Distributed Cache Service

Best Practices

Issue	01
Date	2024-02-27

HUAWEI CLOUD COMPUTING TECHNOLOGIES CO., LTD.

Trademarks and Permissions


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

1 Serializing Access to Frequently Accessed Resources

Background

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

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

Solution

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

The idea of a distributed lock is to provide a globally unique "thing" for different systems to obtain 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

- You have created a Windows ECS.
- You have installed **JDK1.8** (or later) and a development tool (**Eclipse** is used as an example) on the ECS, and downloaded the **Jedis client**.
- You have created a DCS instance and configured the same VPC and subnet for the DCS instance and the ECS.

Procedure

Step 1 Run Eclipse on the ECS 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 {
    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 {
            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 **CaseTest.java** is as follows:

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

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

class ServiceOrder {
```



```
    private final int MAX = 10;

    DistributedLock DLock = new DistributedLock();

    int n = 10;

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

class ThreadBuy extends Thread {
    private ServiceOrder service;

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

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

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

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

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

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

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

```
Processing order for user e04934ddThread-5
Processing order for user a4554180Thread-0
User a4554180Thread-0 is allocated number 2 mobile phone. Number of mobile phones left: 8.
Processing order for user b58eb811Thread-10
User b58eb811Thread-10 is allocated number 3 mobile phone. Number of mobile phones left: 7.
```



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

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

```
Processing order for user eee56fb7Thread-16
User eee56fb7Thread-16 is allocated number 1 mobile phone. Number of mobile phones left: 9.
Processing order for user d6521816Thread-2
User d6521816Thread-2 is allocated number 2 mobile phone. Number of mobile phones left: 8.
Processing order for user d7b3b983Thread-19
User d7b3b983Thread-19 is allocated number 3 mobile phone. Number of mobile phones left: 7.
Processing order for user 36a6b97aThread-15
User 36a6b97aThread-15 is allocated number 4 mobile phone. Number of mobile phones left: 6.
Processing order for user 9a973456Thread-1
User 9a973456Thread-1 is allocated number 5 mobile phone. Number of mobile phones left: 5.
Processing order for user 03f1de9aThread-14
User 03f1de9aThread-14 is allocated number 6 mobile phone. Number of mobile phones left: 4.
Processing order for user 2c315ee6Thread-11
User 2c315ee6Thread-11 is allocated number 7 mobile phone. Number of mobile phones left: 3.
Processing order for user 2b03b7c0Thread-12
User 2b03b7c0Thread-12 is allocated number 8 mobile phone. Number of mobile phones left: 2.
Processing order for user 75f25749Thread-0
User 75f25749Thread-0 is allocated number 9 mobile phone. Number of mobile phones left: 1.
Processing order for user 26c71db5Thread-18
User 26c71db5Thread-18 is allocated number 10 mobile phone. Number of mobile phones left: 0.
Processing order for user c32654dbThread-17
User c32654dbThread-17 order failed.
Processing order for user df94370aThread-7
User df94370aThread-7 order failed.
Processing order for user 0af94cddThread-5
User 0af94cddThread-5 order failed.
Processing order for user e52428a4Thread-13
User e52428a4Thread-13 order failed.
Processing order for user 46f91208Thread-10
User 46f91208Thread-10 order failed.
Processing order for user e0ca87bbThread-9
```



```
User e0ca87bbThread-9 order failed.
Processing order for user f385af9aThread-8
User f385af9aThread-8 order failed.
Processing order for user 46c5f498Thread-6
User 46c5f498Thread-6 order failed.
Processing order for user 935e0f50Thread-3
User 935e0f50Thread-3 order failed.
Processing order for user d3eaae29Thread-4
User d3eaae29Thread-4 order failed.
```

----End

2 Ranking with Redis

The best practice for DCS guides you through ranking using DCS.

Scenario

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 cache, so read/write is fast.
- Multiple types of data structures, such as strings, lists, sets, and hashes are supported.

Operation Guidance

Step 1 Prepare an ECS that runs the Windows OS.

Step 2 Install **JDK1.8** (or later) and a development tool (**Eclipse** is used as an example) on the ECS, and download the **Jedis client**.

Step 3 Create a DCS instance on the DCS console. Ensure that you configure the same VPC and subnet for the DCS instance and the ECS.

Step 4 Run Eclipse on the ECS and create a Java project. Then, create a **productSalesRankDemo.java** file for the example code, and reference the Jedis client as a library to the project.

Step 5 Configure the connection address, port number, and password for the DCS instance in the example code file.

Step 6 Compile and run the code.

----End

Sample Code

```
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) {
        //Instance connection address, which is obtained from the DCS console.
        String host = "192.168.0.246";
        //Redis port number
        int port = 6379;

        Jedis jedisClient = new Jedis(host, port);

        try {
            //Instance password
            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();
        }
    }

}
```

Operation Result

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

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

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


3 Implementing Bullet-Screen and Social Comments with DCS

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 often 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

- You have created an ECS. To create an ECS, see **Creating an ECS**.
- You have created a DCS Redis instance in the same VPC, subnet, and security group as the ECS. To create an instance, see **Buying a DCS Redis Instance**.

Procedure

Step 1 Log in to the prepared ECS. To log in, see **Logging In to an ECS**.

Step 2 Install **JDK1.8** (or later) and **Eclipse** on the ECS, and download the **Jedis client**.

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

Step 3 Run Eclipse on the ECS, create a Java project, and import the Jedis client as a library into the project.

Step 4 Configure the connection address, port, and password of the DCS Redis instance in **Sample Code of Bullet-Screen Comments in Live Streaming** or **Sample Code of Replying to a Comment on a Social Website**.

Step 5 Compile and run the code.

----End

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) {

        String host = "127.0.0.1";

        int port = 6379;

        Jedis jedisClient = new Jedis(host,port);

        try {
            String authMsg = jedisClient.auth("123456");

            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();
        }

    }

}
```

Compile and run the demo. The result is as follows:

```
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-8093-3937dddda97d, 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

```
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 lombok.Data;
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) {

        String host = "127.0.0.1";

        int port = 6379;

        Jedis jedisClient = new Jedis(host,port);

        try {
            String authMsg = jedisClient.auth("123456");

            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
     */
    @Data
    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;
    }
}
```

Compile and run the demo. The result is as follows:

```
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
```


4 Merging Game Servers with DCS

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 Redis**.

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.

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 for using the pub/sub mechanism to implement cross-server game data synchronization is as follows:

```
import redis
 # Create a Redis client connection.
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'
```


5 Flashing E-commerce Sales with DCS

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 often 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

- You have created an ECS. To create an ECS, see **Creating an ECS**.
- You have created a DCS Redis instance in the same VPC, subnet, and security group as the ECS. To create an instance, see **Buying a DCS Redis Instance**.

