华为 9/6

笔试题

1、报文转换（十进制十六进制转换）

```
import java.util.Scanner;
public class Solution1 {
    public static void trans(int len, String[] str) {
        int count = len;
        StringBuilder sb = new StringBuilder();
        for(int i = 1; i < len; i++) {
            if(str[i].equals("A")) {
                sb.append("12 34");
                count++;
            } else if(str[i].equals("B")) {
                sb.append("AB CD");
                count++;
            } else {
                sb.append(str[i]);
            }
            if(i != len - 1) {
                sb.append(" ");
            }
        }
        String length = Integer.toHexString(count);
        length = length.toUpperCase();
        System.out.println(length + " " + sb.toString());
    }
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        String[] str = in.nextLine().split(" ");
        int len = Integer.parseInt(str[0], 16);
        trans(len, str);
    }
}
```

2、质数

```
import java.util.Scanner;
public class Solution2 {
    public static boolean iszhishu(int n) {
        if(n < 2) {
            return false;
        }
        for(int i = n - 1; i >= 2; i--) {
            if(n % i == 0) {
                return false;

            }
```

```
        }
        return true;
    }

    public static int deal(int low, int high) {
        int sumshi = 0;
        int sumge = 0;
        for(int i = low; i < high; i++) {
            if(iszhishu(i)) {
                sumge += i % 10;
                sumshi += (i % 100 - (i % 10)) / 10;
            }
        }
        if(sumge <= sumshi) {
            return sumge;
        } else {
            return sumshi;
        }
    }

    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int low = in.nextInt();
        int high = in.nextInt();
        System.out.println(deal(low, high));
    }
}
```

3、收到消息的人数

```
import java.util.*;
public class Solution3 {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        String name = in.nextLine();
        int len = Integer.parseInt(in.nextLine());
        String[] groups = new String[len];
        for(int i = 0; i < len; i++) {
            groups[i] = in.nextLine();
        }
        List[] lists = new List[groups.length];
        for(int i = 0; i < lists.length; i++) {
            lists[i] = Arrays.asList(groups[i].split(","));
        }
        Set<String> set = new HashSet<>();
        for(List list : lists) {
            if(list.contains(name)) {
                set.addAll(list);
            }
        }
        for(List list : lists) {

            for(int j = 0; j < list.size(); j++) {
```

```
                if(set.contains(list.get(j))) {
                    set.addAll(list);
                    break;
                }
            }
        }
        System.out.println(set.size());
    }
}
```

测试用例：
Jack
3
Jack,Tom,Anny,Lucy
Tom,Danny
Jack,Lily
输出结果：6

面经

1、操作系统学过吧，说说进程怎么调度？

不同环境的调度算法目标不同，因此需要针对不同环境来讨论调度算法。

批处理系统：**批处理系统没有太多的用户操作**，在该系统中，调度算法目标是**保证吞吐量和周转时间**（从提交到终止的时间）。

- **先来先服务 first-come first-serverd（FCFS）**：非抢占式的调度算法，**按照请求的顺序进行调度。** 有利于长作业，但不利于短作业，因为短作业必须一直等待前面的长作业执行完毕才能执行，而长作业又需要执行很长时间，造成了短作业等待时间过长。
- **短作业优先 shortest job first（SJF）**：非抢占式的调度算法，**按估计运行时间最短的顺序进行调度。**长作业有可能会饿死，处于一直等待短作业执行完毕的状态。因为如果一直有短作业到来，那么长作业永远得不到调度。
- **最短剩余时间优先 shortest remaining time next（SRTN）**：最短作业优先的抢占式版本，按剩余运行时间的顺序进行调度。 **当一个新的作业到达时，其整个运行时间与当前进程的剩余时间作比较。如果新的进程需要的时间更少，则挂起当前进程，运行新的进程。否则新的进程等待。**

交互式系统：**交互式系统有大量的用户交互操作，在该系统中调度算法的目标是快速地进行响应。**

- **时间片轮转**：将所有就绪进程按 FCFS 的原则排成一个队列，每次调度时，把 CPU 时间分配给队首进程，该进程可以执行一个时间片。当时间片用完时，由计时器发出时钟中断，调度程序便停止该进程的执行，并将它送往就绪队列的末尾，同时继续把 CPU 时间分配给队首的进程 。
- **优先级调度**：为每个进程分配一个优先级，按优先级进行调度。为了防止低优先级的进程永远等不到调度，可以随着时间的推移增加等待进程的优先级。
- **多级反馈队列**：一个进程需要执行 100 个时间片，如果采用时间片轮转调度算法，那么需要交换 100 次。多级队列是为这种需要连续执行多个时间片的进程考虑，它设置了多个队列，每个队列时间片大小都不同，例如 1,2,4,8,...。进程在第一个队列没执行完，就会被移到下一个队列。这种方式下，之前的进程只需要交换 7 次。每个队列优先权也不同，最上面的优先权最高。因此只有上一个队列没有进程在排队，才能调度当前队列上的进程。可以将这种调度算法看成是时间片轮转调度算法和优先级调度算法的结合。

实时系统：实时系统要求一个请求在一个确定时间内得到响应。分为硬实时和软实时，前者必须满足绝对的截止时间，后者可以容忍一定的超时。

2、进程线程的区别，进程间怎么相互通信，什么是多线程，什么是并发?

进程和线程的区别有以下几点：

- 拥有资源：**进程是资源分配的基本单位，但是线程不拥有资源**，线程可以访问隶属进程的资源。
- 调度：**线程是独立调度的基本单位**，在同一进程中，线程的切换不会引起进程切换，从一个进程中的线程切换到另一个进程中的线程时，会引起进程切换。
- 通信方面：**线程间可以通过直接读写同一进程中的数据进行通信，但是进程通信需要借助 IPC。**

进程间的通信方式：

- 管道：只支持半双工通信（单向交替传输）；**只能在父子进程或者兄弟进程中使用。**
- FIFO：也称为命名管道，**去除了管道只能在父子进程中使用的限制。**
- 消息队列：**消息队列可以独立于读写进程存在**，从而避免了 FIFO 中同步管道的打开和关闭时可能产生的困难；避免了 FIFO 的同步阻塞问题，不需要进程自己提供同步方法；**读进程可以根据消息类型有选择地接收消息，而不像 FIFO 那样只能默认地接收。**
- 信号量：它是一个计数器，用于**为多个进程提供对共享数据对象的访问。**
- 共享存储：允许多个进程共享一个给定的存储区。因为数据不需要在进程之间复制，所以这是最快的一种 IPC。 需要**使用信号量用来同步对共享存储的访问。**多个进程可以将同一个文件映射到它们的地址空间从而实现共享内存。另外 XSI 共享内存不是使用文件，而是使用内存的匿名段。
- 套接字：与其它通信机制不同的是，它**可用于不同机器间的进程通信。**

多线程就是指**一个进程中同时有多个执行路径正在执行。**

并发指在操作系统中，**一个时间段中有几个程序都已处于已启动运行到运行完毕之间，且这几个程序都是在同一个处理机上面，但任意时刻点上只有一个程序在处理机上运行。**

3、spring的IOC，用了哪些设计模式?

