ALIBABA CLOUD

阿里云

云数据库 Redis 版

最佳实践

文档版本：20201106

阿里云

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
危险	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。	危险 重置操作将丢失用户配置数据。
警告	该类警示信息可能会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，恢复业务时间约十分钟。
注意	用于警示信息、补充说明等，是用户必须了解的内容。	注意 权重设置为0，该服务器不会再接受新请求。
说明	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。	单击**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。	在**结果确认**页面，单击**确定**。
Courier字体	命令或代码。	执行 `cd /d C:/window` 命令，进入Windows系统文件夹。
斜体	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。	`switch {active|stand}`

目录

1.企业版最佳实践

1.1. 使用TairGIS实现用户轨迹监测

您可以通过云数据库Redis企业版的TairGIS结构，实现基于点、线、面的用户轨迹监测。

背景信息

基于位置的服务LBS（Location Based Services）使用各种类型的定位技术来获取设备当前的所在位置，通过移动互联网向设备提供信息资源和基础服务。近年来，LBS技术已成为诸多行业应用与研究的热点，在很多应用中起到了举足轻重的作用。

2020年的新冠病毒疫情给全中国乃至世界人民的生命健康带来了巨大威胁，为世界按下了暂停键。中国举全国之力才基本控制住疫情的进一步扩散，暂停的城市开始慢慢复苏，各行各业开始逐步复工、复产、复学。当中国的疫情得到控制时，全球疫情尚未出现拐点，防控形势依然严峻。如果能通过LBS实现用户轨迹监测，不仅可以有效识别风险、保障人群的安全，还可以更好地进行流行病学调查。

云Redis社区版同原生Redis一样，支持Redis Geo命令，可用于描述位置信息，在LBS类应用中能起到一定的作用，但其精度有限，功能较少。相比之下，云Redis企业版（Tair）性能增强系列的TairGIS命令则拥有更全面的功能。

TairGIS是一种使用R-Tree做索引，支持地理信息系统GIS（Geographic Information System）相关接口的数据结构。Redis的原生GEO命令是使用GeoHash和Redis Sorted Set结构完成的，使用1D索引，主要用于点的查询；TairGIS使用2D索引，除了对点的查询，还支持线、面的查询，尤其适合用于判断相交或包含关系，功能更加强大。

TairGIS可以大大降低LBS应用的开发成本。目前常见的儿童和老人的电子围栏安全防护系统，也是TairGIS的典型应用之一。

实现方案

要监测固定人群的行为轨迹，首先要获取用户位置，主要有两种实现方案：

- 用户在手机上主动开启GPS，通过手机GPS定位确定用户位置。
- 与运营商合作确定用户位置。

在类似疫情控制的场景中，监测轨迹的目的是确定用户是否到过某些危险区域，例如疫情高发区等，因此一般无需存储用户的历史轨迹信息，只需要在用户进入到危险区域时报警即可，这也在最大程度上保护了用户隐私。

危险区域可以使用WKT（Well-known text）描述为多边形，保存在TairGIS数据中。用户的轨迹可以用WKT描述成点、线或者多边形，同样保存在TairGIS数据中。之后，使用TairGIS命令即可查询用户轨迹和危险区域的交汇情况，从而判断用户是否经过了危险区域。

> **说明** WKT是一种文本标记语言，用于描述矢量几何对象、空间参照系统及空间参照系统之间的转换。

使用GPS定位或者通过运营商获取用户位置信息后的业务处理方式有所不同，下文将通过示例详细说明。

方案示例

- 使用GPS定位用户位置
  获取用户当前的GPS信息后，可以使用TairGIS的GIS.CONTAINS命令确认该点是否在危险范围内。如果用户在道路上，还可以通过GPS信息匹配道路信息，再使用GIS.INTERSECTS命令确认用户是否即将进入危险区域，实现预警。

用户的GPS信息可以通过WKT描述为POINT（点），例如 `POINT(30 11)` 。道路信息可以通过WKT描述为LINESTRING（线），例如 `LINESTRING (30 10, 40 40)` 。下方的示例代码可以帮助您更好地理解业务的实现逻辑。

```
GIS.ADD your_province your_location 'POLYGON ((30 10, 40 40, 20 40, 10 20, 30 10))' //获取用户GPS信息后将其添加到TairGIS中。
GIS.CONTAINS your_province 'POINT (30 11)'
GIS.INTERSECTS your_province 'LINESTRING (30 10, 40 40)'
```

- 通过运营商获取用户位置
  如果和运营商合作来获取位置信息，在运营商基站部署不密集的地区，获取到的信息一般是一片区域，可能是整个基站的信号覆盖区域，或者基站某方向的扇形区域，该区域通过WKT描述为POLYGON（多边形），例如 `POLYGON ((10 22, 30 45, 16 53， 10 22))` 。之后就可以使用GIS.INTERSECTS命令把该多边形的信息与危险区域进行交汇分析，如下方的示例代码所示。

```
GIS.ADD your_province your_location 'POLYGON ((30 10, 40 40, 20 40, 10 20, 30 10))' //通过运营商基站获取用户位置信息后将其添加到TairGIS中。
GIS.INTERSECTS your_province 'POLYGON ((10 22, 30 45, 16 53, 10 22))'
```

> **说明** 更多TairGIS命令的说明请参见TairGIS命令。

总结

使用云Redis企业版性能增强系列的TairGIS结构，LBS应用可以方便地实现地理信息相关的存储和计算，同时也能满足大并发场景对高性能的需求。

1.2. 高性能分布式锁

分布式锁是大型应用中最常见的功能之一，基于Redis实现分布式锁的方式有很多。本文先介绍并分析常见的分布式锁实现方式，之后结合阿里巴巴集团在使用云数据库Redis企业版和分布式锁方面的业务经验，介绍使用Redis企业版实现高性能分布式锁的实践方案。

分布式锁及其应用场景

应用开发时，如果需要在同进程内的不同线程并发访问某项资源，可以使用各种互斥锁、读写锁；如果一台主机上的多个进程需要并发访问某项资源，则可以使用进程间同步的原语，例如信号量、管道、共享内存等。但如果多台主机需要同时访问某项资源，就需要使用一种在全局可见并具有互斥性的锁了。这种锁就是分布式锁，可以在分布式场景中对资源加锁，避免竞争资源引起的逻辑错误。

分布式锁的特性

- 互斥性
  在任意时刻，只有一个客户端持有锁。
- 不死锁
  分布式锁本质上是一个基于租约（Lease）的租借锁，如果客户端获得锁后自身出现异常，锁能够在一段时间后自动释放，资源不会被锁死。
- 一致性
  硬件故障或网络异常等外部问题，以及慢查询、自身缺陷等内部因素都可能导致Redis发生高可用切换，replica提升为新的master。此时，如果业务对互斥性的要求非常高，锁需要在切换到新的master后保持原状态。

使用原生Redis实现分布式锁

> **说明** 该部分介绍的实现方式同样适用于云Redis社区版。

- 加锁
  在Redis中加锁非常简便，直接使用SET命令即可。示例及关键选项说明如下：

```
SET resource_1 random_value NX EX 5
```

关键选项说明

参数/选项	说明
resource_1	分布式锁的key，只要这个key存在，相应的资源就处于加锁状态，无法被其它客户端访问。
random_value	一个随机字符串，不同客户端设置的值不能相同。
EX	设置过期时间，单位为秒。您也可以使用PX选项设置单位为毫秒的过期时间。
NX	如果需要设置的key在Redis中已存在，则取消设置。

示例代码为resource_1这个key设置了5秒的过期时间，如果客户端不释放这个key，5秒后key将过期，锁就会被系统回收，此时其它客户端就能够再次为资源加锁并访问资源了。

- 解锁
  解锁一般使用DEL命令，但可能存在下列问题。

  i. t1时刻，App1设置了分布式锁resource_1，过期时间为3秒。
  ii. App1由于程序慢等原因等待超过了3秒，而resource_1已经在t2时刻被释放。
  iii. t3时刻，App2获得这个分布式锁。
  iv. App1从等待中恢复，在t4时刻运行 `DEL resource_1` 将App2持有的分布式锁释放了。

  从上述过程可以看出，一个客户端设置的锁，必须由自己解开。因此客户端需要先使用GET命令确认锁是不是自己设置的，然后再使用DEL解锁。在Redis中通常需要用Lua脚本来实现自锁自解：

```
if redis.call("get",KEYS[1]) == ARGV[1] then
  return redis.call("del",KEYS[1])
else
  return 0
end
```

