# SpringBoot&Redis&Lettuce

张洋

目录

**[一：Redis简介](#_Toc24487_WPSOffice_Level1)** **[1](#_Toc24487_WPSOffice_Level1)**

[1.1 Redis 优势](#_Toc29319_WPSOffice_Level2) [2](#_Toc29319_WPSOffice_Level2)

[1.2 Redis与其他key-value存储有什么不同？](#_Toc14459_WPSOffice_Level2) [2](#_Toc14459_WPSOffice_Level2)

[1.3数据结构简介](#_Toc16031_WPSOffice_Level2) [2](#_Toc16031_WPSOffice_Level2)

[1.4 Redis使用场景](#_Toc10598_WPSOffice_Level2) [8](#_Toc10598_WPSOffice_Level2)

[1.5 Redis 单线程架构](#_Toc8685_WPSOffice_Level2) [9](#_Toc8685_WPSOffice_Level2)

**[二：Spring Data Redis Lettuce](#_Toc29319_WPSOffice_Level1)** **[13](#_Toc29319_WPSOffice_Level1)**

[2.1 Lettuce简介](#_Toc15475_WPSOffice_Level2) [13](#_Toc15475_WPSOffice_Level2)

[2.2 客户端的创建](#_Toc5245_WPSOffice_Level2) [13](#_Toc5245_WPSOffice_Level2)

[2.3 单机数据库](#_Toc14781_WPSOffice_Level2) [15](#_Toc14781_WPSOffice_Level2)

[2.4 多机数据库](#_Toc5460_WPSOffice_Level2) [16](#_Toc5460_WPSOffice_Level2)

**[Spring Data Cache](#_Toc14459_WPSOffice_Level1)** **[19](#_Toc14459_WPSOffice_Level1)**



# 一：Redis简介

Redis 是完全开源免费的，遵守BSD协议，是一个高性能的key-value数据库。

Redis 与其他 key - value 缓存产品有以下三个特点：Redis支持数据的持久化，可以将内存中的数据保持在磁盘中，重启的时候可以再次加载 进行使用。Redis不仅仅支持简单的key-value类型的数据，同时还提供list，set，zset，hash等数据结构的存储。Redis支持数据的备份，即master-slave模式的数据备份。

### 1.1 Redis 优势

性能极高 – Redis能读的速度是110000次/s,写的速度是81000次/s 。丰富的数据类型 – Redis支持二进制案例的 Strings, Lists, Hashes, Sets 及 Ordered Sets 数据类型操作。

原子 – Redis的所有操作都是原子性的，同时Redis还支持对几个操作全并后的原子性执行。

丰富的特性 – Redis还支持 publish/subscribe, 通知, key 过期等等特性。

### 1.2 Redis与其他key-value存储有什么不同？

Redis有着更为复杂的数据结构并且提供对他们的原子性操作，这是一个不同于其他数据库的进化路径。Redis的数据类型都是基于基本数据结构的同时对程序员透明，无需进行额外的抽象。

Redis运行在内存中但是可以持久化到磁盘，所以在对不同数据集进行高速读写时需要权衡内存，应为数据量不能大于硬件内存。在内存数据库方面的另一个优点是， 相比在磁盘上相同的复杂的数据结构，在内存中操作起来非常简单，这样Redis可以做很多内部复杂性很强的事情。 同时，在磁盘格式方面他们是紧凑的以追加的方式产生的，因为他们并不需要进行随机访问。

### 1.3数据结构简介

Redis支持五种数据类型：string（字符串），hash（哈希），list（列表），set（集合）及zset(sorted set：有序集合)。

##### 1.3.1 String（字符串）

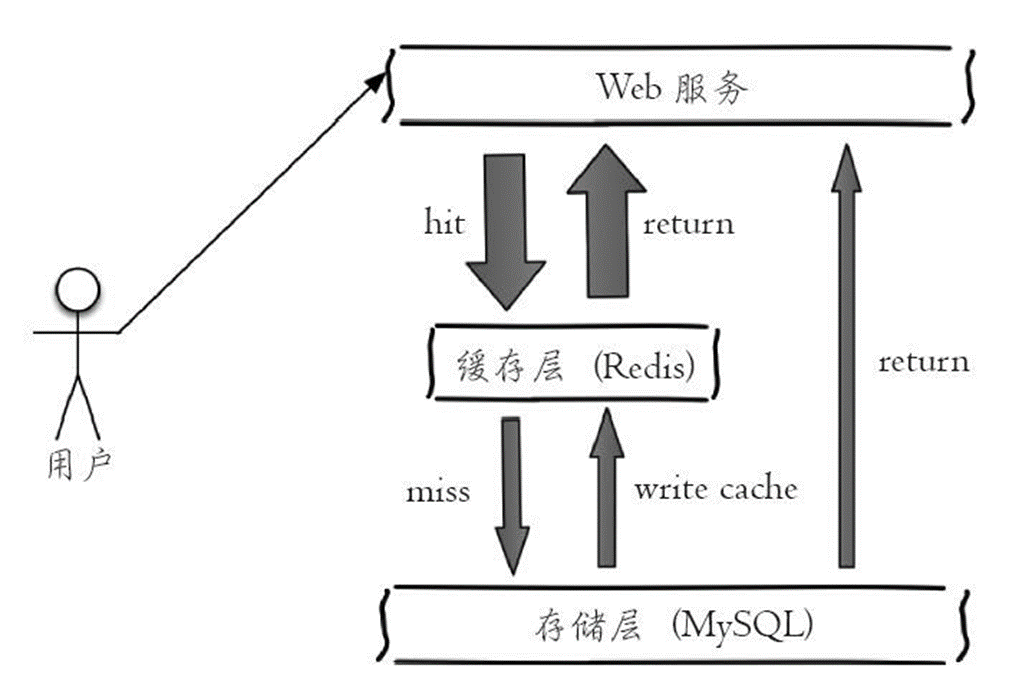
* string是redis最基本的类型，你可以理解成与Memcached一模一样的类型，一个key对应一个value。
* string类型是二进制安全的。意思是redis的string可以包含任何数据。比如jpg图片或者序列化的对象 。
* string类型是Redis最基本的数据类型，一个键最大能存储512MB。

最大字符串为512M，但是大字符串非常不建议

应用场景

1.1缓存功能

下图　Redis + MySQL组成的缓存存储架构



1.2计数

许多应用都会使用Redis作为计数的基础工具，它可以实现快速计数、查询缓存的功能，同时数据可以异步落地到其他数据源。

1.3共享session

如图2-11所示，一个分布式Web服务将用户的Session信息（例如用户登

录信息）保存在各自服务器中，这样会造成一个问题，出于负载均衡的考虑，分布式服务会将用户的访问均衡到不同服务器上，用户刷新一次访问可能会发现需要重新登录，这个问题是用户无法容忍的。

