Java进阶2 第3天

**【学习目标】理解、了解、应用、记忆**

通过今天的学习，参训学员能够：（解释的时候说出二级目标的掌握程度）

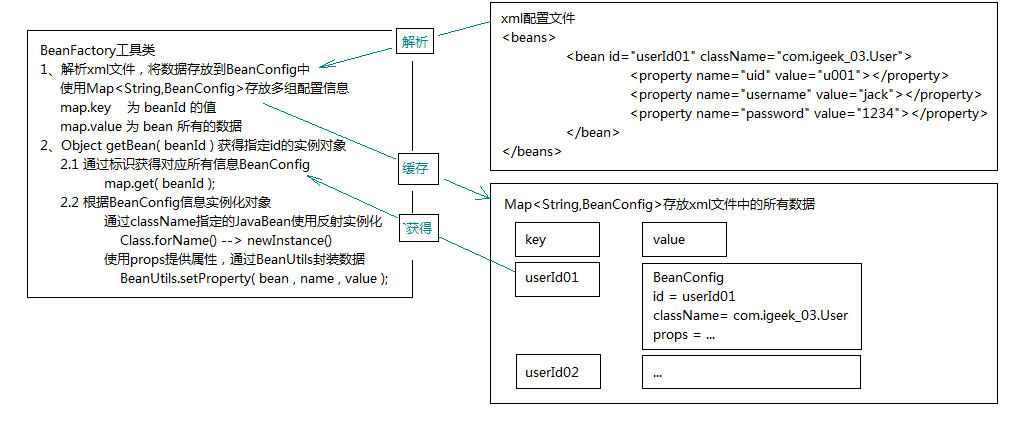
1. **【理解】XML的综合案例**
2. 【理解】阐述XML案例的的需求
3. 【理解】阐述XML综合案例的思路流程
4. 【应用】能够编写代码实现XML综合案例
5. **【应用】注解的应用**
6. 【理解】阐述注解的作用
7. 【理解】阐述JDK常见的注解及应用场景
8. 【理解】阐述自定义注解定义的格式及使用方式
9. 【理解】阐述元注解的作用、格式及使用场景
10. 【应用】编写代码完成注解的综合案例
11. **【应用】阐述类的加载**
12. 【理解】阐述类加载的机制
13. 【应用】编写代码使用引导类加载器、扩展类加载器及应用类加载器加载class文件
14. **【应用】动态代理**
15. 【理解】阐述动态代理的机制是什么
16. 【应用】编写代码使用动态代理对一个接口的方法进行增强

# XML综合案例

## 需求

结合XML、反射的知识点完成一个综合案例。

需求：编写一个BeanFactroy类来读取XML配置文件中配置的JavaBean信息，将读取都的信息使用BeanUtils封装进JavaBean对象，将对象存储在Map集合，并在BeanFactory中提供通过id查找JavaBean的方法。



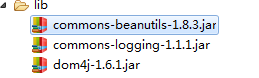
我们可以多次调用 BeanFactory.getBean(“beanId”)，将创建不同对象。

1. BeanFactory首先加载(解析) xml配置文件
2. 将每一个<bean>标签配置内容，封装到BeanConfig对象中
3. 最后将整个xml所有配置内容都封装一个容器，且此容器可以快速查询。将采用Map进行数据存储。

## 准备工作

### 案例代码一

1. **准备jar包**

****

1. **准备JavaBean**

**User类**

|  |
| --- |
| **package** com.igeek\_01;  /\*\*  \* **@ClassName**: User  \* **@Description**: 用户类  \* **@date** 2018年2月7日 下午1:19:07  \* Company www.igeekhome.com  \*  \*/  **public** **class** User {  /\*\*  \* **@Fields** uid : 用户ID  \*/  **private** String uid;  /\*\*  \* **@Fields** userName : 用户名  \*/  **private** String userName;  /\*\*  \* **@Fields** password : 密码  \*/  **private** String password;  /\*\*  \* **@Title**: User  \*/  **public** User() {  **super**();  }  /\*\*  \* **@return** the uid  \*/  **public** String getUid() {  **return** uid;  }  /\*\*  \* **@param** uid the uid to set  \*/  **public** **void** setUid(String uid) {  **this**.uid = uid;  }  /\*\*  \* **@return** the userName  \*/  **public** String getUserName() {  **return** userName;  }  /\*\*  \* **@param** userName the userName to set  \*/  **public** **void** setUserName(String userName) {  **this**.userName = userName;  }  /\*\*  \* **@return** the password  \*/  **public** String getPassword() {  **return** password;  }  /\*\*  \* **@param** password the password to set  \*/  **public** **void** setPassword(String password) {  **this**.password = password;  }  /\*\*  \* **@Title**: toString  \* **@Description**: 用户的toString方法  \* **@return**  \* **@see** java.lang.Object#toString()  \*/  @Override  **public** String toString() {  **return** "User [uid=" + uid + ", userName=" + userName + ", password=" + password + "]";  }  } |

**Book类**

|  |
| --- |
| **package** com.igeek\_01;  /\*\*  \* **@ClassName**: Book  \* **@Description**: 书类  \* **@date** 2018年2月7日 下午1:22:15  \* Company www.igeekhome.com  \*  \*/  **public** **class** Book {  /\*\*  \* **@Fields** bid : 书的编号  \*/  **private** String bid;  /\*\*  \* **@Fields** title : 书名  \*/  **private** String title;  /\*\*  \* **@Fields** price : 价格  \*/  **private** Integer price;  /\*\*  \* **@Title**: Book  \*/  **public** Book() {  **super**();  }  /\*\*  \* **@return** the bid  \*/  **public** String getBid() {  **return** bid;  }  /\*\*  \* **@param** bid the bid to set  \*/  **public** **void** setBid(String bid) {  **this**.bid = bid;  }  /\*\*  \* **@return** the title  \*/  **public** String getTitle() {  **return** title;  }  /\*\*  \* **@param** title the title to set  \*/  **public** **void** setTitle(String title) {  **this**.title = title;  }  /\*\*  \* **@return** the price  \*/  **public** Integer getPrice() {  **return** price;  }  /\*\*  \* **@param** price the price to set  \*/  **public** **void** setPrice(Integer price) {  **this**.price = price;  }  /\*\*  \* **@Title**: toString  \* **@Description**: 书的toString方法  \* **@return**  \* **@see** java.lang.Object#toString()  \*/  @Override  **public** String toString() {  **return** "Book [bid=" + bid + ", title=" + title + ", price=" + price + "]";  }  } |

