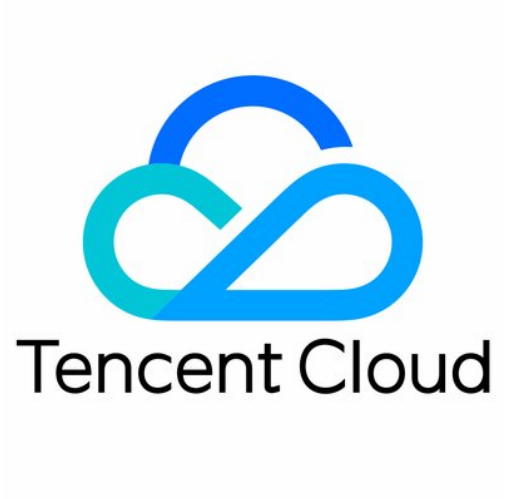
Serverless Application Center

Operation Guide

Product Documentation

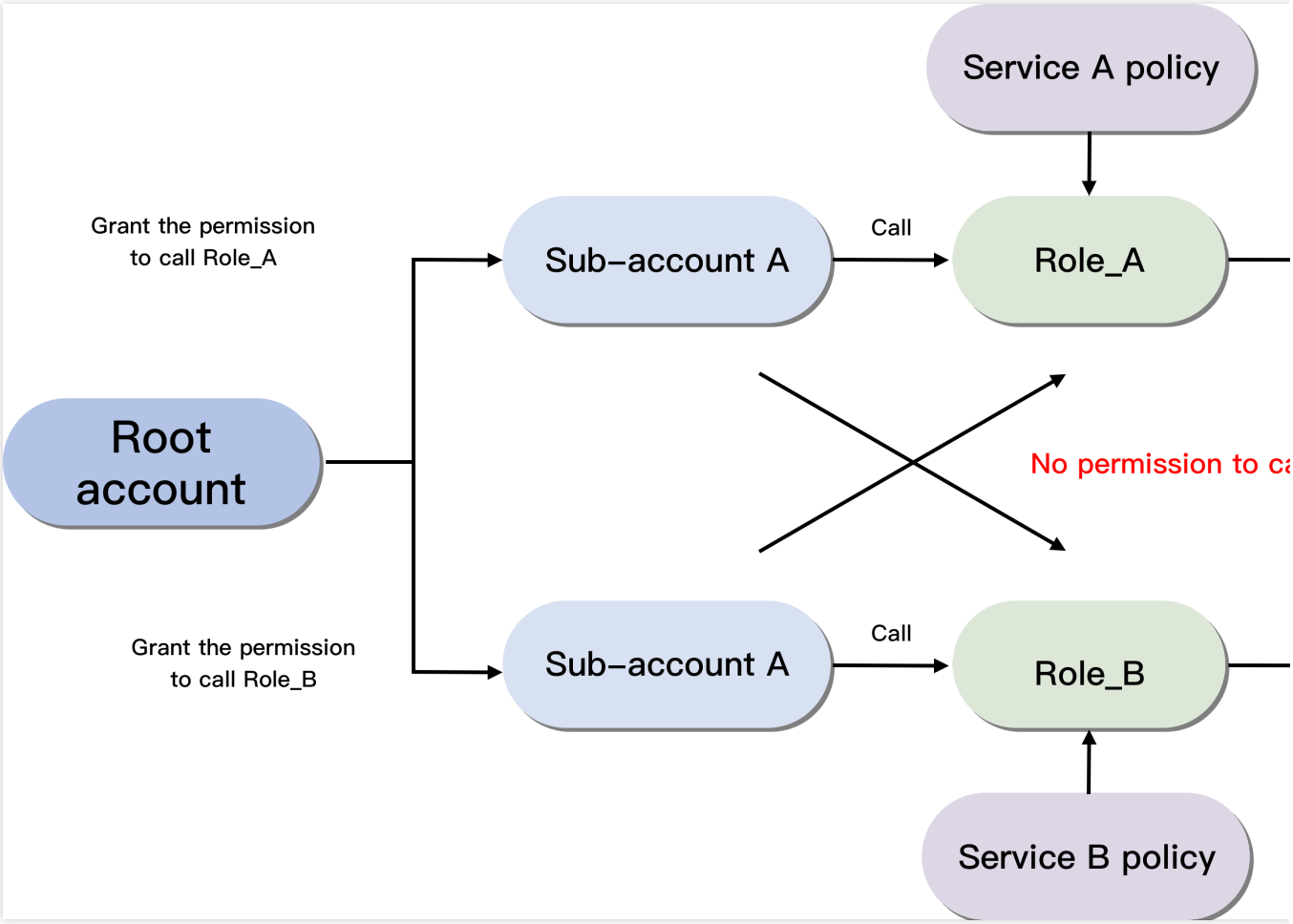


Contents

Operation Guide
Permission Configuration
Configuring Role for Specified Operation

Last updated：2024-12-02 10:48:10

In addition to the default role `SLS_QcsRole`, a root account can also create multiple custom roles and assign them to sub-users, so that they can have only the policies granted by the corresponding roles as needed, which can implement permission control. Its flowchart is as follows:



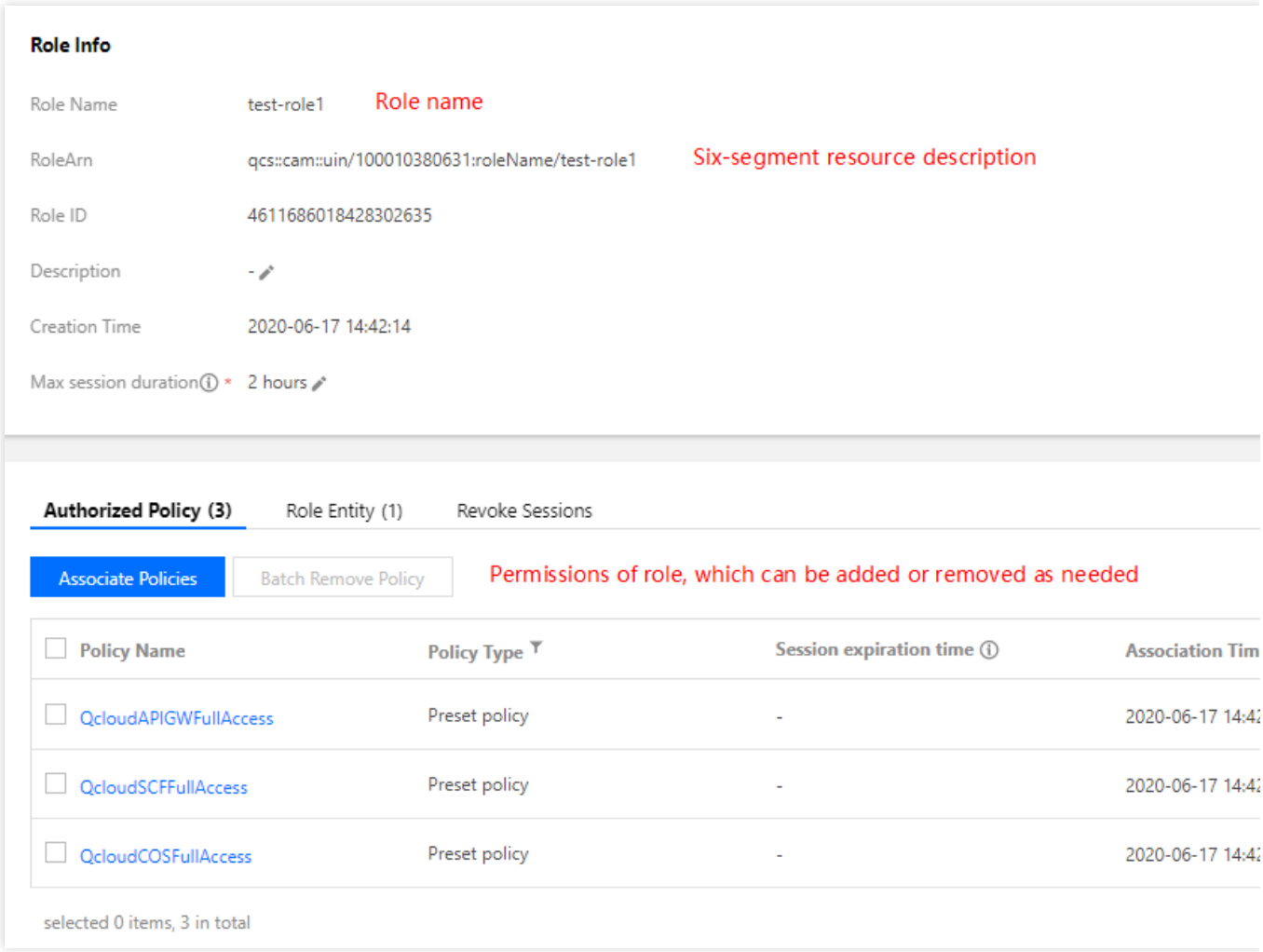
Root Account Configuration Process

You can create a sub-account, configure a role, and grant the role the corresponding policies. The following uses the deployment of an SCF function triggered by API Gateway as an example:

Creating sub-account role

1. Log in to the CAM Console with your root account and click **Role** on the left sidebar.
2. On the role page, click **Create Role** and select **Tencent Cloud Service** as the role entity.
3. Among the services supporting roles, select **Serverless Framework (sls)** and click **Next**.
4. Select presets policies **QcloudCOSFullAccess**, **QcloudAPIGWFullAccess**, and **QcloudSCFFullAccess** and click **Next**.
5. Enter a role name such as `test-role1` and click **Complete**.

You can click the role name to view the role page after configuration:



Configuring role policy

1. Click **Policy** on the left sidebar to enter the policy management page.
2. On the policy management page, click **Create Custom Policy** and select **Create by Policy Syntax**.
3. Select **Blank Template** as the policy template and click **Next**.
4. Enter the policy name and content and click **Complete**.