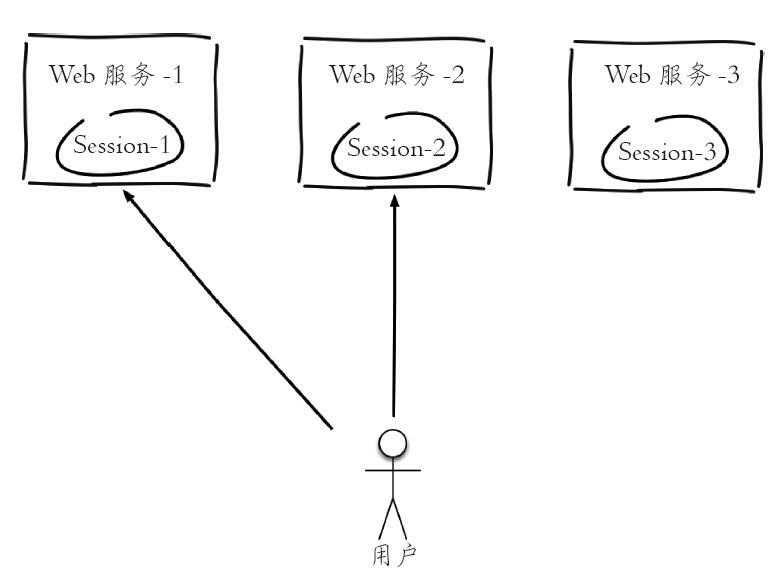


图2-11　Session分散管理

为了解决这个问题，可以使用Redis将用户的Session进行集中管理，如

图2-12所示，在这种模式下只要保证Redis是高可用和扩展性的，每次用户更新或者查询登录信息都直接从Redis中集中获取。

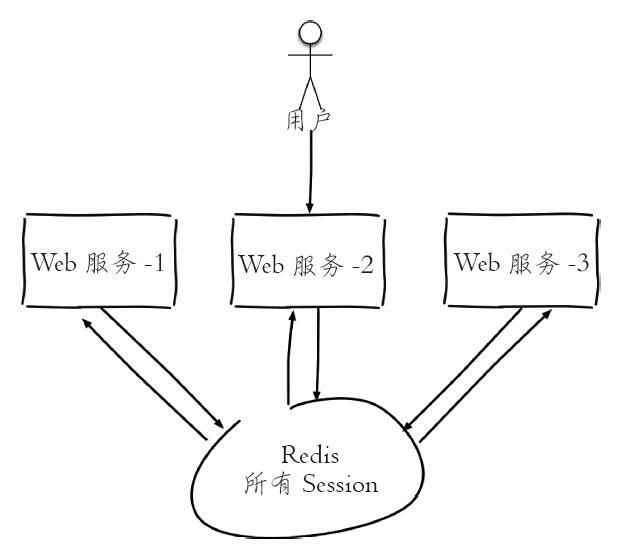


图2-12　Redis集中管理Session

1.4.限速（短信频率）

##### 1.3.2 Hash（哈希）

* Redis hash 是一个键值对集合。
* Redis hash是一个string类型的field和value的映射表，**hash特别适合用于存储对象。**

应用场景

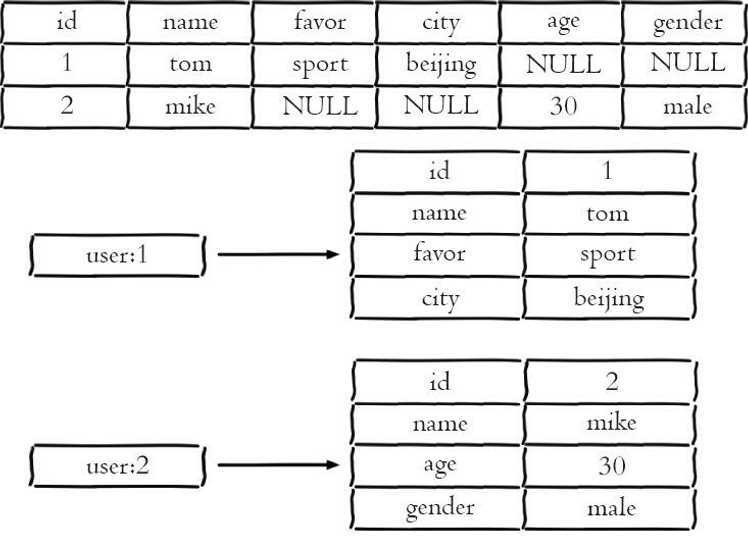
用户的属性作为表的列，每条用户信息作为行。

// 将userInfo变为映射关系使用hmset保存到Redis中

redis.hmset(userRedisKey, transferUserInfoToMap(userInfo));

// 添加过期时间

redis.expire(userRedisKey, 3600);



##### 1.3.3 List（列表）

* Redis 列表是简单的字符串列表，按照插入顺序排序。你可以添加一个元素到列表的头部（左边）或者尾部（右边）。

应用场景

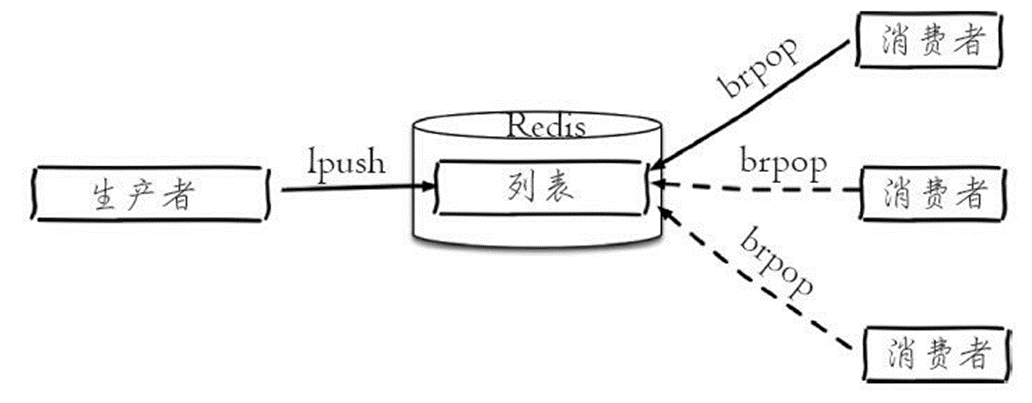
1. 消息队列

lpush+brpop命令组合即可实现阻塞队列，生产

者客户端使用lrpush从列表左侧插入元素，多个消费者客户端使用brpop命令阻塞式的“抢”列表尾部的元素，多个客户端保证了消费的负载均衡和高可用性。

1. 文章列表

每个用户有属于自己的文章列表，现需要分页展示文章列表。此时可以考虑使用列表，因为列表不但是有序的，同时支持按照索引范围获取元素。



##### 1.3.4 Set（集合）

* Redis的Set是string类型的无序集合。
* 集合是通过哈希表实现的，所以添加，删除，查找的复杂度都是O(1)。

应用场景

集合类型比较典型的使用场景是标签（tag）。例如一个用户可能对娱

乐、体育比较感兴趣，另一个用户可能对历史、新闻比较感兴趣，这些兴趣点就是标签。有了这些数据就可以得到喜欢同一个标签的人，以及用户的共同喜好的标签，这些数据对于用户体验以及增强用户黏度比较重要。例如一个电子商务的网站会对不同标签的用户做不同类型的推荐，比如对数码产品比较感兴趣的人，在各个页面或者通过邮件的形式给他们推荐最新的数码产品，通常会为网站带来更多的利益。

1. 给用户添加标签

sadd user:1:tags tag1 tag2 tag5 sadd user:2:tags tag2 tag3 tag5

...

sadd user:k:tags tag1 tag2 tag4

...

