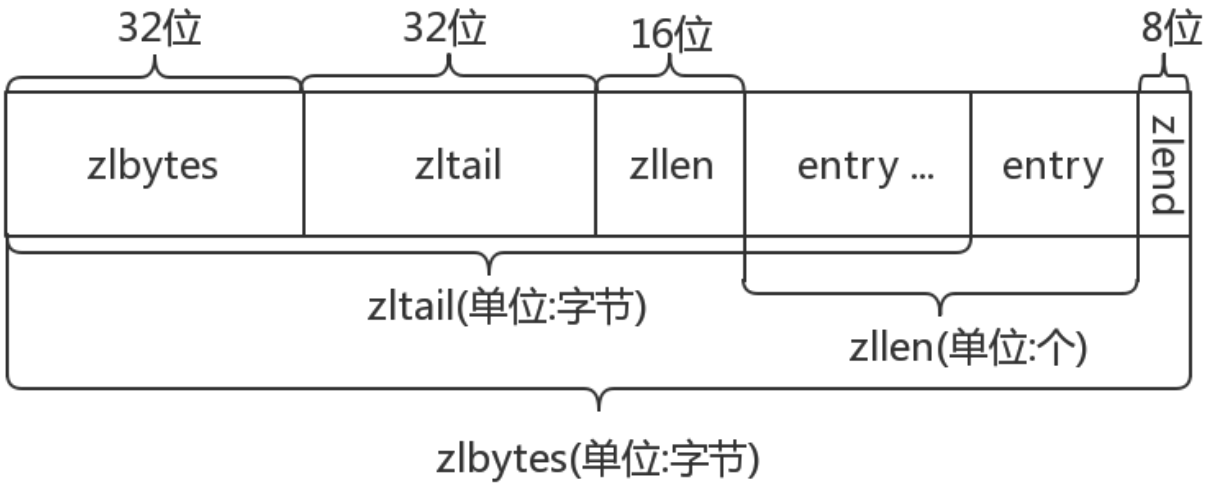
1.为什么要使用缓存
高性能：

假设这么个场景，你有个操作，一个请求过来，吭哧吭哧你各种乱七八糟操作 mysql，半天查出来一个结果，耗时 600ms。但是这个结果可能接下来几个小时都不会变了，或者变了也可以不用立即反馈给用户。那么此时咋办?
缓存啊，折腾 600ms 查出来的结果，扔缓存里，一个 key 对应一个 value，下次再有人查，别走 mysql 折腾 600ms 了，直接从缓存里，通过一个 key 查出来一个 value，2ms 搞定。性能提升 300 倍。
就是说对于一些需要复杂操作耗时查出来的结果，且确定后面不怎么变化，但是有很多读请求，那么直接将查询出来的结果放在缓存中，后面直接读缓存就好。

高并发：
mysql 这么重的数据库，压根儿设计不是让你玩儿高并发的，虽然也可以玩儿，但是天然支持不好。mysql 单机支撑到 2000QPS 也开始容易报警了。
所以要是你有个系统，高峰期一秒钟过来的请求有 1万，那一个 mysql 单机绝对会死掉。你这个时候就只能上缓存，把很多数据放缓存，别放 mysql。缓存功能简单，说白了就是 key-value 式操作，单机支撑的并发量轻松一秒几万十几万，支撑高并发 so easy。单机承载并发量是 mysql 单机的几十倍。

2.使用缓存之后出现的问题：
缓存和数据库双写不一致
https://github.com/doocs/advanced-java/blob/master/docs/high-concurrency/redis-consistence.md
缓存雪崩、缓存穿透、缓存击穿
1.缓存雪崩：redis服务宕机，大量并发的请求打到数据，把数据库压垮，解决
事前：redis 高可用，主从+哨兵，redis cluster，避免全盘崩溃。
事中：本地 ehcache 缓存 + hystrix 限流&降级，避免 MySQL 被打死。
事后：redis 持久化，一旦重启，自动从磁盘上加载数据，快速恢复缓存数据。
2.缓存穿透：恶意攻击，解决
使用redis返回空结果 或者使用布隆过滤器
3.缓存击穿：单个热点key，在缓存失效的时候，大量请求击穿了缓存到数据库，解决
1.缓存基本不怎么更新的场景，永不过期
2.更新不频繁，更新流水好使很短的场景，可以使用分布式锁
3.缓存跟新频繁或者缓存更新耗时的场景，可以使用额外的任务或者监听者在缓存快过期时候主动跟新缓存，延迟过期时间

