

ApsaraDB for Redis

Best Practices

Best Practices

Scenario introduction

ApsaraDB for Redis functions basically the same as Redis. You can use this product to create a ranking function for an online game easily.

Sample code

```
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 GameRankSample {

    static int TOTAL_SIZE = 20;

    public static void main(String[] args)
    {
        //Connection information, obtained from the console
        String host = "xxxxxxxxx.m.cnhz1.kvstore.aliyuncs.com";
        int port = 6379;

        Jedis jedis = new Jedis(host, port);

        try {
            //Instance password
            String authString = jedis.auth("password");//password

            if (!authString.equals("OK"))
            {
                System.err.println("AUTH Failed: " + authString);
                return;
            }

            //Key
            String key = "Game name : Keep Running, Ali!";

            //Clear any data that already exists
            jedis.del(key);
```

```
//Generate several simulated players
List<String> playerList = new ArrayList<String>();
for (int i = 0; i < TOTAL_SIZE; ++i)
{
    //Randomly generate an ID for each player
    playerList.add(UUID.randomUUID().toString());
}

System.out.println("Input all players");
//Record the score for each player
for (int i = 0; i < playerList.size(); i++)
{
    //Randomly generate numbers as the scores of the simulated players
    int score = (int)(Math.random()*5000);

    String member = playerList.get(i);

    System.out.println("Player ID : " + member + " , Player score: " + score);

    //Add the player ID and score to the SortedSet of the corresponding key
    jedis.zadd(key, score, member);
}

//Print out the ranking of all players
System.out.println();
System.out.println(" "+key);
System.out.println(" All players ranking ");

//Obtain the sorted list of players from the SortedSet of the corresponding key
Set<Tuple> scoreList = jedis.zrevrangeWithScores(key, 0, -1);
for (Tuple item : scoreList) {
    System.out.println("Player ID : "+item.getElement()+" , Player score:"+Double.valueOf(item.getScore()).intValue());
}

//Print out Top 5 players
System.out.println();
System.out.println(" "+key);
System.out.println(" Top players");

scoreList = jedis.zrevrangeWithScores(key, 0, 4);
for (Tuple item : scoreList) {
    System.out.println("Player ID : "+item.getElement()+" , Player score:"+Double.valueOf(item.getScore()).intValue());
}

//Print out a list of specific players
System.out.println();
System.out.println(" "+key);
System.out.println(" players with scores from 1000 to 2000");

//Obtain a list of players with scores from 1000 to 2000 from the SortedSet of the corresponding key
scoreList = jedis.zrangeByScoreWithScores(key, 1000, 2000);
for (Tuple item : scoreList) {
    System.out.println("Player ID : "+item.getElement()+" , Player score:"+Double.valueOf(item.getScore()).intValue());
}
```

```

}

} catch (Exception e) {
e.printStackTrace();
}finally{
jedis.quit();
jedis.close();
}
}
}
}

```

Output

After you access the ApsaraDB for Redis instance with the correct address and password and run the above Java code, the following output is displayed:

```

Enter all players
Player ID: 9193e26f-6a71-4c76-8666-eaf8ee97ac86; score: 3860
Player ID: db03520b-75a3-48e5-850a-071722ff7afb; score: 4853
Player ID: d302d24d-d380-4e15-a4d6-84f71313f27a; score: 2931
Player ID: bee46f9d-4b05-425e-8451-8aa6d48858e6; score: 1796
Player ID: ec24fb9e-366e-4b89-a0d5-0be151a8cad0; score: 2263
Player ID: e11ecc2c-cd51-4339-8412-c711142ca7aa; score: 1848
Player ID: 4c396f67-da7c-4b99-a783-25919d52d756; score: 958
Player ID: a6299dd2-4f38-4528-bb5a-aa2d48a9f94a; score: 2428
Player ID: 2e4ec631-1e4e-4ef0-914f-7bf1745f7d65; score: 4478
Player ID: 24235a85-85b9-476e-8b96-39f294f57aa7; score: 1655
Player ID: e3e8e1fa-6aac-4a0c-af80-4c4a1e126cd1; score: 4064
Player ID: 99bc5b4f-e32a-4295-bc3a-0324887bb77e; score: 4852
Player ID: 19e2aa6b-a2d8-4e56-bdf7-8b59f64bd8e0; score: 3394
Player ID: cb62bb24-1318-4af2-9d9b-fbff7280dbec; score: 3405
Player ID: ec0f06da-91ee-447b-b935-7ca935dc7968; score: 4391
Player ID: 2c814a6f-3706-4280-9085-5fe5fd56b71c; score: 2510
Player ID: 9ee2ed6d-08b8-4e7f-b52c-9adfe1e32dda; score: 63
Player ID: 0293b43a-1554-4157-a95b-b78de9edf6dd; score: 1008
Player ID: 674bbdd1-2023-46ae-bbe6-dfcd8e372430; score: 2265
Player ID: 34574e3e-9cc5-43ed-ba15-9f5405312692; score: 3734

Game Name: Keep Running, Ali!
Full Player Ranking
Player ID: db03520b-75a3-48e5-850a-071722ff7afb; score: 4853
Player ID: 99bc5b4f-e32a-4295-bc3a-0324887bb77e; score: 4852
Player ID: 2e4ec631-1e4e-4ef0-914f-7bf1745f7d65; score: 4478
Player ID: ec0f06da-91ee-447b-b935-7ca935dc7968; score: 4391
Player ID: e3e8e1fa-6aac-4a0c-af80-4c4a1e126cd1; score: 4064
Player ID: 9193e26f-6a71-4c76-8666-eaf8ee97ac86; score: 3860
Player ID: 34574e3e-9cc5-43ed-ba15-9f5405312692; score: 3734
Player ID: cb62bb24-1318-4af2-9d9b-fbff7280dbec; score: 3405
Player ID: 19e2aa6b-a2d8-4e56-bdf7-8b59f64bd8e0; score: 3394
Player ID: d302d24d-d380-4e15-a4d6-84f71313f27a; score: 2931
Player ID: 2c814a6f-3706-4280-9085-5fe5fd56b71c; score: 2510
Player ID: a6299dd2-4f38-4528-bb5a-aa2d48a9f94a; score: 2428
Player ID: 674bbdd1-2023-46ae-bbe6-dfcd8e372430; score: 2265

