# Java 注解完全解析

## 说明：

关于注解首先引入官方文档的一句话：Java 注解用于为 Java 代码提供元数据。作为元数据，注解不直接影响你的代码执行，但也有一些类型的注解实际上可以用于这一目的。Java 注解是从 Java5 开始添加到 Java 的。看完这句话也许你还是一脸懵逼，接下我将从注解的定义、元注解、注解属性、自定义注解、注解解析JDK 提供的注解这几个方面再次了解注解（Annotation）

## 注解的定义

日常开发中新建Java类，我们使用class、interface比较多，而注解和它们一样，也是一种类的类型，他是用的修饰符为 @interface

## 注解类的写法

我们新建一个注解MyTestAnnotation

**public** **@interface** MyTestAnnotation {

}

接着我们就可以在类或者方法上作用我们刚刚新建的注解

@MyTestAnnotation

**public** **class** Test {

@MyTestAnnotation

**public** **static** **void** main(String[] args) {

}

}

以上我们只是了解了注解的写法，但是我们定义的注解中还没写任何代码，现在这个注解毫无意义，要如何使注解工作呢？接下来我们接着了解元注解。

## 元注解

元注解顾名思义我们可以理解为注解的注解，它是作用在注解中，方便我们使用注解实现想要的功能。元注解分别有@Retention、 @Target、 @Document、 @Inherited和@Repeatable（JDK1.8加入）五种。

### @Retention

1、Retention英文意思有保留、保持的意思，它表示注解存在阶段是保留在源码（编译期），字节码（类加载）或者运行期（JVM中运行）。在@Retention注解中使用枚举RetentionPolicy来表示注解保留时期

2、@Retention(RetentionPolicy.SOURCE)，注解仅存在于源码中，在class字节码文件中不包含

3、@Retention(RetentionPolicy.CLASS)， 默认的保留策略，注解会在class字节码文件中存在，但运行时无法获得

4、@Retention(RetentionPolicy.RUNTIME)， 注解会在class字节码文件中存在，在运行时可以通过反射获取到

5、如果我们是自定义注解，则通过前面分析，我们自定义注解如果只存着源码中或者字节码文件中就无法发挥作用，而在运行期间能获取到注解才能实现我们目的，所以自定义注解中肯定是使用 @Retention(RetentionPolicy.RUNTIME)

**import** java.lang.annotation.Retention;

**import** java.lang.annotation.RetentionPolicy;

@Retention(RetentionPolicy.***RUNTIME***)

**public** **@interface** MyTestAnnotation {

}

### @Target

Target的英文意思是目标，这也很容易理解，使用@Target元注解表示我们的注解作用的范围就比较具体了，可以是类，方法，方法参数变量等，同样也是通过枚举类ElementType表达作用类型

1、@Target(ElementType.TYPE) 作用接口、类、枚举、注解

2、@Target(ElementType.FIELD) 作用属性字段、枚举的常量

3、@Target(ElementType.METHOD) 作用方法

4、@Target(ElementType.PARAMETER) 作用方法参数

5、@Target(ElementType.CONSTRUCTOR) 作用构造函数

6、@Target(ElementType.LOCAL\_VARIABLE)作用局部变量

7、@Target(ElementType.ANNOTATION\_TYPE)作用于注解（@Retention注解中就使用该属性）

8、@Target(ElementType.PACKAGE) 作用于包

9、@Target(ElementType.TYPE\_PARAMETER) 作用于类型泛型，即泛型方法、泛型类、泛型接口 （jdk1.8加入）

10、@Target(ElementType.TYPE\_USE) 类型使用.可以用于标注任意类型除了 class （jdk1.8加入）

一般比较常用的是ElementType.TYPE类型

**import** java.lang.annotation.ElementType;

**import** java.lang.annotation.Retention;

**import** java.lang.annotation.RetentionPolicy;

**import** java.lang.annotation.Target;

@Retention(RetentionPolicy.***RUNTIME***)

@Target(ElementType.***TYPE***)

**public** **@interface** MyTestAnnotation {

}

### @Documented

Document的英文意思是文档。它的作用是能够将注解中的元素包含到 Javadoc 中去。

### @Inherited

Inherited的英文意思是继承，但是这个继承和我们平时理解的继承大同小异，一个被@Inherited注解了的注解修饰了一个父类，如果他的子类没有被其他注解修饰，则它的子类也继承了父类的注解。

