目录

[1. https://www.jianshu.com/p/c70989bd5f29?from=message&isappinstalled=0 2](#_Toc47627506)

[java基础面试知识点 2](#_Toc47627507)

[2. java中==和equals和hashCode的区别 2](#_Toc47627508)

[3. int、char、long各占多少字节数 3](#_Toc47627509)

[4. int与integer的区别 3](#_Toc47627510)

[5. 谈谈对java多态的理解 5](#_Toc47627511)

[6. String、StringBuffer、StringBuilder区别 6](#_Toc47627512)

[7. 什么是内部类？内部类的作用 7](#_Toc47627513)

[8. jvm内存模型 8](#_Toc47627514)

[9. 抽象类和接口区别 9](#_Toc47627515)

[10. 抽象类的意义 10](#_Toc47627516)

[11. 在使用中抽象类和接口我们该如何选择？ 10](#_Toc47627517)

[12. 抽象类是否可以没有方法和属性？ 10](#_Toc47627518)

[13. 泛型中extends和super的区别 10](#_Toc47627519)

[14. 父类的静态方法能否被子类重写 11](#_Toc47627520)

[15. 进程和线程的区别 12](#_Toc47627521)

[16. final，finally，finalize的区别 13](#_Toc47627522)

[17. 序列化的方式 15](#_Toc47627523)

[18. Serializable 和Parcelable 的区别 16](#_Toc47627524)

[19. 静态属性和静态方法是否可以被继承？是否可以被重写？以及原因？ 17](#_Toc47627525)

[20. 静态内部类的设计意图 18](#_Toc47627526)

[21. 闭包和局部内部类的区别 19](#_Toc47627527)

[22. string 转换成 integer的方式及原理 19](#_Toc47627528)

[java深入源码级 19](#_Toc47627529)

[23. 哪些情况下的对象会被垃圾回收机制处理掉？ 19](#_Toc47627530)

[24. 讲一下常见编码方式？ 20](#_Toc47627531)

[25. utf-8编码中的中文占几个字节；int型几个字节？ 20](#_Toc47627532)

[26. 静态代理和动态代理的区别，什么场景使用？ 20](#_Toc47627533)

[27. Java的异常体系 21](#_Toc47627534)

[28. 谈谈你对解析与分派的认识 23](#_Toc47627535)

[29. 修改对象A的equals方法的签名，那么使用HashMap存放这个对象实例的时候，会调用哪个equals方法？ 24](#_Toc47627536)

[30. Java中实现多态的机制是什么？ 26](#_Toc47627537)

[31. 如何将一个Java对象序列化到文件里？ 27](#_Toc47627538)

[32. 说说你对Java反射的理解 28](#_Toc47627539)

[33. 说说你对Java注解的理解 28](#_Toc47627540)

[34. 说说你对依赖注入的理解 28](#_Toc47627541)

[35. 说一下泛型原理，并举例说明 29](#_Toc47627542)

[36. Java中String的了解 29](#_Toc47627543)

[37. String为什么要设计成不可变的？ 29](#_Toc47627544)

[38. Object类的equal和hashCode方法重写，为什么？ 30](#_Toc47627545)

[数据结构 30](#_Toc47627546)

[39. 常用数据结构简介 30](#_Toc47627547)

[40. 并发集合了解哪些？ 30](#_Toc47627548)

[41. 列举java的集合以及集合之间的继承关系 32](#_Toc47627549)

[42. List,Set,Map的区别 32](#_Toc47627550)

[43. List和Set和Map的实现方式以及存储方式 33](#_Toc47627551)

[44. HashMap的实现原理 34](#_Toc47627552)

[45. HashMap如何put数据（从HashMap源码角度讲解）？ 34](#_Toc47627553)

[46. HashMap怎么手写实现？ 34](#_Toc47627554)

[47. ConcurrentHashMap的实现原理 34](#_Toc47627555)

[48. ArrayMap和HashMap的对比 35](#_Toc47627556)

### <https://www.jianshu.com/p/c70989bd5f29?from=message&isappinstalled=0>

## java基础面试知识点

### java中==和equals和hashCode的区别

“==”：

　　==是运算符，用来比较两个值、两个对象的内存地址是否相等；

“equals()”：

　　equals是Object类的方法，默认情况下比较两个对象是否是同一个对象，内部实现是通过“==”来实现的。

如果想比较两个对象的其他内容，则可以通过重写equals方法，

例如：String类就重写了equals方法，改成了对象的内容是否相等。

具体实现可以参考：<https://www.cnblogs.com/hujinshui/p/10038044.html>

“hashCode()”：

　　hashCode也是Object类里面的方法，返回值是一个对象的哈希码，同一个对象哈希码一定相等，但不同对象哈希码也有可能相等。

如果两个对象通过equals方法比较相等，那么他的hashCode一定相等；

如果两个对象通过equals方法比较不相等，那么他的hashCode有可能相等；

例如以下程序：

class Untitled {

public static void main(String[] args) {

System.out.println("AaAa".hashCode());//2031744

System.out.println("BBBB".hashCode());//2031744

}

}

### int、char、long各占多少字节数

Java基本类型占用的字节数：  
1字节： byte , boolean  
2字节： short , char  
4字节： int , float  
8字节： long , double

编码与中文：  
Unicode/GBK： 中文2字节  
UTF-8： 中文通常3字节，在拓展B区之后的是4字节  
综上，中文字符在编码中占用的字节数一般是2-4个字节。

C语言中short通常是2字节，具体可能和平台编译器相关。

### int与integer的区别

一、区别：

1.Integer是int的包装类，int则是java的一种基本的数据类型；

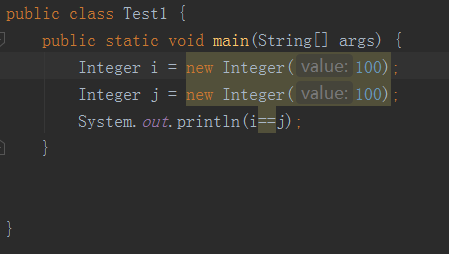
2.Integer变量必须实例化之后才能使用，而int变量不需要实例化；

3.Integer实际是对象的引用，当new一个Integer时，实际上生成一个指针指向对象，而int则直接存储数值

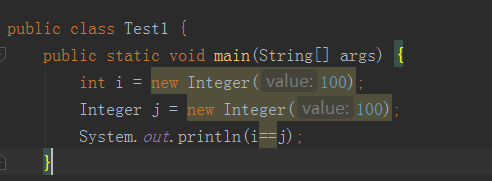
4.Integer的默认值是null，而int的默认值是0。

二、Integer和int的比较

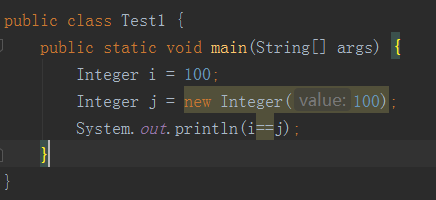
　　1.由于Integer实际是对一个Integer对象的引用，所以两个通过new生成的Integer变量永远是不相同的，因为New生成的是两个不同的对象，其内存地址不同。下面运行的结果为false



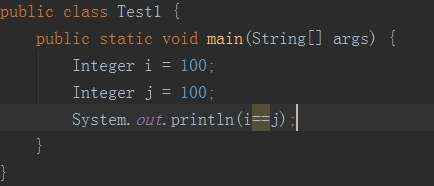
　　2.Integer变量和int变量进行比较时，只要两个变量的值相等，则结果就为True，(因为包装类Integer和基本数据类型比较的时候，java会自动拆箱为int，然后进行比较，实际上就是两个int变量进行比较)，下面运行的结果为true



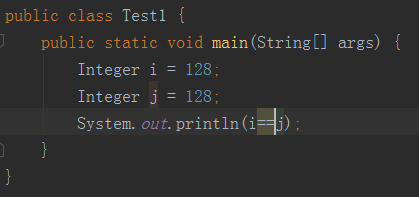
　　3.非new生成的Integer变量和new Integer生成的Integer变量比较的时候，结果为false(因为非new生成的Integer变量指向的是Java常量池中的对象，而new出来的对象指向的是堆中新建的对象，两者内存地址不同)，下面返回的是false



　　4.两个非new出来的Integer对象，进行比较的时候，如果两个变量的值区间在-127~128之间的时候，则返回的结果为true，如果两个变量的变量值不在这个区间，则比较的结果为false。下面返回的是true



　　下面返回的是false



三、java 基本类型与引用类型的区别：

　　1.基本数据类型保存原始值，引用数据类型保存的是引用值(引用值就是指在对象中所处的地理位置)

