云数据库 Redis 版

最佳实践

最佳实践

游戏玩家积分排行榜

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
```

网上商城商品相关性分析

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，运行以上 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
```

消息发布与订阅

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。

管道传输

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
```

事务处理

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
```

通过数据集成将数据导入 Redis

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

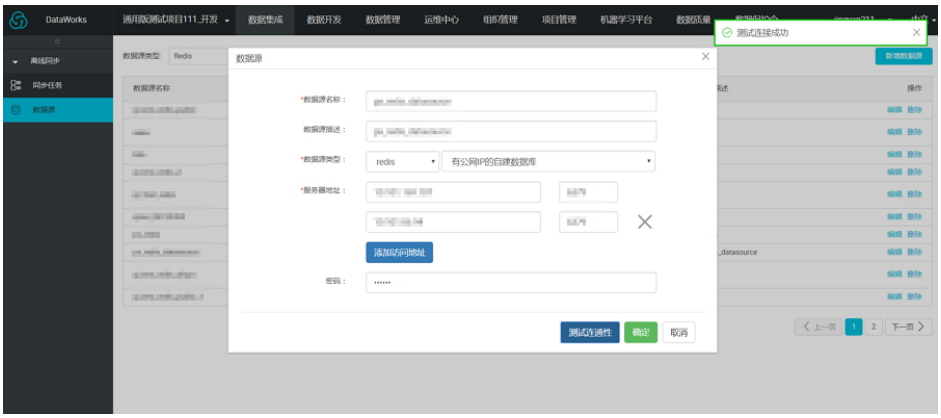
1. 以开发者身份进入**阿里云数加平台**，单击**项目列表**下对应**项目操作**栏中的**进入工作区**。

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**。

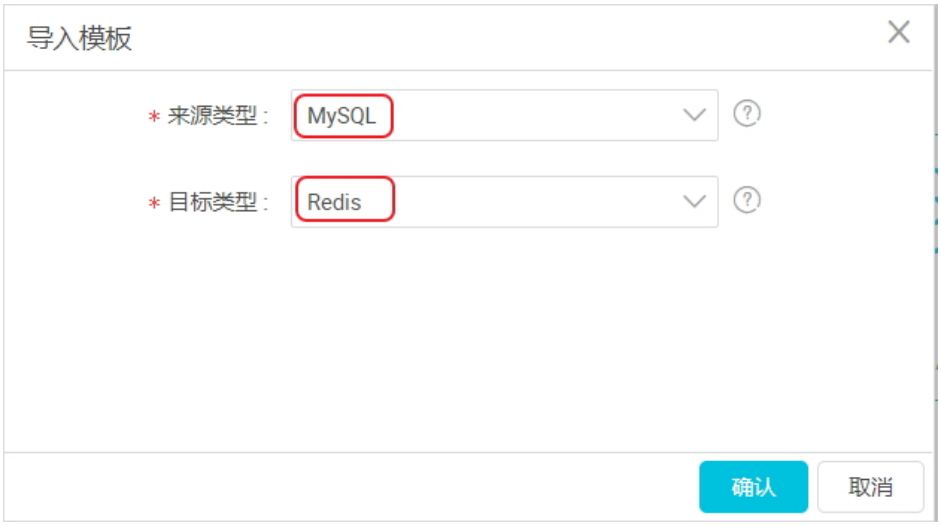
二、配置脚本模式的同步任务

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**。

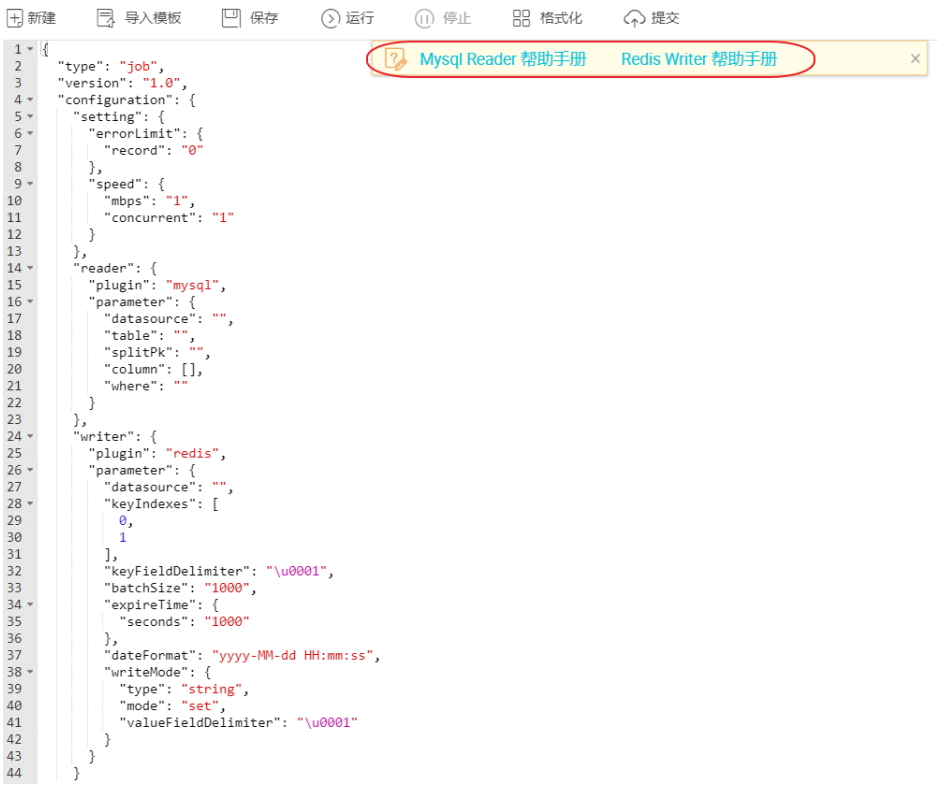
进入顶部菜单栏中的**数据集成**页面，选择**脚本模式**，如下图。

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：




```
{
  "type": "job",
  "version": "1.0",
  "configuration": {
    "setting": {
      "errorLimit": {
        "record": "0"
      },
      "speed": {
        "mbps": "1",
        "concurrent": "1"
      }
    },
    "reader": {
      "plugin": "mysql",
      "parameter": {
        "datasource": "",
        "table": "",
        "splitPk": "",
        "column": [],
        "where": ""
      }
    },
    "writer": {
      "plugin": "redis",
      "parameter": {
        "datasource": "",
        "keyIndexes": [
          0,
          1
        ],
        "keyFieldDelimiter": "\u0001",
        "batchSize": "1000",
        "expireTime": {
          "seconds": "1000"
        },
        "dateFormat": "yyyy-MM-dd HH:mm:ss",
        "writeMode": {
          "type": "string",
          "mode": "set",
          "valueFieldDelimiter": "\u0001"
        }
      }
    }
  }
```

