

分布式缓存的通用方法——《可伸缩服务架构》

本文节选自《可伸缩服务架构：框架与中间件》第四章《缓存的本质和缓存使用的优秀实践》。PS：本文最后有《可伸缩服务架构：框架与中间件》共5本抽奖活动，试试你的运气吧~

4 分布式缓存的通用方法

笔者所在的多家互联网公司大量使用了缓存，对分布式缓存的应用可谓遍地开花，笔者曾供职的一家社交媒体网站，号称是世界上使用缓存最多的公司。毋庸置疑，缓存帮助我们解决了很多性能问题，甚至帮助我们解决了一些并发问题。

4.4.1 缓存编程的具体方法

各种分布式缓存如Redis，都提供了不同语言的客户端API，我们可以使用这些API直接访问缓存，也可以通过注解等方法使用缓存。

1. 编程法

编程法指通过编程的方式直接访问缓存，伪代码如下：

```
String userKey = ...;
User user = (User)cacheService.getObject(userKey)

if (user == null) {
    User user = (User)userService.getUser(userKey)

    if (user != null)
        cacheService.setObject(userKey, user);
}

return user;
```

 开涛的博客

这种方法实现起来简单，但是每次使用时都得敲入类似上面这样的一段代码，很烦琐，可以将这部分内容抽象成一个框架，请参考下面的小节。

2. Spring注入法

spring-data-redis项目 (<https://projects.spring.io/spring-data-redis>) 实现了注入法，通过Bean注入就可以直接使用Spring的缓存模板提供的方法。

首先，引入spring-data-redis包：

```
<dependencies>
  <dependency>
    <groupId>org.springframework.data</groupId>
    <artifactId>spring-data-redis</artifactId>
    <version>2.0.2.RELEASE</version>
  </dependency>
</dependencies>
```

开涛的博客

然后在Spring环境下进行如下配置：

```
<bean id="jedisConnFactory"
      class="org.springframework.data.redis.connection.jedis.JedisConnectionFactory"
      p:use-pool="true"/>

<!--redis 模板定义 -->
<bean id="redisTemplate"
      class="org.springframework.data.redis.core.RedisTemplate"
      p:connection-factory-ref="jedisConnFactory"/>
```

开涛的博客

再通过Spring环境注入使用的服务中：

```
public class UserLinkService{

    // 注入 Redis 的模板
    @Autowired
    private RedisTemplate<String, String> template;

    // 把模板当作 ListOperations 接口类型注入，也可以当作 Value、Set、Zset、HashOperations 接口类型注入
    @Resource(name="redisTemplate")
    private ListOperations<String, String> listOps;

    public void addLink(String userId, URL url) {
        //使用注入的接口类型
        listOps.leftPush(userId, url.toExternalForm());
        //直接使用模板
        redisTemplate.boundListOps(userId).leftPush(url.toExternalForm());
    }
}
```


开涛的博客

3. 注解法

spring-data-redis项目 (<https://projects.spring.io/spring-data-redis>) 实现了注解法，通过注解就可以在一个方法内部使用缓存，缓存操作都是透明的，我们不再需要重复写上面的一段代码。

首先，引入相应的依赖包：

```
<dependency>
  <groupId>org.springframework.data</groupId>
  <artifactId>spring-data-redis</artifactId>
  <version>1.6.0.RELEASE</version>
</dependency>
<dependency>
  <groupId>redis.clients</groupId>
  <artifactId>jedis</artifactId>
  <version>2.7.3</version>
</dependency>
```



然后，通过一个配置Bean配置Redis连接信息，这个配置Bean会通过Spring环境下的Bean扫描载入：


```
package com.robert.cache.redis;

import org.springframework.cache.CacheManager;
import org.springframework.cache.annotation.CachingConfigurerSupport;
import org.springframework.cache.annotation.EnableCaching;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.data.redis.cache.RedisCacheManager;
import org.springframework.data.redis.connection.RedisConnectionFactory;
import org.springframework.data.redis.connection.jedis.JedisConnectionFactory;
import org.springframework.data.redis.core.RedisTemplate;

@Configuration
@EnableCaching
public class RedisCacheAnnotationConfig extends CachingConfigurerSupport {

    @Bean
    public JedisConnectionFactory redisConnectionFactory() {
        JedisConnectionFactory redisConnectionFactory = new JedisConnectionFactory();

        redisConnectionFactory.setHostName("127.0.0.1");
        redisConnectionFactory.setPort(6379);
        return redisConnectionFactory;
    }
}
```



```

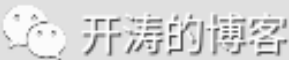
    }

    @Bean
    public RedisTemplate<String, String> redisTemplate(RedisConnectionFactory cf) {
        RedisTemplate<String, String> redisTemplate = new RedisTemplate<String, String>();
        redisTemplate.setConnectionFactory(cf);
        return redisTemplate;
    }

    @Bean
    public CacheManager cacheManager(RedisTemplate redisTemplate) {
        RedisCacheManager cacheManager = new RedisCacheManager(redisTemplate);

        // 这是默认的过期时间，默认为不过期(0)
        cacheManager.setDefaultExpiration(3000);
        return cacheManager;
    }
}

```



再在Spring环境下载入这些配置：


```
<context:component-scan base-package="com.defonds.bdp.cache.redis" />
```

最后，我们就可以通过注解来使用Redis缓存了，这样我们的代码就简单得多了：

```

@Cacheable("user")
public User getUser(String userId) {
    logger.debug("userId=?, user=?", userId, user);
    return this.userService.getUser(userId);
}

