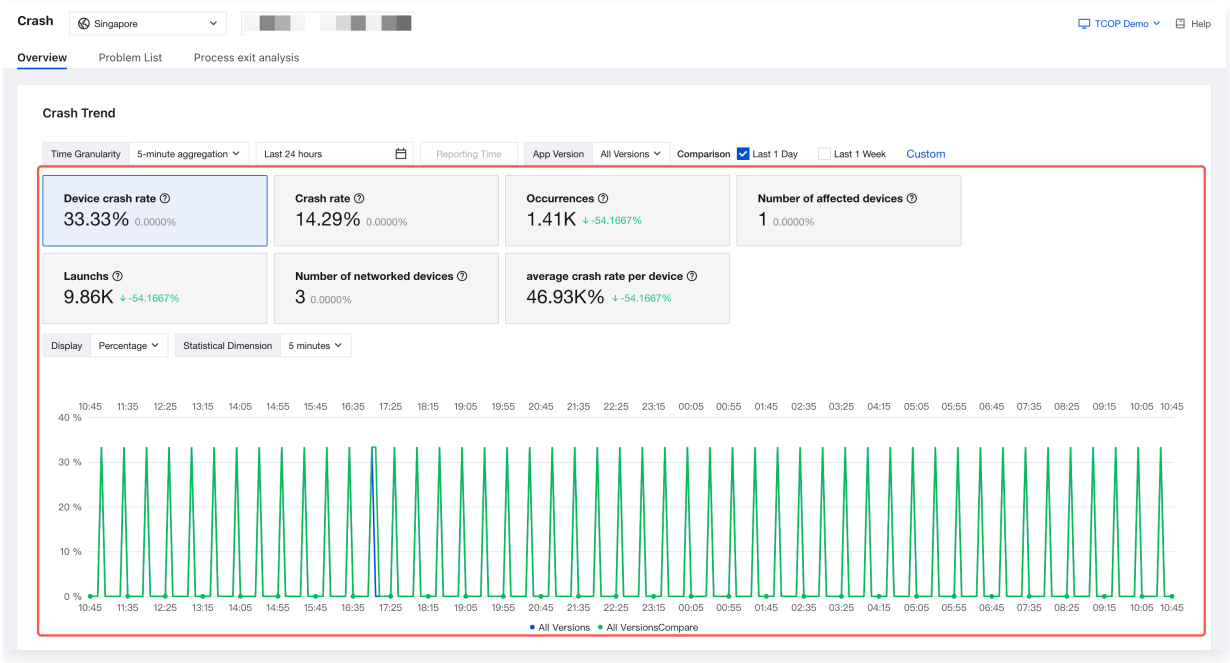
- **Device crash rate**: Device Crash Rate = Affected Device Count / Connected Device Count.
- **Crash rate**: Crash Occurrence Rate = Number of exception cases meeting filter conditions / Number of launches meeting filter conditions. (Currently, the statistics for launch counts only support two filter conditions: time range and App version.) Data is only available for viewing from July 29, 2023 onwards.
- **Occurrences**: The number of occurrences of exception cases that meet the filter conditions.
- **Number of affected devices**: The number of devices associated with exception cases meeting the filter conditions, with duplicates removed based on device ID during exception reporting.



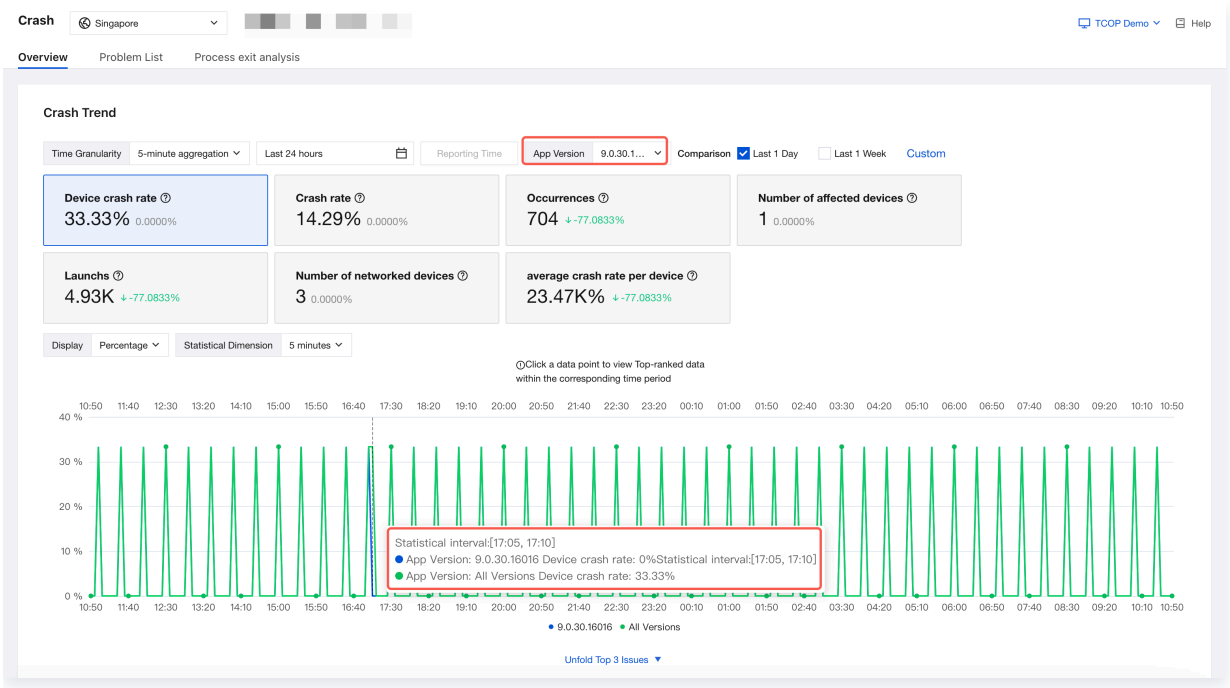
Crash Trend

- **App Version**: Contains all reported App versions. The dropdown allows selection of all versions or a single App version for filtering and viewing.
- **Time Granularity**: Supports aggregation by 5-minute intervals, 1-hour intervals, and daily intervals.
- **Display**: Trend charts can be viewed with coordinate dimensions set to percentage, per mille, or per ten thousand.
- **Statistical Dimension**: Includes two dimensions: 5-minute and cumulative. The 5-minute dimension trend chart aggregates data every 5 minutes based on filter conditions, providing more granular time points. The cumulative dimension trend chart aggregates data every 55 minutes based on filter conditions, offering a more intuitive view of crash trends.

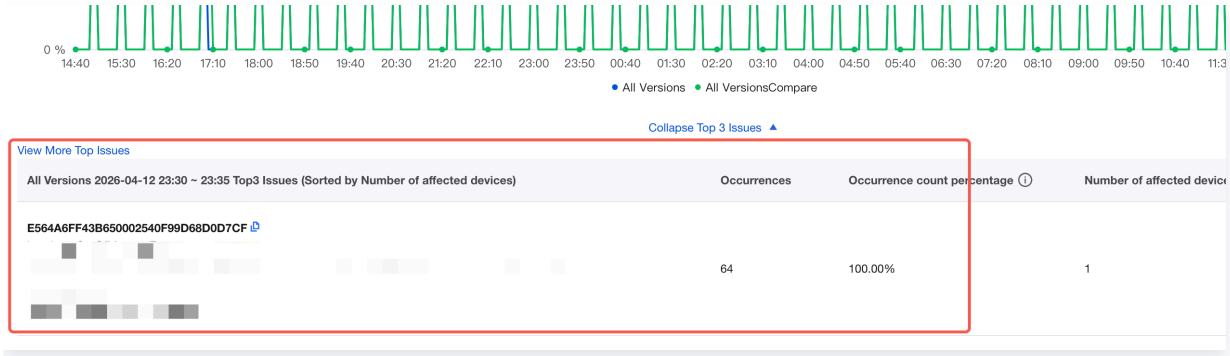
Supports multi-dimensional display of crash trends. At **Time Granularity**, select aggregation by 5 minutes, 1 hour, or by day. Simultaneously, in the right Tab bar, you can switch to display **Device crash rate**, **average crash rate per devices**, **Crash rate**, **Occurrences**, **Launchs**, **Number of affected devices**, **Number of networked devices**.



Supports the version comparison feature. Click **App Version** to select different application versions for comparison in the trend display.



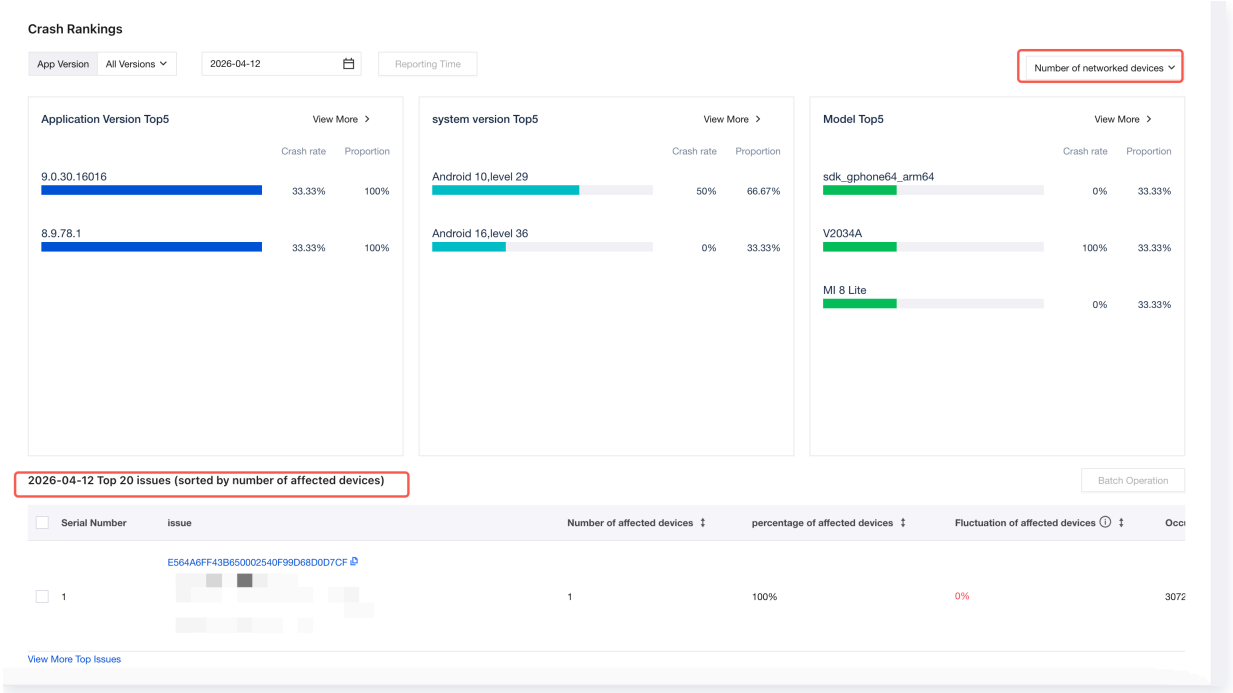
Supports crash analysis for the **Top Issues** within the trend chart time range. In the trend chart, clicking on different data ranges in the curve displays the Top 3 issues and their impact scope for that range below.



Crash Rankings

Supports drill-down statistics on crash rates by app version, system version, and device model. Click the dropdown in the upper-right corner to switch between **Number of affected devices** and **Number of networked devices**.

Supports statistical display of the Top 20 crash issues on specified dates, including their impact scope and fluctuations.



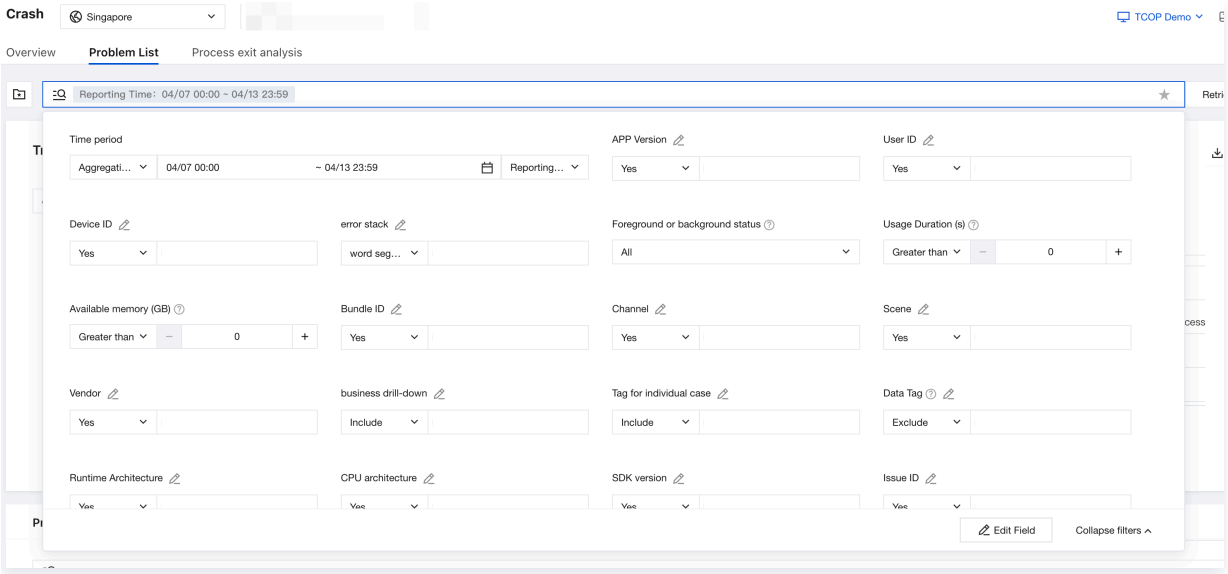
Crash Problem List

Problem Query

In the crash problem list, RUM Pro supports comprehensive individual case query capabilities, primarily as follows:

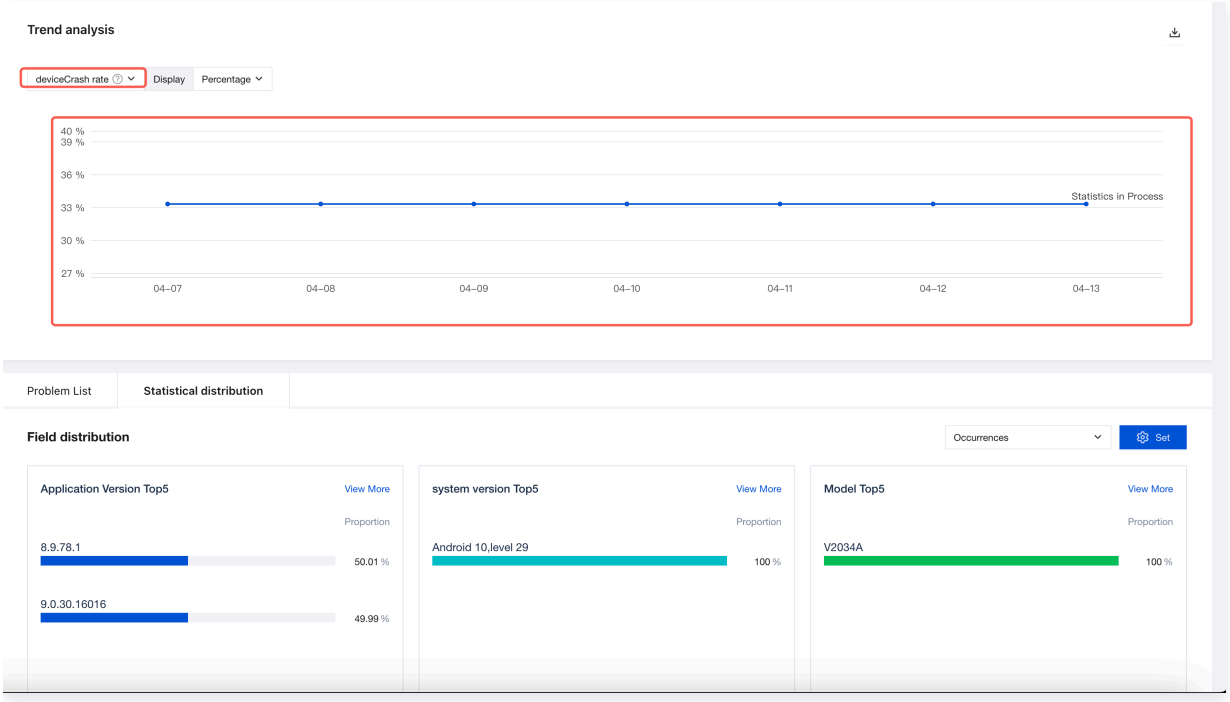
- Supports drill-down search through multi-dimensional fields such as time range, App version, user ID, device ID, foreground/background state, and Issue type.
- Supports advanced search features for error stack traces, including tokenized and non-tokenized search, with field intersections, unions, or fuzzy matching capabilities.
- Supports search for custom issue types such as processing status, assignee, Tag, and tag colors.
- Supports search for business-reported **Custom Fields**.
- Supports searching by **attachment name** within Problem case details, allowing filtering of cases with corresponding attachments reported.

For detailed fields and operation types, please see Data Analysis – Query.



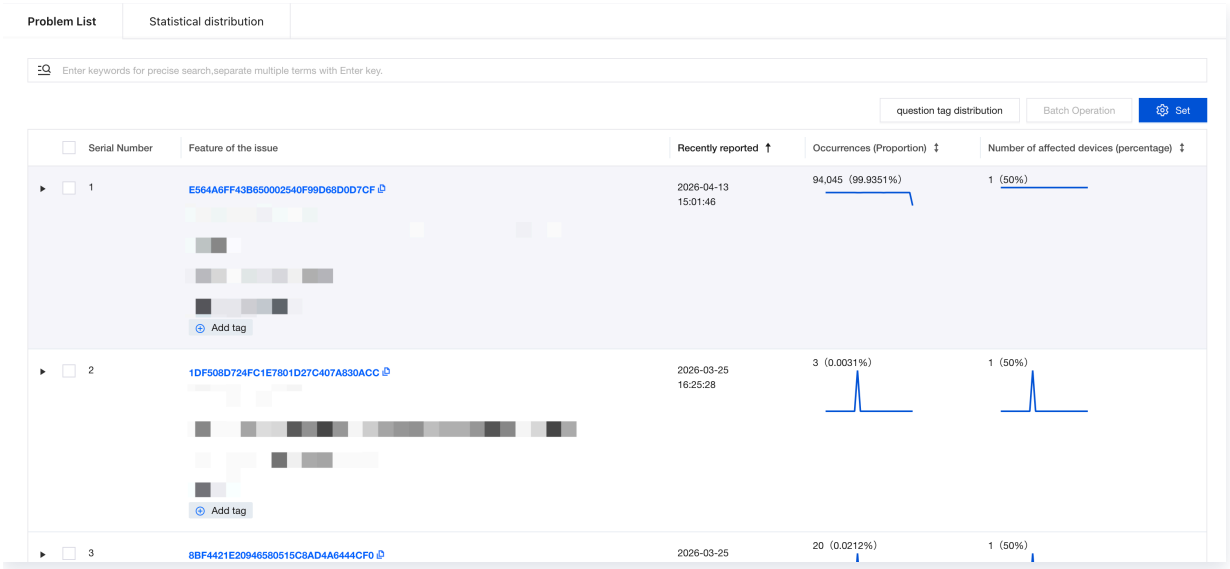
Trend analysis

Basic features are consistent with the Crash Trend in the crash overview.



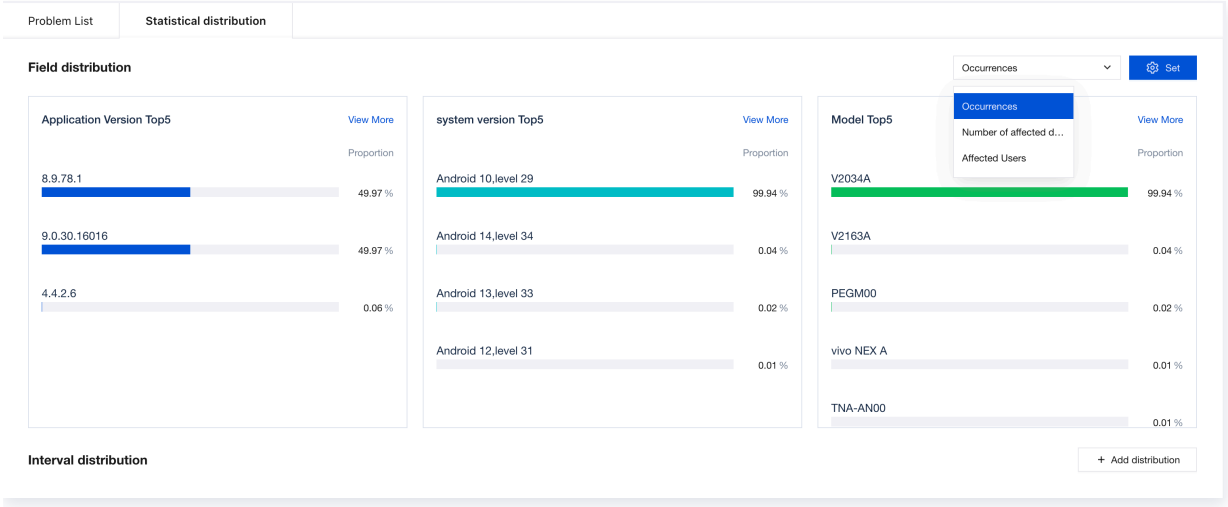
Problem List

The problem list supports problem management features such as feature overview, impact scope assessment, trend analysis, assignee tracking, and processing status management.



Statistical distribution

Supports drill-down statistics on crash rates by app version, system version, and device model. Click the dropdown in the upper-right corner to switch between **Occurrences**, **Number of affected device**, or **Affected Users**.

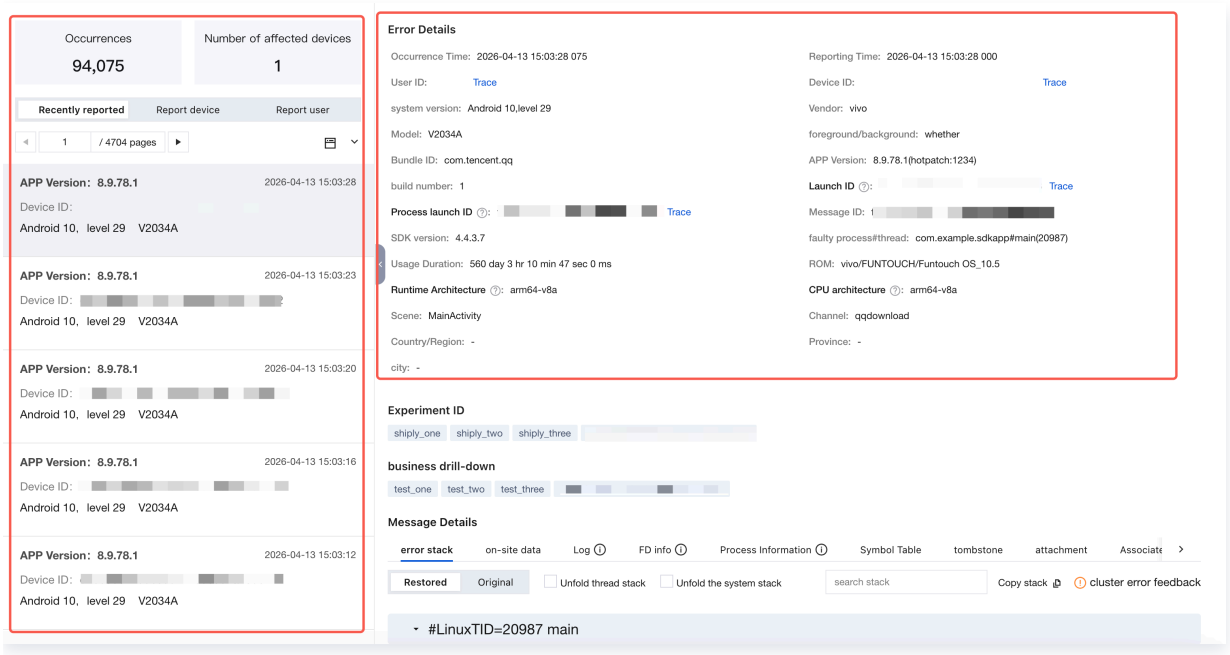


Crash Issue Details

Detailed case infomation

- **Error Details**: Includes information such as Occurrence Time, User ID, system version, Model, Bundle ID, build number, Process launch ID, SDK version, Usage Duration, Runtime Architecture, Scene, Reporting Time, Device ID, Vendor, foreground/background, APP Version, Launch ID, Message ID, faulty process#thread, ROM, CPU architecture, and Channel. This provides a more intuitive display of detailed information about this exception report.

- **Recently reported**: Displays the latest reported issue information, including APP Version, reporting time, Device ID, system version, and device model.
- **Report device**: Sorted by reported device IDs, enabling better search for specific error cases based on device ID. Includes APP Version, reporting time, Device ID, system version, and device model.
- **Report user**: Sorted by reported user IDs, enabling better search for specific error cases based on user ID. Includes APP Version, reporting time, User ID, system version, and device model.

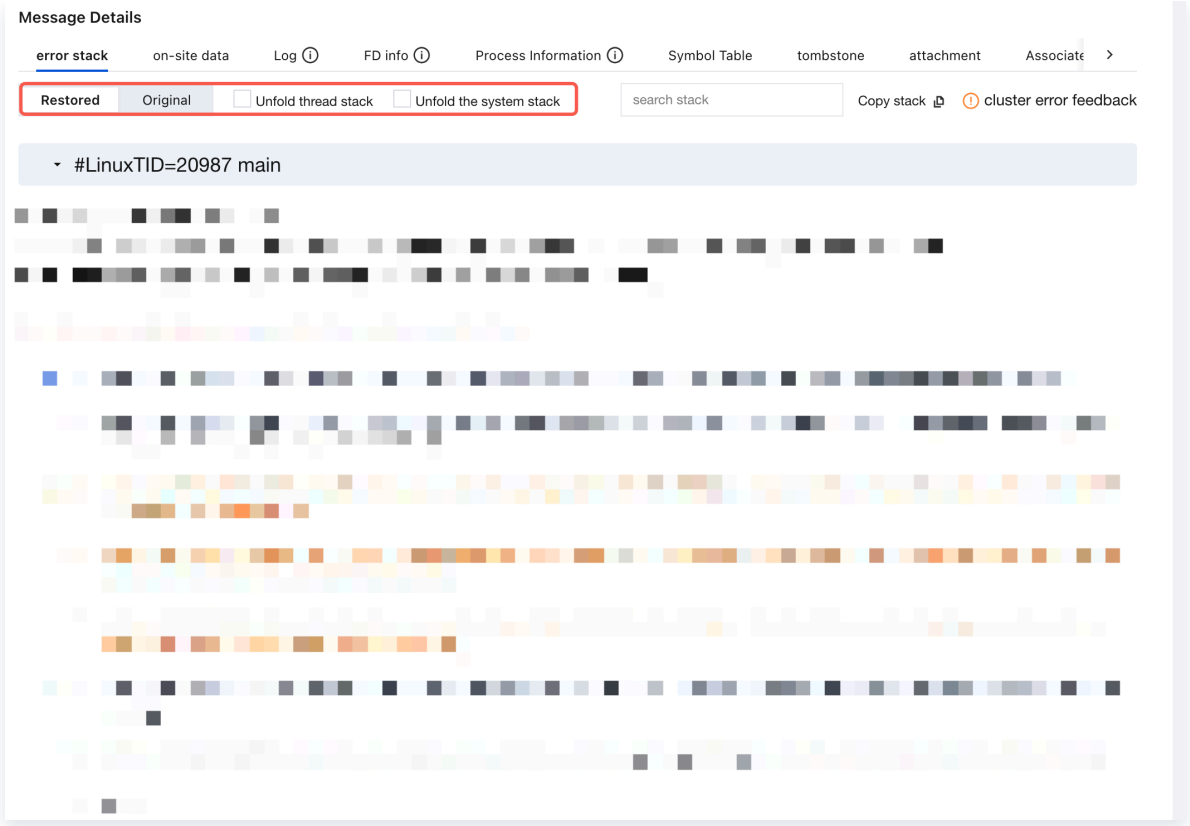


error stack

The stack trace provides detailed information about the current crash issue and serves as the most direct information for analyzing and resolving crash problems.

- In Java crash stack traces, the error stack displays the error message and the current call stack, which typically consists of file name + symbol + offset. If the Java code was obfuscated during compilation, you can upload the obfuscated mapping file to translate the obfuscated symbols. If the mapping file has been uploaded, use the button on the right to switch between displaying the **Original stack** or **Restored stack**, with the restored stack displayed by default.

  If the current thread stack cannot pinpoint the issue, you can click **Unfold thread stack** to view other thread stacks. Other thread stacks also support symbol deobfuscation.



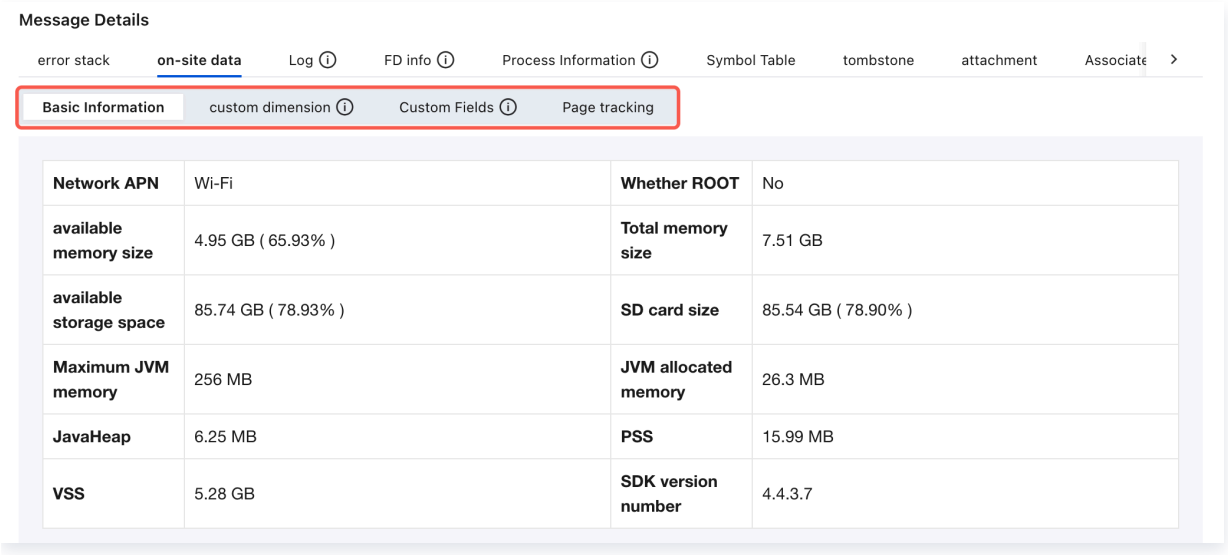
- In the error stack, if the corresponding symbol table for the stack has not been uploaded, the page will indicate that the stack is untranslated. You can click **Upload** to navigate to the symbol table upload page. After uploading the symbol table, click **manual restoration** in the upper-right corner of the page. Wait a few minutes and refresh the page; the stack will then be deobfuscated. For the operation procedure, see Symbol Tables.

on-site data

Crash snapshot data captures the on-site information collected at the moment of a crash, primarily including the following content.

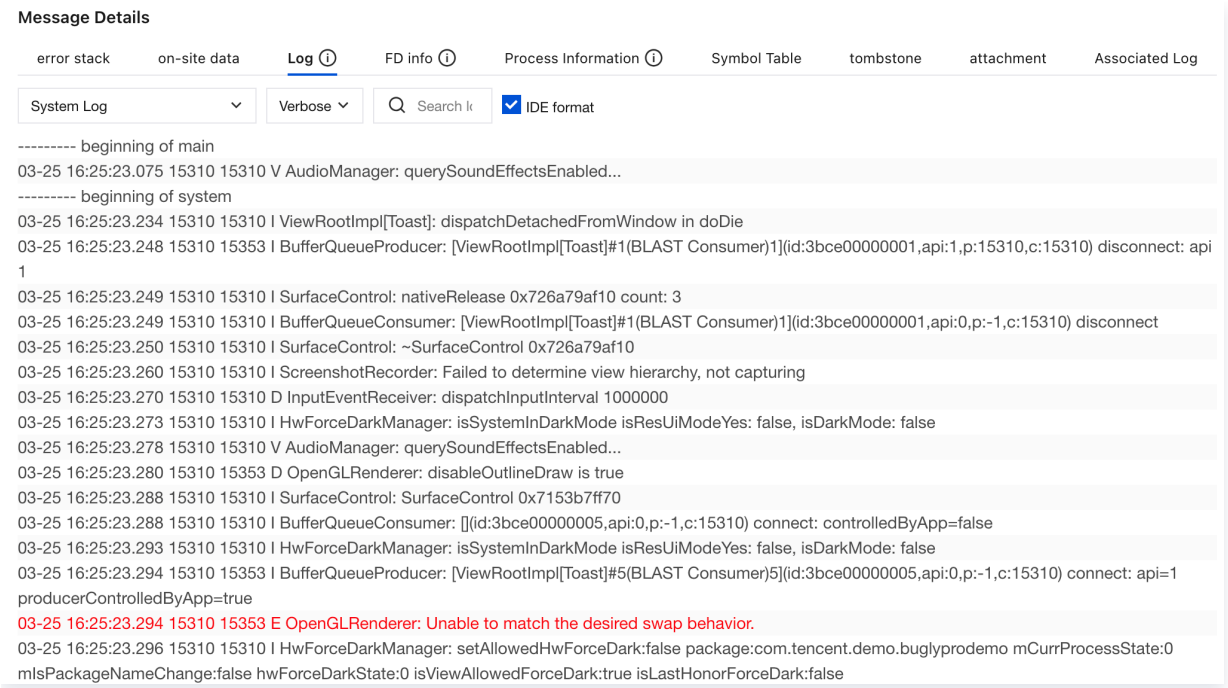
- Basic Information: Includes Network APN, memory status, storage status, JVM status, etc. The description of JVM status is as follows:
  - `Maximum JVM memory`: Indicates the maximum heap memory size that can be used by the JVM.
  - `JVM allocated memory`: Indicates the size of memory currently allocated to the JVM. Typically, the allocated memory is less than the maximum memory. When a Java program requires more memory, the JVM automatically expands the heap memory until it reaches the maximum limit. If the JVM cannot allocate additional memory, it may throw an OutOfMemoryError exception.
  - `JavaHeap`: Indicates the portion of allocated memory used to store Java object instances.
  - `PSS`: Indicates the size of the process's shared memory.
  - `VSS`: Indicates the size of the process's virtual memory.

- custom dimension: Custom dimensions are user-defined data with constrained keys. Keys are defined by the SDK, and applications can set values with different meanings based on actual needs. Values currently support three types: Number, String, and String Array, with each type supporting ten keys.
  - Values of the Number type correspond to dimension keys n1 ~ n10.
  - Values of the String type correspond to dimension keys s1 ~ s10.
  - Values of the String Array type correspond to dimension keys a1 ~ a10.
- Custom Fields: Display custom fields uploaded by users via the API, can refer to custom data.
- Page tracking: Displays the lifecycle tracking of pages prior to a crash, allowing analysis of user page interactions before the crash occurred.



Log

The log section displays the log stream before and after the crash occurrence time, supporting search by log level and keywords.



FD info

FD info displays the FD usage status at the time of crash.
Includes the current number of used FDs, unused FDs, and the maximum number of FDs available per process. FDs are categorized by type, quantity, and proportion. The FD details display the specific paths of opened FD resources, with the number of FDs pointing to each path indicated at the end. FD information can assist in troubleshooting by identifying FD types that are excessively opened during crashes, enabling targeted optimization for these FD types.

Process Information

> **Note:**
> Native crashes report this information, while Java crashes do not report it.

Process information displays details related to the process at the time of crash, including Maps information, Meminfo information, process status, and thread status. The raw content of process information can be viewed in the attachment **state_file.zip**.

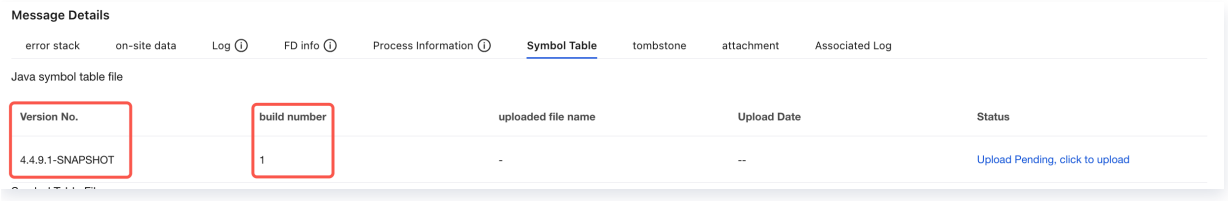
- Maps information represents the memory image of a running user process, allowing visualization of the current process space mapping. Specific memory segments can be filtered by address, segment name, and access permissions. Memory segment filtering supports regex matching, while access permissions require exact matches (e.g., rwxp denotes a private memory segment with readable, writable, and executable permissions).
- Meminfo information indicates the memory usage of user processes, showing how the application's current memory is primarily allocated. The coordinate descriptions in Meminfo information are as follows:

Coordinates	Field	Description
X-axis	Native Heap	Local heap usage.
	Dalvik Heap	Dalvik heap usage.
	Dalvik Other	Memory usage outside the Dalvik heap.
	Stack	Process stack usage.
	Ashmem	Anonymous shared memory usage.
	Other dev	Other device memory usage.
	.so mmap	Shared libraries' memory usage.
	.jar mmap	JAR file's memory usage.
	.apk mmap	APK file's memory usage.
	.ttf mmap	Font file's memory usage.
	.dex mmap	DEX file's memory usage.
	.art mmap	ART file's memory usage.
	.oat mmap	OAT file's memory usage.
	Other mmap	Other memory-mapped files' usage.
	GL mtrack	OpenGL ES memory usage.
	Unknown	Unknown memory usage.
Y-axis	Pss Total	The total physical memory used by the process, including shared libraries, anonymous shared memory, heap and stack, etc.
	Private Dirty	Private Dirty memory size used by the process, i.e., the memory size exclusively used by the process, including heap, stack, anonymous shared memory, etc.
	Private Clean	Private Clean memory size used by the process, i.e., memory which has been cleared and contains no data.
	SwapPss Dirty	Swap dirty page size of the process.

- Process Status and Thread Status indicate the state metrics of the currently running process and its individual threads, respectively. In Thread Status, you can click the arrow next to the thread name to expand the status of a single thread. When hovering over a metric, the description of the corresponding state indicator will be displayed.

Symbo Table

In the message details of a crash instance, you can also upload symbol mapping files. Unlike the symbol mapping file upload in **Settings**, this section only displays the required symbol mapping files for unresolved Java stacks and Native stacks in the current instance's **error stack**.
Java symbol table files are matched based on version number + build number, while so symbol table files are matched based on UUID. Click **Upload Pending, click to upload** to directly upload the corresponding symbol table on this page. After uploading, click **Manual Deobfuscation** to perform the deobfuscation. Wait a few minutes, and the current **error stack** will be deobfuscated. For symbol table related information, refer to the symbol table.



Attachment

RUM Pro supports package download for all attachments. User-customized reported data files can be found in the following attachments:

- The content set via the callback API `getCrashExtraData` is stored in userExtraByteData.
- The content set via the callback API `getCrashExtraMessage` is stored in the user_datas.log file.
- The content set via the custom field `putUserData` is stored in valueMapOthers.txt.
- The large files set via the custom file `setAdditionalAttachmentPaths` are stored in custom_log.zip.

drill-down analysis

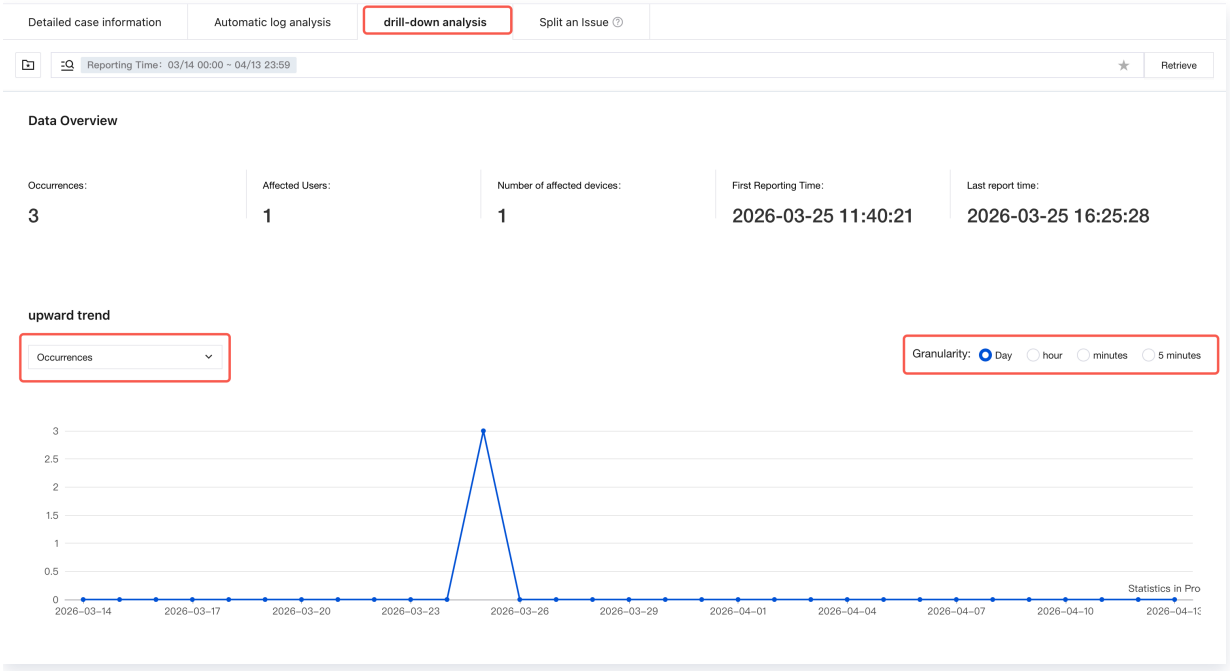
In the crash Issue instance, drill-down information for the Issue can be viewed on the **drill-down analysis** page.

Data Overview

RUM Pro provides data on multiple dimensions for crash Issue analysis, including occurrences, Affected users, Number of affected devices, First Reported Time, and Last reported time.

upward trend

The upward trend page displays statistical charts of Issue reporting trends at different granularities. Granularity options include day, hour, minute, and 5 minutes. Reporting trends can be shown by Occurrences, Affected Users, or Number of affected devices.



Statistical distribution

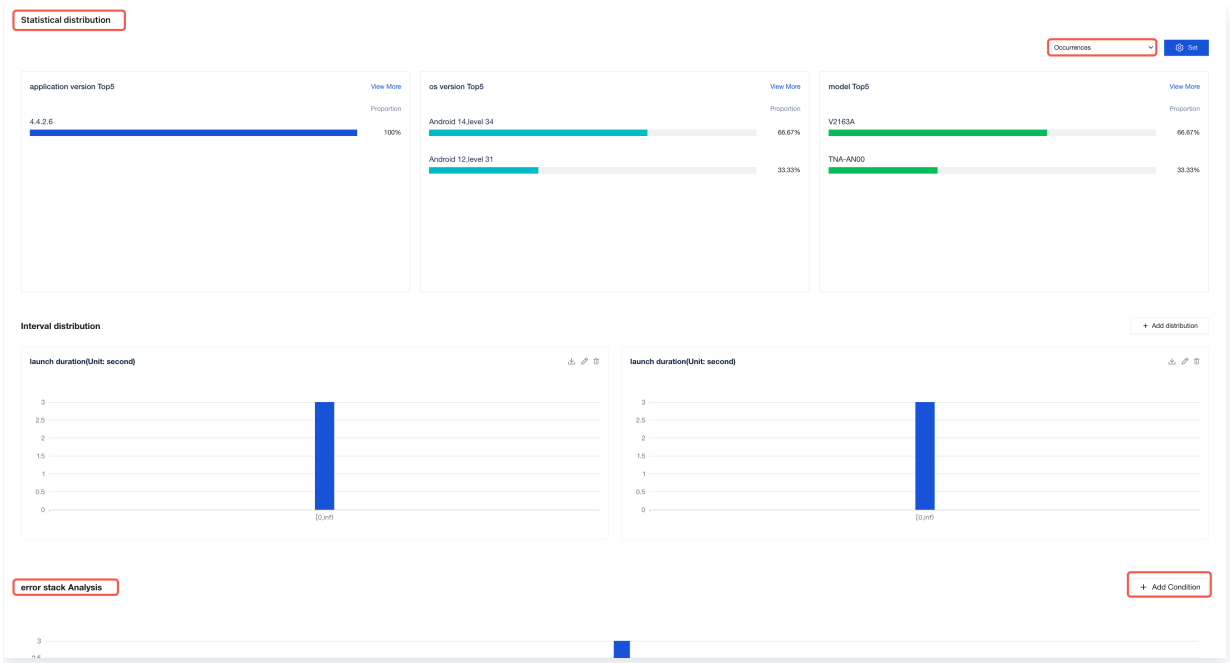
The analysis supports examining Issue problems through pie charts based on different dimensions (such as application version, OS version, model, etc.). The analysis also allows selection by occurrence count, number of affected users, or number of affected devices.
If you need to analyze the distribution proportion of different device models or architectures under this Issue, or examine the proportion of this problem across different application versions, you can use this statistical distribution feature to view the data.

Interval Distribution

Supports analyzing Issue problems in the form of bar charts based on different fields (such as usage duration, available memory, number of device exceptions, number of user exceptions) and different dimensions (such as occurrence count, number of affected users, number of affected devices). Click **Add distribution** in the upper right corner to add a new distribution. In the add distribution section, supports inputting intervals such as [0,5), [5,30), [30,60).

error stack Analysis

Supports analyzing issues based on error stack content. Click **Add Condition** in the upper right corner to add/edit conditions. In the edit condition section, you can add strings to search for, such as searching for the number of individual cases containing the `com` string in the current Issue. The search results are displayed comparatively in bar charts.


Statistical distribution
Occurrences
Set
application version Top5
View More
Proportion
4.4.2.6
100%
os version Top5
View More
Proportion
Android 14,level 34
66.67%
Android 12,level 31
33.33%
model Top5
View More
Proportion
V2163A
66.67%
TNA-AN00
33.33%
Interval distribution
Add distribution
launch duration(Unit: second)
launch duration(Unit: second)
3
2.5
2
1.5
1
0.5
0
(0,inf)
error stack Analysis
Add Condition

ANR

Last updated：2026-05-25 18:53:26

RUM Pro provides comprehensive ANR analysis capabilities, supporting the collection and reporting of ANR exceptions on both Android and iOS platforms to help users better optimize application quality. This article details the ANR analysis capabilities and features of RUM Pro.

Android

ANR Overview

Android ANR reporting content is similar to crash monitoring. You can refer to the crash overview to view and analyze the data.

ANR Issue List

The content reported for Android ANR is similar to crash monitoring. You can refer to the crash issues list to view and analyze the reported issues.

ANR Issue Details

Case Details

In the ANR case details, the content of **Error Details**, **error stack**, **on-site data**, **log**, **FD info**, **Symbol Table**, and **attachment** is similar to crash monitoring. You can refer to crash case details to view and analyze the data.

GC details

To view the GC execution time and Java memory allocation information before ANR occurrences.

- Garbage Collection (GC): A mechanism in Android virtual machines (such as Dalvik and ART) that automatically reclaims memory occupied by unused objects. When objects are no longer referenced, GC marks and frees their occupied memory to prevent memory leaks.
- Block GC: Block Garbage Collection, a phenomenon where application threads are paused (Stop-The-World) during GC execution, causing stuttering in the main thread (UI thread). This is a key issue in Android performance optimization.
- heapAllocated: indicates the currently allocated heap memory size of the process, in KB, corresponding to the calculation result of Runtime.totalMemory() – Runtime.freeMemory().
- heapMaxMemory: indicates the maximum heap memory size that the current process can allocate, in KB, corresponding to the return value of the Runtime.getMaxMemory() interface.

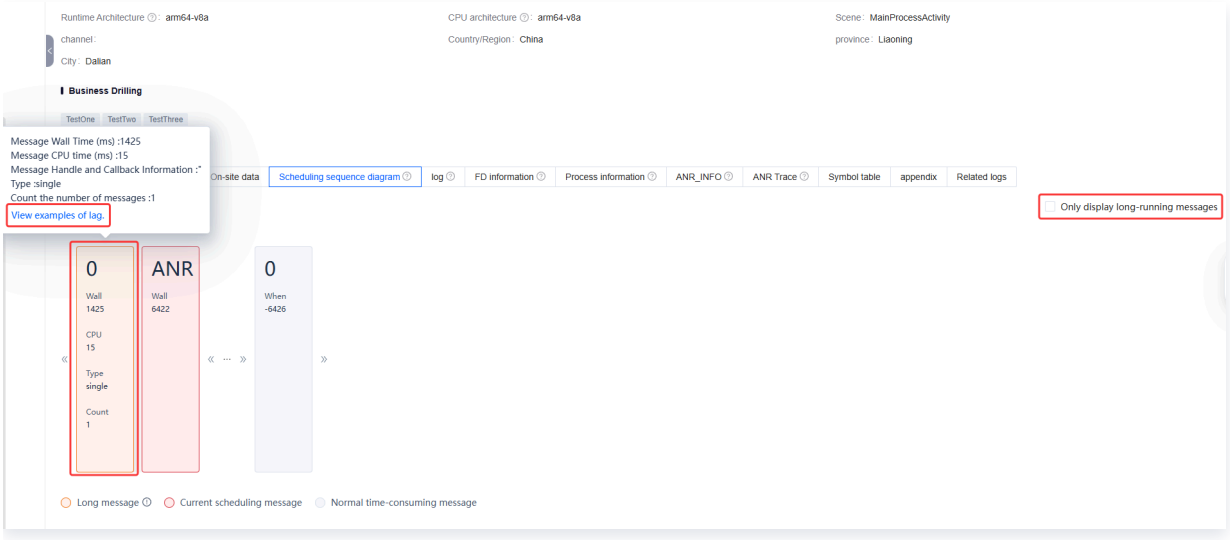
Scheduling sequence diagram

The scheduling sequence diagram displays the message scheduling during ANR occurrences in a sequence diagram format. Click the arrow next to a message to expand details of message scheduling around the ANR occurrence.
Long-duration messages are displayed in yellow cards. Hovering over a message card reveals information such as Wall Time, CPU Time, Message Handler, and Callback for the corresponding message.

Note:

- Wall Time: Actual duration, referring to the actual physical time elapsed from the start to the completion of a message's execution (including all time spent on thread blocking, I/O waiting, CPU scheduling delays, etc.).
- CPU Time: CPU execution time, referring to the actual time the thread occupies the CPU for computation during message execution (excluding waiting time).
- Message Handler: A message processor responsible for dispatching messages to the Looper's message queue and calling the corresponding handling logic upon message execution.
- Callback: A callback API, which is executable logic attached to a Message, and can be an implementation of either Runnable or the Handler.Callback interface.

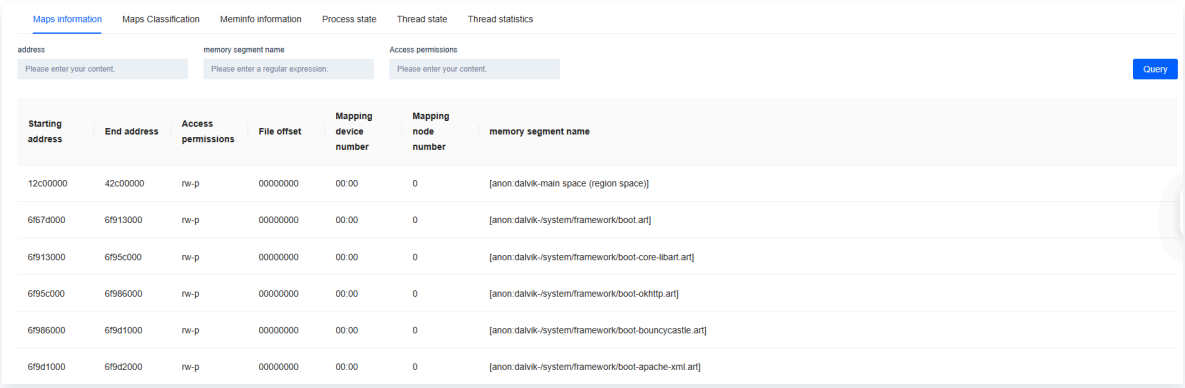
When analyzing ANR issues, you can select **Only display long-running messages** in the upper-right corner of the module to filter long-duration messages. To view detailed stack traces within long-duration messages, click **View examples of lag** in the specified message card. This will redirect you to the stutter page to view individual stutter reports for the device. Analyze the stack traces in these reports to comprehensively determine the causes of ANR.



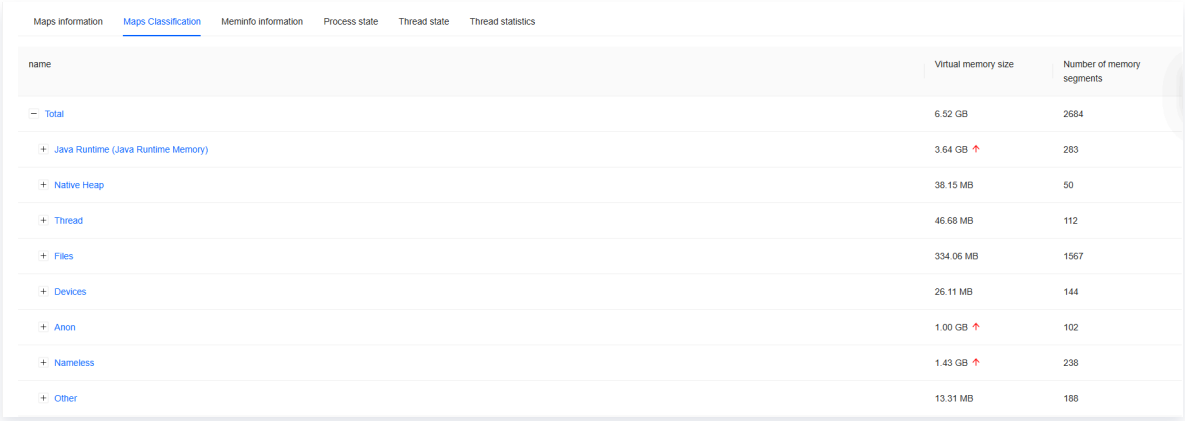
Process Information

Process information includes **Maps information**, **Maps Classification**, **Meminfo infomation**, **Process state**, **Thread state**, and **Thread statistics**.

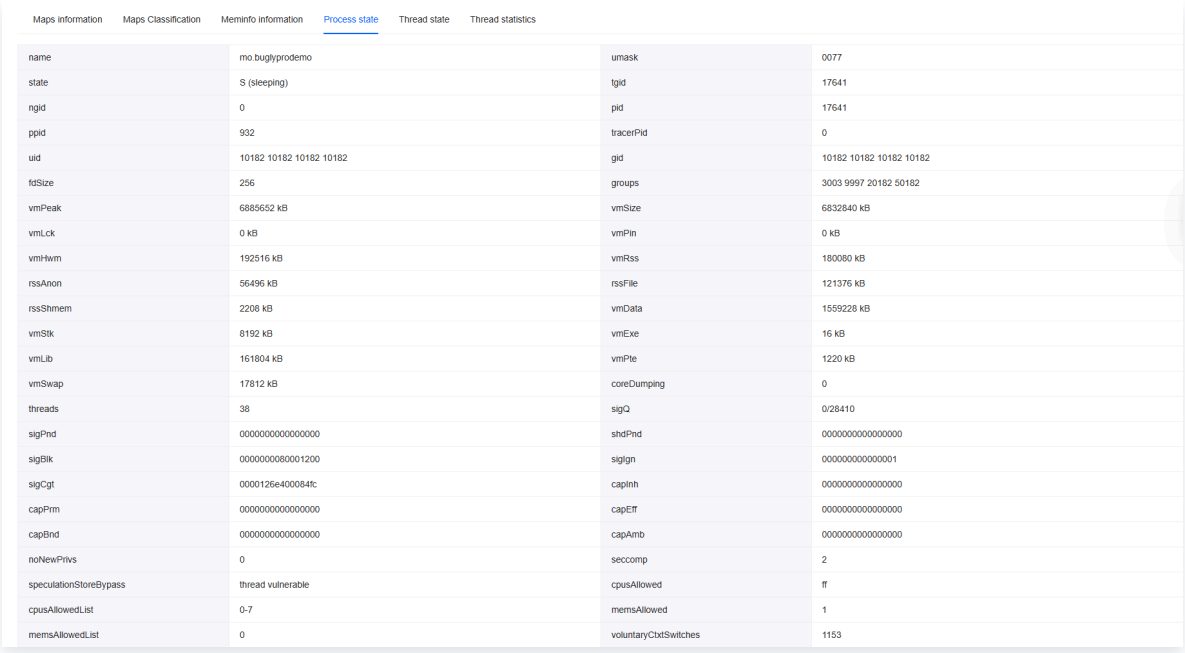
- **Maps information**: Extracted from **/proc/[pid]/maps**, commonly referred to as Maps information, it records memory mapping details for a specified process. This data reveals the process's virtual memory layout, including loaded libraries, stacks, shared memory regions, and more.



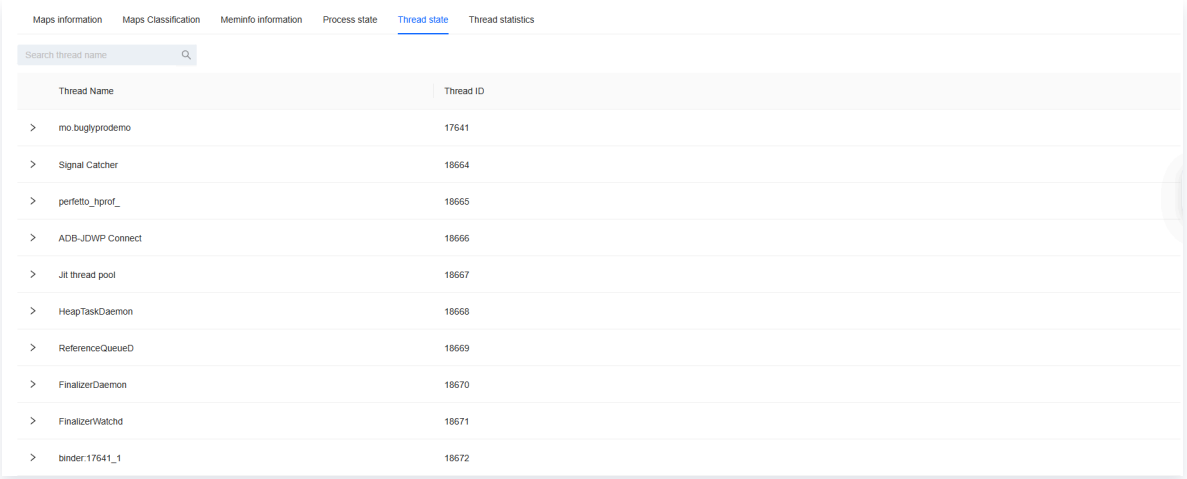
- **Maps Classification**: The classification of Maps information, grouping Maps data of the same type through a tree structure.



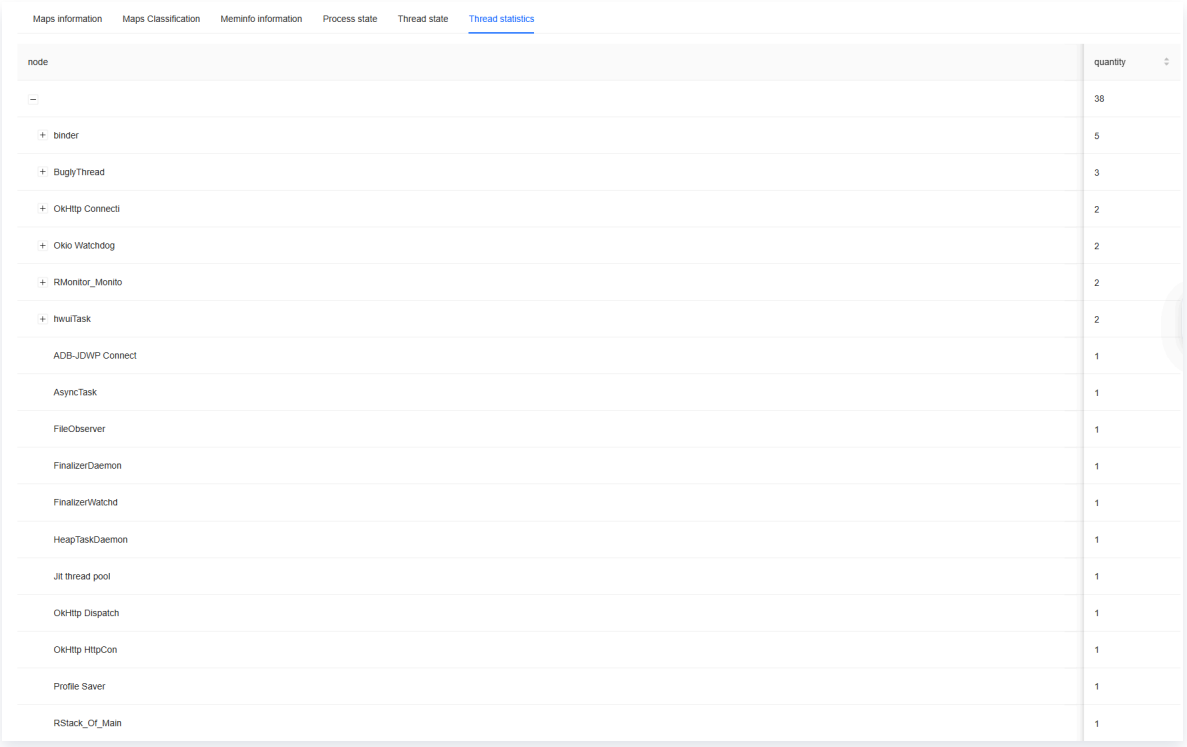
- **Process state**: Extracted from the /proc/[pid]/status file. For details, refer to the Linux kernel official documentation.



- **Thread state**: Extracted from the /proc/[pid]/task/[tid]/stat file, which records runtime statistical information of threads.

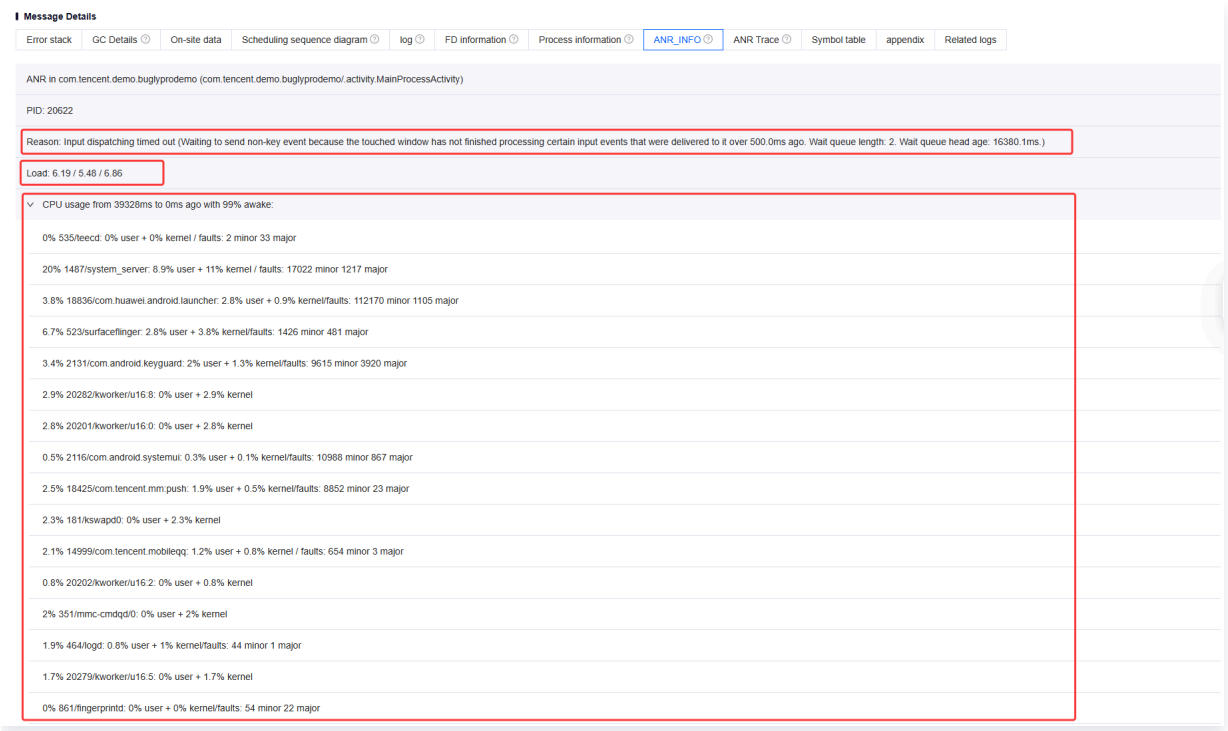


- **Thread statistics**: Categorized statistics of threads by thread name, presented in a tree structure.



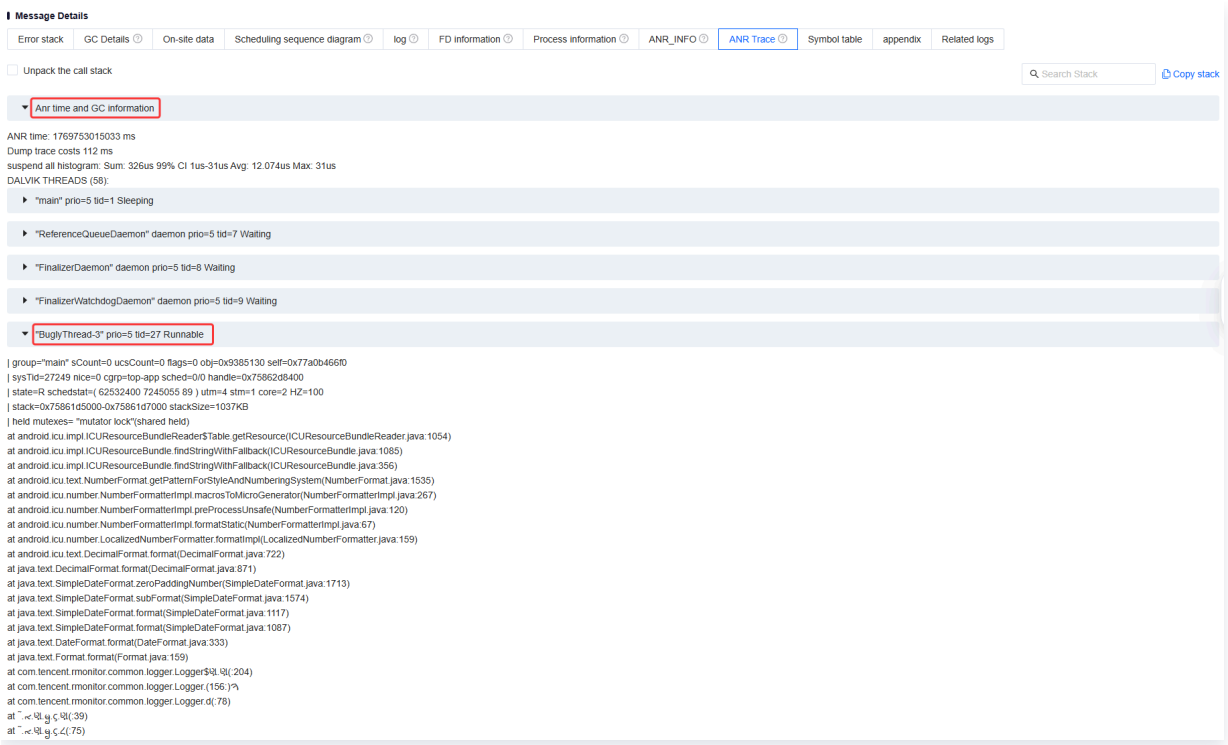
ANR_INFO

ANR_INFO is a system report generated by the application when an ANR occurs, recording the Activity component where the ANR happened, the Process ID of the affected process, and the cause of the ANR. Load represents the system's average load over 1, 5, and 15 minutes when an ANR occurs. The unit of average load is the number of processes, indicating the average count of processes running or waiting for CPU resources during the ANR. CPU Usage indicates detailed CPU utilization during the ANR timeframe. This information can be used to assist in analyzing ANR issues.



ANR Trace

ANR Trace is a log file that contains events and thread stack traces occurring during ANR in the application, mainly consisting of two parts, as shown in the figure below.
Part 1: ANR Time and GC Information. This section shows the duration of the current ANR issue and the GC operations performed by the system. Part 2: Thread Stack Information. This section lists stack trace information for all threads running during the ANR. In addition to stack details, each thread entry includes status information such as Thread ID (tid), Thread Priority (prio), Thread State (state), Thread Scheduling Statistics (schedstat), and CPU time used by the thread (utm and stm). Combining this information, especially the thread stack traces, can quickly help pinpoint and analyze the root causes of ANR issues.

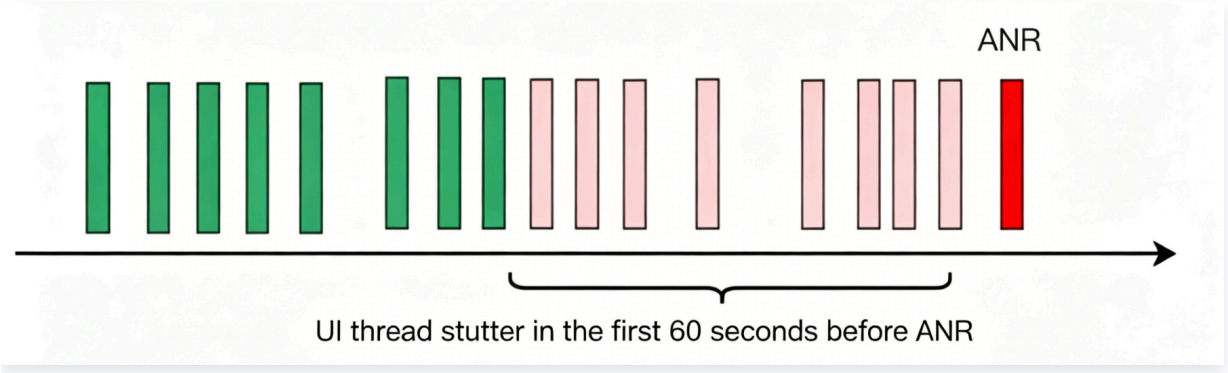


Drill-down analysis

Android ANR reporting content is similar to crash monitoring. You can refer to the crash drill-down analysis to view and analyze the data.

Associate ANR with Stutter List

Associated freeze: A freeze occurring within 60 seconds before an ANR is defined as an ANR-associated freeze.



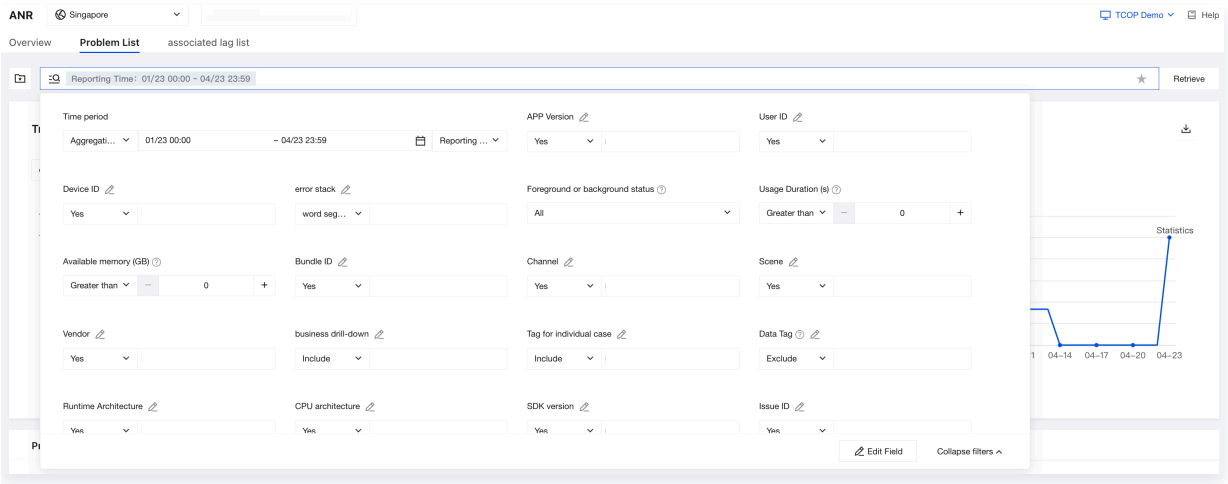
Querying

In the associated freeze list of ANR, RUM Pro supports comprehensive individual case query capabilities, primarily as follows:

- Supports drill-down search through multi-dimensional fields such as time range, App version, user ID, device ID, foreground/background state, and Issue ID.
- Supports search for custom issue types such as processing status, assignee, Tag, and tag colors.

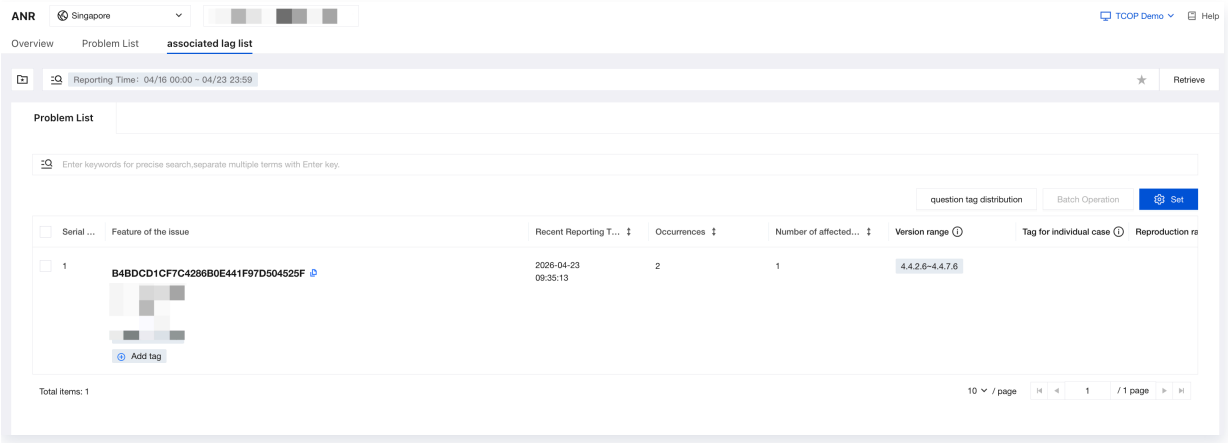
- Supports search for business-reported **Custom Fields**.

For detailed fields and operation types, see the Data Analysis – Query section.



Issue List

The reported content of the Android ANR associated lag list is similar to crash monitoring. You can refer to the Crash Problem List to view and analyze reported issues.



iOS

ANR (Deadlock)

ANR (Application Not Responding) is a term in the Android system that describes a situation where an application fails to respond to user input for a period of time. In the iOS system, it manifests as a SIGKILL with error code 0x8badf00d. On the platform, since it represents the same abnormal behavior, it is collectively referred to as ANR here, facilitating unified understanding across the platform and among developers.

ANR Metric Analysis

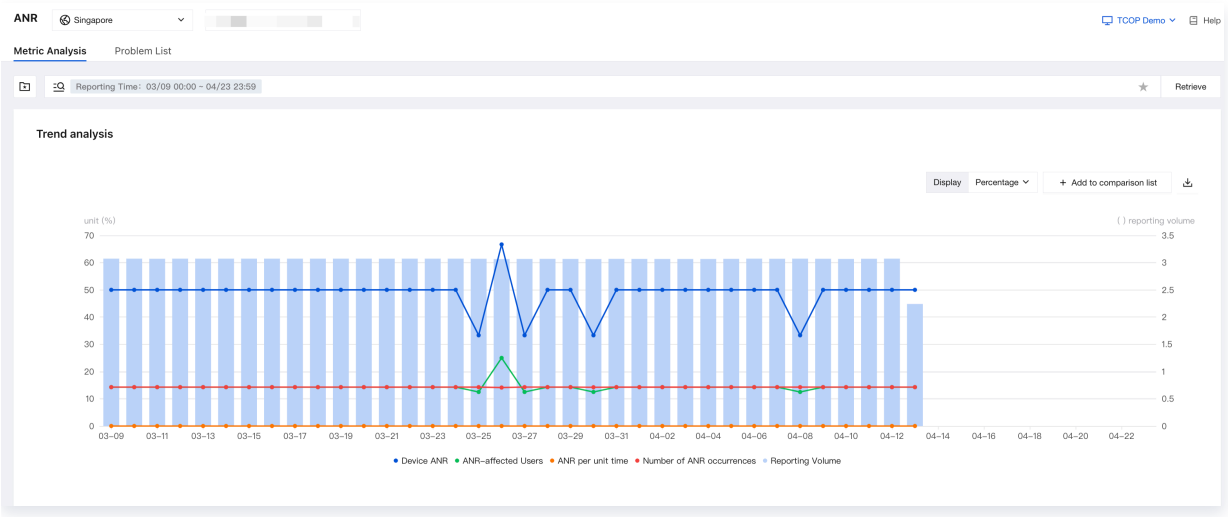
The essence of ANR issues lies in the main thread freeze, causing the App to time out in responding to user input. This triggers the system's WatchDog mechanism, thereby terminating the process. The SDK

determines whether the main thread has frozen by monitoring the execution cycle of the main thread's Runloop. For freezes exceeding the threshold (default: 5 seconds), it marks them as potential ANR events. Combined with App process exit determination, if the process remains in a potential ANR event state upon its last exit, the previous process termination is concluded to be an ANR.
Based on the above, the platform classifies exit scenarios during prolonged freeze states when a process is terminated by SIGKILL while the App is running in the foreground as ANR exceptions.

> **Note:**
> Since the SIGKILL signal directly terminates a process without any signal or notification to the process internally, the platform treats exits after excluding known exits (such as user manually stopping the process, other crashes, etc.) as SIGKILL, which, to some extent, is similar to the FOOM determination method.

To facilitate measuring the severity of ANR events, the platform provides the ratio of **ANR rate (ANR exit count** to **process launch count)**. Additionally, metrics measured from device and user perspectives are introduced, namely device ANR rate and user ANR rate, both of which are calculated after deduplication by device ID and user ID.

- **ANR Rate** = Exit Count / Launch Count
- **Device ANR** = Exit count after deduplication by device ID / Launch count after deduplication by device ID
- **ANR-affected Users** = Exit count after deduplication by user ID / Launch count after deduplication by user ID

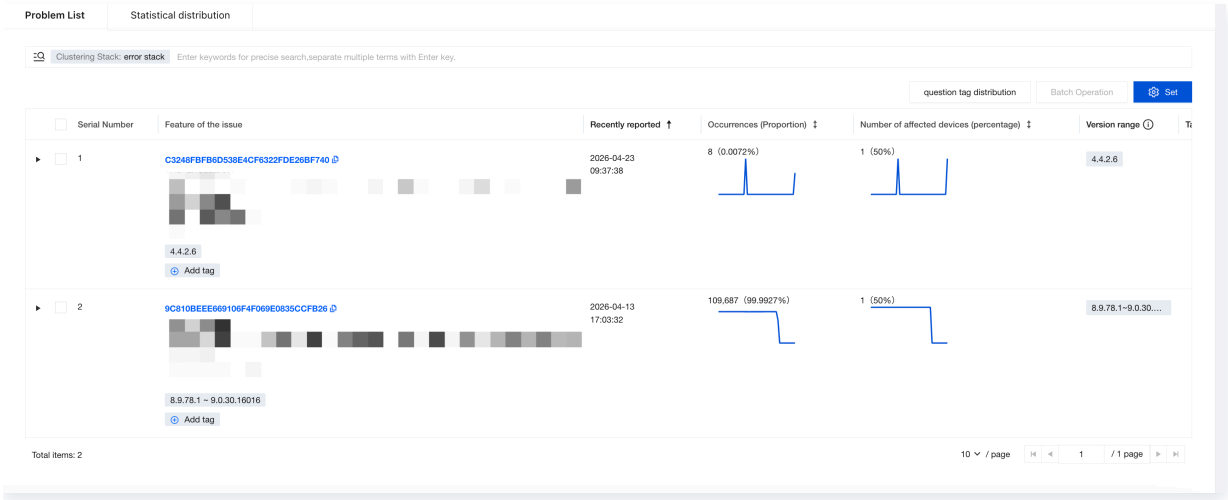


Similar to other metric analyses, it supports filtering and analysis under different conditions to observe metric changes and comparisons.

ANR Problem List

The platform clusters reported ANR cases by the characteristics of the call stacks that caused the exceptions, grouping similar issues into a single issue for statistical tracking. This facilitates business teams

in following up and locating problems. For each issue, the problem list displays metrics such as the number of affected devices and the most recent reporting time, consistent with the presentation of other issue lists.



Call stack feature extraction

As mentioned earlier, the SDK reports the call stacks causing ANRs to the backend. To effectively cluster individual issues, the platform uses **critical time-consuming methods** and takes **the top three functions in these critical time-consuming methods** as their signature.

Since the call stack captures execution states over a continuous period, adjacent identical frames can be merged from the bottom up, allowing the call stack to be transformed into a call tree where each node represents the time consumption of the corresponding method.

In the stack tree, identify a path with significant time consumption as the **critical time-consuming method**, then extract the top three levels of this path as its signature for clustering.

Issue list -- No stack issues

In the issue list, there exists a special issue category without a characteristic stack: "Stackless Issues". The occurrence of stackless issues is caused by the platform's sampling mechanism.

For individual cases where stack sampling is not enabled, although they cannot provide corresponding diagnostic stack information, there are scenarios requiring visibility into ANR occurrences for specific users or devices. Therefore, this issue list displays such ANR cases, all grouped under "Stackless Issues".

> **Note:**
> When the stack sampling rate is set to 100%, stackless issues may still be reported. This is a known platform issue, more prevalent in earlier versions. The cause is that the SDK experiences a slight delay when persisting the stack. If the process is terminated before stack persistence completes, the stack data is lost. The platform is actively optimizing this mechanism to reduce such occurrences.

ANR Case Analysis

Case analysis serves as a critical basis for resolving ANR issues, providing detailed root causes and effective diagnostic data to pinpoint problems. By selecting an issue from the problem list, users can go to its corresponding case list. Similar to other problem lists and case analyses, it displays basic case information such as occurrence time, reporting time, and user/device IDs, along with relevant filtering criteria. Drill-down analysis is also available, showing trends in case reporting and version distribution for the specific issue.
The message details of individual issues provide information such as the stack trace, full thread dump, operation logs, symbol tables, and scene data. Among these, operation logs, symbol tables, and scene data are consistent with the message details of other issues. For reference, see crash issue details. This section focuses on explaining the stack trace and full thread dump.

error stack

The stack trace refers to the periodically collected **main thread call stack** mentioned earlier, presented in three visualization formats: time slices, stack tree, and flame graph. These representations are derived from the same dataset but displayed differently for enhanced analysis.

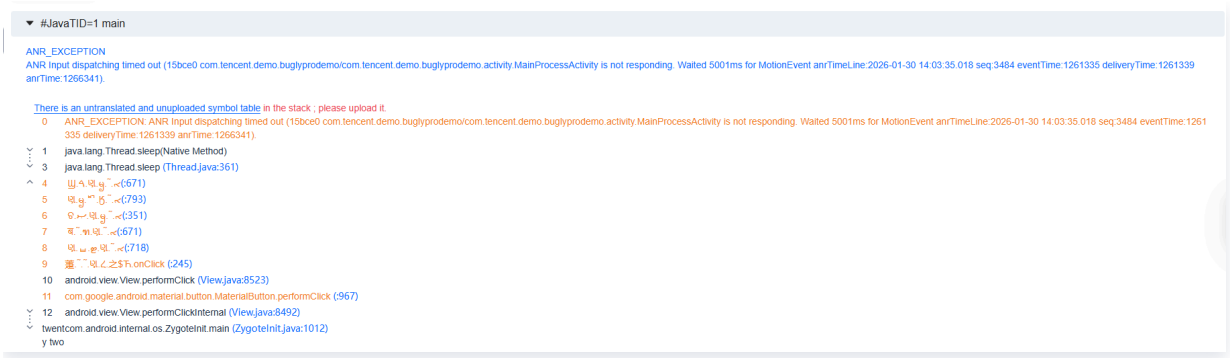
- Time slices: The raw data collected is displayed sequentially according to the collection time intervals. Each "TimeSlice: xx~xx" label indicates the call stack details within the corresponding collection interval, which can be viewed by clicking. During collection and reporting, identical call stacks within the same time slice are merged and displayed accordingly. For example, "TimeSlice: 10~20" indicates that the call stack remains consistent throughout the 10–20 time interval, meaning the same logic was continuously executed.

  The time intervals shown are relative values, representing the sequence of collections within the final collection window. Since reported stacks are collected within the last 5 seconds before a suspected ANR occurrence, serial numbers follow an "earliest to latest" order. For instance, serial number 1 represents the earliest collected data retained at the time of ANR determination, with subsequent numbers following this sequence.
- Stack tree: As a more effective representation, the stack tree aggregates time slice data from the bottom up to form a call tree. Each node represents the execution time of the corresponding method, allowing for intuitive identification of bottleneck methods and their duration.
- Flame graph: a graphical representation of the stack tree.

The stack trace directly reveals the timing and methods causing the freeze. For most issues, this typically identifies the method responsible for the abnormal behavior. However, since the stack trace only captures the main thread's state, some ANR issues may stem from the main thread being blocked while waiting for a lock. In such cases, identifying where the lock is held becomes crucial for root cause analysis. This is where the full thread dump comes into play.

Full thread dump

The full thread dump captures the call stacks of all threads **when a suspected ANR is detected**, including the main thread itself. Its purpose is to assist developers in identifying lock contention issues. Generally, if the main thread freezes due to lock contention, developers can trace identical lock usage patterns in the call stack to check whether other threads hold or are waiting for the same lock. This helps determine whether issues like deadlocks or prolonged lock waits have occurred.



FOOM/OOM

Last updated：2026-05-25 18:53:26

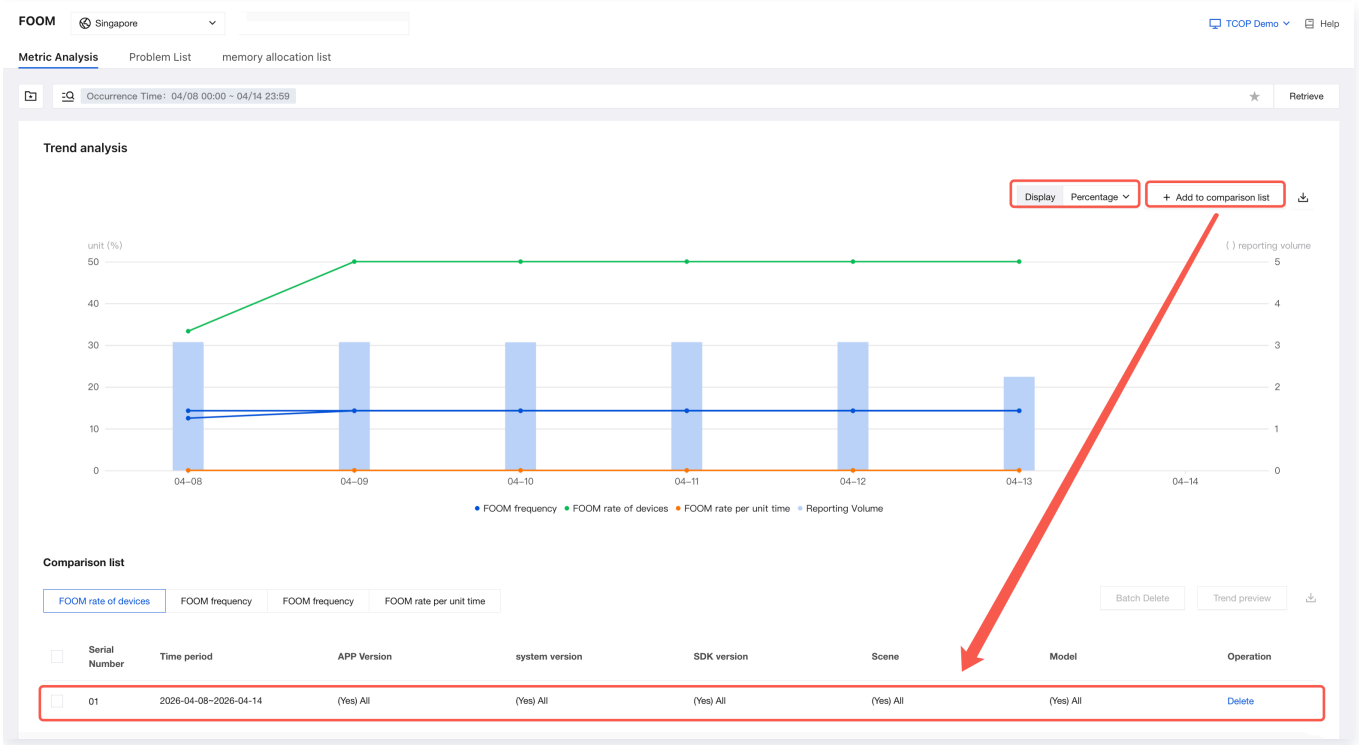
FOOM(iOS)

FOOM (Foreground Out Of Memory) occurs when an APP encounters the system memory limit in the foreground and is terminated by the SIGKILL signal. This article introduces various metric analyses related to FOOM reporting and explains how to view detailed issue reports.

Metric Analysis

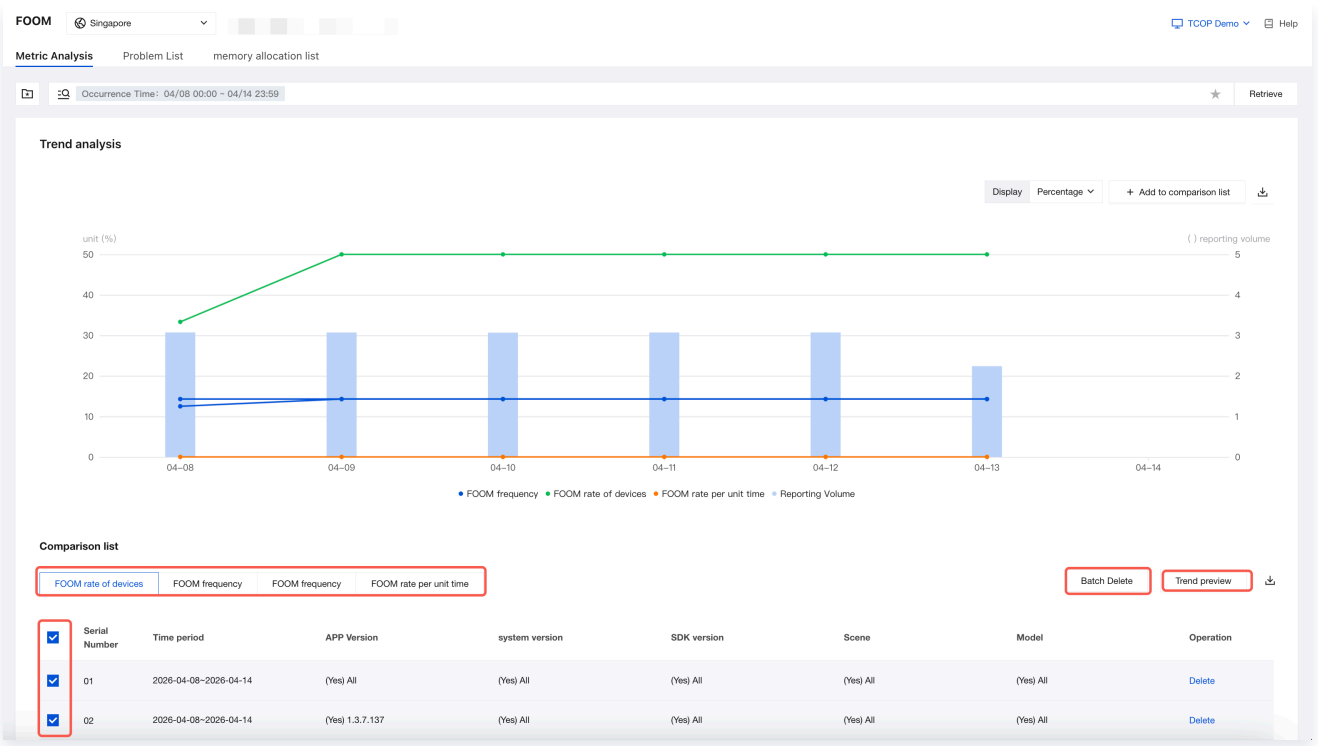
Trend analysis

Trend charts display the changes in different types of FOOM rates (e.g., device FOOM rate, user FOOM rate) over time using line graphs or similar formats, with units expressed as (%). You can adjust the display format via the **Display** options or click **Add to comparison list** to include different data series for comparison.

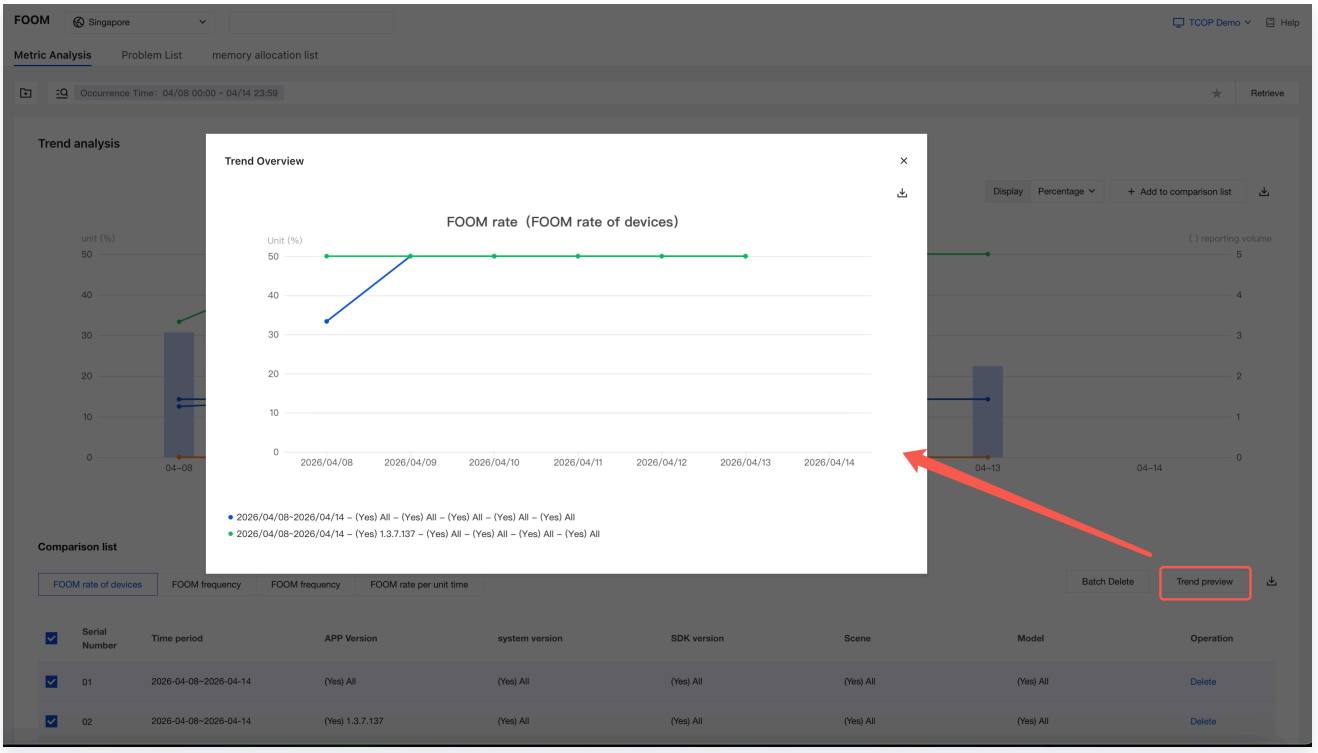


In the comparison list area, you can view FOOM-related data details under different filter conditions, including index, time range, APP version, and other information. You can also perform operations such as delete, batch delete, trend preview, and more.

- After selecting one or more data items to be deleted, click **Delete** or **Batch Delete** to complete the deletion.



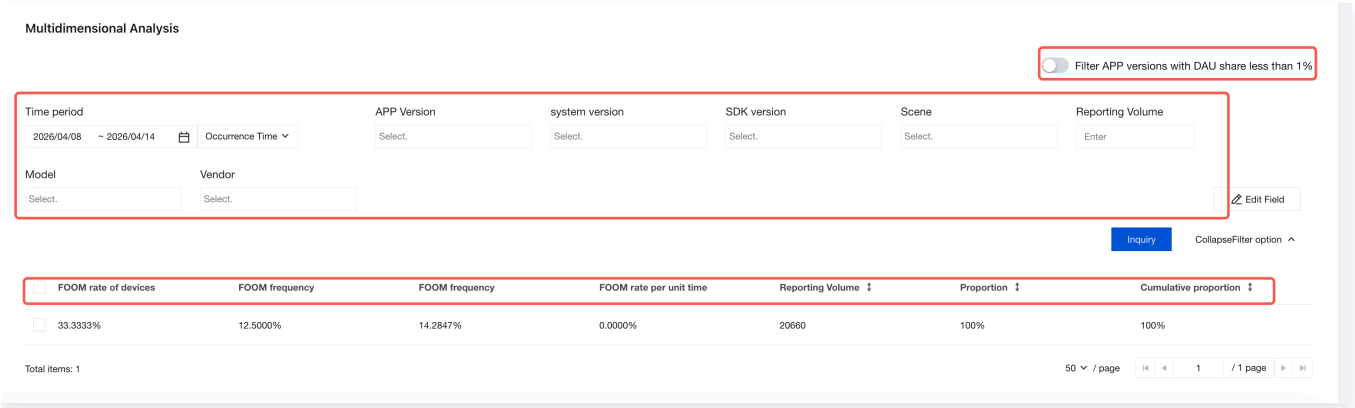
- Click **Trend preview** to view the overview of selected FOOM rates in the comparison list.



Multidimensional Analysis

- Filter condition settings: You can set filter conditions such as time range (default or custom), APP version, system version, SDK version, scenario, report volume, device model, manufacturer, etc. You can also choose whether to filter out APP versions with DAU accounting for less than 1%. After completing the settings, click **Inquiry** to obtain the filtered data. To collapse the filter conditions, click **CollapseFilter option**.

- Data display: This area presents FOOM-related metrics across multiple dimensions, including device FOOM rate, user FOOM rate, occurrence FOOM rate, FOOM rate per unit time, report volume, proportion, and cumulative proportion, helping users analyze FOOM metrics from different perspectives.

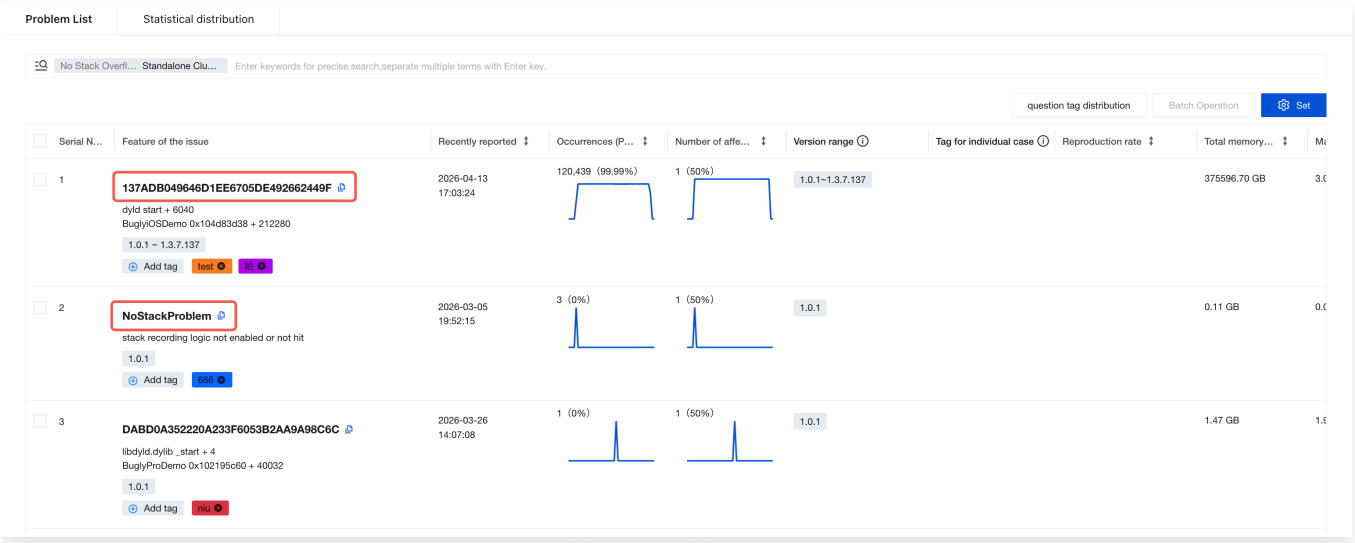


Problem List

The FOOM problem list is similar to the crash monitoring module. For details, see Crash Problem List.

Case clustering

The platform clusters reported individual cases into issues to facilitate problem tracking and resolution by developers. Currently, the platform clusters cases based on allocation stack information extracted from FOOM reports. Due to limitations in FOOM's stack collection policy, not all reported cases contain stack information. Cases without stack data are grouped into a special issue category: **NoStackProblem**. For cases with stack information, clustering is performed by extracting features from the stack traces to form characteristic clusters.

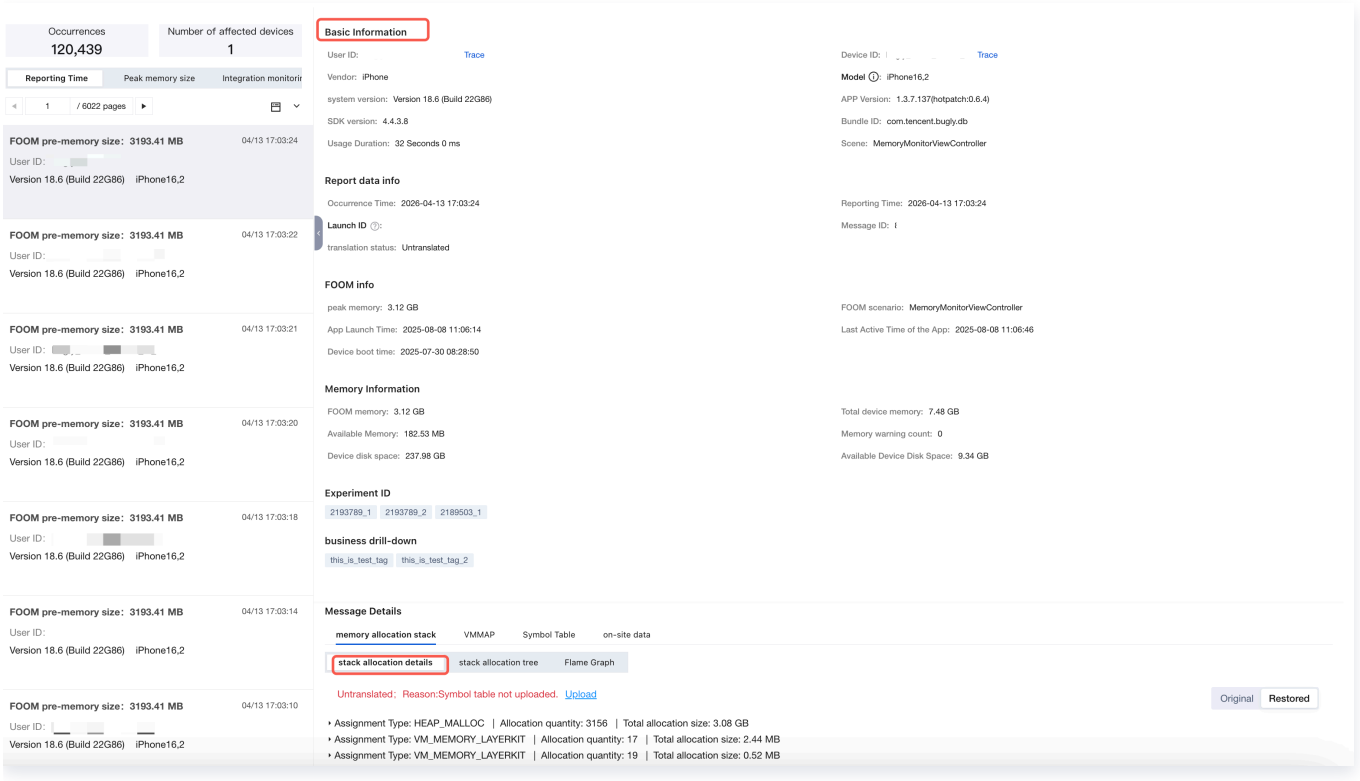


> **Note:**
> Due to the low proportion of enabled stack traces, the vast majority of individual cases are grouped into the **NoStackProblem** category.

