Redis

参考文章

- 程序员库森
- CSDN Redis面试题
- CSDN几率大的Redis面试题（待看...）
- 小林coding

整体

- Conception：
  a. Redis、Memcached
- Question：
  a. 什么是Redis?
    - 本质上是一个key-value类型的数据库，和memcached类似，加载到内存中进行操作，纯内存操作，性能出色，是已知的性能最快的key-value数据库
    - 优点：
      - 读写性能出色
      - 数据结构丰富
      - 支持数据持久化，AOF、RDB两种
      - 支持事务，所有操作是原子性的，可以通过MULTI和EXEC指令
      - 支持主从复制、读写分离
    - 缺点：
      - 内存有限，不适合海量数据的高性能读写
      - 宕机会导致从机未能及时同步

  b. Redis和Memcached相比?

	Memcached	**Redis**
数据类型	简单的字符串	string、list、hash、set、sorted set
持久化	不支持	AOF、RDB

网络IO	非阻塞IO	IO多路复用
集群模式	没有原生的集群模式	主从同步

c. Redis为什么这么快?
   - 使用内存存储，没有磁盘IO的开销
   - 单线程处理请求，避免线程切换和锁资源的开销
   - 使用IO多路复用技术，使用epoll
   - 数据结构丰富，可以直接应用的优化数据
   - 冷热数据分离，热数据在内存中，冷数据在磁盘

d. **为什么用Redis做缓存?**
   - 高性能
     - 热点数据直接操作缓存，速度快
   - 高并发
     - 缓存支持的并发数比数据库多，把数据库中的部分数据转移到缓存中，部分请求直接访问缓存

e. Redis应用场景?
   - 缓存
   - 排行榜
   - 计数器：点击量、播放数
   - 分布式
   - 社交网络中的点、踩
   - 消息队列
   - Session共享

数据结构

- Conception：

a. 数据类型：底层实现

b. string：SDS

c. list：双向链表、压缩链表

d. hash：压缩链表、哈希表/字典、

e. set：哈希表/字典、整数集合

f. zset：压缩链表、跳表

g. GEO、Bitmap、HyperLogLog、Stream

h. quicklist（双向链表）、listpack（压缩链表）

- Question：

a. Redis kv存储方式

- dicEntry存储每一个kv

```
typedef struct dictEntry {
    void *key;
    union {
        void *val;
        uint64_t u64;
        int64_t s64;
        double d;
    } v;
    struct dictEntry *next;     /* Next entry in the same hash bucket. */
    void *metadata[];           /* An arbitrary number of bytes (starting at a
                                 * pointer-aligned address) of size as returned
                                 * by dictType's dictEntryMetadataBytes(). */
} dictEntry;
```

- Redis对key进行计算
- dict存储所有key，因为会有冲突，开链法，所以需要进行rehash

```
struct dict {
    dictType *type;

    dictEntry **ht_table[2];
    unsigned long ht_used[2];

    long rehashidx; /* rehashing not in progress if rehashidx == -1 */

    /* Keep small vars at end for optimal (minimal) struct padding */
    int16_t pauserehash; /* If >0 rehashing is paused (<0 indicates coding error) */
    signed char ht_size_exp[2]; /* exponent of size. (size = 1<<exp) */
};
```

b. 九种数据类型？应用场景？

- String：二进制安全、可以包含任何数据，比如jpg图片或者序列化的对象（常规计数）
- List：简单的字符串列表，按照插入顺序排序（最新回复）
- Hash：键值对集合（）
- Set：无序的**去重**集合（共同好友、共同关注）
- SortedSet：可排序版的set（排行榜）

- Bitmap：位图，一个以位为单位的数据
- Hyperloglog：统计基数，有误差
- Geospatial：地理位置信息（打车定位）
- Stream：为消息队列设计的数据类型

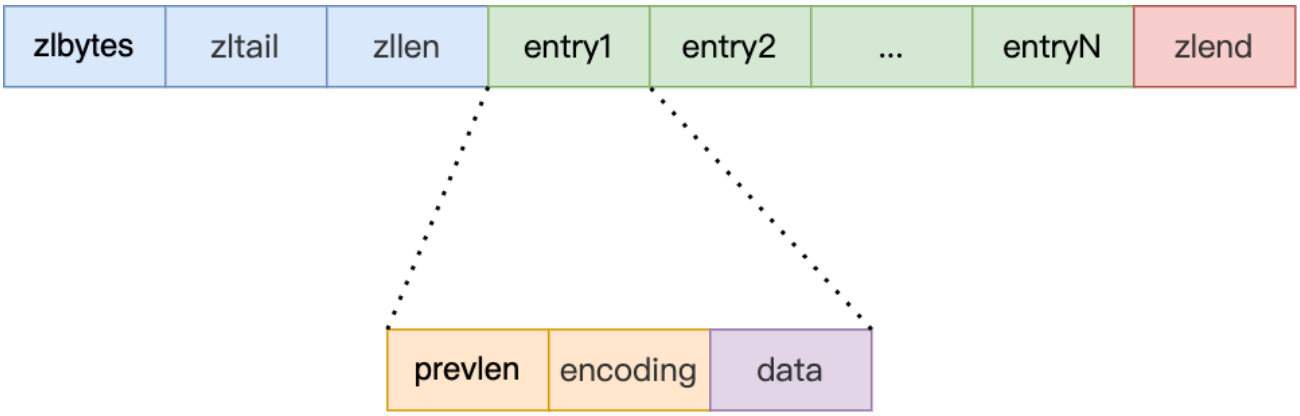
```
SET runoob "菜鸟教程"
GET runoob

HMSET runoob field1 "Hello" field2 "World"
HGET runoob field1

lpush runoob redis
lpush runoob mongodb
lpush runoob rabbitmq
lrange runoob 0 10

sadd runoob redis
sadd runoob rabbitmq
smembers runoob

zadd runoob 0 redis
zadd runoob 0 mongodb
ZRANGEBYSCORE runoob 0 1000
```

c. 底层数据类型：

- SDS：
  - 记录长度
  - 二进制格式，能存储任意数据类型
  - 多种数据类型，节省内存空间
- 链表（双向链表）
  - 双向链表、有头尾指针、有链表长度
  - **void*保存节点值**，链表节点内容多样性
- 哈希表
  - 链式哈希解决冲突
  - rehash：redis使用两个哈希表，数据增多需要扩容的时候，搬移数据到ht[1]上，然后交换

- 渐进式rehash：新增、更新、删除、更新操作时候，都会迁移到ht[1]上，最终都搬到ht[1]的时候上，完成rehash（查询时候如果ht[0]没有查到会去ht[1]上查找）
- 渐进式条件：
  - 负载因子 = 哈希表保存节点数 / 哈希表大小
  - 负载因子 >= 1，没有执行RDB和AOF时候rehash
  - 负载因子 >= 5，不管有无RDB和AOF操作都会强制rehash