Bind the role policy. Here, you need to populate the `resource` parameter with the six-segment description of the

role to be bound to the sub-account:

```
{
 "version": "2.0",
 "statement": [
     {
         "action": [
             "cam:PassRole"
         ],
         "resource": [
             # Six-segment role description (such as `qcs::cam::uin/123456789:roleN
         ],
         "effect": "allow"
     }
 ]
}
```

Note:

The role resource description can be obtained on the role information page.

Associating sub-user with policy

1. Click **User** > **User List** on the left sidebar to enter the user list page.
2. Select the sub-user to be authorized and click **Authorize** in the "Operation" column.
3. Select the created policy and the preset policy **QcloudSLSFullAccess** in the policy list and click **OK** to associate them with the target sub-account so as to bind the role.
4. **(Optional)** If you think that `QcloudSLSFullAccess` contains excessive permissions, you can create a custom policy to grant a specified resource the SLS call permission with the following policy template:

```
{
 "version": "2.0",
 "statement": [
     {
         "action": [
             "sls:*"
         ],
         "resource": [
             # Enter the project resource name (such as `qcs::sls:ap-guangzhou::app
         ],
         "effect": "allow"
     }
 ]
}
```

Note:

The project resource description must strictly follow the CAM specifications. You can also describe the resource more specifically by entering a function name or stage name.

Sub-account Configuration Process

Create a Serverless project locally, add a global configuration item `configRole` in the `serverless.yaml` configuration file, and enter the role name. After the backend successfully checks the permissions, the deployment will be completed.

```
# serverless.yml

component: scf # Name of the imported component, which is required. The `tencent-sc
name: scfdemo # Name of the instance created by this component, which is required
org: test # Organization information, which is optional. The default value is the `
app: scfApp # SCF application name, which is optional
stage: dev # Information for identifying environment, which is optional. The defaul

globalOptions:
   configRole: test-role1     # Name of specified role, which is optional

inputs:
  name: scfFunctionName
  src: ./src
  runtime: Nodejs10.15 # Runtime environment of function. Valid values: Python2.7,
  region: ap-guangzhou
  handler: index.main_handler
  events:
    - apigw:
        name: serverless_api
        parameters:
          protocols:
            - http
            - https
          serviceName:
          description: The service of Serverless Framework
          environment: release
          endpoints:
            - path: /index
              method: GET
```

Note:

If no role is bounded, the sub-account will use `SLS_QcsRole` for SLS deployment by default, and the `configRole` parameter does not need to be set in the configuration file.

Once a role is bounded, please check the `configRole` name in the `yaml` file carefully. An error will be reported if the value is incorrect or empty. A sub-account can use only bounded roles but cannot use other roles.

Granting Permission to Sub-account

If you want to grant a permission to a sub-account, you need to provide the role name and the name of the policy to be associated together to the root account. Then, the root account can grant the permission in CAM Console > **Role**.

Account and Permission Configuration

Last updated：2024-12-02 10:48:10

This document describes several authorization methods of Serverless Cloud Framework and demonstrates actual operations by configuring sub-account permissions.

Prerequisites

Serverless Cloud Framework enables you to quickly deploy your project to **Serverless Application Center (SAC)**. Before the deployment, please make sure that you have registered a Tencent Cloud account.

Authorization Method

Authorizing by scanning code

When you perform deployment by running `scf deploy`, you can scan the QR code for quick authorization and deployment. After you grant the authorization by scanning the code, temporary key information (which will expire in 60 minutes) will be generated and written to the `.env` file in the current directory.

```
TENCENT_APP_ID=xxxxxx     # `AppId` of authorizing account
TENCENT_SECRET_ID=xxxxxx  # `SecretId` of authorizing account
TENCENT_SECRET_KEY=xxxxxx # `SecretKey` of authorizing account
TENCENT_TOKEN=xxxxx       # Temporary token
```

For more information about the permissions obtained during quick authorization, see scf_QcsRole role permission list.

Note

If your account is a **Tencent Cloud sub-account**, policy authorization needs to be first configured by using the root account. For more information about the configuration, see Sub-account Permission Configuration.

Authorizing with local key

To eliminate the need for repeated authorization due to information expiration in case of authorization by scanning the code, you can authorize with a key. Create an `.env` file in the root directory of the project to be deployed and configure the Tencent Cloud `SecretId` and `SecretKey` information:

```
# .env
TENCENT_SECRET_ID=xxxxxxxxxx # `SecretId` of your account
TENCENT_SECRET_KEY=xxxxxxxx # `SecretKey` of your account
```

You can obtain `SecretId` and `SecretKey` in API Key Management.

Note

To ensure the account security, we recommend you use a **sub-account** key for authorization. The sub-account can be used to deploy the project only after being granted the relevant permissions. For more information about the configuration, see Sub-account Permission Configuration.

Configuring with permanent key

You can run the `scf credentials` command to quickly set the persistent storage of the global key information. This command must be configured under the created SCF project. Make sure that you have created a project with `serverless.yml` by using `scf init` or manually.

Below are all the commands:

```
scf credentials   Manage global user authorization information
  set                    Store user authorization information
      --secretId / -i          (Required) `secretId` of the Tencent Cloud CAM accou
      --secretKey / -k         (Required) `secretKey` of the Tencent Cloud CAM acco
      --profile / -n {name}    Authorization name, which is `default` by default
      --overwrite / -o         Overwrite the key with an existing authorization nam
  remove                 Remove user authorization information
      --profile / -n {name}    (Required) Authorization name
  list                   View user authorization information
```

Configure global authorization information:

```
# Configure authorization information through the default profile name
$ scf credentials set --secretId xxx --secretKey xxx

# Configure authorization information through the specified profile name
$ scf credentials set --secretId xxx --secretKey xxx --profile profileName1

# Update the authorization information in the specified profile name
$ scf credentials set --secretId xxx --secretKey xxx --profile profileName1 --overw
```

Delete the global authorization information:

```
$ scf credentials remove --profile profileName1
```

View all the current authorization information:

```
$ scf credentials list
```

Perform deployment by using the global authorization information:

```
# Deploy through the default profile
$ scf deploy
# Deploy through the specified profile
```

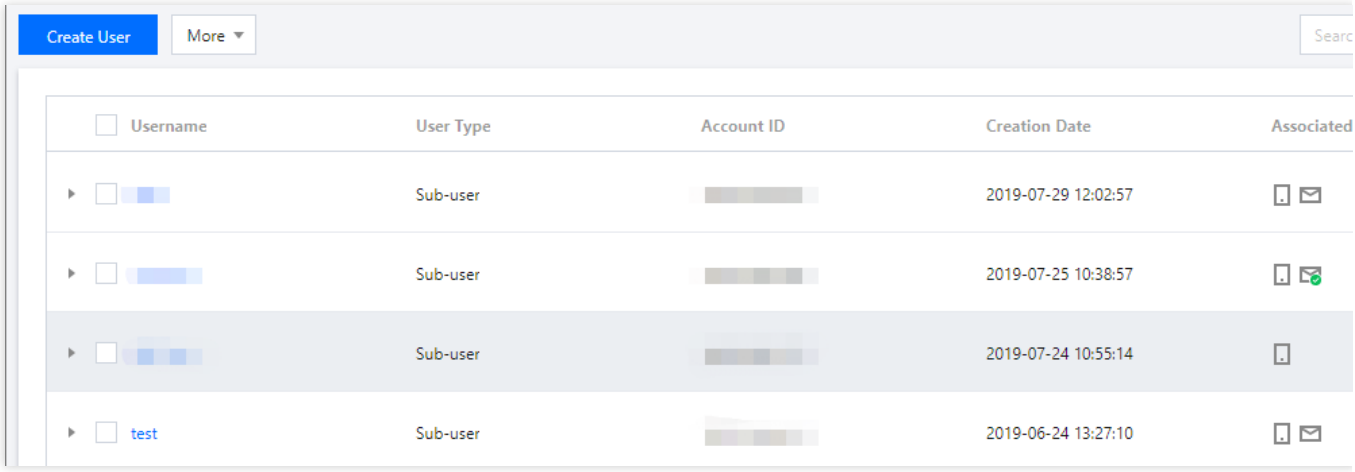
```
$ scf deploy --profile newP
# Ignore global variables and scan the QR code for deployment
$ scf deploy --login
```