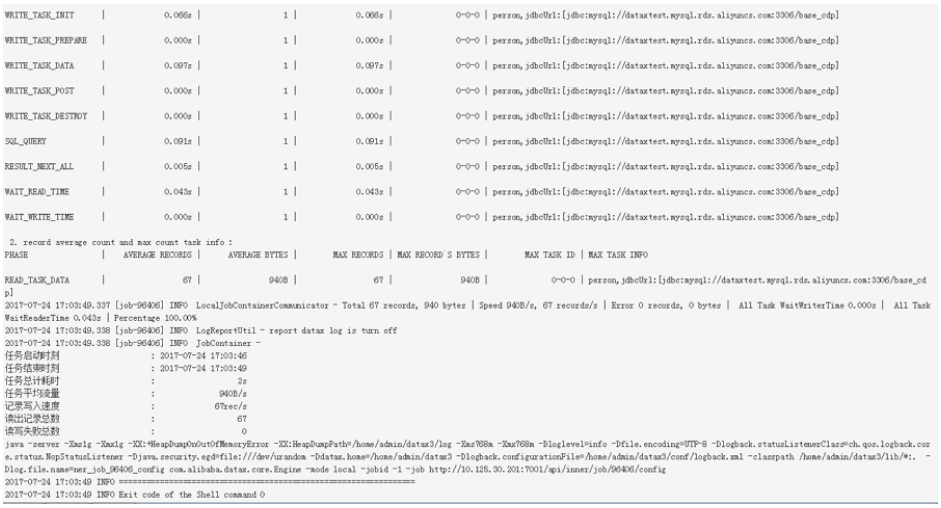
说明： RedisWriter 脚本案例如下：

```
{
"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：



```
WRITE_TASK_INIT     |          0.066s |               1 |          0.066s |            0-0-0 | person,jdbcUrl:[jdbc:mysql://datax test.mysql.rds.aliyuncs.com:3306/base_cdp]

WRITE_TASK_PREPARE  |          0.000s |               1 |          0.000s |            0-0-0 | person,jdbcUrl:[jdbc:mysql://datax test.mysql.rds.aliyuncs.com:3306/base_cdp]

WRITE_TASK_DATA     |          0.097s |               1 |          0.097s |            0-0-0 | person,jdbcUrl:[jdbc:mysql://datax test.mysql.rds.aliyuncs.com:3306/base_cdp]

WRITE_TASK_POST     |          0.000s |               1 |          0.000s |            0-0-0 | person,jdbcUrl:[jdbc:mysql://datax test.mysql.rds.aliyuncs.com:3306/base_cdp]

WRITE_TASK_DESTROY  |          0.000s |               1 |          0.000s |            0-0-0 | person,jdbcUrl:[jdbc:mysql://datax test.mysql.rds.aliyuncs.com:3306/base_cdp]

SQL_QUERY           |          0.091s |               1 |          0.091s |            0-0-0 | person,jdbcUrl:[jdbc:mysql://datax test.mysql.rds.aliyuncs.com:3306/base_cdp]

RESULT_NEXT_ALL     |          0.005s |               1 |          0.005s |            0-0-0 | person,jdbcUrl:[jdbc:mysql://datax test.mysql.rds.aliyuncs.com:3306/base_cdp]

WAIT_READ_TIME      |          0.043s |               1 |          0.043s |            0-0-0 | person,jdbcUrl:[jdbc:mysql://datax test.mysql.rds.aliyuncs.com:3306/base_cdp]

WAIT_WRITE_TIME     |          0.000s |               1 |          0.000s |            0-0-0 | person,jdbcUrl:[jdbc:mysql://datax test.mysql.rds.aliyuncs.com:3306/base_cdp]

 2. record average count and max count task info :
PHASE               |   AVERAGE RECORDS |      AVERAGE BYTES |       MAX RECORDS | MAX RECORD'S BYTES |       MAX TASK ID | MAX TASK INFO