Procedure

Step 1 Log in to the prepared ECS. To log in, see **Logging In to an ECS**.

Step 2 Install **JDK1.8** (or later) and **IntelliJ IDEA** on the ECS. Download the **Jedis client**.

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

Step 3 Run IntelliJ IDEA on the ECS. 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 4 Compile and run the following demo (this example uses Java):

```
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 via the actual address and port.
        JedisPool pool = new JedisPool(config, "127.0.0.1", 6379);
        Jedis jedis = null;
        try {
            jedis = pool.getResource();
            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

6 Reconstructing Application System Databases with DCS

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 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  |
+----+----------+------------+-----------+
```

- 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 created a Linux ECS in the same VPC as the DCS Redis instance. See **Purchasing and Logging In to a Linux ECS**.

Procedure

Step 1 Log in to the ECS.

Step 2 Install MySQL and the Redis client on the ECS 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 in the ECS, and save the script as **migrate.sql**.

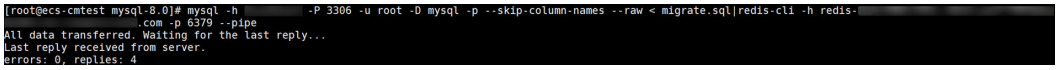
```
SELECT CONCAT(
"*8\r\n",  #8 refers to the number of fields as follows. It depends on the data structure in the MySQL table.
'$', LENGTH('HMSET'), '\r\n',  #HMSET is a Redis command in the data writing process.
'HMSET', '\r\n',
'$', LENGTH(id), '\r\n',   #id is the first field after HMSET. It will 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 ECS 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 6-1 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 **-p** 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



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-xxxxxxxxxxxx.com -p 6379
redis-xxxxxxxxxxxx.com:6379> HGETALL 1
1) "name"
2) "Wilin"
3) " birthday"
4) "1995-06-12"
5) "city"
```



```
6) "Nanjing"
redis-xxxxxxxxxxxx.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

7 Suggestions on Using Redis

Service Usage

Principle	Description	Level	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.	Required	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.	Suggested	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.	Required	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.

Principle	Description	Level	Remarks
	Do not use the **SELECT** command for multi-DB on a single instance.	Required	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.	Required	You can **select a policy** that meets your service requirements. The default eviction policy used by DCS is **volatile-lru**.
Use Redis as cache.	Do not over-rely on Redis transactions.	Suggested	After a transaction is executed, it cannot be rolled back.
	If data is abnormal, clear the cache for data restoration.	Required	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.	Required	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.	Suggested	-
Prevent cache penetration.	Non-critical path operations are passed through to the database. Limit the rate of access to the database.	Suggested	-

Principle	Description	Level	Remarks
	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.	Suggested	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.	Required	• Unless otherwise required, you are not advised to use Redis as a message queue. • Using Redis as a message queue causes capacity, network, performance, and function issues. • If message queues are required, use Kafka for throughput and RocketMQ for reliability.
Select proper specifications.	If service growth causes increases in Redis requests, use Proxy Cluster or Redis Cluster instances.	Required	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.	Required	-
	Do not use large specifications for master/standby instances.	Suggested	Redis forks a process when rewriting AOF or running the **BGSAVE** command. If the memory is too large, responses will be slow.

Principle	Description	Level	Remarks
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.	Suggested	-

Data Design

Category	Principle	Description	Level	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.	Suggested	For example: *service name*:*sub-service name*:*ID*.
	Minimize the key length.	Minimize the key length without compromising clarity of the meaning. Abbreviate common words. For example, **user** can be abbreviated to **u**, and **messages** can be abbreviated to **msg**.	Suggested	Use up to 128 bytes. The shorter the better.
	Do not use special characters except braces ({}).	Do not use special characters such as spaces, line brakes, single or double quotation marks, and other escape characters.	Required	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.	Suggested	Large values may cause unbalanced shards, hot keys, traffic or CPU usage surges, and scaling or migration failures. These problems can be avoided by proper design.

Category	Principle	Description	Level	Remarks
	Use appropriate number of elements in each key.	Do not include too many elements in each Hash, Set, or List. It is recommended that each key contain up to 5000 elements.	Suggested	Time complexity of some commands, such as **HGETALL**, is directly related to the quantity of elements in a key. If commands whose time complexity is O(N) or higher are frequently executed and a key has a large number of elements, there may be slow requests, unbalanced shards, or hot keys.
	Use appropriate data types.	This saves memory and bandwidth.	Suggested	For example, to store multiple attributes of a user, you can use multiple keys, such as **set u:1:name "X"** and **set u:1:age 20**. To save memory usage, you can also use the **HMSET** command to set multiple fields to their respective values in the hash stored at one key.
	Set appropriate timeout.	Do not set a large number of keys to expire at the same time.	Suggested	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	Level	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.	Required	For example, the **HGETALL**, **LRANGE**, **SMEMBERS**, **ZRANGE**, and **SINTER** commands will consume a large number of CPU resources if there is a large number of elements. Alternatively, you can use **SCAN** sister commands, such as **HSCAN**, **SSCAN**, and **ZSCAN** commands.
Do not use high-risk commands.	Do not use high-risk commands such as **FLUSHALL**, **KEYS**, and **HGETALL**, or rename them.	Required	For details, see **Renaming Commands**.
Exercise caution when using the **SELECT** command.	Redis does not have a strong support for multi-DB. Redis is single-threaded, so databases interfere with each other. You are advised to use multiple Redis instances instead of using multi-DB on one instance.	Suggested	-
Use batch operations to improve efficiency.	For batch operations, use the **MGET** command, **MSET** command, or pipelining to improve efficiency, but do not include a large number of elements in one batch operation.	Suggested	**MGET** command, **MSET** command, and pipelining differ in the following ways: • **MGET** and **MSET** are atomic operations, while pipelining is not. • Pipelining can be used to send multiple commands at a time, while **MGET** and **MSET** cannot. • Pipelining must be supported by both the server and the client.

