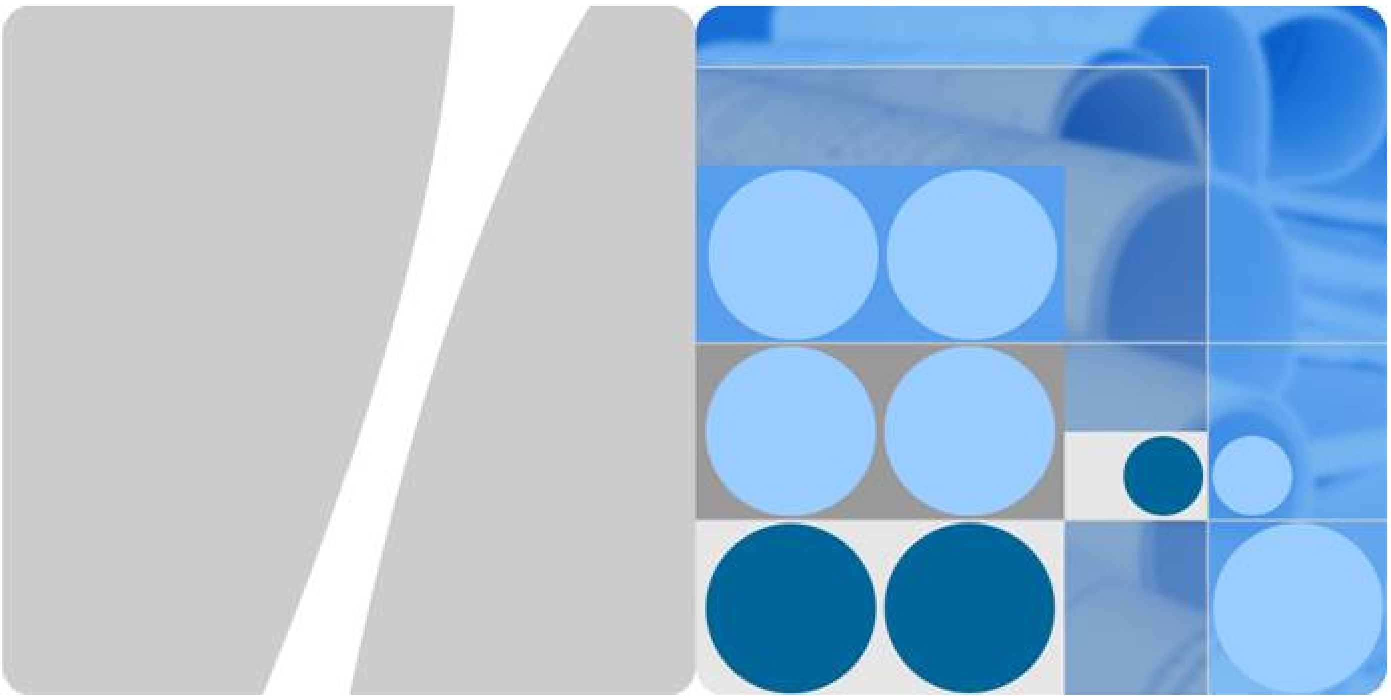

Distributed Cache Service

Best Practices

Date **2024-09-30**

Contents

1 Applications

1.1 Serializing Access to Frequently Accessed Resources

Overview

Application Scenario

In monolithic deployment, you can use Java concurrency APIs such as **ReentrantLock** or **synchronized** to implement mutual exclusion locks. This native lock mechanism provided by Java ensures that multiple threads within a Java VM process are executed concurrently and sequentially.

However, this mechanism may fail in multi-node deployment because a node's lock only takes effect on threads in the Java VM where the node runs. For example, the concurrency level in Internet seckills requires multiple nodes to run at the same time. Assume that requests of two users arrive simultaneously on two nodes. Although the requests can be processed simultaneously on the respective nodes, an inventory oversold problem may still occur because the nodes use different locks.

Solution

To serialize access to resources, ensure that all nodes use the same lock. This requires a distributed lock.

The idea of a distributed lock is to provide a globally unique "thing" for different systems to allocate locks. When a system needs a lock, it asks the "thing" for a lock. In this way, different systems can obtain the same lock.

Currently, a distributed lock can be implemented using cache databases, disk databases, or ZooKeeper.

Implementing distributed locks using DCS Redis instances has the following advantages:

- Simple operation: Locks can be acquired and released by using simple commands such as **SET**, **GET**, and **DEL**.
- High performance: Cache databases deliver higher read/write performance than disk databases and ZooKeeper.

- High reliability: DCS supports both master/standby and cluster instances, preventing single points of failure.

Implementing locks on distributed applications can avoid inventory oversold problems and nonsequential access. The following describes how to implement locks on distributed applications with Redis.

