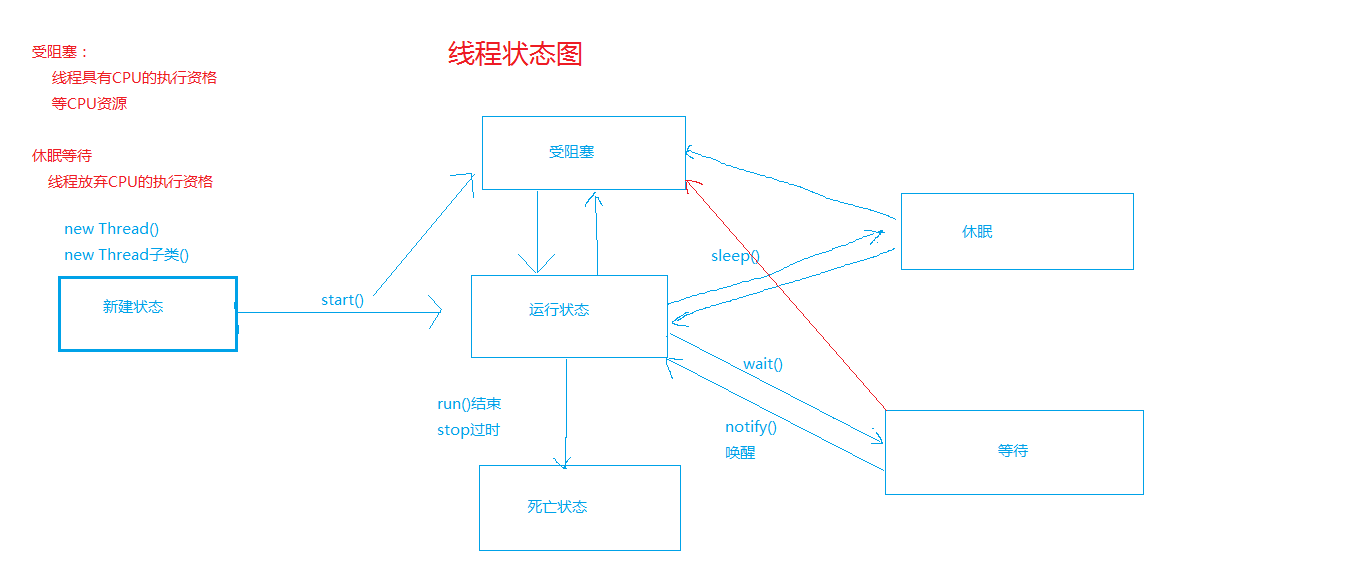
# Java 基础

## 1、== 和 equals

|  |
| --- |
| 1. **== 既可以比较基本类型也可以比较引用类型。对于基本类型就是比较值，对于引用类型就是比较内存地址** 2. **Equals的话，它是属于java.lang.Object类里面的方法，如果该方法没有被重写过默认也是==;我们可以看到String类的equals方法是被重写过的，而且String类在日常开发中用的比较多，久而久之，形成了equals是比较值的错误观点。**   **3 具体要看这有没有重写Object的hashCode方法和equals方法来判断，当hashCode相等时，equals也相等，equals相等时，hashCode也相等。** |

## 2、 线程状态图：



## 3. 线程池原理：



# 尚硅谷面试总结：

## 1、javaSE面试题：-----自增变量

### 1.1 标识符

标识符:

可以用来定义方法、类名称、变量名称的一种标记，在 Java 中所有的标识符的定义风格如下: · 由字母、数字、下划线、$符号组成，其中不能以数字开头，不能是 Java 中的关键字。

例如:demo、x、$Hello 但是从一般的开发来看，一般的标识符用的最多的就是字母、数字、下划线。

### 1.2 运算符(重点)

程序的主要功能就是计算，所以在 Java 中提供了以下的运算符:

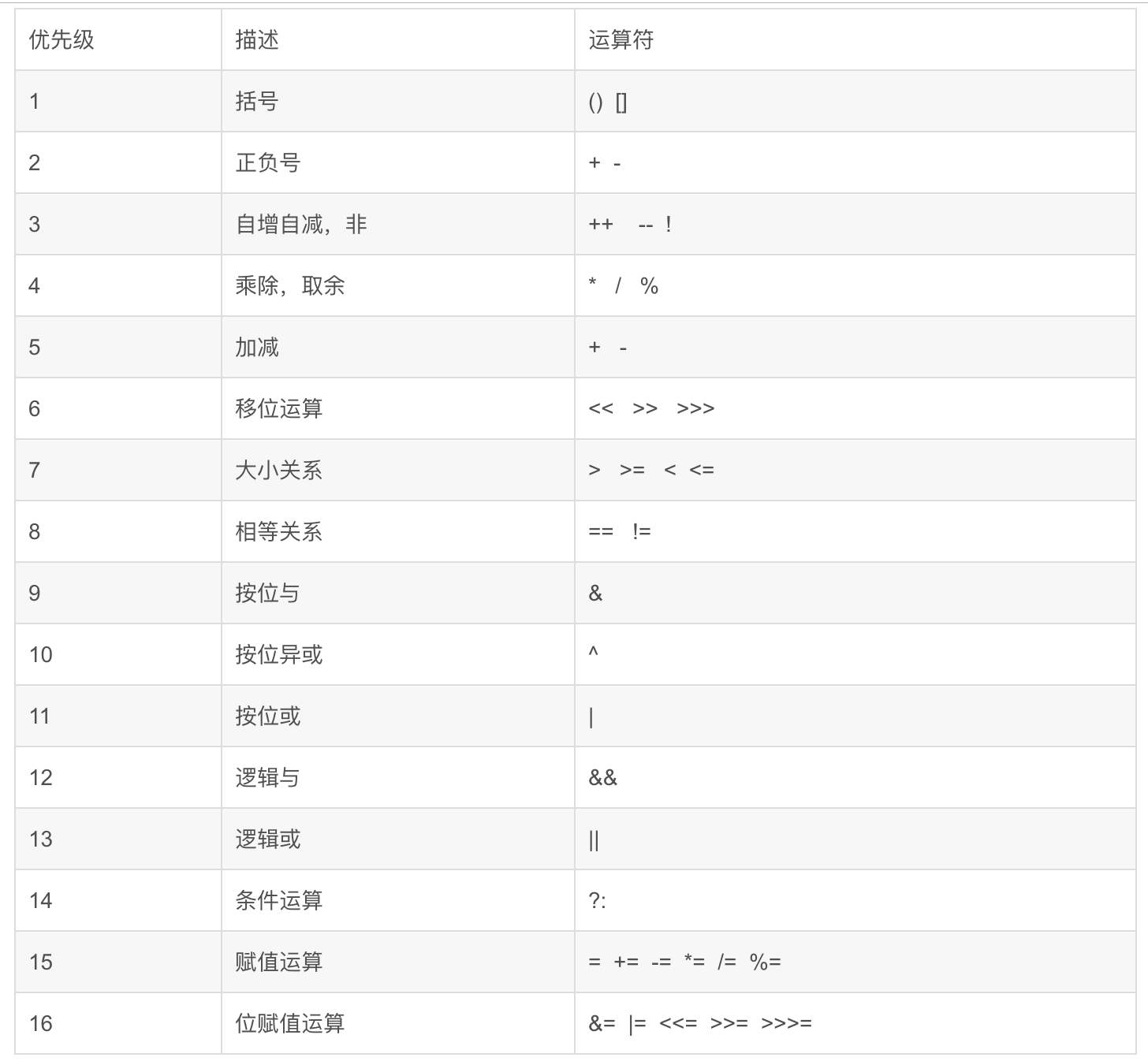
· 赋值运算:=

· 三目运算:布尔表达式 ? 条件满足 : 条不满足

· 数学运算符:+、-、\*、/、%  
· 关系运算符:>、>=、<、<=、==、!=  
· 位运算:>>、<<、>>>、^、~

· 逻辑运算:&&、&、||、|、! · 简便运算符:++、--

#### 运算符优先级：



### 面试题：

|  |
| --- |
| **public class** Test {  **public static void** main(String[] args) {  **int** i = 1;  i = i++;*//i=1* System.***out***.println(i);  **int** j = i++;*//j=1* System.***out***.println(j);  **int** k = i + ++i \* i++;*//k=11，i=4* System.***out***.println(i);  System.***out***.println(j);  System.***out***.println(k);  } } |