READ_TASK_DATA      |                67 |              940B |                67 |              940B |            0-0-0 | person,jdbcUrl:[jdbc:mysql://datax test.mysql.rds.aliyuncs.com:3306/base_cd
p]
2017-07-24 17:03:49.337 [job-96406] INFO  LocalJobContainerCommunicator - Total 67 records, 940 bytes | Speed 940B/s, 67 records/s | Error 0 records, 0 bytes |  All Task WaitWriterTime 0.000s |  All Task
WaitReaderTime 0.043s | Percentage 100.00%
2017-07-24 17:03:49.338 [job-96406] INFO  LogReportUtil - report datax log is turn off
2017-07-24 17:03:49.338 [job-96406] INFO  JobContainer -
任务启动时刻                    : 2017-07-24 17:03:46
任务结束时刻                    : 2017-07-24 17:03:49
任务总计耗时                    :                  2s
任务平均流量                    :              940B/s
记录写入速度                    :             67rec/s
读出记录总数                    :                  67
读写失败总数                    :                   0
java -server -Xms1g -Xmx1g -XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=/home/admin/datax3/log -Xms768m -Xmx768m -Dloglevel=info -Dfile.encoding=UTF-8 -Dlogback.statusListenerClass=ch.qos.logback.cor
e.status.NopStatusListener -Djava.security.egd=file:///dev/urandom -Ddatax.home=/home/admin/datax3 -Dlogback.configurationFile=/home/admin/datax3/conf/logback.xml -classpath /home/admin/datax3/lib/*:.  -
Dlog.file.name=nez_job_96406_config com.alibaba.datax.core.Engine -mode local -jobid -1 -job http://10.125.30.201:7001/api/inner/job/96406/config
2017-07-24 17:03:49 INFO =================================================================
2017-07-24 17:03:49 INFO Exit code of the Shell command 0
```

- RedisWriter 参数说明请参考 RedisWriter 配置。

热点 Key 问题的发现与解决

热点问题概述

产生原因

热点问题产生的原因大致有以下两种：

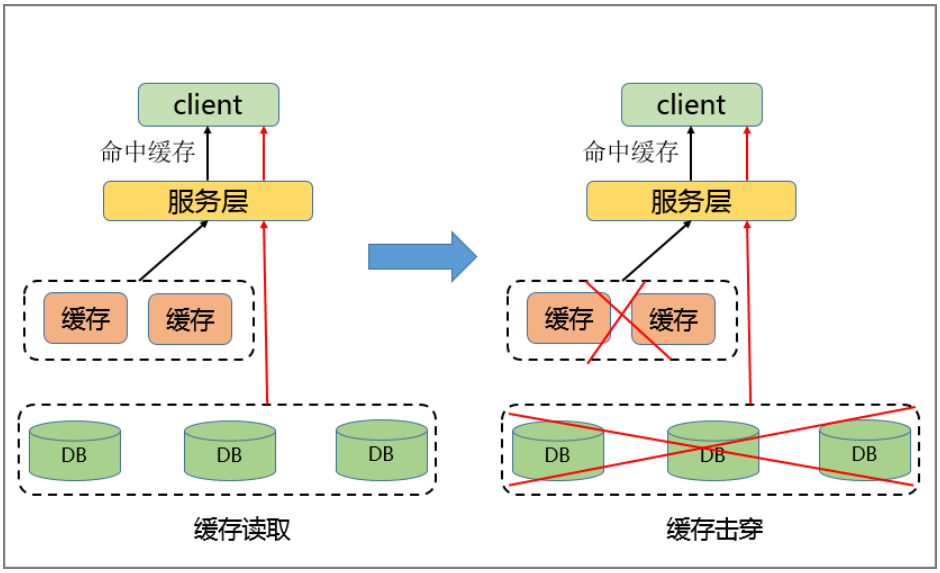
用户消费的数据远大于生产的数据（热卖商品、热点新闻、热点评论、明星直播）

在日常工作生活中一些突发的的事件，例如：双十一期间某些热门商品的降价促销，当这其中的某一件商品被数万次点击浏览或者购买时，会形成一个较大的需求量，这种情况下就会造成热点问题。同理，被大量刊发、浏览的热点新闻、热点评论、明星直播等，这些典型的读多写少的场景也会产生热点问题。

请求分片集中，超过单 Server 的性能极限

