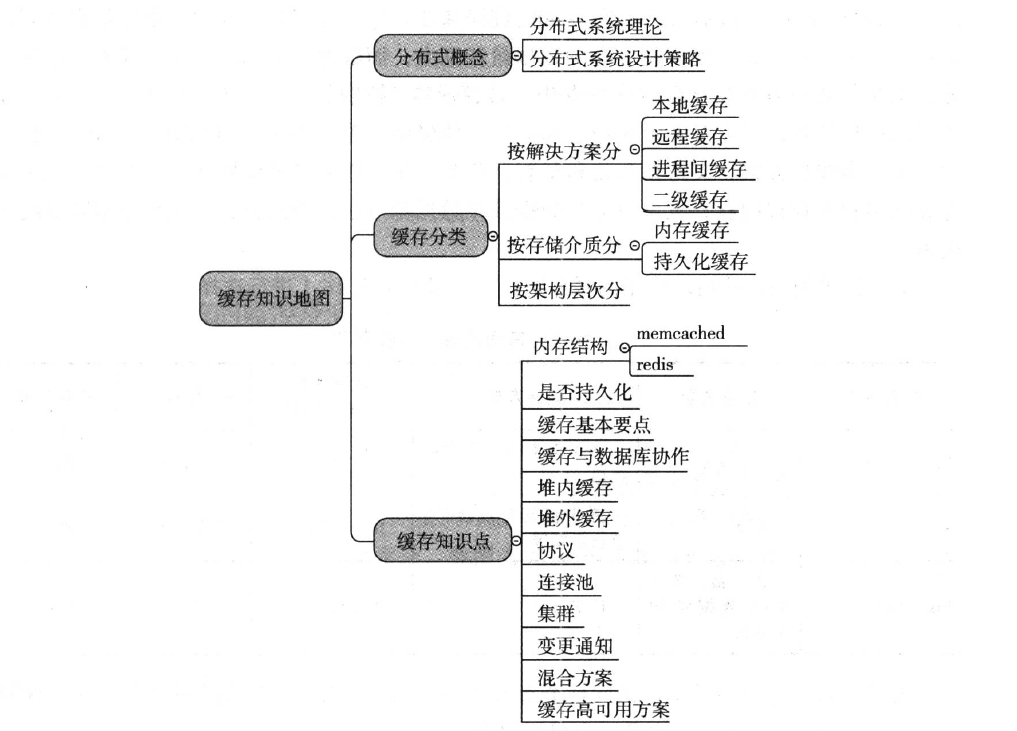
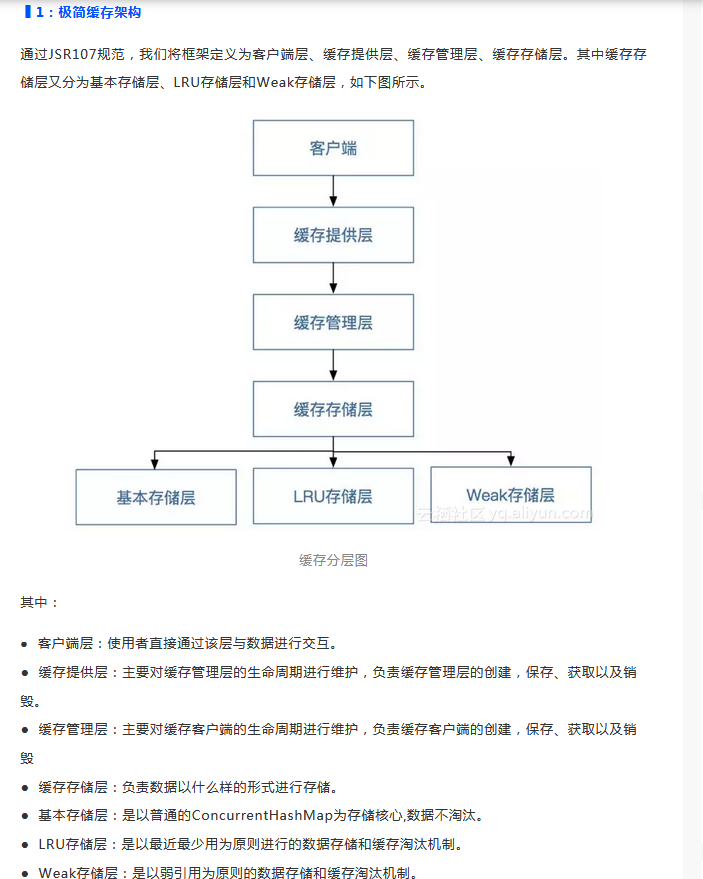
### 1.缓存





#### 基础

**缓存**：是指可以进行高速[数据](https://baike.baidu.com/item/%E6%95%B0%E6%8D%AE)交换的[存储器](https://baike.baidu.com/item/%E5%AD%98%E5%82%A8%E5%99%A8)，它先于[内存](https://baike.baidu.com/item/%E5%86%85%E5%AD%98)与[CPU](https://baike.baidu.com/item/CPU)交换数据，因此[速率](https://baike.baidu.com/item/%E9%80%9F%E7%8E%87)很快, 缓存就是数据交换的缓冲区。

**缓存的工作原理：**是当CPU要读取一个[数据](https://baike.baidu.com/item/%E6%95%B0%E6%8D%AE)时，首先从CPU缓存中查找，找到就立即读取并送给CPU处理；没有找到，就从速率相对较慢的[内存](https://baike.baidu.com/item/%E5%86%85%E5%AD%98)中读取并送给CPU处理，同时把这个[数据](https://baike.baidu.com/item/%E6%95%B0%E6%8D%AE)所在的[数据块](https://baike.baidu.com/item/%E6%95%B0%E6%8D%AE%E5%9D%97)调入缓存中，可以使得以后对整块数据的读取都从缓存中进行，不必再调用内存。正是这样的读取机制使CPU读取缓存的命中率非常高

CPU<—Cache<—Memory

**三级缓存**（包括L1一级缓存、L2二级缓存、L3三级缓存）都是集成在CPU内的缓存，它们的作用都是作为CPU与主内存之间的高速数据缓冲区，L1最靠近CPU核心；L2其次；L3再次。运行速度方面：L1最快、L2次快、L3最慢；容量大小方面：L1最小、L2较大、L3最大。

一级缓存: 位于CPU内核的旁边，是与CPU结合最为紧密的CPU缓存,容量最小，速度最快

二级缓存：L2二级缓存实际上就是L1一级缓存跟主内存之间的缓冲器

三级缓存：L3即为L2与主内存之间的缓冲器

**缓存系统**

1.小型网站或者企业级应用：基于内存的k-v存储的缓存服务器，并且可以实现主备切换，保证可用性

2.大型：需要考虑负载，扩容等问题

<https://blog.csdn.net/zjttlance/article/details/80234341>

**优点**

加速读写，加快访问速度，提高性能

降低后端负载(降低mysql负载)

**缺点**

数据不一致：缓存层和数据层有时间窗口不一致，和更新策略有关

代码维护成本：多了一层缓存逻辑

运维成本：例如redis cluster

**使用场景**

降低后端负载：对于高消耗的SQL：join结果集、分组统计结果；对这些结果进行缓存。

加速请求响应

大量写合并为批量写：如计数器先redis累加再批量写入DB

缓存是分布式系统中的重要组件，主要解决高并发，大数据场景下，热点数据访问的性能问题。提供高性能的数据快速访问。

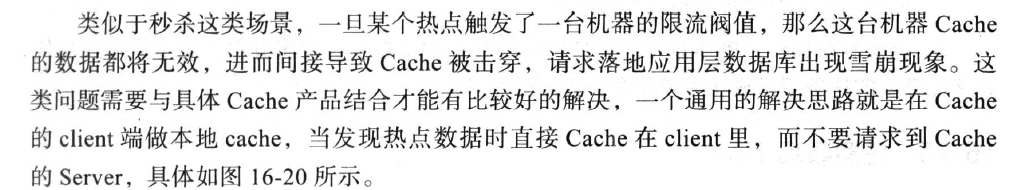
**Cache Aside Pattern**

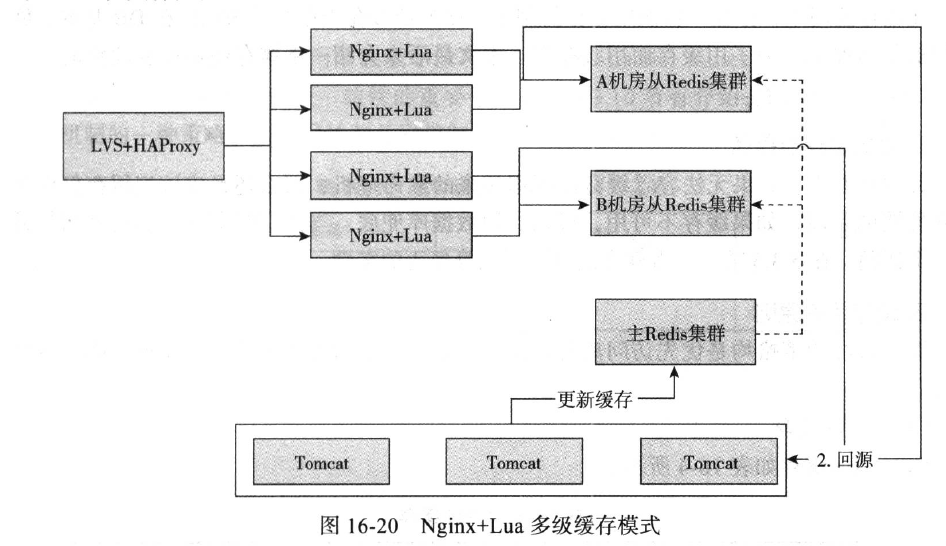
这是最常用最常用的pattern了。其具体逻辑如下：

失效：应用程序先从cache取数据，没有得到，则从数据库中取数据，成功后，放到缓存中。

命中：应用程序从cache中取数据，取到后返回。

更新：先把数据存到数据库中，成功后，再让缓存失效。





#### 更新顺序问题

**更新数据时，是先删除缓存再更新DB，还是先更新DB再删除缓存？**

当数据出现变化的时候，DB和redis的一致性就显得非常重要，目前有两种（基本思路）策略：

1.先删除缓存再更新DB

结论：产生脏数据的概率较大（若出现脏数据，则意味着再不更新的情况下，查询得到的数据均为旧的数据）

比如：两个并发操作，一个是更新操作，另一个是查询操作，更新操作删除缓存后，查询操作没有命中缓存，先把老数据读出来后放到缓存中，然后更新操作更新了数据库。于是，在缓存中的数据还是老的数据，导致缓存中的数据是脏的，而且还一直这样脏下去了。

**2.先更新DB再删除缓存（使用场景多）**

结论：产生脏数据的概率较小，但是会出现一致性的问题；若更新操作的时候，同时进行查询操作，若hit，则查询得到的数据是旧的数据。但是不会影响后面的查询。（代价较小）

（ 一个是查询操作，一个是更新操作的并发，首先，没有了删除cache数据的操作了，而是先更新了数据库中的数据，此时，缓存依然有效，所以，并发的查询操作拿的是没有更新的数据，但是，更新操作马上让缓存的失效了，后续的查询操作再把数据从数据库中拉出来。而不会像文章开头的那个逻辑产生的问题，后续的查询操作一直都在取老的数据。）

该设计模式产生脏数据的可能情况：

一个是读操作，但是没有命中缓存，然后就到数据库中取数据，此时来了一个写操作，写完数据库后，让缓存失效，然后，之前的那个读操作再把老的数据放进去，所以，会造成脏数据。

该情况出现的概率可能非常低，因为这个条件需要发生在读缓存时缓存失效，而且并发着有一个写操作。而实际上数据库的写操作会比读操作慢得多，而且还要锁表，而读操作必需在写操作前进入数据库操作，而又要晚于写操作更新缓存，所有的这些条件都具备的概率基本并不大。

强一致性 实现：通过2PC或是Paxos协议保证一致性；

生产过程中要么保持强一致性，要么拼命的降低并发时脏数据的概率，而Facebook使用了这个降低概率的策略，因为2PC太慢，而Paxos太复杂。当然，最好还是为缓存设置上过期时间。

详细分析见：<https://coolshell.cn/articles/17416.html>

#### 缓存的更新策略(算法)

为了保证CPU访问时有较高的命中率，Cache中的内容应该按一定的算法替换

###### LRU/LFU/FIFO算法



* FIFO(first in first out)

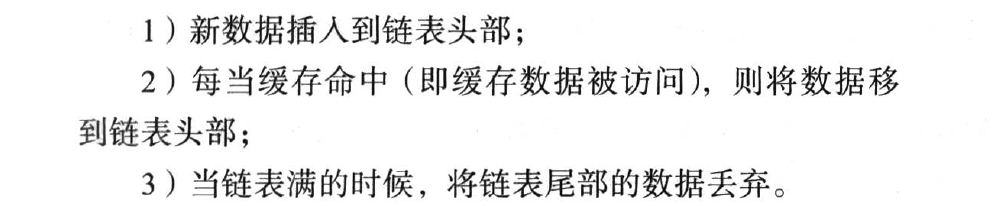
先进先出策略，最先进入缓存的数据在缓存空间不够的情况下（超出最大元素限制）会被优先被清除掉，以腾出新的空间接受新的数据。策略算法主要比较缓存元素的创建时间。在数据实效性要求场景下可选择该类策略，优先保障最新数据可用。

* LFU(less frequently used)

最少使用策略，无论是否过期，根据元素的被使用次数判断，清除使用次数较少的元素释放空间。策略算法主要比较元素的hitCount（命中次数）。在保证高频数据有效性场景下，可选择这类策略。

