# Redis学习目标

RedisDesktopManager图形化管理工具

Redis的学习目标：

应用场景：缓存、分布式锁、集群时session共享、排行榜、最新N条数据、共同关注、计数器、分布式系统全局序列号、消息队列、消息推送、电商购物车

redis数据结构及其应用场景：String、List、Set、sortedSet、Hash

redis事务、管道

redis持久化：RDB、AOF，其异同

主从复制：读写分离、主从同步（快照同步、增量同步）、配置信息（从节点配置、持久化配置）、缺点

哨兵机制：高可用

集群模式：特点（分布式存储、节点互联、失效监测、无中间代理层）、工作机制（槽位、CRC16算法、主从、哨兵、节点互联）、缺点

主从复制提高读写QPS、哨兵机制解决高可用、集群模式解决扩容问题。

SpringBoot操作：工厂类、redisTemplate、SpringCache注解

缓存相关问题：并发竞争key问题、双写一致、缓存穿透、缓存雪崩、缓存预热

## 概述

Redis是一个基于内存的key-value数据库，整个数据库都被加载到内存中，定期通过异步把数据持久化到硬盘上。

Redis的主要缺点是数据库容量容易收到物理内存的限制。不能用作海量数据的高性能读写，因此redis适合的场景主要局限于数据容量小的高性能操作上和运算上。

redis-cli [-h host] [-p port] [-a password] --访问远程服务器

### 优缺点

优点：

1）读写速度快。每秒可处理超过10万次读写操作

2）支持数据持久化。可以将内存中的数据保存到磁盘上，重启的时候再次加载使用

3）支持多种数据类型。有string、list、set、sortedSet、hash五种数据结构。

4）支持事务操作。Redis所有操作都是原子性的，同时Redis还支持对几个操作合并后的原子性执行。

6）读写分离。如master-slave模式

### Redis应用场景

--1）缓存

Redis缓存对象分类：

--1、数据库中单条数据（key=表名+id）。需要在更新时更新缓存，因此不适用频繁更新的数据。

--2、不分页、不实时（多表查询）的列表（key=方法名）。

--3、分页、不实时的列表：把分页结果放到一个map（key=分页标识、value=分页结果）存入list中（key=方法名），设定一定缓存时间expire，这样通过方法名lrange获取有分页列表的数据。

--2）分布式锁

在分布式系统上的某方法同一时间仅允许一条线程执行。比如订单库存量。若单点系统时可以使用synchronized，集群系统使用redis中的setnx和getset方法，对应redisTemplate的setIfAbsent和getAndSet方法，集群系统也可使用zk实现分布式锁，还可以用spring自带的分布式锁。

--相关操作如下（也可以使用spring的@Caching注解）：

setnx lock true #获取锁成功

执行该方法…

del lock #释放锁

--3）集群时的session共享（nginx+tomcat负载均衡配置）

spring session + redis实现session共享

在使用Nginx+Tomcat实现负载均衡时，由于Nginx对不同的请求分发到某一个Tomcat，项目在运行时可能在不同的Tomcat容器里，因此session不同步或者丢失的问题。

--4）排行榜->sortedSet有序集合

--1.设置初始化数据（zadd命令）

zadd userGrade 80 user1

zadd userGrade 80 user1

--2.查看所有玩家分数，由低到高排列（zrange命令）

zrange grade 0 -1 withscores

--3.查看前三名玩家分数，由高到低排列（zrevrange命令）

zrevrange grade 0 2 withscores

--4.增减玩家分数（zincrby命令，正数为增、负数为减）

zincrby grade -1 user1

--移除某玩家（zrem命令）

--5）取最新N条数据->list列表

将最新的500条评论放在list列表中，超出列表部分从数据库中获取。

--6）共同关注->set集合

--7）消息队列。

不推荐，一般使用使用rabbitMq或Kafka中间件

--8）计数器

incr article:readcount:{文章id}

get article:readcount:{文章id}

--9）分布式系统全局序列号

incrby orderId 1000 //redis批量生成序列号提升性能

--10）list应用场景

Stack(栈) = lpush + lpop

Queue(队列) = lpush + rpop

Blocking MQ(阻塞队列) = lpush + Brpop

微信公共号消息推送：

aa关注了bb、cc这两个人。

1、当bb发布消息msg1，则》lpush msg:aa msg1;

2、当cc发布消息msg2，则》lpush msg:aa msg2;

3、aa查看前五条最新消息》lrange msg:aa 0 5;

-11）hash应用场景

Hash电商购物车：

备注：用户Id为key，商品id为field，商品数量为value

1.添加商品》hset cart:userId goodsId goodsNum

2.添加数量》hincrby cart:userId goodsId num

3.商品总数》hlen cart:userId

4.删除商品》hdel cart:userId goodsId

5.获取购物车所有商品》hgetall cart:userId

## 缓存技术对比

### EhCached介绍

Ehcache直接在jvm虚拟机中运行，速度快，效率高，适用于单体应用；Ehcache不适合分布式应用，其缓存数据无法在其它应用共享。

Ehcache常用于单体项目中保存shiro登录用户的权限数据。

### memcached介绍

Memcached只有String这一种类型类型。

Memcached可以利用多核优势，单实例吞吐量极高，可以达到几十万QPS，适用于最大程度扛量。

### memcached与redis比较

数据类型：redis支持多种数据类型，memcached仅支持简单key/value数据类型。

数据持久化：redis支持rdb和aof两种。memcached不支持数据持久化

模型：redis使用单线程，而memcached使用多线程。redis使用单核，而memcached可利用多核。在处理小的文件时redis会比memcached有更高的效率，但是在100K以上时，memcached的效率就会更高一些。

redis只支持单线程，其性能受限于CPU性能，即取决于数据结构、数据大小、以及服务器硬件性能，其在日常环节中的QPS高峰约为1-2w。而memcached具有多核优势，其单实例的吞吐量极高，性能主要取决于存储的key及value的字节大小以及服务器硬件性能，其在日常环节中的QPS高峰约为4-6w。（读写性能qps）

