MYsql 数据表在300-500万条数据后性能开始急速下降

8G运行32G内存下mysql大概支持2000-5000的并发???

**Redis最好使用单数节点**

**缓存穿透**

# 1、安装与集群搭建

**Redis安装**

安装gcc，yum install gcc(解析redis源码)

解压redis，tar-zxvf redis-3.0.0-rcz.tar.gz

进入到redis-3.0.0/src目录下进行安装，安装完成后验证src目录下是否生成了redis-server和redis-cli,然后make install

1 下载地址：http://redis.io/download

2 安装步骤：

3 # 安装gcc

4 yum install gcc

5

6 # 把下载好的redis‐5.0.3.tar.gz放在/usr/local文件夹下，并解压

7 wget http://download.redis.io/releases/redis‐5.0.3.tar.gz

8 tar xzf redis‐5.0.3.tar.gz

9 cd redis‐5.0.3

10

11 # 进入到解压好的redis‐5.0.3目录下，进行编译与安装

12 make

13

14 # 启动并指定配置文件

15 src/redis‐server redis.conf（注意要使用后台启动，所以修改redis.conf里的daemonize改为y

es)

16

17 # 验证启动是否成功

18 ps ‐ef | grep redis

19cd

20 # 进入redis客户端

21 src/redis‐cli

22

23 # 退出客户端

24 quit

25

26 # 退出redis服务：

27 （1）pkill redis‐server

28 （2）kill 进程号

29 （3）src/redis‐cli shutdown

**Redis的复制原理**

**全量同步**  
Redis全量复制一般发生在Slave初始化阶段，这时Slave需要将Master上的所有数据都复制一份。具体步骤如下：   
-  从服务器连接主服务器，发送SYNC命令；   
-  主服务器接收到SYNC命名后，开始执行BGSAVE命令生成RDB文件并使用缓冲区记录此后执行的所有写命令；   
-  主服务器BGSAVE执行完后，向所有从服务器发送快照文件，并在发送期间继续记录被执行的写命令；   
-  从服务器收到快照文件后丢弃所有旧数据，载入收到的快照；   
-  主服务器快照发送完毕后开始向从服务器发送缓冲区中的写命令；   
-  从服务器完成对快照的载入，开始接收命令请求，并执行来自主服务器缓冲区的写命令

**增量同步**  
Redis增量复制是指Slave初始化后开始正常工作时主服务器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主服务器每执行一个写命令就会向从服务器发送相同的写命令，从服务器接收并执行收到的写命令。

**主从复制**

Slaveof 127.0.0.1 6379命令行指定为谁的从节点，也可以在配置文件中配置（不建议），指定为从服务器后会删除原有数据，复制主服务器所有数据

Info replication 查看服务器信息

Slaveof no one清除主从命令（执行后复制的主服务器信息还在）

主服务器挂掉，从服务器变为status up原地待命状态，可读不可写

Shut down 关闭服务器，命令指定的从服务器宕机重启后变为master，需要命令行重新配置

**薪火相传**

Master控制一台从服务器，从服务器控制一台从服务器，一次向下传递，所有服务器中只有一台主服务器，薪火相传实现了去中心化，但串联操作中间节点挂掉后面节点都不能使用，故实际生产环境中不会使用

**哨兵模式**

Sentinel.conf文件内配置，vim sentinel.conf编辑该文件

Sentinel monitor host6379（mastre名随意写）127.0.0.1 6379 1（从服务器获得最低的升级为主服务器的票数）

**集群**

搭建集群需要安装ruby脚本

Yum install ruby

Yum install rubygems

Gem installredis-version 3.0.0(安装redis和ruby的接口)

搭建集群

# ./redis-trip.rb create --replicas 1(主节点数/从节点数) ip（所有节点ip，192.168.0.61：8001 。。。）

前面几台服务器为主节点，后面为从节点，第一个主节点对应最开始的几台从节点，集群的数据写入根据轮训来确定master，拿数据通过重定向拿，不论数据在哪个节点上

Ps -ef|grep redis 看进程，各个redis服务器

集群中的master挂掉重启后变为该小集群的从节点

**集群的分片？**

Cluster nodes查看节点状态

**加入新的服务器**

/user/local/redis-3.0.0/src/redis-trib.rb add-node 192.168.0.61:8007(新节点) 192.168.0.61.8001（集群内任意节点）

节点加入后分配hash槽，找到集群内任意主节点

/user/local/redis-3.0.0/src/redis-trip.rb reshard 主节点ip -->新节点分配多少槽位-->输入接受槽位的服务器ID

每个主节点分配一个范围的槽位给新的主节点，故新的主节点有数个范围的槽位

**给新的主节点加入从服务器**

在从服务器客户端 cluster replicate ID（新主节点的ID）

**删除从节点在**cluster：#/user/local/redis-3.0.0/src/redis-trip.rb del-node 192.168.0.61:8008 ID(从节点ID)

**删除主节点**，首先需要把数据转移到其他主节点上（同加入槽位步骤大体相同）

/user/local/redis-3.0.0/src/redis-trip.rb reshard 192.168.0.61:8007

输入移动槽位数-->移动到哪（集群内任意主节点ID）——>数据源（移动数据的主节点ID）-->done执行

# 2、原理

## 2.1 redis底层

**Redis的单线程和高性能**

Redis 单线程为什么还能这么快？

因为它所有的数据都在内存中，所有的运算都是内存级别的运算，而且单线程避免了多线程的切换性能损耗问题。正因为 Redis 是单线程，所以要小心使用 Redis 指令，对于那些耗时的指令(比如keys)，一定要谨慎使用，一不小心就可能会导致 Redis 卡顿。

