# redis

## redis概述

### reduis的引入

在传统的Java Web项目中，使用数据库进行存储数据，但是有一些致命的弊端，这些弊端主要来自于性能方面。由于数据库持久化数据主要是面向磁盘，而磁盘的读/写比较慢，在一般管理系统上，由于不存在高并发，因此往往没有瞬间需要读/写大量数据的要求，这个时候使用数据库进行读/写是没有太大的问题的，但是在互联网中，往往存在大数据量的需求，比如一些商品抢购的场景，或者是主页讲问量瞬间较大的时候，一瞬间成千上万的

请求就会到来，需要系统在极短的时间内完成成千上万次的读/写操作，这个时候往往不是数据库能够承受的，极其容易造成数据库系统瘫痪，最终导致服务宕机的严重生产问题。为了克服这些问题，Java Web项目往往就引入了NoSQL技术，NoSQL工具也是一种简易的数据库，它主要是一种基于内存的数据库，并提供一定的持久化功能。Redis和MongoDB是当前使用最广泛的NoSQL，而本书主要介绍的是Redis技术，它的性能十分优越，可以支持每秒十几万次的读/写操作，其性能远超数据库，并且支持集群、分布式、

主从同步等配置，原则上可以无限扩展，让更多的数据存储在内存中，而更让我们感到欣喜的是它还能支持一定的事务能力，这在高并发访问的场景下保证数据安全和一致性特别有用。

Redis的性能优越主要来自于3个方面。首先，它是基于ANSIC语言编写的，接近于汇编语言的机器语言，运行十分快速。其次，它是基于内存的读/写，速度自然比数据库的磁盘读/写要快得多。最后，它的数据库结构只有6种数据类型，数据结构比较简单，因此规则较少，而数据库则是范式，完整性、规范性需要考虑的规则比较多，处理业务会比较复杂。所以一般而言Redis的速度是正常数据库的几倍到几十倍，如果把命中率高的数据存储在Redis上，通过Redis读/写和操作这些数据，系统的性能就会远超只使用数据库的情况，所以用好Redis对于Java互联网项目的响应速度和性能是至关重要的。

### redis介绍

Redis:REmote DIctionary Server(远程字典服务器）

属是完全开源免费的，用C语言编写的，遵守BSD协议，是一个高性能的（key/value)分布式内存数据库，基于内存运行并支持持久化的NoSQL数据库，是当前最热门的NoSql数据库之，也被人们称为数据结构服务器。

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。

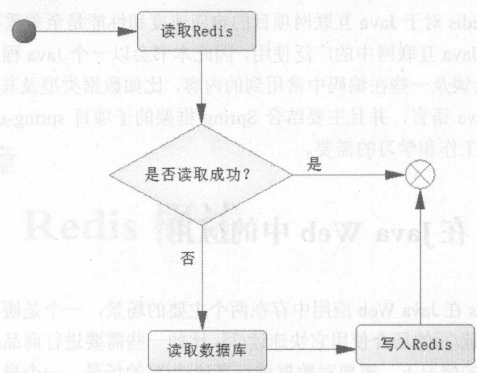
### redis主要作用

1、缓存

在对数据库的读/写操作中，现实的情况是读操作的次数远超写操作，一般是1：9到3:7的比例，所以需要读的可能性是比写的可能性多得多。当发送SQL去数据库进行读取时，数据库就会去磁盘把对应的数据索引回来，而索引磁盘是一个相对缓慢的过程。如果把数据直接放在运行在内存中的Redis服务器上，那么不需要去读/写磁盘了，而是直接读取内存，显然速度会快得多，并且会极大减轻数据库的压力。

而使用内存进行存储数据开销也是比较大的，因为磁盘可以是TGB级别，而且十分廉价，内存一般是几百个GB就相当了不起了，所以内存虽然高效但空间有限，价格也比磁盘高许多，因此使用内存代价较高，并不是想存什么就存什么，因此我们应该考虑有条件的存储数据。一般而言，存储一些常用的数据，比如用户登录的信息；一些主要的业务信息，比如银行会存储一些客户基础信息、银行卡信息、最近交易信息等。一般而言在使Redis 存储的时候，需要从3个方面进行考虑。

业务数据常用吗？命中率如何？如果命中率很低，就没有必要写入缓存。该业务数据是读操作多，还是写操作多，如果写操作多，频繁需要写入数据库，也没有必要使用缓存。业务数据大小如何？如果要存储几百兆字节的文件，会给缓存带来很大的压力，有没有必要？



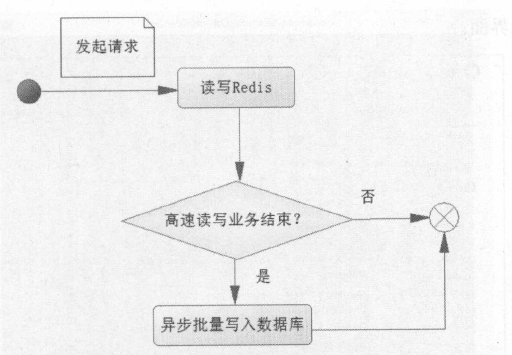
当第一次读取数据的时候，读取Redis的数据就会失败，此时会触发程序读取数据库，把数据读取出来，并且写入Redis。

当第二次及以后读取数据时，就直接读取Redis，读到数据后就结束了流程，这样速度就大大提高了。

2）高速读写场合

在互联网的应用中，往往存在一些需要高速读/写的场合，比如商品的秒杀，抢红包，淘宝、京东的双十一活动或者春运抢票等。这类场合在一个瞬间成千上万的请求就会达到服务器，如果使用的是数据库，一个瞬间数据库就需要执行成千上万的SQL，很容易造成数据库的瓶颈，严重的会导致数据库瘫痪，造成Java Web系统服务崩溃。

在这样的场合的应对办法往往是考虑异步写入数据库，而在高速读/写的场合中单单使用Redis去应对，把这些需要高速读/写的数据，缓存到Redis中，而在满足一定的条件下，触发这些缓存的数据写入数据库中。先看看一次请求操作的流程图，如图17-3所示。



进一步论述这个过程：当一个请求达到服务器，只是把业务数据先在Redis读/写，而没有进行任何对数据库的操作，换句话说系统仅仅是操作Redis缓存，而没有操作数据库，这个速度就比操作数据库要快得多，从而达到需要高速响应的效果。但是一般缓存不能持久化，或者所持久化的数据不太规范，因此需要把这些数据存入数据库，所以在一个请求操作完Redis的读/写后，会去判断该高速读/写的业务是否结束。这个判断的条件住往就是秒杀商品剩余个数为0，抢红包金额为0，如果不成立，则不会操作数据库；如果成立，则触发事件将Redis缓存的数据以批量的形式一次性写入数据库，从而完成持久化的工作。

假设面对的是一个商品秒杀的场景，从上面的流程看，一个用户抢购商品，绝大部分的场合都是在操作内存数据库Redis，而不是磁盘数据库，所以其性能更为优越。只有在商品被抢购一空后才会触发系统把Redis缓存的数据写入数据库磁盘中，这样系统大部分的操作基于内存，就能够在秒杀的场合高速响应用户的请求，达到快速应答。而现实中这种需要高速响应的系统会比上面的分析更复杂，因为这里没有讨论高并发下的数据安全和一致性问题，没有讨论有效请求和无效请求、事务一致性等诸多问题。

### redis与数据库的异同

和数据库一样类，Redis等NoSQL工具也能够存储数据，有人认为NoSQL将来会取代数据库，但是笔者却不那么认为，这里谈谈NoSQL和传统数据库的差异。

首先，NoSQL的数据主要存储在内存中（部分可以持久化到磁盘），而数据库主要是磁盘。其次，NoSQL数据结构比较简单，虽然能处理很多的问题，但是其功能毕竟是有限的，不如数据库的SOL语句强大，支持更为复杂的计算。再次，NoSQL并不完全安全稳定，由于它基于内存，一旦停电或者机器故障数据就很容易丢失数据，其持久化能力也是有限的，而基于磁盘的数据库则不会出现这样的问题。最后，其数据完整性、事务能力、安全性、可靠性及可扩展性都远不及数据库。

基于以上原因，笔者并不认为为NoSQL会取代数据库。毫无疑问，Redis作为一NoSQL是十分成功的，但是它的成功主要是解决互联网系统的一些问题，而主要的问题是性能问题。实际上，在互联网系统大部分的业务场景中，业务都是相对简单的，而难以处理的问题主要是性能问题，特别是那些会员数比较多的高并发服务网站。例如，你可以常常在淘宝或者京东网站上看到一个即将被抢购的商品，有多达几万人的关注，可能一个时刻就发生了成千上万笔业务，此时使用Redis作为缓存数据，就可以明显提升系统的性能，而且这十分有效。所以基于两者之间的区别，笔者认为使用NoSQL去取代数据库，目前还做不到，但是作为一种提高互联网应用性能的辅助工具，它十分有用。

