ALIBABA CLOUD

阿里云

云数据库 Redis 版

最佳实践

文档版本：20220601

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.云数据库Redis开发运维规范

云数据库Redis拥有极强的性能，阿里云结合多年的运维经验，从业务部署、Key的设计、SDK、命令、运维管理等维度展示云数据库Redis开发运维规范，为您设计高效的业务系统提供参考，帮助您充分发挥Redis的能力。

了解Redis性能边界

Redis性能边界

资源类别	说明
计算资源	使用通配符、Lua并发、1对多的PUBSUB、热点Key等会大量消耗计算资源，集群架构下还会导致访问倾斜，无法有效利用所有数据分片。
存储资源	Streaming慢消费、大Key等会占用大量存储资源，集群架构下还会导致数据倾斜，无法有效利用所有数据分片。
网络资源	扫描全库（KEYS命令）、大Value、大Key的范围查询（如HGETALL命令）等会消耗大量的网络资源，且极易引发线程阻塞。 **注意** Redis的高并发能力不等同于高吞吐能力，例如将大Value存在Redis里以期望提升访问性能，此类场景往往不会有特别大的收益，反而会影响Redis整体的服务能力。

集群架构下，热点Key、大Key或大Value等还会引发，在生产环境中，您应当避免触达Redis的性能边界。下文从业务部署、Key的设计、SDK、命令、运维管理等维度展示云数据库Redis开发运维规范，为您设计高效的业务系统提供参考，帮助您充分发挥Redis的能力。存储或访问倾斜

- 业务部署规范
- Key设计规范
- SDK使用规范
- 命令使用规范

- 运维管理规范

业务部署规范

重要程度	规范	说明
★★★★★	确定使用场景为或。高速缓存内存数据库	• 高速缓存：建议关闭AOF以降低开销，同时，由于数据可能会被淘汰，业务设计上避免强依赖缓存中的数据。例如Redis被写满后，会触发数据淘汰策略以挪移出空间给新的数据写入，根据业务的写入量会相应地导致延迟升高。 **注意** 如需使用通过数据闪回按时间点恢复数据功能，AOF功能需保持开启状态。 • 内存数据库：应选购企业版（持久内存型），支持命令级持久化，同时应通过监控报警关注内存使用率。具体操作，请参见报警设置。
★★★★★	就近部署业务，例如将业务部署在同一个专有网络VPC下的ECS实例中。	Redis具备极强的性能，如果部署位置过远（例如业务服务器与Redis实例通过公网连接），网络延迟将极大影响读写性能。 **说明** 针对多地部署应用的场景，您可以通过全球多活功能，借助其提供的跨域复制（Geo-replication）能力，快速实现数据异地灾备和多活，降低网络延迟和业务设计的复杂度。更多信息，请参见Redis全球多活简介。
★★★★☆	为每个业务提供单独的Redis实例。	避免业务混用，尤其需要避免将同一Redis实例同时用作高速缓存和内存数据库业务。带来的影响例如针对某个业务淘汰策略设置、产生的慢请求或执行**FLUSHDB**命令影响将扩散至其他业务。
★★★★☆	设置合理的过期淘汰策略。	云数据库Redis默认的默认逐出策略为，关于各逐出策略的说明，请参见参数支持。volatile-lru
★★★☆☆	合理控制压测的数据和压测时间。	云数据库Redis不会对您压测的数据执行自动删除操作，您需要自行控制压测数据的数据量和压测时间，避免对业务造成影响。

Key设计规范

重要程度	规范	说明
★★★★★	设计合理的Key中Value的大小，推荐小于10 KB。	过大的Value会引发数据倾斜、热点Key、实例流量或CPU性能被占满等问题，应从设计源头上避免此类问题带来的影响。

重要程度	规范	说明

★★★★★	设计合理的Key名称与长度。	• Key名称：
		  ○ 使用可读字符串作为Key名，如果使用Key名拼接库、表和字段名时，推荐使用英文冒号（:）分隔。例如 `project:user:001` 。
		  ○ 在能完整描述业务的前提下，尽量简化Key名的长度，例如 `username` 可简化为 `u` 。
		  ○ 由于大括号（{}）为Redis的hash tag语义，如果使用的是集群架构的实例，Key名称需要正确地使用大括号避免，更多信息，请参见keys-hash-tags。
		  说明 集群架构下执行同时操作多个Key的命令时（例如RENAME命令），如果被操作的Key未使用hash tag让其处于相同的数据分片，则命令无法正常执行。
		  引发数据倾斜
		• 长度：推荐Key名的长度不超过128字节（越短越好）。

★★★★★	对于支持子Key的复杂数据结构，应避免一个Key中包含过多的子Key（推荐低于1,000）。 **说明** 常见的复杂数据结构例如Hash、Set、Zset、Geo、Stream及企业版（性能增强型）特有的TairHash、TairBloom、TairGIS等。	由于某些命令（例如HGETALL）的时间复杂度直接与Key中的子Key数量相关。如果频繁执行时间复杂度为O(N)及以上的命令，且Key中的子Key数量过多容易引发慢请求、数据倾斜或热点Key问题。
★★★★☆	推荐使用串行化方法将Value转变为可读的结构。	由于编程语言的字节码随着版本可能会变化，如果存储裸对象（例如Java Object、C#对象）会导致整个软件栈升级困难，推荐使用串行化方法将Value变成可读的结构。

SDK使用规范

重要程度	规范	说明

重要程度	规范	说明
★★★★★	推荐使用JedisPool或者JedisCluster连接实例。 **说明** 企业版（性能增强型）实例推荐使用TairJedis客户端，支持新数据结构的封装类。使用方法，请参见TairJedis客户端。	如果使用单连接的方式，一旦遇到单次超时则无法自动恢复。关于JedisPool的连接方法，请参见Jedis客户端、JedisPool资源池优化和JedisCluster。
★★★★☆	避免使用Lettuce客户端。	由于Lettuce客户端在请求多次超时后，不再发起自动重连，当云数据库Redis触发高可用导致代理或者数据分片发生切换时，可能出现连接超时导致无法重连。为避免此类风险，推荐您使用Jedis客户端。
★★★★☆	程序客户端需要对超时和慢请求做容错处理。	由于Redis服务可能因网络波动或资源占满引发超时或慢请求，您需要在程序客户端上设计合理的容错机制。
★★★★☆	程序客户端应设置相对宽松的超时重试时间。	如果超时重试时间设置的非常短（例如200毫秒以下），可能引发重试风暴，极易引发业务层雪崩。更多信息，请参见Redis客户端重试指南。

命令使用规范

重要程度	规范	说明
★★★★★	避免执行范围查询（例如KEYS *），使用多次单点查询或SCAN命令来获取延迟优势。	执行范围查询可能导致服务发生抖动、引发慢请求或产生阻塞。
★★★★★	推荐使用扩展数据结构（数据结构模块集成）实现复杂功能，避免使用Lua脚本。	Lua脚本会占用较多的计算和内存资源，且无法被多线程加速，过于复杂或不合理的Lua脚本可能导致资源被占满的情况。
★★★★☆	合理使用管道（pipeline）降低链路的往返时延RTT（Round-trip time）。	如果有多个操作命令需要被迅速提交至服务器端，且客户端不依赖每个操作返回的结果，那么可以通过管道来作为优化性能的批处理工具，注意事项如下： • 管道执行期间客户端将独占与服务器端的连接，推荐为管道单独建立一个连接，将其与常规操作分离。 • 每个管道应包含合理的命令数量（不超过100个）。

重要程度	规范	说明
★★★★☆	正确使用Transaction命令族。	使用事务（Transaction）时，需要注意其限制： • 事务本身没有回滚条件。 • 对于集群架构的实例，需要使用hash tag确保命令所要操作的Key都分布在1个Hash槽中，同时还需要避免hash tag带来的存储倾斜问题。 • 避免在Lua脚本中封装事务命令，可能因编译加载消耗较多的计算资源。
★★★★☆	避免使用Pub和Sub命令族执行大量的消息分发工作。	由于Pub和Sub不支持数据持久化，且不支持ACK应答机制无法实现数据可靠性，当执行大量消息分发工作时（例如订阅客户端数量超过100且Value超过1 KB），订阅客户端可能因服务端资源被占满而无法接收到数据。 **说明** 为提升性能和均衡性，云数据库Redis对Pub和Sub类命令进行了优化，集群架构下，代理节点会根据channel name进行Hash计算，并分配至对应数据节点。

运维管理规范

重要程度	规范	说明
★★★★★	充分了解不同的实例管理操作带来的影响。	在对Redis实例执行变更配置、重启等操作时，实例的状态将发生变化并产生某些影响（例如产生秒级的连接闪断），在操作前您需要充分了解。更多信息，请参见实例状态与影响。
★★★★★	验证客户端程序的差错处理能力或容灾逻辑。	云数据库Redis支持节点健康状态监测，当监测到实例中的主节点不可用时，会自动触发主备切换，保障实例的高可用性。在客户端程序正式上线前，推荐手动触发主备切换，可帮助您验证客户端程序的差错处理能力或容灾逻辑。具体操作，请参见手动执行主备切换。
★★★★★	禁用高耗时或高风险的命令。	生产环境中，无限制地使用命令可能带来诸多问题，例如执行**FLUSHALL**会直接清空全部数据；执行**KEYS**会阻塞Redis服务。为保障业务稳定、高效率地运行，您可以根据实际情况禁用特定的命令，具体操作，请参见禁用高风险命令。
★★★★☆	及时处理阿里云发起的计划内运维操作（即待处理事件）	为提供更优质的服务，持续提升产品性能和稳定性，阿里云会不定期地发起计划内运维操作（即待处理事件），对部分实例所属的机器执行软硬件或网络换代升级（例如数据库小版本升级）。当您收到来自阿里云的事件通知后，您可以查看本次事件的影响，根据业务需求评估是否需要调整执行时间。更多信息，请参见查看并管理待处理事件。
★★★★☆	为核心指标配置监控报警，帮助掌握实例运行状态。	为CPU使用率、内存使用率、带宽使用率等核心指标配置监控报警，实时掌握实例运行状态。具体操作，请参见报警设置。

重要程度	规范	说明
★★★★☆	通过云数据库Redis提供的丰富的运维功能，定期检查实例状态或辅助排查资源消耗异常问题。	• 分析慢日志：帮助您快速找到慢请求问题发生的位置，定位发出请求的客户端IP，为彻底解决超时问题提供可靠的依据。 • 查看监控数据：云数据库Redis支持丰富的性能监控指标，帮助您掌握Redis服务的运行状况和进行问题溯源。 • 发起实例诊断：帮助您从性能水位、访问倾斜情况、慢日志等多方面评估实例的健康状况，快速定位实例的异常情况。 • 发起缓存分析：帮助您快速发现实例中的大Key，帮助您掌握Key在内存中的占用和分布、Key过期时间等信息。 • 实时Top Key统计：帮助您快速发现实例中的热点Key，为进一步的优化提供数据支持。
★★★☆☆	评估并开启审计日志功能。	开通审计日志功能后，可记录写操作的审计信息，为您提供日志的查询、在线分析、导出等功能，助您时刻掌握产品安全及性能情况。更多信息，请参见开通新版审计日志。 **注意** 开通审计日志后，视写入量或审计量可能会对Redis实例造成5%～15%的性能损失。如果业务对Redis实例的写入量非常大，建议仅在运维需要（例如故障排查）期间开通审计功能，以免带来性能损失。

2.Redis客户端重试指南

由于受网络和运行环境的影响，应用程序可能会遇到暂时性的故障，例如瞬时的网络抖动、服务暂时不可用、服务繁忙导致超时等。通过设计自动重试机制可以大幅避免此类故障，保障操作的成功执行。

引发暂时性故障的原因

原因	说明
故障触发了高可用机制	云数据库Redis支持节点健康状态监测，当监测到实例中的主节点不可用时，会自动触发主备切换，例如将主节点和从节点进行互换，保障实例的高可用性。此时，客户端可能会遇到下列暂时性故障： • 秒级的连接闪断。 • 30秒内的只读状态（用于避免主备切换引起潜在的数据丢失风险和双写）。 说明 更多信息，请参见主备切换的原因和影响。
慢查询引起了请求堵塞	执行时间复杂度为O(N)的操作，引发慢查询和请求的堵塞，此时，客户端发起的其他请求可能出现暂时性失败。
复杂的网络环境	由于客户端与Redis服务器之间复杂网络环境引起，可能出现偶发的网络抖动、数据重传等问题，此时，客户端发起的请求可能会出现暂时性失败。

推荐的重试准则

重试准则	说明
仅重试幂等的操作	由于超时可能发生在下述任一阶段： • 该命令由客户端发送成功，但尚未到达Redis。 • 命令到达Redis，但执行超时。 • 命令在Redis中执行结束，但结果返回给客户端时发生超时。 如果执行重试可能导致某个操作在Redis中被重复执行，因此不是所有操作均适合设计重试机制。通常推荐仅重试幂等的操作，例如SET操作，即多次执行SET a b命令，那么a的值只可能是b或执行失败；如果执行LPUSH mylist a则不是幂等的，可能导致mylist中包含多个a元素。
适当的重试次数与间隔	根据业务需求和实际场景调整适当的重试次数与间隔，否则可能引发下述问题： • 如果重试次数不足或间隔太长，应用程序可能无法完成操作而导致失败。 • 如果重试次数过大或间隔过短，应用程序可能会占用过多的系统资源，且可能因请求过多而堵塞在服务器上无法恢复。 常见的重试间隔方式包括立即重试、固定时间重试、指数增加时间重试、随机时间重试等。
避免重试嵌套	避免重试嵌套，否则可能会导致重复的重试且无法停止。
记录重试异常并打印失败报告	在重试过程中，建议在WARN级别上打印重试错误日志，同时，仅在重试失败时打印异常信息。

Jedis客户端

- 在普通JedisPool模式下，Jedis不提供重试功能，推荐使用阿里云的TairJedis（基于Jedis封装），其中封装了Jedis重试类，可快捷实现重试策略。

  > **说明** 如果您的实例为企业版（性能增强型），使用该客户端可以更便捷地使用阿里云自研的数据结构。关于数据结构模块的相关介绍，请参见Tair扩展数据结构的命令。

- 在JedisCluster模式下，您可以配置maxAttempts参数来定义失败时的重试次数（默认值为5）。

重试示例如下：

```
//添加依赖
<dependency>
  <groupId>com.aliyun.tair</groupId>
  <artifactId>alibabacloud-tairjedis-sdk</artifactId>
  <version>填入最新的版本号</version>
</dependency>
//设置key value命令自动重试5次，并且总体重试时间不超过10s，每次重试在类索引之间等待一段时间，如果结束不成功则抛出异常。
int maxRetries = 5; // 最大重试次数
Duration maxTotalRetriesDuration = Duration.ofSeconds(10); // 最大的重试时间，单位为秒
try {
    String ret = new JedisRetryCommand<String>(jedisPool, maxRetries, maxTotalRetriesDurati
on) {
        @Override
        public String execute(Jedis connection) {
            return connection.set("key", "value");
        }
    }.runWithRetries();
} catch (JedisException e) {
     // Indicates that maxRetries attempts have been made or the maximum query time maxTota
lRetriesDuration reached.
    e.printStackTrace();
}
```

Redisson客户端

Redisson客户端提供了两个参数来控制重试逻辑：

- retryAttempts：重试次数，默认为3。
- retryInterval：重试间隔，默认为1,500毫秒。

重试示例如下：

```
Config config = new Config();
config.useSingleServer()
    .setTimeout(1000)
    .setRetryAttempts(3)
    .setRetryInterval(1500) //ms
    .setAddress("redis://127.0.0.1:6379");
RedissonClient connect = Redisson.create(config);
```

StackExchange.Redis

StackExchang.Redis客户端目前仅支持重试时连接，重试示例如下：

```
var conn = ConnectionMultiplexer.Connect("redis0:6380,redis1:6380,connectRetry=3");
```

> **说明** 如需实现API级别的重试策略，请参见Polly。

Lettuce

Lettuce客户端未提供在命令超时后重试的参数，但是您可以通过下述参数来实现命令重试策略：

- at-most-once execution：命令最多执行1次，即0次或1次，如果连接断开并重新连接，命令可能会丢失。
- at-least-once execution（默认）：最少成功执行1次，即可能会在执行时进行多次尝试，保障最少成功执行1次。使用此策略时，如果Redis实例发生了主备切换，此时客户端可能累积了较多的重试命令，主备切换完成后可能会引发Redis实例的CPU使用率激增。