1. **准备XML数据**

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <beans>  <bean id=*"userId01"* className=*"com.igeek\_01.User"*>  <property name=*"uid"* value=*"u001"*></property>  <property name=*"username"* value=*"jack"*></property>  <property name=*"password"* value=*"1234"*></property>  </bean>    <bean id=*"bookId02"* className=*"com.igeek\_01.Book"*>  <property name=*"bid"* value=*"b001"*></property>  <property name=*"title"* value=*"标题"*></property>  <property name=*"price"* value=*"98"*></property>  </bean>  </beans> |

1. **准备测试类XMLTest**

|  |
| --- |
| **package** com.igeek\_01;  **import** org.junit.Test;  /\*\*  \* **@ClassName**: XMLTest  \* **@Description**: XML解析测试类  \* **@date** 2018年2月7日 下午1:25:56  \* Company www.igeekhome.com  \*  \*/  **public** **class** XMLTest {  @Test  **public** **void** testGetBean(){  User user = (User)BeanFactory.*getBean*("userId01");  System.***out***.println(user);    Book book = (Book)BeanFactory.*getBean*("bookId02");  System.***out***.println(book);  }  } |

## 创建BeanConfig对象

BeanConfig对象对应的XML中的配置项，作用将解析的XML封装到这个对象中

属性：id、className、properties

### 案例代码二

**BeanConfig类**

|  |
| --- |
| **package** com.igeek\_01;  **import** java.util.Properties;  /\*\*  \* **@ClassName**: BeanConfig  \* **@Description**: JavaBean配置对象  \* **@date** 2018年2月7日 下午1:34:28  \* Company www.igeekhome.com  \*  \* BeanConfig对象对应的XML中的配置项，作用将解析的XML封装到这个对象中  \*/  **public** **class** BeanConfig {  /\*\*  \* **@Fields** id : bean的id  \*/  **private** String id;  /\*\*  \* **@Fields** className : bean的类名  \*/  **private** String className;  /\*\*  \* **@Fields** props : bean的属性  \*/  **private** Properties props = **new** Properties();  /\*\*  \* **@Title**: BeanConfig  \*/  **public** BeanConfig() {  **super**();  }  /\*\*  \* **@return** the id  \*/  **public** String getId() {  **return** id;  }  /\*\*  \* **@param** id the id to set  \*/  **public** **void** setId(String id) {  **this**.id = id;  }  /\*\*  \* **@return** the className  \*/  **public** String getClassName() {  **return** className;  }  /\*\*  \* **@param** className the className to set  \*/  **public** **void** setClassName(String className) {  **this**.className = className;  }  /\*\*  \* **@return** the props  \*/  **public** Properties getProps() {  **return** props;  }  /\*\*  \* **@param** props the props to set  \*/  **public** **void** setProps(Properties props) {  **this**.props = props;  }  /\*\*  \* **@Title**: toString  \* **@Description**: bean配置对象的toString方法  \* **@return**  \* **@see** java.lang.Object#toString()  \*/  @Override  **public** String toString() {  **return** "BeanConfig [id=" + id + ", className=" + className + ", props=" + props + "]";  }  } |

## 解析XML并封装BeanConfig

### 案例代码三

|  |
| --- |
| **package** com.igeek\_01;  **import** java.util.HashMap;  **import** java.util.List;  **import** java.util.Map;  **import** org.dom4j.Document;  **import** org.dom4j.Element;  **import** org.dom4j.io.SAXReader;  /\*\*  \* **@ClassName**: BeanFactory  \* **@Description**: Bean工厂类  \* **@date** 2018年2月7日 下午2:02:18  \* Company www.igeekhome.com  \*  \*/  **public** **class** BeanFactory {  //1.1 提供Map存放 bean.xml配置文件中所有内容，且易于快速查询，使用Map<String,BeanConfig>  **private** **static** Map<String,BeanConfig> *cache* = **new** HashMap<String, BeanConfig>();    //1.2 解析xml，将数据添加到map中  **static**{  **try** {  //1) 加载xml文件，获得document  SAXReader saxReader = **new** SAXReader();  Document document = saxReader.read("beans.xml");    //2) 获得根元素<beans>  Element rootElement = document.getRootElement();    //3) 获得所有<bean>元素  List<Element> allBeanElement = rootElement.elements("bean");  **for**(Element beanElement : allBeanElement) {  //4) 获得id和className属性  String id = beanElement.attributeValue("id");  String className = beanElement.attributeValue("className");    /\*\*#1 创建BeanConfig, 并封装id和className、\*/  BeanConfig beanConfig = **new** BeanConfig();  beanConfig.setId(id);  beanConfig.setClassName(className);    //5) 获得<property>子标签  List<Element>allPropertyElement = beanElement.elements("property");  **for**(Element propElement : allPropertyElement) {  //6) 获得name和value属性  String name = propElement.attributeValue("name");  String value = propElement.attributeValue("value");    /\*\*#2 将 name和value 封装到BeanConfig.props中\*/  beanConfig.getProps().setProperty(name, value);    }    /\*\*#3 将封装好的BeanConfig，添加到Map中\*/  *cache*.put(id, beanConfig);  }  System.***out***.println("数据初始化成功：" + *cache*);  } **catch** (Exception e) {  //将编译时转换成运行时异常  **throw** **new** RuntimeException(e);  }  }  } |