Prerequisites

- A DCS instance has been created, and is in the **Running** state.
- The network between the client server and the DCS instance is connected:
  - When the client and the DCS Redis instance are in the same VPC:

    By default, networks in a VPC can communicate with each other.
  - When the client and the DCS Redis instance are in different VPCs in the same region:

    If the client and DCS Redis instance are not in the same VPC, connect them by establishing a VPC peering connection. For details, see "Does DCS Support Cross-VPC Access?" in *Distributed Cache Service User Guide* > FAQs.
  - To access a Redis instance of another region on a client

    If the client server and the Redis instance are not in the same region, connect the network using Direct Connect. For details, see *Direct Connect User Guide*.
- You have installed **JDK1.8** (or later) and a development tool (**Eclipse** is used as an example) on the client server, and downloaded the **Jedis client**.

  The development tools and clients mentioned in this document are for example only.

Procedure

Step 1 Run Eclipse on the server and create a Java project. Then, create a distributed lock implementation class **DistributedLock.java** and a test class **CaseTest.java** for the example code, and reference the Jedis client as a library to the project.

Sample code of **DistributedLock.java**:

```
package dcsDemo01;

import java.util.UUID;

import redis.clients.jedis.Jedis;
import redis.clients.jedis.params.SetParams;

public class DistributedLock {
    // Address and port for connecting to the Redis instance. Replace them with the actual values.
    private final String host = "192.168.0.220";
    private final int port = 6379;

    private static final String SUCCESS = "OK";

    public  DistributedLock(){}

    /*
     * @param lockName      Lock name
     * @param timeout       Timeout for acquiring locks
     * @param lockTimeout   Validity period of locks
     * @return              Lock ID
     */
    public String getLockWithTimeout(String lockName, long timeout, long lockTimeout) {
        String ret = null;
        Jedis jedisClient = new Jedis(host, port);

        try {
            // Password for connecting to the Redis instance. Replace it with the actual value.
            String authMsg = jedisClient.auth("passwd");
            if (!SUCCESS.equals(authMsg)) {
                System.out.println("AUTH FAILED: " + authMsg);
            }

            String identifier = UUID.randomUUID().toString();
            String lockKey = "DLock:" + lockName;
            long end = System.currentTimeMillis() + timeout;

            SetParams setParams = new SetParams();
            setParams.nx().px(lockTimeout);

            while(System.currentTimeMillis() < end) {
                String result = jedisClient.set(lockKey, identifier, setParams);
                if(SUCCESS.equals(result)) {
                    ret = identifier;
                    break;
                }

                try {
                    Thread.sleep(2);
                } catch (InterruptedException e) {
                    Thread.currentThread().interrupt();
                }
            }
        }
        catch (Exception e) {
            e.printStackTrace();
        }finally {
            jedisClient.quit();
            jedisClient.close();
        }

        return ret;
    }

    /*
     * @param lockName         Lock name
     * @param identifier    Lock ID
     */
    public void releaseLock(String lockName, String identifier) {
        Jedis jedisClient = new Jedis(host, port);

        try {
            String authMsg = jedisClient.auth("passwd");
            if (!SUCCESS.equals(authMsg)) {
                System.out.println("AUTH FAILED: " + authMsg);
            }

            String lockKey = "DLock:" + lockName;
            if(identifier.equals(jedisClient.get(lockKey))) {
                jedisClient.del(lockKey);
            }
        }
        catch (Exception e) {
            e.printStackTrace();
        }finally {
            jedisClient.quit();
            jedisClient.close();
        }
    }
}
```

NOTICE

The code only shows how DCS implements access control using locks. During actual implementation, deadlock and lock check also need to be considered.

Assume that 20 threads are used to seckill ten Mate 10 mobile phones. The content of the test class **CaseTest.java** is as follows:

```
package dcsDemo01;
import java.util.UUID;

public class CaseTest {
    public static void main(String[] args) {
        ServiceOrder service = new ServiceOrder();
        for (int i = 0; i < 20; i++) {
            ThreadBuy client = new ThreadBuy(service);
            client.start();
        }
    }
}

class ServiceOrder {
    private final int MAX = 10;

    DistributedLock DLock = new DistributedLock();

    int n = 10;

    public void handleOder() {
        String userName = UUID.randomUUID().toString().substring(0,8) + Thread.currentThread().getName();
        String identifier = DLock.getLockWithTimeout("Mate 10", 10000, 2000);
        System.out.println("Processing order for user " + userName + "");
        if(n > 0) {
            int num = MAX - n + 1;
            System.out.println("User "+ userName + " is allocated number " + num + " mobile phone. Number
of mobile phones left: " + (--n) + "");
        }else {
            System.out.println("User "+ userName + " order failed.");
        }
        DLock.releaseLock("Mate 10", identifier);
    }
}

class ThreadBuy extends Thread {
    private ServiceOrder service;

    public ThreadBuy(ServiceOrder service) {
        this.service = service;
    }

    @Override
    public void run() {
        service.handleOder();
    }
}
```

Step 2 Configure the connection address, port number, and password of the DCS instance in the example code file **DistributedLock.java**.

In **DistributedLock.java**, set **host** and **port** to the connection address and port number of the instance. In the **getLockWithTimeout** and **releaseLock** methods, set **passwd** to the instance access password.