总结：一昧的使用redis来进行数据缓存系统，并不是最佳的选择，如果项目根本就不需要复杂的数据结构呢？如果项目的数据只有一小部分使用了缓存，根本不需要使用持久化和备份呢？如果项目不需要在服务器之间进行数据同步呢？在诸如此类情况下，使用redis反而是增加项目成本。

## 安装

redis文件夹：

redis-benchmark.exe # 基准测试

redis-check-aof.exe # aof

redischeck-dump.exe # dump

redis-cli.exe # 客户端

redis-server.exe # 服务器

redis.windows.conf # 配置文件

## Redis数据结构

### String(字符串)

String是redis最基本的类型，一个key对应一个value。一个键最大能存储512MB。

string类型是二进制安全的。即redis的string可以包含任何数据，比如jpg图片或序列化的ruby对象。

相关操作：

set key value --设值

get key --获取值

setnx key value --设值，若key已存在时失败（用于分布式锁）

getset key value --设新值，并返回旧值

strlen key --获取key的字符长度

append key value --将value追加

decr/incr key --递减/递增

decrby/incrby key decrement --减少/增加decrement

ttl key --查看key的过期时间，-1表示为永久

setex key seconds value --设值value，并设置该key的过期时间（秒数）

### 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）。

列表最多可存储2^32-1个元素 (4294967295, 每个列表可存储40多亿)。

相关操作：

lpush key value [value...] --在头部插入所有value

rpush key value [value...] --在尾部插入所有value

lset key index value --设置指定位置index的元素

lpop key --头部弹出元素

rpop key --尾部弹出元素

blpop key [key...] timeout --头部弹出元素，阻塞等待timeout秒

brpop key [key...] timeout --尾部弹出元素，阻塞等待timeout秒

lrange key start stop --获取指定范围（start~stop）的元素列表，0为第一个元素，-1为最后一个元素

### Set(无序集合)

Redis的Set是string类型的无序集合。集合的概念就是一堆不重复值的组合。

集合中最大的成员数为2^32-1(4294967295, 每个集合可存储40多亿个成员)。

相关操作：

sadd key member [member...] --O(n)，插入key中为存在的元素，若该成员存在则被忽略

scard key --O(1)，获取set中的成员数量

sismember key member --O(1)，判断memver是否已存在

smembers key --O(n)，获取Set集合的所有成员

srem key member [member...] --O(n)，移除指定成员

求交集(sinter)、并集(sunion)、差集(sdiff)：

sadd aa t1 t2 t3

sadd bb t2 t3 t4

sinter aa bb #求交集，返回[t2, t3]

sunion aa bb #求并集，返回[t1, t2, t3, t4]

sdiff aa bb #求差集，返回[t1]

### SortedSet(有序集合)

SortedSet中每个元素都会关联一个double类型的分数score。redis正是通过分数来为集合中的成员进行从小到大的排序。

SortedSet中的成员都是按照分数从低到高的顺序存储

相关操作：

zadd key score member [score member...] --O(n)，添加指定成员及其分数

zcard key --O(1)，获取zset中的成员数量

zcount key min max --O(log(n)+m)，获取分数score在min和max之间的成员数量

zincrby key increment member --O(log(n)) --为指定的member添加指定分数increment

zrangebyscore key min max [withscores][limit offset count] --O(log(n)+m)，获取分数在min和max之间的成员

zrem key member [member...] --O(m log(n))，移除指定的成员

zcore key member --O(1)，获取指定member的分数score

### Hash(哈希)

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。

每个hash可以存储2^32-1键值对(40多亿)

相关操作：

hset key field value --O(1)，为key设定键值对，field不存在则新增，存在则修改

hsetnx key field value --存储一个不存在的哈希表key的键值

hget key field --O(1)，获取key中指定field对应value值

hgetall key -- O(n)，获取key包含的所有field/value

hkeys key --O(n)，获取key包含的所有field

hvals key --O(n)，获取key包含的所有value

hmset key field value [field value...] -- O(n)，在一个哈希表key中存储多个键值对

hmget key field [field] -- O(n)，批量获取哈希表key对应的field键值

hdel key field [field] //删除哈希表key中的field键值

hlen key --O(1)，获取key包含的field数量

hexists key field --O(1)，判断指定key中指定field是否存在

hincrby key field increment --为哈希表key中field键的值增加increment

应用场景：

在Mencached中，我们经常将一些结构化的信息打包成hashMap，序列化成一个字符串的值（一般是JSON格式）在客户端存储。修改其中某一项时，通常需要将字符串JSON取出来，修改某一项的值，在序列化成字符串JSON存储。缺点是：消耗必定是很大的，也不适用于一些可能并发操作的场合（如两个并发的操作都需要修改积分）。

redis的Hash结构可以使你在数据库中的update操作一样，只修改某一项属性值。

存储、读取、修改用户属性。

## Redis事务

Redis的事务特征：

-1）在事务中所有命令都会被串行化的顺序执行，在事务执行期间，redis不会再为其它客户端的请求提供任何服务，从而保证了事务中的所有命令被原子的执行

-2）和关系型数据库的事务不同，redis事务中某一条命令执行失败，其后的命令仍然会被执行！

-3）开启事务用multi命令，通过exec/discard命令来提交/回滚该事务内的所有操作。

-4）当使用Append-Only模式时，Redis会通过调用系统函数write将该事务内的所有写操作在本次调用中全部写入磁盘。然而如果在写入的过程中出现系统崩溃，如电源故障导致的宕机，那么此时也许只有部分数据被写入到磁盘，而另外一部分数据却已经丢失。Redis服务器会在重新启动时执行一系列必要的一致性检测，一旦发现类似问题，就会立即退出并给出相应的错误提示。此时，我们就要充分利用Redis工具包中提供的redis-check-aof工具，该工具可以帮助我们定位到数据不一致的错误，并将已经写入的部分数据进行回滚。修复之后我们就可以再次重新启动Redis服务器了。