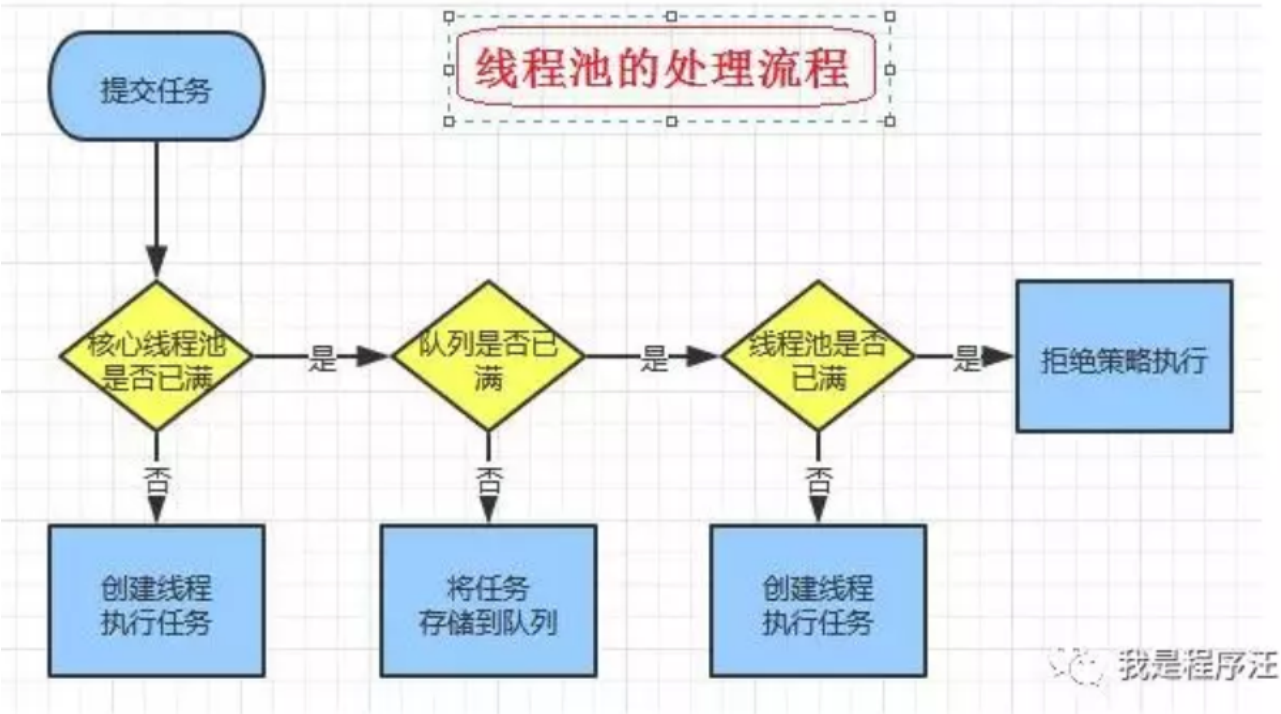
IoC叫控制反转，是Inversion of Control的缩写，**DI（Dependency Injection）叫依赖注入，是对IoC更简单的诠释。**控制反转是把传统上由程序代码直接操控的对象的调用权交给容器，通过容器来实现对象组件的装配和管理。所谓的**"控制反转"就是对组件对象控制权的转移，从程序代码本身转移到了外部容器，由容器来创建对象并管理对象之间的依赖关系。**IoC体现了好莱坞原则 - "Don’t call me, we will call you"。依赖注入的基本原则是应用组件不应该负责查找资源或者其他依赖的协作对象。配置对象的工作应该由容器负责，查找资源的逻辑应该从应用组件的代码中抽取出来，交给容器来完成。DI是对IoC更准确的描述，即组件之间的依赖关系由容器在运行期决定，形象的来说，即由容器动态的将某种依赖关系注入到组件之中。

一个类A需要用到接口B中的方法，那么就需要为类A和接口B建立关联或依赖关系，最原始的方法是在类A中创建一个接口B的实现类C的实例，但这种方法需要开发人员自行维护二者的依赖关系，也就是说当依赖关系发生变动的时候需要修改代码并重新构建整个系统。如果通过一个容器来管理这些对象以及对象的依赖关系，则只需要在类A中定义好用于关联接口B的方法（构造器或setter方法），将类A和接口B的实现类C放入容器中，通过对容器的配置来实现二者的关联。 依赖注入可以通过setter方法注入（设值注入）、构造器注入和接口注入三种方式来实现，Spring支持setter注入和构造器注入，通常使用构造器注入来注入必须的依赖关系，对于可选的依赖关系，则setter注入是更好的选择，setter注入需要类提供无参构造器或者无参的静态工厂方法来创建对象。

Spring 框架中使用到了大量的设计模式，下面列举了比较有代表性的：1、代理模式—在 AOP 和 remoting 中被用的比较多。2、单例模式：在 spring 配置文件中定义的 bean 默认为单例模式。3、模板模式：用来解决代码重复的问题。比如. RestTemplate, JmsTemplate, JpaTemplate。4、委派模式：Spring 提供了 DispatcherServlet 来对请求进行分发。5、工厂模式：BeanFactory 用来创建对象的实例，贯穿于 BeanFactory / ApplicationContext接口的核心理念。6、代理模式：AOP 思想的底层实现技术，Spring 中采用 JDK Proxy 和 CgLib 类库。

4、线程池的作用？

现在服务器端的应用程序几乎都采用了“线程池”技术，这主要是为了提高系统效率。因为**如果服务器对应每一个请求就创建一个线程的话，在很短的一段时间内就会产生很多创建和销毁线程动作，导致服务器在创建和销毁线程上花费的时间和消耗的系统资源要比花在处理实际的用户请求的时间和资源更多。**线程池就是为了尽量减少这种情况的发生。



> 线程池核心参数：
>
> ```
> <bean id="testExecutor"
>       class="org.springframework.scheduling.concurrent.ThreadPoolTaskExecutor">
> <!-- corePoolSize  -->
> <property name ="corePoolSize" value ="5" />
> <!-- 线程池维护线程的最大数量 -->
> <property name ="maxPoolSize" value ="10" />
> <!-- 线程池所使用的缓冲队列 -->
> <property name ="queueCapacity" value ="1000" />
> <!-- keepAliveSeconds  线程池维护线程所允许的空闲时间 -->
> <property name ="keepAliveSeconds" value ="3000" />
> <!-- 核心线程在空闲keepAliveSeconds后也timeout -->
> <property name ="allowCoreThreadTimeOut" value ="true" />
> <!-- 设置线程名称 -->
> <property name="threadNamePrefix" value="studyOrderTaskThreadPool-thread-"/>
> </bean>
> ```

