> 你知道的越多，你不知道的越多

> 点赞再看，养成习惯

> 本文 **GitHub** https://github.com/JavaFamily 上已经收录，有一线大厂面试点思维导图，也整理了很多我的文档，欢迎Star和完善，大家面试可以参照考点复习，希望我们一起有点东西。

前言

作为一个在互联网公司面一次拿一次Offer的面霸，打败了无数竞争对手，每次都只能看到无数落寞的身影失望的离开，略感愧疚（**请允许我使用一下夸张的修辞手法**）。

于是在一个寂寞难耐的夜晚，我痛定思痛，决定开始写互联网技术栈面试相关的文章，希望能帮助各位读者以后面试势如破竹，对面试官进行360°的反击，吊打问你的面试官，让一同面试的同僚瞠目结舌，疯狂收割大厂Offer！

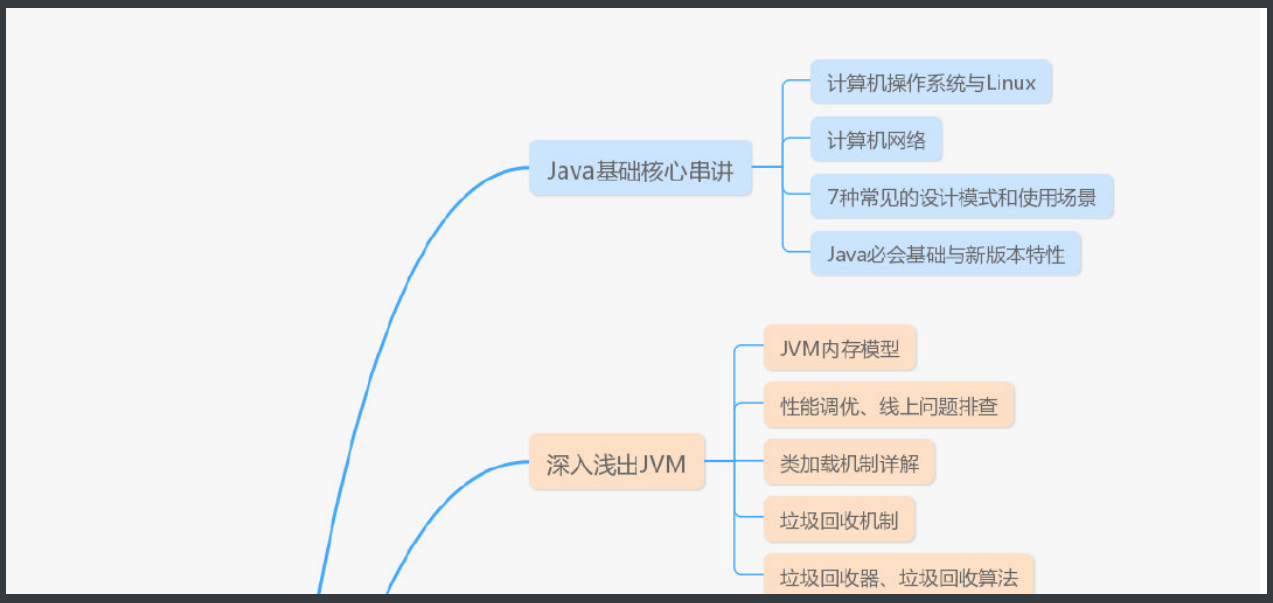
所有文章的名字只是我的噱头，我们应该有一颗谦逊的心，所以希望大家怀着空杯心态好好学，一起进步。