### Redis的高并发

官方的bench-mark数据:测试完成了50个并发执行100000请求。设置和获取的值是一个256字节字符串。结果:读的速度是110000次/s,写的速度是81000次/s. redis尽量少写多读,符合缓存的适用要求。单机redis支撑万级,如果10万+可以采用主从复制的模式。

1, Redis是纯内存数据库,所以读取速度快。

2, Redis使用的是非阻塞10, 10多路复用,减少了线程切换时上下文的切换和竞争。

3. Redis采用了单线程的模型,保证了每个操作的原子性,也减少了线程的上下文切换和竞争。

4, Redis存储结构多样化,不同的数据结构对数据存储进行了优化加快读取的速度。

5. Redis采用自己实现的事件分离器,效率比较高,内部采用非阻塞的执行方式,吞吐能力比较大。

## redis下载安装

### 下载

<https://redis.io/download>

### 解压安装

创建目录：mkdir /usr/local/redis

解压 ：tar -zxvf redis.tar

### 编译

先安装 gcc-c++:yum -y install gcc-c++

的解压目录里面:cd regdis

编译:make

### 安装

安装目录：make install PREFIX=/usr/local/redis

### 安装后的目录

只有一个bin目录



redis-benchmark：性能测试工具

redis-check-aof：修复有问题的AOF文件

redis-check-dump：修复有问题的dump.rdb文件

redis-cli：客户端，操作入口

redis-sentinel：redis集群使用

redis-server：Redis服务器启动命令

### 启动redis

./redis-server

新开一个桌面连接redis

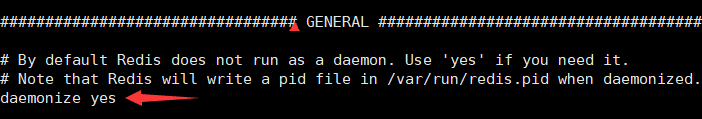
./redis-cli

### 自定义redis的配置文件

找到redis解压目录种的配置文件:redis.conf

赋值到bin目录下：cp redis.conf ../bin/

修改redis.cnf的general(一般配置)的daemonize 为yes，默认为非后台配置，修改为yes后，后台启动。



### 查看redis服务是否启动

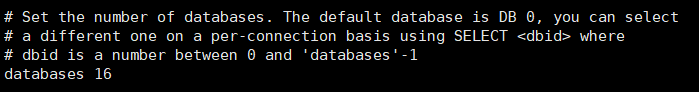
ps -ef|grep redis

验真是否启动：redis-cli -p 6379 登录

发送:ping 他会返回pong



### 查看redis的数据库



redis的默认数据库为16个，0-15，可以通过select n 来切换数据库。

例如：切换到6库

select 7

### redis开机自启

## 常用命令

1、切换数据库

select n

注：默认为16个，从0开始

2、查看当前数据库的key的数量

Dbsize

3、查看所有的键

keys \*

4、清空库或清空所有库

flushdb：删除当前库

flushall：删除所有库

### redis的配置文件

|  |
| --- |
|  |

## Redis数据类型分析

### key

1)keys \* 查看所有的键

2)exists key 名字，判断某个key是否存在,

存在：返回1

不存在：返回0

3)move key db :当前库就没有了，一出了

4)expire key 秒钟：为给定的key设置过期时间

5)ttl key 查看还有多少秒过期，-1表示永不过期-2表示已经过期

已经过期后，将会移除系统

6)type key 查看你的key是什么类型。

7)如果设置相同的key，后一个插入的key会覆盖前面的key。

### 字符串(string)

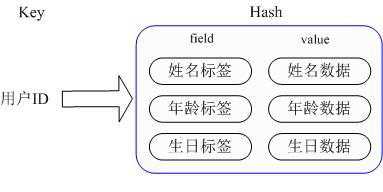
string是redis最基本的类型，你可以理解成与Memcached一模一样的类型，一个key对应一个value。string类型是二进制安全的。意思是redis的string可以包含任何数据。比jpg图片或者序列化的对象。string类型是Redis最基本的数据类型，一个redis中字符串value最多可以是512M

常用命令：

|  |  |
| --- | --- |
| get key | 获取给定键的值 |
| set key | 设置给定键的值 |
| strlen key | 返回给定键的值的长度 |
| incrby key-name amount | 将键存储的值加上整数amont，如果没有mount，默认为加1 |
| decrby key-name amount | 将键存储的值减去整数amont，如果没有mount，默认为减1 |
| getrange | getrange key-name start end，获取一个偏移量start至偏移量end范围内所有字符组成的子串，包括start和end在内 |
| setrange | setrange key-name offset value，将从start偏移量开始的子串设置为给定值 |
| setex key second value | 设置过期时间  setex k4 10 v4 设置键k4生命名为10秒  结果:10秒钟之后返回nil 相当于0 |
| SETNX key value  （SET if Not eXists） | 将 key 的值设为 value ，当且仅当 key 不存在。  若给定的 key 已经存在，则 SETNX 不做任何动作。 |
| mset key1 value1  key2 value2 …… | 同时设置多个值 |
| mget key1 key2 | 同时获取多个值 |
|  |  |

### hash(哈希)

Hash是一个String类型的field和value之间的映射表，即redis的Hash数据类型的key（hash表名称）对应的value实际的内部存储结构为一个HashMap，因此Hash特别适合存储对象。相对于把一个对象的每个属性存储为String类型，将整个对象存储在Hash类型中会占用更少内存。

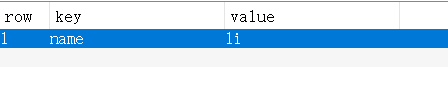


常用命令：

（1）hset——设置key对应的HashMap中的field的value

格式：hset key field value

例如：hset hash name li



（2）hget——获取key对应的HashMap中的field的value

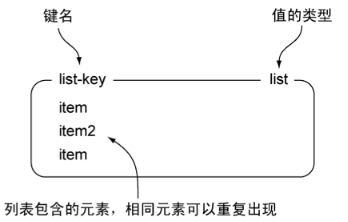
格式：hget key field

（3）hgetall——获取key对应的HashMap中的所有field的value

格式：hgetall key



### List结构



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导列表的头部（左边)或者尾部（右边）。它的底层实际是个链表。

Redis的List类型其实就是每一个元素都是String类型的双向链表。我们可以从链表的头部和尾部添加或者删除元素。这样的List既可以作为栈，也可以作为队列使用。

特点:

1、如果间不存在，则创建新的链表

2、如果键已经存在，新增内容

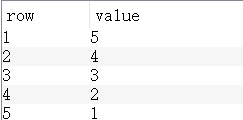
3、如果值全部移除，对应的键也消失了

4、内部使用双向链表实现，所以获取越接近两端的元素速度越快，但通过索引访问时会比较慢

常用命令

添加左边元素：LPUSH 语法：LPUSH key value [value ...] ，返回添加后的列表元素的总个数

LPUSH list 1 2 3 4 5



添加右边元素：RPUSH 语法：RPUSH key value [value ...] ，返回添加后的列表元素的总个数



移除左边第一个元素：LPOP 语法：LPOP key ，返回被移除的元素值

移除右边第一个元素：RPOP 语法：RPOP key ，返回被移除的元素值

获取list中的值 ：lrange 语法：lrange key start end

例如：lrange key 0 -1 全取key的值

列表元素个数：LLEN 语法：LLEN key， 不存在时返回0，redis是直接读取现成的值，并不是统计个数

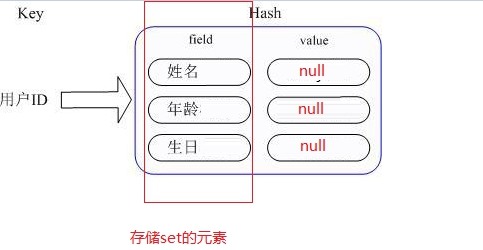
按照索引下标回去元素(从上到下)： 语法：lindex key

删除N个value 语法：lrem key 删除所有key 的值

### set

Redis 集合（Set类型）是一个无序的String类型数据的集合，类似List的一个列表，与List不同的是Set不能有重复的数据。实际上，Set的内部是用HashMap实现的，Set只用了HashMap的key列来存储对象。

可见创建一个HashSet的时候实际上创建了一个HashMap；Set中的元素，只是存放在了底层HashMap的key上，底层HashMap的value列为空，遍历HashSet的时候从HashMap中取出keySet来遍历。



常用命令

（1）sadd——在key对应的set中添加一个元素。

（2）smembers——获取key对应的set的所有元素。

（3）spop——随机返回并删除key对应的set中的一个元素



### Sortset

SortSet顾名思义，是一个排好序的Set，它在Set的基础上增加了一个顺序属性score，这个属性在添加修改元素时可以指定，每次指定后，SortSet会自动重新按新的值排序。

sorted set的内部使用HashMap和跳跃表(SkipList)来保证数据的存储和有序，HashMap里放的是成员到score的映射，而跳跃表里存放的是所有的成员，排序依据是HashMap里存的score。

