阿里云
云数据库 Redis 版

最佳实践

文档版本：20190305

法律声明

阿里云提醒您在阅读或使用本文档之前仔细阅读、充分理解本法律声明各条款的内容。如果您阅读或使用本文档，您的阅读或使用行为将被视为对本声明全部内容的认可。

1. 您应当通过阿里云网站或阿里云提供的其他授权通道下载、获取本文档，且仅能用于自身的合法合规的业务活动。本文档的内容视为阿里云的保密信息，您应当严格遵守保密义务；未经阿里云事先书面同意，您不得向任何第三方披露本手册内容或提供给任何第三方使用。
2. 未经阿里云事先书面许可，任何单位、公司或个人不得擅自摘抄、翻译、复制本文档内容的部分或全部，不得以任何方式或途径进行传播和宣传。
3. 由于产品版本升级、调整或其他原因，本文档内容有可能变更。阿里云保留在没有任何通知或者提示下对本文档的内容进行修改的权利，并在阿里云授权通道中不时发布更新后的用户文档。您应当实时关注用户文档的版本变更并通过阿里云授权渠道下载、获取最新版的用户文档。
4. 本文档仅作为用户使用阿里云产品及服务的参考性指引，阿里云以产品及服务的”现状“、“有缺陷”和“当前功能”的状态提供本文档。阿里云在现有技术的基础上尽最大努力提供相应的介绍及操作指引，但阿里云在此明确声明对本文档内容的准确性、完整性、适用性、可靠性等不作任何明示或暗示的保证。任何单位、公司或个人因为下载、使用或信赖本文档而发生任何差错或经济损失的，阿里云不承担任何法律责任。在任何情况下，阿里云均不对任何间接性、后果性、惩戒性、偶然性、特殊性或刑罚性的损害，包括用户使用或信赖本文档而遭受的利润损失，承担责任（即使阿里云已被告知该等损失的可能性）。
5. 阿里云网站上所有内容，包括但不限于著作、产品、图片、档案、资讯、资料、网站架构、网站画面的安排、网页设计，均由阿里云和/或其关联公司依法拥有其知识产权，包括但不限于商标权、专利权、著作权、商业秘密等。非经阿里云和/或其关联公司书面同意，任何人不得擅自使用、修改、复制、公开传播、改变、散布、发行或公开发表阿里云网站、产品程序或内容。此外，未经阿里云事先书面同意，任何人不得为了任何营销、广告、促销或其他目的使用、公布或复制阿里云的名称（包括但不限于单独为或以组合形式包含”阿里云”、Aliyun”、“万网”等阿里云和/或其关联公司品牌，上述品牌的附属标志及图案或任何类似公司名称、商号、商标、产品或服务名称、域名、图案标示、标志、标识或通过特定描述使第三方能够识别阿里云和/或其关联公司）。
6. 如若发现本文档存在任何错误，请与阿里云取得直接联系。

通用约定

格式	说明	样例
	该类警示信息将导致系统重大变更甚至故障，或者导致人身伤害等结果。	 **禁止:** 重置操作将丢失用户配置数据。
	该类警示信息可能导致系统重大变更甚至故障，或者导致人身伤害等结果。	 **警告:** 重启操作将导致业务中断，恢复业务所需时间约10分钟。
	用于补充说明、最佳实践、窍门等，不是用户必须了解的内容。	 **说明:** 您也可以通过按Ctrl + A选中全部文件。
>	多级菜单递进。	设置 > 网络 > 设置网络类型
粗体	表示按键、菜单、页面名称等UI元素。	单击 **确定**。
`courier` 字体	命令。	执行 `cd /d C:/windows` 命令，进入Windows系统文件夹。
*## *	表示参数、变量。	`bae log list --instanceid` *`Instance_ID`*
[]或者[a|b]	表示可选项，至多选择一个。	`ipconfig` *`[-all|-t]`*
{}或者{a|b}	表示必选项，至多选择一个。	`swich` *`{stand | slave}`*

目录

1 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
```

2 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
```

3 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);
        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。

4 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
```

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_ke
y));
            }
            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
```

6 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

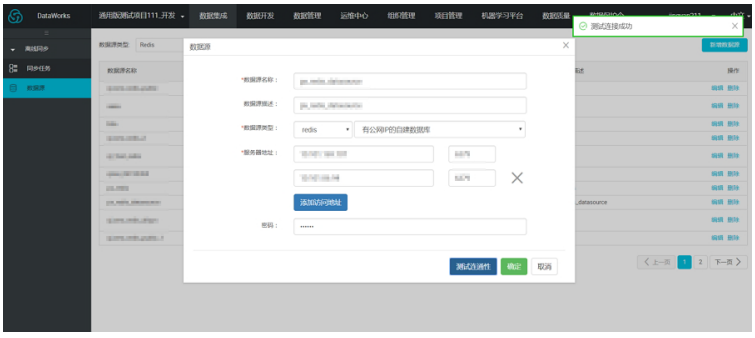
Redis 数据源支持写入 Redis 的通道，可以通过脚本模式配置同步任务 。

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。
1. 单击顶部菜单栏中数据集成模块的数据源。
2. 单击新增数据源。
3. 在新建数据源对话框中，选择数据源类型为 Redis。
4. 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 。

5. 完成上述信息项的配置后，单击测试连通性。
6. 测试连通性通过后，单击确定。

二、配置脚本模式的同步任务

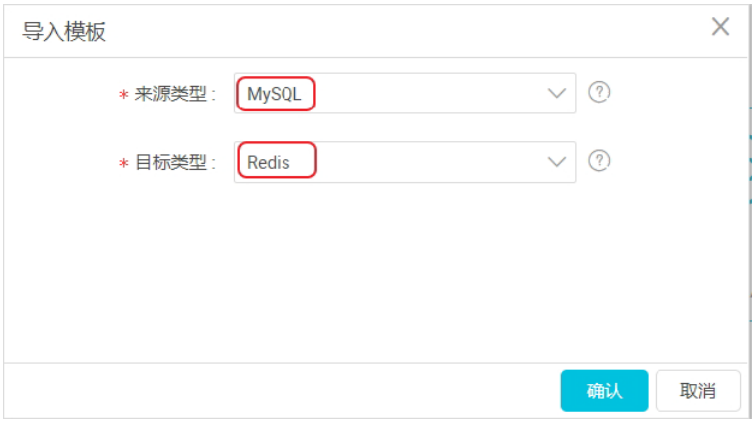
1. 以项目管理员身份进入数加管理控制台，单击大数据开发套件下对应项目操作栏中的进入工作区。

2. 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。

3. 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：



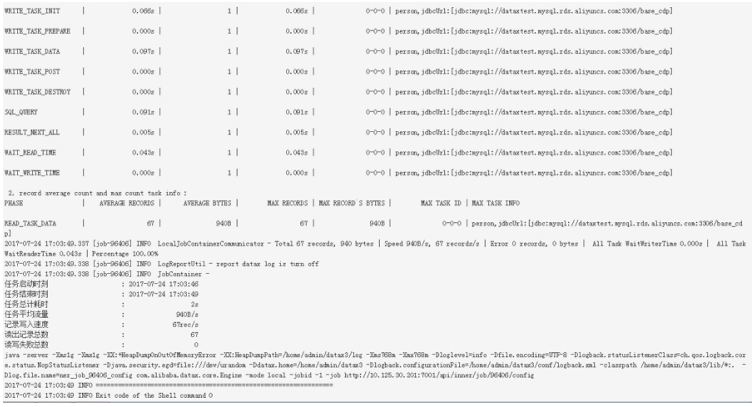
4. 在脚本模式配置页面，根据自身情况进行配置，如有问题可单击右上方的 Redis Writer 帮助手册进行查看。如下图所示：

