1. HashMap and HashTable
   a) AbstractMap -> HashMap, Dictionary -> HashTable
   b) HashMap, iterator, fail-fast，线程不安全; HashTable, enumerator，线程安全
   c) HashMap 可以有 NULL
   d) HashMap 用链表，或者红黑树，HashTable 链表
   e) HashMap 非 synchronized，单线程 HashTable 快
   f) HashSet，无重复的值
   g) HashMap 是基于 hashing 的原理，我们使用 put(key, value)存储对象到 HashMap 中，使用 get(key)从 HashMap 中获取对象。当我们给 put()方法传递键和值时，我们先对键调用 hashCode()方法，返回的 hashCode 用于找到 bucket 位置来储存 Entry 对象。
2. HashMap 扩容过程以及安全问题
   a) Entry 数组当 Hash 冲突时采用拉链法
   b) 当多个线程同时检测到总数量超过门限值的时候就会同时调用 resize 操作，各自生成新的数组并 rehash 后赋给该 map 底层的数组 table，结果最终只有最后一个线程生成的新数组被赋给 table 变量，其他线程的均会丢失。而且当某些线程已经完成赋值而其他线程刚开始的时候，就会用已经被赋值的 table 作为原始数组，这样也会有问题。
   c) Resize：长度是原数组的 2 倍
   d) ReHash：遍历原数组，重新 Hash，Hash 规则改变
   e) 两个线程会造成环形的情况
3. TCP 拥塞机制
   a) 供不应求
   b) 拥塞控制是防止过多的数据注入到网络中，可以使网络中的路由器或链路不致过载，是一个全局性的过程。
      流量控制是点对点通信量的控制，是一个端到端的问题，主要就是抑制发送端发送数据的速率，以便接收端来得及接收。
   c) 标志：
      i. 重传计时器超市
      ii. 接受到三个重复确认
   d) 慢开始，拥塞避免
      i. 试探
      ii. 线性增长
   e) 快重传、快恢复
      i. 立刻重复确认
      ii. 一起使用，使用时慢开始只在连接开始或超时使用
      iii. 三个重复确认时减小门限，重新拥塞避免而不是慢开始
   f) CP 的拥塞控制机制是什么？请简单说说。
      i. 答：我们知道 TCP 通过一个定时器（timer）采样了 RTT 并计算 RTO，但是，如果网络上的延时突然增加，那么，TCP 对这个事做出的应对只有重传数据，然而重传会导致网络的负担更重，于是会导致更大的延迟以及更多的丢包，这就导致了恶性循环，最终形成“网络风暴” —— TCP 的拥塞控制机制就是用于应对这种情况。
      ii. 首先需要了解一个概念，为了在发送端调节所要发送的数据量，定义了一个“拥塞窗口”（Congestion Window），在发送数据时，将拥塞窗口的大小与接收端 ack 的窗口大小做比较，取较小者作为发送数据量的上限。
   g) 拥塞控制主要是四个算法：

i. 1.慢启动：意思是刚刚加入网络的连接，一点一点地提速，不要一上来就把路占满。
连接建好的开始先初始化 cwnd = 1，表明可以传一个 MSS 大小的数据。
每当收到一个 ACK，cwnd++；呈线性上升
每当过了一个 RTT，cwnd = cwnd*2；呈指数让升
阈值 ssthresh（slow start threshold），是一个上限，当 cwnd >= ssthresh 时，就会进入“拥塞避免算法”

ii. 2.拥塞避免：当拥塞窗口 cwnd 达到一个阈值时，窗口大小不再呈指数上升，而是以线性上升，避免增长过快导致网络拥塞。
每当收到一个 ACK，cwnd = cwnd + 1/cwnd
每当过了一个 RTT，cwnd = cwnd + 1
拥塞发生 :当发生丢包进行数据包重传时，表示网络已经拥塞。分两种情况进行处理：
等到 RTO 超时，重传数据包
sshthresh = cwnd /2
cwnd 重置为 1

iii. 3.进入慢启动过程
在收到 3 个 duplicate ACK 时就开启重传，而不用等到 RTO 超时
sshthresh = cwnd = cwnd /2
进入快速恢复算法——Fast Recovery

iv. 4.快速恢复：至少收到了 3 个 Duplicated Acks，说明网络也不那么糟糕，可以快速恢复。
cwnd = sshthresh + 3 * MSS （3 的意思是确认有 3 个数据包被收到了）
重传 Duplicated ACKs 指定的数据包
如果再收到 duplicated Acks，那么 cwnd = cwnd +1
如果收到了新的 Ack，那么，cwnd = sshthresh ，然后就进入了拥塞避免的算法了。

4. TCP 流量控制
   a) 接收端处理数据的速度是有限的，如果发送方的速度太快，就会把缓冲区 u 打满。这个时候如果继续发送数据，就会导致丢包等一系列连锁反应。所以 TCP 支持根据接收端能力来决定发送端的发送速度。这个机制叫做流控制。
   b) 接收端向发送端主机通知自己可以接受数据的大小，这个大小限制就叫做窗口大小
   c) 实际窗口大小是 窗口字段的值左移 M 位;
   d) 接收端将自己可以接收的缓冲区大小放入 TCP 首部中的 “窗口大小” 字段，通过 ACK 端通知发送端;窗口大小字段越大，说明网络的吞吐量越高;
   接收端一旦发现自己的缓冲区快满了，就会将窗口大小设置成一个更小的值通知给发送端;
   发送端接受到这个窗口之后，就会减慢自己的发送速度;
   如果接收端缓冲区满了，就会将窗口置为 0；这时发送方不再发送数据，但是需要定期发送一个窗口探测数据段，使接收端把窗口大小告诉发送端
   当接收端收到从 3001 号开始的数据段后其缓冲区挤满。不得不暂时停止发送数据，之后窗口收到更新通知后才得以继续进行。如果这个通知在途中丢失了，可能导致无法继续通信。所以发送方会时不时发送一个窗口探测的数据段。此数据端仅含一个字节来获取最新的窗口大小。

5. TCP 怎么保证安全机制
   a) 停止等待协议
      i. 每发送完一个分组，就停止发送，等待对方确认，收到确认后再发送下一个分组
      ii. 超时重传

iii. 确认丢失
iv. 确认迟到，丢弃

b) 连续 ARQ 协议
i. 利用滑动窗口，位于滑动窗口内的所有分组都可以连续的发送出去，而不需要逐个等待对方的确认。A 每收到一个确认，就把发送窗口向前滑动一个分组的位置。B 采用累积确认的方式，对按序到达的最后一个分组发送确认（就是最后这个分组的编号），就表示这个分组之前的所有分组都收到了。
ii. 信道利用率高，容易实现，即使确认丢失，也不必重传。
iii. 不能向发送方反映出接收方已经正确收到的所有分组的信息。

c) 会重新排序

6. TCP 头部字段
a) Source Port（源端口）16Bit：源主机的应用程序的端口号
b) Destination Port（目标端口）16Bit：目标主机的应用程序的端口号
c) Sequence Number（序列号）32Bit：发送端发出的不同的 TCP 数据段的序号，数据段在网络传输时，顺序有可能会发生变化。接收端依据序列号按照正确的顺序重组数据。
d) Acknowledge Number（确认序列号）32Bit：用于标识接收端收到的数据段，确认序列号为成功接受的数据段的序列号加 1。
e) Header length（TCP 头部长度）6Bit：TCP 头部正常情况下 20Bytes，如果加 Option 选项，那么 TCP 头部最长为 60Bytes。
f) Flags（标志位）10Bit：URG 表示紧急指针，ACK 表示对于 SYN 的确认，SYN 表示 Request 报文，FIN 字段在传输完成断开连接时使用的。
g) Window（窗口的大小）16Bit：表示主机的缓冲区最大是多少 Bytes，最大值 65525Bytes。用来进行流量控制的。
h) Checksum（校验和）16Bit：校验整个 TCP 数据段，包括 TCP 头部和 TCP 数据。发送端进行计算和记录，接收到进行验证。

7. TCP 什么时候发复位包
a) 重置连接、复位连接，用来关闭异常的连接
b) 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
c) 而接收端收到 RST 包后，也不必发送 ACK 包来确认
d) 到不存在的端口的连接请求
异常终止一个连接
检测半打开连接
e) 什么时候发送 RST 包
i. 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
ii. TCP 收到了一个根本不存在的连接上的分节
iii. 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
f) 尝试手动发送 RST 包
i. 使用 shutdown、close 关闭套接字，发送的是 FIN，不是 RST
ii. 套接字关闭前，使用 sleep。对运行的程序 Ctrl+C，会发送 FIN，不是 RST
iii. 套接字关闭前，执行 return、exit(0)、exit(1)，会发送 FIN、不是 RST

8. TCP & UDP
a) TCP 面向连接（如打电话要先拨号建立连接）。
b) UDP 是无连接的，即发送数据之前不需要建立连接。

c) TCP 提供可靠的服务，通过 TCP 连接传送的数据，无差错，不丢失，不重复，且按序到达。
d) UDP 尽最大努力交付，即不保证可靠交付.