下面我们来看个@Inherited注解例子

/\*\*自定义注解\*/

**import** java.lang.annotation.ElementType;

**import** java.lang.annotation.Inherited;

**import** java.lang.annotation.Retention;

**import** java.lang.annotation.RetentionPolicy;

**import** java.lang.annotation.Target;

@Retention(RetentionPolicy.***RUNTIME***)//运行期间能获取到注解

@Target({ElementType.***TYPE***, ElementType.***METHOD***, ElementType.***FIELD***})//作用到类、方法、属性上

@Inherited//可被继承

**public** **@interface** MyTestAnnotation {

}

/\*\*父类标注自定义注解\*/

@MyTestAnnotation

**public** **class** Father {

}

/\*\*子类\*/

**public** **class** Son **extends** Father{

/\*\*测试子类获取父类自定义注解

\* **@throws** Exception \*/

**public** **static** **void** main(String[] args) **throws** Exception {

//获取Son的class对象

Class<?> sonClass = Class.*forName*("annotation.Son");

// 获取Son类上的注解MyTestAnnotation可以执行成功

MyTestAnnotation annotation = sonClass.getAnnotation(MyTestAnnotation.**class**);

/\*

\* 当MyTestAnnotation有@Inherited注解时结果为@annotation.MyTestAnnotation()

\* 当MyTestAnnotation没有@Inherited注解时结果为null

\*/

System.***out***.println(annotation);

}

}

### @Repeatable

Repeatable的英文意思是可重复的。顾名思义说明被这个元注解修饰的注解可以同时作用一个对象多次，但是每次作用注解又可以代表不同的含义。

下面我们看一个人玩游戏的例子

/\*\*一个人喜欢玩游戏，他喜欢玩英雄联盟，绝地求生，极品飞车，尘埃4等，则我们需要定义一个人的注解，他属性代表喜欢玩游戏集合，一个游戏注解，游戏属性代表游戏名称\*/

/\*\*玩家注解\*/

@Documented

@Retention(RetentionPolicy.***RUNTIME***)

@Target(ElementType.***TYPE***)

**public** **@interface** People {

Game[] value() ;

}

/\*\*游戏注解\*/

@Repeatable(People.**class**)

@Retention(RetentionPolicy.***RUNTIME***)

@Target(ElementType.***TYPE***)

**public** **@interface** Game {

String value() **default** "";

}

/\*\*玩游戏类\*/

@Game(value = "LOL")

@Game(value = "PUBG")

@Game(value = "NFS")

@Game(value = "Dirt4")

**public** **class** PlayGame {

}

通过上面的例子，你可能会有一个疑问，游戏注解中括号的变量是啥，其实这和游戏注解中定义的属性对应。接下来我们继续学习注解的属性。

## 注解的属性

通过上一小节@Repeatable注解的例子，我们说到注解的属性。注解的属性其实和类中定义的变量有异曲同工之处，只是注解中的变量都是成员变量（属性），并且注解中是没有方法的，只有成员变量，变量名就是使用注解括号中对应的参数名，变量返回值注解括号中对应参数类型。相信这会你应该会对上面的例子有一个更深的认识。而@Repeatable注解中的变量则类型则是对应Annotation（接口）的泛型Class。

/\*\*注解Repeatable源码\*/

@Documented

@Retention(RetentionPolicy.***RUNTIME***)

@Target(ElementType.***ANNOTATION\_TYPE***)

**public** **@interface** Repeatable {

/\*\*

\* Indicates the <em>containing annotation type</em> for the

\* repeatable annotation type.

\* **@return** the containing annotation type

\*/

Class<? **extends** Annotation> value();

}

## 注解的本质

注解的本质就是一个Annotation接口

/\*\*Annotation接口源码\*/

**public** **interface** Annotation {

**boolean** equals(Object obj);

**int** hashCode();

Class<? **extends** Annotation> annotationType();

}

通过以上源码，我们知道注解本身就是Annotation接口的子接口，也就是说注解中其实是可以有属性和方法，但是接口中的属性都是static final的，对于注解来说没什么意义，而我们定义接口的方法就相当于注解的属性，也就对应了前面说的为什么注解只有属性成员变量，其实他就是接口的方法，这就是为什么成员变量会有括号，不同于接口我们可以在注解的括号中给成员变量赋值。

