# 反射机制

## 什么是Java反射

就是正在运行，动态获取这个类的所有信息。

## 反射机制的作用

  1，反编译：.class-->.java

   2．通过反射机制访问java对象的属性，方法，构造方法等；

## 反射机制的应用场景

Jdbc 加载驱动-----

Spring IOC

框架

## 反射机制获取类有三种方法

|  |
| --- |
| **//第一种方式：**  **Classc1 = Class.*forName*("Employee");**  **//第二种方式：**  **//java中每个类型都有class 属性.**  **Classc2 = Employee.class;**    **//第三种方式：**  **//java语言中任何一个java对象都有getClass 方法**  **Employeee = new Employee();**  **Classc3 = e.getClass(); //c3是运行时类 (e的运行时类是Employee)** |

## 反射创建对象的方式

|  |
| --- |
| **Class<?> forName = Class.*forName*("com.itmayiedu.entity.User");**  **// 创建此Class 对象所表示的类的一个新实例 调用了User的无参数构造方法.**  **Object newInstance = forName.newInstance();** |

实例化有参构造函数

|  |
| --- |
| **Class<?> forName = Class.*forName*("com.itmayiedu.entity.User");**  **Constructor<?> constructor = forName.getConstructor(String.class, String.class);**  **User newInstance = (User) constructor.newInstance("123", "123");** |

## 反射创建api

|  |  |
| --- | --- |
| 方法名称 | 作用 |
| getDeclaredMethods [] | 获取该类的所有方法 |
| getReturnType() | 获取该类的返回值 |
| getParameterTypes() | 获取传入参数 |
| getDeclaredFields() | 获取该类的所有字段 |
| setAccessible | 允许访问私有成员 |

## 使用反射为类私有属性赋值

|  |
| --- |
| **// 为user对象私有属性赋值**  **Class<?> classUser = Class.*forName*("com.itmayiedu.entity.User");**  **// 获取到当前的所有属性**  **Field[] fields = classUser.getDeclaredFields();**  **for (Field field : fields) {**  **System.*out*.println(field.getName());**  **}**  **// 获取当前所有的方法**  **Method[] declaredMethods = classUser.getDeclaredMethods();**  **for (Method method : declaredMethods) {**  **System.*out*.println(method.getName());**  **}**  **// 初始化对象**  **User user = (User) classUser.newInstance();**  **Field declaredField = classUser.getDeclaredField("id");**  **// 标记为true 允许反射赋值**  **declaredField.setAccessible(true);**  **declaredField.set(user, "20");**  **System.*out*.println("使用反射机制给id赋值为:"+user.getId());** |

## JDBC反射加载驱动

|  |
| --- |
| **public class DemoJDBC {**  **public static void main(String[] args) throws Exception {**  **// 加载驱动类**  **Class.*forName*("com.mysql.jdbc.Driver");**    **// 通过DriverManager获取数据库连接**  **String url = "jdbc:mysql://192.168.1.150/test";**  **String user = "teamtalk";**  **String password = "123456";**  **Connection connection = (Connection) DriverManager.*getConnection*(**  **url, user, password);**    **PreparedStatement statement = (PreparedStatement) connection.prepareStatement(**  **"insert persion (name, age) value (?, ?)");**  **statement.setString(1, "hdu");**  **statement.setInt(2, 21);**  **statement.executeUpdate();**    **ResultSet resultSet = statement.executeQuery("select \* from persion");**  **// 操作ResultSet结果集**  **while (resultSet.next()) {**  **// 第一种获取字段方式**  **System.*out*.println(resultSet.getString(1) + " " +**  **resultSet.getString(2) + " " + resultSet.getString(3));**  **}**  **// 关闭数据库连接**  **resultSet.close();**  **statement.close();**  **connection.close();**  **}**  **}** |