Principle	Description	Level	Remarks
Do not use time-consuming code in Lua scripts.	The timeout of Lua scripts is 5s, so avoid using long scripts.	Required	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.	Required	-
Follow the rules for using Lua on cluster instances.	Follow the rules for using Lua on cluster instances.	Required	• When the **EVAL** or **EVALSHA** command is run, the command parameter must contain at least one key. Otherwise, the client displays the error message "ERR eval/evalsha numkeys must be bigger than zero in redis cluster mode." • When the **EVAL** or **EVALSHA** command is run, a cluster DCS Redis instance uses the first key to compute slots. Ensure that the keys to be operated are in the same slot.
Optimize multi-key operation commands such as **MGET** and **HMGET** with parallel processing and non-blocking I/O.	Some clients do not treat these commands differently. Keys in such a command are processed sequentially before their values are returned in a batch. This process is slow and can be optimized through pipelining.	Suggested	For example, running the **MGET** command on a cluster using Lettuce is dozens of times faster than using Jedis, because Lettuce uses pipelining and non-blocking I/O while Jedis does not have a special plan itself. To use Jedis in such scenarios, you need to implement slot grouping and pipelining by yourself.

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

SDK Usage

Principle	Description	Level	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.	Suggested	Frequently connecting to and disconnecting from Redis will unnecessarily consume a lot of system resources and can cause host breakdown in extreme cases. Ensure that the Redis client connection pool is correctly configured.
The client must perform fault tolerance in case of faults or slow requests.	The client should have fault tolerance and retry mechanisms in case of master/standby switchover, command timeout, or slow requests caused by network fluctuation or configuration errors.	Suggested	See **Redis Client Retry**.

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

O&M and Management

Principle	Description	Level	Remarks
Use passwords in production.	In production systems, use passwords to protect Redis.	Required	-
Ensure security on the live network.	Do not allow unauthorized developers to connect to redis-server in the production environment.	Required	-
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.	Suggested	Master/standby switchover can be triggered manually on the console. It is strongly recommended that you use Lettuce for these drills.
Configure monitoring.	Pay attention to the Redis capacity and expand it before overload.	Required	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.	Suggested	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.	Suggested	-
	Perform routine diagnosis on Redis commands and check whether O(N) commands have potential risks.	Suggested	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.	Suggested	Detect potential risks based on slow query logs and rectify faults as soon as possible.

8 Redis Client Retry

Importance of Retry

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

Scenarios Where Redis Operations Fail

Scenario	Description
Master/standby switchover triggered by a fault	If the master node is faulty due to Redis underlying hardware or other reasons, a master/standby switchover is triggered to ensure that the instance is still available. A master/standby switchover has the following impacts: • Disconnection down to seconds • Read-only for up to 30 seconds
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 **Modifying Specifications**.
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.	Timeout may occur in any of the following phases: • A command is successfully sent by the client but has not reached Redis. • The command has reached Redis, but the execution times out. • Redis has executed the command, but the result returned to the client times out. A retried operation may be repeatedly executed in Redis. Therefore, not all operations are suitable to be retried. You are advised to retry only idempotent operations, such as running the **SET** command. For example, if you run the **SET a b** command multiple times, the value of **a** can only be **b** or the execution fails. If you run **LPUSH mylist a**, which is not idempotent, **mylist** may contain multiple **a** elements.
Configure proper retry times and interval.	Configure the retry times and interval based on service requirements in actual scenarios to prevent the following problems: • If the number of retries is insufficient or the interval is too long, the application may fail to complete operations. • If the number of retries is too large or the interval is too short, the application may occupy too many system resources and the server may be blocked due to too many requests. Common retry interval policies include immediate retry, fixed-interval retry, exponential backoff retry, and random backoff retry.
Avoid retry nesting.	Retry nesting may cause the retry interval to be exponentially amplified.
Record retry exceptions and print failure reports.	During retry, you can print retry error logs at the WARN level.

Jedis Client Configurations

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

Example code:

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

Table 8-1 Recommended Jedis connection pool parameter settings

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

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

Parameter	Description	Recommended Setting
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 **–1** 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 **ping** command when borrowing connections from the resource pool. Invalid connections will be removed.	If your service is extremely sensitive to connections and the performance is acceptable, you can set this parameter to **True**. Generally, you are advised to set this parameter to **False** to enable idle connection detection.

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

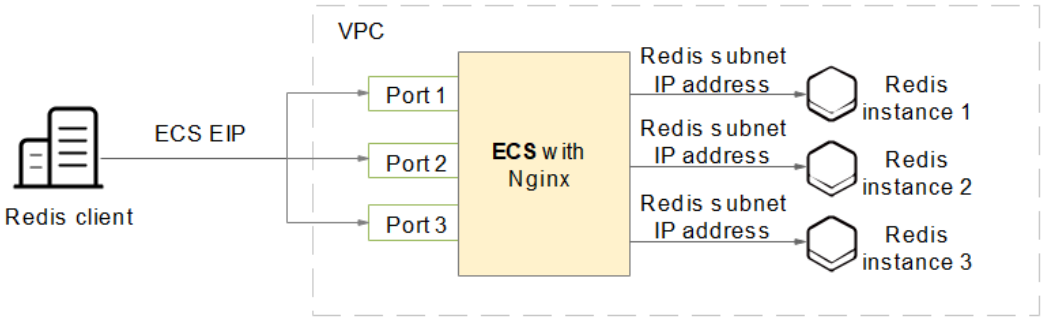
9 Using Nginx for Public Access to Single-node, Master/Standby, or Proxy Cluster DCS Redis Instances

Currently, DCS Redis 4.0, 5.0, and 6.0 instances 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 9-1**, 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 9-1 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

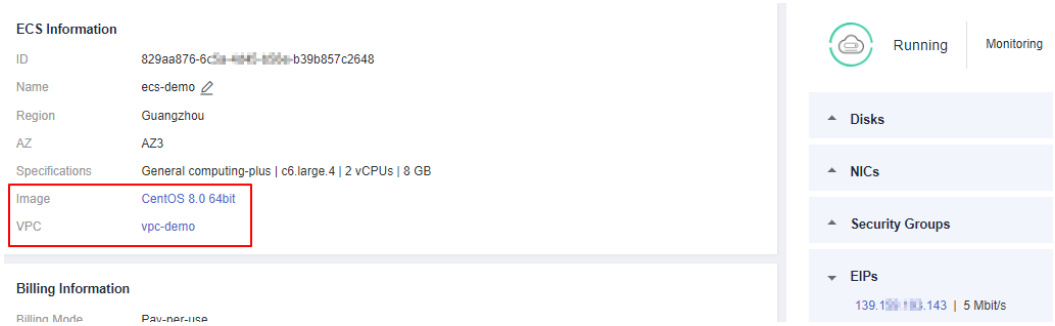
Step 1 Obtain the VPC where the DCS Redis instance is deployed.

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

Figure 9-2 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 9-3 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.el7.ngx.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 9-4 Adding Nginx configurations

