# 1工厂设计模式

## 1.1工厂方法模式

### 概述

* 工厂：就是生产特定产品的
* 实现方式：

1>创建一个抽象工厂类，声明抽象方法

2>写一个具体的抽象工厂类的子类，由子类负责对象的创建

* 优点：后期维护容易，增强了系统的扩展性
* 缺点：需要额外的编写代码，增加了工作量

### 代码

|  |
| --- |
| **抽象动物工厂** |
| **public** **abstract** **class** AnimalFactory {  **public** **abstract** Animal createAnimal();  //public abstract Animal newAnimal();  } |
| **狗工厂** |
| **public** **class** DogFactory **extends** AnimalFactory{  @Override  **public** Animal createAnimal() {  // **TODO** Auto-generated method stub  Dog dog = **new** Dog();  dog.setName("小黑");  dog.setColor("Black");  //.......  **return** dog;  }  } |
| **猪工厂** |
| **public** **class** PigFactory **extends** AnimalFactory{  @Override  **public** Animal createAnimal() {  // **TODO** Auto-generated method stub  Pig pig = **new** Pig();  **return** pig;  }  } |
| 工厂方法类的使用 |
| //使用工厂方法模式创建对象  DogFactory dogFactory = **new** DogFactory();  Animal dog1 = dogFactory.createAnimal();  dog1.eat();    PigFactory pigFactory = **new** PigFactory();  Animal pig2 = pigFactory.createAnimal();  pig2.eat(); |

## 1.2简单工厂模式

### 概述

* 又叫静态工厂方法模式，它定义一个具体的工厂类负责创建**一些**类的实例
* 特点：一个类负责很多对象的创建

### 代码

|  |
| --- |
| **public** **class** AnimalFactory {  /\*public static Dog createDog(){  Dog dog = new Dog();  return dog;  }\*/    //返回值可以是抽象类，接口，具体的类  **public** **static** Animal createDog(){  Dog dog = **new** Dog();  **return** dog;  }  **public** **static** Pig createPig(){  Pig pig = **new** Pig();  **return** pig;  }  } |

# 2类的加载

## 2.1类加载的过程

当程序要使用某个类时，如果该类还未被加载到内存中，则系统会通过**加载，连接，初始化三步**来实现对这个类进行初始化。

**加载**

就是指将class文件读入内存，并为之创建一个Class对象。任何类被使用时系统都会建立一个Class对象。

**连接**

验证 是否有正确的内部结构，并和其他类协调一致

准备 负责为类的静态成员分配内存，并设置默认初始化值

解析 将类的二进制数据中的符号引用替换为直接引用

**初始化**

就是给属性赋值

## 2.2类加载时机 -【面试题】

* **加载的原则-用到的时候才加字节码**
* 比如：
  + 创建类的实例
  + 访问类的静态变量，或者为静态变量赋值
  + 调用类的静态方法
  + 初始化某个类的子类
  + 直接使用java.exe命令来运行某个主类
  + 使用反射方式来强制创建某个类或接口对应的java.lang.Class对象

## 2.3类加载器

### 类加载器的概述

负责将.class文件加载到内存中，并为之生成对应的Class对象。虽然我们不需要关心类加载机制，但是了解这个机制我们就能更好的理解程序的运行。

### 类加载器的分类

* Bootstrap ClassLoader 根类加载器
* Extension ClassLoader 扩展类加载器
* System ClassLoader 系统类加载器

### 各类加载器的作用

**Bootstrap ClassLoader 根类加载器**

也被称为引导类加载器，负责Java核心类的加载

比如System,String等。在JDK中JRE的lib目录下rt.jar文件中

**Extension ClassLoader 扩展类加载器**

负责JRE的扩展目录中jar包的加载。

在JDK中JRE的lib目录下ext目录

**System ClassLoader 系统类加载器**

负责在JVM启动时加载来自java命令的class文件，以及classpath环境变量所指定的jar包和类路径

# 3 反射

## 3.1反射概述

* JAVA反射机制是在运行状态中，对于任意一个类都能够知道这个类的所有属性和方法；
* 对于任意一个对象，都能通过反射够调用它的任意一个方法和属性；
* 要想解剖一个类,必须先要获取到该类的字节码文件对象。