## 反射实例化对象

### 代码案例四

|  |
| --- |
| **package** com.igeek\_01;  **import** java.util.HashMap;  **import** java.util.List;  **import** java.util.Map;  **import** org.apache.commons.beanutils.BeanUtils;  **import** org.dom4j.Document;  **import** org.dom4j.Element;  **import** org.dom4j.io.SAXReader;  /\*\*  \* **@ClassName**: BeanFactory  \* **@Description**: Bean工厂类  \* **@date** 2018年2月7日 下午2:02:18  \* Company www.igeekhome.com  \*  \*/  **public** **class** BeanFactory {  //1.1 提供Map存放 bean.xml配置文件中所有内容，且易于快速查询，使用Map<String,BeanConfig>  **private** **static** Map<String,BeanConfig> *cache* = **new** HashMap<String, BeanConfig>();    //1.2 解析xml，将数据添加到map中  **static**{  **try** {  //1) 加载xml文件，获得document  SAXReader saxReader = **new** SAXReader();  Document document = saxReader.read("beans.xml");    //2) 获得根元素<beans>  Element rootElement = document.getRootElement();    //3) 获得所有<bean>元素  List<Element> allBeanElement = rootElement.elements("bean");  **for**(Element beanElement : allBeanElement) {  //4) 获得id和className属性  String id = beanElement.attributeValue("id");  String className = beanElement.attributeValue("className");    /\*\*#1 创建BeanConfig, 并封装id和className、\*/  BeanConfig beanConfig = **new** BeanConfig();  beanConfig.setId(id);  beanConfig.setClassName(className);    //5) 获得<property>子标签  List<Element>allPropertyElement = beanElement.elements("property");  **for**(Element propElement : allPropertyElement) {  //6) 获得name和value属性  String name = propElement.attributeValue("name");  String value = propElement.attributeValue("value");    /\*\*#2 将 name和value 封装到BeanConfig.props中\*/  beanConfig.getProps().setProperty(name, value);    }    /\*\*#3 将封装好的BeanConfig，添加到Map中\*/  *cache*.put(id, beanConfig);  }  System.***out***.println("数据初始化成功：" + *cache*);  } **catch** (Exception e) {  //将编译时转换成运行时异常  **throw** **new** RuntimeException(e);  }  }    /\*\*  \* **@Title**: getBean  \* **@Description**: 通过指定 bean id 获得具体实例对象  \* **@param** beanId  \* **@return**  \*/  **public** **static** Object getBean(String beanId){    //1 通过beanId从Map中获得对应BeanConfig  BeanConfig beanConfig = *cache*.get(beanId);  **if**(beanConfig == **null**){  **throw** **new** RuntimeException("获得对象["+beanId+"]不存在");  }    **try** {  //2 通过beanConfig.className创建实例对象  String className = beanConfig.getClassName();  Class clazz = Class.*forName*(className);  Object obj = clazz.newInstance();    //3 循环遍历BeanConfig.props，使用BeanUtils进行数据封装  **for**(String name : beanConfig.getProps().stringPropertyNames()){  String value = beanConfig.getProps().getProperty(name);  //使用BeanUtils封装数据  BeanUtils.*setProperty*(obj, name, value);  }    **return** obj;    } **catch** (Exception e) {  **throw** **new** RuntimeException(e);  }  }  } |

# 注解

## 注解概述

* 什么是注解：Annotation注解，是一种代码级别的说明。它是JDK1.5及以后版本引入的一个特性，与类、接口、枚举是在同一个层次
  + 对比注释：注释是给开发人员阅读的，注解是给计算机提供相应信息的。
* 注解的作用：
  1. 编译检查：通过代码里标识注解，让编译器能够实现基本的编译检查。例如：@Override
  2. 代码分析：通过代码里标识注解，对代码进行分析，从而达到取代xml目的。
  3. 编写文档：通过代码里标识注解，辅助生成帮助文档对应的内容

## JDK常见注解

### 常见注解

1. @Deprecated 表示被修饰的方法已经过时。过时的方法不建议使用，但仍可以使用。
   * 一般被标记位过时的方法都存在不同的缺陷：1安全问题；2新的API取代
2. @Override JDK5.0表示复写父类的方法；JDK6.0 还可以表示实现接口的方法
3. @SuppressWarnings 表示抑制警告，被修饰的类或方法如果存在编译警告，将被编译器忽略

deprecation ，或略过时

rawtypes ，忽略类型安全

unused ， 忽略不使用

unchecked ，忽略安全检查

null，忽略空指针

all，忽略所有

### 案例代码五

**AnnotationDemo**

|  |
| --- |
| **package** com.igeek\_02;  /\*\*  \* **@ClassName**: AnnotationDemo  \* **@Description**: "@Deprecated"注释：表示被修饰的方法已经过时  \* **@date** 2018年2月7日 下午2:15:52  \* Company www.igeekhome.com  \*  \*/  **public** **class** AnnotationDemo {  @Deprecated  **public** **void** ~~init~~(){    }  } |

**AnnotationDemo02**

|  |
| --- |
| **package** com.igeek\_02;  /\*\*  \* **@ClassName**: AnnotationDemo2  \* **@Description**: "@Override"注释：JDK5.0表示复写父类的方法；JDK6.0 还可以表示实现接口的方法  \* **@date** 2018年2月7日 下午2:18:26  \* Company www.igeekhome.com  \*  \*/  **public** **class** AnnotationDemo2 {    }  //情况1： jdk1.5表示复写父类的方法  **class** Parent\_2\_1{  /\*\*  \* **@Title**: init  \* **@Description**: 父类中的初始化方法  \*/  **public** **void** init(){  }  }  **class** Son\_2\_1 **extends** Parent\_2\_1{  /\*\*  \* **@Title**: init  \* **@Description**: 重写父类的初始化方法  \* **@see** com.igeek\_02.Parent\_2\_1#init()  \*/  @Override  **public** **void** init() {  }  }  //情况2：jdk1.6 表示方法实现接口声明的方法  **interface** Parent\_2\_2{  /\*\*  \* **@Title**: init  \* **@Description**: 接口中的抽象初始化方法  \*/  **public** **void** init();  }  **class** Son\_2\_2 **implements** Parent\_2\_2 {  /\*\*  \* **@Title**: init  \* **@Description**: 实现接口中的抽象初始化方法  \* **@see** com.igeek\_02.Parent\_2\_2#init()  \*/  @Override  **public** **void** init() {    }  } |