9. HTTP & HTTPS
   a) http 是超文本传输协议，信息是明文传输，https 是具有安全性的 ssl 加密传输协议
   b) http 和 https 使用的是完全不一样的连接方式，端口也不一样，前者默认是 80 端口
   c) http 是无状态的协议，而 https 是由 ssl+http 构建的可进行加密传输、身份认证的网络协议。
   d) http 的无状态是指对事务处理没有记忆能力，缺少状态意味着对后续处理需要的信息没办法提供，只能重新传输这些信息，这样就会增大数据量。另一方面，当不需要信息的时候服务器应答较为快。
10. TCP 心跳包机制
    a) 我们采用的思路是：客户端连接上服务端以后，服务端维护一个在线用户字典，客户端每隔一段时间，向服务器发送一个心跳包，服务器接收到包以后，字典数据的值都会更新为 0；一旦服务端超过规定时间没有接收到客户端发来的包，字典数据将会递增加一，当字典数据的值累计大于等于三，则视为掉线。
11. 虚拟内存
    a) 先将部分程序导入内存，执行完成后导入下一部分程序，给我们的感觉是内存变大了，实际上物理内存的大小并未发生变化
    b) 优点
       i. 将逻辑内存和物理内存分开
       ii. 虚拟内存允许文件和内存通过共享页而为两个或多个进程所共享
    c) 按需调页
       i. 在需要时才调入相应的页
       ii. 支持按需调页的硬件
           1. 页表：该表能够通过有效-无效为或保护位的特定值，将条目设为无效
           2. 次级存储器（交换空间，通常为快速磁盘）
    d) 写时复制
       i. 这种技术允许父进程与子进程开始时共享共享同一个页面，这些页面被标记为为写时复制页，即如果任何一个进程需要对页进行写操作，那么就创建一个共享页的副本
    e) 页面置换
       i. 查找所需页在磁盘上的位置
       ii. 查找一个空闲帧
       iii. 如果有空闲帧，那么就使用它
       iv. 如果没有，使用页置换算法选择一个牺牲帧
       v. 将牺牲帧的内容写到磁盘上，改变页表和帧表
       vi. 将所需页读入空闲帧，改变页表和帧表
       vii. 重新启动程序
       viii. 常见算法：
           1. FIFO 页置换，最先进入的页被置换
           2. 最优页置换 opt，置换最长时间不会使用的页，即能预知将来的情况。但是将来的情况我们无法预知，因此这种算法难以实现

3. LRU 页置换，LRU 置换为每个页关联上上一次使用的时间，当必须置换一页时，LRU 选择最长时间没有使用的页， 最优置换和 LRU 置换都没有 Belady 算法
4. 近似 LRU 页置换
   a) 附加引用位算法，通过在规定时间间隔里记录引用位
   b) 二次机会算法，这种算法只有引用位本身，没有历史位，因此只有一位。当要选择一个页时，检查引用位，如为 0，直接置换，如为 1，则给该页二次机会，同时清 0，寻找下一个 0 位置，所以而二次机会算法的基本算法是 FIFO 算法

(0,0) ——最近未使用且未修改过

(0,1) ——最近未使用但修改过

(1,0)——最近使用但为修改过

(1,1) ——使用且修改过

   c) 增强型二次机会算法，

   d) 基于计数器的页置换， 最不经常使用页置换算法（LFU）——置换出引用次数最小的页

最常使用页置换算法（MFU）——认为最小次数的页可能刚刚调入，且还没有使用

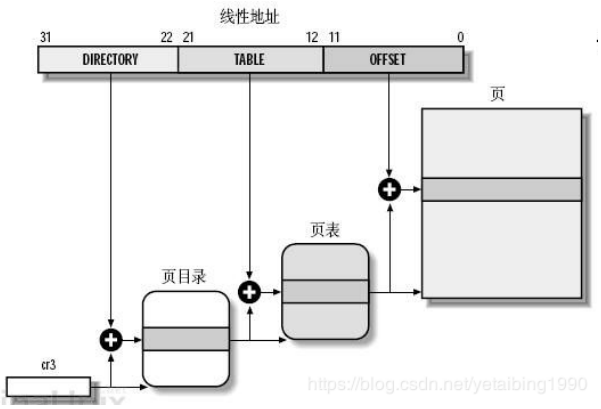
   e) 页缓冲算法， 系统保留一个空闲帧缓冲池，当出现页错误时，会选择牺牲帧，但是牺牲帧写出之前，所需要的页就从缓冲池中读到空闲内存。这样加速了重启

f) 帧分配
   i. 如何在各个进程之间分配一定的空闲内存？
   ii. 分配算法： 平均分配，比例分配
   iii. 全局分配：允许一个进程从所有帧集合中选择一个置换帧
   局部分配：每个进程只能从自己的分配帧中选择置换

g) 系统颠簸
   i. 频繁的页调度操作，原因：当多道程序的程度增加到一定时，会引起颠簸

h) 内核内存的分配
   i. 内核需要为不同大小的数据结构分配内存，因此必须谨慎分配内存
   ii. 有的硬件需要直接和物理内存打交道，因此需要内存常驻在连续的物理页中
   iii. Buddy 系统分配， 内存按 2 的幂的大小进行分配，如果请求大小大于当前 2 的幂内存，那么调整到下一个 2 的幂。
   iv. slab 分配， slab 是由一个或多个物理上连续的页组成。高速缓冲 cache 含有一个或者多个 slab，每个内核数据结构都含有一个 cache，每个 cache 含有内核数据结构的对象实例
      1. 没有因碎片引起的内存浪费。因为每个数据结构都有一个 cache，每个 cache 都有若干个 slab 组成，而每个 slab 又分为若干个和对象大小相同的部分
      2. 内存请求可以快速满足

12. 物理地址，虚拟地址
   a) Dir，Table，Offset——10,10,12



b)

i. 从 cr3 中取出进程的页目录地址（操作系统负责在调度进程的时候，把这个地址装入对应寄存器）；

ii. 根据线性地址前十位，在数组中，找到对应的索引项，因为引入了二级管理模式，页目录中的项，不再是页的地址，而是一个页表的地址。（又引入了一个数组），页的地址被放到页表中去了。

iii. 根据线性地址的中间十位，在页表（也是数组）中找到页的起始地址；

iv. 将页的起始地址与线性地址中最后 12 位相加，得到最终我们想要的葫芦；

13. 进程和线程区别

a) 进程：是执行中一段程序，即一旦程序被载入到内存中并准备执行，它就是一个进程。进程是表示资源分配的的基本概念，又是调度运行的基本单位，是系统中的并发执行的单位。

b) 线程：单个进程中执行中每个任务就是一个线程。线程是进程中执行运算的最小单位。

c) 一个线程只能属于一个进程，但是一个进程可以拥有多个线程。多线程处理就是允许一个进程中在同一时刻执行多个任务

d) 线程是一种轻量级的进程，与进程相比，线程给操作系统带来侧创建、维护、和管理的负担要轻，意味着线程的代价或开销比较小

e) 线程没有地址空间，线程包含在进程的地址空间中。线程上下文只包含一个堆栈、一个寄存器、一个优先权，线程文本包含在他的进程 的文本片段中，进程拥有的所有资源都属于线程。所有的线程共享进程的内存和资源。 同一进程中的多个线程共享代码段(代码和常量)，数据段(全局变量和静态变量)，扩展段(堆存储)。但是每个线程拥有自己的栈段， 寄存器的内容，栈段又叫运行时段，用来存放所有局部变量和临时变量

f) 父和子进程使用进程间通信机制，同一进程的线程通过读取和写入数据到进程变量来通信

g) 进程内的任何线程都被看做是同位体，且处于相同的级别。不管是哪个线程创建了哪一个线程，进程内的任何线程都可以销毁、挂起、恢复和更改其它线程的优先权。线程也要对进程施加控制，进程中任何线程都可以通过销毁主线程来销毁进程，销毁主线程将导致该进程的销毁，对主线程的修改可能影响所有的线程

h) 子进程不对任何其他子进程施加控制，进程的线程可以对同一进程的其它线程施加控制。子进程不能对父进程施加控制，进程中所有线程都可以对主线程施加控制

i) 进程和线程都有 ID/寄存器组、状态和优先权、信息块，创建后都可更改自己的属性，都可与父进程共享资源、都不韪直接访问其他无关进程或线程的资源

14. 为什么需要进程

a) 多道程序在执行时,需要共享系统资源,从而导致各程序在执行过程中出现相互制约的关系,程序的执行表现出间断性的特征.这些特征都是在程序的执行过程中发生的,是动态的过程,

