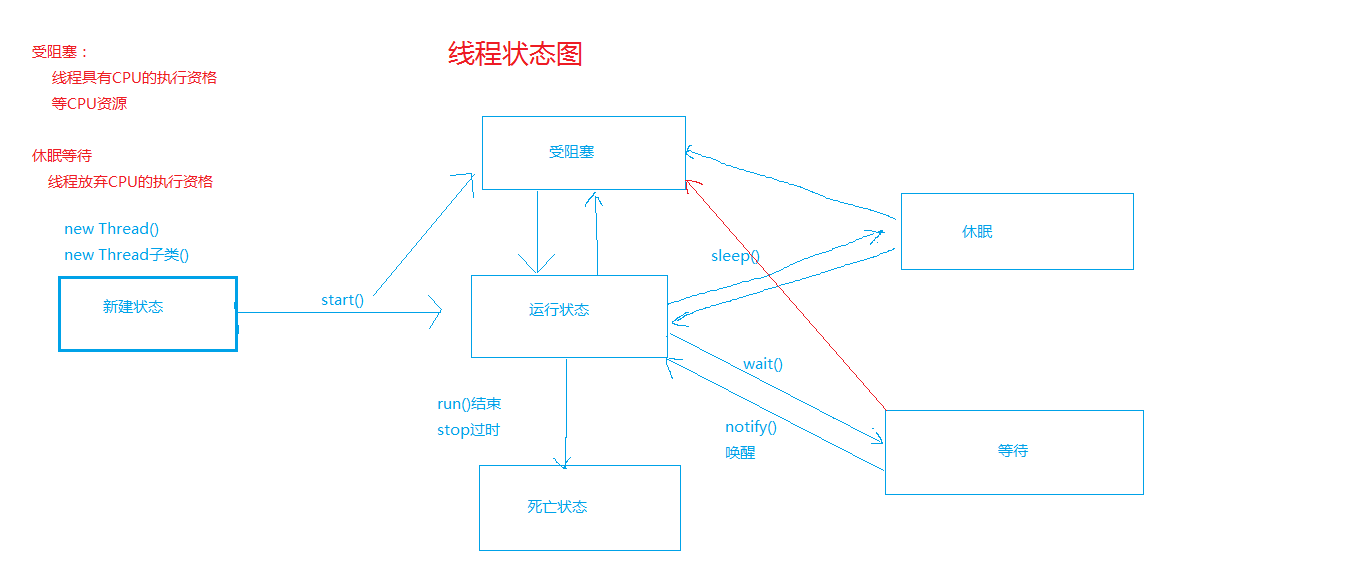
# Java 基础

## 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:

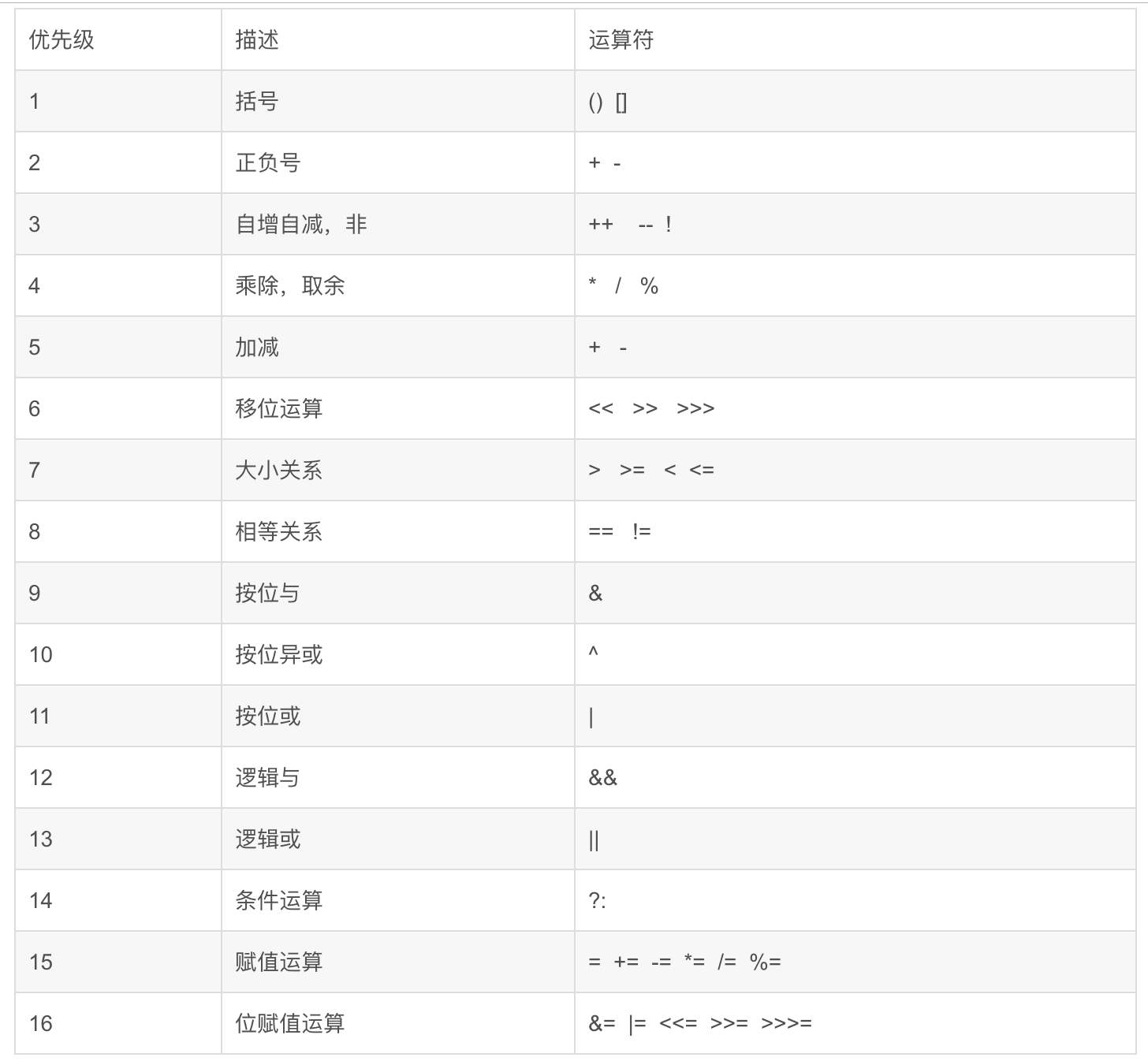
· 赋值运算:=

· 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

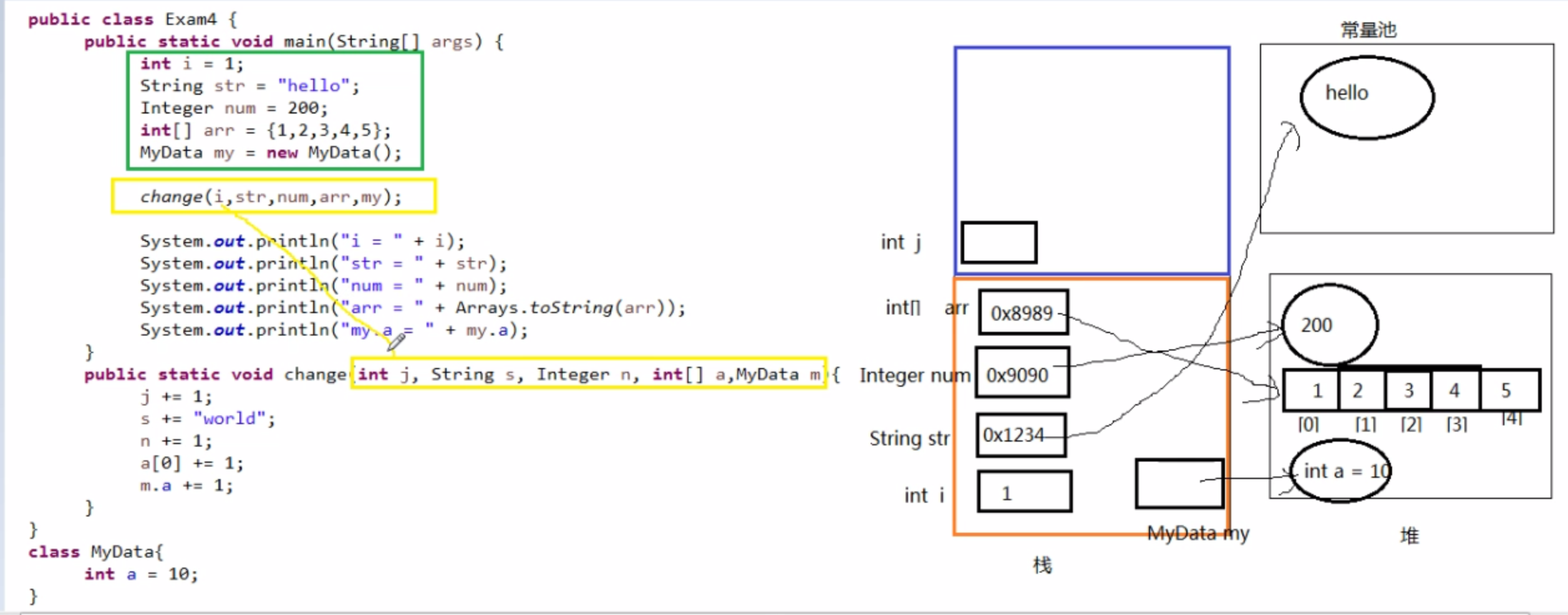
* 传递地址值
* 特殊的类型：String、包装类等对象不可变性

|  |
| --- |
| **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：



## 5. 递归与迭代：

|  |
| --- |
| * 方法调用自身成为递归，利用变量的原值推出新值称为迭代。 * 递归： * 优点：大问题转化为小问题，可以减少代码量，同时代码精简，可读性好； * 缺点：递归调用浪费了空间，而且递归太容易造成堆栈的溢出。 * 迭代： * 优点：代码运行效率好，因为时间只因循环次数增加而增加，而且没有额外的空间开销； * 代码不如递归简洁，可读性好 |