Step 3 Comment out the lock part in the test class **CaseTest**. The following is an example:

```
//The lock code is commented out in the test class:
public void handleOder() {
    String userName = UUID.randomUUID().toString().substring(0,8) + Thread.currentThread().getName();
    //Lock code
    //String identifier = DLock.getLockWithTimeout("Mate 10", 10000, 2000);
    System.out.println("Processing order for user " + userName + "");
    if(n > 0) {
        int num = MAX - n + 1;
        System.out.println("User "+ userName + " is allocated number " + num + " mobile phone. Number of 
mobile phones left: " + (--n) + "")
    }else {
        System.out.println("User "+ userName + " order failed.");
    }
    //Lock code
    //DLock.releaseLock("Mate 10", identifier);
}
```

Step 4 Compile and run a lock-free class. The purchases are disordered, as shown in the following:

```
Processing order for user e04934ddThread-5
Processing order for user a4554180Thread-0
User a4554180Thread-0 is allocated number 2 mobile phone. Number of mobile phones left: 8.
Processing order for user b58eb811Thread-10
User b58eb811Thread-10 is allocated number 3 mobile phone. Number of mobile phones left: 7.
Processing order for user e8391c0eThread-19
Processing order for user 21fd133aThread-13
Processing order for user 1dd04ff4Thread-6
User 1dd04ff4Thread-6 is allocated number 6 mobile phone. Number of mobile phones left: 4.
Processing order for user e5977112Thread-3
Processing order for user 4d7a8a2bThread-4
User e5977112Thread-3 is allocated number 7 mobile phone. Number of mobile phones left: 3.
Processing order for user 18967410Thread-15
User 18967410Thread-15 is allocated number 9 mobile phone. Number of mobile phones left: 1.
Processing order for user e4f51568Thread-14
User 21fd133aThread-13 is allocated number 5 mobile phone. Number of mobile phones left: 5.
User e8391c0eThread-19 is allocated number 4 mobile phone. Number of mobile phones left: 6.
Processing order for user d895d3f1Thread-12
User d895d3f1Thread-12 order failed.
Processing order for user 7b8d2526Thread-11
User 7b8d2526Thread-11 order failed.
Processing order for user d7ca1779Thread-8
User d7ca1779Thread-8 order failed.
Processing order for user 74fca0ecThread-1
User 74fca0ecThread-1 order failed.
User e04934ddThread-5 is allocated number 1 mobile phone. Number of mobile phones left: 9.
User e4f51568Thread-14 is allocated number 10 mobile phone. Number of mobile phones left: 0.
Processing order for user aae76a83Thread-7
User aae76a83Thread-7 order failed.
Processing order for user c638d2cfThread-2
User c638d2cfThread-2 order failed.
Processing order for user 2de29a4eThread-17
User 2de29a4eThread-17 order failed.
Processing order for user 40a46ba0Thread-18
User 40a46ba0Thread-18 order failed.
Processing order for user 211fd9c7Thread-9
User 211fd9c7Thread-9 order failed.
Processing order for user 911b83fcThread-16
User 911b83fcThread-16 order failed.
User 4d7a8a2bThread-4 is allocated number 8 mobile phone. Number of mobile phones left: 2.
```

Step 5 Add the lock code back to **CaseTest**, and compile and run the code. The following shows sequential purchases:

```
Processing order for user eee56fb7Thread-16
User eee56fb7Thread-16 is allocated number 1 mobile phone. Number of mobile phones left: 9.
Processing order for user d6521816Thread-2
User d6521816Thread-2 is allocated number 2 mobile phone. Number of mobile phones left: 8.
Processing order for user d7b3b983Thread-19
User d7b3b983Thread-19 is allocated number 3 mobile phone. Number of mobile phones left: 7.
```

```
Processing order for user 36a6b97aThread-15
User 36a6b97aThread-15 is allocated number 4 mobile phone. Number of mobile phones left: 6.
Processing order for user 9a973456Thread-1
User 9a973456Thread-1 is allocated number 5 mobile phone. Number of mobile phones left: 5.
Processing order for user 03f1de9aThread-14
User 03f1de9aThread-14 is allocated number 6 mobile phone. Number of mobile phones left: 4.
Processing order for user 2c315ee6Thread-11
User 2c315ee6Thread-11 is allocated number 7 mobile phone. Number of mobile phones left: 3.
Processing order for user 2b03b7c0Thread-12
User 2b03b7c0Thread-12 is allocated number 8 mobile phone. Number of mobile phones left: 2.
Processing order for user 75f25749Thread-0
User 75f25749Thread-0 is allocated number 9 mobile phone. Number of mobile phones left: 1.
Processing order for user 26c71db5Thread-18
User 26c71db5Thread-18 is allocated number 10 mobile phone. Number of mobile phones left: 0.
Processing order for user c32654dbThread-17
User c32654dbThread-17 order failed.
Processing order for user df94370aThread-7
User df94370aThread-7 order failed.
Processing order for user 0af94cddThread-5
User 0af94cddThread-5 order failed.
Processing order for user e52428a4Thread-13
User e52428a4Thread-13 order failed.
Processing order for user 46f91208Thread-10
User 46f91208Thread-10 order failed.
Processing order for user e0ca87bbThread-9
User e0ca87bbThread-9 order failed.
Processing order for user f385af9aThread-8
User f385af9aThread-8 order failed.
Processing order for user 46c5f498Thread-6
User 46c5f498Thread-6 order failed.
Processing order for user 935e0f50Thread-3
User 935e0f50Thread-3 order failed.
Processing order for user d3eaae29Thread-4
User d3eaae29Thread-4 order failed.
```

----End

1.2 Ranking with DCS

Overview

Ranking is a function commonly used on web pages and apps. It is implemented by listing key-values in descending order. However, a huge number of concurrent operation and query requests can result in a performance bottleneck, significantly increasing latency.