Issue Details

Case Details

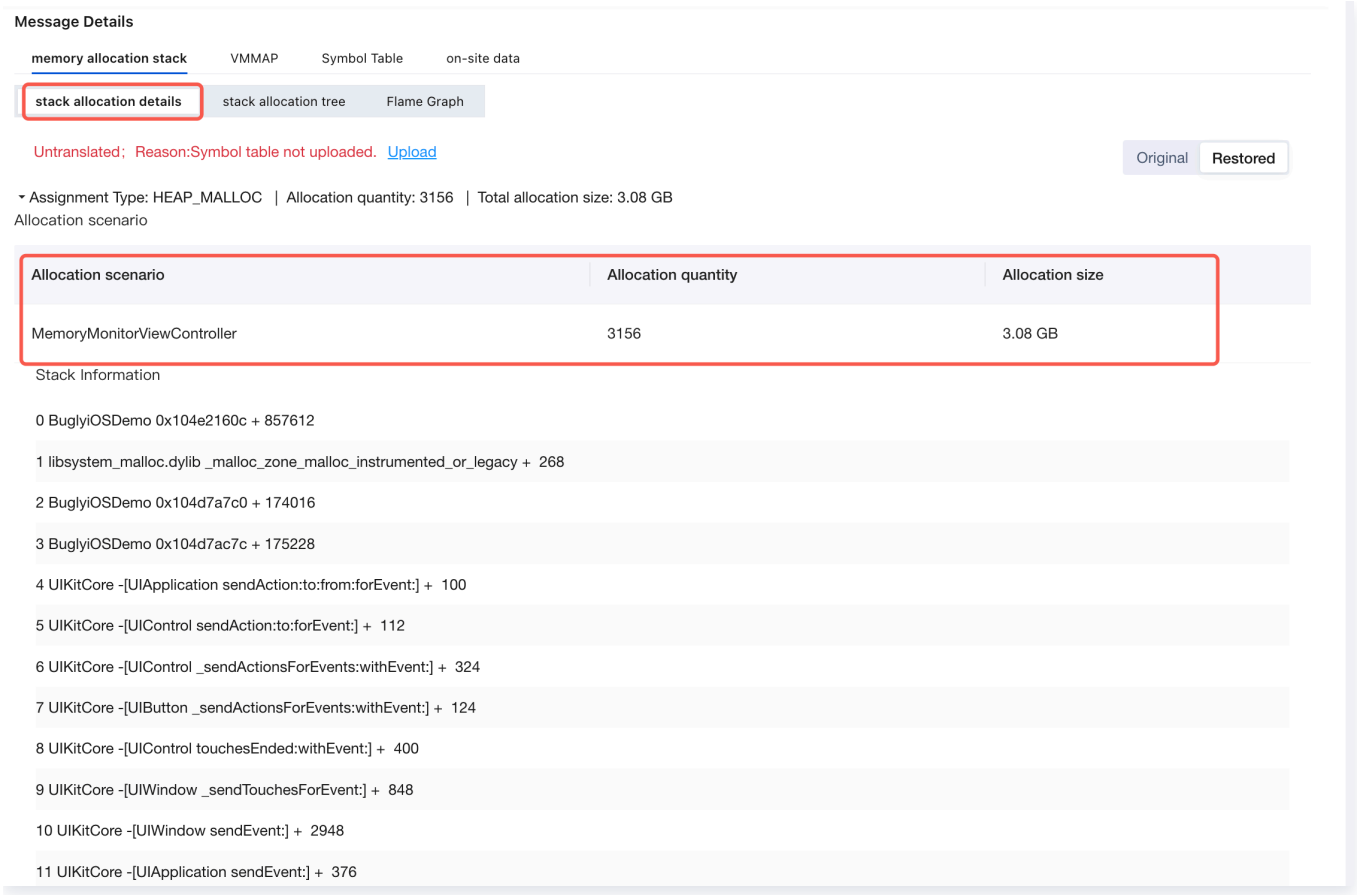
From **FOOM** > **Problem List** go to an issue, which displays all individual cases under that issue. Besides basic information, the most critical part of individual cases is the **stack allocation details** data.



memory allocation stack

Stack Allocation Details

The data in the stack allocation details is the malloc logger information recorded by the SDK during the APP's runtime.
In the malloc logger records, entries are aggregated by the same allocation type and allocation stack, displaying the corresponding count and total size of stack allocations.



Note:

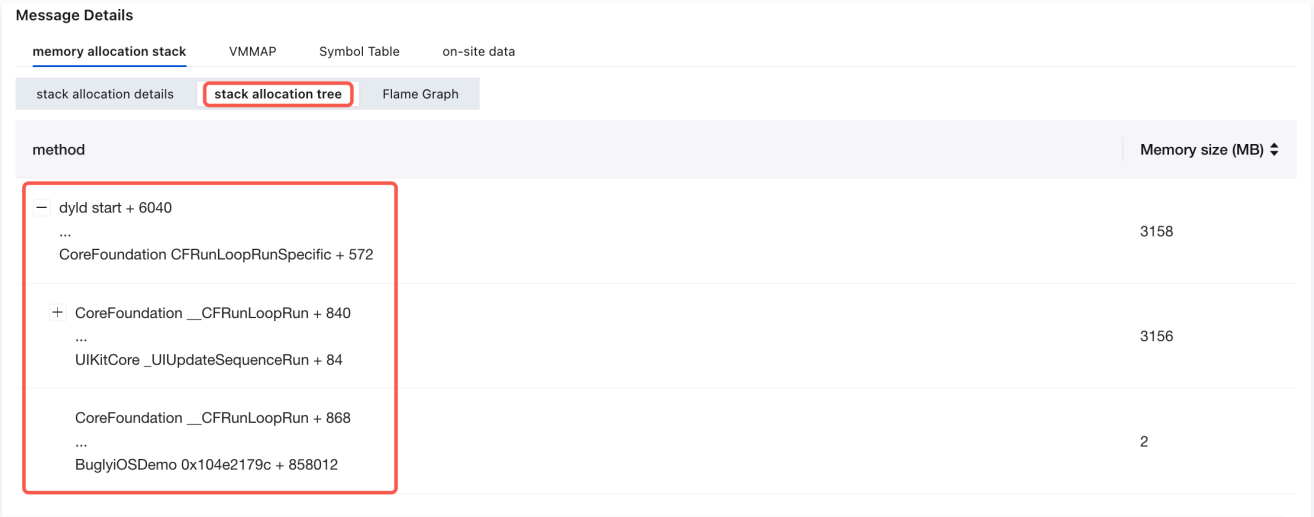
The stack here does not include all memory allocations, only memory operations that were recorded as allocated but not released under the current policy. Allocations not covered by the recording policy will not be included. Under the default recording policy, only allocations meeting all the following conditions are recorded:

- A single allocation exceeds the threshold; the default threshold is 8K, which can be adjusted via configuration.
- The cumulative allocation of the same stack exceeds the threshold; the default threshold is 512K, which can be adjusted via configuration.
- Allocations made before stack allocation records are enabled will not be included: Memory allocated before the recording logic starts will not be included in the records.

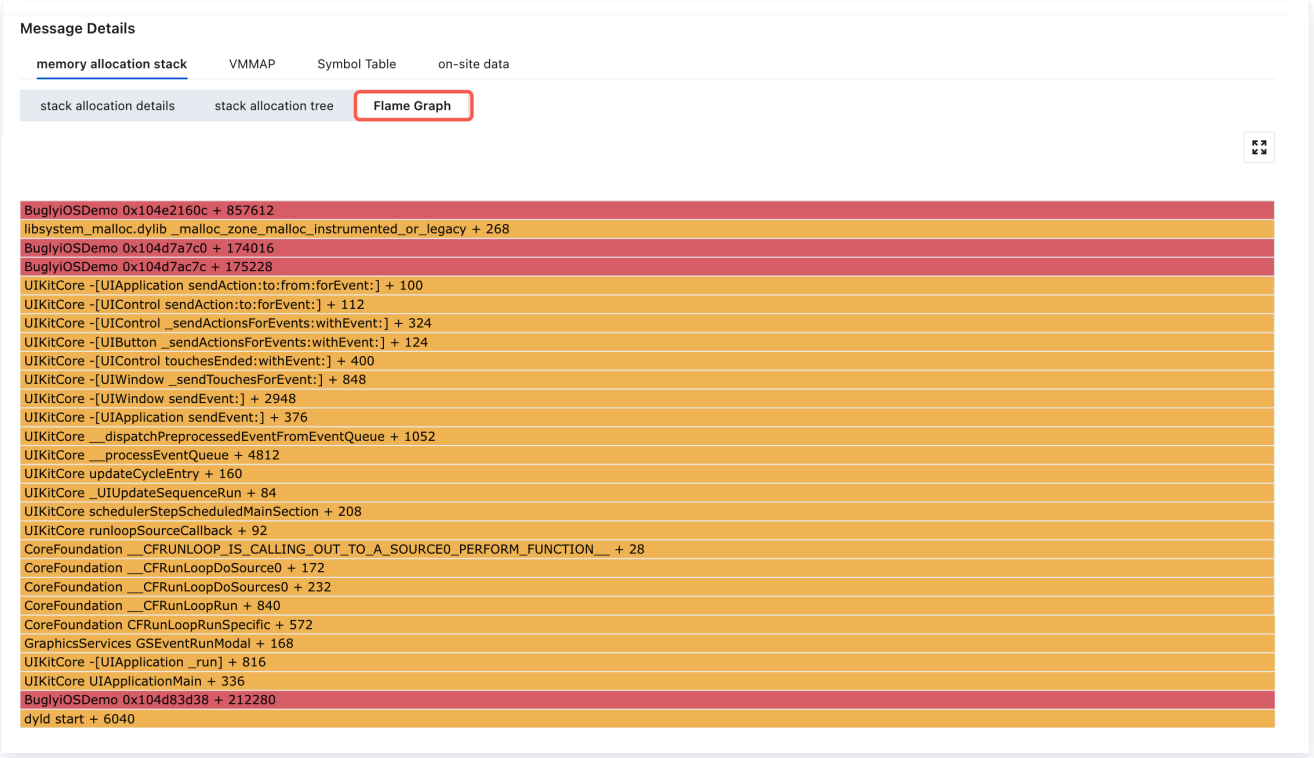
Stack Allocation Tree, Flame Graph

Stack allocation trees and flame graphs are data aggregated from the stack data in **stack allocation details**. They do not carry special meanings but provide a more intuitive way to visually identify the proportion of memory allocations by aggregating data into a tree structure.

- Stack allocation tree

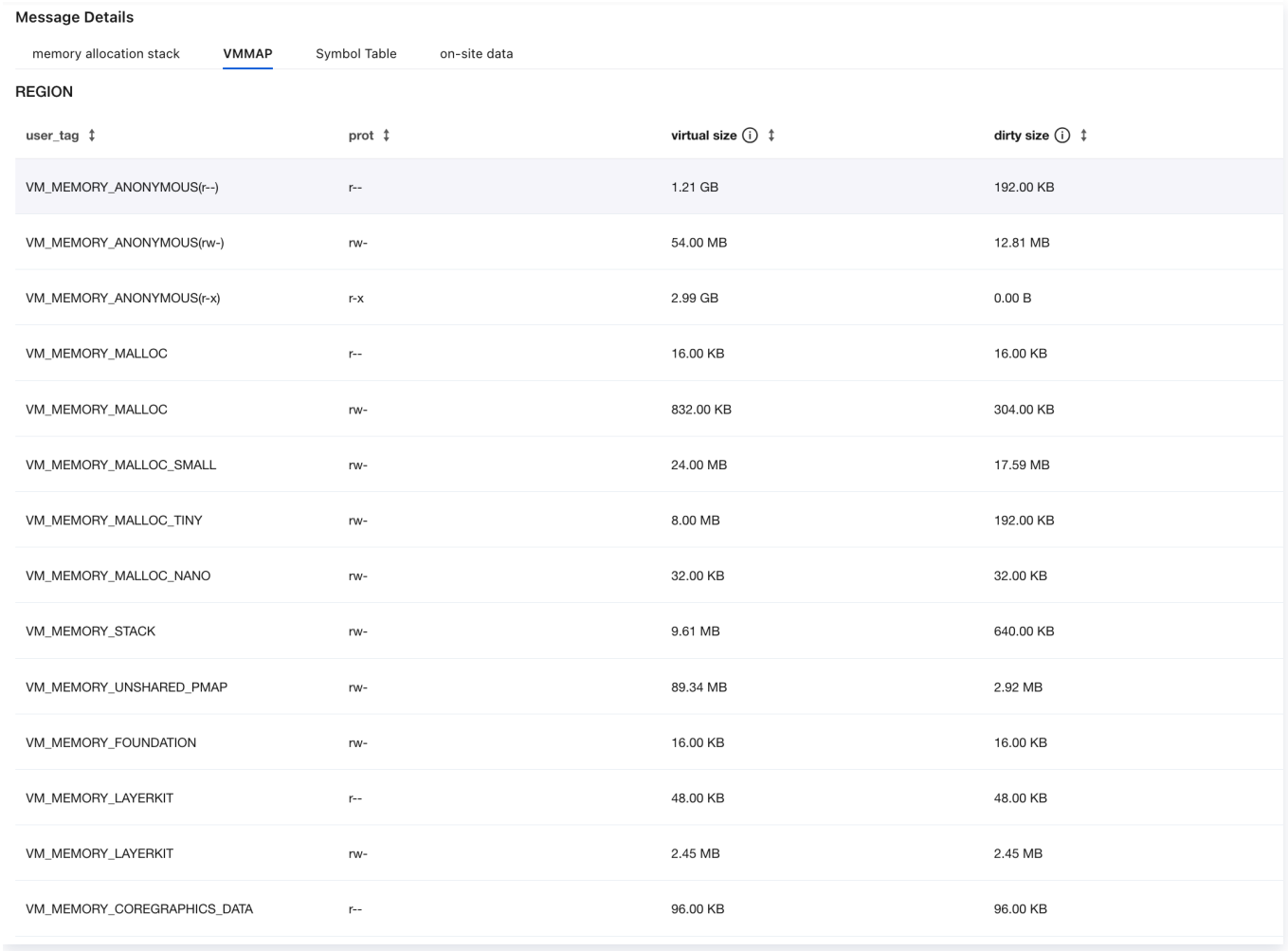


- Flame Graph



VMMAP

VMMAP refers to the memory allocation table of the Virtual Memory Manager (VMM), which records allocation information for all memory regions, including stack memory.



Message Details

memory allocation stack | VMMAP | Symbol Table | on-site data

REGION

user_tag	prot	virtual size	dirty size
VM_MEMORY_ANONYMOUS(r--)	r--	1.21 GB	192.00 KB
VM_MEMORY_ANONYMOUS(rw-)	rw-	54.00 MB	12.81 MB
VM_MEMORY_ANONYMOUS(r-x)	r-x	2.99 GB	0.00 B
VM_MEMORY_MALLOC	r--	16.00 KB	16.00 KB
VM_MEMORY_MALLOC	rw-	832.00 KB	304.00 KB
VM_MEMORY_MALLOC_SMALL	rw-	24.00 MB	17.59 MB
VM_MEMORY_MALLOC_TINY	rw-	8.00 MB	192.00 KB
VM_MEMORY_MALLOC_NANO	rw-	32.00 KB	32.00 KB
VM_MEMORY_STACK	rw-	9.61 MB	640.00 KB
VM_MEMORY_UNSHARED_PMAP	rw-	89.34 MB	2.92 MB
VM_MEMORY_FOUNDATION	rw-	16.00 KB	16.00 KB
VM_MEMORY_LAYERKIT	r--	48.00 KB	48.00 KB
VM_MEMORY_LAYERKIT	rw-	2.45 MB	2.45 MB
VM_MEMORY_COREGRAPHICS_DATA	r--	96.00 KB	96.00 KB

For memory issues outside the above recording policy, stack allocation details may not directly provide effective information. Therefore, VMMAP data was subsequently introduced. VMMAP data, similar to the output of the `vmmap` command, provides usage information of kernel-managed memory, including details such as `user_tag`, `virtual_size`, and `dirty_size`.

- user_tag: Corresponding to "allocation type" in the stack allocation details.
- virtual_size: Indicates the logical address size used by this type and does not represent the actual physical memory occupied.
- dirty_size: Indicates the memory of this type that cannot be reclaimed by the operating system, occupying actual physical memory.

The SDK periodically collects this data, and the collection frequency is correlated with memory metrics. Therefore, the VMMAP information obtained reflects the peak memory state, essentially including all memory usage. Due to the collection frequency, it may not be consistent with the final peak memory.
Through VMMAP information, you can get a general idea of the main memory usage, and further analysis can be performed by combining other factors.

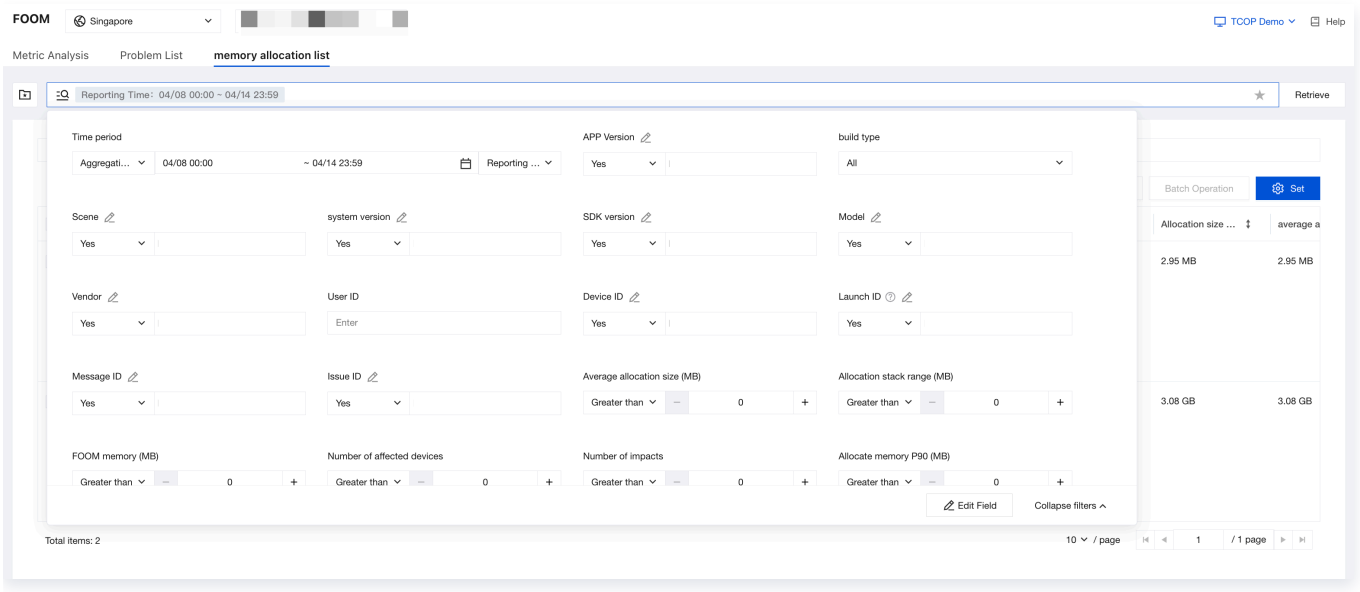
Memory allocation list

The FOOM Memory Allocation List feature is primarily used to query and manage issues related to memory allocation, helping developers or Ops personnel quickly locate and analyze abnormal situations in memory

allocation, such as excessive memory allocation, affecting a large number of devices, etc., so as to handle them promptly and ensure the stable operation of applications.

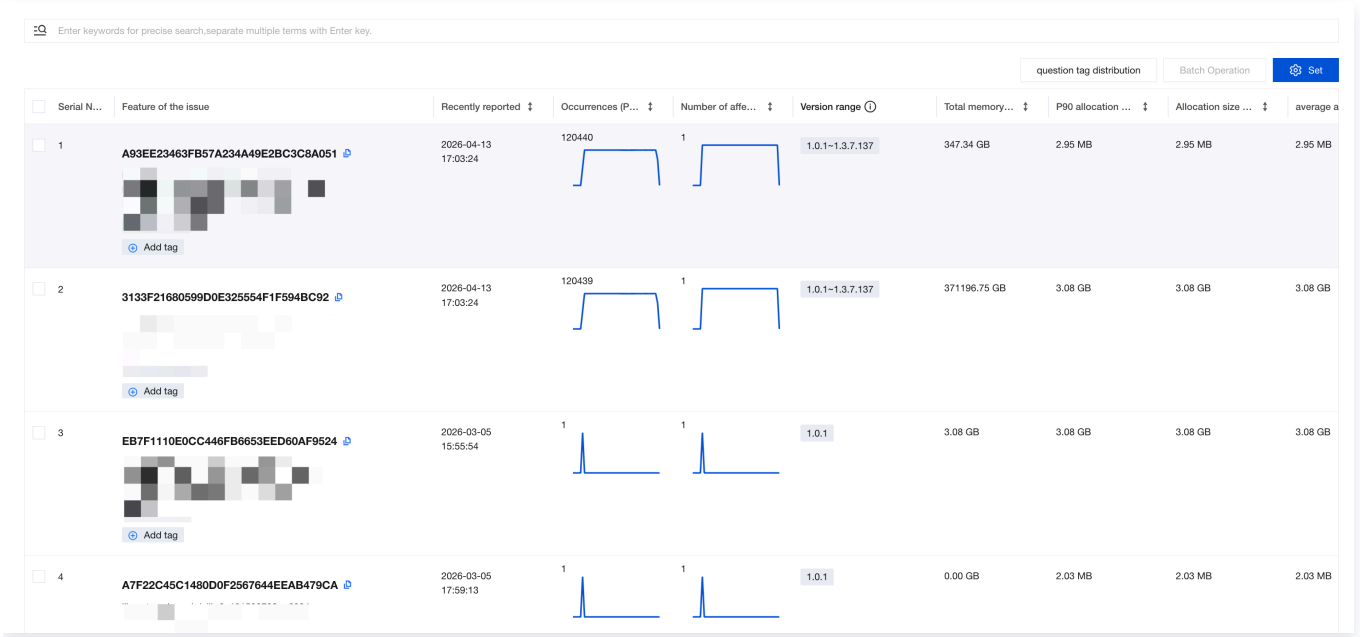
Query

The feature supports users in querying memory allocations via multiple filter options. For details, see Query.



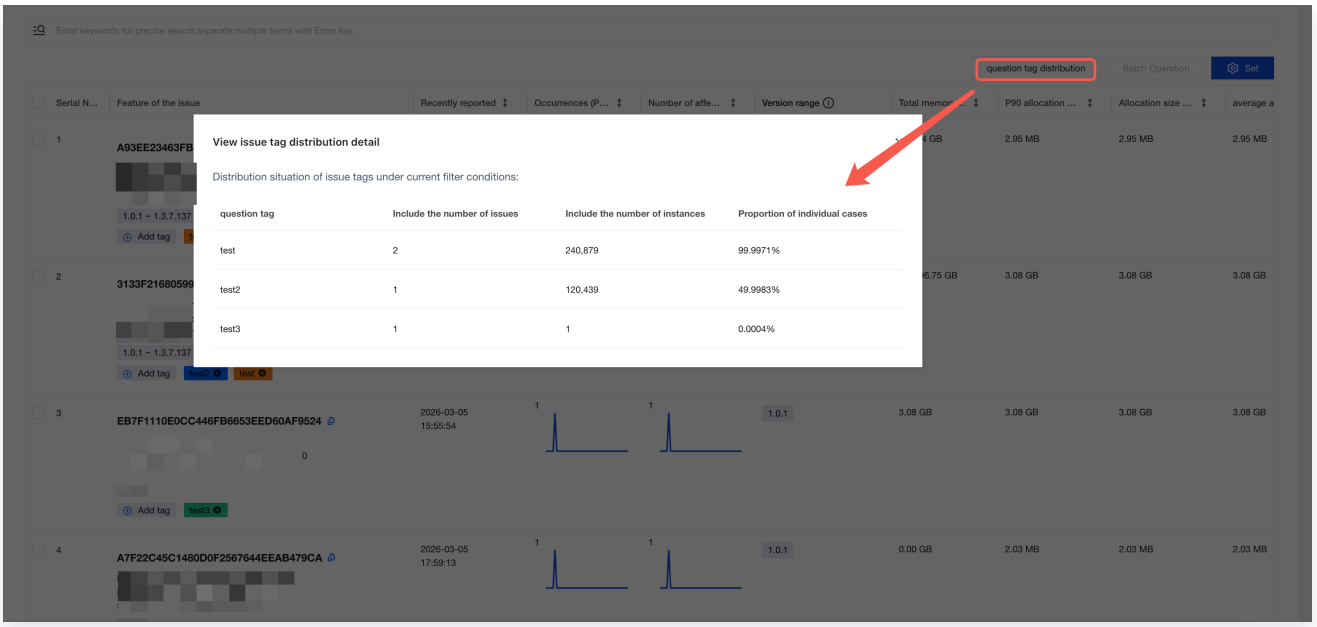
FOOM Allocation List

After a user query, the results will be displayed in the **FOOM Allocation List**, including information such as issue characteristics, last reported time, number of affected devices, report count, and total memory allocation size.

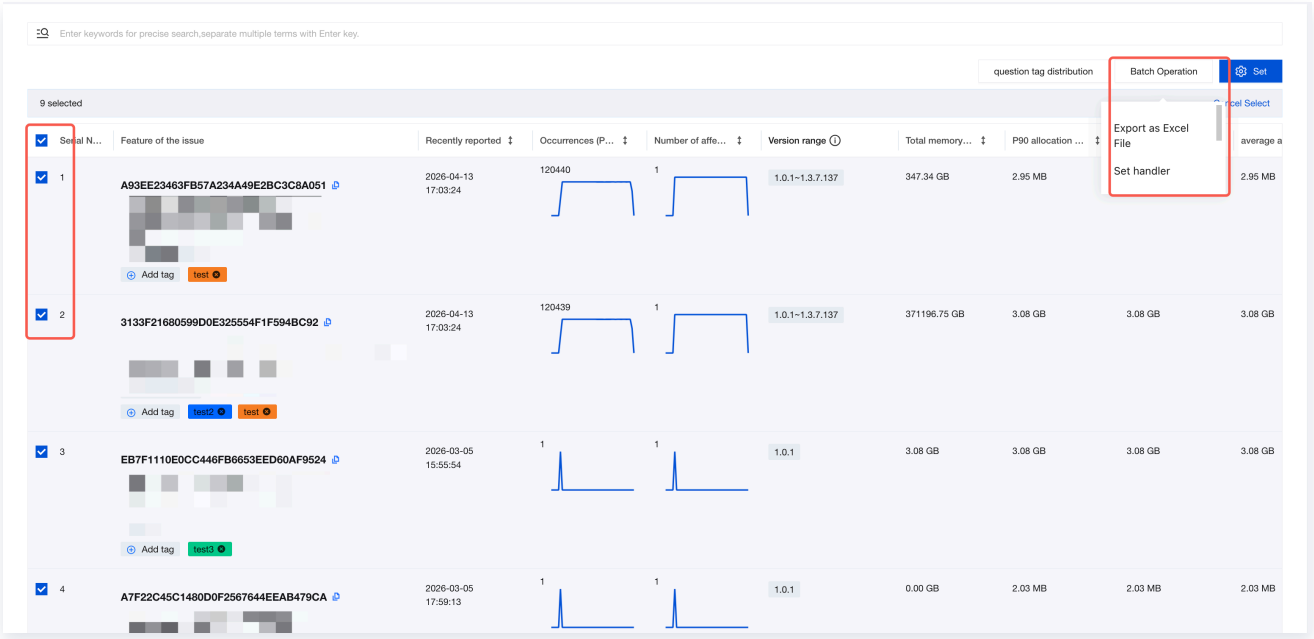


Meanwhile, the following operations can be performed:

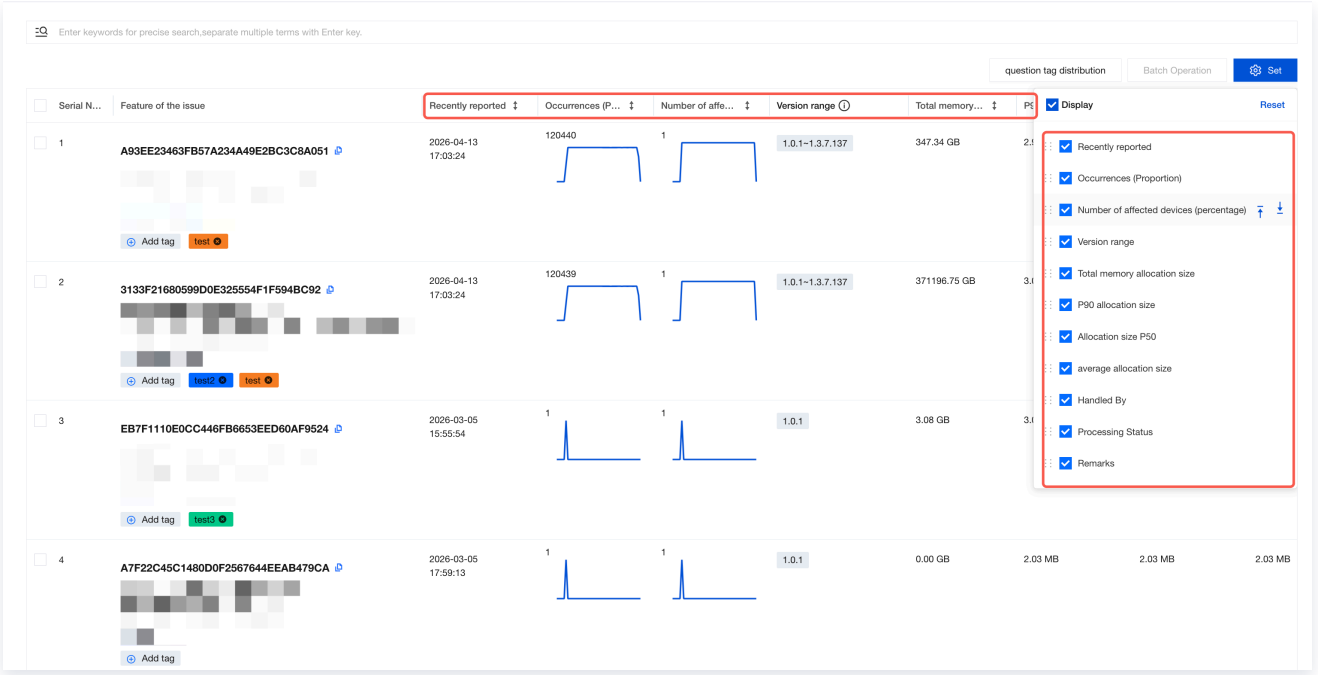
- **question tag distribution**: View the distribution of different issue tags to facilitate statistics and analysis of problem types.



- **Batch Operation**: Perform batch processing for multiple eligible data items to improve work efficiency.



- **Set**: Allows configuring related settings for list display, etc., to meet personalized needs.



OOM(Android)

Background

In Android development, OOM (Out of Memory) issues typically refer to java.lang.OutOfMemoryError exceptions caused by insufficient Java heap memory. However, OOM problems are not limited to Java heap memory and may also involve exceeding resource limits for file descriptors (FD) or native memory address space, as exemplified by the following common java.lang.OutOfMemoryError exception:

```
java.lang.OutOfMemoryError: Could not allocate JNI Env
java.lang.Thread.nativeCreate(Native Method)
java.lang.Thread.start(Thread.java:1063)
kotlinx.coroutines.scheduling.CoroutineScheduler.int createNewWorker()
(CoroutineScheduler.java:485)
kotlinx.coroutines.scheduling.CoroutineScheduler.boolean tryCreateWorker
(long)(CoroutineScheduler.java:440)
kotlinx.coroutines.scheduling.CoroutineScheduler.boolean tryCreateWorker
$default(kotlinx.coroutines.scheduling.CoroutineScheduler,long,int,java.
lang.Object)(CoroutineScheduler.java:431)
kotlinx.coroutines.scheduling.CoroutineScheduler.void signalCpuWork()
(CoroutineScheduler.java:427)
kotlinx.coroutines.scheduling.CoroutineScheduler$Worker.void
beforeTask(int)(CoroutineScheduler.java:758)
kotlinx.coroutines.scheduling.CoroutineScheduler$Worker.void
executeTask(kotlinx.coroutines.scheduling.Task)
(CoroutineScheduler.java:749)
```

```
kotlinx.coroutines.scheduling.CoroutineScheduler$Worker.void runWorker()
(CoroutineScheduler.java:678)
kotlinx.coroutines.scheduling.CoroutineScheduler$Worker.void run()
(CoroutineScheduler.java:665)
```

While it appears to be a virtual machine memory-related issue, from a source code perspective, it actually occurs because there are no available FD resources when creating a thread, resulting in thread creation failure. Consequently, it is reported as a java.lang.OutOfMemoryError exception. This can also be verified from system logs, which show numerous "Too many open files" entries prior to the error:

```
03-18 13:44:17.426 28516
809 W System.err: java.net.ConnectException: failed to connect to up-
hl.3g.qq.com/61.241.53.46 (port 443) after 10000ms: connect failed: EMFI
LE (Too many open files)
03-18 13:44:17.427 28516
809 W System.err: Caused by: android.system.ErrnoException: connect fail
ed: EMFILE (Too many open files)
03-18 13:44:17.519 28516
765 W System.err: java.io.FileNotFoundException: /data/user/0/******.tem
p: open failed: EMFILE (Too many open files)
03-18 13:44:17.519 28516
765 W System.err: Caused by: android.system.ErrnoException: open failed:
 EMFILE (Too many open files)
03-18 13:44:18.239 28516 802 E art : ashmem_create_region failed for
'indirect ref table': Too many open files
03-18 13:44:18.243 28516
801 W System.err: java.net.SocketException: socket failed: EMFILE (Too m
any open files)
03-18 13:44:18.243 28516
801 W System.err: Caused by: android.system.ErrnoException: socket faile
d: EMFILE (Too many open files)
```

In addition to FD resources, insufficient Native memory can also cause thread creation failures. Crash issues caused by exceeding resource limits exhibit diverse stack traces, where the Crash stack is often merely the last straw that breaks the camel's back. Conventional clustering methods based on Crash stacks and exception types thus often yield poor results. For such issues, it is necessary to categorize them from the perspective of resource usage and implement targeted optimizations and solutions. To comprehensively understand and resolve business-related OOM issues, we have reclassified OOM into Java OOM, FD OOM, and Native OOM, while expanding the original definition of the OOM rate metric.

OOM Issue Classification

- Java OOM

  Java OOM is the most common type of OOM issue, typically caused by insufficient Java heap memory. When an application requests more memory than the available space in the Java heap, it throws a java.lang.OutOfMemoryError exception. This may result from Java memory leaks, large object allocations, large bitmaps, or similar factors.

```
java.lang.OutOfMemoryError
Failed to allocate a 176 byte allocation with 5025912 free bytes and 4
908KB until OOM, target footprint 536870912, growth limit 536870912; g
iving up on allocation because <1% of heap free after GC.

java.lang.OutOfMemoryError: Failed to allocate a 176 byte allocation w
ith 5025912 free bytes and 4908KB until OOM, target footprint 53687091
2, growth limit 536870912; giving up on allocation because <1% of heap
 free after GC.
java.util.Arrays.copyOf(Arrays.java:3766)
java.lang.AbstractStringBuilder.ensureCapacityInternal(AbstractStringB
uilder.java:125)
java.lang.AbstractStringBuilder.append(AbstractStringBuilder.java:449)
java.lang.StringBuilder.append(StringBuilder.java:137)
......
```

- FD OOM

  FD OOM refers to OOM issues caused by exceeding file descriptor resource limits. Each process has a limited number of file descriptors available during runtime, which are used to open, read, and write files.

```
*** *** *** *** *** *** *** *** *** *** *** *** *** *** *** ***
Crash type: 'native'
Start time: '2024-03-18T12:32:49.406+0800'
Crash time: '2024-03-18T16:09:33.763+0800'
App version: 'x.x.x.x'
Rooted: 'No'
API level: '27'
Build fingerprint: 'OPPO/R11/R11:8.1.0/OPM1.171019.011/1575877917:user
/release-keys'
ABI: 'arm64'
```

```
pid: 9096, tid: 18434, name: HalleyTempTaskT  >>> com.tencent.xxxxx <<
<
signal 6 (SIGABRT), code -6 (SI_TKILL), fault addr --------
Abort message: 'Could not make wake event fd: Too many open files'
    x0  0000000000000000  x1  0000000000004802  x2  0000000000000006
x3  0000000000000008
    x4  0000007dce36f588  x5  0000007dce36f588  x6  0000007dce36f588
x7  0000007dce36f588
    x8  0000000000000083  x9  0000000010000000  x10 0000007dce36e9b0
x11 cd4becaebf0bf9bc
    x12 cd4becaebf0bf9bc  x13 0000000000000020  x14 ffffffffffffffdf
x15 cd4becaebf0bf9bc
    x16 0000000a2bfd9fa8  x17 0000007ed34b7540  x18 0000000000000001
x19 0000000000002388
    x20 0000000000004802  x21 0000000000000083  x22 000000007018bbc0
x23 0000000000004802
    x24 0000000000000001  x25 00000000701d41f0  x26 0000000015f00088
x27 0000000015f000b0
    x28 0000000015f00100  x29 0000007dce36e9f0
    sp  0000007dce36e9b0  lr  0000007ed3460770  pc  0000007ed3460798

backtrace:
    #00 pc 000000000001e798  /system/lib64/libc.so (abort+120)

#01 pc 0000000000008348  /system/lib64/liblog.so (__android_log_assert
+296)

#02 pc 000000000001542c  /system/lib64/libutils.so (_ZN7android6Looper
C1Eb+308)

#03 pc 000000000011b5e0  /system/lib64/libandroid_runtime.so (_ZN7andr
oid18NativeMessageQueueC1Ev+160)

#04 pc 000000000011bebc  /system/lib64/libandroid_runtime.so (_ZN7andr
oidL34android_os_MessageQueue_nativeInitEP7_JNIEnvP7_jclass+28)
    #05 pc 000000000065b420  /system/framework/arm64/boot-
framework.oat (android.os.Binder.clearCallingIdentity [DEDUPED]+144)
    #06 pc 0000000000c8731c  /system/framework/arm64/boot-
framework.oat (android.os.HandlerThread.run+332)
```

```
#07 pc 000000000054ad88  /system/lib64/libart.so (art_quick_invoke_stu
b+584)

#08 pc 00000000000dcf74  /system/lib64/libart.so (_ZN3art9ArtMethod6In
vokeEPNS_6ThreadEPjjPNS_6JValueEPKc+200)

#09 pc 000000000046d6a0  /system/lib64/libart.so (_ZN3artL18InvokeWith
ArgArrayERKNS_33ScopedObjectAccessAlreadyRunnableEPNS_9ArtMethodEPNS_8
ArgArrayEPNS_6JValueEPKc+100)

#10 pc 000000000046e8cc  /system/lib64/libart.so (_ZN3art35InvokeVirtu
alOrInterfaceWithJValuesERKNS_33ScopedObjectAccessAlreadyRunnableEP8_j
objectP10_jmethodIDP6jvalue+836)

#11 pc 0000000000496e4c  /system/lib64/libart.so (_ZN3art6Thread14Crea
teCallbackEPv+1120)

#12 pc 0000000000074d74  /system/lib64/libc.so (_ZL15__pthread_startPv
+36)
    #13 pc 000000000001fce4  /system/lib64/libc.so (__start_thread+68)
```

- Native OOM

  Native OOM refers to Out of Memory issues caused by exceeding the native memory address space limit. In Android development, applications may use native code (e.g., C/C++) for high-performance tasks. Memory leaks or excessive native memory allocations in native code can deplete the native memory address space, triggering OOM problems related to native memory allocation. Common manifestations include mmap failures or VRAM exhaustion.

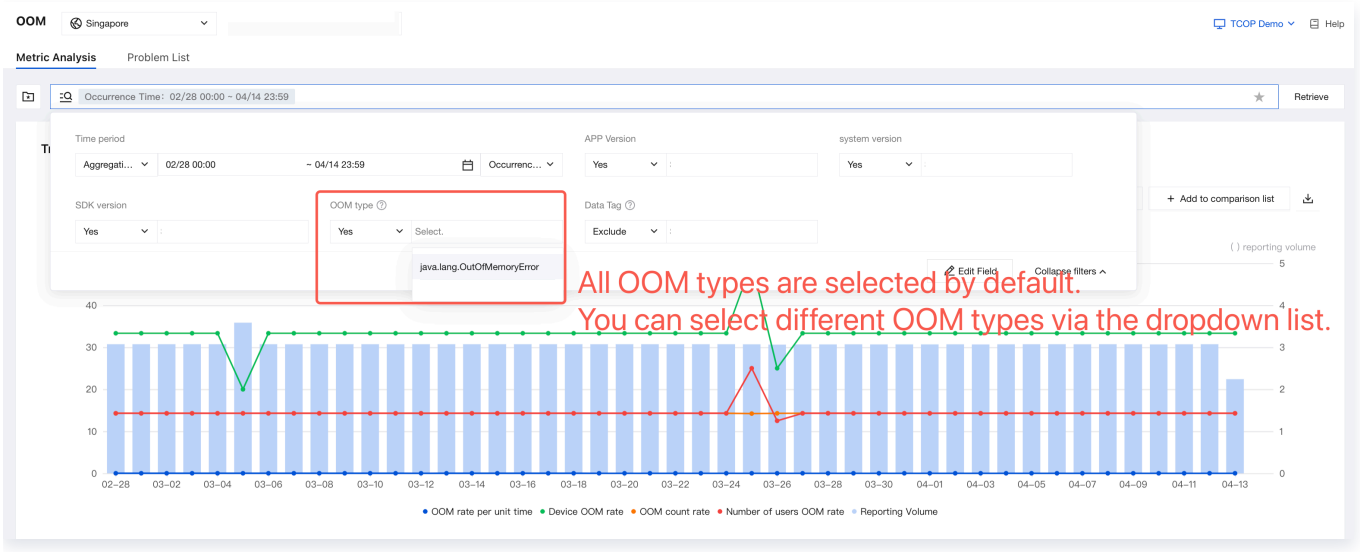
```
03-18 14:39:31.424 9236
30371 W .tencent.xxx: Large object allocation failed: Failed anonymous
 mmap(0x0, 2101248, 0x3, 0x22, -1, 0): Out of memory. See process maps
 in the log.
03-18 14:39:31.437 9236
30371 W .tencent.xxx: Throwing OutOfMemoryError "Failed to allocate a
2097172 byte allocation with 24542160 free bytes and 283MB until OOM,
target footprint 264041288, growth limit 536870912"
(VmSize 4039036 kB)
```

```
03-18 14:39:32.128 9236
4592 D CCodecBufferChannel: [c2.mtk.hevc.decoder#869] DEBUG: elapsed:
mInputMetEos 20, hasPendingOutputsInClient 0, n=1 [in=4 pipeline=0 out
=16]
03-18 14:39:32.492 9236
30526 E CursorWindow: Failed mmap: Out of memory

03-18 15:42:31.466 18578 18864 W Adreno-
GSL: <sharedmem_gpuobj_alloc:2713>: sharedmem_gpumem_alloc: mmap faile
d errno 12 Out of memory
03-18 15:42:32.351 18578 18864 W Adreno-
GSL: <sharedmem_gpuobj_alloc:2713>: sharedmem_gpumem_alloc: mmap faile
d errno 12 Out of memory
03-18 15:42:33.279 18578 18864 W Adreno-
GSL: <sharedmem_gpuobj_alloc:2713>: sharedmem_gpumem_alloc: mmap faile
d errno 12 Out of memory
03-18 15:42:33.321 18578
18864 E OpenGLRenderer: GL error: Out of memory!
```

Metric Analysis

In the OOM rate metric system, java.lang.OutOfMemoryError represents only one category, which we refer to as the Java OOM rate. In addition to the Java OOM rate, we have now introduced the FD OOM rate and Native OOM rate, corresponding to the probability of crash issues caused by FD resources and process Native memory, respectively. The new OOM rate system is accessible in **OOM Types** and requires no modifications on the business side. It can be experienced by upgrading to versions after 4.3.2.



Trend analysis

Trend analysis for OOM is similar to that for FOOM. For details, see FOOM Metric Analysis.

Issue List

The OOM issue list is similar to the crash monitoring module. For details, see Crash Problem List.

> **Note:**
> The benefit of redefining the OOM rate and categorizing OOM issues into distinct types lies in more accurately locating and resolving OOM problems. By monitoring and analyzing different types of OOM rates, we can better understand an application's resource usage across various dimensions, enabling targeted optimization measures.
> - Java OOM rate: Monitoring the Java OOM rate can prompt us to proactively identify memory leaks, optimize garbage collection policies, and adjust heap memory size in order to reduce the occurrence of Java OOM issues.
> - FD OOM rate: Monitoring the FD OOM rate helps identify file descriptor leakage or misuse, enabling timely closure of unused file descriptors to prevent FD OOM issues.
> - Native OOM rate: Monitoring the Native OOM rate helps identify memory leaks or excessive allocation issues in native code, enabling timely release of unused native memory to prevent Native OOM problems.
>
> By categorizing and monitoring different types of OOM rates, we can more precisely locate and resolve OOM issues, enhancing application stability and performance. Additionally, Platform B has provided dedicated monitoring features for each OOM type. We welcome you to use these capabilities.

Issue Details

The OOM issue list is similar to the crash monitoring module. For details, see Crash Issue Details.

Error

Last updated：2026-05-25 18:53:26

Errors refer to exceptions that have been caught by the application, including captured Java exceptions, C# exceptions, JS exceptions, lua exceptions, etc.

Report exceptions

Please see the Integration Guide to call the SDK's API and report errors.

custom exception API

Android

If you need to report errors other than Java Crash, Native Crash, and ANR, such as C# exceptions, you can use the custom exception API to report them to the platform. The reported data is displayed on the **Error** page. For detailed access steps, see the Android SDK Access Guide.

```
/**
 * Report custom exception
 * @param thread The thread where the error occurred; the default is the current thread
 * @param category Exception type (u3d c#: 4 | js: 8 | cocos2d Lua: 6)
 * @param errorType Error type
 * @param errorMsg Error message
 * @param stack Error stack trace
 * @param extraInfo Extra information
 */
public static void postException(Thread thread, int category, String errorType, String errorMsg,
                                 String stack, Map<String, String> extraInfo);

public static void postException(int category, String errorType, String errorMsg,
                                 String stack, Map<String, String> extraInfo);
```

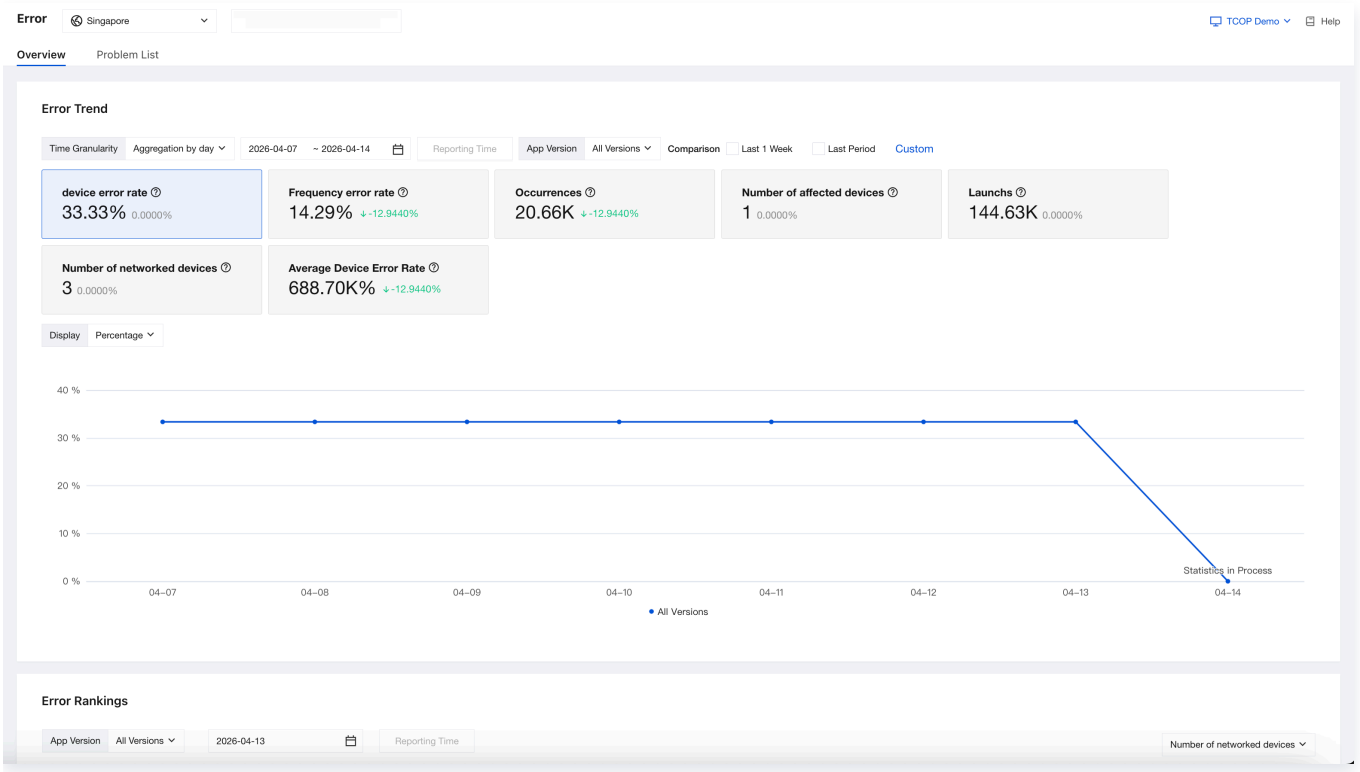
iOS

If you need to report errors other than OC/C++, such as C#/js exceptions, you can use the custom exception API to report them to the platform. The reported data is displayed on the **Error** page. For detailed access steps, see the iOS SDK Access Guide.

```
/**
 *  @brief Report custom error
 *
 *  @param category      Type (Cocoa=3, CSharp=4, JS=5, Lua=6)
 *  @param aName         Name
 *  @param aReason       Error reason
 *  @param aStackArray   Call stack
 *  @param info          Additional data
 *  @param terminate     Whether to terminate the app process after reporting
 */
+ (void)reportExceptionWithCategory:(NSUInteger)category
                               name:(NSString *)aName
                             reason:(NSString *)aReason
                          callStack:(NSArray *)aStackArray
                          extraInfo:(NSDictionary *)info
                       terminateApp:(BOOL)terminate;
```

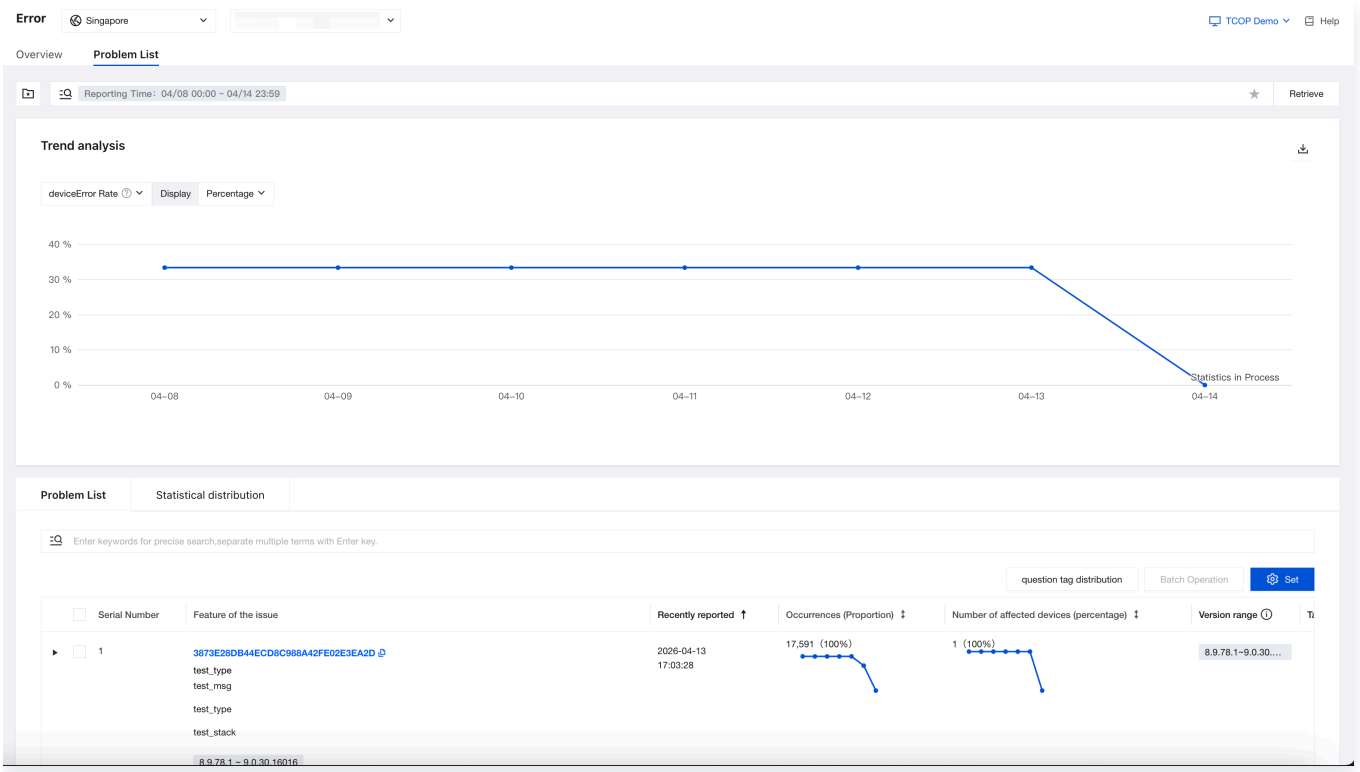
Exception Overview

The exception overview for Android and iOS is similar to the crash monitoring module, both containing an overview page. The overview includes today's errors, error trends, and the error leaderboard. For details, see Crash Overview.



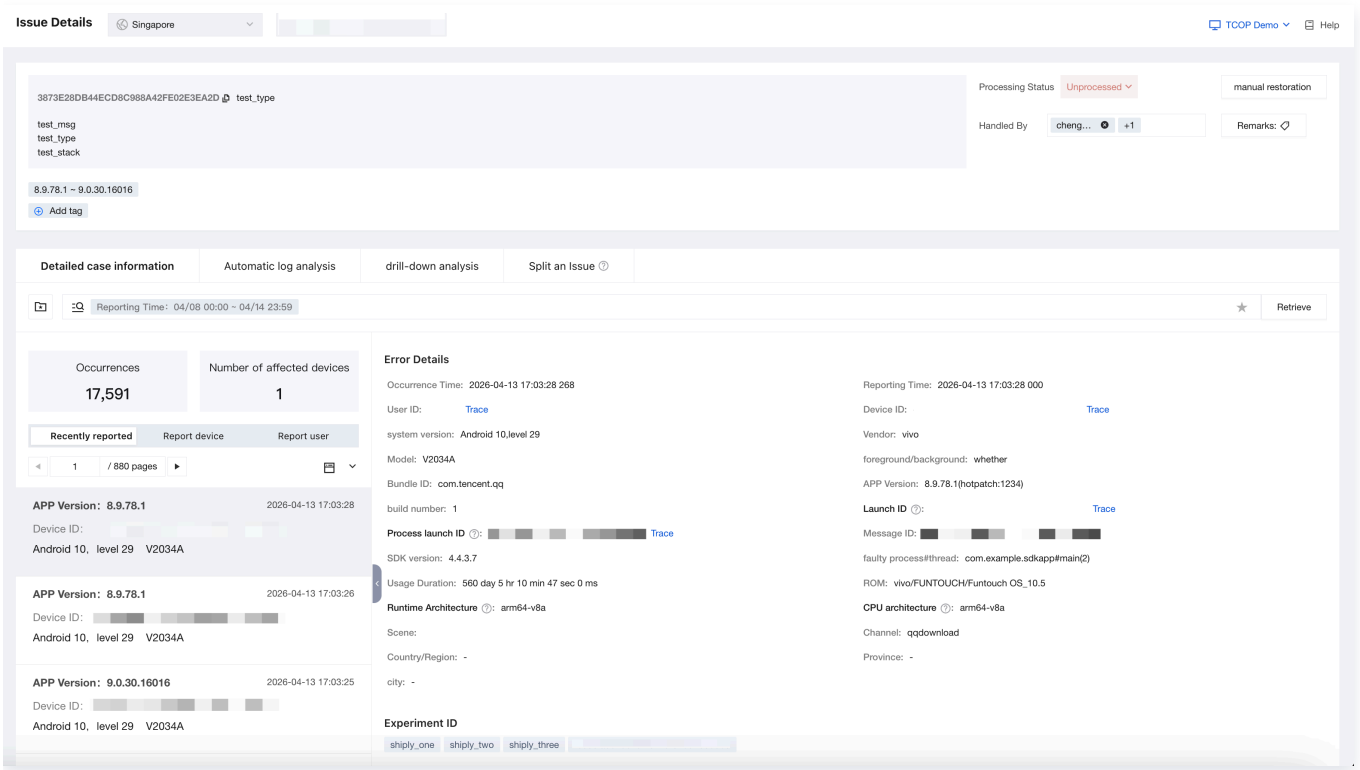
Problem List

The issue list for Android and iOS errors supports rich search criteria, provides statistical analysis capabilities, and displays Top Issues. For details, see Data Analysis – Issue List.



Issue Details

The issue details for Android and iOS errors include three key modules: issue summary, case details, and drill-down analysis. For details, see Data Analysis – Issue Details.



Lag

Last updated：2026-05-25 18:53:26

Overview

Application lag is one of the key reasons leading to user churn. The **Lag Monitoring** module is dedicated to assisting users in managing lag issues. An important approach to lag management is to measure the current state of the application by extracting effective metrics, capturing sufficient information to provide optimization directions. The lag monitoring module measures the application's smoothness using two metrics: FPS and suspension rate. Through lag issue monitoring, it directly captures lag stacks, helping users identify the causes of lag and providing optimization directions.

- Lag metric uses stable and lightweight collection techniques to collect smoothness data throughout the entire application runtime.
- Lag issue monitoring utilizes high-frequency stack trace capturing to provide rich contextual information for accurate cause analysis.
- The Android platform's self-developed rapid stack trace capturing technology effectively improves performance compared to traditional stack trace capturing implementations.
- Both lag metrics and lag issues support custom sampling rates and provide robust data analysis capabilities.

Metric

The module measures application smoothness using two metrics: **FPS** and **hanging rate**.

- FPS data is aggregated by scenario, directly counting user-reported raw records, and calculates average and percentile values based on these raw records.
- Suspension rate is calculated based on user-reported raw records, aggregated daily by device ID. Multiple records generated by a single device in a day are consolidated into a single record. The suspension rate is then computed by averaging and calculating percentile values from these aggregated data sets.

FPS

FrameRate: Frame rate is the number of images that the GPU and CPU can generate during application runtime, measured in frames/second (FramesPerSecond, FPS). It is typically used as a metric to evaluate hardware performance and application smoothness. Key points include:

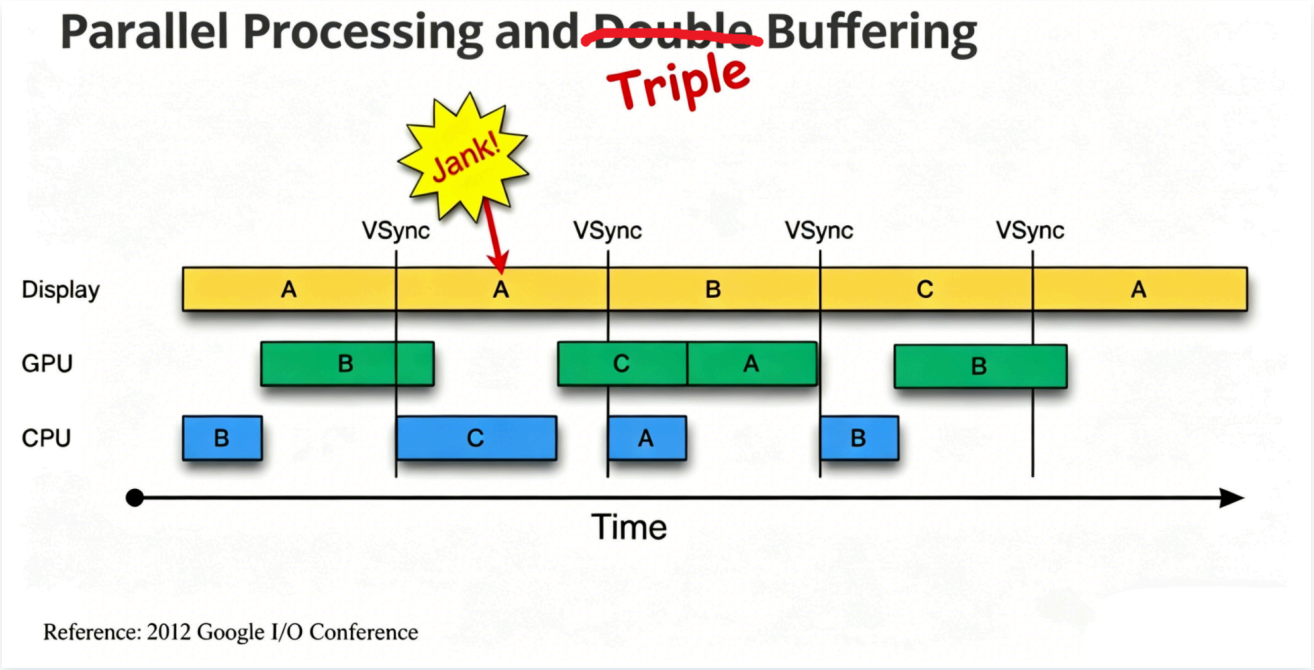
- **Lag Monitoring statistics include FPS during UI refresh, capturing the real user experience of the application.**

  RUM Pro's lag monitoring measures FPS that includes real UI refresh, excluding data from page idle periods. For example, when viewing a photo album page, if the application does not handle asynchronous loading properly, it may experience lag during user scrolling. When the user is quietly viewing photos,

since no time-consuming operations are triggered, it may appear normal. If FPS is calculated based on the total time the user stays on the page, the period without refresh may average out the lag during scrolling. With RUM Pro's lag metric monitoring, this issue is avoided; we only measure FPS that includes real page refresh, so FPS remains unaffected by variations in page idleness duration.

- **The platform normalizes FPS data to ensure compatibility across various refresh rates.**

  Currently, smartphones on the market feature various screen refresh rates, commonly 60Hz, 90Hz, and 120Hz. Simply put, if an application performs exceptionally well with minimal UI update overhead, its frame rate is limited by the screen refresh rate and cannot exceed it. This means that even a high-performance application will exhibit different frame rates across devices with varying screen refresh rates. The impact of different refresh rates on VSYNC signal generation frequency is illustrated in the following figure.



  RUM Pro adopts a normalization approach to unify different screen refresh rates to 60Hz. This means that even across devices with varying refresh rates, the platform reports nearly identical FPS values for the same application.

- **The platform aggregates FPS by scenario, and data from the same scenario during a single run is reported in a consolidated manner.**

  RUM Pro aggregates FPS by scenario during application runtime. The lag metrics module collects data for each scenario. When the user switches scenarios, the module retains data from the previous scenario. The next time the application is launched, the reporting module retrieves data collected during the last run, consolidates identical scenarios, and ultimately generates distinct data records for each scenario.

- **Supports multiple percentile values: P50, P90, P99, helping users better understand actual user experience.**

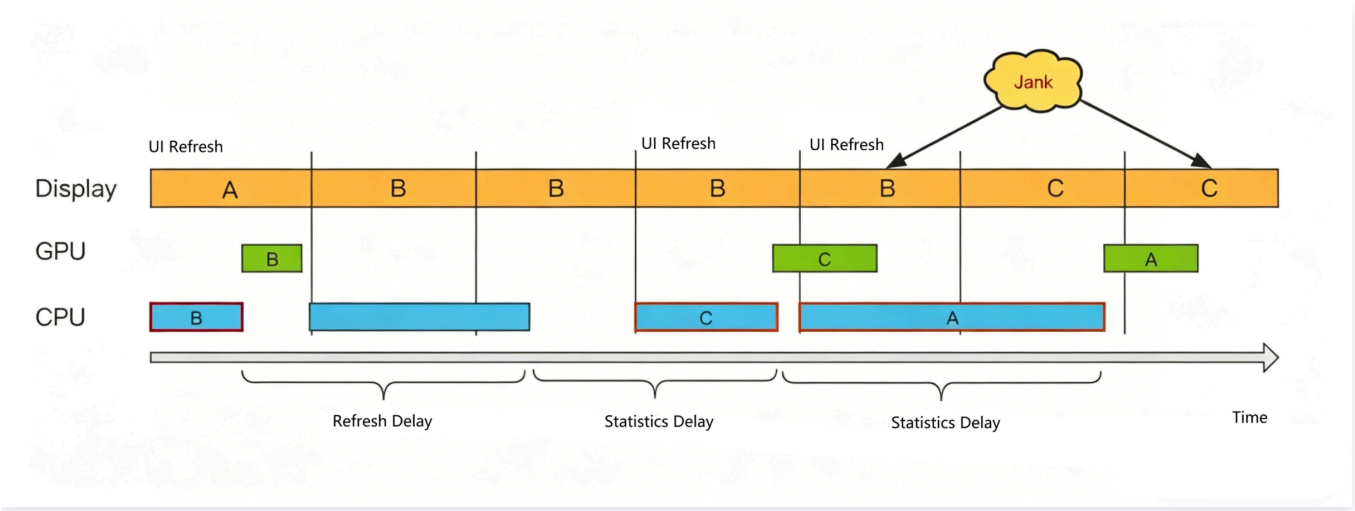
  Scenario data records are reported to the server, which stores these raw records while calculating average and percentile values (P50, P90, P99).

Percentile values (P50, P90, P99):
For example, during a single run of Application A, a user experiences 10 scenarios, generating 10 records reported to the server. With 10 users reporting, a total of 100 records are generated. Each record contains FPS data. The server sorts these 100 records by FPS value in descending order. The 50th data point in this sorted dataset represents the P50 value; the 90th data point represents P90; and the 99th data point represents P99.
In FPS, percentile values are sorted in descending order, thus P50 >= P90 >= P99.

- The SDK's FPS statistics collection method has minimal performance impact on applications.

Hanging rate

For lag monitoring, the suspension rate is calculated as follows: if the refresh delay between two application frames exceeds 200 ms, it is considered that the application fails to respond adequately to user interactions, and this delay is accumulated into the suspension time. The suspension rate of a device is defined as its total suspension time divided by the device's total foreground duration within a day. In other words, the suspension rate in lag monitoring is aggregated daily per device, calculated as: Device Suspension Rate = Total Daily Suspension Time (in seconds) / Total Daily Foreground Duration (in hours).



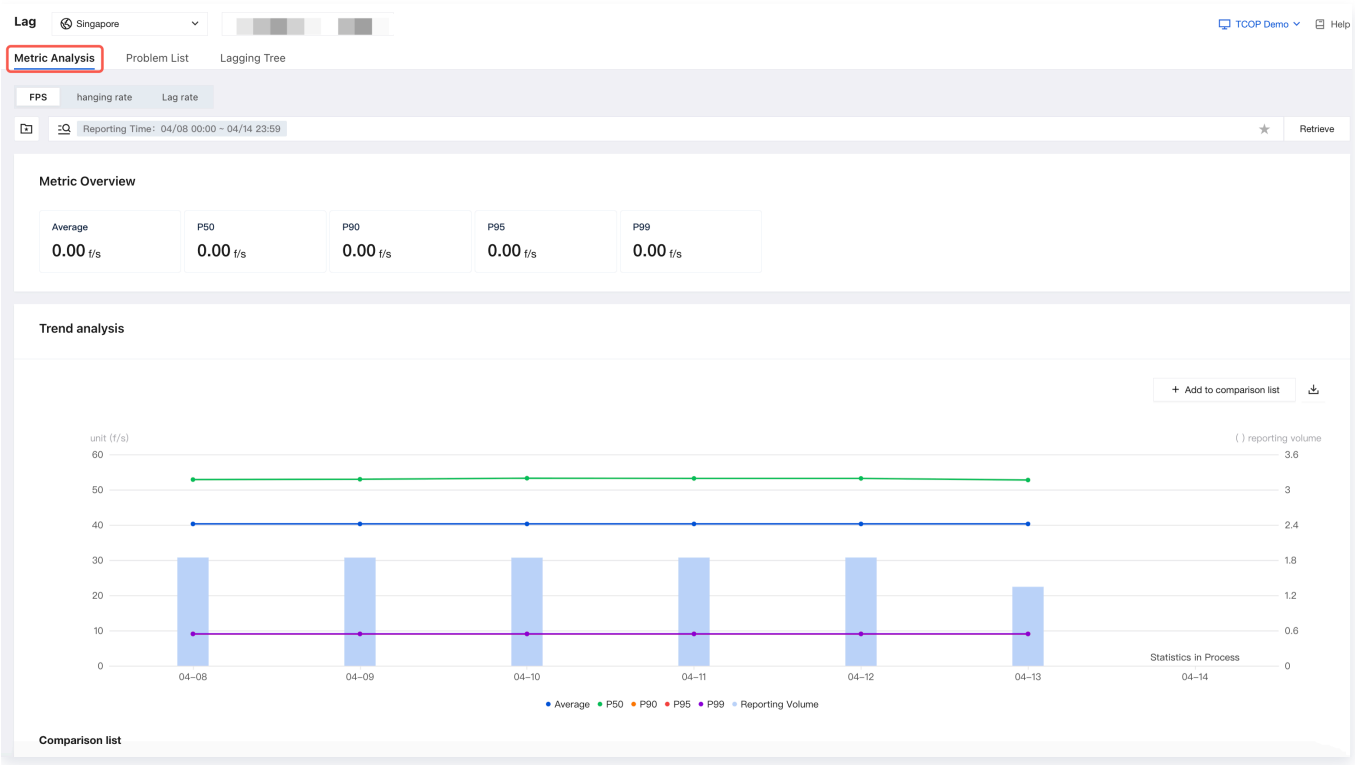
For an application, we focus on percentile values of device suspension rates, such as P50, P90, and P99. For example, for Application A on a given day, 100 devices reported suspension rate data. The backend aggregates by device ID, resulting in 100 records. Each record contains a suspension rate value. These 100 records are sorted in ascending order. In this sorted dataset, the 50th record shows 45.23s/h, thus P50 = 45.23s/h. Similarly, the 90th record shows 134.23s/h, so P90 = 134.23s/h.

Metric Analysis

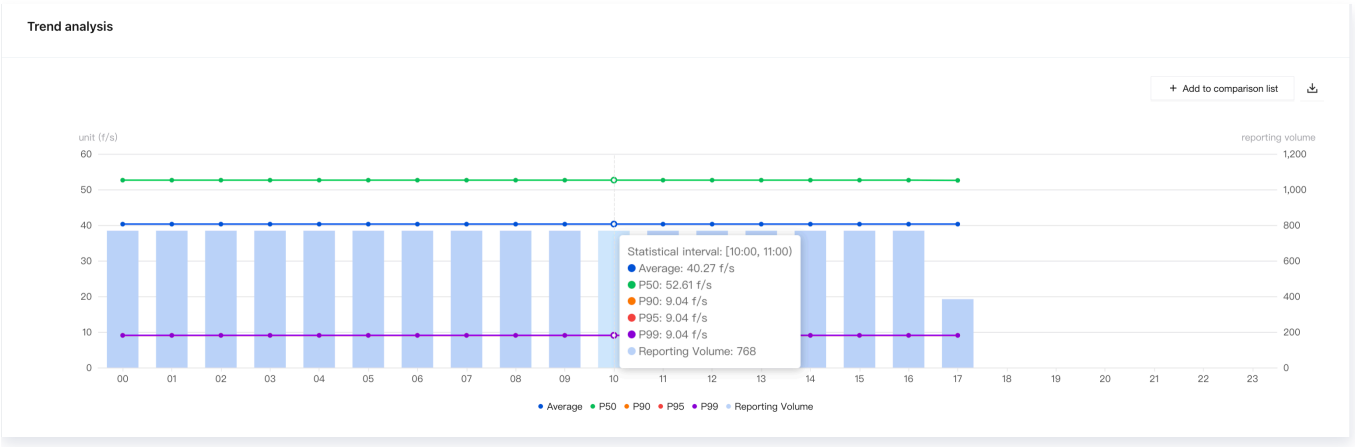
The following content uses FPS metrics as an example for analysis. Suspension rate can be referenced similarly.

Trend analysis

Users can view FPS trend analysis for a day or a specific period. The platform provides multiple statistical methods, including average and percentile values (P50, P90, P99), helping users better understand the overall FPS performance of applications.



Users can query FPS data under specified conditions. As shown in the following figure, the FPS for a specified scenario:

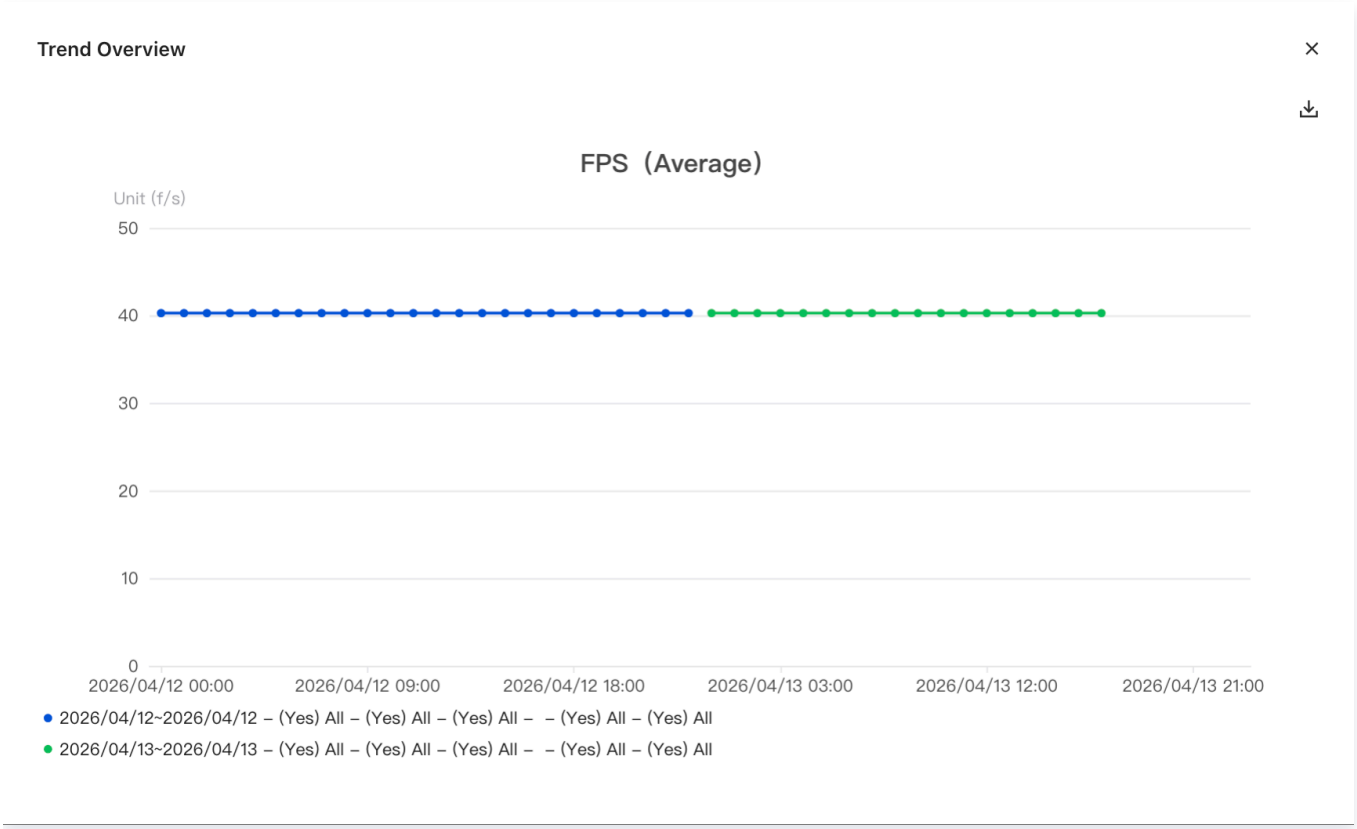


Compare and Analyze

In trend analysis, you can add a set of query results to the comparison list. After adding multiple sets of query results, you can subsequently compare and analyze these datasets within the comparison list.
As shown in the figure below, it is desired to compare and analyze data from two different scenarios. Query the data for these two scenarios separately, add them to the comparison list, and click **Trend Preview** to

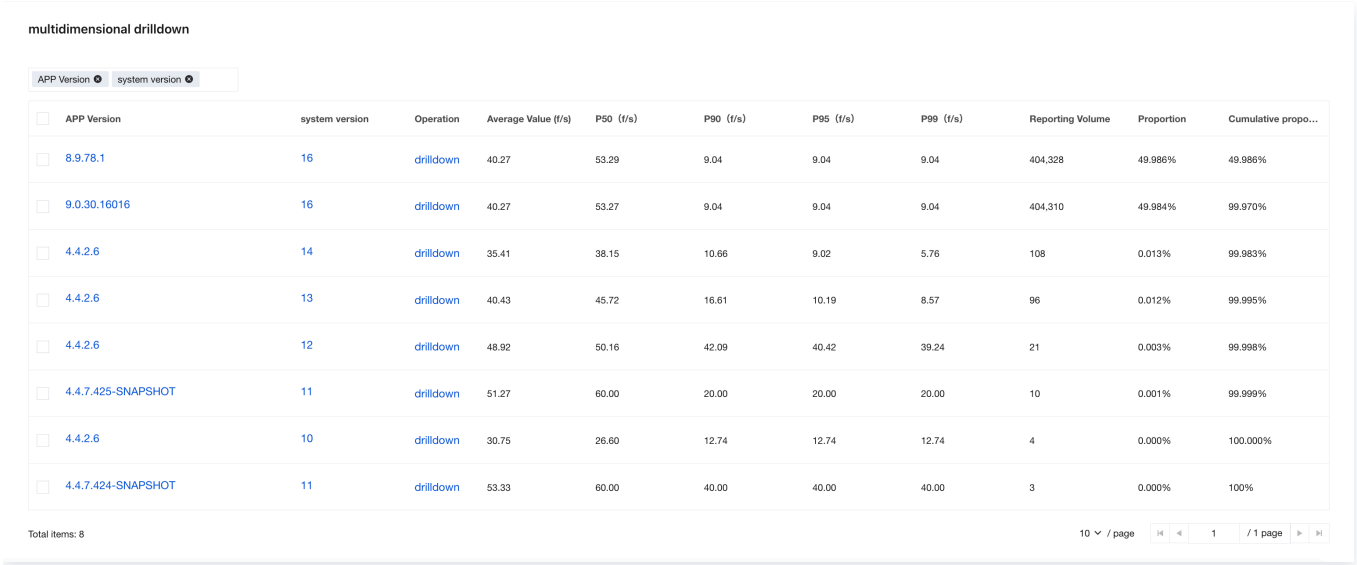
view comparison results across different statistical dimensions.

Trend Preview Sample Chart:



Multidimensional drilldown

In some cases, if you want to view data across all App Versions or all scenarios over a period of time, this can be addressed through multidimensional analysis.



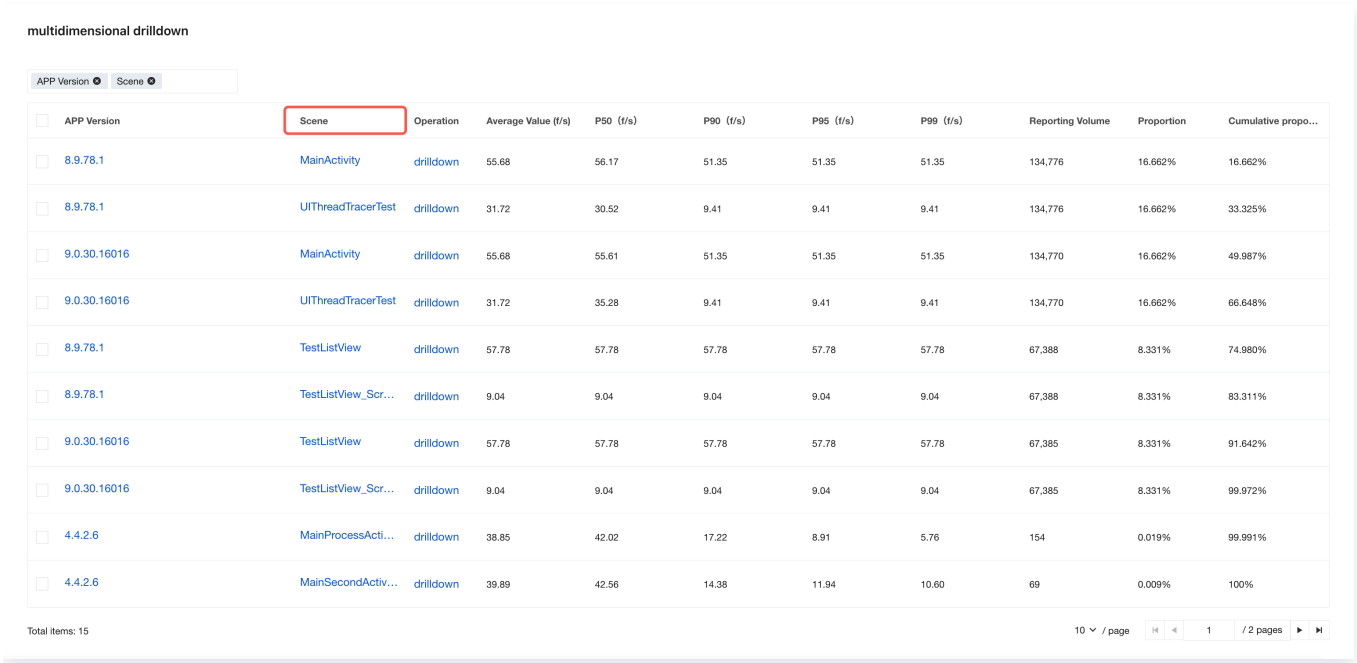
multidimensional drilldown

APP Version | system version

APP Version	system version	Operation	Average Value (f/s)	P50 (f/s)	P90 (f/s)	P95 (f/s)	P99 (f/s)	Reporting Volume	Proportion	Cumulative propo...
8.9.78.1	16	drilldown	40.27	53.29	9.04	9.04	9.04	404,328	49.986%	49.986%
9.0.30.16016	16	drilldown	40.27	53.27	9.04	9.04	9.04	404,310	49.984%	99.970%
4.4.2.6	14	drilldown	35.41	38.15	10.66	9.02	5.76	108	0.013%	99.983%
4.4.2.6	13	drilldown	40.43	45.72	16.61	10.19	8.57	96	0.012%	99.995%
4.4.2.6	12	drilldown	48.92	50.16	42.09	40.42	39.24	21	0.003%	99.998%
4.4.7.425-SNAPSHOT	11	drilldown	51.27	60.00	20.00	20.00	20.00	10	0.001%	99.999%
4.4.2.6	10	drilldown	30.75	26.60	12.74	12.74	12.74	4	0.000%	100.000%
4.4.7.424-SNAPSHOT	11	drilldown	53.33	60.00	40.00	40.00	40.00	3	0.000%	100%

Total items: 8 — 10 / page — 1 / 1 page

It also supports analyzing data for specified versions under specific scenarios.



multidimensional drilldown

APP Version | Scene

APP Version	Scene	Operation	Average Value (f/s)	P50 (f/s)	P90 (f/s)	P95 (f/s)	P99 (f/s)	Reporting Volume	Proportion	Cumulative propo...
8.9.78.1	MainActivity	drilldown	55.68	56.17	51.35	51.35	51.35	134,776	16.662%	16.662%
8.9.78.1	UIThreadTracerTest	drilldown	31.72	30.52	9.41	9.41	9.41	134,776	16.662%	33.325%
9.0.30.16016	MainActivity	drilldown	55.68	55.61	51.35	51.35	51.35	134,770	16.662%	49.987%
9.0.30.16016	UIThreadTracerTest	drilldown	31.72	35.28	9.41	9.41	9.41	134,770	16.662%	66.648%
8.9.78.1	TestListView	drilldown	57.78	57.78	57.78	57.78	57.78	67,388	8.331%	74.980%
8.9.78.1	TestListView_Scr...	drilldown	9.04	9.04	9.04	9.04	9.04	67,388	8.331%	83.311%
9.0.30.16016	TestListView	drilldown	57.78	57.78	57.78	57.78	57.78	67,385	8.331%	91.642%
9.0.30.16016	TestListView_Scr...	drilldown	9.04	9.04	9.04	9.04	9.04	67,385	8.331%	99.972%
4.4.2.6	MainProcessActi...	drilldown	38.85	42.02	17.22	8.91	5.76	154	0.019%	99.991%
4.4.2.6	MainSecondActiv...	drilldown	39.89	42.56	14.38	11.94	10.60	69	0.009%	100%

Total items: 15 — 10 / page — 1 / 2 pages

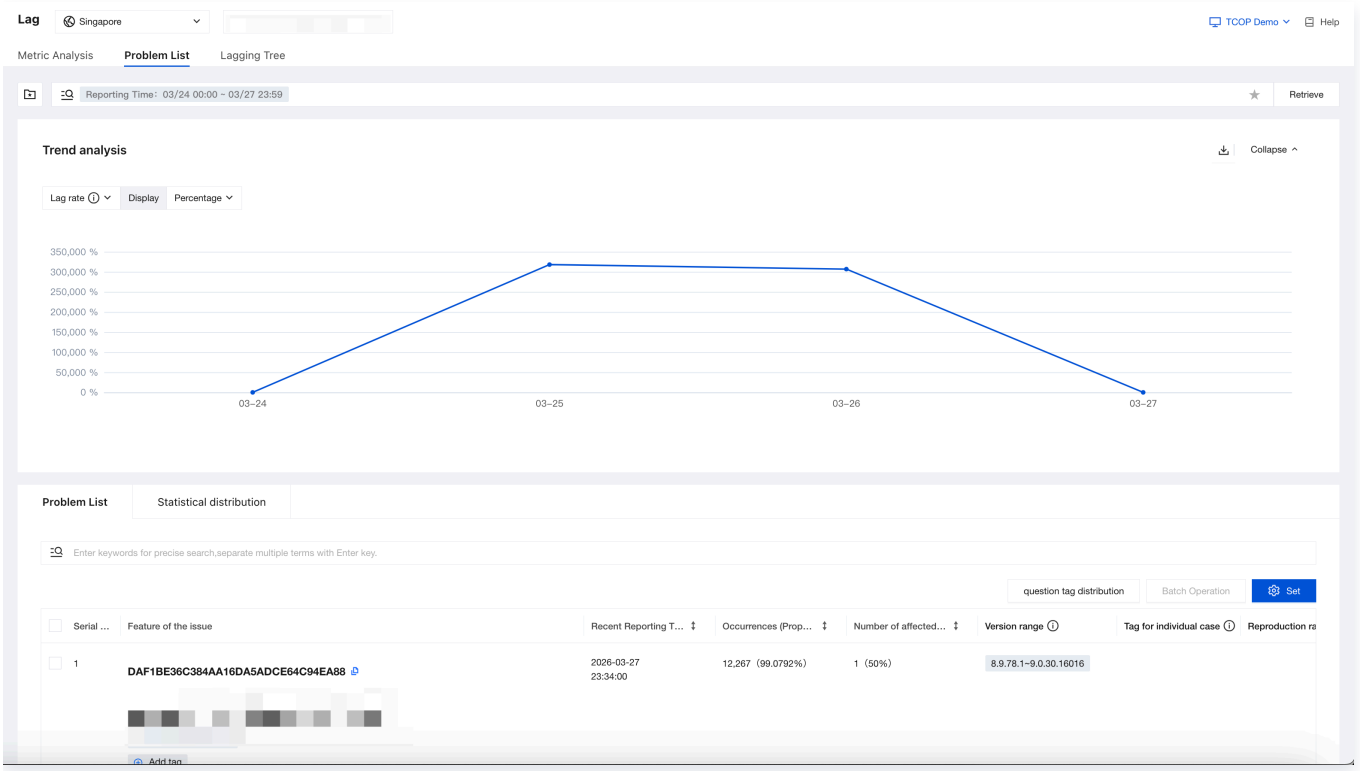
problem monitoring

The core focus of lag Issue monitoring is to help users identify the causes of application lag. By monitoring the processing time of UI thread messages and implementing a high-frequency, continuous stack capturing policy, instances exceeding the threshold are reported to the server. Based on the lag stack tree, the server extracts key time-consuming characteristics, aggregates individual cases into Issues (problems), and generates a list of lag issues.

Problem List

On the **Lag** > **Problem List** page, you can view data monitored for lag issues, with support for filtering and analyzing the data.

- The lag issue list supports rich search criteria. Users can freely configure the desired search fields to display by clicking **Edit Field**. They can also adjust the display of the search criteria area by toggling **Display filter** and **Collapse filters**. The webpage will retain the user's selection;
- After selecting the search criteria, click **Retrieve** to submit the query. Once results are retrieved, the page will automatically refresh.
- The header section of the lag issue list displays summary information of query results, allowing users to see the number of issues meeting the criteria and the proportion of Top N issues.
- Query results are sorted by the reported time of Issues by default. Each Issue includes characteristics, last reported time, average lag duration and its percentile values, average key stack time consumption and its percentile values, maximum leaf node time consumption percentile values, etc. Based on actual needs, users can freely select fields to display via **Set**.
- Clicking on the issue ID allows users to go to the issue details and view the reporting specifics of an Issue.
- When the list displays a large number of statistical fields, the result list becomes too wide to display completely. Users can swipe left or right to view all parts.
- Users can also export the content of this page directly as data in table format.

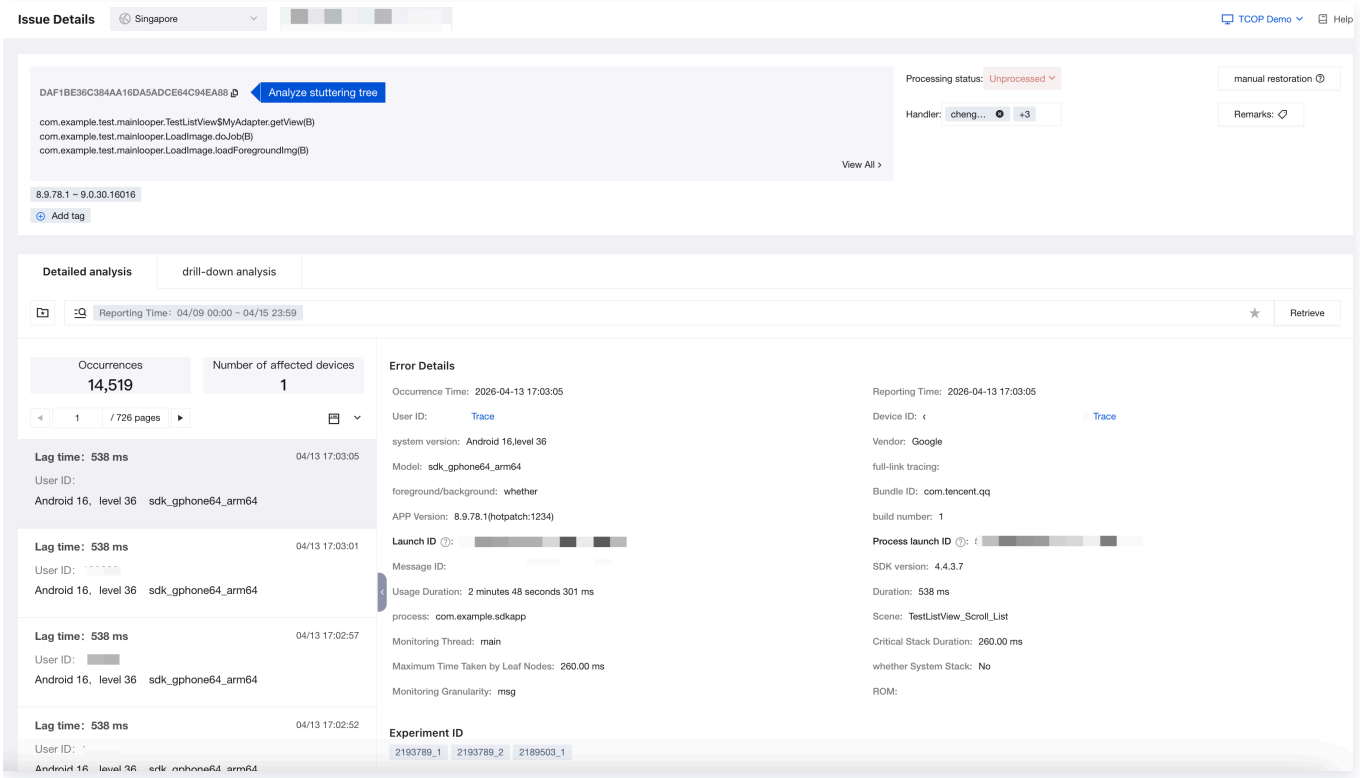


Issue Details

The issue details are divided into three parts: the header section, detailed analysis, and drill-down analysis.

- The header section: includes descriptions at the Issue level, such as the version tag of the Issue, custom tags assigned by users, processing status, assignee, Issue ID, characteristics, etc.

- Case Analysis: focuses on analyzing laglag cases. Users can query cases meeting specific criteria using search conditions.
- Drill-down analysis: focuses on analyzing the entire Issue, including trend analysis, statistical distribution, error stack analysis, etc.

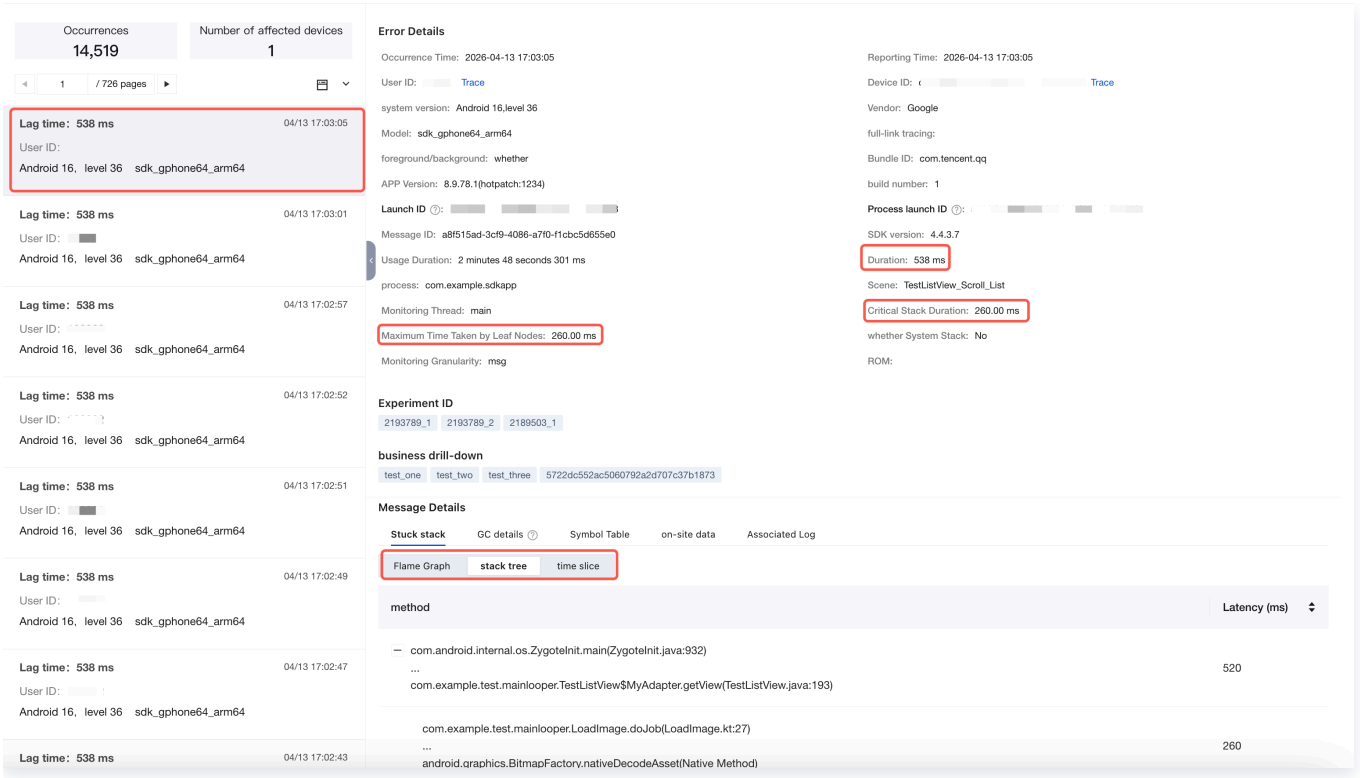


Case Analysis

When viewing a lag case, you can focus on the following information:

1. Prioritize viewing the total lag time consumption, key stack time consumption, and maximum leaf node time consumption. This information helps quickly determine the type of lag.
2. Then, proceed to analyze the stack details of the lag. The current stack snapshot of the lag is presented in three different ways: time slices, stack tree, and flame graph.
   - Typically, start by analyzing the flame graph, as it visually presents an overview of the lag scenario.
   - Then proceed to analyze stack details using time slices or flame graphs.
   - The time consumption in flame graphs and stack trees is estimated through stack sampling.
   - The lag time consumption is accurate, indicating the execution time of a message on the UI thread.

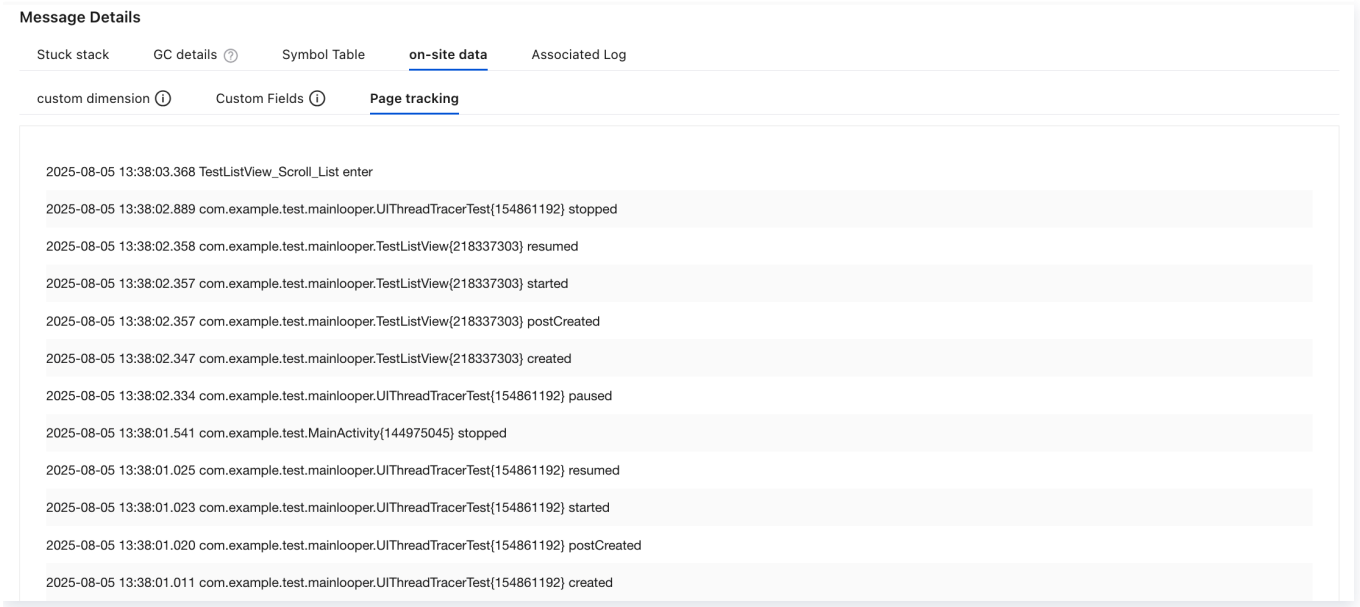
> **Lag Time Consumption and Stack Time Consumption:**
> The lag time consumption indicates the accurate execution duration of a message on the UI thread. The stack time consumption is estimated based on stack sampling frequency and intervals. The current lag monitoring system detects application lag by tracking the execution time of UI thread messages.



Taking the following figure as an example, this stack was captured in 6 consecutive frames. With the current stack sampling interval of 52ms, the estimated time consumption for the stack is 52 x 6 = 312ms.

> **Note:**
> **Stack Sampling Interval**: Android and iOS devices may have different stack sampling intervals. You can derive the interval for this specific case by combining the number of stack samples shown in time slices with the calculated stack sampling time consumption.

Additionally, users can further analyze the lag scenario by combining operation logs and on-site data.



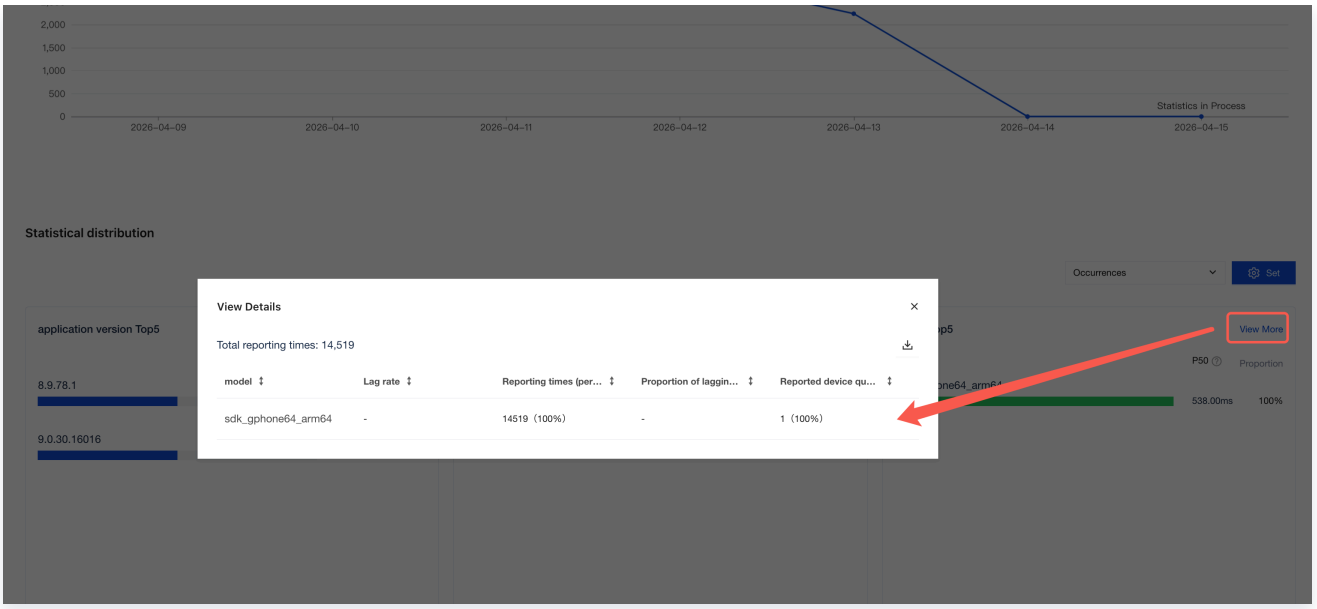
Drill-down analysis

Users can perform drill-down analysis to view the reporting trends of issues and their distribution across various fields.

- Drill-down analysis allows users to set search criteria and view statistical results under specified conditions.
- The upward trend supports viewing data across different statistical dimensions, including occurrence count, number of affected users (deduplicated by user ID), and number of affected devices (deduplicated by device ID). It supports data viewing at various aggregation granularities, with a minimum granularity of minute-level.