## 禁止使用反射机制初始化

将构造函数为私有化

# 设计模式

## 设计模式分类

创建型模式，共五种：**工厂方法模式、抽象工厂模式**、**单例模式**、建造者模式、**原型模式。**

结构型模式，共七种：适配器模式、装饰器模式、代理模式、外观模式、桥接模式、组合模式、享元模式。

行为型模式，共十一种：策略模式、模板方法模式、观察者模式、迭代子模式、责任链模式、命令模式、备忘录模式、状态模式、访问者模式、中介者模式、解释器模式。

## 设计模式的六大原则

### 开闭原则（Open Close Principle）

开闭原则就是说对扩展开放，对修改关闭。在程序需要进行拓展的时候，不能去修改原有的代码，实现一个热插拔的效果。所以一句话概括就是：为了使程序的扩展性好，易于维护和升级。想要达到这样的效果，我们需要使用接口和抽象类，后面的具体设计中我们会提到这点。

### 里氏代换原则（Liskov Substitution Principle）

里氏代换原则(Liskov Substitution Principle LSP)面向对象设计的基本原则之一。 里氏代换原则中说，任何基类可以出现的地方，子类一定可以出现。 LSP是继承复用的基石，只有当衍生类可以替换掉基类，软件单位的功能不受到影响时，基类才能真正被复用，而衍生类也能够在基类的基础上增加新的行为。里氏代换原则是对“开-闭”原则的补充。实现“开-闭”原则的关键步骤就是抽象化。而基类与子类的继承关系就是抽象化的具体实现，所以里氏代换原则是对实现抽象化的具体步骤的规范。—— From Baidu 百科

### 依赖倒转原则（Dependence Inversion Principle）

这个是开闭原则的基础，具体内容：真对接口编程，依赖于抽象而不依赖于具体。

接口隔离原则（Interface Segregation Principle）

这个原则的意思是：使用多个隔离的接口，比使用单个接口要好。还是一个降低类之间的耦合度的意思，从这儿我们看出，其实设计模式就是一个软件的设计思想，从大型软件架构出发，为了升级和维护方便。所以上文中多次出现：降低依赖，降低耦合。

### 迪米特法则（最少知道原则）（Demeter Principle）

为什么叫最少知道原则，就是说：一个实体应当尽量少的与其他实体之间发生相互作用，使得系统功能模块相对独立。

### 合成复用原则（Composite Reuse Principle）

原则是尽量使用合成/聚合的方式，而不是使用继承。

# 单例模式

## 什么是单例

保证一个类只有一个实例，并且提供一个访问该全局访问点

## 单例应用场景

1. Windows的Task Manager（任务管理器）就是很典型的单例模式（这个很熟悉吧），想想看，是不是呢，你能打开两个windows task manager吗？ 不信你自己试试看哦~

2. windows的Recycle Bin（回收站）也是典型的单例应用。在整个系统运行过程中，回收站一直维护着仅有的一个实例。

3. 网站的计数器，一般也是采用单例模式实现，否则难以同步。

4. 应用程序的日志应用，一般都何用单例模式实现，这一般是由于共享的日志文件一直处于打开状态，因为只能有一个实例去操作，否则内容不好追加。

5. Web应用的配置对象的读取，一般也应用单例模式，这个是由于配置文件是共享的资源。

6. 数据库连接池的设计一般也是采用单例模式，因为数据库连接是一种数据库资源。数据库软件系统中使用数据库连接池，主要是节省打开或者关闭数据库连接所引起的效率损耗，这种效率上的损耗还是非常昂贵的，因为何用单例模式来维护，就可以大大降低这种损耗。