递归：

|  |
| --- |
| **public int** f(**int** n){  **if**(n < 1){  **throw new** IllegalArgumentException(n + **"不能小于1"**);  }  **if**(n==1 || n==2){  **return** n;  }  **return** f(n-2) + f(n-1); } |

迭代：

|  |
| --- |
| **public int** f(**int** n){  **if**(n < 1){  **throw new** IllegalArgumentException(n + **"不能小于1"**);  }  **if**(n==1 || n==2){  **return** n;  }  **int** one = 2;*//初始化为走到第二级台阶的走法* **int** two = 1;*//初始化为走到第一级台阶的走法* **int** sum = 0;  **for**(**int** i = 3; i<=n; i++){  *//最后跨2步+最后跨1步的走法* sum = two + one;  two = one;  one = sum;  }  **return** sum; } |

## 6． 成员变量与局部变量

* 就近原则
* 变量的分类
  + 成员变量：类变量、实例变量
  + 局部变量
* 非静态代码块的执行：每次创建实例对象都会执行
* 方法的调用规则：调用一次执行一次

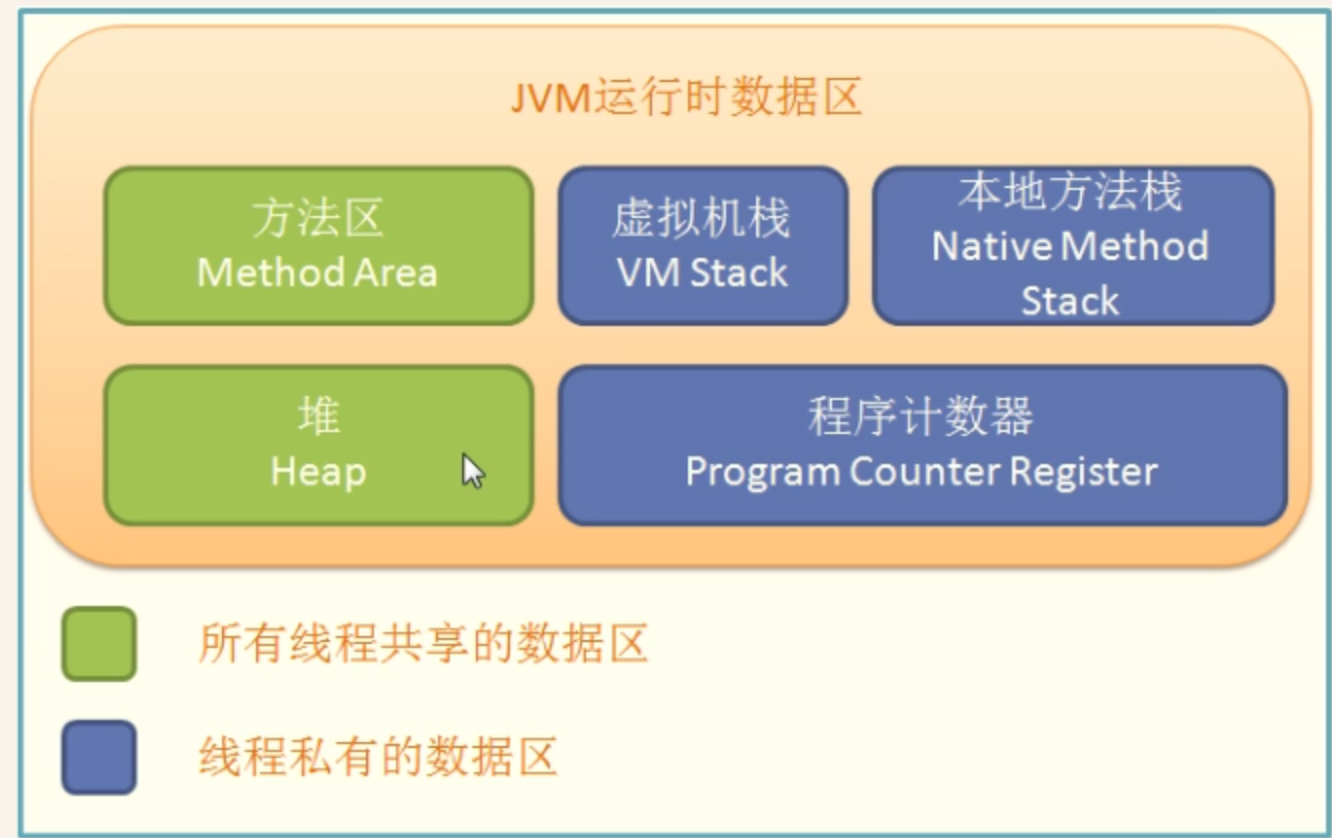
### 6.1 局部变量与成员变量的区别

1. 声名的位置：
   * 局部变量：方法体{}中，形参，代码块{}中
   * 成员变量：类中方法外
     + 类变量：有static修饰
     + 实例变量：没有static修饰
2. 修饰符

