redis相关

笔记本: 我的第一个笔记本
创建时间: 2021/11/16 20:30 **更新时间:** 2021/11/17 11:28
作者: 影
URL: https://www.cnblogs.com/JavaBlackHole/p/7726195.html

---

redis相关

redis（remote dictionary server）：远程字典服务，基于内存的可持久化的日志型、key-value数据库。是一个NoSql数据库，也被成为结构化数据库。
优点：
1.基于内存，读写速度快
2.支持丰富的数据类型：String，list，set，hash，Sorted Set， Geospatial，Hyperloglog，Bitmap
3.支持RDB和AOF两种持久化方案，解决内存数据断电即失的缺陷。

---

redis速度快的原因：
1.采用了多路复用IO阻塞机制
2.数据结构简单，操作节省时间
3.运行在内存，速度快
4.采用单线程，避免了不必要的上下文切换，不用考虑各种锁的问题，不存在加锁，释放锁的操作，不可能出现死锁导致的性能消耗。

---

redis持久化机制：
1.RDB（redis database）：在指定的时间间隔将内存中的数据集快照写入磁盘，恢复时将快照文件（dump.rdb）直接读到内存里，Redis会单独创建（fork）一个子进程来进行持久化，由于父子进程共享内存，采用copy on write（写时复制）策略，父进程继续接收写命令，并写入缓存，子进程将fork那一刻内存中的数据写入临时文件，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了的文件。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，redis默认的就是RDB方式，一般情况下不需要修改这个配置。
RDB持久化触发机制：
• redis.conf中的save 时间 修改次数 指令条件满足（表示某段时间内修改了多少次key就会自动生成.rdb文件）

```
save 900 1
save 300 10
save 60 10000
```

• 执行flushall命令
• 关闭redis
redis修复dump.rdb文件：
• 将.rdb文件放在redis启动目录，启动redis时会自动检查dump.rdb文件并恢复其中的数据。
RDB方式的缺点：
• 无法保证数据完整性，容易丢失最后一次修改的数据
• fork子进程会占用一定内存

2.AOF（append only file）：以日志的形式记录每一条写命令，追加到.aof文件中，不可以改写，redis启动会读取.aof文件并执行每一条命令重建数据。
redis 修复appendonly.aof
可以使用 redis-check-aof —fix appendonly.aof命令对其进行修复。

AOF方式的优点：
• 由于记录每一条写命令，数据完整性有保证；
• 默认每秒同步一次，可能会丢失一秒内的数据（同步参数设置always，no，everysec）

```
# appendfsync always
appendfsync everysec
# appendfsync no
```

AOF方式的缺点：
• 由于不断追加数据，AOF会越来越大（提供重写策略来解决AOF数据冗余问题，默认最大64m进行重写），修复数据会比rdb文件慢；

RDB和AOF方式的选择：
如果想有更高的效率同时对数据完整性要求不高，可以选择RDB方式；
redis4.0提供了混合持久化方式，将rdb的内容写到aof的开头，读取时先加载rdb的内容，再加载aof的内容，这样既能有较快的加载速度，同时避免丢失过多数据。

---

redis持久化数据和缓存怎么扩容？
• 如果redis被当做缓存用，使用一致性哈希实现动态扩容缩容
• 如果redis被当做一个持久化存储使用，必须使用固定的keys-to-nodes映射关系，节点的数量一旦确定不能变化。当redis节点需要动态变化时，必须使用可以在运行时进行数据再平衡的一套系统，使用redis集群可以实现。

---

redis事务支持隔离性：
redis是单进程，保证在执行事务时，不会对事务进行中断，事务可以运行到执行完所有事务队列中的命令为止。因此，redis的事务总是带有隔离性的。

redis事务不具有原子性：
redis事务：由一连串指令构成，没有隔离级别的概念，单条指令保证原子性，但整个事务不保证原子性

mysql事务：1.要么全部执行成功，要么异常全不执行；2.异常发生后，可以回滚，就像没执行过一样；
Redis事务异常以后其他命令依旧执行，没有发生回滚，Redis大部分是命令语法错误引发异常。

redis事务异常类型：
• 编译时异常：语法有问题，编译不通过，所有指令都不会被执行
• 运行时异常：代码有问题，有问题的指令不会执行，其他指令照常执行

---

redis和memcached的比较：
1.性能上：redis使用单核，memcached使用多核。存储小数据时redis性能更高，但数据量超过100k时，memcached性能更好。
因为memcached使用多核，引入了缓存一致性和锁，所以有时高并发下单线程的redis比多线程的memcached效率要高。

