# 第12天【反射和注解】

## 主要内容

1. 反射引入
2. Class类
3. 使用反射创建对象
4. 使用反射操作属性
5. 使用反射执行方法
6. 使用反射操作泛型
7. 反射优缺点
8. 注解的作用
9. 内置注解
10. 元注解
11. 自定义注解
12. 使用反射获取注解

## 学习目标

|  |  |
| --- | --- |
| 知识点 | 要求 |
| 反射引入 | 理解 |
| Class类 | 掌握 |
| 使用反射创建对象 | 掌握 |
| 使用反射操作属性 | 掌握 |
| 使用反射执行方法 | 掌握 |
| 使用反射操作泛型 | 掌握 |
| 反射的优缺点 | 掌握 |
| 注解的作用 | 了解 |
| 内置注解 | 了解 |
| 元注解 | 了解 |
| 自定义注解 | 了解 |
| 使用反射读取注解 | 了解 |

## 一、反射技术

### 1.1反射引入

* 编译时知道类或对象的具体信息，此时直接对类和对象进行操作即可，无需反射（reflection）
* 如果编译不知道类或对象的具体信息，此时应该如何做呢？使用反射来实现。比如类的名称放在XML文件中，属性和属性值放在XML文件中，需要在运行时读取XML文件，动态获取类的信息

#### 【示例1】引入反射

|  |
| --- |
| **public class** Test {  **public static void** main(String[] args) **throws** Exception {  //编码/编译的时候，已经知道要创建哪个类的对象，此时和反射没关系  //创建对象  //Animal an = new Dog();  Animal an = **new** Cat();  //操作属性  an.**nickName** =**"旺财"**; an.**color** = **"黑色"**;  //执行方法  an.shout(); an.shout(**"门口"**);  an.run(); System.**out**.println(an);  //编码/编译时，不知道要创建哪个类的对象，只有根据运行时动态获取内容来创建对象  //使用DOM4J读取xml文件，最终得到了类的完整路径字符串  String className = **"com.bjsxt.why.Cat"**;  //创建对象  //Animal an2 = new "com.bjsxt.why.Cat"();  Class clazz = Class.forName(className);  Object an2 = clazz.newInstance();  //操作属性  //执行方法  } } |

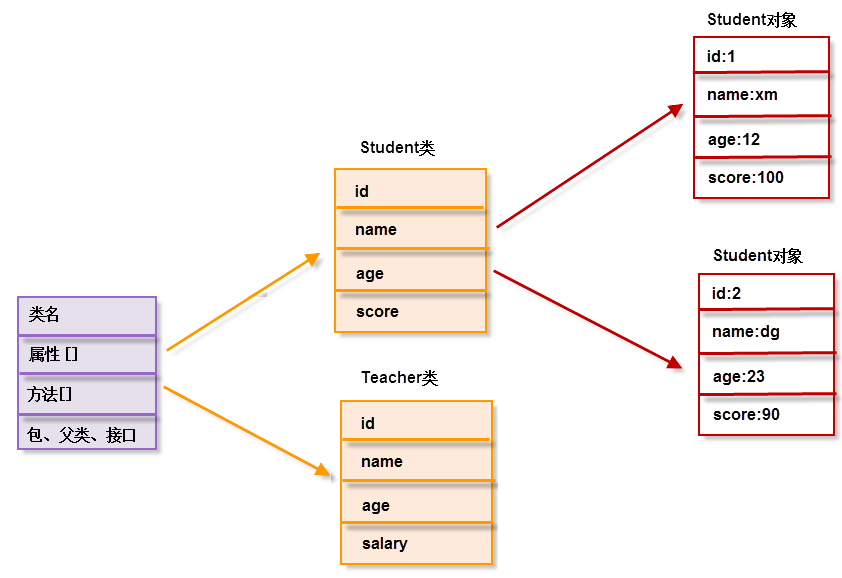
* 反射的应用场合
  + 在编译时根本无法知道该对象或类可能属于哪些类，程序只依靠运行时信息来发现该对象和类的真实信息
  + 比如log4j，Servlet、SSM框架技术都用到了反射

|  |
| --- |
| 比如：log4j  log4j.appender.stdout=org.apache.log4j.ConsoleAppender  log4j.appender.logfile.layout=org.apache.log4j.PatternLayout  比如：Servlet  <servlet>  <servlet-name>HelloServlet</servlet-name>  <servlet-class>com.bjsxt.servlet.HelloServlet</servlet-class>  </servlet>  比如 SSM  <bean id="tm"  class="org..jdbc.datasource.DataSourceTransactionManager">  <property name="dataSource" ref="dataSource"/>  </bean> |

