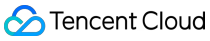
TencentDB for MongoDB

Development Specifications

Product Documentation

Copyright Notice

Trademark Notice



All trademarks associated with Tencent Cloud and its services are owned by the Tencent corporate group, including its parent, subsidiaries and affiliated companies, as the case may be. Trademarks of third parties referred to in this document are owned by their respective proprietors.

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

Development Specifications
Connection and Security Specifications
Connection Method Decision Guide

Last updated：2026-06-15 18:51:12

Scenario Description

In production environments, MongoDB deployments typically evolve from a replica set to a sharded cluster as your business scales. When managing database connections and routing across these architectures, you need to consider the following scenarios:

- Traffic Distribution and Load Balancing: By default, the load balancer (LB) for a sharded cluster uses a source IP hash policy. When client IPs are limited (for example, when a small number of application servers initiate high-concurrency requests), traffic may be concentrated on a subset of mongos nodes, resulting in uneven load distribution.
- Node Scaling and Connection Configuration Maintenance: After scaling mongos nodes in a sharded cluster, if the application does not promptly update the node address list in the connection string, the newly added nodes will not receive any traffic.
- Primary/Secondary Switchover and Topology Awareness: When a primary node switchover occurs, if the application does not promptly detect the topology change and continues sending write requests to the original primary node, those write operations will fail.

Connection Methods Overview

To address these connection challenges, TencentDB for MongoDB provides multiple connection methods for sharded clusters. These methods differ in aspects such as load balancing policies, node auto-discovery, and operational complexity. The following table summarizes them in order of recommended priority.

Sharded Cluster Connection Methods

For sharded clusters, where scaling mongos nodes is common, the auto-discovery capability provided by SRV records offers a significant advantage, making it the recommended solution. The load balancer (LB) address works well in production environments with a large number of client IPs and can meet the requirements of most use cases.

Priority	Connection Method	Protocol Format	Core Advantage
⭐⭐⭐ Recommended	SRV connection	`mongodb+srv://`	Automatically discovers Mongos nodes, requires no

			connection string changes during scaling, and provides driver-level CLB.
⭐⭐ Secondary Choice	LB load balancer address (default)	`mongodb://`	Easy to access, uses a single VIP to shield the backend topology, and is applicable to most production scenarios.
⭐ Supplementary	Connecting to all Mongos	`mongodb://`	Precisely controls traffic distribution and is applicable to special scenarios with uneven LB loads.

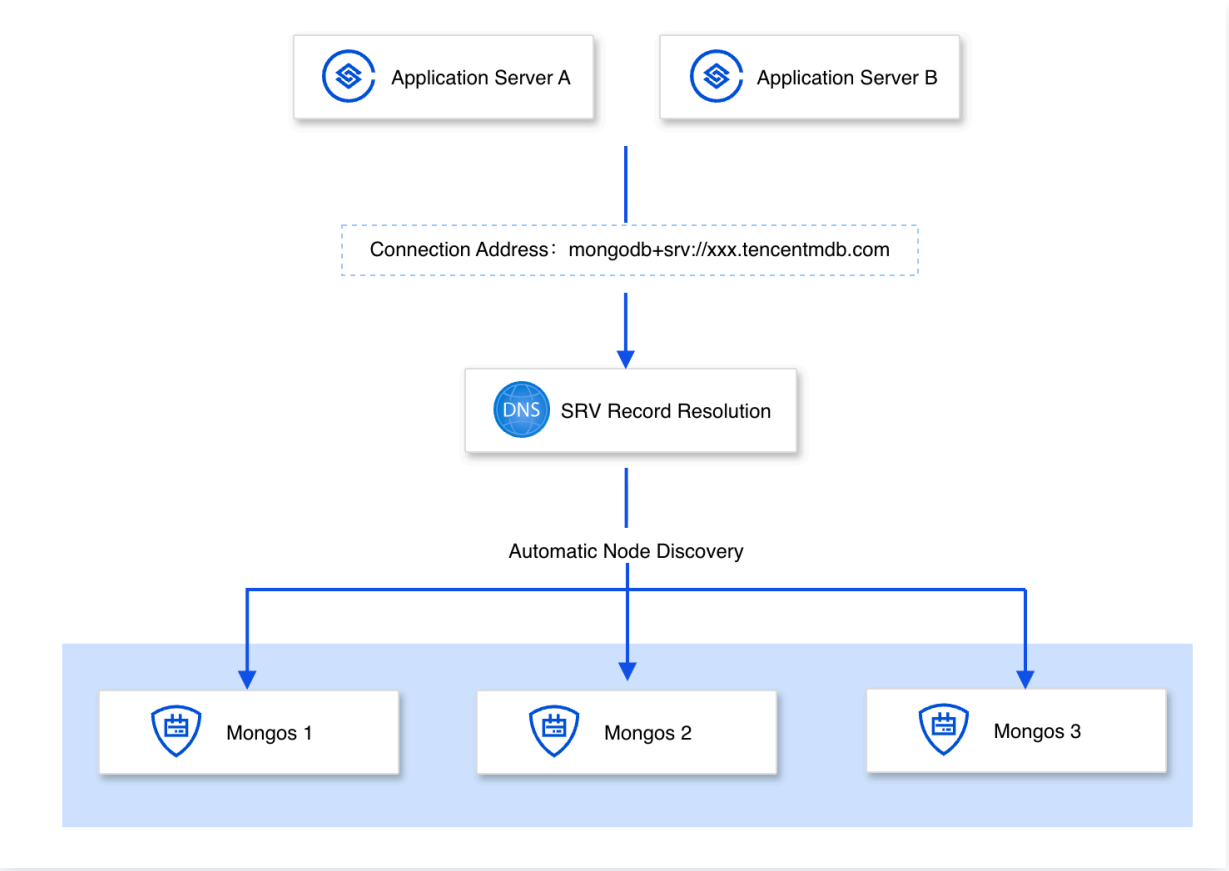
Method 1: SRV Connection (Recommended for Sharded Clusters)

SRV connections leverage the DNS auto-discovery mechanism and the MongoDB driver's native routing capability to enable flexible node access and request-level load balancing. For sharded clusters, the driver automatically detects changes when mongos nodes are scaled in or out, requiring no modifications to the connection string and no application restart. This makes SRV connections the recommended solution for sharded clusters in terms of flexibility. A replica set's node topology typically remains stable, with a fixed set of primary, secondary, and hidden nodes and infrequent node additions or removals. Therefore, the default connection string from the console is sufficient. SRV connections can also serve as an optional solution for replica sets, with the added benefit of a simpler connection string.

Connection Principles

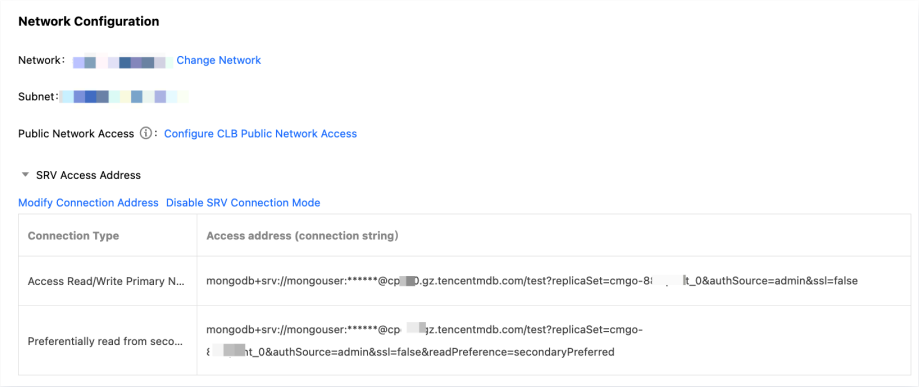
When an application connects via the mongodb+srv:// protocol, the driver queries DNS for SRV records to dynamically obtain the addresses of all mongos nodes in the cluster. When the cluster is scaled in or out, the DNS records are automatically updated. The driver then detects and connects to the new nodes on the next DNS refresh (typically within 60 seconds), with no application restart required.

> **Note:**
> The driver maintains a connection pool with all available mongos nodes and distributes requests across them in a round-robin fashion on the client side. This approach mitigates the single-point bottleneck caused by the source IP hash method used in traditional network load balancers. Furthermore, the official MongoDB driver natively supports session tracking, ensuring that all operations for getMore cursor scans and distributed transactions are routed to the same mongos node. This achieves load balancing while preserving consistency.



Enabling SRV Connection Mode

For MongoDB instances of version 4.0 or later, navigate to the TencentDB for MongoDB console. On the instance details page, locate the **Network Configuration** section and click **Enable SRV Connection Mode** to enable this connection method. For detailed steps, see Enable SRV Connection Mode.