> **说明** 更多信息，请参见Client-Options和Command execution reliability。

重试示例：

```
clientOptions.isAutoReconnect() ? Reliability.AT_LEAST_ONCE : Reliability.AT_MOST_ONCE;
```

相关文档

- 通过客户端程序连接Redis
- 客户端程序SSL加密连接Redis

3.Lua脚本使用规范

云数据库Redis实例支持Lua相关命令，通过Lua脚本可高效地处理CAS（check-and-set）命令，进一步提升Redis的性能，同时可以轻松实现以前较难实现或者不能高效实现的模式。本文介绍通过Redis使用Lua脚本的基本语法与使用规范。

注意事项

数据管理服务DMS控制台目前暂不支持使用Lua脚本等相关命令，请通过客户端或Redis-cli连接Redis实例使用Lua脚本。

基本语法

命令	语法	说明
EVAL	`EVAL script numkeys [key [key ...]] [arg [arg ...]]`	执行给定的脚本和参数，并返回结果。 参数说明： • script：Lua脚本。 • numkeys：指定KEYS[]参数的数量，非负整数。 • KEYS[]：传入的Redis键参数。 • ARGV[]：传入的脚本参数。KEYS[]与ARGV[]的索引均从1开始。 说明 • 与SCRIPT LOAD命令一样，EVAL命令也会将Lua脚本缓存至Redis。 • 混用或滥用KEYS[]与ARGV[]可能会导致Redis产生不符合预期的行为，尤其在集群模式下，详情请参见集群中Lua脚本的限制。
EVALSHA	`EVALSHA sha1 numkeys key [key ...] arg [arg ...]`	给定脚本的SHA1校验和，Redis将再次执行脚本。 使用EVALSHA命令时，若sha1值对应的脚本未缓存至Redis中，Redis会返回NOSCRIPT错误，请通过EVAL或SCRIPT LOAD命令将目标脚本缓存至Redis中后进行重试，详情请参见处理NOSCRIPT错误。
SCRIPT LOAD	`SCRIPT LOAD script`	将给定的script脚本缓存在Redis中，并返回该脚本的SHA1校验和。
SCRIPT EXISTS	`SCRIPT EXISTS script [script ...]`	给定一个（或多个）脚本的SHA1，返回每个SHA1对应的脚本是否已缓存在当前Redis服务。脚本已存在则返回1，不存在则返回0。
SCRIPT KILL	`SCRIPT KILL`	停止正在运行的Lua脚本。
SCRIPT FLUSH	`SCRIPT FLUSH`	清空当前Redis服务器中的所有Lua脚本缓存。

更多关于Redis命令的介绍，请参见Redis官网。

以下为部分命令的示例，本文在执行以下命令前执行了`SET foo value_test`。

- EVAL命令示例：

```
EVAL "return redis.call('GET', KEYS[1])" 1 foo
```

返回示例：

```
"value_test"
```

- SCRIPT LOAD命令示例：

```
SCRIPT LOAD "return redis.call('GET', KEYS[1])"
```

返回示例：

```
"620cd258c2c9c88c9d10db67812ccf663d96bdc6"
```

- EVALSHA命令示例：

```
EVALSHA 620cd258c2c9c88c9d10db67812ccf663d96bdc6 1 foo
```

返回示例：

```
"value_test"
```

- SCRIPT EXISTS命令示例：

```
SCRIPT EXISTS 620cd258c2c9c88c9d10db67812ccf663d96bdc6 ffffffffffffffffffffffffffffffffffffff
ffffff
```

返回示例：

```
1) (integer) 1
2) (integer) 0
```

优化内存、网络开销

现象：

在Redis中缓存了大量功能重复的脚本，占用大量内存空间甚至引发内存溢出（Out of Memory），错误示例如下。

```
EVAL "return redis.call('set', 'k1', 'v1')" 0
EVAL "return redis.call('set', 'k2', 'v2')" 0
```

解决方案：

- 请避免将参数作为常量写在Lua脚本中，以减少内存空间的浪费。

```
# 与错误示例实现相同功能但仅需缓存一次脚本。
EVAL "return redis.call('set', KEYS[1], ARGV[1])" 1 k1 v1
EVAL "return redis.call('set', KEYS[1], ARGV[1])" 1 k2 v2
```

- 更加建议采用如下写法，在减少内存的同时，降低网络开销。

```
SCRIPT LOAD "return redis.call('set', KEYS[1], ARGV[1])"    # 执行后，Redis将返回"55b22c0d0
cedf3866879ce7c854970626dcef0c3"
EVALSHA 55b22c0d0cedf3866879ce7c854970626dcef0c3 1 k1 v1
EVALSHA 55b22c0d0cedf3866879ce7c854970626dcef0c3 1 k2 v2
```

清理Lua脚本的内存占用

现象：

由于Lua脚本缓存将计入Redis的内存使用量中，并会导致used_memory升高，当Redis的内存使用量接近甚至超过maxmemory时，可能引发内存溢出（Out Of Memory），报错示例如下。

```
-OOM command not allowed when used memory > 'maxmemory'.
```

解决方案：

通过客户端执行SCRIPT FLUSH命令清除Lua脚本缓存，但与FLUSHALL不同，SCRIPT FLUSH命令为同步操作。若Redis缓存的Lua脚本过多，SCRIPT FLUSH命令会阻塞Redis较长时间，可能导致实例不可用，请谨慎处理，建议在业务低峰期执行该操作。

> **说明** 在控制台上单击**清除数据**只能清除数据，无法清除Lua脚本缓存。

同时，请避免编写过大的Lua脚本，防止占用过多的内存；避免在Lua脚本中大批量写入数据，否则会导致内存使用急剧升高，甚至造成实例OOM。在业务允许的情况下，建议开启数据逐出（云数据库Redis默认开启，模式为volatile-lru）节省内存空间。但无论是否开启数据逐出，Redis均不会逐出Lua脚本缓存。

处理NOSCRIPT错误

现象：

使用EVALSHA命令时，若sha1值对应的脚本未缓存至Redis中，Redis会返回NOSCRIPT错误，报错示例如下。

```
(error) NOSCRIPT No matching script. Please use EVAL.
```

解决方案：

请通过EVAL命令或SCRIPT LOAD命令将目标脚本缓存至Redis中后进行重试。但由于Redis不保证Lua脚本的持久化、复制能力，Redis在部分场景下仍会清除Lua脚本缓存（例如实例迁移、变配等），这要求您的客户端需具备处理该错误的能力，详情请参见脚本缓存、持久化与复制。

以下为一种处理NOSCRIPT错误的Python Demo示例，该demo利用Lua脚本实现了字符串prepend操作。

> **说明** 您可以考虑通过Python的redis-py解决该类错误，redis-py提供了封装Redis Lua的一些底层逻辑判断（例如NOSCRIPT错误的catch）的Script类。

```
import redis
import hashlib
# strin是一个Lua脚本的字符串，函数以字符串的格式返回strin的sha1值。
def calcSha1(strin):
    sha1_obj = hashlib.sha1()
    sha1_obj.update(strin.encode('utf-8'))
    sha1_val = sha1_obj.hexdigest()
    return sha1_val
class MyRedis(redis.Redis):
    def __init__(self, host="localhost", port=6379, password=None, decode_responses=False):
        redis.Redis.__init__(self, host=host, port=port, password=password, decode_response
s=decode_responses)
    def prepend_inLua(self, key, value):
        script_content = """\
        local suffix = redis.call("get", KEYS[1])
        local prefix = ARGV[1]
        local new_value = prefix..suffix
        return redis.call("set", KEYS[1], new_value)
        """
        script_sha1 = calcSha1(script_content)
        if self.script_exists(script_sha1)[0] == True:      # 检查Redis是否已缓存该脚本。
            return self.evalsha(script_sha1, 1, key, value) # 如果已缓存，则用EVALSHA执行脚本
        else:
            return self.eval(script_content, 1, key, value) # 否则用EVAL执行脚本，注意EVAL有将
脚本缓存到Redis的作用。这里也可以考虑采用SCRIPT LOAD与EVALSHA的方式。
r = MyRedis(host="r-******.redis.rds.aliyuncs.com", password="***:***", port=6379, decode_r
esponses=True)
print(r.prepend_inLua("k", "v"))
print(r.get("k"))
```

处理Lua脚本超时

- 现象：
  由于Lua脚本在Redis中是原子执行的，Lua慢请求可能会导致Redis阻塞。单个Lua脚本阻塞Redis最多5秒，5秒后Redis会给所有其他命令返回如下BUSY error报错，直到脚本执行结束。

  ```
  BUSY Redis is busy running a script. You can only call SCRIPT KILL or SHUTDOWN NOSAVE.
  ```

  解决方案：
  您可以通过SCRIPT KILL命令终止Lua脚本或等待Lua脚本执行结束。

  > **说明**
  > - SCRIPT KILL命令在执行慢Lua脚本的前5秒不会生效（Redis阻塞中）。
  > - 建议您编写Lua脚本时预估脚本的执行时间，同时检查死循环等问题，避免过长时间阻塞Redis导致服务不可用，必要时请拆分Lua脚本。

- 现象：
  若当前Lua脚本已执行写命令，则SCRIPT KILL命令将无法生效，报错示例如下。

```
(error) UNKILLABLE Sorry the script already executed write commands against the dataset.
You can either wait the script termination or kill the server in a hard way using the SHU
TDOWN NOSAVE command.
```

解决方案：
请在控制台的**实例列表**中单击对应实例**重启**，若无法解决，请提交工单处理。

脚本缓存、持久化与复制

现象：

在不重启、不调用SCRIPT FLUSH命令的情况下，Redis会一直缓存执行过的Lua脚本。但在部分情况下（例如实例迁移、变配、版本升级、切换等等），Redis无法保证Lua脚本的持久化，也无法保证Lua脚本能够被同步至其他节点。

解决方案：

由于Redis不保证Lua脚本的持久化、复制能力，请您在本地存储所有Lua脚本，在必要时通过EVAL或SCRIPT LOAD命令将Lua脚本重新缓存至Redis中，避免实例重启、HA切换等操作时Redis中Lua脚本被清空而带来的NOSCRIPT错误。

集群中Lua脚本的限制

Redis Cluster对使用Lua脚本增加了一些限制，云数据库Redis集群版在这个基础上存在如下额外限制：

> **说明** 如果发现无法执行Eval的相关命令，例如提示 `ERR command eval not support for normal user`，请将实例的小版本升级至最新。具体操作，请参见升级小版本。

- 所有key必须在一个slot上，否则返回错误信息：

  ```
  -ERR eval/evalsha command keys must be in same slot\r\n
  ```

  > **说明** 您可以通过CLUSTER KEYSLOT命令获取目标key的哈希槽（hash slot）。

- 对单个节点执行SCRIPT LOAD命令时，不保证将该Lua脚本存入至其他节点中。
- 不支持发布订阅命令，包括**PSUBSCRIBE**、**PUBSUB**、**PUBLISH**、**PUNSUBSCRIBE**、**SUBSCRIBE**和**UNSUBSCRIBE**。
- 不支持UNPACK函数。

若您能够在代码中确保所有操作都在相同slot（如果不能保障这一点，执行会出错），且希望打破Redis集群的Lua限制，可以在控制台将script_check_enable修改为*0*，则后端不会对脚本进行校验，但仍需要使用KEYS数组至少传递一个key，供代理节点执行路由转发。具体操作，请参见设置实例参数。

代理模式的额外限制

- 所有key都应该由KEYS数组来传递，**redis.call/pcall**中调用的Redis命令，key的位置必须是KEYS array（不能使用Lua变量替换KEYS），否则直接返回错误信息：

  ```
  -ERR bad lua script for redis cluster, all the keys that the script uses should be passed
  using the KEYS array\r\n
  ```

  正确与错误命令示例如下：

```
# 本示例的准备工作需执行如下命令
SET foo foo_value
SET {foo}bar bar_value
# 正确示例
EVAL "return redis.call('mget', KEYS[1], KEYS[2])" 2 foo {foo}bar
# 错误示例
EVAL "return redis.call('mget', KEYS[1], '{foo}bar')" 1 foo
EVAL "return redis.call('mget', KEYS[1], ARGV[1])" 1 foo {foo}bar
```

- 调用必须要带有key，否则直接返回错误信息：

```
-ERR for redis cluster, eval/evalsha number of keys can't be negative or zero\r\n
```

正确与错误命令示例如下：

```
# 正确示例
EVAL "return redis.call('get', KEYS[1])" 1 foo
# 错误示例
EVAL "return redis.call('get', 'foo')" 0
```

- 不支持在MULTI、EXEC事务中使用EVAL、EVALSHA、SCRIPT系列命令。

> **说明** 若您需要使用代理模式下受限的部分功能，您可以尝试开通使用云数据库Redis集群版的直连模式。但是由于云数据库Redis集群版在迁移、变配时都会通过proxy代理迁移数据，直连模式下不符合代理模式的Lua脚本会迁移、变配失败。
> 建议您在直连模式下使用Lua脚本时应尽可能符合代理模式下的限制规范，避免后续Lua脚本迁移、变配失败。

4.企业版最佳实践

4.1. 基于TairGIS轻松实现用户轨迹监测和电子围栏

您可以通过Tair的TairGIS结构，轻松实现基于点、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，主要用于点的查询，TairGIS在此基础上还支持线、面的查询，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。

总结

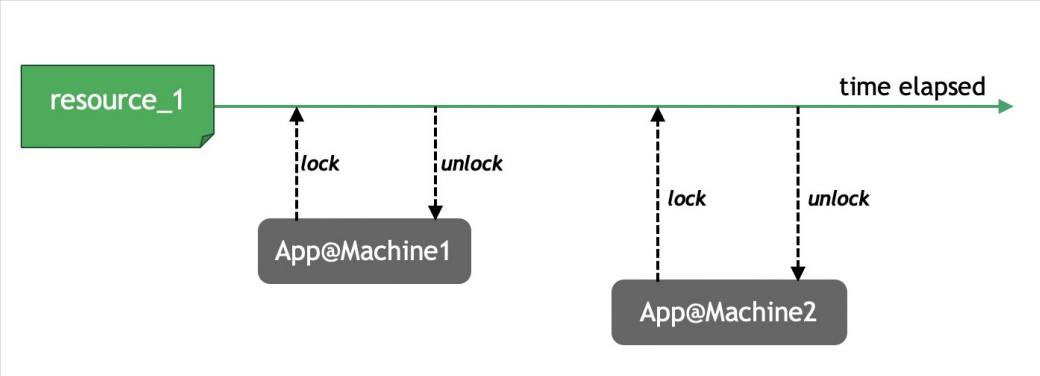
使用云Redis企业版性能增强系列的TairGIS结构，LBS应用可以方便地实现地理信息相关的存储和计算，同时也能满足大并发场景对高性能的需求。