## 注解属性类型

注解属性类型可以有以下列出的类型

1.基本数据类型

2.String

3.枚举类型

4.注解类型

5.Class类型

6.以上类型的一维数组类型

## 注解成员变量赋值

如果注解又多个属性，则可以在注解括号中用“，”号隔开分别给对应的属性赋值，如下例子，注解在父类中赋值属性

@Documented

@Inherited

@Retention(RetentionPolicy.***RUNTIME***)

@Target(ElementType.***TYPE***)

**public** **@interface** MyTestAnnotation {

String name() **default** "mao";

**int** age() **default** 18;

}

@MyTestAnnotation(name = "father",age = 50)

**public** **class** Father {

}

## 获取注解属性

前面我们说了很多注解如何定义，放在哪，现在我们可以开始学习注解属性的提取了，这才是使用注解的关键，获取属性的值才是使用注解的目的。

如果获取注解属性，当然是反射啦，主要有三个基本的方法

/\*\*是否存在对应 Annotation 对象\*/

**public** **boolean** isAnnotationPresent(Class<? **extends** Annotation> annotationClass) {

**return** GenericDeclaration.**super**.isAnnotationPresent(annotationClass);

}

/\*\*获取 Annotation 对象\*/

**public** <A **extends** Annotation> A getAnnotation(Class<A> annotationClass) {

Objects.requireNonNull(annotationClass);

**return** (A) annotationData().annotations.get(annotationClass);

}

/\*\*获取所有 Annotation 对象数组\*/

**public** Annotation[] getAnnotations() {

**return** AnnotationParser.toArray(annotationData().annotations);

}

下面结合前面的例子，我们来获取一下注解属性，在获取之前我们自定义的注解必须使用元注解@Retention(RetentionPolicy.***RUNTIME***)

### 1、注解：

#### *一、MyTestAnnotation*

**import** java.lang.annotation.ElementType;

**import** java.lang.annotation.Inherited;

**import** java.lang.annotation.Retention;

**import** java.lang.annotation.RetentionPolicy;

**import** java.lang.annotation.Target;

/\*\*自定义注解\*/

@Retention(RetentionPolicy.***RUNTIME***)//运行期间能获取到注解

@Target({ElementType.***TYPE***, ElementType.***METHOD***, ElementType.***FIELD***})//作用到类、方法、属性上

@Inherited//可被继承

**public** **@interface** MyTestAnnotation {

String name() **default** "mao";

**int** age() **default** 18;

}

#### *二、People*

**import** java.lang.annotation.ElementType;

**import** java.lang.annotation.Retention;

**import** java.lang.annotation.RetentionPolicy;

**import** java.lang.annotation.Target;

/\*\*

\* 一个人喜欢玩游戏，他喜欢玩英雄联盟，绝地求生，极品飞车，尘埃4等，则我们需要定义一个人的注解，他属性代表喜欢玩游戏集合，

\* 一个游戏注解，游戏属性代表游戏名称\*/

/\*\*玩家注解\*/

@Retention(RetentionPolicy.***RUNTIME***)

@Target({ElementType.***TYPE***, ElementType.***METHOD***})

**public** **@interface** People {

Game[] value();

}

#### *三、Game*

**import** java.lang.annotation.ElementType;

**import** java.lang.annotation.Repeatable;

**import** java.lang.annotation.Retention;

**import** java.lang.annotation.RetentionPolicy;

**import** java.lang.annotation.Target;

/\*\*游戏注解\*/

@Repeatable(People.**class**)

@Retention(RetentionPolicy.***RUNTIME***)

@Target({ElementType.***TYPE***, ElementType.***METHOD***})

**public** **@interface** Game {

String value() **default** "";

}

#### *四、Age*

**import** java.lang.annotation.ElementType;

**import** java.lang.annotation.Retention;

**import** java.lang.annotation.RetentionPolicy;

**import** java.lang.annotation.Target;

@Target(ElementType.***FIELD***)

@Retention(RetentionPolicy.***RUNTIME***)

**public** **@interface** Age {

**int** value() **default** 18;

}

### 2、测试对象

/\*\*父类标注自定义注解\*/

@MyTestAnnotation(name="zhang", age=20)

**public** **class** Father {

@Age(value = 23)

**private** **int** age;

@Game(value = "LOL")

@Game(value = "PUBG")

@Game(value = "NFS")

@Game(value = "Dirt4")