Sub-account Permission Configuration

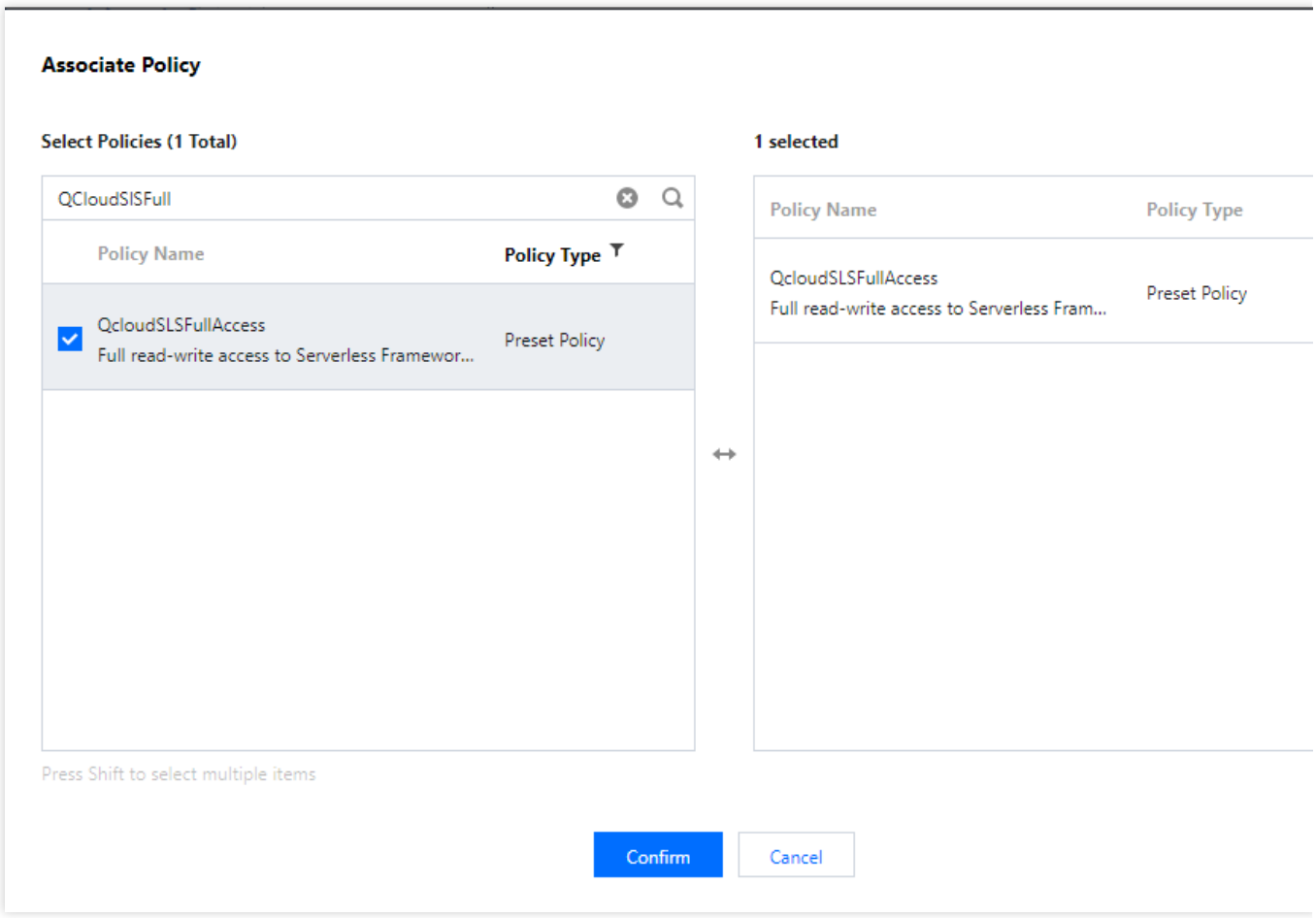
Directions

If you use a Tencent Cloud sub-account, it does not have the operation permissions by default; therefore, it needs to be authorized by the **root account (or a sub-account with the authorization permission)** in the following steps:

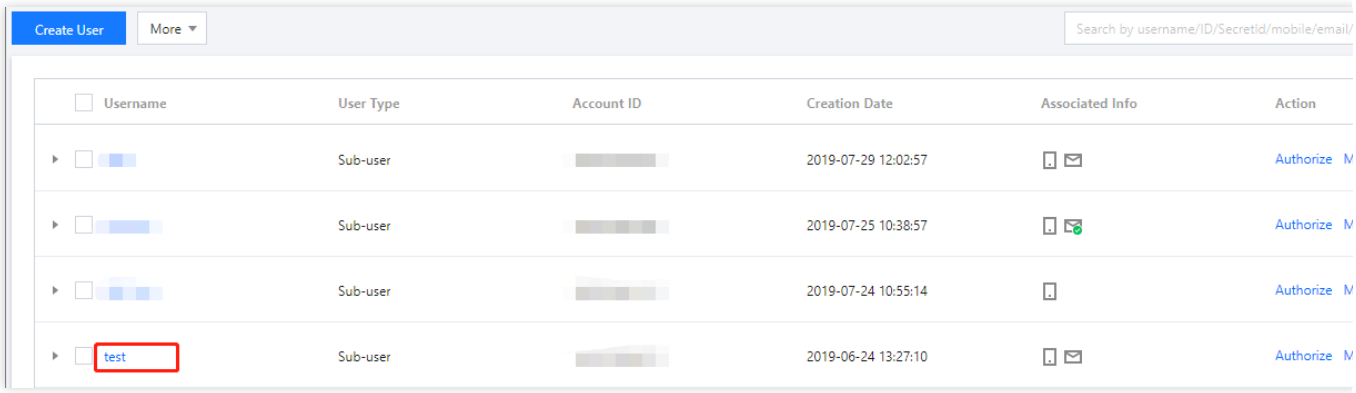
1. On the CAM User List page, select the target sub-account and click **Authorize** in the **Action** column.



2. Search for and select `QcloudscfFullAccess` in the pop-up window and click **OK** to grant the sub-account the permission to manipulate all Serverless Cloud Framework resources.

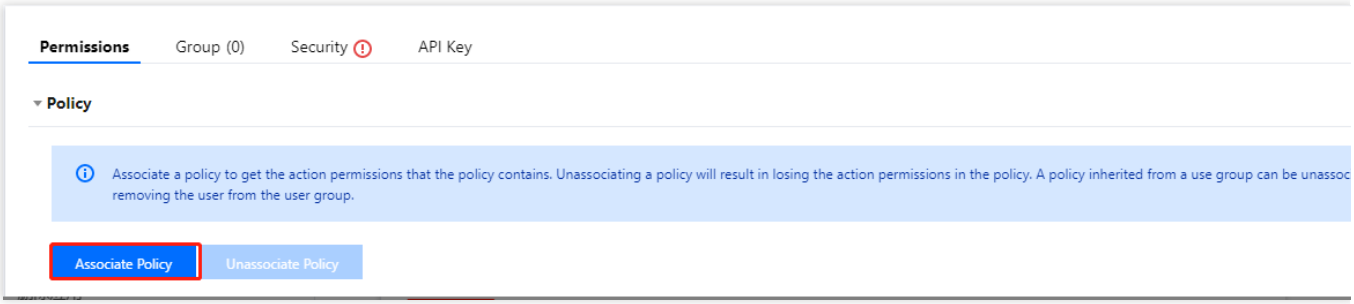


3. On the CAM User List page, select the target sub-account and click the username to go to the user details page.

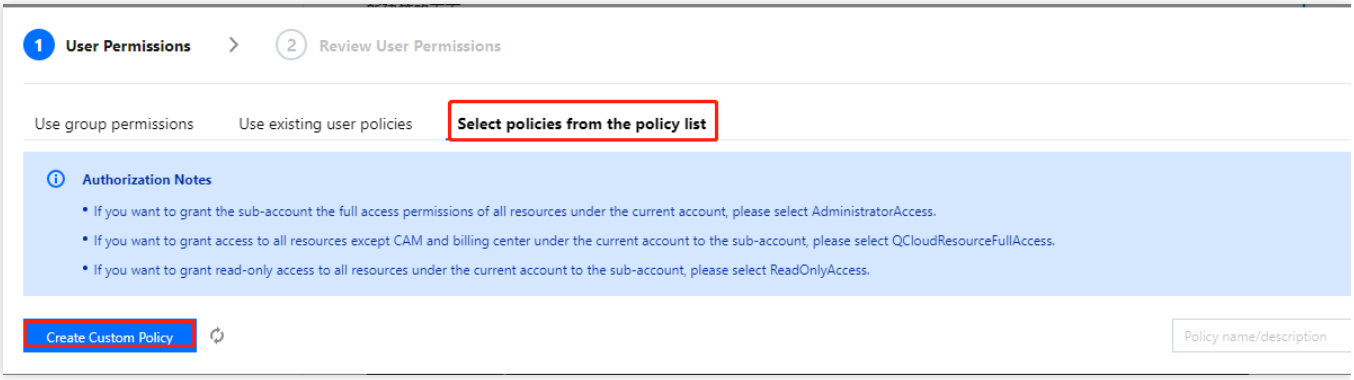


4. Click **Associate Policy**. On the policy adding page, click the **Select policies from the policy list** tab, and then click **Create Custom Policy**.

Policy association page:



Policy creation page:

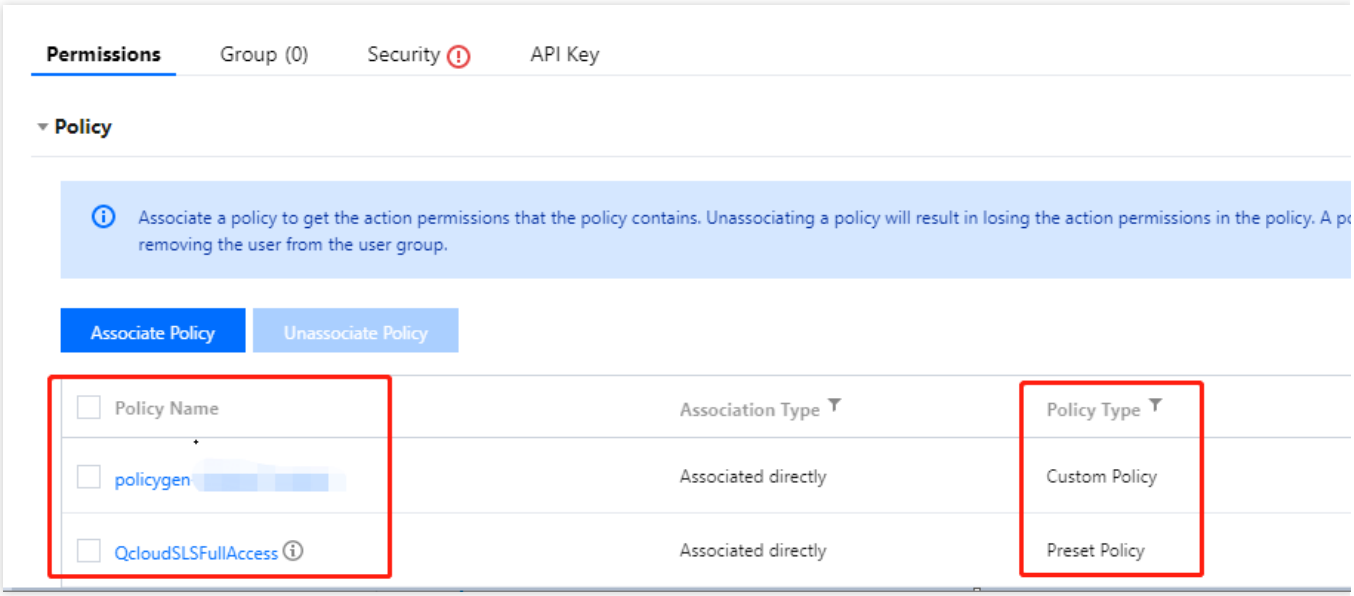


5. Choose **Create by Policy Syntax** > **Blank Template** and enter the following content. Make sure to replace the role parameter with the UIN of your root account:

```
{
 "version": "2.0",
 "statement": [
     {
         "action":[
             "cam:PassRole"
         ],
         "resource": [
             "qcs::cam::uin/${Enter the UIN of your account}:roleName/scf_QcsRole"
         ],
         "effect": "allow"
     },
     {
         "resource": [
             "*"
         ],
         "action":[
             "name/sts:AssumeRole"
         ],
         "effect": "allow"
```

```
        }
    ]
}
```

6. After completing the custom policy configuration, go back to the authorization page in step 4, search for the custom policy just created, click **Next**, and then click **OK** to grant the sub-account the operation permissions of `scf_QcsRole`. At this point, your sub-account should have a custom policy and a preset policy **QcloudscfFullAccess** and can use Serverless Cloud Framework normally.



