云数据库 Redis 版

最佳实践

最佳实践

场景介绍

云数据库 Redis 版在功能上与 Redis 基本一致，因此很容易用它来实现一个在线游戏中的积分排行榜功能。

代码示例

```
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)
{
//连接信息，从控制台可以获得
String host = "xxxxxxxxxx.m.cnhz1.kvstore.aliyuncs.com";
int port = 6379;

Jedis jedis = new Jedis(host, port);

try {
//实例密码
String authString = jedis.auth("password");//password

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

//Key(键)
String key = "游戏名：奔跑吧，阿里！";

//清除可能的已有数据
jedis.del(key);

//模拟生成若干个游戏玩家
```

```
List<String> playerList = new ArrayList<String>();
for (int i = 0; i < TOTAL_SIZE; ++i)
{
//随机生成每个玩家的ID
playerList.add(UUID.randomUUID().toString());
}


System.out.println("输入所有玩家 ");
//记录每个玩家的得分
for (int i = 0; i < playerList.size(); i++)
{
//随机生成数字，模拟玩家的游戏得分
int score = (int)(Math.random()*5000);

String member = playerList.get(i);

System.out.println("玩家ID：" + member + "， 玩家得分: " + score);

//将玩家的ID和得分，都加到对应key的SortedSet中去
jedis.zadd(key, score, member);

}

//输出打印全部玩家排行榜
System.out.println();
System.out.println(" "+key);
System.out.println(" 全部玩家排行榜 ");

//从对应key的SortedSet中获取已经排好序的玩家列表
Set<Tuple> scoreList = jedis.zrevrangeWithScores(key, 0, -1);
for (Tuple item : scoreList) {
System.out.println("玩家ID："+item.getElement()+"， 玩家得分:"+Double.valueOf(item.getScore()).intValue());
}

//输出打印Top5玩家排行榜
System.out.println();
System.out.println(" "+key);
System.out.println(" Top 玩家");

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

//输出打印特定玩家列表
System.out.println();
System.out.println(" "+key);
System.out.println(" 积分在1000至2000的玩家");

//从对应key的SortedSet中获取已经积分在1000至2000的玩家列表
scoreList = jedis.zrangeByScoreWithScores(key, 1000, 2000);
for (Tuple item : scoreList) {
System.out.println("玩家ID："+item.getElement()+"， 玩家得分:"+Double.valueOf(item.getScore()).intValue());
}
```

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

运行结果

在输入了正确的云数据库 Redis 版实例访问地址和密码之后，运行以上 Java 程序，输出结果如下：

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

游戏名：奔跑吧，阿里！
全部玩家排行榜
玩家ID：db03520b-75a3-48e5-850a-071722ff7afb， 玩家得分:4853
玩家ID：99bc5b4f-e32a-4295-bc3a-0324887bb77e， 玩家得分:4852
玩家ID：2e4ec631-1e4e-4ef0-914f-7bf1745f7d65， 玩家得分:4478
玩家ID：ec0f06da-91ee-447b-b935-7ca935dc7968， 玩家得分:4391
玩家ID：e3e8e1fa-6aac-4a0c-af80-4c4a1e126cd1， 玩家得分:4064
玩家ID：9193e26f-6a71-4c76-8666-eaf8ee97ac86， 玩家得分:3860
玩家ID：34574e3e-9cc5-43ed-ba15-9f5405312692， 玩家得分:3734
玩家ID：cb62bb24-1318-4af2-9d9b-fbff7280dbec， 玩家得分:3405
玩家ID：19e2aa6b-a2d8-4e56-bdf7-8b59f64bd8e0， 玩家得分:3394
玩家ID：d302d24d-d380-4e15-a4d6-84f71313f27a， 玩家得分:2931
玩家ID：2c814a6f-3706-4280-9085-5fe5fd56b71c， 玩家得分:2510
玩家ID：a6299dd2-4f38-4528-bb5a-aa2d48a9f94a， 玩家得分:2428
玩家ID：674bbdd1-2023-46ae-bbe6-dfcd8e372430， 玩家得分:2265
玩家ID：ec24fb9e-366e-4b89-a0d5-0be151a8cad0， 玩家得分:2263
玩家ID：e11ecc2c-cd51-4339-8412-c711142ca7aa， 玩家得分:1848
玩家ID：bee46f9d-4b05-425e-8451-8aa6d48858e6， 玩家得分:1796
玩家ID：24235a85-85b9-476e-8b96-39f294f57aa7， 玩家得分:1655
```

```
玩家ID：0293b43a-1554-4157-a95b-b78de9edf6dd， 玩家得分:1008
玩家ID：4c396f67-da7c-4b99-a783-25919d52d756， 玩家得分:958
玩家ID：9ee2ed6d-08b8-4e7f-b52c-9adfe1e32dda， 玩家得分:63

游戏名：奔跑吧，阿里！
Top 玩家
玩家ID：db03520b-75a3-48e5-850a-071722ff7afb， 玩家得分:4853
玩家ID：99bc5b4f-e32a-4295-bc3a-0324887bb77e， 玩家得分:4852
玩家ID：2e4ec631-1e4e-4ef0-914f-7bf1745f7d65， 玩家得分:4478
玩家ID：ec0f06da-91ee-447b-b935-7ca935dc7968， 玩家得分:4391
玩家ID：e3e8e1fa-6aac-4a0c-af80-4c4a1e126cd1， 玩家得分:4064

游戏名：奔跑吧，阿里！
积分在1000至2000的玩家
玩家ID：0293b43a-1554-4157-a95b-b78de9edf6dd， 玩家得分:1008
玩家ID：24235a85-85b9-476e-8b96-39f294f57aa7， 玩家得分:1655
玩家ID：bee46f9d-4b05-425e-8451-8aa6d48858e6， 玩家得分:1796
玩家ID：e11ecc2c-cd51-4339-8412-c711142ca7aa， 玩家得分:1848
```

场景介绍

云数据库 Redis 版在功能上与 Redis 基本一致，因此很容易利用它来实现一个网上商城的商品相关性分析程序。

商品的相关性就是某个产品与其他另外某商品同时出现在购物车中的情况。这种数据分析对于电商行业是很重要的，可以用来分析用户购买行为。例如：

在某一商品的 detail 页面，推荐给用户与该商品相关的其他商品；

在添加购物车成功页面，当用户把一个商品添加到购物车，推荐给用户与之相关的其他商品；

在货架上将相关性比较高的几个商品摆放在一起。

利用云数据库 Redis 版的有序集合，为每种商品构建一个有序集合，集合的成员为和该商品同时出现在购物车中的商品，成员的 score 为同时出现的次数。每次 A 和 B 商品同时出现在购物车中时，分别更新云数据库 Redis 版中 A 和 B 对应的有序集合。

代码示例

