**Java中的注解是如何起作用的**

翻译 [Danishlyy1995](https://me.csdn.net/u012934325) 最后发布于2019-09-16 09:15:28 阅读数 1028 收藏

展开

在Java中注解是很重要的一个组成部分，它是从J2SE 5.0开始就存在的。我们在日常开发的应用中应该已经见过类似于@Override和@Deprecated注解。在这篇文章中，我讨论注解是什么，为什么他们会存在，他们如何起作用，如何自定义注解（有代码示例），注解的有效使用场景，最后会说注解和ADF。这将是一个很长的帖子，所以拿一些咖啡，准备潜入注解的世界。

**1. 注解是什么？**

用一个单词来解释注解的概念，那就是metadata。metadata是关于数据的数据。因此注解是代码的元数据。例如，看一下下面的代码片段：

@Override

public String toString() {

return "This is String Representation of current object.";

}

我重写了toString()方法，并且在代码上面使用了@Override注解。即使我不加@Override注解，代码也可以正常运行部存在任何问题。所以加该注解的优点和该注解代表的含义是什么？@Override注解告诉编译器该方法是一个被重写的方法（有关方法的metadata）。如果父类中不存在任何此类方法，则抛出编译器错误（方法不会覆盖其父类中的方法）。现在，如果我犯一个排版的错误，并且使用方法名字如 toStrring() {double r},并且没有使用@Override注解，我们代码可以成功变异和执行，但是它的输出与我的预期结果不是一致的。所以现在，我们理解注解是什么，但仍然，阅读正式定义是好的。

注解是一种特殊的Java构造，用于修饰类，方法，字段，参数，变量，构造函数或包。  
它是JSR-175选择提供元数据的工具。

**2. 为什么要引入注解？**

在注解之前（甚至之后），XML被广泛用于metadata，并且不知何故，一组特定的应用程序开发人员和架构师认为XML维护变得很麻烦。他们想要的东西可以与代码紧密结合，而不是XML，它与代码非常松散耦合（在某些情况下，几乎是分开的）。如果你谷歌XML与注解，你会发现很多有趣的辩论。一个有趣的观点是，引入了XML配置来将配置与代码分开。最后两个陈述可能会在你的脑海中产生一些怀疑，这两个是创建一个循环，但两者都有其优点和缺点。让我们试着用一个例子来理解。

假设您要设置一些应用程序范围的常量/参数。在这种情况下，XML将是更好的选择，因为这与任何特定的代码片段无关。如果要将某个方法公开为服务，则注解将是更好的选择，因为它需要与该方法紧密耦合，并且方法的开发人员必须意识到这一点。

另一个重要因素是注解定义了在代码中定义元数据的标准方法。在注解之前，人们还使用自己的方式来定义元数据。一些示例使用标记接口，注解，transient关键字等。每个开发人员都需要以自己的方式来决定元数据，但是注解是的东西标准化。

目前，大多数框架都使用XML和Annotations的组合来利用两者的积极方面。

**3. 注解如何起作用以及如何自定义注解**

在开始此解释之前，我建议您下载此示例代码[AnnotationsSample.zip](https://drive.google.com/file/d/0B1N2DVZFnNU0dVdFVjVFeTVtcXc/edit),并在您选择的任何IDE中保持开放，因为它将帮助您更好地理解以下解释。

编写注解非常简单。您可以将注解定义与接口定义进行比较。让我们看两个例子，一个是标准的@Override注解，一个是@Todo的自定义注解

@Target(ElementType.METHOD)

@Retention(RetentionPolicy.SOURCE)

public @interface Override {

}

@Override似乎有点可疑;它没有做任何事情 ，它只是检查是否在父类中定义了一个方法。  
好吧，不要惊讶;我不是在开玩笑。@Override注解的定义只包含那么多代码。  
这是要理解的最重要的部分，我正在重申：**注解只是metadata，不包含任何业务逻辑**。

艰难消化但真实。如果注释不包含逻辑，那么其他人必须做某事并且某人是此注解metadata的使用者。注解仅提供有关定义它的属性（类/方法/包/字段）的信息。使用者是一段代码，它读取此信息然后执行必要的逻辑。

当我们讨论像@Override这样的标准注解时，JVM就是消费者，它在字节码级别起作用。  
这是应用程序开发人员无法控制的东西，也不能用于自定义注解。所以我们需要为我们的注解写消费者。

让我们逐一理解用于编写注解的关键术语。在上面的示例中，您将看到注解用于注解。

当写自定义注解的时候J2SE 5.0在java.lang.annotation包中提供了四种注解可以被使用：

* @Documented:是否将注解放在Javadocs
* @Retention:当需要注解的时候
* @Target:注解作用的位置
* @Inherited: 子类是否获得注解