Note

In addition to the permission to call the default `scf_QcsRole` role, you can also grant the sub-account the permission to call a custom role and control the sub-account permissions with refined permission policies in the custom role. For more information, see Configuring Role for Specified Operation.

scf_QcsRole role permission list

Policy	Description
QcloudCOSFullAccess	Full access to Tencent Cloud Object Storage (COS).
QcloudSCFFullAccess	Full access to Serverless Cloud Function (SCF).
QcloudSSLFullAccess	Full access to SSL Certificate Service.
QcloudTCBFullAccess	Full access to Tencent CloudBase (TCB).
QcloudAPIGWFullAccess	Full access to API Gateway.
QcloudVPCFullAccess	Full access to Virtual Private Cloud (VPC).
QcloudMonitorFullAccess	Full access to Cloud Monitor.

QcloudslsFullAccess	Full access to Serverless Cloud Framework (SLS).
QcloudCDNFullAccess	Full access to Content Delivery Network (CDN).
QcloudCKafkaFullAccess	Full access to CKafka.
QcloudCodingFullAccess	Full access to CODING DevOps.
QcloudPostgreSQLFullAccess	Full access to TencentDB for PostgreSQL.
QcloudCynosDBFullAccess	Full access to TencentDB for CynosDB.
QcloudCLSFullAccess	Full access to Tencent Cloud Log Service (CLS).
QcloudAccessForscfRole	This policy can be associated with the service role (scf_QCSRole) of Serverless Cloud Framework to access other Tencent Cloud service resources by using the quick experience feature of Serverless Cloud Framework. The scf_QCSRole role has the permissions to perform CAM-related operations.

Access Management Configuration

Last updated：2024-12-02 10:48:10

CAM Overview

Cloud Access Management (CAM) is a web-based Tencent Cloud service that helps you securely manage and control access permissions, resources, and use permissions of your Tencent Cloud account. Using CAM, you can create, manage, and terminate users (groups), and control the Tencent Cloud resources that can be used by the specified user through identity and policy management.
Tencent Cloud SLS supports **resource-level authorization**. You can use policy syntax to grant sub-accounts permissions to manage individual resources. For more information, please see Authorization Scheme Examples.

Authorizable Resource Types

SLS supports resource-level authorization. You can grant a specified sub-account the API permission of a specified resource. APIs supporting resource-level authorization include:

API Name	Description	Six-Segment Example of Resource
SaveInstance	Saves the instance information of component	`qcs::sls:${Region}:uin/:appname/${AppName}/stagename`
GetInstance	Gets the instance information of component	`qcs::sls:${Region}:uin/:appname/${AppName}/stagename`
ListInstances	Gets the instance list information of component	`qcs::sls:${Region}:uin/:appname/${AppName}/stagename`
RunComponent	Runs component	`qcs::sls:${Region}:uin/:appname/${AppName}/stagename`

	instance	
RunFinishComponent	Finishes running component instance	`qcs::sls:${Region}:uin/:appname/${AppName}/stagename`

Authorization Scheme Examples

Six-Segment resource description

Parameter	Required	Description
qcs	Yes	Tencent Cloud service abbreviation, which indicates a resource of Tencent Cloud.
project_id	Yes	Project information description, which is only used to enable compatibility with legacy logic.
service_type	Yes	Product abbreviation, which is `sls` for Serverless Framework.
region	Yes	Region information, such as `bj` . For more information, please see Region List.
account	No	Root account of resource owner, such as `uin/164256472` . If it is empty, it indica root account of the CAM user who creates the policy.
resource	Yes	Detailed resource information of each product, which is `qcs::sls:${Region}:uin/:appname/${AppName}/stagename/${Stage` for Serverless Framework

Sample

You can log in to the CAM console as a root account to configure and manage the permissions of Serverless Framework. Currently, Serverless Framework provides two preset policies for **full access permission** and **read-only access permission**:

Full access permission

Grant a sub-account full access to Serverless Framework (SLS).

Policy name: QcloudSLSFullAccess

```
{
  "version": "2.0",
  "statement": [
    {
      "action": [
```

```
          "sls:*"
        ],
        "resource": "*",
        "effect": "allow"
      }
    ]
}
```

Read-only access permission

Grant a sub-account read-only access to Serverless SSR (SLS).

Policy name: QcloudSLSReadOnlyAccess

```
{
  "version": "2.0",
  "statement": [
    {
      "action": [
        "sls:Get*",
        "sls:List*"
      ],
      "resource": "*",
      "effect": "allow"
    }
  ]
}
```

Sub-account Resource Management

The sub-account can access and manage the resources authorized to it by the root account.

If the sub-account has the permission to create resources and pay bills, it can purchase resources by itself in the normal process, and the root account will be charged.

.yml File Specification

Last updated：2024-12-02 10:48:10

Serverless Framework uses the project configuration file `serverless.yml` to identify the application type and configure the resources. After you develop a project locally, you must configure the `.yml` file first before you can deploy the project in the cloud by running the `sls deploy` command to pass the configuration information in `serverless.yml` and the specified parameters or code directory in `inputs` to the Serverless Components deployment engine.

Basic Information

The first-level field in a basic `serverless.yml` file is configured as follows:

```
# Organization information (optional)
app: '' # Application name. If it is left empty, the instance name of the current c
stage: '' # Environment name. The default value is `dev`. We recommend you use the

# Component information
component: scf # Component name, which is required. It is `scf` in this example
name: scfdemo # Component instance name, which is required

# Component parameter configuration, which configures specific resource information
inputs:
```

Detailed Configuration

In the `inputs` field, the corresponding information will be configured for the resources created by each component in the cloud. The SCF Component is taken as an example here. The second-level directory in the `input` field is as follows:

```
inputs:
  name: xxx # Function name, which is `${name}-${stage}-${app}` by default
  src: ./src # Project code path in the default format. Create a specifically named
  handler: index.main_handler # Entry
  runtime: Nodejs10.15 # Runtime environment, which is Nodejs10.15 by default
  region: ap-guangzhou # Function region
  description: This is a function in ${app} application.
  environment: # Environment variable
```

```
    variables: # Environment variable object
      TEST: value
  layers: # Layer configuration
    - name: scfLayer # Layer name
      version: 1 # Version
  events: # Trigger configuration
    - timer: # Scheduled trigger
        parameters:
          cronExpression: '*/5 * * * * * *' # Trigger once every 5 seconds
          enable: true
```

Full Configuration List

Below is the list of full configuration information for each component of Serverless Framework:

Basic components

Component	Full Configuration
SCF	SCF - serverless.yml configuration
Website	Website - serverless.yml configuration
API Gateway	API Gateway - serverless.yml configuration
VPC	VPC - serverless.yml configuration
COS	COS - serverless.yml configuration
PostgreSQL	PostgreSQL - serverless.yml configuration
CynosDB	CynosDB - serverless.yml configuration
CDN	CDN - serverless.yml configuration
Layer	Layer - serverless.yml configuration

Framework components

Component	Full Configuration
Express	Express - serverless.yml configuration
Koa	Koa - serverless.yml configuration

Egg	Egg - serverless.yml configuration
Next.js	Next.js - serverless.yml configuration
Nuxt.js	Nuxt.js - serverless.yml configuration
Flask	Flask - serverless.yml configuration
Django	Django - serverless.yml configuration
Laravel	Laravel - serverless.yml configuration
ThinkPHP	ThinkPHP - serverless.yml configuration

Project Structure

Last updated：2024-12-02 10:48:10

Serverless Cloud Framework deploys applications based on the Serverless components. There are no mandatory requirements for the local project structure, but for the ease of management and deployment, we recommend you organize your application in the following directory structure:

Single-Function Application

For single-function applications, you can place your business code in the `src` directory and import this directory in the `serverless.yml` configuration file to achieve separate management of the project and the configuration file. Below is an example:

```
.
├── serverless.yml  # Configuration file
├── src
│   ├── package.json # Dependency file
│   └── index.js # Entry function
└── .env # Environment variable file
```

Multi-Function/Multi-Resource Application

Serverless Cloud Framework not only supports deploying single-function projects, but also can implement unified deployment at the application level for multi-function projects. You should configure the corresponding configuration file for each function; therefore, we recommend the following directory structure:

```
.
├── package.json # Dependency file
├── function1
│   ├── serverless.yml # Configuration file of function 1
│   └── index1.js # Entry function 1
├── function2
│   ├── serverless.yml # Configuration file of function 2
│   └── index2.js # Entry function 2
└── .env # Environment variable file
```

Under this structure, you only need to run `scf deploy` in the root directory, and Serverless Cloud Framework will automatically traverse all the `.yml` configuration files in the directory to deploy resources.

