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，==是判断两个变量或实例是不是指向同一个内存空间
  - equals是判断两个变量或实例所指向的内存空间的值是不是相同
- 继承、代理、组合,组合优于继承
- StringBuffer (性能差) 和StringBuilder(非线程安全)
- 基本对象Object方法
  - 对象锁
    - getClass()、toString()、equals()、hashCode()、wait()、notify()、notifyAll()、clone()
    - 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/静态变量互斥锋，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。
  - **每个对象都有一把锁和等待队列，一个线程在进入sync代码块时，会尝试获取锁，如果获取不到则会把此线程加入等待队列中去。**
    除了锁的等待队列，还有一种是等待队列，该队列用于线程间的协作。调用wait就会把当前线程放到条件队列上并阻塞，表示当前线程执行不下去了，它需要另外一个条件，让个条件已自己改变不了，需要其他线程改变。当其他线程条件改变后，应该调用Object类的notify方法，notify做的事情就是从条件队列中选一个线程，将其从队列中移除并唤醒
    - 锁语意总结
      - 线程A获取一个锁，实质上是线程A向该下来将要获取这个锁的某个线程发出了（线程A对共享变量所做修改的）消息。
      - 线程B获取一个锁，实质上是线程B接收了之前某个线程发出的（在释放这个锁之前对共享变量所做修改的）消息。
      - 线程A释放锁，随后线程B获取这个锁，这个过程实质上是线程A通过主内存向线程B发送消息。
    - 当线程获取锁时，JMM会把该线程对应的本地内存置为无效。从而使得被监视器保护的临界区代码必须要从主内存中去读取共享变量。
- 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。
  - Proxy类与普通类的唯一区别就是其字节码是由 JVM 在运行时动态生成的而非预存在于任何一个 class 文件中，每次生成动态代理类对象时都需要指定一个类装载器对象：newProxyInstance()方法第一个参数
- java8
  - 接口函数式编程
    - Function：接受一个参数，返回一个参数。
    - Consumer：接受一个参数，不返回参数。
    - Predicate：用于测试是否符合条件，返回boolean类型。
    - Supplier：不接受任何参数，然返回参数
  - lambda ->
  - Stream流
    - Filter 过滤:过滤通过一个predicate接口来过滤并只保留符合条件的元素
    - Sort 排序:返回的是排序好后的Stream。如果你不指定一个自定义的Comparator则会使用默认排序
    - Map 映射:元素按照指定的Function接口来依次将元素转成另外的对象
    - Match:允许检测指定的Predicate是否匹配整个Stream。所有的匹配操作都是最终操作，并返回一个boolean类型的值
    - Count 计数:计数是一个最终操作，返回Stream中元素的个数，返回值类型是long
    - Reduce 规约:允许通过指定的函数来讲stream中的多个元素规约为一个元素，规越后的结果是通过Optional接口表示的
    - 并行Streams:stream()改为parallelStream()
    - Map类型不支持stream.putIfAbsent
  - Optional 接口
- CompletableFuture和future的区别
  - Callable接口类似于Runnable，从名字就可以看出来了，但是Runnable不会返回结果，并且无法抛出返回结果的异常，而Callable功能要强大一些，被线程执行后，可以返回值，这个返回值可以被Future拿到，也就是说，Future可以拿到异步执行任务的返回值
  - 一个任务提交给了Future，Future替我完成这个任务。期间我自己可以去做任何想做的事情。一段时间之后，我就便可以从Future那儿取出结果。
  - Future缺点:Future虽然可以实现获取异步执行结果的需求，但是它没有提供通知的机制，我们无法得知Future什么时候完成。
  - 要么使用阻塞，在future.get()的地方等待future返回的结果，这时又变成同步操作。要么使用isDone()轮询地判断Future是否完成，这样会耗费CPU的资源。
  - CompletableFuture能够将回调放到与任务不同的线程中执行，也能将回调作为继续执行的同步调用，在与任务相同的线程中执行。它避免了传统回调最大的问题，那就是能够将控制流分离到不同的事件处理器中。
  - CompletableFuture弥补了Future模式的缺点。在异步的任务完成后，需要用其结果继续操作时，无需等待。可以直接通过thenAccept、thenApply、thenCompose等方式将回调异步处理的结果交给另外一个异步事件处理线程来处理
- 集合/数据结构
  - 锁表、栈、队列、排序、数组、树
  - set(hashset)
    - 无序且不重复，保证不重复？
  - hashmap
    - 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，不相同就散列其他的地址。
      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的方法是这样规定的：
      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。
    - 数组(Entry)数组,hash值用来计算key在Entry数组的索引)、链表(如果两个元素有相同的hashcode，它们会被放在同一个索引上,以链表(LinkedList)的形式来存储的)
    - 通过hashcode找到数组中的某一元素entry,通过equlas在链表中找到key对应的value
  - 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的引用。
    - LinkedList:双向链表，前后节点
    - vector
  - Queue
    保持一个队列先进先出的顺序
    - 并发队列ConcurrentLinkedQueue
    - 阻塞队列LinkedBlockingQueue
      - take方法在队列空的时候阻塞等待新的可用资源到达
      - put方法在队列满(有界队列的时候)阻塞以等待空位
    - Queue不是线程安全的,BlockingQueue 实现是线程安全的
    - 如何让Queue可以固定长度？
  - 二叉树
    - 在链表中，插入、删除速度很快，但查找速度较慢。
      在数组中，查找速度很快，但插入删除速度很慢。
      为了解决这个问题，找寻一种能够在插入、删除、查找、遍历等操作都相对快的容器，于是人们发明了二叉树。
    - 二叉树是一种重要的数据结构，与数组、向量、链表都是一种顺序容器，它们提供了按位置访问数据的手段。但是有一个缺点，它们都是按照位置来确定数据，想要通过值来获取数据，只能通过遍历的方式。而二叉树在很大程度上解决了这个缺点，二叉树是按值来保存元素，也就值来访问元素。
    - 二叉树由一个个节点组成，一个节点最多只能有两个子节点，从根节点开始左右扩散，分左子节点和右子节点，向下一直分支。

