Redis

[1. 问题 1](#_116)

[1.1. 缓存穿透 1](#_117)

[1.2. 缓存击穿 1](#_118)

[1.3. 缓存雪崩 1](#_119)

[2. 发布/订阅 1](#_112)

[3. 哨兵 1](#_113)

[3.1. 主观下线/客观下线 2](#_114)

[3.2. major选举，主备切换 2](#_115)

[4. 数据类型 2](#_101)

[4.1. 普通数据类型 2](#_102)

[4.2. 特殊数据类型 2](#_103)

[5. 与Memcached的区别 3](#_104)

[6. 持久化 3](#_105)

[6.1. rdb 3](#_106)

[6.2. aof 3](#_107)

[6.3. 企业备份 4](#_125)

[7. 缓存一致性 4](#_126)

[7.1. Cache Aside Pattern 5](#_127)

[7.2. 更新时的步骤 5](#_128)

[7.2.1. 先更新缓存，再更新数据库 5](#_129)

[7.2.2. 先更新数据库，再更新缓存 5](#_130)

[7.2.3. 先删除缓存，再更新数据库 6](#_131)

[7.2.4. 先更新数据库，再删除缓存 6](#_132)

[8. 淘汰策略 7](#_108)

[9. 事务 7](#_109)

[10. 事件驱动模型 7](#_110)

[11. 主从复制 8](#_111)

[12. 使用场景 8](#_120)

[12.1. LRU缓存 8](#_121)

[12.2. 消息队列 8](#_122)

[12.3. 分布式锁 8](#_123)

[12.4. 会话缓存 9](#_124)

# 问题

## 缓存穿透

用户查询一个数据，发现Redis中没有，也就是缓存不命中，于是向持久层数据库查询。发现也没有，于是本次查询失败。当用户很多时，缓存都没有命中，于是请求了持久层数据库，会给持久层数据库造成很大压力。在流量大时，可能DB就挂掉了，要是有人利用不存在的key频繁攻击我们的应用，这就是漏洞。

解决方法：

布隆过滤器

缓存空对象

## 缓存击穿

对于一些设置了过期时间的key，如果这些key可能会在某些时间点被超高并发地访问，是一种非常“热点”的数据。这个时候，需要考虑一个问题：缓存被“击穿”的问题，这个和缓存雪崩的区别在于这里针对某一key缓存，后者则是很多key。

缓存在某个时间点过期的时候，恰好在这个时间点对这个Key有大量的并发请求过来，这些请求发现缓存过期一般都会从后端DB加载数据并回设到缓存，这个时候大并发的请求可能会瞬间把后端DB压垮。

解决方法：

分布式锁

热点数据永不过期

## 缓存雪崩

缓存雪崩是指在我们设置缓存时采用了相同的过期时间，导致缓存在某一时刻同时失效，请求全部转发到DB，DB瞬时压力过重雪崩

处理方法：这里分享一个简单方案就时讲缓存失效时间分散开，比如我们可以在原有的失效时间基础上增加一个随机值，比如1-5分钟随机，这样每一个缓存的过期时间的重复率就会降低，就很难引发集体失效的事件。

# 发布/订阅

subscribe和publish

# 哨兵

## 主观下线/客观下线

## major选举，主备切换

# 数据类型

## 普通数据类型

共有5类普通数据类型：

String

Hash：使用双hash表

List

Set

Zset：底层使用跳表。跳表底层是双向链表，通过多层索引来加快查找速度，并且在插入时随机决定创建几层索引，其缺点在于内存消耗高（因为要创建额外的指针）。

跳表的示意图

￼

查找过程

￼

zset使用跳表而不使用红黑树

1. skiplist的复杂度和红黑树一样，而且实现起来更简单。

2. 在并发环境下skiplist有另外一个优势，红黑树在插入和删除的时候可能需要做一些rebalance的操作，这样的操作可能会涉及到整个树的其他部分，而skiplist的操作显然更加局部性一些，锁需要盯住的节点更少，因此在这样的情况下性能好一些