## 3.2 获取字节码对象三种方式

* Object类的getClass()方法,判断两个对象是否是同一个字节码文件
* 静态属性class,锁对象
* Class类中静态方法forName()

|  |
| --- |
| **public** **static** **void** main(String[] args) **throws** ClassNotFoundException {  //1.Object类的getClass()方法  Person p = **new** Person();  Class clz1 = p.getClass();    //2.静态属性clas  Class clz2 = Person.**class**;    //3.Class类中静态方法forName()  /\*\*  \* 参数 forName(String className) 传类全路路(包名+类名)  \*/  Class clz3 = Class.*forName*("lesson05.Person");    //4.字节码对象在内存中只有一个  System.***out***.println("clz1:" + clz1.hashCode());  System.***out***.println("clz2:" + clz2.hashCode());  System.***out***.println("clz3:" + clz3.hashCode());  } |

## 3.3 案例读取配置文件创建字节码对象

|  |
| --- |
| //1.读取info.txt文件内容  FileReader fr = **new** FileReader("info.txt");  BufferedReader br = **new** BufferedReader(fr);  String className = br.readLine();  br.close();    //2.获取字节码对象  Class clz = Class.*forName*(className);  System.***out***.println(clz); |

## 3.4 通过反射获取参构造方法

### 使用要点：

1.如果要使用反射，先要获取字节码对象

2.通过字节码对象的getConstructor()可以获取到构造方法对象

3.构造方法对象(Contructor),有个newInstance方法创建这个字节码对象

4.反射是在java.lang.reflect这个包中

5.反射的作用一般是用于写框架(ssh,ssm)

### 代码：

|  |
| --- |
| //1.获取字节码对象  Class clz = Teacher.**class**;    //2.获取构造方法  //2.1 无参构造方法  Constructor c1 = clz.getConstructor();    //2.2 通过构造方法创建对象  Teacher teacher1 = (Teacher) c1.newInstance();  System.***out***.println("teacher1:" + teacher1);    //2.3 获取有参构造方法  /\*\*  \* parameterType 参数类型  \*/  Constructor c2 = clz.getConstructor(String.**class**,String.**class**);  Teacher teacher2 = (Teacher) c2.newInstance("gyf","梅州");//相当于调用new Teacher("gyf","梅州")  System.***out***.println("teacher2:" + teacher2);    Constructor c3 = clz.getConstructor(String.**class**,**double**.**class**);  Teacher teacher3 = (Teacher) c3.newInstance("gyf",1.70);//相当于调用new Teacher("gyf","梅州")  System.***out***.println("teacher3:" + teacher3); |

## 3.5 通过反射获取类属性

### 使用要点

1.Class的etField(String)方法可以获取类中的指定字段(可见的),

2.如果是私有的,可以用getDeclaedField("name")方法获取

3.通过set(obj, "李四")方法可以设置指定对象上该字段的值

4.如果是私有的需要先调用setAccessible(true)设置访问权限,

5.调用get(obj)可以获取指定对象中该字段的值

### 代码

|  |
| --- |
| //1.获取字节码对象  Class clz = Teacher.**class**;    //2.获取color字段  Field colorField = clz.getField("color");  System.***out***.println(colorField);    //3.通过反射给字段赋值  Teacher teacher = **new** Teacher();  //teacher.color = "白色";  colorField.set(teacher, "黄色");    //4.获取私有name属性  Field nameField = clz.getDeclaredField("name");  System.***out***.println(nameField);    //5.通过反射给私有属性赋值  nameField.setAccessible(**true**);//设置私有属性可以访问  nameField.set(teacher, "gyf");    System.***out***.println(teacher);    //6.通过反射获取私有属性的值  Object value = nameField.get(teacher);  System.***out***.println(value); |

## 3.6通过反射获取方法并使用

### 使用要点

1.反射中通过Method类描述方法【构造方法：Contructor,字段:Field】

2.通过Class的getMethod可以获取一个方法

3.通过getDeclaredMethod可以获取私有方法

4.如果要调用私有方法，设置访问权限setAccessible

### 代码

|  |
| --- |
| //1.获取字节码对象  Class clz = Teacher.**class**;    Teacher teacher = **new** Teacher();  /\* teacher.say1();  teacher.say2("小黄鸭");  teacher.say3("小黄鸭",2);\*/    //2.通过反射调用方法  //2.1 获取无参方法  Method m1 = clz.getDeclaredMethod("say1");  //2.1 获取有参方法  Method m2 = clz.getDeclaredMethod("say2", String.**class**);  Method m3 = clz.getDeclaredMethod("say3", String.**class**,**int**.**class**);    //设置私有方法可以访问  m1.setAccessible(**true**);  m2.setAccessible(**true**);  m3.setAccessible(**true**);  //3.2 调用方法  m1.invoke(teacher);  m2.invoke(teacher, "小猪");  m3.invoke(teacher, "小狗",98); |