常用命令

（1）zadd ——在key对应的zset中添加一个元素

格式：zadd key sort value sort value sort value

例如zadd zset 1 li 2 sx 3 lw

（2）zrange——获取key对应的zset中指定范围的元素，-1表示获取所有元素

获取全部元素：zrange myzset 0 -1

3、zrem——删除key对应的zset中的一个元素

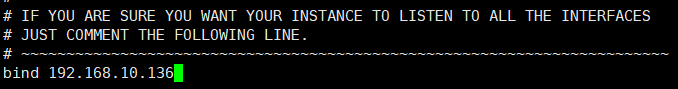
例如：zrem myzset li 删除li这个元素



## Java 访问Redis

### Redis 外网访问

1)修改配置文件redis.conf

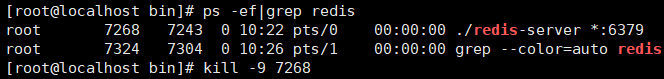


修改127.0.0.7为linux的访问ip

2)修改成功之后重启服务器，在redis 里面没有重启的命令，我们可以通过杀死进程，然后，在启动。

查看当前的进程:ps -ef|grep redis

强制杀死进程：kill -9 进程号



3)再次启动redis-server(加载配置文件)

命令 ./redis-server redis.conf

4）也可以使用redis-cli直接连接

./redis-cli -h 192.168.10.136 -p 6379

## java 连接redis

### 使用工具:jedis maven依赖

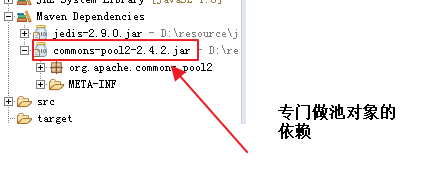
|  |
| --- |
| <properties>  <redis.version>2.9.0</redis.version>  </properties>  <dependencies>  <dependency>  <groupId>redis.clients</groupId>  <artifactId>jedis</artifactId>  <version>${redis.version}</version>  </dependency>  </dependencies> |

测试代码

|  |
| --- |
| **public** **class** App {  **public** **static** **void** main(String[] args) {  // 新建一个jedis的客户端  String host = "192.168.10.136";  Jedis jedis = **null** ;  **try** {  jedis = **new** Jedis(host);  // jedis.set("java", "java");  String value = jedis.get("java");  // System.out.println("set ok");  System.***out***.println("value:"+value);  }**catch** (Exception e) {  e.printStackTrace();  }**finally** {  // 关闭资源的操作  **if**(**null**!=jedis) {  **try** {  jedis.close();  }**catch** (Exception e) {  e.printStackTrace();  }**finally** {  jedis = **null** ;  }  }  }  }  } |

## Redis的连接池管理redis

连接是一个对象，对象池（commons-pool 2apach的工具）





Method(Jedis): Close()

|  |
| --- |
| **public** **class** app {  **private** **static** JedisPoolConfig *poolConfig* = **new** JedisPoolConfig();  **private** **static** JedisPool *pool* = **null**;  **private** **static** **final** String ***host*** = "192.168.10.136";  **static** {  *poolConfig*.setMaxTotal(20);//连接池最大数量  *poolConfig*.setMaxIdle(15); //连接池最大空闲数量  *poolConfig*.setMinIdle(10); //连接池最小空闲数量  *pool* = **new** JedisPool(*poolConfig*,***host***);  }  **public** **static** **void** main(String[] args) {  Jedis jedis = *pool*.getResource();  String value = jedis.get("sx");  System.***out***.println(value);  //连接池归还或者是关闭  jedis.close();  }  } |

## RidesTemplate

由于Redis只能提供基于字符串型的操作，而在Java中使用的却以类对象为主，所以

需要Redis存储的字符串和Java对象相互转换。如果自己编写这些规则，工作量还是比较大的，比如一个角色对象，我们没有办法直接把对象存入Redis中，需要进一步进行转换，所以对操作对象而言，使用Redis还是比较难的。好在Spring对这些进行了封装和支持，它提供了序列化的设计框架和一些序列化的类，使用后它可以通过序列化把Java对象转换，

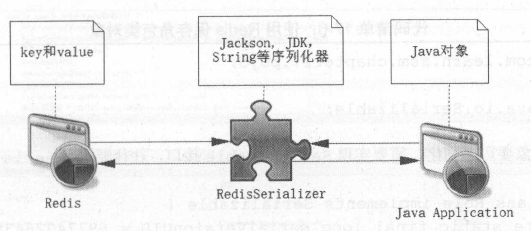
使得Redis能把它存储起来，并且在读取的时候，再把由序列化过的字符串转化为Java对象，这样在Java环境中使用Redis就更加简单了，所以更多的时候可以使用Spring提供Redis Template的机制来使用Redis。

普通的连接使用没有办法把Java对象直接存入Redis，而需要我们自己提供方案，这

时往往就是将对象序列化，然后使用Redis进行存储，而取回序列化的内容后，在通过转

换转变为Java对象，Spring模板中提供了封装的方案，在它内部提供了RedisSerializer接

口（org.springframework.data.redis.serializer.RedisSerializer)和一些实现类。



GenericJackson2JsonRedisSerializer,通用的使用Json2.jar的包，将Redis对象的序列化器。

Jackson2JsonRedisSerializer<T>,通过Jackson2.jar包提供的序列化进行转换。

JdkSerializationRedisSerializer<T>,使用JDK的序列化器进行转化。

OxmSerializer，使用Spring O/X对象Object和XML相互转换

StringRedisSerializer,使用字符串进行序列化。

Generic ToStringSerializer,通过通用的字符串序列化进行相互转换。

使用它们就能够帮助我们把对象通过序列化存储到Redis中，也可以把Redis存储的内容转换为Java对象，为此Spring提供的Redis Template还有两个属性。

## Jedis和Spring 整合

### 添加依赖

|  |
| --- |
| <properties>  <spring-context.version>4.3.20.RELEASE</spring-context.version>  <jedis.version>2.9.0</jedis.version>  </properties>  <dependencies>  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-context</artifactId>  <version>${spring-context.version}</version>  </dependency>  <dependency>  <groupId>redis.clients</groupId>  <artifactId>jedis</artifactId>  <version>${jedis.version}</version>  </dependency>  </dependencies> |

### spring-jedis.xml配置文件

//配置连接池

|  |
| --- |
| <bean id=*"poolConfig"* class=*"redis.clients.jedis.JedisPoolConfig"*>  <property name=*"maxTotal"* value=*"20"*></property>  <property name=*"maxIdle"* value=*"15"*></property>  <property name=*"minIdle"* value=*"10"*></property>  </bean>  <bean id=*"pool"* class=*"redis.clients.jedis.JedisPool"*>  <constructor-arg name=*"host"* value=*"192.168.10.136"*></constructor-arg>  <constructor-arg name=*"poolConfig"* ref=*"poolConfig"*></constructor-arg>  </bean> |

app测试

|  |
| --- |
| **public** **static** **void** main(String[] args) {  ApplicationContext context = **new** ClassPathXmlApplicationContext("classpath:spring-jedis.xml");  JedisPool jedisPool =(JedisPool) context.getBean("pool");  Jedis jedis = jedisPool.getResource();  String value = jedis.get("sx");  System.***out***.println(value);//li  jedis.close();  } |

### 软编码(提取配置文件)

jedis.properties

|  |
| --- |
| jedis.host=192.168.10.136  jedis.max.total=20  jedis.max.idle=15  jedis.min.idle=20 |

xml中加载配置文件(namespace中添加约束)

|  |
| --- |
| <context:property-placeholder location=*"classpath:/properties/\*.properties"* />  <bean id=*"poolConfig"* class=*"redis.clients.jedis.JedisPoolConfig"*>  <property name=*"maxTotal"* value=*"${jedis.max.total}"*></property>  <property name=*"maxIdle"* value=*"${jedis.max.idle}"*></property>  <property name=*"minIdle"* value=*"${jedis.min.idle}"*></property>  </bean>  <bean id=*"pool"* class=*"redis.clients.jedis.JedisPool"*>  <constructor-arg name=*"host"* value=*"${jedis.host}"*></constructor-arg>  <constructor-arg name=*"poolConfig"* ref=*"poolConfig"*></constructor-arg>  </bean> |

## Redis和Spring data整合

## Redis和Spring boot 整合(基础版本)

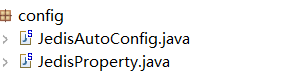
### 创建项目并添加依赖

版本：1.5.18

依赖web，jedis

|  |
| --- |
| <dependency>  <groupId>redis.clients</groupId>  <artifactId>jedis</artifactId>  </dependency> |

### 配置类文件



JedisProperty

|  |
| --- |
| @Configuration  @ConfigurationProperties(prefix="jedis")  **public** **class** JedisProperty {  **private** String host;  **private** Integer maxTotal;  **private** Integer maxIdle;  **private** Integer minIdle;  } |