**public** **void** play(){

}

}

### 3、测试类

**import** java.lang.reflect.Field;

**import** java.lang.reflect.Method;

**public** **class** Test {

**public** **static** **void** main(String[] args) **throws** Exception {

/\*\*

\* 获取类注解属性

\*/

Class<?> fatherClass = Class.*forName*("annotation.Father");

**boolean** annotationPresent = fatherClass.isAnnotationPresent(MyTestAnnotation.**class**);

**if**(annotationPresent){

MyTestAnnotation annotation = fatherClass.getAnnotation(MyTestAnnotation.**class**);

String name = annotation.name();

**int** age = annotation.age();

System.***out***.println(name);

System.***out***.println(age);

}

/\*\*

\* 获取属性值注解属性

\* \*/

Field age = fatherClass.getDeclaredField("age");

annotationPresent = age.isAnnotationPresent(Age.**class**);

**if**(annotationPresent){

Age annotation = age.getAnnotation(Age.**class**);

System.***out***.println(annotation.value());

}

/\*\*

\* 获取方法注解属性

\*/

Method play = fatherClass.getDeclaredMethod("play");

annotationPresent = play.isAnnotationPresent(People.**class**);

**if**(annotationPresent){

People people = play.getAnnotation(People.**class**);

Game[] value = people.value();

**for** (Game game : value) {

System.***out***.println(game.value());

}

}

}

}

### 4、运行结果：

zhang

20

23

LOL

PUBG

NFS

Dirt4

## JDK 提供的注解

| 注解 | 作用 | 注意事项 |
| --- | --- | --- |
| @Override | 它是用来描述当前方法是一个重写的方法，在编译阶段对方法进行检查 | jdk1.5中它只能描述继承中的重写，jdk1.6中它可以描述接口实现的重写,也能描述类的继承的重写 |
| @Deprecated | 它是用于描述当前方法是一个过时的方法 | 无 |
| @SuppressWarnings | 对程序中的警告去除。 | 无 |

## 注解作用与应用

现在我们再次回头看看开头官方文档的那句描述

Java 注解用于为 Java 代码提供元数据。作为元数据，注解不直接影响你的代码执行，但也有一些类型的注解实际上可以用于这一目的。

经过我们前面的了解，注解其实是个很方便的东西，它存活的时间，作用的区域都可以由你方便设置，只是你用注解来干嘛的问题

## 使用注解进行参数配置

下面我们看一个银行转账的例子，假设银行有个转账业务，转账的限额可能会根据汇率的变化而变化，我们可以利用注解灵活配置转账的限额，而不用每次都去修改我们的业务代码。

**import** java.lang.reflect.Method;

/\*\*转账处理业务类\*/

**public** **class** BankService {

**public** **static** **void** main(String[] args) {

*transferMoney*(10000);//转账金额为:10000.0，转账成功

*transferMoney*(20000);//转账金额大于限额，转账失败

}

/\*\*

\* **@param** money 转账金额

\*/

@BankTransferMoney(maxMoney = 15000)

**public** **static** **void** transferMoney(**double** money){

System.***out***.println(*processAnnotationMoney*(money));

}

**private** **static** String processAnnotationMoney(**double** money) {

**try** {

Class<?> bankService = Class.*forName*("annotation.BankService");

Method transferMoney = bankService.getDeclaredMethod("transferMoney", **double**.**class**);

**boolean** annotationPresent = transferMoney.isAnnotationPresent(BankTransferMoney.**class**);

**if**(annotationPresent){

BankTransferMoney annotation = transferMoney.getAnnotation(BankTransferMoney.**class**);

**double** maxMoney = annotation.maxMoney();

**if**(money < maxMoney){

**return**"转账金额为:"+money+"，转账成功";

}**else**{

**return** "转账金额大于限额，转账失败";

}

}

} **catch** (Exception e) {

e.printStackTrace();

}

**return** "转账处理失败";

}

}

通过上面的例子，只要汇率变化，我们就改变注解的配置值就可以直接改变当前最大限额。

## 第三方框架的应用

作为一个Android 开发者，平常我们所使用的第三方框架ButterKnife，Retrofit2，Dagger2等都有注解的应用，如果我们要了解这些框架的原理，则注解的基础知识则是必不可少的。