Business Applications

A gaming company's sharded cluster was originally deployed with 8 mongos nodes. During an operational campaign, a sudden traffic surge required an emergency scale-out to 16 mongos nodes. With the traditional connection string method, the team would have had to manually add the addresses of the 8 new nodes in the application configuration, involving configuration adjustments and rolling restarts across 20+ microservices — a process that would have taken roughly 2 hours. After switching to SRV connections, the newly added mongos nodes were automatically resolved by DNS, and the driver discovered them on the next DNS refresh

(typically within 60 seconds), with no service disruption or application restart required. As a result, the scaling time was reduced from 2 hours to 5 minutes.

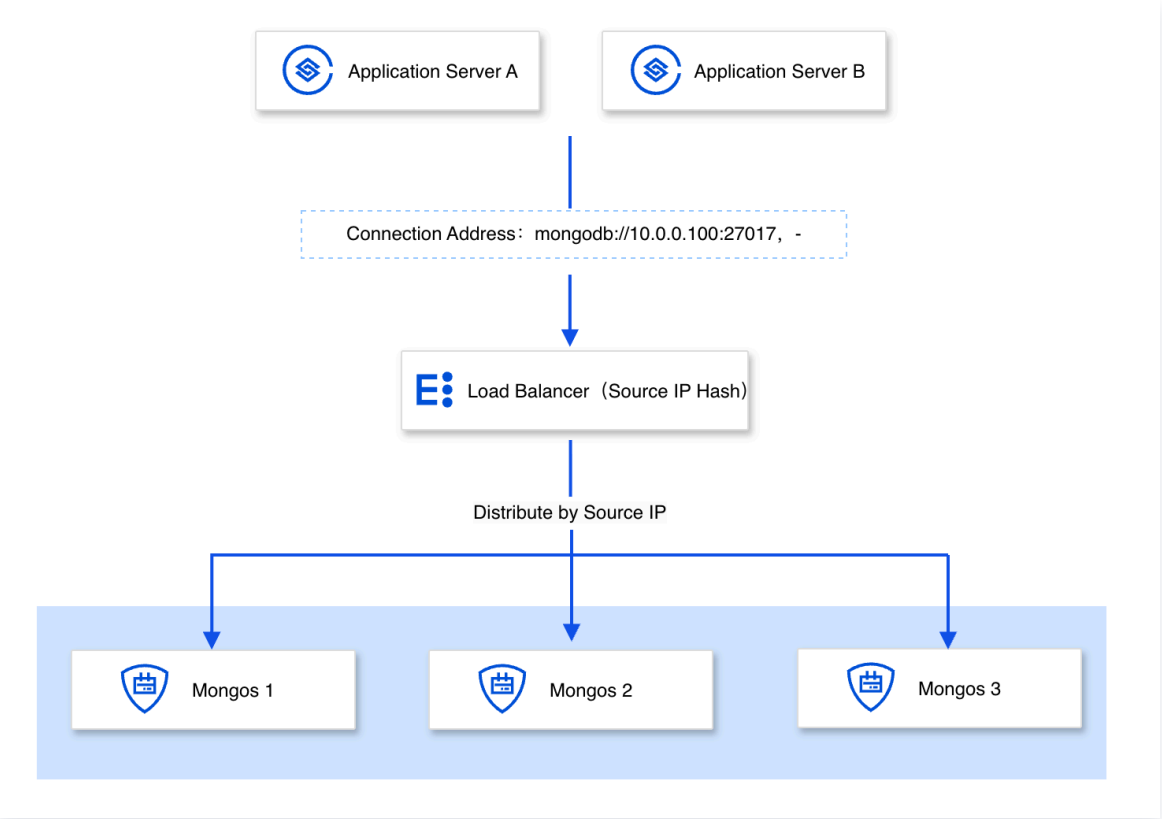
Method 2: CLB Address (Default Connection for Sharded Clusters)

TencentDB for MongoDB sharded cluster instances provide a load balancer (LB) address by default. This single VIP masks the backend topology of multiple Mongos nodes, offering simple access and low Ops costs, making it suitable as the default connection solution for sharded clusters.

Connection Principles

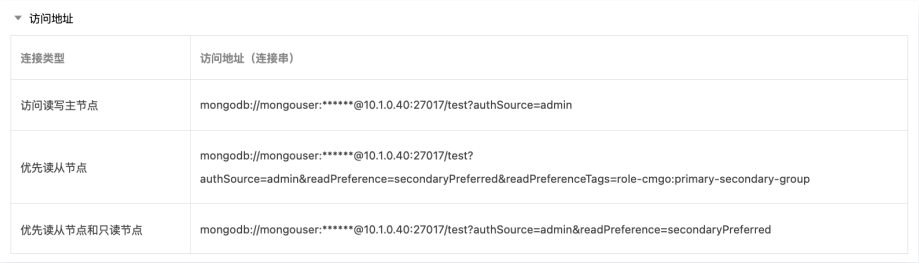
Applications access the database through a virtual IP (VIP) provided by the load balancer service, which masks the real private IP addresses (RSIPs) of the backend mongos nodes. For request distribution, the load balancer uses a source IP hash routing policy, ensuring that requests from the same client IP are always forwarded to the same mongos node. This meets the state consistency requirements for getMore cursor scans and distributed transactions. Furthermore, if a backend mongos node changes (for example, due to a failure and replacement), the load balancer dynamically updates the mapping between the VIP and RSIPs. In this case, the application layer does not need to adjust any connection configurations, enabling a fully transparent switchover of underlying nodes for the business.

> **Note:**
> The load balancer (LB) address is currently applicable only to sharded cluster architectures. For replica sets, we recommend using either the default connection string provided by the console (which already contains the addresses of all nodes) or the SRV connection method. In addition, if you observe uneven load distribution across mongos nodes in scenarios with a limited number of client IPs, we recommend switching to the SRV connection method or directly configuring the addresses of all mongos nodes in the connection string to optimize request distribution.



Obtaining Connection Methods

After the instance is created successfully, you can obtain the LB connection address in the **Access Address** section under **Network Configuration** on the **Instance Details** page of the MongoDB console.



访问地址

连接类型	访问地址（连接串）
访问读写主节点	mongodb://mongouser:******@10.1.0.40:27017/test?authSource=admin
优先读从节点	mongodb://mongouser:******@10.1.0.40:27017/test?authSource=admin&readPreference=secondaryPreferred&readPreferenceTags=role-cmgo:primary-secondary-group
优先读从节点和只读节点	mongodb://mongouser:******@10.1.0.40:27017/test?authSource=admin&readPreference=secondaryPreferred

Business Applications

Take an e-commerce platform as an example. Because its underlying business system uses an older database driver version that cannot parse the mongodb+srv:// protocol, the SRV connection method is not applicable. In such scenarios, using the VIP address provided by the load balancer (LB) to access the sharded cluster becomes the standard alternative. The application only needs to maintain a single VIP configuration to achieve transparent access to the entire cluster. When mongos nodes change or become unhealthy, the backend automatically isolates the faulty nodes and reroutes traffic, ensuring business continuity without any intervention from the application side.

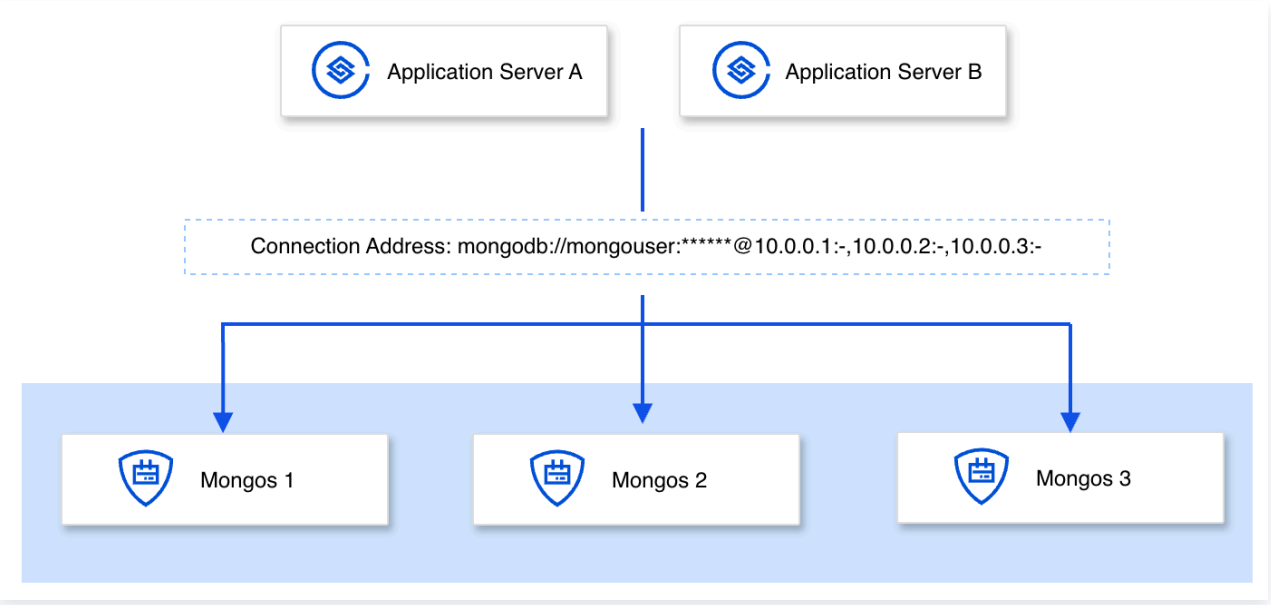
Method 3: Connecting to All Mongos Access Addresses

The sharded cluster supports enabling Mongos access, assigning a fixed address consisting of a shared VIP and an independent VPORT to each Mongos node. When a Mongos node fails, the system automatically binds a new Mongos process to it. The VIP and VPORT remain unchanged, ensuring connection stability.