- Statistical distribution refers to viewing the distribution status across designated fields under specified search conditions.

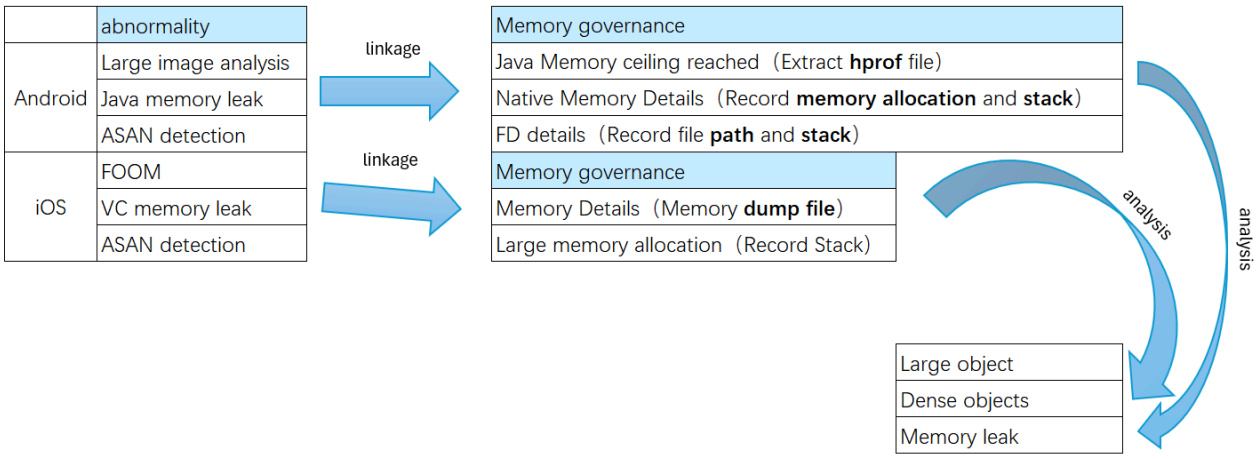
> **Note:**
> **Valid Reports for Statistical Distribution** refer to the reporting status where this field contains valid values. In some scenarios, certain fields may have unavailable or invalid values. Therefore, even with identical search conditions, the number of valid reports may vary across different fields.


2,000
1,500
1,000
500
0
2026-04-09
2026-04-10
2026-04-11
2026-04-12
2026-04-13
2026-04-14
2026-04-15
Statistics in Process
Statistical distribution
Occurrences
Set
application version Top5
View More
8.9.78.1
9.0.30.16016
P50
Proportion
538.00ms
100%
View Details
Total reporting times: 14,519
model
Lag rate
Reporting times (per...
Proportion of laggin...
Reported device qu...
sdk_gphone64_arm64
-
14519 (100%)
-
1 (100%)

Memory

Last updated：2026-05-25 18:53:27

This article introduces the types of metrics supported by the platform memory monitoring module, how to enable monitoring configurations, and the analysis of various indicators.
The following figure illustrates the correlation and analysis logic in memory anomaly and memory governance for the Android and iOS platforms.



Android

Overview

In software development activities, memory management is a particularly complex task, and most tricky bugs are fundamentally related to memory. Memory issues can be categorized into the following three types: memory leaks, memory abuse, and memory access exceptions.

- **Memory Leak**: When memory that should be released is not freed, it causes a memory leak issue. After a leak occurs, the process memory continues to grow, eventually leading to problems such as OOM.
- **Memory Abuse**: Using mechanisms like caching to enhance program performance is a common optimization technique. However, its effectiveness generally depends on the management of cached objects. Slight abnormalities may cache many unused objects, which not only fails to optimize performance but also prevents other modules from allocating memory, thereby causing memory-related performance issues such as GC stuttering.
- **Memory Access Exception**: Memory access exception refers to common issues such as segmentation faults and memory out-of-bounds access. These types of problems are generally difficult to analyze. For instance, in memory out-of-bounds access, there might be no immediate exception at the time of access; it simply corrupts the value at certain addresses. Later, when other modules use this memory, it

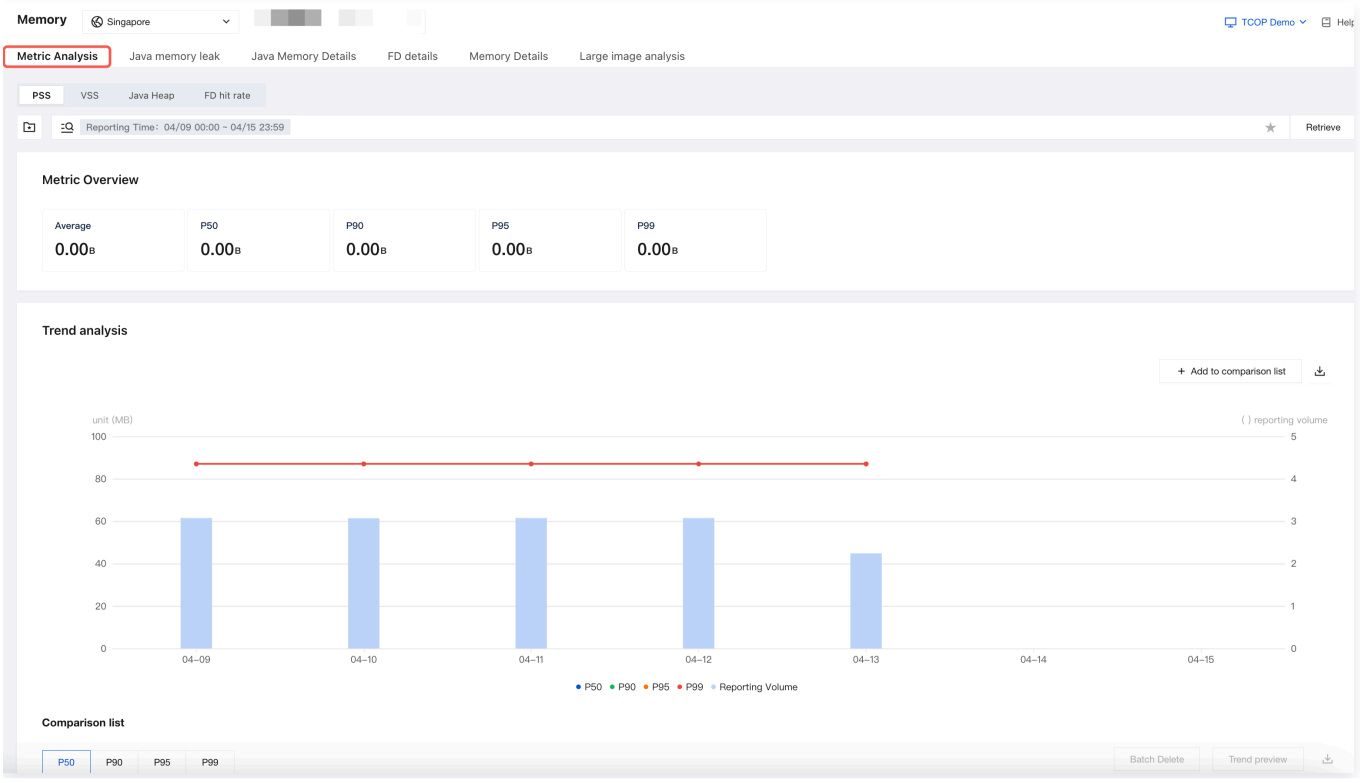
may trigger issues like segmentation faults. However, since it's unknown who corrupted the memory, even knowing the stack trace of the segmentation fault is usually of no help.

Memory issues can severely impact fundamental user experiences, such as crashes, black screens, and stuttering, making monitoring for such problems crucial. Due to frequent usage scenarios and memory abuse, monitoring and identifying memory issues pose significant challenges. To address this, the platform's memory monitoring is primarily divided into **abnormal monitoring** and **memory governance**.

- **Exception Monitoring**: For abnormal situations, the platform provides precise log information, allowing users to directly submit tickets for follow-up, such as Activity leaks, memory out-of-bounds access, large bitmaps, etc.
- **Memory Governance**: Provides capabilities focused on memory governance, such as Java memory details, FD details, and Native memory details. Examples include leakage chains of Activities or stack traces for memory out-of-bounds access. Capabilities like Java memory details and FD details capture log information and report to the backend when resources such as Java memory or FD resources are nearly exhausted. This allows analysis of suspected issues, such as reference chains of oversized Java objects or stack traces indicating excessive FD resource consumption. Based on this information, users can directly identify or further analyze the root causes of problems.

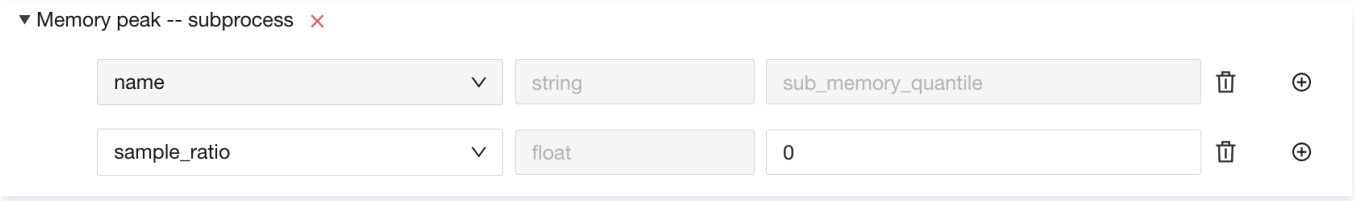
Metric Analysis

Memory metrics are statistical data for overall users, allowing observation of the application's overall memory usage. Additionally, by comparing metrics horizontally or vertically, users can detect whether memory indicators show deterioration. Currently, Android memory monitoring metrics primarily include two data points: **memory peak and FD peak touch rate**. These can be viewed on the Metric Analysis page under Memory on the left side of the interface.

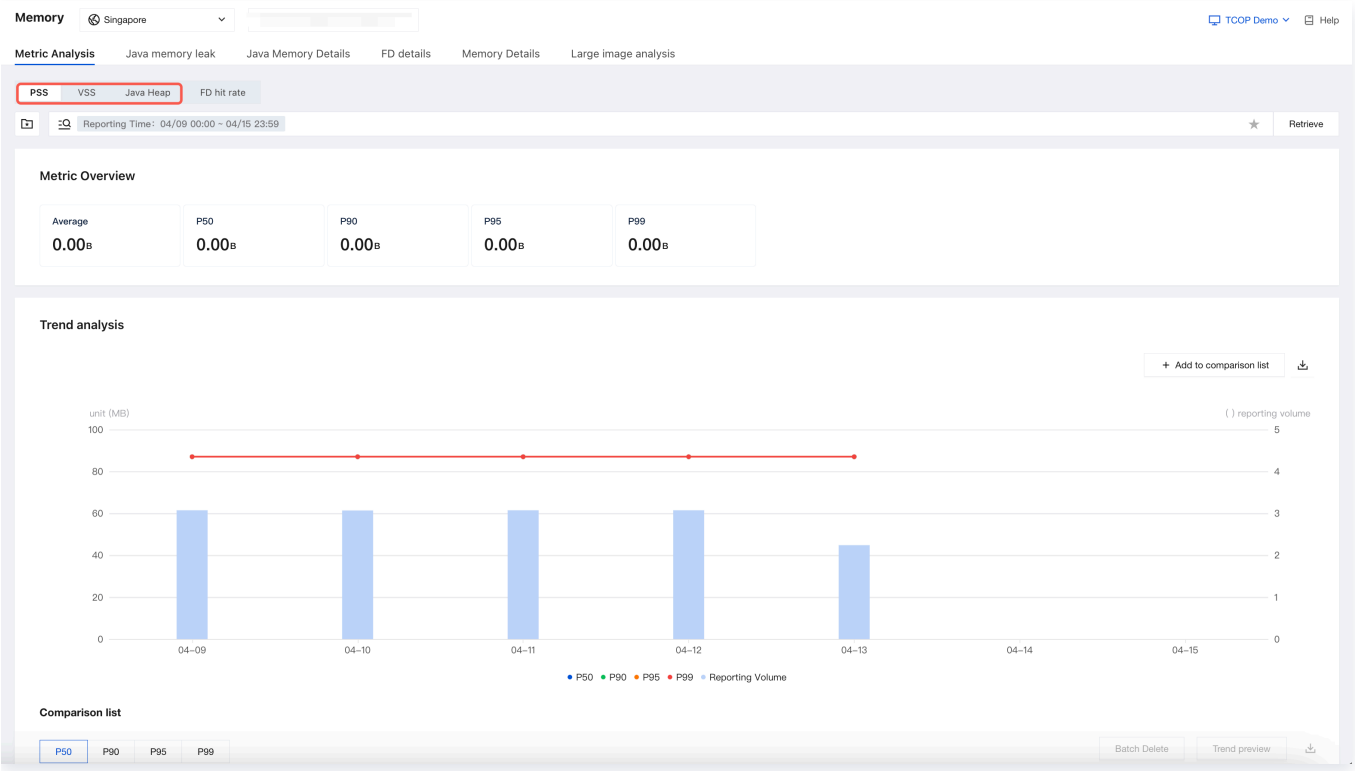


Memory Peak

Memory peak refers to the maximum memory usage during a process's lifecycle, categorized into three types: **PSS**, **VSS**, and **JavaHeap (totalMemory – freeMemory)**. Available memory on any mobile device is limited. The more memory a process consumes, the higher its risk of being terminated by the system in the background, and the more performance issues caused by GC may arise. The memory peak metric can, to some extent, gauge the severity of these issues. By default, the platform collects memory peak data for the main process. To obtain memory peak data for child processes, enable it via the following configuration on the Setting > SDK configuration Edit Configuration page:



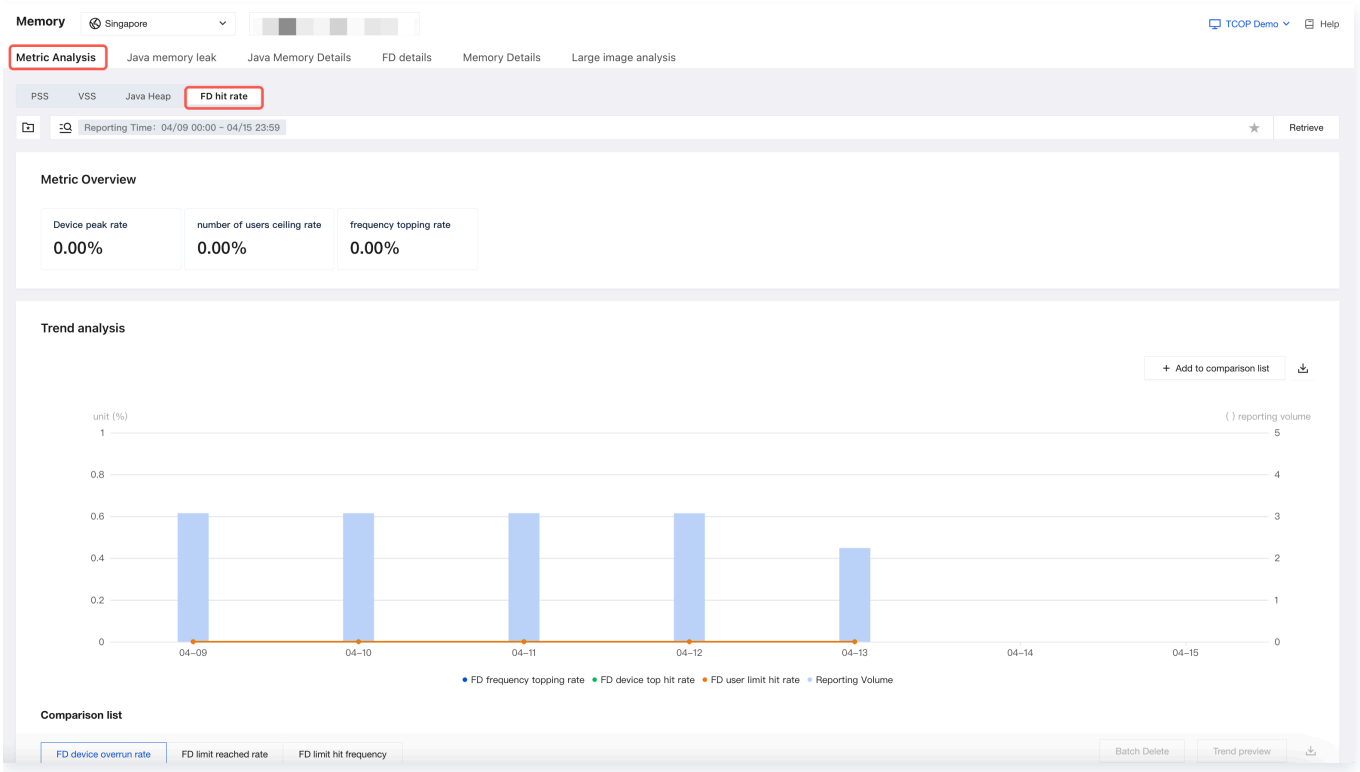
Go to the memory indicator analysis page to view memory peaks, including three metrics: **PSS**, **VSS**, and **Java Heap**, with support for multiple drill-down methods.



FD hit rate

Like memory, FD is also a finite resource. Once a process exceeds the maximum FD resources, it can no longer obtain additional FDs, leading to issues such as crashes and black screens. Particularly in some lower Android versions, the available FD resources for a process are limited to 1024. For large-scale applications, 1024 FD resources are often insufficient to meet demands. Therefore, it is necessary to monitor FD resource usage in our processes. The FD peak touch rate measures the probability of the number of FDs reaching a

predetermined threshold after enabling the FD details feature. This metric is reported only when the FD details monitoring feature is enabled.



Java memory leak

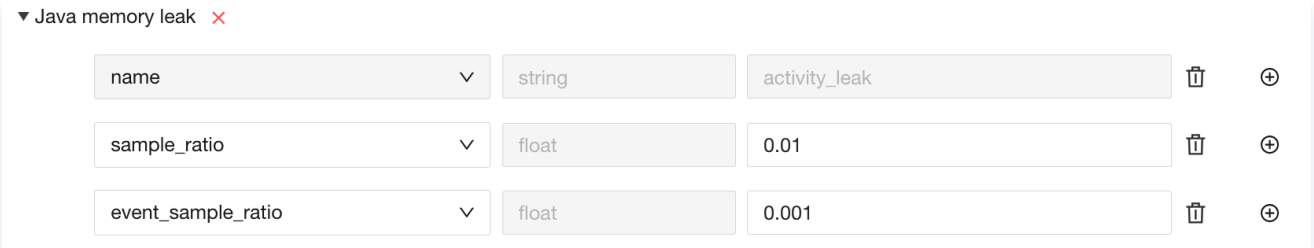
Enabling Method

- The client adds the following code during initialization.

```
RumProBuilder builder = new RumProBuilder(appID, appKey);
build.addMonitor(RumProMonitorName.MEMORY_JAVA_LEAK);
```

- On the Setting > SDK configuration Edit Configuration page, configure according to the following configuration items.

  In the backend, configure the sampling rate for the Java memory leak monitoring feature. Users can adjust the user sampling rate and event sampling rate via the **Java memory leak** configuration.
  - **sample_ratio**: The user sampling rate for the Java memory leak detection feature. 0 disables the feature for all users, 1 enables the feature for all users.
  - **event_sample_ratio**: The event sampling rate for the Java memory leak detection feature. 0 indicates no memory leak events are reported, 1 indicates all memory leak events are reported.



API Description

After enabling the Java memory leak monitoring feature, the platform automatically monitors two types of objects: Activity and Fragment.

- For Activity: By registering an ActivityLifecycleCallback, Activity objects that have executed the onDestroy callback are collected.
- For Fragment: By registering FragmentManager.FragmentLifecycleCallbacks, Fragment objects that have executed the onDestroy callback are collected.

Finally, for the collected destroyed objects, they are detected in a monitoring thread. If the object is still not reclaimed after multiple detections, it is determined that a memory leak has occurred. In addition to Activity and Fragment objects, you can monitor any object through the following platform APIs.

```
RumPro.startInspectLeakObj(leakObj);

/**
 * Set objects for Java memory leak detection
 *
 * @param leakObj Object to be detected
 */
public static void startInspectLeakObj(Object leakObj) {
    ......
}
```

Log Description

To monitor multiple leak objects in the same log file, the platform will delay for a period after detecting an object leak before capturing the hprof dump file. This allows multiple leak objects to be detected in the same log file. When capturing the hprof dump file, if logging is enabled, the following log output will be generated.

> **Note:**
> You can use the adb command to filter logs when capturing the hprof dump file. If there is log output, it indicates that detection has been successfully enabled: `adb logcat -s RMonitor_MemoryLeak_LeakInspector RMonitor_Heap_MemoryDumpHelper`.

- Logs indicating whether objects are leaked:

```
06-08 21:03:03.700 25492 25535 D RMonitor_MemoryLeak_LeakInspector:
Inspecting com.example.test.memory.TestActivityLeak@73988249-e8b3-
4f5b-a880-01f41cb7dc3c Time=1686229383700 count=1
06-08 21:03:04.032 25492 25535 D RMonitor_MemoryLeak_LeakInspector:
Inspecting com.example.test.memory.
......
06-08 21:04:33.796 25492 25535 D RMonitor_MemoryLeak_LeakInspector:
Inspecting com.example.test.memory.TestActivityLeak@73988249-e8b3-
4f5b-a880-01f41cb7dc3c Time=1686229473796 count=19
06-08 21:04:38.968 25492 25535 D RMonitor_MemoryLeak_LeakInspector:
Inspecting com.example.test.memory.TestActivityLeak@73988249-e8b3-
4f5b-a880-01f41cb7dc3c Time=1686229478968 count=20
06-08 21:04:44.138 25492 25535 D RMonitor_MemoryLeak_LeakInspector:
Inspecting com.example.test.memory.TestActivityLeak@73988249-e8b3-
4f5b-a880-01f41cb7dc3c Time=1686229484138 count=21
```

- Logs when dumping the hprof file:

```
06-08 21:07:31.882 25492 29639 D RMonitor_Heap_MemoryDumpHelper:
ReportLog dumpHprof: com.example.test.memory.LeftFragment@51fa088b-
b1f2-464c-8d70-972a1f26c360
06-08 21:07:33.791 25492 29639 D RMonitor_Heap_MemoryDumpHelper: dump
used 1885 ms
06-08 21:07:35.304 25492 29639 D RMonitor_Heap_MemoryDumpHelper:
leakFlag=true,ZipFile=true,leakName=com.example.test.memory.LeftFragme
nt@51fa088b-b1f2-464c-8d70-
972a1f26c360,dumpPath=/storage/emulated/0/Android/data/com.example.sdk
app/files/Tencent/RMonitor/main/Log/dump_com.example.test.memory.LeftF
ragment@51fa088b-b1f2-464c-8d70-972a1f26c360_leak_23-06-08_21.07.33.z
```

- After capturing the log files, they will be uploaded immediately in a Wi-Fi environment. In other cases, the upload will occur after the next process startup. The logs when uploading are as follows:

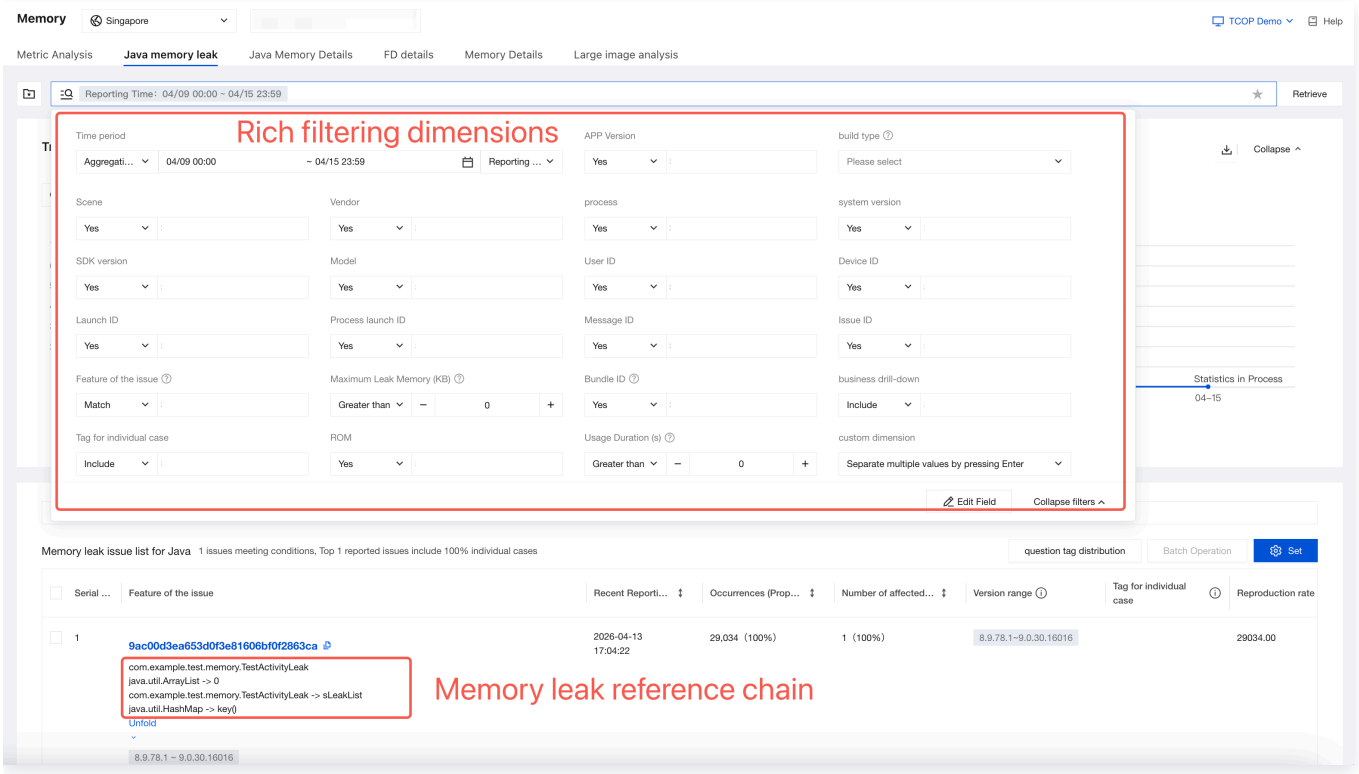
```
06-09 15:58:39.618 23814 23875 I RMonitor_report_File: url: *******
sub_type: activity_leak
```

Console Feature Description

The Java memory leak Web pages mainly consist of two pages: **Issue List** and **Issue Details**.

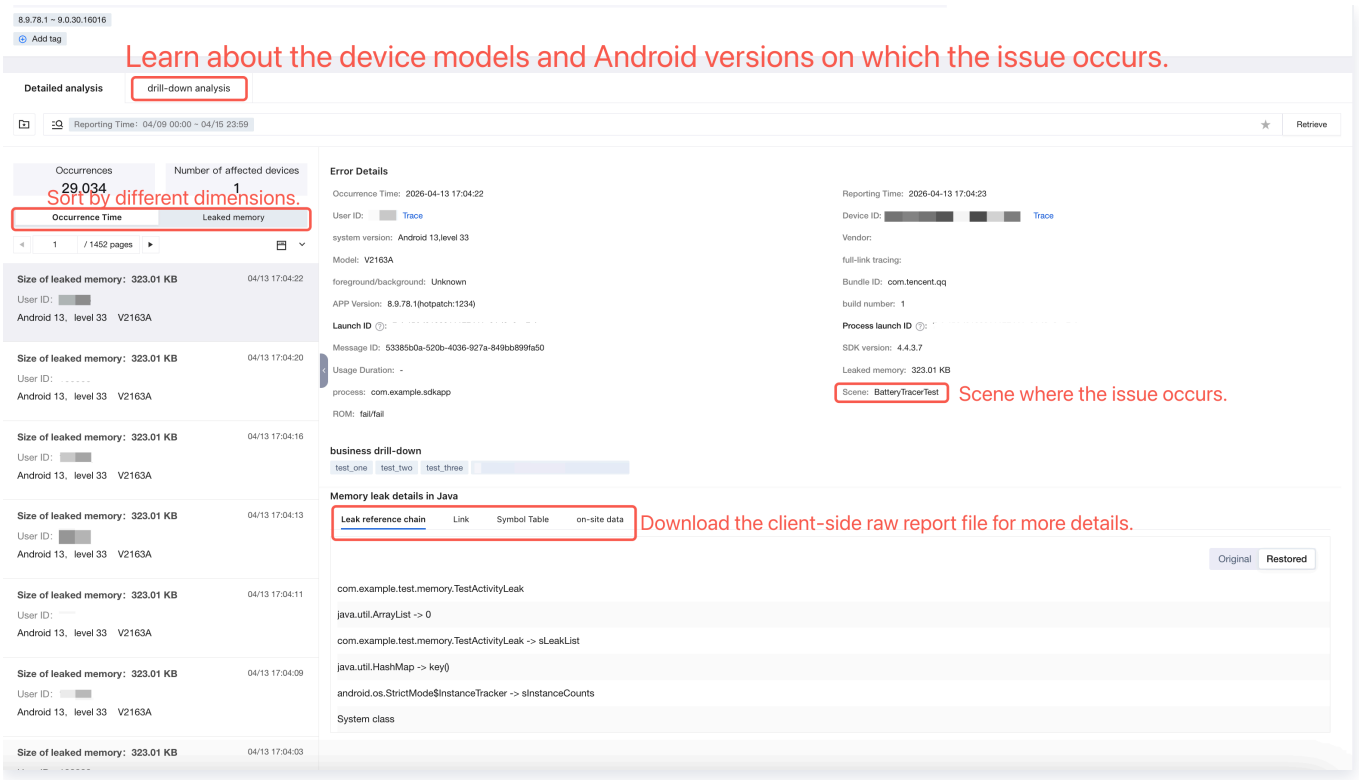
Issue List

- The issue list for Java memory leaks is formed by using memory leak reference chains as features to group cases with the same characteristics.
- The issue list supports rich filter criteria. For specific usage, see Query.



Issue Details

The issue details page provides a clear display of the leak reference chain and enables users to download the memory snapshot at the time of leakage for detailed analysis.



Java Memory Details

As mentioned earlier, after excessive Java memory usage, it not only triggers GC, causing issues such as stuttering and ANR, but more severely, it can directly lead to OOM crashes, severely affecting user experience. The Java memory details feature, after enabling, continuously monitors the current virtual machine heap memory usage. When it exceeds a preset threshold, it automatically collects relevant log information and reports it to the backend. This feature mainly provides the following services:

- The feature implements a capability similar to MAT's "Top Consumers." After log information is reported to the backend, it can analyze issues of concern to the business, such as **Single Report Analysis**, **a single large object**, and **intensive objects**. This eliminates the need for users to manually capture hprof files and drag them to the PC-side for analysis using MAT tools, significantly improving analysis efficiency.
- When frequent GC occurs or the Java heap memory usage continuously exceeds a certain threshold of its maximum value multiple times, a child process automatically dumps the Java heap memory dump file on the mobile end. Dumping heap memory dump files in the background has minimal impact on the user experience.
- It can be used online, leveraging the platform's existing translation capabilities to translate class names and member variable names. Afterward, it intelligently extracts clustering key features to group similar issues together. This automated translation and clustering allow users to easily focus on Top issues, eliminating the cumbersome operations of manual translation required by tools like MAT.

> **Term Explanation:**
> - Single report analysis: Currently supports detecting leaked Activity objects.

- Individual large objects: Analogous to MAT's "Biggest Objects" feature, this refers to a single Java object or class whose Retained Size exceeds a certain threshold. The threshold is currently specified by the platform backend.
- Clustered objects: Analogous to MAT's "Biggest Top-Level Dominator Classes" feature, this refers to a Java class where, although individual objects are small, the total Retained Size of all its Java object instances exceeds a certain threshold. The threshold is currently specified by the platform backend.

Enabling Method

Upgrade to **SDK 4.4.2.2** or later is required to support the Java memory details monitoring feature. On the client side, execute the following code and enable it by adjusting the sampling rate configuration in the backend.

1. The client should add the following code when initializing the SDK.

```
public static void initRumPro(Context context) {
    // 1. Pre-build initialization parameters. These initialization
parameters are required.
    String appID = "a278f01047"; // [Required] Obtain the appID of the
application from the application list on the platform.
    String appKey = "1e5ab6b3-b6fa-4f9b-a3c2-743d31dffe86"; //
[Required] Obtain the appKey of the application from the application
list on the platform.
    RumProBuilder builder = new RumProBuilder(appID, appKey);

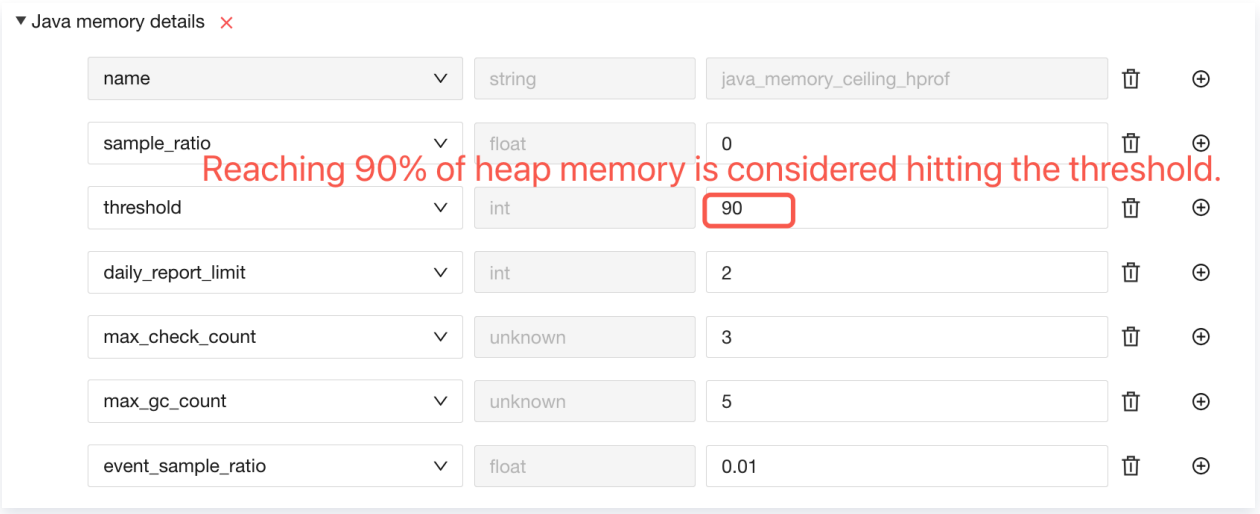
    ......

    // 2. Enable Java memory details
    build.addMonitor(RumProMonitorName.MEMORY_JAVA_CEILING);

    // 3. Initialization. This must be called.
    RumPro.init(context, builder);
}
```

2. On the Setting > SDK configuration edit configuration page, enable the following configuration items:
   - **sample_ratio**: Controls the user sampling rate, indicating how many devices will enable this feature.
   - **event_sample_ratio**: Controls the event sampling rate, indicating whether to report when the Java memory threshold is reached.

- **threshold**: Sets the timing for log file capture. 90 indicates that log capture starts when the heap memory reaches 90% of its maximum value, where the maximum heap memory corresponds to Runtime.getRuntime().maxMemory().



Log Description

- Log for successful feature enabling:

```
07-04 10:xx:xx.xxx 14546  1819 D RMonitor_MemoryCeiling: Start
MemoryCeilingMonitor
07-04 10:xx:xx.xxx 14546  1819 D RMonitor_MemoryCeiling: start detect
memory ceiling
```

- Detected Java memory threshold reached, starting to dump log files:

```
07-04 10:xx:xx.xxx  3713  3713 I .example.sdkapp: hprof: heap dump
"/storage/emulated/0/Android/data/com.example.sdkapp/files/Tencent/RMo
nitor/main/Log/dump_LowMemory_24-07-04_10.20.41.hprof" starting...
```

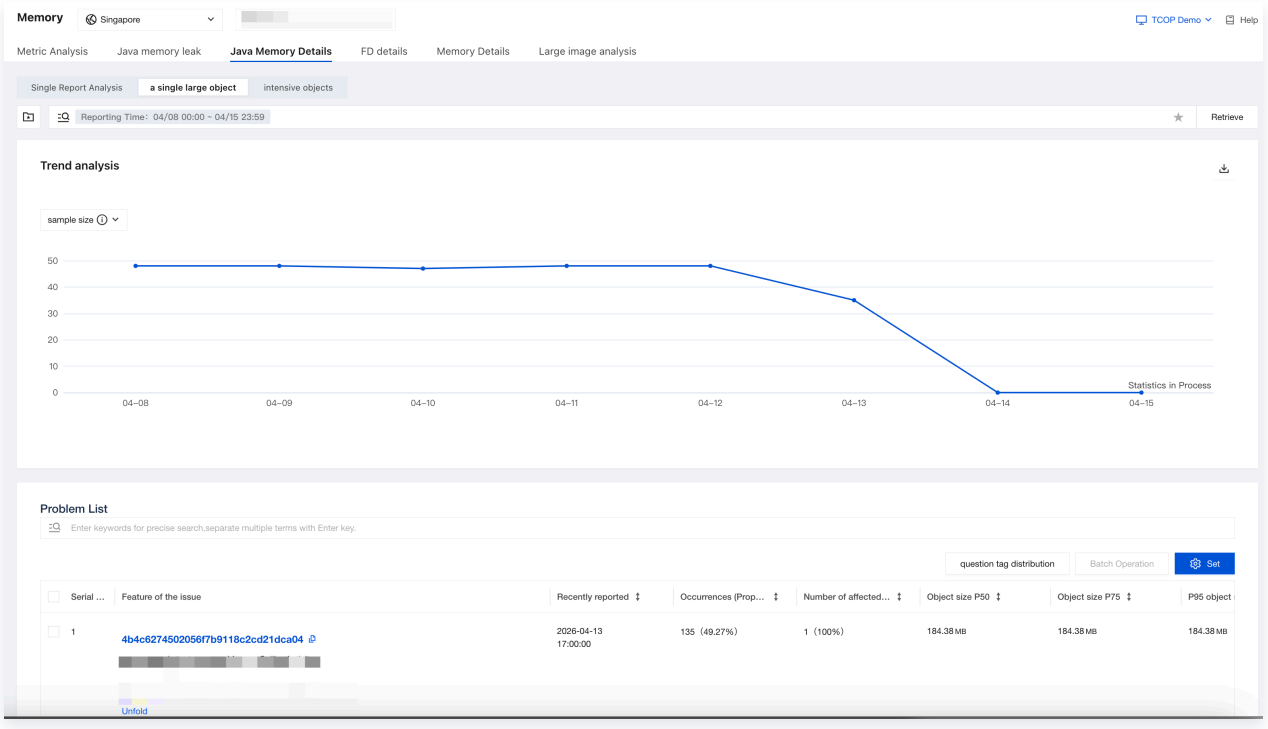
- Logs for reporting log files to the platform backend (reported in real-time under Wi-Fi network; reported after restart under other network conditions):

```
07-04 10:xx:xx.xxx 14546  1918 I RMonitor_report_File: url:
https://xxx.qq.com/v1/xxxx/upload-file?
timestamp=1720059647340&nonce=7153357010e6227230d5deb79ce73ed7,
sub_type: java_memory_ceiling_hprof
```

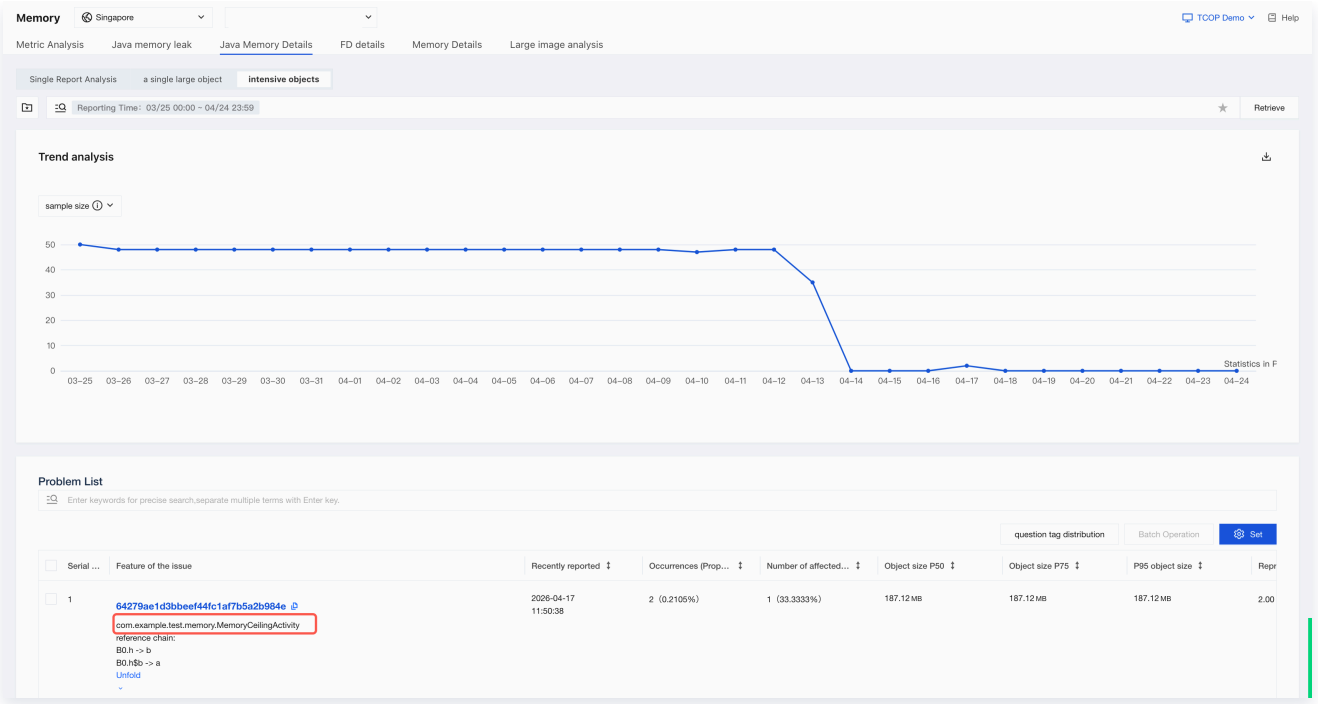
Console Feature Description

When the platform backend receives a Java memory threshold log from a user, it uses a self-developed heap dump analysis tool to identify issues such as **a single large object**, **intensive objects**, and **Single Report Analysis**. Each issue is characterized by key features used for clustering. Thus, a single report typically corresponds to multiple platform issues. The problem list page for single large objects and dense objects primarily consists of three components: filter options, trend analysis, and the issue list. Among these:

- **a single large object & intensive objects**
  - In the query area, many filter options are provided. Among them, the maximum heap memory of the virtual machine and problem characteristics can accurately filter out specific issue problems. For other filter options, refer to Query.
    - Maximum Heap Memory of Virtual Machine (MB): This is a dropdown selection box corresponding to the value of Runtime.getRuntime().maxMemory().
    - Problem characteristics: Specific issues can be filtered using various methods such as "match".
  - Trend analysis:
    - Sample size: the number of individual cases in the problem list.
    - It also supports switching between multiple metrics such as affected device count, affected user count, startup count, and connected device count. Users can select as needed.
  - Issue list: The issue list is categorized by issue, with each issue having its own key characteristics. The key characteristics mainly fall into three types: "reference chain", "dominator tree", and "plain text". The reference chain represents the shortest path from a large object to the GC Root. For large objects that are files or threads, the file path or thread name is used as the clustering feature.

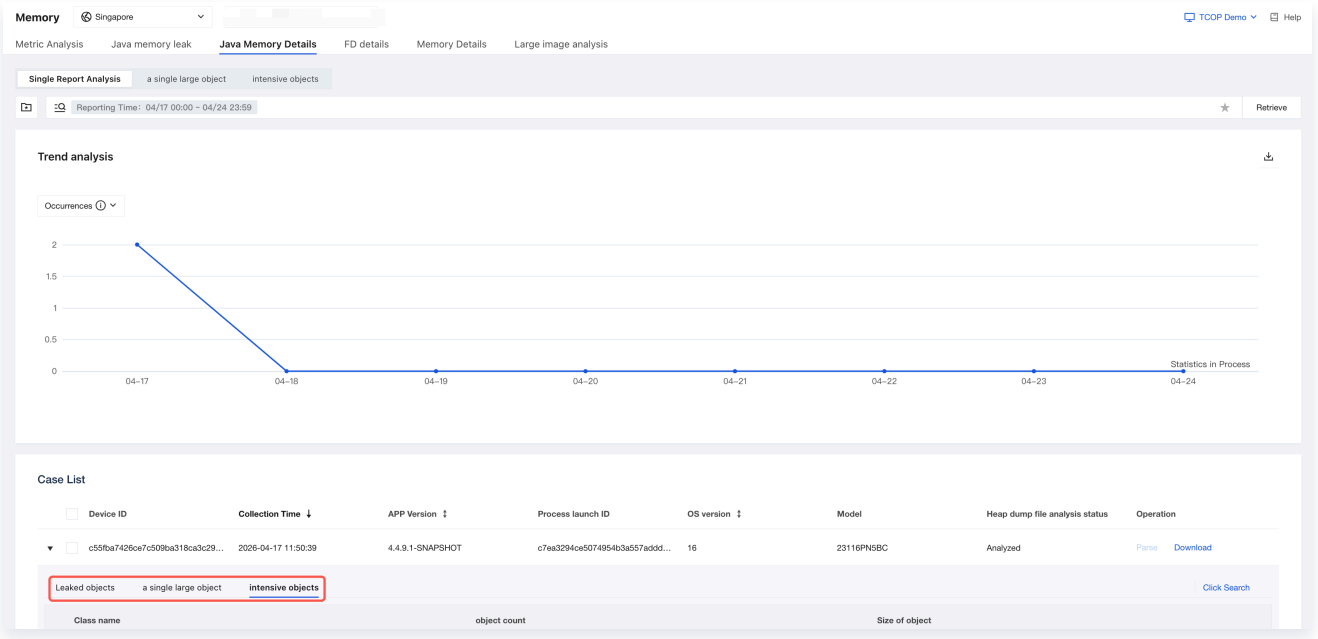


The clustering feature of dense large objects is the class name of that large object:



- **Single Report Analysis**

  The issue list for single large objects and dense objects displays the results after analysis and clustering, whereas the issue list for single-report analysis is unclustered. Each user report occupies a separate entry in the issue list. Clicking on an entry reveals whether that specific report contains memory leaks, along with all single large object and dense object issues. This allows for convenient inspection of the detailed causes behind a specific Java memory threshold event. Additionally, it provides the feature to batch download heap dump hprof files.

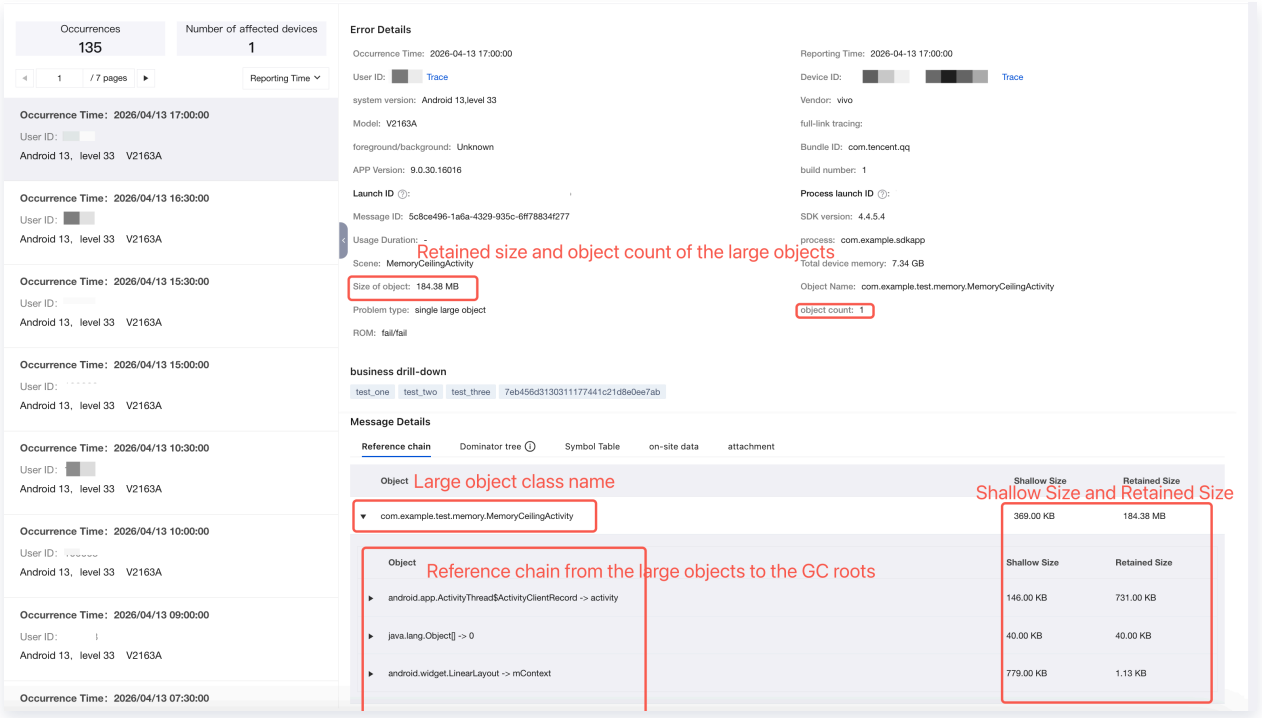


- **Issue Details Page**

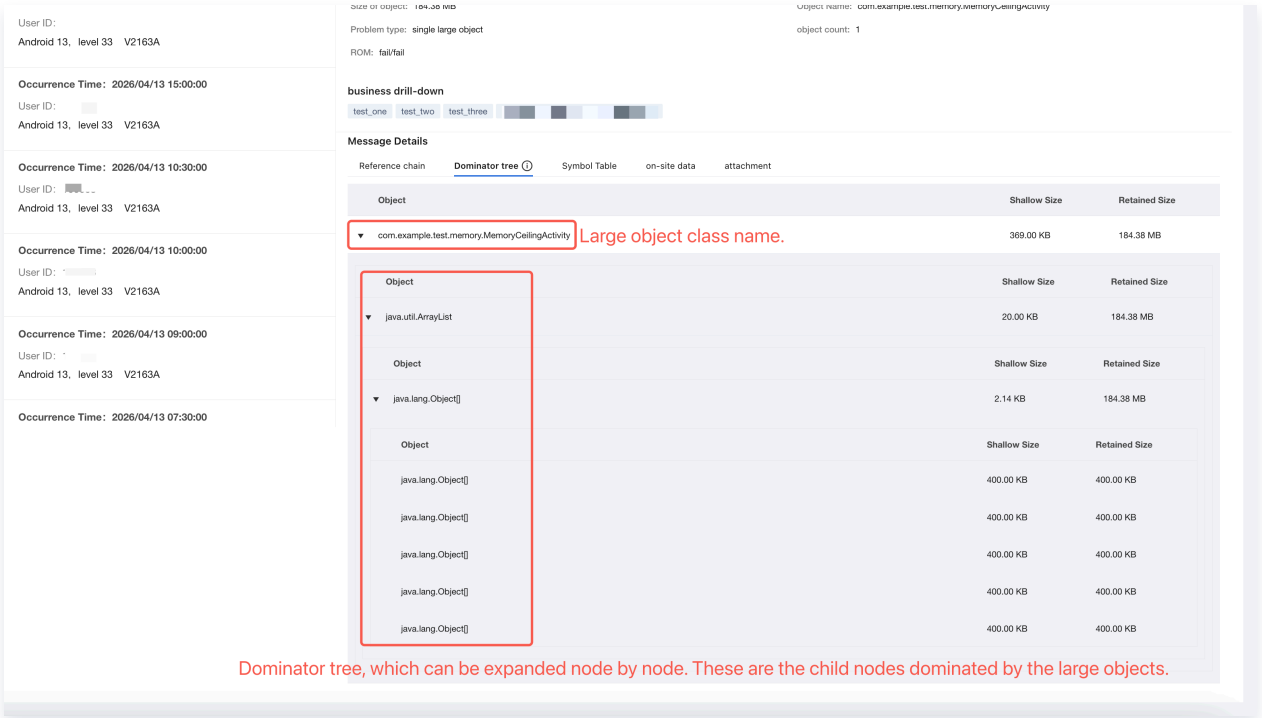
  The issue details page for single large objects and dense objects displays all GC Root reference chains and the memory dominator tree of the large object. To reduce display hierarchy, the dominator tree by

default only shows child node content greater than 10% of the parent node's size. If a leaf node is an array, its elements will continue to be printed. Additionally, in the Attachments Tab, you can download the original heap dump hprof file. After conversion using the hprof-conv tool, it can be locally analyzed with tools like MAT.

- Reference chain:



- DominatorTree:



FD Details

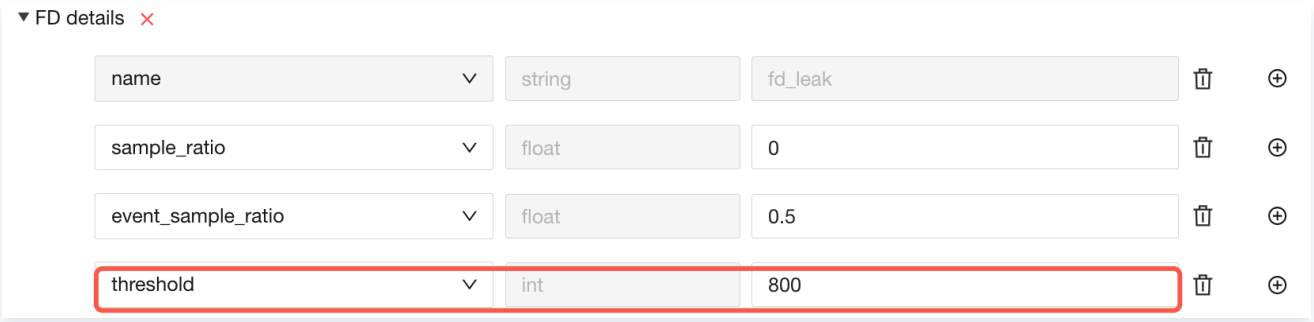
The number of FDs a process can use in Android is limited. Once the maximum is exceeded, the process can no longer allocate FD resources. If an FD is opened but not closed promptly, it leads to FD leaks, which can cause issues such as crashes, black screens, and freezes. To address this, the platform has developed a capability to monitor changes in the number of FDs in a process. If a predetermined threshold is reached, detailed FD information at that moment, along with the allocation function call stack for each FD, is reported to the backend.

Enabling Method

- The client adds the following code during initialization.

```
RumProBuilder builder = new RumProBuilder(appID, appKey);
build.addMonitor(RumProMonitorName.FD_ANALYZE);
```

- On the Setting > SDK configuration Edit Configuration page, configure according to the following configuration items.



  - sample_ratio: Only sampled users will have the FD detail monitoring feature enabled. 0 turns off this feature for all users, while 1 enables it for all users.
  - event_sample_ratio: The probability of reporting when a sampled device reaches the FD threshold. 0 means no FD threshold events are reported, while 1 means all FD threshold events are reported.
  - threshold: Sets the threshold for FD reaching the limit. For example, if set to 800, logs will be reported when the number of FDs reaches 800.

Log Description

```
//1. FD threshold monitoring feature enabled successfully log

05-16 16:33:38.737 19194 11716 I RMonitor_FdLeak_Monitor: fd leak
monitor started.

//2. Log when FD threshold reached is detected.
05-16 16:33:48.815 19194 11716 I RMonitor_FdLeak_Trigger: top fd:
FdStatisticItem{type=8, count=1654}
```

```
//3. Log for dumping reporting information
05-16 16:33:48.863 12359 12359 I .example.sdkapp: hprof: heap dump
"/storage/emulated/0/Android/data/com.example.sdkapp/files/Tencent/RMoni
tor/main/fd_leak/dumpN/A_root/heap.hprof" starting...


//4. Log reporting (reported immediately under WiFi; reported after
process restart under other conditions)
05-16 16:37:17.697 19195 13956 I RMonitor_report_File: url:
https://rmonitor.qq.com/v1/*********/upload-file?
timestamp=1715848637695&nonce=44ef8384331c486250d430b78a29a18a,
sub_type: fd_leak
```

Console Feature Description

The Web page for FD details is similar to that for Java memory leaks. For specifics, refer to Java Memory Leak Console Feature Description.

Native Memory Details

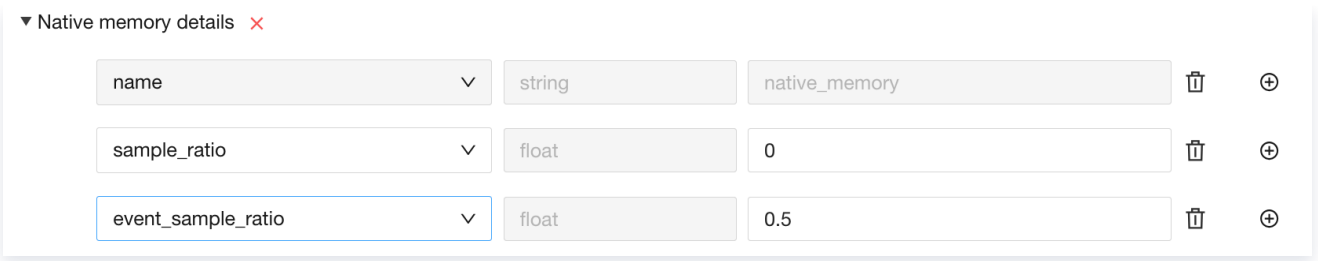
Native memory usage monitoring has always been a challenge on the Android platform. Google has developed various tools for this purpose, but most are designed for local debugging and often encounter issues such as compatibility problems and crashes. The platform's Native memory reaching the threshold intercepts memory allocation and deallocation in processes, captures function call stack information for memory allocations, and reports these unreleased memory allocations along with their call stack information to the backend when the process's memory (PSS or VSS) reaches a certain threshold. Users can search and analyze these memory threshold issues through the platform's Web page.

Enabling Method

- The client adds the following code during initialization.

```
RumProBuilder builder = new RumProBuilder(appID, appKey);
build.addMonitor(RumProMonitorName.NATIVE_MEMORY_ANALYZE);
```

- On the Setting > SDK configuration Edit Configuration page, configure according to the following configuration items.



- sample_ratio: The device sampling rate, which indicates the proportion of devices allowed to enable relevant monitoring items.
- event_sample_ratio: The event sampling rate, which controls the sampling rate for dumping memory and reporting data when a memory threshold (VSS or PSS threshold) is reached.

Console Feature Description

The Web page for Native memory details is similar to that for Java memory leaks. For specifics, refer to Java Memory Leak Console Feature Description.

Big Graph Analytics

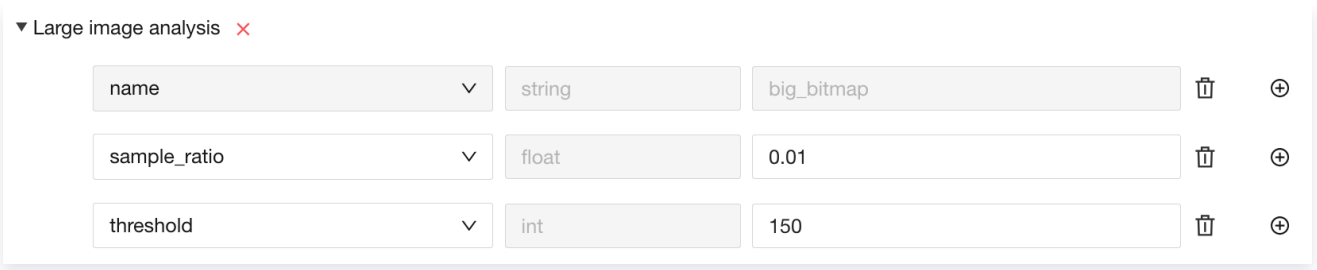
Large bitmaps, also known as over-decoded images, refer to situations where the decoded bitmap's width and height exceed those of the View containing it. The large bitmap monitoring mechanism registers an ActivityLifecycleCallback. When onGlobalLayout occurs for the DecorView, it traverses all Views in the DecorView tree to check whether the width and height of the bitmap corresponding to each View's background or src image exceed the View's actual dimensions. If so, that bitmap is identified as a large bitmap (over-decoded).

Enabling Method

- The client adds the following code during initialization:

```
RumProBuilder builder = new RumProBuilder(appID, appKey);
build.addMonitor(RumProMonitorName.MEMORY_BIG_BITMAP);
```

- On the Setting > SDK configuration Edit Configuration page, configure according to the following configuration items.
  - **sample_ratio**: The user sampling rate for the large bitmap monitoring feature. 0 indicates that all users disable this feature, 1 indicates that all users enable this feature.
  - **threshold**: The large bitmap determination threshold. 150 represents 150%. A bitmap is identified as a large bitmap when (Bitmap width / View width > threshold) or (Bitmap height / View height > threshold).



Log Description

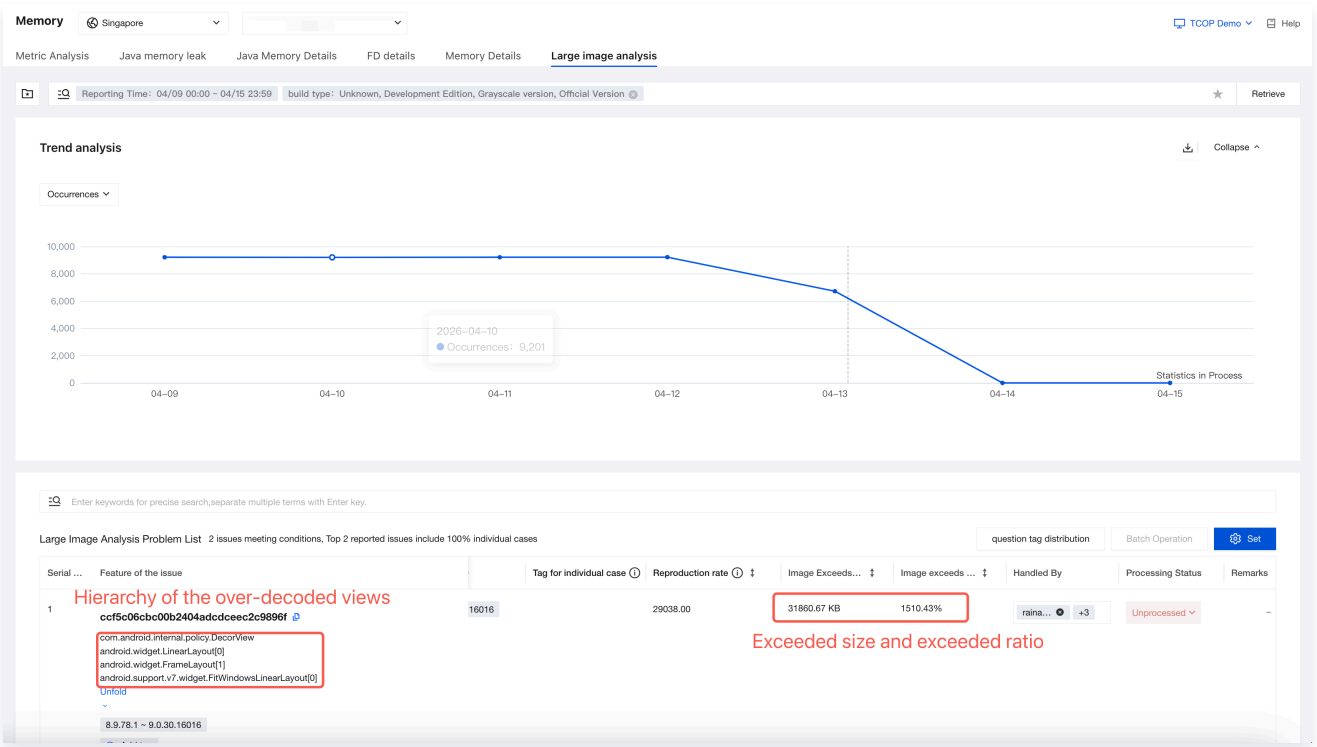
After a large bitmap is detected by the platform, it attempts to immediately report the issue, and typically prints the following log during the reporting process:

```
06-09 17:00:23.000 31628 31822 D RMonitor_report_Json: url: *****
eventName: BigBitmap, client_identify:  ********
```

Console Feature Description

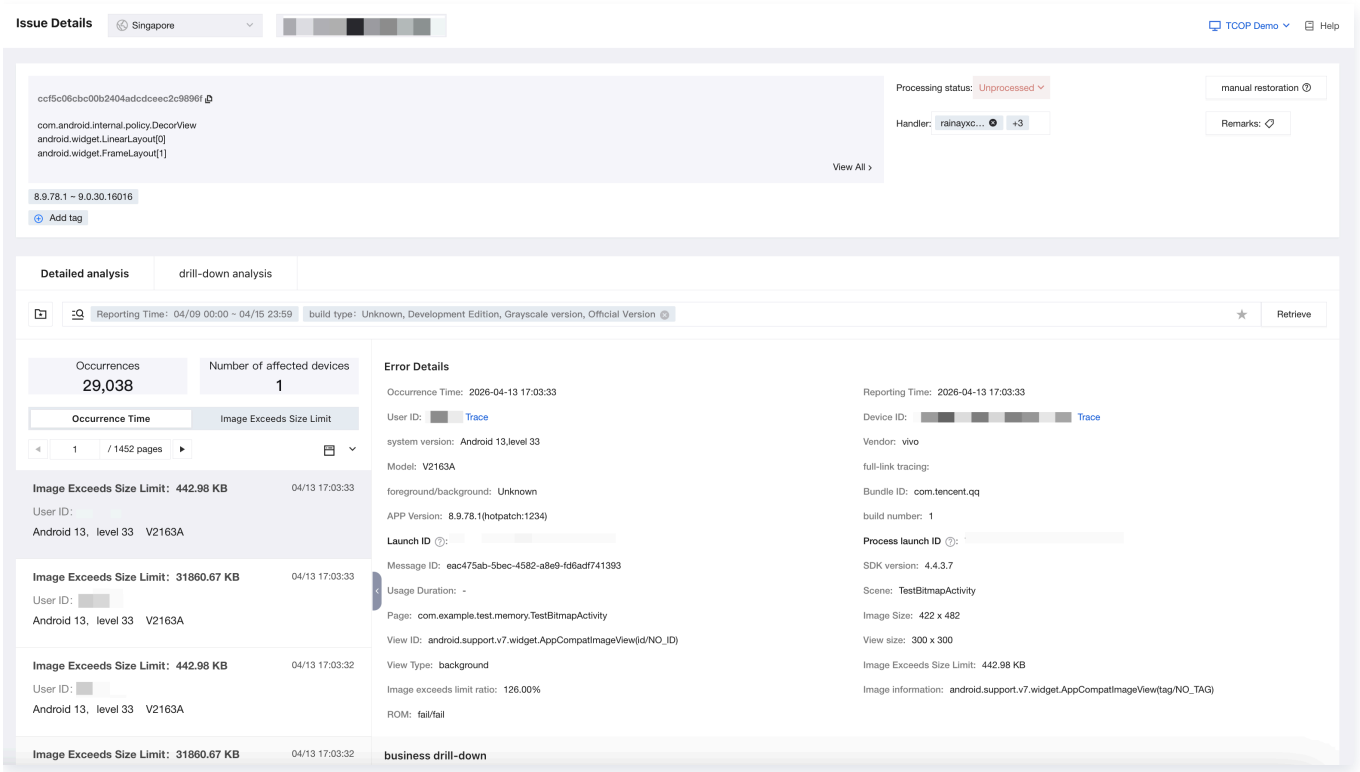
Problem List

- By using the layout hierarchy of the over-decoded image's View as a feature aggregation problem.
- It supports sorting by exceeding image size and exceeding ratio, allowing for quick identification of images with severe decoding excess.



Issue Details

Users can identify corresponding business scenarios based on contextual information and the layout hierarchy of the displayed View.



Field	Description
Image Size	Refers to the size of the image after decoding, measured in pixels, for example: 422 * 482.
View Size	Refers to the size of the View that carries the image, measured in pixels.
Image exceeds limit ratio	Refers to the ratio of the image's pixel count to the View size, calculated as (Image pixels / View pixels) – 1 to derive the excess percentage. For example: ((422*482) / (300*300)) – 1 = 126.00%.
Image Exceeds Size Limit	It is calculated as (Total image pixels – Total View pixels) / 1024 to obtain the exceeding value in KB. For example: ((422*482*4) – (300*300*4)) / 1024 = (813,616 – 360,000) / 1024 = 442.98 KB.
View Type	This field has two values: • background represents the background image resource obtained by calling View.getBackground. • source represents the decoded image resource obtained by calling View.getDrawable.
Page	From the screenshot above, this case occurs on the com.example.memory.TestBitmapActivity page, which is the Activity hosting the View

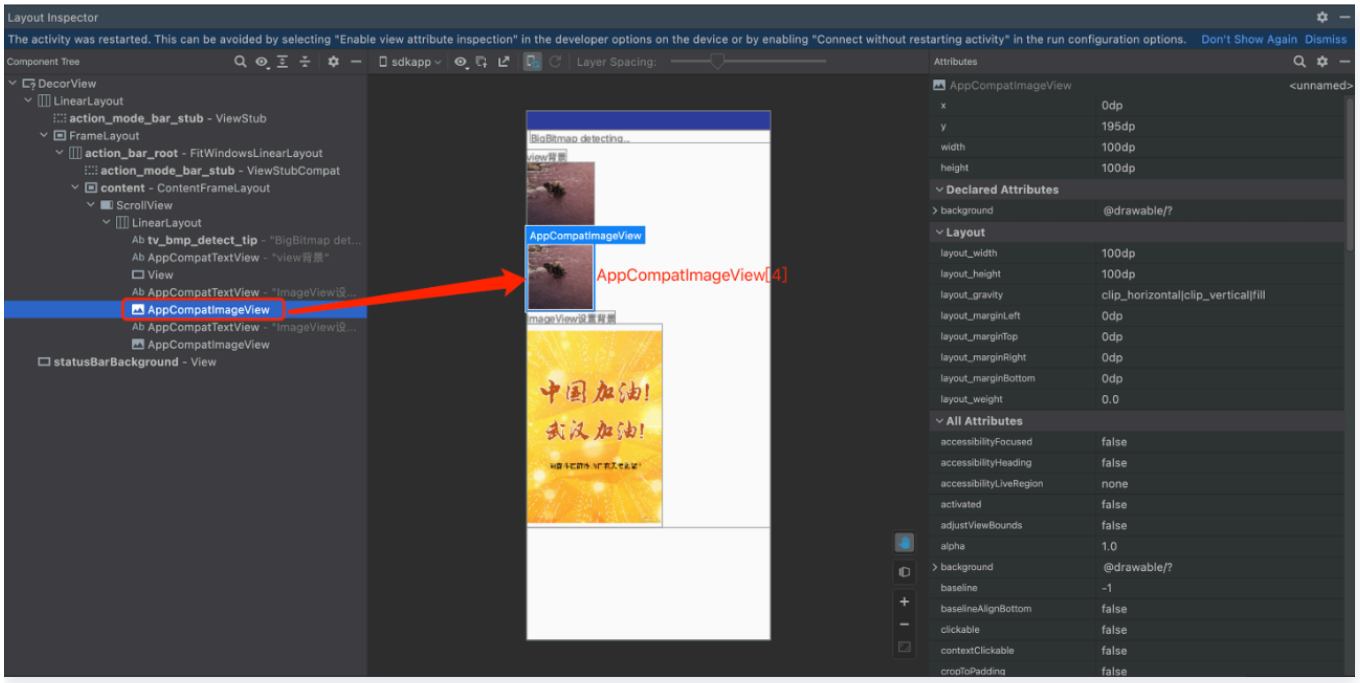
	with exceeding image size and cannot be customized.
Scene	The scene field defaults to the current Activity, but its value can be customized by the business using Bugly.enterScene and Bugly.exitScene. For example, if an Activity contains multiple Fragment switches, the scenario field can identify which Fragment is active after switching.

Example Analysis

Taking the issue in the above problem details as an example, the Page field indicates that the image size exceeding occurred on "com.example.memory.TestBitmapActivity". Additionally, the Page field can be combined with the Scenario field to provide more precise contextual information when locating issues. Then, from the stack details, it can be seen that the over-decoded element is an AppCompatImageView, and the hierarchy of the View can be observed.

```
com.android.internal.policy.DecorView
android.widget.LinearLayout[0] // Here, 0 represents the first child
View of the DecorView.
android.widget.FrameLayout[1] // Similarly, here, 1 represents the
second child View of the LinearLayout.
android.support.v7.widget.FitWindowsLinearLayout[0]
android.support.v7.widget.ContentFrameLayout[1]
android.widget.ScrollView[0]
android.widget.LinearLayout[0]
android.support.v7.widget.AppCompatImageView[4] // The View containing
the final large image is the fourth child View of
android.widget.LinearLayout.
```

It is the fourth child View of the LinearLayout. Therefore, we can locate this View using tools such as Android Studio's **Layout Inspector**, as shown in the figure below:



iOS

Overview

Memory Monitoring is used to monitor and measure the memory usage of apps in production. As a critical resource in devices, memory is shared among various processes within the system. According to iOS system policies, excessive memory usage in the foreground may result in termination by the system; similarly, occupying substantial memory resources in the background can lead to app termination to reclaim resources for foreground apps. Given this, the rational and efficient use of memory resources is particularly crucial. Therefore, the platform provides a memory monitoring module to help businesses evaluate app memory consumption in production, offering essential diagnostic information to identify and optimize memory-related issues.

Metric Analysis

Memory metrics measure the overall memory usage of the App from a holistic perspective, primarily consisting of two indicators.

Front-end/Back-end Memory Peak

Peak Value refers to the maximum physical memory (phys_footprint) used by the App during a single process lifecycle. A higher peak memory value indicates greater memory consumption under extreme conditions. When active in the foreground, a higher peak increases the likelihood of encountering memory limits; similarly, in the background, a higher peak raises the probability of the App being terminated by the system. Therefore, the peak metric is categorized by foreground/background states as **Front-end memory peak** and **Back-end memory peak**.

- **Front-end memory peak**: The higher this value, the greater the probability of FOOM occurrence. Generally, its upward trend (especially P99) is positively correlated with the FOOM rate.
- **Back-end memory peak**: The last memory value collected by the SDK when the app moves to the background. Higher background memory usage increases the probability of the app being terminated by the system. Therefore, it is necessary for businesses to optimize the app's memory usage after backgrounding. Since this metric is collected in-process by the SDK, its value tends to be slightly higher than the system's actual measurement, serving primarily as a reference value.

The peak metric, similar to other metrics, supports querying and comparison with different time granularities and filter criteria. For specific operations, please refer to the Integration Guide.

FOOM rate

FOOM (Foreground Out Of Memory): This is defined as the scenario where an App is terminated by a SIGKILL signal after triggering the system's memory limit while in the foreground.
Since **FOOM is a Crash caused by the SIGKILL signal from the system**, it differs from typical crashes triggered by internal exceptions (which result from abnormal logic execution within the process). Therefore, traditional Crash rate metrics do not account for this type of exit. Additionally, since the SIGKILL signal cannot be directly captured within the process, it is impossible to record accurate FOOM exit events.
In the platform, FOOM events are determined by comprehensively analyzing various states during the App lifecycle, such as whether the App is in the foreground/background, whether any captured exceptions occurred, and the App's last memory usage, to infer whether a FOOM event has occurred. Therefore, FOOM determination results may include some misjudged data (**misjudgment refers to incorrect identification of the exit cause, though it always indicates an abnormal exit**).
The iOS system terminates apps not only when memory usage is excessively high, but also under conditions such as device overheating, excessive CPU scheduling, or triggering excessive I/O operations.

> **Note:**
> **The FOOM rate is the ratio of FOOM occurrences to the number of process launches**, measuring the frequency of FOOM events. Additionally, metrics are included from both device and user perspectives: Device FOOM Rate and User FOOM Rate. Both rates are calculated after deduplication based on Device ID and User ID.

Despite containing a certain degree of misjudgment, this metric still effectively measures the severity of Apps being terminated by the system in the foreground due to excessive memory usage.

Memory diagnostic information

To help businesses optimize the above metrics, the platform provides relevant diagnostic information alongside the metrics. The primary purpose of diagnostic information is to offer insights into key memory usage patterns, enabling businesses to analyze the main causes of excessive memory consumption and

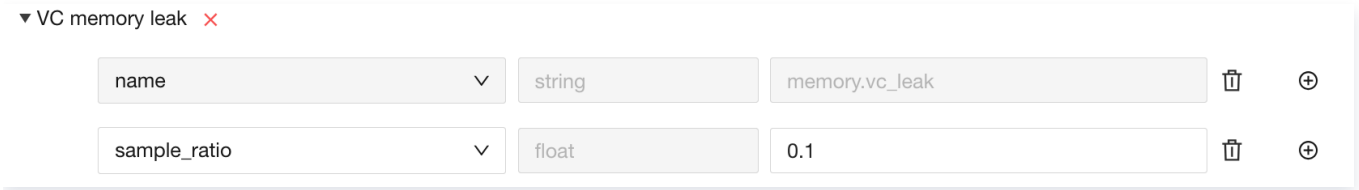
determine how to optimize it. Unlike typical Crash issues, memory usage problems differ in nature. The data collected and provided by the platform does not pinpoint the exact cause of FOOM occurrences.

Note:

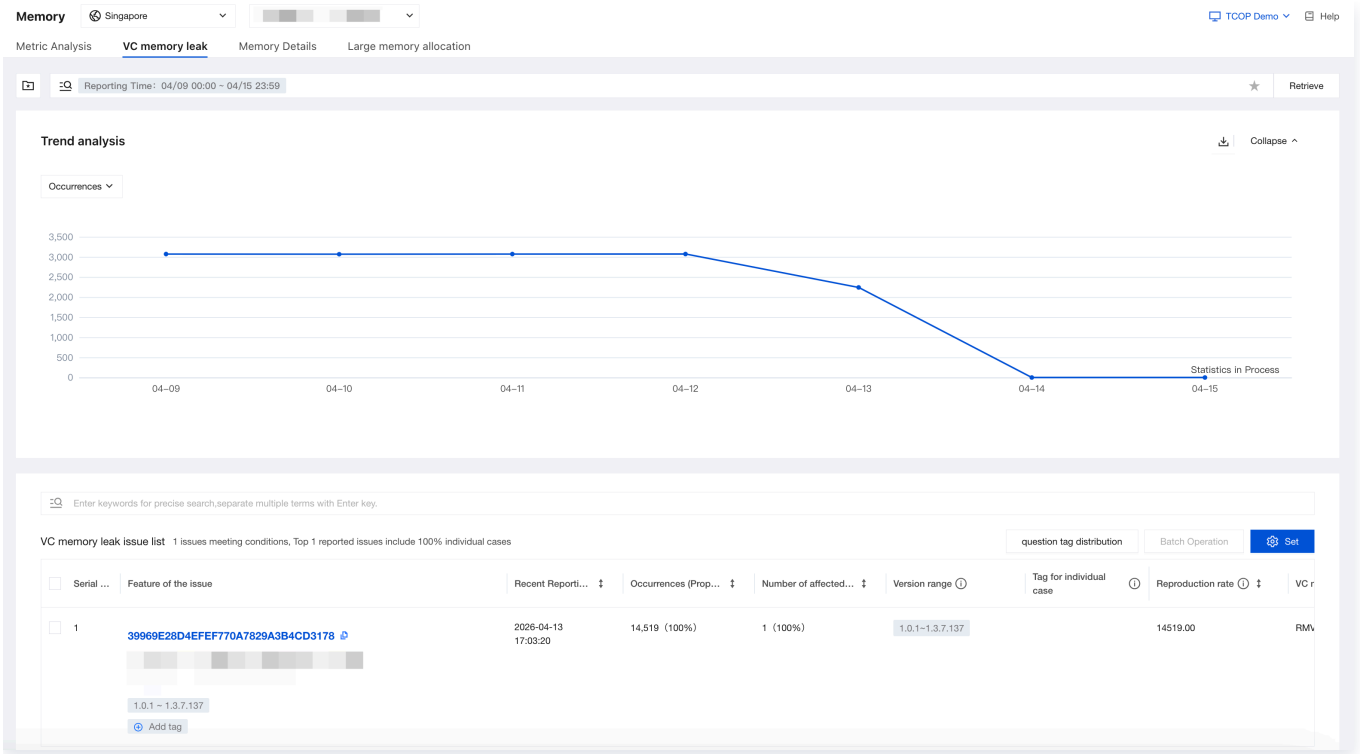
- In general crashes, the issue occurs when the program encounters an exception during execution at a specific point. Therefore, the captured call stack and contextual data precisely indicate the point of exception occurrence, which can be analyzed and resolved.
- The essence of FOOM issues lies in the excessive consumption of memory resources. Therefore, a single memory allocation record cannot conclusively indicate unreasonable memory usage. It requires analyzing all memory allocations within the App to identify business logic that consumes large and unreasonable amounts of memory, followed by optimization based on business requirements. For this reason, diagnosing FOOM memory issues necessitates collecting more information, which inherently introduces performance overhead in the monitoring logic itself. To prevent this monitoring overhead from impacting normal business operations, the platform employs a sampling mechanism (default 1%) during memory diagnostic data collection. Consequently, detailed information is unavailable for the vast majority of individual FOOM cases.

VC memory leak

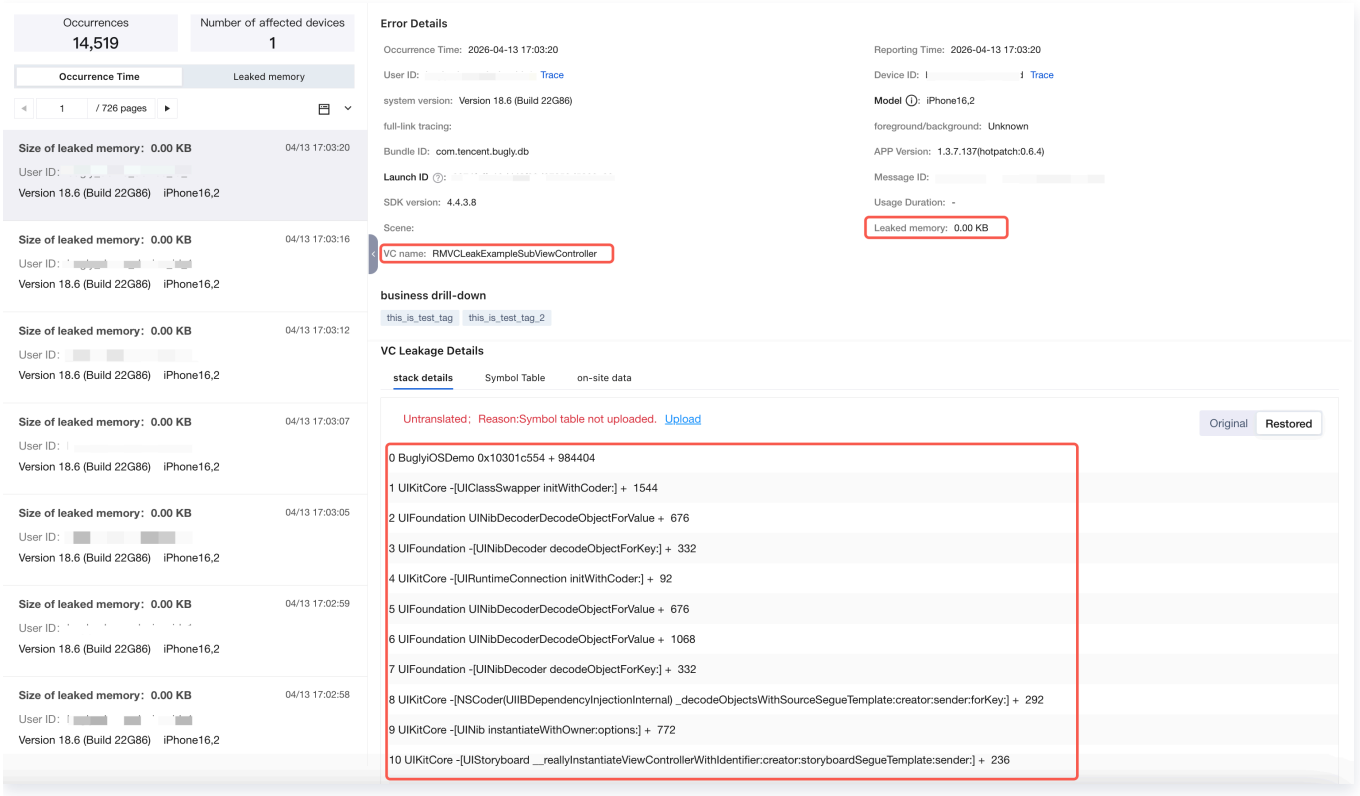
VC leaks refer to situations where UIViewController and its related child objects in iOS apps are not released after use (e.g., pop, dismiss, remove) (with a default 30-second delay for checks). In native iOS applications, the UIKit framework uses UIViewController as the foundational object for organizing product logic and pages. Consequently, most core app logic revolves around UIViewController. The platform monitors all key behaviors of UIViewControllers and tracks related metric changes. Based on this, unreleased VC objects after use are flagged for abnormal reporting. This allows businesses to verify whether such VC objects are properly released, preventing resource waste from unnecessary retention. You can enable this feature via the following configuration in the Setting > SDK configuration edit page:



In certain scenarios, businesses may intentionally retain specific VC objects without releasing them due to operational requirements. The platform cannot distinguish such cases and will still report them. If businesses wish to exclude these reports, they can add them to the allowlist by following the SDK configuration usage guidelines to filter them out.
Similar to large memory allocations, VC leaks primarily provide the creation call stack and name of the VC, in addition to basic information.



VC Leak Details:



It should be specifically noted that in VC leak cases, the value of "**Leaked Memory**" represents the difference between the phys_footprint value when the VC was popped (dismissed, removed) and its value at creation time. If the result is negative, it is recorded as 0. Therefore, this serves as a reference value rather than the actual memory leak amount. During usage, more attention should be paid to whether VCs are reasonably retained, and one should not be overly concerned with the specific leaked memory value.

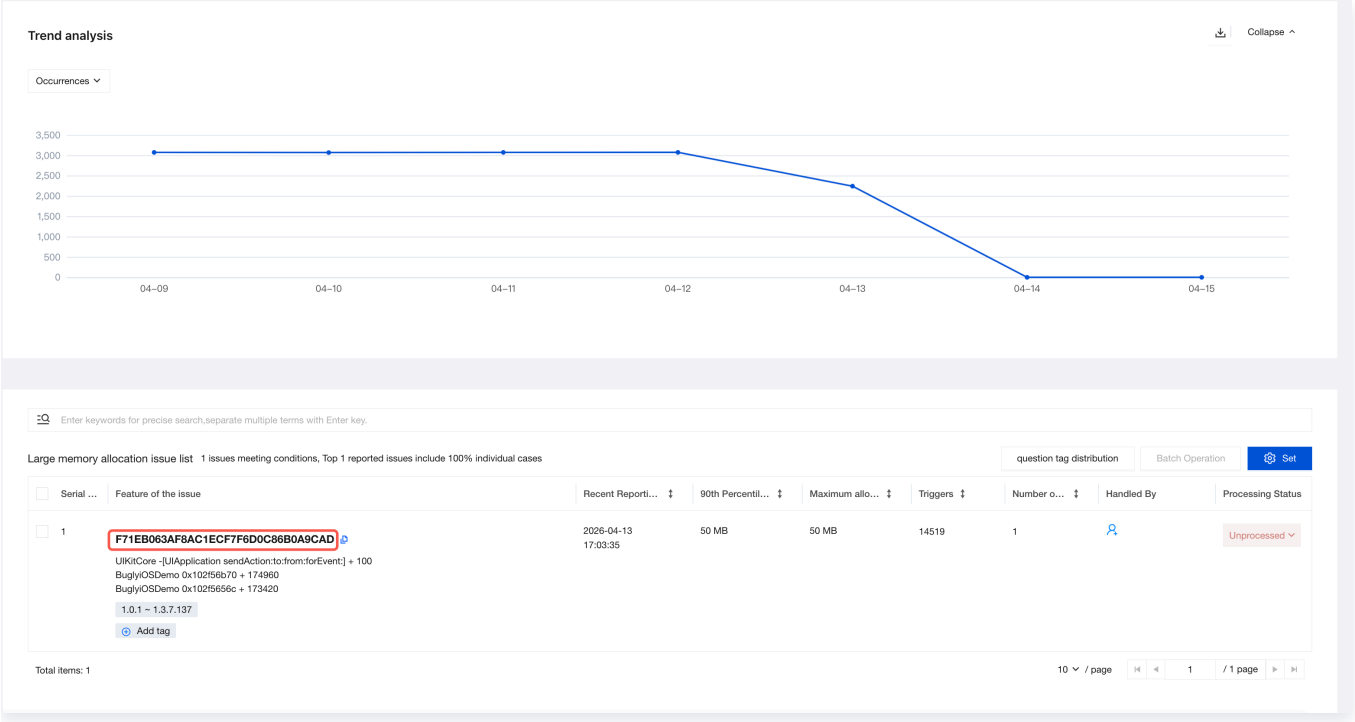
Memory Details

Single-report analysis, single large objects, and dense objects metrics are similar to those on Android. For details, see Android Memory Details.
Reported FOOM cases refer to the detailed information reported by the SDK after detecting a FOOM exit event. The information provided is critical for resolving individual FOOM issues. For details on FOOM case specifics and primary analysis methods, see FOOM Case Details.

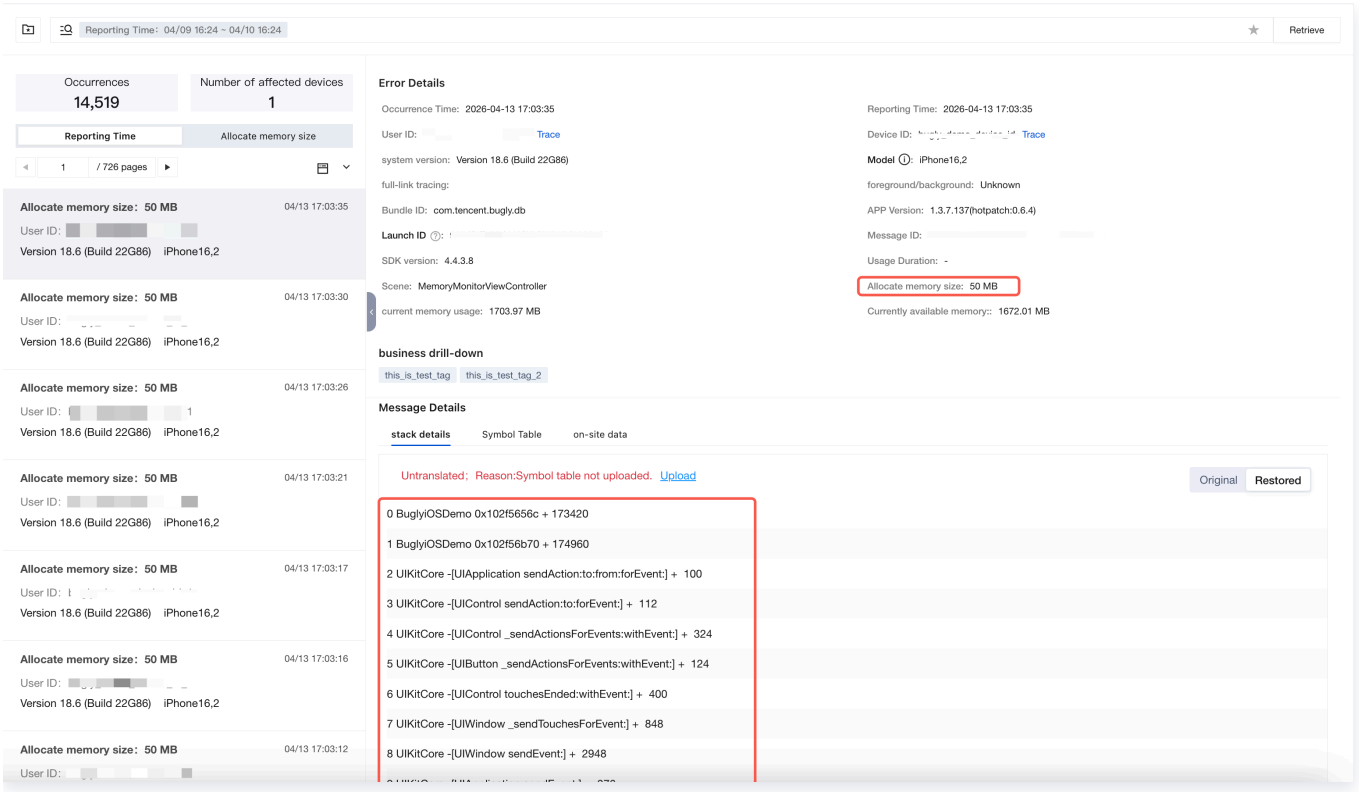
Large memory allocation

Large memory allocations serve as an auxiliary monitoring mechanism, primarily tracking abnormal memory allocation events (single allocations exceeding the specified threshold, default 10 MB) during app runtime. Therefore, **the reported data is primarily for reference purposes. Businesses should focus on whether such allocations are reasonable rather than assuming they necessarily indicate abnormal issues**. You can enable this feature via the following configuration in the Setting > SDK configuration edit page:

In large memory allocations, the platform primarily provides the call stack that triggered the allocation and the allocation size, along with other contextual information such as timestamps and device details. Similar to other case-specific issues, the platform clusters the reported data based on the allocation call stack to form issues, enabling developers to categorize and track problems efficiently.



Issue Details:



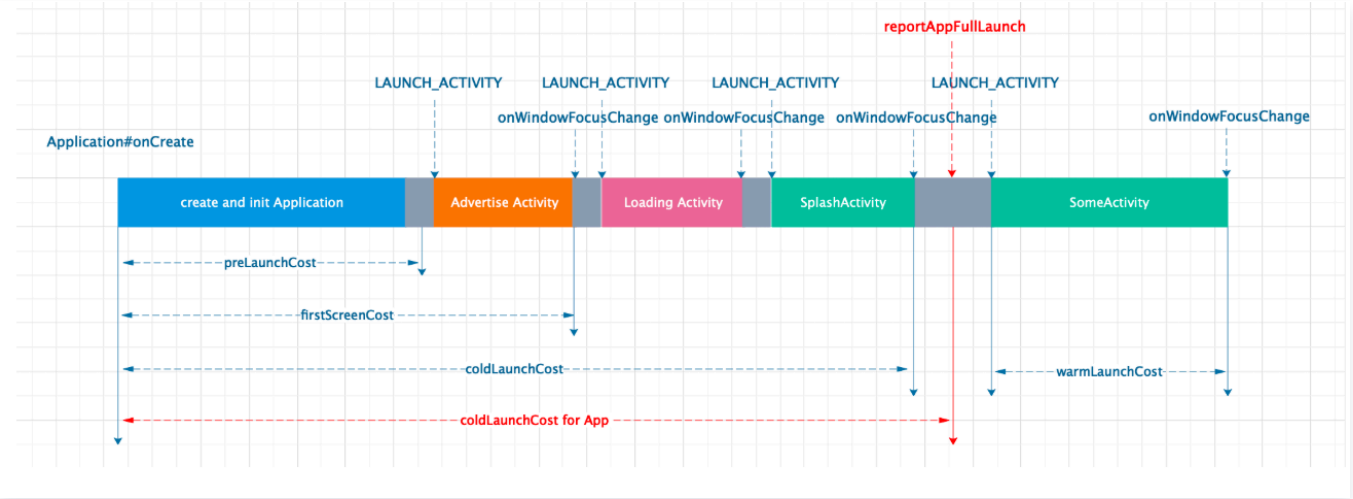
Launch

Last updated：2026-05-25 18:53:26

This article introduces the details of various metrics in the monitoring module and how to view these metrics in the console.

Android

The time consumption per stage of Android launch monitoring is illustrated in the figure below:



- **Default cold launch**

  The time from process creation to the completion of the first frame rendering of the initial Activity. It is calculated as follows:

  default cold launch time = completion time of the first frame rendering of the initial Activity – process creation time

- **Cold Launch**

  The time consumption from process creation to the completion of the first frame rendering of the landing page, or from process creation to the business-customized launch completion point. It is calculated as follows:

  cold launch time = completion time of the first frame rendering of the landing page or business-customized launch completion point – process creation time

  The following are scenarios for Cold Launch:

  - "first_launch": A special case of [Cold Launch], which refers to the first [Cold Launch] after the application is installed, distinguished via the tag "tag_first_launch".
  - "pre_launch": A special case of [Cold Launch], where the process is triggered by non-launch Activity events. This phase is called pre-launch, and after pre-launch, it directly enters the warm launch phase, distinguished via the tag "tag_pre_launch".

- "cold_launch": Indicates a cold launch, from the creation of the application process to the completion of the first rendering of the launch page. Apart from the two special cold launch mentioned earlier, other ordinary cold launch are distinguished via the tag "tag_normal_launch".

- **Warm Launch**

  The time consumption from the opening of an Activity to the completion of its first frame rendering when no Active Activity exists. It is calculated as follows:
  Warm launch time = completion time of the first frame rendering of the initial Activity – onCreate time of the initial Activity
  "warm_launch": Indicates the configuration item for Warm Launch.

iOS

- **Default cold launch**

  The time taken from process creation to the first frame UI of the initial VC being displayed on screen. It is calculated as follows:
  default cold launch time = first frame UI display time of the initial VC – App process creation time
- **Cold Launch**

  The time taken from process creation to the first frame UI display of the initial VC, or from process creation to the business-customized launch completion point. It is calculated as follows:
  Cold launch time = first frame UI display time of the initial VC or business-customized launch completion point – App process creation time
  - App process creation time: Obtain the process launch time from the process's processInfo.
  - main function execution time: business calls related APIs to actively instrument.
  - First frame UI display time: Obtain an approximation via the execution time of the first CA::Transaction::commit.

  The launch report includes three tracking points (Span) by default.
  - _RM_Cold_Launch: From process creation to launch completion, consistent with the cold launch metric scope.
  - _RM_Before_Main: The time from process creation to global variable initialization.
  - _RM_After_Main: The time from the execution of the main function to the launch completion point.

> **Note:**
> Due to the (prewarming) feature introduced in iOS 15, the system preloads certain resources before a user launches the app to improve launch speed. However, this may cause the app launch time data obtained by the platform to appear abnormal, as the prewarming process does not reflect the actual time required for a user-initiated launch. Therefore, to ensure data accuracy, the platform filters out such data. For details about iOS prewarming, refer to About the App Launch sequence.

- **switching between foreground and background**

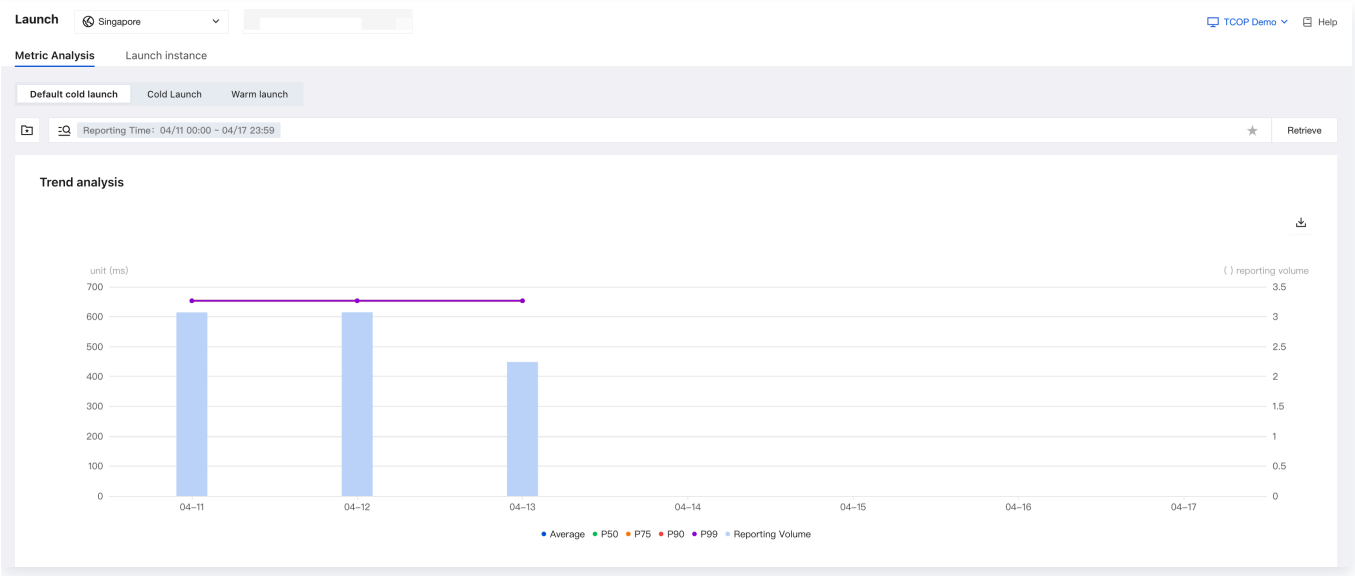
  The application foreground/background switching time refers to the duration from when a user performs an operation (e.g., clicking the icon to bring the app to the foreground or pressing the Home button to send the app to the background) until the application completes the corresponding interface rendering and data processing, reaching an interactive state. The calculation method is as follows:
  Foreground/Background Switching Time = applicationBecomeActive – applicationWillEnterForeground
  - applicationWillEnterForeground: The time when the UIApplicationWillEnterForegroundNotification is received.
  - applicationBecomeActive: The time when the UIApplicationDidBecomeActiveNotification is received. The foreground/background switching report includes one tracking point (Span) by default.
  - _RM_App_Resume: Equivalent to foreground/background switching, that is, the time from applicationWillEnterForeground to applicationBecomeActive.

Data analysis

The following content takes iOS app launch time analysis as an example. For Android, you can refer to the content below.

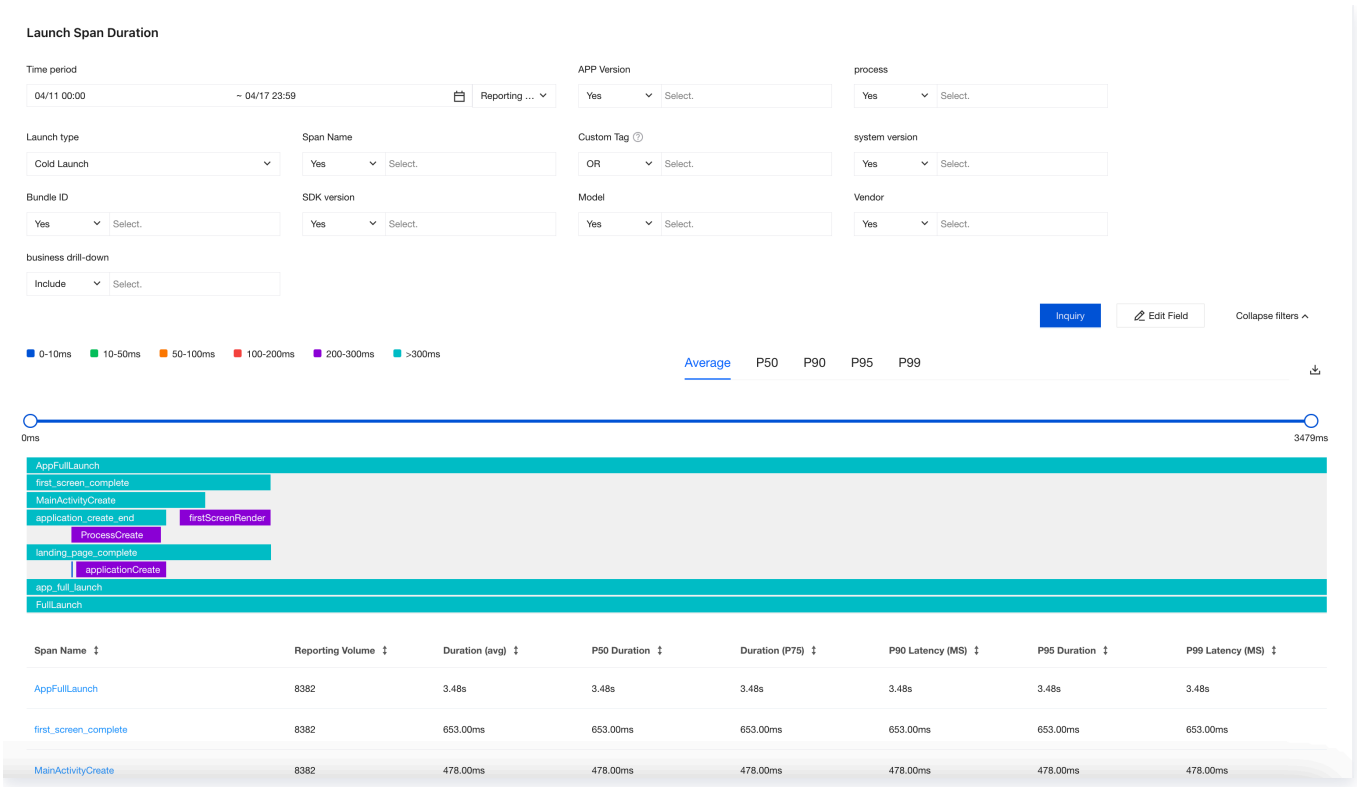
Metric Analysis

The platform supports viewing and drill-down analysis of metrics such as average, P50, P90, and P99. You can view them on the Launch > Metric Analysis page.