```

```

Player ID: ec24fb9e-366e-4b89-a0d5-0be151a8cad0; score: 2263
Player ID: e11ecc2c-cd51-4339-8412-c711142ca7aa; score: 1848
Player ID: bee46f9d-4b05-425e-8451-8aa6d48858e6; score: 1796
Player ID: 24235a85-85b9-476e-8b96-39f294f57aa7; score: 1655
Player ID: 0293b43a-1554-4157-a95b-b78de9edf6dd; score: 1008
Player ID: 4c396f67-da7c-4b99-a783-25919d52d756; score: 958
Player ID: 9ee2ed6d-08b8-4e7f-b52c-9adfe1e32dda; score: 63

```

Game Name: Keep Running, Ali!

Top Players

```

Player ID: db03520b-75a3-48e5-850a-071722ff7afb; score: 4853
Player ID: 99bc5b4f-e32a-4295-bc3a-0324887bb77e; score: 4852
Player ID: 2e4ec631-1e4e-4ef0-914f-7bf1745f7d65; score: 4478
Player ID: ec0f06da-91ee-447b-b935-7ca935dc7968; score: 4391
Player ID: e3e8e1fa-6aac-4a0c-af80-4c4a1e126cd1; score: 4064

```

Game Name: Keep Running, Ali!

Players with Scores from 1000 to 2000

```

Player ID: 0293b43a-1554-4157-a95b-b78de9edf6dd; score: 1008
Player ID: 24235a85-85b9-476e-8b96-39f294f57aa7; score: 1655
Player ID: bee46f9d-4b05-425e-8451-8aa6d48858e6; score: 1796
Player ID: e11ecc2c-cd51-4339-8412-c711142ca7aa; score: 1848

```

Scenario introduction

ApsaraDB for Redis, functions basically the same as Redis, can be used to create correlation analysis programs for items on e-commerce sites.

The correlation between items is the chances where the items are added to a user's shopping cart. The analysis result plays an important role in analyzing users' shopping behavior for the e-commerce industry. For example:

- To recommend related items to a user who lingers on the details page of a certain item;

- To recommend related items to a user who just adds an item to the shopping cart;

- To place highly correlated items together on the shelves.

ApsaraDB for Redis can be used to create a sorted set for each type of item. The members of this set are scored based on how often they appear in the same users' carts with the primary item. Each time items A and B appear in the same user's shopping cart, the respective sorted sets for items A and B in ApsaraDB for Redis are updated.

Sample code

```
package shop.kvstore.aliyun.com;
```

```

import java.util.Set;
import redis.clients.jedis.Jedis;
import redis.clients.jedis.Tuple;

public class AliyunShoppingMall {

    public static void main(String[] args)
    {
        //Information for connecting ApsaraDB for Redis, which can be obtained from the console
        String host = "xxxxxxx.m.cnhza.kvstore.aliyuncs.com";
        int port = 6379;

        Jedis jedis = new Jedis(host, port);

        try {
            //ApsaraDB for Redis instance password
            String authString = jedis.auth("password");//password

            if (!authString.equals("OK"))
            {
                System.err.println("AUTH Failed: " + authString);
                return;
            }

            //Product list
            String key0="Alibaba Cloud:product:beer";
            String key1="Alibaba Cloud:product:chocolate";
            String key2="Alibaba Cloud:product:cola";
            String key3="Alibaba Cloud:product:gum";
            String key4="Alibaba Cloud:product:jerky";
            String key5="Alibaba Cloud:product:chicken wing";

            final String[] aliyunProducts=new String[]{key0,key1,key2,key3,key4,key5};

            //Initialization, clearing any existing data
            for (int i = 0; i < aliyunProducts.length; i++) {
                jedis.del(aliyunProducts[i]);
            }

            //Simulates user shopping behavior
            for (int i = 0; i < 5; i++) { //Simulates users' buying items

                customersShopping(aliyunProducts,i,jedis);

            }

            System.out.println();

            //Generates correlation between items using ApsaraDB for Redis
            for (int i = 0; i < aliyunProducts.length; i++) {
                System.out.println(">>>>>>>>>and"+aliyunProducts[i]+"products bought together<<<<<<<<<<<<<<");
                Set<Tuple> relatedList = jedis.zrevrangeWithScores(aliyunProducts[i], 0, -1);
            }
        }
    }
}