## 3.7 案例越过泛型的检查

|  |
| --- |
| //1.声明泛型集合  List<Integer> list = **new** ArrayList<Integer>();    list.add(110);  list.add(120);  list.add(130);  //list.add("gyf");    //2.通过反射往集合添加字符串  //2.1 获取字节码对象(Class)  Class clz = list.getClass();    //2.2 通过反射获取方法  Method method = clz.getMethod("add", Object.**class**);    //2.3 调用方法  method.invoke(list, "gyf");    System.***out***.println(list); |

## 3.8 写一个通用的方法，设置对象的属性值

|  |
| --- |
|  |

# 4动态代理

## 概述

1.代理：本来应该自己做的事情，请了别人来做，被请的人就是代理对象。

举例：春节回家买票让人代买

2.在Java中java.lang.reflect包下提供了一个Proxy类和一个InvocationHandler接口

3.通过使用这个类和接口就可以生成动态代理对象。

4.JDK提供的代理只能针对接口做代理。我们有更强大的代理cglib

5.Proxy 通过 newProxyInstance(loader,interfaces,h)创建代理对象

5.InvocationHandler的invoke(proxy,method, args)方法会拦截方法的调用

## 案例代码

|  |
| --- |
| **public** **class** Demo01 {  **public** **static** **void** main(String[] args) {    //1.创建对象  UserServiceImpl usi = **new** UserServiceImpl();  /\* usi.registerUser();  usi.deleteUser();\*/    //2.创建代理对象  UserService proxy = (UserService) Proxy.*newProxyInstance*(usi.getClass().getClassLoader(),  usi.getClass().getInterfaces(),  **new** InvocationHandler() {    @Override  **public** Object invoke(Object proxy, Method method, Object[] args) **throws** Throwable {  // **TODO** Auto-generated method stub  System.***out***.println(method);  System.***out***.println("权限检验...");    //拦截了方法  Object returnObj = method.invoke(usi, args);    System.***out***.println("日志记录");  **return** returnObj;  }  });    //System.out.println(proxy.getClass());  proxy.registerUser();  proxy.deleteUser();      }  }  **interface** UserService{  **public** **void** registerUser();  **public** **void** deleteUser();  }  **class** UserServiceImpl **implements** UserService{  @Override  **public** **void** registerUser() {  //System.out.println("权限校验...");    System.***out***.println("注册一个用户");    //System.out.println("日志记录...");  }  @Override  **public** **void** deleteUser() {    //System.out.println("权限校验...");    System.***out***.println("删除一个用户");    //System.out.println("日志记录...");  }  } |

# 5 模版设计模式

模版模式就是定义一个算法的骨架，而将具体的算法延迟到子类中来实现

|  |
| --- |
| **public** **class** Demo02 {  **public** **static** **void** main(String[] args) {  //模版(Template)设计模式  System.***out***.println(**new** Test1().getScheduleTime());  }  }  **abstract** **class** TimeTemplate{  //获取执行时间  **public** **long** getScheduleTime(){  **long** start = System.*currentTimeMillis*();  code();  **long** end = System.*currentTimeMillis*();  **long** delta = end - start;//设置时间差  **return** delta;  }    **public** **abstract** **void** code();  }  **class** Test1 **extends** TimeTemplate{  @Override  **public** **void** code() {  **for**(**int** i=0;i<100000;i++){  System.***out***.println("我喜欢林心茹...");  }  }    } |