### 谈谈对java多态的理解

<https://www.cnblogs.com/eyesCentre/archive/2019/04/12/10697487.html>

1.什么是java的多态

浏览了别人博客中的一些介绍多态的文章，发现大家的描述有点不一样，主要区别在于是否把方法的重写算做多态。一种我比较认同的说法如下：

多态分为两种

a. 编译时多态：方法的重载；

b. **运行时多态**：JAVA运行时系统根据调用该方法的实例的类型来决定选择调用哪个方法则被称为运行时多态。（我们平时说得多的事运行时多态，所以多态主要也是指运行时多态）；

上述描述认为重载也是多态的一种表现，不过多态主要指运行时多态。

2.运行时多态

**a. 面向对象的三大特性**：封装、继承、多态。从一定角度来看，封装和继承几乎都是为多态而准备的。这是我们最后一个概念，也是最重要的知识点。

**b. 多态的定义**：指允许不同类的对象对同一消息做出响应。即同一消息可以根据发送对象的不同而采用多种不同的行为方式。（发送消息就是函数调用）

**c. 实现多态的技术称为**：动态绑定（dynamic binding），是指在**执行期间**判断所引用对象的实际类型，根据其实际的类型调用其相应的方法。

**d. 多态的作用**：消除类型之间的耦合关系。

**e. 现实中，关于多态的例子不胜枚举**。比方说按下 F1 键这个动作，如果当前在 Flash 界面下弹出的就是 AS 3 的帮助文档；如果当前在 Word 下弹出的就是 Word 帮助；在 Windows 下弹出的就是 Windows 帮助和支持。同一个事件发生在不同的对象上会产生不同的结果。

下面是多态存在的三个必要条件，要求大家做梦时都能背出来！

多态存在的三个必要条件  
一、要有继承；  
二、要有重写；  
三、父类引用指向子类对象。

**多态的好处**：

1.可替换性（substitutability）。多态对已存在代码具有可替换性。例如，多态对圆Circle类工作，对其他任何圆形几何体，如圆环，也同样工作。  
2.可扩充性（extensibility）。多态对代码具有可扩充性。增加新的子类不影响已存在类的多态性、继承性，以及其他特性的运行和操作。实际上新加子类更容易获得多态功能。例如，在实现了圆锥、半圆锥以及半球体的多态基础上，很容易增添球体类的多态性。  
3.接口性（interface-ability）。多态是超类通过方法签名，向子类提供了一个共同接口，由子类来完善或者覆盖它而实现的。如图8.3 所示。图中超类Shape规定了两个实现多态的接口方法，computeArea()以及computeVolume()。子类，如Circle和Sphere为了实现多态，完善或者覆盖这两个接口方法。  
4.灵活性（flexibility）。它在应用中体现了灵活多样的操作，提高了使用效率。  
5.简化性（simplicity）。多态简化对应用软件的代码编写和修改过程，尤其在处理大量对象的运算和操作时，这个特点尤为突出和重要。

### String、StringBuffer、StringBuilder区别

**String** 类不可变，内部维护的char[] 数组长度不可变，为final修饰，String类也是final修饰，不存在扩容。字符串拼接，截取，都会生成一个新的对象。频繁操作字符串效率低下，因为每次都会生成新的对象。

**StringBuilder** 类内部维护可变长度char[] ， 初始化数组容量为16，存在扩容， 其append拼接字符串方法内部调用System的native方法，进行数组的拷贝，不会重新生成新的StringBuilder对象。

它是非线程安全的字符串操作类， 其每次调用 toString方法而重新生成的String对象，不会共享StringBuilder对象内部的char[]，会进行一次char[]的copy操作。

**StringBuffer** 类内部维护可变长度char[]， 基本上与StringBuilder一致，但其为线程安全的字符串操作类，大部分方法都采用了Synchronized关键字修改，以此来实现在多线程下的操作字符串的安全性。

其toString方法而重新生成的String对象，会共享StringBuffer对象中的toStringCache属性（char[]），但是每次的StringBuffer对象修改，都会置null该属性值。

### 什么是内部类？内部类的作用

**什么是内部类**：  
将一个类定义在另一个类里面或者一个方法里面，这样的类称为内部类。

**内部类的作用：**

1.成员内部类  
成员内部类可以无条件访问外部类的所有成员属性和成员方法（包括private成员和静态成员）。  
当成员内部类拥有和外部类同名的成员变量或者方法时，会发生隐藏现象，即默认情况下访问的是成员内部类的成员。

2.局部内部类  
局部内部类是定义在一个方法或者一个作用域里面的类，它和成员内部类的区别在于局部内部类的访问仅限于方法内或者该作用域内。

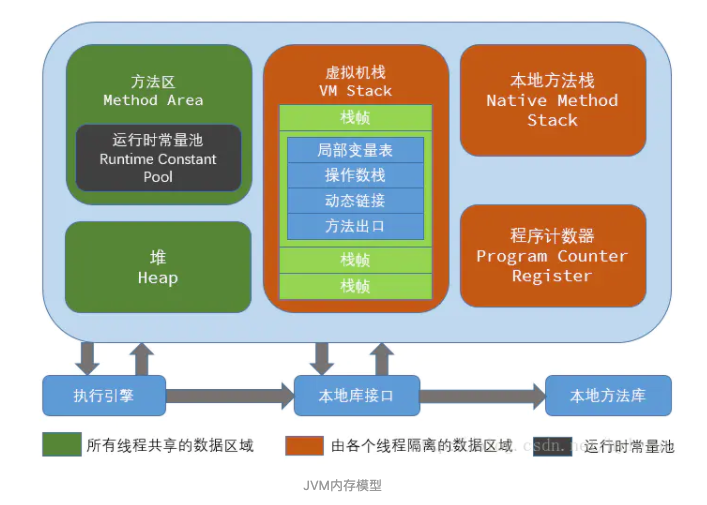
3.匿名内部类  
匿名内部类就是没有名字的内部类

4.静态内部类  
指被声明为static的内部类，他可以不依赖内部类而实例，而通常的内部类需要实例化外部类，从而实例化。静态内部类不可以有与外部类有相同的类名。不能访问外部类的普通成员变量，但是可以访问静态成员变量和静态方法（包括私有类型）  
一个 静态内部类去掉static 就是成员内部类，他可以自由的引用外部类的属性和方法，无论是静态还是非静态。但是不可以有静态属性和方法、

**作用**  
1.每个内部类都能独立的继承一个接口的实现，所以无论外部类是否已经继承了某个(接口的)实现，对于内部类都没有影响。内部类使得多继承的解决方案变得完整， 　　  
2.方便将存在一定逻辑关系的类组织在一起，又可以对外界隐藏。 　　  
3.方便编写事件驱动程序 　　  
4.方便编写线程代码

### jvm内存模型

<https://www.jianshu.com/p/0ecf020614cb>





### 抽象类和接口区别

一 什么时候使用抽象类和接口

* 如果你拥有一些方法并且想让它们中的一些有默认实现，那么使用抽象类吧。
* 如果你想实现多重继承，那么你必须使用接口。由于Java不支持多继承，子类不能够继承多个类，但可以实现多个接口。因此你就可以使用接口来解决它。
* 如果基本功能在不断改变，那么就需要使用抽象类。如果不断改变基本功能并且使用接口，那么就需要改变所有实现了该接口的类。

<https://www.cnblogs.com/songhuiqiang/p/10647835.html>



### 抽象类的意义

1. 因为抽象类不能实例化对象，所以必须要有子类来实现它之后才能使用。这样就可以把一些具有相同属性和方法的组件进行抽象，这样更有利于代码和程序的维护。
2. 当又有一个具有相似的组件产生时，只需要实现该抽象类就可以获得该抽象类的那些属性和方法。

所以，Java中抽象类对于代码的维护和重用有很好的帮助，也是Java面向对象的一个重要体现

### 在使用中抽象类和接口我们该如何选择？

\* 如果你拥有一些方法并且想让它们中的一些有默认实现，那么使用抽象类吧。

\* 如果你想实现多重继承，那么你必须使用接口。由于Java不支持多继承，子类不能够继承多个类，但可以实现多个接口。因此你就可以使用接口来解决它。

\* 如果基本功能在不断改变，那么就需要使用抽象类。如果不断改变基本功能并且使用接口，那么就需要改变所有实现了该接口的类。

### 抽象类是否可以没有方法和属性？

答案是肯定的，可以。

    抽象类中可以没有抽象方法，但有抽象方法的一定是抽象类。所以，java中 抽象类里面可以没有抽象方法。注意即使是没有抽象方法和属性的抽象类，也不能被实例化。

### 泛型中extends和super的区别

<https://blog.csdn.net/qq_36898043/article/details/79655309>

<? extends T>和<? super T>是Java泛型中的“通配符（Wildcards）”和“边界（Bounds）”的概念。

* <? extends T>：是指 “上界通配符（Upper Bounds Wildcards）”
* <? super T>：是指 “下界通配符（Lower Bounds Wildcards）”

