1.json、session、cookie

json是javascript里面轻量级的数据交换格式，我是在写flask网页里面运用了百度echarts然后用到的，我觉得它和python里面的字典dictionary很相似，所以我是当做字典来理解的。

session也是在写网页的时候碰到的，后来去了解了一下，是在浏览器浏览网站的时候，服务器生成一个session，然后将唯一的sessionID发送给客户端，整个session一直持续到会话结束，也就是浏览器关闭为止。session是一个容器，可以存储会话中的对象。

cookie则和session不一样，cookie是存在客户端的本地的，在浏览器访问网页的时候，服务器会根据输入的内容和选择的什么的生成cookie存储在本地。然后客户端第二次访问的时候，服务器就会根据本地的cookie生成特定的内容，广告也可以利用这个来对客户进行进一步细分。

application是存储在服务器，直到应用程序关闭。

get： 获取 / 查询资源，安全

post： 更新资源

2.c/c++的内存管理

在c里面，内存分为五个区：堆、栈、自由存储区、全局/静态存储区和常量存储区。

栈：函数的局部变量在此创建，函数执行结束就会自动释放。

堆：new分配的内存，一般由程序员手动delete，如果没有，则在程序结束后，操作系统自动回收。

自由存储区：malloc分配的内存，是由free释放的。

全局/静态存储区：全局变量和静态变量在此分配。

常量存储区：存放的是常量，不允许修改。

3.计算机网络

OSI模型分为七层：物理层、数据链路层、网络层、传输层、会话层、表示层以及应用层。

物理层：提供为建立、维护和拆除物理链路所需要的机械的、电气的特性。

数据链路层：网络层实体间提供数据发送和接收的功能和过程，提供数据链路的流控。

网络层：将具体的物理传送对高层透明，具有路由选择、拥塞控制等功能。

传输层：负责建立、维护、拆除连接。选择网络层提供的最合适的服务，在系统之间提供可靠的端到端的传输和流量控制。

会话层：提供进程之间建立、维护和结束会话的功能。

表示层：数据转换、格式化以及文本压缩。

应用层：为用户服务，事务处理程序、文件传送协议和网络管理。

三次握手、四次挥手：

握手：1.客户端发送SYN建立联机请求。2.服务器收到后确认联机信息，发送ACK、SYN。3.客户端检查ACK，若正确则发送ACK给服务器，服务器确认后就建立了连接。

挥手：由于TCP连接是全双工的，每个方向都要单独关闭。

1.客户端发送一个FIN，来关闭客户端到服务器的数据传送

2.服务器收到FIN之后，发回一个ACK，来确认客户端到服务器断开。

3.服务器没有要发送的数据时，发送一个FIN给客户端，来关闭服务器与客户端的连接。

4.客户端发回ACK确认。

4.mongodb：非关系型数据库。非关系型数据库是阉割版的关系型数据库。两张表之间的依赖性没有关系型数据那么强。其简单易部署，数据存在缓存而不是硬盘；存储格式是键值对、文档、图片，可以存储对象和集合各种形式。

5.排序

1.直接插入排序：稳定的，O(N2)
一个数组，第一个起，看做有序序列，并将其设置为哨兵A，将后面的依次与其作比较，如果后面的B>A，就插入有序序列放在A后面，若B<A，则与有序序列A前面的有序子列遍历，找到合适位置插入，如果等于在放在A后面。
2.希尔排序，不稳定，O(N2)
对于数组，选取一个增量，比如选b，然后从数组第一个数开始，将下标为0，b,2b,3b....等分为一组，在将1,1+b,1+2b,1+3b...等分为第二组，以此类推直到分完，每一组直接排序，然后，再重复之前一部，其中b取其一半，直到其等于1为止。
3.简单选择排序 O(N2) 稳定
选出数组中最小的，与第一位交换，然后剩下的找最小的，与第二位交换，以此类推
4.二元选择排序 O(N2) 稳定
就是简单排序改良版，同时选取最小与最大的，最小放第一位，最大房最后一位。
5.堆排序 O(Nlog2N) 稳定
树形选择排序，下标为i的数小于或等于下标为2i与2i+1。堆顶元素，也就是树根，是最小项，这是最小堆。先把数组调整成堆，然后输出堆顶，再把剩下的数调整成堆，输出堆顶，依次类推。建堆先整理成完全二叉树，最后一个节点的根为n/2；然后从这个子树开始调整成堆，小的在左，大的在右，然后依次向前调整，直至树成堆。在输出一个堆顶后，将最左边的数放在堆顶，如果不满足堆，则根节点与左右子树中较小的交换，如果左右子树不满足堆，重复刚才的步骤，子树的根与左右中较小的交换。直到成堆。
6.冒泡排序 O(N2) 稳定
第一个与第二个换，大者放后面，然后第二个与第三个，第三个与四个，以此类推，第一轮结束，大的会出现在最后，然后对前面仍是未排序好的重复前面的步骤，直到有序
7.快速排序 O(Nlog2N) 稳定
随意选一个数，一般是第一个，然后开始遍历，小的放在它左边，大的放后边，然后这个数找到其最终位置，然后分别在左边的一组数和右边的一组数中，分别重复之前的步骤，以此类推。直到完全有序。
8.改进快速排序 O(Nlog2N) 稳定
取一个阈值，对阈值以后的数进行快速排序，然后对整个数组插入排序。
9.归并排序 O(Nlog2N) 不稳定
是对元组的递归排序，先把每个数看成一个元组，然后两两合并成一个元组，再在新的元组数组里面对两个元组两两合并，然后全部合并成一个元组。而两个元组合并的时候，把其中一个作为参照，比如A,B，，因为A、B本来就是有序的，把B作为参照，每一个B中的元素插入A中。
10.桶排序 O(N)，需要额外的空间开销
举个例子，有100个数，这些数是在1到200的范围内。然后设定一个区间长度，比如我选区间长度是20，就可以把数值在1到200按照递增分为十个桶，1-20，21-40，依次类推。然后对每个桶内进行任意排序。然后每个桶依次输出。当区间长度是1的时候，时间复杂度最低，但是需要空间复杂度最高。
11.基数排序 O(N) 稳定
先比较个位，排序，再比较十位，排序，依次类推，比较完最高位。如果是属性的话，先比较低优先级的属性，再比较高优先级的属性。

6.go语言评价
垃圾回收机制不完善，不支持动态加载类库，过度强调编译器速度导致功能不完善。
并发性好，部署简单，执行性能好。

7.linux的shell
关于shell编程我接触的不多，不过之前有一次部署网页在服务器上以服务启动，参考网上别人的方法修改了脚本。然后python的shell编程了解过一些。

8.项目

9.学习能力强，如何体现
10.比别人强的地方，把他放大

11.JVM
JVM的结构：
类加载器：在JVM启动时或者在类运行时将需要的class加载到JVM中。
执行引擎：负责class文件中包含的字节码指令。
内存区：JVM运行的时候操作所分配的内存区。
本地方法接口：主要是调用C/C++实现的本地方法及返回结果。

内存区域的划分：
程序计数器PC：当前线程所执行的字节码的行号指示器，不会抛出OOM。
JAVA虚拟机栈：存储基本数据类型和对象引用，生命周期和线程相同。调用方法的过程转换为栈帧入栈出栈的过程。64位的long和double会占用2个局部变量的空间。超出栈深度抛出SOF异常，无法申请内存抛出OOM。
本地方法栈：服务于native方法的调用。与JAVA虚拟机栈相似。
java堆：线程共享，虚拟机启动时创建，存储所有的对象实例及数组。垃圾回收的主要回收区域。分为新生代和老生代。堆中可能划分多个线程私有的缓冲区，可以通过-Xmx -Xms来调节堆大小，没有内存空间则OOM。
方法区：线程共享，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。现在没有了永久代，方法区放在了与堆不相连的本地内存区域—元空间。其最大问题为碎片化，也会OOM。
运行时常量池：存储在元空间中，主要存放Class字节码文件中包含的编译期生成的字面量和符号引用，也会在运行时增加。也会OOM。
直接内存：直接分配的堆外内存，以存储在堆中的DirectByteBuffer对象作为这块内存的引用直接操作。