- 续租
  当客户端发现在锁的租期内无法完成操作时，就需要延长锁的持有时间，进行续租（renew）。同解锁一样，客户端应该只能续租自己持有的锁。在Redis中可使用如下Lua脚本来实现续租：

```
if redis.call("get",KEYS[1]) == ARGV[1] then
  return redis.call("expire",KEYS[1], ARGV[2])
else
  return 0
end
```

使用Redis企业版实现分布式锁

使用Redis企业版性能增强型实例的String增强命令，无需Lua即可实现分布式锁。

- 加锁
  加锁方式与原生Redis相同，使用SET 命令：

```
SET resource_1 random_value NX EX 5
```

- 解锁
  直接使用Redis企业版的CAD命令即可实现优雅而高效的解锁：

```
/* if (GET(resource_1) == my_random_value) DEL(resource_1) */
CAD resource_1 my_random_value
```

- 续租
  续租可以直接使用CAS命令实现：

```
CAS resource_1 my_random_value my_random_value EX 10
```

> 说明 CAS命令不会检查新设置的value和原value是否相同。

基于Jedis的示例代码

- 定义CAS/CAD命令

```
enum TairCommand implements ProtocolCommand {
  CAD("CAD"), CAS("CAS");

  private final byte[] raw;

  TairCommand(String alt) {
   raw = SafeEncoder.encode(alt);
  }

  @Override
  public byte[] getRaw() {
   return raw;
  }
}
```

- 加锁

```
public boolean acquireDistributedLock(Jedis jedis,String resourceKey, String randomValue, int expireTim
e) {
  SetParams setParams = new SetParams();
  setParams.nx().ex(expireTime);
  String result = jedis.set(resourceKey,randomValue,setParams);
  return "OK".equals(result);
}
```

- 解锁

```
public boolean releaseDistributedLock(Jedis jedis,String resourceKey, String randomValue) {
  jedis.getClient().sendCommand(TairCommand.CAD,resourceKey,randomValue);
  Long ret = jedis.getClient().getIntegerReply();
  return 1 == ret;
}
```

- 续租

```
public boolean renewDistributedLock(Jedis jedis,String resourceKey, String randomValue, int expireTime
) {
  jedis.getClient().sendCommand(TairCommand.CAS,resourceKey,randomValue,randomValue,"EX",Strin
g.valueOf(expireTime));
  Long ret = jedis.getClient().getIntegerReply();
  return 1 == ret;
}
```

如何保障一致性

Redis的主从同步（replication）是异步进行的，如果向master发送请求修改了数据后master突然出现异常，发生高可用切换，缓冲区的数据可能无法同步到新的master（原replica）上，导致数据不一致。如果丢失的数据跟分布式锁有关，则会导致锁的机制出现问题，从而引起业务异常。下文介绍三种保障一致性的方法。

- 使用红锁（RedLock）
  红锁是Redis作者提出的一致性解决方案。红锁的本质是一个概率问题：如果一个主从架构的Redis在高可用切换期间丢失锁的概率是 `k%` ，那么相互独立的N个Redis同时丢失锁的概率是多少？如果用红锁来实现分布式锁，那么丢锁的概率是 `(k%)^N` 。鉴于Redis极高的稳定性，此时的概率已经完全能满足产品的需求。

  > **说明** 红锁的实现并非这样严格，一般保证 `M(1<M=<N)` 个同时锁上即可，但通常仍旧可以满足需求。

  红锁的问题在于：
  - 加锁和解锁的延迟较大。
  - 难以在集群版或者标准版（主从架构）的Redis实例中实现。

- 占用的资源过多，为了实现红锁，需要创建多个互不相关的云Redis实例或者自建Redis。

- 使用WAIT命令。
  Redis的**WAIT**命令会阻塞当前客户端，直到这条命令之前的所有写入命令都成功从master同步到指定数量的replica，命令中可以设置单位为毫秒的等待超时时间。在云Redis版中使用**WAIT**命令提高分布式锁一致性的示例如下：

  ```
  SET resource_1 random_value NX EX 5
  WAIT 1 5000
  ```

  使用以上代码，客户端在加锁后会等待数据成功同步到replica才继续进行其它操作，最大等待时间为5000毫秒。执行**WAIT**命令后如果返回结果是1则表示同步成功，无需担心数据不一致。相比红锁，这种实现方法极大地降低了成本。
  需要注意的是：
  - **WAIT**只会阻塞发送它的客户端，不影响其它客户端。
  - **WAIT**返回正确的值表示设置的锁成功同步到了replica，但如果在正常返回前发生高可用切换，数据还是可能丢失，此时**WAIT**只能用来提示同步可能失败，无法保证数据不丢失。您可以在WAIT返回异常值后重新加锁或者进行数据校验。
  - 解锁不一定需要使用**WAIT**，因为锁只要存在就能保持互斥，延迟删除不会导致逻辑问题。

- 使用阿里云数据库Redis企业版
  在不考虑组合方案的情况下：
  - 使用红锁最大优势是Redis节点越多则一致性越强。
  - 使用**WAIT**命令最大优势是实现成本低。

  如果使用阿里云数据库Redis企业版：
  - 其特有的高可用HA和数据持久化机制能够有效保护数据安全、确保服务的稳定性，不使用多Redis节点或**WAIT**命令也能提供较高的一致性。
  - 性能增强型实例的**CAS/CAD**命令可以极大降低分布式锁的开发和管理成本，提升锁的性能。
  - 性能增强型实例的性能增强型特性使其能够提供三倍于原生Redis的性能，即使是大并发的分布式锁也不会影响正常的Redis服务。

1.3. 并发控制与乐观锁

在大量请求并发访问和更新Redis中储存的共享资源时，必须有一种精准高效的并发控制机制来防止逻辑异常和数据错误，乐观锁就是这样一种机制。比起原生Redis，云数据库Redis版性能增强型实例集成的TairString模块能帮助您实现性能更高、成本更低的乐观锁。

并发与Last-Writer-Win

下图展示了一个典型的并发导致资源竞争的场景：

1. 初始状态，string类型数据key_1的值为 `hello` 。
2. t1时刻，App1读取到key_1的值 `hello` 。
3. t2时刻，App2读取到key_1的值 `hello` 。
4. t3时刻，App1将key_1的值修改为 `world` 。
5. t4时刻，App2将key_1的值修改为 `universe` 。

key_1的值是由最后一次写入决定的，到了t4时刻，App1对key_1的认知已经出现了明显的误差，后续操作很可能出现问题，这就是所谓的Last-Writer-Win。要解决Last-Writer-Win问题，就需要保证访问并更新string数据这个操作的原子性，或者说，将作为共享资源的string数据转变为具有原子性的变量。您可以使用Redis企业版性能增强型实例的TairString数据结构，构建高性能的乐观锁来达成这个效果。

使用TairString实现乐观锁

TairString，又称为exString（extended string），是一种带版本号的string类型数据结构。原生Redis String仅由key和value组成，而TairString不仅包含key和value，还携带了版本（version），极为适合乐观锁等场景。详细介绍及命令解析请参见TairString命令。

> **说明** TairString与Redis原生String是两种不同的数据结构，相关命令不可混用。

TairString有以下特性：

- 每个key都有对应的version，用于说明key当前的版本。使用**EXSET**命令创建一个key时，默认其version为1。
- 对某个key使用**EXGET**时，可以获取到value和version两个字段。
- 更新TairString的value时，需要校验version，如果校验失败会返回异常信息`ERR update version is stale`。
- value更新后version自动加1。
- 除了比特位（bit）相关操作外，TairString可以覆盖原生Redis String的所有其它功能。

因为这些特性，TairString类型的数据本身就具有锁的机制，使用TairString实现乐观锁就非常方便了，示例如下：

```
while(true){
  {value, version} = EXGET(key);   // 获取key的value和version
  value2 = update(...);          // 先将新value保存到value2
  ret = EXSET(key, value2, version); // 尝试更新key并将返回值赋予变量ret
  if(ret == OK)
    break;                  // 如果返回值为OK则更新成功，跳出循环
  else if (ret.contanis("version is stale"))
    continue;                 // 如果返回值包含"version is stale"则更新失败，重复循环
}
```

> **说明**
>
> - 删除TairString后，即便以相同的key重新设置一条TairString，其version也会是1，而不会继承原TairString的version。
> - 使用ABS选项可以跳过version校验强行覆盖version并更新TairString，详情参见EXSET。

降低乐观锁的性能消耗

前文的示例代码中，如果在执行**EXGET**后该共享资源被其它客户端更新了，当前客户端会获取到更新失败的异常信息，然后重复循环，再次执行**EXGET**获取共享资源的当前value和version，直到更新成功，这样每次循环都有两次访问Redis的IO操作。如果使用TairString的EXCAS命令，可以将两次访问减少为一次，极大地节约系统资源消耗，提升高并发场景下的服务性能。