```
#   * Official [illegible] 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 9-5 Starting Nginx and verifying the start

```
[root@k[illegible]emo src]# service nginx restart
Redirecting to /bin/systemctl restart nginx.service
[root@k[illegible]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**. 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 9-6 Checking the ECS security group

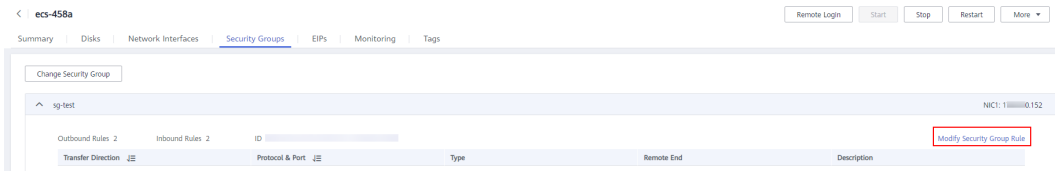
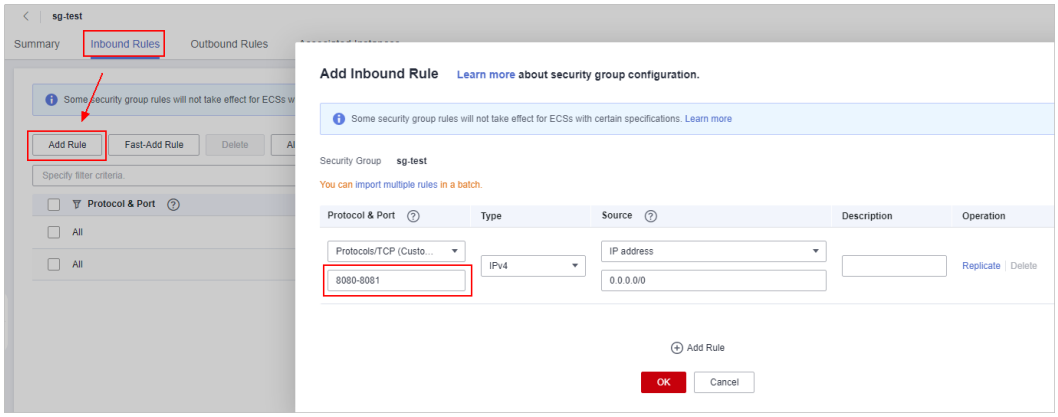


Figure 9-7 Adding an inbound rule for the security group



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 9-8 Accessing the first DCS Redis instance using Nginx

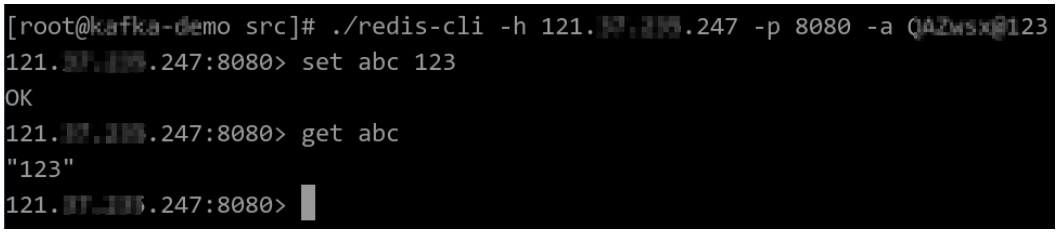
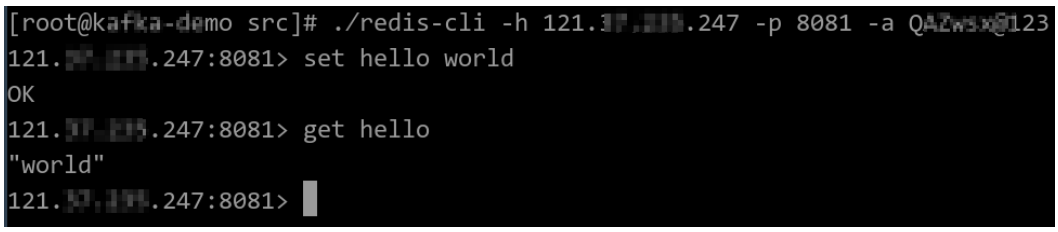


Figure 9-9 Accessing the second DCS Redis instance using Nginx



The jump server has now been set up. You can access Redis over public networks.

----End

10 Using SSH Tunneling for Public Access to a DCS Instance

Context

VPCs are used to ensure network security of public cloud services, such as DCS. Your DCS instance can be accessed only by an ECS that is in the same VPC as the instance.

Solution

If an EIP is bound to a Huawei Cloud ECS, you can remotely access the ECS from a local computer.

You can create an SSH tunnel as a proxy to connect your DCS instance and local computer to achieve proxy forwarding.

NOTE

- Redis Cluster DCS Redis 4.0, 5.0, or 6.0 instances do not support public access using this solution.
- 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

You have a DCS instance and a local computer that can connect to the Internet. Tools such as MobaXterm and the Redis client have been installed.

You have an ECS that meets the following requirements:

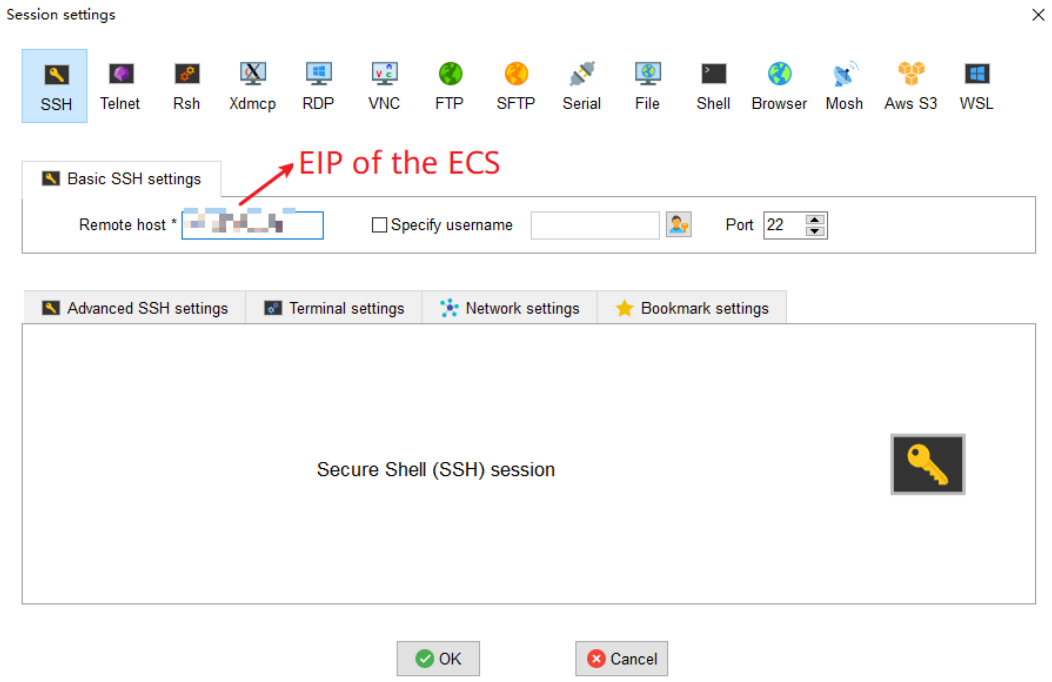
- The ECS is bound with an EIP for public access.
- The VPC and subnet configured for the ECS are the same as those configured for the DCS instance.
- Security group rules have been correctly configured for the ECS.
- The ECS runs the Linux OS.

If these 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.

Using MobaXterm to Create a Tunnel as a Jump Server

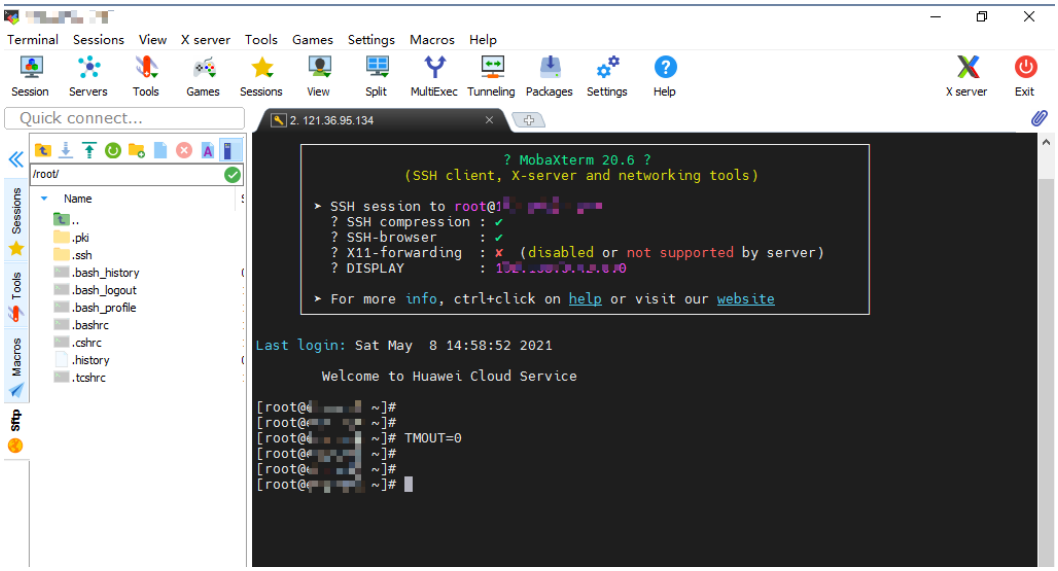
Step 1 Create an SSH session for connecting to the ECS using port 22.

Figure 10-1 Creating an SSH session



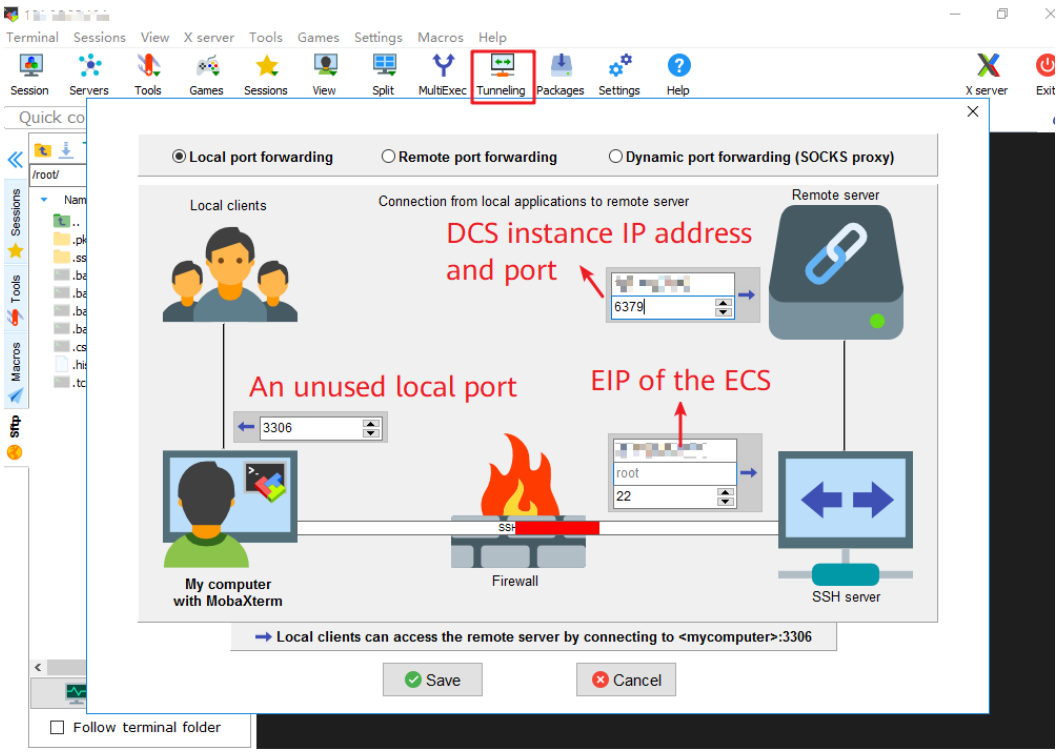
Step 2 After the session is configured, enter the username and password to log in to the ECS. After login, enter "TMOUT=0" to prevent the session from being automatically closed due to timeout.