1. 给标签添加用户

sadd tag1:users user:1 user:3 sadd tag2:users user:1 user:2 user:3

...

sadd tagk:users user:1 user:2

##### 1.3.5 zset(sorted set：有序集合)

* Redis zset 和 set 一样也是string类型元素的集合,且不允许重复的成员。
* 不同的是每个元素都会关联一个double类型的分数。redis正是通过分数来为集合中的成员进行从小到大的排序。
* zset的成员是唯一的,但分数(score)却可以重复。

应用场景

有序集合比较典型的使用场景就是排行榜系统。例如视频网站需要对用户上传的视频做排行榜，榜单的维度可能是多个方面的：按照时间、按照播放数量、按照获得的赞数。本节使用赞数这个维度，记录每天用户上传视频的排行榜。主要需要实现以下4个功能。

1. 添加用户赞数 （如老席上传朋友圈一开始获得3赞）

zadd user:ranking:2016\_03\_15 Mr.xi 3

如果之后再获得一个赞，可以使用zincrby：

Zincrby user:ranking:2016\_03\_15 Mr.xi 1

1. 取消用户赞数

zrem user:ranking:2016\_03\_15 Mr.xi

1. 展示获取赞数最多的十个用户

zrevrangebyrank user:ranking:2016\_03\_15 0 9

### 1.4 Redis使用场景

##### Redis 可以用来做什么？

###### 1 缓存

缓存机制几乎所有的大型网站都会有，合理的使用可以加快数据的访问速度，而且有效的降低后端的数据访问压力。它提供了过期键处理的机制设置，并且灵活控制最大内存和内存溢出淘汰策略。所以良好合理的缓存设计可以为网站的稳定提供较高的保障。

###### 2排行榜系统

排行榜系统几乎所有网站都会有涉及，例如按照热度，时间维度，各种复杂的设计维度进行排行，Redis提供了列表和有序集合的数据结构，有效的使用可以方便的构建很多排行需求。

###### 3 计数

计数器在网站中至关重要，例如视频网站有播放数，电商网站有浏览数，为了保障数据的实时性，会采用每次操作加1计数，如果并发量很大对关系型数据库是一种挑战，Redis天然的支持计数的功能，而且性能非常好。

###### 4 社交网站

赞/踩，共同好友/喜好，下拉/刷新，粉丝，推送等社交必备功能，在访问量较大的情况下，而传统的关系型数据库不适合这种数据类型，Redis的数据结构相对容易实现这些功能

###### 5 消息队列

很多应用网站都会使用消息队列，因为其具有业务解耦，非实时业务削峰

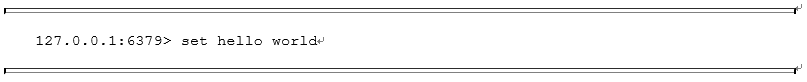
### 1.5 Redis 单线程架构

Redis使用了单线程架构和I/O多路复用模型来实现高性能的内存数据库服务。**理解单线程模型是使用和运维Redis的关键。**

##### 1.5.1引出单线程模型

现在开启了三个redis-cli客户端同时执行命令。

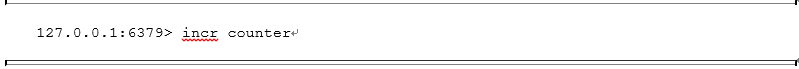
客户端1设置一个字符串键值对：



客户端2对counter做自增操作：



客户端3对counter做自增操作：



其中第2步是重点要讨论的，因为Redis是单线程来处理命令的，所以一条命令从客户端达到服务端不会立刻被执行，所有命令都会进入一个队列中，然后逐个被执行。所以上面3个客户端命令的执行顺序是不确定的（如图2-4所示），但是可以确定不会有两条命令被同时执行（如图2-5所示），所以两条incr命令无论怎么执行最终结果都是2，不会产生并发问题，这就是Redis单线程的基本模型。但是像发送命令、返回结果、命令排队肯定不

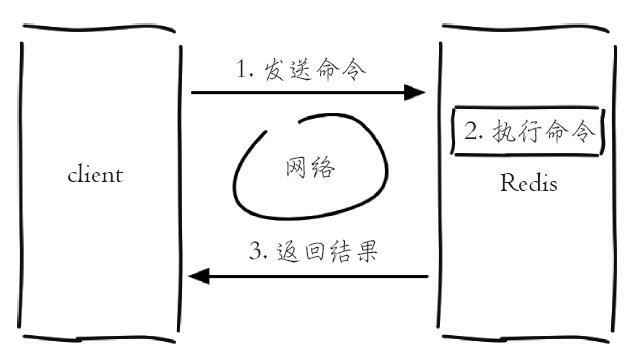
像描述的这么简单，Redis使用了I/O多路复用技术来解决I/O的问题，下一节将进行介绍。

图2-3　Redis客户端与服务端请求过程

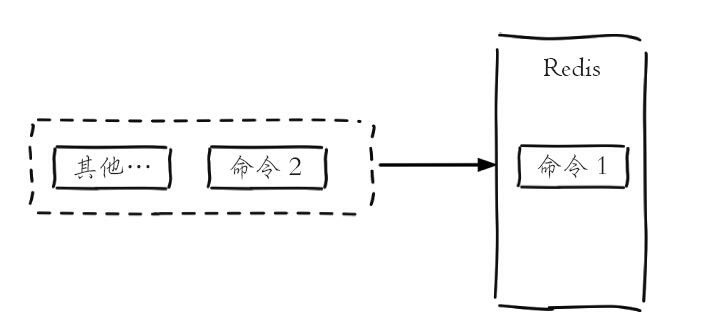


图2-4　所有命令在一个队列里排队等待被执行

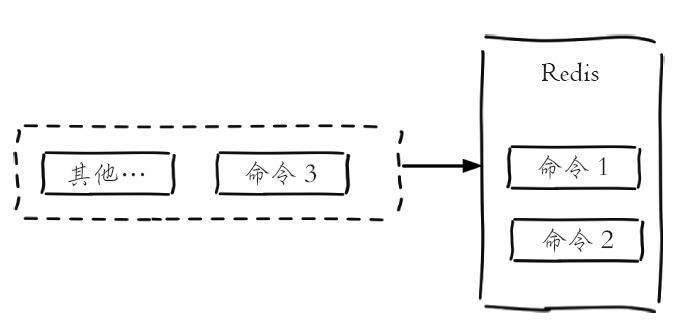


图2-5　不存在多个命令被同时执行的情况

2.为什么单线程还能这么快

通常来讲，单线程处理能力要比多线程差，例如有10000斤货物，每辆车的运载能力是每次200斤，那么要50次才能完成，但是如果有50辆车，只要安排合理，只需要一次就可以完成任务。那么为什么Redis使用单线程模型会达到每秒万级别的处理能力呢？可以将其归结为三点：

第一，纯内存访问，Redis将所有数据放在内存中，内存的响应时长大约为100纳秒，这是Redis达到每秒万级别访问的重要基础。

第二，非阻塞I/O，Redis使用epoll作为I/O多路复用技术的实现，再加上Redis自身的事件处理模型将epoll中的连接、读写、关闭都转换为事件，不在网络I/O上浪费过多的时间，如图2-6所示。（备注：epoll是[Linux内核](https://baike.baidu.com/item/Linux%E5%86%85%E6%A0%B8" \t "https://baike.baidu.com/item/epoll/_blank)为处理大批量[文件描述符](https://baike.baidu.com/item/%E6%96%87%E4%BB%B6%E6%8F%8F%E8%BF%B0%E7%AC%A6/9809582" \t "https://baike.baidu.com/item/epoll/_blank)而作了改进的poll，是Linux下多路复用IO接口select/poll的增强版本，它能显著提高程序在大量[并发连接](https://baike.baidu.com/item/%E5%B9%B6%E5%8F%91%E8%BF%9E%E6%8E%A5/3763280" \t "https://baike.baidu.com/item/epoll/_blank)中只有少量活跃的情况下的系统CPU利用率。）