* LRU(least recently used):最常用

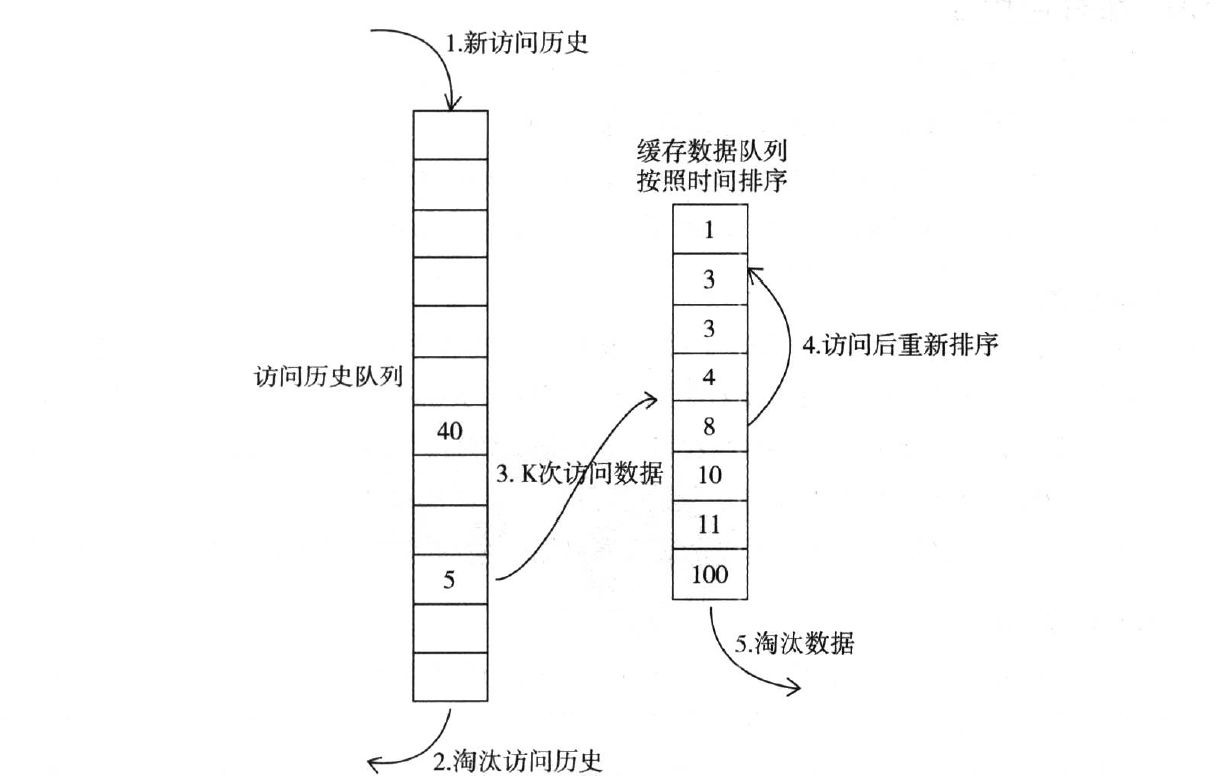
最近最少使用策略，无论是否过期，根据元素最后一次被使用的时间戳，清除最远使用时间戳的元素释放空间。策略算法主要比较元素最近一次被get使用时间。在热点数据场景下较适用，优先保证热点数据的有效性。



问题：偶发性的，周期性的批量操作会引起LRU效率下降，容易引起缓存污染（在缓存机制中，会存在把不常用的数据读取到缓存中的现象，这种现象称为缓存污染）



LRU-K:为了解决缓存污染，一般来说维护两个队列，用来记录缓存数据被访问的历史，另外一个用来记录按照时间排序的数据，只有访问数据数据达到K次的时候才放入缓存队列，当需要淘汰数据时，淘汰第K次访问时间距离当前时间最大的数据



1. 数据第一次被访问，加入到访问历史列表；

2. 如果数据在访问历史列表里后没有达到K次访问，则按照一定规则（FIFO，LRU）淘汰；

3. 当访问历史队列中的数据访问次数达到K次后，将数据索引从历史队列删除，将数据移到缓存队列中，并缓存此数据，缓存队列重新按照时间排序；

4. 缓存数据队列中被再次访问后，重新排序；

5. 需要淘汰数据时，淘汰缓存队列中排在末尾的数据，即：淘汰“倒数第K次访问离现在最久”的数据。

LRU-K具有LRU的优点，同时能够避免LRU的缺点，实际应用中LRU-2是综合各种因素后最优的选择，LRU-3或者更大的K值命中率会高，但适应性差，需要大量的数据访问才能将历史访问记录清除掉

【命中率】

LRU-K降低了“缓存污染”带来的问题，命中率比LRU要高。

【复杂度】

LRU-K队列是一个优先级队列，算法复杂度和代价比较高。

【代价】

由于LRU-K还需要记录那些被访问过、但还没有放入缓存的对象，因此内存消耗会比LRU要多；当数据量很大的时候，内存消耗会比较可观。

LRU-K需要基于时间进行排序（可以需要淘汰时再排序，也可以即时排序），CPU消耗比LRU要高。

超时剔除：例如expire

主动更新：开发控制生命周期（最终一致性，时间间隔比较短）

低一致性：最大内存和淘汰策略

高一致性：超时剔除和主动更新结合，最大内存和淘汰策略兜底。

#### 缓存穿透，缓存击穿，缓存雪崩

<https://www.jianshu.com/p/b57d0773ee96>

* 缓存穿透：

访问一个不存在的key，缓存不起作用，请求会穿透到DB，流量大时DB会挂掉。

方案：

1.采用布隆过滤器，使用一个足够大的bitmap，用于存储可能访问的key，不存在的key直接被过滤；

2.访问key未在DB查询到值，也将空值写进缓存，但可以设置较短过期时间。

* 缓存雪崩（大量key） / 缓存失效

大量的key设置了相同的过期时间，导致在缓存在同一时刻全部失效，造成瞬时DB请求量大、压力骤增，引起雪崩。

方案：可以给缓存设置过期时间时加上一个随机值时间，使得每个key的过期时间分布开来，不会集中在同一时刻失效。

* 缓存击穿（单个key）

一个存在的key，在缓存过期的一刻，同时有大量的请求，这些请求都会击穿到DB，造成瞬时DB请求量大、压力骤增。

方案：在访问key之前，采用SETNX（set if not exists）来设置另一个短期key来锁住当前key的访问，访问结束再删除该短期key。

#### 缓存架构

高可用分布式系统架构设计

面临难题：如何解决分布式系统中的服务高可用问题？避免多层服务依赖因为少量故障导致系统崩溃？

解决方案：资源隔离+限流+降级+熔断+超时控制

后端缓存的23个关键关注点

<https://yq.aliyun.com/articles/674436?utm_content=m_1000028077>

#### 缓存的技术方案分类

1是做实时性比较高的那块数据，比如说库存，销量之类的这种数据，我们采取的实时的缓存+数据库双写的技术方案，双写一致性保障的方案。

2是做实时性要求不高的数据，比如说商品的基本信息，等等，我们采取的是三级缓存架构的技术方案，就是说由一个专门的数据生产的服务，去获取整个商品详情页需要的各种数据，经过处理后，将数据放入各级缓存中。

3高并发以及高可用的复杂系统中的缓存架构都有哪些东西

1）在大型的缓存架构中，redis是最最基础的一层。高并发，缓存架构中除了redis，还有其他的组成部分，但是redis至关重要。

如果你的数据量不大（10G以内），单master就可以。redis持久化+备份方案+容灾方案+replication（主从+读写分离）+sentinal（哨兵集群，3个节点，高可用性）

如果你的数据量很大（1T+），采用redis cluster。多master分布式存储数据，水平扩容,自动进行master -> slave的主备切换。

2）最经典的缓存+数据库读写的模式，cache aside pattern。读的时候，先读缓存，缓存没有的话，那么就读数据库。更新缓存分以下两种方式：

数据发生变化时，先更新缓存，然后再更新数据库。这种适用于缓存的值相对简单，和数据库的值一一对应，这样更新比较快。

数据发生变化时，先删除缓存，然后再更新数据库，读数据的时候再设置缓存。这种适用于缓存的值比较复杂的场景。比如可能更新了某个表的一个字段，然后其对应的缓存，是需要查询另外两个表的数据，并进行运算，才能计算出缓存最新的值的。这样更新缓存的代价是很高的。如果你频繁修改一个缓存涉及的多个表，那么这个缓存会被频繁的更新，频繁的更新缓存代价很高。而且这个缓存的值如果不是被频繁访问，就得不偿失了。

大部分情况下，建议适用删除更新的方式。其实删除缓存，而不是更新缓存，就是一个lazy计算的思想，不要每次都重新做复杂的计算，不管它会不会用到，而是让它到需要被使用的时候再重新计算。

3）数据库与缓存双写不一致问题的解决方案

问题：并发请求的时候，数据发生了变更，先删除了缓存，然后要去修改数据库，此时还没修改。另一个请求过来，去读缓存，发现缓存空了，去查询数据库，查到了修改前的旧数据，放到了缓存中。

方案：数据库与缓存更新与读取操作进行异步串行化。（引入队列）

更新数据的时候，将相应操作发送到一个jvm内部的队列中。读取数据的时候，如果发现数据不在缓存中，那么将重新读取数据的操作也发送到同一个jvm内部的队列中。队列消费者串行拿到对应的操作，然后一条一条的执行。这样的话，一个数据变更的操作，先执行删除缓存，然后再去更新数据库，但是还没完成更新。此时如果一个读请求过来，读到了空的缓存，那么可以先将缓存更新的请求发送到队列中，此时会在队列中积压，然后同步等待缓存更新完成。

这里有两个可以优化的点：

一个队列中，其实多个读缓存，更新缓存的请求串在一起是没意义的，而且如果读同一缓存的大量请求到来时，会依次进入队列等待，这样会导致队列最后一个的请求响应时间超时。因此可以做过滤，如果发现队列中已经有一个读缓存，更新缓存的请求了，那么就不用再放个新请求操作进去了，直接等待前面的更新操作请求完成即可。如果请求还在等待时间范围内，不断轮询发现可以取到值了，那么就直接返回; 如果请求等待的时间超过一定时长，那么这一次直接从数据库中读取当前的旧值。

如果请求量特别大的时候，可以用多个队列，每个队列对应一个线程。每个请求来时可以根据请求的标识id进行hash路由进入到不同的队列。

4）大型缓存全量更新问题的解决方案

问题：缓存数据很大时，可能导致redis的吞吐量就会急剧下降，网络耗费的资源大。如果不维度化，就导致多个维度的数据混合在一个缓存value中。而且不同维度的数据，可能更新的频率都大不一样。拿商品详情页来说，如果现在只是将1000个商品的分类批量调整了一下，但是如果商品分类的数据和商品本身的数据混杂在一起。那么可能导致需要将包括商品在内的大缓存value取出来，进行更新，再写回去，就会很坑爹，耗费大量的资源，redis压力也很大

方案：缓存维度化。举个例子：商品详情页分三个维度：商品维度，商品分类维度，商品店铺维度。将每个维度的数据都存一份，比如说商品维度的数据存一份，商品分类的数据存一份，商品店铺的数据存一份。那么在不同的维度数据更新的时候，只要去更新对应的维度就可以了。大大减轻了redis的压力。

5）通过多级缓存，达到高并发极致，同时给缓存架构最后的安全保护层。

6）分布式并发缓存重建的冲突问题的解决方案

问题：假如数据在所有的缓存中都不存在了（LRU算法弄掉了），就需要重新查询数据写入缓存。对于分布式的重建缓存，在不同的机器上，不同的服务实例中，去做上面的事情，就会出现多个机器分布式重建去读取相同的数据，然后写入缓存中。

方案：分布式锁：如果你有多个机器在访问同一个共享资源，那么这个时候，如果你需要加个锁，让多个分布式的机器在访问共享资源的时候串行起来。

7）缓存冷启动的问题的解决方案

问题：新系统第一次上线，此时在缓存里可能是没有数据的。或者redis缓存全盘崩溃了，数据也丢了。导致所有请求打到了mysql。导致mysql直接挂掉。

方案：缓存预热。

提前给redis中灌入部分数据，再提供服务

肯定不可能将所有数据都写入redis，因为数据量太大了，第一耗费的时间太长了，第二根本redis容纳不下所有的数据，需要根据当天的具体访问情况，实时统计出访问频率较高的热数据，然后将访问频率较高的热数据写入redis中，肯定是热数据也比较多，我们也得多个服务并行读取数据去写，并行的分布式的缓存预热。

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Single is simple, double is trouble, triple is terrible, multiple is miracle

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

### 2.负载均衡

##### 负载均衡

当一台服务器的性能达到极限时，我们可以使用服务器集群来提高网站的整体性能。那么，在服务器集群中，需要有一台服务器充当调度者的角色，用户的所有请求都会首先由它接收，调度者再根据每台服务器的负载情况将请求分配给某一台后端服务器去处理。

那么在这个过程中，调度者如何合理分配任务，保证所有后端服务器都将性能充分发挥，从而保持服务器集群的整体性能最优，这就是负载均衡问题。组件：HAProxy，Nginx，LVS

##### 算法/调度策略

**1. 随机算法**

Random随机，按权重设置随机概率。在一个截面上碰撞的概率高，但调用量越大分布越均匀，而且按概率使用权重后也比较均匀，有利于动态调整提供者权重。

**2. 轮询及加权轮询**

轮询(Round Robbin)当服务器群中各服务器的处理能力相同时，且每笔业务处理量差异不大时，最适合使用这种算法。 轮循，按公约后的权重设置轮循比率。

加权轮询(Weighted Round Robbin)为轮询中的每台服务器附加一定权重的算法。比如服务器1权重1，服务器2权重2，服务器3权重3，则顺序为1-2-2-3-3-3-1-2-2-3-3-3- ......

**3. 最小连接及加权最小连接**

最少连接(Least Connections)在多个服务器中，与处理连接数(会话数)最少的服务器进行通信的算法。即使在每台服务器处理能力各不相同，每笔业务处理量也不相同的情况下，也能够在一定程度上降低服务器的负载。

加权最少连接(Weighted Least Connection)为最少连接算法中的每台服务器附加权重的算法，该算法事先为每台服务器分配处理连接的数量，并将客户端请求转至连接数最少的服务器上。

**4. 哈希算法**

一致性哈希一致性Hash，相同参数的请求总是发到同一提供者。当某一台提供者挂时，原本发往该提供者的请求，基于虚拟节点，平摊到其它提供者，不会引起剧烈变动。

**5. IP地址散列**

通过管理发送方IP和目的地IP地址的散列，将来自同一发送方的分组(或发送至同一目的地的分组)统一转发到相同服务器的算法。当客户端有一系列业务需要处理而必须和一个服务器反复通信时，该算法能够以流(会话)为单位，保证来自相同客户端的通信能够一直在同一服务器中进行处理。

### 3.分布式

##### 分布式系统设计策略

1.心跳检测：不是万能的，服务器繁忙或者网络故障都可能导致该机制失效，没有返回不代表宕机

周期心跳检测机制：每隔n秒向节点集群发心跳监测请求，并设定超时时间，如果超时仍然没有返回，则认为已经死亡，剔除集群

累计失效检测机制

2.重试机制

3.容错机制：容故障，而非错误，是为了保障分布式系统的可用性和健壮性，典型案例是缓存失效雪崩的问题：设置默认值

4.保障高可用性：主备，互备（Master-Master:Multi-Master 代表git，最终一致性），集群（ZK：需要解决主控节点本身的高可用问题）

5.负载均衡：调度分发，把网络请求和计算分配到合适的服务器上面，达到较好的可用性

LVS、HAProxy，Nginx（策略：轮询，最小连接，IPhash，基于权重的负载均衡：把请求更多分发到配置很好的服务器）

##### 分布式系统设计通用性问题

1.全局ID生成

uuid ，时钟序列，全局唯一的机器识别号 简单易用 占空间大 id生成表以及结合缓存

2.hash取模分配：最常见的数据分配方式，好处是只需要计算就可以映射出节点和数据的关系，而不需要存储映射，缺点在于id分布不均匀，数据倾斜

3.路由表

4.一致性hash：

优点在于增加或者删除只会影响临节点，引入虚节点尽可能均匀分布节点和数据（解决平衡性问题：hash的结果能够尽可能分布到所有缓存中去）

解决单调性（hash的结果应该确保已经分配的内容能够映射到原来的缓存中，避免在节点增减而导致不能命中）和分散性问题

5.数据拆分

幂等性:简单地说，就是对接口的多次调用所产生的结果和调用一次是一致的。

解决方式：每一个不同的业务操作赋予其唯一性。这个唯一性是通过对不同操作所对应的唯一的内容特性生成一个唯一的全局ID来实现的。基本原则为：相同的操作生成相同的全局ID；不同的操作生成不同的全局ID

