

redis五大数据类型



redis键（key）

常用命令

命令	注释
keys *	获取所有的key
select 0（库角标）	选择第几个库
move key 0（库角标）	将当前的数据库key移动到某个数据库,目标库有，则不能移动
flush 0（库角标）	清除指定库
randomkey	获取随机key
type key	key的类型
set key value	设置key和value
get key	获取key的value
mset key1 value1 key2 value2	批量设置key、value
mget key1 key2 key3	批量获取value
del key	删除key
exists key	判断是否存在key
expire key second（秒）	给key设置过期时间
pexpire key millisecond（毫秒）	给key设置过期时间
persist key	删除过期时间，持久保存key

String类型

简介

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redis中字符串value最多可以是 512M

存储结构类似： key:value

常用命令

命令	注释
set key value	设置值
get key	获取值
getrange key start end	获取指定范围的value
getset key value	设置新value，并返回旧value
getbit key offset	获取字符串中某个位置的字符
mget key1 key2	获取多个value
setex key second（秒） value	设置key、value，同时设置过期时间
setnx key value	key不存在时设置key
setrange key offset value	用新value替换老value部分字符，从offset开始替换
strlen key	获取value长度
mset key1 value1 key2 value2	批量设置key、value
msetnx key1 value1 key2 value2	批量设置，当且仅当所有要设置的key都不存在时
psetex key milliseconds（毫秒） value	设置过期时间，单位毫秒
incr key	如果value是数字，使用这个语法使数字自增1
incrby key increment	给value增加指定的值increment
decr key	给value减去1
decrby key decrement	给value减去指定的值decrement
append key value	将value追加到key原来的value尾部

List类型

简介

它是一个字符串链表， left、right都可以插入添加；

如果键不存在，创建新的链表； 如果键已存在，新增内容； 如果值全移除，对应的键也就消失了。

链表的操作无论是头和尾效率都极高，但假如是对中间元素进行操作，效率就很惨淡了。

key 是列表的名称， value 是列表。

存储结构类似： key:[value1, value2, value3, value4]

常用命令

命令	注释
blpop key timeout	在timeout时间内，获取并移除列表的第一个元素
brpop key1 timeout	在timeout时间内，获取并移除列表的最后一个元素
brpoplpush source destination timeout	在timeout时间内，从source列表中取出一个值，放到destination列表中
lindex key index	获取列表index位置的值
linsert key BEFORE	AFTER value1 value2
llen key	返回列表的长度
lpop key	获取并移除列表的第一个元素
lpush key value value2	将一个或多个value插入到列表的头部
lpushx key value	当key已经存在的时候，向列表的头部插入value
lrange key start end	获取列表部分数据，从start到end范围
lrem key count value	count>0，从列表的头部开始算起，移除count个value相同的数据；count<0，从列表的尾部开始算起，移除count绝对值个value相同的数据；count=0，全部移除value相同的数据
lset key index value	在列表index位置设置value
ltrim key start end	保留start到end内的数据，其余的全部删除

rpop key	获取并移除列表最后一个元素
rpoplpush source destination	移除source列表最后一个元素，并把该元素添加到destination列表的头部
R PUSH key value1 value2	将一个或多个value添加到列表的尾部
rpushx key value	为已经存在的列表添加值

Hash类型

简介

hash 是一个 string 类型的 field（字段） 和 value（值） 的映射表，hash 特别适合用于存储对象。

存储结构类似： `key:{field1:value1, field2:value2, field3:value3}`

如： `HMSET keyName name "redis tutorial" description "redis basic commands for caching"`

常用命令

命令	注释
hdel key field1 field2	删除key中一个或多个field及value
hexists key field	查看哈希表key中， 指定的field字段是否存在
hget key field	在key中查找field字段的value值
hgetall key	获取在哈希表中指定 key 的所有字段和值
hincrby key field increment	为哈希表 key 中的指定字段的整数值加上增量 increment
hincrbyfloat key field increment	为哈希表 key 中的指定字段的浮点数值加上增量 increment 。
hkeys key	获取所有哈希表中的字段
hlen key	获取哈希表中字段的数量
hmget key field1 field2	获取所有指定字段的值
hmset key field1 value1 field2 value2	同时将多个 field-value对设置到哈希表 key 中
hset key field value	将哈希表 key 中的字段 field 的值设为 value 。
hsetnx key field value	只有在字段 field 不存在时， 设置哈希表字段的值。

hvals key	获取哈希表中所有值。
hscan key cursor [MATCH pattern] [COUNT count]	迭代哈希表中的键值对。

set类型

简介

Redis 的 Set 是 String 类型的无序集合。集合成员是唯一的，这就意味着集合中不能出现重复的数据。

Redis 中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)。

存储结构类似： key:("value1", "value2", "value3")

常用命令

命令	注释
sadd key value1 value2	向集合添加一个或多个成员
scard key	获取集合的成员数大小
sdiff key1 key2 key3	返回第一个集合与其他集合之间的差异。
sdiffstore destination key1 key2	返回给定所有集合的差集并存储在 destination 新key 中
sinter key1 key2	返回给定所有集合的交集
sinterstore destination key1 key2	返回给定所有集合的交集并存储在 destination 新key 中
sismember key value	判断key的集合中是否存在value
smembers key	返回key集合中所有的value
smove source destination value	将 value 元素从 source的key 集合移动到 destination 的key 集合中
spop key	随机获取并移除key中的一个value
srandmember key count	随机返回集合中count个value
srem key value1 value2	移除集合中一个或多个value
sunion key1 key2	返回所有给定集合的并集
sunionstore destination key1 key2	所有给定集合的并集存储在 destination key集合中
SSCAN key cursor [MATCH pattern] [COUNT count]	迭代集合中的元素

Zset(sorted set)类型

简介

Redis 有序集合和集合一样也是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。

有序集合的成员是唯一的,但分数(score)却可以重复。

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)。 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)。

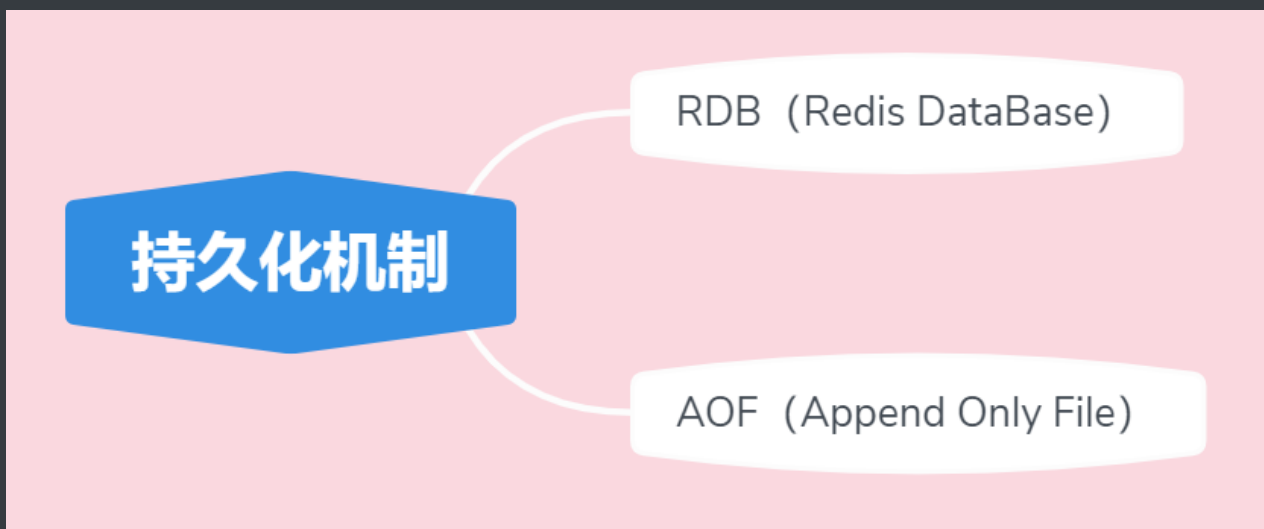
存储结构类似： `key:{score1:value1, score2:value2, score3:value3, }`

常用命令

命令	注释
<code>zadd key score1 value1 score2 value2</code>	向有序集合添加一个或多个成员，或者更新已存在成员的分数
<code>zcard key</code>	获取有序集合的成员数
<code>zcount key min max</code>	计算在有序集合中指定区间分数的成员数
<code>zincrby key increment value</code>	有序集合中对指定成员的分数加上增量 increment
<code>zinterstore destination numkeys key [key ...]</code>	计算给定的一个或多个有序集的交集并将结果集存储在新的有序集合 key 中
<code>zlexcount key min max</code>	在有序集合中计算指定字典区间内成员数量
<code>zrange key start stop [WITHSCORES]</code>	通过索引区间返回有序集合指定区间内的成员
<code>zrangebylex key min max [LIMIT offset count]</code>	通过字典区间返回有序集合的成员
<code>zrangebyscore key min max [WITHSCORES] [LIMIT]</code>	通过分数返回有序集合指定区间内的成员
<code>zrank key value</code>	返回有序集合中指定成员的索引
<code>zrem key value [value ...]</code>	移除有序集合中的一个或多个成员
<code>zremrangebylex key min max</code>	移除有序集合中给定的字典区间的所有成员
<code>zremrangebyrank key start end</code>	移除有序集合中给定的排名区间的所有成员
<code>zremrangebyscore key min max</code>	移除有序集合中给定的分数区间的所有成员

zrevrange key start stop [WITHSCORES]	返回有序集中指定区间内的成员，通过索引，分数从高到低
zrevrangebyscore key max min [WITHSCORES]	返回有序集中指定分数区间内的成员，分数从高到低排序
zrevrank key value	返回有序集合中指定成员的排名，有序集成员按分数值递减(从大到小)排序
zscore key value	返回有序集中，成员的分数值
zunionstore destination numkeys key [key ...]	计算给定的一个或多个有序集的并集，并存储在新的 key 中
zscan key cursor [MATCH pattern] [COUNT count]	迭代有序集合中的元素（包括元素成员和元素分值）

Redis持久化两种方式



RDB持久化

RDB是什么

RDB持久化的机制是在 一段时间内 达到 某修改次数 ，就把内存数据快照Snapshot 持久化 到硬盘上，比如：配置1分钟内修改100次，达到这个条件时，就会进行持久化操作。RDB文件格式是 `dump.rdb`

如何配置

Save the DB on disk:

```
save <seconds> <changes>
```

Will save the DB if both the given number of seconds and the given number of write operations against the DB occurred.

In the example below the behaviour will be to save:

after 900 sec (15 min) if at least 1 key changed

after 300 sec (5 min) if at least 10 keys changed

after 60 sec if at least 10000 keys changed

即：在redis.conf文件里配置，截图上的save <seconds> <changes>

如：save 1 100（一分钟内修改100次）

如何停止：在redis.conf文件里配置 save ""，或者通过命令 config set save ""

触发RDB几种方式

- 自动触发

就是上面说的redis.conf里的save配置

- 手动触发

执行 save 命令：save时只管保存，其它不管，全部阻塞

执行 bgsave 命令：Redis会在后台异步进行快照操作，快照同时还可以响应客户端请求。可以通过lastsave 命令获取最后一次成功执行快照的时间

执行 flushall 命令，也会产生dump.rdb文件，但里面是空的，无意义

持久化原理-fork

Redis会单独创建（fork）一个子进程来进行持久化，会先将数据写入到一个临时文件中，待持久化过程都结束了，再用这个临时文件替换上次持久化好的文件。

整个过程中，主进程是不进行任何IO操作的，这就确保了极高的性能，如果需要进行大规模数据的恢复，且对于数据恢复的完整性不是非常敏感。

那RDB方式要比AOF方式更加的高效。RDB的缺点是最后一次持久化后的数据可能丢失。

Fork的作用是复制一个与当前进程一样的进程。新进程的所有数据（变量、环境变量、程序计数器等），数值都和原进程一致，但是是一个全新的进程，并作为原进程的子进程。

将持久化文件重新加载到内存中

Redis是 基于内存 的，所以要将 硬盘 上的数据重新 加载到内存 中提供服务。

- 将备份文件 (dump.rdb) 移动到 redis 安装目录并启动服务即可，redis就会自动加载文件数据至内存了。Redis 服务器在载入 RDB 文件期间，会一直处于阻塞状态，直到载入工作完成为止。
- 获取 redis 的安装目录可以使用 config get dir 命令

RDB优势与劣势

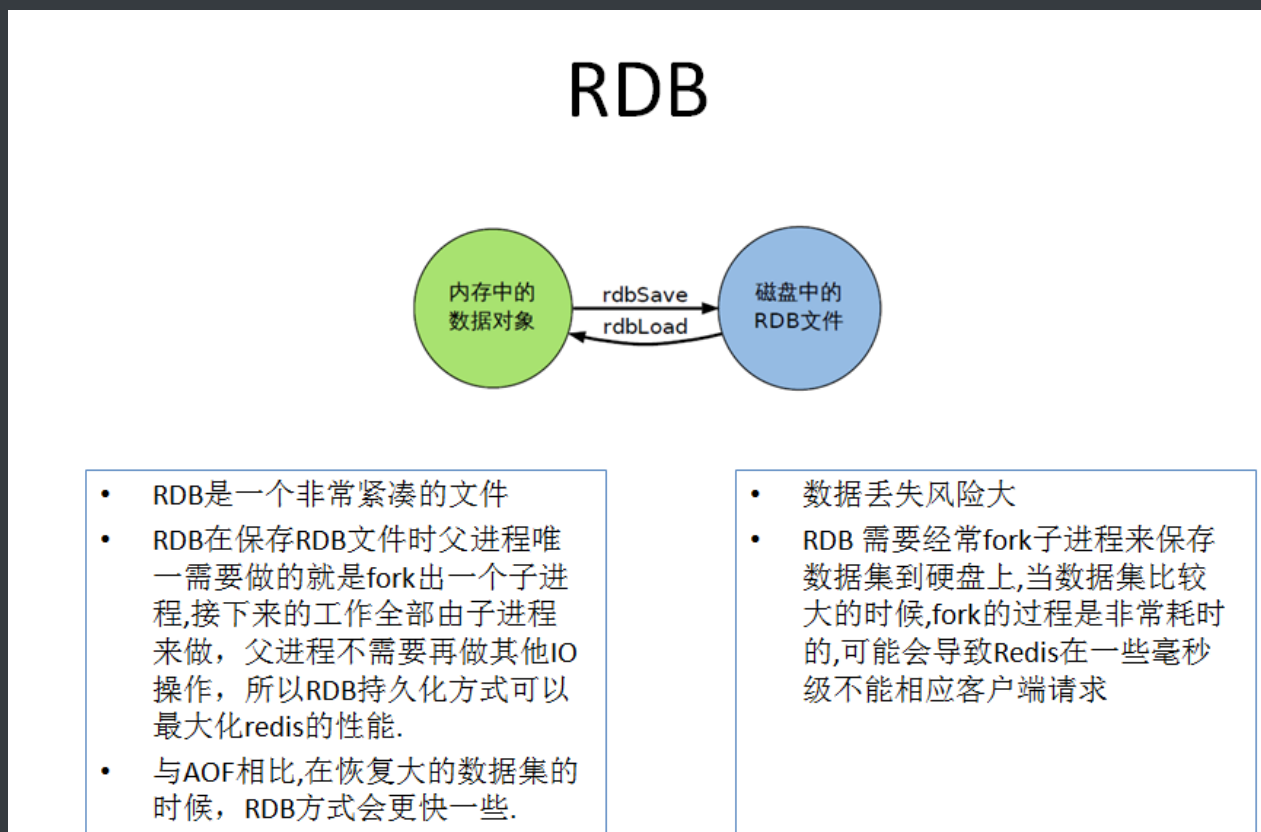
优势

- 适合大规模的数据恢复
- 对数据完整性和一致性要求不高

劣势

- 在一定间隔时间做一次备份，所以如果redis意外down掉的话，就会丢失最后一次快照后的所有修改。
- Fork的时候，内存中的数据被 克隆 了一份， 大致2倍 的膨胀性能需要考虑

RDB总结



AOF持久化

AOF是什么

Aof保存的是 appendonly.aof 文件，是将Redis所有的 写命令（增删改）记录到这个日志文件中，读命令 不记录。

只允许在文件末尾追加内容，不允许改写文件。

Redis启动的时候就会读取该文件，简而言之，就是将文件中的命令 重新执行 一遍，完成数据恢复到内存的工作。

如何配置

```
##### APPEND ONLY MODE #####

# By default Redis asynchronously dumps the dataset on disk. This mode is
# good enough in many applications, but an issue with the Redis process or
# a power outage may result into a few minutes of writes lost (depending on
# the configured save points).
#
# The Append Only File is an alternative persistence mode that provides
# much better durability. For instance using the default data fsync policy
# (see later in the config file) Redis can lose just one second of writes in a
# dramatic event like a server power outage, or a single write if something
# wrong with the Redis process itself happens, but the operating system is
# still running correctly.
#
# AOF and RDB persistence can be enabled at the same time without problems.
# If the AOF is enabled on startup Redis will load the AOF, that is the file
# with the better durability guarantees.
#
# Please check http://redis.io/topics/persistence for more information.

appendonly no

# The name of the append only file (default: "appendonly.aof")

appendfilename "appendonly.aof"

# The fsync() call tells the Operating System to actually write data on disk
# instead of waiting for more data in the output buffer. Some OS will really flush
# data on disk, some other OS will just try to do it ASAP.
```