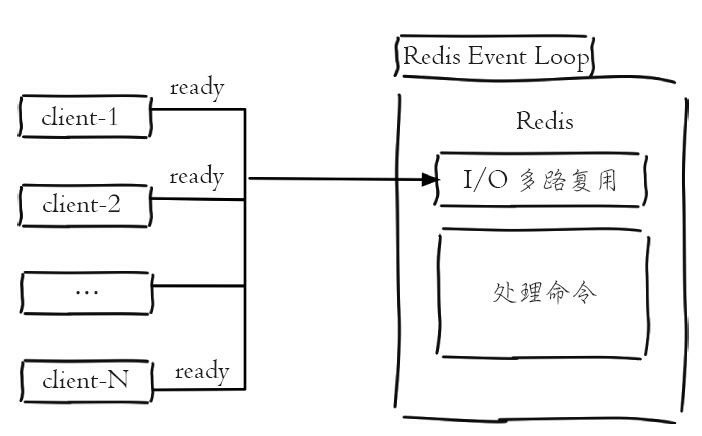


图2-6　Redis使用IO多路复用和自身事件模型

第三，单线程避免了线程切换和竞态产生的消耗。

既然采用单线程就能达到如此高的性能，那么也不失为一种不错的选

择，因为单线程能带来几个好处：第一，单线程可以简化数据结构和算法的实现。如果对高级编程语言熟悉的读者应该了解并发数据结构实现不但困难而且开发测试比较麻烦。第二，单线程避免了线程切换和竞态产生的消耗，对于服务端开发来说，锁和线程切换通常是性能杀手。

但是单线程会有一个问题：对于每个命令的执行时间是有要求的。如果

某个命令执行过长，会造成其他命令的阻塞，对于Redis这种高性能的服务来说是致命的，所以Redis是面向快速执行场景的数据库。



# 二：Spring Data Redis Lettuce

### 2.1 Lettuce简介

1.Lettuce是一个完全无阻塞的 [Redis](https://redis.io/)客户端，使用[netty](https://netty.io/)构建，提供[响应](https://projectreactor.io/)，异步和同步数据访问；

2.基于java8的丰富了[RedisFuture（CompletionStage）](http://redis.paluch.biz/docs/api/releases/latest-4.x/com/lambdaworks/redis/RedisFuture.html)和Reactive类型[Flux [N]](https://projectreactor.io/docs/core/release/api/reactor/core/publisher/Flux.html)和[Mono [0 | 1]的](https://projectreactor.io/docs/core/release/api/reactor/core/publisher/Mono.html)异步API 等新特性；

3.在非阻塞i/o特性上提供了低延迟通信，支持 背压的 NIO TCP 网络引擎，epoll TCP和Unix域套接字。完全支持[Reactive](https://projectreactor.io/) Streaming。

4. Redis Standalone，Master / Slave，Redis Sentinel和Redis Cluster。Lettuce与Redis原生支持的所有操作模型相连。分区容错，读取从节点和传输级安全性为高度可伸缩的应用程序提供了必要的基础。

5.动态api：Lettuce的动态Redis命令接口利用动态模块API和自定义API接口。

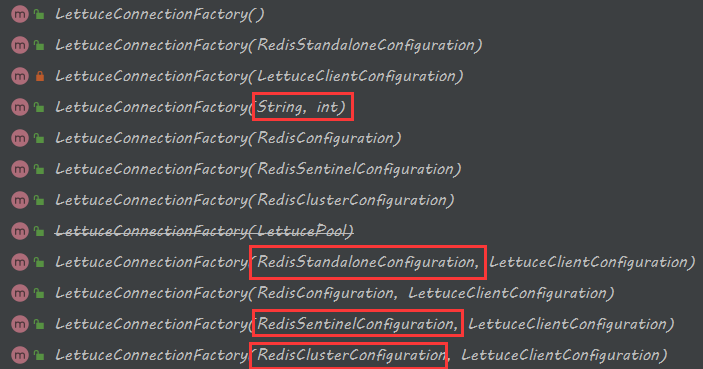
### 2.2 Lettuce客户端的创建

##### 2.2.1客户端版本

Spring-boot-starter-data-redis 2.1.5 当中包含了lettuce-core-io 5.1.6（最新是5.1.7）

##### 2.2.2 创建Lettuce客户端连接

RedisConnection为Redis通信提供核心构建块，它处理与Redis后端的通信。它还会自动将底层连接库异常转换为Spring的一致DAO异常[层次结构](https://docs.spring.io/spring/docs/5.1.7.RELEASE/spring-framework-reference/data-access.html" \l "dao-exceptions)，**RedisConnection通过创建活动对象RedisConnectionFactory。下面是源码中的构建器，提供多种连接Redis服务的方式。**



最简单方法RedisConnectionFactory是通过IoC容器配置适当的连接器并将其注入using类，看下示例

/\*\*\*

\* 集群配置

\* @return

\*/

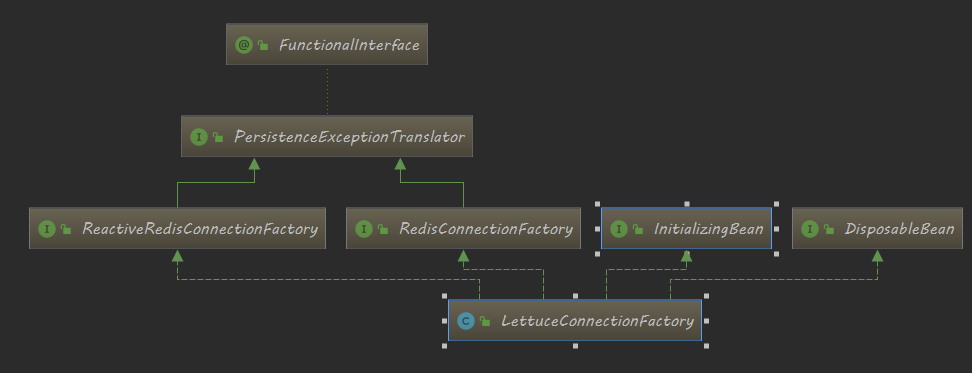
@Bean

public RedisConnectionFactory lettuceClusterConnectionFactory() {

RedisClusterConfiguration redisClusterConfiguration = new RedisClusterConfiguration(clusterProperties.getNodes());

return new LettuceConnectionFactory(redisClusterConfiguration);

}



/\*\*\*

\* RedisTemplate启用

\* @param redisConnectionFactory

\* @return

\*/

@Bean

Public RedisTemplate<String, Serializable> redisCacheTemplate(LettuceConnectionFactory redisConnectionFactory) {

RedisTemplate<String, Serializable> template = new RedisTemplate<>();

template.setKeySerializer(new StringRedisSerializer());

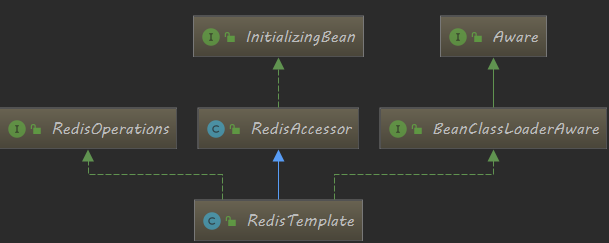
template.setValueSerializer(new GenericJackson2JsonRedisSerializer());

template.setConnectionFactory(redisConnectionFactory);