```

```

for (Tuple item : relatedList) {
    System.out.println("Product name : "+item.getElement()+" , bought together
times:"+ Double.valueOf(item.getScore()).intValue());
}

System.out.println();
}

} catch (Exception e) {
    e.printStackTrace();
}finally{
    jedis.quit();
    jedis.close();
}
}

private static void customersShopping(String[] products, int i, Jedis jedis) {

    //Simulates three possibilities of shopping behavior at random
    int bought=(int)(Math.random()*3);

    if(bought==1){
        //Simulates the business logic: The user buys the following items
        System.out.println("User"+i+"bought"+products[0]+"," +products[2]+"," +products[1]);

        //Adds the correlations between the items to SortSet in ApsaraDB for Redis
        jedis.zincrby(products[0], 1, products[1]);
        jedis.zincrby(products[0], 1, products[2]);
        jedis.zincrby(products[1], 1, products[0]);
        jedis.zincrby(products[1], 1, products[2]);
        jedis.zincrby(products[2], 1, products[0]);
        jedis.zincrby(products[2], 1, products[1]);

    }else if(bought==2){
        //Simulates the business logic: The user buys the following items
        System.out.println("User"+i+"bought"+products[4]+"," +products[2]+"," +products[3]);

        //Adds the correlations between the items to SortSet in ApsaraDB for Redis
        jedis.zincrby(products[4], 1, products[2]);
        jedis.zincrby(products[4], 1, products[3]);
        jedis.zincrby(products[3], 1, products[4]);
        jedis.zincrby(products[3], 1, products[2]);
        jedis.zincrby(products[2], 1, products[4]);
        jedis.zincrby(products[2], 1, products[3]);

    }else if(bought==0){
        //Simulates the business logic: The user buys the following items
        System.out.println("User"+i+"bought"+products[1]+"," +products[5]);

        //Adds the correlations between the items to SortSet in ApsaraDB for Redis
        jedis.zincrby(products[5], 1, products[1]);
        jedis.zincrby(products[1], 1, products[5]);
    }

}
}

```

After you access the ApsaraDB for Redis instance with the correct address and password and run the above Java code, the following output is displayed:

[illegible]

ApsaraDB for Redis provides publishing (pub) and subscription (sub) functions, just like Redis. This allows multiple clients to subscribe to messages published by a client.

It must be noted that messages published using ApsaraDB for Redis are “non-persistent” . That means the message publisher is only responsible for publishing a message and does not save previously sent messages, whether or not they were received by anyone. Thus, messages are “lost once published.” Message subscribers receive messages published after their subscription. They will not receive the earlier messages in the channel.

In addition, a message publisher (publish client) does not connect to a server exclusively. While

publishing messages, you can perform more operations (for example, List) from the same client at the same time. However, a message subscriber (subscribe client) connects to a server exclusively. That is, during the subscription, the client may not perform any other operations. Rather, the operations are blocked while the client is waiting for messages in the channel. Thus, message subscribers must use a dedicated server connection or thread (see the example below).

Sample code

For the message publisher (publish client)

```
package message.kvstore.aliyun.com;

import redis.clients.jedis.Jedis;

public class KVStorePubClient {

    private Jedis jedis;//
    public KVStorePubClient(String host,int port, String password){
        jedis = new Jedis(host,port);

        //KVStore instance password
        String authString = jedis.auth(password);//password

        if (!authString.equals("OK"))
        {
            System.err.println("AUTH Failed: " + authString);
            return;
        }
    }

    public void pub(String channel,String message){
        System.out.println(" >>> PUBLISH > Channel:"+channel+" > Send Message:"+message);

        jedis.publish(channel, message);
    }

    public void close(String channel){
        System.out.println(" >>> PUBLISH End > Channel:"+channel+" > Message:quit");

        //The message publisher stops sending by sending a "quit" message
        jedis.publish(channel, "quit");

    }

}
```

For the message subscriber (subscribe client)

```
package message.kvstore.aliyun.com;

import redis.clients.jedis.Jedis;
```

```

import redis.clients.jedis.JedisPubSub;

public class KVStoreSubClient extends Thread{

    private Jedis jedis;
    private String channel;
    private JedisPubSub listener;

    public KVStoreSubClient(String host,int port, String password){
        jedis = new Jedis(host,port);

        //ApsaraDB for Redis instance password
        String authString = jedis.auth(password);//password

        if (!authString.equals("OK"))
        {
            System.err.println("AUTH Failed: " + authString);
            return;
        }
    }

    public void setChannelAndListener(JedisPubSub listener,String channel){
        this.listener=listener;
        this.channel=channel;
    }

    private void subscribe(){

        if(listener==null || channel==null){
            System.err.println("Error:SubClient> listener or channel is null");
        }

        System.out.println(" >>> SUBSCRIBE > Channel:"+channel);
        System.out.println();

        //When the recipient is listening for subscribed messages, the process is blocked until the quit message is received
        (passively) or the subscription is cancelled actively
        jedis.subscribe(listener, channel);
    }

    public void unsubscribe(String channel){
        System.out.println(" >>> UNSUBSCRIBE > Channel:"+channel);
        System.out.println();

        listener.unsubscribe(channel);
    }

    @Override
    public void run() {
        try{

            System.out.println();
            System.out.println("-----SUBSCRIBE Begin-----");

            subscribe();

```

```

System.out.println("-----SUBSCRIBE End-----");
System.out.println();
}catch(Exception e){
e.printStackTrace();
}

}

}

