

个人资料



AlbertFly



访问： 81239次

积分： 2681

等级：

排名： 第10011名

原创： 148篇 转载： 285篇

译文： 0篇 评论： 5条

文章搜索

文章分类

- mysql (32)
- MQ中间件 (12)
- 多线程 (5)
- 并发 (4)
- spring (33)
- web server (42)
- html (1)
- js (17)
- jsp (19)
- css (4)
- redis&缓存 (28)
- 架构 (6)
- j2se (55)
- 工具使用 (21)
- zookeeper (2)
- jvm (18)
- 大数据 (2)
- tomcat (21)
- hibernate (8)
- 分布式 (4)
- 通信&netty (31)
- linux (21)
- 数据机构 算法 (0)
- 性能优化 (23)
- 设计模式原则 (8)
- maven (6)
- go (1)
- nosql (2)
- nginx (10)
- 读书 (6)
- 生活 (1)
- mybatis (2)
- shiro (8)
- 工作感悟 (2)
- shell (11)
- rpc (6)

文章存档

- 2016年10月 (6)
- 2016年09月 (52)
- 2016年08月 (69)
- 2016年07月 (45)
- 2016年06月 (76)

展开

阅读排行

- 使用Spring Data Redis# (2245)
- 对于组合模式的理解 (2187)
- Java 应用一般架构 (2083)
- mysql timeStamp默认值 (1931)
- DUBBO配置规则详解 (1914)
- MAT使用的几张图例技巧 (1698)
- SpringMVC基于session (1671)
- java8之Lambda表达式 4 (1643)
- mysql5.7.10启动修改初 (1630)
- Kafka 设计与原理 详解 (1209)

【1024程序员节】我们的世界不只0和1 【观点】有了深度学习，你还学传统机器学习算法么？ 【知识库】深度学习知识图谱上线啦

## Java注解处理器使用详解

2016-09-01 19:16 110人阅读 评论(0) 收藏 举报

分类： j2se (54)

目录(?)

在这篇文章中，我将阐述怎样写一个注解处理器(Annotation Processor)。在这篇教程中，首先，我将向您解释什么是注解器，你可以利用这个强大的工具做什么以及不能做什么；然后，我将一步一步实现一个简单的注解器。

### 一些基本概念

在开始之前，我们首先申明一个非常重要的问题：我们并不讨论那些在运行时（Runtime）通过反射机制运行处理的注解，而是讨论在编译时（Compile time）处理的注解。

注解处理器是一个在javac中的，用来编译时扫描和处理的注解的工具。你可以为特定的注解，注册你自己的注解处理器。到这里，我假设你已经知道什么是注解，并且知道怎么申明的一个注解类型。如果你不熟悉注解，你可以在这[官方文档](#)中得到更多信息。注解处理器在Java 5开始就有了，但是从Java 6（2006年12月发布）开始才有可用的API。过了一段时间，Java世界才意识到注解处理器的强大作用，所以它到最近几年才流行起来。

一个注解的注解处理器，以Java代码（或者编译过的字节码）作为输入，生成文件（通常是 .java 文件）作为输出。这具体的含义什么呢？你可以生成Java代码！这些生成的Java代码是在生成的java文件中，所以你不能修改已经存在的Java类，例如向已有的类中添加方法。这些生成的Java文件，会同其他普通的手动编写的Java源代码一样被javac编译。

### 虚处理器 AbstractProcessor

我们首先看一下处理器的API。每一个处理器都是继承于 AbstractProcessor ，如下所示：

```
package com.example;

public class MyProcessor extends AbstractProcessor {

    @Override
    public synchronized void init(ProcessingEnvironment env){ }

    @Override
    public boolean process(Set<? extends TypeElement> annoations, RoundEnvironment e
nv) { }

    @Override
    public Set<String> getSupportedAnnotationTypes() { }

    @Override
    public SourceVersion getSupportedSourceVersion() { }

}
```

init(ProcessingEnvironment env)：每一个注解处理器类都必须有一个空的构造函数。然而，这里有一个特殊的 init() 方法，它会被注解处理工具调用，并输入 ProcessingEnviroment 参数。 ProcessingEnviroment 提供很多有用的工具类 Elements ， Types 和 Filer 。后面我们将看到详细的内容。

process(Set<? extends TypeElement> annotations, RoundEnvironment env)：这相当于每个处理器的主函数 main() 。你在这里写你的扫描、评估和处理注解的代码，以及生成Java文件。输入参数 RoundEnviroment ，可以让你查询出包含特定注解的被注解元素。后面我们将看到详细的内容。

getSupportedAnnotationTypes()：这里你必须指定，这个注解处理器是注册给哪个注解的。注意，它的返回值是一个字符串的集合，包含本处理器想要处理的注解类型的合法全称。换句话说，你在这里定义你的注解处理器注册到哪些注解上。

getSupportedSourceVersion()：用来指定你使用的Java版本。通常这里返回 SourceVersion.latestSupported() 。然而，如果你有足够的理由只支持Java 6的话，你也可以返回 SourceVersion.RELEASE\_6 。我推荐你使用前者。

在Java 7中，你也可以使用注解来代替 getSupportedAnnotationTypes() 和 getSupportedSourceVersion() ，像这样：

```
@SupportedSourceVersion(SourceVersion.latestSupported())
@SupportedAnnotationTypes({
    // 合法注解全名的集合
})
public class MyProcessor extends AbstractProcessor {

    @Override
    public synchronized void init(ProcessingEnvironment env){ }

    @Override
    public boolean process(Set<? extends TypeElement> annoations, RoundEnvironment e
nv) { }

}
```

因为兼容的原因，特别是针对Android平台，我建议从重载 getSupportedAnnotationTypes() 和 getSupportedSourceVersion() 方法代替 @SupportedAnnotationTypes 和 @SupportedSourceVersion 。

评论排行

- Tomcat 通过listener 启动 (2)
- 23种设计模式全解析-- 设 (2)
- SpringMVC在Controller (1)
- 解析 filter+注解+Handler (0)
- RabbitMQ Spring AMQP (0)
- redis多个线程操作单个k (0)
- 构建高并发高可用的电商 (0)
- 用Redis存储Tomcat集群 (0)
- jsp页面中的\$(param.xxx (0)
- SpringMVC - controller中 (0)