## 特殊数据类型

共有3种特殊数据类型：

Geospatial：地理信息

HyperLoglog：在loglog算法基础上改进（使用调和平均数）

BitMap：位图

HyperLoglog实际上是基于伯努利试验。伯努利试验实际上来源于抛硬币，对于一次抛硬币，我们都知道最终出现正反面的概率都是50%。假设一直抛硬币，直到它出现正面为止，这是一次完整的试验。假设进行了n次的伯努利试验，第一次伯努利试验出现正面经历的抛掷次数为k1，第n次试验的抛掷次数则是kn。这k1到kn中显然有一个kmax。

通过结合极大似然估计的方法，发现n和kmax之间存在这样的关系：n=2^(kmax)

最后得到以下公式：

￼

# 与Memcached的区别

Redis支持持久化，Memcached不支持

Memcached仅支持字符串，Redis支持更多数据类型

Memcached不支持分布式，Redis使用哨兵可形成可靠集群

Redis可以将很久没用的value交换到磁盘，而Memcached会一直存在内存

Memcached将内存分割成特定长度的块来存储数据，但这种方法内存利用率不高

# 持久化

## rdb

简而言之，就是在不同的时间点，将redis存储的数据生成快照并存储到磁盘等介质上。

触发时机（会生成dump.rdb）：

save规则，手动save（save，bgsave）

退出redis

恢复方法：在包含有dump.rdb目录下启动redis，即可恢复。

RDB文件在生成时会检查每个键的过期时间，过期键不会被添加进RDB文件里。

## aof

则是换了一个角度来实现持久化，那就是将redis执行过的所有写指令记录下来，在下次redis重新启动时，只要把这些写指令从前到后再重复执行一遍，就可以实现数据恢复了。AOF恢复程度更高，但是恢复速度更慢。

如果appendonly.aof中有错误，可以使用redis-check-aof进行修复，会删除产生错误的那次操作。

AOF重写：重新生成一个AOF文件替换原有的AOF文件，这里的重写不会对原有的文件进行读取、分析或写入，而是把数据库中的键值对折算成命令，重写写入文件。可以通过bgrewriteaof命令在后台重写，重写时Redis会维护一个AOF重写缓冲区（记录在重写期间收到的写命令），在重写完成后追加都AOF文件末尾。

对于过期键，当它被惰性删除或定期删除之后，程序会向AOF文件追加一条DEL命令显示记录该键已被删除。

## 企业备份

rdb备份：

写crontab定时调度脚本去做数据备份。

每小时都copy一份redis的rdb文件到一个其他目录中，这个目录里的rdb文件仅仅保留48小时内的。也就是每小时都做备份，保留2天内的rdb，只保留48个rdb。

每天0点0分copy一份redis的rdb文件到一个其他目录中，这个保留一个月的。也就是按天备份。

每天半夜找个时间将当前服务上的所有rdb备份都上传到云服务上。

恢复方案：

停止redis进程

删除坏掉的rdb和aof持久化文件。

修改配置文件关闭redis的aof持久化。

找到最新备份的rdb文件扔到redis的持久化目录里。（这里最新的肯定是按照小时备份的最后一个）

启动Redis进程

执行config set appendonly yes动态打开aof持久化。

也就是说打开aof的操作不是修改配置文件然后重启，而是先热修改让他生成aof，这次生成肯定是会带着内存中完整的数据的。

等aof文件生成后再修改redis配置文件打开aof。

重启redis进程。

完美收官。

# 缓存一致性

有两个前提：

1. 缓存必须要有过期时间

2. 保证数据库与缓存的最终一致性即可，不必追求强一致性。

为什么必须要有过期时间？

总结来讲就是：通过设置过期时间，使得热点数据被缓存，而非热点数据随时间被淘汰；另外过期时间的限制，使得数据库和缓存的不一致仅持续一定时间。

首先对于缓存来说，当它的命中率越高的时候，我们的系统性能也就越好。如果某个缓存项没有过期时间，而它命中的概率又很低，这就是在浪费缓存的空间。而如果有了过期时间，且在某个缓存项经常被命中的情况下，我们可以在每次命中的时候都刷新一下它的过期时间，这样也就保证了热点数据会一直在缓存中存在，从而保证了缓存的命中率，提高了系统的性能。

