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

## 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）**

## 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

### @Overricle 案例演示

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

### @ Deprecated案例演示

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

### @ SuppressWarnings  案例演示

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

## 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：  
[@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。

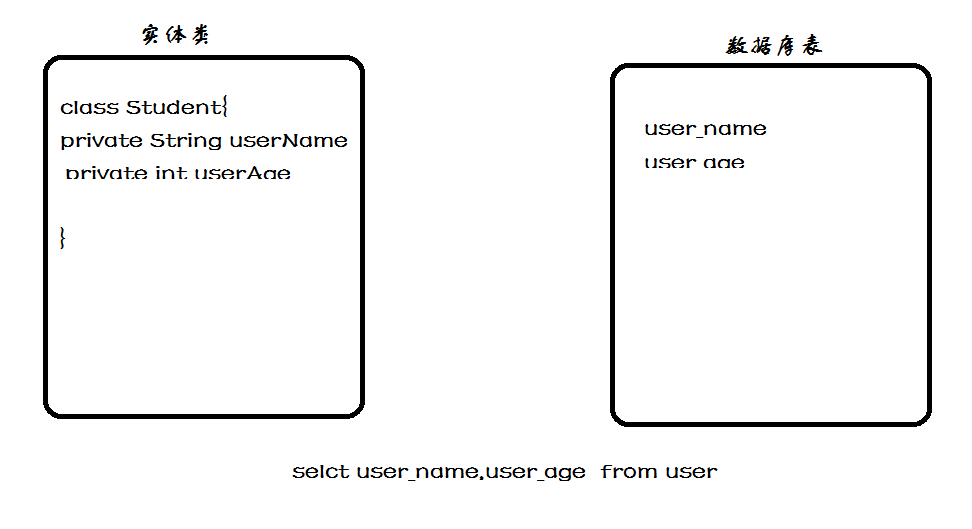
|  |
| --- |
| @Target(value = { ElementType.***METHOD***, ElementType.***TYPE*** })  @Retention(RetentionPolicy.***RUNTIME***)  **public** **@interface** OneAnnotation {  **int** beanId() **default** 0;  String className() **default** "";  String[]arrays();  } |

**使用:**

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

## 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。**



### 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();  } |

### 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();**  **}** |

### 完整代码实现

|  |
| --- |
| **@Table(value = "it\_user")**  **class User {**  **@ProPerty(value = "it\_id", leng = 10)**  **private String id;**  **@ProPerty(value = "it\_name", leng = 10)**  **private String name;**  **public String getId() {**  **return id;**  **}**  **public void setId(String id) {**  **this.id = id;**  **}**  **public String getName() {**  **return name;**  **}**  **public void setName(String name) {**  **this.name = name;**  **}**  **}**  **@Target(value = ElementType.*TYPE*)**  **@Retention(RetentionPolicy.*RUNTIME*)**  **@interface Table {**  **String value();**  **}**  **@Retention(RetentionPolicy.*RUNTIME*)**  **@interface ProPerty {**  **String value();**  **int leng();**  **}**  **public class Test002 {**  **public static void main(String[] args) throws ClassNotFoundException {**  **Class<?> forName = Class.*forName*("com.itmayiedu.User");**  **StringBuffer sf = new StringBuffer();**  **sf.append(" select ");**  **// 获取当前的所有的属性**  **Field[] declaredFields = forName.getDeclaredFields();**  **for (int i = 0; i < declaredFields.length; i++) {**  **Field field = declaredFields[i];**  **ProPerty proPertyAnnota = field.getDeclaredAnnotation(ProPerty.class);**  **String proPertyName = proPertyAnnota.value();**  **sf.append(" " + proPertyName);**  **if (i < declaredFields.length - 1) {**  **sf.append(" , ");**  **}**  **}**  **Table tableAnnota = forName.getDeclaredAnnotation(Table.class);**  **// 表的名称**  **String tableName = tableAnnota.value();**  **sf.append(" from " + tableName);**  **System.*out*.println(sf.toString());**  **}**  **}** |

# 常用设计模式

## 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！

## 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：



创建型模式  
前面讲过，社会化的分工越来越细，自然在软件设计方面也是如此，因此对象的创建和对象的使用分开也就成为了必然趋势。因为对象的创建会消耗掉系统的很多资源，所以单独对对象的创建进行研究，从而能够高效地创建对象就是创建型模式要探讨的问题。这里有6个具体的创建型模式可供研究，它们分别是：  
  
[简单工厂模式（Simple Factory）](http://blog.csdn.net/itpinpai/article/details/51163863)  
[工厂方法模式（Factory Method）](http://blog.csdn.net/itpinpai/article/details/51163863)  
[抽象工厂模式（Abstract Factory）](http://blog.csdn.net/itpinpai/article/details/51163863)  
创建者模式（Builder）  
原型模式（Prototype）  
[单例模式（Singleton）](http://blog.csdn.net/itpinpai/article/details/51355342)  
说明：严格来说，简单工厂模式不是GoF总结出来的23种设计模式之一。  
  