```

For the message listener

```

package message.kvstore.aliyun.com;

import redis.clients.jedis.JedisPubSub;

public class KVStoreMessageListener extends JedisPubSub{

@Override
public void onMessage(String channel, String message) {

System.out.println(" <<< SUBSCRIBE< Channel:" + channel + " > Recceived Message:" + message );
System.out.println();

//When a quit message is received, the subscription is cancelled (passively)
if(message.equalsIgnoreCase("quit")){
this.unsubscribe(channel);
}
}

@Override
public void onPMessage(String pattern, String channel, String message) {
// TODO Auto-generated method stub

}

@Override
public void onSubscribe(String channel, int subscribedChannels) {
// TODO Auto-generated method stub

}

@Override
public void onUnsubscribe(String channel, int subscribedChannels) {
// TODO Auto-generated method stub

}

@Override
public void onPUnsubscribe(String pattern, int subscribedChannels) {
// TODO Auto-generated method stub

}

@Override

```

```

public void onPSubscribe(String pattern, int subscribedChannels) {
// TODO Auto-generated method stub

}
}

```

Sample main process

```

package message.kvstore.aliyun.com;

import java.util.UUID;

import redis.clients.jedis.JedisPubSub;

public class KVStorePubSubTest {

//Information for connecting ApsaraDB for Redis, which can be obtained from the console
static final String host = "xxxxxxxxx.m.cnha.kvstore.aliyuncs.com";
static final int port = 6379;
static final String password="password";//password

public static void main(String[] args) throws Exception{
KVStorePubClient pubClient = new KVStorePubClient(host, port,password);

final String channel = "KVStore channel-A";

//The message sender starts sending messages, but there are no subscribers, the messages will not be received
pubClient.pub(channel, "Alibaba Cloud message 1: ( no subscribers , this message will not be received ) ");

//Message recipient
KVStoreSubClient subClient = new KVStoreSubClient(host, port,password);

JedisPubSub listener = new KVStoreMessageListener();
subClient.setChannelAndListener(listener, channel);

//The message recipient starts subscribing
subClient.start();

//The message sender continues sending messages
for (int i = 0; i < 5; i++) {

String message=UUID.randomUUID().toString();
pubClient.pub(channel, message);
Thread.sleep(1000);
}

//The message recipient cancels the subscription
subClient.unsubscribe(channel);

Thread.sleep(1000);
pubClient.pub(channel, "Alibaba Cloud message 2 : ( subscription canceled, this message will not be received ) ");

//The message publisher stops sending by sending a "quit" message
//When other message recipients, if any, receive "quit" in listener.onMessage(), the "unsubscribe" operation is

```

```

performed.
pubClient.close(channel);

}

}

```

Output

After you access the ApsaraDB for Redis instance with the correct address and password and run the above Java code, the following output is displayed.

```

>>> PUBLISH > Channel:KVStore Channel-A > Sends the message Alibaba Cloud Message 1: (there are no
subscribers yet, so no one would receive the message)

-----SUBSCRIBE starts-----
>>> SUBSCRIBE > Channel:KVStore Channel-A

>>> PUBLISH > Channel:KVStore Channel-A > Sends the message 0f9c2cee-77c7-4498-89a0-1dc5a2f65889
<<< SUBSCRIBE < Channel:KVStore Channel-A > Receives the message 0f9c2cee-77c7-4498-89a0-1dc5a2f65889

>>> PUBLISH > Channel:KVStore Channel-A > Sends the message ed5924a9-016b-469b-8203-7db63d06f812
<<< SUBSCRIBE < Channel:KVStore Channel-A > Receives the message ed5924a9-016b-469b-8203-7db63d06f812

>>> PUBLISH > Channel:KVStore Channel-A > Sends the message f1f84e0f-8f35-4362-9567-25716b1531cd
<<< SUBSCRIBE < Channel:KVStore Channel-A > Receives the message f1f84e0f-8f35-4362-9567-25716b1531cd

>>> PUBLISH > Channel:KVStore Channel-A > Sends the message 746bde54-af8f-44d7-8a49-37d1a245d21b
<<< SUBSCRIBE < Channel:KVStore Channel-A > Receives the message 746bde54-af8f-44d7-8a49-37d1a245d21b

>>> PUBLISH > Channel:KVStore Channel-A > Sends the message 8ac3b2b8-9906-4f61-8cad-84fc1f15a3ef
<<< SUBSCRIBE < Channel:KVStore Channel-A > Receives the message 8ac3b2b8-9906-4f61-8cad-84fc1f15a3ef

>>> UNSUBSCRIBE > Channel:KVStore Channel-A

-----SUBSCRIBE ends-----

>>> PUBLISH > Channel:KVStore Channel-A > Sends the message Alibaba Cloud Message 2: (the subscription has
been cancelled, so the message is not received)
>>> PUBLISH ends > Channel:KVStore Channel-A > Message: quit

```

The example above demonstrates a situation with one publisher and one subscriber. In actual situations, there may be many publishers and subscribers, as well as many message channels. In this case, you have to make slightly changes to the code above.

Scenario introduction

ApsaraDB for Redis provides a pipeline mechanism similar to that of Redis. A client interacts with a server through one-way pipelines, one for sending requests and the other for receiving responses.

You can send operation requests successively from the client to the server; however, the client receives the response to each request from the server until it sends a quit message to the server.

Pipelines can be very useful, for example, when several operation commands need to be quickly submitted to the server, but the responses and operation results are not required right away. In this case, Pipelines can be used as a batch processing tool to optimize the performance. The performance is enhanced mainly because the overhead of the TCP connection is reduced.