Redis 单线程如何处理那么多的并发客户端连接？

Redis的IO多路复用：redis利用epoll来实现IO多路复用，将连接信息和事件放到队列中，依次放到文件事件分派器，事件分派器将事件分发给事件处理器。

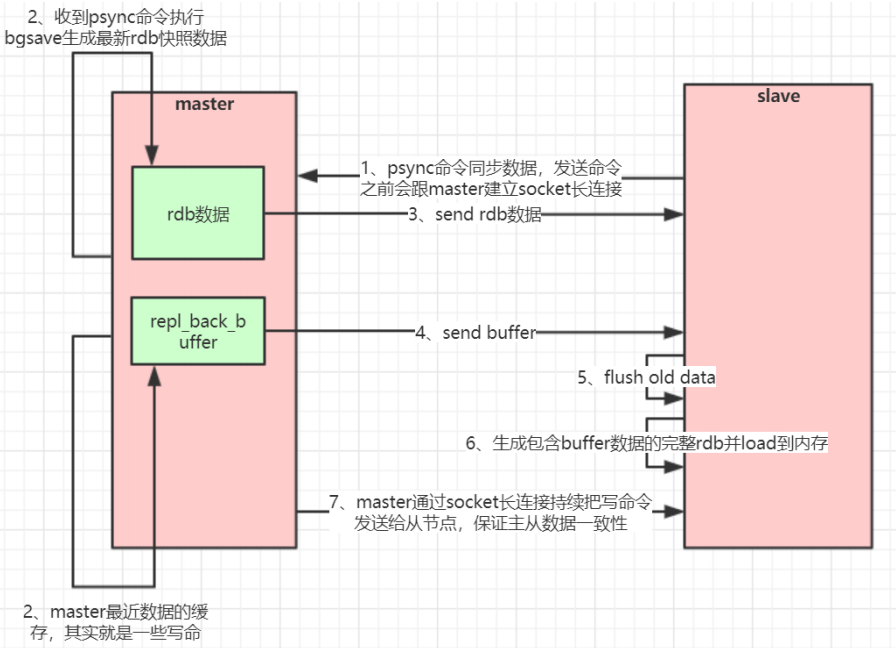
Nginx也是采用IO多路复用原理解决C10K问题

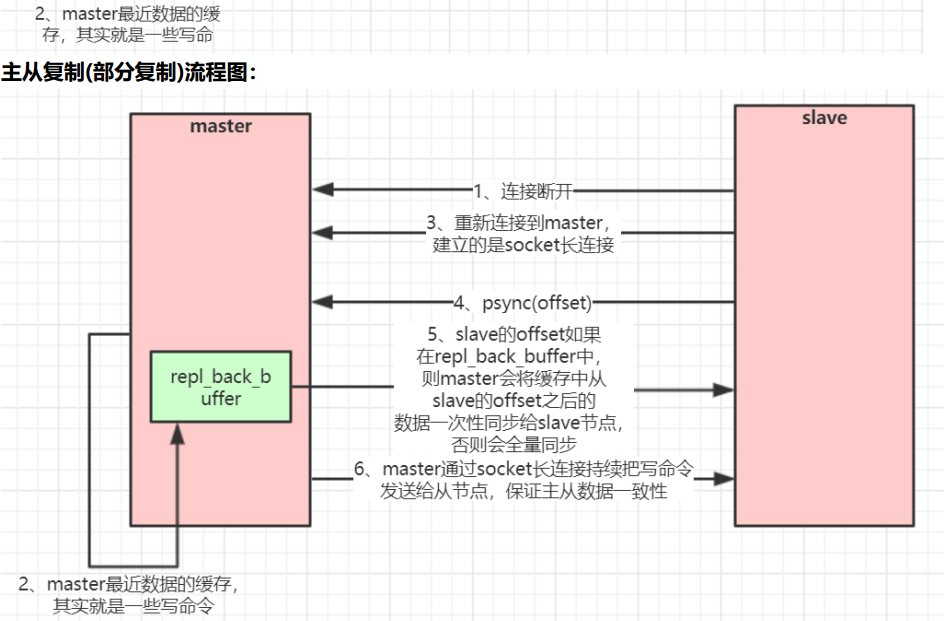
## 2.2redis主从数据同步

如果你为master配置了一个slave， 不管这个slave是否是第一次连接上Master， 它都会发送一个SYNC命令(redis2.8版本之前的命令)给master请求复制数据。master收到SYNC命令后， 会在后台进行数据持久化通过bgsave生成最新的rdb快照文件，持久化期间，master会继续接收客户端的请求，它会把这些可能修改数据集的请求缓存在内存中。 当持久化进行完毕以后， master会把这份rdb文件数据集发送给slave，slave会把接收到的数据进行持久化生成rdb，然后再加载到内存中。 然后，master再将之前缓存在内存中的命令发送给slave。当master与slave之间的连接由于某些原因而断开时， slave能够自动重连Master， 如果master收到了多个slave并发连接请求， 它只会进行一次持久化， 而不是一个连接一次， 然后再把这一份持久化的数据发送给多个并发连接的slave。