7. 多线程的线程池的设计一般也是采用单例模式，这是由于线程池要方便对池中的线程进行控制。

8. 操作系统的文件系统，也是大的单例模式实现的具体例子，一个操作系统只能有一个文件系统。

9. HttpApplication 也是单位例的典型应用。熟悉ASP.Net(IIS)的整个请求生命周期的人应该知道HttpApplication也是单例模式，所有的HttpModule都共享一个HttpApplication实例.

## 单例优缺点

优点：

1.在单例模式中，活动的单例只有一个实例，对单例类的所有实例化得到的都是相同的一个实例。这样就 防止其它对象对自己的实例化，确保所有的对象都访问一个实例

2.单例模式具有一定的伸缩性，类自己来控制实例化进程，类就在改变实例化进程上有相应的伸缩性。

3.提供了对唯一实例的受控访问。

4.由于在系统内存中只存在一个对象，因此可以 节约系统资源，当 需要频繁创建和销毁的对象时单例模式无疑可以提高系统的性能。

5.允许可变数目的实例。

6.避免对共享资源的多重占用。

缺点：

1.不适用于变化的对象，如果同一类型的对象总是要在不同的用例场景发生变化，单例就会引起数据的错误，不能保存彼此的状态。

2.由于单利模式中没有抽象层，因此单例类的扩展有很大的困难。

3.单例类的职责过重，在一定程度上违背了“单一职责原则”。

4.滥用单例将带来一些负面问题，如为了节省资源将数据库连接池对象设计为的单例类，可能会导致共享连接池对象的程序过多而出现连接池溢出；如果实例化的对象长时间不被利用，系统会认为是垃圾而被回收，这将导致对象状态的丢失。

## 单例创建方式

1. **饿汉式:类初始化时,会立即加载该对象，线程天生安全,调用效率高。**
2. **懒汉式: 类初始化时,不会初始化该对象,真正需要使用的时候才会创建该对象,具备懒加载功能。**
3. 静态内部方式:结合了懒汉式和饿汉式各自的优点，真正需要对象的时候才会加载，加载类是线程安全的。
4. 枚举单例: 使用枚举实现单例模式 优点:实现简单、调用效率高，枚举本身就是单例，由jvm从根本上提供保障!避免通过反射和反序列化的漏洞， 缺点没有延迟加载。
5. 双重检测锁方式 (因为JVM本质重排序的原因，可能会初始化多次，不推荐使用)

### 饿汉式

|  |
| --- |
| //饿汉式  **public** **class** SingletonDemo01 {  // 类初始化时,会立即加载该对象，线程天生安全,调用效率高  **private** **static** SingletonDemo01 *singletonDemo01* = **new** SingletonDemo01();  **private** SingletonDemo01() {  System.***out***.println("SingletonDemo01初始化");  }  **public** **static** SingletonDemo01 getInstance() {  System.***out***.println("getInstance");  **return** *singletonDemo01*;  }  **public** **static** **void** main(String[] args) {  SingletonDemo01 s1 = SingletonDemo01.*getInstance*();  SingletonDemo01 s2 = SingletonDemo01.*getInstance*();  System.***out***.println(s1 == s2);  }  } |

### 懒汉式

|  |
| --- |
| //懒汉式  **public** **class** SingletonDemo02 {  //类初始化时，不会初始化该对象，真正需要使用的时候才会创建该对象。  **private** **static** SingletonDemo02 *singletonDemo02*;  **private** SingletonDemo02() {    }  **public** **synchronized** **static** SingletonDemo02 getInstance() {  **if** (*singletonDemo02* == **null**) {  *singletonDemo02* = **new** SingletonDemo02();  }  **return** *singletonDemo02*;  }  **public** **static** **void** main(String[] args) {  SingletonDemo02 s1 = SingletonDemo02.*getInstance*();  SingletonDemo02 s2 = SingletonDemo02.*getInstance*();  System.***out***.println(s1 == s2);  }  } |

### 静态内部类

|  |
| --- |
| // 静态内部类方式  **public** **class** SingletonDemo03 {  **private** SingletonDemo03() {  System.***out***.println("初始化..");  }  **public** **static** **class** SingletonClassInstance {  **private** **static** **final** SingletonDemo03 ***singletonDemo03*** = **new** SingletonDemo03();  }  // 方法没有同步  **public** **static** SingletonDemo03 getInstance() {  System.***out***.println("getInstance");  **return** SingletonClassInstance.***singletonDemo03***;  }  **public** **static** **void** main(String[] args) {  SingletonDemo03 s1 = SingletonDemo03.*getInstance*();  SingletonDemo03 s2 = SingletonDemo03.*getInstance*();  System.***out***.println(s1 == s2);  }  } |