```



4.4.2 应用层访问缓存的模式

应用层访问分布式缓存的服务架构模式分为：双读双写、异步更新和串联模式。

1. 双读双写

双读双写的架构如图4-12所示。

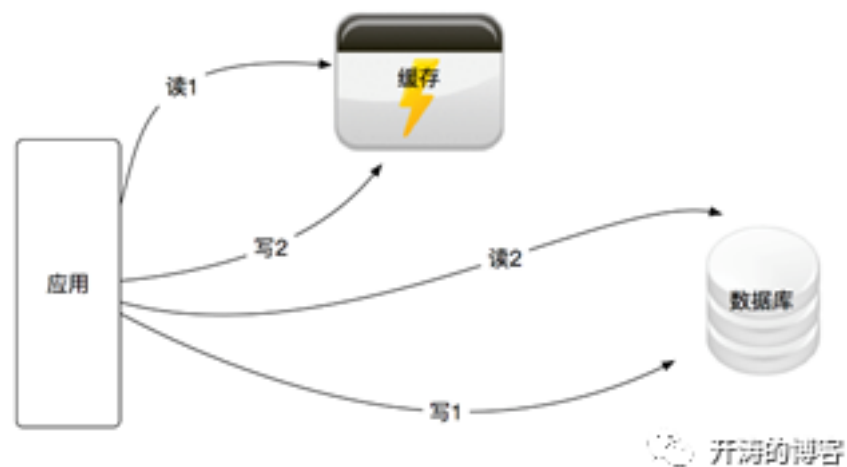


图4-12

这是我们最常用的缓存服务架构模式，对于读操作，我们先读缓存，如果缓存不存在这份数据，则再读数据库，读取数据库后再回写缓存；对于写操作，我们先写数据库，再写缓存。

这种方式实现起来简单，但是对应用层不透明，应用层需要处理读写顺序的逻辑，可参考4.4.1节。

2. 异步更新

异步更新的架构如图4-13所示。在异步更新的方式中，应用层只读写缓存，在这种情况下，全量数据会被保存在缓存中，并且不设置缓存系统的过期时间，由异步的更新服务将数据库里变更的或者新增的数据更新到缓存中。也有通过MySQL的binlog将MySQL中的更新操作推送到缓存的实现方法，这种方法与异步更新如出一辙，以Facebook的方案（请参考论文Scaling Memcache at Facebook）为例，如图4-14所示。

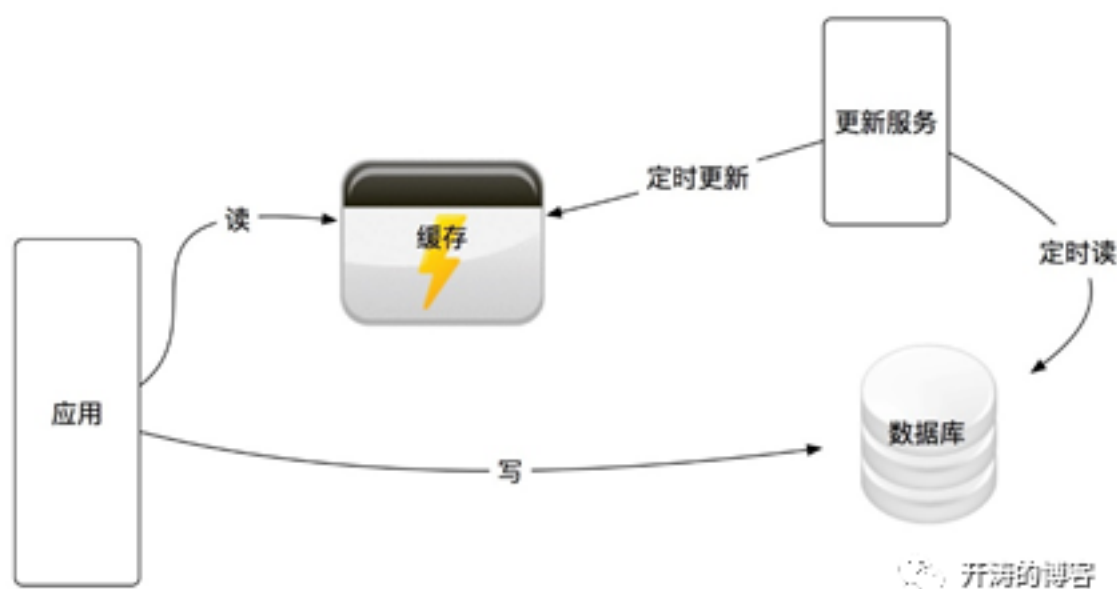


图4-13

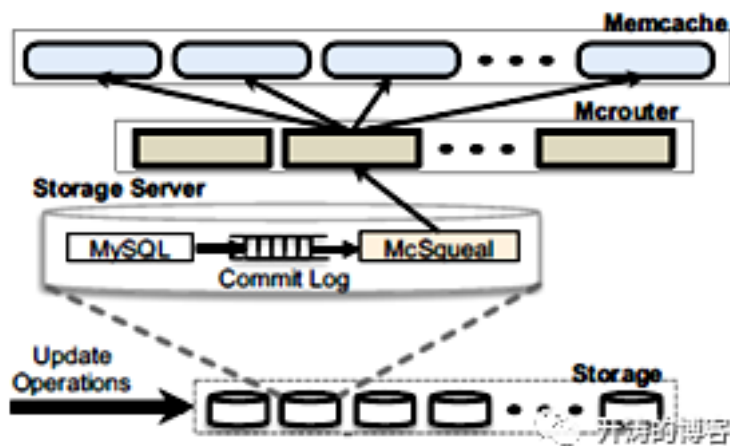


图4-14

这种方法实现起来稍微复杂，增加了更新服务，这里的更新服务需要定时调度任务的设计，请参考第6章的内容，这需要更多的开发和运维成本，在设计异步服务时要充分保证异步服务的可用性，要有完善的监控和报警，否则缓存数据将和数据库不一致。但是在这种模式下性能最好，因为应用层读缓存即可，不需要读取数据库。

3. 串联模式

串联模式的架构如图4-15所示。



图4-15

在这种串联模式下，应用直接在缓存上进行读写操作，缓存作为代理层，根据需求和配置与数据库进行读写操作。

在微服务的设置中并不推荐采用这种服务的串联模式，因为它在应用和数据库中间增加了一层代理层，需要设计和维护这多出的一层，还要保证高可用性，成本较高，但是这种模式有着特殊的场景，比如我们需要在代理层开启缓存加速，例如Varnish等。

4.4.3 分布式缓存分片的三种模式

在互联网行业里，我们做的都是用户端的产品，有调查称中国有6亿互联网用户，这么多的用户会给互联网应用带来了海量的请求，这也需要存储海量的数据，因此，单机的缓存满足不了对海量数据缓存的需求。

我们通常通过多个缓存节点来缓存大量的临时数据，来加速缓存的存取速度，例如，可以把微博的粉丝关系存储在缓存中，在获取某个用户有权限看见的微博时，我们就可以使用这些粉丝关系，粉丝关系的数据量非常大，一个大V用户可能有几千万或者上亿的关注量，可想而知，我们需要多大的内存才能够存储这么多的粉丝关系。