(默认是no，yes就打开aof持久化)

即：在redis.conf文件里配置，截图上的改成 appendonly yes 。

持久化策略

通过Appendfsync配置

- Appendfsync Always

每次发生 数据 变更 会被立即 记录 到磁盘， 性能较差 但 数据完整性比较好

- Appendfsync Everysec

出厂默认推荐，异步操作， 每秒 记录， 如果 一秒 内 宕机 ， 有数据 丢失

AOF启动/恢复/修复

同样我们需要将AOF文件 加载 到 内存 中之后才能 使用 ， 如果 AOF 文件被 破坏 了，我们该如何 修复 呢？

- 正常恢复到内存中

将有数据的aof文件复制一份保存到对应目录，目录路径可以通过 `config get dir` 命令获取，重新启动Redis就可以了

- 异常恢复文件到内存中

备份异常AOF文件，使用命令对文件进行修复：`redis-check-aof --fix 文件名`，然后重新启动Redis就可以了

Rewrite重写AOF文件

什么是Rewrite

AOF采用 文件追加 方式，文件会 越来越大 为避免出现此种情况，新增了重写机制。

当AOF 文件的大小 超过所设定的 阈值 时，Redis就会 启动 AOF文件的内容 压缩 ， 只保留可以恢复数据的 最小指令集 .可以使用命令 `bgrewriteaof` 进行重写文件

Rewrite原理

AOF文件持续增长而过大时，会fork出一条 新进程 来将文件重写(也是先写临时文件最后再rename)。

遍历 新进程 的内存中数据，每条记录有一条的Set语句。重写aof文件的操作，并没有读取旧的aof文件。

而是将整个内存中的数据库内容用命令的方式重写了一个 新的aof 文件，这点和快照有点类似

触发重写机制

Redis会记录上次重写时的AOF大小，默认配置是当AOF文件大小是上次rewrite后大小的一倍，且文件大于64M时触发

AOF优势/劣势

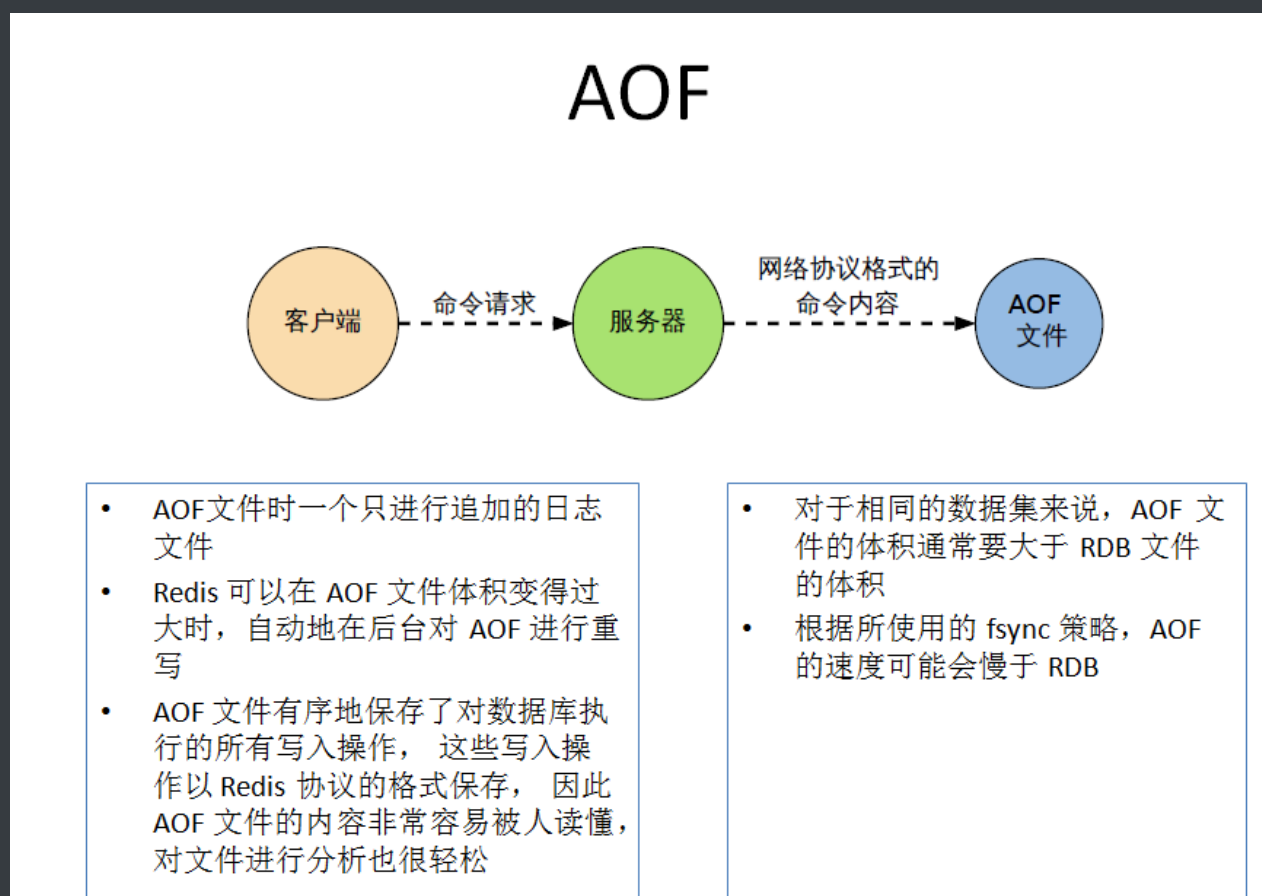
优势

- 每次修改同步：appendfsync always同步持久化，每次发生数据变更会被立即记录到磁盘，性能较差但数据完整性比较好
- 每秒同步：appendfsync everysec异步操作，每秒记录，如果一秒内宕机，仅一秒内的数据丢失

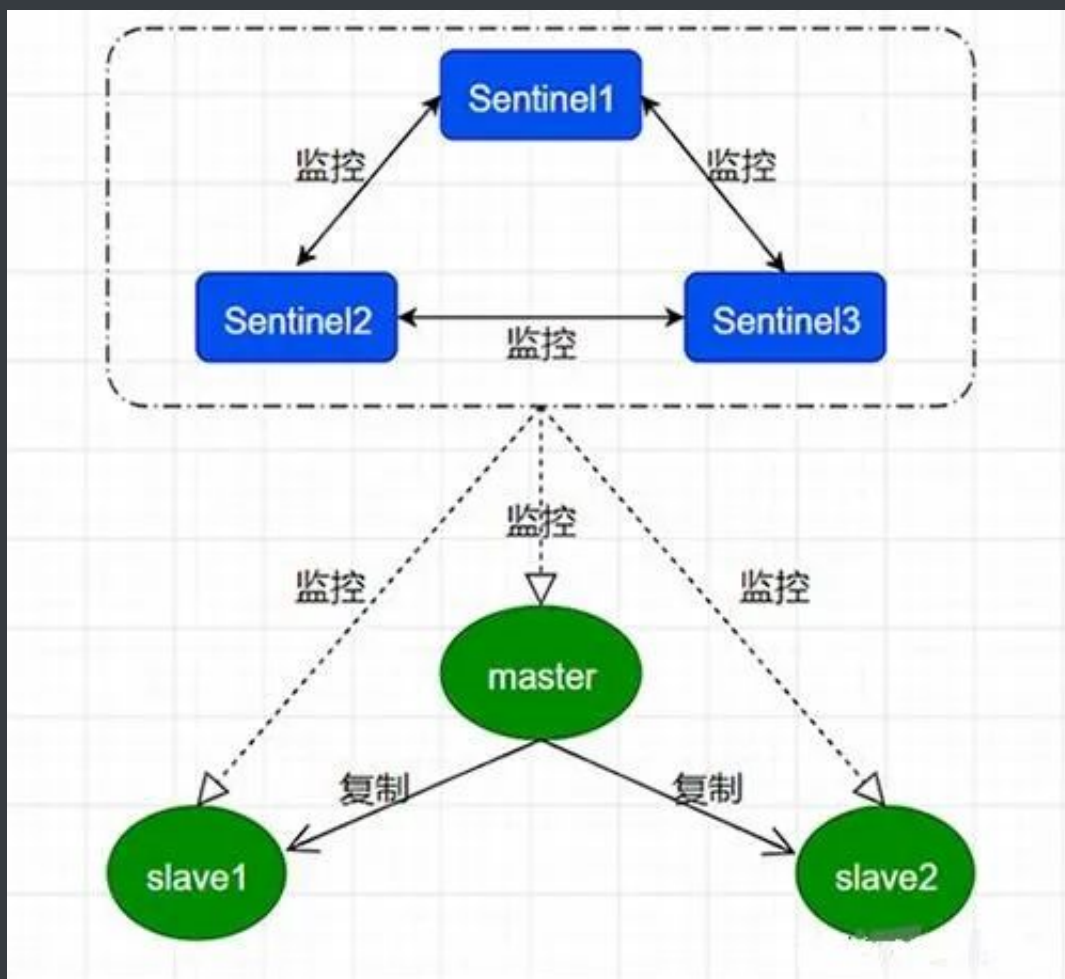
劣势

- 相同数据集的数据而言aof文件要远大于rdb文件，恢复速度慢于rdb
- Aof运行效率要慢于rdb,每秒同步策略效率较好，不同步效率和rdb相同

AOF总结



哨兵简介



Sentinel（哨兵）是Redis 高可用 的解决方案，可以运行多个Sentinel组成一个哨兵分布式系统。

这里我们的 哨兵机制 就是解决这个问题： 故障转移 ， 如果主节点挂掉， 就进行 主从切换 ， 让从节点 升级 为主节点， 继续对外提供服务。

使用流言协议(gossip protocols)来接收主机 是否下线 ； 并使用投票协议(agreement protocols)来决定是否执行 自动故障迁移 ； 以及选择哪个从服务器作为 新的主服务器 。

Sentinel哨兵职责如下：

- 监控 (Monitoring): Sentinel 会不断地定期检查你的主服务器和从服务器是否运作正常。
- 提醒 (Notification): 当被监控的某个 Redis 服务器出现问题时， Sentinel 可以通过 API 向管理员或者其他应用程序发送通知。
- 自动故障迁移 (Automatic failover): 当一个主服务器 不能正常工作 时， Sentinel 会开始一次 自动故障迁移 操作， 它会将失效主服务器的其中一个从服务器 升级 为新的主服务器， 并让失效主服务器的其他从服务器改为复制新的主服务器; 当 客户端 试图连接失效的主服务器时， 集群也会向客户端返回新主服务器的地址， 使得集群可以使用新主服务器代替失效服务器。
- 统一的配置管理： 连接者询问sentinel取得主从的地址。

哨兵搭建

机器准备

主从复制 的搭建，之前的文章讲过了，大家可以去参考

服务器名称	节点类型	IP地址	端口
Node1	Redis服务1(主节点Master)	192.168.14.101	6379
Node2	Redis服务2(从节点slave1)	192.168.14.102	6380
Node3	Redis服务3(从节点slave2)	192.168.14.103	6381
Sentinel1	哨兵服务1	192.168.14.101	26379
Sentinel2	哨兵服务2	192.168.14.102	26380
Sentinel3	哨兵服务3	192.168.14.103	26381

五个主要配置讲解

在每个主从Redis目录下新建一个名为sentinel.conf的文件，在该文件下配置如下命令。

命令总格式：`sentinel <option_name> <master_name> <option_value>`

一：配置sentinel监控master

示例：`sentinel monitor mymaster 127.0.0.1 6380 1`

详解：sentinel监控的master的名字叫做mymaster，地址为127.0.0.1:6380；sentinel在集群中，需要多个sentinel互相沟通来确认某个master是否真的死了；数字1代表，当集群中有1个sentinel认为master死了时，才能真正认为该master已经不可用了。

二：配置sentinel心跳

示例：`sentinel down-after-milliseconds mymaster 5000`

详解：sentinel向master发送心跳PING，确认master是否存活，如果master在 down-after-milliseconds 时间（单位毫秒）范围内没有给sentinel回应 PONG，或者回复一个错误消息，那么sentinel就主观的认为这个master不可用了

三：配置主从切换时，同步新master的slave个数

示例：`sentinel parallel-syncs mymaster 1`

详解：在发生failover主备切换时，这个选项指定了最多可以有多少个slave同时对新的master进行同步数据。这个数字越小，完成failover所需的时间就越长，但是如果这个数字越大，就意味着越多的slave因为replication而不可用。可以设为 1 来保证每次只有一个slave处于不能处理命令请求的状态

四：配置故障转移最大时间

示例：sentinel failover-timeout mymaster 60000（毫秒）

详解：若Sentinel进程在该配置值内未能完成故障转移的操作，则认为本次故障转移操作失败。

五：配置报警脚本

示例：sentinel notification-script mymaster

详解：Sentinel检测到Master主服务器异常时，所要调用的报警脚本。

sentinel配置文件示例

大家按照这个配置，分别给3个sentinel节点进行配置

```
1  # 哨兵sentinel实例运行的端口 默认26379
2  port 26379
3
4  #以守护进程模式启动
5  daemonize yes
6
7  # 哨兵sentinel的工作目录
8  dir /tmp
9
10 #日志文件名
11 logfile "sentinel_26379.log"
12
13 # sentinel监控的master主机
14 sentinel monitor mymaster 192.168.1.108 6379 2
15
16 # sentinel连接主从密码验证，注意必须为主从设置一样的密码
17 # sentinel auth-pass <master-name> <password>
18 sentinel auth-pass mymaster 1234
19
20 # 指定多少毫秒之后 主节点没有应答哨兵sentinel 此时 哨兵主观上认为主节点下线 默认30秒
21 sentinel down-after-milliseconds mymaster 30000
22
23 sentinel parallel-syncs mymaster 1
24
25 # 失效转移最大时间设置
26 sentinel failover-timeout mymaster 180000
27
28 #如果了这个脚本路径，那么必须保证这个脚本存在于这个路径，并且是可执行的，否则sentinel无法正常启动成功。
29 sentinel notification-script mymaster /var/redis/notify.sh
```


启动sentinel

方式1：redis-sentinel redis-sentinel.conf

方式2：redis-server sentinel.conf --sentinel

验证主从切换

kill掉master主节点，模拟主机出现故障

```
3930:X 10 Feb 2020 20:07:56.836 # +monitor master mymaster 192.168.14.101 6379 quorum 2
3930:X 10 Feb 2020 20:07:56.837 * +slave slave 192.168.14.102:6380 192.168.14.102 6380 @ mymaster 192.168.14.101 6379
3930:X 10 Feb 2020 20:07:56.838 * +slave slave 192.168.14.103:6381 192.168.14.103 6381 @ mymaster 192.168.14.101 6379
3930:X 10 Feb 2020 20:07:57.143 * +sentinel sentinel afa691650ea9ad99cbfa6e9a484c7316e7bc9061 192.168.14.103 26381 @ mymaster 192.168.14.101 6379
3930:X 10 Feb 2020 20:08:02.219 * +sentinel sentinel 33927256d42fe2e4166d64263072c6c5578e3890 192.168.14.101 26379 @ mymaster 192.168.14.101 6379
3930:X 10 Feb 2020 20:09:05.039 # +sdown master mymaster 192.168.14.101 6379
3930:X 10 Feb 2020 20:09:05.058 # +new-epoch 1
3930:X 10 Feb 2020 20:09:05.059 # +vote-for-leader afa691650ea9ad99cbfa6e9a484c7316e7bc9061 1
3930:X 10 Feb 2020 20:09:05.111 # +odown master mymaster 192.168.14.101 6379 #quorum 3/2
3930:X 10 Feb 2020 20:09:05.111 # Next failover delay: I will not start a failover before Mon Feb 10 20:15:05 2020
3930:X 10 Feb 2020 20:09:05.366 # +config-update-from sentinel afa691650ea9ad99cbfa6e9a484c7316e7bc9061 192.168.14.103 26381 @ mymaster 192.168.14.101 6379
3930:X 10 Feb 2020 20:09:05.366 # +switch-master mymaster 192.168.14.101 6379 192.168.14.103 6381
3930:X 10 Feb 2020 20:09:05.366 * +slave slave 192.168.14.102:6380 192.168.14.102 6380 @ mymaster 192.168.14.103 6381
3930:X 10 Feb 2020 20:09:05.366 * +slave slave 192.168.14.101:6379 192.168.14.101 6379 @ mymaster 192.168.14.103 6381
3930:X 10 Feb 2020 20:09:35.452 # +sdown slave 192.168.14.101:6379 192.168.14.101 6379 @ mymaster 192.168.14.103 6381
```