2.内存空间和数据量大小： memcached可以修改最大内存，使用LRU算法；redis可以使用虚拟内存，突破了物理内存的限制。

3.数据类型与操作：memcached数据结构单一，仅用来缓存数据；redis支持k/v以外的list,set,hash等多种数据类型。

4.可靠性：memcached不支持数据持久化，断电或重启后数据消失，但稳定性是有保证的；
redis支持数据持久化和数据恢复，允许单点故障，但也会付出性能的代价。
memcached只是个内存缓存，对可靠性要求不高，而redis更倾向于内存数据库，有一定的可靠性保证。

---

redis和memcached的应用场景选择：

适用于redis的场景：

1、复杂数据结构，value 的数据是哈希，列表，集合，有序集合等这种情况下，会选择redis，因为memcache 无法满足这些数据结构，最典型的的使用场景是，用户订单列表， 用户消息，帖子评论等。

2、需要进行数据的持久化功能，但是注意，不要把redis 当成数据库使用，如果redis 挂了，内存能够快速恢复热数据，不会将压力瞬间压在数据库上，没有cache 预热的过程。对于只读和数据一致性要求不高的场景可以采用持久化存储

3、高可用，redis 支持集群，可以实现主动复制，读写分离，而对于memcached 如果想要实现高可用，需要进行二次开发。

4、存储的内容比较大，redis存储的value最大为512M， memcached 存储的value 最大为1M。

适用于memcached的场景：

1、纯KV,数据量非常大的业务，使用memcached 更合适，原因是，

a)memcache 的内存分配采用的是预分配内存池的管理方式，能够省去内存分配的时间，redis 是临时申请空间，可能导致碎片化。

b)虚拟内存使用，memcached 将所有的数据存储在物理内存里，redis 有自己的vm 机制，理论上能够存储比物理内存更多的数据，当数据超量时，引发swap,把冷数据刷新到磁盘上，从这点上，数据量大时，memcache 更快

c)网络模型，memcached 使用非阻塞的IO 复用模型，redis 也是使用非阻塞的IO 复用模型，但是redis 还提供了一些非KV 存储之外的排序，聚合功能，复杂的CPU 计算，会阻塞整个IO 调度，从这点上由于redis 提供的功能较多，memcache 更快些

d) 线程模型，memcached 使用多线程，主线程监听，worker 子线程接受请求，执行读写，这个过程可能存在锁冲突。redis 使用的单线程，虽然无锁冲突，但是难以利用多核的特性提升吞吐量。

---

redis的key的寻址方式：

在数据的放置策略上，Redis Cluster将整个 key的数值域分成16384个哈希槽，每个节点上可以存储一个或多个哈希槽，也就是说当前Redis Cluster支持的最大节点数就是16384。当有某个key被映射到某个节点负责的槽，那么这个节点负责为这个key提供服务。至于哪个节点存储哪些槽，可以手动指定，也可以在初始化时生成。节点用比特位标识自己是否拥有某个槽 。

redis 持久化，底层实现，优缺点：

RDB(Redis DataBase:在不同的时间点将redis 的内存数据生成的快照同步到磁盘等介质上):内存到硬盘的快照，定期更新。缺点：耗时，耗性能(fork+io 操作)，易丢失数据。
AOF(Append Only File：将redis 所执行过的所有指令都记录下来，在下次redis 重启时，只需要执行指令就可以了):写日志。缺点：体积大，恢复速度慢。

bgsave 做镜像全量持久化，aof 做增量持久化。因为bgsave 会消耗比较长的时间，不够实时，在停机的时候会导致大量的数据丢失，需要aof 来配合，在redis 实例重启时，优先使用aof 来恢复内存的状态，如果没有aof 日志，就会使用rdb 文件来恢复。Redis 会定期做aof 重写，压缩aof 文件日志大小。Redis4.0 之后有了混合持久化的功能，将bgsave 的全量和aof 的增量做了融合处理，这样既保证了恢复的效率又兼顾了数据的安全性。bgsave 的原理，fork 和cow，fork 是指redis 通过创建子进程来进行bgsave 操作，cow （写时复制）指的是copy on write，子进程创建后，父子进程共享数据段，父进程继续提供读写服务，写脏的页面数据会逐渐和子进程分离开来。