推荐文章

- \* 2016 年最受欢迎的编程语言是什么？
- \* Chromium扩展（Extension）的页面（Page）加载过程分析
- \* Android Studio 2.2 来啦
- \* 手把手教你做音乐播放器（二）技术原理与框架设计
- \* JVM 性能调优实战之：使用阿里开源工具 TProfiler 在海量业务代码中精确定位性能代码

最新评论

- SpringMVC在Controller层中注了mryangi2: 我也好奇它是怎么实现的
- 23种设计模式全解析-- 设计模式: AlbertFly: @qq\_34478076:道友好眼光 这个文章我自己置顶没事看看 老容易忘
- 23种设计模式全解析-- 设计模式: qq\_34478076: 那么吊的博文，我来做第一个评论的
- Tomcat 通过listener 启动netty 那 AlbertFly: @me\_is\_vector:你试试就知道了 我觉着tomcat关了 按照这种方法就应该停止服务了，你...
- Tomcat 通过listener 启动netty 那 me\_is\_vector: 如果关闭了tomcat，netty服务会关闭吗？

接下来的你必须知道的事情是，注解处理器是运行它自己的虚拟机JVM中。是的，你没有看错，javac启动一个完整Java虚拟机来运行注解处理器。这对你意味着什么？你可以使用任何你在其他java应用中使用的的东西。使用guava。如果你愿意，你可以使用依赖注入工具，例如dagger或者其他你想要的类库。但是不要忘记，即使是一个很小的处理，你也要像其他Java应用一样，注意**算法**效率，以及**设计模式**。

### 注册你的处理器

你可能会问，我怎样处理器 MyProcessor 到javac中。你必须提供一个 .jar 文件。就像其他jar文件一样，你打包你的注解处理器到此文件中。并且，在你的jar中，你需要打包一个特定的文件

javax.annotation.processing.Processor 到 META-INF/services 路径下。所以，你的jar文件看起来就像下面这样：



打包进MyProcessor.jar中的 javax.annotation.processing.Processor 的内容是，注解处理器的合法的全名列表，每一个元素换行分割：

```
com.example.MyProcessor
com.foo.OtherProcessor
net.blabla.SpecialProcessor
```

把 MyProcessor.jar 放到你的builpath中，javac会自动检查和读取 javax.annotation.processing.Processor 中的内容，并且注册 MyProcessor 作为注解处理器。

### 例子：工厂模式

是时候来说一个实际的例子了。我们将使用maven工具来作为我们的编译系统和依赖管理工具。如果你不熟悉maven，不用担心，因为maven不是必须的。本例子的完成代码在Github上。

开始之前，我必须说，要为此教程找到一个需要用注解处理器解决的简单问题，实在并不容易。这里我们将实现一个非常简单的工厂模式（不是抽象工厂模式）。这将对注解处理器的API做一个非常简明的介绍。所以，这个问题的程序并不是那么有用，也不是一个真实世界的例子。所以在此申明，你将学习关于注解处理过程的相关内容，而不是设计模式。

我们将要解决的问题是：我们将实现一个披萨店，这个披萨店给消费者提供两种披萨（“Margherita”和“Calzone”）以及提拉米苏甜点(Tiramisu)。

看一下如下的代码，不需要做任何更多的解释：

```
public interface Meal {
    public float getPrice();
}

public class MargheritaPizza implements Meal {

    @Override public float getPrice() {
        return 6.0f;
    }
}

public class CalzonePizza implements Meal {

    @Override public float getPrice() {
        return 8.5f;
    }
}

public class Tiramisu implements Meal {

    @Override public float getPrice() {
        return 4.5f;
    }
}
```

为了在我们的披萨店 PizzasStore 下订单，消费者需要输入餐(Meal)的名字。

```
public class PizzaStore {

    public Meal order(String mealName) {

        if (mealName == null) {
            throw new IllegalArgumentException("Name of the meal is null!");
        }

        if ("Margherita".equals(mealName)) {
            return new MargheritaPizza();
        }

        if ("Calzone".equals(mealName)) {
            return new CalzonePizza();
        }

        if ("Tiramisu".equals(mealName)) {
            return new Tiramisu();
        }

        throw new IllegalArgumentException("Unknown meal '" + mealName + "'");
    }

    public static void main(String[] args) throws IOException {
        PizzaStore pizzaStore = new PizzaStore();
        Meal meal = pizzaStore.order(readConsole());
        System.out.println("Bill: $" + meal.getPrice());
    }
}
```

正如你所见，在 order() 方法中，我们有很多的 if 语句，并且如果我们每添加一种新的披萨，我们都要添加一条新的 if 语句。但是等一下，使用注解处理和工厂模式，我们可以让注解处理器来帮我们自动生成这些 if 语句。如此以来，我们期望的是如下的代码：

```
public class PizzaStore {

    private MealFactory factory = new MealFactory();

    public Meal order(String mealName) {
        return factory.create(mealName);
    }

    public static void main(String[] args) throws IOException {
        PizzaStore pizzaStore = new PizzaStore();
        Meal meal = pizzaStore.order(readConsole());
        System.out.println("Bill: $" + meal.getPrice());
    }
}
```

MealFactory 应该是如下的样子：

```
public class MealFactory {

    public Meal create(String id) {
        if (id == null) {
            throw new IllegalArgumentException("id is null!");
        }
        if ("Calzone".equals(id)) {
            return new CalzonePizza();
        }

        if ("Tiramisu".equals(id)) {
            return new Tiramisu();
        }

        if ("Margherita".equals(id)) {
            return new MargheritaPizza();
        }

        throw new IllegalArgumentException("Unknown id = " + id);
    }
}
```

### @Factory 注解

你能猜到么：我们想用注解处理器自动生成 MealFactory 。更一般的说，我们将想要提供一个注解和一个处理器来生成工厂类。

我们先来看一下 @Factory 注解：

```
@Target(ElementType.TYPE) @Retention(RetentionPolicy.CLASS)
public @interface Factory {

    /**
     * 工厂的名字
     */
    Class type();

    /**
     * 用来表示生成哪个对象的唯一id
     */
    String id();
}
```