PECS原则

最后看一下什么是PECS（Producer Extends Consumer Super）原则，已经很好理解了：

* 频繁往外读取内容的，适合用上界Extends。
* 经常往里插入的，适合用下界Super。

### 父类的静态方法能否被子类重写

<https://www.cnblogs.com/sunbr/p/11545010.html>

静态：

　　在编译时所分配的内存会一直存在（不会被回收），直到程序退出内存才会释放这个空间，在实例化之前这个方法就已经存在于内存，跟类的对象没什么关系。子类中如果定义了跟父类相同名称、相同的形参列表、相同返回值类型的静态方法，就满足重写的条件，于是在内存中又分配了一块给子类的静态方法，重写了父类方法。

父类代码

[制代码](javascript:void(0);)

1 public class Fu {

2 public static void show() {

3 System.out.println("父类的静态方法");

4 }

5 public void method() {

6 System.out.println("父类的一般方法");

7 }

8 }

[制代码](javascript:void(0);)

子类代码

[制代码](javascript:void(0);)

public class Zi extends Fu {

public static void main(String[] args) {

Fu fu = new Zi();

fu.show();

fu.method();

}

public static void show() {

System.out.println("子类的静态");

}

public void method() {

System.out.println("子类的一般方法");

}

}

[制代码](javascript:void(0);)

 输出结果是：

 父类的静态方法

 子类的一般方法

　　当父类引用指向子类对象，只会调用父类的静态方法，此行为并不具有多态性！所以子类重写父类的静态方法，并没有实际意义！

### 进程和线程的区别

<https://www.jianshu.com/p/2dc01727be45>

<https://www.cnblogs.com/zhehan54/p/6130030.html>

一、进程与线程的区别：

1、进程是资源分配的最小单位，线程是程序执行的最小单位（资源调度的最小单位）  
2、进程有自己的独立地址空间，每启动一个进程，系统就会为它分配地址空间，建立数据表来维护代码段、堆栈段和数据段，这种操作非常昂贵。  
而线程是共享进程中的数据的，使用相同的地址空间，因此CPU切换一个线程的花费远比进程要小很多，同时创建一个线程的开销也比进程要小很多。  
3、线程之间的通信更方便，同一进程下的线程共享全局变量、静态变量等数据，而进程之间的通信需要以通信的方式（IPC)进行。不过如何处理好同步与互斥是编写多线程程序的难点。  
4、但是多进程程序更健壮，多线程程序只要有一个线程死掉，整个进程也死掉了，而一个进程死掉并不会对另外一个进程造成影响，因为进程有自己独立的地址空间。  
参考：<https://www.cnblogs.com/zhehan54/p/6130030.html>

#### 二、进程与线程的资源：

1、堆与栈

堆：　是大家共有的空间，分全局堆和局部堆。全局堆就是所有没有分配的空间，局部堆就是用户分配的空间。堆在操作系统对进程初始化的时候分配，运行过程中也可以向系统要额外的堆，但是记得用完了要还给操作系统，要不然就是内存泄漏。  
栈：是个线程独有的，保存其运行状态和局部自动变量的。栈在线程开始的时候初始化，每个线程的栈互相独立，因此，栈是　thread safe的。操作系统在切换线程的时候会自动的切换栈，就是切换　ＳＳ／ＥＳＰ寄存器。栈空间不需要在高级语言里面显式的分配和释放。

2、其他

线程共享的环境包括：进程代码段、进程的公有数据(利用这些共享的数据，线程很容易的实现相互之间的通讯)、进程打开的文件描述符、信号的处理器、进程的当前目录和进程用户ID与进程组ID。  
进程拥有这许多共性的同时，还拥有自己的个性。有了这些个性，线程才能实现并发性。这些个性包括：  
1.线程ID  
每个线程都有自己的线程ID，这个ID在本进程中是唯一的。进程用此来标  
识线程。  
2.寄存器组的值  
由于线程间是并发运行的，每个线程有自己不同的运行线索，当从一个线  
程切换到另一个线程上时，必须将原有的线程的寄存器集合的状态保存，以便  
将来该线程在被重新切换到时能得以恢复。  
3.线程的堆栈  
堆栈是保证线程独立运行所必须的。  
线程函数可以调用函数，而被调用函数中又是可以层层嵌套的，所以线程  
必须拥有自己的函数堆栈，使得函数调用可以正常执行，不受其他线程的影  
响。  
4.错误返回码  
由于同一个进程中有很多个线程在同时运行，可能某个线程进行系统调用  
后设置了errno值，而在该线程还没有处理这个错误，另外一个线程就在此时  
被调度器投入运行，这样错误值就有可能被修改。  
所以，不同的线程应该拥有自己的错误返回码变量。  
5.线程的信号屏蔽码  
由于每个线程所感兴趣的信号不同，所以线程的信号屏蔽码应该由线程自  
己管理。但所有的线程都共享同样的信号处理器。  
6.线程的优先级  
由于线程需要像进程那样能够被调度，那么就必须要有可供调度使用的参  
数，这个参数就是线程的优先级。  
参考：<https://blog.csdn.net/shuilan0066/article/details/7683315>

3、进程与线程的同步

进程：无名管道、有名管道、信号、共享内存、消息队列、信号量  
进程：互斥量、读写锁、自旋锁、线程信号、条件变量

### final，finally，finalize的区别

<https://www.cnblogs.com/ktao/p/8586966.html>

1. final

　　在java中，final可以用来修饰类，方法和变量（成员变量或局部变量）。下面将对其详细介绍。

1.1 修饰类

　　当用final修饰类的时，表明该类不能被其他类所继承。当我们需要让一个类永远不被继承，此时就可以用final修饰，但要注意：

final类中所有的成员方法都会隐式的定义为final方法。

1.2 修饰方法

使用final方法的原因主要有两个：

　　(1) 把方法锁定，以防止继承类对其进行更改。

　　(2) 效率，在早期的java版本中，会将final方法转为内嵌调用。但若方法过于庞大，可能在性能上不会有多大提升。因此在最近版本中，不需要final方法进行这些优化了。

final方法意味着“最后的、最终的”含义，即此方法不能被重写。

**注意：若父类中final方法的访问权限为private，将导致子类中不能直接继承该方法，因此，此时可以在子类中定义相同方法名的函数，此时不会与重写final的矛盾，而是在子类中重新地定义了新方法。**

class A{

private final void getName(){

}

}

public class B extends A{

public void getName(){

}

public static void main(String[]args){

System.out.println("OK");

}

}

1.3 修饰变量

 　　final成员变量表示常量，只能被赋值一次，赋值后其值不再改变。类似于C++中的const。

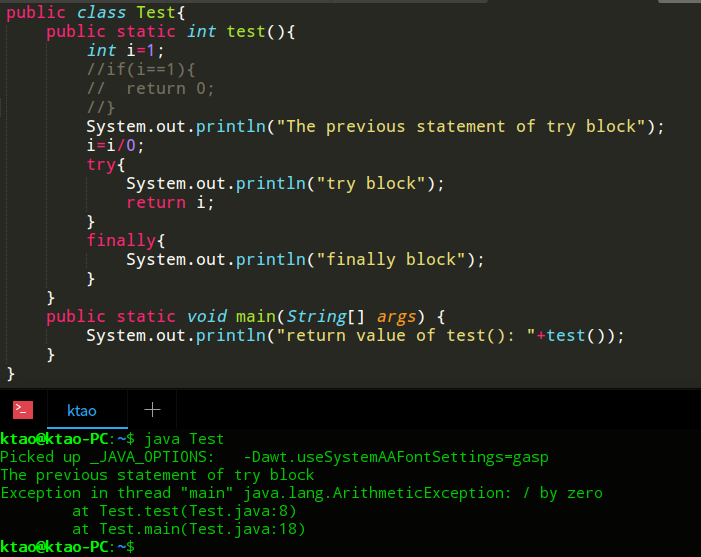
　　当final修饰一个基本数据类型时，表示该基本数据类型的值一旦在初始化后便不能发生变化；如果final修饰一个引用类型时，则在对其初始化之后便不能再让其指向其他对象了，但该引用所指向的对象的内容是可以发生变化的。本质上是一回事，因为引用的值是一个地址，final要求值，即地址的值不发生变化。

　　final修饰一个成员变量（属性），必须要显示初始化。这里有两种初始化方式，一种是在变量声明的时候初始化；第二种方法是在声明变量的时候不赋初值，但是要在这个变量所在的类的所有的构造函数中对这个变量赋初值。