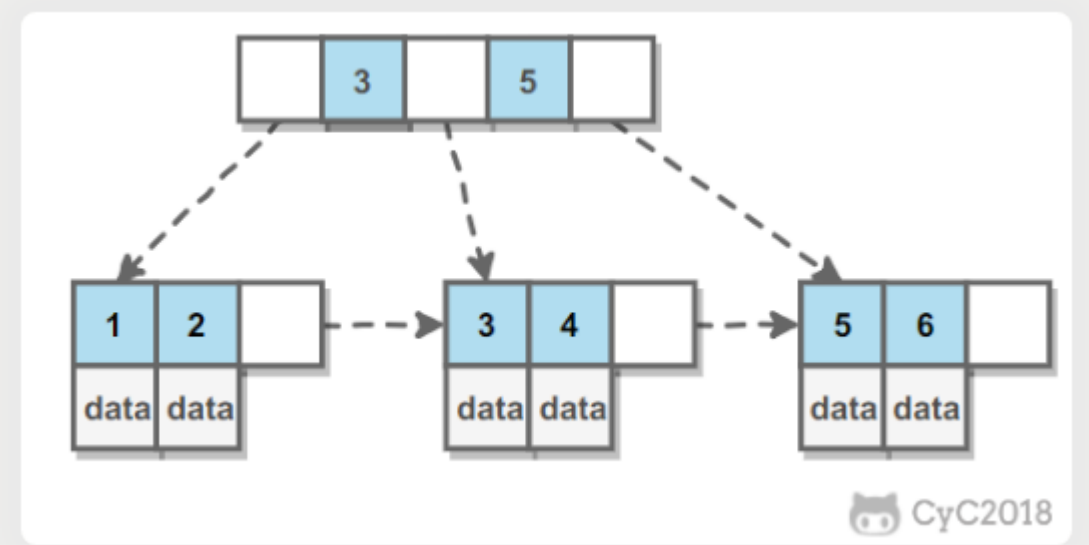
5、二叉树BTree，数组与链表的区别？

数据库索引就是一种**加快海量数据查询**的关键技术。索引是**关系数据库中对某一列或多个列的值进行预排序的数据结构。通过使用索引，可以让数据库系统不必扫描整个表，而是直接定位到符合条件的记录**，这样就大大加快了查询速度。

1）数据结构：

- B Tree 指的是 Balance Tree，也就是平衡树。平衡树是一颗查找树，并且所有叶子节点位于同一层。

- B+ Tree 是**基于 B Tree 和叶子节点顺序访问指针进行实现**，它具有 B Tree 的平衡性，并且**通过顺序访问指针来提高区间查询的性能。**
- 在 B+ Tree 中，一个节点中的 key 从左到右非递减排列 。



2）操作：

进行查找操作时，**首先在根节点进行二分查找，找到一个 key 所在的指针，然后递归地在指针所指向的节点进行查找。**直到查找到叶子节点，然后在叶子节点上进行二分查找，找出 key 所对应的 data。

插入删除操作会破坏平衡树的平衡性，因此在插入删除操作之后，需要对树进行一个分裂、合并、旋转等操作来维护平衡性。

3）与红黑树的比较：

红黑树等平衡树也可以用来实现索引，但是文件系统及数据库系统普遍采用 B+ Tree 作为索引结构，主要有以下两个原因：

> - 更少的查找次数
>
> 平衡树查找操作的时间复杂度和树高 h 相关，O(h)=O(logdN)，其中 d 为每个节点的出度。
>
> 红黑树的出度为 2，而 B+ Tree 的出度一般都非常大，所以红黑树的树高 h 很明显比 B+ Tree 大非常多，查找的次数也就更多。
> - 利用磁盘预读特性
>
> 为了减少磁盘 I/O 操作，磁盘往往不是严格按需读取，而是每次都会预读。预读过程中，磁盘进行顺序读取，顺序读取不需要进行磁盘寻道，并且只需要很短的磁盘旋转时间，速度会非常快。
>
> 操作系统一般将内存和磁盘分割成固定大小的块，每一块称为一页，内存与磁盘以页为单位交换数据。数据库系统将索引的一个节点的大小设置为页的大小，使得一次 I/O 就能完全载入一个节点。并且可以利用预读特性，相邻的节点也能够被预先载入。

数组：

- 在内存中，数组是一块**连续的区域**；
- 数组需要**预留空间，在使用前要先申请占内存的大小，可能会浪费内存空间**；
- **插入数据和删除数据效率低**（插入数据时，这个位置后面的数据在内存中都要向后移。删除数据时，这个数据后面的数据都要往前移动 ）；
- **随机读取效率很高。**因为数组是连续的，知道每一个数据的内存地址，可以直接找到给地址的数据；
- **不利于扩展，数组定义的空间不够时要重新定义数组。**

链表：

- 在**内存中可以存在任何地方，不要求连续**；
- **每一个数据都保存了下一个数据的内存地址**，通过这个地址找到下一个数据；
- **增加数据和删除数据很容易**；
- **查找数据时效率低，因为不具有随机访问性**，所以访问某个位置的数据都要从第一个数据开始访问，然后根据第一个数据保存的下一个数据的地址找到第二个数据，以此类推；
- **不指定大小，扩展方便。**链表大小不用定义，数据随意增删。

6、手写单例模式？

> 1、单例模式(Singleton Pattern)：单例模式**确保某一个类只有一个实例，而且自行实例化并向整个系统提供这个实例**，这个类称为单例类，它**提供全局访问的方法。**
>
> 2、单例模式的要点有三个：
>
> - 某个类只能有一个实例；
> - 它必须自行创建这个实例；
> - 它必须自行向整个系统提供这个实例。
>
> 3、优缺点?
>
> 优点：**节约资源，节省时间。**
>
> - 由于频繁使用的对象，可以省略创建对象所花费的时间，这对于那些重量级的对象而言，是很重要的。
> - 因为不需要频繁创建对象，我们的GC压力也减轻了。
>
> 缺点：简单的单例模式设计开发都比较简单，但是**复杂的单例模式需要考虑线程安全等并发问题**，引入了部分复杂度。**职责过重**（既充当工厂角色，提供工厂方法，同时又充当了产品角色。包含一些业务方法）。
>
> 4、三要素：
>
> - 一个静态类变量；
> - 一个私有构造方法；
> - 一个全局静态的类方法。
>
> 5、类型：懒汉式（线程不安全）、饿汉式（天生线程安全）。
>
> 饿汉式：初始化类时，直接就创建唯一实例；
>
> （1）
>
> ```
> public class Singleton {
>     private static Singleton instance = new Singleton();////私有静态变量
>     private Singleton() {}//私有构造函数
>     public static Singleton getInstance() {//公有静态函数
>         return instance;
>     }
> }
> ```
>
> （2）枚举方式：（线程安全，**可防止反射构建**），在枚举类对象被反序列化的时候，保证反序列的返回结果是同一对象。

```
public enum Singleton {
    INSTANCE;
}
```

懒汉式：

（1）双重校验锁：

```
//第一次判断 instance是否为空是为了确保返回的实例不为空
//第二次判断 instance是否为空是为了防止创建多余的实例
public class Singleton {
    private volatile static Singleton instance = null;
    private Singleton() {}
    public static Singleton getInstance() {
        if(instance == null) {
            synchronized(Singleton.class){
                if(instance == null) {//线程B
                    instance = new Singleton();//并非是一个原子操作     线程A
                    /**
                    * 会被编译器编译成如下JVM指令：
                    * memory = allocate(); - 1、分配对象的内存空间
                    * ctorInstance(memory); - 2、初始化对象
                    * instance = memory; - 3、设置instance指向刚分配的内存地址
                    */
                }
            }
        }
        return instance;
    }
}
```

synchronized同步块里面能够保证只创建一个对象。但是**通过在synchronized的外面增加一层判断，就可以在对象一经创建以后，不再进入synchronized同步块。这种方案不仅减小了锁的粒度，保证了线程安全，性能方面也得到了大幅提升。** 进入Synchronized 临界区以后，还要再做一次判空。因为当两个线程同时访问的时候，线程A构建完对象，线程B也已经通过了最初的判空验证，不做第二次判空的话，线程B还是会再次构建instance对象。