* 局部变量：final
* 成员变量：public、protected、private、final、static、volatile、transient

1. 值存储的位置

* 局部变量：栈
* 实例变量：堆
* 类变量：方法区



|  |
| --- |
| **堆(Heap)**,此内存区域的唯一目的就是存放对象实例，几乎所有的对象实例都在这里分配内存。这一点在java虚拟机规范中的描述是：所有的对象实例以及数组都要在堆上分配。  **通常所说的栈（stack），**是指虚拟机栈。虚拟机栈用于存储局部变量表等。局部变量表存放了编译期可知长度的各种基本数据类型(Boolean、byte、char、short、int、float、long、double)、对象引用（reference类型，它不等于对象本身，是对象在堆内存的首地址）。方法执行完，自动释放。  **方法区（Method Area）**用于存储已被虚拟机加载的类信息、常亮、静态变量、即时编译器编译后的代码等数据。 |

1. 作用域

* 局部变量：从声名处开始，到所属的}结尾
* 实例变量：在当前类中“this.”（有事this.可以缺省），在其他类中“对象名.”访问
* 类变量：在当前类中“类名.”（有时类名.可以省略），在其他类中“类名.”或“对象名.”访问

1. 生命周期

* 局部变量：每一个线程，每一次调用执行都是新的生命周期
* 实例变量：随着对象的创建而初始化，随着对象的被回收而消亡，每一个对象的实例变量是独立的。
* 类变量：随着类的初始化而初始化，随着类的卸载而消亡，该类的所有对象的类变量是共享的

### 6.2 当局部变量与xx变量重名时，如何区分：

|  |
| --- |
| 1. 局部变量与实例变量重名   在实例变量前面加“this.”   1. 局部变量与类变量重名   在类变量前面加“类名.” |

|  |
| --- |
| /\*\*  \* 局部变量与成员变量  \*/  public class Exam5 {  static int s;  int i;  int j;  {  int i = 1;  i++;  j++;  s++;  }  public void test(int j){  j++;  i++;  s++;  }  public static void main(String[] args) {  Exam5 obj1 = new Exam5();  Exam5 obj2 = new Exam5();  obj1.test(10);  obj1.test(20);  obj2.test(30);  System.out.println(obj1.i + " "+ obj1.j +" " + obj1.s);  System.out.println(obj2.i + " "+ obj2.j +" " + obj2.s);  }  } |

## 7. SSM---面试题：Spring Bean 的作用域之间有什么区别

在Spring 中，可以在<bean>元素的scope属性里设置bean的作用域，以决定这个bean是单实例的还是多实例的。

默认情况下，spring只为每个在IOC容器里声明的bean创建唯一一个实例，整个IOC容器范围内都能共享该实例：所有后续的getBean()调用和bean引用都将返回这个唯一的bean实例。该作用域被称为singleton，它是所有bean的默认作用域。

|  |  |
| --- | --- |
| 类别 | 说明 |
| Singleton | 在SpringIOC容器中仅存一个Bean实例，Bean以但实例的方式存在 |
| Prototype | 每次调用getBean()时都会返回一个新的实例 |
| Request | 每次HTTP请求都会创建一个新的Bean，该作用域仅适用于WebApplicationContext环境 |
| Session | 同一个HTTP Session共享一个Bean，不同的HTTP Session使用不同的Bean。该作用域仅适用于WebApplicationContext环境 |

