

# Distributed Cache Service

## Best Practices

**Issue** 01  
**Date** 2025-01-07



**Copyright © Huawei Cloud Computing Technologies Co., Ltd. 2025. All rights reserved.**

No part of this document may be reproduced or transmitted in any form or by any means without prior written consent of Huawei Cloud Computing Technologies Co., Ltd.

## **Trademarks and Permissions**



HUAWEI and other Huawei trademarks are the property of Huawei Technologies Co., Ltd.

All other trademarks and trade names mentioned in this document are the property of their respective holders.

## **Notice**

The purchased products, services and features are stipulated by the contract made between Huawei Cloud and the customer. All or part of the products, services and features described in this document may not be within the purchase scope or the usage scope. Unless otherwise specified in the contract, all statements, information, and recommendations in this document are provided "AS IS" without warranties, guarantees or representations of any kind, either express or implied.

The information in this document is subject to change without notice. Every effort has been made in the preparation of this document to ensure accuracy of the contents, but all statements, information, and recommendations in this document do not constitute a warranty of any kind, express or implied.

## **Huawei Cloud Computing Technologies Co., Ltd.**

Address: Huawei Cloud Data Center Jiaoxinggong Road  
Qianzhong Avenue  
Gui'an New District  
Gui Zhou 550029  
People's Republic of China

Website: <https://www.huaweicloud.com/intl/en-us/>

---

# Contents

---

<b>1 DCS Best Practices</b> .....	<b>1</b>
<b>2 Service Application</b> .....	<b>4</b>
2.1 Serializing Access to Frequently Accessed Resources.....	4
2.2 Ranking with DCS.....	9
2.3 Implementing Bullet-Screen and Social Comments with DCS.....	12
2.4 Merging Game Servers with DCS.....	17
2.5 Flashing E-commerce Sales with DCS.....	20
2.6 Reconstructing Application System Databases with DCS.....	23
2.7 Upgrading a Redis 3.0 Instance.....	26
<b>3 Network Connection</b> .....	<b>31</b>
3.1 Using Nginx for Public Access to DCS.....	31
3.2 Using SSH Tunneling for Public Access to DCS.....	36
3.3 Using ELB for Public Access to DCS.....	40
3.4 Connecting a Client to DCS Through CCE.....	44
3.5 Configuring Redis Client Retry.....	49
<b>4 Usage Guide</b> .....	<b>56</b>
4.1 DCS Data Security.....	56
4.2 Suggestions on Using DCS.....	60
4.3 Detecting and Handling Big Keys and Hot Keys.....	69
4.4 Configuring a Redis Pipeline.....	74
4.5 Optimizing the Jedis Connection Pool.....	78

# 1 DCS Best Practices

This section summarizes best practices of Distributed Cache Service (DCS) in common scenarios. Each practice is given a description and procedure.

**Table 1-1** DCS best practices

Best Practice	Description
<a href="#">Serializing Access to Frequently Accessed Resources</a>	Describes how to implement locks on distributed applications with Redis. Serializing access to hot resources with locks avoids oversold inventory or disordered access in flash sales.
<a href="#">Ranking with DCS</a>	Describes how to rank best-selling offerings using DCS for Redis.
<a href="#">Implementing Bullet-Screen and Social Comments with DCS</a>	Describes how to display a key-value list, such as streaming or social comments, in descending order from different dimensions using DCS for Redis.
<a href="#">Merging Game Servers with DCS</a>	Describes how to synchronize servers using Redis. During game server provisioning and merging, game developers must consider how to synchronize data among different servers. With the pub/sub message queuing mechanism of DCS for Redis, data changes on one game server can be published to Redis channels. Other game servers can subscribe to the channels to receive messages of changes.
<a href="#">Flashing E-commerce Sales with DCS</a>	Describes how to satisfy high concurrency in e-commerce flash sales using DCS for Redis as the database cache. Clients can access Redis to query inventories and place orders.
<a href="#">Reconstructing Application System Databases with DCS</a>	Describes how to migrate data, taking the example of migrating a table from a MySQL database to DCS for Redis.

Best Practice	Description
<a href="#">Upgrading a Redis 3.0 Instance</a>	Describes how to upgrade a DCS Redis 3.0 instance through data migration and IP switchover. You are advised to upgrade your DCS Redis 3.0 instances as soon as possible. DCS for Redis 4.0 and later are compatible with Redis 3.0.
<a href="#">Using Nginx for Public Access to DCS</a>	Huawei Cloud DCS Redis 4.0 and later cannot be bound with elastic IP addresses (EIPs) and cannot be accessed over public networks directly. This best practice describes how to access a single-node, master/standby, read/write splitting, or Proxy Cluster DCS Redis 4.0, 5.0, or 6.0 instance by using a jump server. This solution is unavailable for public access to Redis Cluster instances.
<a href="#">Using SSH Tunneling for Public Access to DCS</a>	Huawei Cloud DCS Redis 4.0 and later cannot be bound with EIPs and cannot be accessed over public networks directly. This best practice describes how to create an SSH tunnel as a proxy to connect your DCS instance and local computer to achieve proxy forwarding. In this way, single-node, master/standby, read/write splitting, and Proxy Cluster DCS Redis instances in a VPC can be accessed. This solution is unavailable for public access to Redis Cluster instances.
<a href="#">Using ELB for Public Access to DCS</a>	Huawei Cloud DCS Redis 4.0 and later cannot be bound with EIPs and cannot be accessed over public networks directly. This best practice describes how to access a single-node, master/standby, read/write splitting, or Proxy Cluster instance or a node in a Redis Cluster instance through public networks by enabling cross-VPC backend on a load balancer.
<a href="#">Connecting a Client to DCS Through CCE</a>	More and more applications are deployed in containers. This practice describes how to deploy Redis clients in cluster containers to access DCS through Cloud Container Engine (CCE).
<a href="#">Configuring Redis Client Retry</a>	This best practice describes the retry configuration of the Jedis client. A complete automatic retry mechanism can greatly reduce the impact of infrastructure or running environment faults.
<a href="#">DCS Data Security</a>	This best practice provides actionable guidance for enhancing the overall security of using DCS.

Best Practice	Description
<a href="#">Suggestions on Using DCS</a>	This best practice describes suggestions on using DCS for Redis in terms of the service, data design, commands, SDKs, and O&M management.
<a href="#">Detecting and Handling Big Keys and Hot Keys</a>	This best practice describes how to find and optimize big keys and hot keys when using DCS for Redis.
<a href="#">Configuring a Redis Pipeline</a>	This best practice describes how to use Redis pipelines. DCS for Redis supports the native pipelining.
<a href="#">Optimizing the Jedis Connection Pool</a>	JedisPool is the connection pool of the Jedis client. This best practice describes how to configure JedisPool for better Redis performance and resource utilization.

# 2 Service Application

---

## 2.1 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 seckill 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?](#)
  - 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 [What Is Direct Connect.](#)
  - For public access  
For details about how to access a DCS Redis 4.0/5.0/6.0 instance on a client over a public network, see [Using Nginx for Public Access to DCS](#) or [Using ELB for Public Access to DCS](#).
- 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 handleOrder() {
    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

## 2.2 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?](#)
  - 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 [What Is Direct Connect.](#)
  - For public access  
For details about how to access a DCS Redis 4.0/5.0/6.0 instance on a client over a public network, see [Using Nginx for Public Access to DCS](#) or [Using ELB for Public Access to DCS.](#)
- 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-b393-60aa738eacee, Sales volume: 4903
Product ID: product-89b64402-c624-4cf1-8532-ae1b4ec4cab, 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
```

## 2.3 Implementing Bullet-Screen and Social Comments with DCS

### Overview

#### Application Scenario

Scenarios such as bullet-screen comments in videos or live streaming and commenting and replying on a social website require high live efficiency and interactivity. A platform must ensure a very low latency to support such services. Comments are sorted by time in reverse order. If a relational database is adopted, the sorting efficiency becomes lower and lower as comments pile up.

#### Solution

Using DCS for Redis, a key-value list can be displayed in descending order from multiple dimensions. Take live commenting as an example. Bullet-screen comments can be ordered according to their weighted score calculated using their timestamp and then displayed as sorted sets (zsets). The content can be directly stored as values. Zset can also be applied to social websites. Since the quantity of comments and replies is huge, they require ordered display and local persistence. The primary key ID of a comment can be stored as a value, and the content of the comment is stored in the database and queried with the ID.

## 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?](#)
  - 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 [What Is Direct Connect.](#)
  - For public access  
For details about how to access a DCS Redis 4.0/5.0/6.0 instance on a client over a public network, see [Using Nginx for Public Access to DCS](#) or [Using ELB for Public Access to DCS.](#)
- 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, and import the Jedis client as a library to the project.

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

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

- Sample code of bullet-screen comments in live streaming

```
package org.example.task;

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 VideoBulletScreenDemo {

    static final int MESSAGE_NUM = 30;

    public static void main(String[] args) {

        // Address and port for connecting to the Redis instance. Replace them with the actual values.
        String host = "127.0.0.1";
        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);
}

String key = "Live comment list";

jedisClient.del(key);

// Randomly spawn bullets.
List<String> messageList = new ArrayList<>();
for (int i = 0; i < MESSAGE_NUM; i++){
    messageList.add("message-" + UUID.randomUUID().toString());
}

// Timestamp of random spawn.
for (int i = 0; i < messageList.size(); i++){
    String message = messageList.get(i);
    int sales = (int)(Math.random()*1000);
    long time = System.currentTimeMillis() + sales;
    // Insert as sorted set of Redis.
    jedisClient.zadd(key,time,message);
}

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

// Obtain all lists and output in chronological order.
Set<Tuple> sortedMessageList = jedisClient.zrangeWithScores(key, 0, -1);
for (Tuple message : sortedMessageList){
    System.out.println("bullets content: " + message.getElement() + ", sent time: " +
Double.valueOf(message.getScore()).longValue());
}

System.out.println();
System.out.println(" The latest 5 bullets");

Set<Tuple> sortedTopList = jedisClient.zrevrangeWithScores(key,0,4);
for (Tuple product : sortedTopList){
    System.out.println("bullets content: " + product.getElement() + ", sent time: " +
Double.valueOf(product.getScore()).longValue());
}
} catch (Exception e) {
    e.printStackTrace();
} finally {
    jedisClient.quit();
    jedisClient.close();
}
}
}
```

- Sample code of replying to a comment on a social website

```
package org.example.task;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Set;
import java.util.UUID;
import redis.clients.jedis.Jedis;
import redis.clients.jedis.Tuple;
public class SiteCommentsDemo {
    // Total comments and replies.
    static final int COMMENT_NUM = 20;
    public static void main(String[] args) {
        // Address and port for connecting to the Redis instance. Replace them with the actual values.
        String host = "127.0.0.1";
```

```
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);
    }
    String key = "List of replies to comments on a social website";
    jedisClient.del(key);
    HashMap<Integer, Comment> map = new HashMap<>();
    // Randomly spawn objects for comment replies.
    List<Comment> commentList = new ArrayList<>();
    for (int i = 0; i < COMMENT_NUM; i++){
        Comment comment = new Comment();
        comment.setId(i+1);
        comment.setContent(UUID.randomUUID().toString().substring(0,8));
        long time = System.currentTimeMillis();
        Thread.sleep(50);
        comment.setTime(time);
        // Randomly spawn replies.
        if (i > 0 && Math.random() < 0.5){
            comment.setParentId((int)(Math.random()*i) + 1);
        }
        commentList.add(comment);
        map.put(comment.getId(),comment);
        // Insert as sorted set of Redis.
        jedisClient.zadd(key,time,String.valueOf(comment.getId()));
    }
    System.out.println(" " + key);
    // Obtain all lists and output in chronological order.
    Set<Tuple> sortedCommentList = jedisClient.zrangeWithScores(key, 0, -1);
    for (Tuple comment : sortedCommentList){
        Integer commentId = Integer.valueOf(comment.getElement());
        Comment tmpComment = map.get(commentId);
        System.out.println("comment ID: " + comment.getElement() + " comment parent ID: " +
tmpComment.getParentId() + ", comment time: " +
Double.valueOf(comment.getScore()).longValue());
    }
    System.out.println();
    System.out.println(" The latest 5 replies");
    Set<Tuple> sortedTopList = jedisClient.zrevrangeWithScores(key,0,4);
    for (Tuple comment : sortedTopList){
        Integer commentId = Integer.valueOf(comment.getElement());
        Comment tmpComment = map.get(commentId);
        if (tmpComment.getParentId() != null){
            System.out.println("comment ID: " + comment.getElement() + " reply:" +
tmpComment.getParentId() + " comment content:" + tmpComment.getContent() + ", comment time:
" + Double.valueOf(comment.getScore()).longValue());
        }else {
            System.out.println("comment ID: " + comment.getElement() + ", comment time: " +
Double.valueOf(comment.getScore()).longValue());
        }
    }
} catch (Exception e) {
    e.printStackTrace();
} finally {
    jedisClient.quit();
    jedisClient.close();
}
}
/**
 * comment data object
 */
static class Comment{
    // Comment ID
    private Integer id;
    // Comment content
```

```
private String content;
// Comment time
private Long time;
// Comment parent ID of a reply
private Integer parentId;
public Integer getId() {
    return id;
}
public void setId(Integer id) {
    this.id = id;
}
public String getContent() {
    return content;
}
public void setContent(String content) {
    this.content = content;
}
public Long getTime() {
    return time;
}
public void setTime(Long time) {
    this.time = time;
}
public Integer getParentId() {
    return parentId;
}
public void setParentId(Integer parentId) {
    this.parentId = parentId;
}
}
```

**Step 4** Configure the connection address, port, and password of the DCS Redis instance in the sample code.

**Step 5** Compile and run the code.

----End

## Operation Result

- Sample code of bullet-screen comments in live streaming:

Live comment list

```
bullets content: message-07f1add5-2f85-4309-9f31-313c860b33dc, sent time: 1686902337377
bullets content: message-2062e817-3145-4d8b-af7f-46f334c8569c, sent time: 1686902337394
bullets content: message-ad36a0ca-e8bd-4883-a091-e12a25c00106, sent time: 1686902337396
bullets content: message-f02f9960-bb57-49ae-b7d8-6bd6d3ad3d14, sent time: 1686902337412
bullets content: message-5ca39948-866e-4e54-a469-f958cae843f6, sent time: 1686902337457
bullets content: message-5cc8b4ba-da61-4d01-9625-cf2e7337ef10, sent time: 1686902337489
bullets content: message-15378516-18ce-4da7-bd3c-35c57dd65602, sent time: 1686902337495
bullets content: message-1b280525-53e5-4fc6-a3e7-fb8e71eef85e, sent time: 1686902337540
bullets content: message-adf876d1-e747-414e-92a2-397fc329bd58, sent time: 1686902337541
bullets content: message-1d8d7901-164f-4dd4-abb4-6f2345164b0e, sent time: 1686902337582
bullets content: message-fb35b1b4-277a-48bf-b22b-80070aae8475, sent time: 1686902337667
bullets content: message-973b1b03-bf95-44d8-ab91-0c317b2d61b3, sent time: 1686902337755
bullets content: message-1481f883-757d-47f7-b8c0-df024d6e64a4, sent time: 1686902337770
bullets content: message-b79292ca-2409-43fb-aaf0-e33f3b9d9c8d, sent time: 1686902337820
bullets content: message-66b0e955-d509-4475-9ae5-12fb86cf9596, sent time: 1686902337844
bullets content: message-12b6d15a-037a-47ee-8294-8625d202c0a0, sent time: 1686902337907
bullets content: message-fbc06323-da2a-44b8-874b-d2cf1a737064, sent time: 1686902337927
bullets content: message-7a0f787c-aff1-422f-9e62-4beda0cd5914, sent time: 1686902337977
bullets content: message-8ba5e4e0-22af-4f80-90a6-35062967e0fd, sent time: 1686902337992
bullets content: message-fa9e1169-e918-4141-9805-87edcf84c379, sent time: 1686902338000
bullets content: message-5d17be15-ba2e-461f-aba5-65c20c21d313, sent time: 1686902338059
bullets content: message-dcedc840-1be7-496a-b781-5b79c2091fe5, sent time: 1686902338067
bullets content: message-9e39eb28-6629-4d4c-8970-2acdc0e81a5c, sent time: 1686902338102
bullets content: message-030b11fe-c258-4ca2-ac82-5e6ca1eb688f, sent time: 1686902338211
bullets content: message-93322018-a987-47ba-8093-3937d4dda97d, sent time: 1686902338242
```

```
bullets content: message-bc04a9b0-ec83-4a24-83f6-0a4f25ee8896, sent time: 1686902338281
bullets content: message-c6dd96d0-c938-41e4-b5d8-6275fdf83050, sent time: 1686902338290
bullets content: message-12b70173-1b86-4370-a7ea-dc0ade135422, sent time: 1686902338312
bullets content: message-a39c2ef8-8167-4945-b60d-355db6c69005, sent time: 1686902338318
bullets content: message-2c3bf2fb-5298-472c-958c-c4b53d734e89, sent time: 1686902338326
```

The latest 5 bullets

```
bullets content: message-2c3bf2fb-5298-472c-958c-c4b53d734e89, sent time: 1686902338326
bullets content: message-a39c2ef8-8167-4945-b60d-355db6c69005, sent time: 1686902338318
bullets content: message-12b70173-1b86-4370-a7ea-dc0ade135422, sent time: 1686902338312
bullets content: message-c6dd96d0-c938-41e4-b5d8-6275fdf83050, sent time: 1686902338290
bullets content: message-bc04a9b0-ec83-4a24-83f6-0a4f25ee8896, sent time: 1686902338281
```

Process finished with exit code 0

- **Sample code of replying to a comment on a social website:**

List of replies to comments on a social website