Figure 10-2 Entering "TMOUT=0"



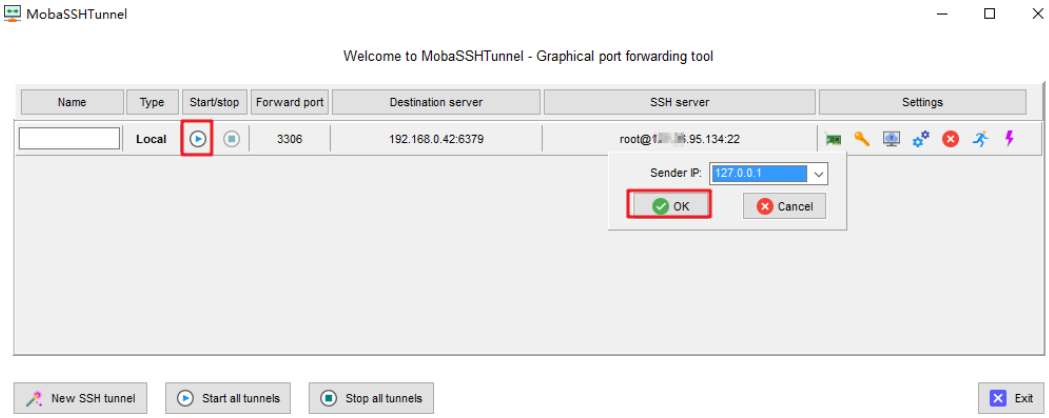
Step 3 Click **Tunneling** to create a tunnel.

Figure 10-3 Creating a tunnel



Step 4 Set the local IP address to 127.0.0.1 and start the tunnel.

Figure 10-4 Starting the tunnel



Step 5 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:

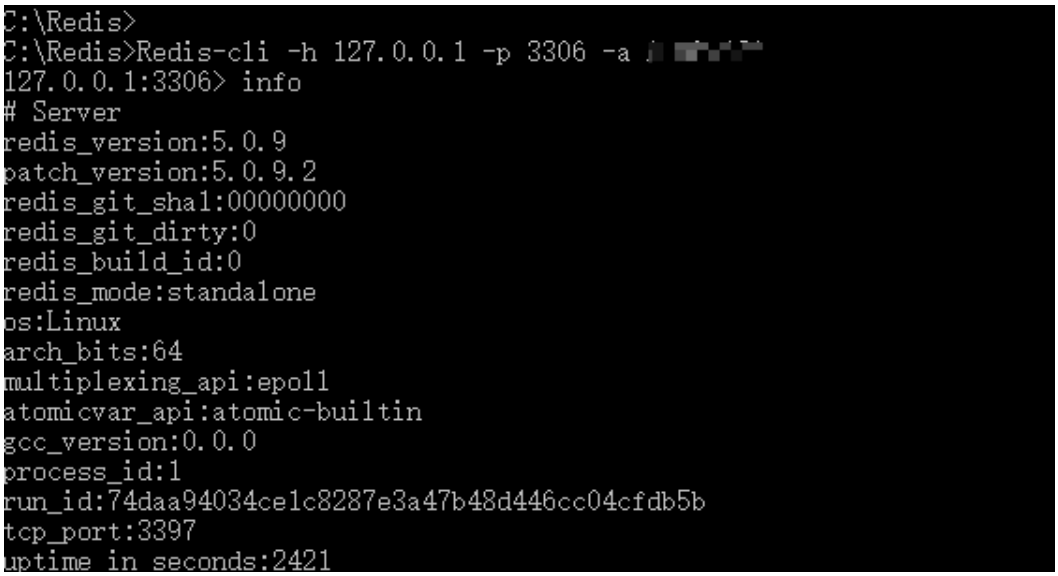
-h *{host name}*: localhost or 127.0.0.1, which is the same as the local IP address configured for the tunnel.

-p *{port number}*: 3306, which is the same as the forward port configured for the tunnel.

-a *{password}*: password of the DCS instance.

Step 6 If the connection is successful, the following information is displayed.

Figure 10-5 Successfully connecting to a DCS instance



----End

11 Using ELB for Public Access to DCS

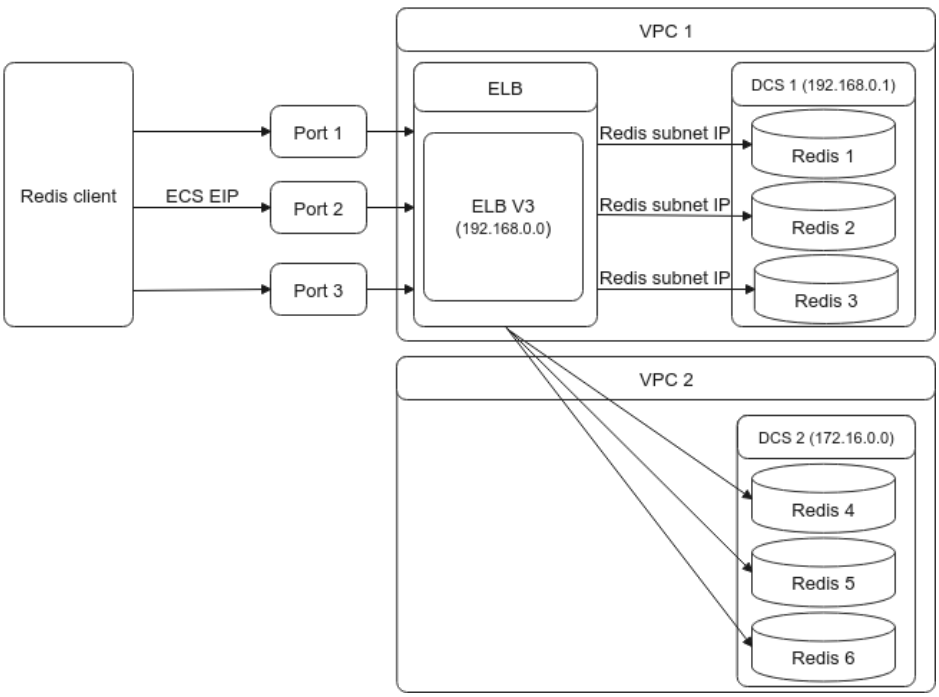
Currently, DCS Redis 4.0, 5.0, and 6.0 instances cannot be bound with elastic IP addresses (EIPs) and cannot be accessed over public networks directly. This section describes how to access a single-node, master/standby, read/write splitting, or Proxy Cluster instance or a node in a Redis Cluster instance 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 11-1 Process for accessing DCS through ELB



Configurations

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.

NOTE

- 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 **Cross-VPC Backend** when creating a dedicated load balancer.

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

NOTE

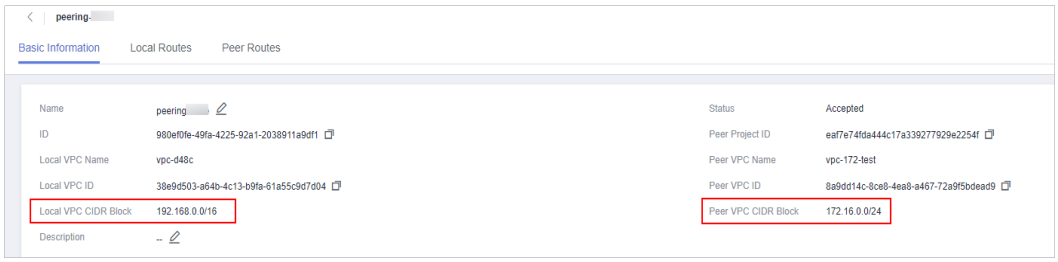
1. When adding a backend server, click the **Cross-VPC Backend Servers** tab and then click **Add Cross-VPC Backend Server**.
2. **In the Add Cross-VPC Backend Server dialog box, enter the IP address and port number of your DCS instance.**
3. A Redis Cluster DCS instance contains multiple master/replica pairs. When configuring a cross-VPC backend server, you can add the IP address and port number of any master or replica node.
4. 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

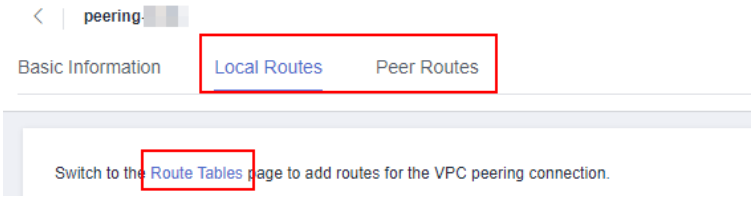
Even if your load balancer and DCS instance are in the same VPC, you still need to create a VPC peering connection. For the local VPC, select the VPC where your load balancer and DCS instance are located. For the peer VPC, select another VPC.

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**.



Step 7 Configure local and peer routes for the VPC peering connection.