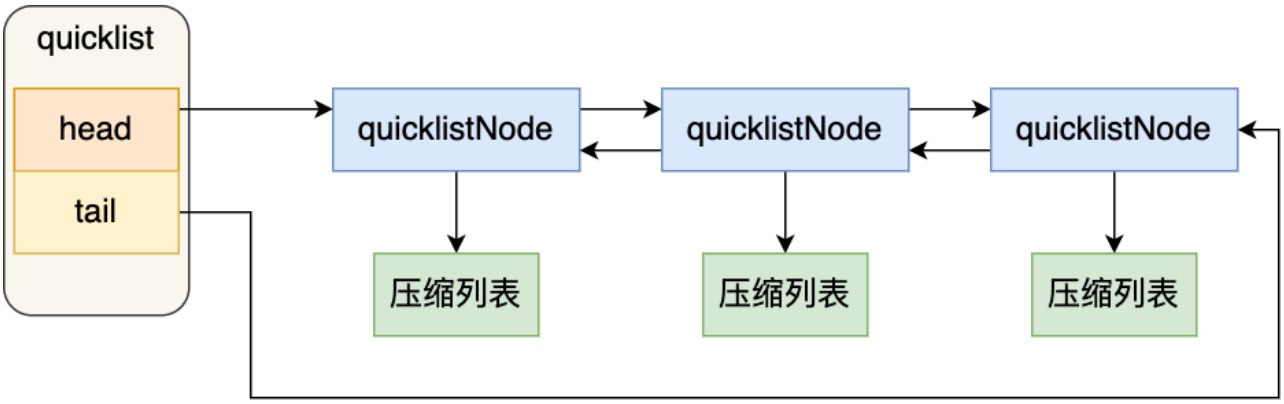
- 压缩链表



- 根据数据类型和大小进行分配，节省空间
- 连锁更新问题：前一个节点长度大于254字节，当前节点prevlen长度为1字节（2^8）变为5字节保存

- 整数集合：
  - 一个数组
  - 升级操作：节省空间，不能降级

- 跳表：
  - 单纯性能，跳表和红黑树性能相差不大，**并发环境下**，更新数据时，跳跃表需要更新的部分更少，锁的东西比较少，竞争锁的代价也更小
  - Redis选用跳表的原因
    - 内存占用：平衡树每个节点还需要保存两个左右指针，跳表每个节点包含的指针数目平均为 1/(1-p)，Redis中平均使用1.33个指针
    - 遍历操作：平衡树拿到节点后进行中序遍历很困难，跳表有前后指针
    - 实现难度：跳表实现更简单
- quicklist：
  - 对链表的更新，把每个节点换成了压缩列表 **双向链表 + 压缩列表**
    - 对双向链表的优势：压缩列表是内存连续的，比每个节点都离散的存法，更能利用缓存都读到缓存里
    - 对压缩列表的优势：把一整个压缩列表切分成了多个节点，连锁更新影响



  - 当前节点的压缩列表不能容纳时候，再创建下一个节点
- listpack
  - quicklist中每个压缩列表还是存在**连锁更新**的问题
  - 压缩链表不再记录前一个节点的长度，记录当前节点的长度，避免连锁更新

持久化AOF、RDB

- Conception：
  a. AOF（Append Only File）
  b. AOF重写
  c. RDB（Redis DataBase）
  d. 混合持久化
- Question：

a. AOF、优点、缺点?
   - 定义：以日志的形式记录每个操作，记录写指令不记录读指令，只许追加文件不允许修改，AOF保存的是appendonly.aof文件，恢复的时候从前到后执行一次
   - AOF重写：一直追加写文件会又来越大，重写机制可以进行文件压缩，只保留可以恢复数据的最小指令集
   - 机制：每次修改同步、每秒同步、不同步
   - 优点：
     - 后台线程每秒执行fsync，数据最多丢失一秒
     - 直接追加在文件末尾，写入性能高
   - 缺点：
     - aof文件大，恢复速度慢；
     - 数据恢复慢，不适合做冷备
b. RDB、优点、缺点?
   - 定义：指定时间间隔将数据快照写入磁盘，恢复的时候将快照文件直接读入内存
   - 优点：
     - 适合大规模数据恢复
   - 缺点：
     - 每隔一段时间才备份，Redis宕机的话会丢失数据，适合对数据完整性、一致性要求不高
c. 如何选择?
   - 对数据不敏感，可以关闭持久化
   - 可以承受数分钟的损失，如做缓存，只开RDB
   - 做内存数据库，RDB、AOF都开启，RDB时候做数据备份，AOF保证数据不丢失
d. 混合持久化?
   - Redis4.0
   - 前半段是RDB格式全量数据，后半段是AOF增量数据
   - 优点：绝大部分是RDB，加载速度快，增量是AOF，避免了数据丢失
   - 缺点：兼容性差，Redis4.0之前不识别该aof文件；阅读性差，前半段是RDB
e. Redis持久化数据和缓存怎么做扩容?
   - 如果Redis被当做缓存使用，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。否则的话（即Redis节点需要动态变化的情况），必须使用可以在运行时进行数据再平衡的一套系统，而当前只有Redis集群可以做到这样。
- 看参考文章CSDN里有很多Redis面试题

过期键删除策略

- Conception：
  a. 定时删除
  b. 惰性删除
  c. 定期删除
  d. 持久化相关
  e. 主从相关
- Question：
  a. 过期键的删除策略？

	定时删除	**惰性删除**	**定期删除**
方法	设置定时器，到时删除	过期不删除，再次访问时候检查	每隔一段时间对一部分键进查
优点	对内存优化	对CPU友好	减少了对CPU的影响，释放分内存
缺点	对CPU不友好	对内存不友好，造成内存泄露	难以衡量定时时长

  b. 过期时间和永久有效怎么设置？
     - EXPIRE、PEXPIRE：设置生存时间（秒/毫秒精度）
     - EXPIREAT、PEXPIREAT：设置过期时间（秒/毫秒精度）
  c. 实际使用：惰性删除+定期删除
  d. Redis持久化时，过期键处理策略？
     - AOF
       - AOF写入阶段：该值还没有被删除，保留过期键；真正删除时候追加一条**DEL命令**
       - AOF重写阶段：已过期的键不会被保存到重写后的AOF文件中
     - RDB
       - RDB生成阶段：对key进行检查，过期的key不会被载入到数据库中

- RDB加载阶段：
  - 主服务器：载入时候，会对key进行检查，过期时候不会被加载到数据库中
  - 从服务器：无论是否过期都会被载入（主从服务器同步时候，从服务器数据会被清空，所以过期键写到从服务器影响不大）

e. Redis主从模式中，过期键处理策略?
   - 从库对过期键删除是**被动**的（从库读到过期键还是会返回），主库key到期时候，在AOF中追加一条DEL命令，同步到所有从库

内存淘汰策略

- Concetion：
  a. noeviction
  b. volatile-lru、allkeys-lru
  c. volatile-lfu、allkeys-lfu
  d. volatile-random、allkeys-random
  e. volatile-ttl（越早过期越早删除）
  f. Redis4.0新增：两种lfu
- Question：
  a. 内存淘汰策略?
     - 设置了maxmemory选项，redis内存达到上限，使用方法释放空间对key进行淘汰
     - 不进行数据淘汰：noevction（不进行内存淘汰，返回错误）
     - 在**设置了过期时间**的数据种进行淘汰：volatile-ttl（越早过期越先删除）、volatile-random、volatile-lru、volatile-lfu
     - 在**所有数据范围**内进行淘汰：allkeys-random、allkeys-lru、allkeys-lfu
     - 优先使用allkeys-lru
  b. LRU、LFU实现
     - 见源码解读