**AnnotationDemo3**

|  |
| --- |
| **package** com.igeek\_02;  **import** java.io.Serializable;  **import** java.util.ArrayList;  **import** java.util.List;  /\*\*  \* **@ClassName**: AnnotationDemo3  \* **@Description**: "@SuppressWarnings"注释： 表示抑制警告，被修饰的类或方法如果存在编译警告，将被编译器忽略  \* **@date** 2018年2月7日 下午2:22:10  \* Company www.igeekhome.com  \*  \* deprecation ，或略过时  \* rawtypes ，忽略类型安全  \* unused ， 忽略不使用  \* unchecked ，忽略安全检查  \* null，忽略空指针  \* all，忽略所有  \*/  @SuppressWarnings("serial")  **public** **class** AnnotationDemo3 **implements** Serializable{    @SuppressWarnings({ "unused", "rawtypes", "null", "deprecation" })  **public** **static** **void** main(String[] args) {    List list = **new** ArrayList();    String str = **null**;  str.toString();    **new** Thread().~~stop~~();  }  } |

## 自定义注解

### 自定义注解--定义与使用

* 定义注解使用关键字： @interface
  1. 定义类： class
  2. 定义接口：interface
  3. 定义枚举：enum

// #1 定义注解

@interface MyAnno1{

}

* 定义带有属性的注解

//#2 定义含有属性的注解

@interface MyAnno2{

public String username() default "jack";

}

* 属性格式：修饰符 返回值类型 属性名() [default 默认值]
  1. 修饰符：默认值 public abstract ，且只能是public abstract。



* 1. 返回值类型：基本类型、字符串String、Class、注解、枚举，以及以上类型的一维数组



* 1. 属性名：自定义
  2. default 默认值：可以省略
* 注解使用的注意事项：
  1. 注解可以没有属性，如果有属性需要使用小括号括住。例如：@MyAnno1 或 @MyAnno1()
  2. 属性格式：属性名=属性值，多个属性使用逗号分隔。例如：@MyAnno2(username="rose")
  3. 如果属性名为value，且当前只有一个属性，value可以省略。
  4. 如果使用多个属性时，k的名称为value不能省略
  5. 如果属性类型为数组，设置内容格式为：{ 1,2,3 }。例如：arrs = {"java","igeek"}
  6. 如果属性类型为数组，值只有一个{} 可以省略的。例如：arrs = "igeek"
  7. 一个对象上，注解只能使用一次，不能重复使用。

### 代码案例六

**Anno01.java**

|  |
| --- |
| **package** com.igeek\_03;  /\*\*  \* **@ClassName**: Anno01  \* **@Description**: 自定义注解01  \* **@date** 2018年2月7日 下午2:30:00  \* Company www.igeekhome.com  \*  \*/  **public** **@interface** Anno01 {  } |

**Anno02.java**

|  |
| --- |
| **package** com.igeek\_03;  /\*\*  \* **@ClassName**: Anno02  \* **@Description**: 自定义注解02  \* **@date** 2018年2月7日 下午2:30:40  \* Company www.igeekhome.com  \*  \* 格式：修饰符返回值类型属性名() default 默认值  \* 修饰符：public abstract  \* 返回值类型：基本类型、String、Class、注解、枚举，以及以上类型的一位数组  \* 属性名：自定义  \* 默认值：可以省略  \*/  **public** **@interface** Anno02 {  **public** **abstract** String username() **default** "默认值";  **public** **int** age() ;  **public** Class clazz();  **public** Anno01 anno();  **public** Color color();  }  //枚举：相当于多例。Color.RED  **enum** Color{  ***RED***,***BLUE***;  }  //自定义多例  **class** Color2{  **private** Color2(){    }  **public** **static** **final** Color2 ***RED*** = **new** Color2();  **public** **static** **final** Color2 ***BLUE*** = **new** Color2();  } |

**Anno03.java**

|  |
| --- |
| **package** com.igeek\_03;  /\*\*  \* **@ClassName**: Anno03  \* **@Description**: 自定义注解03  \* **@date** 2018年2月7日 下午2:30:00  \* Company www.igeekhome.com  \*  \*/  **public** **@interface** Anno03 {  **public** String[] hobbies();  } |

**Anno04.java**

|  |
| --- |
| **package** com.igeek\_03;  /\*\*  \* **@ClassName**: Anno04  \* **@Description**: 自定义注解04  \* **@date** 2018年2月7日 下午2:30:00  \* Company www.igeekhome.com  \*  \*/  **public** **@interface** Anno04 {  **public** String[] value();  } |

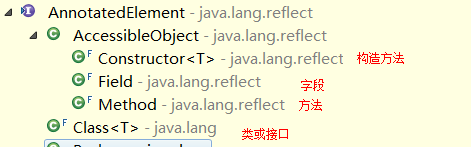
**TestAnnotation.java**

|  |
| --- |
| **package** com.igeek\_03;  **import** java.util.Date;  /\*\*  \* **@ClassName**: AnnotationDemo  \* **@Description**: 自定义注解的测试类  \* **@date** 2018年2月7日 下午2:35:42  \* Company www.igeekhome.com  \*  \*/  @Anno01  @Anno02(username="jack", age=18, clazz=Date.**class**, anno=@Anno01, color=Color.***RED***)  //@Anno03(hobbies={"抽烟","喝酒","打牌"}) //数组表示使用{}括住  @Anno03(hobbies="抽烟") //如果只有一个值，{}可以省略  //@Anno04(value={"a","b"})  //@Anno04(value="a")  @Anno04("a") //只有一对属性时，且属性名为value，可以省略属性名  //同一个对象中，同一个注解只能使用一次。  **public** **class** AnnotationDemo {  } |

### 自定义注解--解析和元注解

#### 解析

如果给类、方法等添加注解，如果需要获得注解上设置的数据，那么我们就必须对注解进行解析，JDK提供java.lang.reflect.AnnotatedElement接口允许在运行时通过反射获得注解。



