java基础

基础

- 多态：编译时类型和运行时类型,Person person（编译时类型）= new Student()（运行时类型）
- hashCode：对象存储的物理地址，equals与==（内存地址是否同一个）
- 基本对象Object方法
  - getClass()
  - toString()
  - equals()
  - hashCode()
  - wait()
  - notify()
  - notifyAll()
  - finalize(),GC清理对象之前所调用的清理方法
  - clone()
  - 对象锁
    - 所有对象都自动含有单一的锁，线程可以使用synchronized关键字来获取对象上的锁
    - JVM负责跟踪对象被加锁的次数。如果一个对象被解锁，其计数变为0。在任务（线程）第一次给对象加锁的时候，计数变为1。每当这个相同的任务（线程）在此对象上获得锁时，计数会递增。
  - 类锁
    - 对于同步静态方法/静态变量互斥体，由于一个class不论被实例化多少次，其中的静态方法和静态变量在内存中都只由一份。所以，一旦一个静态的方法被申明为synchronized。此类所有的实例化对象在调用此方法，共用同一把锁，我们称之为类锁。一旦一个静态变量被作为synchronized block的互斥体。进入此同步区域时，都要先获得此静态变量的对象锁。
- 继承、代理、组合,组合优于继承
- 异常
  - 运行时异常：NullPointerException，ClassCastException，IndexOutOfBoundsException，ArithmeticException
  - 编译时异常：ClassNotFoundException，IllegalArgumentException，NoSuchMethodException
- Dynamic Proxy
  - Dynamic Proxy是这样一种class：它是在运行时生成的class，在生成它时你必须提供一组interface给它，然后该class就宣称它实现了这些 interface。

集合

- set(hashset)无序且不重复，保证不重复？
  - HashSet中add方法调用的是底层HashMap中的put()方法
  - 1、当集合要添加新的元素时，先调用这个元素的hashCode方法
  - 2、如果这个位置上没有元素，它就可以直接存储在这个位置上，不用再进行任何比较了；
  - 3、如果这个位置上已经有元素了， 就调用它的equals方法与新元素进行比较，相同的话就不存了，不相同就散列其它的地址。
  - 4、所以这里存在一个冲突解决的问题。这样一来实际调用equals方法的次数就大大降低了，几乎只需要一两次。
  - 所以，Java对于eqauls方法和hashCode方法是这样规定的：
  - 1)、如果两个对象相同，那么它们的hashCode值一定要相同；
  - 2)、如果两个对象的hashCode相同，它们并不一定相同 。上面说的对象相同指的是用eqauls方法比较。
- list线性存储，可重复
  - ArrayList：基于数组实现的，是一个动态数组，自动扩容机制1.5倍，Arrays.copyOf(elementData, newCapacity)
  - 容量动态增长： 当数组容量不够用时，创建一个比原数组容量大的新数组，将数组中的元素“搬”到新数组，再将新的元素也放入新数组，最后将新数组赋给原数组即可。
  - LinkedList:链表
  - vector
- Queue保持一个队列(先进先出)的顺序
- hashmap
  - put:往hashmap里面存放key-value对的时候，都会为它们实例化一个Entry对象，这个Entry对象就会存储在前面提到的Entry数组table中
  - 1、对key做null检查。如果key是null，会被存储到table[0]，因为null的hash值总是0。
  - 2、key的hashcode()方法会被调用，然后计算hash值。hash值用来找到存储Entry对象的数组的索引。有时候hash函数可能写的很不好，所以JDK的设计者添加了另一个叫做hash()的方法，它接收刚才计算的hash值作为参数。如果你想了解更多关于hash()函数的东西，可以参考：hashmap中的hash和indexFor方法
  - 3、indexFor(hash,table.length)用来计算在table数组中存储Entry对象的精确的索引。
  - 4、在我们的例子中已经看到，如果两个key有相同的hash值(也叫冲突)，他们会以链表的形式来存储。所以，这里我们就迭代链表。
  - 1）如果在刚才计算出来的索引位置没有元素，直接把Entry对象放在那个索引上。
  - 2）如果索引上有元素，然后会进行迭代，一直到Entry->next是null。当前的Entry对象变成链表的下一个节点。
  - 3）如果我们再次放入同样的key会怎样呢？逻辑上，它应该替换老的value。事实上，它确实是这么做的。在迭代的过程中，会调用equals()方法来检查key的相等性(key.equals(k))，如果这个方法返回true，它就会用当前Entry的value来替换之前的value。
  - concurrenthashmap:该类将 Map 的存储空间分为若干块，每块拥有自己的锁，大大减少了多个线程争夺同一个锁的情况