而传统的程序本身是一组指令的集合,是一个静态的概念,无法描述程序在内存中的执行情况,即我们无法从程序的字面上看出它何时执行,何时停顿,也无法看出它与其它执行程序的关系,因此,程序这个静态概念已不能如实反映程序并发执行过程的特征.为了深刻描述程序动态执行过程的性质,人们引入“进程（Process)” 概念

15. 进程切换
    a) 用户模式和特权模式
        i. 用户调用一个操作系统的服务（系统 API 接口）：把系统调用标识符和参数放在一个预定义的区域，然后通过执行一个特殊的指令中断用户级别程序的运行，并把执行模式切换为特权模式
    b) 创建：
        i. 给新进程分配一个惟一的进程标识号
        ii. 给进程分配空间
        iii. 初始化进程控制块
        iv. 设置正确的连接，例如， 把新进程放置到就绪队列
        v. 创建或扩充其他的数据结构。 例如，操作系统执行 top 命令，要能统计进程的 memory 使用率和 cpu 使用率
    c) 进程切换可以在任何时刻发生
        i. 中断： 与某种类型的外部事件有关， 例如完成一次 I/O 操作。它将切换到特权模式，执行中断处理例程
            1. 时钟中断： 操作系统确定当前运行的进程的执行时间是否超过最大允许的时间段。如果超过了，就发出这个时钟中断，发生进程切换
            2. I/O 中断： 发生 I/O 活动的时候，发出这个中断。 如果 I/O 活动是一个或多个进程正在等待的事件，操作系统把所有相应的阻塞态进程转换到就绪态；操作系统必须决定是继续执行当前处于运行态的进程，还是让具有高优先级的就绪态进程抢占这个进程
            3. 内存失效： 处理器访问一页虚拟内存地址，且此地址单元不再内存中时，操作系统必须从辅存中把包含这个地址单元的内存块（页）调入主存。 在发出调入内存块的 I/O 请求后，操作系统可能会执行一个进程切换，以恢复另一个进程的执行，发生内存失效的进程被设置为阻塞态，当想要的块调入内存中时，又会发出一个中断，操作系统收到后会被该进程设置为就绪态
        ii. 陷阱： 正在运行的进程产生一个错误。 例如：非法访问文件。它也将切换到特权模式，执行中断处理例程。操作系统会确定错误是否是致命的。 如果是致命的，当前运行的进程被被切换到退出态，并发生进程切换； 如果不是致命的，操作系统可能会试图恢复或通知用户，也可能进行一次进程切换或者不进行进程切换
16. 索引查询
    a) Linear Search
    b) Binary Search
    c) BST Search
    d) Hash Search
    e) 分块查找
    f) B-tree
        i. 平衡多路查找树
    g) B+tree
        i. 每个节点的指针上限为 2d 而不是 2d+1；

ii. 内节点不存储 data，只存储 key；
iii. 叶子节点不存储指针；
iv. 加载硬盘到内存->发生 IO->Binary Search->锁定指针->加载硬盘

17. 辅助索引和覆盖索引、聚族索引
    a) 聚集索引（主键索引）
        i. 聚集索引就是按照每张表的主键构造一颗 B+树，同时叶子节点中存放的即为整张表的记录数据。
        ii. 聚集索引的叶子节点称为数据页，聚集索引的这个特性决定了索引组织表中的数据也是索引的一部分。
    b) 辅助索引（二级索引）
        i. 叶子节点=键值+书签
    c) 覆盖索引
        i. 解释一： 就是 select 的数据列只用从索引中就能够取得，不必从数据表中读取，换句话说查询列要被所使用的索引覆盖。
        ii. 解释二： 索引是高效找到行的一个方法，当能通过检索索引就可以读取想要的数据，那就不需要再到数据表中读取行了。如果一个索引包含了（或覆盖了）满足查询语句中字段与条件的数据就叫 做覆盖索引。
        iii. 解释三：是非聚集组合索引的一种形式，它包括在查询里的 Select、Join 和 Where 子句用到的所有列（即建立索引的字段正好是覆盖查询语句[select 子句]与查询条件[Where 子句]中所涉及的字段，也即，索引包含了查询正在查找的所有数据）。
18. 乐观锁，悲观锁
    a) 悲观锁：总是假设最坏的情况，每次去拿数据的时候都认为别人会修改，所以每次在拿数据的时候都会上锁，这样别人想拿这个数据就会阻塞直到它拿到锁（共享资源每次只给一个线程使用，其它线程阻塞，用完后再把资源转让给其它线程）。传统的关系型数据库里边就用到了很多这种锁机制，比如行锁，表锁等，读锁，写锁等，都是在做操作之前先上锁
        i. 一般多写的场景下用悲观锁就比较合适
    b) 乐观锁：总是假设最好的情况，每次去拿数据的时候都认为别人不会修改，所以不会上锁，但是在更新的时候会判断一下在此期间别人有没有去更新这个数据，可以使用版本号机制和 CAS 算法实现。乐观锁适用于多读的应用类型，这样可以提高吞吐量，像数据库提供的类似于 write_condition 机制，其实都是提供的乐观锁。
        i. 一般是在数据表中加上一个数据版本号 version 字段，表示数据被修改的次数，当数据被修改时，version 值会加一
        ii. 即 compare and swap（比较与交换），是一种有名的无锁算法
            1. 需要读写的内存值 V
            2. 进行比较的值 A
            3. 拟写入的新值 B
            4. 当且仅当 V 的值等于 A 时，CAS 通过原子方式用新值 B 来更新 V 的值，否则不会执行任何操作（比较和替换是一个原子操作）。一般情况下是一个自旋操作，即不断的重试
        iii. ABA 问题
        iv. 循环时间长开销大（自旋 CAS（不成功就一直循环执行））
19. 表锁，行锁

a) 表锁和行锁锁的粒度不一样，表锁锁住的是一整张表，行锁锁住的是表中的一行数据,行锁是开销最大的锁策略，表锁是开销最小的锁策略。
b) InnoDB 使用的是行级锁，MyISAM 使用的是表级锁。
c) 共享锁又称读锁（S 锁），一个事务获取了共享锁，其他事务可以获取共享锁，不能获取排他锁，其他事务可以进行读操作，不能进行写操作。
d) 排他锁又称写锁（X 锁），如果事务 T 对数据 A 加上排他锁后，则其他事务不能再对 A 加任任何类型的封锁。获准排他锁的事务既能读数据，又能修改数据。

20. 意向锁
    a) 让表锁和行锁共存
    b) 有了意向锁之后，前面例子中的事务 A 在申请行锁（写锁）之前，数据库会自动先给事务 A 申请表的意向排他锁。当事务 B 去申请表的写锁时就会失败，因为表上有意向排他锁之后事务 B 申请表的写锁时会被阻塞。
    c) 当一个事务在需要获取资源的锁定时，如果该资源已经被排他锁占用，则数据库会自动给该事务申请一个该表的意向锁。如果自己需要一个共享锁定，就申请一个意向共享锁。如果需要的是某行（或者某些行）的排他锁定，则申请一个意向排他锁。
    首先可以肯定的是，意向锁是表级别锁。意向锁是表锁是有原因的。

    d) 当我们需要给一个加表锁的时候，我们需要根据意向锁去判断表中有没有数据行被锁定，以确定是否能加成功。如果意向锁是行锁，那么我们就得遍历表中所有数据行来判断。如果意向锁是表锁，则我们直接判断一次就知道表中是否有数据行被锁定了。
21. 死锁
    a) 原因
        i. 互斥条件：一个资源每次只能被一个进程使用。
        ii. 请求与保持条件：一个进程因请求资源而阻塞时，对已获得的资源保持不放。
        iii. 不剥夺条件:进程已获得的资源，在末使用完之前，不能强行剥夺。
        iv. 循环等待条件:若干进程之间形成一种头尾相接的循环等待资源关系。
    b) 语句
        i. 按同一顺序访问对象。
        ii. 避免事务中的用户交互。
        iii. 保持事务简短并在一个批处理中。
        iv. 使用低隔离级别。
        v. 使用绑定连接。
22. 主从复制
    a) What：
        i. 为了减轻主库的压力，应该在系统应用层面做读写分离，写操作走主库，读操作走从库
            1. **同步策略**：Master 要等待所有 Slave 应答之后才会提交（MySql 对 DB 操作的提交通常是先对操作事件进行二进制日志文件写入然后再进行提交）。
            2. **半同步策略**：Master 等待至少一个 Slave 应答就可以提交。
            3. **异步策略**：Master 不需要等待 Slave 应答就可以提交。
            4. **延迟策略**：Slave 要至少落后 Master 指定的时间。
        ii. 基于语句的复制，即 Statement Based Replication（SBR）：记录每一条更改数据的 sql
            1. 优点：binlog 文件较小，节约 I/O，性能较高。