Connection Principles

The system assigns fixed virtual IPs and ports to mongos nodes, and the underlying mechanism continuously monitors node health. If a failure occurs, the system automatically switches the access address to a healthy node within seconds. Because the external entry point remains static, upper-layer applications achieve transparent self-healing without any configuration changes, ensuring continuous and stable business connections.

> **Note:**
> If you have configured all Mongos nodes in the connection string, you must **synchronously** update the connection string on the application side after the number of Mongos nodes in the instance is adjusted.



Enabling Mongos Access Addresses

In the MongoDB console, select **Shard Instance**. Go to the **Mongos Node** tab under **Node Management** on the instance details page, and click **Enable Mongos Access Address**. For detailed steps, see Enable Mongos Access Address.

Comparison and Selection of Connection Methods

TencentDB for MongoDB offers multiple connection methods for sharded clusters, each with different strengths in access complexity, load balancing effectiveness, and operational flexibility. The following table

compares them across key decision-making dimensions and provides corresponding recommendations, helping you quickly identify the connection method best suited for your business scenario.

Decision Dimension	SRV Connection	LB Address	Connect to All Mongos
Access complexity	Low: a short connection string	Low: single VIP	Medium: List all nodes.
During Node Changes	No need to modify the connection string	No need to modify the connection string	⚠ Must update the connection string.
Load balancing policy	Driver-level balancing (connection-level)	Source IP Hash	Driver-level balancing (connection-level)
When client IPs are few	✔ Balanced	⚠ Possibly unbalanced	✔ Balanced
Number of connections characteristics	Number of connections amplification (connecting to all Mongos)	No number of connections amplification	Number of connections amplification (connecting to all Mongos)

Documentation

- Connecting to a Tencent Cloud MongoDB Instance
- Enabling SRV Connection Mode
- Mongos Load Balancing and Connection Solutions
- Enable Mongos Access Address.
- Enable Public Network Access.

Connection Configuration Specifications

Last updated：2026-06-15 18:51:12

Scenario Description

Once the database connection topology (network routing) is established, connection pool configuration and security safeguards at the application code level directly determine system stability and the security of your data assets. In daily development and troubleshooting, teams frequently encounter the following high-risk scenarios caused by misconfiguration:

- **Connection Limit Exceeded Risk**: In a microservices architecture, frequent use of short-lived connections or a maxPoolSize set too high across multiple application instances can cause the total number of concurrent connections to exceed the database's connection limit. This causes new requests to queue or time out, affecting overall system availability.
- **Inappropriate Read/Write Policy Matching**: Read/write separation must align with your business scenario. Reading from a secondary node during core transactions can lead to data inconsistency due to replication lag between the primary and secondary nodes. Conversely, if read/write separation is not configured for high-frequency analytical and reporting queries, the performance load on the primary node will increase significantly.
- **Data Security and Compliance Risks**: During development or testing, some services may enable public network access for the database without configuring strong authentication passwords. Such configurations — lacking network boundary controls and least-privilege isolation — fail to meet security and compliance requirements and significantly increase the risk of data leakage, malicious tampering, and ransomware attacks.

Connection Specification Overview

Specification No.	Specification Name	Core Requirement	Applicable Scope	Constraint Level
Specification 1	Correctly configure core connection string parameters	• You must use a high-availability connection address. • You must include `authSource=admin`. • You must specify `replicaSet` for the replica set. • Enable `retryWrites` and `retryReads`.	Replica set / Sharded cluster	Mandatory

Specification 2	Use connection pools (long connections)	Avoid using short connections in production environments. Use connection pools to reuse connections.	Replica set / Sharded cluster	Mandatory
Specification 3	Properly configuring connection pool parameters	Calculate `maxPoolSize` based on business concurrency. Keep the total number of connections below 80% of the instance limit.	Replica set / Sharded cluster	Mandatory
Specification 4	Selecting read preference based on business scenarios	For core strong-consistency services, read from the Primary. For non-sensitive services, read from a Secondary. For read/write splitting, `secondaryPreferred` is recommended.	Replica set / Sharded cluster	Recommendation
Specification 5	Do not disable authentication or arbitrarily expose to the public network in the production environment.	• Enable authentication and restrict access to the private network only. • Access from the public network must go through a CLB with security group and IP address restrictions configured.	Replica set / Sharded cluster	Mandatory

Specification 1: Configuring Core Connection String Parameters

The configuration of the connection string (URI) parameters directly determines the availability of the database. You must adhere to the following four core principles:

1. Access address: Single-point connections are strictly prohibited.
    - Instruction: The connection string must contain all available nodes within the cluster (Seed List).
    - Reason: To avoid a single point of failure. Pointing to a single node only prevents the driver from automatically discovering the new Primary during a node switchover, resulting in a loss of high availability.
2. Authentication parameters: Specify `authSource`.
    - Instruction: Explicitly specify the authentication database parameter.
        - Users created through the console: `authSource=admin` is configured by default.
        - Users created through the command line: Configure them to the logical database to which they belong.

- Reason: To ensure accurate authentication logic and avoid connection authentication failures caused by inconsistent default databases.

3. Architecture awareness: You must include `replicaSet`.
   - Instruction: The replica set connection string must include `replicaSet=<replica-set-name>`.
   - Reason: Only by explicitly specifying this parameter can the driver automatically redirect traffic to the new primary node during a failure, achieving seamless switchover.
4. Disaster recovery enhancement: Enable automatic retries (recommended).
   - Instruction: Append the parameters `retryWrites=true` and `retryReads=true`.
   - Reason: To improve network fault tolerance. During transient network jitter or master-slave elections, the driver layer automatically initiates retries to minimize the impact of failures.

Key Connection String Parameters

Parameter	Required or Not	Description	Example Value
`authSource`	Yes	The authentication database. Tencent Cloud uniformly sets it to admin.	`admin`
`replicaSet`	Required for replica set	Obtain the replica set name from the console.	`cmgo-xxxxxxxx`
`readPreference`	Recommendation	Read preference setting	`secondaryPreferred`
`maxPoolSize`	Recommendation	Maximum number of connections in the connection pool.	`150`
`retryWrites`	Recommendation	Automatic write retry	`true`
`retryReads`	Recommendation	Automatic read retry	`true`
`w`	Required for core business	Write acknowledgment level	`majority`

Business Applications

A financial system runs on a replica set architecture (1 primary + 2 secondaries), but the connection string does not specify the replicaSet parameter. Without this parameter, the MongoDB driver treats the connection as standalone mode and connects directly to the single node specified in the connection string, with no awareness of the overall replica set topology.

- No exceptions occurred during normal operation. However, during a scheduled maintenance operation, a primary failover took place on the primary node. The original primary was demoted to a secondary, and another secondary was elected as the new primary. At this point, the driver continued sending write requests to the original node (now demoted to a secondary). Since secondary nodes reject write operations by default, a large number of not master errors appeared on the application side, and write failures persisted for approximately 15 minutes.
- The operations team manually restarted the application and updated the connection address to restore service. After adding the replicaSet=cmgo-xxxxxxxx parameter to the connection string, the driver runs in replica set mode, automatically detects topology changes, and routes write requests to the new primary within seconds of a failover. This eliminates the need to restart applications or modify connection addresses, enabling automatic failure recovery.

Specification 2: Using Connection Pools (Long Connections)

For production environments, use connection pools (persistent connections) and avoid short-lived connection mode.

- In short-lived connection mode, each database request must go through the full sequence of TCP three-way handshake, TLS negotiation (if encryption is enabled), and SCRAM authentication. Establishing a single connection typically takes 10—50 ms.
- In high-concurrency scenarios, frequent connection creation and destruction not only significantly increase request latency but also quickly exhaust the instance's connection limit, causing subsequent requests to fail due to timeouts when connections cannot be obtained.
- Connection pools pre-establish connections and reuse them across multiple requests, eliminating the overhead of repeated handshakes and authentication. This allows the application to maintain a stable, predictable number of connections under high concurrency, with response latency at the millisecond level.