上面截图红框框住的几个重要信息，这里先介绍最后一行，switch-master mymaster 192.168.14.101 6379 192.168.14.103 6381，表示master服务器由 6379 切换为 6381 端口的redis服务器。

PS:+switch-master 表示切换主节点

查看6381端口Redis服务器

通过命令info replication查看，我们发现,6381的Redis服务已经切换成master节点了。

另外，也可以查看sentinel.conf 配置文件，里面的 sentinel monitor mymaster 192.168.14.101 6379 2 也自动更改为6381了

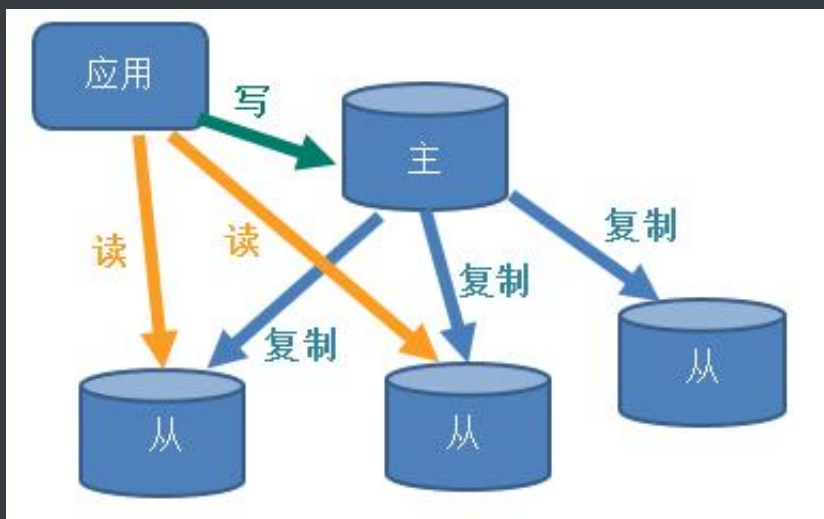

```
127.0.0.1:6381> info replication
# Replication
role:master
connected_slaves:1
slave0:ip=192.168.14.102,port=6380,state=online,offset=67239,lag=0
master_replid:cb69633f1981ff68e6e754481a129ca61b11e5fa
master_replid2:3044f673bd9ded5af250ed443e5ec96039984987
master_repl_offset:67239
second_repl_offset:10068
repl_backlog_active:1
repl_backlog_size:1048576
repl_backlog_first_byte_offset:1
repl_backlog_histlen:67239
```

至此，哨兵模式搭建验证完成。

Redis主从复制

概念

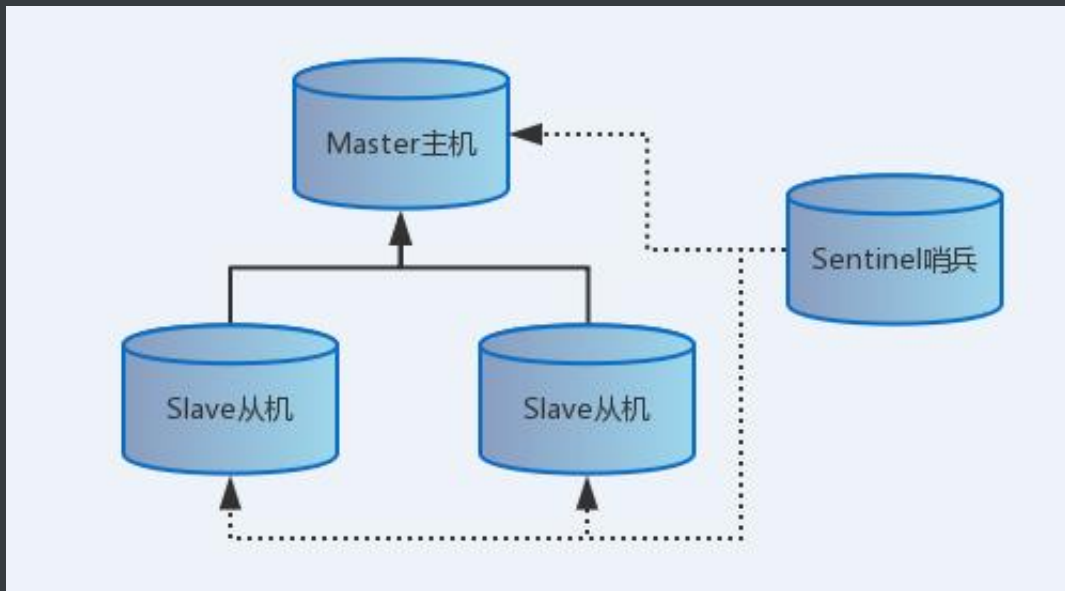
Redis的主从复制概念和MySQL的主从复制大概类似。一台 主机master，一台 从机slaver。master 主机数据更新后根据配置和策略，自动同步到slaver从机，Master以 写为主，Slave以 读为主。



主要用途

- 读写分离：适用于读多写少的应用，增加多个从机，提高读的速度，提高程序并发
- 数据容灾恢复：从机复制主机的数据，相当于数据备份，如果主机数据丢失，那么可以通过从机存储的数据进行恢复。
- 高并发、高可用集群实现的基础：在高并发的场景下，就算主机挂了，从机可以进行主从切换，从机自动成为主机对外提供服务。

一主多从配置



环境准备

老哥太穷了，就用一台机器模拟三个机器。

- 第一步： 将redis.conf复制3份，分别是redis6379.conf、redis6380.conf、redis6381.conf
- 第二步： 修改三个redis.conf文件里的port端口、pid文件名、日志文件名、rdb文件名
- 第三步： 分别打开三个窗口模拟三台服务器，并开启redis服务。

查看当前3台机器主从角色

先用命令 `info replication` 看看3台机器目前的 角色 是什么。

```
1 # 三台机器都是这个状态
2 127.0.0.1:6379> info replication
3 # 角色是master主机
4 role:master
5 # 从机个数为0
6 connected_slaves:0
```

设置主从关系

这里注意，我们只设置从机就可以了，不用设置主机。我们选择 6380 和 6381 作为 从机 。 6379 作为 主机 。

```
1 # 6380 端口
2 127.0.0.1:6380> SLAVEOF 127.0.0.1 6379
3
4 # 6381 端口
5 127.0.0.1:6381> SLAVEOF 127.0.0.1 6379
```

再次查看3台机器目前角色

再次执行命令：`info replication`

```
1  # 主机
2  127.0.0.1:6379> info replication
3  role:master # 角色: 主机
4  connected_slaves:2 #连接的从机个数, 以及从机IP和端口
5  slave0:ip=127.0.0.1,port=6380,state=online,offset=98,lag=1
6  slave1:ip=127.0.0.1,port=6381,state=online,offset=98,lag=1
7
8  # 从机1
9  127.0.0.1:6380> info replication
10 role:slave # 角色: 从机
11 master_host:127.0.0.1 # 主机的IP和端口
12 master_port:6379
13
14 # 从机2
15 127.0.0.1:6381> info replication
16 role:slave # 角色: 从机
17 master_host:127.0.0.1 # 主机的IP和端口
18 master_port:6379
```

搭建成功, 试验一把

- 全量复制: 从机会把主机之前的数据全部都同步过来, 大家可以在从机上get 某key试试。
- 增量复制: 当主机新增数据时, 从机会将该新增数据同步过来, 大家可以在主机上执行命令set key value, 然后在从机上get 该key, 看是否能获取到。

读写分离

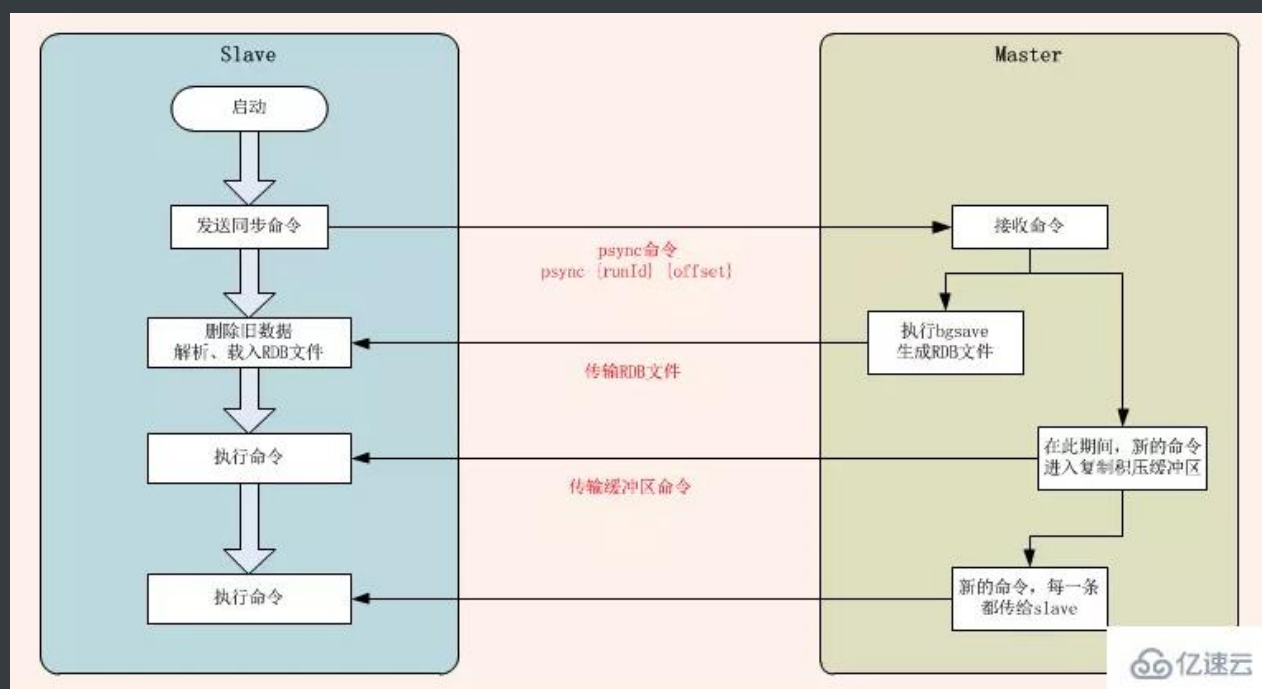
Redis的从机 默认 不允许进行 写操作 , 大家可以在从机上执行命令 `set key value` , 会报错。

```
1  # 6380从机
2  127.0.0.1:6380> set k3 v3
3  (error) READONLY You can't write against a read only slave.
```

主从复制原理



全量复制



①slave发送psync，由于是第一次复制，不知道master的runid，自然也不知道offset，所以发送psync
? -1

②master收到请求，发送master的runid和offset给从节点。

③从节点slave保存master的信息

④主节点bgsave保存rdb文件

⑤主机点发送rdb文件

并且在④和⑤的这个过程中产生的数据，会写到复制缓冲区repl_back_buffer之中去。

⑥主节点发送上面两个步骤产生的buffer到从节点slave

⑦从节点清空原来的数据，如果它之前有数据，那么久会清空数据

⑧从节点slave把rdb文件的数据装载进自身。

全量复制的开销

①bgsave时间

②rdb文件网络传输时间

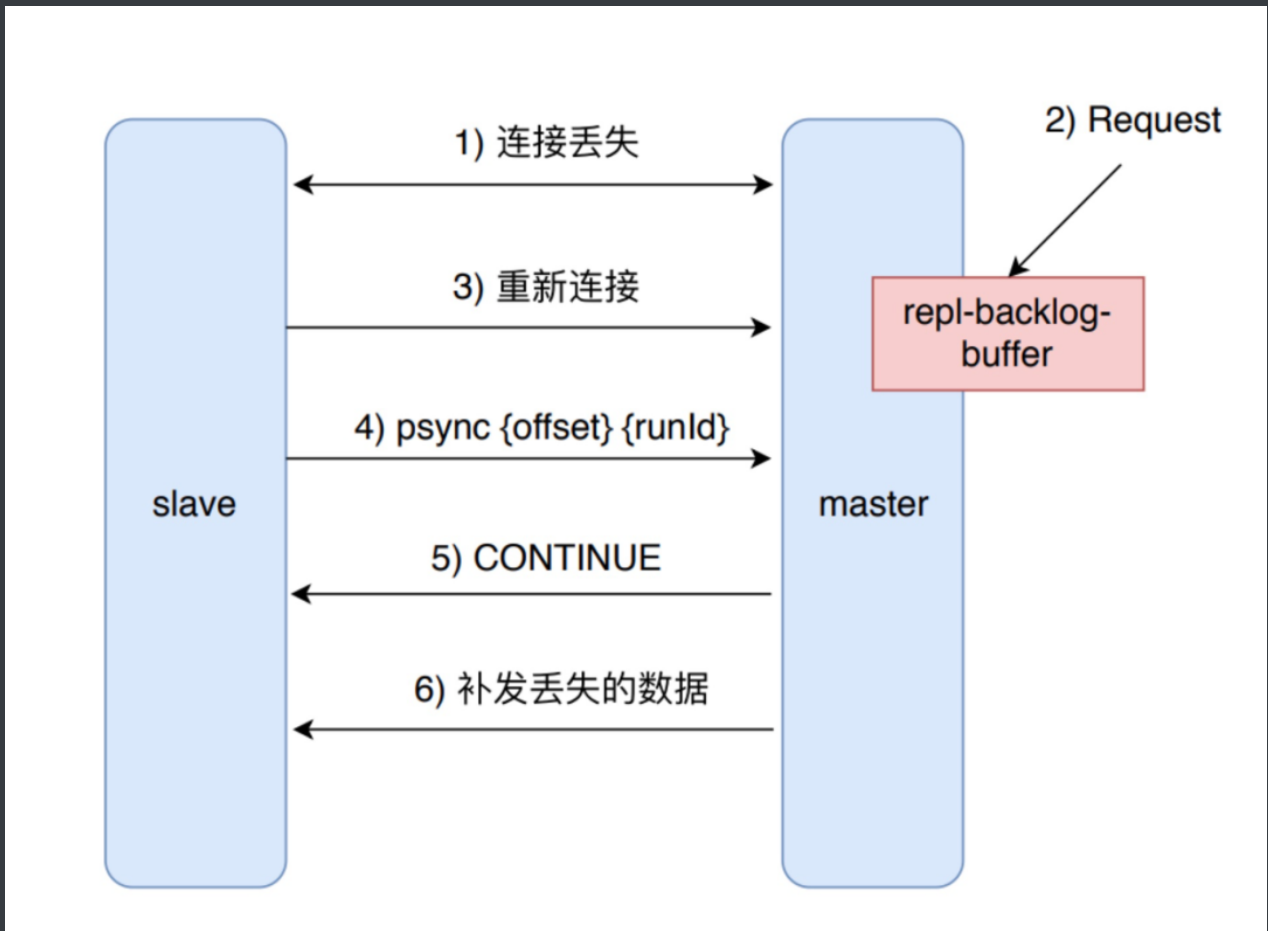
③从节点清空数据的

④从节点加载rdb的时间

⑤可能的aof重写时间，这是针对从节点，例如开启了aof之后，从节点添加buffer数据时候，可能需要aof重写

基于上面的原因，有的情况下不适合使用全量复制，例如网络抖动之后，从节点只需要传送一部分数据，不需要传送全部数据， redis2.8 之后实现了部分复制功能

部分复制



①假设发送网络抖动或者别的情况，暂时失去了连接

②这个时候，master还在继续往buffer里面写数据

③slave重新连接上了master

④slave向master发送自己的offset和runid

⑤master判断slave的offset是否在buffer的队列里面，如果是，那就返回continue给slave，否则需要进行全量复制（因为这说明已经错过了很多数据了）

⑥master发送从slave的offset开始到缓冲区队列结尾的数据给slave

为什么要用 redis ？ 为什么要用缓存？

高性能：

假如用户第一次访问数据库中的某些数据。这个过程会比较慢，因为是从硬盘上读取的。将该用户访问的数据存在缓存中，这样下一次再访问这些数据的时候就可以直接从缓存中获取了。操作缓存就是直接操作内存，所以速度相当快。如果数据库中的对应数据改变的之后，同步改变缓存中相应的数据即可！

高并发：

直接操作缓存能够承受的请求是远远大于直接访问数据库的，所以我们可以考虑把数据库中的部分数据转移到缓存中去，这样用户的一部分请求会直接到缓存这里而不用经过数据库。

为什么要用 redis 而不用 map/guava 做缓存?

