面试高频考点整理

1. 项目相关

1. **介绍一下你简历上写的项目？自己主要做了什么？**
2. **你觉得项目里给你最大的挑战是什么？遇到了什么问题？如何解决的？从中学到了什么？**

> 项目里面会不断出现各种问题，比如数据量过大造成的内存溢出问题，如何让程序运行效率更高，如何证明我们的算法比别人的算法效率高，如何找到新的观点来支撑我们现有的理论，如何向导师和师兄进行沟通完成接下来的工作。

3. **项目的架构图能画一下不？**
4. **觉得项目有哪些地方可以改进完善？（比如：可以加一个 redis 缓存把热点数据缓存起来）**
5. **有没有遇到过内存泄漏的场景?**

---

2. 操作系统

1. **进程和线程的区别?**

> 1、**进程是资源分配的最小单位，线程是任务执行的最小单位。（关键句）**
> 2、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比进程要小得多，同时创建一个线程的开销也比进程要小很多。
> 3、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。

2. **进程的调度算法有哪些？（主要）**

> 1. 先来先服务，从后备队列选择最先进入的作业，调入内存。
> 2. 时间片轮转法，遵循先来先服务原则，但是一次只能运行一个固定的时间片。
> 3. 短作业优先，从后备队列选择估计运行时间最短的作业，调入内存。平均等待时间、平均周转时间最少。
> 4. 多级反馈队列调度算法
> 5. 优先级调度，分为非剥夺式和剥夺式。

3. **5种IO模型?**

> **关注消息通信机制：**
>
> **同步：调用一个功能，在功能结果没有返回之前，一直等待结果返回。**
>
> **异步：调用一个功能，调用立刻返回，但调用者不能立刻得到结果。调用者可以继续后续的操作，其结果一般通过状态，回调函数来通知调用者。**

等待调用结果时的状态：

阻塞：调用一个函数，当调用结果返回之前，当前线程会被挂起，只有得到结果之后才会返回。

非阻塞：调用一个函数，不能立刻得到结果之前，调用不能阻塞当前线程。

一个输入操作通常包括两个阶段：

- **等待数据准备好**
- **从内核向进程复制数据**

对于一个套接字上的输入操作，第一步通常涉及等待数据从网络中到达。当所等待数据到达时，它被复制到内核中的某个缓冲区。第二步就是把数据从内核缓冲区复制到应用进程缓冲区。

阻塞IO模型：应用进程被阻塞，直到数据从内核缓冲区复制到应用进程缓冲区中才返回。

非阻塞IO模型：进程发起IO系统调用后，内核返回一个错误码而不会被阻塞；应用进程可以继续执行，但是需要不断的执行系统调用来获知 I/O 是否完成。如果内核缓冲区有数据，内核就会把数据返回进程。

IO复用模型：使用 select 或者 poll 等待数据，可以等待多个套接字中的任何一个变为可读。这一过程会被阻塞，当某一个套接字可读时返回，之后把数据从内核复制到进程中。（在多路复用IO模型中，会有一个线程不断去轮询多个socket的状态，只有当socket真正有读写事件时，才真正调用实际的IO读写操作。因为在多路复用IO模型中，只需要使用一个线程就可以管理多个socket，并且只有在真正有socket读写事件进行时，才会使用IO资源，所以它大大减少了资源占用。）

信号驱动IO模型：当进程发起一个IO操作，会向内核注册一个信号处理函数，然后进程返回不阻塞；当内核数据就绪时会发送一个信号给进程，进程便在信号处理函数中调用IO读取数据。

异步IO模型：当进程发起一个IO操作，进程返回不阻塞，但也不能返回结果；内核把整个IO处理完后，会通知进程结果。如果IO操作成功则进程直接获取到数据。

4. **select、poll和epoll的区别？epoll的底层使用的数据结构。**

select ，poll和epoll允许应用程序监视一组文件描述符，等待一个或者多个描述符成为就绪状态，从而完成 I/O 操作。

select 和 poll 的功能基本相同，不过在一些实现细节上有所不同。

- select 的描述符类型使用数组实现，FD_SETSIZE 大小默认为 1024，因此默认只能监听少于 1024 个描述符。如果要监听更多描述符的话，需要修改 FD_SETSIZE 之后重新编译；而 poll 没有描述符数量的限制，poll 中的描述符是 pollfd 类型的数组。；
- poll 提供了更多的事件类型，并且对描述符的重复利用上比 select 高。
- 如果一个线程对某个描述符调用了 select 或者 poll，另一个线程关闭了该描述符，会导致调用结果不确定。

select 和 poll 速度都比较慢，每次调用都需要将全部描述符从应用进程缓冲区复制到内核缓冲区。

当某个进程调用epoll_create()方法时，内核会创建一个eventpoll对象。

创建epoll对象后，可以用epoll_ctl()向内核注册新的描述符或者是改变某个文件描述符的状态。已注册的描述符在内核中会被维护在一棵红黑树上，通过回调函数内核会将 I/O 准备好的描述符加入到一个链表中管理，进程调用 epoll_wait() 便可以得到事件完成的描述符。

就绪列表：epoll使用双向链表来实现就绪队列，是一种能够快速插入和删除的数据结构。

索引结构：epoll使用红黑树去监听并维护所有文件描述符。

epoll 的描述符事件有两种触发模式：LT（水平触发）和 ET（边沿触发）。

当 epoll_wait() 检测到描述符事件到达时，将此事件通知进程，进程可以不立即处理该事件，下次调用 epoll_wait() 会再次通知进程。

和 LT 模式不同的是，通知之后进程必须立即处理事件，下次再调用 epoll_wait() 时不会再得到事件到达的通知。

边沿触发仅触发一次，水平触发会一直触发。

5. **进程的通信方式有哪些？线程呢?**

> **进程间的通信方式**：
>
> 1. **管道/匿名管道(Pipes)**：用于具有亲缘关系的父子进程间或者兄弟进程之间的通信。**所谓的管道，就是内核里面的一串缓存。**从管道的一段写入的数据，实际上是缓存在内核中的，另一端读取，也就是从内核中读取这段数据。另外，管道传输的数据是无格式的流且大小受限。
> 2. **有名管道(Names Pipes)**：匿名管道由于没有名字，只能用于亲缘关系的进程间通信。为了克服这个缺点，提出了有名管道。有名管道严格遵循**先进先出(first in first out)**。有名管道以磁盘文件的方式存在，可以实现本机任意两个进程通信。
> 3. **消息队列(Message Queuing)**：**消息队列是保存在内核中的消息链表，具有特定的格式，存放在内存中并由消息队列标识符标识。管道和消息队列的通信数据都是先进先出的原则。**与管道（无名管道：只存在于内存中的文件；命名管道：存在于实际的磁盘介质或者文件系统）不同的是消息队列存放在内核中，只有在内核重启(即，操作系统重启)或者显示地删除一个消息队列时，该消息队列才会被真正的删除。消息队列可以实现消息的随机查询，消息不一定要以先进先出的次序读取，也可以按消息的类型读取比 FIFO 更有优势。**消息队列克服了信号承载信息量少，管道只能承载无格式字节流以及缓冲区大小受限等缺。**
> 4. **信号(Signal)**：信号是一种比较复杂的通信方式，用于通知接收进程某个事件已经发生；（对于异常情况下的工作模式，就需要用「信号」的方式来通知进程，信号事件的来源主要有硬件来源（如键盘 Cltr+C ）和软件来源（如 kill 命令）。比如，Ctrl+C 产生 `SIGINT` 信号，表示终止该进程，Ctrl+Z 产生 `SIGTSTP` 信号，表示停止该进程，但还未结束）
> 5. **信号量(Semaphores)**：信号量是一个计数器，用于多进程对共享数据的访问，信号量的意图在于进程间同步。这种通信方式主要用于解决与同步相关的问题并避免竞争条件。（信号量其实是一个整型的计数器，主要用于实现进程间的互斥与同步，而不是用于缓存进程间通信的数据）
> 6. **共享内存(Shared memory)**：使得多个进程可以访问同一块内存空间，不同进程可以及时看到对方进程中对共享内存中数据的更新。这种方式需要依靠某种同步操作，如互斥锁和信号量等。可以说这是最有用的进程间通信方式。（共享内存的机制，就是拿出一块虚拟地址空间来，映射到相同的物理内存中）
> 7. **套接字(Sockets)**：此方法主要用于在客户端和服务器之间通过网络进行通信。套接字是支持 TCP/IP 的网络通信的基本操作单元，可以看做是不同主机之间的进程进行双向通信的端点，简单的说就是通信的两方的一种约定，用套接字中的相关函数来完成通信过程。
>
> ```
> int socket(int domain, int type, int protocal)
> ```

> **线程间的通信方式**：
>
> 1. **互斥量(Mutex)**：采用互斥对象机制，只有拥有互斥对象的线程才有访问公共资源的权限。比如 Java 中的 synchronized 关键词和各种 Lock 都是这种机制。
> 2. **信号量(Semphares)**: 它允许同一时刻多个线程访问同一资源，但是需要控制同一时刻访问此资源的最大线程数量。
> 3. **事件(Event)**: Wait/Notify：通过通知操作的方式来保持多线程同步，还可以方便的实现多线程优先级的比较操作。

6. **fork函数的作用?**

> 在Linux中fork函数是非常重要的函数，它的作用是从已经存在的进程中创建一个子进程，而原进程称为父进程。
>
> 调用fork(),当控制转移到内核中的fork代码后，内核开始做：
> 1.分配新的内存块和内核数据结构给子进程。
> 2.将父进程部分数据结构内容拷贝至子进程。
> 3.将子进程添加到系统进程列表。
> 4.fork返回，开始调度。

> 特点：
>
> - 调用一次，返回两次

- 并发执行
- 相同但是独立的地址空间
- fork的返回值 : fock函数调用一次却返回两次；向父进程返回子进程的ID，向子进程中返回0，
  1. fork的子进程返回为0；
  2. 父进程返回的是子进程的pid。
- fokr调用失败的原因
  1. 系统中有太多进程。
  2. 实际用户的进程数超过限制。

7. **协程的概念?**

> 协程是一种用户态的轻量级线程，协程的调度完全由用户控制。协程拥有自己的寄存器上下文和栈。协程调度切换时，将寄存器上下文和栈保存到其他地方，在切回来的时候，恢复先前保存的寄存器上下文和栈，直接操作栈则基本没有内核切换的开销，可以不加锁的访问全局变量，所以上下文的切换非常快。
>
> 对操作系统而言，线程是最小的执行单元，进程是最小的资源管理单元。无论是进程还是线程，都是由操作系统所管理的。
>
> 协程不是被操作系统内核所管理的，而是完全由程序所控制，也就是在用户态执行。这样带来的好处是性能大幅度的提升，因为不会像线程切换那样消耗资源。
>
> - 协程既不是进程也不是线程，协程仅仅是一个特殊的函数，协程它进程和进程不是一个维度的。
> - 一个进程可以包含多个线程，一个线程可以包含多个协程。
> - 一个线程内的多个协程虽然可以切换，但是多个协程是串行执行的，只能在一个线程内运行，没法利用CPU多核能力。
> - 协程与进程一样，切换是存在上下文切换问题的。

8. **linux进程和线程?**

> 进程通过fork()创建
>
> 线程通过pthread_create()函数创建

9. **僵尸进程产生的原因?**

> 僵尸进程是指它的父进程没有等待(调用wait/waitpid)。
>
> **如果子进程先结束而父进程后结束，即子进程结束后，父进程还在继续运行但是并未调用wait/waitpid那子进程就会成为僵尸进程。**
>
> 但如果子进程后结束，即父进程先结束了，但没有调用wait/waitpid来等待子进程的结束，此时子进程还在运行，父进程已经结束。那么并不会产生僵尸进程。应为每个进程结束时，系统都会扫描当前系统中运行的所有进程，看看有没有哪个进程时刚刚结束的这个进程的子进程，如果有，就有init来接管它，成为它的父进程。
>
> 进程设置僵尸状态的目的是维护子进程的信息，以便父进程在以后某个时间获取。要在当前进程中生成一个子进程，一般需要调用fork这个系统调用，fork这个函数的特别之处在于一次调用，两次返回，一次返回到父进程中，一次返回到子进程中，可以通过返回值来判断其返回点。如果子进程先于父进程退出， 同时父进程又没有调用wait/waitpid，则该子进程将成为僵尸进程。
>
> 在每个进程退出的时候，内核释放该进程所有的资源，包括打开的文件，占用的内存。但是仍然保留了一些信息（如进程号pid 退出状态 运行时间等）。这些保留的信息直到进程通过调用wait/waitpid时才会释放。这样就导致了一个问题，如果没有调用wait/waitpid的话，那么保留的信息就不会释放。比如进程号就会被一直占用了。但系统所能使用的进程号的有限的，如果产生大量的僵尸进程，将导致系统没有可用的进程号而导致系统不能创建进程。所以我们应该避免僵尸进程。
>
> **如果进程不调用wait / waitpid的话， 那么保留的那段信息就不会释放，其进程号就会一直被占用，但是系统所能使用的进程号是有限的，如果大量的产生僵死进程，将因为没有可用的进程号而导致系统不能产生新的进程. 此即为僵尸进程的危害，应当避免。**

10. **孤儿进程产生的原因?**

> **孤儿进程：一个父进程退出，而它的一个或多个子进程还在运行，那么那些子进程将成为孤儿进程。孤儿进程将被init进程(进程号为1)所收养，并由init进程对它们完成状态收集工作。**
>
> **孤儿进程是没有父进程的进程，孤儿进程这个重任就落到了init进程身上，因此孤儿进程并不会有什么危害。**

11. **讲一下虚拟内存。虚拟内存和物理内存的关系是什么?**

> 虚拟内存使得应用程序认为它拥有一个**连续的地址空间，而实际上，它通常是被分隔成多个物理内存碎片，还有一部分存储在外部磁盘存储器上，在需要时进行数据交换。**
>
> 虚拟内存可以让程序可以拥有超过系统物理内存大小的可用内存空间。虚拟内存让每个进程拥有一片连续完整的内存空间。

> 局部性原理表现在以下两个方面：
>
> 1. **时间局部性**： 如果程序中的某条指令一旦执行，不久以后该指令可能再次执行；如果某数据被访问过，不久以后该数据可能再次被访问。
> 2. **空间局部性**： 一旦程序访问了某个存储单元，在不久之后，其附近的存储单元也将被访问。

> 操作系统将内存抽象成地址空间。每个程序拥有自己的地址空间，这个地址空间被分割成多个块，每一块称为一页。这些页被映射到物理内存，但不需要映射到连续的物理内存，也不需要所有页都必须在物理内存中。当程序引用到不在物理内存中的页时，会将缺失的部分从磁盘装入物理内存。

> 页面置换算法：
>
> **OPT 页面置换算法（最佳页面置换算法）**： 所选择的被换出的页面将是最长时间内不再被访问，通常可以保证获得最低的缺页率。
>
> **FIFO（First In First Out） 页面置换算法（先进先出页面置换算法）**：总是淘汰最先进入内存的页面，即选择在内存中驻留时间最久的页面进行淘汰。
>
> **LRU （Least Currently Used）页面置换算法（最近最久未使用页面置换算法）**： 将最近最久未使用的页面换出。需要在内存中维护一个所有页面的链表。当一个页面被访问时，将这个页面移到链表表头。这样就能保证链表表尾的页面是最近最久未访问的。
>
> **LFU （Least Frequently Used）页面置换算法（最少使用页面置换算法）**： 该置换算法选择在之前时期使用最少的页面作为淘汰页。被使用次数最少的数据优先淘汰。每个数据块都有一个引用计数，按照引用计数排序，具有相同计数的数据块按时间排序。

12. **分段和分页讲一下？以及对应的场景?**

> **操作系统的内存管理机制了解吗？内存管理有哪几种方式?**
>
> 1. **块式管理**： 将内存分为几个固定大小的块，每个块中只包含一个进程。
> 2. **页式管理**： 把主存分为大小相等且固定的一页一页的形式，页较小，相对相比于块式管理的划分力度更大，提高了内存利用率，减少了碎片。页式管理通过页表对应逻辑地址和物理地址。
> 3. **段式管理**： 页式管理虽然提高了内存利用率，但是页式管理其中的页实际并无任何实际意义。 段式管理把主存分为一段段的，最重要的是段是有实际意义的，每个段定义了一组逻辑信息。 段式管理通过段表对应逻辑地址和物理地址。例如,有主程序段 MAIN、子程序段 X、数据段 D 及栈段 S 等。 段式管理通过段表对应逻辑地址和物理地址。
> 4. **段页式管理**：段页式管理机制结合了段式管理和页式管理的优点。段页式管理机制就是把主存先分成若干段，每个段又分成若干页。

> **分段和分页**：
>
> 1. 共同点
>   - 分页机制和分段机制都是为了提高内存利用率，较少内存碎片。
>   - 页和段都是离散存储的，所以两者都是离散分配内存的方式。但是，每个页和段中的内存是连续的。
> 2. 区别
>   - 页的大小是固定的，由操作系统决定；而段的大小不固定，取决于我们当前运行的程序。

- 分页仅仅是为了满足操作系统内存管理的需求，而段是逻辑信息的单位，在程序中可以体现为代码段，数据段，能够更好满足用户的需要。

13. **讲一下用户态和内核态？所有的系统调用都会进入到内核态吗？**

> 操作系统（Operating System，简称 OS）是管理计算机硬件与软件资源的程序。
>
> 根据进程访问资源的特点，我们可以把进程在系统上的运行分为两个级别：
>
> 1. 用户态(user mode) : 用户态运行的进程或可以直接读取用户程序的数据。
> 2. 内核态(kernel mode):可以简单的理解系统态运行的进程或程序几乎可以访问计算机的任何资源，不受限制。
>
> 运行的程序基本都是运行在用户态。如果我们调用操作系统提供的内核态级别的子功能那就需要系统调用了。
>
> 系统调用：与系统态级别的资源有关的操作（如文件管理、进程控制、内存管理等），都必须通过系统调用方式向操作系统提出服务请求，并由操作系统代为完成。
>
> 系统调用是操作系统为应用程序提供能够访问到内核态的资源的接口。

补充：

> 用户态切换到内核态的几种方式
>
> - 系统调用: 系统调用是用户态主动要求切换到内核态的一种方式， 用户应用程序通过操作系统调用内核为上层应用程序开放的接口来执行程序。
> - 异常: 当cpu在执行用户态的应用程序时，发生了某些不可知的异常。 于是当前用户态的应用进程切换到处理此异常的内核的程序中去。
> - 硬件设备的中断: 当硬件设备完成用户请求后，会向cpu发出相应的中断信号， 这时cpu会暂停执行下一条即将要执行的指令，转而去执行与中断信号对应的应用程序， 如果先前执行的指令是用户态下程序的指令，那么这个转换过程也是用户态到内核台的转换。

14. **如何查看占用内存比较多的进程？**

> ps aux | sort -k4nr | head -N
>
> `head`：-N可以指定显示的行数，默认显示10行。
>
> `ps`：a---指代所有的进程，u---userid---执行该进程的用户id，x---指代显示所有程序，不以终端机来区分。ps -aux的输出格式如下：
>
> `sort -k4nr`中：k代表从根据哪一个关键词排序，后面的数字4表示按照第四列排序；n指代numberic sort，根据其数值排序；r指代reverse，这里是指反向比较结果，输出时默认从小到大，反向后从大到小。%MEM在第4个位置，-k4按照内存占用排序。%CPU在第三个位置，-k3表示按照cpu占用率排序。

15. **通过进程id查看占用的端口，通过端口号查看占用的进程id？**

> 通过进程id查看占用的端口 : `netstat -nap | grep 进程id`
>
> 通过端口号查看占用的进程id : `netstat -nap | grep 端口号`

16. **平常用什么linux命令比较多？如何打开文件并进行查找某个单词？怎么在某个目录下找到包含txt的文件？**

> - **`pwd`：** 显示当前所在位置
> - `sudo + 其他命令`：以系统管理者的身份执行指令，也就是说，经由 sudo 所执行的指令就好像是 root 亲自执行。
> - **`grep 要搜索的字符串 要搜索的文件 --color`：** 搜索命令，--color代表高亮显示
> - **`ps -ef`/`ps aux`：** 这两个命令都是查看当前系统正在运行进程，两者的区别是展示格式不同。如果想要查看特定的进程可以使用这样的格式：**`ps aux|grep redis`** （查看包括redis字符串的进程），也可使用`pgrep redis -a`。
>
>   注意：如果直接用ps（（Process Status））命令，会显示所有进程的状态，通常结合grep命令查看某进程的状态。

- **`kill -9` 进程的`pid`：** 杀死进程（-9 表示强制终止），先用ps查找进程，然后用kill杀掉。
- `find 目录 参数`： 寻找目录（查）。在/home目录下查找以.txt结尾的文件名:`find /home -name "*.txt"`
- **`ls`**或者**`ll`**:（ll是ls -l的别名，ll命令可以看到该目录下的所有目录和文件的详细信息）： 查看目录信息。
- **`free`** : 显示系统内存的使用情况，包括物理内存、交换内存(swap)和内核缓冲区内存。
- **`tar -zcvf` 打包压缩后的文件名 要打包压缩的文件**: 打包并压缩文件，一般情况下打包和压缩是一起进行的，打包并压缩后的文件的后缀名一般.tar.gz。c：压缩。
- **`tar -xvf` 压缩文件 `-C` 解压的位置**: 解压压缩包。x: 解压。
- **`wget`**: 是从远程下载的工具。
- **`vmstat`**: 虚拟内存性能监控、CPU监控。
- **`top`**: 常用来监控Linux的系统状况，比如CPU、内存的使用，显示系统上正在运行的进程。load average：系统负载，就是进程队列的长度。当这个值>cpu核心数的时候就说明有进程在等待处理了，是负载过重。

```
find /home -name "*.txt"

cat a.txt | grep redis
```

17. **用过Ping命令么？简单介绍一下。TTL是什么意思?**

> `ping`: 查看与某台机器的连接情况。TTL：生存时间。数据报被路由器丢弃之前允许通过的网段数量。

18. **怎么判断一个主机是不是开放某个端口?**

> `telnet 127.0.0.1 3389` `telnet IP地址 端口`

19. **Linux文件系统?**

> inode 是 linux/unix 文件系统的基础。
>
> 硬盘的最小存储单位是扇区(Sector)，块(block)由多个扇区组成。文件数据存储在块中。块的最常见的大小是 4kb，约为 8 个连续的扇区组成（每个扇区存储 512 字节）。一个文件可能会占用多个块，但是一个块只能存放一个文件。
>
> 我们将文件存储在了块(block)中，但是我们还需要一个空间来存储文件的 **元信息 metadata**： 如某个文件被分成几块、每一块在的地址、文件拥有者，创建时间，权限，大小等。这种 **存储文件元信息的区域就叫 inode**，译为索引节点：**i (index) + node。** 每个文件都有一个 inode，存储文件的元信息。
>
> 可以使用 `stat` 命令可以查看文件的 inode 信息。每个 inode 都有一个号码，Linux/Unix 操作系统不使用文件名来区分文件，而是使用 inode 号码区分不同的文件。
>
> **inode 就是用来维护某个文件被分成几块、每一块在的地址、文件拥有者，创建时间，权限，大小等信息。**

> - **inode**： 记录文件的属性信息，可以使用 stat 命令查看 inode 信息。
> - **block**： 实际文件的内容，如果一个文件大于一个块时候，那么将占用多个 block，但是一个块只能存放一个文件。（因为数据是由 inode 指向的，如果有两个文件的数据存放在同一个块中，就会乱套了）

20. **硬链接和软链接?**

> - 硬链接： 硬连接指通过索引节点`inode`来进行的连接，即每一个硬链接都是一个指向对应区域的文件。
> - 软链接： 保存了其代表的文件的绝对路径，是另外一种文件，在硬盘上有独立的区块，访问时替换自身路径。

21. **中断的分类?**

> 1. 中断可以分为同步中断（synchronous）和异步中断(asynchronous)。
> 2. 中断可分为硬中断和软中断。
> 3. 中断可分为可屏蔽中断（Maskable interrupt）和非屏蔽中断（Nomaskable interrupt）。

> 同步中断是在指令执行时由CPU主动产生的，受到CPU控制，其执行点是可控的。

> 异步中断是CPU被动接收到的，由外设发出的电信号引起，其发生时间不可预测。

22. **软中断和硬中断?**

> 从本质上讲，中断(硬)是一种电信号，当设备有某种事情发生的时候，他就会产生中断，通过总线把电信号发送给中断控制器。如果中断的线是激活的，中断控制器就把电信号发送给处理器的某个特定引脚。处理器于是立即停止自己正在做的事，跳到中断处理程序的入口点，进行中断处理。
>
> 硬中断是由硬件产生的，比如，像磁盘，网卡，键盘，时钟等。每个设备或设备集都有它自己的IRQ（中断请求）。
>
> 软中断是由当前正在运行的进程所产生的。
>
> 软中断比硬中断少了一个硬件发送信号的步骤。产生软中断的进程一定是当前正在运行的进程，因此它们不会中断CPU。但是它们会中断调用代码的流程。如果硬件需要CPU去做一些事情，那么这个硬件会使CPU中断当前正在运行的代码。

---

3. Java 基础

1. **StringBuilder 和 StringBuffer（StringBuffer 是线程安全的，StringBuilder 是不安全的）**
2. **Java实现连续空间的内存分配? java的内存管理 对象的分配与释放**

> 基本数据类型的数组，存放在栈内存里，连续分配
>
> 对象数组,在栈内存里的引用是连续分配的，实际数据分配在堆内存，不是连续分配的;

> java的内存管理，对象的分配与释放
>
> 分配:
>
> 通过new为每个对象申请内存空间（基本类型除外）所有对象都在堆中分配空间;
>
> 释放:
>
> 对象的释放是由垃圾回收机制决定和执行的,这样极大的简化CG(垃圾处理装置)的负担,当然同时也为程序员带来便利(例如c语言需要手动的去处理已经不在使用的对象,如果遗忘内存就会被越占越多)。
>
> 2大类:堆内存与栈内存
>
> （1）在**函数中**定义的基本类型变量（即基本类型的局部变量）和对象的**引用变量**（即对象的变量名指向该对象的内存地址）都在栈内存中分配；
>
> （2）堆内存用来存储由**new创建的对象和数组**以及**对象的实例变量**（即全局变量）
>
> （3）每个线程包含一个栈区，保存基础数据类型的对象和自定义对象的引用，每个栈中的数据（原始类型和对象引用）都是私有的，其他栈不能访问。还有一个方法区：存储所有对象数据共享区域，存储静态变量和普通方法、静态方法、常量、字符串常量等信息，又叫静态区，是所有线程共享的。

3. **创建对象的方式有哪几种?**

> 1. new Obj..()
> 2. clone()：使用Object类的clone方法。
> 3. 反射
>
> > 1. 调用public无参构造器 ，若是没有，则会报异常：Object o = clazz.newInstance();
> > 2. 有带参数的构造函数的类，先获取到其构造对象，再通过该构造方法类获取实例：
> >
> > / /获取构造函数类的对象
> >
> > Constroctor constroctor = User.class.getConstructor(String.class);

```
// 使用构造器对象的newInstance方法初始化对象
Object obj = constroctor.newInstance("name");
```

4. 通过反序列化来创建对象： 实现Serializable接口。

4. **接口和抽象类有什么区别?**
5. **深拷贝和浅拷贝区别?**
6. **讲一讲封装，继承，多态(重要)。**

> 编译时多态
>
> **方法重载**都是编译时多态。根据实际参数的数据类型、个数和次序，Java在编译时能够确定执行重载方法中的哪一个。
>
> **方法覆盖**表现出两种多态性，当对象引用本类实例时，为编译时多态，否则为运行时多态。
>
> 运行时多态
>
> 通过父类对象引用变量引用子类对象来实现。当父类对象引用子类实例时。
>
> 通过接口类型变量引用实现接口的类的对象来实现 。
>
> 运行时多态主要是通过继承和接口实现的。

7. **泛型是什么?类型擦除?**

> 泛型：将类型当作参数传递给一个类或者是方法。
>
> 类型擦除：**Java的泛型是伪泛型，这是因为Java在编译期间，所有的泛型信息都会被擦掉，正确理解泛型概念的首要前提是理解类型擦除。**
>
> Java的泛型基本上都是在编译器这个层次上实现的，在生成的字节码中是不包含泛型中的类型信息的，使用泛型的时候加上类型参数，在编译器编译的时候会去掉，这个过程成为类型擦除。
>
> **原始类型** 就是擦除去了泛型信息，最后在字节码中的类型变量的真正类型，无论何时定义一个泛型，相应的原始类型都会被自动提供，类型变量擦除，并使用其限定类型（无限定的变量用Object）替换。
>
> **Java编译器是通过先检查代码中泛型的类型，然后在进行类型擦除，再进行编译。当具体的类型确定后，泛型提供了一种类型检测的机制，只有相匹配的数据才能正常的赋值，否则编译器就不通过。**

8. **如何实现静态代理？有啥缺陷?**

> 1. 为现有的每一个类都编写一个**对应的**代理类，并且让它实现和目标类相同的接口。
> 2. 在创建代理对象时，通过构造器塞入一个目标对象，然后在代理对象的方法内部调用目标对象同名方法，并在调用前后增加一些其他方法。比如打印日志。代理对象 = 增强代码 + 目标对象。
>
> 需要为每一个目标类编写对应的代理类，工作量太大了。

9. **动态代理的作用？在哪些地方用到了？ (*AOP、RPC 框架中都有用到*)**

> 为其它对象提供一种代理以控制对这个对象的访问控制，在程序运行时，通过反射机制动态生成。
>
> 动态代理的调用处理程序必须实现InvocationHandler接口，及使用Proxy类中的newProxyInstance方法动态的创建代理类。

10. **JDK 的动态代理和 CGLIB 有什么区别?**

> **JDK 动态代理只能只能代理实现了接口的类，而 CGLIB 可以代理未实现任何接口的类。** 另外， CGLIB 动态代理是通过生成一个被代理类的子类来拦截被代理类的方法调用，因此不能代理声明为 final 类型的类和方法。就二者的效率来说，大部分情况都是 JDK 动态代理更优秀，随着 JDK 版本的升级，这个优势更加明显。

11. **谈谈对 Java 注解的理解，解决了什么问题?**

> **Java 语言中的类、方法、变量、参数和包等都可以注解标记，程序运行期间我们可以获取到相应的注解以及注解中定义的内容，这样可以帮助我们做一些事情。比如说 Spring 中如果检测到说你的类被 `@Component` 注解标记的话，Spring 容器在启动的时候就会把这个类归为自己管理，这样你就可以通过 `@Autowired`注解注入这个对象了。**

12. **Java 反射？反射有什么缺点？你是怎么理解反射的（为什么框架需要反射）？**

> **反射介绍：**
>
> JAVA 反射机制是在运行状态中，对于任意一个类，都能够知道这个类的所有属性和方法；对于任意一个对象，都能够调用它的任意一个方法和属性；这种动态获取的信息以及动态调用对象的方法的功能称为 java 语言的反射机制。

> **反射的优缺点如下：**
>
> - **优点：** 运行期类型的判断，动态加载类，提高代码灵活度。
> - **缺点：** 1,性能瓶颈：反射相当于一系列解释操作，通知 JVM 要做的事情，性能比直接的 java 代码要慢很多。2,安全问题，让我们可以动态操作改变类的属性同时也增加了类的安全隐患。

> **为什么框架需要反射技术？**
>
> 在我们平时的项目开发过程中，基本上很少会直接使用到反射机制，但这不能说明反射机制没有用，实际上有很多设计、开发都与反射机制有关。动态代理设计模式也采用了反射机制，还有我们日常使用的 Spring / Hibernate 等框架也大量使用到了反射机制。
>
> 1. 我们在使用 JDBC 连接数据库时使用 `Class.forName()`通过反射加载数据库的驱动程序；
> 2. Spring 框架的 IOC（动态加载管理 Bean）创建对象以及 AOP（动态代理）功能都和反射有联系；
> 3. 动态配置实例的属性；