缓存问题

- Conception：
  a. 缓存和数据库的数据不一致
  b. 缓存雪崩
  c. 缓存击穿

d. 缓存穿透

e. 缓存预热、缓存降级

f. 缓存写回策略

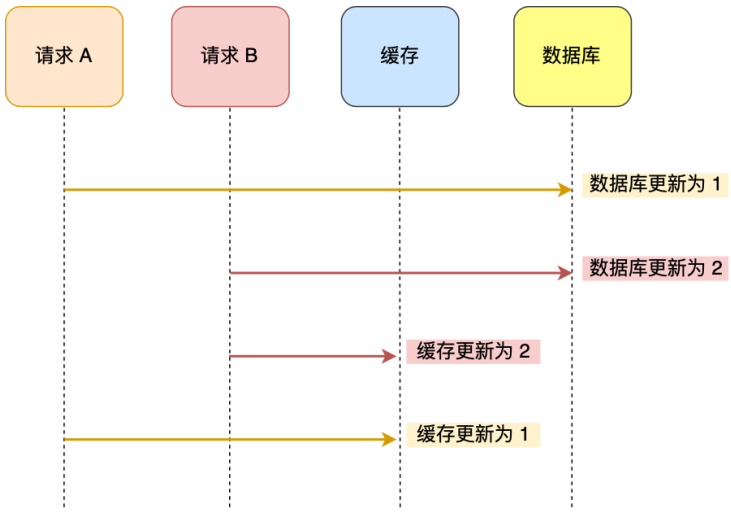
- Question：

a. 缓存异常四种类型？

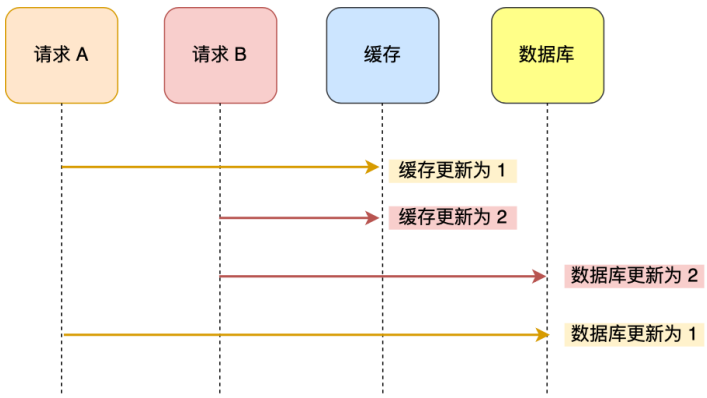
  - 缓存与数据库的数据不一致
  - 缓存雪崩
  - 缓存击穿
  - 缓存穿透

b. 缓存与数据库的数据不一致？

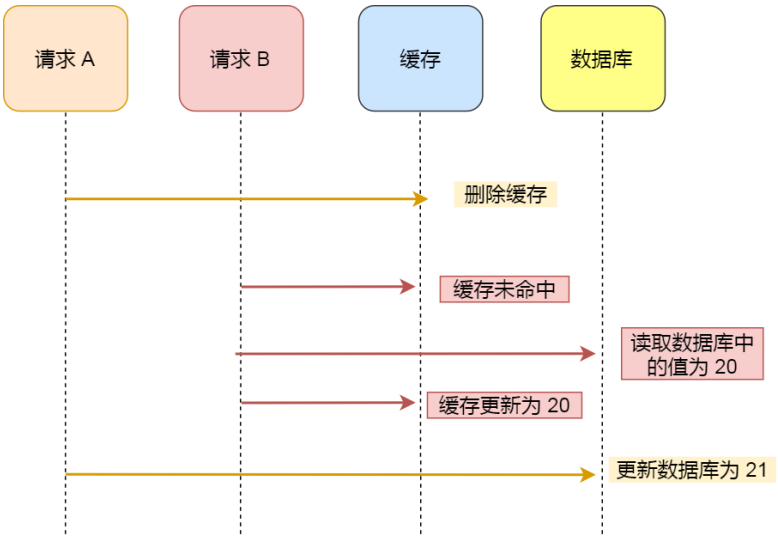
  - 数据库和缓存如何保证一致性？
  - 导致问题的原因：多线程
  - 方案：
    - 先更新数据库，再更新缓存



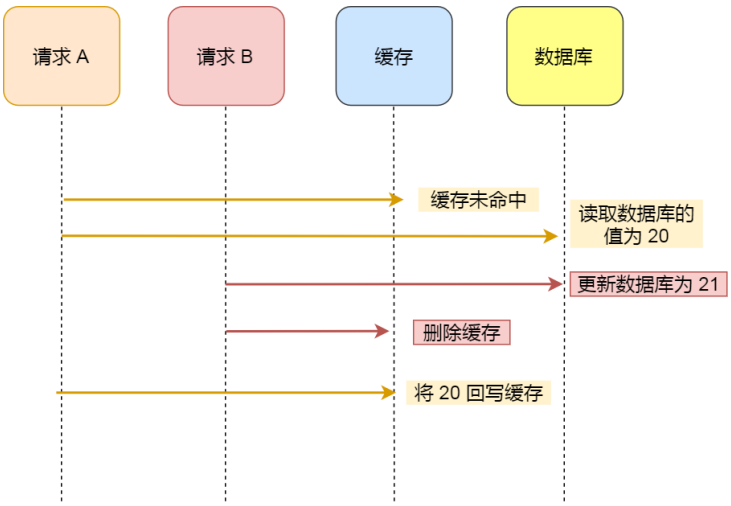
    - 先更新缓存，再更新数据库



    - 先删除缓存，再更新数据库



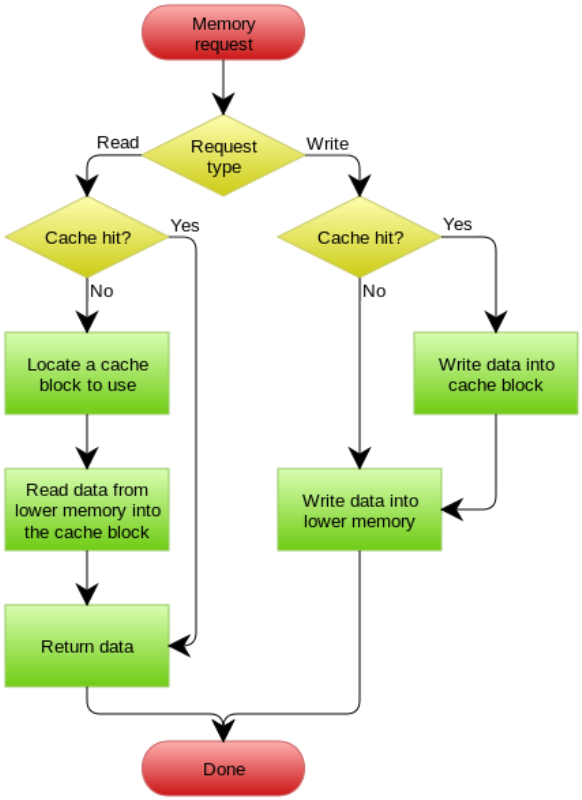
- 先更新数据库，再删除缓存
  - **业界最常用方法；缓存写入速度**比数据库写入速度快的多，很难出现请求A的写回缓存迟迟没有实现



- 延时双删，分布式锁

c. 缓存更新策略

- Cache Aside（旁路缓存）
  - 读策略：命中缓存直接返回；未命中从数据库读，写回缓存
  - 写策略：先修改数据库，再删除缓存
  - 适合读多写少，写太多时候，缓存删除频繁
- Read/Write Through（读穿、写穿）
  - 应用只感知缓存而不感知数据库，存储自己维护自己的Cache



- Write Back（写回）
  - 只更新缓存，之后批量更新到数据库；适合写多的场景
  - 问题：数据不是强一致的，会有数据丢失的风险
  - Redis不能应用的原因：Redis没有异步更新数据库的功能

d. 缓存雪崩？
   - 定义：大量key失效/缓存宕机，大量访问请求打在数据库上，数据库宕机
   - 解决：
     - 均匀过期，加上小的随机数；
     - 二级缓存， cache1为原始缓存，cache2为拷贝缓存，cache1缓存失效为短期，cache2缓存失效为长期
     - 服务降级（非核心数据不允许访问数据库）；服务熔断（暂停访问）；
     - 使用主从集群；

e. 缓存击穿？
   - 定义：某个热key失效，大量请求打在数据库上
   - 解决：
     - 热点数据不过期；
     - 缓存失败后使用**互斥锁**或者队列控制阻止对热key的访问

f. 缓存穿透？

- 定义：缓存和数据库种都没有需要的key，数据库压力过大
- 解决：
  - 使用**布隆过滤器**判断key是否存在，不存在不允许访问数据库；
  - 缓存空值，不存在的值在缓存直接返回

