自定义注解与设计模式

# 课程目标

**熟悉注解底层实现原理**

**完成ORM框架底层原理**

**常用设计模式**

**单例、工厂、代理**

# 自定义注解

## 1.1什么是注解？

Jdk1.5新增新技术，注解。很多框架为了简化代码，都会提供有些注解。可以理解为插件，是代码级别的插件，在类的方法上写：@XXX，就是在代码上插入了一个插件。

注解不会也不能影响代码的实际逻辑，仅仅起到辅助性的作用。

**注解分类：内置注解(也成为元注解 jdk 自带注解)、自定义注解（Spring框架）**

## 1.2 什么是内置注解

 比如

（1） @SuppressWarnings   再程序前面加上可以在javac编译中去除警告--阶段是SOURCE  
（2） @Deprecated   带有标记的包，方法，字段说明其过时----阶段是SOURCE  
（3）@Overricle   打上这个标记说明该方法是将父类的方法重写--阶段是SOURCE

### 1.1 @Overricle 案例演示

|  |
| --- |
| **@Override**  **public String toString() {**  **return null;**  **}** |

### 1.2 @ Deprecated案例演示

|  |
| --- |
| **new Date().parse("");** |

### 1.3 @ SuppressWarnings  案例演示

|  |
| --- |
| @SuppressWarnings({ "all" })  **public** **void** save() {  java.util.List list = **new** ArrayList();  } |

## 1.3 实现自定义注解

元注解的作用就是负责注解其他注解。Java5.0定义了4个标准的meta-annotation类型，它们被用来提供对其它 annotation类型作说明。Java5.0定义的元注解：  
[**1.@Target**](mailto:1.@Target)

@Target说明了Annotation所修饰的对象范围：Annotation可被用于 packages、types（类、接口、枚举、Annotation类型）、类型成员（方法、构造方法、成员变量、枚举值）、方法参数和本地变量（如循环变量、catch参数）。在Annotation类型的声明中使用了target可更加明晰其修饰的目标。

1. CONSTRUCTOR:用于描述构造器
2. FIELD:用于描述域
3. LOCAL\_VARIABLE:用于描述局部变量
4. METHOD:用于描述方法
5. PACKAGE:用于描述包
6. PARAMETER:用于描述参数
7. TYPE:用于描述类、接口(包括注解类型) 或enum声明

[**2.@Retention**](mailto:2.@Retention)

**表示需要在什么级别保存该注释信息，用于描述注解的生命周期（即：被描述的注解在什么范围内有效）**  
**3.@Documented  
4.@Inherited**

代码:

使用@interface 定义注解。

|  |
| --- |
| @Target(value = { ElementType.***METHOD***, ElementType.***TYPE*** })  @Retention(RetentionPolicy.***RUNTIME***)  **public** **@interface** OneAnnotation {  **int** beanId() **default** 0;  String className() **default** "";  String[]arrays();  } |

**使用:**

|  |
| --- |
| @OneAnnotation(beanId = 123, className = "className", arrays = { "111", "222" })  **public** **void** add() {  } |

## 1.3 实现ORM框架映射

完成案例，ORM框架实体类与表字段不一致,底层生成sql语句原理。

### 1.3.1自定义表映射注解

|  |
| --- |
| /\*\*  \*  \* **@classDesc**: 功能描述:(**自定义表映射注解** )  \* **@author**: 余胜军  \* **@createTime**: 2017年8月27日 上午12:09:53  \* **@version**: v1.0  \* **@copyright**:上海每特教育科技有限公司  \*/  @Target(value = { ElementType.***TYPE*** })  @Retention(RetentionPolicy.***RUNTIME***)  **public** **@interface** SetTable {  /\*\*  \*  \* **@methodDesc**: 功能描述:(对应数据库表名称)  \* **@author**: 余胜军  \* **@param**: **@return**  \* **@createTime**:2017年8月27日 上午12:10:49  \* **@returnType**:@return String  \* **@copyright**:上海每特教育科技有限公司  \*/  String value();  } |