> **获取 Class 对象的两种方式**
>
> 如果我们动态获取到这些信息，我们需要依靠 Class 对象。Class 类对象将一个类的方法、变量等信息告诉运行的程序。Java 提供了两种方式获取 Class 对象:
>
> 1.知道具体类的情况下可以使用：
>
> ```
> Class alunbarClass = TargetObject.class;
> ```
>
> 但是我们一般是不知道具体类的，基本都是通过遍历包下面的类来获取 Class 对象
>
> 2.通过 `Class.forName()`传入类的路径获取：
>
> ```
> Class alunbarClass1 = Class.forName("cn.javaguide.TargetObject");
> ```

13. **内存泄露和内存溢出的场景。**

> 内存泄漏：内存泄漏是指无用对象（不再使用的对象）持续占有内存或无用对象的内存得不到及时释放，从而造成内存空间的浪费称为内存泄漏。
>
> Java内存泄漏的根本原因是长生命周期的对象持有短生命周期对象的引用就很可能发生内存泄漏，尽管短生命周期对象已经不再需要，但是因为长生命周期持有它的引用而导致不能被回收，这就是Java中内存泄漏的发生场景。
>
> 内存溢出：指程序运行过程中无法申请到足够的内存而导致的一种错误。内存溢出通常发生于OLD段或Perm段垃圾回收后，仍然无内存空间容纳新的Java对象的情况。

> **内存泄露的场景**
>
> 1. **静态集合类引起内存泄漏**：静态成员的生命周期是整个程序运行期间。比如：Map是在堆上动态分配的对象，正常情况下使用完毕后，会被gc回收。而如果Map被static修饰，且没有删除机制，静态成员是不会被回收的，所以会导致这个很大的Map一直停留在堆内存中。懒初始化`static`变量，且尽量避免使用。

2. **当集合里面的对象属性被修改后，再调用remove()方法时不起作用**：修改hashset中对象的参数值，且参数是计算哈希值的字段。当一个对象被存储进HashSet集合中以后，就不能修改这个对象中的那些参与计算哈希值的字段，否则对象修改后的哈希值与最初存储进HashSet集合中时的哈希值就不同了。
3. **各种连接对象(IO流对象、数据库连接对象、网络连接对象)使用后未关闭**：因为每个流在操作系统层面都对应了打开的文件句柄，流没有关闭，会导致操作系统的文件句柄一直处于打开状态，而jvm会消耗内存来跟踪操作系统打开的文件句柄。
4. **监听器的使用**：在释放对象的同时没有相应删除监听器的时候也可能导致内存泄露。
5. **不正确使用单例模式是引起内存泄漏**：单例对象在初始化后将在JVM的整个生命周期中存在（以静态变量的方式），如果**单例对象持有外部的引用**，那么这个对象将不能被JVM正常回收，导致内存泄漏。

解决措施

```
1.尽量减少使用静态变量，类的静态变量的生命周期和类同步的。

2.声明对象引用之前，明确内存对象的有效作用域，尽量减小对象的作用域，将类的成员变量改写为方法内的局部变量；

3.减少长生命周期的对象持有短生命周期的引用；

4.使用StringBuilder和StringBuffer进行字符串连接，Sting和StringBuilder以及StringBuffer等都可以代表字符串，其中String字符串代表的是不可变的字符串，后两者表示可变的字符串。如果使用多个String对象进行字符串连接运算，在运行时可能产生大量临时字符串，这些字符串会保存在内存中从而导致程序性能下降。

5.对于不需要使用的对象手动设置null值，不管GC何时会开始清理，我们都应及时的将无用的对象标记为可被清理的对象；

6.各种连接（数据库连接，网络连接，IO连接）操作，务必显示调用close关闭。
```

内存溢出场景

1. **JVM Heap（堆）溢出：OutOfMemoryError: Java heap space**： 发生这种问题的原因是java虚拟机创建的对象太多，在进行垃圾回收之间，虚拟机分配的到堆内存空间已经用满了。

   JVM在启动的时候会自动设置JVM Heap的值， 可以利用JVM提供的-Xmn -Xms -Xmx等选项可进行设置。Heap的大小是新生代和老年代之和。

   解决方法：1. 手动设置JVM Heap（堆）的大小。2. 检查程序，看是否有死循环或不必要地重复创建大量对象。

2. **Metaspace溢出： java.lang.OutOfMemoryError: Metaspace**：程序中使用了大量的jar或class，使java虚拟机装载类的空间不够，与metaspace大小有关。

   方法区用于存放Java类型的相关信息。在类装载器加载class文件到内存的过程中，虚拟机会提取其中的类型信息，并将这些信息存储到方法区。当需要存储类信息而方法区的内存占用又已经达到-XX:MaxMetaspaceSize设置的最大值，将会抛出OutOfMemoryError异常。对于这种情况的测试，基本的思路是运行时产生大量的类去填满方法区，直到溢出。

   解决方法:通过-XX:MetaspaceSize和-XX:MaxMetaspaceSize设置永久代大小即可。

3. **栈溢出： java.lang.StackOverflowError : Thread Stack space**：线程的方法嵌套调用层次太多(如递归调用)，以致于把栈区溢出了。

   解决方法：

   1：修改程序。2：通过 -Xss: 来设置每个线程的Stack大小即可。

14. **讲一下，强引用，弱引用，软引用，虚引用。**

1. 强引用：被强引用关联的对象不会被回收。使用 new 一个新对象的方式来创建强引用。

```
Object obj = new Object();
```

2. 软引用：被软引用关联的对象只有在内存不够的情况下才会被回收。使用 SoftReference 类来创建软引用。

```
Object obj = new Object();
SoftReference<Object> sf = new SoftReference<Object>(obj);
obj = null;  // 使对象只被软引用关联
```

3. 弱引用：被弱引用关联的对象一定会被回收，也就是说它只能存活到下一次垃圾回收发生之前。

使用 WeakReference 类来创建弱引用。

```
Object obj = new Object();
WeakReference<Object> wf = new WeakReference<Object>(obj);
obj = null;
```

4. 虚引用：一个对象是否有虚引用的存在，不会对其生存时间造成影响，也无法通过虚引用得到一个对象。

15. **讲一下Java的NIO，AIO, BIO?**

BIO (Blocking I/O)：同步阻塞I/O模式，数据的读取写入必须阻塞在一个线程内等待其完成。

NIO (Non-blocking/New I/O): NIO 是一种同步非阻塞的 I/O 模型，对应 java.nio 包，提供了 Channel，Selector，Buffer 等抽象。Java NIO使我们可以进行非阻塞IO操作。比如说，单线程中从通道读取数据到buffer，同时可以继续做别的事情，当数据读取到buffer中后，线程再继续处理数据。写数据也是一样的。另外，非阻塞写也是如此。一个线程请求写入一些数据到某通道，但不需要等待它完全写入，这个线程同时可以去做别的事情。JDK 的 NIO 底层由 epoll 实现。

通常来说NIO中的所有IO都是从 Channel（通道） 开始的。

- 从通道进行数据读取 ：创建一个缓冲区，然后请求通道读取数据。
- 从通道进行数据写入 ：创建一个缓冲区，填充数据，并要求通道写入数据。

AIO (Asynchronous I/O)：异步非阻塞IO模型，异步 IO 是基于事件和回调机制实现的，也就是应用操作之后会直接返回，不会堵塞在那里，当后台处理完成，操作系统会通知相应的线程进行后续的操作。AIO 的应用还不是很广泛。

16. **Java中finalize()方法的使用?**

finalize()是Object的protected方法，子类可以覆盖该方法以实现资源清理工作，GC在回收对象之前调用该方法。

finalize()方法中一般用于释放非Java 资源（如打开的文件资源、数据库连接等）,或是调用非Java方法（native方法）时分配的内存（比如C语言的malloc()系列函数）。

避免使用的原因：

首先，由于finalize()方法的调用时机具有不确定性，从一个对象变得不可到达开始，到finalize()方法被执行，所花费的时间这段时间是任意长的。我们并不能依赖finalize()方法能及时的回收占用的资源，可能出现的情况是在我们耗尽资源之前，gc却仍未触发，因而通常的做法是提供显示的close()方法供客户端手动调用。
另外，重写finalize()方法意味着延长了回收对象时需要进行更多的操作，从而延长了对象回收的时间。

17. **GC Root 对象有哪些**

- 虚拟机栈(栈帧中的本地变量表)中引用的对象
- 本地方法栈(Native方法)中引用的对象
- 方法区中类静态属性引用的对象
- 方法区中常量引用的对象

18. **Java中Class.forName和ClassLoader的区别?**

类的加载：

加载：通过类的全限定名获取二进制字节流，将二进制字节流转换成方法区中的运行时数据结构，在内存中生成Java.lang.class对象；

链接：执行下面的校验、准备和解析步骤，其中解析步骤是可以选择的；

校验：检查导入类或接口的二进制数据的正确性；（文件格式验证，元数据验证，字节码验证，符号引用验证）

准备：给类的静态变量分配内存并初始化内存空间；

解析：将常量池中的符号引用转成直接引用；

初始化：激活类的静态变量的初始化Java代码和静态Java代码块，并初始化程序员设置的变量值。

在java中Class.forName()和ClassLoader都可以对类进行加载。ClassLoader就是遵循**双亲委派模型**最终调用启动类加载器的类加载器，实现的功能是通过一个类的全限定名来获取描述此类的二进制字节流，获取到二进制流后放到JVM中。classloader只干一件事情，就是将.class文件加载到jvm中，不会执行static中的内容。

Class.forName()方法实际上也是调用的CLassLoader来实现的。Class.forName()除了将类的.class文件加载到jvm中之外，还会对类进行初始化，执行类中的static块。

```
@CallerSensitive
 public static Class<?> forName(String className)
             throws ClassNotFoundException {
     Class<?> caller = Reflection.getCallerClass();
     return forName0(className, true, ClassLoader.getClassLoader(caller), caller);
 }
```

最后调用的方法是forName0这个方法，在这个forName0方法中的第二个参数被默认设置为了true，这个参数代表是否对加载的类进行初始化，设置为true时会类进行初始化，代表会执行类中的静态代码块，以及对静态变量的赋值等操作。

19. **讲一下CopyOnWriteArrayList和CopyOnWriteArraySet?**

CopyOnWrite容器：写时复制的容器。当我们往一个容器添加元素的时候，不直接往当前容器添加，而是先将当前容器进行Copy，复制出一个新的容器，然后新的容器里添加元素，添加完元素之后，再将原容器的引用指向新的容器。这样做的好处是我们可以对CopyOnWrite容器进行并发的读，而不需要加锁，因为当前容器不会添加任何元素。所以CopyOnWrite容器也是一种读写分离的思想，读和写不同的容器。

以下代码是向ArrayList里添加元素，可以发现在添加的时候是需要加锁的，否则多线程写的时候会Copy出N个副本出来。

```
public boolean add(T e) {
 final ReentrantLock lock = this.lock;
 lock.lock();
 try {
     Object[] elements = getArray();
     int len = elements.length;
     // 复制出新数组
     Object[] newElements = Arrays.copyOf(elements, len + 1);
     // 把新元素添加到新数组里
     newElements[len] = e;
     // 把原数组引用指向新数组
     setArray(newElements);
     return true;
 } finally {
     lock.unlock();
 }
}

final void setArray(Object[] a) {
```

```
 array = a;
}
```

读的时候不需要加锁，如果读的时候有多个线程正在向ArrayList添加数据，读还是会读到旧的数据，因为写的时候不会锁住旧的ArrayList。

```
public E get(int index) {
 return get(getArray(), index);
}
```

CopyOnWrite并发容器用于读多写少的并发场景。

CopyOnWrite的缺点

CopyOnWrite容器有很多优点，但是同时也存在两个问题，即**内存占用问题**和**数据一致性问题**。所以在开发的时候需要注意一下。

内存占用问题。因为CopyOnWrite的写时复制机制，所以在进行写操作的时候，内存里会同时驻扎两个对象的内存，旧的对象和新写入的对象（注意:在复制的时候只是复制容器里的引用，只是在写的时候会创建新对象添加到新容器里，而旧容器的对象还在使用，所以有两份对象内存）。如果这些对象占用的内存比较大，比如说200M左右，那么再写入100M数据进去，内存就会占用300M，那么这个时候很有可能造成频繁的Yong GC和Full GC。

针对内存占用问题，可以通过压缩容器中的元素的方法来减少大对象的内存消耗，比如，如果元素全是10进制的数字，可以考虑把它压缩成36进制或64进制。或者不使用CopyOnWrite容器，而使用其他的并发容器，如ConcurrentHashMap。

数据一致性问题。CopyOnWrite容器只能保证数据的最终一致性，不能保证数据的实时一致性。所以如果你希望写入的的数据，马上能读到，请不要使用CopyOnWrite容器。

20. **说一下你最用的比较多得模式（我说的工厂模式和观察者模式），再实现一个单例模式。 为什么要用voliate修饰? 出现synchronized为啥还需要voliate，以及synchronized能保证啥?**

观察者模式：观察者模式定义了一系列对象之间的一对多关系。当一个对象改变状态, 其他依赖着都会受到通知。车辆的数据时不断更新的，需要监控数据的变化，当有新数据时就通知观测者observers。

迭代器模式：提供一种顺序访问聚合对象元素的方法。 hasNext() 和 next() 方法。

代理模式：代理模式给某一个对象提供一个代理对象，并由代理对象控制对原对象的引用。

JDK 动态代理和 CGLIB 动态代理均是实现 Spring AOP 的基础。

适配器模式：将一个接口转换成客户希望的另一个接口，使接口不兼容的那些类可以一起工作。

21. **单例模式代码（重要）**

```
// 线程安全，调用效率高，但是不能延时加载，单例未使用的时候便创建完成，可能造成资源浪费。
class Singleton1 {
 private static Singleton1 instance = new Singleton1();
 private Singleton1() {}
 public static Singleton1 getInstance() {
     return instance;
 }
}

// 线程安全，调用效率不高，但是能延时加载，线程安全通过synchronized实现
class Singleton2 {
 private static Singleton2 instance;
 private Singleton2() {}
 public static synchronized Singleton2 getSingleton() {
     if (instance == null) {
         instance = new Singleton2();
     }
```

```
        return instance;
    }
}

// 双重校验锁，线程安全，延迟加载。
// instance = new Singleton3(); 分为三个过程。
// 1. 为instance分配内存空间
// 2. 初始化instance
// 3. 将instance指向分配的内存空间
// 变量如果没有声明成volatile，多线程下会导致一个线程获得一个未初始化的实例。
class Singleton3 {
    private static volatile Singleton3 instance;
    private Singleton3() {}
    public static Singleton3 getSingleton() {
        if (instance == null) {
            synchronized (Singleton3.class) {
                if (instance == null) {
                    instance = new Singleton3();
                }
            }
        }
        return instance;
    }
}

// 静态内部类可以不依赖外部类的实例而被实例化。只有调用getSingleton()才进行初始化。
class Singleton4 {
    private Singleton4() {}
    private static class Inner {
        private static Singleton4 instance = new Singleton4();
    }
    public static Singleton4 getSingleton() {
        return Inner.instance;
    }

}
```

---

4. 集合框架

1. **ArrayList的扩容机制?**

ArrayList扩容发生在add()方法调用的时候，下面是add()方法的源码：

```
public boolean add(E e) {
    //扩容
    ensureCapacityInternal(size + 1);  // Increments modCount!!
    elementData[size++] = e;
    return true;
}
```

ArrayList扩容的关键方法grow():

```
private void grow(int minCapacity) {
    // 获取到ArrayList中elementData数组的内存空间长度
    int oldCapacity = elementData.length;
```

```
    // 扩容至原来的1.5倍
    int newCapacity = oldCapacity + (oldCapacity >> 1);
    // 再判断一下新数组的容量够不够，够了就直接使用这个长度创建新数组，
    // 不够就将数组长度设置为需要的长度
    if (newCapacity - minCapacity < 0)
        newCapacity = minCapacity;
        //若预设值大于默认的最大值检查是否溢出
    if (newCapacity - MAX_ARRAY_SIZE > 0)
         newCapacity = hugeCapacity(minCapacity);
    // 调用Arrays.copyOf方法将elementData数组指向新的内存空间时newCapacity的连续空间
    // 并将elementData的数据复制到新的内存空间
     elementData = Arrays.copyOf(elementData, newCapacity);
    }
```

int newCapacity = oldCapacity + (oldCapacity >> 1);

oldCapacity >> 1 右移运算符 原来长度的一半 再加上原长度也就是每次扩容是原来的1.5倍

之前的所有都是确定新数组的长度，确定之后就是把老数组copy到新数组中，这样数组的扩容就结束了

以上的一切都是ArrayList扩容的第一步，第二步就没啥说的了，就是把需要添加的元素添加到数组的最后一位

2. **HashMap 的底层实现、JDK 1.8 的时候为啥将链表转换成红黑树？HashMap 的负载因子、HashMap 和 Hashtable 的区别？HashMap如何实现扩容?**

> **HashMap是用数组+链表+红黑树进行实现的，当添加一个元素（key-value）时，就首先计算元素key的hash值，并根据hash值来确定插入数组中的位置，但是可能存在其他元素已经被放在数组同一位置了，这个时候便使用链表来解决哈希冲突，当链表长度太长的时候，便将链表转换为红黑树来提高搜索的效率。**
>
> HashMap是基于拉链法实现的一个散列表，内部由数组和链表和红黑树实现。
>
> 1. 数组的初始容量为16，而容量是以2的次方扩充的，一是为了提高性能使用足够大的数组，二是为了能使用位运算代替取模预算。
> 2. 数组是否需要扩充是通过负载因子判断的，如果当前元素个数为数组容量的0.75时，就会扩充数组。这个0.75就是默认的负载因子，可由构造传入。
> 3. 为了解决碰撞，数组中的元素是单向链表类型。当链表长度到达一个阈值时（>=8），会将链表转换成红黑树提高性能。而当链表长度缩小到另一个阈值时（<=6），又会将红黑树转换回单向链表提高性能，这里是一个平衡点。
>
> 当个数不多的时候，直接链表遍历更方便，实现起来也简单。而红黑树的实现要复杂的多。

3. **ConcurrentHashMap的底层实现。**

> 底层数据结构： JDK1.7的 ConcurrentHashMap 底层采用 分段的数组+链表 实现，JDK1.8 采用的数据结构跟HashMap1.8的结构一样，数组+链表/红黑二叉树。Hashtable 和 JDK1.8 之前的 HashMap 的底层数据结构类似都是采用 数组+链表 的形式，数组是 HashMap 的主体，链表则是主要为了解决哈希冲突而存在的；
>
> 实现线程安全的方式（重要）：
> ① 在JDK1.7的时候，ConcurrentHashMap（分段锁） 对整个桶数组进行了分割分段(Segment)，每一把锁只锁容器其中一部分数据，多线程访问容器里不同数据段的数据，就不会存在锁竞争，提高并发访问率。 到了 JDK1.8 的时候已经摒弃了Segment的概念，而是直接用 Node 数组+链表+红黑树的数据结构来实现，并发控制使用 synchronized 和 CAS 来操作。（JDK1.6以后 对 synchronized锁做了很多优化） 整个看起来就像是优化过且线程安全的 HashMap，虽然在JDK1.8中还能看到 Segment 的数据结构，但是已经简化了属性，只是为了兼容旧版本；synchronized只锁定当前链表或红黑二叉树的首节点，这样只要hash不冲突，就不会产生并发，效率又提升N倍。(TreeBin: 红黑二叉树节点 Node: 链表节点)
>
> ② Hashtable(同一把锁) :使用 synchronized 来保证线程安全，效率非常低下。当一个线程访问同步方法时，其他线程也访问同步方法，可能会进入阻塞或轮询状态，如使用 put 添加元素，另一个线程不能使用 put 添加元素，也不能使用 get，竞争会越来越激烈效率越低。

> **为什么 ConcurrentHashMap 的读操作不需要加锁?**

> ConcurrentHashMap 在jdk1.7中是采用Segment + HashEntry + ReentrantLock的方式进行实现的，而1.8中放弃了Segment臃肿的设计，取而代之的是采用Node + CAS + Synchronized来保证并发安全进行实现。
>
> 总结：定义成volatile的变量，能够在线程之间保持可见性，能够被多线程同时读，而不会读到过期的值。由于get操作中只需要读不需要写共享变量，所以可以不用加锁。之所以不会读到过期的值，依据Java内存模型的happen before原则，对volatile字段的写入操作先于读操作，get总能拿到最新的值。
>
> - get操作全程不需要加锁是因为Node的成员val是用volatile修饰的和数组用volatile修饰没有关系。
> - 数组用volatile修饰主要是保证在数组扩容的时候保证可见性。

> **HashMap，LinkedHashMap，TreeMap 有什么区别？HashMap ，TreeMap，LinkedHashMap 使用场景？**
>
> LinkedHashMap 保存了记录的插入顺序，在用 Iterator 遍历时，先取到的记录肯定是先插入的；遍历比 HashMap 慢；
>
> TreeMap 实现 SortMap 接口，能够把它保存的记录根据键排序（默认按键值升序排序，也可以指定排序的比较器）
>
> 一般情况下，使用最多的是 HashMap。
>
> HashMap：在 Map 中插入、删除和定位元素时；
>
> TreeMap：在需要按自然顺序或自定义顺序遍历键的情况下；
>
> LinkedHashMap：在需要输出的顺序和输入的顺序相同的情况下。

4. **有哪些集合是线程不安全的，又有哪些集合是线程不安全的？怎么解决呢？**

> **线程安全的集合类**
>
> Vector
>
> Stack
>
> Hashtable
>
> java.util.concurrent包下所有的集合类（ConcurrentHashMap， CopyOnWriteArrayList和CopyOnWriteArraySet等）

5. **什么是快速失败(fail-fast)、能举个例子吗？什么是安全失败(fail-safe)呢？**

> **快速失败(fail-fast)**
>
> 快速失败(fail-fast)是 Java 集合的一种**错误检测机制。在使用迭代器对集合进行遍历的时候**，我们在多线程下操作非安全失败(fail-safe)的集合类可能就会触发 fail-fast 机制，导致抛出 `ConcurrentModificationException` 异常。另外，在单线程下，如果在遍历过程中对集合对象的内容进行了修改的话也会触发 fail-fast 机制。
>
> 举个例子：多线程下，如果线程 1 正在对集合进行遍历，此时线程 2 对集合进行修改（增加、删除、修改），或者线程 1 在遍历过程中对集合进行修改，都会导致线程 1 抛出 `ConcurrentModificationException` 异常。
>
> **安全失败(fail-safe)**
>
> 采用安全失败机制的集合容器，在遍历时不是直接在集合内容上访问的，而是**先复制原有集合内容，在拷贝的集合上进行遍历。**所以，在遍历过程中对原集合所作的修改并不能被迭代器检测到，故不会抛 `ConcurrentModificationException` 异常。

6. **HashMap 多线程操作导致死循环问题**

> 主要原因在于并发下的 rehash 会造成元素之间会形成一个循环链表。不过，jdk 1.8 后解决了这个问题，但是还是不建议在多线程下使用 HashMap，因为多线程下使用 HashMap 还是会存在其他问题比如数据丢失。

---

5. 多线程

1. **Java 怎么实现线程?**

> ① 继承 Thread 类并重写 run 方法。实现简单，但不符合里氏替换原则，不可以继承其他类。
>
> ② 实现 Runnable 接口并重写 run 方法。避免了单继承局限性，实现解耦。
>
> ③实现 Callable 接口并重写 call 方法。可以获取线程执行结果的返回值，并且可以抛出异常。

2. **Java 线程通信的方式?**

> Java 采用共享内存模型，线程间的通信总是隐式进行，整个通信过程对程序员完全透明。
>
> volatile 告知程序任何对变量的读需要从主内存中获取，写必须同步刷新回主内存，保证所有线程对变量访问的可见性。
>
> synchronized 确保多个线程在同一时刻只能有一个处于方法或同步块中，保证线程对变量访问的原子性、可见性和有序性。
>
> 等待通知机制指一个线程 A 调用了对象的 wait 方法进入等待状态，另一线程 B 调用了对象的 notify/notifyAll 方法，线程 A 收到通知后结束阻塞并执行后序操作。对象上的 wait 和 notify/notifyAll 完成等待方和通知方的交互。
>
> 如果一个线程执行了某个线程的 join 方法，这个线程就会阻塞等待执行了 join 方法的线程终止，这里涉及等待/通知机制。join 底层通过 wait 实现，线程终止时会调用自身的 notifyAll 方法，通知所有等待在该线程对象上的线程。
>
> 管道 IO 流用于线程间数据传输，媒介为内存。PipedOutputStream 和 PipedWriter 是输出流，相当于生产者，PipedInputStream 和 PipedReader 是输入流，相当于消费者。管道流使用一个默认大小为 1KB 的循环缓冲数组。输入流从缓冲数组读数据，输出流往缓冲数组中写数据。当数组已满时，输出流所在线程阻塞；当数组首次为空时，输入流所在线程阻塞。
>
> ThreadLocal 是线程共享变量，但它可以为每个线程创建单独的副本，副本值是线程私有的，互相之间不影响。

3. **在多线程情况下如何保证线程安全。**

4. **写一个死锁的例子**

> ```
> public class DeadLock implements Runnable {
>  public int flag = 1;
>  //静态对象是类的所有对象共享的
>  private static Object o1 = new Object(), o2 = new Object();
>  @Override
>  public void run() {
>      System.out.println("flag=" + flag);
>      if (flag == 1) {
>          synchronized (o1) {
>              try {
>                  Thread.sleep(500);
>              } catch (Exception e) {
>                  e.printStackTrace();
>              }
>              synchronized (o2) {
>                  System.out.println("1");
>              }
>          }
>      }
>      if (flag == 0) {
>          synchronized (o2) {
>              try {
>                  Thread.sleep(500);
>              } catch (Exception e) {
> ```

```
                e.printStackTrace();
            }
            synchronized (o1) {
                System.out.println("0");
            }
        }
    }
}

public static void main(String[] args) {

    DeadLock td1 = new DeadLock();
    DeadLock td2 = new DeadLock();
    td1.flag = 1;
    td2.flag = 0;
    //td1,td2都处于可执行状态，但JVM线程调度先执行哪个线程是不确定的。
    //td2的run()可能在td1的run()之前运行
    new Thread(td1).start();
    new Thread(td2).start();

}
}
```

5. **讲一下volatile关键字的作用。**

> 1. **保证了不同线程对该变量操作的内存可见性。**
> 2. **禁止指令重排序。**

> **当写一个volatile变量时，JMM将本地内存更改的变量写回到主内存中。**
>
> **当取一个volatile变量时，JMM将使线程对应的本地内存失效，然后线程将从主内存读取共享变量。**
>
> volatile 可以保证线程可见性且提供了一定的有序性，但是无法保证原子性。在 JVM 底层是基于内存屏障实现的。
>
> - 当对非 volatile 变量进行读写的时候，每个线程先从内存拷贝变量到 CPU 缓存中。如果计算机有多个 CPU，每个线程可能在不同的 CPU 上被处理，这意味着每个线程可以拷贝到不同的本地内存中。
> - 而声明变量是 volatile 的，JVM 保证了每次读变量都从内存中读，跳过本地内存这一步，所以就不会有可见性问题
>   - **对 volatile 变量进行写操作时，会在写操作后加一条 store 屏障指令，将工作内存中的共享变量刷新回主内存；**
>   - **对 volatile 变量进行读操作时，会在读操作前加一条 load 屏障指令，从主内存中读取共享变量；**
>
> **volatile可以通过内存屏障防止指令重排序问题。硬件层面的内存屏障分为读屏障和写屏障。**
>
> **对于读屏障来说，在指令前插入读屏障，可以让高速缓存中的数据失效，强制重新从主内存加载数据。**
> **对于写屏障来说，在指令后插入写屏障，能让写入缓存中的最新数据更新写入主内存，让其他线程可见。**

6. **synchronized 作用，讲一讲底层实现。**

> synchronized关键字解决的是多个线程之间访问资源的同步性，调用操作系统内核态做同步，synchronized关键字可以保证被它修饰的方法或者代码块在任意时刻只能有一个线程执行。
>
> 在 Java 早期版本中，synchronized属于重量级锁，效率低下，因为监视器锁（monitor）是依赖于底层的操作系统的 Mutex Lock 来实现的，Java 的线程是映射到操作系统的原生线程之上的。JDK1.6对锁的实现引入了大量的优化，如自旋锁、适应性自旋锁、锁消除、锁粗化、偏向锁、轻量级锁等技术来减少锁操作的开销。

> **synchronized关键字最主要的三种使用方式：**
>
> - **修饰实例方法:** 作用于当前对象实例加锁，进入同步代码前要获得当前对象实例的锁

- **修饰静态方法:** 也就是给当前类加锁，会作用于类的所有对象实例。因为访问静态 synchronized 方法占用的锁是当前类的锁，而访问非静态 synchronized 方法占用的锁是当前实例对象锁。
- **修饰代码块:** 指定加锁对象，对给定对象加锁，进入同步代码库前要获得给定对象的锁。

synchronized 关键字底层原理属于 JVM 层面。

① synchronized 同步语句块的情况：**synchronized 同步语句块的实现使用的是 monitorenter 和 monitorexit 指令，其中 monitorenter 指令指向同步代码块的开始位置，monitorexit 指令则指明同步代码块的结束位置。**

② synchronized 修饰方法的的情况：**JVM 通过该 ACC_SYNCHRONIZED 访问标志来辨别一个方法是否声明为同步方法，从而执行相应的同步调用。**

7. **ReetrantLock 和 synchronized 的区别**
8. **说说 synchronized 关键字和 volatile 关键字的区别**

- **volatile关键字**是线程同步的**轻量级实现**，所以**volatile性能肯定比synchronized关键字要好。**但是**volatile关键字只能用于变量而synchronized关键字可以修饰方法以及代码块。**synchronized关键字在JavaSE1.6之后进行了主要包括为了减少获得锁和释放锁带来的性能消耗而引入的偏向锁和轻量级锁以及其它各种优化之后执行效率有了显著提升，**实际开发中使用 synchronized 关键字的场景还是更多一些。**
- **多线程访问volatile关键字不会发生阻塞，而synchronized关键字可能会发生阻塞**
- **volatile关键字能保证数据的可见性，但不能保证数据的原子性。synchronized关键字两者都能保证。**
- **volatile关键字主要用于解决变量在多个线程之间的可见性，而 synchronized关键字解决的是多个线程之间访问资源的同步性。**

9. **ReetrantLock实现方式**

锁的获取过程：

1. 通过cas操作来修改state状态，表示争抢锁的操作，如果能够获取到锁，设置当前获得锁状态的线程。compareAndSetState(0, 1)
2. 如果没有获取到锁，尝试去获取锁。acquire(1)。

   （1）通过tryAcquire尝试获取独占锁，如果成功返回true，失败返回false。如果是同一个线程来获得锁，则直接增加重入次数，并返回true。

   （2）如果tryAcquire失败，则会通过addWaiter方法将当前线程封装成Node，添加到AQS队列尾部

   （3）acquireQueued，将Node作为参数，通过自旋去尝试获取锁。（如果前驱为head才有资格进行锁的抢夺。）

   （4）如果获取锁失败，则挂起线程。

锁的释放过程：

1. 释放锁。
2. 如果锁能够被其他线程获取，唤醒后继节点中的线程。一般情况下只要唤醒后继结点的线程就行了，但是后继结点可能已经取消等待，所以从队列尾部往前回溯，找到离头结点最近的正常结点，并唤醒其线程。

在获得同步锁时，同步器维护一个同步队列，获取状态失败的线程都会被加入到队列中并在队列中进行自旋；移出队列（或停止自旋）的条件是前驱节点为头节点且成功获取了同步状态。在释放同步状态时，同步器调用tryRelease(int arg)方法释放同步状态，然后唤醒头节点的后继节点。

10. **AQS原理**

AQS是一个用来构建锁和同步器的框架。AQS核心思想是，如果被请求的共享资源空闲，则将当前请求资源的线程设置为有效的工作线程，并且将共享资源设置为锁定状态。如果被请求的共享资源被占用，那么就**需要一套线程阻塞等待以及被唤醒时锁分配的机制**，这个机制AQS是用CLH队列锁实现的，即将暂时获取不到锁的线程加入到队列中。