```
comment id: 1 comment parentid: null, comment time: 1684745729506
comment id: 2 comment parentid: 1, comment time: 1684745729567
comment id: 3 comment parentid: null, comment time: 1684745729630
comment id: 4 comment parentid: 3, comment time: 1684745729692
comment id: 5 comment parentid: 3, comment time: 1684745729755
comment id: 6 comment parentid: 4, comment time: 1684745729819
comment id: 7 comment parentid: null, comment time: 1684745729879
comment id: 8 comment parentid: 6, comment time: 1684745729942
comment id: 9 comment parentid: null, comment time: 1684745730006
comment id: 10 comment parentid: 7, comment time: 1684745730069
comment id: 11 comment parentid: null, comment time: 1684745730132
comment id: 12 comment parentid: 9, comment time: 1684745730194
comment id: 13 comment parentid: null, comment time: 1684745730256
comment id: 14 comment parentid: 9, comment time: 1684745730320
comment id: 15 comment parentid: null, comment time: 1684745730382
comment id: 16 comment parentid: 1, comment time: 1684745730444
comment id: 17 comment parentid: null, comment time: 1684745730508
comment id: 18 comment parentid: 12, comment time: 1684745730570
comment id: 19 comment parentid: null, comment time: 1684745730631
comment id: 20 comment parentid: 12, comment time: 1684745730694
```

The latest 5 replies

```
comment id: 20 reply:12 comment content:877ba7f1, comment time: 1684745730694
comment id: 19, comment time: 1684745730631
comment id: 18 reply:12 comment content:b29f2077, comment time: 1684745730570
comment id: 17, comment time: 1684745730508
comment id: 16 reply:1 comment content:9f31200e, comment time: 1684745730444
```

## 2.4 Merging Game Servers with DCS

### Overview

#### Application Scenario

Merging game servers is a strategy for some large online games. After running a game for a while, game providers set up a new server to attract new players. As users shift to the new server, game developers usually merge the new server and the old one, so new and old players can play together for a better game experience. During this process, game developers must consider how to synchronize data among different servers.

#### Solution

DCS for Redis can be used in the following game server merge scenarios:

- **Cross-server data synchronization**

After servers merger, data on multiple servers needs to be synchronized to ensure consistency. With the pub/sub message queuing mechanism of Redis, data changes can be published to Redis channels. Other game servers can subscribe to the channels to receive messages of changes.

- **Cross-server resource sharing**

After servers merge, resources on multiple servers, such as player props and gold coins, can be shared. The distributed lock mechanism of Redis can ensure mutual exclusion among multiple servers in resource access.

- **Cross-server ranking**

After servers merge, rankings on multiple servers can be combined to show the ranking over all servers. Sorted sets in Redis can store ranking data and perform calculation and query.

For details about cross-server resource sharing, see [Serializing Access to Frequently Accessed Resources](#). For details about cross-server ranking, see [Ranking with DCS](#).

The following describes how to implement cross-server data synchronization through pub/sub message queuing in Redis.

---

**NOTICE**

When using Redis for game server merge, you need to consider data consistency, performance, and security. Issues such as data errors, performance bottlenecks, and security vulnerabilities should be avoided.

---

## 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?](#)
  - 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 [What Is Direct Connect](#).
  - For public access  
For details about how to access a DCS Redis 4.0/5.0/6.0 instance on a client over a public network, see [Using Nginx for Public Access to DCS](#) or [Using ELB for Public Access to DCS](#).

## Procedure

- Step 1** Use the **Redis()** method from the redis-py library to create a Redis client connection on each game server.
- Step 2** Use the **pubsub()** method to create a Redis subscriber and publisher on each game server. They will be used for subscribing to messages from other game servers and publishing data changes on the local server. When a server needs to update data, it publishes updates to the Redis message queue. Other servers will receive the updates and update their local data.
- Step 3** Define a **publish\_update()** method to publish updates, and use the **subscriber.listen()** method in the **listen\_updates()** method to listen to updates.
- Step 4** Once an update is captured, the **handle\_update()** method is invoked to process the update and update local data. In game servers, the **publish\_update()** method can be invoked to publish updates, and the **listen\_updates()** method can be invoked to listen to updates.

----End

## Sample Code

The sample code (Python 2) for using the redis-py-based pub/sub mechanism to implement cross-server game data synchronization is as follows:

```
import redis
# Create a Redis client connection. Replace the Redis instance connection address and port with the actual values.
redis_client = redis.Redis(host='localhost', port=6379, db=0)
# Create a subscriber.
subscriber = redis_client.pubsub()
subscriber.subscribe('game_updates')
# Create a publisher.
publisher = redis_client
# Publish updates.
def publish_update(update):
    publisher.publish('game_updates', update)
# Process updates.
def handle_update(update):
    # Update local data.
    print('Received update:', update)
# Listen to updates.
def listen_updates():
    for message in subscriber.listen():
        if message['type'] == 'message':
            update = message['data']
            handle_update(update)
# Invoke publish_update().
publish_update('player_data_updated')
# Invoke listen_updates().
listen_updates()
```

Result:

```
D:\workspace\pythonProject\venv\Scripts\python.exe D:\workspace\pythonProject\test2.py
Received update: b'player_data_updated'
```

## 2.5 Flashing E-commerce Sales with DCS

### Overview

#### Application Scenario

An e-commerce flash sale is like an online auction. To attract customers, merchants release a small number of scarce offerings on the platform. Platforms receive dozens or even hundreds of more order placements than usual. However, only a few customers can place orders successfully. The traffic distribution process of an e-commerce flash sales system is as follows:

1. User requests: When users place orders, the requests enter the load balancing server.
2. Load balancing: The load balancing server distributes requests to multiple backend servers based on certain algorithms. The algorithms include round robin, random, and least connections.
3. Service processing logic: Backend servers receive requests and verify the requested quantity and user identity.
4. Inventory deduction: If the inventory is robust, the backend server deducts stocks, generates an order, and returns a success message to the user. If the inventory is insufficient, the backend server returns a failure message.
5. Order processing: Backend servers save the order information to the database and perform asynchronous processing such as notifying users of the order status.
6. Cache update: Backend servers update the inventory information in the cache for the next flash sale request.

The database is accessed multiple times during the flash sale process. Row-level locking is usually used to restrict access. The database can be accessed and an order can be placed only after a lock is obtained. However, the database is blocked by the sheer number of order requests.

#### Solution

As the cache of the database, DCS for Redis has the following advantages for clients to access Redis for inventory query and order placement:

- Redis offers high read/write speed and concurrency performance to meet the high concurrency requirements of e-commerce flash sales systems.
- Redis supports high-availability architecture such as master/standby and cluster. Data persistence is supported, so data can be restored even if the server breaks down.
- Redis supports transactions and atomic operations to guarantee the consistency and accuracy of operations.
- Redis caches offering and user information to reduce the database load.

In this example, the hash structure of Redis shows the offering information. **total** refers to the total amount, **booked** refers to the number of placed orders, and **remain** refers to the inventory.

```
"product": {  
  "total": 200  
  "booked":0  
  "remain":200  
}
```

During inventory deduction, the server sends a request to Redis for placing an order. Redis is single-threaded, and Lua can guarantee the atomicity of multiple commands. Run the following Lua script to deduct inventory:

```
local n = tonumber(ARGV[1])  
if not n or n == 0 then  
  return 0  
end  
local vals = redis.call("HMGET", KEYS[1], "total", "booked", "remain");  
local booked = tonumber(vals[2])  
local remain = tonumber(vals[3])  
if booked <= remain then  
  redis.call("HINCRBY", KEYS[1], "booked", n)  
  redis.call("HINCRBY", KEYS[1], "remain", -n)  
  return n;  
end  
return 0
```

## 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?](#)
  - 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 [What Is Direct Connect.](#)
  - For public access  
For details about how to access a DCS Redis 4.0/5.0/6.0 instance on a client over a public network, see [Using Nginx for Public Access to DCS](#) or [Using ELB for Public Access to DCS.](#)
- **JDK1.8** (or later) and **IntelliJ IDEA** have been installed on the client server. Download the [Jedis client](#).  
The development tools and clients mentioned in this document are for example only.

## Procedure

- Step 1** Run IntelliJ IDEA on the server. Create a Maven project, create a **SecondsKill.java** file, and paste the sample code into it. In **pom.xml**, import Jedis:

```
<dependency>  
  <groupId>redis.clients</groupId>  
  <artifactId>jedis</artifactId>
```



```
<version>4.2.0</version>  
</dependency>
```

## Step 2 Compile and run the following demo (this example uses Java).

Change the Redis connection address and port to the actual values.

```
package com.huawei.demo;  
import java.util.ArrayList;  
import java.util.*;  
  
import redis.clients.jedis.Jedis;  
import redis.clients.jedis.JedisPool;  
import redis.clients.jedis.JedisPoolConfig;  
  
public class SecondsKill {  
    private static void InitProduct(Jedis jedis) {  
        jedis.hset("product", "total", "200");  
        jedis.hset("product", "booked", "0");  
        jedis.hset("product", "remain", "200");  
    }  
  
    private static String LoadLuaScript(Jedis jedis) {  
        String lua = "local n = tonumber(ARGV[1])\n"  
            + "if not n or n == 0 then\n"  
            + "return 0\n"  
            + "end\n"  
            + "local vals = redis.call(\"HMGET\", KEYS[1], \"total\", \"booked\", \"remain\");\n"  
            + "local booked = tonumber(vals[2])\n"  
            + "local remain = tonumber(vals[3])\n"  
            + "if booked <= remain then\n"  
            + "redis.call(\"HINCRBY\", KEYS[1], \"booked\", n)\n"  
            + "redis.call(\"HINCRBY\", KEYS[1], \"remain\", -n)\n"  
            + "return n;\n"  
            + "end\n"  
            + "return 0";  
        String scriptLoad = jedis.scriptLoad(lua);  
  
        return scriptLoad;  
    }  
  
    public static void main(String[] args) {  
        JedisPoolConfig config = new JedisPoolConfig();  
        // Maximum connections  
        config.setMaxTotal(30);  
        // Maximum idle connections  
        config.setMaxIdle(2);  
        // Connect to Redis. Replace the Redis instance connection address and port with the actual values.  
        JedisPool pool = new JedisPool(config, "127.0.0.1", 6379);  
        Jedis jedis = null;  
        try {  
            jedis = pool.getResource();  
            jedis.auth("password"); //Configure the password of the instance. You do not need to set  
this parameter for password-free access.  
            System.out.println(jedis);  
  
            // Initialize product information.  
            InitProduct(jedis);  
  
            // Load the Lua script.  
            String scriptLoad = LoadLuaScript(jedis);  
  
            List<String> keys = new ArrayList<>();  
            List<String> vals = new ArrayList<>();  
            keys.add("product");  
  
            // Request 15 items.  
            int num = 15;  
            vals.add(String.valueOf(num));  
  
            // Run the Lua script.
```

```
jedis.evalsha(scriptLoad, keys, vals);
System.out.println("total:"+jedis.hget("product", "total")+"\n"+"booked:"+jedis.hget("product",
    "booked")+"\n"+"remain:"+jedis.hget("product","remain"));

} catch (Exception ex) {
    ex.printStackTrace();
} finally {
    if (jedis != null) {
        jedis.close();
    }
}
}
```

Result:

```
total:200
booked:15
remain:185
```

----End

## 2.6 Reconstructing Application System Databases with DCS

### Overview

#### Application Scenario

With the development of database applications like the Internet, service demands are increasing rapidly. As the data volume and concurrent access volume are increasing exponentially, conventional relational databases can hardly support upper-layer services. Conventional databases are faced with issues such as complex structure, high maintenance costs, poor access performance, limited functions, and difficulty adapting to changes in data models or modes.

#### Solution

As a cache layer between the application and database, Redis can solve the above issues and improve data read speed, reduce database load, improve application performance, and ensure data reliability.

Data can be migrated from conventional relational databases such as MySQL to Redis. Since data in Redis is stored in the key-value structure, you need to convert the data structure in conventional databases. The following sections describe how to migrate a table from MySQL to DCS for Redis.

### Prerequisites

- You have a DCS Redis instance as the target database. For details, see [Buying a DCS Redis Instance](#).

#### NOTE

If your source is the Huawei Cloud MySQL database, select a DCS Redis instance in the same VPC as the database.

- You have a MySQL database with a table as the source data.  
For example, create a table named **student\_info** with 4 columns. After migration, the values in the **id** column of the table will be the hash keys in

Redis, the names of the other columns will be the hash fields, and their values will be the field values.

```
mysql> select * from student_info;
+----+-----+-----+-----+
| id | name   | birthday | city   |
+----+-----+-----+-----+
| 1  | Wilin  | 1995-06-12 | Nanjing |
| 2  | Xiaoming | 1994-06-10 | Guangzhou |
| 3  | John   | 1995-09-03 | NewYork |
| 4  | Anbei  | 1969-10-19 | Dongjing |
+----+-----+-----+-----+
```

- The server of the MySQL database can communicate with the DCS instance.
  - When the MySQL database and Redis instance are in the same VPC:  
By default, networks in a VPC can communicate with each other.
  - When the MySQL database and Redis instance are in different VPCs in the same region:  
If the VPC of the MySQL database and DCS Redis instance are not in the same VPC, they can be connected using a VPC peering connection. For details, see [Does DCS Support Cross-VPC Access?](#).
  - When the VPCs of the MySQL database and the DCS Redis instance are not in the same region:  
If the MySQL database and the Redis instance are not in the same region, connect the network using Direct Connect. For details, see [What Is Direct Connect](#).
  - For public access  
For details about how to access a DCS Redis 4.0/5.0/6.0 instance on a MySQL database server over a public network, see [Using Nginx for Public Access to DCS](#) or [Using ELB for Public Access to DCS](#).
- **JDK1.8** (or later) and **IntelliJ IDEA** have been installed on the MySQL database server. Download the [Jedis client](#).

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

## Procedure

- Step 1** Log in to the MySQL database server.
- Step 2** Install the Redis client on the server to extract, transmit, and convert data. For details about Redis client installation, see [redis-cli](#).
- Step 3** Analyze the source data structure, create the following script on the server, and save the script as **migrate.sql**.

```
SELECT CONCAT(
  "*"8\r\n", #8 refers to the number of fields as follows. It depends on the data structure in the MySQL table.
  '$', LENGTH('HMSET'), '\r\n', #HMSET is a Redis command in the data writing process.
  'HMSET', '\r\n',
  '$', LENGTH(id), '\r\n', #id is the first field after HMSET. It will be transferred into Redis as a hash key.
  id, '\r\n',
  '$', LENGTH('name'), '\r\n', #'name' will be transferred into the hash field as strings, and other arguments
  such as 'birthday' are applied in the same way.
  'name', '\r\n',
```

```
'$', LENGTH(name), '\r\n', #name is a variable representing the company name in the MySQL table. It will
be transferred to be the value corresponding to the field of the last argument 'name'. Other variables such
as birthday are applied in the same way.
name, '\r\n',
'$', LENGTH(' birthday'), '\r\n',
' birthday', '\r\n',
'$', LENGTH(birthday), '\r\n',
birthday, '\r\n',
'$', LENGTH('city'), '\r\n',
'city', '\r\n',
'$', LENGTH(city), '\r\n',
city, '\r'
)
FROM student_info AS s
```

**Step 4** Run the following command on the server to migrate data:

```
mysql -h <MySQL host> -P <MySQL port> -u <MySQL username> -D <MySQL database name> -p --skip-
column-names --raw < migrate.sql | redis-cli -h <Redis host> -p<Redis port> --pipe -a <Redis password>
```

**Table 2-1** Parameters

Parameter	Description	Example
-h	Address of the MySQL database.	xxxxxx
-P	Port of MySQL.	3306
-u	Username of MySQL.	root
-D	Database whose table is to be migrated.	mysql
-p	Password of MySQL. If MySQL does not have a password, leave this parameter blank. For security, you can enter <b>-p</b> only, and enter your password when prompted by the command window after running the command.	xxxxxx
--skip-column-names	The column names will not be written in query results.	No need to be set.
--raw	No escape in outputting column values.	No need to be set.
-h after redis-cli	Address of Redis.	redis-xxxxxxxxxxxx.com
-p after redis-cli	Port of Redis.	6379
--pipe	Use Redis pipelining to transmit data.	No need to be set.
-a	Password of Redis. It does not need to be set if your Redis does not have a password.	xxxxxx

```
[root@ecs-cmtest mysql-8.0]# mysql -h redis-xxxxxx.com -P 3306 -u root -D mysql -p --skip-column-names --raw < migrate.sql | redis-cli -h redis-xxxxxx.com -p 6379 --pipe
All data transferred. Waiting for the last reply...
Last reply received from server.
errors: 0, replies: 4
```

In this screenshot, the Redis instance does not have a password. In the result, **errors** refers to the number of errors during running, and **replies** refers to the number of replies received. If **errors** is **0**, and **replies** is equal to the the number of records in the MySQL table, the table is migrated successfully.

**Step 5** One piece of MySQL data corresponds to one hash in Redis. Run the **HGETALL** command for query and verification. Result:

```
[root@ecs-cmtest mysql-8.0]# redis-cli -h redis-xxxxxx.com -p 6379
redis-xxxxxx.com:6379> HGETALL 1
1) "name"
2) "Wilin"
3) " birthday"
4) "1995-06-12"
5) "city"
6) "Nanjing"
redis-xxxxxx.com:6379> HGETALL 4
1) "name"
2) "Anbei"
3) " birthday"
4) "1969-10-19"
5) "city"
6) "Dongjing"
```

#### NOTE

You can adjust the migration plan based on actual query needs. For example, you can convert other columns in MySQL to the hash keys, and convert the **id** column to the field.

----End

## 2.7 Upgrading a Redis 3.0 Instance

### Overview

Redis has not updated Redis 3.0 since the release of a minor version on May 19, 2019. Huawei Cloud DCS also announced the discontinuation of DCS for Redis 3.0 in March 2021.

You are advised to upgrade your DCS Redis 3.0 to a later version as soon as possible. DCS for Redis 4.0/5.0/6.0 are compatible with Redis 3.0.

Currently, DCS does not support direct instance upgrades. In this case, migrate the data of the earlier instance to that of a later one. Follow the instructions below to upgrade a DCS Redis 3.0 instance by migrating data and switching IPs.

 **NOTE**


- DCS Redis 3.0 instances support public access, while DCS Redis 4.0/5.0/6.0 instances do not. If your services rely on public access, do not perform the upgrade.
- Upgrading the Redis version through data migration may have the following impacts on services:
  - The source and target Redis instance IP addresses need to be switched after data migration is complete. During the switching, the instances will become read-only within 1 minute and be interrupted for 30 seconds.
  - If the target instance does not share the password as the source, the password needs to be updated after data synchronization is complete. Stop the services during the update. In this case, you are advised to use the same password for the source and target instances.
- Upgrade instances during off-peak hours.

## Prerequisites

- You have created a DCS Redis instance of a later version. This instance must be in the same VPC and subnet, of the same instance type, configured with the same password as the source instance, and have specifications greater than or equal to the source instance. For example, to upgrade a 16 GB master/standby DCS Redis 3.0 instance to a DCS Redis 5.0 one, prepare a master/standby DCS Redis 5.0 instance with at least 16 GB memory.  
For details about how to create a DCS Redis instance, see [Buying a DCS Redis Instance](#).
- You have manually backed up data of the source DCS Redis 3.0 instance. For details about how to back up data, see [How Do I Export DCS Redis Instance Data?](#).

## Migrating Instance Data

**Step 1** Log in to the [DCS console](#).

**Step 2** Click  in the upper left corner of the console and select the region where your source instance is located.

**Step 3** In the navigation pane, choose **Data Migration**. The migration task list is displayed.

**Step 4** In the upper right corner, click **Create Online Migration Task**.

**Step 5** Enter the task name and description.

**Step 6** Configure the VPC, subnet, and security group for the migration task.

- Select the VPC where the source and target Redis instances are, so they can be connected with the task.
- The migration task uses a tenant IP address (**Migration ECS** displayed on the **Basic Information** page of the task). If the target Redis has an IP whitelist, this IP address must be added to the whitelist.
- To allow the VM used by the migration task to access the source and target instances, set an outbound rule for the task's security group to allow traffic through the IP addresses and ports of the source and target instances. By default, all outbound traffic is allowed.

- Step 7** After the migration task is created, click **Configure** in the **Operation** column of the task on the **Online Migration** tab page to configure the source Redis and target Redis.
- Step 8** Select **Full + Incremental** for **Migration Type**. IP switching can be performed on the console only for instances using full and incremental migration. Selecting **Full** requires a manual IP change of the Redis instance.

**Table 2-2** Migration type description

Migration Type	Description
Full	Suitable for scenarios where services can be interrupted. Data is migrated at one time. <b>Source instance data updated during the migration will not be migrated to the target instance.</b>
Full + incremental	Suitable for scenarios requiring minimal service downtime. The incremental migration parses logs to ensure data consistency between the source and target instances.  Once the migration starts, it remains <b>Migrating</b> until you click <b>Stop</b> in the <b>Operation</b> column. After the migration is stopped, data in the source instance will not be lost, but data will not be written to the target instance. When the transmission network is stable, the delay of incremental migration is within seconds. The actual delay depends on the transmission quality of the network link.

- Step 9** If **Migration Type** is set to **Full + Incremental**, you can specify a bandwidth limit. The data synchronization rate can be kept around the bandwidth limit.
- Step 10** Specify **Auto-Reconnect**. If this option is enabled, automatic reconnections will be performed indefinitely in the case of a network exception.  
  
Full synchronization will be triggered and requires more bandwidth if incremental synchronization becomes unavailable. Exercise caution when enabling this option.
- Step 11** For **Source Redis** and **Target Redis**, select the Redis 3.0 instance to be upgraded and the new Redis instance.
- Step 12** If the source and target Redis instances are password-protected, enter **Source Redis Instance Password** and **Target Redis Instance Password**, and click **Test Connection** respectively to check whether the network can be connected. If the source and target Redis instances are password-free, click **Test Connection**.
- Step 13** **Source DB** and **Target DB** specify migration databases. For example, if you enter **5** for source DB and **6** for target DB, data in DB5 of the source Redis will be migrated to the DB6 of the target Redis. If the source DB is not specified but the target DB is specified, all source data will be migrated to the specified target DB by default. If the target DB is not specified, data will be migrated to the corresponding target DB. Leave **Source DB** and **Target DB** blank in this operation.
- Step 14** Click **Next**.

**Step 15** Confirm the migration task details and click **Submit**.

Go back to the data migration task list. After the migration is successful, the task status changes to **Successful**.

**NOTE**

- Once incremental migration starts, it remains **Migrating**.
- To manually stop migration, click **Stop**.
- After data migration, duplicate keys will be overwritten.

If the migration fails, click the migration task and check the log on the **Migration Logs** page.

----End

## Verifying the Migration

Before data migration, if the target Redis has no data, check data integrity after the migration is complete in the following way:

1. Connect to the source Redis and the target Redis. Connect to Redis by referring to [redis-cli](#).
2. Run the **info keyspace** command to check the values of **keys** and **expires**.

```
192.168.1.217:6379> info keyspace
# Keyspace
db0:keys=81869,expires=0,avg_ttl=0
192.168.1.217:6379>
```

3. Calculate the differences between the values of **keys** and **expires** of the source Redis and the target Redis. If the differences are the same, the data is complete and the migration is successful.

During full migration, source Redis data updated during the migration will not be migrated to the target instance.

## Switching DCS Instance IPs

The prerequisites for switching source and target Redis instance IP addresses are as follows. The target Redis can be accessed automatically on a client after the switch.

- The source and target must be basic edition Redis instances excluding Redis Cluster ones. This function is unavailable for professional edition and Redis Cluster instances.
- For sources of DCS Redis 3.0 instances, contact the administrator to enable the whitelist for Redis 3.0 instance IP switches. The instance IP addresses can be switched only when the source instance is a DCS Redis 3.0 instance and the target instance is a basic edition DCS Redis 4.0, 5.0, or 6.0 instance.
- The IP addresses of a source or target instance with public access enabled cannot be switched on the console. Manually change the IP addresses if needed.
- **Full + Incremental** must be selected in [Step 8](#).
- The source and target Redis instance ports must be consistent.



---

**NOTICE**

1. IP switching stops online migration tasks.
  2. When the source is a Redis 3.0 instance, the instance will be read-only for one minute and disconnected for 30 seconds during an IP switch.
  3. If your application cannot reconnect to Redis or handle exceptions, you may need to restart the application after the IP switching.
  4. If the source is a master/standby instance, the IP address of the standby node will not be switched. Ensure that this IP address is not used by your applications.
  5. If your applications use a domain name to connect to Redis, the domain name will be used for the source instance. Select **Yes** for **Switch Domain Name**.
  6. Ensure that the passwords of the source and target instances are the same. If they are different, verification will fail after the switching.
  7. After the IP addresses of a DCS Redis 3.0 instance are switched, synchronize the security group of the source to the whitelist of the target.
- 

**Step 1** On the **Data Migration > Online Migration** page, when the migration task status changes to **Incremental migration in progress**, choose **More > Switch IP** in the **Operation** column.

**Step 2** In the **Switch IP** dialog box, select whether to switch the domain name.

 **NOTE**

- If a Redis domain name is used on the client, switch it or you must modify the domain name on the client.
- If the domain name switch is not selected, only the instance IP addresses will be switched.

**Step 3** Click **OK**. The IP address switching task is submitted successfully. When the status of the migration task changes to **IP switched**, the IP address switching is complete.

----End

## Verifying Service Functions

- Verify that service functions are normal. For example, check whether an error is reported when the client accesses Redis.
- Check whether key performance metrics are normal, such as **Connected Clients**, **Ops per Second**, **CPU Usage**, and **Memory Usage**.

# 3 Network Connection

## 3.1 Using Nginx for Public Access to DCS

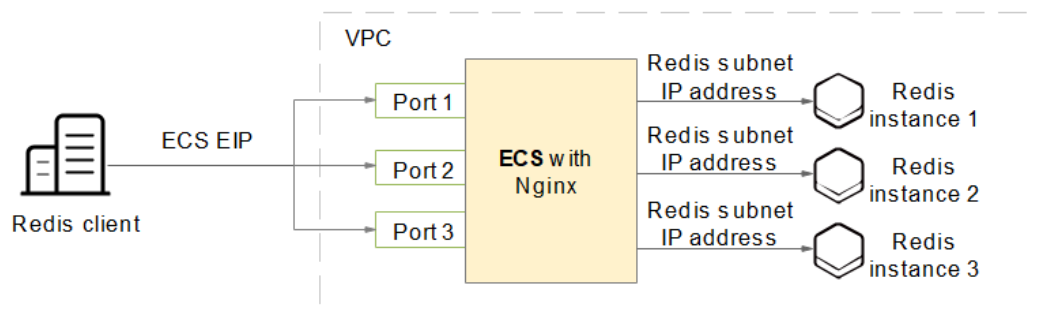
### Overview

Currently, Huawei Cloud DCS Redis 4.0 and later cannot be bound with elastic IP addresses (EIPs) and cannot be accessed over public networks directly.

This section describes how to access a single-node, master/standby, read/write splitting, or Proxy Cluster DCS Redis 4.0, 5.0, or 6.0 instance by using a jump server. **This solution cannot be used to access a Redis Cluster instance over public networks.**

As shown in [Figure 3-1](#), the ECS where Nginx is installed is a jump server. The ECS is in the same VPC as the DCS Redis instances and can access the DCS Redis instances through the subnet IP addresses. After an EIP is bound to the ECS, the ECS can be accessed over the public network. Nginx can listen on multiple ports and forward requests to different DCS Redis instances.

**Figure 3-1** Accessing DCS Redis instances in a VPC by using Nginx



### NOTE

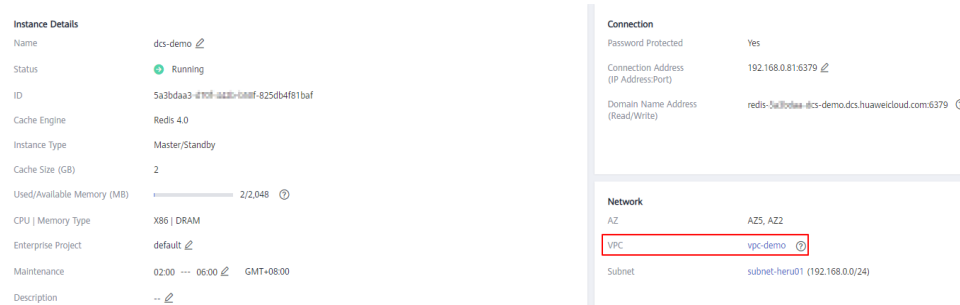
Do not use public network access in the production environment. Client access exceptions caused by poor public network performance will not be included in the SLA.

## Buying an ECS

**Step 1** Obtain the VPC where the DCS Redis instance is deployed.

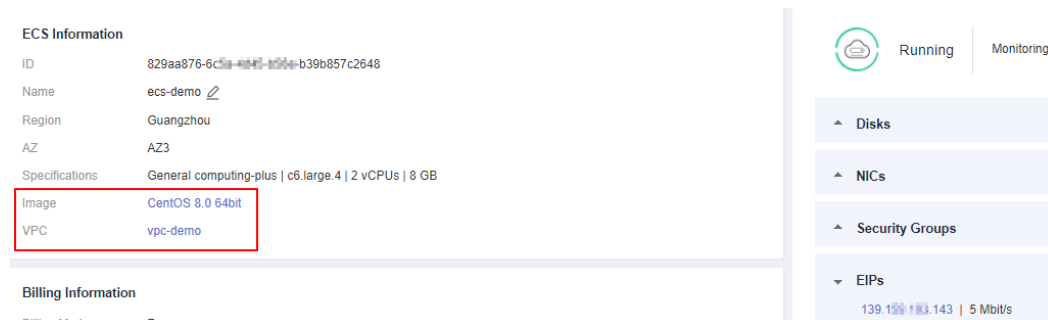
As shown in the following figure, the master/standby instance is deployed in the **vpc-demo** VPC.

**Figure 3-2** DCS Redis instance details



**Step 2** Buy an ECS. Configure the ECS with the **vpc-demo** VPC, bind an EIP to the ECS, and select the bandwidth as required.

**Figure 3-3** ECS details



----End

## Installing Nginx

After buying an ECS, install Nginx on the ECS. The following uses CentOS 7.x as an example to describe how to install Nginx. The commands vary depending on the OS.

**Step 1** Run the following command to add Nginx to the Yum repository:

```
sudo rpm -Uvh http://nginx.org/packages/centos/7/noarch/RPMS/nginx-release-centos-7-0.el7ngx.noarch.rpm
```

**Step 2** Run the following command to check whether Nginx has been added successfully:

```
yum search nginx
```

**Step 3** Run the following command to install Nginx:

```
sudo yum install -y nginx
```

**Step 4** Run the following command to install the stream module:

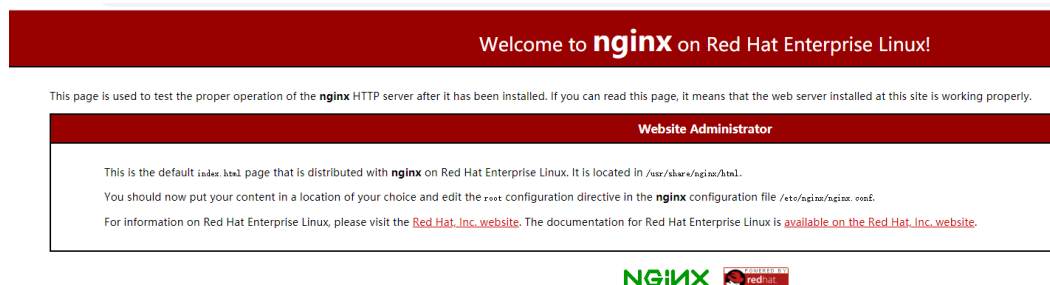
```
yum install nginx-mod-stream --skip-broken
```

**Step 5** Run the following commands to start Nginx and set it to run automatically upon system startup:

```
sudo systemctl start nginx.service
sudo systemctl enable nginx.service
```

**Step 6** In the address box of a browser, enter the server address (the EIP of the ECS) to check whether Nginx is installed successfully.

If the following page is displayed, Nginx has been installed successfully.



----End

## Setting Up Nginx

After installing Nginx, configure request forwarding rules to specify the ports that Nginx listens on and the DCS Redis instances that Nginx forwards requests to.

**Step 1** Open and modify the configuration file.

```
cd /etc/nginx
vi nginx.conf
```

The following is a configuration example. To access multiple DCS Redis instances over public networks, configure multiple **server** sections and configure the DCS Redis instance connection addresses for **proxy\_pass**.

```
stream {
    server {
        listen 8080;
        proxy_pass 192.168.0.5:6379;
    }
    server {
        listen 8081;
        proxy_pass 192.168.0.6:6379;
    }
}
```

### NOTE

Set **proxy\_pass** to the IP address of the DCS Redis instance in the same VPC. You can obtain the IP address from the **Connection** area on the DCS instance details page.

**Figure 3-4** Adding Nginx configurations

```
# * Official Russian Documentation: http://nginx.org/ru/docs/

user nginx;
worker_processes auto;
error_log /var/log/nginx/error.log;
pid /run/nginx.pid;

# Load dynamic modules. See /usr/share/doc/nginx/README.dynamic.
include /usr/share/nginx/modules/*.conf;

events {
    worker_connections 1024;
}

stream {
    server {
        listen 8080;
        proxy_pass 192.168.0.5:6379;
    }
    server {
        listen 8081;
        proxy_pass 192.168.0.6:6379;
    }
}
```

**Step 2** Restart Nginx.

```
service nginx restart
```

**Step 3** Verify whether Nginx has been started.