设置过期时间还有一个好处，就是当数据库跟缓存出现数据不一致的情况时，这个可以作为一个最后的兜底手段。也就是说，当数据确实出现不一致的情况时，过期时间可以保证只有在出现不一致的时间点到缓存过期这段时间之内，数据库跟缓存的数据是不一致的，因此也保证了数据的最终一致性。

## Cache Aside Pattern

失效：程序先从缓存中读取数据，如果没有命中，则从数据库中读取，成功之后将数据放到缓存中

命中：程序先从缓存中读取数据，如果命中，则直接返回

更新：程序先更新数据库，在删除缓存

## 更新时的步骤

### 先更新缓存，再更新数据库

我们都知道不管是操作数据库还是操作缓存，都有失败的可能。如果我们先更新缓存，再更新数据库，假设更新数据库失败了，那数据库中就存的是老数据。当然你可以选择重试更新数据库，那么再极端点，负责更新数据库的机器也宕机了，那么数据库中的数据将一直得不到更新，并且当缓存失效之后，其他机器再从数据库中读到的数据是老数据，然后再放到缓存中，这就导致先前的更新操作被丢失了，因此这么做的隐患是很大的。

从数据持久化的角度来说，数据库当然要比缓存做的好，我们也应当以数据库中的数据为主，所以需要更新数据的时候我们应当首先更新数据库，而不是缓存。

### 先更新数据库，再更新缓存

这里主要有两个问题，首先是并发的问题：假设线程A（或者机器A，道理是一样的）和线程B需要更新同一个数据，A先于B但时间间隔很短，那么就有可能会出现：

线程A更新了数据库

线程B更新了数据库

线程B更新了缓存

线程A更新了缓存

按理说线程B应该最后更新缓存，但是可能因为网络等原因，导致线程B先于线程A对缓存进行了更新，这就导致缓存中的数据不是最新的。

第二个问题是，我们不确定要更新的这个缓存项是否会被经常读取，假设每次更新数据库都会导致缓存的更新，有可能数据还没有被读取过就已经再次更新了，这就造成了缓存空间的浪费。另外，缓存中的值可能是经过一系列计算的，而并不是直接跟数据库中的数据对应的，频繁更新缓存会导致大量无效的计算，造成机器性能的浪费。

综上所述，更新缓存这一方案是不可取的，我们应当考虑删除缓存。

### 先删除缓存，再更新数据库

这个方案的问题也是很明显的，假设现在有两个请求，一个是写请求A，一个是读请求B，那么可能出现如下的执行序列：

请求A删除缓存

请求B读取缓存，发现不存在，从数据库中读取到旧值

请求A将新值写入数据库

请求B将旧值写入缓存

这样就会导致缓存中存的还是旧值，在缓存过期之前都无法读到新值。这个问题在数据库读写分离的情况下会更明显，因为主从同步需要时间，请求B获取到的数据很可能还是旧值，那么写入缓存中的也会是旧值。

### 先更新数据库，再删除缓存

终于来到我们最常用的方案了，但是最常用并不是说就一定不会有任何问题，我们依然假设有两个请求，请求A是查询请求，请求B是更新请求，那么可能会出现下述情形：

先前缓存刚好失效

请求A查数据库，得到旧值

请求B更新数据库

请求B删除缓存

请求A将旧值写入缓存

上述情况确实有可能出现，但是出现的概率可能不高，因为上述情形成立的条件是在读取数据时，缓存刚好失效，并且此时正好又有一个并发的写请求。考虑到数据库上的写操作一般都会比读操作要慢，（这里指的是在写数据库时，数据库一般都会上锁，而普通的查询语句是不会上锁的。当然，复杂的查询语句除外，但是这种语句的占比不会太高）并且联系常见的数据库读写分离的架构，可以合理认为在现实生活中，读请求的比例要远高于写请求，因此我们可以得出结论。这种情况下缓存中存在脏数据的可能性是不高的。