如何定位OOM异常：
通用方法：运行时加入 -Xx:PrintGCDetails可以查看GC的日志信息。
java堆内存：配置参数 -Xx:+HeapDumpOnOutOfMemoryError可以在内存溢出时Dump出堆转储快照来分析。通过 -Xmx与 -Xms来改变堆大小，以免溢出。导出快照后，导入到Jvisualvm中进行查看，可以看到各个类对象所占用的空间百分比，根据这个来判断占用资源最多的对象是否需要，若需要扩大堆，否则查看程序BUG。
虚拟机栈和本地方法栈：栈容量只由-Xss参数来设定，无论增大栈帧还是缩小栈深度，都抛出SOF异常。且栈分配越大，越容易出现内存溢出，因为每个线程独享栈空间，多线程时很容易总和超过本机内存大小。此时需要减小堆大小，减小栈容量。
方法区和运行时常量池溢出：可以用-Xx:PermSize和-Xx:MaxPermSize来限制方法区大小，间接限制常量池容量，JDK8应使用-XX:MetaspaceSize来控制元空间大小。-XX:MinMetaspaceFreeRatio

与-XX:MaxMetaspaceFreeRatio来设置元空间空闲比例。使用动态代理和JSP应用时，会动态产生类，经常导致方法区内存溢出。
本机直接内存溢出：使用-Xx:MaxDirectMemorySize来指定，默认-Xmx。直接内存溢出的特点是堆Dump较小。

垃圾回收算法：
标记清除算法：先标记再回收(标记和清除效率都不高，且回收后会产生大量的内存碎片)
复制算法：使用一般，需要回收时将需要的部分移动到另一半，然后直接清除。(实现简单，运行高效，但是运行时内存少了一半)。改进版新生代分为Eden、From Survivor、To Survivor，比例是8:1:1。只浪费了百分之十，当Survivor不够用时，需要老年代来分配担保。
标记整理算法：老年代选用该算法，先标记，然后将还存活的对象向一段移动，并清理掉端边界以外的内存。
分代收集算法：即新生代使用复制算法，老年代使用标记整理算法。
分代的目的就是优化GC性能。新生代分为三部分：Eden、From Survivor、To Survivor。比例是8:1:1。新创建的对象都会被分配到Eden区，经过第一次Minor GC后，如果仍然存活则移到Suvivor区，对象在Survivor区每经历一次Minor GC都会增加一岁，到达一定年龄会移到老年代。新生代中的对象基本是朝生夕死(80%以上)，所以其回收算法是复制算法。内存分为两块，每次只用一块，当这一块用完时就将还活着的对象复制到另一块上，这不会产生碎片。GC开始时，对象只存在于Eden区和From Suvivor区，To区是空的。GC后，所有Eden活着的对象复制到To中，而From中仍然存活的没有到达年龄阈值的移往To，达到的移往老年代。Eden和From在这次GC后会被清空，而且From变成To，To变成From。保证GC后名为To那一块是空的，直到在一次GC中，To被填满，则其中所有对象移到老年代。

双亲委派机制：
JVM预定义了三种类型类加载器：

启动类加载器(bootstrap)：本地代码实现的类装入器，负责将JAVA_HOME/lib下面的类库加载到内存中。开发者无法获取类加载器的引用。
标准扩展类加载器：负责将JAVA_HOME/lib/ext下的类库加载到内存中。开发者可以直接使用标准扩展类加载器。
应用程序类加载器：负责将系统类路径(CLASSPATH)中指定的类库加载到内存中，开发者可以直接使用。
双亲委派机制的描述：某个特定类加载器在接到加载类的请求时，首先将加载任务委托给父类加载器，依次递归，如果父类加载器可以完成类加载任务，就成功返回，不然自己加载。越是基础的类越是被上层的类加载器进行加载，保证了java程序的稳定运行。
比如自己定义的java.lang.String类，在用String的时候，用BootstrapLoader进行加载，它会发现已经加载过了jdk的string就不会加载自定义的string，防止重复加载，保证了安全。

12.java并发编程
并发工具类
a) 等待多线程完成的CountDownLatch :允许一个或者多个线程等待其他线程完成操作。主线程需要等待其他线程完成后操作，可以使用Thread.join方法，让当前执行线程等待join线程执行结束，其实现原理就是不停地检查join线程是否存活，直到join线程终止后，线程的this.notifyAll()方

法会被调用。CountDownLatch也可以实现join的功能，构造函数接受一个int参数作为计数器，想等待N个线程或者一个线程的N个执行步骤完成就传入N，使用CountDownLatch的countDown方法，N回减1，await方法会阻塞当前线程，直到N变为0。也可以使用await来定时。

b) 同步屏障CyclicBarrier：让一组线程到达一个屏障时被阻塞，直到最后一个线程到达屏障，屏障才会开门，所有屏障拦截的线程才会继续执行。默认构造函数是CycliBarrier(int)，参数是表示屏障拦截的线程数量，每个线程通过await方法告诉CyclicBarrier该线程已经到达屏障，然后该线程被阻塞。CyclicBarrier多用于多线程计算数据，最后合并计算结果的场景。CyclicBarrier的计数器可以使用reset()方法重置。还有getNumberWaiting方法可以获取被阻塞的线程数量，isBoken方法来了解阻塞线程是否被中断。

c) 控制并发线程数的Semaphore：用来控制同时访问特定资源的线程数量。用于做流量控制，多用于公共资源有限或者数据库的连接。构造函数Semaphore(int)接受一个整型的数字，表示可用的线程数。使用acquire方法获取一个许可证，使用完调用release方法归还许可证。

d) 线程间交换数据的Exchanger：线程间进行协作的工具类，提供一个同步点，在这个同步点，两个线程可以用exchange方法彼此交换数据。多用于遗传算法，还可以进行两个人的校对工作

线程池

a)原理：线程池先判断核心线程里的线程是否都在执行任务，如果不是则创建一个新的工作线程来执行任务；线程池判断工作队列是否已满，如果未满则将新提交的任务存储在这个工作队列中；判断线程池中的线程是够都处于工作状态，如果没有则创建线程执行任务，如果满了交给饱和策略 。

b)线程池的使用：通过ThreadPoolExecutor创建线程池。
corePoolSize参数指定线程池基本大小，当提交一个任务到线程池时，不管是否有空闲进程都会创建新进程，等到需要执行的任务数大于线程池基本大小就不会再创建。如果调用了线程池prestartAllCoreThreads方法，线程池会提前创建并启动所有基本线程。
RunableTaskQueue用于保存等待执行的任务的阻塞队列。
maxiumPoolSize线程池最大数量，使用无界的任务队列，这个参数就没什么效果。
ThreadFactory用于设置创建线程的工厂，给线程设置更有意义的名字。
RejectedExecutorHandler饱和策略，当队列和线程池都满了，说明线程池处于饱和状态，可以用四种策略：直接抛出异常；只用调用者所在的线程来运行任务；丢弃队列里最近的一个任务并执行当前任务；不处理。
KeepAliveTime线程活动保持时间：线程的工作线程空闲后，保持存活的时间，如果任务很多而且每个任务的执行时间很短，那么可以调大时间保持线程利用率。
TimeUnit：线程活动保持时间的单位。