命令列表：

multi --用于标记事务的开始，其后执行的命令都将被存入命令队列，直到执行exec时，这些命令才会被原子的执行

exec --执行再一个事务内命令队列中的所有命令，同时将当前连接的状态恢复为正常状态，即非事务状态

discard --回滚事务队列中的所有命令，同时将当前连接的状态恢复为正常状态，即非事务状态

watch key [key...] --在multi命令执行之前，可以指定待监控的keys，然而再执行exec之前，如果被监控的keys发生修改，exec将放弃执行该事务队列中的所有命令。始终返回OK

unwatch --取消当前事务中指定监控的Keys，如果执行了EXEC或DISCARD命令，则无需再手工执行该命令了，因为在此之后，事务中所有被监控的Keys都将自动取消。始终返回OK

Redis事务设置方式：

-1）stringRedisTemplate.setEnableTransactionSupport(true); //默认为false

-2）使用 SessionCallback, 在同一个 Redis Connection 中执行事务: 成功执行事务

测试：

@Test

public void testMultiSuccess() {

// 开启事务支持，在同一个 Connection 中执行命令

stringRedisTemplate.setEnableTransactionSupport(true);

stringRedisTemplate.multi();

stringRedisTemplate.opsForValue().set("name", "username");

stringRedisTemplate.opsForValue().set("age", "19");

System.out.println(stringRedisTemplate.exec()); // [true, true]

}

不开启Redis事务时使用stringRedisTemplate的multi()和exec()方法会报错：

\*错误用法：没有开启事务支持，事务执行会失败

抛出异常：Error in execution; nested exception is io.lettuce.core.RedisCommandExecutionException: ERR EXEC without MULTI

原因：enableTransactionSupport值默认为false，导致每一个RedisConnection都是重新获取的。所以执行的multi和exec方法不在同一Connection中

# 存储机制-持久化

相关链接：

https://www.cnblogs.com/dengtr/p/5085287.html

https://blog.csdn.net/zhu\_tianwei/article/details/44892737

https://blog.csdn.net/zangdaiyang1991/article/details/91464488

## 问题描述

redis是内存数据库，它把数据存储在内存中，这样加快读取速度的同时页对数据安全性产生新的问题，即当redis所在服务器发生宕机后，redis里的所有数据会丢失。

解决：redis提供了持久化功能-RDB和AOF，即将内存中的数据写入硬盘中。

两种持久化方式：

-1）RDB持久化：会将内存中的数据写入dump.rdb文件。Redis重启时将该文件的数据重新加载进内存。

如：save 900 1 #表示900秒内至少有1次写操作，则触发RDB

-2）AOF持久化：默认关闭，将所有的写操作追加到appendonly.aof文件中。Redis重启时将该文件的写指令从头到尾执行来还原数据集。

如：appendfsync everysec #每秒进行一次文件同步

附：两种都使用时，redis会优先使用AOF来还原数据集。

## RDB持久化

工作原理：满足一定规则时，会将内存中的数据全量生成一份副本存储到硬盘上（写入dump.rdb文件），这个过程称为“快照”。

你可以配置redis的持久化策略，例如数据集中每N秒钟有超过M次更新，就将数据写入磁盘；或者可以手工调用SAVE或BGSAVE命令。

一定规则如下：

1. 在指定时间间隔内发生了指定次数的写操作（save、bgsave）
2. 执行flushall命令
3. 执行复制（replication）时

RDB持久化分两种：SAVE和BGSAVE

1）SAVE：阻塞式持久化，redis的主进程把内存里的数据库状态写入到RDB文件中，在该文件创建结束之前redis不能处理任何命令请求。

2）BGSAVE：非阻塞式持久化，它会创建一个子进程专门把内存中的数据库状态写入RDB文件里，同时主进程还可以处理来自客户端的命令请求。这个内存使用率会加大！！

快照执行的过程：

1. redis使用fork函数复制一份当前进程（父进程）的副本（子进程）；
2. 父进程继续处理来自客户端的请求，子进程开始将内存中的数据写入硬盘中的临时文件；
3. 当子进程写完所有的数据后，用该临时文件替换旧的rdb文件，至此，一次快照操作完成。

注：rdb文件存储的是fork函数发生那一刻的内存数据。原因：fork函数发生，父进程和子进程共享同一块内存数据，当父进程需要修改其中的某片数据时，操作系统会将该片数据复制一份以保证子进程不受影响，因此fork之后修改的数据没有写入rdb文件。

配置RDB：

修改配置文件（redis/conf/redis.conf）：

--修改配置文件需要重启redis服务器才能生效

save 900 1 #表示900秒内至少有1次写操作，则触发RDB

save 300 10

save 60 10000

dbfilename dump.rdb #指定RDB的文件名称

dir /data/dbs/redis/6381 #指定RDB文件的存放目录

> redis-cli #重启redis服务器

> config get save #查看redis持久化配置

1) "save"

2) "900 1 300 10"