4.2. 基于TairString实现高性能分布式锁

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Tair（Redis企业版）和分布式锁方面的业务经验，介绍使用Tair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
```

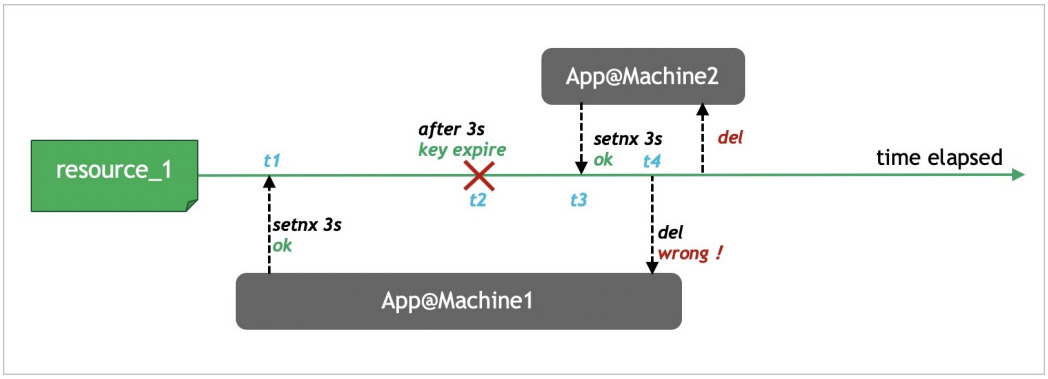
关键选项说明

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。

参数/选项	说明
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Tair实现分布式锁

使用Tair性能增强型或持久内存型实例的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 expireTime) {
    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, i
nt expireTime) {
    jedis.getClient().sendCommand(TairCommand.CAS,resourceKey,randomValue,randomValue,"EX
",String.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` 。Redis节点越多则一致性越强，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，实现成本低。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Tair（Redis企业版）
  - Tair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。
  - Tair性能增强型实例能提供三倍于原生Redis的性能，即使是大并发的分布式锁也不会影响正常的实例服务。

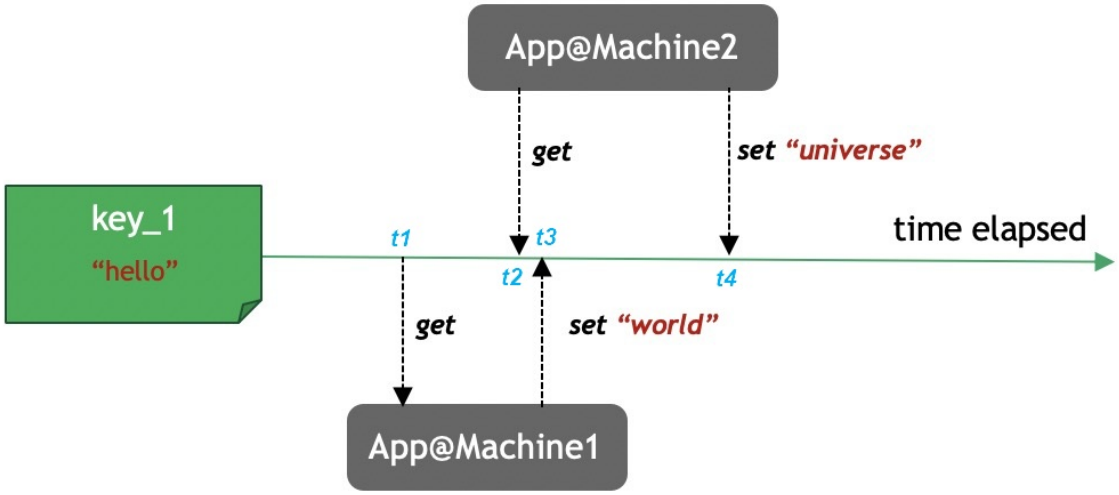
- Tair持久内存型实例使用Intel 傲腾™持久内存，掉电数据不丢失，每个写操作将在持久化成功之后返回，保证了数据的实时持久化。同时，持久内存版型实例还支持配置主备实例间同步方式为半同步，保证写入数据并同步至备节点后，才成功返回客户端（若出现备节点故障、网络异常等情况会降级为异步同步），保证高可用切换后数据不丢失。

4.3. 基于TairString实现高性能乐观锁

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。

> **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val
ue
    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;
 }
```

4.4. 基于TairString实现高效限流器

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（Bounded Counter），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。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。例如配置QPS（Query Per Second）限制，您可以使用TairString**EXINCRBY**命令，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.sendCommand(TairCommand.EXINCRBY,rateLimitor,"1","EX","1","MAX",String.valueO
f(limiter), "KEEPTTL");
        // 设置限流器，EX 1表示1秒之后过期，MAX limiter表示限制最大量为limiter，KEEPTTL表示不修改
已经存在的exstring的过期时间。
        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;
    }
}
```

4.5. 基于TairZset轻松实现多维排行榜

TairZset是阿里云自研的数据结构，可实现256维度的double类型的分值排序。

原生Zset痛点

原生Redis支持的排序结构Sorted Set（也称Zset）只支持1个double类型的分值排序，实现多维度排序时较为困难。例如通过IEEE 754结合拼接的方式实现多维度排序，此类方式存在实现复杂、精度下降、ZINCRBY命令无法使用等局限性。

TairZset介绍

借助阿里云自研的TairZset数据结构，可帮助您轻松实现多维度排序能力，相较于传统方案具有如下优势：

- 最大支持256维的double类型的分值排序（排序优先级为从左往右）。
  对于多维score而言，左边的score优先级大于右边的score，以一个三维score为例：score1#score2#score3，TairZset在比较时，会先比较score1，只有score1相等时才会比较score2，否则就以score1的比较结果作为整个score的比较结果。同样，只有当score2相等时才会比较score3。若所有维度分数都相同，则会按照元素顺序（ascii顺序）进行排序。
  为了方便理解，可以把#想象成小数点（.），例如0#99、99#90和99#99大小关系可以理解为0.99 < 99.90 < 99.99，即0#99 < 99#90 < 99#99。
- 支持EXZINCRBY命令，不再需要取回当前数据，在本地增加值后再拼接写回Redis。
- 支持和原生Zset相似的API。
- 提供和的能力。普通排行榜分布式架构排行榜
- 提供开源TairJedis客户端，无需任何编解码封装，您也可以参考开源自行实现封装其他语言版本。

> **说明** 关于本文中使用的TairZset相关命令，详细解释，请参见TairZset。

应用场景

排序需求常见于各类游戏、应用、奖牌等排行榜中，通常业务对排序的需求如下：

- 支持增删改查和反向排序，可根据分数范围获取相应用户。
- 快速返回排序请求的结果。
- 具备扩展能力（即），在数据分片容量或计算能力不足时，可以将其扩展到其他数据分片。分布式架构排行榜

实现奖牌榜

排名	参与方	金牌	银牌	铜牌
1	A	32	21	16
2	B	25	29	21
3	C	20	7	12
4	D	14	4	16
5	E	13	21	18
6	F	13	17	14

在奖牌榜中，从金、银、铜牌的维度对参与方进行排名，先按照金牌数量排序；如果金牌数量一致，再以银牌数量排序；如果银牌数量也一致，再按照铜牌数量排序。在上述数据中，参与方E和F的金牌数相等，但是银牌数参与方E大于F，因此参与方E排名靠前，通过TairZset的多维排序能力，您只需要使用简单的API即可完成该需求。

客户端设置依赖如下，采用阿里云的TairJedis-SDK。

```
<dependency>
    <groupId>com.aliyun.tair</groupId>
    <artifactId>alibabacloud-tairjedis-sdk</artifactId>
    <version>1.6.0</version>
</dependency>
```

示例代码如下：

```
JedisPool jedisPool = new JedisPool();
// 创建排行榜
LeaderBoard lb = new LeaderBoard("leaderboard", jedisPool, 10, true, false);
// 如果金牌数相同，按照银牌数排序，否则继续按照铜牌
//                      金牌 银牌 铜牌
lb.addMember("A",     32,  21, 16);
lb.addMember("D",     14,  4,  16);
lb.addMember("C",     20,  7,  12);
lb.addMember("B",     25,  29, 21);
lb.addMember("E",     13,  21, 18);
lb.addMember("F",     13,  17,  14);
// 获取 A 的排名
lb.rankFor("A"); // 1
// 获取top3
lb.top(3);
// [{"member":"A","score":"32#21#16","rank":1},
// {"member":"B","score":"25#29#21","rank":2},
// {"member":"C","score":"20#7#12","rank":3}]
// 获取整个排行榜
lb.allLeaders();
// [{"member":"A","score":"32#21#16","rank":1},
// {"member":"B","score":"25#29#21","rank":2},
// {"member":"C","score":"20#7#12","rank":3},
// {"member":"D","score":"14#4#16","rank":4},
// {"member":"E","score":"13#21#18","rank":5},
// {"member":"F","score":"13#17#14","rank":6}]
```

实现实时、小时、日、周和月维度的排行榜

该场景下的需求是实现月榜，那么这个Key就从月的维度进行索引。

利用TairZset的多级索引能力可以轻松实现不同时间范围的排行榜。本案例中，月度的所有数据存储在1个Key中（名称为julyZset），写入演示数据如下：

```
EXZINCRBY julyZset 7#2#6#16#22#100 7#2#6#16#22_user1
EXZINCRBY julyZset 7#2#6#16#22#50 7#2#6#16#22_user2
EXZINCRBY julyZset 7#2#6#16#23#70 7#2#6#16#23_user1
EXZINCRBY julyZset 7#2#6#16#23#80 7#2#6#16#23_user1
```

说明

- `7#2#6#16#22#100` 表示7月第2周6号16点22分，更新其分数为100。
- `7#2#6#16#22_user1` 表示此时间点更新的用户，用户名加入了具体时间前缀。

排行榜类型	具体实现的命令和返回结果

小时级别实时排行榜，即从当前时间往前推算一个小时（例如16:23~15:23）。
说明 如果访问非常频繁，可以将结果进行缓存。
	查询命令：
	`EXZREVRANGEBYSCORE julyZset 7#2#6#16#23#0 7#2#6#15#23#0`
	返回结果：
	`1) "7#2#6#16#22_user1"`
	`2) "7#2#6#16#22_user2"`

固定1小时排行榜，例如查询16:00~17:00时间段的排行榜。	查询命令：
	`EXZREVRANGEBYSCORE julyZset 7#2#6#17#0#0 7#2#6#16#0#0`
	返回结果：
	`1) "7#2#6#16#22_user1"`
	`2) "7#2#6#16#22_user2"`

日排行榜，例如查询7月5号的日排行榜。	在查询前，插入一条7月5号的数据：
	`EXZINCRBY julyZset 7#2#5#10#23#70 7#2#5#10#23_user1`
	返回结果：
	`"7#2#5#10#23#70"`
	查询命令：
	`EXZREVRANGEBYSCORE julyZset 7#2#6#0#0#0 7#2#5#0#0#0`
	返回结果：
	`1) "7#2#5#10#23_user1"`

周排行榜，例如查询7月第2周的排行榜。	查询命令：
	`EXZREVRANGEBYSCORE julyZset 7#3#0#0#0#0 7#2#0#0#0#0`
	返回结果：
	`1) "7#2#6#16#22_user1"`
	`2) "7#2#6#16#22_user2"`
	`3) "7#2#5#10#23_user1"`

排行榜类型	具体实现的命令和返回结果

月排行榜，例如查询7月的月排行榜。	在查询前，插入一条7月20号的数据：
	`EXZINCRBY julyZset 7#4#20#12#20#50 7#4#20#12#20_user1`
	返回结果：
	`"7#4#20#12#20#50"`
	查询命令：
	`EXZREVRANGEBYSCORE julyZset 7#6#0#0#0#0 7#0#0#0#0#0`
	返回结果
	`1) "7#4#20#12#20_user1"`
	`2) "7#2#6#16#22_user1"`
	`3) "7#2#6#16#22_user2"`
	`4) "7#2#5#10#23_user1"`

4.6. 基于TairTS实现秒级监控

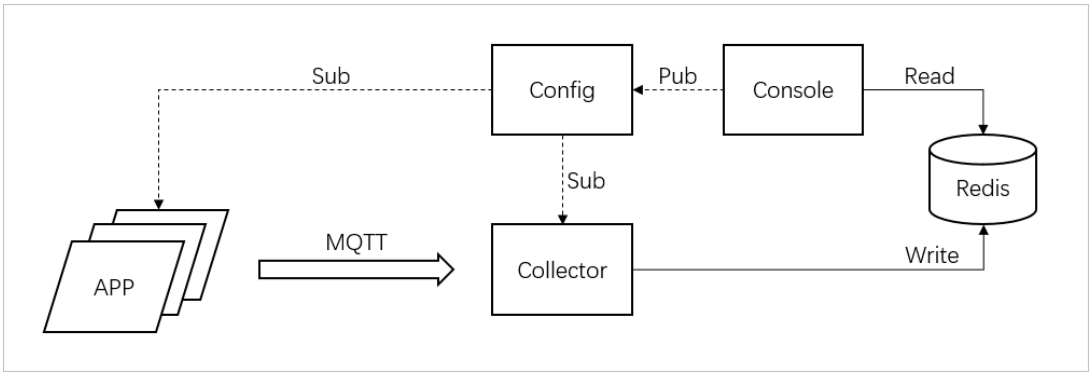
随着不断增长的监控指标与数据流量，监控系统变得越来越复杂，同时对监控系统的时效性提出了更高的要求。本文介绍基于TairTS轻松搭建高并发场景的秒级监控系统。

TairTS简介

TairTS为Tair自研的Module，依托Tair（Redis企业版，采用多线程模型）提供实时且高并发的查询、写入性能，支持对历史时序数据的更新或累加，支持时间线级别的TTL设定，保证每条时间线都可以按时间窗口自动滚动，同时采用高效的Gorilla压缩算法与特定存储，极大降低存储成本。更多信息，请参见TairTS。

秒级监控介绍

秒级监控架构图示例



本文以上图为例介绍秒级监控架构。Console发布秒级监控配置，对应APP接收配置后通过MQTT协议写入到Collector中，Collector处理数据后写入到Redis数据库中。

- 高并发查询场景

在高并发查询场景中，TairTS不仅可以保证查询的性能，还支持降采样、属性过滤、分批查询、多种数值函数等条件下的聚合操作，满足不同业务进行多维度筛选与查看，同时支持将批量查询与聚合计算集成到单条命令中，减少网络交互，实现毫秒级响应，帮助您第一时间定位问题。

- 高并发写入场景
  在高并发写入场景中，随着APP规模的增大，单点的Collector将会成为瓶颈。为此，TairTS支持对历史时序数据的原地更新或累加，保障多Collector并发写入正确性，同时节省内存空间。并发写入的代码示例如下：

```
import com.aliyun.tair.tairts.TairTs;
import com.aliyun.tair.tairts.params.ExtsAggregationParams;
import com.aliyun.tair.tairts.params.ExtsAttributesParams;
import com.aliyun.tair.tairts.results.ExtsSkeyResult;
import redis.clients.jedis.Jedis;
public class test {
    protected static final String HOST = "127.0.0.1";
    protected static final int PORT = 6379;
    public static void main(String[] args) {
        try {
            Jedis jedis = new Jedis(HOST, PORT, 2000 * 100);
            if (!"PONG".equals(jedis.ping())) {
                System.exit(-1);
            }
            TairTs tairTs = new TairTs(jedis);
            //Cluster模式时如下：
            //TairTsCluster tairTsCluster = new TairTsCluster(jedisCluster);
            String pkey = "cpu_load";
            String skey1 = "app1";
            long startTs = (System.currentTimeMillis() - 100000) / 1000 * 1000;
            long endTs = System.currentTimeMillis() / 1000 * 1000;
            String startTsStr = String.valueOf(startTs);
            String endTsStr = String.valueOf(endTs);
            tairTs.extsdel(pkey, skey1);
            long num = 5;
            //Collector A 并发更新。
            for (int i = 0; i < num; i++) {
                double val = i;
                long ts = startTs + i*1000;
                String tsStr = String.valueOf(ts);
                ExtsAttributesParams params = new ExtsAttributesParams();
                params.dataEt(1000000000);
                String addRet = tairTs.extsrawincr(pkey, skey1, tsStr, val, params);
            }
            ExtsAggregationParams paramsAgg = new ExtsAggregationParams();
            paramsAgg.maxCountSize(10);
            paramsAgg.aggAvg(1000);
            System.out.println("Collector A并发更新后结果：");
            ExtsSkeyResult rangeByteRet = tairTs.extsrange(pkey, skey1, startTsStr, endTs
Str, paramsAgg);
            for (int i = 0; i < num; i++) {
                System.out.println("    ts: " + rangeByteRet.getDataPoints().get(i).getTs
() + ", value: " + rangeByteRet.getDataPoints().get(i).getDoubleValue());
            }
            //Collector B并发更新。
            for (int i = 0; i < num; i++) {
```

```
                double val = i;
                long ts = startTs + i*1000;
                String tsStr = String.valueOf(ts);
                ExtsAttributesParams params = new ExtsAttributesParams();
                params.dataEt(1000000000);
                String addRet = tairTs.extsrawincr(pkey, skey1, tsStr, val, params);
            }
            System.out.println("Collector B 并发更新后结果：");
            rangeByteRet = tairTs.extsrange(pkey, skey1, startTsStr, endTsStr, paramsAgg)
;
            for (int i = 0; i < num; i++) {
                System.out.println("    ts: " + rangeByteRet.getDataPoints().get(i).getTs
() + ", value: " + rangeByteRet.getDataPoints().get(i).getDoubleValue());
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