Meanwhile, if you import the creation of other cloud resources in the function project, you can also use the same directory structure:

```
.
├── package.json # Dependency file
├── src
│   ├── serverless.yml # Function configuration file
│   └── index1.js # Entry function
├── cos
│   └── serverless.yml # COS bucket configuration file
├── db
│   └── serverless.yml # Database configuration file
└── .env # Environment variable file
```

Local Debugging

Last updated：2024-12-02 10:48:10

Overview

With the local debugging capabilities of Serverless Framework, you can run code in your local simulation environment, send simulated test events, and get information such as execution logs of the function code.

Prerequisites

The Node.js environment has been installed in your system.

Note:

Currently, the commands are supported only for Node.js and Python runtimes. In order to ensure that the results of cloud-based deployment and local execution are consistent, we recommend you install the same runtime version locally and in the cloud. For example, if you use Node.js 12.x in the cloud, we recommend you install Node.js 12.x locally.

Currently, only the SCF component supports local debugging.

Local debugging is supported only for event functions. For web functions, please test them as instructed in Cloud Test.

Directions

The code can be triggered locally by running the `sls invoke local` command. The Serverless Framework CLI will run the corresponding code in the specified local directory according to the specified function template configuration file and then implement the execution in the local SCF simulation environment through the specified trigger event.

The related commands are as follows:

```
invoke local .................. Invoke the local function
    --function / -f ............ Function name (you can only specify a function nam
    --data / -d ................ Serialized event data to be passed to the invoked
    --path / -p ................ Path to the event JSON file to be passed to the in
    --context .................. Serialized context data to be passed to the invoke
    --contextPath / -x ......... Path to the context JSON file to be passed to the
    --env / -e ................. Overwrite environment variable information, such a
    --config / -c ..............Path to serverless config file
```

Directions

The following uses Node.js as an example to describe how to perform local debugging:

1. Run the following command to initialize the sample code.

```
sls init scf-nodejs && cd scf-nodejs
```

2. Create the test event template `test.json` in the directory. Below is an example:

```
{
          "value": "test",
          "text": "Hello World event template",
          "context": {
              "key1": "test value 1",
              "key2": "test value 2"
          }
}
```

3. Create a `.env` file and enter your permanent key. Below is an example:

```
# .env
TENCENT_SECRET_ID=xxxxxxxxxx # `SecretId` of your account
TENCENT_SECRET_KEY=xxxxxxxx # `SecretKey` of your account
```

Note:

You can also scan the QR code to deploy and get a temporary key to automatically generate the configuration file.

4. Run the following command to view the invocation result locally.

```
sls invoke local -p xxxx.json
```

Below is a sample:

```
# sls invoke local -p test.json
Hello World
{
  value: 'test',
  text: 'Hello World event template',
  context: { key1: 'test value 1', key2: 'test value 2' }
}
undefined
{}
------------------------------------------
Serverless: invocation succeeded

{
```

```
    value: "test",
    text: "Hello World event template",
    context: {
      key1: "test value 1",
      key2: "test value 2"
    }
}
```

Building Application

Last updated：2024-12-02 10:48:10

Overview

After Serverless Framework is installed, you can initialize a project template and build a multiple-component application as instructed in this document.

Prerequisites

You have installed Serverless Framework.

Directions

Initializing project template

You can quickly initialize a demo project by running the following command and modify it for further development:

```
sls init scf-demo
```

This command can quickly build a basic function application locally with the following directory structure:

```
.
├── serverless.yml  # Configuration file
└── src
    └── index.js # Entry function
```

You can enter this directory and develop your project based on the demo template.

Note:

`sls init` can quickly initialize multiple project templates. You can run the `sls registry` command to view all supported project templates.

Building multiple-component application

Serverless Framework provides multiple basic resource components, which you can mix and use to quickly create and deploy resources in the cloud, thus eliminating the need for manual operations in the console (for more information, please see Basic Component List and Configuration Method).

This document uses deploying a function project triggered by a COS trigger as an example to describe how to import multiple components into your project and quickly complete the deployment. The steps are as follows:

1. Adjust the structure of the project directory, create a COS folder, and write the configuration file `serverless.yml` for the COS component in this directory. The adjusted structure of the directory is as follows:

```
.
├── src
│   ├── serverless.yml # Function configuration file
│   └── index.js # Entry function
├── cos
│   └── serverless.yml # COS bucket configuration file
└── .env # Environment variable file
```

A sample `.yml` file for the COS component is provided below. For more information on all configuration items, please see COS Component Configuration.

```
app: appDemo
stage: dev

component: cos
name: cosdemo

inputs:
  bucket: my-bucket
  region: ap-guangzhou
```

2. Modify the `.yml` configuration file for the SCF project and impot the deployment result of the COS component according to the following syntax in the trigger configuration part:

```
app: appDemo
stage: dev

component: scf
name: scfdemo
inputs:
  ...
  events:
   - cos: # COS trigger
        parameters:
          bucket: ${output:${stage}:${app}:cosdemo.bucket}
```

Note:

When deploying multiple component instances in the same project, you need to make sure that the `app` and `stage` parameters of each project are the same; otherwise, they cannot be successfully imported.

3. In the project root directory, run `sls deploy` to complete COS bucket creation and use the output of the COS component as the input of the SCF component to configure the trigger.

Variable import description

`serverless.yml` supports multiple ways to import variables:

Import basic Serverless parameters

In the `inputs` field, you can directly import basic Serverless configuration information through the `${org}` and `${app}` syntax.

Import environment variables

In `serverless.yml`, you can directly import the environment variable configuration (including the environment variable configuration in the `.env` file and variable parameters manually configured in the environment) through the `${env}` syntax.

For example, you can import the environment variable `REGION` through `${env:REGION}`.

Import the output results of other components

If you want to import the output information of other component instances into the current component configuration file, you can configure it by using the following syntax: `${output:[app]:[stage]:[instance name].[output]}`

Sample `.yml` file:

```
app: demo
component: scf
name: rest-api
stage: dev

inputs:
  name: ${stage}-${app}-${name} # The final name is "acme-prod-ecommerce-rest-api"
  region: ${env:REGION} # `REGION=` information specified in the environment variab
  vpcName: ${output:prod:my-app:vpc.name} # Get the output information of other com
  vpcName: ${output:${stage}:${app}:vpc.name} # The above methods can also be used
```

In-cloud Debugging

Last updated：2024-12-02 10:48:10

Development Mode

The development mode enables you to write code for and develop and debug projects in development status more easily, so that you can continuously focus on the process from development to debugging while minimizing the interruptions caused by other tasks such as packaging and update.

Entering development mode

Under a project, you can run `serverless dev` to enter the development mode:
Below is an example:

```
$ sls dev
serverless ⚡ framework
Dev Mode - Watching your Component for changes and enabling streaming logs, if supp

Debugging listening on ws://127.0.0.1:9222.
For help see https://nodejs.org/en/docs/inspector.
Please open chorme, and visit chrome://inspect, click [Open dedicated DevTools for
---------------------- The realtime log ----------------------
17:13:38 - express-api-demo - deployment
region: ap-guangzhou
apigw:
  serviceId:   service-b77xtibo
  subDomain:   service-b77xtibo-1253970226.gz.apigw.tencentcs.com
  environment: release
  url:         http://service-b77xtibo-1253970226.gz.apigw.tencentcs.com/release/
scf:
  functionName: express_component_6r6xkh60k
  runtime:      Nodejs10.15
  namespace:    default

express-api-demo › Watching
```

After you enter the development mode, the Serverless tool will output the deployed content and start continuous file monitoring. When a code file is modified, it will be automatically deployed again to sync the local file to the cloud.
Deploy again and output the deployment information:

```
express-api-demo › Deploying ...
Debugging listening on ws://127.0.0.1:9222.
For help see https://nodejs.org/en/docs/inspector.
```

```
Please open chorme, and visit chrome://inspect, click [Open dedicated DevTools for
---------------------- The realtime log ----------------------
21:11:31 - express-api-demo - deployment
region: ap-guangzhou
apigw:
  serviceId:   service-b7dlqkyy
  subDomain:   service-b7dlqkyy-1253970226.gz.apigw.tencentcs.com
  environment: release
  url:         http://service-b7dlqkyy-1253970226.gz.apigw.tencentcs.com/release/
scf:
  functionName: express_component_uo5v2vp
  runtime:      Nodejs10.15
  namespace:    default
```

Note:

Currently, `serverless dev` supports only by the Node.js 10 runtime environment. It will support real-time logging in more environments such as Python and PHP.

Exiting development mode

You can press Ctrl+C to exit the development mode (`dev` mode).

```
express-api-demo › Disabling Dev Mode & Closing ...

express-api-demo › Dev Mode Closed
```

In-cloud Debugging: Node.js 10+

For projects whose runtime environment is Node.js 10+, you can connect them to in-cloud debugging by enabling in-cloud debugging and using a debugging tool such as Chrome DevTools or VS Code Debugger.

Enabling in-cloud debugging

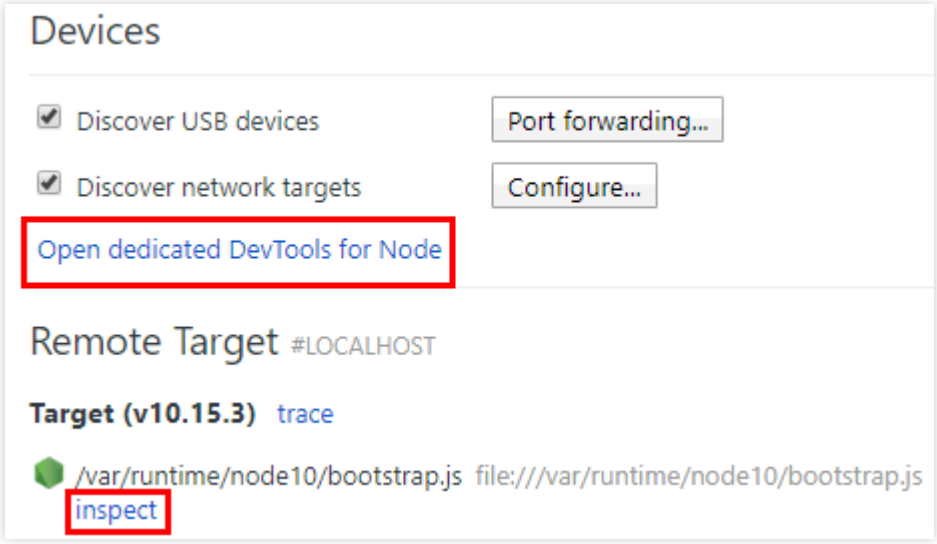
When you enter the development mode as instructed above, if the project is a function whose runtime environment is Node.js 10 or above, in-cloud debugging will be automatically enabled and debugging information will be output.
For example, when you enable the development mode, if the following information is output, in-cloud debugging has been enabled for this function.

```
Debugging listening on ws://127.0.0.1:9222.
For help see https://nodejs.org/en/docs/inspector.
Please open chorme, and visit chrome://inspect, click [Open dedicated DevTools for
```

Using Chrome DevTools

The following steps are used as an example to describe how to use DevTools in Chrome to connect to a remote environment for debugging:

1. Start the Chrome browser.
2. Enter `chrome://inspect/` in the address bar to access it.
3. You can open DevTools in two ways as shown below:



4. (Recommended) Click **Open dedicated DevTools for Node** under "Devices".
5. Select **inspect** under a specific target in "Remote Target #LOCALHOST".