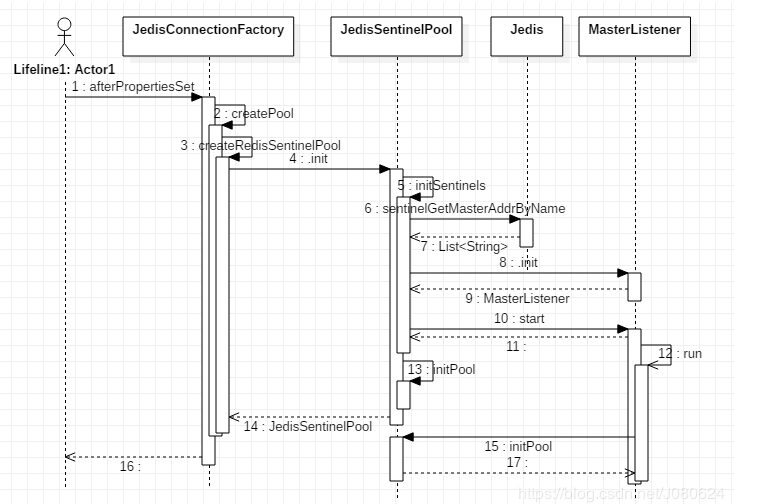
template.setEnableTransactionSupport(true);

return template;

}



对比jedis和lettuce 在哨兵模式下的连接



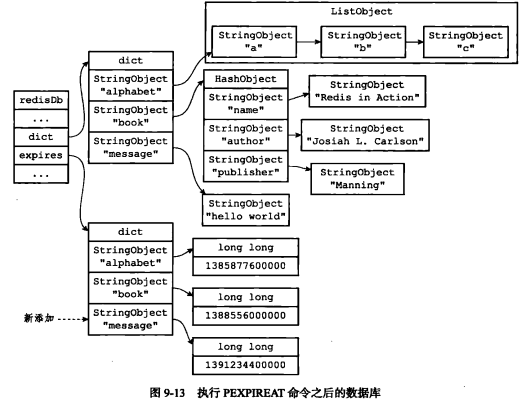
### 2.3 Redis应用特性

##### 2.3.1 设置键的生存时间或者过期时间

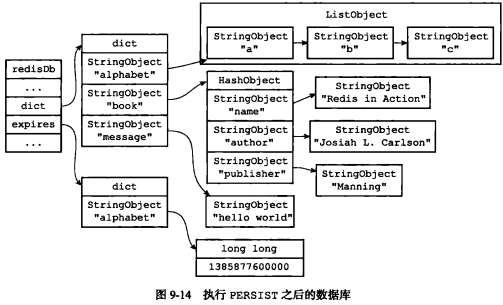
Redis 有四个不同的命令可以用于设置键的生存时间(键可以存在多久)或过期时间(键什么时候被删除):

* EXPIRE <KEY> <TTL> : 将键的生存时间设为 ttl 秒
* PEXPIRE <KEY> <TTL> :将键的生存时间设为 ttl 毫秒
* EXPIREAT <KEY> <timestamp> :将键的过期时间设为 timestamp 所指定的秒数时间戳
* PEXPIREAT <KEY> <timestamp>: 将键的过期时间设为 timestamp 所指定的毫秒数时间戳.

举例：



紧接着移除book和message的过期时间



##### 2.3.2 过期时间的删除策略

从前面介绍了设置过期时间然后保存在过期时间字典中，并且用TTL查看过期时间，那么删除键的策略有哪些？

1. 定时删除：在建立过期时间键时，立即建立一个Timer，在这个键要过期时立刻删除。（主动）
2. 惰性删除：在建立键之后，每次获取键值是来判断这个这个键是否过期，过期删除，否就返回。（被动）
3. 定时删除：定时每隔一段时间就去过期键字典中找到过期的键进行删除（主动）

总结：定时删除是时间换空间，占cpu的内存太多时间，影响服务器响应时间和吞吐量

惰性删除与前者相反，对cpu友好，它对内存是最不友好的： 如果一个键已经过期， 而这个键又仍然保留在数据库中，最终redis服务器采用：**惰性删除加上定期删除， 这两个策略相互配合，可以很好地在合理利用 CPU 时间和节约内存空间之间取得平衡。**

**附加：在使用redis设置值过程中带上过期时间很有必要，如下**

redisTemplate.opsForValue().set("one", "1",1l ,TimeUnit.MINUTES);

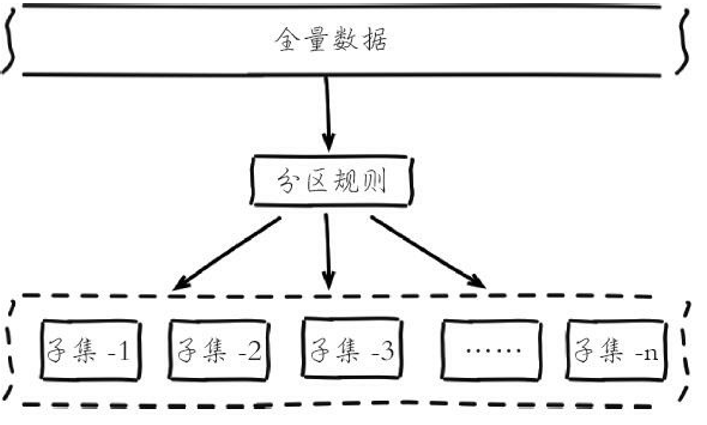
### 2.4 多机数据库

#### 2.4.1 数据分布

##### 2.4.1.1 数据分布理论

分布式数据库首先要解决把整个数据集按照分区规则映射到多个节点的问题，即把数据集划分到多个节点上，每个节点负责整体数据的一个子集。如图10-1所示。

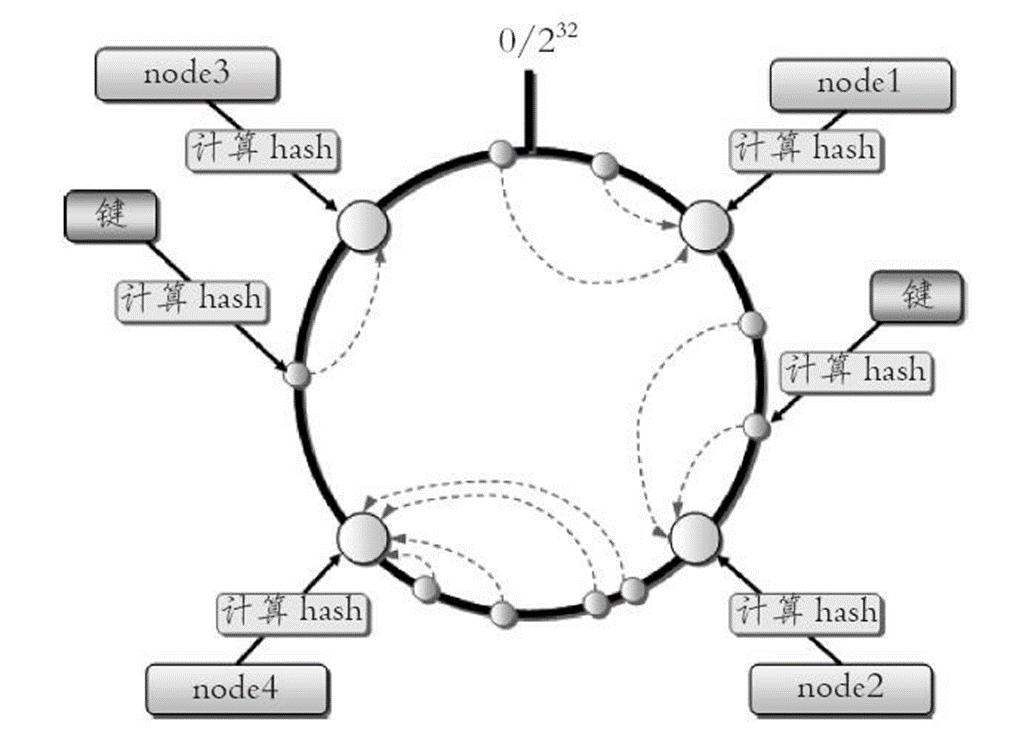
需要重点关注的是数据分区规则。常见的分区规则有哈希分区和顺序分区两种。





##### 2.4.1.2 一致性哈希分区

一致性哈希分区（Distributed Hash Table）实现思路是为系统中每个节点分配一个token，范围一般在0~232，这些token构成一个哈希环。数据读写执行节点查找操作时，先根据key计算hash值，然后顺时针找到第一个大于等于该哈希值的token节点，如图