在通用的解决方案中，我们会对这些大数据量进行切片，数据被分成大小相等的分片，一个缓存节点负责存储其中的多个分片。分片通常有三种实现方式，包括客户端分片、代理分片和集群分片。

1．客户端分片

对缓存进行客户端分片的方案如图4-16所示。

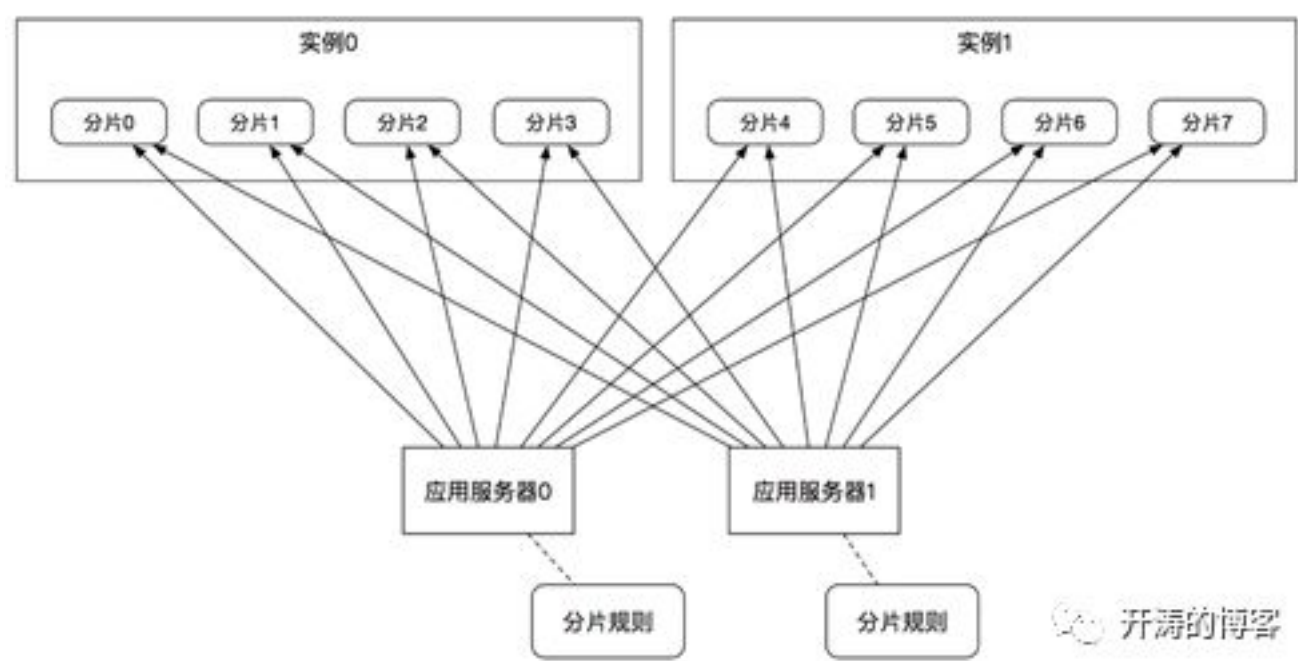


图4-16

客户端分片通过应用层直接操作分片逻辑，分片规则需要在同一个应用的多个节点间保存，每个应用层都嵌入一个操作切片的逻辑实现，一般通过依赖Jar包来实现。笔者曾工作过的几家大的互联网公司都有内部的缓存分片的实现，多数是采用在应用层直接实现的方式，应用层分片的性能更好，实现简单，有问题时容易定位和修复，4.6节介绍的开源项目 `redic` (<https://gitee.com/robertleepeak/redic>) 也是采用这种方案实现的。

这种解决方案的性能很好，实现起来比较简单，适合快速上线，而且切分逻辑是自己开发的，如果在生产上出了问题，则都比较容易解决；但是它侵入了业务逻辑的实现，会让缓存服务器保持的应用程序连接比较多，这要看应用服务器池的节点数量，需要提前进行容量评估，请参考《分布式服务架构：原理、设计与实战》第3章的内容。

2．代理分片

对缓存进行代理分片的方案如图4-17所示。

代理分片就是在应用层和缓存服务器中间增加一个代理层，把分片的路由规则配置在代理层，代理层对外提供与缓存服务器兼容的接口给应用层，应用层的开发人员不用关心分片规则，只需关心业务逻辑的实现，待业务逻辑实现以后，在代理层配置路由规则即可。