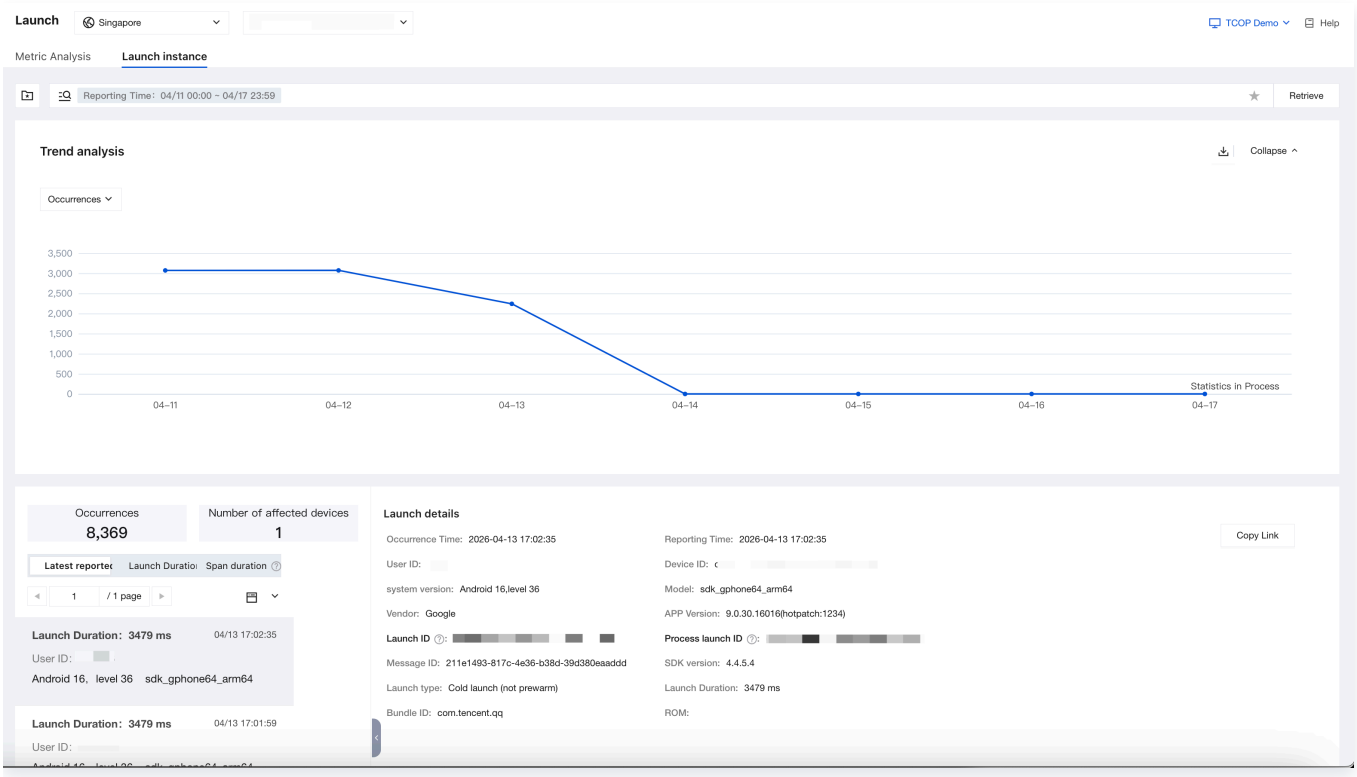
Span Duration Analysis

Users can add some Span information via the tracking API in the SDK during the launch phase for monitoring and troubleshooting the launch duration logic.



Query Launch instance

The launch instance dashboard supports searching for specific launch instances through defined filter criteria for detailed time analysis. You can view it on the Launch > Launch instances page.



Network

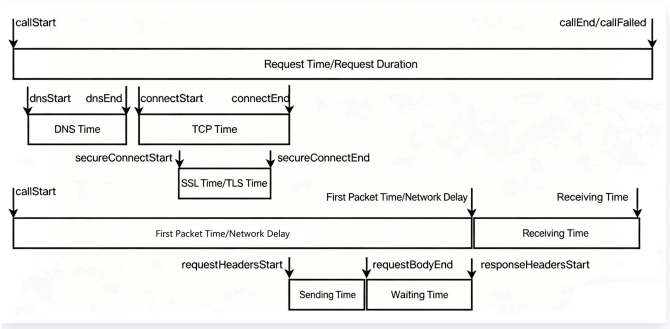
Last updated：2026-05-25 18:53:26

This article provides an overview of the network monitoring module for Android and iOS platforms, its configuration, and data analysis.

Android

Function Introduction

Network monitoring utilizes OkHttp3's EventListener to monitor the quality of HTTP requests, including request duration, success rate, data transfer volume, and time consumption at key stages during the request process.

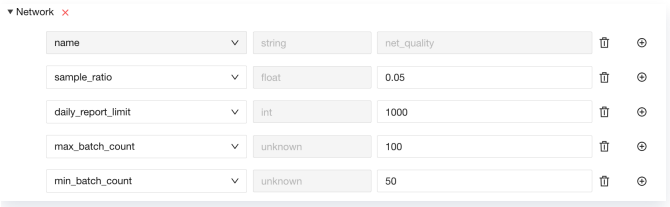


Currently, the following capabilities can be achieved through an instrumentation-free solution:

- You can follow the SDK integration guide and use the OkHttpClient created by using BuglyListenerFactory to monitor the quality of HTTP / HTTPS requests.
- The BuglyURLStreamHandlerFactory proxies the system's native HttpURLConnection, enabling monitoring of HTTP / HTTPS requests implemented through native APIs.

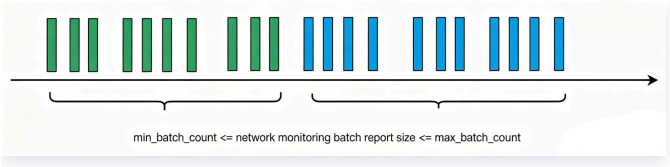
Open configuration

Network monitoring is disabled by default in the local configuration. You need to enable it on the Portal. On the Setting > SDK configuration page, configure by adding the net_quality configuration item and adjusting the sampling rate and other configuration items.



- **sample_ratio**: Device sampling rate, which indicates the proportion of devices allowed to enable network monitoring.

- **daily_report_limit**: Indicates the maximum number of times data reporting for network monitoring is allowed per day. Data for network monitoring is reported in batches, and the specific batching is determined by max_batch_count and min_batch_count.
- **max_batch_count**: Indicates the maximum number of HTTP request quality detailed data entries included in a single network monitoring data report. The default value is 100.
- **min_batch_count**: Indicates the minimum number of detailed HTTP request quality data entries included in a single network monitoring report. The default value is 50.



Note:

- During the testing phase, it is recommended to set sample_ratio to 1. After testing, you can change the device sampling rate based on actual needs.
- During the testing phase, it is recommended to set min_batch_count to 5 or 10 for faster data reporting to the backend. After testing, you can change the value based on your business needs or restore it to the default value.

SDK Integration

Upgrade the SDK

```
implementation "com.tencent.Bugly:Bugly:4.4.2.5"
```

Using APIs

1. Proxies the system's native HttpURLConnection.

   If you do not want to proxy HTTP requests using the system's native interface, you do not need to perform this step. If you want to proxy HTTP requests using the system's native interface, it is recommended to enable the proxy as early as possible in Application#onCreate.

   ```
   // Enables the proxy.
   BuglyURLStreamHandlerFactory.init();
   ```

   After enabling the proxy, you can also disable it using the following code.

   ```
   // Disables the proxy.
   ```

```
BuglyURLStreamHandlerFactory.reset();
```

2. Modify the OkHttpClient creation process.
   - Use BuglyListenerFactory to create an OkHttpClient.

```
OkHttpClient client = new OkHttpClient.Builder()

.eventListenerFactory(BuglyListenerFactory.getInstance())
                .build();
```

   - If executed asynchronously, wrap the Callback using BuglyCallbackProxy.

```
call.enqueue(new BuglyCallbackProxy(callback));
```

   - If your business logic has its own EventListener.Factory, the Factory can be added using the following code.

```
BuglyListenerFactory.getInstance().addFactory(myFactory)
```

   - If you do not need the Factory, you can remove it using the following code.

```
BuglyListenerFactory.getInstance().removeFactory(myFactory)
```

3. Refer to the demo code to construct test data.
   - Build an asynchronous request.

```
OkHttpClient client = new
OkHttpClient.Builder().eventListenerFactory(BuglyListenerFactory.g
etInstance()).build();

Request request = new Request.Builder().url(url).build();

Call call = client.newCall(request);

call.enqueue(new BuglyCallbackProxy(callback));
```

   - Build a synchronous request.

```
OkHttpClient client = new
OkHttpClient.Builder().eventListenerFactory(BuglyListenerFactory.g
etInstance()).build();

Request request = new Request.Builder().url(url).build();

try (Response response = call.execute()) {
    if (response.isSuccessful()) {
        InputStream inputStream = response.body().byteStream();
        ...
    }
} catch (IOException exception) {
   //
}
```

Enable checking

1. Check the configuration pull to verify that network monitoring is enabled.
   - Log Tag: RMonitor_config: dump for onConfigLoad.
   - If the monitoring item allows enabling, logs similar to net_quality:true will appear; if the monitoring item does not allow enabling, logs similar to net_quality:false will appear.

```
2024-08-07 13:08:37.143 4707-4777/com.tencent.demo.Buglyprodemo
I/RMonitor_config: dump for onConfigLoad, {battery_metric:false,
memory_quantile:true, io:false, list_metric:false,
work_thread_lag:false, sub_memory_quantile:false, fd_leak:false,
looper_metric:true, activity_leak:true, battery_element:false,
battery:false, launch_metric:true, battery_ele_metric:false,
looper_stack:true, java_memory_ceiling_hprof:false,
native_memory:true, http:false, traffic_detail:false,
net_quality:true, device:false, db:false, big_bitmap:true,
traffic:false}
```

2. Check whether network monitoring is properly enabled.

```
2024-08-07 13:09:03.768 4939-4939/com.tencent.demo.Buglyprodemo
I/RMonitor_net_quality: getFactory, ret: true, factory: null,
errorMsg: null
```

```
2024-08-07 13:09:03.786 4939-4939/com.tencent.demo.Buglyprodemo
I/RMonitor_net_quality: setURLStreamHandlerFactory success.
2024-08-07 13:09:03.786 4939-4939/com.tencent.demo.Buglyprodemo
I/RMonitor_net_quality: init, true
2024-08-07 13:09:03.807 4939-4997/com.tencent.demo.Buglyprodemo
I/RMonitor_net_quality: start
```

3. Check data reporting.

```
2024-08-07 13:10:56.226 4939-5009/com.tencent.demo.Buglyprodemo
D/RMonitor_report: reportInternal-onSuccess, pluginName: net_quality,
dbId: 7
```

Network Exception

- Result code: Generally used to indicate errors during the HTTP connection establishment phase.

  On the Android platform, there are only two result codes: 0 and –1. 0 indicates the default value, while –1 indicates that an exception occurred during the connection establishment phase. The specific cause of the exception is described in the exception information.
- HTTP status code: In the HTTP protocol, it is a three–digit code used to indicate the server's processing result of a request. Each status code has a specific meaning, used to indicate different situations such as request success, redirection, client error, or server error. The following are some common HTTP status codes and their meanings:
  - 200 OK: The request is successful, and the server has successfully processed the request.
  - 301 Moved Permanently: Permanent redirect. The requested resource has been permanently moved to a new URL.
  - 400 Bad Request: Client error. The server cannot understand the syntax or parameters of the request.
  - 404 Not Found: Resource not found. The server is unable to find the requested resource.
  - 500 Internal Server Error: Server error. The server encountered an unexpected error while processing the request.
- Exception information: Describes the specific cause of the exception, such as domain name resolution exception, Unable to resolve host "www.xxx.com": No address associated with hostname.
- Exception type: Describes the classification of exceptions, which refers to categorizing exception information. For example, if the exception message is Unable to resolve host "www.xxx.com": No address associated with hostname, the extracted exception type would be Unable to resolve host.

iOS

Function Introduction

Network monitoring is used to monitor the network usage of an App. By monitoring the universal network request component NSURLSession, it tracks data such as HTTP request latency, success rate, data transfer volume, and time consumption during critical phases of the request process.

Open configuration

- Similar to traffic monitoring, network monitoring will only take effect when the corresponding SDK configuration is enabled alongside the sub-module activation. For SDK configuration details, refer to SDK Configuration.
- For the configuration details of network monitoring, which are consistent with the Android version, you may refer to Android Network Monitoring Configuration for details.
- Unlike Android, the iOS side does not implement complex reporting frequency control (due to differences in implementation between the two platforms). Therefore, fields such as max_batch_count and min_batch_count do not need to be configured.

> **Note:**
> Although network monitoring and traffic monitoring belong to the same module, their feature configurations are independent and non-exclusive. Therefore, they can be enabled or disabled separately. Additionally, due to data volume considerations, it is not recommended to set the sampling rate of network monitoring too high.

SDK Integration

Network monitoring is a sub-module of the SDK (sharing the network monitoring module with traffic monitoring). It does not require separate integration by the business; simply upgrading to a supported SDK version will include the corresponding module. Therefore, when initializing the SDK, ensure that the enabled modules include Bugly_MODULE_NETWORK or use the RM_MODULE_ALL type. For details, refer to iOS SDK Integration Guide.

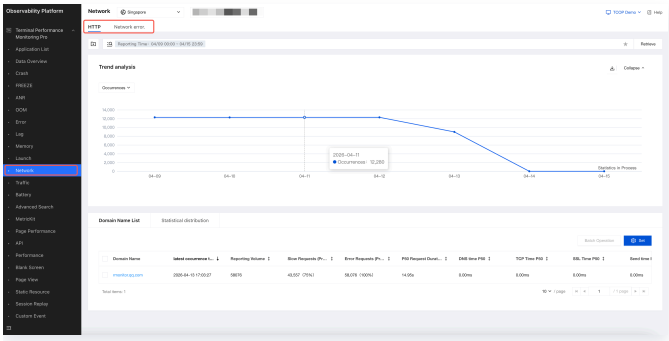
> **Note:**
> - Unlike traffic monitoring, Network Quality Monitoring requires an understanding of network protocols. Therefore, the SDK currently only supports HTTP requests made using NSURLSession. Requests initiated via NSURLConnection, CFNetwork, or other Socket-based connections will not be covered by network monitoring.
> - Network monitoring requires version 2.8.1.5 or later.

Data analysis

The following content uses the Android platform's network monitoring module as an example for explanation. For iOS, refer to the content below.

Query

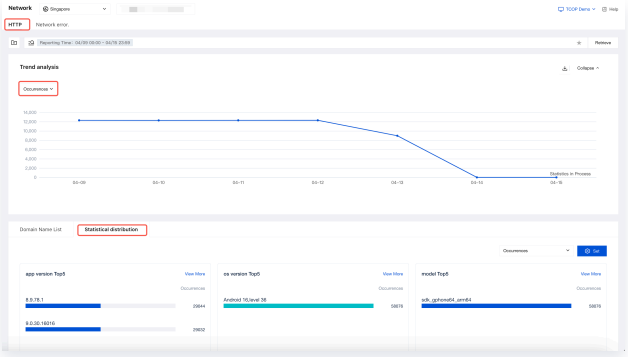
The feature supports users in querying network issues via multiple filter options. For details, see Query.



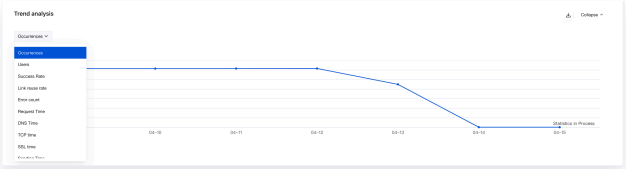
HTTP

HTTP module contains performance data of all monitored HTTP requests, such as success rate, request duration, time consumption at key stages, and data transfer size.

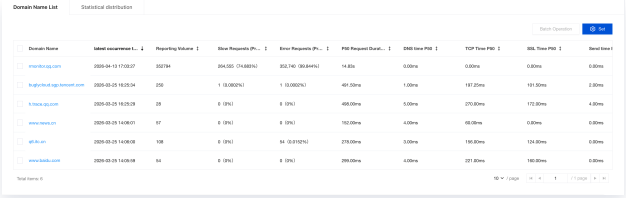
- HTTP module: Displays data from the last 7 days by default, including Top domains and APIs sorted by reporting volume, overall trends, statistical distribution data, and URL-aggregated lists.



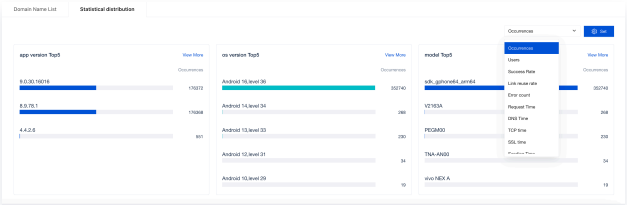
- Trend analysis supports multiple metrics such as Occurrences, Users, Success Rate, and Link reuse rate. Users can select relevant metrics to view analytical charts based on their needs.



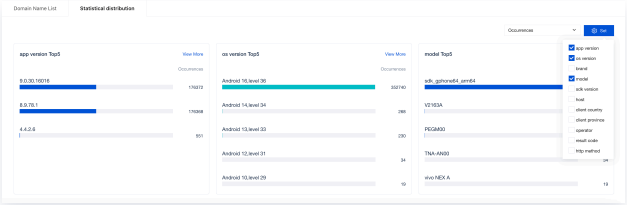
- Domain Name List: Currently aggregated by domain and API.



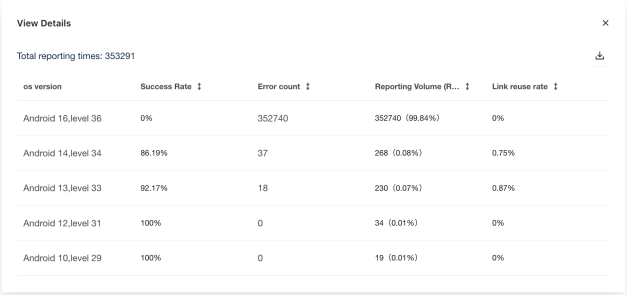
- Statistical distribution supports multiple metrics such as Occurrences, Users, Success Rate, and Link reuse rate. Users can select relevant metrics to view distribution charts based on their needs.



Click **Set** to select multiple drill-down metrics for display.



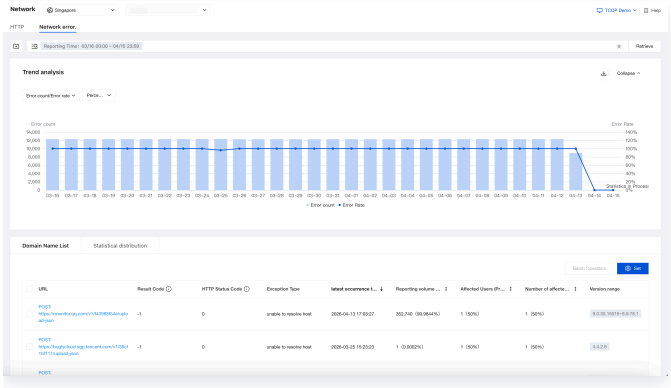
- Click **View More** to view the reporting details of the current drill-down metrics.



- In addition to the overall trend, we may focus on data from specific domains and APIs. This can be achieved using multi-dimensional drill-down. Click the domain or API you wish to analyze to view the data of the specified domain/API on the right side.

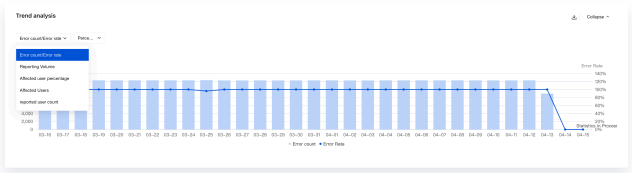
Network error

Network errors include data on all HTTP request errors, and the success rate of HTTP requests is measured by the error rate metric.

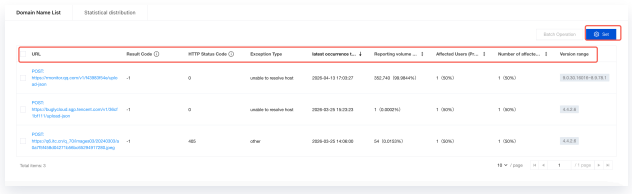


- Trend analysis supports multiple metrics such as Error count/Error rate, Reporting Volume, Affected user percentage, Affected Users, and reported user count. Users can choose to display these metrics on

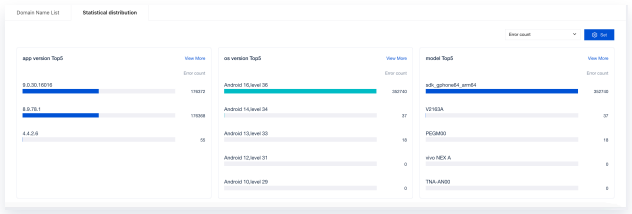
demand.



- Domain list: The domain list is currently clustered by `HTTP method + domain + API + error code`. The list supports displaying multiple dimensions such as URL, Result Code, HTTP Status Code. You can click **Set** to configure the fields.



- Statistical distribution: Similar to HTTP, network errors also provide rich drill-down fields, enabling users to analyze the reporting proportion across multiple dimensions.



Traffic

Last updated：2026-05-25 18:53:27

This article introduces the overview, configuration, and usage of features in the traffic monitoring module for Android and iOS platforms.

Android

Feature Background

The core feature of traffic monitoring is its ability to track application data usage in real time. Through real-time monitoring, you can stay informed about data consumption patterns and promptly identify issues such as unexpected traffic usage. This helps you avoid exceeding data plans or incurring additional charges. Additionally, traffic monitoring assists in optimizing application power consumption and reducing unnecessary data requests to deliver a more efficient user experience and extend the battery life. This feature also enables drilling down to the domain level, allowing you to identify which domains or services consume a large amount of traffic, facilitating targeted measures to optimize data transmission.

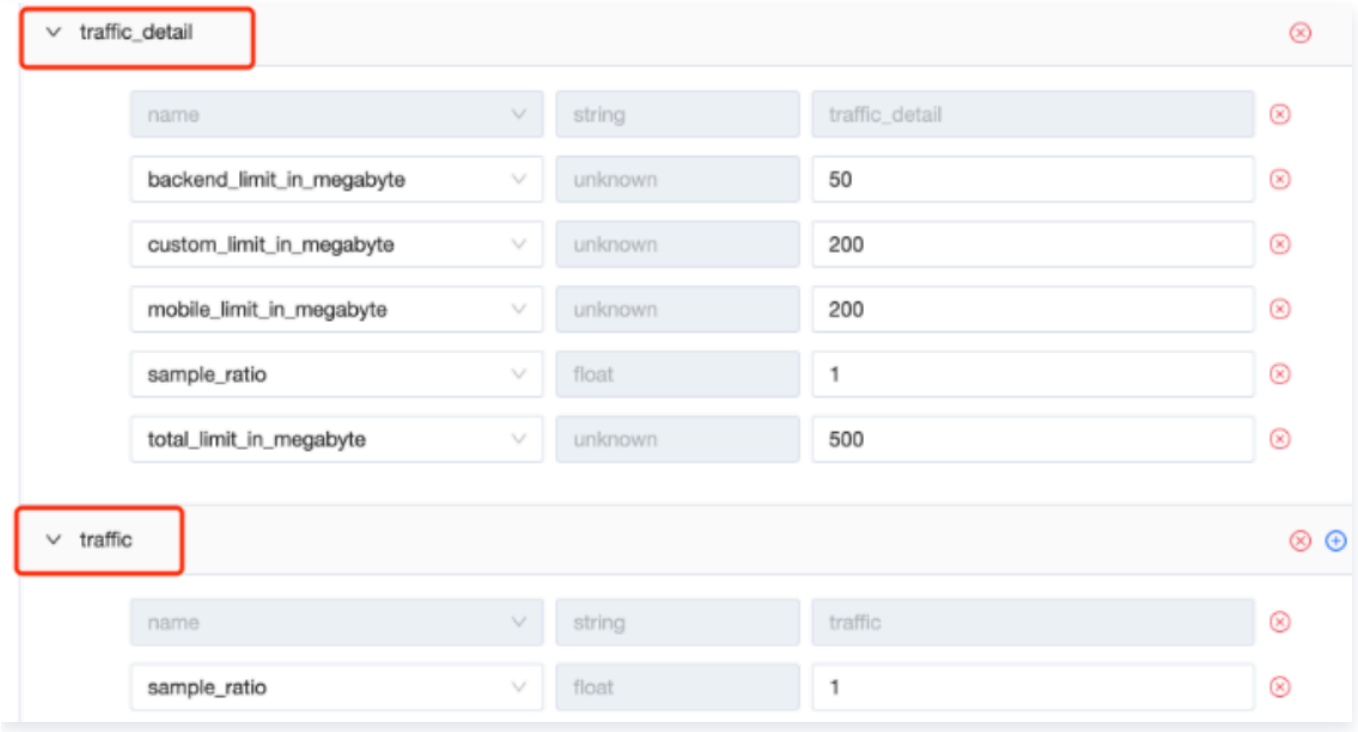
Open configuration

In Setting > SDK configuration, enable traffic monitoring.

- traffic: refers to the per-process traffic, for which the device sampling rate can be configured.
- traffic_detail: refers to the 10-minute traffic, for which the device sampling rate and the threshold for abnormal traffic can be configured.
- The unit for these traffic thresholds is MB.
- During the integration phase, it is recommended to set both device sampling rates (sample_ratio) to 1.0, i.e., full sampling.

Parameter	Configuration field name	Description
Total traffic is excessive.	total_limit_in_megabyte	The traffic consumed within 10 minutes exceeds the configured threshold for abnormal traffic.
Background traffic is excessive.	backend_limit_in_megabyte	The traffic consumed in the background scenario within 10 minutes exceeds the configured threshold for background abnormal traffic.
Mobile traffic is excessive.	mobile_limit_in_megabyte	The mobile traffic consumed within 10 minutes exceeds the configured threshold for abnormal mobile traffic.

Custom scenario traffic is excessive.	custom_limit_in_megabyte	The custom scenario defined by the business checks whether the traffic consumption exceeds the defined threshold.



SDK Integration

The integration approach for traffic monitoring is not significantly different from other features such as lag, Java memory leaks, etc. The monitoring capabilities of the SDK are generally jointly controlled by the client and backend configuration. That is, the client must execute the following statement during initialization while enabling the traffic monitoring feature in the backend configuration to activate it.

```
RumProBuilder.addMonitor(RumProMonitorName.TRAFFIC);
RumProBuilder.addMonitor(RumProMonitorName.TRAFFIC_DETAIL);

RumPro.init(application, RumProBuilder);
```

Enable success logs

```
12-20 16:13:27.470 21408 21449 I TrafficMonitor: traffic monitor start
```

Report logs

```
1. Traffic report logs every 10 minutes (reported every 10 minutes).
```

```
04-22 01:04:26.665 17704 17758 D RMonitor_report_Json: url:
********** eventName: traffic_detail, client_identify:
****6415c412a9ed43cd0f43d601****

2. Process traffic metrics report logs (reported 5 minutes after process
restart).
04-22 01:04:26.665 17704 17758 D RMonitor_report_Json: url:
********** eventName: traffic, client_identify:
*****415c412a9ed43cd0f43*****
```

Custom Scenario Traffic Reporting API

```
CustomTrafficStatistic.getInstance().addHttpToQueue(SocketInfo
socketInfo);
```

SocketInfo contains many other fields. To collect traffic statistics, you need to assign values only to the following fields. Other member variables can be ignored.

```
class SocketInfo {
    receivedBytes;   // Bytes received.
    sendBytes;       // Bytes sent.
    networkType;     // Network type. Set to 1 for Wi-Fi traffic, 2 for
5G traffic, or 3 for no network.
    frontState;      // Foreground and background state. Set to 1 for
foreground, or 2 for background.
    host;            // Domain name, such as www.baidu.com.
    startTimeStamp;  // Timestamp when the network request starts, in
milliseconds.
    ......
}
```

iOS

Feature Background

The core feature of traffic monitoring is its ability to track application data usage in real time. Through real-time monitoring, you can stay informed about data consumption patterns and promptly identify issues such as unexpected traffic usage. This helps you avoid exceeding data plans or incurring additional charges. Additionally, traffic monitoring assists in optimizing application power consumption and reducing

unnecessary data requests to deliver a more efficient user experience and extend the battery life. This feature also enables drilling down to the domain level, allowing you to identify which domains or services consume a large amount of traffic, facilitating targeted measures to optimize data transmission.

Open configuration

- To enable traffic monitoring, you must also ensure that the traffic monitoring module is enabled in the SDK configuration. For details about the SDK configuration, refer to SDK Configuration.
- For the traffic monitoring configuration, which is consistent with the Android version, refer to Android Traffic Monitoring Configuration for reference.

SDK Integration

Traffic monitoring is a submodule of the SDK and does not require separate integration by the business. Simply upgrade to a supported SDK version to include the corresponding module. Therefore, when initializing the SDK, ensure that the enabled modules include RumPro_MODULE_NETWORK or use the RM_MODULE_ALL type. For details, refer to SDK Initialization.

> **Note:**
> Traffic monitoring requires version 2.7.53 or later.

Traffic Monitoring Enablement Verification

After you have successfully started the traffic monitoring feature, the following log indicates that the traffic monitoring feature is enabled:

```
[RumPro][Event][RumProNetworkMonitor.mm:281]RumPro_MODUEL_NETWORK start
network monitor: 1048576, 1048576, 1048576
```

Traffic monitoring reports process traffic or 10-minute traffic when the process starts and every 10 minutes. When reporting is triggered, the following log appears:

```
[RumPro][Event][RMReportQueue.m:601][Report] [resource.traffic] report
id:xxxx error:(null)

// or

[RumPro][Event][RMReportQueue.m:601][Report] [resource.traffic_detail]
report id:xxxx error:(null)
```

Custom API

- By default, the SDK monitors all traffic within the App, including HTTP network requests made via NSURLSession and TCP network requests made via BSD Socket.
- However, due to the limitations of BSD Socket, the monitoring information can only provide IP and port details. To offer more flexibility in network traffic monitoring, RUM Pro provides a custom network monitoring API.
- In RumProNetworkTracer+Public.h and RumPro_network_tracer.hpp, interfaces for custom network traffic monitoring are defined for invocation at the Objective-C layer or C++ layer, respectively, to implement custom network traffic monitoring.
- Calling the traceConnectComplete API of the tracer object automatically records the traffic monitoring information captured by this tracer to the SDK and reports it.

> **Note:**
> Due to identified issues with network monitoring on iOS 12.0 and earlier systems, starting from version 2.7.55.1, network monitoring no longer supports devices running iOS 12.0 or earlier, but only supports iOS 13.0 and later systems.

For example:

```
NSString *peerName = @"..."; // Connection URL and other information
RumProNetworkTracer *tracer = [RumProNetworkTracer
tracerWithPeerName:peerName

type:RumProNetworkConnectHTTP|RumProNetworkConnectCustom];
// This method injects the tracer object into the _self object. In
asynchronous processing scenarios, injecting it into the context
object facilitates later retrieval.
[tracer injectToObj:_self];
// The connection starts.
[tracer traceConnectStart];
// Record connection sending data
[tracer traceConnectSend:request.HTTPBody.length];

// ... other logic

// This method retrieves the tracer object from the registered objects
via the context object.
RumProNetworkTracer *tracer = [RumProNetworkTracer
tracerFromObj:_self];
// The tracer completes and records itself to the SDK.
```

```
[tracer traceConnectComplete];
```

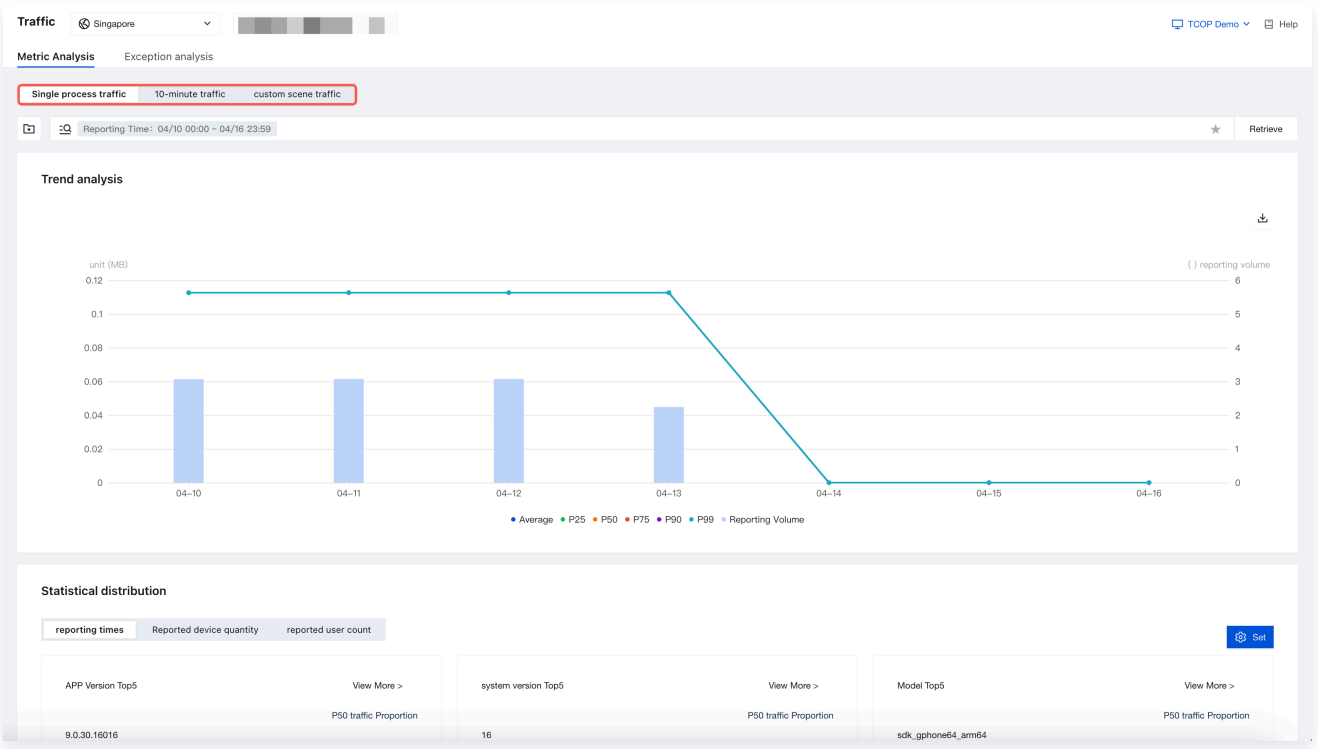
- The usage of other APIs is similar. For details, refer to the definitions in the file.

Usage of Feature

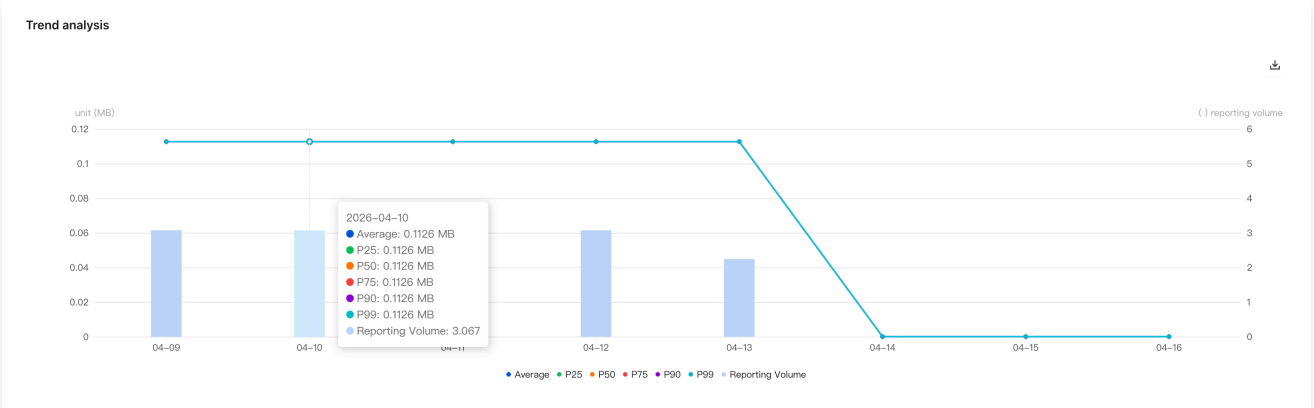
The following content uses the Android platform's traffic monitoring module as an example for explanation. For iOS, refer to the content below.

Metric Analysis

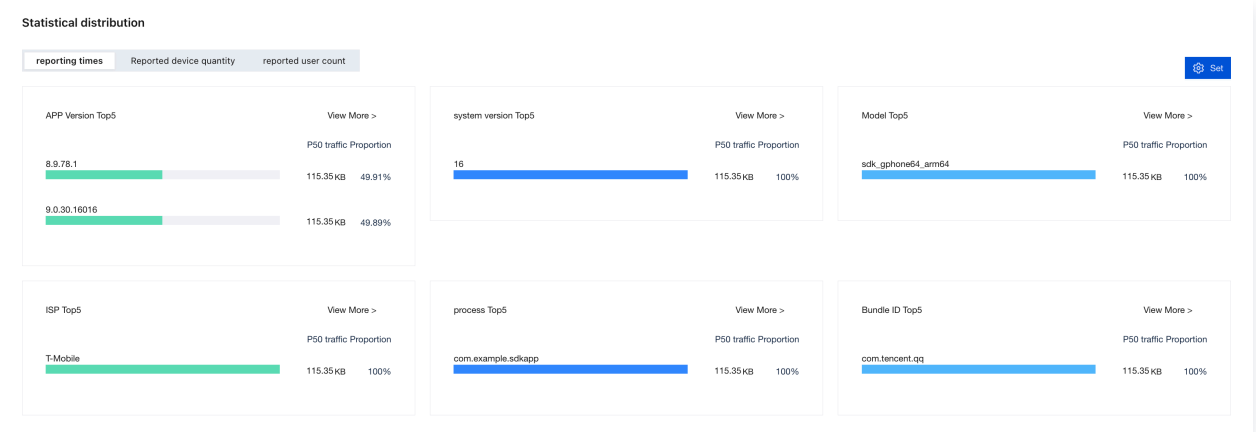
- The metrics analysis includes three Tabs: Per-process traffic, 10-minute traffic, and Custom scenario traffic.
  - **Single process traffic**: Refers to all traffic consumed by a process during a single execution.
  - **10-minute traffic**: Describes all traffic consumed by sampled devices during a 10-minute monitoring period.
  - **custom scene traffic**: Describes either per-process traffic or 10-minute traffic within business-critical scenarios.



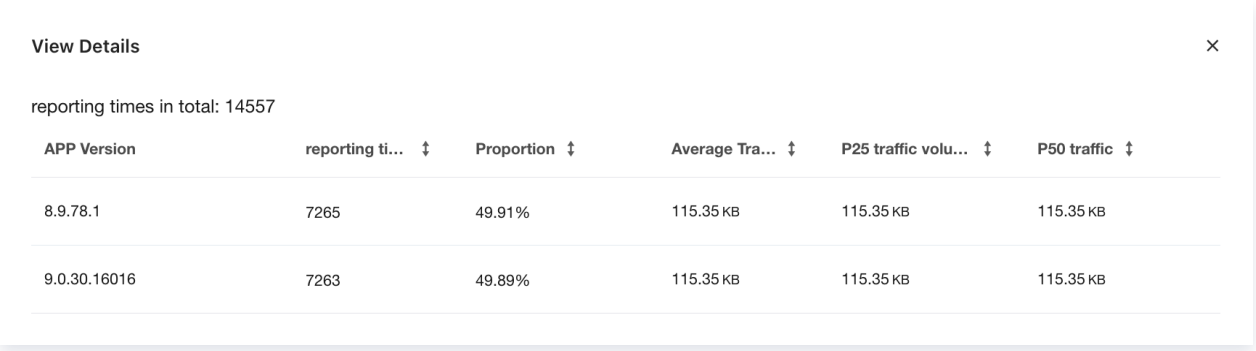
- The metrics of all three Tabs include trend analysis, statistical distribution, and traffic attribution.
  - Whether it is per-process traffic or 10-minute traffic, each report counts as one sample.
  - Reporting Volume represents the sample size. As shown below, the Reporting Volume is 106,857, indicating a total of 106,857 reports. The average and percentile values such as P25 are calculated based on these reported samples.



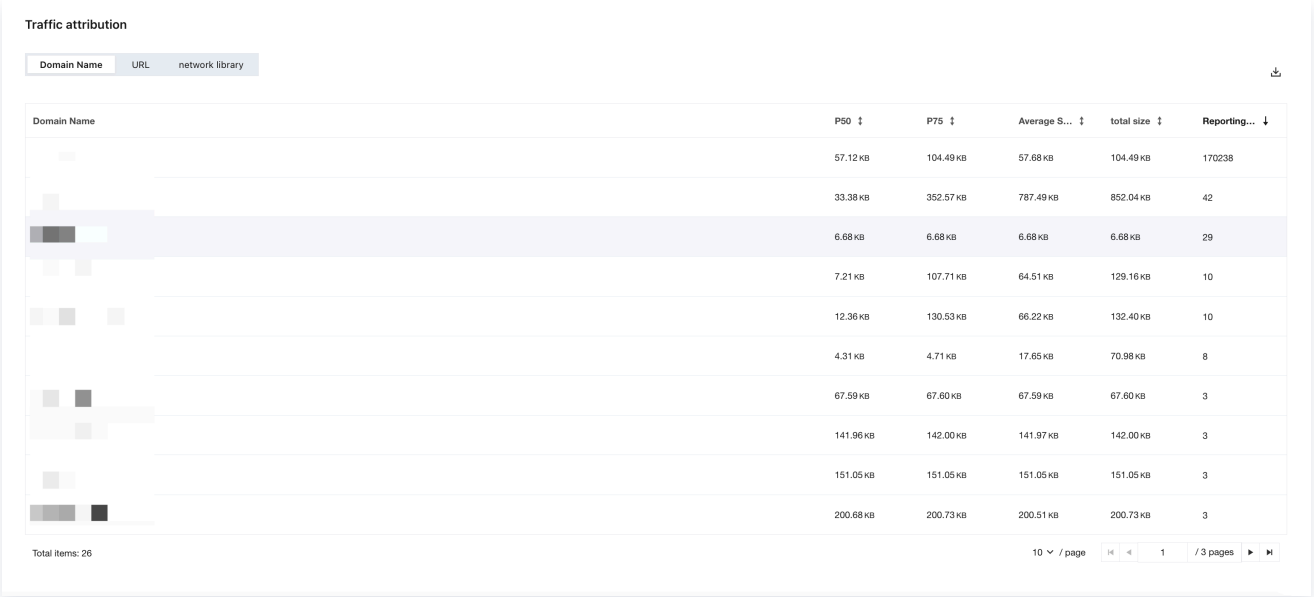
- Statistical distribution shows the distribution of these samples across specified dimensions.
  - The distribution by App version shows that version 8.9.78.1 has the highest number of reported samples, accounting for 49.91%.



  - Click **View More** to view the reporting details.



- Traffic Attribution currently supports domains, URLs, and network libraries.



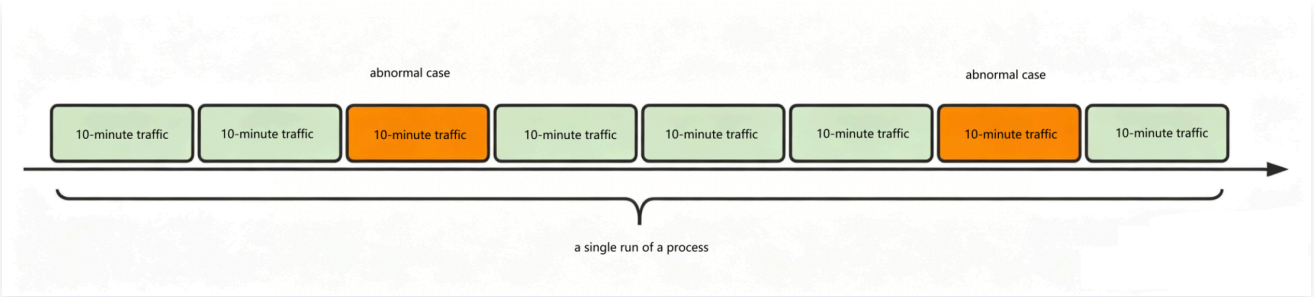
Exception analysis

- **During the statistical period, if traffic consumption exceeds the configured threshold, it is determined that an abnormal situation will occur.**

  After an exception is detected, detailed connection information is collected and reported to the backend. Users can configure their own exception detection thresholds.

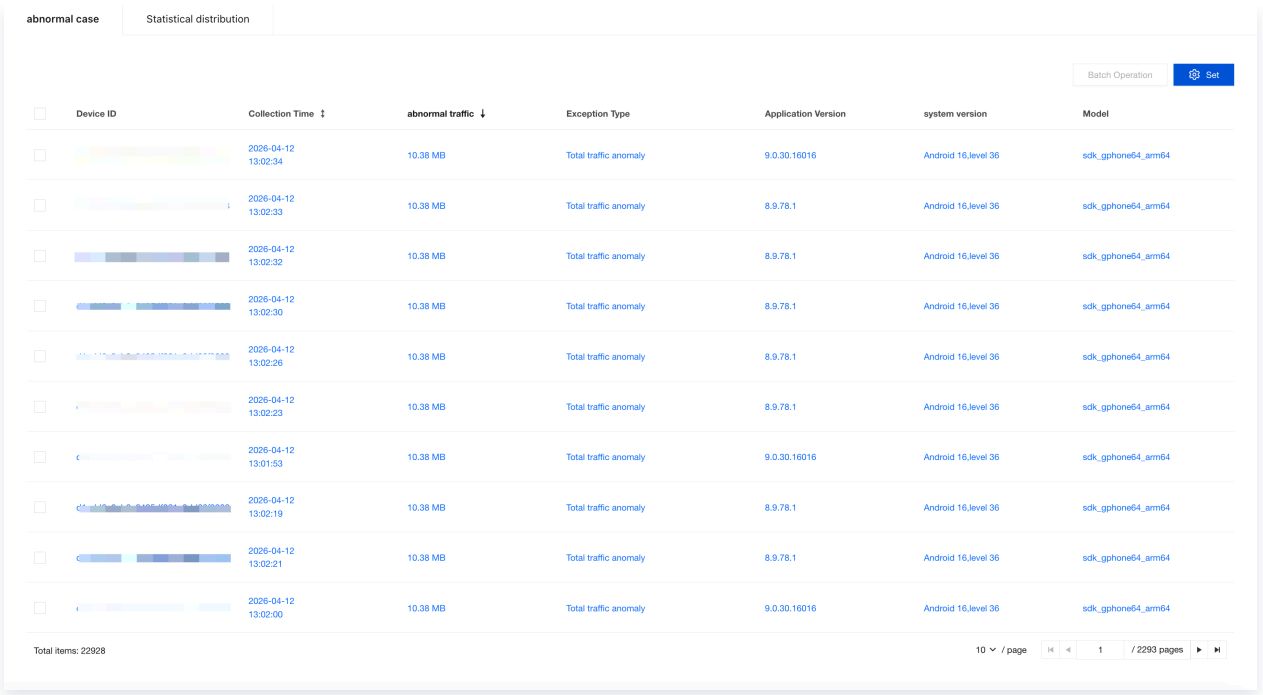
Parameter	Configuration field name	Description
Total traffic is excessive.	total_limit_in_megabyte	The traffic consumed within 10 minutes exceeds the configured threshold for abnormal traffic.
Background traffic is excessive.	backend_limit_in_megabyte	The traffic consumed in the background scenario within 10 minutes exceeds the configured threshold for background abnormal traffic.
Mobile traffic is excessive.	mobile_limit_in_megabyte	The mobile traffic consumed within 10 minutes exceeds the configured threshold for abnormal mobile traffic.
Custom scenario traffic is excessive.	custom_limit_in_megabyte	The custom scenario defined by the business checks whether the traffic consumption exceeds the defined threshold.

- **The Abnormal Analysis page displays abnormal metrics and trends.**
  - In abnormal metrics, the numerator represents the reported abnormal instances, while the denominator is the reported sample size of the 10-minute traffic.
  - The currently defined abnormalities are all based on the results of 10-minute traffic statistics.

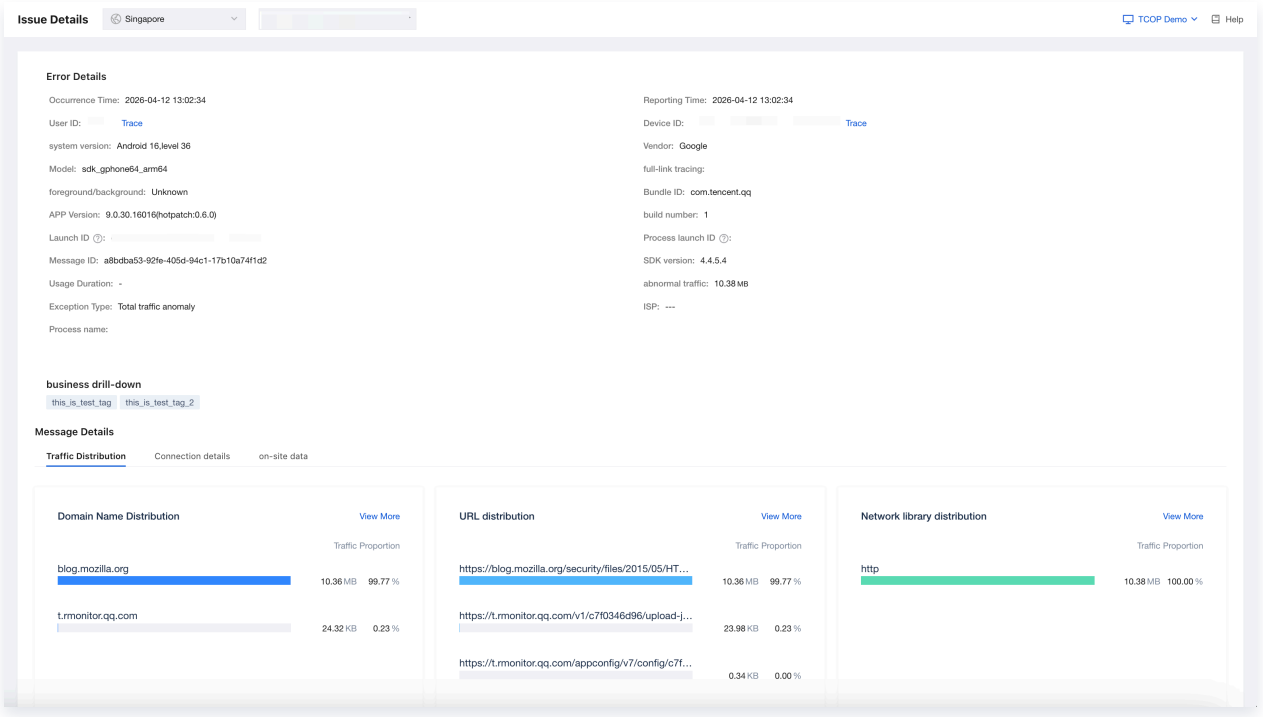


Metric	Description
Exceptions	Reported volume of abnormal traffic under filter conditions.
Exception rate	Reported volume of abnormal traffic under filter conditions / Total reported sampling volume under filter conditions (10min, only supports time range + App version).
Affected Users	Affected users of abnormal traffic under filter conditions.
User impact rate	Affected users of abnormal traffic under filter conditions / Total reported users under filter conditions (10min).
Number of affected devices	Affected devices of abnormal traffic under filter conditions.
Device Impact Rate	Affected devices of abnormal traffic under filter conditions / Total reported devices under filter conditions (10min).

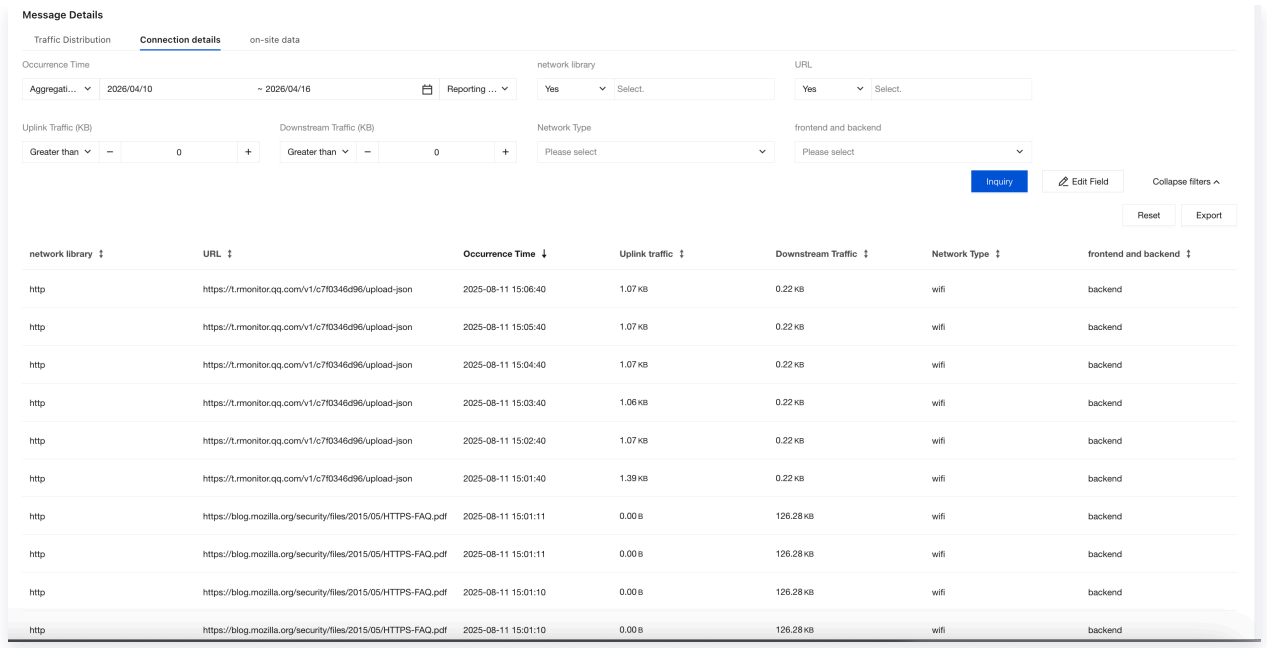
- **After selecting filter conditions, users can view exception cases that meet the conditions.**
  - Users can sort exception cases by dimensions such as abnormal traffic or collection time to locate desired cases for analysis.



- Click an exception case to go to its details and view the connection details.



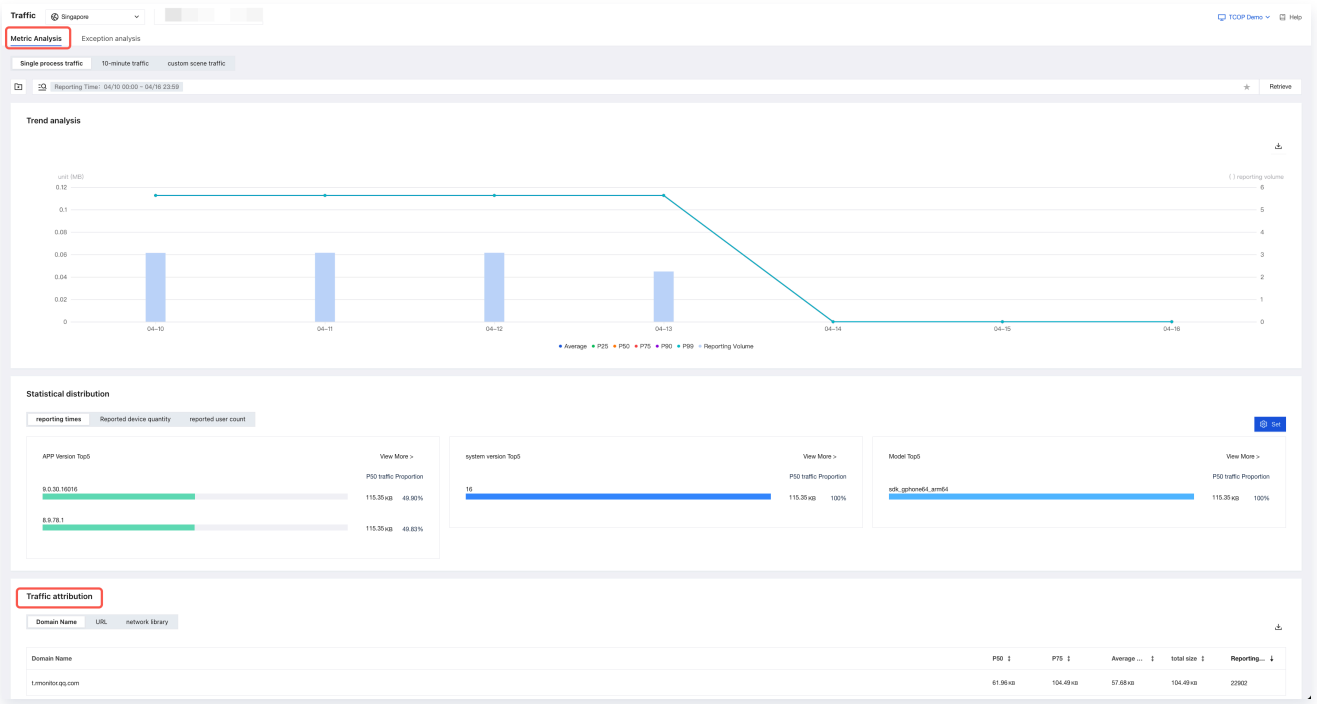
- The connection details in the case details display the traffic consumption details of each monitored connection within this statistical period.



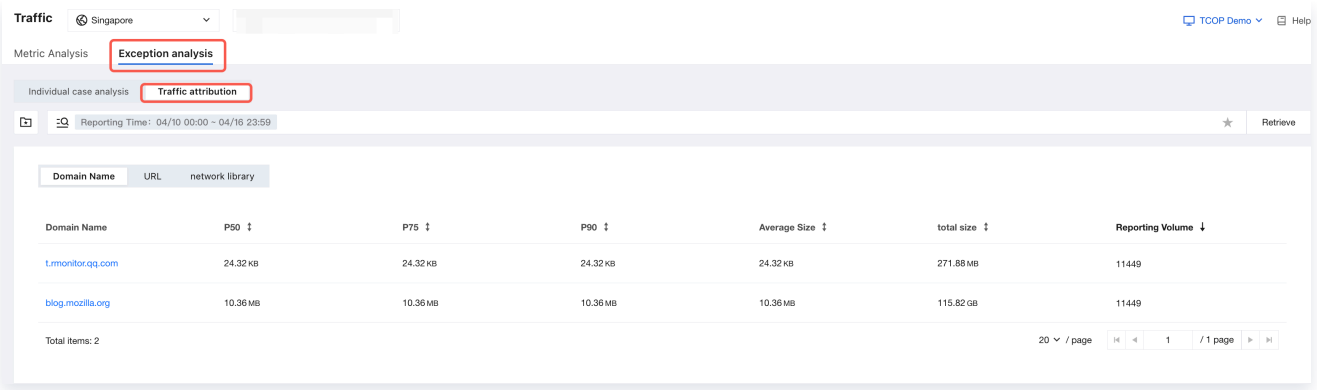
Traffic attribution

Whether it's metric analysis or abnormal analysis, both include traffic attribution.

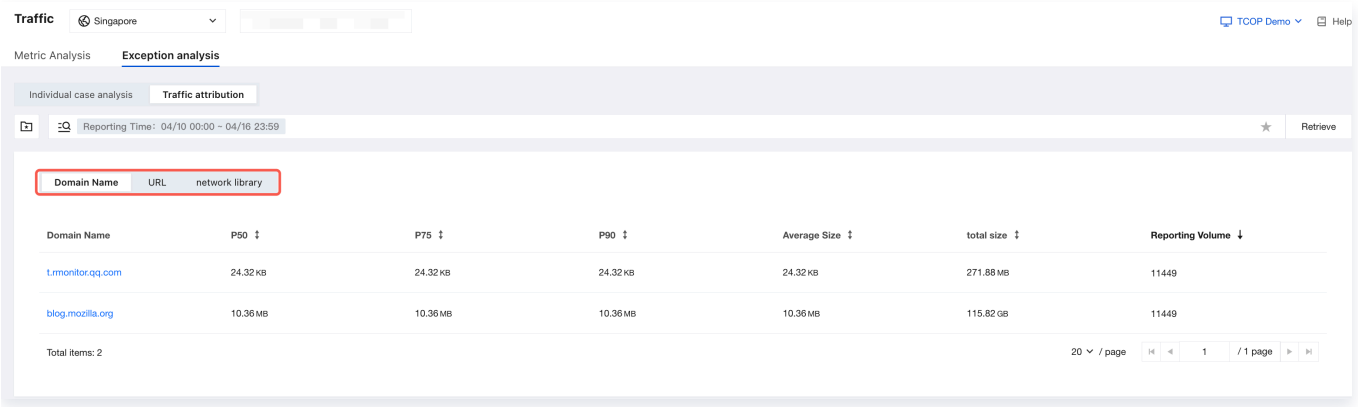
- Metric Analysis:



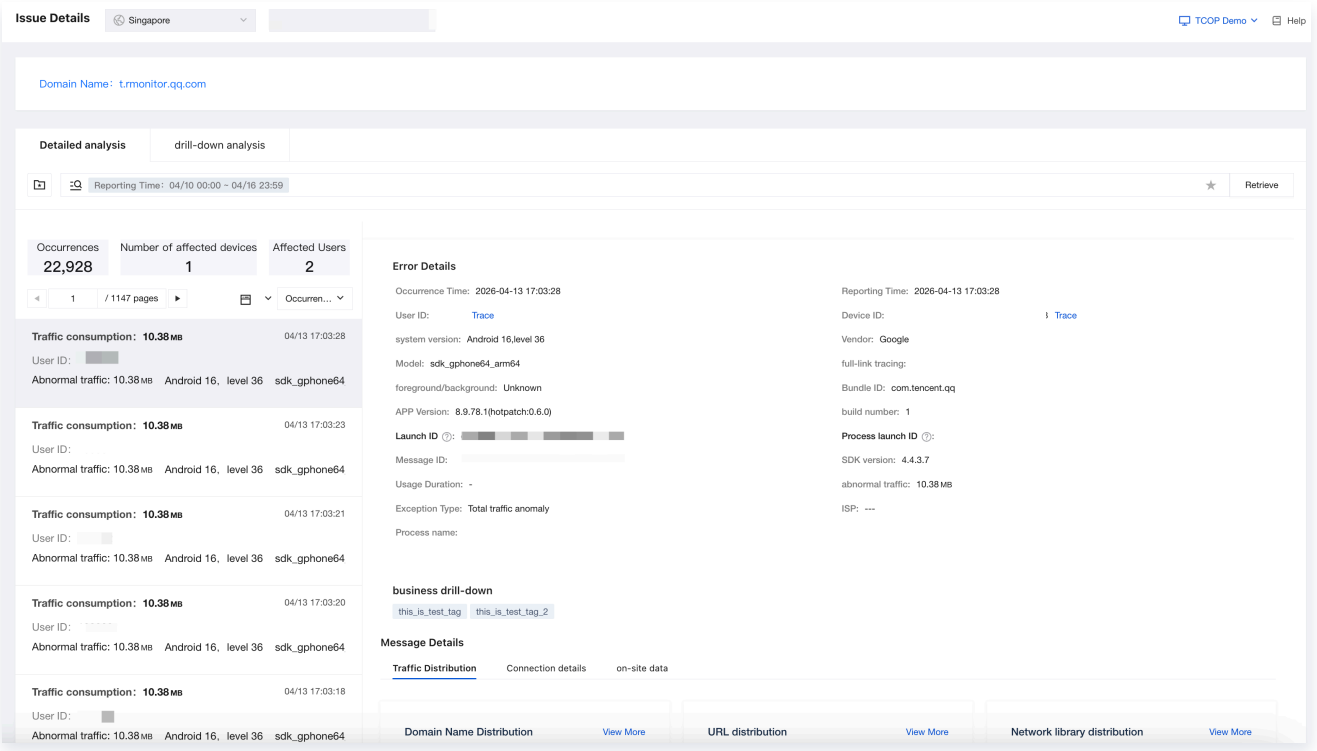
- Exception Analysis:



Traffic attribution aims to perform attribution analysis on consumed traffic. Currently, it only supports attribution based on domains, URL, and network libraries.



- Currently, for URL attribution, except for custom scenarios, it only supports up to the domain level.
- In the traffic attribution of exception cases, you can also click on an attribution to view all exception cases that meet the attribution criteria.



Page Performance

Last updated：2026-05-25 18:53:27

This article introduces the overview, configuration, and usage of features in the page performance monitoring module for Android and iOS platforms.

Android

Background

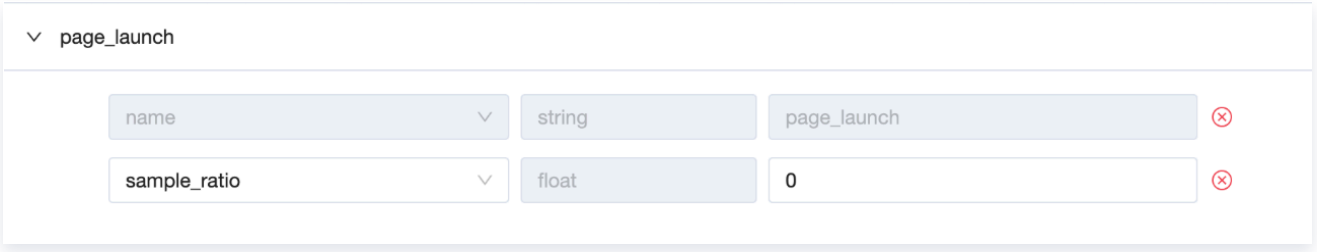
User experience (User Experience, UX) is a critical factor in mobile application development. With the advancement in smartphone hardware performance and the continuous increase in user demands, users have higher expectations for application responsiveness and smoothness. Page startup time is a key metric for user experience, especially in mobile applications. An extended page startup time can frustrate users and may even result in user churn. By default, the platform's page startup monitoring tracks cold start time for Activities, including two types of metrics: **Page Rendering Duration** and **Page loading duration**.

- **Page Rendering Duration:** The time from the call of Activity's onCreate to the completion of the first frame rendering.
- **Page loading duration:** Its value is the same as the page rendering time by default, but users can customize the loading end time of the Activity via the reportActivityFullLaunch API.

Enabling Steps

1. On the Setting > SDK configuration Edit Configuration page, add the following configuration items.

   sample_ratio: Controls the user sampling rate, indicating how many devices will enable this feature. 1 indicates that all devices are enabled, while 0 indicates that all devices are disabled.



2. Before calling the Bugly.init API, add the following code in the client. For details about how to call the Bugly.init API, see SDK Initialization.

```
...
BuglyBuilder.addMonitor(BuglyMonitorName.PAGE_LAUNCH); // Add this
statement
...
```

```
Bugly.init(context, builder);
```

API Description

In addition to monitoring the overall time consumption on the page, users can also call the following APIs to monitor the time consumption of each Span during the page launch process. A Span can be understood as a sub-stage in the page launch process. The following are the APIs of the **com.tencent.rmonitor.pagelaunch.PageLaunchMonitor** class, which is implemented as a singleton. You can retrieve this singleton instance via **PageLaunchMonitor.getInstance()**.

```
/**
  * Mark the beginning of a Span during the activity launch process
  * @param activity
  * @param name, Span name
  * @param parentName, parent Span name
  */
public void startSpan(Activity activity, String name, String
parentName);


/**
 * Mark the end of a Span during the activity launch process
 * @param activity
 * @param name Span name, which must match the parameter of startSpan.
 */
public void endSpan(Activity activity, String name);


/**
 * The user-defined end time of the Activity launch process uses the
time when this API is called as the end time of the Activity launch.
 * @param activity
 */
public void reportActivityFullLaunch(Activity activity);
```

Note that startSpan and endSpan must be called in paired matching; only one record will be saved for the same Span name, meaning the latter Span with the same name will overwrite the former.

Sample Code

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    PageLaunchMonitor.getInstance().startSpan(this, "MemoryTracerTest",
"");
    PageLaunchMonitor.getInstance().startSpan(this, "onCreate",
"MemoryTracerTest");
    startMemoryTrace();
    PageLaunchMonitor.getInstance().endSpan(this, "onCreate");
}

@Override
public void onResume() {
    super.onResume();

    PageLaunchMonitor.getInstance().startSpan(this, "onResume",
"MemoryTracerTest");
    reStartMemoryTrace();
    PageLaunchMonitor.getInstance().endSpan(this, "onResume");
    PageLaunchMonitor.getInstance().endSpan(this, "MemoryTracerTest");

    // Call reportLaunchFinished to mark the end of the Activity launch
only after the page loading is completed.
    new Thread (new Runnable() {
        reportLaunchFinished();
    }).start();
}

private void reportLaunchFinished() {
    PageLaunchMonitor.getInstance().reportActivityFullLaunch(this);
}
```

The above example contains three Spans: "MemoryTracerTest", "onCreate", "onResume", among which MemoryTracerTest is the root Span, and its duration is the sum of the two child Spans, onCreate and onResume.

Log Description

- Log for successful feature enabling:

```
10-04 10:xx:xx.xxx 14546  1819 D PageLaunchPlugin: start page launch
monitor
```

- Log for successful feature disabling:

```
10-04 10:xx:xx.xxx 14546  1819 D PageLaunchPlugin: stop page launch
monitor
```

- Logs are reported after backgrounding, uploading all recorded Activity cold start time data:

```
10-22 10:11:44.465 15996 16178 V RMonitor_report_Json: url:
https://xxx.qq.com/v1/xxxx/upload-json?
timestamp=1720059647340&nonce=xxxxxxxx eventName: page_launch,
client_identify: f55e3c66c53520933d0b8c8687da6398
```

iOS

Background

In iOS applications, page loading time is a crucial performance metric. If a page takes too long to launch, it may degrade user experience and even impact the overall performance of the application. Page performance monitoring primarily focuses on tracking page loading time. By default, the platform's page loading monitoring feature tracks the loading time of UIViewController, which is divided into two categories: **page rendering time** and **page loading time**.

- **Page rendering time**: The duration from the call of UIViewController's loadView: to the call of viewDidAppear:.
- **Page loading time**: Its value is the same as the page rendering time by default, but users can customize the loading end time of VC through the endVCRender API.

Enabling Steps

Page load time analysis is a new feature. Similar to other features on the platform, it requires executing the following code on the client-side to enable it, while also configuring an appropriate sampling rate in the platform backend.

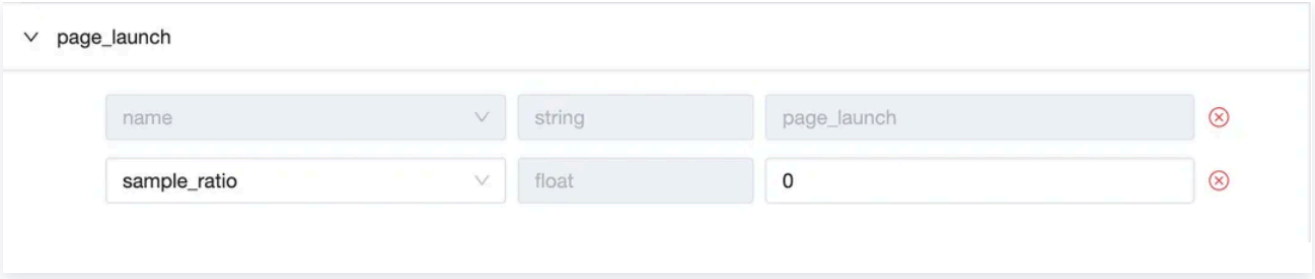
1. Include Bugly_MODULE_PAGE_LAUNCH or use RM_MODULE_ALL in the startup module.

```
[Bugly start:@[Bugly_MODUEL_PAGE_LAUNCH, /*...*/] config:config
completeHandler:^{/*...*/}];
// Or
```

```
[Bugly start:RM_MODULE_ALL config:config completeHandler:^{/*...*/}];
```

2. On the edit configuration page of App Configuration > SDK Configuration, add the following configuration items:

   sample_ratio: Controls the user sampling rate, indicating how many devices will enable this feature. 1 indicates that all devices are enabled, while 0 indicates that all devices are disabled.



API Description

In addition to monitoring the page rendering and loading time, users can also call the following APIs to monitor the time consumption of each Span during the page launch process, which can be understood as sub-stages in the page launch process.
The following is the definition of BuglyPageLoadMonitor.

```
@interface BuglyPageLoadMonitor : NSObject

/**
 * Marks the current ViewController as rendered.
 * If this method is never called, viewDidAppear is used as the end
time.
 * @param vc The corresponding ViewController object
 */
+ (void)endVCRender:(UIViewController *)vc;

/**
 * Add a group of Spans for the current ViewController
 * @param Spans The added Spans
 * @param vc The corresponding ViewController object
 */
+ (void)addSpans:(NSArray<RMSpan *> *)Spans forVC:(UIViewController
*)vc;

@end
```

The definition of RMSpan is consistent with the Span in launch monitoring. For details, refer to Launch. For the same ViewController, only the last record of a same-named Span will be saved, meaning the latter overwrites the former's time consumption data.

Log Description

- After successful enabling, the module enabling log output during SDK startup will contain the corresponding module name:

```
[Bugly][Event][BuglyDAUReporter.m:100][SDK setup] Activate Module:(
    "launch.page_launch"
    ...
)
```

- Similarly, the reporting logs contain the following information:

```
[Bugly][Event][RMReportQueue.m:601][Report] [launch.page_launch]
report id:xxx error:(null)
```

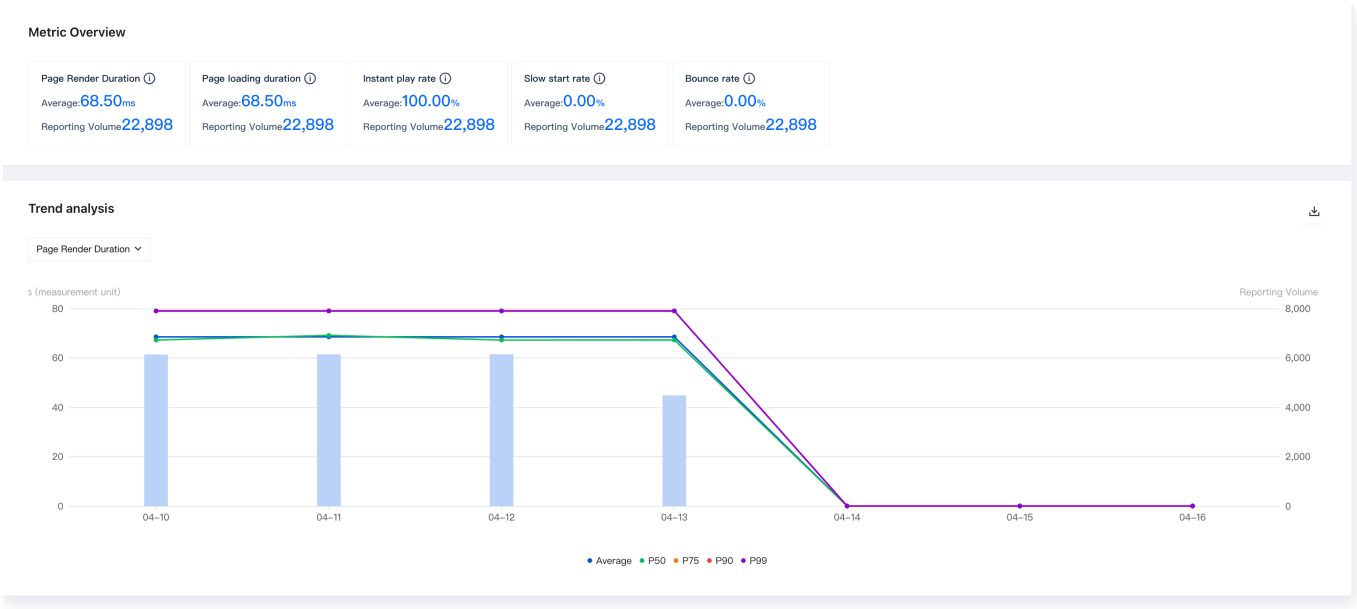
Data analysis

The following content uses the page performance module on the Android platform as an example for explanation. iOS can refer to the following content for analysis.

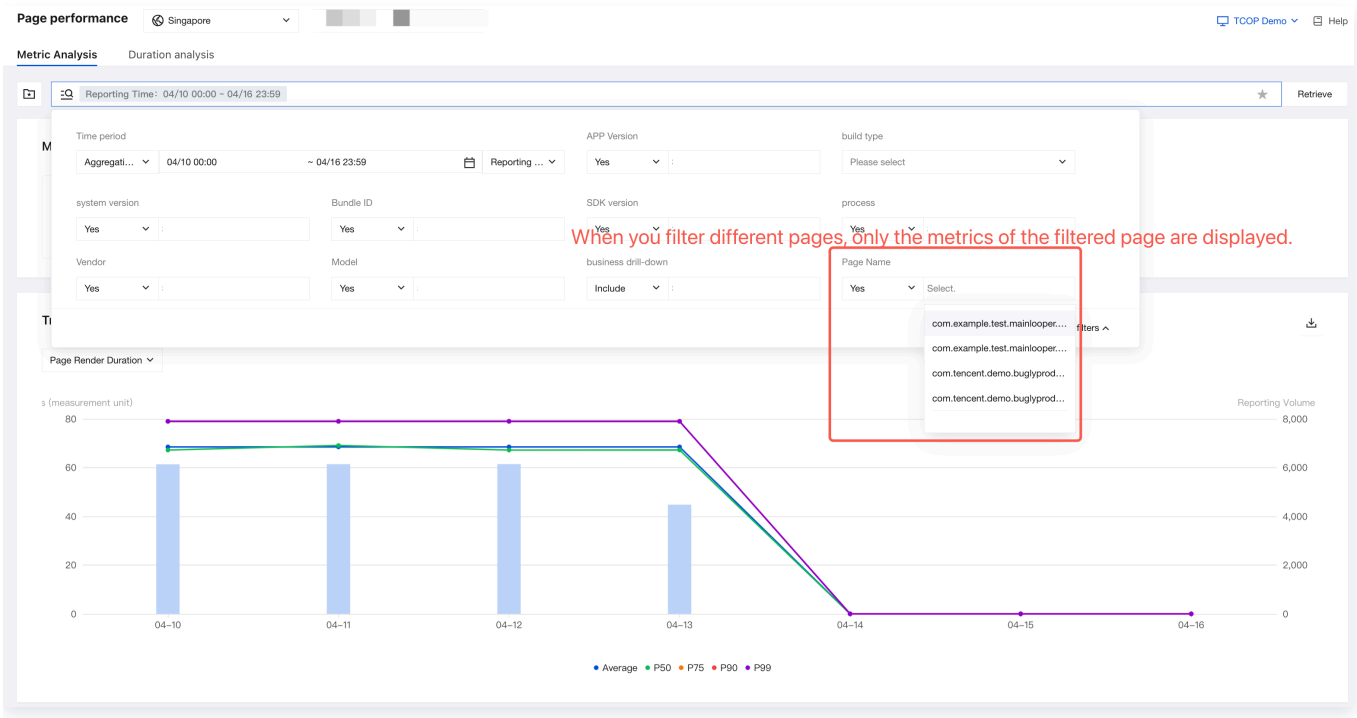
Metric Analysis

On the metric analysis page, the following information can be viewed:

- **Page Render Duration** average and percentile values.
- **Page loading duration** average and percentile values.
- **Instant play rate**: The percentage of cases where page rendering time is less than 1 second.
- **Slow start rate**: The percentage of cases where page rendering time exceeds 3 seconds.
- **Bounce rate**: The percentage of page visits where users exit before rendering completes (which can be used to measure the probability of users encountering a persistent white screen followed by exiting the page).

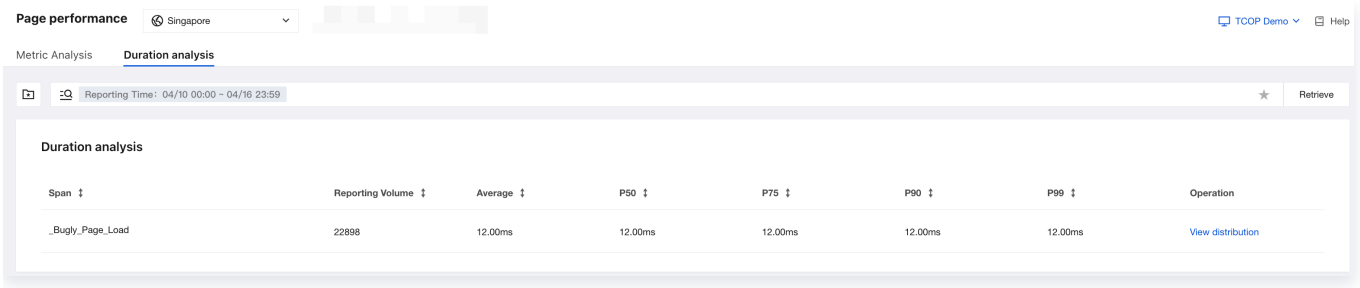


The query area provides numerous filter options. For details, refer to Query. For example, to view metric data only for the com.example.test.mainlooper.TestListView page, simply filter in the page name filter box:



Analyze time consumption

The Performance Analysis page, similar to the overall time consumption metrics of the page, clearly displays the average and percentile values for each Span, and also supports multiple filter options.



Page View

Last updated：2026-05-26 10:17:57

This document provides an overview of the Page View monitoring module for the React Native platform and explains how to use its features. For detailed information, refer to the React Native Integration Guide.

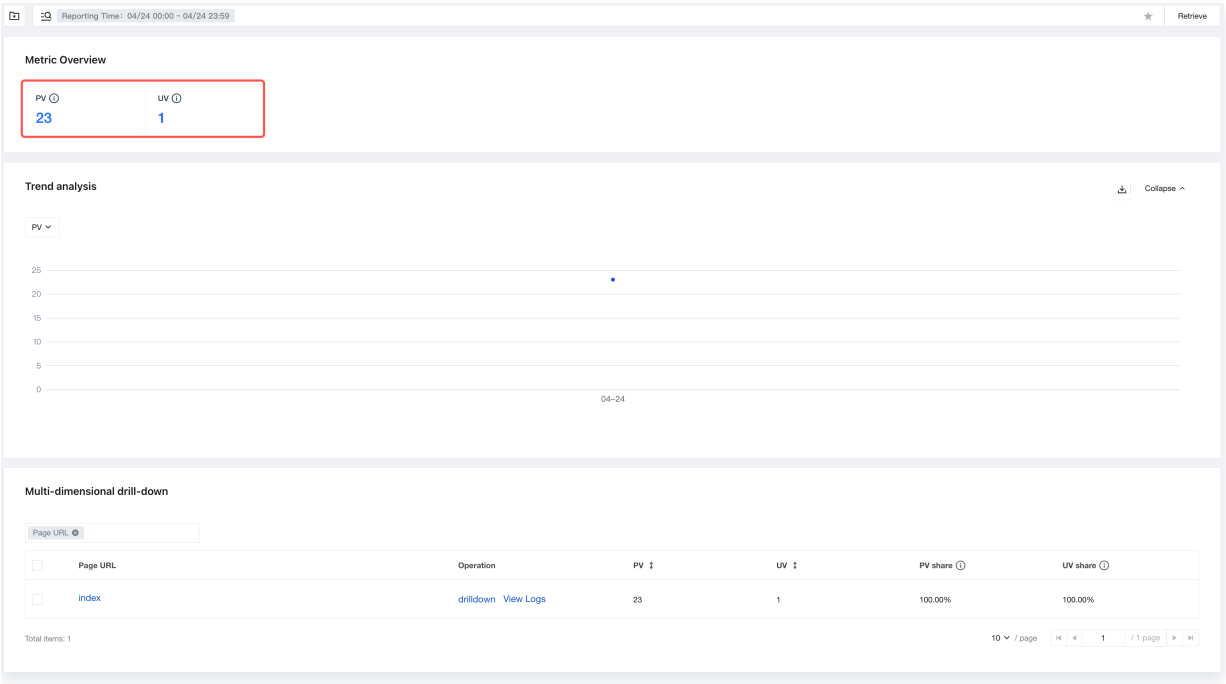
Filter region

For detailed information, please refer to Data Analysis – Query.

Metrics Overview

Displaying core summary metrics for page view.

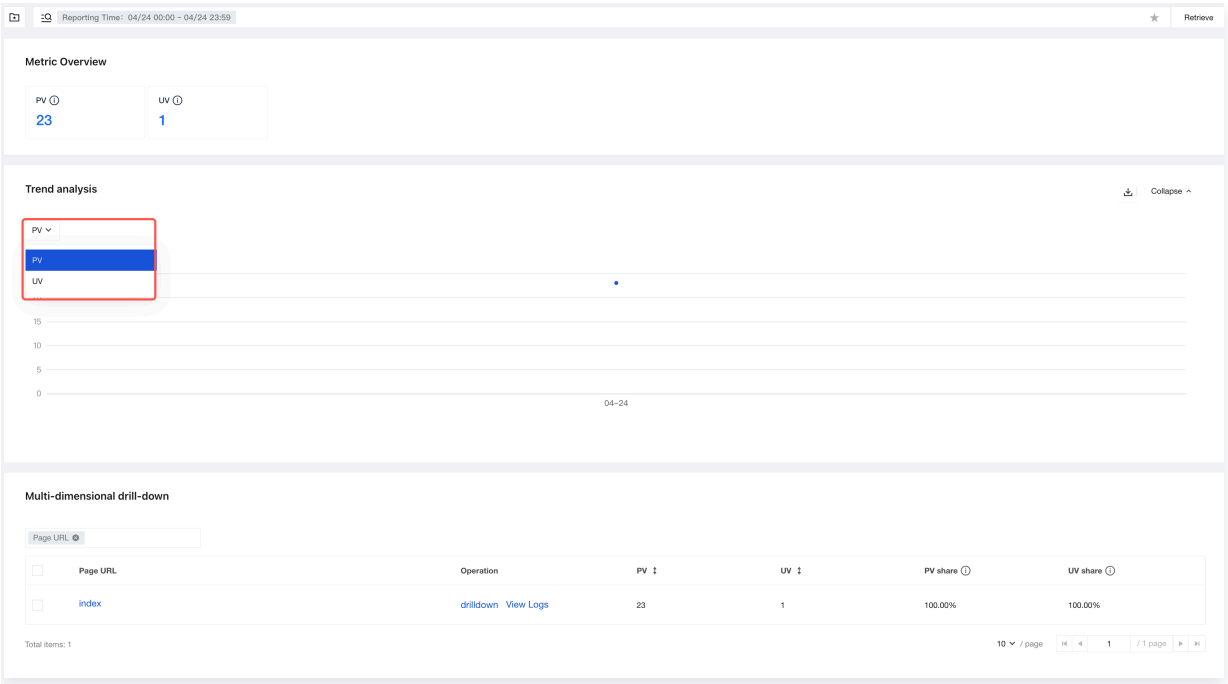
- **PV (Page Views)**: The total number of page views.
- **UV (Unique Visitors)**: The number of unique users accessing the page.



Trend analysis

Displaying the time trends of PV (Page Views) and UV (Unique Visitors) to facilitate observation of traffic pattern changes.

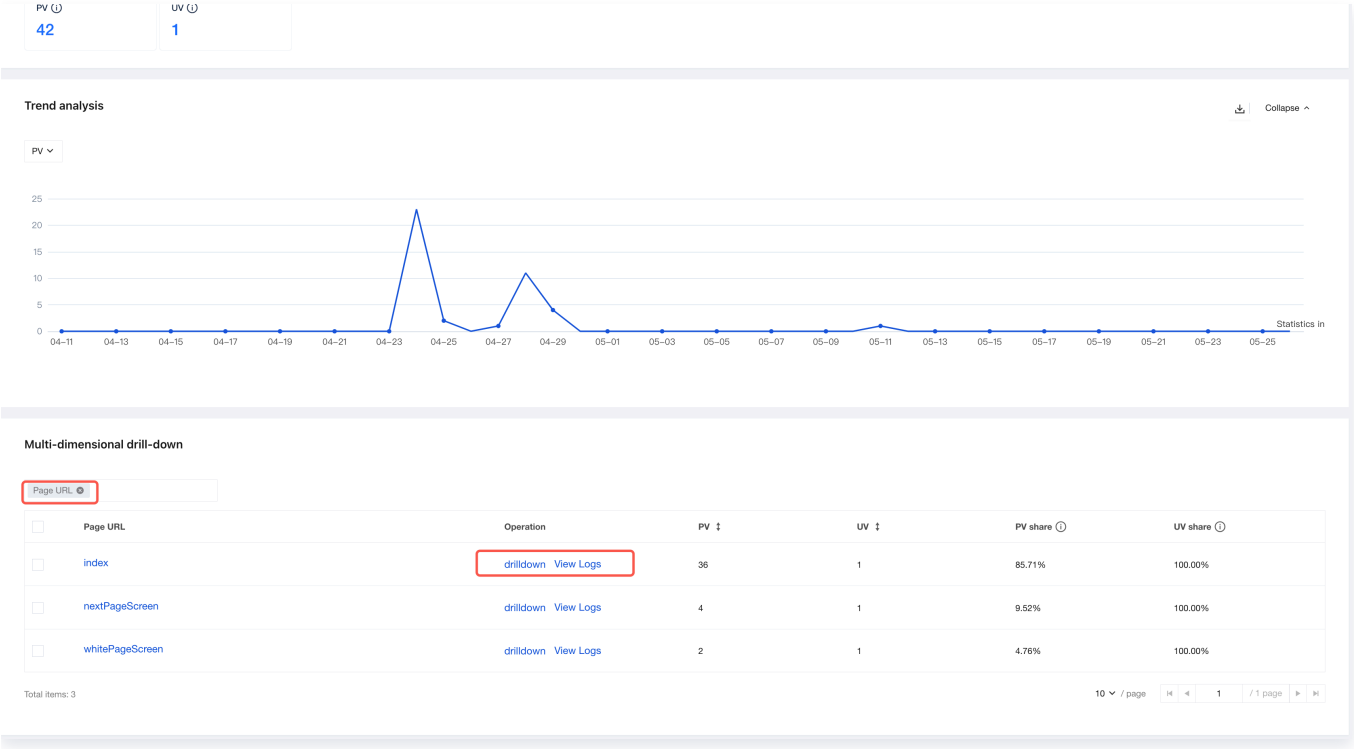
- **Metric Switching**: The left dropdown can switch the display of **PV** or **UV** trends.
- **Line Chart Description**: The chart displays the numerical changes of the corresponding metric over the time axis (the current example shows the trend of PV), enabling intuitive identification of peak/off-peak traffic periods.
- Click the **Download** icon on the right to export trend analysis data within the current time range.
- Supports clicking **Collapse** and **Expand** to adjust the display state of the module.



Multi-dimensional drill-down

This section displays detailed data for specific page URLs, supporting granular viewing.

- **Page URL**: Displays the specific address of the page.
- **Operation Bar**: Supports **drilldown** and **View Logs**.
  - drilldown: View data at a more granular dimension.
  - View Logs: View API request logs.
- **Statistical Fields**: Include PV, UV, PV share, and UV share.



Custom Event

Last updated：2026-05-25 18:53:27

This article introduces the overview of the Custom Event monitoring module for the React Native platform and the usage of features. For detailed information, please refer to React Native Integration Guide.

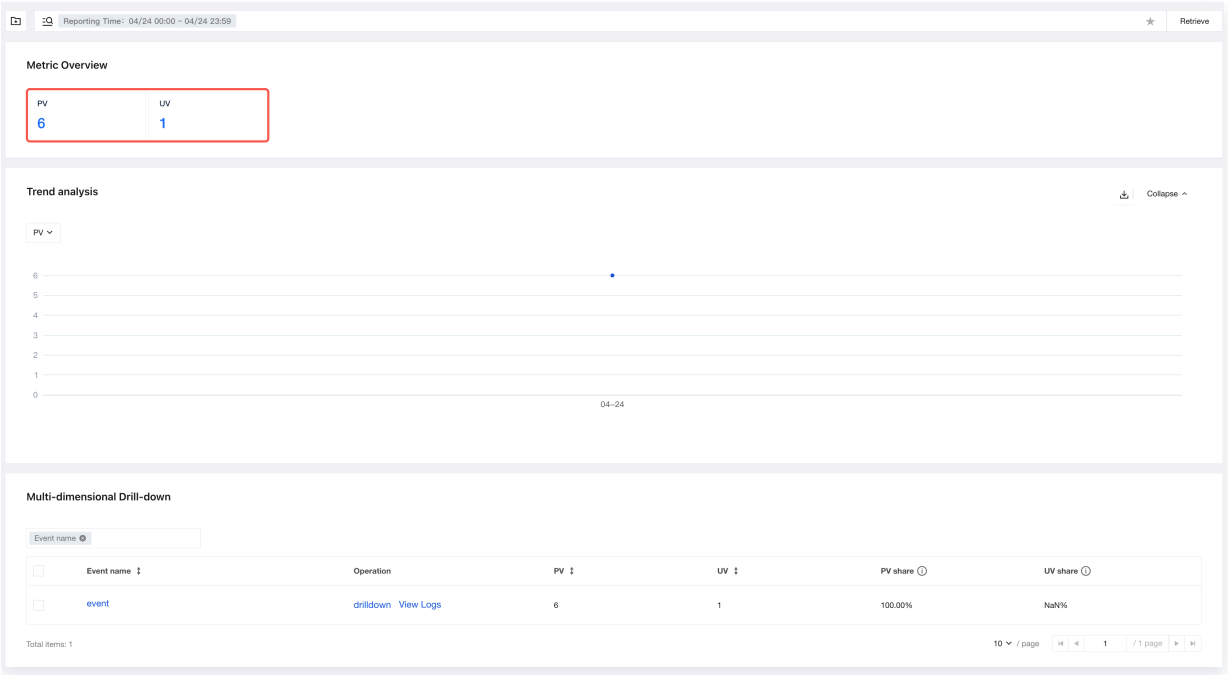
Filter region

For detailed information, please refer to Data Analysis – Query.

Metrics Overview

Display core metrics of events to rapidly assess the overall impact scope.

- **Metric Description**:
  - **PV (Event Report Volume)**: In the example below, it is 6, indicating that a total of 6 event reports were received within this time range.
  - **UV (Unique Users)**: In the example, it is 1, indicating that 1 unique users triggered these events.
- **Abnormality Identification**: If PV or UV suddenly surges or an abnormal event accounts for an excessively high proportion, you can directly go to the **Trend analysis** or **Multi-dimensional Drill-down** module for further investigation.

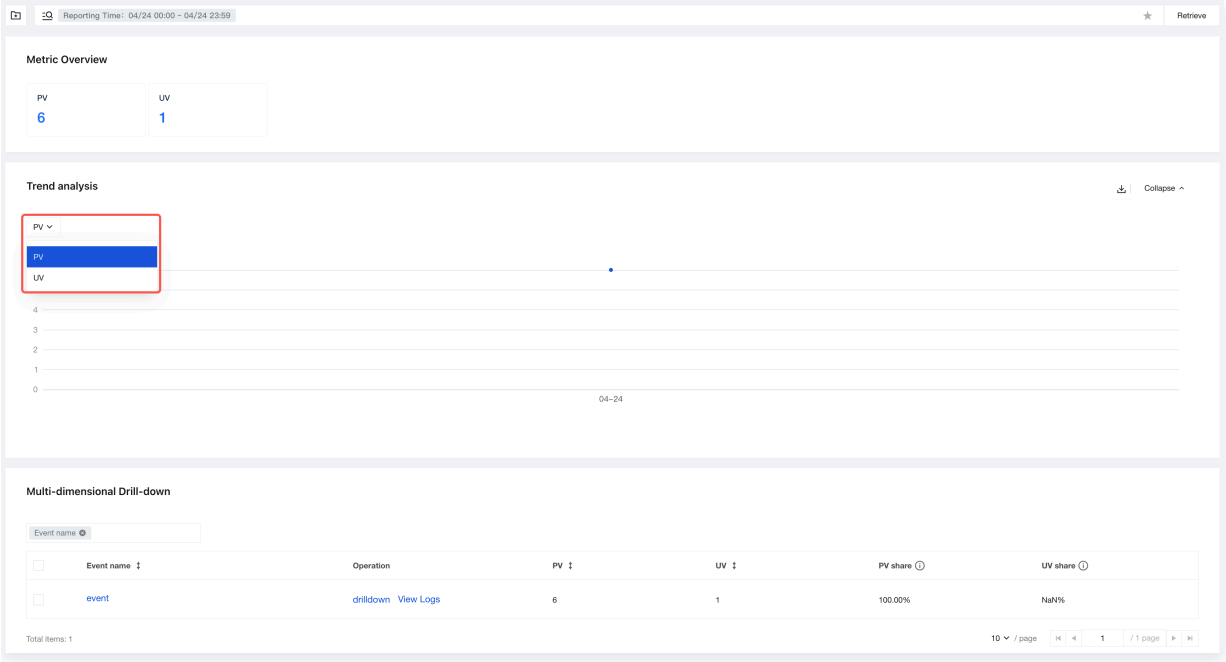


Trend analysis

Analyze the trend of event volume over time to determine whether the issue is sporadic or persistent.

- **Metric Switching**: The left dropdown allows switching between **PV** and **UV** metrics to view temporal trends from different dimensions.

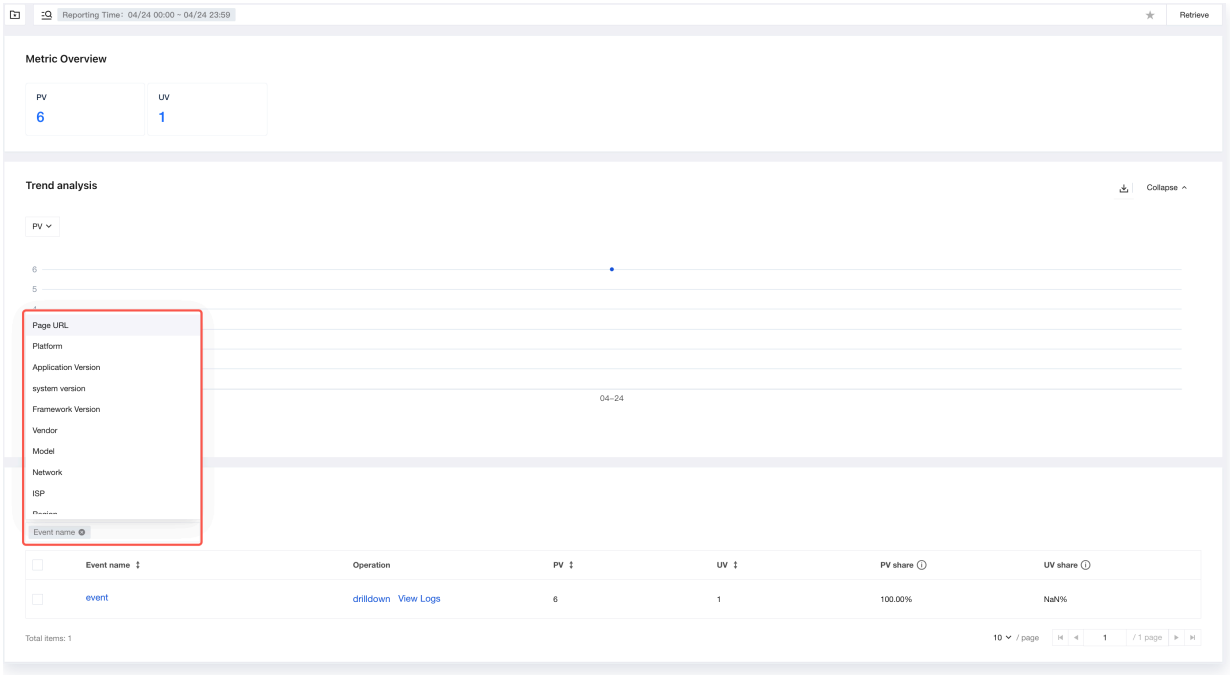
- **Timeline Interaction**: Hover over data points on the timeline to view specific values at that time point. Supports zooming the timeline to adjust the time granularity (e.g., switching from daily to hourly), and focusing on abnormal peak periods, such as the peaks on 11–26 and 12–04 in the example below.
- **Abnormal Point Positioning**: Focus on the peaks in the trend chart. Clicking on a peak data point will link to the multi–dimensional drill–down module, allowing you to view the specific event distribution during that time period.
- **Data Download**: Click the **Download** icon on the right to export trend analysis data within the current time range.
- Supports clicking **Collapse** and **Expand** to adjust the display state of the module.



Multi–dimensional drill–down

Displays the distribution of events by event name dimension to locate high–frequency abnormal events.

- **List View**: The table displays data by **Event Name** by default, including PV, UV, PV Proportion, and UV Proportion. In the example below, XXX Request Successful (PV Proportion 54.32%) is the most frequent event, while renderer–main–error (PV Proportion 33.33%) is the primary abnormal event. Click the dropdown on the left to select other metrics.
- **Drill Down and Log Viewing**:
  - Click the **drilldown** in the operation column to go to the multi–dimensional analysis of the event, for example, drill down by system version or application version.
  - Click **View Logs** in the operation column to obtain detailed reported logs for the event, such as error messages and device information, directly identifying the technical cause.


Reporting Time: 04/24 00:00 ~ 04/24 23:59
Retrieve
Metric Overview
PV
6
UV
1
Trend analysis
Collapse
PV
6
5
Page URL
Platform
Application Version
system version
Framework Version
Vendor
Model
Network
ISP
04-24
Event name
Event name
Operation
PV
UV
PV share
UV share
event
drilldown
View Logs
6
1
100.00%
NaN%
Total items: 1
10 / page
1
/ 1 page

Custom Speed

Last updated：2026-05-25 18:53:27

This article introduces the overview of the Custom Speed monitoring module for the React Native platform and the usage of features. For detailed information, please refer to React Native Integration Guide.

Filter region

For detailed information, please refer to Data Analysis – Query.

Metrics Overview

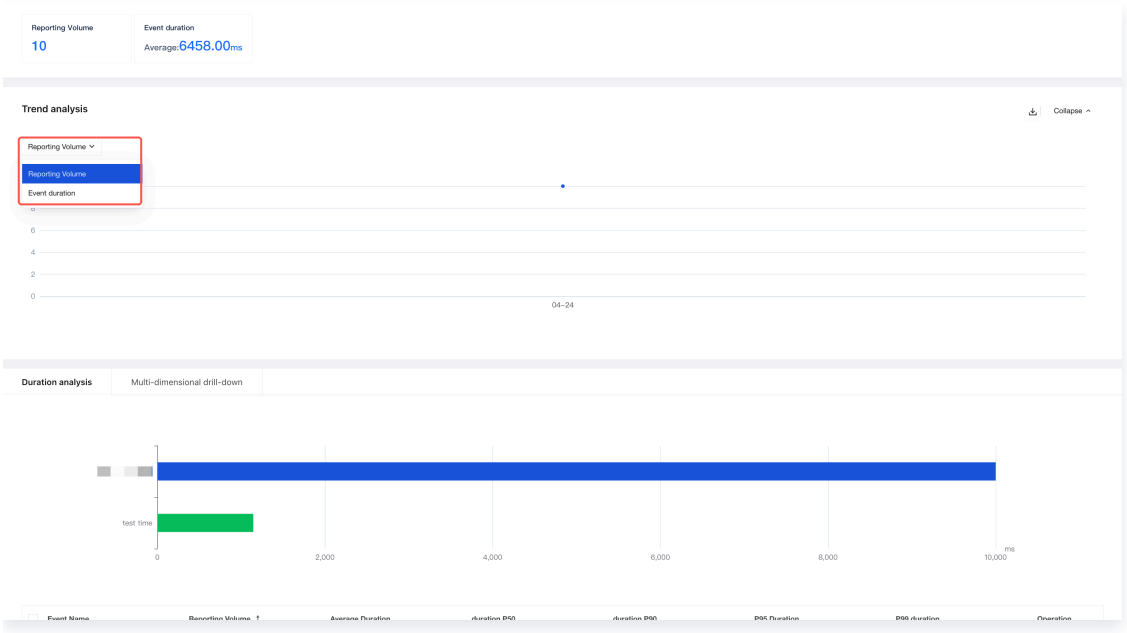
Display core baseline data of custom speed test events to rapidly assess the overall performance.

- **Metric Description**:
  - **Reporting Volume**: indicates the total number of custom speed test events reported within the specified time range. For example, the 5 in the figure below indicates that a total of 5 custom speed test events were reported.
  - **Event Duration**: reflects the average duration of all reported events. A higher value indicates more sluggish operations, for example, 6458ms in the figure below.
- **Abnormality Identification**: If the event duration average suddenly increases (such as exceeding 2000ms) or the reporting volume surges, you can directly go to the **Trend analysis** or **Multi-dimensional drill-down** module for further investigation.

Trend analysis

Analyze the trend of event reporting volume over time to determine whether the issue is sporadic or persistent.

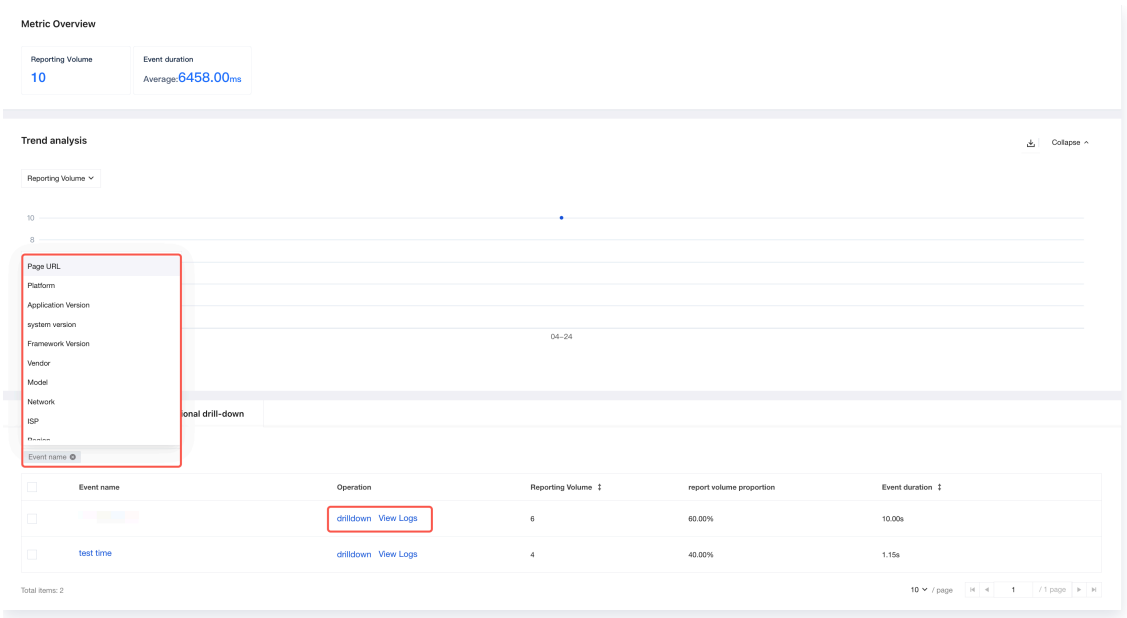
- **Metric Switching**: Click the **Reporting Volume** dropdown to switch to the **Event Duration** metric and view temporal trends from different dimensions.
- **Timeline Interaction**: Hover over data points on the timeline to view specific values at that time point. Supports zooming the timeline to adjust the time granularity, for example, switching from daily to hourly, and focusing on abnormal peak periods, such as the peak on 12–04 in the example below.
- **Abnormal Point Positioning**: Focus on the peaks in the trend chart. Clicking on a peak data point will link to the multi–dimensional drill–down module, allowing you to view the specific event distribution during that time period.
- **Data Download**: Click the **Download** icon on the right to export trend analysis data within the current time range.
- Supports clicking **Collapse** and **Expand** to adjust the display state of the module.



Multi–dimensional drill–down

Displays the distribution of custom speed test events by event name dimension to locate abnormal events with excessive duration.

- **List View**: Displays data by **Event Name** by default, including reporting volume, reporting volume percentage, and average event duration. In the example below, custom speed test is the most frequent event (60.00%), while complexOperation is the most time–consuming event (average 6.00s). Click the dropdown list on the left to drill down into other metrics.
- **Drill Down and Log Viewing**:
  - Click the **drilldown** in the operation column to go to the multi–dimensional analysis of the event, for example, drill down by system version or application version.
  - Click **View Logs** in the operation column to obtain detailed reporting logs for the event, such as trigger scenarios and device information, to directly locate the technical cause of excessive duration.


Metric Overview
Reporting Volume
10
Event duration
Average:6458.00ms
Trend analysis
Collapse
Reporting Volume
10
8
Page URL
Platform
Application Version
system version
Framework Version
Vendor
Model
Network
ISP
04-24
Event name
Event name
Operation
Reporting Volume
report volume proportion
Event duration
drilldown View Logs
6
60.00%
10.00s
test time
drilldown View Logs
4
40.00%
1.15s
Total items: 2
10 / page
1
/ 1 page

Log

Last updated：2026-05-25 18:53:27

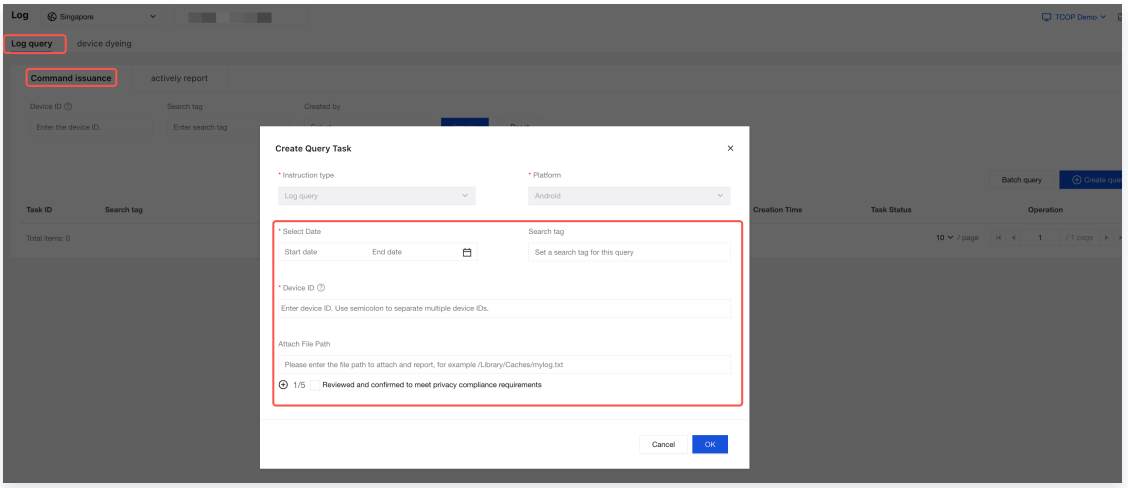
This article introduces the configuration and usage of the platform's log feature. The following content uses the Android platform log module as an example for explanation. iOS and Harmony can refer to the following content.

Log Query

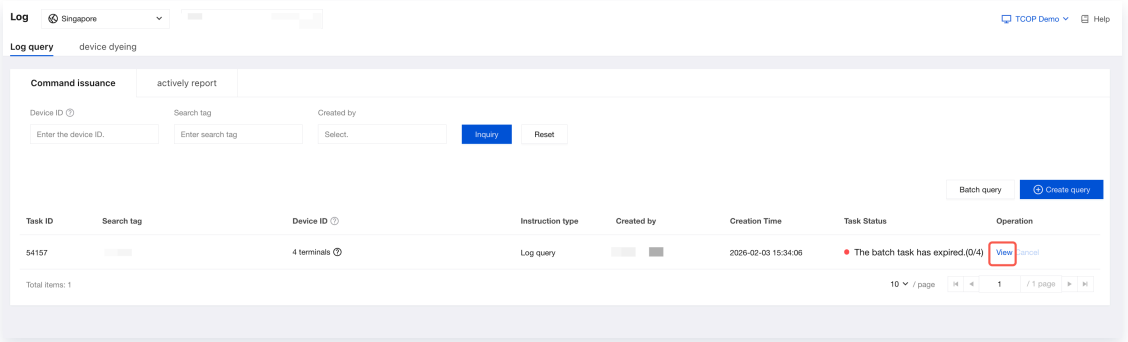
Issuing a Command

The platform provides the feature to issue log retrieval commands. You can issue commands and query reporting results on the Log query > Command issuance page in the console.

1. On the Issue Command page, click Create query to create a log retrieval command. You need to select the **Select Date** and **Device ID** for the logs to be retrieved. You can add tags and attachment file paths for the retrieval.

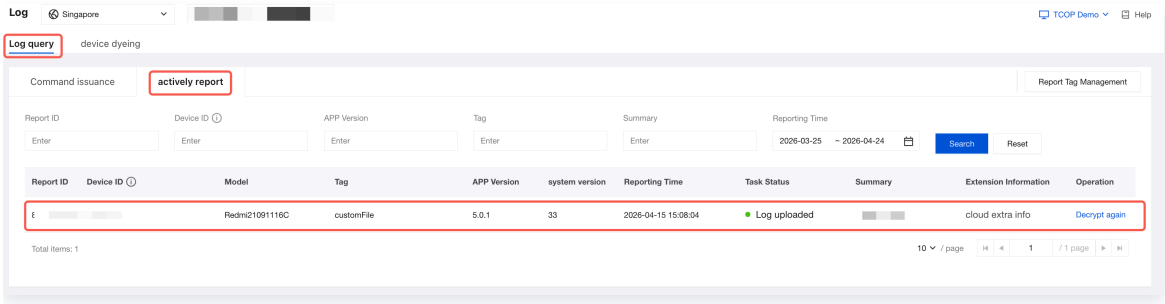


2. After the command is issued, wait for the SDK to pull the configurations and upload the log files.
3. On the Issue Command list page, you can click View to see the latest logs retrieved by the current task.



Actively reporting

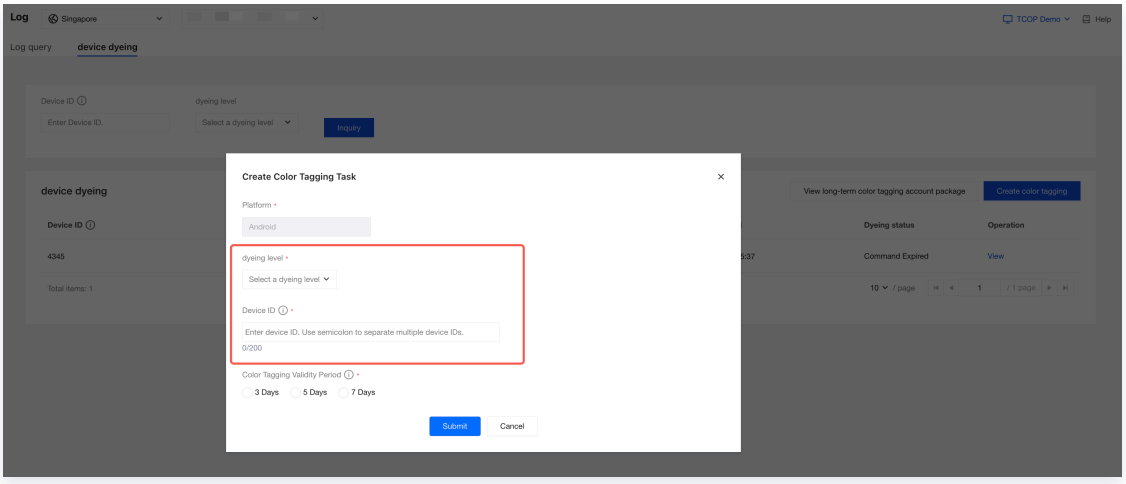
The platform supports businesses actively calling the reporting API to upload custom logs, files, and other information. For details, refer to Data Reporting Verification.



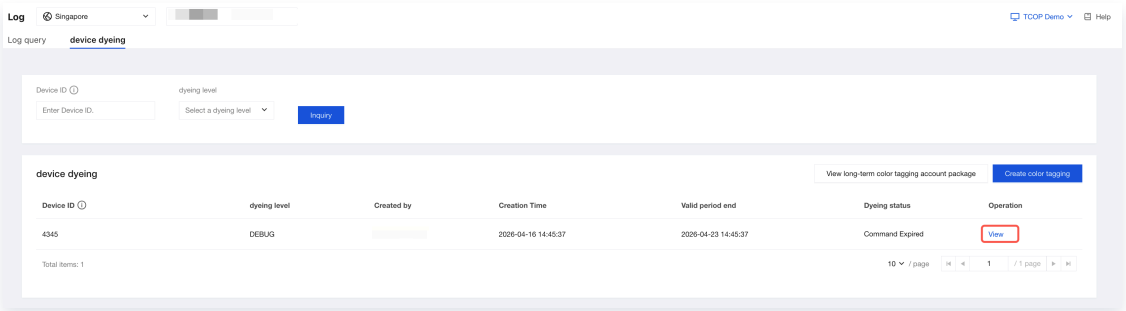
Device dyeing

The platform management console provides the capability to issue log dyeing instructions, enabling online configuration of the log level for local printing.

1. On the Log > device dyeing page in the console, you can issue log dyeing instructions. On the **device dyeing** page, click Create color tagging, select the **dyeing Level**, enter the **Device ID**, and click **Submit**.



2. After the command is issued, the SDK will automatically call the log level setting API to adjust the current log printing level upon pulling the configurations.
3. On the device dyeing list page, you can click View to see the details of the staining task.



Setting
SDK configuration

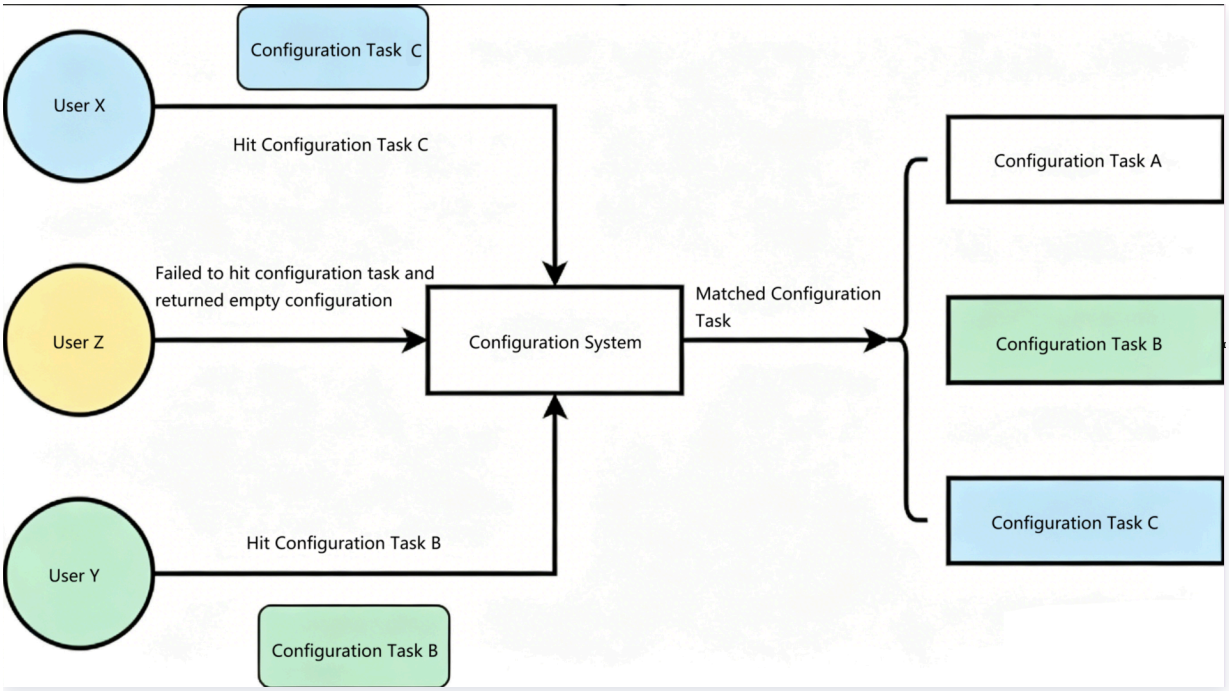
Last updated：2026-05-25 18:53:28

RUM Pro supports controlling the behavior of the SDK through task configuration in the management console. Users can control whether the monitoring feature is enabled and whether uploading custom files is allowed through configuration. This article mainly introduces how to use the SDK configuration of RUM Pro.

Overview

The platform supports rich configuration capabilities. Users can control the behavior of the SDK through configuration in RUM Pro > Setting > SDK configuration, such as whether to enable a specific monitoring feature or whether to allow uploading custom files.
After the application is created, the configuration task list is empty, and users need to create configuration tasks. To modify existing configuration tasks, the system also supports making adjustments. All operations performed on configurations will be displayed in the **Configuration modification history**.
Users can create multiple configuration tasks, setting appropriate priorities and filtering criteria for each task. When an application pulls configurations, the configuration system selects configuration tasks that meet the filtering criteria based on the parameters in the request and returns the task with the highest priority to the application.

- Filtering criteria: Describes the characteristics that users who pull configurations must meet for a configuration task.
- Priority: Describes how to select a task when a user matches multiple configuration tasks. Currently, the task with the highest priority is selected. If multiple matched tasks have the same priority, the configuration system will randomly select one task to return.



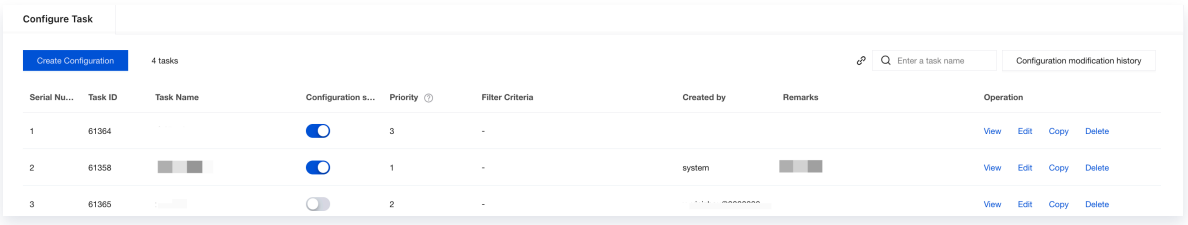
Create configuration task

1. Go to the RUM Pro > Setting > SDK configuration page, click **Create Configuration** to create a new configuration task for the corresponding product.
2. Configuration task with the following configuration items:

Configuration Item	Description
Task Name	Configuration task name.
Condition	The configuration task sets filtering criteria to restrict that only devices meeting the criteria can pull configurations. By default, the filtering criteria are empty, meaning no restrictions apply. The filtering criteria follow an AND relationship, requiring all conditions of the current configuration task to be met. The filtering criteria for the current task will be displayed in the configuration task list. For supported configuration items, see Filtering Criteria.
Priority	Describes how to select a task when a user matches multiple configuration tasks. Currently, the task with the highest priority is selected. If multiple matched tasks have the same priority, the configuration system will randomly select one task to return.
Remarks	None.

Configuration content.	Configuration content involves detailed configuration items such as SDK sampling and collection mode. For specifics, see Configuration Content.

3. After configuration, click **Submit** to view the newly created task in the configuration task list. The newly created task is disabled by default and requires manual enabling by the user.

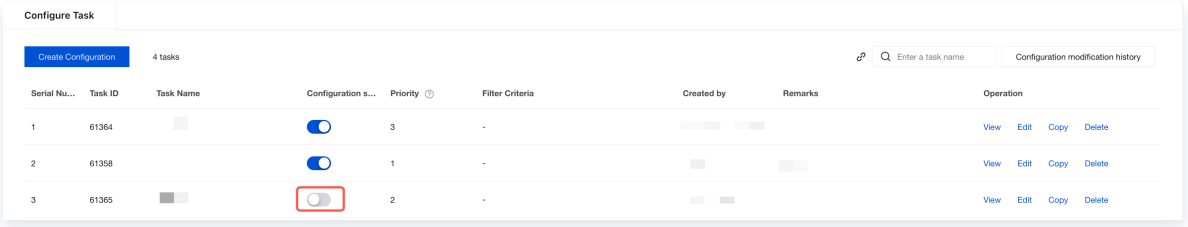


Manage configuration tasks

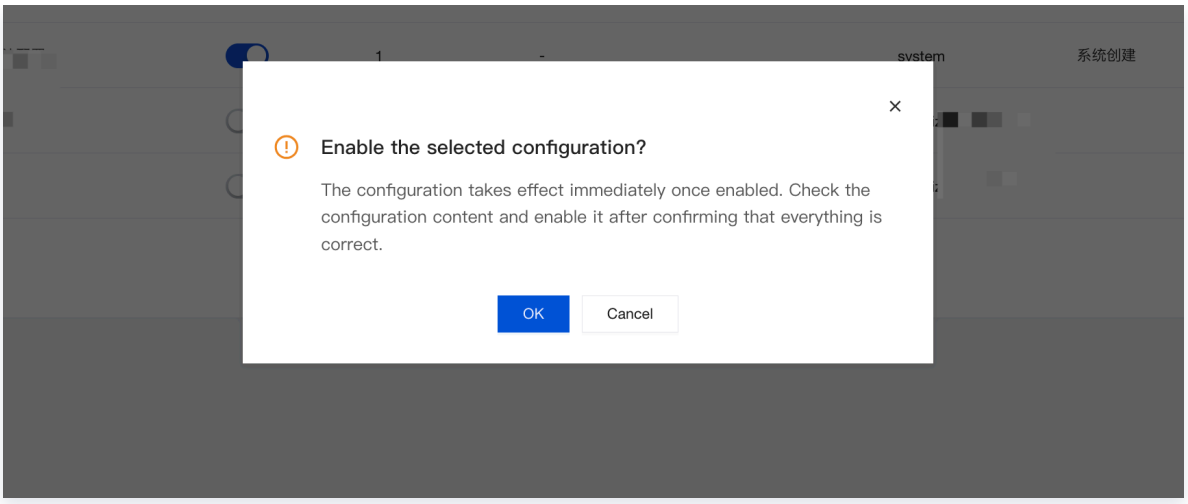
Enable/Disable Configuration

1. Users can directly operate the configuration status to enable or disable configuration tasks.

> **Note:**
> When applications pull configurations, the configuration system only matches against enabled configuration tasks. Disabled tasks do not participate in matching, and applications cannot pull configurations from disabled tasks. To temporarily deprecate a configuration task, users can disable it.

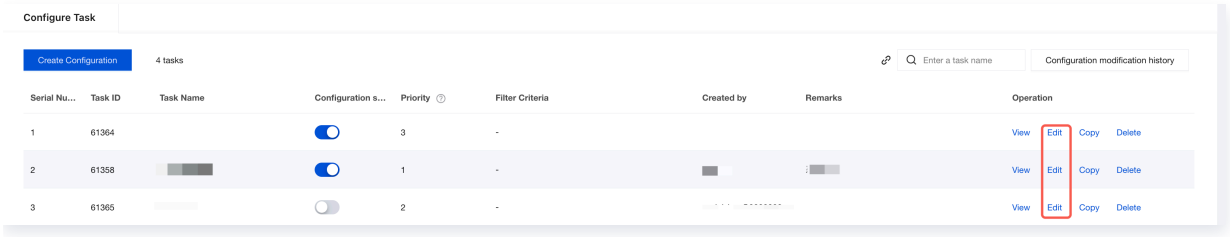


2. Click **OK** to enable or disable the configuration task.

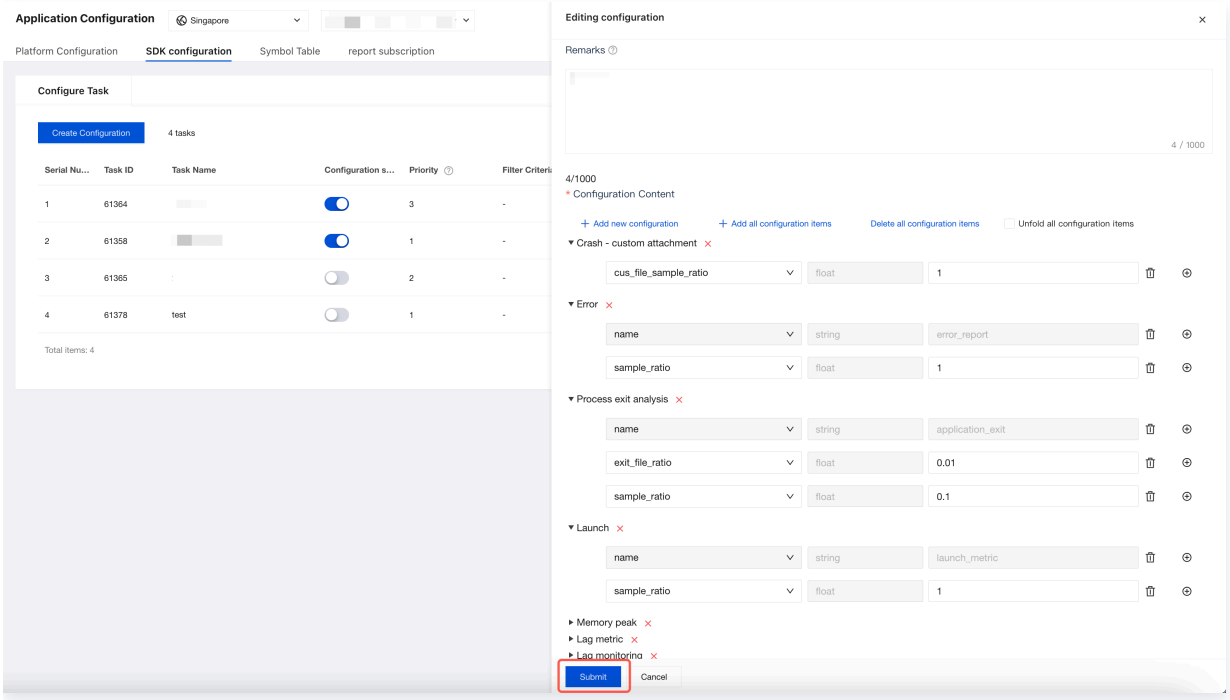


Modify configuration tasks

Both active configuration tasks (enabled tasks) and inactive configuration tasks (disabled tasks) can be modified. Click **Edit** in the Actions column to make changes.

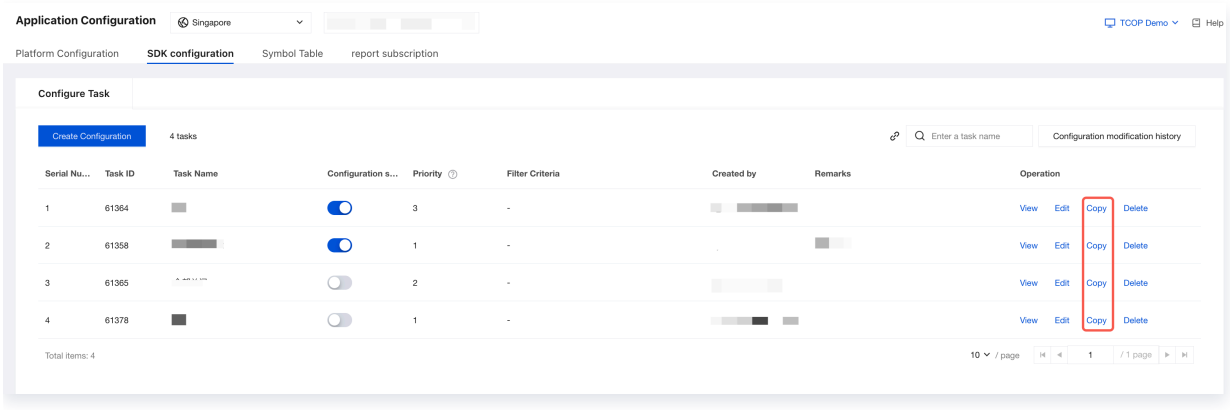


After modifying the configuration task, click **Submit** to complete the update.

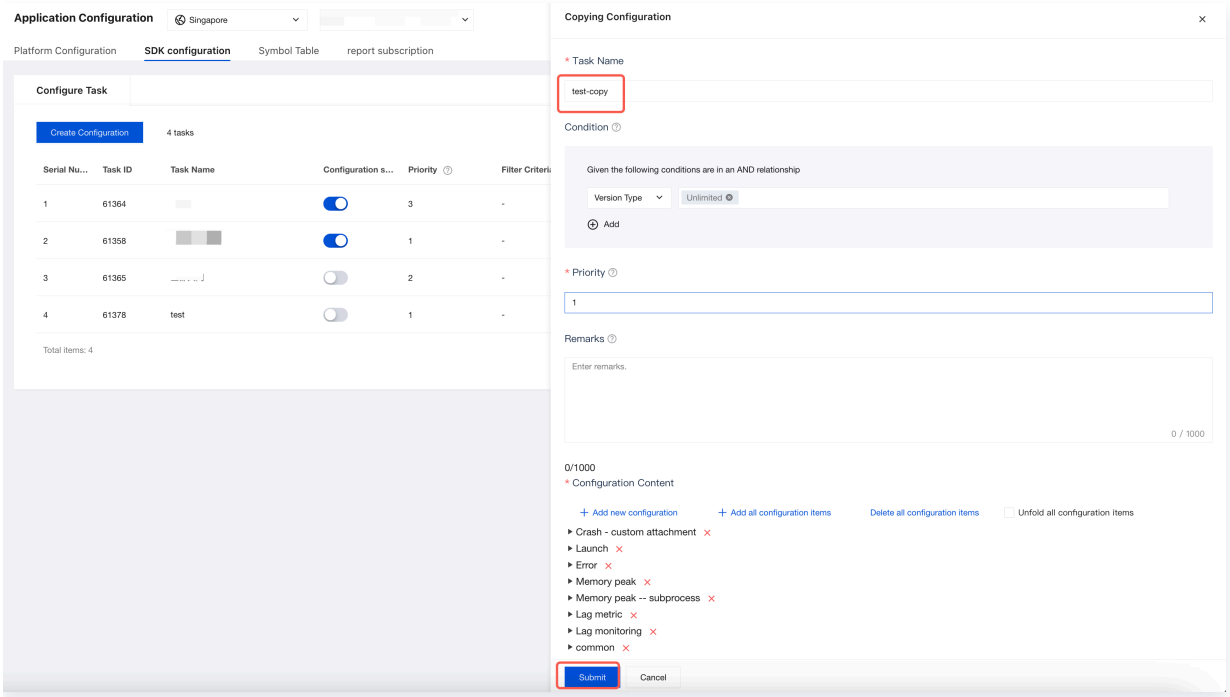


Copy Configuration Task

Both active configuration tasks (enabled tasks) and inactive configuration tasks (disabled tasks) can be copied. Click **Copy** in the Actions column to create a copy of the specified task configuration and enter edit mode.



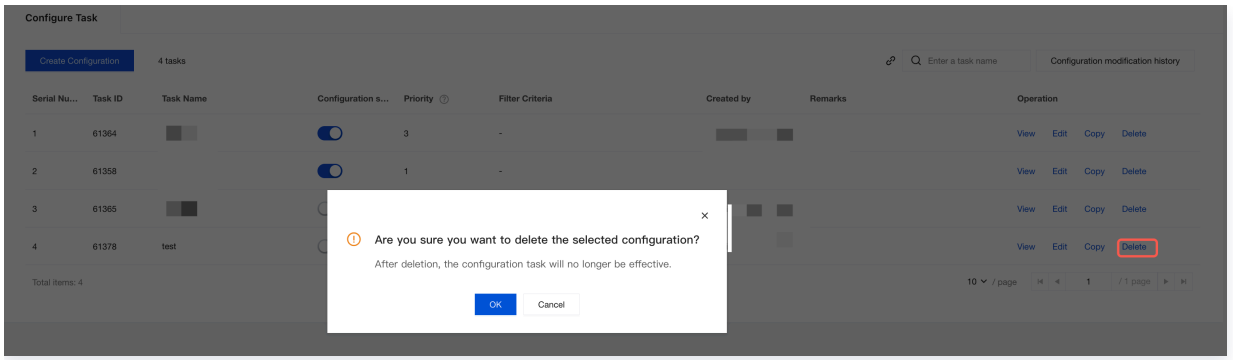
Modify the configuration task according to the copy purpose. After clicking **Submit**, a new configuration will be generated.



Delete the configuration task

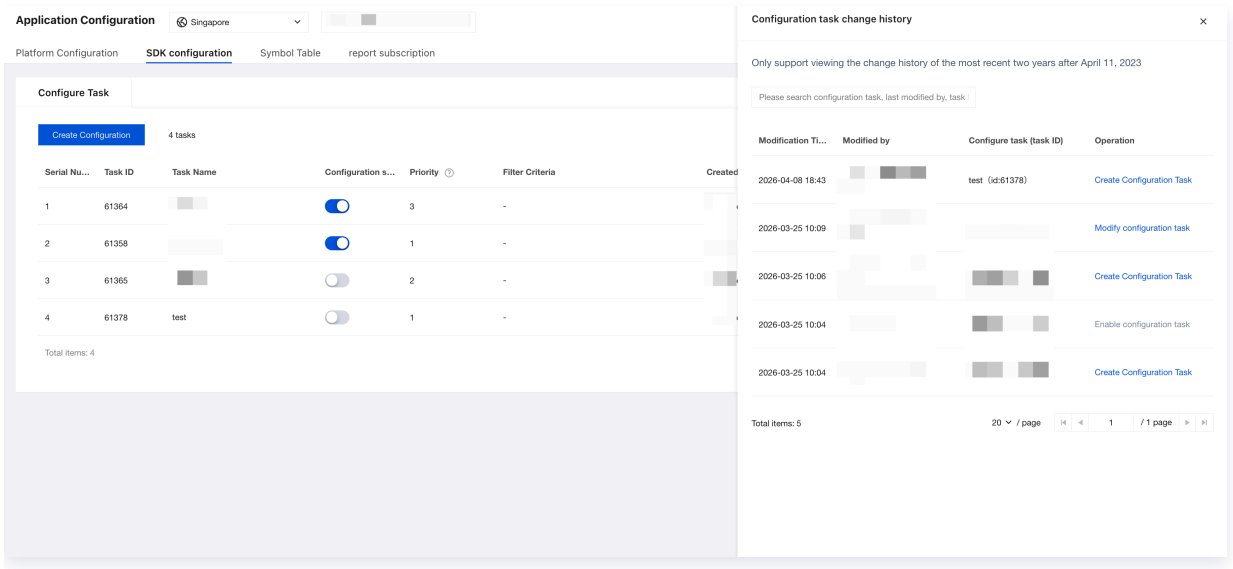
Both active configuration tasks (enabled tasks) and inactive configuration tasks (disabled tasks) can be deleted. After an active configuration task is deleted, applications will no longer pull the deleted configuration task during subsequent fetches.

Click **Delete** in the Actions column, and then click **OK** in the pop-up window to complete the deletion.

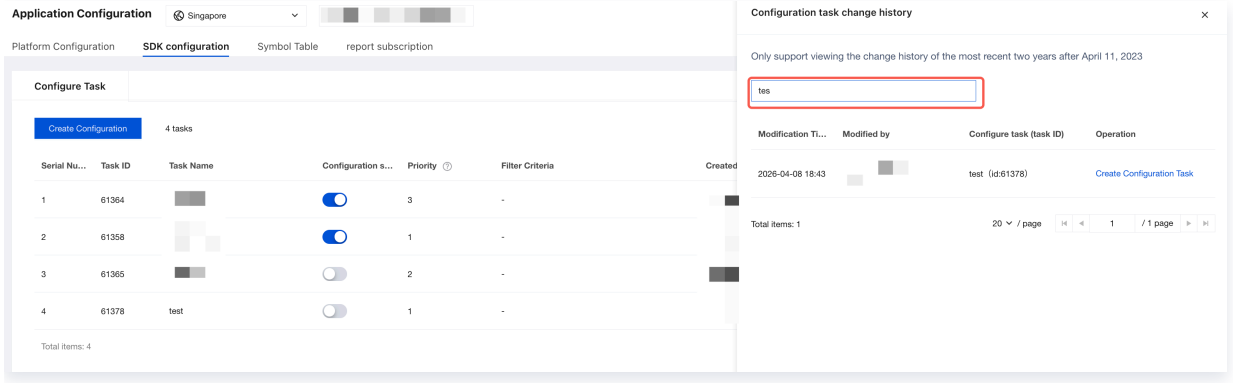


View Modification History

Configuration modification history records any operations performed by users on configuration tasks, including creating new configuration tasks, enabling or disabling configuration tasks, editing configuration tasks, and deleting configuration tasks.

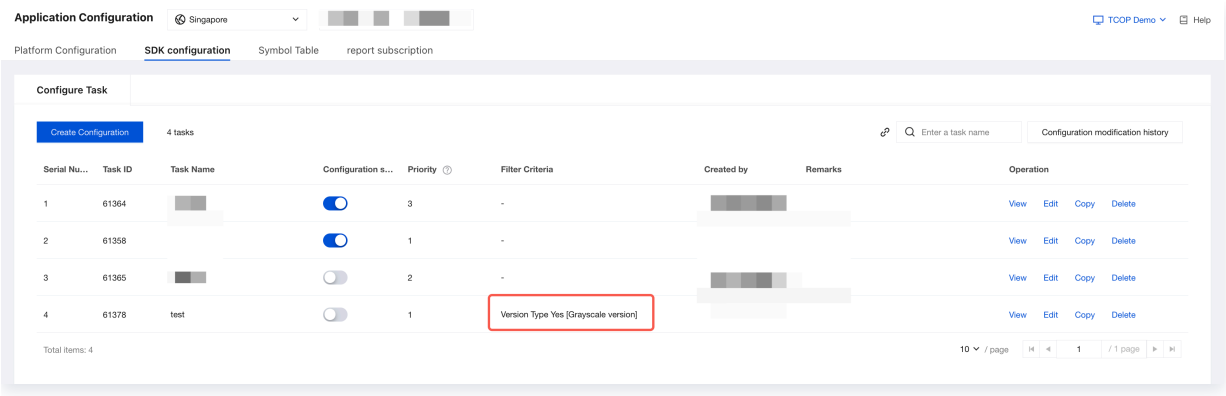


Configuration modification history supports viewing changes for a specific configuration task. As shown in the following example, you can view the modification history of the allowlist configuration. It also supports viewing modification history for specific users.



Filter Criteria

The filtering criteria will be displayed in the configuration task list.

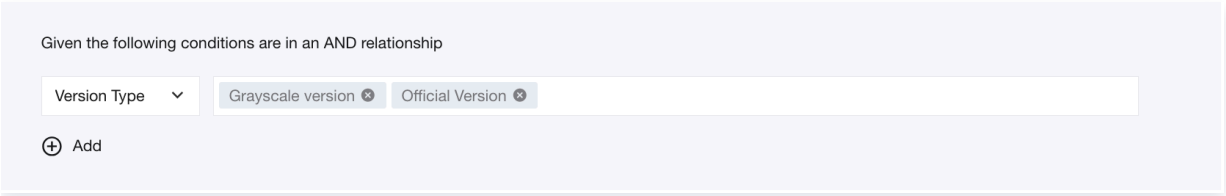


Version type

Version type refers to the application version type, used to describe the current application version. It supports development version, gray version, and official version, and allows multiple selections, defaulting to no restriction. If the application is not set, the default value of the application version type is **Unlimited**. It is recommended to set the correct application version type when initializing the SDK.

- **Android SDK**: Set the application version type via the initialization parameter `BuglyBuilder#appVersionType` .
- **iOS SDK**: Set the application version type via the `BuglyConfig#setBuildConfig` API.

Example: Set the Version Type for pulling configurations. Only applications with Grayscale version and Official Version can pull the current configuration task.

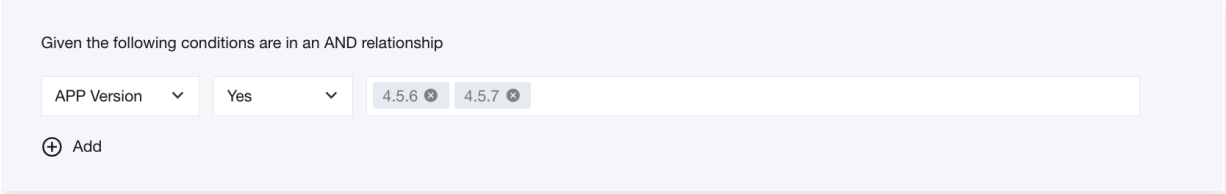


APP Version

APP Version refers to the application version, which describes the version number of the application that integrates the SDK.

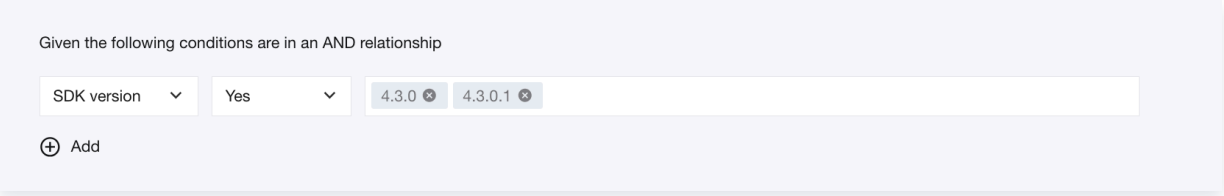
- **Android SDK**: Set the APP Version via the initialization parameter BuglyBuilder#appVersion.
- **iOS SDK**: Set the application version via the `BuglyConfig#setAppVersion` API.

Example: The current configuration task can be pulled only when the APP Version is "4.5.6" or "4.5.7".



SDK version

The SDK version refers to the version of the RUM Pro SDK, corresponding to the SDK version used during integration. It supports multiple versions, with no restriction by default.
Example: The current configuration task can be pulled only when the SDK version is "4.3.0" or "4.3.0.1".

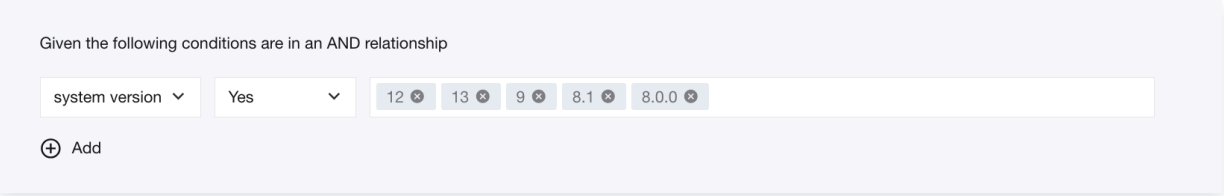


system version

system version refers to the version of the operating system on which the application is currently running. The SDK directly reads the system version property. The system version supports multiple values by default with no restriction.

- **Android SDK:** Automatically obtains the `Build.VERSION.RELEASE` field during SDK initialization, with formats such as "12", "13", "9", "8.0.0".
- **iOS SDK**: Automatically obtains the `[[UIDevice currentDevice] systemVersion]` field during SDK initialization, with formats such as "14.1", "16.1".

Example: The current configuration task can be pulled only when the system version is 12, 13, 9, 8.1, or 8.0.0.

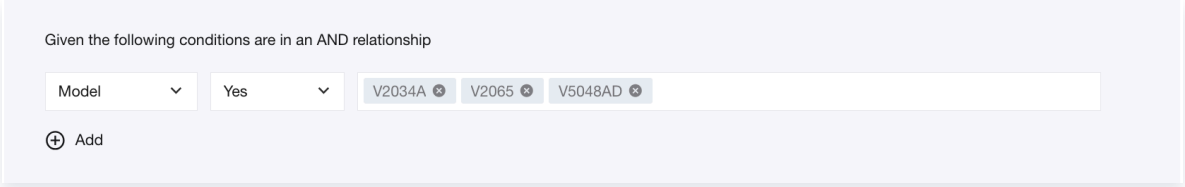


Model

Device model refers to the type of equipment, generally used to denote a specific model of device.

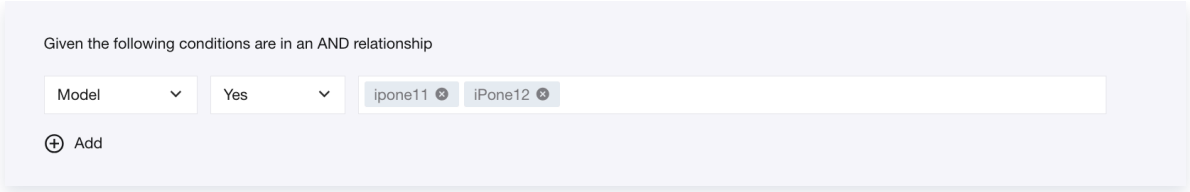
- **Android SDK**: Set the device model via the initialization parameter `BuglyBuilder#deviceModel`. If not set by the application, Build.MODEL will be automatically read during initialization.

  Example: The current configuration task can be pulled only when the device model is V2034A, V2065, or V5048AD.



- **iOS SDK:** Automatically obtains the sysctlbyname field during SDK initialization, with formats such as "iPhone13,4", "iPhone15,3".

  Example: The current configuration task can be pulled only when the device model is iphone11 or iPhone12.

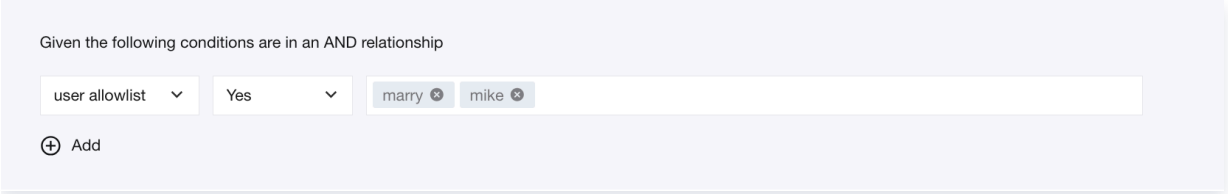


user allowlist

The user allowlist corresponds to the user IDs in the SDK, supports multiple values, and has no restriction by default.

- **Android SDK**: Set the user ID via the initialization parameter `BuglyBuilder#userId`. If not set by the application, the user ID will be empty.
- **iOS SDK**: Set the user ID via the `BuglyInitParam#setUserId` API. If not set by the application, the user ID will be empty.

Example: The current configuration task can be pulled only when the user ID is "marry" or "mike".

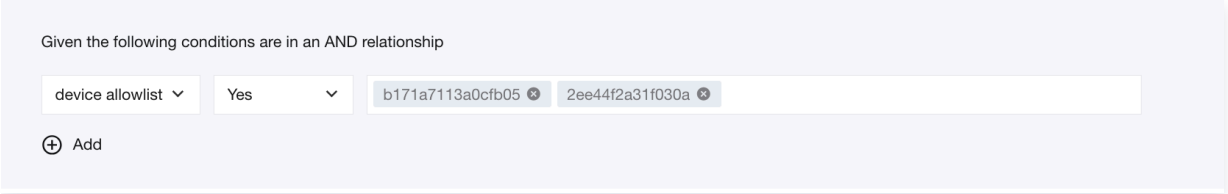


device allowlist

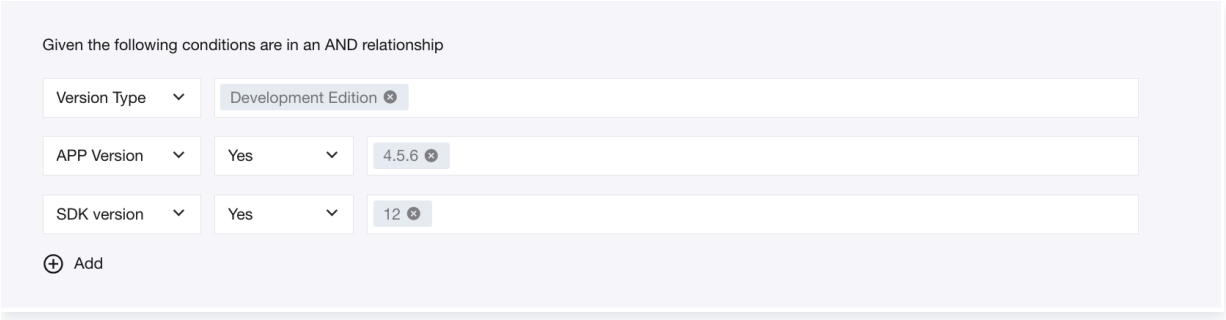
The device allowlist corresponds to the device IDs in the SDK, supports multiple values, and has no restriction by default.

- **Android SDK**: Set the device ID via the initialization parameter `BuglyBuilder#uniqueId`. If not set by the application, the SDK first reads it from the SP cache. If the cache is empty, a random ID is generated and stored in the cache. Once the user sets the device ID, the SDK overrides the SP cache with the application-set device ID.
- **iOS SDK**: Set the device ID via the `BuglyInitParam#setDeviceId` API. If not set by the application, the device ID will be empty. This field is used to collect statistics on device crash rate, device FOOM rate, and other device abnormal rates. Be sure to set it.

Example: The current configuration task can be pulled only when the device ID is "b171a7113a0cfb05" or "2ee44f2a31f030a".



Multi-condition combination example: If the edition type is development edition, the APP version is "4.6.7", and the OS is Android 12, then this configuration task meets the conditions.



Configuration content

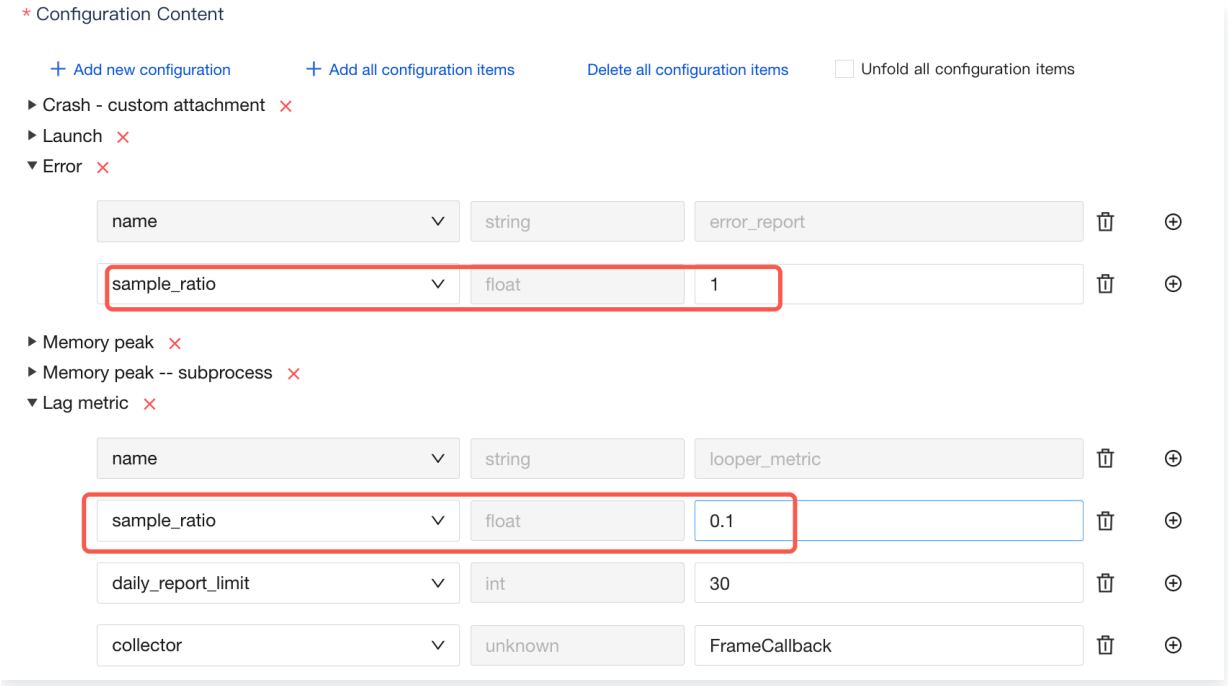
Android and iOS platforms support slightly different configuration items. For detailed descriptions of the configuration items, see the integration guides for both SDKs. Apart from the specific configuration differences, the logic remains consistent on both platforms.

device sampling ratio

Most configuration items include a "sample_ratio" parameter, which indicates the device sampling rate for that monitoring capability. Currently, background sampling is implemented. The configuration system notifies the SDK based on the devices that pull the configuration, determining whether the current device is allowed to enable or disable a specific monitoring item. The value range is (0,1):

- 1: Indicates that all devices are allowed to enable a specific monitoring item.
- 0: Indicates that all devices are not allowed to enable a specific monitoring item.
- A value a in the range (0,1): Indicates that some devices are allowed to enable a specific monitoring item with probability a, while others are not. For example, a=0.3 means 30% of devices are allowed to enable the specified monitoring item, and 70% are not.

Example: All devices are allowed to enable error monitoring, while only 10% of devices are allowed to enable the lag metric feature.



Android Configuration Items

- Crash, ANR, and OOM data is 100% reported, does not support sampling, and is enabled by default. All other monitoring items support sampling. Generally, performance metric monitoring is enabled by default, while abnormal performance monitoring is disabled by default. You can adjust these settings through SDK configuration. During integration, please ensure the enabled features meet your expectations.
- To modify the configuration switch and specify the sampling rate for monitoring items, go to RUM Pro > Setting > SDK configuration.
- During the integration period, full sampling is recommended—adjust all sampling rates to 1.0 to facilitate data reporting verification. After integration is complete, adjust the sampling rates as needed. Alternatively, create multiple configuration tasks based on the edition type (corresponding to the initialized builder.appVersionType), configuring development tasks, canary tasks, and production tasks. Enable all monitoring items for development tasks, perform on-demand sampling during the canary phase, and reduce sampling in production releases according to requirements.

The configuration parameters for each monitoring item are described as follows:

Quality Monitoring

Monitoring Items	Configuration Item	Configuration Parameter	Parameter Description
Crash Monitoring	"crash"	"cus_file_sample_ratio"	The sampling rate for reporting custom files, with a value range of [0,1]. • 1 indicates report all • 0 indicates do not report

			• Otherwise, sampled reporting is applied; by default, report all.
Error Reporting	"error_report"	"sample_ratio"	Error Reporting sampling rate, with a value range of [0,1]. • 1 indicates report all • 0 indicates do not report • Otherwise, user-sampled reporting is applied; by default, report all.

Launch Monitoring

Monitoring Items	Configuration Item	Configuration Parameter	Parameter Description
Launch Metrics	"launch_metric"	"sample_ratio"	Device sampling rate, with a value range of [0,1]. • 1 indicates enable all • 0 indicates disable all • Otherwise, enable with a specified probability.

Lag Monitoring

Monitoring Items	Configuration Item	Configuration Parameter	Parameter Description
Lag Metrics	"looper_metric"	"sample_ratio"	Device sampling rate, with a value range of [0,1] • 1 indicates enable all • 0 indicates disable all • Otherwise, enable with a specified probability.
Lag Monitoring	"looper_stack"	"sample_ratio"	Device sampling rate, with a value range of [0,1]. • 1 indicates enable all • 0 indicates disable all • Otherwise, enable with a specified probability.
		"event_sample_ratio"	Message sampling rate, with a value range of [0,1]. • 1 indicates to monitor all UI thread messages. • 0 indicates to monitor no UI thread messages. • Otherwise, specify a probability to sample and monitor messages from UI threads.

		"stack_interval_ms"	Lag stack capture interval, in ms.
		"threshold"	Lag detection threshold, in ms.
		"detect_strategy"	Lag monitoring granularity. Currently supports msg and vsync, with msg as the default value.

Memory Monitoring

Monitoring Items	Configuration Item	Configuration Parameter	Parameter Description
Memory Peak	"memory_quantile"	"sample_ratio"	Device sampling rate, with a value range of [0,1]. • 1 indicates enable all • 0 indicates disable all • Otherwise, enable with a specified probability.
Java Memory Leak Monitoring	"activity_leak"	"sample_ratio"	Device sampling rate, with a value range of [0,1]. • 1 indicates enable all • 0 indicates disable all • Otherwise, enable Activity leak monitoring with a specified sampling probability.
		"event_sample_ratio"	Event sampling rate, with a value range of [0,1]. • 1 indicates that when Activity's onDestroy is called or when startInspectLeakObj is called, fully enable Java object leak inspection. • 0 indicates do not enable Java object leak inspection; • Otherwise, enable Java object leak inspection with a specified sampling probability; For example, a value of 0.5 indicates that when Activity's onDestroy is called or when startInspectLeakObj is called, there is a 50% probability of enabling Java object leak inspection.
Java Memory Peak Monitoring	"java_memory_ceiling_hprof"	"sample_ratio"	Device sampling rate, with a value range of [0,1]. • 1 indicates enable all • 0 indicates disable all • Otherwise, enable Java memory peak monitoring with a specified sampling probability.

		"threshold"	Memory peak threshold percentage, with a value range of [0,100], indicates that when the threshold percentage of Java's maximum available memory is reached, it is identified as a memory peak event.
		"event_sample_ratio"	Event Sampling Rate, with a value range of [0,1], controls the probability of reporting events when Java memory reaches its peak threshold.
Big Graph Analytics	"big_bitmap"	"sample_ratio"	Device sampling rate, with a value range of [0,1]. • 1 indicates enable all • 0 indicates disable all • Otherwise, enable with a specified probability.
		"threshold"	Image over-limit threshold, value range: (100, Integer maximum value), image over-limit rate = Length of decoded image × Width of decoded image / (Length of display area × Width of display area) × 100. Large bitmap detection checks whether the decoded image size exceeds its display area. For example, a value of 150 indicates that when the image over-limit rate > 150, it is considered an abnormal situation and should be reported.
Native Memory Peak Monitoring	"native_memory"	"sample_ratio"	Device sampling rate, with a value range of [0,1]. • 1 indicates enable all • 0 indicates disable all • Otherwise, enable with a specified probability.
		"event_sample_ratio"	Event Sampling Rate, with a value range of [0,1], controls the sampling rate for dumping memory and reporting data when memory usage reaches the peak threshold (VSS or PSS peak).
FD Peak Monitoring	"fd_leak"	"sample_ratio"	Device sampling rate, with a value range of [0,1]. • 1 indicates enable all • 0 indicates disable all • Otherwise, enable with a specified probability.
		"event_sample_ratio"	Event Sampling Rate, with a value range of [0,1], controls the probability of reporting events when FD reaches its peak threshold.

		"threshold"	The threshold for determining FD peak, representing the number of FDs that reach the peak limit.

Java Memory Leak Monitoring Configuration Instructions:

The differences between "sample_ratio" and "event_sample_ratio" for "activity_leak" are as follows:

- "sample_ratio": Indicates the proportion of devices allowed to monitor the lifecycle of Activity/Fragment. It invokes startInspectLeakObj when Activity executes onDestroy, or when Fragment executes onFragmentDestroyed or onFragmentViewDestroyed.
- "event_sample_ratio": Indicates the probability of enabling the Java object leak detection process when startInspectLeakObj is invoked (whether triggered by the application or automatically monitored).

Process Exit Analysis

Monitoring Items	Configuration Item	Configuration Parameter	Parameter Description
Process Exit Analysis	"application_exit"	"sample_ratio"	Device sampling rate, with a value range of [0,1]. • 1 indicates enable all • 0 indicates disable all • Otherwise, enable with a specified probability.
		"event_sample_ratio"	Attachment Sampling Rate, with a value range of [0,1], indicates that when tombs attachments or trace attachments exist, they are reported with a specified probability.

Traffic Monitoring

Monitoring Items	Configuration Item	Configuration Parameter	Parameter Description
Traffic Abnormal Analysis	"traffic_detail"	"sample_ratio"	Device sampling rate, with a value range of [0,1]. • 1 indicates enable all • 0 indicates disable all • Otherwise, enable with a specified probability.
		"total_limit_in_megabyte"	The traffic consumed within 10 minutes exceeds the configured threshold for abnormal traffic, in

			MB.
		"mobile_limit_in_megabyte"	The mobile traffic consumed within 10 minutes has exceeded the configured threshold for abnormal mobile traffic, in MB.
		"backend_limit_in_megabyte"	The traffic consumed in the background scenario within 10 minutes exceeds the configured threshold for abnormal background traffic, in MB.
		"custom_limit_in_megabyte"	The traffic consumed in the business-defined custom scenario within 10 minutes exceeds the configured threshold, in MB.
		"filter_local_address"	Whether to filter local addresses in abnormal traffic monitoring, which takes a Boolean value.
		"metric_event_sample_ratio"	10-minute traffic metrics event sampling rate, with a value range of [0,1]
		"error_event_sample_ratio"	10-minute traffic abnormal events sampling rate, with a value range of [0,1]
traffic metrics	"traffic"	"sample_ratio"	Device sampling rate, with a value range of [0,1]. • 1 indicates enable all • 0 indicates disable all • Otherwise, enable with a specified probability.

Battery monitoring

Monitoring Items	Configuration Item	Configuration Parameter	Parameter Description
foreground battery metrics	"battery_metric"	"sample_ratio"	Device sampling rate, with a value range of [0,1]. • 1 indicates enable all • 0 indicates disable all • Otherwise, enable with a specified probability.
		"daily_report_limit"	Maximum daily report count for monitoring items
Battery consumption factor metrics	"battery_ele_metric"	"sample_ratio"	Device sampling rate, with a value range of [0,1]. • 1 indicates enable all

			• 0 indicates disable all • Otherwise, enable with a specified probability.
		"daily_report_limit"	Maximum daily report count for monitoring items
Abnormal cases of battery consumption factors	"battery_element"	"sample_ratio"	Device sampling rate, with a value range of [0,1]. • 1 indicates enable all • 0 indicates disable all • Otherwise, enable with a specified probability.
		"daily_report_limit"	Maximum daily report count for monitoring items
		"single_location_duration_in_ms"	Maximum duration for single GPS use, in ms
		"total_location_duration_in_ms"	Total GPS update duration within 10 minutes, in ms
		"max_location_open_num"	Maximum count of GPS activations within 10 minutes
		"single_wakelock_duration_in_ms"	Maximum duration for single WakeLock use, in ms
		"total_wakelock_duration_in_ms"	Maximum total WakeLock usage duration within 10 minutes, in ms
		"max_wakelock_open_num"	Maximum count of WakeLock activations within 10 minutes
		"max_alarm_open_num"	Maximum count of alarm activations within 10 minutes
		"max_wakeup_alarm_open_num"	Maximum count of wakeup-type alarm activations within 10 minutes

Network Monitoring

Monitoring Items	Configuration Item	Configuration Parameter	Parameter Description

Network Quality Monitoring	"net_quality"	"sample_ratio"	Device sampling rate, with a value range of [0,1]. • 1 indicates enable all • 0 indicates disable all • Otherwise, enable with a specified probability.
		"daily_report_limit"	Maximum daily report count for monitoring items
		"max_batch_count"	One report can contain at most [X] request detail entries.
		"min_batch_count"	One report must contain at least [X] request detail entries.

ASan Memory Error Detection

Monitoring Items	Configuration Item	Configuration Parameter	Parameter Description
ASan Memory Error Detection	"asan"	"sample_ratio"	Device sampling rate, with a value range of [0,1]. • 1 indicates enable all • 0 indicates disable all • Otherwise, enable with a specified probability.
		"slot_size"	Allocatable memory Slot Page size, in KB. The larger the size, the greater the memory overhead. It is recommended to use the default value.
		"slot_count"	Allocatable memory Slot Page count. The more pages, the more memory allocated for error detection, but the greater the memory overhead. It is recommended to use the default value.
		"max_sample_gap_count"	The maximum value for cyclic sampling of random numbers. The higher the value, the lower the probability of sampling hits during memory allocation.

		"left_side_align_percentage"	The probability of left-aligned memory allocation in the ASan Memory Pool, ranging from [0,100]. A lower value makes it easier to detect Overflow issues, while a higher value facilitates Underflow detection.
		"right_side_perfect_align"	Whether to forgo page alignment for right-aligned memory allocations in the ASan Memory Pool. Disabling page alignment facilitates Overflow detection but may introduce potential crash risks.
		"ignore_overlapped_reading"	Whether to ignore ASan errors of the repeated access type.
		"target_so_patterns"	The SO files to be detected. If left empty, all SO files are detected by default, supporting regular expressions. When memory issues can be traced to specific SOs, it is recommended to set this value to reduce stability problems.
		"ignore_so_patterns"	The SO files to be ignored. If not empty, ignore no SO files by default, supporting regular expressions.

Page loading time

Monitoring Items	Configuration Item	Configuration Parameter	Parameter Description
Page loading time	"page_launch"	"sample_ratio"	Device sampling rate, with a value range of [0,1]. • 1 indicates enable all • 0 indicates disable all • Otherwise, enable with a specified probability.

iOS configuration items

- Crash data is reported at 100% and does not support sampling. All other monitoring capabilities support sampling, but they are disabled by default.
- To modify the configuration switch and specify the sampling rate for monitoring items, go to RUM Pro > Setting > SDK configuration.

- During the integration period, full sampling is recommended—adjust all sampling rates to 1.0 to facilitate data reporting verification. After integration is complete, adjust the sampling rates as needed. Alternatively, create multiple configuration tasks based on the edition type (corresponding to the initialized buglyconfig.buildConfig), configuring development tasks, canary tasks, and production tasks. Enable all monitoring items for development tasks, perform on-demand sampling during the canary phase, and reduce sampling in production releases according to requirements.

Lag Metrics Monitoring

Field	Type	Description	Default Value	Example	Supported Version
suspend_threshold_ms	int	Suspension rate threshold	200	200	–
scroll_fluency_enable	bool	Swiping fps reporting	false	true	–
scroll_fluency_beacon_report_sample	float	Swiping fps Lighthouse reporting sampling, 0–1, precision 0.0001	0	0.1	–
fluency_beacon_report_sample	float	fps&Suspension rate Lighthouse reporting sampling, 0–1, precision 0.0001	0	0.1	–

Lag Issue Monitoring

Field	Type	Description	Default Value	Example	Supported Version
stuck_threshold_ms	int	Lag threshold, must be greater than 0.	200	200	–

optimization_backtrace_ratio	float	Stack trace sampling ratio, range [0, 1], precision 0.0001. Takes effect only when lag stack trace capturing is enabled.	0	0.5	(2.7.45, 2.7.47]
skip_first_frame	bool	Whether to enable first-frame stack trace optimization. When enabled, the first two frames are skipped, and stack capturing begins from the third frame. Value: 0 or 1	0	0	(2.7.45, 2.7.47]

Lag Issues – Worker Thread Monitoring

Field	Type	Description	Default Value	Example	Supported Version
threads	array	Sub-thread configuration group	Refer to the field meaning descriptions below	–	version 2.7.29 and above

Description of each sub-configuration field in threads:

Field	Type	Description	Default Value	Example	Supported Version
stuck_threshold_ms	int	Lag threshold, must be greater than 0.	200	200	version 2.7.29 and above
sample_ratio	float	Enable sampling rate, with a value range of 0 – 1.	1	1	version 2.7.29 and above
enabled	bool	Whether to enable sub-threads	true	true	version 2.7.29 and above
thread_name	string	thread name	–	–	version 2.7.29 and above

ANR monitoring

Field	Type	Description	Default Value	Exam	Supported

				ple	Version
deadlock_threshold_ms	int	Freeze threshold, must be greater than 0.	5000	5000	-
backtrace_mode	int	Stutter stack sampling mode: • 0:CThreadStack • 1: Preliminarily optimized stack sampling logic • 2: Secondarily optimized stack sampling logic	0	1	version 2.7.47 and above
backtrace_mode_ratio	float	Stack trace sampling ratio, range [0, 1], precision 0.0001. Takes effect only when lag stack trace capturing is enabled and backtrace_mode is configured. Local Sampling	0	0.4	version 2.7.47 and above
skip_first_frame	bool	Whether to enable first-frame stack trace optimization. After enabling, it will skip the first two frames and launch stack capturing from the third frame. Boolean value, 0 or 1	0	1	version 2.7.47 and above
backtrace_frequency	int	Stack sampling frequency: captures stack every x frames. High-end devices: 1, Low-end devices: 3. Recommended values	3	3	version 2.7.49 and above

Launch stack capture

Slow-launch stack sampling. When enabled, stack sampling is triggered if the launch duration exceeds launch_backtrace_threshold_ms. If the duration exceeds launch_deadlock_threshold_ms, the launch is considered abnormal, and data is reported to the lag module. Filtering can be performed using the custom field "BuglyAppState" with the value "coldLaunch".

Field	Type	Description	Default Value	Example	Supported Version
sample_ratio	float	launch abnormal monitoring sampling rate	0	0.4	version 2.7.49 and above

launch_deadlock_threshold_ms	int	Slow-launch threshold, must be greater than 0.	5000	5000	version 2.7.49 and above
launch_backtrace_threshold_ms	int	Threshold for launch stack sampling, must be greater than 0.	3000	3000	version 2.7.49 and above

Memory Metrics Monitoring

Field	Type	Description	Default Value	Example	Supported Version
vmmap	bool	vmmap information collection switch	true	false	[2.7.5 – 2.7.21], deprecated after 2.7.21, changed to frequency control.
vmmap_update_interval	int	vmmap information collection interval in seconds, where 0 indicates disabled	3	3	version 2.7.21 and above
strict_foom_judgment	bool	foom strict mode switch	true	true	version 2.7.49-beta.4 and above
app_life_cycle_swizzle	bool	Whether to enable hook app delegate switch	true	true	version 2.7.50-beta and above
vmmap_with_malloc_zone	bool	Whether vmmap information collection includes malloc zone information	false	false	version 2.7.51 and above
memory_check_interval	int	memory information update frequency interval	1000	1000	version 2.7.53 and above

FOOM monitoring

Field	Type	Description	Default Value	Exam	Supported Version

				pl e	
malloc_stack_enable_sa mple_ratio	flo at	foom sampling ratio, 0–1, SDK random sampling, precision 0.0001	0	0. 1	–
backtrace_recording_en able	bo ol	Enable asynchronous call stack recording, which incurs some performance overhead and is disabled by default.	false	fal se	version 2.7.45 and above
malloc_stack_storage_t hreshold	int	The minimum threshold for storing local memory stack records in cache; when the cumulative allocation of a single stack reaches this value, it will be recorded.	5242 88	52 42 88	–
malloc_size_record_thre sholds	arr ay	The array of size threshold ranges for local memory stack records; stack records are captured when the size of memory allocated by the user falls within one of these ranges.	Refer to the field mean ing descr iption s below	–	version 2.7.17 and above

Description of each sub-configuration field in malloc_size_record_thresholds

Field	Typ e	Description	Default Value	Examp le	Suppor ted Version
min	long	The minimum value of a single memory allocation that needs to be recorded, representing the lower bound of the threshold range (inclusive of the min value).	8192	8192	version 2.7.17 and above
max	long	The maximum value of a single memory allocation that needs to be recorded,	LONG_MAX	112589	version 2.7.17

		representing the upper bound of the threshold range (exclusive of the max value).			and above
stack_cache_max_count	long	The maximum number of stack records	300000	300000	version 2.7.17 and above

VC Memory Leak Monitoring

Field	Type	Description	Default Value	Example	Supported Version
filter_list	array	The allowlist, where VCs in the list are not monitored.	[]	["BuglyViewController"]	version 2.7.51 and above

Large Memory Allocation Monitoring

Field	Type	Description	Default Value	Example	Supported Version
chuck_threshold	int	The threshold for identifying large memory allocations	ULONG_MAX	10485760	-

Memory Details Monitoring

Field	Type	Description	Default Value	Example	Supported Version
dump_threshold	int	Memory map dump threshold: when physical memory usage reaches this value, it triggers the memory map dump operation.	ULONG_MAX	4294967296	-
dump_threshold_rate	float	The memory map dump threshold ratio: when this value is set, the product of the device's physical memory value and this ratio will be	0	0.4	version 2.7.39.2 and above

		used as the threshold for triggering memory map dumps.			
dump_type	string	The memory map dump type. Optional values include DumpFull, DumpLight, and DumpTiny. Any value not listed above is considered DumpNone.	DumpFull	DumpTiny	version 2.7.39.2 and above

Note:

- DumpFull includes memory map data with reference relationships, and the Dump operation consumes more resources.
- DumpLight: excludes reference relationships, only retains type information, and the Dump operation has lower resource consumption.
- DumpTiny: only retains size information, and the Dump operation has minimal resource consumption.

Crash Monitoring

The bugly crash feature switch has no special fields and is enabled by default if not configured. Supported in version 2.7.49 and above.

Field	Type	Description	Default Value	Example	Supported Version
use_jce_report	bool	Whether json reporting is used	0	0	version 2.7.50 and above
disable_unhandled_crash	bool	Whether reporting custom crash stacks is supported	1	0	version 2.7.51 and above

Error Monitoring

The custom error reporting switch is enabled by default if not configured. Supported in version 2.7.49 and above.

Field	Type	Description	Default Value	Example	Supported Version
use_jce_report	bool	Whether json	0	0	version 2.7.50

		reporting is used			and above

Crash zombie detection

The zombie detection capability switch is supported in version 2.7.1 and above.

Crash Custom Attachment Upload Control

The custom crash file upload feature is disabled by default. Supported in version 2.7.39 and above.

Crash minidump collection control

The crash minidump attachment information feature, when enabled, collects minidump information during a crash and reports it upon the second launch, disabled by default. Supported in version 2.7.55 and above.

Field	Type	Description	Default Value	Example	Supported Version
minidump_file_size	int	The size of the minidump mmap file	256 * 1024	256 * 1024	version 2.7.55 and above

Crash public attachment upload control

The crash common_file public attachment upload capability allows the client to upload attachments with different filenames for each event, enabled by default. Supported in version 2.7.55 and above.

Launch Metrics Collection Control

The launch metrics configuration controls whether background–to–foreground metrics are reported, disabled by default. Cold launch are not affected.

Field	Type	Description	Default Value	Example	Supported Version
enabled	bool	The switch returned by the backend does not require configuration.	0	0	(–, 2.7.49)
application_resume_beacon_report_sample	float	Sampling Configuration for Reporting Background–to–Foreground Time Consumption in Lighthouse	0	0.1	(–, 2.7.49)

cold_launch_beacon_report_sample	float	Cold launch Lighthouse reporting sampling switch	0	0.1	(–, 2.7.49)

Cold launch metrics collection control

The configuration for cold launch metrics is enabled by default when unconfigured.

Field	Type	Description	Default Value	Example	Supported Version
enabled	bool	The switch returned by the backend does not require configuration.	1	0	version 2.7.49 and above
cold_launch_beacon_report_sample	float	Cold launch Lighthouse reporting sampling switch	0	0.1	version 2.7.49 and above

Background–to–Foreground Metric Collection Control

The configuration for background–to–foreground metrics is enabled by default when unconfigured.

Field	Type	Description	Default Value	Example	Supported Version
enabled	bool	The switch returned by the backend does not require configuration.	1	0	version 2.7.49 and above
application_resume_beacon_report_sample	float	Sampling Configuration for Reporting Background–to–Foreground Time Consumption in Lighthouse	0	0.1	version 2.7.49 and above

metric kit collection control

Field	Type	Description	Default	Example	Supported Version

			Value		
available_report_types	array	Configuration for allowed metrickit reporting types. Defaults to empty; no filtering is performed if empty or not configured.	null	–	version 2.7.55–beta.6 and above

DAU Collection Control

DAU reporting switch is enabled by default. Supported in version 2.7.49 and above.
Recommended configuration:

```
{
  "name": "common.dau",
  "sample_ratio": 1,
  "daily_report_limit": 10000
}
```

Operation Path Collection Control

The vc operation path reporting switch was added after sdk version 2.7.3 (excluding this version).

Field	Type	Description	Default Value	Example	Supported Version
max_operation_seq_count	uint	Maximum number of records: 50	20	50	version 2.7.3 and above

10-minute traffic collection control

Traffic abnormal monitoring. After enabling, App traffic monitoring will adopt finer granularity, with aggregated reports every 10 minutes. When exceeding specified thresholds, detailed network request information will be reported.

Field	Type	Description	Default Value	Example	Supported Version
total_limit_in_megabyte	uint	Total traffic abnormal threshold (MB)	500	500	version 2.7.51 and above

mobile_limit_in_megabyte	uint	Cellular network abnormal threshold (MB)	200	200	version 2.7.51 and above
background_limit_in_megabyte	uint	Background network abnormal threshold (MB)	50	50	version 2.7.51 and above

Page performance collection control

Page loading time. When enabled, it reports the time cost of opening each UIViewController in the App, and is disabled by default. Supported in version 2.8.0–beta and above.

Harmony configuration item

- Crash data is reported at 100% and does not support sampling. All other monitoring capabilities support sampling, but they are disabled by default.
- To modify the configuration switch and specify the sampling rate for monitoring items, go to RUM Pro > Setting > SDK configuration.
- During the integration period, full sampling is recommended—adjust all sampling rates to 1.0 to facilitate data reporting verification. After integration is complete, adjust the sampling rates as needed. Alternatively, create multiple configuration tasks based on the edition type (corresponding to the initialized buglyconfig.buildConfig), configuring development tasks, canary tasks, and production tasks. Enable all monitoring items for development tasks, perform on-demand sampling during the canary phase, and reduce sampling in production releases according to requirements.

The configuration parameters for each monitoring item are described as follows.

Quality Monitoring

Monitoring Items	Configuration Item	Configuration Parameter	Parameter Description
Error Reporting	"error_report"	"sample_ratio"	Error Reporting sampling rate, with a value range of [0,1]. • 1 indicates report all • 0 indicates do not report • Other indicates user sampling reporting

Lag Monitoring

Monitoring	Configuration Item	Configuration Parameter	Parameter Description

Items			
Lag Metrics	"looper_metric"	"sample_ratio"	Device sampling rate, with a value range of [0,1]. • 1 indicates enable all • 0 indicates disable all • Otherwise, enable with a specified probability.
		"daily_report_limit"	Device daily reporting limit for Lag Metrics, default value 300 times
		"loop_time_out_threshold"	Setting the threshold for timeout messages in Lag Metrics statistics, default value 150 ms
Lag Instances	"looper_stack"	"sample_ratio"	Device sampling rate, with a value range of [0,1]. • 1 indicates enable all • 0 indicates disable all • Otherwise, enable with a specified probability.

Symbol Table Overview

Last updated：2026-05-25 18:53:27

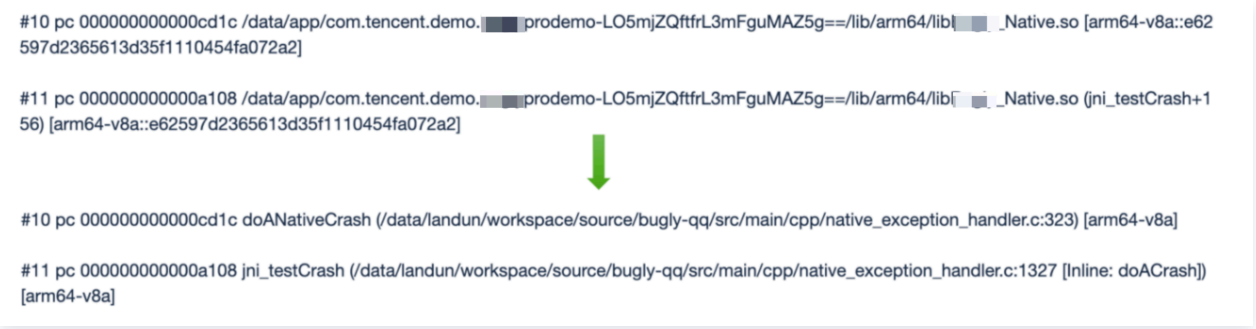
symbol table

Definition

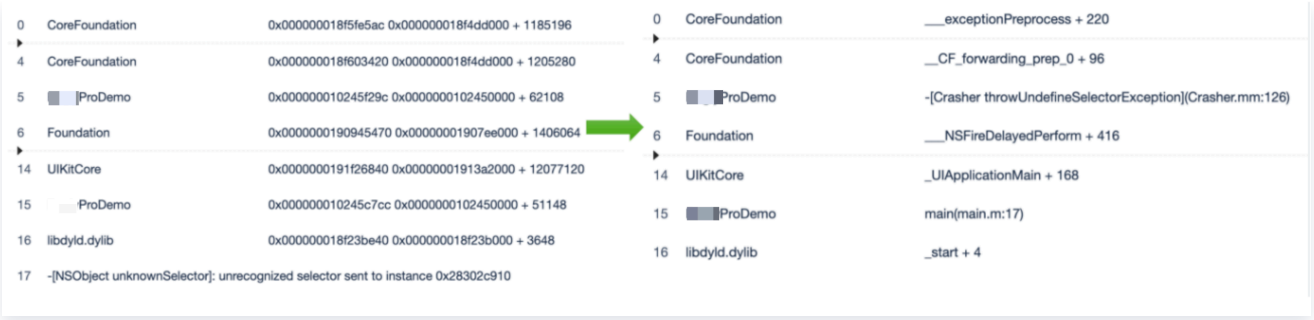
A symbol table has slightly different meanings across platforms, but its purpose is always to restore obfuscated stack traces to human-readable file names, method names, and line numbers.

- On the Android platform, there are two types of symbol tables: shared object (SO) symbol tables and Java mapping files. For detailed explanations, see Android Symbol Tables.
  - Android Java Stack Deobfuscation:

  - Android Native Stack Deobfuscation:



- On the iOS platform, a symbol table refers to a dSYM file. For detailed explanations, see iOS Symbol Tables. An example of iOS stack trace deobfuscation is as follows:



- On the Harmony platform, there are two types of symbol tables: shared object (SO) symbol tables and nameCache & SourceMaps files. For detailed explanations, see Harmony (Hongmeng) Symbol Tables.

Distinguishing Symbol Tables

During deobfuscation, the required symbol table for restoration can be determined by the context of the exception stack trace:

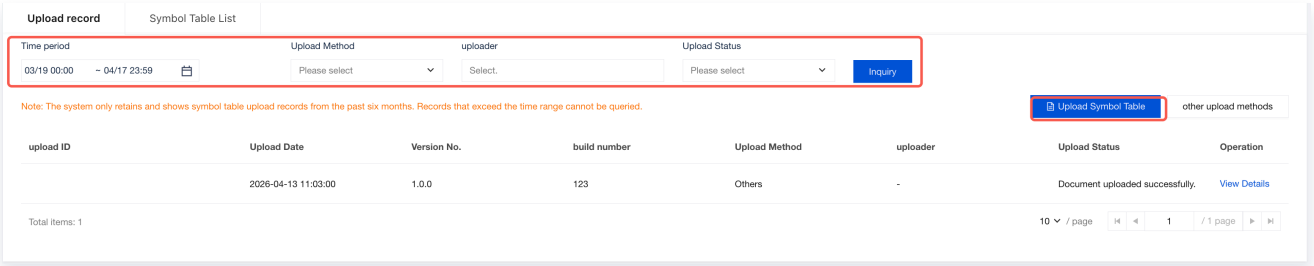
Platform	Type	Description
Android Platform	Android Java Stack Trace	• The symbol table file is the mapping file. • Locate the symbol table file via the App version and build number. • For the same App version and build, multiple mapping files can be uploaded simultaneously and distinguished by their filenames. • During symbol deobfuscation, eligible mapping files are identified by the App version and build number. These mapping files are then combined into a single comprehensive mapping before translation.
	Android Native Stack Trace	• The symbol table is an SO containing symbol information. • Symbol tables can be located using the UUID of the SO, meaning that during the same build, both the SO containing symbol tables and the SO with symbol tables removed can be generated, and they share the same UUID. • App version, build number, CPU architecture, and module name are merely for enhanced readability, while the SO's UUID serves as the unique identifier.

iOS Platform	iOS Stack Trace	• The symbol table file is the dSYM file. • Locate the symbol table via UUID. • App version, build number, CPU architecture, and module name are merely for enhanced readability, while the UUID serves as the unique identifier.
Harmony Platform	JS Stack Trace	• The symbol table file is the SourceMap file. • Locate the symbol table file via the App version and build number. • For the same App version and build, multiple SourceMap files can be uploaded simultaneously and distinguished by their filenames. • During symbol deobfuscation, eligible SourceMap files are identified by the App version and build number. All these files are then combined into a single comprehensive SourceMap before the translation process.

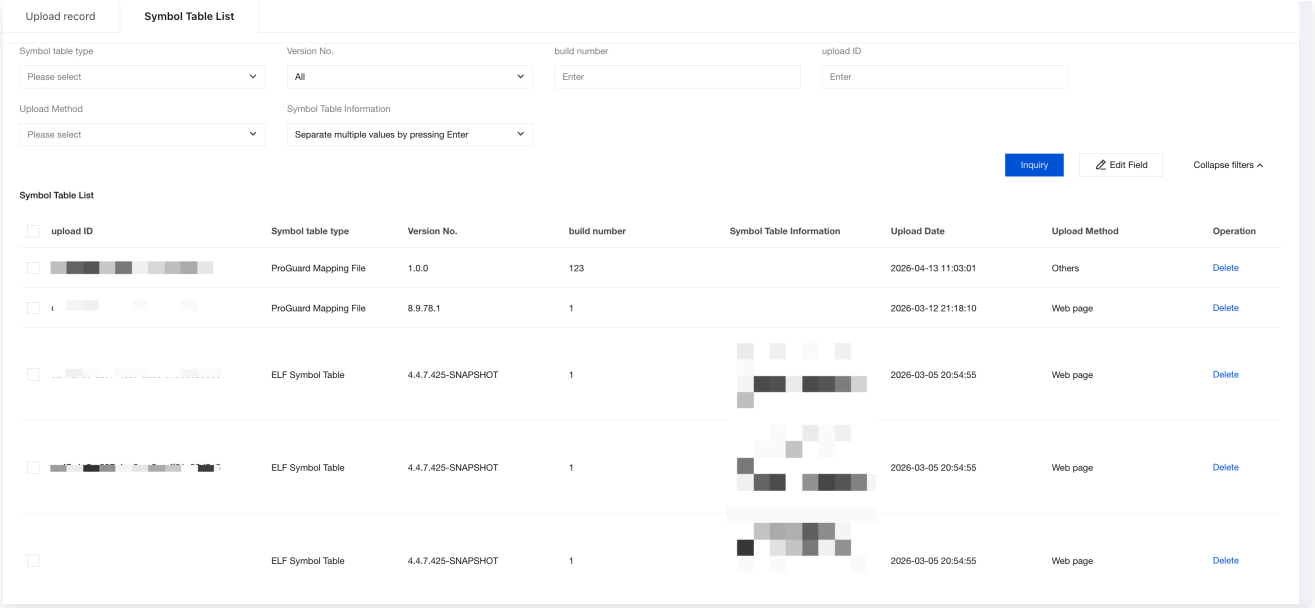
Manage symbol tables

Users can view the symbol table information for the current product under Setting > Symbol Table.

1. Check whether the specified symbol table has been uploaded by applying filter criteria.
2. Upload the symbol table; for details, see Upload the symbol table.
3. Click **View Details** to view the overall status of the product symbol table.



Users can query uploaded symbol table information in the **Symbol Table List** Tab using filter criteria such as Symbol table type, Version No., build number, upload ID, Upload Method, and Symbol Table Infomation.

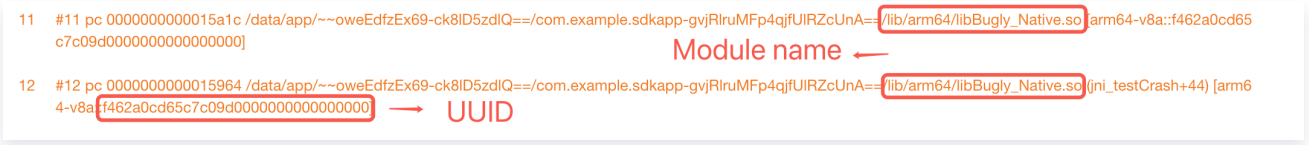


UUID

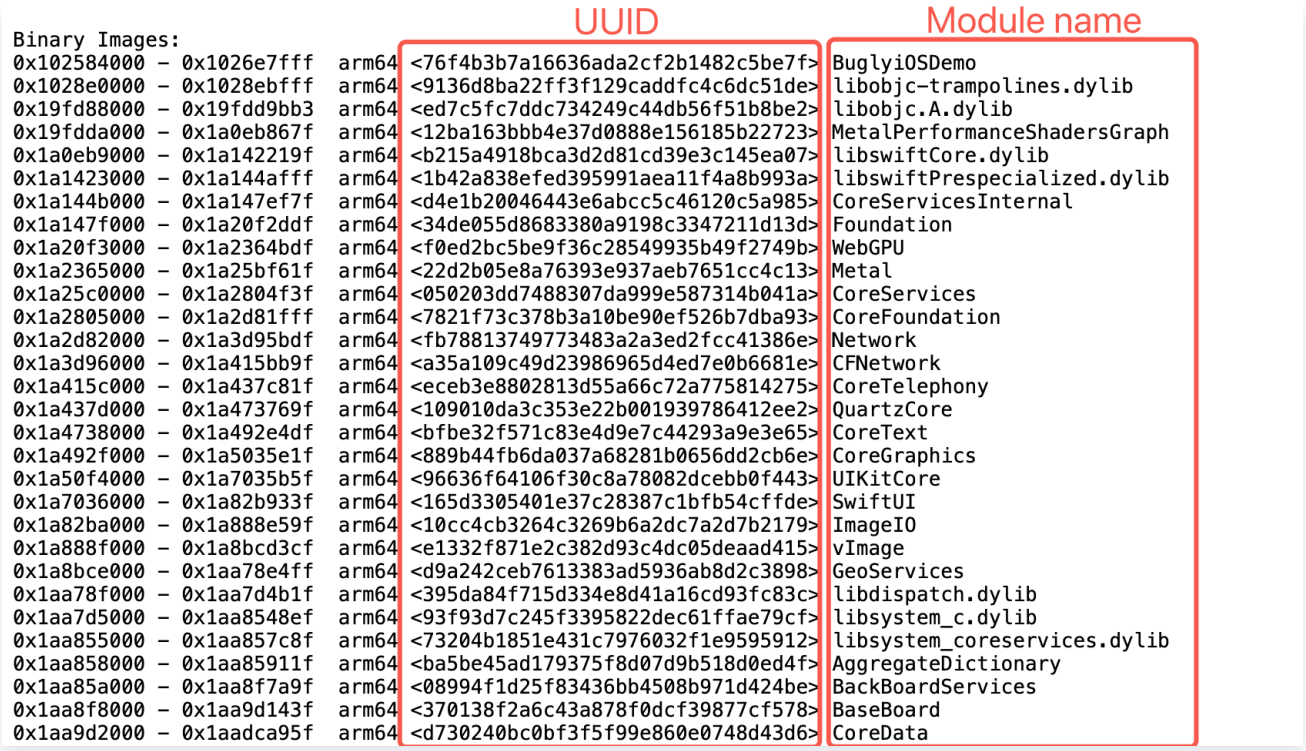
Definition

The UUID is used to uniquely identify a module. Both Android's SO symbol tables and iOS's dSYM files use UUIDs to uniquely identify a symbol table file. After capturing an exception stack, the SDK will obtain the UUID of the corresponding module. The symbol deobfuscation platform then uses this UUID to search for the corresponding symbol table file.

- Example 1: The Android Native stack contains UUID information, and the deobfuscation system uses the UUID to search for the symbol table.



- Example 2: In iOS exceptions, the Binary Images section of the crash.log file contains mappings of module address ranges to UUIDs and module names.



UUID Retrieval Guidance

Android SO UUID

There are two methods for generating UUIDs in Android NDK.

- **Generate at compile time**

  Read the **.note.gnu.build-id** Section; if a value is found, use the UUID generated during compilation.

```
readelf -x .note.gnu.build-id libBugly.SO

Hex dump of section '.note.gnu.build-id':
  0x000001c8 04000000 14000000 03000000 474e5500 ............GNU.
  0x000001d8 c21c5005 a226dedc 83a5a5b3 1611a3d6 ..P..&..........
  0x000001e8 27713fd9                             'q?.
```

  Among them:

  - 0x000001c8: indicates the offset address of the .note.gnu.build-id section in the elf file.
  - 04000000: indicates the name size, which corresponds to 4 bytes in decimal.
  - 14000000: indicates the size occupied by the UUID, corresponding to 20 bytes in decimal.
  - 03000000: indicates the type.
  - 474e5500 is 4 bytes, which corresponds to the name GNU.
  - c21c5005 a226dedc 83a5a5b3 1611a3d6 27713fd9 totals 20 bytes, corresponding to the UUID generated during the compilation process. The compiled UUID is 160 bits, while the platform uses 16 bytes (128 bits). If the UUID exceeds 128 bits, the platform discards the first 32 bits. Therefore, the UUID actually obtained by the platform here is a226dedc83a5a5b31611a3d627713fd9, which is the last 16 bytes.

> **Note:**
> The platform uses a 16-byte UUID, specifically the last 16 bytes of the UUID generated during compilation.

- **Generate at runtime**

  When the .note.gnu.build-id section is absent in an elf file, the platform's method of generating a 16-byte UUID by hashing the first 4096 bytes of the .text section may cause conflicts due to minor byte differences. Even expanding the calculation scope to the entire .text section cannot resolve UUID conflicts between two code versions differing only by a few lines of whitespace (where .text content remains identical but compiles to different UUIDs). Moreover, runtime calculation of the entire .text section impacts program efficiency. While developers might disable .note.gnu.build-id generation for compilation efficiency, compilation time is negligible compared to runtime overhead. Therefore, to strictly guarantee UUID uniqueness, the .note.gnu.build-id must be generated during SO build configuration. You can enable .note.gnu.build-id generation during compilation through the following two methods:

  - NDK build: Configure LOCAL_LDFLAGS += -Xlinker --build-id in Android.mk.
  - Clang build: Refer here.

Mach-O UUID

Mach-O UUID is generated during build time and stored in the Mach-O header. There are two methods to query the UUID of a dSYM.

- Read from the command line

```
dwarfdump -u path/to/compile/executable
```

- Read from the code

```
#import <mach-o/ldsyms.h>
NSString *executableUUID()
{
    const uint8_t *command = (const uint8_t *)(&_mh_execute_header +
1);
    for (uint32_t idx = 0; idx < _mh_execute_header.ncmds; ++idx) {
        if (((const struct load_command *)command)->cmd == LC_UUID) {
            command += sizeof(struct load_command);
            return [NSString stringWithFormat:@"%02X%02X%02X%02X-
%02X%02X-%02X%02X-%02X%02X-%02X%02X%02X%02X%02X%02X",
                    command[0], command[1], command[2], command[3],
                    command[4], command[5],
                    command[6], command[7],
                    command[8], command[9],
                    command[10], command[11], command[12],
command[13], command[14], command[15]];
        } else {
            command += ((const struct load_command *)command)-
>cmdsize;
        }
    }
    return nil;
}
```

Under what circumstances will UUIDs be the same

Each binary file in an app—the main app executable, frameworks, and app extensions—has its own dSYM file. The compiled binary and its companion dSYM file are tied together by a build UUID, that's recorded by both the built binary and dSYM file. If you build two binaries from the same source code but with different Xcode versions or build settings, the build UUIDs for the two binaries won't match. A binary and a dSYM file

are only compatible with each other when they have identical build UUIDs. Keep the dSYM files for the specific builds you distribute, and use them when diagnosing issues from crash reports.
The above explanation is referenced from the article Building Your App to Include Debugging Information.
This rule also applies to the Android Native / Linux platform.

Note:
The module UUIDs will be equal only when both conditions are simultaneously met:
- The code is completely identical.
- The build platform configuration is completely identical.

Upload the symbol table

Last updated：2026-05-25 18:53:28

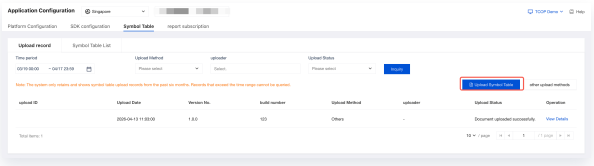
Support uploading symbol tables through two methods: web page upload and symbol table upload tool.

Upload symbol tables via the web page

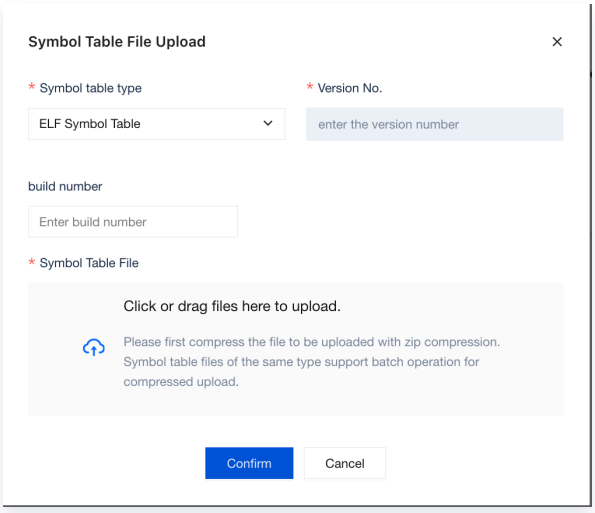
Upload portal

Entry 1: Settings/Symbol Table

1. On the Setting > Symbol Table page, click **Upload Symbol Table**.



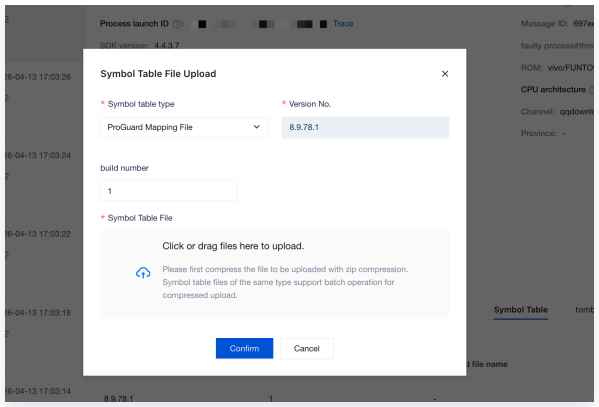
2. In the symbol table upload dialog box, select the symbol table type, Version No., build number, and then upload the symbol table file.



Entry 2: Case Details/Symbol Table

1. When analyzing exception cases, if symbols are found untranslated, you can directly switch to the Symbol Table Tab, click **Upload Pending, click to upload** to quickly upload the symbol table.

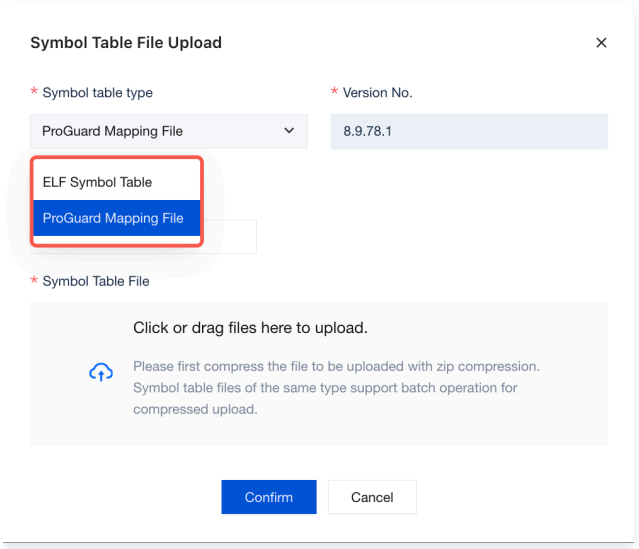
2. When uploading a symbol table on the case details page, the system assumes the user intends to upload the specified symbol table file. It will automatically populate the Version No. and build number. For SO or dSYM files, the UUID will also be auto-filled and validated.



Upload Steps

Upload the Android mapping file

- Compress the mapping file to be uploaded into a mapping.zip file and drag it to the upload area.
- When uploading for the first time, it will prompt that the upload is successful. When uploading again, it will prompt for overwrite upload. If you confirm to overwrite, click **upload continue** to overwrite the file with the same name.



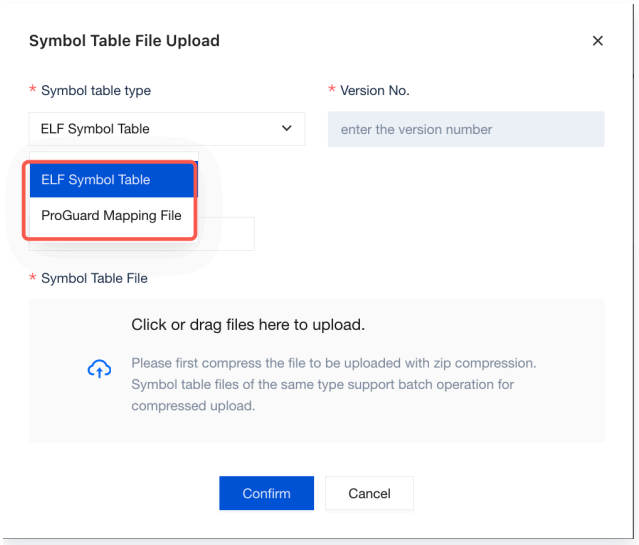
Note:

For mapping files with the same version number and build number, a batch upload feature is currently supported, distinguished by file names. During operation, place multiple mapping files (it is recommended to name the host file as mapping.txt and other component or plugin files as xxx_mapping.txt) in the same directory, compress them into a zip file, and upload. Subsequently, the translation service assembles a large mapping file prioritizing mapping.txt over xxx_mapping.txt. For a single stack trace line, it first uses the host mapping file for translation, then applies the component or plugin mapping file.

Upload Android SO file

Compress the SO symbol table file to be uploaded into a zip file and drag it to the upload area.

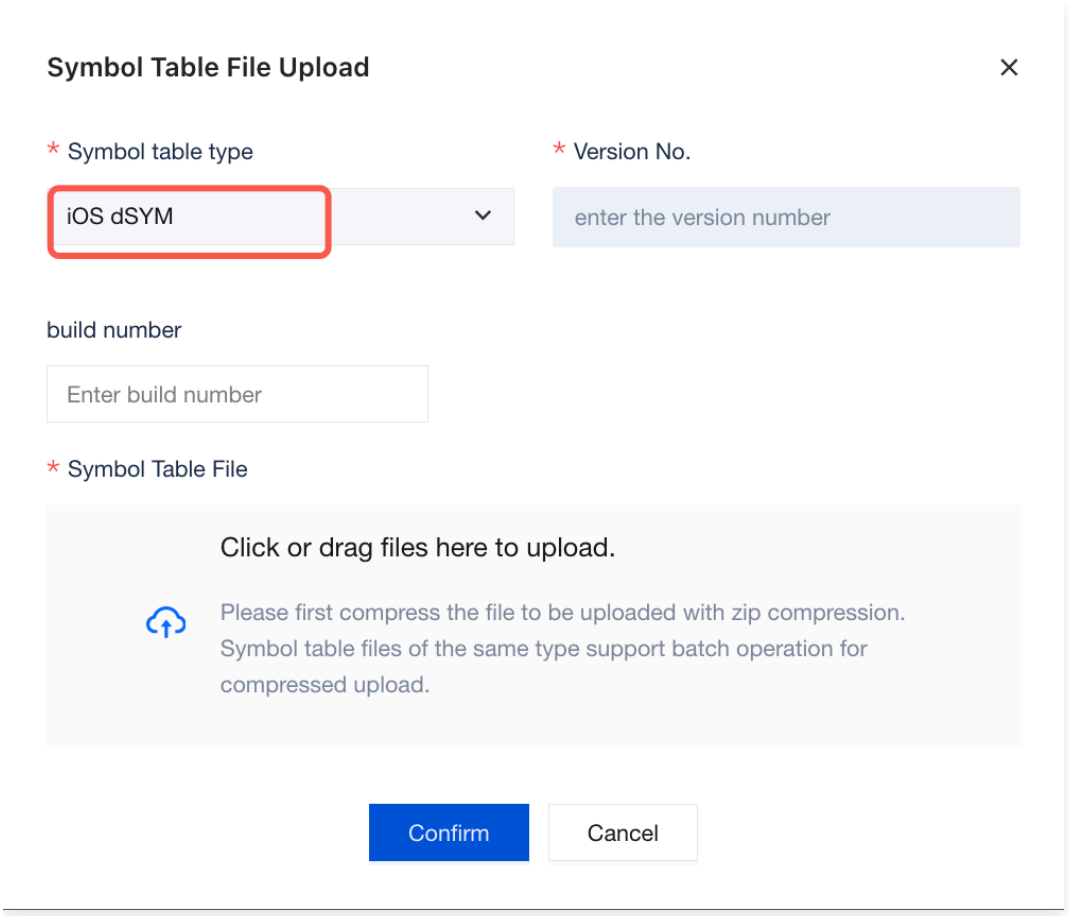
> **Note:**
> For SO symbol table files with the same version number and build number, batch upload is supported. These files are uniquely identified by UUID. If a file with the same UUID is detected as already uploaded, the system will prompt the user whether to overwrite it.



Upload iOS dSYM file

Compress the dSYM file to be uploaded into a zip file and drag it to the upload area.

> **Note:**
> For dSYM files with the same version number and build number, batch upload is supported. These files are uniquely identified by UUID. If a file with the same UUID is detected as already uploaded, the system will prompt the user whether to overwrite it.



Upload the Harmony SO file

The operation is the same as uploading Android SO files.

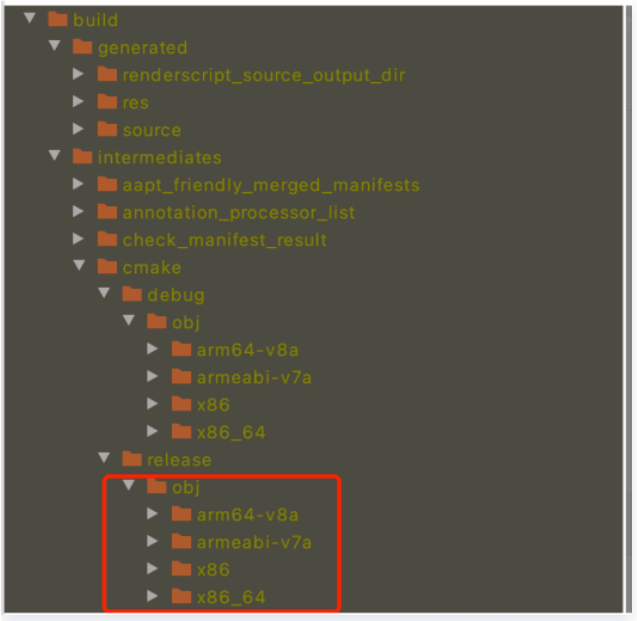
Upload Harmony nameCache & SourceMaps files

The operation is the same as uploading Android mapping files, and supports independent upload of nameCache and SourceMaps files.

Example

Batch upload SO symbol table files

As shown in the figure below, under the **build/intermediates/cmake/release/obj** directory, you can see directories for different CPU architectures. You can directly compress the entire obj directory to obtain the obj.zip file, then drag the obj.zip file directly to the upload area, and click **Confirm** to complete the process. When the file is large, the process may take longer; please wait patiently. Once uploaded successfully, you can view the batch-uploaded files on the web page.



Upload the symbol table for the Android application plugin

Android mapping files locate the symbol table for a specific exception through the App version and build number. The platform's symbol table system supports uploading multiple mapping files for a specified App version and build number (it is recommended to name the host file as mapping.txt and plugin or component files as xxx_mapping.txt). You can compress all files in a directory into a zip file for batch upload, or upload them in batches. If an incorrect mapping file for a plugin is uploaded, it can be overwritten, and the platform will indicate the overwrite status.

> **Important:**
> Android mapping files are uniquely identified by the App version, build number, and file name, regardless of the specific file content. That is, even if the same mapping file is uploaded twice, as long as there is a difference in any of the three elements—App version number, build number, or file name—it will not be recognized as an overwrite upload.

Symbol Table Upload Tool upload

Operation Steps

1. Download the symbol table upload tool. The latest version of the symbol table upload tool is **V3.4.24**. Please click to download the symbol table tool.
2. After downloading the tool, you will see bugly_cloud_sgp_localtool_3.4.24.jar.

3. Prepare the Java runtime environment.

Note:

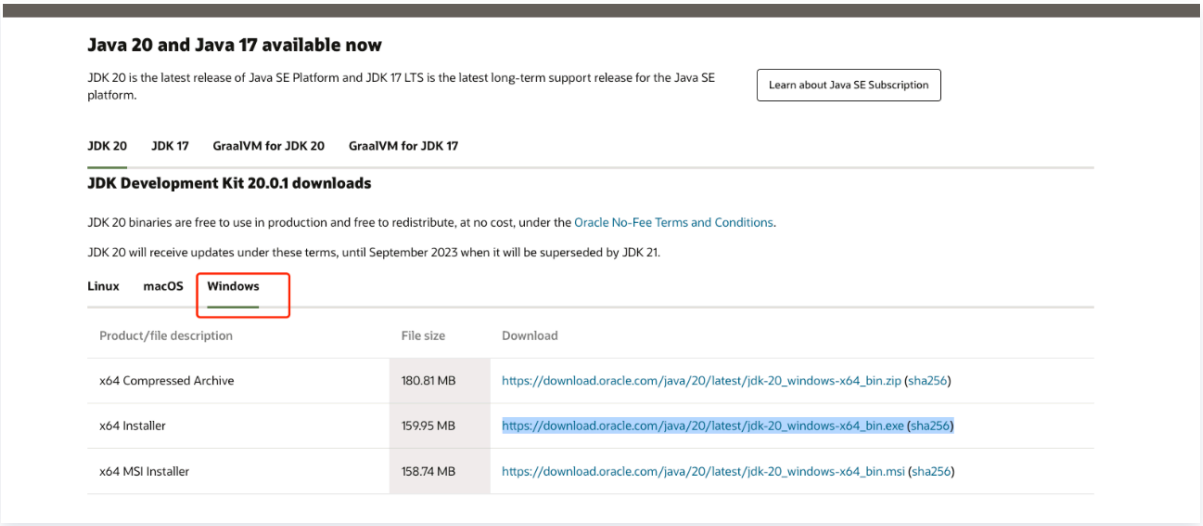
- The symbol table upload tool is a jar toolkit that depends on the Java environment, compiled under JDK version 8, and therefore can run in JRE environments of Java version 8 and above.
- Currently, runtime environments for Java 8, Java 11, Java 17, and Java 19 have been verified on Mac platforms.

Windows platform

The steps to install JRE (Java Runtime Environment) on Windows are as follows.

1. Visit the official Oracle website to download the JRE installer for Windows. Please access the following link and select the JRE installer package suitable for your Windows version (32-bit or 64-bit).