结构型模式  
在解决了对象的创建问题之后，对象的组成以及对象之间的依赖关系就成了开发人员关注的焦点，因为如何设计对象的结构、继承和依赖关系会影响到后续程序的维护性、代码的健壮性、耦合性等。对象结构的设计很容易体现出设计人员水平的高低，这里有7个具体的结构型模式可供研究，它们分别是：  
  
[外观模式/门面模式（Facade门面模式）](http://blog.csdn.net/itpinpai/article/details/51363049)  
适配器模式（Adapter）  
代理模式（Proxy）  
[装饰模式（Decorator）](http://blog.csdn.net/itpinpai/article/details/51556237)  
桥梁模式/桥接模式（Bridge）  
[组合模式（Composite）](http://blog.csdn.net/itpinpai/article/details/47361815)  
享元模式（Flyweight）  
  
行为型模式  
在对象的结构和对象的创建问题都解决了之后，就剩下对象的行为问题了，如果对象的行为设计的好，那么对象的行为就会更清晰，它们之间的协作效率就会提高，这里有11个具体的行为型模式可供研究，它们分别是：  
  
模板方法模式（Template Method）  
[观察者模式（Observer）](http://blog.csdn.net/itpinpai/article/details/51556237)  
[状态模式（State）](http://blog.csdn.net/itpinpai/article/details/51580737)  
[策略模式（Strategy）](http://blog.csdn.net/itpinpai/article/details/51596125)  
[职责链模式（Chain of Responsibility）](http://blog.csdn.net/itpinpai/article/details/51601956)  
命令模式（Command）  
[访问者模式（Visitor）](http://blog.csdn.net/itpinpai/article/details/51644922)  
[调停者模式（Mediator）](http://blog.csdn.net/itpinpai/article/details/51656708)  
备忘录模式（Memento）  
迭代器模式（Iterator）  
[解释器模式（Interpreter）](http://blog.csdn.net/itpinpai/article/details/51657199)

## 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。

## 单例模式

#### 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。

#### 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*;**  **}**  **}** |

##### 双重检验锁

|  |
| --- |
| **// 懒汉式 第二种写法 效率高 双重检验锁**  **static public Singleton getSingleton2() {**  **if (*singleton* == null) { // 第一步检验锁**  **synchronized (Singleton.class) { // 第二步检验锁**  **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*;**  **}**  **}** |

## 工厂模式

### 什么是工厂模式？

**实现创建者和调用者分离**

### 简单工厂

|  |
| --- |
| **interface Car {**  **void run();**  **}**  **class AoDi implements Car {**  **@Override**  **public void run() {**  **System.*out*.println("我是奥迪....");**  **}**  **}**  **class BenChi implements Car {**  **@Override**  **public void run() {**  **System.*out*.println("我是奔驰....");**  **}**  **}**  **class CarFactory {**  **public static Car createCar(String name) {**  **Car car = null;**  **switch (name) {**  **case "奥迪":**  **car = new AoDi();**  **break;**  **case "奔驰":**  **car = new BenChi();**  **break;**  **default:**  **break;**  **}**  **return car;**  **}**  **}**  **public class Test002 {**  **public static void main(String[] args) {**  **Car car = CarFactory.*createCar*("奔驰");**  **car.run();**  **}**  **}** |

### 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....");  }  } |

|  |
| --- |
| **interface Car {**  **void run();**  **}**  **class AoDi implements Car {**  **@Override**  **public void run() {**  **System.*out*.println("我是奥迪....");**  **}**  **}**  **class BenChi implements Car {**  **@Override**  **public void run() {**  **System.*out*.println("我是奔驰....");**  **}**  **}**  **class BenChiFactory {**  **public static Car createCar() {**  **return new BenChi();**  **}**  **}**  **class AoDiFactory {**  **public static Car createCar() {**  **return new AoDi();**  **}**  **}**  **public class Test002 {**  **public static void main(String[] args) {**  **// Car car = CarFactory.createCar("奔驰");**  **// car.run();**  **Car aodi = AoDiFactory.*createCar*();**  **Car benchi = BenChiFactory.*createCar*();**  **aodi.run();**  **benchi.run();**  **}**  **}** |

## 代理模式

### 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。



### 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

### 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）

### 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();  }  } |

### JDK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();  }  } |

### 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();**  **}**  **}** |

### CGLIB与JDK动态代理区别

区别：  
java动态代理是利用反射机制生成一个实现代理接口的匿名类，在调用具体方法前调用InvokeHandler来处理。而cglib动态代理是利用asm开源包，对代理对象类的class文件加载进来，通过修改其字节码生成子类来处理。  
1、如果目标对象实现了接口，默认情况下会采用JDK的动态代理实现AOP   
2、如果目标对象实现了接口，可以强制使用CGLIB实现AOP   
3、如果目标对象没有实现了接口，必须采用CGLIB库，spring会自动在JDK动态代理和CGLIB之间转换