Volatile：

- 可见性；
- 防止指令重排序：**防止new Singleton时指令重排序导致其他线程获取到未初始化完的对象。**指令顺序并非一成不变，有可能会经过JVM和CPU的优化，指令重排成下面的顺序：1，3，2。在3执行完毕，2未执行之前，被线程2抢占了，这时instance已经是非null了（但却没有初始化），所以线程2会直接返回instance，然后使用，报错。

（2）静态内部类：线程安全，懒加载

```
//INSTANCE对象初始化的时机并不是在单例类Singleton被加载的时候，而是在调用getInstance方法，使得静态内部类SingletonHolder被加载的时候。因此这种实现方式是利用classloader的加载机制来实现懒加载，并保证构建单例的线程安全。
public class Singleton {
    private static class SingletonHolder {
        private static final Singleton INSTANCE = new Singleton();
    }
    private Singleton() {}
    public static Singleton getInstance() {
        return SingletonHolder.INSTANCE;
    }
}
```

7、线程安全，synchronized关键字？

线程安全是多**线程**编程时的计算机程序代码中的一个概念。 在拥有共享数据的多条**线程**并行执行的程序中，**线程安全**的代码会通过同步机制保证各个**线程**都可以正常且正确的执行，不会出现数据污染等意外情况。

为了解决并发编程中存在的**原子性、可见性和有序性**问题，提供了一系列和并发处理相关的关键字，比如`synchronized`、`volatile`、`final`、`concurren包`等。

synchronized：

- 被`synchronized`修饰的代码块及方法，**在同一时间，只能被单个线程访问。**
- `synchronized`，是Java中用于**解决并发情况下数据同步访问**的一个很重要的关键字。当我们想要**保证一个共享资源在同一时间只会被一个线程访问到时，我们可以在代码中使用`synchronized`关键字对类或者对象加锁。**

> synchronized与原子性：原子性是指**一个操作是不可中断的，要全部执行完成，要不就都不执行。** 线程是CPU调度的基本单位。CPU有时间片的概念，会根据不同的调度算法进行线程调度。当一个线程获得时间片之后开始执行，在时间片耗尽之后，就会失去CPU使用权。所以在多线程场景下，由于时间片在线程间轮换，就会发生原子性问题。
>
> 被`synchronized`修饰的代码在同一时间只能被一个线程访问，在锁未释放之前，无法被其他线程访问到 。
>
> synchronized与可见性：可见性是指**当多个线程访问同一个变量时，一个线程修改了这个变量的值，其他线程能够立即看得到修改的值。**Java内存模型规定了所有的变量都存储在主内存中，每条线程还有自己的工作内存，线程的工作内存中保存了该线程中是用到的变量的主内存副本拷贝，线程对变量的所有操作都必须在工作内存中进行，而不能直接读写主内存。不同的线程之间也无法直接访问对方工作内存中的变量，线程间变量的传递均需要自己的工作内存和主存之间进行数据同步进行。所以，就可能出现线程1改了某个变量的值，但是线程2不可见的情况。
>
> 被`synchronized`修饰的代码，在开始执行时会加锁，执行完成后会进行解锁。而为了保证可见性，有一条规则是这样的：**对一个变量解锁之前，必须先把此变量同步回主存中。**这样解锁后，后续线程就可以访问到被修改后的值。
>
> synchronized与有序性：有序性即**程序执行的顺序按照代码的先后顺序执行。** 除了引入了时间片以外，由于处理器优化和指令重排等，CPU还可能对输入代码进行乱序执行，比如load->add->save 有可能被优化成load->save->add 。这就是可能存在有序性问题。

> `synchronized`是无法禁止指令重排和处理器优化的。也就是说，`synchronized`无法避免上述提到的问题。Java程序中天然的有序性可以总结为一句话：如果在本线程内观察，所有操作都是天然有序的。如果在一个线程中观察另一个线程，所有操作都是无序的。 由于`synchronized`修饰的代码，同一时间只能被同一线程访问。那么也就是单线程执行的。所以，可以保证其有序性。

volatile：`volatile`通常被比喻成"轻量级的`synchronized`"，也是Java并发编程中比较重要的一个关键字。和`synchronized`不同，`volatile`是一个变量修饰符，**只能用来修饰变量。无法修饰方法及代码块等。** `volatile`的用法比较简单，只需要**在声明一个可能被多线程同时访问的变量时，使用`volatile`修饰**就可以了。

> volatile与可见性：当多个线程访问同一个变量时，一个线程修改了这个变量的值，其他线程能够立即看得到修改的值。
>
> Java中的`volatile`关键字提供了一个功能，那就是**被其修饰的变量在被修改后可以立即同步到主内存，被其修饰的变量在每次是用之前都从主内存刷新**。因此，可以使用`volatile`来保证多线程操作时变量的可见性。
>
> volatile与有序性：有序性即程序执行的顺序按照代码的先后顺序执行。
>
> 而`volatile`除了可以保证数据的可见性之外，还有一个强大的功能，那就是他可以**禁止指令重排优化**等。通过禁止指令重排优化，就可以保证代码程序会严格按照代码的先后顺序执行。 volatile是通过**内存屏障**来来禁止指令重排的。
>
> **内存屏障（Memory Barrier）**是一类同步屏障指令，是CPU或编译器在对内存随机访问的操作中的一个同步点，使得此点之前的所有读写操作都执行后才可以开始执行此点之后的操作。
>
> volatile与原子性：原子性是指一个操作是不可中断的，要全部执行完成，要不就都不执行。
>
> **volatile是不能保证原子性的** 。

比较：

- synchronized是一种锁机制，存在阻塞问题和性能问题，而volatile并不是锁，所以不存在阻塞和性能问题。
- volatile借助了内存屏障来帮助其解决可见性和有序性问题，而内存屏障的使用还为其带来了一个禁止指令重排的附件功能，所以在有些场景中是可以避免发生指令重排的问题的。

8、接口和抽象类的区别?

- 抽象类是用来**捕捉子类的通用特性**的。它不能被实例化，只能被用作子类的超类。抽象类是被用来创建继承层级里子类的模板。
- **接口是抽象方法的集合。**如果一个类实现了某个接口，那么它就继承了这个接口的抽象方法。

区别：

> 从语法层面来说：
>
> - 抽象类可以提供成员方法的实现细节，而接口中只能存在抽象方法
> - 抽象类中成员变量可以是多种类型，接口中成员变量必须用public，static，final修饰
> - 一个类只能继承一个抽象类，但可以实现多个接口
> - 抽象类中允许含有静态代码块和静态方法，接口不能
>
> 从设计层面而言：
>
> - 抽象类是对整一个类的属性，行为等方面进行抽象，而接口则是对行为抽象。就好比飞机和鸟，抽象类抽象出的是飞行物类。而接口则是抽闲出飞行方法。
> - 抽象类是一个模板式的设计，当在开发过程中出现需求更改的情况，只需要更改抽象类而不需要更改它的子类。接口是一种辐射性设计，当接口的内容发生改变时，需要同时对实现它的子类进行相应的修

> 改。
> - 抽象类可以类比为模板，而接口可以类比为协议。

9、内存泄漏与内存溢出?