操作互斥性：常见的做法便是通过分布式锁来处理对共享资源的抢占，详见[3.分布式锁](#_3.分布式锁)

#### 一致性

CAP理论提出了一致性，可用性和分区容忍性的取舍问题

Paxos，Raft，2PC，3PC给出了一致性的解决方案

2PC是用来保证跨多个节点操作的原子性

Paxos协议用来确保多个节点对某个投票（比如选举）达成一致性

Lease机制主要解决网络拥塞或者瞬断的情况下，出现双主情况的解法

分布式事物基本理论:基本遵循CPA理论或者Base理论，采用柔性事物特征，软状态或者最终一致性特点保证分布式事物一致性问题。

分布式事物一致性常见解决方案:

1.2pc两段提交协议（同步阻塞，单点问题）

2.3pc三段提交协议(弥补两端提交协议缺点)

3.消息中间件最终一致性

**4. 补偿事务（TCC）**

TCC 其实就是采用的补偿机制，其核心思想是：针对每个操作，都要注册一个与其对应的确认和补偿（撤销）操作。它分为三个阶段：

* Try 阶段主要是对业务系统做检测及资源预留
* Confirm 阶段主要是对业务系统做确认提交，Try阶段执行成功并开始执行 Confirm阶段时，默认 Confirm阶段是不会出错的。即：只要Try成功，Confirm一定成功。
* Cancel 阶段主要是在业务执行错误，需要回滚的状态下执行的业务取消，预留资源释放。

举个例子，假入 Bob 要向 Smith 转账，思路大概是：  
我们有一个本地方法，里面依次调用  
1、首先在 Try 阶段，要先调用远程接口把 Smith 和 Bob 的钱给冻结起来。  
2、在 Confirm 阶段，执行远程调用的转账的操作，转账成功进行解冻。  
3、如果第2步执行成功，那么转账成功，如果第二步执行失败，则调用远程冻结接口对应的解冻方法 (Cancel)。

**优点：** 跟2PC比起来，实现以及流程相对简单了一些，但数据的一致性比2PC也要差一些

**缺点：** 缺点还是比较明显的，在2,3步中都有可能失败。TCC属于应用层的一种补偿方式，所以需要程序员在实现的时候多写很多补偿的代码，在一些场景中，一些业务流程可能用TCC不太好定义及处理。

一致性级别：

**1、强一致性**

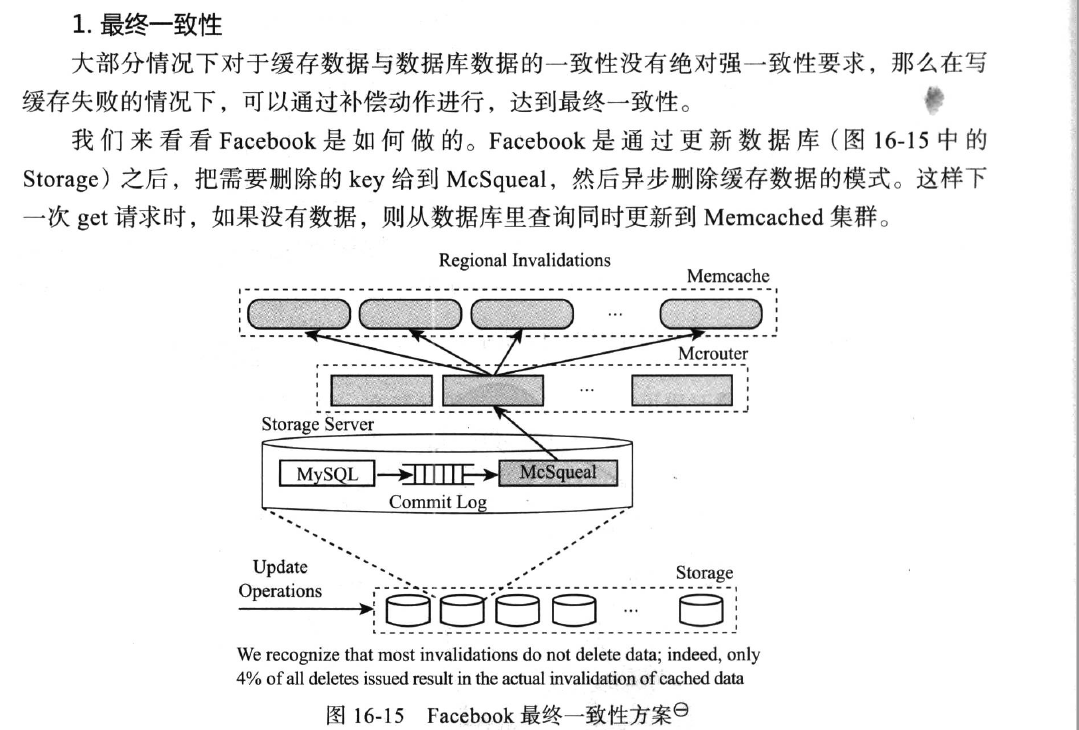
这种一致性级别是最符合用户直觉的，它要求系统写入什么，读出来的也会是什么，用户体验好，但实现起来往往对系统的性能影响大

**2、弱一致性**

这种一致性级别约束了系统在写入成功后，不承诺立即可以读到写入的值，也不久承诺多久之后数据能够达到一致，但会尽可能地保证到某个时间级别（比如秒级别）后，数据能够达到一致状态

**3、最终一致性（绝大多数场景下采用）**

最终一致性是弱一致性的一个特例，系统会保证在一定时间内，能够达到一个数据一致的状态。这里之所以将最终一致性单独提出来，是因为它是弱一致性中非常推崇的一种一致性模型，也是业界在大型分布式系统的数据一致性上比较推崇的模型



##### Paxos和Raft

Paxos：分布式系统中，多个节点就某一提议达成一致的协议

Raft：领导者选举，日志复制，安全性等模块

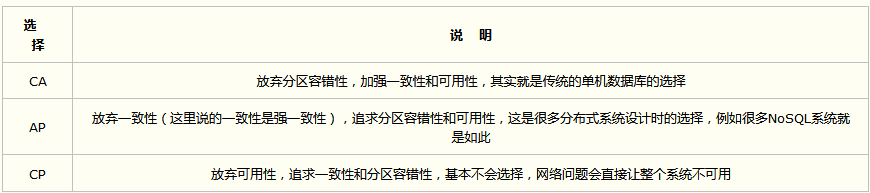
##### BASE和CAP

 数据一致性(consistency)：所有节点在同一时刻能够看到同样的数据，也即“强一致性”；

 服务可用性(availability)：所有读写请求在一定时间内得到响应，可终止、不会一直等待；

 分区性(partition-tolerance)：因为网络故障导致的系统分区不影响系统正常运行

定理：任何分布式系统只可同时满足二点，没法三者兼顾



对于一个分布式系统而言，分区容错性是一个最基本的要求。因为 既然是一个分布式系统，那么分布式系统中的组件必然需要被部署到不同的节点，否则也就无所谓分布式系统了，因此必然出现子网络。而对于分布式系统而言，网 络问题又是一个必定会出现的异常情况，因此分区容错性也就成为了一个分布式系统必然需要面对和解决的问题。因此系统架构师往往需要把精力花在如何根据业务 特点在C（一致性）和A（可用性）之间寻求平衡。

**BASE理论**

BASE是Basically Available（基本可用）、Soft state（软状态）和Eventually consistent（最终一致性）三个短语的缩写。BASE理论是对CAP中一致性和可用性权衡的结果，其来源于对大规模互联网系统分布式实践的总结， 是基于CAP定理逐步演化而来的。BASE理论的核心思想是：即使无法做到强一致性，但每个应用都可以根据自身业务特点，采用适当的方式来使系统达到最终一致性。接下来看一下BASE中的三要素：

**1、基本可用**

基本可用是指分布式系统在出现不可预知故障的时候，允许损失部分可用性----注意，这绝不等价于系统不可用。比如：

（1）响应时间上的损失。正常情况下，一个在线搜索引擎需要在0.5秒之内返回给用户相应的查询结果，但由于出现故障，查询结果的响应时间增加了1~2秒

（2）系统功能上的损失：正常情况下，在一个电子商务网站上进行购物的时候，消费者几乎能够顺利完成每一笔订单，但是在一些节日大促购物高峰的时候，由于消费者的购物行为激增，为了保护购物系统的稳定性，部分消费者可能会被引导到一个降级页面

**2、软状态**

软状态指允许系统中的数据存在中间状态，并认为该中间状态的存在不会影响系统的整体可用性，即允许系统在不同节点的数据副本之间进行数据同步的过程存在延时

**3、最终一致性**

最终一致性强调的是所有的数据副本，在经过一段时间的同步之后，最终都能够达到一个一致的状态。因此，最终一致性的本质是需要系统保证最终数据能够达到一致，而不需要实时保证系统数据的强一致性。

总的来说，BASE理论面向的是大型高可用可扩展的分布式系统，和传统的事物ACID特性是相反的，**它完全不同于ACID的强一致性模型，而是通过牺牲强一致性来获得可用性，并允许数据在一段时间内是不一致的，但最终达到一致状态**。但同时，在实际的分布式场景中，不同业务单元和组件对数据一致性的要求是不同的，因此在具体的分布式系统架构设计过程中，ACID特性和BASE理论往往又会结合在一起。

柔性事务满足BASE理论（基本可用，最终一致）  
刚性事务满足ACID理论

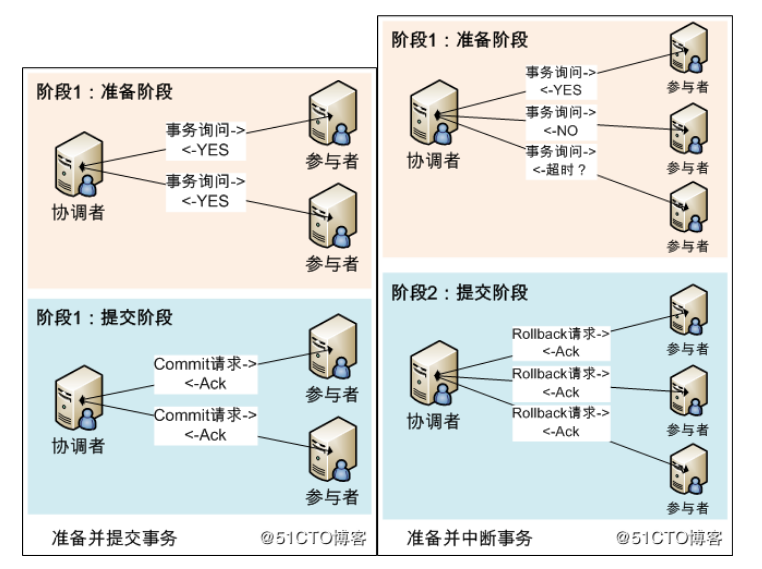
柔性事务分为两阶段型，补偿型，异步确保型。最大努力通知型几种。

##### 2PC/3PC

###### 2PC

2PC，将事务的提交过程分为：准备阶段和提交阶段。事务的发起者称协调者，事务的执行者称参与者。

　附如下示意图：



2PC的缺陷

1、同步阻塞：最大的问题即同步阻塞，即：所有参与事务的逻辑均处于阻塞状态。

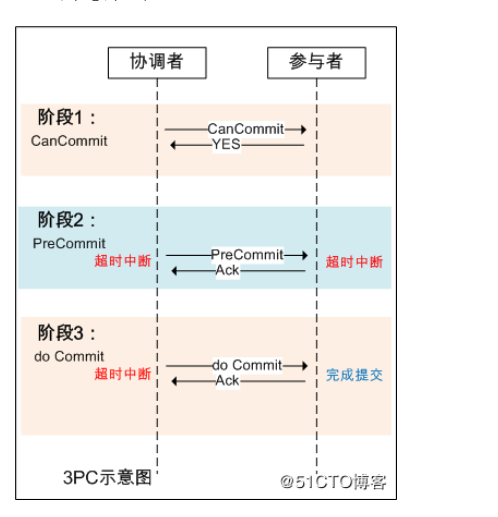
2、单点：协调者存在单点问题，如果协调者出现故障，参与者将一直处于锁定状态。

3、脑裂：在阶段2中，如果只有部分参与者接收并执行了Commit请求，会导致节点数据不一致。

由于2PC存在如上同步阻塞、单点、脑裂问题，因此又出现了2PC的改进方案，即3PC。

###### 3PC

3PC，三阶段提交协议，是2PC的改进版本，即将事务的提交过程分为CanCommit、PreCommit、DoCommit三个阶段来进行处理。只要PreCommit成功，则一定要保证DoCommit提交成功，即使协调器在下一阶段不可用或者调用超时，这是协议的基本思想，在工业环境中，一般通过重试补偿机制来确保DoCommit提交成功。

附示意图如下：

3PC的优点和缺陷

优点：降低了阻塞范围，引入了超时机制，在等待超时后协调者或参与者会中断事务。

缺陷：脑裂问题依然存在，即在参与者收到PreCommit请求后等待最终指令，如果此时协调者无法与参与者正常通信，会导致参与者继续提交事务，造成数据不一致。

详见:

<https://blog.csdn.net/lizhen1114/article/details/80110317>

##### 脑裂问题

脑裂是指在主备切换时，由于切换不彻底或其他原因，导致客户端和Slave误以为出现两个active master，最终使得整个集群处于混乱状态

避免：在slaver切换的时候不在检查到老的master出现问题后马上切换，而是在休眠一段足够的时间，确保老的master已经获知变更并且做了相关的shutdown清理工作了然后再注册成为master就能避免这类问题了，这个休眠时间一般定义为与zookeeper定义的超时时间就够了，但是这段时间内系统可能是不可用的，但是相对于数据不一致的后果我想还是值得的。

当然最彻底的解决这类问题的方案是将master HA集群做成peer2peer的，屏蔽掉外部Zookeeper的依赖。每个节点都是对等的没有主次，这样就不会存在脑裂的问题，但是这种ha解决方案需要使用两阶段，paxos这类数据一致性保证协议来实现，不可避免的会降低系统数据变更的系统，如果系统中主要是对master的读取操作很少更新就很适合了。

