Redis 学习

jask

09/18/2024

Redis 作用

缓存

缓存机制几乎在所有的大型网站都有使用，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。Redis 提供了键值过期时间设置，并且也提供了灵活控制最大内存和内存溢出后的淘汰策略。

排行榜

排行榜系统几乎存在于所有的网站，例如按照热度排名的排行榜，按照发布时间的排行榜，按照各种复杂维度计算出的排行榜,Redis 提供了列表和有序集合数据结构，合理地使用这些数据结构可以很方便地构建各种排行榜系统。

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。

数据结构

redis 是单线程的，这一点不能忽略。

string

hash

list

set

有序集合类型的内部编码有两种：

·ziplist(压缩列表)：当有序集合的元素个数小于 zset-max-ziplist- entries 配置（默认 128 个），同时每个元素的值都小于 zset-max-ziplist-value 配置（默认 64 字节）时,Redis 会用 ziplist 来作为有序集合的内部实现,ziplist 可以有效减少内存的使用。

·skiplist(跳跃表)：当 ziplist 条件不满足时，有序集合会使用 skiplist 作为内部实现，因为此时 ziplist 的读写效率会下降。

zset

快照

快照是内存数据的二进制序列化形式，在存储上非常紧凑

在服务线上请求的同时，Redis 还需要进行内存快照，内存快照要求 Redis 必须进行文件 IO 操作，可文件 IO 操作是不能使用多路复用 API。

Redis 使用操作系统的多进程 COW(Copy On Write) 机制来实现快照持久化

Redis 在持久化时会调用 glibc 的函数 fork 产生一个子进程，快照持久化完全交给子进程来处理，父进程继续处理客户端请求。子进程刚刚产生时，它和父进程共享内存里面的代码段和数据段。这时你可以将父子进程想像成一个连体婴儿，共享身体。这是 Linux 操作系统的机制，为了节约内存资源，所以尽可能让它们共享起来。在进程分离的一瞬间，内存的增长几乎没有明显变化。

AOF

AOF(append only file) 持久化：以独立日志的方式记录每次写命令重启时再重新执行 AOF 文件中的命令达到恢复数据的目的。AOF 的主要作用是解决了数据持久化的实时性，目前已经是 Redis 持久化的主流方式。

Redis 提供了 bgrewriteaof 指令用于对 AOF 日志进行瘦身。其原理就是开辟一个子进程对内存进行遍历转换成一系列 Redis 的操作指令，序列化到一个新的 AOF 日志文件中。

序列化完毕后再将操作期间发生的增量 AOF 日志追加到这个新的 AOF 日志文件中，追加完毕后就立即替代旧的 AOF 日志文件了，瘦身工作就完成了。

fork

当 Redis 做 RDB 或 AOF 重写时，一个必不可少的操作就是执行 fork 操作创建子进程，对于大多数操作系统来说 fork 是个重量级错误。虽然 fork 创建的子进程不需要拷贝父进程的物理内存空间，但是会复制父进程的空间内存页表。

阻塞

Redis 是典型的单线程架构，所有的读写操作都是在一条主线程中完成的。当 Redis 用于高并发场景时，这条线程就变成了它的生命线。

内在原因包括：不合理地使用 API 或数据结构、CPU 饱和、持久化阻塞等。

外在原因包括:CPU 竞争、内存交换、网络问题等。

内存回收策略

Redis 的内存回收机制主要体现在以下两个方面：

- 删除到达过期时间的键对象。
- 内存使用达到 maxmemory 上限时触发内存溢出控制策略。

Redis 并不总是可以将空闲内存立即归还给操作系统。

如果当前 Redis 内存有 10G，当你删除了 1GB 的 key 后，再去观察内存，你会发现内存变化不会太大。原因是操作系统回收内存是以页为单位，如果这个页上只要有一个 key 还在使用，那么它就不能被回收。Redis 虽然删除了 1GB 的 key，但是这些 key 分散到了很多页面中，每个页面都还有其它 key 存在，这就导致了内存不会立即被回收。

不过，如果你执行 flushdb，然后再观察内存会发现内存确实被回收了。原因是所有的 key 都干掉了，大部分之前使用的页面都完全干净了，会立即被操作系统回收。

删除过期键对象

Redis 所有的键都可以设置过期属性，内部保存在过期字典中。由于进程内保存大量的键，维护每个键精准的过期删除机制会导致消耗大量的 CPU，对于单线程的 Redis 来说成本过高，因此 Redis 采用惰性删除和定时任务删除机制实现过期键的内存回收。