### 1.3.2自定义字段属性

|  |
| --- |
| **/\*\***  **\***  **\* @classDesc: 功能描述:(定义字段属性)**  **\* @author: 余胜军**  **\* @createTime: 2017年8月27日 上午12:13:32**  **\* @version: v1.0**  **\* @copyright:上海每特教育科技有限公司**  **\*/**  **@Retention(RetentionPolicy.*RUNTIME*)**  **public @interface SetProperty {**  **/\*\***  **\***  **\* @methodDesc: 功能描述:(字段名称)**  **\* @author: 余胜军**  **\* @param: @return**  **\* @createTime:2017年8月27日 上午12:14:02**  **\* @returnType:@return String**  **\* @copyright:上海每特教育科技有限公司**  **\*/**  **String name();**  **/\*\***  **\***  **\* @methodDesc: 功能描述:(长度)**  **\* @author: 余胜军**  **\* @param: @return**  **\* @createTime:2017年8月27日 上午12:14:25**  **\* @returnType:@return int**  **\* @copyright:上海每特教育科技有限公司**  **\*/**  **int leng();**  **}** |

### 1.3.3自定义注解代码实现

|  |
| --- |
| **/\*\***  **\***  **\* @classDesc: 功能描述:(使用java自定义注解 模拟ORM框架注解版本 )**  **\* @author: 余胜军**  **\* @createTime: 2017年8月27日 上午12:41:52**  **\* @version: v1.0**  **\* @copyright:上海每特教育科技有限公司**  **\*/**  **public class Main {**  **public static void main(String[] args) throws ClassNotFoundException {**  **// 1.反射class**  **Class<?> classForName = Class.*forName*("com.entity.Sudent");**  **// 2.获取表名称注解F**  **SetTable setTable = classForName.getAnnotation(SetTable.class);**  **// 3.获取所有的成员属性**  **Field[] declaredFields = classForName.getDeclaredFields();**  **StringBuffer sf = new StringBuffer();**  **sf.append(" select ");**  **String fromName = setTable.value();**  **for (int i = 0; i < declaredFields.length; i++) {**  **Field field = declaredFields[i];**  **// 4.属性字段**  **SetProperty sb = field.getAnnotation(SetProperty.class);**  **sf.append(" " + sb.name() + " ");**  **if (i == declaredFields.length - 1) {**  **sf.append(" from ");**  **} else {**  **sf.append(" , ");**  **}**  **}**  **sf.append(" " + fromName);**  **System.*out*.println(sf.toString());**  **}**  **}** |

# 常用设计模式

## 2.1 什么是设计模式？

**设计模式（Design pattern）是一套被反复使用、多数人知晓的、经过分类编目的、代码设计经验的总结。使用设计模式是为了可重用代码、让代码更容易被他人理解、保证代码可靠性。 毫无疑问，设计模式于己于他人于系统都是多赢的，设计模式使代码编制真正工程化，设计模式是软件工程的基石，如同大厦的一块块砖石一样。项目中合理的运用设计模式可以完美的解决很多问题，每种模式在现在中都有相应的原理来与之对应，每一个模式描述了一个在我们周围不断重复发生的问题，以及该问题的核心解决方案，这也是它能被广泛应用的原因。本章系Java之美[从菜鸟到高手演变]系列之设计模式，我们会以理论与实践相结合的方式来进行本章的学习，希望广大程序爱好者，学好设计模式，做一个优秀的软件工程师！**