当master和slave断开重连后， 一般都会对整份数据进行复制。 但从redis2.8版本开始， master和slave断开重连后支持部分复制

**数据部分复制**从2.8版本开始，slave与master能够在网络连接断开重连后只进行部分数据复制。  
master会在其内存中创建一个复制数据用的缓存队列，缓存最近一段时间的数据，master和它所有的slave都维护了复制的数据下标offset和master的进程id， 因此，当网络连接断开后，slave会请求master继续进行未完成的复制，从所记录的数据下标开始。如果master进程id变化了，或者从节点数据下标offset太旧，已经不在master的缓存队列里了，那么将会进行一次全量数据的复制。  
从2.8版本开始，redis改用可以支持部分数据复制的命令PSYNC去master同步数据  
主从复制(全量复制)流程图：

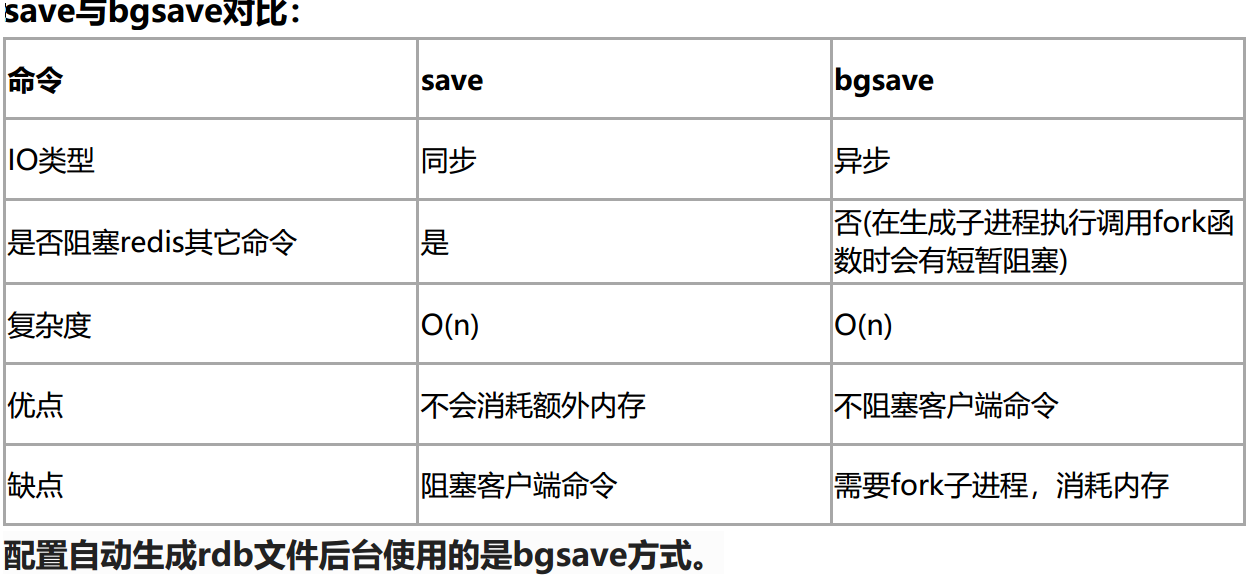




# 3、redis持久化

## 3.1RDB快照（snapshort）

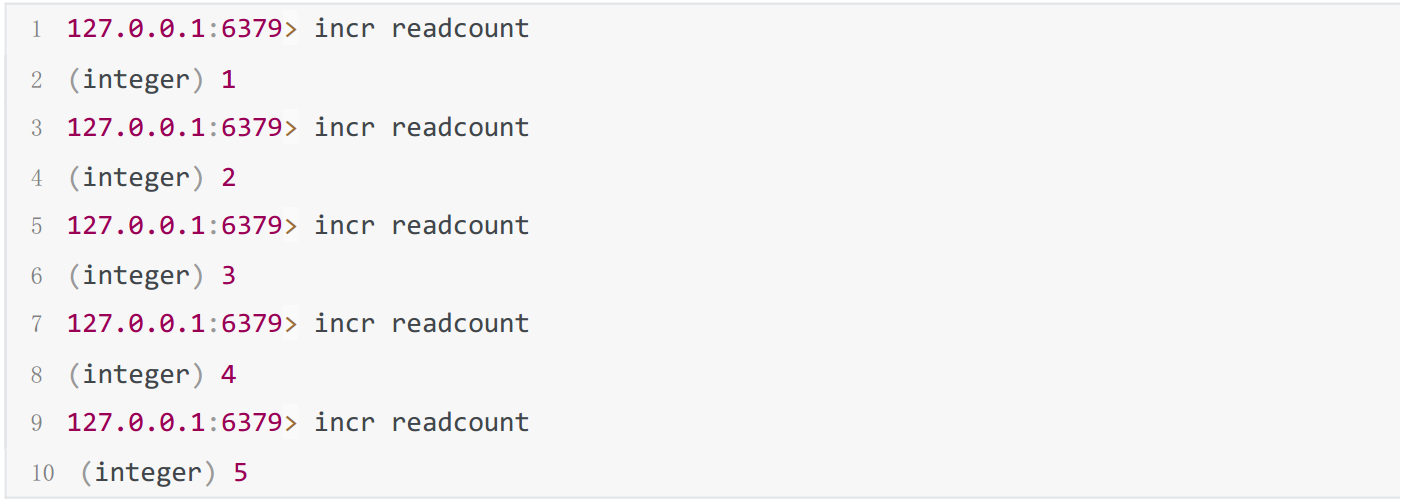
在默认情况下，Redis 将内存数据库快照保存在名字为 dump. rdb 的二进制文件中，**文件经过压缩，内容为二进制，不可读**。你可以对 Redis 进行设置，让它在“ N 秒内数据集至少有 M 个改动” 这一条件被满足时，自动保存一次数据集。  
比如说， 以下设置会让 Redis 在满足“ 60 秒内有至少有 1000 个键被改动” 这一条件时， 自动保存一次数据集：  
# save 60 1 000  
**关闭RDB只需要将所有的save保存策略注释掉即可**  
还可以手动执行命令生成RDB快照， 进入redis客户端执行命令save或bgsave可以生成dump.rdb文件，  
每次命令执行都会将所有redis内存快照到一个新的rdb文件里， 并覆盖原有rdb快照文件。  
save是同步命令， bgsave是异步命令， bgsave会从redis主进程fork（fork()是linux函数） 出一个子进程专门用来生成rdb快照文件