执行结果：

```
Collector A并发更新后结果：
    ts: 1597049266000, value: 0.0
    ts: 1597049267000, value: 1.0
    ts: 1597049268000, value: 2.0
    ts: 1597049269000, value: 3.0
    ts: 1597049270000, value: 4.0
Collector B并发更新后结果：
    ts: 1597049266000, value: 0.0
    ts: 1597049267000, value: 2.0
    ts: 1597049268000, value: 4.0
    ts: 1597049269000, value: 6.0
    ts: 1597049270000, value: 8.0
```

4.7. 基于TairZset实现分布式架构排行榜

TairZset是阿里云自研的数据结构，可实现256维度的double类型的分值排序。借助Tair自研客户端可实现分布式架构排行榜的能力，即可将计算任务分布至多个Key（子排行榜）中完成，您可自定义该Key的数量（默认为10），Tair会将自动数据分散到10个Key中（子排行榜）完成计算，实现分布式架构排行榜 。

背景信息

实现分布式架构排行榜有精确排名法和非精确排名法（线性插值法）两种解决方案。

实现分布式架构排行榜的解决方案

解决方案	说明

解决方案	说明
精确排名法（推荐）	将数据分别分配到在不同的Key上进行计算，查询时，查询目标数据在各Key中的排名并进行汇总。 例如自定义底层为3个Key，创建一个排行榜（Key）并插入3,000个值（成员），Tair会将3,000个值分别分配到3个Key（子排行榜）中。查询时，FindRank(x) 分别在3个Key（子排行榜）中获取到了3个排名：124、183和156，表示x分别在3个Key中排在第124、183和156位，那么x的实际排名为463（即124+183+156）。 • 优势：排名精确。 • 劣势：获取排名的为m*O(log(N))。时间复杂度
线性插值法（TairZset暂未实现）	将数据进行分段，记录每个区间的数量和最高排名（即此区间中的最高排名），对于区间中介于当前区间最低与最高之间的分数，可通过线性插值法进行估算排名。 • 优势：获取排名速度相对较快，时间复杂度为 O(m)。 • 劣势：排名为估算的值，可能存在误差。

本文将使用精确排名法实现分布式架构排行。

> **说明** 关于本文中使用的TairZset相关命令，请参见TairZset。

前提条件

使用Tair自研客户端，更多信息，请参见TairJedis。

实现分布式架构排行榜

在实现同一基本功能时，普通排行榜和分布式架构排行榜的实现方案如下：

基本功能	普通排行榜		分布式架构排行榜	
	实现方案	时间复杂度	实现方案	时间复杂度
插入成员	通过EXZADD插入元素。	O(log(N))	通过 `crc(key) & m` 计算要存放的目标Key，然后通过EXZADD插入元素。	O(log(N))
更新成员分值	通过EXZINCRBY更新分数。	O(log(N))	通过 `crc(key) & m` 计算要存放的目标Key，通过EXZINCRBY更新分数。	O(log(N))
移除成员	通过EXZREM移除成员。	O(M*log(N))	通过 `crc(key) & m` 计算要操作的Key，然后通过EXZREM移除成员。	O(log(N))

基本功能	普通排行榜		分布式架构排行榜	
	实现方案	时间复杂度	实现方案	时间复杂度
查询成员数量	通过EXZCARD查询成员数量。	O(1)	分别通过EXZCARD查询成员数量，然后相加。	O(m) 说明 本列中的m代表分片数
查询总页数	通过EXZCARD查询成员数量，然后除以PAGE_SIZE（每页可展示的记录数）。	O(1)	分别通过EXZCARD查询成员数量，然后相加，得出的结果再除以PAGE_SIZE（每页可展示的记录数）。	O(m)
某分数范围内的总成员数	通过EXZCOUNT查询。	O(log(N))	分别执行EXZCOUNT，然后合并结果。	m*O(log(N))
移除某分数范围内的成员	通过EXZREMRANGEBYSCORE移除。	O(log(N)+M)	分别执行EXZREMRANGEBYSCORE。	m*O(log(N))
获取成员分值	通过EXZSCORE查询。	O(1)	通过 `crc(key) & m` 计算要操作的Key，然后执行EXZSCOR。	O(1)
获取成员排名	通过EXZRANK查询。	O(log(N))	在每一个Key（子排行榜）调用EXZRANKBYSCORE，然后相加。	m*O(log(N))
同时获取成员分值和排名	通过EXZSCORE和EXZRANK查询。	O(log(N))	1. 通过 `crc(key) & m` 计算要操作的Key，然后执行EXZSCORE。 2. 在每一个Key（子排行榜）调用 EXZRANKBYSCORE，然后相加。	m*O(log(N))
查询第top i个成员	通过EXZRANGE查询。	O(log(N)+M)	通过EXZRANGE查询每个top i，然后合并并得出最终结果。	m*O(log(N))
获取排行榜第 i 页	通过EXZRANGE查询。	O(log(N))	将获取页数之前的元素都获取到，然后排序以获得最终的页数。	m*O(log(N))
设置过期时间	通过EXPIRE设置。	O(1)	给每个元素设置过期。	O(m)
删除排行榜	通过DEL删除。	O(N)	同时删除Key中的每个元素。	m * O(N)

示例代码如下：

```
public class DistributedLeaderBoradExample {
    private static final int shardKeySize = 10;  // 底层子排行榜的数量
    private static final int pageSize = 10;      // 排行榜每页包含的个数
    private static final boolean reverse = true; // 本示例为从大到小排序
    private static final boolean useZeroIndexForRank = false; // 本示例以1作为排名起点
    public static void main(String[] args) {
        JedisPool jedisPool = new JedisPool();
        // 创建分布式架构排行榜
        DistributedLeaderBoard dlb = new DistributedLeaderBoard("distributed_leaderboard",
jedisPool,
            shardKeySize, pageSize, reverse, useZeroIndexForRank);
        // 如果金牌数相同，按照银牌数排序，否则继续按照铜牌
        //                      金牌 银牌 铜牌
        dlb.addMember("A",     32,  21, 16);
        dlb.addMember("D",     14,  4,  16);
        dlb.addMember("C",     20,  7,  12);
        dlb.addMember("B",     25,  29, 21);
        dlb.addMember("E",     13,  21, 18);
        dlb.addMember("F",     13,  17,  14);
        // 获取 A 的排名
        dlb.rankFor("A"); // 1
        System.out.println(dlb.rankFor("A"));
        // 获取top3
        dlb.top(3);
        System.out.println(dlb.top(3));
        // [{"member":"A","score":"32#21#16","rank":1},
        // {"member":"B","score":"25#29#21","rank":2},
        // {"member":"C","score":"20#7#12","rank":3}]
    }
```

参数说明：

参数	类型	说明
shardKeySize	int	底层子排行榜的数量，默认为10，无法动态扩容，需要在业务初期做好规划。
pageSize	int	排行榜每页包含的个数，默认为10。
reverse	boolean	取值说明： • false（默认）：按从小到大排序。 • true：按从大到小排序。
useZeroIndexForRank	boolean	取值说明： • true（默认）：以0作为排名起点。 • false：以1作为排名起点。

4.8. 基于TairRoaring实现人群圈选方案

您可以通过Tair（Redis企业版）的TairRoaring功能快速搭建高性能的目标用户筛选服务。

TairRoaring简介

用户标签筛选场景往往应用于个性化推荐、精准营销等具体业务场景，通过不同的标签辅以不同的运营营销，从而实现资源投放方的商业利益最大化。

该类业务通常具备如下特点：

- 用户标签极多，需要较高的存储空间及良好的扩展能力。
- 用户规模大，需按照众多维度添加标签，数据分布较为离散。
- 计算量大，应用会根据不同策略需选取不同标签的用户，且对性能、实时性均有较高要求。

Bitmap（又名Bitset）数据结构可以较好地实现以上需求，使用少量的存储空间来实现海量数据的查询优化。Redis社区版支持Bitmap运算，但是原生Bitmap往往难以应付超大规模的人群打标问题：

- 原生Bitmap受限于keyspace的大小，对于稀疏场景会出现空间效率急剧降低的情况。
- 使用string进行Bitmap操作时，很多计算逻辑需要上载到用户代码逻辑中执行，一来一回增加了3次额外的RTT（Round-Trip Time，往返时延）。
- 原生Redis存储Bitmap时极易产生大key，会对集群稳定性带来极大的挑战。

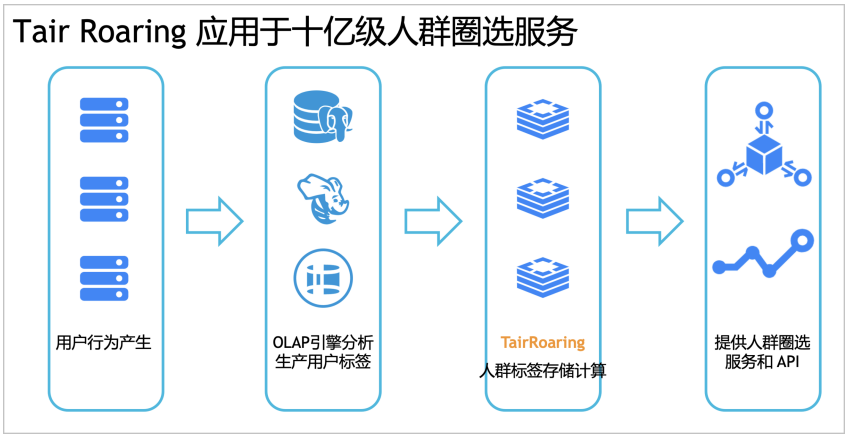
Tair Roaring属于高度工程优化的Bitmap实现：

- 通过2层索引和引入多种动态容器（Container），平衡了多种场景下性能和空间效率。
- 同时使用了包括SIMD instructions、Vectorization、PopCnt算法在内的等多种工程优化，提升了计算效率，实现了高效的时空效率。
- 基于Tair企业版提供的强大计算性能和极高的稳定性，为用户场景保驾护航。

相比较原生Bitmap，TairRoaring具有更低的内存占用、更高的集合计算效率，同时依托Tair高性能，提供更低的延迟和更高的吞吐。

目标用户筛选流程

人群筛选类业务往往包括模型的生成和筛选等多个步骤。



1. 用户特征的原始数据往往存储在关系型数据库中，通过行schema存储用户的不同维度特征。
2. 按需对原始数据进行处理，生成用户UID到人群Tag信息的映射关系。
3. 定时更新（导入）用户UID到标签信息的映射关系到TairRoaring引擎中（一般为T+1，表示业务的第二天导入前一天的数据）。
4. 通过TairRoaring加速业务计算：
   - 查询目标用户与人群Tag的关系。
     例如，判断用户（user1）是否属于人群Tag-A (编号 16161)。

     ```
     TR.GETBIT user1 16161
     ```

   - 通过 `AND` 、 `OR` 、 `DIFF` 等操作构造逻辑人群，并对逻辑人群信息进行计算。
     例如，获取同时属于人群Tag-B和Tag-C的所有目标用户。

     ```
     TR.BITOP result AND Tag-B Tag-C
     ```

   - 也存在部分从Tag信息到用户UID映射关系的场景，如风险控制场景等。
     例如，查询用户（user1）是否属于某个人群Tag-A中。

     ```
     TR.GETBIT Tag-A user1
     ```

5.通用最佳实践

5.1. 使用云数据库Redis版实现即时通信场景中的多端同步

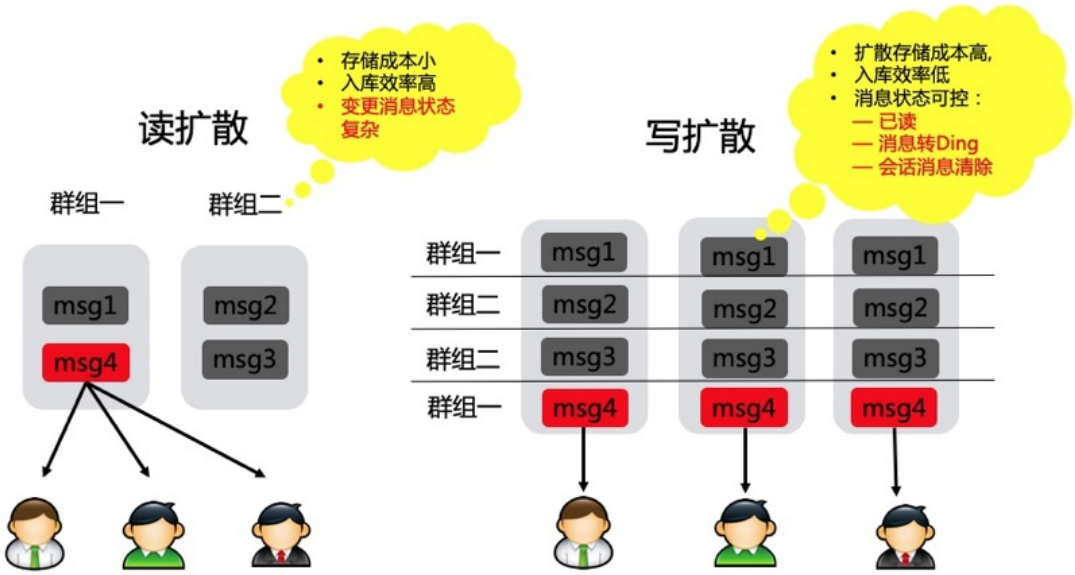
随着即时通信（Instant Messaging）场景和客户端种类的不断丰富，多端通信已经成为普遍趋势。本文为您介绍一种使用云数据库Redis版在IM场景中实现多端同步的方案。

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：

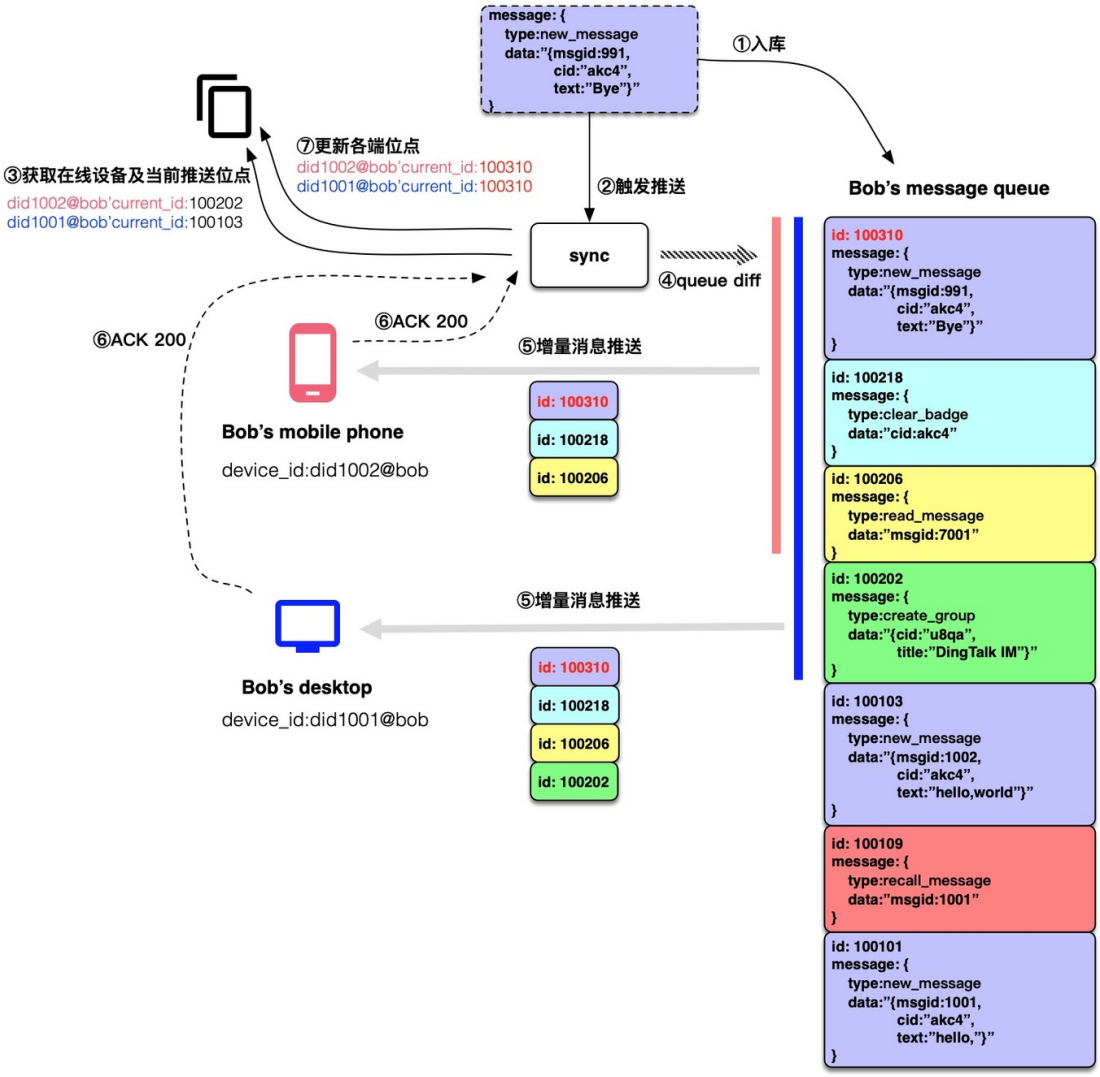
   ```
   struct message {
     int type; // 业务类型
     string data; // 业务数据
   }
   ```