或者引入仲裁机制，即第三方，第三方也需要实现HA

##### 租约(lease)机制

Lease 是由授权者授予的在一段时间内的承诺。

授权者一旦发出 lease，则无论接受方是否收到，也无论后续接收方处于何种状态，只要 lease 不过期，授权者一定遵守承诺，按承诺的时间、内容执行。

接收方在有效期内可以使用颁发者的承诺，只要 lease 过期，接收方放弃授权，不再继续执行，要重新申请Lease。

用于处理缓存一致性，选举，脑裂(确定集群中结点的状态)，还可以实现分布式下的读写锁等。

##### 分布式锁

当多个进程不在同一个系统中，用分布式锁控制多个进程对资源的访问，分布式锁实现的关键是在分布式的应用服务器外，搭建一个存储服务器，存储锁信息

同一个共享资源，那么这个时候，如果你需要加个锁，让多个分布式的机器在访问共享资源的时候串行起来。

#### 分布式锁原理及实现方式

目前几乎很多大型网站及应用都是分布式部署的，分布式场景中的数据一致性问题一直是一个比较重要的话题。分布式的CAP理论告诉我们“任何一个分布式系统都无法同时满足一致性（Consistency）、可用性（Availability）和分区容错性（Partition tolerance），最多只能同时满足两项。”所以，很多系统在设计之初就要对这三者做出取舍。在互联网领域的绝大多数的场景中，都需要牺牲强一致性来换取系统的高可用性，系统往往只需要保证“最终一致性”，只要这个最终时间是在用户可以接受的范围内即可。

针对分布式锁的实现，目前比较常用的有以下几种方案：

**基于数据库实现分布式锁：表锁或者排它锁**

**基于缓存（redis，memcached）实现分布式锁**

**基于Zookeeper实现分布式锁 ：临时顺序节点**

zookeeper分布式锁的解决并发冲突的方案

（1）变更缓存重建以及空缓存请求重建，更新redis之前，都需要先获取对应商品id的分布式锁

（2）拿到分布式锁之后，需要根据时间版本去比较一下，如果自己的版本新于redis中的版本，那么就更新，否则就不更新

（3）如果拿不到分布式锁，那么就等待，不断轮询等待，直到自己获取到分布式的锁

详见：<http://www.hollischuang.com/archives/1716>

Redis:在很多应用中，有些场景需要加锁处理，比如：秒杀，全局递增ID，楼层生成等等。大部分的解决方案是基于DB实现的，Redis为单进程单线程模式，采用队列模式将并发访问变成串行访问，且多客户端对Redis的连接并不存在竞争关系。其次Redis提供一些命令SETNX，GETSET，可以方便实现分布式锁机制。

SETNX：（SET if Not eXists）SETNX key val  
 当且仅当key不存在时，set一个key为val的字符串，返回1；若key存在，则什么都不做，返回0。

expire ：expire key timeout  
 为key设置一个超时时间，单位为second，超过这个时间锁会自动释放，避免死锁。

delete：delete key  
 删除key

在使用Redis实现分布式锁的时候，主要就会使用到这三个命令。

实现思想

获取锁的时候，使用setnx加锁，并使用expire命令为锁添加一个超时时间，超过该时间则自动释放锁，锁的value值为一个随机生成的UUID，通过此在释放锁的时候进行 判断。

获取锁的时候还设置一个获取的超时时间，若超过这个时间则放弃获取锁。

释放锁的时候，通过UUID判断是不是该锁，若是该锁，则执行delete进行锁释放。

使用zookeeper的可靠性是要大于使用redis实现的分布式锁的，但是相比而言，redis的性能更好。

详见：<https://www.cnblogs.com/liuyang0/p/6744076.html>

Memcached 可以使用 add 命令，该命令只有KEY不存在时，才进行添加，或者不会处理。Memcached 所有命令都是原子性的，并发下add 同一个KEY ，只会一个会成功。

利用这个原理，可以先定义一个 锁 LockKEY ，add 成功的认为是得到锁。并且设置[过期超时] 时间，保证宕机后，也不会死锁。

在具体操作完后，判断是否此次操作已超时。如果超时则不删除锁，如果不超时则删除锁。

详见：<https://www.cnblogs.com/zrhai/p/4015989.html>

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

这短短的一生,我们最终都会失去。你不妨大胆一些,爱一个人,攀一座山,追一个梦

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

#### 其他

处理高并发的业务逻辑是:

前端:异步请求+资源静态化+cdn

后端:请求队列+轮询分发+负载均衡+共享缓存

数据层:redis缓存+数据分表+写队列

存储:raid阵列+热备

网络:dns轮询+DDOS攻击防护

### 4.面试

#### 数据结构和算法

书籍推荐：

大话数据结构

##### 排序算法

冒泡，插入，归并是稳定的排序算法

多维度排序

时间复杂度

快排，hash排序，堆排序比较重要

##### 红黑树

一种加上了额外平衡条件的二叉搜索树，典型的用途是实现关联数组

**set,map的底层实现，查找时间复杂度logn**

红黑树的特性:

（1）每个节点或者是黑色，或者是红色。

（2）根节点是黑色。

（3）每个叶子节点（NIL）是黑色。 [注意：这里叶子节点，是指为空(NIL或NULL)的叶子节点！]

（4）如果一个节点是红色的，则它的子节点必须是黑色的。

（5）从一个节点到该节点的子孙节点的所有路径上包含相同数目的黑节点。

**RBTree和AVL区别**

红黑树的算法时间复杂度和AVL相同(O(logn))，但统计性能比AVL树更高。

红黑树并不追求“完全平衡”

任何不平衡都会在三次旋转之内解决

红黑树旋转复杂度O(1)，AVL是O(n)

##### [B树](https://www.baidu.com/s?wd=B-%25E6%25A0%2591&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1YknhnkujfsPWDLuHf1ryFh0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EPH6dP1m4P1mLrHbdrHTzn1md)和B+树

1. B+树中只有叶子节点会带有指向记录的指针（ROWID），而B树则所有节点都带有，在内部节点出现的索引项不会再出现在叶子节点中。

2. B+树中所有叶子节点都是通过指针连接在一起，而B树不会。

##### 字典树(前缀树)

单词查找树，[Trie树](http://www.wutianqi.com/?p=1359)，是一种树形结构，是一种哈希树的变种。典型应用是用于统计，排序和保存大量的字符串（但不仅限于字符串），所以经常被搜索引擎系统用于文本词频统计。它的优点是：利用字符串的公共前缀来节约存储空间，最大限度地减少无谓的字符串比较，查询效率比哈希表高。

**Trie的核心思想是空间换时间**。利用字符串的公共前缀来降低查询时间的开销以达到提高效率的目的。

Trie的数据结构定义：

**#define MAX 26  
typedef struct Trie     
{     
    Trie \*next[MAX];    //根据需要变化  
    int v;    
};**   
Trie \*root;

next是表示每层有多少种类的数，如果只是小写字母，则26即可，若改为大小写字母，则是52，若再加上数字，则是62了，这里根据题意来确定。  
v可以表示一个字典树到此有多少相同前缀的数目，这里根据需要应当学会自由变化。

##### TopK问题：最小/大堆

<https://blog.csdn.net/juzihongle1/article/details/70212243>

code: <https://blog.csdn.net/LYJwonderful/article/details/80304660>

##### 动态规划

基本思想也是将待求解问题分解成若干个子问题，先求解子问题，然后从[这些子](https://baike.baidu.com/item/%E8%BF%99%E4%BA%9B%E5%AD%90)问题的解得到原问题的解

动态规划一般可分为线性动规，区域动规，树形动规，背包动规四类。

举例：

线性动规：拦截导弹，合唱队形，挖地雷，建学校，剑客决斗等；

区域动规：石子合并， 加分二叉树，统计单词个数，炮兵布阵等；

树形动规：贪吃的九头龙，二分查找树，聚会的欢乐，数字三角形等；

背包问题：01背包问题，完全背包问题，分组背包问题，二维背包，装箱问题等

其他：连续最长子数和，最长递增字符串等

##### 贪心算法

在对问题求解时，总是做出在当前看来是最好的选择。也就是说，不从整体最优上加以考虑，他所做出的是在某种意义上的局部最优解。

##### Hash

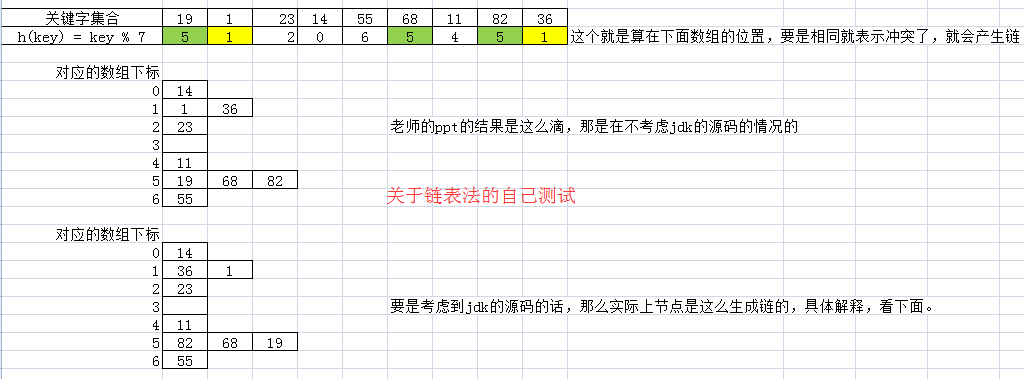
一种将任意长度的消息压缩到某一固定长度的消息摘要的函数。一般用于加密和快速查找

hash冲突解决方法：开放定址法（线性探测或者随机探测），再hash法，链地址法（当前位置加子节点）

线性探测

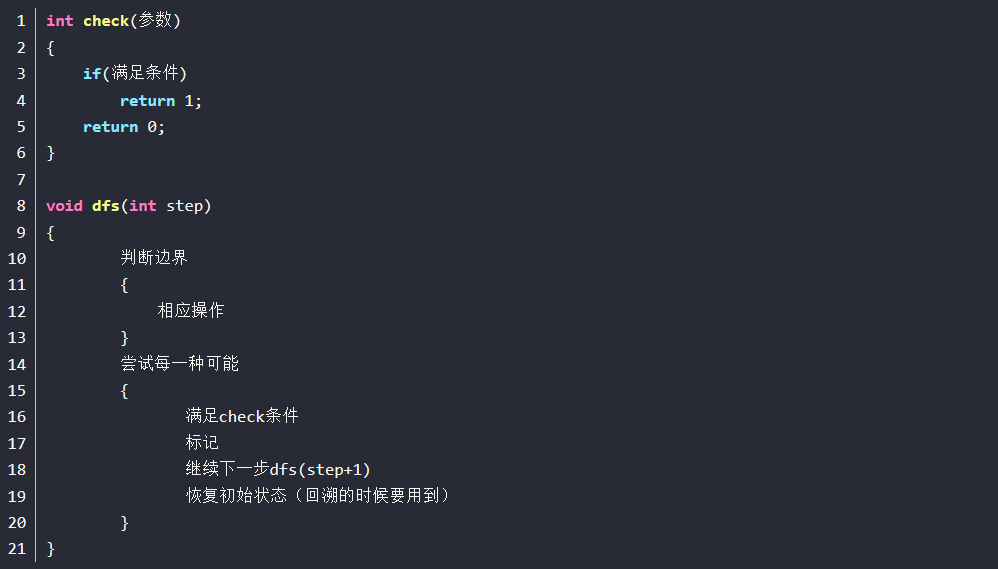


链地址法



常见题：数组中出现n次的数，字符串中第一个只出现一次的字符等

##### 图:DFS（栈）&BFS（队列）



##### 布隆过滤器

bitmap实现，优点省内存，缺点有误算率，原理是K次hash填充位图

##### 一致性hash

增加或者删除只影响临近节点，数据分布的一种方式

其目的是为了克服传统哈希分布在服务器节点数量变化时大量数据迁移的问题

##### 环形缓冲区ringbuffer

它是一个环（首尾相接的环），你可以把它用做在不同上下文（线程）间传递数据的buffer

是为解决某些特殊情况下的竞争问题提供了一种免锁的方法。这种特殊的情况就是当生产者和消费者都只有一个，而在其它情况下使用它也是必须要加锁的。对应在Linux内核中有对它的定义：

struct kfifo {

         unsigned char \*buffer;

         unsigned int size;

         unsigned int in;

         unsigned int out;

         spinlock\_t \*lock;

};

其中buffer指向存放数据的缓冲区，size是缓冲区的大小，in是写指针下标，out是读buffer指针下标，lock是加到struct kfifo上的自旋锁（上面说不加锁不是这里的锁），防止多个进程并发访问此数据结构。当in==out时，说明缓冲区为空；当(in-out)==size时，说明缓冲区已满。

优点：

在可靠消息传递方面有很好的性能

它是数组，所以要比链表快，而且有一个容易预测的访问模式（内存连续），你可以为数组预先分配内存，使得数组对象一直存在

##### 并查集

并查集是一种树型的数据结构，用于处理一些不相交集合（Disjoint Sets）的合并及查询问题。常常在使用中以森林来表示。典型用途是解决朋友圈问题，A和B是朋友，B和C是朋友，则ABC组成一个朋友圈。

手写代码：字符串操作，链表操作(合并翻转是否有环等)，全排列问题，快排，最小公共祖先等

最小公共祖先：递归

TreeNode\* lowestCommonAncestor(TreeNode\* root, TreeNode\* p, TreeNode\* q) {

if(root==NULL || p==root || q==root) return root;

TreeNode\* left = lowestCommonAncestor(root->left, p, q);

TreeNode\* right = lowestCommonAncestor(root->right, p, q);

if(left && right) return root;

return left? left: right;

}

#### 网络部分

书籍推荐：

TCP/IP协议

计算机自顶向下

##### TCP超时重传

　　原理是在发送某一个数据以后就开启一个计时器，在一定时间内如果没有得到发送的数据报的ACK报文，那么就重新发送数据，直到发送成功为止。

　　影响超时重传机制协议效率的一个关键参数是**重传超时时间**（RTO，Retransmission TimeOut）。RTO的值被设置过大过小都会对协议造成不利影响。

　　（1）RTO设长了，重发就慢，没有效率，性能差。

　　（2）RTO设短了，重发的就快，会增加网络拥塞，导致更多的超时，更多的超时导致更多的重发。

　　连接往返时间（RTT，Round Trip Time），指发送端从发送TCP包开始到接收它的立即响应所消耗的时间。

##### TCP滑动窗口

作用：（1）提供TCP的可靠性；（2）提供TCP的流控特性

TCP的滑动窗口的可靠性也是建立在“确认重传”基础上的。

发送窗口只有收到对端对于本段发送窗口内字节的ACK确认，才会移动发送窗口的左边界。

接收端可以根据自己的状况通告窗口大小，从而控制发送端的接收，进行流量控制。

糊涂窗口综合症

当发送端应用进程产生数据很慢、或接收端应用进程处理接收缓冲区数据很慢，或二者兼而有之；就会使应用进程间传送的报文段很小，特别是有效载荷很

##### TCP拥塞控制

　　拥塞控制是一个全局性的过程； 流量控制是点对点通信量的控制

**TCP拥塞控制4个核心算法：慢开始（slow start）、拥塞避免（Congestion Avoidance）、快速重传（fast retransmit）、快速回复（fast recovery）**

　　拥塞窗口（cwnd，congestion window），其大小取决于网络的拥塞程度，并且动态地在变化。

　　慢开始算法的思路就是，不要一开始就发送大量的数据，先探测一下网络的拥塞程度，也就是说由小到大逐渐增加拥塞窗口的大小。

为了防止cwnd增长过大引起网络拥塞，还需设置一个慢开始门限ssthresh状态变量。ssthresh的用法如下：

当cwnd < ssthresh时，使用慢开始算法。

当cwnd > ssthresh时，改用拥塞避免算法。

       快重传和快恢复

　　 快速重传(Fast retransmit)要求接收方在收到一个失序的报文段后就立即发出重复确认（为的是使发送方及早知道有报文段没有到达对方），而不要等到自己发送数据时捎带确认。