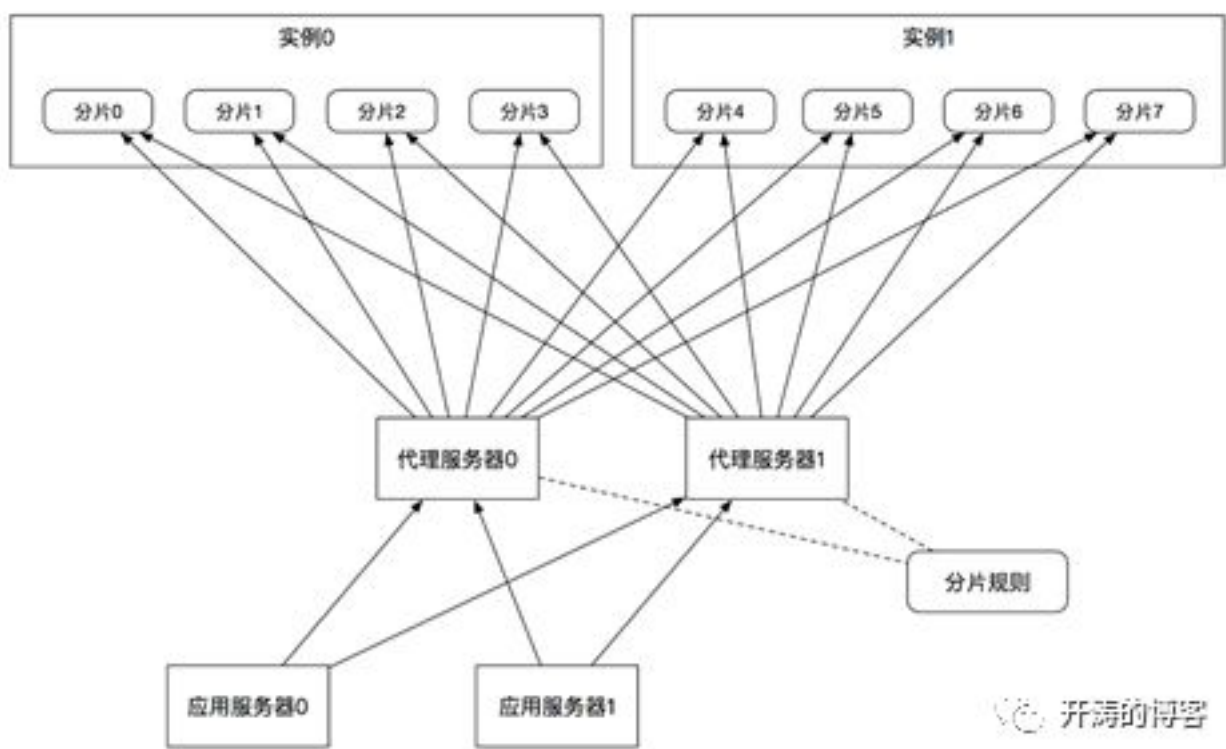


图4-17

这种方案的好处是让应用层开发人员专注于业务逻辑的实现，把缓存分片的配置留给代理层做，具体可以由运维人员来实施；缺点是增加了代理层。尽管代理层是轻量级的转发协议，但是毕竟要实现缓存协议的解析，并通过分片的路由规则来路由请求，对每个缓存操作都增加了一层代理网络传输，对性能是有影响的，对增加的代理层也需要进行维护，也有硬件成本，还要有能够解决Bug的技术专家，成本很高。

流行的Codis框架就是代理分片的典型实现。

3 . 集群分片

缓存自身提供的集群分片方案如图4-18所示。

有的缓存自身提供了集群功能，集群可以实现分片和高可用特性，我们只需要把它们当成一个由多个缓存服务器节点组成的大缓存机器来使用即可，分片和高可用等对应用层是透明的，由运维人员配置即可使用，典型的就是Redis 3.0提供的Cluster。我们已经在4.3.2节介绍了Redis集群。

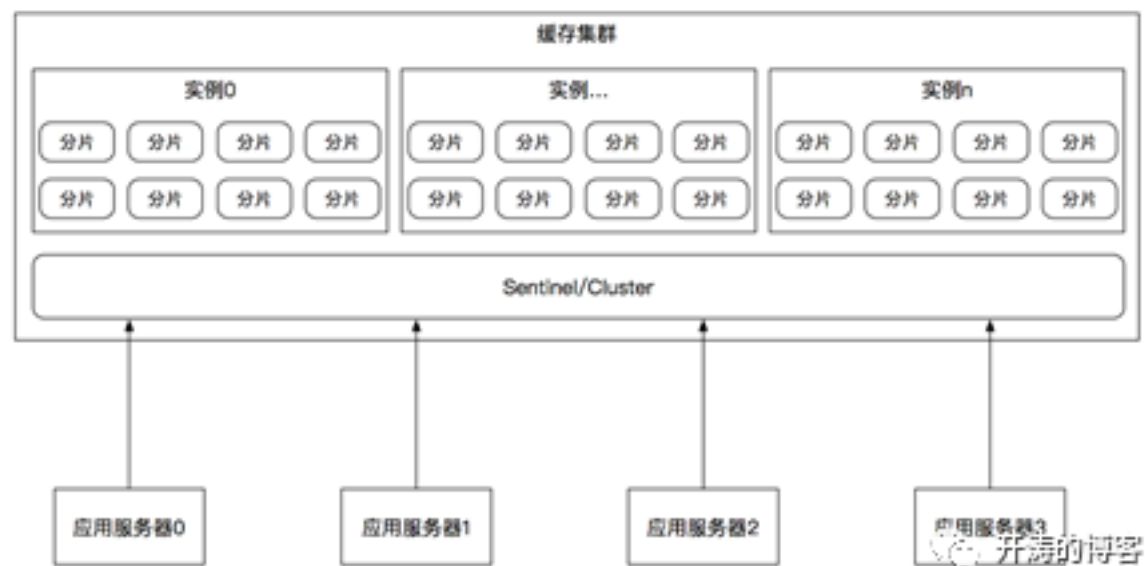


图4-18

4.4.4 分布式缓存的迁移方案

处理分布式缓存迁移是比较困难的，通常我们将其分为平滑迁移和停机迁移。这里讲解通用的迁移方案，扩容实际上是迁移的一种特殊案例，我们在下面学习的方案全部适用。我们会在讲解该方案的过程中，以扩容为例来说明相应的步骤和实现细节。