- 内存泄漏（memory leak）：是指**程序在申请内存后，无法释放已申请的内存空间**，一次内存泄漏似乎不会有大的影响，但内存泄漏堆积后的后果就是内存溢出。

  > 常见的内存泄漏：
  >
  > - 单例造成内存泄漏：由于单例的静态特性使得其生命周期和应用的生命周期一样长，如果一个对象已经不再需要使用了，而单例对象还持有该对象的引用，就会使得该对象不能被正常回收，从而导致了内存泄漏。
  > - 资源未关闭造成的内存泄漏：对于使用了File，Stream等资源，应该在Activity销毁时及时关闭或者注销，否则这些资源将不会被回收，从而造成内存泄漏。

- 内存溢出 （out of memory）：指**程序申请内存时，没有足够的内存供申请者使用**，或者说，给了你一块存储int类型数据的存储空间，但是你却存储long类型的数据，那么结果就是内存不够用，此时就会报错OOM，即所谓的内存溢出。

10、方法重载、方法重写?

- 重写指的是**在Java的子类与父类中有两个名称、参数列表都相同的方法的情况。**由于他们具有相同的方法签名，所以**子类中的新方法将覆盖父类中原有的方法。**
- 简单说，就是**方法有同样的名称，但是参数列表不相同的情形**，这样的同名不同参数的方法之间，互相称之为重载方法。 应该注意的是，返回值不同，其它都相同不算是重载。

区别：位置不一样 ，一个是说父子类，一个是当前类中；方法参数变化，一个不变化，一个是参数个数可以不一样。

> 1、重载是一个**编译期**概念、重写是一个**运行期间**概念。
>
> 2、重载遵循所谓"编译期绑定"，即在编译时根据参数变量的类型判断应该调用哪个方法。
>
> 3、重写遵循所谓"运行期绑定"，即在运行的时候，根据引用变量所指向的实际对象的类型来调用方法。
>
> 4、因为在编译期已经确定调用哪个方法，所以重载并不是多态。而重写是多态。重载只是一种语言特性，是一种语法规则，与多态无关，与面向对象也无关。（注：严格来说，**重载是编译时多态**，即静态多态。但是，Java中提到的多态，在不特别说明的情况下都指动态多态）

11、http协议和socket协议的区别？

> http协议：
>
> http 为短连接：客户端发送请求都需要服务器端回送响应.请求结束后，主动释放链接，因此为短连接。通常的做法是，不需要任何数据，也要保持每隔一段时间向服务器发送"保持连接"的请求。这样可以保证客户端在服务器端是"上线"状态。
>
> HTTP连接使用的是"请求-响应"方式，不仅在请求时建立连接，而且客户端向服务器端请求后，服务器才返回数据。
>
> socket连接：
>
> 要想明白 Socket，必须要理解 TCP 连接。
>
> TCP 三次握手：握手过程中并不传输数据，在握手后服务器与客户端才开始传输数据，理想状态下，TCP 连接一旦建立，在通讯双方中的任何一方主动断开连接之前 TCP 连接会一直保持下去。

> Socket 是对 TCP/IP 协议的封装，Socket 只是个接口不是协议，通过 Socket 我们才能使用 TCP/IP 协议，除了 TCP，也可以使用 UDP 协议来传递数据。
>
> 创建 Socket 连接的时候，可以指定传输层协议，可以是 TCP 或者 UDP，当用 TCP 连接，该Socket就是个TCP连接，反之。
>
> **Socket为长连接：**通常情况下Socket 连接就是 TCP 连接，因此 Socket 连接一旦建立,通讯双方开始互发数据内容，直到双方断开连接。在实际应用中，由于网络节点过多，在传输过程中，会被节点断开连接，因此要通过轮询高速网络，该节点处于活跃状态。

若双方是 Socket 连接，可以由服务器直接向客户端发送数据。

若双方是 HTTP 连接，则服务器需要等客户端发送请求后，才能将数据回传给客户端。

12、TCP和HTTP的区别？

TPC/IP协议是传输层协议，主要解决数据如何在网络中传输，而HTTP是应用层协议，主要解决如何包装数据。关于TCP/IP和HTTP协议的关系，网络有一段比较容易理解的介绍："我们在传输数据时，可以只使用（传输层）TCP/IP协议，但是那样的话，如果没有应用层，便无法识别数据内容，如果想要使传输的数据有意义，则必须使用到应用层协议，应用层协议有很多，比如HTTP、FTP、TELNET等，也可以自己定义应用层协议。WEB使用HTTP协议作应用层协议，以封装HTTP 文本信息，然后使用TCP/IP做传输层协议将它发到网络上。"

13、http和https区别?