分布式锁

比如一个操作要修改用户的状态，修改状态需要先读出用户的状态，在内存里进行修改，改完了再存回去。如果这样的操作同时进行了，就会出现并发问题，因为读取和保存状态这两个操作不是原子的。(Wiki 解释：所谓原子操作是指不会被线程调度机制打断的操作；这种操作一旦开始，就一直运行到结束，中间不会有任何 context switch 线程切换。)

这个时候就要使用到分布式锁来限制程序的并发执行。

分布式锁本质上要实现的目标就是在 Redis 里面占一个“茅坑”，当别的进程也要来占时，发现已经有人蹲在那里了，就只好放弃或者稍后再试。占坑一般是使用 setnx(set if not exists) 指令，只允许被一个客户端占坑。先来先占，用完了，再调用 del 指令释放茅坑。

但是有个问题，如果逻辑执行到中间出现异常了，可能会导致 del 指令没有被调用，这样就会陷入死锁，锁永远得不到释放。于是我们在拿到锁之后，再给锁加上一个过期时间，比如 5s，这样即使中间出现异常也可以保证 5 秒之后锁会自动释放。

```
> setnx lock:codehole true
> expire lock:codehole 5
> del lock:codehole
```

但是以上逻辑还有问题。如果在 setnx 和 expire 之间服务器进程突然挂掉了，可能是因为机器掉电或者是被人为杀掉的，就会导致 expire 得不到执行，也会造成死锁。

这种问题的根源就在于 setnx 和 expire 是两条指令而不是原子指令。如果这两条指令可以一起执行就不会出现问题。也许你会想到用 Redis 事务来解决。但是这里不行，因为 expire 是依赖于 setnx 的执行结果的，如果 setnx 没抢到锁，expire 是不应该执行的。事务里没有 if-else 分支逻辑，事务的特点是一口气执行，要么全部执行要么一个都不执行。

后来引入了 set 指令的拓展参数，使得 setnx 和 expire 可以一起执行。

超时问题

Redis 的分布式锁不能解决超时问题，如果在加锁和释放锁之间的逻辑执行的太长，以至于超出了锁的超时限制，就会出现问题。因为这时候锁过期了，第二个线程重新持有了这把锁，但是紧接着第一个线程执行完了业务逻辑，就把锁给释放了，第三个线程就会在第二个线程逻辑执行完之间拿到了锁。

有一个更加安全的方案是为 set 指令的 value 参数设置为一个随机数，释放锁时先匹配随机数是否一致，然后再删除 key。但是匹配 value 和删除 key 不是一个原子操作，Redis 也没有提供类似于 delifequals 这样的指令，这就需要使用 Lua 脚本来处理了，因为 Lua 脚本可以保证连续多个指令的原子性执行。

延时队列

Redis 的 list(列表) 数据结构常用来作为异步消息队列使用，

使用 rpush/lpush 操作入队列，

使用 lpop 和 rpop 来出队列。

队列空了怎么办?

客户端是通过队列的 pop 操作来获取消息，然后进行处理。处理完了再接着获取消息，再进行处理。如此循环往复，这便是作为队列消费者的客户端的生命周期。

可是如果队列空了，客户端就会陷入 pop 的死循环，不停地 pop，没有数据，接着再 pop，又没有数据。这就是浪费生命的空轮询。空轮询不但拉高了客户端的 CPU，redis 的 QPS 也会被拉高，如果这样空轮询的客户端有几十来个，Redis 的慢查询可能会显著增多。

通常我们使用 sleep 来解决这个问题，让线程睡一会，睡个 1s 钟就可以了。不但客户端的 CPU 能降下来，Redis 的 QPS 也降下来了。

用上面睡眠的办法可以解决问题。但是有个小问题，那就是睡眠会导致消息的延迟增大。

队列延迟

有没有什么办法能显著降低延迟呢？你当然可以很快想到：那就把睡觉的时间缩短点。这种方式当然可以，不过有没有更好的解决方案呢？当然也有，那就是 blpop/brpop。

这两个指令的前缀字符 b 代表的是 blocking，也就是阻塞读。阻塞读在队列没有数据的时候，会立即进入休眠状态，一旦数据到来，则立刻醒过来。消息的延迟几乎为零。用 blpop/brpop 替代前面的 lpop/rpop，就完美解决了上面的问题。.

空闲连接问题?