JedisAutoConfig

|  |
| --- |
| @Configuration  **public** **class** JedisAutoConfig {  @Autowired  **private** JedisProperty jedisProperty;  @Bean("jedisPool")  **public** JedisPool pool(){  JedisPoolConfig poolConfig = **new** JedisPoolConfig();  poolConfig.setMaxIdle(jedisProperty.getMinIdle());  poolConfig.setMaxTotal(jedisProperty.getMaxTotal());  poolConfig.setMinIdle(jedisProperty.getMinIdle());  **return** **new** JedisPool(poolConfig,jedisProperty.getHost());  }  } |

yum

|  |
| --- |
| jedis:  host: 192.168.10.136  max-total: 20  max-idle: 15  min-idle: 10 |

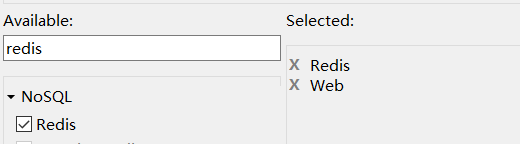
### 测试

|  |
| --- |
| @SpringBootApplication  @RestController  **public** **class** RedisSpringbootApplication {  @Autowired  **private** JedisPool jedisPool;    @RequestMapping("index")  **public** String getInfo() {  Jedis jedis = jedisPool.getResource();  **return** jedis.get("sx");  }  } |

## Spring boot 整合redis(进阶版)

### 创建项目

加载redis的依赖



### 配置文件

yum

|  |
| --- |
| spring:  redis:  host: 192.168.10.136  pool:  max-active: 25  max-idle: 15  min-idle: 10 |

## redis操作

|  |
| --- |
| **public** **class** TestRedis {  ApplicationContext context = **new** ClassPathXmlApplicationContext("classpath:spring-jedis.xml");  JedisPool jedisPool =(JedisPool) context.getBean("pool");  Jedis jedis = jedisPool.getResource();  //1、普通的字符串操作  @Test  **public** **void** test1() {  jedis.set("sx", "li");  String value = jedis.get("sx");  System.***out***.println(value);  }    //2、字节操作  @Test  **public** **void** test2() {  //jedis.set("byte".getBytes(), "byte1".getBytes());  String value = jedis.get("byte");  System.***out***.println(value);//byte1  //对象转为字节  User user = **new** User("sx", "18", "女");  jedis.set("user1".getBytes(), SerializableUtils.*serializable*(user));  }      //3、json的序列化对象  @Test  **public** **void** test3() {  User userjson = **new** User("lee", "22", "男");  jedis.set("userjson", JSON.*toJSONString*(userjson));  }    //4、hash的操作  @Test  **public** **void** test4() {  //写入  //jedis.hset("man", "name", "sx");  //jedis.hset("man", "age", "12");  //获取单个值  String name = jedis.hget("man","name");  String age = jedis.hget("man","age");  System.***out***.println(name+"+"+age);//sx+12  //获取键的全部值  Map<String, String> man = jedis.hgetAll("man");  System.***out***.println(man);  }  //zset集合  @Test  **public** **void** test5() {  //写入  /\*jedis.zadd("pad", 1000,"ls");  jedis.zadd("pad", 2000,"sx");  jedis.zadd("pad", 3000,"df");  jedis.zadd("pad", 4000,"ad");  jedis.zadd("pad", 5100,"df");\*/  //从小到大获取值,获取前三个值  Set<String> pad = jedis.zrange("pad", 0, 2);  **for** (String str : pad) {  System.***out***.println(str);  }  //从大到小获取值,获取前三个值  Set<String> pad1 = jedis.zrevrange("pad", 0, 2);  **for** (String str : pad1) {  System.***out***.println(str);  }  }  } |

## 在spring 里面利用Redis 实现缓存

### 使用切面实现缓存

切面：环绕通知

添加aspect 依赖

|  |
| --- |
| <dependency>  <groupId>org.aspectj</groupId>  <artifactId>aspectjweaver</artifactId>  <version>1.9.2</version>  </dependency> |

切面类

|  |
| --- |
| @Component  @Aspect  **public** **class** CacheAspect {  @Around("execution (\* com.sxt.service.impl.UserServiceImpl.\*(..))")  **public** Object aroud(ProceedingJoinPoint point) **throws** Throwable {  System.***out***.println("方法执行前");  Object result = point.proceed(point.getArgs());  System.***out***.println("方法执行后");  **return** result;  }  } |

配置：开启aop

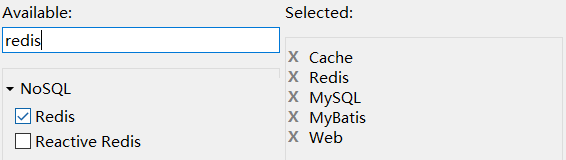
|  |
| --- |
| <!--开启aop-->  <aop:aspectj-autoproxy expose-proxy=*"true"*></aop:aspectj-autoproxy> |

## Spring boot data项目(自带缓存)

添加缓存的注解

|  |  |
| --- | --- |
| @CacheConfig | 提取相同配置的 |
| @Cacheable | 给方法加缓存 |
| @CacheEvient | （删除缓存的值） |
| @Cacheput | （更新缓存的值） |

### 新建项目并选择依赖



依赖：

|  |
| --- |
| <!-- 分页插件的依赖 -->  <dependency>  <groupId>com.github.pagehelper</groupId>  <artifactId>pagehelper-spring-boot-starter</artifactId>  <version>1.2.5</version>  </dependency> |

### 配置文件

|  |
| --- |
| spring:  datasource:  driver-class-name: com.mysql.cj.jdbc.Driver  url: jdbc:mysql://192.168.10.136:3303/hm  username: root  password: 123456  redis:  host: 192.168.10.136  port: 6379  jedis:  pool:  max-active: 25  max-idle: 15  min-idle: 10  mybatis:  mapper-locations:  - classpath:/mapper/\*.xml  #pagehepler配置  pagehelper:  helper-dialect: mysql |

service

|  |
| --- |
| **public** **interface** UserService {    List<User> findAll(**int** page,**int** size);  User findById(Integer id);  User saveUser(User user);  User updateUser(User user);    **void** deleteUser(User user);  **void** deleteUser(Integer id);  } |

serviceimpl

|  |
| --- |
|  |

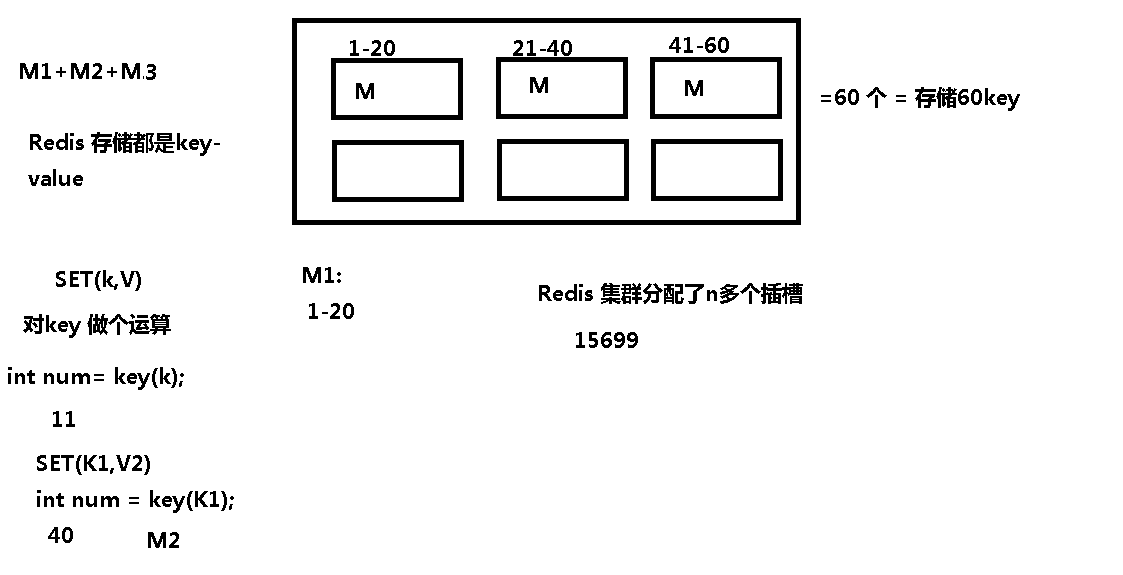
controller

|  |
| --- |
| @RestController  @RequestMapping("api/user")  **public** **class** UserController {  @Autowired  **private** UserService userService;  @PostMapping("/")  **public** String saveUser(User user) {  userService.saveUser(user);  **return** "ok";  }  @DeleteMapping("/{id}")  **public** String deletUser(@PathVariable("id")Integer id) {  userService.deleteUser(id);  **return** "ok";  }  @PutMapping("/")  **public** String updateUser(User user) {  userService.updateUser(user);  **return** "ok";  }  @GetMapping("/{id}")  **public** User findUser(@PathVariable("id")Integer id) {  **return** userService.findById(id);  }  @GetMapping("/")  **public** List<User> findUsers(@RequestParam(defaultValue="1")Integer page,@RequestParam(defaultValue="10")Integer size){  **return** userService.findAll(page, size);  }  } |