Ranking using DCS for Redis provides the following advantages:

- Data is stored in the memory, so read/write is fast.
- Multiple types of data structures, such as strings, lists, sets, and hashes are supported.

Prerequisites

- A DCS instance has been created, and is in the **Running** state.
- The network between the client server and the DCS instance is connected:
  - When the client and the DCS Redis instance are in the same VPC:

    By default, networks in a VPC can communicate with each other.
  - When the client and the DCS Redis instance are in different VPCs in the same region:

If the client and DCS Redis instance are not in the same VPC, connect them by establishing a VPC peering connection. For details, see "Does DCS Support Cross-VPC Access?" in *Distributed Cache Service User Guide* > FAQs.

  – To access a Redis instance of another region on a client

    If the client server and the Redis instance are not in the same region, connect the network using Direct Connect. For details, see *Direct Connect User Guide*.

- You have installed **JDK1.8** (or later) and a development tool (**Eclipse** is used as an example) on the client server, and downloaded the **Jedis client**.

  The development tools and clients mentioned in this document are for example only.

Procedure

Step 1 Run Eclipse on the server. Choose **File** > **New Project** to create a Java project named **dcsDemo02**.

Step 2 Choose **New** > **Class** to create a **productSalesRankDemo.java** file.

Step 3 Copy the following demo code to the **productSalesRankDemo.java** file.

```
package dcsDemo02;

import java.util.ArrayList;
import java.util.List;
import java.util.Set;
import java.util.UUID;

import redis.clients.jedis.Jedis;
import redis.clients.jedis.Tuple;

public class productSalesRankDemo {
    static final int PRODUCT_KINDS = 30;

    public static void main(String[] args) {
        // Address and port for connecting to the Redis instance. Replace them with the actual values.
        String host = "192.168.0.246";
        int port = 6379;

        Jedis jedisClient = new Jedis(host, port);

        try {
            // Password for connecting to the Redis instance. Replace it with the actual value.
            String authMsg = jedisClient.auth("******");
            if (!authMsg.equals("OK")) {
                System.out.println("AUTH FAILED: " + authMsg);
            }

            //Key
            String key = "Best-seller Rankings";

            jedisClient.del(key);

            //Generate product data at random
            List<String> productList = new ArrayList<>();
            for(int i = 0; i < PRODUCT_KINDS; i ++) {
                productList.add("product-" + UUID.randomUUID().toString());
            }

           //Generate sales volume at random
            for(int i = 0; i < productList.size(); i ++) {
                int sales = (int)(Math.random() * 20000);
                String product = productList.get(i);
               //Insert sales volume into Redis SortedSet
                jedisClient.zadd(key, sales, product);
            }

            System.out.println();
            System.out.println("                   "+key);

           //Obtain all lists and display the lists by sales volume
            Set<Tuple> sortedProductList = jedisClient.zrevrangeWithScores(key, 0, -1);
            for(Tuple product : sortedProductList) {
               System.out.println("Product ID: " + product.getElement() + ", Sales volume: "
                        + Double.valueOf(product.getScore()).intValue());
            }

            System.out.println();
            System.out.println("                   "+key);
            System.out.println("                   Top 5 Best-sellers");

            //Obtain the top 5 best-selling products and display the products by sales volume
            Set<Tuple> sortedTopList = jedisClient.zrevrangeWithScores(key, 0, 4);
            for(Tuple product : sortedTopList) {
               System.out.println("Product ID: " + product.getElement() + ", Sales volume: "
                        + Double.valueOf(product.getScore()).intValue());
            }
        }
        catch (Exception e) {
            e.printStackTrace();
        }
        finally {
            jedisClient.quit();
            jedisClient.close();
        }
    }

}
```

Step 4 Configure the connection address, port, and password for the DCS instance in the example code file.

Step 5 Compile and run the code.