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。

5.2. 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

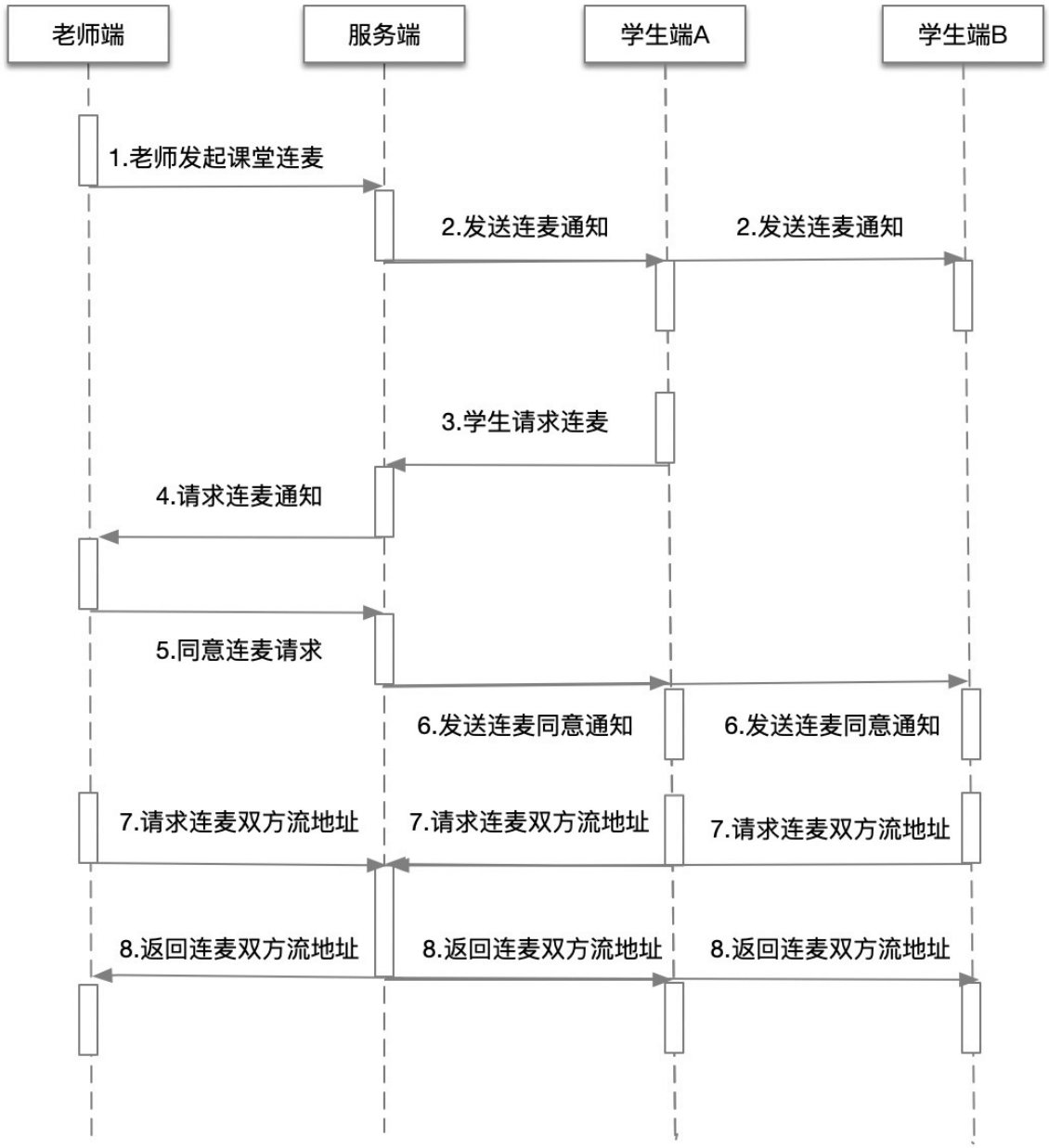
随着互联网直播的流行，直播已经走向了各行各业，老师使用直播应用进行线上教学也屡见不鲜，尤其是在2020年的新冠肺炎抗疫大作战中，在线课堂成为技术抗疫的重要一环，切实解决了师生无法到校上课的困境。

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。

5.3. 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

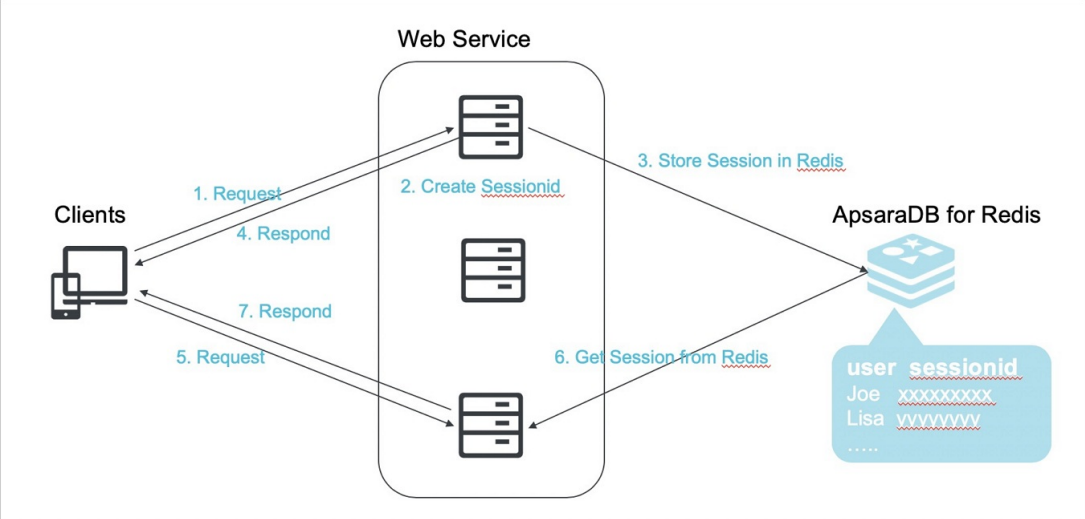
- 已创建用于保存会话的Redis实例。相关操作，请参见创建实例。
- 如果通过内网连接Redis实例，Redis实例与部署应用的ECS实例需在同一VPC中，或者同属经典网络且在同一地域。
- 已将ECS实例的内网IP地址添加到Redis实例的白名单中。相关操作，请参见设置白名单。

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Spring 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：

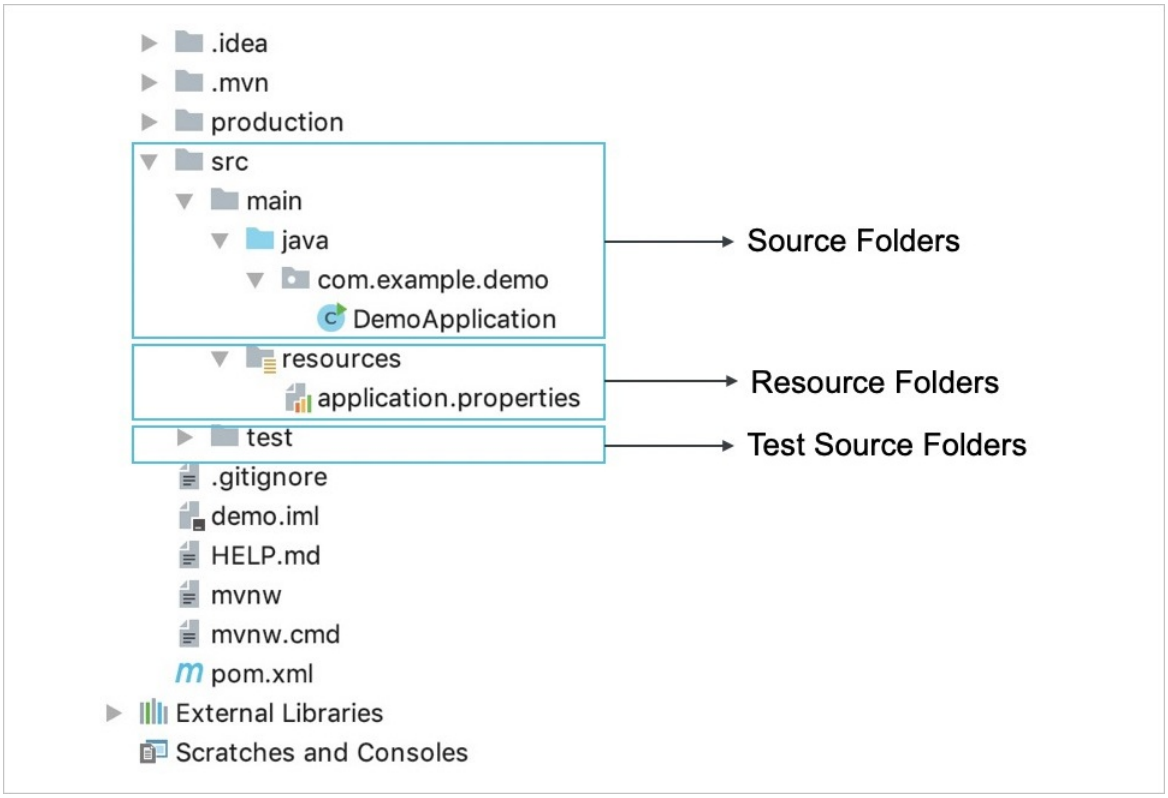
- 支持集群会话（Clustered Sessions）
  Spring Session使支持集群会话变得很简单，不必依赖于应用程序容器（例如Tomcat等）提供的特定解决方案。
- 支持多用户会话
  Spring Session支持在单个浏览器实例中管理多个用户的会话，例如谷歌浏览器中多用户同时登录的情况。
- 支持RESTful API
  Spring Session允许在标头（header）中提供会话ID以支持RESTful API。

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.EnableRedisHtt
pSession;
@SpringBootApplication
@EnableRedisHttpSession
public class DemoApplication {
    public static void main(String[] args) {
        SpringApplication.run(DemoApplication.class, args);
    }
}
```

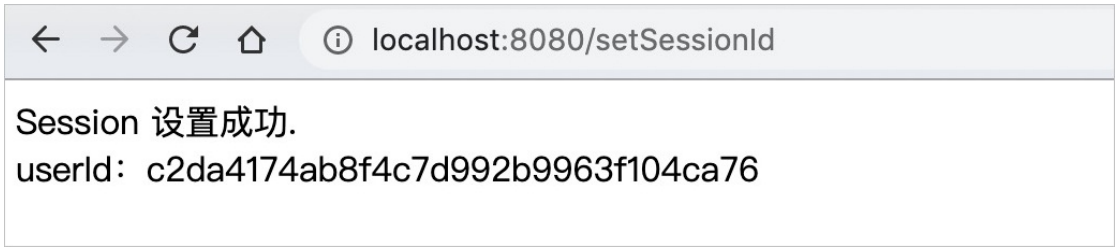
5. 编写业务逻辑代码。

   下方是使用Controller来处理HTTP请求的示例代码。

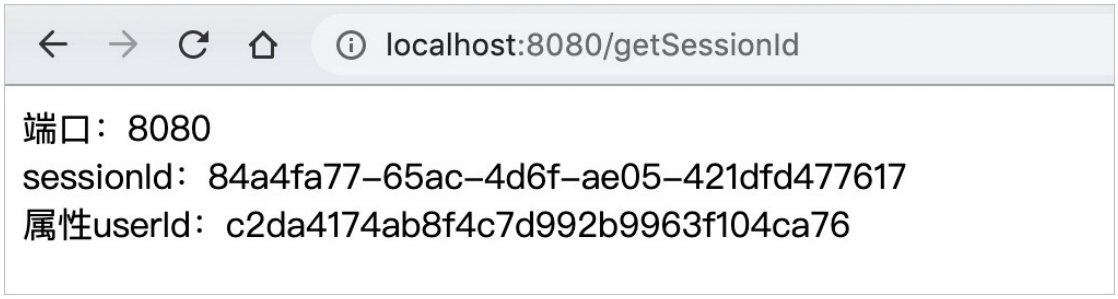
```
package com.example.demo.controller;
import javax.servlet.http.HttpServletRequest;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseBody;
import java.util.UUID;
@Controller
@RequestMapping("/")
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。

5.4. 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

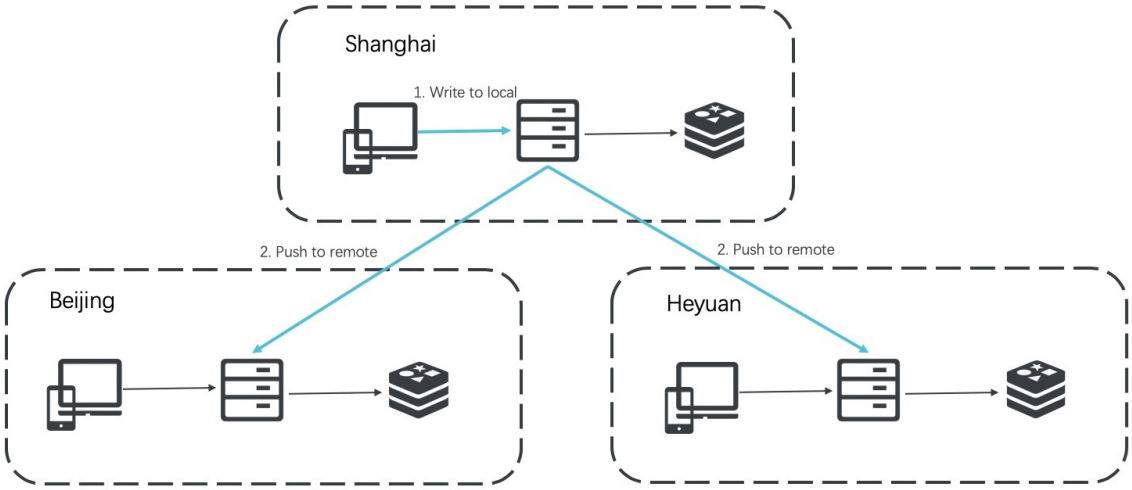
- 需求
  - 因用户遍布全国，部署应用服务的地域需要间隔稍远，尽量使全国各地用户在访问业务时都能获得较理想的访问速度。