### ★bean的作用域

可以通过scope属性来指定bean的作用域

-singleton：默认值。当IOC容器一创建就会创建bean的实例，而且是单例的，每次得到的都是同一个

-prototype：原型的。当IOC容器一创建不再实例化该bean，每次调用getBean方法时再实例化该bean，而且每调用一次创建一个对象

-request：每次请求实例化一个bean

-session：在一次会话中共享一个bean

## 8． SSM面试题 Spring支持的常用数据库事务传播属性和事务隔离级别？

### ☆propagation：用来设置事务的传播行为

事务的传播行为：一个方法运行在了一个开启了事务的方法中时，当前方法是使用原来的事务还是开启一个新的事务。

* Propagation.REQUIRED:默认值，使用原来的事务
* Propagation.REQUIRES\_NEW：将原来的事务挂起，开启一个新的事务

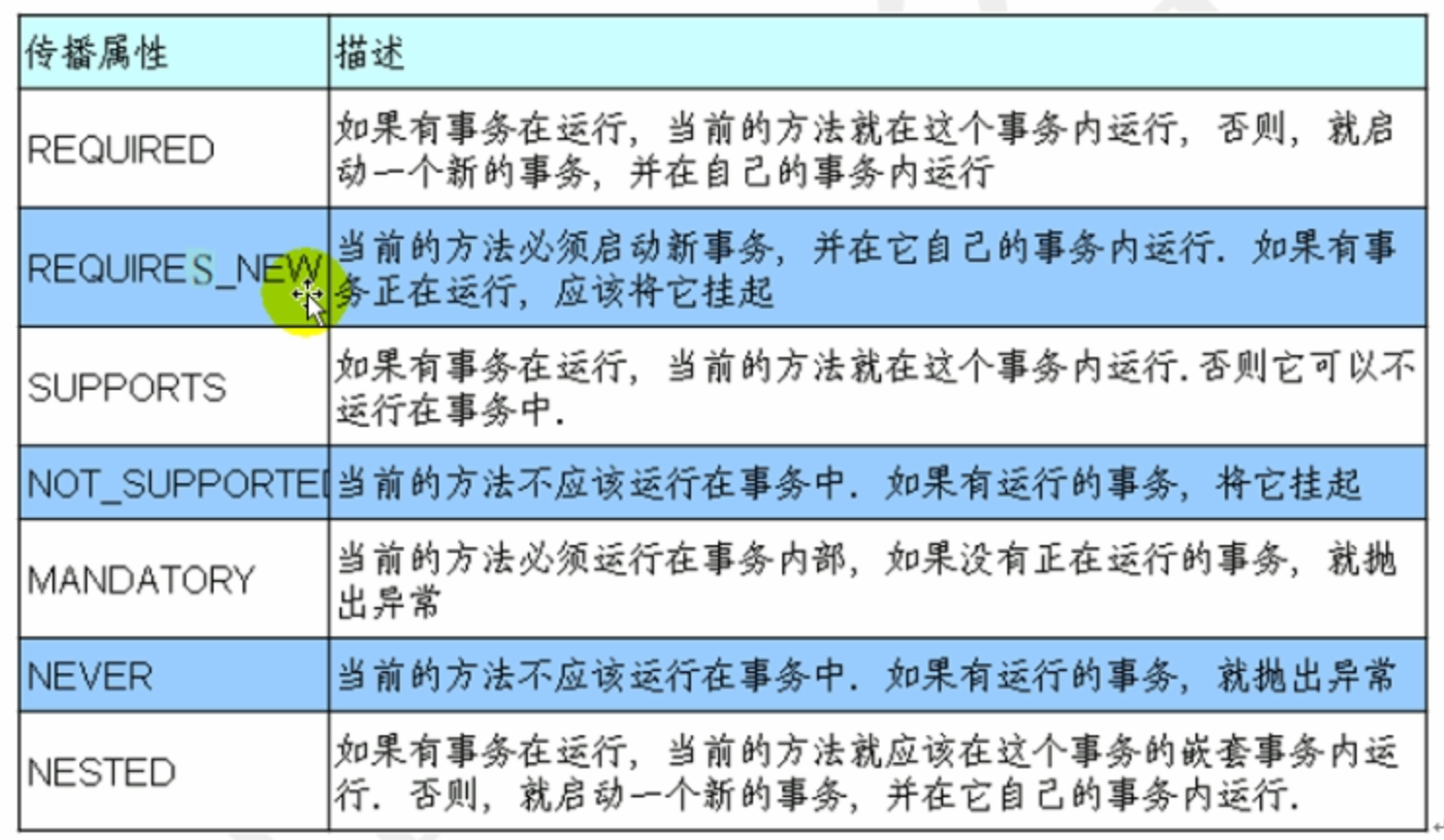
### ☆isolation：用来设置事务的隔离级别

* Isolation.REPEATABLE\_READ:可重复读，mysql默认的隔离级别
* Isolation.READ\_COMMITTED:读已提交，Oracle默认的隔离级别，开发时通常使用的隔离级别。