```
netstat -an|grep 808
```

**Figure 3-5** Starting Nginx and verifying the start

```
[root@kvm-emo src]# service nginx restart
Redirecting to /bin/systemctl restart nginx.service
[root@kvm-emo src]# netstat -an |grep 808
tcp        0      0 0.0.0.0:8080          0.0.0.0:*             LISTEN
tcp        0      0 0.0.0.0:8081          0.0.0.0:*             LISTEN
unix  2      [ ACC ]     STREAM  LISTENING   18084  /var/lib/sss/pipes/private/sbus-monitor
unix  3      [   ]     STREAM  CONNECTED   18086  /var/lib/sss/pipes/private/sbus-monitor
unix  3      [   ]     STREAM  CONNECTED   18085
```

If Nginx is listening on ports 8080 and 8081, Nginx has been started successfully.

----End

## (Optional) Persistent Connections

If persistent connections ("pconnect" in Redis terminology) are required for public network access, add the following configuration in [Configuring Nginx](#):

- Timeout of a connection from Nginx to the server

```
stream {
    server {
        listen 8080;
        proxy_pass 192.168.0.5:6379;
        proxy_socket_keepalive on;
        proxy_timeout 60m;
        proxy_connect_timeout 60s;
    }
    server {
        listen 8081;
        proxy_pass 192.168.0.6:6379;
        proxy_socket_keepalive on;
        proxy_timeout 60m;
        proxy_connect_timeout 60s;
    }
}
```

The default value of **proxy\_timeout** is **10m** (10 minutes). You can set it to **60m** or other values as required. For details about this parameter, see [the Nginx official website](#).

- Timeout of a connection from the client to Nginx

```
http {
    keepalive_timeout 3600s;
}
```

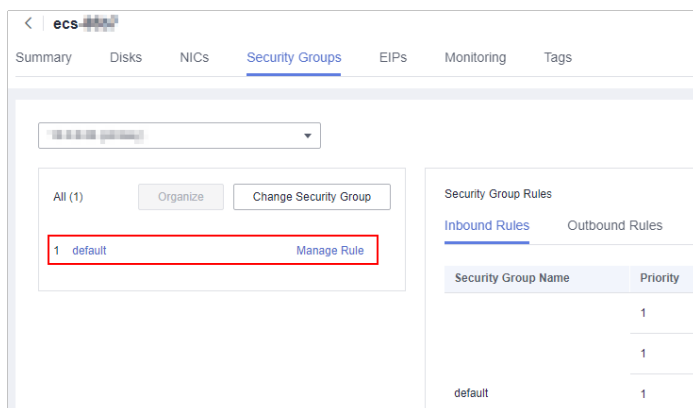
The default value of **keepalive\_timeout** is **75s**. You can set it to **3600s** or other values as required. For details about this parameter, see [the Nginx official website](#).

## Accessing DCS Redis Instances Using Nginx

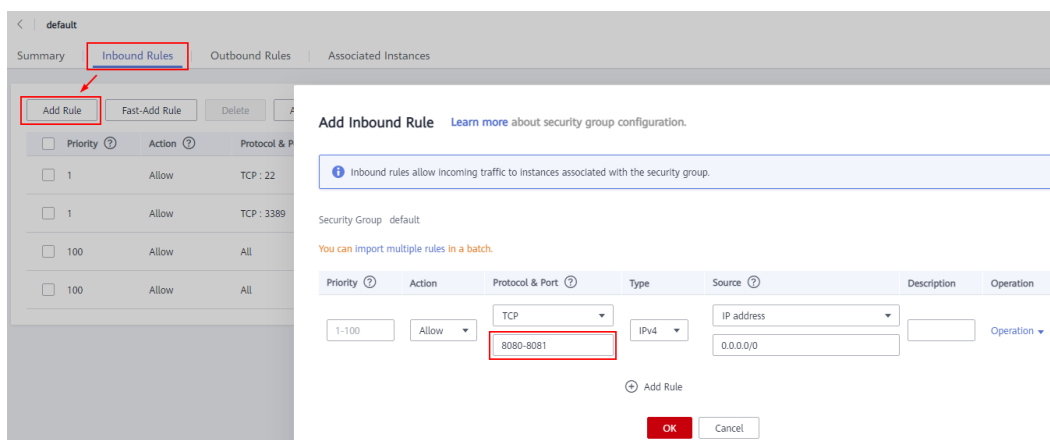
**Step 1** Log in to the ECS console and check the security group rules of the ECS that serves as the jump server. Ensure that access over ports 8080 and 8081 is allowed.

1. Click the ECS name to go to the details page.
2. On the **Security Groups** tab page, click **Modify Security Group Rule**. The security group configuration page is displayed.

**Figure 3-6** Checking the ECS security group



**Figure 3-7** Adding an inbound rule for the security group



**Step 2** In the public network environment, open the redis-cli and run the following command to check whether the login and query are successful.

**NOTE**

Ensure that redis-cli has been installed in the public network environment by referring to [redis-cli](#).

```
./redis-cli -h {myeip} -p {port} -a {mypassword}
```

In the preceding command, *{myeip}* indicates the host connection address, which should be replaced with the EIP of the ECS. Replace *{port}* with the listening port of Nginx.

As shown in the following figures, the two listening ports are 8080 and 8081, which correspond to two DCS Redis instances.

**Figure 3-8** Accessing the first DCS Redis instance using Nginx

```
[root@kafka-demo src]# ./redis-cli -h 121.37.115.247 -p 8080 -a QAZwsx@123
121.37.115.247:8080> set abc 123
OK
121.37.115.247:8080> get abc
"123"
121.37.115.247:8080> █
```

**Figure 3-9** Accessing the second DCS Redis instance using Nginx

```
[root@kafka-demo src]# ./redis-cli -h 121.37.115.247 -p 8081 -a QAZwsx@123
121.37.115.247:8081> set hello world
OK
121.37.115.247:8081> get hello
"world"
121.37.115.247:8081> █
```

----End

## 3.2 Using SSH Tunneling for Public Access to DCS

### Overview

Currently, Huawei Cloud DCS Redis 4.0 and later cannot be bound with elastic IP addresses (EIPs) and cannot be accessed over public networks directly.

This section describes how to create an SSH tunnel as a proxy to connect your DCS instance and local computer to achieve proxy forwarding. In this way, single-node, master/standby, read/write splitting, and Proxy Cluster DCS Redis 4.0/5.0/6.0 instances in a VPC can be accessed. **This solution is unavailable for public access to Redis Cluster instances.**

**NOTE**

Do not use public network access in the production environment. Client access exceptions caused by poor public network performance will not be included in the SLA.

## Prerequisites

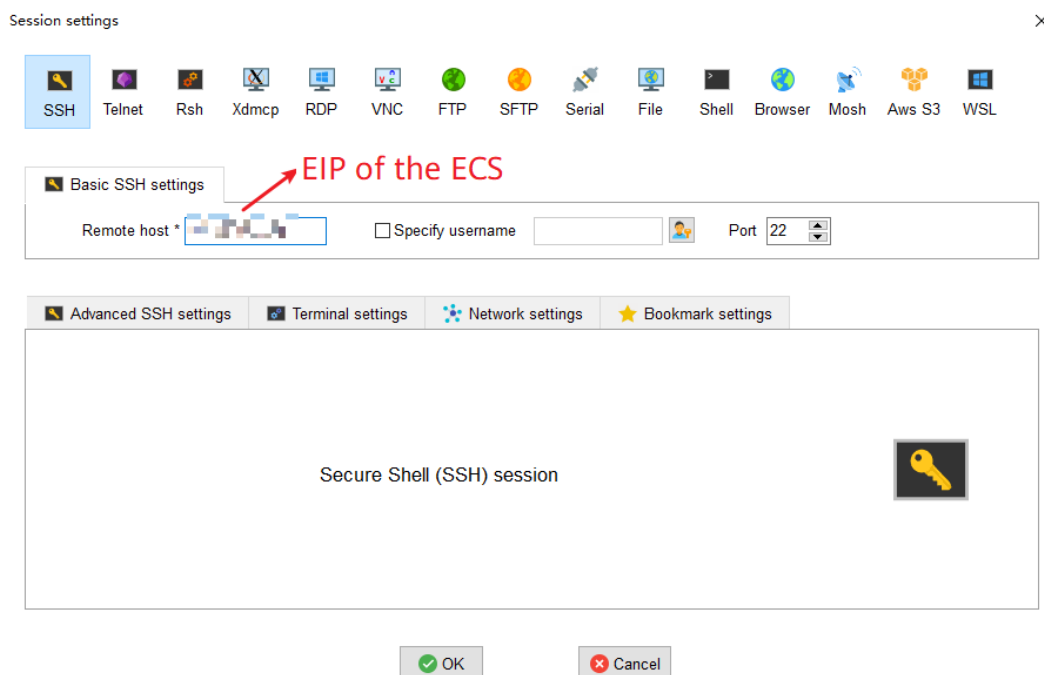
- A DCS instance has been created, and is in the **Running** state.
- Apply for an Elastic Cloud Server (ECS). If the following prerequisites are met, the ECS can communicate with the DCS instance and you can remotely connect to the ECS using SSH from a local computer.
  - The ECS is bound with an EIP for public access.
  - The VPC and subnet configured for the ECS are the same as those configured for the DCS instance.
  - Security group rules have been correctly configured for the ECS.
  - The ECS runs the Linux OS.
- You have a local computer that can connect to the Internet. Tools such as MobaXterm and the Redis client have been installed.

## Procedure

**Step 1** Open MobaXterm on the local PC.

**Step 2** Create an SSH session for connecting to the ECS using port 22.

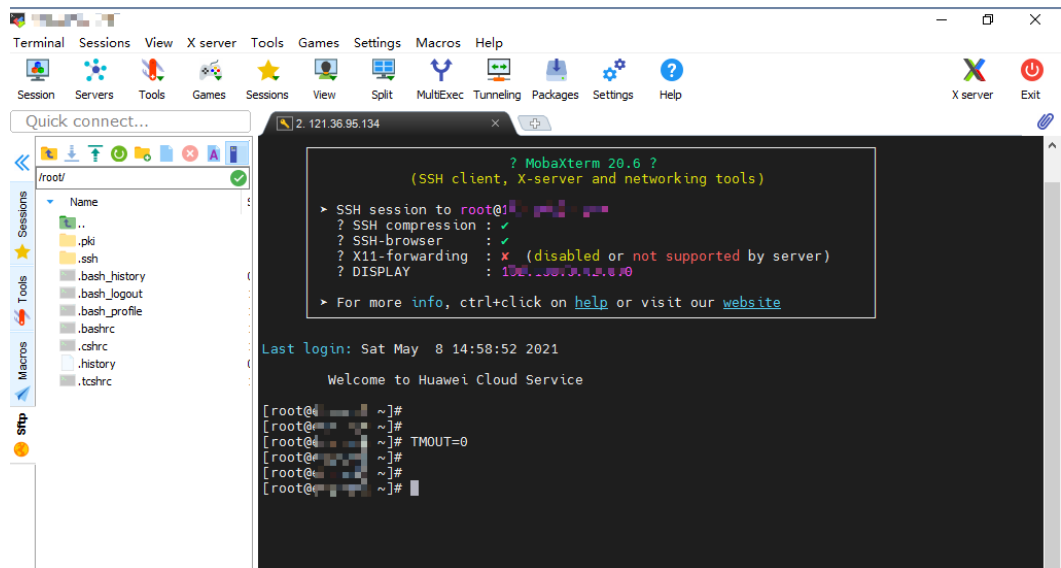
**Figure 3-10** Creating an SSH session



**Step 3** After the SSH is configured, enter the username and password to log in to the ECS. After login, enter **TMOUT=0** to prevent the session from being automatically closed due to timeout.

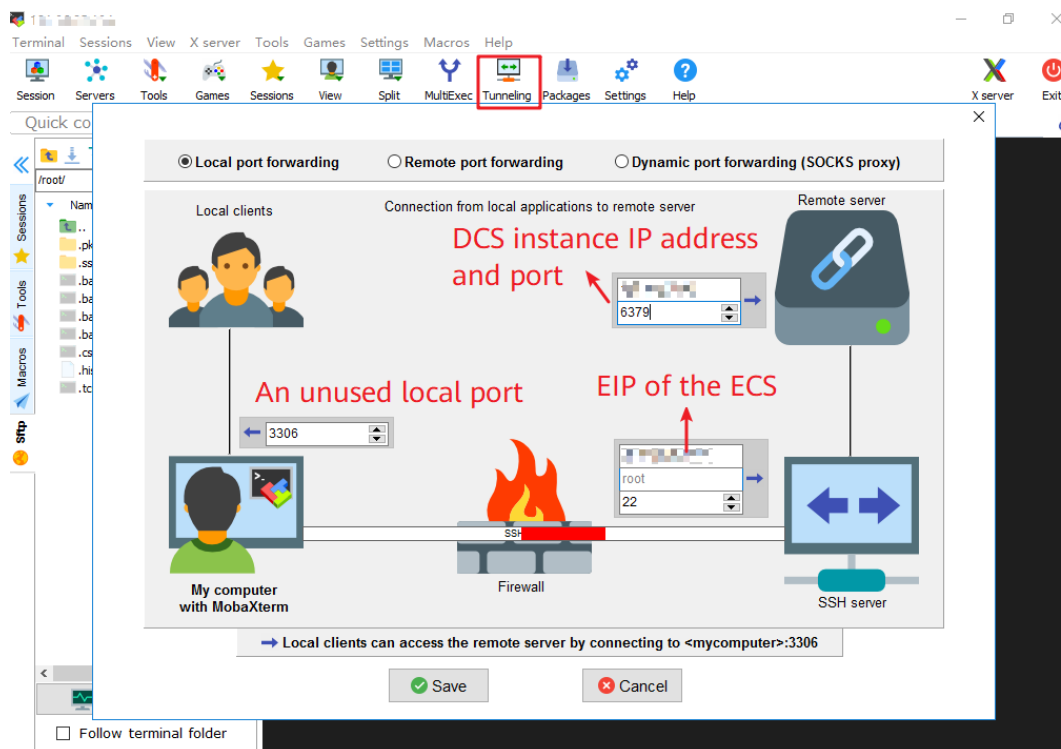


Figure 3-11 Entering "TMOUT=0"



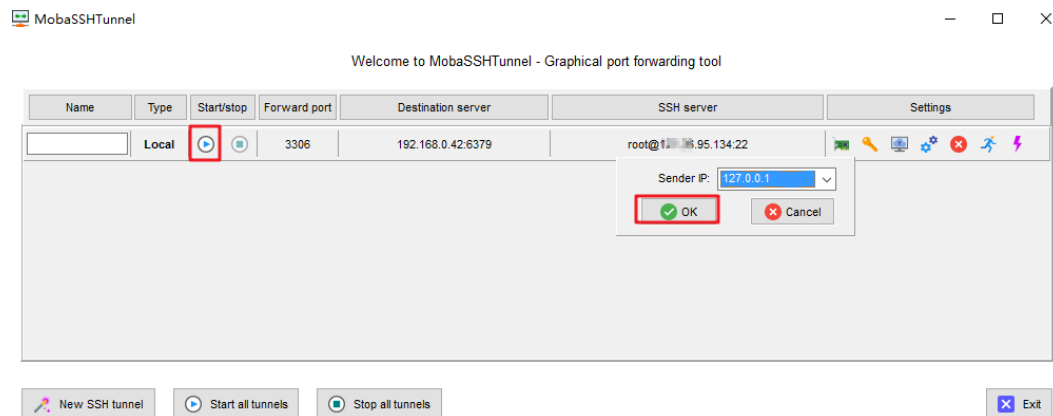
Step 4 Click **Tunneling** to create a tunnel.

Figure 3-12 Creating a tunnel



Step 5 Set the local IP address to 127.0.0.1 and start the tunnel.

**Figure 3-13** Starting the tunnel



**Step 6** Open the Redis client on the local computer. The following uses the Redis CLI as an example. Run the following command to access the DCS instance:

```
Redis-cli -h 127.0.0.1 -p 3306 -a {password}
```

Parameter description:

- **-h {host name}**: localhost or 127.0.0.1, which is the same as the local IP address configured for the tunnel.
- **-p {port number}**: 3306, which is the same as the forward port configured for the tunnel.
- **-a {password}**: password of the DCS instance.

**Step 7** If the connection is successful, the following information is displayed.

**Figure 3-14** Successfully connecting to a DCS instance