g. 缓存预热？
- 定义：系统上线前，提前将缓存数据加载到缓存中，避免开始对数据库的大量访问

h. 缓存降级？
- 定义：对于非核心数据自动降级，不去数据库查询，返回默认值，保证核心服务可用

集群

- Conception：

a. 单机模式

b. 主从

c. 读写分离

d. Sentinel哨兵

e. Cluster分片集群

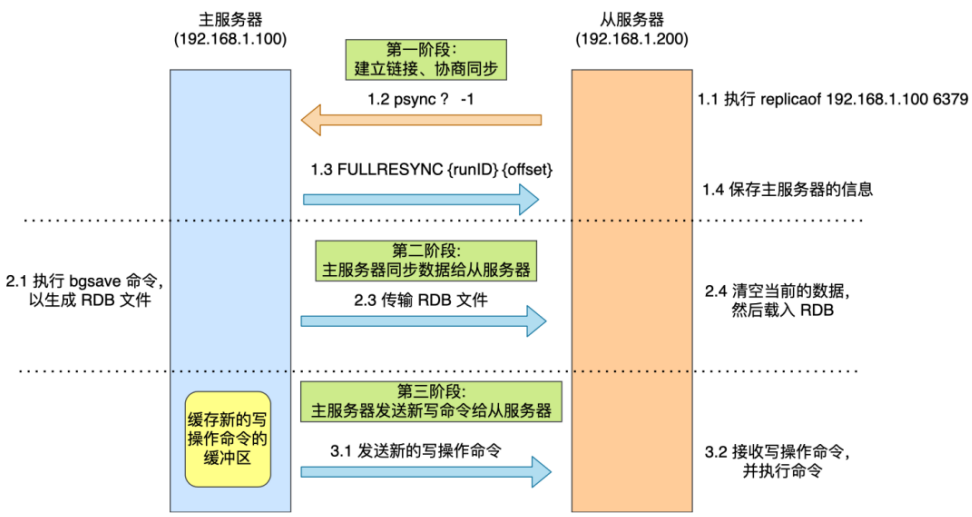
f. 脑裂

g. Redis自研

- Question：

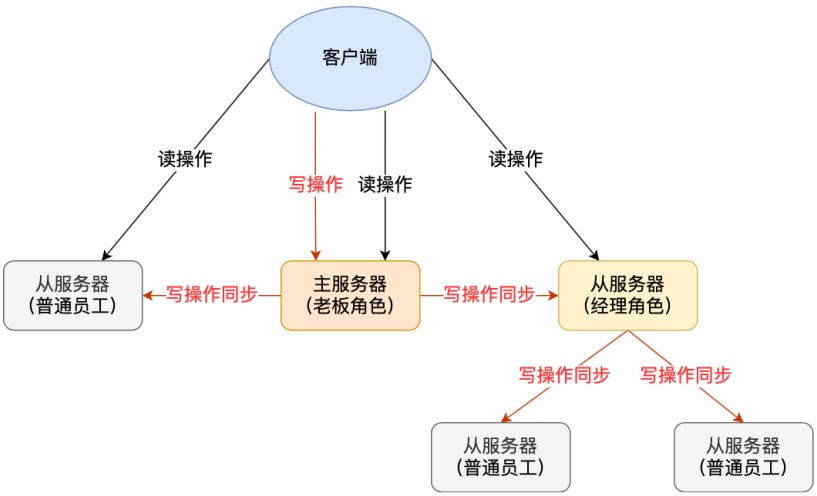
a. Redis常见的使用方式？
- 单机
- 主从 读写分离：

1. **主从第一次同步**：采用全量复制



- 第一阶段
  - 启动一个slave node时候，发送一个PSYNC命令给master
  - 主服务器用FULLRESYNC作为响应命令返回
- 第二阶段
  - master生成RDB，发送RDB
  - 此时新增的命令放在replication buffer中
- 第三阶段
  - 将replication_buffer中的命令发送给从服务器
- 后续：维护一个TCP长连接，进行命令传播

2. **ping-pong检测机制**
   - 一半以上的节点认为没有pong回应，集群认为节点挂掉
   - 主节点：10s向从节点发送ping命令，判断从节点存活
   - 从节点：1s向主节点上报复制偏移量，检查数据是否丢失
3. **从库可以分担主库全量复制的压力**



   - 主从全量复制对于主库来说压力大
   - 可以让从库对新来的slave进行同步
4. **断点续传、网络断了怎么办（repl_backlog_buffer）**
   - repl_backlog_buffer是一个环形缓冲区，master和slave会记录自己写到和读到的位置
   - 重连后判断master_backlog_buffer和slave_backlog_buffer之间的差距
   - 问题：环形可能导致覆盖，master会采取全量同步，合理设置repl_backlog_buffer的大小（second * write_size_per_second）
5. **repl_backlog_buffer和replication_buffer**

    - repl_backlog_buffer：**所有从库共用一个，增量**复制时候用于比较复制进度
    - replication_buffer：**每个从库都有一个，全量**复制阶段记录新来的命令
6. slave不会过期key，只会等待master过期key发送del命令

- **Sentinel**：
1. 定义：哨兵选举新的master，但是每个节点存储的数据是一样的，浪费空间
2. **主客观下线**
    - 主观下线：**某个哨兵节点**规定时间内没有收到响应
    - 客观下线：主观下线后向其他哨兵发送命令，哨兵赞同票数超过quorum值认为客观下线
3. 哨兵职责：
    - **监控**：每秒向主从库发送PING命令，超时标记为下线（主观下线），当有 N/2 + 1 个哨兵判断才能客观下线
    - **候选者选出哨兵leader**：每个哨兵节点向其他哨兵发出通知，超过半数和quorum值则成为leader
    - **更换主节点**：
        - 按照断开连接的时长、优先级、复制进度、ID号大小进行选主（原主库需要清空本地数据，再和新主库进行全量同步）；
        - 把新主库的信息发送给其他从库；
        - 旧主节点上线时候，设置为从节点
4. **哨兵之间、哨兵和从库如何相互发现?**
    - pub/sub机制，发布/订阅机制
    - 通过订阅主库的"__sentinel__:hello"频道进行相互发现
    - 哨兵向主库发送INFO命令获得从库信息
5. **哨兵数量要>=3，2个哨兵时1个挂了没办法选主**
6. **脑裂**
    - 定义：同时有两个主节点，都能接收写请求
    - 问题：往不同的主节点上写入数据，导致数据丢失（**原主库会先清空数据再全量同步**）
    - 解决：设定min-slaves-to-write（只要有几个slave）和min-slaves-max-lag（主从复制延迟不能超过多少s），**原主库从而被限制接收客户请求**

- **Cluster**：
1. 所有master的容量总和就是可缓存的总容量，n主n从

2. 如何分配?
    - 固定16384个哈希槽，对于n个节点各分配16384/n个槽
    - 每个槽对应的节点会在存储到客户端
    - 对于键值对使用RCR16计算一个值对16384取模
3. 槽对应的节点变化了怎么办?
    - 已经转移完：返回MOVED，对新的节点进行访问
    - 正在转移：返回ASK，客户端需要向新节点发送ASKING（新节点允许客户端接下来发送的命令）
4. Cluster中节点通信的机制?
    - gossip协议（包括ping，pong，meet，fail等等）
    - 节点之间不断通信，保证整个集群所有节点的数据是完整的

索引

- Conception：

a. 跳表skiplist、zset实现

- Question：

a. 使用跳表原因：

- Redis是内存数据库，B+树的提出为了IO数据库准备，B+树的每个节点的数量都是一个MySQL分区页的大小
- 红黑树在并发环境下使用不方便，锁的粒度更大；跳表锁的东西更少

b. **Redis索引**

- 本身没有表结构，更没有主键
- 使用set原因
    - set集合的成员元素项保证唯一。
    - 每个成员元素项可以指定一个分数score。支持范围查询。
- 使用事务维护索引
- 索引格式：
    - xxx_index:users:used_id:1:age
    - 表名_主键列_主键值_索引列
- 文章

事务

- Conception：
  a. 相关命令
- Question：
  a. 相关命令

```
//如果EXEC之前balanced被 其他客户端 改动，EXEC失败
WATCH balanced

//开启事务
MULTI
//事务命令
...
...
//执行事务
EXEC


//取消事务
DISCARD
//取消监控
UNWATCH
```

b. Redis中的事务?