### 事务的传播行为简介

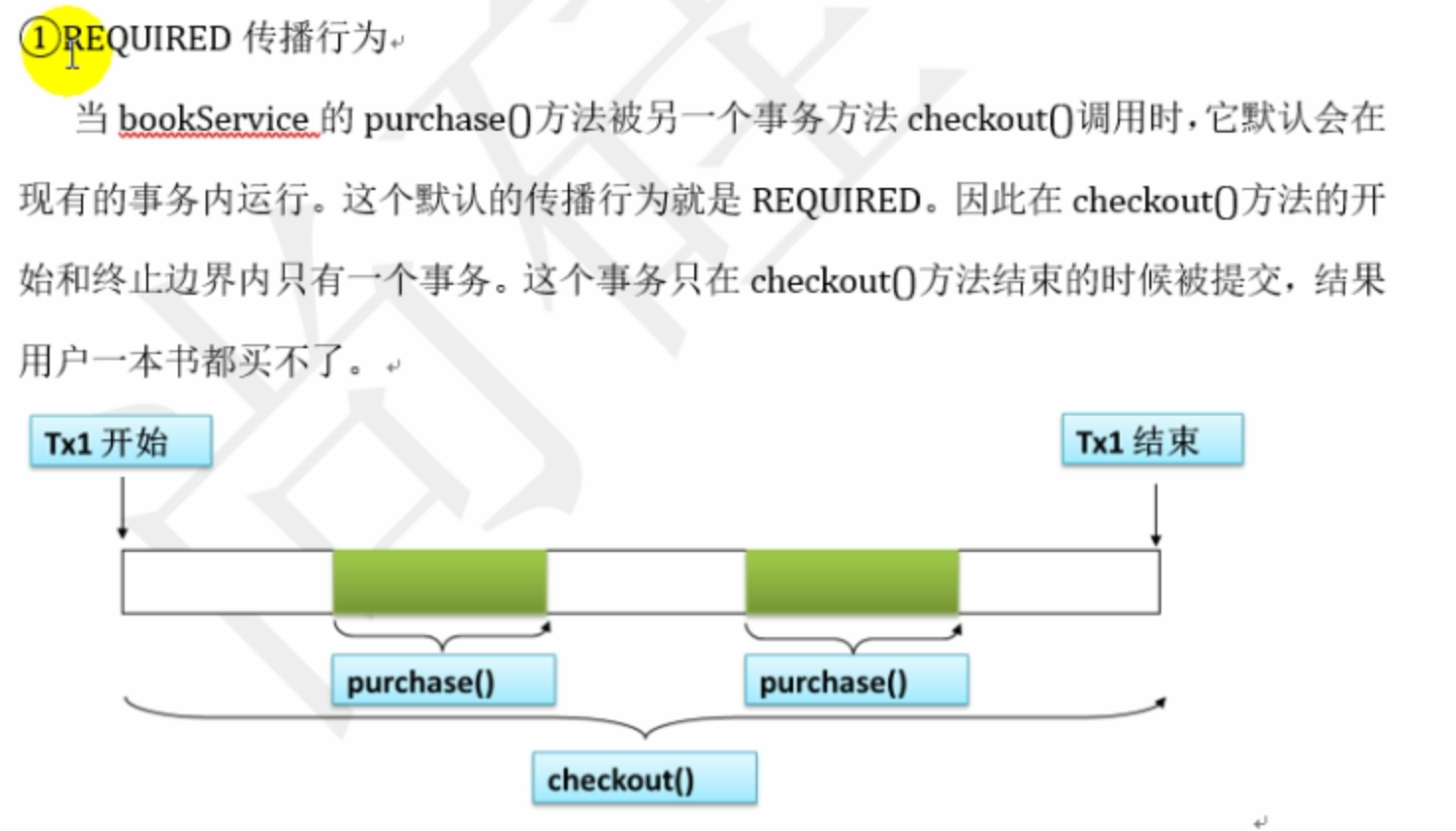
当事务方法被另一个事务方法调用时，必须指定事务应该如何传播。例如：方法可能继续在现有事务中运行，也可能开启一个新事务，并在自己的事务中运行。

事务的传播行为可以由传播属性指定。Spring定义了7种类的传播行为。



#### REQUIRED

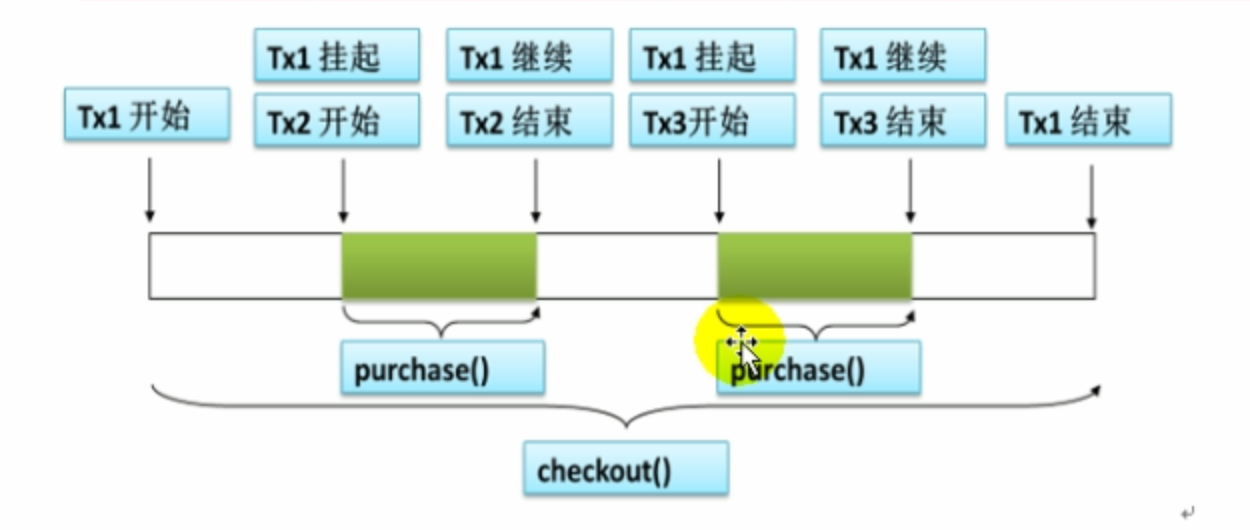
当bookService的purchase()方法被另一个事务方法checkout()调用时，它默认会在现有的事务内运行。这个默认的传播行为就是REQUIRED。因此在checkout()方法的开始和终止边界内只有一个事务。这个事务只在checkout()方法结束的时候被提交，结果用户一本书都买不了。