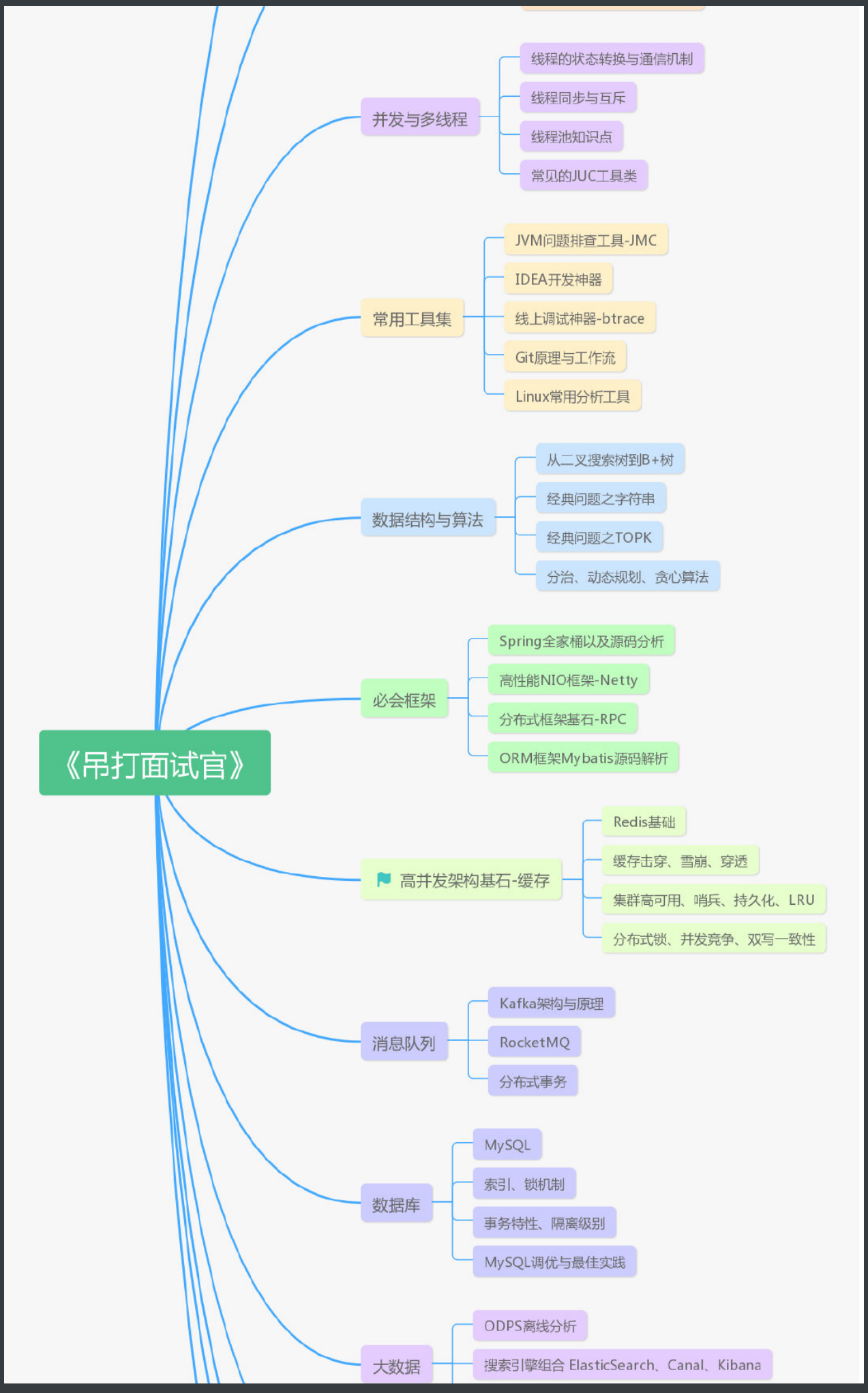
絮叨

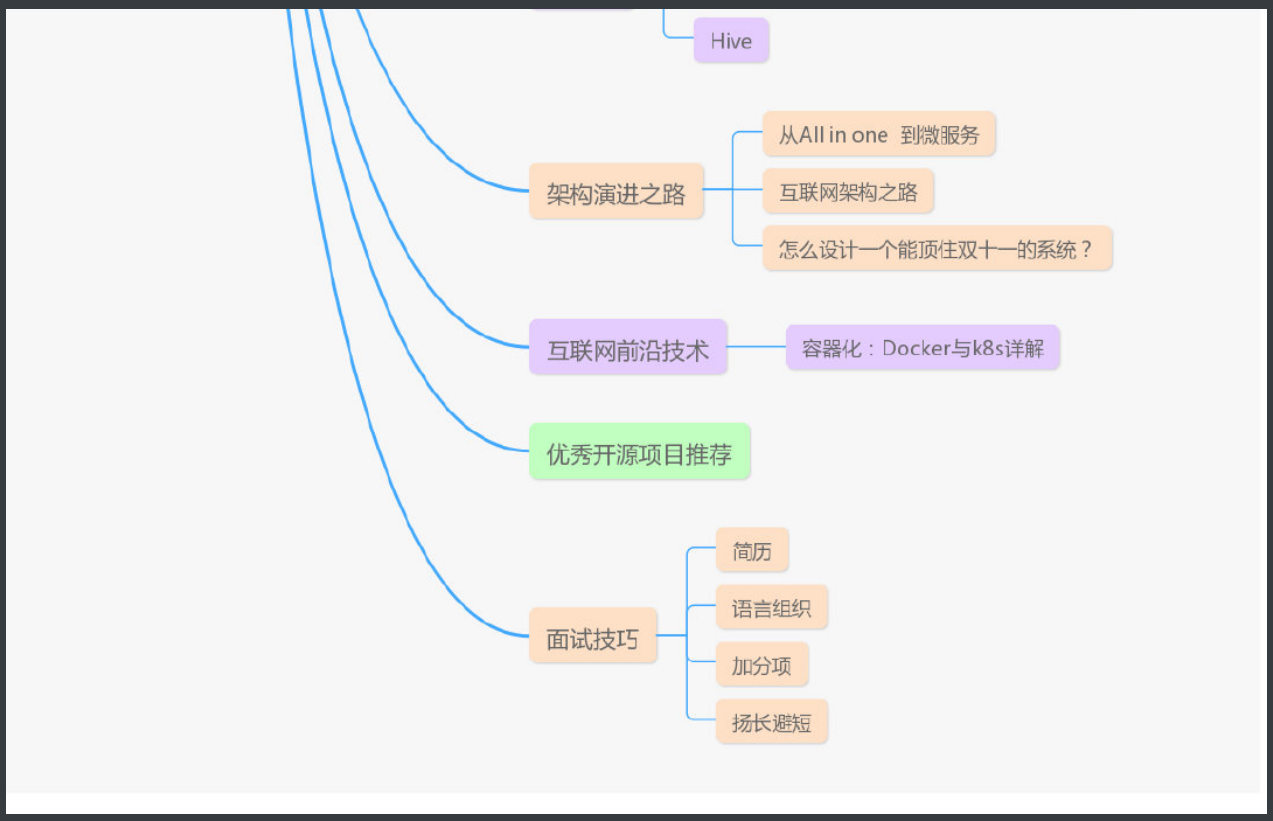
上一期因为是在双十一一直在熬夜的大环境下完成的，所以我自己觉得质量明显没之前的好，我这不一睡好就加班加点准备补偿大家，来点干货。（熬夜太容易感冒了，这次**点个赞**别白嫖了！）

顺带提一嘴，我把我准备写啥画了一个思维导图，以后总不能每篇都放个贼大的图吧，就开源到了我的**GitHub**，大家有兴趣可以去完善和**Star**。

这篇我就先放出来大家看看，感觉还是差点意思，等大家完善了。




《吊打面试官》
并发与多线程
线程的状态转换与通信机制
线程同步与互斥
线程池知识点
常见的JUC工具类
常用工具集
JVM问题排查工具-JMC
IDEA开发神器
线上调试神器-btrace
Git原理与工作流
Linux常用分析工具
数据结构与算法
从二叉搜索树到B+树
经典问题之字符串
经典问题之TOPK
分治、动态规划、贪心算法
必会框架
Spring全家桶以及源码分析
高性能NIO框架-Netty
分布式框架基石-RPC
ORM框架Mybatis源码解析
高并发架构基石-缓存
Redis基础
缓存击穿、雪崩、穿透
集群高可用、哨兵、持久化、LRU
分布式锁、并发竞争、双写一致性
消息队列
Kafka架构与原理
RocketMQ
分布式事务
数据库
MySQL
索引、锁机制
事务特性、隔离级别
MySQL调优与最佳实践
大数据
ODPS离线分析
搜索引擎组合 ElasticSearch、Canal、Kibana