EXCAS命令可以在调用时携带一个用于校验的version值，如果校验成功则直接更新TairString的value，如果校验失败则返回三个字段：

- `update version is stale`
- value
- version

更新失败后可以直接得到TairString当前的版本，无需再次查询，将原本每个循环需要进行两次的访问减少到一次。示例如下：

```
while(true){
  {ret, value, version} = excas(key, new_value, old_version)  // 直接尝试用CAS命令置换value
  if(ret == OK)
    break;  // 如果返回值为OK则更新成功，跳出循环
  else (if ret.contanis("update version is stale"))  // 如果返回值包含"update version is stale"则更新失败，更新两个变量
    update(value);
    old_version = version;
}
```

1.4. 限流器

在限量抢购或者限时秒杀类场景中，除了要有效应对秒杀前后的流量高峰，还需要防止发生接受的下单量超过商品限购数量的问题，云数据库Redis企业版性能增强型实例的TairString结构支持简洁高效的限流器，可以很好地解决订单超量问题。本文介绍的方案也适用于其它需要限速或者限流的场景。

抢购限流器

TairString是Redis企业版性能增强型实例集成了阿里巴巴Tair后新增的数据结构，比原生Redis String功能更加强大，除了比特位（bit）操作外能够覆盖原生Redis String的所有功能。

TairString的**EXINCRBY/EXINCRBYFLOAT**命令与原生Redis String的**INCRBY/INCRBYFLOAT**命令功能类似，都可对value进行递增或递减运算，但**EXINCRBY/EXINCRBYFLOAT**支持更多选项，例如*EX*、*NX*、*VER*、*MIN*、*MAX*等，详细说明请参见TairString命令。下文介绍的方案涉及*MIN*与*MAX*两个选项：

选项	说明
MIN	设置TairString value的最小值。
MAX	设置TairString value的最大值。

使用原生Redis String实现抢购，代码逻辑复杂，一旦管理不当，容易出现漏网订单，即明明商品已经抢完，却还有用户收到抢购成功的提示，造成不良影响，而使用TairString，只需要非常简单的代码即可实现严谨的订单数量限制，伪代码如下：

```
if(EXINCRBY(key_iphone, -1, MIN:0) == "would overflow")
  run_out();
```

限流计数器

与抢购限流器类似，使用EXINCRBY命令的MAX选项可以实现限流计数器，伪代码如下：

```
if(EXINCRBY(rate_limitor, 1, MAX:1000) == "would overflow")
  traffic_control();
```

限流计数器的应用场景很多，例如并发限流、访问频率限制、密码修改次数限制等等。以并发限流为例，在请求的并发量突然超过系统的性能限制时，为了防止服务彻底崩溃引发更大的问题，采用限速器限制并发量，保证系统处理能力内的请求得到及时回应，是一种较合理的临时解决方案。使用TairString**EXINCRBY**命令，您可以通过简单的代码设置一个并发限流器：

```
public boolean tryAcquire(Jedis jedis,String rateLimitor,int limiter){
  try {
    jedis.getClient().sendCommand(TairCommand.EXINCRBY,rateLimitor,"1","EX","1","MAX",String.valueOf(
limiter));  // 设置限流器
    jedis.getClient().getIntegerReply();
    return true;
  }catch (Exception e){
    if(e.getMessage().contains("increment or decrement would overflow")){  // 检查返回结果中是否包含错误信
息
      return false;
    }
    throw e;
  }
}
```

1.5. TairHash内存消耗与淘汰策略最佳实践

云数据库Redis增强版（Tair）性能增强系列的TairHash数据结构支持高效的动态淘汰策略，可以快速释放内存空间，代价则是会提高TairHash数据的内存消耗。本章节为您介绍TairHash的内存消耗情况和两种淘汰策略，帮助您在淘汰效率和内存消耗间寻找平衡点，降低业务成本。

原生Redis hash与TairHash占用内存的对比测试

在不设置过期时间时，原生Redis hash与云数据库Redis增强版（Tair）性能增强系列的TairHash结构占用的内存相差很小，详细对比请参见下方测试。

- 测试一
  测试环境

测试对象	测试命令	测试条件
TairHash	`EXHSET tairhashkey field value`	◦ field大小：1 KB。 ◦ value大小：1 KB。 ◦ 关闭AOF与RDB备份。 ◦ 实例为单节点，无副本。
原生Redis hash	`HSET hashkey field value`	

测试结果

field数量	TairHash占用内存	原生Redis hash占用内存
10000	29.79 MB	29.79 MB
100000	297.02 MB	297.02 MB
1000000	2.9 GB	2.9 GB

- 测试二
  测试环境

测试对象	测试命令	测试条件
TairHash	EXHSET tairhashkey field value	◦ field大小：64 Byte。 ◦ value大小：10 KB。 ◦ 关闭AOF与RDB备份。 ◦ 实例为单节点，无副本。
原生Redis hash	HSET hashkey field value	

测试结果

field数量	TairHash占用内存	原生Redis hash占用内存
10000	104.17 MB	103.79 MB
100000	1.02 GB	1.02 GB
1000000	10.19 GB	10.19 GB

- 测试三
  测试环境

测试对象	测试命令	测试条件
TairHash	EXHSET tairhashkey field value	◦ field大小：64 Byte。 ◦ value大小：64 Byte。 ◦ 关闭AOF与RDB备份。 ◦ 实例为单节点，无副本。
原生Redis hash	HSET hashkey field value	

测试结果

field数量	TairHash占用内存	原生Redis hash占用内存
10000	2.39 MB	2.02 MB
100000	25.33 MB	19.31 MB
1000000	253.29 MB	191.1 MB

从以上测试可以看出，在没有淘汰策略影响的情况下，原生Redis hash与TairHash的内存占用量相近甚至相同。但如果为TairHash field加上过期时间，淘汰策略就会对TairHash的内存占用量产生影响。

TairHash淘汰策略解析

与原生Redis类似，Tair性能增强系列的TairHash结构也支持两种淘汰策略：主动淘汰（Active Expiration）和被动淘汰（Passive Expiration）。

- 主动淘汰
  TairHash的被动淘汰策略与原生Redis有所不同：
  - 原生Redis会运行周期性任务，每次在设置了过期时间的key中随机选择一个，检查其是否过期，如果过期则将其淘汰，如果没有过期则不淘汰，效率比较低下。
  - Tair性能增强系列也会通过周期性任务来检查所有设置了过期时间的TairHash field，如果该field已经过期，则将其删除并释放其占用的内存，但Tair性能增强系列使用最小堆（min-heap）将TairHash field按过期时间全排序，每次选取的一定是过期时间最小的field。这种主动淘汰方式的优点是效率比原生Redis高，缺点是会产生额外的内存消耗，详细说明请参见主动淘汰策略内存消耗分析。
- 被动淘汰
  TairHash的被动淘汰策略与原生Redis相似，客户端访问已过期的TairHash field时，会触发被动淘汰策略，删除该field并释放内存。TairHash被动淘汰流程举例如下：
  i. 一个TairHash field刚刚过期，尚未被主动淘汰，此时客户端使用**EXHGET**命令来获取该field。
  ii. Redis实例判断这个field是否过期。
  iii. 由于该field已经过期，Redis实例会直接将其删除，并释放其占有的内存，然后向客户端返回一个空值。

  被动淘汰优点是几乎没有额外的内存消耗，缺点是如果一个field在过期后一直没有被访问，那么其占用的内存将一直得不到释放。

两种淘汰策略的生效规则请参见淘汰策略生效规则。在实际业务中，建议结合使用主动淘汰和被动淘汰两种策略，详细说明请参见最佳实践。

淘汰策略生效规则

TairHash淘汰策略生效规则如下：

- 默认同时开启主动淘汰和被动淘汰。
- 被动淘汰策略对所有设置了过期时间的TairHash field生效。
- 主动淘汰策略对每个field单独生效。使用可以为TairHash设置过期时间的命令时，添加*NOACTIVE*选项即可取消对目标field的主动淘汰。

> **说明** TairHash命令列表及详细使用说明请参见TairHash命令。

由此可见，如果在设置过期时间时不添加*NOACTIVE*选项，主动淘汰策略就会生效，产生额外的内存消耗。这些额外内存消耗的详细说明请参见主动淘汰策略内存消耗分析。