2. 缺点：不是所有的数据更改都会写入 binlog 文件中，尤其是使用 MySQL 中的一些特殊函数（如 LOAD_FILE()、UUID()等）和一些不确定的语句操作，从而导致主从数据无法复制的问题。

iii. 基于行的复制，即 Row Based Replication（RBR）：不记录 sql，只记录每行数据的更改细节

1. 优点：详细的记录了每一行数据的更改细节，这也意味着不会由于使用一些特殊函数或其他情况导致不能复制的问题。
2. 缺点：由于 row 格式记录了每一行数据的更改细节，会产生大量的 binlog 日志内容，性能不佳，并且会增大主从同步延迟出现的几率。

iv. 混合复制（Mixed）

1. 一般的语句修改使用 statment 格式保存 binlog，如一些函数，statement 无法完成主从复制的操作，则采用 row 格式保存 binlog，MySQL 会根据执行的每一条具体的 sql 语句来区分对待记录的日志形式，也就是在 Statement 和 Row 之间选择一种

b) Why:

i. 性能方面：MySQL 复制是一种 Scale-out 方案，也即 “水平扩展” ，将原来的单点负载扩散到多台 Slave 机器中去，从而提高总体的服务性能。

ii. 故障恢复：同时存在多台 Slave 提供读操作服务，如果有一台 Slave 挂掉之后我们还可以从其他 Slave 读取，如果配置了主从切换的话，当 Master 挂掉之后我们还可以选择一台 Slave 作为 Master 继续提供写服务，这大大增加了应用的可靠性。

iii. 数据分析：实时数据可以存储在 Master，而数据分析可以从 Slave 读取，这样不会影响 Master 的性能。

c) How:

i. Master:

1. binlog dump 线程：当主库中有数据更新时，那么主库就会根据按照设置的 binlog 格式，将此次更新的事件类型写入到主库的 binlog 文件中，此时主库会创建 log dump 线程通知 slave 有数据更新，当 I/O 线程请求日志内容时，会将此时的 binlog 名称和当前更新的位置同时传给 slave 的 I/O 线程。

ii. Slave:

1. I/O 线程：该线程会连接到 master，向 log dump 线程请求一份指定 binlog 文件位置的副本，并将请求回来的 binlog 存到本地的 relay log 中，relay log 和 binlog 日志一样也是记录了数据更新的事件，它也是按照递增后缀名的方式，产生多个 relay log（ host_name-relay-bin.000001）文件，slave 会使用一个 index 文件（ host_name-relay-bin.index）来追踪当前正在使用的 relay log 文件。
2. SQL 线程：该线程检测到 relay log 有更新后，会读取并在本地做 redo 操作，将发生在主库的事件在本地重新执行一遍，来保证主从数据同步。此外，如果一个 relay log 文件中的全部事件都执行完毕，那么 SQL 线程会自动将该 relay log 文件删除掉。

23. 红黑树

a) 红黑树保证最长路径不超过最短路径的二倍，因而近似平衡。
b) 每个节点颜色不是黑色，就是红色
c) 根节点是黑色的
d) 如果一个节点是红色，那么它的两个子节点就是黑色的（没有连续的红节点）
e) 对于每个节点，从该节点到其后代叶节点的简单路径上，均包含相同数目的黑色节

f) 插入：
   i. 先插再调色
   ii. 根节点为 NULL，直接插入新节点并将其颜色置为黑色
   iii. 根节点不为 NULL，找到要插入新节点的位置
   iv. 插入新节点
   v. 判断新插入节点对全树颜色的影响，更新调整颜色

24. 一范式，二范式，三范式
   a) 一：符合 1NF 的关系中的每个属性都不可再分
   b) 二：1NF->2NF，要求实体的属性完全依赖于主关键字
   c) 三：要求一个数据库表中不包含已在其它表中已包含的非主关键字信息

25. 程序如何跑起来的

a)

26. Get & Post
   a) HTTP 报文层面： GET 将请求信息放在 URL，请求信息和 url 之间以问号隔开，请求信息的格式为键值对。POST 放在报文体中，想获取请求信息，必须解析报文，因此安全性交 get 方式要高一些。虽然 POST 放在报文体，但通过抓包依然是可以很轻松的获取到信息的，所以从安全性上来讲，两者并没有太多的区别，HTTP 也并不是安全的，具体要解决传输过程中的安全问题，还要靠 HTTPS. GET 有长度限制，POST 则无。
   b) 数据库层面： GET 符合幂等性和安全性，POST 不符合。幂等性的定义就是对数据库的一次操作、多次操作，获得的结果是一致的；安全性则是对数据库的操作没有改变数据库的数据。GET 请求是用来做查询操作的，因此不会改变数据库中原有的数据，而 POST 请求则是提交数据，因此会改变数据库中的数据，其次，POST 请求方式每次获得的结果都有可能不一样，因为 POST 请求是作用在上一级的 URL 上的，则每一次请求都会添加一份新资源，这也是 POST 和 PUT 的最大区别。PUT 是幂等的。
   c) 其他层面： GET 可以被缓存、被存储，而 POST 不行。GET 请求会保存在浏览器的浏览记录中，以 GET 请求的 URL 可以保存为浏览器书签，而 POST 不具备这些功能。Get 表达的是密等的，安全的，因此绝大部分 get 的请求，通常超过 90%都直接被 cdn 缓存了，这能大大减少 web 服务器的负担，而 post 是非密等的，有副作用的操作，所以必须交友 web 服务器处理。

27. Cookie & Session (HTTP)

a) Cookie:

i.

ii. Cookie 是客户端的解决方案。是有服务器发送给客户端的特殊信息，以文本形式存放在客户端。

iii. 当用户使用浏览器访问一个支持 cookie 的网站的时候，用户会提供包括用户名在内的个人信息，并且提交给服务器。紧接着服务器会向客户端回传响应的超文本的同时，也会发回这些个人信息，当然这些信息并不是存放在 http 响应体 responsedbody 中，而是存放在 http 响应头 respnonsedhead，当用户端浏览器接受到来自服务器的响应之后，浏览器会将这些信息存放在一个统一的位置。客户端再次请求的时候，会把 Cookie 回发至服务器。

iv. 客户端发用一个 http 请求到服务端，服务端发送一个 http 响应到客户端，其中包括 set-cookie 头部，客户端在发送一个 http 请求和 cookie 头部到服务器端。此时服务器端发送一个 http 响应到客户端。

b) Session

i. 服务器端的机制，在服务器上保存的信息（服务器使用的类似扇类表的结构来存储信息。）

ii. 当程序要为某个客户端的请求创建一个 session 的时候，服务器先检查这个客户端的请求里是否包含了 session 标识，称为 session id，如果包含，则说明以前已经为此客户端创建过 session，服务器直接根据 session id 检索出来使用。如果客户端请求不包含 session id，则为此客户端创建一个 session，并生成一个与此相关的 session id，session_id 的值既不会凭空消失，又不容易被找到规律的字符串，这个 sessionid 将会在本次响应中，回发给客户端进行保存。把 session 信息回发给客户端，进行保存。

iii. 实现方式：

1. 使用 Cookie：服务器给每一个 session 分配一个唯一的 jsessionid,并通过 cookie 发送给客户端，当客户端发送起新的请求的时候，将在 cookie 头中携带这个 jsessionid，这样服务去能够找到对应的 session。
2. 使用 url 回写：是指服务器在发送给浏览器页面的所有链接中，都携带就 sessionid 的参数，这样客户端点击任何一个链接，都会把 sessionid 带回服务器。
3. Tomcat 默认使用的是 cookie 和 url 回写，如果发现客户端支持 cookie 就继续使用 cookie，停止使用 url 回写，如果发现 cookie 被禁用，就使用 url 回写。
4. Cookie 数据存放在客户的浏览器上，Session 数据放在服务器上
   Session 相对于 Cookie 更安全
   若考虑减轻服务器负担，应当使用 Cookie