　　 快重传算法规定，发送方只要一连收到3个重复确认就应当立即重传对方尚未收到的报文段，而不必继续等待设置的重传计数器时间到期。

　　  快速恢复(Fast Recovery)

##### time\_wai

time\_wait状态产生的原因:调用close()发起主动关闭的一方，在发送最后一个ACK之后会进入time\_wait的状态，也就说该发送方会保持2MSL时间之后才会回到初始状态。MSL值得是数据包在网络中的最大生存时间

作用：为实现TCP全双工连接的可靠释放

**1）保证A发送的最后一个ACK报文段能够到达B。2）防止“已失效的连接请求报文段”出现在本连接中。**

time\_wait状态如何避免

首先服务器可以设置SO\_REUSEADDR套接字选项来通知内核，如果端口忙，但TCP连接位于TIME\_WAIT状态时可以重用端口。在一个非常有用的场景就是，如果你的服务器程序停止后想立即重启，而新的套接字依旧希望使用同一端口，此时SO\_REUSEADDR选项就可以避免TIME\_WAIT状态。

TCP协议如何来保证传输的可靠性

1.分段：将数据截断为合理的长度

2.超时重发

3.保持首部校验和

4.对失序数据进行重新排序

5.对于重复数据，能够丢弃重复数据

6.滑动窗口：流量控制

7.对于收到的请求，给出确认响应

##### UDP如何实现可靠传输

传输层无法保证数据的可靠传输，只能通过应用层来实现了。实现的方式可以参照tcp可靠性传输的方式，只是实现不在传输层，实现转移到了应用层。

最简单的方式是在应用层模仿传输层TCP的可靠性传输。下面不考虑拥塞处理，可靠UDP的简单设计。

* 1、添加seq/ack机制，确保数据发送到对端
* 2、添加发送和接收缓冲区，主要是用户超时重传。
* 3、添加超时重传机制。

##### 三次握手与四次挥手

三次握手的目的是建立可靠的通信信道，说到通讯，简单来说就是数据的发送与接收，而三次握手最主要的目的就是双方确认自己与对方的发送与接收机能正常。《计算机网络》第四版中讲“三次握手”的目的是“为了防止已失效的连接请求报文段突然又传送到了服务端，因而产生错误”

为什么A还要发送一次确认呢？可以二次握手吗？

主要为了防止已失效的连接请求报文段突然又传送到了B，因而产生错误

根本原因是，一方发送FIN只表示自己发完了所有要发的数据，但还允许对方继续把没发完的数据发过来。

三次握手的目的是“为了防止已经失效的连接请求报文段突然又传到服务端，因而产生错误

为什么连接的时候是三次握手，关闭的时候却是四次握手？

答：因为当Server端收到Client端的SYN连接请求报文后，可以直接发送SYN+ACK报文。其中ACK报文是用来应答的，SYN报文是用来同步的。但是关闭连接时，当Server端收到FIN报文时，很可能并不会立即关闭SOCKET，所以只能先回复一个ACK报文，告诉Client端，"你发的FIN报文我收到了"。只有等到我Server端所有的报文都发送完了，我才能发送FIN报文，因此不能一起发送。故需要四步握手。

##### [[TCP/IP] TCP第三次握手失败怎么办](https://www.cnblogs.com/taoshihan/p/11215289.html)

**三次握手**

客户端 ==> SYN是1同步 ,ACK确认标志是0,seq序号是x ==> 服务器  
客户端 <== SYN是1同步 ,ACK确认标志是1,seq序号是y,ack确认号是x+1 <==服务器  
客户端 ==> ACK确认标志是1,seq序号是x+1,ack确认号是y+1 ==>服务器

server端发送了SYN+ACK报文后就会启动一个定时器，等待client返回的ACK报文。如果第三次握手失败的话client给server返回了ACK报文，server并不能收到这个ACK报文。那么server端就会启动超时重传机制，超过规定时间后重新发送SYN+ACK，重传次数根据/proc/sys/net/ipv4/tcp\_synack\_retries来指定，默认是5次。如果重传指定次数到了后，仍然未收到ACK应答，那么一段时间后，server自动关闭这个连接。但是client认为这个连接已经建立，如果client端向server写数据，server端将以RST包响应

RST标示复位、用来异常的关闭连接。

1. 发送RST包关闭连接时，不必等缓冲区的包都发出去，直接就丢弃缓冲区中的包，发送RST。

2. 而接收端收到RST包后，也不必发送ACK包来确认。

二、什么时候发送RST包

1. 建立连接的SYN到达某端口，但是该端口上没有正在监听的服务。

2. TCP收到了一个根本不存在的连接上的分节。

3. 请求超时。 使用setsockopt的SO\_RCVTIMEO选项设置recv的超时时间。接收数据超时时，会发送RST包。



FIN\_WAIT\_1: 这个状态要好好解释一下，其实FIN\_WAIT\_1和FIN\_WAIT\_2状态的真正含义都是表示等待对方的FIN报文。而这两种状态的区别 是：FIN\_WAIT\_1状态实际上是当SOCKET在ESTABLISHED状态时，**它想主动关闭连接，**向对方发送了FIN报文，此时该SOCKET即 进入到FIN\_WAIT\_1状态。**而当对方回应ACK报文后，则进入到FIN\_WAIT\_2状态**，当然在实际的正常情况下，无论对方何种情况下，都应该马 上回应ACK报文，所以FIN\_WAIT\_1状态一般是比较难见到的，而FIN\_WAIT\_2状态还有时常常可以用netstat看到。   
 FIN\_WAIT\_2：上面已经详细解释了这种状态，实际上FIN\_WAIT\_2状态下的SOCKET，表示半连接，也即有一方要求close连接，但另外还告诉对方，我暂时还有点数据需要传送给你，稍后再关闭连接。

 TIME\_WAIT: 表示收到了对方的FIN报文，并发送出了ACK报文，就等2MSL后即可回到CLOSED可用状态了。如果FIN\_WAIT\_1状态下，收到了对方同时带 FIN标志和ACK标志的报文时，可以直接进入到TIME\_WAIT状态，而无须经过FIN\_WAIT\_2状态。   
 CLOSING: 这种状态比较特殊，实际情况中应该是很少见，属于一种比较罕见的例外状态。正常情况下，当你发送FIN报文后，按理来说是应该先收到（或同时收到）对方的 ACK报文，再收到对方的FIN报文。但是CLOSING状态表示你发送FIN报文后，并没有收到对方的ACK报文，反而却也收到了对方的FIN报文。什 么情况下会出现此种情况呢？其实细想一下，也不难得出结论：那就是如果双方几乎在同时close一个SOCKET的话，那么就出现了双方同时发送FIN报 文的情况，也即会出现CLOSING状态，表示双方都正在关闭SOCKET连接。   
 CLOSE\_WAIT: 这种状态的含义其实是表示在等待关闭。怎么理解呢？当对方close一个SOCKET后发送FIN报文给自己，你系统毫无疑问地会回应一个ACK报文给对 方，此时则进入到CLOSE\_WAIT状态。接下来呢，实际上你真正需要考虑的事情是察看你是否还有数据发送给对方，如果没有的话，那么你也就可以 close这个SOCKET，发送FIN报文给对方，也即关闭连接。所以你在CLOSE\_WAIT状态下，需要完成的事情是等待你去关闭连接。   
 LAST\_ACK: 这个状态还是比较容易好理解的，它是被动关闭一方在发送FIN报文后，最后等待对方的ACK报文。当收到ACK报文后，也即可以进入到CLOSED可用状态了。

 为什么TIME\_WAIT状态还需要等2MSL后才能返回到CLOSED状态？  因为虽然双方都同意关闭连接了，而且握手的4个报文也都发送完毕，按理可以直接回到CLOSED 状态(就好比从SYN\_SENT 状态到ESTABLISH 状态那样)，但是我们必须假想网络是不可靠的，你无法保证你(客户端)最后发送的ACK报文一定会被对方收到，就是说对方处于LAST\_ACK 状态下的SOCKET可能会因为超时未收到ACK报文，而重发FIN报文，所以这个TIME\_WAIT 状态的作用就是用来重发可能丢失的ACK报文。

 关闭TCP连接一定需要4次挥手吗?  不一定，4次挥手关闭TCP连接是最安全的做法。但在有些时候，我们不喜欢TIME\_WAIT 状态(如当MSL数值设置过大导致服务器端有太多TIME\_WAIT状态的TCP连接，减少这些条目数可以更快地关闭连接，为新连接释放更多资源)，这时我们可以通过设置SOCKET变量的SO\_LINGER标志来避免SOCKET在close()之后进入TIME\_WAIT状态，这时将通过发送RST强制终止TCP连接(取代正常的TCP四次握手的终止方式)。但这并不是一个很好的主意，TIME\_WAIT 对于我们来说往往是有利的。

##### TCP和UDP区别

key:TCP是一种面向连接的、可靠的、字节流服务

1.面向链接：TCP面向链接，面向连接意味着两个使用TCP的应用（通常是一个客户和一个服务器）在彼此交换数据之前必须通过三次握手先建立一个TCP连接。在一个TCP中仅有两方彼此通信，多播和广播不能用于TCP。UDP是不可靠的传输，传输前不需要建立链接，可以应用多播和广播实现一对多的通信。

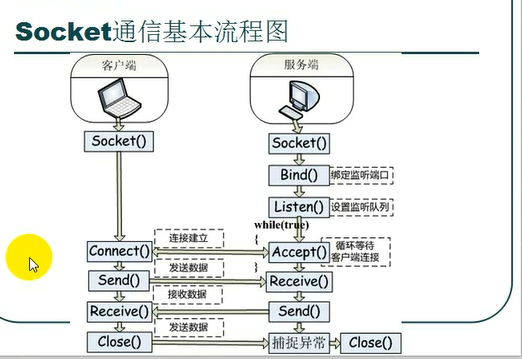
2.可靠性：TCP提供端到端的流量控制，对收到的数据进行确认，采用超时重发，对失序的数据进行重新排序等机制保证数据通信的可靠性。而UDP是一种不可靠的服务，接收方可能不能收到发送方的数据报。

3.TCP是一种流模式的协议，UDP是一种数据报模式的协议。进程的每个输出操作都正好产生一个UDP数据报，并组装成一份待发送的IP数据报。TCP应用程序产生的全体数据与真正发送的单个IP数据报可能没有什么联系。TCP会有粘包和半包的现象。

4.效率上：速度上，一般TCP速度慢，传输过程中需要对数据进行确认，超时重发，还要对数据进行排序。UDP没有这些机制所以速度快。数据比例，TCP头至少20个字节，UDP头8个字节，相对效率高。组装效率上：TCP头至少20个字节，UDP头8个字节，系统组装上TCP相对慢。

5.用途上：用于TCP可靠性，http，ftp使用。而由于UDP速度快，视频，在线游戏多用UDP，保证实时性

##### Socket 过程



##### 粘包现象

1 什么是粘包现象

**TCP粘包**是指发送方发送的若干包数据到接收方接收时粘成一包，从接收缓冲区看，后一包数据的头紧接着前一包数据的尾。

2 为什么出现粘包现象

**（1）发送方原因**

　　我们知道，TCP默认会使用Nagle算法。而Nagle算法主要做两件事：1）只有上一个分组得到确认，才会发送下一个分组；2）收集多个小分组，在一个确认到来时一起发送。

　　所以，正是Nagle算法造成了发送方有可能造成粘包现象。

**（2）接收方原因**

　　TCP接收到分组时，并不会立刻送至应用层处理，或者说，应用层并不一定会立即处理；实际上，TCP将收到的分组保存至接收缓存里，然后应用程序主动从缓存里读收到的分组。这样一来，如果TCP接收分组的速度大于应用程序读分组的速度，多个包就会被存至缓存，应用程序读时，就会读到多个首尾相接粘到一起的包。

3 什么时候需要处理粘包现象

　　（1）如果发送方发送的多个分组本来就是同一个数据的不同部分，比如一个很大的文件被分成多个分组发送，这时，当然不需要处理粘包的现象；

　　（2）但如果多个分组本毫不相干，甚至是并列的关系，我们就一定要处理粘包问题了。比如，我当时要接收的每个分组都是一个有固定格式的商品信息，如果不处理粘包问题，每个读进来的分组我只会处理最前边的那个商品，后边的就会被丢弃。这显然不是我要的结果。

4 如何处理粘包现象

**（1）发送方**

　　对于发送方造成的粘包现象，我们可以通过关闭Nagle算法来解决，使用TCP\_NODELAY选项来关闭Nagle算法。

**（2）接收方**

　　遗憾的是TCP并没有处理接收方粘包现象的机制，我们只能在应用层进行处理。

**（3）应用层处理**

　　应用层的处理简单易行！并且不仅可以解决接收方造成的粘包问题，还能解决发送方造成的粘包问题。

　　解决方法就是循环处理：应用程序在处理从缓存读来的分组时，读完一条数据时，就应该循环读下一条数据，直到所有的数据都被处理；但是如何判断每条数据的长度呢？

　　两种途径：

　　　　1）格式化数据：每条数据有固定的格式（开始符、结束符），这种方法简单易行，但选择开始符和结束符的时候一定要注意每条数据的内部一定不能出现开始符或结束符；

　　　　2）发送长度：发送每条数据的时候，将数据的长度一并发送，比如可以选择每条数据的前4位是数据的长度，应用层处理时可以根据长度来判断每条数据的开始和结束。

详见：<https://www.cnblogs.com/qiaoconglovelife/p/5733247.html>

##### 长连接和短连接

[1.HTTP/1.0](http://1.HTTP/1.0)默认使用短连接,HTTP/1.1开始,默认使用长连接

2.HTTP协议的长连接和短连接,实质是就是TCP协议的长连接和短连接

3.tcp协议建立连接需要三次握手,这个过程会耗费网络资源和时间

4.tcp协议断开连接需要四次握手,这个过程会耗费网络资源和时间

**一.长连接和短连接**

长连接:是指在一个TCP连接上可以发送多个数据包,但是如果没有数据包发送时,也要双方发检测包以维持这个链连接

短连接:当双方需要有数据交互的时候,就建立一个TCP连接,本次交互完成后,就断开这个连接

注:双方指客户端和服务端

**二.各自优缺点及使用场景**

长连接可以省去较多建立连接和关闭连接的操作,所以比较节省资源和时间,但是长连接如果一直存在的话,第一需要很多探测包的发送来维持这个连接,第二对服务器将是很大的负荷