缓存分为本地缓存和分布式缓存。以 Java 为例，使用自带的 map 或者 guava 实现的是本地缓存，最主要的特点是轻量以及快速，生命周期随着 jvm 的销毁而结束，并且在多实例的情况下，每个实例都需要各自保存一份缓存，缓存不具有一致性。

使用 redis 或 memcached 之类的称为分布式缓存，在多实例的情况下，各实例共用一份缓存数据，缓存具有一致性。缺点是需要保持 redis 或 memcached 服务的高可用，整个程序架构上较为复杂。

redis 的线程模型是怎么样的?

redis 内部使用文件事件处理器 file event handler，这个文件事件处理器是单线程的，所以 redis 才叫做单线程的模型。它采用 IO 多路复用机制同时监听多个 socket，根据 socket 上的事件来选择对应的事件处理器进行处理。

文件事件处理器的结构包含 4 个部分：

- 多个 socket
- IO 多路复用程序
- 文件事件分派器
- 事件处理器（连接应答处理器、命令请求处理器、命令回复处理器）多个 socket 可能会并发产生不同的操作，每个操作对应不同的文件事件，但是 IO 多路复用程序会监听多个 socket，会将 socket 产生的事件放入队列中排队，事件分派器每次从队列中取出一个事件，把该事件交给对应的事件处理器进行处理。

redis 和 memcached 的区别?

存储方式不同：memcache 把数据全部存在内存之中，断电后会挂掉，数据不能超过内存大小；Redis 有部份存在硬盘上，这样能保证数据的持久性。

数据支持类型：memcache 对数据类型支持相对简单；Redis 有复杂的数据类型。

使用底层模型不同：它们之间底层实现方式，以及与客户端之间通信的应用协议不一样，Redis 自己构建了 vm 机制，因为一般的系统调用系统函数的话，会浪费一定的时间去移动和请求。value 值大小不同：Redis 最大可以达到 1gb；memcache 只有 1mb。

如何实现 redis 事务?

Redis 通过 MULTI、EXEC、WATCH 等命令来实现事务(transaction)功能。事务提供了一种将多个命令请求打包，然后一次性、按顺序地执行多个命令的机制，并且在事务执行期间，服务器不会中断事务而改去执行其他客户端的命令请求，它会将事务中的所有命令都执行完毕，然后才去处理其他客户端的命令请求。

在传统的关系式数据库中，常常用 ACID 性质来检验事务功能的可靠性和安全性。在 Redis 中，事务总是具有原子性（Atomicity）、一致性（Consistency）和隔离性（Isolation），并且当 Redis 运行在某种特定的持久化模式下时，事务也具有持久性（Durability）。

什么是 RedLock?

获取当前时间（start）。

依次向 N 个 Redis 节点请求锁。请求锁的方式与从单节点 Redis 获取锁的方式一致。为了保证在某个 Redis 节点不可用时该算法能够继续运行，获取锁的操作都需要设置超时时间，需要保证该超时时间远小于锁的有效时间。这样才能保证客户端在向某个 Redis 节点获取锁失败之后，可以立刻尝试下一个节点。

计算获取锁的过程总共消耗多长时间（consumeTime = end - start）。如果客户端从大多数 Redis 节点（ $\geq N/2 + 1$ ）成功获取锁，并且获取锁总时长没有超过锁的有效时间，这种情况下，客户端会认为获取锁成功，否则，获取锁失败。

如果最终获取锁成功，锁的有效时间应该重新设置为锁最初的有效时间减去 consumeTime。

如果最终获取锁失败，客户端应该立刻向所有 Redis 节点发起释放锁的请求。

说说 Redis 都有哪些应用场景?

缓存：这应该是 Redis 最主要的功能了，也是大型网站必备机制，合理地使用缓存不仅可以加快数据的访问速度，而且能够有效地降低后端数据源的压力。

共享Session：对于一些依赖 session 功能的服务来说，如果需要从单机变成集群的话，可以选择 redis 来统一管理 session。

消息队列系统：消息队列系统可以说是一个大型网站的必备基础组件，因为其具有业务解耦、非实时业务削峰等特性。Redis 提供了发布订阅功能和阻塞队列的功能，虽然和专业的消息队列比还不够足够强大，但是对于一般的消息队列功能基本可以满足。比如在分布式爬虫系统中，使用 redis 来统一管理 url 队列。

分布式锁：在分布式服务中。可以利用 Redis 的 setnx 功能来编写分布式的锁，虽然这个可能不是太常用。当然还有诸如排行榜、点赞功能都可以使用 Redis 来实现，但是 Redis 也不是什么都可以做，比如数据量特别大时，不适合 Redis，我们知道 Redis 是基于内存的，虽然内存很便宜，但是如果你每天的数据量特别大，比如几亿条的用户行为日志数据，用 Redis 来存储的话，成本相当的高。

单线程的 Redis 为什么这么快?

Redis 有多快？官方给出的答案是读写速度 10 万/秒，如果说这是在单线程情况下跑出来的成绩，你会不会惊讶？为什么单线程的 Redis 速度这么快？原因有以下几点：

纯内存操作：

- Redis 是完全基于内存的，所以读写效率非常的高，当然 Redis 存在持久化操作，在持久化操作都是 fork 子进程和利用 Linux 系统的页缓存技术来完成，并不会影响 Redis 的性能。
- 单线程操作：单线程并不是坏事，单线程可以避免频繁的上下文切换，频繁的上下文切换也会影响性能的。
- 合理高效的数据结构
- 采用了非阻塞 I/O 多路复用机制：多路I/O复用模型是利用 select、poll、epoll 可以同时监察多个流的 I/O 事件的能力，在空闲的时候，会把当前线程阻塞掉，当有一个或多个流有 I/O 事件时，就从阻塞态中唤醒，于是程序就会轮询一遍所有的流（epoll 是只轮询那些真正发出了事件的流），并且只依次顺序的处理就绪的流，这种做法就避免了大量的无用操作。

说一说 Redis 的数据过期淘汰策略？

Redis 中数据过期策略采用定期删除+惰性删除策略。

1、定期删除、惰性删除策略是什么？

- 定期删除策略：Redis 启用一个定时器定时监视所有的 key，判断key是否过期，过期的话就删除。这种策略可以保证过期的 key 最终都会被删除，但是也存在严重的缺点：每次都遍历内存中所有的数据，非常消耗 CPU 资源，并且当 key 已过期，但是定时器还处于未唤起状态，这段时间内 key 仍然可以用。
- 惰性删除策略：在获取 key 时，先判断 key 是否过期，如果过期则删除。这种方式存在一个缺点：如果这个 key 一直未被使用，那么它一直在内存中，其实它已经过期了，会浪费大量的空间。

2、定期删除+惰性删除策略是如何工作的？

这两种策略天然的互补，结合起来之后，定时删除策略就发生了一些改变，不在是每次扫描全部的 key 了，而是随机抽取一部分 key 进行检查，这样就降低了对 CPU 资源的损耗，惰性删除策略互补了为检查到的key，基本上满足了所有要求。

但是有时候就是那么的巧，既没有被定时器抽取到，又没有被使用，这些数据又如何从内存中消失？没关系，还有内存淘汰机制，当内存不够用时，内存淘汰机制就会上场。Redis 内存淘汰机制有以下几种策略：

- volatile-lru：从已设置过期时间的数据集（server.db[i].expires）中挑选最近最少使用的数据淘汰
- volatile-ttl：从已设置过期时间的数据集（server.db[i].expires）中挑选将要过期的数据淘汰
- volatile-random：从已设置过期时间的数据集（server.db[i].expires）中任意选择数据淘汰
- allkeys-lru：当内存不足以容纳新写入数据时，在键空间中，移除最近最少使用的key（这个是最常用的）
- allkeys-random：从数据集（server.db[i].dict）中任意选择数据淘汰
- no-eviction：禁止驱逐数据，永不过期，也就是说当内存不足以容纳新写入数据时，新写入操作会报错。这个应该没人使用吧！（默认值）

4.0版本后增加以下两种：

- volatile-lfu：从已设置过期时间的数据集(server.db[i].expires)中挑选最不经常使用的数据淘汰

- allkeys-lfu: 当内存不足以容纳新写入数据时, 在键空间中, 移除最经常使用的key

手写一个 LRU 算法

```
1  class LRUCache<K, V> extends LinkedHashMap<K, V> {
2      private final int CACHE_SIZE;
3
4      /**
5       * 传递进来最多能缓存多少数据
6       *
7       * @param cacheSize 缓存大小
8       */
9      public LRUCache(int cacheSize) {
10         // true 表示让 linkedHashMap 按照访问顺序来进行排序, 最近访问的放在头部, 最
            老访问的放在尾部。
11         super((int) Math.ceil(cacheSize / 0.75) + 1, 0.75f, true);
12         CACHE_SIZE = cacheSize;
13     }
14
15     @Override
16     protected boolean removeEldestEntry(Map.Entry<K, V> eldest) {
17         // 当 map中的数据量大于指定的缓存个数的时候, 就自动删除最老的数据。
18         return size() > CACHE_SIZE;
19     }
20 }
```

jedis 和 Redisson 有哪些区别?

jedis: 提供了比较全面的 Redis 命令的支持。

Redisson: 实现了分布式和可扩展的 Java 数据结构, 与 jedis 相比 Redisson 的功能相对简单, 不支持排序、事务、管道、分区等 Redis 特性。

请问Redis的rehash怎么做的, 为什么要渐进rehash, 渐进rehash又是怎么实现的?

因为redis是单线程, 当K很多时, 如果一次性将键值对全部rehash, 庞大的计算量会影响服务器性能, 甚至可能会导致服务器在一段时间内停止服务。不可能一步完成整个rehash操作, 所以redis是分多次、渐进式的rehash。渐进性哈希分为两种:

1) 操作redis时, 额外做一步rehash

对redis做读取、插入、删除等操作时, 会把位于table[dict->rehashidx]位置的链表移动到新的dictht中, 然后把rehashidx做加一操作, 移动到后面一个槽位。

2) 后台定时任务调用rehash

后台定时任务rehash调用链，同时可以通过server.hz控制rehash调用频率

请问Redis的数据类型底层怎么实现？

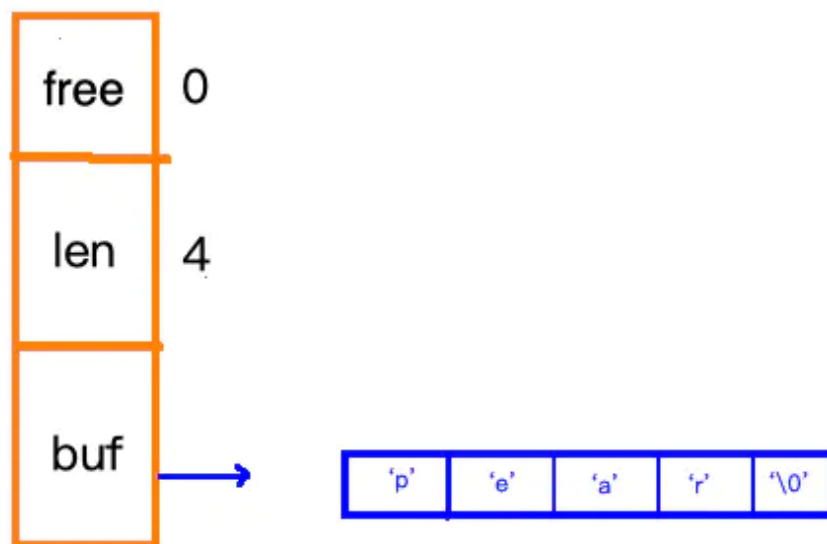
- 1) 字符串：整数值、embstr编码的简单动态字符串、简单动态字符串（SDS）
- 2) 列表：压缩列表、双端链表
- 3) 哈希：压缩列表、字典
- 4) 集合：整数集合、字典
- 5) 有序集合：压缩列表、跳跃表和字典

动态字符串SDS

SDS是"simple dynamic string"的缩写。redis中所有场景中出现的字符串，基本都是由SDS来实现的

- 所有非数字的key。例如set msg "hello world" 中的key msg.
- 字符串数据类型的值。例如` set msg "hello world"中的msg的值"hello world"
- 非字符串数据类型中的“字符串值”。例如RPUSH fruits "apple" "banana" "cherry"中的"apple" "banana" "cherry"

SDS长这样：



- free:还剩多少空间
- len:字符串长度
- buf:存放的字符数组

空间预分配

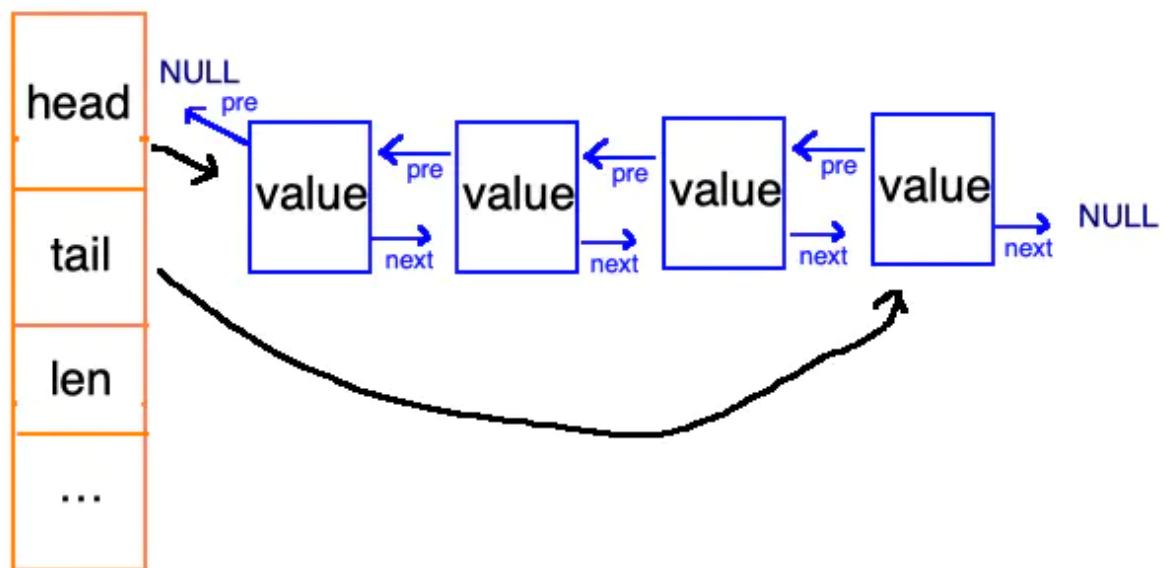
为减少修改字符串带来的内存重分配次数，sds采用了“一次管够”的策略：

- 若修改之后sds长度小于1MB,则多分配现有len长度的空间
- 若修改之后sds长度大于等于1MB，则扩充除了满足修改之后的长度外，额外多1MB空间

惰性空间释放

为避免缩短字符串时候的内存重分配操作，sds在数据减少时，并不立刻释放空间。

双向链表



分两部分，一部分是“统筹部分”：橘黄色，

一部分是“具体实施方”：蓝色。

主体“统筹部分”：

- head指向具体双向链表的头
- tail指向具体双向链表的尾
- len双向链表的长度

具体“实施方”：一目了然的双向链表结构，有前驱pre有后继next

由list和listNode两个数据结构构成。

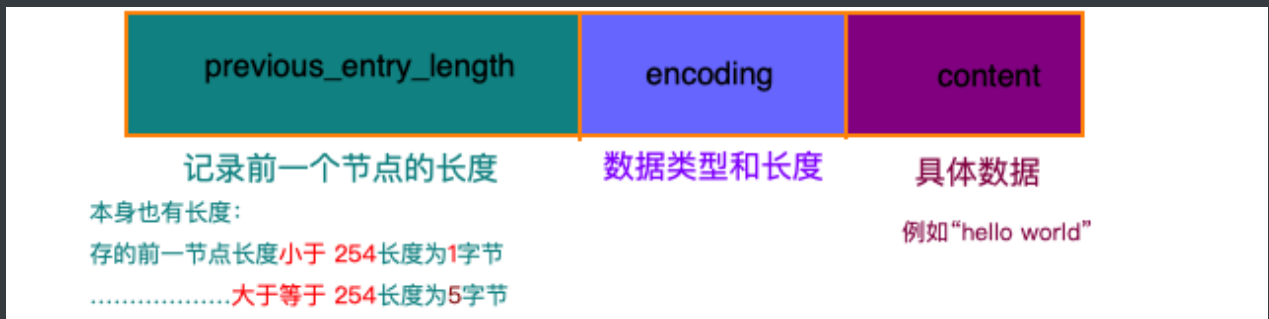
ziplist

压缩列表。redis的列表键和哈希键的底层实现之一。此数据结构是为了节约内存而开发的。

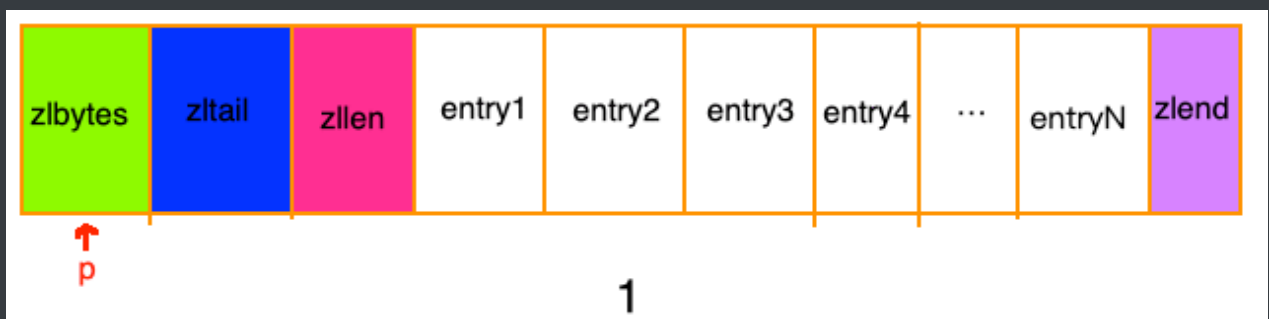
和各种语言的数组类似，它是由连续的内存块组成的，这样一来，由于内存是连续的，就减少了很多内存碎片和指针的内存占用，进而节约了内存。