1 . 平滑迁移

平滑迁移适合对可用性要求较高的场景，例如，线上的交易服务对缓存依赖较大，不能忍受停机带来的业务损失，也没有交易的低峰期，我们对此只能采用平滑迁移的方式。

平滑迁移使用的是双写方案，方案分成4个步骤：双写、迁移历史数据、切读、下双写。

这种方式还有一个变种，就是不需要迁移老数据，在第1步中双写后，在一

定的时间里通过新规则对新缓存进行写入，新缓存已经有了足够的数据，这样我们就不要再迁移旧数据，直接进入第3步即可。

首先，假设我们的应用现在使用了具有两个分片的缓存集群，通过关键字哈希的方式进行路由，如图4-19所示。



图4-19

因为两个分片已经不能满足缓存容量的需求，所以现在需要扩容到4个分片，达到原来两倍的缓存总大小，因此我们需要迁移。

迁移的具体过程如下。

第1步，双写。按照新规则和旧规则同时往新缓存和旧缓存中写数据，如图4-20所示。

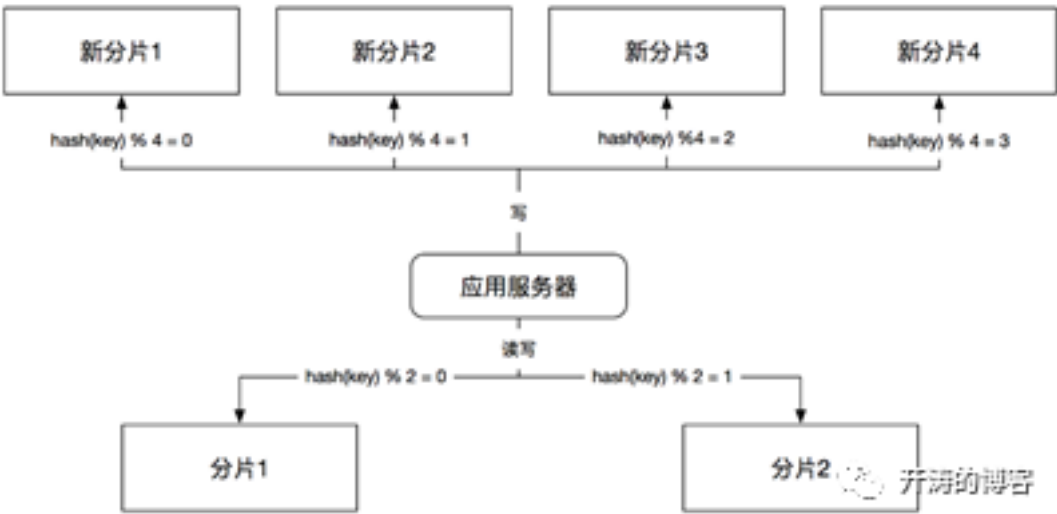


图4-20

这里，我们仍然按照旧的规则，也就是关键字哈希除以2取余来路由分片，同时按照新的规则，也就是关键字哈希除以4取余来路由到新的4个分片上，来完成数据的双写。

这个步骤有优化的空间，因为是在成倍扩容的场景下，所以我们不需要准备

4个全新的分片。新规则中前两个分片的数据，其实是旧规则中两个分片数据的子集，并且规则一致，所以我们可以重用前两个分片，也就是说一共需要两个新的分片，用来处理关键字哈希取余后为2和3的情况；使用旧的缓存分片来处理关键字哈希取余后0和1的情况即可。如图4-21所示。

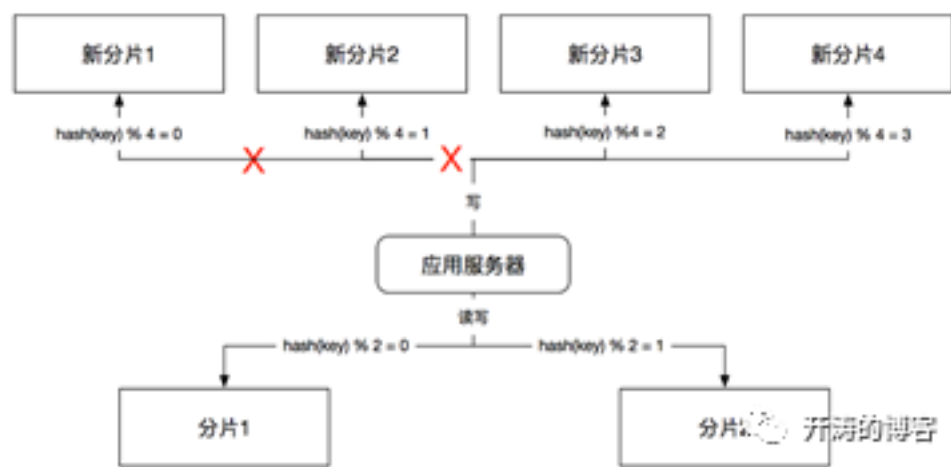


图4-21

第2步，迁移历史数据。把旧缓存集群中的历史数据读取出来，按照新的规则写到新的缓存集群中，如图4-22所示。

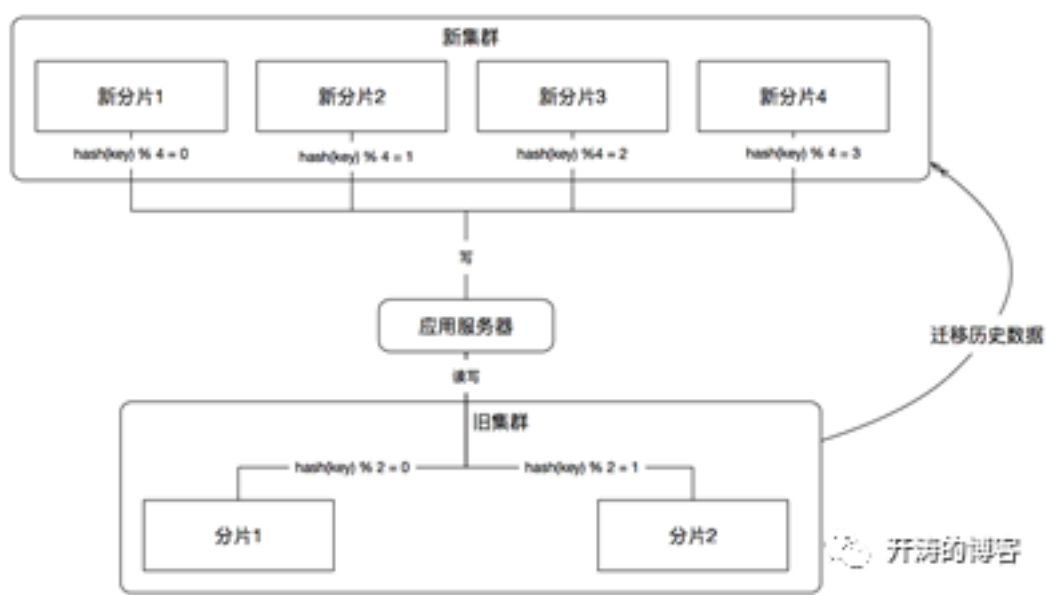


图4-22

在这个过程中，我们需要迁移历史数据，在迁移的过程中可能需要迁移工具，这也需要一部分开发工作量。在迁移后，我们还需要对迁移的数据进行验证，表明我们的数据迁移成功。

在某些应用场景下，缓存数据并不是应用强依赖的，在缓存里获取不到数据，可以回源到数据库获取，因此在这种场景下通过容量评估，数据库可以承受回源导致的压力增加，就可以避免迁移旧数据。在另一种场景下，缓存数据一般是具有时效性的，应用在双写期间不断向新的集群中写入新数据，