上图所示的逻辑是在checkOut()方法里面循环调用purchase()。

#### ② REQUIRES\_NEW 传播行为

表示该方法必须启动一个新事务，并在自己的事务内运行。如果有事务在运行，就应该先挂起它。



### 事务的隔离级别

#### 数据库事务并发问题：

假设现在有两个事务：Transaction01和Transaction02并发执行。

1. 脏读
   1. Transcation01读取某条记录值从20修改为30.
   2. Transcation02读取了Transcation01更新后的值：30
   3. Transcation01回滚，AGE值恢复到了20
   4. Transcation02读取到30就是一个无效的值。
2. 不可重复读
   1. Transcation01读取AGE值为20
   2. Transcation02将AGE值修改为30
   3. Transcation01再次读取AGE值为30，和第一次读取不一致。
3. 幻读：
   1. Transcation01读取了STUDENT表中的一部分数据
   2. Transcation02向STUDENT表中插入了新的行
   3. Transcation01读取了STUDENT表时，多出了一些行

#### 隔离级别：



数据库系统必须具有隔离并发运行各个事务的能力，使它们不会相互影响，避免各种并发问题。一个事务与其他事务隔离的程度称为隔离级别。SQL标准中国规定了多种隔离级别，不同隔离级别对应不同的干扰程度，隔离级别越高，数据一致性就越好，但并发性弱。