----End

Operation Result

Compile and run the preceding Demo code. The operation result is as follows:

```
Best-seller Rankings
Product ID: product-b290c0d4-e919-4266-8eb5-7ab84b19862d, Sales volume: 18433
Product ID: product-e61a0642-d34f-46f4-a720-ee35940a5e7f, Sales volume: 18334
Product ID: product-ceeab7c3-69a7-4994-afc6-41b7bc463d44, Sales volume: 18196
Product ID: product-f2bdc549-8b3e-4db1-8cd4-a2ddef4f5d97, Sales volume: 17870
Product ID: product-f50ca2de-7fa4-45a3-bf32-23d34ac15a41, Sales volume: 17842
Product ID: product-d0c364e0-66ec-48a8-9ac9-4fb58adfd033, Sales volume: 17782
Product ID: product-5e406bbf-47c7-44a9-965e-e1e9b62ed1cc, Sales volume: 17093
Product ID: product-0c4d31ee-bb15-4c88-b319-a69f74e3c493, Sales volume: 16432
Product ID: product-a986e3a4-4023-4e00-8104-db97e459f958, Sales volume: 16380
Product ID: product-a3ac9738-bed2-4a9c-b96a-d8511ae7f03a, Sales volume: 15305
Product ID: product-6b8ad4b7-e134-480f-b3ae-3d35d242cb53, Sales volume: 14534
Product ID: product-26a9b41b-96b1-4de0-932b-f78d95d55b2d, Sales volume: 11417
Product ID: product-1f043255-a1f9-40a0-b48b-f40a81d07e0e, Sales volume: 10875
Product ID: product-c8fee24c-d601-4e0e-9d18-046a65e59835, Sales volume: 10521
Product ID: product-5869622b-1894-4702-b750-d76ff4b29163, Sales volume: 10271
Product ID: product-ff0317d2-d7be-4021-9d25-1f997d622768, Sales volume: 9909
Product ID: product-da254e81-6dec-4c76-928d-9a879a11ed8d, Sales volume: 9504
Product ID: product-fa976c02-b175-4e82-b53a-8c0df96fe877, Sales volume: 8630
Product ID: product-0624a180-4914-46b9-84d0-9dfbbdaa0da2, Sales volume: 8405
Product ID: product-d0079955-eaea-47b2-845f-5ff05a110a70, Sales volume: 7930
Product ID: product-a53145ef-1db9-4c4d-a029-9324e7f728fe, Sales volume: 7429
Product ID: product-9b1a1fd1-7c3b-4ae8-9fd3-ab6a0bf71cae, Sales volume: 5944
Product ID: product-cf894aee-c1cb-425e-a644-87ff06485eb7, Sales volume: 5252
Product ID: product-8bd78ba8-f2c4-4e5e-b393-60aa738eceae, Sales volume: 4903
Product ID: product-89b64402-c624-4cf1-8532-ae1b4ec4cabc, Sales volume: 4527
Product ID: product-98b85168-9226-43d9-b3cf-ef84e1c3d75f, Sales volume: 3095
Product ID: product-0dda314f-22a7-464b-ab8c-2f8f00823a39, Sales volume: 2425
Product ID: product-de7eb085-9435-4924-b6fa-9e9fe552d5a7, Sales volume: 1694
Product ID: product-9beadc07-aab0-438c-ac5e-bcc72b9d9c36, Sales volume: 1135
Product ID: product-43834316-4aca-4fb2-8d2d-c768513015c5, Sales volume: 256

            Best-seller Rankings
            Top 5 Best-sellers
Product ID: product-b290c0d4-e919-4266-8eb5-7ab84b19862d, Sales volume: 18433
Product ID: product-e61a0642-d34f-46f4-a720-ee35940a5e7f, Sales volume: 18334
Product ID: product-ceeab7c3-69a7-4994-afc6-41b7bc463d44, Sales volume: 18196
Product ID: product-f2bdc549-8b3e-4db1-8cd4-a2ddef4f5d97, Sales volume: 17870
Product ID: product-f50ca2de-7fa4-45a3-bf32-23d34ac15a41, Sales volume: 17842
```

2 Connections

2.1 Configuring Redis Client Retry

Importance of Retry

Both the client and server may encounter temporary faults (such as transient network or disk jitter, service unavailability, or invoking timeout, due to infrastructure or running environment reasons). As a result, Redis operations may fail. You can design automated retry mechanisms to reduce the impact of such faults and ensure successful execution.

Scenarios Where Redis Operations Fail

Scenario	Description
Master/standby switchover triggered by a fault	If the master node is faulty due to Redis underlying hardware or other reasons, a master/standby switchover is triggered to ensure that the instance is still available. A master/standby switchover has the following impacts: ● Instance disconnection down to seconds ● Read-only for 30s at most
Read-only during specification modification	During specification modification, the instance may be disconnected for seconds and read-only for minutes. For more information about the impact of specification modification, see section "Modifying Specifications" in the *Distributed Cache Service User Guide*.
Request blockage caused by slow queries	Operations whose time complexity is O(N) cause slow queries and request blockage. In this case, other client requests may temporarily fail.
Complex network environment	Due to the complex network environment between the client and the Redis server, network jitter, packet loss, and data retransmission may occur occasionally. In this case, client requests may temporarily fail.

Scenario	Description
Complex hardware issues	Client requests may temporarily fail due to occasional hardware faults, such as VM HA and disk latency jitter.