However, the client using pipelines in the app connects to the server exclusively, and non-pipeline operations are blocked until the pipelines are closed. If you need to perform other operations at the same time, you can establish a dedicated connection for pipeline operations to separate them from conventional operations.

Sample code 1

Performance comparison

```
package pipeline.kvstore.aliyun.com;

import java.util.Date;

import redis.clients.jedis.Jedis;
import redis.clients.jedis.Pipeline;

public class RedisPipelinePerformanceTest {

    static final String host = "xxxxxx.m.cnhza.kvstore.aliyuncs.com";
    static final int port = 6379;
    static final String password = "password";

    public static void main(String[] args) {

        Jedis jedis = new Jedis(host, port);

        //ApsaraDB for Redis instance password
        String authString = jedis.auth(password);// password

        if (!authString.equals("OK")) {
            System.err.println("AUTH Failed: " + authString);
            jedis.close();
            return;
        }

        //Execute several commands successively
        final int COUNT=5000;

        String key = "KVStore-Tanghan";

        // 1 ---Without using pipeline operations---
```

```

jedis.del(key);//Initializes the key

Date ts1 = new Date();
for (int i = 0; i < COUNT; i++) {
    //Sends a request and receives the response
    jedis.incr(key);
}
Date ts2 = new Date();

System.out.println("Without Pipeline > value is:"+jedis.get(key)+" > Operating time : " + (ts2.getTime() -
ts1.getTime())+ "ms");

//2 ----Using pipeline operations---

jedis.del(key);//Initializes the key

Pipeline p1 = jedis.pipelined();

Date ts3 = new Date();
for (int i = 0; i < COUNT; i++) {
    //Sends a request
    p1.incr(key);
}

//Receives the response
p1.sync();

Date ts4 = new Date();

System.out.println("Using Pipeline > value is:"+jedis.get(key)+" > Operating time : " + (ts4.getTime() -
ts3.getTime())+ "ms");

jedis.close();
}
}

```

Output 1

After you access the ApsaraDB for Redis instance with the correct address and password and run the preceding Java code, the following output is displayed. The output shows that the performance is enhanced with pipelines.

```

Without pipelines > value: 5000 > Time elapsed: 5844&nbsp;ms
With pipelines > value: 5000 > Time elapsed: 78&nbsp;ms

```

Sample code 2

With pipelines defined in Jedis, responses are processed in two methods, as shown in the following

sample code:

```
package pipeline.kvstore.aliyun.com;

import java.util.List;
import redis.clients.jedis.Jedis;
import redis.clients.jedis.Pipeline;
import redis.clients.jedis.Response;

public class PipelineClientTest {

    static final String host = "xxxxxxx.m.cnhza.kvstore.aliyuncs.com";
    static final int port = 6379;
    static final String password = "password";

    public static void main(String[] args) {

        Jedis jedis = new Jedis(host, port);

        //ApsaraDB for Redis instance password
        String authString = jedis.auth(password);// password

        if (!authString.equals("OK")) {
            System.err.println("AUTH Failed: " + authString);
            jedis.close();
            return;
        }

        String key = "KVStore-Test1";

        jedis.del(key);//Initialization

        // ----- Method 1
        Pipeline p1 = jedis.pipelined();

        System.out.println("-----Method 1-----");

        for (int i = 0; i < 5; i++) {
            p1.incr(key);
            System.out.println("Pipeline send requests");
        }

        // After sending all requests, the client starts receiving responses
        System.out.println("Sending requests completed , start to receive response");
        List<Object> responses = p1.syncAndReturnAll();

        if (responses == null || responses.isEmpty()) {
            jedis.close();
            throw new RuntimeException("Pipeline error: no response received");
        }
        for (Object resp : responses) {
            System.out.println("Pipeline received response: " + resp.toString());
        }
    }
}
```

```

System.out.println();

//----- Method 2
System.out.println("-----Method 2-----");

jedis.del(key);//Initialization

Pipeline p2 = jedis.pipelined();

//Declare the responses first
Response<Long> r1 = p2.incr(key);
System.out.println("Pipeline sending requests");

Response<Long> r2 = p2.incr(key);
System.out.println("Pipeline sending requests");

Response<Long> r3 = p2.incr(key);
System.out.println("Pipeline sending requests");

Response<Long> r4 = p2.incr(key);
System.out.println("Pipeline sending requests");

Response<Long> r5 = p2.incr(key);
System.out.println("Pipeline sending requests");

try{
r1.get(); //Errors are generated because the client has not yet started receiving responses
}catch(Exception e){
System.out.println(" <<< Pipeline error: not start to receive response yet >>> ");
}

// After sending all requests, the client starts receiving responses
System.out.println("Sending requests completed, start to receive response");
p2.sync();

System.out.println("Pipeline receiving response: " + r1.get());
System.out.println("Pipeline receiving response: " + r2.get());
System.out.println("Pipeline receiving response: " + r3.get());
System.out.println("Pipeline receiving response: " + r4.get());
System.out.println("Pipeline receiving response: " + r5.get());

jedis.close();
}

}