启动类

|  |
| --- |
| @SpringBootApplication  @MapperScan("com.sxt.mapper")  @EnableCaching //这里开启缓存  **public** **class** RedisSpringbootDataRedisApplication {  **public** **static** **void** main(String[] args) {  SpringApplication.*run*(RedisSpringbootDataRedisApplication.**class**, args);  }  } |

## Redis伪集群的搭建



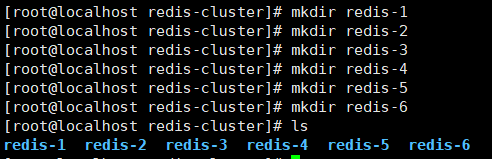
### 伪集群规划

|  |  |  |  |
| --- | --- | --- | --- |
| 文件夹 | Ip | 端口 | 角色 |
| Redis-1 | 192.168.10.136 | 7001 |  |
| Redis-2 | 192.168.10.136 | 7002 |  |
| Redis-3 | 192.168.10.136 | 7003 |  |
| Redis-4 | 192.168.10.136 | 7004 |  |
| Redis-5 | 192.168.10.136 | 7005 |  |
| Redis-6 | 192.168.10.136 | 7005 |  |

### 创建文件夹

1)在/usr/local下创建redis.cluster文件夹。

2)在redis.cluster文件家中创建redis-[1-6]



3)将redis的服务器文件redis-server复制到集群文件夹中。

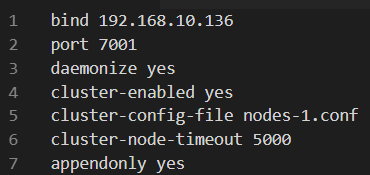
cp redis-server /usr/local/redis-cluster/

### 准备配置文件

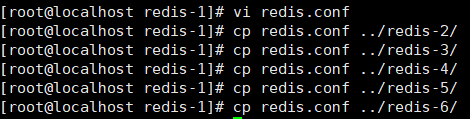
最小化的配置文件

|  |
| --- |
| bind ip  port 7000 端口  daemonize yes 后台运行  cluster-enabled yes 开启集群  cluster-config-file nodes.conf 不用你写，只有集群启动，该文件自动生成，但是该文件必须不同才行  cluster-node-timeout 5000 集群里面的机器互相ping，超时时间  appendonly yes # 开启redis的持久化的功能 |

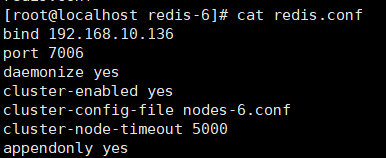
redis-1



复制到其他文件夹中



redis-2

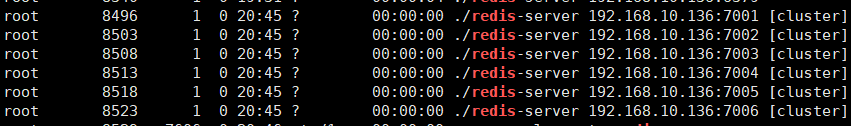


### 开启redis

使用各自的配置文件启动redis

|  |
| --- |
| ./redis-server ./redis-1/redis.conf.  ./redis-server ./redis-2/redis.conf.  ./redis-server ./redis-3/redis.conf.  ./redis-server ./redis-4/redis.conf.  ./redis-server ./redis-5/redis.conf.  ./redis-server ./redis-6/redis.conf. |

查看机器启动状态:ps -ef | grep redis

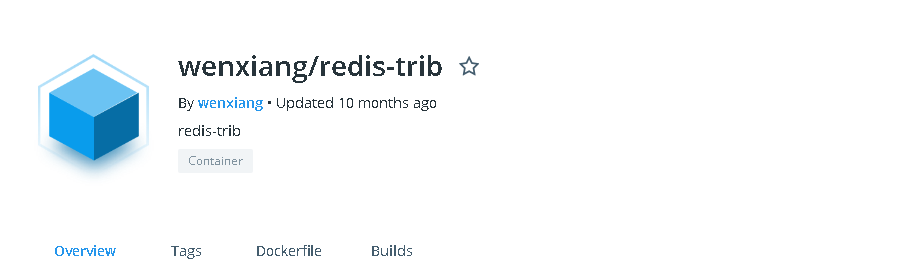


### 使用Ruby脚步搭建

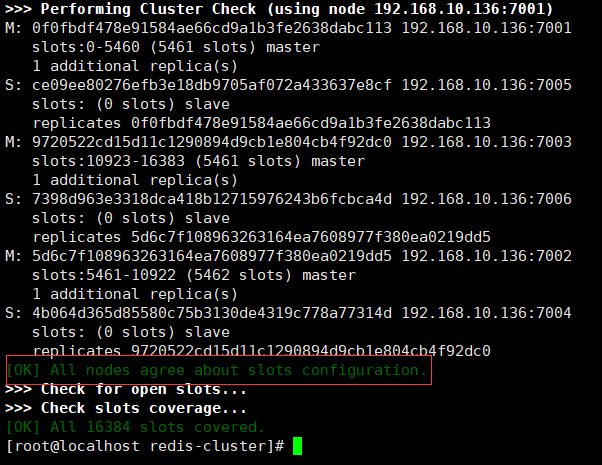
官方写了一个脚本，来配置到底哪个是主机，哪个是从机，需要我们运行一下。

曲线运行脚本，使用docker 运行该脚本

下载镜像：



|  |
| --- |
| docker run -t -i wenxiang/redis-trib create --replicas 1 192.168.10.136:7001 192.168.10.136:7002 192.168.10.136:7003 192.168.10.136:7004 192.168.10.136:7005 192.168.10.136:7006 |



### 连接redis集群



### 普通java项目连接集群

|  |
| --- |
| **public** **class** app {  **private** **static** JedisCluster *jedisCluster* = **null**;  **private** **static** JedisPoolConfig *poolConfig* = **new** JedisPoolConfig();  **private** **static** Set<HostAndPort> *nodes* = **new** HashSet<HostAndPort>();  **static** {  *poolConfig*.setMaxTotal(20);//连接池最大数量  *poolConfig*.setMaxIdle(15); //连接池最大空闲数量  *poolConfig*.setMinIdle(10); //连接池最小空闲数量  *nodes*.add(**new** HostAndPort("192.168.10.136", 7001));  *nodes*.add(**new** HostAndPort("192.168.10.136", 7002));  *nodes*.add(**new** HostAndPort("192.168.10.136", 7003));  *nodes*.add(**new** HostAndPort("192.168.10.136", 7004));  *nodes*.add(**new** HostAndPort("192.168.10.136", 7005));  *nodes*.add(**new** HostAndPort("192.168.10.136", 7006));  *jedisCluster* = **new** JedisCluster(*nodes*,*poolConfig*);  }  **public** **static** **void** main(String[] args) **throws** IOException {  *jedisCluster*.set("sx", "123456");  String value = *jedisCluster*.get("sx");  System.***out***.println(value);  //连接池归还或者是关闭  *jedisCluster*.close();  }  } |

### 在spring中连接redis集群

配置文件

|  |
| --- |
| <bean id=*"poolConfig"* class=*"redis.clients.jedis.JedisPoolConfig"*>  <property name=*"maxTotal"* value=*"${jedis.max.total}"*></property>  <property name=*"maxIdle"* value=*"${jedis.max.idle}"*></property>  <property name=*"minIdle"* value=*"${jedis.min.idle}"*></property>  </bean>  <!--连接单个redis -->  <!-- <bean id="pool" class="redis.clients.jedis.JedisPool">  <constructor-arg name="host" value="${jedis.host}"></constructor-arg>  <constructor-arg name="poolConfig" ref="poolConfig"></constructor-arg>  </bean> -->  <!--连接redis集群 -->  <bean id=*"jedisCluster"* class=*"redis.clients.jedis.JedisCluster"*>  <constructor-arg name=*"poolConfig"* ref=*"poolConfig"*></constructor-arg>  <constructor-arg name=*"nodes"* ref=*"nodes"*></constructor-arg>  </bean>  扫描配置文件  <context:property-placeholder location=*"classpath:/properties/\*.properties"* /> |

配置类，获取配置信息

|  |
| --- |
| @Configuration  **public** **class** JedisClusterNodes {  @Value("${redis.node1}") // ip:port  **private** String node1;  @Value("${redis.node2}")  **private** String node2;  @Value("${redis.node3}")  **private** String node3;  @Value("${redis.node4}")  **private** String node4;  @Value("${redis.node5}")  **private** String node5;  @Value("${redis.node6}")  **private** String node6;  @Bean("nodes") // 将对象放入容器里面  **public** HashSet<HostAndPort> nodes() {  HashSet<HostAndPort> nodes = **new** HashSet<HostAndPort>();  nodes.add(**new** HostAndPort(node1.split(":")[0], Integer.*valueOf*(node1.split(":")[1])));  nodes.add(**new** HostAndPort(node2.split(":")[0], Integer.*valueOf*(node2.split(":")[1])));  nodes.add(**new** HostAndPort(node3.split(":")[0], Integer.*valueOf*(node3.split(":")[1])));  nodes.add(**new** HostAndPort(node4.split(":")[0], Integer.*valueOf*(node4.split(":")[1])));  nodes.add(**new** HostAndPort(node5.split(":")[0], Integer.*valueOf*(node5.split(":")[1])));  nodes.add(**new** HostAndPort(node6.split(":")[0], Integer.*valueOf*(node6.split(":")[1])));  **return** nodes;  }  } |

redis.properties

|  |
| --- |
| redis.node1=192.168.10.136:7001  redis.node2=192.168.10.136:7002  redis.node3=192.168.10.136:7003  redis.node4=192.168.10.136:7004  redis.node5=192.168.10.136:7005  redis.node6=192.168.10.136:7006  jedis.max.total=25  jedis.max.idle=15  jedis.min.idle=10 |