```
https://www.oracle.com/java/technologies/downloads/#jdk20-windows
```



2. After the download is complete, double-click the downloaded installer file (for example: jdk-20_windows-x64_bin.exe) to launch the installation program.
3. In the installation wizard, follow the prompts. Typically, you can use the default settings. Click **Next** to continue the installation process.
4. After the installation is complete, click **Close** to exit the installation wizard.
5. To ensure that JRE is correctly installed and accessible from the command line, open Command Prompt (press the Win key, type "cmd", and press Enter).
6. Enter the following command in Command Prompt and press the Enter key:

```
java -version
```

If the installation is successful, you will see output similar to the following, displaying the installed Java version:

```
java version "20.0.1"
Java(TM) SE Runtime Environment (build 20.0.1+9-29)
Java HotSpot(TM) 64-Bit Server VM (build 20.0.1+9-29, mixed mode)
```

Now, you have successfully installed JRE on Windows.

Linux platform

The steps to install JRE (Java Runtime Environment) on Linux platforms are as follows. The following uses Ubuntu as an example:

1. Open the Terminal (shortcut: Ctrl + Alt + T).
2. Update the package list.

   ```
   sudo apt update
   ```

3. Install the default Java Runtime Environment (typically the latest version of OpenJDK).

   ```
   sudo apt install default-jre
   ```

   Enter your password and press Enter, then press the "Y" key when prompted to confirm the installation.
4. After the installation is complete, verify whether JRE has been successfully installed. Enter the following command in the Terminal and press the Enter key:

   ```
   java -version
   ```

   If the installation is successful, you will see output similar to the following, displaying the installed Java version.

   ```
   openjdk version "11.0.11" 2021-04-20
   OpenJDK Runtime Environment (build 11.0.11+9-Ubuntu-
   0ubuntu2.20.04)
   OpenJDK 64-Bit Server VM (build 11.0.11+9-Ubuntu-0ubuntu2.20.04,
   mixed mode, sharing)
   ```

Now, you have successfully installed JRE on the Linux (Ubuntu) platform.

> **Note:**
> Here, the OpenJDK version of JRE is installed. If you require Oracle JRE, visit the official Oracle website to download the JRE installation package for Linux and follow the official documentation for installation.

Mac platform

To install multiple Java versions on Mac, you can use the package manager Homebrew. The following are the steps to install and manage multiple Java versions.

1. Install Homebrew (if not already installed) by opening the Terminal and running the following command.

```
/bin/bash -c "$(curl -fsSL
https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh
)"
```

   After the installation is complete, the Terminal will display corresponding prompts.
2. Add Homebrew's cask version repository.

```
brew tap homebrew/cask-versions
```

3. Install the required Java versions. For example, if you need to install Java 8 and Java 11, run the following commands:

```
brew install --cask adoptopenjdk8
brew install --cask adoptopenjdk11
```

   You can install other Java versions by replacing the number.
4. Check the installed Java versions by running the following command.

```
/usr/libexec/java_home -V
```

   This will display the installed Java versions and their paths.

5. To switch between Java versions, you can set the JAVA_HOME environment variable. For example, to switch to Java 8, run the following command:

```
export JAVA_HOME=$(/usr/libexec/java_home -v 1.8)
```

To switch to Java 11, run:

```
export JAVA_HOME=$(/usr/libexec/java_home -v 11)
```

You can switch to other Java versions by replacing the number.

6. To make the switch persistent, add the above export command to your shell configuration file (such as .bashrc, .bash_profile, or .zshrc). This way, the JAVA_HOME environment variable will be automatically set whenever you open a new terminal window. Now, you have installed multiple Java versions on your Mac and can switch between them as needed.
7. You can check the current Java version via the following command.

```
java -version
```

If the installation is successful, you will see output similar to the following, displaying the installed Java version.

```
openjdk version "1.8.0_282"
OpenJDK Runtime Environment (AdoptOpenJDK)(build 1.8.0_282-b08)
OpenJDK 64-Bit Server VM (AdoptOpenJDK)(build 25.282-b08, mixed
mode)
```

4. Files can be uploaded using tools.

The command format is as follows:

```
java -jar rumpro-upload-symbol.jar -appid <App ID>
                                   -appkey <App KEY>
                                   -version <App Version or SO
Version>
                                   -buildNo <App Build Number>
                                   -platform <Platform>
```

```
                                    -inputSymbol <Original Symbol File
Path>
                                    -inputMapping <mapping.txt file
Path>
```

The parameter description is as follows:

Parameter	Required	Description
appId	Yes	The unique identifier assigned to the product by the platform can be viewed in the **application list**.
appkey	Yes	The identifier assigned to the product by the platform can be viewed in the **application list**.
version	Yes	The version number. When uploading a Java mapping.txt file, it refers to the corresponding App version number; when uploading an SO or dSYM file, it refers to the version number of the corresponding SO or dSYM. **Note:** • Avoid including special characters such as ( ) in the version number, otherwise it may throw an error during execution. • If a mapping file is reported, the version number here must match the actual version number of the App to which the stack to be restored belongs. This is because each App version corresponds to a unique mapping.txt file, and the stack cannot be restored if they do not match. • If you are only uploading SO symbol tables, it is recommended to fill in the version number corresponding to the SO for easy identification during symbol table review. Regardless of what you enter, it does not affect symbol resolution, as the SO file is uniquely identified by its UUID. • If you are only uploading dSYM symbol tables, it is recommended to fill in the version number corresponding to the dSYM for easy identification during symbol table review. Regardless of what you enter, it does not affect symbol resolution, as the dSYM file is uniquely identified by its UUID.

buildNo	No	The build number. If you are uploading a mapping.txt file and have used a build number during SDK initialization, you must provide the correct build number when uploading the symbol table; otherwise, it will result in the failure to restore Java stack traces.
platform	Yes	Platform type. Currently supported options are Android, iOS, and Mac. **Note: Case sensitivity must be correct**.
inputSymbol	Yes	The directory path where the original symbol tables [dSYM, SO] are located. For Android platforms containing both mapping and SO files, enter the common parent directory storing both original symbol tables here, or specify the mapping file path via inputMapping.
inputMapping	Yes	The directory path where the mapping.txt file for the Android platform is located. **Specific to Android platform, ignore for iOS**.

Example

Case 1: Upload an SO symbol table file

Reference command line:

```
java -jar rumpro-upload-symbol.jar -appid a278f01047 -appkey 1e5ab6b3-b6fa-4f9b-a3c2-743d31dffe86  -version 4.3.0 -platform Android -inputSymbol /Users/lilyxie/Downloads/upload_target/obj/arm64-v8a/libbugly_dumper.SO
```

Results Summary:

```
...
##[info]args is ArgsParser{appId='a278f01047'appKey='1e5ab6b3-b6fa-4f9b-a3c2-743d31dffe86', appPackage='com.tencent.demo.rumprodemo', appVersion='4.3.0', appBuildNo='null', platformId=Android', enviroment=null', symbolPathName='/Users/lilyxie/Downloads/upload_target/obj/arm64-v8a/libbugly_dumper.SO', mappingPathName='null', appPathName='null'}
...
##[info]request jSOn is {"appID":"a278f01047","authSign":"1e5ab6b3-b6fa-4f9b-a3c2-743d31dffe86","appVersion":"4.3.0","appBundleID":"com.tencent.demo.rumprodemo","appPlatform":1,"appBuildNumber":null,"fileType":201,"fileSize":83101,
```

```
"fileMD5":"7237e936f6bf7e8ed52a70f67c02aa31","clientType":7,"clientVersi
on":"1.0.48","buildPlatform":1,
"buildID":null,"buildName":"null&null","fileInfoList":
[{"name":"buglySymbol&arm64-
v8a&libbugly_dumper&a5cd2db99a1f45b45469979205f09730.zip",
"UUID":"a5cd2db99a1f45b45469979205f09730","arch":"arm64-
v8a","type":101,"moduleName":"libbugly_dumper"}],
"buildRepo":null,"buildBranch":null,"buildCommitID":null}
##[info]request upload Info Url is https://symbol-
v2.bugly.qq.com/trpc.eff_tool.symbol_upload_gateway.SymbolUploadGateway/
uploadInfo
##[info]request gitRepoUrl is null
##[info]request gitRepoBranch is null
##[info]request gitRepoHeadCommitId is null
##[info]envtype is null
##[info]getAppModuleList is null
##[info]retCode: 200 response message:
{"statusCode":0,"msg":"success","uploadReqID":"a278f01047-2e0f441c-da63-
46a3-b03b-6b5ce1bdb008"}
##[info]now begin to uploadFileContent
##[info]request uploadFileurl is https://symbol-
v2.bugly.qq.com/trpc.eff_tool.symbol_upload_gateway.SymbolUploadGateway/
uploadFile
##[info]retCode: 200 response message:
{"statusCode":0,"msg":"success","uploadReqID":"a278f01047-2e0f441c-da63-
46a3-b03b-6b5ce1bdb008"}
```

Example 2: Uploading multiple SO symbol table files

Reference command line:

```
java -jar rumpro-upload-symbol.jar -appid a278f01047 -appkey 1e5ab6b3-
b6fa-4f9b-a3c2-743d31dffe86  -version 4.3.0 -platform Android -
inputSymbol
/Users/lilyxie/Downloads/upload_target/obj/arm64-v8a
```

inputSymbol:
When uploading multiple SO symbol table files, inputSymbol requires the directory path of these files to be uploaded. For instance, the example demonstrates uploading all results from the entire arm64-

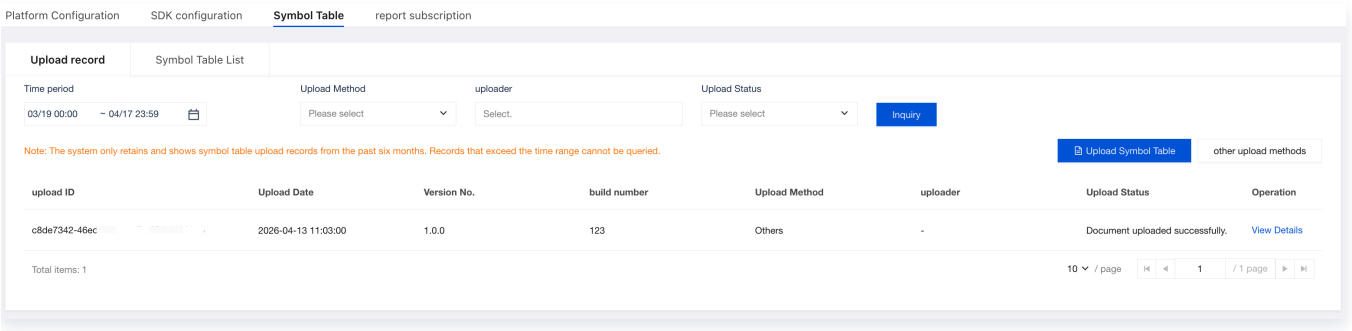
v8a architecture.

Results Summary:

```
...
##[info]rumpro tools android params: -appid a278f01047 -appkey 1e5ab6b3-
b6fa-4f9b-a3c2-743d31dffe86
-version 4.3.0 -platform Android -inputSymbol
/Users/lilyxie/Downloads/upload_target/obj/arm64-v8a
...
##[info]request jSOn is {"appID":"a278f01047","authSign":"1e5ab6b3-b6fa-
4f9b-a3c2-743d31dffe86",
"appVersion":"4.3.0","appBundleID":"com.tencent.demo.rumprodemo","appPla
tform":1,
"appBuildNumber":null,"fileType":201,"fileSize":1359658,"fileMD5":"3a54e
6b66b6901c6a51c528ea36d86dd",
"clientType":7,"clientVersion":"1.0.48","buildPlatform":1,"buildID":null
,"buildName":"null&null",
"fileInfoList":[{"name":"buglySymbol&arm64-
v8a&librmonitor_natmem&2655b543f9689d5dbcac22f995a18a44.zip",
"UUID":"2655b543f9689d5dbcac22f995a18a44","arch":"arm64-
v8a","type":101,"moduleName":"librmonitor_natmem"},
{"name":"buglySymbol&arm64-
v8a&libBugly_Native&e62597d2365613d35f1110454fa072a2.zip",
"UUID":"e62597d2365613d35f1110454fa072a2","arch":"arm64-
v8a","type":101,"moduleName":"libBugly_Native"},
{"name":"buglySymbol&arm64-
v8a&libbugly_dumper&a5cd2db99a1f45b45469979205f09730.zip",
"UUID":"a5cd2db99a1f45b45469979205f09730","arch":"arm64-
v8a","type":101,"moduleName":"libbugly_dumper"},
{"name":"buglySymbol&arm64-
v8a&librmonitor_memory_dump&3958037915dd881290ba72a3f598d862.zip",
"UUID":"3958037915dd881290ba72a3f598d862","arch":"arm64-
v8a","type":101,"moduleName":"librmonitor_memory_dump"},
{"name":"buglySymbol&arm64-
v8a&librmonitor_fd_stack&e8d847dac7e5fbdb8c134ccc74799e7d.zip",
"UUID":"e8d847dac7e5fbdb8c134ccc74799e7d","arch":"arm64-
v8a","type":101,"moduleName":"librmonitor_fd_stack"},
```

```
{"name":"buglySymbol&arm64-
v8a&librmonitor_core&54ef01f7521cf885a88c387fb6782875.zip","UUID":"54ef0
1f7521cf885a88c387fb6782875",
"arch":"arm64-v8a","type":101,"moduleName":"librmonitor_core"},
{"name":"buglySymbol&arm64-
v8a&librmonitor_base&5990bc6d36f0643987af905c60b3d8f8.zip",
"UUID":"5990bc6d36f0643987af905c60b3d8f8","arch":"arm64-
v8a","type":101,"moduleName":"librmonitor_base"}],
"buildRepo":null,"buildBranch":null,"buildCommitID":null}
##[info]request upload Info Url is https://symbol-
v2.bugly.qq.com/trpc.eff_tool.symbol_upload_gateway.SymbolUploadGateway/
uploadInfo
##[info]request gitRepoUrl is null
##[info]request gitRepoBranch is null
##[info]request gitRepoHeadCommitId is null
##[info]envtype is null
##[info]getAppModuleList is null
##[info]retCode: 200 response message:
{"statusCode":0,"msg":"success","uploadReqID":"a278f01047-aae1c0a6-492b-
4ebc-b6d7-9ed51e3571b1"}
##[info]now begin to uploadFileContent
##[info]request uploadFileurl is https://symbol-
v2.bugly.qq.com/trpc.eff_tool.symbol_upload_gateway.SymbolUploadGateway/
uploadFile
##[info]retCode: 200 response message:
{"statusCode":0,"msg":"success","uploadReqID":"a278f01047-aae1c0a6-492b-
4ebc-b6d7-9ed51e3571b1"}
```


Verify symbol table upload results:



Example 3: Upload mapping.txt file

Reference command line:

```
java -jar rumpro-upload-symbol.jar -appid a278f01047 -appkey 1e5ab6b3-
b6fa-4f9b-a3c2-743d31dffe86  -version 4.3.1 -buildNo 1 -platform Android
-inputMapping
/Users/temp/app/build/outputs/mapping/r8/release/mapping.txt
```

Results Summary:

```
...
##[info]args is ArgsParser{appId='a278f01047'appKey='1e5ab6b3-b6fa-4f9b-
a3c2-743d31dffe86', appPackage='com.tencent.demo.rumprodemo',
appVersion='4.3.1', appBuildNo='1', platformId=Android',
enviroment=null', symbolPathName='null',
mappingPathName='/Users/lilyxie/
workspace/apm/QAPM_SDK/app/build/outputs/mapping/r8/release/mapping.txt'
, appPathName='null'}
...
##[info]symbolFileDetail is
[{"name":"mapping&616cc28aa0dfdfce4cfcef72750a4583.zip","type":102}]
##[info]request jSOn is {"appID":"a278f01047","authSign":"1e5ab6b3-b6fa-
4f9b-a3c2-743d31dffe86","appVersion":"4.3.1",
"appBundleID":"com.tencent.demo.rumprodemo","appPlatform":1,"appBuildNum
ber":null,"fileType":202,"fileSize":727073,
"fileMD5":"b409dc5beb80613974290862 73d1e023","clientType":7,"clientVersi
on":"1.0.48","buildPlatform":1,"buildID":null,
"buildName":"null&null","fileInfoList":
[{"name":"mapping&616cc28aa0dfdfce4cfcef72750a4583.zip","type":102}],"bu
ildRepo":null,
"buildBranch":null,"buildCommitID":null}
##[info]request upload Info Url is https://symbol-
v2.bugly.qq.com/trpc.eff_tool.symbol_upload_gateway.SymbolUploadGateway/
uploadInfo
##[info]request gitRepoUrl is null
##[info]request gitRepoBranch is null
##[info]request gitRepoHeadCommitId is null
##[info]envtype is null
##[info]getAppModuleList is null
##[info]retCode: 200 response message:
{"statusCode":0,"msg":"success","uploadReqID":"a278f01047-72fd1c79-7096-
4818-8244-61248c5e0fe2"}
```

```
##[info]now begin to uploadFileContent
##[info]request uploadFileurl is https://symbol-
v2.bugly.qq.com/trpc.eff_tool.symbol_upload_gateway.SymbolUploadGateway/
uploadFile
##[info]retCode: 200 response message:
{"statusCode":0,"msg":"success","uploadReqID":"a278f01047-72fd1c79-7096-
4818-8244-61248c5e0fe2"}
```

Case 4: Simultaneously upload SO files and mapping.txt files

Reference command line:

```
java -jar rumpro-upload-symbol.jar -appid a278f01047 -appkey 1e5ab6b3-
b6fa-4f9b-a3c2-743d31dffe86  -version 4.3.0 -buildNo 2 -platform Android
-inputSymbol /Users/lilyxie/Downloads/upload_target/obj/arm64-v8a -
inputMapping
/Users/lilyxie/workspace/apm/QAPM_SDK/app/build/outputs/mapping/r8/relea
se/mapping.txt
```

Results Summary:

```
...
##[info]args is ArgsParser{appId='a278f01047'appKey='1e5ab6b3-b6fa-4f9b-
a3c2-743d31dffe86', appPackage='com.tencent.demo.rumprodemo',
appVersion='4.3.0', appBuildNo='2', platformId=Android',
enviroment=null',
symbolPathName='/Users/lilyxie/Downloads/upload_target/obj/arm64-v8a',
mappingPathName='/Users/lilyxie/workspace/apm/QAPM_SDK/app/build/outputs
/mapping/r8/release/mapping.txt', appPathName='null'}
...
##[info]request jSOn is {"appID":"a278f01047","authSign":"1e5ab6b3-b6fa-
4f9b-a3c2-743d31dffe86","appVersion":"4.3.0",
"appBundleID":"com.tencent.demo.rumprodemo","appPlatform":1,"appBuildNum
ber":null,"fileType":200,"fileSize":2086710,"fileMD5":"5e89401e93597eded
44527a9345bc43c",
"clientType":7,"clientVersion":"1.0.48","buildPlatform":1,"buildID":null
,"buildName":"null&null","fileInfoList":
[{"name":"buglySymbol&arm64-
v8a&librmonitor_natmem&2655b543f9689d5dbcac22f995a18a44.zip","UUID":"265
```

```
5b543f9689d5dbcac22f995a18a44",
"arch":"arm64-v8a","type":101,"moduleName":"librmonitor_natmem"},
{"name":"buglySymbol&arm64-
v8a&libBugly_Native&e62597d2365613d35f1110454fa072a2.zip",
"UUID":"e62597d2365613d35f1110454fa072a2","arch":"arm64-
v8a","type":101,"moduleName":"libBugly_Native"},
{"name":"buglySymbol&arm64-
v8a&libbugly_dumper&a5cd2db99a1f45b45469979205f09730.zip","UUID":"a5cd2d
b99a1f45b45469979205f09730",
"arch":"arm64-v8a","type":101,"moduleName":"libbugly_dumper"},
{"name":"buglySymbol&arm64-
v8a&librmonitor_memory_dump&3958037915dd881290ba72a3f598d862.zip",
"UUID":"3958037915dd881290ba72a3f598d862","arch":"arm64-
v8a","type":101,"moduleName":"librmonitor_memory_dump"},
{"name":"buglySymbol&arm64-
v8a&librmonitor_fd_stack&e8d847dac7e5fbdb8c134ccc74799e7d.zip","UUID":"e
8d847dac7e5fbdb8c134ccc74799e7d",
"arch":"arm64-v8a","type":101,"moduleName":"librmonitor_fd_stack"},
{"name":"buglySymbol&arm64-
v8a&librmonitor_core&54ef01f7521cf885a88c387fb6782875.zip",
"UUID":"54ef01f7521cf885a88c387fb6782875","arch":"arm64-
v8a","type":101,"moduleName":"librmonitor_core"},
{"name":"buglySymbol&arm64-
v8a&librmonitor_base&5990bc6d36f0643987af905c60b3d8f8.zip","UUID":"5990b
c6d36f0643987af905c60b3d8f8",
"arch":"arm64-v8a","type":101,"moduleName":"librmonitor_base"},
{"name":"mapping&616cc28aa0dfdfce4cfcef72750a4583.zip","type":102}],
"buildRepo":null,"buildBranch":null,"buildCommitID":null}
##[info]request upload Info Url is https://symbol-
v2.bugly.qq.com/trpc.eff_tool.symbol_upload_gateway.SymbolUploadGateway/
uploadInfo
##[info]request gitRepoUrl is null
##[info]request gitRepoBranch is null
##[info]request gitRepoHeadCommitId is null
##[info]envtype is null
##[info]getAppModuleList is null
##[info]retCode: 200 response message:
{"statusCode":0,"msg":"success","uploadReqID":"a278f01047-c6dee54b-a8c0-
4ec8-b478-5cce8dc228bf"}
##[info]now begin to uploadFileContent
```

```
##[info]request uploadFileurl is https://symbol-
v2.bugly.qq.com/trpc.eff_tool.symbol_upload_gateway.SymbolUploadGateway/
uploadFile
##[info]retCode: 200 response message:
{"statusCode":0,"msg":"success","uploadReqID":"a278f01047-c6dee54b-a8c0-
4ec8-b478-5cce8dc228bf"}
```

Case 5: Upload a dSYM file

Reference command line:

```
java -jar rumpro-upload-symbol.jar -appid a53fd62f6e -appkey aa1c666f-
d5e7-446b-8dc7-3bcb3bda7c7f  -version 2.3.0 -platform IOS -inputSymbol
/Users/lilyxie/workspace/temp/dSYM/Alamofire.framework.dSYM
```

Results Summary:

```
...
##[info]args is ArgsParser{appId='a53fd62f6e'appKey='aa1c666f-d5e7-446b-
8dc7-3bcb3bda7c7f',
appPackage='com.tencent.rumpro.demo.presentation.ios',
appVersion='2.3.0', appBuildNo='null', platformId=IOS',
enviroment=null',
symbolPathName='/Users/lilyxie/workspace/temp/dSYM/Alamofire.framework.d
SYM',
mappingPathName='null', appPathName='null'}
...
##[info]request jSOn is {"appID":"a53fd62f6e","authSign":"aa1c666f-d5e7-
446b-8dc7-3bcb3bda7c7f","appVersion":"2.3.0",
"appBundleID":"com.tencent.rumpro.demo.presentation.ios","appPlatform":2
,"appBuildNumber":null,"fileType":201,"fileSize":350816,
"fileMD5":"f69a32d33b16756a75d1716f728cef00","clientType":7,"clientVersi
on":"1.0.48","buildPlatform":1,"buildID":null,"buildName":"null&null",
"fileInfoList":
[{"name":"buglySymbol&Alamofire&arm64&5471ad32641a3f9f87610b133693b882.z
ip","UUID":"5471ad32641a3f9f87610b133693b882",
"arch":"arm64","type":101,"moduleName":"Alamofire"},
{"name":"buglySymbol&Alamofire&x86_64&60e900a3b60035b98fd84ccb13c4ec55.z
ip",
```

```
"UUID":"60e900a3b60035b98fd84ccb13c4ec55","arch":"x86_64","type":101,"mo
duleName":"Alamofire"}],"buildRepo":null,"buildBranch":null,"buildCommit
ID":null}
##[info]request upload Info Url is https://symbol-
v2.bugly.qq.com/trpc.eff_tool.symbol_upload_gateway.SymbolUploadGateway/
uploadInfo
##[info]request gitRepoUrl is null
##[info]request gitRepoBranch is null
##[info]request gitRepoHeadCommitId is null
##[info]envtype is null
##[info]getAppModuleList is null
##[info]retCode: 200 response message:
{"statusCode":0,"msg":"success","uploadReqID":"a53fd62f6e-ab4b2b2a-b074-
4ecd-b798-4d1d9458ba36"}
```

Case 6: Upload multiple dSYM files

Reference command line:

```
java -jar rumpro-upload-symbol.jar -appid a53fd62f6e -appkey aa1c666f-
d5e7-446b-8dc7-3bcb3bda7c7f  -version 2.3.0
-platform IOS -inputSymbol /Users/lilyxie/workspace/temp/dSYM
```

Results Summary:

```
...
##[info]args is ArgsParser{appId='a53fd62f6e'appKey='aa1c666f-d5e7-446b-
8dc7-3bcb3bda7c7f',
appPackage='com.tencent.rumpro.demo.presentation.ios',
appVersion='2.3.0', appBuildNo='null', platformId=IOS',
enviroment=null', symbolPathName='/Users/lilyxie/workspace/temp/dSYM',
mappingPathName='null', appPathName='null'}
...
##[info]request jSOn is {"appID":"a53fd62f6e","authSign":"aa1c666f-d5e7-
446b-8dc7-3bcb3bda7c7f","appVersion":"2.3.0",
"appBundleID":"com.tencent.rumpro.demo.presentation.ios","appPlatform":2
,"appBuildNumber":null,"fileType":201,"fileSize":4673680,
```

```
"fileMD5":"64c428045dfdcb0c376bea51760dd0df","clientType":7,"clientVersi
on":"1.0.48","buildPlatform":1,"buildID":null,"buildName":"null&null",
"fileInfoList":
[{"name":"buglySymbol&APTimeZones&arm64&659c3d9d48de30b6b47792cebd69ad22
.zip","UUID":"659c3d9d48de30b6b47792cebd69ad22",
"arch":"arm64","type":101,"moduleName":"APTimeZones"},
{"name":"buglySymbol&APTimeZones&x86_64&cae5639b99b030bd8c25d501a1e9262f
.zip",
"UUID":"cae5639b99b030bd8c25d501a1e9262f","arch":"x86_64","type":101,"mo
duleName":"APTimeZones"},
{"name":"buglySymbol&NearbyWeather&arm64&cd705ee49b29330dbcfacbd00774c7f
f.zip",
"UUID":"cd705ee49b29330dbcfacbd00774c7ff","arch":"arm64","type":101,"mod
uleName":"NearbyWeather"},
{"name":"buglySymbol&NearbyWeather&x86_64&f9813c5ec1a932f7a06e4167448ac9
a2.zip",
"UUID":"f9813c5ec1a932f7a06e4167448ac9a2","arch":"x86_64","type":101,"mo
duleName":"NearbyWeather"},
{"name":"buglySymbol&Rswift&arm64&915c4f2d6c6c3ce4821c6e8df1ce7001.zip",
"UUID":"915c4f2d6c6c3ce4821c6e8df1ce7001","arch":"arm64","type":101,"mod
uleName":"Rswift"},
{"name":"buglySymbol&Rswift&x86_64&1d7a56292ec73517a165b3a4645ab9b3.zip"
,
"UUID":"1d7a56292ec73517a165b3a4645ab9b3","arch":"x86_64","type":101,"mo
duleName":"Rswift"},
{"name":"buglySymbol&FMDB&arm64&31fc108031e339b780ebfd18f1877861.zip",
"UUID":"31fc108031e339b780ebfd18f1877861","arch":"arm64","type":101,"mod
uleName":"FMDB"},
{"name":"buglySymbol&FMDB&x86_64&79f5048815c43a6e99a8471edc4e071c.zip",
"UUID":"79f5048815c43a6e99a8471edc4e071c","arch":"x86_64","type":101,"mo
duleName":"FMDB"},
{"name":"buglySymbol&PKHUD&x86_64&805ab0b5ac413f2a8205de88ab83374f.zip",
"UUID":"805ab0b5ac413f2a8205de88ab83374f","arch":"x86_64","type":101,"mo
duleName":"PKHUD"},
{"name":"buglySymbol&PKHUD&arm64&4e20af3ed52a338b8f33683a462cc275.zip",
"UUID":"4e20af3ed52a338b8f33683a462cc275","arch":"arm64","type":101,"mod
uleName":"PKHUD"},
{"name":"buglySymbol&RxRelay&arm64&f16c9ca211c332609db10ab1be487ef2.zip"
,
```

"UUID":"f16c9ca211c332609db10ab1be487ef2","arch":"arm64","type":101,"mod
uleName":"RxRelay"},
{"name":"buglySymbol&RxRelay&x86_64&8c769ff24d4231ed98b0a95ee0967a51.zip
",
"UUID":"8c769ff24d4231ed98b0a95ee0967a51","arch":"x86_64","type":101,"mo
duleName":"RxRelay"},
{"name":"buglySymbol&RxCocoa&x86_64&34e3cf0ae4603507b361340ffcf2787a.zip
",
"UUID":"34e3cf0ae4603507b361340ffcf2787a","arch":"x86_64","type":101,"mo
duleName":"RxCocoa"},
{"name":"buglySymbol&RxCocoa&arm64&72cd027e9b59367a9a7b5febac26452c.zip"
,
"UUID":"72cd027e9b59367a9a7b5febac26452c","arch":"arm64","type":101,"mod
uleName":"RxCocoa"},
{"name":"buglySymbol&RxSwift&arm64&d4975416847c3104aa199584ca9e9c14.zip"
,
"UUID":"d4975416847c3104aa199584ca9e9c14","arch":"arm64","type":101,"mod
uleName":"RxSwift"},
{"name":"buglySymbol&RxSwift&x86_64&03ea574855783e2080ccb8473c8c70d6.zip
",
"UUID":"03ea574855783e2080ccb8473c8c70d6","arch":"x86_64","type":101,"mo
duleName":"RxSwift"},
{"name":"buglySymbol&TextFieldCounter&arm64&3e1d2e3091c03c6a926132fd8e67
87c2.zip",
"UUID":"3e1d2e3091c03c6a926132fd8e6787c2","arch":"arm64","type":101,"mod
uleName":"TextFieldCounter"},
{"name":"buglySymbol&TextFieldCounter&x86_64&5b22d2600ef2323693cbd4168f9
442e9.zip",
"UUID":"5b22d2600ef2323693cbd4168f9442e9","arch":"x86_64","type":101,"mo
duleName":"TextFieldCounter"},
{"name":"buglySymbol&SwiftMonkey&x86_64&5eb2bb9f2535313bb6d52d20cb997c3f
.zip",
"UUID":"5eb2bb9f2535313bb6d52d20cb997c3f","arch":"x86_64","type":101,"mo
duleName":"SwiftMonkey"},
{"name":"buglySymbol&SwiftMonkey&arm64&ce1692374ecb37fbb0cf58671f91687d.
zip",
"UUID":"ce1692374ecb37fbb0cf58671f91687d","arch":"arm64","type":101,"mod
uleName":"SwiftMonkey"},
{"name":"buglySymbol&RxFlow&arm64&05ab5fa62ccc33ac92bee8302f3c03ac.zip",

```
"UUID":"05ab5fa62ccc33ac92bee8302f3c03ac","arch":"arm64","type":101,"mod
uleName":"RxFlow"},
{"name":"buglySymbol&RxFlow&x86_64&8b4df307db4a34d2b003d65eb5f6990f.zip"
,
"UUID":"8b4df307db4a34d2b003d65eb5f6990f","arch":"x86_64","type":101,"mo
duleName":"RxFlow"},
{"name":"buglySymbol&NearbyWeatherUITests&arm64&2d853bcf1c703a6ebb0767da
e49033d3.zip",
"UUID":"2d853bcf1c703a6ebb0767dae49033d3","arch":"arm64","type":101,"mod
uleName":"NearbyWeatherUITests"},
{"name":"buglySymbol&NearbyWeatherUITests&x86_64&98bcb7abb3a1376593d8b88
be64102d4.zip",
"UUID":"98bcb7abb3a1376593d8b88be64102d4","arch":"x86_64","type":101,"mo
duleName":"NearbyWeatherUITests"},
{"name":"buglySymbol&Alamofire&arm64&5471ad32641a3f9f87610b133693b882.zi
p",
"UUID":"5471ad32641a3f9f87610b133693b882","arch":"arm64","type":101,"mod
uleName":"Alamofire"},
{"name":"buglySymbol&Alamofire&x86_64&60e900a3b60035b98fd84ccb13c4ec55.z
ip",
"UUID":"60e900a3b60035b98fd84ccb13c4ec55","arch":"x86_64","type":101,"mo
duleName":"Alamofire"}],"buildRepo":null,"buildBranch":null,"buildCommit
ID":null}
##[info]request upload Info Url is https://symbol-
v2.bugly.qq.com/trpc.eff_tool.symbol_upload_gateway.SymbolUploadGateway/
uploadInfo
##[info]request gitRepoUrl is null
##[info]request gitRepoBranch is null
##[info]request gitRepoHeadCommitId is null
##[info]envtype is null
##[info]getAppModuleList is null
##[info]retCode: 200 response message:
{"statusCode":0,"msg":"success","uploadReqID":"a53fd62f6e-7b7c95ad-c497-
4297-b8af-f919741228bb"}
##[info]now begin to uploadFileContent
##[info]request uploadFileurl is https://symbol-
v2.bugly.qq.com/trpc.eff_tool.symbol_upload_gateway.SymbolUploadGateway/
uploadFile
##[info]retCode: 200 response message:
{"statusCode":0,"msg":"success","uploadReqID":"a53fd62f6e-7b7c95ad-c497-
```

```
4297-b8af-f919741228bb"}
```

Case 7: Upload Harmony platform symbol table

If you need to upload HarmonyOS SO files, first set the temporary environment variable USE_LLVM to true. Reference command line:

```
export USE_LLVM=true
```

Execute the symbol table upload command.

```
java -jar rumpro-upload-symbol.jar -appid a53fd62f6e -appkey aa1c666f-
d5e7-446b-8dc7-3bcb3bda7c7f -version 1.0.0 -platform Harmony -
inputSymbol
./harmony_test/libdemo.SO -inputMapping harmony_test/obfuscation -
buildNo 0
```

Note:

- Before uploading HarmonyOS SO symbol tables, you must set the USE_LLVM environment variable; otherwise, the upload may fail.
- When uploading, the platform must be set to Harmony, paying attention to case sensitivity.
- Uploading nameCache and SourceMaps reuses the Android platform's –inputMapping parameter. The difference is that when the platform is HarmonyOS, files containing both nameCache and SourceMaps will be retrieved simultaneously.
- Supports uploading a single SO or multiple SOs, allows separate uploads of nameCache and SourceMaps, and enables simultaneous uploads of SO with nameCache and SourceMaps. For examples, refer to Case 1, Case 2, Case 3, and Case 4. Ensure the platform is set correctly.
- Replace the jar package name in the command with the actually used jar package name.

Upload successful. Results Summary:

```
##[info]args is ArgsParser{appId='a53fd62f6e', appKey='aa1c666f-d5e7-
446b-8dc7-3bcb3bda7c7f', appPackage='null', appVersion='1.0.0',
appBuildNo='2', platformId='Harmony',
enviroment='null', symbolPathName='./libdemo.SO',
mappingPathName='harmony_test/obfuscation', appPathName='null'}
##[info]param is : AtomParam{bundleId='null', version='1.0.0',
secret='null', platform='Harmony', platformModule='null',
```

```
symbolPath='./libdemo.SO', appPath='null',
mappingPath='harmony_test/obfuscation', secretSource='input',
secretInputAppId='a53fd62f6e', secretInputKey='aa1c666f-d5e7-446b-8dc7-
3bcb3bda7c7f', buildNo='2',
buildPackageType='app', appModuleList='null', moduleName='null',
envType='null'}
##[info]deleteDirFiles file path is
/Users/zhuangchihui/Documents/TestData/buglybin/symbolTempStore/bugly_na
meCache&85272d438d47884151c9b48bf36b5131.zip
##[info]deleteDirFiles file path is
/Users/zhuangchihui/Documents/TestData/buglybin/symbolTempStore/SOurceMa
ps&7fc36306136e9f57db449eb0d61963a1.zip
##[info]deleteDirFiles file path is
/Users/zhuangchihui/Documents/TestData/buglybin/symbolTempStore
##[info]deleteDirFiles file path is
/Users/zhuangchihui/Documents/TestData/buglybin/symbolResult/BuglyMappin
gListZip.zip
##[info]deleteDirFiles file path is
/Users/zhuangchihui/Documents/TestData/buglybin/symbolResult
##[info]deleteDirFiles file path is
/Users/zhuangchihui/Documents/TestData/buglybin
##[info]packSymbolFile file
##[info]workspace:null
##[info]buildPackageType:app
##[info]find mapping file:[harmony_test/obfuscation/SOurceMaps.jSOn,
harmony_test/obfuscation/bugly_nameCache.jSOn]
##[info]checkSymbolUpload request jSOn is
{"appID":"a53fd62f6e","symbolUUIDs":
["b151a429a057c9b78803bac7f4b15278"]}
##[info]retCode: 200 response message: {"statusCode":0,"msg":"All symbol
tables have already been uploaded.","notExistUUIDs":[]}
...
##[info]request jSOn is {"appID":"a53fd62f6e","authSign":"aa1c666f-d5e7-
446b-8dc7-
3bcb3bda7c7f","appVersion":"1.0.0","appBundleID":null,"appPlatform":11,"
appBuildNumber":"0",
"fileType":202,"fileSize":18657,"fileMD5":"80efa7c601bc5fe0c9c065c669997
786","clientType":4,"clientVersion":"3.4.2","buildPlatform":1,"buildID":
null,"buildName":"null&null",
```

```
"fileInfoList":
[{"name":"SOurceMaps&7fc36306136e9f57db449eb0d61963a1.zip","type":102},
{"name":"bugly_nameCache&85272d438d47884151c9b48bf36b5131.zip","type":10
2}],
"buildRepo":null,"buildBranch":null,"buildCommitID":null,"appType":0}
##[info]retCode: 200 response message:
{"statusCode":0,"msg":"success","uploadReqID":"a53fd62f6e-84fd4052-44c7-
45da-8b2e-ac59f3846733"}
```

Symbol table classification

Last updated：2026-05-25 18:53:28

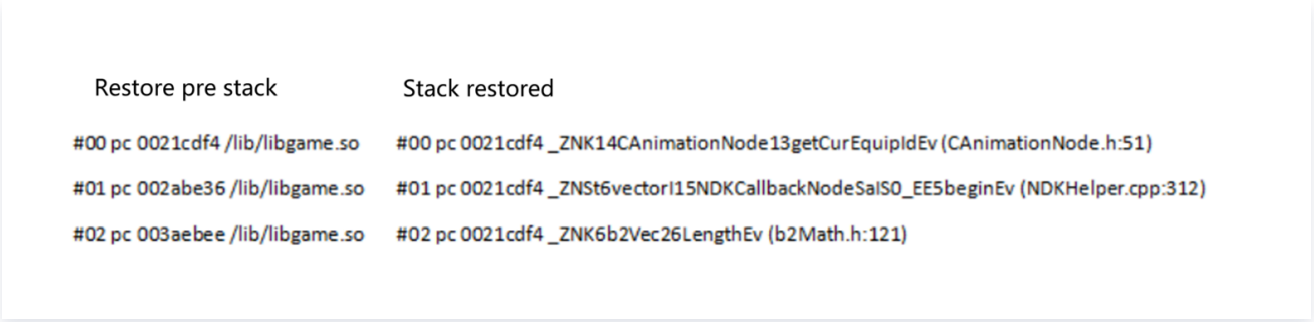
This article primarily covers the definition, lookup, and detailed explanations of various configurations for symbol tables on the Android, iOS, and Harmony platforms.

Android symbol table

In the Android platform, symbol tables include two types: SO symbol tables and Java mapping files.

- SO symbol table: A symbol table is a mapping table between memory addresses and function names, file names, and line numbers. The symbol table elements are: `<start address> <end address> <function> [<filename:line number>]`.

  For Native Crash, to quickly and accurately locate the code position where a user's App crashes, the platform uses symbol tables to parse and restore the program stack of the App crash, as shown in the following figure:



- mapping file: A mapping file is used to deobfuscate obfuscated Java code. For details, see Code Obfuscation.

  Android mapping files are associated by App version and build number. For example, for the specified exception in the example above, the platform locates the matching symbol table file by App version and build number. For a given App version and build number, the platform supports uploading multiple symbol table files, which are distinguished by file names.

  The platform recommends that business parties name the host mapping file as mapping.txt, and name plugin or component symbol table files as xxx_mapping.txt. These symbol table files can be uploaded in batches or multiple times. For details, see Uploading Plugin Symbol Tables for Android Apps.

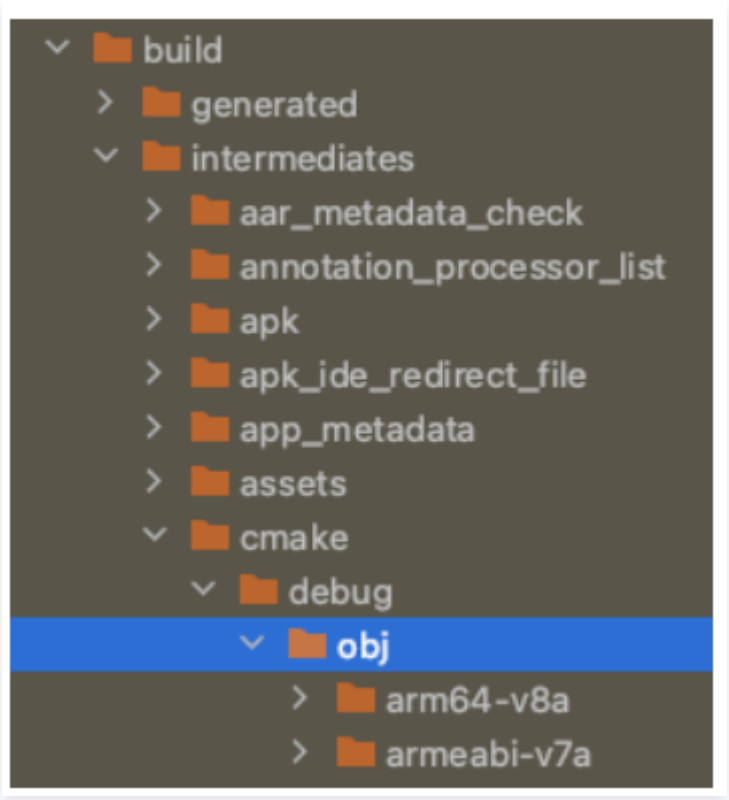
Look up symbol tables

Location of SO Symbol Table

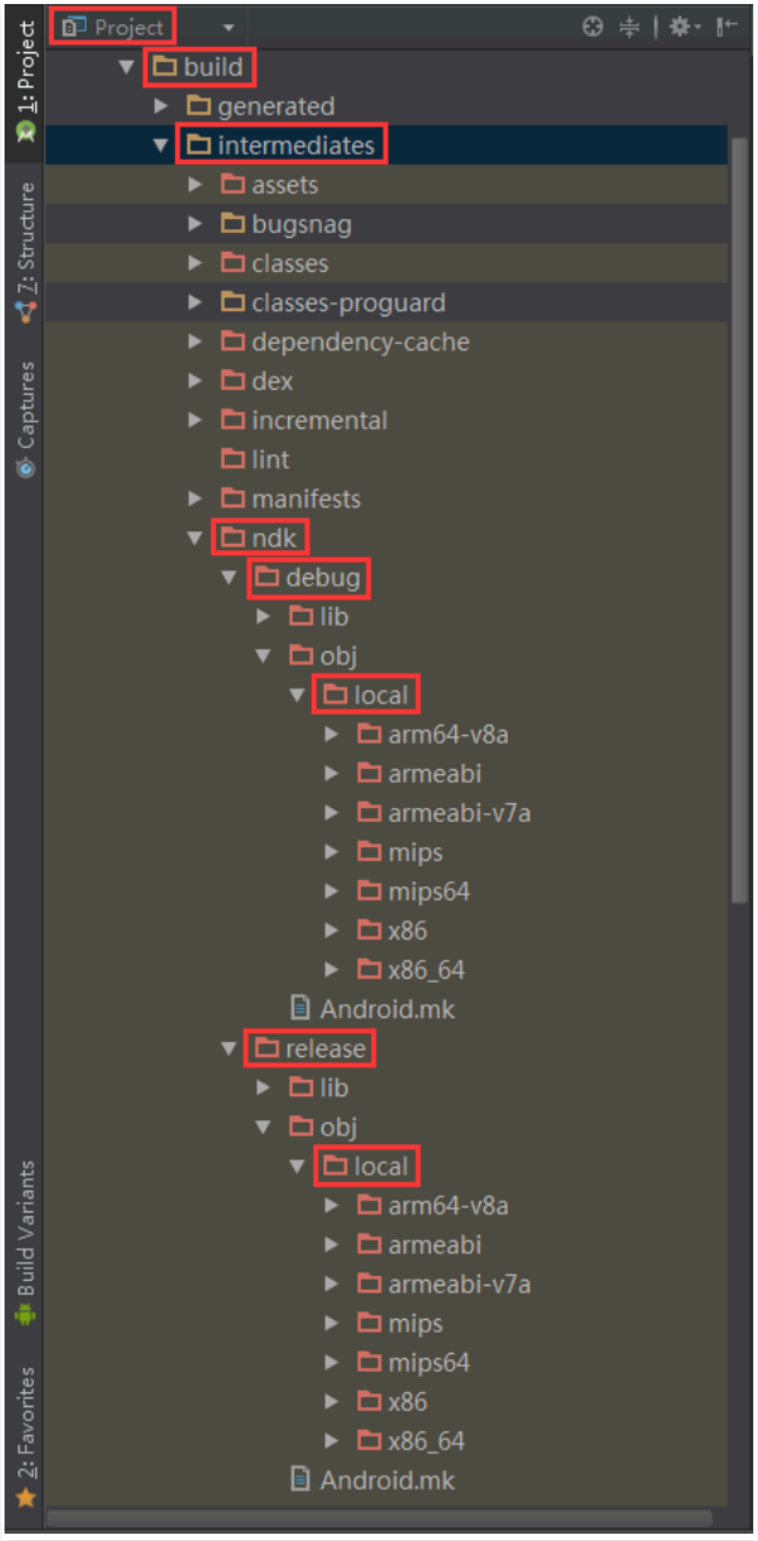
In the Android platform, Debug SO files refer to SO files with debugging information, which contain symbol information for stack restoration.

> **Note:**
> To facilitate retrieving the Debug SO files corresponding to crashes and restoring the stack, it is recommended to back up Debug SO files during each build or release of the App version, or to upload the symbol tables to the platform using the symbol table upload tool.

- **CMake Compilation Project:** By default, the Debug SO files compiled in Debug mode are located at `<Project Folder>/<Module>/build/intermediates/cmake/debug/obj/local<Architecture>/`.

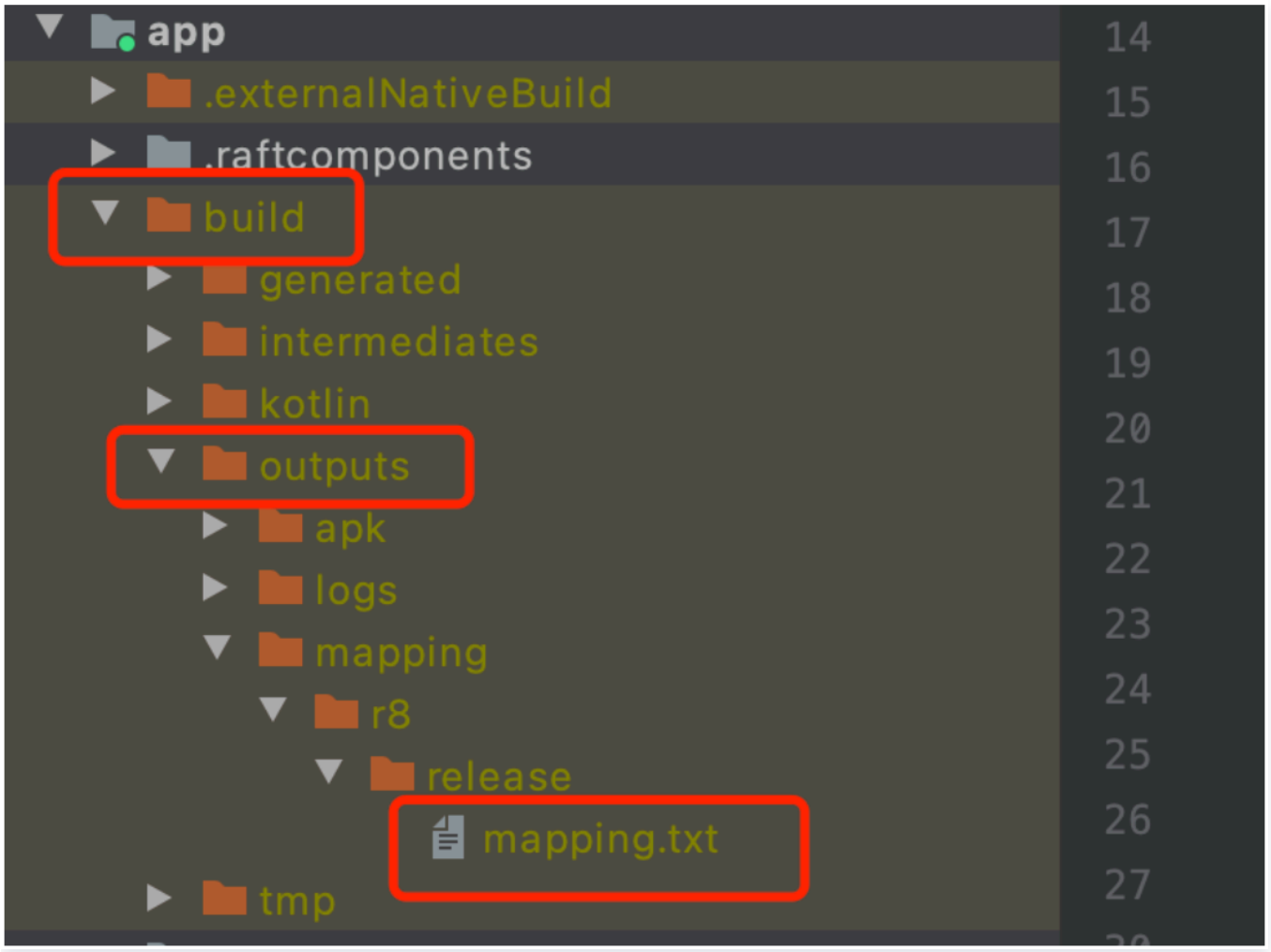


- **NDK Compilation Project:** By default, the Debug SO files compiled in Debug mode are located at `<Project Folder>/<Module>/build/intermediates/ndk/debug/obj/local<Architecture>/`.



mapping file location

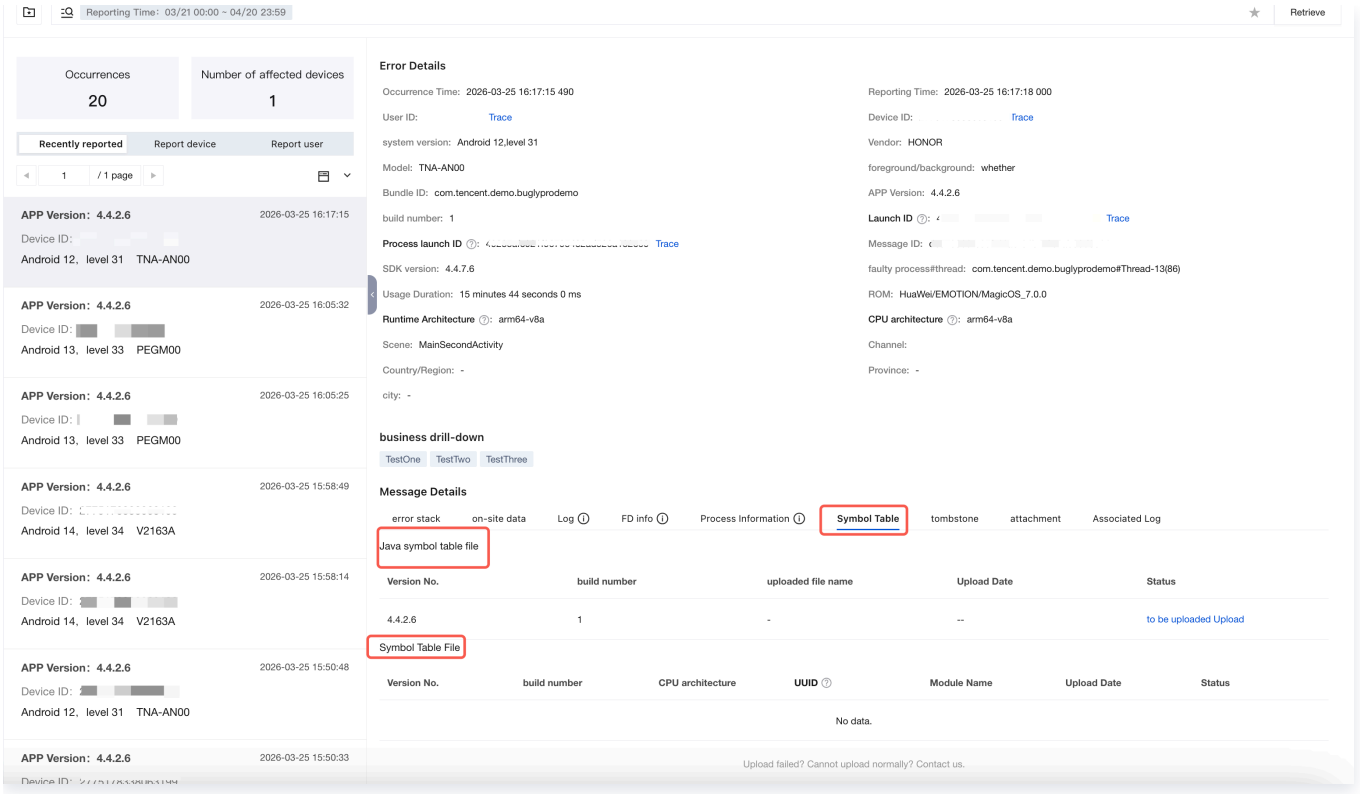
By default, the mapping file is located at `<Project Folder>/<Module>/build/outputs/mapping/<build-type>/` .



Match symbol tables with crashes

For Native Crash stack restoration, the platform needs to match symbol table files based on UUID. Only when the UUID of the uploaded symbol table file matches the UUID of the SO file in the Native Crash stack (matching from the end backwards) can the stack be accurately restored.
For Java stack restoration, the current platform uses **App version number + build number** to identify and locate mapping files.

- To view the UUID of the symbol table file, for details, see UUID Extraction Guide.
- In **Issue Details** > **Message Details** > **Symbol Table**, view the symbol table information in the current exception stack.



View the UUID of the Debug SO files

The UUID of the symbol table file is consistent with that of the Debug SO file. Therefore, you can view the UUID of the Debug SO file using the symbol table file generated by the symbol table tool. The steps are as follows: Generate the symbol table file (.zip) –> Unzip the symbol table file (.symbol) –> Open the symbol table file using a text editor.

```
1  File:   xxx/libxxx.so
2  Format: ELF/32-Bit
3  Arch:   armeabi-v7a
4  Symbols:    123456
5  Tool Version:   2.6.1
6  File Version:   1.4
7  SHA-1: 2980f23f1c92b1e973ea58f310cece170b0aa197
8  Built Time: 2016-06-05 06:40:03
9  Symbol table:
```

The "SHA–1" information of the symbol table file is the UUID of the Debug SO file and also the UUID of the symbol table file. If the file is large, it is recommended to use a text editor such as "Sublime Text" to open the symbol table file. For more details, see UUID Extraction Guide.

> **Note:**
> Since the platform has adopted a new UUID calculation rule, to correctly match the SO file corresponding to the Crash stack, please use version 2.5.0 or later of the symbol table tool.

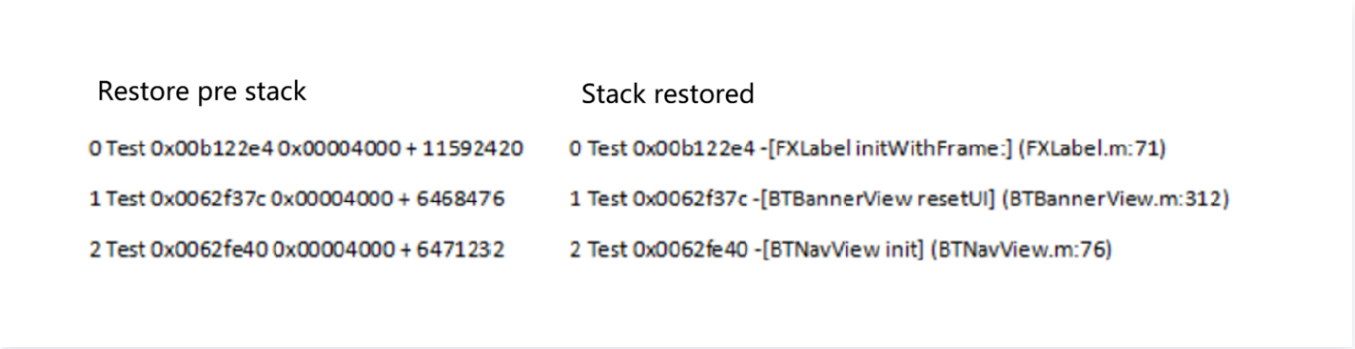
Locate the Debug SO file corresponding to the Crash

If the symbol table file or Debug SO file corresponding to the Crash can no longer be found locally, but the Native project code for the App version associated with the Crash can still be retrieved, it is recommended to recompile the Debug SO file using NDK and generate the symbol table file with the symbol table tool.

> **Note:**
> If the Debug SO file cannot be retrieved from the Native project code, the Crash stack cannot be restored. Therefore, it is recommended to back up the Debug SO file when building or releasing each App version.

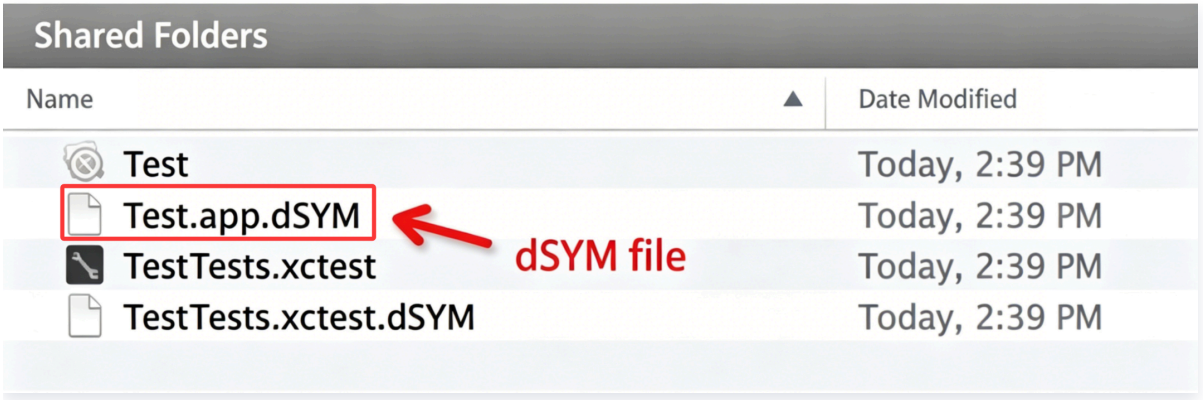
iOS symbol table

A symbol table is a mapping table between memory addresses and function names, file names, and line numbers. The symbol table elements are: `<start address> <end address> <function> [<filename:line number>]`.
To quickly and accurately locate the code position where a user's App crashes, the platform uses symbol tables to parse and restore the program stack of the App Crash, as shown in the following figure:



dSYM file

dSYM stands for Debug Symbols, namely the symbol table file. Symbols correspond to classes, functions, variables, etc. This symbol table file maps memory to symbols, such as function names, file names, line numbers, and plays a crucial role in crash log analysis.
In the iOS platform, the dSYM file refers to an object file with debugging information, typically named: xxx.app.dSYM. As shown in the following figure:

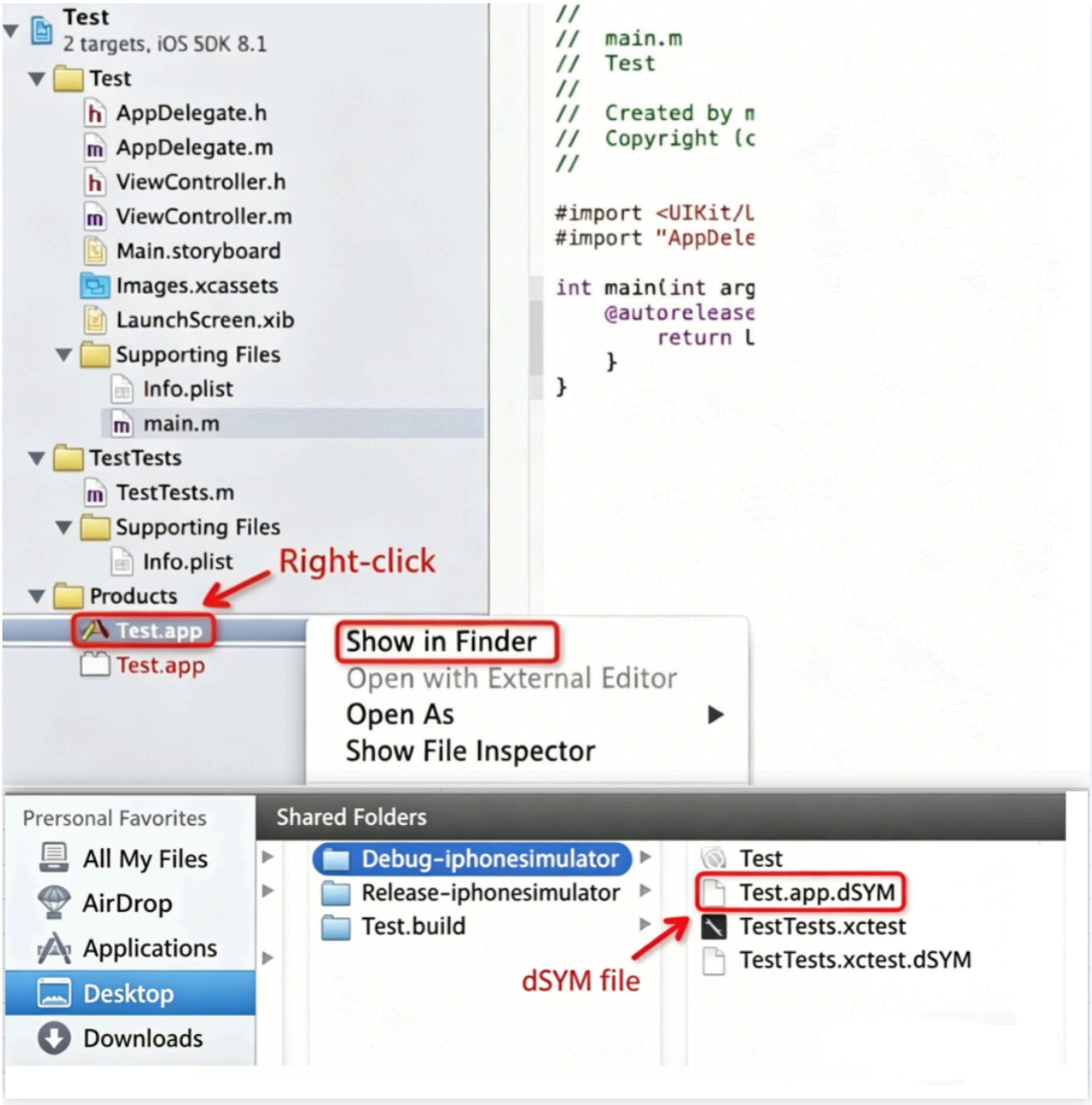


Back up symbol table files:
To facilitate retrieving the dSYM file corresponding to a Crash and restoring stack information, it is recommended to back up the dSYM file when building or releasing each App version.

Locate the dSYM file

Generally, after compiling the project, the dSYM file is located in the same directory as the App file. The following uses Xcode as the IDE to provide detailed instructions on locating the dSYM file.

1. Go to Xcode.
2. Open the project (that has been compiled).
3. Locate **Products** in the left sidebar.
4. Right-click the compiled "xxx.app", then click **Show in Finder**.



Note:

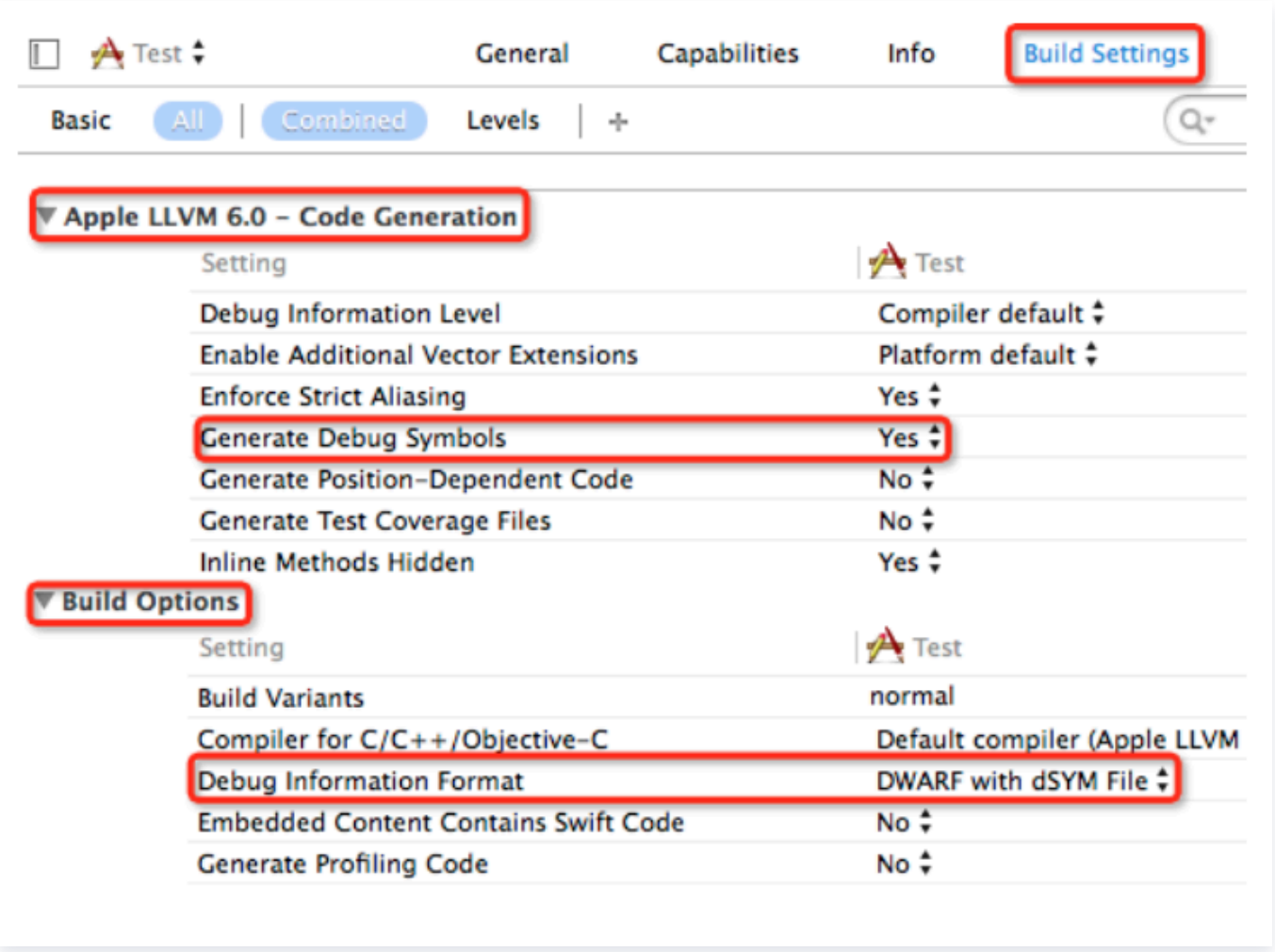
If there are multiple dSYM files, you can specify the input as the directory containing the dSYM files or the project directory when uploading using the symbol table tool.

Xcode Debug Compilation Configuration

By default, Xcode Release builds generate dSYM files, while Debug builds do not. The corresponding Xcode configurations are as follows:

```
XCode -> Build Settings -> Code Generation -> Generate Debug Symbols ->
Yes

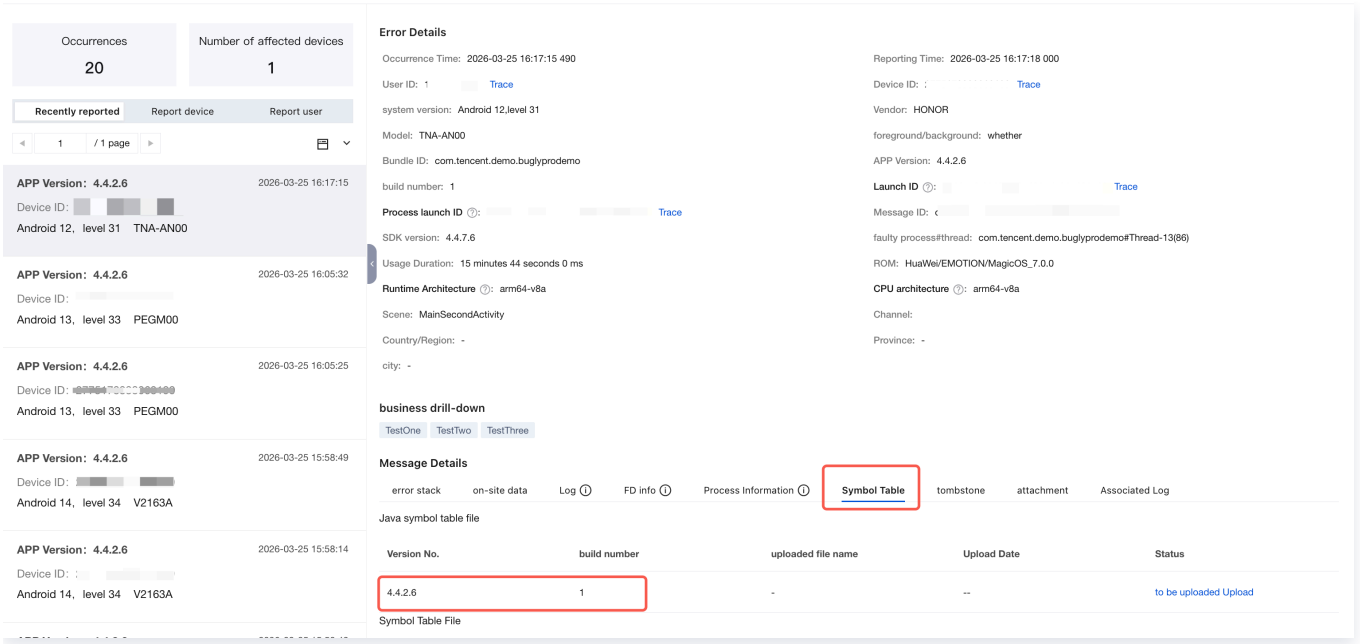
XCode -> Build Settings -> Build Option -> Debug Information Format ->
DWARF with dSYM File
```



The dSYM file corresponds to the UUID of the Crash

When restoring a Crash stack, the platform needs to match symbol table files based on UUID. Only when the UUID of the uploaded symbol table file matches the UUID of the Crash-related App can the stack be accurately restored.

- To view the UUID of the symbol table file, for details, see UUID Extraction Guide.
- In **Issue Details** > **Message Details** > **Symbol Table**, view the symbol table information in the current exception stack.



View the UUID of the dSYM file

- View UUID via command

```
xcrun dwarfdump --UUID <dSYM file>
```

- View UUID via symbol table file

  The UUID of the symbol table file is consistent with that of the dSYM file. Therefore, you can view the UUID of the dSYM file using the symbol table file generated by the symbol table tool. The steps are as follows: Generate the symbol table file (.zip) –> Unzip the symbol table file (.symbol) –> Open it with a text editor.

```
File:   xxx/xxx.app.dSYM/Contents/Resources/DWARF/xxx
Format: Mach-O/64-Bit
Arch:   arm64
Symbols:    123456
Tool Version:   2.4.1
File Version:   1.4
UUID:   42f6f43fd33130279600c024c1490ca4
Built Time: 2016-06-12 02:24:23
Symbol table:
```

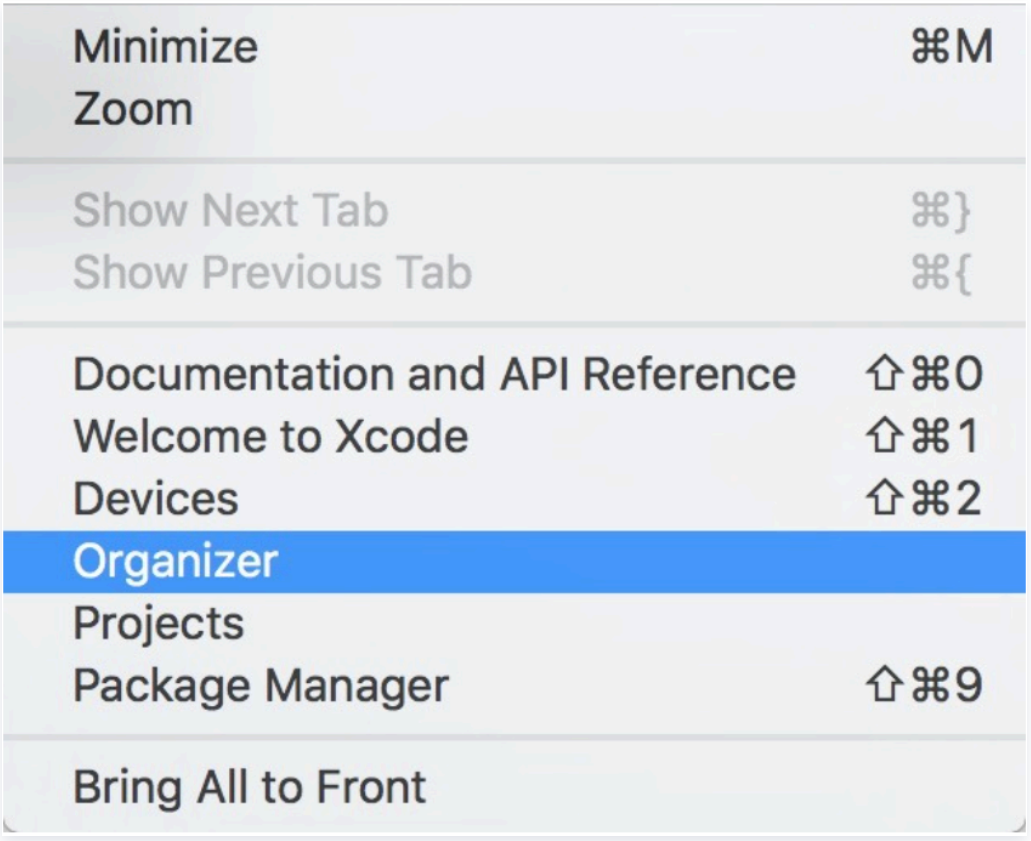
  The "UUID" information in the symbol table file is the UUID of the Debug SO file and also the UUID of the symbol table file itself. If the file is large, it is recommended to use a text editor such as "Sublime Text" to open the symbol table file.

Retrieve the dSYM file corresponding to the App published to the App Store

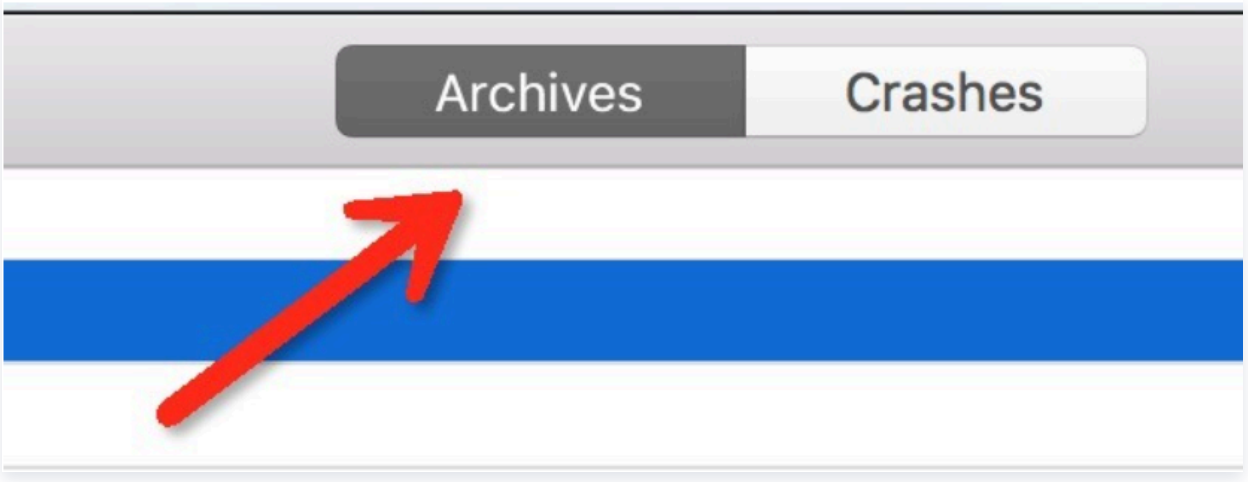
Support retrieving the dSYM file corresponding to the App published to the App Store via Xcode, iTunes Connect, or the mdfind tool.

Retrieve via Xcode

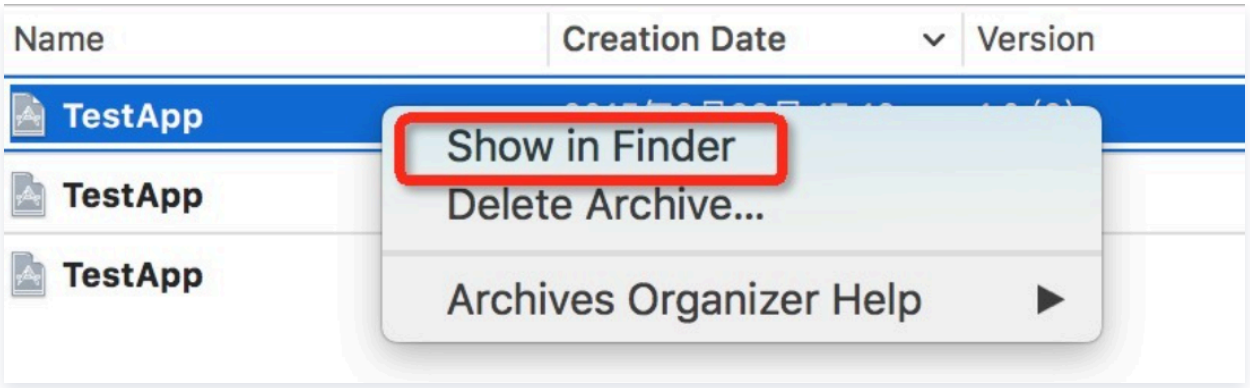
1. Open the top menu bar in Xcode > Window > Organizer window.



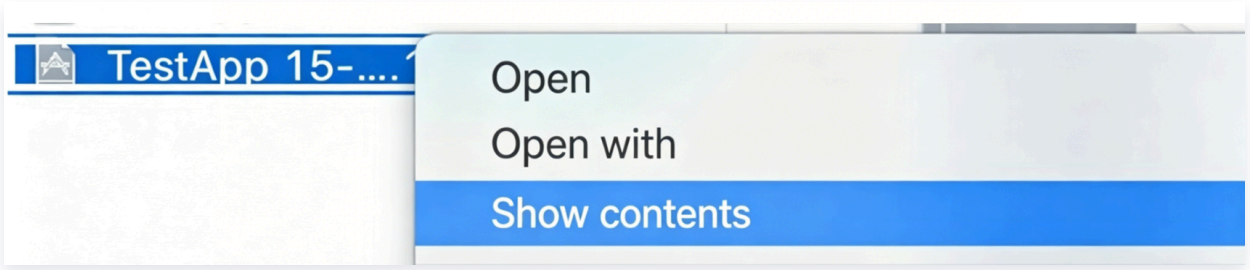
2. Open the top menu bar in Xcode, and select the **Archives** Tag.



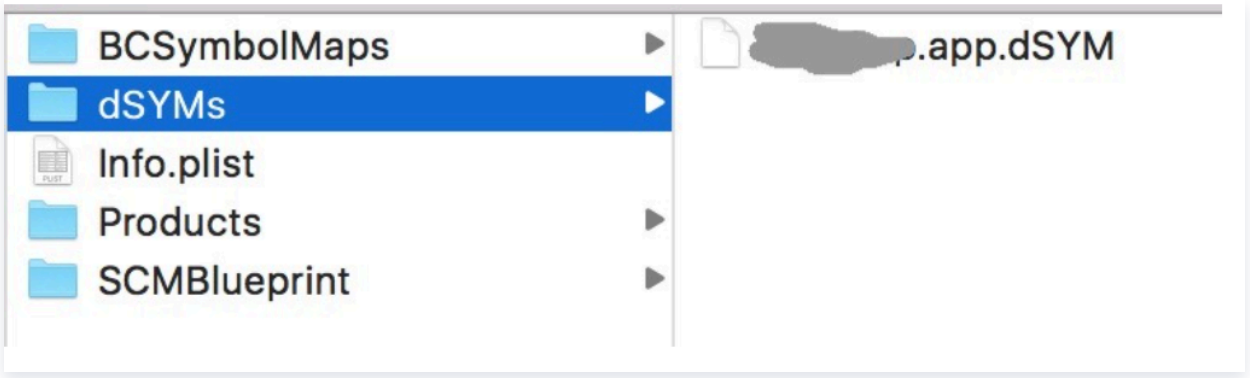
3. Locate the published archived package, right-click the corresponding archived package, and select the **Show in Finder** action.



4. Right-click the located archived file and select **Show contents**.

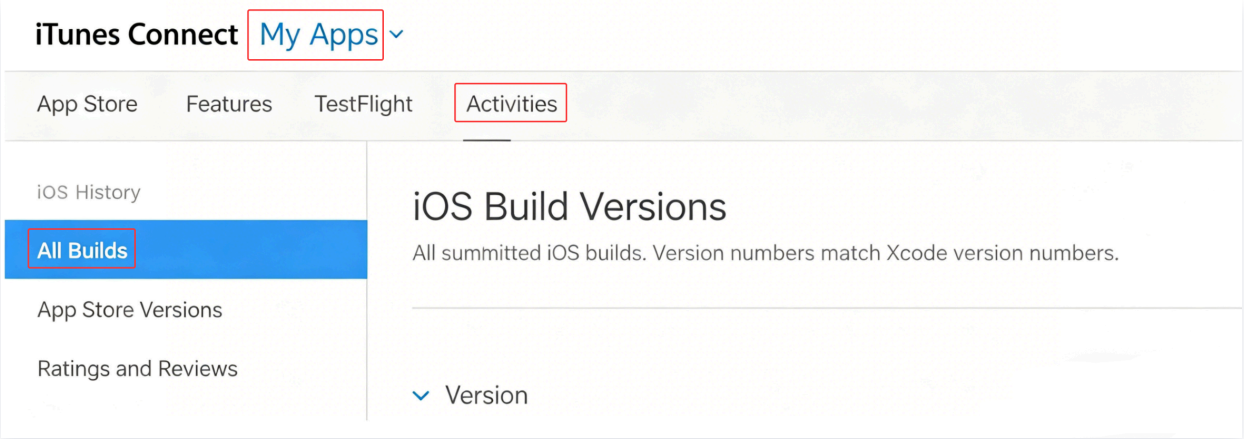


5. Select the dSYMs directory, inside which are the downloaded dSYM files.

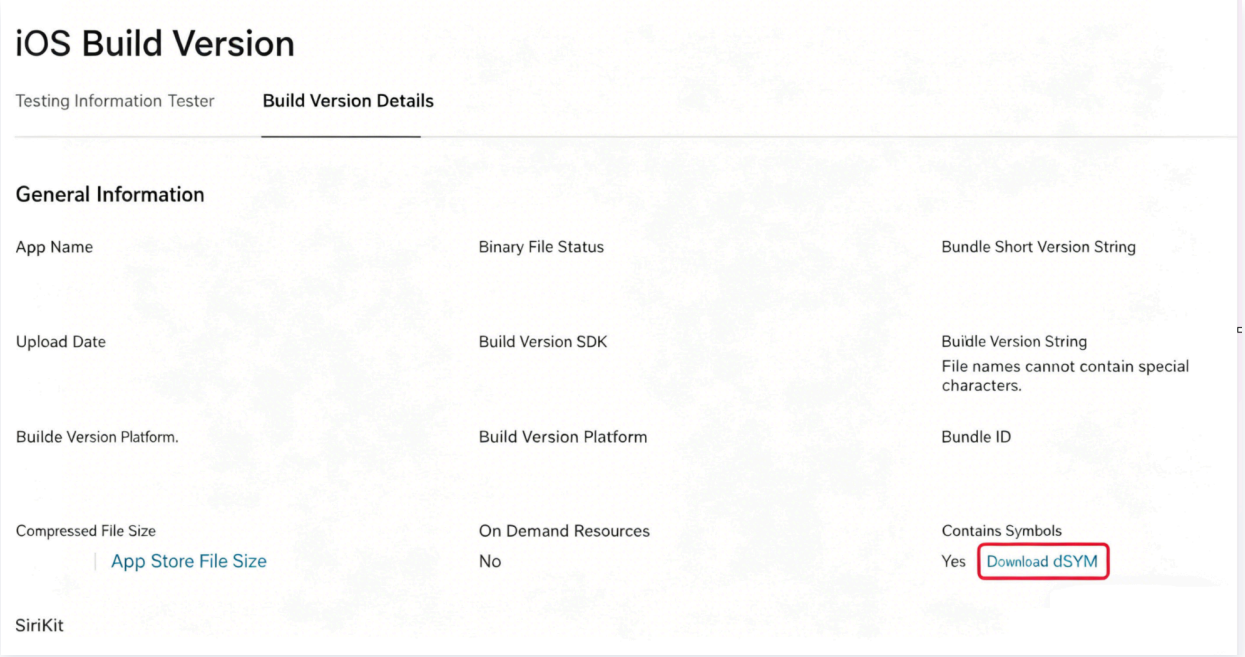


Retrieve via iTunes Connect

1. Log in to iTunes Connect.
2. Go to **My Apps** > **Activities** page.



3. In **All Builds**, select a specific version and click **Download dSYM** (Download dSYM) to download the dSYM file.



Retrieve using the mdfind tool

1. Locate the UUID corresponding to the Crash on the platform's issue page.
2. Then, in the Mac Shell, use the mdfind command to locate the dSYM file.