这种方式相比节点取余最大的好处在于加入和删除节点只影响哈希环中相邻的节点，对其他节点无影响。但一致性哈希分区存在几个问题：

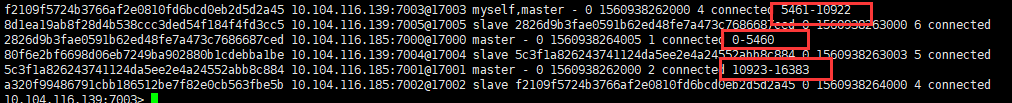
·加减节点会造成哈希环中部分数据无法命中，需要手动处理或者忽略这部分数据，因此一致性哈希常用于缓存场景。

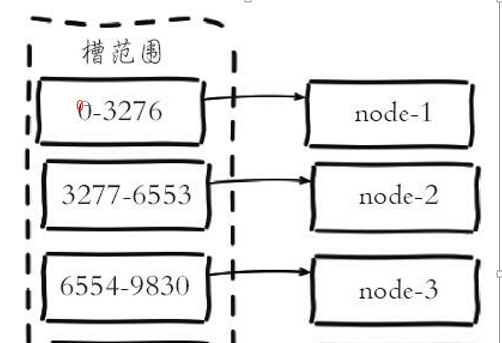
·当使用少量节点时，节点变化将大范围影响哈希环中数据映射，因此这种方式不适合少量数据节点的分布式方案。

·普通的一致性哈希分区在增减节点时需要增加一倍或减去一半节点才能保证数据和负载的均衡。

##### 2.4.1.3 Redis数据分区

Redis Cluser采用虚拟槽分区，所有的键根据哈希函数映射到0~16383整数槽内，计算公式：slot=CRC16（key）&16383。每一个节点负责维护一部分槽以及槽所映射的键值数据



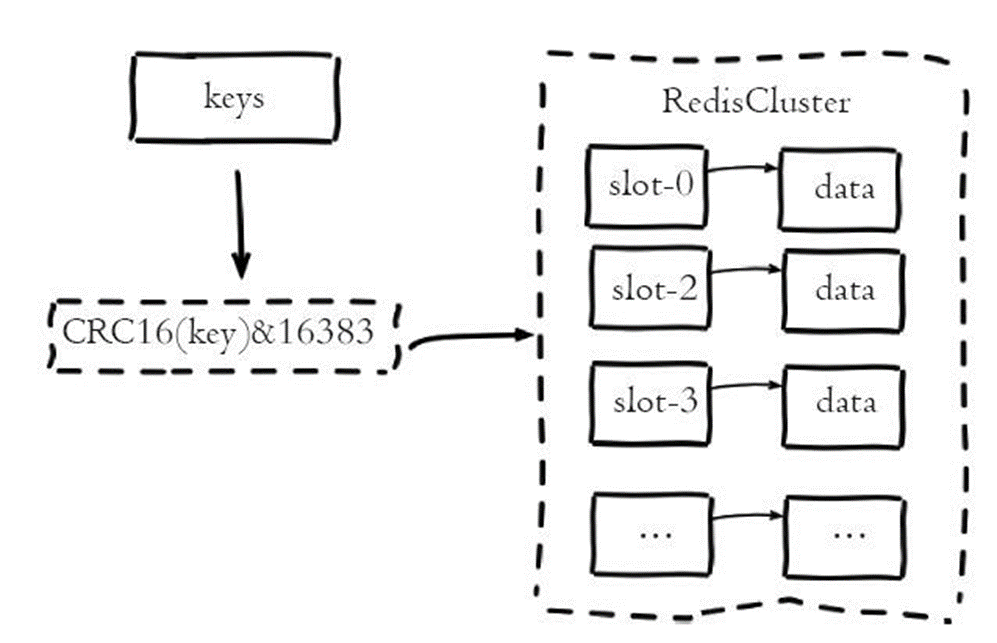


10923-16383

5461-10922

0-5460

**使用CRC16（key）&16383将键映射到槽上**



Redis虚拟槽分区的特点：

* 解耦数据和节点之间的关系，简化了节点扩容和收缩难度。
* 节点自身维护槽的映射关系，不需要客户端或者代理服务维护槽分区元数据。
* 支持节点、槽、键之间的映射查询，用于数据路由、在线伸缩等场景。
* 数据分区是分布式存储的核心，理解和灵活运用数据分区规则对于掌握Redis Cluster非常有帮助

##### 2.4.1.4 集群功能限制

Redis集群相对单机在功能上存在一些限制，需要开发人员提前了解，在使用时做好规避。限制如下：

* key批量操作支持有限。如mset、mget，目前只支持具有相同slot值的 key执行批量操作。对于映射为不同slot值的key由于执行mget、mget等操作可能存在于多个节点上因此不被支持。
* **key事务操作支持有限。同理只支持多key在同一节点上的事务操作，当多个key分布在不同的节点上时无法使用事务功能**。
* key作为数据分区的最小粒度，因此不能将一个大的键值对象如 hash、list等映射到不同的节点。
* 不支持多数据库空间。单机下的Redis可以支持16个数据库，集群模式下只能使用一个数据库空间，即db0。
* 复制结构只支持一层，从节点只能复制主节点，不支持嵌套树状复制结构。

#### Redis性能

##### Pipeline

Redis客户端执行一条命令分为如下四个过程：

1发送命令----》2命令排队------》3命令执行------》4返回结果

其中1到4称为Round Trip Time（RTT，往返时间）。

Redis提供了批量操作命令（例如mget、mset等），有效地节约RTT。但

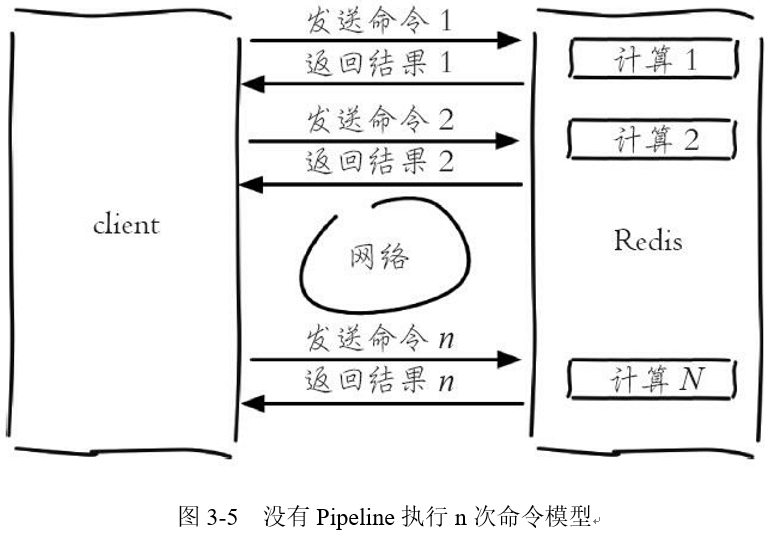
大部分命令是不支持批量操作的，例如要执行n次hgetall命令，并没有

mhgetall命令存在，需要消耗n次RTT。Redis的客户端和服务端可能部署在不同的机器上。例如客户端在北京，Redis服务端在上海，两地直线距离约为

1300公里，那么1次RTT时间=1300×2/（300000×2/3）=13毫秒（光在真空中传输速度为每秒30万公里，这里假设光纤为光速的2/3），那么客户端在1秒内大约只能执行80次左右的命令，这个和Redis的高并发高吞吐特性背道而驰。