Short Connections vs. Long Connections

Dimension	Short-lived connection	Long-lived Connection (Connection Pool)
Connection establishment	Create a new connection for each request	Reusing established connections
Authentication overhead	Authentication is required for each request (10-50ms).	Authentication only on the first connection
Number of connections	Increases linearly with QPS	Stable and controllable
Applicable Scenario	Not suitable for production environments	All production environments

Business Applications

An e-commerce platform initially used PHP's short-lived connection mode, where every database request established a new connection that was immediately destroyed after the request completed. The system ran normally under daily traffic (around 1,000 QPS). However, during a major promotional event, the QPS surged from 1,000 to 10,000. Because each request required a dedicated connection, the rate of connection establishment could not keep up with the rate of incoming requests. The number of connections to the MongoDB instance soared from 200 to 3,000 (the instance's default limit) within minutes. As a result, all new requests timed out while queuing for connections. The average API response time deteriorated from 50 ms to over 5 seconds, causing a large number of order submission failures.
After switching to a connection pool (maxPoolSize=200), the application pre-established a batch of connections at startup and reused them across requests, eliminating the overhead of repeated handshakes and authentication. Even when the QPS reached 50,000 during the following year's promotional event, the actual number of connections remained stable below 500, and the API response time consistently stayed at the millisecond level.

> **Note:**
> For architectures such as Serverless and PHP-FPM that cannot maintain persistent connection pools in application memory, it is recommended to introduce a middleware proxy (Proxy) between the business layer and the database to implement connection pooling.

Specification 3: Configuring Connection Pool Parameters

The connection pool size (maxPoolSize) is not always better when larger. If the connection pool size is configured too high, the number of global connections can be instantly saturated, causing new requests to be rejected by the database. If the connection pool size is configured too low, local queuing occurs during peak business hours, unnecessarily increasing request latency. To correctly configure the connection pool, you must comprehensively consider the time consumption of a single request, the scaling boundaries of application instances, and the network access topology.

Calculating Connections for Different Connection Methods

The theoretical total number of connections across all applications must not exceed 80% of the MongoDB instance's connection limit, leaving headroom for sudden business growth or reconnections after a primary failover. For sharded clusters, the driver's connection pool establishment logic exhibits a "multiplier amplification effect" due to how the cluster is accessed. Before performing calculations, clarify the following three key variables:

- A (Application): The total number of business applications currently deployed.
- P (Pool Size): The maxPoolSize value configured for a single application.
- M (Mongos): The total number of Mongos nodes, which is resolved through SRV or specified in the connection string.

Connection Access Method	Driver Connection Behavior	Theoretical Total Connection Count Formula
LB CLB (private network VIP, and so on)	The driver treats the LB as a single node, establishes only one connection pool with the LB, and the LB distributes requests at the underlying layer.	A x P
SRV record connection	The driver automatically parses the SRV record and establishes an independent connection pool for each Mongos node behind the cluster.	A x P x M
Direct connection to all Mongos	The driver establishes an independent connection pool for each Mongos node explicitly configured in the connection string (URI).	A x P x M

maxPoolSize Parameter Estimation Recommendations

Most connection limit exceeded failures are caused by developers blindly using default values or incorrectly equating QPS with number of connections. Follow the steps below to deduce the appropriate maxPoolSize for a single application:

Step 1: Calculating the Base Concurrency Requirement for a Single Application

The number of concurrent connections depends on the request volume and processing speed. The formula is: Basic Requirement per Application = (Expected Peak QPS per Application × Average Database Response Time in Seconds) × 1.5 (Anti-Jitter Factor). For example, with a single-instance peak QPS of 1000 and an average query time of 0.02 seconds (20 ms), the basic concurrent requirement is 1000 × 0.02 × 1.5 = 30.

Step 2: Calculating the Final Value Based on the Topology Architecture

- If the deployment is in CLB mode: you can directly use the basic requirement, meaning maxPoolSize ≈ 30.
- For SRV or direct multi-Mongos mode: Because requests are automatically distributed by the driver to the connection pools of M nodes, you must proportionally reduce the pool size to prevent the global total number of connections from exceeding the limit by a multiple. That is, maxPoolSize ≈ Basic Requirement ÷ M.

Connection Pool Parameter Settings Reference

Parameter	Description	Recommended Value	Consequences of Improper Configuration

`maxPoolSize`	Maximum Number of Connections	50–200	Too large: Exceeds limit; Too small: Queuing
`minPoolSize`	Minimum Number of Connections	5–20	Too small: Slow cold start
`maxIdleTimeMS`	Maximum Idle Time	60000–300000	Too small: Frequent re-creation
`waitQueueTimeoutMS`	Wait Timeout	5000–10000	Too short: Normal requests are rejected
`connectTimeoutMS`	Connection Timeout	10000–30000	Too short: Fails during network jitter

Business Applications

A content platform deployed 20 applications, all connecting to a sharded cluster through a CLB. The maxPoolSize for each application was set to 200 (the driver's default value). The theoretical total number of connections was 20 × 200 = 4,000, which had already created a hidden risk of exceeding the MongoDB instance's connection limit (3,000).

- During daily off-peak hours, the connection pool was not fully utilized and operated normally due to low actual concurrency. However, during a business scale-out, five new applications were added. When all 25 applications started simultaneously and initialized their connection pools, the number of connections instantly approached 5,000, triggering the instance's connection limit protection. As a result, the five newly started applications all reported `connection pool exhausted` errors and failed to connect to the database, leaving the newly launched feature modules completely unavailable.
- After troubleshooting, the team adjusted `maxPoolSize` from 200 to 100. This reduced the theoretical total to 25 × 100 = 2,500 connections — 83% of the instance's connection limit — while reserving capacity for future scaling to 30 applications. At the same time, `minPoolSize=10` was set to ensure sufficient warm connections during cold starts, resolving the connection avalanche issue during scaling.

Specification 4: Selecting Read Preference Based on Business Scenarios

The readPreference setting determines which node the MongoDB driver routes read requests to. Different business scenarios have different consistency requirements: core transactions such as transfers and order placement require the most up-to-date data and must read from the primary node. In contrast, latency-tolerant scenarios such as analytical reporting and historical queries can read from secondary nodes to offload traffic from the primary. An inappropriate choice leads to two common issues:Routing strongly consistent reads to a secondary node can cause the application to read stale data, triggering logical errors. Concentrating all reads on the primary node leaves secondary node resources idle and turns the primary into a bottleneck under high concurrency.

Read Preference Selection Decision Table

> **Note:**
> When read/write separation is required for your business, using `secondaryPreferred` is recommended for higher availability. If a business only accesses read-only nodes, configure two or more read-only nodes to achieve load balancing for read requests. The connection string for read-only nodes can be directly obtained from the network configuration on the instance details page.

Read Preference	Consistency	Use Cases
`primary`	Strong consistency	Check balance after transfer, check order status after placing an order
`primaryPreferred`	Strong consistency preferred	For general services, read operations fall back to a Secondary when the Primary is unavailable.
`secondary`	Eventual consistency	Read-only scenarios, report queries, data analysis
`secondaryPreferred`	Eventual consistency preferred	Read-intensive scenarios, offloading Primary pressure (Recommended)
`nearest`	Node-dependent	Geographically distributed deployment, accessing the nearest node

Business Applications

In a bank's core system, the account balance query API was configured with `readPreference=secondary`. After a user completed a transfer (written to the primary node) and immediately queried the balance, the data on the secondary node had not yet been updated due to a 10—100 ms replication lag. As a result, the API returned the pre-transfer balance. Seeing that the transfer succeeded but the balance remained unchanged, users frequently initiated duplicate transfers or contacted customer service to complain. After the API was changed to `readPreference=primary`, balance queries were served directly from the primary node, ensuring that the accurate balance was returned immediately after a transfer, and customer complaints dropped significantly. For non-real-time operations such as historical bill queries and monthly report exports, `readPreference=secondaryPreferred` was retained. This routed approximately 60% of the read traffic to secondary nodes, reducing the primary node's CPU utilization from 85% to 50% and improving both read and write latency.

> **Note:**
> When secondary or secondaryPreferred is enabled, your business logic must be designed to tolerate the risk of Stale Reads due to replication latency.

Specification 5: Disabling Authentication or Exposing to the Public Network Is Prohibited in Production Environments

For MongoDB instances in a production environment, you must enable authentication and allow access only via a private network (VPC). An instance with authentication not enabled is equivalent to granting full read/write permissions to any client that can access its IP address. Once discovered by scanning, it faces the risk of data leakage or being held for ransom through database deletion. If your business genuinely requires public network access, you must configure the public network access service through CLB and simultaneously meet the following conditions: configure security groups for both the CLB and the MongoDB instance, allowing only specified client public IP addresses; enable SSL/TLS encryption to prevent data interception during transmission over the public network; use a strong password (12 characters or more, containing uppercase and lowercase letters, numbers, and special characters) to reduce the risk of brute-force attacks.