```
C:\Redis>
C:\Redis>Redis-cli -h 127.0.0.1 -p 3306 -a {password}
127.0.0.1:3306> info
# Server
redis_version:5.0.9
patch_version:5.0.9.2
redis_git_sha1:00000000
redis_git_dirty:0
redis_build_id:0
redis_mode:standalone
os:Linux
arch_bits:64
multiplexing_api:epoll
atomicvar_api:atomic-builtin
gcc_version:0.0.0
process_id:1
run_id:74daa94034ce1c8287e3a47b48d446cc04cfdb5b
tcp_port:3397
uptime in seconds:2421
```

----End

## 3.3 Using ELB for Public Access to DCS

### Overview

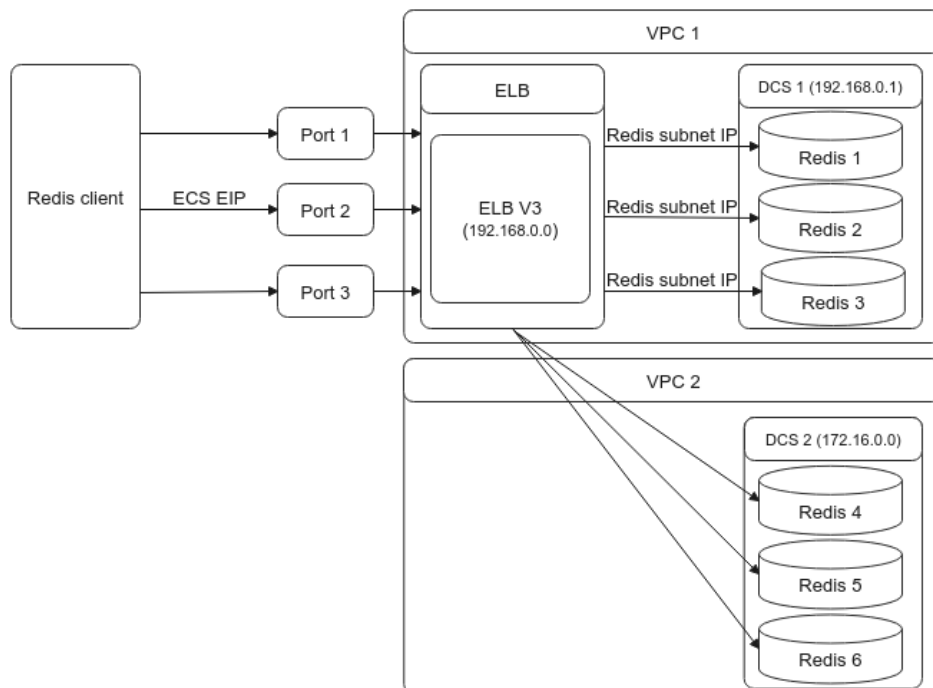
Currently, Huawei Cloud DCS Redis 4.0 and later cannot be bound with elastic IP addresses (EIPs) and cannot be accessed over public networks directly. This section describes how to access a single-node, master/standby, read/write splitting, or Proxy Cluster instance or a node in a Redis Cluster instance through public networks by enabling cross-VPC backend on a load balancer.

#### NOTE

- Due to cluster node address translation, you cannot access a Redis Cluster as a whole. You can only access individual nodes in the cluster.
- Do not use public network access in the production environment. Client access exceptions caused by poor public network performance will not be included in the SLA.

The following figure shows the process for accessing DCS through ELB.

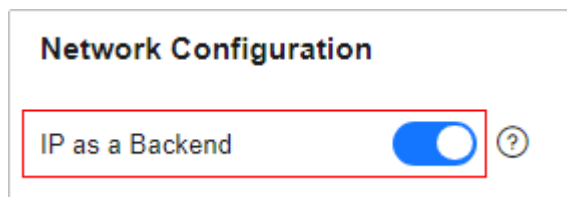
**Figure 3-15** Process for accessing DCS through ELB



### Interconnecting ELB with a DCS Instance

- Step 1** [Create a VPC](#) or use an existing one.
- Step 2** [Buy a DCS Redis instance](#). Record the IP address and port number of the instance.
- Step 3** [Create a dedicated load balancer](#).
  - A shared load balancer does not support cross-VPC backend servers. Therefore, it cannot be bound to a DCS instance.

- For **Specification**, select **Network load balancing (TCP/UDP)**.
- To access the DCS instance over public networks, enable **IP as a Backend** when creating a dedicated load balancer.



**Step 4** Add a TCP listener to the dedicated load balancer.

- On the **Add Backend Server** tab page, choose **IP as Backend Servers > Add IP as Backend Server**.
- Enter the IP address, port, and other parameters of your DCS instance.
- A Redis Cluster DCS instance contains multiple master/replica pairs. When configuring **IP as a Backend**, you can add the IP address and port of any master or replica node.
- If you enable **Health Check**, you do not need to manually configure the port. By default, the service port of the backend server will be used.

**Step 5** Create a VPC peering connection. For the local VPC, select the VPC where your load balancer is located. For the peer VPC, select the VPC where your DCS instance is located.

**NOTE**

Even if your load balancer and DCS instance are in the same VPC, you still need to create a VPC peering connection. For the local VPC, select the VPC where your load balancer and DCS instance are located. For the peer VPC, select another VPC.

**Step 6** Click the name of the VPC peering connection to go to its details page. Obtain **Local VPC CIDR Block** and **Peer VPC CIDR Block**.

Name	peering	Status	Accepted
ID	980e10fe-49fa-4225-92a1-2038911a9d11	Peer Project ID	eaaf7e74f0a444c17a339277929e2254f
Local VPC Name	vpc-d48c	Peer VPC Name	vpc-172-test
Local VPC ID	38e9d503-a64b-4c13-b9fa-61a55c9d7d04	Peer VPC ID	8a9dd14c-8ce8-4ee8-a467-72a9f5bdead9
Local VPC CIDR Block	192.168.0.0/16	Peer VPC CIDR Block	172.16.0.0/24
Description	-		

**Step 7** Click **Add Route** and configure local and peer routes for the VPC peering connection.

1. Local route: Configure the peer VPC CIDR block in **Destination** on the **Add Route** dialog box.
2. Peer route: Select **Add a route for the other VPC** and configure the local VPC CIDR block in **Destination**, and click **OK**.

**NOTE**

If the load balancer and the DCS instance are in the same VPC, you do not need to add a peer route.

**Step 8** Perform a health check on the IP address of the DCS instance. If the health check result is **Healthy**, the added cross-VPC backend IP address can be used.

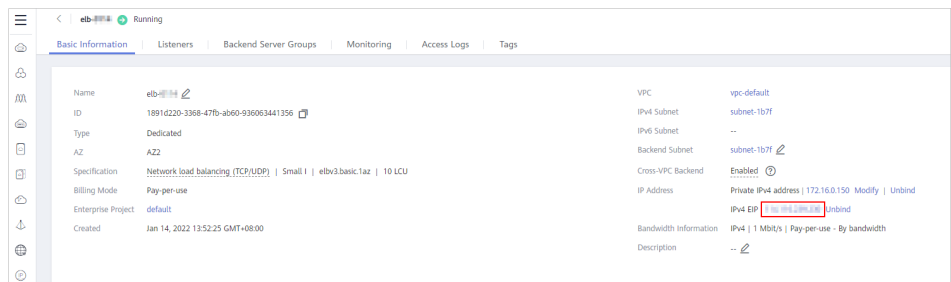
1. In the navigation pane of **Network Console**, choose **Elastic Load Balance > Backend Server Groups**.
2. Click the name of the created backend server group to go to its details page.
3. On the **Backend Servers > IP as Backend Servers** tab page, view the health check result of the DCS instance IP address.



----End

## Accessing a DCS Instance on a Client

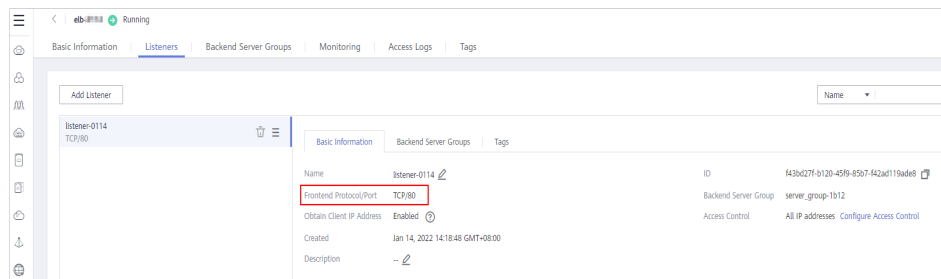
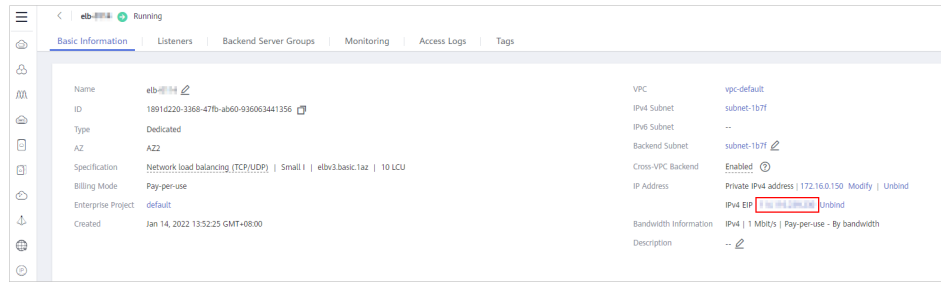
- Accessing a single node of a Redis Cluster instance on a client through ELB
  - a. View the basic information of the load balancer created in [Step 3](#).



- b. [Buy an ECS](#), log in to it, and install the Redis client by referring to [redis-cli](#).
- c. On the Redis client, connect to the DCS instance using the IP address and port number configured in [Step 4](#). If you use the EIP and port number of the load balancer, an error will be reported.

```
[root@~]# /usr/local/redis/redis-5.0.12/src/redis-cli -h 172.16.0.244 -p 80 -a cxx1234.
Warning: Using a password with '-a' or '-u' option on the command line interface may not be safe.
172.16.0.244:80> get name
(error) MOVED 5798 172.16.0.244:6379
172.16.0.244:80>
[root@~]# /usr/local/redis/redis-5.0.12/src/redis-cli -h 172.16.0.244 -p 6379 -a cxx1234.
Warning: Using a password with '-a' or '-u' option on the command line interface may not be safe.
172.16.0.244:6379> get name
(nil)
172.16.0.244:6379> set name china
OK
172.16.0.244:6379> get name
"china"
172.16.0.244:6379>
```

- Accessing a single-node, master/standby, read/write splitting, or Proxy Cluster instance on a client through ELB
  - a. View the IPv4 EIP and port number of the load balancer created in [Step 3](#).



- b. Buy an ECS, log in to it, and install the Redis client by referring to [redis-cli](#).
- c. Use redis-cli to access the load balancer using its EIP and port number (which is 80).

```
[root@ecs-... elb-test-0001 src]# /usr/local/redis/redis-5.0.12/src/redis-cli -h 101...196 -p 80 -n 0
```

- d. Write a key through ELB.

```
[root@ecs-... -test-0001 src]# /usr/local/redis/redis-5.0.12/src/redis-cli -h ... -p 80 -n 0
> get name
(nil)
> set name elb-test
OK
> get name
"elb-test"
```

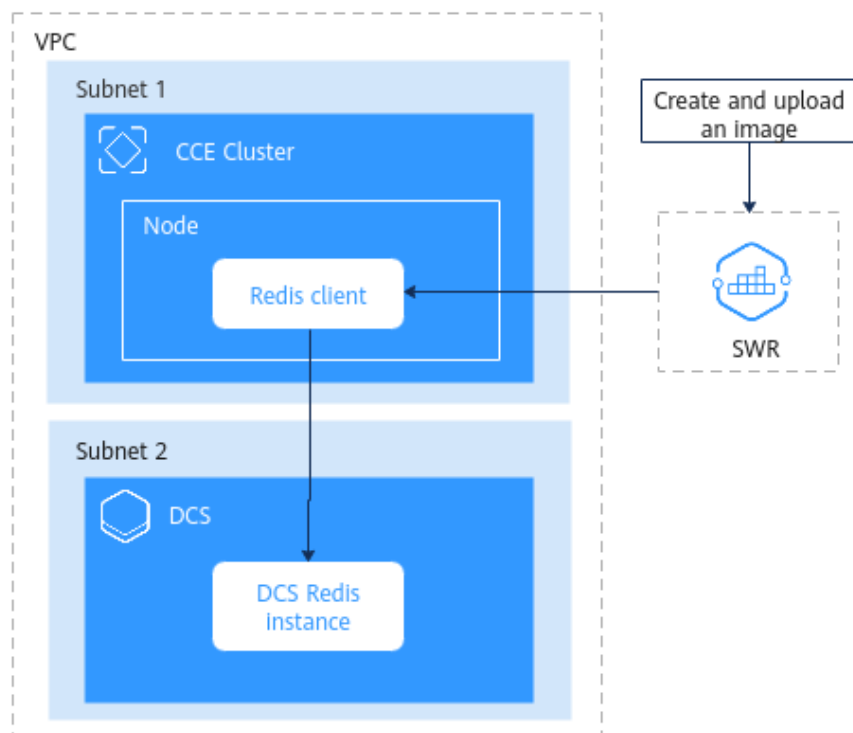
- e. Log in to the DCS console. On the **Cache Manager** page, choose **More > Connect to Redis** in the row that contains the DCS instance created in [Step 2](#). Check whether the key written in [d](#) exists.



## 3.4 Connecting a Client to DCS Through CCE

### Overview

With the development of the container technology, more and more applications are deployed in containers. This section describes how to deploy a Redis client in a Cloud Container Engine (CCE) cluster container and connect it to DCS.



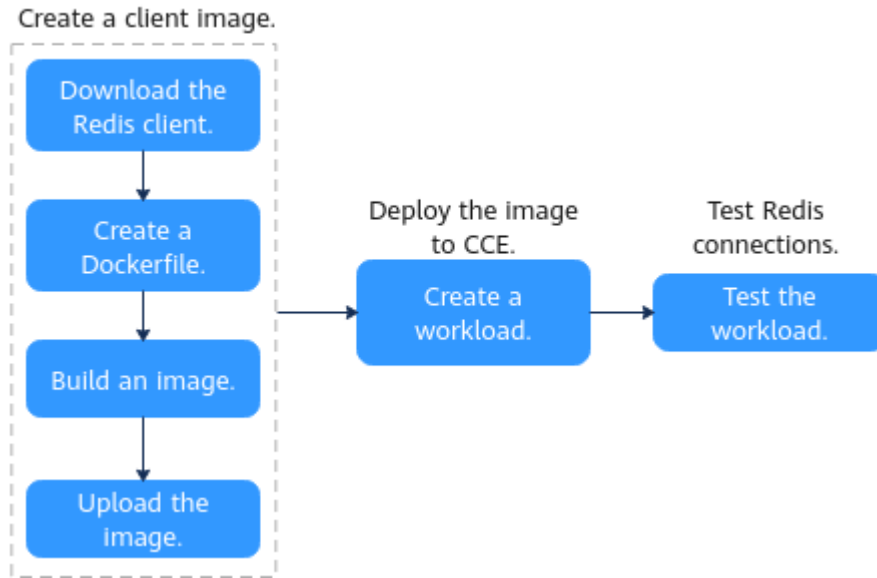
### Prerequisites

Prepare the following resources:

- **VPC and subnet**, for example, **vpc-test**. For details, see [Creating a VPC](#).  
(Optional) Create two subnets. Place your DCS instance in one subnet (subnet 1) and your CCE cluster in the other (subnet 2).
- **DCS instance**, for example, **dcs-test**. For details, see [Buying a DCS Redis Instance](#).  
When creating a DCS instance, select the created VPC (**vpc-test**) and subnet 1.
- **CCE cluster**, for example, **cce-test**. For details, see [Buying a CCE Cluster](#).  
When creating a CCE cluster, set **Network Model** to **VPC network**, and select VPC **vpc-test** and subnet 2.
- **CCE node pool**, for example, **cce-test-nodepool**. For details, see [Creating a Node Pool](#).

When creating a node pool, set **Node Type** to **Elastic Cloud Server (VM)**, **Container Engine** to **Docker**, **OS** to **CentOS 7.6**, and bind an existing EIP or create one.

## Procedure

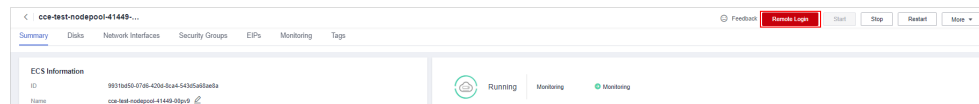


## Creating a Client Image

### Step 1 Download a Redis client.

1. Log in to the CCE cluster node.

Click the name of the created node pool. On the displayed page, click **Remote Login** in the upper right corner.



2. Run the **gcc --version** command to check whether the GCC compiler for compiling the Redis program is installed in the OS. The following figure shows that the GCC compiler has been installed.

```
[root@cce-test-nodepool-1148-03p9 ~]# gcc --version
gcc (GCC) 4.8.5 20150623 (Red Hat 4.8.5-44)
Copyright (C) 2015 Free Software Foundation, Inc.
This is free software; see the source for copying conditions. There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

If the GCC compiler is not installed, run the following commands to install it:

```
yum -y install gcc
yum -y install gcc-c++
```

3. Run the following command to create the **redis** directory in the **home** directory, and then go to the directory:  
`cd /home && mkdir redis && cd redis`
4. Run the following command to download the Redis client. The following takes version 5.0.13 as an example.



```
wget https://download.redis.io/releases/redis-5.0.13.tar.gz
```

```
[root@cce-test-nodepool-41449-88pv9 ~]# cd /home && mkdir redis && cd redis
[root@cce-test-nodepool-41449-88pv9 redis]# wget https://download.redis.io/releases/redis-5.0.13.tar.gz
--2022-12-14 13:59:19-- https://download.redis.io/releases/redis-5.0.13.tar.gz
Resolving download.redis.io (download.redis.io)... 45.88.125.1
Connecting to download.redis.io (download.redis.io)[45.88.125.1]:443... connected.
HTTP request sent, awaiting response... 288 OK
Length: 1995566 (1.9M) [Application/octet-stream]
Saving to: 'redis-5.0.13.tar.gz'
redis-5.0.13.tar.gz 100%[*****] 1.98M 4.35MB/s in 0.4s
2022-12-14 13:59:21 (4.35 MB/s) -- 'redis-5.0.13.tar.gz' saved [1995566/1995566]
```

5. Decompress the Redis package, go to the Redis directory, run the compilation command, and return to the Redis directory.

```
tar xvfz redis-5.0.13.tar.gz
cd redis-5.0.13 && make redis-cli
cd ..
```

### Step 2 Create a Dockerfile.

Run the **vim Dockerfile** command to create a Dockerfile and enter the following information:

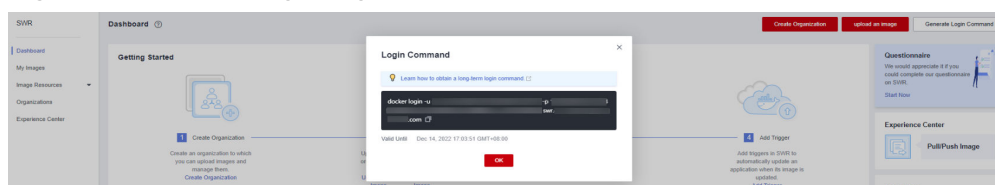
```
FROM centos:7
RUN useradd -d /home/redis -m redis
COPY ./redis-5.0.13 /home/redis/redis-5.0.13
RUN chown redis:redis /home/redis/redis-5.0.13 -R
USER redis
ENV HW_HOME=/home/redis/redis-5.0.13
ENV PATH=$HW_HOME/src:$PATH
WORKDIR /home/redis/
```

Press **Esc** to exit the editing mode and run **:wq!** to save the configuration and exit the editing interface.