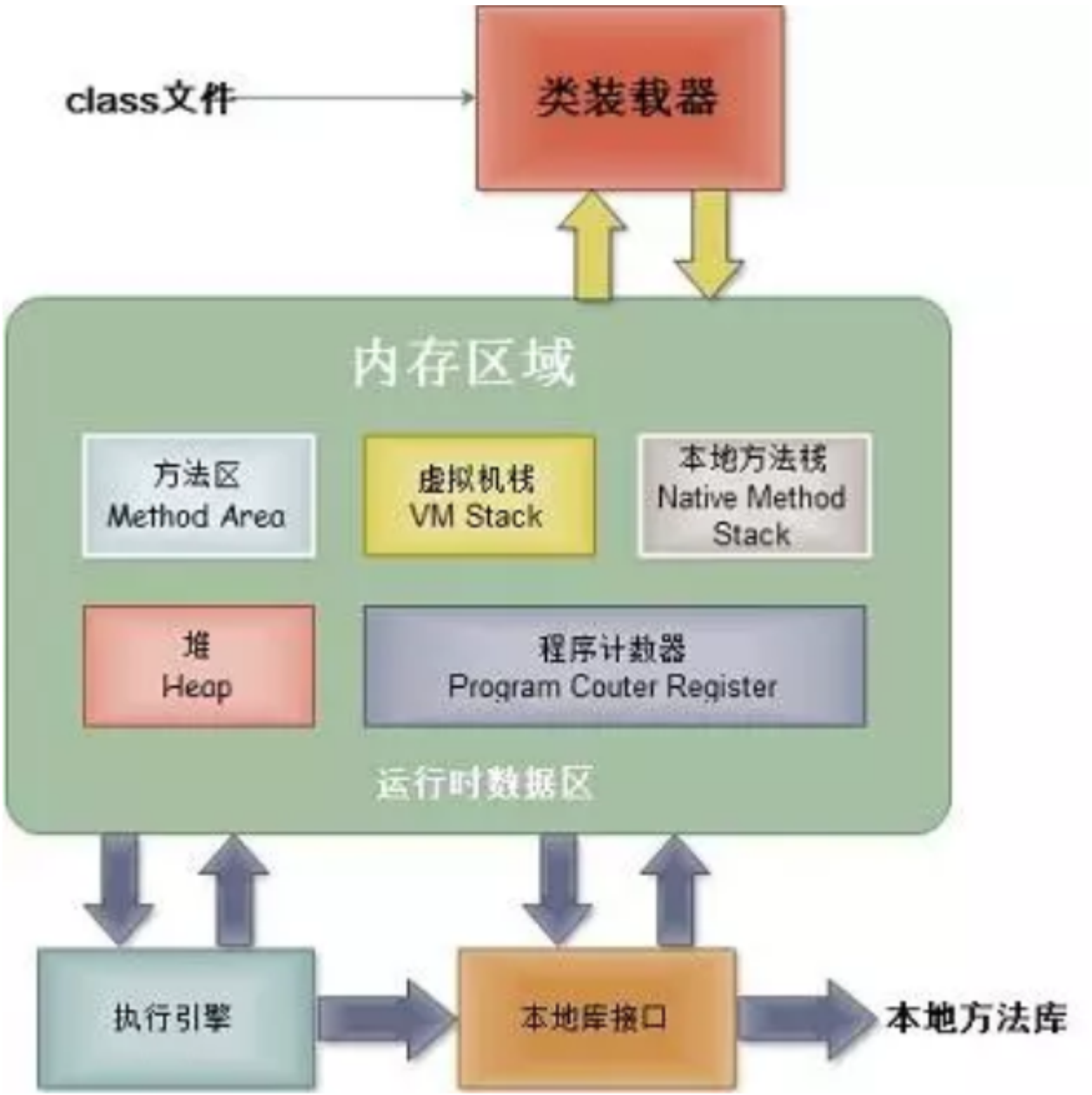
HTTP协议传输的数据都是未加密的，也就是明文的，因此使用HTTP协议传输隐私信息非常不安全，为了保证这些隐私数据能加密传输，于是网景公司设计了**SSL（Secure Sockets Layer）协议用于对HTTP协议传输的数据进行加密，从而就诞生了HTTPS。**简单来说，**HTTPS协议是由SSL+HTTP协议构建的可进行加密传输、身份认证的网络协议，要比http协议安全。**

HTTPS和HTTP的区别主要如下：

- **https协议需要到ca申请证书，一般免费证书较少，因而需要一定费用。**
- **http是超文本传输协议，信息是明文传输，https则是具有安全性的ssl加密传输协议。**
- **http和https使用的是完全不同的连接方式，用的端口也不一样，前者是80，后者是443。**
- **http的连接很简单，是无状态的；HTTPS协议是由SSL+HTTP协议构建的可进行加密传输、身份认证的网络协议，比http协议安全。**

14、Java的内存管理机制？垃圾回收?

Java内存区域划分：



- 程序计数器：可以看做是**当前线程所执行的字节码的行号指示器。**字节码解释器工作时就是通过改变这个计数器的值来选取下一条需要执行的字节码指令。（线程私有的内存区域）
- Java虚拟机栈：描述**Java方法执行的内存模型，每个方法执行的同时会创建一个栈帧**，栈帧用于存储局部变量表、操作数栈、动态链接、方法出口等信息。每个方法从调用直至执行完成的过程，就对应着一个栈帧在虚拟机栈中入栈到出栈的过程。（线程私有的）
- 本地方法栈：本地方法栈与虚拟机栈的区别：虚拟机栈为虚拟机执行Java方法服务（也就是字节码），而**本地方法栈为虚拟机使用到的Native方法服务。**
- Java堆：Java**堆是被所有的线程共享的一块内存区域**，在虚拟机启动时创建。Java堆的唯一目的就是**存放对象实例**，几乎所有的对象实例都在这里分配内存。
- 方法区：**被所有的线程共享的一块内存区域。**它**用于存储已被虚拟机加载的类信息、常量、静态变量、即时编译器编译后的代码等数据。**

垃圾回收(Garbage Collection)是**Java虚拟机(JVM)垃圾回收器提供的一种用于在空闲时间不定时回收无任何对象引用的对象占据的内存空间的一种机制。**

引用：如果Reference类型的数据中存储的数值代表的是另外一块内存的起始地址，就称这块内存代表着一个引用。

> （1）强引用（Strong Reference）：在 Java中最常见的就是强引用，把一个对象赋给一个引用变量，这个引用变量就是一个强引用。当一个对象被强引用变量引用时，它处于可达状态，它是不可能被垃圾回收机制回收的，即使该对象以后永远都不会被用到 JVM 也不会回收。因此强引用是造成 Java 内存泄漏的主要原因之一。只要强引用还存在，垃圾收集器就永远不会回收掉被引用的对象。
>
> （2）软引用（Soft Reference）：软引用需要用 SoftReference 类来实现，对于只有软引用的对象来说，当系统内存足够时它不会被回收，当系统内存空间不足时它会被回收。软引用通常用在对内存敏感的程序中。
>
> （3）弱引用（Weak Reference）：弱引用需要用 WeakReference 类来实现，它比软引用的生存期更短，对于只有弱引用的对象 来说，只要垃圾回收机制一运行，不管 JVM 的内存空间是否足够，总会回收该对象占用的内存。
>
> （4）虚引用（Phantom Reference）：虚引用需要 PhantomReference 类来实现，它不能单独使用，必须和引用队列联合使用。 虚引用的主要作用是跟踪对象被垃圾回收的状态。

垃圾：无任何对象引用的对象。

判断对象是否是垃圾的算法：

- 引用计数算法（Reference Counting Collector）：**给对象添加一个引用计数器，每当有一个地方引用他时，计数器就加一，引用失败计数器减一**，计数器为零的对象就不可能再被使用。
- 根搜索算法（Tracing Collector）：通过一系列的称为"GC Roots"的对象作为起始点，从这些节点开始向下搜索，搜索走过的路径称为引用链。**当一个对象到GC Roots没有任何引用链相连的时候，则证明此对象是不可用的。**否则即是可达的。

GC Root可以是：虚拟机栈（栈帧中的本地变量表）中引用的对象；本地方法栈中JNI（即一般说的native方法）引用的对象；方法区中的静态变量和常量引用的对象。

回收：清理“垃圾”占用的内存空间而非对象本身。

标记—清除算法：分为“标记”和“清除”两个阶段：首先**标记出所需回收的对象，在标记完成后统一回收掉所有被标记的对象**，算法最大的问题是**内存碎片化严重 。**

标记—整理算法：标记的过程与标记—清除算法中的标记过程一样，但**对标记后出的垃圾对象的处理情况有所不同，它不是直接对可回收对象进行清理，而是让所有的对象都向一端移动，然后直接清理掉端边界以外的内存。**

复制算法（Copying Collector）：**将内存按容量分为大小相等的两块，每次只使用其中的一块（对象面），当这一块的内存用完了，就将还存活着的对象复制到另外一块内存上面（空闲面），然后再把已使用过的内存空间一次清理掉。**可用内存被压缩到了原本的一半。且存活对象增多的话， Copying 算法的效率会大大降低。