想法是这样的：我们将使用同样的 type() 注解那些属于同一个工厂的类，并且用注解的 id() 做一个映射，例如从 "Calzone" 映射到 "ClzonePizza" 类。我们应用 @Factory 注解到我们的类中，如下：

```
@Factory(
    id = "Margherita",
    type = Meal.class
)
public class MargheritaPizza implements Meal {

    @Override public float getPrice() {
        return 6f;
    }
}
```

```
@Factory(
    id = "Calzone",
    type = Meal.class
)
public class CalzonePizza implements Meal {

    @Override public float getPrice() {
        return 8.5f;
    }
}
```

```
@Factory(
    id = "Tiramisu",
    type = Meal.class
)
public class Tiramisu implements Meal {

    @Override public float getPrice() {
        return 4.5f;
    }
}
```

你可能会问你自己，我们是否可以只把 @Factory 注解应用到我们的 Meal 接口上？答案是，注解是不能继承的。一个类 class X 被注解，并不意味着它的子类 class Y extends X 会自动被注解。在我们开始写处理器的代码之前，我们先规定如下一些规则：



1. 只有类可以被 `@Factory` 注解，因为接口或者抽象类并不能用 `new` 操作实例化；
2. 被 `@Factory` 注解的类，必须至少提供一个公开的默认构造器（即没有参数的构造函数）。否则我们没法实例化一个对象。
3. 被 `@Factory` 注解的类必须直接或者间接的继承于 `type()` 指定的类型；
4. 具有相同的 `type` 的注解类，将被聚合在一起生成一个工厂类。这个生成的类使用Factory后缀，例如 `type = Meal.class`，将生成 `MealFactory` 工厂类；
5. `id` 只能是String类型，并且在同一个 `type` 组中必须唯一。

## 处理器

我将通过添加代码和一段解释的方法，一步一步的引导你来构建我们的处理器。省略号( ... )表示省略那些已经讨论过的或者将在后面的步骤中讨论的代码，目的是为了我们的代码有更好的可读性。正如我们前面说的，我们完整的代码可以在[Github](#)上找到。好了，让我们来看一下我们的处理器类 `FactoryProcessor` 的骨架：

```
@AutoService(Processor.class)
public class FactoryProcessor extends AbstractProcessor {

    private Types typeUtils;
    private Elements elementUtils;
    private Filer filer;
    private Messenger messenger;
    private Map<String, FactoryGroupedClasses> factoryClasses = new LinkedHashMap<String, FactoryGroupedClasses>();

    @Override
    public synchronized void init(ProcessingEnvironment processingEnv) {
        super.init(processingEnv);
        typeUtils = processingEnv.getTypeUtils();
        elementUtils = processingEnv.getElementUtils();
        filer = processingEnv.getFiler();
        messenger = processingEnv.getMessenger();
    }

    @Override
    public Set<String> getSupportedAnnotationTypes() {
        Set<String> annotataions = new LinkedHashSet<String>();
        annotataions.add(Factory.class.getCanonicalName());
        return annotataions;
    }

    @Override
    public SourceVersion getSupportedSourceVersion() {
        return SourceVersion.latestSupported();
    }

    @Override
    public boolean process(Set<? extends TypeElement> annotations, RoundEnvironment roundEnv) {
        ...
    }
}
```

你看到在代码的第一行是 `@AutoService(Processor.class)`，这是什么？这是一个其他注解处理器中引入的注解。`AutoService` 注解处理器是Google开发的，用来生成 `META-INF/services/javax.annotation.processing.Processor` 文件的。是的，你没有看错，我们可以在注解处理器中使用注解。非常方便，难道不是么？在 `getSupportedAnnotationTypes()` 中，我们指定本处理器将处理 `@Factory` 注解。

## Elements和TypeMirrors

在 `init()` 中我们获得如下引用：

- Elements：一个用来处理 `Element` 的工具类（后面将做详细说明）；
- Types：一个用来处理 `TypeMirror` 的工具类（后面将做详细说明）；
- Filer：正如这个名字所示，使用Filer你可以创建文件。

在注解处理过程中，我们扫描所有的Java源文件。源代码的每一个部分都是一个特定类型的 `Element`。换句话说：`Element` 代表程序的元素，例如包、类或者方法。每个 `Element` 代表一个静态的、语言级别的构件。在下面的例子中，我们通过注释来说明这个：

```
package com.example;    // PackageElement

public class Foo {        // TypeElement

    private int a;         // VariableElement
    private Foo other;     // VariableElement

    public Foo () {}       // ExecuteableElement

    public void setA (     // ExecuteableElement
        int newA         // TypeElement
    ) {}
}
```

你必须换个角度来看源代码，它只是结构化的文本，他不是可运行的。你可以想象它就像你将来去解析的XML文件一样（或者是编译器中抽象的语法树）。就像XML解释器一样，有一些类似DOM的元素。你可以从一个元素导航到它的父或者子元素上。

举例来说，假如你有一个代表 `public class Foo` 类的 `TypeElement` 元素，你可以遍历它的孩子，如下：

```
TypeElement fooClass = ... ;
for (Element e : fooClass.getEnclosedElements()){ // iterate over children
    Element parent = e.getEnclosingElement(); // parent == fooClass
}
```

正如你所见，Element代表的是源代码。 `TypeElement` 代表的是源代码中的类型元素，例如类。然而， `TypeElement` 并不包含类本身的信息。你可以从 `TypeElement` 中获取类的名字，但是你获取不到类的信息，例如它的父类。这种信息需要通过 `TypeMirror` 获取。你可以通过调用 `elements.asType()` 获取元素的 `TypeMirror` 。

### 搜索@Factory注解

我们来一步一步实现 `process()` 方法。首先，我们从搜索被注解了 `@Factory` 的类开始：

```
@AutoService(Processor.class)
public class FactoryProcessor extends AbstractProcessor {

    private Types typeUtils;
    private Elements elementUtils;
    private Filer filer;
    private Messenger messenger;
    private Map<String, FactoryGroupedClasses> factoryClasses = new LinkedHashMap<String, FactoryGroupedClasses>();
    ...

    @Override
    public boolean process(Set<? extends TypeElement> annotations, RoundEnvironment roundEnv) {

        // 遍历所有被注解了@Factory的元素
        for (Element annotatedElement : roundEnv.getElementsAnnotatedWith(Factory.class)) {

            ...

        }
    }
    ...
}
```