## 2.2 设计模式的分类？

总体来说设计模式分为三大类：

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。

其实还有两类：并发型模式和线程池模式。用一个图片来整体描述一下：



## 2.3 设计模式的六大原则

**1、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。

**2、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

**3、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。

**4、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。

**5、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。

**6、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。

### 2.1 单例模式

#### 2.1.1什么是单例模式？

单例保证一个对象JVM中只能有一个实例,常见单例 懒汉式、饿汉式

什么是懒汉式,就是需要的才会去实例化,线程不安全。

什么是饿汉式,就是当class文件被加载的时候，初始化，天生线程安全。

#### 2.1.2单例写法

**懒汉式代码**

|  |
| --- |
| **class SingletonTest {**  **public static void main(String[] args) {**  **Singleton sl1 = Singleton.*getSingleton*();**  **Singleton sl2 = Singleton.*getSingleton*();**  **System.*out*.println(sl1 == sl2);**  **}**  **}**  **public class Singleton {**  **// 当需要的才会被实例化**  **private static Singleton *singleton*;**  **private Singleton() {**  **}**  **synchronized public static Singleton getSingleton() {**  **if (*singleton* == null) {**  ***singleton* = new Singleton();**  **}**  **return *singleton*;**  **}**  **}** |

**饿汉式代码**

|  |
| --- |
| **class SingletonTest1 {**  **public static void main(String[] args) {**  **Singleton1 sl1 = Singleton1.*getSingleton*();**  **Singleton1 sl2 = Singleton1.*getSingleton*();**  **System.*out*.println((sl1 == sl2)+"-");**  **}**  **}**  **public class Singleton1 {**  **//当class 文件被加载初始化**  **private static Singleton1 *singleton* = new Singleton1();**  **private Singleton1() {**  **}**  **public static Singleton1 getSingleton() {**  **return *singleton*;**  **}**  **}** |

## 2.4 工厂模式

### 2.4.1什么是工厂模式？

**实现创建者和调用者分离**

### 2.4.2简单工厂代码

|  |
| --- |
| **public interface Car {**  **public void run();**  **}**  **public class AoDi implements Car {**  **@Override**  **public void run() {**  **System.*out*.println("奥迪....");**  **}**  **}**  **public interface Car {**  **public void run();**  **}** |

|  |
| --- |
| **public** **class** CarFactory {  **static** **public** Car createCar(String carName) {  Car car = **null**;  **if** (carName.equals("奥迪")) {  car = **new** AoDi();  } **else** **if** (carName.equals("奔驰")) {  car = **new** BenChi();  }  **return** car;  }  **public** **static** **void** main(String[] args) {  Car car1 = CarFactory.*createCar*("奥迪");  Car car2 = CarFactory.*createCar*("奔驰");  car1.run();  car2.run();  }  } |

### 2.4.3工厂方法

|  |
| --- |
| public interface Car {  public void run();  }  public class AoDi implements Car {  @Override  public void run() {  System.*out*.println("奥迪....");  }  }  public class BenChi implements Car {  @Override  public void run() {  System.*out*.println("奔驰....");  }  } |

|  |
| --- |
| **public class AoDiChiFactory {**  **static public Car createCar() {**  **return new AoDi();**  **}**  **}**  **public interface BenChiFactory {**  **static public Car createCar() {**  **return new BenChi();**  **}**  **}**  **public class Main {**  **public static void main(String[] args) {**  **Car c1 = AoDiChiFactory.*createCar*();**  **Car c2 = BenChiFactory.*createCar*();**  **c1.run();**  **c2.run();**  **}**  **}** |

## 2.5 代理模式

## 2.5.1什么是代理？

通过代理控制对象的访问,可以详细访问某个对象的方法，在这个方法调用处理，或调用后处理。既(AOP微实现) ,AOP核心技术面向切面编程。



## 2.5.1代理应用场景

安全代理 可以屏蔽真实角色

远程代理 远程调用代理类RMI

延迟加载 先加载轻量级代理类,真正需要在加载真实

## 2.5.2代理的分类

静态代理(静态定义代理类)

动态代理(动态生成代理类)