If you cannot open the target or there are no targets, please check whether configuration of `localhost:9229` or `localhost:9222` exists in "Configure" under "Devices", which corresponds to the output after in-cloud debugging is enabled.

6. In DevTools opened after you click **Open dedicated DevTools for Node**, you can click the **Sources** tab to view the remote code. The actual code of the function is in the `/var/user/` directory.

On the **Sources** tab, the code that you want to view may be loaded. More remote files will be displayed as the debugging proceeds.

7. Open a file as needed and set a breakpoint at the specified position in it.
8. If you trigger the function in any means such as URL access, page, command, or API, the remote environment will start running and be interrupted at the breakpoint to wait for further operations.
9. On the tool bar on the right of DevTools, you can continue the execution of an interrupted program or perform other operations such as step-over, step-into, and step-out on it. You can also directly view the current variables or set the variables that you want to track. For more information on how to use DevTools, please see the DevTools user guide.

Exiting in-cloud debugging

When you exit the development mode, in-cloud debugging will be disabled automatically.

Command Debugging

The Serverless Framework SCF component supports triggering functions with the `invoke` command for debugging. For a function successfully deployed by running `sls deploy`, enter the project directory and run the following command to invoke it:

```
sls invoke --inputs function=functionName clientContext='{"weights":{"2":0.1}}'
```

Note:

The `invoke` command must be executed in the same directory as the `serverless.yml` file deployed for the function.

`clientContext` is the JSON string passed when the function is triggered. You can simulate different triggering events according to the JSON string format in the triggering event template.

Deploying Application

Last updated：2024-12-02 10:48:09

Overview

After developing your project locally, you can quickly deploy the application, view deployment information, and perform function debugging.

Prerequisites

You have developed your project locally (for more information, please see Project Development).

Directions

Quick deployment

Serverless Framework enables you to quickly deploy your project in the cloud by following the steps below:

```
sls deploy
```

After you enter this command, Serverless Framework CLI will perform the following operations:

1. Scan QR code to authorize

You can authorize by scanning the QR code. After that, the CLI tool will write the generated temporary key information into the `.env` file in the current directory. The temporary key is valid for 2 hours. After it expires, you will be asked to scan the QR code again to authorize for deployment purposes.
If you don't want to scan the QR code repeatedly, you can also configure a permanent key in the `.env` file in the project directory:

```
# .env
TENCENT_SECRET_ID=xxxxxxxxxx # `SecretId` of your account
TENCENT_SECRET_KEY=xxxxxxxx # `SecretKey` of your account
```

You can get `SecretId` and `SecretKey` in API Key Management.

2. Package and upload

After authorization is completed, Serverless Framework CLI will automatically package and upload your project according to the project code path configured in the `serverless.yml` file.

3. Deploy in the cloud

Resources will be created in the cloud for the uploaded project according to the parameters configured in the `.yml` file. After the deployment is completed, the command line will output the resource information.

Advanced capabilities

View specific log information during deployment:

```
sls deploy --debug
```

Switch the specified traffic to the `$latest` function version during multi-version deployment and the rest traffic to the last published function version for grayscale release:

```
sls deploy --inputs traffic=0.1 public=true
```

There are multiple Serverless instances in the application directory, and you want to update the specified project only:

```
sls deploy --target xxx
```

For example, in the project root directory, you can run the `sls deploy --target ./cos` command to update the COS instance only without affecting other instances.

```
.
├── src
│   ├── serverless.yml
│   └── index1.js
├── cos
│   └── serverless.yml
├── db
│   └── serverless.yml
└── .env
```

Viewing deployment information

After completing the deployment, you can run the following command to view the configuration information of the project:

```
sls info
```

Debugging function

Note:

Currently, this command is supported only for function projects deployed through the Serverless Framework SCF component. It will be gradually supported for other components in the future.

The Serverless Framework SCF component supports triggering functions with the `invoke` command for debugging. For a function successfully deployed by running `sls deploy`, enter the project directory and run the function invocation command to remotely debug the function resources in the cloud. The debugging result will be output on the command line:

```
sls invoke  --inputs function=functionName  clientContext='{"weights":{"2":0.1}}'
```

The `invoke` command must be executed in the same directory as the `serverless.yml` file deployed for the function.

`clientContext` is the JSON string passed when the function is triggered. You can simulate different triggering events according to the JSON string format in the triggering event template.

FAQs

If a proxy is configured in your environment, the following problems may occur:

Problem 1: the wizard does not pop up by default when `serverless` is entered.

Solution: make sure that your IP is in the Chinese mainland and add the `SERVERLESS_PLATFORM_VENDOR=tencent` configuration item to the `.env` file.

Problem 2: after `sls deploy` is entered, the deployment reports a network error.

Solution: add the following proxy configuration to the `.env` file.

```
HTTP_PROXY=http://127.0.0.1:12345 # Replace "12345" with your proxy port
HTTPS_PROXY=http://127.0.0.1:12345 # Replace "12345" with your proxy port
```

Deleting Application

Last updated：2024-12-02 10:48:09

Basic Features

You can quickly delete cloud resources by running the following command:

```
sls remove
```

Advanced Features

View specific log information during deletion:

```
sls remove --debug
```

There are multiple Serverless instances in the application directory, and you want to delete the specified project only:

```
sls remove --target xxx
```

For example, in the project root directory, you can run the `sls remove --target ./cos` command to delete the COS instance only without affecting other instances.

```
.
├── src
│   ├── serverless.yml
│   └── index1.js
├── cos
│   └── serverless.yml
├── db
│   └── serverless.yml
└── .env
```

FAQs

What cloud resources will be removed when `sls remove` is executed?

Serverless Framework removes cloud resources according to the `.yml` configuration file. Resources created through the `.yml` file will be deleted, while imported existing resources will not. For example:

When deploying a function through the SCF component, if you choose to create an API Gateway trigger for deployment, then when you run the command for deletion, the created function and API Gateway resources will be deleted.
If you select an existing API Gateway trigger for deployment, then only the function resources will be deleted, while the used API Gateway trigger will not.

List of Supported Commands

Last updated：2024-12-02 10:48:09

Serverless Application Center (SLS) is deployed based on Serverless Framework and supports the following CLI commands:

`serverless registry` : Lists available components.

`serverless registry publish` : Publishes components to the SLS component registry.

`--dev` : Publishes components of the `@dev` version for development or testing.

`serverless init xxx` : Downloads a specified template from the component registry by entering the name of the template to download after `init` , such as "$ serverless init fullstack".

`sls init xxx --name my-app` : Customizes the project directory name.

`--debug` : Lists log information during template download.

`serverless deploy` : Deploys a component instance in the cloud.

`--debug` : Lists log information such as the deployment operations and the status output by `console.log()` during component deployment.

`---inputs publish=true` : Publishes a new version during function deployment.

`---inputs traffic=0.1` : Switches 10% of the traffic to the `$latest` function version during deployment and switches the rest of the traffic to the last published function version.

Note:

The legacy command format `sls deploy --inputs.key=value` has been changed to `sls deploy --inputs key=value` since Serverless CLI v3.2.3. Legacy commands cannot be used in new versions of Serverless CLI. If you have upgraded Serverless CLI, please use the new commands.

`serverless remove` : Removes a component instance from the cloud.

`--debug` : Lists log information such as the removal operations and the status output by `console.log()` during component removal.

`serverless info` : Gets and displays the information about a component instance.

`--debug` : Lists more `state` values.

`serverless dev` : Enables the development mode ("DEV Mode") and automatically deploys changed information when component status changes are detected. In development mode, information such as execution logs, invocation information, and errors can be displayed on the CLI in real time. The development mode also supports in-cloud debugging for Node.js applications.

Multi-Function Application Deployment

Last updated：2024-12-02 10:48:10

You can quickly build and deploy a multi-function application based on the Tencent Cloud multi-scf component, which greatly reduces the development costs of complex applications.

Prerequisites

Serverless Framework has been installed. For more information, please see Installing Serverless Framework.
Your account has the Serverless Framework permissions. For more information, please see Account and Permission Configuration.