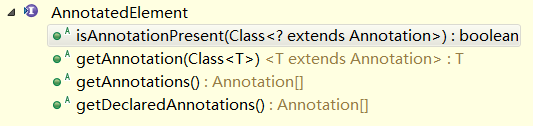
常用方法：

boolean isAnnotationPresent(Class annotationClass) 当前对象是否有注解

T getAnnotation(Class<T>annotationClass) 获得当前对象上指定的注解

Annotation[] getAnnotations() 获得当前对象及其从父类上继承的，所有的注解

Annotation[] getDeclaredAnnotations() 获得当前对象上所有的注解



**测试**

|  |
| --- |
| **package** com.igeek\_03\_03;  /\*\*  \* **@ClassName**: MyAnno  \* **@Description**: 我的自定义注解  \* **@date** 2018年2月7日 下午2:45:20  \* Company www.igeekhome.com  \*  \*/  **public** **@interface** MyAnno {  } |

|  |
| --- |
| **package** com.igeek\_03\_03;  /\*\*  \* **@ClassName**: MyAnnoDemo  \* **@Description**: 使用自定义注解  \* **@date** 2018年2月7日 下午2:46:14  \* Company www.igeekhome.com  \*  \*/  @MyAnno  **public** **class** MyAnnoDemo {  } |

|  |
| --- |
| **package** com.igeek\_03\_03;  /\*\*  \* **@ClassName**: MyAnnoTest  \* **@Description**: 自定义注解的测试类  \* **@date** 2018年2月7日 下午2:46:39  \* Company www.igeekhome.com  \*  \*/  **public** **class** MyAnnoTest {  **public** **static** **void** main(String[] args) {  **boolean** b = MyAnnoDemo.**class**.isAnnotationPresent(MyAnno.**class**);  System.***out***.println(b); //false  }  } |

当运行上面程序后，我们希望输出结果是true，但实际是false。TestAnno2类上有@MyAnno1注解，但运行后不能获得，因为每一个自定义注解，需要使用JDK提供的元注解进行修饰才可以真正的使用。

#### 元注解

元注解：用于修饰注解的注解。（用于修饰自定义注解的JDK提供的注解）

JDK提供4种元注解：

@Retention 用于确定被修饰的自定义注解生命周期

RetentionPolicy.SOURCE 被修饰的注解只能存在源码中，字节码class没有。

用途：提供给编译器使用。

RetentionPolicy.CLASS 被修饰的注解只能存在源码和字节码中，运行时内存中没有。

用途：JVM java虚拟机使用

RetentionPolicy.RUNTIME 被修饰的注解存在源码、字节码、内存（运行时）。

用途：取代xml配置

@Target 用于确定被修饰的自定义注解 使用位置

ElementType.TYPE 修饰 类、接口

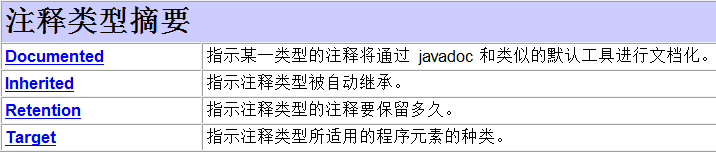
ElementType.CONSTRUCTOR 修饰构造

ElementType.METHOD 修饰方法

ElementType.FIELD 修饰字段

@Documented 使用javaDoc生成 api文档时，是否包含此注解 (了解)

@Inherited 如果父类使用被修饰的注解，子类是否继承。（了解）





修改注解类，在运行测试实例，输出结果为：true。

|  |
| --- |
| **package** com.igeek\_03\_03;  **import** java.lang.annotation.Retention;  **import** java.lang.annotation.Target;  **import** java.lang.annotation.ElementType;  **import** java.lang.annotation.RetentionPolicy;  /\*\*  \* **@ClassName**: MyAnno  \* **@Description**: 我的自定义注解  \* **@date** 2018年2月7日 下午2:45:20  \* Company www.igeekhome.com  \*  \*/  @Target(ElementType.***TYPE***)  @Retention(RetentionPolicy.***RUNTIME***)  **public** **@interface** MyAnno {  } |

### 案例代码七

**MyAnnoDemo.java**

|  |
| --- |
| **package** com.igeek\_03\_04;  /\*\*  \* **@ClassName**: MyAnnoDemo  \* **@Description**: 使用自定义注解  \* **@date** 2018年2月7日 下午2:55:27  \* Company www.igeekhome.com  \*  \*/  @MyAnno("测试数据")  **public** **class** MyAnnoDemo {    **public** **static** **void** main(String[] args) {  //运行时，获得指定类上面的注解，从而获得对应数据  **boolean** b = MyAnnoDemo.**class**.isAnnotationPresent(MyAnno.**class**);  **if**(b){  MyAnno myAnno = MyAnnoDemo.**class**.getAnnotation(MyAnno.**class**);  String value = myAnno.value();  System.***out***.println(value);  } **else** {  System.***out***.println("没有注解");  }  }    /\*\*  \* **@Title**: MyAnnoDemo  \*/  @MyAnno  **public** MyAnnoDemo(){    }    /\*\*  \* **@Title**: init  \* **@Description**: 初始化方法  \*/  @MyAnno  **public** **void** init(){    }  } |

**MyAnno.java**

|  |
| --- |
| **package** com.igeek\_03\_04;  **import** java.lang.annotation.Retention;  **import** java.lang.annotation.Target;  **import** java.lang.annotation.ElementType;  **import** java.lang.annotation.RetentionPolicy;  /\*\*  \* **@ClassName**: MyAnno  \* **@Description**: 我的自定义注解  \* **@date** 2018年2月7日 下午2:45:20  \* Company www.igeekhome.com  \*  \* 元注解：jdk提供一套注解，元注解用于修饰自定义注解。  \* 用于控制自定义注解使用位置、生命周期等基本信息。  \*  \* **@Target** 用于限定自定义注解使用位置  \* **@Target**(ElementType.CONSTRUCTOR) 只能在构造方法使用  \* **@Target**(ElementType.METHOD) 只能在普通方法使用  \* **@Target**(ElementType.FIELD) 只能在字段使用  \* **@Target**(ElementType.TYPE) 只能在类、接口使用  \*  \* **@Retention** 用于确定自定义注解生命周期  \* **@Retention**(RetentionPolicy.SOURCE)  \* 自定义注解只在源码有效，编译之后将删除(class文件没有)。提供编译器使用  \* **@Retention**(RetentionPolicy.CLASS)  \* 自定义注解只在源码和字节码有效，编译之后有，运行时内存没有。提供JVM使用  \* **@Retention**(RetentionPolicy.RUNTIME)  \* 自定义注解在源码、字节码和内存都有效。提供程序使用，用于取代xml配置文件  \*/  @Target({ElementType.***TYPE***,ElementType.***CONSTRUCTOR***,ElementType.***METHOD***})  @Retention(RetentionPolicy.***RUNTIME***)  **public** **@interface** MyAnno {  **public** String value() **default** "默认值";  } |