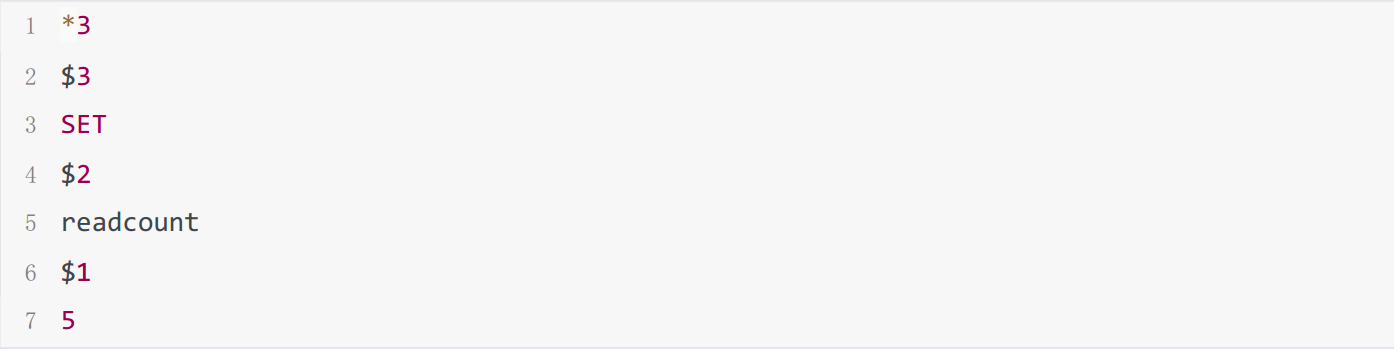
## 3.2AOF（append-only-file）

快照功能并不是非常耐久（durable）：如果 Redis 因为某些原因而造成故障停机，那么服务器将丢失  
最近写入、且仍未保存到快照中的那些数据。从 1 .1 版本开始， Redis 增加了一种完全耐久的持久化方  
式： AOF 持久化， 将修改的每一条指令记录进文件appendonly.aof中  
你可以通过修改配置文件来打开 AOF 功能：  
# appendonly yes  
从现在开始，每当 Redis 执行一个改变数据集的命令时（比如 SET） ，这个命令就会被追加到 AOF 文  
件的末尾。  
这样的话，当 Redis 重新启动时，程序就可以通过重新执行 AOF 文件中的命令来达到重建数据集的目  
的。你可以配置 Redis 多久才将数据 fsync 到磁盘一次。  
有三个选项：  
**appendfsync always**：每次有新命令追加到 AOF 文件时就执行一次 fsync ，非常慢，也非常安全。  
**appendfsync everysec**： 每秒 fsync一次，足够快（和使用 RDB 持久化差不多），并且在故障时只会丢失1秒钟的数据。（1秒内如果没有写操作则不会进行日志追加）  
**appendfsync no：** 从不 fsync 将数据交给操作系统来处理。 更快，也更不安全的选择。推荐（并且也是默认的措施为每秒 fsync 一次， 这种 fsync 策略可以兼顾速度和安全性。  
**AOF重写**

AOF文件里可能有太多没用指令， 所以AOF会定期根据内存的最新数据生成aof文件  
例如， 执行了如下几条命令：



重写后AOF文件里变成



如下两个配置可以控制AOF自动重写频率  
# auto-aof-rewrite-min-size 64mb //aof文件至少要达到64M才会自动重写， 文件太小恢复速度本  
来就很快， 重写的意义不大  
# auto-aof-rewrite-percentage 1 00 //aof文件自上一次重写后文件大小增长了1 00%则再次触发重  
写当然AOF还可以手动重写， 进入redis客户端执行命令bgrewriteaof重写AOF  
注意， AOF重写redis会fork出一个子进程去做， 不会对redis正常命令处理有太多影响