CLH队列是一个虚拟的双向队列（虚拟的双向队列即不存在队列实例，仅存在结点之间的关联关系）。AQS是将每条请求共享资源的线程封装成一个CLH锁队列的一个结点（Node）来实现锁的分配。

AQS使用一个int成员变量来表示同步状态，通过内置的FIFO队列来完成获取资源线程的排队工作。AQS使用CAS对该同步状态进行原子操作实现对其值的修改。

状态信息通过protected类型的getState，setState，compareAndSetState进行操作。

AQS定义两种资源共享方式

Exclusive（独占）：只有一个线程能执行，如ReentrantLock。又可分为公平锁和非公平锁：

Share（共享）：多个线程可同时执行。

以ReentrantLock为例，state初始化为0，表示未锁定状态。A线程lock()时，会调用tryAcquire()独占该锁并将state+1。此后，其他线程再tryAcquire()时就会失败，直到A线程unlock()到state=0（即释放锁）为止，其它线程才有机会获取该锁。当然，释放锁之前，A线程自己是可以重复获取此锁的（state会累加），这就是可重入的概念。但要注意，获取多少次就要释放多么次，这样才能保证state是能回到零态的。

再以CountDownLatch以例，任务分为N个子线程去执行，state也初始化为N（注意N要与线程个数一致）。这N个子线程是并行执行的，每个子线程执行完后countDown()一次，state会CAS(Compare and Swap)减1。等到所有子线程都执行完后(即state=0)，会unpark()主调用线程，然后主调用线程就会从await()函数返回，继续后余动作。

11. **interrupt，interrupted与isInterrupted方法的区别? 如何停止一个正在运行的线程**

（1）interrupt()方法的作用实际上是：在线程受到阻塞时抛出一个中断信号，这样线程就得以退出阻塞状态。

（2）interrupted()调用的是currentThread().isInterrupted(true)方法，即说明是返回当前线程的是否已经中断的状态值，而且有清理中断状态的机制。

测试当前线程是否已经中断，线程的中断状态由该方法清除。即如果连续两次调用该方法，则第二次调用将返回false（在第一次调用已清除flag后以及第二次调用检查中断状态之前，当前线程再次中断的情况除外）所以，interrupted()方法具有清除状态flag的功能

（3）isInterrupted()调用的是isInterrupted(false)方法，意思是返回线程是否已经中断的状态，它没有清理中断状态的机制。

```
public void interrupt() {
 if (this != Thread.currentThread())
     checkAccess();
 synchronized (blockerLock) {
     Interruptible b = blocker;
     if (b != null) {
         interrupt0();           // Just to set the interrupt flag
         b.interrupt(this);
         return;
     }
 }
 interrupt0();
}
```

interrupt() 方法用于中断线程。调用该方法的线程的状态为将被置为"中断"状态。
中断可以理解为线程的一个标识位属性，它表示一个运行中的线程是否被其他线程进行了中断操作。中断好比其他线程对该线程打了个招呼，其他线程通过调用该线程的interrupt()方法对其进行中断操作。
注意：**线程中断仅仅是置线程的中断状态位，不会停止线程。需要用户自己去监视线程的状态为并做处理。**

```
public static boolean interrupted() {
 return currentThread().isInterrupted(true);
}
```

interrupted() 检测当前线程是否已经中断，是则返回true，否则false，并清除中断状态。换言之，如果该方法被连续调用两次，第二次必将返回false，除非在第一次与第二次的瞬间线程再次被中断。如果中断调用时线程已经不处于活动状态，则返回false。

```
public boolean isInterrupted() {
 return isInterrupted(false);
}
```

isInterrupted() 检测当前线程是否已经中断，是则返回true，否则false。中断状态不受该方法的影响。如果中断调用时线程已经不处于活动状态，则返回false。

在java中有以下3种方法可以终止正在运行的线程：

- 使用stop方法强行终止，但是不推荐这个方法，因为stop和suspend及resume一样都是过期作废的方法
- 使用interrupt()方法中断线程

12. **线程池作用？Java 线程池有哪些参数？阻塞队列有几种？拒绝策略有几种？线程池的工作机制？（非大厂会问：有哪些线程池）**

- **降低资源消耗。**通过重复利用已创建的线程降低线程创建和销毁造成的消耗。
- **提高响应速度。**当任务到达时，任务可以不需要的等到线程创建就能立即执行。
- **提高线程的可管理性。**线程是稀缺资源，如果无限制的创建，不仅会消耗系统资源,还会降低系统的稳定性，使用线程池可以进行统一的分配，调优和监控。

线程池通过 ThreadPoolExecutor 的方式进程创建

```
public ThreadPoolExecutor(int corePoolSize,
                          int maximumPoolSize,
                          long keepAliveTime,
                          TimeUnit unit,
                          BlockingQueue<Runnable> workQueue,
                          ThreadFactory threadFactory,
                          RejectedExecutionHandler handler)
```

`ThreadPoolExecutor` 3 个最重要的参数：

- **`corePoolSize`：**核心线程数线程数定义了最小可以同时运行的线程数量。
- **`maximumPoolSize`：**当队列中存放的任务达到队列容量的时候，当前可以同时运行的线程数量变为最大线程数。
- **`workQueue`:** 当新任务来的时候会先判断当前运行的线程数量是否达到核心线程数，如果达到的话，新任务就会被存放在队列中。

`ThreadPoolExecutor`其他常见参数:

1. **`keepAliveTime`**:当线程池中的线程数量大于 `corePoolSize` 的时候，如果这时没有新的任务提交，核心线程外的线程不会立即销毁，而是会等待，直到等待的时间超过了 `keepAliveTime`才会被回收销毁；
2. **`unit`** : `keepAliveTime` 参数的时间单位。
3. **`threadFactory`** :executor 创建新线程的时候会用到。
4. **`handler`** :饱和策略。关于饱和策略下面单独介绍一下。

阻塞队列有几种

用来保存等待被执行的任务的阻塞队列，且任务必须实现Runable接口，在JDK中提供了如下阻塞队列：

1、ArrayBlockingQueue（有界队列）：基于数组结构的有界阻塞队列，按FIFO排序任务；

2、LinkedBlockingQuene（有/无界队列（基于链表的，传参就是有界，不传就是无界）：基于链表结构的阻塞队列，按FIFO排序任务，吞吐量通常要高于ArrayBlockingQuene；

3、SynchronousQuene（同步移交队列（需要一个线程调用put方法插入值，另一个线程调用take方法删除值））：一个不存储元素的阻塞队列，每个插入操作必须等到另一个线程调用移除操作，否则插入操作一直处于阻塞状态，吞吐量通常要高于LinkedBlockingQuene；

4、PriorityBlockingQuene（具有优先级的、无限阻塞队列）：具有优先级的无界阻塞队列；

`ThreadPoolExecutor` 饱和策略定义:

如果当前同时运行的线程数量达到最大线程数量并且队列也已经被放满了任时，`ThreadPoolTaskExecutor` 定义一些策略:

- **`ThreadPoolExecutor.AbortPolicy`**：抛出 `RejectedExecutionException`来拒绝新任务的处理。
- **`ThreadPoolExecutor.CallerRunsPolicy`**：调用执行自己的线程运行任务。也就是直接在调用`execute`方法的线程中运行(run)被拒绝的任务，如果执行程序已关闭，则会丢弃该任务。因此这种策略会降低对于新任务提交速度，影响程序的整体性能。如果您的应用程序可以承受此延迟并且你要求任何一个任务请求都要被执行的话，你可以选择这个策略。
- **`ThreadPoolExecutor.DiscardPolicy`**： 不处理新任务，直接丢弃掉。
- **`ThreadPoolExecutor.DiscardOldestPolicy`**： 此策略将丢弃最早的未处理的任务请求。

线程池的工作过程

1. 提交任务后，线程池先判断线程数是否达到了核心线程数（corePoolSize）。如果未达到线程数，则创建核心线程处理任务；否则，就执行下一步；
2. 接着线程池判断任务队列是否满了。如果没满，则将任务添加到任务队列中；否则，执行下一步；
3. 接着因为任务队列满了，线程池就判断线程数是否达到了最大线程数。如果未达到，则创建非核心线程处理任务；否则，就执行饱和策略，默认会抛出RejectedExecutionException异常。

常见线程池

1. newFixedThreadPool：最大线程和核心线程一致，用的是LinkedBlockingQueue，无限容量。

```
public static ExecutorService newFixedThreadPool(int nThreads) {
    return new ThreadPoolExecutor(nThreads, nThreads,
                                  0L, TimeUnit.MILLISECONDS,
                                  new LinkedBlockingQueue<Runnable>());
}
```

2. newSingleThreadExecutor：最大线程和核心线程一致，用的是LinkedBlockingQueue，无限容量。

```
public static ExecutorService newSingleThreadExecutor() {
    return new FinalizableDelegatedExecutorService
        (new ThreadPoolExecutor(1, 1,
                                0L, TimeUnit.MILLISECONDS,
                                new LinkedBlockingQueue<Runnable>()));
}
```

3. newCachedThreadPool：没有核心线程,直接向 SynchronousQueue 中提交任务，如果有空闲线程，就去取出任务执行。如果没有空闲线程，就新建一个。执行完任务的线程有 60 秒生存时间，如果在这个时间内可以接到新任务，才可以存活下去。

```
public static ExecutorService newCachedThreadPool() {
    return new ThreadPoolExecutor(0, Integer.MAX_VALUE,
                                  60L, TimeUnit.SECONDS,
                                  new SynchronousQueue<Runnable>());
}
```

4. newScheduledThreadPool：核心线程和最大线程都有，采用DelayedWorkQueue 队列。

```
public static ScheduledExecutorService newScheduledThreadPool(int corePoolSize) {
    return new ScheduledThreadPoolExecutor(corePoolSize);
}
public ScheduledThreadPoolExecutor(int corePoolSize) {
    super(corePoolSize, Integer.MAX_VALUE,
          DEFAULT_KEEPALIVE_MILLIS, MILLISECONDS,
          new DelayedWorkQueue());
}
private static final long DEFAULT_KEEPALIVE_MILLIS = 10L;
```

13. **线程池拒绝策略分别使用在什么场景?**

1. AbortPolicy中止策略：丢弃任务并抛出RejectedExecutionException异常。

   使用场景：这个就没有特殊的场景了，但是有一点要正确处理抛出的异常。当自己自定义线程池实例时，使用这个策略一定要处理好触发策略时抛的异常，因为他会打断当前的执行流程。

2. DiscardPolicy丢弃策略：ThreadPoolExecutor.DiscardPolicy：丢弃任务，但是不抛出异常。如果线程队列已满，则后续提交的任务都会被丢弃，且是静默丢弃。

   使用场景：如果你提交的任务无关紧要，你就可以使用它 。

3. DiscardOldestPolicy弃老策略：丢弃队列最前面的任务，然后重新提交被拒绝的任务。

   使用场景：这个策略还是会丢弃任务，丢弃时也是毫无声息，但是特点是丢弃的是老的未执行的任务，而且是待执行优先级较高的任务。基于这个特性，能想到的场景就是，发布消息和修改消息，当消息发布出去后，还未执行，此时更新的消息又来了，这个时候未执行的消息的版本比现在提交的消息版本要低就可以被丢弃了。

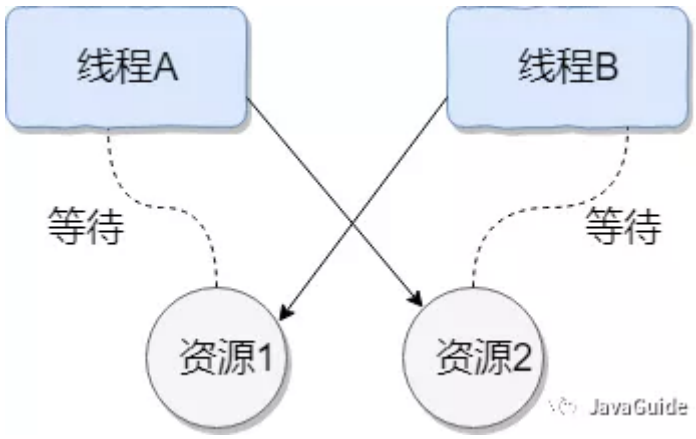
4. CallerRunsPolicy调用者运行策略：由调用线程处理该任务。

   使用场景：一般在不允许失败的、对性能要求不高、并发量较小的场景下使用。

14. **线程死锁，解除线程死锁有哪几种方式？(两次栽倒这题上了，时间太久又忘记了，如何解决很重要)**

线程死锁描述的是这样一种情况：多个线程同时被阻塞，它们中的一个或者全部都在等待某个资源被释放。由于线程被无限期地阻塞，因此程序不可能正常终止。

如下图所示，线程 A 持有资源 2，线程 B 持有资源 1，他们同时都想申请对方的资源，所以这两个线程就会互相等待而进入死锁状态。



解决死锁的策略

- 死锁预防
- 死锁避免
- 死锁检测
- 死锁解除

1. **死锁预防：** 破坏导致死锁必要条件中的任意一个就可以预防死锁。
   （1）**破坏占有和等待条件：** 一次性申请所有资源，之后不再申请资源，如果不满足资源条件则得不到资

源分配。

（2）**破坏不可剥夺条件：** 当一个进程获得某个不可剥夺的资源时，提出新的资源申请，若不满足，则释放所有资源。

（3）**破坏循环等待条件：** 按某一顺序申请资源，释放资源则反序释放。

2. **死锁避免：** 进程在每次申请资源时判断这些操作是否安全。
3. **死锁检测：** 判断系统是否属于死锁的状态，如果是，则执行死锁解除策略。
4. **死锁解除：** 将某进程所占资源进行强制回收，然后分配给其他进程。（与死锁检测结合使用的）

15. **ThreadLocal 是什么，应用场景是什么，原理是怎样的?**

通常情况下，我们创建的变量是可以被任何一个线程访问并修改的。**如果想实现每一个线程都有自己的专属本地变量该如何解决呢？** JDK 中提供的`ThreadLocal`类正是为了解决这样的问题。**`ThreadLocal`类主要解决的就是让每个线程绑定自己的值，可以将`ThreadLocal`类形象的比喻成存放数据的盒子，盒子中可以存储每个线程的私有数据。**

如果你创建了一个`ThreadLocal`变量，那么访问这个变量的每个线程都会有这个变量的本地副本，这也是`ThreadLocal`变量名的由来。他们可以使用 `get()` 和 `set()`方法来获取默认值或将其值更改为当前线程所存的副本的值，从而避免了线程安全问题。

`ThreadLocal`最终的变量是放在了当前线程的 `ThreadLocalMap` 中，并不是存在 `ThreadLocal` 上，`ThreadLocal` 可以理解为只是`ThreadLocalMap`的封装，传递了变量值。 我们可以把 `ThreadLocalMap` 理解为 `ThreadLocal` 类实现的定制化的 `HashMap`。`ThrealLocal` 类中可以通过`Thread.currentThread()`获取到当前线程对象后，直接通过`getMap(Thread t)`可以访问到该线程的`ThreadLocalMap`对象。

每个`Thread`中都具备一个`ThreadLocalMap`，而`ThreadLocalMap`可以存储以`ThreadLocal`为 key ， Object 对象为 value 的键值对。

```
ThreadLocalMap(ThreadLocal<?> firstKey, Object firstValue) {
......
}
```

比如我们在同一个线程中声明了两个 `ThreadLocal` 对象的话，会使用 `Thread`内部都是使用仅有那个 `ThreadLocalMap` 存放数据的，`ThreadLocalMap`的 key 就是 `ThreadLocal`对象，value 就是 `ThreadLocal` 对象调用`set`方法设置的值。

16. **ThreadLocal类为什么要加上private static修饰?**

首先，private修饰与ThreadLocal本身没有关系，private更多是在安全方面进行考虑。static修饰这个变量，这个变量是针对一个线程内所有操作共享的，所以设置为静态变量，所有此类实例共享此静态变量，也就是说在类第一次被使用时装载，只分配一块存储空间，所有此类的对象(只要是这个线程内定义的)都可以操控这个变量。（设置为static可以避免每个线程从任务队列中获取task后重复创建ThreadLocal所关联的对象）

可以解决内存泄露问题（看下一问）。

17. **ThreadLocal 有什么缺陷？如果线程池的线程使用ThreadLocal 会有什么问题?**

采用线性探测的方式。ThreadLocalMap如何解决冲突?

```
public class Thread implements Runnable {
......
ThreadLocal.ThreadLocalMap threadLocals = null;
ThreadLocal.ThreadLocalMap inheritableThreadLocals = null;
......
}
```

`ThreadLocalMap` 中使用的 key 为 `ThreadLocal` 的弱引用,而 value 是强引用。所以，如果 `ThreadLocal` 没有被外部强引用的情况下，在垃圾回收的时候，key 会被清理掉，而 value 不会被清理掉。这样一来，`ThreadLocalMap` 中就会出现key为null的Entry。假如我们不做任何措施的话，value 永远无法被GC 回收，这个时候就可能会产生内存泄露。ThreadLocalMap实现中已经考虑了这种情况，在调用 `set()`、`get()`、

remove() 方法的时候，会清理掉 key 为 null 的记录。使用完 ThreadLocal方法后 最好手动调用remove()方法。

```
static class Entry extends WeakReference<ThreadLocal<?>> {
    /** The value associated with this ThreadLocal. */
    Object value;
    Entry(ThreadLocal<?> k, Object v) {
        super(k);
        value = v;
    }
}
```

在ThreadLocalMap中，也是用Entry来保存K-V结构数据的。但是Entry中key只能是ThreadLocal对象，这点被Entry的构造方法已经限定死了。**Entry继承自WeakReference（弱引用，生命周期只能存活到下次GC前），但只有Key是弱引用类型的，Value并非弱引用。**由于ThreadLocalMap的key是弱引用，而Value是强引用。这就导致了一个问题，ThreadLocal在没有外部对象强引用时，**发生GC时弱引用Key会被回收，而Value不会回收。**当线程没有结束，但是ThreadLocal已经被回收，则可能导致线程中存在ThreadLocalMap<null, Object>的键值对，**造成内存泄露。**（ThreadLocal被回收，ThreadLocal关联的线程共享变量还存在）。

为了防止此类情况的出现，我们有两种手段。

- 1、使用完线程共享变量后，显示调用ThreadLocalMap.remove()方法清除线程共享变量；

既然Key是弱引用，那么我们要做的事，就是在调用ThreadLocal的get()、set()方法时完成后再**调用remove方法，将Entry节点和Map的引用关系移除**，这样整个Entry对象在GC Roots分析后就变成不可达了，下次GC的时候就可以被回收。

- 2、JDK建议ThreadLocal定义为private static，这样ThreadLocal的弱引用问题则不存在了。

18. **介绍一下 Java 有哪些锁**

(synchronized、juc 提供的锁如 ReentrantLock、CountDownLatch、CyclicBarrier、Semaphore等)

- 公平锁/非公平锁 (重要)
- 可重入锁
- 独享锁/共享锁 (重要)
- 互斥锁/读写锁
- 乐观锁/悲观锁 (重要)
- 偏向锁/轻量级锁/重量级锁 (重要)
- 自旋锁

1. 公平锁/非公平锁

   公平锁是指多个线程按照申请锁的顺序来获取锁。
   非公平锁是指多个线程获取锁的顺序并不是按照申请锁的顺序，有可能后申请的线程比先申请的线程优先获取锁。有可能，会造成优先级反转或者饥饿现象。

   对于Java ReentrantLock而言，通过构造函数指定该锁是否是公平锁，默认是非公平锁。非公平锁的优点在于吞吐量比公平锁大。对于Synchronized而言，也是一种非公平锁。由于其并不像ReentrantLock是通过AQS的来实现线程调度，所以并没有任何办法使其变成公平锁。

2. 可重入锁

   可重入锁又名递归锁，是指在同一个线程在外层方法获取锁的时候，在进入内层方法会自动获取锁。

   对于Java ReentrantLock而言，是一个可重入锁，其名字是Re entrant Lock重新进入锁。
   对于Synchronized而言，也是一个可重入锁。可重入锁的一个好处是可一定程度避免死锁。

3. 独享锁/共享锁 (互斥锁/读写锁)

   独享锁是指该锁一次只能被一个线程所持有。
   共享锁是指该锁可被多个线程所持有。

> 对于Java ReentrantLock而言，其是独享锁。但是对于Lock的另一个实现类ReadWriteLock，其读锁是共享锁，其写锁是独享锁。
>
> 上面讲的独享锁/共享锁就是一种广义的说法，互斥锁/读写锁就是具体的实现。
>
> 互斥锁在Java中的具体实现就是ReentrantLock
> 读写锁在Java中的具体实现就是ReadWriteLock

> 4. 乐观锁/悲观锁
>
> 乐观锁与悲观锁不是指具体的什么类型的锁，而是指**看待并发同步的角度。**
>
> 对于同一个数据的并发操作，悲观锁采取加锁的形式。悲观的认为，不加锁的并发操作一定会出问题。
> 乐观锁在更新数据的时候，主要就是两个步骤：冲突检测和数据更新。乐观的认为，不加锁的并发操作是没有事情的。当多个线程尝试使用CAS同时更新同一个变量时，只有其中一个线程能更新变量的值，而其它线程都失败，**失败的线程并不会被挂起，而是被告知这次竞争中失败，并可以再次尝试。**
>
> 从上面的描述我们可以看出，悲观锁适合写操作非常多的场景，乐观锁适合读操作非常多的场景，不加锁会带来大量的性能提升。
> 悲观锁在Java中的使用，就是利用各种锁。
> 乐观锁在Java中的使用，是无锁编程，常常采用的是CAS算法，典型的例子就是原子类，通过CAS自旋实现原子操作的更新。
>
> > CAS包含三个参数 CAS（V,E,N）。V表示要更新的变量，E表示预期的值，N表示新值。仅当要更新的变量值等于预期的值时，才会将要更新的变量值的值设置成新值，否则什么都不做。

> 5. 偏向锁/轻量级锁/重量级锁
>
> 这三种锁是指锁的状态，并且是针对Synchronized。
>
> 偏向锁是指一段同步代码一直被一个线程所访问，那么该线程会自动获取锁。降低获取锁的代价。
> 轻量级锁是指当锁是偏向锁的时候，被另一个线程所访问，偏向锁就会升级为轻量级锁，其他线程会通过自旋的形式尝试获取锁，不会阻塞，提高性能。
> 重量级锁是指当锁为轻量级锁的时候，另一个线程虽然是自旋，但自旋不会一直持续下去，当自旋一定次数的时候，还没有获取到锁，就会进入阻塞，该锁膨胀为重量级锁。重量级锁会让其他申请的线程进入阻塞，性能降低。

> 6. 自旋锁
>
> 自旋锁是指尝试获取锁的线程不会立即阻塞，而是采用循环的方式去尝试获取锁，这样的好处是减少线程上下文切换的消耗，缺点是循环会消耗CPU。

19. **乐观锁和悲观锁讲一下，哪些地方用到。**

> 乐观锁与悲观锁不是指具体的什么类型的锁，而是指**看待并发同步的角度。**
> 悲观锁对于同一个数据的并发操作，悲观锁采取加锁的形式。悲观的认为，不加锁的并发操作一定会出问题。
> 乐观锁在更新数据的时候，会采用尝试更新，不断重试的方式更新数据。乐观的认为，不加锁的并发操作是没有事情的。
>
> **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，数据库的for update SQL语句。Java中`synchronized`和`ReentrantLock`等独占锁就是悲观锁思想的实现。
>
> **乐观锁适用于多读的应用类型，这样可以提高吞吐量。**在Java中`java.util.concurrent.atomic`包下面的原子变量类就是使用了乐观锁的一种实现方式**CAS**实现的。
>
> **乐观锁适用于写比较少的情况下（多读场景）**，**一般多写的场景下用悲观锁就比较合适**，一般会经常产生冲突，这就会导致上层应用会不断的进行retry，这样反倒是降低了性能。

---

6. JVM

1. **讲一下 JVM 的内存结构**

> 堆，方法区(元空间)，程序计数器，虚拟机栈，本地方法栈。

```
堆设置
-Xms:初始堆大小
-Xmx:最大堆大小
-Xmn:新生代堆最大可用值
-XX:PermSize：表示非堆区初始内存分配大小，其缩写为permanent size（持久化内存）
-XX:MaxPermSize：表示对非堆区分配的内存的最大上限。
备注：在Java8中永久代的参数-XX:PermSize 和-XX：MaxPermSize已经失效。
```

2. **Minor gc 和 Full gc 的区别，详细介绍。**

> 针对 HotSpot VM 的实现，它里面的 GC 其实准确分类只有两大种：
>
> 部分收集 (Partial GC)：
>
> - 新生代收集（Minor GC / Young GC）：只对新生代进行垃圾收集；
> - 老年代收集（Major GC / Old GC）：只对老年代进行垃圾收集。需要注意的是 Major GC 在有的语境中也用于指代整堆收集；
> - 混合收集（Mixed GC）：对整个新生代和部分老年代进行垃圾收集。
>
> 整堆收集 (Full GC)：收集整个 Java 堆和方法区。
>
> minor GC、major GC和full GC。对新生代进行垃圾回收叫做minor GC，对老年代进行垃圾回收叫做major GC，同时对新生代、老年代和永久代进行垃圾回收叫做full GC，Full GC 一般消耗的时间比较长，远远大于 Minor GC，因此，有时候我们必须降低Full GC 发生的频率。
>
> 许多major GC是由minor GC触发的，所以很难将这两种垃圾回收区分开。Major GC通常是跟Full GC是等价的，收集整个GC堆。

> 触发条件：
>
> - Minor GC：新生代中的eden区域分配满了的时候触发。Minor GC后新生代中有部分存活对象会晋升到老年代中，所以老年代的占用量会提高。
> - Full GC：
>
> （1）**调用 System.gc()**
>
> （2）**老年代空间不足**：晋升进入老年代的对象大小大于老年代的可用内存，这个时候会触发Full GC。
>
> （3）**Metaspace区内存达到阈值**：从JDK8开始，永久代(PermGen)的概念被废弃掉了，取而代之的是一个称为Metaspace的存储空间。Metaspace使用的是本地内存，而不是堆内存，也就是说在默认情况下Metaspace的大小只与本地内存大小有关。
>
> （4）**统计得到的Minor GC晋升到老年代的平均大小大于老年代的剩余空间**：当准备触发一次Minor GC的时候，如果发现统计数据说之前Minor GC的平均晋升大小比目前老年代的空间大，则不会触发Minor GC，转而触发Major GC。
>
> （5）**老年代连续空间不足**：JVM如果判断老年代没有做足够的连续空间来放置大对象，那么就会引起Full GC。
>
> （6）**CMS GC时出现promotion failed和concurrent mode failure**：执行 CMS GC 的过程中同时有对象要放入老年代，而此时老年代空间不足（可能是 GC 过程中浮动垃圾过多导致暂时性的空间不足），便会报 Concurrent Mode Failure 错误，并触发 Full GC。（美团三面时问过，问的是浮动垃圾是什么？导致老年代空间不足会发生什么？还是要灵活应变，差点不记得了）

3. **方法区和永久代的关系?**

> 方法区也被称为永久代。**方法区和永久代的关系很像 Java 中接口和类的关系，类实现了接口，而永久代就是 HotSpot 虚拟机对虚拟机规范中方法区的一种实现方式。**方法区是 Java 虚拟机规范中的定义，是一种规范，而永久代是一种实现。

4. **JDK 1.8 HotSpot 的永久代为啥被彻底移除？有哪些常用参数?**

> JDK 1.8 的时候，方法区（HotSpot 的永久代）被彻底移除了（JDK1.7 就已经开始了），取而代之是元空间，元空间使用的是直接内存。
>
> ```
> -XX:MetaspaceSize=N //设置 Metaspace 的初始（和最小大小）
> -XX:MaxMetaspaceSize=N //设置 Metaspace 的最大大小
> ```
>
> **为什么要将永久代 (PermGen) 替换为元空间 (MetaSpace) 呢?**
>
> (1) 整个永久代有一个 JVM 本身设置固定大小上限，无法进行调整，而元空间使用的是直接内存，受本机可用内存的限制，虽然元空间仍旧可能溢出，但是比原来出现的几率会更小。
>
> 你可以使用 -XX：MaxMetaspaceSize 标志设置最大元空间大小，默认值为 unlimited，这意味着它只受系统内存的限制。-XX：MetaspaceSize 调整标志定义元空间的初始大小如果未指定此标志，则 Metaspace 将根据运行时的应用程序需求动态地重新调整大小。
>
> (2) 元空间里面存放的是类的元数据，这样加载多少类的元数据就不由 MaxPermSize 控制了, 而由系统的实际可用空间来控制，这样能加载的类就更多了。(永久代的调优是很困难的，虽然可以设置永久代的大小，但是很难确定一个合适的大小)

5. **主要进行 gc 的区域。永久代会发生 gc 吗？元空间呢?**

> 主要进行 gc 的区域是堆，就 HotSpot 虚拟机来说，永久代会发生 gc (full gc)，但是，元空间使用的是直接内存不会发生 gc。

6. **各种垃圾回收算法和回收器，说出自己的理解。**

> **垃圾回收算法**：标记-清除，标记-整理，复制，分代收集。
>
> **垃圾回收器**：新生代垃圾收集器：Serial，ParNew，Parallel Scanvenge。
>
> 老年代垃圾收集器：CMS，Serial Old，Parallel Old。
>
> 整堆收集器：G1；
>
> 除了 CMS 和 G1 之外，其它垃圾收集器都是以串行的方式执行。

> **Serial**：新生代，复制算法，单线程收集，必须暂停所有工作线程，直到完成。
>
> **ParNew**： ParNew垃圾收集器是Serial收集器的多线程版本。
>
> **Parallel Scavenge**：高吞吐量为目标，即减少垃圾收集时间，让用户代码获得更长的运行时间。
>
> **Serial Old**：Serial收集器的老年代版本，采用"标记-整理"算法。
>
> **Parallel Old**：Parallel Old垃圾收集器是Parallel Scavenge收集器的老年代版本。采用"标记-整理"算法。
>
> **CMS**：CMS 收集器是一种 **“标记-清除”算法**实现的，它的运作过程相比于前面几种垃圾收集器来说更加复杂一些。整个过程分为四个步骤：
>
> 1. **初始标记**：仅标记一下GC Roots能直接关联到的对象；2. **并发标记**：进行 GC Roots Tracing 的过程，也就是从GC Roots开始找到它能引用的所有其它对象。3. **重新标记**：修正并发标记期间因用户程序继续运作而导致标记变动的那一部分对象的标记记录。4. **并发清除**：回收所有的垃圾对象， 开启用户线程，GC 线程开始对**未标记**的区域做清扫。
> - 并发阶段占用cpu资源，拖慢用户程序，降低吞吐量，CMS默认启用（CPU + 3)/4个线程执行。
> - 无法处理浮动垃圾，并发清理阶段用户程序产生的垃圾，成为浮动垃圾，无法被当次处理。
> - 基于标记-清除算法的CMS，会使老年代产生很多空间碎片，不利于大对象的使用

> **G1**：G1 可以直接对新生代和老年代一起回收。G1 把堆划分成多个大小相等的独立区域，维护一个优先列表，每次根据允许的收集时间，优先回收价值最大的 区域。
>
> 1. 初始标记。2. 并发标记。3. 最终标记：为了修正在并发标记期间因用户程序继续运作而导致标记产生变动的那一部分标记记录。4. 筛选回收：首先对各个区域中的回收价值和成本进行排序，根据用户所期望的 GC 停顿时间来制定回收计划。

> G1收集器的设计目标是取代CMS收集器，它同CMS相比，在以下方面表现的更出色：
>
> - G1是一个有**整理内存**过程的垃圾收集器，**不会产生很多内存碎片。**
> - CMS采用的是标记清除垃圾回收算法，可能会产生不少的内存碎片。
> - G1的Stop The World(STW)更可控，G1在停顿时间上添加了**预测机制**，用户可以**指定期望停顿时间。**