## 综合案例

### 案例分析

* 模拟Junit测试，首先需要编写自定义注解@MyTest，并添加元注解，保证自定义注解只能修改方法，且在运行时可以获得。
* 其次编写目标类（测试类），然后给目标方法（测试方法）使用@MyTest注解
* 最后编写测试类，使用main方法模拟Junit的右键运行。

### 案例代码八

**MyTest.java**

|  |
| --- |
| **package** com.igeek\_04;  **import** java.lang.annotation.Retention;  **import** java.lang.annotation.Target;  **import** java.lang.annotation.ElementType;  **import** java.lang.annotation.RetentionPolicy;  /\*\*  \* **@ClassName**: MyTest  \* **@Description**: 我的自定义测试注解  \* **@date** 2018年2月7日 下午3:00:54  \* Company www.igeekhome.com  \*  \*/  @Target(ElementType.***METHOD***)  @Retention(RetentionPolicy.***RUNTIME***)  **public** **@interface** MyTest {  } |

**MyTestDemo.java**

|  |
| --- |
| **package** com.igeek\_04;  /\*\*  \* **@ClassName**: MyTestDemo  \* **@Description**: 我的自定义测试方法  \* **@date** 2018年2月7日 下午3:02:27  \* Company www.igeekhome.com  \*  \*/  **public** **class** MyTestDemo {    /\*\*  \* **@Title**: testDemo01  \* **@Description**: 测试方法1  \*/  @MyTest  **public** **void** testDemo01(){  System.***out***.println("demo01");  }    /\*\*  \* **@Title**: testDemo02  \* **@Description**: 测试方法2  \*/  //@MyTest  **public** **void** testDemo02(){  System.***out***.println("demo02");  }    /\*\*  \* **@Title**: testDemo03  \* **@Description**: 测试方法3  \*/  @MyTest  **public** **void** testDemo03(){  System.***out***.println("demo03");  }  } |

**MyTestDemoTest.java**

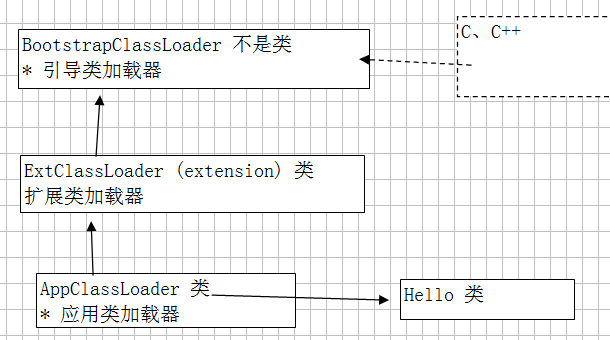
|  |
| --- |
| **package** com.igeek\_04;  **import** java.lang.reflect.Method;  /\*\*  \* **@ClassName**: MyTestDemoTest  \* **@Description**: 通过自定义注解MyTest自动执行测试方法  \* **@date** 2018年2月7日 下午3:04:58  \* Company www.igeekhome.com  \*  \* 模拟右键运行:  \* 当前指定类，所有的方法，是否有@MyTest注解，如果有将运行  \* 1.当前类  \* new 实例  \* 2.获得所有的方法  \* 3.判断方法上是否有指定的注解  \* 4.如果有，运行类  \*/  **public** **class** MyTestDemoTest {  **public** **static** **void** main(String[] args) **throws** Exception {  //模拟右键运行  //当前指定类，所有的方法，是否有@MyTest注解，如果有将运行  //1.当前类  Class clazz = MyDemo.**class**;  // new 实例  Object obj = clazz.newInstance();  //2.获得所有的方法  Method[] allMethod = clazz.getMethods();  **for**(Method method : allMethod){  //3.判断方法上是否有指定的注解  **boolean** b = method.isAnnotationPresent(MyTest.**class**);  **if**(b){  // 4.如果有，运行类  method.invoke(obj);  }  }  }  } |

# 类的加载

## 类加载--理论

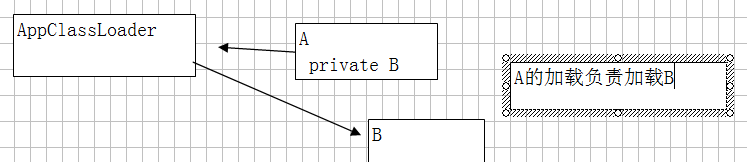
* 类加载器：类加载器是负责加载类的对象。将class文件（硬盘）加载到内存生成Class对象。