历史数据会逐渐过时，并被从旧的集群中删除，在一定的时间流逝后，在新的集群中自然就有了最新的数据，也就不再需要迁移历史数据了，但是这需要进行评估和验证。

第3步，切读。把应用层所有的读操作路由到新的缓存集群上，如图4-23所示。

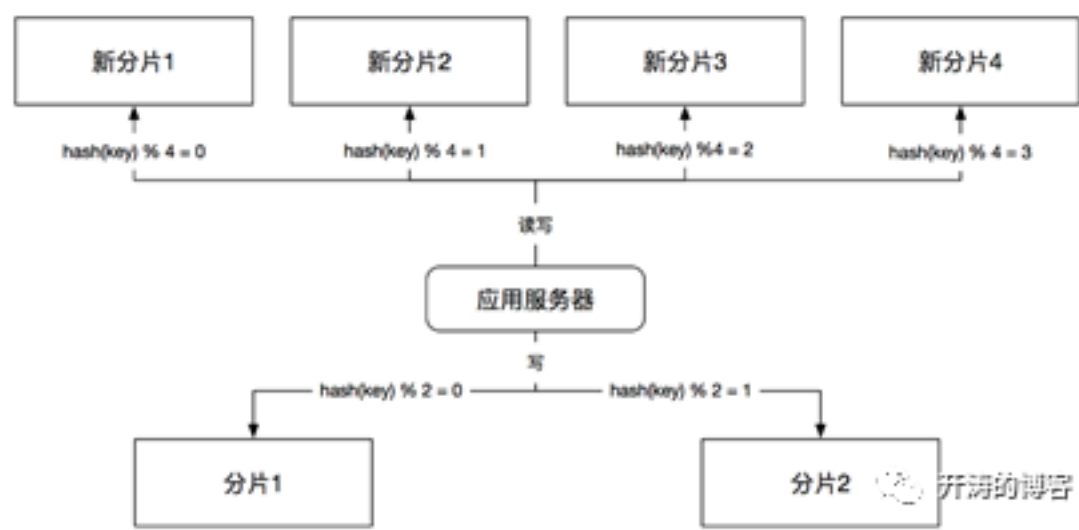


图4-23

这一步把应用中读取的操作的缓存数据源转换成新的缓存集群，这时应用的读写操作已经完全发生在新的数据库集群上了。这一步一般不需要上线代码，我们会在一开始上双写时就实现开关逻辑，这里只需要将读的开关切换到新的集群即可。

第4步，下线双写。把写入旧的集群的逻辑下线，如图4-24所示。

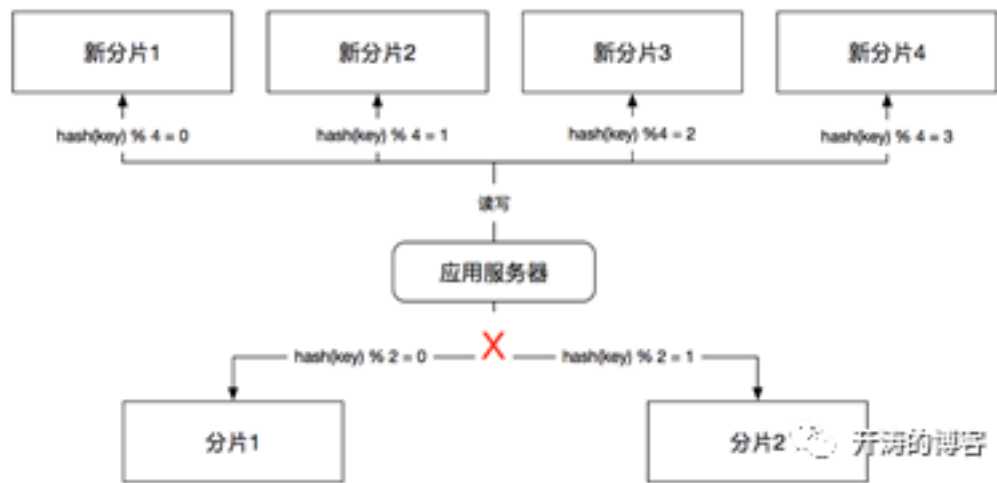


图4-24

这一步通常是在双写和切读后验证没有任何问题，并保证数据一致性的情况下，才把这部分代码下线的。同时可以把旧的分片下线，如果是扩容的场景，并且重用了旧的分片1和分片2，则还可以清理分片1和分片2中的冗余数

据。

2．停机迁移

停机迁移的方法比较简单，通常分为停止应用、迁移历史数据、更改应用的数据源、启动应用这4个步骤，如图4-25所示。

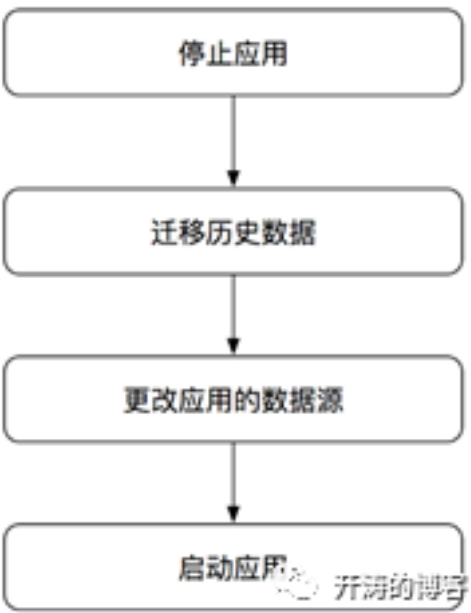


图4-25

具体的迁移步骤如下。

- (1) 停机应用，先将应用停止服务。
- (2) 迁移历史数据，按照新的规则把历史数据迁移到新的缓存集群中。
- (3) 更改应用的数据源配置，指向新的缓存集群。
- (4) 重新启动应用。

这种方式的好处是实现比较简单、高效，能够有效避免数据的不一致，但是需要由业务方评估影响，一般在晚上交易量比较小或者非核心服务的场景下比较适用。

3．一致性哈希

实际上，Redis的客户端Jedis本身实现了基于一致性哈希的客户端路由框架，这种框架的好处是便于动态扩容，当一致性哈希中的节点的负载较高时，我们可以动态地插入更多的节点，来减少已存节点的压力。

一致性哈希算法是在1997年由麻省理工学院的Karger等人在解决分布式缓存问题时提出的一种方案，设计目标是解决网络中的热点问题，后来在分布式系统也得到了广泛应用。研究过Redis和Memcache缓存的人一般都了解一致性哈希算法，他们都在客户端实现了一致性哈希。