当函数的参数类型声明为final时，说明该参数是只读型的。即你可以读取使用该参数，但是无法改变该参数的值。

2. finally

 　　finally作为异常处理的一部分，它只能用在try/catch语句中，并且附带一个语句块，表示这段语句最终一定会被执行（不管有没有抛出异常），经常被用在需要释放资源的情况下。（×）（这句话其实存在一定的问题）



3. finalize

　　finalize()是在java.lang.Object里定义的，也就是说每一个对象都有这么个方法。这个方法在gc启动，该对象被回收的时候被调用。其实gc可以回收大部分的对象（凡是new出来的对象，gc都能搞定，一般情况下我们又不会用new以外的方式去创建对象），所以一般是不需要程序员去实现finalize的。   
特殊情况下，需要程序员实现finalize，当对象被回收的时候释放一些资源，比如：一个socket链接，在对象初始化时创建，整个生命周期内有效，那么就需要实现finalize，关闭这个链接。   
　　使用finalize还需要注意一个事，调用super.finalize();

### 序列化的方式

<https://blog.csdn.net/ahuqihua/article/details/81331316>

**序列化和反序列化**  
序列化：可以将对象转化成一个字节序列，便于存储。  
反序列化：将序列化的字节序列还原  
优点：可以实现对象的"持久性”， 所谓持久性就是指对象的生命周期不取决于程序。  
  
序列化需要：  
所需类：ObjectInputStream和ObjectOutputStream  
方法： readObject()和writeObject();

**序列化方式一： 实现Serializable接口(隐式序列化)**通过实现Serializable接口，这种是隐式序列化(不需要手动)，这种是最简单的序列化方式，会自动序列化所有非static和 transient关键字修饰的成员变量。

**序列化方式二：实现Externalizable接口。(显式序列化)**  
Externalizable接口继承自Serializable, 我们在实现该接口时，必须实现writeExternal()和readExternal()方法，而且只能通过手动进行序列化，并且两个方法是自动调用的，因此，这个序列化过程是可控的，可以自己选择哪些部分序列化

**序列化方式三：实现Serializable接口+添加writeObject()和readObject()方法。(显+隐序列化)**  
如果想将方式一和方式二的优点都用到的话，可以采用方式三， 先实现Serializable接口，并且添加writeObject()和readObject()方法。注意这里是添加，不是重写或者覆盖。但是添加的这两个方法必须有相应的格式。

* 1，方法必须要被private修饰                                ----->才能被调用
* 2，第一行调用默认的defaultRead/WriteObject(); ----->隐式序列化非static和transient
* 3，调用read/writeObject()将获得的值赋给相应的值  --->显式序列化

### Serializable 和Parcelable 的区别

<https://www.cnblogs.com/berylqliu/p/6261479.html>

Parcelable和Serializable的区别：  
    android自定义对象可序列化有两个选择一个是Serializable和Parcelable  
    一、对象为什么需要序列化  
        1.永久性保存对象，保存对象的字节序列到本地文件。  
        2.通过序列化对象在网络中传递对象。  
        3.通过序列化对象在进程间传递对象。  
    二、当对象需要被序列化时如何选择所使用的接口  
        1.在使用内存的时候Parcelable比Serializable的性能高。  
        2.Serializable在序列化的时候会产生大量的临时变量，从而引起频繁的GC（内存回收）。  
        3.Parcelable不能使用在将对象存储在磁盘上这种情况，因为在外界的变化下Parcelable不能很好的保证数据的持续性。

### 静态属性和静态方法是否可以被继承？是否可以被重写？以及原因？

<https://blog.csdn.net/qdh186/article/details/79807057>

1)先说是否可以被继承

public class One {

//静态属性和静态方法是否可以被继承？

public static String one\_1 = "one";

public static void oneFn() {

System.out.println("oneFn");

}

}

public class Two extends One{

//空

}

public class MyTest {

//静态属性和静态方法是否可以被继承？是否可以被重写？以及原因？

public static void main(String[] args) {

One one = new Two();

one.oneFn();

String one\_1 = One.one\_1;

System.out.println("One.one\_1>>>>>>>"+one\_1);

String one\_12 = one.one\_1;

System.out.println("one.one\_1>>>>>>>"+one\_12);

}

}

//打印结果如下

oneFn

One.one\_1>>>>>>>one

one.one\_1>>>>>>>one

以上打印说明了：  
父类的静态属性和方法可以被子类继承

2）是否可以被重写？  
再来看这段代码

public class One {

//静态属性和静态方法是否可以被重写？以及原因？

public static String one\_1 = "one";

public static void oneFn() {

System.out.println("oneFn");

}

}

public class Two extends One {

public static String one\_1 = "two";

public static void oneFn() {

System.out.println("TwoFn");

}

}

public class MyTest {

//静态属性和静态方法是否可以被继承？是否可以被重写？以及原因？

public static void main(String[] args) {

One one = new Two();

one.oneFn();

String one\_1 = One.one\_1;

System.out.println("One.one\_1>>>>>>>"+one\_1);

String one\_12 = one.one\_1;

System.out.println("one.one\_1>>>>>>>"+one\_12);

}

}

//打印结果如下

//oneFn

//One.one\_1>>>>>>>one

//one.one\_1>>>>>>>one

以上打印说明  
当父类的引用指向子类时，使用对象调用静态方法或者静态变量，是调用的父类中的方法或者变量。并没有被子类改写。  
所以我认为不可以被子类重写。

3）原因  
static修饰函数/变量时，其实是全局函数/变量，它只是因为java强调对象的要挂，它与任何类都没有关系。靠这个类的好处就是这个类的成员函数调用static方法不用带类名。

注意：static关键字可以用修饰代码块.static代码块可以置于类中的任何一个位置，并可以有多个static代码块。在类初次被加载时，会按照静态代码块的顺序来执行，并且只会执行一次。

### 静态内部类的设计意图

在成员内部类中要注意两点：

1. 成员内部类中不能存在任何static的变量和方法；
2. 成员内部类是依附于外围类的，所以只有先创建了外围类才能够创建内部类。

**静态内部类与非静态内部类之间存在一个最大的区别：**

非静态内部类在编译完成之后会隐含地保存着一个引用，该引用是指向创建它的外围内，但是静态内部类却没有。

没有这个引用就意味着：

1. 它的创建是不需要依赖于外围类的。
2. 它不能使用任何外围类的非static成员变量和方法。

### 闭包和局部内部类的区别

比较常见的对闭包的定义是：引用了自由变量的一段代码或函数，被引用的自由变量和函数（一段代码）共同存在，即使离开了创造它的环境

### string 转换成 integer的方式及原理

<https://www.jianshu.com/p/9eebb4f2ccb1>

总结

1. parseInt(String s)--内部调用parseInt(s,10)（默认为10进制）
2. 正常判断null，进制范围，length等
3. 判断第一个字符是否是符号位
4. 循环遍历确定每个字符的十进制值
5. 通过\*= 和-= 进行计算拼接
6. 判断是否为负值 返回结果。

## java深入源码级

### 哪些情况下的对象会被垃圾回收机制处理掉？

<https://blog.csdn.net/codeyanbao/article/details/103512911>

1.所有实例都没有活动线程访问。  
2.没有被其他任何实例访问的循环引用实例。

3.Java 中有不同的引用类型。判断实例是否符合垃圾收集的条件都依赖于它的引用类型。

### 讲一下常见编码方式？

<https://blog.csdn.net/byf0521hlyp/article/details/80365045>

**ASCII编码：**用来表示英文，它使用1个字节表示，其中第一位规定为0，其他7位存储数据，一共可以表示128个字符。

**拓展ASCII编码：**用于表示更多的欧洲文字，用8个位存储数据，一共可以表示256个字符

**GBK/GB2312/GB18030：**表示汉字。GBK/GB2312表示简体中文，GB18030表示繁体中文。

**Unicode编码：**包含世界上所有的字符，是一个字符集。

**UTF-8：**是Unicode字符的实现方式之一，它使用1-4个字符表示一个符号，根据不同的符号而变化字节长度。

### utf-8编码中的中文占几个字节；int型几个字节？

utf-8的编码规则：

如果一个字节，最高位为0，表示这是一个ASCII字符（00~7F）  
如果一个字节，以11开头，连续的1的个数暗示这个字符的字节数

一个utf8数字占1个字节

一个utf8英文字母占1个字节

少数是汉字每个占用3个字节，多数占用4个字节。

### 静态代理和动态代理的区别，什么场景使用？

<https://blog.csdn.net/asd051377305/article/details/80490432>

**(1)什么是代理？**

大道理上讲代理是一种软件设计模式，目的地希望能做到代码重用。具体上讲，代理这种设计模式是通过不直接访问被代理对象的方式，而访问被代理对象的方法。这个就好比 商户---->明星经纪人(代理)---->明星这种模式。我们可以不通过直接与明星对话的情况下，而通过明星经纪人(代理)与其产生间接对话。