Security Configuration Requirements

1. Network boundary isolation (VPC and security groups): Access is restricted to the private network by default. Public network access must undergo strict network boundary control.
   - Default private network direct connection: In a production environment, you must restrict access to private network access only via application servers within the same VPC.
   - Compliant public network access architecture: If your business (such as cross-region debugging or external data direct connection) genuinely requires public network access, you must not directly bind a public IP address to the instance. You must establish the public network access path through CLB and enforce two-layer security group control:
     - CLB security group: Only allow the public IP addresses of specific clients (such as the company's egress IP) and the listener port, implementing the first layer of interception.
     - MongoDB security group: Only allow the specified client public IP addresses and port 27017. You must also allow the VIP address of the CLB instance to ensure the CLB node's health checks on the backend database function properly.
2. Access authentication and least privilege authorization: Disable password-free login and strictly separate application accounts from administrative accounts.
   - Mandatory authentication enforcement: Authentication must be enabled for the instance.
   - Strong password validation: Database passwords must be longer than 12 characters and contain uppercase and lowercase letters, numbers, and special characters. Avoid using weak passwords such as admin123 or root to prevent brute-force attacks.

- Adhere to the principle of Least Privilege: Do not allow business applications to directly connect to the database using the root account or super accounts with the dbAdminAnyDatabase privilege. You must create a dedicated database account for each specific business module and grant it only the readWrite permission for the corresponding business database.

3. Transport layer encryption (TLS/SSL): All transmissions involving public network links must be encrypted.

   Preventing packet capture and man-in-the-middle attacks: Once the public network access link is enabled, you must enable the SSL/TLS encrypted transmission feature in the console. Additionally, explicitly append tls=true to the application-side connection string to ensure that data is not intercepted in plaintext or tampered with during transmission over the untrusted public network.

Security Configuration Self-Check List

Check Dimension	Check Item	Compliance Standard
Network Layer	Public Network Exposure Surface	Access is allowed only via a private network; public network access must be proxied through CLB and configured with an allowlist.
Network Layer	Security group policy	Do not configure the 0.0.0.0/0 allow-all rule. Only allow access from specified source IPs.
Authentication Layer	Identity authentication	Enable the authentication feature.
Authentication Layer	Password complexity	The length must be at least 12 characters, containing uppercase letters, lowercase letters, digits, and special symbols.
Authorization Layer	Principle of Least Privilege	Application accounts have only read and write permissions on the target database, and no administrative permissions.
Transport Layer	SSL/TLS encryption	Enable SSL/TLS when public network access is enabled.
Application	Secure storage of	Do not hardcode database account passwords in the code. Use environment variables or the KMS service instead.

layer	credentials

Documentation

- Connecting to a Tencent Cloud MongoDB Instance
- Enabling SRV Connection Mode
- Mongos Load Balancing and Connection Solutions
- Enable Public Network Access

Configuring SDK Connections

Last updated：2026-06-15 18:51:12

Proper SDK connection configuration is fundamental to ensuring stable operation of the MongoDB service. The parameters in the connection string directly determine the authentication method, failover behavior, connection pool efficiency, and read/write reliability. Missing or incorrect configuration of any of these items can lead to connection failure, performance degradation, or data consistency issues in a production environment. This document provides connection configuration examples for four mainstream SDKs: Python, Java, Node.js, and Go. All examples come pre-configured with production-recommended settings, including connection pooling, timeouts, retries, write concern (`w=majority`), and read preference (`secondaryPreferred`). They can be used directly in production or have their parameter values adjusted based on actual business needs. Replace the username, password, and address in the connection string with your actual instance information (which can be obtained from the Tencent Cloud console > MongoDB > Instance Details page).

Python(pymongo)

Within the Python ecosystem, `pymongo` is the officially maintained MongoDB driver library. It provides a synchronous connection mode, making it suitable for scenarios such as Web backends (such as Flask and Django) and data processing scripts. The following example shows the complete configuration for four connection methods, including production-recommended parameters such as connection pooling, timeouts, retries, and read/write policies. The connection pool in `pymongo` is automatically managed internally by the driver. The number of connections is controlled via `maxPoolSize` and `minPoolSize`, eliminating the need for manual creation or release of connections.

```
from pymongo import MongoClient
from pymongo.read_preferences import ReadPreference

# Sharded Cluster: SRV Connection (Automatically Discovers Mongos, No
Connection String Modification Required for Scaling)
client = MongoClient(
    "mongodb+srv://mongouser:password@xxx.tencentcdb.com/admin",
    authSource="admin",

    # Connection Pool Configuration
    maxPoolSize=150,
    minPoolSize=10,
    maxIdleTimeMS=120000,
    waitQueueTimeoutMS=5000,
```

```
    # Timeout Configuration
    connectTimeoutMS=10000,
    serverSelectionTimeoutMS=5000,

    # Reliability Configuration
    retryWrites=True,
    retryReads=True,
    w="majority",

    # Read Preference
    readPreference=ReadPreference.SECONDARY_PREFERRED
)

# Replica Set: Use the Console Default Connection String (Contains All
Node Addresses)
client = MongoClient(

"mongodb://mongouser:password@10.0.0.100:27017,10.0.0.101:27017,10.0.0.1
02:27017/admin",
    authSource="admin",
    replicaSet="cmgo-xxxxxxxx",
    maxPoolSize=150,
    retryWrites=True,
    w="majority",
    readPreference=ReadPreference.SECONDARY_PREFERRED
)

# Sharded Cluster: LB Address Connection (Default Method, Applicable to
the Vast Majority of Scenarios)
client = MongoClient(
    "mongodb://mongouser:password@10.0.0.100:27017/admin",
    authSource="admin",
    maxPoolSize=150,
    retryWrites=True,
    w="majority",
    readPreference=ReadPreference.SECONDARY_PREFERRED
)
```

```
# Sharded Cluster: Connect to All Mongos (Requires Enabling Mongos
Access Addresses First)
client = MongoClient(

"mongodb://mongouser:password@10.0.0.100:27017,10.0.0.100:27018,10.0.0.1
00:27019/admin",
    authSource="admin",
    maxPoolSize=150,
    retryWrites=True,
    w="majority",
    readPreference=ReadPreference.SECONDARY_PREFERRED
)

# Connection Health Check
def check_connection():
    try:
        client.admin.command('ping')
        print("MongoDB connection is normal")
        return True
    except Exception as e:
        print(f"MongoDB connection failure: {e}")
        return False
```

Java(MongoDB Driver)

In the Java ecosystem, `mongodb-driver-sync` is the official synchronous driver provided by MongoDB and is widely used in enterprise-level projects such as Spring Boot and microservices. The Java driver configures connection parameters using the `MongoClientSettings` builder pattern, centrally managing configurations such as connection pooling, timeouts, write concern, and read preference. The following example encapsulates four connection methods as independent functions, sharing the same set of `MongoClientSettings` configurations to facilitate switching between different architectures. The connection pool is maintained internally by the driver. Applications only need to call the corresponding function to obtain a `MongoClient` instance, eliminating the need for manual connection lifecycle management.

```
import com.mongodb.ConnectionString;
import com.mongodb.MongoClientSettings;
import com.mongodb.ReadPreference;
import com.mongodb.WriteConcern;
import com.mongodb.client.MongoClient;
```

```
import com.mongodb.client.MongoClients;
import java.util.concurrent.TimeUnit;

public class MongoDBConfig {

    // Sharded Cluster: SRV Connection
    public static MongoClient createClientWithSRV() {
        String uri =
"mongodb+srv://mongouser:password@xxx.tencentcdb.com/admin"
                  + "?authSource=admin";
        return createClientFromUri(uri);
    }

    // Replica Set: Console Default Connection String (Contains All Node
Addresses)
    public static MongoClient createClientWithReplicaSet() {
        String uri = "mongodb://mongouser:password"
                  +
"@10.0.0.100:27017,10.0.0.101:27017,10.0.0.102:27017/admin"
                  + "?authSource=admin&replicaSet=cmgo-xxxxxxxx";
        return createClientFromUri(uri);
    }

    // Sharded Cluster: LB Address Connection (Applicable to the Vast
Majority of Scenarios)
    public static MongoClient createClientWithLB() {
        String uri =
"mongodb://mongouser:password@10.0.0.100:27017/admin"
                  + "?authSource=admin";
        return createClientFromUri(uri);
    }

    // Sharded Cluster: Connect to All Mongos
    public static MongoClient createClientWithAllMongos() {
        String uri = "mongodb://mongouser:password"
                  +
"@10.0.0.100:27017,10.0.0.100:27018,10.0.0.100:27019/admin"
                  + "?authSource=admin";
        return createClientFromUri(uri);
    }
```

```
    private static MongoClient createClientFromUri(String uri) {
        MongoClientSettings settings = MongoClientSettings.builder()
            .applyConnectionString(new ConnectionString(uri))

            // Connection Pool Configuration
            .applyToConnectionPoolSettings(builder -> builder
                .maxSize(150)
                .minSize(10)
                .maxConnectionIdleTime(120, TimeUnit.SECONDS)
                .maxWaitTime(5, TimeUnit.SECONDS))

            // Timeout Configuration
            .applyToSocketSettings(builder -> builder
                .connectTimeout(10, TimeUnit.SECONDS)
                .readTimeout(30, TimeUnit.SECONDS))

            .applyToClusterSettings(builder -> builder
                .serverSelectionTimeout(5, TimeUnit.SECONDS))

            // Reliability Configuration
            .retryWrites(true)
            .retryReads(true)
            .writeConcern(WriteConcern.MAJORITY)
            .readPreference(ReadPreference.secondaryPreferred())

            .build();

        return MongoClients.create(settings);
    }
}
```