* 反射的作用
  + 动态创建对象
  + 动态操作属性
  + 动态调用方法
  + 动态操作泛型和注解
* 在JDK中，主要由以下类来实现Java反射机制，都位于java.lang.reflect包中
  + Class类：代表一个类
  + Constructor 类：代表类的构造方法
  + Field 类：代表类的成员变量(属性)
  + Method类：代表类的成员方法

### 1.2反射的入口—Class类

* Class类是Java 反射机制的起源和入口
  + 用于获取与类相关的各种信息
  + 提供了获取类信息的相关方法
  + Class类继承自Object类



* Class类是所有类的共同的图纸
  + 每个类有自己的对象，好比图纸和实物的关系
  + 每个类也可看做是一个对象，有共同的图纸Class，存放类的结构信息，比如类的名字、属性、方法、构造方法、父类和接口，能够通过相应方法取出相应信息
* Class类的对象称为类对象。

#### 【示例2】认识Class类

|  |
| --- |
| **public class** TestClass1 {  **public static void** main(String[] args) **throws** Exception {  //1.获取一个类的结构信息（类对象 Class对象）  Class clazz = Class.forName(**"com.bjsxt.why.Dog"**);  //2.从类对象中获取类的各种结构信息  //2.1 获取基本结构信息  System.**out**.println(clazz.getName());  System.**out**.println(clazz.getSimpleName());  System.**out**.println(clazz.getSuperclass());  System.**out**.println(Arrays.toString(clazz.getInterfaces()));  //2.2 获取构造方法  //只能得到public修饰的构造方法  //Constructor[] constructors = clazz.getConstructors();  //可以得到所有的构造方法  Constructor[] constructors = clazz.getDeclaredConstructors();   System.**out**.println(constructors.**length**);  **for**(Constructor con :constructors){  //System.out.println(con.toString());  System.**out**.println(con.getName() + **"||"** +Modifier.toString(con.getModifiers())  +**" ||"** + Arrays.toString(con.getParameterTypes()));  }  //Constructor con = clazz.getConstructor();//获取无参数构造方法  //Constructor con = clazz.getConstructor(String.class,String.class);  Constructor con = clazz.getDeclaredConstructor(String.**class**,String.**class**);  System.**out**.println(con);  //2.3 获取属性  //Field[] fields = clazz.getFields();  Field [] fields = clazz.getDeclaredFields();  System.**out**.println(fields.**length**);  **for**(Field f :fields){  System.**out**.println(f);  }  //Field f = clazz.getField("color");  //private 默认 protecte public都可以获取，但不包括父类的  Field f = clazz.getDeclaredField(**"age"**);  System.**out**.println(f);  //2.3 获取方法  //Method[] methods = clazz.getMethods();  Method [] methods = clazz.getDeclaredMethods();  **for**(Method m : methods){ System.**out**.println(m); }  //Method m = clazz.getMethod("shout",String.class);  //Method m = clazz.getMethod("run");//public  Method m = clazz.getDeclaredMethod(**"run"**);  System.**out**.println(m);  } } |

**Class类的常用方法**

|  |  |
| --- | --- |
| 文件名 | 说 明 |
| getFields() | 获得类的public类型的属性。 |
| getDeclaredFields() | 获得类的所有属性 |
| getField(String name) | 获得类的指定属性 |
| getMethods() | 获得类的public类型的方法 |
| getMethod (String name,Class [] args) | 获得类的指定方法 |
| getConstrutors() | 获得类的public类型的构造方法 |
| getConstrutor(Class[] args) | 获得类的特定构造方法 |
| newInstance() | 通过类的无参构造方法创建对象 |
| getName() | 获得类的完整名字 |
| getPackage() | 获取此类所属的包 |
| getSuperclass() | 获得此类的父类对应的Class对象 |

**获取一个类的类对象的多种方式**

|  |  |
| --- | --- |
| 方 法 | 示 例 |
| Class.forName() | Class clazz = Class.forName("java.lang.Object");  Class.forName("oracle.jdbc.driver.OracleDriver"); |
| 类名.class | Class c1 = String.class;  Class c2 = Student.class;  Class c2 = int.class |
| 对象名.getClass() | String str=“sxt";  Class clazz = str.getClass(); |
| 对象名.getSuperClass() | Student stu = new Student();  Class c1 = stu.getClass();  Class c2 = stu.getSuperClass(); |
| 包装类.TYPE | Class c1 = Integer.TYPE; （内部基本数据类型的Class对象） |