> config set save "2100 20" #修改redis持久化配置

通过rdb文件恢复数据：

将dump.rdb文件拷贝到redis中的bin目录下，重启redis服务器即可。

RDB优点：

-1）适合大规模的数据恢复

-2）如果业务对数据完整性和一致性要求不高，rdb是很好的选择

RDB缺点：

-1）rdb并不健壮，当系统停止、或无意中redis被kill掉，最后写入redis的数据就会丢失。

-2）备份时占用内存翻倍。

## AOF持久化

工作原理：将所有的写操作追加到appendonly.aof文件中。Redis重启时将该文件的写指令从头到尾执行来还原数据集。

具体步骤：

AOF的持久化是通过命令追加、文件写入和文件同步三个步骤实现。

-1）当redis开启AOF后，服务端每执行一次写操作（如set、sadd、rpush）就会把该条命令追加到一个单独的AOF缓冲区的末尾；

-2）然后把AOF缓冲区的数据写入AOF文件里；

-3）最后将文件同步。

配置AOF：

修改配置文件（redis/conf/redis.conf）：

--修改配置文件需要重启redis服务器才能生效

appendonly yes #开启AOF，默认关闭

appendfsync everysec #指定AOF持久化策略，每秒进行一次文件同步。

appendfilename "appendonly.aof" #指定AOF文件名称

dir /data/dbs/redis/6381 #指定AOF文件存放目录

指定AOF持久化策略(appendfsync)：

如：appendfsync no

-1）always：每执行一个写操作进行一次文件同步，把内存缓冲区的aof缓冲数据真正写入aof文件里。

-2）everysec（默认）：每秒进行一次文件同步，把内存缓冲区的aof缓冲数据真正写入aof文件里。

-3）no：不同步，即当内存缓冲区的空间被填满或超过了设定的时限后系统自动执行文件同步。

配置重写触发机制：

> auto-aof-rewrite-percentage 100 #当前写入日志文件的大小占到初始日志文件大小的某个百分比时触发Rewrite

> auto-aof-rewrite-min-size 64mb #本次Rewrite最小的写入数据量

重写原理：aof机制是将写操作追加到aof文件中，这样会使得文件原来越大。因此redis在实现重写时，会fork出一条新进程读取内存中的数据，并重新写到一个临时文件中。并没有读取旧文件（内容过大）。最后替换旧的aof文件。

附：当aof文件的大小是上次rewrite后大小的2倍，且文件大于64M时触发。

通过aof文件恢复数据：

将appendonly.aof文件拷贝到redis中的bin目录下，重启redis服务器即可。

AOF机制中为什么会有同步操作？

答：为了提高文件的写效率，都会有一个写入策略，即当你往硬盘写入数据时，操作系统不是实时的将数据写入硬盘，而是先把数据暂时保存在一个内存缓冲区里，等到这个内存缓冲区被填满或超过了设定时限后才会真正地把缓冲区内的数据写入硬盘中。即当redis进行第二步文件写入时，从用户的角度看是已经把AOF缓冲区里的数据写入到AOF文件了，但对系统而言只不过是把AOF缓冲区的内容放到了另一个内存缓冲区里而已，之后redis还需要进行文件同步把该内存缓冲区里的数据真正写入硬盘上才算是完成了一次持久化。