所有的类加载器，都是 java.lang.ClassLoader 的子类



* 使用 **类.class.getClassLoader()** 获得加载自己的类加载器
* 类加载器加载机制：全盘负责委托机制

全盘负责：A类如果要使用B类（不存在），A类加载器C必须负责加载B类。



委托机制：A类加载器如果要加载资源B，必须询问父类加载是否加载。

如果加载，将直接使用。

如果没有机制，自己再加载。

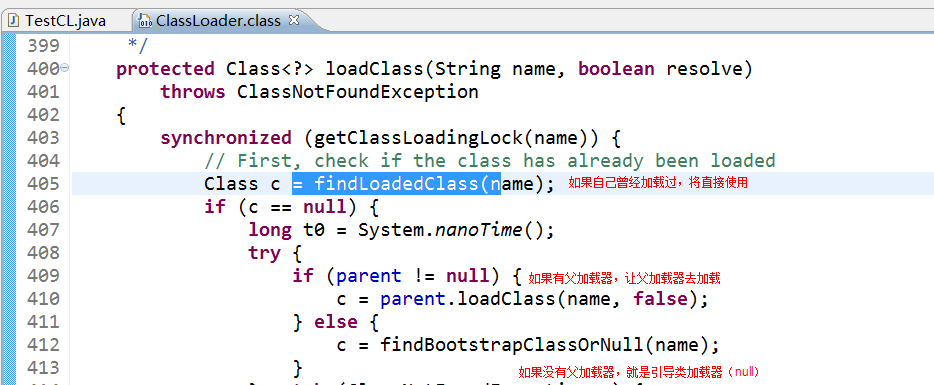
* 采用 全盘负责委托机制 保证一个class文件只会被加载一次，形成一个Class对象。
* 注意：

如果一个class文件，被两个类加载器加载，将是两个对象。

提示 com.igeek.Hello 不能强制成 com.igeek.Hello

h.getClass() -->A h.getClass() -->B

自定义类加载，可以将一个class文件加载多次。



## 类加载--演示

### 案例代码九

**ClassLoaderDemo01Test.java**

|  |
| --- |
| **package** com.igeek\_02;  **import** org.junit.Test;  /\*\*  \* **@ClassName**: ClassLoaderDemo01Test  \* **@Description**: 类加载器的测试类1  \* **@date** 2018年2月7日 下午3:15:21  \* Company www.igeekhome.com  \*  \*/  **public** **class** ClassLoaderDemo01Test {    /\*\*  \* **@Title**: testDemo01  \* **@Description**: 确定引导类加载器，加载内容：rt.jar  \*/  @Test  **public** **void** testDemo01(){  // 确定引导类加载器，加载内容：rt.jar （Runtime ）  // \* JDK固定配置信息，sun.boot.class.path用于表示引导类加载器所加载的内容  String paths = System.*getProperty*("sun.boot.class.path");  String[] allPath = paths.split(";");  **for**(String p : allPath){  System.***out***.println(p);  }  }    /\*\*  \* **@Title**: demo02  \* **@Description**: 确定String类引导类加载器，类型：null  \*/  @Test  **public** **void** testDemo02(){  // 确定引导类加载器，类型：null  ClassLoader cl = String.**class**.getClassLoader();  System.***out***.println(cl);  }  } |

**ClassLoaderDemo02Test.java**

|  |
| --- |
| **package** com.igeek\_02;  **import** org.junit.Test;  /\*\*  \* **@ClassName**: ClassLoaderDemo02Test  \* **@Description**: 类加载器的测试类2  \* **@date** 2018年2月7日 下午3:15:21  \* Company www.igeekhome.com  \*  \*/  **public** **class** ClassLoaderDemo02Test {    /\*\*  \* **@Title**: testDemo01  \* **@Description**: 确定扩展类加载器，加载内容：jre/lib/ext  \*/  @Test  **public** **void** testDemo01(){  // 确定扩展类加载器，加载内容：jre/lib/ext  // \* JDK固定配置信息，java.ext.dirs用于表示扩展类加载器所加载的内容  String paths = System.*getProperty*("java.ext.dirs");  String[] allPath = paths.split(";");  **for**(String p : allPath){  System.***out***.println(p);  }  }    /\*\*  \* **@Title**: testDemo02  \* **@Description**: 确定扩展类加载器，类型：Launcher$ExtClassLoader  \*/  @Test  **public** **void** testDemo02(){  // 确定扩展类加载器，类型：Launcher$ExtClassLoader  ClassLoader cl = ClassLoaderDemo02Test.**class**.getClassLoader().getParent();  System.***out***.println(cl);  }  } |

**ClassLoaderDemo03Test.java**

|  |
| --- |
| **package** com.igeek\_02;  **import** org.junit.Test;  /\*\*  \* **@ClassName**: ClassLoaderDemo03Test  \* **@Description**: 类加载器的测试类3  \* **@date** 2018年2月7日 下午3:15:21  \* Company www.igeekhome.com  \*  \*/  **public** **class** ClassLoaderDemo03Test {    /\*\*  \* **@Title**: testDemo01  \* **@Description**: 确定应用类加载器，加载内容：项目/bin  \*/  @Test  **public** **void** testDemo01(){  // 确定应用类加载器，加载内容：项目/bin (编译后内容) ，自己编写类由应用类加载加载  // \* JDK固定配置信息，java.class.path用于表示应用类加载器所加载的内容  String paths = System.*getProperty*("java.class.path");  String[] allPath = paths.split(";");  **for**(String p : allPath){  System.***out***.println(p);  }  }    /\*\*  \* **@Title**: testDemo02  \* **@Description**: 确定扩展类加载器，类型：Launcher$AppClassLoader  \*/  @Test  **public** **void** testDemo02(){  // 确定应用类加载器，类型：Launcher$AppClassLoader  ClassLoader cl = ClassLoaderDemo03Test.**class**.getClassLoader();  System.***out***.println(cl);  }  } |

**ClassLoaderDemo04Test.java**

|  |
| --- |
| **package** com.igeek\_02;  **import** org.junit.Test;  /\*\*  \* **@ClassName**: ClassLoaderDemo04Test  \* **@Description**: 类加载器的测试类4  \* **@date** 2018年2月7日 下午3:15:21  \* Company www.igeekhome.com  \*  \*/  **public** **class** ClassLoaderDemo04Test {    /\*\*  \* **@Title**: testDemo01  \* **@Description**: 3个类加载的关系  \*/  @Test  **public** **void** testDemo01(){  //3个类加载的关系  ClassLoader c1 = ClassLoaderDemo04.**class**.getClassLoader();  System.***out***.println(c1); //应用 (AppClassLoader)    ClassLoader c2 = c1.getParent();  System.***out***.println(c2); //扩展(ExtClassLoader)    ClassLoader c3 = c2.getParent();  System.***out***.println(c3); //引导(null)  }  } |