```
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)
{
//ApsaraDB for Redis的连接信息，从控制台可以获得
String host = "xxxxxxxx.m.cnhza.kvstore.aliyuncs.com";
int port = 6379;

Jedis jedis = new Jedis(host, port);

try {
//ApsaraDB for Redis的实例密码
String authString = jedis.auth("password");//password

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

//产品列表
String key0="阿里云:产品:啤酒";
String key1="阿里云:产品:巧克力";
String key2="阿里云:产品:可乐";
String key3="阿里云:产品:口香糖";
String key4="阿里云:产品:牛肉干";
String key5="阿里云:产品:鸡翅";


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

//初始化，清除可能的已有旧数据
for (int i = 0; i < aliyunProducts.length; i++) {
jedis.del(aliyunProducts[i]);
}


//模拟用户购物
for (int i = 0; i < 5; i++) {//模拟多人次的用户购买行为

customersShopping(aliyunProducts,i,jedis);

}

System.out.println();

//利用ApsaraDB for Redis来输出各个商品间的关联关系
for (int i = 0; i < aliyunProducts.length; i++) {
System.out.println(">>>>>>>>>>>与"+aliyunProducts[i]+"一起被购买的产品有<<<<<<<<<<<<<<<");
Set<Tuple> relatedList = jedis.zrevrangeWithScores(aliyunProducts[i], 0, -1);

for (Tuple item : relatedList) {
System.out.println("商品名称："+item.getElement()+"， 共同购买次数:"+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) {

//简单模拟3种购买行为，随机选取作为用户的购买选择
int bought=(int)(Math.random()*3);

if(bought==1){
//模拟业务逻辑：用户购买了如下产品
System.out.println("用户"+i+"购买了"+products[0]+","+products[2]+","+products[1]);

//将产品之间的关联情况记录到ApsaraDB for Redis的SortSet之中
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){
//模拟业务逻辑：用户购买了如下产品
System.out.println("用户"+i+"购买了"+products[4]+","+products[2]+","+products[3]);

//将产品之间的关联情况记录到ApsaraDB for Redis的SortSet之中
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){
//模拟业务逻辑：用户购买了如下产品
System.out.println("用户"+i+"购买了"+products[1]+","+products[5]);

//将产品之间的关联情况记录到ApsaraDB for Redis的SortSet之中
jedis.zincrby(products[5], 1, products[1]);
jedis.zincrby(products[1], 1, products[5]);
}

}
}
```

运行结果

在输入了正确的云数据库 Redis 版实例访问地址和密码之后，运行以下 Java 程序，输出结果如下：

```
用户0购买了阿里云:产品:巧克力,阿里云:产品:鸡翅
用户1购买了阿里云:产品:牛肉干,阿里云:产品:可乐,阿里云:产品:口香糖
用户2购买了阿里云:产品:啤酒,阿里云:产品:可乐,阿里云:产品:巧克力
用户3购买了阿里云:产品:牛肉干,阿里云:产品:可乐,阿里云:产品:口香糖
用户4购买了阿里云:产品:巧克力,阿里云:产品:鸡翅

>>>>>>>>>>与阿里云:产品:啤酒一起被购买的产品有<<<<<<<<<<<<<<<
商品名称：阿里云:产品:巧克力， 共同购买次数:1
商品名称：阿里云:产品:可乐， 共同购买次数:1

>>>>>>>>>>与阿里云:产品:巧克力一起被购买的产品有<<<<<<<<<<<<<<<
商品名称：阿里云:产品:鸡翅， 共同购买次数:2
商品名称：阿里云:产品:啤酒， 共同购买次数:1
商品名称：阿里云:产品:可乐， 共同购买次数:1

>>>>>>>>>>与阿里云:产品:可乐一起被购买的产品有<<<<<<<<<<<<<<<
商品名称：阿里云:产品:牛肉干， 共同购买次数:2
商品名称：阿里云:产品:口香糖， 共同购买次数:2
商品名称：阿里云:产品:巧克力， 共同购买次数:1
商品名称：阿里云:产品:啤酒， 共同购买次数:1

>>>>>>>>>>与阿里云:产品:口香糖一起被购买的产品有<<<<<<<<<<<<<<<
商品名称：阿里云:产品:牛肉干， 共同购买次数:2
商品名称：阿里云:产品:可乐， 共同购买次数:2

>>>>>>>>>>与阿里云:产品:牛肉干一起被购买的产品有<<<<<<<<<<<<<<<
商品名称：阿里云:产品:可乐， 共同购买次数:2
商品名称：阿里云:产品:口香糖， 共同购买次数:2

>>>>>>>>>>与阿里云:产品:鸡翅一起被购买的产品有<<<<<<<<<<<<<<<
商品名称：阿里云:产品:巧克力， 共同购买次数:2
```

场景介绍

云数据库 Redis 版也提供了与 Redis 相同的消息发布（pub）与订阅（sub）功能。即一个 client 发布消息，其他多个 client 订阅消息。

需要注意的是，云数据库 Redis 版发布的消息是“非持久”的，即消息发布者只负责发送消息，而不管消息是否有接收方，也不会保存之前发送的消息，即发布的消息“即发即失”；消息订阅者也只能得到订阅之后的消息，频道（channel）中此前的消息将无从获得。

此外，消息发布者（即 publish 客户端）无需独占与服务器端的连接，您可以在发布消息的同时，使用同一个客户端连接进行其他操作（例如 List 操作等）。但是，消息订阅者（即 subscribe 客户端）需要独占与服务器端的连接，即进行 subscribe 期间，该客户端无法执行其他操作，而是以阻塞的方式等待频道（channel）中的消息；因此消息订阅者需要使用单独的服务器连接，或者需要在单独的线程中使用（参见如下示例）。

代码示例

消息发布者（即 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的实例密码
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(" >>> 发布(PUBLISH) > Channel:"+channel+" > 发送出的Message:"+message);

jedis.publish(channel, message);
}

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

//消息发布者结束发送，即发送一个“quit”消息；
jedis.publish(channel, "quit");

}

}
```

消息订阅者 (即 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的实例密码
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(" >>> 订阅(SUBSCRIBE) > Channel:"+channel);
System.out.println();