1. 读未提交：READ UNCOMMITED

允许Transcation01读取Transcation02未提交的修改。

1. 读已提交：READ COMMITED

要求Transcation01只能读取Transcation02已提交的修改。

1. 可重复读：REPEATABLE READ

确保Transcation01可以多次从一个字段中读取到相同的值，即Transcation01执行期间禁止其它事务对这个字段进行更新。

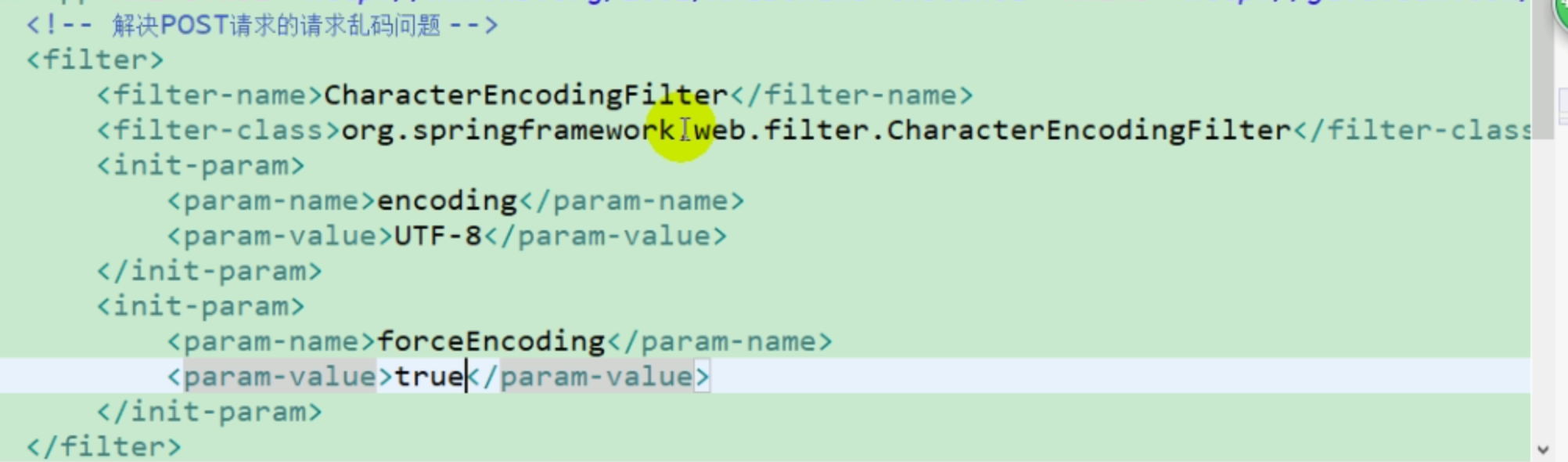
1. 串行化：SERIALIZABLE

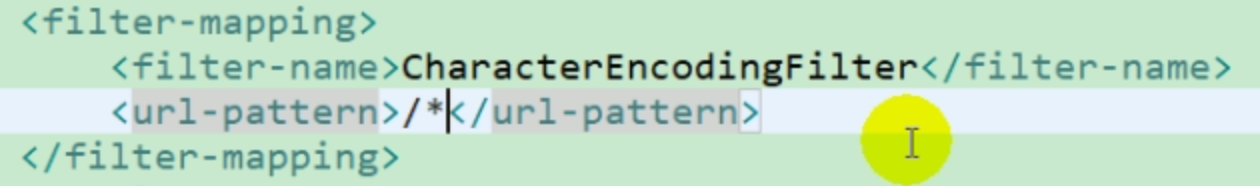
确保Transcation01可以多次从一个表中读取到相同的行，在Transcation01执行期间，禁止其它事务对这个表进行添加、更新、删除操作。可以避免任何并发问题。但性能十分低下。





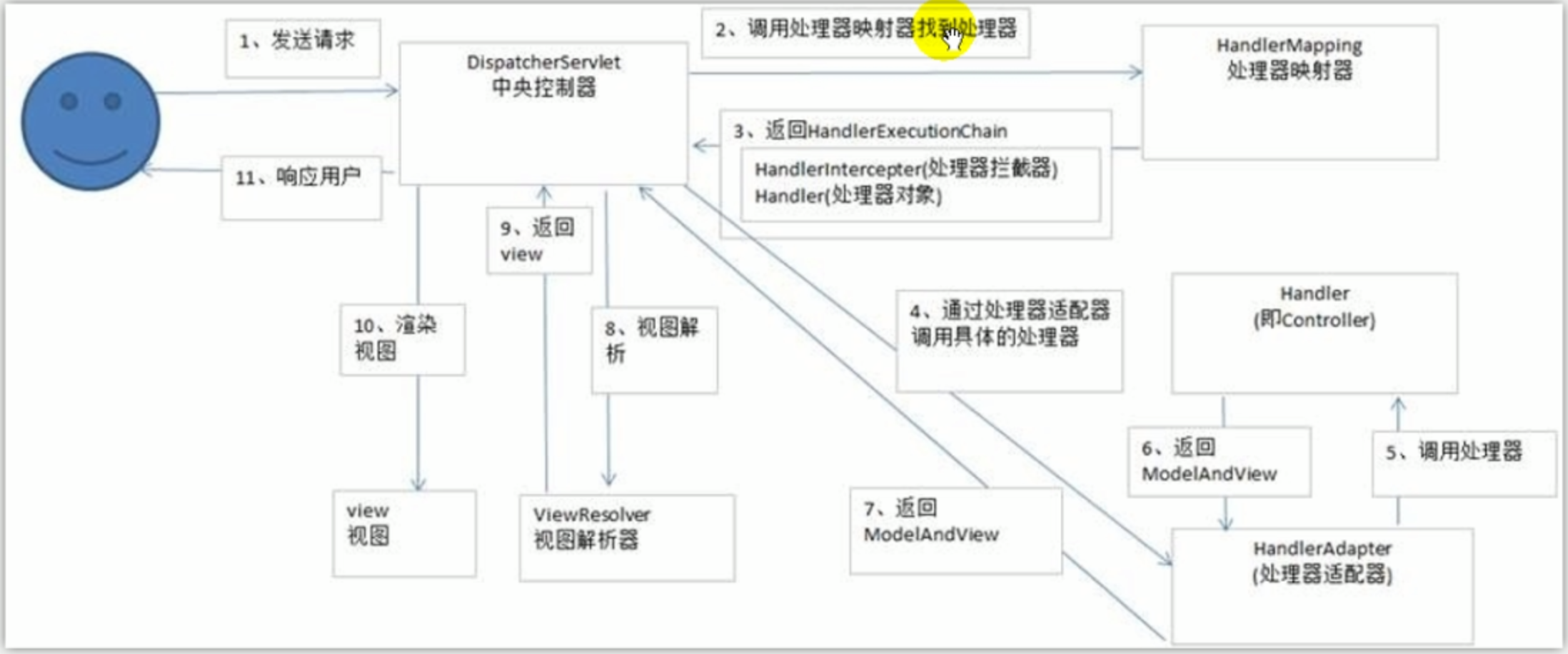
## 9. SSM面试题：springMVC如何解决post请求乱码问题：





重点：filter中的filter-name与filter-mapping中的filter-name中的值必须相同。

## 10、SSM面试题：springMVC处理流程



## 11.SSM面试题：在mybatis中当实体类中的属性名和表中的子段名不一样，怎么办？

1. 写sql语句时起别名
2. 在mybatis的全局配置文件中开启驼峰命名规则



1. 在Mapper映射文件中使用resultMap来自定义映射规则