c)向线程池提交任务：Execute方法用于提交无需返回值的任务，无法判断线程是否执行成功；
Submit()方法用于提交需要返回值的任务，线程池会返回一个future类型的对象，通过这个future对象可以判断任务是否执行成功，通过future的get()方法取得返回值，get会阻塞当前线程直到任务的完成。

d)关闭线程池：遍历线程池中的工作线程，然后逐个调用线程的interrupt方法来中断线程。
Shutdown：将线程池状态设置成SHUTDOWN，然后中断所有没有在执行的任务的线程，
ShutdownNow：将线程池状态设置成STOP，然后尝试停止所有在执行或暂停任务的线程。

e) 合理配置线程池：
首先分析任务特性：

1.CPU密集任务、IO密集任务、混合任务。
2.任务优先级：高中低
3.任务执行时间：长中短
4.任务依赖性：是否依赖其他资源
CPU密集型应该配置尽可能小的线程，NCPU+1个线程的线程池。IO密集型不是一直在执行，所以尽可能多地配置多的线程，2N *CPU。优先级不同的采用优先队列PriorityBlockingQueue来处理。依赖数据库连接的任务，因为线程提交SQL，后面需要等待数据库返回结果，等待时间越长则CPU空闲时间就越长，那么线程数应该设置的越大，更好的利用CPU。
建议使用有界队列。

f)线程池监控：方便在出现问题的时候，可以根据线程池的使用状况快速定位问题，可以通过参数进行监控。属性如下：
taskCount:线程需要执行的任务数量
completedTaskCount:线程在运行过程中已完成的任务数量
largestPoolSize:线程池里曾经创建过的最大线程数
getPoolSize:线程池的线程数量
getActiveCount:获取活动的线程数
自定义线程继承线程池重写代码-实现监控

Executors框架：
Executor框架是一个根据一组执行策略调用，调度，执行和控制的异步任务的框架。无限制的创建线程会引起应用程序内存溢出，所以创建一个线程池是更好地解决方案，可以限制线程数量并且可以回收利用这些线程。利用Executors框架可以非常方便的创建一个线程池。

13.spring 框架
spring特点：spring框架的核心功能可以应用在任何java应用框架，对java ee平台上的web应用程序有更好的扩展性。其目标是使java ee应用程序的开发更加简捷，通过使用POJO为基础的编程模型促进良好的编程风格。轻量级。控制反转。面向切面编程。容器。MVC框架。事务管理。异常处理。java项目都是向解耦化和组件化发展，提高代码复用率，而且便于维护。

控制反转：运行时被装配器对象来绑定耦合对象的一种编程技巧，对象之间耦合关系在编译时是未知的。传统编程方式中，业务逻辑的流程是由应用程序中的早已被设定好关联关系的对象来决定的。在使用控制反转的情况下，业务逻辑的流程是由对象关系图来决定，该关系对象图由装配器负责实例化，这种实现方式还可以将对象之间的关联关系的定义抽象化。而绑定的过程是通过“依赖注入”实现的。控制反转是一种以给予应用程序中目标组件更多控制为目的设计范式，并在我们的实际工作中起到了有效作用。就是被调用类的实例由原先的调用类控制创建、销毁转变成由Spring的容器管理。
控制反转即IoC，它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”概念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。
IoC是一个很大的概念，可以用不同的方式来实现。其主要实现方式有两种：<1>依赖查找：容器提供回调接口和上下文环境给组件。EJB和Apache Avalon都使用这种方式。<2>依赖注入：组件不做定位查询，只提供普通的Java方法让容器去决定依赖关系。后者是时下最流行的IoC类型，其又有接口注入，设值注入和构造子注入三种方式。

依赖注入：是在编译阶段尚未知所需的功能是来自哪个类的情况下，将其他对象所依赖的功能对象实例化的模式。依赖注入是控制反转的基础。在JAVA中依赖注入有：构造器、SETTER方法、接口注入。

BeanFactory：BeanFactory可以理解为含有bean集合的工厂类。包含了种bean的定义，以便于在接收到客户端请求时将对应的bean实例化。BeanFactory还能在实例化对象时生成协作类之间的关系，将Bean自身与bean客户端的配置中解放出来。BeanFactory还包含了bean生命周期的控制、调用客户端的初始化方法和销毁方法。

Application Context：如同bean factory一样具有bean定义、关联关系设置，根据请求分发bean的功能。但ApplicationContext还再次基础上提供了其他功能：支持国际化的文本消息、统一的资源文件读取方式、已在监听器中注册的bean事件。
三种常见的ApplicationContext实现方式：
1.ClassPathXmlApplicationContext:从classpath的xml配置文件中读取上下文。并生成上下文定义。应用程序上下文从程序环境变量中取得。
2.FileSystemXmlApplicationContext:从文件系统的Xml配置文件中读取上下文。
3.XmlWebApplicationContext:由Web应用的Xml文件读取上下文。

Spring框架安全性：
1.Spring并没有对单例bean进行任何多线程的封装处理。关于单例bean的线程安全和并发问题需要开发者自己去搞定，但实际上大部分的spring bean并没有可变状态(比如Servlet、DAO类)，所以某种程度上Spring单例bean是线程安全的。如果你的bean有多重状态的话(View、Model对象)就需要自行保证线程安全。
2.最浅显的解决办法就是将多态bean的作用于由“Singleton”变更“prototype”。

Spring AOP：分离不变和变化的代码，提取相同的代码，合并横切关注点。AspectJ是java语言的AOP实现。AOP实现可以分为两类，一种是静态AOP，在编译阶段就对程序进行修改，增强了目标类。一种是动态AOP，是AOP框架在运行阶段动态生成AOP代理(在内存中以JDK动态代理生成了目标类的接口或cglib动态地生成AOP代理类即目标类的子类)，以实现对目标的增强。Spring的AOP默认使用JDK的动态代理，也可以使用cglib，如需对成员变量的访问和更细也做为增强处理的连接点，则可以考虑使用AspectJ。具体使用哪种来生成AOPProxyFactory根据AdvisedSupport对象的配置来决定，默认的策略师如果目标类是接口，则使用JDK动态代理技术，否则使用cglib。

AOP基本概念：切面Aspect、连接点JoinPoint、增强处理(通知)Advice、切入点Piontcut、引入、目标对象、AOP代理、织入Weaving(编译时增强和运行时增强)。

配置AOP的过程：在配置文件中开启aop:aspectj-autoproxy注解支持。定义业务bean和切面@Aspect，定义切入点@PoinCut(execution(*org(…)))、定义增强处理(通知、前置@Before、后置@After、异常@AfterThrowing、返回@AfterReturning<可以注解returning属性获得返回值>、环绕@Around<形参必须使用ProceedingJoinPoint类型，执行目标方法为该对象proceed()方法，该类型还可以得到方法传入的参数、目标对象和代理对象>)

事务：一系列的动作，综合在一起才是一个完整的工作单元，必须全部完成，有一个失败就会回滚到最开始状态。事务管理来保证数据完整性和一致性。

事务特性：1.原子性：一系列动作当做一个整体，是一个原子操作，确保全部完成或全部不起作用

2.一致性：一旦事务完成，无论成功与否，系统必须确保它所建模的业务处于抑制状态，而不会是部分成功部分失败，在现实中的数据不应被破坏。
3.隔离性：可能许多事务会同时处理相同数据，因此事务之间隔离开来，放置数据破坏
4.持久性：一旦事务完成，无论发生什么系统错误，其结果都不应该受到影响，这样就能从任何系统崩溃中恢复过来。