Recommended Retry Rules

Retry Rule	Description
Retry only idempotent operations.	Timeout may occur in any of the following phases: ● A command is successfully sent by the client but has not reached Redis. ● The command has reached Redis, but the execution times out. ● Redis has executed the command, but the result returned to the client times out. A retried operation may be repeatedly executed in Redis. Therefore, not all operations are suitable to be retried. You are advised to retry only idempotent operations, such as running the **SET** command. For example, if you run the **SET a b** command multiple times, the value of **a** can only be **b** or the execution fails. If you run **LPUSH mylist a**, which is not idempotent, **mylist** may contain multiple **a** elements.
Configure proper retry times and interval.	Configure the retry times and interval based on service requirements in actual scenarios to prevent the following problems: ● If the number of retries is insufficient or the interval is too long, the application may fail to complete operations. ● If the number of retries is too large or the interval is too short, the application may occupy too many system resources and the server may be blocked due to too many requests. Common retry interval policies include immediate retry, fixed-interval retry, exponential backoff retry, and random backoff retry.
Avoid retry nesting.	Retry nesting may cause the retry interval to be exponentially amplified.
Record retry exceptions and print failure reports.	During retry, you can print retry error logs at the WARN level.

Jedis Client Retry Configurations

- Retries are not supported in native JedisPool mode (for single-node, master/standby, and Proxy Cluster instances). However, you can implement retries by referring to **JedisClusterCommand**.
- Retries are supported in JedisCluster mode. You can set the **maxAttempts** parameter to define the number of retry times when a failure occurs. The default value is **5**. By default, all JedisCluster operations invoke the retry method.

  Example code:

```
@Bean
JedisCluster jedisCluster() {
    Set<HostAndPort> hostAndPortsSet = new HashSet<>();
    hostAndPortsSet.add(new HostAndPort("{dcs_instance_address}", 6379));
    JedisPoolConfig jedisPoolConfig = new JedisPoolConfig();
    jedisPoolConfig.setMaxIdle(100);
    jedisPoolConfig.setMinIdle(1);
    jedisPoolConfig.setMaxTotal(1000);
    jedisPoolConfig.setMaxWaitMilis(2000);
    jedisPoolConfig.setMaxAttempts(5);
    return new JedisCluster(hostAndPortsSet, jedisPoolConfig);
}
```

Table 2-1 Recommended Jedis connection pool parameter settings

Parameter	Description	Recommended Setting
maxTotal	Maximum number of connections	Set this parameter based on the number of HTTP threads of the web container and reserved connections. Assume that the **maxConnections** parameter of the Tomcat Connector is set to **150** and each HTTP request may concurrently send two requests to Redis, you are advised to set this parameter to at least 400 (150 x 2 + 100). **Limit**: The value of **maxTotal** multiplied by the number of client nodes (CCE containers or service VMs) must be less than the maximum number of connections allowed for a single DCS Redis instance. For example, if **maxClients** of a master/standby DCS Redis instance is 10,000 and **maxTotal** of a single client is 500, the maximum number of clients is 20.
maxIdle	Maximum number of idle connections	Use the same configuration as **maxTotal**.

Parameter	Description	Recommended Setting
minIdle	Minimum number of idle connections	Generally, you are advised to set this parameter to 1/X of **maxTotal**. For example, the recommended value is **100**. In performance-sensitive scenarios, you can set this parameter to the value of **maxIdle** to prevent the impact caused by frequent connection quantity changes. For example, set this parameter to **400**.
maxWaitMillis	Maximum waiting time for obtaining a connection, in milliseconds	The recommended maximum waiting time for obtaining a connection from the connection pool is the maximum tolerable timeout of a single service minus the timeout for command execution. For example, if the maximum tolerable HTTP failure is 15s and the timeout of Redis requests is 10s, set this parameter to 5s.
timeout	Command execution timeout, in milliseconds	This parameter indicates the maximum timeout for running a Redis command. Set this parameter based on the service logic. You are advised to set this timeout to least 210 ms to ensure network fault tolerance. For special detection logic or environment exception detection, you can adjust this timeout to seconds.

Parameter	Description	Recommended Setting
minEvictableIdleTimeMil-lis	Idle connection eviction time, in milliseconds. If a connection is not used for a period longer than this, it will be released.	If you do not want the system to frequently re-establish disconnected connections, set this parameter to a large value (xx minutes) or set this parameter to **–1** and check idle connections periodically.
timeBetweenEviction-RunsMillis	Interval for detecting idle connections, in milliseconds	The value is estimated based on the number of idle connections in the system. For example, if this interval is set to 30s, the system detects connections every 30s. If an abnormal connection is detected within 30s, it will be removed. Set this parameter based on the number of connections. If the number of connections is too large and this interval is too short, request resources will be wasted. If there are hundreds of connections, you are advised to set this parameter to 30s. The value can be dynamically adjusted based on system requirements.
testOnBorrow	Indicates whether to check the connection validity using the **ping** command when borrowing connections from the resource pool. Invalid connections will be removed.	If your service is extremely sensitive to connections and the performance is acceptable, you can set this parameter to **True**. Generally, you are advised to set this parameter to **False** to enable idle connection detection.

Parameter	Description	Recommended Setting
testWhileIdle	Indicates whether to use the **ping** command to monitor the connection validity during idle resource monitoring. Invalid connections will be destroyed.	True
testOnReturn	Indicates whether to check the connection validity using the **ping** command when returning connections to the resource pool. Invalid connections will be removed.	False
maxAttempts	Number of connection retries when JedisCluster is used	Recommended value: 3–5. Default value: **5**. Set this parameter based on the maximum timeout intervals of service APIs and a single request. The maximum value is **10**. If the value exceeds **10**, the processing time of a single request is too long, blocking other requests.