- 原子性（不保证）：
  - 批量的事务命令会全部执行，或者全部不执行；但是中间某条命令发生语法错误也不会退出事务，后面的命令继续执行；不会回滚
  - 命令错误会导致EXEC报错，语法错误EXEC之后全部执行
  - 事务执行过程中不会被客户端的其他命令中断
- 隔离性：
  - 单进程程序，保证了执行事务时不会被中断
- 一致性：
  - 数据库执行前是一致的，事务执行只会，数据库也是一致的
- 持久性：
  - 开启AOF、RDB时支持持久性

c. Redis为什么不支持回滚?
   - Redis只会因为语法错误失败/命令用在了错误的键上而失败，入队时候不能发现，这些问题是由于编程错误（程序员自己的错误，没有机制能避免程序员自己造成的错误），应该在开发过程中发现，而不应该出现在生产环境中
   - 不支持回滚可以保证Redis的简单快速

d. Redis事务的其他实现?
   - Lua脚本：Redis保证脚本内的命令都执行，也不提供回滚，部分命令错误也依旧执行
   - 中间标记变量判断是否完成：编写复杂

e. Redis事务：
   - redis通过MULTI、EXEC、WATCH等命令来实现事务机制，事务执行过程将一系列多个命令按照顺序一次性执行，并且在执行期间，事务不会被中断，也不会去执行客户端的其他请求，直到所有命令执行完毕。事务的执行过程如下：
     - 服务端收到客户端请求，事务以MULTI开始
     - 如果客户端正处于事务状态，则会把事务放入队列同时返回给客户端QUEUED，反之则直接执行这个命令
     - 当收到客户端EXEC命令时，WATCH命令监视整个事务中的key是否有被修改，如果有则返回空回复到客户端表示失败，否则redis会遍历整个事务队列，执行队列中保存的所有命令，最后返回结果给客户端
     - WATCH的机制本身是一个CAS的机制，被监视的key会被保存到一个链表中，如果某个key被修改，那么REDIS_DIRTY_CAS标志将会被打开，这时服务器会拒绝执行事务。
     - 具体使用

线程模型

- Conception：

a. 单线程

b. Redis6.0多线程

- Question：

a. Redis为何选择单线程?
   - 数据库是IO密集型而不是CPU密集型，Redis几乎是纯内存操作，执行速度快，真正的瓶颈在于网络I/O，Redis使用单线程的I/O多路复用来解决
   - 单线程一些好处：
     - 避免切换线程的上下文开销
     - 避免加锁的开销

- 单线程简单可维护

b. **Redis为什么用单线程还很快?**

- 10W qps/s
- 大部分操作在内存中完成，高效的数据结构；机器瓶颈是内存或者网络带宽
- 单线程避免多线程竞争加锁的开销
- I/O多路复用机制监听多个端口

c. **Redis真的是单线程?**

- Redis2.6：启动2个后台线程，分别处理关闭文件、AOF刷盘
- Redis4.0：新增一个后台线程，异步释放Redis内存，lazyfree线程（删除操作交给后台线程，避免阻塞主线程）
- Redis6.0：执行命令是单线程，处理网络数据的读写和协议的解析是多线程I/O

d. **Redis6.0为何引入多线程?**

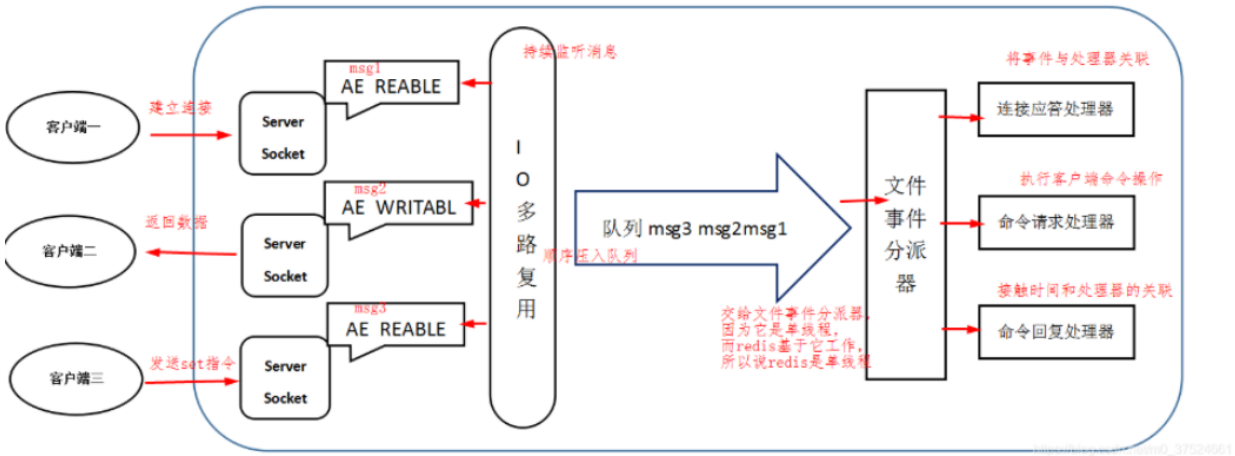
- 问题：I/O瓶颈明显，线上流量变大
- 解决方法：网络的I/O优化：零拷贝/DPDK技术（有局限不适用于Redis的复杂网络I/O场景）、多核优势
- 好处：充分利用CPU资源，分担I/O读写负荷