一致性哈希的逻辑如图4-26所示。

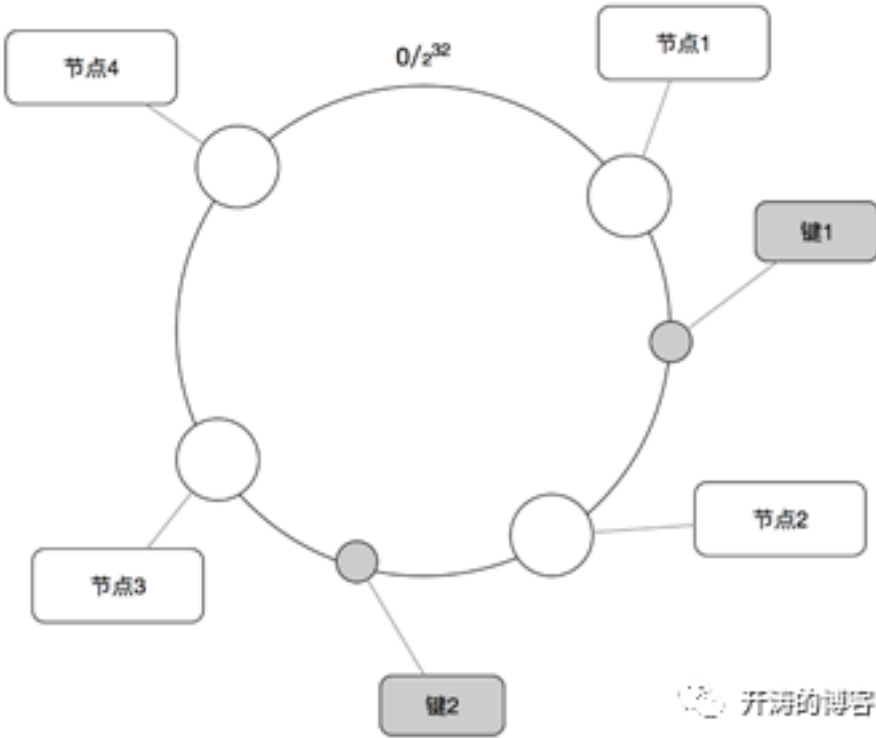


图4-26

在收到访问一个主键的请求后，可通过下面的流程寻找这个主键的存储节点。

- (1) 求出Redis服务器（节点）的哈希值，并将其配置到0-232的圆（continuum）上。
- (2) 采用同样的方法求出存储数据的键的哈希值，并映射到相同的圆上。
- (3) 从数据映射到的位置开始顺时针查找，找到的第1台服务器就是将数据保存的位置。
- (4) 如果在寻找的过程中超过232仍然找不到节点，就会保存到第1台服务器上。

在扩容的场景下添加一台服务器节点5时，只有在圆上增加服务器的位置到逆时针方向的第一台服务器上的键会受到影响，如图4-27所示。

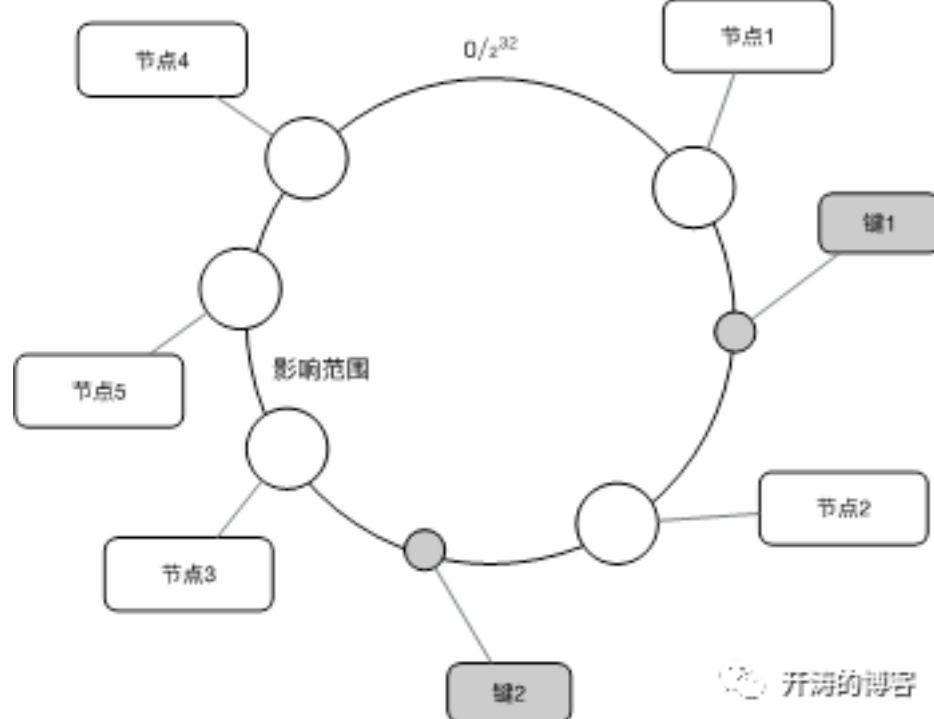


图4-27

我们看到，在节点3和节点4之间增加了节点5，影响范围是节点3到节点5之间的数据，而并不影响其他节点的数据，因此，这为缓存的扩容提供了便利性，当缓存压力增加且缓存容量不够时，我们通常可以通过在线增加节点的方式来完成扩容。

4.4.5 缓存穿透、缓存并发和缓存雪崩

缓存穿透、缓存并发和缓存雪崩是常见的由于并发量大而导致的缓存问题，本节讲解其产生原因和解决方案。

缓存穿透通常是由恶意攻击或者无意造成的；缓存并发是由设计不足造成的；缓存雪崩是由缓存同时失效造成的，三种问题都比较典型，也是难以防范和解决的。本节给出通用的解决方案，以供在缓存设计的过程中参考和使用。

1. 缓存穿透

缓存穿透指的是使用不存在的key进行大量的高并发查询，这导致缓存无法命中，每次请求都要穿透到后端数据库系统进行查询，使数据库压力过大，甚至使数据库服务被压死。

我们通常将空值缓存起来，再次接收到同样的查询请求时，若命中缓存并且值为空，就会直接返回，不会透传到数据库，避免缓存穿透。当然，有时恶意袭击者可以猜到我们使用了这种方案，每次都会使用不同的参数来查询，