## 注解的作用

提供信息给编译器： 编译器可以利用注解来检测出错误或者警告信息，打印出日志。

编译阶段时的处理： 软件工具可以用来利用注解信息来自动生成代码、文档或者做其它相应的自动处理。

运行时处理： 某些注解可以在程序运行的时候接受代码的提取，自动做相应的操作。

正如官方文档的那句话所说，注解能够提供元数据，转账例子中处理获取注解值的过程是我们开发者直接写的注解提取逻辑，处理提取和处理 Annotation 的代码统称为 APT（Annotation Processing Tool)。上面转账例子中的processAnnotationMoney方法就可以理解为APT工具类。

## 模拟Spring的注解

在使用Spring的时候可以发现，在对象的属性上加一个@Value注解可以将注解里的值设置到对象中，那么这是怎么实现的呢，让我们通过例子来演示实现细节：

### **一、Value注解**：

**import** java.lang.annotation.ElementType;

**import** java.lang.annotation.Retention;

**import** java.lang.annotation.RetentionPolicy;

**import** java.lang.annotation.Target;

@Retention(RetentionPolicy.***RUNTIME***)

@Target(ElementType.***FIELD***)

**public** **@interface** Value {

String value();

}

### **二、Student对象**

**public** **class** Student {

/\*\* 学号 \*/

@Value(value = "20160001")

**private** **long** id;

/\*\* 名字 \*/

@Value(value = "孔明")

**private** String name;

/\*\* 年龄 \*/

@Value(value = "20")

**private** **int** age;

/\*\* 年级 \*/

@Value(value = "1")

**private** **int** grade;

/\*\* 专业 \*/

@Value(value = "土木工程")

**private** String major;

/\*\* 学校 \*/

@Value(value = "武汉大学")

**private** String school;

**public** **long** getId() {

**return** id;

}

**public** **void** setId(**long** id) {

**this**.id = id;

}

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

**public** **int** getAge() {

**return** age;

}

**public** **void** setAge(**int** age) {

**this**.age = age;

}

**public** **int** getGrade() {

**return** grade;

}

**public** **void** setGrade(**int** grade) {

**this**.grade = grade;

}

**public** String getMajor() {

**return** major;

}

**public** **void** setMajor(String major) {

**this**.major = major;

}

**public** String getSchool() {

**return** school;

}

**public** **void** setSchool(String school) {

**this**.school = school;

}

**public** Student() {

**super**();

}

**public** Student(**long** id, String name, **int** age, **int** grade, String major,

String school) {

**super**();

**this**.id = id;

**this**.name = name;

**this**.age = age;

**this**.grade = grade;

**this**.major = major;

**this**.school = school;

}

@Override

**public** String toString() {

**return** "Student [id=" + id + ", name=" + name + ", age=" + age

+ ", grade=" + grade + ", major=" + major + ", school="

+ school + "]";

}

}

### **三、注解具体实现类：**

**import** java.lang.reflect.Field;

**import** java.lang.reflect.Method;

**import** java.text.SimpleDateFormat;

**import** java.util.Date;

**import** java.util.Locale;

**public** **class** SpringAnnotation {

/\*\*

\* 通过类名创建对象进行赋值

\* \*/

**public** **static** Object getBean(String clazzName) {

**try** {

**return** *getBean*(Class.*forName*(clazzName));

} **catch** (Exception e) {

e.printStackTrace();

}

**return** **null**;

}

/\*\*

\* 通过对象进行赋值

\* \*/

**public** **static** Object getBean(Object bean) {

**return** *getBean*(bean.getClass());

}

/\*\*

\* 通过class进行赋值

\* \*/

**public** **static** Object getBean(Class<?> clazz) {

Object bean = **null**;

**try** {

bean = clazz.newInstance();

// 反射获取对象的所有属性

Field[] fields = clazz.getDeclaredFields();

// 反射获取对象的所有方法

Method[] methods = clazz.getDeclaredMethods();

**for** (Field field : fields) {

// 判断属性是否添加了Value注解

**if** (field.isAnnotationPresent(Value.**class**)) {

// 匹配属性对应的set方法名称

String fieldSetName = *parSetName*(field.getName());

**if** (!*checkSetMet*(fieldSetName, methods))

**continue**;

// 获取注解的值

String value = field.getAnnotation(Value.**class**).value();

// 获取属性的类型名称

String fieldType = field.getType().getSimpleName();

// 获取到属性相应的set方法

Method m = clazz.getDeclaredMethod(fieldSetName,

field.getType());