7. **zgc？ zgc vs g1？ （我懵逼了~我只是听过有这个东西，完全没有去了解过）**

> ZGC适用于大内存低延迟服务的内存管理和回收。
>
> ZGC：JDK11 中加入的具有实验性质的低延迟垃圾收集器，目标是尽可能在不影响吞吐量的前提下，实现在任意堆内存大小都可以把停顿时间限制在 10ms 以内的低延迟。基于 Region 内存布局，不设分代，使用了读屏障、染色指针和内存多重映射等技术实现可并发的标记整理。ZGC 的 Region 具有动态性，是动态创建和销毁的，并且容量大小也是动态变化的。

8. **如何对进行性能分析？会用到哪些命令？**

> **top**：查看当前所有进程的使用情况，CPU占有率，内存使用情况，服务器负载状态等参数。
>
> **jps**：列出正在运行的虚拟机进程。
>
> **jstat**：可以用来监视虚拟机各种运行状态信息（堆和非堆的大小及其内存使用量）。
>
> **jstack**：生成虚拟机当前时刻的线程快照。
>
> **jinfo**：可以查看虚拟机的各项参数。
>
> **jmap：**：生成堆转储快照，查看内存占用情况。
>
> **jconsole**：一个java GUI监视工具，可以以图表化的形式显示各种数据。并可通过远程连接监视远程的服务器的jvm进程。

---

7. 数据库

1. **数据库三大范式**

> 第一范式：每个列都不可以再拆分。
>
> 第二范式：在第一范式的基础上，非主属性完全依赖于主键。
>
> 第三范式：在第二范式的基础上，非主属性只依赖于主键，不存在传递依赖。

2. **关系型数据库和非关系型数据库的区别？**

> 关系型数据库最典型的数据结构是表，由二维表及其之间的联系所组成的一个数据组织。
>
> 非关系型数据库严格上不是一种数据库，应该是一种数据结构化存储方法的集合，可以是文档或者键值对等。

3. **char和varchar的区别**

> 1. char是固定长度，varchar长度可变。varchar：**如果原先存储的位置无法满足其存储的需求**，就需要一些额外的操作，根据存储引擎的不同，有的会采用**拆分机制**，有的采用**分页机制。**
> 2. char和varchar的存储字节由**具体的字符集**来决定;
> 3. char是固定长度，长度不够的情况下，用空格代替。varchar表示的是实际长度的数据类型

4. **内连接、左连接和外连接?**

> 左外连接　以左表为主表，可以查询左表存在而右表为 null 的记录。
>
> 右外连接　以右表为主表，可以查询右表存在而左表为 null 的记录。
>
> 内连接　查询左右表同时满足条件的记录，两边都不可为 null。

5. **MySQL 有哪些聚合函数?**

> ① max 求最大值。② min 求最小值。③ count 统计数量。④ avg 求平均值。⑤ sum 求和。

6. **说一下 MVCC**

> MVCC就是多版本并发控制。MVCC解决的问题是读写互相不阻塞的问题，每次更新都产生一个新的版本，读的话可以读历史版本。MVCC 是一种并发控制的方法，一般在数据库管理系统中，实现对数据库的并发访问。MVCC是行级锁的一个变种，但是它在很多情况下避免了加锁操作，因此开销更低。虽然实现机制有所不同，但大都实现了非阻塞的读操作，写操作也只锁定必要的行。
>
> 在Mysql的InnoDB引擎中就是指在读已提交(READ COMMITTD)和可重复读(REPEATABLE READ)这两种隔离级别下的事务对于SELECT操作会访问版本链中的记录的过程。
>
> 这就使得别的事务可以修改这条记录，反正每次修改都会在版本链中记录。SELECT可以去版本链中拿记录，这就实现了读-写，写-读的并发执行，提升了系统的性能。 InnoDB只查找版本(DB_TRX_ID)早于当前事务版本的数据行。
>
> > **版本链**
> >
> > 在InnoDB引擎表中，它的聚簇索引记录中有两个必要的隐藏列：
> >
> > **trx_id**这个id用来存储的每次对某条聚簇索引记录进行修改的时候的事务id。
> >
> > **roll_pointer**每次**对哪条聚簇索引记录有修改的时候**，都会**把老版本写入undo log中**。这个roll_pointer就是存了一个指针，它**指向这条聚簇索引记录的上一个版本的位置**，通过它来获得上一个版本的记录信息。

7. **为什么要使用索引?**

> 1. 通过创建唯一性索引，可以保证数据库表中每一行数据的唯一性。
> 2. 可以大大加快 数据的检索速度（大大减少的检索的数据量）, 这也是创建索引的最主要的原因。
> 3. 帮助服务器避免排序和临时表。
> 4. 将随机IO变为顺序IO。
> 5. 可以加速表和表之间的连接，特别是在实现数据的参考完整性方面特别有意义。

8. **数据库原理相关补充**

> 局部性原理与磁盘预读 由于存储介质的特性，磁盘本身存取就比主存慢很多，再加上机械运动耗费，磁盘的存取速度往往是主存的几百分分之一，因此为了提高效率，要尽量减少磁盘I/O。为了达到这个目的，磁盘往往不是严格按需读取，而是每次都会预读，即使只需要一个字节，磁盘也会从这个位置开始，顺序向后读取一定长度的数据放入内存。这样做的理论依据是计算机科学中著名的局部性原理：当一个数据被用到时，其附近的数据也通常会马上被使用。程序运行期间所需要的数据通常比较集中。
>
> 由于磁盘顺序读取的效率很高（不需要寻道时间，只需很少的旋转时间），因此对于具有局部性的程序来说，预读可以提高I/O效率。
>
> 预读的长度一般为页（page）的整倍数。页是计算机管理存储器的逻辑块，硬件及操作系统往往将主存和磁盘存储区分割为连续的大小相等的块，每个存储块称为一页（在许多操作系统中，页的大小通常为4k），主存和磁盘以页为单位交换数据。当程序要读取的数据不在主存中时，会触发一个缺页异常，此时系统会向磁盘发出读盘信号，磁盘会找到数据的起始位置并向后连续读取一页或几页载入内存中，然后异常返回，程序继续运行。
>
> 先从B-Tree分析，根据B-Tree的定义，可知检索一次最多需要访问h个节点。数据库系统的设计者巧妙利用了磁盘预读原理，将一个节点的大小设为等于一个页，这样每个节点只需要一次I/O就可以完全载入。为了达到这个目的，在实际实现B-Tree还需要使用如下技巧：

> 每次新建节点时，直接申请一个页的空间，这样就保证一个节点物理上也存储在一个页里，加之计算机存储分配都是按页对齐的，就实现了一个node只需一次I/O。
>
> B-Tree中一次检索最多需要h-1次I/O（根节点常驻内存），渐进复杂度为O(h)=O(logdN)。一般实际应用中，出度d是非常大的数字，通常超过100，因此h非常小（通常不超过3）。
>
> 而红黑树这种结构，h明显要深的多。由于逻辑上很近的节点（父子）物理上可能很远，无法利用局部性，所以红黑树的I/O渐进复杂度也为O(h)，效率明显比B-Tree差很多。
>
> 综上所述，用B-Tree作为索引结构效率是非常高的。

9. **说一聚簇索引和非聚簇索引的有什么不同**?

> **1. 聚集索引即索引结构和数据一起存放的索引。主键索引属于聚集索引。**
>
> 优点：聚集索引的查询速度非常的快，因为整个B+树本身就是一颗多叉平衡树，叶子节点也都是有序的，定位到索引的节点，就相当于定位到了数据。
>
> 缺点：1. 依赖于有序的数据，不是有序的数据的话，插入或查找的速度肯定比较慢。2. 更新代价大。

> **2. 非聚集索引即索引结构和数据分开存放的索引。叶子节点存的是键值和数据所在物理地址**
>
> 优点：更新代价比聚集索引要小 。
>
> 缺点：1. 依赖于有序的数据，不是有序的数据的话，插入或查找的速度肯定比较慢。2. 可能会二次查询(回表)，当查到索引对应的指针或主键后，可能还需要根据指针或主键再到数据文件或表中查询。

10. **关于索引的各种轰炸。Mysql的索引，以及B+树与hash索引的区别，为什么不采用B树而采用B+树？B树和B+树的区别**

> B+树：非叶子节点不存储data，只存储索引，这样可以放更多的索引，data只存在叶子节点，这样到达叶子节点的路径查询长度都一样，使用b+树索引更加稳定。叶子节点用双向指针连接，提高区间访问的性能。B+ 树索引，底层是多路查询平衡树，节点是天然有序的（左节点小于服节点，右节点大于父节点），所以对于范围查找的时候不需要做全表扫描；
>
> hash索引：底层是哈希表，数据存储在哈希表中顺序是没有关联的，所以他不适合范围查找，如果要范围查找就需要全表扫描，他只适合全值扫描；简单的来说就是hash索引适合等值查找，不适合范围查找。

> MySQL索引使用的数据结构主要有**BTree索引** 和 **哈希索引** 。对于哈希索引来说，底层的数据结构就是哈希表，因此在绝大多数需求为单条记录查询的时候，可以选择哈希索引，查询性能最快；其余大部分场景，建议选择BTree索引。

> MySQL的BTree索引使用的是B树中的B+Tree，但对于主要的两种存储引擎的实现方式是不同的。
>
> - **MyISAM:** B+Tree叶节点的data域存放的是数据记录的地址。在索引检索的时候，首先按照B+Tree搜索算法搜索索引，如果指定的Key存在，则取出其 data 域的值，然后以 data 域的值为地址读取相应的数据记录。这被称为“非聚簇索引”。
> - **InnoDB:** 其数据文件本身就是索引文件。相比MyISAM，索引文件和数据文件是分离的，其表数据文件本身就是按B+Tree组织的一个索引结构，树的叶节点data域保存了完整的数据记录。这个索引的key是数据表的主键，因此InnoDB表数据文件本身就是主索引。这被称为“聚簇索引（或聚集索引）”。而其余的索引都作为辅助索引，辅助索引的data域存储相应记录主键的值而不是地址，这也是和MyISAM不同的地方。**在根据主索引搜索时，直接找到key所在的节点即可取出数据；在根据辅助索引查找时，则需要先取出主键的值，再走一遍主索引。 因此，在设计表的时候，不建议使用过长的字段作为主键，也不建议使用非单调的字段作为主键，这样会造成主索引频繁分裂。**

11. **红黑树和B+树的使用场景?**

> 红黑树和B树应用场景有何不同?
> 2者都是有序数据结构，可用作数据容器。红黑树多用在内部排序，即全放在内存中的。B树多用在内存里放不下，大部分数据存储在外存上时。因为B树层数少，因此可以确保每次操作，读取磁盘的次数尽可能的少。
> 在数据较小，可以完全放到内存中时，红黑树的时间复杂度比B树低。反之，数据量较大，外存中占主要部分时，B树因其读磁盘次数少，而具有更快的速度。

> 分析数据结构问题的时候，权衡三个因素: 查找速度，数据量，内存使用。
>
> B树(B+树)相对于平衡二叉树的不同是，每个节点包含的关键字增多了，特别是在B树应用到数据库中的时候，数据库充分利用了磁盘块的原理（磁盘数据存储是采用块的形式存储的，每个块的大小为4K，每次IO进行数据读取时，同一个磁盘块的数据可以一次性读取出来）把节点大小限制和充分使用在磁盘块大小范围；把树的节点关键字增多后树的层级比原来的二叉树少了，减少数据查找的次数和复杂度;

12. **B+比B树更适合实际应用中操作系统的文件索引和数据库索引?**

> 1) B+树的磁盘读写代价更低
>
> B+的内部结点并没有指向关键字具体信息的指针。因此其内部结点相对B树更小。如果把所有同一内部结点的关键字存放在同一盘块中，那么盘块所能容纳的关键字数量也越多。一次性读入内存中的需要查找的关键字也就越多。相对来说IO读写次数也就降低了。
>
> 2) B+tree的查询效率更加稳定
>
> 由于非终结点并不是最终指向文件内容的结点，而只是叶子结点中关键字的索引。所以任何关键字的查找必须走一条从根结点到叶子结点的路。所有关键字查询的路径长度相同，导致每一个数据的查询效率相当。
>
> 3）B树只适合随机检索，而B+树同时支持随机检索和顺序检索；
>
> 4）增删文件时，效率更高。因为B+树的叶子节点包含所有关键字，并以有序的链表结构存储，这样可很好提高增删效率。

> B+树的叶子节点有一条链相连，而B树的叶子节点各自独立。
>
> 由于B+树的内部节点只存放键，不存放值，因此，一次读取，可以在内存页中获取更多的键，有利于更快地缩小查找范围。B+树的叶节点由一条链相连，因此，当需要进行一次全数据遍历的时候，B+树只需要使用O(logN)时间找到最小的一个节点，然后通过链进行O(N)的顺序遍历即可。而B树则需要对树的每一层进行遍历，这会需要更多的内存置换次数，因此也就需要花费更多的时间

13. **非聚簇索引一定会回表查询吗?**

> 不一定，这涉及到查询语句所要求的字段是否全部命中了索引，如果全部命中了索引，那么就不必再进行回表查询。

14. **MySql的存储引擎，以及InnoDB和MyISAM的区别?**

> 存储引擎: InnoDB和MyISAM
>
> 1. **是否支持行级锁** : MyISAM 只有表级锁(table-level locking)，而InnoDB 支持行级锁(row-level locking)和表级锁，默认为行级锁。
> 2. **是否支持事务和崩溃后的安全恢复： MyISAM** 强调的是性能，每次查询具有原子性,其执行速度比InnoDB类型更快，但是不提供事务支持。但是**InnoDB** 提供事务支持事务，外部键等高级数据库功能。具有事务(commit)、回滚(rollback)和崩溃修复能力(crash recovery capabilities)的事务安全(transaction-safe (ACID compliant))型表。
> 3. **是否支持外键：** MyISAM不支持，而InnoDB支持。
> 4. **是否支持MVCC** ： 仅 InnoDB 支持。应对高并发事务，MVCC比单纯的加锁更高效；MVCC只在 `READ COMMITTED` 和 `REPEATABLE READ` 两个隔离级别下工作；MVCC可以使用 乐观锁和悲观锁来实现；各数据库中MVCC实现并不统一。

15. **Mysql联合索引的最左匹配原则。给出联合索引(a,b)，select *from table where a>0 and b>0，是否走索引，哪个走索引，哪个不走，以及从联合索引的底层结构去解释为什么？**（字段a走索引，字段b不走索引）

> 联合索引即由多列属性组成索引。
>
> 当B+树的数据项是复合的数据结构，比如(num,name,age)的时候，B+数是按照从左到右的顺序来建立搜索树的，B+树会**优先比较num来确定下一步的所搜方向**，如果num相同再依次比较name和age，最后得到检索的数据；
>
> 范围查询列可以使用索引（前提必须满足最左前缀），范围列后面的列无法使用索引。同时，索引最多作用于一个范围列。

16. **讲一下最左前缀原则?**

> 最左前缀原则是发生在复合索引上的，只有复合索引才会有所谓的左和右之分。当查询条件精确匹配左边连续一个或多个列时，索引可以被使用。
>
> 假设创建的联合索引由三个字段组成:
>
> ```
> ALTER TABLE table ADD INDEX index_name (num,name,age)
> ```
>
> 那么当查询的条件有为:num / (num AND name) / (num AND name AND age)时，索引才生效。所以在创建联合索引时，尽量把查询最频繁的那个字段作为最左(第一个)字段。查询的时候也尽量以这个字段为第一条件。

17. **left join，right join，inner join，full join之间的区别**

> 1. inner join，在两张表进行连接查询时，只保留两张表中完全匹配的结果集。
> 2. left join，在两张表进行连接查询时，会返回左表所有的行，即使在右表中没有匹配的记录。
> 3. right join，在两张表进行连接查询时，会返回右表所有的行，即使在左表中没有匹配的记录。
> 4. full join，在两张表进行连接查询时，返回左表和右表中所有没有匹配的行。
>
> 在 `join` 的过程中，其实就是从驱动表里面依次(注意理解这里面的依次)取出每一个值，然后去非驱动表里面进行匹配。

18. **为什么InnoDB表必须有主键，并且推荐使用整形的自增主键?**

> 因为叶子节点是按顺序排列的，如果是非自增的话，就会插入的时候频繁的分裂页（效率降低）
>
> 1、如果设置了主键，那么InnoDB会选择主键作为聚集索引、如果没有显式定义主键，则InnoDB会选择第一个不包含有NULL值的唯一索引作为主键索引、如果也没有这样的唯一索引，则InnoDB会选择内置6字节长的ROWID作为隐含的聚集索引(ROWID随着行记录的写入而主键递增)。
>
> 2、如果表使用自增主键
> 那么每次插入新的记录，记录就会顺序添加到当前索引节点的后续位置，主键的顺序按照数据记录的插入顺序排列，自动有序。当一页写满，就会自动开辟一个新的页
>
> 3、如果使用非自增主键（如果身份证号或学号等）
> 由于每次插入主键的值近似于随机，因此每次新纪录都要被插到现有索引页得中间某个位置，此时MySQL不得不为了将新记录插到合适位置而移动数据，甚至目标页面可能已经被回写到磁盘上而从缓存中清掉，此时又要从磁盘上读回来，这增加了很多开销，同时频繁的移动、分页操作造成了大量的碎片，得到了不够紧凑的索引结构，后续不得不通过OPTIMIZE TABLE来重建表并优化填充页面。

19. **间隙锁讲解一下**?

> for update是在数据库中上锁用的，可以为数据库中的行上一个排它锁。当一个事务的操作未完成时候，其他事务可以读取但是不能写入或更新。for update 仅适用于InnoDB，并且必须开启事务，在begin与commit之间才生效。
>
> InnoDB默认是行级别的锁，当有明确指定的主键时候（且主键存在），是行级锁。否则是表级别。
>
> ```
> SELECT * FROM foods WHERE id=1 FOR UPDATE;
> SELECT * FROM foods WHERE id=1 and name='咖啡色的羊驼' FOR UPDATE;
> ```

> 间隙锁的目的是为了防止幻读，其主要通过两个方面实现这个目的：
> （1）防止间隙内有新数据被插入
> （2）防止已存在的数据，更新成间隙内的数据
>
> innodb自动使用间隙锁的条件：
>
> （1）必须在可重复读级别下
> （2）检索条件必须有索引

> next-key锁其实包含了记录锁和间隙锁，即锁定一个范围，并且锁定记录本身，InnoDB默认加锁方式是next-key 锁。

20. **数据库问题，说一下从你打开命令行到发送请求，mysql服务器的整个相应流程? (当问到需要介绍数据库底层时可以这样回答)**

> - MySQL 主要分为 Server 层和引擎层。
> - Server 层主要包括连接器、查询缓存、分析器、优化器、执行器，同时还有一个日志模块（binlog），这个日志模块所有执行引擎都可以共用, redolog 只有 InnoDB 有。
> - 引擎层是插件式的，目前主要包括，MyISAM,InnoDB,Memory 等。

> - **连接器：** 身份认证和权限相关(登录 MySQL 的时候)。
> - **查询缓存:** 执行查询语句的时候，会先查询缓存（MySQL 8.0 版本后移除，因为这个功能不太实用）。
> - **分析器:** 没有命中缓存的话，SQL 语句就会经过分析器，分析器说白了就是要先看你的 SQL 语句要干嘛，再检查你的 SQL 语句语法是否正确。先词法分析，再语法分析
> - **优化器：** 按照 MySQL 认为最优的方案去执行。
> - **执行器:** 执行语句，然后从存储引擎返回数据。

> 查询语句：
>
> ```
> select * from tb_student  A where A.age='18' and A.name=' 张三 ';
> ```
>
> - 先检查该语句是否有权限，如果没有权限，直接返回错误信息，如果有权限，在 MySQL8.0 版本以前，会先查询缓存，以这条 sql 语句为 key 在内存中查询是否有结果，如果有直接缓存，如果没有，执行下一步。
> - 通过分析器进行词法分析，提取 sql 语句的关键元素，比如提取上面这个语句是查询 select，提取需要查询的表名为 tb_student,需要查询所有的列，查询条件是这个表的 id='1'。然后判断这个 sql 语句是否有语法错误，比如关键词是否正确等等，如果检查没问题就执行下一步。
> - 接下来就是优化器进行确定执行方案，上面的 sql 语句，可以有两种执行方案：
>
>   ```
>   a.先查询学生表中姓名为“张三”的学生，然后判断是否年龄是 18。
>   b.先找出学生中年龄 18 岁的学生，然后再查询姓名为“张三”的学生。
>   ```
>
>   那么优化器根据自己的优化算法进行选择执行效率最好的一个方案（优化器认为，有时候不一定最好）。那么确认了执行计划后就准备开始执行了。
> - 进行权限校验，如果没有权限就会返回错误信息，如果有权限就会调用数据库引擎接口，返回引擎的执行结果。

> 更新语句：
>
> ```
> update tb_student A set A.age='19' where A.name=' 张三 ';
> ```
>
> 我们来给张三修改下年龄，在实际数据库肯定不会设置年龄这个字段的，不然要被技术负责人打的。其实条语句也基本上会沿着上一个查询的流程走，只不过执行更新的时候肯定要记录日志啦，这就会引入日志模块了，MySQL 自带的日志模块式 **bin log（归档日志）** ，所有的存储引擎都可以使用，我们常用的 InnoDB 引擎还自带了一个日志模块 **redo log（重做日志）**，我们就以 InnoDB 模式下来探讨这个语句的执行流程。流程如下：
>
> - 先查询到张三这一条数据，如果有缓存，也是会用到缓存。
> - 然后拿到查询的语句，把 age 改为 19，然后调用引擎 API 接口，写入这一行数据，InnoDB 引擎把数据保存在内存中，同时记录 redo log，此时 redo log 进入 prepare 状态，然后告诉执行器，执行完成了，随时可以提交。
> - 执行器收到通知后记录 binlog，然后调用引擎接口，提交 redo log 为提交状态。
> - 更新完成。
>
> **这里肯定有同学会问，为什么要用两个日志模块，用一个日志模块不行吗?**

> 这是因为最开始 MySQL 并没与 InnoDB 引擎(InnoDB 引擎是其他公司以插件形式插入 MySQL 的)，MySQL 自带的引擎是 MyISAM，但是我们知道 redo log 是 InnoDB 引擎特有的，其他存储引擎都没有，这就导致会没有 crash-safe 的能力(crash-safe 的能力即使数据库发生异常重启，之前提交的记录都不会丢失)，binlog 日志只能用来归档。
>
> 并不是说只用一个日志模块不可以，只是 InnoDB 引擎就是通过 redo log 来支持事务的。那么，又会有同学问，我用两个日志模块，但是不要这么复杂行不行，为什么 redo log 要引入 prepare 预提交状态？这里我们用反证法来说明下为什么要这么做？
>
> - **先写 redo log 直接提交，然后写 binlog**，假设写完 redo log 后，机器挂了，binlog 日志没有被写入，那么机器重启后，这台机器会通过 redo log 恢复数据，但是这个时候 binlog 并没有记录该数据，后续进行机器备份的时候，就会丢失这一条数据，同时主从同步也会丢失这一条数据。
> - **先写 binlog，然后写 redo log**，假设写完了 bin log，机器异常重启了，由于没有 redo log，本机是无法恢复这一条记录的，但是 bin log 又有记录，那么和上面同样的道理，就会产生数据不一致的情况。
>
> 如果采用 redo log 两阶段提交的方式就不一样了，写完 binglog 后，然后再提交 redo log 就会防止出现上述的问题，从而保证了数据的一致性。那么问题来了，有没有一个极端的情况呢？假设 redo log 处于预提交状态，binglog 也已经写完了，这个时候发生了异常重启会怎么样呢？ 这个就要依赖于 MySQL 的处理机制了，MySQL 的处理过程如下：
>
> - 判断 redo log 是否完整，如果判断是完整的，就立即提交。
> - 如果 redo log 只是预提交但不是 commit 状态，这个时候就会去判断 binlog 是否完整，如果完整就提交 redo log, 不完整就回滚事务。
>
> 这样就解决了数据一致性的问题。

> - 查询语句的执行流程如下：权限校验（如果命中缓存）--> 查询缓存 --> 分析器 --> 优化器 --> 权限校验 --> 执行器 --> 引擎
> - 更新语句执行流程如下：分析器 --> 权限校验 --> 执行器 --> 引擎 -- redo log(prepare 状态) --> binlog --> redo log(commit状态)

21. **讲一下redo log，undo log，binlog?**

> redo Log：重做日志用来实现事务的**持久性**，用于记录事务操作的变化，记录的是**数据修改之后的值**。redo log由两部分组成：redo log buffer和redo log file。当事务提交(COMMIT)时，必须先将该事务的所有重做日志缓冲写入到重做日志文件进行持久化，才能COMMIT成功。MySQL宕机时，通过读取Redo Log中的数据，对数据库进行恢复。
>
> undo Log：回滚日志用来实现事务的回滚和多版本并发控制(MVCC)。Undo Log和Redo Log正好相反，记录的是数据**被修改前**的信息。undo log包括：insert undo log和update undo log。
>
> binlog：记录了对MySQL数据库执行更改的所有操作。

22. **MySQL 的数据如何恢复到任意时间点?**

> 恢复到任意时间点以定时的做全量备份，以及备份增量的 binlog 日志为前提。恢复到任意时间点首先将全量备份恢复之后，再此基础上回放增加的 binlog 直至指定的时间点。

23. **一张数据库表如果要删除大量的数据如何提高效率，如何做?**

> （1）抽取需要保留的数据到备份表中；
> （2）删除旧表数据；
> （3）备份表中的数据再插入旧表；

24. **Mysql如何保证一致性?**

> 通过预写式日志，undo log保证原子性，redo log保证持久性，设置隔离级别，保证并发事务进行的时候，保证数据一致性。恢复机制会将redo log中已提交的事务重做，保证事务的持久性；而undo log中未提交的事务进行回滚，保证事务的原子性。

> - 原子性：语句要么全执行，要么全不执行，是事务最核心的特性，事务本身就是以原子性来定义的；实现主要基于undo log

- 持久性：保证事务提交后不会因为宕机等原因导致数据丢失；实现主要基于redo log
- 隔离性：保证事务执行尽可能不受其他事务影响；InnoDB默认的隔离级别是RR，RR的实现主要基于锁机制、数据的隐藏列、undo log和next-key lock机制
- 一致性：事务追求的最终目标，一致性的实现既需要数据库层面的保障，也需要应用层面的保障

25. **数据库的主从复制?**

> MySQL主从复制的流程
>
> 1. 主库db的更新事件(update、insert、delete)被写到binlog
> 2. 主库创建一个dump thread，把binlog的内容发送到从库
> 3. 从库启动并发起连接，连接到主库
> 4. 从库启动之后，创建一个I/O线程，读取主库传过来的binlog内容并写入到relay log
> 5. 从库启动之后，创建一个SQL线程，从relay log里面读取内容，并解析成sql语句逐一执行

> **MySQL主从复制是一个异步的复制过程，主库发送更新事件到从库，从库读取更新记录，并执行更新记录，使得从库的内容与主库保持一致。**
>
> **主从同步过程中主服务器有一个工作线程dump thread，从服务器有两个工作线程I/O thread和SQL thread。主库把外界接收的SQL请求记录到自己的binlog日志中，从库的I/O thread去请求主库的binlog日志，并将binlog日志写到relay log(中继日志)中，然后从库重做中继日志的SQL语句。主库通过dump thread给从库I/O thread传送binlog日志。**

> binlog：binary log，主库中保存所有更新事件日志的二进制文件。binlog是数据库服务启动的一刻起，保存数据库所有变更记录（数据库结构和内容）的文件。在主库中，只要有更新事件出现，就会被依次地写入到binlog中，之后会推送到从库中作为从库进行复制的数据源。

26. **Mysq的事务隔离级别有哪几种，说一下可重复读解决了什么问题？还有什么问题没解决？幻读如何解决的？幻读的具体场景？说一下间隙锁如何实施的?**

> 并发事务带来哪些问题?
>
> - **脏读（Dirty read）:** 当一个事务正在访问数据并且对数据进行了修改，而这种修改还没有提交到数据库中，这时另外一个事务也访问了这个数据，然后使用了这个数据。因为这个数据是还没有提交的数据，那么另外一个事务读到的这个数据是“脏数据”，依据“脏数据”所做的操作可能是不正确的。
> - **丢失修改（Lost to modify）:** 指在一个事务读取一个数据时，另外一个事务也访问了该数据，那么在第一个事务中修改了这个数据后，第二个事务也修改了这个数据。这样第一个事务内的修改结果就被丢失，因此称为丢失修改。 例如：事务1读取某表中的数据A=20，事务2也读取A=20，事务1修改A=A-1，事务2也修改A=A-1，最终结果A=19，事务1的修改被丢失。
> - **不可重复读（Unrepeatableread）:** 指在一个事务内多次读同一数据。在这个事务还没有结束时，另一个事务也访问该数据。那么，在第一个事务中的两次读数据之间，由于第二个事务的修改导致第一个事务两次读取的数据可能不太一样。这就发生了在一个事务内两次读到的数据是不一样的情况，因此称为不可重复读。
>
>   (读取数据的事务将会禁止写事务(但允许读事务)，写事务则禁止任何其他事务。Mysql默认使用该隔离级别。这可以通过“共享读锁”和“排他写锁”实现，即事物需要对某些数据进行修改必须对这些数据加 X 锁，读数据时需要加上 S 锁，当数据读取完成并不立刻释放 S 锁，而是等到事物结束后再释放)
> - **幻读（Phantom read）:** 幻读与不可重复读类似。它发生在一个事务（T1）读取了几行数据，接着另一个并发事务（T2）插入了一些数据时。在随后的查询中，第一个事务（T1）就会发现多了一些原本不存在的记录，就好像发生了幻觉一样，所以称为幻读。

> **事务是逻辑上的一组操作，要么都执行，要么都不执行。**
>
> - **READ-UNCOMMITTED(读取未提交)：** 最低的隔离级别，允许读取尚未提交的数据变更，**可能会导致脏读、幻读或不可重复读。**
> - **READ-COMMITTED(读取已提交)：** 允许读取并发事务已经提交的数据，**可以阻止脏读，但是幻读或不可重复读仍有可能发生。**
> - **REPEATABLE-READ(可重复读)：** 对同一字段的多次读取结果都是一致的，除非数据是被本身事务自己所修改，**可以阻止脏读和不可重复读，但幻读仍有可能发生。**

- **SERIALIZABLE(可串行化)：** 最高的隔离级别，完全服从ACID的隔离级别。所有的事务依次逐个执行，这样事务之间就完全不可能产生干扰，也就是说，**该级别可以防止脏读、不可重复读以及幻读。**

MySQL InnoDB 存储引擎的默认支持的隔离级别是 **REPEATABLE-READ（可重读）**

InnoDB 存储引擎在 **REPEATABLE-READ（可重读）** 事务隔离级别下使用的是Next-Key Lock 锁算法，因此可以避免幻读的产生，这与其他数据库系统(如 SQL Server) 是不同的。所以说InnoDB 存储引擎的默认支持的隔离级别是 **REPEATABLE-READ（可重读）** 已经可以完全保证事务的隔离性要求，即达到了 SQL标准的 **SERIALIZABLE(可串行化)** 隔离级别。

InnoDB存储引擎的锁的算法有三种：

- Record lock：单个行记录上的锁
- Gap lock：间隙锁，锁定一个范围，不包括记录本身
- Next-key lock：record+gap 锁定一个范围，包含记录本身

27. **Mysql一条sql非常慢，如何进行分析?**

分以下两种情况来讨论。

1、大多数情况是正常的，只是偶尔会出现很慢的情况。

2、在数据量不变的情况下，这条SQL语句一直以来都执行的很慢。

第一种情况：偶尔

（1）**数据库在刷新脏页**（当内存数据页跟磁盘数据页内容不一致的时候，我们称这个内存页为“脏页”。）

当我们要往数据库插入一条数据、或者要更新一条数据的时候，我们知道数据库会在**内存**中把对应字段的数据更新了，但是更新之后，这些更新的字段并不会马上同步持久化到**磁盘**中去，而是把这些更新的记录写入到 redo log 日记中去，等到空闲的时候，在通过 redo log 里的日记把最新的数据同步到**磁盘**中去。（**如果redo log写满了，这个时候就没办法等到空闲的时候再把数据同步到磁盘的，只能暂停其他操作，全身心来把数据同步到磁盘中去的，而这个时候，**就会导致我们平时正常的SQL语句突然执行的很慢）

（2）**拿不到锁，所要执行的语句涉及到了别人对表或行加的锁**

第二种情况：一直

（1）**所要查询的字段没有索引（全表扫描）**

（2）**字段有索引，但却没有用索引，由于对字段进行运算、函数操作导致无法用索引。**

```
select * from t where c - 1 = 1000;
```

```
select * from t where pow(c,2) = 1000;
```

（4）**数据库选错了索引（主要在回表查询的时候，二次查询可能导致，由于统计的失误，导致系统没有走索引，而是走了全表扫描**）

---

8. Redis

1. **Redis介绍**

Redis 就是一个内存数据库。**Redis 除了做缓存之外，Redis 也经常用来做分布式锁，甚至是消息队列。**

Redis 提供了多种数据类型来支持不同的业务场景。Redis 还支持事务 、持久化、Lua 脚本、多种集群方案。

- 完全基于内存，绝大部分请求是纯粹的内存操作，非常快速。

- 数据结构简单，对数据操作也简单，**Redis**中的数据结构是专门进行设计的；
- 采用单线程，避免了不必要的上下文切换和竞争条件，也不存在多进程或者多线程导致的切换而消耗**CPU**，不用去考虑各种锁的问题，不存在加锁释放锁操作，没有因为可能出现死锁而导致的性能消耗；
- 使用多路I/O复用模型，非阻塞IO；

2. **Redis的原子性?**

> Redis所有单个命令的执行都是原子性的，这与它的单线程机制有关；
> **对Redis来说，执行get、set以及eval等API，都是一个一个的任务，这些任务都会由Redis的线程去负责执行，任务要么执行成功，要么执行失败，这就是Redis的命令是原子性的原因。**

3. **为什么要用缓存(Redis)?**

> **高性能**
>
> 假如用户第一次访问数据库中的某些数据的话，这个过程是比较慢，毕竟是从硬盘中读取的。但是，如果说，用户访问的数据属于高频数据并且不会经常改变的话，那么我们就可以很放心地将该用户访问的数据存在缓存中。
>
> **这样有什么好处呢?** 那就是保证用户下一次再访问这些数据的时候就可以直接从缓存中获取了。操作缓存就是直接操作内存，所以速度相当快。
>
> 不过，要保持数据库和缓存中的数据的一致性。 如果数据库中的对应数据改变的之后，同步改变缓存中相应的数据即可!