Node.js(MongoDB Driver)

In the Node.js ecosystem, `mongodb` is the official driver package. It is based on an asynchronous I/O model and is suitable for scenarios such as high-concurrency Web services (such as Express, Koa, and NestJS) and Serverless functions. The following example extracts common parameters such as connection pooling, timeouts, retries, and read/write policies into a `commonOptions` object. The four connection methods share the same set of configurations, and switching connection methods only requires modifying the URI. The connection pool of the Node.js driver is also automatically managed by the driver. After each

call to `client.connect()`, connections from the pool can be reused, eliminating the need to manually create or close individual connections.

```
const { MongoClient, ReadPreference } = require('mongodb');

// General Connection Configuration
const commonOptions = {
    maxPoolSize: 150,
    minPoolSize: 10,
    maxIdleTimeMS: 120000,
    waitQueueTimeoutMS: 5000,
    connectTimeoutMS: 10000,
    serverSelectionTimeoutMS: 5000,
    retryWrites: true,
    retryReads: true,
    w: 'majority',
    readPreference: ReadPreference.SECONDARY_PREFERRED
};

// Sharded Cluster: SRV Connection
async function connectWithSRV() {
    const uri =
"mongodb+srv://mongouser:password@xxx.tencentcdb.com/admin?
authSource=admin";
    const client = new MongoClient(uri, commonOptions);
    await client.connect();
    await client.db('admin').command({ ping: 1 });
    console.log('SRV connection successful');
    return client;
}

// Replica Set: Console Default Connection String
async function connectReplicaSet() {
    const uri =
"mongodb://mongouser:password@10.0.0.100:27017,10.0.0.101:27017,10.0.0.1
02:27017/admin?authSource=admin&replicaSet=cmgo-xxxxxxxx";
    const client = new MongoClient(uri, commonOptions);
    await client.connect();
    console.log('Replica set connection successful');
```

```
    return client;
}

// Sharded Cluster: LB Address Connection (Applicable to the Vast
Majority of Scenarios)
async function connectWithLB() {
    const uri = "mongodb://mongouser:password@10.0.0.100:27017/admin?
authSource=admin";
    const client = new MongoClient(uri, commonOptions);
    await client.connect();
    console.log('LB connection successful');
    return client;
}

// Sharded Cluster: Connect to All Mongos
async function connectWithAllMongos() {
    const uri =
"mongodb://mongouser:password@10.0.0.100:27017,10.0.0.100:27018,10.0.0.1
00:27019/admin?authSource=admin";
    const client = new MongoClient(uri, commonOptions);
    await client.connect();
    console.log('All Mongos connection successful');
    return client;
}
```

Go(mongo-driver)

In the Go ecosystem, `mongo-driver` is the official driver library maintained by MongoDB, suitable for high-performance backend services and microservices architectures. The Go driver configures connection parameters through the `options.Client()` chained call, supporting a complete set of configuration items such as connection pooling, timeouts, retries, and read/write policies. The following example encapsulates common configurations in the `createClient` function. The four connection methods can reuse the same configuration logic by simply passing in different URIs. The Go driver requires passing a `context.Context` during connection to control timeouts. It is recommended to create a `mongo.Client` instance once at application startup and reuse it throughout the entire lifecycle to avoid repeatedly establishing connections.

```
package main
```

```
import (
    "context"
    "log"
    "time"

    "go.mongodb.org/mongo-driver/mongo"
    "go.mongodb.org/mongo-driver/mongo/options"
    "go.mongodb.org/mongo-driver/mongo/readpref"
    "go.mongodb.org/mongo-driver/mongo/writeconcern"
)

// Sharded Cluster: SRV Connection
func createClientWithSRV() (*mongo.Client, error) {
    uri := "mongodb+srv://mongouser:password@xxx.tencentcdb.com/admin?
authSource=admin"
    return createClient(uri)
}

// Replica Set: Console Default Connection String (Contains All Node
Addresses)
func createClientWithReplicaSet() (*mongo.Client, error) {
    uri :=
"mongodb://mongouser:password@10.0.0.100:27017,10.0.0.101:27017,10.0.0.1
02:27017/admin" +
           "?authSource=admin&replicaSet=cmgo-xxxxxxxx"
    return createClient(uri)
}

// Sharded Cluster: LB Address Connection (Applicable to the Vast
Majority of Scenarios)
func createClientWithLB() (*mongo.Client, error) {
    uri := "mongodb://mongouser:password@10.0.0.100:27017/admin?
authSource=admin"
    return createClient(uri)
}

// Sharded Cluster: Connect to All Mongos
func createClientWithAllMongos() (*mongo.Client, error) {
    uri :=
"mongodb://mongouser:password@10.0.0.100:27017,10.0.0.100:27018,10.0.0.1
```

```
00:27019/admin" +
            "?authSource=admin"
    return createClient(uri)
}

func createClient(uri string) (*mongo.Client, error) {
    clientOpts := options.Client().ApplyURI(uri).
        // Connection Pool Configuration
        SetMaxPoolSize(150).
        SetMinPoolSize(10).
        SetMaxConnIdleTime(120 * time.Second).

        // Timeout Configuration
        SetConnectTimeout(10 * time.Second).
        SetServerSelectionTimeout(5 * time.Second).

        // Reliability Configuration
        SetRetryWrites(true).
        SetRetryReads(true).
        SetWriteConcern(writeconcern.Majority()).
        SetReadPreference(readpref.SecondaryPreferred())

    ctx, cancel := context.WithTimeout(context.Background(),
10*time.Second)
    defer cancel()

    client, err := mongo.Connect(ctx, clientOpts)
    if err != nil {
        return nil, err
    }

    // Connection Health Check
    if err := client.Ping(ctx, readpref.Primary()); err != nil {
        return nil, err
    }
    log.Println("MongoDB connection is normal")

    return client, nil
}
```

Data Modeling and Schema Design Specifications

Last updated：2026-06-16 12:02:06

Scenarios

MongoDB's flexible document model facilitates rapid business iteration. In practical applications, however, this flexibility must be combined with sound design principles to ensure long-term system stability. Based on past online operational experience, if adequate modeling guidance is lacking during the initial phase, the following common data modeling pitfalls can introduce performance challenges during periods of rapid business growth:

- Divergent naming conventions: The coexistence of naming styles such as OrderDetail, order_detail, and orderdetail within the same project increases communication costs for later code maintenance and data troubleshooting.
- Unbounded array growth: Developers habitually push continuously generated child data, such as a user's complete historical login logs, into an array within the same document. Without a truncation mechanism, the 16MB physical limit of a single MongoDB BSON document may be reached, causing some write requests to be blocked.
- Field type drift: A single field, such as age, contains a mixture of numeric and string types. This not only increases frontend parsing complexity but may also, under certain conditions, affect the efficiency of underlying indexes, leading to unexpected query results.
- Excessive collection fragmentation: If a single cluster contains too many collections (exceeding 5000), it increases the metadata management load on the underlying WiredTiger storage engine, potentially prolonging instance startup time and, in extreme cases, increasing memory overhead.
- Over-reliance on collection splitting and joins: Developers habitually apply relational database thinking, forcibly splitting the "order master collection" and "order detail collection" into two separate collections. This practice degrades an operation that could have been completed with a single embedded read into multiple round-trip network I/O calls.

Objective of this document: To help you fully leverage the advantages of the MongoDB document model and avoid designing MongoDB with a relational database mindset.

Naming Specifications

Specification 1: Prohibition on Using System Database Names

- **Core Actions:**

  Business data is strictly prohibited from occupying or sharing MongoDB's system-reserved databases (including but not limited to admin, local, and config).

Rule	Requirement	Correct Example	Incorrect Example
Red line	Isolate business data from system metadata.	db_config (independent business database)	admin,local,config

- **Business Application:**

  A team, for convenience, directly created a business collection within the admin database to store system configuration data. Because the admin database handles core functions such as user authentication and cluster management, high-priority system locks may need to be obtained during administrative operations, competing with business reads and writes and causing the entire database's response to slow down dramatically. Performance recovered only after the configuration data was migrated to a separate business database.