28. HTTP/HTTPS

a) HTTP：是互联网上应用最为广泛的一种网络协议，是一个客户端和服务器端请求和应答的标准（TCP），用于从 WWW 服务器传输超文本到本地浏览器的传输协议，它可以使浏览器更加高效，使网络传输减少。

b) HTTPS：是以安全为目标的 HTTP 通道，简单讲是 HTTP 的安全版，即 HTTP 下加入 SSL 层，HTTPS 的安全基础是 SSL，因此加密的详细内容就需要 SSL。

c) HTTPS 协议的主要作用可以分为两种：一种是建立一个信息安全通道，来保证数据传输的安全；另一种就是确认网站的真实性。

d) 区别：

   i. https 协议需要到 ca 申请证书，一般免费证书较少，因而需要一定费用

   ii. http 是超文本传输协议，信息是明文传输，https 则是具有安全性的 ssl 加密传输协议。

   iii. http 和 https 使用的是完全不同的连接方式，用的端口也不一样，前者是 80，后者是 443。

   iv. http 的连接很简单，是无状态的；HTTPS 协议是由 SSL+HTTP 协议构建的可进行加密传输、身份认证的网络协议，比 http 协议安全。

e) HTTP 工作原理

   i. Client 与 Server 建立连接，单击某个超链接，HTTP 的工作开始。

   ii. 连接建立后，Client 发送一个请求给 Server，请求方式的格式为：统一资源标识符（URL）、协议版本号，后边是 MIME 信息包括请求修饰符，Client 信息和可能的内容。

   iii. Server 接到请求后，给予相应的响应信息，其格式为一个状态行，包括信息的协议版本号、一个成功或错误的代码，后边是 MIME 信息包括 Server 信息、实体信息和可能的内容。

   iv. Client 接收 Server 返回的信息通过浏览器显示在用户的显示屏上，然后 Client 和 Server 断开连接。

f) HTTPS 工作原理

   i. Client 使用 HTTPS 的 URL 访问 Web 服务器，要求与 Web 服务器建立 SSL 连接。

   ii. Web 服务器收到客户端请求后，会将网站的证书信息（证书中包含公钥）传送一份给客户端。

   iii. 客户端的浏览器与 Web 服务器开始协商 SSL 连接的安全等级，也就是信息加密的等级。

   iv. 客户端的浏览器根据双方同意的安全等级，建立会话密钥，然后利用网站的公钥将会话密钥加密，并传送给网站。

   v. Web 服务器利用自己的私钥解密出会话密钥。

   vi. Web 服务器利用会话密钥加密与客户端之间的通信。

g) HTTPS 优缺点

   i. 优点

1. 使用 HTTPS 协议可认证用户和服务器，确保数据发送到正确的客户机和服务器；
2. HTTPS 协议是由 SSL+HTTP 协议构建的可进行加密传输、身份认证的网络协议，要比 http 协议安全，可防止数据在传输过程中不被窃取、改变，确保数据的完整性。
3. HTTPS 是现行架构下最安全的解决方案，虽然不是绝对安全，但它大幅增加了中间人攻击的成本。

ii. 缺点：

1. HTTPS 协议握手阶段比较费时，会使页面的加载时间延长近 50%，增加 10%到 20%的耗电；
2. HTTPS 连接缓存不如 HTTP 高效，会增加数据开销和功耗，甚至已有的安全措施也会因此而受到影响；
3. SSL 证书需要钱，功能越强大的证书费用越高，个人网站、小网站没有必要一般不会用。
4. SSL 证书通常需要绑定 IP，不能在同一 IP 上绑定多个域名，IPv4 资源不可能支撑这个消耗。
5. HTTPS 协议的加密范围也比较有限，在黑客攻击、拒绝服务攻击、服务器劫持等方面几乎起不到什么作用。最关键的，SSL 证书的信用链体系并不安全，特别是在某些国家可以控制 CA 根证书的情况下，中间人攻击一样可行。

h) HTTP 切换 HTTPS

i. 可以做兼容

29. 存储引擎

a) MyISAM 存储引擎、innoDB 存储引擎、MEMORY 存储引擎、ARCHIVE 存储引擎
b) 存储引擎其实就是对于数据库文件的一种存取机制，如何实现存储数据，如何为存储的数据建立索引以及如何更新，查询数据等技术实现的方法。
c) MySQL 中的数据用各种不同的技术存储在文件（或内存）中，这些技术中的每一种技术都使用不同的存储机制，索引技巧，锁定水平并且最终提供广泛的不同功能和能力。在 MySQL 中将这些不同的技术及配套的相关功能称为存储引擎。
d) 查看引擎：

i. show engines; // 查看 mysql 所支持的存储引擎，以及从中得到 mysql 默认的存储引擎。
ii. show variables like '% storage_engine' ; // 查看 mysql 默认的存储引擎
iii. show create table tablename ; // 查看具体某一个表所使用的存储引擎，这个默认存储引擎被修改了！
iv. show table status from database where name= "tablename" //准确查看某个数据库中的某一表所使用的存储引擎

e) MyISAM 优缺点：MyISAM 的优势在于占用空间小，处理速度快。缺点是不支持事务的完整性和并发性。（B+）
f) innoDB 优缺点：InnoDB 的优势在于提供了良好的事务处理、崩溃修复能力和并发控制。缺点是读写效率较差，占用的数据空间相对较大。（B+）

g) MEMORY：使用存储在内存中的数据来创建表，而且所有的数据也都存储在内存中。（hash）速度快，也可以（B）生命周期很短，一般只使用一次。

h) ARCHIVE：该存储引擎非常适合存储大量独立的、作为历史记录的数据。区别于 InnoDB 和 MyISAM 这两种引擎，ARCHIVE 提供了压缩功能，拥有高效的插入速度，但是这种引擎不支持索引，所以查询性能较差一些。

30. InnoDB：支持事务处理，支持外键，支持崩溃修复能力和并发控制。如果需要对事务的完整性要求比较高（比如银行），要求实现并发控制（比如售票），那选择 InnoDB 有很大的优势。如果需要频繁的更新、删除操作的数据库，也可以选择 InnoDB，因为支持事务的提交（commit）和回滚（rollback）。

    MyISAM：插入数据快，空间和内存使用比较低。如果表主要是用于插入新记录和读出记录，那么选择 MyISAM 能实现处理高效率。如果应用的完整性、并发性要求比 较低，也可以使用。如果数据表主要用来插入和查询记录，则 MyISAM 引擎能提供较高的处理效率

    MEMORY：所有的数据都在内存中，数据的处理速度快，但是安全性不高。如果需要很快的读写速度，对数据的安全性要求较低，可以选择 MEMOEY。它对表的大小有要求，不能建立太大的表。所以，这类数据库只使用在相对较小的数据库表。如果只是临时存放数据，数据量不大，并且不需要较高的数据安全性，可以选择将数据保存在内存中的 Memory 引擎，MySQL 中使用该引擎作为临时表，存放查询的中间结果

    如果只有 INSERT 和 SELECT 操作，可以选择 Archive，Archive 支持高并发的插入操作，但是本身不是事务安全的。Archive 非常适合存储归档数据，如记录日志信息可以使用 Archiv

    注意，同一个数据库也可以使用多种存储引擎的表。如果一个表要求比较高的事务处理，可以选择 InnoDB。这个数据库中可以将查询要求比较高的表选择 MyISAM 存储。如果该数据库需要一个用于查询的临时表，可以选择 MEMORY 存储引擎。

31. 索引失效

    a) like 是以%开头的查询语句：不在第一个位置可以起作用

    b) 使用多列索引的查询语句：使用第一个字段才能使用索引

    c) 使用 OR 关键字查询语句：查询语句的查询条件中只有 OR 关键字，且 OR 前后的两个条件中列都是索引时，查询中才会使用索引。否则，查询将不使用索引。

    d) 如果列类型是字符串，那一定要在条件中将数据使用引号引用起来,否则不使用索引

1)没有查询条件，或者查询条件没有建立索引
2) 在查询条件上没有使用引导列
3) 查询的数量是大表的大部分，应该是30%以上。
4) 索引本身失效
5) 查询条件使用函数在索引列上，或者对索引列进行运算，运算包括(+，-，*，/，！等) 错误的例子：select * from test where id-1=9; 正确的例子：select * from test where id=10;
6) 对小表查询
7) 提示不使用索引
8) 统计数据不真实
9) CBO计算走索引花费过大的情况。其实也包含了上面的情况，这里指的是表占有的block要比索引小。
10)隐式转换导致索引失效.这一点应当引起重视.也是开发中经常会犯的错误. 由于表的字段tu_mdn定义为varchar2(20),但在查询时把该字段作为number类型以where条件传给Oracle,这样会导致索引失效. 错误的例子：select * from test where tu_mdn=13333333333; 正确的例子：select * from test where tu_mdn='13333333333';
12) 1,<> 2,单独的>,<,(有时会用到，有时不会)
13）like “%_” 百分号在前.
14）表没分析.
15）单独引用复合索引里非第一位置的索引列.
16）字符型字段为数字时在where条件里不添加引号.
17）对索引列进行运算.需要建立函数索引.
18）not in ,not exist.
19）当变量采用的是times变量，而表的字段采用的是date变量时.或相反情况。
20）B-tree索引 is null不会走,is not null会走,位图索引 is null,is not null 都会走
21）联合索引 is not null 只要在建立的索引列（不分先后）都会走, in null时 必须要和建立索引第一列一起使用,当建立索引第一位置条件是is null 时,其他建立索引的列可以是is null（但必须在所有列 都满足is null的时候）,或者=一个值； 当建立索引的第一位置是=一个值时,其他索引列可以是任何情况（包括is null =一个值）,以上两种情况索引都会走。其他情况不会走。

e)