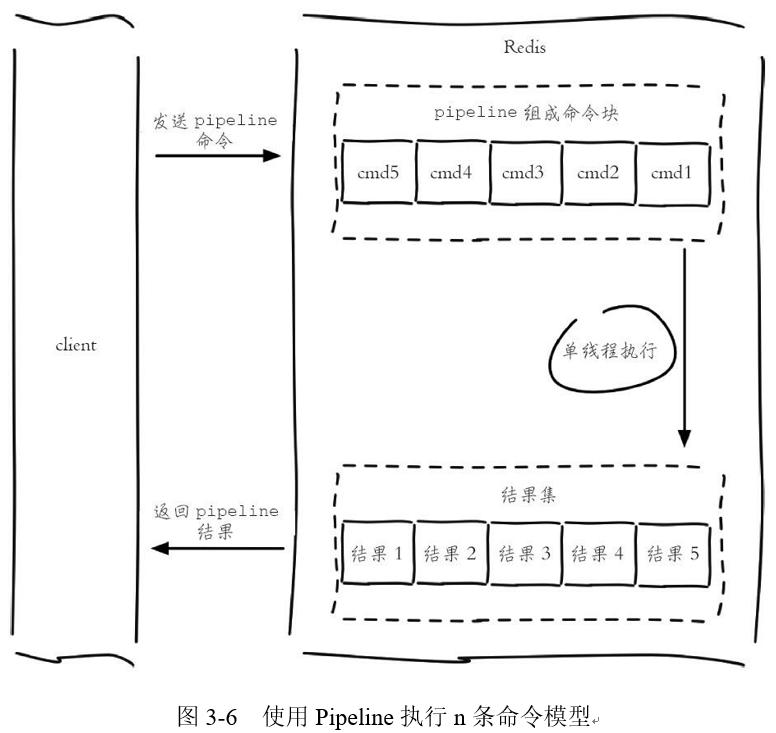
Pipeline（流水线）机制能改善上面这类问题，它能将一组Redis命令进

行组装，通过一次RTT传输给Redis，再将这组Redis命令的执行结果按顺序返回给客户端，图3-5为没有使用Pipeline执行了n条命令，整个过程需要n次 RTT。



Pipeline执行速度一般比逐条执行要快。

·客户端和服务端的网络延时越大，Pipeline的效果越明显。



#### 2.4.3复制

#### 2.4.4 哨兵机制

Redis的主从复制模式下，一旦主节点由于故障不能提供服务，需要人工将从节点晋升为主节点，同时还要通知应用方更新主节点地址，对于很多应用场景这种故障处理的方式是无法接受的。Redis从2.8开始正式提供了Redis Sentinel（哨兵）架构来解决这个问题。

Redis Sentinel具有以下几个功能：

·监控：Sentinel节点会定期检测Redis数据节点、其余Sentinel节点是否可达。

·通知：Sentinel节点会将故障转移的结果通知给应用方。

·主节点故障转移：实现从节点晋升为主节点并维护后续正确的主从关系。

·配置提供者：在Redis Sentinel结构中，客户端在初始化的时候连接的是Sentinel节点集合，从中获取主节点信息。

同时看到，Redis Sentinel包含了若个Sentinel节点，这样做也带来了两个好处：

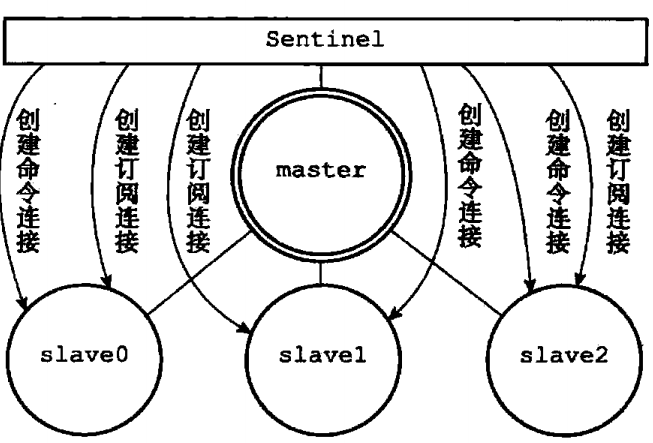
·对于节点的故障判断是由多个Sentinel节点共同完成，这样可以有效地防止误判。

·Sentinel节点集合是由若干个Sentinel节点组成的，这样即使个别Sentinel 节点不可用，整个Sentinel节点集合依然是健壮的。

但是Sentinel节点本身就是独立的Redis节点，只不过它们有一些特殊，

它们不存储数据，只支持部分命令。下一节将完整介绍Redis Sentinel的部署过程，相信在安装和部署完Redis Sentinel后，读者能更清晰地了解Redis