#### 【示例3】获取一个类的类对象的三种方式

|  |
| --- |
| **public class** TestClass2 {  **public static void** main(String[] args) **throws** Exception {  //1.获取一个类的结构信息（类对象 Class对象）  // 1.1Class.forName(类的完整路径字符串);  //Class clazz = Class.forName("java.lang.String");  //1.2 类名.class  // Class clazz = String.class;  //1.3 对象名.getClass()  String str = **"bjsxt"**;  Class clazz = str.getClass();  //Integer in = new Integer(20);  //2.从类对象中获取类的各种结构信息  System.**out**.println(clazz.getName());  System.**out**.println(clazz.getSimpleName());  System.**out**.println(clazz.getSuperclass());  System.**out**.println(Arrays.toString(clazz.getInterfaces()));  } } |

其中类名.class、对象名.getClass()方式在编码时已经知道了要操作的类，而Class.forName()方式在操作的时候，可以知道，也可以不知道要操作的类。所以当编码时还不知道要操作的具体类，就只能使用Class.forName()方式了。

类名.class的好处在于不仅可以应用于普通的类、接口，还可以获取基本数据类型、数组的Class对象信息。

|  |
| --- |
| **更多细节**  Class对象的产生离不开类加载的过程。一个类被加载到内存并供我们使用需要经历如下三个阶段。    Class类没有公共的构造方法（有private构造方法），Class对象是在类加载的时候由Java虚拟机以及通过调用类加载器中的 defineClass 方法自动构造的，因此不能显式地直接new一个Class对象。  注意：一个类不管创建多个对象，它的类对象在内存中只有一份，第一次加载类的时候创建，位于方法区中。  .class相对两种两种方法更简单，更安全。通过字面量的方法获取Class对象的引用不会自动初始化该类。更加有趣的是字面常量的获取Class对象引用方式不仅可应用于普通的类，也可应用用接口，数组及基本数据类型，这点在反射技术应用传递参数时很有帮助。 |

### 1.3使用反射创建对象

**调用无参数构造方法创建对象**

* 方法1：通过Class的newInstance()方法
  + 该方法要求该Class对象的对应类有无参构造方法
  + 执行newInstance()实际上就是执行无参构造方法来创建该类的实例
* 方法2：通过Constructor的newInstance()方法
  + 先使用Class对象获取指定的Constructor对象
  + 再调用Constructor对象的newInstance()创建Class对象对应类的对象
  + 通过该方法可选择使用指定构造方法来创建对象

#### 【示例4】通过Class的newInstance()方法创建对象

|  |
| --- |
| **public class** TestConstructor1 {  **public static void** main(String[] args) **throws** Exception{  //不使用反射创建对象  //Dog dog = new Dog();  //使用反射创建对象  //1.获取类的完整路径字符串  String className = **"com.bjsxt.why.Dog"**;  //2.根据完整路径字符串获取Class对象信息  Class clazz = Class.forName(className);  //3.直接使用Class的方法创建对象  Object obj = clazz.newInstance();  System.**out**.println(obj.toString());  } } |

#### 【示例5】通过Constructor的newInstance()方法创建对象

|  |
| --- |
| **public class** TestConstructor2 {  **public static void** main(String[] args) **throws** Exception{  //不使用反射创建对象  //Dog dog = new Dog();  //使用反射创建对象  //1.获取类的完整路径字符串  String className = **"com.bjsxt.why.Dog"**;  //2.根据完整路径字符串获取Class对象信息  Class clazz = Class.forName(className);  //3.获取无参数构造方法  Constructor con = clazz.getConstructor();  //4.使用无参数构造方法来创建对象  Object obj = con.newInstance();  System.**out**.println(obj);  } } |

**调用有参数构造方法创建对象**

只能通过Constructor的newInstance()方法来创建对象

#### 【示例6】通过Constructor的newInstance()方法创建对象

|  |
| --- |
| **public class** TestConstructor3 {  **public static void** main(String[] args) **throws** Exception {  //不使用反射创建对象 // Dog dog = new Dog("旺财","黑色"); // System.out.println(dog);  //使用反射创建对象  //1.读取配置文件，或者类的完整路径字符串  String className = **"com.bjsxt.why.Dog"**;  //2.根据类的完整路径字符串获取Class信息  Class clazz = Class.forName(className);  //3.从Class信息中获取有参数构造方法  // Constructor con = clazz.getConstructor(String.class,String.class);  Constructor con = clazz.getDeclaredConstructor(String.**class**,String.**class**);//指定形参  //4.使用反射创建对象  //突破封装性的限制，即使是private、默认的也可以访问  con.setAccessible(**true**);   Object obj = con.newInstance(**"旺财1"**,**"黑色2"**);//传递实参  System.**out**.println(obj);  } } |