1. On the **Local Routes** and **Peer Routes** tab pages, click the link to go to the route tables page.



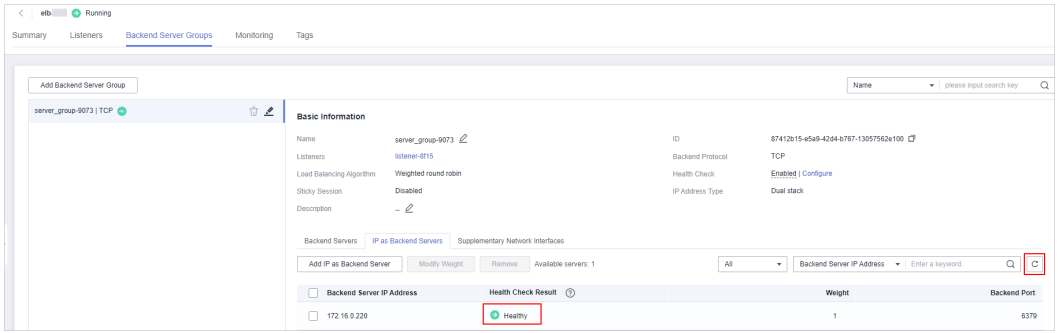
2. Local route: On the summary page of local routes, click **Add Route**. In the displayed **Add Route** dialog box, set **Destination** to the value of **Peer VPC CIDR Block** of the VPC peering connection, set **Next Hop Type** to **VPC peering connection**, set **Next Hop** to the VPC peering connection created in **5**, and click **OK**.
3. Peer route: On the summary page of peer routes, click **Add Route**. In the displayed **Add Route** dialog box, set **Destination** to the value of **Local VPC CIDR Block** of the VPC peering connection, set **Next Hop Type** to **VPC peering connection**, set **Next Hop** to the VPC peering connection created in **5**, 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.

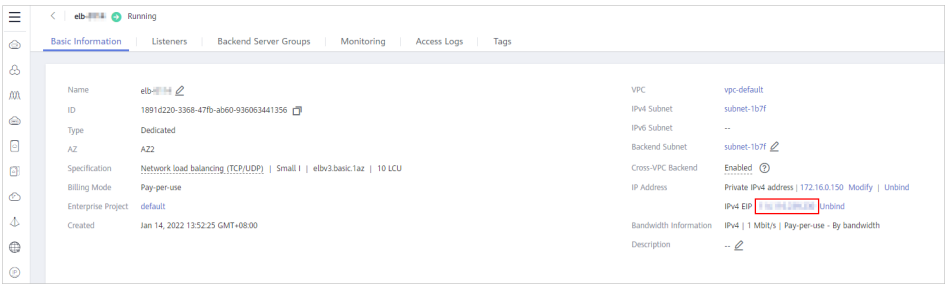
1. On the **Elastic Load Balance** page, click the name of the created load balancer. The basic information page of the load balancer is displayed.
2. On the **Backend Server Groups** > **IP as Backend Servers** tab page, view the health check result of the DCS instance IP address.



----End

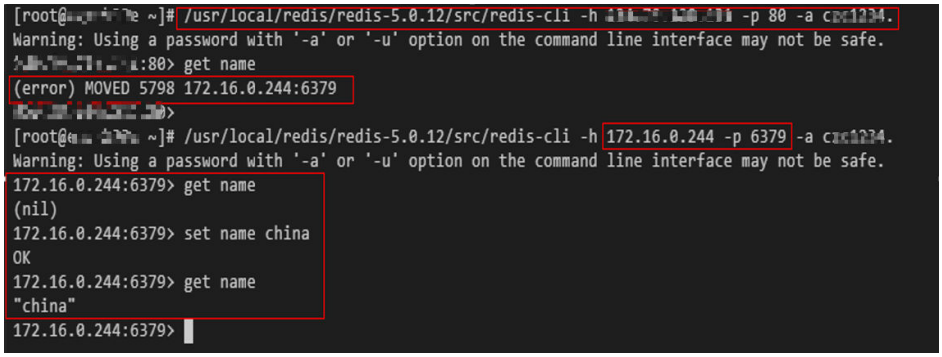
Connecting to the DCS Instance Through ELB

- Connecting to a node in a Redis Cluster DCS instance through ELB
  a. View the basic information of the load balancer created in **Step 3**.



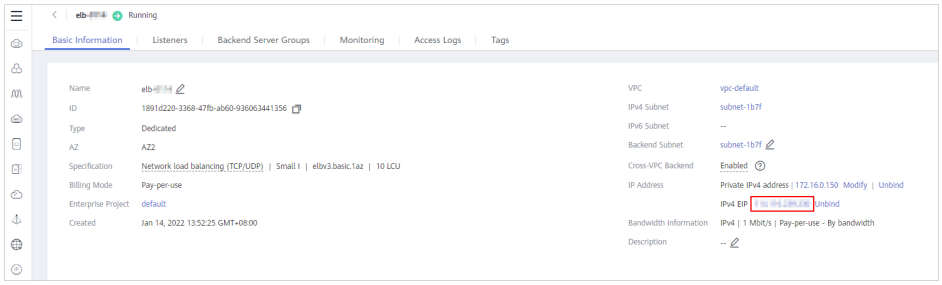
b. **Buy an ECS**, log in to it, and install the Redis client by referring to **redis-cli**.

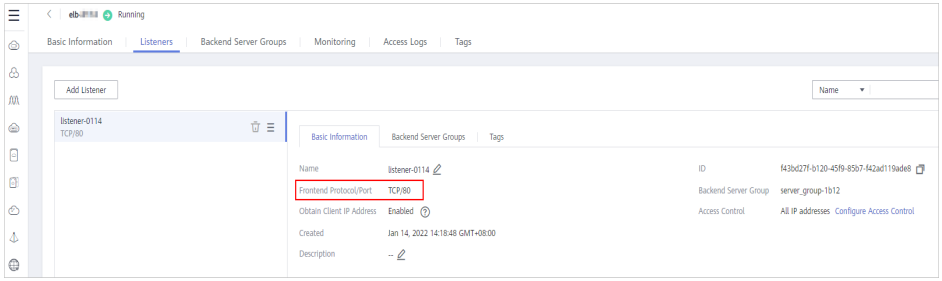
c. On the Redis client, connect to the DCS instance using the IP address and port number configured in **Step 4**. If you use the EIP and port number of the load balancer, an error will be reported.



- Connecting to a single-node, master/standby, read/write splitting, or Proxy Cluster DCS instance through ELB

  a. View the IPv4 EIP and port number of the load balancer created in **Step 3**.

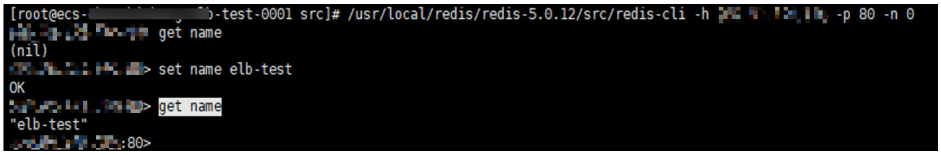




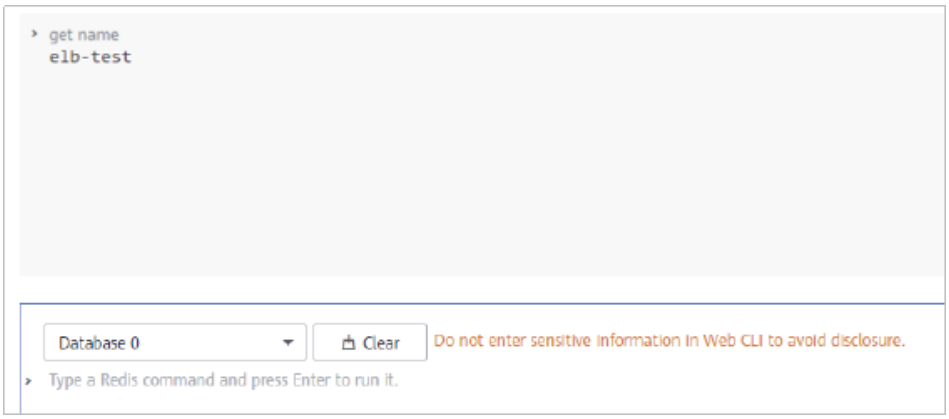
  b. **Buy an ECS**, log in to it, and install the Redis client by referring to **redis-cli**.

  c. Use redis-cli to access the load balancer using its EIP and port number (which is 80).

d. Write a key through ELB.



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.



12 Detecting and Handling Big Keys and Hot Keys

Definitions of Big Keys and Hot Keys

Term	Definition
Big key	There are two types of big keys: • 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. • 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.