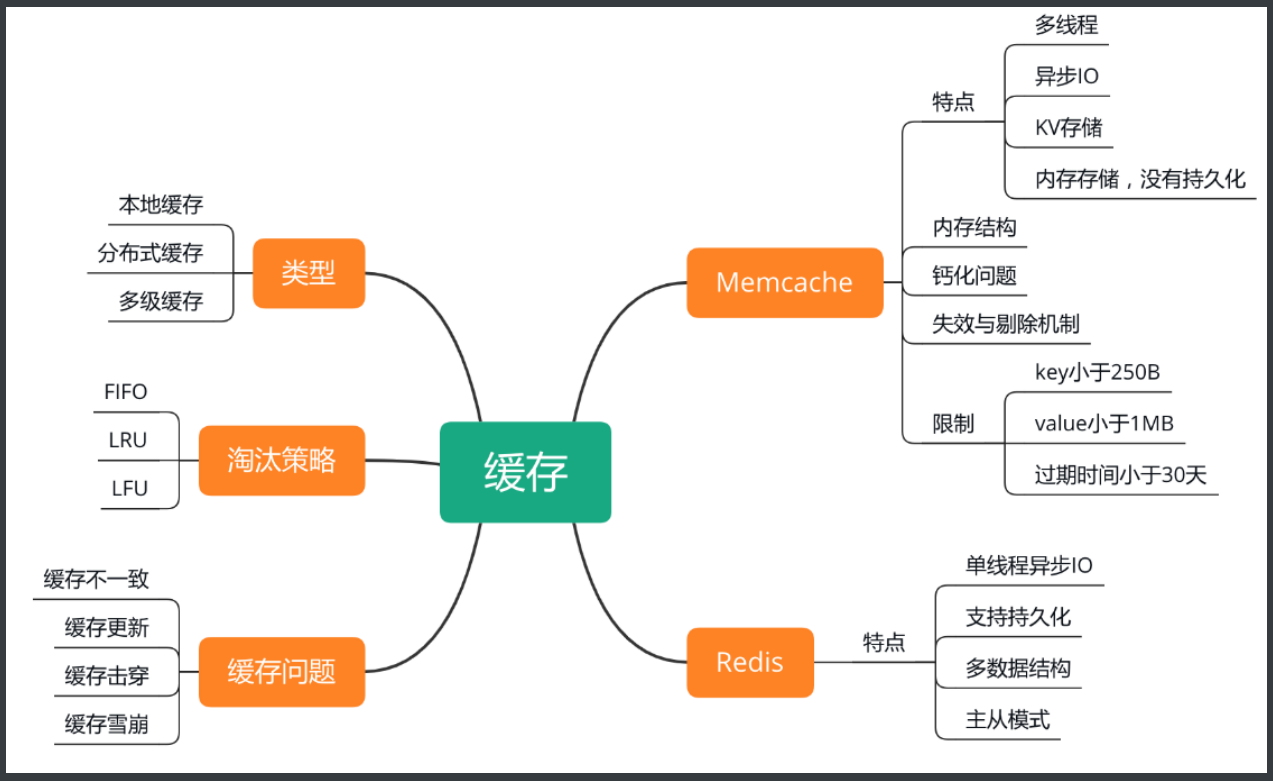
回望过去

上一期吊打系列我们提到了**Redis**相关的一些知识，还没看的小伙伴可以回顾一下

- 《吊打面试官》系列-Redis基础
- 《吊打面试官》系列-缓存雪崩、击穿、穿透
- 《吊打面试官》系列-Redis哨兵、持久化、主从、手撕LRU
- 《吊打面试官》系列-Redis终章-凛冬将至、FPX-新王登基

这期我就从缓存到一些常见的问题讲一下，有一些我是之前提到过的，不过可能大部分仔是第一次看，我就重复发一下。

缓存知识点



缓存有哪些类型?

缓存是高并发场景下提高热点数据访问性能的一个有效手段，在开发项目时会经常使用到。

缓存的类型分为：**本地缓存**、**分布式缓存**和**多级缓存**。

本地缓存：

本地缓存就是在进程的内存中进行缓存，比如我们的 **JVM** 堆中，可以用 **LRUMap** 来实现，也可以使用 **Ehcache** 这样的工具来实现。

本地缓存是内存访问，没有远程交互开销，性能最好，但是受限于单机容量，一般缓存较小且无法扩展。

分布式缓存：

分布式缓存可以很好得解决这个问题。

分布式缓存一般都具有良好的水平扩展能力，对较大数据量的场景也能应付自如。缺点就是需要进行远程请求，性能不如本地缓存。

多级缓存：

为了平衡这种情况，实际业务中一般采用**多级缓存**，本地缓存只保存访问频率最高的部分热点数据，其他的热点数据放在分布式缓存中。

在目前的一线大厂中，这也是最常用的缓存方案，单考单一的缓存方案往往难以撑住很多高并发的场景。

淘汰策略

不管是本地缓存还是分布式缓存，为了保证较高性能，都是使用内存来保存数据，由于成本和内存限制，当存储的数据超过缓存容量时，需要对缓存的数据进行剔除。

一般的剔除策略有 **FIFO** 淘汰最早数据、**LRU** 剔除最近最少使用、和 **LFU** 剔除最近使用频率最低的数据几种策略。

- **noeviction**:返回错误当内存限制达到并且客户端尝试执行会让更多内存被使用的命令（大部分的写入指令，但DEL和几个例外）
- **allkeys-lru**: 尝试回收最少使用的键（LRU），使得新添加的数据有空间存放。
- **volatile-lru**: 尝试回收最少使用的键（LRU），但仅限于在过期集合的键,使得新添加的数据有空间存放。
- **allkeys-random**: 回收随机的键使得新添加的数据有空间存放。
- **volatile-random**: 回收随机的键使得新添加的数据有空间存放，但仅限于在过期集合的键。
- **volatile-ttl**: 回收在过期集合的键，并且优先回收存活时间（TTL）较短的键,使得新添加的数据有空间存放。

  如果没有键满足回收的前提条件的话，策略**volatile-lru**, **volatile-random**以及**volatile-ttl**就和 noeviction 差不多了。

其实在大家熟悉的**LinkedHashMap**中也实现了Lru算法的，实现如下：

```
final Map<Long, TimeoutInfoHolder> timeoutInfoHandlers =
        Collections.synchronizedMap(new LinkedHashMap<Long, TimeoutInfoHolder>(100, .75F, true) {
            @Override
            protected boolean removeEldestEntry(Map.Entry eldest) {
                return size() > 100;
            }
        });
```

当容量超过100时，开始执行**LRU**策略：将最近最少未使用的 **TimeoutInfoHolder** 对象 **evict** 掉。

真实面试中会让你写LUR算法，你可别搞原始的那个，那真TM多，写不完的，你要么怼上面这个，要么怼下面这个，找一个数据结构实现下Java版本的LRU还是比较容易的，知道啥原理就好了。

```java
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;
    }
}
```