这里并没有什么高深的技术。 `roundEnv.getElementsAnnotatedWith(Factory.class)` 返回所有被注解了 `@Factory` 的元素的列表。你可能已经注意到，我们并没有说“所有被注解了 `@Factory` 的类的列表”，因为它真的是返回 `Element` 的列表。请记住：`Element` 可以是类、方法、变量等。所以，接下来，我们必须检查这些 `Element`是否是一个类：

```
@Override
public boolean process(Set<? extends TypeElement> annotations, RoundEnvironment roundEnv) {

    for (Element annotatedElement : roundEnv.getElementsAnnotatedWith(Factory.class)) {

        // 检查被注解为@Factory的元素是否是一个类
        if (annotatedElement.getKind() != ElementKind.CLASS) {

            ...

        }
    }
    ...
}
```

为什么要这么做？我们要确保只有class元素被我们的处理器处理。前面我们已经学习到类是用 `TypeElement` 表示。我们为什么不这样判断呢 `if (!(annotatedElement instanceof TypeElement))`？这是错误的，因为接口（`interface`）类型也是 `TypeElement`。所以在注解处理器中，我们要避免使用 `instanceof`，而是配合 `TypeMirror` 使用 `EmentKind` 或者 `TypeKind`。

### 错误处理

在 `init()` 中，我们也获得了一个 `Messenger` 对象的引用。 `Messenger` 提供给注解处理器一个报告错误、警告以及提示信息的途径。它不是注解处理器开发者的日志工具，而是用来写一些信息给使用此注解器的第三方开发者的。在[官方文档](#)中描述了消息的不同级别。非常重要的是 `Kind.ERROR`，因为这种类型的信息用来表示我们的注解处理器处理失败了。很有可能是第三方开发者错误的使用了 `@Factory` 注解（例如，给接口使用了 `@Factory` 注解）。这个概念和传统的Java应用有点不一样，在传统Java应用中我们可能就抛出一个异常 `Exception`。如果你在 `process()` 中抛出一个异常，那么运行注解处理器的JVM将会崩溃（就像其他Java应用一样），使用我们注解处理器 `FactoryProcessor` 第三方开发者将会从 `javac` 中得到非常难懂的出错信息，因为它包含 `FactoryProcessor` 的堆栈跟踪（`Stacktace`）信息。因此，注解处理器就有一个 `Messenger` 类，它能够打印非常优美的错误信息。除此之外，你还可以连接到出错的元素。在像IntelliJ这种现代的IDE（集成开发环境）中，第三方开发者可以直接点击错误信息，IDE将会直接跳转到第三方开发者项目的出错的源文件的相应的行。

我们重新回到 `process()` 方法的实现。如果遇到一个非类类型被注解 `@Factory`，我们发出一个出错信息：

```
public boolean process(Set<? extends TypeElement> annotations, RoundEnvironment roundEnv) {

    for (Element annotatedElement : roundEnv.getElementsAnnotatedWith(Factory.class)) {

        // 检查被注解为@Factory的元素是否是一个类
        if (annotatedElement.getKind() != ElementKind.CLASS) {
            error(annotatedElement, "Only classes can be annotated with @%s",
                Factory.class.getSimpleName());
            return true; // 退出处理
        }
        ...
    }

    private void error(Element e, String msg, Object... args) {
        messenger.printMessage(
            Diagnostic.Kind.ERROR,
            String.format(msg, args),
            e);
    }

}
```

让Messenger显示相关出错信息，更重要的是注解处理器程序必须完成运行而不崩溃，这就是为什么在调用 `error()` 后直接 `return` 。如果我们不直接返回，`process()` 将继续运行，因为 `messenger.printMessage(Diagnostic.Kind.ERROR)` 不会停止此进程。因此，如果我们在打印错误信息以后不返回的话，我们很可能就会运行到一个`NullPointerException`等。就像我们前面说的，如果我们继续运行 `process()` ，问题是如果在 `process()` 中抛出一个未处理的异常，javac将会打印出内部的`NullPointerException`，而不是 `Messenger` 中的错误信息。

## 数据模型

在继续检查被注解@Fractory的类是否满足我们上面说的5条规则之前，我们将介绍一个让我们更方便继续处理的**数据结构**。有时候，一个问题或者解释器看起来如此简单，以至于**程序员**倾向于用一个面向过程方式来写整个处理器。但是你知道吗？一个注解处理器任然是一个Java程序，所以我们需要使用面向对象编程、接口、设计模式，以及任何你将在其他普通Java程序中使用的技巧。

我们的 `FactoryProcessor` 非常简单，但是我们仍然想要把一些信息存为对象。在 `FactoryAnnotatedClass` 中，我们保存被注解类的数据，比如合法的类的名字，以及@Factory注解本身的一些信息。所以，我们保存 `TypeElement` 和处理过的@Factory注解：

```
public class FactoryAnnotatedClass {

    private TypeElement annotatedClassElement;
    private String qualifiedSuperClassName;
    private String simpleTypeName;
    private String id;

    public FactoryAnnotatedClass(TypeElement classElement) throws IllegalArgumentException {
        this.annotatedClassElement = classElement;
        Factory annotation = classElement.getAnnotation(Factory.class);
        id = annotation.id();

        if (StringUtils.isEmpty(id)) {
            throw new IllegalArgumentException(
                String.format("id() in @%s for class %s is null or empty! that's not allowed",
                    Factory.class.getSimpleName(), classElement.getQualifiedName().toString()));
        }

        // Get the full QualifiedTypeName
        try {
            Class<?> clazz = annotation.type();
            qualifiedSuperClassName = clazz.getCanonicalName();
            simpleTypeName = clazz.getSimpleName();
        } catch (MirroredTypeException mte) {
            DeclaredType classTypeMirror = (DeclaredType) mte.getTypeMirror();
            TypeElement classTypeElement = (TypeElement) classTypeMirror.asElement();
            qualifiedSuperClassName = classTypeElement.getQualifiedName().toString();
            simpleTypeName = classTypeElement.getSimpleName().toString();
        }

        /**
         * 获取在{@link Factory#id()}中指定的id
         * return the id
         */
        public String getId() {
            return id;
        }

        /**
         * 获取在{@link Factory#type()}指定的类型合法全名
         *
         * @return qualified name
         */
        public String getQualifiedFactoryGroupName() {
            return qualifiedSuperClassName;
        }

        /**
         * 获取在{@link Factory#type()}{@link Factory#type()}指定的类型的简单名字
         *
         * @return qualified name
         */
        public String getSimpleFactoryGroupName() {
            return simpleTypeName;
        }

        /**
         * 获取被@Factory注解的原始元素
         */
        public TypeElement getTypeElement() {
            return annotatedClassElement;
        }
    }
}
```