主动淘汰策略内存消耗分析

为了提高主动淘汰效率，Tair性能增强系列内部使用一个min-heap（最小堆）将field按过期时间进行全排序并建立索引。每当一个TairHash field被设置了过期时间，实例就会创建一个最小堆节点（heap node）并将其加入到min-heap中，heap node中保存着对key和field的索引，以便在heap node过期时找到对应的TairHash和field，从而将其删除并释放内存。由于heap node增加了对key和field的引用，key和field占用的内存在命令执行完时无法被立即释放，因此，相对于单纯使用被动淘汰策略，使用主动淘汰策略时多出了如下内存消耗：

- 所有heap node本身占用的内存

- key占用的内存
- 所有带有过期时间的field本身占用的内存

您可以通过下方的对比测试直观地了解不同淘汰策略下TairHash的内存占用情况。

不同淘汰策略下TairHash占用内存的对比测试

- 测试一
  测试环境

测试对象	测试命令	测试条件
无过期时间的TairHash	`EXHSET tairhashkey field value`	◦ field大小：1 KB。 ◦ value大小：1 KB。 ◦ 关闭AOF与RDB备份。 ◦ 实例为单节点，无副本。
有过期时间的TairHash	`EXHSET tairhashkey field value EX 1000`	

  测试结果

field数量	无过期时间的TairHash占用内存	有过期时间的TairHash占用内存
10000	29.79 MB	46.03 MB
100000	297.02 MB	460.36 MB
1000000	2.9 GB	4.6 GB

- 测试二
  测试环境

测试对象	测试命令	测试条件
无过期时间的TairHash	`EXHSET tairhashkey field value`	◦ field大小：64 Byte。 ◦ value大小：64 Byte。 ◦ 关闭AOF与RDB备份。 ◦ 实例为单节点，无副本。
有过期时间的TairHash	`EXHSET tairhashkey field value EX 1000`	

  测试结果

field数量	TairHash占用内存	原生Redis hash占用内存
10000	2.39 MB	4.38 MB
100000	25.33 MB	45.16 MB
1000000	253.29 MB	451.66 MB

- 测试三
  测试环境

测试对象	测试命令	测试条件
无过期时间的TairHash	EXHSET tairhashkey field value	○ field大小：64 Byte。 ○ value大小：64 Byte。 ○ 关闭AOF与RDB备份。 ○ 实例为单节点，无副本。 ○ 设置过期时间时通过*NOACTIVE*选项关闭主动淘汰。
有过期时间的TairHash	EXHSET tairhashkey field value EX 1000 NOACTIVE	

测试结果

field数量	TairHash占用内存	原生Redis hash占用内存
10000	2.39 MB	2.39 MB
100000	25.33 MB	25.33 MB
1000000	253.29 MB	253.29 MB

最佳实践

TairHash为以下命令提供了*NOACTIVE*选项：

- EXHSET
- EXHEXPIRE
- EXHEXPIREAT
- EXHPEXPIRE
- EXHPEXPIREAT
- EXHINCRBY
- EXHINCRBYFLOAT

从淘汰策略生效规则可知，在使用上述命令为TairHash field设置过期时间时，不设置*NOACTIVE*选项就会激活主动淘汰。如果设置了*NOACTIVE*，则主动淘汰策略不会应用于目标field，也不会产生额外的内存消耗，但缺陷也很明显：field过期后，只要没有被访问，占用的内存空间就不会被释放。

> **注意** 使用EXHGETALL、EXHKEYS、EXHVALS和EXHSCAN命令访问TairHash不触发被动淘汰，不会删除过期field并释放内存。为了尽可能快地响应客户端，降低产生慢查询的风险，这些命令只会把过期的field从返回结果中过滤掉。

建议您根据业务特点决定是否使用主动淘汰策略，或者对哪些数据使用主动淘汰策略。例如：

- 在数据过期时间短、数据量增长快的场景，推荐尽量采用主动淘汰策略，在数据过期后快速将其淘汰，释放内存给新的数据。
- 在数据过期时间长、数据量增长较慢的场景，可以少采用或者不采用主动淘汰策略，这样反而能降低总体内存消耗。
- 在冷热数据分界清晰的场景，您还可以对访问频率较低的数据采用主动淘汰策略，提高内存释放效率，而在写入热点数据时通过*NOACTIVE*选项取消主动淘汰，兼顾淘汰效率与内存消耗。

2.通用最佳实践

2.1. 使用云数据库Redis版实现即时通信场景中的多端同步

随着即时通信（Instant Messaging）场景和客户端种类的不断丰富，多端通信已经成为普遍趋势。本文为您介绍一种使用云数据库Redis版在IM场景中实现多端同步的方案。

消息存储模型

通常，IM系统的核心架构分为三个部分：消息管理模块、消息同步模块、通知模块。这三个模块的作用如下：

- 消息管理模块主要负责接收和存储消息。
- 消息同步模块主要负责存储和推送下行消息数据及其状态。
- 通知模块主要负责维护第三方通道和通知功能。

消息管理模块的核心是消息存储模型，存储模型的选型直接影响着消息同步模块的实现。消息、会话、会话与消息组织关系的实现方式在业界各主流IM系统中都不尽相同，但无外乎两种形式：写扩散读聚合、读扩散写聚合。读、写扩散是消息在群组会话中的存储形式，其详细说明如下。

- 在读扩散场景中，消息归属于会话，相当于数据库中存储着一张conversation_message表，其中包含该会话产生的所有消息。这种存储形式的好处是消息入库效率高，只保存会话与消息的绑定关系即可。
- 在写扩散场景中，会话产生的消息投递到message_inbox表中，该表类似于个人邮件的收件箱，其中保存着个人的所有会话，会话中的消息按其产生的时间顺序排列。这种存储形式的好处是能实现灵活的消息状态管理，会话中的每条消息在面向不同的接收者时可以呈现出不同的状态。

如果采用读扩散的方式，在大并发修改数据的场景下，数据一致性处理效率和数据变更效率会成为系统性能瓶颈。因此，下文介绍的案例采用写扩散的方式实现消息存储模型，以更高的存储成本支持更高的更新性能。

消息同步模块

多端同步的核心问题在于多端数据的一致性，IM系统需要记录消息的顺序和每个端的同步点，从而实现消息的最终一致性。

既然采用写扩散的方式来记录消息，系统需要：

- 为每个用户创建一个message_inbox，用于储存该用户的消息。
- 为每一条消息创建一个自增的sync_id，用于记录消息的顺序。
- 记录用户在每个客户端上的同步ID。

通过对用户在各客户端上的数据进行对比和同步，就可以实现多端数据同步，详细的实现方式如下。

1. 提炼数据结构。从IM系统中的各类事件中提炼出统一的消息数据结构，这些事件包括新消息、已读消息、增删会话信息等。消息数据结构示例如下：

```
struct message {
 int type; // 业务类型
 string data; // 业务数据
}
```

2. 进行存储产品选型。选型依据主要有以下两点：
   - 系统需要为message_inbox中的每条消息分配一个自增的sync_id，所以用于存储消息数据的产品需要能实现原子递增队列。
   - 完整的消息需要在IM系统的消息管理模块中保存到持久化存储（例如PolarDB）中，而message_inbox数据则无需持久化存储，只需存储一段时间（例如一周）即可，所以对存储的容量要求并不高。

   支持计数器功能和Sorted Sets结构的云数据库Redis版正好能满足上述要求。
3. 通过云数据库Redis版的Hash结构来存储每个用户在客户端上的同步ID。

场景案例

下图基于一个案例展示了多端同步的详细实现方式。

> **说明** 图中的Bob为虚拟的用户名。

新消息入库以后，推送消息逻辑被触发，系统根据用户名获取到所有客户端设备的当前点位，然后从消息队列中获取历史点位到最新点位间的所有消息，再将其推送到客户端设备。推送完成后，更新设备的当前点位信息。关键步骤的示例代码如下。

1. 新消息入库：

```
sync_id = INCR bob
ZADD bob $sync_id message:{type:new_message, data:"{msgid:991,cid:123,text:"hello"}"}
```

2. 获取消息范围：

```
ZRangeByScore bob 100103 100310
```

3. 获取客户端设备的点位：

```
HGETALL bob
```

4. 加入或更新客户端设备信息：

```
HSET bob dev_1001 100103
HSET bob dev_1002 100202
```

总结

IM通信已经成为互联网环境中最常见通信方式之一，借助云数据库Redis版丰富的数据结构，您可以构建出高可用的IM系统。不仅是本文提到的消息同步模块，IM系统的消息存储模块也可以使用Redis进行加速，最终构建出支持大规模访问的可靠IM系统。

2.2. 使用云数据库Redis版助力在线课堂应用

在线教育已经成为当下的热点行业之一，云数据库Redis版丰富的数据结构可以帮助您快速实现在线课堂应用的相关功能。

背景信息

随着互联网直播的流行，直播已经走向了各行各业，老师使用直播应用进行线上教学也屡见不鲜，尤其是在2020年的新冠肺炎抗疫大作战中，在线课堂成为技术抗疫的重要一环，切实解决了师生无法到校上课的困境。