Memcache

注意后面会把 **Memcache** 简称为 MC。

先来看看 MC 的特点：

- MC 处理请求时使用多线程异步 IO 的方式，可以合理利用 CPU 多核的优势，性能非常优秀；
- MC 功能简单，使用内存存储数据；
- MC 的内存结构以及钙化问题我就不细说了，大家可以查看官网了解下；
- MC 对缓存的数据可以设置失效期，过期后的数据会被清除；
- 失效的策略采用延迟失效，就是当再次使用数据时检查是否失效；
- 当容量存满时，会对缓存中的数据进行剔除，剔除时除了会对过期 key 进行清理，还会按 LRU 策略对数据进行剔除。

另外，使用 MC 有一些限制，这些限制在现在的互联网场景下很致命，成为大家选择 **Redis**、**MongoDB**的重要原因：

- key 不能超过 250 个字节；
- value 不能超过 1M 字节；
- key 的最大失效时间是 30 天；
- 只支持 K-V 结构，不提供持久化和主从同步功能。

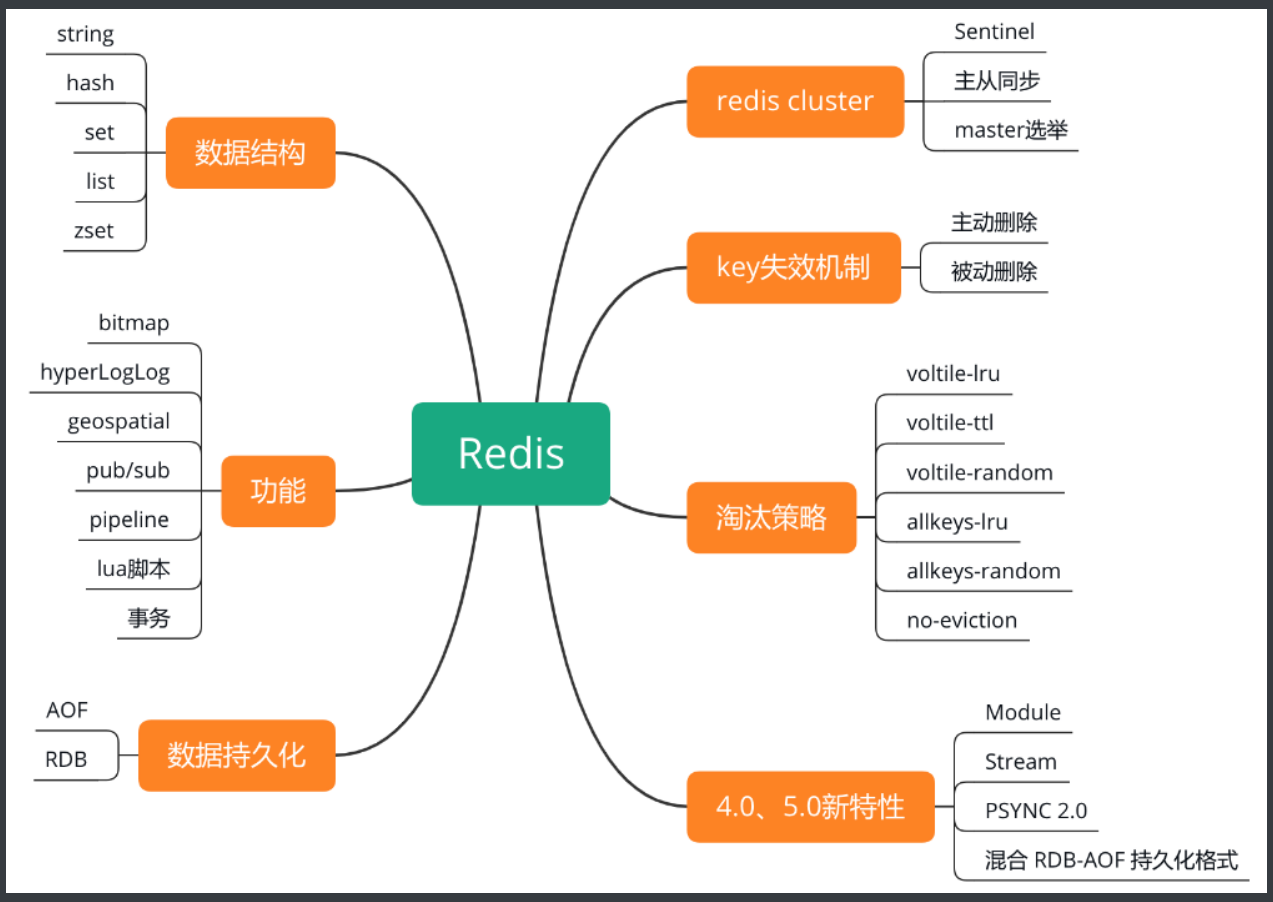
Redis

先简单说一下 **Redis** 的特点，方便和 MC 比较。

- 与 MC 不同的是，Redis 采用单线程模式处理请求。这样做的原因有 2 个：一个是因为采用了非阻塞的异步事件处理机制；另一个是缓存数据都是内存操作 IO 时间不会太长，单线程可以避免线程上下文切换产生的代价。
- **Redis** 支持持久化，所以 Redis 不仅仅可以用作缓存，也可以用作 NoSQL 数据库。
- 相比 MC，**Redis** 还有一个非常大的优势，就是除了 K-V 之外，还支持多种数据格式，例如 list、set、sorted set、hash 等。
- **Redis** 提供主从同步机制，以及 **Cluster** 集群部署能力，能够提供高可用服务。

详解 Redis

Redis 的知识点结构如下图所示。



功能

来看 **Redis** 提供的功能有哪些吧！

我们先看基础类型：

String：

String 类型是 **Redis** 中最常使用的类型，内部的实现是通过 **SDS**（Simple Dynamic String ）来存储的。SDS 类似于 **Java** 中的 **ArrayList**，可以通过预分配冗余空间的方式来减少内存的频繁分配。

这是最简单的类型，就是普通的 set 和 get，做简单的 KV 缓存。

但是真实的开发环境中，很多仔可能会把很多比较复杂的结构也统一转成**String**去存储使用，比如有的仔他就喜欢把对象或者**List**转换为**JSONString**进行存储，拿出来再反序列话啥的。

我在这里就不讨论这样做的对错了，但是我还是希望大家能在最合适的场景使用最合适的数据结构，对象找不到最合适的但是类型可以选最合适的嘛，之后别人接手你的代码一看这么**规范**，诶这小伙子**有点东西**呀，看到你啥都是用的**String**，**垃圾！**