代码很多，但是最重要的部分是在构造函数中。其中你能找到如下的代码：

```
Factory annotation = classElement.getAnnotation(Factory.class);
id = annotation.id(); // Read the id value (like "Calzone" or "Tiramisu")

if (StringUtils.isEmpty(id)) {
    throw new IllegalArgumentException(
        String.format("id() in @%s for class %s is null or empty! that's not allowed",
            Factory.class.getSimpleName(), classElement.getQualifiedName().toString()));
}
```

这里我们获取@Factory注解，并且检查id是否为空？如果为空，我们将抛出`IllegalArgumentException`异常。你可能感到疑惑的是，前面我们说了不要抛出异常，而是使用 `Messenger` 。这里仍然不矛盾。我们抛出内部的异常，你在将在后面看到会在 `process()` 中捕获这个异常。我这样做出于一下两个原因：



1. 我想示意我们应该像普通的Java程序一样编码。抛出和捕获异常是非常好的Java编程实践；
2. 如果我们想要在 `FactoryAnnotatedClass` 中打印信息，我需要也传入 `Messenger` 对象，并且我们在错误处理一节中已经提到，为了打印 `Messenger` 信息，我们必须成功停止处理器运行。如果我们使用 `Messenger` 打印了错误信息，我们怎样告知 `process()` 出现了错误呢？最容易，并且我认为最直观的方式就是抛出一个异常，然后让 `process()` 捕获之。

接下来，我们将获取 `@Fractory` 注解中的 `type` 成员。我们比较关心的是合法的全名：

```
try {
    Class<?> clazz = annotation.type();
    qualifiedGroupClassName = clazz.getCanonicalName();
    simpleFactoryGroupName = clazz.getSimpleName();
} catch (MirroredTypeException mte) {
    DeclaredType classTypeMirror = (DeclaredType) mte.getTypeMirror();
    TypeElement classTypeElement = (TypeElement) classTypeMirror.asElement();
    qualifiedGroupClassName = classTypeElement.getQualifiedName().toString();
    simpleFactoryGroupName = classTypeElement.getSimpleName().toString();
}
```

这里有一点小麻烦，因为这里的类型是一个 `java.lang.Class` 。这意味着，他是一个真正的Class对象。因为注解处理是在编译Java源代码之前。我们需要考虑如下两种情况：

1. 这个类已经被编译：这种情况是：如果第三方 `.jar` 包含已编译的被`@Factory`注解 `.class` 文件。在这种情况下，我们可以想 `try` 中那块代码中所示直接获取 `Class` 。
2. 这个还没有被编译：这种情况是我们尝试编译被`@Fractory`注解的源代码。这种情况下，直接获取Class会抛出 `MirroredTypeException` 异常。幸运的是，`MirroredTypeException`包含一个 `TypeMirror` ，它表示我们未编译类。因为我们已经知道它必定是一个类类型（我们已经在前面检查过），我们可以直接强制转换为 `DeclaredType` ，然后读取 `TypeElement` 来获取合法的名字。

好了，我们现在还需要一个数据结构 `FactoryGroupedClasses` ，它将简单的组合所有的 `FactoryAnnotatedClasses` 到一起。

```
public class FactoryGroupedClasses {

    private String qualifiedClassName;

    private Map<String, FactoryAnnotatedClass> itemsMap =
        new LinkedHashMap<String, FactoryAnnotatedClass>();

    public FactoryGroupedClasses(String qualifiedClassName) {
        this.qualifiedClassName = qualifiedClassName;
    }

    public void add(FactoryAnnotatedClass toInsert) throws IdAlreadyUsedException {

        FactoryAnnotatedClass existing = itemsMap.get(toInsert.getId());
        if (existing != null) {
            throw new IdAlreadyUsedException(existing);
        }

        itemsMap.put(toInsert.getId(), toInsert);
    }

    public void generateCode(Elements elementUtils, Filer filer) throws IOException {
        ...
    }
}
```

正如你所见，这是一个基本的 `Map<String, FactoryAnnotatedClass>` ，这个映射表用来映射`@Factory.id()`到 `FactoryAnnotatedClass`。我们选择 `Map` 这个数据类型，是因为我们要确保每个id是唯一的，我们可以很容易通过map查找实现。 `generateCode()` 方法将被用来生成工厂类代码（将在后面讨论）。