相对而言,短连接则不需要服务器承担太大负荷,只要存在的连接就都是有用连接,但如果客户端请求频繁,就会在TCP的建立连接和关闭连接上浪费较大的资源和时间

**三.使用场景**

一般而言,像京东,淘宝这些大型的网站,随时随刻有成千上万的用户对服务端发送请求,一般使用短连接,因为如果用长连接的话,用户越来越多,服务器一般扛不住这么多长连接

而即时通讯(比如QQ)一般使用的是长连接(UDP长连接),但并不是永久连接,一般也会有一个保持的时间,比如30分钟,24小时等,因为即时通讯是频繁的发送请求,使用长连接只需要建立一次连接,比较划算,同时再根据业务设置保持时间,超过这个时间就断开连接,也一定程度上保证了服务器的压力不会过大

同理,网络游戏一般也使用长连接,同理即时通讯

如果已经建立了连接，但是客户端突然出现故障了怎么办？

TCP有一个保活计时器，服务器每收到一次客户端的请求后都会重新复位这个计时器

##### keepalive

设置Keepalive参数，检测已中断的客户连接

在TCP中有一个Keep-alive的机制可以检测死连接，原理很简单，TCP会在空闲了一定时间后发送数据给对方

TCP有序：

主机每次发送数据时，TCP就给每个数据包分配一个序列号并且在一个特定的时间内等待接收主机对分配的这个序列号进行确认，如果发送主机在一个特定时间内没有收到接收主机的确认，则发送主机会重传此数据包。

##### TCP的四个定时器

重传定时器、坚持定时器、保活定时器、2MSL定时器

##### RTT & RTO

RTT往返时间测量：  
TCP的超时时间很大程度上是依赖报文段的往返时间。因此测量往返时间显得尤为重要。  
因为链路上的网络流量或者路由器等的存在，往返时间一般不会是一成不变的，可能会经常发生变化。  
最初的TCP规范这样协议：RTT(Round-Trip Time)表示往返时间，用M表示测量到的RTT。  
更新过的RTT = 0.9\*RTT + 0.1\*M 表示，我估计新的往返时间将是，0.9倍的之前的RTT + 0.1倍的新测量的RTT。（0.9叫平滑因子）

RTT = a \* RTT(前一个)+(1-a)RTT(当前)  
得到了新的估计RTT，推荐的重传超时时间RTO（Retransmission TimeOut）的值应该设置为   
RTO = RTT\*b （这里的b是一个推荐值为2的时延离散因子）。超时时间就是大约2倍的往返时间

##### 从输入URL到页面展示的详细过程

1、输入网址

2、DNS解析获取IP

3、建立tcp连接

4、客户端发送HTTP请求

5、服务器处理请求

6、服务器响应请求

7、浏览器展示HTML

8、浏览器发送请求获取其他在HTML中的资源。

##### [TCP之种种连接异常](https://www.cnblogs.com/wanpengcoder/p/5356776.html)

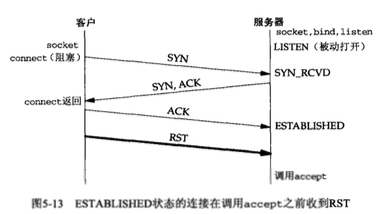
connect出错：

(1) 若TCP客户端没有收到syn分节的响应，则返回ETIMEOUT错误；调用connect函数时，内核发送一个syn，若无响应则等待6s后再发送一个，若仍然无响应则等待24s后在发送一个，若总共等待75s后仍未收到响应则返回本错误；

(2) 若对客户的syn响应是rst，则表明该服务器在我们指定的端口上没有进程在等待与之连接，这是一种硬错误，客户一收到rst马上返回ECONNREFUSED错误；

(3) 若客户发送的syn在中间的某个路由器上引发了目的不可达icmp错误，则认为是一种软错误。客户主机内核保存该消息，并按照第一种情况的时间间隔继续发送syn，咋某个规定时间后仍未收到响应，则把保存的消息作为EHOSTUNREACH或者ENETUNREACH错误返回给进程；

2. accept返回前连接中止：



在比较忙的服务器中，在建立三次握手之后，调用accept之前，可能出现客户端断开连接的情况；如，三次握手之后，客户端发送rst，然后服务器调用accept。posix指出这种情况errno设置为CONNABORTED;

注意Berkeley实现中，没有返回这个错误，而是EPROTO，同时完成三次握手的连接会从已完成队列中移除；在这种情况下，如果我们用select监听到有新的连接完成，但之后又被从完成队列中删除，此时如果调用阻塞accept就会产生阻塞；

解决办法：

(1) 使用select监听套接字是否有完成连接的时候，总是把这个监听套接字设置为非阻塞；

(2) 在后续的accept调用中忽略以下错误，EWOULDBLOCK(Berkeley实现，客户中止连接), ECONNABORTED(posix实现，客户中止连接), EPROTO(serv4实现，客户中止连接)和EINTR(如果有信号被捕获)；

3. 服务器进程终止(崩溃)：

在客户端和服务器端建立连接之后，使用kill命令杀死服务器进程，进程终止会关闭所有打开的描述符，这导致了其向客户端发送了一个FIN，而客户端则响应了一个ack，这就完成了tcp连接终止的前半部分，只代表服务器不在发送数据了；但是客户端并不知道服务器端已经终止了，当客户端向服务器写数据的时候，由于服务器进程终止，所以响应了rst，如果我们使用select等方式，能够立即知道当前连接状态；如下：

(1) 如果对端tcp发送数据，那么套接字可读，并且read返回一个大于0的值(读入字节数)；

(2) 如果对端tcp发送了fin(对端进程终止)，那么该套接字变为可读，并且read返回0(EOF)；

(3) 如果对端tcp发送rst(对端主机崩溃并重启)，那么该套接字变为可读，并且read返回-1，errno中含有确切错误码；

4. sigpipe信号：

当一个进程向某个收到rst的套接字执行写操作的时候，内核向该进程发送一个SIGPIPE信号，该信号的默认行为是终止进程，因此进程必须捕获它以免不情愿的被终止；

不论进程是捕捉了该信号并从信号处理函数中返回，还是简单忽略该信号，写操作都讲返回EPIPE错误；

5. 服务器主机崩溃：

建立连接之后，服务器主机崩溃，此时如果客户端发送数据，会发现客户端会在一定时间内持续重传, 试图从服务器接收一个ACK[一般重传12次（源自Berkeley的实现）]，当重传时间超过指定时间后，服务器仍然没有响应，那么返回的是ETIMEDOUT；

6. 服务器主机不可达：

建立连接之后，服务器主机未崩溃，但是由于中间路由器故障灯，判定主机或网络不可达，此时如果客户端发送数据，会发现客户端会在一定时间内持续重传，视图从服务器端收到数据的ack，当重传时间超过指定时间后，服务器仍然没有响应，那么返回的是EHOSTUNREACH或ENETUNREACH；

 (5,6都属于服务端进程崩溃)

7. 服务器主机崩溃后重启：

当服务器主机崩溃重启后，之前所有的tcp连接丢失，此时服务器若收到来自客户端的数据，会响应一个rst；客户端调用read将返回一个ECONNRESET错误；

8. 服务器主机关机：

系统关机时，init进程给所有进程发送SIGTERM信号，等待固定的时间，然后给所有仍在运行的进程发送SIGKILL信号，我们的进程会被SIGTERM或者SIGKILL信号终止，所以与前面服务器进程终止相同，进程关闭所有描述符，并发送fin，完成服务器端的半关闭；

#### 数据库

##### 索引

**1.索引的优点和缺点**

建立索引的优点

1.大大加快数据的检索速度;

2.创建唯一性索引，保证数据库表中每一行数据的唯一性;

3.加速表和表之间的连接;

4.在使用分组和排序子句进行数据检索时，可以显著减少查询中分组和排序的时间。

索引的缺点

1.索引需要占物理空间。

2.当对表中的数据进行增加、删除和修改的时候，索引也要动态的维护，降低了数据的维护速度。

**索引类型**

唯一索引

唯一索引是不允许其中任何两行具有相同索引值的索引。

主键索引

数据库表经常有一列或列组合，其值唯一标识表中的每一行。该列称为表的主键。

在数据库关系图中为表定义主键将自动创建主键索引，主键索引是唯一索引的特定类型。该索引要求主键中的每个值都唯一。当在查询中使用主键索引时，它还允许对数据的快速访问。

聚集索引

在聚集索引中，表中行的物理顺序与键值的逻辑（索引）顺序相同。一个表只能包含一个聚集索引。

如果某索引不是聚集索引，则表中行的物理顺序与键值的逻辑顺序不匹配。与非聚集索引相比，聚集索引通常提供更快的数据访问速度

##### 乐观锁与悲观锁

悲观锁：假定会发生并发冲突，屏蔽一切可能违反数据完整性的操作。悲观锁的并发访问性不好。

乐观锁：假设不会发生并发冲突，只在提交操作时检查是否违反数据完整性。乐观锁不能解决脏读的问题。

##### 数据库范式

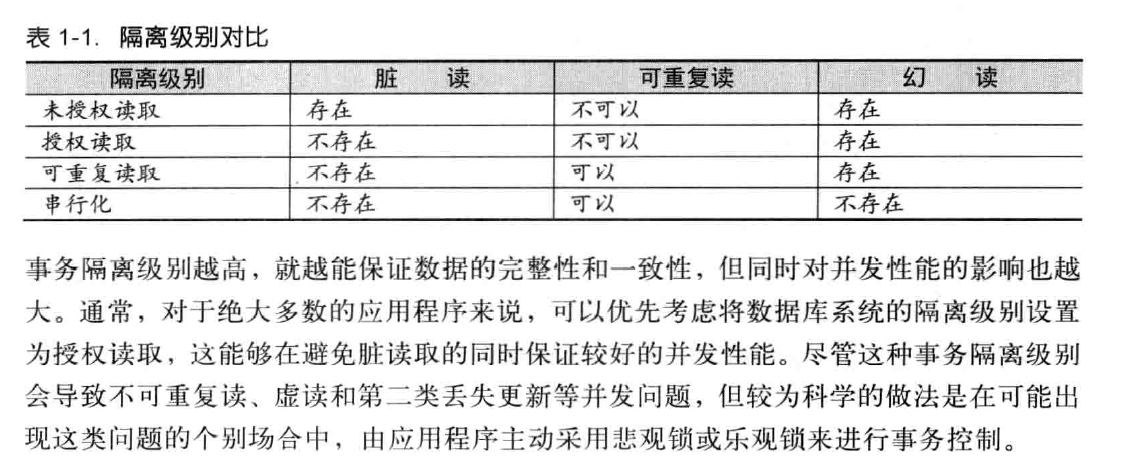
1NF的定义为：符合1NF的关系中的每个属性都不可再分,1NF是所有关系型数据库的最基本要求，

2NF在1NF的基础之上，消除了非主属性对于码的部分函数依赖。

3NF在2NF的基础之上，消除了非主属性对于码的传递函数依赖。也就是说， 如果存在非主属性对于码的传递函数依赖，则不符合3NF的要求。

BCNF范式在 3NF 的基础上消除主属性对于码的部分与传递函数依赖

##### 事务性：ACID



##### innodb和myisam的区别

innodb，聚集索引，支持外键和事务(commit)、回滚(rollback)和崩溃修复能力(crash recovery capabilities)的事务安全(transaction-safe (ACID compliant))，不支持全文索引，不支持计数，统计的时候会遍历，写效率高

myisam，非聚集索引，不支持外键事务，支持全文索引，支持计数，查询效果较好（读）

事务性以及隔离级别

ACID  未授权读取 授权读取（oracle） 可重复读（mysql）  串行性

levelDB：LSM树

mysql：innodb和myisam

#### 系统部分

书籍推荐：

UNIX环境高级编程

深入理解计算机系统

##### select,poll,epoll

时间复杂度：O(n),O(n),O(1)

select，poll，epoll都是IO多路复用的机制。I/O多路复用就通过一种机制，可以监视多个描述符，一旦某个描述符就绪（一般是读就绪或者写就绪），

能够通知程序进行相应的读写操作。但select，poll，epoll本质上都是同步I/O，因为他们都需要在读写事件就绪后自己负责进行读写，也就是说这个

读写过程是阻塞的，而异步I/O则无需自己负责进行读写，异步I/O的实现会负责把数据从内核拷贝到用户空间。

epoll:

epoll有EPOLLLT和EPOLLET两种触发模式，LT是默认的模式，ET是“高速”模式。LT模式下，只要这个fd还有数据可读，每次 epoll\_wait都会返回它的事件，提醒用户程序去操作，而在ET（边缘触发）模式中，它只会提示一次，直到下次再有数据流入之前都不会再提示了，无 论fd中是否还有数据可读。所以在**ET模式下，read一个fd的时候一定要把它的buffer读光**，也就是说一直读到read的返回值小于请求值，或者 遇到EAGAIN错误。还有一个特点是，epoll使用“事件”的就绪通知方式，通过epoll\_ctl注册fd，一旦该fd就绪，内核就会采用类似callback的回调机制来激活该fd，epoll\_wait便可以收到通知。

epoll为什么要有EPOLL ET触发模式？

如果采用EPOLLLT模式的话，系统中一旦有大量你不需要读写的就绪文件描述符，它们每次调用epoll\_wait都会返回，这样会大大降低处理程序检索自己关心 的就绪文件描述符的效率。而采用EPOLLET这种边沿触发模式的话，当被监控的文件描述符上有可读写事件发生时，epoll\_wait()会通知处理程序去读写。

如果这次没有把数据全部读写完(如读写缓冲区太小)，那么下次调用epoll\_wait()时，它不会通知你，也就是它只会通知你一次，直到该文件描述符上出现

第二次可读写事件才会通知你！！！这种模式比水平触发效率高，系统不会充斥大量你不关心的就绪文件描述符

epoll的优点：

1、没有最大并发连接的限制，能打开的FD的上限远大于1024（1G的内存上能监听约10万个端口）；

2、效率提升，不是轮询的方式，不会随着FD数目的增加效率下降。只有活跃可用的FD才会调用callback函数；

即Epoll最大的优点就在于它只管你“活跃”的连接，而跟连接总数无关，因此在实际的网络环境中，Epoll的效率就会远远高于select和poll。

3、 内存拷贝，利用mmap()文件映射内存加速与内核空间的消息传递；即epoll使用mmap减少复制开销。

select、poll、epoll 区别总结：

1、支持一个进程所能打开的最大连接数

select单个进程所能打开的最大连接数有FD\_SETSIZE宏定义，其大小是32个整数的大小（在32位的机器上，大小就是3232，同理64位机器上FD\_SETSIZE为3264），