- 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。

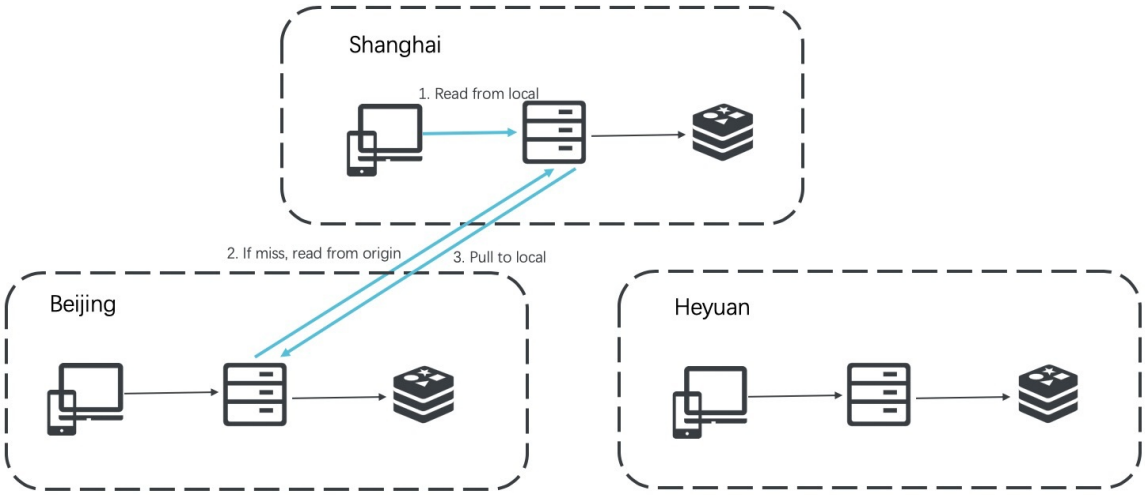
结合以上需求分析结果制定的业务方案如下：

- 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，架构图如下。



- 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，架构图如下。



- 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。

5.5. 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。

```
MySQL [custm_info]> SELECT * FROM company;
+----------------------------------+----------------------------------+------------+----------------------------------+----------------------------+--------------------------+
| id                               | name                             | sdate      | email                            | domain                     | city                     |
+----------------------------------+----------------------------------+------------+----------------------------------+----------------------------+--------------------------+
| [redacted]d96b5                  | [redacted]unter                  | 1986-05-23 | [redacted].com                   | [redacted]@example.net     | Michaelborough           |
| [redacted]662a7                  | [redacted]                       | 1970-12-11 | [redacted]                       | [redacted]@example.com     | Pamelaborough            |
| [redacted]db6c1                  | [redacted]                       | 2001-09-06 | [redacted]                       | [redacted]example.net      | Port Melodybury          |
| [redacted]38c2a                  | [redacted]                       | 1979-06-08 | [redacted]th.org                 | [redacted]ple.org          | New Tracymouth           |
| [redacted]65613                  | [redacted] Hernandez             | 1975-11-01 | [redacted]                       | [redacted]@example.net     | North Matthewhaven       |
| [redacted]bb993                  | [redacted]d Wagner               | 2004-06-29 | [redacted]                       | [redacted]le.net           | East Angelamouth         |
| [redacted]132b1                  | [redacted]                       | 2018-01-03 | [redacted]mith.com               | [redacted]ple.org          | Bradleychester           |
| [redacted]0898c                  | [redacted]                       | 1971-12-07 | [redacted]                       | [redacted]e.org            | East Christianhaven      |
| [redacted]a5882                  | [redacted]                       | 1976-07-14 | [redacted]                       | [redacted]e.org            | Port Christopherberg     |
```

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'
)
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
-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

选项	说明	示例值
-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-bp1xxxxxxxxxxxx.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

```
[root@ ~]# mysql -h rm-bp1 .mysql.rds.aliyuncs.com -P 3306 -u  -D custm_info -
p --skip-column-names --raw < mysql_to_redis.sql | redis-cli -h r-bp1 .redis.rds.aliyuncs.com --pipe -a
Enter password:
All data transferred. Waiting for the last reply...
Last reply received from server.
errors: 0, replies: 100
```

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。

```
r-bp1[redacted].redis.rds.aliyuncs.com:6379> HGETALL [redacted]6b132b1
 1) "name"
 2) "[redacted]ons"
 3) "sdate"
 4) "2018-01-03"
 5) "email"
 6) "[redacted]@[redacted]smith.com"
 7) "domain"
 8) "[redacted]@example.org"
 9) "city"
10) "Bradleychester"
```

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。

5.6. 游戏玩家积分排行榜

云数据库Redis兼容开源Redis，本文通过简单的示例，为您演示通过云数据库Redis实现在线游戏中的积分排行榜功能。

体验开发者云实验室

您可以通过开发者云实验室，免费获取一台配置了CentOS 7.7的ECS实例（云服务器）和一个Redis 5.0数据库实例，根据云实验室的教程，使用Java语言实现基于Redis数据库的游戏玩家积分排行榜功能。更多信息，请体验云实验室。

环境说明

云产品	说明
ECS实例	• 操作系统为Ubuntu 16.04.6。 • 网络类型为专有网络，且与Redis实例属于同一专有网络。
Redis实例	网络类型为专有网络，且与ECS实例属于同一专有网络。

> **说明** 如果Redis实例与ECS的专有网络不同，您可以更换Redis实例的专有网络，具体操作，请参见更换专有网络VPC或交换机；如果网络类型不同，解决方法请参见ECS实例与Redis实例的网络类型不同时如何连接。

操作步骤

1. 设置Redis实例的白名单，保障ECS实例和Redis实例可以互通。
   i. 获取ECS实例的内网IP地址。具体操作，请参见查询ECS实例的IP地址。
   ii. 将获取到的ECS内网IP地址添加至Redis实例的白名单中。具体操作，请参见设置白名单。
2. 登录ECS实例。具体操作，请参见连接方式概述。
3. 在ECS实例中，执行下述命令安装环境。

```
sudo apt-get update
sudo apt-get install openjdk-8-jdk
apt install maven
```

4. 执行下述命令，下载并解压示例代码。

```
wget https://docs-aliyun.cn-hangzhou.oss.aliyun-inc.com/assets/attach/120287/cn_zh/1615470698355/source.tar.gz
tar xvf source.tar.gz && cd source
```

5. 执行vim src/main/java/test/GameRankSample.java命令，按照要求修改示例代码中各参数的值。

> **说明** 执行命令后，系统将进入编辑界面，输入*a*可进入编辑模式。

参数配置示例

```
public static void main(String[] args) {
  //Connection information. This information can be obtained from the console
  String host = "r-bp[redacted].redis.rds.aliyuncs.com";

  int port = 6379;
  Jedis jedis = new Jedis(host, port);
  try {
    //Instance password
    String authString = jedis.auth("F[redacted]"); //password
    if (!authString.equals("OK")) {
      System.err.println("AUTH Failed: " + authString);
      return;
    }
```

参数	说明
String host	分别填入Redis实例的专有网络连接地址与端口号，关于如何获取连接地址和端口号，请参见查看连接地址。
port	
String authString	账号的密码（该账号需具备**读写**权限），根据选取账号的不同，密码的填写格式有一定区别。关于如何创建账号，请参见创建与管理账号。 **说明** ◦ 默认账号（即以实例ID命名的账号）：直接填写密码即可。 ◦ 新创建的账号：密码格式为 `<user>:<password>` 。例如自定义账号为testaccount，密码为Rp829dlwa，密码需填写为testaccount:Rp829dlwa。

6. 按下Esc键退出编辑模式，输入*:wq*并按回车键保存配置并退出编辑界面。
7. 执行下述命令，运行示例代码。

```
mvn clean package assembly:single -DskipTests
java -classpath target/demo-0.0.1-SNAPSHOT.jar test.GameRankSample
```

执行结果

示例代码注释

```
package test;
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) {
    //连接地址，可通过Redis控制台获取
    String host = "r-gs50a75e1968****.redis.hangzhou.rds.aliyuncs.com";
    int port = 6379;
    Jedis jedis = new Jedis(host, port);
    try {
      //实例密码
      String authString = jedis.auth("Pass!123"); //password
      if (!authString.equals("OK")) {
        System.err.println("AUTH Failed: " + authString);
        return;
      }
      //Key（键）
      String key = "Game name: Keep Running, Alibaba Cloud!";
      //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) {
        //为每个玩家随机生成一个ID
        playerList.add(UUID.randomUUID().toString());
      }
      System.out.println("Inputs all players ");
      //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("Player ID:" + member + ", Player Score: " + score);
        //将玩家ID和分数添加到相应键的SortedSet中。
        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(" Ranking list of all players");
      //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(
          "Player ID:" +
          item.getElement() +
          ", Player Score:" +
          Double.valueOf(item.getScore()).intValue()
        );
      }
      //输出打印前五名玩家的信息
      System.out.println();
      System.out.println("       " + key);
      System.out.println("       Top players");
      scoreList = jedis.zrevrangeWithScores(key, 0, 4);
      for (Tuple item : scoreList) {
        System.out.println(
          "Player ID:" +
          item.getElement() +
          ", Player Score:" +
          Double.valueOf(item.getScore()).intValue()
        );
      }
      //输出打印特定玩家列表
      System.out.println();
      System.out.println("         " + key);
      System.out.println(" Players with scores from 1,000 to 2,000");
      //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(
          "Player ID:" +
          item.getElement() +
          ", Player Score:" +
          Double.valueOf(item.getScore()).intValue()
        );
      }
    } catch (Exception e) {
      e.printStackTrace();
    } finally {
      jedis.quit();
      jedis.close();
    }
```

```
    }
}
```

5.7. 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:"+D
ouble.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]+", "+produc
ts[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]+", "+produc
ts[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
```

5.8. 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:"+mes
sage);
        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“unsub
scribe”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-77c7-4498-89a0-1dc5a2f
65889
  <<< 订阅(SUBSCRIBE)< Channel:KVStore频道-A >接收到的Message:0f9c2cee-77c7-4498-89a0-1dc5a2f
65889
  >>> 发布(PUBLISH) > Channel:KVStore频道-A > 发送出的Message:ed5924a9-016b-469b-8203-7db63d0
6f812
  <<< 订阅(SUBSCRIBE)< Channel:KVStore频道-A >接收到的Message:ed5924a9-016b-469b-8203-7db63d0
6f812
  >>> 发布(PUBLISH) > Channel:KVStore频道-A > 发送出的Message:f1f84e0f-8f35-4362-9567-25716b1
531cd
  <<< 订阅(SUBSCRIBE)< Channel:KVStore频道-A >接收到的Message:f1f84e0f-8f35-4362-9567-25716b1
531cd
  >>> 发布(PUBLISH) > Channel:KVStore频道-A > 发送出的Message:746bde54-af8f-44d7-8a49-37d1a24
5d21b
  <<< 订阅(SUBSCRIBE)< Channel:KVStore频道-A >接收到的Message:746bde54-af8f-44d7-8a49-37d1a24
5d21b
  >>> 发布(PUBLISH) > Channel:KVStore频道-A > 发送出的Message:8ac3b2b8-9906-4f61-8cad-84fc1f1
5a3ef
  <<< 订阅(SUBSCRIBE)< Channel:KVStore频道-A >接收到的Message:8ac3b2b8-9906-4f61-8cad-84fc1f1
5a3ef
  >>> 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。

5.9. 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.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，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);
                    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
```