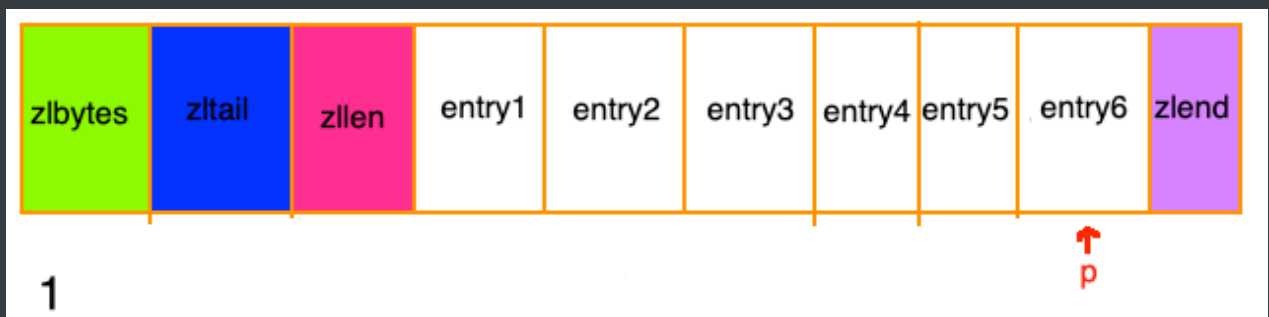
然后文中的entry的结构是这样的：



元素的遍历



然后再根据ziplist节点元素中的previous_entry_length属性，来逐个遍历：

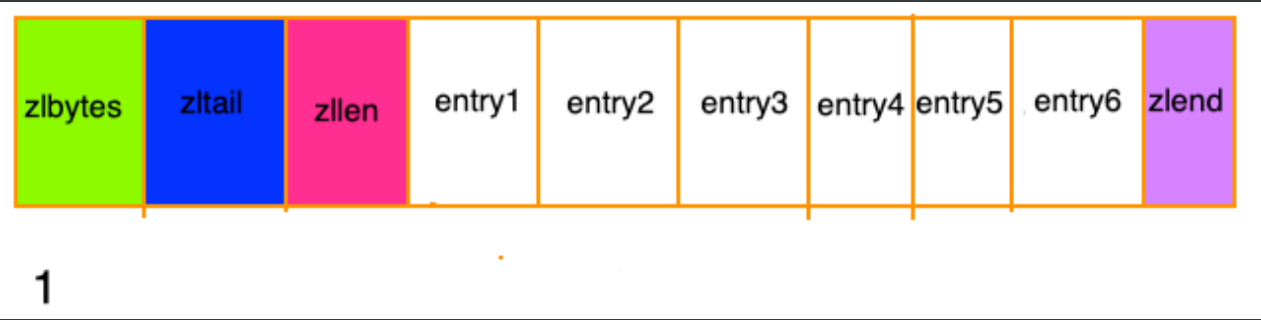


连锁更新

再次看看entry元素的结构，有一个previous_entry_length字段，他的长度要么都是1个字节，要么都是5个字节：

- 前一节点的长度小于254字节，则previous_entry_length长度为1字节
- 前一节点的长度大于254字节，则previous_entry_length长度为5字节

假设现在存在一组压缩列表，长度都在250字节至253字节之间，突然新增一新节点new， 长度大于等于254字节，会出现：



程序需要不断的对压缩列表进行空间重分配工作，直到结束。

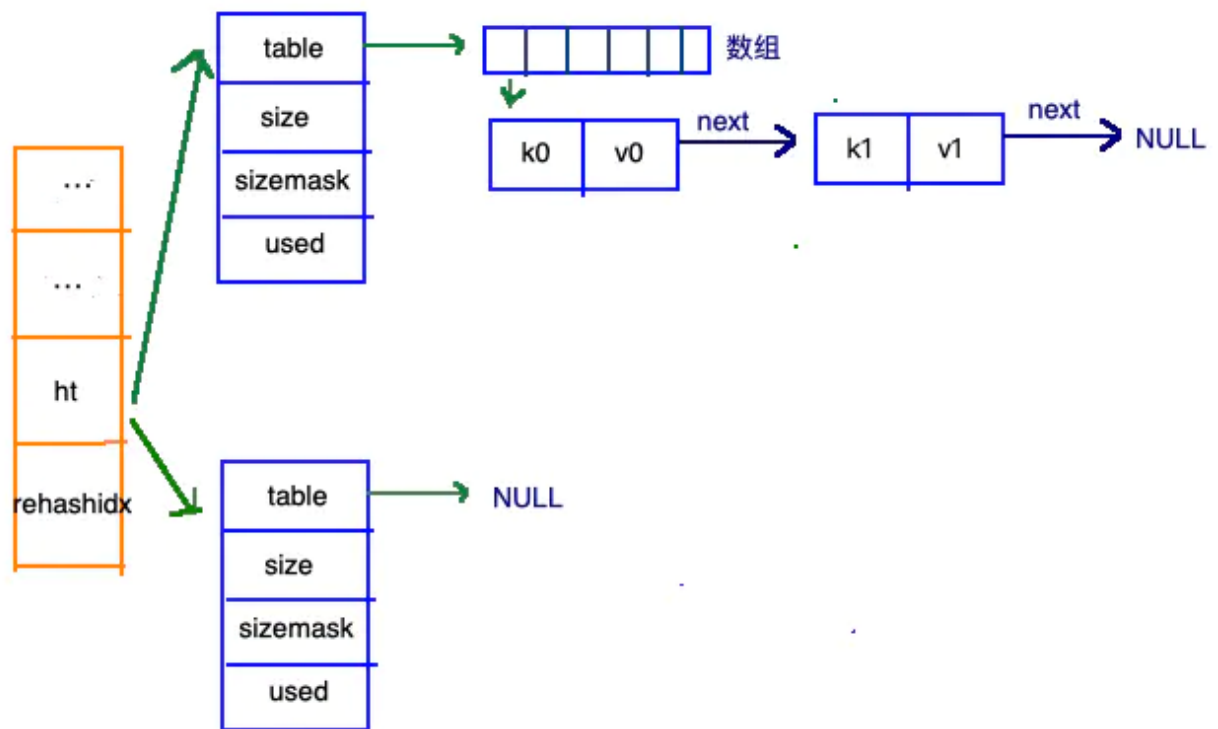
除了增加操作，删除操作也有可能带来“连锁更新”。 请看下图，ziplist中所有entry节点的长度都在250字节至253字节之间，big节点长度大于254字节，small节点小于254字节。



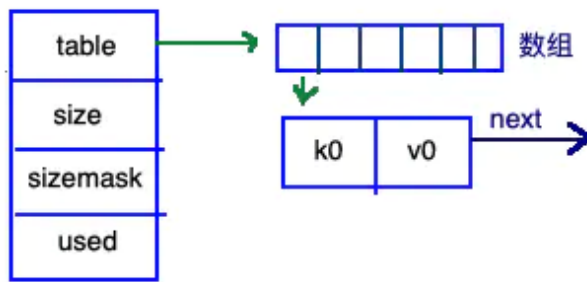
哈希表

哈希表略微有点复杂。哈希表的制作方法一般有两种，一种是：开放寻址法，一种是拉链法。redis的哈希表的制作使用的是拉链法。

整体结构如下图：



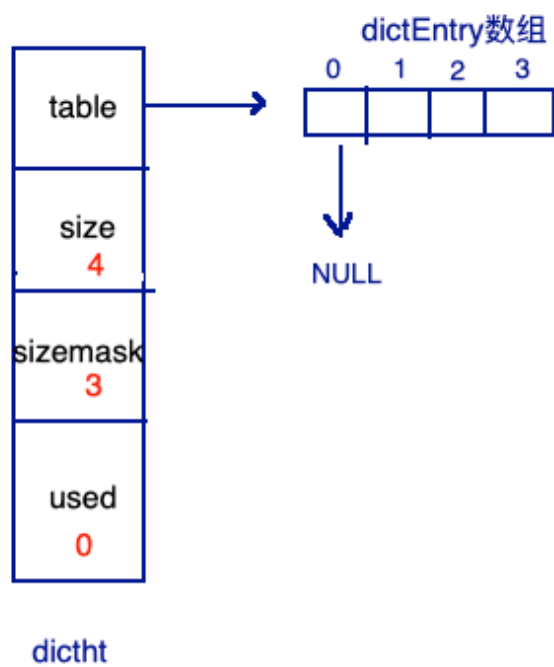
也是分为两部分：左边橘黄色部分和右边蓝色部分，同样，也是”统筹“和”实施“的关系。具体哈希表的实现，都是在蓝色部分实现的。先来看看蓝色部分：



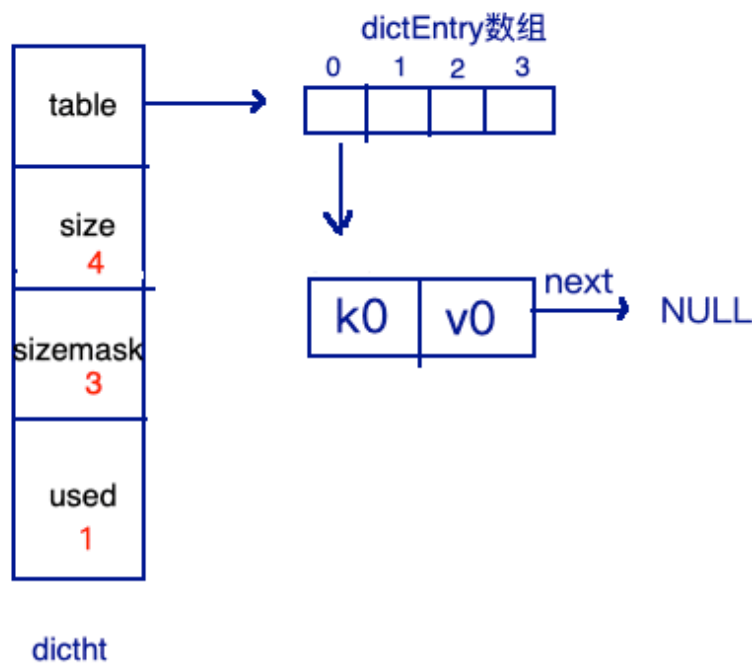
这也分为左右两边“统筹”和“实施”的两部分。

右边部分很容易理解：就是通常拉链表实现的哈希表的样式；数组就是bucket，一般不同的key首先会定位到不同的bucket，若key重复，就用链表把冲突的key串起来。

新建key的过程：



假如重复了:



此时，又新增一个key1和value1

1

rehash

再来看看哈希表总体图中左边橘黄色的“统筹”部分，其中有两个关键的属性：ht和rehashidx。ht是一个数组，有且只有俩元素ht[0]和ht[1];其中，ht[0]存放的是redis中使用的哈希表，而ht[1]和rehashidx和哈希表的rehash有关。

rehash指的是重新计算键的哈希值和索引值，然后将键值对重排的过程。

加载因子（load factor） = $ht[0].used / ht[0].size$ 。

扩容和收缩标准

扩容：

- 没有执行BGSAVE和BGREWRITEAOF指令的情况下，哈希表的加载因子大于等于1。
- 正在执行BGSAVE和BGREWRITEAOF指令的情况下，哈希表的加载因子大于等于5。