### Step 3 Build a client image.

1. Choose **Software Repository for Container** from the service list to go to the SWR console.
2. Click **Create Organization** in the upper right corner and enter an organization name to create an organization. You can also use an existing organization. (Click **Organization Management** in the navigation pane to view organizations.)
3. On the SWR **Dashboard** page, click **Generate Login Command** in the upper right corner to obtain and copy the login command. (**swr.xxxxxx.com** at the end of the login command is the image repository address.)

Figure 3-16 Obtaining a login command



4. Run the copied login command on the CCE node to log in to SWR.

Figure 3-17 Logging in to SWR

```
[root@cce-test-nodepool-41449-88pv9 ~]# docker login -u
WARNING: Using --password via the CLI is insecure. Use --password-stdin.
WARNING: Your password will be stored unencrypted in /root/.docker/config.json.
Configure a credential helper to remove this warning. See
https://docs.docker.com/engine/reference/commandline/login/#credential-store
Login Succeeded
```

5. Run the following command to build an image:  
docker build -t {Image repository address}/{Organization name}/{Image name :version}.

*Image repository address* indicates the address of the image repository, which is at the end of the login command. *Organization name* indicates the name of

the organization created in **b**. *Image name* indicates the name of the image to be built. *version* indicates the image version. Replace them with the actual values. For example, **docker build -t swr.xxxxxx.com/study1/redis:v1**.

**Figure 3-18** Building an image

```

Login Succeeded
[root@cce-test-nodpool-41449-00pv9 redis]# docker build -t swr.xxxxxx.com/study1/redis:v1 .
Sending build context to Docker daemon 74.5MB
Step 1/8 : FROM centos:7
7: Pulling from library/centos
75f829a71a1c: Pull complete
Digest: sha256:fe2347002c630d5d61bf2f28f21246ad1c21cc6fd343e70b4cf1e5102f8711a9
Status: Downloaded newer image for centos:7
--> 7e6257c9f8d8
Step 2/8 : RUN useradd -d /home/redis -m redis
--> Running in 720605d929b0
Removing intermediate container 720605d929b0
--> d3cc4ace4bd9
Step 3/8 : COPY ./redis-5.0.13 /home/redis/redis-5.0.13
--> 139d612ff3e6
Step 4/8 : RUN chown redis:redis /home/redis/redis-5.0.13 -R
--> Running in af3e6e630d12
Removing intermediate container af3e6e630d12
--> c0cbea94f230
Step 5/8 : USER redis
--> Running in 491ac483837b
Removing intermediate container 491ac483837b
--> bd598f78f953
Step 6/8 : ENV HW_HOME=/home/redis/redis-5.0.13
--> Running in b17161cbb50c
Removing intermediate container b17161cbb50c
--> 15430e8aa760
Step 7/8 : ENV PATH=$HW_HOME/src:$PATH
--> Running in 5f96cc1791fa
Removing intermediate container 5f96cc1791fa
--> 2e553b91f2d3
Step 8/8 : WORKDIR /home/redis/
--> Running in 49ad2619419c
Removing intermediate container 49ad2619419c
--> 0749f3f4d04e
Successfully built 0749f3f4d04e
Successfully tagged swr.xxxxxx.com/study1/redis:v1
    
```

**Step 4** Run the following command to upload the client image to SWR:

```
docker push {Image repository address}/{Organization name}/{Image name :version}
```

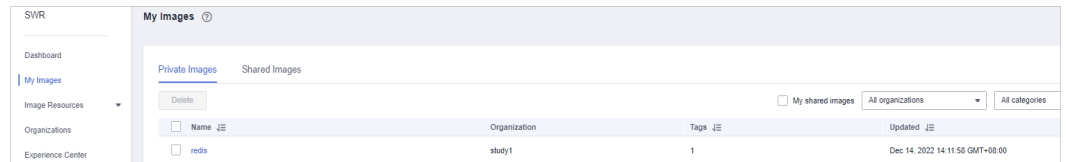
**Figure 3-19** Uploading an image

```

[root@cce-test-nodpool-41449-00pv9 redis]# docker push swr.xxxxxx.com/study1/redis:v1
The push refers to repository [swr.xxxxxx.com/study1/redis]
43a5f3e4aa76: Pushed
4bf8f59f51ac: Pushed
750a9898d5b0: Pushed
613be09ab3c0: Layer already exists
v1: digest: sha256:f9c2f5ad24e9ec5c06a7f3728f9917b94db5fb40055314b8e71a908d563f6948 size: 1161
    
```

**Step 5** After the image is uploaded, you can view it on the **My Images** page of the SWR console.

**Figure 3-20** Viewing images



----End

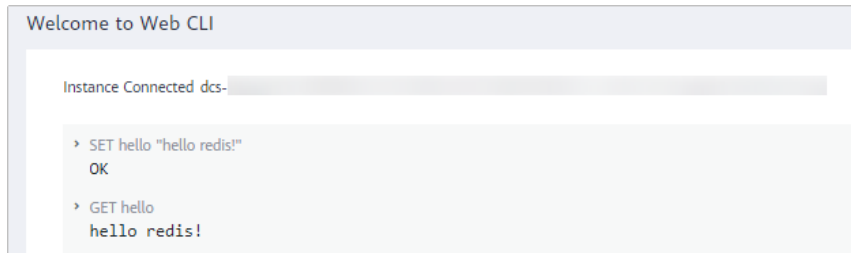
## Deploying an Image on CCE

**Step 1** On the DCS console, click the created Redis instance **dcs-test** to go to the details page.

**Step 2** In the **Connection** area, obtain the IP address and port number of the Redis instance.

**Step 3** Click **Connect to Redis** in the upper right corner to use the Web CLI.

**Step 4** On the Web CLI, run a **SET** command. The following figure uses **SET hello "hello redis!"** as an example.



**Step 5** On the CCE console, click the created CCE cluster **cce-test**.

**Step 6** In the navigation pane, choose **Workloads**. Click **Create Workload** in the upper right corner. For details, see [Creating a Workload](#).

- In the **Container Settings > Basic Info** area, set **Image Name** to the name of the created Redis image.
- In the **Container Settings > Lifecycle** area, configure the parameters as follows:

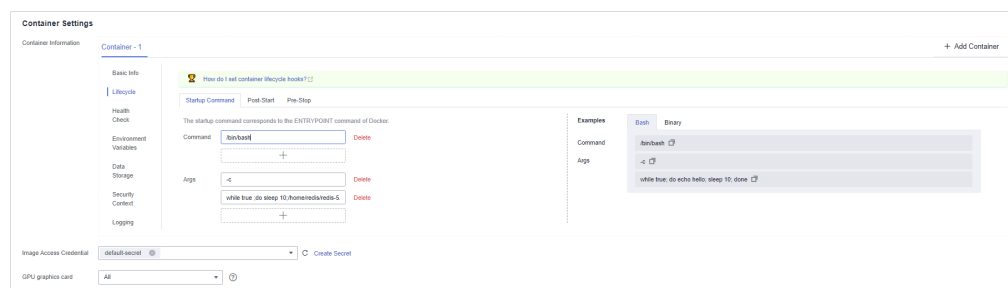
**Command:** `/bin/bash`

**Args:** `-c`

`while true ;do sleep 10;/home/redis/redis-5.0.13/src/redis-cli -h 10.0.0.0 -p 6379 -a DCS instance password get hello;done`

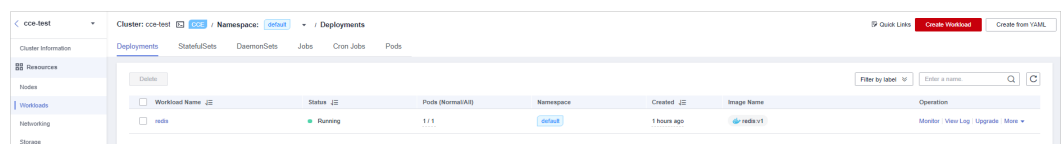
In the preceding command, **10.0.0.0** indicates the IP address of the DCS instance, **6379** indicates the port number of the DCS instance, **DCS instance password** indicates the password of the DCS instance, and **hello** indicates the data set when you connect to Redis through the Web CLI. Replace them with the actual values.

**Figure 3-21** Configuring lifecycle parameters



**Step 7** If the workload is in the **Running** state, it has been successfully created.

**Figure 3-22** Checking the workload status



----End

## Testing the Redis Connection

- Step 1** Log in to the CCE cluster node. For details, see [Step 1.1](#).
- Step 2** Download and configure the kubectl configuration file by referring to [Connecting to a Cluster Using kubectl](#).
- Step 3** Run the following command. If **Running** is returned, the Redis container is running.

```
kubectl get pod -n default
```

```
[root@cce-test-17441 home]# kubectl get pod -n default
NAME                READY   STATUS    RESTARTS   AGE
redis-595c6bf7c5-z7f2b  1/1     Running   0           35s
```

- Step 4** Run the following command to view the logs of the Redis container:

```
kubectl logs --tail 10 -f redis-xxxxxxx -n default
```

**redis-xxxxxxx** indicates the name of the created workload pod. (Click the workload name to view the workload pod name.)

```
[root@cce-test-17441 home]# kubectl logs --tail 10 -f redis-595c6bf7c5-z7f2b -n default
Warning: Using a password with '-a' or '-u' option on the command line interface may not be safe.
hello redis!
Warning: Using a password with '-a' or '-u' option on the command line interface may not be safe.
hello redis!
Warning: Using a password with '-a' or '-u' option on the command line interface may not be safe.
hello redis!
Warning: Using a password with '-a' or '-u' option on the command line interface may not be safe.
hello redis!
```

In the command output, the information returned by DCS is **hello redis!**, which is the data set when you [connected to Redis](#).

- Step 5** The test is complete.

----End

## 3.5 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 causes instance disconnection for 15 to 30s:

Scenario	Description
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 <a href="#">Modifying Specifications</a> .
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.
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.	<p>Timeout may occur in any of the following phases:</p> <ul style="list-style-type: none"><li>• A command is successfully sent by the client but has not reached Redis.</li><li>• The command has reached Redis, but the execution times out.</li><li>• Redis has executed the command, but the result returned to the client times out.</li></ul> <p>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 <b>SET</b> command. For example, if you run the <b>SET a b</b> command multiple times, the value of <b>a</b> can only be <b>b</b> or the execution fails. If you run <b>LPUSH mylist a</b>, which is not idempotent, <b>mylist</b> may contain multiple <b>a</b> elements.</p>