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。

7 热点Key问题的发现与解决

热点问题概述

产生原因

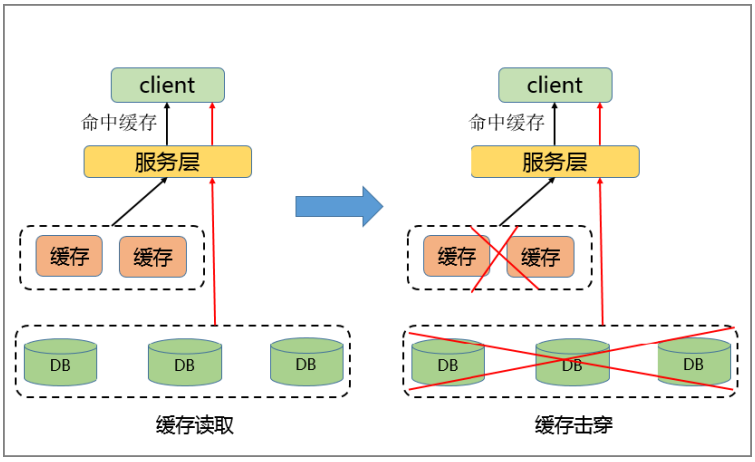
热点问题产生的原因大致有以下两种：

- 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



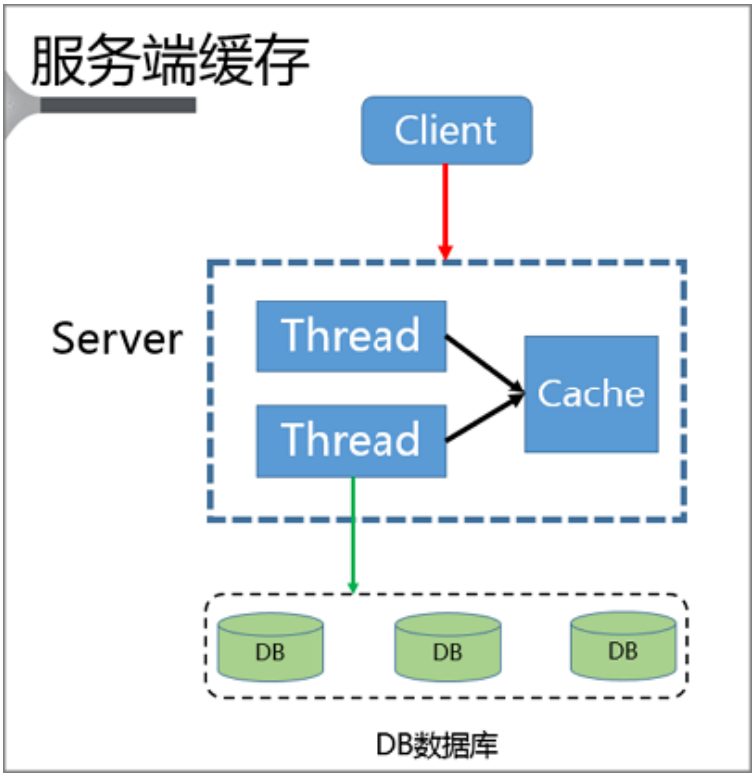
- 流量集中，达到物理网卡上限。
- 请求过多，缓存分片服务被打垮。
- DB击穿，引起业务雪崩。

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

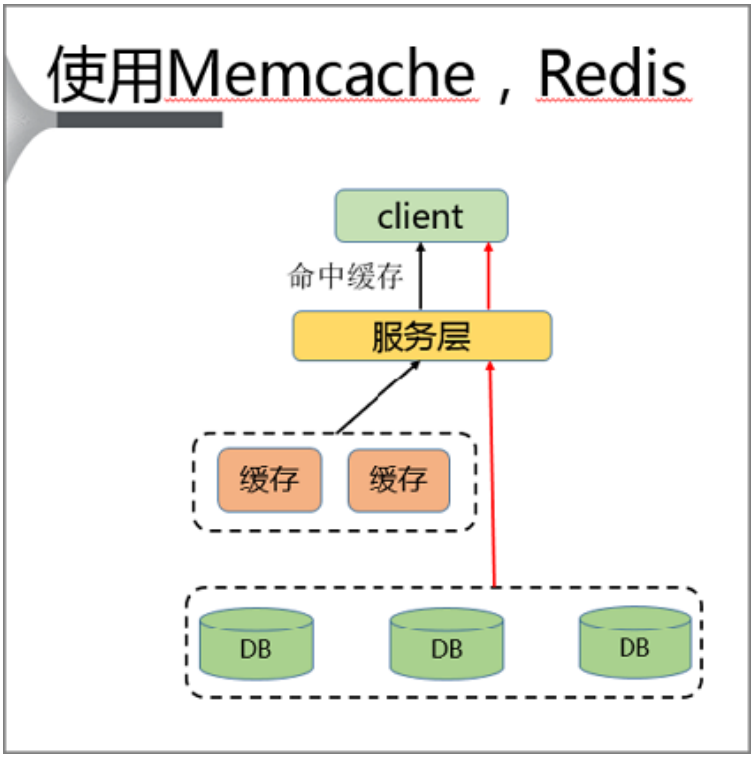


首先Client会将请求发送至Server上，而Server又是一个多线程的服务，本地就具有一个基于Cache LRU策略的缓存空间。当Server本身就拥堵时，Server不会将请求进一步发送给DB而是直接返回，只有当Server本身畅通时才会将Client请求发送至DB，并且将该数据重新写入到缓存中。此时就完成了缓存的访问跟重建。