发生地点：**一般发生在堆内存中，因为大部分的对象都储存在堆内存中。**

分代收集算法：

Java的堆内存基于Generation算法（Generational Collector）划分为**新生代、年老代和持久代。**新生代又被进一步划分为Eden和Survivor区，最后Survivor由FromSpace（Survivor0）和ToSpace（Survivor1）组成。所有通过new创建的对象的内存都在堆中分配。分代收集基于这样一个事实：**不同的对象的生命周期是不一样的。因此，可以将不同生命周期的对象分代，不同的代采取不同的回收算法进行垃圾回收（GC）**，以便提高回收效率。

- 新生代与复制算法：目前大部分 JVM 的 GC **对于新生代都采取 Copying 算法**，因为**新生代中每次垃圾回收都要回收大部分对象，即要复制的操作比较少**，但通常并不是按照 1： 1 来划分新生代。一般将新生代 划分为一块较大的 Eden 空间和两个较小的 Survivor 空间(From Space, To Space)，每次使用 Eden 空间和其中的一块 Survivor 空间，当进行回收时，将该两块空间中还存活的对象复制到另 一块 Survivor 空间中。

- 老年代与标记整理算法：**老年代因为每次只回收少量对象，因而采用标记-整理算法。**

15、tcp滑动窗口和拥塞避免？

滑动窗口机制：

滑动窗口协议的基本原理就**是在任意时刻，发送方都维持了一个连续的允许发送的帧的序号，称为发送窗口；同时，接收方也维持了一个连续的允许接收的帧的序号，称为接收窗口。**发送窗口和接收窗口的序号的上下界不一定要一样，甚至大小也可以不同。不同的滑动窗口协议窗口大小一般不同。发送方窗口内的序列号代表了那些已经被发送，但是还没有被确认的帧，或者是那些可以被发送的帧。

举例：

发送和接受方都会维护一个数据帧的序列，这个序列被称作窗口。发送方的窗口大小由接受方确定，目的在于控制发送速度，以免接受方的缓存不够大，而导致溢出，同时控制流量也可以避免网络拥塞。图中的4,5,6号数据帧已经被发送出去，但是未收到关联的ACK，7,8,9帧则是等待发送。可以看出发送端的窗口大小为6，这是由接受端告知的（事实上必须考虑拥塞窗口cwnd，这里暂且考虑cwnd>rwnd）。此时如果发送端收到4号ACK，则窗口的左边缘向右收缩，窗口的右边缘则向右扩展，此时窗口就向前“滑动了”，即数据帧10也可以被发送。

拥塞避免：

TCP的拥塞控制由4个核心算法组成：

“慢启动”（Slow Start）

“拥塞避免”（Congestion avoidance）

“快速重传 ”（Fast Retransmit）

“快速恢复”（Fast Recovery）

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。这一点和流量控制很像，但是出发点不同。流量控制是为了让接收方能来得及接收，而**拥塞控制是为了降低整个网络的拥塞程度。**

> 慢开始与拥塞避免：
>
> 发送的最初执行慢开始，令 cwnd = 1，发送方只能发送 1 个报文段；当收到确认后，将 cwnd 加倍，因此之后发送方能够发送的报文段数量为：2、4、8 ...
>
> 注意到慢开始每个轮次都将 cwnd 加倍，这样会让 cwnd 增长速度非常快，从而使得发送方发送的速度增长速度过快，网络拥塞的可能性也就更高。设置一个慢开始门限 ssthresh，当 cwnd >= ssthresh 时，进入拥塞避免，每个轮次只将 cwnd 加 1。
>
> 如果出现了超时，则令 ssthresh = cwnd / 2，然后重新执行慢开始。
>
> 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。例如已经接收到 M1 和 M2，此时收到 M4，应当发送对 M2 的确认。
>
> 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。例如收到三个 M2，则 M3 丢失，立即重传 M3。
>
> 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 ssthresh = cwnd / 2 ，cwnd = ssthresh，注意到此时直接进入拥塞避免。
>
> 慢开始和快恢复的快慢指的是 cwnd 的设定值，而不是 cwnd 的增长速率。慢开始 cwnd 设定为 1，而快恢复 cwnd 设定为 ssthresh。

16、arp协议和ip协议？

ARP 实现由 IP 地址得到 MAC 地址。

每个主机都有一个 ARP 高速缓存，里面有**本局域网上的各主机和路由器的 IP 地址到 MAC 地址的映射表。**

如果主机 A 知道主机 B 的 IP 地址，但是 ARP 高速缓存中没有该 IP 地址到 MAC 地址的映射，此时**主机 A 通过广播的方式发送 ARP 请求分组**，主机 B 收到该请求后会发送 ARP 响应分组给主机 A 告知其 MAC 地址，随后主机 A 向其高速缓存中写入主机 B 的 IP 地址到 MAC 地址的映射。

ARP 高效运行的关键是**每个主机上都有一个 ARP 的高速缓存。**

IP（Internet Protocol）协议是TCP/IP协议族的核心组成部分，是目前应用最广的网络互联协议。IP层对应于ISO/OSI七层参考模型中的网络层。通过IP数据包和IP地址屏蔽掉了不同的物理网络（如以太网、令牌环网等）的帧格式、地址格式等各种底层物理网络细节，使得各种物理网络的差异性对上层协议不复存在，从而使网络互联成为可能。IP协议的主要功能有：无连接数据报传送、数据报路由选择和差错控制。

17、访问页面的过程？

- 首先对输入的url**进行DNS解析**（找出对端服务器ip地址）。优先访问浏览器缓存，如果未命中则访问OS缓存，最后再访问DNS服务器，然后DNS服务器会递归式的查找域名记录。
- 通过ip地址，**建立TCP请求**（三次握手协议）。查询到域名对应的ip地址之后，浏览器设置Remote Address，以及端口号port等信息，确保传输层的正常通信（即TCP正常连接）。
- **发送HTTP请求。**（传输层连接已经建立完成，准备发送HTTP请求）。
- **服务器处理请求并返回HTTP报文。**（Http服务器读取HTTP Get报文，生成一个HTTPP响应报文，将web页面内容放入报文主体中，发回给主机）。
- **解析渲染服务器的响应数据。**（浏览器收到HTTP响应报文后，抽取出web页面内容，之后进行渲染，显示web页面）。
- **断开TCP连接**：TCP四次挥手。

18、排序?