Development and deployment steps

For details of sample projects, please see Case List.
1. Develop your application project locally. This document takes a project with two functions as an example. The application directory structure is as follows:

```
./multi-scf-demo
├── index
│   ├── index.js # Main function 1
│   ├── package.json
│   └── scf_bootstrap # Bootstrap file for HTTP-triggered functions, which can be i
├── user
│   ├── index.js # Main function 2
│   ├── package.json
│   └── scf_bootstrap # Bootstrap file for HTTP-triggered functions, which can be i
└── serverless.yml # YML configuration file
```

2. In the **root directory**, create a `serverless.yml` file and configure relevant parameters for your project by referring to the following sample YML. For more configuration content, please see Full Configuration.

```
app: multi-scf # Application name
component: multi-scf # Component type, which is `multi-scf` here
name: web_demo # Customizable instance name

inputs:
  src:
 # The code directory must be specified here, and SCF will automatically split the
 src: ./
 exclude:
   - .env
  region: ap-guangzhou # Region
  runtime: Nodejs12.16 # Function language version
```

```
 memorySize: 512
 timeout: 3
 type: web  # Function type, which is HTTP-triggered function here
 functions:
index:
  src: ./index # Entry function of function 1
  handler: scf_bootstrap # Bootstrap file
user:
  src: ./user # Entry function of function 2
  handler: scf_bootstrap # Bootstrap file
 triggers: # Trigger configuration
- type: apigw
  parameters:
    name: serverless
    protocols:
      - https
      - http
    apis:
      - path: /
        method: ANY
        # The API function configuration has a higher priority than the outer func
        function: index
      - path: /user
        method: ANY
        # The API function configuration has a higher priority than the outer func
        function: user
```

3. After completing the configuration, run `sls deploy` in the root directory to test whether the project is successfully deployed.

Application launch in console

Submit the application through a **ticket**. Note that your project must include the following:

Parameter	Description
Basic configuration parameter list	Basic configuration parameter list
Advanced configuration parameter list	Optional
Application name, overview, documentation link, and tag	For block display in the console

Basic Component List

Last updated：2024-12-02 10:48:09

The use instructions and full configuration documents of the basic components of Serverless Framework are as follows:

Component	Use Instructions	Full Configuration
SCF component	Use Instructions	serverless.yml Configuration
Website component	Use Instructions	serverless.yml Configuration
API Gateway component	Use Instructions	serverless.yml Configuration
VPC component	Use Instructions	serverless.yml Configuration
COS component	Use Instructions	serverless.yml Configuration
PostgreSQL component	Use Instructions	serverless.yml Configuration
CynosDB component	Use Instructions	serverless.yml Configuration
CDN component	Use Instructions	serverless.yml Configuration
Layer component	Use Instructions	serverless.yml Configuration
CynosDB component	Use Instructions	serverless.yml Configuration

Connecting to MySQL Database

Last updated：2024-12-02 10:48:10

Overview

Currently, TDSQL-C for MySQL supports serverless billing. In this billing mode, the service is billed based on the actual computing and storage usage which is calculated by second and settled by hour. The TDSQL-C component of Serverless Framework also supports creating this type of databases.
This document uses a function written in Node.js as an example to describe how to quickly create a TDSQL-C for MySQL serverless instance and call it in SCF.

Directions

Step	Description
Step 1. Configure environment variables	-
Step 2. Configure a VPC	Use the Serverless Framework VPC component to create a VPC and subnet for communications between the function and the database.
Step 3. Configure Serverless DB	Use the Serverless Framework TDSQL-C component to create a MySQL instance to provide database services for the function project.
Step 4. Write business code	Use the Serverless DB SDK to call the database. SCF allows you to directly call the Serverless DB SDK to connect to and manage a PostgreSQL database.
Step 5. Deploy an application	Use Serverless Framework to deploy the project in the cloud and test it in the SCF console.
Step 6. Remove the project (optional)	You can use Serverless Framework to remove the project.

Step 1. Configure environment variables

1. Create a local directory to store code and dependent modules. This document uses the `test-MySQL` folder as an example.

```
mkdir test-MySQL && cd test-MySQL
```

2. Currently, TDSQL-C Serverless only supports four regions: `ap-beijing-3`, `ap-guangzhou-4`, `ap-shanghai-2`, and `ap-nanjing-1`, so you need to create the `.env` file in the project root directory and then configure the two environment variables `REGION` and `ZONE`:

```
# .env
REGION=xxx
ZONE=xxx
```

Step 2. Configure a VPC

1. Create a `VPC` folder in the `test-MySQL` directory.

```
mkdir VPC && cd VPC
```

2. Create a `serverless.yml` file in `VPC` and use the VPC component to create the VPC and subnet.

The sample content of `serverless.yml` is as follows (for all configuration items, please see the product documentation):

```
#serverless.yml
app: mysql-app
stage: dev
component: vpc # (required) name of the component. In that case, it's vpc.
name: mysql-app-vpc # (required) name of your vpc component instance.
inputs:
    region: ${env:REGION}
    zone: ${env:ZONE}
    vpcName: serverless-mysql
    subnetName: serverless-mysql
```

Step 3. Configure Serverless DB

1. Create a `DB` folder in `test-MySQL`.

2. Create a `serverless.yml` file in the `DB` folder and enter the following content to use the Serverless Framework component to configure the TCB environment:

The sample content of `serverless.yml` is as follows (for all configuration items, please see the product documentation):

```
# serverless.yml
app: mysql-app
stage: dev
component: cynosdb
name: mysql-app-db
inputs:
```

```
  region: ${env:REGION}
  zone: ${env:ZONE}
  vpcConfig:
    vpcId: ${output:${stage}:${app}:mysql-app-vpc.vpcId}
    subnetId: ${output:${stage}:${app}:mysql-app-vpc.subnetId}
```

Step 4. Write the business code and configuration file

1. Create an `src` folder in `test-MySQL` to store the business logic code and relevant dependencies.
2. Create an `index.js` file in the `src` folder and enter the following sample code, so that you can use the SDK to connect to the MySQL database through the function and call the database in the environment:

```
exports.main_handler = async (event, context, callback) => {
    var mysql      = require('mysql2');
    var connection = mysql.createConnection({
    host     : process.env.HOST,
    user     : 'root',
    password : process.env.PASSWORD
    });
    connection.connect();
    connection.query('SELECT 1 + 1 AS solution', function (error, results, fields)
    if (error) throw error;
    console.log('The solution is: ', results[0].solution);
    });
    connection.end();
 }
```

3. Install the required dependent modules.

```
npm install mysql2
```

4. After writing the business code and installing the dependencies, create a `serverless.yml` file as shown below:

```
app: mysql-app
stage: dev
component: scf
name: mysql-app-scf

inputs:
  src: ./
  functionName: ${name}
  region: ${env:REGION}
  runtime: Nodejs10.15
```

```
  timeout: 30
  vpcConfig:
    vpcId: ${output:${stage}:${app}:mysql-app-vpc.vpcId}
    subnetId: ${output:${stage}:${app}:mysql-app-vpc.subnetId}
  environment:
    variables:
      HOST: ${output:${stage}:${app}:mysql-app-db.connection.ip}
      PASSWORD: ${output:${stage}:${app}:mysql-app-db.adminPassword}
```

Step 5. Deploy

After the creation, the project directory structure is as follows:

```
./test-MySQL
├── vpc
│   └── serverless.yml # VPC configuration file
├── db
│   └── serverless.yml # Database configuration file
├── src
│   ├── serverless.yml # SCF component configuration file
│   ├── node_modules # Project dependency file
│   └── index.js # Entry function
└── .env # Environment variable file
```

1. Run the following command for deployment in `test-MySQL` on the command line:

```
sls deploy
```

Note:

During deployment, you need to scan the QR code to authorize. If you don't have a Tencent Cloud account yet, please sign up first.

If your account is a sub-account, please get the authorization first as instructed in Account and Permission Configuration.

If the following result is returned, the deployment is successful:

```
mysql-app-vpc:
  region:        xxx
  zone:          xxx
  vpcId:         xxxx-xxx
  ...

mysql-app-db:
  dbMode:        xxxx
  region:        xxxx
  zone:          xxxx
```

```
  ...

mysql-app-scf:
  functionName:  xxxx
  description:   xxx
  ...

59s › test-MySQL › "deploy" ran for 3 apps successfully.
```

2. After the deployment succeeds, you can view and debug the function in the SCF console.

Step 6. Remove the project (optional)

Run the following command in the `test-MySQL` directory to remove the project:

```
sls remove
```

If the following result is returned, the removal is successful:

```
serverless ⚡ framework
4s › test-MySQL › Success
```

Sample Code

Python

In Python, you can use the built-in **pymysql** dependency package in the SCF environment to connect to the database. The sample code is as follows:

```
# -*- coding: utf8 -*-
from os import getenv

import pymysql
from pymysql.err import OperationalError

mysql_conn = None

def __get_cursor():
    try:
        return mysql_conn.cursor()
    except OperationalError:
        mysql_conn.ping(reconnect=True)
        return mysql_conn.cursor()

def main_handler(event, context):
```

```
    global mysql_conn
    if not mysql_conn:
        mysql_conn = pymysql.connect(
        host        = getenv('DB_HOST', '<YOUR DB HOST>'),
        user        = getenv('DB_USER','<YOUR DB USER>'),
        password    = getenv('DB_PASSWORD','<YOUR DB PASSWORD>'),
        db          = getenv('DB_DATABASE','<YOUR DB DATABASE>'),
        port        = int(getenv('DB_PORT','<YOUR DB PORT>')),
        charset     = 'utf8mb4',
        autocommit  = True
        )

    with __get_cursor() as cursor:
        cursor.execute('select * from employee')
        myresult = cursor.fetchall()
        print(myresult)
        for x in myresult:
            print(x)
```

Node.js

Node.js allows you to use a connection pool for connection, which supports automatic reconnection to effectively avoid connection unavailability due to connection release by the SCF underlying layer or database. The sample code is as follows:

Note:

Before using a connection pool, you need to install the **mysql2** dependency package first. For more information, please see Dependency Installation.

```
'use strict';

const DB_HOST       = process.env[`DB_HOST`]
const DB_PORT       = process.env[`DB_PORT`]
const DB_DATABASE   = process.env[`DB_DATABASE`]
const DB_USER       = process.env[`DB_USER`]
const DB_PASSWORD   = process.env[`DB_PASSWORD`]

const promisePool = require('mysql2').createPool({
  host              : DB_HOST,
  user              : DB_USER,
  port              : DB_PORT,
  password          : DB_PASSWORD,
  database          : DB_DATABASE,
  connectionLimit   : 1
}).promise();
```

```
exports.main_handler = async (event, context, callback) => {
  let result = await promisePool.query('select * from employee');
  console.log(result);
}
```