## RDB和AOF比较

-1）redis默认开启rdb持久化方式，在指定时间间隔内、发生指定次数的写操作，则将内存中的数据写入到磁盘中。

-2）rdb适合大规模的数据恢复（数据文件较小）。但rdb每次持久化都会内存翻倍。

-3）aof手动开启，默认每秒将写指令追加到aof文件中。

-4）aof的数据完整性比rdb高，但记录的内容过多时，在数据恢复中需要执行写指令，因而影响效率。

# 几种常见的配置模式

Redis的主要使用方式有：

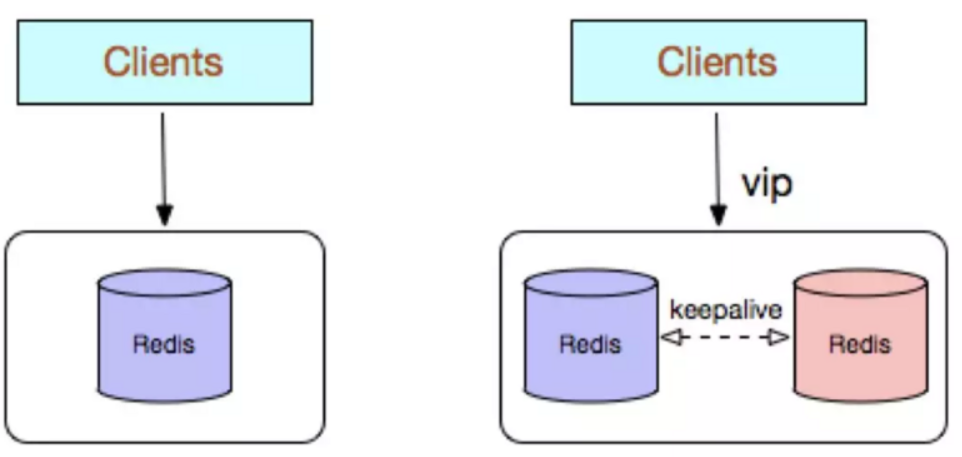
Redis单副本

Redis多副本（主从）

Redis Sentinel（哨兵）

Redis cluster

## Redis单副本



Redis单副本：采用单个Redis节点部署架构，没有备用节点实时同步数据，不提供数据持久化和备份策略，适用于数据可靠性不高的纯缓存业务场景。

缺点：

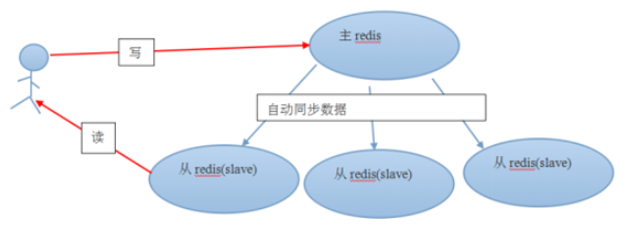
1）当缓存使用。进程重启后数据会丢失

2）高性能受限于单核CPU的处理能力（Redis是单线程机制），CPU为主要颈瓶

## Redis多副本（主从复制）

把原先一台redis服务器做的工作，现在交给多个服务器去完成，可以明显降低每台服务器的工作量。

一个redis服务器负载“写”（增加、修改、删除）数据，其它服务器负载“读”数据。主服务器中的数据会“自动”同步给从服务器。



主从复制：

即redis主从复制可以把主节点的数据复制给从节点。

-1）配置master只能为写操作，slave只能为读操作，这样实现了读写分离。

-2）从节点可以备份主节点的数据，在主节点宕机后，起到接替主节点工作的作用。

-3）为保证主库写的能力，一般不在主库做持久化，而是在从库做持久化。

在没有哨兵机制之前，主从切换需要人工介入，其步骤为：

-1）启动从节点为主节点：slaveof no one

-2）旧主节点的其它从节点变成新主节点的从节点：slaveof new master

-3）通知应用方redis主节点变更，修改客户端调用的地址并重启客户端

-4）旧主节点变成新主节点的从节点：slaveof new master

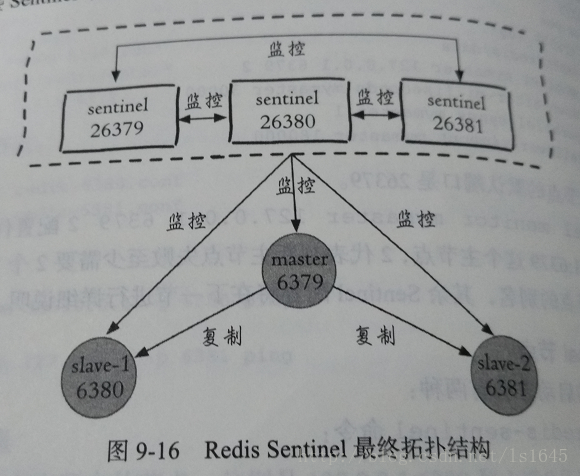
## Redis Sentinel（哨兵）

为了达到redis的高可用，有两种部署方式：1、主从模式+哨兵机制；2、集群模式。哨兵机制是redis2.8开始支持，集群模式是redis3.0开始支持的。

哨兵机制存在意义：

-1）监控主节点和从节点是否正常运行

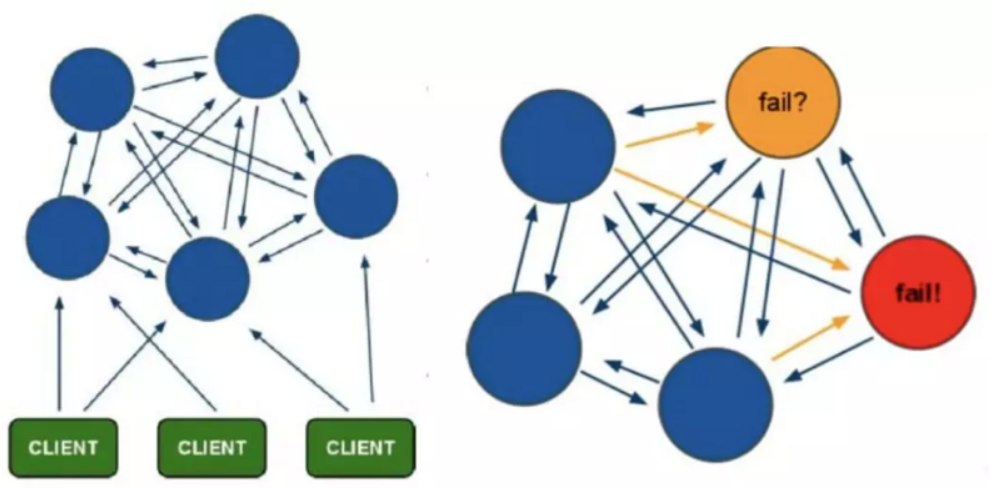
-2）主节点在出现宕机后，自动将从节点切换为主节点



通过哨兵机制实现redis主从复制及Java调用：

相关教程：https://blog.csdn.net/weixin\_40138160/article/details/78911864

## Redis Cluster（集群）



# 主从复制（Redis Replication）

相关链接：

https://blog.csdn.net/zangdaiyang1991/article/details/91464488

## 概念

核心功能：读写分离、主从同步（快照同步、增量同步）

主从复制模式中包含一个主节点（master）与一个或多个从节点（slave）。

读写分离：master节点负责处理写请求，slave节点负责处理读请求。好处：对于写请求少，读请求多的场景，例如电商详情页，通过这种读写分离的操作可以大幅提高并发量，通过增加redis从节点的数量可以使得redis的QPS达到10W+。

主从同步：master节点处理写请求后，需将数据同步到slave节点。分为快照同步和增量同步。

相关联技术：

哨兵机制-当主节点发生宕机后，从节点会切换成主节点。

集群模式-几个主节点之间做集群。

原理：

-1）slave节点发送SYNC命令给master节点；

-2）若slave节点是第一次连接master节点，会触发一次full resynchronization，master会启动一个后台线程，开始生成一份rdb快照文件，同时会将客户端收到的写命令缓存到内存中，rdb文件生成完毕后，master将该文件发送给slave，slave先写入本地磁盘，然后从磁盘加载到内存中，然后master将内存中的写命令发送给slave，slave也会执行这些写命令；