问题1：Exception in thread "main" java.lang.NoSuchMethodException: com.bjsxt. why.Dog.<init>(java.lang.String, java.lang.String)。

原因：getConstructor只能获取public方法，无法获取其他修饰符修饰的方法。

解决：调用getDeclaredConstructor()解决，可获取非public修饰的构造方法

问题2：Exception in thread "main" java.lang.IllegalAccessException: Class com. bjsxt. TestConstructor3 can not access a member of class com.bjsxt.Dog with modifiers "

原因：可以获取非public修饰的构造方法，不等于可以运行非public修饰的构造方法，受到了封装性的限制

解决：调用con.setAccessible(**true**);方法，可以突破封装性的限制。

**反射优点**

功能强大

1）编码时不知道具体的类型，可以使用反射动态操作

2) 突破封装的限制，即使private的成员也可以进行操作

**反射缺点：**

1).代码繁琐，可读性差

2).突破封装的限制，即使private的成员也可以进行操作（既是优点也是缺点）

### 本节作业

1. 理解反射的作用和使用场合
2. 理解Class类的含义和作用
3. 使用反射调用无参数和有参数构造方法创建对象

## 二、反射技术

### 2.1 使用反射操作属性

通过Class对象的getFields()或者getField()方法可以获得该类所包括的全部Field属性或指定Field属性。Field类提供了以下方法来访问属性

* + getXxx(Object obj)：获取obj对象该Field的属性值。此处的Xxx对应8个基本数据类型，如果该属性类型是引用类型则直接使用get(Object obj)
  + setXxx(Object obj,Xxx val)：将obj对象的该Field赋值val。此处的Xxx对应8个基本数据类型，如果该属性类型是引用类型则直接使用set(Object obj, Object val)
  + setAccessible(Boolean flag)：若flag为true，则取消属性的访问权限控制，即使private属性也可以进行访问

#### 【示例7】使用反射操作属性

|  |
| --- |
| **public class** TestField {  **public static void** main(String[] args) **throws** Exception{  //不使用反射操作属性 // Dog dog = new Dog(); // dog.nickName = "旺财"; // System.out.println(dog.nickName);  //使用反射操作属性 实际操作中使用反射直接操作属性也不多  //1.获取类的完整路径字符串  String className = **"com.bjsxt.why.Dog"**;  //2.得到类对象  Class clazz = Class.forName(className);  //3.使用反射创建对象  //Object dog = clazz.newInstance();  Object dog = clazz.getConstructor().newInstance();  //4.获取属性  Field f1 = clazz.getField(**"color"**);  //Field f2 = clazz.getField("age");  Field f2 = clazz.getDeclaredField(**"age"**);  //5.给属性赋值  f1.set(dog,**"黑色1"**); // dog.color ="黑色";  f2.setAccessible(**true**);//突破权限的控制  f2.set(dog,10);  //6.输出给属性  System.**out**.println(f1.get(dog)); //dog.color  System.**out**.println(f2.get(dog)); //dog.age  System.**out**.println(dog);  } } |

### 2.2 使用反射执行方法

* 通过Class对象的getMethods() 方法可以获得该类所包括的全部public方法, 返回值是Method[]
* 通过Class对象的getMethod()方法可以获得该类所包括的指定public方法, 返回值是Method
* 每个Method对象对应一个方法，获得Method对象后，可以调用其invoke() 来调用对应方法
* Object invoke(Object obj,Object [] args):obj代表当前方法所属的对象的名字，args代表当前方法的参数列表，返回值Object是当前方法返回值，即执行当前方法的结果。