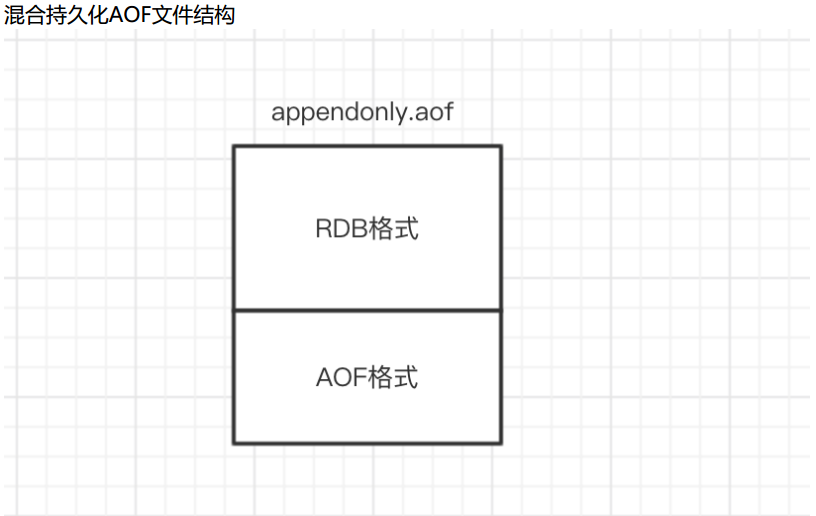


redis启动时如果既有rdb文件又有aof文件则优先选择aof文件恢复数据， 因为aof一般来说数据更全一点。

## 3.3 Redis 4.0 混合持久化

重启 Redis 时，我们很少使用 RDB来恢复内存状态， 因为会丢失大量数据。 我们通常使用 AOF 日 志重放，但是重放 AOF 日 志性能相对 RDB来说要慢很多，这样在 Redis 实例很大的情况下，启动需要花费很长的时间。Redis 4.0 为了解决这个问题，带来了一个新的持久化选项——混合持久化。  
通过如下配置可以开启混合持久化：  
# aof-use-rdb-preamble yes  
如果开启了混合持久化，AOF在重写时，不再是单纯将内存数据转换为RESP命令写入AOF文件，而是将重写这一刻之前的内存做RDB快照处理， 并且将RDB快照内容和增量的AOF修改内存数据的命令存在一起， 都写入新的AOF文件，新的文件一开始不叫appendonly.aof，等到重写完新的AOF文件才会进行改名， 原子的覆盖原有的AOF文件，完成新旧两个AOF文件的替换。  
于是在 Redis 重启的时候，可以先加载 RDB 的内容， 然后再重放增量 AOF 日 志就可以完全替代之前的

AOF 全量文件重放， 因此重启效率大幅得到提升。  
混合持久化AOF文件结构



# 4、redis使用优化

## 4.1redis缓存穿透、失效、雪崩

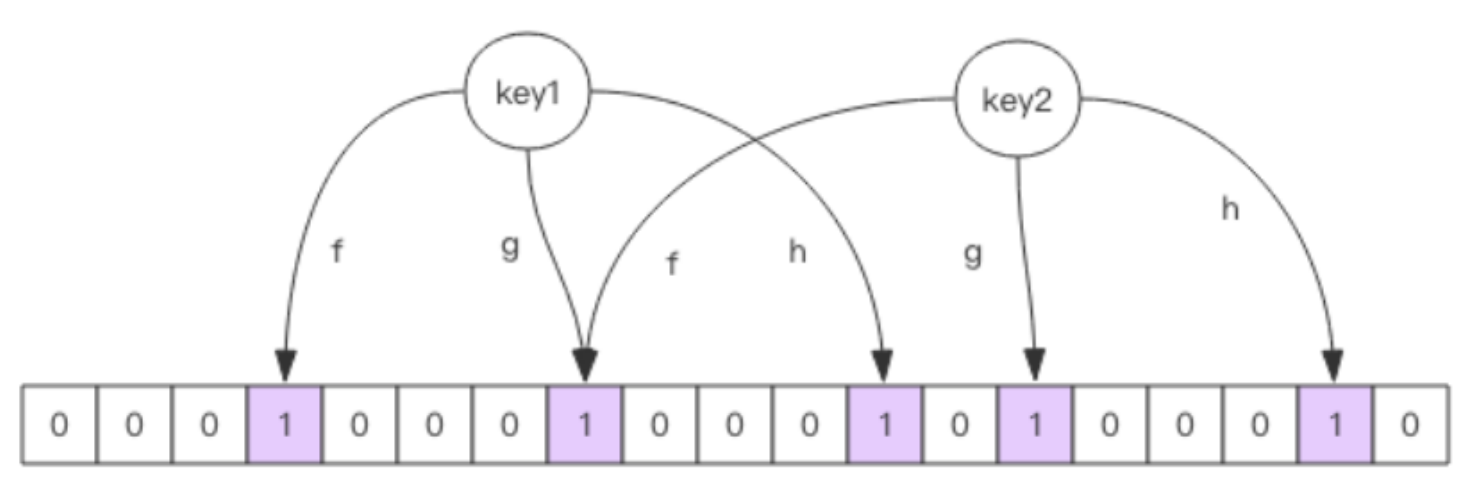
### 4.1.1 缓存穿透

**缓存穿透**是指查询一个根本不存在的数据，缓存层和存储层都不会命中，通常出于容错的考虑，如果从存储层查不到数据则不写入缓存层。  
缓存穿透将导致不存在的数据每次请求都要到存储层去查询， 失去了缓存保护后端存储的意义。  
造成缓存穿透的基本原因有两个：  
第一、自身业务代码或者数据出现问题。  
第二、一些恶意攻击、爬虫等造成大量空命中。  
缓存穿透问题解决方案：  
**1、 缓存空对象**





2、 布隆过滤器  
对于恶意攻击， 向服务器请求大量不存在的数据造成的缓存穿透， 还可以用布隆过滤器先做一次过滤， 对于不存在的数据布隆过滤器一般都能够过滤掉， 不让请求再往后端发送。 当布隆过滤器说某个值存在时， 这个值可能不存在；当它说不存在时，那就肯定不存在。