事务管理器：Spring并不直接管理事务，而是提供多种事务管理器，它们将事务管理的职责委托给Hibernate或者JTA等持久化机制所提供的相关平台框架的事务来实现。接口提供了getTransaction、commit、rollback等方法。为不同事物API提供一致编程模型。

事务属性：
1.传播行为：当事务方法被另一个事务方法调用时需指定事务如何传播。
2.隔离级别：定义一个事务可能受其他并发事务影响的程度。
并发事务引起的问题：多个事务并发运行，经常会操作相同的数据来完成各自任务，并发虽然是必须的，但可能会导致：脏读、不可重复读、幻读。
3.只读：数据库以此优化
4.事务超时：超时自动回滚，释放资源。
5.回滚规则：遇到运行期异常才会回滚，检查型异常不回滚。可以自己定义哪些异常回滚哪些不回滚。

事务状态：是否为新事物、是否有恢复点、设置为只回滚、是否为只回滚、是否已完成。
编程式事务：允许用户在代码中精确定义事务的边界，侵入到业务代码里，提供了详细的事务管理，而声名式事务基于AOP有助于用户将操作做与事务规则进行解耦，不影响业务代码的具体实现。

14.设计模式
类的三种关系：依赖(use-a)、聚合(has-a)、继承(is-a)。
设计模式六大原则：
1.开闭原则：对扩展开放、对修改关闭。接口和抽象类
2.里氏代换：任何基类可以出现的地方，子类也可以出现。
3.依赖倒转：针对接口编程，依赖于抽象而不依赖于具体。
4.接口隔离：使用了多个隔离的接口优于单个的接口。降低依赖、降低耦合
5.迪米特法则(最少知道原则)：一个实体应当尽量少的与其他实体之间发生相互作用，使得系统功能模块相对独立。
6.合成复用原则：原则是尽量使用合成/聚合的方式，而不是使用继承。

23种设计模式：
1.工厂方法模式：
a.普通工厂：建立一个工厂类，对实现同一接口的一些类进行实例创建。
b.多个工厂方法模型：普通的改进，普通如果传递字符串出错，则无法创建对象，多个工厂模型提供多个工厂方法分别创造对象。
c.静态工厂方法模型：将上面的多个工厂方法模型里的方法设置为静态，不需要创建实例，直接调用。

2.抽象工厂模式：工厂方法模式的类创建依赖于工厂，如果扩展程序，必须对工厂类进行修改，违背闭包原则，创建多个工厂类，一旦需要增加新的功能，直接增加新的工厂类即可，无需修改之前代码。
3.单例模式：保证在jvm中该对象只有一个实例存在。好处：某些类创建比较频繁，对于大型对象，这是很大的系统开销。省去new操作符，降低了系统内存的使用频率，减轻GC压力。有些类如同交易所的核心交易引擎，只有单例模式才能保证交易服务器独立控制整个流程。如果没有线程安全保护的类放入多线程环境下就会出问题，可以再getInstance方法前加入synchronized关键字，再改进可以再只有第一次创建对象的时候需要上锁，在getInstance内部，先判断instance是否为null，若是便对instance上锁。但是JAVA里面新建和复制是分开操作，可以构建一个内部类来维护单例。
4.建造者模式：工厂类模式提供的是创建单个类的模式，而建造者模式是将各种产品集中进行管理，创建复合对象。
5.原型模式：将一个对象为原型，对其复制克隆，产生和原对象类似的新对象。
6.适配器模式：将某个类的接口转换成客户端期望的另一个接口表示，目的是消除由于接口不匹配所造成的类的兼容性问题。分三类：
a.类的适配器模式：一个Source类，拥有一个待适配的方法，目标接口是Targetable，通过Adapter类，将Source的功能扩展到Targetable里。这样Targetable接口的实现类就具有了Source类的功能。希望一个类转换成满足另一个新接口的类时，创建一个新类继承原有的类实现新的接口。
b.对象的适配器模式：基本思路与类的适配器模式相同，只是将Adapter类作修改，不继承Source类，而是持有Source类的实例，以达到解决兼容性的问题。当希望一个对象转换成满足另一个新接口的对象时，创建一个Wrapper类，持有原类的一个实例，在Wrapper类的方法中，调用实例的方法。
c.接口的适配器模式：如果我们的一个借口拥有多个抽象方法，当我们写该接口的实现类时，必须实现该接口的所有方法，这明显浪费时间，并不是所有方法都是被需要的，所以我们引入了接口的适配器模式，借助于一个类，该类实现了该接口，实现了所有方法，而我们写一个类继承该类，就只需重写我们需要的方法。当不希望实现一个接口的所有方法时，可以创建一个类Wrapper，实现所有的方法，我们写类的时候直接继承该类。
7.装饰模式：给一个对象增加一些新的功能，而且是动态地，要求装饰对象和被装饰对象实现同一个接口，装饰对象持有被装饰对象的实例。Source类是被装饰类，Decorator是装饰类，可以为Source类动态添加一些功能。应用场景：需要扩展一个类的功能；动态为一个对象增加功能，还能动态撤销，但是会产生过多相似的对象，不易排错。
8.代理模式：代理模式就是多出一个代理类，代替原对象进行一些操作。应用场景：如果已有的方法在使用的时候需要对原有的方法进行改进：如果修改原有的方法就违反了开闭原则，所以采用一个代理类调用原有方法，且对产生的结果进行控制，可以将功能划分的更加清晰，有助于后期维护。

9.外观模式：为了解决类与类之间的依赖关系，将其关系放在一个Facade类中，降低类之间的耦合度，该模式没有涉及到接口。
10.桥接模式：将事物和其具体实现分开，使他们可以各自独立的变化。用意：将抽象化和实现化解耦，使得二者可以独立变化。
11.组合模式：又名部分-整体模式，在处理树形结构问题比较方便。场景：将多个对象组合一起操作。
12.享元模式：主要为了实现对象的共享，即共享池。当系统中对象多的时候可以减少内存开销，通常与工厂模式一起使用。
13.策略模式：定义了一系列算法，并将每个算法疯涨，使其可以互相替换，且算法的变化不会影响到使用算法的客户。需要设计一个接口，为一系列实现类提供统一方法，多个实现类实现接口，设计一个抽象类(可有可无，属于辅助类)，提供辅助函数。策略模式决定权

14.模板方法模式：一个抽象类中，有一个主方法，再定义n个方法，可以抽象可以实际，定义一个类，继承该抽象类，重写抽象方法，通过主方法调用抽象类，实现对子类的调用。
15.观察者模式：当一个对象变化时，所有其他依赖该对象的对象都会收到通知，并且随之变化。
16.迭代器模式：顺序访问聚集中的对象。MyCollection定义了集合的操作，MyIterator定义了一系列迭代操作，且持有Collection实例。
17.责任链模式：有多个对象，每个对象持有对下一个对象的引用，这样就会形成一条链，请求在这条链上传递，直到某一对象决定处理该请求，但是发出者不清楚哪个对象会处理。可以在隐瞒客户端的情况下，对系统进行动态调整。链接上的请求可以是一条链，可以使一个树或一个环，模式本身并不约束，需要我们自己实现，在同一时刻，命令只允许由一个对象传给另一个对象。
18.命令模式：举个例子：司令员下令让士兵去干件事情，整个事情来考虑，司令员的作用是发出口令，口令经过传递，到了士兵，士兵去执行。三者互相解耦，任何一方都不用依赖他人，只需做好自己的事。目的就是达到命令的发出者和执行者之间解耦，实现请求和执行分开。
19.备忘录模式：主要目的是保存一个对象的某个状态，以便在适当的时候恢复对象。假设有原始类A，A中有各种属性，A可以决定需要备份的属性，备忘录类B用来存储A的一些内部状态，类C用来存储备忘录，且只能存储，不能修改。
20.状态模式：当对象状态改变，同时改变其行为。通过改变状态来改变行为，且你的好友能看到你的变化。
21.访问者模式：把数据结构和作用于结构上的操作解耦合，使得操作集合可以相对自由地演化，适用于数据结构相对稳定和算法又易变化的系统。将有关行为集中到一个访问者对象中，其改变不影响系统数据结构，缺点是增加新的数据结构很难。适用场景：香味一个现有类增加新功能，考虑以下几点：会不会与现有功能出现兼容问题，会不会需要再添加，如果类不允许修改代码怎么办？这时使用访问者模式。