多线程

- 共享内存：Java内存模型规定所有的变量都是存在主存当中（类似于前面说的物理内存），每个线程都有自己的工作内存（类似于前面的高速缓存）
- wait、sleep区别
  - Thread.sleep不会导致锁行为的改变，如果当前线程是拥有锁的，那么Thread.sleep不会让线程释放锁。如果能够帮助你记忆的话，可以简单认为和锁相关的方法都定义在Object类中，因此调用Thread.sleep是不会影响锁的相关行为。
  - Thread.sleep和Object.wait都会暂停当前的线程，对于CPU资源来说，不管是哪种方式暂停的线程，都表示它暂时不再需要CPU的执行时间。OS会将执行时间分配给其它线程。区别是，调用wait后，需要别的线程执行notify/notifyAll才能够重新获得CPU执行时间。
  - wait/notify和sleep：wait放在同步代码块中的,当线程执行wait()时，会把当前的锁释放，然后让出CPU，进入等待状态。等待时wait会释放锁，而sleep一直持有锁。Wait(notify)通常被用于线程间交互，sleep通常被用于暂停执行。
  - wait()方法与notify()必须要与synchronized(resource)一起使用
- 线程的状态
  - NEW
  - RUNNABLE
  - Thread.State.BLOCKED（阻塞）表示线程正在获取锁时，因为锁不能获取到而被迫暂停执行下面的指令，一直等到这个锁被别的线程释放。BLOCKED状态下线程，OS调度机制需要决定下一个能够获取锁的线程是哪个，这种情况下，就是产生锁的争用，无论如何这都是很耗时的操作。
  - WAITING
    某一等待线程的线程状态。
    线程因为调用了Object.wait()或Thread.join()而未运行，就会进入WAITING状态。
  - TIMED_WAITING
    具有指定等待时间的某一等待线程的线程状态。
    线程因为调用了Thread.sleep()，或者加上超时值来调用Object.wait()或Thread.join()而未运行，则会进入TIMED_WAITING状态。
  - TERMINATED
- 原子性(AtomicInteger)、可见性、有序性
- 锁类型
  - 自旋锁
  - 可重入锁
  - 同步
  - 互斥
- volatile不保证原子性
  - 一旦一个共享变量（类的成员变量、类的静态成员变量）被volatile修饰之后，那么就具备了两层语义：
  - 1）保证了不同线程对这个变量进行操作时的可见性，即一个线程修改了某个变量的值，这新值对其他线程来说是立即可见的。
  - 2）禁止进行指令重排序。
  - 解决的是内存可见性的问题，会使得所有对volatile变量的读写都会直接刷到主存，即保证了变量的可见性
- ThreadLocal
  - 每个线程都有一个ThreadLocal就是每个线程都拥有了自己独立的一个变量，竞争条件被彻底消除了
- lock
  - public class Lock{ private boolean isLocked = false; public synchronized void lock() throws InterruptedException{ while(isLocked){ //不用if，而用while，是为了防止假唤醒 wait(); } isLocked = true; } public synchronized void unlock(){ isLocked = false; notify(); }}
- 线程池、ExecutorService
- future模式、callable