排序类别	排序方法	时间复杂度			空间复杂度	稳定性	复杂性
		平均情况	最坏情况	最好情况			
交换排序	冒泡排序	$O(n^2)$	$O(n^2)$	$O(n)$	$O(1)$	稳定	简单
	快速排序	$O(nlog_2n)$	$O(n^2)$	$O(nlog_2n)$	$O(nlog_2n)$	不稳定	较复杂
插入排序	直接插入排序	$O(n^2)$	$O(n^2)$	$O(n)$	$O(1)$	稳定	简单
	希尔排序	$O(nlog_2n)$	$O(n^2)$		$O(1)$	不稳定	较复杂
选择排序	简单选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	不稳定	简单
	堆排序	$O(nlog_2n)$	$O(nlog_2n)$	$O(nlog_2n)$	$O(1)$	不稳定	较复杂
归并排序	归并排序	$O(nlog_2n)$	$O(nlog_2n)$	$O(nlog_2n)$	$O(n)$	稳定	较复杂
基数排序	基数排序	$O(d * n)$	$O(d * n)$	$O(d * n)$	$O(n)$	稳定	较复杂

```
//归并排序：将数组分成两部分，分别进行排序，然后归并起来。
public static void sort(int[] num, int low, int high) {
    int mid = low + (high - low) / 2;
    if(low < high) {
        sort(num, low, mid);
        sort(num, mid + 1, high);
        merge(num, low, mid, high);
    }
}
public static void merge(int[] num, int low, int mid, int high) {
    int[] temp = new int[high - low + 1];
    int i = low;//左指针
    int j = mid + 1;//右指针
    int k = 0;
    while(i <= mid && j <= high) {//将较小的数先移入新数组中
        if(num[i] <= num[j]) {
            temp[k++] = num[i++];
        } else {
            temp[k++] = num[j++];
        }
    }
    //把左边剩余的数移入数组
    while(i <= mid) {
        temp[k++] = num[i++];
    }
    //把右边剩余的数移入数组
    while(j <= high) {
        temp[k++] = num[j++];
    }
    //将新数组中的值拷入原数组
    k = 0;
    while(low <= high) {
        num[low++] = temp[k++];
    }
}
//快速排序：通过一个切分元素将数组分为两个子数组。左子数组小于等于切分元素，右子数组大于等于切分元素，将这两个子数组排序也就将整个数组排序了。
public class quick {
    public static void sort(int[] num, int left, int right) {
        if(left > right) {
            return;
        } else {
            int base  = divide(num, left, right);
            sort(num, left, base - 1);
            sort(num, base + 1, right);
        }
    }
    public static int divide(int[] num, int left, int right) {
        int base = num[left];
        while(left < right) {
            //从数组右端开始，向左遍历，直到找到小于base的数
            while(left < right && num[right] > base) {
                right--;
```

```
            }
            //找到了比base小的元素，将这个元素放到最左边的位置
            num[left] = num[right];
            //从数组左端开始，向右遍历，直到找到大于base的数
            while(left < right && num[left] < base) {
                left++;
            }
            //找到了比base大的元素，将这个元素放到最右边的位置
            num[right] = num[left];
        }
        //最后将base放到left位置，此时，left左边的值都比它小，右边的值都比它大
        num[left] = base;
        return left;
    }
}
//堆排序
public class HeadSort {
    public static void buildHeap(int[] arr, int parent, int length) {
        for(int child = 2 * parent + 1; child < length; child = 2 * child + 1) {
            if(child + 1 < length && arr[child] < arr[child + 1]) {// 让child先指向子节点中最大的
节点
                child++;
            }
            if(arr[parent] < arr[child]) {// 如果发现子节点更大，则进行值的交换
                swap(arr, parent, child);
                // 下面就是非常关键的一步了
                // 如果子节点更换了，那么，以子节点为根的子树会不会受到影响呢?
                // 所以，循环对子节点所在的树继续进行判断
                parent = child;
            } else {
                break;
            }
        }
    }

    public static void swap(int[] arr, int i, int j) {
        int temp = arr[i];
        arr[i] = arr[j];
        arr[j] = temp;
    }

    public static void sort(int[] arr){
        for(int parent = arr.length / 2 - 1; parent >= 0; parent--) {//构建最大堆
            buildHeap(arr, parent, arr.length);
        }
        for(int i = arr.length - 1; i > 0; i--) {//把最大的元素扔在最后面
            swap(arr, 0, i);
            buildHeap(arr, 0, i);//将arr中前i - 1个记录重新调整为最大堆
        }
    }
}
```

19、单链表反转?

```
public class ListNode {
    int val;
    ListNode next = null;

    ListNode(int val) {
        this.val = val;
    }
}
//递归
public ListNode ReverseList(ListNode head) {
    if(head == null || head.next == null) {
        return head;
    }
    ListNode newHead = ReverseList(head.next);
    head.next.next = head;
    head.next = null;
    return newHead;
}
//非递归
public ListNode ReverseList(ListNode head) {
    ListNode newHead = null;//已反转的链表
    while(head != null) {
        ListNode next = head.next;
        head.next = newHead;
        newHead = head;
        head = next;
    }
    return newHead;
}
```

20、TCP/UDP？

- **TCP提供面向对象的连接，通信前要建立三次握手机制的连接，UDP提供无连接的传输，传输前不用建立连接**
- **TCP提供可靠的，有序的，不丢失的传输，UDP提供不可靠的传输**
- **TCP提供面向字节流的传输，它能将信息分割成组，并在接收端将其充足，UDP提供面向数据报的传输，没有分组开销**
- **TCP提供拥塞控制，流量控制机制，UDP没有**
- **每一条TCP连接只能是点到点的；UDP支持一对一，一对多，多对一和多对多的交互通信**

21、虚拟内存？

虚拟内存的目的是**为了让物理内存扩充成更大的逻辑内存，从而让程序获得更多的可用内存。**

为了更好的管理内存，**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，由硬件执行必要的映射，将缺失的部分装入物理内存并重新执行失败的指令。**

虚拟内存允许程序不用将地址空间中的每一页都映射到物理内存，也就是说**一个程序不需要全部调入内存就可以运行，这使得有限的内存运行大程序成为可能。**例如有一台计算机可以产生 16 位地址，那么一个程序的地址空间范围是 0~64K。该计算机只有 32KB 的物理内存，虚拟内存技术允许该计算机运行一个 64K 大小的程序。

22、进程、线程?

- 拥有资源： **进程是资源分配的基本单位，但是线程不拥有资源**，线程可以访问隶属进程的资源。
- 调度： **线程是独立调度的基本单位，在同一进程中，线程的切换不会引起进程切换，从一个进程中的线程切换到另一个进程中的线程时，会引起进程切换。**
- 系统开销： **由于创建或撤销进程时，系统都要为之分配或回收资源，如内存空间、I/O 设备等，所付出的开销远大于创建或撤销线程时的开销。**类似地，在进行进程切换时，涉及当前执行进程 CPU 环境的保存及新调度进程 CPU 环境的设置，而线程切换时只需保存和设置少量寄存器内容，开销很小。
- 通信方面： **线程间可以通过直接读写同一进程中的数据进行通信，但是进程通信需要借助 IPC。** 进程间通信（**IPC**，Inter-Process Communication）。

华为 9/8

专业一面

自我介绍

香农公式

测试打电话功能

了解啥测试方法

手撕双向链表的插入

你机试100分是个啥水平，题还记得不

linux会不

为啥投这个部门

还有啥问题

专业二面

自我介绍

手撕判断二叉树是否相等

写测试你上面代码的测试用例

机试第二题记得不

linux会不

安卓会不

操作系统还记得啥

还有啥问题

业务主管面

自我介绍

为啥测开

了解啥测试知识

自动化测试工具都有啥

如何快速看懂开发的代码及接口

你觉得你上一个问题回答的咋样?

家在哪

怎么看待加班

为啥来华为

你在学校项目中觉得有成就感的事

愿意开发吗

怎么看外界对华为的各种说法

还有啥问题