> **高并发**
>
> 直接操作缓存能够承受的数据库请求数量是远远大于直接访问数据库的，所以我们可以考虑把数据库中的部分数据转移到缓存中去，这样用户的一部分请求会直接到缓存这里而不用经过数据库。进而，我们也就提高的系统整体的并发。

4. **Redis的单线程理解**

> 其中执行命令阶段，由于Redis是单线程来处理命令的，所有到达服务端的命令都不会立刻执行，所有的命令都会进入一个队列中，然后逐个执行，并且多个客户端发送的命令的执行顺序是不确定的，但是可以确定的是不会有两条命令被同时执行，不会产生并发问题，这就是Redis的单线程基本模型。
>
> **Redis基于Reactor模式开发了自己的网络事件处理模型——文件事件处理器，由于文件事件处理器是单线程方式运行的，所以我们一般都说 Redis 是单线程模型。文件事件处理器使用I/O多路复用程序来同时监听多个套接字，并根据套接字目前执行的任务来为套接字关联不同的事件处理器。**
>
> I/O 多路复用技术的使用让 Redis 不需要额外创建多余的线程来监听客户端的大量连接，降低了资源的消耗。

> 文件事件处理器（file event handler）主要是包含 4 个部分：
>
> - 多个 socket（客户端连接）
> - IO 多路复用程序（支持多个客户端连接的关键）
> - 文件事件分派器（将 socket 关联到相应的事件处理器）
> - 事件处理器（连接应答处理器、命令请求处理器、命令回复处理器）

> Reactor模式
>
> Reactor模式(反应器模式)是一种处理一个或多个客户端并发交付服务请求的事件设计模式。当请求抵达后，服务处理程序使用I/O多路复用策略，然后同步地派发这些请求至相关的请求处理程序。

> Redis是单线程，快的原因：
>
> 1）完全基于内存，绝大部分请求是纯粹的内存操作，非常快速。数据存在内存中，类似于 HashMap，HashMap 的优势就是查找和操作的时间复杂度都是O(1)；
>
> 2）数据结构简单，对数据操作也简单，Redis 中的数据结构是专门进行设计的；
>
> 3）采用单线程，避免了不必要的上下文切换和竞争条件，也不存在多进程或者多线程导致的切换而消耗CPU，不用去考虑各种锁的问题，不存在加锁释放锁操作，没有因为可能出现死锁而导致的性能消耗；

4）使用多路 I/O 复用模型，非阻塞 IO；

5）使用底层模型不同，它们之间底层实现方式以及与客户端之间通信的应用协议不一样，Redis 直接自己构建了 VM 机制 ，因为一般的系统调用系统函数的话，会浪费一定的时间去移动和请求；

5. **Redis主从复制**

1、redis的复制功能是支持多个数据库之间的数据同步。一类是主数据库（master）一类是从数据库（slave），主数据库可以进行读写操作，当发生写操作的时候自动将数据同步到从数据库，而从数据库一般是只读的，并接收主数据库同步过来的数据，一个主数据库可以有多个从数据库，而一个从数据库只能有一个主数据库。

2、通过redis的复制功能可以很好的实现数据库的读写分离，提高服务器的负载能力。主数据库主要进行写操作，而从数据库负责读操作。

过程：（同步和命令传播）

1：当一个从数据库启动时，会向主数据库发送sync命令。

2：主数据库接收到sync命令后会开始一个线程并在后台保存快照（执行rdb操作），并将保存期间接收到的命令缓存起来。

3：当快照完成后，redis会将快照文件和所有缓存的命令发送给从数据库。

4：从数据库收到后，会载入快照文件并执行收到的缓存的命令。

Redis2.8之后支持完整重同步和部分重同步。

SYNC 命令

每次执行 SYNC 命令，主从服务器需要执行如下动作：

1. **主服务器** 需要执行 BGSAVE 命令来生成 RDB 文件，这个生成操作会 **消耗** 主服务器大量的 **CPU、内存和磁盘 I/O 的资源**；
2. **主服务器** 需要将自己生成的 RDB 文件 发送给从服务器，这个发送操作会 **消耗** 主服务器 **大量的网络资源**，并对主服务器响应命令请求的时间产生影响；
3. 接收到 RDB 文件的 **从服务器** 需要载入主服务器发来的 RBD 文件，并且在载入期间，从服务器 **会因为阻塞而没办法处理命令请求**；

特别是当出现 **断线重复制** 的情况是时，为了让从服务器补足断线时确实的那一小部分数据，却要执行一次如此耗资源的 SYNC 命令，显然是不合理的。

PSYNC 命令的引入

所以在 **Redis 2.8** 中引入了 PSYNC 命令来代替 SYNC，它具有两种模式：

1. **全量复制：** 用于初次复制或其他无法进行部分复制的情况，将主节点中的所有数据都发送给从节点，是一个非常重型的操作；
2. **部分复制：** 用于网络中断等情况后的复制，只将 **中断期间主节点执行的写命令** 发送给从节点，与全量复制相比更加高效。**需要注意** 的是，如果网络中断时间过长，导致主节点没有能够完整地保存中断期间执行的写命令，则无法进行部分复制，仍使用全量复制；

部分复制的原理主要是靠主从节点分别维护一个 **复制偏移量**，有了这个偏移量之后断线重连之后一比较，之后就可以仅仅把从服务器断线之后确实的这部分数据给补回来了。

6. **Redis如何保证高可用?**

Redis保证高可用主要通过哨兵模式进行实现。他提供了对master的监控和故障转移，当master节点出现故障后，可以自动通过选举选出一台slave做master，且哨兵之间也可以做集群部署，相互监测。防止单个哨兵死掉的情况。

7. **redis的5种数据结构，redis的zset底层用的什么数据结构？跳表。介绍一下、画一下基本结构，搜索插入数据过程，时间复杂度。**

Redis 有 5 种基础数据结构，它们分别是：**string(字符串)**、**list(列表)**、**hash(字典)**、**set(集合)** 和 **zset(有序集合)**。

类型	存储值	应用场景
String	字符串、整数、浮点数	简单的键值对缓存
List	列表	存储列表型数据结构，例如：评论列表、商品列表
Set	无序集合	适合交集、并集、查集操作，例如朋友关系
Zset	有序集合	去重后排序，适合排名场景
Hash	哈希	结构化数据，比如存储对象

string

Redis 底层对于字符串的定义SDS实现

分别需要记录已使用字节的长度len，记录当前字节数组总共分配的字节数量alloc，字节数组

优点：

1. **获取字符串长度为 O(1) 级别的操作**
2. 杜绝 **缓冲区溢出/内存泄漏** 的问题
3. 保证二进制安全

hash

Redis 中的字典相当于 Java 中的 **HashMap**，内部实现也差不多类似，都是通过 **"数组 + 链表"** 的链地址法来解决部分 **哈希冲突**，同时这样的结构也吸收了两种不同数据结构的优点。

实际上字典结构的内部包含两个 hashtable，通常情况下只有一个 hashtable 是有值的，但是在字典扩容缩容时，需要分配新的 hashtable，然后进行 **渐进式搬迁**

渐进式 rehash 会在 rehash 的同时，保留新旧两个 hash 结构，如上图所示，查询时会同时查询两个 hash 结构，然后在后续的定时任务以及 hash 操作指令中，循序渐进的把旧字典的内容迁移到新字典中。当搬迁完成了，就会使用新的 hash 结构取而代之。

zset

zset的编码有**ziplist（压缩链表）**和**skiplist（跳表）**两种。

内部编码：① ziplist（key <= 128 且 member <= 64B）。② skiplist（key > 128 或 member > 64B）。

ziplist（压缩链表）

当zset满足以下两个条件的时候，使用ziplist：

> 1. 保存的元素少于128个
> 2. 保存的所有元素大小都小于64字节

ziplist 是由一系列特殊编码的**连续内存块组成的顺序型数据结构。压缩列表节点组成**：previous_entry_length 表示前一个节点的长度，如果长度能够使用1个字节保存，则就使用一个字节保存，否则使用5个字节保存（第一个字节会被填充为全1）； encoding表示当前节点数据的类型和长度。content表示节点数据，可以是一个字节数组或者整数。

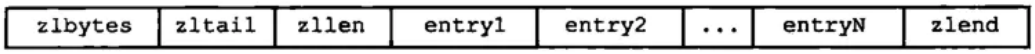


图 7-1　压缩列表的各个组成部分

表 7-1　压缩列表各个组成部分的详细说明

属性	类型	长度	用　途
zlbytes	uint32_t	4 字节	记录整个压缩列表占用的内存字节数：在对压缩列表进行内存重分配，或者计算 zlend 的位置时使用
zltail	uint32_t	4 字节	记录压缩列表表尾节点距离压缩列表的起始地址有多少字节：通过这个偏移量，程序无须遍历整个压缩列表就可以确定表尾节点的地址
zllen	uint16_t	2 字节	记录了压缩列表包含的节点数量：当这个属性的值小于 UINT16_MAX（65535）时，这个属性的值就是压缩列表包含节点的数量；当这个值等于 UINT16_MAX 时，节点的真实数量需要遍历整个压缩列表才能计算得出
entryX	列表节点	不定	压缩列表包含的各个节点，节点的长度由节点保存的内容决定
zlend	uint8_t	1 字节	特殊值 0xFF（十进制 255），用于标记压缩列表的末端

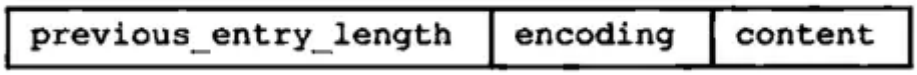


图 7-4　压缩列表节点的各个组成部分

skiplist（跳表）

```
typedef struct zset{
     //跳跃表
     zskiplist *zsl;
     //字典
     dict *dice;
} zset;
```

skiplist编码的有序集合底层是一个命名为zset的结构体，而一个zset结构同时包含一个字典和一个跳跃表。跳跃表按score从小到大保存所有集合元素。而字典则保存着从member到score的映射，这样就可以用O(1)的复杂度来查找member对应的score值。虽然同时使用两种结构，但它们会通过指针来共享相同元素的member和score，因此不会浪费额外的内存。

假如我们单独使用字典，虽然能以 O(1) 的时间复杂度查找成员的分值，但是因为字典是以无序的方式来保存集合元素，所以每次进行范围操作的时候都要进行排序；假如我们单独使用跳跃表来实现，虽然能执行范围操作，但是查找操作有 O(1)的复杂度变为了O(logN)。因此Redis使用了两种数据结构来共同实现有序集合。

跳表的产生就是为了解决链表过长的问题，通过增加链表的多级索引来加快原始链表的查询效率。这样的方式可以让查询的时间复杂度从O(n)提升至O(logn)。**最低层的链表维护了跳表内所有的元素，每上面一层链表都是下面一层的子集。**

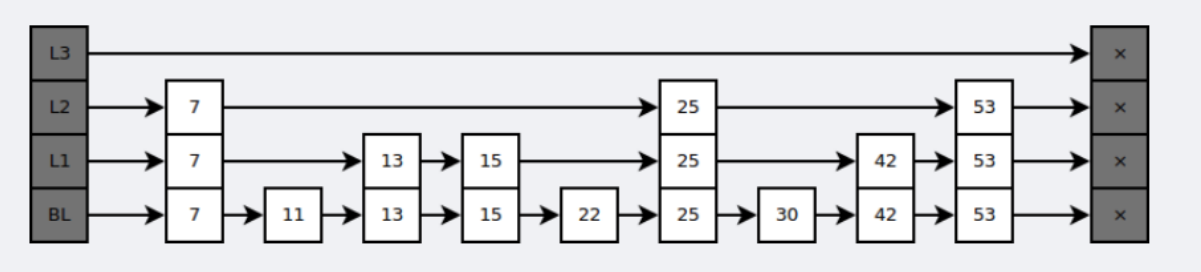
跳表内的所有链表的元素都是排序的。查找时，可以从顶级链表开始找。一旦发现被查找的元素大于当前链表中的取值，就会转入下一层链表继续找。这也就是说在查找过程中，搜索是跳跃式的。

- backward字段是指向链表前一个节点的指针。只有第1层链表是一个双向链表。
- level[]存放指向各层链表后一个节点的指针。

```
/* ZSETs use a specialized version of Skiplists */
typedef struct zskiplistNode {
    // value
    sds ele;
    // 分值
    double score;
    // 后退指针
```

```
    struct zskiplistNode *backward;
    // 层
    struct zskiplistLevel {
        // 前进指针
        struct zskiplistNode *forward;
        // 跨度
        unsigned long span;
    } level[];
} zskiplistNode;

typedef struct zskiplist {
    // 跳跃表头指针
    struct zskiplistNode *header, *tail;
    // 表中节点的数量
    unsigned long length;
    // 表中层数最大的节点的层数
    int level;
} zskiplist;
```



8. **redis为什么使用跳表而不是用红黑树?**

1. **在做范围查找的时候，平衡树比skiplist操作要复杂。**在平衡树上，我们找到指定范围的小值之后，还需要以中序遍历的顺序继续寻找其它不超过大值的节点。如果不对平衡树进行一定的改造，这里的中序遍历并不容易实现。而在skiplist上进行范围查找就非常简单，只需要在找到小值之后，对第1层链表进行若干步的遍历就可以实现。
2. **平衡树的插入和删除操作可能引发子树的调整，逻辑复杂**，而skiplist的插入和删除只需要修改相邻节点的指针，操作简单又快速。
3. **从内存占用上来说，skiplist比平衡树更灵活一些。**一般来说，平衡树每个节点包含2个指针（分别指向左右子树），而skiplist每个节点包含的指针数目平均为1/(1-p)，具体取决于参数p的大小。如果像Redis里的实现一样，取p=1/4，那么平均每个节点包含1.33个指针，比平衡树更有优势。
4. 查找单个key，skiplist和平衡树的时间复杂度都为O(log n)，大体相当；而哈希表在保持较低的哈希值冲突概率的前提下，查找时间复杂度接近O(1)，性能更高一些。所以我们平常使用的各种Map或dictionary结构，大都是基于哈希表实现的。
5. 从算法实现难度上来比较，skiplist比平衡树要简单得多。

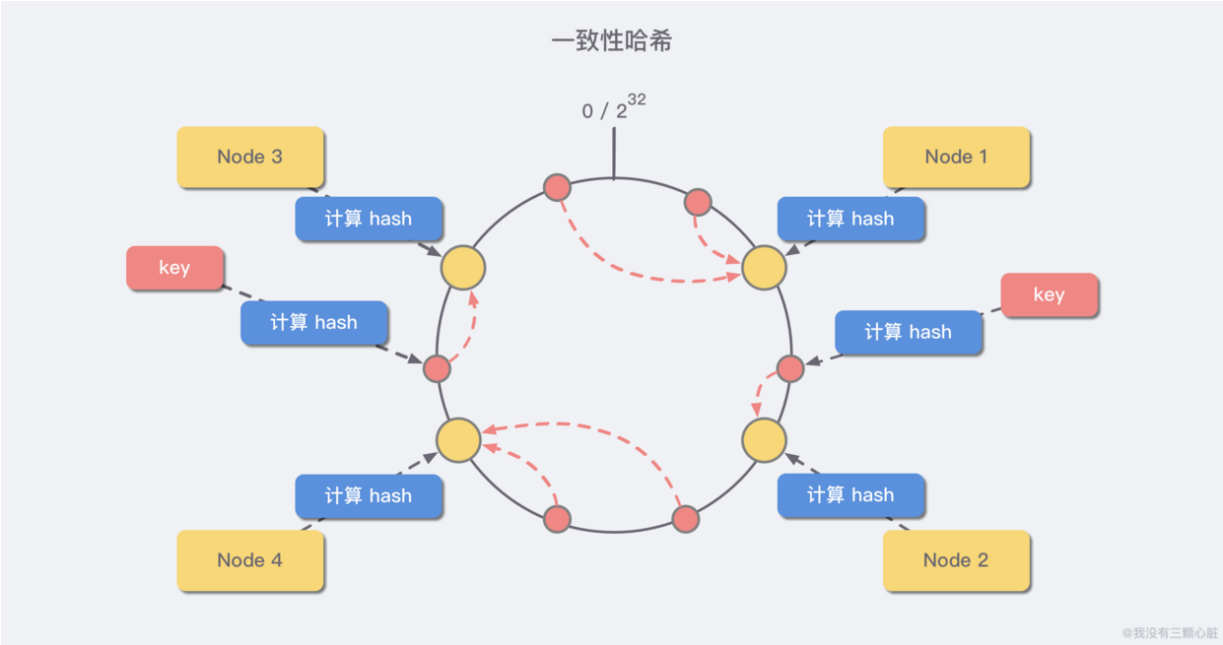
9. **Redis集群数据分区方案?**

（1）哈希取余分区。（2）一致性哈希分区。（3）带虚拟节点的一致性哈希分区。

（1）哈希取余分区思路非常简单：计算 key 的 hash 值，然后对节点数量进行取余，从而决定数据映射到哪个节点上。

不过该方案最大的问题是，**当新增或删减节点时**，节点数量发生变化，系统中所有的数据都需要 **重新计算映射关系**，引发大规模数据迁移。

（2）一致性哈希算法将 **整个哈希值空间** 组织成一个虚拟的圆环，范围是 *[0 , 2^32-1]*，对于每一个数据，根据 key 计算 hash 值，确数据在环上的位置，然后从此位置沿顺时针行走，找到的第一台服务器就是其应该映射到的服务器：



与哈希取余分区相比，一致性哈希分区将 **增减节点的影响限制在相邻节点**。以上图为例，如果在 node1 和 node2 之间增加 node5，则只有 node2 中的一部分数据会迁移到 node5；如果去掉 node2，则原 node2 中的数据只会迁移到 node4 中，只有 node4 会受影响。

一致性哈希分区的主要问题在于，当 **节点数量较少** 时，增加或删减节点，**对单个节点的影响可能很大**，造成数据的严重不平衡。还是以上图为例，如果去掉 node2，node4 中的数据由总数据的 1/4 左右变为 1/2 左右，与其他节点相比负载过高。

（3）带虚拟节点的一致性哈希分区：该方案在 **一致性哈希分区的基础上**，引入了 **虚拟节点** 的概念。为了避免出现数据倾斜问题，一致性 Hash 算法引入了虚拟节点的机制，也就是每个机器节点会进行多次哈希，最终每个机器节点在哈希环上会有多个虚拟节点存在，使用这种方式来大大削弱甚至避免数据倾斜问题。这样就解决了服务节点少时数据倾斜的问题。

Redis 集群使用的便是该方案，其中的虚拟节点称为 **槽（slot）**。槽是介于数据和实际节点之间的虚拟概念，每个实际节点包含一定数量的槽，每个槽包含哈希值在一定范围内的数据。

在使用了槽的一致性哈希分区中，**槽是数据管理和迁移的基本单位**。槽 **解耦** 了 **数据和实际节点** 之间的关系，增加或删除节点对系统的影响很小。仍以上图为例，系统中有 4 个实际节点，假设为其分配 16 个槽(0-15)；

- 槽 0-3 位于 node1；4-7 位于 node2；以此类推....

如果此时删除 node2，只需要将槽 4-7 重新分配即可，例如槽 4-5 分配给 node1，槽 6 分配给 node3，槽 7 分配给 node4；可以看出删除 node2 后，数据在其他节点的分布仍然较为均衡。

10. **为什么redis集群的最大槽数是16384个?**

Redis 集群并没有使用一致性hash，而是引入了哈希槽的概念。Redis 集群有16384个哈希槽，每个key通过CRC16校验后对16384取模来决定放置哪个槽，集群的每个节点负责一部分hash槽。Redis 集群包含了 16384 个哈希槽，每个 Key 经过计算后会落在一个具体的槽位上，而槽位具体在哪个机器上是用户自己根据自己机器的情况配置的，机器硬盘小的可以分配少一点槽位，硬盘大的可以分配多一点。如果节点硬盘都差不多则可以平均分配。所以哈希槽这种概念很好地解决了一致性哈希的弊端。

在redis节点发送心跳包时需要把所有的槽放到这个心跳包里，以便让节点知道当前集群信息，16384=16k，在发送心跳包时使用char进行bitmap压缩后是2k（2 * 8 (8 bit) * 1024(1k) = 2K），也就是说使用2k的空间创建了16k的槽数。

虽然使用CRC16算法最多可以分配65535（2^16-1）个槽位，65535=65k，压缩后就是8k（8 * 8 (8 bit) * 1024(1k) = 8K），也就是说需要需要8k的心跳包，作者认为这样做不太值得；并且一般情况下一个redis集群不会有超过1000个master节点，所以16k的槽位是个比较合适的选择。

11. **缓存雪崩，缓存穿透，如何解决?**

缓存穿透：大量请求的 key 根本不存在于缓存中，导致请求直接到了数据库上，根本没有经过缓存这一层。

> 解决办法:
>
> 1. **缓存无效 key**：如果缓存和数据库都查不到某个 key 的数据就写一个到 Redis 中去并设置过期时间。
> 2. **布隆过滤器**：对请求进行过滤。可以非常方便地判断一个给定数据是否存在于海量数据中。
>   1. 使用布隆过滤器中的哈希函数对元素值进行计算，得到哈希值（有几个哈希函数得到几个哈希值）。
>   2. 根据得到的哈希值，在位数组中把对应下标的值置为 1。

> **缓存雪崩**：缓存在同一时间大面积的失效，后面的请求都直接落到了数据库上，造成数据库短时间内承受大量请求。
>
> 解决办法:
>
> **针对 Redis 服务不可用的情况（如果缓存层因为某些问题不能提供服务，所有请求都会到达存储层，对数据库造成巨大压力）：**
>
> 1. 采用 Redis 集群，避免单机出现问题整个缓存服务都没办法使用。
> 2. 限流，避免同时处理大量的请求。
> 3. 构建多级缓存，增加本地缓存，降低请求直达存储层概率。
>
> **针对热点缓存失效的情况（对于热数据的访问量非常大，在其缓存失效的瞬间，大量请求直达存储层，导致服务崩溃）：**
>
> 1. 合理设置缓存过期时间来实现。
> 2. 缓存永不失效。

12. **如何保证缓存与数据库双写时的数据一致性?**

> 一般情况下我们都是这样使用缓存的：先读缓存，缓存没有的话，就读数据库，然后取出数据后放入缓存，同时返回响应。这种方式很明显会存在缓存和数据库的数据不一致的情况。
>
> 你只要用缓存，就可能会涉及到缓存与数据库双存储双写，你只要是双写，就一定会有数据一致性的问题，那么你如何解决一致性问题?
>
> 一般来说，如果允许缓存可以稍微的跟数据库偶尔有不一致的情况，也就是说如果你的系统**不是严格要求** “缓存+数据库” 必须保持一致性的话，最好不要做这个方案，即：**读请求和写请求串行化**，串到一个**内存队列**里去。
>
> 串行化可以保证一定不会出现不一致的情况，但是它也会导致系统的吞吐量大幅度降低，用比正常情况下多几倍的机器去支撑线上的一个请求。把一些列的操作都放到队列里面，顺序肯定不会乱，但是并发高了，这队列很容易阻塞，反而会成为整个系统的弱点，瓶颈。

> 最经典的缓存+数据库读写的模式
>
> - 读的时候，先读缓存，缓存没有的话，就读数据库，然后取出数据后放入缓存，同时返回响应。
> - 更新的时候，**先更新数据库，然后再删除缓存。**
>
> 这里是懒加载的思想，需要被使用的时候再重新计算。

13. **如何解决 Redis 的并发竞争 Key 问题?**

> Redis 的并发竞争 Key 的问题也就是多个系统同时对一个 key 进行操作，但是最后执行的顺序和我们期望的顺序不同，这样也就导致了结果的不同!
>
> 推荐一种方案：分布式锁（zookeeper 和 Redis 都可以实现分布式锁）。（如果不存在 Redis 的并发竞争 Key 问题，不要使用分布式锁，这样会影响性能）
>
> 基于 zookeeper 临时有序节点可以实现的分布式锁。大致思想为：每个客户端对某个方法加锁时，在 zookeeper 上的与该方法对应的指定节点的目录下，生成一个唯一的瞬时有序节点。 判断是否获取锁的方式很简单，只需要判断有序节点中序号最小的一个。 当释放锁的时候，只需将这个瞬时节点删除即可。同时，其可以避免服务宕机导致的锁无法释放，而产生的死锁问题。完成业务流程后，删除对应的子节点释放锁。
>
> 在实践中，当然是从以可靠性为主。所以首推 Zookeeper。

---

9. 网络

1. **为什么网络要分层?**

> 说到分层，我们先从我们平时使用框架开发一个后台程序来说，我们往往会按照每一层做不同的事情的原则将系统分为 三层（复杂的系统分层可能会更多）：
>
> 1. Repository（数据库操作）
> 2. Service（业务操作）
> 3. Controller（数据交互）
>
> **网络分层的原则：每一层独立于其它层完成自己的工作，而不需要相互依赖，上下层之间通过标准结构来互相通信，简单易用又具有拓展性。**
>
> **复杂的系统需要分层，因为每一层都需要专注于一类事情。我们的网络分层的原因也是一样，每一层只专注于做一类事情。**
>
> **为什么计算机网络要分层呢？** ,我们再来较为系统的说一说：
>
> 1. **各层之间相互独立**：各层之间相互独立，各层之间不需要关心其他层是如何实现的，只需要知道自己如何调用下层提供好的功能就可以了（可以简单理解为接口调用）。**这个和我们对开发时系统进行分层是一个道理。**
> 2. **提高了整体灵活性**： 每一层都可以使用最适合的技术来实现，你只需要保证你提供的功能以及暴露的接口的规则没有改变就行了。**这个和我们平时开发系统的时候要求的高内聚、低耦合的原则也是可以对应上的。**
> 3. **大问题化小**： 分层可以将复杂的网络间题分解为许多比较小的、界线比较清晰简单的小问题来处理和解决。这样使得复杂的计算机网络系统变得易于设计，实现和标准化。**这个和我们平时开发的时候，一般会将系统功能分解，然后将复杂的问题分解为容易理解的更小的问题是相对应的，这些较小的问题具有更好的边界（目标和接口）定义。**

2. **TCP/IP 4 层模型了解么?**

> TCP/IP 4 层模型:
>
> 1. 应用层
> 2. 传输层
> 3. 网络层
> 4. 网络接口层

3. **HTTP 是哪一层的协议？http常见的状态码。**

> HTTP 协议 属于应用层的协议。
>
> **HTTP 协议是基于 TCP 协议的**，发送 HTTP 请求之前首先要建立 TCP 连接也就是要经历 3 次握手。目前使用的 HTTP 协议大部分都是 1.1。在 1.1 的协议里面，默认是开启了 Keep-Alive 的，这样的话建立的连接就可以在多次请求中被复用了。
>
> 另外， **HTTP 协议是"无状态"的协议，它无法记录客户端用户的状态** 一般我们都是通过 Session 来记录客户端用户的状态。

	类别	原因短语
1XX	Informational（信息性状态码）	接收的请求正在处理
2XX	Success（成功状态码）	请求正常处理完毕
3XX	Redirection（重定向状态码）	需要进行附加操作以完成请求
4XX	Client Error（客户端错误状态码）	服务器无法处理请求
5XX	Server Error（服务器错误状态码）	服务器处理请求出错

状态码	简要说明
200	客户端请求成功
301	请求的网页已永久移动到新位置，后应使用新位置
302	临时性重定向，请求的资源临时分配了新位置，本次请求暂且使用新位置
304	自从上次请求后，请求的网页未修改过。服务器返回此响应时，不会返回网页内容。
403	请求被服务器拒绝
404	服务器无法找到请求的URL

4. **HTTP 和 HTTPS 什么区别**?