线上排查能力

- 命令
  - top：查看系统内存和CPU
  - top -H pid：抓取占用cpu高的线程
  - jstack: 查看某个java进程内的线程堆栈
  - jmap:查看内存使用情况
    - jmap -dump:format=b,file=文件名 [pid],整个JVM 中内存信息
  - thread dump
  - heap dump
  - MAT
  - CPU飙高排查
  - 内存飙高排查
- linux基础命令
  - sudo su admin
  - 一个文件夹及其子文件夹赋权:chmod -R 777 xxx
  - rm -rf /home/admin/iworkpublic/target/iworkpublic.war，-r递归删除，-f强制删除
  - ls/ls -l
  - find -name "*.war"（全局查找文件）
  - tail -f jmonitor.log |grep "isStart"
  - cat jmonitor.log |grep "isStart"(文件内查找，concatenate的简称)
  - tar zxf /home/admin/iworkpublic/target/iworkpublic.tgz -C /home/admin/iworkpublic/target/
  - -c创建压缩文件,-x解开压缩文件,-f目标文件名,-C指定解压到的目录
  - 复制：scp root@192.168.120.204:/opt/soft/nginx-0.5.38.tar.gz /opt/soft/

设计模式

- 代理模式：通过代理对象访问目标对象.这样做的好处是:可以在目标对象实现的基础上,增强额外的功能操作,即扩展目标对象的功能.
  - 静态代理:代理对象需要与目标对象实现一样的接口
  - 动态代理（JDK代理,接口代理）：代理对象不需要实现接口,但是目标对象一定要实现接口,否则不能用动态代理
  - Cglib代理：适用目标对象只是一个单独的对象,并没有实现任何的接口的情况
- 观察者模式：
  从根本上说，该模式必须包含两个角色：观察者和被观察对象。观察者和被观察者之间存在“观察”的逻辑关联.当被观察者发生改变的时候，观察者就会观察到这样的变化，并且做出相应的响应。
  1、观察者
  （Observer）将自己注册到被观察对象（Subject）中，被观察对象将观察者存放在一个容器（list）里。
  2、被观察对象
  被观察对象发生了某种变化，从容器中得到所有注册过的观察者，将变化通知观察者。
  3、撤销观察
  观察者告诉被观察者要撤销观察，被观察者从容器中将观察者去除。
- 命令模式：某个方法需要完成某个功能，完成这个功能的大部分步骤已经确定，但可能有少量具体步骤无法确定，必须等到执行该方法时才能确定
- 适配器模式：想使用一个已经存在的类，但是该类不符合接口需求（Adaptee被适配者，Target目标接口，Adapter适配器）