在线下教学中，师生互动是不可或缺的环节。通过互动，老师能够更好地掌握学生的学习情况，增加课堂的趣味性，学生通过互动能够集中注意力，提升学习效果。为了有更好的线上教学体验，在线课堂也需要师生互动。在线课堂应用可以通过连麦功能，实现在线的师生互动。

在线课堂连麦的一般流程为：

1. 老师发起课堂连麦。
2. 学生请求连麦。
3. 老师统一连麦。
4. 开始连麦互动。
5. 老师结束课堂连麦。

在直播应用中，在线课堂连麦的业务流程图如下所示。

从流程图可以看出，连麦过程中，在线课堂应用需要管理两个队列：申请连麦队列和麦在线队列。

- 申请连麦队列是老师让同学发言时，申请连麦的同学列表。
- 麦在线队列是老师选择连麦同学后，可通过麦克风发言的用户（含老师和同学）列表。

没有连麦时，麦在线队列中只有老师一人。连麦结束后，应用将发言同学从申请连麦队列和麦在线队列中移除。

使用云数据库Redis版的list和hash结构，您可以快速实现有序队列（申请连麦队列）和无序队列（麦在线队列）的管理，还可以在连麦结束后快速地删除相关信息。下文为您介绍具体的实现方案。

实现方案

- 连麦队列
  使用list结构保存连麦队列，方便按时间顺序展示队列：
  - 使用课堂ID作为list的key。
  - 使用学生ID作为list的element。
- 麦在线队列
  麦在线队列主要用于展示已连麦的用户，对顺序没有要求，因此可使用hash结构保存：
  - 使用课堂ID作为hash的key。
  - 使用用户（含学生和老师）ID作为hash的field。
  - 使用详细的连麦信息作为field的value。

示例代码

- 学生提交连麦申请：

```
RPUSH your_class_id studentC_id
RPUSH your_class_id studentA_id
RPUSH your_class_id studentB_id
```

- 展示连麦队列：

```
LRANGE your_class_id 0 MAX_CLASS_NUM
```

- 老师选择其中一个学生或多个学生，同意连麦：

```
HSET your_class_id studentA_id OnlineDetailDO
```

- 老师和其中一个学生完成连麦，挂掉通话：

```
HDEL your_class_id studentA_id
LREM your_class_id 0 studentA_id
```

总结

借助云数据库Redis版丰富的数据结构和优秀的性能，在线课堂应用可以对人员信息进行轻量级的管理，让师生在线上教育场景中获得更好的体验和教学效果。

2.3. 使用Redis在Web应用中实现会话管理

会话（session）管理是Java Web应用不可或缺的功能，使用云数据库Redis版和Spring Session可以便捷地实现会话管理。

前提条件

- 已创建用于保存会话的Redis实例。相关操作，请参见创建实例。
- 如果通过内网连接Redis实例，Redis实例与部署应用的ECS实例需在同一VPC中，或者同属经典网络且在同一地域。
- 已将ECS实例的内网IP地址添加到Redis实例的白名单中。相关操作，请参见设置白名单。

背景信息

浏览网页通常依赖于HTTP协议，但HTTP的特性之一是无状态，即不会保存事务处理进度，在交互场景中没有记忆能力。如果仅有HTTP协议，用户在同一网站浏览两个不同页面时进行的某些交互性操作将毫无意义。例如，访问淘宝时，如果用户同时打开购物车页和商品的详情页，在详情页将若干心仪的商品加入购物车，然后再刷新购物车页，会发现之前加入购物车的商品都没有保存成功。因此，Web应用需要使用会话管理技术，例如Cookie和Session，将用户会话完整保存下来，甚至在分布式服务中共享。

Cookie是客户端技术，将用户信息保存在本地，信息容量受限于用户使用的浏览器，且有一定的安全风险。Session是服务端技术，将用户信息保存在服务端，信息容量可扩展，同时在一定程度上降低了安全风险，弥补了Cookie技术的不足。使用Sping Session与云数据库Redis版可以轻松实现Web应用中的会话管理。

Web应用中的会话管理

Spring Session是近年来较为流行的轻量级框架Spring Boot下的一个项目，提供用户会话信息管理服务和相关API。Spring Session可以用透明的方式集成HttpSession，从而实现Web应用中的会话管理，该方案有以下优势：

- 支持集群会话（Clustered Sessions）
  Spring Session使支持集群会话变得很简单，不必依赖于应用程序容器（例如Tomcat等）提供的特定解决方案。
- 支持多用户会话
  Spring Session支持在单个浏览器实例中管理多个用户的会话，例如谷歌浏览器中多用户同时登录的情况。
- 支持RESTful API

Spring Session允许在标头（header）中提供会话ID以支持RESTful API。

云数据库Redis版在该方案被用于储存用户会话信息，提供高性能的会话信息存取服务。

操作步骤

1. 使用Spring Initializr创建Spring Boot Web项目。默认生成的项目目录如下。

2. 在*pom.xml*中配置Maven依赖。

```
<dependencies>
 <!-- Spring boot --> <dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter</artifactId>
 </dependency>
 <!-- Spring Web -->
 <dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
 </dependency>
 <!-- redis -->
 <dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-redis</artifactId>
 </dependency>
 <!-- Spring session -->
 <dependency>
  <groupId>org.springframework.session</groupId>
  <artifactId>spring-session-data-redis</artifactId>
 </dependency>
</dependencies>
```

3. 在*src/main/resources/application.properties*中配置Redis信息。

> **说明** 此处需要配置云数据库Redis实例的连接地址，您可以根据需要选择内网连接地址或外网连接地址。

```
spring.session.store-type = REDIS
spring.session.redis.flush-mode = on-save
spring.session.redis.namespace = spring：session
spring.redis.host = r-bp1xxxxxxxxxxxxxxx.redis.rds.aliyuncs.com
spring.redis.password = myPassword
spring.redis.port = 6379
```

4. 配置*src/main/java/com.example.demo/DemoApplication.java*，开启Spring Session。

```
package com.example.demo;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.session.data.redis.config.annotation.web.http.EnableRedisHttpSession;

@SpringBootApplication
@EnableRedisHttpSession
public class DemoApplication {

  public static void main(String[] args) {
    SpringApplication.run(DemoApplication.class, args);
  }

}
```

5. 编写业务逻辑代码。下方是使用Controller来处理HTTP请求的示例代码。

```
package com.example.demo.controller;

import javax.servlet.http.HttpServletRequest;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseBody;
import java.util.UUID;

@Controller
@RequestMapping("/")
public class SessionController{
  @RequestMapping(value="/getSessionId")
  @ResponseBody
  public String getSessionId(HttpServletRequest request){
    int port = request.getLocalPort();
    String sessionId = request.getSession().getId();
    String userId = request.getSession().getAttribute("userId").toString();

    return "端口：" + port
        + "<br/>sessionId：" + sessionId
        +"<br/>属性userId："+ userId;
  }
  @RequestMapping(value="/setSessionId")
  @ResponseBody
  public String setSessionId(HttpServletRequest request){
    String userId = UUID.randomUUID().toString().replaceAll("-", "");
    request.getSession().setAttribute("userId", userId);
    return "Session 设置成功.<br />userId：" + userId;
  }
}
```

6. 启动Spring Boot服务。
7. 测试会话设置和读取。
   i. 在浏览器中访问*localhost:8080/setSessionId*。

   ii. 在浏览器中访问*localhost:8080/getSessionId*。

案例总结

只要业务中有会话管理需求，都可以使用云数据库Redis版存储会话信息。Spring Boot等一些当前流行的框架可以帮助您更方便、快捷地集成云Redis，轻松实现分布式会话管理。

2.4. 使用Redis实现多地容灾的会话管理

会话（session）管理是互联网应用的重要功能，当业务在多地部署时，会话管理就有了就近访问和多地容灾的需求，云数据库Redis版可以帮助业务实现高效的会话管理。

背景信息

随着业务规模不断扩大，应用的使用者可能需要在不同的地域使用服务，此时通常需要采用多地容灾架构来部署应用，这样既可以实现就近服务，从而提高用户的访问速度，又能在服务发生单地故障时，通过异地容灾快速恢复正常服务，提高可用性和可靠性。