//接收者在侦听订阅的消息时，将会阻塞进程，直至接收到quit消息（被动方式），或主动取消订阅
jedis.subscribe(listener, channel);
}

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

listener.unsubscribe(channel);
}

@Override
public void run() {
try{

System.out.println();
System.out.println("----------订阅消息SUBSCRIBE 开始-------");

subscribe();

System.out.println("----------订阅消息SUBSCRIBE 结束-------");
System.out.println();
}catch(Exception e){
e.printStackTrace();
}

}

}
```

消息监听者

```
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(" <<< 订阅(SUBSCRIBE)< Channel:" + channel + " >接收到的Message:" + message );
System.out.println();

//当接收到的message为quit时，取消订阅(被动方式)
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

}
}
```

示例主程序

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

import java.util.UUID;

import redis.clients.jedis.JedisPubSub;
```

```
public class KVStorePubSubTest {

//ApsaraDB for Redis的连接信息，从控制台可以获得
static final String host = "xxxxxxxxxx.m.cnhza.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频道-A";

//消息发送者开始发消息，此时还无人订阅，所以此消息不会被接收
pubClient.pub(channel, "Aliyun消息1：（此时还无人订阅，所以此消息不会被接收）");

//消息接收者
KVStoreSubClient subClient = new KVStoreSubClient(host, port,password);

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

//消息接收者开始订阅
subClient.start();


//消息发送者继续发消息
for (int i = 0; i < 5; i++) {

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

//消息接收者主动取消订阅
subClient.unsubscribe(channel);

Thread.sleep(1000);
pubClient.pub(channel, "Aliyun消息2：（此时订阅取消，所以此消息不会被接收）");

//消息发布者结束发送，即发送一个“quit”消息；
//此时如果有其他的消息接收者，那么在listener.onMessage()中接收到“quit”时，将执行“unsubscribe”操作。
pubClient.close(channel);

}

}
```

运行结果

在输入了正确的云数据库 Redis 版实例访问地址和密码之后，运行以上 Java 程序，输出结果如下。

```
>>> 发布(PUBLISH) > Channel:KVStore频道-A > 发送出的Message:Aliyun消息1：（此时还无人订阅，所以此消息不会被接收）
```

```
----------订阅消息SUBSCRIBE 开始-------
>>> 订阅(SUBSCRIBE) > Channel:KVStore频道-A

>>> 发布(PUBLISH) > Channel:KVStore频道-A > 发送出的Message:0f9c2cee-77c7-4498-89a0-1dc5a2f65889
<<< 订阅(SUBSCRIBE)< Channel:KVStore频道-A >接收到的Message:0f9c2cee-77c7-4498-89a0-1dc5a2f65889

>>> 发布(PUBLISH) > Channel:KVStore频道-A > 发送出的Message:ed5924a9-016b-469b-8203-7db63d06f812
<<< 订阅(SUBSCRIBE)< Channel:KVStore频道-A >接收到的Message:ed5924a9-016b-469b-8203-7db63d06f812

>>> 发布(PUBLISH) > Channel:KVStore频道-A > 发送出的Message:f1f84e0f-8f35-4362-9567-25716b1531cd
<<< 订阅(SUBSCRIBE)< Channel:KVStore频道-A >接收到的Message:f1f84e0f-8f35-4362-9567-25716b1531cd

>>> 发布(PUBLISH) > Channel:KVStore频道-A > 发送出的Message:746bde54-af8f-44d7-8a49-37d1a245d21b
<<< 订阅(SUBSCRIBE)< Channel:KVStore频道-A >接收到的Message:746bde54-af8f-44d7-8a49-37d1a245d21b

>>> 发布(PUBLISH) > Channel:KVStore频道-A > 发送出的Message:8ac3b2b8-9906-4f61-8cad-84fc1f15a3ef
<<< 订阅(SUBSCRIBE)< Channel:KVStore频道-A >接收到的Message:8ac3b2b8-9906-4f61-8cad-84fc1f15a3ef

>>> 取消订阅(UNSUBSCRIBE) > Channel:KVStore频道-A

----------订阅消息SUBSCRIBE 结束-------

>>> 发布(PUBLISH) > Channel:KVStore频道-A > 发送出的Message:Aliyun消息2：（此时订阅取消，所以此消息不会被接收）
>>> 发布(PUBLISH)结束 > Channel:KVStore频道-A > Message:quit
```

以上示例中仅演示了一个发布者与一个订阅者的情况，实际上发布者与订阅者都可以为多个，发送消息的频道（channel）也可以是多个，对以上代码稍作修改即可。

场景介绍

云数据库 Redis 版提供了与 Redis 相同的管道传输（pipeline）机制。管道（pipeline）将客户端 client 与服务器端的交互明确划分为单向的发送请求（Send Request）和接收响应（Receive Response）：用户可以将多个操作连续发给服务器，但在此期间服务器端并不对每个操作命令发送响应数据；全部请求发送完毕后用户关闭请求，开始接收响应获取每个操作命令的响应结果。

管道（pipeline）在某些场景下非常有用，比如有多个操作命令需要被迅速提交至服务器端，但用户并不依赖每个操作返回的响应结果，对结果响应也无需立即获得，那么管道就可以用来作为优化性能的批处理工具。性能提升的原因主要是减少了 TCP 连接中交互往返的开销。

不过在程序中使用管道请注意，使用 pipeline 时客户端将独占与服务器端的连接，此期间将不能进行其他“非管道”类型操作，直至 pipeline 被关闭；如果要同时执行其他操作，可以为 pipeline 操作单独建立一个连接，将其与常规操作分离开来。

代码示例1

性能对比.

```
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的实例密码
String authString = jedis.auth(password);// password

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

//连续执行多次命令操作
final int COUNT=5000;

String key = "KVStore-Tanghan";

// 1 ---不使用pipeline操作---

jedis.del(key);//初始化key

Date ts1 = new Date();
for (int i = 0; i < COUNT; i++) {
//发送一个请求，并接收一个响应（Send Request and Receive Response）
jedis.incr(key);

}
Date ts2 = new Date();

System.out.println("不用Pipeline > value为:"+jedis.get(key)+" > 操作用时：" + (ts2.getTime() - ts1.getTime())+ "ms");


//2 ----对比使用pipeline操作---

jedis.del(key);//初始化key

Pipeline p1 = jedis.pipelined();

Date ts3 = new Date();
```

```
for (int i = 0; i < COUNT; i++) {
//发出请求 Send Request
p1.incr(key);
}

//接收响应 Receive Response
p1.sync();

Date ts4 = new Date();

System.out.println("使用Pipeline > value为:"+jedis.get(key)+" > 操作用时：" + (ts4.getTime() - ts3.getTime())+ "ms");

jedis.close();

}

}
```