这就需要对输入的参数进行过滤，例如，如果我们使用ID进行查询，则可以对ID的格式进行分析，如果不符合产生ID的规则，就直接拒绝，或者在ID上放入时间信息，根据时间信息判断ID是否合法，或者是否是我们曾经生成的ID，这样可以拦截一定的无效请求。

当然，每个设计人员都应该对服务的可用性和健壮性负责，应该建设健壮的服务，让我们的服务像不倒翁一样，因此，我们需要对服务设计限流和熔断等功能，请参考《分布式服务架构：原理、设计与实战》中第1章关于微服务设计模式的内容。

2. 缓存并发

缓存并发的问题通常发生在高并发的场景下，当一个缓存key过期时，因为访问这个缓存key的请求量较大，多个请求同时发现缓存过期，因此多个请求会同时访问数据库来查询最新数据，并且回写缓存，这样会造成应用和数据库的负载增加，性能降低，由于并发较高，甚至会导致数据库被压死。

我们通常有3种方式来解决这个问题。

1) 分布式锁

使用分布式锁，保证对于每个key同时只有一个线程去查询后端服务，其他线程没有获得分布式锁的权限，因此只需要等待即可。这种方式将高并发的压力转移到了分布式锁，因此对分布式锁的考验很大。

2) 本地锁

与分布式锁类似，我们通过本地锁的方式来限制只有一个线程去数据库中查询数据，而其他线程只需等待，等前面的线程查询到数据后再访问缓存。但是，这种方法只能限制一个服务节点只有一个线程去数据库中查询，如果一个服务有多个节点，则还会有多个数据库查询操作，也就是说在节点数量较多的情况下并没有完全解决缓存并发的问题。

3) 软过期

软过期指对缓存中的数据设置失效时间，就是不使用缓存服务提供的过期时间，而是业务层在数据中存储过期时间信息，由业务程序判断是否过期并更新，在发现了数据即将过期时，将缓存的时效延长，程序可以派遣一个线程

去数据库中获取最新的数据，其他线程这时看到延长了的过期时间，就会继续使用旧数据，等派遣的线程获取最新数据后再更新缓存。

也可以通过异步更新服务来更新设置软过期的缓存，这样应用层就不用关心缓存并发的问题了。

3. 缓存雪崩

缓存雪崩指缓存服务器重启或者大量缓存集中在某一个时间段内失效，给后端数据库造成瞬时的负载升高的压力，甚至压垮数据库的情况。

通常的解决办法是对不同的数据使用不同的失效时间，甚至对相同的数据、不同的请求使用不同的失效时间，例如，我们要缓存user数据，会对每个用户的数据设置不同的缓存过期时间，可以定义一个基础时间，假设10秒，然后加上一个两秒以内的随机数，过期时间为10~12秒，就会避免缓存雪崩。

4.4.6 缓存对事务的支持

在使用Redis缓存的业务场景时经常会有这样的需求：要求递减一个变量，如果递减后变量小于等于0，则返回一个标志；如果成功，则返回剩余的值，类似于数据库事务的实现。

在实现中需要注意服务器端的多线程问题及客户端的多线程问题。在服务器端可以利用服务器单线程执行LUA脚本来保证，或者通过WATCH、EXEC、DISCARD、EXEC来保证。

在Redis中支持LUA脚本，由于Redis使用单线程实现，因此我们首先给出LUA脚本的实现方案。在如下代码中，我们看到变量被递减，并判断是否将小于0的操作放到LUA脚本里，利用Redis的单线程执行的特性完成这个原子递减的操作：

```

/**
 * Implemented by LUA. Minus a key by a value, then return the left value.
 * If the left value is less than 0, return -1; if error, return -1.
 *
 * @param key
 *         the key of the redis variable.
 * @param value
 *         the value to minus off.
 * @return the value left after minus. If it is less than 0, return -1; if
 *         error, return -1.
 */
public long decrByUntil0Lua(String key, long value) {
    // If any error, return -1.
    if (value <= 0)
        return -1;

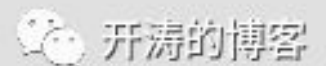
    // The logic is implemented in LUA script which is run in server thread,
    // which is single thread in one server.
    String script = " local leftvalue = redis.call('get', KEYS[1]); "
        + " if ARGV[1] - leftvalue > 0 then return nil; else "
        + " return redis.call('decrby', KEYS[1], ARGV[1]); end; ";

    Long leftValue = (Long) jedis.eval(script, 1, key, "" + value);

    // If the left value is less than 0, return -1.
    if (leftValue == null)
        return -1;

    return leftValue;
}

```



还可以通过Redis对事务的支持方法watch和multi来实现，类似于一个CAS方法的实现，如果对热数据有竞争，则会返回失败，然后重试直到成功：

```

/**
 * Implemented by CAS. Minus a key by a value, then return the left value.
 * If the left value is less than 0, return -1; if error, return -1.
 *
 * No synchronization, because redis client is not shared among multiple
 * threads.
 *
 * @param key
 *         the key of the redis variable.
 * @param value
 *         the value to minus off.
 * @return the value left after minus. If it is less than 0, return -1; if
 *         error, return -1.
 */
public long decrByUntil0Cas(String key, long value) {
    // If any error, return -1.
    if (value <= 0)
        return -1;

    // Start the CAS operations.
    jedis.watch(key);

    // Start the transation.
    Transaction tx = jedis.multi();

```



```

    // Decide if the left value is less than 0, if no, terminate the
    // transation, return -1;
    String curr = tx.get(key).get();
    if (Long.valueOf(curr) - value < 0) {
        tx.discard();
        return -1;
    }

    // Minus the key by the value
    tx.decrBy(key, value);

    // Execute the transaction and then handle the result
    List<Object> result = tx.exec();

    // If error, return -1;
    if (result == null || result.isEmpty()) {
        return -1;
    }

    // Extract the first result
    for (Object rt : result) {
        return Long.valueOf(rt.toString());
    }

    // The program never comes here.
    return -1;
}

```



=====

本次送的是《可伸缩服务架构：框架与中间件》，共5本，规则：抽奖~



开涛

发起了一个抽奖活动



奖品: 可伸缩服务架构： 框架与中间件
× 5

05月07日 08:00 自动开奖



长按识别小程序，参与抽奖

本文节选自《可伸缩服务架构：框架与中间件》，喜欢的朋友可以扫描下方二维码购买



更多精彩内容请关注公众号：开涛的博客~



开明博客