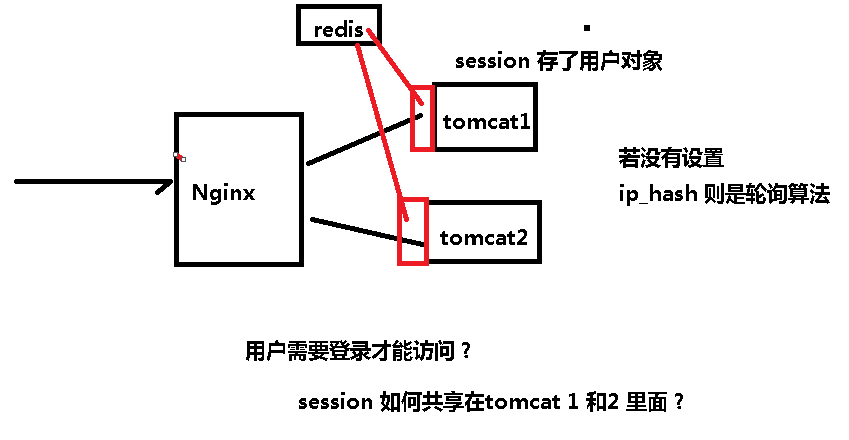
### 在springboot中连接集群

|  |
| --- |
| spring:  redis:  cluster: 修改yum配置文件  nodes:  - 192.168.10.136:7001  - 192.168.10.136:7002  - 192.168.10.136:7003  - 192.168.10.136:7004  - 192.168.10.136:7005  - 192.168.10.136:7005  jedis:  pool:  max-active: 25  max-idle: 15  min-idle: 10 |

## 项目里面redis的使用

### session的存储

用户登录的session共享



### redis做缓存使用

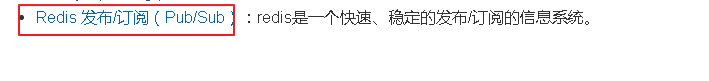
菜单数据

权限数据

### 购物车redis的实现

### 发布订阅功能（redis的发布订阅太简单了，不能用）

Mq解决发布订阅。



## redis的问题

### 删除key的策略

Redis 内存满了删除key的策略

Allkey-lru： 所有的key 都可以参与删除（推荐）

VoliteKey 只有设置过期的key 参与删除

Redis 过期删除如何实现？

立即删除：cpu

过期就删除：

Set(key,value)

给这个key设置过期时间。类似给key有个定时器

惰性删除：耗费内存

给key设置过期时间，key过期了，它并不会删除

当再次访问该key时，才触发删除

定期删除：折中的方案(该方案是默认的)

Redis 一段时间检查所有设置了过期时间的key，若过期，删除

## redis的持久化

### 持久化

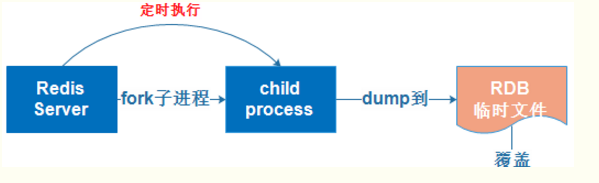
Redis是一种高级key-value数据库。它跟memcached类似，不过数据可以持久化，而且支持的数据类型很丰富。有字符串，链表，集 合和有序集合。支持在服务器端计算集合的并，交和补集(difference)等，还支持多种排序功能。所以Redis也可以被看成是一个数据结构服务 器。

Redis的所有数据都是保存在内存中，然后不定期的通过异步方式保存到磁盘上(这称为“半持久化模式”)；也可以把每一次数据变化都写入到一个append only file(aof)里面(这称为“全持久化模式”)。

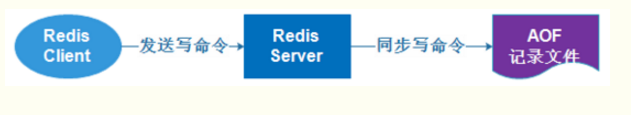
由于Redis的数据都存放在内存中，如果没有配置持久化，redis重启后数据就全丢失了，于是需要开启redis的持久化功能，将数据保存到磁 盘上，当redis重启后，可以从磁盘中恢复数据。redis提供两种方式进行持久化，一种是RDB持久化（原理是将Reids在内存中的数据库记录定时 dump到磁盘上的RDB持久化），另外一种是AOF（append only file）持久化（原理是将Reids的操作日志以追加的方式写入文件）。那么这两种持久化方式有什么区别呢，改如何选择呢？网上看了大多数都是介绍这两 种方式怎么配置，怎么使用，就是没有介绍二者的区别，在什么应用场景下使用。

### 两种持久化的不同

RDB持久化是指在指定的时间间隔内将内存中的数据集快照写入磁盘，实际操作过程是fork一个子进程，先将数据集写入临时文件，写入成功后，再替换之前的文件，用二进制压缩存储。



AOF持久化以日志的形式记录服务器所处理的每一个写、删除操作，查询操作不会记录，以文本的方式记录，可以打开文件看到详细的操作记录。



### 二者优缺点

RDB存在哪些优势呢？

1). 一旦采用该方式，那么你的整个Redis数据库将只包含一个文件，这对于文件备份而言是非常完美的。比如，你可能打算每个小时归档一次最近24小时的数 据，同时还要每天归档一次最近30天的数据。通过这样的备份策略，一旦系统出现灾难性故障，我们可以非常容易的进行恢复。

2). 对于灾难恢复而言，RDB是非常不错的选择。因为我们可以非常轻松的将一个单独的文件压缩后再转移到其它存储介质上。

3). 性能最大化。对于Redis的服务进程而言，在开始持久化时，它唯一需要做的只是fork出子进程，之后再由子进程完成这些持久化的工作，这样就可以极大的避免服务进程执行IO操作了。

4). 相比于AOF机制，如果数据集很大，RDB的启动效率会更高。

RDB又存在哪些劣势呢？

1). 如果你想保证数据的高可用性，即最大限度的避免数据丢失，那么RDB将不是一个很好的选择。因为系统一旦在定时持久化之前出现宕机现象，此前没有来得及写入磁盘的数据都将丢失。

2). 由于RDB是通过fork子进程来协助完成数据持久化工作的，因此，如果当数据集较大时，可能会导致整个服务器停止服务几百毫秒，甚至是1秒钟。

AOF的优势有哪些呢？

1). 该机制可以带来更高的数据安全性，即数据持久性。Redis中提供了3中同步策略，即每秒同步、每修改同步和不同步。事实上，每秒同步也是异步完成的，其效率也是非常高的，所差的是一旦系统出现宕机现象，那么这一秒钟之内修改的数据将会丢失。而每修改同步，我们可以将其视为同步持久化，即每次发生的数据变化都会被立即记录到磁盘中。可以预见，这种方式在效率上是最低的。至于无同步，无需多言，我想大家都能正确的理解它。

2). 由于该机制对日志文件的写入操作采用的是append模式，因此在写入过程中即使出现宕机现象，也不会破坏日志文件中已经存在的内容。然而如果我们本次操作只是写入了一半数据就出现了系统崩溃问题，不用担心，在Redis下一次启动之前，我们可以通过redis-check-aof工具来帮助我们解决数据一致性的问题。

3). 如果日志过大，Redis可以自动启用rewrite机制。即Redis以append模式不断的将修改数据写入到老的磁盘文件中，同时Redis还会创建一个新的文件用于记录此期间有哪些修改命令被执行。因此在进行rewrite切换时可以更好的保证数据安全性。

4). AOF包含一个格式清晰、易于理解的日志文件用于记录所有的修改操作。事实上，我们也可以通过该文件完成数据的重建。

有哪些呢？

1). 对于相同数量的数据集而言，AOF文件通常要大于RDB文件。RDB 在恢复大数据集时的速度比 AOF 的恢复速度要快。

2). 根据同步策略的不同，AOF在运行效率上往往会慢于RDB。总之，每秒同步策略的效率是比较高的，同步禁用策略的效率和RDB一样高效。

二者选择的标准，就是看系统是愿意牺牲一些性能，换取更高的缓存一致性（aof），还是愿意写操作频繁的时候，不启用备份来换取更高的性能，待手动运行save的时候，再做备份（rdb）。rdb这个就更有些 eventually consistent的意思了。