在服务端读数据进行访问时，往往会对数据进行分片切分，此过程中会在某一主机 Server 上对相应的 Key 进行访问，当访问超过 Server 极限时，就会导致热点 Key 问题的产生。

热点问题的危害



流量集中，达到物理网卡上限

请求过多，缓存分片服务被打垮

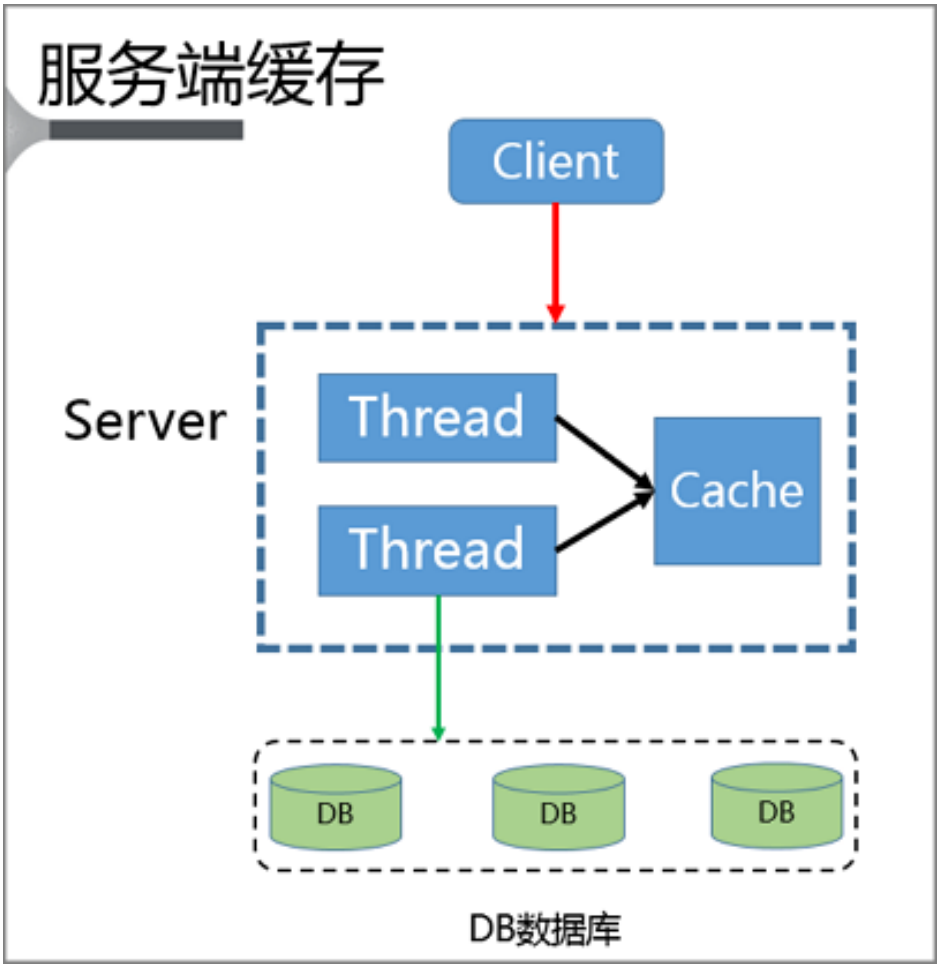
DB 击穿，引起业务雪崩

如前文讲到的，当某一热点 Key 的请求在某一主机上超过该主机网卡上限时，由于流量的过度集中，会导致服务器中其它服务无法进行。如果热点过于集中，热点 Key 的缓存过多，超过目前的缓存容量时，就会导致缓存分片服务被打垮现象的产生。当缓存服务崩溃后，此时再有请求产生，会缓存到后台 DB 上，由于DB 本身性能较弱，在面临大请求时很容易发生请求穿透现象，会进一步导致雪崩现象，严重影响设备的性能。

常见解决方案

通常的解决方案主要集中在对客户端和 Server 端进行相应的改造。

服务端缓存方案



首先 Client 会将请求发送至 Server 上，而 Server 又是一个多线程的服务，本地就具有一个基于 Cache LRU 策略的缓存空间。当 Server 本身就拥堵时，Server 不会将请求进一步发送给 DB 而是直接返回，只有当 Server 本身畅通时才会将 Client 请求发送至 DB，并且将该数据重新写入到缓存中。此时就完成了缓存的访问跟重建。

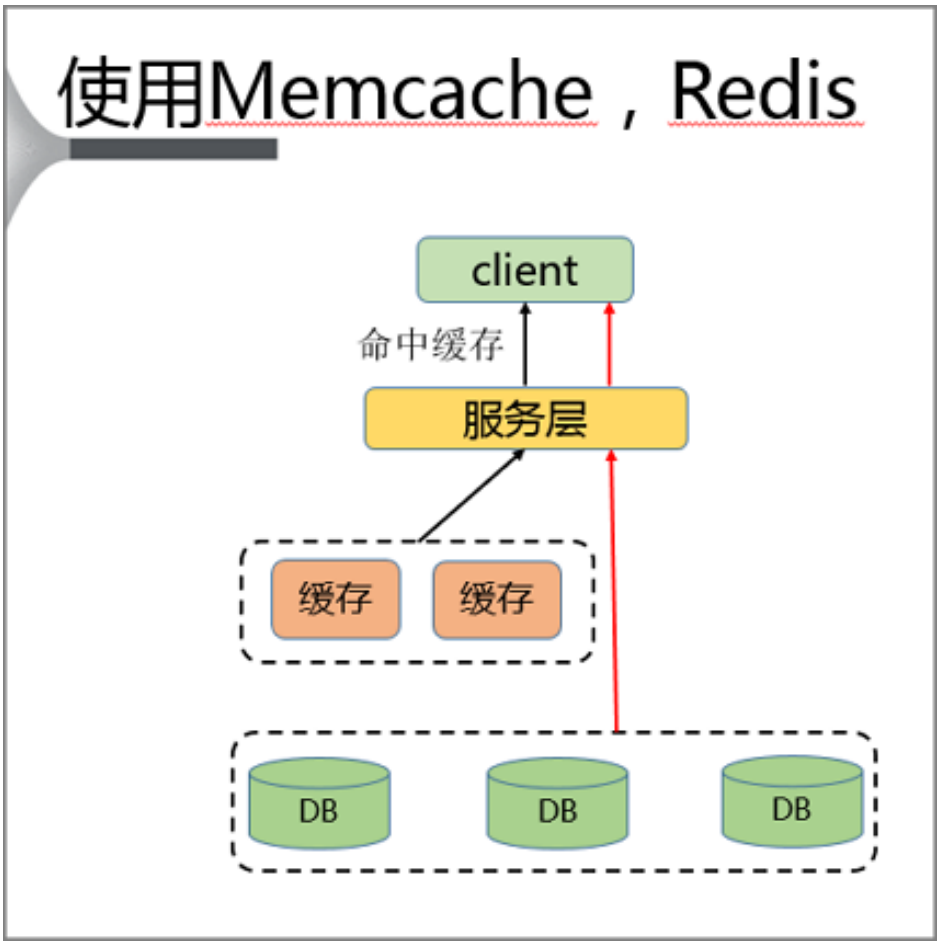
但该方案也存在以下问题：

缓存失效，多线程构建缓存问题

缓存丢失，缓存构建问题

脏读问题

使用 Memcache、Redis 方案



该方案通过在客户端单独部署缓存的方式来解决热点 Key 问题。使用过程中 Client 首先访问服务层，再对同一主机上的缓存层进行访问。该种解决方案具有就近访问、速度快、没有带宽限制的优点，但是同时也存在以下问题。

内存资源浪费

脏读问题

使用本地缓存方案

使用本地缓存则存在以下问题：

需要提前获知热点

缓存容量有限

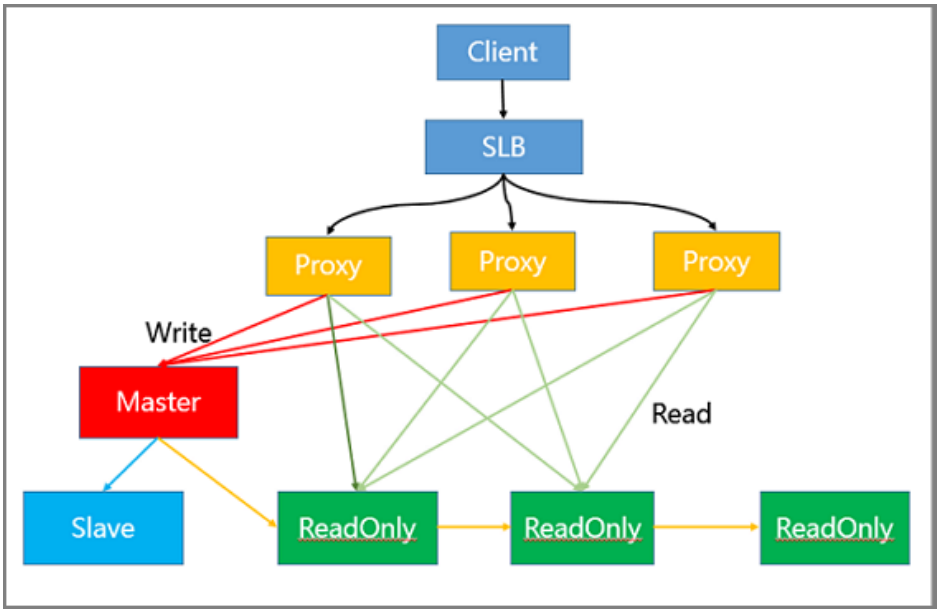
不一致性时间增长

热点 Key 遗漏

传统的热点解决方案都存在各种各样的问题，那么究竟该如何解决热点问题呢？

阿里云数据库解热点之道

读写分离方案解决热读



架构中各节点的作用如下：

SLB 层做负载均衡

Proxy 层做读写分离自动路由

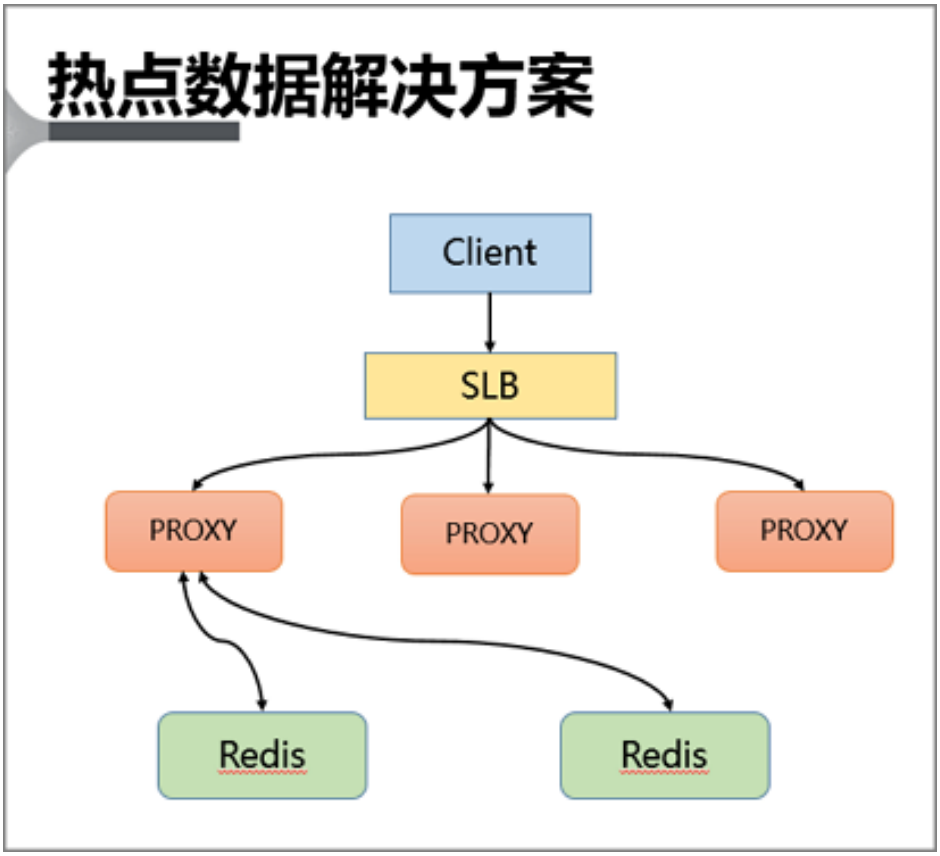
Master 负责写请求

ReadOnly 节点负责读请求

Slave 节点和 Master 节点做高可用

实际过程中 Client 将请求传到 SLB，SLB 又将其分发至多个 Proxy 内，通过 Proxy 对请求的识别，将其进行分类发送。例如，将同为 Write 的请求发送到 Master 模块内，而将 Read 的请求发送至 ReadOnly 模块。而模块中的只读节点可以进一步扩充，从而有效解决热点读的问题。读写分离同时具有可以灵活扩容读热点能力、可以存储大量热点Key、对客户端友好等优点。

热点数据解决方案



该方案通过主动发现热点并对其进行存储来解决热点 Key 的问题。首先 Client 也会访问 SLB，并且通过 SLB 将各种请求分发至 Proxy 中，Proxy 会按照基于路由的方式将请求转发至后端的 Redis 中。

在热点 key 的解决上是采用在服务端增加缓存的方式进行。具体来说就是在 Proxy 上增加本地缓存，本地缓存采用 LRU 算法来缓存热点数据，后端 db 节点增加热点数据计算模块来返回热点数据。

Proxy 架构的主要有以下优点：

Proxy 本地缓存热点，读能力可水平扩展

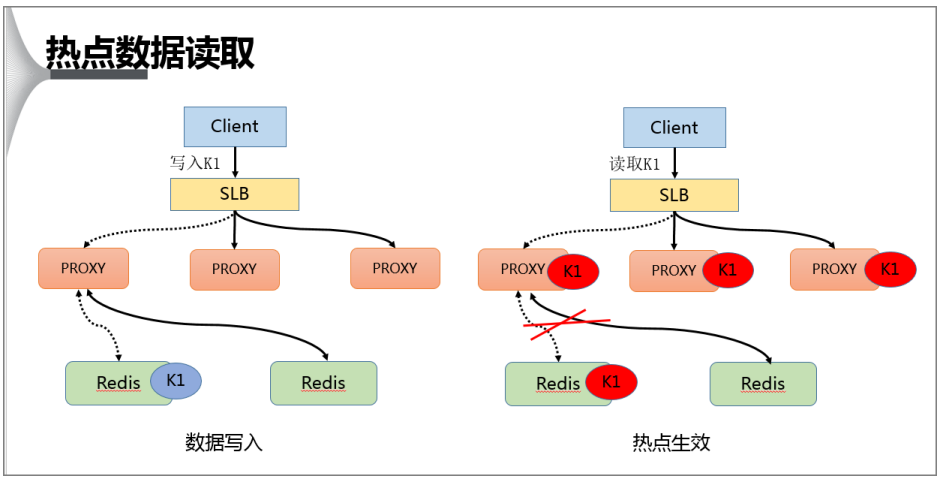
DB 节点定时计算热点数据集合

DB 反馈 Proxy 热点数据

对客户端完全透明，不需做任何兼容

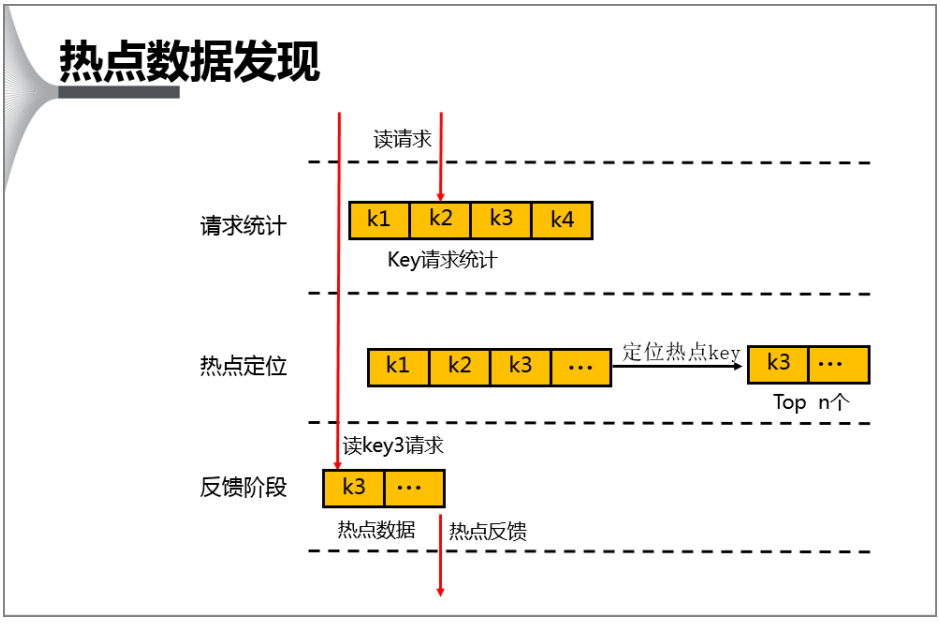
热点 key 处理

热点数据的读取



在热点 Key 的处理上主要分为写入跟读取两种形式，在数据写入过程当 SLB 收到数据 K1 并将其通过某一个 Proxy 写入一个 Redis，完成数据的写入。假若经过后端热点模块计算发现 K1 成为热点 key 后， Proxy 会将该热点进行缓存，当下次客户端再进行访问 K1 时，可以不经 Redis。最后由于 proxy 是可以水平扩充的，因此可以任意增强热点数据的访问能力。

热点数据的发现