```

Output 2

After you access the ApsaraDB for Redis instance with the correct address and password and run the above Java code, the following output is displayed:

```
-----Method 1-----
```

```

Pipeline sends a request
Pipeline sends a request
Pipeline sends a request
Pipeline sends a request
Pipeline sends a request
After sending all requests, the client starts receiving responses
Pipeline receives response 1
Pipeline receives response 2
Pipeline receives response 3
Pipeline receives response 4
Pipeline receives response 5

-----Method 2-----
Pipeline sends a request
Pipeline sends a request
Pipeline sends a request
Pipeline sends a request
Pipeline sends a request
<<< Pipeline error: The client has not yet started receiving responses >>>
After sending all requests, the client starts receiving responses
Pipeline receives response 1
Pipeline receives response 2
Pipeline receives response 3
Pipeline receives response 4
Pipeline receives response 5

```

Scenario introduction

ApsaraDB for Redis supports a mechanism to define transactions, as in Redis. This allows you to use the MULTI, EXEC, DISCARD, WATCH, and UNWATCH commands to execute atomic transactions.

Note that the definition of transaction in Redis is slightly different from that in relational databases. If an operation fails or the transaction is cancelled by the DISCARD command, Redis does not perform transaction rollback.

Sample code 1: Two clients operate on different keys

```

package transcation.kvstore.aliyun.com;

import java.util.List;

import redis.clients.jedis.Jedis;
import redis.clients.jedis.Transaction;

public class KVStoreTranscationTest {

    static final String host = "xxxxxx.m.cnhza.kvstore.aliyuncs.com";
    static final int port = 6379;
    static final String password = "password";

```

```

/**Note that the two keys have different values
static String client1_key = "KVStore-Transcation-1";
static String client2_key = "KVStore-Transcation-2";

public static void main(String[] args) {

Jedis jedis = new Jedis(host, port);

//ApsaraDB for Redis instance password
String authString = jedis.auth(password);//password

if (!authString.equals("OK")) {
System.err.println("AUTH Failed: " + authString);
jedis.close();
return;
}

jedis.set(client1_key, "0");

//Starts another thread to simulate another client
new KVStoreTranscationTest().new OtherKVStoreClient().start();

Thread.sleep(500);

Transaction tx = jedis.multi();//Starts the transaction

//The following operations are collectively submitted to the server for processing as "atomic operations"
tx.incr(client1_key);
tx.incr(client1_key);
Thread.sleep(400);//Here, the thread's suspension has no effect on the successive operations in the transaction, but
it also suspends the operations of other threads
tx.incr(client1_key);
Thread.sleep(300);//Here, the thread's suspension has no effect on the successive operations in the transaction, but
it also suspends the operations of other threads
tx.incr(client1_key);
Thread.sleep(200);//Here, the thread's suspension has no effect on the successive operations in the transaction, but
it also suspends the operations of other threads
tx.incr(client1_key);

List<Object> result = tx.exec();//Submitted for execution

//Parses and prints out the results
for(Object rt : result){
System.out.println("Client 1 > in transaction> "+rt.toString());
}

jedis.close();
}

class OtherKVStoreClient extends Thread{
@Override
public void run() {

Jedis jedis = new Jedis(host, port);
//ApsaraDB for Redis instance password

```

```
String authString = jedis.auth(password);// password

if (!authString.equals("OK")) {
    System.err.println("AUTH Failed: " + authString);
    jedis.close();
    return;
}

jedis.set(client2_key, "100");

for (int i = 0; i < 10; i++) {
    try {
        Thread.sleep(300);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }

    System.out.println("Client 2 > "+jedis.incr(client2_key));
}

jedis.close();
}
}
```

Output 1

After you access the ApsaraDB for Redis instance with the correct address and password and run the above Java code, the following output is displayed. Here, we can see that Client 1 and Client 2 are in different threads. The operations in the transaction submitted by Client 1 are executed sequentially. Client 2 requests for operating on another key during this period, but the operation is blocked and Client 2 has to wait until all the operations in Client 1's transaction have been completed.

```
Client 2 > 101
Client 2 > 102
Client 2 > 103
Client 2 > 104
Client 1 > Transaction > 1
Client 1 > Transaction > 2
Client 1 > Transaction > 3
Client 1 > Transaction > 4
Client 1 > Transaction > 5
Client 2 > 105
Client 2 > 106
Client 2 > 107
Client 2 > 108
Client 2 > 109
Client 2 > 110
```

Sample code 2: Two clients operate on the same key

By slightly modifying the code above, we can have the two clients operate on the same key. The other parts of the code remain unchanged.

```
... ..  
  
/**Note that the values of the keys are the same  
static String client1_key = "KVStore-Transcation-1";  
static String client2_key = "KVStore-Transcation-1";  
  
... ..
```

Output 2

After running the modified Java code, the output is displayed as follows. We can see that the two clients are in different threads but operate on the same key. However, while Client 1 uses the transaction mechanism to operate on this key, Client 2 is blocked and has to wait until all the operations in Client 1's transaction are completed.

```
Client 2 > 101  
Client 2 > 102  
Client 2 > 103  
Client 2 > 104  
Client 1 > Transaction > 105  
Client 1 > Transaction > 106  
Client 1 > Transaction > 107  
Client 1 > Transaction > 108  
Client 1 > Transaction > 109  
Client 2 > 110  
Client 2 > 111  
Client 2 > 112  
Client 2 > 113  
Client 2 > 114  
Client 2 > 115
```

Background

With the 11/11 shopping spree in full swing, ApsaraDB for Redis showed that it could provide full assurance to businesses during this important and demanding event. Currently, ApsaraDB for Redis provides a standard single copy, a standard dual copy, and a cluster edition.