-3）若不是第一次连接，master node仅赋值给slave缺少的数据即可。

-4）slave节点若因网络故障和master断开，会自动重连，master如果发现有多个slave都来重新连接，仅会启动一个rdb save操作，用一份数据服务所有slave。

## 主从同步分类

### 快照同步：

Redis全量复制一般发生在slave初始化阶段，这时候需要将master上的所有数据都复制一份。

流程如下：

-1）从节点slave连接主节点master后，发送sync命令；

-2）master接收到sync命令后，开始执行bgsave命令生成rdb文件，并使用缓冲区记录此后所执行的所有写命令；

-3）master在bgsave执行完后，向所有slave发送快照文件，并在发送期间记录被执行的写命令；

-4）slave接收到快照文件后丢弃所有旧数据，载入收到的快照数据；

-5）master快照发送完后，开始向slave发送缓冲区的写命令；

-6）slave完成对快照的载入，开始接收命令请求，并执行来自master缓冲区的写命令

### 增量同步：

Redis增量复制是指Slave初始化后开始正常工作时master发生的写操作同步到从服务器的过程。

增量复制的过程主要是master每执行一个写命令就会向slave发送相同的写命令，slave接收并执行收到的写命令。

## 配置文件

步骤：

--1）启动

cd /usr/local/redis-3.2.1/src

./redis-server

--2）修改配置(redis.conf)

port 6379 --端口号修改

requirepass redis123 --设置密码，redis123为密码

#配置主从：主库的配置文件不用修改，从库的配置文件需要修改，因为从库需要绑定主库，以便可以获取主库的数据

slaveof 192.168.1.100 6379 --主库的IP和端口号

masterauth redis123 --主库设定的密码

--3）要让配置文件的属性生效，那么启动的redis时，要将配置文件加上去

cd /usr/local/redis-3.2.1/src

./redis-server redis.conf

--4）客户端远程连接redis

步骤一：config set requirepass 123 --123为密码

步骤二：redis-cli.exe -h 192.168.1.100 -p 6379 --客户端登录

步骤三：auth 123 --认证，123为密码

本地端设置密码后，要使用密码登录；如果Redis重启的话，密码需要重新设置

--5）持久化配置：

主库配置后，为保证主库写的能力，一般不在主库做持久化，而是在从库做持久化

1）主库配置：

--将save注释，不使用rdb

# save 900 1

# save 300 10

# save 60 10000

--不使用aof

appendonly no

2）从库配置：

save 900 1

save 300 10

save 60 10000

appendonly yes

这样做的优缺点：

优点：保证了主库写的能力；

缺点：主库挂掉后，重启主库，然后进行第一次写的动作后，主库会先生成rdb文件，然后传输给从库，从而覆盖从库原先的rdb文件，造成数据丢失。但是第二次写的时候，主库会以快照的方式把数据直接传给从库，不会重新生成rdb文件。

解决方案：先复制从库中的数据到主库后，再启动主库。

# 哨兵机制

背景：使用redis做主从复制时，若master宕机后，redis本身（包括它的很多客户端）都没有实现自动进行主备切换功能。

Redis-Sentinel是redis官方推荐的高可用（HA）解决方案，其本身也是一个独立运行的redis进程，它能监控多个多个master-slave集群，发现master宕机后能进行主备切换。

Sentinel也支持集群：很显然，只使用单个sentinel进程来监控redis集群是不可靠的，当sentinel进程宕掉后，就不能进行master宕机后的主备切换了。

# Redis集群模式

## 概念

背景：哨兵模式解决了主从复制不能自动故障转移，达不到高可用的问题，但还是存在难以在线扩容，Redis容量受限于单机配置的问题。

Redis Cluster模式不建议使用pipeline和multi-keys操作，减少max redirect产生的场景。

Redis采用无中心架构：

1、分布式存储：Redis集群模式实现了分布式存储，即每台节点存储不同的内容，来解决在线扩容的问题。

2、节点互联：所有的redis节点彼此互联(PING-PONG机制)，内部使用二进制协议优化传输速度和带宽。

3、失效检测：节点的fail是通过集群中超过半数的节点检测失效时才生效。

4、无中间代理层：客户端与redis节点直连，客户端不需要连接集群所有节点，连接集群中任何一个可用节点即可。

## Cluster工作机制：

1、在Redis的每个节点上，都有一个插槽（slot），取值范围为0-16383；

2、当我们存取key的时候，Redis会根据CRC16的算法得出一个结果，然后把结果对16384求余数，这样每个key都会对应一个编号在0-16383之间的哈希槽，通过这个值，去找到对应的插槽所对应的节点，然后直接自动跳转到这个对应的节点上进行存取操作；

3、为了保证高可用，Cluster模式也引入主从复制模式，一个主节点对应一个或者多个从节点，当主节点宕机的时候，就会启用从节点；

4、当其它主节点ping一个主节点A时，如果半数以上的主节点与A通信超时，那么认为主节点A宕机了。如果主节点A和它的从节点都宕机了，那么该集群就无法再提供服务了。

## Cluster模式优缺点

优点：

1、无中心架构，数据按照slot分布在多个节点。

2、集群中的每个节点都是平等的关系，每个节点都保存各自的数据和整个集群的状态。每个节点都和其他所有节点连接，而且这些连接保持活跃，这样就保证了我们只需要连接集群中的任意一个节点，就可以获取到其他节点的数据。