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

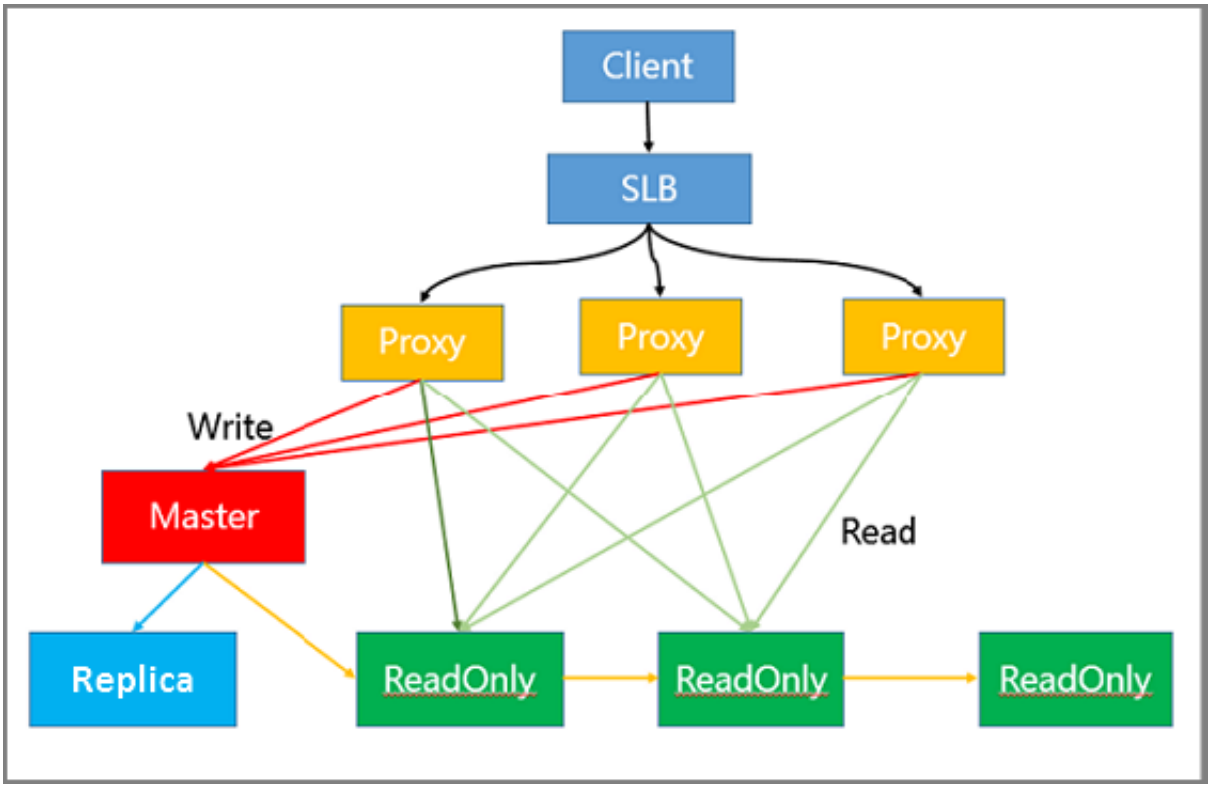
使用本地缓存则存在以下问题：

- 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

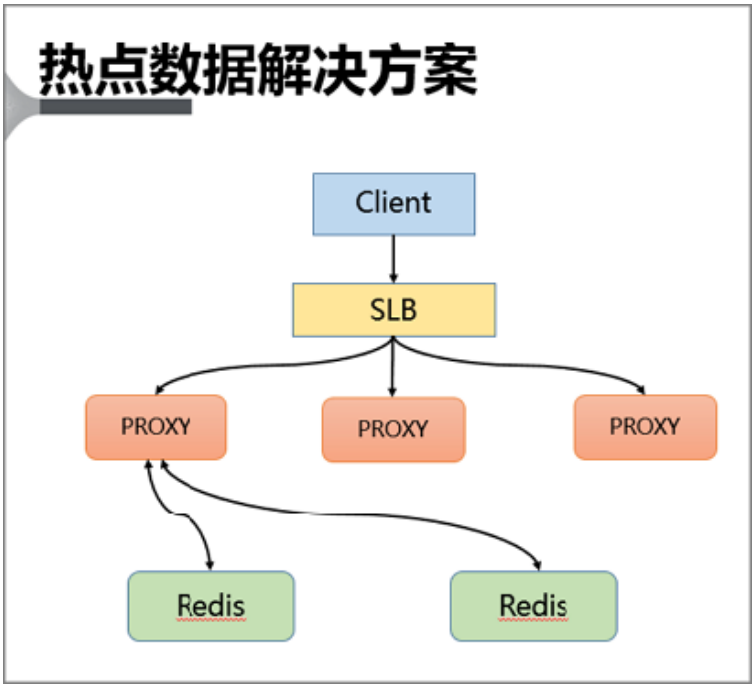


架构中各节点的作用如下：

- 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
- Replica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。

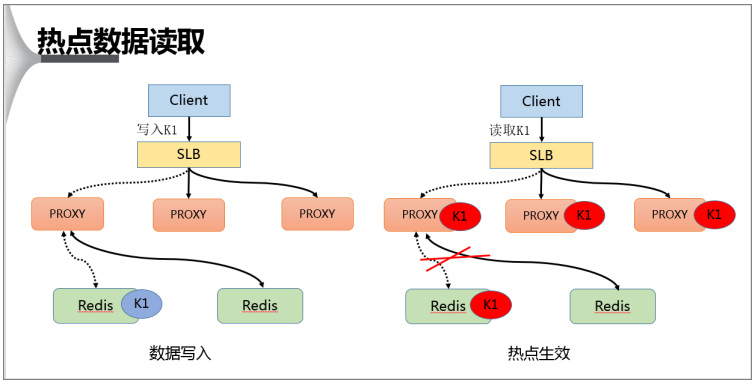
在热点key的解决上是采用在服务端增加缓存的方式进行。具体来说就是在Proxy上增加本地缓存，本地缓存采用LRU算法来缓存热点数据，后端db节点增加热点数据计算模块来返回热点数据。

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

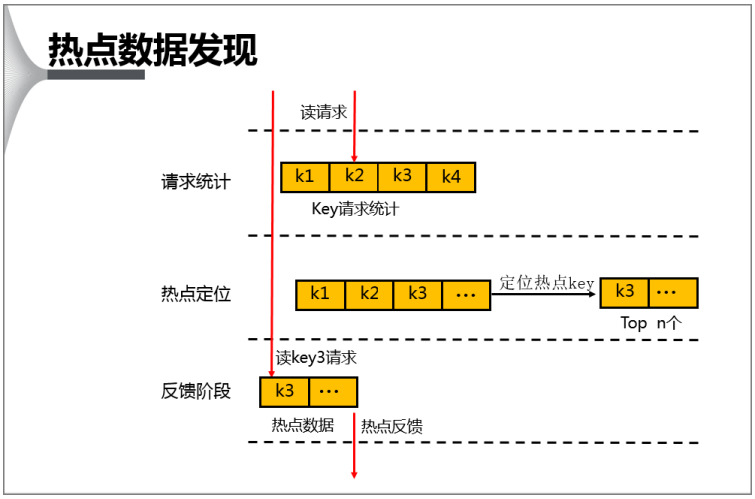
热点key处理

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。

8 解密 Redis 助力双十一背后的技术

背景介绍

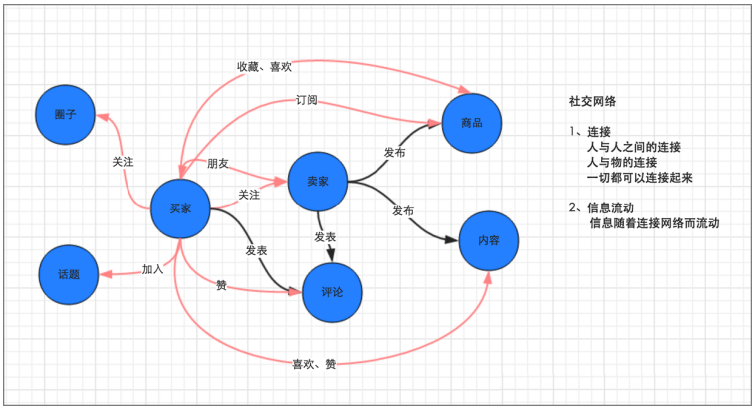
双十一如火如荼，云数据库 Redis 版也圆满完成了双十一的保障工作。目前云数据库 Redis 版提供了标准单副本、标准双副本和集群版本。

