

Redis

Redis为什么用跳表而不用红黑树

Redis是用跳表来实现有序集合。

Redis 中的有序集合支持的核心操作主要有下面这几个：插入一个数据；删除一个数据；查找一个数据；按照区间查找数据（比如查找值在 [100, 356] 之间的数据）；迭代输出有序序列。

其中，插入、删除、查找以及迭代输出有序序列这几个操作，红黑树也可以完成，时间复杂度跟跳表是一样的。但是，按照区间来查找数据这个操作，红黑树的效率没有跳表高。

Redis持久化机制

很多时候我们需要持久化数据也就是将内存中的数据写入到硬盘里面，大部分原因是为了之后重用数据（比如重启机器、机器故障之后恢复数据），或者是为了防止系统故障而将数据备份到一个远程位置。

Redis不同于Memcached的很重要一点就是，Redis支持持久化，而且支持两种不同的持久化操作。**Redis的一种持久化方式叫快照（snapshotting, RDB），另一种方式是只追加文件（append-only file,AOF）。**

快照（snapshotting）持久化（RDB）

Redis可以通过创建快照来获得存储在内存里面的数据在某个时间点上的副本。Redis创建快照之后，可以对快照进行备份，可以将快照复制到其他服务器从而创建具有相同数据的服务器副本（Redis主从结构，主要用来提高Redis性能），还可以将快照留在原地以便重启服务器的时候使用。

快照持久化是Redis默认采用的持久化方式，在redis.conf配置文件中默认有此下配置：

```
save 900 1      #在900秒(15分钟)之后，如果至少有1个key发生变化，Redis就会自动触发
                 BGSAVE命令创建快照。
save 300 10     #在300秒(5分钟)之后，如果至少有10个key发生变化，Redis就会自动触发
                 BGSAVE命令创建快照。
save 60 10000   #在60秒(1分钟)之后，如果至少有10000个key发生变化，Redis就会自动触发
                 BGSAVE命令创建快照。
```

优点：全量数据快照，文件小，恢复快

缺点：无法保存最近一次快照之后的数据

AOF（append-only file）持久化

与快照持久化相比，AOF持久化 的实时性更好，因此已成为主流的持久化方案。默认情况下Redis没有开启AOF（append only file）方式的持久化，可以通过appendonly参数开启：

```
appendonly yes
```

开启AOF持久化后每执行一条会更改Redis中的数据的命令，Redis就会将该命令写入硬盘中的AOF文件。AOF文件的保存位置和RDB文件的位置相同，都是通过dir参数设置的，默认的文件名是appendonly.aof。

在Redis的配置文件中存在三种不同的 AOF 持久化方式，它们分别是：

```
appendfsync always    #每次有数据修改发生时都会写入AOF文件,这样会严重降低Redis的速度
appendfsync everysec  #每秒钟同步一次,显示地将多个写命令同步到硬盘
appendfsync no        #让操作系统决定何时进行同步
```

为了兼顾数据和写入性能,用户可以考虑 `appendfsync everysec`选项,让Redis每秒同步一次AOF文件,Redis性能几乎没受到任何影响。而且这样即使出现系统崩溃,用户最多只会丢失一秒之内产生的数据。当硬盘忙于执行写入操作的时候,Redis还会优雅的放慢自己的速度以便适应硬盘的最大写入速度。

优点: 可读性高, 适合保存增量数据, 数据不易丢失

缺点: 文件体积较大, 恢复时间长

Redis 4.0 对于持久化机制的优化

Redis 4.0 开始支持 RDB 和 AOF 的混合持久化(默认关闭, 可以通过配置项 `aof-use-rdb-preamble` 开启)。

如果把混合持久化打开, AOF 重写的时候就直接把 RDB 的内容写到 AOF 文件开头。这样做的好处是可以结合 RDB 和 AOF 的优点, 快速加载同时避免丢失过多的数据。当然缺点也是有的, AOF 里面的 RDB 部分是压缩格式不再是 AOF 格式, 可读性较差。

BGSAVE做镜像全量持久化, AOF做增量持久啊。

数据淘汰策略

可以设置内存最大使用量, 当内存使用量超出时, 会施行数据淘汰策略。

Redis 具体有 6 种淘汰策略:

策略	描述
volatile-lru	从已设置过期时间的数据集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中任意选择数据淘汰
allkeys-lru	从所有数据集中挑选最近最少使用的数据淘汰
allkeys-random	从所有数据集中任意选择数据进行淘汰
noeviction	禁止驱逐数据