布隆过滤器就是一个大型的位数组和几个不一样的无偏 hash 函数。 所谓无偏就是能够把元素的 hash 值算得比较均匀。  
向布隆过滤器中添加 key 时， 会使用多个 hash 函数对 key 进行 hash 算得一个整数索引值然后对位数组长度进行取模运算得到一个位置，每个 hash 函数都会算得一个不同的位置。再把位数组的这几个位置都置为 1 就完成了 add 操作。  
向布隆过滤器询问 key 是否存在时，跟 add 一样，也会把 hash 的几个位置都算出来，看看位数组中这几个位置是否都为 1 ， 只要有一个位为 0， 那么说明布隆过滤器中这个key 不存在。 如果都是 1 ， 这并不能说明这个key 就一定存在， 只是极有可能存在， 因为这些位被置为 1 可能是因为其它的 key 存在所致。 如果这个位数组比较稀疏， 这个概率就会很大，如果这个位数组比较拥挤， 这个概率就会降低。  
这种方法适用于数据命中不高、数据相对固定、实时性低（通常是数据集较大）的应用场景，代码维护较为复杂，但是缓存空间占用很少。  
可以用guvua包自带的布隆过滤器，引入依赖：







### 4.1.2 缓存失效

由于大批量缓存在同一时间失效可能导致大量请求同时穿透缓存直达数据库， 可能会造成数据库瞬间压力过大甚至挂掉， 对于这种情况我们在批量增加缓存时最好将这一批数据的缓存过期时间设置为一个时间段内的不同时间。  
示例伪代码：





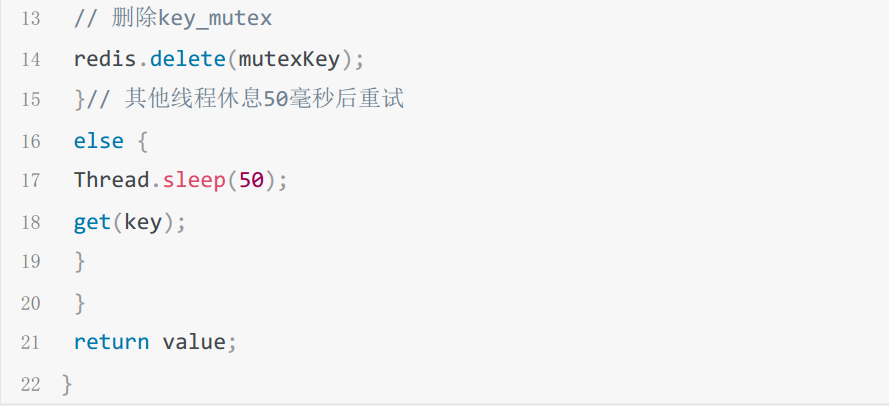
### 4.1.3缓存雪崩

缓存雪崩指的是缓存层支撑不住或宕掉后，流量会像奔逃的野牛一样，打向后端存储层。  
由于缓存层承载着大量请求，有效地保护了存储层，但是如果缓存层由于某些原因不能提供服务(比如超大并发过来，缓存层支撑不住，或者由于缓存设计不好，类似大量请求访问bigkey，导致缓存能支撑的并发急剧下降)，于是大量请求都会达到存储层，存储层的调用量会暴增，造成存储层也会级联宕机的情况。  
预防和解决缓存雪崩问题，可以从以下三个方面进行着手。  
1）保证缓存层服务高可用性， 比如使用Redis Sentinel或Redis Cluster。  
2）依赖隔离组件为后端限流并降级。 比如使用Hystrix限流降级组件。  
3）提前演练。 在项目上线前，演练缓存层宕掉后，应用以及后端的负载情况以及可能出现的问题， 在此基础上做一些预案设定。

**热点缓存key重建优化**  
开发人员使用“缓存+过期时间”的策略既可以加速数据读写，又保证数据的定期更新，这种模式基本能够满足绝大部分需求。但是有两个问题如果同时出现，可能就会对应用造成致命的危害：  
当前key是一个热点key（例如一个热门的娱乐新闻），并发量非常大。  
重建缓存不能在短时间完成，可能是一个复杂计算，例如复杂的SQL、多次IO、多个依赖等。  
在缓存失效的瞬间，有大量线程来重建缓存，造成后端负载加大，甚至可能会让应用崩溃。

要解决这个问题主要就是要避免大量线程同时重建缓存。  
我们可以利用互斥锁来解决， 此方法只允许一个线程重建缓存， 其他线程等待重建缓存的线程执行完， 重新从缓存获取数据即可。  
示例伪代码：





## 4.2 开发规范与性能优化

### 4.2.1开发规范

一、 键值设计  
**1 . key名设计**(1 )【建议】:可读性和可管理性  
以业务名(或数据库名)为前缀(防止key冲突)， 用冒号分隔， 比如业务名:表名:id