e. **Redis6.0多线程性能如何?**

- 几乎翻倍

f. **Redis的线程模型?**

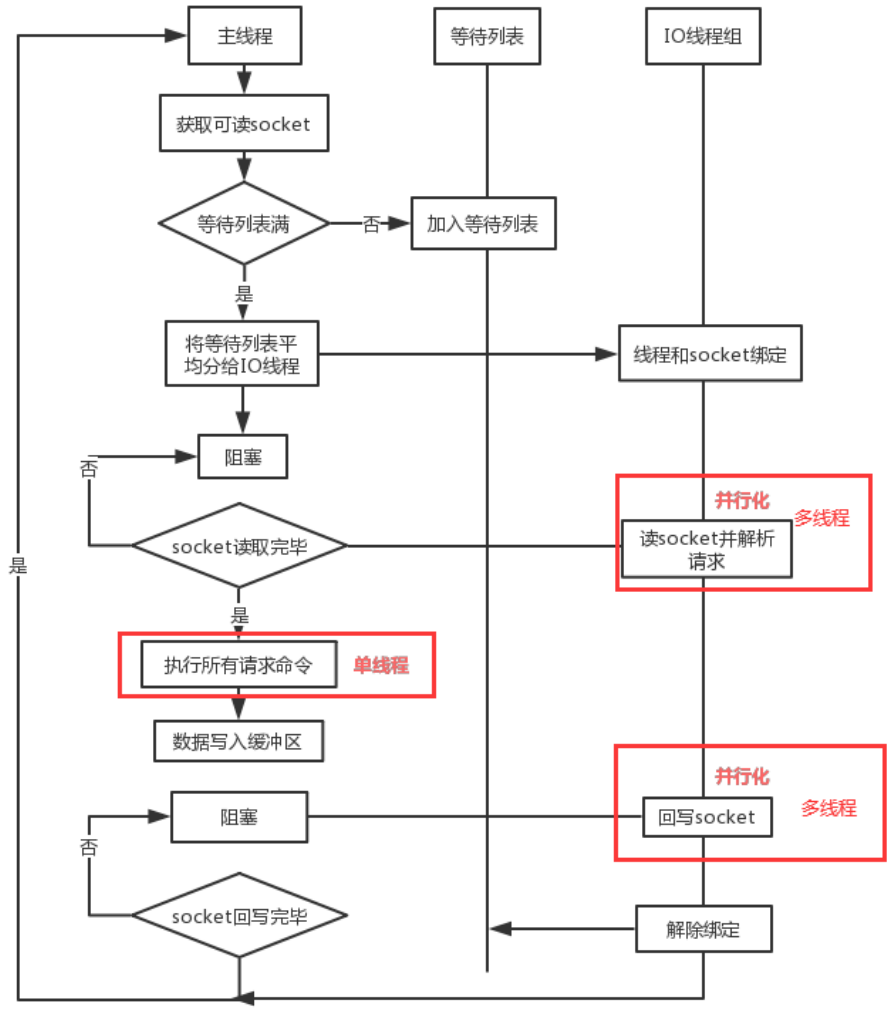
- Redis6.0之前



- Redis6.0之后
- 流程如下：

- 主线程获取 socket 放入等待列表
- 将 socket 分配给各个 IO 线程（并不会等列表满）
- 主线程阻塞等待 IO 线程（多线程）读取 socket 完毕
- 主线程执行命令 - 单线程（如果命令没有接收完毕，会等 IO 下次继续）
- 主线程阻塞等待 IO 线程（多线程）将数据回写 socket 完毕（一次没写完，会等下次再写）
- 解除绑定，清空等待队列

- 特点如下：
  - IO 线程要么同时在读 socket，要么同时在写，不会同时读或写
  - IO 线程只负责读写 socket 解析命令，不负责命令处理（主线程串行执行命令）
  - IO 线程数可自行配置



g. **Redis6.0是否默认开启多线程?**

- 否，在conf文件进行配置
- io-threads-do-reads yes

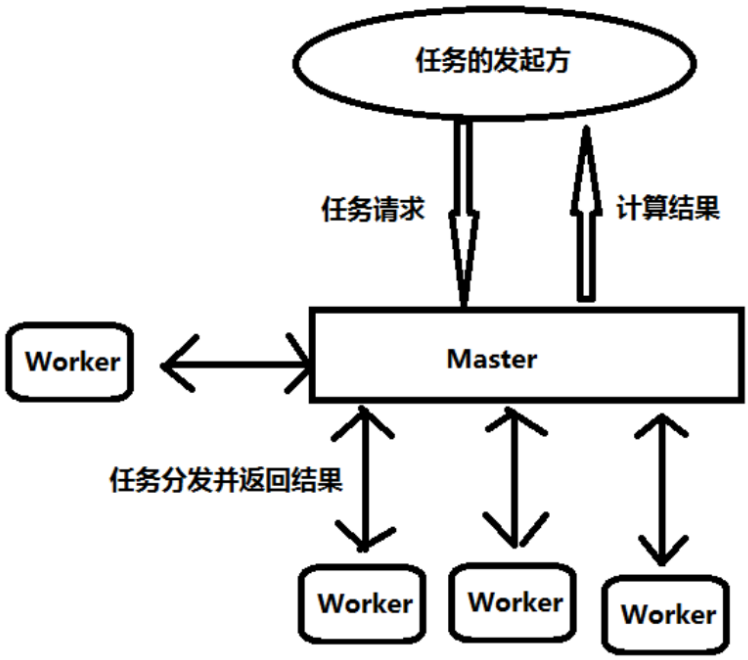
- io-threads 线程数
- 官方建议：4 核的机器建议设置为 2 或 3 个线程，8 核的建议设置为 6 个线程，线程数一定要小于机器核数，尽量不超过8个。

h. **多线程是否存在线程安全问题?**

- 不会，多线程只复杂网络数据读写和协议解析，执行命令依旧是单线程

i. **Redis6.0和Memcached多线程对比?**

- 相同点：都用了Master-Worker模型



- 不同点：
  - Memcached逻辑处理也在Worker线程，模型简单并且实现了线程隔离
  - Redis的逻辑处理依旧又交还在Master线程，模型复杂度增加但是线程并发安全问题解决

j. Redis6.0讲解文章

- Question：

a. 如何提高 Redis 命中率?

b. 怎么优化 Redis 的内存占用?

源码解读

LRU

- Least Recently Used 最近最少使用
- 问题：
  - 需要维护一个链表，占用内存
  - 每当有数据插入、再次访问时候，需要多次链表操作
- Redis近似LRU实现

a. 内存淘汰机制与redis.conf 中的两个配置参数有关
   - maxmemory：使用的最大内存容量
   - maxmemory-policy：设定了内存淘汰策略

b. 使用**时钟值**记录，有全局LRU时钟，每个值也有自己的时间戳，当当前使用内存已经超过maxmemory时候，近似LRU算法会随机选择一些键值对，形成待淘汰集合，根据它们的访问时间戳，选出最久的数据，进行淘汰

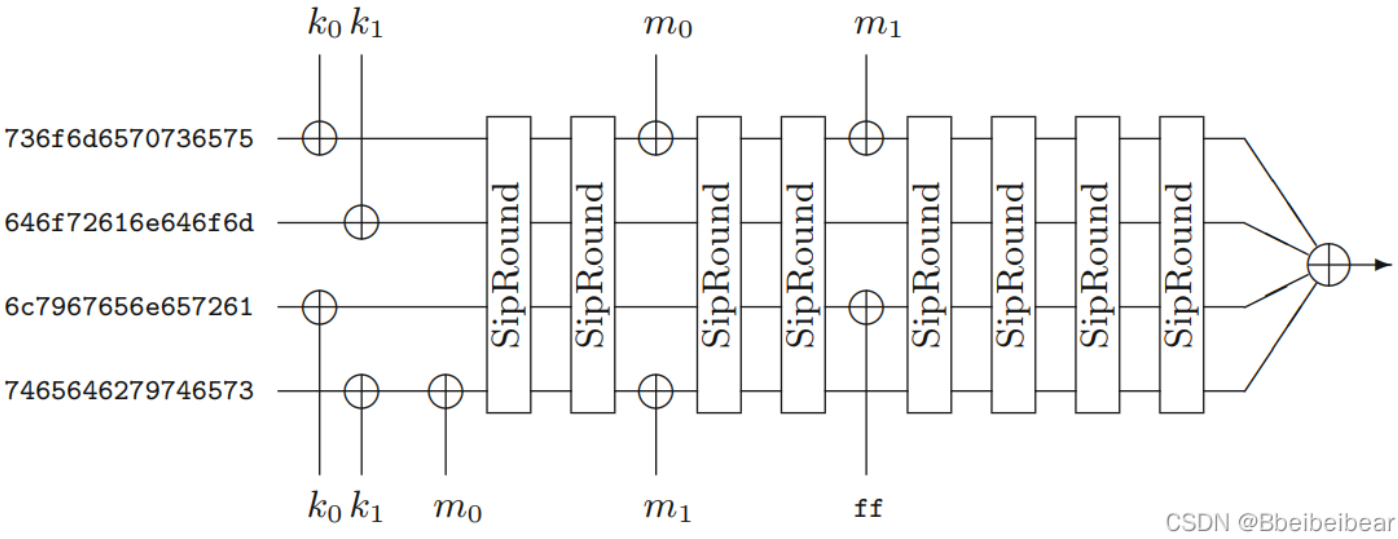
LFU

- Least Frequent Used 最不频繁使用
- 问题：
  - 应该算的是访问**频率**，而不是访问次数（eg：A 15分钟被访问15次，B 5分钟被访问10次，此时应该淘汰A）
- Redis近似LFU实现

a. 访问次数如何更新：
   i. 问题：8bit最多就是255，每次加1很快就溢出
   ii. 策略：先根据上次访问时间到目前时间，进行衰减操作；之后增加时候还要根据概率进行访问次数的增加，已有访问次数越大的键值对，访问次数越难增加