运行结果1

在输入了正确的云数据库 Redis 版实例访问地址和密码之后，运行以上 Java 程序，输出结果如下。从中可以看出使用 pipeline 的性能要快的多。

```
不用Pipeline > value为:5000 > 操作用时：5844ms
使用Pipeline > value为:5000 > 操作用时：78ms
```

代码示例2

在 Jedis 中使用管道（pipeline）时，对于响应数据（response）的处理有两种方式，请参考以下代码示例。

```
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 = "xxxxxxxx.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的实例密码
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);//初始化

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

System.out.println("-----方法1-----");

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


// 发送请求完成，开始接收响应
System.out.println("发送请求完成，开始接收响应");
List<Object> responses = p1.syncAndReturnAll();

if (responses == null || responses.isEmpty()) {
jedis.close();
throw new RuntimeException("Pipeline error: 没有接收到响应");
}
for (Object resp : responses) {
System.out.println("Pipeline接收响应Response: " + resp.toString());
}
System.out.println();

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

jedis.del(key);//初始化

Pipeline p2 = jedis.pipelined();

//需要先声明Response
Response<Long> r1 = p2.incr(key);
System.out.println("Pipeline发送请求");

Response<Long> r2 = p2.incr(key);
System.out.println("Pipeline发送请求");

Response<Long> r3 = p2.incr(key);
System.out.println("Pipeline发送请求");

Response<Long> r4 = p2.incr(key);
System.out.println("Pipeline发送请求");

Response<Long> r5 = p2.incr(key);
```

```
System.out.println("Pipeline发送请求");

try{
r1.get(); //此时还未开始接收响应，所以此操作会出错
}catch(Exception e){
System.out.println(" <<< Pipeline error：还未开始接收响应 >>> ");
}


// 发送请求完成，开始接收响应
System.out.println("发送请求完成，开始接收响应");
p2.sync();

System.out.println("Pipeline接收响应Response: " + r1.get());
System.out.println("Pipeline接收响应Response: " + r2.get());
System.out.println("Pipeline接收响应Response: " + r3.get());
System.out.println("Pipeline接收响应Response: " + r4.get());
System.out.println("Pipeline接收响应Response: " + r5.get());

jedis.close();
}

}
```

运行结果2

在输入了正确的云数据库 Redis 版实例访问地址和密码之后，运行以上 Java 程序，输出结果如下:

```
-----方法1-----
Pipeline发送请求
Pipeline发送请求
Pipeline发送请求
Pipeline发送请求
Pipeline发送请求
发送请求完成，开始接收响应
Pipeline接收响应Response: 1
Pipeline接收响应Response: 2
Pipeline接收响应Response: 3
Pipeline接收响应Response: 4
Pipeline接收响应Response: 5

-----方法2-----
Pipeline发送请求
Pipeline发送请求
Pipeline发送请求
Pipeline发送请求
Pipeline发送请求
<<< Pipeline error：还未开始接收响应 >>>
发送请求完成，开始接收响应
Pipeline接收响应Response: 1
Pipeline接收响应Response: 2
Pipeline接收响应Response: 3
Pipeline接收响应Response: 4
Pipeline接收响应Response: 5
```

场景介绍

云数据库 Redis 版支持 Redis 中 定义的“事务（transaction）”机制，即用户可以使用 MULTI，EXEC，DISCARD，WATCH，UNWATCH 指令用来执行原子性的事务操作。

需要强调的是，Redis 中定义的**事务**，并不是关系数据库中严格意义上的事务。当 Redis 事务中的某个操作执行失败，或者用 DISCARD 取消事务时候，Redis 并不执行“事务回滚”，在使用时要注意这点。

代码示例1：两个 client 操作不同的 key

```
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";

//**注意这两个key的内容是不同的
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的实例密码
String authString = jedis.auth(password);//password

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

jedis.set(client1_key, "0");

//启动另一个thread，模拟另外的client
new KVStoreTranscationTest().new OtherKVStoreClient().start();

Thread.sleep(500);

Transaction tx = jedis.multi();//开始事务

//以下操作会集中提交服务器端处理，作为“原子操作”
```

```
tx.incr(client1_key);
tx.incr(client1_key);
Thread.sleep(400);//此处Thread的暂停对事务中前后连续的操作并无影响，其他Thread的操作也无法执行
tx.incr(client1_key);
Thread.sleep(300);//此处Thread的暂停对事务中前后连续的操作并无影响，其他Thread的操作也无法执行
tx.incr(client1_key);
Thread.sleep(200);//此处Thread的暂停对事务中前后连续的操作并无影响，其他Thread的操作也无法执行
tx.incr(client1_key);

List<Object> result = tx.exec();//提交执行

//解析并打印出结果
for(Object rt : result){
System.out.println("Client 1 > 事务中> "+rt.toString());
}

jedis.close();
}

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

Jedis jedis = new Jedis(host, port);
// ApsaraDB for Redis的实例密码
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();
}
}
}
```

运行结果1

在输入了正确的云数据库 Redis 版实例访问地址和密码之后，运行以上 Java 程序，输出结果如下。从中可以看到 client1 和 client2 在两个不同的 Thread 中，client1 所提交的事务操作都是集中顺序执行的，在此期间

尽管 client2 是对另外一个 key 进行操作，它的命令操作也都被阻塞等待，直至 client1 事务中的全部操作执行完毕。

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

代码示例2：两个 client 操作相同的 key

对以上的代码稍作改动，使得两个 client 操作同一个 key，其余部分保持不变。

```
    ... ...

//**注意这两个key的内容现在是相同的
static String client1_key = "KVStore-Transcation-1";
static String client2_key = "KVStore-Transcation-1";

... ...
```

运行结果2

再次运行修改后的此 Java 程序，输出结果如下。可以看到不同 Thread 中的两个 client 在操作同一个 key，但是当 client1 利用事务机制来操作这个 key 时，client2 被阻塞不得不等待 client1 事务中的操作完全执行完毕。

```
Client 2 > 101
Client 2 > 102
Client 2 > 103
Client 2 > 104
Client 1 > 事务中> 105
Client 1 > 事务中> 106
Client 1 > 事务中> 107
Client 1 > 事务中> 108
Client 1 > 事务中> 109
Client 2 > 110
Client 2 > 111
```

```
Client 2 > 112
Client 2 > 113
Client 2 > 114
Client 2 > 115
```