对于 db 上热点数据的发现，首先会在一个周期内对 Key 进行请求统计，在达到请求量级后会对热点 Key 进行热点定位，并将所有的热点 Key 放入一个小的 LRU 链表内，在通过 Proxy 请求进行访问时，若 Redis 发现待访点是一个热点，就会进入一个反馈阶段，同时对该数据进行标记。

DB 计算热点时，主要运用的方法和优势有：

基于统计阀值的热点统计

基于统计周期的热点统计

基于版本号实现的无需重置初值统计方法

DB 计算同时具有对性能影响极其微小、内存占用极其微小等优点

两种方案对比

通过上述对比分析可以看出，阿里云在解决热点 Key 上较传统方法相比都有较大的提高，无论是基于读写分离方案还是热点数据解决方案，在实际处理环境中都可以做灵活的水平能力扩充、都对客户端透明、都有一定的数据不一致性。此外读写分离模式可以存储更大量的热点数据，而基于 Proxy 的模式有成本上的优势。

解密 Redis 助力双十一背后的技术

背景介绍

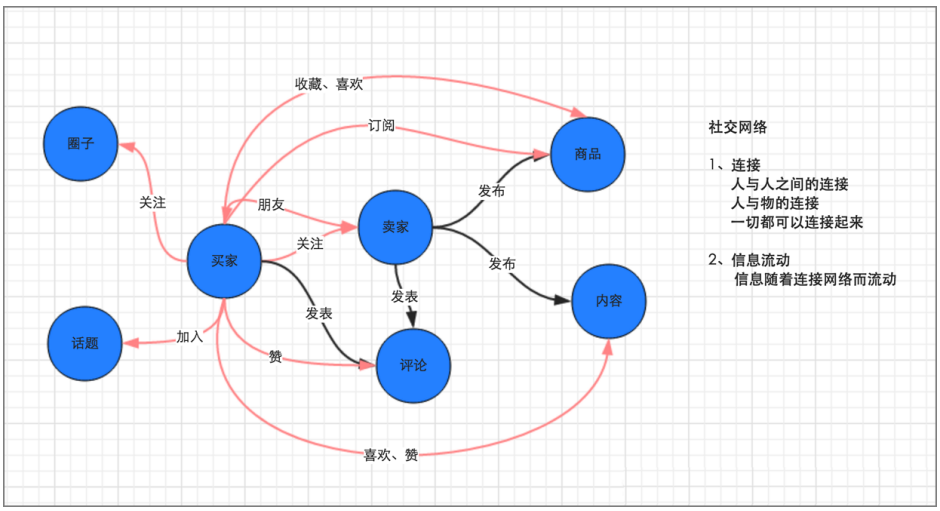
双十一如火如荼，云数据库 Redis 版也圆满完成了双十一的保障工作。目前云数据库 Redis 版提供了标准单副本、标准双副本和集群版本。

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

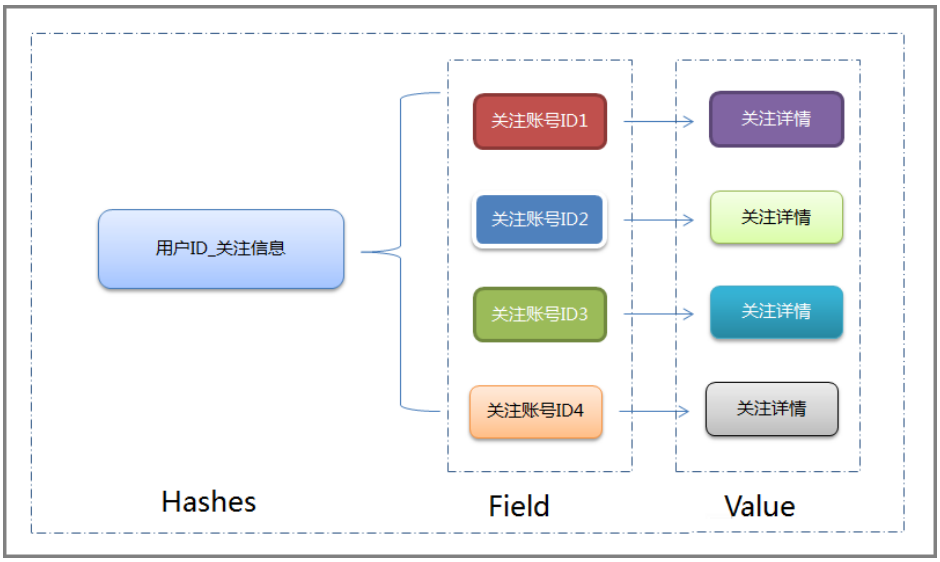
微淘社区承载了亿级淘宝用户的社交关系链，每个用户都有自己的关注列表，每个商家有自己的粉丝信息，整个微淘社区承载的关系链如下图所示。



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

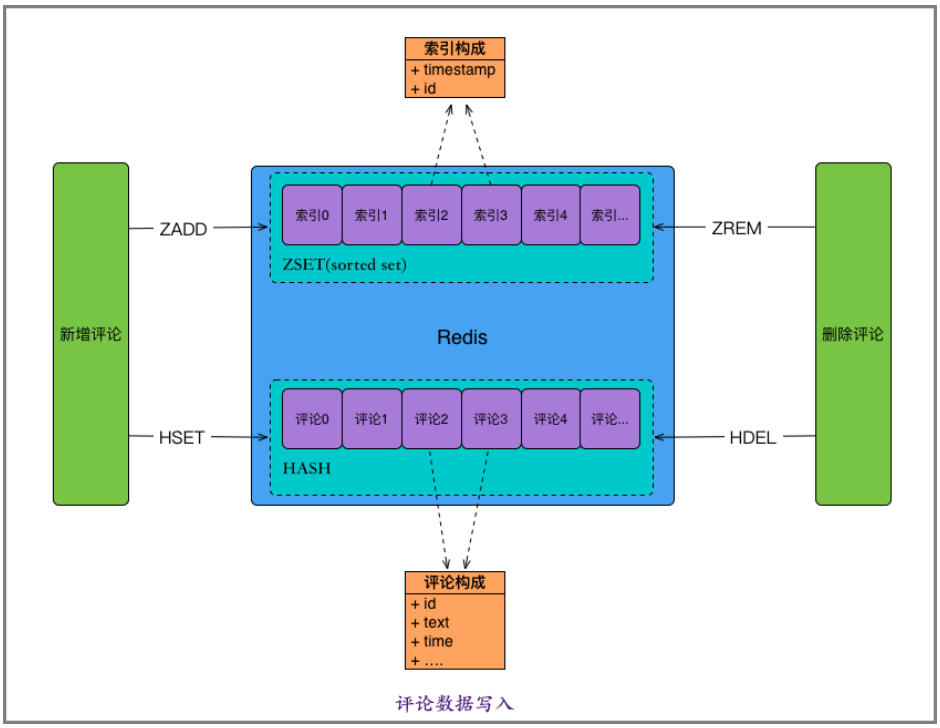
双十一用户在观看无线端直播的时候，需要对直播对应的评论进行刷新动作，主要有以下三种模式:

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：

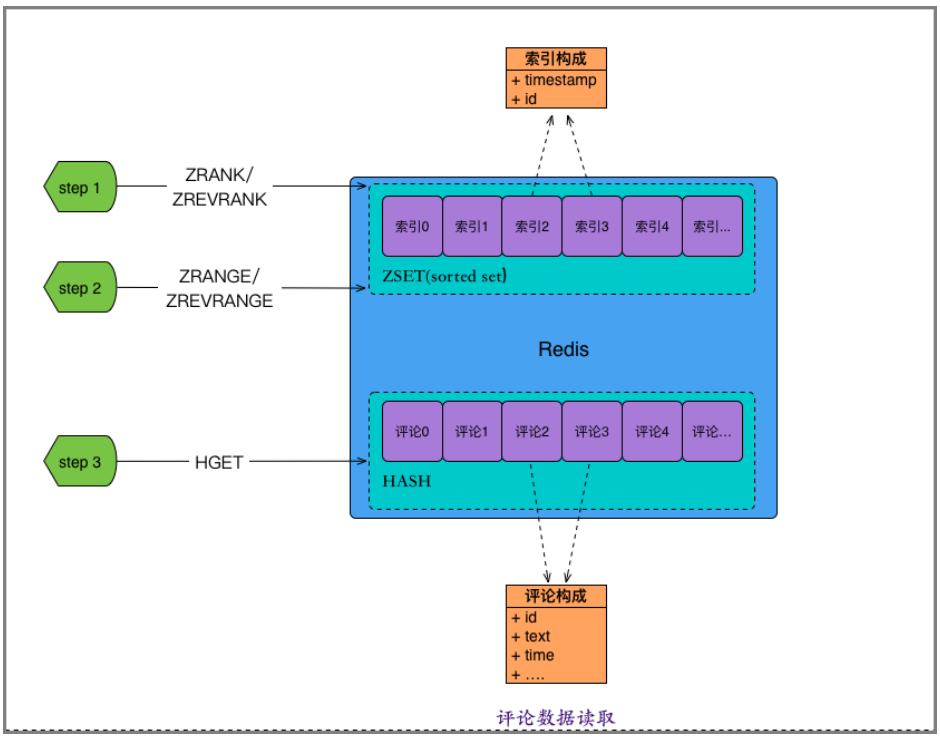


用户在刷新列表之后后台需要获取对应的评论信息，获取的流程如下：

获取当前索引位置

获取索引列表

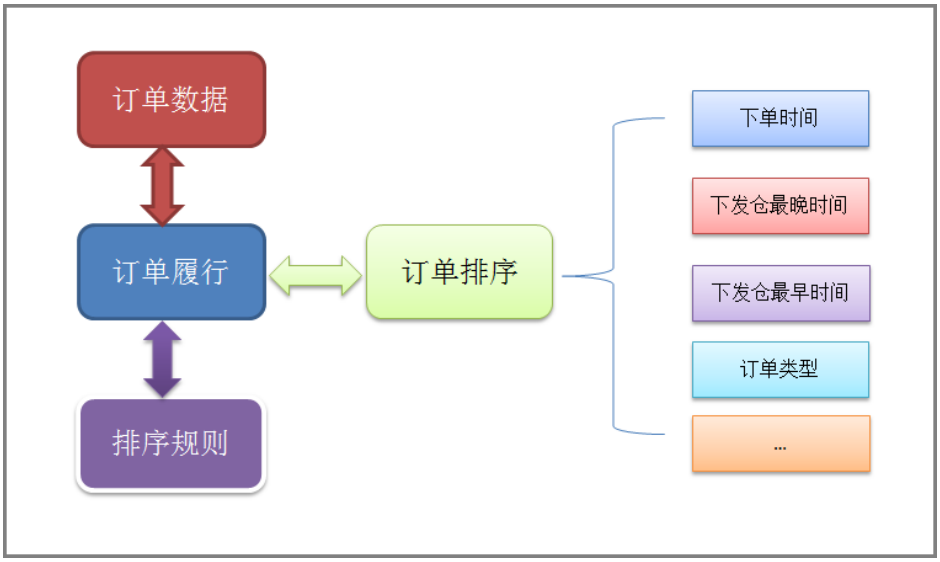
获取评论数据



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。



使用 Redis 搭建电商秒杀系统

背景

秒杀活动是绝大部分电商选择的低价促销、推广品牌的方式。不仅可以给平台带来用户量，还可以提高平台知名度。一个好的秒杀系统，可以提高平台系统的稳定性和公平性，获得更好的用户体验，提升平台的口碑，从而提升秒杀活动的最大价值。

本文讨论云数据库 Redis 版缓存设计高并发的秒杀系统。

秒杀的特征

秒杀活动对稀缺或者特价的商品进行定时定量售卖，吸引成大量的消费者进行抢购，但又只有少部分消费者可以下单成功。因此，秒杀活动将在较短时间内产生比平时大数十倍，上百倍的页面访问流量和下单请求流量。

秒杀活动可以分为3个阶段：

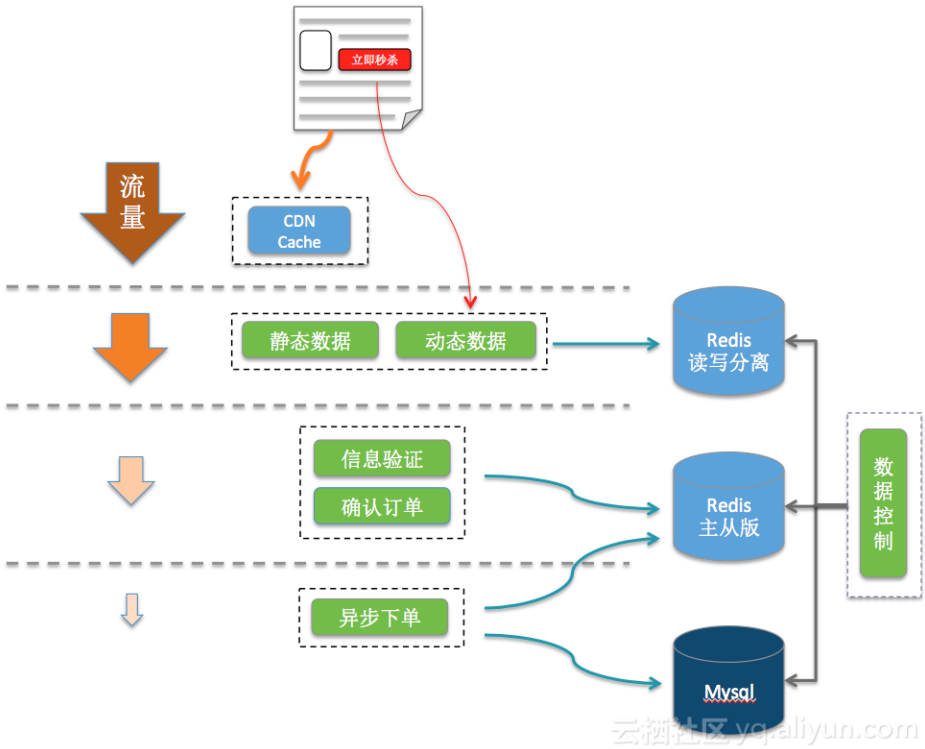
秒杀前：用户不断刷新商品详情页，页面请求达到瞬时峰值。

秒杀开始：用户点击秒杀按钮，下单请求达到瞬时峰值。

秒杀后：一部分成功下单的用户不断刷新订单或者产生退单操作，大部分用户继续刷新商品详情页等待退单机会。

消费者提交订单，一般做法是利用数据库的行级锁，只有抢到锁的请求可以进行库存查询和下单操作。但是在高并发的情况下，数据库无法承担如此大的请求，往往会使整个服务 blocked，在消费者看来就是服务器宕机。

秒杀系统



秒杀系统的流量虽然很高，但是实际有效流量是十分有限的。利用系统的层次结构，在每个阶段提前校验，拦截无效流量，可以减少大量无效的流量涌入数据库。

利用浏览器缓存和 CDN 抗压静态页面流量

秒杀前，用户不断刷新商品详情页，造成大量的页面请求。所以，我们需要把秒杀商品详情页与普通的商品详情页分开。对于秒杀商品详情页尽量将能静态化的元素静态化处理，除了秒杀按钮需要服务端进行动态判断，其他的静态数据可以缓存在浏览器和 CDN 上。这样，秒杀前刷新页面导致的流量进入服务端的流量只有很小的一部分。

利用读写分离 Redis 缓存拦截流量

CDN 是第一级流量拦截，第二级流量拦截我们使用支持读写分离的 Redis。在这一阶段我们主要读取数据，读写分离 Redis 能支持高大60万以上 qps 的，完全可以支持需求。

首先通过数据控制模块，提前将秒杀商品缓存到读写分离 Redis，并设置秒杀开始标记如下：