缓存穿透、缓存击穿、缓存雪崩解决方案？

缓存穿透：指查询一个一定不存在的数据，如果从存储层查不到数据则不写入缓存，这将导致这个不存在的数据每次请求都要到DB 去查询，可能导致DB 挂掉。

解决方案：1.查询返回的数据为空，仍把这个空结果进行缓存，但过期时间会比较短；2.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 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DB 的查询。

缓存击穿：对于设置了过期时间的key，缓存在某个时间点过期的时候，恰好这时间点对这个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DB 压垮。

解决方案：

1.使用互斥锁：

简化版：一个线程从缓存取数据，如果没有，获取锁去存储层取数据，并把数据写入缓存，此时其他线程无法访问存储层，必须等待。

进阶版：（理论上）对每个key设置锁，如果缓存查不到这个key的值，当前线程获取锁去访问存储层，把数据写入缓存。此时，所有访问这个key的线程需要等待，而访问其他key的线程不受影响。

2.永远不过期：物理不过期，但逻辑过期（后台异步线程去刷新：缓存层面不设置过期时间，逻辑层面为每个key设置过期时间，当超过过期时间，使用单独的线程去更新缓存）。

缓存雪崩：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。与缓存击穿的区别：雪崩是很多key，击穿是某一个key 缓存。

解决方案：

1.将缓存失效时间分散开，比如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。

2.可以采用哨兵模式或分布式建立redis集群将缓存层设计成高可用的，即使个别节点宕机也不会影响服务的提供；

3.采用多级缓存，不同级别的缓存设置不同的过期时间；

缓存与数据库不一致怎么办

• 首先第一种情况：如果我们先删缓存，再写数据库，在高并发的情况下，当第一个线程删除了缓存，还没来得及写数据库，这时第二个线程来读取数据，这时因为缓存删除了数据，所以会去读数据库，然而这时因为还没写入，所以读取到的就是旧数据。读完之后，会把这个数据存入缓存中（这时第一个线程已经将新的修改的值写入缓存了），这样缓存中的值就会被覆盖成修改前的数据（即旧数据）。

• 解决方法：对于第一种情况，通常要求写的操作不会太频繁。

  • 先操作缓存，但是不删除缓存，将缓存修改为一个特殊值（即和业务无关的值：比如-1000等等这样的），客户端读取缓存时，发现是特殊值，就休眠一小会（再休眠的时候，再进行删除缓存，写数据库），再去查一次Redis。这个解决办法可能存在的问题：特殊值可能存在义务侵入。休眠时间可能会出现多次，对性能有一定影响。

  • 延时双删：先删除缓存，然后再写数据库，休眠一会，再删除一次缓存。这个解决办法可能存在的问题：如果写操作频繁，还是会存在数据脏数据的可能。

• 先写数据库，再删缓存：如果数据库写完以后，缓存删除失败，数据就会不一致。

• 解决办法：

  • 给缓存设置一个过期时间。问题：过期时间内，缓存数据不会更新

  • 将热点数据缓存设置永不过期，但是在value里面设置一个逻辑上的过期时间，另外起一个后台线程，扫面这些key，对于逻辑上过期的缓存，进行删除。

---

redis分布式锁的设计与优化：

最简单的分布式锁：使用SETNX和DEL两个命令即可。存在的可能问题：如果获取锁的进程失败，那么它就永远不会解锁。那这个锁就会被锁死。

  • 优化：给锁设置一个过期时长（这样还是会有可能造成锁死的问题）

  • 优化：将锁的内容设置成过期时间，SETNX获取锁失败的时候，拿这个时间跟当前时间比对，如果时过期时间的锁，就先删除锁，再重新上锁。（这样可能存在的问题，在高并发的情况下，会存在多个进程同时拿到锁的情况）

主从数据库不一致如何解决
场景描述，对于主从库，读写分离，如果主从库更新同步有时差，就会导致主从库数据的不一致
1、忽略这个数据不一致，在数据一致性要求不高的业务下，未必需要时时一致性
2、强制读主库，使用一个高可用的主库，数据库读写都在主库，添加一个缓存，提升数据读取的性能。
3、选择性读主库，添加一个缓存，用来记录必须读主库的数据，将哪个库，哪个表，哪个主键，作为缓存的 key,设置缓存失效的时间为主从库同步的时间，如果缓存当中有这个数据，直接读取主库，如果缓存当中没有这个主键，就到对应的从库中读取。