> 1. **端口** ： HTTP的URL由“http://”起始且默认使用端口80，而HTTPS的URL由“https://”起始且默认使用端口443。
> 2. 安全性和资源消耗：
>
>   HTTP协议运行在TCP之上，所有传输的内容都是明文，客户端和服务器端都无法验证对方的身份。HTTPS是运行在SSL之上的HTTP协议，SSL运行在TCP之上。所有传输的内容都经过加密，加密采用对称加密，但对称加密的密钥用服务器方的证书进行了非对称加密。所以说，HTTP 安全性没有 HTTPS 高，但是 HTTPS 比HTTP耗费更多服务器资源。
>
>   - 对称加密：密钥只有一个，加密解密为同一个密码，且加解密速度快，典型的对称加密算法有DES、AES等；
>   - 非对称加密：密钥成对出现（且根据公钥无法推知私钥，根据私钥也无法推知公钥），加密解密使用不同密钥（公钥加密需要私钥解密，私钥加密需要公钥解密），相对对称加密速度较慢，典型的非对称加密算法有RSA、DSA等。

5. **讲一下对称加密算法和非对称加密算法?**

> 对称密钥加密，加密和解密使用同一密钥。运算速度快；无法安全地将密钥传输给通信方。典型的对称加密算法有DES、AES等
>
> 非对称密钥加密，加密和解密使用不同的密钥。通信发送方获得接收方的公开密钥之后，就可以使用公开密钥进行加密，接收方收到通信内容后使用私有密钥解密。可以更安全地将公开密钥传输给通信发送方；运算速度慢。典型的非对称加密算法有RSA、DSA等
>
> HTTPS 采用的加密方式: HTTPS 采用混合的加密机制。所有传输的内容都经过加密，加密采用对称加密，但对称加密的密钥用服务器方的证书进行了非对称加密。

6. **HTTP2.0讲一下**

> 1. **二进制传输**：HTTP/2 采用二进制格式传输数据，HTTP/2 将请求和响应数据分割为更小的帧，并且它们采用二进制编码。
> 2. **多路复用**：在 HTTP/2 中引入了多路复用的技术。在 HTTP/2 中，采用了二进制分帧，使性能有了极大提升。
> 3. **Header压缩**：HTTP/2 对首部采取了压缩策略，请求一发送了所有的头部字段，第二个请求则只需要发送差异数据，这样可以减少冗余数据，降低开销。

7. **HTTP报文详解？详细说一下请求报文，以及HTTP和TCP的区别**

> HTTP有两种报文：请求报文和响应报文

> HTTP请求报文主要包括请求行、请求头部以及请求的数据（实体）三部分
>
> **请求行（HTTP请求报文的第一行）**
>
> 请求行由方法字段、URL字段和HTTP协议版本字段。其中，方法字段严格区分大小写，当前HTTP协议中的方法都是大写，方法字段如下介绍如下：
>
> **请求头部：位于请求行的下面, 是一个个的key-value值**
>
> **空行(CR+LF)：请求报文用空行表示header和请求数据的分隔**
>
> **请求数据**：GET方法没有携带数据， POST方法会携带一个body

> HTTP的响应报文包括：状态行，响应头部，相应的数据(响应体)
>
> **状态行**包括：HTTP版本号，状态码和状态值组成。
>
> **响应头**类似请求头，是一系列key-value值
>
> **空白行：同上，响应报文也用空白行来分隔header和数据**
>
> **响应体**：响应的数据

8. **TCP三次握手的过程，以及三次握手的原因?**

> 假设 A 为客户端，B 为服务器端。
>
> - 首先 B 处于 LISTEN（监听）状态，等待客户的连接请求。
> - A 向 B 发送连接请求报文，SYN=1，ACK=0，选择一个初始的序号 x。
> - B 收到连接请求报文，如果同意建立连接，则向 A 发送连接确认报文，SYN=1，ACK=1，确认号为 x+1，同时也选择一个初始的序号 y。
> - A 收到 B 的连接确认报文后，还要向 B 发出确认，确认号为 y+1，序号为 x+1。
> - B 收到 A 的确认后，连接建立。

> **三次握手的目的是建立可靠的通信信道，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是正常的。**
>
> 第三次握手是为了防止失效的连接请求到达服务器，让服务器错误打开连接。

9. **TCP四次挥手的过程，以及四次挥手的原因?**

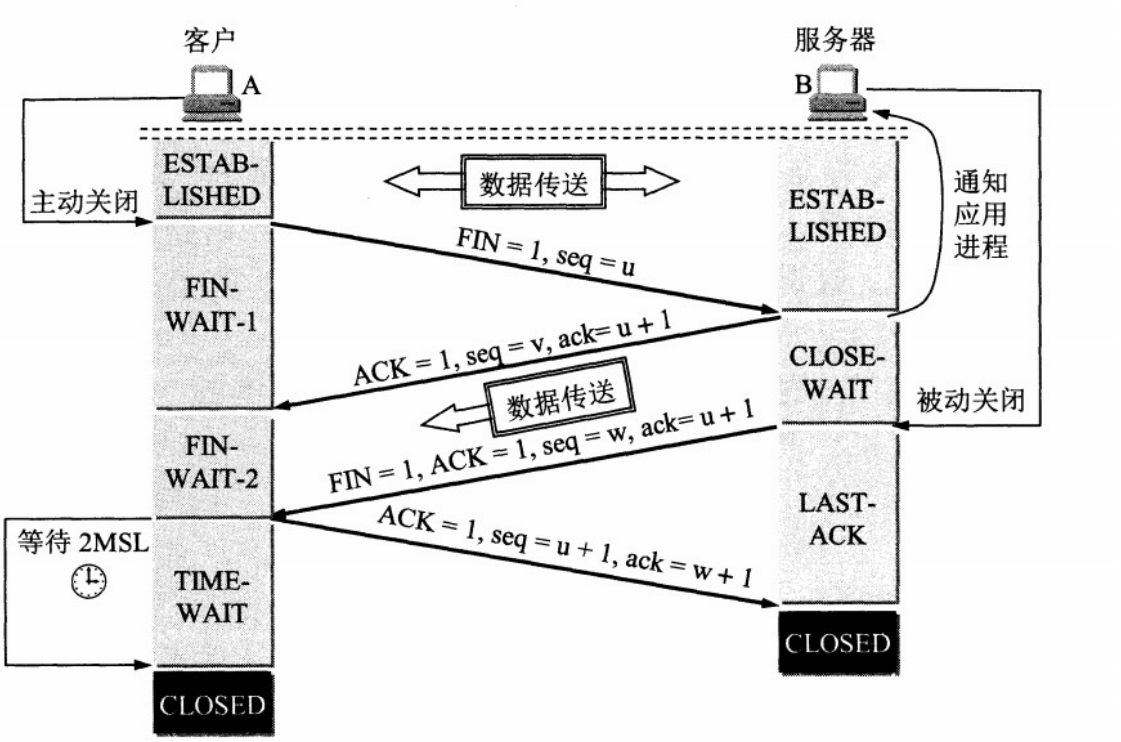


图 5-29　TCP 连接释放的过程

假设 A 为客户端，B 为服务器端。

- A 发送连接释放报文，FIN=1。
- B 收到之后发出确认，它发回一 个 ACK确认报文，确认序号为收到的序号加1。此时 TCP 属于半关闭状态，B 能向 A 发送数据但是 A 不能向 B 发送数据。
- 当 B 不再需要连接时，发送连接释放报文，FIN=1。
- A 收到后发出ACK 确认报文，并将确认序号设置为收到序号加1，进入 TIME-WAIT 状态，等待 2 MSL（最大报文存活时间）后释放连接。
- B 收到 A 的确认后释放连接。

CLOSE-WAIT 状态问题：

客户端发送了 FIN 连接释放报文之后，服务器收到了这个报文，就进入了 CLOSE-WAIT 状态。这个状态是为了让服务器端发送还未传送完毕的数据，传送完毕之后，服务器会发送 FIN 连接释放报文。

TIME-WAIT 状态问题(这个问题问过很多次但总是答得不甚满意)：

客户端接收到服务器端的 FIN 报文后进入此状态，此时并不是直接进入 CLOSED 状态，还需要等待一个时间计时器设置的时间 2MSL。这么做有两个理由：

- 确保最后一个确认报文能够到达。如果 B 没收到 A 发送来的确认报文，那么就会重新发送连接释放请求报文，A 等待一段时间就是为了处理这种情况的发生。MSL 是最大报文段寿命，等待 2MSL 可以保证 A 发送的最后一个确认报文被 B 接收，如果该报文丢失，B 会超时重传之前的 FIN+ACK 报文，保证 B 正常进入 CLOSED 状态。
- 等待一段时间是为了让本连接持续时间内所产生的所有报文都从网络中消失，使得下一个新的连接不会出现旧的连接请求报文。2MSL 后，本连接中的所有报文就都会从网络中消失，防止已失效请求造成异常。

通信双方建立TCP连接后，主动关闭连接的一方就会进入TIME_WAIT状态。

10. **TCP滑动窗口是干什么的？TCP的可靠性体现在哪里？拥塞控制如何实现的？**

滑动窗口：窗口是缓存的一部分，用来暂时存放字节流。发送方和接收方各有一个窗口，接收方通过 TCP 报文段中的窗口字段告诉发送方自己的窗口大小，发送方根据这个值和其它信息设置自己的窗口大小。

发送窗口内的字节都允许被发送，接收窗口内的字节都允许被接收。接收窗口只会对窗口内最后一个按序到达的字节进行确认。如果发送窗口内的字节已经发送并且收到了确认，那么就将发送窗口向右滑动一定距离，直到第一个字节不是已发送并且已确认的状态；接收窗口的滑动类似，接收窗口左部字节已经发送确认并交付主机，就向滑动接收窗口。

流量控制如何实现：流量控制是为了控制发送方发送速率，保证接收方来得及接收。

接收方发送的确认报文中的窗口字段可以用来控制发送方窗口大小，从而影响发送方的发送速率。将窗口字段设置为 0，则发送方不能发送数据。

拥塞控制：网络中对资源的需求超过可用量的情况就叫拥塞，拥塞控制就是减少注入网络的数据，减轻路由器和链路的负担，这是一个全局性问题，涉及网络中的所有路由器和主机，而流量控制是一个端到端的问题。如果网络出现拥塞，分组将会丢失，此时发送方会继续重传，从而导致网络拥塞程度更高。因此当出现拥塞时，应当控制发送方的速率。TCP 主要通过四个算法来进行拥塞控制：慢开始、拥塞避免、快重传、快恢复。

发送方需要维护一个叫做拥塞窗口（cwnd）的状态变量。

1. 慢开始与拥塞避免

   发送的最初执行慢开始，令拥塞窗口大小为1，发送方只能发送1个报文段；当收到确认后，将拥塞窗口大小加倍。设置一个慢开始门限，当 拥塞窗口的大小大于慢开始门限时，进入拥塞避免，每个轮次只将拥塞窗口加1。如果出现了超时，则令慢开始门限 = 拥塞窗口大小 / 2，然后重新执行慢开始。

2. 快重传与快恢复

   在接收方，要求每次接收到报文段都应该对最后一个已收到的有序报文段进行确认。在发送方，如果收到三个重复确认，那么可以知道下一个报文段丢失，此时执行快重传，立即重传下一个报文段。在这种情况下，只是丢失个别报文段，而不是网络拥塞。因此执行快恢复，令慢开始门限 = 拥塞窗口大小 / 2，拥塞窗口大小 = 慢开始门限 ，注意到此时直接进入拥塞避免。

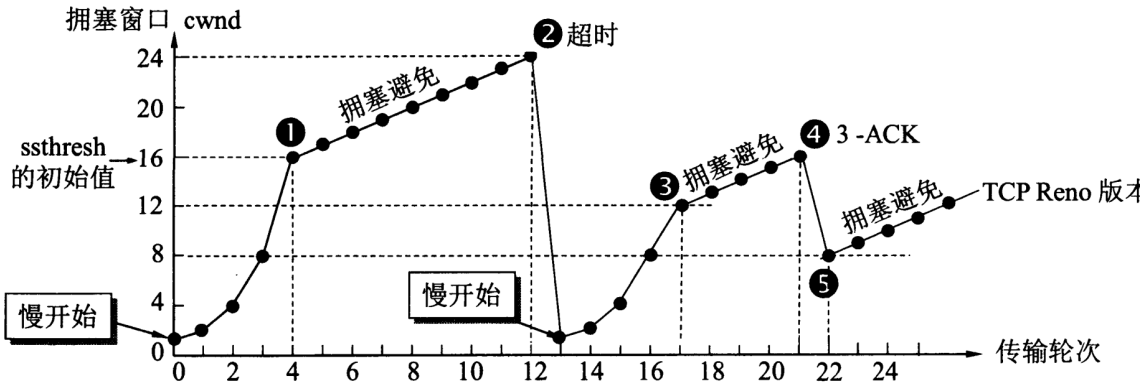


图 5-25　TCP 拥塞窗口 cwnd 在拥塞控制时的变化情况

主要的考虑还是要区分包的丢失是由于链路故障还是乱序等其他因素引发。两次冗余ACK时很可能是乱序造成的！三次冗余ACK(三个重复确认)时很可能是丢包造成的！

（主要）TCP 使用超时重传来实现可靠传输：如果一个已经发送的报文段在超时时间内没有收到确认，那么就重传这个报文段。

1. 应用数据被分割成 TCP 认为最适合发送的数据块。
2. TCP 给发送的每一个包进行编号，接收方对数据包进行排序，把有序数据传送给应用层。
3. **校验和：** TCP 将保持它首部和数据的检验和。这是一个端到端的检验和，目的是检测数据在传输过程中的任何变化。如果收到段的检验和有差错，TCP 将丢弃这个报文段和不确认收到此报文段。
4. TCP 的接收端会丢弃重复的数据。
5. **流量控制：** TCP 连接的每一方都有固定大小的缓冲空间，TCP的接收端只允许发送端发送接收端缓冲区能接纳的数据。当接收方来不及处理发送方的数据，能提示发送方降低发送的速率，防止包丢失。TCP 使用的流量控制协议是可变大小的滑动窗口协议。 （TCP 利用滑动窗口实现流量控制）
6. **拥塞控制：** 当网络拥塞时，减少数据的发送。

7. **ARQ协议：** 也是为了实现可靠传输的，它的基本原理就是每发完一个分组就停止发送，等待对方确认。在收到确认后再发下一个分组。
8. **超时重传：** 当 TCP 发出一个段后，它启动一个定时器，等待目的端确认收到这个报文段。如果不能及时收到一个确认，将重发这个报文段。

11. **TCP和UDP有什么区别？及其适用的场景。**

- 用户数据报协议 UDP是无连接的，尽最大可能交付，没有拥塞控制，面向报文（对于应用程序传下来的报文不合并也不拆分，只是添加 UDP 首部），支持一对一、一对多、多对一和多对多的交互通信。
- 传输控制协议 TCP是面向连接的，提供可靠交付，有流量控制，拥塞控制，提供全双工通信，面向字节流（把应用层传下来的报文看成字节流，把字节流组织成大小不等的数据块），每一条 TCP 连接只能是点对点的（一对一）。

- TCP应用场景：
  效率要求相对低，但对准确性要求相对高的场景。因为传输中需要对数据确认、重发、排序等操作，相比之下效率没有UDP高。举几个例子：文件传输、接受邮件、远程登录。
- UDP应用场景：
  适用于一次只传送少量数据、对可靠性要求不高的应用环境。举几个例子：QQ聊天、在线视频、网络语音电话、广播通信（广播、多播）。

UDP为何快?

1.不需要建立连接

2.对于收到的数据，不用给出确认

3.没有超时重发机制

4.没有流量控制和拥塞控制

12. **Mac 地址和 ip 地址的区别？既然有了 Mac 地址，为什么还要 ip 地址呢?**

MAC地址是烧录在网卡或者接口上的**物理地址**，具有**全球唯一性**，一般不能被改变。IP地址是网络中的主机或接口在网络中的**逻辑地址**，在**同一个网络内具有唯一性。**

13. **当你打开一个电商网站，都需要经历哪些过程？分别用到了什么协议。**

1. 浏览器查找域名的IP地址 （DNS：获取域名对应的IP）
2. 浏览器向web服务器发送HTTP请求（cookies会随着请求发送给服务器）
3. 服务器处理请求 （请求 处理请求 参数、cookies、生成一个HTML响应）
4. 服务器返回HTTP报文，发回一个HTML响应。
5. 浏览器解析渲染页面，浏览器开始显示HTML。
6. 连接结束

使用的协议:

DNS: 获取域名对应的IP

TCP: 与服务器建立TCP连接

IP: 建立TCP协议时，需要发送数据，发送数据在网络层上使用IP协议

OSPF：IP数据包在路由器之间，路由选择使用OSPF协议

ARP：路由器在与服务器进行通信的过程中，将IP地址装换成MAC地址

HTTP：客户端浏览器与Web服务器之间的应用层通信协议，在TCP建立完成后，使用HTTP协议访问网页

14. **电子邮件的发送过程?**

一个电子邮件系统由三部分组成：用户代理、邮件服务器以及邮件协议。

邮件协议包含发送协议和读取协议，发送协议常用 SMTP，读取协议常用 POP3 和 IMAP。

1. 用户A的邮箱是QQ邮箱，他要发往的邮箱是163邮箱，用户A写好一封邮件点击发送，即提交到了QQ邮箱服务器，使用的是SMTP协议。

2. QQ邮箱会对A发送邮件的收件地址进行解析，判断是否为内部邮箱的账号，如果也是QQ邮箱，会直接存储到自己的存储空间，如果不是则会发送到指定邮箱服务器，使用的也是SMTP协议。
3. 163的邮箱服务器收到邮件后会再次判断该邮件是否为自己的邮件，如果是则存到自己的存储空间，等待POP3服务去读取邮件。
4. 用户B收到消息后，打开客户端访问163服务器，调用POP3服务。
5. Pop3服务接到指令后，读取存储空间中发送给B的未读邮件服务。
6. 将读取到的邮件返回给客户端软件。

15. **DNS解析过程，DNS劫持了解吗?**

> DNS完成的工作是：域名到IP地址的解析。将域名和IP地址相互映射的一个分布式数据库。

> 域名解析大概分为两步：
>
> - 第一步：向本地域名服务器发起查询请求，请求报文里面含有需要查询的域名；
> - 第二步：本地域名服务器返回DNS响应，响应报文中含有DNS解析的IP地址；
>
> 在DNS解析过程中使用的是迭代查询——本地域名服务器向根域名服务器发起查询，根域名服务器告诉本地域名服务器下一步该往哪个域名服务器走，一步步按照域名服务的查询路径找到域名对应的IP返回给请求发起方。

> 第一步：客户机提出域名解析请求，并将该请求发送给本地域名服务器。
>
> 第二步：当本地域名服务器收到请求后，就先查询本地缓存，如果有该纪录项，则本地域名服务器就直接把查询结果返回。
>
> 第三步：如果本地缓存中没该纪录，则本地域名服务器就直接把请求发给根域名服务器，然后根域名服务器再返回给本地域名服务器一个所查询域(根子域)主域名服务器地址。
>
> 第四步：本地服务器再向上一步返回的域名服务器发送请求，然后接受请求的服务器查询自己的缓存，如果没该纪录，则返回相关下级域名服务器地址。
>
> 第五步：重复第四步，直到找到正确纪录。
>
> 第六步：本地域名服务器把返回结果保存到缓存，以备下一次使用，同时还将结果返回给客户机。

> （1）递归查询：本机向本地域名服务器发出一次查询请求，就静待最终的结果。如果本地域名服务器无法解析，自己会以DNS客户机的身份向其它域名服务器查询，直到得到最终的IP地址告诉本机
> （2）迭代查询：本地域名服务器向根域名服务器查询，根域名服务器告诉它下一步到哪里去查询，然后它再去查，每次它都是以客户机的身份去各个服务器查询。

> DNS在进行区域传输的时候使用TCP，普通的查询使用UDP。为什么查询是使用UDP呢?
>
> 域名解析时使用UDP协议：
> 客户端向DNS服务器查询域名，一般返回的内容都不超过512字节，用UDP传输即可。不用经过TCP三次握手，这样DNS服务器负载更低，响应更快。
>
> 区域传送时使用TCP，主要有一下两点考虑：
> 1.辅域名服务器会定时（一般时3小时）向主域名服务器进行查询以便了解数据是否有变动。如有变动，则会执行一次区域传送，进行数据同步。区域传送将使用TCP而不是UDP，因为数据同步传送的数据量比一个请求和应答的数据量要多得多。
> 2.TCP是一种可靠的连接，保证了数据的准确性。

> DNS劫持：在DNS服务器中，将www..com的域名对应的IP地址进行了变化。你解析出来的域名对应的IP，在劫持前后不一样。
> HTTP劫持：你DNS解析的域名的IP地址不变。在和网站交互过程中的劫持了你的请求。在网站发给你信息前就给你返回了请求。

> DNS在区域传输的时候使用TCP协议,其他时候使用UDP协议。

16. **GET和POST有什么不一样?**

> GET和POST是HTTP请求的两种基本方法（记不住全部，只记这么点）

1. 最直观的区别就是GET把参数包含在URL中，POST通过request body传递参数。
2. GET请求在URL中传送的参数是有长度限制的，而POST没有。
3. GET比POST更不安全，因为参数直接暴露在URL上，所以不能用来传递敏感信息。

17. **session和cookie的问题?**

> Cookie 和 Session都是用来跟踪浏览器用户身份的会话方式
>
> **Cookie 一般用来保存用户信息，Session 的主要作用就是通过服务端记录用户的状态**
>
> Cookie 数据保存在客户端(浏览器端)，Session 数据保存在服务器端。相对来说 Session 安全性更高。如果要在 Cookie 中存储一些敏感信息，不要直接写入 Cookie 中，最好能将 Cookie 信息加密然后使用到的时候再去服务器端解密。

18. **HTTP是不保存状态的协议,如何保存用户状态?**

> HTTP 是一种无状态协议。HTTP 协议自身不对请求和响应之间的通信状态进行保存。主要通过session机制来进行解决，Session 的主要作用就是通过服务端记录用户的状态。
>
> 在服务端保存 Session 的方法很多，最常用的就是内存和数据库(比如是使用内存数据库redis保存)。既然 Session 存放在服务器端，那么我们如何实现 Session 跟踪呢？大部分情况下，我们都是通过在 Cookie 中附加一个 Session ID 来方式来跟踪。
>
> **Cookie 被禁用怎么办?** 最常用的就是利用 URL 把 Session ID 直接附加在URL路径的后面。

19. **Arp协议?**

> Arp协议能通过接收端的ip地址解析到mac地址。
>
> 如果发送端和目标端的主机都在同一个网段，发送端发送数据帧前检查是否拥有接收端的mac地址，如果没有，则启动arp，先检查缓存ip-mac表中是否有接收端的mac地址，如果有则直接拿来即用，如果没有则在本网段（局域网）广播arp包，本网段各计算机都收到arp请求，从发送来的数据中检查请求过来的ip地址与自己是否一致，如果不一致，则丢弃，如果ip一致，则单播返回mac地址给请求的计算机，发送端便获取到了接收端的mac地址，接收到接收端的mac地址它还会缓存一份，用于下次拿来即用。
>
> 如果请求端和目标端的主机不在同一个网段呢？arp广播的数据是被路由阻断的，不能跨到不同的网段进行广播的，因为这样广播会导致广播数据泛滥。如果不在同一个网段，则请求端拿到的目标端的mac地址其实是它网关的mac地址，将数据帧给到网关再进行下一跳转发，下一跳同样在自己的网段寻找到目标主机mac地址或再找到下一跳mac地址。

20. **DDos攻击了解吗?**

> 分布式拒绝服务，一般来说是指攻击者利用一些被控制的设备对目标网站在较短的时间内发起大量请求，大规模消耗目标网站的主机资源，让它无法正常服务。

21. **SQL注入攻击?**

> 针对程序员编写时的疏忽，将SQL语句传递到服务器解析并执行的一种攻击手段。

---

10. 分布式

1. **如何理解分布式?**

> 多个能独立运行的结点组成。各个结点利用计算机网络进行信息传递，从而实现共同的“目标或者任务”。主要可以完成一些诸如分布式计算，分布式存储之类的工作。涉及到的主要问题就是保证分布式系统的一致性，分布式系统高可用的问题。

2. **分布式锁的实现方式**

> 在分布式场景下，需要同步的进程可能位于不同的节点上，那么就需要使用分布式锁。
>
> 分布式锁的实现方式主要有：数据库的唯一索引，Redis 的 SETNX 指令，Redis 的 RedLock 算法，Zookeeper 的有序节点。
>
> **数据库的唯一索引**：获得锁时向表中插入一条记录，释放锁时删除这条记录。
>
> **Redis 的 SETNX 指令**：使用 SETNX（set if not exist）指令插入一个键值对，如果 Key 已经存在，那么会返回 False，否则插入成功并返回 True。
>
> **Redis 的 RedLock 算法**：使用了多个 Redis 实例来实现分布式锁。1. 尝试从 N 个互相独立 Redis 实例获取锁；2. 计算获取锁消耗的时间，只有时间小于锁的过期时间，并且从大多数（N / 2 + 1）实例上获取了锁，才认为获取锁成功；3. 如果获取锁失败，就到每个实例上释放锁。
>
> **Zookeeper 的有序节点**：1. 创建一个锁目录 /lock；2. 当一个客户端需要获取锁时，在 /lock 下创建临时的且有序的子节点；3. 客户端获取 /lock 下的子节点列表，判断自己创建的子节点是否为当前子节点列表中序号最小的子节点，如果是则认为获得锁；否则监听自己的前一个子节点，获得子节点的变更通知后重复此步骤直至获得锁；4. 执行业务代码，完成后，删除对应的子节点。

> 锁无法释放问题，单点问题，不可重入问题，非阻塞问题，性能问题。
>
> 1. 基于数据库
>
>   (1) 锁无法释放，解锁失败的话其他进程无法获得该锁。
>
>   (2) 不可重入，已经获得锁的进程必须重新获得锁。
>
>   (3) 只能是非阻塞锁，插入失败就报错了，无法重试。
>
> 2. 基于缓存（redis）
>
>   单点问题：很多缓存都是集群部署的。锁无法释放问题：可以解决redis可对记录设置过期时间，防止系统崩溃锁无法自动释放。非阻塞问题：while重复执行，消耗CPU资源。使用超时机制控制锁的释放不是十分靠谱。可重入：主机信息和线程信息保存起来，检查是不是当前锁的拥有者。
>
> 3. 基于ZooKeeper
>
>   有效解决单点问题，不可重入问题（节点数据进行比对），非阻塞问题（绑定监听器），以及锁无法释放问题（客户端挂掉则删除节点），性能上不如用缓存进行实现。
>
>   > redis分布式锁，其实需要自己不断去尝试获取锁，比较消耗性能。
>   >
>   > zk分布式锁，获取不到锁，注册个监听器即可，不需要不断主动尝试获取锁，性能开销较小。
>   >
>   > 另外一点就是，如果是redis获取锁的那个客户端bug了或者挂了，那么只能等待超时时间之后才能释放锁；而zk的话，因为创建的是临时znode，只要客户端挂了，znode就没了，此时就自动释放锁

3. **讲一下负载均衡算法**

> 负载均衡器会根据集群中每个节点的负载情况，将用户请求转发到合适的节点上。
>
> 1. 轮询：轮询算法把每个请求轮流发送到每个服务器上。
> 2. 加权轮询：加权轮询是在轮询的基础上，根据服务器的性能差异，为服务器赋予一定的权值，性能高的服务器分配更高的权值。
> 3. 最少连接：最少连接算法就是将请求发送给当前最少连接数的服务器上。
> 4. 加权最少连接：在最少连接的基础上，根据服务器的性能为每台服务器分配权重，再根据权重计算出每台服务器能处理的连接数。
> 5. 随机算法：把请求随机发送到服务器上。
> 6. 源地址哈希法：通过对客户端 IP 计算哈希值之后，再对服务器数量取模得到目标服务器的序号。

4. **分布式事务解决方案?**

分布式事务就是指事务的参与者、支持事务的服务器、资源服务器以及事务管理器分别位于不同的分布式系统的不同节点之上。简单的说，就是一次大的操作由不同的小操作组成，这些小的操作分布在不同的服务器上，且属于不同的应用，分布式事务需要保证这些小操作要么全部成功，要么全部失败。本质上来说，分布式事务就是为了保证不同数据库的数据一致性。

1. 两阶段提交（2PC）

   两阶段提交（Two-phase Commit，2PC），通过引入协调者来协调参与者的行为，并最终决定这些参与者是否要真正执行事务。

   1. 准备阶段: 协调者向所有参与者发送事务内容，并等待答复，参与者发回事务执行结果。
   2. 提交阶段: 如果事务在每个参与者上都执行成功，协调者发送通知让参与者提交事务；否则，协调者发送通知让参与者回滚事务。

   > XA 是一个两阶段提交协议，该协议分为以下两个阶段
   >
   > - 第一阶段：事务协调器要求每个涉及到事务的数据库预提交(precommit)此操作，并反映是否可以提交.
   > - 第二阶段：事务协调器要求每个数据库提交数据。

   优点： 尽量保证了数据的强一致，适合对数据强一致要求很高的关键领域。（其实也不能100%保证强一致）

   缺点： 实现复杂，牺牲了可用性，对性能影响较大，不适合高并发高性能场景。

2. 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（一致性）及资源预留（准隔离性）。
   - Confirm 阶段主要是对业务系统做确认提交，执行真正的业务。
   - 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。

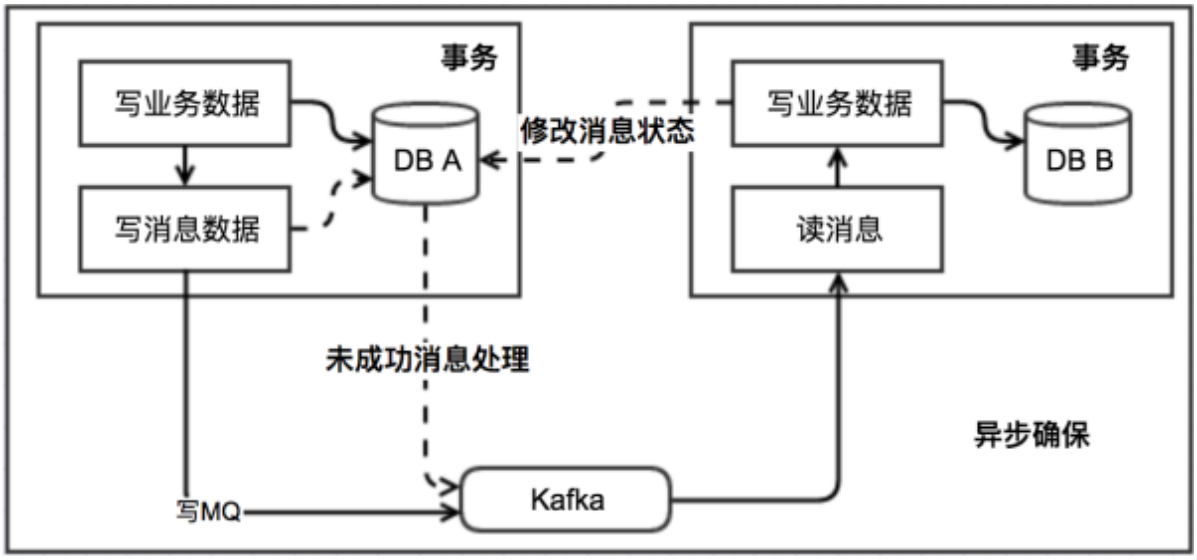
   优点： 跟2PC比起来，实现以及流程相对简单了一些，但数据的一致性比2PC也要差一些

   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。

3. 消息事务

   所谓的消息事务就是基于消息中间件的两阶段提交，本质上是对消息中间件的一种特殊利用，它是将本地事务和发消息放在了一个分布式事务里，保证要么本地操作成功成功并且对外发消息成功，要么两者都失败。

   原理：

   1. 在分布式事务操作的一方完成写业务数据的操作之后向本地消息表发送一个消息。

2. 之后将本地消息表中的消息转发到 Kafka 等消息队列中，在分布式事务操作的另一方从消息队列中读取一个消息，并执行消息中的操作。
3. 如果转发成功则将消息从本地消息表中删除，否则继续重新转发。

基于消息中间件的两阶段提交往往用在高并发场景下，将一个分布式事务拆成一个消息事务（A系统的本地操作+发消息）+B系统的本地操作，其中B系统的操作由消息驱动，只要消息事务成功，那么A操作一定成功，消息也一定发出来了，这时候B会收到消息去执行本地操作，如果本地操作失败，消息会重投，直到B操作成功，这样就变相地实现了A与B的分布式事务。

这种方案遵循BASE理论，采用的是最终一致性，笔者认为是这几种方案里面比较适合实际业务场景的，即不会出现像2PC那样复杂的实现(当调用链很长的时候，2PC的可用性是非常低的)，也不会像TCC那样可能出现确认或者回滚不了的情况。

优点： 一种非常经典的实现，避免了分布式事务，实现了最终一致性。在 .NET中 有现成的解决方案。

缺点： 消息表会耦合到业务系统中，如果没有封装好的解决方案，会有很多杂活需要处理。

5. **讲一下分布式和分布式一致性协议**

CAP

分布式系统不可能同时满足一致性（C：Consistency）、可用性（A：Availability）和分区容忍性（P：Partition Tolerance），最多只能同时满足其中两项。在分布式系统中，分区容忍性必不可少，因为需要总是假设网络是不可靠的。因此，CAP 理论实际上是要在可用性和一致性之间做权衡。

一致性（C：Consistency）：一致性指的是多个数据副本是否能保持一致的特性，在一致性的条件下，系统在执行数据更新操作之后能够从一致性状态转移到另一个一致性状态。

可用性（A：Availability）：可用性指分布式系统在面对各种异常时可以提供正常服务的能力。

分区容忍性（P：Partition Tolerance）：在分区容忍性条件下，分布式系统在遇到任何网络分区故障的时候，仍然需要能对外提供一致性和可用性的服务。

BSAE

BASE 是基本可用（Basically Available）、软状态（Soft State）和最终一致性（Eventually Consistent）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，它的核心思想是：**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。**