```
"goodsId_count": 100 //总数
"goodsId_start": 0 //开始标记
"goodsId_access": 0 //接受下单数
```

秒杀开始前，服务集群读取 goodsId_Start 为 0，直接返回未开始。

数据控制模块将 goodsId_start 改为1，标志秒杀开始。

服务集群缓存开始标记位并开始接受请求，并记录到 redis 中 goodsId_access，商品剩余数量为 (goodsId_count - goodsId_access)。

当接受下单数达到 goodsId_count 后，继续拦截所有请求，商品剩余数量为 0。

可以看出，最后成功参与下单的请求只有少部分可以被接受。在高并发的情况下，允许稍微多的流量进入。因此可以控制接受下单数的比例。

利用主从版 Redis 缓存加速库存扣量

成功参与下单后，进入下层服务，开始进行订单信息校验，库存扣量。为了避免直接访问数据库，我们使用主从版 Redis 来进行库存扣量，主从版 Redis 提供10万级别的 QPS。使用 Redis 来优化库存查询，提前拦截秒杀失败的请求，将大大提高系统的整体吞吐量。

通过数据控制模块提前将库存存入 Redis，将每个秒杀商品在 Redis 中用一个 hash 结构表示。

```
"goodsId" : {
"Total": 100
"Booked": 100
}
```

扣量时，服务器通过请求 Redis 获取下单资格，通过以下 lua 脚本实现，由于 Redis 是单线程模型，lua 可以保证多个命令的原子性。