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。

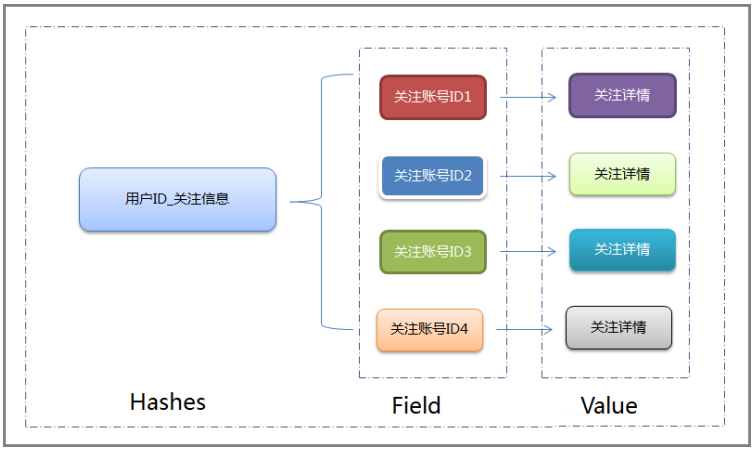
微淘社区之亿级关系链存储

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

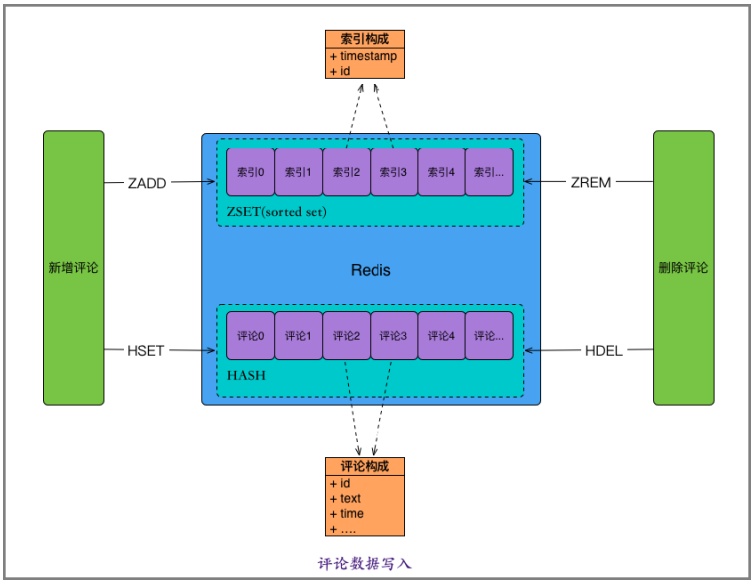


天猫直播之评论商品游标分页

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。

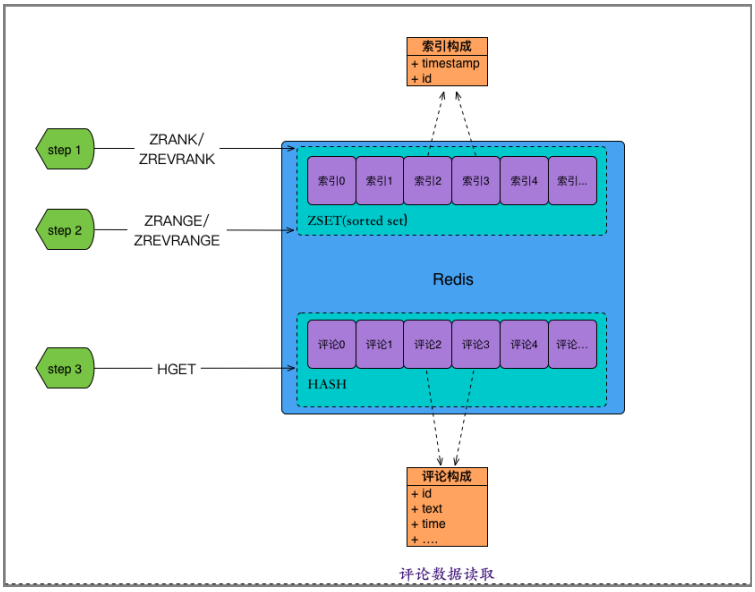
无线直播系统使用 Redis 优化该场景的业务，保证了直播评论接口的成功率，并能够保证5万以上的 TPS 和毫秒级的 response time 请求。直播系统对于每个直播会写入两份数据，分别为索引和评论数据，索引数据为 SortedSet 的数据结构用于对评论的排序，而评论数据使用 Hashes 进行存储，在获取评论的时候通过索引拿到需要的索引 id 之后通过 Hashes 的读取来获得评论的列表。评论的写入过程如下:



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:

1. 获取当前索引位置
2. 获取索引列表

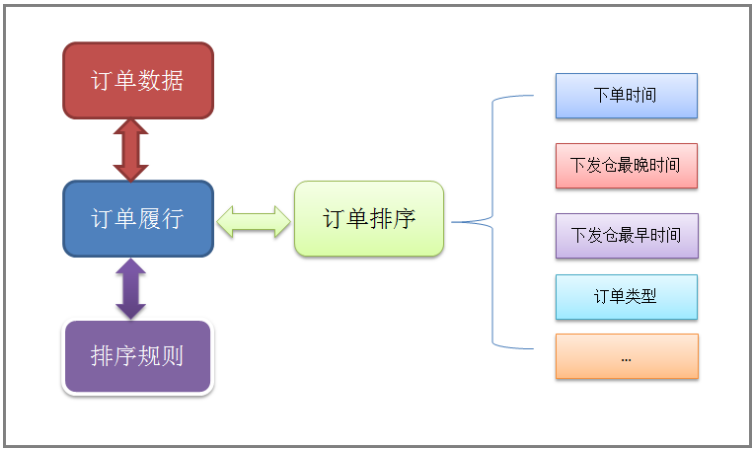
3. 获取评论数据



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。



9 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，replica作为备库不对外提供服务，只有在发生HA的时候，replica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

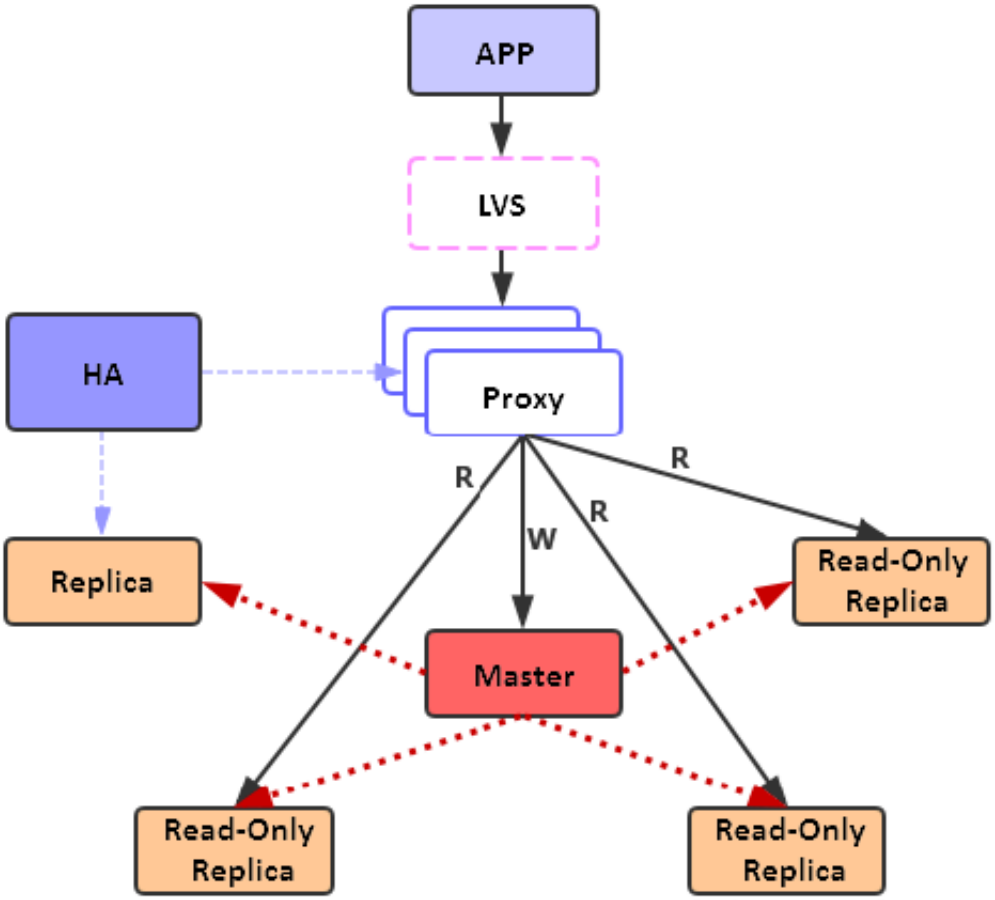
Redis集群模式有redis-proxy、master、replica、HA等几个角色。在读写分离实例中，新增read-only replica角色来承担读流量，replica作为热备不提供服务，架构上保持对现有集群规格的兼容性。redis-proxy按权重将读写请求转发到master或者某个read-only replica上；HA负责监控DB节点的健康状态，异常时发起主从切换或重搭read-only replica，并更新路由。

一般来说，根据master和read-only replica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read-only replica直接和master保持同步，每个read-only replica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，read-only replica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，read-only replica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read-only replica的增加而降低。此外，星型架构会让master的出口带宽随着read-only replica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。