PHP

In PHP, you can use the **pdo_mysql** or **mysqli** dependency package for data connection. The sample code is as follows:

pdo_mysql

```
<?php
function handler($event, $context) {
try{
$pdo = new PDO('mysql:host= getenv("DB_HOST");dbname= getenv("DB_DATABASE"),getenv(
$pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
}catch(PDOException $e){
echo 'Databases connection failed: '.$e->getMessage();
exit;
}
}
```

mysqli

```
<?php
function main_handler($event, $context) {
   $host = "";
  $username = "";
  $password = "";

  // Create a connection
  $conn = mysqli_connect($servername, $username, $password);

  // Test the connection
  if (!$conn) {
      die("Connection failed: " . mysqli_connect_error());
      }
  echo "Connected successfully";

  mysqli_close($conn);
  echo "Disconnected";
}
?>
```

Java

1. Please install the following dependencies as instructed in Dependency Installation.

```
<dependencies>
    <dependency>
        <groupId>com.tencentcloudapi</groupId>
        <artifactId>scf-java-events</artifactId>
        <version>0.0.2</version>
    </dependency>
    <dependency>
        <groupId>com.zaxxer</groupId>
        <artifactId>HikariCP</artifactId>
        <version>3.2.0</version>
    </dependency>
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>8.0.11</version>
    </dependency>
</dependencies>
```

2. Use HikariCP for connection. The sample code is as follows:

```
package example;

import com.qcloud.scf.runtime.Context;
import com.qcloud.services.scf.runtime.events.APIGatewayProxyRequestEvent;
import com.qcloud.services.scf.runtime.events.APIGatewayProxyResponseEvent;
import com.zaxxer.hikari.HikariConfig;
import com.zaxxer.hikari.HikariDataSource;

import javax.sql.DataSource;
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.HashMap;
import java.util.Map;

public class Http {
    private DataSource dataSource;

    public Http() {
        HikariConfig config = new HikariConfig();
        config.setJdbcUrl("jdbc:mysql://" + System.getenv("DB_HOST") + ":"+ System.
```

```
        config.setUsername(System.getenv("DB_USER"));
        config.setPassword(System.getenv("DB_PASSWORD"));
        config.setDriverClassName("com.mysql.jdbc.Driver");
        config.setMaximumPoolSize(1);
        dataSource = new HikariDataSource(config);
    }

    public String mainHandler(APIGatewayProxyRequestEvent requestEvent, Context con
        System.out.println("start main handler");
        System.out.println("requestEvent: " + requestEvent);
        System.out.println("context: " + context);

        try (Connection conn = dataSource.getConnection(); PreparedStatement ps = c
            ResultSet rs = ps.executeQuery();
            while (rs.next()) {
                System.out.println(rs.getInt("id"));
                System.out.println(rs.getString("first_name"));
                System.out.println(rs.getString("last_name"));
                System.out.println(rs.getString("address"));
                System.out.println(rs.getString("city"));
            }
        } catch (SQLException e) {
            e.printStackTrace();
        }

        APIGatewayProxyResponseEvent apiGatewayProxyResponseEvent = new APIGatewayP
        apiGatewayProxyResponseEvent.setBody("API GW Test Success");
        apiGatewayProxyResponseEvent.setIsBase64Encoded(false);
        apiGatewayProxyResponseEvent.setStatusCode(200);

        Map<String, String> headers = new HashMap<>();
        headers.put("Content-Type", "text");
        headers.put("Access-Control-Allow-Origin", "*");
        apiGatewayProxyResponseEvent.setHeaders(headers);

        return apiGatewayProxyResponseEvent.toString();
    }
}
```

SCF DB SDK for MySQL

For ease of use, the SCF team encapsulated the code related to connection pools in Node.js and Python as SCF DB SDK for MySQL. Please refer to Dependency Installation for installation and use. With this SDK, you can connect to MySQL, TDSQL-C, or TDSQL for MySQL databases and performs operations such as insertion and query.
SCF DB SDK for MySQL has the following features:

It can automatically initialize the database client from environment variables.
It can maintain a persistent database connection globally and handle reconnection after disconnection.
The SCF team will continuously check issues to ensure that the database connection is available, so you don't need to pay attention to connection issues.

1. SDK for Node.js

```
'use strict';
const database = require('scf-nodejs-serverlessdb-sdk').database;

exports.main_handler = async (event, context, callback) => {
  let pool = await database('TESTDB2').pool()
  pool.query('select * from coffee',(err,results)=>{
    console.log('db2 callback query result:',results)
  })
  // no need to release pool

  console.log('db2 query result:',result)
}
```

Note:

For specific usage of the SDK for Node.js, please see SCF DB SDK for MySQL.

2. SDK for Python

```
from serverless_db_sdk import database

def main_handler(event, context):
    print('Start Serverlsess DB SDK function')

    connection = database().connection(autocommit=False)
    cursor = connection.cursor()

    cursor.execute('SELECT * FROM name')
    myresult = cursor.fetchall()

    for x in myresult:
        print(x)
```

Quickly Deploying Web Function

Last updated：2024-12-02 10:56:39

Overview

Web function is a new function capability in SCF. Compared with event function that has limits on the event format, web function focuses on optimization of web service scenarios and can directly send HTTP requests to URLs to trigger function execution. For more information, please see Function Overview.
The Serverless Framework SCF component now supports deploying web functions. therefore, you can use it to quickly create and deploy web functions.

Directions

1. Run the following command to initialize the serverless web function template.

```
sls init http-demo
```

2. Enter the demo project and view the directory structure as shown below:

```
. http-demo
├── serverless.yml  # Configuration file
├── package.json # Dependency file
├── scf_bootstrap # Project bootstrap file
└── index.js # Service function
```

Here, `scf_bootstrap` is the project bootstrap file. For the specific writing rules, please see Bootstrap File Description.
3. Open `serverless.yml` to view the configuration information.
You only need to add the `type` parameter in `yml` to specify the function type and deploy the web function.
Note:
For web functions, there is no need to specify the entry function.
If the `type` parameter is not entered, the function will be an event function by default.
If there is no `scf_bootstrap` bootstrap file in the local code, you can specify the `entryFile` parameter in `yml` to specify the entry function, and the component will generate a default `scf_bootstrap` file for you to complete the deployment based on the runtime language. After the deployment is completed, you need to modify the content of the `scf_bootstrap` file in the SCF console according to the actual needs of your project.
Below is a sample `yml` file:

```
component: scf
name: http
inputs:
  src:
    src: ./
    exclude:
      - .env
  # Specify web type as the function type
  type: web
  name: web-function
  region: ap-guangzhou
  runtime: Nodejs12.16
  # For Node.js, you can enable automatic dependency installation
  installDependency: true
  events:
    - apigw:
        parameters:
          protocols:
            - http
            - https
          environment: release
          endpoints:
            - path: /
              method: ANY
```

4. In the root directory, run `sls deploy` to complete the service deployment. Below is a sample:

```
$ sls deploy
serverless ⚡components
Action: "deploy" - Stage: "dev" - App: "http" - Name: "http"
type:         web
functionName: web-function
description:  This is a function in http application
namespace:    default
runtime:      Nodejs12.16
handler:
memorySize:   128
lastVersion:  $LATEST
traffic:      1
triggers:
 -
   NeedCreate:  true
   created:     true
   serviceId:   service-xxxxxx
   serviceName: serverless
```

```
    subDomain:   service-xxxxxx.cd.apigw.tencentcs.com
    protocols:   http&https
    environment: release
    apiList:
      -
        path:            /
        method:          ANY
        apiName:         index
        created:         true
        authType:        NONE
        businessType:    NORMAL
        isBase64Encoded: false
        apiId:           api-xxxxxx
        internalDomain:
        url:             https://service-xxxx.cd.apigw.tencentcs.com/release/
18s › http › Success
```

Relevant Commands

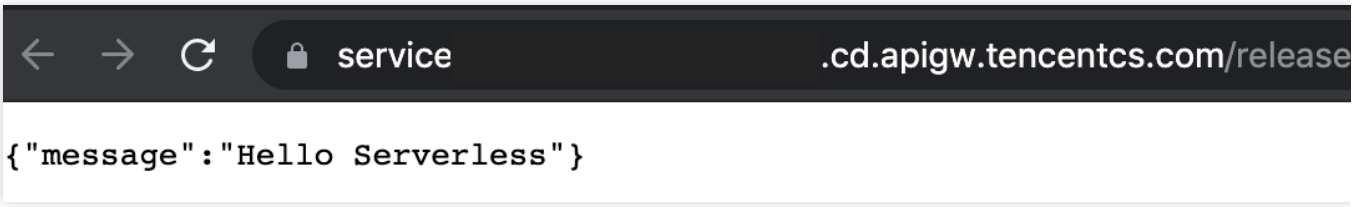
Viewing access log

Similar to event function, you can directly run the `sls log` command to view the latest 10 logs of the deployed function. Below is a sample:

```
$ sls log
serverless ⚡components
Action: "log" - Stage: "dev" - App: "http" - Name: "http"
-
  requestId:   xxxxx
  retryNum:    0
  startTime:   1624262955432
  memoryUsage: 0.00
  duration:    0
  message:
    """
    """
-
  requestId: xxxxx
  retryNum:    0
  startTime:   1624262955432
  memoryUsage: 0.00
  duration:    0
  message:
    """
    """
```

Testing service

Scheme 1: directly open the output path URL in a browser, and if it can be accessed normally, the function is successfully created, as shown below:



Scheme 2: use other HTTP testing tools such as CURL and Postman to test the web function you have successfully created. Below is a sample test with CURL:

```
curl https://service-xxx.cd.apigw.tencentcs.com/release/
```

Deleting service

Run the following command to remove your deployed cloud resources.

```
sls remove
```

Web framework migration

Serverless Framework CLI provides an HTTP component specifically for web framework deployment, which can quickly implement features such as web framework deployment, layer creation, static/dynamic resource separation, and CDN acceleration. For usage, please see Deploying Framework on Command Line.