(2)【建议】 ： 简洁性  
保证语义的前提下， 控制key的长度， 当key较多时， 内存占用也不容忽视， 例如：



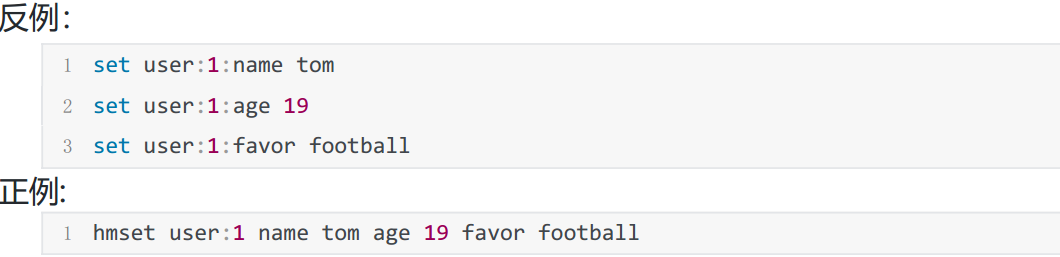
(3)【强制】 ： 不要包含特殊字符  
反例： 包含空格、 换行、 单双引号以及其他转义字符

2. value设计  
(1)【强制】：拒绝bigkey(防止网卡流量、慢查询)  
在Redis中，一个字符串最大512MB，一个二级数据结构（例如hash、 list、 set、 zset） 可以存  
储大约40亿个(2^32-1 )个元素，但实际中如果下面两种情况， 我就会认为它是bigkey。  
1、字符串类型：它的big体现在单个value值很大，一般认为超过10KB就是bigkey。  
2、非字符串类型：哈希、列表、集合、有序集合，它们的big体现在元素个数太多。  
一般来说，string类型控制在10KB以内，hash、list、set、zset元素个数不要超过5000。  
反例：一个包含200万个元素的list。  
非字符串的bigkey，不要使用del删除， 使用hscan、 sscan、 zscan方式渐进式删除， 同时要注  
意防止bigkey过期时间自动删除问题(例如一个200万的zset设置1 小时过期，会触发del操作，造成阻塞，可以将大集合拆分成小集合，错开过期）  
bigkey的危害：  
1、导致redis阻塞  
2、网络拥塞  
bigkey也就意味着每次获取要产生的网络流量较大， 假设一个bigkey为1 MB， 客户端每秒访问  
量为1000， 那么每秒产生1000MB的流量， 对于普通的千兆网卡(按照字节算是128MB/s)的服务  
器来说简直是灭顶之灾，而且一般服务器会采用单机多实例的方式来部署，也就是说一个bigkey  
可能会对其他实例也造成影响， 其后果不堪设想。  
3. 过期删除  
有个bigkey， 它安分守己（只执行简单的命令， 例如hget、 lpop、 zscore等），但它设置了过  
期时间，当它过期后，会被删除，如果没有使用Redis 4.0的过期异步删除(lazyfree-lazyexpire yes)， 就会存在阻塞Redis的可能性。  
bigkey的产生：  
一般来说，bigkey的产生都是由于程序设计不当，或者对于数据规模预料不清楚造成的，来看几个例子：  
(1) 社交类： 粉丝列表， 如果某些明星或者大v不精心设计下， 必是bigkey。  
(2) 统计类： 例如按天存储某项功能或者网站的用户集合， 除非没几个人用， 否则必是bigkey。  
(3) 缓存类： 将数据从数据库load出来序列化放到Redis里， 这个方式非常常用， 但有两个地方需  
要注意， 第一， 是不是有必要把所有字段都缓存； 第二， 有没有相关关联的数据， 有的同学为了  
图方便把相关数据都存一个key下， 产生bigkey。  
如何优化bigkey  
1、拆  
big list： list1 、 list2、 ...listN  
big hash： 可以讲数据分段存储， 比如一个大的key，假设存了1 百万的用户数据，可以拆分成

200个key，每个key下面存放5000个用户数据（每个key设计数字编号用于定位数据存储于哪个key上）

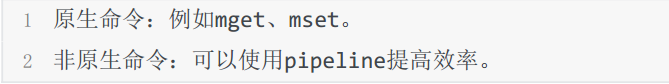
2. 如果bigkey不可避免， 也要思考一下要不要每次把所有元素都取出来(例如有时候仅仅需要  
hmget，而不是hgetall)， 删除也是一样， 尽量使用优雅的方式来处理。

(2)【推荐】 ： 选择适合的数据类型。  
例如： 实体类型(要合理控制和使用数据结构， 但也要注意节省内存和性能之间的平衡)

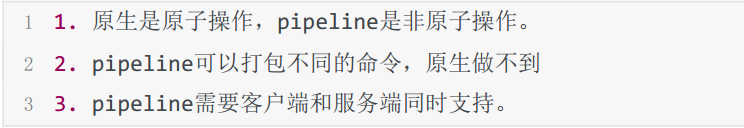


### 4.2.2 命令使用

1.【推荐】 O(N)命令关注N的数量  
例如hgetall、 lrange、 smembers、 zrange、 sinter等并非不能使用， 但是需要明确N的值。 有  
遍历的需求可以使用hscan、 sscan、 zscan代替。  
2.【推荐】：禁用命令  
禁止线上使用keys、 flushall、 flushdb等， 通过redis的rename机制禁掉命令， 或者使用scan的  
方式渐进式处理。  
3.【推荐】合理使用select  
redis的多数据库较弱，使用数字进行区分，很多客户端支持较差，同时多业务用多数据库实际还  
是单线程处理，会有干扰。  
4.【推荐】 使用批量操作提高效率