Specification 2: Database Naming

- **Core Actions:**

  It is recommended to use a database name with the prefix `db_` + lowercase letters + underscores.
- **Naming Conventions and Examples:**

Rule	Requirement	Correct Example	Incorrect Example
Prefix	Start with db_	db_order	Order
Character Set	Lowercase letters + underscores	db_user_center	db.user.center
Length	⩽ 64 bytes	db_payment	db-payment

- **Business Application:**

  A development team created a database named `UserCenter`. However, in the production environment (Linux), the connection strings for some microservices were mistakenly written as `usercenter`. Because MongoDB is strictly case-sensitive for database names in Linux environments, a brand new empty database was inadvertently created in production, causing widespread null pointer errors in core business queries. Furthermore, due to the lack of a unified prefix, DBAs found it difficult to quickly identify which databases were core business ones when cleaning up historically abandoned temporary databases within the same cluster. After the naming convention of a `db_` prefix followed by all lowercase letters (such as `db_user_center`) is fully implemented, case-sensitivity issues for database names in cross-platform deployments were eliminated, and the security boundary for Ops became tightly controlled.

Specification 3: Collection Naming

- **Core Actions:**

It is recommended to use a collection name with the `t_` prefix + lowercase letters + underscores, following the "module_entity" format.

- **Naming Conventions and Examples:**

Rule	Requirement	Correct Example	Incorrect Example
Prefix	Start with `t_`	`t_order_detail`	`OrderDetail`
Format	Module_Entity	`t_user_address`	`t_user-address`
Disable	Not starting with `system.`	`t_system_config`	`system.config`
Collection Sharding	Time suffix	`t_log_202403`	`t_log$202403`

- **Business Application:**

  A project used `system.orders` as the order collection name. Because `system.` is a reserved prefix in MongoDB, some underlying management operations mistakenly identified it as a system internal collection and skipped it directly, ultimately causing incomplete auto-backup data. The issue was resolved after it was renamed to `t_orders`.

Specification 4: Field Naming

- **Core Actions**:

  It is recommended to uniformly adopt camelCase (lower camel case) or snake_case (snake case) for naming fields. Field design should be "self-explanatory" and avoid excessive description. The use of meaningless abbreviations is strictly prohibited.

- **Field Naming Comparison**:

```
// Recommended: Clear semantics and consistent style
{
    "_id": ObjectId("..."),
    "userName": "Zhang San",           // camelCase style
    "createTime": ISODate("..."),
    "orderItems": [...],
    "totalAmount": 199.00
}

// Not recommended: Confusing naming
```

```
{
    "_id": ObjectId("..."),
    "UN": "Zhang San",                  // Unclear abbreviation
    "Create_Time": ISODate("..."), // Mixed style
    "oi": [...],                    // Unclear meaning
    "_total": 199.00                // Business fields starting with an
underscore are prone to conflict with system fields.
}
```

- **Business Application**:

  A team had inconsistent field naming. Within the same collection, four different notations — `createTime`, `Create_Time`, `create_time`, and `CT` — were used to represent the creation time. Developers frequently misspelled field names, resulting in empty query results and increasing troubleshooting time from minutes to hours. After the naming convention was standardized, development efficiency improved significantly.

Document Design Specifications

Specification 5: Controlling Single Document Size

- **Core Action**: The size of a single document is recommended to be kept within 100KB and not exceed the 16MB limit. The larger the document volume, the more significant the impact on read/write performance, memory usage, and network transmission.
- **Large Document Processing Policy**:

Scenario	Solution	Example
Excessive Array Elements	Split into multiple documents	User posts: one document per post
Storing Large Files	Using GridFS	Images, videos, and large logs
Large Text Content	Compression at the business layer	Storing HTML content after compression
Oversized Files	Using COS + URL referencing	Store files in COS and store URLs in MongoDB.

- **Method for Detecting Document Size**:

```
// View the size of a single document
Object.bsonsize(db.collection.findOne({ _id: xxx }))
```

```
// View the average document size of a collection (unit: bytes)
db.collection.stats().avgObjSize
```

- **Business Application**:

  A social media platform stored all user posts in the `posts` array of each user's document. After active users published thousands of posts, the document size exceeded 16MB, preventing new posts from being written and causing users to complain about posting failures. The issue was completely resolved by changing to a design where each post is stored as a separate document linked by the user ID.

Specification 6: Controlling Nesting Levels

- **Core Actions**:

  It is recommended to keep the nesting level of documents within 3–5 layers to avoid overly deep nesting logic.

```
// Recommendation: Maintain a moderate nesting level (2-3 layers)
{
    "_id": ObjectId("..."),
    "orderId": "ORD202403001",
    "customer": {                    // Layer 1
        "name": "Zhang San",
        "contact": {                 // Layer 2
            "phone": "13800138000",
            "email": "zhangsan@example.com"
        }
    },
    "items": [{ "productId": "P001", "quantity": 2 }]  // Layer 1
(array)
}

// Not recommended: Excessively deep nesting
{
    "level1": {
        "level2": {
            "level3": {
                "level4": {
                    "level5": {
                        "data": "Too deep to be maintained"
                    }
```

```
                    }
                }
            }
        }
    }
```

- **Business Application**:

  A configuration system was designed with an 8-layer nested configuration structure. Modifying the innermost configuration requires constructing a complex `$set` path, such as `"a.b.c.d.e.f.g.value"`. Developers frequently wrote incorrect paths, causing configuration updates to fail and preventing the creation of effective indexes for deep fields. After the structure was flattened through refactoring, both configuration updates and queries became simple and efficient.

Specification 7: Embedded vs. Referenced Design

- **Core Actions**:

  Prioritize embedded design. Use a referenced design only in necessary scenarios, such as when large data volumes are involved or when frequent independent updates are required.
- **Design Decision** collection:

Consideration Factor	Prefer [Embedded]	Prefer [Referenced]
Read Mode	Data is always read together.	Data is frequently read individually.
Total Data Volume	The child data volume is small and its scale is limited.	Large child data volume or an unlimited growth trend.
Update Frequency	Child data is rarely updated independently.	Child data is frequently updated independently.
Relationship Type	One-to-one, one-to-few	One-to-many, many-to-many
Sharing	Child data belongs exclusively to a single parent document.	Child data is frequently shared by multiple documents.

- **Embedded Design Example**:

```
// Order + Order Items: Embedded (Always queried together; order items
do not exist independently)
{
    "_id": ObjectId("..."),
```

```
    "orderId": "ORD202403001",
    "customerId": "C001",
    "items": [
        { "productId": "P001", "name": "Product A", "quantity": 2,
"price": 99.00 },
        { "productId": "P002", "name": "Product B", "quantity": 1,
"price": 199.00 }
    ],
    "totalAmount": 397.00,
    "status": "paid",
    "createTime": ISODate("2024-03-15T10:30:00Z")
}
```

- **Referenced Design Example**:

```
// User + Article: Referenced (Articles are frequently queried
independently and their number grows indefinitely)
// User Document
{
    "_id": ObjectId("user_001"),
    "userName": "Zhang San",
    "email": "zhangsan@example.com"
}

// Article Document (referencing the user via authorId)
{
    "_id": ObjectId("article_001"),
    "title": "MongoDB Best Practices",
    "authorId": ObjectId("user_001"),  // referencing the user
    "content": "...",
    "createTime": ISODate("...")
}
```

- **Hybrid Mode** redundantly embeds frequently accessed sub-data while preserving reference relationships.

```
// ✅ Hybrid Mode: Redundantly embeds product names and prices
(snapshots) in orders while preserving productId references.
{
    "orderId": "ORD001",
```

```
    "items": [
        {
            "productId": ObjectId("..."),  // reference (used to
associate with the latest product information)
            "name": "Product A",               // redundant (snapshot
at order placement to prevent product renaming from affecting
historical orders)
            "price": NumberDecimal("99.00") // redundant (price
snapshot at order placement)
        }
    ]
}
```

- **Business Application:**

  An e-commerce system separates orders and order items into two collections (relational thinking). Querying order details requires first querying the order, then querying the order items, and finally performing application-layer joins. During major promotions, the API latency surged from 50ms to 800ms. After an embedded design was adopted (embedding order items within the order document), a single query returned complete data and latency dropped to 30 ms, and the code is also significantly simplified.

Specification 8: Considerations for Array Design

- **Core Actions**:

  It is recommended to keep the number of elements in a single array under 1000. Designing arrays that grow indefinitely is strictly prohibited.