22.中介者模式：降低类之间耦合，因为如果类之间有依赖关系，不利于功能的扩展和维护，因为只要修改一个对象，所有关联对象都要修改，中介者模式只关心和Mediator类之间的关系，具体类类之间的关系和调度交给Mediator就行
23.解释器模式：一般用于编译器开发，如正则表达式。

15.mysql
索引：
1.越小的数据类型需要的空间越小，处理更快；简单的数据类型越好，整形比字符处理开销更小，应使用内置的日期和时间数据类型，而不是字符串，以及用整形存储IP地址；避免使用NULL。
2.整形是最好的选择，更快的处理，可以设置为AUTO_INCREMENT；字符串：消耗更大的空间，处理慢，索引的位置也是随机的，导致页面分裂。
3.索引类型：B树索引；Hash索引，只有Memory存储引擎显式支持hash索引，由于索引仅包含hash code和记录指针，所以MYSQL不能通过使用索引避免读取记录，不能使用hash索引排序，hash索引不支持键的部分匹配，它是通过整个索引值来计算hash值的，hash索引只支持等值比较；空间索引(R-TREE)：MyISAM支持空间索引，主要用于地理空间数据类型；全文索引(Full-text)全文索引是MyISAM的一个特殊索引类型，主要用于全文检索。
4.索引原则：搜索的索引列不一定是所要选择的列；使用唯一索引；使用短索引；利用最左前索引；不要过度索引；考虑在列上进行的比较类型。
5.高性能索引策略：
聚簇索引：保证关键字的值相近的元组存储的物理位置也相同，且一个表只能有一个聚簇索引。
覆盖索引：如果索引包含满足查询的所有数据，就成为覆盖索引。能大大提高查询性能，只需要读取索引而不用读取数据，有以下优点：索引项比记录小，MYSQL访问更少的数据；索引都按值的大

小顺序存储，相比随机访问需要更少的IO；大多数据引擎能更好的缓存索引；对InnoDB表尤为有用；覆盖索引不能使任何索引，只有B-TREE索引存储相应的值。
利用索引进行排序：一种是使用fileSort，二时按索引顺序扫描。
索引与加锁：索引对InnoDB非常重要，它可以让查询锁更少的元组。

优化：
选用最适用的字段属性：表越小查询越快，表中字段宽度尽可能小；字段设置为NOTNULL；某些文本如省份、性别设置为ENMU类型。
使用连接(JOIN)代替子查询(SUB-QUERIES)：因为MYSQL不需要再内存中创建临时表来完成这个逻辑上的需要两个步骤的查询工作。
使用联合(UNION)：把临时表的两条或者更多的SELECT查询合并到一个查询中。
事务：在一系列的语句中，可能某一条语句出错，那么整个语句块的操作都会变得不确定，比如一个数据同时插入两个关联的表中，可能第一个表成功更新的时候数据库出现意外状况导致第二个表没有完成会造成数据不完整甚至破坏数据库中的数据。事务使语句块中的每一条要么都成功要么都失败，保持数据库中数据一致性和完整性。
锁定表：事务虽然维护数据库完整但是其独占性会影响数据库性能，在事务执行中数据库会被锁定，其他用户只能等待，可以用锁定表获得更好的性能。
使用外键：锁定表能维护数据完整性却不能保证数据库关联性，外键能保护关联性。
使用索引：提高数据库性能，令数据库服务器以更快的速度检索特定的行。
优化的查询语句：SQL语句使用不当索引无法发挥作用。

16.中间件

我们对于消息系统、数据库、服务化接口的抽象等，涉及数据分离的过程中，在分离过程中，就会涉及到分离后系统间、数据库间的交互。java中间价就是我们处理数据间交互、连接数据分离后两个系统间的通信。中间件不属于任何一个开发项目，就是让我们对应系统间或者数据库间的数据流通无感知。
1.远程过程调用和对象访问中间件：主要解决分布式环境下应用的互相访问问题，这也是支撑应用服务化功能的基础。
2.消息中间件：解决应用之间的消息传递、解耦、异步问题。
3.数据访问中间件：主要解决应用访问数据库的共性问题的组件。
以JDBC为例，数据库本地维护了一个数据访问中间件，在访问数据库的时候，配置的地址其实是直接连到JDBC这个数据访问中间件，如果我们执行查询数据或者对数据库的操作都是通过JDBC来连接数据库，然后通过JDBC查询完成数据库以后再返回给我们应用程序，而其中查询过程我们是不可知的。

17.socket

socket两种通信协议可以选择，一种是数据报通信，另一种是流通信。
数据报通信：UDP，无连接的协议，每次发送数据包都要发送本机的socket描述符合接收端的socket描述符，每次通信都要额外的数据。
流通信协议：TCP，基于连接，在使用前，在一对socket之间建立连接，其中一个座位服务器进行监听连接请求，另一个客户端进行连接请求。一旦连接好就可以单向或者双向进行数据传输。

18.多线程

多线程

1. java中有几种方法可以实现一个线程?

可以使用Runnable，Callable，Thread或者线程池

2. 如何停止一个正在运行的线程?

条件变量，可以使用正在运行的线程，支持线程中断，那么可以使用线程中断的方法停止这个线程；通常是定义一个volatile的状态变量，在运行线程线程中读这个变量，其它线程中修改这个变量；或者使用isInterrupted()方法。

3 notify()和notifyAll()有什么区别?

随机唤醒和唤醒全部，notify是随机唤醒一个等待某个资源的线程，进入就绪队列等待CPU的调度，notifyAll是唤醒所有的，进入就绪队列等待CPU调度

4. sleep()和 wait()有什么区别?

定时等待和在监视器上等待，不同范畴。sleep方法是在指定的时间内让正在执行的线程暂停执行，但不会释放锁。而wait方法是让当前线程等待，直到其他线程调用对象的notify或notifyAll方法。wait方法会释放掉锁，使别的线程有机会占用锁。

5. 什么是Daemon线程？它有什么意义?

守护线程，不需要上层逻辑介入的后台线程 ，比如GC。守护线程是后台线程，它通常属于低优先级的线程，守护用户线程，通常做一些垃圾清理的工作，比如GC。当所有的用户线程退出后，守护线程不会立即退出，还出执行一段时间。

6. java如何实现多线程之间的通讯和协作?

中断和 直接或间接访问对方实例。同步和互斥，资源互斥、协调竞争是要解决的因，同步是竞争协调的果。可以使用synchronized/notify/notifyAll以及Lock/Condition, CyclicBarrier/Semaphore/Countdownlatch。线程的join以及Future/FutureTask是为了解决多线程计算后的结果统计

锁

1. 什么是可重入锁（ReentrantLock）？

更高级的锁，附加更多特性。ReentrantLock 相对于固有锁synchronized，同样是可重入的，在某些vm版本上提供了比固有锁更高的性能，提供了更丰富的锁特性，比如可中断的锁，可等待的锁，平等锁以及非块结构的加锁。从代码上尽量用固有锁，vm会对固有锁做一定的优化，并且代码可维护和稳定。只有在需要ReentrantLock的一些特性时，可以考虑用ReentrantLock实现

2. 当一个线程进入某个对象的一个synchronized的实例方法后，其它线程是否可进入此对象的其它方法?