　优势：兼顾了懒汉模式的内存优化（使用时才初始化）以及饿汉模式的安全性（不会被反射入侵）。

　劣势：需要两个类去做到这一点，虽然不会创建静态内部类的对象，但是其 Class 对象还是会被创建，而且是属于永久带的对象。

### 枚举方式

#### 什么是枚举

枚举本身是单例的，一般用于项目中定义常量。

|  |
| --- |
| **enum** UserEnum {  ***HTTP\_200***(200, "请求成功"),***HTTP\_500***(500,"请求失败");  **private** Integer code;  **private** String name;  UserEnum(Integer code, String name) {  **this**.code = code;  **this**.name = name;  }  **public** Integer getCode() {  **return** code;  }  **public** **void** setCode(Integer code) {  **this**.code = code;  }  **public** String getName() {  **return** name;  }  **public** **void** setName(String name) {  **this**.name = name;  }  }  **public** **class** TestEnum {  **public** **static** **void** main(String[] args) {  System.***out***.println(UserEnum.***HTTP\_500***.getCode());  }  } |

|  |
| --- |
| /使用枚举实现单例模式 优点:实现简单、枚举本身就是单例，由jvm从根本上提供保障!避免通过反射和反序列化的漏洞 缺点没有延迟加载  **public** **class** User {  **public** **static** User getInstance() {  **return** SingletonDemo04.***INSTANCE***.getInstance();  }  **private** **static** **enum** SingletonDemo04 {  ***INSTANCE***;  // 枚举元素为单例  **private** User user;  **private** SingletonDemo04() {  System.***out***.println("SingletonDemo04");  user = **new** User();  }  **public** User getInstance() {  **return** user;  }  }  **public** **static** **void** main(String[] args) {  User u1 = User.*getInstance*();  User u2 = User.*getInstance*();  System.***out***.println(u1 == u2);  }  } |

### 双重检测锁

|  |
| --- |
| **public** **class** SingletonDemo04 {  **private** SingletonDemo04 singletonDemo04;  **private** SingletonDemo04() {  }  **public** SingletonDemo04 getInstance() {  **if** (singletonDemo04 == **null**) {  **synchronized** (**this**) {  **if** (singletonDemo04 == **null**) {  singletonDemo04 = **new** SingletonDemo04();  }  }  }  **return** singletonDemo04;  }  } |

### 单例防止反射漏洞攻击

在构造函数中，只能允许初始化化一次即可。

|  |
| --- |
| **private** **static** **boolean** *flag* = **false**;  **private** SingletonDemo04() {  **if** (*flag* == **false**) {  *flag* = !*flag*;  } **else** {  **throw** **new** RuntimeException("单例模式被侵犯！");  }  }  **public** **static** **void** main(String[] args) {  } |

### 如何选择单例创建方式

如果不需要延迟加载单例，可以使用枚举或者饿汉式，相对来说枚举性好于饿汉式。

如果需要延迟加载，可以使用静态内部类或者懒韩式，相对来说静态内部类好于懒韩式。

# UML使用

# 工厂模式

## 什么是工厂模式

实现了创建者和调用者分离，工厂模式分为简单工厂、工厂方法、抽象工厂模式

## 工厂模式好处

工厂模式是我们最常用的实例化对象模式了，是用工厂方法代替new操作的一种模式。

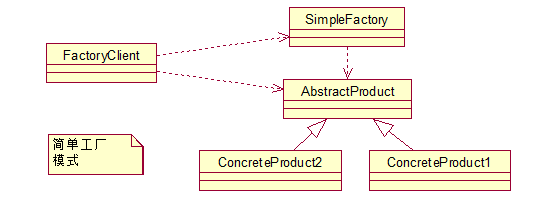
利用工厂模式可以降低程序的耦合性，为后期的维护修改提供了很大的便利。

将选择实现类、创建对象统一管理和控制。从而将调用者跟我们的实现类解耦。

## 工厂模式分类

### 简单工厂模式

简单工厂模式相当于是一个工厂中有各种产品，创建在一个类中，客户无需知道具体产品的名称，只需要知道产品类所对应的参数即可。但是工厂的职责过重，而且当类型过多时不利于系统的扩展维护。



