**Override 和 Overload的区别**：

|  |
| --- |
| Override:跟面向对象有关，是多态的具体体现，注意被覆盖的函数名称，参数类型，返回类型都相同，简单的说就是子类重写了父类的方法。  Overload: 跟面向对象无关，是指编写一个与已有函数同名但是参数列表不一样的函数，其是一种语法规则 |

**进程和线程的区别**：

|  |
| --- |
| 1: 不同的操作系统资源管理方式  进程是系统进行资源分配和调度的一个独立单位，线程是进程的一个实体,是CPU调度和分派的基本单位  2: 内存占用的不同：  进程有独立的地址空间，而线程只是一个进程中的不同执行路径。线程有自己的堆栈和局部变量，但线程之间没有单独的地址空间。  3:效率问题：  进程在执行过程中拥有独立的内存单元，而多个线程共享内存，从而极大地提高了程序的运行效率  4:健壮型问题：  多进程的程序要比多线程的程序健壮，因为单独的内存单元，不容易引起并发性的操作  5:总之：  线程和进程在使用上各有优缺点：线程执行开销小，但不利于资源的管理和保护；而进程正相反。同时，线程适合于在SMP机器上运行，而进程则可以跨机器迁移。  6:形象的例子：（工厂和车间的例子） |

**实现一个栈(stack)**:用数组

|  |
| --- |
| 1: 底层用数组来实现，构造函数包括有参和无参的构造函数  2: 类的成员变量包括，Object[] arr, int top, int maxSize; => 注意数组的类型是object,表示什么类型都能装  3: 实现pop,push的操作都是基于top成员变量的设计，  pop() => top-1  push()=> top+1  但是要注意边界条件的检测，初始化的时候，top为-1. |

**实现一个队列(queue)**:用数组

|  |
| --- |
| 思想与上面类似，不过需要两个index来标记：  front 和 rear,代表队列的第一个元素和最后一个元素，当然操作也复杂一点，  例如超过maxSize的情况下，我们考虑用mod来处理：  this.rear = (this.rear+1) % this.maxSize; => 入队列  this.front = (this.front+1) % this.maxSize; =>出队列 |

**线程的五种状态和描述**:

|  |
| --- |
| **新建(New)：**new一个Thread对象或者其子类对象就是创建一个线程，当一个线程对象被创建，但是没有开启，这个时候，只是对象线程对象开辟了内存空间和初始化数据。  **就绪(Runnable)：**新建的对象调用start方法，就开启了线程，线程就到了就绪状态。       在这个状态的线程对象，具有执行资格，没有执行权。  **运行(Running)：**当线程对象获取到了CPU的资源。在这个状态的线程对象，既有执行资格，也有执行权。  **冻结(Blocked)：**运行过程中的线程由于某些原因(比如wait,sleep)，释放了执行资格和执行权。当然，他们可以回到运行状态。只不过，不是直接回到。 而是先回到就绪状态。  **死亡(Dead)：**当线程对象调用的run方法结束，或者直接调用stop方法，就让线程对象死亡，在内存中变成了垃圾。 |

**线程start()和run()方法的区别**:

|  |
| --- |
| 调用start方法方可启动线程，而run方法只是thread的一个普通方法，调用run方法不能实现多线程；具体的说：  **Start()方法:**  start方法用来启动线程,实现了多线程运行,这时无需等待run方法体代码执行完毕而直接继续执行下面的  代码。通过调用Thread类的start()方法来启动一个线程,这时此线程处于就绪(可运行)状态，并没有运行，  一旦得到cpu时间片(执行权),就开始执行run()方法,这里方法run()称为线程体，  它包含了要执行的这个线程的内容，Run方法运行结束,此线程随即终止。  **Run()方法:**  run()方法只是Thread类的一个普通方法,如果直接调用Run方法,程序中依然只有主线程这一个线程,  其程序执行路径还是只有一条，还是要等待run方法体执行完毕后才可继续执行下面的代码，  这样就没有达到多线程的目的 |

**线程sleep()和wait()方法的区别**:

|  |
| --- |
| (1)这两个方法来自不同的类，sleep()来自Thread类，和wait()来自Object类。  (2)sleep是Thread的静态类方法，谁调用的谁去睡觉，即使在a线程里调用了b的sleep方法，实际上还是a去睡觉， 要让b线程睡觉要在b的代码中调用sleep。而wait()是Object类的非静态方法  (3)sleep()释放资源不释放锁，而wait()释放资源释放锁；  (4)使用范围：wait,notify和notifyAll只能在同步控制方法或者同步控制块里面使用,而sleep可以在任何地方使用 |