可进入非synchronized方法

3. synchronized和java.util.concurrent.locks.Lock的异同?

后者具有更丰富的特性

4. 乐观锁和悲观锁的理解及如何实现，有哪些实现方式?

乐观锁是假设我已经拿到锁，悲观所是我必须拿到锁，前者用CAS，后者用mutex

并发框架

1. SynchronizedMap和ConcurrentHashMap有什么区别?

后者具有更高的并发

2. CopyOnWriteArrayList可以用于什么应用场景?

多读少写

线程安全

1. 什么叫线程安全？servlet是线程安全吗?

在多线程调用情况下，依然表现正常

2. 同步有几种实现方法?

锁和volatile

3. volatile有什么用？能否用一句话说明下volatile的应用场景?

保持可见性，在1写N读的情况下比较适合

4. 请说明下java的内存模型及其工作流程。

JVM的内存模型可简单理解为，有一块整个JVM共享的主内存，每个线程有自己的变量复本，线程持有的复本与主内存之间使用内部的数据协议

5. 为什么代码会重排序?

编译器旨在提升性能

并发容器和框架

1. 如何让一段程序并发的执行，并最终汇总结果?

使用CyclicBarrier 在多个关口处将多个线程执行结果汇总

CountDownLatch 在各线程执行完毕后向总线程汇报结果

1. 可以使用Callable+FutureTask+Executors+Future操作，让收集结果的线程阻塞等待与get()方法等待结果
2. 可以使用Thread的join方法来阻塞等待结果
3. 可以使用CountDownLatch来让收集结果的线程等待
4. 可以使用CyclicBarrier来收让收集结果的线程等待，

CountDownLatch是CyclicBarrier的一种，所以CountDownLatch能解决的问题，CyclicBarrier一定可以

2. 如何合理的配置java线程池？如CPU密集型的任务，基本线程池应该配置多大？IO密集型的任务，基本线程池应该配置多大？用有界队列好还是无界队列好？任务非常多的时候，使用什么阻塞队列能获取最好的吞吐量?

IO密集型时，大部分线程都阻塞，故需要多配置线程数，2*cpu核数

有界队列和无界队列的配置需区分业务场景，一般情况下配置有界队列，在一些可能会有爆发性增长的情况下使用无界队列。

任务非常多时，使用非阻塞队列使用cas操作替代锁可以获得好的吞吐量。

3. 如何使用阻塞队列实现一个生产者和消费者模型？请写代码。

4. 多读少写的场景应该使用哪个并发容器，为什么使用它？比如你做了一个搜索引擎，搜索引擎每次搜索前需要判断搜索关键词是否在黑名单里，黑名单每天更新一次。

CopyOnWriteArrayList这个容器适用于多读少写…读写并不是在同一个对象上。在写时会大面积复制数组，所以写的性能差，在写完成后将读的引用改为执行写的对象

Java中的锁

1. 如何实现乐观锁（CAS）？如何避免ABA问题?

比较内存值和期望值,替换内存值为要替换值,带参数版本来避免aba问题，在读取和替换的时候进行判定版本是否一致

2. 读写锁可以用于什么应用场景?

多读少写，读写锁支持多个读操作并发执行，写操作只能由一个线程来操作

3. 什么时候应该使用可重入锁?

重入锁指的是在某一个线程中可以多次获得同一把锁，在线程中多次操作有锁的方法。可轮询，可中断，定时，非块，公平队列等高级特性时候使用可重入锁

4. 什么场景下可以使用volatile替换synchronized?

只需要保证共享资源的可见性的时候可以使用volatile替代，synchronized保证可操作的原子性一致性和可见性。volatile适用于新值不依赖于就值的情形,单线程修改变量或不依赖当前值，且不与其他变量构成不变性条件时候使用volatile

并发工具

1. 如何实现一个流控程序，用于控制请求的调用次数?

ReentrantLock 和synchronized比较：

性能上：

1.为什么JUC框架出现LOCK?

ReentrantLock并不是替代synchronized的方法，而是当内置锁不适用时，作为一种可选的高级功能。

2.那么Synchronized有哪些缺点?

①. 只有一个condition与锁相关联，这个condition是什么？就是synchronized对针对的对象锁。

②. synchronized无法中断一个正在等待获得锁的线程，也即多线程竞争一个锁时，其余未得到锁的线程只能不停的尝试获得锁，而不能中断。这种情况对于大量的竞争线程会造成性能的下降等后果。

可见ReentrantLock 是对synchronized补充。

3.我们面对ReentrantLock和synchronized改如何选择?

Synchronized相比Lock，为许多开发人员所熟悉，并且简洁紧凑，如果现有程序已经使用了内置锁，那么尽量保持代码风格统一，尽量不引入Lock，避免两种机制混用，容易令人困惑，也容易发生错误。

在Synchronized无法满足需求的情况下，Lock可以作为一种高级工具，这些功能包括“可定时的、可轮询的与可中断的锁获取操作，公平队列，以及非块结构的锁”否则还是优先使用Synchronized。

最后，未来更可能提升Synchronized而不是Lock的性能，因为Synchronized是JVM的内置属性，他能执行一些优化，例如对线程封闭的锁对象的锁消除优化，通过增加锁的粒度来消除内置锁的同步，而如果基于类库的锁来实现这些功能，则可能性不大

Collection

1. Java集合Collection框架是什么？列出集合框架的一些好处?

在每一种编程语言都有集合的使用，最初的Java版本包含了几类集合：向量，堆栈，哈希表，数组。但在更大的范围使用是在Java 1.2中集合框架想出了该组的所有集合接口，实现和算法。 Java集合的线程安全操作和使用泛型和并发集合类等。它还包括阻塞的接口及其实现在Java并发包。

集合框架的好处是：

使用核心集合类，而不需要实现我们自己的集合类，减少了开发工作。

使用经过测试的集合框架类提高代码质量。

使用JDK附带的集合类减少代码维护的工作。

可重用性和互操作性

2. 泛型集合框架的好处是什么呢?

Java 1.5中附带泛型和所有收集接口和接口实现的大量使用。泛型允许我们提供一个集合可以包含Object类型，所以如果你尝试添加任何其他类型的元素，它会引发编译时错误。这就避免了在运行时抛出，因为你会得到编译错误。泛型使代码更干净，因为我们并不需要使用溯型casting和instanceof检查。它也增加了运行时的好处，因为不生成的做类型检查字节码指令。

3. Java集合框架的基本接口是什么?

Collection是集合层次的根。一个集合包含一组对象作为其元素。 Java平台不提供任何直接实现这个接口。

Set 是一个不能包含重复的元素的集合。此接口模型代表数学Set的抽象，用来代表一组Set，如一副扑克牌。

List是有序集合，可以包含重复的元素。您可以从它的索引访问任何元素。更像是动态长度的数组列表。

一个Map是一个键映射值的对象。一个Map不能包含重复键：每个key只能映射一个值。

其他一些接口Queue, Dequeue, Iterator, SortedSet，SortedMap的和listIterator。

4. 为什么要集合不能继承Cloneable和Serializable接口?

Collection接口指定一组称为元素的对象。元素如何被组织取决于具体实现。例如，一些LIST实现允许重复的元素，而SET不允许。Collection是一种抽象表示，而克隆和序列化重在执行，应该是在Collection具体实现子类中根据具体元素组织情况来实现。因此，强制在所有实现都要有克隆和序列化是不够灵活的，具有限制性。

5. 为什么MAP接口不实现Collection接口?