|  |
| --- |
| **public** **interface** Car {  **public** **void** run();  }  **public** **class** AoDi **implements** Car {  **public** **void** run() {  System.***out***.println("我是奥迪汽车..");  }  }  **public** **class** JiLi **implements** Car {  **public** **void** run() {  System.***out***.println("我是吉利汽车...");  }  }  **public** **class** CarFactory {  **public** **static** Car createCar(String name) {  **if** (StringUtils.*isEmpty*(name)) {  **return** **null**;  }  **if**(name.equals("奥迪")){  **return** **new** AoDi();  }  **if**(name.equals("吉利")){  **return** **new** JiLi();  }  **return** **null**;  }  }  **public** **class** Client01 {  **public** **static** **void** main(String[] args) {  Car aodi =CarFactory.*createCar*("奥迪");  Car jili =CarFactory.*createCar*("吉利");  aodi.run();  jili.run();  }  } |

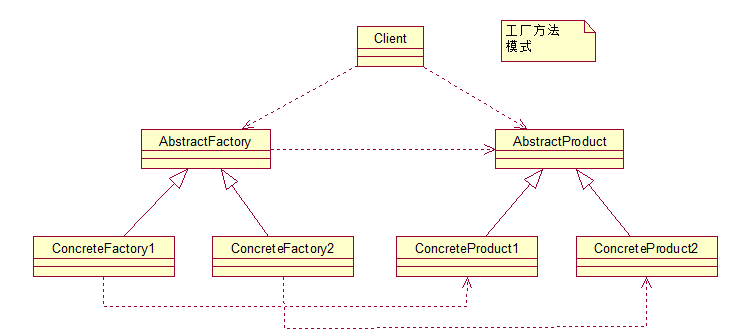
#### 单工厂的优点/缺点

优点：简单工厂模式能够根据外界给定的信息，决定究竟应该创建哪个具体类的对象。明确区分了各自的职责和权力，有利于整个软件体系结构的优化。

缺点：很明显工厂类集中了所有实例的创建逻辑，容易违反GRASPR的高内聚的责任分配原则

### 工厂方法模式

### 什么是工厂方法模式



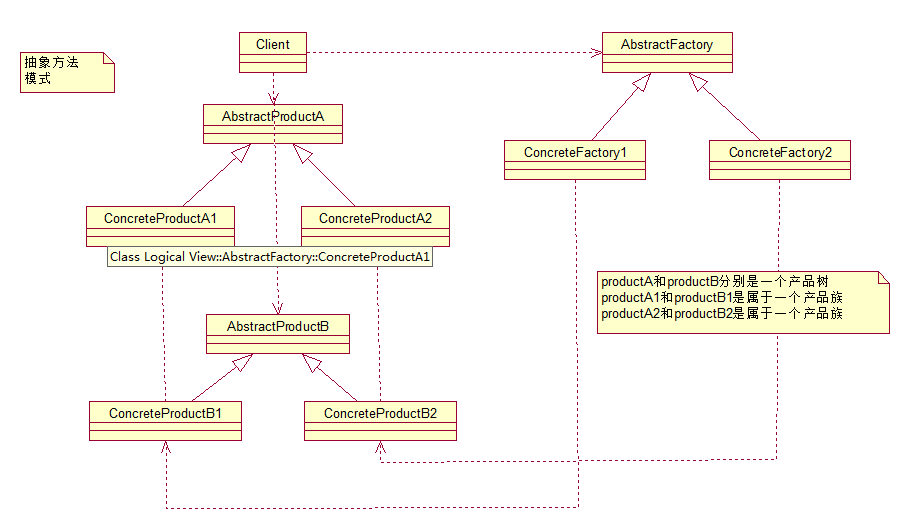
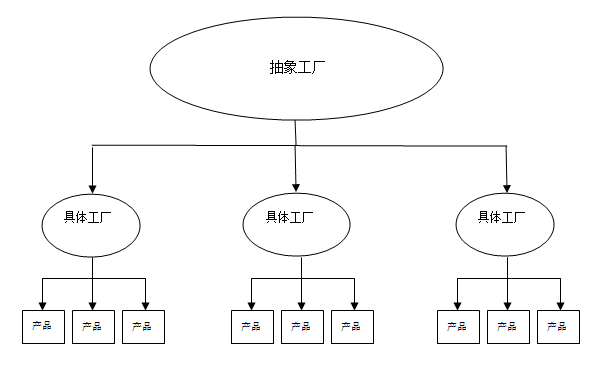
工厂方法模式Factory Method，又称多态性工厂模式。在工厂方法模式中，核心的工厂类不再负责所有的产品的创建，而是将具体创建的工作交给子类去做。该核心类成为一个抽象工厂角色，仅负责给出具体工厂子类必须实现的接口，而不接触哪一个产品类应当被实例化这种细节。

|  |
| --- |
| **public** **interface** Car {  **public** **void** run();  }  **public** **class** AoDi **implements** Car {  **public** **void** run() {  System.***out***.println("我是奥迪汽车..");  }  }  **public** **class** JiLi **implements** Car {  **public** **void** run() {  System.