# 6 枚举

## 6.1 概述

1.枚举是指将变量的值一一列出来,可以称为『数据集』 。

举例：一周只有7天，一年只有12个月，一年有四个季节等。

2.Java中enum通过声明的类称为枚举类

3.枚举其实就是限定范围，防止不应该发生的事情发生

4.枚举注意事项

定义枚举类要用关键字enum

所有枚举类都是Enum的子类

枚举类的第一行上必须是枚举项，最后一个枚举项后的分号是可以省略的，但是如果枚举类有其他的东西，这个分号就不能省略。建议不要省略

枚举类可以有构造器，但必须是private的，它默认的也是private的。

枚举类也可以有抽象方法，但是枚举项必须重写该方法

枚举在switch语句中的使用

1. 枚举是一个特殊类

## 6.2 枚举的声明

|  |
| --- |
| //一年有四个季节  **enum** Season{  //春夏秋冬  ***SPRING***("春天"){  @Override  **public** **void** test() {  }  },  ***SUMMER***("夏天"){  @Override  **public** **void** test() {  // **TODO** Auto-generated method stub  System.***out***.println("夏天夏天夏天悄悄过去...");    }  },  ***AUTUMN***("秋天"){  @Override  **public** **void** test() {  // **TODO** Auto-generated method stub    }  },  ***WINTER***("冬天"){  @Override  **public** **void** test() {  // **TODO** Auto-generated method stub    }  };    /\*Season(){  System.out.println("空参的构造方法");  }\*/    **private** String s;  **private** Season(String s){  **this**.s = s;  }    **public** **abstract** **void** test();  } |

## 6.3 枚举的常见方法

* int ordinal() 枚举项都有索引，从0开始
* int compareTo(E o)
* String name() 枚举项名称
* String toString()
* <T> T valueOf(Class<T> type,String name)通过字节码对象获取枚举对象
* values() 此方法虽然在JDK文档中查找不到，但每个枚举类都具有该方法，它遍历枚举类的所有枚举值非常方便

|  |
| --- |
| **public** **class** Demo01 {  **public** **static** **void** main(String[] args) {  //枚举类的常见方法  //1.获取枚举对象  Season s1 = Season.***SPRING***;  Season s2 = Season.***WINTER***;  System.***out***.println("s1:" + s1.ordinal());  System.***out***.println("s2:" + s2.ordinal());    //2.比较枚举[索引相减]  System.***out***.println(s1.compareTo(s2));    //3.打印枚举项名称  System.***out***.println("s1 name:" + s1.name());  System.***out***.println("s1:" + s1.toString());//枚举项名称    //4.通过字节码对象获取枚举对象【没啥用】  Season s3 = Season.*valueOf*(Season.**class**, "SPRING");  System.***out***.println("s3:" + s3);    //5.类方法，.values() 返回枚举数组  System.***out***.println("枚举的遍历");  Season[] seasons = Season.*values*();  **for**(Season s : seasons){  System.***out***.println(s);  }  }  }  **enum** Season{  ***SPRING***,***SUMMER***,***AUTUMN***,***WINTER***;  } |

# 7 JDK新特性

## 7.1 JDK7的新特性

|  |
| --- |
| **public** **static** **void** main(String[] args) **throws** Exception {  //JDK7的六个新特性回顾和讲解  // A:二进制字面量  System.***out***.println(0b1000);    // B:数字字面量可以出现下划线  System.***out***.println(1000\_0000);  // C:switch 语句可以用字符串  String s = "A";  **switch** (s) {  **case** "A":    **break**;  **case** "B":    **break**;  **default**:  **break**;  }    // D:泛型简化,菱形泛型  //List<String> list = new ArrayList<String>();  List<String> list = **new** ArrayList<>();    // E:异常的多个catch合并,每个异常用或|  **try**{  **int** i = 10 / 0;//算术异常  **int**[] arr = **null**;  System.***out***.println(arr[0]);//空指针异常  }**catch** (ArithmeticException | NullPointerException e) {  // **TODO**: handle exception  }    // F:try-with-resources 语句 //这样写不用关流  **try**(  FileInputStream fis = **new** FileInputStream("a.txt");  ){  fis.read();  }  } |

## 7.2 JDK8的新特性

* 接口中可以定义有方法体的方法
* 如果是非静态,必须用default修饰
* 如果是静态的就不用了

|  |
| --- |
| **interface** Animal{  //接口只能声明方法，不能实现,就是不能有方法体  //public void eat();      //jdk1.8可以有方法体  **public** **default** **void** eat(){  System.***out***.println("吃饭...");  }    **public** **static** **void** sleep(){  System.***out***.println("睡.....");  }  }  **class** Dog **implements** Animal{    } |