当然我们可以对进行修改，然后重新编译内核，但是性能可能会受到影响，这需要进一步的测试。

poll本质上和select没有区别，但是它没有最大连接数的限制，原因是它是基于链表来存储的

epoll虽然连接数有上限，但是很大，1G内存的机器上可以打开10万左右的连接，2G内存的机器可以打开20万左右的连接

2、FD剧增后带来的IO效率问题

select因为每次调用时都会对连接进行线性遍历，所以随着FD的增加会造成遍历速度慢的“线性下降性能问题”。

poll同上

epoll因为epoll内核中实现是根据每个fd上的callback函数来实现的，只有活跃的socket才会主动调用callback，所以在活跃socket较少的情况下，使用epoll没有前面两者

的线性下降的性能问题，但是所有socket都很活跃的情况下，可能会有性能问题。

3、 消息传递方式

select内核需要将消息传递到用户空间，都需要内核拷贝动作

poll同上

epoll:epoll通过内核和用户空间共享一块内存来实现的。

总结：

综上，在选择select，poll，epoll时要根据具体的使用场合以及这三种方式的自身特点。

1、表面上看epoll的性能最好，但是在连接数少并且连接都十分活跃的情况下，select和poll的性能可能比epoll好，毕竟epoll的通知机制需要很多函数回调。

2、select低效是因为每次它都需要轮询。但低效也是相对的，视情况而定，也可通过良好的设计改善

详见：https://www.cnblogs.com/aspirant/p/9166944.html

##### 硬链接&软链接

1)默认不带参数情况下，ln命令创建的是硬链接。

2)硬链接文件与源文件的inode节点号相同，而软链接文件的inode节点号与源文件不同。\*\*\*\*\*\*

3)ln命令不能对目录创建硬链接，但可以创建软链接，对目录的软链接会经常被用到。

4)删除软链接文件,对源文件及硬链接文件无任何影响；

5)删除文件的硬链接文件，对源文件及软链接文件无任何影响；

6)删除链接文件的原文件，对硬链接文件无影响，会导致其软链接失效（红底白字闪烁状）；

7)同时删除原文件及其硬链接文件，整个文件才会被真正的删除。

8)很多硬件设备中的快照功能，使用的就类似硬链接的原理。

9)软链接可以跨文件系统，硬链接不可以跨文件系统

10)软链接以路径的方式存在，硬链接以文件的方式存在（不占用空间）

##### 综合

线程间通信方式：互斥锁、条件变量和信号量

进程间通信方式：5种

死锁条件以及避免方式

全缓冲行缓冲，强符号弱符号，大小端模式以及代码判断

fork考点：COW，子进程返回0以及父进程返回子进程id的原因，和vfork区别，守护进程，执行顺序

##### TLB

页表一般都很大，并且存放在内存中，所以处理器引入MMU后，读取指令、数据需要访问两次内存：首先通过查询页表得到物理地址，然后访问该物理地址读取指令、数据。为了减少因为MMU导致的处理器性能下降，引入了TLB，TLB是Translation Lookaside Buffer的简称，可翻译为“地址转换后援缓冲器”，也可简称为“快表”。简单地说，TLB就是页表的Cache，其中存储了当前最可能被访问到的页表项，其内容是部分页表项的一个副本。只有在TLB无法完成地址翻译任务时，才会到内存中查询页表，这样就减少了页表查询导致的处理器性能下降。

 TLB中的项由两部分组成：标识和数据。标识中存放的是虚地址的一部分，而数据部分中存放物理页号、存储保护信息以及其他一些辅助信息。虚地址与TLB中项的映射方式有三种：全关联方式、直接映射方式、分组关联方式

TLB是一种高速缓存，内存管理硬件使用它来改善虚拟地址到物理地址的转换速度。当前所有的个人桌面，笔记本和服务器处理器都使用TLB来进行虚拟地址到物理地址的映射。使用TLB内核可以快速的找到虚拟地址指向物理地址，而不需要请求RAM内存获取虚拟地址到物理地址的映射关系。这与data cache和instruction caches有很大的相似之处

详见：<https://www.cnblogs.com/alantu2018/p/9000777.html>

##### 进程和线程

线程和进程各自有什么区别和优劣呢？

进程是资源分配的最小单位，线程是程序执行的最小单位。

进程有自己的独立地址空间，每启动一个进程，系统就会为它分配地址空间，建立数据表来维护代码段、堆栈段和数据段，这种操作非常昂贵。而线程是共享进程中的数据的，使用相同的地址空间，因此CPU切换一个线程的花费远比进程要小很多，同时创建一个线程的开销也比进程要小很多。

线程之间的通信更方便，同一进程下的线程共享全局变量、静态变量等数据，而进程之间的通信需要以通信的方式（IPC)进行。不过如何处理好同步与互斥是编写多线程程序的难点。

多进程程序更健壮，多线程程序只要有一个线程死掉，整个进程也死掉了，而一个进程死掉并不会对另外一个进程造成影响，因为进程有自己独立的地址空间。

**为什么要使用线程？与进程相比有哪些好处**：

1,和进程相比，它是一种非常"节俭"的多任务操作方式。在linux[系统](https://www.2cto.com/os/)下，启动一个新的进程必须分配给它独立的地址空间，建立众多的数据表来维护它的代码段、堆栈段和数据段，这是一种"昂贵"的多任务工作方式。  
2,运行于一个进程中的多个线程，它们之间使用相同的地址空间，而且线程间彼此切换所需时间也远远小于进程间切换所需要的时间。据统计，一个进程的开销大约是一个线程开销的30倍左右。(较轻的上下文切换开销- 不用切换地址空间，不用更改寄存器，不用刷新TLB。)  
3,线程间方便的通信机制。对不同进程来说，它们具有独立的数据空间，要进行数据的传递只能通过进程间通信的方式进行，这种方式不仅费时，而且很不方便。线程则不然，由于同一进城下的线程之间贡献数据空间，所以一个线程的数据可以直接为其他线程所用，这不仅快捷，而且方便。(创建速度快，方便高效的数据共享)  
  
除以上优点外，多线程程序作为一种多任务、并发的工作方式，还有如下优点：  
1、使多CPU系统更加有效。操作系统会保证当线程数不大于CPU数目时，不同的线程运行于不同的CPU上。  
2、改善程序结构。一个既长又复杂的进程可以考虑分为多个线程，成为几个独立或半独立的运行部分，这样的程序才会利于理解和修改。

  一个进程中的所有线程共享该进程的地址空间，但它们有各自独立的（私有的）栈(stack)

  堆：　是大家共有的空间  分全局堆和局部堆。全局堆就是所有没有分配的空间，局部堆就是用户分配的空间

 栈：是个线程独有的，保存其运行状态和局部自动变量的

##### 五种I/O 模式

阻塞I/O           (Linux下的I/O操作默认是阻塞I/O，即open和socket创建的I/O都是阻塞I/O)

非阻塞 I/O        (可以通过fcntl或者open时使用O\_NONBLOCK参数，将fd设置为非阻塞的I/O)

I/O 多路复用     (I/O多路复用，通常需要非阻塞I/O配合使用)

信号驱动 I/O    (SIGIO)

异步 I/O

##### 同步与异步

同步和异步关注的是消息通信机制

所谓同步，就是在发出一个\*调用\*时，在没有得到结果之前，该\*调用\*就不返回。但是一旦调用返回，就得到返回值了。换句话说，就是由\*调用者\*主动等待这个\*调用\*的结果。

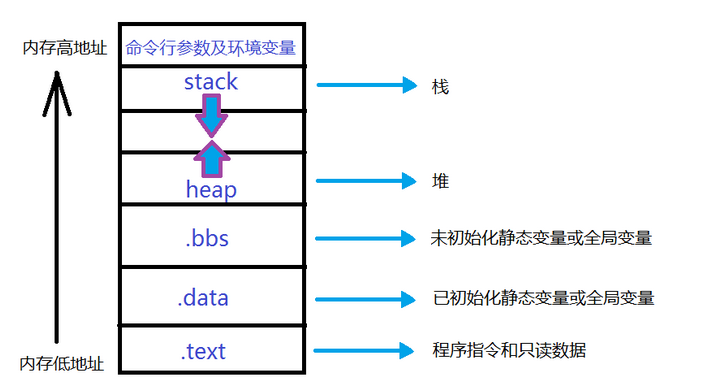
而异步则是相反，\*调用\*在发出之后，这个调用就直接返回了，所以没有返回结果。换句话说，当一个异步过程调用发出后，调用者不会立刻得到结果。而是在\*调用\*发出后，\*被调用者\*通过状态、通知来通知调用者，或通过回调函数处理这个调用

##### 阻塞与非阻塞

阻塞和非阻塞关注的是程序在等待调用结果（消息，返回值）时的状态.

阻塞调用是指调用结果返回之前，当前线程会被挂起。调用线程只有在得到结果之后才会返回。  
 非阻塞调用指在不能立刻得到结果之前，该调用不会阻塞当前线程。

##### 内存分布



数据区：放已初始化的全局变量、静态变量（全局和局部）、常量数据。

堆栈区别

缓存级别，内存释放，数据类型等角度

内存问题

内存溢出，内存泄漏，内存重复释放等

段错误原因

非法访问内存，比如向只读区写入数据，向非法地址写数据，栈溢出等

动态链接和静态链接的区别、

动态链接是只建立一个引用的接口，而真正的代码和数据存放在另外的可执行模块中，在运行时再装入；而静态链接是把所有的代码和数据都复制到本模块中，运行时就不再需要库了。

#### 内核部分

书籍：Linux内核设计与实现

<https://www.cnblogs.com/wang_yb/p/3514730.html>

重点知识：虚拟文件系统，进程调度，内存管理机制(slab)

#### Linux C/C++（部分）

书籍：

C++ primer 5th

6E（effective系列，exceptional系列）

深入探索C++对象模型

<https://www.cnblogs.com/kevinCK/p/4788161.html>

##### [new/delete](http://1.new/delete)和malloc/free的区别

1. malloc和free是库函数，而new和delete是C++操作符；

2. new自己计算需要的空间大小，比如’int \* a = new，malloc需要指定大小，例如’int \* a = malloc(sizeof(int))’；

3. new在动态分配内存的时候可以初始化对象，调用其构造函数，delete在释放内存时调用对象的析构函数。而malloc只分配一段给定大小的内存，并返回该内存首地址指针，如果失败，返回NULL。

4. new是C++操作符，是关键字，而operate new是C++库函数

5. opeartor new /operator delete可以重载，而malloc不行

6. new可以调用malloc来实现，但是malloc不能调用new来实现

7. 对于数据C++定义new[]专门进行动态数组分配，用delete[]进行销毁。new[]会一次分配内存，然后多次调用构造函数；delete[]会先多次调用析构函数，然后一次性释放。

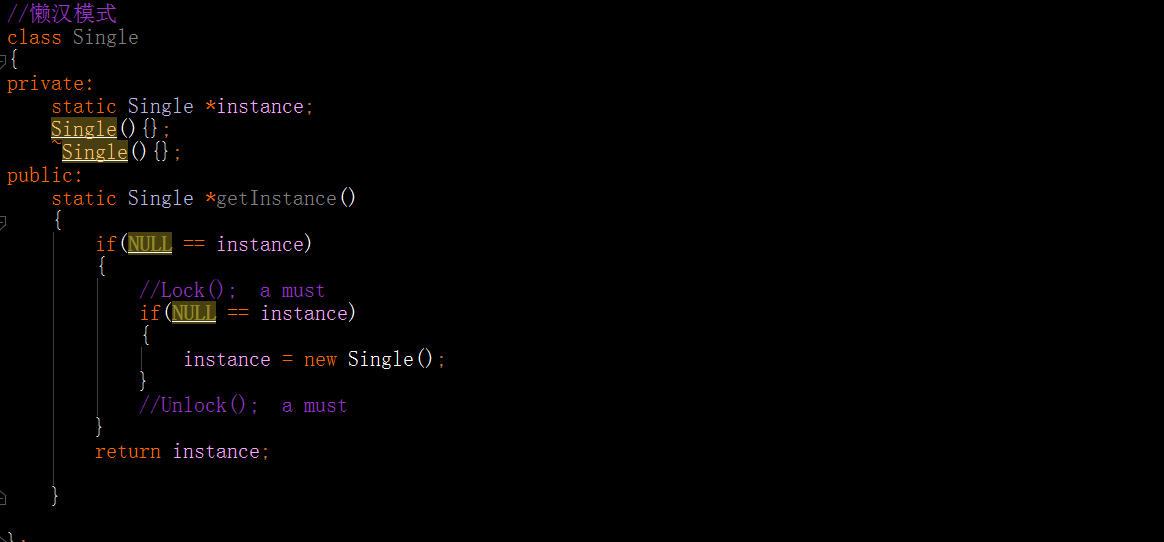
malloc

 malloc的工作机制 ：它有一个将可用的内存块连接为一个长长的列表的所谓空闲链表。

调用malloc函数时，它沿连接表寻找一个大到足以满足用户请求所需要的内存块。然后，将该内存块一分为二（一块的大小与用户请求的大小相等，另一块的大小就是剩下的字节）。接下来，将分配给用户的那块内存传给用户，并将剩下的那块（如果有的话）返回到连接表上。

调用free函数时，它将用户释放的内存块连接到空闲链上。

##### 单例模式



懒汉模式加锁是为了线程安全



##### 左值引用和右值引用

所谓右值引用就是必须绑定到右值的引用，通过&& 获得，右值引用只能绑定到一个将要销毁的对象上，因此可以自由的移动其资源，

左值引用，也就是常规引用，不能绑定到将要转换的表达式，字面常量或者返回的右值表达式上

 引用 & 是一个类型变量的别名。

  通常使用 “ & ”是作为普通的左值引用，左值是可以放在赋值号 “ = ”的左边的值。

  &&是右值引用，右值是只能放在 “ = ” 右边的值，右值没有一个 “ 变量 ”名字。如：

int && k =i+j；

  i+j 返回的值可以视做是放在一个临时存放内存空间里的，这个空间并不能用来赋值，你不能些i+j=5等； && 可以把这种临时的值和存在变量里的值区分开来。

##### STL容器的几种迭代器以及对应的容器

输入迭代器，输出迭代器，前向迭代器，双向迭代器，随机访问迭代器）

顺序容器：vector,deque是随机访问迭代器；list是双向迭代器

容器适配器：stack,queue,priority\_queue没有迭代器

关联容器：set,map,multiset,multimap是双向迭代器

unordered\_set,unordered\_map,unordered\_multiset,unordered\_multimap是前向迭代器

##### 智能指针

shared\_ptr:允许多个指针指向同一个对象

unique\_ptr:独占所指向的对象

weak\_ptr:弱引用，指向shared\_ptr管理的对象

shared\_ptr<int>/auto t = make\_shared<int>(5);

场景：

程序不知道需要多少个对象

程序不知道对象的准确类型

程序在多个对象间共享数据

作用：存储指向动态分配（堆）对象指针的类，用于生存期控制,能够确保自动正确的销毁动态分配的对象，防止内存泄露

实现:维护一个指针和计数器(+1,-1)