```
local n = tonumber(ARGV[1])
if not n or n == 0 then
return 0
end
local vals = redis.call("HMGET", KEYS[1], "Total", "Booked");
local total = tonumber(vals[1])
local blocked = tonumber(vals[2])
if not total or not blocked then
return 0
end
if blocked + n <= total then
redis.call("HINCRBY", KEYS[1], "Booked", n)
return n;
end
return 0
```

先使用SCRIPT LOAD将 lua 脚本提前缓存在 Redis，然后调用EVALSHA调用脚本，比直接调用EVAL节省网络带宽：

```
redis 127.0.0.1:6379>SCRIPT LOAD "lua code"
"438dd755f3fe0d32771753eb57f075b18fed7716"

redis 127.0.0.1:6379>EVAL 438dd755f3fe0d32771753eb57f075b18fed7716 1 goodsId 1
```

秒杀服务通过判断 Redis 是否返回抢购个数 n，即可知道此次请求是否扣量成功。

使用主从版 Redis 实现简单的消息队列异步下单入库

扣量完成后，需要进行订单入库。如果商品数量较少的时候，直接操作数据库即可。如果秒杀的商品是1万，甚至10万级别，那数据库锁冲突将带来很大的性能瓶颈。因此，利用消息队列组件，当秒杀服务将订单信息写入消息队列后，即可认为下单完成，避免直接操作数据库。

消息队列组件依然可以使用 Redis 实现，在 R2 中用 list 数据结构表示。

```
orderList {
[0] = {订单内容}
[1] = {订单内容}
[2] = {订单内容}
...
}
```

将订单内容写入 Redis:

```
LPUSH orderList {订单内容}
```

异步下单模块从 Redis 中顺序获取订单信息，并将订单写入数据库。

```
BRPOP orderList 0
```

通过使用 Redis 作为消息队列，异步处理订单入库，有效的提高了用户的下单完成速度。

数据控制模块管理秒杀数据同步

最开始，利用读写分离 Redis 进行流量限制，只让部分流量进入下单。对于下单检验失败和退单等情况，需要让更多的流量进来。因此，数据控制模块需要定时将数据库中的数据进行一定的计算，同步到主从版 Redis，同时再同步到读写分离的 Redis，让更多的流量进来。

Redis 读写分离技术解析

背景

云数据库 Redis 版不管主从版还是集群规格，slave 作为备库不对外提供服务，只有在发生 HA 的时候，slave

提升为 master 后才承担读写流量。这种架构读写请求都在 master 上完成，一致性较高，但性能受到 master 数量的限制。经常有用户数据较少，但因为流量或者并发太高而不得不升级到更大的集群规格。

为满足读多写少的业务场景，最大化节约用户成本，云数据库 Redis 版推出了读写分离规格，为用户提供透明、高可用、高性能、高灵活的读写分离服务。

架构

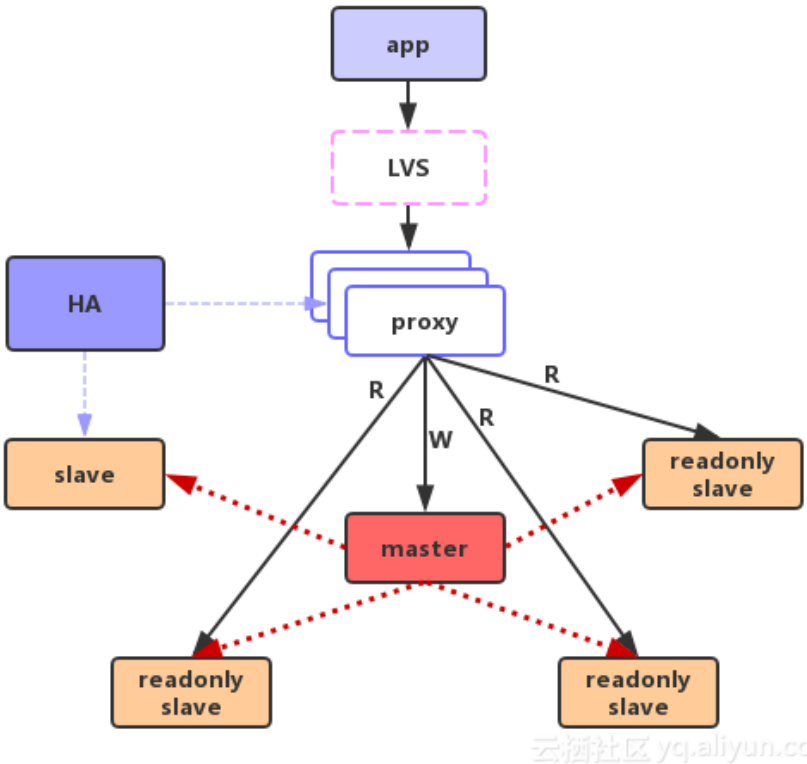
Redis 集群模式有 redis-proxy、master、slave、HA 等几个角色。在读写分离实例中，新增 readonly slave 角色来承担读流量，slave 作为热备不提供服务，架构上保持对现有集群规格的兼容性。redis-proxy 按权重将读写请求转发到 master 或者某个 readonly slave 上；HA 负责监控 DB 节点的健康状态，异常时发起主从切换或重搭 readonly slave，并更新路由。

一般来说，根据 master 和 readonly slave 的数据同步方式，可以分为两种架构：星型复制和链式复制。

星型复制

星型复制就是将所有的 readonly slave 直接和 master 保持同步，每个 readonly slave 之间相互独立，任何一个节点异常不影响到其他节点，同时因为复制链比较短，readonly slave 上的复制延迟比较小。

Redis 是单进程单线程模型，主从之间的数据复制也在主线程中处理，readonly slave 数量越多，数据同步对 master 的 CPU 消耗就越严重，集群的写入性能会随着 readonly slave 的增加而降低。此外，星型架构会让 master 的出口带宽随着 readonly slave 的增加而成倍增长。master 上较高的 CPU 和网络负载会抵消掉星型复制延迟较低的优势，因此，星型复制架构会带来比较严重的扩展问题，整个集群的性能会受限于 master。

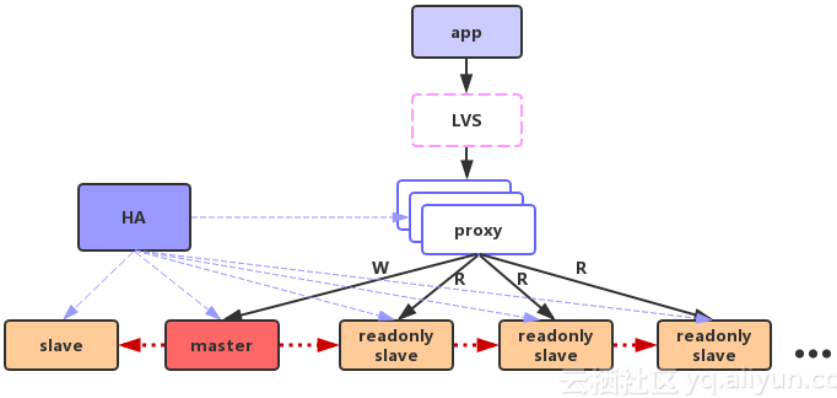


链式复制

链式复制将所有的 readonly slave 组织成一个复制链，如下图所示，master 只需要将数据同步给 slave 和复制链上的第一个 readonly slave。

链式复制解决了星型复制的扩展问题，理论上可以无限增加 readonly slave 的数量，随着节点的增加整个集群的性能也可以基本上呈线性增长。

链式复制的架构下，复制链越长，复制链末端的 readonly slave 和 master 之间的同步延迟就越大，考虑到读写分离主要使用在对一致性要求不高的场景下，这个缺点一般可以接受。但是如果复制链中的某个节点异常，会导致下游的所有节点数据都会大幅滞后。更加严重的是这可能带来全量同步，并且全量同步将一直传递到复制链的末端，这会对服务带来一定的影响。为了解决这个问题，读写分离的 redis 都使用阿里云优化后的 binlog 复制版本，最大程度的降低全量同步的概率。



结合上述的讨论和比较，redis 读写分离选择链式复制的架构。

Redis 读写分离优势

透明兼容

读写分离和普通集群规格一样，都使用了 redis-proxy 做请求转发，多 shard 时部分命令使用存在一定的限制，但从主从升级单分片读写分离，或者从集群升级到多分片的读写分离集群可以做到完全兼容。

用户和 redis-proxy 建立连接，redis-proxy 会识别出客户端连接发送过来的请求是读还是写，然后按照权重作负载均衡，将请求转发到后端不同的 DB 节点中，写请求转发给 master，读操作转发给 readonly slave（master 默认也提供读，可以通过权重控制）。

用户只需要购买读写分离规格的实例，直接使用任何客户端即可直接使用，业务不用做任何修改就可以开始享受读写分离服务带来的巨大性能提升，接入成本几乎为0。

高可用

高可用模块（HA）监控所有 DB 节点的健康状态，为整个实例的可用性保驾护航。master 宕机时自动切换到新主。如果某个 readonly slave 宕机，HA 也能及时感知，然后重搭一个新的 readonly slave，下线宕机节点。

除 HA 之外，redis-proxy 也能实时感知每个 readonly slave 的状态。在某个 readonly slave 异常期间，redis-proxy 会自动降低这个节点的权重，如果发现某个 readonly slave 连续失败超过一定次数以后，会暂时屏蔽异常节点，直到异常消失以后才会恢复其正常权重。

redis-proxy 和 HA 一起做到尽量减少业务对后端异常的感知，提高服务可用性。

高性能

对于读多写少的业务场景，直接使用集群版本往往不是最合适的方案，现在读写分离提供了更多的选择，业务可以根据场景选择最适合的规格，充分利用每一个 readonly slave 的资源。

目前单 shard 对外售卖 1 master + 1/3/5 readonly slave 多种规格（如果有更大的需求可以提工单反馈），提供 60万 QPS 和 192 MByte/s 的服务能力，在完全兼容所有命令的情况下突破单机的资源限制。后续将去掉规格限制，让用户根据业务流量随时自由的增加或减少 readonly slave 数量。

规格	QPS	带宽
1 master	8-10万读写	10-48 MB
1 master + 1 readonly_slave	10万写 + 10万读	20-64 MB
1 master + 3 readonly_slave	10万写 + 30万读	40-128 MB
1 master + 5 readonly_slave	10万写 + 50万读	60-192 MB

后续

redis 主从异步复制，从 readonly slave 中可能读到旧的数据，使用读写分离需要业务可以容忍一定程度的数据不一致，后续将会给客户更灵活的配置和更大的自由，比如配置可以容忍的最大延迟时间。