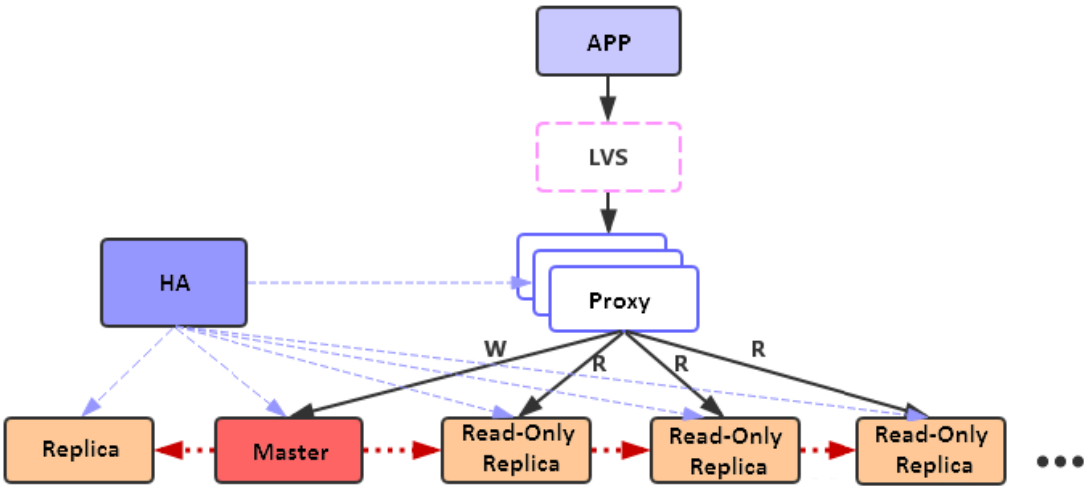


链式复制

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

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read-only replica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read-only replica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，多分片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read-only replica（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read-only replica宕机，HA也能及时感知，然后重搭一个新的read-only replica，下线宕机节点。

除HA之外，redis-proxy也能实时感知每个read-only replica的状态。在某个read-only replica异常期间，redis-proxy会自动降低这个节点的权重，如果发现某个read-only replica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read-only replica的资源。

目前单shard对外售卖1 master + 1/3/5 read-only replica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 MB/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read-only replica数量。

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

后续

Redis主从异步复制，从read-only replica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。

10 JedisPool 资源池优化

合理的 JedisPool 资源池参数设置能够有效地提升 Redis 性能。本文档将对 JedisPool 的使用和资源池的参数进行详细说明，并提供优化配置的建议。

使用方法

以 Jedis 2.9.0 为例，其 Maven 依赖如下：

```
<dependency>
    <groupId>redis.clients</groupId>
    <artifactId>jedis</artifactId>
    <version>2.9.0</version>
    <scope>compile</scope>
</dependency>
```

Jedis 使用 Apache Commons-pool2 对资源池进行管理，在定义 JedisPool 时需注意其关键参数 GenericObjectPoolConfig（资源池）。该参数的使用示例如下，其中的参数的说明请参见下文。

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

JedisPool 的初始化方法如下：

```
// redisHost为实例的IP，  redisPort 为实例端口，redisPassword 为实例的密码，
timeout 既是连接超时又是读写超时
JedisPool jedisPool = new JedisPool(jedisPoolConfig, redisHost,
redisPort, timeout, redisPasswor//d);
//执命令如下
Jedis jedis = null;
try {
    jedis = jedisPool.getResource();
    //具体的命令
    jedis.executeCommand()
} catch (Exception e) {
    logger.error(e.getMessage(), e);
} finally {
    //在 JedisPool 模式下，Jedis 会被归还给资源池
    if (jedis != null)
        jedis.close();
```

```
}
```

参数说明

Jedis 连接就是连接池中 JedisPool 管理的资源， JedisPool 保证资源在一个可控范围内，并且保障线程安全。使用合理的 `GenericObjectPoolConfig` 配置能够提升 Redis 的服务性能，降低资源开销。下列两表将对一些重要参数进行说明，并提供设置建议。

表 10-1: 资源设置与使用相关参数

参数	说明	默认值	建议
maxTotal	资源池中的最大连接数	8	参见关键参数设置建议。
maxIdle	资源池允许的最大空闲连接数	8	参见关键参数设置建议。
minIdle	资源池确保的最少空闲连接数	0	参见关键参数设置建议。
blockWhenExhausted	当资源池用尽后，调用者是否要等待。只有当值为 true 时，下面的 `maxWaitMillis` 才会生效。	true	建议使用默认值。
maxWaitMillis	当资源池连接用尽后，调用者的最大等待时间（单位为毫秒）。	-1（表示永不超时）	不建议使用默认值。
testOnBorrow	向资源池借用连接时是否做连接有效性检测（ping）。检测到的无效连接将会被移除。	false	业务量很大时候建议设置为 false，减少一次 ping 的开销。
testOnReturn	向资源池归还连接时是否做连接有效性检测（ping）。检测到无效连接将会被移除。	false	业务量很大时候建议设置为 false，减少一次 ping 的开销。
jmxEnabled	是否开启 JMX 监控	true	建议开启，请注意应用本身也需要开启。

空闲 Jedis 对象检测由下列四个参数组合完成， `testWhileIdle` 是该功能的开关。

表 10-2: 空闲资源检测相关参数

名称	说明	默认值	建议
testWhileIdle	是否开启空闲资源检测。	false	true
timeBetweenEvictionRunsMillis	空闲资源的检测周期（单位为毫秒）	-1（不检测）	建议设置，周期自行选择，也可以默认也可以使用下方 `JedisPoolConfig` 中的配置。

名称	说明	默认值	建议
minEvictableIdleTimeMillis	资源池中资源的最小空闲时间（单位为毫秒），达到此值后空闲资源将被移除。	180000（即30分钟）	可根据自身业务决定，一般默认值即可，也可以考虑使用下方 JeidsPoolConfig 中的配置。
numTestsPerEvictionRun	做空闲资源检测时，每次检测资源的个数。	3	可根据自身应用连接数进行微调，如果设置为-1，就是对所有连接做空闲监测。

为了方便使用，Jedis 提供了 JedisPoolConfig ，它继承了 GenericObjectPoolConfig 在空闲检测上的一些设置。

```
public class JedisPoolConfig extends GenericObjectPoolConfig {
  public JedisPoolConfig() {
    // defaults to make your life with connection pool easier :)
    setTestWhileIdle(true);
    //
    setMinEvictableIdleTimeMillis(60000);
    //
    setTimeBetweenEvictionRunsMillis(30000);
    setNumTestsPerEvictionRun(-1);
    }
}
```

 说明:

可以在 org.apache.commons.pool2.impl.BaseObjectPoolConfig 中查看全部默认值。

关键参数设置建议

maxTotal（最大连接数）

想合理设置maxTotal（最大连接数）需要考虑的因素较多，如：

- 业务希望的 Redis 并发量；
- 客户端执行命令时间；
- Redis资源，例如 nodes （如应用个数等） * maxTotal 不能超过 Redis 的最大连接数；
- 资源开销，例如虽然希望控制空闲连接，但又不希望因为连接池中频繁地释放和创建连接造成不必要的开销。

假设一次命令时间，即 borrow|return resource 加上 Jedis 执行命令 （含网络耗时）的平均耗时约为1ms，一个连接的 QPS 大约是1000，业务期望的 QPS 是50000，那么理论上需要的资源池大小是 50000 / 1000 = 50。

但事实上这只是个理论值，除此之外还要预留一些资源，所以 maxTotal 可以比理论值大一些。这个值不是越大越好，一方面连接太多会占用客户端和服务端资源，另一方面对于 Redis 这种高 QPS 的服务器，如果出现大命令的阻塞，即使设置再大的资源池也无济于事。

maxIdle 与 minIdle

maxIdle 实际上才是业务需要的最大连接数，maxTotal 是为了给出余量，所以 maxIdle 不要设置得过小，否则会有 new Jedis （新连接）开销，而 minIdle 是为了控制空闲资源检测。

连接池的最佳性能是 maxTotal = maxIdle ，这样就避免了连接池伸缩带来的性能干扰。但如果并发量不大或者 maxTotal 设置过高，则会导致不必要的连接资源浪费。