**(2)什么情况下使用代理？**

(1)设计模式中有一个设计原则是开闭原则，是说对修改关闭对扩展开放，我们在工作中有时会接手很多前人的代码，里面代码逻辑让人摸不着头脑，这时就很难去下手修改代码，那么这时我们就可以通过代理对类进行增强。

(2)我们在使用RPC框架的时候，框架本身并不能提前知道各个业务方要调用哪些接口的哪些方法 。那么这个时候，就可用通过动态代理的方式来建立一个中间人给客户端使用，也方便框架进行搭建逻辑，某种程度上也是客户端代码和框架松耦合的一种表现。

(3)Spring的AOP机制就是采用动态代理的机制来实现切面编程。

(3)静态代理和动态代理

我们根据加载被代理类的时机不同，将代理分为静态代理和动态代理。如果我们在代码编译时就确定了被代理的类是哪一个，那么就可以直接使用静态代理；如果不能确定，那么可以使用类的动态加载机制，在代码运行期间加载被代理的类这就是动态代理，比如RPC框架和Spring AOP机制。

(4)静态代理

我们先创建一个接口，遗憾的是java api代理机制求被代理类必须要实现某个接口，对于静态代理方式代理类也要实现和被代理类相同的接口；对于动态代理代理类则不需要显示的实现被代理类所实现的接口。

### Java的异常体系

<https://www.cnblogs.com/aspirant/p/10790803.html>

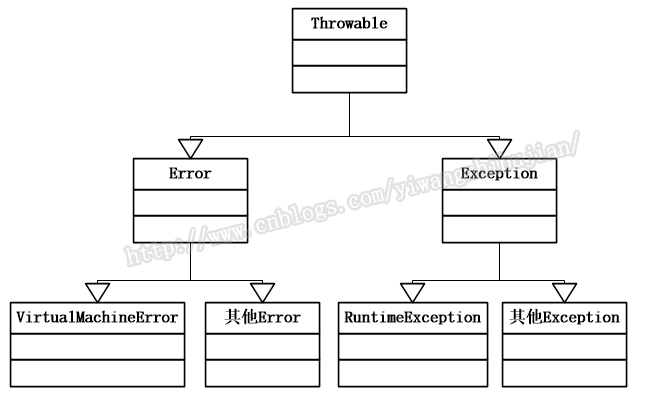
Java把异常作为一种类，当做对象来处理。所有异常类的基类是Throwable类，两大子类分别是Error和Exception。

　　系统错误由Java虚拟机抛出，用Error类表示。Error类描述的是内部系统错误，例如Java虚拟机崩溃。这种情况仅凭程序自身是无法处理的，在程序中也不会对Error异常进行捕捉和抛出。

　　异常（Exception）又分为RuntimeException(运行时异常)和CheckedException(检查时异常)，两者区别如下：

* RuntimeException：程序运行过程中才可能发生的异常。一般为代码的逻辑错误。例如：类型错误转换，数组下标访问越界，空指针异常、找不到指定类等等。
* CheckedException：编译期间可以检查到的异常，必须显式的进行处理（捕获或者抛出到上一层）。例如：IOException, FileNotFoundException等等。
* 先来看看java中异常的体系结构图解：

1、简单的：



2、复杂的：



throw、throws关键字  
    throw关键字是用于方法体内部，用来抛出一个Throwable类型的异常。如果抛出了检查异常，则还应该在方法头部声明方法可能抛出的异常类型。该方法的调用者也必须检查处理抛出的异常。如果所有方法都层层上抛获取的异常，最终JVM会进行处理，处理也很简单，就是打印异常消息和堆栈信息。throw关键字用法如下：

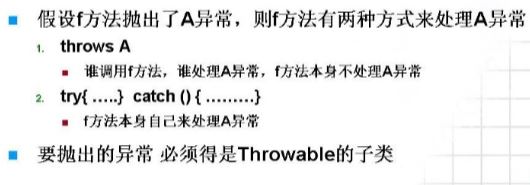
public static void test() throws Exception

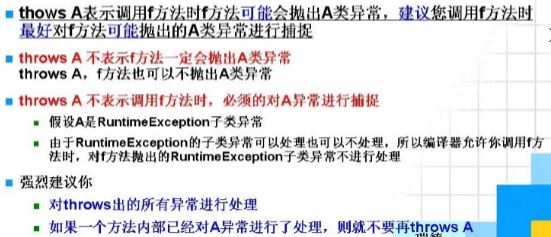
{

throw new Exception("方法test中的Exception");

}

   throws关键字用于方法体外部的方法声明部分，用来声明方法可能会抛出某些异常。仅当抛出了检查异常，该方法的调用者才必须处理或者重新抛出该异常。当方法的调用者无力处理该异常的时候，应该继续抛出.





### 谈谈你对解析与分派的认识

<https://www.jianshu.com/p/44790a602e11>

1）解析

Java中方法调用的目标方法在Class文件里面都是常量池中的符号引用，在类加载的解析阶段，会将其中的一部分符号引用转化为直接引用。这种解析的前提是：方法在程序真正运行之前就有一个可以确定的调用版本，并且这个方法的调用版本在运行期是不可改变的，即“编译期可知，运行期不可变”，这类目标的方法的调用称为解析（Resolve）。

只要能被invokestatic和invokespecial指令调用的方法，都可以在解析阶段中确定唯一的调用版本，符合条件的有静态方法（invokestatic指令）、私有方法、实例构造方法、父类方法（这3个是invokespecial指令），它们在类加载的的解析阶段就会将符号引用解析为该方法的直接引用。

2）分派

分派是多态性的体现，Java虚拟机底层提供了我们开发中“重载”(Overload)“和重写”(Override)的底层实现。其中重载属于静态分派，而重写则是动态分派的过程。  
解析调用一定是个静态的过程，在编译期就完全确定，在类加载的解析阶段就将涉及的符号引用全部转变为可以确定的直接引用，不会延迟到运行期再去完成。

**常量池**

常量池可以简单理解为class文件的资源从库，这种数据类型是Class文件结构中与其他项目关联最多的数据类型，也是占用Class文件空间最大的项目之一。在常量池中主要存放**字面量**和**符号引用**。字面量比较接近Java语言层面的常量概念，比如文本字符串、声明为final的常量值等（百度百科的解释是字面量是用双引用号引住的一系列字符）。符号引用则主要包括三类常量：

* 类和接口的全限定名
* 字段的名称和描述符
* 方法的名称和描述符。

**符号引用与直接引用的关联**

* 符号引用是一组符号，用来描述所引用的目标，符号是以任何形式存在的字面量。对于符号引用Java虚拟机并没有严格的限制。规定只需要使用的时候能够无歧义定位到目标就可以。常量池存在于Class文件中，而Class文件是必须首先通过Java虚拟机的类加载机制加载到内存中（确切的说是方法区这个内存区域，回顾一下，方法区存放的主要是对象的实例，这个Class文件是虚拟机对外接受访问的接口）。符号引用属于常量池中的内容，那么是不是说符号引用的目标已经加载到内存中了呢？答案是否定的，因为**符号引用与虚拟机的内存布局无关，符号引用的目标并不一定已经加载到内存中了。**
* 直接引用可以是直接指向引用目标的指针、相对偏移量或者是一个能够间接定位到目标的句柄。直接引用是和虚拟机的内存布局有关的，同一个符号引用在不同的虚拟机上翻译的直接引用一般是不同的。**如果有了直接引用，那么引用的目标必定是存在内存中的。**