32. 索引类型
    a) 有序数组：
        i. 这样，我们在查找数据的时候，就可以通过 id 这个列，在数据表中进行二分查找，二分查找的时间复杂度为 O(logn)，这是非常快的
        ii. 另外由于数据是有序存放的，所以支持范围查找，只要找到起始值然后往后扫描判断，就可以检索出范围内的值
        iii. 但是使用有序数组的情况，如果是插入或者删除数据，就会非常的麻烦。可以想象，插入数据需要将后半部数据往后挪动一个位置，删除数据需要将后半部数据往前挪动一个位置，这样的代价是非常大的，所以这也是没有使用有序数组来组织索引的原因
    b) Hash Table
        i. 理论上哈希表的时间复杂度是 O(1)，所以查找数据会非常快
        ii. 但是哈希表是无序的，所以我们无法利用索引进行范围查找，只能利用索引来进行等值查询
    c) BST
        i. 查找的时间复杂度为 O(logn)，所以查找速度也非常快
        ii. 是有序的，所以也支持范围查找
        iii. 二叉搜索树的节点在文件中是随机存放的，所以可能读取一个节点就需要一个磁盘 IO，恰恰二叉搜索树都会比较高，如一棵一百万个元素的平衡二叉树就有十几层高度了，也就是大部分情况下检索一次数据就需要十几次磁盘 IO，这个代价太高了，所以一般二叉搜索树也不会被用来作索引
        iv. 可以使用 B 树
    d) B-
        i. 每个节点都是一个页，可以存放多个数据节点，每页中的节点都是有序的
        ii. B-Tree 的查找过程是，因为每个页中的节点都是有序的，所以在每个页中都可以使用二分查找，而 B-Tree 的高度又会很低，所以查找效率会很快

e) B+

i. 非节点只存放索引 key，不存放数据，数据只存在于叶子节点

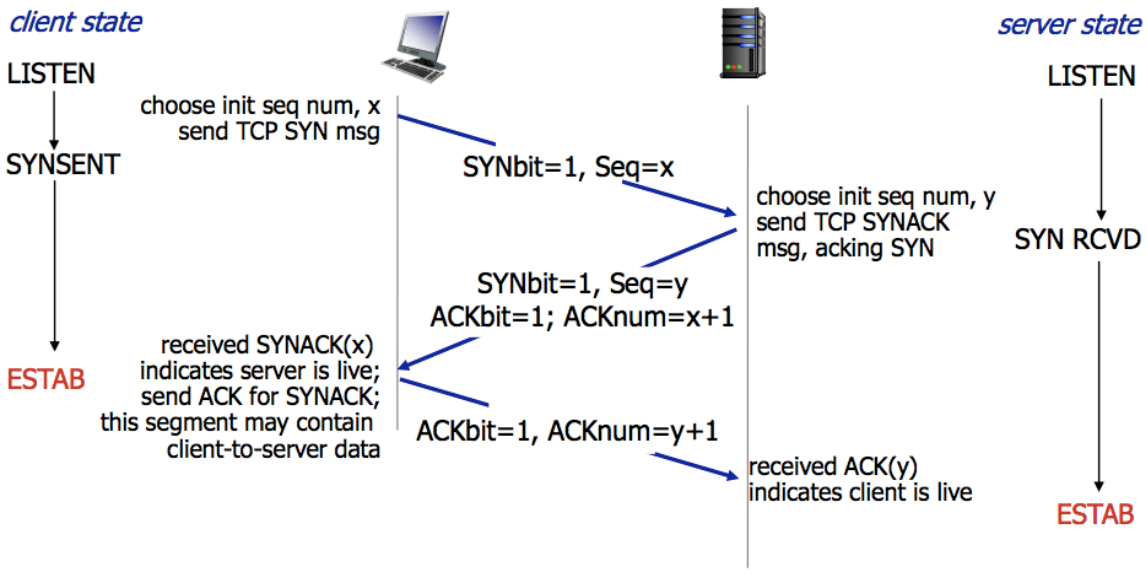
ii. 叶子节点页之间使用链表连接

iii. B+Tree 的磁盘读写代价更低：由于非叶子节点只存放索引不存放数据，所以每个节点可以存放更多的索引，一次读取查找的关键字更多，树的高度更低

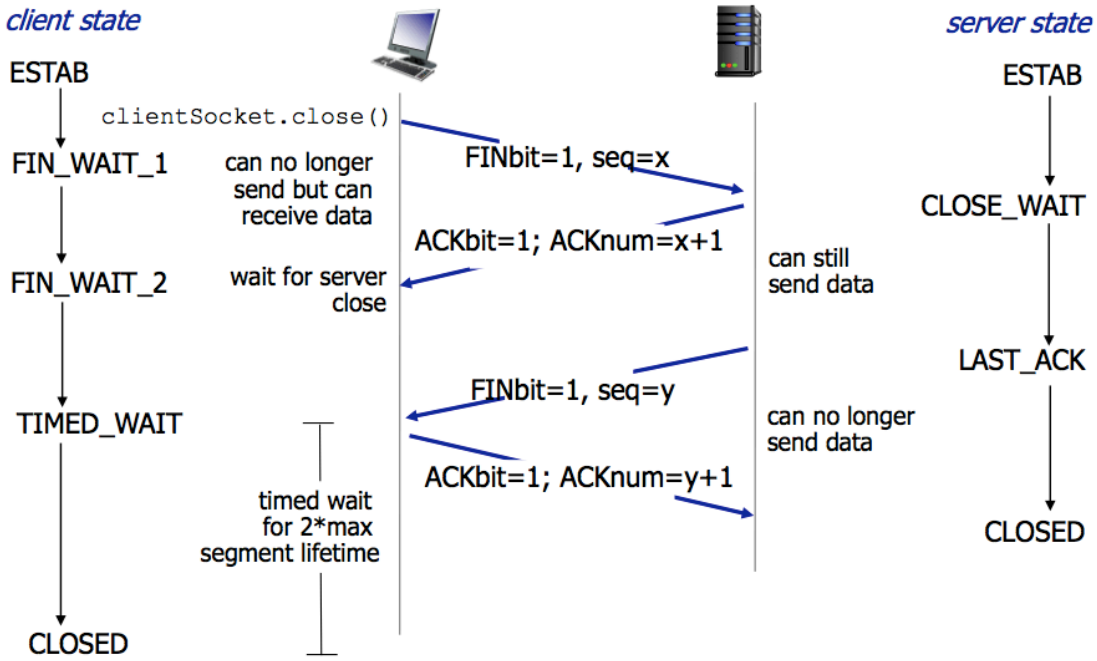
iv. B+Tree 的查询效率更加稳定，因为只有叶子节点存在数据，所以每次查询的路径长度都是相同的

v. B+Tree 更适合范围查询，因为 B-Tree 的非叶子节点存放数据，所以需要使用中序遍历来查询，而 B+Tree 只有叶子节点有数据，叶子节点之间使用链表连接，所以只要顺序扫描进行，更加方便

33. (1)对于主健/unique constraint ， oracle/sql server/mysql 等都会自动建立唯一索引；
(2)主键不一定只包含一个字段，所以如果你在主键的其中一个字段建唯一索引还是必要的；
(3)主健可作外健，唯一索引不可；
(4)主健不可为空，唯一索引可；
(5)主健也可是多个字段的组合；
(6)主键与唯一索引不同的是：
a.有 not null 属性；
b.每个表只能有一个。



34.



35.