@Documented：一个简单的市场注解，告诉您是否在Java文档中添加注解。

@Retention ：定义注解应保留多长时间。

* RetentionPolicy.SOURCE: 在编译期间丢弃。编译完成后，这些注解没有任何意义，因此它们不会写入字节码。示例：@Override，@ SuppressWarnings
* RetentionPolicy.CLASS: 在类加载期间丢弃。在进行字节码级后处理时很有用。这是默认值。
* RetentionPolicy.RUNTIME: 不会丢弃。注解应该可以在运行时进行反射。这是我们通常用于自定义注解的内容。

@Target: 可以放置注解的位置。如果不指定，则可以将注解放在任何位置。以下是有效值。  
这里的一个重点是它只是包容性，这意味着如果你想要对7个属性进行注解并且只想要只排除一个属性，则需要在定义目标时包括所有7个。

* ElementType.TYPE (class, interface, enum)
* ElementType.FIELD (instance variable)
* ElementType.METHOD
* ElementType.PARAMETER
* ElementType.CONSTRUCTOR
* ElementType.LOCAL\_VARIABLE
* ElementType.ANNOTATION\_TYPE (on another annotation)
* ElementType.PACKAGE (remember package-info.java)

@Inherited: 控制注解是否应该影响子类。

现在，注解定义中的内容是什么？注解仅支持基本数据类型，字符串和枚举。注解的所有属性都定义为方法，也可以提供默认值。

@Target(ElementType.METHOD)

@Retention(RetentionPolicy.RUNTIME)

@interface

Todo {

public enum Priority {LOW, MEDIUM, HIGH}

public enum Status {STARTED, NOT\_STARTED}

String author() default "Yash";

Priority priority() default Priority.LOW;

Status status() default Status.NOT\_STARTED;

}

以下是如何使用上述注解的示例：

@Todo(priority = Todo.Priority.MEDIUM, author = "Yashwant", status = Todo.Status.STARTED)

public void incompleteMethod1() {

//Some business logic is written

//But it’s not complete yet

}

如果注释中只有一个属性，则应将其命名为value，并且在使用时可以在没有属性名称的情况下使用它。

@interface Author{

String value();

}

@Author("Yashwant")

public void someMethod() {

}

到现在为止还挺好。我们定义了自定义注解并将其应用于某些业务逻辑方法。现在，是时候写一个消费者了。为此，我们需要使用反射。如果您熟悉反射代码，您就知道反射提供了Class，Method和Field对象。所有这些都有一个getAnnotation()方法，它返回注解对象。我们需要将此对象转换为自定义注解（在使用instanceOf()检查之后），然后，我们可以调用自定义注解中定义的方法。  
让我们看一下使用上面注解的示例代码：

Class businessLogicClass = BusinessLogic.class;

for(Method method : businessLogicClass.getMethods()) {

Todo todoAnnotation = (Todo)method.getAnnotation(Todo.class);

if(todoAnnotation != null) {

System.out.println(" Method Name : " + method.getName());

System.out.println(" Author : " + todoAnnotation.author());

System.out.println(" Priority : " + todoAnnotation.priority());

System.out.println(" Status : " + todoAnnotation.status());

}

}

**用例注解**

注解非常强大，Spring和Hibernate等框架非常广泛地使用注解进行日志记录和验证。  
注解可以在使用标记接口的位置使用。标记接口适用于完整的类，但您可以定义可以在单个方法上使用的注解，例如，某个方法是否作为服务方法公开。

在servlet规范3.0中，引入了许多注解，尤其是与servlet安全性相关的注解。我们来看看几个：

* HandlesTypes - 用于声明传递给ServletContainerInitializer的应用程序类数组。
* HttpConstraint - 此注解表示应用于具有HTTP协议方法类型的所有请求的安全性约束，这些类型在ServletSecurity注解中未由相应的HttpMethodConstraint表示。
* HttpMethodConstraint - 特定的安全性约束可以应用于不同类型的请求，通过在ServletSecurity注解内使用此注解来区分HTTP协议方法类型。
* MultipartConfig - 此注解用于指示声明它的Servlet期望使用multipart / form-data MIME类型进行请求。
* ServletSecurity - 在Servlet实现类上声明此注解，以强制对HTTP协议请求进行安全性约束。
* WebFilter - 用于声明Servlet过滤器的注解。
* WebInitParam - 用于在WebFilter或WebServlet注解内的Servlet或Filter上声明初始化参数的注解。
* WebListener - 用于在给定Web应用程序上下文中为各种类型的事件声明侦听器的注解。
* WebServlet- 此注解用于声明Servlet的配置。