### 修改对象A的equals方法的签名，那么使用HashMap存放这个对象实例的时候，会调用哪个equals方法？

会调用对象的equals方法。  
“==”如果是基本类型的话就是看他们的数据值是否相等就可以。  
如果是引用类型的话，比较的是栈内存局部变量表中指向堆内存中的指针的值是否相等  
“equals”如果对象的equals方法没有重写的话，equals方法和“==”是同一种。  
hashcode是返回对象实例内存地址的hash映射。  
理论上所有对象的hash映射都是不相同的。  
问题：  
1.你用过hashmap么？  
答：是的，然后回答HashMap的一些特性，譬如HashMap可以接受null键值和值，而Hashtable则不能；HashMap是非synchronized;HashMap很快；以及HashMap储存的是键值对等等。  
2.你知道hashmap的工作原理么？  
答：HashMap是基于hashing的原理，我们使用put(key, value)存储对象到HashMap中，使用get(key)从HashMap中获取对象。当我们给put()方法传递键和值时，我们先对键调用hashCode()方法，返回的hashCode用于找到bucket位置来储存Entry对象。”这里关键点在于指出，HashMap是在bucket中储存键对象和值对象，作为Map.Entry。这一点有助于理解获取对象的逻辑。如果你没有意识到这一点，或者错误的认为仅仅只在bucket中存储值的话，你将不会回答如何从HashMap中获取对象的逻辑。这个答案相当的正确，也显示出面试者确实知道hashing以及HashMap的工作原理。但是这仅仅是故事的开始，当面试官加入一些Java程序员每天要碰到的实际场景的时候，错误的答案频现。  
for (Map.Entry<Integer, Integer> entry : map.entrySet()) {  
System.out.println(entry.getKey() + ":" + entry.getValue());  
}  
3.当两个对象的hashcode相同会发生什么？  
因为即使hashcode相同，但是equals不同。  
因为hashcode相同，所以它们的bucket位置相同，‘碰撞’会发生。因为HashMap使用链表存储对象，这个Entry(包含有键值对的Map.Entry对象)会存储在链表中。  
4.如果两个键的hashcode相同，你如何获取值对象？  
当我们调用get()方法，HashMap会使用键对象的hashcode找到bucket位置，然后获取值对象。  
5.面试官提醒他如果有两个值对象储存在同一个bucket？  
将会遍历链表直到找到值对象  
6.因为你并没有值对象去比较，你是如何确定确定找到值对象的？  
找到bucket位置之后，会调用keys.equals()方法去找到链表中正确的节点，最终找到要找的值对象。

一些优秀的开发者会指出使用不可变的、声明作final的对象，并且采用合适的equals()和hashCode()方法的话，将会减少碰撞的发生，提高效率。不可变性使得能够缓存不同键的hashcode，这将提高整个获取对象的速度，使用String，Interger这样的wrapper类作为键是非常好的选择。

7.如果HashMap的大小超过了负载因子(load factor)定义的容量，怎么办？

默认的负载因子大小为0.75，也就是说，当一个map填满了75%的bucket时候，和其它集合类(如ArrayList等)一样，将会创建原来HashMap大小的两倍的bucket数组，来重新调整map的大小，并将原来的对象放入新的bucket数组中。这个过程叫作rehashing，因为它调用hash方法找到新的bucket位置。

8.使用ConcurrentHashMap在多线程下的操作

一个ConcurrentHashMap由多个segment组成，每一个segment都包含了一个HashEntry数组的hashtable， 每一个segment包含了对自己的HashEntry的操作，比如get，put，replace等操作，这些操作发生的时候，对自己的HashEntry进行锁定。由于每一个segment写操作只锁定自己的HashEntry，所以可能存在多个线程同时写的情况，性能无疑好于只有一个HashEntry锁定的情况。

2.Segment继承了ReentrantLock，所以它就是一种可重入锁（ReentrantLock)。在ConcurrentHashMap，一个Segment就是一个子哈希表，Segment里维护了一个HashEntry数组，并发环境下，对于不同Segment的数据进行操作是不用考虑锁竞争的。

3.在ConcurrentHashMap的remove，put操作还是比较简单的，都是将remove或者put操作交给key所对应的segment去做的，所以当几个操作不在同一个segment的时候就可以并发的进行。在这里面定义了一个  
volatile count，count = c的操作必须在modCount，table等操作的后面，这样才能保证这些变量操作的可见性。  
4.在锁定了这个segement后，get操作并没有使用lock，因为他得到了上面的count，所以count != 0之后，我们可以认为对应的hashtable是最新的，当然由于读取的时候没有加锁，在get的过程中，可能会有更新。当发现根据key去找元素的时候，但发现找得的key对应的value为null，这个时候可能会有其他线程正在对这个元素进行写操作，所以需要在使用锁的情况下在读取一下value，以确保最终的值。

### Java中实现多态的机制是什么？

<https://blog.csdn.net/github_37130188/article/details/89931885>

多态就是指一个引用变量倒底会指向哪个类的实例对象，该引用变量发出的方法调用到底是哪个类中实现的方法，必须在由程序运行期间才能决定。

因为在程序运行时才确定具体的类，这样，不用修改源程序代码，就可以让引用变量绑定到各种不同的类实现上，从而导致该引用调用的具体方法随之改变，即不修改程序代码就可以改变程序运行时所绑定的具体代码，让程序可以选择多个运行状态，这就是多态性。

**特点：**

1. 指向子类的父类引用由于向上转型了，它只能访问父类中拥有的方法和属性，而对于子类中存在而父类中不存在的方法，该引用是不能使用的，尽管是重载该方法。
2. 若子类重写了父类中的某些方法，在调用该些方法的时候，必定是使用子类中定义的这些方法（动态连接、动态调用）。

### 如何将一个Java对象序列化到文件里？

<https://www.jianshu.com/p/cb970e5cd5b8>

将对象序列化到文件

1）对象需要实现Seralizable接口  
public class StudentBean implements Serializable {  
······  
}

2）通过ObjectOutputStream的writeObject()方法写入  
和ObjectInputStream的readObject()方法来进行读取

//存进去  
try {  
ObjectOutputStream os = new ObjectOutputStream(  
new FileOutputStream("D:/student.txt"));  
os.writeObject(studentList);  
os.close();  
} catch (FileNotFoundException e) {  
// TODO Auto-generated catch block  
e.printStackTrace();  
} catch (IOException e) {  
// TODO Auto-generated catch block  
e.printStackTrace();  
}  
//读出来  
try {  
ObjectInputStream is = new ObjectInputStream(  
new FileInputStream("D:/student.txt"));  
ArrayList<StudentBean> list = new ArrayList<StudentBean>();  
list = (ArrayList<StudentBean>) is.readObject();  
for (int i = 0; i < list.size(); i++) {  
System.out.println(list.get(i).toString());  
}  
} catch (FileNotFoundException e) {  
// TODO Auto-generated catch block  
e.printStackTrace();  
} catch (IOException e) {  
// TODO Auto-generated catch block  
e.printStackTrace();  
} catch (ClassNotFoundException e) {  
// TODO Auto-generated catch block  
e.printStackTrace();  
}

### 说说你对Java反射的理解

<https://www.cnblogs.com/dashuaiguo/p/9896439.html>

**含义：**所谓的反射机制就是java语言在运行时拥有一项自观的能力。通过这种能力可以彻底的了解自身的情况为下一步的动作做准备。

**Java的反射机制的实现要借助于4个类：**Class，Constructor，Field，Method。

**Java反射的作用**：在Java运行时环境中，对于任意一个类，可以知道这个类有哪些属性和方法。对于任意一个对象，可以调用它的任意一个方法。这种动态获取类的信息以及动态调用对象的方法的功能来自于Java 语言的反射（Reflection）机制。

**Java 反射机制主要提供了以下功能:**

1. 在运行时判断任意一个对象所属的类。
2. 在运行时构造任意一个类的对象。
3. 在运行时判断任意一个类所具有的成员变量和方法。
4. 在运行时调用任意一个对象的方法

### 说说你对Java注解的理解

<https://blog.csdn.net/zhenliangit0918/article/details/81427994>

关于注解，我们知道，它是从JDK5及其以后JDK版本开始支持Java注解； Java注解（Annotation）也叫作元数据，以‘@注解名’在代码中存在，它是一种在源代码中标注的特殊标记，可以标注源代码中的类、属性、方法、参数等代码，主要用于创建文档，跟踪代码中的依赖性，甚至执行基本编译时检查。

Java注解按照是否含有参数分为三种：  
1、不带参数，语法结构为：@Annotation

2、带一个参数的注解，语法结构为：@Annotation(参数)

3、带多个参数的注解，语法结构为：@Annotiation(参数1, 参数2, 参数3...)

### 说说你对依赖注入的理解

<https://www.jianshu.com/p/827216fa9e02>

<https://www.cnblogs.com/narojay/p/10812614.html>

### 说一下泛型原理，并举例说明

泛型是Java 1.5的新特性，泛型的本质是参数化类型，也就是说所操作的数据类型被指定为一个参数。这种参数类型可以用在类、接口和方法的创建中，分别称为泛型类、泛型接口、泛型方法。

### Java中String的了解

<https://blog.csdn.net/qq_34490018/article/details/82110578>

### String为什么要设计成不可变的？

1. 字符串常量池的需要

字符串常量池(String pool,String intern pool, String保留池) 是Java堆内存中一个特殊的存储区域, 当创建一个String对象时,假如此字符串值已经存在于常量池中,则不会创建一个新的对象,而是引用已经存在的对象。

如下面的代码所示,将会在堆内存中只创建一个实际String对象.