好了这些都是题外话了，道理还是希望大家记在心里，习惯成自然嘛，小习惯成就你。

String的实际应用场景比较广泛的有：

- **缓存功能：String**字符串是最常用的数据类型，不仅仅是**Redis**，各个语言都是最基本类型，因此，利用**Redis**作为缓存，配合其它数据库作为存储层，利用**Redis**支持高并发的特点，可以大大加快系统的读写速度、以及降低后端数据库的压力。
- **计数器：**许多系统都会使用**Redis**作为系统的实时计数器，可以快速实现计数和查询的功能。而且最终的数据结果可以按照特定的时间落地到数据库或者其它存储介质当中进行永久保存。
- **共享用户Session：**用户重新刷新一次界面，可能需要访问一下数据进行重新登录，或者访问页面缓存**Cookie**，但是可以利用**Redis**将用户的**Session**集中管理，在这种模式只需要保证**Redis**的高可用，每次用户**Session**的更新和获取都可以快速完成。大大提高效率。

Hash：

这个是类似 **Map** 的一种结构，这个一般就是可以将结构化的数据，比如一个对象（前提是**这个对象没嵌套其他的对象**）给缓存在 **Redis** 里，然后每次读写缓存的时候，可以就操作 **Hash** 里的**某个字段**。

但是这个的场景其实还是多少单一了一些，因为现在很多对象都是比较复杂的，比如你的商品对象可能里面就包含了很多属性，其中也有对象。我自己使用的场景用得不是那么多。

List：

List 是有序列表，这个还是可以玩儿出很多花样的。

比如可以通过 **List** 存储一些列表型的数据结构，类似粉丝列表、文章的评论列表之类的东西。

比如可以通过 **lrange** 命令，读取某个闭区间内的元素，可以基于 **List** 实现分页查询，这个是很棒的一个功能，基于 **Redis** 实现简单的高性能分页，可以做类似微博那种下拉不断分页的东西，性能高，就一页一页走。

比如可以搞个简单的消息队列，从 **List** 头怼进去，从 **List** 屁股那里弄出来。

List本身就是我们在开发过程中比较常用的数据结构了，热点数据更不用说了。

- **消息队列：Redis**的链表结构，可以轻松实现阻塞队列，可以使用左进右出的命令组成来完成队列的设计。比如：数据的生产者可以通过**Lpush**命令从左边插入数据，多个数据消费者，可以使用**BRpop**命令阻塞的“抢”列表尾部的数据。
- 文章列表或者数据分页展示的应用。

  比如，我们常用的博客网站的文章列表，当用户量越来越多时，而且每一个用户都有自己的文章列表，而且当文章多时，都需要分页展示，这时可以考虑使用**Redis**的列表，列表不但有序同时还支持按照范围内获取元素，可以完美解决分页查询功能。大大提高查询效率。

Set：

Set 是无序集合，会自动去重的那种。

直接基于 **Set** 将系统里需要去重的数据扔进去，自动就给去重了，如果你需要对一些数据进行快速的全局去重，你当然也可以基于 **JVM** 内存里的 **HashSet** 进行去重，但是如果你的某个系统部署在多台机器上呢？得基于**Redis**进行全局的 **Set** 去重。

可以基于 **Set** 玩儿交集、并集、差集的操作，比如交集吧，我们可以把两个人的好友列表整一个交集，看看俩人的共同好友是谁？对吧。

反正这些场景比较多，因为对比很快，操作也简单，两个查询一个**Set**搞定。

Sorted Set：

Sorted set 是排序的 **Set**，去重但可以排序，写进去的时候给一个分数，自动根据分数排序。

有序集合的使用场景与集合类似，但是set集合不是自动有序的，而**Sorted set**可以利用分数进行成员间的排序，而且是插入时就排序好。所以当你需要一个有序且不重复的集合列表时，就可以选择**Sorted set**数据结构作为选择方案。

- 排行榜：有序集合经典使用场景。例如视频网站需要对用户上传的视频做排行榜，榜单维护可能是多方面：按照时间、按照播放量、按照获得的赞数等。

- 用**Sorted Sets**来做带权重的队列，比如普通消息的score为1，重要消息的score为2，然后工作线程可以选择按score的倒序来获取工作任务。让重要的任务优先执行。

  微博热搜榜，就是有个后面的热度值，前面就是名称

高级用法：

Bitmap：

位图是支持按 bit 位来存储信息，可以用来实现 **布隆过滤器（BloomFilter）**；

HyperLogLog:

供不精确的去重计数功能，比较适合用来做大规模数据的去重统计，例如统计 UV；

Geospatial:

可以用来保存地理位置，并作位置距离计算或者根据半径计算位置等。有没有想过用Redis来实现附近的人？或者计算最优地图路径?

这三个其实也可以算作一种数据结构，不知道还有多少朋友记得，我在梦开始的地方，Redis基础中提到过，你如果只知道五种基础类型那只能拿60分，如果你能讲出高级用法，那就觉得你**有点东西**。

pub/sub：

功能是订阅发布功能，可以用作简单的消息队列。

Pipeline：

可以批量执行一组指令，一次性返回全部结果，可以减少频繁的请求应答。

Lua：

Redis 支持提交 **Lua** 脚本来执行一系列的功能。

我在前电商老东家的时候，秒杀场景经常使用这个东西，讲道理有点香，利用他的原子性。

话说你们想看秒杀的设计么？我记得我面试好像每次都问啊，想看的直接**点赞**后评论秒杀吧。

事务：

最后一个功能是事务，但 **Redis** 提供的不是严格的事务，**Redis** 只保证串行执行命令，并且能保证全部执行，但是执行命令失败时并不会回滚，而是会继续执行下去。

持久化

Redis 提供了 RDB 和 AOF 两种持久化方式，RDB 是把内存中的数据集以快照形式写入磁盘，实际操作是通过 fork 子进程执行，采用二进制压缩存储；AOF 是以文本日志的形式记录 **Redis** 处理的每一个写入或删除操作。

RDB 把整个 Redis 的数据保存在单一文件中，比较适合用来做灾备，但缺点是快照保存完成之前如果宕机，这段时间的数据将会丢失，另外保存快照时可能导致服务短时间不可用。

AOF 对日志文件的写入操作使用的追加模式，有灵活的同步策略，支持每秒同步、每次修改同步和不同步，缺点就是相同规模的数据集，AOF 要大于 RDB，AOF 在运行效率上往往会慢于 RDB。