3 Suggestions

3.1 DCS Usage

Service Usage

Principle	Description	Remarks
Separate hot data from cold data.	You can store frequently accessed data (hot data) in Redis, and infrequently accessed data (cold data) in databases such as MySQL and Elasticsearch.	Infrequently accessed data stored in the memory occupies Redis space and does not accelerate access.
Differentiate service data.	Store unrelated service data in different Redis instances.	This prevents services from affecting each other and prevents single instances from being too large. This also enables you to quickly restore services in case of faults.
	Do not use the **SELECT** command for multi-DB on a single instance.	Multi-DB on a single Redis instance does not provide good isolation and is no longer in active development by open-source Redis. You are advised not to depend on this feature in the future.

Principle	Description	Remarks
Set a proper eviction policy.	If the eviction policy is set properly, Redis can still function when the memory is used up unexpectedly.	You can that meets your service requirements.You can select a policy that meets your service requirements by configuring the **maxmemory-policy** parameter. For details, see section "Modifying Configuration Parameters" in the *Distributed Cache Service User Guide*.
Use Redis as cache.	Do not over-rely on Redis transactions.	After a transaction is executed, it cannot be rolled back.
	If data is abnormal, clear the cache for data restoration.	Redis does not have a mechanism or protocol to ensure strong data consistency. Therefore, services cannot over-rely on the accuracy of Redis data.
	When using Redis as cache, set expiration on all keys. Do not use Redis as a database.	Set expiration as required, but a longer expiration is not necessarily better.
Prevent cache breakdown.	Use Redis together with local cache. Store frequently used data in the local cache and regularly update it asynchronously.	-
Prevent cache penetration.	Non-critical path operations are passed through to the database. Limit the rate of access to the database.	-

Principle	Description	Remarks
Do not use Redis as a message queue.	In pub/sub scenarios, do not use Redis as a message queue.	● Unless otherwise required, you are not advised to use Redis as a message queue. ● Using Redis as a message queue causes capacity, network, performance, and function issues. ● If message queues are required, use Kafka for throughput and RocketMQ for reliability.
Select proper specifications.	If service growth causes increases in Redis requests, use Proxy Cluster or Redis Cluster instances.	Scaling up single-node and master/standby instances only expands the memory and bandwidth, but cannot enhance the computing capabilities.
	In production, do not use single-node instances. Use master/standby or cluster instances.	-
	Do not use large specifications for master/standby instances.	Redis forks a process when rewriting AOF or running the **BGSAVE** command. If the memory is too large, responses will be slow.
Prepare for degradation or disaster recovery.	When a cache miss occurs, data is obtained from the database. Alternatively, when a fault occurs, allow another Redis to take over services automatically.	-

Data Design

Category	Principle	Description	Remarks
Keys	Keep the format consistent.	Use the service name or database name as the prefix, followed by colons (:). Ensure that key names have clear meanings.	For example: *service name*:*sub-service name*:*ID*.
	Minimize the key length.	Minimize the key length without compromising clarity of the meaning. Abbreviate common words. For example, **user** can be abbreviated to **u**, and **messages** can be abbreviated to **msg**.	Use up to 128 bytes. The shorter the better.
	Do not use special characters except braces ({}).	Do not use special characters such as spaces, line brakes, single or double quotation marks, and other escape characters.	Redis uses braces ({}) to signify hash tags. Braces in key names must be used correctly to avoid unbalanced shards.
Values	Use appropriate value sizes.	Keep the value of a key within 10 KB.	Large values may cause unbalanced shards, hot keys, traffic or CPU usage surges, and scaling or migration failures. These problems can be avoided by proper design.

Category	Principle	Description	Remarks
	Use appropriate number of elements in each key.	Do not include too many elements in each Hash, Set, or List. It is recommended that each key contain up to 5000 elements.	Time complexity of some commands, such as **HGETALL**, is directly related to the quantity of elements in a key. If commands whose time complexity is O(N) or higher are frequently executed and a key has a large number of elements, there may be slow requests, unbalanced shards, or hot keys.
	Use appropriate data types.	This saves memory and bandwidth.	For example, to store multiple attributes of a user, you can use multiple keys, such as **set u:1:name "X"** and **set u:1:age 20**. To save memory usage, you can also use the **HMSET** command to set multiple fields to their respective values in the hash stored at one key.
	Set appropriate timeout.	Do not set a large number of keys to expire at the same time.	When setting key expiration, add or subtract a random offset from a base expiry time, to prevent a large number of keys from expiring at the same time. Otherwise, CPU usage will be high at the expiry time.