基本可用（Basically Available）：指分布式系统在出现故障的时候，保证核心可用，允许损失部分可用性。

软状态（Soft State）：允许系统不同节点的数据副本之间进行同步的过程存在时延。

最终一致性（Eventually Consistent）：最终一致性强调的是系统中所有的数据副本，在经过一段时间的同步后，最终能达到一致的状态。

Paxos

Paxos算法需要解决的问题就是如何在一个可能发生上述异常的分布式系统中，快速且正确地在集群内部对**某个数据的值**达成一致

包括三个阶段：Prepare 阶段，Accept 阶段，Learn 阶段。

主要有三类节点：

- 提议者（Proposer）：提议一个值；
- 接受者（Acceptor）：对每个提议进行投票；
- 告知者（Learner）：被告知投票的结果，不参与投票过程。

Paxos算法分为**两个阶段。**具体如下：

- **阶段一：**

  (a) Proposer选择一个**提案编号N**，然后向**半数以上**的Acceptor发送编号为N的**Prepare请求。**

  (b) 如果一个Acceptor收到一个编号为N的Prepare请求，且N**大于**该Acceptor已经**响应过的**所有**Prepare请求**的编号，那么它就会将它已经**接受过的编号最大的提案（如果有的话）**作为响应反馈给Proposer，同时该Acceptor承诺**不再接受**任何**编号小于N的提案。**

- **阶段二：**

  (a) 如果Proposer收到**半数以上**Acceptor对其发出的编号为N的Prepare请求的**响应**，那么它就会发送一个针对**[N,V]提案**的**Accept请求**给**半数以上**的Acceptor。注意：V就是收到的**响应**中**编号最大的提案的value**，如果响应中**不包含任何提案**，那么V就由Proposer**自己决定。**

  (b) 如果Acceptor收到一个针对编号为N的提案的Accept请求，只要该Acceptor**没有**对编号**大于N**的**Prepare请求**做出过**响应**，它就**接受该提案。**

6. **Hadoop介绍**

> Hadoop一个能够允许大量数据在计算机集群中，通过使用简单的编程模型进行分布式处理的框架。
>
> Hadoop最基础的几个模块有:
>
> **Common**: 支持其他模块的公用工具包。
>
> **HDFS**: 一个可高吞吐访问应用数据的分布式文件系统。
>
> **YARN**: 一个管理集群服务器资源和任务调度的框架。
>
> **MapReduce**: 基于Yarn对大数据集进行并行计算的系统。

7. **MapReduce编程模型**

> 简单在于其编程模型只包含map和reduce两个过程，map的主要输入是一对<key , value>值，经过map计算后输出一对<key , value>值；然后将相同key合并，形成<key , value>集合；再将这个<key , value>集合输入reduce，经过计算输出零个或多个<key , value>对。
>
> 每个map任务的计算结果都会写入到本地文件系统，等map任务快要计算完成的时候，MapReduce计算框架会启动shuffle过程，对map产生的每个<key , value>进行reduce分区选择，然后通过http通信发送给对应的reduce进程。reduce端对收到的<key , value>进行排序和合并，相同的key放在一起，组成一个<key , value>集合传递给reduce执行。

8. **Flink的状态机制?**

> Flink 内置的很多算子，包括源 source，数据存储 sink 都是有状态State的。
>
> 在 Flink 中，状态始终与特定Task/Operator相关联。而Check point可以理解为把State数据持久化存储了。Flink 会以 Check point 的形式对各个任务的状态进行快照，用于保证故障恢复时的状态一致性。Flink 通过状态后端来管理状态和checkpoint的存储，状态后端可以有不同的配置选择。

9. **Flink的容错机制?**

> 流数据的一致性就是：成功处理故障并恢复之后得到的结果与没有发生任何故障得到的结果相比，前者具有正确性。也就是故障的发生是否影响得到的结果。
>
> 在流处理过程，一致性分为3个级别：
>
> - at-most-once：至多一次。故障发生之后，计算结果可能丢失，就是无法保证结果的正确性；
> - at-least-once：至少一次。计算结果可能大于正确值，但绝不会小于正确值，就是计算程序发生故障后可能多算，但是绝不可能少算；
> - exactly-once：精确一次。系统保证发生故障后得到的计算结果的值和正确值一致；
>
> Flink的容错机制保证了exactly-once，也可以选择at-least-once。Flink的容错机制是通过对数据流不停的做快照（snapshot）实现的。Flink做快照的过程是基于“轻量级异步快照”的算法，其核心思想就是在计算过程中保存中间状态和在数据流中对应的位置，在系统故障恢复时，系统会从最新的一个checkpoint开始重新计算，对应的数据源也会在对应的位置“重放“。这里的“重放”可能会导致数据的二次输出。
>
> 当算子收到其中一个数据源的barriers，而未收到另一个数据源的barriers时，会将先到barriers的数据源中的数据先缓冲起来，等待另一个barriers，当收到两个barriers即接收到全部数据源的barrier时，会做checkpoint，保存barriers位置和状态，释放缓冲中的数据，释放一个对应的barriers。这里需要注意是，当缓存中数据没有被发射完时，是不会处理后续数据的，这样是为了保证数据的有序性。
>
> **屏障（Barriers）**

> 在Flink做分布式快照过程中核心是Barriers的使用。这些Barriers是在数据接入到Flink之初就注入到数据流中，并随着数据流向每个算子。
>
> - 算子对Barriers是免疫的，即Barriers是不参与计算的；
> - Barriers和数据的相对位置是保持不变的，而且Barriers之间是线性递增的；
>
> Barriers将数据流分成了一个个数据集。当barriers流经算子时，会触发与checkpoint相关的行为，保存的barriers的位置和状态（中间计算结果）。
>
> **状态（State）**
>
> 在一次snapshot中，算子会在接受到其数据源的**所有**barriers的以后snapshot它们的状态，然后在发射barriers到输出流中，直到最后所有的sink算子都完成snapshot才算完成一次snapshot。其中，在准备发射的barriers形成之前，state 形式是可以改变的，之后就不可以了。state的存贮方式是可以配置的，如HDFS，默认是在JobManager的内存中。
>
> **异步快照（asynchronous state snapshot）**
>
> 上述描述中，需要**等待**算子接收到所有barriers后，开始做snapshot，存储对应的状态后，再进行下一次snapshot，其状态的存储是同步的，这样可能会造成因snapshot引起较大延时。可以让算子在存储快照时继续处理数据，让快照存储异步在后台运行。为此，算子必须能生成一个 state 对象，保证后续状态的修改不会改变这个 state 对象。异步状态快照，其可以让算子接受到barriers后开始在后台异步拷贝其状态，而不必等待所有的barriers的到来。一旦后台的拷贝完成，将会通知JobManager。只有当所有的sink接收到这个barriers，和所有的有状态的算子都确认完成状态的备份时，一次snapshot才算完成。

10. **Flink的高可用?**

> 本质也就是JobManager 高可用(HA)，JobManager协调每个flink任务部署。它负责任务调度和资源管理。 默认情况下，每个flink集群只有一个JobManager，这将导致一个单点故障：如果JobManager挂了，则不能提交新的任务，并且运行中的程序也会失败。 Flink利用 ZooKeeper在所有正在运行的 JobManager 实例之间进行分布式协调。
>
> 对于Standalone集群模式下的JobManager高可用通常的方案是：Flink集群的任一时刻只有一个leading JobManager，并且有多个standby JobManager。当leader失败后，standby通过选举出一个JobManager作为新的leader。这个方案可以保证没有单点故障的问题。
>
> 在运行高可用性 YARN 集群时，**我们不会运行多个 JobManager (ApplicationMaster) 实例**，而只运行一个，该JobManager实例失败时，YARN会将其重新启动。

11. **Flink 程序在面对数据高峰期时如何处理?**

> 使用 Kafka 把数据先放到消息队列里面作为数据源，再使用 Flink 进行消费，不过这样会影响到一点实时性。

12. **请详细解释一下 Flink 的 Watermark 机制?**

> Watermark 本质是 Flink 中衡量 EventTime 进展的一个机制，主要用来处理乱序数据。Watermark是Flink为了处理EventTime 窗口计算提出的一种机制，本质上也是一种时间戳。
>
> Flink中包括三种时间，Event time：事件产生的时间，它通常由事件的时间戳描述。Ingestion Time：事件进入Flink的时间。Processing Time：事件被处理时当前系统的时间。
>
> 目前Apache Flink 有两种生产Watermark的方式，Punctuated - 数据流中每一个递增的EventTime都会产生一个Watermark。Periodic - 周期性的(一定时间间隔或者达到一定的记录条数)产生一个Watermark。
>
> 输入数据是，根据Event time将其划分到不同的window中，如果window中有数据，则当Watermark时间 >= Event time时，就符合Window的触发条件。

---

11. Spring

1. **谈一谈Spring?**

> Spring 是一种轻量级开发框架，旨在提高开发人员的开发效率以及系统的可维护性。我们一般说 Spring 框架指的都是 Spring Framework，它是很多模块的集合，使用这些模块可以很方便地协助我们进行开发。 核心容器中的Core 组件是Spring 所有组件的核心，Beans 组件和 Context 组件是实现IOC和依赖注入的基础，AOP组件用来实现面向切面编程。

2. **Spring中的设计模式**

> 简单工厂模式：Spring 中的 BeanFactory，根据传入一个唯一的标识来获得 Bean 实例。
>
> 工厂方法模式：Spring 的 FactoryBean 接口的 `getObject` 方法。
>
> 单例模式：Spring 的 ApplicationContext 创建的 Bean 实例都是单例对象。
>
> 代理模式：Spring 的 AOP。
>
> 适配器模式：Spring MVC 中的 HandlerAdapter，由于 handler 有很多种形式，包括 Controller、HttpRequestHandler、Servlet 等，但调用方式又是确定的，因此需要适配器来进行处理，根据适配规则调用 handle 方法。

3. **Spring AOP 和 IOC 的底层实现。**

> **Spring AOP**
>
> AOP是面向切面编程，将那些与业务无关的，却可以被所有业务所调用的功能封装起来，从而使得业务逻辑各部分之间的耦合度降低，提高程序的可重用性，提高开发的效率。AOP 的实现原理就是代理模式。
>
> 代理模式的核心作用就是通过代理，控制对其他对象的访问，并对方法进行一定的增强。
>
> JDK 动态代理是利用反射机制生成一个实现代理接口的匿名类。动态代理类技术核心 Proxy类和一个 InvocationHandler 接口。每个代理的实例都有一个与之关联的 InvocationHandler 实现类。通过为Proxy类指定ClassLoader对象和一组interface来创建动态代理。JDK 动态代理只能只能代理实现了接口的类。
>
> CGLIB 采用了非常底层的字节码技术，其原理是通过字节码技术为一个类创建子类，并在子类中采用方法拦截的技术拦截所有父类方法的调用，顺势织入横切逻辑。
>
> Spring AOP 中的代理使用逻辑了：如果目标对象实现了接口，默认情况下会采用 JDK 的动态代理实现 AOP；如果目标对象没有实现了接口，则采用 CGLIB 库。

```
public class DefaultAopProxyFactory implements AopProxyFactory, Serializable {

    @Override
    public AopProxy createAopProxy(AdvisedSupport config) throws AopConfigException {
        if (config.isOptimize() || config.isProxyTargetClass() ||
hasNoUserSuppliedProxyInterfaces(config)) {
        Class<?> targetClass = config.getTargetClass();
        if (targetClass == null) {
                throw new AopConfigException("TargetSource cannot determine target class:
" +
                "Either an interface or a target is required for proxy creation.");
        }
        // 判断目标类是否是接口或者目标类是否Proxy类型，若是则使用JDK动态代理
        if (targetClass.isInterface() || Proxy.isProxyClass(targetClass)) {
                return new JdkDynamicAopProxy(config);
        }
        // 配置了使用CGLIB进行动态代理或者目标类没有接口，那么使用CGLIB的方式创建代理对象
        return new ObjenesisCglibAopProxy(config);
        }
        else {
        // 上面的三个方法没有一个为true，那使用JDK的提供的代理方式生成代理对象
        return new JdkDynamicAopProxy(config);
```

```
            }
        }
    //其他方法略.....
    }
```

Spring IOC:

IOC是控制反转，是一种通过描述（XML或注解）并通过第三方去生产或获取特定对象的方式。在Spring中实现控制反转的是IoC容器，其实现方法是依赖注入DI。

1. 定义用来描述bean配置的Java类。
2. 解析bean的配置，将bean的配置信息转换为BeanDefinition对象保存在内存中，Spring中采用ConcurrentHashMap进行对象存储。
3. 遍历存放BeanDefinition的Map对象，逐条取出BeanDefinition对象，获取bean的配置信息，利用Java的反射机制实例化对象，将实例化后的对象保存在另外一个Map中。

IoC 的思想就是两方之间不互相依赖，由第三方容器来管理相关资源。这样有什么好处呢?

1. 对象之间的耦合度或者说依赖程度降低;
2. 资源变的容易管理；比如你用 Spring 容器提供的话很容易就可以实现一个单例。

4. **讲一讲Spring，Spring MVC，Spring Boot。（Spring Boot 了解不？和 Spring 啥区别？spring与springboot比较，SpringBoot的自动启动）**

Spring与Spring boot比较

一、SpringBoot是能够创建出独立的Spring应用程序的

二、简化Spring配置

- Spring由于其繁琐的配置，一度被人成为“配置地狱”，各种XML、Annotation配置，让人眼花缭乱，而且如果出错了也很难找出原因。
- Spring Boot项目就是为了解决配置繁琐的问题，最大化的实现convention over configuration(**约定大于配置**)。
- - 提供一系列的依赖包来把其它一些工作做成开箱即用其内置一个’Starter POM’，对项目构建进行了**高度封装**，最大化简化项目构建的配置。

三、嵌入式Tomcat，Jetty容器，无需部署WAR包

5. **Spring Bean的生命周期。**

Bean的生命周期可以表达为：Bean的定义——Bean的初始化——Bean的使用——Bean的销毁 。

当一个 bean 被实例化时，它可能需要执行一些初始化使它转换成可用状态。同样，当 bean 不再需要，并且从容器中移除时，可能需要做一些清除工作。

init-method 属性指定一个方法，在实例化 bean 时，立即调用该方法。同样，destroy-method 指定一个方法，只有从容器中移除 bean 之后，才能调用该方法。

1. Spring容器 从XML 文件中读取bean的定义，并实例化bean。
2. Spring根据bean的定义填充所有的属性。
3. 如果Bean实现了BeanNameAware 接口，Spring 传递bean 的ID 到 setBeanName方法。
4. 如果Bean 实现了 BeanFactoryAware 接口， Spring传递beanfactory 给setBeanFactory 方法。
5. 如果有任何与bean相关联的BeanPostProcessors，Spring会在postProcesserBeforeInitialization()方法内调用它们。
6. 如果bean实现IntializingBean了，调用它的afterPropertySet方法，如果bean声明了初始化方法，调用此初始化方法。
7. 如果有BeanPostProcessors 和bean关联，这些bean的postProcessAfterInitialization() 方法将被调用。
8. 如果bean实现了 DisposableBean，它将调用destroy()方法。

6. **Spring Boot 的启动类源码有了解过吗?**

7. **Spring事务的实现原理?**

> Spring事务的本质其实就是数据库对事务的支持，没有数据库的事务支持，spring是无法提供事务功能的。
>
> 事务的实现原理。如果说你加了一个 `@Transactional` 注解，此时 Spring 会使用 AOP 思想，对你的这个方法在执行之前，先去开启一个事务。执行完毕之后，根据你的方法是否报错，来决定回滚还是提交事务。

> 1. 配置文件开启注解驱动，在相关的类和方法上通过注解@Transactional标识。
> 2. spring 在启动的时候会去解析生成相关的bean，这时候会查看拥有相关注解的类和方法，并且为这些类和方法生成代理，并根据@Transaction的相关参数进行相关配置注入，这样就在代理中为我们把相关的事务处理掉了（开启正常提交事务，异常回滚事务）。
> 3. 真正的数据库层的事务提交和回滚是通过bin log或者redo log实现的。

8. **用过哪些Spring注解，说一下@Autowired依赖注入的底层实现。（依赖注入的原理）**

> @Autowired 是按类型自动转配的。自动从spring的上下文找到合适的bean来注入，原理是利用反射机制为类的属性赋值的操作。
>
> 在容器启动，为对象赋值的时候，遇到@Autowired注解，会用后置处理器机制，来创建属性的实例，然后再利用反射机制，将实例化好的属性，赋值给对象上，这就是@Autowired的原理。
>
> 整个过程，解析bean-->保存beanDefition到IOC容器-->根据beanDefition实例化-->根据BeanpostProcessor依赖注入。

> @Configuration 和 @Bean：**@Configuration** 的注解类表示这个类可以使用 Spring IoC 容器作为 bean 定义的来源。**@Bean** 注解告诉 Spring，一个带有 @Bean 的注解方法将返回一个对象，该对象应该被注册为在 Spring 应用程序上下文中的 bean。
>
> @Service，@Controller，@Repository分别标记类是Service层类，Controller层类，数据存储层的类，spring扫描注解配置时，会标记这些类要生成bean。@Component是一种泛指，标记类是组件，spring扫描注解配置时，会标记这些类要生成bean。
>
> @Autowired和@Resource是用来修饰字段，构造函数，或者设置方法，并做注入的。而@Service，@Controller，@Repository，@Component则是用来修饰类。

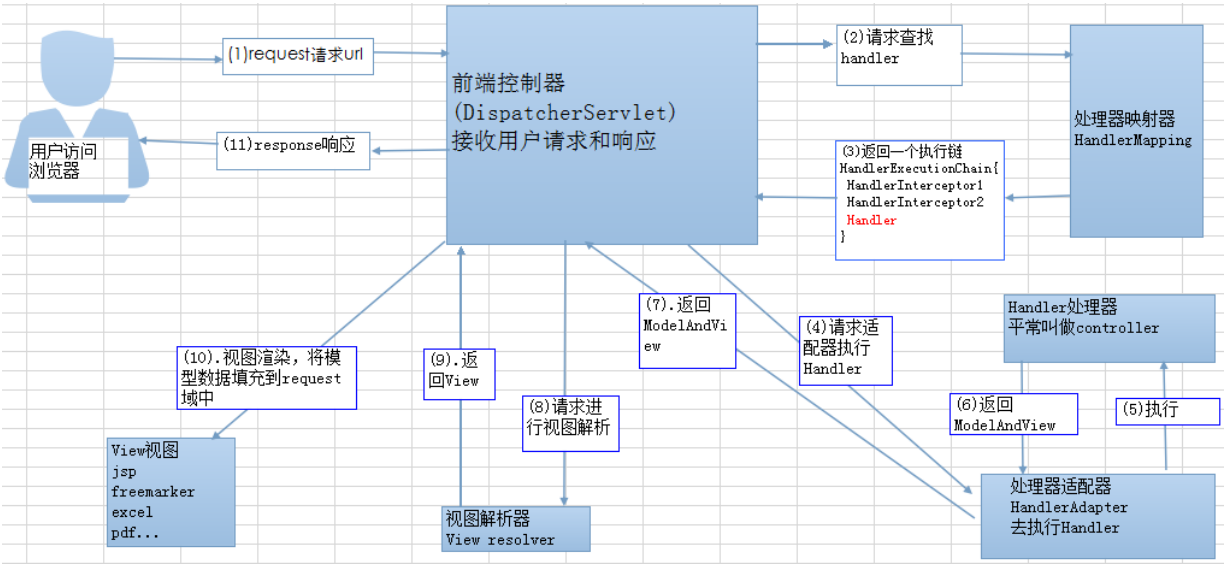
9. **Spring的三级缓存**

> Spring启动过程大致如下：
> 1.加载配置文件
> 2.解析配置文件转化beanDefination，获取到bean的所有属性、依赖及初始化用到的各类处理器等
> 3.创建beanFactory并初始化所有单例bean
> 4.注册所有的单例bean并返回可用的容器，一般为扩展的applicationContext。

> 1. `singletonObjects`：用于存放完全初始化好的 bean，**从该缓存中取出的 bean 可以直接使用**
> 2. `earlySingletonObjects`：提前曝光的单例对象的cache，存放原始的 bean 对象（尚未填充属性），用于解决循环依赖
> 3. `singletonFactories`：单例对象工厂的cache，存放 bean 工厂对象，用于解决循环依赖
>
> 所有单例的bean初始化完成后会存放在一个Map(**singletonObjects**)中。**一级缓存之后的其他缓存(二三级缓存)就是为了解决循环依赖**！先将没有填充属性的对象缓存起来，需要的时候先去用这个对象，不必等待一个对象完整的初始化好。而为什么是三级缓存不是二级缓存呢，这里笼统的来说还是方便 Spring 或者开发者们去拓展一些东西（比如后置处理器）。

10. **SpringMVC运行过程?**

> 客户端发送请求-> 前端控制器 DispatcherServlet 接受客户端请求 -> 找到处理器映射 HandlerMapping 解析请求对应的 Handler-> HandlerAdapter 会根据 Handler 来调用真正的处理器开处理请求，并处理相应的业务逻辑 -> 处理器返回一个模型视图 ModelAndView -> 视图解析器进行解析 -> 返回一个视图对象->前端控制器 DispatcherServlet 渲染数据（Moder）->将得到视图对象返回给用户



流程说明（重要）：

（1）客户端（浏览器）发送请求，直接请求到 DispatcherServlet。

（2）DispatcherServlet 根据请求信息调用 HandlerMapping，解析请求对应的 Handler。

（3）解析到对应的 Handler（也就是我们平常说的 Controller 控制器）后，开始由 HandlerAdapter 适配器处理。

（4）HandlerAdapter 会根据 Handler 来调用真正的处理器开处理请求，并处理相应的业务逻辑。

（5）处理器处理完业务后，会返回一个 ModelAndView 对象，Model 是返回的数据对象，View 是个逻辑上的 View。

（6）ViewResolver 会根据逻辑 View 查找实际的 View。

（7）DispaterServlet 把返回的 Model 传给 View（视图渲染）。

（8）把 View 返回给请求者（浏览器）

---

12. Docker

1. 介绍一下K8S、Docker技术，和传统虚拟机有什么不一样?

> Docker是一个开源的软件容器平台，开发者可以打包他们的应用及依赖到一个可移植的容器中，发布到流行的Linux机器上，也可实现虚拟化。
>
> k8s是一个开源的容器集群管理系统，可以实现容器集群的自动化部署、自动扩容、维护等功能。

> 容器就是将软件打包成标准化单元，以用于开发、交付和部署。
>
> **容器是一个应用层抽象，用于将代码和依赖资源打包在一起。 多个容器可以在同一台机器上运行，共享操作系统内核，但各自作为独立的进程在用户空间中运行 。**
>
> **虚拟机 (VM) 是一个物理硬件层抽象，传统虚拟机技术是虚拟出一套硬件后，在其上运行一个完整操作系统，在该系统上再运行所需应用进程。**

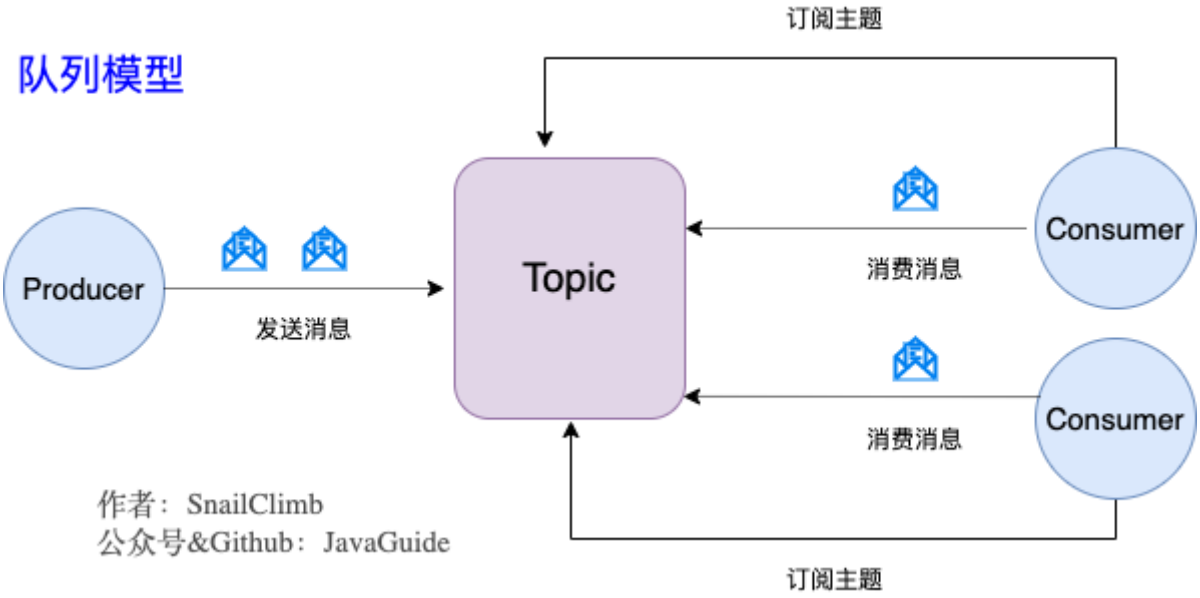
> **Docker 中有非常重要的三个基本概念**
>
> - **镜像（Image）：Docker 镜像是一个特殊的文件系统，除了提供容器运行时所需的程序、库、资源、配置等文件外，还包含了一些为运行时准备的一些配置参数（如匿名卷、环境变量、用户等）。**
> - **容器（Container）：容器是镜像运行时的实体。容器可以被创建、启动、停止、删除、暂停等 。**
> - **仓库（Repository）：镜像仓库是 Docker 用来集中存放镜像文件的地方类似于我们之前常用的代码仓库。**

13. 消息队列

1. **Kafka 是什么？主要应用场景有哪些?**

> Kafka 是一个分布式流式处理平台。
>
> 流平台具有三个关键功能：
>
> 1. **消息队列**：发布和订阅消息流，这个功能类似于消息队列，这也是 Kafka 也被归类为消息队列的原因。
> 2. **容错的持久方式存储记录消息流**： Kafka 会把消息持久化到磁盘，有效避免了消息丢失的风险·。
> 3. **流式处理平台：** 在消息发布的时候进行处理，Kafka 提供了一个完整的流式处理类库。
>
> Kafka 主要有两大应用场景：
>
> 1. **消息队列** ：建立实时流数据管道，以可靠地在系统或应用程序之间获取数据。
> 2. **数据处理：** 构建实时的流数据处理程序来转换或处理数据流。

2. **kafka的消息模型**

> 发布-订阅模型
>
> 发布订阅模型（Pub-Sub） 使用**主题（Topic）** 作为消息通信载体，类似于**广播模式**；发布者发布一条消息，该消息通过主题传递给所有的订阅者。
>
> **在发布 - 订阅模型中，如果只有一个订阅者，那它和队列模型就基本是一样的了。所以说，发布 - 订阅模型在功能层面上是可以兼容队列模型的。**
>
> 
>
>
> 补充：
>
> 当生产者把数据丢进topic时，写在partition上，那partition是怎么将其持久化的呢？Kafka是将partition的数据写在**磁盘**的(消息日志)，不过Kafka只允许**追加写入**(顺序访问)，避免缓慢的随机 I/O 操作。Kafka也不是partition一有数据就立马将数据写到磁盘上，它会先**缓存**一部分，等到足够多数据量或等待一定的时间再批量写入(flush)。

14. 经验智力题和经典算法题

1. **有一个100万（N）数据，怎么快速的查找到前最大的100（n）个数?**

> 1. 先取出前100个数，维护一个100个数的最小堆，遍历一遍剩余的元素。
>
>   （1）取前n个元素，建立一个小顶堆。时间复杂度为O(nlogn)，为堆排序的时间复杂度。
>
>   （2）顺序读取后续元素，每次读取一个元素，如果该元素比堆顶元素小，直接丢弃。 否则，如果大于堆顶元素，则用该元素替换堆顶元素，然后保持最小堆性质。时间复杂度为：O(N-n)*O(logn);
>
>   总的时间复杂度为O(Nlongn) 。
> 2. 根据快速排序划分的思想（快速选择），将所有数划分成大于某个数和小于某个数的区间。每次在符合条件的区间再次进行划分，直到正好100个数。时间复杂度为O(N*n)
> 3. 分块查找：将这100万的数字，平分为100份，从每一份中取出最大的100个数字；将这1万个数字组合在一起，找到最大的100个数。

2. **海量日志数据，提取出某日访问百度次数最多的K个IP。**（海量数据找重复次数最多的个数）

> 1. 按照IP地址的 Hash(IP) mod 1000，把海量IP日志分别存储到1000个小文件中。
> 2. 对于每一个小文件，可以构建一个IP为key，出现次数为value的HashMap，同时记录前K个次数最多的IP地址；
> 3. 如果n = 1，可以得到1000个小文件中的出现次数最多的IP，再依据常规的排序算法得到总体上出现次数最多的IP；
> 4. 如果n > 1，将 1000 * K 条结果，存储到一个文件中，对其进行排序，得到最多的K个IP。
> 5. 或最后借助堆这个数据结构，找出Top K。

3. **给定a、b两个文件，各存放50亿个url，每个url各占64字节，内存限制是4G，让你找出a、b文件共同的url?**

> 1. 遍历文件a，对每个url求取 hash(url) % 1000，然后根据所取得的值将url分别存储到1000个小文件。
> 2. 遍历文件b，对每个url求取 hash(url) % 1000，然后根据所取得的值将url分别存储到1000个小文件。
> 3. 求每对小文件中相同的url时，可以把其中一个小文件的url存储到hash_set中。然后遍历另一个小文件的每个url，看其是否在刚才构建的hash_set中，如果是，那么就是共同的url，存到文件里面就可以了。

4. **给40亿个不重复的unsigned int的整数，没排过序的，然后再给一个数，如何快速判断这个数是否在那40亿个数当中?**

> 使用位图Bit-Map，读入40亿个数，设置相应的bit位，读入要查询的数，查看相应bit位是否为1，为1表示存在，为0表示不存在。

5. **如何使用2G内存对10G数据进行排序——外部排序算法**

6. **部门工资最高的员工**

> ```
> SELECT Employee.Name AS Department, Department.Name AS Employee, Salary
> FROM Employee, Department
> WHERE Employee.DepartmentId = Department.Id
>  AND (DepartmentId, Salary) IN (
>      SELECT DepartmentId, MAX(Salary)
>      FROM Employee
>      GROUP BY DepartmentId)
> ```

7. **一个深度为5的满二叉树的节点个数有多少个?**

> 每一层有2^{i-1}个节点。
>
> 满二叉树的总节点个数为$2^i - 1$

8. **如何判断一个单链表是不是有环?**

> 快慢指针实现。

9. **a = 1， b = 4， 如何交换，不能使用第三个变量。**

> a = a + b
>
> b = a - b
>
> a = a - b

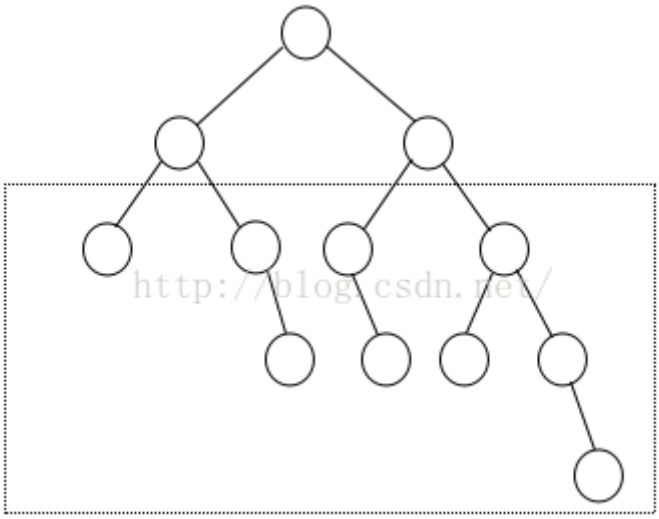
10. **有两个杯子，一个杯子的容量是3L，一个杯子的容量是5L，如何得到4L的水**