细节的点大家去高可用这章看，特别是两者的优缺点，以及怎么抉择。

《吊打面试官》系列-Redis哨兵、持久化、主从、手撕LRU

高可用

来看 Redis 的高可用。Redis 支持主从同步，提供 Cluster 集群部署模式，通过 Sentine l哨兵来监控 Redis 主服务器的状态。当主挂掉时，在从节点中根据一定策略选出新主，并调整其他从 slaveof 到新主。

选主的策略简单来说有三个：

- slave 的 priority 设置的越低，优先级越高；
- 同等情况下，slave 复制的数据越多优先级越高；
- 相同的条件下 runid 越小越容易被选中。

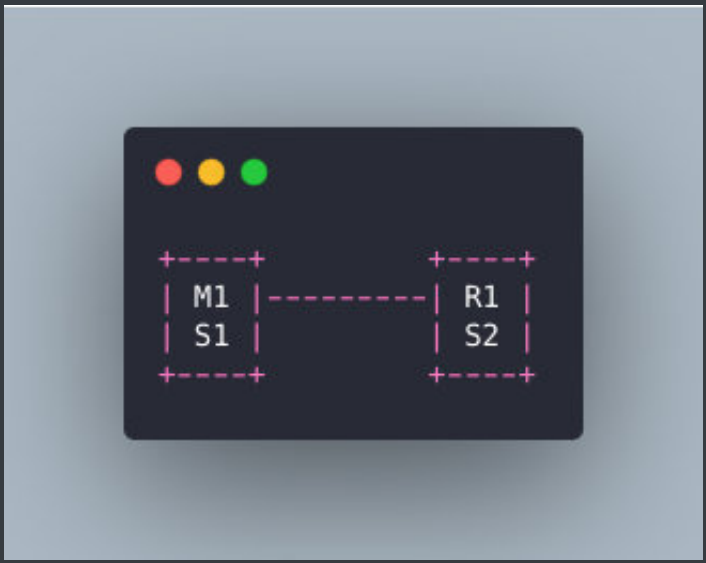
在 Redis 集群中，sentinel 也会进行多实例部署，sentinel 之间通过 Raft 协议来保证自身的高可用。

Redis Cluster 使用分片机制，在内部分为 16384 个 slot 插槽，分布在所有 master 节点上，每个 master 节点负责一部分 slot。数据操作时按 key 做 CRC16 来计算在哪个 slot，由哪个 master 进行处理。数据的冗余是通过 slave 节点来保障。

哨兵

哨兵必须用三个实例去保证自己的健壮性的，哨兵+主从并**不能保证数据不丢失**，但是可以保证集群的**高可用**。

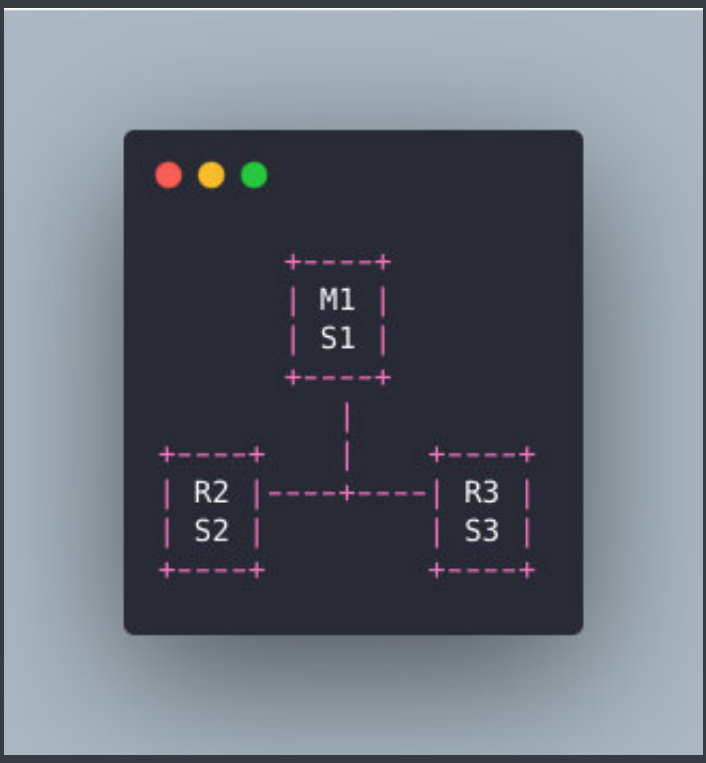
为啥必须要三个实例呢？我们先看看两个哨兵会咋样。



master宕机了 s1和s2两个哨兵只要有一个认为你宕机了就切换了，并且会选举出一个哨兵去执行故障，但是这个时候也需要大多数哨兵都是运行的。

那这样有啥问题呢？M1宕机了，S1没挂那其实是OK的，但是整个机器都挂了呢？哨兵就只剩下S2个裸屌了，没有哨兵去允许故障转移了，虽然另外一个机器上还有R1，但是故障转移就是不执行。