收缩：

- 加载因子小于0.1时，程序自动开始对哈希表进行收缩操作。

扩容和收缩的数量

扩容：第一个大于等于 $ht[0].used * 2$ 的 2^n (2 的 n 次方幂)。

收缩：第一个大于等于 $ht[0].used$ 的 2^n (2 的 n 次方幂)。

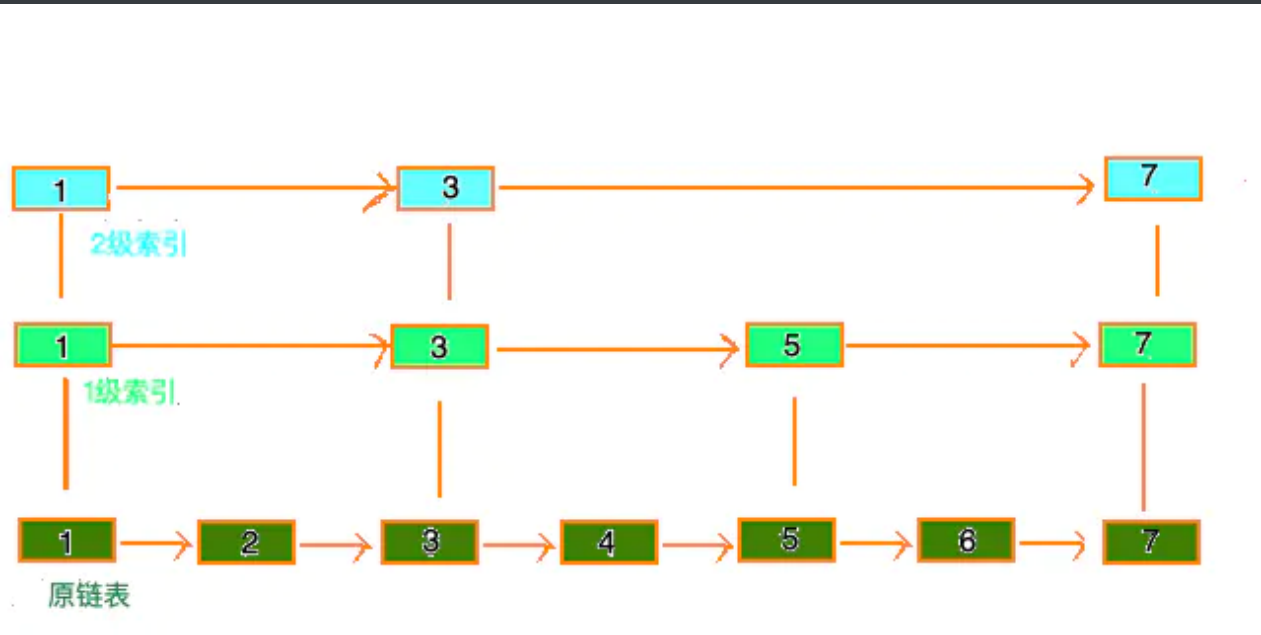
intset

整数集合是集合键的底层实现方式之一。

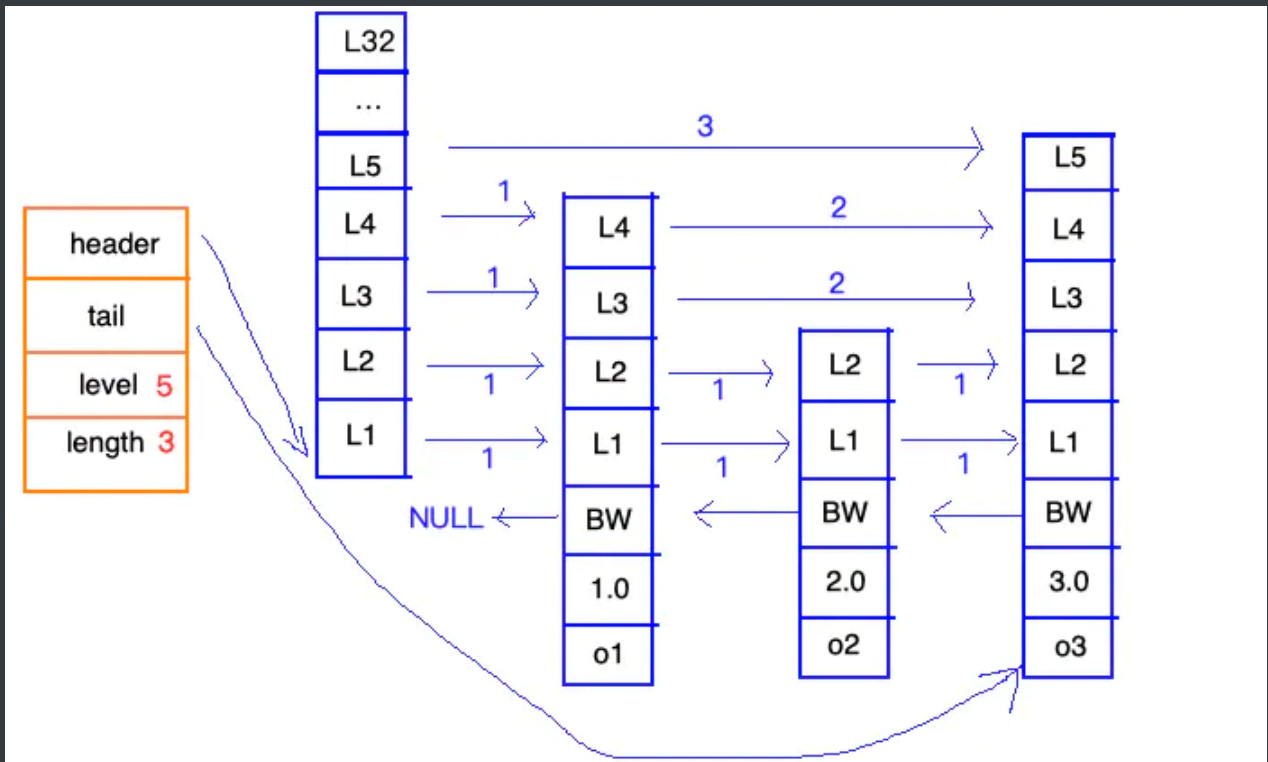


跳表

跳表这种数据结构长这样：



redis中把跳表抽象成如下所示：



看这个图，左边“统筹”，右边实现。 统筹部分有以下几点说明：

- header: 跳表表头
- tail:跳表表尾
- level:层数最大的那个节点的层数
- length: 跳表的长度

实现部分有以下几点说明：

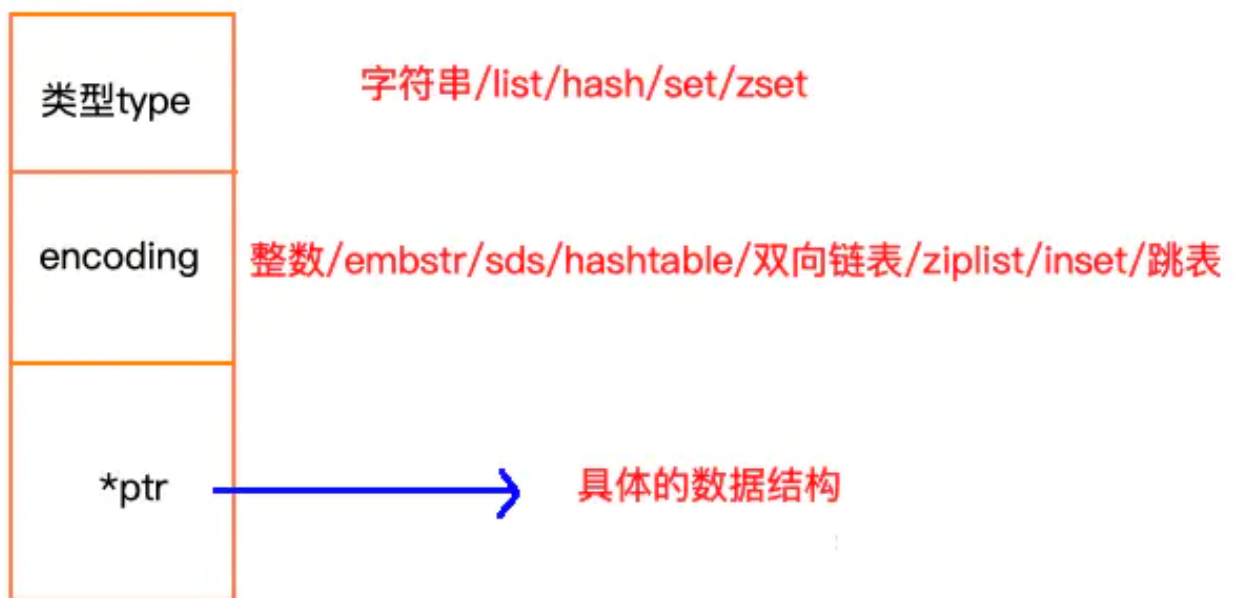
- 表头：是链表的哨兵节点，不记录主体数据。
- 是个双向链表
- 分值是有顺序的
- o1、o2、o3是节点所保存的成员，是一个指针，可以指向一个SDS值。
- 层级高度最高是32。没每次创建一个新的节点的时候，程序都会随机生成一个介于1和32之间的值作为level数组的大小，这个大小就是“高度”

redis五种数据结构的实现

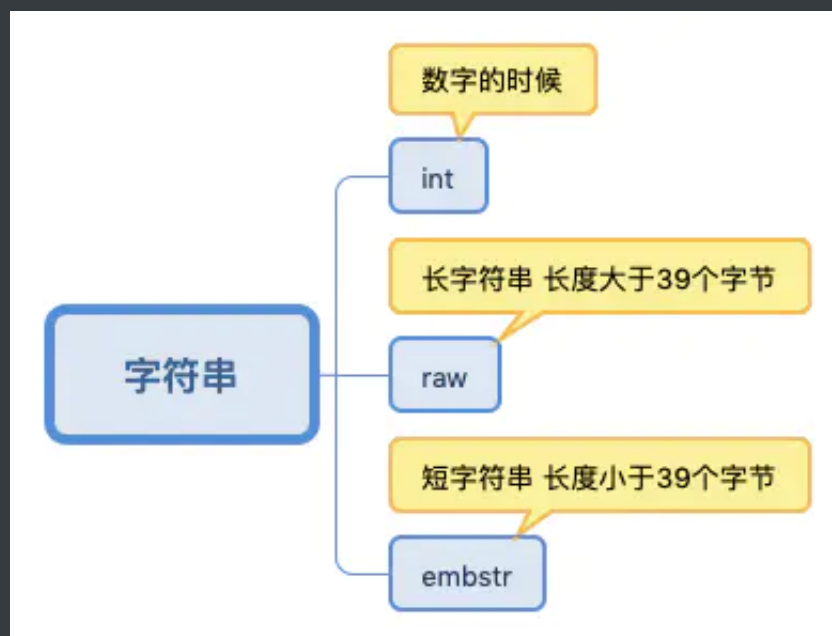
redis对象

redis中并没有直接使用以上所说的各种数据结构来实现键值数据库，而是基于一种对象，对象底层再间接的引用上文所说的具体的数据结构。

结构如下图：

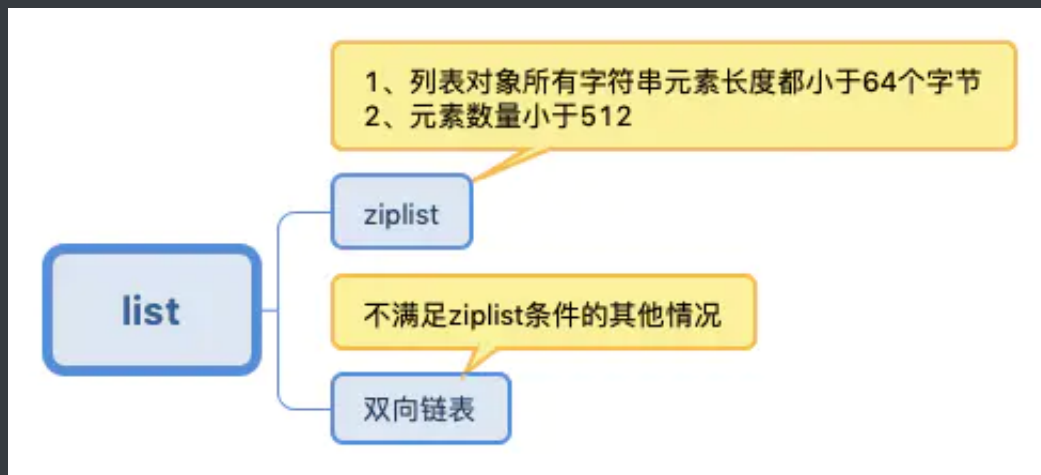


字符串

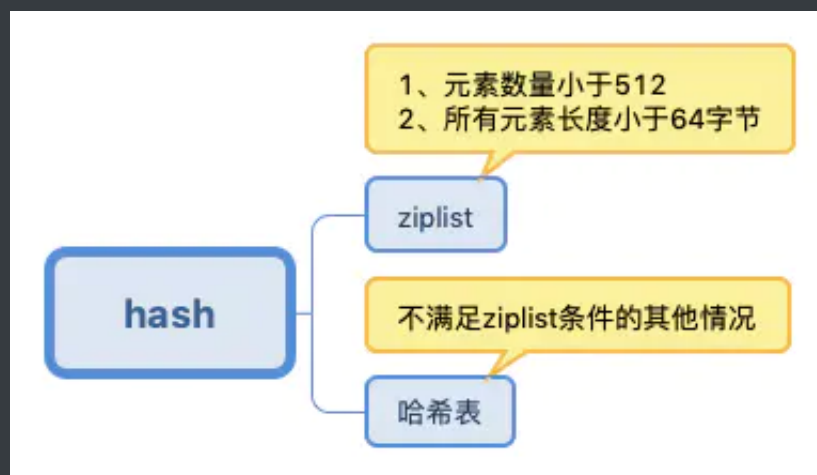


其中：embstr和raw都是由SDS动态字符串构成的。唯一区别是：raw是分配内存的时候，redisobject和sds 各分配一块内存，而embstr是redisobject和在一块儿内存中。

列表



hash



set



zset



缓存雪崩



举例

双十一期间，所有用户一打开淘宝就是进入 首页，首页的压力非常大，为了提高并发，将网站 首页数据 都缓存到 redis 里，所有的redis key 失效时间 都是 3小时。

双十一当天大量用户 剁手狂欢，这时候3个小时过去了，redis里首页的key 缓存全部失效，这时候redis里查询不到数据了，只能去 数据库 中查询，造成数据库无法响应 挂掉。

用户进不去首页没法剁手了，马爸爸 就 不开心 了，把这个程序员外派到 非洲 了。

一句话总结

在高并发下，大量缓存key在 同一时间失效，大量请求直接落在数据库上，导致数据库宕机。

解决方案

- 随机设置key失效时间，避免大量key集体失效。

```
1 setRedis (Key, value, time + Math.random() * 10000);
```


- 若是集群部署，可将热点数据均匀分布在不同的Redis库中也能够避免key全部失效问题
- 不设置过期时间
- 跑定时任务，在缓存失效前刷进新的缓存

缓存穿透

举例

老哥做了一个网站 火了，动了别人的蛋糕，于是开始 疯狂攻击 老哥的网站，由于老哥 网络安全 方面学艺不精被人钻了空子。

某人用脚本疯狂的给老哥发送请求，查询 `id = -1` 的数据，redis并没有这样的数据，这时候就 穿透 redis，直接打到了 数据库 上。

半夜老哥在睡觉并没有察觉，他疯狂攻击老哥一晚上，结果把 数据库 搞挂了，然后老哥的 网站 也挂了。

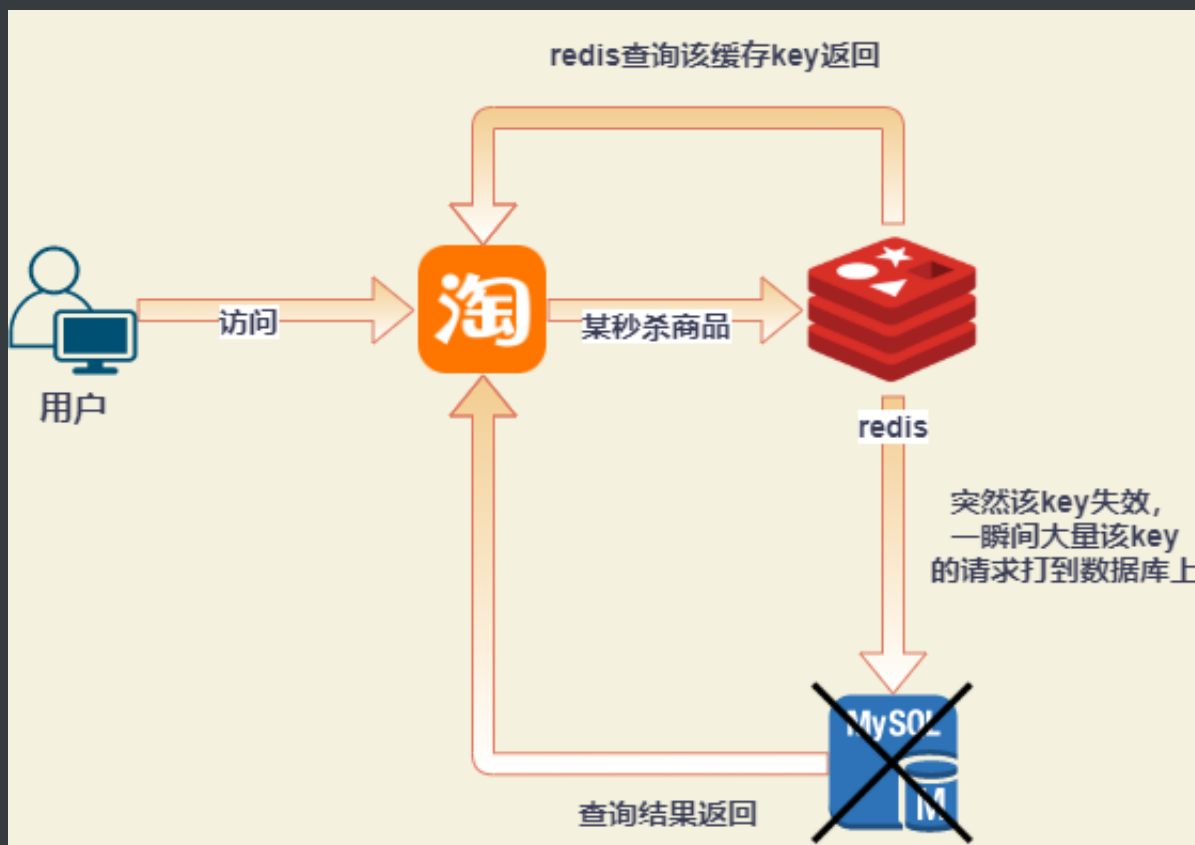
一句话总结

redis缓存 和 数据库 中没有相关数据(例用户直接携带 `id<=0` 的参数不断发起请求)，redis中没有这样的数据， 无法进行拦截，直接被穿透到 数据库，导致数据库压力过大宕机。

解决方案

- 对不存在的数据缓存到redis中，设置key，value值为null(不管是数据未null还是系统bug问题)，并设置一个短期过期时间段，避免过期时间过长影响正常用户使用。
- 拉黑该IP地址
- 对参数进行校验，不合法参数进行拦截
- 布隆过滤器 将所有可能存在的数据哈希到一个足够大的bitmap(位图)中，一个一定不存在的数据会被 这个bitmap拦截掉，从而避免了对底层存储系统的查询压力。

缓存击穿



举例

双十一 马爸爸 突发奇想，想 拍卖 自己 穿了20年的老布鞋 ，并且附带本人签名，程序员将该鞋的信息存到了redis中，设置了 3小时 过期。寻思3小时够他们抢了吧，但他低估了马爸爸的魅力。

该商品引起了一千万人关注，这些人不断的竞拍这双鞋，价格越拍越高，马爸爸乐开了花。

竞拍了 2小时59 分，马上就要拍到一个亿了，突然这双鞋在redis里的key数据 过期了 ，导致该key的大量请求，都打到了数据库，直接导致数据库挂掉了，服务无法响应。

竞拍到此结束，鞋没卖出去，马爸爸又不开心了，把这个程序员也 外派到非洲 了。

一句话总结

某一个 热点key，在不停地扛着高并发，当这个热点key在 失效的一瞬间 ，持续的高并发访问就 击破缓存 直接访问数据库，导致数据库宕机。

解决方案

- 设置热点数据"永不过期"
- 加上互斥锁：上面的现象是多个线程同时去查询数据库的这条数据，那么我们可以在第一个查询数据的请求上使用一个互斥锁来锁住它

其他的线程走到这一步拿不到锁就等着，等第一个线程查询到了数据，然后将数据放到redis缓存起来。后面的线程进来发现已经有缓存了，就直接走缓存

```

1  # 简单的分布式锁实现，之后我们重点会讲分布式锁
2  public String get(key) {
3      String value = redis.get(key);
4      if (value == null) { //代表缓存值过期
5          //设置3min的超时，防止del操作失败的时候，下次缓存过期一直不能load db
6          String keynx = key.concat(":nx");
7          if (redis.setnx(keynx, 1, 3 * 60) == 1) { //代表设置成功
8              value = db.get(key);
9              redis.set(key, value, expire_secs);
10             redis.del(keynx);
11         } else {
12             //这个时候代表同时时候的其他线程已经load db并回设到缓存了，这时候重试获取缓存值即可
13             sleep(50);
14             get(key); //重试
15         }
16     } else {
17         return value;
18     }
19 }