数据集成简介

数据集成（Data Integration）是阿里集团对外提供的可跨异构数据存储系统的、可靠、安全、低成本、可弹性扩展的数据同步平台，为20多种数据源提供不同网络环境下的离线(全量/增量)数据进出通道。详细的数据源类型列表请参考支持的数据源类型。您可以通过数据集成向云数据库 Redis 版进行数据的导入数据。

一、创建 Redis 数据源

Redis 数据源支持写入 Redis 的通道，可以通过脚本模式配置同步任务 。

注意：

只有项目管理员角色才能够新建数据源，其他角色的成员仅能查看数据源。

如您想用子账号创建数据集成任务，需赋予子账号相应的权限。具体请参考：开通阿里云主账号、设置子账号。

操作步骤

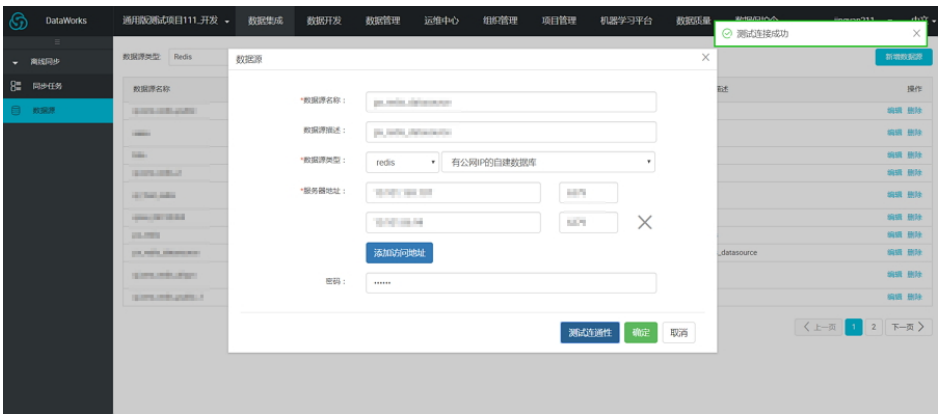
1. 以开发者身份进入阿里云数加平台，单击**项目列表**下对应**项目操作**栏中的**进入工作区**。