String s1 = "abcd";

String s2 ="abcd";

示意图如下所示:

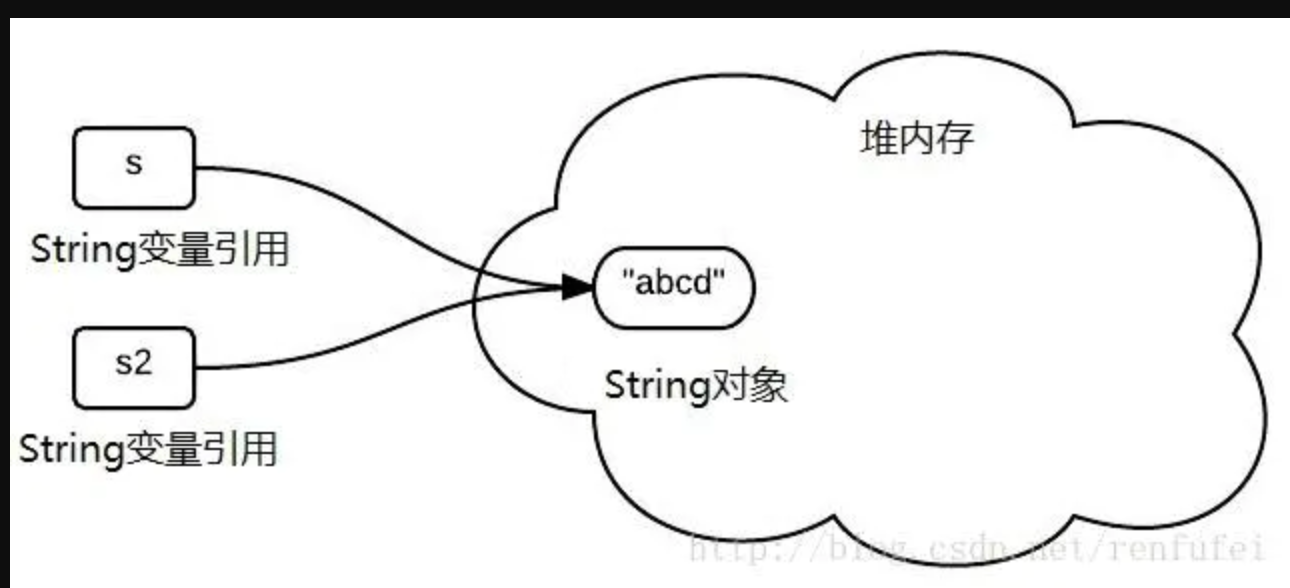


图1

假若字符串对象允许改变,那么将会导致各种逻辑错误,比如改变一个对象会影响到另一个独立对象. 严格来说，这种常量池的思想,是一种优化手段.

请思考: 假若代码如下所示，s1和s2还会指向同一个实际的String对象吗?

String s1= "ab" + "cd";

String s2="abc" + "d";

也许这个问题违反新手的直觉, 但是考虑到现代编译器会进行常规的优化, 所以他们都会指向常量池中的同一个对象. 或者,你可以用 jd-gui 之类的工具查看一下编译后的class文件.

2. 允许String对象缓存HashCode

Java中String对象的哈希码被频繁地使用, 比如在hashMap 等容器中。

字符串不变性保证了hash码的唯一性,因此可以放心地进行缓存.这也是一种性能优化手段,意味着不必每次都去计算新的哈希码. 在String类的定义中有如下代码:

private int hash;//用来缓存HashCode

3. 安全性

String被许多的Java类(库)用来当做参数,例如 网络连接地址URL,文件路径path,还有反射机制所需要的String参数等, 假若String不是固定不变的,将会引起各种安全隐患。

假如有如下的代码:

boolean connect(string s){

if (!isSecure(s)) {

throw new SecurityException();

}

// 如果在其他地方可以修改String,那么此处就会引起各种预料不到的问题/错误

    causeProblem(s);

}

总体来说, String不可变的原因包括 设计考虑,效率优化问题,以及安全性这三大方面. 事实上,这也是Java面试中的许多 "为什么" 的答案。

### Object类的equal和hashCode方法重写，为什么？

<https://blog.csdn.net/l707941510/article/details/87371081>

## 数据结构

### 常用数据结构简介

<https://www.cnblogs.com/zhaoziming-zhaogaojun/p/9559377.html>

### 并发集合了解哪些？

**并发List**

Vector和CopyOnWriteArrayList是两个线程安全的List，Vector读写操作都用了同步，相对来说更适用于写多读少的场合，CopyOnWriteArrayList在写的时候会复制一个副本，对副本写，写完用副本替换原值，读的时候不需要同步，适用于写少读多的场合。

**并发Set**

CopyOnWriteArraySet基于CopyOnWriteArrayList来实现的，只是在不允许存在重复的对象这个特性上遍历处理了一下。

**并发Map**

ConcurrentHashMap是专用于高并发的Map实现，内部实现进行了锁分离，get操作是无锁的。

**并发的Queue**

在并发队列上JDK提供了两套实现，一个是以ConcurrentLinkedQueue为代表的高性能队列，一个是以BlockingQueue接口为代表的阻塞队列。ConcurrentLinkedQueue适用于高并发场景下的队列，通过无锁的方式实现，通常ConcurrentLinkedQueue的性能要优于BlockingQueue。BlockingQueue的典型应用场景是生产者-消费者模式中，如果生产快于消费，生产队列装满时会阻塞，等待消费。

**并发的Dueue**

Queue是一种双端队列，它允许在队列的头部和尾部进行出队和入队的操作。Dueue实现类有非线程安全的LinkedList、ArrayDueue和线程安全的LinkedBlockingDueue。LinkedBlockingDueue没有进行读写锁的分离，因此同一时间只能有一个线程对其操作，因此在高并发应用中，它的性能要远远低于LinkedBlockingQueue，更低于ConcurrentLinkedQueue。