经典的哨兵集群是这样的：



M1所在的机器挂了，哨兵还有两个，两个人一看他不是挂了嘛，那我们就选举一个出来执行故障转移不就好了。

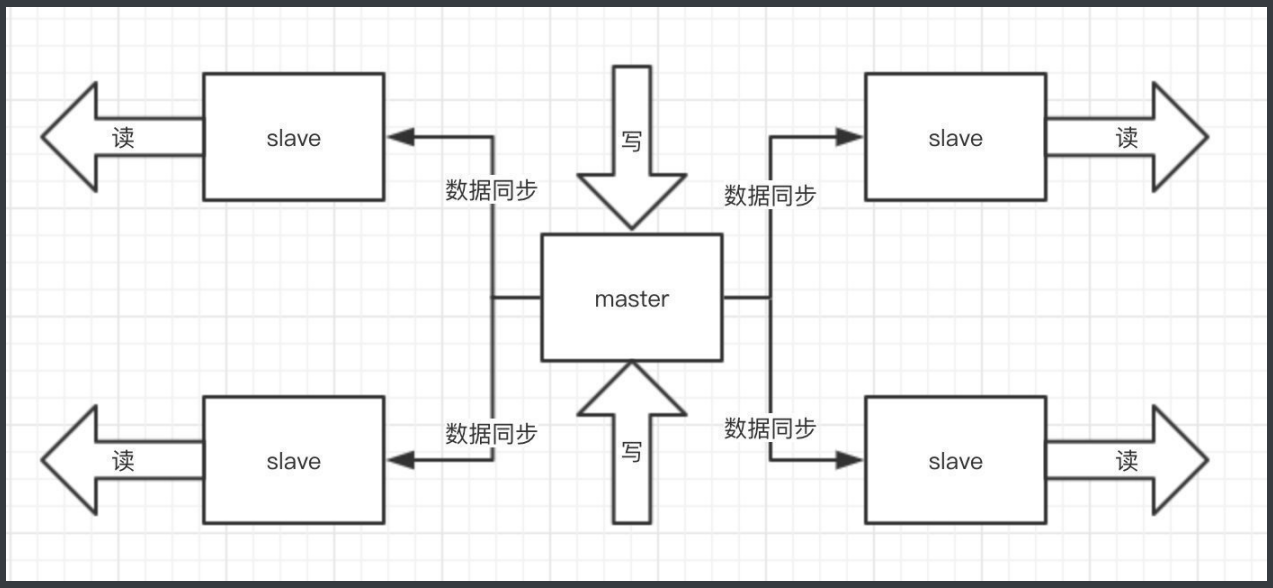
暖男我，小的总结下哨兵组件的主要功能：

- 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。

主从

提到这个，就跟我前面提到的数据持久化的**RDB**和**AOF**有着比密切的关系了。

我先说下为啥要用主从这样的架构模式，前面提到了单机**QPS**是有上限的，而且**Redis**的特性就是必须支撑读高并发的，那你一台机器又读又写，**这谁顶得住啊**，不当人啊！但是你让这个master机器去写，数据同步给别的slave机器，他们都拿去读，分发掉大量的请求那是不是好很多，而且扩容的时候还可以轻松实现水平扩容。



你启动一台slave 的时候，他会发送一个**psync**命令给master ，如果是这个slave第一次连接到master，他会触发一个全量复制。master就会启动一个线程，生成**RDB**快照，还会把新的写请求都缓存在内存中，**RDB**文件生成后，master会将这个**RDB**发送给slave的，slave拿到之后做的第一件事情就是写进本地的磁盘，然后加载进内存，然后master会把内存里面缓存的那些新命名都发给slave。

我发出来之后来自CSDN的网友：Jian_Shen_Zer 问了个问题：

主从同步的时候，新的slaver进来的时候用**RDB**，那之后的数据呢？有新的数据进入master怎么同步到slaver啊

敖丙答：笨，**AOF**嘛，增量的就像**MySQL**的**Binlog**一样，把日志增量同步给从服务就好了

key 失效机制

Redis 的 key 可以设置过期时间，过期后 Redis 采用主动和被动结合的失效机制，一个是和 MC 一样在访问时触发被动删除，另一种是定期的主动删除。

定期+惰性+内存淘汰

缓存常见问题

缓存更新方式

这是决定在使用缓存时就该考虑的问题。

缓存的数据在数据源发生变更时需要对缓存进行更新，数据源可能是 DB，也可能是远程服务。更新的方式可以是主动更新。数据源是 DB 时，可以在更新完 DB 后就直接更新缓存。

当数据源不是 DB 而是其他远程服务，可能无法及时主动感知数据变更，这种情况下一般会选择对缓存数据设置失效期，也就是数据不一致的最大容忍时间。

这种场景下，可以选择失效更新，key 不存在或失效时先请求数据源获取最新数据，然后再次缓存，并更新失效期。

但这样做有个问题，如果依赖的远程服务在更新时出现异常，则会导致数据不可用。改进的办法是异步更新，就是当失效时先不清除数据，继续使用旧的数据，然后由异步线程去执行更新任务。这样就避免了失效瞬间的空窗期。另外还有一种纯异步更新方式，定时对数据进行分批更新。实际使用时可以根据业务场景选择更新方式。

数据不一致

第二个问题是数据不一致的问题，可以说只要使用缓存，就要考虑如何面对这个问题。缓存不一致产生的原因一般是主动更新失败，例如更新 DB 后，更新 **Redis** 因为网络原因请求超时；或者是异步更新失败导致。

解决的办法是，如果服务对耗时不是特别敏感可以增加重试；如果服务对耗时敏感可以通过异步补偿任务来处理失败的更新，或者短期的数据不一致不会影响业务，那么只要下次更新时可以成功，能保证最终一致性就可以。

缓存穿透

缓存穿透。产生这个问题的原因可能是外部的恶意攻击，例如，对用户信息进行了缓存，但恶意攻击者使用不存在的用户id频繁请求接口，导致查询缓存不命中，然后穿透 DB 查询依然不命中。这时会有大量请求穿透缓存访问到 DB。

解决的办法如下。

1. 对不存在的用户，在缓存中保存一个空对象进行标记，防止相同 ID 再次访问 DB。不过有时这个方法并不能很好解决问题，可能导致缓存中存储大量无用数据。
2. 使用 **BloomFilter** 过滤器，BloomFilter 的特点是存在性检测，如果 BloomFilter 中不存在，那么数据一定不存在；如果 BloomFilter 中存在，实际数据也有可能会不存在。非常适合解决这类的问题。

缓存击穿

缓存击穿，就是某个热点数据失效时，大量针对这个数据的请求会穿透到数据源。

解决这个问题有如下办法。

1. 可以使用互斥锁更新，保证同一个进程中针对同一个数据不会并发请求到 DB，减小 DB 压力。
2. 使用随机退避方式，失效时随机 sleep 一个很短的时间，再次查询，如果失败再执行更新。
3. 针对多个热点 key 同时失效的问题，可以在缓存时使用固定时间加上一个小的随机数，避免大量热点 key 同一时刻失效。

缓存雪崩

缓存雪崩，产生的原因是缓存挂掉，这时所有的请求都会穿透到 DB。

解决方法：

1. 使用快速失败的熔断策略，减少 DB 瞬间压力；
2. 使用主从模式和集群模式来尽量保证缓存服务的高可用。

实际场景中，这两种方法会结合使用。

老朋友都知道为啥我**没有大篇幅介绍**这个几个点了吧，我在之前的文章实在是写得太详细了，忍不住**点赞**那种，我这里就不做重复拷贝了。

- **《吊打面试官》系列-Redis基础**
- **《吊打面试官》系列-缓存雪崩、击穿、穿透**
- **《吊打面试官》系列-Redis哨兵、持久化、主从、手撕LRU**
- **《吊打面试官》系列-Redis终章*凛冬将至*、*FPX*新王登基**