Hot key	A key is defined as a hot key if it is frequently requested or if it occupies a large number of resources. For example: • In a cluster instance, a shard processes 10,000 requests per second, among which 3000 are performed on the same key. • 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 **HGETALL** operation on a Hash key.

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

Impact of Big Keys and Hot Keys

Category	Impact
Big key	**Instance specifications fail to be modified.** 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.
	Data migration fails. 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.
	Cluster shards are unbalanced. • 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. • The bandwidth usage of shards is unbalanced. For example, flow control is repeatedly triggered on a shard.
	Latency of client command execution increases. Slow operations on a big key block other commands, resulting in a large number of slow queries.
	Flow control is triggered on the instance. 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.
	Master/standby switchover is triggered. If the high-risk **DEL** operation is performed on a big key, the master node may be blocked for a long time, causing a master/standby switchover.
Hot key	**Cluster shards are unbalanced.** 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.

Category	Impact
	CPU usage surges. 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.
	Cache breakdown may occur. 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.

Big keys and hot keys can be avoided through proper design. For details, see **Suggestions on Using Redis**.

Detecting Big Keys and Hot Keys

Method	Description
Through **Big Key Analysis** and **Hot Key Analysis** on the DCS console	See **Analyzing Big Keys and Hot Keys**.
By using the **bigkeys** and **hotkeys** options on redis-cli	• redis-cli uses the **bigkeys** 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 **redis-cli -h** *<Instance connection address>* **-p** *<Port number>* **-a** *<Password>* **--bigkeys**. • In Redis 4.0 and later, you can use the **hotkeys** option to quickly find hot keys in redis-cli. Run this command during service running to find hot keys: **redis-cli -h** *<Instance connection address>* **-p** *<Port number>* **-a** *<Password>* **--hotkeys**. The hot key details can be obtained from the summary part in the returned result.

Method	Description