```

最后总结

雪崩是 大面积 的key缓存失效；穿透是redis里 不存在 这个缓存key；击穿是redis 某一个热点 key突然失效，最终的受害者都是数据库。

思考

未雨绸缪：将redis、MySQL等搭建成高可用的集群，防止单点。

亡羊补牢：服务中进行限流 + 降级，防止MySQL被打崩溃。

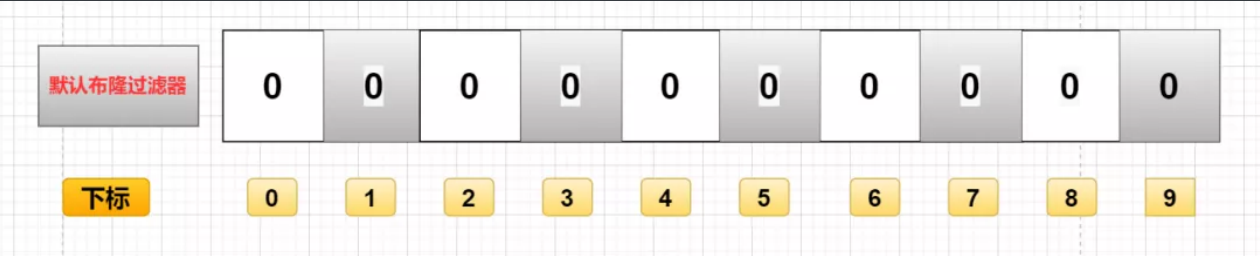
重振旗鼓：Redis 持久化 RDB+AOF，宕机重启，自动从磁盘上加载数据，快速恢复缓存数据。

什么是布隆过滤器

布隆过滤器（Bloom Filter），是1970年，由一个叫布隆的小伙子提出的，距今已经五十年了，和老哥一样老。

它实际上是一个很长的二进制向量和一系列随机映射函数，二进制大家应该都清楚，存储的数据不是0就是1，默认是0。

主要用于判断一个元素是否在一个集合中，0代表不存在某个数据，1代表存在某个数据。



布隆过滤器用途

- 解决Redis缓存穿透（今天重点讲解）
- 在爬虫时，对爬虫网址进行过滤，已经存在布隆中的网址，不在爬取。
- 垃圾邮件过滤，对每一个发送邮件的地址进行判断是否在布隆的黑名单中，如果在就判断为垃圾邮件。

以上只是简单的用途举例，大家可以举一反三，灵活运用在工作中。

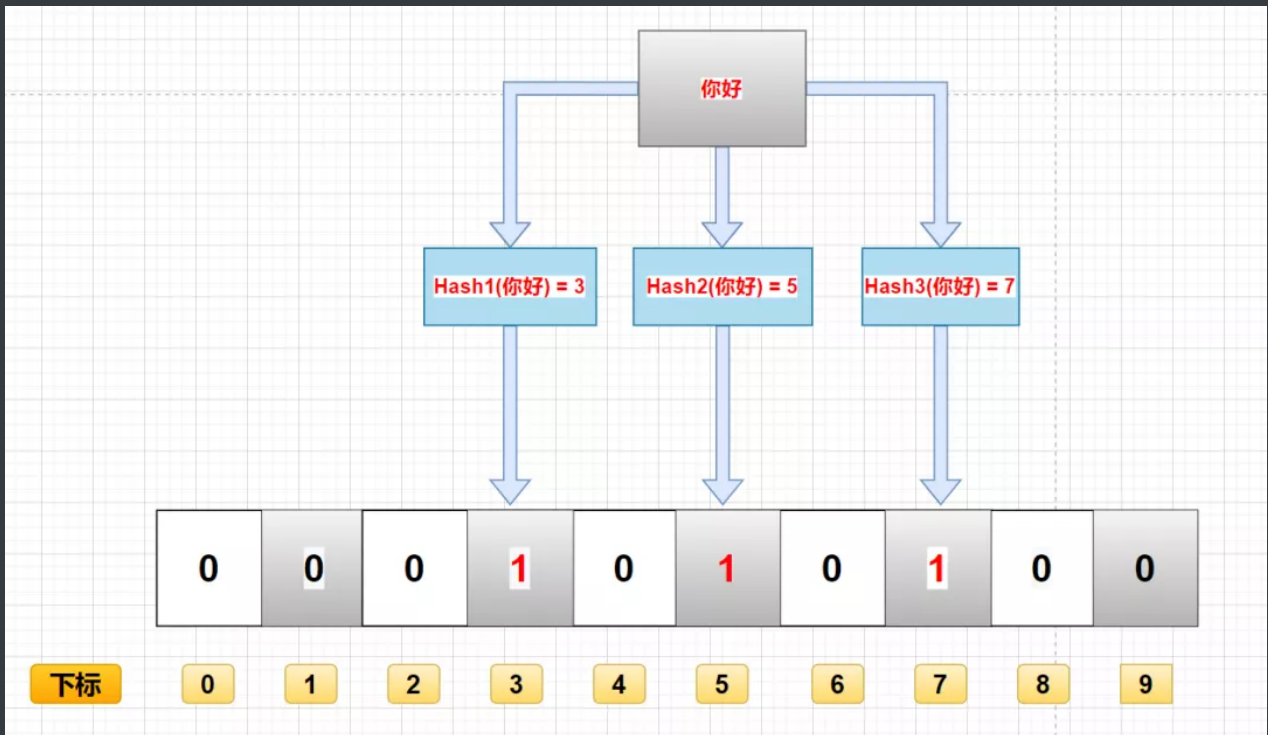
布隆过滤器原理

存入数据过程

布隆过滤器上面说了，就是一个二进制数据的集合。当一个数据加入这个集合时，经历如下洗礼（这里有缺点，下面会讲）：

- 通过K个哈希函数计算该数据，返回K个计算出的hash值
- 这些K个hash值映射到对应的K个二进制的数组下标
- 将K个下标对应的二进制数据改成1。

例如，第一个哈希函数返回x，第二个第三个哈希函数返回y与z，那么：X、Y、Z对应的二进制改成1。



查询数据过程

布隆过滤器主要作用就是查询一个数据，在不在这个二进制的集合中，查询过程如下：

- 通过K个哈希函数计算该数据，对应计算出的K个hash值
- 通过hash值找到对应的二进制的数组下标
- 判断：如果存在一处位置的二进制数据是0，那么该数据不存在。如果都是1，该数据存在集合中。（这里有缺点，下面会讲）

删除数据过程

一般不能删除布隆过滤器里的数据，这是一个缺点之一，我们下面会分析。

布隆过滤器的优缺点

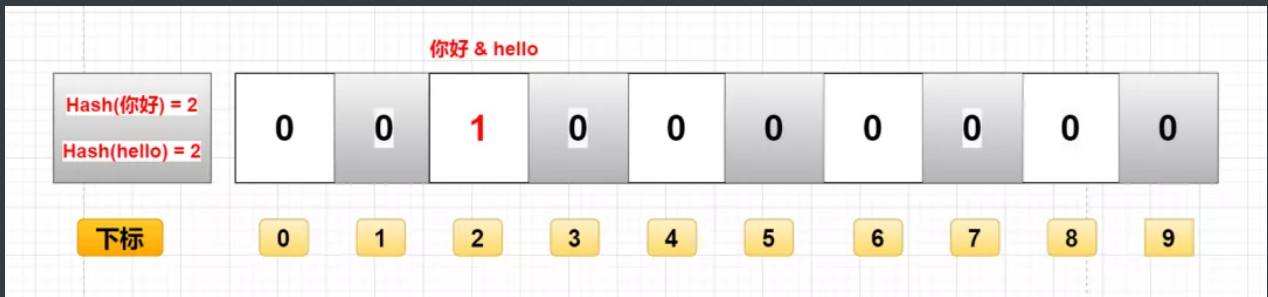
优点

- 由于存储的是二进制数据，所以占用的空间很小
- 它的插入和查询速度是非常快的，时间复杂度是 $O(K)$ ，可以联想一下HashMap的过程
- 保密性很好，因为本身不存储任何原始数据，只有二进制数据

缺点

这就要回到我们上面所说的那些缺点了。

添加数据是通过计算数据的hash值，那么很有可能存在这种情况：两个不同的数据计算得到相同的hash值。



例如图中的“你好”和“hello”，假如最终算出hash值相同，那么他们会将同一个下标的二进制数据改为1。

这个时候，你就不知道下标为2的二进制，到底是代表“你好”还是“hello”。

由此得出如下缺点：

一、存在误判

假如上面的图没有存"hello"，只存了"你好"，那么用"hello"来查询的时候，会判断"hello"存在集合中。

二、删除困难

还是用上面的举例，因为“你好”和“hello”的hash值相同，对应的数组下标也是一样的。

这时候老哥想去删除“你好”，将下标为2里的二进制数据，由1改成了0。

那么我们是不是连“hello”都一起删了呀。（0代表有这个数据，1代表没有这个数据）

实现布隆过滤器方式

有很多种实现方式，其中一种就是Guava提供的实现方式。

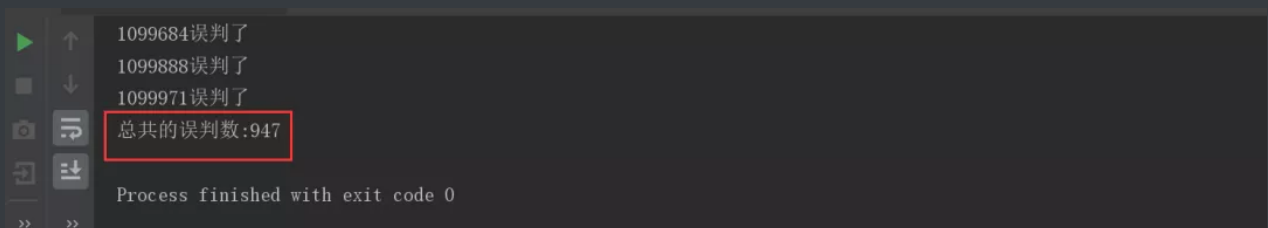
```
1 import com.google.common.hash.BloomFilter;
2 import com.google.common.hash.Funnels;
3
4 public class BloomFilterCase {
5
6     /**
7      * 预计要插入多少数据
8      */
9     private static int size = 1000000;
10
11     /**
12      * 期望的误判率
13      */
14     private static double fpp = 0.01;
```

```

15
16  /**
17   * 布隆过滤器
18   */
19   private static BloomFilter<Integer> bloomFilter =
BloomFilter.create(Funnels.integerFunnel(), size, fpp);
20
21
22   public static void main(String[] args) {
23       // 插入10万样本数据
24       for (int i = 0; i < size; i++) {
25           bloomFilter.put(i);
26       }
27
28       // 用另外十万测试数据，测试误判率
29       int count = 0;
30       for (int i = size; i < size + 100000; i++) {
31           if (bloomFilter.mightContain(i)) {
32               count++;
33               System.out.println(i + "误判了");
34           }
35       }
36       System.out.println("总共的误判数:" + count);
37   }
38   }

```

运行结果：



```

1099684误判了
1099888误判了
1099971误判了
总共的误判数:947
Process finished with exit code 0

```

10万数据里有947个误判，约等于0.01%，也就是我们代码里设置的误判率： $fpp = 0.01$ 。

布隆过滤器深入分析代码

核心BloomFilter.create方法

```

1  @VisibleForTesting
2      static <T> BloomFilter<T> create(
3          Funnel<? super T> funnel, long expectedInsertions, double fpp,
4          Strategy strategy) {
5      }

```

这里有四个参数：

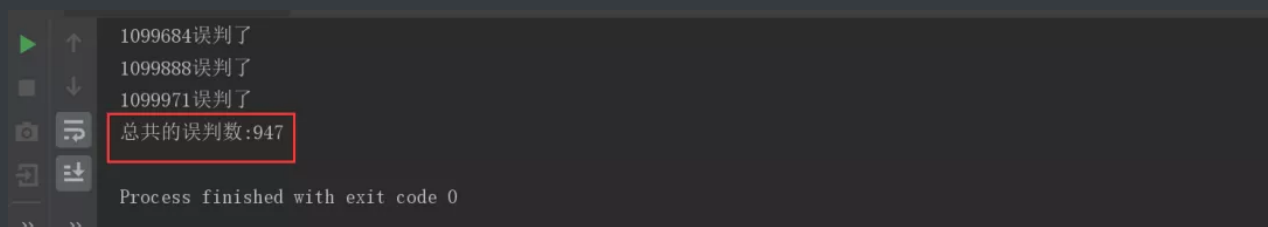
- funnel：数据类型(一般是调用Funnels工具类中的)
- expectedInsertions：期望插入的值的个数
- fpp：误判率(默认值为0.03)
- strategy：哈希算法

我们重点讲一下fpp参数

fpp误判率

情景一：fpp = 0.01

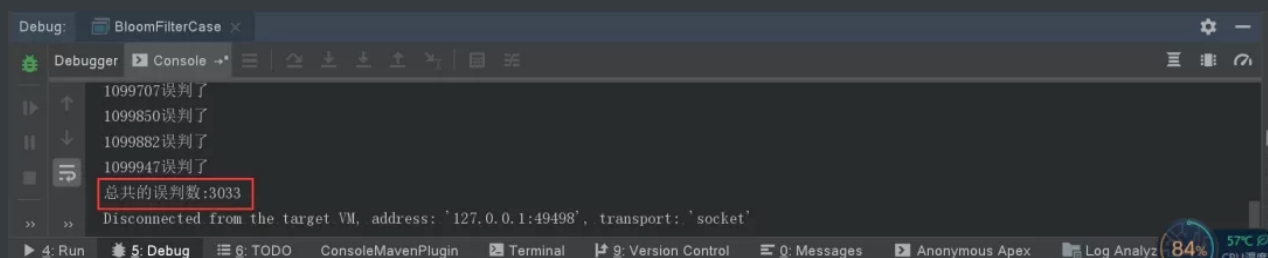
误判个数：947



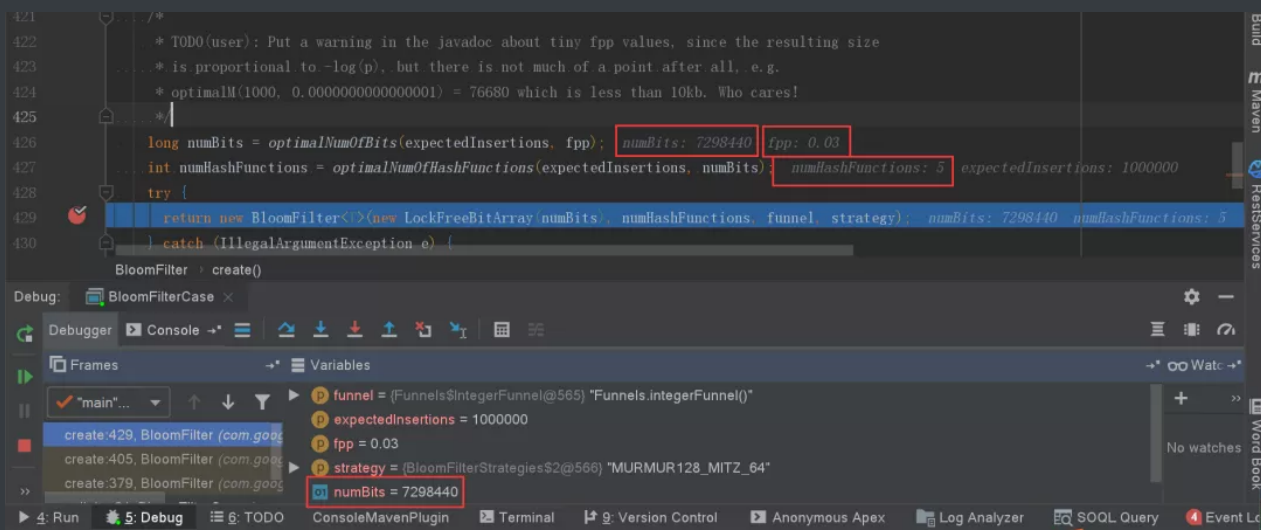
占内存大小：9585058位数

情景二：fpp = 0.03（默认参数）

误判个数：3033



占内存大小：7298440位数



情景总结

- 误判率可以通过fpp参数进行调节
- fpp越小，需要的内存空间就越大：0.01需要900多万位数，0.03需要700多万位数。
- fpp越小，集合添加数据时，就需要更多的hash函数运算更多的hash值，去存储到对应的数组下标里。（忘了去看上面的布隆过滤存入数据的过程）

上面的numBits，表示存一百万个int类型数字，需要的位数为7298440，700多万位。理论上存一百万个数，一个int是4字节32位，需要481000000=3200万位。