Retry Rule	Description
Configure proper retry times and interval.	<p>Configure the retry times and interval based on service requirements in actual scenarios to prevent the following problems:</p> <ul style="list-style-type: none"> <li>• If the number of retries is insufficient or the interval is too long, the application may fail to complete operations.</li> <li>• 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.</li> </ul> <p>Common retry interval policies include immediate retry, fixed-interval retry, exponential backoff retry, and random backoff retry.</p>
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.setMaxWaitMillis(2000);
    jedisPoolConfig.setMaxAttempts(5);
    return new JedisCluster(hostAndPortsSet, jedisPoolConfig);
}
```

**Table 3-1** Recommended Jedis connection pool parameter settings

Parameter	Description	Recommended Setting
maxTotal	Maximum number of connections	<p>Set this parameter based on the number of HTTP threads of the web container and reserved connections. Assume that the <b>maxConnections</b> parameter of the Tomcat Connector is set to <b>150</b> 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).</p> <p><b>Limit:</b> The value of <b>maxTotal</b> 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.</p> <p>For example, if <b>maxClients</b> of a master/standby DCS Redis instance is 10,000 and <b>maxTotal</b> of a single client is 500, the maximum number of clients is 20.</p>
maxIdle	Maximum number of idle connections	Use the same configuration as <b>maxTotal</b> .

Parameter	Description	Recommended Setting
minIdle	Minimum number of idle connections	<p>Generally, you are advised to set this parameter to 1/X of <b>maxTotal</b>. For example, the recommended value is <b>100</b>.</p> <p>In performance-sensitive scenarios, you can set this parameter to the value of <b>maxIdle</b> to prevent the impact caused by frequent connection quantity changes. For example, set this parameter to <b>400</b>.</p>
maxWaitMillis	Maximum waiting time for obtaining a connection, in milliseconds	<p>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.</p>
timeout	Command execution timeout, in milliseconds	<p>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 at least 210 ms to ensure network fault tolerance. For special detection logic or environment exception detection, you can adjust this timeout to seconds.</p>



Parameter	Description	Recommended Setting
minEvictableIdleTimeMillis	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 <b>-1</b> and check idle connections periodically.
timeBetweenEvictionRunsMillis	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 <b>ping</b> 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 <b>True</b> . Generally, you are advised to set this parameter to <b>False</b> to disable idle connection detection.

Parameter	Description	Recommended Setting
testWhileIdle	Indicates whether to use the <b>ping</b> 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 <b>ping</b> 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: <b>5</b> . Set this parameter based on the maximum timeout intervals of service APIs and a single request. The maximum value is <b>10</b> . If the value exceeds <b>10</b> , the processing time of a single request is too long, blocking other requests.

# 4 Usage Guide

---

## 4.1 DCS Data Security

Security is a shared responsibility between Huawei Cloud and you. Huawei Cloud is responsible for the security of cloud services to provide a secure cloud. As a tenant, you should properly use the security capabilities provided by cloud services to protect data, and securely use the cloud. For details, see [Shared Responsibilities](#).

This section provides actionable guidance for enhancing the overall security of using DCS. You can continuously evaluate the security status of your DCS resources, enhance their overall security defense by combining multiple security capabilities provided by DCS, and protect data stored in DCS from leakage and tampering both at rest and in transit.

Make security configurations from the following dimensions to meet your service needs.

- [Protecting Data Through Access Control](#)
- [Encrypting Data Before Storage](#)
- [Data Restoration and Disaster Recovery](#)
- [Transmission Encryption with SSL](#)
- [Checking for Abnormal Data Access](#)
- [Using the Latest SDKs for Better Experience and Security](#)
- [Using Other Cloud Services for Additional Data Security](#)

### Protecting Data Through Access Control

Correctly use the access control capability provided by DCS to prevent your data from being stolen or damaged.

1. **Set only the minimum permissions for IAM users with different roles to prevent data leakage or misoperations caused by excessive permissions.**  
To better isolate and manage permissions, you are advised to configure an independent IAM administrator and grant them the permission to manage IAM policies. The IAM administrator can create different user groups based on

your service requirements. User groups correspond to different data access scenarios. By adding users to user groups and binding IAM policies to user groups, the IAM administrator can grant different data access permissions to employees in different departments based on the principle of least privilege. For details, see [Permissions Management](#).

2. **Configure a whitelist or security group to protect your data from abnormal reads or other operations.**

By configuring an IP address whitelist or inbound and outbound security group rules, you can control the network range for connecting to your instance and prevent exposure to untrusted third parties.

DCS Redis 4.0/5.0/6.0 basic edition instances are controlled by whitelists. For details, see [Managing IP Address Whitelist](#).

DCS Redis 6.0 professional instances are controlled by security group rules. For details, see [How Do I Configure a Security Group?](#) Do not set the source to 0.0.0.0/0 in the inbound rules of a security group.

3. **Do not use high-risk commands, to prevent attackers from directly damaging Redis.**

You can rename high-risk commands to disable them if they are not used in your service. Learn more about [disabled commands](#) and [commands that can be renamed](#).

4. **Use a non-default port to prevent scanning attacks.**

The default listening port of the Redis server is 6379, which is vulnerable to scanning attacks. You can use a port in the range from 1 to 65535. For details, see [Customizing a Port](#).

5. **Limit the maximum number of client connections to avoid resource exhaustion and DoS risks.**

The **maxclients** parameter of Redis determines the maximum number of clients that can be concurrently connected to an instance. The default value is **10000**, and the value can range from **1000** to **50000**. Excess connection requests will be rejected.

**Set a proper client connection limit based on your application scenario.** For details about how to modify the **maxclients** parameter, see [Modifying Configuration Parameters](#).

6. **Limit the idle time of Redis connections based on service requirements.**

To prevent idle client connections from occupying resources for a long time, you can set the **timeout** parameter on the console. Client connections that remain idle for the period specified by this parameter will be closed. The default value of **timeout** is **0**, indicating that the server does not proactively disconnect idle clients. The value ranges from 0 to 7200, in seconds.

You are not advised to set it to **0**. For example, you can set it to 3,600 seconds. For details about how to modify the **timeout** parameter, see [Modifying Configuration Parameters](#).

7. **Configure a password for accessing your DCS instance to prevent unauthorized clients from operating it by mistake.** In this way, clients can be authenticated for access, improving instance security.

You can set a password when [buying an instance](#), or [reset the password](#) of an existing instance.

8. **Use different DCS instances for different services to prevent instance faults from affecting multiple services.**

## Encrypting Data Before Storage

RDB and AOF persistent files in open-source Redis do not support encryption. Therefore, DCS does not support data encryption. If you have sensitive data, please encrypt it before writing it to DCS.

## Data Restoration and Disaster Recovery

Build restoration and disaster recovery (DR) capabilities in advance to prevent data from being deleted or damaged by mistake in abnormal data processing scenarios.

1. **Enable automated instance backup to quickly restore data in abnormal scenarios.**

DCS instances can be backed up automatically or manually. The automated backup function is disabled by default. After it is enabled, you can restore backup data to the instance. Backup data of an instance is stored for a maximum of 7 days. For details about automated backup, see [Configuring a Backup Policy](#).

Manual backups are user-initiated full backups of instances. The backup data is stored in Huawei Cloud OBS buckets and removed upon deletion of the corresponding instance.

2. **Use cross-AZ replication for data DR.**

A master/standby or cluster DCS instance can be deployed within an AZ or across multiple AZs for HA. For cross-AZ deployment, DCS initiates and maintains data synchronization. High availability is achieved by having a standby node take over in the event that a failure occurs on the master node. When operations are read-heavy, you can use DCS Redis 4.0 or later instances that support read/write splitting, or cluster instances that have multiple replicas. DCS maintains data synchronization between the master and replicas. You can connect to different addresses of an instance to isolate read and write operations.

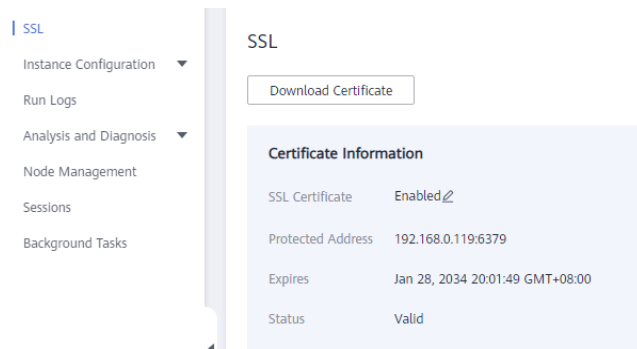
The screenshot shows the configuration interface for a DCS instance. The 'Version' is set to 5.0. The 'Instance Type' is set to 'Master/Standby'. The 'Replicas' are set to 2. The 'Primary AZ' is set to AZ5, and the 'Standby AZ' is set to AZ1. A red box highlights the 'Primary AZ' and 'Standby AZ' options, indicating cross-AZ deployment. A note at the bottom states: 'If your application has high availability requirements, deploy your instance across AZs to enhance fault tolerance.'

For cross-AZ deployment, DCS initiates and maintains data synchronization. High availability is achieved by having a standby node take over in the event that a failure occurs on the master node.

## Transmission Encryption with SSL

To prevent data from being stolen or damaged during transmission, use SSL encryption to access DCS.

Currently, only DCS for Redis 6.0 basic edition supports SSL encryption. You are advised to use DCS Redis 6.0 basic edition instances and enable SSL for them.



## Checking for Abnormal Data Access

1. **Enable Cloud Trace Service (CTS) to record all DCS access operations for future audit.**

CTS records operations on the cloud resources in your account. You can use the logs generated by CTS to perform security analysis, track resource changes, audit compliance, and locate faults.

After you enable CTS and configure a tracker, CTS can record management and data traces of DCS for auditing. For details, see [Viewing DCS Audit Logs](#).

2. **Use Cloud Eye for real-time monitoring and alarm reporting on security events.**

When using DCS, you may encounter error responses from the server. Huawei Cloud provides the Cloud Eye service to automatically monitor your DCS instances in real time, generate alarms, and send notifications, so that you can learn about the requests, traffic, and error responses of your DCS instances in real time.

Cloud Eye is enabled automatically after you create a DCS instance. For details, see [DCS Metrics](#) and [Configuring DCS Monitoring and Alarms](#).

## Using the Latest SDKs for Better Experience and Security

Upgrade SDKs to the latest version to better protect your data and DCS usage. Download the latest SDK in your desired language from [SDK Overview](#).

## Using Other Cloud Services for Additional Data Security

### Ensuring DCS resource security using SecMaster

SecMaster provides you with built-in checks that are included in Cloud Security Compliance Check 1.0, DJCP 2.0 Level 3 Requirements, and Network Security, to check key configurations of DCS, generate alarms for configurations with security risks, and provide hardening suggestions and guidelines. You can use the resource

management function of SecMaster to quickly learn about the DCS security status and locate security risks. For details, see [Baseline Inspection Overview](#).

## 4.2 Suggestions on Using DCS

### Service Usage

Principle	Description	Remarks
Deploy services nearby to reduce latency.	If your service and DCS instance are deployed far from each other (not in the same region) or with a high latency (connected through public networks), the read/write performance will be greatly affected by the latency.	If your service is latency-sensitive, do not create cross-AZ DCS Redis instances.
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 <b>SELECT</b> 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.
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 <a href="#">select a policy</a> that meets your service requirements. The default eviction policy used by DCS is <b>volatile-lru</b> .
Use Redis as cache.	Do not over-rely on Redis transactions.	After a transaction is executed, it cannot be rolled back.

Principle	Description	Remarks
	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.	-
	If the requested data is not found in Redis, read-only DB instances are accessed. You can use domain names to connect to read-only DB instances.	The idea is that the request does not go to the main database. You can use domain names to connect to multiple read-only DB instances. If a fault occurs, you can add such instances for emergency handling.
Do not use Redis as a message queue.	In pub/sub scenarios, do not use Redis as a message queue.	<ul style="list-style-type: none"> <li>• Unless otherwise required, you are not advised to use Redis as a message queue.</li> <li>• Using Redis as a message queue causes capacity, network, performance, and function issues.</li> <li>• If message queues are required, use Kafka for throughput and RocketMQ for reliability.</li> </ul>



Principle	Description	Remarks
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 <b>BGSAVE</b> 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: <i>service name.sub-service name.ID</i> .
	Minimize the key length.	Minimize the key length without compromising clarity of the meaning. Abbreviate common words. For example, <b>user</b> can be abbreviated to <b>u</b> , and <b>messages</b> can be abbreviated to <b>msg</b> .	Use up to 128 bytes. The shorter the better.

Category	Principle	Description	Remarks
	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.
	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 <b>HGETALL</b> , 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 <b>set u:1:name "X"</b> and <b>set u:1:age 20</b> . To save memory usage, you can also use the <b>HMSET</b> command to set multiple fields to their respective values in the hash stored at one key.

Category	Principle	Description	Remarks
	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 <b>HGETALL</b> , <b>LRange</b> , <b>SMembers</b> , <b>ZRange</b> , and <b>SInter</b> commands will consume a large number of CPU resources if there is a large number of elements. Alternatively, you can use <b>SCAN</b> sister commands, such as <b>HSCAN</b> , <b>SSCAN</b> , and <b>ZSCAN</b> commands.
Do not use high-risk commands.	Do not use high-risk commands such as <b>FLUSHALL</b> , <b>KEYS</b> , and <b>HGETALL</b> , or rename them.	For details, see <a href="#">Renaming Commands</a> .
Exercise caution when using the <b>SELECT</b> 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.	-

Principle	Description	Remarks
Use batch operations to improve efficiency.	For batch operations, use the <b>MGET</b> command, <b>MSET</b> command, or pipelining to improve efficiency, but do not include a large number of elements in one batch operation.	<p><b>MGET</b> command, <b>MSET</b> command, and pipelining differ in the following ways:</p> <ul style="list-style-type: none"> <li>• <b>MGET</b> and <b>MSET</b> are atomic operations, while pipelining is not.</li> <li>• Pipelining can be used to send multiple commands at a time, while <b>MGET</b> and <b>MSET</b> cannot.</li> <li>• Pipelining must be supported by both the server and the client.</li> </ul>
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.
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.	<ul style="list-style-type: none"> <li>• When the <b>EVAL</b> or <b>EVALSHA</b> 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."</li> <li>• When the <b>EVAL</b> or <b>EVALSHA</b> 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.</li> </ul>

Principle	Description	Remarks
Optimize multi-key operation commands such as <b>MGET</b> and <b>HMGET</b> 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 <b>MGET</b> 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.
Do not use the <b>DEL</b> command to directly delete big keys.	Deleting big keys, especially Sets, using <b>DEL</b> blocks other requests.	In Redis 4.0 and later, you can use the <b>UNLINK</b> command to delete big keys safely. This command is non-blocking.  In versions earlier than Redis 4.0: <ul style="list-style-type: none"> <li>• To delete big Hashes, use <b>HSCAN + HDEL</b> commands.</li> <li>• To delete big Lists, use the <b>LTRIM</b> command.</li> <li>• To delete big Sets, use <b>SSCAN + SREM</b> commands.</li> <li>• To delete big Sorted Sets, use <b>ZSCAN + ZREM</b> commands.</li> </ul>

## 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.

Principle	Description	Remarks
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 <a href="#">Configuring Redis Client Retry</a> .
Set appropriate interval and number of retries.	Do not set the retry interval too short or too long.	<ul style="list-style-type: none"> <li>• If the retry interval is very short, for example, shorter than 200 milliseconds, a retry storm may occur, and can easily cause service avalanche.</li> <li>• 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.</li> </ul>

Principle	Description	Remarks
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.	<p>Lettuce has the following problems:</p> <ul style="list-style-type: none"> <li>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. For details, see <a href="#">How Do I Handle an Error When I Use Lettuce to Connect to a Redis Cluster Instance After Specification Modification?</a></li> <li>Lettuce cannot validate connections in the connection pool. If an invalid connection is used, services will fail and may become unavailable in minutes.</li> </ul>

## 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.	-
Verify the fault handling capability or disaster recovery logic 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.

Principle	Description	Remarks
Configure monitoring.	Pay attention to the Redis capacity and expand it before overload.	Configure CPU, memory, and bandwidth alarms based on the alarm thresholds.
Perform routine health checks.	Perform routine checks on the memory usage of each node and whether the memory usage of the master nodes is balanced.	If memory usage is unbalanced, big keys exist and need to be split and optimized.
	Perform routine analysis on hot keys and check whether there are frequently accessed keys.	-
	Perform routine diagnosis on Redis commands and check whether O(N) commands have potential risks.	Even if an O(N) command is not time-consuming, it is recommended that R&D engineers analyze whether the value of N will increase with service growth.
	Perform routine analysis on slow query logs.	Detect potential risks based on slow query logs and rectify faults as soon as possible.

## 4.3 Detecting and Handling Big Keys and Hot Keys

### Definitions of Big Keys and Hot Keys

 NOTE

The definitions are for reference only. The actual service scenarios must be considered when you define big keys and hot keys.

Term	Definition
Big key	<p>There are two types of big keys:</p> <ul style="list-style-type: none"> <li>Keys that have a large value. If the size of a single String key exceeds 10 KB, or if the size of all elements of a key combined exceeds 50 MB, the key is defined as a big key.</li> <li>Keys that have a large number of elements. If the number of elements in a key exceeds 5000, the key is defined as a big key.</li> </ul>



Term	Definition
Hot key	<p>A hot key is most frequently accessed, or consumes significant resources. For example:</p> <ul style="list-style-type: none"> <li>• In a cluster instance, a shard processes 10,000 requests per second, among which 3000 are performed on the same key.</li> <li>• In a cluster instance, a shard uses a total of 100 Mbits/s inbound and outbound bandwidth, among which 80 Mbits/s is used by the <b>HGETALL</b> operation on a Hash key.</li> </ul>

## Impact of Big Keys and Hot Keys

Category	Impact
Big key	<p><b>Instance specifications fail to be modified.</b></p> <p>Specification modification of a Redis Cluster instance involves rebalancing (data migration between nodes). Redis has a limit on key migration. If the instance has any single key bigger than 512 MB, the modification will fail when big key migration between nodes times out. The bigger the key, the more likely the migration will fail.</p>
	<p><b>Data migration fails.</b></p> <p>During data migration, if a key has many elements, other keys will be blocked and will be stored in the memory buffer of the migration ECS. If they are blocked for a long time, the migration will fail.</p>
	<p><b>Cluster shards are unbalanced.</b></p> <ul style="list-style-type: none"> <li>• The memory usage of shards is unbalanced. For example, if a shard uses a large memory or even uses up the memory, keys on this shard are evicted, and resources of other shards are wasted.</li> <li>• The bandwidth usage of shards is unbalanced. For example, flow control is repeatedly triggered on a shard.</li> </ul>
	<p><b>Latency of client command execution increases.</b></p> <p>Slow operations on a big key block other commands, resulting in a large number of slow queries.</p>
	<p><b>Flow control is triggered on the instance.</b></p> <p>Frequently reading data from big keys exhausts the outbound bandwidth of the instance, triggering flow control. As a result, a large number of commands time out or slow queries occur, affecting services.</p>

Category	Impact
	<p><b>Master/standby switchover is triggered.</b></p> <p>If the high-risk <b>DEL</b> operation is performed on a big key, the master node may be blocked for a long time, causing a master/standby switchover.</p>
Hot key	<p><b>Cluster shards are unbalanced.</b></p> <p>If only the shard where the hot key is located is busy processing service queries, there may be performance bottlenecks on a single shard, and the compute resources of other shards may be wasted.</p>
	<p><b>CPU usage surges.</b></p> <p>A large number of operations on hot keys may cause high CPU usage. If the operations are on a single cluster shard, the CPU usage of the shard where the hot key is located will surge. This will slow down other requests and the overall performance. If the service volume increases sharply, a master/standby switchover may be triggered.</p>
	<p><b>Cache breakdown may occur.</b></p> <p>If Redis cannot handle the pressure on hot keys, requests will hit the database. The database may break down as its load increases dramatically, affecting other services.</p>

Big keys and hot keys can be avoided through proper design. For details, see [Suggestions on Using DCS](#).

## Detecting Big Keys and Hot Keys

Method	Description
Through <b>Big Key Analysis</b> and <b>Hot Key Analysis</b> on the DCS console	See <a href="#">Analyzing Big Keys and Hot Keys</a> .

Method	Description
<p>By using the <b>bigkeys</b> and <b>hotkeys</b> options on redis-cli</p>	<ul style="list-style-type: none"> <li>redis-cli uses the <b>bigkeys</b> option to traverse all keys in a Redis instance and returns the overall key statistics and the biggest key of six data types: Strings, Lists, Hashes, Sets, Zsets, and Streams. The command is <b>redis-cli -h &lt;Instance connection address&gt; -p &lt;Port number&gt; -a &lt;Password&gt; --bigkeys</b>.</li> <li>In Redis 4.0 and later, you can use the <b>hotkeys</b> option to quickly find hot keys in redis-cli. Run this command during service running to find hot keys: <b>redis-cli -h &lt;Instance connection address&gt; -p &lt;Port number&gt; -a &lt;Password&gt; --hotkeys</b>. The hot key details can be obtained from the summary part in the returned result.</li> </ul>
<p>Searching for big keys using Redis commands</p>	<p>If there is a pattern of big keys, for example, the prefix is <b>cloud:msg:test</b>, you can use a program to scan for keys that match the prefix, and then run commands to query the number of members in the key and query the key sizes to find big keys.</p> <ul style="list-style-type: none"> <li>Commands for querying the number of members: <b>LLEN, HLEN, XLEN, ZCARD, SCARD</b></li> <li>Commands for querying the memory usage of keys: <b>DEBUG OBJECT, MEMORY USAGE</b></li> </ul> <p><b>CAUTION</b> This method consumes a large number of computing resources. To ensure service running, do not use this method on instances with heavy service pressure.</p>
<p>Searching for big keys using redis-rdb-tools</p>	<p><b>redis-rdb-tools</b> is an open-source tool for analyzing Redis RDB snapshot files. You can use it to analyze the memory usage of all keys in a Redis instance.</p> <p>To use this method, you must <b>export the RDB file of an instance</b> on the <b>Backups &amp; Restorations</b> page of the DCS console.</p> <p><b>CAUTION</b> This method does not affect service running, but is not as timely as online analysis.</p>

## Optimizing Big Keys and Hot Keys

Category	Method
Big key	<p><b>Split big keys.</b></p> <p>Scenarios:</p> <ul style="list-style-type: none"> <li>• <b>If the big key is a String</b>, you can split it into several key-value pairs and use <b>MGET</b> or a pipeline consisting of multiple <b>GET</b> operations to obtain the values. In this way, the pressure of a single operation can be split. For a cluster instance, the operation pressure can be evenly distributed to multiple shards, reducing the impact on a single shard.</li> <li>• <b>If the big key contains multiple elements, and the elements must be operated together</b>, the big key cannot be split. You can remove the big key from Redis and store it on other storage media instead. This scenario should be avoided by design.</li> <li>• <b>If the big key contains multiple elements, and only some elements need to be operated each time</b>, separate the elements. Take a Hash key as an example. Each time you run the <b>HGET</b> or <b>HSET</b> command, the result of the hash value modulo <math>N</math> (customized on the client) determines which key the field falls on. This algorithm is similar to that used for calculating slots in Redis Cluster.</li> </ul>
	<p><b>Store big keys on other storage media.</b></p> <p>If a big key cannot be split, it is not suitable to be stored in Redis. You can store it on other storage media, such as <b>SFS</b> or other NoSQL databases, and delete the big key from Redis.</p> <p><b>CAUTION</b> Do not use the <b>DEL</b> command to delete big keys. Otherwise, Redis may be blocked or even a master/standby switchover may occur. The <b>UNLINK</b> command can be used to delete big keys in Redis 4.0 and later.</p>
	<p><b>Set appropriate expiration and delete expired data regularly.</b></p> <p>Appropriate expiration prevents expired data from remaining in Redis. Due to Redis's lazy free, expired data may not be deleted in time. If this occurs, <a href="#">scan expired keys</a>.</p>
Hot key	<p><b>Split read and write requests.</b></p> <p>If a hot key is frequently read, <a href="#">configure read/write splitting</a> on the client to reduce the impact on the master node. You can also add replicas to meet the read requirements, but there cannot be too many replicas. In DCS, replicas replicate data from the master in parallel. The replicas are independent of each other and the replication delay is short. However, if there is a large number of replicas, CPU usage and network load on the master node will be high.</p>

Category	Method
	<p><b>Use the client cache or local cache.</b></p> <p>If you know what keys are frequently used, you can design a two-level cache architecture (client/local cache and remote Redis). Frequently used data is obtained from the local cache first. The local cache and remote cache are updated with data writes at the same time. In this way, the read pressure on frequently accessed data can be separated. This method is costly because it requires changes to the client architecture and code.</p>
	<p><b>Design a circuit breaker or degradation mechanism.</b></p> <p>Hot keys can easily result in cache breakdown. During peak hours, requests are passed through to the backend database, causing service avalanche. To ensure availability, the system must have a circuit breaker or degradation mechanism to limit the traffic and degrade services if breakdown occurs.</p>

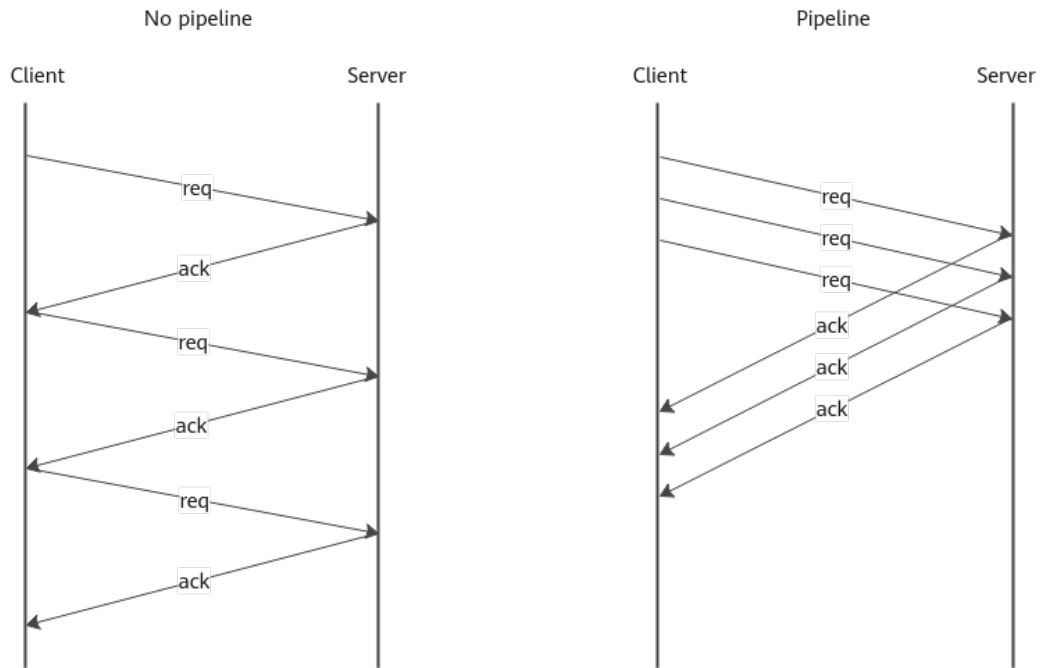
## 4.4 Configuring a Redis Pipeline

### Overview

DCS supports Redis pipelining. This technique sends multiple commands to the Redis server at once, reducing network latency and improving performance.

Without a pipeline, a client sends a command to Redis, waits for the server to return the result, then sends the next command, and so on. With a pipeline, a client sends commands to Redis without waiting for the results. After all commands are sent, the client closes the request, starts to receive responses, and matches them to the commands in sequence.

**Figure 4-1** Comparing the network communication with and without a pipeline



Generally in pipelining, Redis clients send commands in batches, receive all results, and then return them to upper-layer services. This mechanism reduces the network round-trip time (RTT), system calls of read() and write(), and process switchovers, and improves program efficiency and performance.

Use pipelining to perform Redis operations in batches for better performance, if your services does not need to obtain the result of each operation immediately.

**NOTE**

- Pipelines exclusively use the client-server connection. Other operations cannot be performed until the pipelines are closed. To perform other operations at the same time, set up a connection dedicated to pipelines.
- For more information, see [Redis pipelining](#).

**Notes and Constraints**

- Pipelining cannot ensure atomicity.  
Pipelining sends client commands in batches. The server parses each command and executes them one by one in sequence. During this process, the server may execute commands from other clients. For atomicity, use transactions or Lua scripts.
- Pipelining does not support rollback if an error occurs.
- Pipelining does not feature transactions. Do not use it if commands depend on each other.  
Some clients, such as redis-py, wrap pipelines in transactional commands **MULTI** and **EXEC**. Note the differences between pipelines and transactions. For restrictions on transactions, see [Redis transactions](#).
- Due to buffer limits of the server and some clients, do not use many commands in a single pipeline.

- There are constraints in the cluster Redis architecture. For example, keys cannot be accessed across slots in a single command; the "-MOVED" error occurs when data that is not on the current node is accessed. Therefore, ensure commands in pipelines are executable when you use pipelining in cluster architecture. For more information, see [Command Restrictions](#).

## Comparing Performance

The following code compares the performance with and without a pipeline.