单击顶部菜单栏中**数据集成**模块的**数据源**。

单击**新增数据源**。

在**新建数据源**对话框中，选择**数据源类型**为 **Redis**。

配置 Redis 数据源的各个信息项，如下图所示。



注意：若账号没有授权数据集成默认角色，需要前往 RAM 进行角色授权。

配置项具体说明如下：

数据源名称： 由英文字母、数字、下划线组成且需以字符或下划线开头，长度不超过60个字符。

数据源描述： 对数据源进行简单描述，不得超过80个字符 。

数据源类型：当前选择的数据源类型为 Redis：有公网IP的自建数据库。

服务地址：格式为 host:port。

添加访问地址：添加访问地址，格式为 host:port。

密码：数据库对应的密码 。

完成上述信息项的配置后，单击**测试连通性**。

测试连通性通过后，单击**确定**。

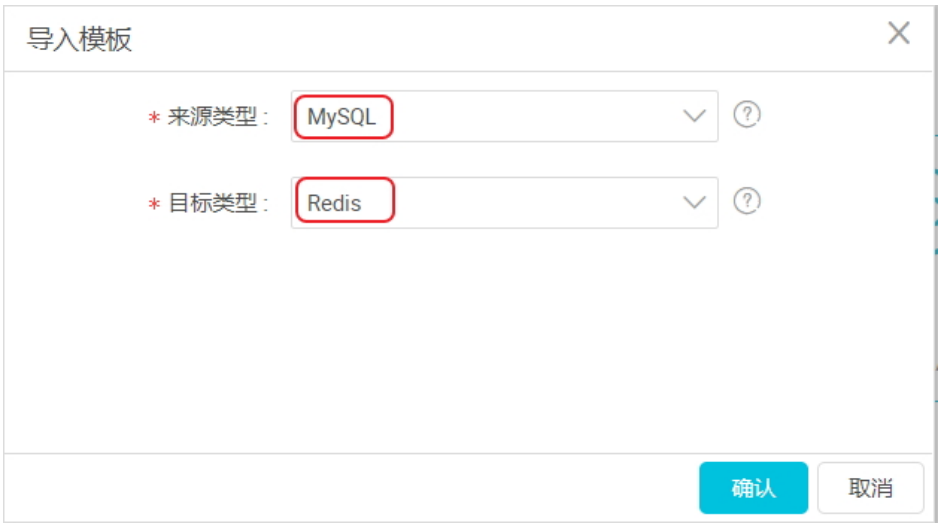
二、配置脚本模式的同步任务

以项目管理员身份进入**数加管理控制台**，单击**大数据开发套件**下对应**项目操作**栏中的**进入工作区**。

进入顶部菜单栏中的**数据集成**页面，选择**脚本模式**，如下图。

说明：Redis 不支持向导模式。进入脚本界面你可以选择相应的模板，此模板包含了同步任务的主要参数，将相关的信息填写完整，但是脚本模式不能转化成向导模式。

在**导入模板**对话框中选择需要的**来源类型**和**目标类型**，并单击**确认**。如下图所示：



在**脚本模式配置**页面，根据自身情况进行配置，如有问题可单击右上方的 **Redis Writer 帮助手册**进

行查看。如下图所示：

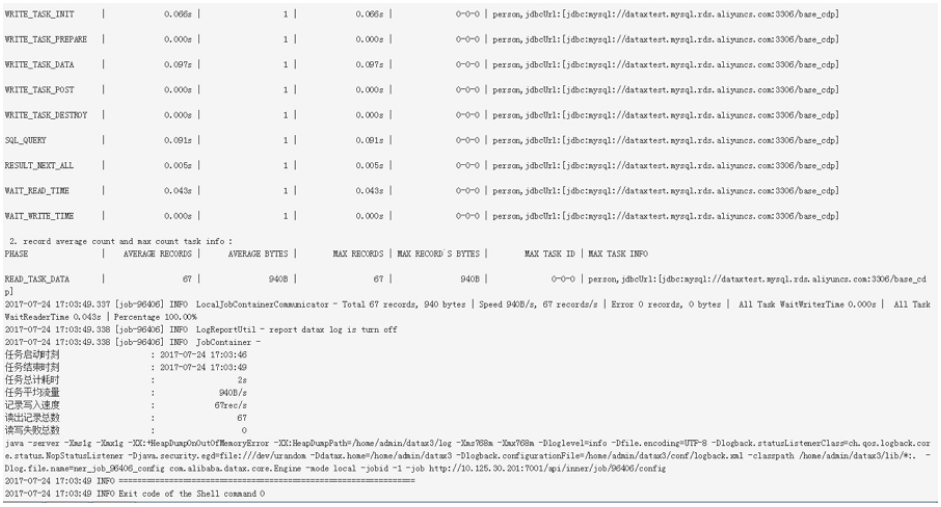
说明： RedisWriter 脚本案例如下：

```
{
"type": "job",
"configuration": {
"setting": {
"speed": {
"concurrent": "1",//并发数
"mbps": "1"//同步能达到的最大数率
},
"errorLimit": {
"record": "0"
}
},
"reader": {
"parameter": {
"splitPk": "id",//切分键
"column": [
"id",
"name",
"year"
],
"table": "person",//表名
"where": "",//
"datasource": "px_mysql"//数据源名，建议数据源都先添加数据源后再配置同步任务,此配置项填写的内容必须要与添加的数据源名称保持一致
},
"plugin": "mysql"
```

```
},
"writer": {
"parameter": {
"expireTime": {
"seconds": "1000"//相对当前时间的秒数，该时间指定了从现在开始多长时间后数据失效
},
"keyFieldDelimiter": "\u0001",//写入 redis 的 key 分隔符。比如: key=key1\u0001id,如果 key 有多个需要拼接时，该值为必填项，如果 key 只有一个则可以忽略该配置项。
"writeMode": {
"valueFieldDelimiter": "\u0001",//value 类型是 string 时，value 之间的分隔符，比如 value1\u0001value2\u0001value3；
"type": "string",//value类型
"mode": "set"//写入的模式,存储这个数据，如果已经存在则覆盖
},
"batchSize": "1000",//一次性批量提交的记录数大小
"dateFormat": "yyyy-MM-dd HH:mm:ss",//时间格式
"keyIndexes": [
0,
1
],//keyIndexes 表示源端哪几列需要作为 key（第一列是从 0 开始），如果是第一列和第二列需要组合作为 key，那么 keyIndexes 的值则为 [0,1]。

"datasource": "px_redis_datasource"//数据源名，建议数据源都先添加数据源后再配置同步任务,此配置项填写的内容必须要与添加的数据源名称保持一致
},
"plugin": "redis"
}
},
"version": "1.0"
}
```

运行结果如下：



- RedisWriter 参数说明请参考 RedisWriter 配置。

首先，我们来理解下 Redis 中 timeout 的概念。

什么是 timeout 呢?

所谓 timeout 就是超过业务方设定的超时时间依然得不到响应，就可以认为是 timeout 了，因为一般的 Redis 请求基本上都是毫秒级别的，而业务一般会设置几十秒甚至一二百秒作为超时时间限制。一但 Redis 的延时过高，超过了业务设定的时间限制，就会出现 timeout 问题。

造成 timeout 超时的原因可能有哪些呢？

在部署了监控系统的前提下，可以先通过监控数据进行分析，寻找问题的方向。

如果没能得到什么有价值的信息的话，也可以从以下几个方面进行排查：

一、慢查询

确认是否使用慢查询,可以使用slowlog get num查看相应的慢命令：

127.0.0.1:6379> SLOWLOG get 10

1) (integer) 4 #慢操作索引

2) (integer) 1466059344 #事件发生的Unix时间戳

3) (integer) 250133 #事件耗时微妙(0.25s,250ms,250133us)

4) “debug” #时间具体命令以及相应参数

2) “sleep”

3) “.25”

127.0.0.1:6379> CONFIG SET slowlog-log-slower-than 1000 #设置当key的操作超过多长时间就会别加入到slowlog队列 默认单位us(0.001s) 默认是超过10ms。

注意：

unix时间戳转换方式：

date -d@’1466059344’ “+ %Y-%m-%d %H:%M:%S”

date -d”2016-06-16 16:50:50” ‘+%s’

注意：一般大量的key删除操作，以及keys遍历操作都可能会造成超时

参考：http://redis.io/commands/slowlog

二、几个关键项

查看redis状态的几个关键项：内存使用情况，当前链接客户端数量，ops等；

使用info命令其实就可以看出来具体的状态信息，具体后续再分析。

三、透明大页

查看透明大页是否禁止掉（Transparent huge pages）；

官方建议是禁止掉比较好，线上测试其实效果不是特别明显；

Linux 下查看默认内存页大小（getconf PAGESIZE），默认是4K；

设置 hugepage 的数量；sysctl vm.nr_hugepages = 1024

echo never > /sys/kernel/mm/transparent_hugepage/enabled

四、是否为虚拟机

查看 Redis 主机是否为虚拟机，这样会有内在延迟；

测试延迟：

./redis-cli --intrinsic-latency 100

这个命令可以在 server 段进行判断是否 Redis 有延迟，在客户端通过 -h -p 参数可以进行对比一下是否为网络上的影响。

五、启用延迟监控

启用延迟监控：

latency monitoring:

可以找出相应的延迟的敏感代码路径；

延迟记录按照不同的时间进行时间流分隔统计；

从时间序列中获取原始数据并进行报表；

分析报表并提供人类可读的报告，并且根据度量值进行判断。

时间序列（time series）：

每个时间延迟任务都会记录到时间序列

每个时间序列包含160个元素

每个元素为一组：unix 时间戳和事件执行所消耗的时间长

同一时间同一事件发生的延迟时间是会记录在一个时间序列。因此即使一个给定事件连续的延迟是可以衡量的(比如用户设置临界值太低)，也至少得180s才能可达

最大延迟时间中的每一个元素都会被记录下来

怎样启用 latency monitoring:

首先对于用户场景来说，什么是高延迟。应用请求查询少于1ms并且短时间有比较少客户端的应用经历2m的延迟是可以接受的。

因此，开启 latency monitor 首先需要设置延迟时间阈值在毫秒级别（latency threshold）。

当然了，也需要根据实际情况进行设定相应的值了。

CONFIG SET latency-monitor-threshold 100

（latency-monitor 的阈值不能大于 slowlog 的值）

注意：延迟监控所需要的内存是非常小的，能增加内存是最好。

latency 命令报告出来的相关信息：

用户接口使用 latency 命令进行调用，同时参数后面可以接很多其他的子命令 latency latest 记录最后的延迟事件的记录。每个事件包含以下几个变量：(事件名、事件延迟状态时间戳、延迟时间 ms、此时间最大延迟)：