为了使用户获得较好的跨地域使用体验，应用的会话管理功能同样需要具备多地容灾能力，以下两个场景展示了具有多地容灾会话管理功能的应用给用户带来的优质体验。

- 用户场景一
  用户A在上海注册并登录某应用后，出差到了北京。因为该应用的会话管理功能是异地容灾部署的，用户A在北京尝试使用其服务时，不需要重新登录。
- 用户场景二
  上海用户B最近经常使用某应用，感觉一直很稳定。实际上，该应用在上海地域的会话管理服务器曾在几天前出现过一次故障，期间，应用从北京地域的服务器获取到了其会话信息，因此用户B的使用体验没有受到影响。

下文基于一个案例对如何使用Redis实现多地容灾的会话管理进行了详细说明。

业务案例

- 需求
  - 因用户遍布全国，部署应用服务的地域需要间隔稍远，尽量使全国各地用户在访问业务时都能获得较理想的访问速度。
  - 如果应用服务发生单地故障，尽量不要影响用户的会话，因此需要在多地间同步数据，保持全局会话信息一致。

结合以上需求分析结果制定的业务方案如下：

- 地域选择
  选择上海、北京、河源三个阿里云服务地域，分别覆盖华东区域、华北区域和华南区域，这样也能较好地兼顾其它区域。在这三个地域分别创建云数据库Redis版实例。
- 数据同步
  在业务层实现地域间的数据同步，其优势如下：
  - 灵活性强，可以根据业务中数据的时效性决定采用同步还是异步的方式同步数据。
  - 在进行读操作时，可以通过补偿回写机制避免额外的写同步操作，详情请参见下文的写操作说明。

  具体实现方案请参见下方的写操作和读操作说明。
- 写操作
  当用户在某地创建会话时，应用异步地将数据推送到其它两个地域，其架构图和示例代码如下。

```
IF (OK == Redis_SH-> Set (sessionid, sessionInfo, 600))
  Redis_BJ-> Set(sessionid, sessionInfo, 600)
  Redis_HY-> Set(sessionid, sessionInfo, 600)
```

- 读操作

如果用户获取会话信息的请求因某种原因被发送到了异地的Redis实例，例如上海用户的请求发送到了北京，则优先从当地（北京）的Redis实例读取数据。如果在北京的Redis实例中没有查询到所请求的数据，则返回源地域（上海）的Redis实例中读取数据，然后再将该数据写回到北京的实例。架构图和示例代码如下。

```
IF (localIp match Heyuan)
  sessionInfo = Redis_HY-> GET(sessionid)
  IF sessionInfo == NULL
  sessionInfo = Redis_SH-> GET(sessionid)
```

- 会话信息结构
  为了区分用户写入会话的区域，sessionid生成后，应用会替换其第一个字符。例如，从上海写入的sessionid，其首字母会被替换为s，北京则为b，河源则为h。通过这种方法，应用就能够判断会话源自于哪个地域。
  在sessionid的有效期内，如果上海（源地域）的用户在北京发起了请求，应用会将更新的数据同步地写入用户当前所在地域（北京）和源地域（上海）的Redis实例，再异步写入其它地域（河源）的Redis实例，以这样的方式保证sessionid与其对应信息的一致性。

案例总结

本文介绍的使用Redis实现多地容灾会话管理的方案，不依赖于Redis产品的异地数据同步功能，而是通过业务层实现，具备更好的灵活性，可以满足更多大规模应用的服务需求。

2.5. 将MySQL数据迁移到Redis

使用Redis的管道传输功能，您可以将RDS MySQL或本地MySQL的数据快速迁移到Redis中。使用其它引擎的RDS数据库也可以参照本文的方法将数据迁移到Redis中。

场景介绍

在应用与数据库之间使用Redis作为缓存层，扩展传统关系型数据库的服务能力，从而优化业务的生态体系，是Redis的经典应用场景之一。将业务中的热数据保存到Redis，用户通过应用直接从Redis中快速获取常用数据，或者在交互式应用中使用Redis保存活跃用户的会话，都可以极大地降低后端关系型数据库的负载，提升用户体验。

使用Redis作为缓存首先需要将关系型数据库中的数据传输到Redis中。关系型数据库中库表结构的数据无法直接传入以键值结构保存数据的Redis数据库，迁移前需要将源端数据转换为特定的结构。这篇最佳实践以MySQL向Redis整表迁移为例，介绍如何通过原生工具进行简单高效地迁移。MySQL的表数据将通过Redis Pipeline传输并保存到Redis Hash中。

> **说明** 本文使用阿里云RDS MySQL实例和云数据库Redis版实例作为迁移的源端和目的端，运行迁移命令的Linux环境安装在ECS实例中。三者同在一个VPC，因此可以互通。

您可以用类似的方法将其它关系型数据库中的数据迁移到Redis中。这种从源端数据库提取数据，转换格式后传入异构数据库中的方式也适用于其它异构数据库之间的数据迁移。

前提条件

- 已创建作为源端的RDS MySQL实例且其中已存在可供迁移的表数据。
- 已创建作为目的端的云数据库Redis版实例。
- 已创建Linux系统的ECS实例。
- 以上三个实例在同一地域的同一VPC中。

- RDS MySQL和Redis实例的白名单中已经放通了ECS实例的内网地址。
- ECS中已安装了MySQL和Redis，用于进行数据的提取、转换和传输。

> **说明** 以上前提条件仅在您的环境在阿里云上时适用，如果您要在本地环境使用本文的方法，请确保用于执行迁移的Linux服务器能够连通源端的关系型数据库和目的端的Redis数据库。

迁移前的数据

本文展示的是迁移**custm_info**库中**company**表储存的测试数据的过程。**company**表中包含的测试数据如下所示。

表中共有6列，迁移后，MySQL表中 `id` 列的值将成为Redis中hash的key，其余列的列名将成为hash的field，而列的值则作为field对应的value。您可以根据实际场景调整迁移步骤中的脚本和命令。

迁移步骤

1. 分析源端数据结构，在ECS中创建如下的迁移脚本，保存到名为mysql_to_redis.sql的文件中。

```
SELECT CONCAT(
 "*12\r\n", #这里的12是下方字段的数量，由MySQL表中的数据结构决定
 '$', LENGTH('HMSET'), '\r\n', #HMSET是在Redis中写入数据时使用的命令
 'HMSET', '\r\n',
 '$', LENGTH(id), '\r\n', #id是HMSET字段后的第一个字段，迁移后会成为Redis Hash中的key
 id, '\r\n',
 '$', LENGTH('name'), '\r\n', #'name'将以字符串形式传入hash中，作为其中一个field。下面的'sdate'等与它相同
 'name', '\r\n',
 '$', LENGTH(name), '\r\n', #name是一个变量，代表了MySQL表中公司的名称，迁移后会成为上一参数'name'生成的field所对应的value。下面的sdate等与它相同
 name, '\r\n',
 '$', LENGTH('sdate'), '\r\n',
 'sdate', '\r\n',
 '$', LENGTH(sdate), '\r\n',
 sdate, '\r\n',
 '$', LENGTH('email'), '\r\n',
 'email', '\r\n',
 '$', LENGTH(email), '\r\n',
 email, '\r\n',
 '$', LENGTH('domain'), '\r\n',
 'domain', '\r\n',
 '$', LENGTH(domain), '\r\n',
 domain, '\r\n',
 '$', LENGTH('city'), '\r\n',
 'city', '\r\n',
 '$', LENGTH(city), '\r\n',
 city, '\r\n'
)
FROM company AS c
```

2. 在ECS中使用如下命令迁移数据。

```
mysql -h <MySQL host> -P <MySQL port> -u <MySQL username> -D <MySQL database name> -p --skip-column-names --raw < mysql_to_redis.sql | redis-cli -h <Redis host> --pipe -a <Redis password>
```

选项说明

选项	说明	示例值

选项	说明	示例值
-h	MySQL数据库的连接地址 **说明** 此处是指命令中的第一个 *-h*。	rm-bp1xxxxxxxxxxx.mysql.rds.aliyuncs.com **说明** 请使用Linux服务器连接MySQL数据库所需的地址。
-P	MySQL数据库的服务端口	3306
-u	MySQL数据库的用户名	testuser
-D	需要迁移的MySQL表所在的库	mydatabase
-p	MySQL数据库的连接密码 **说明** - 如无密码则无需设置该选项。 - 为了提高安全性，您可以只输入 *-p*，不在其后输入密码，执行命令后再根据命令行提示输入密码。	Mysqlpwd233
--skip-column-names	不在查询结果中写入列名。	无需设置值。
--raw	输出列的值时不进行转义。	无需设置值。
-h	Redis的连接地址 **说明** 这里指 `redis-cli` 之后的 *-h*。	r-bp1xxxxxxxxxxx.redis.rds.aliyuncs.com **说明** 请使用Linux服务器连接Redis数据库所需的地址。
--pipe	使用Redis的Pipeline功能进行传输。	无需设置值。
-a	Redis的连接密码 **说明** 如无密码或不需要密码则无需设置该选项。	Redispwd233