JVM

- 类加载
  - 类的加载器是将.class文件,加载到内存（JVM）中，并生成Java.lang.class对象。通过ClassLoader加载类，只需要一个名字，你就可以创建一个类的实例，所以，真正要创建的类名，可以做成一个配置项，独立于程序的编译和发布。如热部署等。
  - 在JVM中，一个实例是通过本身的类名+加载它的ClassLoader识别的，也就是说不同的ClassLoader加载同一个类在JVM是不同的。
  - BootStrapClassLoader,ExtClassLoader,AppClassLoader(所有自定义的父加载器)
  - URLClassLoader和ClassLoader区别
    ClassLoader只能加载classpath下面的class，而URLClassLoader可以加载任意路径下的class。
  - Class对象生成
    - Class.forName("类名字符串全称,包名+类名")
    - 类名.class
    - 实例对象.getClass()
    - this.getClass().getClassLoader().loadClass()显式加载class文件到内存
  - 自定义classLoader
    - 重写ClassLoader类的findClass方法
    - defineClass这个方法很简单就是将class文件的字节数组变成一个class对象，这个方法肯定不能重写，内部实现是在C/C++代码中实现的
  - ClassNotFoundException问题:this.getClass().getClassLoader().getResource("").toString查找当前classpath目录下有无指定当前的class文件
  - 获取资源：getResourceAsStream
- 内存组件
  - java堆：存储java对象的地方，是被所有java线程所共享的。所有通过new创建的对象的内存都在堆中分配，其大小可以通过-Xmx和-Xms来控制。堆被划分为新生代和旧生代，新生代又被进一步划分为Eden和Survivor区
  - 方法区（所有类信息，常量池，静态字段，方法），又叫静态区，永久区，跟堆一样，被所有的线程共享，如：private static boolean flag;
  - java栈：总是和线程关联在一起，JVM运行实际程序的实体是线程，每个线程创建时JVM都会为它创建一个堆栈,主要存放一些基础类型变量数据和对象句柄（引用），注意:int（栈中）和Integer(堆中)
  - 类和类加载器：同样需要存储空间，这个区叫永久代PermGen区(见方法区)
  - NIO：NIO direct memory
  - 类中方法的局部变量包括原生数据类型和对象引用都是静态分配内存的。静态内存空间挡这段代码运行结束时回收。对象的内存空间是动态分配的，对象什么时候被回收也是不确定的，只有等到这个对象不再使用时才被回收
- 基于分代的垃圾收集算法
  - 把对象按照寿命长短分组，新创建的对象分在年轻代，如果对象经过几次回收后仍然存活，再把这个对象分到年老代，年老代的收集频度不像年轻代那么频繁，这样就减少了每次垃圾收集所要扫描的对象数量，提高了效率
  - 新生代：新创建的对象被分在年轻代。新建的对象都是用新生代分配内存，Eden空间不足的时候，会把存活的对象转移到Survivor中，新生代大小可以由-Xmn来控制，也可以用-XX:SurvivorRatio来控制Eden和Survivor的比例。旧生代用于存放新生代中经过多次垃圾回收 (也即Minor GC) 仍然存活的对象
  - 年老代：如果对象经过几次回收之后任然存活，再把这个对象分到年老代，如果年老代也满了，将会触发fullGC回收整个堆内存
  - perm区:主要存放类的Class对象，如果一个类被频繁的加载，也会导致perm区满，perm区的垃圾回收也是由fullGC触发的
- 当Eden区域分配内存时，发现空间不足，JVM就会触发Minor GC（新生代 GC），程序中System.gc()也可以来触发。
- 啥时候会触发Full GC？
  - 旧生代空间不足
  - Perm空间满
- GC：对于GC来说，当程序员创建对象时，GC就开始监控这个对象的地址、大小以及使用情况。通常，GC采用有向图的方式记录和管理堆(heap)中的所有对象。通过这种方式确定哪些对象是"可达的"，哪些对象是"不可达的"。当GC确定一些对象为"不可达"时，GC就有责任回收这些内存空间。
- 虚拟机给每个对象定义了一个对象年龄（Age）计数器。如果对象在 Eden 出生并经过第一次 Minor GC 后仍然存活，并且能被 Survivor 容纳的话，将被移动到 Survivor 空间中，并将对象年龄设为 1。对象在 Survivor 区中每熬过一次 Minor GC，年龄就增加 1 岁，当它的年龄增加到一定程度（默认为 15 岁）时，就会被晋升到老年代中。对象晋升老年代的年龄阈值，可以通过参数 -XX:MaxTenuringThreshold 来设置。

算法

- 冒泡排序
- 快速排序
- 二分法查找
  - 采用二分法查找时，数据需是有序不重复的
  - 题目：将排序数组按绝对值大小排序

IO

- 交互方式
  - 同步
    - 一个任务的完成需要依赖另一个任务时，只有等待被依赖的任务完成，依赖的任务才能完成
  - 阻塞
    - CPU停下来等待一个慢的操作完成以后，CPU菜接着完成其他的事
  - 同步非阻塞
- IO性能提升
  - 同步非阻塞，但是会增加CPU消耗
  - 异步阻塞，分布式数据库常用到，通常有一份是同步阻塞的记录，而还有两个备份记录写到其他机器上，这些备份记录通常采用异步阻塞的方式写IO
- 网络ＩＯ优化
  - 减少网络交互次数
  - 减少网络数据量大小
  - 尽量减少编码
    - 通常在网络IO中数据传输都是以字节形式进行，也就是通常要序列化。但是我们发送的要传输的数据都是字符形式的，从字符到字节必须编码，但是这个编码过程比较耗时，所以在经过网络IO传输时，尽量以字节形式发送。
- 磁盘IO优化
  - 增加缓存，减少磁盘访问次数
- 字节的操作接口：InputStream,OutputStream,最小的存储单位都是字节，而不是字符，字符到字节必须经过编码转换
- InputStreamReader:字节到字符的转化桥梁，byte[]转String,是编码，String转byte[]，是解码