3、可线性扩展到1000多个节点，节点可动态添加或删除

4、能够实现自动故障转移，节点之间通过gossip协议交换状态信息，用投票机制完成slave到master的角色转换

缺点：

1、数据通过异步复制，不保证数据的强一致性；

2、slave充当“冷备”，不能缓解读压力；

3、批量操作限制，目前只支持具有相同slot值的key执行批量操作，对mset、mget、sunion等操作支持不友好；

4、key事务操作支持有限，只支持多key在同一节点的事务操作，多key分布不同节点时无法使用事务功能

5、不支持多数据库空间，单机redis可以支持16个db，集群模式下只能使用一个，即db0

# SpringBoot操作Redis

相关链接：

https://www.cnblogs.com/nfcm/p/7833032.html

## 连接redis工厂类

详细步骤：

--1）添加jar包

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-data-redis</artifactId>

</dependency>

--这个starter引入了jedis和spring-data-redis两个redis核心的包

--2）创建redis配置类（配置redis工厂类）

@Configuration

public class RedisConfig {

@Bean

public RedisConnectionFactory redisCF() {

// 如果什么参数都不设置，默认连接本地6379端口

JedisConnectionFactory factory = new JedisConnectionFactory();

factory.setPort(6379);

factory.setHostName("localhost");

return factory;

}

}

--3）测试

前提：启动redis服务（点击: C:\Redis\redis-server.exe）

@Autowired

private RedisConnectionFactory factory;

@Test

public void testRedisCF() {

RedisConnection conn = factory.getConnection();// 得到一个连接

conn.set("hello".getBytes(), "world".getBytes());

System.out.println(new String(conn.get("hello".getBytes()))); //输出world，说明已成功获取连接，并往redis添加数据

}

## 模板类配置(redisTemplate)

我们发现redis的key和value都是byte数组类型（使用很麻烦），于是spring为我们带来了模板（redisTemplate）。

SpringDataRedis提供了两个模板：

1）RedisTemplate（key是String类型，value是Object类型，使用JDK的序列化策略）

2）StringRedisTemplate（key和value都是String类型，都使用String的序列化策略）

### RedisTemplate模板类

**RedisTemplate中定义了对5种数据结构的操作:**

redisTemplate.opsForValue();//操作字符串

redisTemplate.opsForList();//操作list

redisTemplate.opsForSet();//操作set

redisTemplate.opsForZSet();//操作有序set

redisTemplate.opsForHash();//操作hash

实例：

--1）配置模板bean

@Configuration

public class RedisConfig {

@Bean

public RedisConnectionFactory redisCF() {

JedisConnectionFactory factory = new JedisConnectionFactory();

factory.setPort(6379);

factory.setHostName("localhost");

return factory;

}

@Bean

public RedisTemplate redisTemplate(RedisConnectionFactory factory) {

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

template.setConnectionFactory(factory); //设置连接工厂

return template;

}

}

--2）测试

@Autowired

private RedisTemplate<String, Object> redisTemplate;

@Test

public void testRedisTemplate() {

redisTemplate.opsForValue().set("key1", "value1");

System.out.println(redisTemplate.opsForValue().get("key1")); //输出value1

}

### key和value的序列化

当保存一条数据时，key和value都要被序列化成json数据，取出来时被序列化成对象，key和value都会使用序列化器进行序列化。

Spring提供RedisSerializer<T>接口，并提供了多个序列化器（如StringRedisSerializer），用来描述各种序列化。

RedisTemplate会默认使用JdkSerializationRedisSerializer，这意味着key和value都会通过Java进行序列化（需要实现Serializable接口）。

StringRedisTemplate默认会使用StringRedisSerializer。

设置序列化器：

redisTemplate.setKeySerializer(new StringRedisSerializer()); //设置key

redisTemplate.setValueSerializer(new Jackson2JsonRedisSerializer); //设置value

redisTemplate.setHashKeySerializer(hashKeySerializer); //设置hash key

redisTemplate.setHashKeySerializer(hashKeySerializer); //设置hash value

## SpringCache注解

前言：上面的配置只是为了让我们了解原理，实际上我们使用会更简单点。

实例：

--1）重写RedisConfig类

@Configuration

@EnableCaching//开启缓存，自动发现并配置CacheManager

public class RedisConfig extends CachingConfigurerSupport {

@Bean

@Primary

public RedisTemplate redisTemplate(RedisConnectionFactory factory) {

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

template.setConnectionFactory(factory);

return template;

}

// 缓存管理器CacheManager

@Bean

public RedisCacheManager cacheManager(RedisTemplate redisTemplate) {

return new RedisCacheManager(redisTemplate);

}

}

说明：

附：RedisConnectionFactory-> RedisTemplate-> RedisCacheManager

大家发现我们并没有注册RedisConnectionFactory，那是因为spring默认帮我们读取application.properties文件并且注册了一个factorybean。

通过CacheManager可以使用@Cacheable、@CachePut、@CacheEvict这三个注解。

--2）在resources下的application.properties下配置

# REDIS (RedisProperties)

# Redis数据库索引（默认为0）

spring.redis.database=0

# Redis服务器地址

spring.redis.host=127.0.0.1

# Redis服务器连接端口

spring.redis.port=6379

# Redis服务器连接密码（默认为空）

spring.redis.password=

# 连接池最大连接数（使用负值表示没有限制）

spring.redis.pool.max-active=8

# 连接池最大阻塞等待时间（使用负值表示没有限制）

spring.redis.pool.max-wait=-1

# 连接池中的最大空闲连接

spring.redis.pool.max-idle=8

# 连接池中的最小空闲连接

spring.redis.pool.min-idle=0

# 连接超时时间（毫秒）

spring.redis.timeout=0

# Redis相关问题

问：redis缓存和本地缓存用途？

## redis并发竞争key问题

描述：集群环境下多个子系统去set一个key，如某个key1，系统A设置key1=valueA，系统B设置key1=valueB，系统C设置key1=valueC。期望key1的顺序为valuaA->valueB->valueC。