虽然Map接口和它的实现是集合框架的一部分，但是MAP不是集合，而且集合也不是地图。因此，它实现集合接口没有任何意义。

6. Iterator是什么?

Iterator接口提供遍历集合的方法。从一个集合中使用迭代方法，我们可以得到迭代器实例。迭代器允许呼叫者在迭代过程中从集合中删除元素。

7. 枚举Enumeration 和Iterator接口之间的差异是什么?

枚举是快迭代两倍，使用非常少的内存。枚举适合基本需求。但Iterator是更安全，因为它总是拒绝其他线程修改它正在迭代集合中的对象。

8. 为什么没有方法像Iterator.add()将元素添加到集合?

这是语义不清，对于迭代一个集合，必须保证迭代的顺序。但是请注意，ListIterator确实提供了一个add的操作，而且它保证迭代的顺序。

9. 为什么迭代器没有不用移动光标来直接获得下一个元素的方法?

它可以基于当前Iterator接口之上实现，但很少使用，没有意义。

10. Iterator和listIterator之间有什么不同?

我们可以使用迭代器Iterator遍历Set和List集合，而ListIterator只可以使用List。

迭代器遍历只有向前的方向，而ListIterator可以用来在两个方向遍历。

ListIterator继承Iterator接口，并配备了额外的功能，如添加元素，更换一个元素，能获得上一个和

下一个元素的索引位置。

1. 使用iterate遍历列表List的不同方法是什么？

```
List<String> strList = new ArrayList<>();
//using for-each loop
    for(String obj : strList){
       System.out.println(obj);
    }
    //using iterator
    Iterator<String> it = strList.iterator();
    while(it.hasNext()){
       String obj = it.next();
       System.out.println(obj);
    }
```

2. 你怎么理解迭代器的快速失败fail-fast 的特点?

迭代器的快速失败fail-fast 属性会检查对集合的结构进行任何的修改，每次我们尝试获得下一个元素。如果发现有任何修改，它抛出ConcurrentModificationException。所有迭代器的实现都是快速失败的设计，除了像ConcurrentHashMap和CopyOnWriteArrayList。

3. 快速失败 fail-fast和故障安全fail-safe之间的不同是什么?

迭代器故障安全性fail-safe以克隆方式工作，因此它不会影响集合中的任何修改。所有java.util包中的集合类是快速失败的fail-fast，而java.util.concurrent中的类都是故障安全fail-safe。快速失败迭代器抛出ConcurrentModificationException，而失败安全fail-safe的迭代器从不抛出ConcurrentModificationException。

4. 迭代集合时如何避免ConcurrentModificationException?

使用并发集合类来避免ConcurrentModificationException， 如使用CopyOnWriteArrayList 替代ArrayList.

5. 为什么没有Iterator接口的具体实现?

每个集合返回一个迭代器用来遍历自己。 这使得集合类能够选择是否迭代器是快速失败 fail-fast或故障安全fail-safe。例如ArrayList的迭代器是快速失败的，而CopyOnWriteArrayList的迭代器是故障安全。

6. UnsupportedOperationException的是什么?

UnsupportedOperationException异常，用于指示该操作不被支持。它广泛用于在JDK类，，对于所有添加和删除操，集合框架java.util.Collections.UnmodifiableCollection抛出这个异常。

7. HashMap如何工作?

HashMap以哈希算法方式工作，在put和get方法被调用时，使用hashCode（）和equals（）来配合：当我们使用put方法，HashMap使用key的hashCode（）散列在键 - 值存储对中找出索引。k-v条目Entry存储在LinkedList，因此如果有已经存在的Entry，它就使用equals（）方法来检查其相应的键key是否已经存在，如果是的，它覆盖原值，否则它创建一个新Enrty条目和存储这个键-值；当

我们通过键key调用get方法，再次使用的hashCode（）找到数组中的索引，然后使用equals（）方法来找到正确的条目，并返回它的值。

HashMap的容量涉及加载因子，阈值调整大小。 HashMap的初始默认容量为32和负载系数为0.75。阈值是容量乘以负载因子，每当我们尝试添加一个条目，Map的大小如果大于阈值时，HashMap扩大Map内容到一个新的更大阵列容量。容量总是2的幂，所以，如果你知道你需要存储大量的键 - 值对，例如，在数据库中的数据缓存，初始化HashMap中正确的容量和负载因子是个好主意。

8. hashCode（）和equals（）的关系是什么?

如果o1.equals（O2），o1.hashCode（）== o2.hashCode（）应该永远是真的
如果o1.hashCode（）== o2.hashCode， 它并不意味着o1.equals（O2）将是真的

9. 能使用任何类作为Map的key吗?
可以，但是遵循：
如果类重写equals（）方法，还应该覆盖重写hashCode（）方法。
如果一个类的字段没有在equals（）方法使用，你不应该使用它的hashCode（）方法
用作键Key的类最好是不可变 的，hashCode（）值可以缓存以提高运行速度。

10. Map 提供的Collection 视图之间有什么区别?
Set keySet():返回此Map中包含的键的Set视图。
Collection values():返回此Map中包含的值的Collection视图
Set<Map.Entry<K, V>> entrySet():返回在此Map中包含的映射关系的Set视图

1.callable
callable能返回结果，runnable不能返回结果，future可以拿到callable的返回结果。

2.所有的类都继承于object类，你用过的object的直接子类有哪些，object的常用方法有哪些?
Boolean，Character, Class, Math, Void等等。protected Object clone(),int hashCode(), String toString(), equals(), getClass(),

3.String, StringBuffer, StringBuilder区别
String：声明是public final，不允许修改的，如果修改其值就会在内存多创建一个空间来保存新的字符串值。一般使用另外两个。
StringBuffer和StringBuilder继承AbstractStringBuilder，但是StringBuffer大部分方法是synchronized，线程安全的，而StringBuilder却没有。而且StringBuilder可以操作StringBuffer，反过来不行。正是因为其要维持安全锁，所以StringBuffer没有StringBuilder效率高。

4. arraylist和linklist的区别，hashmap和hashset的区别，常用的集合有哪些

ArrayList 采用的是数组形式来保存对象的，这种方式将对象放在连续的位置中，所以最大的缺点就是插入删除时非常麻烦

LinkedList 采用的将对象存放在独立的空间中，而且在每个空间中还保存下一个链接的索引 但是缺点就是查找非常麻烦 要从第一个索引开始

Hashtable和HashMap类有三个重要的不同之处。第一个不同主要是历史原因。Hashtable是基于陈旧的Dictionary类的，HashMap是Java 1.2引进的Map接口的一个实现。