Command Usage

Principle	Description	Remarks
Exercise caution when using commands with time complexity of O(N).	Pay attention to the value of N for commands whose time complexity is O(N). If the value of N is too large, Redis will be blocked and the CPU usage will be high.	For example, the **HGETALL**, **LRANGE**, **SMEMBERS**, **ZRANGE**, and **SINTER** commands will consume a large number of CPU resources if there is a large number of elements. Alternatively, you can use **SCAN** sister commands, such as **HSCAN**, **SSCAN**, and **ZSCAN** commands.
Do not use high-risk commands.	Do not use high-risk commands such as **FLUSHALL**, **KEYS**, and **HGETALL**, or rename them.	For details, see section "Renaming Commands" in the *User Guide*.
Exercise caution when using the **SELECT** command.	Redis does not have a strong support for multi-DB. Redis is single-threaded, so databases interfere with each other. You are advised to use multiple Redis instances instead of using multi-DB on one instance.	-
Use batch operations to improve efficiency.	For batch operations, use the **MGET** command, **MSET** command, or pipelining to improve efficiency, but do not include a large number of elements in one batch operation.	**MGET** command, **MSET** command, and pipelining differ in the following ways: ● **MGET** and **MSET** are atomic operations, while pipelining is not. ● Pipelining can be used to send multiple commands at a time, while **MGET** and **MSET** cannot. ● Pipelining must be supported by both the server and the client.
Do not use time-consuming code in Lua scripts.	The timeout of Lua scripts is 5s, so avoid using long scripts.	Long scripts: time-consuming sleep statements or long loops.

Principle	Description	Remarks
Do not use random functions in Lua scripts.	When invoking a Lua script, do not use random functions to specify keys. Otherwise, the execution results will be inconsistent between the master and standby nodes, causing data inconsistency.	-
Follow the rules for using Lua on cluster instances.	Follow the rules for using Lua on cluster instances.	● When the **EVAL** or **EVALSHA** command is run, the command parameter must contain at least one key. Otherwise, the client displays the error message "ERR eval/ evalsha numkeys must be bigger than zero in redis cluster mode." ● When the **EVAL** or **EVALSHA** command is run, a cluster DCS Redis instance uses the first key to compute slots. Ensure that the keys to be operated are in the same slot.
Optimize multi-key operation commands such as **MGET** and **HMGET** with parallel processing and non-blocking I/O.	Some clients do not treat these commands differently. Keys in such a command are processed sequentially before their values are returned in a batch. This process is slow and can be optimized through pipelining.	For example, running the **MGET** command on a cluster using Lettuce is dozens of times faster than using Jedis, because Lettuce uses pipelining and non-blocking I/O while Jedis does not have a special plan itself. To use Jedis in such scenarios, you need to implement slot grouping and pipelining by yourself.

Principle	Description	Remarks
Do not use the **DEL** command to directly delete big keys.	Deleting big keys, especially Sets, using **DEL** blocks other requests.	In Redis 4.0 and later, you can use the **UNLINK** command to delete big keys safely. This command is non-blocking. In versions earlier than Redis 4.0: ● To delete big Hashes, use **HSCAN** + **HDEL** commands. ● To delete big Lists, use the **LTRIM** command. ● To delete big Sets, use **SSCAN** + **SREM** commands. ● To delete big Sorted Sets, use **ZSCAN** + **ZREM** commands.

SDK Usage

Principle	Description	Remarks
Use connection pools and persistent connections ("pconnect" in Redis terminology).	The performance of short connections ("connect" in Redis terminology) is poor. Use clients with connection pools.	Frequently connecting to and disconnecting from Redis will unnecessarily consume a lot of system resources and can cause host breakdown in extreme cases. Ensure that the Redis client connection pool is correctly configured.
The client must perform fault tolerance in case of faults or slow requests.	The client should have fault tolerance and retry mechanisms in case of master/standby switchover, command timeout, or slow requests caused by network fluctuation or configuration errors.	See **Configuring Redis Client Retry**.

Principle	Description	Remarks
Set appropriate interval and number of retries.	Do not set the retry interval too short or too long.	● If the retry interval is very short, for example, shorter than 200 milliseconds, a retry storm may occur, and can easily cause service avalanche. ● If the retry interval is very long or the number of retries is set to a large value, the service recovery may be slow in the case of a master/standby switchover.
Avoid using Lettuce.	Lettuce is the default client of Spring and stands out in terms of performance. However, Jedis is more stable because it is better at detecting and handling connection errors and network fluctuations. Therefore, Jedis is recommended.	Lettuce has the following problems: ● By default, Lettuce does not have cluster topology update configurations. When the cluster topology changes (for example after a master/standby switchover or scaling), new nodes cannot be identified, causing service failures. ● Lettuce cannot validate connections in the connection pool. If an invalid connection is used, services will fail.

O&M and Management

Principle	Description	Remarks
Use passwords in production.	In production systems, use passwords to protect Redis.	-
Ensure security on the live network.	Do not allow unauthorized developers to connect to redis-server in the production environment.	-

Principle	Description	Remarks
Verify the fault handling capability of the service.	Organize drills in the test environment or pre-production environment to verify service reliability in Redis master/ standby switchover, breakdown, or scaling scenarios.	Master/standby switchover can be triggered manually on the console. It is strongly recommended that you use Lettuce for these drills.
Configure monitoring.	Pay attention to the Redis capacity and expand it before overload.	Configure CPU, memory, and bandwidth alarms based on the alarm thresholds.