> 反过来想，首先4L的水肯定是在5L的杯子里面的到。4L的水可以通过3L + 1L得到，也可以通过5L - 1L得到。
>
> （1）1L的水怎么得到呢？5L的杯子和3L的水可以凑出2L的空间。那么在把3L的水倒2L到5L杯子中，最后就只剩下1L的水了。
>
> 得到了1L的水，再加上3L的水便有了4L的水。
>
> （2）用方程进行量化：3x + 5y = 4。用 3L 和 5L 的水桶装出 4L 水,可以, 3+3-5+3=4。也就是说3L的杯子需要装满三次，需要倒出一次5L的水。

11. **红黑树和平衡二叉树?**

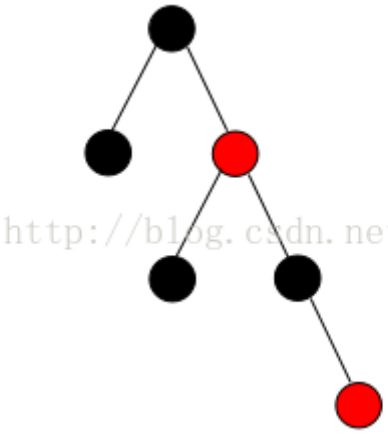
> 排序二叉树虽然可以快速检索，但在最坏的情况下：如果插入的节点集本身就是有序的，要么是由小到大排列，要么是由大到小排列，那么最后得到的排序二叉树将变成链表：所有节点只有左节点（如果插入节点集本身是大到小排列）；或所有节点只有右节点（如果插入节点集本身是小到大排列）。在这种情况下，排序二叉树就变成了普通链表，其检索效率就会很差。
>
> 红黑树
>
> - 性质 1：节点非红即黑。
> - 性质 2：根节点永远是黑色的。
> - 性质 3：所有的叶节点都是空节点（即 null），并且是黑色的。
> - 性质 4：每个红色节点的两个子节点都是黑色。（从每个叶子到根的路径上不会有两个连续的红色节点）
> - 性质 5：从任一节点到其子树中每个叶子节点的路径都包含相同数量的黑色节点。
>
> 红黑树最重要的性质：**从根到叶子的最长的可能路径小于等于最短的可能路径的两倍长。**
>
> 红黑树并不是真正意义上的平衡二叉树，但在实际应用中，红黑树的统计性能要高于平衡二叉树，但极端性能略差。(对于AVL树，任何一个节点的两个子树高度差不会超过1；对于红黑树，则是不会相差两倍以上)
>
> 对于给定的黑色高度为 N 的红黑树，从根到叶子节点的最短路径长度为 N-1，最长路径长度为 2 * (N-1)。对于红黑树，插入，删除，查找的复杂度都是O(log N)。任何不平衡都会在3次旋转之内解决。
>
> 红黑树通过上面这种限制来保证它大致是平衡的——因为红黑树的高度不会无限增高，这样保证红黑树在最坏情况下都是高效的，不会出现普通排序二叉树的情况。
>
> 由于红黑树只是一个特殊的排序二叉树，因此对红黑树上的只读操作与普通排序二叉树上的只读操作完全相同，只是红黑树保持了大致平衡，因此检索性能比排序二叉树要好很多。
>
> 但在红黑树上进行插入操作和删除操作会导致树不再符合红黑树的特征，因此插入操作和删除操作都需要进行一定的维护，以保证插入节点、删除节点后的树依然是红黑树。

> 平衡二叉树的最差情形：
>
> 由平衡二叉树的定义可知，左子树和右子树最多可以相差1层高度，那么多个在同一层的子树就可以依次以相差1层的方式来递减子树的高度，如下图所示是一个拥有4棵子树的树的层高最大差情形。也就是说，一颗高度为H的平衡二叉树，其内部子树高度差最多为[H / 2]。



红黑树的最差情形：

红黑树中红节点的父亲和孩子必须是黑节点，且从根到叶子节点经过的黑节点个数相同，因此**红黑树最小深度是路径上只有黑节点，最大深度是路径上红黑节点相互间隔(重要)，因此最大深度 <= 最小深度的两倍**，最大深度是2 * log2（n+1）。



> 对于AVL树，任何一个节点的两个子树高度差不会超过1；对于红黑树，则是不会相差两倍以上
>
> 红黑树的插入删除元素的效率高于平衡二叉树，而查询时间差于平衡二叉树。红黑树的树高可能更高。

12. **如何实现LRU算法?**

(1) 分析 Operations
1. 首先最基本的操作就是能够从里面读信息；
2. 那还得能加入新的信息，新的信息进来就是 most recently used 了；
3. 在加新信息之前，还得看看有没有空位，如果没有空间了，得先把老的踢出去，那就需要能够找到那个老家伙并且删除它；
4. 那如果加入的新信息是缓存里已经有的，那意思就是 key 已经有了，要更新 value，那就只需要调整一下这条信息的 priority，它已经从上一次被使用升级为最新使用的了。

(2) 找寻数据结构
1. 第一个操作很明显，我们需要一个能够快速查找的数据结构，非 HashMap 莫属，可是后面的操作 HashMap 就不行了。
2. LinkedList，按照从老到新的顺序，删除、插入、移动，都可以是 O(1) 的！但是删除时还需要一个 previous pointer 才能删掉，所以我需要一个 Doubly List.

数据结构： HashMap + DoublyList（双向链表）

(3) 定义清楚数据结构的内容

选好了数据结构之后，还需要定义清楚每个数据结构具体存储的是什么，这两个数据结构是如何联系的，这才是核心问题。

1. 读信息，直接利用HashMap读取Answer，时间复杂度O（1）。

2. 加入一组新的数据，如果没有这个Key，加进来，添加到链表的首部；如果已经有这个Key，HashMap这里要更新一下Value，还需要吧该节点移动到链表的首部，因为最新被使用了。

为了达到更新链表的操作，需要记录节点的位置。因此HashMap中不是直接存放的Value，而是存放一个记录Value的节点指针。得到了节点自然也就能得到value。

之后我们更新、移动每个节点时，它的 reference 也不需要变，所以 HashMap也不用改动，动的只是当前节点的指针指向pre， next.

最后的数据机构如下：

```
HashMap                Doubly List
Key    =>  Node    =>    Value, pre, next
```

Java 中的 LinkedHashMap 已经做了很好的实现。

(4) 6. 总结

1. 第一个操作，get() API，直接读取并更新节点在链表中位置即可；

2，第二个操作，put() API，画图的时候边讲边写，每一步都从 high level 到 detail 再到代码，把代码模块化。

```
put() => 有 Key => 更新Value, 更新节点在链表中的位置
      => 无 Key => 有空位置么 => 有    => appendHead()
                              => 没有 => remove() + appendHead()
```

```
public class LRUCacheDoublyList {

 /**
     * Your LRUCache object will be instantiated and called as such:
     * LRUCache obj = new LRUCache(capacity);
     * int param_1 = obj.get(key);
     * obj.put(key,value);
     */

    // HashMap: <key = Question, value = ListNode>
    // LinkedList: <Answer>

    class Node {
        int key;
        int value;
        Node pre;
        Node next;

        public Node(int key, int value) {
            this.key = key;
            this.value = value;
        }
    }

    private HashMap<Integer, Node> hashMap;
    private Node head;
    private Node tail;
    private int cap;

    public LRUCacheDoublyList(int capacity) {
        hashMap = new HashMap<>(capacity);
        cap = capacity;
    }

    public int get(int key) {
        Node node = hashMap.get(key);
        if (node == null) {
```

```
            return -1;
        } else {
            int val = node.value;
            remove(node);
            appendHead(node);
            return val;
        }
    }

    public void put(int key, int value) {
        Node node = hashMap.get(key);
        if (node == null) {
            node = new Node(key, value);
            if (hashMap.size() >= cap) {
                //为什么要在Node中记录key的原因是：删除HashMap中的reference
                hashMap.remove(tail.key);
                remove(tail);
            }
            appendHead(node);
            hashMap.put(key, node);
        } else {
            node.value = value;
            remove(node);
            appendHead(node);
        }
    }

    private void remove(Node curr) {
        if(head == tail) {
            head = tail = null;
        } else if (curr == head) {
            head = curr.next;
            head.pre = null;
            curr.next = null;
        } else if (curr == tail) {
            tail = curr.pre;
            tail.next = null;
            curr.pre = null;
        } else {
            curr.pre.next = curr.next;
            curr.next.pre = curr.pre;
            curr.next = null;
            curr.pre = null;
        }
    }

    private void appendHead(Node curr) {
        if (head == null && tail == null) {
            head = curr;
            tail = curr;
        } else {
            head.pre = curr;
            curr.next = head;
            head = curr;
        }
    }
}
```

13. **手撕代码，二叉树的先序和中序重建二叉树，最后再输出后续遍历结果。**
14. **手撕代码，0～n-1中缺失的数字?**

面试官没有提出要求时，需要询问是有序的还是无序的，如果是有序的话可以使用二分查找，时间复杂度O(logn)，如果是无序的数组的话，可以使用位运算或加和的方式解决问题，时间复杂度O(n)。

```
public class Main {
 public int missingNumber(int[] nums) {
     if (nums.length == 0) {
         return 0;
     }
     int left = 0;
     int right = nums.length - 1;
     while (left <= right) {
         int mid = left + ((right - left) >> 1);
         if (nums[mid] != mid) {
             right = mid - 1;
         } else {
             left = mid + 1;
         }
     }
     //注意这个位置为left，nums[left]为第一个 nums[left] ! = left的数，其前一个为所求答案正好
是left
     return left;
 }
 public class Main {

 public int missingNumber(int[] nums) {
     int res = nums.length;
     for (int i = 0; i < nums.length; i++) {
         res ^= nums[i] ^ i;
     }
     return res;
 }

 public int missingNumber(int[] nums) {
     int res = 0;
     for (int i = 0; i < nums.length; i++) {
         res += i - nums[i];
     }
     res = nums.length + res;
     return res;
 }
}
}
```

15. **二叉搜索树的第k大节点?**

```
class Solution {
    public int kthLargest(TreeNode root, int k) {
        int count = 1;
        Deque<TreeNode> stack = new LinkedList<>();
        while (root != null || !stack.isEmpty()) {
            while (root != null) {
                stack.addLast(root);
                root = root.right;
            }
            TreeNode pop = stack.pollLast();
            if (count == k) {
                return pop.val;
            }
            count++;
            root = pop.left;
```

```
        }
        return 0;
    }
}
```

16. **给定一个链表，删除链表的倒数第 N 个节点，并且返回链表的头结点。**

```
//我们可以使用两个指针而不是一个指针。第一个指针从列表的开头向前移动 n+1 步，而第二个指针将从列表的开头出发。现在，这两个指针被 n 个结点分开。我们通过同时移动两个指针向前来保持这个恒定的间隔，直到第一个指针到达最后一个结点。此时第二个指针将指向从最后一个结点数起的第 n 个结点。我们重新链接第二个指针所引用的结点的 next 指针指向该结点的下下个结点。
public ListNode removeNthFromEnd(ListNode head, int n) {
 ListNode dummy = new ListNode(0);
 dummy.next = head;
 ListNode first = dummy;
 ListNode second = dummy;
 // Advances first pointer so that the gap between first and second is n nodes apart
 for (int i = 0; i <= n; i++) {
     first = first.next;
 }
 // Move first to the end, maintaining the gap
 while (first != null) {
     first = first.next;
     second = second.next;
 }
 second.next = second.next.next;
 return dummy.next;
}
```

17. **微信抢红包设计**

18. **两根香，一根烧完1小时，如何测量15分钟**

> 先将一根香的一端点燃，另一根香的两端全部点燃。当第二根香全部烧完时，此时已经过了半个小时。再将第一根香的另一端也点燃，那么此时第一根香剩下部分烧完的时间就是 15 min。

19. **有 25 匹马和 5 条赛道，赛马过程无法进行计时，只能知道相对快慢。问最少需要几场赛马可以知道前 3 名。**

> 先把 25 匹马分成 5 组，进行 5 场赛马，得到每组的排名。再将每组的第 1 名选出，进行 1 场赛马，按照这场的排名将 5 组先后标为 A、B、C、D、E。可以知道，A 组的第 1 名就是所有 25 匹马的第 1 名。而第 2、3 名只可能在 A 组的 2、3 名，B 组的第 1、2 名，和 C 组的第 1 名，总共 5 匹马，让这 5 匹马再进行 1 场赛马，前两名就是第 2、3 名。所以总共是 **5+1+1=7** 场赛马。

20. **有 9 个球，其中 8 个球质量相同，有 1 个球比较重。要求用 2 次天平，找出比较重的那个球。**

> 将这些球均分成 3 个一组共 3 组，选出 2 组称重，如果 1 组比较重，那么重球在比较重的那 1 组；如果 1 组重量相等，那么重球在另外 1 组。对比较重的那 1 组的 3 个球再分成 3 组，重复上面的步骤。

21. **两个线程交替打印两个数组中的元素**

```
public class PrintArray {
 public static void main(String[] args) {
     char[] digit = new char[]{'1','2','3','4','5','6','7','8','9'};
     char[] letter = new char[]{'A','B','C','D','E','F','G','H','I'};

     ReentrantLock lock = new ReentrantLock();
     Condition condition1 = lock.newCondition();
     Condition condition2 = lock.newCondition();

     Thread t1 = new Thread(() -> {
```

```
            lock.lock();
            try {
                for (int i = 0; i < digit.length; i++) {
                    System.out.println(digit[i]);
                    condition2.signal();
                    condition1.await();
                }
                condition2.signal();
            } catch (InterruptedException e) {
                e.printStackTrace();
            } finally {
                lock.unlock();
            }
        }, "t1");

        Thread t2 = new Thread(() -> {
            lock.lock();
            try {
                for (int i = 0; i < letter.length; i++) {
                    System.out.println(letter[i]);
                    condition1.signal();
                    condition2.await();
                }
                condition1.signal();
            } catch (InterruptedException e) {
                e.printStackTrace();
            } finally {
                lock.unlock();
            }
        }, "t2");

        t1.start();
        t2.start();
    }
}
```

```
public class PrintArray {
    public static void main(String[] args) {
        final Object obj = new Object();
        char[] number = {'1','2','3','4','5','6','7','8','9'};
        char[] letter = {'A','B','C','D','E','F','G','H','I'};

        new Thread(() -> {
            synchronized (obj){
                for(char num : number){
                    System.out.print(num + " ");
                    try {
                        obj.notify(); //叫醒其他线程，这里就是t2
                        obj.wait(); //让自己阻塞，让出锁
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
                obj.notify(); //必须要有，因为两个线程的try里面的最后一步是阻塞，如果线程执行完了还在阻塞肯定不对，必须要唤醒，才能正确结束程序
            }
        }).start();

        new Thread(() -> {
            synchronized (obj){
```

```
            for(char let : letter){
                System.out.print(let + " ");
                try {
                    obj.notify(); //叫醒其他线程，这里是t1
                    obj.wait(); //让自己阻塞，让出锁
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
            obj.notify(); //同上
        }
    }).start();
   }
}
```

22. **创建3个线程，t1、t2、t3，让让t1在t2之前执行，t2在t3之前执行**

```
public class Join {
 public static void main(String[] args) {

    Thread t1 = new Thread(() -> {
        System.out.println(Thread.currentThread().getName());
    }, "1");

    Thread t2 = new Thread(() -> {
        try {
            t1.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println(Thread.currentThread().getName());
    }, "2");

    Thread t3 = new Thread(() -> {
        try {
            t2.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println(Thread.currentThread().getName());
    }, "3");

    t1.start();
    t2.start();
    t3.start();
 }
}
```

23. **三个线程交替打印ABC**

```
public class PrintAbcByOrder {
 public static void main(String[] args) {
    Print p = new Print();
    new Thread(() -> {
        for (int i = 0; i < 5; i++) {
            p.printA();
        }
    }, "A").start();
```

```
        new Thread(() -> {
            for (int i = 0; i < 5; i++) {
                p.printB();
            }
        }, "B").start();
        new Thread(() -> {
            for (int i = 0; i < 5; i++) {
                p.printC();
            }
        }, "C").start();
    }
}

class Print {
    private ReentrantLock lock = new ReentrantLock();
    Condition a = lock.newCondition();
    Condition b = lock.newCondition();
    Condition c = lock.newCondition();
    private int statue = 1;

    public void printA() {
        lock.lock();
        try {
            while (statue != 1) {
                a.await();
            }
            System.out.println("A");
            statue = 2;
            b.signal();
        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            lock.unlock();
        }
    }

    public void printB() {
        lock.lock();
        try {
            while (statue != 2) {
                b.await();
            }
            System.out.println("B");
            statue = 3;
            c.signal();
        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            lock.unlock();
        }
    }

    public void printC() {
        lock.lock();
        try {
            while (statue != 3) {
                c.await();
            }
            System.out.println("C");
            statue = 1;
            a.signal();
```

```
        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            lock.unlock();
        }
    }
}
```

24. **生产者消费者模式**

```
public class ProducerAndComsumer {
    public static void main(String[] args) {
        Repository repository = new Repository();
        Thread t1 = new Thread(new Producer(repository), "t1");
        Thread t2 = new Thread(new Producer(repository), "t2");
        Thread t3 = new Thread(new Producer(repository), "t3");
        Thread t4 = new Thread(new Consumer(repository), "t4");
        Thread t5 = new Thread(new Consumer(repository), "t5");
        Thread t6 = new Thread(new Consumer(repository), "t6");
        t1.start();
        t2.start();
        t3.start();
        t4.start();
        t5.start();
        t6.start();
    }
}

class Repository {
    int capacity = 10;
    LinkedList<Object> queue = new LinkedList<>();

    public void produce() {
        synchronized (queue) {
            if (queue.size() == capacity) {
                try {
                    queue.wait();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
            queue.offer(new Object());
            queue.notifyAll();
        }
    }

    public void consume() {
        synchronized (queue) {
            if (queue.size() == 0) {
                try {
                    queue.wait();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
            queue.poll();
            queue.notifyAll();
        }
    }
}
```

```
class Producer implements Runnable {

 Repository repository;

 public Producer(Repository repository) {
     this.repository = repository;
 }

 @Override
 public void run() {
     while (true) {
         try {
             Thread.sleep(2000);
             repository.produce();
         } catch (InterruptedException e) {
             e.printStackTrace();
         }
     }
 }
}

class Consumer implements Runnable {

 Repository repository;

 public Consumer(Repository repository) {
     this.repository = repository;
 }

 @Override
 public void run() {
     while (true) {
         try {
             Thread.sleep(2000);
             repository.consume();
         } catch (InterruptedException e) {
             e.printStackTrace();
         }
     }
 }
}
```

25. **随机数产生转换（1~5到1~7）**

> 题目
> 给定一个随机数生成器，这个生成器能均匀生成1到5（1,5）的随机数，如何使用这个生成器生成均匀分布的1到7（1,7）的数?
>
> 思路
> 方法一：生成两个（1,5）的随机数，这样一共是25种情况，注意这两个数是有顺序的，从这25种情况中，取前21种，每三种代表（1,7）中的一个数字，如果取到的是这21种以外的情况，丢掉重新取。
>
> 方法二：生成三个（1,5）的随机数，分别表示一个二进制位，其中1和2映射为0，3跳过，4和5映射为1。这样产生的三位二进制数，即1-8这8个数字都是等概率的。如果产生的是8，那么丢弃即可。
>
> 方法三：生成两个（1,5）的随机数，产生一个两位的五进制数，5 * (random5() – 1) + random5()。这个公式能够等概率产生1-25，即第一个随机数代表：0,5,10,15,20，地位代表1,2,3,4,5。这样对这个数字（1-25的数字），采用方法一的方法，只用1-21，分7分，代表1-7,22-25这4个数字扔掉。

26. **二叉树的最小深度和二叉树的最大深度?**

```
//二叉树的最小深度
public class Main {
 public int minDepth(TreeNode root) {
     if (root == null) {
         return 0;
     }
     if (root.left == null && root.right == null) {
         return 1;
     }
     int min = Integer.MAX_VALUE;
     if (root.left != null) {
         min = Math.min(min, minDepth(root.left));
     }
     if (root.right != null) {
         min = Math.min(min, minDepth(root.right));
     }
     return min + 1;
 }

}
```

```
//二叉树的最大深度
public class Main {
 public int maxDepth(TreeNode root) {
     if (root == null) {
         return 0;
     }
     int left = maxDepth(root.left);
     int right = maxDepth(root.right);
     return left > right ? left + 1 : right + 1;
 }
}
```

27. **实现sqrt()**

```
class Solution {
 public int mySqrt(int x) {
     if (x == 0 || x == 1) {
         return x;
     }
     int left = 1;
     int right = x;
     while (left <= right) {
         int mid = left + ((right - left) >> 1);
         if (x / mid == mid) {
             return mid;
         } else if (x / mid > mid) {
             left = mid + 1;
         } else {
             right = mid - 1;
         }
     }
     return right;
 }
}
```

28. **已知 sqrt (2)约等于 1.414，要求不用数学库，求 sqrt (2)精确到小数点后 10 位。**

```
class Solution{
```

```
    double sqrt2() {
    double low = 1.4, high = 1.5;
    double mid = (low + high) / 2;

    while (high - low > EPSILON) {
        if (mid * mid > 2) {
            high = mid;
        } else {
            low = mid;
        }
        mid = (high + low) / 2;
    }

    return mid;
    }
}
```

29. **二叉树的前中后序遍历的非递归实现。**

```
public class BinaryTree {

 class TreeNode {
     int val;
     TreeNode left;
     TreeNode right;
     TreeNode(int val) {
         this.val = val;
     }
 }

 /**
     * 从当前节点开始遍历：（当入栈时访问节点内容，则为前序遍历；出栈时访问，则为中序遍历）
     * 1. 若当前节点存在，就存入栈中，并访问左子树；
     * 2. 直到当前节点不存在，就出栈，并通过栈顶节点访问右子树；
     * 3. 不断重复12，直到当前节点不存在且栈空。
     * 只需要在入栈、出栈的时候，分别进行节点访问输出，即可分别得到前序、中序的非递归遍历代码！
     */

    public void preOrder(TreeNode root, ArrayList<Integer> ans) {
        if (root == null) {
            return;
        }
        LinkedList<TreeNode> stack = new LinkedList<>();
        TreeNode cur = root;
        while (cur != null || !stack.isEmpty()) {
            while (cur != null) {
                stack.offerLast(cur);
                ans.add(cur.val);
                cur = cur.left;
            }
            cur = stack.pollLast();
            cur = cur.right;
        }
    }

    public void inOrder(TreeNode root, ArrayList<Integer> ans) {
        if (root == null) {
            return;
        }
        LinkedList<TreeNode> stack = new LinkedList<>();
```

```
        TreeNode cur = root;
        while (cur != null || !stack.isEmpty()) {
            while (cur != null) {
                stack.offerLast(cur);
                cur = cur.left;
            }
            cur = stack.pollLast();
            ans.add(cur.val);
            cur = cur.right;
        }
    }


    /**
     * 从当前节点开始遍历：
     * 1. 若当前节点存在，就存入栈中，第一次访问，然后访问其左子树；
     * 2. 直到当前节点不存在，需要回退，这里有两种情况：
     *   1）从左子树回退，通过栈顶节点访问其右子树（取栈顶节点用，但不出栈）
     *   2）从右子树回退，这时需出栈，并取出栈节点做访问输出。（需要注意的是，输出完毕需要置当前节点为空，以便继续回退。具体可参考代码中的 cur = NULL）
     * 3. 不断重复12，直到当前节点不存在且栈空。
     */
    public void postOrder(TreeNode root, ArrayList<Integer> ans) {
        if (root == null) {
            return;
        }
        LinkedList<TreeNode> stack = new LinkedList<>();
        TreeNode cur = root;
        TreeNode pre = null;
        while (cur != null || !stack.isEmpty()) {
            while (cur != null) {
                stack.offerLast(cur);
                cur = cur.left;
            }
            cur = stack.peekLast();
            if (cur.right == null || cur.right == pre) {
                ans.add(cur.val);
                stack.pollLast();
                pre = cur;
                cur = null;
            } else {
                cur = cur.right;
            }
        }
    }

    public List<List<Integer>> levelOrder(TreeNode root) {
        if (root == null) {
            return new ArrayList<>();
        }
        List<List<Integer>> ans = new LinkedList<>();
        LinkedList<TreeNode> queue = new LinkedList<>();
        queue.offerLast(root);
        while (!queue.isEmpty()) {
            List<Integer> levelAns = new LinkedList<>();
            for (int i = queue.size(); i > 0; i--) {
                TreeNode cur = queue.pollFirst();
                levelAns.add(cur.val);
                if (cur.left != null) {
                    queue.offerLast(cur.left);
                }
```

```
                if (cur.right != null) {
                    queue.offerLast(cur.right);
                }
            }
            ans.add(levelAns);
        }
        return ans;
    }

}
```

30. **实现一个只包含+，-，*，/的数学运算**

```
//自己后来写的，将就看吧。模拟操作数栈
public class Calculate {
 public static void main(String[] args) {
     System.out.println(calculate("1+2*3*3/5+4/2"));
 }
 public static int calculate(String str) {
     LinkedList<Integer> numStack = new LinkedList<>();
     LinkedList<Character> opStack = new LinkedList<>();
     for (int i = 0; i < str.length(); i++) {
         if (str.charAt(i) >= '0' && str.charAt(i) <= '9') {
             if (!opStack.isEmpty() && (opStack.peekLast() == '*' || opStack.peekLast()
== '/')) {
                 int a = numStack.pollLast();
                 int b = str.charAt(i) - '0';
                 if (opStack.peekLast() == '*') {
                     numStack.offerLast(a * b);
                 } else {
                     numStack.offerLast(a / b);
                 }
                 opStack.pollLast();
             } else {
                 numStack.offerLast(str.charAt(i) - '0');
             }
         } else {
             opStack.offerLast(str.charAt(i));
         }

     }
     while (!opStack.isEmpty()) {
         char opt = opStack.pollFirst();
         int a = numStack.pollFirst();
         int b = numStack.pollFirst();
         if (opt == '+') {
             numStack.offerFirst(a + b);
         } else {
             numStack.offerFirst(a - b);
         }
     }
     return numStack.pollLast();
 }
}
```

31. **字典树的实现。**

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。**

```
class Trie {
```

```
    private TrieNode root;

    /** Initialize your data structure here. */
    public Trie() {
        root = new TrieNode();
    }

    /** Inserts a word into the trie. */
    public void insert(String word) {
        if (word == null || word.length() == 0) {
            return;
        }
        TrieNode cur = root;
        for (int i = 0; i < word.length(); i++) {
            char a = word.charAt(i);
            if (cur.getContent().get(a) == null) {
                TrieNode p = new TrieNode();
                cur.getContent().put(a, p);
                cur = p;
            } else {
                cur = cur.getContent().get(a);
            }
        }
        cur.setEnd(true);
    }

    /** Returns if the word is in the trie. */
    public boolean search(String word) {
        if (word == null || word.length() == 0) {
            return false;
        }
        TrieNode cur = root;
        for (int i = 0; i < word.length(); i++) {
            char a = word.charAt(i);
            if ((cur = cur.getContent().get(a)) == null) {
                return false;
            }
        }
        return cur.isEnd();
    }

    /** Returns if there is any word in the trie that starts with the given prefix. */
    public boolean startsWith(String prefix) {
        if (prefix == null || prefix.length() == 0) {
            return false;
        }
        TrieNode cur = root;
        for (int i = 0; i < prefix.length(); i++) {
            char a = prefix.charAt(i);
            if ((cur = cur.getContent().get(a)) == null) {
                return false;
            }
        }
        return true;
    }

    public static class TrieNode {
        private boolean isEnd;
        private HashMap<Character, TrieNode> content; //可以用数组，也可以用Hash表
```

```
        public TrieNode() {
            isEnd = false;
            content = new HashMap<>();
        }

        public boolean isEnd() {
            return isEnd;
        }

        public void setEnd(boolean end) {
            isEnd = end;
        }

        public HashMap<Character, TrieNode> getContent() {
            return content;
        }

        public void setContent(HashMap<Character, TrieNode> content) {
            this.content = content;
        }
    }
}
```

---

15. 高频算法题

1. 快排
2. 前缀树
3. 判断链表有环
4. 求环的入口
5. 最大公因数
6. 两个链表找交点
7. topK
8. LRU
9. 两个栈实现一个队列
10. 树层次遍历第一层从左往右，第二层从右向左
11. 最长公共子序列
12. 最长上升子序列
13. 最大子段和
14. 股票买卖系列
15. 合并k个链表
16. k个一组反转链表
17. 接雨水
18. 零钱兑换
19. 正则表达式匹配
20. 最长连续序列

```java
//给定一个未排序的整数数组，找出最长连续序列的长度。
//为什么用这个呢？因为这个时间复杂度是O(n)，空间复杂度是O(n)，典型的时间换空间问题。为什么不用直接排序再找连续序列呢？因为直接排序的时间复杂度是O(nlogn)。我他嘛已经两次倒在这个问题上了，每次给的都是时间复杂度最优的算法，但面试官问问为啥要弄得这么复杂干嘛，直接排个序不就完了嘛？？？既然这样，非要说个时间复杂度的比较才行，我还是太菜了，一下子没反应过来是要对方法进行比较。一次字节，一次阿里。看来是注定无缘了吧。
class Solution {
    public int longestConsecutive(int[] nums) {
        if (nums == null || nums.length == 0) {
            return 0;
        }
        Set<Integer> numSet = new HashSet<>();
        for (int i = 0; i < nums.length; i++) {
            numSet.add(nums[i]);
        }
        int ans = 0;
        for (int i = 0; i < nums.length; i++) {
            int num = nums[i];
            if (numSet.contains(num)) {
                int count = 1;
                int left = num - 1;
                int right = num + 1;
                numSet.remove(num);
                while (numSet.contains(left)) {
                    numSet.remove(left);
                    left--;
                    count++;
                }
                while (numSet.contains(right)) {
                    numSet.remove(right);
                    right++;
                    count++;
                }
                ans = Math.max(ans, count);
            }
        }
        return ans;

    }
}
```

---

16. 其他

1. **工作想 base 在哪里？为什么？**
2. **平时有什么兴趣爱好？**
3. **自己未来有什么规划？**
4. **平时是如何学习新技术的？（官网/书籍/博客/视频）**
5. **一般遇到问题如何解决？（Google 和 Stackoverflow）**
6. **介不介意加班？**
7. **印象最深的事情是什么？**
8. **运维开发是做什么的？**

> 运维工程师的工作核心主要是保障产品上线后的稳定运行，对在此期间出现的各种问题进行快速解决，并在日常工作中不断优化系统架构和部署的合理性，以提升系统服务。可能会开发一些日志数据的统计工具对系统的相关运行情况进行监控和分析。

9. **你有什么问题想问我？（重要）**

> 通用回答：
>
> （1）这是哪个部门，这个部门正在做什么，如果我进来我会接触到什么。
>
> （2）您认为我在哪些方面可以提高自己。
>
> > 1. 面对 HR 或者其他 Level 比较低的面试官时，你可以问：
> >
> > 能不能谈谈你作为一个公司老员工对公司的感受?
> >
> > 能不能问一下，你当时因为什么原因选择加入这家公司的呢或者说这家公司有哪些地方吸引你?
> >
> > 有什么地方你觉得还不太好或者可以继续完善吗？
> >
> > 我觉得我这次表现的不是太好，您有什么建议或者评价给我吗？
> >
> > 接下来我会有一段空档期，有什么值得注意或者建议学习的吗？
>
> > 2. 面对部门领导时，你可以问：
> >
> > 部门的主要人员分配以及对应的主要工作能简单介绍一下吗？
> >
> > 未来如果我要加入这个团队，你对我的期望是什么？
> >
> > 公司对新入职的员工的培养机制是什么样的呢？
> >
> > 以您来看，这个岗位未来在公司内部的发展如何？
> >
> > 团队现在面临的最大挑战是什么？
> >
> > 3. 面对 Level 比较高的(比如总裁，老板)时，你可以问：
> >
> > 贵公司的发展目标和方向是什么？
> >
> > 与同行业的竞争者相比，贵公司的核心竞争优势在什么地方？
> >
> > 公司现在面临的最大挑战是什么？