运行示例

> **说明** 执行结果中的 errors 表示执行过程中的错误数， replies 表示收到的回复数。如果 errors 为0，且 replies 与MySQL表中的记录数相同，则整表迁移成功。

迁移后的数据

迁移完成后，一条MySQL表记录对应一条Redis Hash数据。使用**HGETALL**命令查询一条记录可以看到如下结果。

您可以根据实际场景中需要的查询方式调整迁移方案，例如把MySQL数据中的其它列转换为hash中的key，而把id列转换为field，或者直接省略id列。

2.6. 游戏玩家积分排行榜

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。

点我去体验

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()).in
tValue());
        }
        //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()).in
tValue());
        }
        //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()).in
tValue());
        }
      } 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
```

2.7. 网上商城商品相关性分析

您可以使用云数据库Redis版搭建网上商城的商品相关性分析程序。

场景介绍

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.get
Score()).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
```

2.8. 消息发布与订阅

云数据库Redis版也提供了与Redis相同的消息发布（publish）与订阅（subscribe）功能。即一个客户端发布消息，其他多个客户端订阅消息。

场景介绍

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。

视频介绍

您可以观看以下视频了解Redis发布订阅（Pub/Sub）功能的实现、相关接口、以及应用场景等信息，视频时长约14分钟。

2.9. 管道传输

云数据库Redis版提供了与Redis相同的管道传输（pipeline）机制。

场景介绍

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，例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.getT
ime())+ "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，详情请参见以下代码示例。

```
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);
```

```
            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
```

2.10. 事务处理

云数据库Redis版支持Redis中定义的事务（transaction）机制。

场景介绍

您可以使用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。可以看到不同线程中的两个client在操作同一个key，但是当client 1利用事务机制来操作这个key时，client 2被阻塞不得不等待client 1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
```

2.11. 解密Redis助力双十一背后的技术

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。

背景介绍

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。

2.12. 使用Redis搭建电商秒杀系统

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能支持高达60万以上qps，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
```

1. 秒杀开始前，服务集群读取goodsId_Start为0，直接返回未开始。
2. 数据控制模块将goodsId_start改为1，标志秒杀开始。
3. 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）。
4. 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>EVALSHA 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。

1. 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}
     ...
   }
   ```

2. 将订单内容写入Redis：

   ```
   LPUSH orderList {订单内容}
   ```

3. 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。

2.13. Redis读写分离技术解析

云数据库Redis读写分离版支持多个只读节点，能够为高并发且读多写少的场景提供合适的支持。

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，例如配置可以容忍的最大延迟时间。

2.14. 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。

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。

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。
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ECS个数等） * maxTotal不能超过Redis的最大连接数（可在实例详情页面查看）；
- 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1s/1ms = 1000，而业务期望的单个Redis的QPS是50000（业务总的QPS/Redis分片个数），那么理论上需要的资源池大小（即MaxTotal）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。如果您的业务存在突峰访问，建议设置这两个参数的值相等；如果并发量不大或者maxIdle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 {

  }
}
```

2.15. 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。
   i. 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）。

   ii. 输入 `auth <password>` 进行认证。

   > **说明** `password` 为Redis集群的密码。

   > **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。

   > **说明**
   > imonitor命令与info、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。
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_to_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)
```

```
    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。

> 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。

2.16. 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 message222222222222
(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，示例如下：

```
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

- 查询直播系统中的热点Key请参见查询实时热点Key。
- 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。

2.17. 解析Redis持久化的AOF文件

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，例如：+OKrn。
- 错误信息：
  以-号开头，结尾为rn的字符串，例如：-ERR Readonlyrn。
- 整数：
  以冒号开头，结尾为rn，开头和结尾之间为整数，例如（: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，例如：$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，例如：
  *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']
```

2.18. 广告点击数实时统计（Redis + Spark）

本文将介绍使用Spark StructuredStreaming与Redis Stream实现实时广告点击数统计。

前提条件

- 已创建引擎版本为5.0或以上的Redis实例，创建步骤请参见创建实例。
- 已创建Spark集群，创建步骤请参见分析集群管理。
- Spark集群和Redis实例处于同一VPC，并且可以互相访问。

> **说明** Redis实例和Spark集群均需在白名单中添加对方内网IP地址才能互相访问。
>
> - 设置Redis实例白名单请参见设置白名单
> - 设置Spark集群白名单请参见设置白名单。
> - 如果不清楚Redis/Spark内网IP地址，可以在白名单中添加Redis和Spark所在VPC的IP地址段。

场景介绍

某广告公司在网页上投递动态图片广告，广告的展现形式是根据热点图片动态生成的。为了收益最大化，需要统计广告的点击数来决定哪些广告可以继续投放，哪些广告需要更换。大部分的广告生命周期很短，实时获取广告的点击数可以让我们快速确定哪些广告是关键业务。基于以上诉求可以选择StructuredStreaming + Redis Stream作为解决方案。

- Spark StructuredStreaming是Spark在2.0后推出的基于Spark SQL上的一种实时处理流数据的框架。处理时延可达毫秒级别。
- Redis Stream是在Redis 5.0后引入的一种新的数据结构，可高速收集、存储和分布式处理数据，处理时延可达亚毫秒级别。

> **说明**
>
> - Spark-Redis是X-Pack Spark发布的连接器， 用于连通Spark与Redis，该工具由四个Redis依赖的jar包组成。下载链接与安装方式请参见Spark对接Redis。
> - 如需了解更多Spark相关信息，请参见Spark 基本介绍。
> - 如需了解更多Redis Stream相关信息，请参见Introduction to Redis Streams。

业务数据链路简介

业务数据链路

如上图所示，广告点击数据通过手机或者电脑传递到数据库中进行**数据提取**，提取后的数据经过**数据处理**计算实时的点击数，最后存储到数据库，通过**数据查询**进行统计分析，统计每个广告的点击总数。

根据数据特点，整个数据链路的数据输入输出如下：

- 输入
  针对每个点击事件我们使用asset id以及cost两个字段来表示一个广告信息，例如：

```
asset [asset id] cost [actual cost]
asset aksh1hf98qwdst9q7 cost 39
asset aksh1hf98qwdst9q8 cost 19
```

- 输出
  经过数据处理后，我们把结果集存储到一个数据表中，数据表可以使用SQL查询，例如：

```
select asset, count from clicks order by count desc

asset       count
----------------    -----
aksh1hf98qwdst9q7   2392
aksh1hf98qwdst9q8   2010
aksh1hf98qwdst9q6   1938
```

数据处理流简介

数据处理流

如上图所示，点击数据会存储到Redis Stream，然后StructuredStreaming对数据进行消费以及聚合处理，处理完成后将结果返回Redis，最后通过Spark SQL查询Redis进行统计分析。

处理流	简介
数据提取	Redis Stream是Redis内置的数据结构，具备每秒百万级别的读写能力，存储的数据可以根据时间自动排序。
	Spark-Redis连接器可以将Redis Stream作为数据源，把Redis Stream数据对接到Spark引擎。
数据处理	Spark-Redis连接器可以将获取的Redis Stream数据转换成Spark的DataFrames数据。
	在StructuredStreaming处理流数据的过程中，可以对微批次数据或者整体数据进行查询。
	数据的处理结果可以通过自定义的 `writer` 输出到不同的目的地，本场景中我们直接把数据输出到Redis的Hash数据结构。
数据查询	Spark-Redis连接器可以把Redis的数据结构映射成Spark的DataFrames，我们需要将DataFrames创建成一个临时表，表的字段映射Redis的Hash数据结构，再使用Spark-SQL进行实时的数据查询。

开发步骤

1. Redis Stream存储数据。
   Redis Streams是append-only的数据结构。部署Redis Streams后可以使用redis-cli向Redis发送数据。

   > **说明**
   > - 请使用Redis 5.0以上版本的redis-cli工具。
   > - redis-cli使用方法请参见redis-cli连接。

   向clicks发送点击数据，命令如下所示：

```
XADD clicks MAXLEN ~ 1000000 * asset aksh1hf98qw7tt9q7 cost 29
```

数据提取

2. 数据处理。

数据处理

在StructuredStreaming中把数据处理步骤分成3个子步骤：

i. 从Redis Stream读取、处理数据。

a. 使用Spark-Redis连接器创建一个SparkSession，填写Redis连接信息。

```
val spark = SparkSession
    .builder()
    .appName("StructuredStreaming on Redis")
    .config("spark.redis.host", r-bp1xxxxxxxxxxxx.redis.rds.aliyuncs.com)
    .config("spark.redis.port", 6379)
    .config("spark.redis.auth", redisPassword)
    .getOrCreate()
```