#### 【示例8】使用反射执行方法

|  |
| --- |
| **public class** TestMethod {  **public static void** main(String[] args) **throws** Exception{  //不使用反射执行方法 // Dog dog = new Dog(); // dog.shout(); // int result = dog.add(10,20); // System.out.println(result);  //使用反射执行方法  //1.获取类的完整路径字符串  String className = **"com.bjsxt.why.Dog"**;  //2.得到类对象  Class clazz = Class.forName(className);  //3.使用反射创建对象  //Object dog = clazz.newInstance();  Object dog = clazz.getConstructor().newInstance();  //4.获取方法  Method m1 = clazz.getMethod(**"shout"**);  Method m2 = clazz.getMethod(**"add"**,**int**.**class**,**int**.**class**);  //5.使用反射执行方法  m1.invoke(dog);//dog.shout();  Object result = m2.invoke(dog,10,20);  System.**out**.println(result);  } } |

### 2.3 使用反射操作泛型

没有出现泛型之前，Java中的所有数据类型包括：

* primitive types：基本类型
* raw type：原始类型。不仅仅指平常所指的类，还包括数组、接口、注解、枚举等结构。

Class类的一个具体对象代表一个指定的原始类型和基本类型。

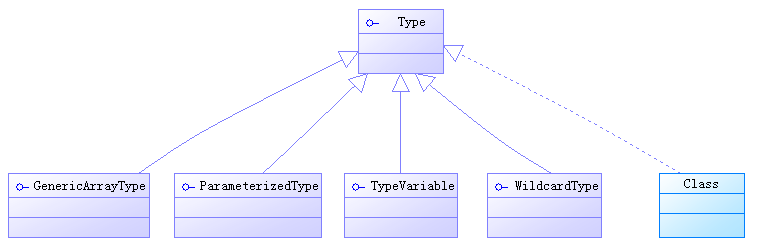
泛型出现之后，也就扩充了数据类型：

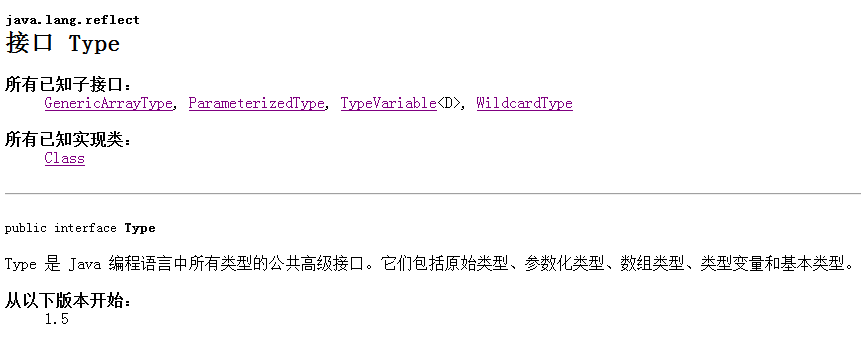
* parameterized types（参数化类型）：就是我们平常所用到的泛型List<T>、Map<K,V>的List和Map
* type variables（类型变量）：比如List<T>中的T等。（注意和参数化类型的区别）
* array types（数组类型）：并不是我们工作中所使用的数组String[] 、byte[]（这种都属于Class），而是带泛型的数组，比如List<T>[]，T[]
* WildcardType（泛型表达式类型 通配符类型）：例如List< ? extends Number>

Java采用泛型擦除机制来引入泛型。但是擦除的是方法体中局部变量上定义的泛型，在泛型类、泛型接口中定义的泛型，在成员变量、成员方法上定义的泛型，依旧会保存（可以理解为定义泛型信息保留，使用泛型信息擦除）。保留下来的信息可以通过反射获取。

另外一方面，Class类的一个具体对象代表一个指定的原始类型和基本类型，和泛型相关的新扩充进来的类型不好被统一到Class类中，否则会涉及到JVM指令集的修改，是很致命的。

为了能通过反射操作泛型，但是实现扩展性而不影响之前操作，Java就新增了ParameterizedType, TypeVariable, GenericArrayType, WildcardType几种类型来代表不能被归一到Class类中的类型但是又和原始类型齐名的类型。