**单例模式下的饿汉和懒汉式子以及懒汉式子的多线程表示**:

|  |
| --- |
| 1:  饿汉：  class Single  {  private Single(){}//将构造函数私有化，不让别的类建立该类对象  private static final Single s=new Single();//自己建立一个对象  public static Single getInstance()//提供一个公共访问方式  {  return s;  }  }  懒汉线程不安全：(延时加载)  class Single  {  private Single(){}  private static Single s;  public static Single getInstance()  {  if(s==null)  s=new Single();  return s;  }  }  上述如果多个线程同时操作懒汉式时就有可能出现线程安全问题，解决线程安全问题可以加同步来解决。但是加了同步之后，每一次都要比较锁，效率就变慢了，  所以可以加**双重判断**来提高程序效率。  public static Single getInstance()  {  **if(s==null)//当不是null时候直接return**  {  synchronized(Single.class)  {  **if(s==null) //锁内部再判断一次，确保只有一个实例**  s=new Single();  }  }  return s;  }  2:采用静态加载的方法：  public class FooSingleton3 {      public final static FooSingleton3 INSTANCE = new FooSingleton3();      private FooSingleton3() { }      public static FooSingleton3 getInstance() { return INSTANCE; }      public void bar() { }  }  采静态代码只加载一次，因此实现了单例模式  3: 使用Enum 类型：  public enum FooEnumSingleton {      INSTANCE;      public static FooEnumSingleton getInstance() { return INSTANCE; }      public void bar() { }  }  枚举的缺点是它无法从另一个基类继承，因为它已经继承自java.lang.Enum。  枚举的一个优点是，如果你之后希望有“二例（dualton）”或“三例（tringleton）”，只需要增加新的枚举实例即可。例如，有了一个单例的缓存之后，你也许还想给缓存引入多个层**次**。  注意问题：序列化，延时加载问题，是否可以多继承问题，线程安全问题。 |

**线程中为什么wait()、notify()、notifyAll()**:

|  |
| --- |
| 为什么这些方法要定义在Object类中：  (1)这些方法只存在于同步中；  (2)使用这些方法时必须要指定所属的锁，即被哪个锁调用这些方法；  (3)而锁可以是任意对象，所以任意对象调用的方法就定义在Object中。  线程间的通信：  为什么要通信：多线程并发执行的时候， 如果需要指定线程等待或者唤醒指定线程， 那么就需要通信.比如生产者消费者的问题，  怎样通信：在同步代码块中, 使用锁对象的wait()方法可以让当前线程等待, 直到有其他线程唤醒为止.  使用锁对象的notify()方法可以唤醒一个等待的线程，或者notifyAll唤醒所有等待的线程.  多线程间通信用sleep很难实现，睡眠时间很难把握。 |

**线程中如果在静态方法使用同步什么后果：**

|  |
| --- |
| 同步静态方法时会获取该类的“Class”对象，所以当一个线程进入同步的静态方法中时，线程监视器获取类本身的对象锁，其它线程不能进入这个类的任何静态同步方法。它不像实例方法，因为多个线程可以同时访问不同实例同步实例方法。 |

**java和python中传递值还是传递引用的问题**:

|  |
| --- |
| Python中的传递问题：  那么Python中参数传递是传值，还是传引用呢？准确的回答：都不是。之所以不是传值，因为没有产生复制，而且函数拥有与调用者同样的对象。而似乎更像是C++的传引用？但是有时却不能改变实参的值。只能这样说：对于不可变的对象，它看起来像C++中的传值方式；对于可变对象，它看起来像C++中的按引用传递。  Java中是传递值：链接：**https://www.zhihu.com/question/31203609**  准确的说，对于基本数据类型或者是不可变的对象，对任何参数对象的操作都会产生新的对象（类似于python的immutable对象），对于其他引用对象，传递的是引用地址的复制，细细品味下面两个例子的区别：  **1：**提供了改变自身方法的引用类型 StringBuilder sb = new StringBuilder("iphone");  void foo(StringBuilder builder)  { builder.append("4"); }  foo(sb); // sb 被改变了，变成了"iphone4"。  **2：**提供了改变自身方法的引用类型，但是不使用，而是使用赋值运算符。 StringBuilder sb = new StringBuilder("iphone");  void foo(StringBuilder builder)  { builder = new StringBuilder("ipad"); }  foo(sb); // sb 没有被改变，还是 "iphone"  **另外java中的”=”的理解：赋值的操作** 1:= 是赋值操作 任何包含=的如+=、-=、 /=等等，都内含了赋值操作）。不再是你以前理解的数学含义了，而+ - \* /和 = 在java中更不是一个级别，换句话说， = 是一个动作，一个可以改变内存状态的操作，一个可以改变变量的符号，而+ - \* /却不会。  2: 对于基本数据类型变量，= 操作是完整地复制了变量的值。  3: 对于非基本数据类型变量，= 操作是复制了变量的引用。   1. 4: 参数本身是变量，参数传递本质就是一种 = 操作。参数是变量，所有我们对变量的操作、变量能有的行为，参数都有。所以把C语言里参数是传值啊、传指针啊的那套理论全忘掉，参数传递就是 = 操作。 |

