java基础

基础

- 多态：编译时类型和运行时类型
- 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
- 继承、代理、组合
- Exception,checked,unchecked

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)
  - 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。

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。

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。

- 线程的状态
  - NEW
  - RUNNABLE
  - BLOCKED
  - 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
- 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
- 线程池、ExecutorService
- future模式、callable

JVM

- 类加载
  - ClassLoader主要对类的请求提供服务，当JVM需要某类时，它根据名称向ClassLoader要求这个类，然后由ClassLoader返回这个类的class对象。
  - 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
  - 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
  - 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)
  - 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
  - NIO：NIO direct memory
- 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
  - young区：新创建的对象被分在年轻代
  - old区：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old区也满了，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

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

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
  - thread dump
  - heap dump
- 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)
  - 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

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）