// 进行反射的操作，将值注入到属性中

bean = *invoke*(m, field, fieldType, value, bean);

}

}

} **catch** (Exception e) {

e.printStackTrace();

}

**return** bean;

}

/\*\*

\* 转换类型并且将值注入到属性中

\* \*/

**private** **static** Object invoke(Method m, Field field, String fieldType,

String value, Object bean) **throws** Exception {

**if** ("String".equals(fieldType)) {

m.invoke(bean, value);

} **else** **if** ("Integer".equals(fieldType) || "int".equals(fieldType)) {

Integer intval = Integer.*parseInt*(value);

m.invoke(bean, intval);

} **else** **if** ("Long".equalsIgnoreCase(fieldType)) {

Long temp = Long.*parseLong*(value);

m.invoke(bean, temp);

} **else** **if** ("Short".equalsIgnoreCase(fieldType)) {

**short** temp = Short.*parseShort*(value);

m.invoke(bean, temp);

} **else** **if** ("Byte".equalsIgnoreCase(fieldType)) {

**byte** temp = Byte.*parseByte*(value);

m.invoke(bean, temp);

} **else** **if** ("Float".equalsIgnoreCase(fieldType)) {

**float** temp = Float.*parseFloat*(value);

m.invoke(bean, temp);

} **else** **if** ("Double".equalsIgnoreCase(fieldType)) {

Double temp = Double.*parseDouble*(value);

m.invoke(bean, temp);

} **else** **if** ("Boolean".equalsIgnoreCase(fieldType)) {

Boolean temp = Boolean.*parseBoolean*(value);

m.invoke(bean, temp);

} **else** **if** ("Character".equals(fieldType) || "char".equals(fieldType)) {

**if** (value.length() > 1) {

**throw** **new** RuntimeException("field [" + field.getName()

+ "]'s value [" + value + "] is not a Character!");

}

**char** temp = value.toCharArray()[0];

m.invoke(bean, temp);

} **else** **if** ("Date".equals(fieldType)) {

Date temp = *parseDate*(value);

m.invoke(bean, temp);

} **else** {

System.***out***.println("not supper type" + fieldType);

}

**return** bean;

}

/\*\*

\* 格式化属性的set方法

\* \*/

**private** **static** String parSetName(String fieldName) {

**if** (**null** == fieldName || "".equals(fieldName)) {

**return** **null**;

}

**return** "set" + fieldName.substring(0, 1).toUpperCase()

+ fieldName.substring(1);

}

/\*\*

\* 判断是否有属性对应的set方法

\* \*/

**private** **static** **boolean** checkSetMet(String fieldSetName, Method[] methods) {

**for** (Method method : methods) {

**if** (fieldSetName.equals(method.getName()))

**return** **true**;

}

**return** **false**;

}

/\*\*

\* 格式化日期

\* \*/

**private** **static** Date parseDate(String datestr) {

**if** (**null** == datestr || "".equals(datestr)) {

**return** **null**;

}

**try** {

String fmtstr = **null**;

**if** (datestr.indexOf('-') > 0 && datestr.indexOf(':') > 0) {

fmtstr = "yyyy-MM-dd HH:mm:ss";

} **else** **if** (datestr.indexOf('/') > 0 && datestr.indexOf(':') > 0) {

fmtstr = "yyyy/MM/dd HH:mm:ss";

} **else** **if** (datestr.indexOf('-') > 0 && datestr.indexOf(':') < 0) {

fmtstr = "yyyy-MM-dd";

} **else** **if** (datestr.indexOf('/') > 0 && datestr.indexOf(':') < 0) {

fmtstr = "yyyy/MM/dd";

} **else** {

**throw** **new** RuntimeException("Date type[" + datestr

+ "] not supported!");

}

SimpleDateFormat sdf = **new** SimpleDateFormat(fmtstr, Locale.***UK***);

**return** sdf.parse(datestr);

} **catch** (Exception e) {

**return** **null**;

}

}

}

### **四、测试类**

**public** **class** SpringTest {

**public** **static** **void** main(String[] args) {

Object bean = SpringAnnotation.*getBean*("annotation.Student");

System.***out***.println(bean);

}

}

### **五、测试结果：**

Student [id=20160001, name=孔明, age=20, grade=1, major=土木工程, school=武汉大学]

## 所有示例：