**Bloom Filter 处理大数据的检查是否存在于某集合中**:

|  |
| --- |
| 好处：  我们使用bloom过滤器能够使我们的搜索时间维持在o(1)的水平，而不用去考虑文件的规模，另外它的空间复杂度也维持在一个可观的水平。  局限性：  **在判断一个元素是否属于某个集合时，有可能会把不属于这个集合的元素误认为属于这个集合（false positive）。但不会把属于这个集合的元素误认为不属于这个集合（False Negative）**.  试用场景：  Bloom Filter不适合那些“零错误”的应用场合。而在能容忍低错误率的应用场合下，Bloom Filter通过极少的错误换取了存储空间的极大节省。  推导：  假设 Hash 函数以等概率条件选择并设置 Bit Array 中的某一位，**m** 是该位数组的大小**，k** 是 Hash 函数的个数，那么位数组中某一特定的位在进行元素插入时的 Hash 操作中没有被置位的概率是：  http://pic002.cnblogs.com/images/2012/274814/2012071316492245.png  那么在所有 k 次 Hash 操作后该位都没有被置 "1" 的概率是：  http://pic002.cnblogs.com/images/2012/274814/2012071316510534.png  如果我们插入了 n 个元素，因而该位为 "1"的概率是：  http://pic002.cnblogs.com/images/2012/274814/2012071316520528.png  现在检测某一元素是否在该集合中。标明某个元素是否在集合中所需的 k 个位置都按照如上的方法设置为 "1"，但是该方法可能会使算法错误的认为某一原本不在集合中的元素却被检测为在该集合中（False Positives），该概率由以下公式确定：  http://pic002.cnblogs.com/images/2012/274814/2012071317030828.png  上述结果是在**假定由每个 Hash 计算出需要设置的位（bit） 的位置是相互独立为前提计算出来的**，不难看出，随着 m （位数组大小）的增加，假正例（False Positives）的概率会下降，同时随着插入元素个数 n 的增加，False Positives的概率又会上升，对于给定的m，n，如何选择Hash函数个数 k 由以下公式确定：  http://pic002.cnblogs.com/images/2012/274814/2012071317175487.png  此时False Positives的概率为：  http://pic002.cnblogs.com/images/2012/274814/2012071317184158.png  而对于给定的False Positives概率 p，如何选择最优的位数组大小 m 呢，  http://pic002.cnblogs.com/images/2012/274814/2012071317223869.png  上式表明，位数组的大小最好与插入元素的个数成线性关系，对于给定的 m，n，k，假正例概率最大为：  http://pic002.cnblogs.com/images/2012/274814/2012071317253418.png |

**亚线性算法判断某数组是否全是0**:

|  |
| --- |
| 假定数组大小是n,如果我们依次扫描数组中每个元素，时间是O(n)  但是对于亚线性的算法，我们通过在原数组的随机抽样-s, 然后根据已知数据的某些信息，得到这些抽样数据和原始数据的关系，今儿得到误差，典型的**用误差来换时间和空间**：  假设数据满足e远离：即数量为1的元素个数>n\*e,  那么当我们随机抽取s=2/e 个元素，误差是：  p(抽样中没有1)<= (1-e)s e-es=e-2<1/3  证据引理：如果一次测试以大于等于p的概率获得一个证据，那么s=2/p轮测试得到的证据概率大于等于2/3. |

总结：常见的复杂度转换的问题：

1: 空间换时间： 动态规划，bitmap(位排序)，bucket 排序…

2: 误差换时间或者空间：bloom filter, 亚线性时间和空间算法。

**GET $ POST**:

|  |
| --- |
| Information sent from a form with the GET method is **visible to everyone** (all variable names and values are displayed in the URL). GET also has limits on the amount of information to send. The limitation is about 2000 characters. However, because the variables are displayed in the URL, it is possible to bookmark the page. This can be useful in some cases.  GET may be used for sending non-sensitive data.  **Note:** GET should NEVER be used for sending passwords or other sensitive information!  Information sent from a form with the POST method is **invisible to others** (all names/values are embedded within the body of the HTTP request) and has **no limits** on the amount of information to send.  Moreover POST supports advanced functionality such as support for multi-part binary input while uploading files to server.  However, because the variables are not displayed in the URL, it is not possible to bookmark the page. |

**怎样用位运算来计算绝对值**:

|  |
| --- |
| 1:正常判断，首先判断是整数还是负数，再决定是否翻转每位  int i=a>>31;  system.out.println(i==0?a:~a+1);  2使用异或: 任何数与0异或都是本身，与-1(OXFFFFFFFF)异或都是取反。  int a = -11111;  int j = a>>31;  System.out.println((a^j)-j)  举几个例子实践一下！ |