127.0.0.1:6379> latency latest

1) “command” #事件为command

2) (integer) 1466059344 #2016-6-16 14:42:24

3) (integer) 250 #延迟时间0.25s

4) (integer) 1000 #command事件延迟最大时间为1s

latency history events 此命令可以打印相应延迟事件相关的时间和耗时

127.0.0.1:6379> latency history command

1) 1) (integer) 1466059051

2) (integer) 10

2) 1) (integer) 1466059324

2) (integer) 13

3) 1) (integer) 1466059332

2) (integer) 1000

4) 1) (integer) 1466059344

2) (integer) 250

latency reset cevents 重置事件的相关延迟操作，清空记录：

127.0.0.1:6379> latency reset command

(integer) 1

127.0.0.1:6379> latency graph command

command - high 700 ms, low 100 ms (all time high 700 ms)

o#

_#||

o||||

_#|||||

2222111

6421961

sssssss

另外，一般情况下大量的删除，过期以及淘汰（由 maxmemory-policy 控制的）的大对象，也会造成Redis阻塞，进而造成相应的延迟。如果经常有比较大的对象进行删除，过期和淘汰的，建议将这些对象分割成一些小对象。即对比较大的 key 进行拆分。

参考：http://redis.io/topics/latency-monitor

六、持久化对延迟

持久化对延迟造成的影响：

一般情况来说，持久话的也会影响延迟，因为持久话操作必须对内存中的数据线进行一次 save 操作。因此必须根据以下相关的参数进行持久性（durability）和延迟/性能（latency/performance）做相应的权衡：

AOF + fsync always：这种方式会比较慢一些；

AOF + fsync every second：这个将是折中的一种方案；

AOF + fsync every second + no-appendfsync-on-rewrite option set to yes：比较好的一种方式，但是需要避免在往磁盘同步的时候进行 fsync；

AOF + fsync never. 磁盘压力会比较小；

RDB.

背景介绍

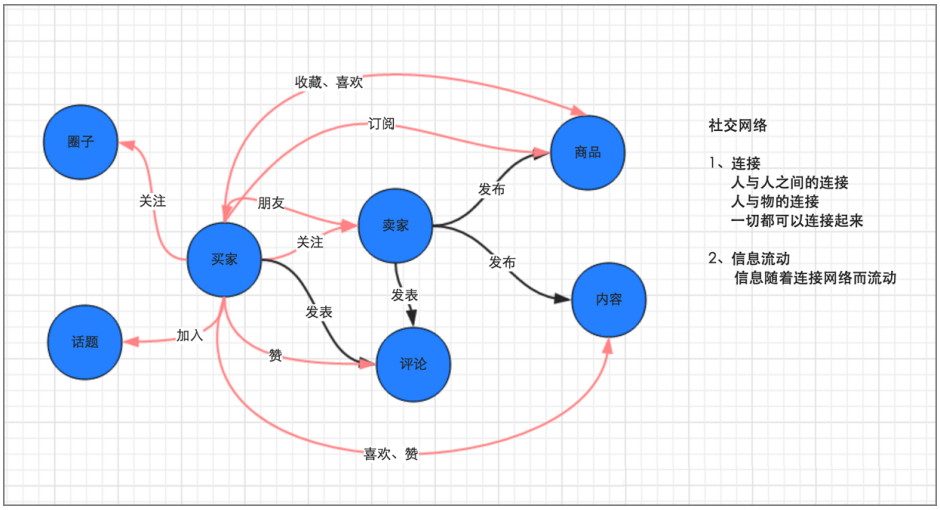
双十一如火如荼，云数据库 Redis 版也圆满完成了双十一的保障工作。目前云数据库 Redis 版提供了标准单副本、标准双副本和集群版本。

标准单副本和标准双副本 Redis 具有很高的兼容性，并且支持 Lua 脚本及地理位置计算。集群版本具有大容量、高性能的特性，能够突破 Redis 单线程的单机性能极限。

云数据库 Redis 版默认双机热备并提供了备份恢复支持，同时阿里云 Redis 源码团队持续对 Redis 进行优化升级，提供了强大的安全防护能力。本文将选取双十一的一些业务场景简化之后进行介绍，实际业务场景会比本文复杂。

微淘社区之亿级关系链存储

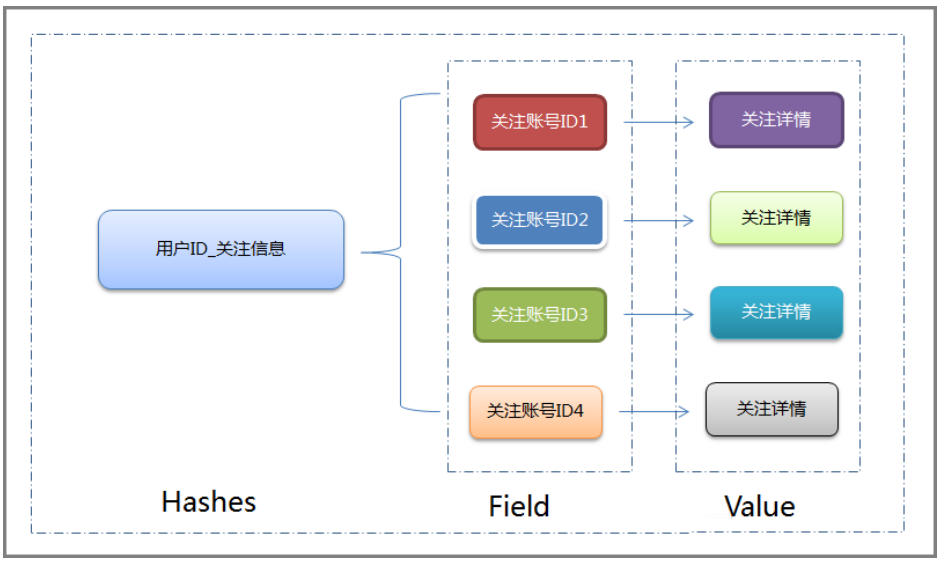
微淘社区承载了亿级淘宝用户的社交关系链，每个用户都有自己的关注列表，每个商家有自己的粉丝信息，整个微淘社区承载的关系链如下图所示。



如果选用传统的关系型数据库模型表达如上的关系信息，会使业务设计繁杂，并且不能获得良好的性能体验。微淘社区使用 Redis 集群缓存存储社区的关注链，简化了关注信息的存储，并保证了双十一业务丝滑一般的体验。微淘社区使用了 Hashes 存储用户之间的关注信息，存储结构如下，并提供了以下两种的查询接口：