#### 【示例9】使用反射获取泛型类型

|  |
| --- |
| **public class** TestGeneric { **public void** method1(Map<Integer, Student> map, List<Student> list, String str) {}  **public** Map<Integer, Student> method2() { **return null**; }  **public static void** main(String[] args) **throws** NoSuchMethodException {  Class clazz = TestGeneric.**class**;  Method method1 =  clazz.getMethod(**"method1"**, Map.**class**, List.**class**, String.**class**);  *//获取参数类型（不带泛型）* Class[] paramTypes = method1.getParameterTypes();  **for** (Class clazz2 : paramTypes) {  System.***out***.println(clazz2);  }  *//获取参数类型（带泛型）* Type[] types = method1.getGenericParameterTypes();  System.***out***.println(types.**length**);  **for** (Type type : types) {  System.***out***.println(type);  **if** (type **instanceof** ParameterizedType) {  Type typeArgs[] =  ((ParameterizedType) type).getActualTypeArguments();  **for** (Type arg : typeArgs) {  System.***out***.println(**"\t"** + arg);  }  }  }  *//获取返回值类型(不带泛型)* Method method2 = clazz.getMethod(**"method2"**);  Class returnType = method2.getReturnType();  System.***out***.println(returnType);  *//获取返回值类型(带泛型)* Type returnType2 = method2.getGenericReturnType();  Type[] typeArgs = ((ParameterizedType) returnType2).getActualTypeArguments();  **for** (Type type : typeArgs) {  System.***out***.println(**"\t"** + type);  }  *//获取数组元素的类型  //int [] arr = new int[10];* Student[] arr = **new** Student[10];  Class componentType = arr.getClass().getComponentType();  System.***out***.println(componentType);  } } |

给集合添加泛型后，可以限制元素类型，提高安全性。使用反射还可以突破泛型的限制；

#### 【示例10】使用反射突破泛型的限制

|  |
| --- |
| **public class** TestGeneric {  **public static void** main(String[] args) **throws** Exception {  //不是反射  List<String> list = **new** ArrayList<String>();  list.add(**"Java"**);  list.add(**"MySQL"**);  list.add(**"MyBatis"**); // list.add(new Date()); // list.add(100);  //使用反射调用add  //先得到List的结构信息Class  //Class clazz = Class.forName("java.util.ArrayList");  //Class clazz = ArrayList.class;  Class clazz = list.getClass();  //获取add方法  Method method = clazz.getMethod(**"add"**,Object.**class**);  //使用反射调用add方法  method.invoke(list,100);  method.invoke(list,**new** Date());  System.**out**.println(list);  } } |

### 本节作业

1. 使用反射操作属性
2. 使用反射执行方法
3. 使用反射获取参数和返回值类型的泛型
4. 使用反射突破泛型限制

## 三、注解

### 3.1 认识注解

Annotation ，JDK1.5新提供的技术

我们在编程中经常会使用到注解，作用有：

1）编译检查：比如@SuppressWarnings, @Deprecated 和 @Override 都具有编译检查作用

2）替代配置文件：使用反射来读取注解信息

目前大部分框架(如Spring)都使用了注解简化代码并提高编码的效率（使用注解之前使用的xml进行配置）

注解其实就是代码里的特殊标记，它用于替代配置文件：传统方式通过配置文件告诉类如何运行，有了注解技术后，开发人员可以通过注解告诉类如何运行。

在Java技术里注解的典型应用是：可以通过反射技术去得到类里面的注解，以决定怎么去运行类。 注解可以标记在包、类、属性、方法，方法参数以及局部变量上，且同一个地方可以同时标记多个注解。

注解可以在编译（source），类加载（class），运行时（runtime）被读取，并执行相应的处理，以便于其他工具补充信息或者进行部署

### 3.2 内置注解

主要有三个内置注解

* @Override - 检查该方法是否是重载方法。如果发现其父类，或者是引用的接口中并没有该方法时，会报编译错误。
* @Deprecated - 标记过时方法。如果使用该方法，会报编译警告。
* @SuppressWarnings - 指示编译器去忽略注解中声明的警告。



从 Java 7 开始，额外添加了 3 个注解:

* @SafeVarargs - Java 7 开始支持，忽略任何使用参数为泛型变量的方法或构造函数调用产生的警告。
* @FunctionalInterface - Java 8 开始支持，标识一个匿名函数或函数式接口。
* @Repeatable - Java 8 开始支持，标识某注解可以在同一个声明上使用多次。

#### 【示例11】认识内置注解

|  |
| --- |
| @SuppressWarnings(value={**"all"**}) **public class** Student **implements** Comparable<Student>, Serializable {  @Override  **public int** compareTo(Student o) {*//implements a method* **return** 0;  }  @Override  **public** String toString() { *//override a method* **return super**.toString();  }  **public static void** main(String[] args) {  Date date = **new** Date();  System.***out***.println(date.toLocaleString());  Student stu = **new** Student();  stu.method1();  List list = **new** ArrayList();  }  @Deprecated  **public void** method1(){  System.***out***.println(**"=========="**);  }  **public void** method2(){  Date date = **new** Date();  System.***out***.println(date.toLocaleString());  } } **class** TestStudent{  @SuppressWarnings(value=**"deprecation"**)  **public static void** main(String[] args) {  Student stu = **new** Student();  stu.method1();  } } |