### 小结：

* 赋值=，最后计算
* =右边的从左到右加载依次压入操作数栈
* 实际先算哪个，看运算符优先级
* 自增、自减操作都是直接修改变量的值，不经过操作数栈
* 最后的赋值之前，临时结果也是存储在操作数栈中

## 2、javaSE面试题-----单例设计模式

### 2.1、 什么是singleton？

|  |
| --- |
| Singleton：在Java中即指单例设计模式，它是软件开发中最常见的设计模式之一。  单例设计模式：即某个类在整个系统中只能有一个实例对象可被获取和使用的代码模式。  例如：代表JVM运行环境的Runtime类  要点：   * + 一是某个类只能有一个实例 * 构造器私有化   + 二是它必须自行创建这个实例 * 含有一个该类的静态变量来保存这个唯一的实例   + 三是它必须自行向整个系统提供这个实例 * 对外提供获取该实例对象的方法：   （1）直接暴露（2）用静态变量的get方法获取 |

几种常见形式：

* 饿汉式：直接创建对象，不存在线程安全问题
* 直接实例化饿汉式（简洁直观）

|  |
| --- |
| */\*\*  \* 饿汉式： 直接实例化饿汉式  \* 在类初始化时直接创建实例对象，不管你是否需要这个对象都会创建  \* (1)构造器私有化  \* （2）自行创建，并且用静态变量保存  \* (3)向外提供这个实例  \* （4）强调这是一个单例，我们可以用final修饰  \*/* **public class** Singleton1 {  **public static final** Singleton1 ***INSTANCE*** = **new** Singleton1();  **private** Singleton1(){   } } |

* 枚举式（最简洁）

|  |
| --- |
| */\*\*  \* 枚举式饿汉式：  \* 枚举类型： 表示该类型的对象是有限的几个  \* 我们可以限定一个，就成为了单例  \*/* **public enum** Singleton2 {  ***INSTANCE*** } |

* 静态代码块饿汉式（适合复杂实例化）

|  |
| --- |
| */\*\*  \* 饿汉式：静态代码块饿汉式  \*/* **public class** Singleton3 {  **public static final** Singleton3 ***INSTANCE***;  **private** String **info**;  **static** {  Properties pro = **new** Properties();  **try** { *// pro.load(Singleton3.class.getClassLoader().getResourceAsStream("properties/singleproperties"));* pro.load(**new** FileReader(**"properties/single.properties"**));  } **catch** (IOException e) {  e.printStackTrace();  }  String single = pro.getProperty(**"single"**);  ***INSTANCE*** = **new** Singleton3(single);  }  **private** Singleton3(String info){  **this**.**info** = info;  }   **public** String getInfo() {  **return info**;  }   **public void** setInfo(String info) {  **this**.**info** = info;  }   @Override  **public** String toString() {  **return "Singleton3{"** +  **"info='"** + **info** + **'\''** +  **'}'**;  } } |

single.properties 文件：

|  |
| --- |
| **single**=**guigu** |

* 懒汉式：延迟创建对象
* 线程不安全（适用于单线程）

|  |
| --- |
| */\*\*  \* 懒汉式：  \* 延迟创建这个实例对象  \* 线程不安全的（适用于单线程）  \* （1）构造器私有化  \* （2）用一个静态变量的保存这个唯一的实例  \* (3) 提供一个静态方法，获取这个实例对象  \*/* **public class** Singleton4 {  **private static** Singleton4 *instance*;  **private** Singleton4(){   }  **public static** Singleton4 getInstance(){  **if**(*instance* == **null**){  **try** {  Thread.*sleep*(100);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  *instance* = **new** Singleton4();  }  **return** *instance*;  }  } |