**并发锁重入锁ReentrantLock**

ReentrantLock是一种互斥锁的实现，就是一次最多只能一个线程拿到锁；

**读写锁ReadWriteLock**

读写锁有读取和写入两种锁，读取锁允许多个读取的线程同时持有，而写入锁只能有一个线程持有。

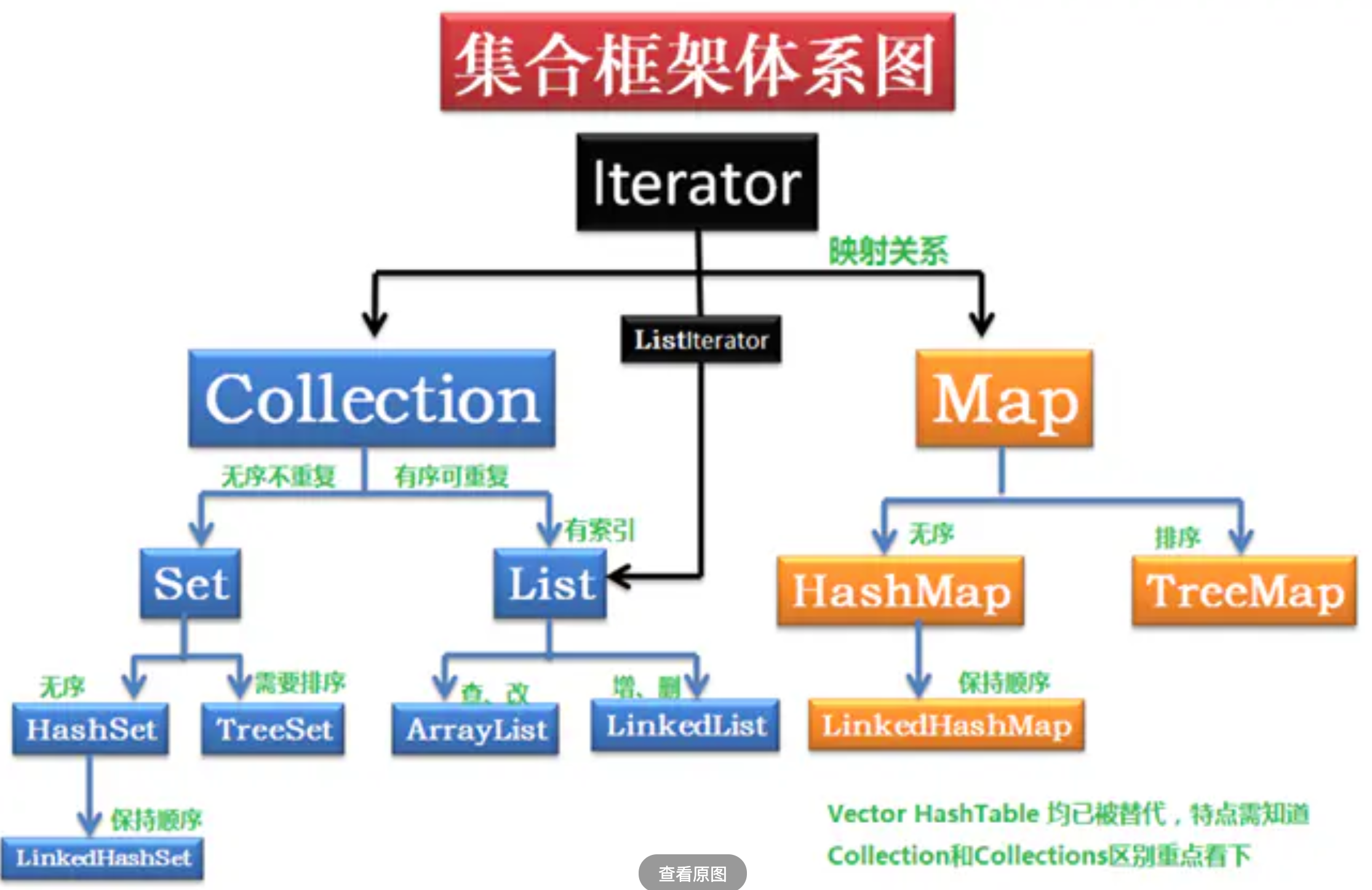
**条件Condition**

调用Condition对象的相关方法，可以方便的挂起和唤醒线程。

### 列举java的集合以及集合之间的继承关系

<https://www.jianshu.com/p/e203b23acf45>

java集合：  
集合类：  
集合类存放的都是对象的引用，而非对象本身，出于表达上的便利，我们称集合中的对象就是指集合中对象的引用（reference)。集合类型主要有3种：set(集)、list(列表)和map(映射)。



### List,Set,Map的区别

<https://www.cnblogs.com/jxxblogs/p/11561629.html>

1、List、Set都是继承自Collection接口，Map则不是

2、List特点：元素有放入顺序，元素可重复 ，Set特点：元素无放入顺序，元素不可重复，重复元素会覆盖掉，（注意：元素虽然无放入顺序，但是元素在set中的位置是有该元素的HashCode决定的，其位置其实是固定的，加入Set 的Object必须定义equals()方法 ，另外list支持for循环，也就是通过下标来遍历，也可以用迭代器，但是set只能用迭代，因为他无序，无法用下标来取得想要的值。）

3、Set和List对比：

Set：检索元素效率低下，删除和插入效率高，插入和删除不会引起元素位置改变。

List：和数组类似，List可以动态增长，查找元素效率高，插入删除元素效率低，因为会引起其他元素位置改变。

4、Map适合储存键值对的数据

5、线程安全集合类与非线程安全集合类 :

* LinkedList、ArrayList、HashSet是非线程安全的，Vector是线程安全的;
* HashMap是非线程安全的，HashTable是线程安全的;
* StringBuilder是非线程安全的，StringBuffer是线程安全的。

### List和Set和Map的实现方式以及存储方式

List：

常用实现方式有：ArrayList和LinkedList

ArrayList  的存储方式：数组，查询快

LinkedList的存储方式：链表，插入，删除快

Set：

常用实现方式有：HashSet和TreeSet

HashSet的存储方式：哈希码算法，加入的对象需要实现hashcode（）方法，快速查找元素

TreeSet的存储方式：按序存放，想要有序就要实现Comparable接口  
  
附加：

集合框架提供了2个实用类：collections（排序，复制、查找）和Arrays对数组进行（排序，复制、查找）

Map：

常用实现方式有：HashMap和TreeMap

HashMap的存储方式：哈希码算法，快速查找键值

TreeMap存储方式：对键按序存放

### HashMap的实现原理

<https://www.cnblogs.com/fenjyang/p/11486403.html>

<https://www.cnblogs.com/fenjyang/p/11486403.html>

### HashMap如何put数据（从HashMap源码角度讲解）？

<https://blog.csdn.net/u011277123/article/details/80467564>

### HashMap怎么手写实现？

<https://www.cnblogs.com/chenfei-java/p/10674341.html>

### ConcurrentHashMap的实现原理

<https://www.jianshu.com/p/865c813f2726>

**总结与思考**

其实可以看出JDK1.8版本的ConcurrentHashMap的数据结构已经接近HashMap，相对而言，ConcurrentHashMap只是增加了同步的操作来控制并发，从JDK1.7版本的ReentrantLock+Segment+HashEntry，到JDK1.8版本中synchronized+CAS+HashEntry+红黑树,相对而言，总结如下思考

1. JDK1.8的实现降低锁的粒度，JDK1.7版本锁的粒度是基于Segment的，包含多个HashEntry，而JDK1.8锁的粒度就是HashEntry（首节点）
2. JDK1.8版本的数据结构变得更加简单，使得操作也更加清晰流畅，因为已经使用synchronized来进行同步，所以不需要分段锁的概念，也就不需要Segment这种数据结构了，由于粒度的降低，实现的复杂度也增加了
3. JDK1.8使用红黑树来优化链表，基于长度很长的链表的遍历是一个很漫长的过程，而红黑树的遍历效率是很快的，代替一定阈值的链表，这样形成一个最佳拍档
4. JDK1.8为什么使用内置锁synchronized来代替重入锁ReentrantLock，我觉得有以下几点
   1. 因为粒度降低了，在相对而言的低粒度加锁方式，synchronized并不比ReentrantLock差，在粗粒度加锁中ReentrantLock可能通过Condition来控制各个低粒度的边界，更加的灵活，而在低粒度中，Condition的优势就没有了
   2. JVM的开发团队从来都没有放弃synchronized，而且基于JVM的synchronized优化空间更大，使用内嵌的关键字比使用API更加自然
   3. 在大量的数据操作下，对于JVM的内存压力，基于API的ReentrantLock会开销更多的内存，虽然不是瓶颈，但是也是一个选择依据

### ArrayMap和HashMap的对比

<https://www.jianshu.com/p/80f6b53e2b98>

### HashSet与HashMap怎么判断集合元素重复？

<https://blog.csdn.net/github_37130188/article/details/96508223>

HashMap中判断元素是否相同主要有两个方法，一个是判断key是否相同，一个是判断value是否相同;

HashSet不能添加重复的元素，当调用add（Object）方法时候，首先会调用Object的hashCode方法判hashCode是否已经存在，如不存在则直接插入元素；

如果已存在则调用Object对象的equals方法判断是否返回true，如果为true则说明元素已经存在，如为false则插入元素。

发现HashSet竟然是借助HashMap来实现的，利用HashMap中Key的唯一性，来保证HashSet中不出现重复值。

从这段代码中可以看出，HashMap中的Key是根据对象的hashCode() 和 euqals()来判断是否唯一的。

结论：为了保证HashSet中的对象不会出现重复值，在被存放元素的类中必须要重写hashCode()和equals()这两个方法。

### 集合Set实现Hash怎么防止碰撞

<https://blog.csdn.net/zeb_perfect/article/details/52574915>

### 数组和链表的区别

<https://www.cnblogs.com/wzb0228/p/11670436.html>

### 二叉树的深度优先遍历和广度优先遍历的具体实现

## （四） 线程、多线程和线程池

### 开启线程的三种方式？

<https://www.cnblogs.com/lgjava/p/9997126.html>

### 为什么要有线程，而不是仅仅用进程？

<https://blog.csdn.net/github_37130188/article/details/97255901>

### run()和start()方法区别

<https://blog.csdn.net/weixin_43258908/article/details/89228353>

### 如何控制某个方法允许并发访问线程的个数？

public class SemaphoreTestMain {

static Semaphore sSemaphore = new Semaphore(6);

public static void main(String[] args) {

final SemaphoreTestMain semaphoreTestMain = new SemaphoreTestMain();

for (int i = 0 ; i < 1000;i++)

{

Thread myThread = new Thread()

{

@Override

public void run() {

super.run();

try {

semaphoreTestMain.test();

} catch (InterruptedException e) {

e.printStackTrace();

}

}

};

myThread.setName("threat index:" + i);

myThread.start();

}

}

public void test() throws InterruptedException {

sSemaphore.acquire();

System.out.println(Thread.currentThread().getName() + "--in");

Thread.sleep(1000);

System.out.println(Thread.currentThread().getName() + "--out" );

sSemaphore.release();

}

}

static Semaphore sSemaphore = new Semaphore(6) 表示该方法最多允许几个线程访问  
sSemaphore.acquire():调用一次，允许进入方法的线程数量减一  
sSemaphore.release()：调用一次，允许进入方法的线程数量加1  
当允许进入方法的线程数量减为0的时候其他线程就等待，直到允许的线程数量大于0  
sSemaphore.availablePermits()：可以获取当前允许进入方法的线程数量

### 在Java中wait和seelp方法的不同