### 3.3 元注解

元注解是指注解的注解，在JDK 1.5中提供了4个标准的用来对注解类型进行注解的注解类。可以使用这4个元注解来对我们自定义的注解类型进行注解

1. @Retention用来约束注解的生命周期，分别有三个值，源码级别（source），类文件级别（class）或者运行时级别（runtime），**若没有 @Retention，则默认是 RetentionPolicy.CLASS**。其含有如下：

* SOURCE：注解将被编译器丢弃（该类型的注解信息只会保留在源码里，源码经过编译后，注解信息会被丢弃，不会保留在编译好的class文件里）
* CLASS：注解在class文件中可用，但会被VM丢弃（该类型的注解信息会保留在源码里和class文件里，在执行的时候，不会加载到虚拟机中）。
* RUNTIME：注解信息将在运行期(JVM)也保留，因此可以通过反射机制读取注解的信息（源码、class文件和执行的时候都有注解的信息），如SpringMvc中的@Controller、@Autowired、@RequestMapping等。

1. @Target -用来约束注解可以应用的地方（如方法、类或字段），其中ElementType是枚举类型。**若没有 @Target，则该 Annotation 可以用于任何地方**。

|  |
| --- |
| public enum ElementType {  /\*\*标明该注解可以用于类、接口（包括注解类型）或enum声明\*/  TYPE,  /\*\* 标明该注解可以用于字段(域)声明，包括enum实例 \*/  FIELD,  /\*\* 标明该注解可以用于方法声明 \*/  METHOD,  /\*\* 标明该注解可以用于参数声明 \*/  PARAMETER,  /\*\* 标明注解可以用于构造函数声明 \*/  CONSTRUCTOR,  /\*\* 标明注解可以用于局部变量声明 \*/  LOCAL\_VARIABLE,  /\*\* 标明注解可以用于注解声明(应用于另一个注解上)\*/  ANNOTATION\_TYPE,  /\*\* 标明注解可以用于包声明 \*/  PACKAGE,  /\*\*  \* 标明注解可以用于类型参数声明（1.8新加入）  \* @since 1.8  \*/  TYPE\_PARAMETER,  /\*\*  \* 类型使用声明（1.8新加入)  \* @since 1.8  \*/  TYPE\_USE  } |

1. @Documented - 标记这些注解是否包含在用户文档中。
2. @Inherited - 指示注解类型被自动继承。当@InheritedAnno注解加在某个类A上时，假如类B继承了A，则B也会带上该注解。

### 本节作业

1. 注解的作用
2. 内置注解及其作用
3. 元注解及其作用

## 四、注解

### 4.1 自定义注解

#### 【示例12】自定义注解

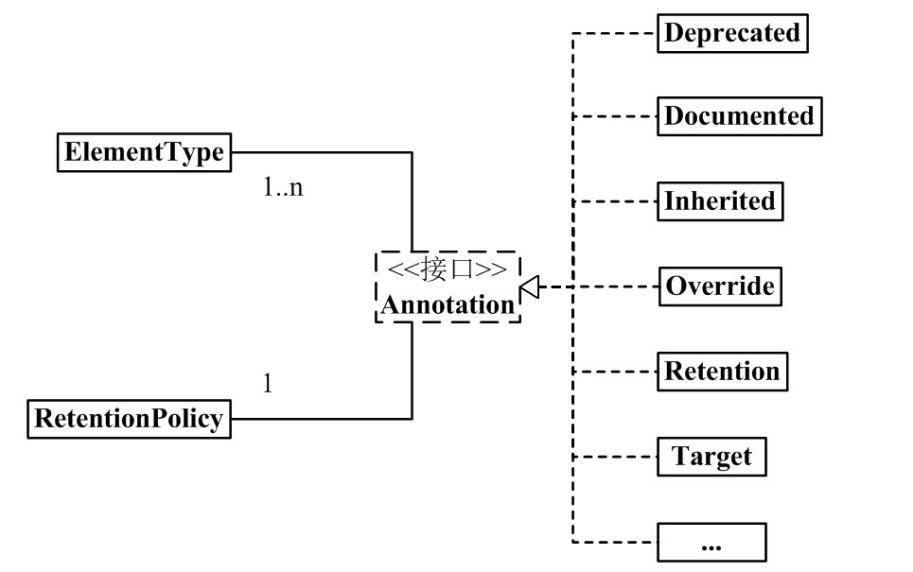
|  |
| --- |
| @Retention(RetentionPolicy.***RUNTIME***) @Target(value= {ElementType.***METHOD***,ElementType.***TYPE***}) **public** @**interface** MyAnnoation {  **int** id() **default** 0;  String name() **default ""**;  **double** [] scoreArr() **default** {}; }  **public** @**interface** MyAnnotation2 {  *//如果只有一个配置参数，一般命名为value* String value();  }  @MyAnnotation2(**"bjsxt"**) @MyAnnoation **public class** TestAnnotation {  @MyAnnoation(id=5,name=**"张三"**,scoreArr = {78,89,34})  **public static void** main(String[] args) {  }  @MyAnnotation2(value=**"sxt"**)  **public void** method1(){  } } |