### 常用配置

RDB持久化配置

Redis会将数据集的快照dump到dump.rdb文件中。此外，我们也可以通过配置文件来修改Redis服务器dump快照的频率，在打开6379.conf文件之后，我们搜索save，可以看到下面的配置信息：

save 900 1 #在900秒(15分钟)之后，如果至少有1个key发生变化，则dump内存快照。

save 300 10 #在300秒(5分钟)之后，如果至少有10个key发生变化，则dump内存快照。

save 60 10000 #在60秒(1分钟)之后，如果至少有10000个key发生变化，则dump内存快照。

AOF持久化配置

在Redis的配置文件中存在三种同步方式，它们分别是：

appendfsync always #每次有数据修改发生时都会写入AOF文件。

appendfsync everysec #每秒钟同步一次，该策略为AOF的缺省策略。

appendfsync no #从不同步。高效但是数据不会被持久化。

## Redis的分布式锁实现

Redis分布式锁的基本流程并不难理解，但要想写得尽善尽美，也并不是那么容易。在这里，我们需要先了解分布式锁实现的三个核心要素：

### 加锁

最简单的方法是使用setnx命令。key是锁的唯一标识，按业务来决定命名。比如想要给一种商品的秒杀活动加锁，可以给key命名为 “lock\_sale\_商品ID” 。而value设置成什么呢？我们可以姑且设置成1。加锁的伪代码如下：

setnx（key，1）

当一个线程执行setnx返回1，说明key原本不存在，该线程成功得到了锁；当一个线程执行setnx返回0，说明key已经存在，该线程抢锁失败。

### 解锁

有加锁就得有解锁。当得到锁的线程执行完任务，需要释放锁，以便其他线程可以进入。释放锁的最简单方式是执行del指令，伪代码如下：

del（key）

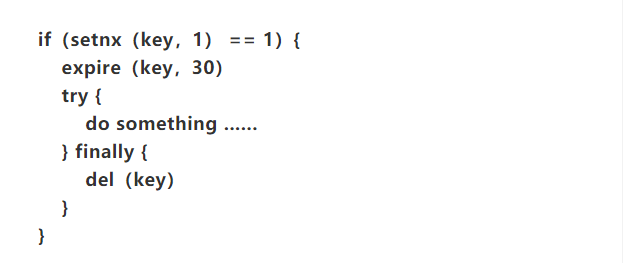
释放锁之后，其他线程就可以继续执行setnx命令来获得锁。

### 锁超时

锁超时是什么意思呢？如果一个得到锁的线程在执行任务的过程中挂掉，来不及显式地释放锁，这块资源将会永远被锁住，别的线程再也别想进来。

所以，setnx的key必须设置一个超时时间，以保证即使没有被显式释放，这把锁也要在一定时间后自动释放。setnx不支持超时参数，所以需要额外的指令，伪代码如下：

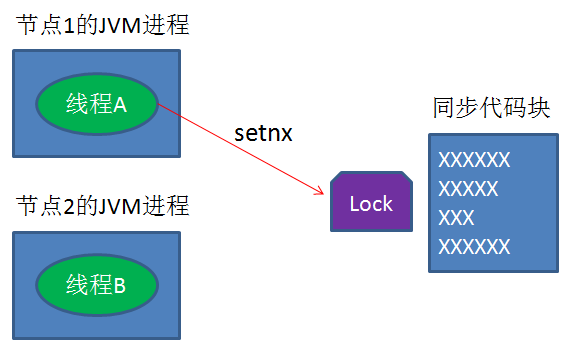
expire（key， 30）



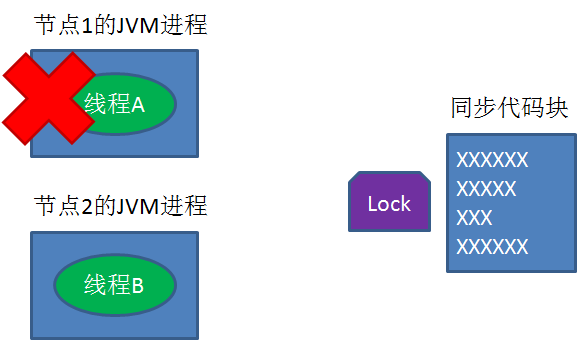
### 问题

1. setnx和expire的非原子性

设想一个极端场景，当某线程执行setnx，成功得到了锁：



setnx刚执行成功，还未来得及执行expire指令，节点1 Duang的一声挂掉了。



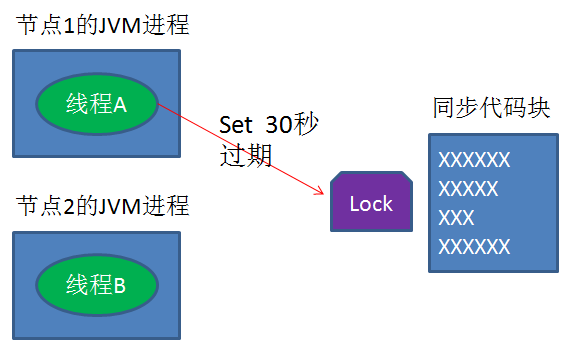
这样一来，这把锁就没有设置过期时间，变得“长生不老”，别的线程再也无法获得锁了。

怎么解决呢？setnx指令本身是不支持传入超时时间的，幸好Redis 2.6.12以上版本为set指令增加了可选参数，伪代码如下：

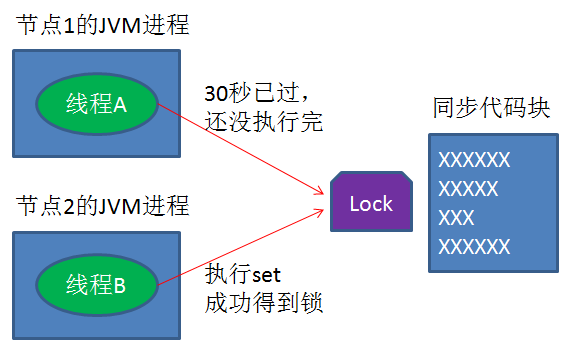
set（key，1，30，NX）

**del 导致误删**

又是一个极端场景，假如某线程成功得到了锁，并且设置的超时时间是30秒。



如果某些原因导致线程A执行的很慢很慢，过了30秒都没执行完，这时候锁过期自动释放，线程B得到了锁。



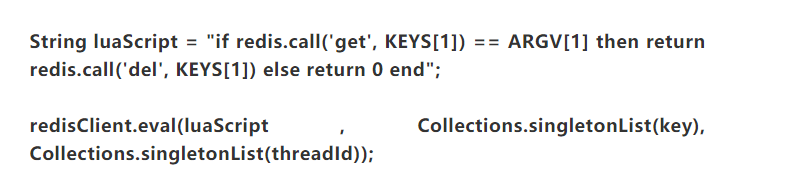
随后，线程A执行完了任务，线程A接着执行del指令来释放锁。但这时候线程B还没执行完，线程A实际上删除的是线程B加的锁。

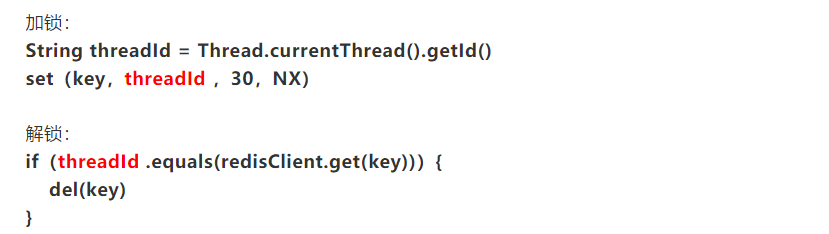
怎么避免这种情况呢？可以在del释放锁之前做一个判断，验证当前的锁是不是自己加的锁。

至于具体的实现，可以在加锁的时候把当前的线程ID当做value，并在删除之前验证key对应的value是不是自己线程的ID。

但是，这样做又隐含了一个新的问题，判断和释放锁是两个独立操作，不是原子性。

我们都是追求极致的程序员，所以这一块要用Lua脚本来实现：





**这样一来，验证和删除过程就是原子操作了。**

## 如何保证缓存和数据库一致性

### 不更新缓存，而是删除缓存

大部分观点认为，做缓存不应该是去更新缓存，而是应该删除缓存，然后由下个请求去去缓存，发现不存在后再读取数据库，写入缓存。

**原因一：线程安全角度**

**同时有请求A和请求B进行更新操作，那么会出现**

（1）线程A更新了数据库

（2）线程B更新了数据库

（3）线程B更新了缓存

（4）线程A更新了缓存

这就出现请求A更新缓存应该比请求B更新缓存早才对，但是因为网络等原因，B却比A更早更新了缓存。这就导致了脏数据，因此不考虑。

**原因二：业务场景角度**

（1）如果你是一个写数据库场景比较多，而读数据场景比较少的业务需求，采用这种方案就会导致，数据压根还没读到，缓存就被频繁的更新，浪费性能。

（2）如果你写入数据库的值，并不是直接写入缓存的，而是要经过一系列复杂的计算再写入缓存。那么，每次写入数据库后，都再次计算写入缓存的值，无疑是浪费性能的。显然，删除缓存更为适合。