Sentinel的整体架构。



**哨兵监控**

26381

26379

26380

7003

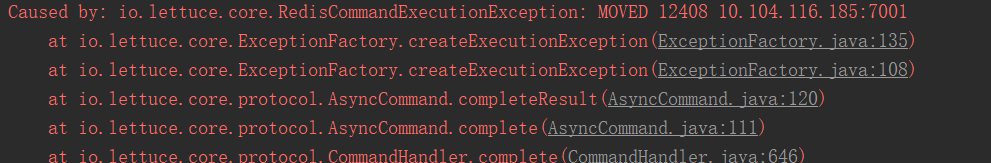
7001

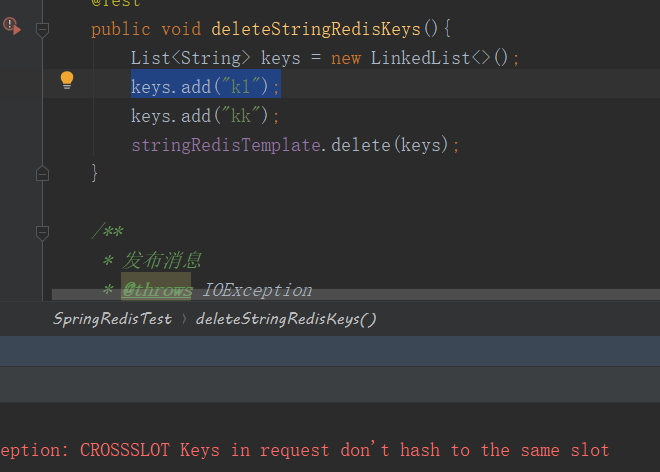
7000

70004

70002

70005





# 三：Spring Data Cache

### 3.1 Spring缓存抽象

1. Spring从3.1开始定义了org.springframework.cache.Cache和org.springframework.cache.CacheManager接口来统一不同的缓存技术；并支持使用JCache（JSR-107）注解简化我们开发；Spring Boot会尝试检测以下提供程序（按指示的顺序）：（[Generic](https://docs.spring.io/spring-boot/docs/2.1.6.RELEASE/reference/html/boot-features-caching.html" \l "boot-features-caching-provider-generic" \o "33.1.1 Generic)，[JCache (JSR-107)](https://docs.spring.io/spring-boot/docs/2.1.6.RELEASE/reference/html/boot-features-caching.html" \l "boot-features-caching-provider-jcache" \o "33.1.2 JCache (JSR-107)) ，[EhCache 2.x](https://docs.spring.io/spring-boot/docs/2.1.6.RELEASE/reference/html/boot-features-caching.html" \l "boot-features-caching-provider-ehcache2" \o "33.1.3 EhCache 2.x)，[Hazelcast](https://docs.spring.io/spring-boot/docs/2.1.6.RELEASE/reference/html/boot-features-caching.html" \l "boot-features-caching-provider-hazelcast" \o "33.1.4 Hazelcast)，[Infinispan](https://docs.spring.io/spring-boot/docs/2.1.6.RELEASE/reference/html/boot-features-caching.html" \l "boot-features-caching-provider-infinispan" \o "33.1.5 Infinispan)，[Couchbase](https://docs.spring.io/spring-boot/docs/2.1.6.RELEASE/reference/html/boot-features-caching.html" \l "boot-features-caching-provider-couchbase" \o "33.1.6 Couchbase)，[Redis](https://docs.spring.io/spring-boot/docs/2.1.6.RELEASE/reference/html/boot-features-caching.html" \l "boot-features-caching-provider-redis" \o "33.1.7 Redis)，[Caffeine](https://docs.spring.io/spring-boot/docs/2.1.6.RELEASE/reference/html/boot-features-caching.html" \l "boot-features-caching-provider-caffeine" \o "33.1.8 Caffeine)，[Simple](https://docs.spring.io/spring-boot/docs/2.1.6.RELEASE/reference/html/boot-features-caching.html" \l "boot-features-caching-provider-simple" \o "33.1.9 Simple)）

* Cache接口为缓存的组件规范定义，包含缓存的各种操作集合；
* Cache接口下Spring提供了各种xxxCache的实现；如RedisCache，EhCacheCache ,ConcurrentMapCache等；
* 每次调用需要缓存功能的方法时，Spring会检查检查指定参数的指定的目标方法是否已经被调用过；如果有就直接从缓存中获取方法调用后的结果，如果没有就调用方法并缓存结果后返回给用户。下次调用直接从缓存中获取。
* 使用Spring缓存抽象时我们需要关注以下两点；

1、确定方法需要被缓存以及他们的缓存策略

2、从缓存中读取之前缓存存储的数据

### 3.2几个重要概念&缓存注解

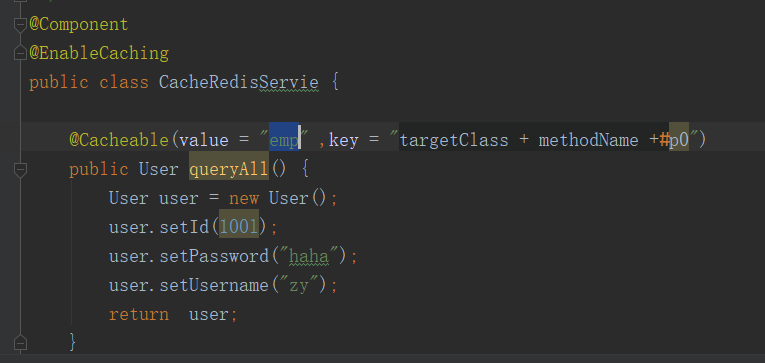


**@Cacheable/@CachePut/@CacheEvict 主要的参数**



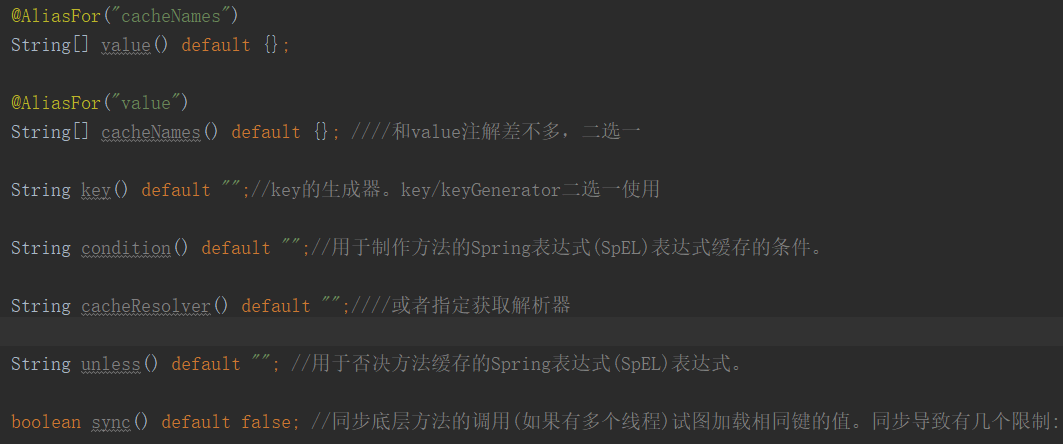
### 3.3 **.缓存@Cacheable**

@Cacheable注解会先查询是否已经有缓存，有会使用缓存，返回值存储在关联的缓存中。请注意自动处理Java8的{@code Optional}返回类型及其如果存在，内容将存储在缓存中。



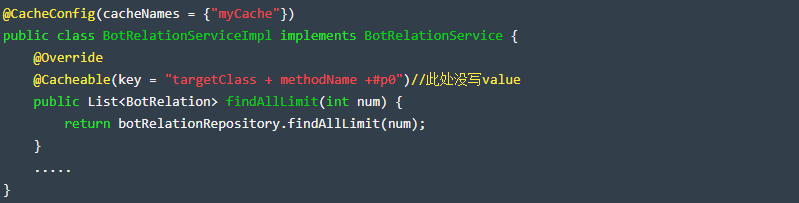
**深入源码，查看它的其它属性**

我们打开@Cacheable注解的源码，可以看到该注解提供的其他属性，如：



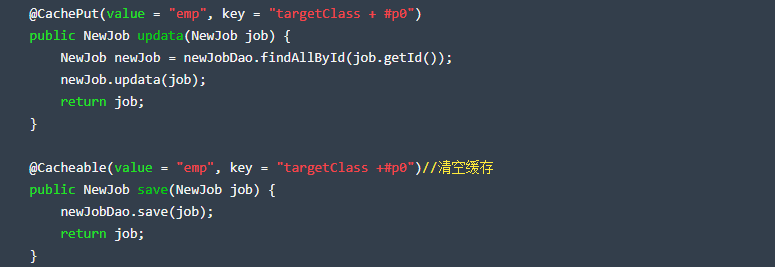
### **3.4配置@CacheConfig**

当我们需要缓存的地方越来越多，你可以使用@CacheConfig(cacheNames = {"myCache"})注解来统一指定value的值，这时可省略value，如果你在你的方法依旧写上了value，那么依然以方法的value值为准。



### 3.5**更新@CachePut**

@CachePut注解的作用 主要针对方法配置，能够根据方法的请求参数对其结果进行缓存，和 @Cacheable 不同的是，它每次都会触发真实方法的调用 。简单来说就是用户更新缓存数据。但需要注意的是该注解的value 和 key 必须与要更新的缓存相同，也就是与@Cacheable 相同。示例：



### 3.6 **清除@CacheEvict**

@CachEvict 的作用 主要针对方法配置，能够根据一定的条件对缓存进行清空 。

| **属性** | **解释** | **示例** |
| --- | --- | --- |
| allEntries | 是否清空所有缓存内容，缺省为 false，如果指定为 true，则方法调用后将立即清空所有缓存 | @CachEvict(value=”testcache”,allEntries=true) |
| beforeInvocation | 是否在方法执行前就清空，缺省为 false，如果指定为 true，则在方法还没有执行的时候就清空缓存，缺省情况下，如果方法执行抛出异常，则不会清空缓存 | @CachEvict(value=”testcache”，beforeInvocation=true) |

示例：