5.10. 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。

> **说明** 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不会执行事务回滚。

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
```

5.11. 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

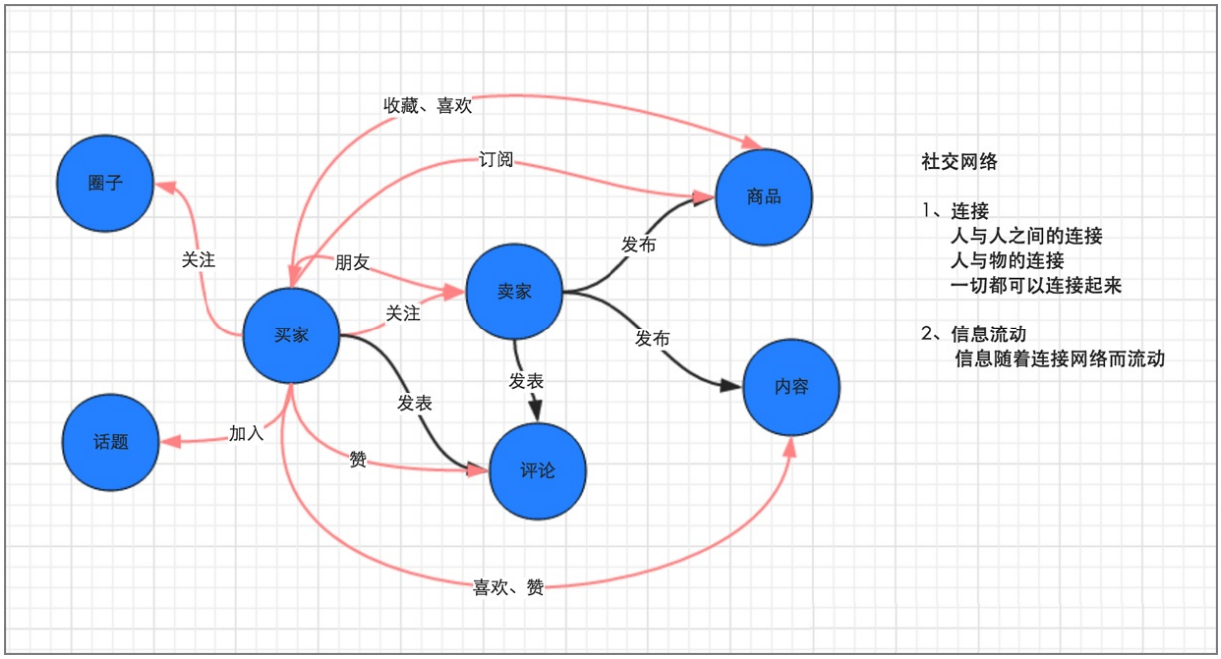
目前云数据库Redis版提供了标准单副本、标准双副本和集群版本。

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。

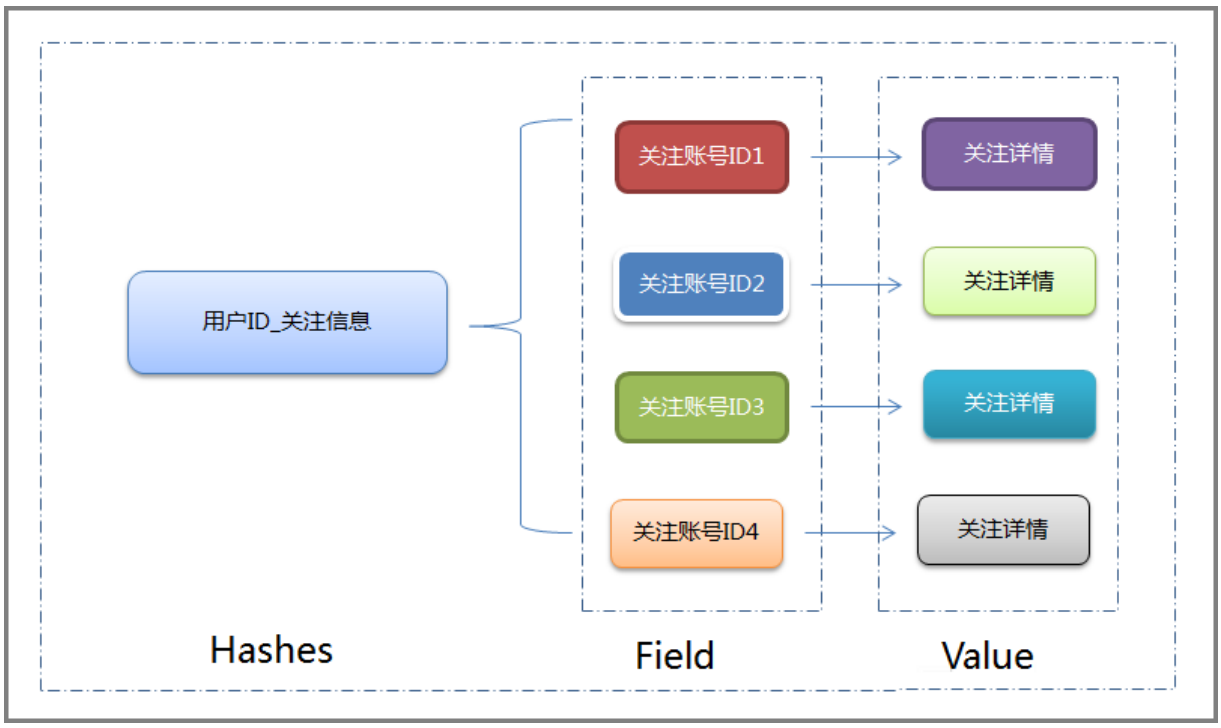
微淘社区之亿级关系链存储

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

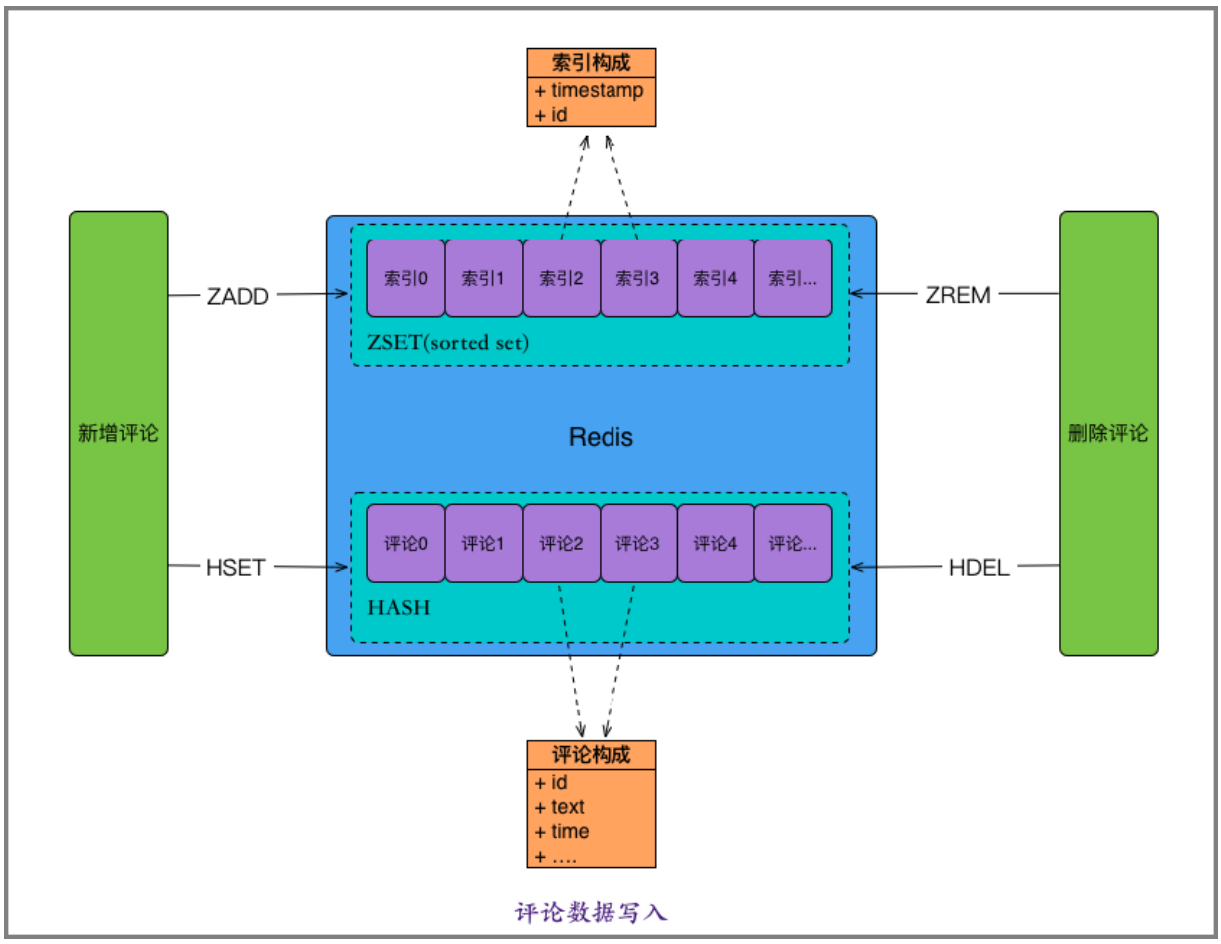


天猫直播之评论商品游标分页

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：

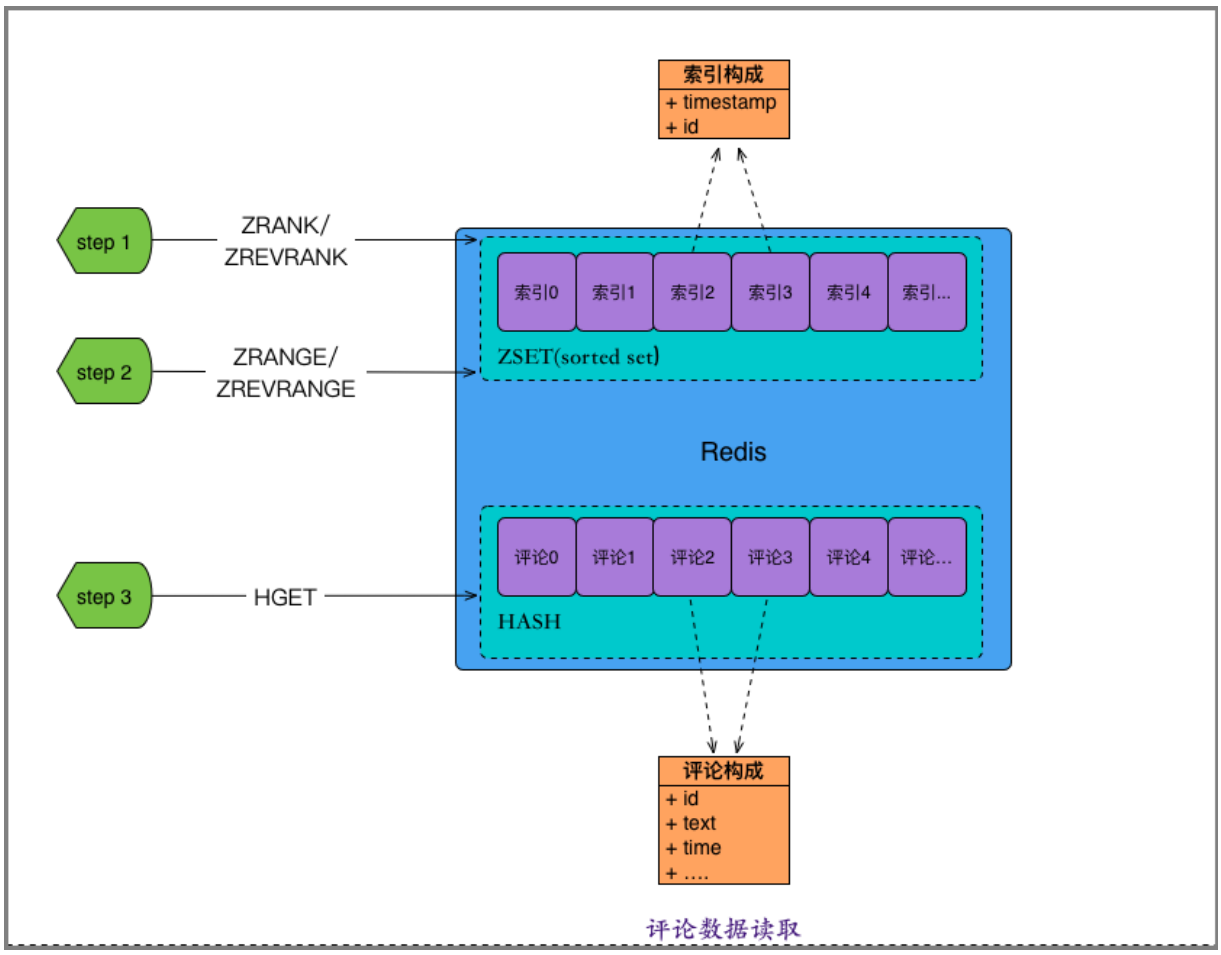
- 增量下拉：从指定位置向上获取指定个数（增量）的评论。
- 下拉刷新：获取最新的指定个数的评论。
- 增量上拉：从指定位置向下获取指定个数（增量）的评论。

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：

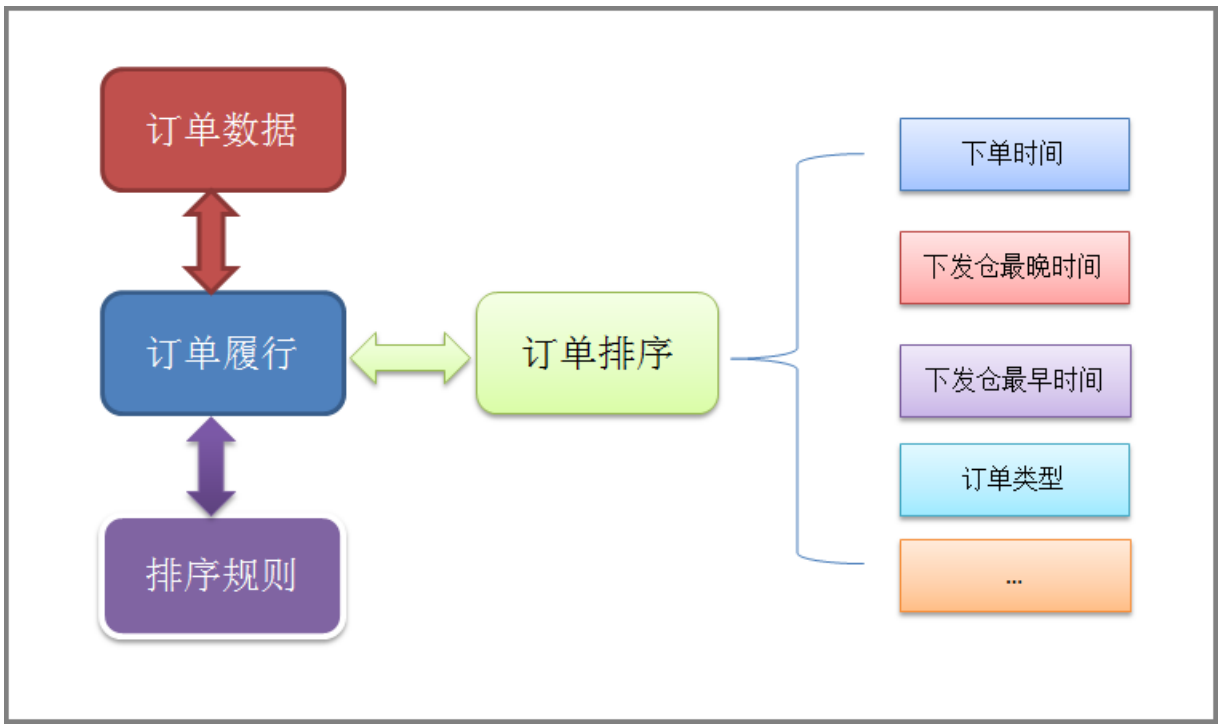
1. 获取当前索引位置
2. 获取索引列表
3. 获取评论数据



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。



5.12. 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

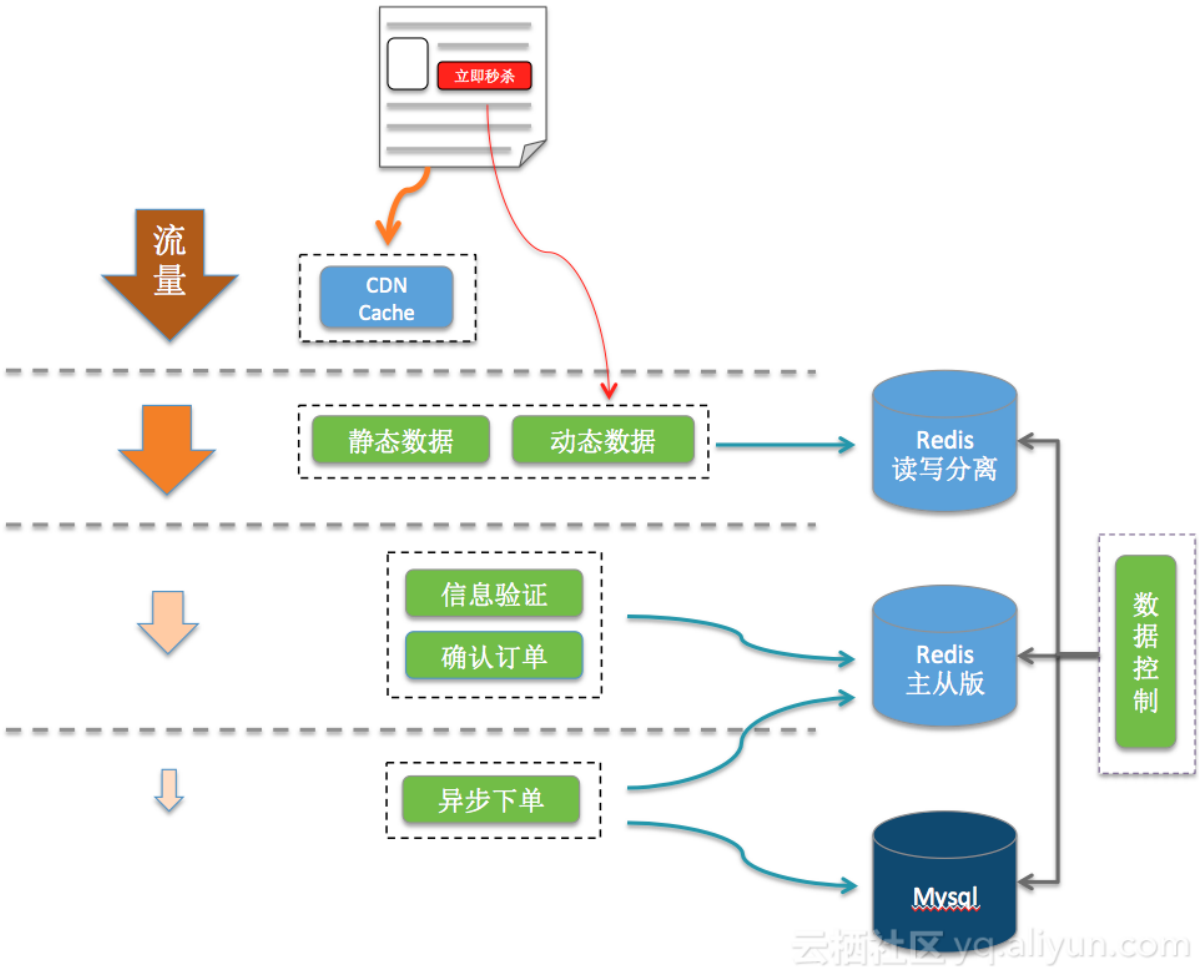
秒杀活动对稀缺或者特价的商品进行定时定量售卖，吸引成大量的消费者进行抢购，但又只有少部分消费者可以下单成功。因此，秒杀活动将在较短时间内产生比平时大数十倍，上百倍的页面访问流量和下单请求流量。

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": 0
}
```

说明 `goodsId` 表示商品ID，`Total` 表示该商品的库存数量，`Booked` 表示该商品已被订购的数量。

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，步骤如下：

1. 缓存lua脚本至Redis。

   ```
   SCRIPT LOAD "lua code"
   ```

   返回结果为：

   ```
   "438dd755f3fe0d32771753eb57f075b18fed7716"
   ```

2. 调用该lua脚本。

   ```
   EVALSHA 438dd755f3fe0d32771753eb57f075b18fed7716 1 goodsId 1
   ```

返回结果如下，表示扣减了1个库存：

```
(integer) 1
```

> **说明** 此时，执行HGET goodsId Booked命令，可查看到返回的值为 `"1"` ，即该商品已被订购的数量为1。

秒杀服务可通过判断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。

5.13. 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

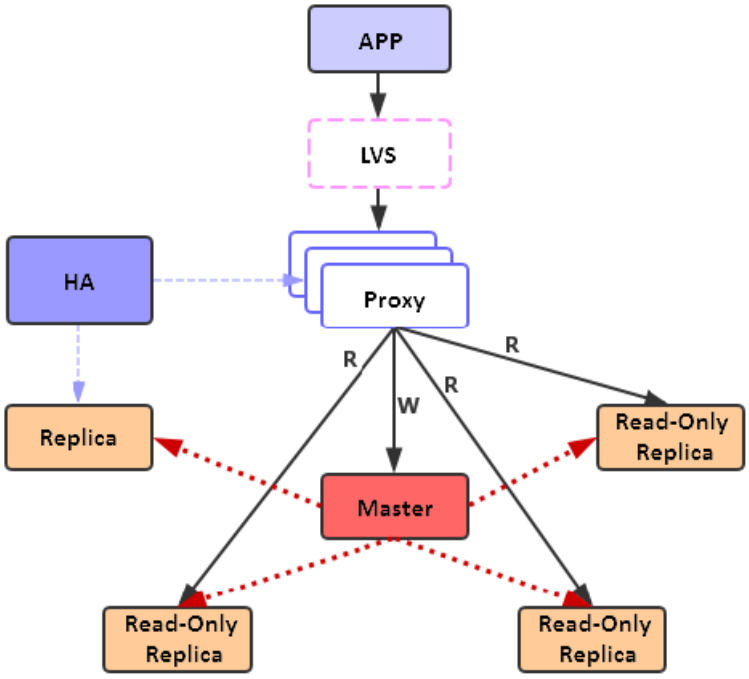
Redis集群模式有redis-proxy、master、replica、HA等几个角色。在读写分离实例中，新增read-only replica角色来承担读流量，replica作为热备不提供服务，架构上保持对现有集群规格的兼容性。redis-proxy按权重将读写请求转发到master或者某个read-only replica上；HA负责监控DB节点的健康状态，异常时发起主从切换或重搭read-only replica，并更新路由。