# 淘汰策略

当内存使用量达到最大时，进行数据淘汰策略。共分为以下几种：

lru是最近最久未使用，lfu（从4.0开始）是最近最不经常使用，volatile是针对设置了有效期的key。

volatile-lru

volatile-ttl

volatile-random

volatile-lfu

allkeys-lru

allkeys-random

allkeys-lfu

noeviction：不对数据进行淘汰，只是报错。

驱逐过程如下：

客户端运行新命令，从而添加更多数据。

Redis会检查内存使用情况，如果大于使用maxmemory限制，则会根据策略逐出密钥。

执行新命令，依此类推。

# 事务

通过multi开启事务，通过exec执行或discard丢弃事务。

事务会受到异常的影响：

编译型异常：如命令打错等，如果出现该类异常，事务所有命令都不能执行。

运行时异常：运行时才会发生的异常，如对非法字符串incr。出现该类异常，事务其他命令可以正常执行。

# 事件驱动模型

Redis服务器通过socket实现与客户端（或其他Redis服务器）的交互，文件事件就是服务器对socket操作的抽象。Redis通过监听这些socket产生的文件事件并处理这些事件，实现对客户端调用的响应。

文件事件处理器的类型

连接应答处理器：处理连接相关的事件

命令请求处理器：负责读取通过socket发送来的命令

命令回复处理器：当Redis处理完命令后，会产生AW\_WRITEABLE事件，将数据回复给client

时间事件：

定时任务

周期任务（如定期删除）

服务器将时间事件都放在一个无序链表中（不是按时间顺序排序，而是按照ID排序，新产生的时间事件放在链表的表头），每次时间事件执行器运行时，它就遍历整个链表，查找所有已经到达的时间事件，并调用相应的事件处理器。

定期删除是另一种过期键删除方式。Redis 会维护一个过期字典（如下图所示），所有声明了过期时间的键都会被添加进这个字典中。周期操作函数 serverCron 执行时，会在规定时间内随机检查一部分键的过期时间，并删除其中的过期键。

# 主从复制

master创建快照，发送给slave，同时记录执行的写命令。

快照发送结束，发送缓冲区的写命令

从服务器丢弃旧数据，载入master发来的快照数据，然后接收master发来的写命令

master每执行一次写命令，就向从服务器发送相同的写命令

# 使用场景

## LRU缓存

# 设置最大内存使用量

maxmemory 100mb

# 设置驱除策略

maxmemory-policy volatile-lru/allkeys-lru

# 设置采样点

maxmemory-sample 10

## 消息队列

使用subscribe channel和publish channel msg来构建轻量级消息队列。

## 分布式锁

单Redis实例：

获取锁：set resource\_name PX 30000 NX

释放锁：使用lua脚本

if redis.call("get", keys[1]) == ARGV[1] then

return redis.call("del", keys[1])

else

return 0

end

RedLock算法：

假设有5个Redis master节点。

获取锁：

获取当前unix时间。

依次尝试从N个实例，使用相同的key和随机值获取锁。

客户端使用当前时间减去开始获取锁时间（步骤1记录的时间）就得到锁已经使用的时间。仅当大多数（这里是3个节点）的Redis节点都获取到锁，并且使用的时间小于锁失效时间时，才算获得锁成功。

如果获取到锁，key的真正有效时间等于有效时间 - 获取锁花费的时间。

如果某些原因，获取锁失败，客户端应该在所有Redis实例上解锁，并在随机延迟后重试（防止脑裂）

释放锁：直接向所有Redis发送释放锁命令即可。

## 会话缓存