36. ACID

a) ACID，指数据库事务正确执行的四个基本要素的缩写。包含：原子性（Atomicity）、一致性（Consistency）、隔离性（Isolation）、持久性（Durability）。一个支持事务（Transaction）的数据库，必须要具有这四种特性，否则在事务过程（Transaction processing）当中无法保证数据的正确性，交易过程极可能达不到交易方的要求。

b) 原子性：整个事务中的所有操作，要么全部完成，要么全部不完成，不可能停滞在中间某个环节。事务在执行过程中发生错误，会被回滚（Rollback）到事务开始前的状态，就像这个事务从来没有执行过一样。

c) 一致性：一个事务可以封装状态改变（除非它是一个只读的）。事务必须始终保持系统处于一致的状态，不管在任何给定的时间并发事务有多少。

d) 隔离性：如果有两个事务，运行在相同的时间内，执行相同的功能，事务的隔离性将确保每一事务在系统中认为只有该事务在使用系统。

e) 持久性：在事务完成以后，该事务对数据库所作的更改便持久的保存在数据库之中，并不会被回滚。

37. 负载均衡

a) 四层：主要通过报文中的目标地址和端口，再加上负载均衡设备设置的服务器选择方式，决定最终选择的内部服务器。

b) 七层：主要通过报文中的真正有意义的应用层内容，再加上负载均衡设备设置的服务器选择方式，决定最终选择的内部服务器。

c) 这种方式可以对客户端的请求和服务器的响应进行任意意义上的修改，极大的提升了应用系统在网络层的灵活性

d) 四层模式下这些 SYN 攻击都会被转发到后端的服务器上；而七层模式下这些 SYN 攻击自然在负载均衡设备上就截止，不会影响后台服务器的正常运营。

e) 四层 TCP，七层 HTTP

f) 是否真的必要，七层应用的确可以提高流量智能化，*同时必不可免的带来设备配置复杂，负载均衡压力增高以及故障排查上的复杂性等问题*。在设计系统时需要考虑四层七层同时应用的混杂情况。

g) 是否真的可以提高安全性。例如 SYN Flood 攻击，七层模式的确将这些流量从服务器屏蔽，但*负载均衡设备本身要有强大的抗 DDoS 能力*，否则即使服务器正常而作为中枢调度的负载均衡设备故障也会导致整个应用的崩溃。

h) 是否有足够的灵活度。七层应用的优势是可以让整个应用的流量智能化，但是负载均衡设备需要提供完善的七层功能，满足客户根据不同情况的基于应用的调度。*最简单的一个考核就是能否取代后台 Nginx 或者 Apache 等服务器上的调度功能*。能够提供一个七层应用开发接口的负载均衡设备，可以让客户根据需求任意设定功能，才真正有可能提供强大的灵活性和智能性。

i)

三、负载均衡的算法?

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。存在慢的提供者累积请求问题，比如：第二台机器很慢，但没挂，当请求调到第二台时就卡在那，久而久之，所有请求都卡在调到第二台上。
- 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

- 普通哈希
- 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。

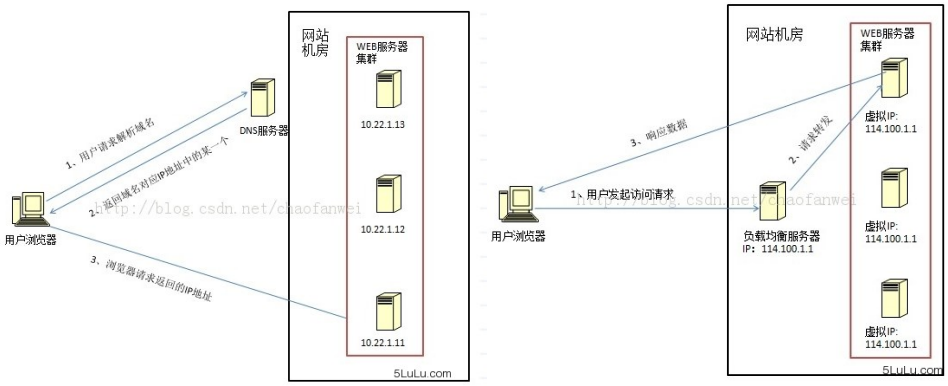
5. IP地址散列

- 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。

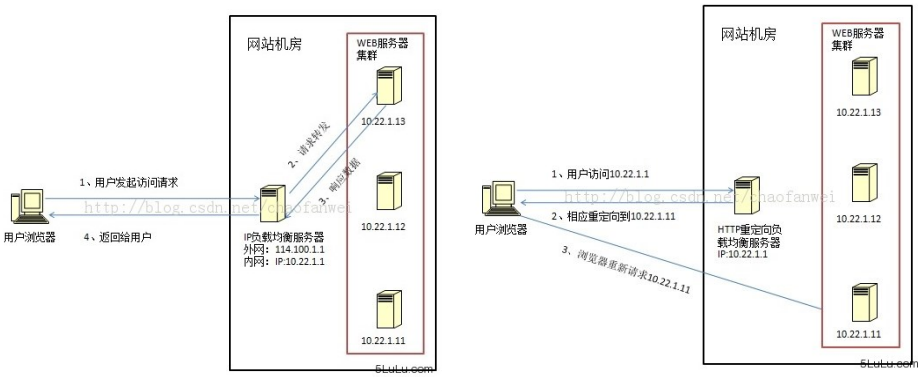
6.URL散列

- 通过管理客户端请求URL信息的散列，将发送至相同URL的请求转发至同一服务器的算法。

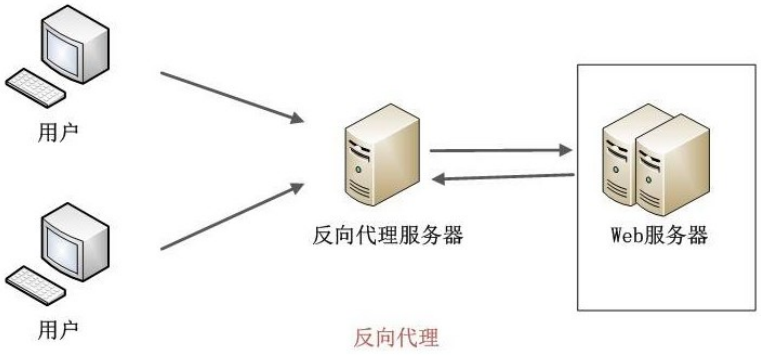
j)



k)



l)



38. MySQL 优化

a) 选取最适用的字段属性

MySQL可以很好的支持大数据量的存取，但是一般说来，数据库中的表越小，在它上面执行的查询也就会越快。因此，在创建表的时候，为了获得更好的性能，我们可以将表中字段的宽度设得尽可能小。

例如，在定义邮政编码这个字段时，如果将其设置为CHAR(255),显然给数据库增加了不必要的空间，甚至使用VARCHAR这种类型也是多余的，因为CHAR(6)就可以很好的完成任务了。同样的，如果可以的话，我们应该使用MEDIUMINT而不是BIGIN来定义整型字段。

另外一个提高效率的方法是在可能的情况下，应该尽量把字段设置为NOTNULL，这样在将来执行查询的时候，数据库不用去比较NULL值。
对于某些文本字段，例如“省份”或者“性别”，我们可以将它们定义为ENUM类型。因为在MySQL中，ENUM类型被当作数值型数据来处理，而数值型数据被处理起来的速度要比文本类型快得多。这样，我们又可以提高数据库的性能。

b) 使用连接（JOIN）来代替子查询(Sub-Queries)

MySQL从4.1开始支持SQL的子查询。这个技术可以使用SELECT语句来创建一个单列的查询结果，然后把这个结果作为过滤条件用在另一个查询中。例如，我们要将客户基本信息表中没有任何订单的客户删除掉，就可以利用子查询先从销售信息表中将所有发出订单的客户ID取出来，然后将结果传递给主查询，如下所示:

c) 使用联合(UNION)来代替手动创建的临时表

MySQL从4.0的版本开始支持union查询，它可以把需要使用临时表的两条或更多的select查询合并的一个查询中。在客户端的查询会话结束的时候，临时表会被自动删除，从而保证数据库整齐、高效。使用union来创建查询的时候，我们只需要用UNION作为关键字把多个select语句连接起来就可以了，要注意的是所有select语句中的字段数目要想同。下面的例子就演示了一个使用UNION的查询。

d) 事务

尽管我们可以使用子查询（Sub-Queries）、连接（JOIN）和联合（UNION）来创建各种各样的查询，但不是所有的数据库操作都可以只用一条或少数几条SQL语句就可以完成的。更多的时候是需要用到一系列的语句来完成某种工作。但是在这种情况下，当这个语句块中的某一条语句运行出错的时候，整个语句块的操作就会变得不确定起来。设想一下，要把某个数据同时插入两个相关联的表中，可能会出现这样的情况：第一个表中成功更新后，数据库突然出现意外状况，造成第二个表中的操作没有完成，这样，就会造成数据的不完整，甚至会破坏数据库中的数据。要避免这种情况，就应该使用事务，它的作用是：要么语句块中每条语句都操作成功，要么都失败。换句话说，就是可以保持数据库中数据的一致性和完整性。事物以BEGIN关键字开始，COMMIT关键字结束。在这之间的一条SQL操作失败，那么，ROLLBACK命令就可以把数据库恢复到BEGIN开始之前的状态。