但要注意控制一次批量操作的元素个数(例如500以内， 实际也和元素字节数有关)。  
注意两者不同：

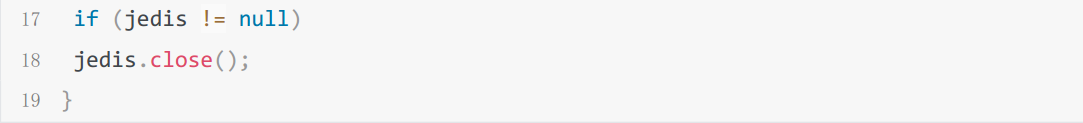


5.【建议】 Redis事务功能较弱， 不建议过多使用， 可以用lua替代

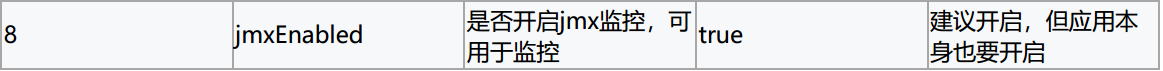
### 4.2.3客户端的使用

1.【推荐】  
避免多个应用使用一个Redis实例  
正例：不相干的业务拆分， 公共数据做服务化。  
2.【推荐】  
使用带有连接池的数据库， 可以有效控制连接， 同时提高效率， 标准使用方式：









优化建议：  
**1） maxTotal：** 最大连接数， 早期的版本叫maxActive  
实际上这个是一个很难回答的问题， 考虑的因素比较多：  
业务希望Redis并发量  
客户端执行命令时间  
Redis资源： 例如 nodes(例如应用个数) \* maxTotal 是不能超过redis的最大连接数  
maxclients。  
资源开销： 例如虽然希望控制空闲连接(连接池此刻可马上使用的连接)，但是不希望因  
为连接池的频繁释放创建连接造成不必靠开销。  
以一个例子说明，假设:  
一次命令时间（borrow|return resource + Jedis执行命令(含网络) ） 的平均耗时约为  
1 ms， 一个连接的QPS大约是1000  
业务期望的QPS是50000  
那么理论上需要的资源池大小是50000 / 1000 = 50个。 但事实上这是个理论值， 还要考虑到要  
比理论值预留一些资源， 通常来讲maxTotal可以比理论值大一些。  
但这个值不是越大越好，一方面连接太多占用客户端和服务端资源， 另一方面对于Redis这种高  
QPS的服务器，一个大命令的阻塞即使设置再大资源池仍然会无济于事。  
2） maxIdle和minIdle  
maxIdle实际上才是业务需要的最大连接数， maxTotal是为了给出余量， 所以maxIdle不要设置  
过小， 否则会有new Jedis(新连接)开销。  
连接池的最佳性能是maxTotal = maxIdle， 这样就避免连接池伸缩带来的性能干扰。 但是如果  
并发量不大或者maxTotal设置过高， 会导致不必要的连接资源浪费。 一般推荐maxIdle可以设置  
为按上面的业务期望QPS计算出来的理论连接数， maxTotal可以再放大一倍。  
minIdle（最小空闲连接数），与其说是最小空闲连接数，不如说是"至少需要保持的空闲连接  
数"， 在使用连接的过程中， 如果连接数超过了minIdle， 那么继续建立连接， 如果超过了  
maxIdle，当超过的连接执行完业务后会慢慢被移出连接池释放掉。

如果系统启动完马上就会有很多的请求过来，那么可以给redis连接池做预热，比如快速的创建一  
些redis连接，执行简单命令，类似ping()，快速的将连接池里的空闲连接提升到minIdle的数  
量。  
连接池预热示例代码：





总之， 要根据实际系统的QPS和调用redis客户端的规模整体评估每个节点所使用的连接池大小。  
3.【建议】  
高并发下建议客户端添加熔断功能(例如netflix hystrix)  
4.【推荐】  
设置合理的密码，如有必要可以使用SSL加密访问  
5.【建议】  
**Redis对于过期键有三种清除策略：  
被动删除**：当读/写一个已经过期的key时，会触发惰性删除策略，直接删除掉这个过期key  
**主动删除**： 由于惰性删除策略无法保证冷数据被及时删掉，所以Redis会定期主动淘汰一批已过期的key  
当前已用内存超过maxmemory限定时， 触发主动清理策略，当REDIS运行在主从模式时， 只有主结点才会执行被动和主动这两种过期删除策略，然后把删除操作” del key” 同步到从结点。  
**第三种策略的情况如下**：当前已用内存超过maxmemory限定时，会触发主动清理策略，根据自身业务类型， 选好maxmemory-policy(最大内存淘汰策略)，设置好过期时间。 如果不设置最大内存，当 Redis 内存超出物理内存限制时， 内存的数据会开始和磁盘产生频繁的交换 (swap)，会让 Redis 的性能急剧下降。  
默认策略是volatile-lru， 即超过最大内存后，在过期键中使用lru算法进行key的剔除， 保证不过  
期数据不被删除，但是可能会出现OOM问题。  
其他策略如下：  
allkeys-lru：根据LRU算法删除键，不管数据有没有设置超时属性，直到腾出足够空间为止。  
allkeys-random：随机删除所有键，直到腾出足够空间为止。  
volatile-random: 随机删除过期键，直到腾出足够空间为止。  
volatile-ttl： 根据键值对象的ttl属性， 删除最近将要过期数据。 如果没有，回退到noeviction策略。  
noeviction： 不会剔除任何数据，拒绝所有写入操作并返回客户端错误信息"(error)  
OOM command not allowed when used memory"， 此时Redis只响应读操作。