用户 A 是否和用户 B 产生过关注关系

用户 A 的主动关系列表



天猫直播之评论商品游标分页

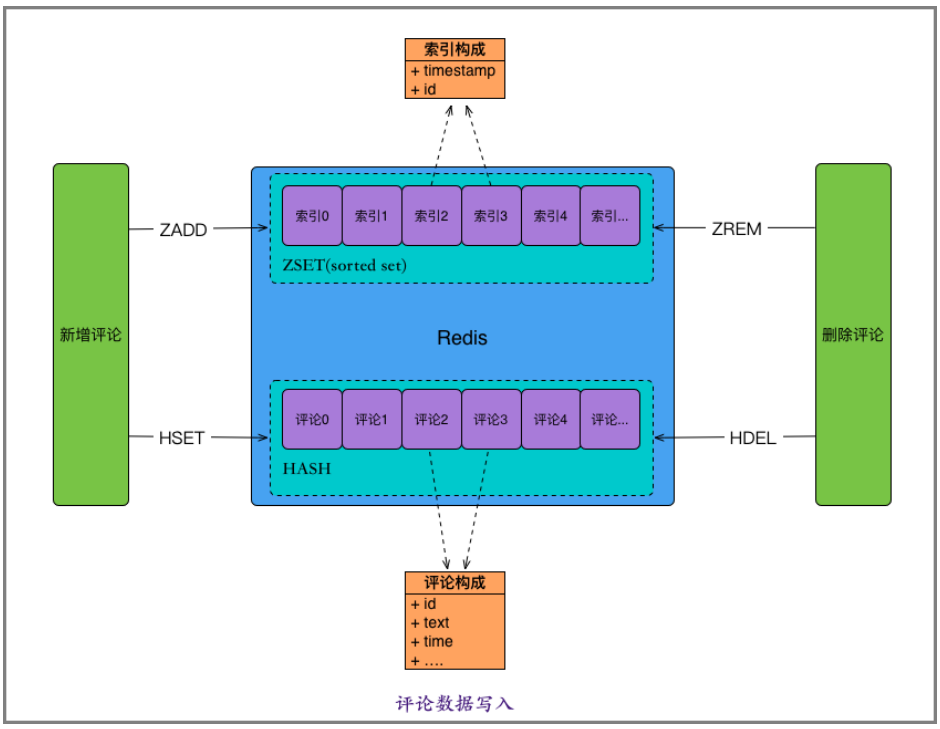
双十一用户在观看无线端直播的时候，需要对直播对应的评论进行刷新动作，主要有以下三种模式:

增量下拉：从指定位置向上获取指定个数（增量）的评论。

下拉刷新：获取最新的指定个数的评论。

增量上拉：从指定位置向下获取指定个数（增量）的评论。

无线直播系统使用 Redis 优化该场景的业务，保证了直播评论接口的成功率，并能够保证5万以上的 TPS 和毫秒级的 response time 请求。直播系统对于每个直播会写入两份数据，分别为索引和评论数据，索引数据为 SortedSet 的数据结构用于对评论的排序，而评论数据使用 Hashes 进行存储，在获取评论的时候通过索引拿到需要的索引 id 之后通过 Hashes 的读取来获得评论的列表。评论的写入过程如下：

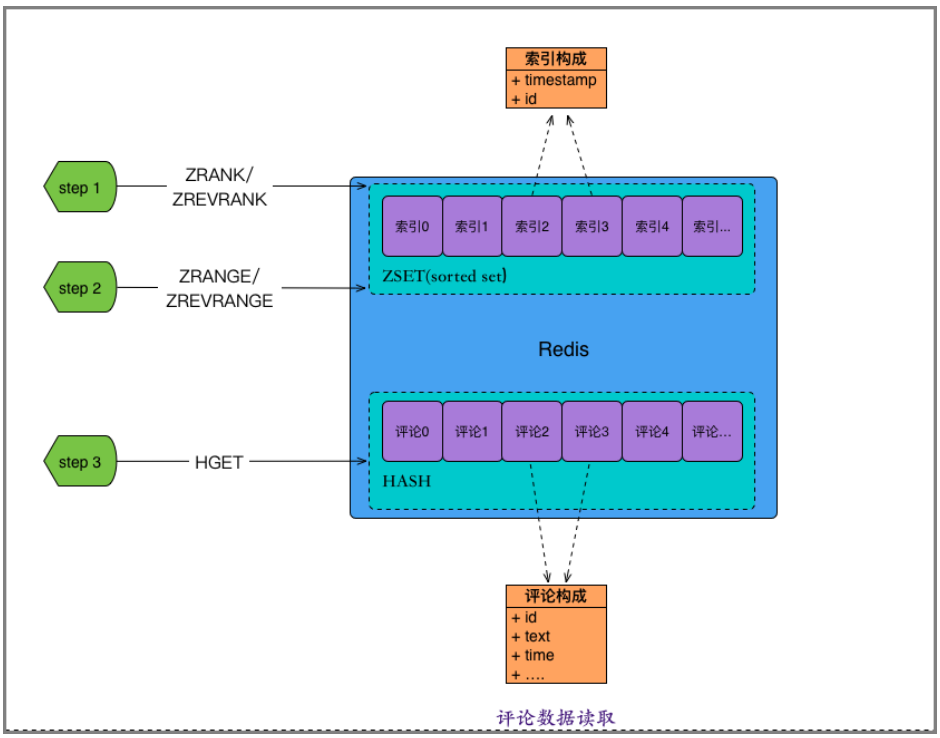


用户在刷新列表之后后台需要获取对应的评论信息，获取的流程如下：

获取当前索引位置

获取索引列表

获取评论数据



菜鸟单据履行中心之订单排序

双十一用户在产生一个交易订单之后会随之产生一个物流订单，需要经过菜鸟仓配系统处理。为了让仓配各个阶段能够更加智能的协同作业，决策系统会根据订单信息指定出对应的订单履行计划，包括什么时候下发仓、什么时候出库、什么时候配送揽收、什么时候送达等信息。单据履行中心根据履行计划，对每个阶段按照对应的时间去履行物流服务。由于仓、配的运力有限，对于有限的运力下，期望最早作业的单据是业务认为优先级最高的单据，所以订单在真正下发给仓或者配之前，需要按照优先级进行排序。

订单履行中心通过使用 Redis 来对所有的物流订单进行排序决定哪个订单是最高优先级的。