作为内存数据库, 出于对性能和内存消耗的考虑, Redis 的淘汰算法实际实现上并非针对所有 key, 而是抽样一小部分并且从中选出被淘汰的 key。

使用 Redis 缓存数据时, 为了提高缓存命中率, 需要保证缓存数据都是热点数据。可以将内存最大使用量设置为热点数据占用的内存量, 然后启用 `allkeys-lru` 淘汰策略, 将最近最少使用的数据淘汰。

Redis 4.0 引入了 `volatile-lfu` 和 `allkeys-lfu` 淘汰策略, LFU 策略通过统计访问频率, 将访问频率最少的键值对淘汰。

缓存穿透、缓存击穿、缓存雪崩概念及解决方案

缓存穿透

概念

访问一个不存在的key，缓存不起作用，请求会穿透到DB，流量大时DB会挂掉。

解决方案

1. 采用布隆过滤器，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查询到值，也将空值写进缓存，但可以设置较短过期时间。

缓存雪崩

概念

大量的key设置了相同的过期时间，导致在缓存在同一时刻全部失效，造成瞬时DB请求量大、压力骤增，引起雪崩。

解决方案

可以给缓存设置过期时间时加上一个随机值时间，使得每个key的过期时间分布开来，不会集中在同一时刻失效。

缓存击穿

概念

一个存在的key，在缓存过期的一刻，同时有大量的请求，这些请求都会击穿到DB，造成瞬时DB请求量大、压力骤增。

解决方案

在访问key之前，采用SETNX（set if not exists）来设置另一个短期key来锁住当前key的访问，访问结束再删除该短期key。

Redis为什么那么快

- 1、完全基于内存，绝大部分请求是纯粹的内存操作，非常快速。数据存在内存中，类似于HashMap，HashMap的优势就是查找和操作的时间复杂度都是O(1)；
- 2、数据结构简单，对数据操作也简单，Redis中的数据结构是专门进行设计的；
- 3、采用单线程，避免了不必要的上下文切换和竞争条件，也不存在多进程或者多线程导致的切换而消耗CPU，不用去考虑各种锁的问题，不存在加锁释放锁操作，没有因为可能出现死锁而导致的性能消耗；
- 4、使用多路I/O复用模型，非阻塞IO；
- 5、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直接自己构建了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；

Redis为什么是单线程的

官方FAQ表示，因为Redis是基于内存的操作，CPU不是Redis的瓶颈，Redis的瓶颈最有可能是机器内存的大小或者网络带宽。既然单线程容易实现，而且CPU不会成为瓶颈，那就顺理成章地采用单线程的方案了（毕竟采用多线程会有很多麻烦！）。

redis 的并发竞争问题是什么？如何解决这个问题？了解 redis 事务的 CAS 方案吗？

这个也是线上非常常见的一个问题，就是**多客户端同时并发写**一个 key，可能本来应该先到的数据后到了，导致数据版本错了；或者是多客户端同时获取一个 key，修改值之后再写回去，只要顺序错了，数据就错了。

而且 redis 自己就有天然解决这个问题的 CAS 类的乐观锁方案。

某个时刻，多个系统实例都去更新某个 key。可以基于 zookeeper 实现分布式锁。每个系统通过 zookeeper 获取分布式锁，确保同一时间，只能有一个系统实例在操作某个 key，别人都不允许读和写。

你要写入缓存的数据，都是从 mysql 里查出来的，都得写入 mysql 中，写入 mysql 中的时候必须保存一个时间戳，从 mysql 查出来的时候，时间戳也查出来。