JVM

- 类加载
  - 作用：负责将.class文件加载到内存中，并为之生成对应的Class对象-------------------反射前提
  - 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-Xms和-Xmx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
  - 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)
  - 面试？
    线程是new的实例对象是在堆里还是栈内呢？
    变量引用地址存在栈里，但是这个地址指向的对象的内容是在堆内存中，因为这个引用不会被回收的线程归属，所以是线程安全的
  - 方法区,永久代（所有类信息，常量池，静态字段，方法），又叫静态区，跟堆一样，被所有的线程共享，如：private static boolean flag。永久代是HotspotJVM的对外的概念，是方法区的一种实现
  - 类和类加载器：同样需要存储空间，这个区叫永久代PermGen区(元方法区)
  - 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。静态内存空间指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，新创建的对象会在年轻代，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来控制新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？
  - 年老代空间不足
  - 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。
- 性能优化
  - JVM调优
    - JVM内存大小设置
      - 每个线程栈大小
      - JVM最大值
      - 年轻代大小
      - 持久带大小
    - 垃圾回收器选择
      - 串行收集器
      - 并行收集器（吞吐量优先）
      - 并发收集器（响应时间优先）

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才能接着完成其他的事
  - 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，分布式数据库用到，通常有一台是同步阻塞的记录，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I O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，字节到字节必须经过编码转换
- 字符的操作接口：Reader,Write
- InputStreamReader:字节到字符的转化桥梁(需要做编码)，byte[]转String,是编码，String转byte[]，是解码
- netty
- mina