您可以根据实际总 QPS 和调用 Redis 的客户端规模整体评估每个节点所使用的连接池大小。

使用监控获取合理值

在实际环境中，比较可靠的方法是通过监控来尝试获取参数的最佳值。可以考虑通过 JMX 等方式实现监控，从而找到合理值。

常见问题

资源不足

下面两种情况均属于无法从资源池获取到资源。

- 超时：

```
redis.clients.jedis.exceptions.JedisConnectionException: Could not
get a resource from the pool
…
Caused by: java.util.NoSuchElementException: Timeout waiting for
idle object
at org.apache.commons.pool2.impl.GenericObjectPool.borrowObject(
GenericObjectPool.java:449)
```

- blockWhenExhausted 为 false ，因此不会等待资源释放：

```
redis.clients.jedis.exceptions.JedisConnectionException: Could not
get a resource from the pool
…
Caused by: java.util.NoSuchElementException: Pool exhausted
at org.apache.commons.pool2.impl.GenericObjectPool.borrowObject(
GenericObjectPool.java:464)
```

此类异常的原因不一定是资源池不够大，请参见关键参数设置建议中的分析。建议从网络、资源池参数设置、资源池监控（如果对 JMX 监控）、代码（例如没执行jedis.close()）、慢查询、DNS等方面进行排查。

预热 JedisPool

由于一些原因（如超时时间设置较小等），项目在启动成功后可能会出现超时。JedisPool 定义最大资源数、最小空闲资源数时，不会在连接池中创建 Jedis 连接。初次使用时，池中没有资源使用则会先 `new Jedis`，使用后再放入资源池，该过程会有一定的时间开销，所以建议在定义 JedisPool 后，以最小空闲数量为基准对 JedisPool 进行预热，示例如下：

```
List<Jedis> minIdleJedisList = new ArrayList<Jedis>(jedisPoolConfig.
getMinIdle());

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

for (int i = 0; i < jedisPoolConfig.getMinIdle(); i++) {
    Jedis jedis = null;
    try {
        jedis = minIdleJedisList.get(i);
        jedis.close();
    } catch (Exception e) {
        logger.error(e.getMessage(), e);
    } finally {

    }
}
```

11 集群实例特定子节点中热点Key的分析方法

您可以使用阿里云自研的 imonitor 命令监控 Redis 集群中某一节点的请求状态，并利用请求解析工具 redis-faina 快速地从监控数据中分析出热点 Key 和命令。

背景信息

在使用云数据库 Redis 集群版的过程中，如果某一节点上的热点 Key 流量过大，可能导致服务器中其它服务无法进行。若热点 Key 的缓存超过当前的缓存容量，就会产生缓存分片服务负载过高，进而造成缓存雪崩等严重问题。

您可以利用云数据库 Redis 版的性能监控和报警规则对集群状况进行实时监控并设置告警，在发现特定子节点负载突出时，使用 imonitor 命令查看该节点的客户端请求，并使用 redis-faina 分析出热点 Key 。

前提条件

- 已部署与云数据库 Redis 集群版互通的 ECS 实例。
- ECS 实例中已安装 Python 和 Telnet。

> **说明:**
> 本文中的示例环境使用 CentOS 7.4 系统和 Python 2.7.5。

操作步骤

1. 在 ECS 实例中，以 Telnet 方式连接到 Redis 集群。
   a. 使用`# telnet <host> <port>`连接到 Redis 集群。

   > **说明:**
   > `host`为 Redis 集群的连接地址，`port`为连接端口（默认为6379）。

   b. 输入`auth <password>`进行认证。

   > **说明:**

`password`为 Redis 集群的密码。

```
Welcome to Alibaba Cloud Elastic Compute Service !

[root@redisTest ~]# telnet r-b[redacted]4.redis.rds.aliyuncs.com 6379
Trying 1[redacted]...
Connected to r-b[redacted]4.redis.rds.aliyuncs.com.
Escape character is '^]'.
auth [redacted]a
+OK
```

说明:

返回`+OK`表示连接成功。

2. 使用`imonitor <db_idx>`收集目的节点的请求数据。

```
imonitor 0
+OK
+1543975816.789076 [0 [redacted]] "INFO" "replication"
+1543975833.071774 [0 [redacted]] "INFO" "replication"
+1543975842.251665 [0 127.0.0.1:42442] "INFO" "keyspace"
+1543975842.262597 [0 127.0.0.1:42442] "INFO" "all"
+1543975848.336031 [0 [redacted]] "INFO" "replication"
```

说明:

`imonitor` 命令与 `iinfo`、`iscan` 类似，在 `monitor` 命令的基础上新增了一个参数，用户指定 `monitor` 执行的节点（db_idx），db_idx 的范围是 [0, nodecount)，nodecount 可以通过 `info` 命令获取，或者从控制台上的实例拓扑图中查看。

本例中目的节点的 db_idx 为 0 。

返回`+OK`后将会持续输出监控到的请求记录。

3. 根据需要收集一定数量的监控数据，之后输入 `QUIT` 命令并按 Enter 关闭 Telnet 连接。
4. 将监控数据保存到一个 *.txt* 文件中，删除行首的 “+”（可在文本编辑工具中使用全部替换的方式）删除。保存的文件如下。

```
[root@redisTest ~]# cat imonitorOut.txt
1543995847.659482 [0 [redacted]] "INFO" "replication"
1543995856.057381 [0 127.0.0.1:58802] "INFO" "keyspace"
1543995856.070002 [0 127.0.0.1:58802] "INFO" "all"
1543995861.653458 [0 [redacted]] "INFO" "ALL"
1543995862.782848 [0 [redacted]] "INFO" "ALL"
1543995862.799096 [0 [redacted]] "INFO" "ALL"
1543995862.863230 [0 [redacted]] "INFO" "CLUSTER"
1543995862.876389 [0 [redacted]] "scan" "0" "MATCH" "*" "COUNT" "3000"
1543995862.942649 [0 [redacted] "INFO" "replication"
1543995862.943303 [0 [redacted]] "TYPE" "customer:18016"
1543995862.955943 [0 [redacted]] "TYPE" "customer:17167"
```