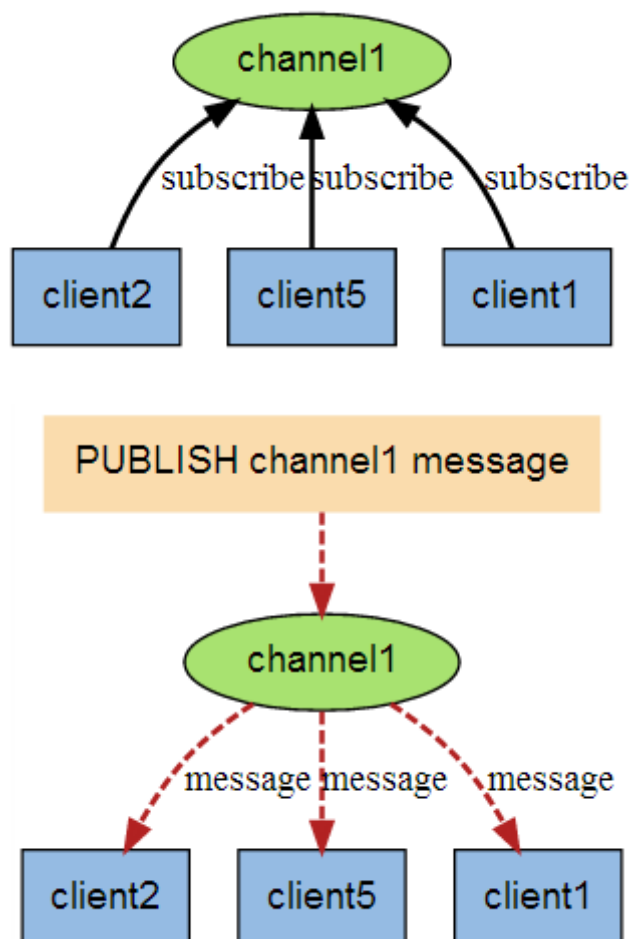
每次要**写之前，先判断**一下当前这个 value 的时间戳是否比缓存里的 value 的时间戳要新。如果是的话，那么可以写，否则，就不能用旧的数据覆盖新的数据。

Redis发布订阅

Redis 发布订阅(pub/sub)是一种消息通信模式：发送者(pub)发送消息，订阅者(sub)接收消息。

Redis 客户端可以订阅任意数量的频道。

下图展示了频道 channel1，以及订阅这个频道的三个客户端—— client2、client5 和 client1 之间的关系：



以下实例演示了发布订阅是如何工作的。在我们实例中我们创建了订阅频道名为 **redisChat**:

```
redis 127.0.0.1:6379> SUBSCRIBE redisChat
```

```
Reading messages... (press Ctrl-C to quit)
```

```
1) "subscribe"  
2) "redisChat"  
3) (integer) 1
```

现在，我们先重新开启个 redis 客户端，然后在同一个频道 redisChat 发布两次消息，订阅者就能接收到消息。

```
redis 127.0.0.1:6379> PUBLISH redisChat "Redis is a great caching technique"
```

```
(integer) 1
```

```
redis 127.0.0.1:6379> PUBLISH redisChat "Learn redis by runoob.com"
```

```
(integer) 1
```

订阅者的客户端会显示如下消息

```
1) "message"  
2) "redisChat"  
3) "Redis is a great caching technique"  
1) "message"  
2) "redisChat"  
3) "Learn redis by runoob.com"
```

消息发布是无状态的，无法保证可达。

Redis主从同步

全同步过程

- master 执行 bgsave，在本地生成一份 rdb 快照文件。
- master node 将 rdb 快照文件发送给 slave node，如果 rdb 复制时间超过 60秒（repl-timeout），那么 slave node 就会认为复制失败，可以适当调大这个参数(对于千兆网卡的机器，一般每秒传输 100MB，6G 文件，很可能超过 60s)
- master node 在生成 rdb 时，会将所有新的写命令缓存在内存中，在 slave node 保存了 rdb 之后，再将新的写命令复制给 slave node。
- 如果在复制期间，内存缓冲区持续消耗超过 64MB，或者一次性超过 256MB，那么停止复制，复制失败。

```
client-output-buffer-limit slave 256MB 64MB 60
```

- slave node 接收到 rdb 之后，清空自己的旧数据，然后重新加载 rdb 到自己的内存中，同时**基于旧的数据版本**对外提供服务。
- 如果 slave node 开启了 AOF，那么会立即执行 BGREWRITEAOF，重写 AOF。

增量同步过程

- 如果全量复制过程中，master-slave 网络连接断掉，那么 slave 重新连接 master 时，会触发增量复制。
- master 直接从自己的 backlog 中获取部分丢失的数据，发送给 slave node，默认 backlog 就是 1MB。

- master 就是根据 slave 发送的 psync 中的 offset 来从 backlog 中获取数据的。

哨兵

哨兵功能

- 集群监控：负责监控 redis master 和 slave 进程是否正常工作。
- 消息通知：如果某个 redis 实例有故障，那么哨兵负责发送消息作为报警通知给管理员。
- 故障转移：如果 master node 挂掉了，会自动转移到 slave node 上。
- 配置中心：如果故障转移发生了，通知 client 客户端新的 master 地址。