```
public static void main(String[] args) {
    // Set the Redis instance connection address and port.
    Jedis jedis = new Jedis("127.0.0.1", 6379);

    // Run commands consecutively.
    final int COUNT=5000;
    String key = "key";

    // 1 ---No pipelines are used.---
    jedis.del(key); // Initialize the key.
    long t1 = System.currentTimeMillis();
    for (int i = 0; i < COUNT; i++) {
        // Send a request and receive a response.
        jedis.incr(key);
    }
    long t2 = System.currentTimeMillis();
    System.out.println("No Pipeline > value:"+jedis.get(key)+" > Duration:" + (t2 - t1) + "ms");

    // 2 ----A pipeline is used.---
    jedis.del(key); // Initialize the key.
    Pipeline p1 = jedis.pipelined();
    long t3 = System.currentTimeMillis();
    for (int i = 0; i < COUNT; i++) {
        // Send a request.
        p1.incr(key);
    }
    // Receive a response.
    p1.sync();
    long t4 = System.currentTimeMillis();

    System.out.println("Pipeline used > value:"+jedis.get(key)+" > Duration:" + (t4 - t3)+ "ms");
    jedis.close();}
```

The result shows that the performance is better when a pipeline is used.

```
No pipeline > value:5000 > Duration:1204ms
Pipeline used > value:5000 > Duration:9ms
```

## Processing the Response Data

The following code shows two methods of processing the response data when a pipeline is used in Jedis.

```
public static void main(String[] args) {
    // Set the Redis instance connection address and port.
    Jedis jedis = new Jedis("127.0.0.1", 6379);
    String key = "key";
    jedis.del(key); // Initialize

    // ----- Method 1 -----
    Pipeline p1 = jedis.pipelined();
    System.out.println("-----Method 1-----");
    for (int i = 0; i < 5; i++) {
        p1.incr(key);
        System.out.println("Pipeline sends a request.");
    }
}
```

```
// The request is sent. Start receiving the response.
System.out.println("The request is sent. Start receiving the response.");
List<Object> responses = p1.syncAndReturnAll();
if (responses == null || responses.isEmpty()) {
    jedis.close();
    throw new RuntimeException("Pipeline error: no response.");
}
for (Object resp : responses) {
    System.out.println("Pipeline receives a response: " + resp.toString());
}
System.out.println();

// ----- Method 2 -----
System.out.println("-----Method 2-----");
jedis.del(key); // Initialize
Pipeline p2 = jedis.pipelined();

// Declare a response.
Response<Long> r1 = p2.incr(key);
System.out.println("Pipeline sends a request.");
Response<Long> r2 = p2.incr(key);
System.out.println("Pipeline sends a request.");
Response<Long> r3 = p2.incr(key);
System.out.println("Pipeline sends a request.");
Response<Long> r4 = p2.incr(key);
System.out.println("Pipeline sends a request.");
Response<Long> r5 = p2.incr(key);
System.out.println("Pipeline sends a request.");
try {
    r1.get(); // An exception is thrown because the response is still pending.
} catch (Exception e) {
    System.out.println("<<< Pipeline error: Receiving the response has not started yet. >>> ");
}
// The request is sent and responses start to be received.
System.out.println("The request is sent. Start receiving the response.");
p2.sync();
System.out.println("Pipeline receives the response: " + r1.get());
System.out.println("Pipeline receives the response: " + r2.get());
System.out.println("Pipeline receives the response: " + r3.get());
System.out.println("Pipeline receives the response: " + r4.get());
System.out.println("Pipeline receives the response: " + r5.get());
jedis.close();}
```

### Result:

```
-----Method 1-----
Pipeline sends a request.
Pipeline sends a request.
Pipeline sends a request.
Pipeline sends a request.
Pipeline sends a request.
The request is sent. Start receiving the response.
Pipeline receives the response: 1
Pipeline receives the response: 2
Pipeline receives the response: 3
Pipeline receives the response: 4
Pipeline receives the response: 5
-----Method 2-----
Pipeline sends a request.
Pipeline sends a request.
Pipeline sends a request.
Pipeline sends a request.
Pipeline sends a request.
<<< Pipeline error: Receiving the response has not started yet. >>>
The request is sent. Start receiving the response.
Pipeline receives the response: 1
Pipeline receives the response: 2
Pipeline receives the response: 3
Pipeline receives the response: 4
Pipeline receives the response: 5
```



## 4.5 Optimizing the Jedis Connection Pool

### Overview

JedisPool is the connection pool of the Jedis client. This section describes how to configure JedisPool for better Redis performance and resource utilization.

### Using JedisPool

The following Maven dependency is for Jedis 5.1.3.

```
<dependency>
  <groupId>redis.clients</groupId>
  <artifactId>jedis</artifactId>
  <version>5.1.3</version>
</dependency>
```

Jedis manages resource pools using Apache Commons-pool2. The key parameter **GenericObjectPoolConfig** (resource pool) is required to define JedisPool. This parameter can be used as follows. For details, see [JedisPool Parameters](#).

```
GenericObjectPoolConfig jedisPoolConfig = new GenericObjectPoolConfig();
jedisPoolConfig.setMaxTotal(...);
jedisPoolConfig.setMaxIdle(...);
jedisPoolConfig.setMinIdle(...);
jedisPoolConfig.setMaxWaitMillis(...);
```

JedisPool is initialized as follows:

```
// redisHost is the IP address of the Redis instance. redisPort is the port of the Redis instance. redisPassword
is the password of the Redis instance. timeout is the connection or read/write timeout.
JedisPool jedisPool = new JedisPool(jedisPoolConfig, redisHost, redisPort, timeout, redisPassword);

// Execution
Jedis jedis = null;
try {
  jedis = jedisPool.getResource();
  // Specific command
  jedis.set("key", "value");
} catch (Exception e) {
  logger.error(e.getMessage(), e);
} finally {
  // With JedisPool, Jedis connections will be returned to the resource pool.
  if (jedis != null)
    jedis.close();
}
```

### JedisPool Parameters

JedisPool manages Jedis connections in a connection pool, which ensures resource control and thread security. **GenericObjectPoolConfig** can improve Redis performance at lower costs. [Table 4-1](#) and [Table 4-2](#) describe parameters and their configuration suggestions.

**Table 4-1** Parameters related to resource configuration and utilization

Parameter	Description	Default Value	Suggestion
maxTotal	Maximum number of connections in a resource pool	8	<a href="#">Suggestions on Key Parameters</a>
maxIdle	Maximum number of idle connections allowed in a resource pool	8	<a href="#">Suggestions on Key Parameters</a>
minIdle	Minimum number of idle connections allowed in a resource pool	0	<a href="#">Suggestions on Key Parameters</a>
blockWhenExhausted	Whether a caller awaits when a resource pool is used up <ul style="list-style-type: none"> <li>• <b>true</b>: yes</li> <li>• <b>false</b>: no</li> </ul> <b>maxWaitMillis</b> is valid only when this parameter is set to <b>true</b> .	true	Use the default value.
maxWaitMillis	Maximum time that a caller waits after a resource pool is used up, in milliseconds. The value <b>-1</b> indicates that the caller keeps waiting.	-1	Specify a time.
testOnBorrow	Whether to test validity of connections borrowed from a resource pool (ping). Invalid connections will be removed. <ul style="list-style-type: none"> <li>• <b>true</b>: yes</li> <li>• <b>false</b>: no</li> </ul>	false	Set to <b>false</b> during heavy service hours, saving resources a ping.
testOnReturn	Whether to test validity of connections returned to a resource pool (ping). Invalid connections will be removed. <ul style="list-style-type: none"> <li>• <b>true</b>: yes</li> <li>• <b>false</b>: no</li> </ul>	false	Set to <b>false</b> during heavy service hours, saving resources a ping.
jmxEnabled	Whether to enable JMX monitoring. <ul style="list-style-type: none"> <li>• <b>true</b>: yes</li> <li>• <b>false</b>: no</li> </ul>	true	Enable it, also for the application.

**Table 4-2** describes the parameters for testing idle Jedis objects.

**Table 4-2** Idle resource testing parameters

Parameter	Description	Default Value	Suggestion
testWhileIdle	Whether to test validity of idle connections using ping. Invalid ones will be destroyed.	false	true
timeBetweenEvictionRunsMillis	Interval of looking for idle resources, in milliseconds. -1: disabled	-1	Specify an interval, use the default value, or use the <b>JedisPoolConfig</b> .
minEvictableIdleTimeMillis	Minimum idle duration in a resource pool, in milliseconds. Resources that are idle longer than this will be removed.	1,800,000 (30 minutes)	Specify it as required. Generally, use the default value. <b>JedisPoolConfig</b> can be used.
numTestsPerEvictionRun	Number of idle resources to be tested each time.	3	Adjust this parameter as required. The value -1 indicates that all idle connections are tested.

Jedis provides **JedisPoolConfig** which inherits certain idle connection testing settings of **GenericObjectPoolConfig**.

```
public class JedisPoolConfig extends GenericObjectPoolConfig {
    public JedisPoolConfig() {
        setTestWhileIdle(true);
        setMinEvictableIdleTimeMillis(60000);
        setTimeBetweenEvictionRunsMillis(30000);
        setNumTestsPerEvictionRun(-1);
    }
}
```

 **NOTE**

All of the default values are available in **org.apache.commons.pool2.impl.BaseObjectPoolConfig**.

## Suggestions on Key Parameters

- **maxTotal**  
Consider the following factors:

- The Redis concurrency required by services
- Execution duration on a client
- Redis resources, such as the number of shards
- **maxTotal** must be within the maximum number of Redis connections. For details about the maximum Redis connections, see [How Do I View Current Concurrent Connections and Maximum Connections of a DCS Redis Instance?](#).
- Resource overhead. For example, control idle connections while avoiding frequently releasing and creating connections.

Assume that the average duration of executing a command is 1 ms. This execution covers the resource borrowing and returning, and network transmission. The QPS of a connection is about 1000 (1s/1 ms). The expected QPS per instance is 50,000 (Total service QPS/Number of Redis shards). Ideally, the required resource pool size (**maxTotal**) is 50 (50,000/1000).

In reality, more resources need to be reserved. Therefore, **maxTotal** can be greater. However, a large **maxTotal** allows many connections, which consume client and server resources. Furthermore, large commands with high QPS may block Redis and a large resource pool does not work.

- **maxIdle** and **minIdle**

**maxIdle** is the maximum number of connections required by services. **maxTotal** reserves resources. Do not use tiny **maxIdle**. Otherwise, new connection overhead occurs. **minIdle** controls idle resource testing.

Setting **maxIdle** to the same as **maxTotal** achieves an optimal connection pool and avoids performance impact caused by connection pool scaling. This combination fits in peak service hours, but causes resource wastes when the number of concurrent connections is small or **maxIdle** is excessively high.

The connection pool size of a node can be estimated based on the total QPS and the client scale.

- **Using monitoring for appropriate values**

In actual environments, an optimal parameter can be obtained through monitoring, such as JMX.

## Common Errors

- **Insufficient resources**

The following two cases indicate that resources cannot be obtained from the pool. This is not necessarily due to small resource pools (see [Suggestions on Key Parameters](#)). The network, resource pool parameter settings, resource pool monitoring (JMX), code (for example, `jedis.close()` is not executed), slow queries, and DNS may also be the cause.

- a. **Timeout:**

```
redis.clients.jedis.exceptions.JedisConnectionException: Could not get a resource from the pool
...Caused by: java.util.NoSuchElementException: Timeout waiting for idle object
at org.apache.commons.pool2.impl.GenericObjectPool.borrowObject
```

- b. **When a resource pool is used up, it does not wait for resource release if **blockWhenExhausted** is set to **false**:**

```
redis.clients.jedis.exceptions.JedisConnectionException: Could not get a resource from the pool
...Caused by: java.util.NoSuchElementException: Pool exhausted
at org.apache.commons.pool2.impl.GenericObjectPool.borrowObject
```

- **JedisPool preheating**

A project may time out after start for some reasons (for example, small timeout interval). When the maximum number of resources and minimum number of idle resources are defined by JedisPool, Jedis connections are not created in the connection pool. In initial use, if no resource in the pool is used, a new Jedis connection is created and then added to the resource pool. This process takes some time. Therefore, you are advised to preheat JedisPool based on the minimum number of idle connections after defining JedisPool. The following is an example:

```
List<Jedis> minIdleJedisList = new ArrayList<Jedis>(jedisPoolConfig.getMinIdle());
for (int i = 0; i < jedisPoolConfig.getMinIdle(); i++) {
    Jedis jedis = null;
    try {
        jedis = pool.getResource();
        minIdleJedisList.add(jedis);
        jedis.ping();
    } catch (Exception e) {
        logger.error(e.getMessage(), e);
    } finally {
    }
}
for (int i = 0; i < jedisPoolConfig.getMinIdle(); i++) {
    Jedis jedis = null;
    try {
        jedis = minIdleJedisList.get(i);
        jedis.close();
    } catch (Exception e) {
        logger.error(e.getMessage(), e);
    } finally {
    }
}
```