***out***.println("我是吉利汽车...");  }  }  **public** **class** JiLiFactory **implements** CarFactory {  **public** Car createCar() {  **return** **new** JiLi();  }  }  **public** **class** AoDiFactory **implements** CarFactory {  **public** Car createCar() {    **return** **new** AoDi();  }  }  **public** **class** Client {  **public** **static** **void** main(String[] args) {  Car aodi = **new** AoDiFactory().createCar();  Car jili = **new** JiLiFactory().createCar();  aodi.run();  jili.run();  }  } |

### 抽象工厂模式

#### 什么是抽象工厂模式

抽象工厂简单地说是工厂的工厂，抽象工厂可以创建具体工厂，由具体工厂来产生具体产品。



|  |
| --- |
| //发动机  **public** **interface** Engine {  **void** run();  **void** start();  }  **class** EngineA **implements** Engine {  **public** **void** run() {  System.***out***.println("转的快!");  }  **public** **void** start() {  System.***out***.println("启动快,自动档");  }  }  **class** EngineB **implements** Engine {  **public** **void** run() {  System.***out***.println("转的慢!");  }  **public** **void** start() {  System.***out***.println("启动快,手动档");  }  }  //座椅  **public** **interface** Chair {  **void** run();  }  **class** ChairA **implements** Chair{  **public** **void** run() {  System.***out***.println("可以自动加热!");  }    }  **class** ChairB **implements** Chair{  **public** **void** run() {  System.***out***.println("不能加热!");  }    }  **public** **interface** CarFactory {  // 创建发动机  Engine createEngine();  // 创建座椅  Chair createChair();  }  **public** **class** JiLiFactory **implements** CarFactory {  **public** Engine createEngine() {    **return** **new** EngineA();  }  **public** Chair createChair() {    **return** **new** ChairA();  }  }  **public** **class** Client002 {  **public** **static** **void** main(String[] args) {  CarFactory carFactory=**new** JiLiFactory();  Engine engine=carFactory.createEngine();  engine.run();  engine.start();  }    } |

# 建造者模式

# 模板方法

# 原型模式

# 适配模式

# 装饰模式

# 代理模式

# 外观模式

# 策略模式

# 观察者模式