Searching for big keys using Redis commands	If there is a pattern of big keys, for example, the prefix is **cloud:msg:test**, 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. • Commands for querying the number of members: **LLEN**, **HLEN**, **XLEN**, **ZCARD**, **SCARD** • Commands for querying the memory usage of keys: **DEBUG OBJECT**, **MEMORY USAGE** **CAUTION** This method consumes a large number of computing resources. To ensure service running, do not use this method on instances with heavy service pressure.
Searching for big keys using redis-rdb-tools	**redis-rdb-tools** 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. To use this method, you must **export the RDB file of an instance** on the **Backups & Restorations** page of the DCS console. **CAUTION** This method does not affect service running, but is not as timely as online analysis.

Optimizing Big Keys and Hot Keys

Category	Method
Big key	**Split big keys.** Scenarios: • **If the big key is a String**, you can split it into several key-value pairs and use **MGET** or a pipeline consisting of multiple **GET** 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. • **If the big key contains multiple elements, and the elements must be operated together**, 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. • **If the big key contains multiple elements, and only some elements need to be operated each time**, separate the elements. Take a Hash key as an example. Each time you run the **HGET** or **HSET** command, the result of the hash value modulo *N* (customized on the client) determines which key the field falls on. This algorithm is similar to that used for calculating slots in Redis Cluster.
	Store big keys on other storage media. 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 NoSQL databases, and delete the big key from Redis. **CAUTION** Do not use the **DEL** command to delete big keys. Otherwise, Redis may be blocked or even a master/standby switchover may occur.
	Set appropriate expiration and delete expired data regularly. 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, **scan expired keys**.
Hot key	**Split read and write requests.** If a hot key is frequently read, **configure read/write splitting** 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.

Category	Method
	Use the client cache or local cache. 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.
	Design a circuit breaker or degradation mechanism. 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.