参数	描述	示例
spark.redis.host	Redis的连接地址，查看Redis连接地址请参见查看连接地址。	r-bp1xxxxxxxxxxxx.redis.rds.aliyuncs.com
spark.redis.port	Redis的服务端口，默认为6379。	6379
spark.redis.auth	Redis的连接密码。	redisPassword

b. 在Spark中创建schema。
例如，我们给流数据命名为“clicks”，需要将参数“stream.kes”设置为“clicks”。由于Redis Stream中的数据包含两个字段：“asset”和“cost”，所以我们要创建StructType映射这两个字段。

```
val clicks = spark
    .readStream
    .format("redis")
    .option("stream.keys", redisTableName)
    .schema(StructType(Array(
     StructField("asset", StringType),
     StructField("cost", LongType)
    )))
    .load()
```

c. 统计每个asset的点击次数。

> **说明** 可以创建一个DataFrames根据asset汇聚数据。

```
val bypass = clicks.groupBy("asset").count()
```

d. 启动StructuredStreaming。

```
val query = bypass
    .writeStream
    .outputMode("update")
    .foreach(clickWriter)
    .start()
```

ii. 存储数据到Redis。
使用Redis的Java客户端Jedis连接到Redis，向Redis写数据。

> **说明** 使用Jedis连接的方法请参见Jedis客户端。

```
class ClickForeachWriter(redisHost: String, redisPort: String, redisPassword: String) extends Forea
chWriter[Row] {

 var jedis: Jedis = _

 def connect() = {
  val shardInfo: JedisShardInfo = new JedisShardInfo(redisHost, redisPort.toInt)
  shardInfo.setPassword(redisPassword)
  jedis = new Jedis(shardInfo)
 }

 override def open(partitionId: Long, version: Long): Boolean = {
  true
 }

 override def process(value: Row): Unit = {

  val asset = value.getString(0)
  val count = value.getLong(1)
  if (jedis == null) {
   connect()
  }

  jedis.hset("click:" + asset, "asset", asset)
  jedis.hset("click:" + asset, "count", count.toString)
  jedis.expire("click:" + asset, 300)

 }

 override def close(errorOrNull: Throwable): Unit = {}
}
```

iii. 运行StructuredStreaming程序。
程序完成打包后，可以通过Spark控制台提交任务，运行Spark StructuredStreaming任务。

```
--class com.aliyun.spark.redis.StructuredStremingWithRedisStream
--jars /spark_on_redis/ali-spark-redis-2.3.1-SNAPSHOT_2.3.2-1.0-SNAPSHOT.jar,/spark_on_redis/commons-pool2-2.0.jar,/spark_on_redis/jedis-3.0.0-20181113.105826-9.jar
--driver-memory 1G
--driver-cores 1
--executor-cores 1
--executor-memory 2G
--num-executors 1
--name spark_on_polardb
/spark_on_redis/structuredstreaming-0.0.1-SNAPSHOT.jar
<Host> <Port> <Password> <Stream>
```

参数说明

参数	描述	示例
Host	Redis内网连接地址。	r-bp1xxxxxxxxxxxx.redis.rds.aliyuncs.com
Port	Redis默认服务端口。	6379
Password	Redis的连接密码。	redisPassword
Stream	Redis的Stream名称。	clicks

3. 读取Redis Hash数据库。使用Spark SQL创建表来读取Redis Hash数据库，命令如下所示。

```
CREATE TABLE IF NOT EXISTS clicks(asset STRING, count INT)
USING org.apache.spark.sql.redis
OPTIONS (
'host' 'r-bp1xxxxxxxxxxxx.redis.rds.aliyuncs.com',
'port' '6379',
'auth' 'redisPassword',
'table' 'click'
)
```

参数	描述	示例
host	Redis的内网连接地址，查看Redis连接地址请参见查看连接地址。	r-bp1xxxxxxxxxxxx.redis.rds.aliyuncs.com

参数	描述	示例
port	Redis的服务端口，默认为6379。	6379
auth	Redis的连接密码。	redisPassword
table	Redis的Hash表名称。	click

4. 执行查询语句。查询clicks的点击数据，查询命令如下所示：

```
select * from clicks;
```

执行查询语句

Spark SQL通过Spark-Redis连接器直接查询Redis数据，统计了广告的点击数。

2.19. 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。

> **说明** 云数据库Redis版已支持通过审计日志直接查询热点key，可以帮助您更方便、精准地查询到Redis服务中的热点key，详情请参见查询历史热点Key。

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。

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说明及设置方法。

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。

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：

执行结果的 summary部分即是分析得出的热点Key。

2.20. Redis内存分析方法

Redis中的数据结构对服务的性能有着举足轻重的影响，如果大key较多，容易形成性能瓶颈，甚至降低业务稳定性。定期分析内存并根据分析结果优化内存，可以保持服务的稳定和高效。为了不影响线上Redis服务的运行，您可以使用**BGSAVE**命令生成RDB文件，再结合redis-rdb-tools和SQLite进行静态分析。

前提条件

- 已创建Linux系统的ECS实例。
- ECS中已安装了SQLite。

> 说明
> - 如果云数据库Redis版实例的版本为4.0或以上，您可以使用Redis控制台的缓存分析功能直接进行缓存分析并查看大key，详情请参见缓存分析。
> - 云数据库Redis版控制台的缓存分析功能不支持2.8版本的实例，如需分析2.8版本实例的内存数据，请使用本文介绍的方法。

创建备份文件

- 云数据库Redis版可以在控制台上备份数据和下载RDB文件，详细步骤请参见手动备份（立即备份）和备份存档。
- 自建Redis可在客户端执行**BGSAVE**生成RDB文件。

redis-rdb-tools简介

在获取到备份文件后，需要使用redis-rdb-tools生成内存快照。redis-rdb-tools是一个基于Python的RDB文件解析工具，主要有以下三个功能：

- 生成内存快照；
- 将RDB文件中的数据转换为JSON格式；
- 对比两个RDB文件，发现差异。

安装redis-rdb-tools

您可以在以下两种方式中任选其一安装redis-rdb-tools：

- 在ECS中使用PYPI安装：

```
pip install rdbtools
```

- 在ECS中使用源码安装：

```
git clone https://github.com/sripathikrishnan/redis-rdb-tools
cd redis-rdb-tools
sudo python setup.py install
```

使用redis-rdb-tools生成快照

在ECS中使用如下命令生成CSV格式的快照：

```
rdb -c memory dump.rdb > memory.csv
```

生成的快照包含如下几列的数据：

- 数据库ID；
- 数据类型；
- key；
- 内存使用量（Byte），包含key、value和其它值的容量；

> **说明** 内存使用量是理论上的近似值，一般略低于实际值。

- 编码。

生成的CSV文件示例如下：

```
$head memory.csv
database,type,key,size_in_bytes,encoding,num_elements,len_largest_element
0,string,"orderAt:377671748",96,string,8,8,
0,string,"orderAt:413052773",96,string,8,8,
0,sortedset,"Artical:Comments:7386",81740,skiplist,479,41,
0,sortedset,"pay:id:18029",2443,ziplist,84,16,
0,string,"orderAt:452389458",96,string,8,8
```

在CSV文件生成后，您需要将每行末尾的逗号（,）删除，以便进行下一步的导入。

将CSV文件导入SQLite数据库

SQLite是一款轻型的关系型数据库。将CSV文件中的数据导入到SQLite数据库中之后，即可使用SQL语句分析Redis的内存数据。

> **说明**
>
> - SQLite版本必须是3.16.0以上。
> - 导入前请通过批量编辑或其它方式删除CSV文件中每行末尾的逗号（,）。

使用如下命令导入数据：

```
sqlite3 memory.db
sqlite> create table memory(database int,type varchar(128),key varchar(128),size_in_bytes int,encoding var
char(128),num_elements int,len_largest_element varchar(128));
sqlite>.mode csv memory
sqlite>.import memory.csv memory
```

使用redis-rdb-tools分析内存快照

将数据导入SQLite数据库后，可根据需要使用SQL语句进行分析，部分分析方式的示例如下：

- 查询内存中的key总数：

  ```
  sqlite>select count(*) from memory;
  ```

- 查询内存占用总量：

  ```
  sqlite>select sum(size_in_bytes) from memory;
  ```

- 查询内存占用量最高的10个key：

  ```
  sqlite>select * from memory order by size_in_bytes desc limit 10;
  ```

- 查询元素数量1000以上的list：

  ```
  sqlite>select * from memory where type='list' and num_elements > 1000;
  ```