### 匹配标准

我们继续实现 `process()` 方法。接下来我们想要检查被注解的类必须有只要一个公开的构造函数，不是抽象类，继承于特定的类型，以及是一个公开类：

```
public class FactoryProcessor extends AbstractProcessor {

    @Override
    public boolean process(Set<? extends TypeElement> annotations, RoundEnvironment roundEnv) {

        for (Element annotatedElement :
            roundEnv.getElementsAnnotatedWith(Factory.class)) {

            ...

            // 因为我们已经知道它是ElementKind.CLASS类型，所以可以直接强制转换
            TypeElement typeElement = (TypeElement) annotatedElement;

            try {
                FactoryAnnotatedClass annotatedClass =
                    new FactoryAnnotatedClass(typeElement); // throws IllegalArgumentException
            } catch (IllegalArgumentException e) {
                return true; // 已经打印了错误信息，退出处理过程
            }
            // @Factory.id()为空
            error(typeElement, e.getMessage());
            return true;
        }

        ...
    }

    private boolean isValidClass(FactoryAnnotatedClass item) {

        // 转换为TypeElement，含有更多特定的方法
    }
}
```

```
Java注解处理器使用详解 - Flyer的后花园 - 博客频道 - CSDN.NET

TypeElement classElement = item.getTypeElement();

if (!classElement.getModifiers().contains(Modifier.PUBLIC)) {
    error(classElement, "The class %s is not public.",
        classElement.getQualifiedName().toString());
    return false;
}

// 检查是否是一个抽象类
if (classElement.getModifiers().contains(Modifier.ABSTRACT)) {
    error(classElement, "The class %s is abstract. You can't annotate abstract cla
sses with @%",
        classElement.getQualifiedName().toString(), Factory.class.getSimpleName
());
    return false;
}

// 检查继承关系：必须是@Factory.type()指定的类型子类
TypeElement superClassElement =
    elementUtils.getTypeElement(item.getQualifiedFactoryGroupName());
if (superClassElement.getKind() == ElementKind.INTERFACE) {
    // 检查接口是否实现了
    if(!classElement.get
Interfaces().contains(superClassElement.asType())) {
        error(classElement, "The class %s annotated with @%s must implement the inte
rface %s",
            classElement.getQualifiedName().toString(), Factory.class.getSimpleName
(),
            item.getQualifiedFactoryGroupName());
        return false;
    }
} else {
    // 检查子类
    TypeElement currentClass = classElement;
    while (true) {
        TypeMirror superClassType = currentClass.getSuperclass();

        if (superClassType.getKind() == TypeKind.NONE) {
            // 到达了基本类型(java.lang.Object), 所以退出
            error(classElement, "The class %s annotated with @%s must inherit from
%s",
                classElement.getQualifiedName().toString(), Factory.class.getSimpleNam
e(),
                item.getQualifiedFactoryGroupName());
            return false;
        }

        if (superClassType.toString().equals(item.getQualifiedFactoryGroupName())) {
            // 找到了要求的父类
            break;
        }

        // 在继承树上继续向上搜寻
        currentClass = (TypeElement) typeUtils.asElement(superClassType);
    }
}

// 检查是否提供了默认公开构造函数
for (Element enclosed : classElement.getEnclosedElements()) {
    if (enclosed.getKind() == ElementKind.CONSTRUCTOR) {
        ExecutableElement constructorElement = (ExecutableElement) enclosed;
        if (constructorElement.getParameters().size() == 0 && constructorElement.get
Modifiers()
            .contains(Modifier.PUBLIC)) {
            // 找到了默认构造函数
            return true;
        }
    }
}

// 没有找到默认构造函数
error(classElement, "The class %s must provide an public empty default construct
or",
    classElement.getQualifiedName().toString());
return false;
}
}
```

我们这里添加了 isValidClass() 方法，来检查是否我们所有的规则都被满足了：

必须是公开类： classElement.getModifiers().contains(Modifier.PUBLIC)

必须是非抽象类： classElement.getModifiers().contains(Modifier.ABSTRACT)

必须是 @Factoy.type() 指定的类型的子类或者接口的实现：首先我们使用 elementUtils.getTypeElement(item.getQualifiedFactoryGroupName()) 创建一个传入的 Class ( @Factoy.type() )的元素。是的，你可以仅仅通过已知的合法类名来直接创建 TypeElement （使用TypeMirror）。接下来我们检查它是一个接口还是一个类： superClassElement.getKind() == ElementKind.INTERFACE 。所以我们这里有两种情况：如果是接口，就判断 classElement.getInterfaces().contains(superClassElement.asType()) ；如果是类，我们就必须使用 currentClass.getSuperclass() 扫描继承层级。注意，整个检查也可以使用 typeUtils.isSubtype() 来实现。

类必须有一个公开的默认构造函数：我们遍历所有的闭元素 classElement.getEnclosedElements() ，然后检查 ElementKind.CONSTRUCTOR 、 Modifier.PUBLIC 以及 constructorElement.getParameters().size() == 0 。

如果所有这些条件都满足， isValidClass() 返回 true ，否则就打印错误信息，并且返回 false 。

### 组合被注解的类

一旦我们检查 isValidClass() 成功，我们将添加 FactoryAnnotatedClass 到对应的 FactoryGroupedClasses 中，如下：



```
public class FactoryProcessor extends AbstractProcessor {

    private Map<String, FactoryGroupedClasses> factoryClasses =
        new LinkedHashMap<String, FactoryGroupedClasses>();

    @Override
    public boolean process(Set<? extends TypeElement> annotations, RoundEnvironment roundEnv) {
        ...
        try {
            FactoryAnnotatedClass annotatedClass =
                new FactoryAnnotatedClass(typeElement); // throws IllegalArgumentException on

            if (!isValidClass(annotatedClass)) {
                return true; // 错误信息被打印，退出处理流程
            }

            // 所有检查都没有问题，所以可以添加了
            FactoryGroupedClasses factoryClass =
                factoryClasses.get(annotatedClass.getQualifiedFactoryGroupName());
            if (factoryClass == null) {
                String qualifiedGroupName = annotatedClass.getQualifiedFactoryGroupName();
                factoryClass = new FactoryGroupedClasses(qualifiedGroupName);
                factoryClasses.put(qualifiedGroupName, factoryClass);
            }

            // 如果和其他的@Factory标注的类的id相同冲突，
            // 抛出IdAlreadyUsedException异常
            factoryClass.add(annotatedClass);
        } catch (IllegalArgumentException e) {
            // @Factory.id()为空 --> 打印错误信息
            error(typeElement, e.getMessage());
            return true;
        } catch (IdAlreadyUsedException e) {
            FactoryAnnotatedClass existing = e.getExisting();
            // 已经存在
            error(annotatedElement,
                "Conflict: The class %s is annotated with @%s with id ='%s' but %s already uses the same id",
                typeElement.getQualifiedName().toString(), Factory.class.getSimpleName(),
                existing.getTypeElement().getQualifiedName().toString());
            return true;
        }
    }
    ...
}
```

代码生成

我们已经手机了所有的被 @Factory 注解的类保存到为 FactoryAnnotatedClass ，并且组合到了 FactoryGroupedClasses 。现在我们将为每个工厂生成Java文件了：

```
@Override
public boolean process(Set<? extends TypeElement> annotations, RoundEnvironment roundEnv) {
    ...
    try {
        for (FactoryGroupedClasses factoryClass : factoryClasses.values()) {
            factoryClass.generateCode(elementUtils, filer);
        }
    } catch (IOException e) {
        error(null, e.getMessage());
    }

    return true;
}
```