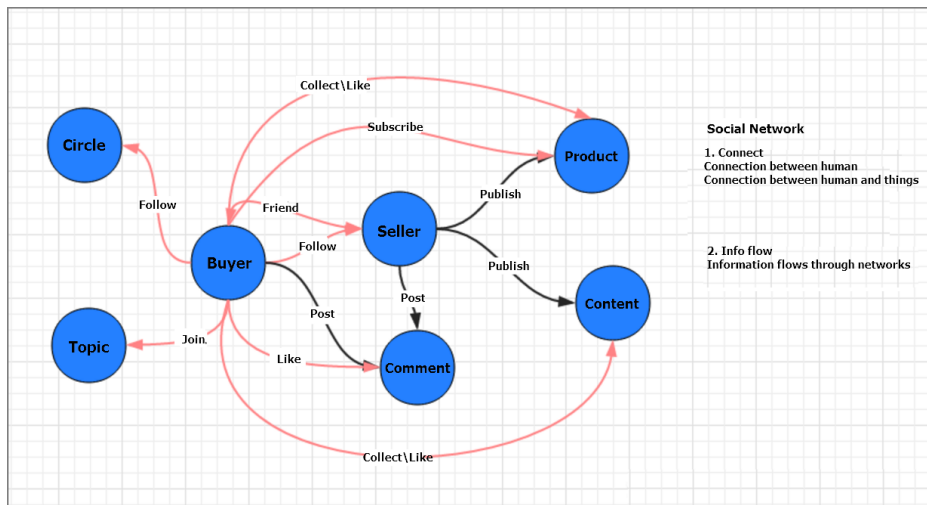
The standard single and dual copies feature high compatibility and support Lua scripts and geographic location computation. The cluster edition provides large capacity and high performance, which breaks through the performance limitations of the single-thread, standalone Redis.

ApsaraDB for Redis works in dual-host hot standby mode by default and supports backup and

recovery. It is under continuous optimization and upgrade of Alibaba Cloud's Redis team. It comes with powerful security defense capabilities. Here, we will introduce this product using several 11/11 business scenarios. These scenarios have been simplified for the purpose of easy understanding.

Storing the social relation for hundreds of millions of users in Weitao Community

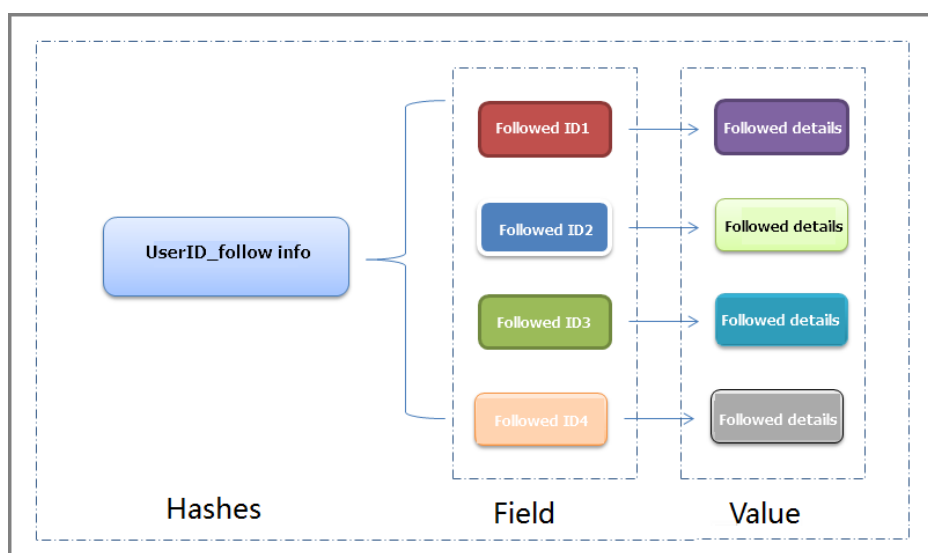
The Weitao Community carries social relation for hundreds of millions of Taobao users. Taobao users can specify their follow lists and shops have data about their followers. The overall social relation is as shown below.



If a traditional relational database model is used to express the relation chain, this will complicate business design and would turn out to be inferior performance. The Weitao Community uses a cluster of ApsaraDB for Redis to cache followers chains, which simplifies the storage of followers data and ensures a smooth business experience during 11/11. The Weitao Community uses Hashes to store the relation chains. The storage structure is as shown below and the following two interfaces are provided for querying:

Whether Users A and B are followers of each other

List of items User A is following



Paginating comments to a live broadcasting in Tmall

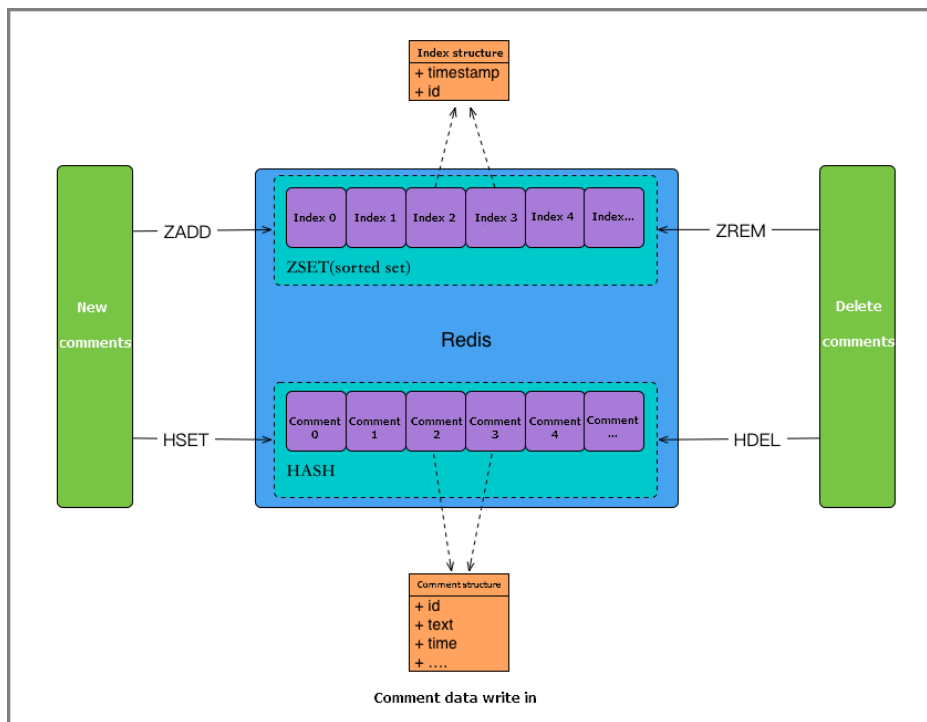
When mobile users view live broadcasts during 11/11, they can get more comments to the live broadcasts in three ways:

Incremental pull down: Gets a specified number (increment) of comments from the specified position up.

Pull-down refresh: Gets a specified number of the latest comments.

Incremental pull up: Gets a specified number (increment) of comments from the specified position down.

The wireless live broadcasting system uses Redis to optimize the business scenario. This ensures the success rate for the live comment interface and guarantees over 50,000 transactions per second (TPS) and a request response time in milliseconds. The live broadcasting system writes two sets of the data for each broadcast, indexes and comments. The indexes are written in SortedSet data structure to sort comments, while comments are stored in Hashes. A comment can be retrieved after an index is used to obtain the index ID, which is used to read the Hashes and to obtain a list of comments. A comment is written in the following process:

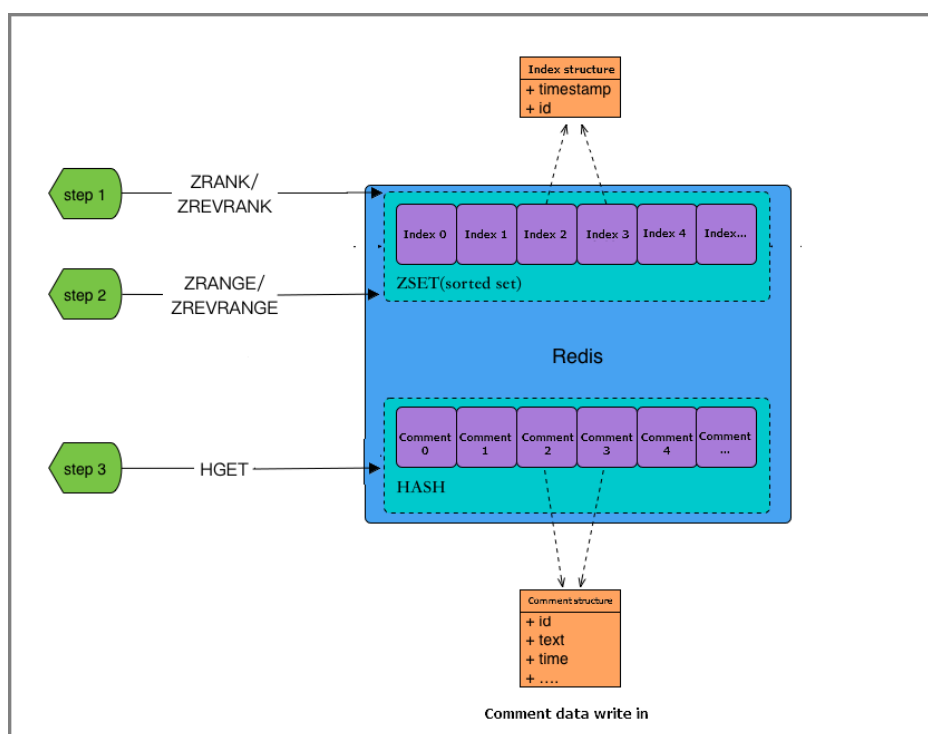


After a user refreshes the list, the background retrieves the corresponding comments. This process is as follows:

Retrieve the index ID.

Retrieve the index list.

Retrieve the comment.



Sorting orders in Cainiao document fulfillment center

After a user buys an item on 11/11, a corresponding logistics order is created, which is to be processed by Cainiao warehouse and logistics system. The decision making system generates an order fulfillment plan based on the order information so that each stage of warehouse and logistics works with each other smartly. The plan specifies the time for issuing the order to the warehouse, the time for outbound delivery, the time for item collection, and the time for delivering the item. The document fulfillment center carries out each stage of logistics services against the fulfillment plan. Due to the limited capacities of warehouses and logistics, documents to be processed first are orders considered to be of the highest priorities. Therefore, orders are sorted by priorities before being issued.

ApsaraDB for Redis is used to sort logistics orders and to determine their priorities.