- **Example: Infinitely Growing Array vs. Independent Document Association**:

```
// Not recommended: unbounded, infinitely growing arrays

{
    "userId": "user_10001",
    "orders": [
        { "orderId": "ORD_001", "amount": 99.00, "date":
ISODate("...") },
        { "orderId": "ORD_002", "amount": 158.00, "date":
ISODate("...") },
        // ... Active users may accumulate tens of thousands of
orders, triggering the 16MB limit.
    ]
}
```

```
// Recommended: Split array elements into separate documents and
associate them via foreign keys.
// User Document
{
    "userId": "user_10001",
    "name": "Zhang San",
    "orderCount": 1024
}

// Order Document (Independent Collection)
{
    "orderId": "ORD_001",
    "userId": "user_10001",   // foreign key association
    "amount": 99.00,
    "date": ISODate("2024-03-15T10:30:00Z")
}
```

- **Business Application**:

  An IoT platform stores all sensor readings in the `readings` array of a device document. After the platform had been running for a year, the array for an active device contained hundreds of thousands of readings, and the document exceeded 16MB, preventing new data from being written. After switching to the "bucket pattern" (one document per hour), the size of a single document stabilized below 100KB.

> **Note:**
> - For scenarios where only the most recent N records need to be retained (for example, only the last 10 login records), it is recommended to use the $push operator with the $slice modifier during write operations. This automatically maintains a fixed upper limit for the array length at the database level, avoiding multiple reads and truncations at the application layer.
> - For time-series data such as IoT or monitoring data, MongoDB 5.0+ has introduced native Time Series Collections. You should prioritize using MongoDB's native Time Series collections over manually implementing a bucket pattern, as this can achieve higher compression rates and better query performance.

Data Type Specifications

Specification 9: Selecting the Correct Data Type

- **Core Actions**:

Use the Date type for dates, Decimal128 for monetary amounts, and ObjectId or an incrementing Long for IDs.

- **Data Type Selection Collection:**

Scenario	Recommended Type	Not Recommended Type	Potential Issue
Date and time	Date	String	Cannot use native date operations and range query optimizations.
Financial amount	Decimal128	Double	Loss of floating-point precision, leading to discrepancies in financial reconciliation.
Document primary key	ObjectId (default)	Random string.	Non-incrementing random IDs cause frequent page splits, severely slowing write performance. The first 4 bytes of an ObjectId are a second-level timestamp, providing roughly incrementing characteristics. This causes B-Tree index writes to be concentrated at the tail, avoiding page splits caused by random insertion.
Large integer ID	NumberLong	String	Cannot perform numerical comparisons and range sorting.
Status flag	String (enumeration value)	Numeric	Magic Numbers have unclear meanings, making maintenance difficult later.

- **Data Type Usage Example:**

```
// Correct Type Usage
{
    "_id": ObjectId("65f3a2b8c1d2e3f4a5b6c7d8"),  // ObjectId
    "orderId": NumberLong("20240315000001"),       // Large integer
    "amount": NumberDecimal("199.99"),             // Use Decimal128
for monetary amounts
    "createTime": ISODate("2024-03-15T10:30:00Z"), // Use Date for
dates
    "status": "paid"                               // Use string
enumeration for status
}
```

```
// Incorrect Type Usage
{
    "_id": "random-uuid-string",          // Random strings impact
performance
    "orderId": "20240315000001",          // Strings cannot be sorted
numerically
    "amount": 199.99,                     // Double has precision
issues
    "createTime": "2024-03-15 10:30:00",  // Strings cannot be used
for date arithmetic
    "status": 1                           // Numeric meaning is
unclear
}
```

- **Business Application**:

  A financial system stored monetary amounts using the Double type. Calculating `0.1 + 0.2` yielded `0.30000000000000004`, and after cumulative calculations, the discrepancy with bank reconciliation amounted to hundreds of CNY. After switching to Decimal128, calculations became precise to the cent, and reconciliation was completely accurate.

Specification 10: _id Field Usage Specifications

- **Core Actions**:

  Unless there are specific requirements, use the default ObjectId as the _id. If business needs require a custom _id, it is recommended that it have an incremental characteristic.
- **Example:**

```
// Recommendation: Use the default ObjectId
{ "_id": ObjectId("65f3a2b8c1d2e3f4a5b6c7d8") }

// Acceptable: Custom incremental ID (must ensure the incremental
characteristic)
{ "_id": NumberLong("20240315000001") }

// Prohibited: Random strings (impact write performance)
{ "_id": "550e8400-e29b-41d4-a716-446655440000" }
```

- **Business Application**:

A system used random UUIDs as primary keys. As data volume grew, random writes to the index tree caused a sharp increase in disk I/O and triggered frequent page splits, reducing write QPS from 10,000 to 3,000. After the switch was made to monotonically increasing ObjectIds, the I/O bottleneck was eliminated by leveraging the sequential append characteristic, and performance returned to normal.

Schema Validation

Specification 11: Configuring Schema Validation for Core Collections

- **Core Actions**:

  Use the JSON Schema validation feature provided by MongoDB to ensure type and format consistency for data writes at the database engine level.

  > **Note:**
  > - `validationLevel: "moderate"` mode: Validates only newly written and updated documents, not existing ones (suitable for legacy data migration scenarios).
  > - Schema validation has a certain impact on **write performance** (typically < 5%). Collections with high write frequency need to be evaluated.
  > - Use the `collMod` command to modify the validation rules of an existing collection.

- **Schema Validation Example**:

```
// Create a collection with validation rules
db.createCollection("t_users", {
    validator: {
        $jsonSchema: {
            bsonType: "object",
            required: ["userName", "email", "createTime"],
            properties: {
                userName: {
                    bsonType: "string",
                    minLength: 2,
                    maxLength: 50,
                    description: "Username, required, 2-50 characters"
                },
                email: {
                    bsonType: "string",
                    pattern: "^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\\.[a-
zA-Z]{2,}$",
```

```
                    description: "Email, required, must conform to
email format"
                },
                age: {
                    bsonType: "int",
                    minimum: 0,
                    maximum: 150,
                    description: "Age, optional, an integer from 0 to
150"
                },
                status: {
                    enum: ["active", "inactive", "deleted"],
                    description: "Status, enumeration value"
                },
                createTime: {
                    bsonType: "date",
                    description: "Creation time, required"
                }
            }
        }
    },
    validationLevel: "strict",    // strict: validates all writes
    validationAction: "error"     // error: rejects writes if
validation fails
});
```

- **Business Application**:

  In an e-commerce system, the `price` field lacked type constraints. Some entries stored numbers like `99.00`, some stored strings like `"99.00"`, and some even stored objects like `{value: 99}`. Price sorting and comparison became completely chaotic, and the "spend 100, get 20 off" promotion logic failed. After Schema validation was added, dirty data was rejected for writing. After the existing data was cleaned, the feature returned to normal.

Capacity Planning

Specification 12: Controlling the Number of Collections

- **Core Actions**:

  It is recommended to keep the total number of collections per instance under 5000.
- **Impact of Excessive Number of Collections**:

Impact	Description
Long instance startup time	During engine startup, the metadata information of all collections needs to be loaded one by one.
High memory consumption	The metadata of each collection persistently resides in the cache, occupying business memory.
File handle consumption	Each collection corresponds to multiple underlying data files, making it easy to hit the system limit.
Complex Ops	The time required for routine operations such as status monitoring, data migration, and major version upgrades will increase exponentially.
Backup timeout or failure	Traversing massive metadata can easily cause physical backup tasks to time out severely, and may even completely fail to generate a physical backup.

- **Business Application**:

  An IoT platform initially adopted a "one device, one collection" model. As the business grew, over 50,000 collections were generated in a single database. The massive metadata caused a sharp increase in memory overhead, and the instance restart time deteriorated from seconds to tens of minutes. Backup tasks severely timed out or even terminated directly due to the need to traverse the extensive metadata, preventing the completion of full backups. The practice of sharding by device was abandoned, and the data was consolidated into a single Time Series Collection. A compound index was created using deviceId + timestamp. After optimization, the number of collections was reduced to single digits, and both startup and backup operations returned to normal.

Data Modeling Checklist

To ensure compliance with the specifications, it is recommended to check against the following checklist item by item during the project launch review phase:

Check Item	Verification Method	Passing Criteria
1. Naming Convention Consistency	Review all involved database/collection/field naming code.	Fully comply with the naming and prefix rules in this document.
2. Document Size Controllability	Sample and execute Object.bsonsize(doc).	Core document size is recommended to be controlled within 1 MB.
3. Reasonable Nesting Depth	Review the JSON structure tree of core documents.	Maximum nesting depth ⩽ 3–5 layers

4. Avoiding Unbounded Arrays	Thoroughly review data write and append logic.	Arrays have a clear business upper limit, or have adopted a bucketing pattern/$slice truncation mechanism.
5. Data Type Strictness	Review entity class field type definitions.	Dates are recommended to use Date, and monetary amounts are recommended to use Decimal128.
6. Enabling Schema Validation	Check collection creation scripts or validator configurations.	Constraints have been fully configured for the required fields, field types, and format validations of core collections.
7. Recommended Maximum of 5000 Collections per Instance.	Estimate and execute show collections statistics.	Estimated/Actual Number of Business Collections per Database ⩽ 100