写Java文件，和写其他普通文件没有什么两样。使用 Filer 提供的 Writer 对象，我们可以连接字符串来写我们生成的Java代码。幸运的是，Square公司（因为提供了许多非常优秀的开源项目二非常有名）给我们提供了 JavaWriter ，这是一个高级的生成Java代码的库：

```
public class FactoryGroupedClasses {

    /**
     * 将被添加到生成的工厂类的名字中
     */
    private static final String SUFFIX = "Factory";

    private String qualifiedClassName;

    private Map<String, FactoryAnnotatedClass> itemsMap =
        new LinkedHashMap<String, FactoryAnnotatedClass>();
    ...

    public void generateCode(Elements elementUtils, Filer filer) throws IOException {

        TypeElement superClassName = elementUtils.getTypeElement(qualifiedClassName);
        String factoryClassName = superClassName.getSimpleName() + SUFFIX;

        JavaFileObject jfo = filer.createSourceFile(qualifiedClassName + SUFFIX);
        Writer writer = jfo.openWriter();
        JavaWriter jw = new JavaWriter(writer);

        // 写包名
        PackageElement pkg = elementUtils.getPackageOf(superClassName);
        if (!pkg.isUnnamed()) {
            jw.emitPackage(pkg.getQualifiedName().toString());
            jw.emitEmptyLine();
        } else {
            jw.emitPackage("");
        }
    }
}
```

```
Java注解处理器使用详解 - Flyer的后花园 - 博客频道 - CSDN.NET

jw.beginType(factoryClassName, "class", EnumSet.of(Modifier.PUBLIC));
jw.emitEmptyLine();
jw.beginMethod(qualifiedClassName, "create", EnumSet.of(Modifier.PUBLIC), "String", "id");

jw.beginControlFlow("if (id == null)");
jw.emitStatement("throw new IllegalArgumentException(\"id is null!\")");
jw.endControlFlow();

for (FactoryAnnotatedClass item : itemsMap.values()) {
    jw.beginControlFlow("if (%s.equals(id))", item.getId());
    jw.emitStatement("return new %s()", item.getTypeElement().getQualifiedName().toString());
    jw.endControlFlow();
    jw.emitEmptyLine();
}

jw.emitStatement("throw new IllegalArgumentException(\"Unknown id = \" + id)\");
jw.endMethod();
jw.endType();
jw.close();
}
```

注意：因为JavaWriter非常非常的流行，所以很多处理器、库、工具都依赖于JavaWriter。如果你使用依赖管理工具，例如maven或者gradle，假如一个库依赖的JavaWriter的版本比其他的库新，这将会导致一些问题。所以我建议你直接拷贝重新打包JavaWiter到你的注解处理器代码中（实际它只是一个Java文件）。

更新：JavaWrite现在已经被 JavaPoet 取代了。

### 处理循环

注解处理过程可能会多于一次。官方javadoc定义处理过程如下：

注解处理过程是一个有序的循环过程。在每次循环中，一个处理器可能被要求去处理那些在上一次循环中产生的源文件和类文件中的注解。第一次循环的输入是运行此工具的初始输入。这些初始输入，可以看成是虚拟的第0此的循环的输出。

一个简单的定义：一个处理循环是调用一个注解处理器的 process() 方法。对应到我们的工厂模式的例子中： FactoryProcessor 被初始化一次（不是每次循环都会新建处理器对象），然而，如果生成了新的源文件 process() 能够被调用多次。听起来有点奇怪不是么？原因是这样的，这些生成的文件中也可能包含@Factory注解，它们还将会被 FactoryProcessor 处理。

例如我们的 PizzaStore 的例子中将会经过3次循环处理：

| Round | Input  | Output           |
|-------|--|------------------|
| 1     | CalzonePizza.javaTiramisu.javaMargheritaPizza.java<br>Meal.java<br>PizzaStore.java | MealFactory.java |
| 2     | MealFactory.java   | — none —         |
| 3     | — none —   | — none —         |

我解释处理循环还有另外一个原因。如果你看一下我们的 FactoryProcessor 代码你就能注意到，我们收集数据和保存它们在一个私有的域中 Map<String, FactoryGroupedClasses> factoryClasses 。在第一轮中，我们检测到了MagheritaPizza, CalzonePizza和Tiramisu，然后生成了MealFactory.java。在第二轮中把MealFactory作为输入。因为在MealFactory中没有检测到@Factory注解，我们预期并没有错误，然而我们得到如下的信息：

```
Attempt to recreate a file for type com.hannesdorfmann.annotationprocessing101.factory.MealFactory
```

这个问题是因为我们没有清除 factoryClasses ，这意味着，在第二轮的 process() 中，任然保存着第一轮的数据，并且会尝试生成在第一轮中已经生成的文件，从而导致这个错误的出现。在我们的这个场景中，我们知道只有在第一轮中检查 @Factory 注解的类，所以我们可以简单的修复这个问题，如下：

```
@Override
public boolean process(Set<? extends TypeElement> annotations, RoundEnvironment roundEnv) {
    try {
        for (FactoryGroupedClasses factoryClass : factoryClasses.values()) {
            factoryClass.generateCode(elementUtils, filer);
        }

        // 清除factoryClasses
        factoryClasses.clear();

    } catch (IOException e) {
        error(null, e.getMessage());
    }
    ...
    return true;
}
```

我知道这有其他的方法来处理这个问题，例如我们也可以设置一个布尔值标签等。关键的点是：我们要记住注解处理过程是需要经过多轮处理的，并且你不能重载或者重新创建已经生成的源代码。

### 分离处理器和注解

如果你已经看了我们的[代码库](#)，你将发现我们组织我们的代码到两个maven模块中了。我们这么做是因为，我们想让我们的工厂模式的例子的使用者，在他们的工程中只编译注解，而包含处理器模块只是为了编译。有点晕？我们举个例子，如果我们只有一个包。如果另一个开发者想要把我们的工厂模式处理器用于他的项目中，他就必须包含 @Factory 注解和整个 FactoryProcessor 的代码（包括FactoryAnnotatedClass和FactoryGroupedClasses）

到他们项目中。我非常确定的是，他并不需要在他已经编译好的项目中包含处理器相关的代码。如果你是一个Android的开发者，你肯定听说过65k个方法的限制（即在一个dex文件中，只能寻址65000个方法）。如果你在FactoryProcessor中使用guava，并且把注解和处理器打包在一个包中，这样的话，Android APK安装包中不只是包含FactoryProcessor的代码，而也包含了整个guava的代码。Guava有大约20000个方法。所以分开注解和处理器是非常有意义的。