其实如果业务非常简单，只是去数据库拿一个值，写入缓存，那么更新缓存也是可以的。但是，淘汰缓存操作简单，并且带来的副作用只是增加了一次cache miss，建议作为通用的处理方式。

### 先操作缓存，还是先操作数据库

那么问题就来了，我们是先删除缓存，然后再更新数据库，还是先更新数据库，再删缓存呢？

对于一个不能保证事务性的操作，一定涉及“哪个任务先做，哪个任务后做”的问题，解决这个问题的方向是：如果出现不一致，谁先做对业务的影响较小，就谁先执行。

假设先淘汰缓存，再写数据库：第一步淘汰缓存成功，第二步写数据库失败，则只会引发一次Cache miss。

假设先写数据库，再淘汰缓存：第一步写数据库操作成功，第二步淘汰缓存失败，则会出现DB中是新数据，Cache中是旧数据，数据不一致

。

沈剑老师说的没有问题，不过没完全考虑好并发请求时的数据脏读问题，让我们再来看看孤独烟老师《分布式之数据库和缓存双写一致性方案解析》：

**先删缓存，再更新数据库**

该方案会导致请求数据不一致

同时有一个请求A进行更新操作，另一个请求B进行查询操作。那么会出现如下情形:

（1）请求A进行写操作，删除缓存

（2）请求B查询发现缓存不存在

（3）请求B去数据库查询得到旧值

（4）请求B将旧值写入缓存

（5）请求A将新值写入数据库

上述情况就会导致不一致的情形出现。而且，如果不采用给缓存设置过期时间策略，该数据永远都是脏数据。

所以先删缓存，再更新数据库并不是一劳永逸的解决方案，再看看先更新数据库，再删缓存这种方案怎么样？

先更新数据库，再删缓存这种情况不存在并发问题么？

不是的。假设这会有两个请求，一个请求A做查询操作，一个请求B做更新操作，那么会有如下情形产生

（1）缓存刚好失效

（2）请求A查询数据库，得一个旧值

（3）请求B将新值写入数据库

（4）请求B删除缓存

（5）请求A将查到的旧值写入缓存

ok，如果发生上述情况，确实是会发生脏数据。

发生上述情况有一个先天性条件，就是步骤（3）的写数据库操作比步骤（2）的读数据库操作耗时更短，才有可能使得步骤（4）先于步骤（5）。可是，大家想想，数据库的读操作的速度远快于写操作的（不然做读写分离干嘛，做读写分离的意义就是因为读操作比较快，耗资源少），因此步骤（3）耗时比步骤（2）更短，这一情形很难出现。

**先更新数据库，再删缓存依然会有问题，不过，问题出现的可能性会因为上面说的原因，变得比较低！**

所以，如果你想实现基础的缓存数据库双写一致的逻辑，那么在大多数情况下，在不想做过多设计，增加太大工作量的情况下，请先更新数据库，再删缓存!

### 我非要数据库和缓存数据强一致怎么办

那么，如果我非要保证绝对一致性怎么办，先给出结论：

没有办法做到绝对的一致性，这是由CAP理论决定的，缓存系统适用的场景就是非强一致性的场景，所以它属于CAP中的AP。

所以，我们得委曲求全，可以去做到BASE理论中说的最终一致性。

最终一致性强调的是系统中所有的数据副本，在经过一段时间的同步后，最终能够达到一个一致的状态。因此，最终一致性的本质是需要系统保证最终数据能够达到一致，而不需要实时保证系统数据的强一致性

大佬们给出了到达最终一致性的解决思路，主要是针对上面两种双写策略（先删缓存，再更新数据库/先更新数据库，再删缓存）导致的脏数据问题，进行相应的处理，来保证最终一致性。

### 缓存延时双删

问：先删除缓存，再更新数据库中避免脏数据？

答案：采用延时双删策略。

上文我们提到，在先删除缓存，再更新数据库的情况下，如果不采用给缓存设置过期时间策略，该数据永远都是脏数据。

那么延时双删怎么解决这个问题呢？

（1）先淘汰缓存

（2）再写数据库（这两步和原来一样）

（3）休眠1秒，再次淘汰缓存

这么做，可以将1秒内所造成的缓存脏数据，再次删除。

那么，这个1秒怎么确定的，具体该休眠多久呢？

针对上面的情形，读者应该自行评估自己的项目的读数据业务逻辑的耗时。然后写数据的休眠时间则在读数据业务逻辑的耗时基础上，加几百ms即可。这么做的目的，就是确保读请求结束，写请求可以删除读请求造成的缓存脏数据。

**如果你用了mysql的读写分离架构怎么办？**

ok，在这种情况下，造成数据不一致的原因如下，还是两个请求，一个请求A进行更新操作，另一个请求B进行查询操作。

（1）请求A进行写操作，删除缓存

（2）请求A将数据写入数据库了，

（3）请求B查询缓存发现，缓存没有值

（4）请求B去从库查询，这时，还没有完成主从同步，因此查询到的是旧值

（5）请求B将旧值写入缓存

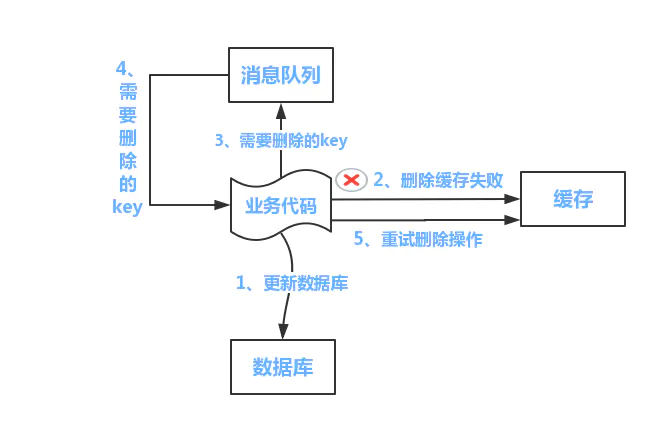
（6）数据库完成主从同步，从库变为新值

上述情形，就是数据不一致的原因。还是使用双删延时策略。只是，睡眠时间修改为在主从同步的延时时间基础上，加几百ms。

### 删缓存失败了怎么办：重试机制

看似问题都已经解决了，但其实，还有一个问题没有考虑到，那就是删除缓存的操作，失败了怎么办？比如延时双删的时候，第二次缓存删除失败了，那不还是没有清除脏数据吗？

解决方案就是再加上一个重试机制，保证删除缓存成功。



流程如下所示

（1）更新数据库数据；

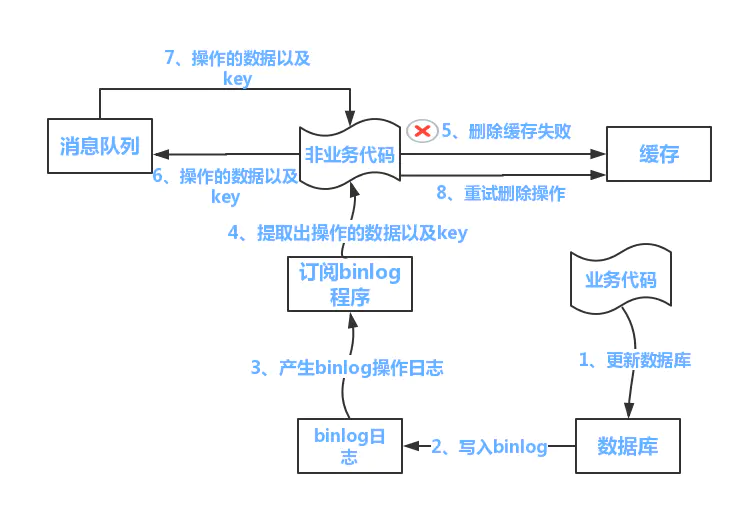
（2）缓存因为种种问题删除失败

（3）将需要删除的key发送至消息队列

（4）自己消费消息，获得需要删除的key

（5）继续重试删除操作，直到成功

然而，该方案有一个缺点，对业务线代码造成大量的侵入。于是有了方案二，在方案二中，启动一个订阅程序去订阅数据库的binlog，获得需要操作的数据。在应用程序中，另起一段程序，获得这个订阅程序传来的信息，进行删除缓存操作。



流程如下图所示：

（1）更新数据库数据

（2）数据库会将操作信息写入binlog日志当中

（3）订阅程序提取出所需要的数据以及key

（4）另起一段非业务代码，获得该信息

（5）尝试删除缓存操作，发现删除失败

（6）将这些信息发送至消息队列

（7）重新从消息队列中获得该数据，重试操作。

而读取binlog的中间件，可以采用阿里开源的cana

好了，到这里我们已经把缓存双写一致性的思路彻底梳理了一遍，下面就是我对这几种思路徒手写的实战代码，方便有需要的朋友参考。

## 缓存和数据库一致性实战

https://www.jianshu.com/p/93f16ca6b396