BEGIN; INSERTINTOsalesinfoSETCustomerID=14;UPDATEinventorySETQuantity=11WHEREitem='book';COMMIT;

事务的另一个重要作用是当多个用户同时使用相同的数据源时，它可以利用锁定数据库的方法来为用户提供一种安全的访问方式，这样可以保证用户的操作不被其它的用户所干扰。

e) 锁定表

尽管事务是维护数据库完整性的一个非常好的方法，但却因为它的独占性，有时会影响数据库的性能，尤其是在很大的应用系统中。由于在事务执行的过程中，数据库将会被锁定，因此其它的用户请求只能暂时等待直到该事务结束。如果一个数据库系统只有少数几个用户来使用，事务造成的影响不会成为一个太大的问题；但假设有成千上万的用户同时访问一个数据库系统，例如访问一个电子商务网站，就会产生比较严重的响应延迟。

f) 使用外键

锁定表的方法可以维护数据的完整性，但是它却不能保证数据的关联性。这个时候我们就可以使用外键。

g) 使用索引

索引是提高数据库性能的常用方法，它可以令数据库服务器以比没有索引快得多的速度检索特定的行，尤其是在查询语句当中包含有MAX(),MIN()和ORDERBY这些命令的时候，性能提高更为明显。

那该对哪些字段建立索引呢?

一般说来，索引应建立在那些将用于JOIN,WHERE判断和ORDERBY排序的字段上。尽量不要对数据库中某个含有大量重复的值的字段建立索引。对于一个ENUM类型的字段来说，出现大量重复值是很有可能的情况

h) 优化的查询语句

绝大多数情况下，使用索引可以提高查询的速度，但如果SQL语句使用不恰当的话，索引将无法发挥它应有的作用。

下面是应该注意的几个方面。

- 首先，最好是在相同类型的字段间进行比较的操作。

> 在MySQL3.23版之前，这甚至是一个必须的条件。例如不能将一个建有索引的INT字段和BIGINT字段进行比较；但是作为特殊的情况，在CHAR类型的字段和VARCHAR类型字段的字段大小相同的时候，可以将它们进行比较。

- 其次，在建有索引的字段上尽量不要使用函数进行操作。

> 例如，在一个DATE类型的字段上使用YEAE()函数时，将会使索引不能发挥应有的作用。所以，下面的两个查询虽然返回的结果一样，但后者要比前者快得多。

39. 八种方法

1、创建索引

对于查询占主要的应用来说，索引显得尤为重要。很多时候性能问题很简单的就是因为我们忘了添加索引而造成的，或者说没有添加更为有效的索引导致。如果不加索引的话，那么查找任何哪怕只是一条特定的数据都会进行一次全表扫描，如果一张表的数据量很大而符合条件的结果又很少，那么不加索引会引起致命的性能下降。但是也不是什么情况都非得建索引不可，比如性别可能就只有两个值，建索引不仅没什么优势，还会影响到更新速度，这被称为过度索引。

2、复合索引

比如有一条语句是这样的：select * from users where area='beijing' and age=22;

如果我们是在area和age上分别创建单个索引的话，由于mysql查询每次只能使用一个索引，所以虽然这样已经相对不做索引时全表扫描提高了很多效率，但是如果在area、age两列上创建复合索引的话将带来更高的效率。如果我们创建了(area, age, salary)的复合索引，那么其实相当于创建了(area,age,salary)、(area,age)、(area)三个索引，这被称为最佳左前缀特性。因此我们在创建复合索引时应该将最常用作限制条件的列放在最左边，依次递减。

3、索引不会包含有NULL值的列

只要列中包含有NULL值都将不会被包含在索引中，复合索引中只要有一列含有NULL值，那么这一列对于此复合索引就是无效的。所以我们在数据库设计时不要让字段的默认值为NULL。

4、使用短索引

对串列进行索引，如果可能应该指定一个前缀长度。例如，如果有一个CHAR(255)的 列，如果在前10 个或20 个字符内，多数值是惟一的，那么就不要对整个列进行索引。短索引不仅可以提高查询速度而且可以节省磁盘空间和I/O操作。

5、排序的索引问题

mysql查询只使用一个索引，因此如果where子句中已经使用了索引的话，那么order by中的列是不会使用索引的。因此数据库默认排序可以符合要求的情况下不要使用排序操作；尽量不要包含多个列的排序，如果需要最好给这些列创建复合索引。

6、like语句操作

一般情况下不鼓励使用like操作，如果非使用不可，如何使用也是一个问题。like “%aaa%” 不会使用索引而like “aaa%”可以使用索引。

7、不要在列上进行运算

select * from users where YEAR(adddate)<2007;

将在每个行上进行运算，这将导致索引失效而进行全表扫描，因此我们可以改成

select * from users where adddate< ‘2007-01-01’;

8、不使用NOT IN和<>操作

NOT IN和<>操作都不会使用索引将进行全表扫描。NOT IN可以NOT EXISTS代替，id<>3则可使用id>3 or id<3来代替。

40. 高并发网络库

进程 线程 协程 异步

并发编程（不是并行）目前有四种方式：多进程、多线程、协程和异步。

- 多进程编程在python中有类似C的os.fork,更高层封装的有multiprocessing标准库
- 多线程编程python中有Thread和threading
- 异步编程在linux下主+要有三种实现select，poll，epoll
- 协程在python中通常会说到yield，关于协程的库主要有greenlet,stackless,gevent,eventlet等实现。

进程

- 不共享任何状态
- 调度由操作系统完成
- 有独立的内存空间（上下文切换的时候需要保存栈、cpu寄存器、虚拟内存、以及打开的相关句柄等信息，开销大）
- 通讯主要通过信号传递的方式来实现（实现方式有多种，信号量、管道、事件等，通讯都需要过内核，效率低）

线程

- 共享变量（解决了通讯麻烦的问题，但是对于变量的访问需要加锁）
- 调度由操作系统完成（由于共享内存，上下文切换变得高效）
- 一个进程可以有多个线程，每个线程会共享父进程的资源（创建线程开销占用比进程小很多，可创建的数量也会很多）
- 通讯除了可使用进程间通讯的方式，还可以通过共享内存的方式进行通信（通过共享内存通信比通过内核要快很多）

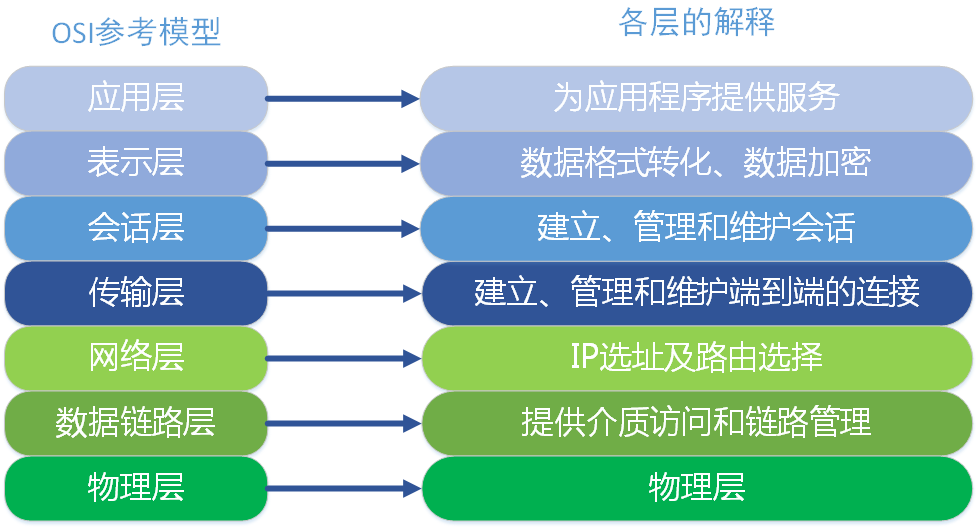
协程

- 调度完全由用户控制
- 一个线程（进程）可以有多个协程
- 每个线程（进程）循环按照指定的任务清单顺序完成不同的任务（当任务被堵塞时，执行下一个任务；当恢复时，再回来执行这个任务；任务间切换只需要保存任务的上下文，没有内核的开销，可以不加锁的访问全局变量）
- 协程需要保证是非堵塞的且没有相互依赖
- 协程基本上不能同步通讯，多采用异步的消息通讯，效率比较高

总结

- 进程拥有自己独立的堆和栈，既不共享堆，亦不共享栈，进程由操作系统调度
- 线程拥有自己独立的栈和共享的堆，共享堆，不共享栈，线程亦由操作系统调度(标准线程是的)
- 协程和线程一样共享堆，不共享栈，协程由程序员在协程的代码里显示调度

41.



发送从应用，接收从物理