Jdk自带动态代理

Cglib 、javaassist（字节码操作库）

## 2.5.3 静态代理

静态代理需要自己生成代理类

|  |
| --- |
| **public** **class** XiaoMing **implements** Hose {  @Override  **public** **void** mai() {  System.***out***.println("我是小明,我要买房啦!!!!haha ");  }  }  **class** Proxy **implements** Hose {  **private** XiaoMing xiaoMing;  **public** Proxy(XiaoMing xiaoMing) {  **this**.xiaoMing = xiaoMing;  }  **public** **void** mai() {  System.***out***.println("我是中介 看你买房开始啦!");  xiaoMing.mai();  System.***out***.println("我是中介 看你买房结束啦!");  }  **public** **static** **void** main(String[] args) {  Hose proxy = **new** Proxy(**new** XiaoMing());  proxy.mai();  }  } |

## 2.5.4JDK动态代理(不需要生成代理类)

实现InvocationHandler 就可以了。

|  |
| --- |
| **public** **interface** Hose {  /\*\*  \*  \* **@methodDesc**: 功能描述:(买房代理)  \* **@author**: 余胜军  \* **@param**:  \* **@createTime**:2017年8月27日 上午2:54:34  \* **@returnType**: void  \* **@copyright**:上海每特教育科技有限公司  \*/  **public** **void** mai();  }  **public** **class** XiaoMing **implements** Hose {  @Override  **public** **void** mai() {  System.***out***.println("我是小明,我要买房啦!!!!haha ");  }  } |

|  |
| --- |
| **public** **class** JDKProxy **implements** InvocationHandler {  **private** Object tarjet;  **public** JDKProxy(Object tarjet) {  **this**.tarjet = tarjet;  }  @Override  **public** Object invoke(Object proxy, Method method, Object[] args) **throws** Throwable {  System.***out***.println("我是房产中介.....开始监听你买房啦!");  Object oj = method.invoke(tarjet, args);  System.***out***.println("我是房产中介.....结束监听你买房啦!");  **return** oj;  }  }  **class** Test222 {  **public** **static** **void** main(String[] args) {  XiaoMing xiaoMing = **new** XiaoMing();  JDKProxy jdkProxy = **new** JDKProxy(xiaoMing);  Hose hose=(Hose) Proxy.*newProxyInstance*(xiaoMing.getClass().getClassLoader(), xiaoMing.getClass().getInterfaces(), jdkProxy);  hose.mai();  }  } |

## 2.5.6 CGLIB动态代理

实现

|  |
| --- |
| **import java.lang.reflect.Method;**  **import net.sf.cglib.proxy.Enhancer;**  **import net.sf.cglib.proxy.MethodInterceptor;**  **import net.sf.cglib.proxy.MethodProxy;**  **public class Cglib implements MethodInterceptor {**  **@Override**  **public Object intercept(Object o, Method method, Object[] args, MethodProxy methodProxy) throws Throwable {**  **System.*out*.println("我是买房中介 ， 开始监听你买房了....");**  **Object invokeSuper = methodProxy.invokeSuper(o, args);**  **System.*out*.println("我是买房中介 ， 开结束你买房了....");**  **return invokeSuper;**  **}**  **}**  **class Test22222 {**  **public static void main(String[] args) {**  **Cglib cglib = new Cglib();**  **Enhancer enhancer = new Enhancer();**  **enhancer.setSuperclass(XiaoMing.class);**  **enhancer.setCallback(cglib);**  **Hose hose = (Hose) enhancer.create();**  **hose.mai();**  **}**  **}** |

## 2.5.7CGLIB与JDK动态代理区别

jdk动态代理是由[Java](http://lib.csdn.net/base/java" \t "_blank" \o "Java 知识库)内部的反射机制来实现的，cglib动态代理底层则是借助asm来实现的。总的来说，反射机制在生成类的过程中比较高效，而asm在生成类之后的相关执行过程中比较高效（可以通过将asm生成的类进行缓存，这样解决asm生成类过程低效问题）。还有一点必须注意：jdk动态代理的应用前提，必须是目标类基于统一的接口。如果没有上述前提，jdk动态代理不能应用。

注:asm其实就是java字节码控制.