考点与加分项

拿笔记一下！

考点

面试的时候问你缓存，主要是考察缓存特性的理解，对 **MC**、**Redis** 的特点和使用方式的掌握。

- 要知道缓存的使用场景，不同类型缓存的使用方式，例如：

1. - 对 DB 热点数据进行缓存减少 DB 压力；对依赖的服务进行缓存，提高并发性能；
2. - 单纯 K-V 缓存的场景可以使用 **MC**，而需要缓存 list、set 等特殊数据格式，可以使用 **Redis**；
3. - 需要缓存一个用户最近播放视频的列表可以使用 **Redis** 的 list 来保存、需要计算排行榜数据时，可以使用 **Redis** 的 zset 结构来保存。

- 要了解 MC 和 **Redis** 的常用命令，例如原子增减、对不同数据结构进行操作的命令等。
- 了解 MC 和 **Redis** 在内存中的存储结构，这对评估使用容量会很有帮助。
- 了解 MC 和 **Redis** 的数据失效方式和剔除策略，比如主动触发的定期剔除和被动触发延期剔除
- 要理解 **Redis** 的持久化、主从同步与 **Cluster** 部署的原理，比如 **RDB** 和 **AOF** 的实现方式与区别。
- 要知道缓存穿透、击穿、雪崩分别的异同点以及解决方案。
- 不管你有没有电商经验我觉得你都应该知道秒杀的具体实现，以及细节点。
- .....

  欢迎去GitHub补充

加分项

如果想要在面试中获得更好的表现，还应了解下面这些加分项。

- 是要结合实际应用场景来介绍缓存的使用。例如调用后端服务接口获取信息时，可以使用本地+远程的多级缓存；对于动态排行榜类的场景可以考虑通过 **Redis** 的 **Sorted set** 来实现等等。
- 最好你有过分布式缓存设计和使用经验，例如项目中在什么场景使用过 **Redis**，使用了什么数据结

构，解决哪类的问题；使用 MC 时根据预估值大小调整 **McSlab** 分配参数等等。

- 最好可以了解缓存使用中可能产生的问题。比如 **Redis** 是单线程处理请求，应尽量避免耗时较高的单个请求任务，防止相互影响；**Redis** 服务应避免和其他 CPU 密集型的进程部署在同一机器；或者禁用 Swap 内存交换，防止 **Redis** 的缓存数据交换到硬盘上，影响性能。再比如前面提到的 MC 钙化问题等等。
- 要了解 **Redis** 的典型应用场景，例如，使用 **Redis** 来实现分布式锁；使用 **Bitmap** 来实现 **BloomFilter**，使用 **HyperLogLog** 来进行 UV 统计等等。
- 知道 Redis4.0、5.0 中的新特性，例如支持多播的可持久化消息队列 Stream；通过 Module 系统来进行定制功能扩展等等。
- .....

还是那句话欢迎去GitHub补充。

总结

这次是对我**Redis**系列的总结，这应该是**Redis**相关的最后一篇文章了，其实四篇看下来的小伙伴很多都从**一知半解**到了**一脸懵逼**，哈哈开个玩笑。

我觉得我的方式应该还好，大部分小伙伴还是比较能理解的，这篇之后我就不会写**Redis**相关的文章了(秒杀看大家想看的热度吧)，有啥问题可以微信找我，**下个系列写啥**?

大家不用急，下个系列前我会发个有意思的文章，是我在公司代码创意大赛拿奖的文章，我觉得还是**有点东西**，我忍不住分享一下，顺便就在那期发起**投票**吧哈哈。

我看到很多小伙伴都有评论说想看别的，大概搜集了一下，还没留言的这期赶紧哟：

掘金

愚辛：想看计算机基础，网络和操作系统那些（FPX牛脾）

cherish君：讲讲dubbo经常遇到的面试题目，太多人喜欢问dubbo😃

Java架构养成记：真的很香啊，下一期讲Dubbbo（重点SPI）然后讲MQ好吗

CSDN

小殿下：看完了所有的redis篇 希望可以出ssm

博客园

程然：Dubbo Dubbo

开源中国

linshi2019：这期明显是赶工之作啊

敖丙：这条我回一下，鞭策我，我很喜欢，不过说实话还是希望大家**理解**下，我双十一熬夜三天了，现在给你们写的时候也是值班回家2点左右了，我一天吃饭工作时间肯定是固定的，想写点东西就只有挤出睡觉时间了，这种产出肯定没周末全情投入写的来的质量高。

其实第一期看过来的小伙伴应该也知道，我在**排版**，还有很多**文案**，**配图**其实我一直都有在改进的，光是名词高亮我都要弄很久，因为怕大家看单一的黑白色调枯燥。

我是真的用心在搞，还是希望大家支持下理解下。

知乎、简书、思否、慕课手记没人看不知道为啥，懂行的老铁可以跟我说一下。

我只想说你们想看的肯定都在我开头和**GITHub**那个图里吧，问题不大，后面都会写的。

鸣谢

最后感谢下，新浪微博的技术专家张雷。

他于2013年加入**新浪微博**，作为核心技术人员参与了微博服务化、混合云等多个重点项目，是微博开源的**RPC**框架**Motan**的技术负责人，同时也负责微博的**Service Mesh**方案的研发与推广，专注于高可用架构及服务中间件开发方向。

他负责的**Motan**框架每天承载着万亿级别的请求调用，是微博平台服务化的基石，每次的突发热点事件、每次的春晚流量高峰，都离不开**Motan**框架的支撑与保障。此外，他也多次应邀在**ArchSummit**、**WOT**、**GIAC**技术峰会做技术分享。

感谢他对文章部分文案提供的支持和思路。

点关注，不迷路

好了各位，以上就是这篇文章的全部内容了，能看到这里的人呀，都是**人才**。

我后面会每周都更新几篇一线互联网大厂面试和常用技术栈相关的文章，非常感谢**人才**们能看到这里，如果这个文章写得还不错，觉得「敖丙」我**有点东西**的话 **求点赞**👍 **求关注**❤️ **求分享**👥 对暖男我来说真的 **非常有用**！！！

创作不易，各位的支持和认可，就是我创作的最大动力，我们下篇文章见！

敖丙 | 文 【原创】

如果本篇博客有任何错误，请批评指教，不胜感激！

> 文章每周持续更新，可以微信搜索「 **三太子敖丙** 」第一时间阅读和催更（比博客早一到两篇哟），本文 **GitHub** https://github.com/JavaFamily 已经收录，有一线大厂面试点思维导图，也整理了很多我的文档，欢迎Star和完善，大家面试可以参照考点复习，希望我们一起有点东西。