5. 创建进行请求分析的 Python 脚本，保存为 *redis-faina.py* 。代码如下。

```
#! /usr/bin/env python
import argparse
import sys
from collections import defaultdict
import re

line_re_24 = re.compile(r"""
    ^(?P<timestamp>[\d\.]+)\s(\(db\s(?P<db>\d+)\)\s)?"(?P<command>\w
+)"(\s"(?P<key>[^(?<!\\)"]+)(?<!\\)")?(\s(?P<args>.+))?$
    """, re.VERBOSE)

line_re_26 = re.compile(r"""
    ^(?P<timestamp>[\d\.]+)\s\[(?P<db>\d+)\s\d+\.\d+\.\d+\.\d+:\d+]\
s"(?P<command>\w+)"(\s"(?P<key>[^(?<!\\)"]+)(?<!\\)")?(\s(?P<args>.
+))?$
    """, re.VERBOSE)

class StatCounter(object):

    def __init__(self, prefix_delim=':', redis_version=2.6):
        self.line_count = 0
        self.skipped_lines = 0
        self.commands = defaultdict(int)
        self.keys = defaultdict(int)
        self.prefixes = defaultdict(int)
        self.times = []
        self._cached_sorts = {}
        self.start_ts = None
        self.last_ts = None
        self.last_entry = None
        self.prefix_delim = prefix_delim
        self.redis_version = redis_version
        self.line_re = line_re_24 if self.redis_version < 2.5 else
line_re_26

    def _record_duration(self, entry):
        ts = float(entry['timestamp']) * 1000 * 1000 # microseconds
        if not self.start_ts:
            self.start_ts = ts
            self.last_ts = ts
        duration = ts - self.last_ts
        if self.redis_version < 2.5:
            cur_entry = entry
        else:
            cur_entry = self.last_entry
            self.last_entry = entry
        if duration and cur_entry:
            self.times.append((duration, cur_entry))
        self.last_ts = ts

    def _record_command(self, entry):
        self.commands[entry['command']] += 1

    def _record_key(self, key):
        self.keys[key] += 1
        parts = key.split(self.prefix_delim)
        if len(parts) > 1:
            self.prefixes[parts[0]] += 1

    @staticmethod
    def _reformat_entry(entry):
```

```
        max_args_to_show = 5
        output = '"%(command)s"' % entry
        if entry['key']:
            output += ' "%(key)s"' % entry
        if entry['args']:
            arg_parts = entry['args'].split(' ')
            ellipses = ' ...' if len(arg_parts) > max_args_to_show
else ''
            output += ' %s%s' % (' '.join(arg_parts[0:max_args_t
o_show]), ellipses)
        return output


    def _get_or_sort_list(self, ls):
        key = id(ls)
        if not key in self._cached_sorts:
            sorted_items = sorted(ls)
            self._cached_sorts[key] = sorted_items
        return self._cached_sorts[key]

    def _time_stats(self, times):
        sorted_times = self._get_or_sort_list(times)
        num_times = len(sorted_times)
        percent_50 = sorted_times[int(num_times / 2)][0]
        percent_75 = sorted_times[int(num_times * .75)][0]
        percent_90 = sorted_times[int(num_times * .90)][0]
        percent_99 = sorted_times[int(num_times * .99)][0]
        return (("Median", percent_50),
                ("75%", percent_75),
                ("90%", percent_90),
                ("99%", percent_99))

    def _heaviest_commands(self, times):
        times_by_command = defaultdict(int)
        for time, entry in times:
            times_by_command[entry['command']] += time
        return self._top_n(times_by_command)

    def _slowest_commands(self, times, n=8):
        sorted_times = self._get_or_sort_list(times)
        slowest_commands = reversed(sorted_times[-n:])
        printable_commands = [(str(time), self._reformat_entry(entry
)) \
                              for time, entry in slowest_commands]
        return printable_commands

    def _general_stats(self):
        total_time = (self.last_ts - self.start_ts) / (1000*1000)
        return (
            ("Lines Processed", self.line_count),
            ("Commands/Sec", '%.2f' % (self.line_count / total_time
))
        )

    def process_entry(self, entry):
        self._record_duration(entry)
        self._record_command(entry)
        if entry['key']:
            self._record_key(entry['key'])

    def _top_n(self, stat, n=8):
        sorted_items = sorted(stat.iteritems(), key = lambda x: x[1
], reverse = True)
        return sorted_items[:n]
```

```
    def _pretty_print(self, result, title, percentages=False):
        print title
        print '=' * 40
        if not result:
            print 'n/a\n'
            return

        max_key_len = max((len(x[0]) for x in result))
        max_val_len = max((len(str(x[1])) for x in result))
        for key, val in result:
            key_padding = max(max_key_len - len(key), 0) * ' '
            if percentages:
                val_padding = max(max_val_len - len(str(val)), 0) *
 ' '
                val = '%s%s\t(%.2f%%)' % (val, val_padding, (float(
val) / self.line_count) * 100)
            print key,key_padding,'\t',val
        print


    def print_stats(self):
        self._pretty_print(self._general_stats(), 'Overall Stats')
        self._pretty_print(self._top_n(self.prefixes), 'Top Prefixes
', percentages = True)
        self._pretty_print(self._top_n(self.keys), 'Top Keys',
percentages = True)
        self._pretty_print(self._top_n(self.commands), 'Top Commands
', percentages = True)
        self._pretty_print(self._time_stats(self.times), 'Command
Time (microsecs)')
        self._pretty_print(self._heaviest_commands(self.times), '
Heaviest Commands (microsecs)')
        self._pretty_print(self._slowest_commands(self.times), '
Slowest Calls')

    def process_input(self, input):
        for line in input:
            self.line_count += 1
            line = line.strip()
            match = self.line_re.match(line)
            if not match:
                if line != "OK":
                    self.skipped_lines += 1
                continue
            self.process_entry(match.groupdict())

if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument(
        'input',
        type = argparse.FileType('r'),
        default = sys.stdin,
        nargs = '?',
        help = "File to parse; will read from stdin otherwise")
    parser.add_argument(
        '--prefix-delimiter',
        type = str,
        default = ':',
        help = "String to split on for delimiting prefix and rest of
 key",
        required = False)
    parser.add_argument(
        '--redis-version',
```

```
        type = float,
        default = 2.6,
        help = "Version of the redis server being monitored",
        required = False)
    args = parser.parse_args()
    counter = StatCounter(prefix_delim = args.prefix_delimiter,
redis_version = args.redis_version)
    counter.process_input(args.input)
    counter.print_stats()
```

说明:

以上脚本来自 *redis-faina*。

6. 使用`python redis-faina imonitorOut.txt`命令解析监控数据。其中*imonitorOut.txt*为本文示例中保存的监控数据。

```
[root@redisTest ~]# python redis-faina.py imonitorOut.txt
Overall Stats
=====================================
Lines Processed          311
Commands/Sec             0.88

Top Prefixes
=====================================
customer                 132     (42.44%)
user_agent               24      (7.72%)
simple_registration      12      (3.86%)
detailed_registration    9       (2.89%)
company                  4       (1.29%)

Top Keys
=====================================
customer:1446            122     (39.23%)
ALL                      68      (21.86%)
replication              29      (9.32%)
all                      15      (4.82%)
keyspace                 15      (4.82%)
user_agent:17358         8       (2.57%)
user_agent:10722         4       (1.29%)
customer:4968            1       (0.32%)

Top Commands
=====================================
INFO    128     (41.16%)
HGET    121     (38.91%)
TYPE    50      (16.08%)
HLEN    3       (0.96%)
TTL     3       (0.96%)
HSCAN   3       (0.96%)
scan    1       (0.32%)
GET     1       (0.32%)

Command Time (microsecs)
=====================================
Median          603448.0
75%             1556677.0
90%             5215846.0
99%             8019603.0

Heaviest Commands (microsecs)
=====================================
INFO    231775519.75
HGET    103355620.75
GET     7377767.75
```

说明:

在以上分析结果中， Top Keys 显示该时间段内请求次数最多的键， Top Commands 显示使用最频繁的命令。您可以根据分析情况解决热点 Key 问题。

12 使用 Redis 搭建视频直播间信息系统

您可以使用云数据库 Redis 版方便快捷地构建大流量、低延迟的视频直播间消息服务。

背景信息

视频直播间作为直播系统对外的表现形式，是整个系统的核心之一。除了视频直播窗口外，直播间的在线用户、礼物、评论、点赞、排行榜等数据信息时效性高，互动性强，对系统时延有着非常高的要求，非常适合使用 Redis 缓存服务来处理。

本篇最佳实践将向您展示使用云数据库 Redis 版搭建视频直播间信息系统的示例。您将了解三类信息的构建方法：

- 实时排行类信息
- 计数类信息
- 时间线信息

实时排行类信息

实时排行类信息包含直播间在线用户列表、各种礼物的排行榜、弹幕消息（类似于按消息维度排序的的消息排行榜）等，适合使用 Redis 中的有序集合（sorted set）结构进行存储。

Redis 集合使用空值散列表（hash table）实现，因此对集合的增删改查操作的时间复杂度都是O（1）。有序集合中的每个成员都关联一个分数（score），可以方便地实现排序等操作。下面以增加和返回弹幕消息为例对有序集合在直播间信息系统中的实际运用进行说明。

- 以 unix timestamp + 毫秒数为分值，记录 user55 的直播间增加的5条弹幕：

```
redis> ZADD user55:_danmu 1523959031601166 message111111111111
(integer) 1
11.160.24.14:3003> ZADD user55:_danmu 1523959031601266 message222
222222222
(integer) 1
11.160.24.14:3003> ZADD user55:_danmu 1523959088894232 message33333
(integer) 1
11.160.24.14:3003> ZADD user55:_danmu 1523959090390160 message444444
(integer) 1
11.160.24.14:3003> ZADD user55:_danmu 1523959092951218 message5555
(integer) 1
```