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。

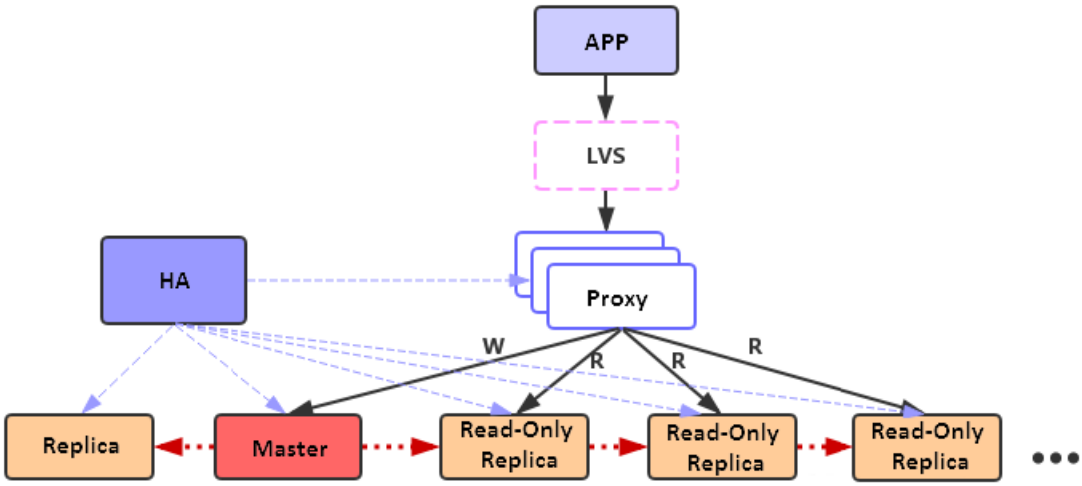


链式复制

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

规格	QPS	带宽
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。

5.14. 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, redisPassword);
//执行命令如下
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。

空闲资源检测相关参数

名称	说明	默认值	建议
testWhileIdle	是否在空闲资源监测时通过ping命令监测连接有效性，无效连接将被销毁。	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。	1,800,000（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 th
e pool
…
Caused by: java.util.NoSuchElementException: Timeout waiting for idle object
at org.apache.commons.pool2.impl.GenericObjectPool.borrowObject(GenericObjectPool.java:44
9)
```

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

```
redis.clients.jedis.exceptions.JedisConnectionException: Could not get a resource from th
e pool
…
Caused by: java.util.NoSuchElementException: Pool exhausted
at org.apache.commons.pool2.impl.GenericObjectPool.borrowObject(GenericObjectPool.java:46
4)
```

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新建一个 `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 {
    }
}
```

5.15. 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。

```
Welcome to Alibaba Cloud Elastic Compute Service !

[root@redisTest ~]# telnet r-b          4.redis.rds.aliyuncs.com 6379
Trying 1          ...
Connected to r-b          4.redis.rds.aliyuncs.com.
Escape character is '^]'.
auth          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                    ] "INFO" "replication"
+1543975833.071774 [0                    ] "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                    ] "INFO" "replication"
```

> **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。

```
[root@redisTest ~]# cat imonitorOut.txt
1543995847.659482 [0                    ] "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                    ] "INFO" "ALL"
1543995862.782848 [0                    ] "INFO" "ALL"
1543995862.799096 [0                    ] "INFO" "ALL"
1543995862.863230 [0                    ] "INFO" "CLUSTER"
1543995862.876389 [0                    ] "scan" "0" "MATCH" "*" "COUNT" "3000"
1543995862.942649 [0                    ] "INFO" "replication"
1543995862.943303 [0                    ] "TYPE" "customer:18016"
1543995862.955943 [0                    ] "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_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)
    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_cou
nt) * 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 = Tr
ue)
        self._pretty_print(self._top_n(self.keys), 'Top Keys', percentages = True)
        self._pretty_print(self._top_n(self.commands), 'Top Commands', percentages = Tr
ue)
        self._pretty_print(self._time_stats(self.times), 'Command Time (microsecs)')
        self._pretty_print(self._heaviest_commands(self.times), 'Heaviest Commands (mic
rosecs)')
        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.re
dis_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
HLEN    6155302.75
HSCAN   2166953.0
TYPE    2031287.75
scan    66260.0
TTL     35047.25

Slowest Calls
============================================
8397898.75      "INFO" "replication"
8101143.0       "INFO" "ALL"
8079963.75      "INFO" "ALL"
```

> 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。

5.16. 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) "follower"
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请参见实时Top Key统计。
- 使用离线全量Key分析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。

5.17. 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']
```

5.18. 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。

```
r-[redacted].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设置实例参数。

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：

```
[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。

5.19. 将ECS实例自动加入和移出Redis实例白名单

本教程介绍如何使用弹性伸缩生命周期挂钩挂起ECS实例，并结合运维编排服务OOS的模板，实现将ECS实例自动加入和移出Redis实例白名单。

前提条件

- 使用本教程进行操作前，请确保您已经注册了阿里云账号。如还未注册，请先完成账号注册。
- 已创建伸缩组，且伸缩组处于启用状态。
- 已创建Redis实例。
- 已为OOS服务创建RAM角色。该RAM角色的可信实体必须为**阿里云服务**，受信服务为**运维编排服务**，且拥有执行OOS模板的权限。具体操作，请参见为OOS服务设置RAM权限。

> **说明** 本教程中使用的示例RAM角色为OOSServiceRole，您也可以使用其他自定义的RAM角色。

背景信息

伸缩组支持关联负载均衡实例和RDS实例，但是暂时不能关联Redis实例。如果您有业务数据存储在Redis实例上，手动配置ECS实例加入或移出Redis实例白名单，操作效率较低。您可以通过生命周期挂钩和OOS模板将ECS实例自动加入和移出Redis实例白名单。

操作步骤

本教程以OOS公共模板ACS-ESS-LifeCycleModifyRedisIPWhitelist为例，实现在扩容时将ECS实例加入Redis实例白名单。步骤如下：

- 步骤一：对RAM角色授予OOS服务权限
- 步骤二：为扩容活动创建生命周期挂钩并触发扩容
- 步骤三：查看Redis实例白名单
- （可选）步骤四：查看OOS执行情况

> **说明** 如果需要在缩容时将ECS实例移出Redis实例白名单，创建适用于弹性收缩活动的生命周期挂钩并触发缩容即可。

步骤一：对RAM角色授予OOS服务权限

您需要拥有OOS的执行权限才能执行OOS的模板。执行ACS-ESS-LifeCycleModifyRedisIPWhitelist中定义的运维操作时涉及云服务器ECS、弹性伸缩、云数据库Redis的资源。

1. 登录RAM控制台。
2. 创建权限策略。
   i. 在左侧导航栏，单击**权限管理 > 权限策略**。
   ii. 单击**创建权限策略**。
   iii. 在**新建自定义权限策略**页面，指定权限配置，然后单击**确定**。

   本教程中使用的配置如下表所示，未提及的配置保持默认即可。

配置项	说明
策略名称	填写ESSHookPolicyForRedisWhitelist。
配置模式	选择**脚本配置**。
策略内容	输入以下内容： `{ "Version": "1", "Statement": [ { "Action": [ "ecs:DescribeInstances" ], "Resource": "*", "Effect": "Allow" }, { "Action": [ "kvstore:ModifySecurityIps" ], "Resource": "*", "Effect": "Allow" }, { "Action": [ "ess:CompleteLifecycleAction" ], "Resource": "*", "Effect": "Allow" } ] }`

3. 为OOSServiceRole授予权限策略。
   i. 在左侧导航栏，单击**身份管理 > 角色**。
   ii. 找到OOSServiceRole，在**操作**列，单击**添加权限**。

   为OOS服务扮演的RAM角色OOSServiceRole添加权限即可完成授权。

iii. 在**添加权限**页面，指定权限配置，然后单击**确定**。

本教程中使用的配置如下表所示，未提及的配置保持默认即可。

配置项	说明
授权范围	选择**整个云账号**。
选择权限	添加自定义策略ESSHookPolicyForRedisWhitelist。

步骤二：为扩容活动创建生命周期挂钩并触发扩容

在创建生命周期挂钩时选择通知方式为OOS模板并设置相关参数，即可在触发扩容活动时将ECS实例自动加入Redis实例白名单。

1. 登录弹性伸缩控制台。
2. 在左侧导航栏中，单击**伸缩组管理**。
3. 在顶部菜单栏处，选择地域。
4. 找到待操作的伸缩组，选择任一种方式打开伸缩组详情页面。
   - 在**伸缩组名称/ID**列，单击伸缩组ID。
   - 在**操作**列，单击**查看详情**。
5. 为扩容活动创建生命周期挂钩。
   i. 在页面上方，单击**生命周期挂钩**页签。
   ii. 单击**创建生命周期挂钩**。

iii. 指定生命周期挂钩配置，然后单击**确认**。

本教程中使用的配置如下表所示，未提及的配置保持默认即可。

配置项	说明
名称	输入ESSHookForAddRedisWhitelist。
适用的伸缩活动类型	选择**弹性扩张活动**。
超时时间	输入适当的超时时间，例如300秒。 **说明** 超时时间即用于执行自定义操作的时间，若超时时间过短，可能导致自定义操作失败，请评估自定义操作耗时并设置适当的超时时间。
执行策略	选择**继续**。
通知方式	模板配置如下： ■ 通知方式：选择**OOS模板**。 ■ OOS模板类型：选择**公共模板**。 ■ 公共模板：选择ACS-ESS-LifeCycleModifyRedisIPWhitelist。 ACS-ESS-LifeCycleModifyRedisIPWhitelist的执行参数配置如下： ■ **Redis实例ID**：输入Redis实例的ID。 ■ **修改IP白名单的方式**：选择Append，对应弹性扩张活动，将ECS实例加入Redis实例白名单。 ■ **执行使用到的权限的来源**：选择OOSServiceRole，步骤一中已为RAM角色OOSServiceRole添加操作ECS、弹性伸缩、Redis资源的权限，OOS服务扮演该RAM角色即可拥有相关权限。

6. 触发扩容。

本教程中以手动执行伸缩规则为例，您也可以通过定时任务、报警任务等方式触发扩容。

说明 手动执行伸缩规则触发扩缩容时，生命周期挂钩会生效，但手动添加或移出已有ECS实例时，生命周期挂钩不会生效。

i. 在页面上方，单击**伸缩规则与伸缩活动**页签，然后单击**伸缩规则**页签。

ii. 单击**创建伸缩规则**。

iii. 设置伸缩规则的属性，然后单击**确认**。

本教程中使用的配置如下表所示，未提及的配置保持默认即可。

配置项	说明
规则名称	输入Add1。
伸缩规则类型	选择**简单规则**。
执行的操作	设置为增加1台。

iv. 在**伸缩规则**页面，找到新建的伸缩规则Add1，在**操作**区域，单击**执行**。

v. 单击**确定**。

执行伸缩规则后自动创建1台ECS实例，由于伸缩组内已创建生命周期挂钩ESSHookForAddRedisWhitelist，ECS实例会被挂起，同时自动通知OOS服务执行ACS-ESS-LifeCycleModifyRedisIPWhitelist中定义的运维操作。

步骤三：查看Redis实例白名单

1. 登录云数据库Redis版管理控制台。
2. 在左侧导航栏，单击**实例列表**。
3. 找到Redis实例，在**实例ID/名称**区域，单击实例ID。
4. 在左侧导航栏，单击**白名单设置**。

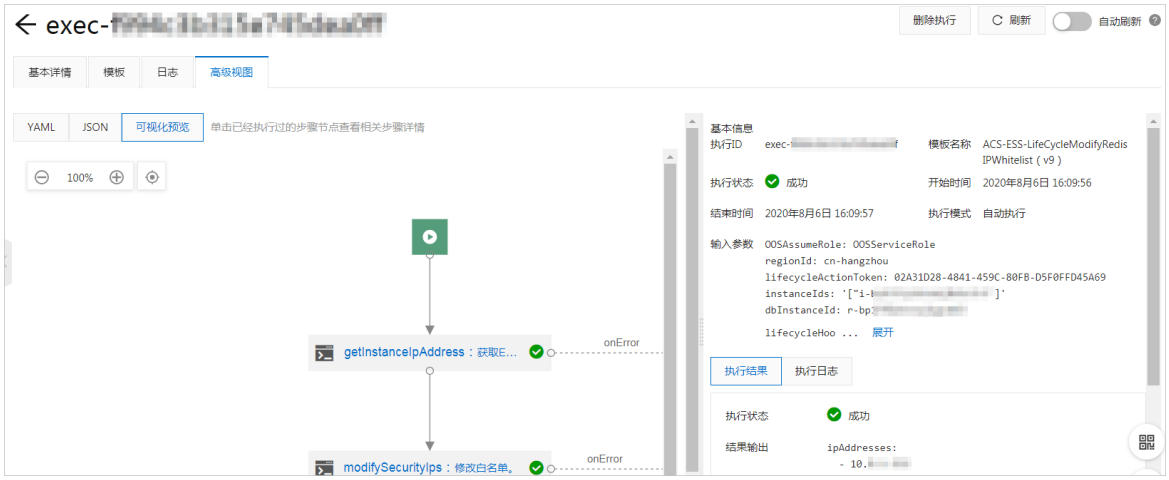
如下图所示，Redis实例白名单中加入了新建ECS实例的私有IP，符合使用公共模板ACS-ESS-LifeCycleModifyRedisIPWhitelist的预期。

如果成功创建了ECS实例，但是新建ECS实例的私有IP并没有加入Redis实例白名单，请前往OOS控制台查看运维任务执行情况，具体步骤请参见（可选）步骤四：查看OOS执行情况。

（可选）步骤四：查看OOS执行情况

1. 登录OOS管理控制台。
2. 在左侧导航栏，单击**执行管理**。
3. 按开始时间找到执行，然后在**操作**列，单击**详情**。
4. 单击**高级视图**。

执行结果页签中显示执行状态。



如果执行失败，**执行结果**页签中也会显示相关的报错信息。

常见问题

如果运维任务执行失败，请根据执行结果中的报错信息排查原因。常见的报错信息及解决方案如下：

- 报错信息：`Forbidden.Unauthorized message: A required authorization for the specified action is not supplied.`
  解决方案：请检查是否为RAM角色OOSServiceRole添加了相应的权限，例如步骤一中的示例权限。您需要为RAM角色添加操作权限，确保OOS服务能够操作OOS模板中涉及的资源。
- 报错信息：`Forbidden.RAM message: User not authorized to operate on the specified resource, or this API doesn't support RAM.`
  解决方案：请检查是否为RAM角色OOSServiceRole添加了相应的权限，例如步骤一中的示例权限。您需要为RAM角色添加操作权限，确保OOS服务能够操作OOS模板中涉及的资源。
- 报错信息：`LifecycleHookIdAndLifecycleActionToken.Invalid message: The specified lifecycleActionToken and lifecycleActionId you provided does not match any in process lifecycle action.`
  解决方案：请评估生命周期挂钩的超时时间，确保在超时时间内可以执行完OOS模板中定义的运维任务。