##### 深拷贝和浅拷贝

浅拷贝只是对指针的拷贝，拷贝后两个指针指向同一个内存空间

深拷贝不但对指针进行拷贝，而且对指针指向的内容进行拷贝，经深拷贝后的指针是指向两个不同地址的指针

对含有指针成员的对象进行拷贝时，必须要自己定义拷贝构造函数，使拷贝后的对象指针成员有自己的内存空间，即进行深拷贝，这样就避免了内存泄漏发生

 class Student

 {

 private:

 int num;

 char \*name;

 public:

 Student();

 ~Student();

 Student(const Student &s);//拷贝构造函数，const防止对象被改变

 };

 Student::Student(const Student &s)

 {

 name = new char(20);

 memcpy(name, s.name, strlen(s.name));

 cout << "copy Student" << endl;

 }

##### Const修饰符

**const int \*p;**  //p所指向的变量值不能被修改。(\*p)是const

p=&i;   //p指向i；\*p=8;  (错误）

**const int \*p和int const \*p是相同的**

**int \*const p;**  //表示一旦得到了某个变量的地址，不能再指向其他变量

int i,j;   p=&i; \*p=8;(正确) p=&j;  (错误)

##### 初始化列表的使用场景以及效率高的原因

声明为引用类型的成员变量

成员变量或基类未声明默认构造函数

声明为const类型的成员变量

原因：少调用一次构造函数，避免产生临时变量

##### 综合

i++和++i哪个效率高及原因

++i，直接返回引用少产生一个临时变量（i++中需要一个临时变量用来保存改变前的值）

空类大小以及原因

1 为对象保留独一无二的内存

参数一般传递引用，效率更好，传值意味着需要拷贝副本，基本类型都可以

是否能在构造函数，析构函数中抛出异常

一.  析构函数

     参照《Effective C++》中条款08：别让异常逃离析构函数。

　  总结如下：

     1. 不要在析构函数中抛出异常！虽然C++并不禁止析构函数抛出异常，但这样会导致程序过早结束或出现不明确的行为。

     2. 如果某个操作可能会抛出异常，class应提供一个普通函数（而非析构函数），来执行该操作。目的是给客户一个处理错误的机会。

     3. 如果析构函数中异常非抛不可，那就用try catch来将异常吞下，但这样方法并不好，我们提倡有错早些报出来。

二.  构造函数

     总结如下：

     1. 构造函数中抛出异常，会导致析构函数不能被调用，但对象本身已申请到的内存资源会被系统释放（已申请到资源的内部成员变量会被系统依次逆序调用其析构函数）。

     2. 因为析构函数不能被调用，所以可能会造成内存泄露或系统资源未被释放。

     3. 构造函数中可以抛出异常，但必须保证在构造函数抛出异常之前，把系统资源释放掉，防止内存泄露。（如何保证？？？使用auto\_ptr？？？）

最后总结如下：

1. 构造函数中尽量不要抛出异常，能避免的就避免，如果必须，要考虑不要内存泄露！

2. 不要在析构函数中抛出异常

##### 友元函数

下面两种情况需要使用友元函数：

（1）运算符重载的某些场合需要使用友元。

（2）两个类要共享数据的时候。

使用友元函数的优缺点

优点：能够提高效率，表达简单、清晰。 节约了调用类的成员函数开销

缺点：友元函数破环了封装机制

友元函数的位置

友元函数是类外的函数，所以它的声明可以放在类的私有段或公有段且没有区别

##### 宏定义和展开、内联函数区别

宏是由预处理器对宏进行替代，而内联函数是通过编译器控制来实现的。

宏定义不检查函数参数，只是展开，相对来说，内联函数会检查参数类型，所以更安全

内联函数是代码被插入到调用者代码处的函数。内联函数通过避免被调用的开销来提高执行效率，尤其是它能够通过调用（“过程化集成”）被编译器优化。内联函数是真正的函数，只是在需要用到的时候，内联函数像宏一样的展开，所以取消了函数的参数压栈，减少了调用的开销。

##### C++ external的作用

在C++中extern还有另外一种作用，用于指示C或者C＋＋函数的调用规范。比如在C＋＋中调用C库函数，就需要在C＋＋程序中用extern “C”声明要引用的函数。这是给链接器用的，告诉链接器在链接的时候用C函数规范来链接。主要原因是C＋＋和C程序编译完成后在目标代码中命名规则不同，用此来解决名字匹配的问题。

##### 重写，重载，多态

重写：在同一个类中，函数名相同，参数列表不同

重载：也叫覆盖，子类重新定义父类中有相同名称相同参数的虚函数，主要是在继承关系中出现的，被重写的函数必须是virtual的，重写函数的访问修饰符可以不同，尽管virtual是private的，子类中重写函数改为public,protected也可以，体现了多态。

a编译时多态性：通过重载函数实现 。  
b运行时多态性：通过虚函数实现

##### private&protected&public

1.类的一个特征就是封装，public和private作用就是实现这一目的。所以：

用户代码（类外）可以访问public成员而不能访问private成员；private成员只能由类成员（类内）和友元访问。

2.类的另一个特征就是继承，protected的作用就是实现这一目的。所以：

protected成员可以被派生类对象访问，不能被用户代码（类外）访问。

private只能由该类中的函数、其友元函数访问,不能被任何其他访问，该类的对象也不能访问

protected: 可以被该类中的函数、子类的函数、以及其友元函数访问,但不能被该类的对象访问

public: 可以被该类中的函数、子类的函数、其友元函数访问,也可以由该类的对象访问

##### 指针和引用区别

1. 指针是一个实体，而引用仅是个别名；

2. 引用使用时无需解引用(\*)，指针需要解引用；

3. 引用只能在定义时被初始化一次，之后不可变；指针可变；

4. 引用没有 const，指针有 const；

5. 引用不能为空，指针可以为空；

##### vector和list的区别

1.vector数据结构

vector和数组类似，拥有一段连续的内存空间，并且起始地址不变。

因此能高效的进行随机存取，时间复杂度为o(1);

但因为内存空间是连续的，所以在进行插入和删除操作时，会造成内存块的拷贝，时间复杂度为o(n)。

另外，当数组中内存空间不够时，会重新申请一块内存空间并进行内存拷贝。

2.list数据结构

list是由双向链表实现的，因此内存空间是不连续的。

只能通过指针访问数据，所以list的随机存取非常没有效率，时间复杂度为o(n);

但由于链表的特点，能高效地进行插入和删除。

##### 构造函数&析构函数

多重类构造和析构的顺序

先调用基类的构造函数，在调用派生类的构造函数

先构造的后析构，后构造的先析构

构造函数能否是虚函数以及原因

不能

简单讲就是没有意义。虚函数的作用在于通过子类的指针或引用来调用父类的那个成员函数。而构造函数是在创建对象时自己主动调用的，不可能通过子类的指针或者引用去调用。

网络上还有一个很普遍的解释是这样的：虚函数相应一个指向vtable虚函数表的指针，但是这个指向vtable的指针事实上是存储在对象的内存空间的。假设构造函数是虚的，就须要通过 vtable来调用，但是对象还没有实例化，也就是内存空间还没有，怎么找vtable呢？所以构造函数不能是虚函数。

析构函数什么时候设置为virtual(有虚函数的时候，实现多态)以及原因(防止内存泄漏)

构造函数

初始化对象的数据成员

拷贝构造函数的参数（const A&）,const一般不省略，防止数据被意外修改，&不可以省略，因为会不断调用拷贝构造函数

基类的析构函数写成virtual虚析构函数：为了防止内存泄漏

基类指针指向了派生类对象，而基类中的析构函数却是非virtual的，之前讲过，虚函数是动态绑定的基础。现在析构函数不是virtual的，因此不会发生动态绑定，而是静态绑定，指针的静态类型为基类指针，因此在delete时候只会调用基类的析构函数，而不会调用派生类的析构函数。这样，在派生类中申请的资源就不会得到释放，就会造成内存泄漏

析构函数在下边3种情况时被调用：

1.对象生命周期结束，被销毁时；

2.delete指向对象的指针时，或delete指向对象的基类类型指针，而其基类虚构函数是虚函数时；

3.对象i是对象o的成员，o的析构函数被调用时，对象i的析构函数也被调用。

c++的static的变量的初始化顺序以及作用

声明顺序就是初始化顺序，C++类里面静态类型成员变量的初始化顺序和声明的顺序不一致，和初始化语句的先后顺序有关

作用：C(限制作用域和设置存储域)，C++(类成员为static，只能被static函数访问，在类中的static成员变量属于整个类所有，对类的所有对象只有一份拷贝。)

##### static\_cast、const\_cast、reinterpret\_cast、dynamic\_cast

static\_cast、const\_cast、reinterpret\_cast、dynamic\_cast

const\_cast：仅用于去掉完全同类型的const,volatile约束

static\_cast：使用类型可在继承的方向上向上或向下转换，不进行安全检查。

dynamic\_cast:动态转换，使用类型可在继承的方向上向上或向下转换，进行安全检查。

reinterpret\_cast：强制类型转换，不改变原有数据，只是重新解释内存

引用几乎都是通过指针来实现的，所以通过引用传递对象实际上是传递指针。因此，如果是一个很小的对象——例如int— — 传值实际上会比传引用更高效。

##### 多态

C++的多态性用一句话概括就是：在基类的函数前加上virtual关键字，在派生类中重写该函数，运行时将会根据对象的实际类型来调用相应的函数。如果对象类型是派生类，就调用派生类的函数；如果对象类型是基类，就调用基类的函数

  1：用virtual关键字申明的函数叫做虚函数，虚函数肯定是类的成员函数。

  2：存在虚函数的类都有一个一维的虚函数表叫做虚表，类的对象有一个指向虚表开始的虚指针。虚表是和类对应的，虚表指针是和对象对应的。

  3：多态性是**一个接口多种实现**，是面向对象的核心，分为类的多态性和函数的多态性。

  4：多态用虚函数来实现，结合动态绑定.

  5:纯虚函数是虚函数再加上 = 0；

  6：抽象类是指包括至少一个纯虚函数的类。

纯虚函数:virtual void fun()=0;即抽象类！必须在子类实现这个函数，即先有名称，没有内容，在派生类实现内容

纯虚函数可以有函数体吗？ 可以，不可实例化

##### virtual函数的实现原理

虚函数表由于一旦产生就具有不变性，所以编译器就会经量把它放到稳定（或者说是只读）的内存区。

C++中虚函数是如何实现的呢？不少资料中都提到过，C++通过虚函数表和虚函数表指针来实现virtual function机制，具体而言：

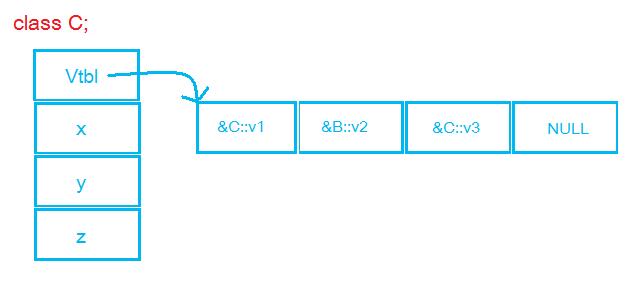
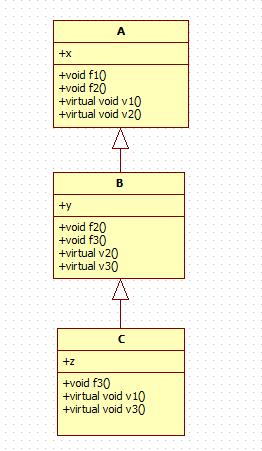
对于一个class，产生一堆指向virtual functions的指针，这些指针被统一放在一个表格中。这个表格被称为虚函数表，英文又称做virtual table（vtbl）。

每一个对象中都添加一个指针，指向相关的virtual table。通常这个指针被称作虚函数表指针（vptr）。出于效率的考虑，该指针通常放在对象实例最前面的位置（第一个slot处）。每一个class所关联的type\_info信息也由virtual table指出（通常放在表格的最前面）。

每当创建一个包含有虚函数的类或从包含有虚函数的类派生一个类时，编译器就会为这个类创建一个虚函数表（VTABLE）保存该类所有虚函数的地址，其实这个VTABLE的作用就是保存自己类中所有虚函数的地址，可以把VTABLE形象地看成一个函数指针数组，这个数组的每个元素存放的就是虚函数的地址。在每个带有虚函数的类 中，编译器秘密地置入一指针，称为v p o i n t e r（缩写为V P T R），指向这个对象的V TA B L E。 当构造该派生类对象时，其成员VPTR被初始化指向该派生类的VTABLE。所以可以认为VTABLE是该类的所有对象共有的，在定义该类时被初始化；而VPTR则是每个类对象都有独立一份的，且在该类对象被构造时被初始化。  
      通过基类指针做虚函数调 用时（也就是做多态调用时），编译器静态地插入取得这个V P T R，并在V TA B L E表中查找函数地址的代码，这样就能调用正确的函数使晚捆绑发生

**虚函数表vtable在Linux/Unix中存放在可执行文件的只读数据段中**

虚函数表是在什么时期建立的？ 表格中的virtual functions地址是如何被建构起来的？在C++中，virtual functions（可经由其class object被调用）可以在编译时期获知



后端面试题：

<https://github.com/chankeh/cpp-backend-reference/blob/master/back-end.md>

<https://github.com/Making-It>

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

New York is 3 hours ahead of California, but it does not make California slow.

纽约时间比加州时间早三个小时，但加州时间并没有变慢。

Someone graduated at the age of 22, but waited 5 years before securing a good job!

有人22岁就毕业了，但等了五年才找到好的工作！

Someone became a CEO at 25, and died at 50.

有人25岁就当上CEO，却在50岁去世。

While another became a CEO at 50, and lived to 90 years.

也有人迟到50岁才当上CEO，然后活到90岁。

Someone is still single, while someone else got married.

有人依然单身，同时也有人已婚。

Obama retires at 55, but Trump starts at 70.

奥巴马55岁就退休，川普70岁才开始当总统。

Absolutely everyone in this world works based on their Time Zone.

世上每个人本来就有自己的发展时区。

People around you might seem to go ahead of you, some might seem to be behind you.

身边有些人看似走在你前面，也有人看似走在你后面。

But everyone is running their own RACE, in their own TIME.

但其实每个人在自己的时区有自己的步程。

Don’t envy them or mock them.

不用嫉妒或嘲笑他们。

They are in their TIME ZONE, and you are in yours!

他们都在自己的时区里，你也是！

Life is about waiting for the right moment to act.

生命就是等待正确的行动时机。

So, RELAX.

所以，放轻松。

You’re not LATE.

你没有落后。

You’re not EARLY.

你没有领先。

You are very much ON TIME, and in your TIME ZONE Destiny set up for you.

在命运为你安排的属于自己的时区里，一切都准时。

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/