如果线程一直阻塞在哪里，Redis 的客户端连接就成了闲置连接，闲置过久，服务器一般会主动断开连接，减少闲置资源占用。这个时候 blpop/brpop 会抛出异常来。

锁冲突处理

三种策略:

1、直接抛出异常，通知用户稍后重试;

2、sleep 一会再重试;

3、将请求转移至延时队列，过一会再试;

直接抛出特定类型的异常

这种方式比较适合由用户直接发起的请求，用户看到错误对话框后，会先阅读对话框的内容，再点击重试，这样就可以起到人工延时的效果。

sleep

sleep 会阻塞当前的消息处理线程，会导致队列的后续消息处理出现延迟。如果碰撞的比较频繁或者队列里消息比较多，sleep 可能并不合适。如果因为个别死锁的 key 导致加锁不成功，线程会彻底堵死，导致后续消息永远得不到及时处理。

HyperLogLog

HyperLogLog 提供不精确的去重计数方案，虽然不精确但是也不是非常不精确，标准误差是 0.81%

HyperLogLog 提供了两个指令 pfadd 和 pfcount，根据字面意义很好理解，一个是增加计数，一个是获取计数。pfadd 用法和 set 集合的 sadd 是一样的，来一个用户 ID，就将用户 ID 塞进去就是。pfcount 和 scard 用法是一样的，直接获取计数值。

pfmerge 适合什么场合用?

HyperLogLog 除了上面的 pfadd 和 pfcount 之外，还提供了第三个指令 pfmerge，用于将多个 pf 计数值累加在一起形成一个新的 pf 值。

比如在网站中我们有两个内容差不多的页面，运营说需要这两个页面的数据进行合并。其中页面的 UV 访问量也需要合并，那这个时候 pfmerge 就可以派上用场了。

pf 的内存占用为什么是 12k ？

Redis 的 HyperLogLog

实现中用到的是 16384 个桶，也就是 2^14，每个桶的 maxbits 需要 6 个 bits 来存储，最大可以表示 maxbits=63，于是总共占用内存就是 2^14 * 6 / 8 = 12k 字节。

布隆过滤器

布隆过滤器可以理解为一个不怎么精确的 set 结构，当你使用它的 contains 方法判断某个对象是否存在时，它可能会误判。但是布隆过滤器也不是特别不精确，只要参数设置的合理，它的精确度可以控制的相对足够精确，只会有小小的误判概率。

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。

原理:

每个布隆过滤器对应到 Redis 的数据结构里面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位 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。

简单限流

当系统的处理能力有限时，如何阻止计划外的请求继续对系统施压，这是一个需要重视的问题。

如何使用 Redis 来实现简单限流策略?

系统要限定用户的某个行为在指定的时间里只能允许发生 N 次，如何使用 Redis 的数据结构来实现这个限流的功能?

解决方案

这个限流需求中存在一个滑动时间窗口，想想 zset 数据结构的 score 值，是不是可以通过 score 来圈出这个时间窗口来。而且我们只需要保留这个时间窗口，窗口之外的数据都可以砍掉。那这个 zset 的 value 填什么比较合适呢? 它只需要保证唯一性即可，用 uuid 会比较浪费空间，那就改用毫秒时间戳吧。

GeoHash

业界比较通用的地理位置距离排序算法是 GeoHash 算法，Redis 也使用 GeoHash 算法。GeoHash 算法将二维的经纬度数据映射到一维的整数，这样所有的元素都将在挂载到一条线上，距离靠近的二维坐标映射到一维后的点之间距离也会很接近。当我们想要计算「附近的人时」，首先将目标位置映射到这条线上，然后在这个一维的线上获取附近的点就行了。

那这个映射算法具体是怎样的呢? 它将整个地球看成一个二维平面，然后划分成了一系列正方形的方格，就好比围棋棋盘。所有的地图元素坐标都将放置于唯一的方格中。方格越小，坐标越精确。然后对这些方格进行整数编码，越是靠近的方格编码越是接近。

Pub/Sub

Redis 消息队列的不足之处，那就是它不支持消息的多播机制。

消息多播允许生产者生产一次消息，中间件负责将消息复制到多个消息队列，每个消息队列由相应的消费组进行消费。它是分布式系统常用的一种解耦方式，用于将多个消费组的逻辑进行拆分。支持了消息多播，多个消费组的逻辑就可以放到不同的子系统中。

如果是普通的消息队列，就得将多个不同的消费组逻辑串接起来放在一个子系统中，进行连续消费。

小对象压缩

ziplist

如果 Redis 内部管理的集合数据结构很小，它会使用紧凑存储形式压缩存储。

Redis 的 ziplist 是一个紧凑的字节数组结构。

Redis 的 intset 是一个紧凑的整数数组结构，它用于存放元素都是整数的并且元素个数较少的 set 集合。

过期策略

Redis 所有的数据结构都可以设置过期时间，时间一到，就会自动删除。

过期的 key 集合

redis 会将每个设置了过期时间的 key 放入到一个独立的字典中，以后会定时遍历这个字典来删除到期的 key。除了定时遍历之外，它还会使用惰性策略来删除过期的 key，所谓惰性策略就是在客户端访问这个 key 的时候，redis 对 key 的过期时间进行检查，如果过期了就立即删除。定时删除是集中处理，惰性删除是零散处理。