* 线程安全（适用于多线程）

|  |
| --- |
| */\*\*  \* 懒汉式：  \* 线程安全的（适用于多线程）  \* （1）构造器私有化  \* （2）用一个静态变量的保存这个唯一的实例  \* (3) 提供一个静态方法，获取这个实例对象  \*/* **public class** Singleton5 {  **private static** Singleton5 *instance*;  **private** Singleton5(){  }  **public static** Singleton5 getInstance(){  **if**(*instance* == **null**){*//判空，如果instance存在，则不再进入锁中创建对象，保证效率* **synchronized** (Singleton5.**class**){  **if**(*instance* == **null**){  **try** {  Thread.*sleep*(100);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  *instance* = **new** Singleton5();  }  }  }  **return** *instance*;  } } |

* 静态内部类形式（适用于多线程）

|  |
| --- |
| */\*\*  \* 懒汉式： 静态内部类形式  \* 线程安全的（适用于多线程）  \* （1）构造器私有化  \* （2）用一个静态变量的保存这个唯一的实例  \* (3) 提供一个静态方法，获取这个实例对象  \*/* **public class** Singleton6 {  **private** Singleton6(){  }   */\*\*  \* 在内部类被加载和初始化时，才创建INSTANCE实例对象  \* 静态内部类不会自动随着外部类的加载和初始化而初始化，它是要单独去加载和初始化  \* 因为是在内部类加载和初始化时，创建的，因此线程是安全的。  \*/* **private static class** Inner{  **private static final** Singleton6 ***INSTANCE*** = **new** Singleton6();  }  **public static** Singleton6 getInstance(){  **return** Inner.***INSTANCE***;  } } |

#### 小结：

|  |
| --- |
| * 如果是饿汉式，枚举形式最简单 * 如果是懒汉式，静态内部类形式最简单 |

## 3.类初始化和实例初始化过程：

### 3.1 类初始化过程

1. 一个类要创建实例需要先加载并初始化该类

* Main方法所在的类需要先加载和初始化

1. 一个子类要初始化需要先初始化父类
2. 一个类初始化就是执行<clinit>()类方法

* <clinit>()方法由静态变量赋值代码和静态代码块组成
* 类变量显示赋值代码和静态代码块代码从上到下顺序执行
* <clinit>()方法只执行一次

### 3.2 实例初始化过程

1) 实例初始化就是执行<init>()方法

* <init>()方法可能重载有多个，有几个构造器就有几个<init>方法
* <init>()方法由非静态实例变量显示赋值代码和非静态代码块、对应构造器代码组成
* 非静态实例变量显示赋值代码和非静态代码代码块从上到下顺序执行，而对应构造器的代码最后执行
* 每次创建实例对象，调用对应构造器，执行的就是对应<init>方法
* <init>方法的首行是super()或super(实参列表)，即应对父类的<init>方法

### 3.3 方法的重写Override

1) 哪些方法不可以被重写

* Final方法
* 静态方法
* Private等子类中不可见方法

2) 对象的多态性

* 子类如果重写了父类方法，通过子类对象调用的一定是子类重写过得代码
* 非静态方法默认调用对象是this
* This对象在构造器或者说<init>方法中就是正在创建的对象

## 4. 方法的参数传递机制：

1）形参是基本数据类型

* 传递数据值

2） 实参是引用数据类型

* 传递地址值
* 特殊的类型：String、包装类等对象不可变性

|  |
| --- |
| **public class** Exam4 {   **public static void** main(String[] args) {  **int** i = 1;  String str = **"hello"**;  Integer num = 200;  **int**[] arr = {1,2,3,4,5};  MyData my = **new** MyData();  *change*(i,str,num,arr,my);   System.***out***.println(**"i="** + i);  System.***out***.println(**"str="** + str);  System.***out***.println(**"num="** + num);  System.***out***.println(**"arr="** + Arrays.*toString*(arr));  System.***out***.println(**"my="** + my.**a**);  }   **public static void** change(**int** j,String s,Integer n,**int**[] a,MyData m){  j += 1;  s += **"world"**;  n += 1;  a[0] += 1;  m.**a** += 1;  } }  **class** MyData {  **int a** = 10; } |

结果：

|  |
| --- |
| i=1  str=hello  num=200  arr=[2, 2, 3, 4, 5]  my=11 |

分析：