```
mdfind "com_apple_xcode_dsym_UUIDs == <UUID>"
```

> **Note:**
> Note that when using mdfind, the UUID requires format conversion (adding "–"), for example: 12345678–1234–1234–1234–xxxxxxxxxxxx.

For example, if the UUID of the dSYM to be located is: E30FC309DF7B3C9F8AC57F0F6047D65F, use the following command to locate the dSYM file.

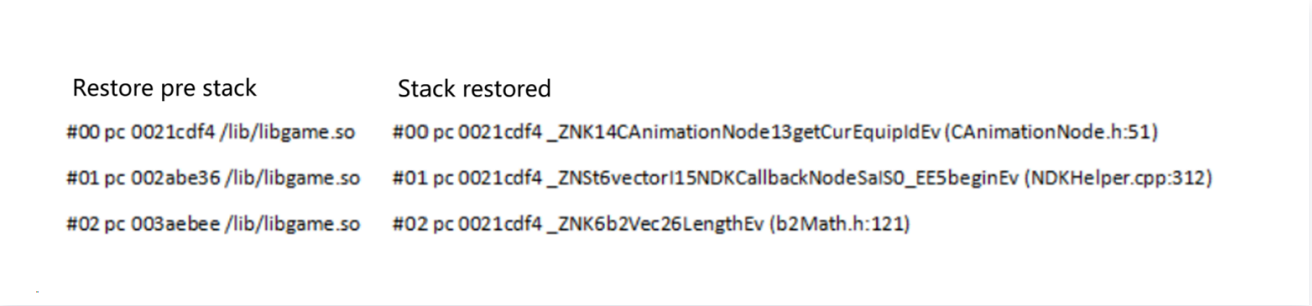
```
mdfind "com_apple_xcode_dsym_UUIDs == E30FC309-DF7B-3C9F-8AC5-
7F0F6047D65F"
                                    |12345678-1234-1234-1234-
xxxxxxxxxxxx|
```

Note:
We recommend backing up the dSYM file corresponding to the App each time you build or release an App version.

HarmonyOS (Hongmeng) Symbol Table

- SO symbol table: A symbol table is a mapping table between memory addresses and function names, file names, and line numbers. The symbol table elements are: `<start address> <end address> <function> [<filename:line number>]`.

  For HarmonyOS Native Crash, to quickly and accurately locate the code position where a user's HAP encounters a crash, the platform uses symbol tables to parse and restore the program stack of the crashed HAP. For example, as shown in the following figure:



- nameCache & SourceMaps files: Harmony's nameCache & SourceMaps files are similar to Android's mapping: The nameCache file is used for mapping symbols and variable names before and after obfuscation in ts/js files, while the SourceMaps file maps line numbers before and after obfuscation in ts/js files.

  In the HarmonyOS IDE, configure the obfuscated build for HAP or HAR packages. For details, see HarmonyOS Code Obfuscation Rules Configuration and Building Closed-Source HAR in HarmonyOS.

  nameCache & SourceMaps files are associated via `HAP version + build number`. For example, for the specified crash instance above, the platform locates matching symbol table files by `HAP version + build number`. For a given HAP version and build number, the platform supports uploading multiple

symbol table files, distinguished by their filenames. For instance, under the same version number + build number, the following files can be uploaded simultaneously.

- nameCache.JSON
- SourceMaps.JSON
- rumpro_nameCache.JSON
- rumpro_SourceMaps.JSON

The platform recommends that services name the host's nameCache and SourceMaps files as nameCache.JSON and SourceMaps.JSON, respectively. Symbol table files for plugins or components should be named xxx_nameCache.JSON or xxx_SourceMaps.JSON. These symbol table files can be uploaded via batch upload or multiple uploads. If a file with the same name already exists under the same version number + build number, uploading it again will overwrite the previous file with the same name.

Note:

- Currently, the HarmonyOS HAR package does not support generating SourceMaps files through obfuscated builds.
- If the link for building closed-source rules cannot be accessed, you need to apply for HarmonyOS development permissions from Huawei.

Look up symbol tables

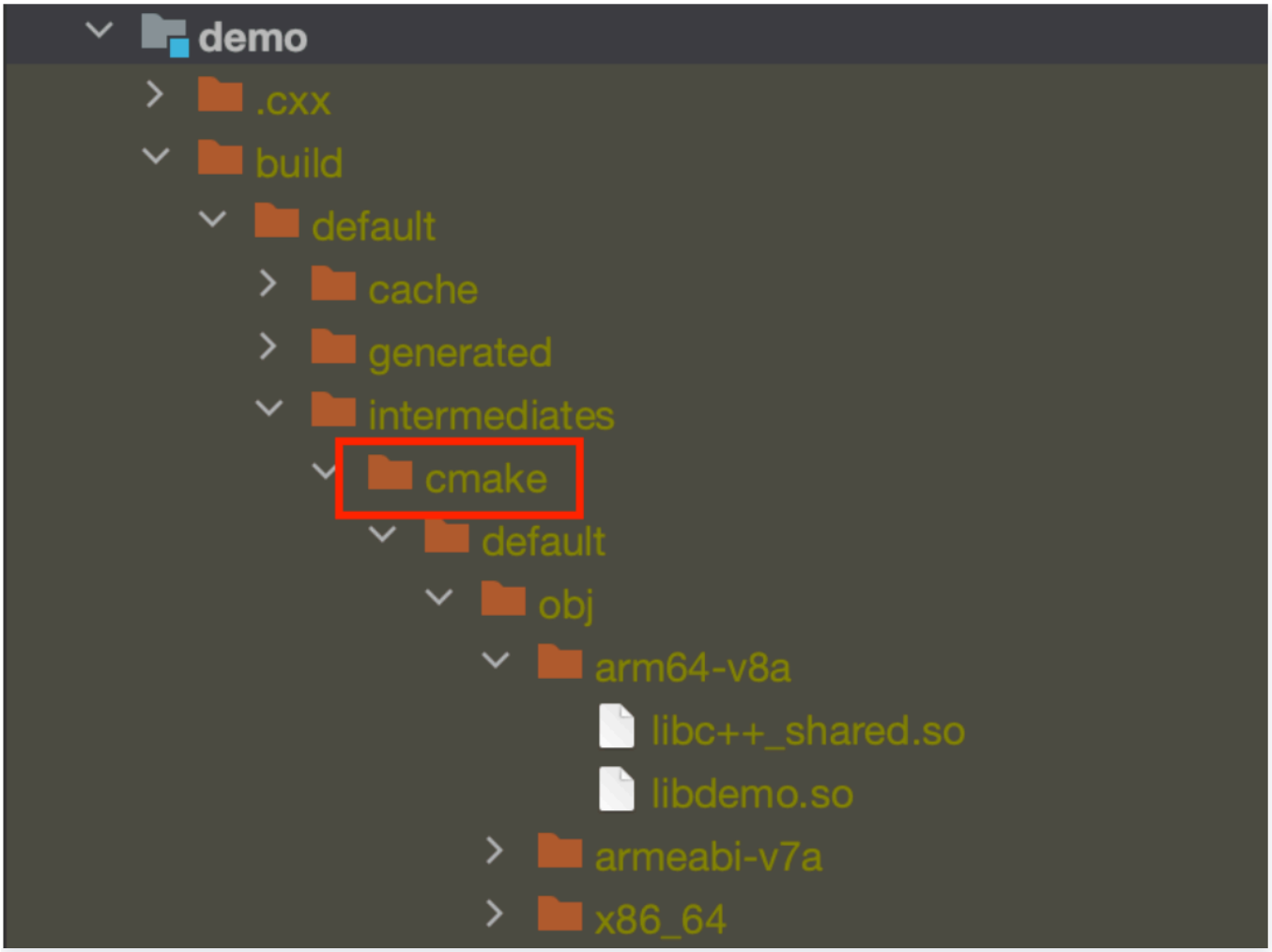
Location of SO Symbol Table

In the Harmony platform, Debug SO files refer to SO files containing debugging information, which includes symbol information for users to restore the stack. The Harmony platform compiles SO files using DWARF 5 format debugging information by default, while the Android platform typically uses DWARF 4/3 format debugging information.

Note:

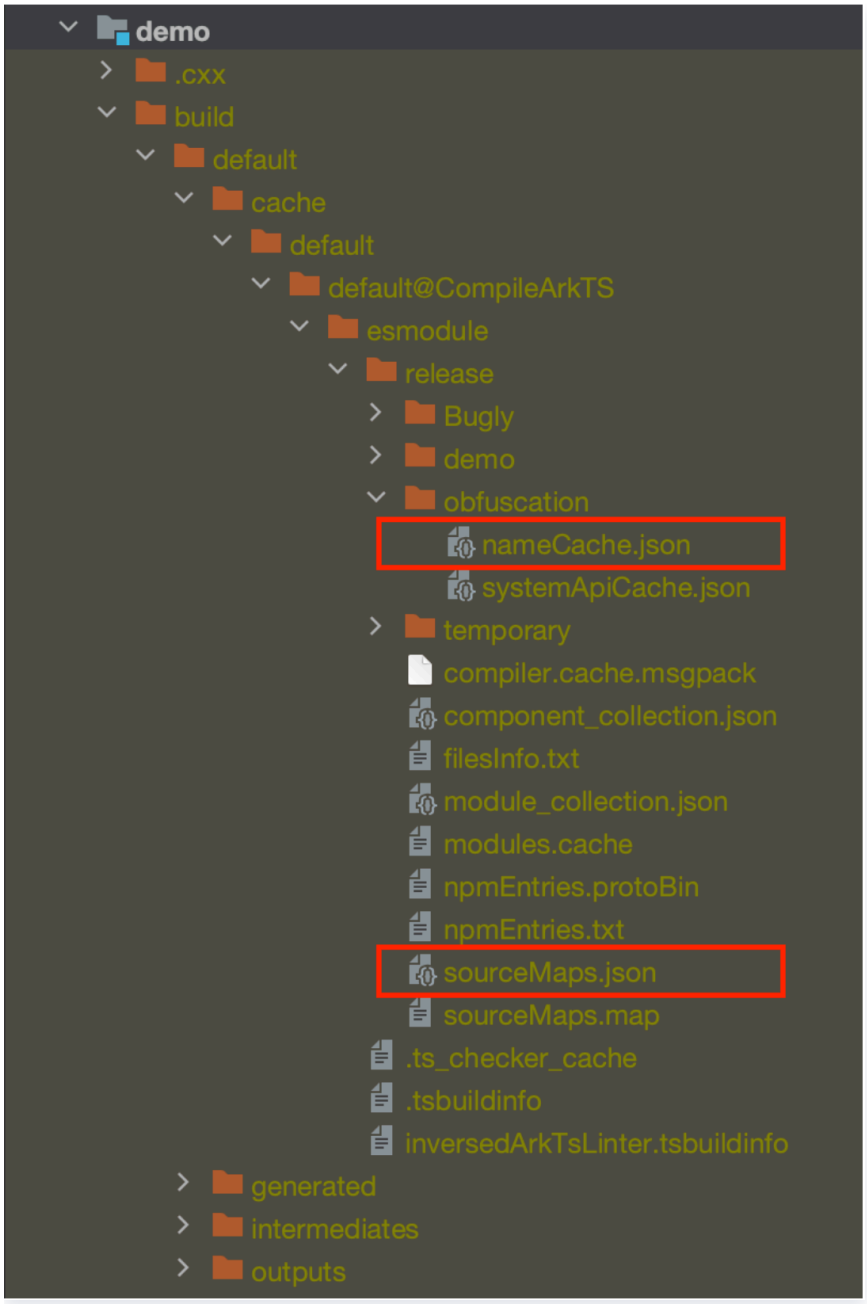
To facilitate locating the Debug SO files corresponding to crashes and restoring the stack, it is recommended to back up the Debug SO files each time you build or release a HAP version. Alternatively, you can use the platform's symbol table upload tool to upload the symbol tables to the platform.

By default, Harmony platform Native code is compiled using CMake. The compiled Debug SO files are located at `<Project Folder>/<Module>/build/default/intermediates/cmake/default/obj/<Architecture>/`



nameCache & SourceMaps file location

- By default, the nameCache file is located at: `<Project Folder>/<Module>/build/default/cache/default/default@CompileArkTs/esmodule/release/obfuscation/nameCache.JSON/`
- By default, the SourceMaps file is located at: `<Project Folder>/<Module>/build/default/cache/default/default@CompileArkTs/esmodule/release/SourceMaps.JSON/`



Reminder:

To facilitate locating the ts stack corresponding to crashes, it is recommended to back up the nameCache and SourceMaps files each time you build or release a HAP version. Alternatively, you can use the platform's symbol table upload tool to upload the symbol tables to the platform.

Matching SO Symbol Tables to Crashes

Harmony platform also uses SO UUID to match symbol table files. For matching rules and UUID extraction rules, see Android Symbol Tables.

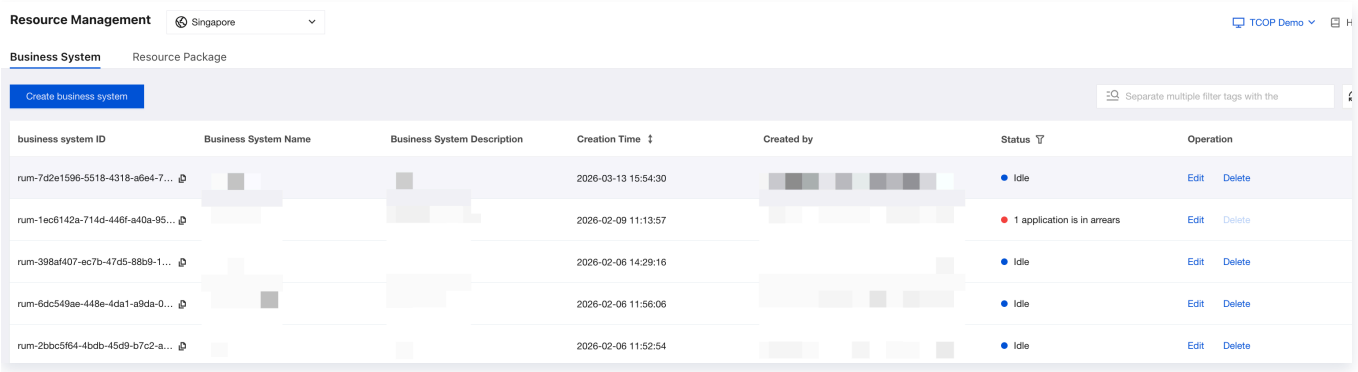
Resource Management

Last updated：2026-05-25 18:53:28

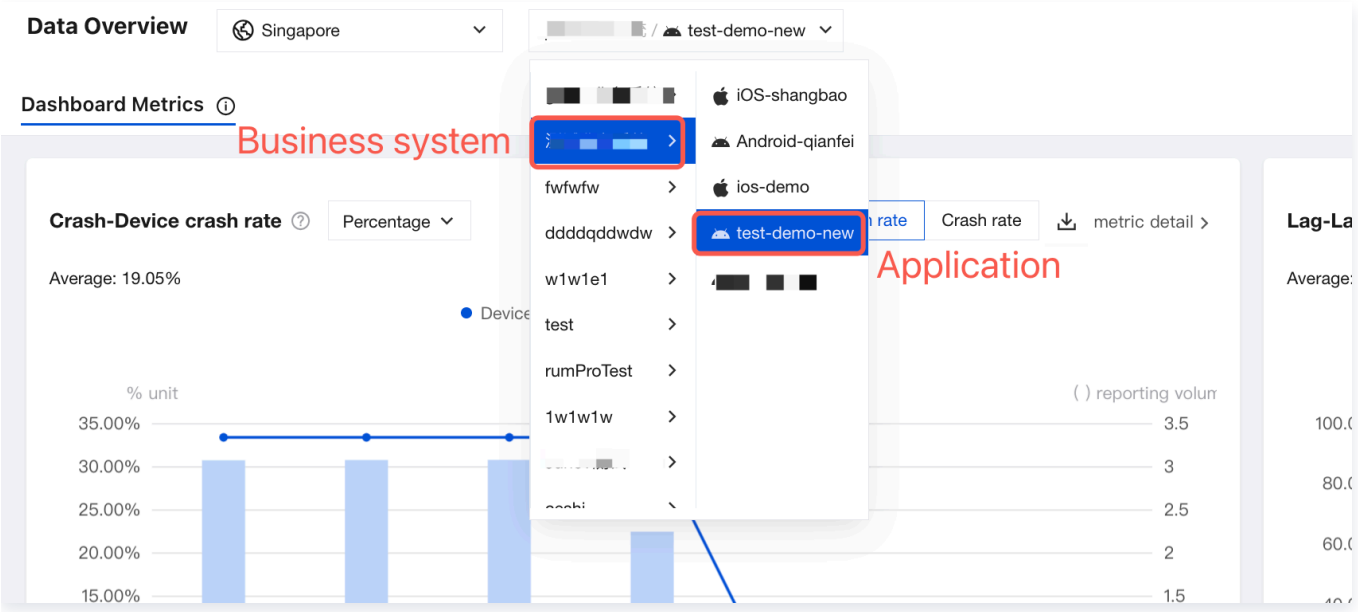
The resource management page of RUM Pro contains two modules: business systems and resource packages.

Business System

A business system is similar to the concept of an application group, where a single business system can contain multiple applications.

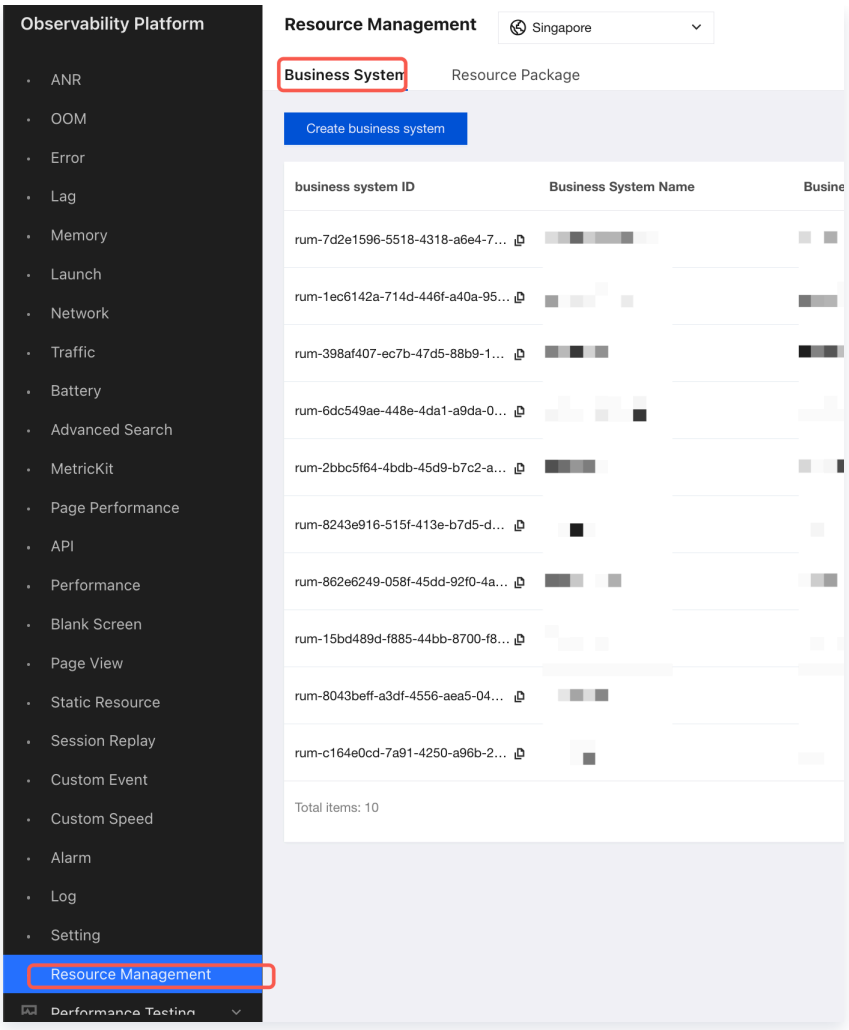


As shown in the figure below, when switching applications, you need to first select the **Business system**, then select the **application** under the business system.

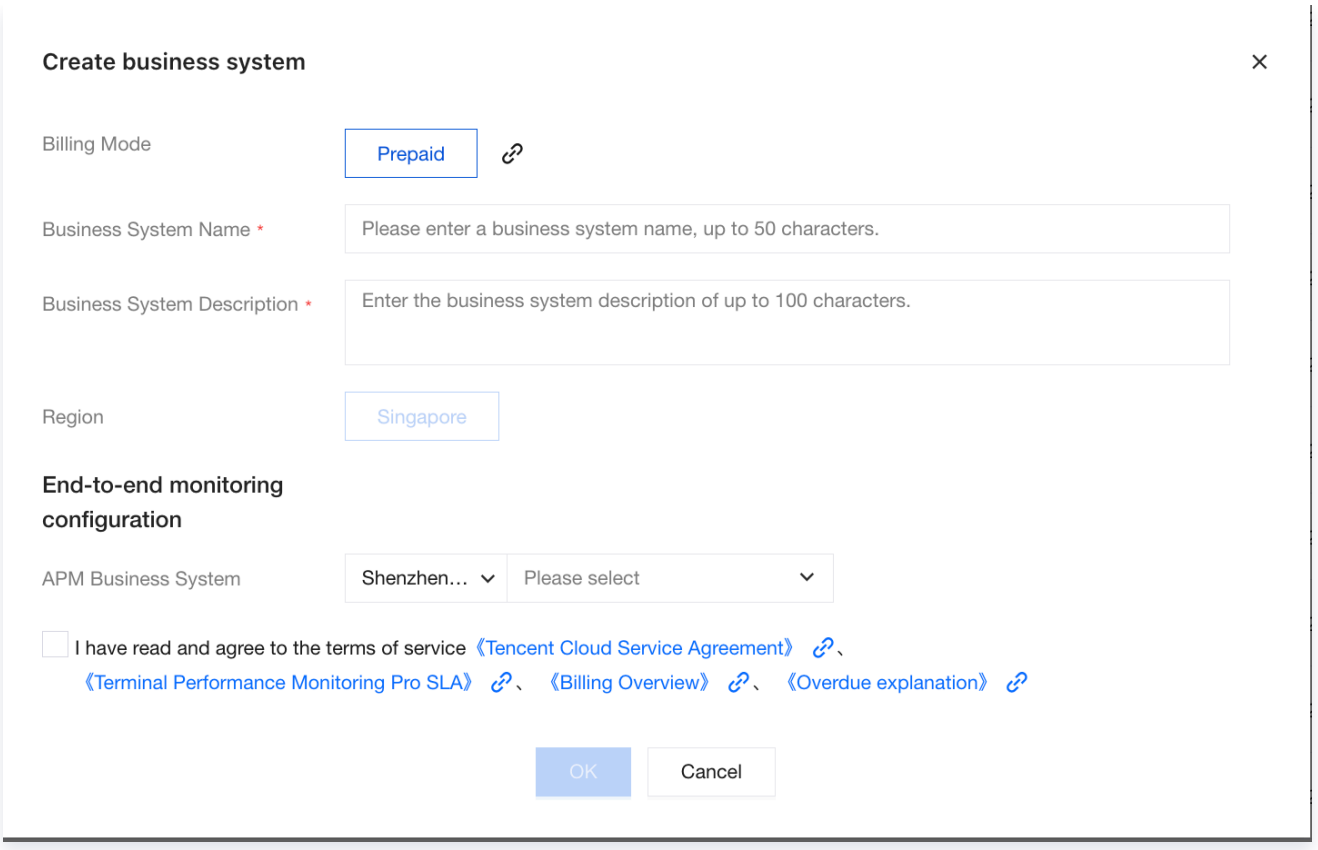


Creating Business Systems

1. Log in to the TCOP console.
2. In the left menu bar, select **Terminal Performance Monitoring Pro** > **Resource Management** > **Business System**.



3. On the Business System Management page, click **Create business system** to configure system information.



The following table describes the configuration information.

Configuration Item	Description
Billing Mode	Supports **Prepaid**.
Business System Name	The custom business system name.
Business System Description	Custom business system description.
Region	Data in different regions is isolated, and the business system cannot be changed after creation.

4. After the configuration is complete, select the service terms and then click **OK** to proceed.

Resource pack

RUM Pro adopts the prepaid mode. You need to purchase a resource pack and bind it to the application before enabling the service. For details, see Purchase Instructions.
It provides two viewing perspectives for resource packs:

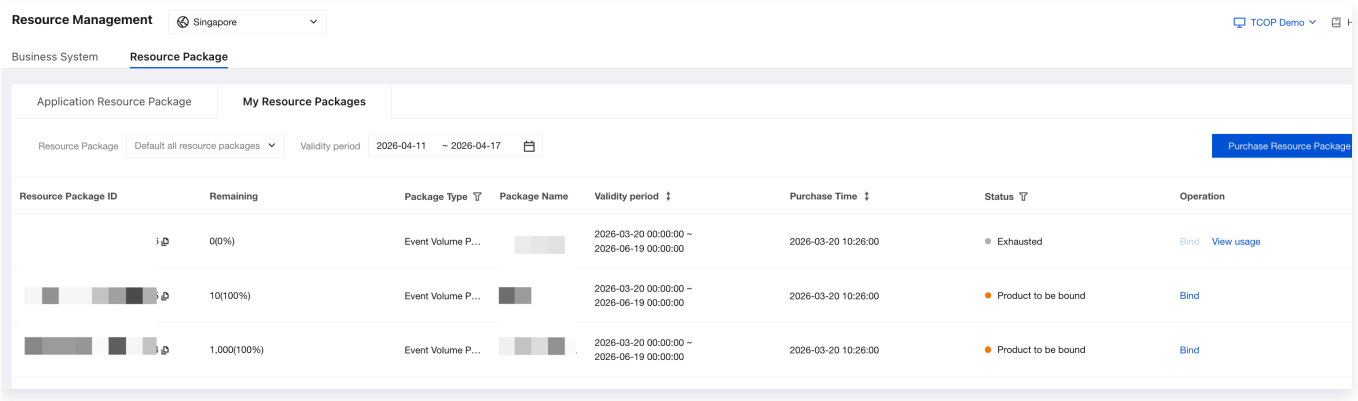
- My resource packs: View all resource packs you have purchased, including bound applications, usage, and supports binding/unbinding operations.
- Application resource packs: View the resource packs bound to specific applications and their usage details.

Purchasing a Resource Package

For details on purchasing resource packs, see Purchase Instructions.

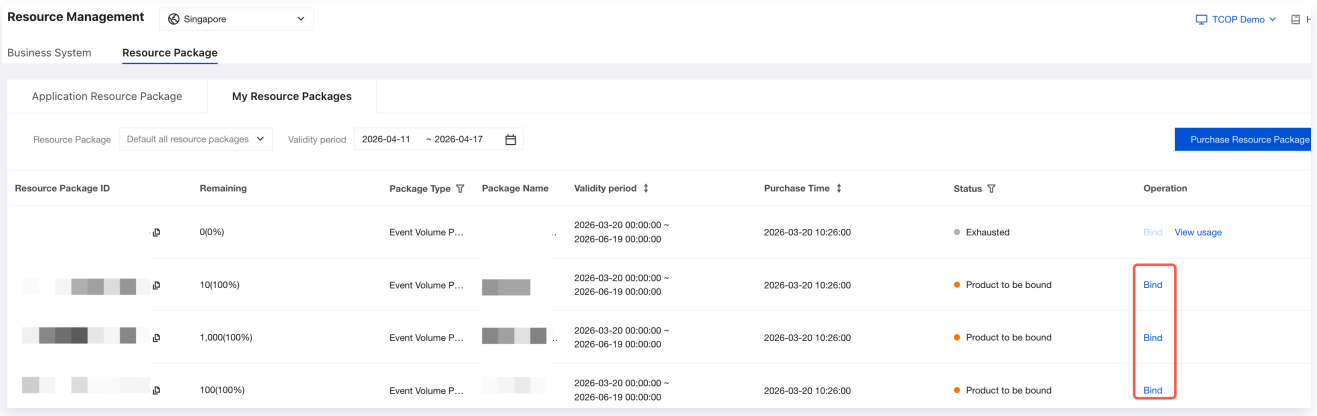
My Resource Packages

The My Resource Packages page allows you to view all resource packs purchased by the current user. Users can bind applications, unbind applications, and view usage.

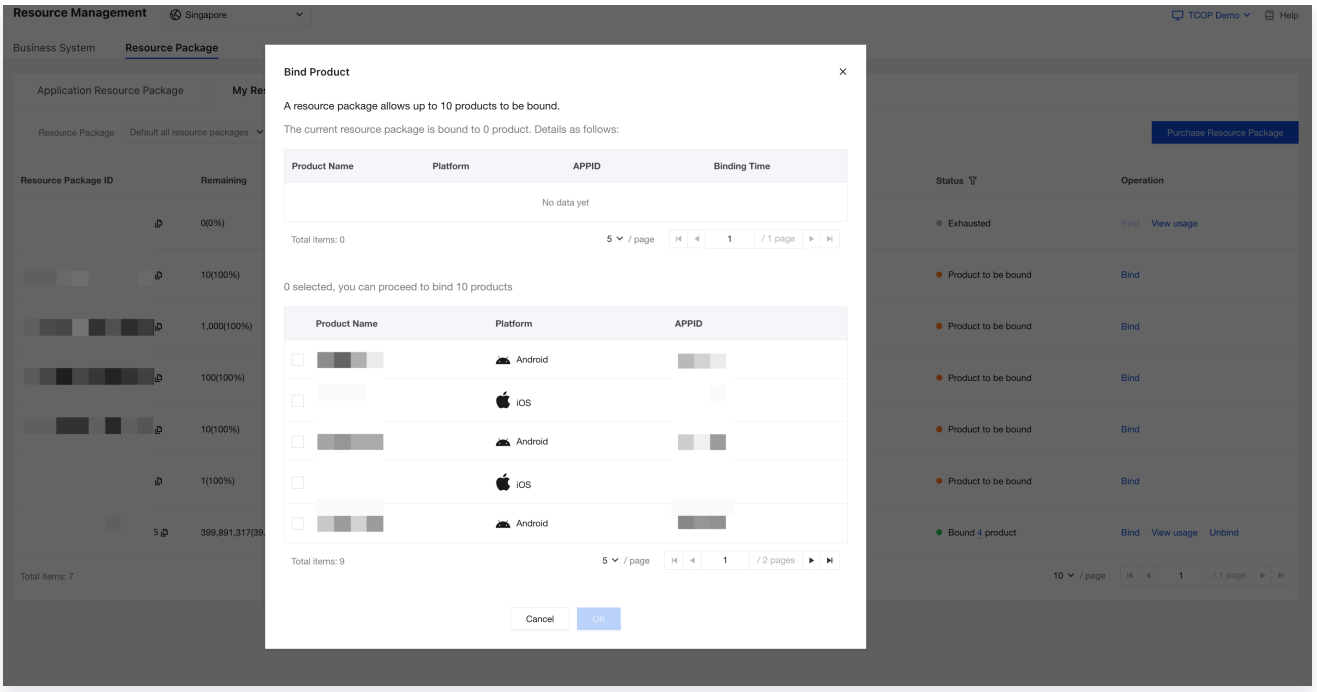


Binding an Application

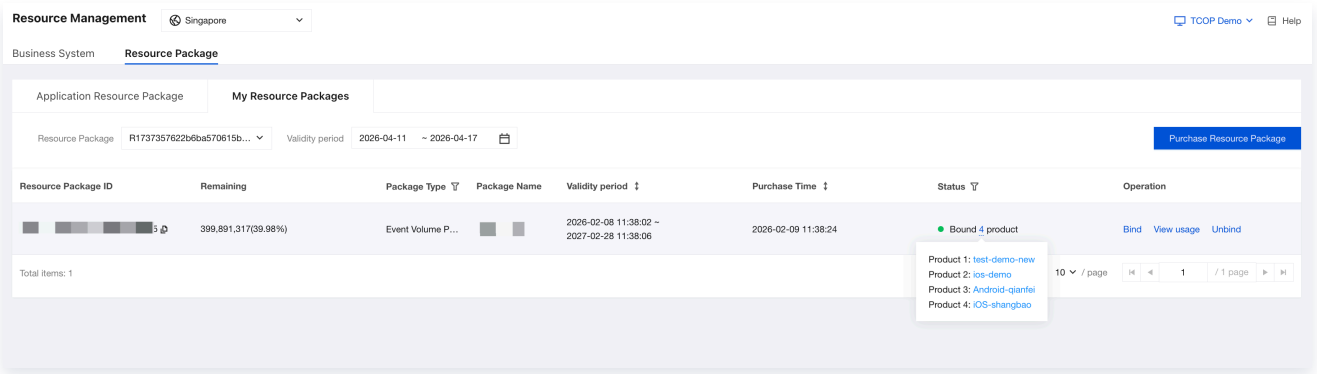
1. Select the resource pack to be bound and click **Bind**.



2. Select the applications to be bound. Each resource pack allows binding to a maximum of 10 applications.



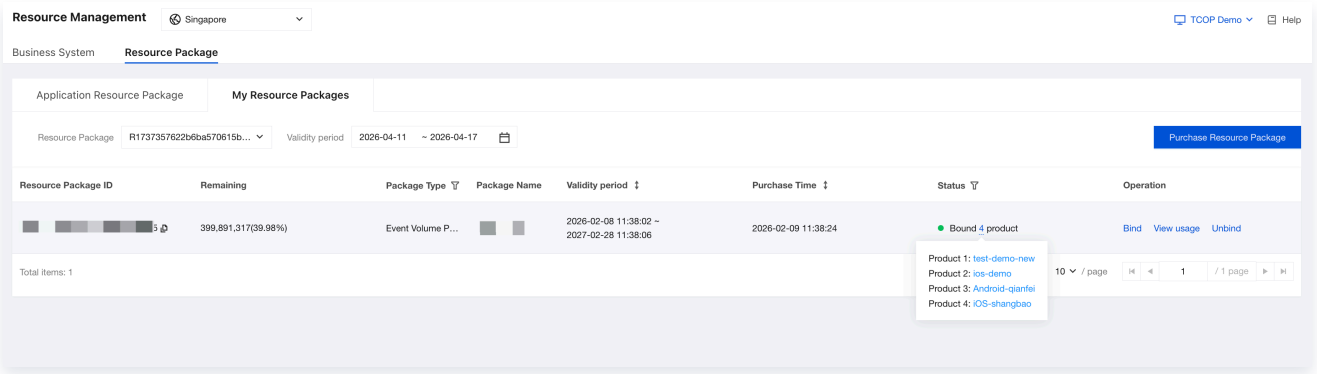
3. After binding is successful, you can view the applications bound to the resource.



Unbind Application

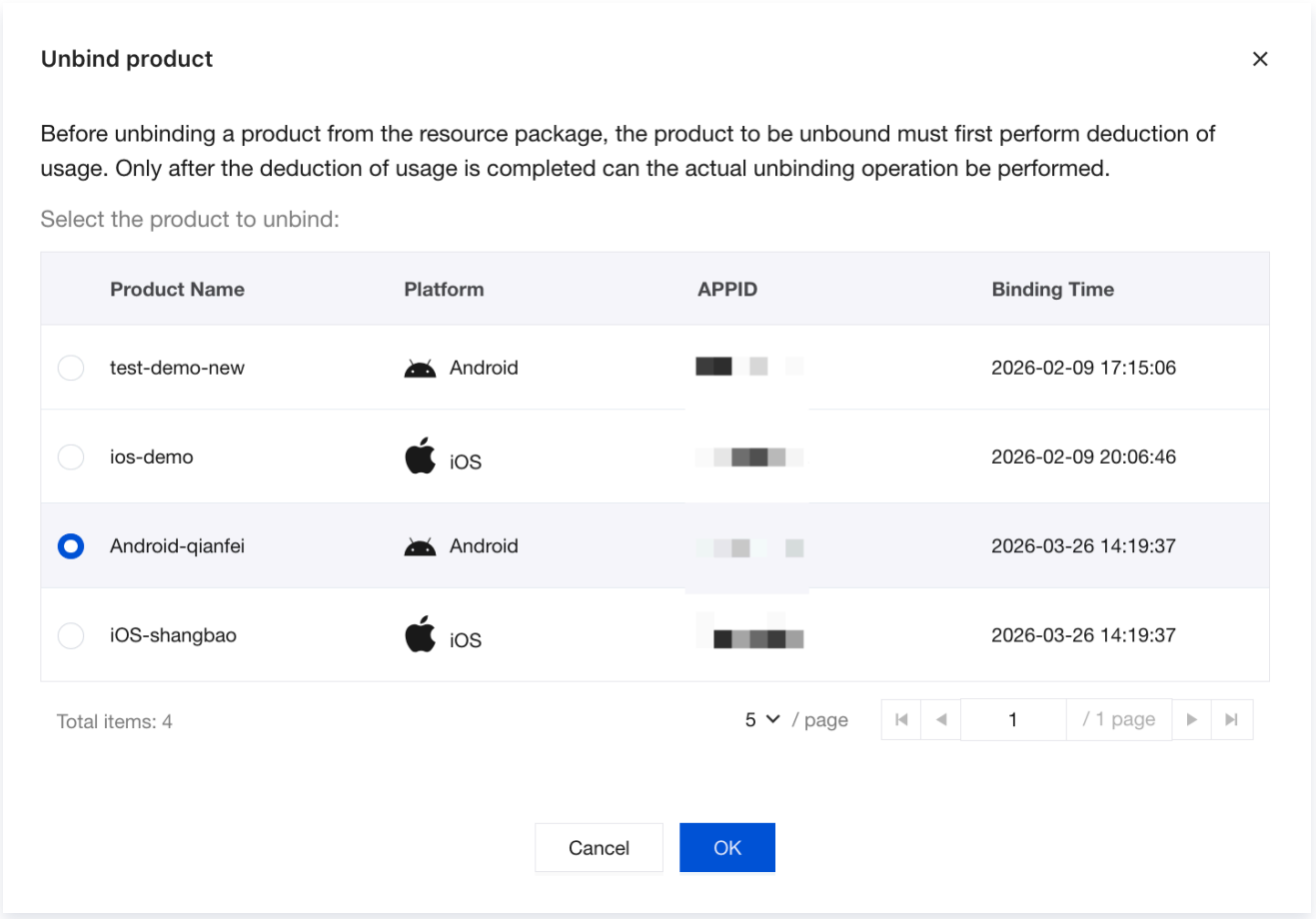
If you want to remove the binding relationship between an application and a resource pack, you can perform the unbinding operation on the resource pack. The procedure is as follows:

1. Click **Unbind** for the specified resource pack.

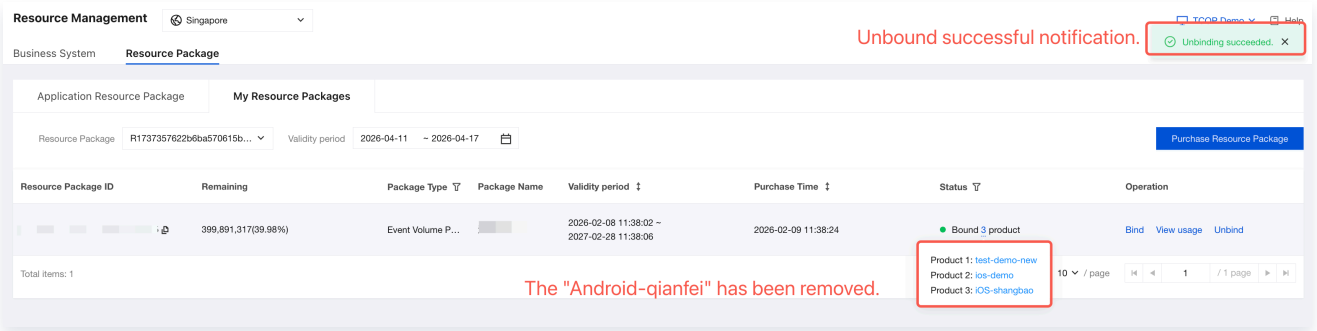


2. Select the application to be unbound. Only one application can be unbound at a time. Before unbinding an application from the resource pack, the application to be unbound must complete usage deduction.

Only after completing the usage deduction is the unbinding operation allowed to be actually performed.

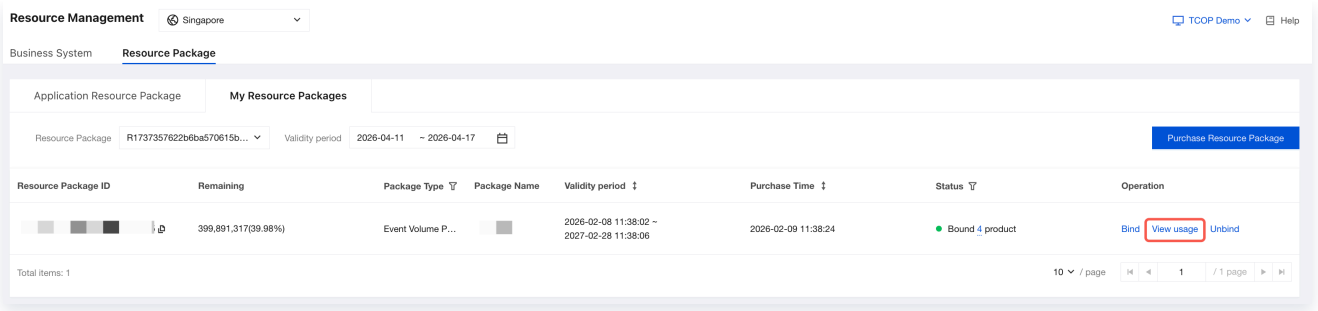


3. After successful unbinding, you will receive a notification. When you check the bound applications of the resource pack again, the unbound application has been removed.

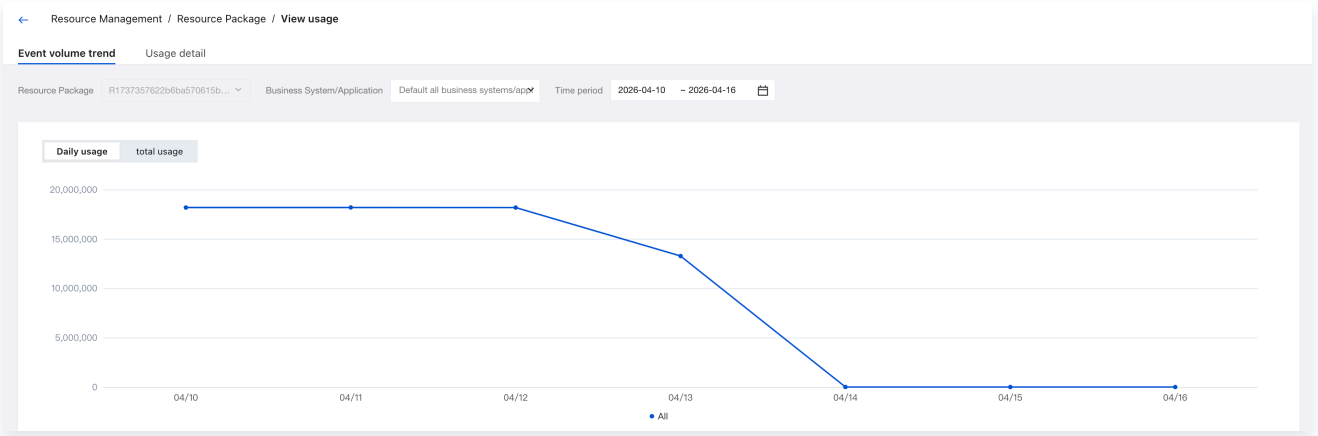


View usage

1. Click **View usage** for the resource pack to view its consumption details.



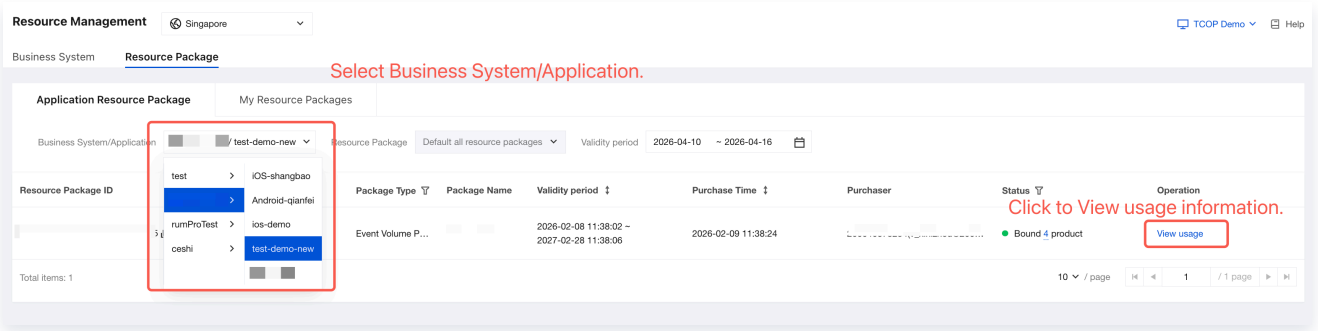
2. The current version supports viewing the consumption trend and usage details of resource packs.



Application Resource Package

Application resource packs allow you to view the usage of resource packs from the application's perspective.

1. Select the business system/application to view, and click **View usage**.



2. You can view the resource consumption of a specific application from the application's perspective.

Data Analysis Overview

Last updated：2026-05-25 18:53:28

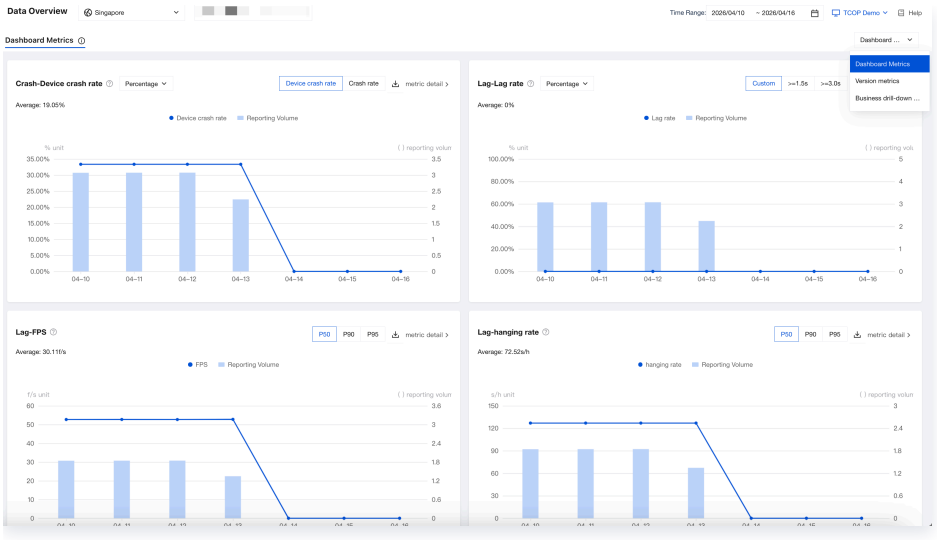
RUM Pro supports powerful data analysis capabilities, focusing on the following features:

- Supports rich search fields with multiple matching rules, enabling users to flexibly and freely query specific data.
- Supports flexible statistical analysis capabilities, enabling users to freely analyze trends and distributions under specific conditions.
- Supports an efficient condition comparison feature, assisting users to locate new issues and degradation issues with one click and easily identify the causes of abnormal metrics.
- Supports custom data analysis and feature integration, quickly correlating various abnormal issues to offer new approaches for solving complex problems.

Data Overview

The purpose of the Data Overview is to enable users to quickly understand the quality status of the application and its trends. The Data Overview includes **Dashboard Metrics** and **Version metrics**.

- **Dashboard Metrics**: Filters data for App versions with DAU less than 1% to eliminate the impact of Debug version data. By default, displays the dashboard data for the last 30 days. Users can adjust the time range, with support for viewing data up to the last 180 days.
- **Version metrics**: The platform determines the release date of application versions based on the number of connected devices, using the date when the number of connected devices for an application version first reaches or exceeds 500 as its release date. By default, it displays the metric data of the latest 5 released versions. Users can freely edit the application versions they wish to view, along with their corresponding release dates. Additionally, it provides three time range selection modes: Normal Mode, Release Mode, and Custom Mode.

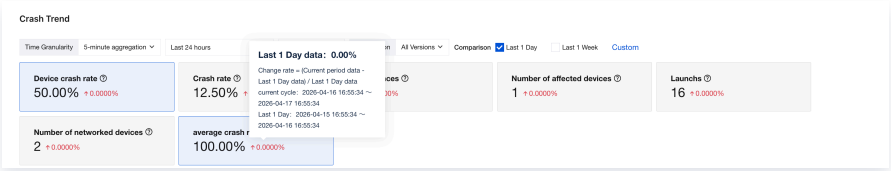


Exception Overview

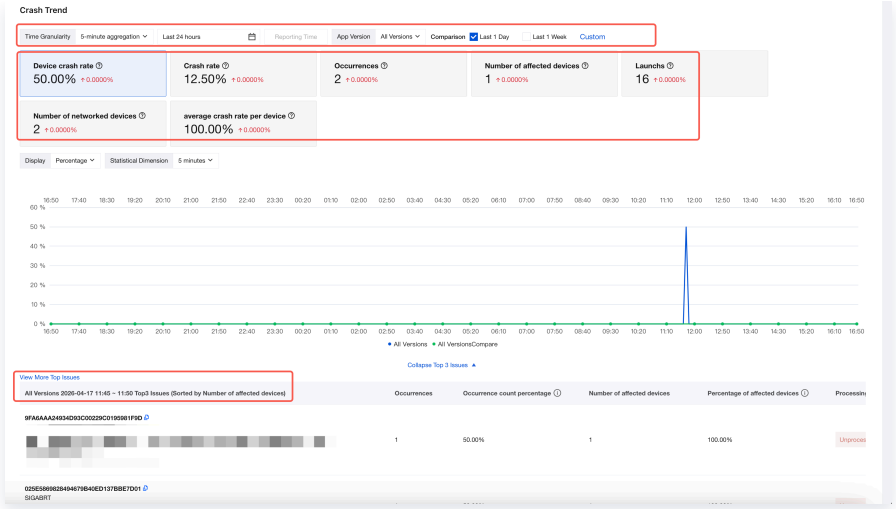
The crash, error, Android ANR, and OOM metric analysis page is an abnormal overview, aiming to provide users with general analysis capabilities for core quality metrics.
Abnormal Overview mainly includes: Today's Abnormalities, Abnormal Trend, Top3 Issues List, Abnormal Distribution, and Top20 Issues List. The following content uses the Crash Page as an example to introduce the display descriptions of Today's Abnormalities, Abnormal Trend, Top3 Issues List, Abnormal Distribution, and Top20 Issues List.

- Today's Abnormalities: Displays today's abnormal data in real time.

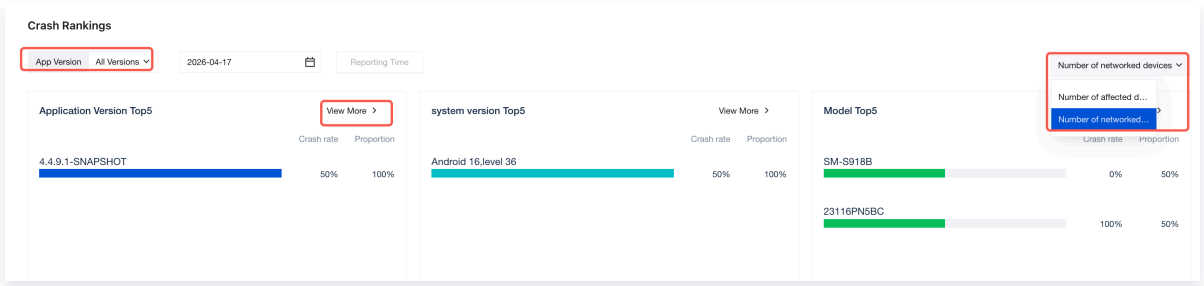
> **Note:**
> Day-over-day comparison = (Today's current data - Same-period data from yesterday) / Same-period data from yesterday



- Abnormal Trend: Displays the abnormal reporting trend at multiple aggregation granularities.
  - Supports three aggregation granularities: 5-minute aggregation, 1 Hour aggregation, and Aggregation by day.
  - Supports drill-down analysis and comparative analysis of App Versions.
  - Displays data across multiple statistical dimensions.
  - Supports viewing the Top3 issues of aggregated nodes at 5-minute aggregation and 1 Hour aggregation granularities.



- Abnormal Distribution: Displays the statistical distribution of three key fields: Application Version, system version, and Model. By default, displays the Top5 distribution data. Users can click **View More** to view all distribution data.



- Top20 Issues List: Displays the Top20 issues sorted by the number of affected devices on the statistical date. Users can click **View More Top Issues** to view additional Top issues in the issues list.

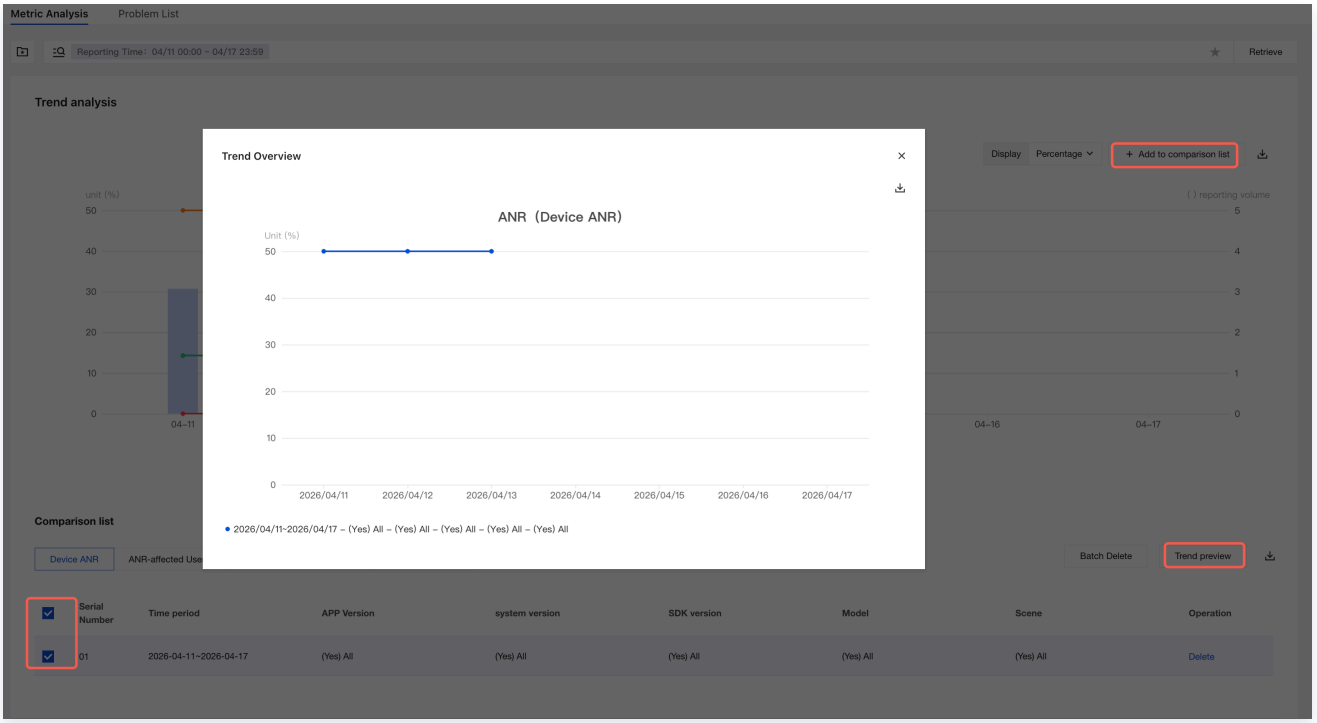
> **Note:**
>
> - Fluctuation in the number of affected devices = Percentage of affected devices on the current day – Percentage of affected devices on the previous day
> - Fluctuation in occurrence count = Percentage of occurrences on the current day – Percentage of occurrences on the previous day

Metric Analysis

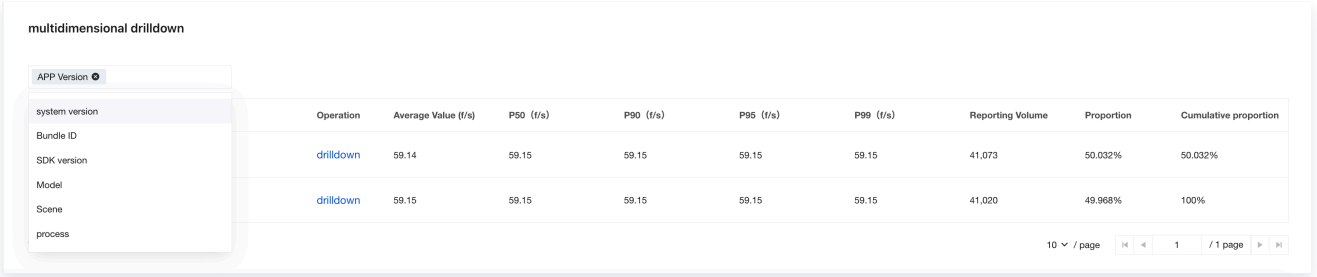
iOS ANR and FOOM, stutter, memory, and app launch time metrics analysis is primarily used to analyze the trend changes of corresponding module metrics.
Metric analysis includes trend analysis, comparative analysis, and multidimensional analysis. Using the ANR page as an example, the following describes the display descriptions for trend analysis, comparative analysis, and multidimensional analysis.

- Trend Analysis: Aims to display the trend changes of monitored item metrics under specified conditions.

- Comparative Analysis: Aims to compare the differences in changes between the trends of two sets of metrics.



- Multidimensional Analysis: Compares differences in metric data by combining multiple dimensions.



Queries

RUM Pro supports robust query capabilities. For details, see Query.

Issue List

RUM Pro supports users in analyzing various metrics through the exception overview and metrics analysis pages. In addition to metrics, the platform provides issue analysis for each monitored item via the issue list page.
The issue list supports robust search criteria, allowing users to freely analyze exception reporting under various conditions. The features of the issue list are divided into two main categories: Statistical Analysis and Top Issue Analysis.

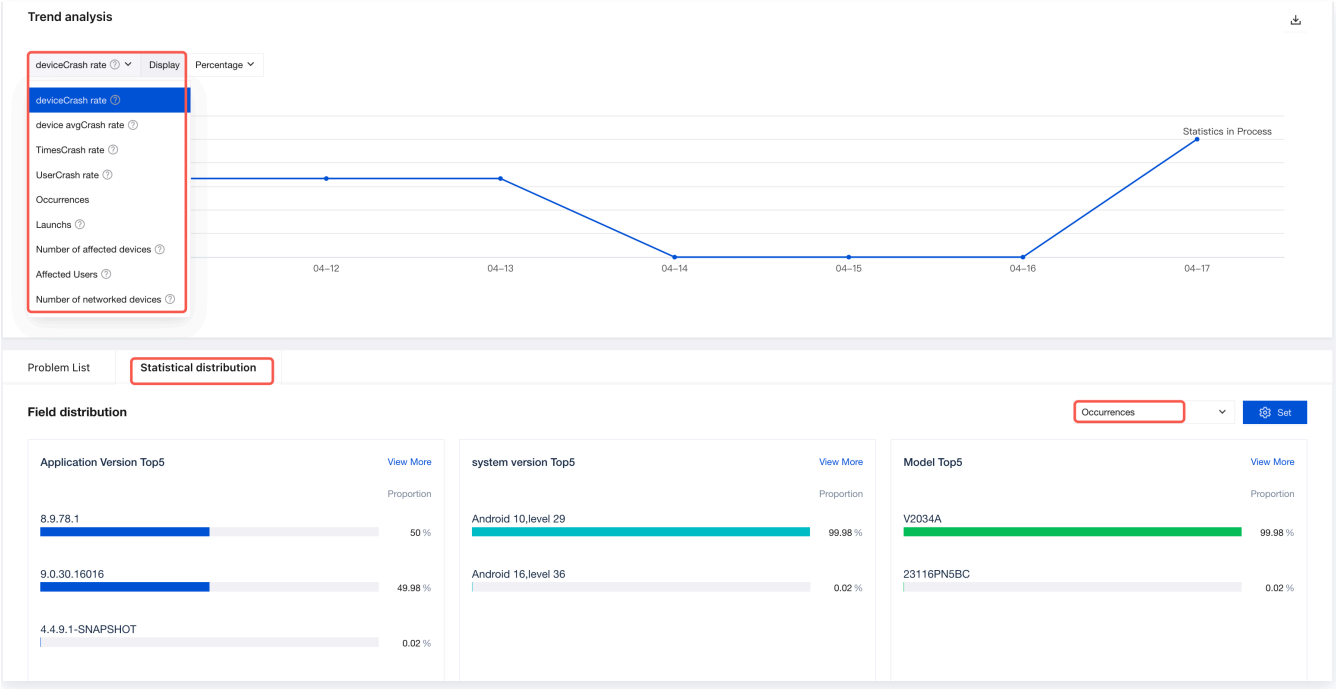
Statistical Analysis

Statistical analysis includes Trend analysis and Statistical distribution.

- For crash, ANR, OOM, and error issues, the list currently supports statistical analysis capabilities.
- The filter options for the issue list are grouped into two categories: one is individual case filters, which select cases meeting specific conditions such as reporting time, App version, user ID, error stack, etc.; the other is issue filters, such as issue tags, ticket status, assignee, processing status, etc.
- Statistical analysis applies to individual case filters. By setting search criteria, you can view trend data for cases meeting these conditions, such as **Occurrences**, **Number of affected devices**, **Affected Users**, etc.
- The connected device count is a special case, currently supporting only two filter criteria: time range and App version. For example: if a user selects the last 7 days, App version 4.5.6, and background state for crash cases, the occurrence count, affected user count, and affected device count all refer to cases meeting these three conditions simultaneously. However, for the connected device count, it only represents cases meeting both the last 7 days and App version 4.5.6 criteria, as it does not support foreground/background state filtering.
- The **deviceCrash rate**, **TimesCrash rate**, and **UserCrash rate** in trend analysis refer to the number of abnormal cases meeting the criteria / the number of connected reports meeting the criteria. Note that the connected reports metric only supports two filter conditions: time range and App version.
- Statistical analysis supports not only the **Application Version**, **system version**, and **Model** fields but also additional fields such as Foreground or background status, Error process, Error thread, Vendor, CPU

architecture, ROOT or Not, etc.

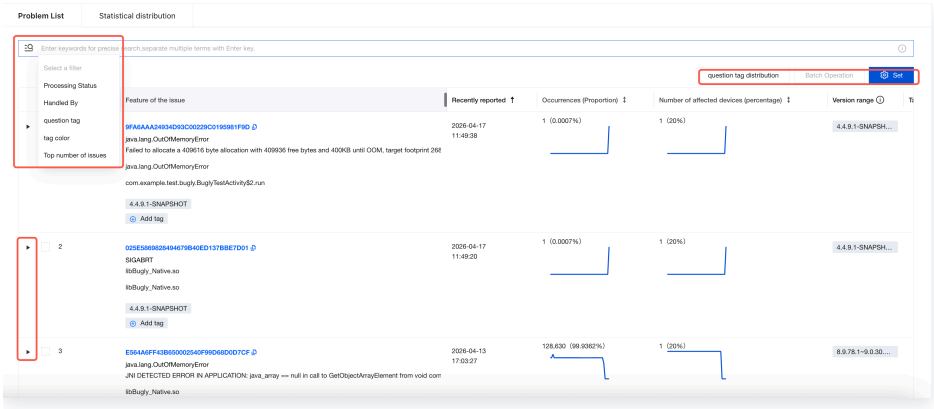
- Users can click to view the distribution details of a specific field, showing the percentage distribution of occurrence count, affected devices, and affected users.



Top Issue Analysis

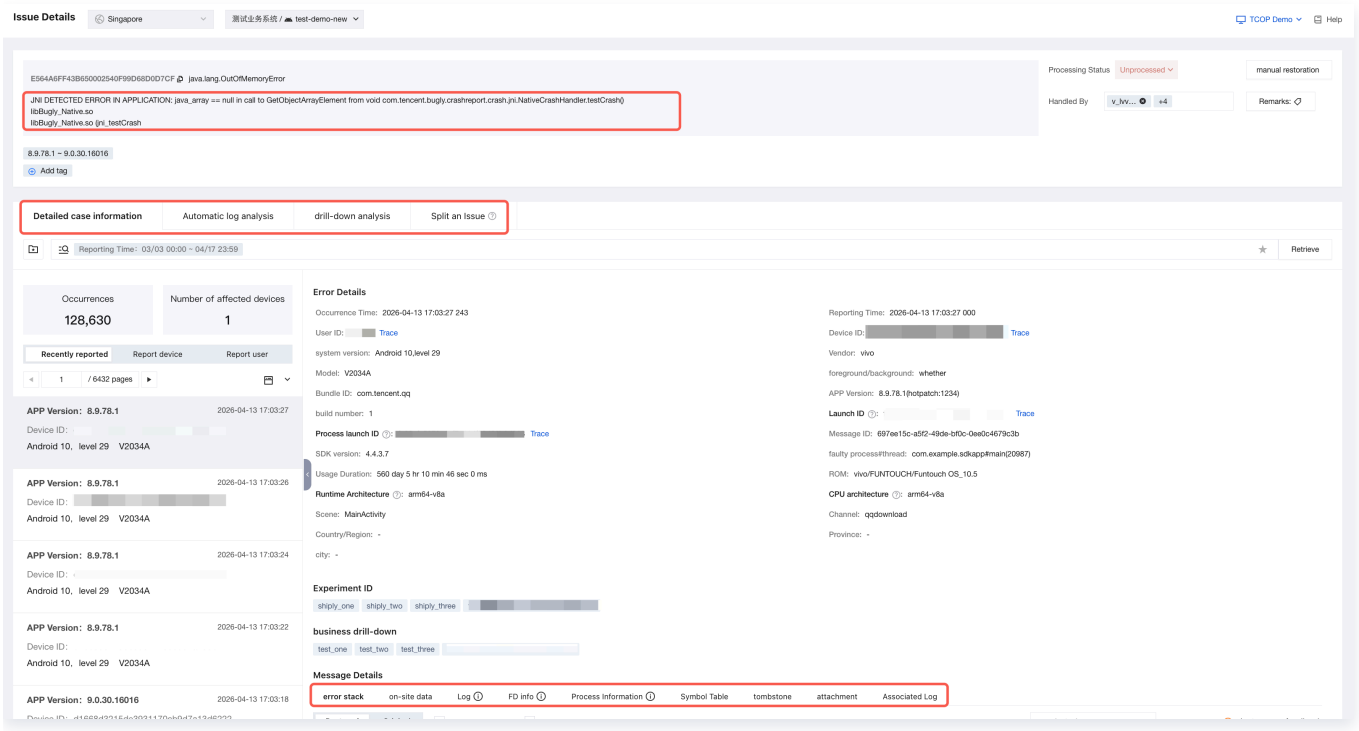
Based on the search criteria set by users, the system searches for abnormal reporting cases that meet the conditions and displays aggregated Issues according to problem characteristics. Issues are shown by default sorted by most recently reported, while allowing users to switch to other sorting options.

- Users can search for specific issues meeting the criteria based on the result list. When users do not need to search based on issues, they can collapse the issue filter options.
- Supports users exporting the result list.
- Allows users to freely customize the column headers of the result list.
- Version Tag: Displays the version range in which the issue occurs. **This result is derived from case data collected over the last 90 days**.
- Supports users adding custom tags to facilitate problem categorization. **question tags** are searchable.
- By clicking the triangle button, users can view the detailed exception stack trace, or directly click the issue ID to go to the issue details for in-depth analysis.



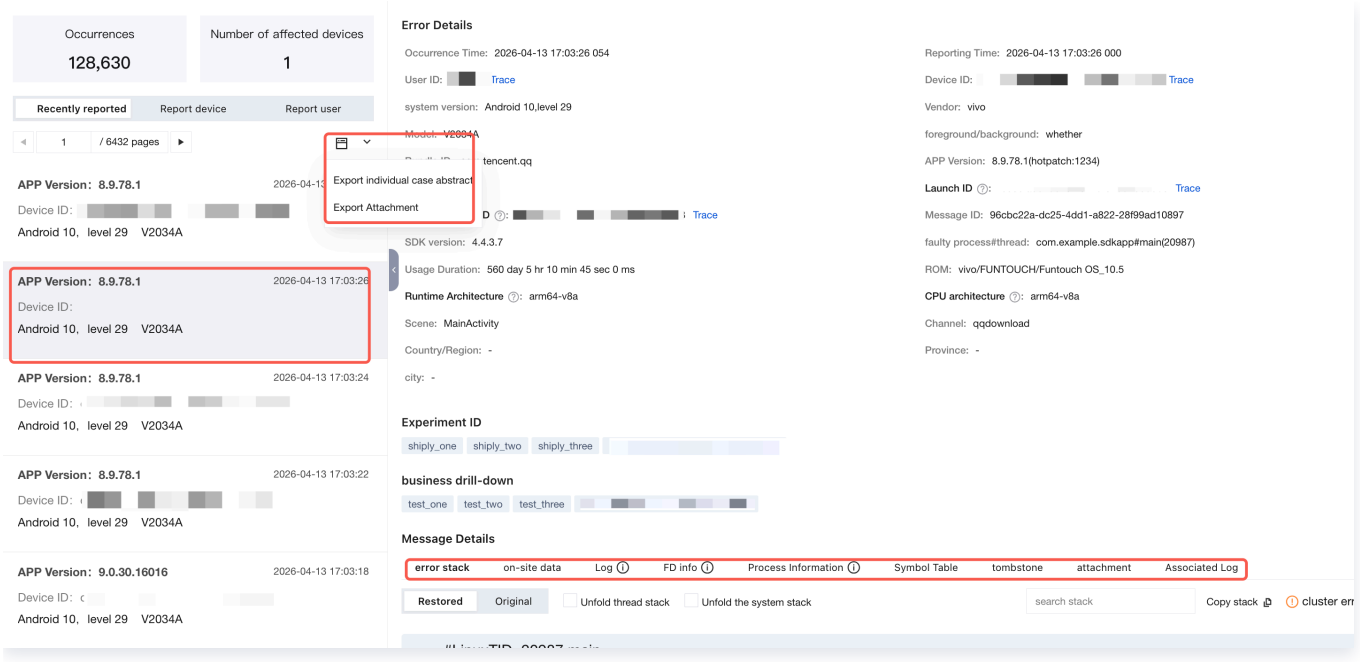
Issue Details

On the issue list page, clicking the issue ID will go to the issue details. The issue details focus on providing a detailed analysis of a specific Issue.



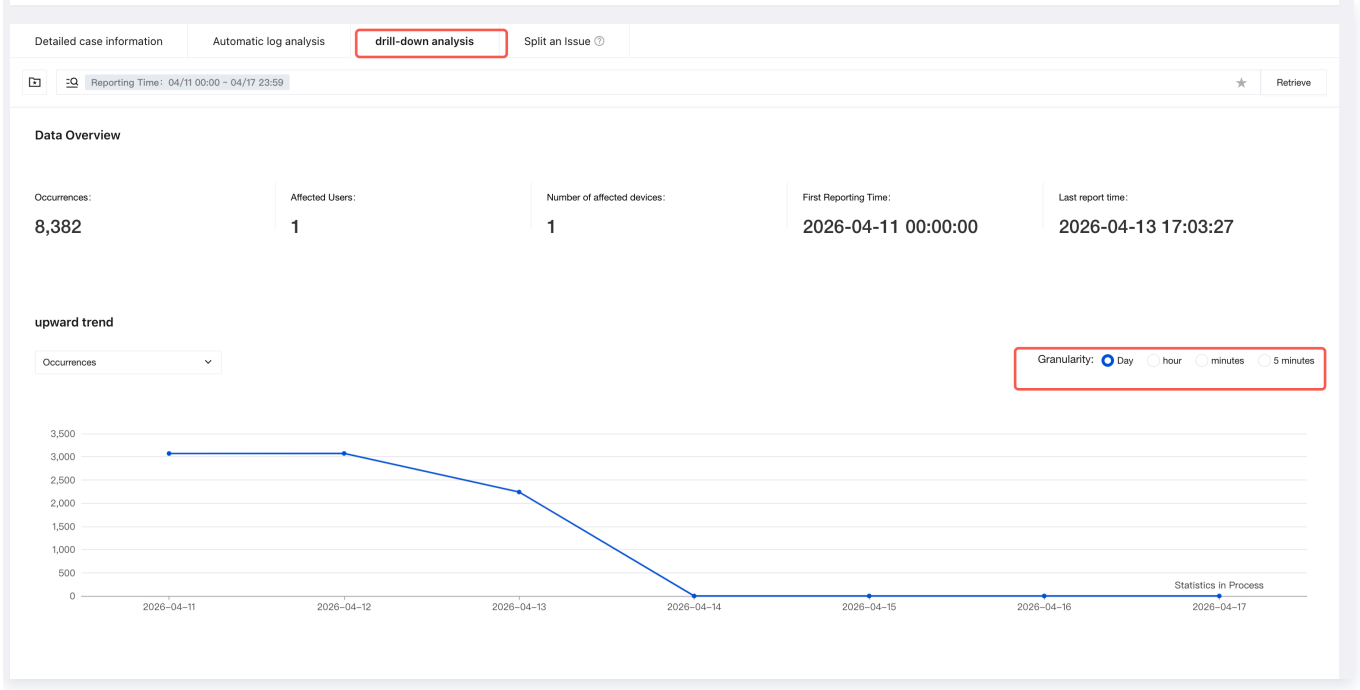
Case Details

The core purpose of case details is to help users analyze the abnormal scene of a specific Issue. Users can query cases under specified conditions, which contain rich on-site information.



Drill-down analysis

- Provides rich filter criteria, enabling users to analyze data under specified conditions.
- Upward Trend: Supports multi-granularity aggregated data down to the minute level, enabling more real-time analysis.
- Statistical Distribution: Provides statistical analysis capabilities for multiple fields, enabling more comprehensive analysis.
- Error stack trace analysis supports user customization.



business drill-down

Last updated：2026-05-25 18:53:28

Business drill-down is a data analysis method that starts from macro-level business data, employs multi-dimensional and layered detailed analysis to pinpoint the root causes of data anomalies or driving factors behind outstanding performance, enabling precise attribution and business optimization. This article introduces the use cases and methodologies of business drill-down.

Use Cases

RUM Pro has added a **business drill-down** field, allowing users to tag data with multiple sets of labels, and simultaneously supports drill-down analysis on individual cases (numerator) and the number of connected devices (denominator). For example, the following usage scenarios:

- To categorize device models (low-end, mid-range, high-end) into tiers, enabling separate viewing of quality data for each tier.
- To categorize users (regular users and VIP users) into regular and VIP groups, aiming to determine the crash rate of VIP users.
- To understand the distribution channels used by the application and determine the crash rate for a specific channel.

> **Note:**
> In the SDK configuration, users only need to mark the current device, device model tier, whether they are VIP users, and the download channel via the API. For example:
> - High-end devices, VIP users, Huawei AppGallery ==> setTestLabels("HighPerformanceDevice|VIP|HWAppStore")
> - Low-end devices, regular users, official website ==> setTestLabels("LowPerformanceDevice|Normal|Official")

Operation Steps

Step 1: Configure business drill-down in the SDK

- Android SDK:

```
/**
 * Configures business drill-down tags. To configure multiple tags,
separate them with a vertical bar (|).
 * A maximum of 30 tags can be configured. If more than 30 tags are
specified, only the first 30 tags are used.
```

```
 * Each tag cannot exceed 1024 characters in length. Tags longer than
this limit cannot be added successfully.
 * Example: "test_one|test_two|test_three".
 * SDK version: 4.4.1+
 * @param labels Business drill-down tags.
 */
public static void setTestLabels(String labels);
```

For specific operation steps, see Android SDK Access Guide.

- iOS SDK:

```
/**
 * To update business drill-down tags, call this method after the
RumPro sdk initialization is complete (i.e., after the setup
completeHandler callback). Otherwise, data loss may occur.
 * SDK version: 2.7.53.3+
 * @params tagArr String array (max length: 30, each string max
length: 1024 bytes)
 */
+ (void)updateTestTags:(NSArray<NSString *> *)tagArr;
```

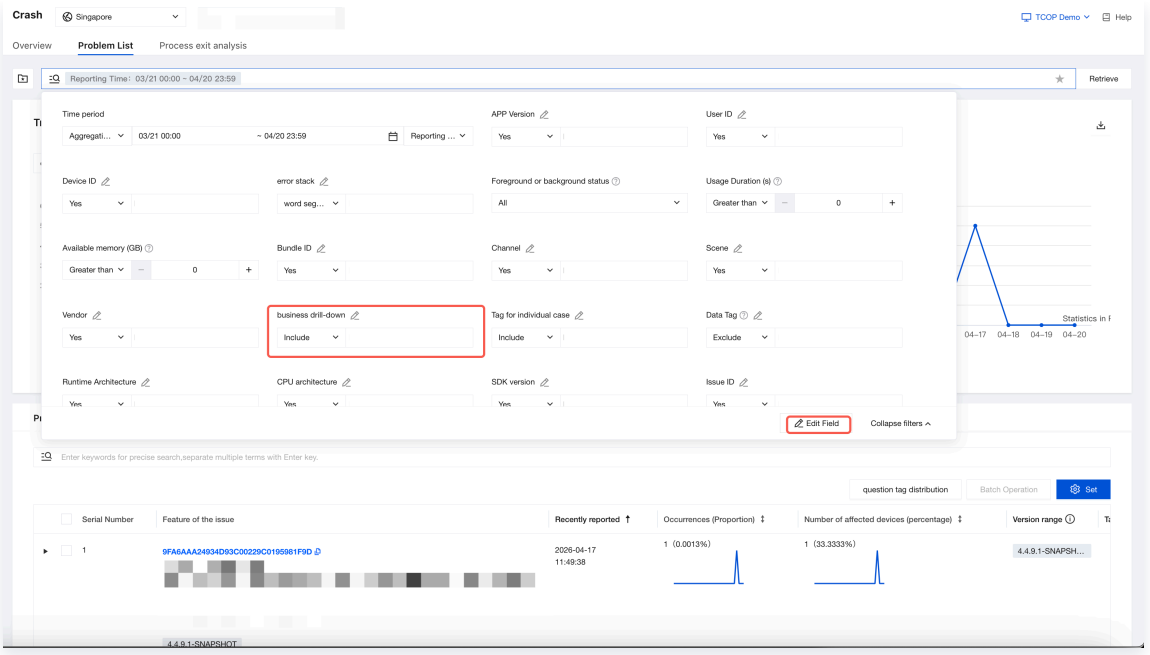
For specific operation steps, see iOS SDK Access Guide.

Step 2: Use business drill-down

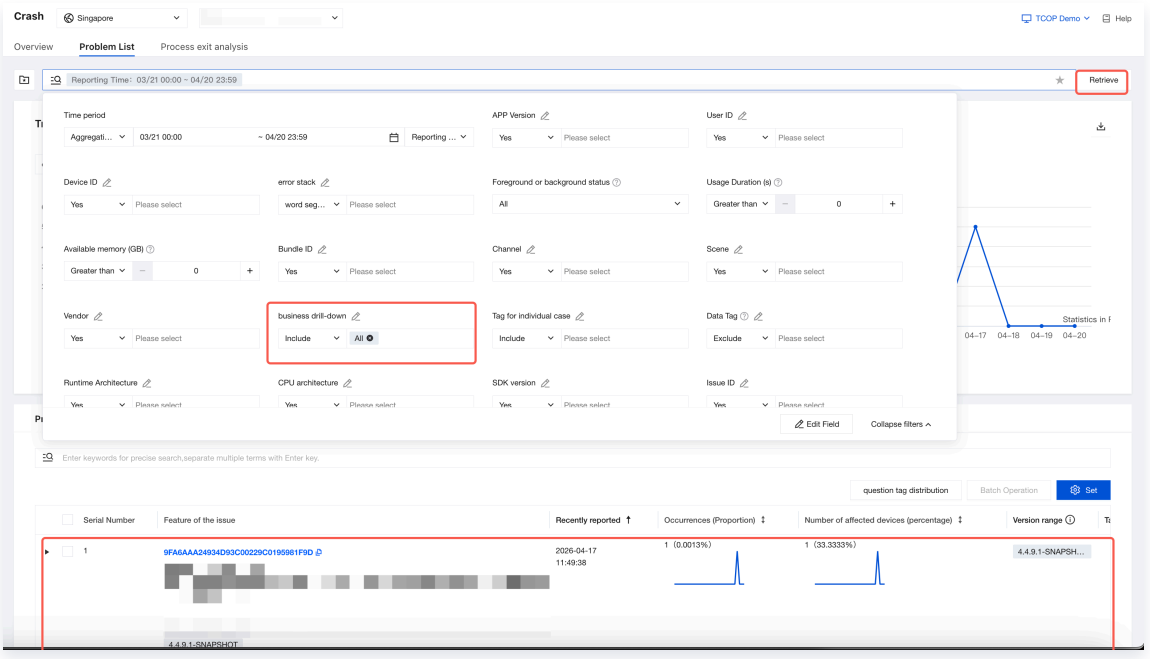
The following content provides usage instructions for business drill-down using the crash page as an example. Other pages can refer to the following content.

Search Analysis

1. By clicking **Edit Field** to add the **business drill-down** filter option.

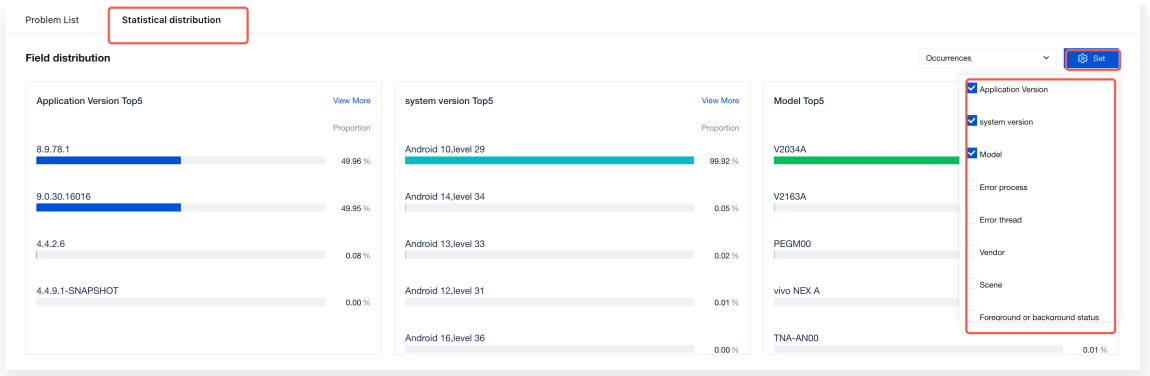


2. Select the desired Tag data and click **Retrieve** to retrieve data matching the business drill-down filter criteria.

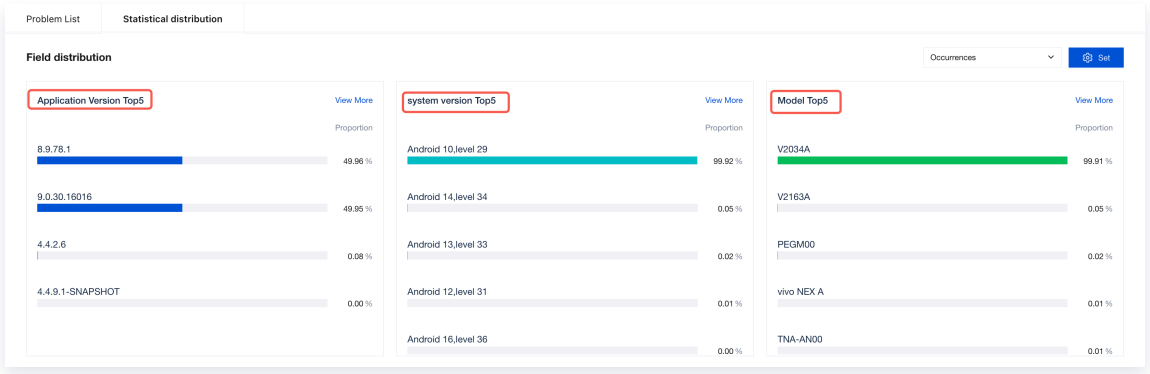


Statistical distribution

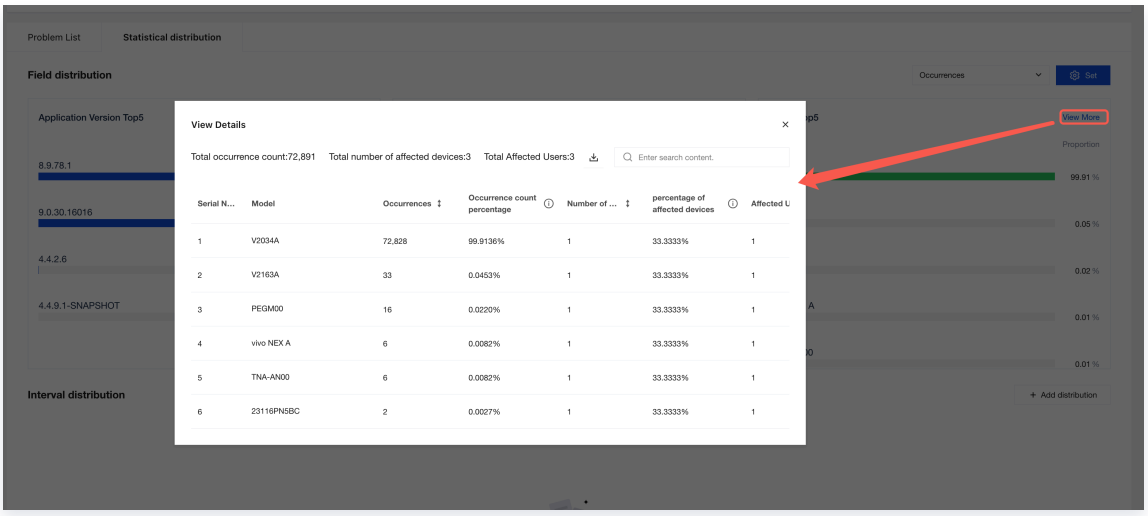
1. By clicking **Set**, you can add the statistical distribution for **business drill-down**.



2. View the Top5 distribution of business drill-down.



3. Click **View More** to view the complete **Business Drill-down** data.



Query

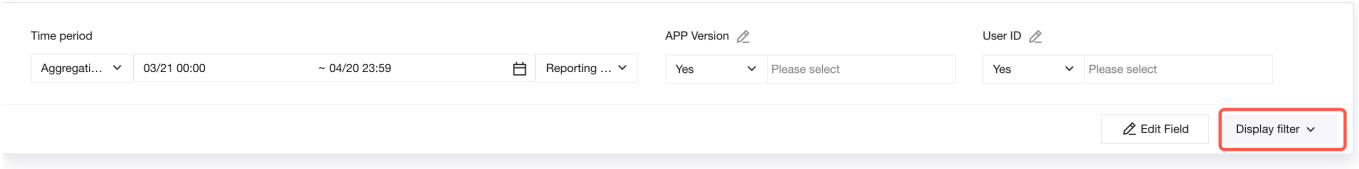
Last updated：2026-05-25 18:53:28

This article aims to explain tips on using the search feature in pages such as the issue list and issue details.

Manage filter fields

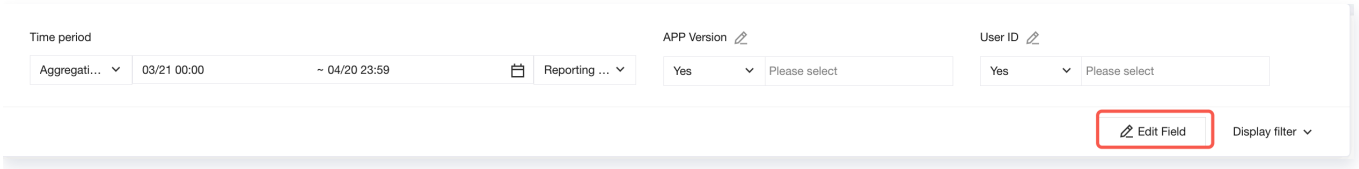
Expand filter options and collapse filter options

After opening the page, by default, it is in the **collapsed** state. You can click **Display filter** to display all **added** filter fields.



Edit the field

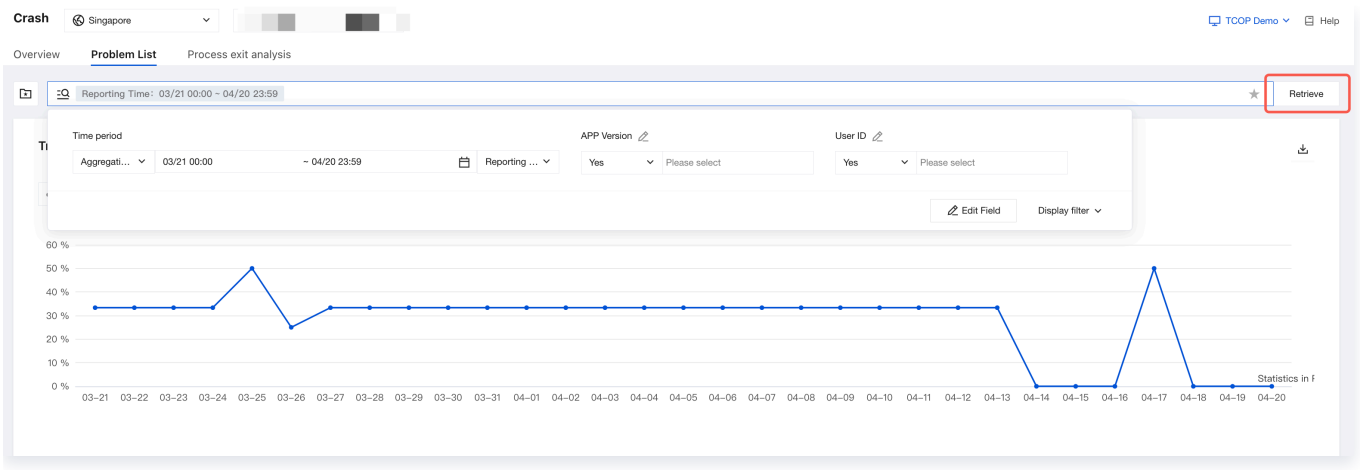
By default, all filter fields supported on the current page are displayed. Users can click **Edit Field** to freely adjust the fields they wish to display.



> **Note:**
> After the user adjusts the displayed filter options, the platform saves the user's settings locally in the browser. If the cache is not cleared, the next time they go to the same page, the fields displayed are those set last time.

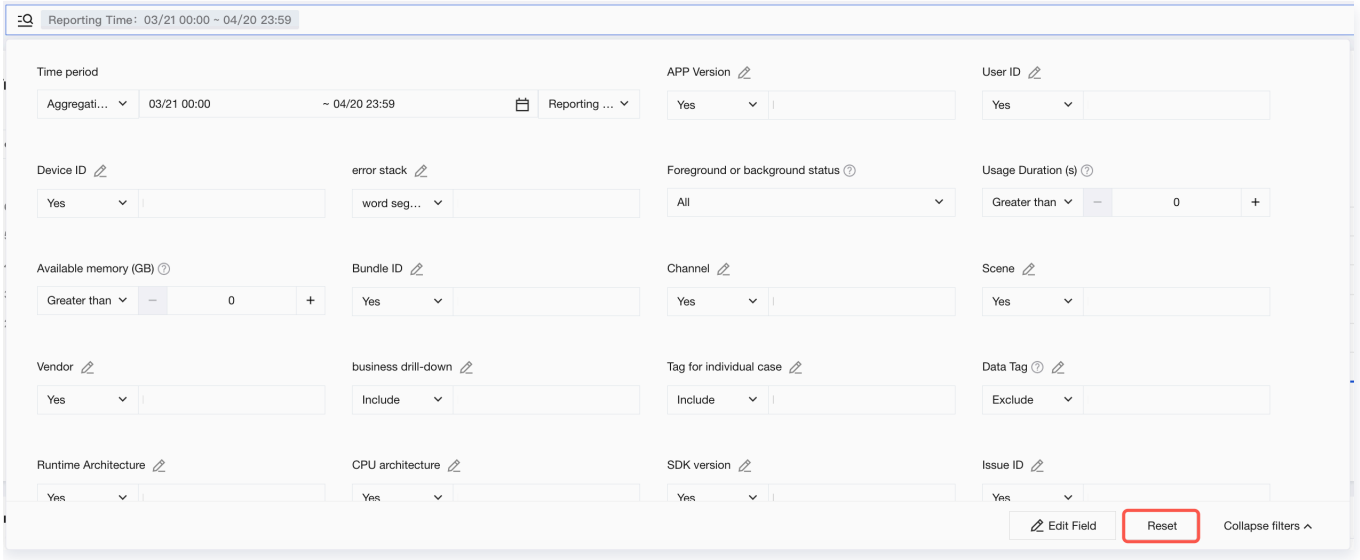
Search

After adjusting the filter fields, the search does not take effect immediately; you need to manually click **Retrieve** to actually execute the search operation.



Reset

You can clear all filter options by clicking **Reset** on the page.

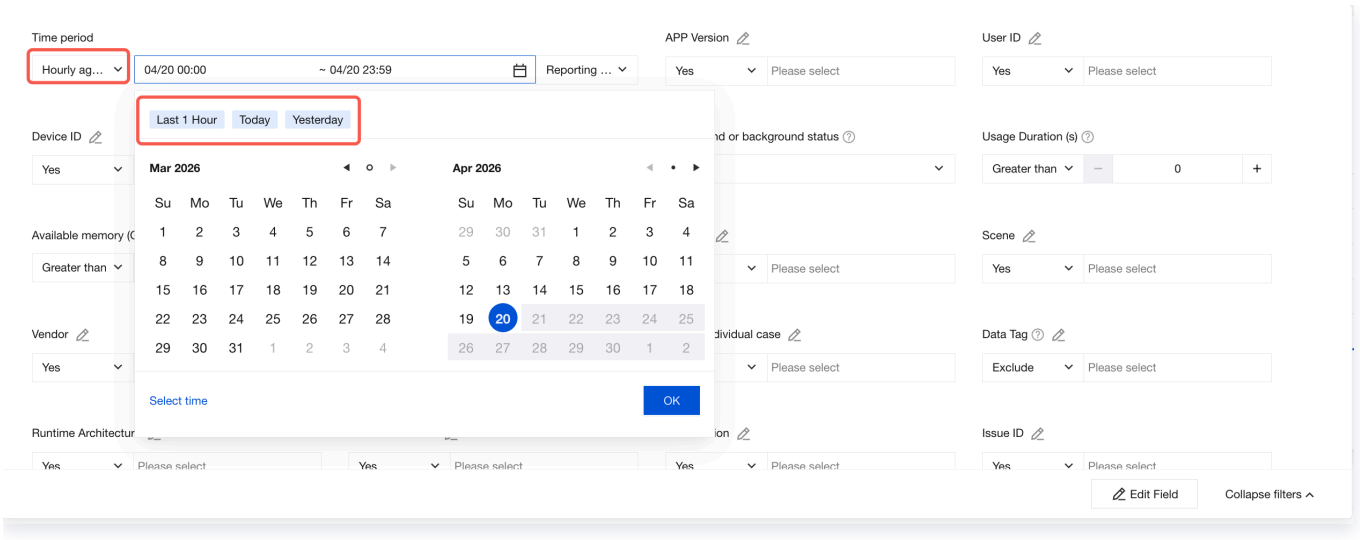


Filter Field Type

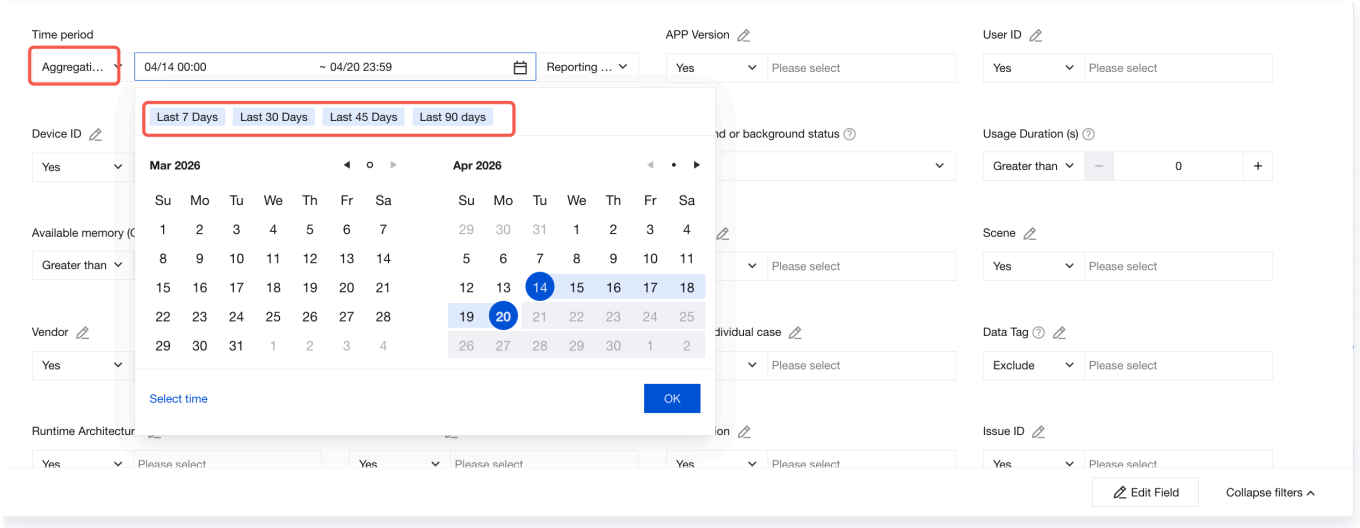
The filter options supported by the platform can be categorized into these types: Time Range, Version Type, Numeric Type, String Type, String Array Type, Enumerable Type, Special Type. Different types support different search modes, with most search modes supporting multiple keywords. Below are detailed descriptions of these filter types.

Time Range

Time Range supports five aggregation granularities: 5-minute, hour, day, week, and month.
Among these, 5-minute and hourly aggregations only support data from the last 48 hours, including custom, Last 1 Hour, Today, and Yesterday modes.



The day, week, and month aggregation granularities support modes such as custom, Last 7 Days, Last 30 Days, Last 45 Days and last 90 days.



Edition type

The version type fields include filter fields such as App version, SDK version, and system version. They support **Yes**, **No**, **Greater than**, **Greater than or equal to**, **Less than**, **Less Than or Equal to**, **interval**, **Match**, **Not Matched**, **Empty**, **Not Empty**, **Match at the beginning**, **Beginning mismatch**, **Ends With Match**, **End mismatch**, and **Regular Expression** search modes.
Version type fields essentially store a string that conforms to version specifications. Therefore, version type filter options possess all search modes available for both numeric type and string type filter options.

> **Version Comparison:**
> The platform follows industry-recommended version formats, i.e., "major.minor.patch.build". It first parses the version field and normalizes it into four parts (A.B.C.D), padding with zeros if fewer than four components exist. Version comparison follows the priority rule: major version (A) > minor

version (B) > patch (C) > build (D). For example: "4.5.2.1" > "4.5.2.0" > "4.5.1.99" > "4.4.2.100". This parsing and comparison logic also applies to system versions, e.g., "Android 7.1.2,level 25" > "Android 6.0.1,level 23" > "Android 5.1,level 22" > "Android 5.0.2,level 21".
Here are the detailed comparison rules:

1. When determining priority levels, the version must (MUST) be split into major version, minor version, patch version, and pre-release version in sequence for comparison (build metadata is not included in this comparison list).
2. Compare each component from left to right sequentially, and the first differing value determines the priority level: major version, minor version, and patch version are compared numerically. For example: 1.0.0 < 2.0.0 < 2.1.0.
3. When the major version, minor version, and patch are identical, pre-release versions have lower priority. For example: 1.0.0-alpha < 1.0.0.
4. When two pre-release versions share the same major, minor, and patch versions, their priority level (MUST) be determined by comparing dot-separated identifiers from left to right until a difference is found:
   - Numeric identifiers are compared numerically.
   - When identifiers contain letters or hyphens, they are compared lexically in ASCII sort order.
   - Numeric identifiers have lower priority than non-numeric identifiers.
   - If the initial identifiers are identical, the pre-release version with more fields has higher priority. For example: 1.0.0-alpha < 1.0.0-alpha.1 < 1.0.0-alpha.beta < 1.0.0-beta < 1.0.0-beta.2 < 1.0.0-beta.11 < 1.0.0-rc.1 < 1.0.0.

Search Mode	Description
Yes	Exact string comparison supports input of multiple keywords in an OR relationship, meaning the condition is satisfied when the field's value equals any one of the input keywords.
No	Exact string comparison supports input of multiple keywords in an AND relationship, meaning the condition is satisfied when the field's value does not equal any of the input keywords.
Greater than/Greater than or equal to	Version comparison only allows one keyword as input, indicating that the condition is satisfied when the field's value is greater than / greater than or equal to the version represented by the input keyword.
Less than/Less	Version comparison only allows one keyword as input, indicating that the condition is satisfied when the field's value is less than / less than or equal to the version

Than or Equal to	represented by the input keyword.
interval	Version comparison requires input of two keywords, for example, if the input keywords are A and B, the condition is satisfied when A <= x <= B.
Empty	When the field is an empty string / or when the corresponding field is not reported, the condition is satisfied. The platform does not distinguish between no value and an empty string.
Not empty	The condition is satisfied when the field is a non-empty string, that is, when the string length is greater than 0.
Match	Fuzzy string matching supports input of multiple keywords in an OR relationship, meaning the condition is satisfied when the field's value contains any one of the input keywords.
Not Matched	Fuzzy string matching supports input of multiple keywords in an AND relationship, meaning the condition is satisfied when the field's value does not contain any of the input keywords.
Match at the beginning	Fuzzy string matching supports input of multiple keywords in an OR relationship, meaning the condition is satisfied when the field's value starts with any of the input keywords.
Does not start with	Fuzzy string matching supports input of multiple keywords in an AND relationship, meaning the condition is satisfied when the field's value does not start with any of the input keywords.
Ends With Match	Fuzzy string matching supports input of multiple keywords in an OR relationship, meaning the condition is satisfied when the field's value ends with any of the input keywords.
End mismatch	Fuzzy string matching supports input of multiple keywords in an AND relationship, meaning the condition is satisfied when the field's value does not end with any of the input keywords.
Regular Expression	Fuzzy string matching supports input of multiple keywords in an OR relationship, meaning the condition is satisfied when the field's value matches any of the input keywords.

The following demonstrates version type searches through several examples of **App version**.

Search Purpose	operator	Input keyword	Description

Search for instances with versions greater than '4.5.2'.	Greater than	'4.5.2'	Instances with version '4.5.2.1' satisfy the condition, but those with version '4.5.2.0' do not.
Search for instances of all subversions for versions '4.5.2' and '4.5.3'.	Starts with	'4.5.2.', '4.5.3.'	If the trailing '.' is omitted, versions such as '4.5.20.xx' and similar will also be included.
Search for instances of all subversions with hotpatch version 499.	Ends with	'(hotpatch:499)'	The App version consists of a major version number and a hotpatch version number, represented in the standard format: '4.5.3.1(hotpatch:499)', where '4.5.3.1' is the major version number.

Value Type

Filter items of numerical type include fields such as **Usage Duration**, **Available memory**, and **Lag time**. Supported search modes are **Equal to**, **Not equal to**, **Greater than**, **Greater than or equal to**, **Less than**, **Less Than or Equal to**, and **interval**.
Filter items of numerical type store values that are either integers or floating-point numbers.

Search Mode	Meaning
Equal to	It supports entering multiple keywords, which are in an OR relationship, meaning the condition is met when the field value equals any one of the input keywords.
Not equal to	It supports entering multiple keywords, which are in an AND relationship, meaning the condition is met when the field value does not equal any one of the input keywords.
Greater than	Only a single keyword input is supported, indicating that the condition is met when the field value is greater than the input keyword.
Greater than or equal to	Only a single keyword input is supported, indicating that the condition is met when the field value is greater than or equal to the input keyword.
Less than	Only a single keyword input is supported, indicating that the condition is met when the field value is less than the input keyword.

Less Than or Equal to	Only a single keyword input is supported, indicating that the condition is met when the field value is less than or equal to the input keyword.
interval	It supports entering two keywords, such as left and right, indicating that the condition is met when the field value equals either of them or falls between them.

The following demonstrates numeric search through several examples of **Usage Duration**.

Search Purpose	operator	Input keyword	Description
Search for cases where the **Usage Duration** is greater than 30 seconds	Greater than	30	The unit of usage duration is seconds, so entering 30 indicates 30 seconds.
Search for cases where the **Usage Duration** is between 30 minutes and 1 hour	Interval	1800, 3600	The unit of usage duration is seconds, where 30 minutes equals 1800 seconds, and 1 hour equals 3600 seconds.

The following demonstrates numeric search using the custom dimension field n1. Assume that the business uses n1 to represent the tier classification of device models, divided into five tiers ranging from 1 to 5 (lowest to highest).

Search Purpose	operator	Input keyword	Description
Search for cases reported by models of tier 1 and tier 2	Yes	1,2	Indicates searching for cases where n1 = 1 or n1 = 2
Search for cases reported by models of tiers other than 1 and 3	No	1, 3	Indicates searching for cases where n1 != 1 and n1 != 3
Search for cases reported by models of tier 3 and above	Greater than or equal to	3	Indicates searching for cases where n1 >= 3
Search for cases reported by models of tier 2 to tier 4	Interval	2, 4	Indicates searching for cases where 2 <= n1 <= 4

String Type

Supports **Yes**, **No**, **Match**, **Not Matched**, **Match simultaneously**, **Empty**, **Not Empty**, **Match at the beginning**, **Beginning mismatch**, **Ends With Match**, **End mismatch**, **Regular expression match**, **Regular expression mismatch** search modes.
String-type filter options include **error stack**, **faulty thread**, **faulty process**, **Message ID**, **Device ID**, **User ID**, **Scene**, **Model**, **Vendor**, etc.

Search Mode	Meaning
Yes	Exact string comparison supports input of multiple keywords in an OR relationship, meaning the condition is satisfied when the field's value equals any one of the input keywords.
No	Exact string comparison supports input of multiple keywords in an AND relationship, meaning the condition is satisfied when the field's value does not equal any of the input keywords.
Empty	When the field is an empty string / or when the corresponding field is not reported, the condition is satisfied. The platform does not distinguish between no value and an empty string.
Not Empty	The condition is satisfied when the field is a non-empty string, that is, when the string length is greater than 0.
Match	Fuzzy string matching supports input of multiple keywords in an OR relationship, meaning the condition is satisfied when the field's value contains any one of the input keywords.
Match simultaneously	Fuzzy string matching supports input of multiple keywords in an AND relationship, meaning the condition is satisfied when the field's value contains all of the input keywords.
Not Matched	Fuzzy string matching supports input of multiple keywords in an AND relationship, meaning the condition is satisfied when the field's value does not contain any of the input keywords.
Match at the beginning	Fuzzy string matching supports input of multiple keywords in an OR relationship, meaning the condition is satisfied when the field's value starts with any of the input keywords.
Beginning	Fuzzy string matching supports input of multiple keywords in an AND relationship, meaning the condition is satisfied when the field's value does not start with any of the input keywords.

mismatch	
Ends With Match	Fuzzy string matching supports input of multiple keywords in an OR relationship, meaning the condition is satisfied when the field's value ends with any of the input keywords.
End mismatch	Fuzzy string matching supports input of multiple keywords in an AND relationship, meaning the condition is satisfied when the field's value does not end with any of the input keywords.
Regular expression match	Fuzzy string matching supports input of multiple keywords in an OR relationship, meaning the condition is satisfied when the field's value matches any of the input keywords.
Regular expression mismatch	Fuzzy string matching supports input of multiple keywords in an AND relationship, meaning the condition is satisfied when the field's value does not match any of the input keywords.

The following demonstrates string search through several samples of **stack trace**.

Search Purpose	operator	Input keyword	Description
Search for stack trace instances that contain the package name 'com.tencent.rumpro'.	Match	'com.tencent.rumpro'	Stacks typically contain multiple function frames, where containment matching is required rather than prefix matching.
Search for stack trace instances that contain both the 'com.tencent.rumpro' and 'com.tencent.rmonitor' package names.	Simultaneous match	'com.tencent.rumpro', 'com.tencent.rmonitor'	Simultaneous matching indicates that the stack trace must contain all specified keywords.

The following uses several examples of **error process** to demonstrate string search. Assume a business contains multiple processes, as detailed below:

Business processes	Process Type
com.tencent.demo	Main process
com.tencent.demo:video_1	Video processes
com.tencent.demo:video_2	Video processes
com.tencent.demo:game_1	Game processes
com.tencent.demo:game_2	Game processes
com.tencent.demo:xg_vip_service	Service processes
com.tencent.demo:pm_service	Service processes
com.tencent.demo:decode_service	Service processes
com.tencent.demo:encode_service	Service processes

Search Purpose	operator	Input keyword	Description
Search for the instances that only appear in the main process.	Yes	com.tencent.demo	Adopting exact match of field values
Search for the instances that only appear in the video process.	Starts with	com.tencent.demo:video_	Find the video process by prefix matching.
Search for the instances that appear in the video process or game process.	Starts with	com.tencent.demo:video , com.tencent.demo:game	By entering multiple keywords to achieve the desired effect.
Search for the instances that do not appear in the service processes.	Does not end with	_service	Search for instances not occurring in service processes through suffix mismatch.

String array type

The string array type indicates that the value of a field is a string array, including fields such as **custom dimension's a1–a10**, **experiment ID**, **Tag for individual case**, **business drill-down**.
The string array type supports search patterns such as **Include**, **Exclude**, **contains both**, **Match**, **Match simultaneously**, **Empty**, **Not Empty**, **Match at the beginning**, **Ends With Match**, and others. For the

comparison of string arrays, most patterns ultimately compare each string in the string array.

Search Mode	Meaning
Include	Exact string comparison supports inputting multiple keywords, which are in an OR relationship, meaning the condition is met when any of these keywords is present in the field value.
Exclude	Exact string comparison supports inputting multiple keywords in an AND relationship, meaning the condition is met only when none of the keywords are present in the field value.
contains both	Exact string comparison supports inputting multiple keywords in an AND relationship, meaning the condition is met only when all keywords are present in the field value.
Match	Fuzzy string comparison supports inputting multiple keywords in an OR relationship, meaning the condition is met when any string in the field value contains any of the input keywords.
Match simultaneously	Fuzzy string comparison supports inputting multiple keywords in an AND relationship, meaning the condition is met when any string in the field value contains all input keywords.
Empty	The condition is met when the field value is an empty array.
Not Empty	The condition is met when the field value is not an empty array.
Match at the beginning	Fuzzy string comparison supports inputting multiple keywords in an OR relationship, meaning the condition is satisfied when any string in the field value starts with any of the input keywords.
EndsWith Match	Fuzzy string comparison supports inputting multiple keywords in an OR relationship, meaning the condition is satisfied when any string in the field value ends with any of the input keywords.

For example, a user uses the a1 field to report the names of recently visited pages. The login page contains two Models: 'LoginModeOnePage' and 'LoginModeTwoPage'. Below are some examples of the a1 field values reported.

Reported Cases	a1 field value

1	['LoginModeOnePage', 'MessageListPage', 'FriendDetailPage']
2	['LoginModeTwoPage', 'MessageListPage', 'FriendListPage']
3	['LoginModeTwoPage', 'MessageListPage', 'FriendListPage', 'FriendDetailPage']
4	['MessageListPage', 'FriendListPage', 'FriendDetailPage']
5	['FriendListPage', 'MessageListPage', 'FriendDetailPage']

Below are several examples of **custom dimension a1** to demonstrate string array type search.

Search Purpose	operator	Input keyword	Description
Search for cases that include the login page, such as cases 1, 2, and 3 mentioned above.	Include	'LoginModeOnePage', 'LoginModeTwoPage'	The login page contains two Models: 'LoginModeOnePage' and 'LoginModeTwoPage'.
Search for cases that include the login page, such as cases 1, 2, and 3 mentioned above.	Starts with	'Login'	Using 'Login' to match the two Models of the login page.
Search for cases that include both the login page and the friend detail page, such as cases 1 and 3 mentioned above.	Simultaneous match	'Login','FriendDetailPage'	Here, simultaneous inclusion cannot be used because 'LoginModeOnePage' and 'LoginModeTwoPage' do not appear at the same time.
Search for cases that have not performed login operations, such as cases 4 and 5 mentioned above.	Not included	'LoginModeOnePage', 'LoginModeTwoPage'	Not included is the negation of included.

Enumerable Type

Enumerated types are multi-select dropdown boxes that include "All" and all available options for the field. Filter options for enumerated types include fields such as **build type**, **Foreground or background status**, and **Device Type**.

Enumerated types may store boolean values, strings, or integers. Regardless of the type, enumerated types are always those with enumerable values, and the number of distinct values is typically not excessive—generally not exceeding 10 items.

Enumerated types default to "All selected," meaning no filtering by any specific type. Users can select the types to match via the dropdown, which typically supports multiple selection.

Filter Field	Enum value
build type	All, Developer Edition, Canary Edition, Release Edition
Foreground or background status	All, Foreground, Background
Device Type	Real Device + Simulator, Real Device, Simulator

The following demonstrates the search for enumerated types through several examples of **Edition Type**.

Search Purpose	Selected items	Description
Search all reported cases in the Developer Edition.	Developer Edition	This indicates that only cases in the Developer Edition will be searched.
Search for reported cases in the Canary Edition and Release Edition.	Canary Edition, Release Edition	This indicates that the condition is met as long as it is one of the Canary Edition or Release Edition types.
Search for all reported cases in all editions.	No limit	When the "No limit" option is selected, no other options can be chosen simultaneously.

Special Type

Special type filter options support two search modes: **OR, AND**, including fields such as **linkage marking**, **exception site**, **attachment**.

Search Mode	Meaning
OR	It supports selecting multiple options, which are in an OR relationship.
AND	It supports selecting multiple options, which are in an AND relationship.

The following demonstrates special type searches through several samples of **linkage marks** in the crash issue list.

Search Purpose	opera tor	Input keyword	Description

Search for cases where lag or Java memory leaks occurred before a crash.	OR	Lag, Java memory leak	The condition is met as long as the case has experienced lag or Java memory leak.
Search for cases where both lag and Java memory leaks occurred before a crash.	AND	Lag, Java memory leak	Only cases where both lag and Java memory leaks have occurred meet the condition.

Custom Data

Last updated：2026-05-25 18:53:29

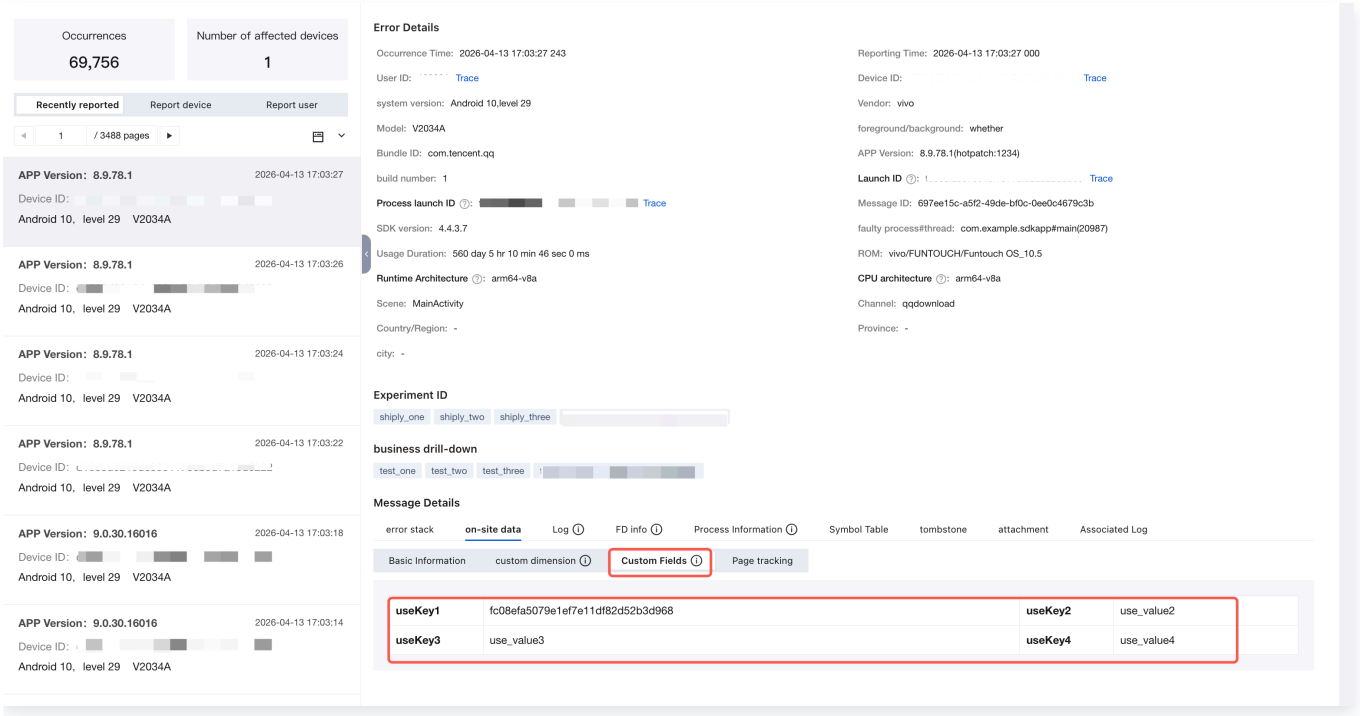
The platform supports application-level settings for user-defined data. When reporting metrics or issues, obtain the custom data configured at the application level to facilitate custom analysis. Currently, two types of custom data are supported: custom fields and custom dimensions.

Custom fields

Custom fields are custom data with freely definable keys. Keys and values are both of the String type, and the specific values are freely defined by the application. Currently, only issues support configuring custom fields, while performance metrics are not currently supported.

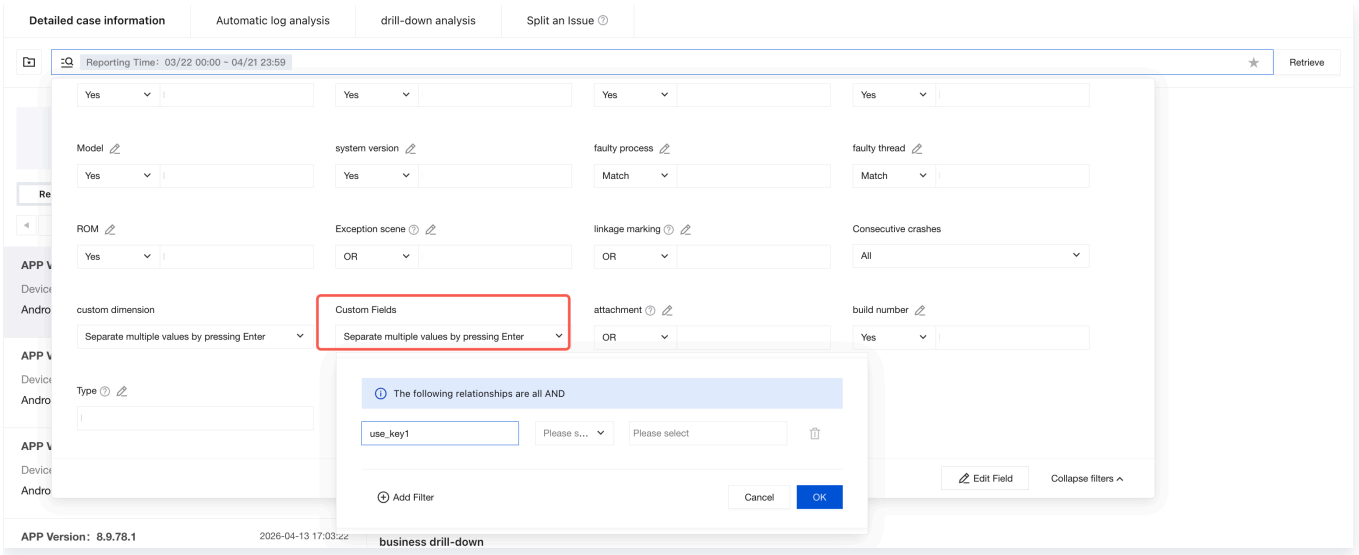
View custom fields

Users can view the custom fields of the current case in **Issue Details** > **Message Details** > **on-site data**.



Search Custom Fields

Users can also search for cases containing specified custom fields in the search criteria.



custom dimension

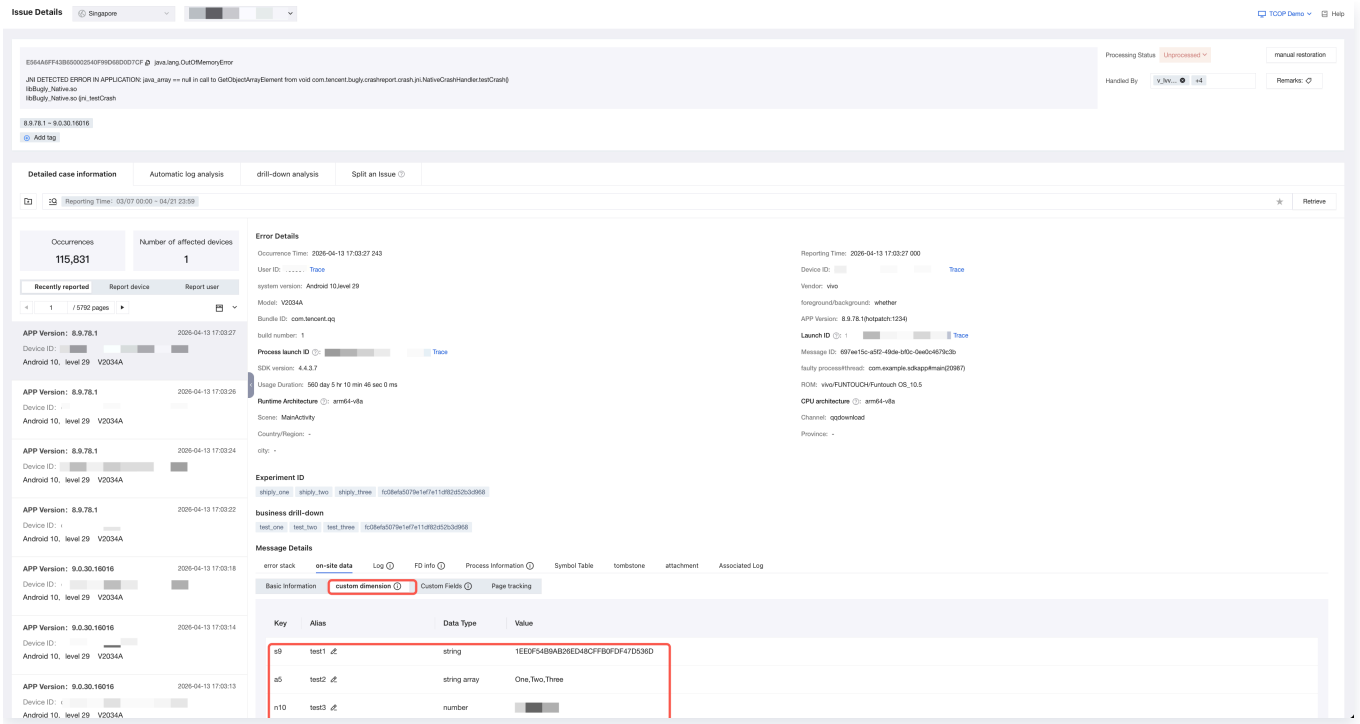
Custom dimensions are user-defined data with constrained keys. Keys are defined by the SDK, and applications can set values with different meanings based on actual needs. Values currently support three types: Number, String, and String Array, with each type supporting ten keys.

- Values of the Number type correspond to dimension keys n1 – n10.
- Values of the String type correspond to dimension keys s1 – s10.
- Values of the String Array type correspond to dimension keys a1 – a10.

> **The purpose of introducing custom dimensions:**
> RUM Pro introduced custom dimensions after balancing search efficiency, analytical capabilities, storage costs, and computational costs. While the usage remains similar with a Key/Value configuration approach, custom dimensions employ SDK-predefined keys. Upon receiving data, backend processing parses and stores this information in a highly optimized format. Crucially, this storage optimization enables the platform to deliver robust custom analytical capabilities in future releases.

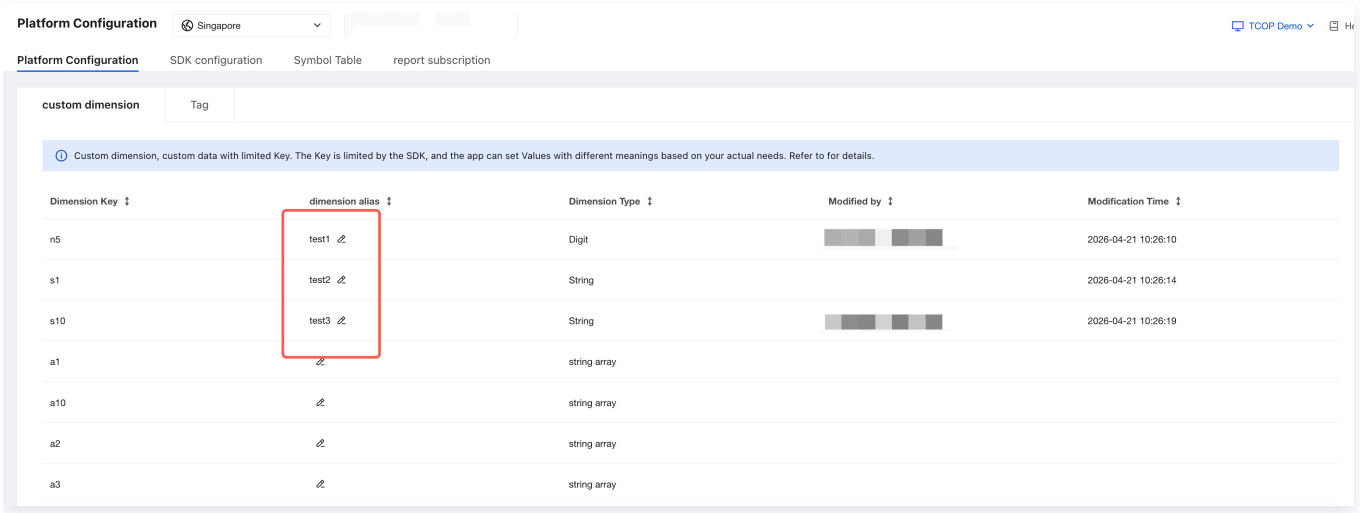
View custom dimension

Users can view the custom dimensions of the current case in **Issue Details** > **Message Details** > **on-site data**.

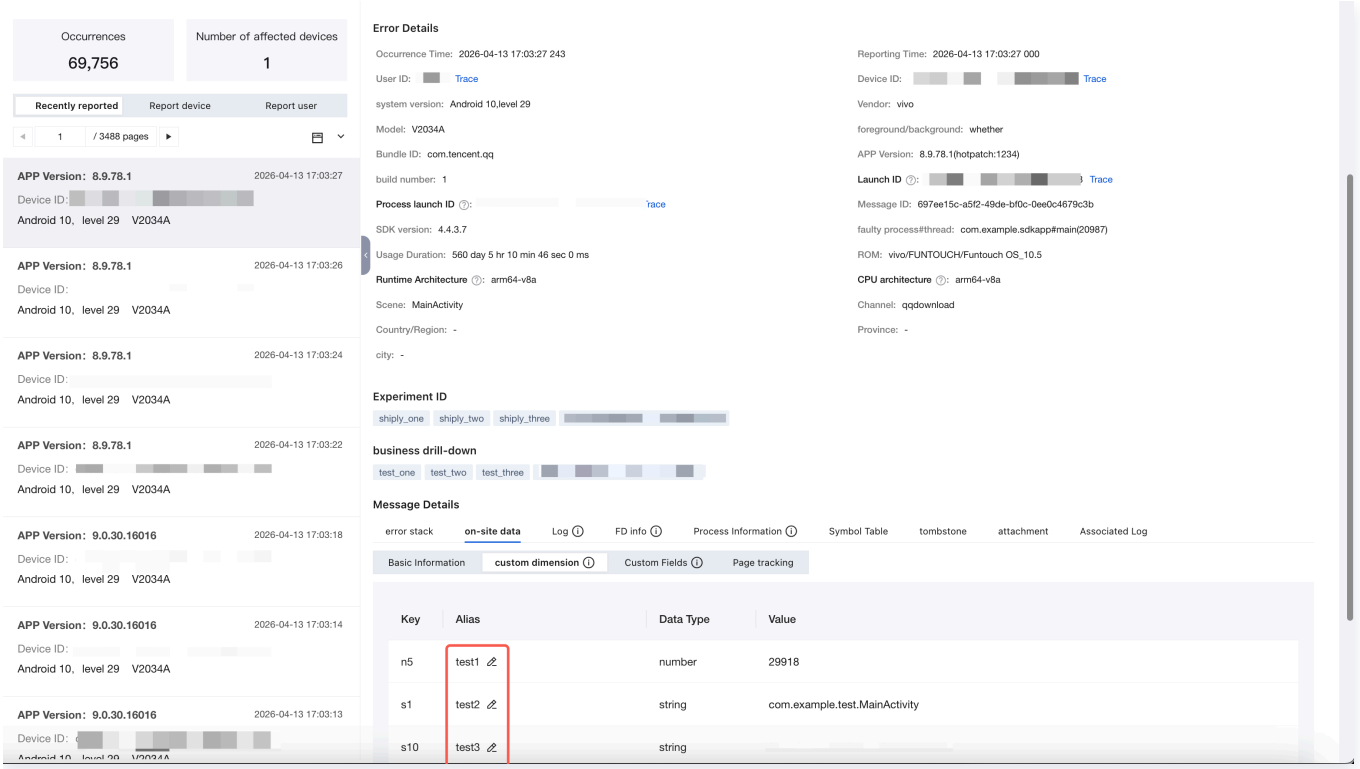


Set dimension alias

Users can set dimension aliases for these custom dimensions in the Setting > Platform Configuration > custom dimension page, assigning an appropriate alias to interpret the meaning of each custom dimension.



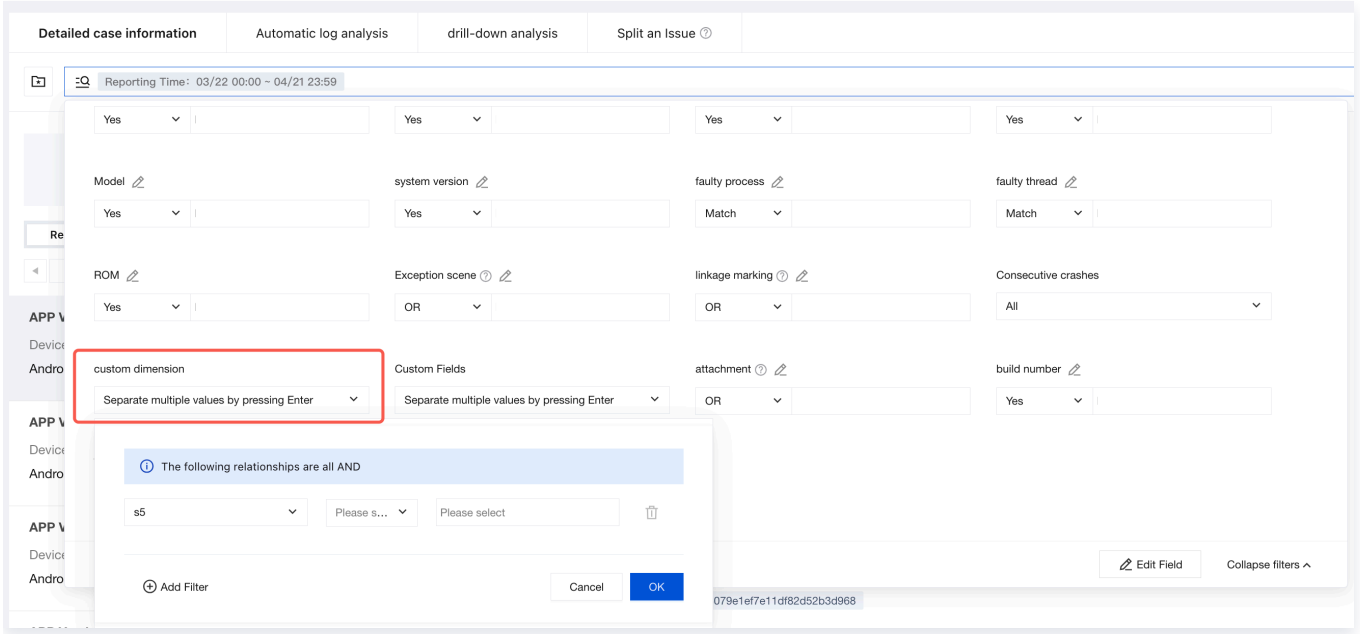
After updating dimension aliases, refresh the page in **Issue Details** > **Message Details** > **on-site data** to view the dimension aliases set for the corresponding custom dimensions.



Search by dimension alias

After setting dimension aliases, users can directly search for custom dimensions via these aliases.

> **Note:**
> The example shows that custom dimensions of the Number type offer enhanced search capabilities, currently supporting exact Match, equal to, greater than, greater than or equal to, less than, less than or equal to, and range-based searches.



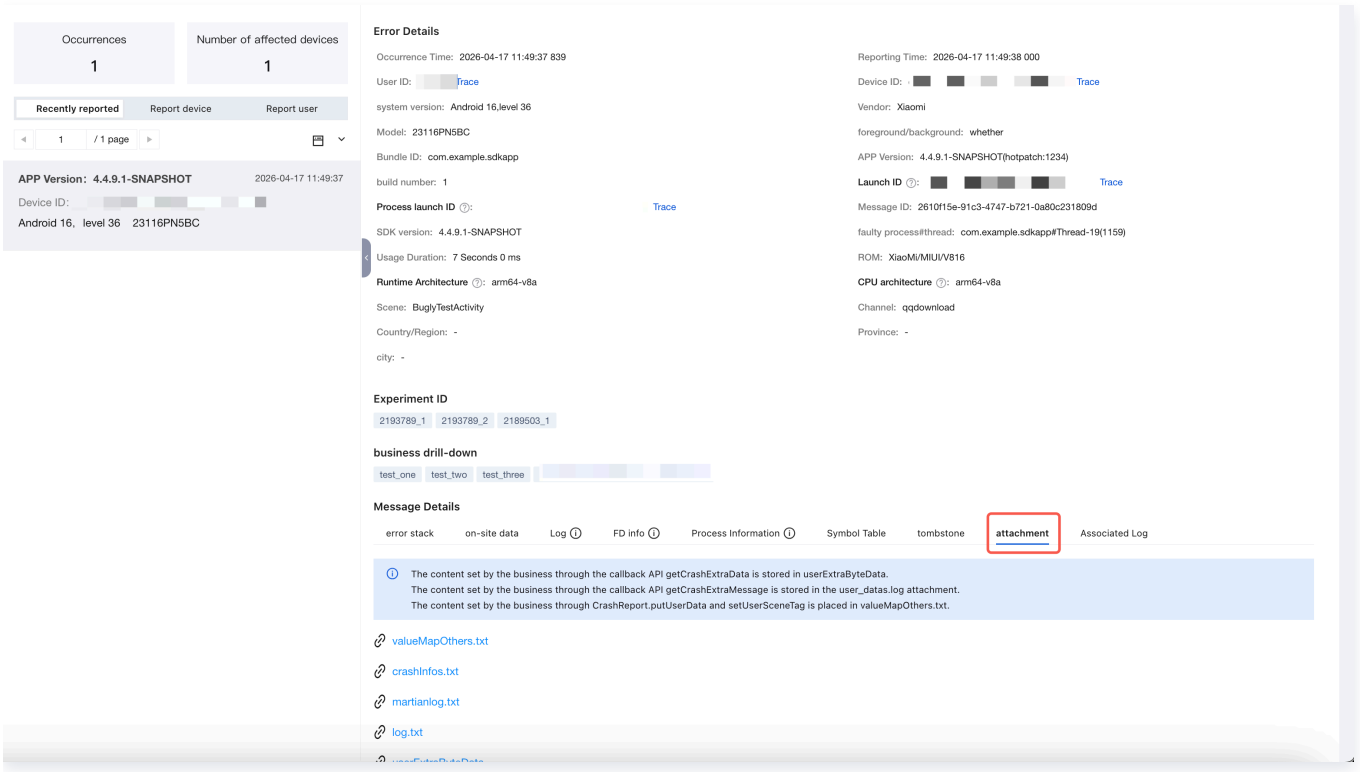
Attachment

Last updated：2026-05-25 18:53:29

The attachments provide on-site abnormal data in file format, containing two types of data: platform-default collected data and business-custom collected data.

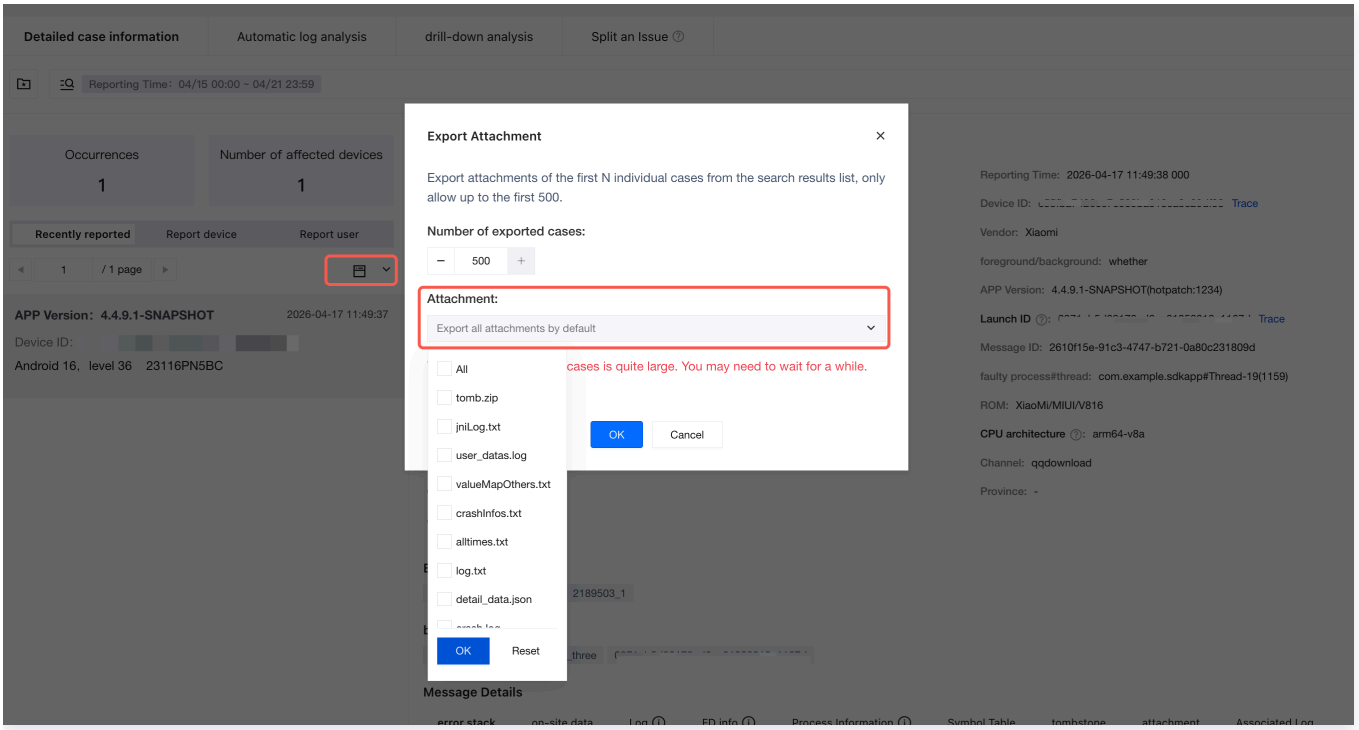
Feature Introduction

In the case details on the platform, the **attachment** Tab displays all attachments for the current exception case. Users can click to view attachment contents online. Users can also click **Compress and download** to download all attachments for the current case.



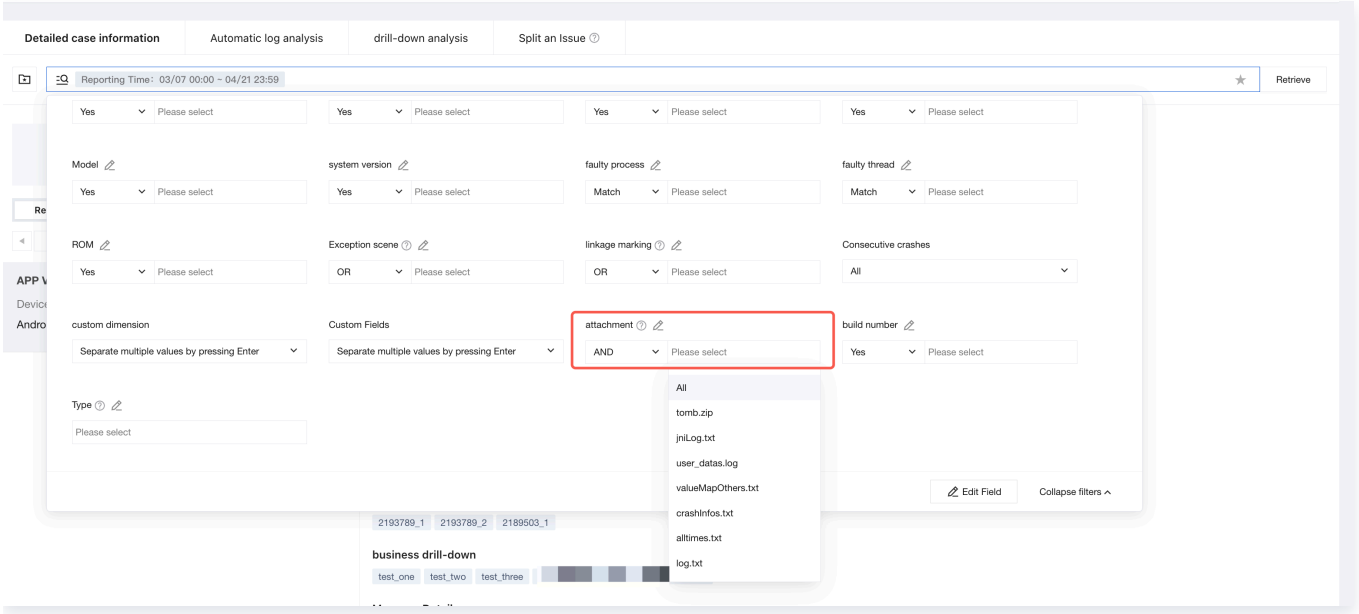
Export in batches

Supports the feature of batch exporting case attachments. Click the download icon on the left > **Export Attachment**, select the attachments to be downloaded, then click **OK**, as shown in the figure below:



Search attachments

Users can use the **attachment** filter option to search for cases containing specified attachments. When multiple attachments are selected, it represents an OR relationship, filtering out exception cases that contain any of the following attachments.



Upload Custom Attachments

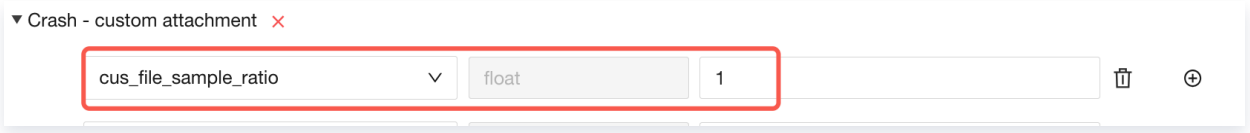
Android

Businesses can use the custom attachment upload API exposed by the SDK to supplement additional on-site business data in the form of attachments.

- Custom attachments are supported starting from Android SDK version 4.3.0.3.
- Using this API requires initializing both the SDK performance and SDK quality modules. For details, see the SDK Access Guide.
- The size limit for custom files after compression is 10 MB. If the size exceeds this limit, the file will be deleted directly and will not be uploaded.
- This API can be called multiple times, and subsequent calls will override the values set previously.