多线程

- 线程池
  - ReentrantReadWriteLock
  - lock
    - http://blog.csdn.net/ghsau/article/details/7461369/
    - 使用synchronized可以实现同步，但是缺点是同时只有一个线程可以访问共享变量，但是实际上，对于多个读操作操作共享变量时候是不需要同步的，synchronized时候无法实现多个读线程同时执行，而大部分情况下读操作次数多于写操作，所以这大大降低了并发性，所以出现了ReentrantReadWriteLock，它可以实现读写分离，运行多个线程同时进行读取，但是最多运行一个写现线程存在
  - 死锁
    - 有两个线程A，B，和两个对象a,b。现在A正在调用a.调用b,之后A想调用b，B正在使用b.调用完b，之后想调动a。
    - 于是A 和 两个线程分别抱着a,b的锁不放开，互相等对方放开锁，然后自己就可以执行下一步。于是就发生了死锁
  - ThreadLocal
    - 在ThreadLocal类中有一个Map，用于存储每一个线程的“变量副本”，Map中元素的键为线程对象，而值对应线程的变量副本。
    - ThreadLocal就是象变空间换时间，Synchronized用于线程间的数据同步，而ThreadLocal则用于线程间的数据隔离，隔离了多个线程对数据的数据共享
  - futureTask(实现了两个接口，Runnable和Future)，future(拿到结果)，callable(产生结果)，runnable
- 并发容器
  - 线程池ThreadPoolExecutor、ExecutorService,Spring的ThreadPoolTaskExecutor
  - 线程池至少应包含线程池管理器、工作线程、任务缓冲队、任务接口callable\runnable（任务）
  - 有界队列和无界队列？通过LinkedBlockingQueue的构造参数
    LinkedBlockingQueue构造的时候若没有指定大小，则默认大小为Integer.MAX_VALUE
    当然也可以在构造函数的参数中指定大小
  - 阻塞队列LinkedBlockingQueue与ArrayBlockingQueue区别
    - 内部都是使用ReentrantLock和Condition来保证生产和消费的同步；
      当队列为空，消费者线程被阻塞；当队列满，生产者线程被阻塞；
    - 锁机制不同：ArrayBlockingQueue生产者和消费者使用的是同一把锁
  - concurrenthashmap:该类将 Map 的存储空间分为若干块(segment)，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
  - Concurrent,尽量保证读不加锁，并且修改的时候不影响读
  - CopyOnWrite,通俗的理解是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。http://ifeve.com/java-copy-on-write/
- 算法
  - 排序
    - 快速排序算法(最常用)
  - 查找

基本思想是：通过一趟排序将要排序的数据分割成独立的两部分，其中一部分的所有数据都比另外一部分的所有数据都要小，然后再按此方法对这两部分数据分别进行快速排序，整个排序过程可以递归进行，以此达到整个数据变成有序序列。

线上排查能力

- 排查命令
  - 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 -dump:format=b,file=文件名 [pid],整个JVM 中内存信息
  - jmap:查看内存使用情况
  - thread dump
  - heap dump
  - netstat
    - CPU飙高排查
  - 线程ID匹配jstack的调用栈
  - 内存飙高排查
  1、内存溢出 out of memory，是指程序在申请内存时，没有足够的内存空间供其使用，出现out of memory；
  2、内存泄露 memory leak，是指程序在申请内存后，无法释放已申请的内存空间，一次内存泄露危害可以忽略，但内存泄露堆积后果很严重，无论多少内存,迟早会被占光。
  3、memory leak会最终会导致out of memory！
- 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
  - ps -ef|grep admin
  - sh appctl.sh pubstart
  - rm -rf /home/admin/workpublic/target/workpublic.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)
  - grep "server" nginx-proxy.conf(查找某个文件是否包含此内容
  - less,作用跟more一样，查看某个文件时，你可以按一下“ /” 键，然后输入一个word回车，这样就可以查找这个word了。如果是多个这样word可以按“ n” 键显示下一个
  - tar zxf /home/admin/workpublic/target/workpublic.tgz -C /home/admin/workpublic/target/
  - -c创建压缩文件, -x解开压缩文件, -t目标文件名, -C指定解压到的目录
  - 跳板机输入：scp root@172.16.71.62:/home/admin/logs/app_log.2018-05-11.log ~，然后再用ftp连接跳板机，使用get命令下载到本地
  - grep是查找匹配条件的行，find是搜索匹配条件的文件
  - grep "server" /*./* 表示当前目录下所有文件
  - 查看目录下的文件和文件夹大小：进入目录，du -sh *
  - yum install -y 类库
    - 基於RPM包管理，能够从指定的服务器自动下载RPM包并且安装，可以自动处理依赖性关系，并且一次安装所有依赖的软体包
    - rpm和yum的区别
  - rpm只能安装已经下载到本地机器上的rpm包, yum能在线下载并安装rpm包,能更新系统,且还能自动处理包与包之间的依赖问题,这个是rpm工具所不具备的
  - tomcat启动：./bin/startup.sh,killall nginx
  - netstat -tpln
  - nohup npm start > nohup.out 2>&1 & exit 后台启动
  - rm -rf directory

设计模式

- 代理模式
  - 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,不能对final进行代理
  - spring默认使用jdk动态代理，但jdk动态代理是针对接口做代理的。如果类不是实现的接口的时候，就会使用cglib代理
  - 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
- 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、删除观察
  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 ）
- 单一职责
- 开闭原则
  - 软件实体应该是可以扩展的，但是不可修改的
- 依赖倒置原则

基础工具

- 代理:charles,客户端的请求代理（A请求改为请求B ）
- 反向代理：ngrok
- 浏览器代理：Switcheroo Redirector