也许最重要的不同是Hashtable的方法是同步的，而HashMap的方法不是。这就意味着，虽然你可以不用采取任何特殊的行为就可以在一个多线程的应用程序中用一个Hashtable，但你必须同样地为一个HashMap提供外同步。一个方便的方法就是利用Collections类的静态的synchronizedMap()方法，它创建一个线程安全的Map对象，并把它作为一个封装的对象来返回。这个对象的方法可以让你同步访问潜在的HashMap。这么做的结果就是当你不需要同步时，你不能切断Hashtable中的同步（比如在一个单线程的应用程序中），而且同步增加了很多处理费用。
第三点不同是，只有HashMap可以让你将空值作为一个表的条目的key或value。HashMap中只有一条记录可以是一个空的key，但任意数量的条目可以是空的value。这就是说，如果在表中没有发现搜索键，或者如果发现了搜索键，但它是一个空的值，那么get()将返回null。如果有必要，用containKey()方法来区别这两种情况。
一些资料建议，当需要同步时，用Hashtable，反之用HashMap。但是，因为在需要时，HashMap可以被同步，HashMap的功能比Hashtable的功能更多，而且它不是基于一个陈旧的类的，所以有人认为，在各种情况下，HashMap都优先于Hashtable。
关于Properties
有时侯，你可能想用一个hashtable来映射key的字符串到value的字符串。DOS、Windows和Unix中的环境字符串就有一些例子，如key的字符串PATH被映射到value的字符串C:\WINDOWS;C:\WINDOWS\SYSTEM。Hashtables是表示这些的一个简单的方法，但Java提供了另外一种方法。
Java.util.Properties类是Hashtable的一个子类，设计用于String keys和values。Properties对象的用法同Hashtable的用法相象，但是类增加了两个节省时间的方法，你应该知道。
Store()方法把一个Properties对象的内容以一种可读的形式保存到一个文件中。Load()方法正好相反，用来读取文件，并设定Properties对象来包含keys和values。
注意，因为Properties扩展了Hashtable，你可以用超类的put()方法来添加不是String对象的keys和values。这是不可取的。另外，如果你将store()用于一个不包含String对象的Properties对象，store()将失败。作为put()和get()的替代，你应该用setProperty()和getProperty()，它们用String参数。

5.throwable有哪些子类，你遇到过哪些运行时异常
error和exception。ClassCastException(类转换异常)
IndexOutOfBoundsException(数组越界)
NullPointerException(空指针)
ArrayStoreException(数据存储异常，操作数组时类型不一致)
还有IO操作的BufferOverflowException异常

6.创建线程的几种方法
继承Thread类或者创建类实现了Runnable接口
7.对i++多线程访问你会怎么做
比如一个类，里面有成员变量或者静态变量i，还有自增操作的方法， 在方法声明前使用synchronized，保证在访问i 的时候，直到方法结束前，没有别的线程方法能访问。
8.java会出现内存泄露吗，如果会，在哪种情况下

内存泄露是指无用对象（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，从而造成的内存空间的浪费称为内存泄露。内存泄露有时不严重且不易察觉，这样开发者就不知道存在内存泄露，但有时也会很严重，会提示你Out of memory。
那么，Java内存泄露根本原因是什么呢？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露，尽管短生命周期对象已经不再需要，但是因为长生命周期对象持有它的引用而导致不能被回收，这就是java中内存泄露的发生场景。具体主要有如下几大类：
1、静态集合类引起内存泄露：
像HashMap、Vector等的使用最容易出现内存泄露，这些静态变量的生命周期和应用程序一致，他们所引用的所有的对象Object也不能被释放，因为他们也将一直被Vector等引用着。
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。
3、监听器
在java 编程中，我们都需要和监听器打交道，通常一个应用当中会用到很多监听器，我们会调用一个控件的诸如addXXXListener()等方法来增加监听器，但往往在释放对象的时候却没有记住去删除这些监听器，从而增加了内存泄漏的机会。
4、各种连接
比如数据库连接（dataSourse.getConnection()），网络连接(socket)和io连接，除非其显式的调用了其close（）方法将其连接关闭，否则是不会自动被GC 回收的。对于Resultset 和Statement 对象可以不进行显式回收，但Connection 一定要显式回收，因为Connection 在任何时候都无法自动回收，而Connection一旦回收，Resultset 和Statement 对象就会立即为NULL。但是如果使用连接池，情况就不一样了，除了要显式地关闭连接，还必须显式地关闭Resultset Statement 对象（关闭其中一个，另外一个也会关闭），否则就会造成大量的Statement 对象无法释放，从而引起内存泄漏。这种情况下一般都会在try里面去的连接，在finally里面释放连接。
5、内部类和外部模块等的引用
内部类的引用是比较容易遗忘的一种，而且一旦没释放可能导致一系列的后继类对象没有释放。此外程序员还要小心外部模块不经意的引用，例如程序员A 负责A 模块，调用了B 模块的一个方法如：
public void registerMsg(Object b);
这种调用就要非常小心了，传入了一个对象，很可能模块B就保持了对该对象的引用，这时候就需要注意模块B 是否提供相应的操作去除引用。
6、单例模式
不正确使用单例模式是引起内存泄露的一个常见问题，单例对象在被初始化后将在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对象的引用，那么这个外部对象将不能被jvm正常回收，导致内存泄露，

9.抽象类和接口的区别
接口是公开的，里面不能有私有的方法或变量，是用于让别人使用的，而抽象类是可以有私有方法或私有变量的，
另外，实现接口的一定要实现接口里定义的所有方法，而实现抽象类可以有选择地重写需要用到的方法，一般的应用里，最顶级的是接口，然后是抽象类实现接口，最后才到具体类实现。

还有，接口可以实现多重继承，而一个类只能继承一个超类，但可以通过继承多个接口实现多重继承，接口还有标识（里面没有任何方法，如Remote接口）和数据共享（里面的变量全是常量）的作用.

10.Spring优缺点
优点

1.使用Spring的IOC容器，将对象之间的依赖关系交给Spring，降低组件之间的耦合性，让我们更专注于应用逻辑
2.可以提供众多服务，事务管理，WS等。
3.AOP的很好支持，方便面向切面编程。
4.对主流的框架提供了很好的集成支持
5.Spring DI机制降低了业务对象替换的复杂性。
6.Spring属于低侵入，代码污染极低。
7.Spring的高度可开放性，并不强制依赖于Spring，开发者可以自由选择Spring部分或全部
缺点
1.jsp中要写很多代码、控制器过于灵活，缺少一个公用控制器
2.Spring不支持分布式，这也是EJB仍然在用的原因之一。

11.索引为什么查询快，其数据结构
索引就是通过事先排好序，从而在查找时可以应用二分查找等高效率的算法。
数据结构是B树

12.什么时候用redis
适合放一些频繁使用的热点数据，放在内存读写快。

13.遍历删除list元素
1.通过增强的for循环删除符合条件的多个元素
2.通过增强的for循环删除符合条件的一个元素
3.通过普通的for删除删除符合条件的多个元素
4.通过Iterator进行遍历删除符合条件的多个元素

```
1.  Iterator<Student> stuIter = students.iterator();
2.      while (stuIter.hasNext()) {
3.          Student student = stuIter.next();
4.          if (student.getId() % 2 == 0)
5.              stuIter.remove();//这里要使用Iterator的remove方法移除当前对象，如果使用List的remove方法，则同样会出现ConcurrentModificationException
6.      }
```

14.死锁原因条件
产生死锁的原因主要是：
（1） 因为系统资源不足。
（2） 进程运行推进的顺序不合适。
（3） 资源分配不当等。
如果系统资源充足，进程的资源请求都能够得到满足，死锁出现的可能性就很低，否则
就会因争夺有限的资源而陷入死锁。其次，进程运行推进顺序与速度不同，也可能产生死锁。
产生死锁的四个必要条件：
（1） 互斥条件：一个资源每次只能被一个进程使用。
（2） 请求与保持条件：一个进程因请求资源而阻塞时，对已获得的资源保持不放。
（3） 不剥夺条件:进程已获得的资源，在末使用完之前，不能强行剥夺。

（4） 循环等待条件:若干进程之间形成一种头尾相接的循环等待资源关系。
这四个条件是死锁必要条件，只要系统发生死锁，这些条件必然成立，而只要上述条件之一不满足，就不会发生死锁。
死锁的解除与预防：
理解了死锁的原因，尤其是产生死锁的四个必要条件，就可以最大可能地避免、预防和解除死锁。所以，在系统设计、进程调度等方面注意如何不让这四个必要条件成立，如何确定资源的合理分配算法，避免进程永久占据系统资源。此外，也要防止进程在处于等待状态的情况下占用资源。因此，对资源的分配要给予合理的规划。