```
CrashReport.setAdditionalAttachmentPaths(String[] files);
```

- Custom files are uploaded only during the next startup.
- The SDK disables this feature by default. To enable it, add the following configuration on the Portal side.



- After the next startup, you can check the "sub_type: custom_log" keyword logs to confirm whether custom files have been uploaded.

```
12-07 20:42:26.673 15758 15862 I RMonitor_report_File: url:
https://rmonitor.qq.com/v1/2ff6123857/custom/upload-file?
sign=22efd184a95c12d7faaecb8e8edc3266&timestamp=1670416946671&nonc
e=ee48217de804345e1634d0adf58e8825, sub_type: custom_log
```

iOS

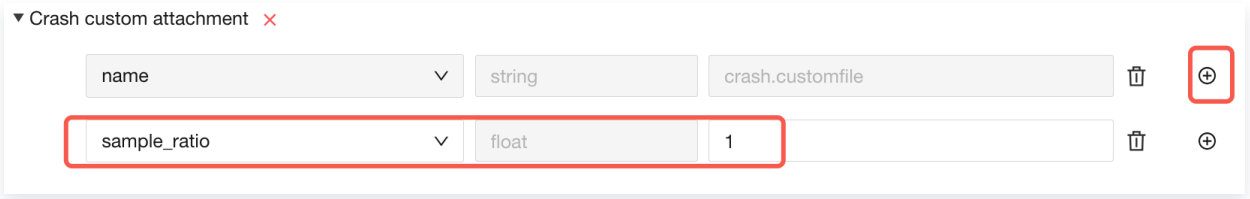
- Support for custom attachments is available starting from iOS SDK version 2.7.51.
- Refer to the following code to set the path for custom attachments. After an exception occurs, the attachments will be packaged and uploaded from the corresponding directory.

```
/**
 * Set the collection of determined paths for custom files.
 * The compressed file must not exceed 10 MB and will be uploaded
on the next startup.
```

```
 */
+ (void)setAdditionalAttachmentPaths:(NSArray<NSString
*>*)pathArray;

Usage example:
[RumPro setAdditionalAttachmentPaths:[NSArray
arrayWithObject:filePath]];
```

- Custom files are uploaded only during the next startup.
- The SDK disables this feature by default. To enable it, add the following configuration on the Portal side.
- **Calling this API when a crash occurs will not take effect**. You need to set the file paths in advance.
- If you do not see the custom file option in the current configuration, click the plus icon on the far right of the last configuration item to add a custom file configuration item.



Android

The following content mainly introduces the information recorded for each file in the Attachments Tab of Android reported case details and the meaning of each piece of information.

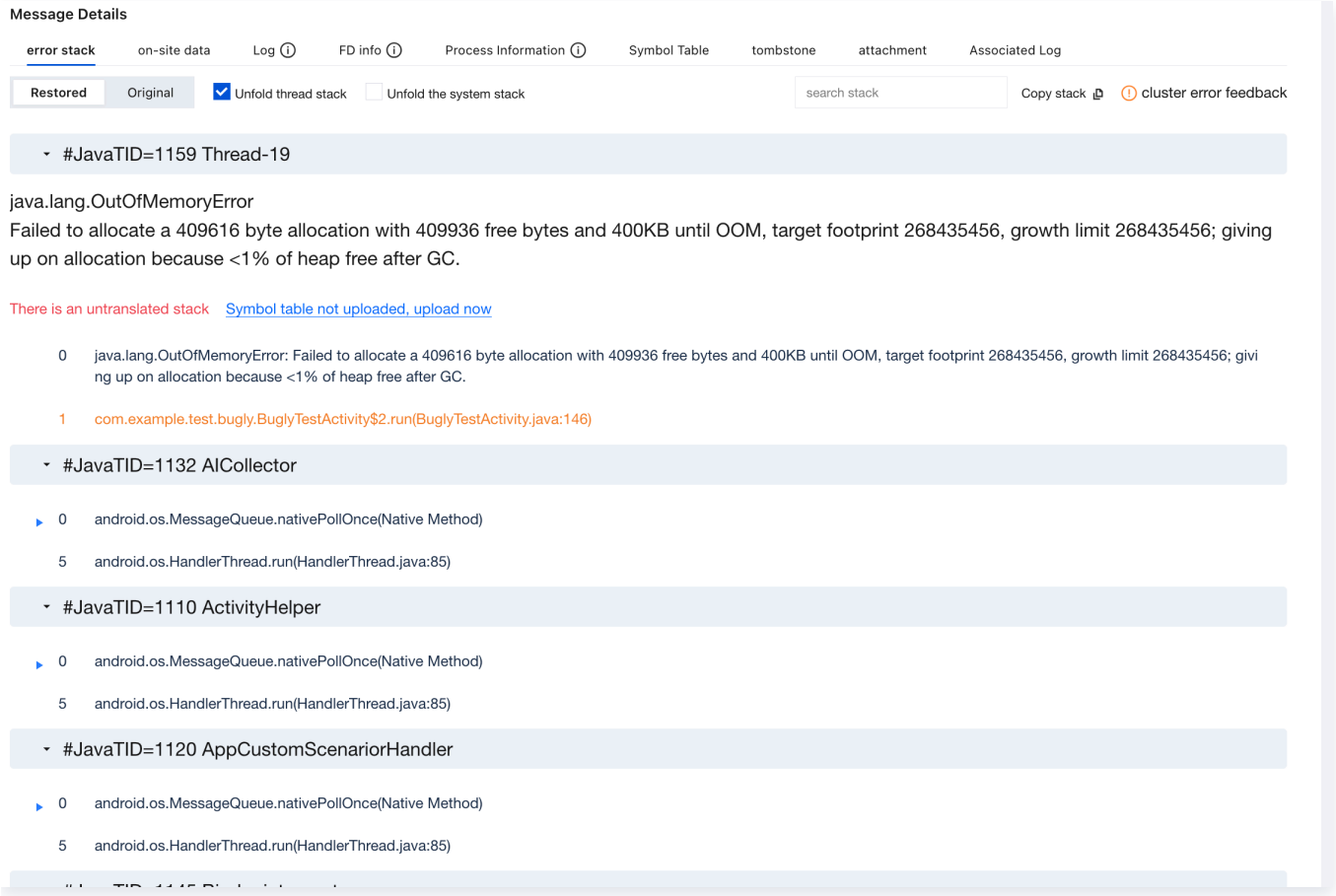
Attachments Overview

Classification/Tab	Source
error stack	Exported in the attachment "crash.log" from the crash details API.
tombstone	From attachment tomb.zip
Crash Context / Basic Information	Exported to the attachment "detail_data.JSON" from the crash details API.
Crash Context / Custom Fields	From attachment "valueMapOthers.txt"
Crash Context / Page Tracking	From attachment martianlog.txt

Logs	From attachment log.txt
FD Information	From attachment crashInfos.txt
Process Information	From attachment state_file.zip
Information supplemented via the callback API getCrashExtraData	From attachment extraMessage.txt
Information supplemented via the callback API getCrashExtraMessage	Exported in the attachment user_datas.log
The content configured via CrashReport.putUserData and setUserSceneTag	Exported in the attachment valueMapOthers.txt
ANR_INFO	From attachment anrMessage.txt
ANR_Trace	From attachment trace.zip

crash.log

Stores error stack information, including the stack of the current exception thread and the stacks of other threads, consistent with the content displayed in the Error Stack Tab.



Why there are two different stacks for the crashed thread with different process IDs preceding them?

1. For Native exceptions, both the Native stack and Java stack are captured during exception handling at the Native layer, obtaining the process ID at the Native level.
2. The stacks of other threads are treated as additional information, processed at the Java layer, where the JVM stacks are retrieved, and the process ID corresponds to the one described within the JVM.
3. The current stacks of other threads include all JVM thread stacks of the application at the time of the exception, which may also include the currently crashed thread, but do not include threads created at the Native layer.

tomb.zip

Corresponding to the system's "tombstone" file, generated when an application or system process crashes. This file contains critical information at the time of the crash, such as stack traces, memory maps, and register states. The file is generated following the format modeled after the system's "tombstone" file. tombstone contains the following information sections:

- **Exception Summary**

```
*** *** *** *** *** *** *** *** *** *** *** *** *** *** *** ***
Crash type: 'Native'
Start time: '2023-12-22T15:55:39.381+0800'
Crash time: '2023-12-22T15:58:30.119+0800'
App version: '4.4.0-beta.3'RumPro
Rooted: 'No'
API level: '30'
Build fingerprint:
'vivo/PD1824/PD1824:11/RP1A.200720.012/compiler03061748:user/release-
keys'
ABI: 'arm64'
pid: 26730, tid: 26730, name: .example.sdkapp  >>> com.example.sdkapp
<<<
signal 6 (SIGABRT), code -1 (SI_QUEUE), fault addr --------
Abort message: 'JNI DETECTED ERROR IN AppLICATION: jarray was NULL
in call to GetObjectArrayElement     from void
com.tencent.RumPro.crashreport.crash.jni.NativeCrashHandler.testCrash(
)'
```

- **Register Information**

```
    x0  0000000000000000  x1  000000000000686a  x2  0000000000000006
x3  0000007fdab30cc0
    x4  00000077f497b000  x5  00000077f497b000  x6  00000077f497b000
x7  000000000042906e
    x8  00000000000000f0  x9  b8ae21afa5964f35  x10 0000000000000000
x11 ffffffc0fffffbdf
    x12 0000000000000001  x13 00000000658541a6  x14 0005e9acbf980fa7
x15 000077c18d549120
    x16 00000077f138b948  x17 00000077f1368630  x18 00000077f3d3a000
x19 000000000000686a
    x20 000000000000686a  x21 00000000ffffffff  x22 000000000000000b
x23 000000000000000b
    x24 000000776caa41c5  x25 0000000000000001  x26 000000776cabb177
x27 000000776d072000
    x28 b4000077f3452340  x29 0000007fdab30d40
    sp  0000007fdab30ca0  lr  00000077f1317d24  pc  00000077f1317d50
```

- Exception Stack

```
backtrace:
    #00 pc 000000000008cd50
/apex/com.android.runtime/lib64/bionic/libc.SO (abort+164)
    #01 pc 00000000005320a0  /apex/com.android.art/lib64/libart.SO
(_ZN3art7Runtime5AbortEPKc+2340)
    #02 pc 000000000001394c  /system/lib64/libbase.SO
(_ZZN7android4base10SetAborterEONSt3__18functionIFvPKcEEEEN3$_38__invo
keES4_+76)
    #03 pc 00000000000130cc  /system/lib64/libbase.SO
(_ZN7android4base10LogMessageD1Ev+312)
    #04 pc 0000000000369b00  /apex/com.android.art/lib64/libart.SO
(_ZN3art9JavaVMExt8JniAbortEPKcS2_+2596)
    #05 pc 0000000000369b78  /apex/com.android.art/lib64/libart.SO
(_ZN3art9JavaVMExt9JniAbortVEPKcS2_St9__va_list+108)
    #06 pc 000000000035b7c4  /apex/com.android.art/lib64/libart.SO
(_ZN3art12_GLOBAL__N_111ScopedCheck6AbortFEPKcz+144)
    ...
    #81 pc 0000000000088188
/apex/com.android.runtime/lib64/bionic/libc.SO (__libc_init+108)
```

- **Build ID**

  Build ID is a unique identifier generated during the compilation process, used to identify specific binary files. The primary purpose of Build ID is to help developers determine the exact version of the binary file being executed during debugging. In Linux systems, Build ID is typically generated by the compiler or linker during compilation, often as a hash value of source code, compilation options, and other information. If two ELF files share the same Build ID, they are considered identical versions of the library.

```
build id:
    /apex/com.android.runtime/lib64/bionic/libc.SO (BuildId:
4995223f2954ed2746d96c2f1c7939a1. FileSize: 1314184. LastModified:
2009-01-01T08:00:00.000+0800. MD5: 0eabd0f3735afdc6317062180a4369ec)
    ...
    /system/bin/app_process64 (BuildId:
c8e30518cdd6709068b3a08e6bfe4a76. FileSize: 33832. LastModified: 2009-
01-01T08:00:00.000+0800. MD5: 68ed3a8e223ff95392828c5b5dd451f3)
```

  On Mac, the Build ID of an SO file can be directly read using the readelf tool.

```
> readelf -n libNative.SO

Displaying notes found in: .note.gnu.build-id
  Owner                Data size    Description
  GNU                  0x00000014   NT_GNU_BUILD_ID (unique build ID
bitstring)
    Build ID: dae1f4c9f93715c9de15c86161a1af37b5cce24a
```

- **Stack Memory**

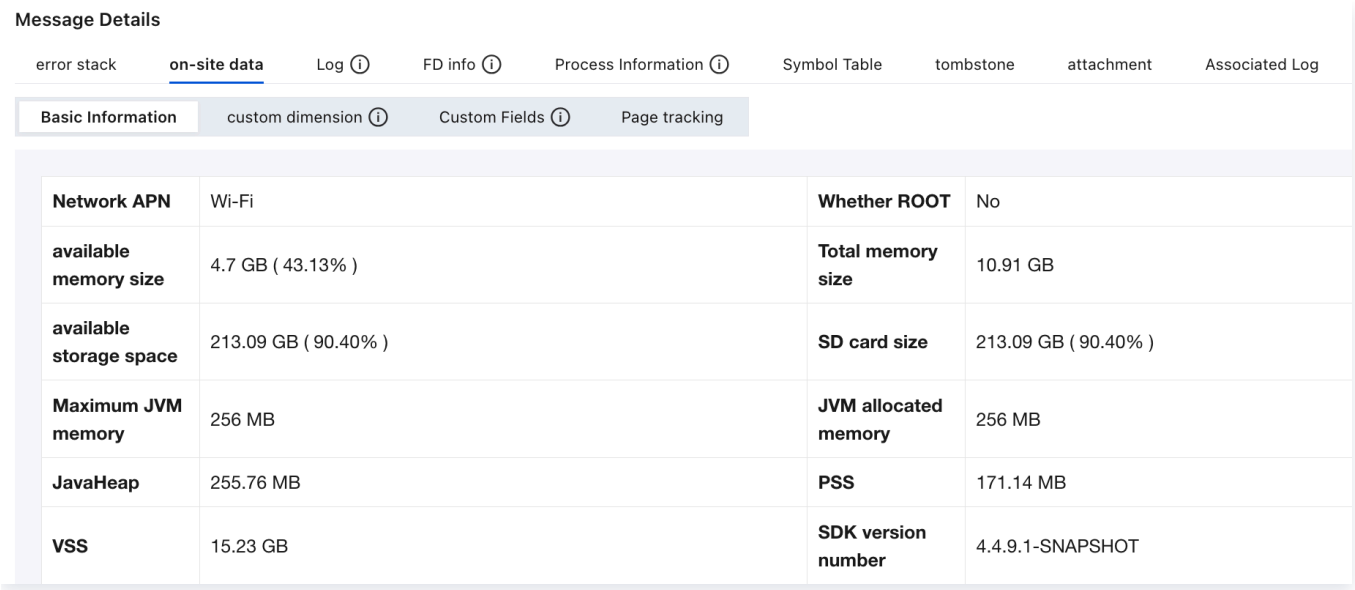
```
stack:
         0000007fdab30c20  00000077f3814000  [anon:stack_and_tls:main]
         0000007fdab30c28  0000000000000002
         0000007fdab30c30  0000007fdab30d58  [stack]
         0000007fdab30c38  0000000000000000
         0000007fdab30c40  000000776cac3c9e
/apex/com.android.art/lib64/libart.SO
         0000007fdab30c48  00000077f12d0c84
/apex/com.android.runtime/lib64/bionic/libc.SO (malloc+44)
         0000007fdab30c50  00000076f4d262c0  [anon:libc_malloc]
         0000007fdab30c58  b8ae21afa5964f35
```

```
          0000007fdab30c60  0000007fdab30d40  [stack]
          0000007fdab30c68  00000077f1317ce8
/apex/com.android.runtime/lib64/bionic/libc.SO (abort+60)
          0000007fdab30c70  000000000000000b
          0000007fdab30c78  000000776cac3c9e
/apex/com.android.art/lib64/libart.SO
          0000007fdab30c80  000000000000000b
          0000007fdab30c88  00000076fd2b9a00  [anon:libc_malloc]
          0000007fdab30c90  00000077f4a797f0  [anon:.bss]
          0000007fdab30c98  00000076f4c6e080  [anon:libc_malloc]
    #00   0000007fdab30ca0  0000000000000001
    ...
    ........  ........
    #81   0000007fdab35f80  0000007fdab35fd0  [stack]
          0000007fdab35f88  0000005e8de33048  /system/bin/app_process64
(main)
          0000007fdab35f90  0000000000000000
          0000007fdab35f98  0000000000000000
          0000007fdab35fa0  0000000000000000
          0000007fdab35fa8  0000000000000000
          0000007fdab35fb0  0000000000000000
          0000007fdab35fb8  0000005e8de37000  /system/bin/app_process64
          0000007fdab35fc0  0000005e8de37010  /system/bin/app_process64
          0000007fdab35fc8  0000005e8de37028  /system/bin/app_process64
          0000007fdab35fd0  0000000000000000
          0000007fdab35fd8  00000077f49caa68
/apex/com.android.runtime/bin/linker64 (__dl__start+8)
          0000007fdab35fe0  0000000000000006
          0000007fdab35fe8  0000007fdab36290  [stack]
          0000007fdab35ff0  0000007fdab362aa  [stack]
          0000007fdab35ff8  0000007fdab362b3  [stack]
```

- **Memory near general registers and memory information.**

detail_data.JSON

- Basic data and page tracking are on-site information automatically captured by the SDK during exceptions, with the basic data exported in the attachment "detail_data.JSON".
- The "App Channel" shown in the custom fields comes from the "appInfo" in "detail_data.JSON".



valueMapOthers.txt

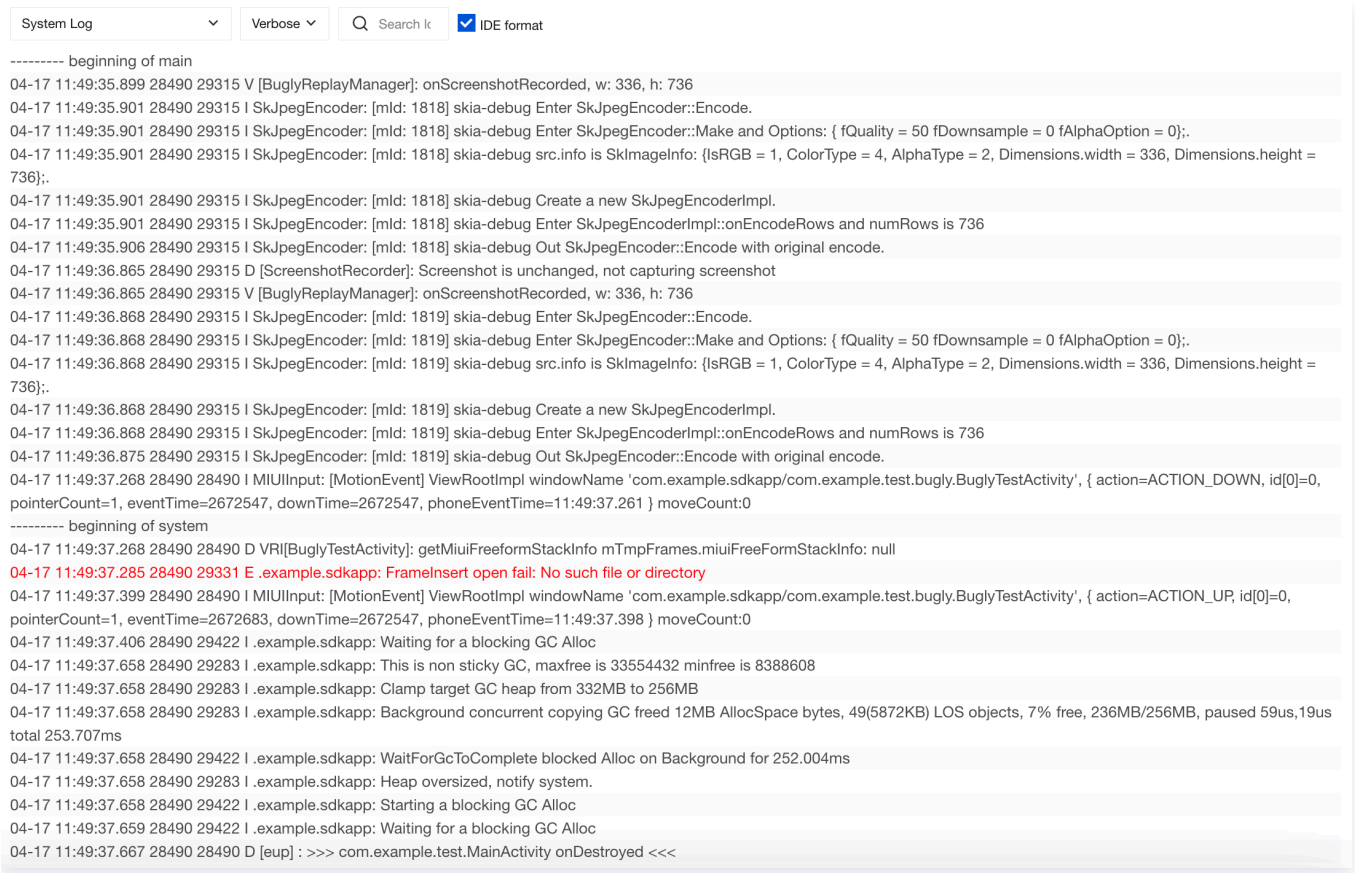
- Custom fields refer to the Key/Value pairs set by the business through the CrashReport.putUserData API, with the relevant content exported in the attachment "valueMapOthers.txt".
- The tags configured via the CrashReport.setUserSceneTag API are exported in the attachment "valueMapOthers.txt".
- The "App Channel" shown in the custom fields comes from the "appInfo" in "detail_data.JSON".
- The content displayed in the custom fields section of the on-site data is consistent with the attachment "valueMapOthers.txt".

martianlog.txt

- Page tracking data from the period before the exception occurred is automatically recorded by the SDK.
- This data is also displayed in **on-site data** > **Page tracking** sections.

log.txt

The system logs displayed in the Log Tab of the exception scene correspond to the attachment "log.txt".



crashInfos.txt

- The FD information displayed in the FD Tab of the exception scene comes from the attachment "crashInfos.txt".
- In addition, the attachment "crashInfos.txt" also contains System SO info, with information sourced from the "/proc/$pid/maps" file.

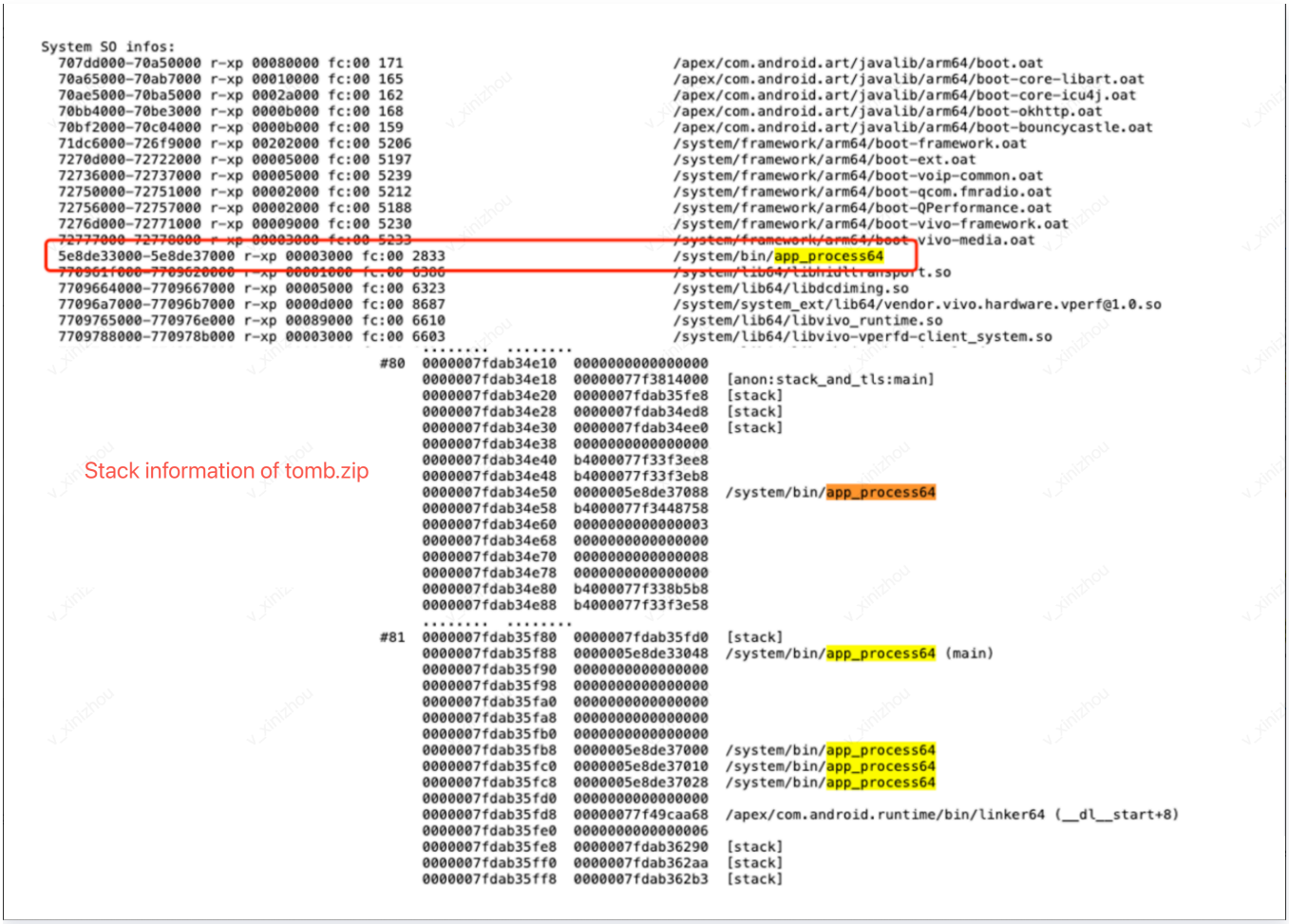
```
System SO infos:
  707dd000-70a50000 r-xp 00080000 fc:00 171
/apex/com.android.art/Javalib/arm64/boot.oat
  70a65000-70ab7000 r-xp 00010000 fc:00 165
/apex/com.android.art/Javalib/arm64/boot-core-libart.oat
  ...
```

Taking the first line as an example, this information describes the SO file's location in memory, access permissions, offset on disk, and other relevant details.

- "707dd000–70a50000": This is the address range of the file in memory. It means the file is mapped to memory starting at address 0x707dd000 and ending at address 0x70a50000.
- "r–xp": This indicates the access permissions of the file in memory. Here, "r" stands for readable, "x" for executable, and "p" for private.

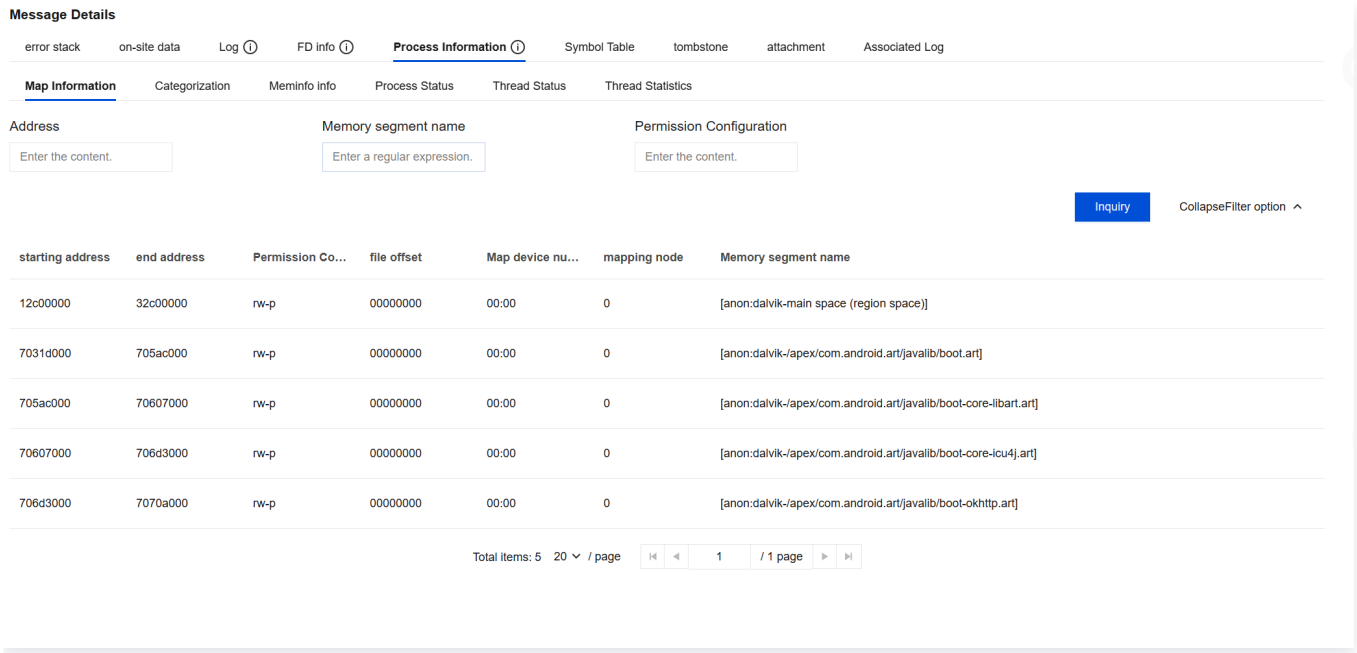
- "00080000": This is the file's offset on disk, indicating that the starting position of the file in the disk is offset by 0x00080000 from the beginning of the file.
- "fc:00": This represents the device number and inode number of the file, used to uniquely identify the file.
- "171": This is the file descriptor of the file, used to identify the file in the operating system.

The stack in tomb.zip also uses this information to identify the SO files corresponding to certain addresses.



state_file.zip

- The process information in the exception details comes from the attachment "state_file.zip".
- Maps information comes from the file "/proc/$pid/maps".
- Meminfo information indicates the memory usage of user processes, showing how the application's current memory is primarily allocated.
- Process Status and Thread Status indicate the state metrics of the currently running process and its individual threads, respectively.



user_datas.log

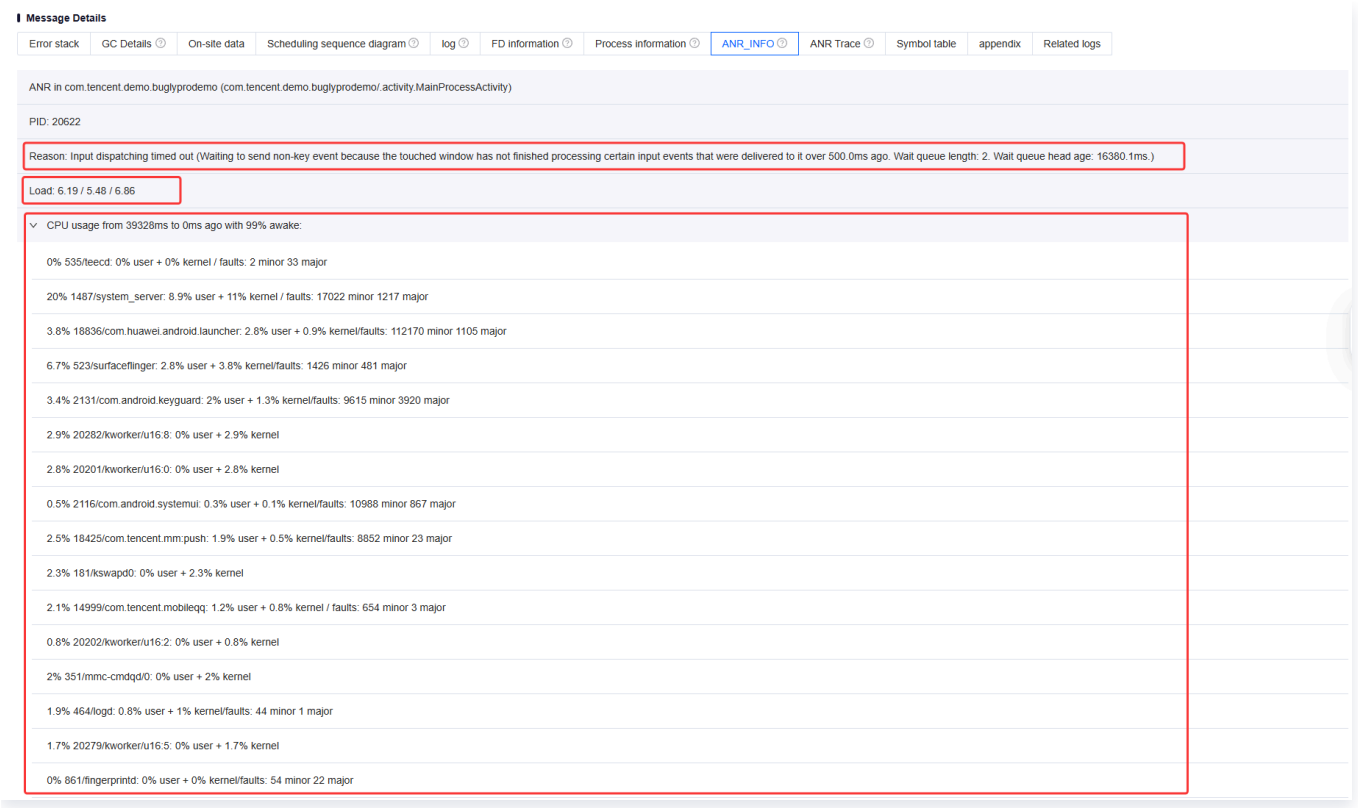
The information supplemented via the callback API getCrashExtraMessage is exported in the attachment "user_datas.log".

extraMessage.txt

The information supplemented via the callback API getCrashExtraData is exported in the attachment "extraMessage.txt".

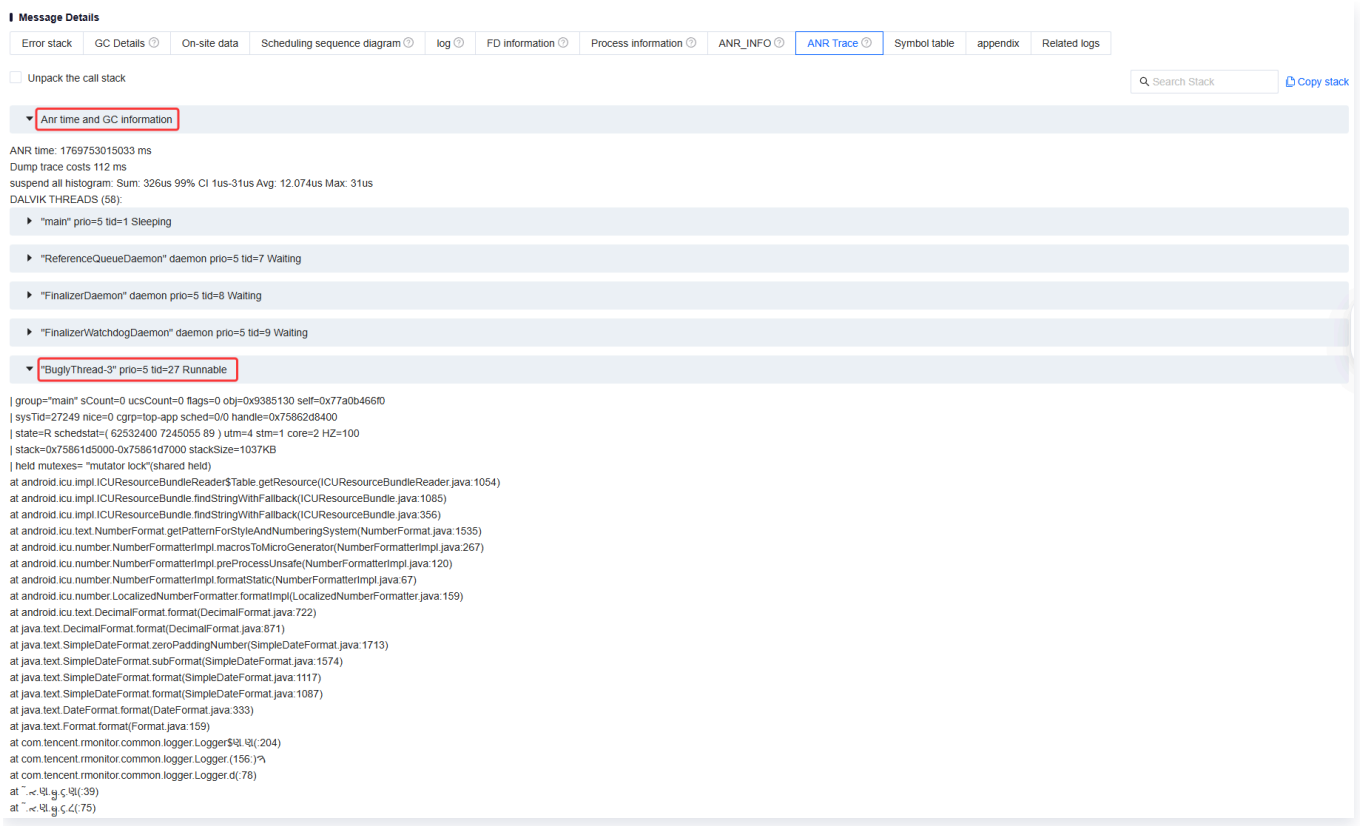
anrMessage.txt

- The information from "anrMessage.txt" is displayed in "ANR_INFO".
- ANR_INFO is a system report generated by the application when an ANR occurs, recording the Activity component where the ANR happened, the Process ID of the affected process, and the cause of the ANR.
- Load represents the system's average load over 1, 5, and 15 minutes when an ANR occurs. The unit of average load is the number of processes, indicating the average count of processes running or waiting for CPU resources during the ANR. CPU Usage indicates detailed CPU utilization during the ANR timeframe. This information can be used to assist in analyzing ANR issues.



trace.zip

- The information from "trace.zip" is displayed in "ANR Trace".
- ANR Trace is a log file that contains events and thread stack traces occurring during ANR in the application.
- Part 1: ANR Time and GC Information. This section shows the duration of the current ANR issue and the GC operations performed by the system.
- Part 2: Thread Stack Information. This section lists stack trace information for all threads running during the ANR. In addition to stack details, each thread entry includes status information such as Thread ID (tid), Thread Priority (prio), Thread State (state), Thread Scheduling Statistics (schedstat), and CPU time used by the thread (utm and stm). Combining this information, especially the thread stack traces, can quickly help pinpoint and analyze the root causes of ANR issues.



iOS

The following content mainly introduces the information recorded for each file in the Attachments Tab of iOS reported case details and the meaning of each piece of information.

Attachments Overview

Logs are currently not supported on the iOS platform.

Classification/Tab	Source
error stack	Exported in the attachment "crash.log" from the crash details API.
Crash Context / Basic Information	Exported to the attachment "detail_data.JSON" from the crash details API.
Crash Context / Custom Fields	From attachment "valueMapOthers.txt"
Crash Context / Page Tracking	From attachment "martianlog.txt"
Information supplemented via the API "setUserValue"	Exported in the attachment user_datas.log
Information supplemented via the callback API attachmentForException	Exported in the attachment "crash_attach.log"

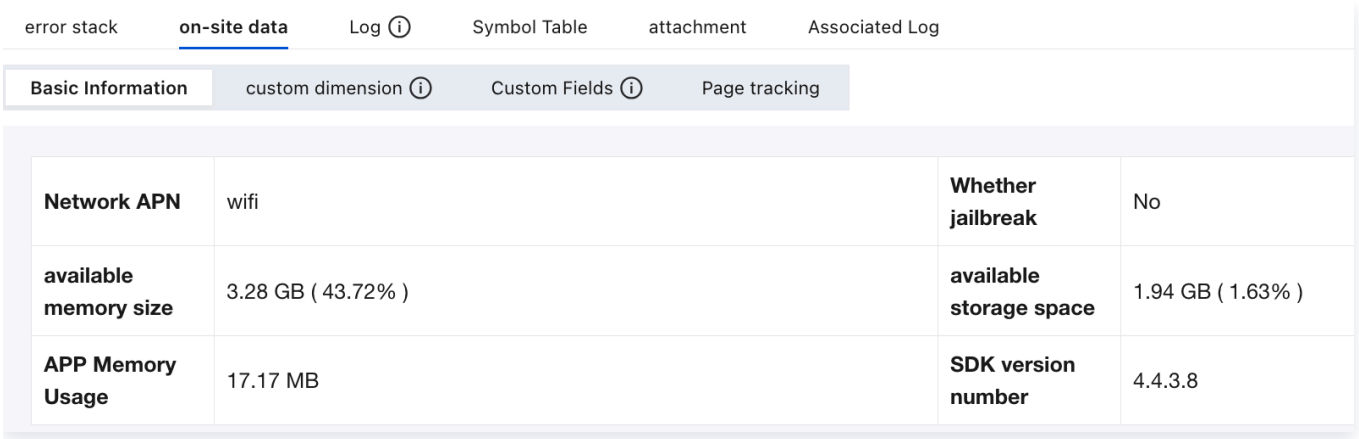
Memory Information	Attachment "meminfo.txt"

crash.log

- Stores error stack information, including the stack of the current exception thread and the stacks of other threads, consistent with the content displayed in the Error Stack Tab.
- Stacks of different threads are separated by blank lines, with the first line indicating the thread ID and thread name.
- It contains the register information of the current exception thread.
- Contains information about all modules.

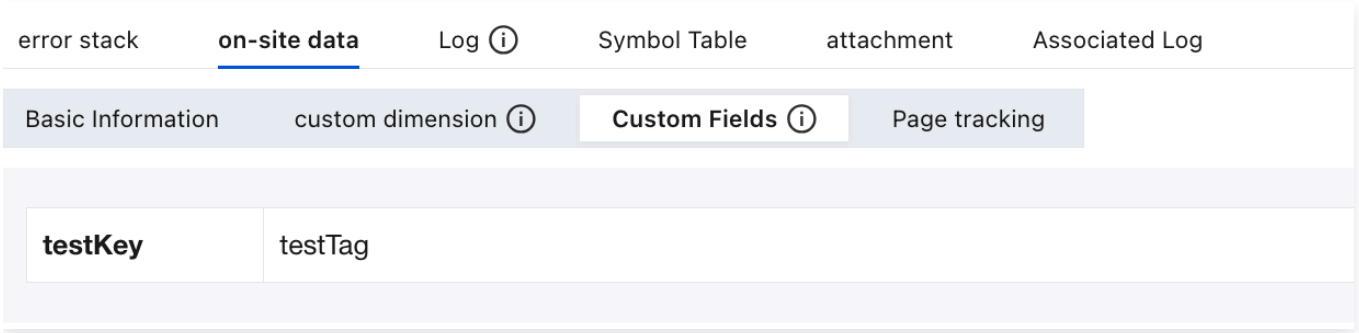
detail_data.JSON

- Basic data and page tracking are on-site information automatically captured by the SDK during exceptions, with the basic data exported in the attachment "detail_data.JSON".
- The "App Channel" displayed in the custom fields comes from the "appInfo" in "detail_data.JSON".



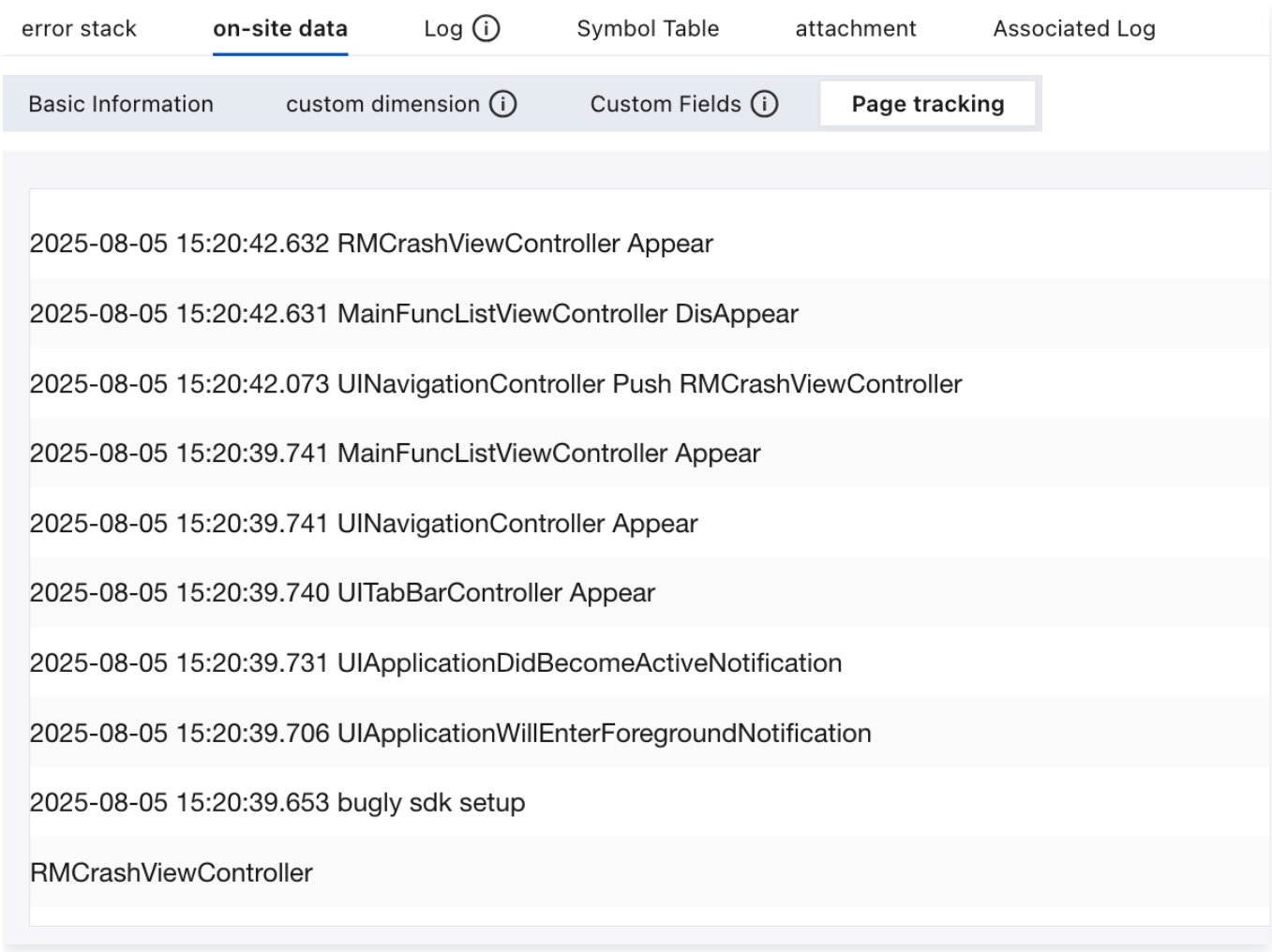
valueMapOthers.txt

- Contains the content set by the business via the API SetUserValue.
- The content displayed in the custom fields section of the on-site data is consistent with the attachment "valueMapOthers.txt".



martianlog.txt

- Page tracking data from the period before the exception occurred is automatically recorded by the SDK.
- This data is also displayed in **on-site data** > **Page tracking** sections.



user_datas.log

- Contains the content set by the business via the API SetUserValue.
- The content displayed in the custom fields section of the on-site data is consistent with the attachment "user_datas.log".

crash_attach.log

Content set by the business via the callback API attachmentForException.

meminfo.txt

Memory overview information at the time of exception.

```
                timestamp : 1703744718
footprint memory bytes : 712707504
 resident memory bytes : 324501504
  virtual memory bytes : 420352311296
            page faults : 190808
               page-ins : 32334
  copy-on-write faults : 4072
     user time in task : 1
   system time in task : 0
```

FAQs

Last updated：2026-05-25 18:53:30

Crash-Related

Why is the local crash not reported, and how to troubleshoot it?

1. Confirm that the SDK has been initialized locally. During initialization, set the SDK's debug mode `builder.debugMode` to true. Use the `logcat -s CrashReport eup` command in adb debugging to filter the logs. If you see the following information printed by RUM Pro, it indicates that the SDK has been initialized locally.

```
06-13 11:03:09.597 27513 27513 W eup     : rumpro debug mode is
enabled. Please disable isDebug when you release. -- Running in debug
model for 'isDebug' is enabled. Please disable it when you release.
06-13 11:03:09.597 27513 27513 E eup     : ----------------------------
------------------------------------------------------------------
06-13 11:03:09.597 27513 27513 W eup     : rumpro debug mode will
exhibit the following behavior -- The following list shows the
behaviour of debug model:
06-13 11:03:09.597 27513 27513 W eup     : [1] Output detailed rumpro
SDK log -- More detailed log of rumpro SDK will be output to logcat;
06-13 11:03:09.597 27513 27513 W eup     : [2] Every crash will be
uploaded immediately -- Every crash caught by rumpro will be uploaded
immediately.
06-13 11:03:09.597 27513 27513 W eup     : [3] Custom logs will be
output to Logcat. -- Custom log will be output to logcat.
06-13 11:03:09.597 27513 27513 E eup     : ----------------------------
------------------------------------------------------------------
```

2. Confirm whether exceptions can be properly captured by the SDK. After triggering a crash or ANR issue, if the following log is printed, it indicates that the crash exception can be successfully captured by the SDK.

```
06-13 11:11:41.777 29271 30104 E eup     : #++++++++++Record By
rumpro++++++++++#
06-13 11:11:41.777 29271 30104 E eup     : # You can use
rumpro(https://console.cloud.tencent.com/monitor/rumpro) to get more
Crash Detail!
```

```
06-13 11:11:41.777 29271 30104 E eup      : # PKG NAME:
com.tencent.demo.rumprodemo
06-13 11:11:41.777 29271 30104 E eup      : # App VER: 1.0.0
06-13 11:11:41.777 29271 30104 E eup      : # SDK VER: 4.3.0.30-
SNAPSHOT
06-13 11:11:41.778 29271 30104 E eup      : # LAUNCH TIME: 2023-06-13
11:09:41
06-13 11:11:41.778 29271 30104 E eup      : # CRASH TYPE: JAVA_CRASH
06-13 11:11:41.778 29271 30104 E eup      : # CRASH TIME: 2023-06-13
11:11:41
06-13 11:11:41.778 29271 30104 E eup      : # CRASH PROCESS:
com.tencent.demo.rumprodemo
06-13 11:11:41.778 29271 30104 D eup      : isAppForeground:true
06-13 11:11:41.778 29271 30104 E eup      : # CRASH FOREGROUND: true
06-13 11:11:41.778 29271 30104 E eup      : # CRASH THREAD: Thread-9
06-13 11:11:41.778 29271 30104 E eup      : # REPORT ID: cf5fb020-4c42-
4cee-8d78-299940bd4d56
06-13 11:11:41.779 29271 30104 E eup      : # CRASH DEVICE: V2034A
UNROOT
06-13 11:11:41.780 29271 30104 E eup      : # RUNTIME AVAIL
RAM:5672812544 ROM:87523766272 SD:87314046976
06-13 11:11:41.780 29271 30104 E eup      : # RUNTIME TOTAL
RAM:8068177920 ROM:116628795392 SD:116419080192
06-13 11:11:41.780 29271 30104 E eup      : # CRASH STACK:
06-13 11:11:41.780 29271 30104 E eup      : java.lang.RuntimeException:
This Crash create for Test! You can go to rumpro see more detail!
06-13 11:11:41.780 29271 30104 E eup      :     at
com.tencent.rumpro.proguard.dz.dq(Unknown Source:16)
06-13 11:11:41.780 29271 30104 E eup      :     at
com.tencent.rumpro.library.rumpro.testCrash(Unknown Source:28)
06-13 11:11:41.780 29271 30104 E eup      :     at
com.tencent.demo.rumprodemo.MainActivity$2$1.run(MainActivity.java:85)
06-13 11:11:41.780 29271 30104 E eup      :     at
java.lang.Thread.run(Thread.java:919)
06-13 11:11:41.780 29271 30104 E eup      :
#++++++++++++++++++++++++++++++++++++++++#
```

3. Check whether exceptions can be reported normally. You can search for the `crash upload` keyword in the filtered logs mentioned above, which will print logs indicating `crash type + whether the rep`

`ort was successful`. If the report is successful, you will see a sample print like the following. If unsuccessful, please contact us.

```
06-13 11:25:44.752  1566  1795 I eup      : java.lang.RuntimeException,
crash upload success!
```

Cause Analysis of Other Issues:

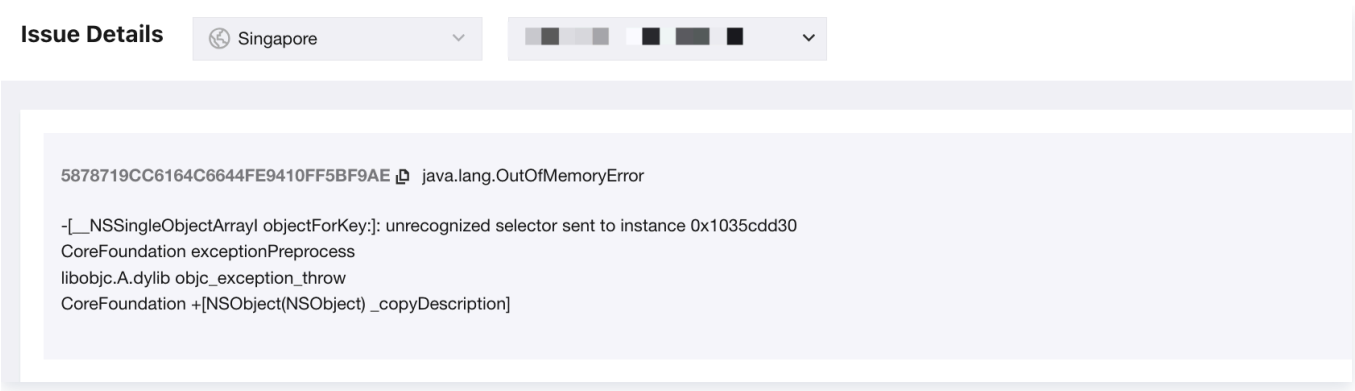
- There might be a slight delay in reporting. If you have confirmed that the log indicating successful reporting has been printed, you can wait a few minutes and refresh the page to check whether it has been reported.
- If the SDK fails to capture crash exceptions properly, check whether other logic in your service has registered exception capture functions or signals, or whether other third-party exception capture SDKs are integrated. **It is recommended to integrate only one exception capture SDK in your service**, as other SDK capture logic may interfere with the SDK's exception capture.

Why is there no data displayed for certain information (such as process information)?

- **Process Information**: Process information will not be reported for Java crashes. For Native crashes or ANR issues, the reporting path for process information is separate from crash reporting. After a crash occurs, the user must restart the App to report process information, which will be automatically associated with the corresponding exception case. If the process information section is not shown, it may be because the user did not restart the App.
- **Partial On-Site Data for Native Crashes**: During a Native crash, if JVM-related on-site data cannot be obtained, the reason may be that the JVM environment is corrupted, preventing the crash handling logic from entering the Java layer for collection.
- **Other Information**: Related to the crash capture processing chain, possibly due to system logic interruption causing incomplete collection of information. For details, please contact us for assistance.

Why are the same crash issues not grouped under the same Issue?

RUM Pro extracts certain stack frames from the crash **stack trace** as signatures for grouping. The extracted stack signatures are displayed at the top of the **Issue Details** page. If stack traces that visually appear identical are not grouped under the same Issue, it is due to differences between the stack traces that result in variations in the extracted signatures.
If the extracted stack signature is deemed not reasonable enough, you can contact us for consultation and feedback.

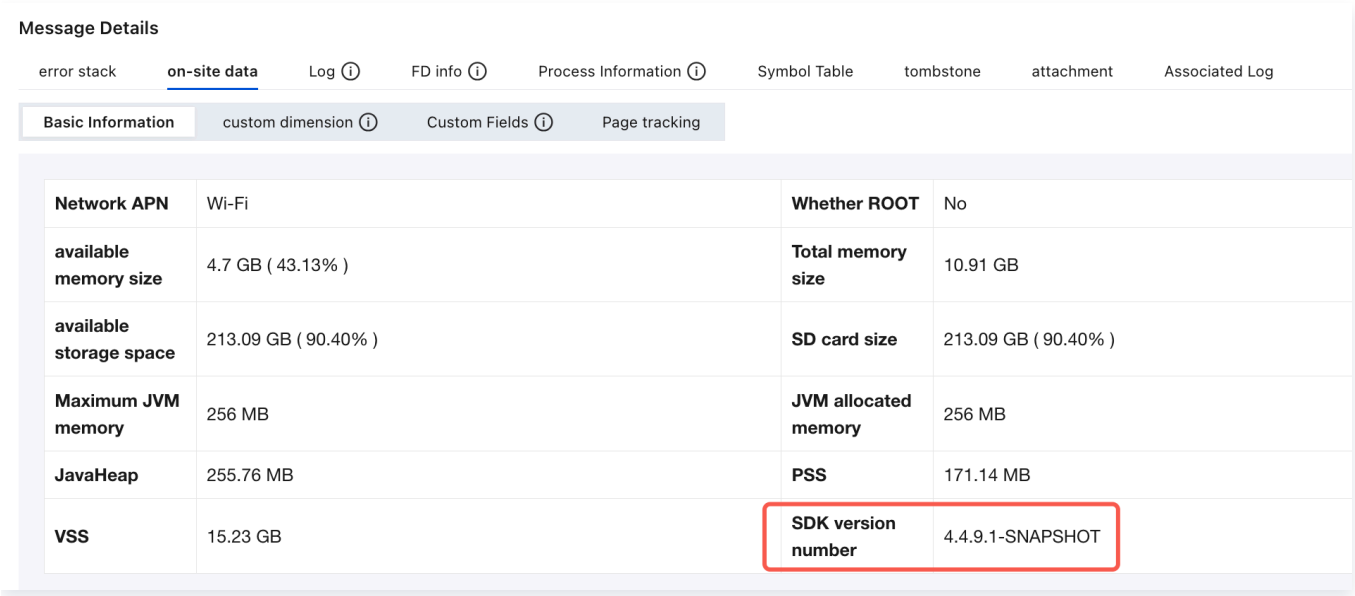


Why does the connected device count fluctuate significantly? Why does the crash rate fluctuate significantly?

- The count of connected devices is calculated based on device IDs. Incorrect configuration of device IDs (uniqueId) may lead to abnormal statistics for connected devices. For example, setting identical or fixed values for different devices' IDs is not allowed, as fluctuations in connected device counts may be affected by this.
- Fluctuations in crash rate may be influenced by the connected device count. If the connected device count remains stable while the crash rate shows significant fluctuations, the issue should be investigated based on specific business logic.

How to determine the SDK version reported in crash issues?

You can first filter App versions using the criteria to locate an individual reported case for the current App version. In the **on-site data** of the reported case, you can locate the **SDK version number** field to identify the version number of the currently integrated SDK.



Why is the reported stack trace not translated?

If the **error stack** indicates that the stack is untranslated, you can perform the following self-checks first:

- Symbol tables are matched using `version number + build number` for Java mapping symbol tables, while UUID is used for matching Native so symbol tables.
- If it is a Native stack, check whether the uploaded Native SO has been stripped. Stripped SO files do not contain symbol tables and cannot be translated. You can use the `file` command to check if the SO has been stripped. Locally execute `file libxxx.so` to print SO-related information. *not stripped* indicates an unstripped SO, while *stripped* indicates a stripped SO.
- If the issue persists, please refer to the Symbol Table FAQ.

ANR related (Android)

Why are local ANRs not being reported? How to troubleshoot this issue?

Please refer to the Crash Reporting Troubleshooting Process.

Why is some information (such as ANR_INFO and ANR Trace) not displayed?

- **ANR Information**: Some ANR details, such as ANR Trace and ANR_INFO, may fail to be obtained due to vendor-customized ANR handling logic. On certain device models, the system immediately terminates the process upon detecting ANR signals, leaving insufficient time to capture the corresponding ANR information.
- **Other Information**: Related to the crash capture processing chain, possibly due to system logic interruption causing incomplete collection of information. For details, please contact us for consultation and feedback.

Why are identical ANR issues not grouped under the same Issue?

Please refer to Why identical crash issues are not grouped under the same Issue.

What are the conditions for ANR occurrences?

In Android, the system detects an ANR (Application Not Responding) event occurrence in the following scenarios:

- When an application performs time-consuming operations on the main thread, such as network requests or lengthy computations, without timely response to user input events (e.g., clicking the screen or pressing keys), the system considers the application unresponsive.
- If an application fails to respond to user input events within 5 seconds, the system considers it unresponsive.
- When an application performs time-consuming operations in BroadcastReceiver or Service components without timely completion and release of the wake lock, the system considers the application unresponsive.

How to optimize for ANR; what optimization suggestions are there?

- Avoid performing time-consuming operations on the main thread.
- Handle time-consuming operations with asynchronous tasks.
- Use multithreading to handle time-consuming operations.
- Use Handler to handle messages.

Jank-Related

How is the suspension rate calculated?

- Suspension rate is calculated per device on a daily basis, including the suspension time when the frame time exceeds 200ms while the application is in the foreground, and the duration the application is in the foreground.
- Device suspension rate = Total suspension time of the device in a day (in seconds) / Total foreground duration of the device in a day (in hours). The unit of suspension rate is seconds per hour (s/h).

How is FPS counted?

FPS is aggregated by scenario, currently counting the FPS of actual UI refreshes in the application while excluding data from periods with no UI refresh. The FPS data displayed on the platform undergoes normalization to accommodate different screen refresh rates.

How is the time consumption of stagnant methods calculated?

- In stuttering issue monitoring, method execution time is estimated through continuous stack sampling combined with sampling intervals. For example, if a stack is sampled consecutively 6 times with a 52ms interval, its execution time is calculated as 6 × 52 = 312ms.
- This estimation method has a margin of error equal to the sampling interval, but the jank duration is accurate, representing the execution time of a UI thread message. The current jank monitoring determines whether the application has experienced jank by tracking the processing time of UI thread messages.

In the stack tree or flame graph, what does a stack with a processing time of 52ms represent?

If a stack's recorded time is 52ms, it indicates the stack was sampled only once during continuous sampling and does not necessarily represent its actual execution time. When a stack is sampled N times (where N > 1), its execution time must be greater than (N-1)*52ms.

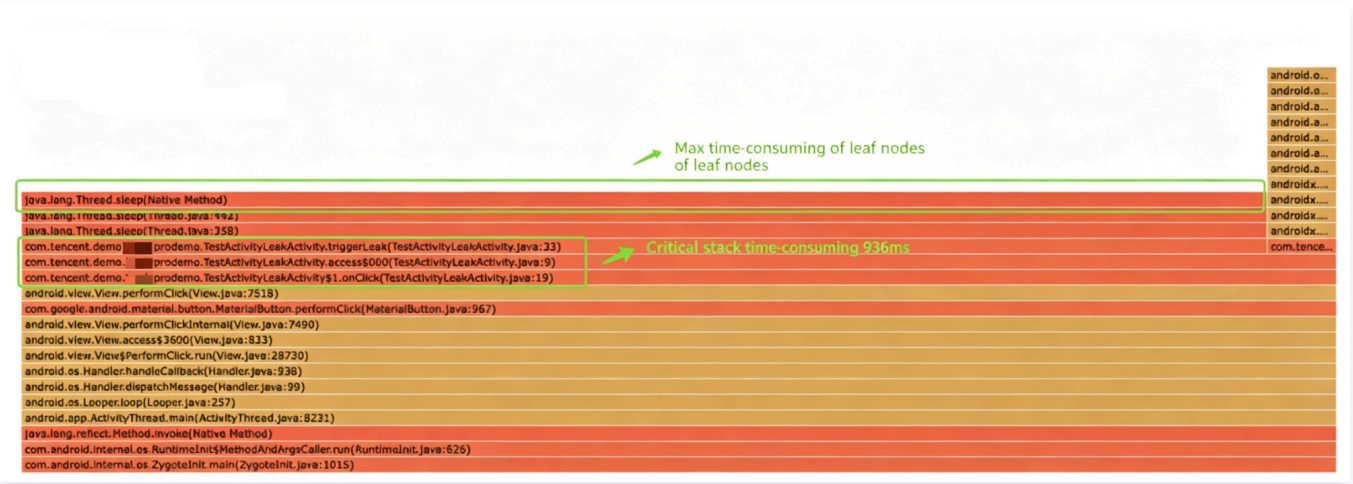
> **Sampling Interval:**

> Device models for both Android and iOS may have different sampling intervals. You can determine the sampling interval for this specific case by combining the sampling count displayed in the time slice with the calculated sampling duration.

What is critical stack execution time?

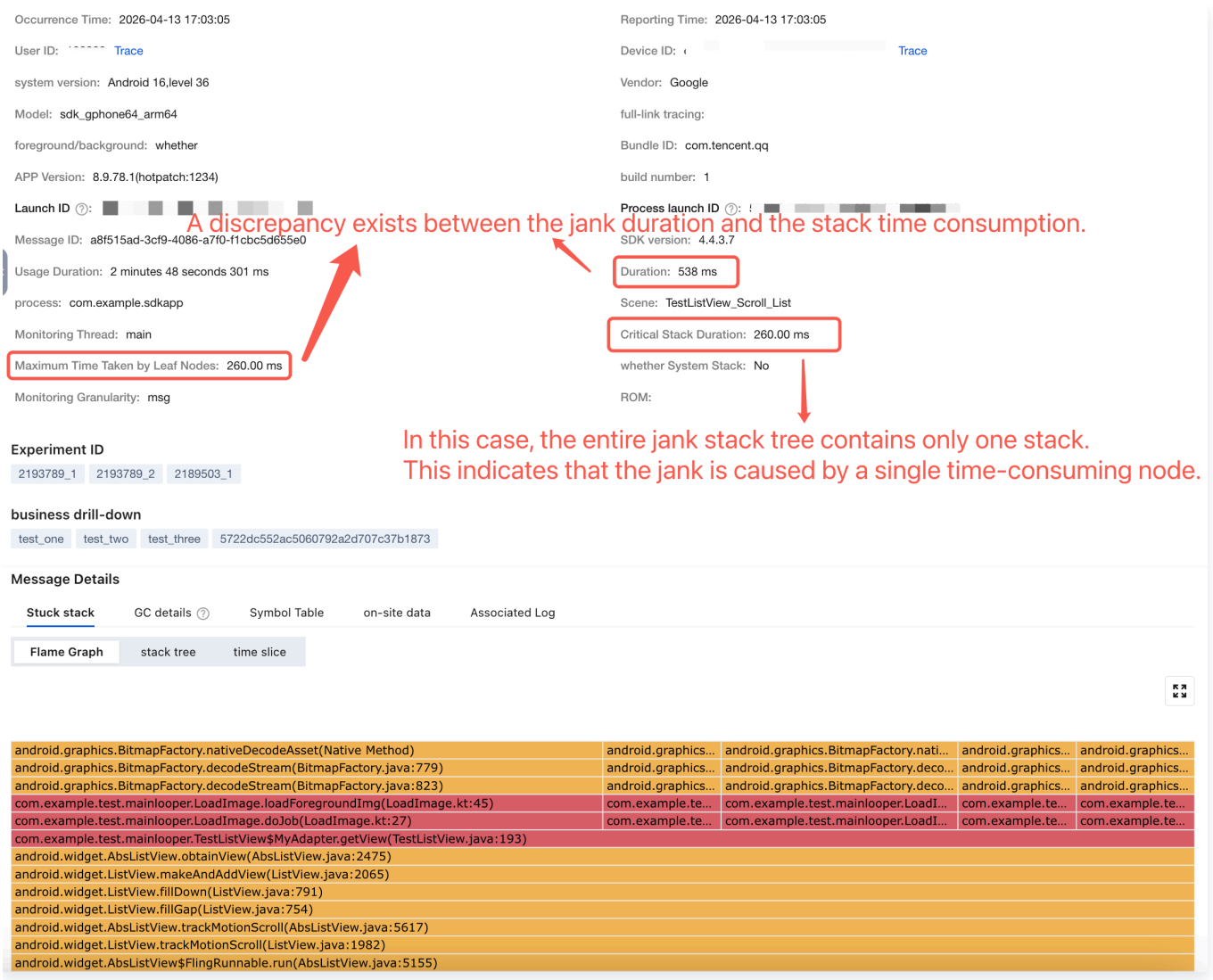
Each case contains a stack tree. To aggregate issues, features need to be extracted. Extracting key stacks incurs time consumption, causing stuttering. As shown below is the key stack of a case.

```
com.tencent.demo.[redacted]prodemo.TestActivityLeakActivity$1.onClick(TestActivityLeakActivity.java)
com.tencent.demo.[redacted]prodemo.TestActivityLeakActivity.access$000(TestActivityLeakActivity.java)
com.tencent.demo.[redacted]prodemo.TestActivityLeakActivity.triggerLeak(TestActivityLeakActivity.java)
```



The significance of the maximum time consumption at leaf nodes

The maximum time consumption of leaf nodes refers to the highest time consumption among all leaf nodes in a stack tree. If a stuttering issue exhibits a high maximum leaf node time consumption—especially when approaching the key stack time—it indicates that the leaf node stack is the primary cause of stuttering. Scenarios such as deadlocks, UI threads performing I/O operations, or UI threads accessing databases often result in significantly high maximum time consumption at leaf nodes.

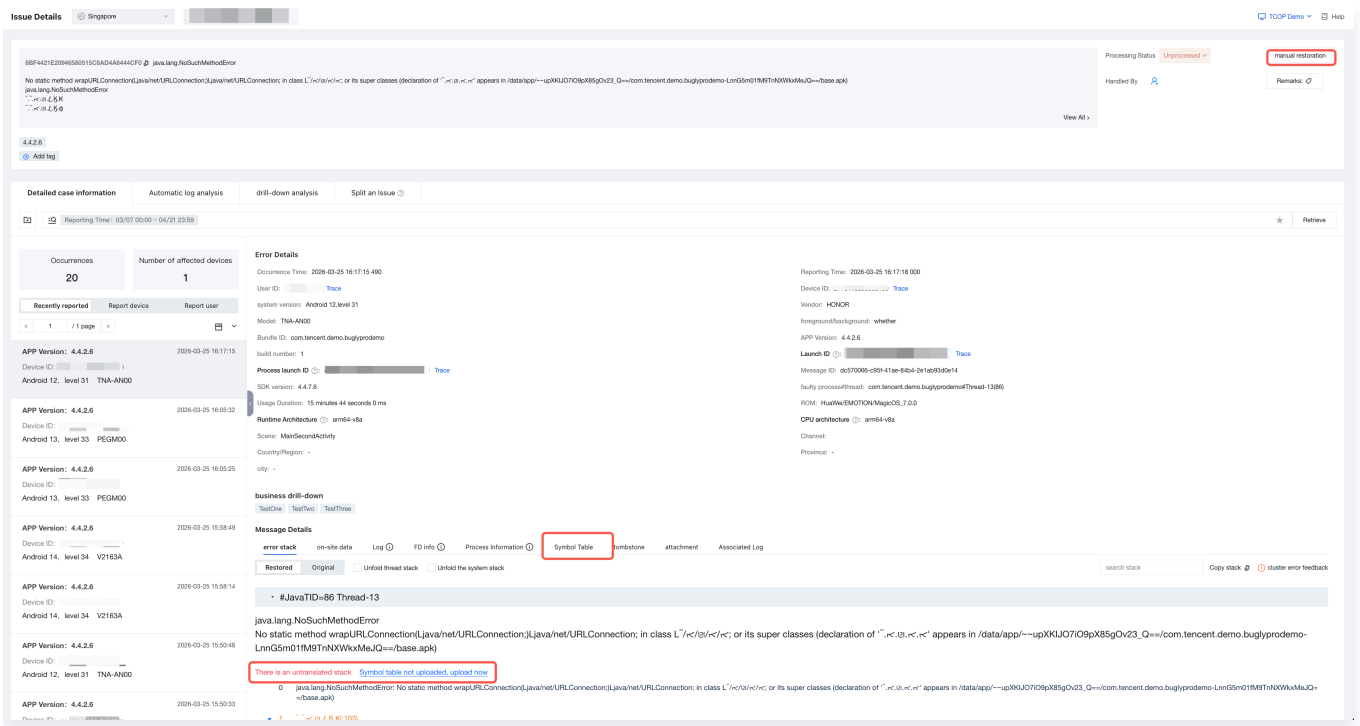


Symbol Table Related

Why does the exception stack prompt appear untranslated?

The main reason the exception stack trace shows untranslated is that the corresponding symbol table has not been uploaded.

1. You can click **Symbol table not uploaded,upload now** to switch to the Symbol Table Tab and upload the specified symbol table according to the instructions.
2. After confirming the symbol table has been successfully uploaded, click **Issue Details** and then select **manual restoration** at the top to perform stack restoration.



> **Note:**
> After manually restoring the stack, the exception stack trace changes, which alters the case features generated from it. This will cause the restored case to be aggregated into other issues.

Why file drag-and-drop fails during web upload

Whether it's an so symbol table, dSYM file, or Java mapping file, users must first compress them into a zip file before dragging and dropping for upload. The upload component only supports zip files and does not accept files in other formats.

- For mapping files, they must be named mapping.txt, compressed into mapping.txt.zip using zip, and then dragged into the upload area.
- For so files, first compress the target so file into a zip archive, such as compressing libnative.so into libnative.so.zip, then drag it into the upload area.
- For dSYM files, first compress the target dSYM file into a zip archive, such as compressing RUMPro.dSYM into RUMPro.dSYM.zip, then drag and upload the RUMPro.dSYM.zip file.

Why does the symbol table file status still show as not uploaded when the webpage indicates a successful upload?

There are two possibilities for this situation:

- Case 1: The symbol table failed to be uploaded successfully. Refer to Why File Upload or Drag-and-Drop Failed to ensure the symbol table file is properly named and compressed into a zip archive.
- Case 2: The symbol table file has been uploaded, but there is a delay in processing. You may need to refresh the page.

Why does the upload prompt indicate success for the mapping file, but the uploaded symbol table file cannot be found?

In this case, you can first try refreshing the page. If the uploaded symbol table still cannot be found after refreshing, or if the symbol table Tab indicates that the file has not been uploaded, it may be due to the mapping file not being properly named.

- For mapping files, they must be named mapping.txt, compressed into mapping.txt.zip using zip, and then dragged into the upload area.
- The mapping file is identified using `App version number + build number` as the Key. Please verify whether the version number and build number are correctly entered.

Why is a Java environment required for uploading symbol tables?

The symbol table extraction tool relies on the Java environment. It extracts only essential information, which can significantly reduce the file size required for upload.

The symbol table upload failure prompt indicates a UUID mismatch

The UUID of the symbol table changes with each build. Therefore, only the symbol table file generated from that specific build can restore crash reports uploaded after the same build.

Failure to configure symbol table restoration will not affect exception reporting, but it will impact the ability to restore crash stack traces

Without symbol tables, stack traces cannot be restored to the classes or methods in the code along with source file line numbers, which may impact crash deduplication. However, this does not affect exception reporting.

Do I need to configure symbol tables for each version?

Yes, each App version requires a corresponding symbol table (Java Mapping files, so Symbol files, and Apple C++ dSYM files). The configuration is only effective for the specified version, and duplicate configurations will override existing ones.

Does the Java crash stack lack line number information (e.g., displaying "Unknown Source")?

Add the setting to keep line numbers in the ProGuard configuration, for example:

```
-keepattributes SourceFile,LineNumberTable
```

Are symbol tables required for Unity Android projects?

- For Unity projects, the Java code generated by the Android project consists of only a few entry classes and does not require ProGuard obfuscation. Therefore, there is no need to configure symbol tables (i.e., mapping information).
- For Unity projects, the default Native libraries (such as libmono.so, libunity.so) loaded in the Android project do not have debug symbols. Thus, developers are unable to acquire the corresponding symbol information for configuration.

> **Note:**
> If developers have independently developed functional components (.jar or .so) integrated into Unity projects, they need to configure corresponding symbol table information (with the "Development Build" option enabled, the debug version of .so files is located in the development directory. These can be generated using the platform's symbol table tool and uploaded to version control).

Why does the stack remain unreadable after the symbol table has been correctly uploaded?

- To resolve this issue: it is recommended to re-upload the symbol table and then click **Manual Restore** to trigger re-restoration.
- If the stack trace remains unreadable after re-uploading the symbol table, it is likely that there is an issue with the symbol table or the captured stack. You can manually deobfuscate using the addresses in the stack offline to verify whether the system-captured stack can be properly deobfuscated.

How to determine whether the uploaded symbol table is valid?

Check whether the source file names and line numbers have been restored.

What does "invalid file" mean; how to handle it?

The uploaded file format is incorrect; the system detected that it is not a symbol table file. The system scanning rule is: traversing .so files in the first-level directory, otherwise considered invalid. It is recommended to upload a single .so file. Future versions will optimize the scanning rules.