总结：

* 定义注解的关键字是@interface
* 自定义注解中可以定义多个配置参数，不是成员方法,不是成员变量；说明参数的名称，以及参数值的类型
* 如果只有一个配置参数，一般命名为value
* 如果配置参数是value，并且只有一个配置参数，value可以省略

注意：

* 定义注解时，意味着它实现了 java.lang.annotation.Annotation 接口，即该注解就是一个Annotation。
* 和我们通常 implements实现接口的方法不同。Annotation 接口的实现细节都由编译器完成。通过 @interface 定义注解后，该注解不能继承其他注解或接口。
* 注解常见的API及其关系如下



### 4.2 使用反射读取注解

目前大部分框架(如Spring、MyBatis、SpringMVC)都使用了注解简化代码并提高编码的效率（使用注解之前使用的xml进行配置）。

ORM，Object-Relationl Mapping，对象关系映射，它的作用是在关系型数据库和对象之间作一个映射，这样我们在具体的操作数据库的时候，只要像平时操作对象一样操作它就可以了，ORM框架会根据映射完成对数据库的操作，就不需要再去和复杂的SQL语句打交道了。常用的ORM框架有MyBatis和Hibernate。



在ORM中，数据库表对应Java实体类，数据库表的字段对应Java实体类的成员变量，数据库表的记录对应Java实体类的对象。

其实ORM可以借助注解来进行映射，并使用反射读取注解信息完成最终的操作。

#### 【示例13】模拟实现MyBatis的注解并使用反射读取

|  |
| --- |
| @Retention(value = RetentionPolicy.**RUNTIME**) @Target(value = ElementType.**TYPE**) **public** @**interface** Table {  String value(); }  @Retention(value = RetentionPolicy.**RUNTIME**) @Target(value = ElementType.**FIELD**) **public** @**interface** Column {  String columnName(); //列名  String columnType(); //列类型  **int** length(); //列长度  **int** precision() **default** 0;//小数位数 }  @Table(value = **"t\_student"**) **public class** Student {  @Column(columnName=**"id"**,columnType = **"int"**,length=6)  **private int id**;  @Column(columnName = **"sname"**,columnType = **"varchar"**,length = 10)  **private** String **name**;  @Column(columnName = **"score"**,columnType = **"double"**,length = 4,precision = 1)  **private double score**; }  **public class** TestORM {  **public static void** main(String[] args) **throws** Exception {  String className =**"com.bjsxt.annotation3.Student"**;  Class clazz = Class.forName(className);  //获取类的所有注解  Annotation [] annotations = clazz.getAnnotations();  **for** (Annotation annotation:annotations ) {  System.**out**.println(annotation);  }  //获取类的指定注解  Table annotation =(Table) clazz.getAnnotation(Table.**class**);  System.**out**.println(annotation);  System.**out**.println(annotation.value());  //获取id属性的注解  Field idField = clazz.getDeclaredField(**"id"**);  Column idColumn =(Column)idField.getAnnotation(Column.**class**);  System.**out**.println(idColumn.columnName());  System.**out**.println(idColumn.columnType());  System.**out**.println(idColumn.length());  System.**out**.println(idColumn.precision());   //获取name属性的注解  //获取score属性的注解  //拼接create DDL语句，通过JDBC创建数据库表 excuteUpdate()  //根据Student类id、name、score的值，对T\_Student表进行添 //加、修改、删除操作；将T\_Student表的一条记录的各列的数据取出来，存//入一个Student对象中  } } |

### 本节作业

1. 自定义注解
2. 模拟实现ORM框架的注解及其注解信息的读取