哈希算法

- Redis siphash
- k0k1是密钥拆的，m0m1是字符串拆的

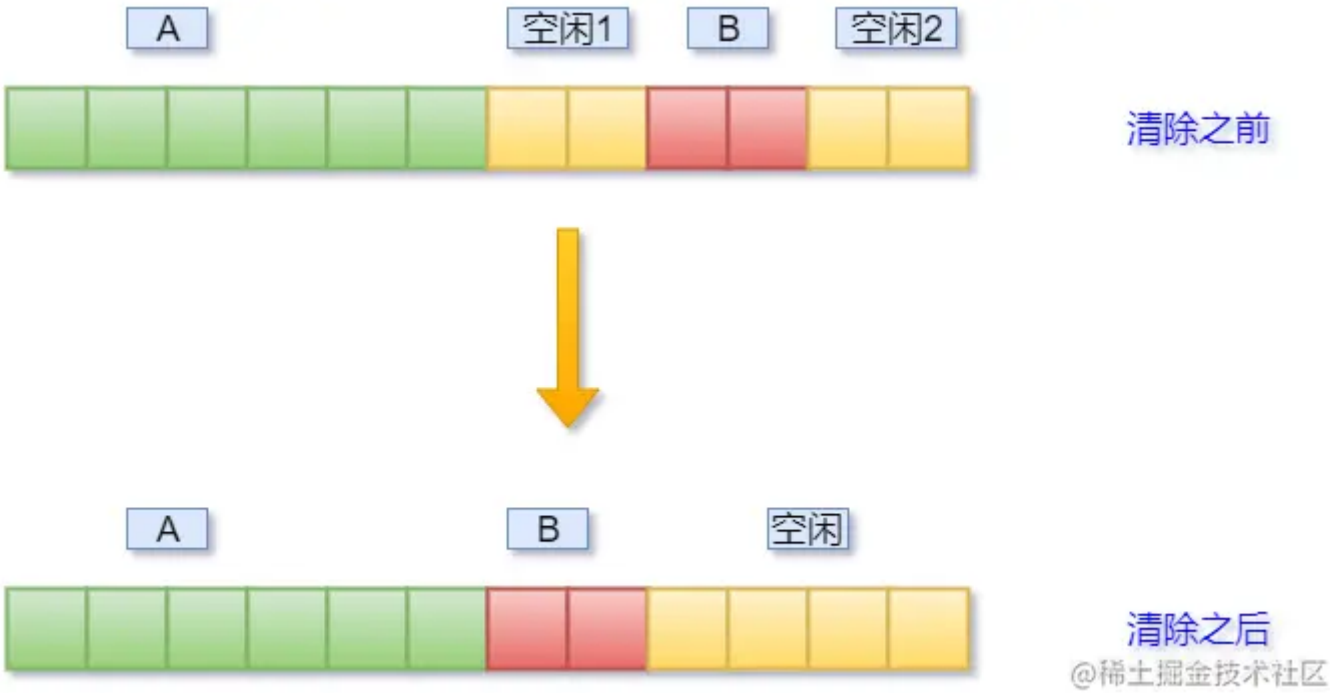


内存分配器

- redis在编译的时候可以指定内存分配器：libc、jemalloc、tcmalloc
- **jemalloc**是默认的内存分配器，因为在**减小内存碎片**方面有优势
  - 问题：Redis键值对大小不确定，会产生很多的内存碎片；
  - jemalloc在64位系统中会分的很细，可以选择最合适的内存快进行存储

Category	Spacing	Size
Small	8	[8]
	16	[16, 32, 48, ..., 128]
	32	[160, 192, 224, 256]
	64	[320, 384, 448, 512]
	128	[640, 768, 896, 1024]
	256	[1280, 1536, 1792, 2048]
	512	[2560, 3072, 3584]
Large	4 KiB	[4 KiB, 8 KiB, 12 KiB, ..., 4072 KiB]
Huge	4 MiB	[4 MiB, 8 MiB, 12 MiB, ...]

  - 文章
- Redis第四弹，删除了大量数据后，为什么内存占用还是很高？
  - 解决：



实战问题

热key

- 问题：大量请求访问某个key，流量过大，导致redis宕机
- 解决方案：
  - 增加分片副本，均衡流量
  - 将结果缓存到本地
  - 将热key分散到不同的服务器中

大key

- 问题：key的**value**特别大，String值大于10KB，Hash/List/Set/Zset元素超过5000个；key相较于value还多一个做hashcode和比较的过程（链表中进行遍历比较key）
- 影响：
  - 序列化与反序列化过程中花费的时间很大
  - Redis核心工作线程是单线程，任务是串行执行的，获取和删除影响工作线程导致后续任务阻塞
  - 客户端阻塞，很久没有执行完，客户端很久没有反应
  - 对AOF、RDB持久化的影响
  - 分片集群时候导致数据倾斜
- 如何寻找：

- redis-cli -bigkeys：只能找到Top1的大key

```
1 redis-cli -h xxx --bigkeys
```

- SCAN命令
- redis-rdb-tools

```
1 rdb dump.rdb -c memory --bytes 10240 -f redis.csv
```

- 如何解决：
  - Redis4.0之前（渐进式删除）：
    - DEL命令阻塞，使用SCAN命令，每次延续之前的迭代过程
  - Redis4.0之后（惰性删除）：
    - UNLINK命令安全地删除，异步执行，非阻塞方式
  - 压缩key：
    - 序列化、压缩算法压缩string；但是序列化和反序列化存在时间消耗
  - 拆分key：
    - 压缩后仍然是大key，需要拆分，然后用事务读取多个部分的key
    - 进行分片，不同元素计算后分到不同的片
- 参考：

a. 大key问题

  - 大key问题讲解
  - 原因：一个歌单的收藏人有谁（value是一个list，很大）
  - 影响：Redis执行命令是单线程的，客户端超时，并发量下降
  - 解决：
    - 可删除：>4.0渐进式删除，<4.0惰性删除
    - 不可删除：压缩、拆分

大key对于持久化的影响

- AOF
  - AOF日志有三种策略

- Always 策略就是每次写入 AOF 文件数据后，就执行 fsync() 函数；**（所以只有Always时候会影响持久化）**
- Everysec 策略就会创建一个异步任务来执行 fsync() 函数；
- No 策略就是永不执行 fsync() 函数;

- AOF重写、RDB快照
  - 分别通过 `fork()` 函数创建一个子进程来处理任务，**创建页表、写时复制时候阻塞父进程**

分布式锁

- Conception：

a. 相关命令

b. Redis、MySQL、Zookeeper分布式锁

c. RedLock

- Question：

a. 分布式锁实现?

- 互斥：同一时刻只能有一个线程获得锁。
- 可重入：当一个线程获取锁后，还可以再次获取这个锁，避免死锁发生。
- 高可用：当小部分节点挂掉后，仍然能够对外提供服务。
- 高性能：要做到高并发、低延迟。
- 支持阻塞和非阻塞：Synchronized是阻塞的，ReentrantLock.tryLock()就是非阻塞的
- 支持公平锁和非公平锁：Synchronized是非公平锁，ReentrantLock(boolean fair)可以创建公平锁

b. Redis实现分布式锁

- 一个进程中的多个线程竞争某一资源时候，加锁实现并发安全

```
1 SETNX try_lock 1
2 DEL try_lock
3 EXPIRE
```