3.redis的数据结构和使用场景
string
hash 用户的购物车列表
list 用户的粉丝列表、文章的评论列表lrange 分页查询、可以搞个简单的消息队列
set 去重操作，可以做微信共同的好友、粉丝功能
sorted set 去重但可以排序 排行榜、热搜榜之类的功能
Ziplist:



4.为什么使用单线程

1.使用单线程模型能带来更好的可维护性，方便开发和调试；
2.使用单线程模型也能并发的处理客户端的请求；
3.Redis 服务中运行的绝大多数操作的性能瓶颈都不是 CPU；而是网络 I/O

5.redis的高可用

https://www.cnblogs.com/mzhaox/p/11218096.html

持久化、复制、哨兵 和 集群

持久化：持久化是 最简单的 高可用方法。它的主要作用是 数据备份，即将数据存储在 硬盘，保证数据不会因进程退出而丢失。

复制：复制是高可用 Redis 的基础，哨兵 和 集群 都是在 复制基础 上实现高可用的。复制主要实现了数据的多机备份以及对于读操作的负载均衡和简单的故障恢复。缺陷是故障恢复无法自动化、写操作无法负载均衡、存储能力受到单机的限制。

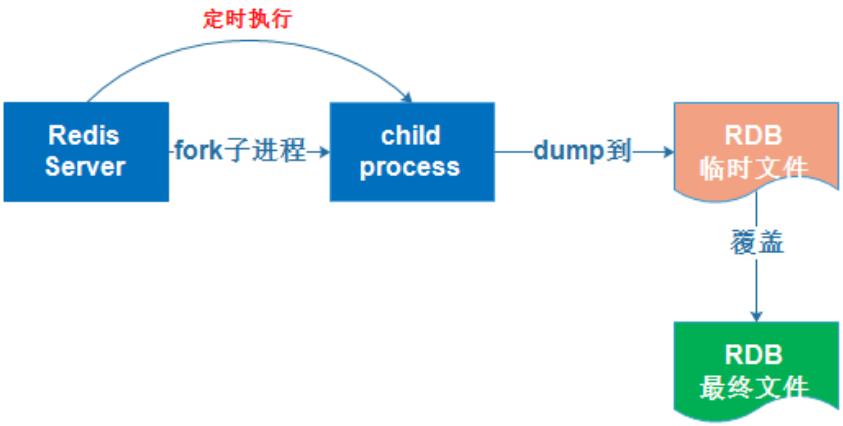
哨兵：在复制的基础上，哨兵实现了 自动化 的 故障恢复。缺陷是 写操作 无法 负载均衡，存储能力 受到 单机 的限制。

集群：通过集群，Redis 解决了 写操作 无法 负载均衡 以及 存储能力 受到 单机限制 的问题，实现了较为 完善 的 高可用方案。

Redis cluster:主要是针对海量数据+高并发+高可用的场景
分布式寻址算法:一致性hash算法、hash曹（slot）

6.redis的持久化方案

RDB持久化是指在指定的时间间隔内将内存中的数据集快照写入磁盘，实际操作过程是fork一个子进程，先将数据集写入临时文件，写入成功后，再替换之前的文件，用二进制压缩存储。



AOF持久化以日志的形式记录服务器所处理的每一个写、删除操作，查询操作不会记录，以文本的方式记录，可以打开文件看到详细的操作记录。

7.redis的过期策略

定期删除

100ms随机找一些过期的key进行删除

惰性删除

没有被定期删除的如果被访问到了并且发现过期了，也会被删除

内存淘汰机制

最长常用的就是LRU

LRU算法

```
class LRUCache<K, V> extends LinkedHashMap<K, V> {
    private final int CACHE_SIZE;
    /**
     * 传递进来最多能缓存多少数据
     *
     * @param cacheSize 缓存大小
     */
    public LRUCache(int cacheSize) {
        // true 表示让 linkedHashMap 按照访问顺序来进行排序，最近访问的放在头部，最老访问的放在尾部。
        super((int) Math.ceil(cacheSize / 0.75) + 1, 0.75f, true);
        CACHE_SIZE = cacheSize;
    }
    @Override
    protected boolean removeEldestEntry(Map.Entry<K, V> eldest) {
        // 当 map中的数据量大于指定的缓存个数的时候，就自动删除最老的数据。
        return size() > CACHE_SIZE;
    }
}
```