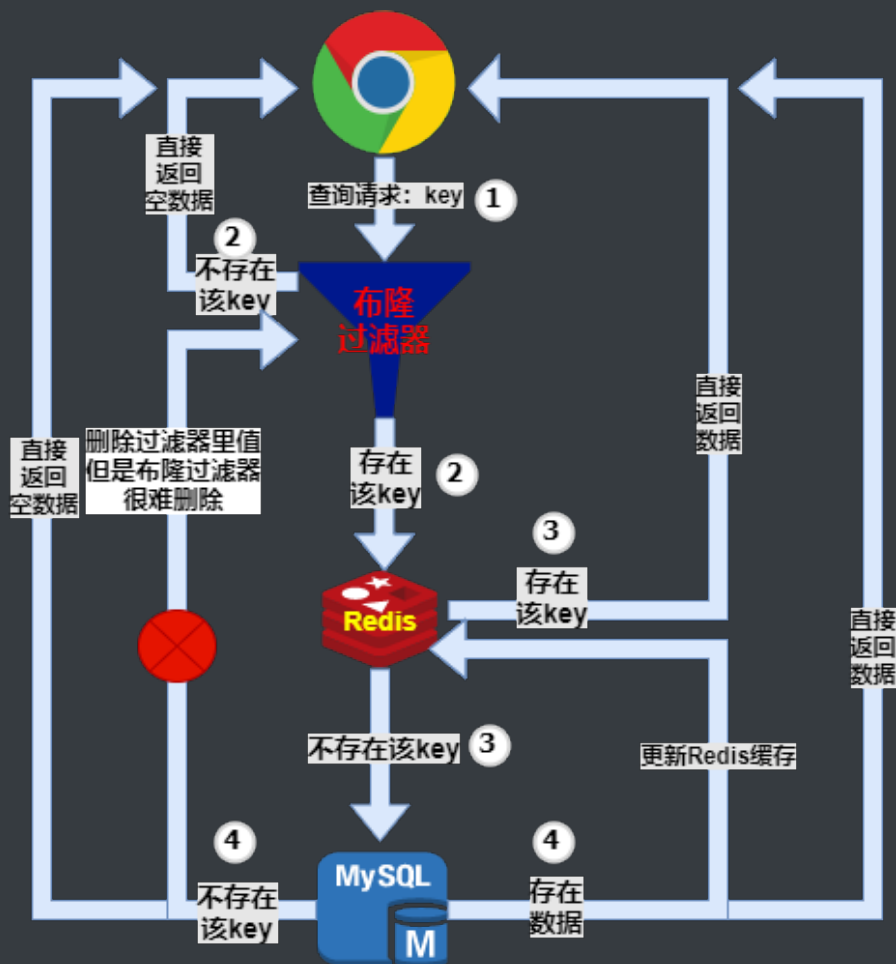
如果使用HashMap去存，按HashMap50%的存储效率，需要6400万位。可以看出BloomFilter的存储空间很小，只有HashMap的1/10左右

上面的numHashFunctions表示需要几个hash函数运算，去映射不同的下标存这些数字是否存在（0 or 1）。

布隆过滤器解决缓存穿透

其实布隆过滤器本质来讲，就是起到一个黑名单或者白名单的作用。我们从这两个角度去分析缓存穿透问题。

白名单解决缓存穿透



白名单

存在的问题:

布隆过滤器里的数据, 存在误判, 如果没在白名单里的数据被误判存在于过滤器里的话, 会穿透到数据库, 不过误判的几率本来就很很小, 所以穿透问题不大

注意的问题:

必须把所有的key都放到布隆过滤器和Redis里, 否则请求会被直接返回空数据

注意问题

- 如果没在白名单里的数据被误判存在于过滤器里的话, 会穿透到数据库, 不过误判的几率本来就很很小, 所以穿透问题不大。
- 必须把所有的查询key都放到布隆过滤器和Redis里, 否则请求会被直接返回空数据。

代码实现

```
1  import com.alibaba.fastjson.JSON;
2  import com.bilibili.itlaoge.model.User;
3  import org.redisson.Redisson;
4  import org.redisson.api.RBloomFilter;
5  import org.redisson.api.RBucket;
6  import org.redisson.api.RedissonClient;
7  import org.redisson.config.Config;
8
9  /**
10   * 解决缓存穿透-白名单
11   */
12  public class RedissonBloomFilter {
13
14      /**
```

```
15     * 构造Redisson
16     */
17     static RedissonClient redisson = null;
18
19     static RBloomFilter<String> bloomFilter = null;
20
21     static {
22         Config config = new Config();
23         config.useSingleServer().setAddress("redis://127.0.0.1:6379");
24
25         //构造Redisson
26         redisson = Redisson.create(config);
27         //构造布隆过滤器
28         bloomFilter = redisson.getBloomFilter("userIdFilter");
29
30         // 将查询数据放入Redis缓存和布隆过滤器里
31         initData(redisson, bloomFilter);
32     }
33
34     private static void initData(RedissonClient redisson,
35     RBloomFilter<String> bloomFilter) {
36
37         //初始化布隆过滤器：预计元素为100000000L,误差率为3%
38         bloomFilter.tryInit(100000000L,0.01);
39
40         //将id为1的数据，插入到布隆过滤器中
41         bloomFilter.add("1");
42         bloomFilter.add("2");
43
44         // 将id为1对应的user数据，插入到Redis缓存中
45         redisson.getBucket("1").set("{id:1, userName:'张三', age:18}");
46     }
47
48     public static void main(String[] args) {
49
50         User user = getUserById(2L);
51         System.out.println("user对象为: " + JSON.toJSONString(user));
52     }
53
54     public static User getUserById(Long id) {
55
56         if (null == id) {
57             return null;
58         }
59
60         String idKey = id.toString();
```

```

59
60     // 开始模拟缓存穿透
61     // 前端查询请求key
62     if (bloomFilter.contains(idKey)) {
63
64         // 通过了过滤器白名单校验，去Redis里查询真正的数据
65         RBucket<Object> bucket = redisson.getBucket(idKey);
66         Object object = bucket.get();
67
68         // 如果Redis有数据，直接返回该数据
69         if (null != object) {
70             System.out.println("从Redis里面查询出来的");
71             String userStr = object.toString();
72             return JSON.parseObject(userStr, User.class);
73         }
74
75         // 如果Redis为空，去查询数据库
76         User user = selectByDb(idKey);
77         if (null == user) {
78             return null;
79         } else {
80             // 将数据重新刷进缓存
81
82             redisson.getBucket(id.toString()).set(JSON.toJSONString(user));
83             return user;
84         }
85
86         return null;
87     }
88
89     private static User selectByDb(String id) {
90         System.out.println("从MySQL里面查询出来的");
91         User user = new User();
92         user.setId(1L);
93         user.setUserName("张三");
94         user.setAge(18);
95         return user;
96     }
97
98 }

```

```

1  /**
2   * 用户实体类

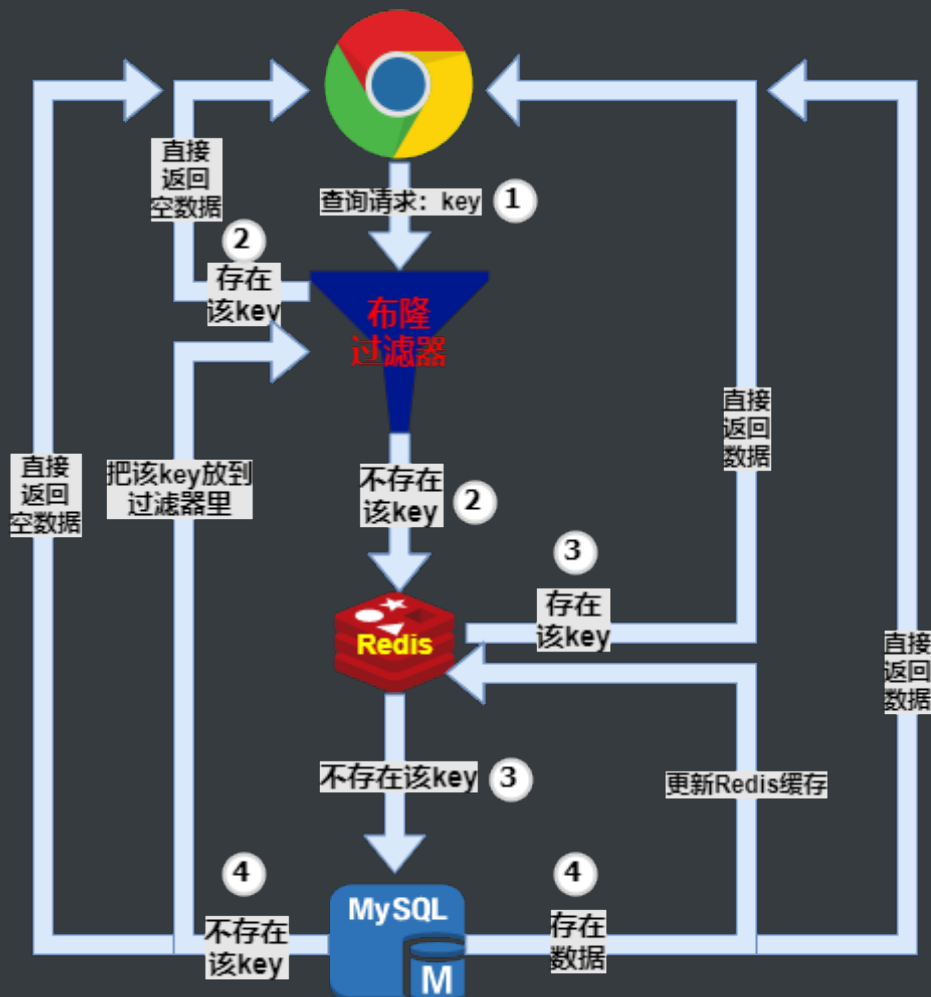
```

```

3  * @author hp
4  */
5  public class User implements Serializable {
6
7      public static String maYunPhone = "18890019390";
8
9      private Long id;
10
11     /**
12      * 用户名
13      */
14     private String userName;
15
16     /**
17      * 年龄
18      */
19     private Integer age;
20 }

```

黑名单解决缓存穿透



黑名单

存在的问题:

布隆过滤器里的数据, 存在误判, 如果正常数据被误判存在黑名单里的话, 会直接返回空数据

注意的问题:

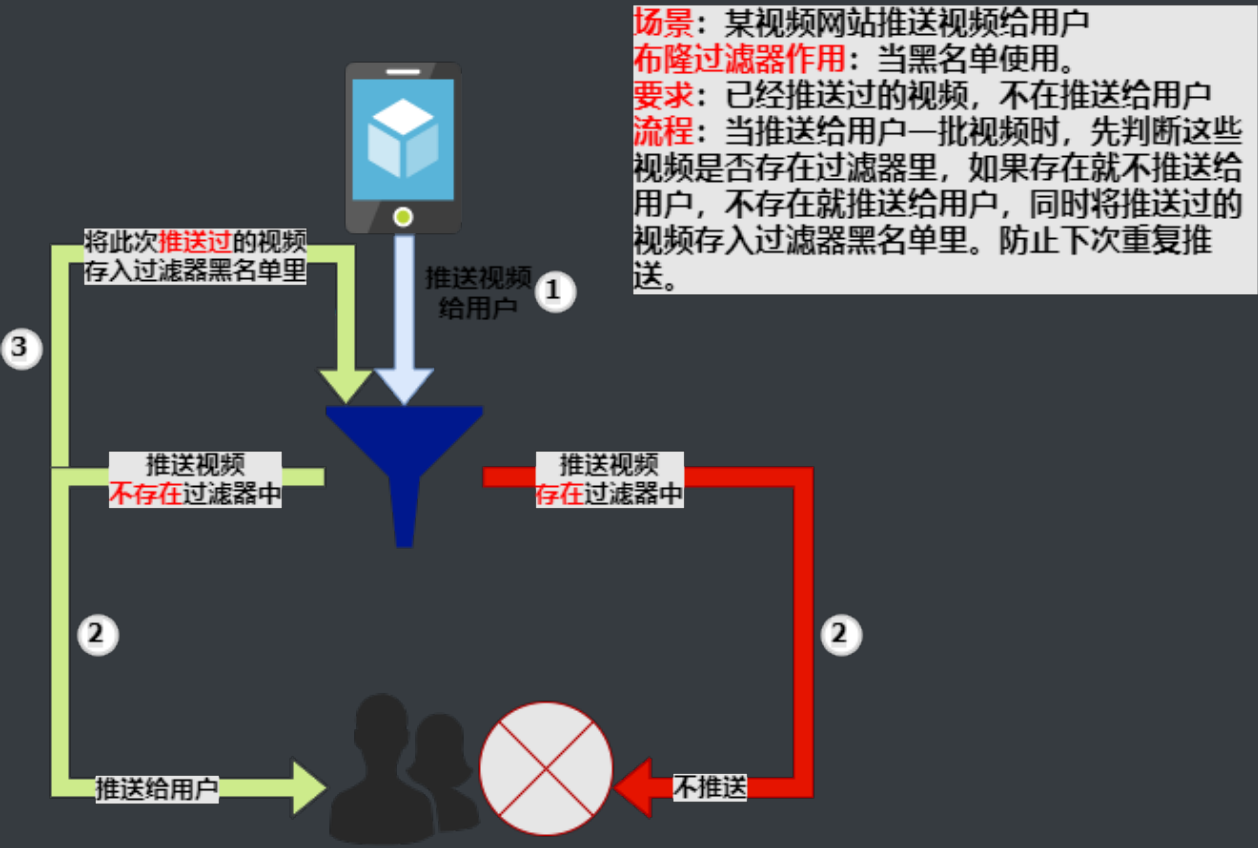
黑名单里的数据要很全面才行, 否则会有穿透问题

注意问题

- 布隆过滤器里的数据，存在误判，如果正常数据被误判存在黑名单里的话，会直接返回空数据。
- 黑名单里的数据要很全面才行，否则会有比较严重的穿透问题。
- 本来是在黑名单里的非法数据，之后有可能是正常数据。如：用id大于100万的数来请求，我们数据库里只有10万数据，这时候如果把id放进黑名单里。等数据达到100万的时候，就会出现问題。

布隆过滤器其他应用场景举例

视频推送场景（黑名单）



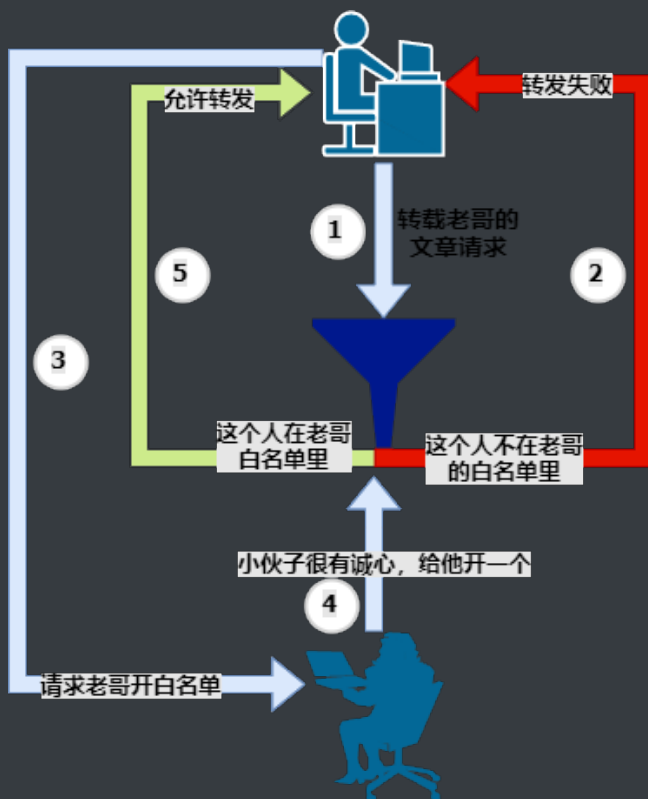
背景：某视频网站给用户推送视频

布隆过滤器作用：当黑名单使用。

要求：对于某用户，已经推送过的视频，不在进行推送。

流程：当推送给用户一批视频时，先判断这些视频是否存在过滤器里；如果存在就不推送给用户，不存在就推送给用户；同时将推送过的视频存入过滤器黑名单里，防止下次重复推送。

转载视频/文章案例（白名单）



场景：某用户想转载老哥的文章
布隆过滤器作用：当白名单使用。
要求：在老哥转发白名单里的，有转发文章的权限
流程：某用户想转发老哥的文章，由于没在白名单里，转发失败。于是找到老哥开白名单，老哥把他加入了白名单里后，允许转发了。

背景：某用户想转载老哥的文章。

布隆过滤器作用：当白名单使用。

要求：在老哥转发白名单里的，有转发文章的权限。

流程：某用户想转发老哥的文章，由于没在白名单里，转发失败。于是找到老哥开白名单，老哥把他加入了白名单里后，允许转发了。