解决方案：

-1）分布式锁+时间戳。时间段valueA<valueB<valueC，若先执行valueB则不执行valueA

-2）利用消息队列，将set方法变成串行访问

## 缓存和数据库双写一致

读请求：先读缓存->无则读数据库->更新缓存

写请求：先更新数据库，后删除缓存。（删除的数据在进行读请求后会重新加入缓存）

## 缓存穿透

### 概念

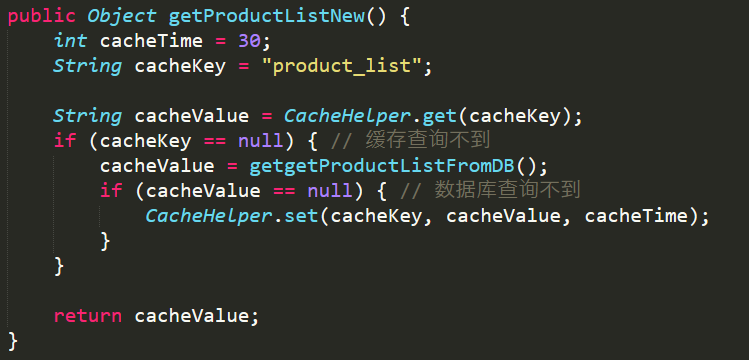
访问不存在的数据，缓存不起作用，导致请求穿透到数据库，若存在大量这样的请求会导致数据库崩溃。

### 解决方案

-1）使用布隆过滤器：使用一个足够大的bitmap，用于存储可能访问的key，不存在的key直接被过滤。

-2）访问key未在DB查询有值，也将空值写入缓存，但设置较短的过期时间。这样第二次到缓存中获取就有值了，而不会继续访问数据库。

方案二的伪代码：



说明：

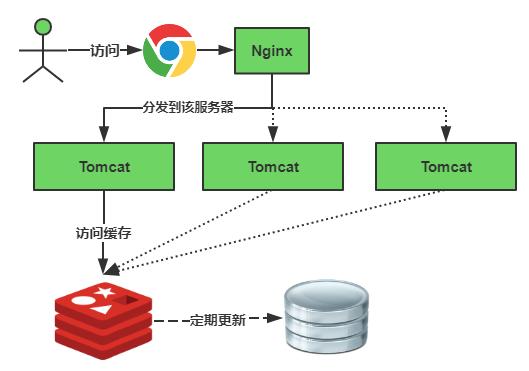
把空结果，也给缓存起来，这样下次同样的请求就可以直接返回空了，即可以避免当查询的值为空时引起的缓存穿透。同时也可以单独设置个缓存区域存储空值，对要查询的key进行预先校验，然后再放行给后面的正常缓存处理逻辑。

## 缓存雪崩

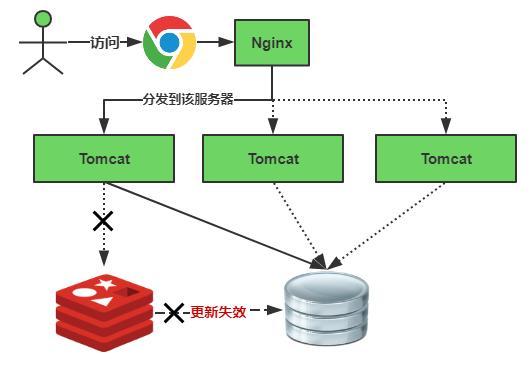
### 概念

当缓存服务器重启或大量缓存数据集中在某一时间段失效，所有原本应该访问缓存的请求都去访问数据库，造成瞬间DB请求量大、压力剧增，引起雪崩。

缓存正常从redis中获取，示意图如下：



缓存失效瞬间，示意图如下：



### 解决方案

三种具体解决方案：

-1）数据库限流（通过加锁或队列来控制操作数据库的线程数量）

-2）设置过期标记更新缓存

-3）缓存失效时间增加1~5分钟随机值，使得缓存不在同一时间失效。

另：做二级缓存，A1为原始缓存，A2为拷贝缓存。A1失效时，可以访问A2，A1缓存的失效时间较短，A2缓存的失效时间为长期。

方案一：加锁排队

一般并发量不高时，使用最多的解决方案是加锁排队，伪代码如下：



说明：

加锁排队只是为了减轻数据库的压力，并没有提高系统吞吐量。假设在高并发下，缓存重建期间key是锁着的，这时候1000个请求999个都在阻塞。

注：在分布式环境下还需要解决分布式锁的问题。线程还会被阻塞，用户体验很差！因此，在真正的高并发场景下很少使用！

方案二：设置过期标记更新缓存

给每一个缓存数据增加相应的缓存标记，记录缓存是否失效，如果缓存标记失效则更新数据缓存。伪代码如下：



说明：

-1）缓存标记：记录缓存数据是否过期，若过期会触发通知另外的线程在后台去更新实际key的缓存

-2）缓存数据：它的过期时间比缓存标记的时间延长1倍（如标记缓存时间30分钟，数据缓存设置为60分钟）。这样，当缓存标记key过期后，实际缓存还能把旧数据返回给调用端，直到另外的线程在后台更新完成后，才会返回新缓存。

## 缓存预热

缓存预热就是系统上线后，将相关的缓存数据直接加载到缓存系统。这样可避免在用户请求时，先查询数据库后，再将数据缓存的问题！

即用户直接查询事先被预热的缓存数据！

解决思路：

-1）直接写个缓存刷新页面，上线时手动操作；

-2）数据量不大，可以在项目启动时自动进行加载；

-3）定时刷新缓存。