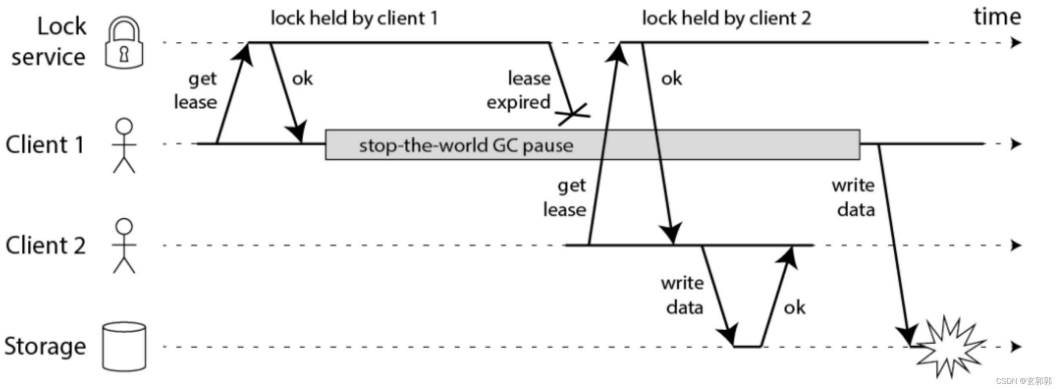
- 增加**过期时间**：一段时间后自动被删除，避免死锁，加锁的客户端宕机，try_lock一直在redis中
- **看门狗自动续时**：检测是否还在执行业务，如果还在执行，重新设置过期时间，避免业务流程长，过期时间到了还没处理完，锁就被释放了
- RedLock：至少有5个实例，加锁成本过重

- 客户端获取当前时间()
- 向N个Redis节点执行加锁策略
- 完成了和所有节点的加锁策略，计算整个加锁过程时间消耗

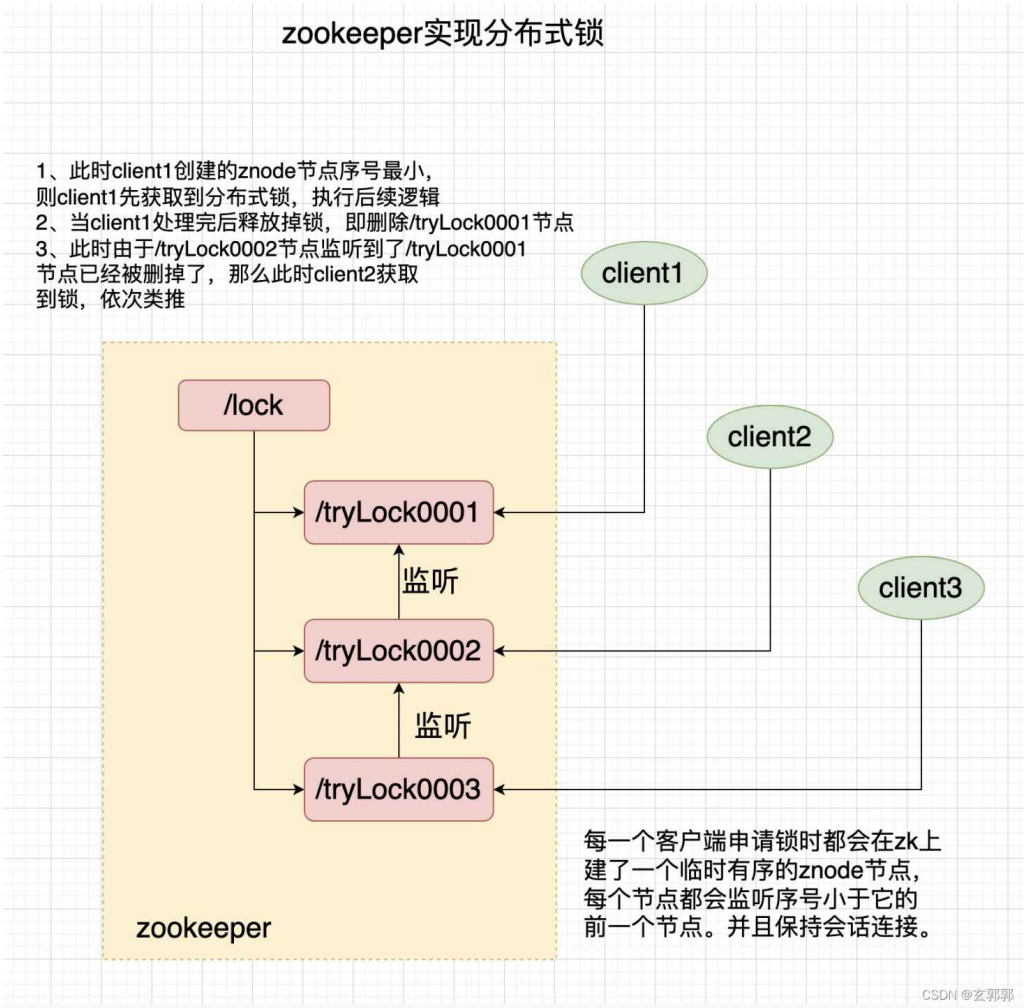
- 加锁成功条件：
  - 超过半数以上Redis节点（>= N/2 + 1）成功获取到了锁
  - 客户端获取总耗时没有超过所有的有效时间

- 加锁成功：重新计算这把锁的有效时间，有效时间 = 设置的有效时间 - 获取的锁的时间

- 分布式系统中的NPC问题：2个客户端持有锁，做业务上的兜底



- zookeeper实现分布式锁：监听获得锁



	Redis	zookeeper
优点	性能好，天然支持高并发	不用设置过期时间，监听机制等待锁
缺点	获取锁失败需要轮询操作；大多数情况redis无法保证数据强一致性	性能不如redis；网络不稳定时候，多个节点同时获得锁

- 实际使用

```
SET lock_key unique_value NX PX 10000
lock_key 就是 key 键；
unique_value 是客户端生成的唯一的标识，区分来自不同客户端的锁操作；
NX 代表只在 lock_key 不存在时，才对 lock_key 进行设置操作；
PX 10000 表示设置 lock_key 的过期时间为 10s，这是为了避免客户端发生异常而无法释放锁。
```

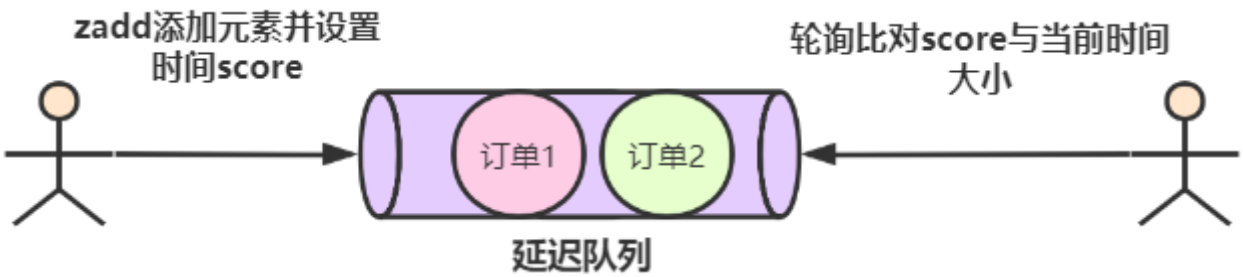
Redis解决超卖问题

- 场景：
  - 此时商品A库存为1件，当多个用户同时进行购买时，同时读到了当前的库存为1，于是都被允许下单，扣减库存，从而使库存为负数，导致超卖。
  - redis是单线程运行的所以任务不会出现线程不安全问题
- 解决：
  - 悲观锁：在操作数据之前先获取锁，确保线程串行执行
    - 优点：简单粗暴
    - 缺点：性能一般
  - 乐观锁：不加锁，在更新数据时去判断有没有其它线程对数据做了修改
    - 优点：性能好
    - 缺点：存在成功率低的问题

Redis实现延迟队列

- 场景：
  - 电商，超过一段时间后未付款
  - 网约车/外卖，规定时间内未接单，取消订单
- 实现：Zset用Score属性存储延迟执行的时间

```
1 zadd score1 value1 命令就可以一直往内存中生产消息
2 zrangebysocre 查询符合条件的所有待处理的任务
```



管道技术

- 好处：解决多个命令串行执行的网络等待，把多个命令整合到一起发送给服务端处理后统一返回给客户端
- 问题：注意避免发送的命令过大，或管道内的数据太多而导致的网络阻塞