## 生成的类的实例化

你已经看到了，在这个 `PizzaStore` 的例子中，生成了 `MealFactory` 类，它和其他手写的Java类没有任何区别。进而，你需要就想其他Java对象，手动实例化它：

```
public class PizzaStore {

    private MealFactory factory = new MealFactory();

    public Meal order(String mealName) {
        return factory.create(mealName);
    }
    ...
}
```

如果你是一个Android的开发者，你应该也非常熟悉一个叫做 `ButterKnife` 的注解处理器。在ButterKnife中，你使用 `@InjectView` 注解Android的View。ButterKnifeProcessor生成一个 `MyActivity$$ViewInjector`，但是在ButterKnife你不需要手动调用 `new MyActivity$$ViewInjector()` 实例化一个ButterKnife注入的对象，而是使用 `Butterknife.inject(activity)`。ButterKnife内部使用反射机制来实例化 `MyActivity$$ViewInjector()` 对象：

```
try {
    Class<?> injector = Class.forName(clsName + "$$ViewInjector");
} catch (ClassNotFoundException e) { ... }
```

但是反射机制不是很慢么，我们使用注解处理来生成本地代码，会不会导致很多的反射性能的问题？的确，反射机制的性能确实是一个问题。然而，它不需要手动去创建对象，确实提高了开发者的开发速度。ButterKnife中有一个哈希表HashMap来缓存实例化过的对象。所以 `MyActivity$$ViewInjector` 只是使用反射机制实例化一次，第二次需要 `MyActivity$$ViewInjector` 的时候，就直接冲哈希表中获得。

`FragmentArgs` 非常类似于ButterKnife。它使用反射机制来创建对象，而不需要开发者手动来做这些。FragmentArgs在处理注解的时候生成一个特别的查找表类，它其实就是一种哈希表，所以整个FragmentArgs库只是在第一次使用的时候，执行一次反射调用，一旦整个 `Class.forName()` 的Fragemnt的参数对象被创建，后面的都是本地代码运行了。

作为一个注解注解处理器的开发者，这些都由你来决定，为其他的注解器使用者，在反射和可用性上找到一个好的平衡。

## 总结

到此，我希望你对注解处理过程有一个非常深刻的理解。我必须再次说明一下：注解处理器是一个非常强大的工具，减少了很多无聊的代码的编写。我也想提醒的是，注解处理器可以做到比我上面提到的工厂模式的例子复杂很多的事情。例如，泛型的类型擦除，因为注解处理器是发生在类型擦除（type erasure）之前的（译者注：类型擦除可以参考[这里](#)）。就像你所看到的，你在写注解处理的时候，有两个普遍的问题你需要处理：第一问题，如果你想在其他类中使用ElementUtils, TypeUtils和Messenger，你就必须把他们作为参数传进去。在我为Android开发的注解器 `AnnotatedAdapter` 中，我尝试使用Dagger（一个依赖注入库）来解决这个问题。在这个简单的处理中使用它听起来有点过头了，但是它确实很好用；第二个问题，你必须做查询 `Elements` 的操作。就想我之前提到的，处理Element就解析XML或者HTML一样。对于HTML你可以是用[jQuery](#)，如果在注解处理器中，有类似于jQuery的库那那绝对是酷毙了。如果你知道有类似的库，请在下面的评论告诉我。

请注意的是，在FactoryProcessor代码中有一些缺陷和陷阱。这些“错误”是我故意放进去的，是为了演示一些在开发过程中的常见错误（例如“Attempt to recreate a file”）。如果你想基于FactoryProcessor写你自己注解处理器，请不要直接拷贝粘贴这些陷阱过去，你应该从最开始就避免它们。

我在后续的博客中将会写注解处理器的单元测试，敬请关注。

顶  
0

踩  
0

- 上一篇
- 关于BeanUtils.copyProperties的用法和优缺点
- 下一篇
- 电商网站秒杀与抢购的系统架构

我的同类文章

j2se（54）

• 子类可以继承到父类上的注...

2016-09-27

阅读 21

• 获得java类的所有属性

2016-09-08

阅读 27

• Java中可重入锁ReentrantL...

2016-09-01

阅读 68

• Java 容器相关知识全面总结

2016-09-01

阅读 63

• java中注解的使用与实例(一)

2016-09-01

阅读 32

• HashMap与ConcurrentHas...

2016-09-12

阅读 15

• Lock ReentrantLock tryLoc...

2016-09-02

阅读 135

• 你真的了解一段Java程序的...

2016-09-01

阅读 54

• 关于BeanUtils.copyProperti...

2016-09-01

阅读 23



• Java读写txt文件中中文乱码问题

2016-08-31

阅读 63

更多文章

### 参考知识库

|   |                                       |   |  |
|---|---------------------------------------|---|--|
|  | <b>jQuery</b> 知识库<br>4361 关注   910 收录 |  | <b>Android</b> 知识库<br>19788 关注   1712 收录 |
|---|---------------------------------------|---|--|





**Java** 知识库  
16385 关注 | 1321 收录



算法与数据结构知识库  
7785 关注 | 3123 收录



**Java EE**知识库  
7307 关注 | 705 收录



**Java SE**知识库  
14375 关注 | 459 收录

猜你在找

查看评论

暂无评论

您还没有登录,请[\[登录\]](#)或[\[注册\]](#)

\* 以上用户言论只代表其个人观点，不代表CSDN网站的观点或立场

核心技术类目

全部主题   Hadoop   AWS   移动游戏   Java   Android   iOS   Swift   智能硬件   Docker   OpenStack  
VPN   Spark   ERP   IE10   Eclipse   CRM   JavaScript   数据库   Ubuntu   NFC   WAP   jQuery  
BI   HTML5   Spring   Apache   .NET   API   HTML   SDK   IIS   Fedora   XML   LBS   Unity  
Splashtop   UML   components   Windows Mobile   Rails   QEMU   KDE   Cassandra   CloudStack  
FTC   coremail   OPhone   CouchBase   云计算   iOS6   Rackspace   Web App   SpringSide   Maemo  
Compuware   大数据   aptech   Perl   Tornado   Ruby   Hibernate   ThinkPHP   HBase   Pure   Solr  
Angular   Cloud Foundry   Redis   Scala   Django   Bootstrap