# 动态代理

## 动态代理概述

java代理有jdk动态代理、cglib代理，这里只说下jdk动态代理，jdk动态代理主要使用的是java反射机制（既java.lang.reflect包）。

### Proxy类

动态代理：程序运行时，使用JDK提供工具类（Proxy），动态创建一个类，此类一般用于代理。

代理：你 -- 代理（增强） -- 厂商

代理类：

目标类：被代理的

动态代理使用前提：必须有接口

Object proxyObj = Proxy.newProxyInstance(参数1,参数2,参数3);

参数1：ClassLoader，负责将动态创建类，加载到内存。 当前类.class.getClassLoader();

参数2：Class[] interfaces ,代理类需要实现的所有接口（确定方法）,被代理类实例.getClass().getInterfaces();

参数3：InvocationHandler, 请求处理类，代理类不具有任何功能，代理类的每一个方法执行时，调用处理类invoke方法。

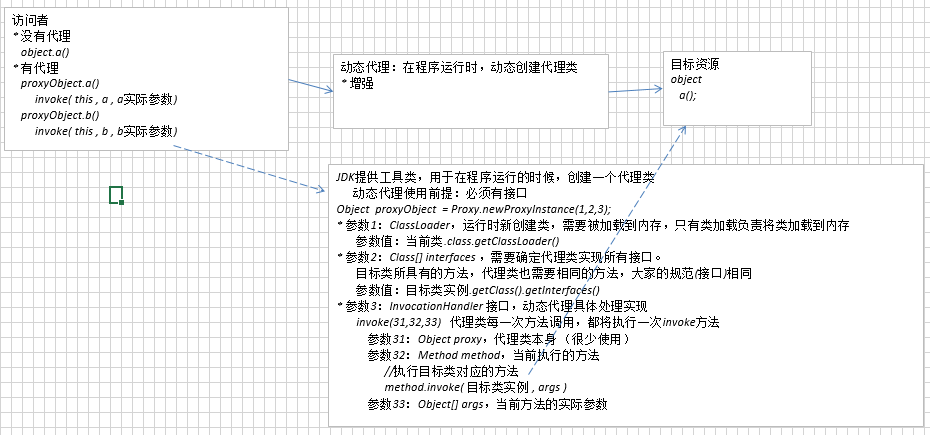
invoke(Object proxy ,Method method,Object[] args)

参数1：代理实例

参数2：当前执行的方法

参数3：方法实际参数。

## 动态代理原理



## 动态代理案例

### 需求分析

自定一个MyCollections，在该类中定义一个unmodifiableList方法实现，使用动态代理对该方法增强，使list对象不能再加入元素。

### 案例代码十

**MyCollections.java**

|  |
| --- |
| **package** com.igeek\_04;  **import** java.lang.reflect.InvocationHandler;  **import** java.lang.reflect.Method;  **import** java.lang.reflect.Proxy;  **import** java.util.List;  /\*\*  \* **@ClassName**: MyCollections  \* **@Description**: 自定义集合工具类  \* **@date** 2018年2月7日 下午4:07:14  \* Company www.igeekhome.com  \*  \* 该类中定义一个unmodifiableList方法实现，使用动态代理对该方法增强，使list对象不能再加入元素  \*/  **public** **class** MyCollections {  @SuppressWarnings({ "unchecked", "rawtypes" })  **public** **static** List<String> unmodifiableList(**final** List<String> list) {  //list 所有功能都有 (目标类)  //proxyList希望不能进行增删改，只能查询 (代理类)  // 参数1：ClassLoader，动态代理需要一个类加载器  ClassLoader loader = MyCollections.**class**.getClassLoader();  // 参数2：Class[] interfaces 需要与目标类接口保持一致  Class[] interfaces = list.getClass().getInterfaces();    List<String> proxyList = (List<String>)Proxy.*newProxyInstance*(loader, interfaces, **new** InvocationHandler(){  //代理类的每一个方法调用一次，处理类invoke方法都执行一次  @Override  **public** Object invoke(Object proxy, Method method, Object[] args) **throws** Throwable {    //处理（增强）--不允许添加  //1 获得方法名  String methodName = method.getName();  //2 eq不同方法的处理  **if**("add".equals(methodName)){  **throw** **new** UnsupportedOperationException("操作不允许");  }  //处理类中直接执行目标类对应的方法  **return** method.invoke(list, args);  }  });  **return** proxyList;  }  } |

**ProxyDemoTest.java**

|  |
| --- |
| **package** com.igeek\_04;  **import** java.util.ArrayList;  **import** java.util.Collections;  **import** java.util.List;  **import** org.junit.Test;  /\*\*  \* **@ClassName**: ProxyDemoTest  \* **@Description**: 动态代理测试类  \* **@date** 2018年2月7日 下午4:11:47  \* Company www.igeekhome.com  \*  \*/  **public** **class** ProxyDemoTest {  /\*\*  \* **@Title**: testDemo01  \* **@Description**: 使用Collections.unmodifiableList完成禁止添加功能  \*/  @Test  **public** **void** testDemo01(){  List<String> list = **new** ArrayList<>();  list.add("abc");  String s = list.get(0);  System.***out***.println(s);    List<String> list2 = Collections.*unmodifiableList*(list);    String s2 = list2.get(0);  System.***out***.println(s2);    list2.add("123"); //不允许  System.***out***.println(list2.size());    //不支持操作异常，当前list2不允许修改。  //java.lang.UnsupportedOperationException  }    /\*\*  \* **@Title**: testDemo02  \* **@Description**: 使用MyCollections.unmodifiableList完成禁止添加功能  \*/  @Test  **public** **void** testDemo02(){  List<String> list = **new** ArrayList<>();  list.add("abc");  String s = list.get(0);  System.***out***.println(s);    List<String> list2 = MyCollections.*unmodifiableList*(list);    String s2 = list2.get(0);  System.***out***.println(s2);    list2.add("123"); //不允许  System.***out***.println(list2.size());    //不支持操作异常，当前list2不允许修改。  //java.lang.UnsupportedOperationException  }  } |

重点和总结

1、XML的综合应用

2、Java中的注解

4、如何自定义注解

5、类加载概述

6、动态代理原理及应用