- 返回最新的3条弹幕信息：

```
redis> ZREVRANGEBYSCORE user55:_danmu +inf -inf LIMIT 0 3
1) "message5555"
2) "message444444"
```

```
3) "message33333"
```

- 返回指定时间段内的3条弹幕信息：

```
redis> ZREVRANGEBYSCORE user55:_danmu 1523959088894232 -inf LIMIT 0
3
1) "message33333"
2) "message222222222222"
3) "message111111111111"
```

计数类信息

计数类信息以用户相关数据为例，有未读消息数、关注数、粉丝数、经验值等等。这类消息适合以Redis中的散列（hash）结构进行存储。比如关注数可以用如下的方法处理：

```
redis> HSET user:55 follower 5
(integer) 1
redis> HINCRBY user:55 follower 1 //关注数+1
(integer) 6
redis> HGETALL user:55
1) "follow"
2) "6"
```

时间线信息

时间线信息是以时间为维度的信息列表，典型有主播动态、新帖等。这类信息是按照固定的时间顺序排列，可以使用列表（list）或者有序列表来存储，请参考以下示例。

```
redis> LPUSH user:55_recent_activitiy  '{datetime:201804112010,type:
publish,title:开播啦,content:加油}'
(integer) 1
redis> LPUSH user:55_recent_activitiy '{datetime:201804131910,type:
publish,title:请假,content:抱歉，今天有事鸽一天}'
(integer) 2
redis> LRANGE user:55_recent_activitiy 0 10
1) "{datetime:201804131910,type:publish,title:\xe8\xaf\xb7\xe5\x81\x87
\",content:\xe6\x8a\xb1\xe6\xad\x89\xef\xbc\x8c\xe4\xbb\x8a\xe5\xa4\
xa9\xe6\x9c\x89\xe4\xba\x8b\xe9\xb8\xbd\xe4\xb8\x80\xe5\xa4\xa9}"
2) "{datetime:201804112010,type:publish,title:\xe5\xbc\x80\xe6\x92\xad
\xe5\x95\xa6,content:\xe5\x8a\xa0\xe6\xb2\xb9}"
```

相关资源

- 直播系统常见的热点 Key 问题的解决方法请参见热点Key问题的发现与解决。
- 使用 Redis 内存分析方法排除业务中潜在的风险点，找到业务性能瓶颈。
- 云数据库 Redis 集群版助您解决高并发问题。

13 解析Redis持久化的AOF文件

背景信息

在日常开发测试中，为了方便查看历史命令和查看某个Key的记录，需要对AOF文件进行解析。

Redis持久化模式

- RDB 快照模式：该模式用于生成某个时间点的备份信息，并且会对当前的Key value进行编码，然后存储在rdb文件中。
- AOF 持久化模式：该模式类似binlog的形式，会记录服务器所有的写请求，在服务重启时用于恢复原有的数据。

AOF持久化模式的详细说明

Redis客户端和服务端之间通过RESP (REdis Serialization Protocol)进行通信。RESP协议主要由以下几种数据类型组成，每种数据类型的定义如下：

- 简单字符串：

  以+号开头，结尾为rn，比如：+OKrn。
- 错误信息：

  以-号开头，结尾为rn的字符串，比如：-ERR Readonlyrn。
- 整数：

  以冒号开头，结尾为rn，开头和结尾之间为整数，比如（:1rn）。
- 大字符串：

  以$开头，随后为该字符串长度和rn，长度限制512M，最后为字符串内容和rn，比如：$0rnrn。
- 数组：

  以*开头，随后指定数组元素个数并通过rn划分，每个数组元素都可以为上面的四种，比如：*1rn$4rnpingrn。

Redis客户端发送给服务端的是一个数组命令，服务端根据不同命令的实现方式进行回复，并记录到AOF文件中。

AOF文件解析

这里通过Python代码调用hiredis库来进行Redis AOF文件的解析，代码如下：

```
#!/usr/bin/env python
```

```
""" A redis appendonly file parser
"""

import logging
import hiredis
import sys

if len(sys.argv) != 2:
   print sys.argv[0], 'AOF_file'
   sys.exit()
file = open(sys.argv[1])
line = file.readline()
cur_request = line
while line:
    req_reader = hiredis.Reader()
    req_reader.setmaxbuf(0)
    req_reader.feed(cur_request)
    command = req_reader.gets()
    try:
        if command is not False:
            print command
            cur_request = ''
    except hiredis.ProtocolError:
        print 'protocol error'
    line = file.readline()
    cur_request += line
file.close
```

使用以上脚本解析一个AOF文件的结果如下。得到如下结果后方便您随时查看某个Key相关的操作。

```
['PEXPIREAT', 'RedisTestLog', '1479541381558']
['SET', 'RedisTestLog', '39124268']
['PEXPIREAT', 'RedisTestLog', '1479973381559']
['HSET', 'RedisTestLogHash', 'RedisHashField', '16']
['PEXPIREAT', 'RedisTestLogHash', '1479973381561']
['SET', 'RedisTestLogString', '79146']
```

14 Redis 4.0 热点Key查询方法

高性能是Redis最大的特点，保障Redis的性能是Redis使用过程中的必要举措。可能导致Redis性能问题的因素各种各样，而热点Key是其中最常见的因素之一。找出热点Key有利于进一步处理问题，本文介绍利用Redis 4.0版本新增特性查询热点Key的方法。

背景信息

Redis 4.0新增了`allkey-lfu`和`volatile-lfu`两种数据逐出策略，同时还可以通过`OBJECT`命令来获取某个key的访问频度，如下图所示。

```
r-                    .redis.rds.aliyuncs.com:6379> OBJECT FREQ mylist
(integer) 220
```

Redis 原生客户端也增加了`--hotkeys`选项，可以快速帮您找出业务中的热点Key。

> 说明:
> 本文旨在介绍热点Key发现方法，从而优化Redis的性能，因此适用于已经拥有一定的云数据库Redis版使用基础，且在寻求进阶技巧的用户。如果您刚开始接触Redis，建议先阅读产品简介和快速入门。

前提条件

- 拥有与Redis实例互通的ECS实例；
- ECS中已经安装了Redis 4.0以上版本；

  > 说明:
  > 目的为使用其自带的工具redis-cli。

- 云数据库Redis版实例的`maxmemory-policy`参数设置为`volatile-lfu`或`allkeys-lfu`。

  > 说明:
  > 参数修改的方法请参见参数设置。

操作步骤

1. 在有业务进行时，使用以下命令查询热点Key。

```
redis-cli -h r-***************.redis.rds.aliyuncs.com -a <password>
 --hotkeys
```

> 说明:

本文使用*redis-benchmark*模拟业务中大量写入的场景。

表 14-1: 选项说明

名称	说明
-h	指定Redis的连接地址。
-a	指定Redis的认证密码。
--hotkeys	用来查询热点Key。

执行结果

执行命令后得到的结果示例如下：

```
[root@yaozhou src]# redis-cli -h r-[redacted].redis.rds.aliyuncs.com --hotkeys

# Scanning the entire keyspace to find hot keys as well as
# average sizes per key type.  You can use -i 0.1 to sleep 0.1 sec
# per 100 SCAN commands (not usually needed).

[21.01%] Hot key 'key:__rand_int__' found so far with counter 167
[39.46%] Hot key 'mylist' found so far with counter 167
[67.29%] Hot key 'counter:__rand_int__' found so far with counter 51
[82.73%] Hot key 'myset:__rand_int__' found so far with counter 63

-------- summary -------

Sampled 5008 keys in the keyspace!
hot key found with counter: 167 keyname: key:__rand_int__
hot key found with counter: 167 keyname: mylist
hot key found with counter: 63  keyname: myset:__rand_int__
hot key found with counter: 51  keyname: counter:__rand_int__
```

执行结果的`summary`部分即是分析得出的热点Key。