JAVA 注解的基本原理

以前，『XML』是各大框架的青睐者，它以松耦合的方式完成了框架中几乎所有的配置，但是随着项目越来越庞大，『XML』的内容也越来越复杂，维护成本变高。

于是就有人提出来一种标记式高耦合的配置方式，『注解』。方法上可以进行注解，类上也可以注解，字段属性上也可以注解，反正几乎需要配置的地方都可以进行注解。

关于『注解』和『XML』两种不同的配置模式，争论了好多年了，各有各的优劣，注解可以提供更大的便捷性，易于维护修改，但耦合度高，而 XML 相对于注解则是相反的。

追求低耦合就要抛弃高效率，追求效率必然会遇到耦合。本文意不再辨析两者谁优谁劣，而在于以最简单的语言介绍注解相关的基本内容。

注解的本质

「java.lang.annotation.Annotation」接口中有这么一句话，用来描述『注解』。

> The common interface extended by all annotation types
> 所有的注解类型都继承自这个普通的接口（Annotation）

这句话有点抽象，但却说出了注解的本质。我们看一个 JDK 内置注解的定义：

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.SOURCE)
public @interface Override {

}
```

这是注解 @Override 的定义，其实它本质上就是：

```
public interface Override extends Annotation{

}
```

需要JAD反编译才行，其他的编译工具不会反编译成接口，参考：这篇文章。

没错，注解的本质就是一个继承了 Annotation 接口的接口。有关这一点，你可以去反编译任意一个注解类，你会得到结果的。

一个注解准确意义上来说，只不过是一种特殊的注释而已，如果没有解析它的代码，它可能连注释都不如。

而解析一个类或者方法的注解往往有两种形式，一种是编译期直接的扫描，一种是运行期反射。反射的事情我们待会说，而编译器的扫描指的是编译器在对 java 代码编译字节码的过程中会检测到某个类或者方法被一些注解修饰，这时它就会对于这些注解进行某些处理。

典型的就是注解 @Override，一旦编译器检测到某个方法被修饰了 @Override 注解，编译器就会检查当前方法的方法签名是否真正重写了父类的某个方法，也就是比较父类中是否具有一个同样的方法签名。

这一种情况只适用于那些编译器已经熟知的注解类，比如 JDK 内置的几个注解，而你自定义的注解，编译器是不知道你这个注解的作用的，当然也不知道该如何处理，往往只是会根据该注解的作用范围来选择是否编译进字节码文件，仅此而已。

元注解

『元注解』是用于修饰注解的注解，通常用在注解的定义上，例如：

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.SOURCE)
public @interface Override {

}
```

这是我们 @Override 注解的定义，你可以看到其中的 @Target，@Retention 两个注解就是我们所谓的『元注解』，『元注解』一般用于指定某个注解生命周期以及作用目标等信息。

JAVA 中有以下几个『元注解』：

- @Target：注解的作用目标
- @Retention：注解的生命周期
- @Documented：注解是否应当被包含在 JavaDoc 文档中
- @Inherited：是否允许子类继承该注解

其中，@Target 用于指明被修饰的注解最终可以作用的目标是谁，也就是指明，你的注解到底是用来修饰方法的？修饰类的？还是用来修饰字段属性的。

@Target 的定义如下：

```java
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.ANNOTATION_TYPE)
public @interface Target {
    /**
     * Returns an array of the kinds of elements an annota
     * can be applied to.
     * @return an array of the kinds of elements an annota
     * can be applied to
     */
    ElementType[] value();
}
```

我们可以通过以下的方式来为这个 value 传值：

```
@Target(value = {ElementType.FIELD})
```

被这个 @Target 注解修饰的注解将只能作用在成员字段上，不能用于修饰方法或者类。其中，ElementType 是一个枚举类型，有以下一些值：

- ElementType.TYPE：允许被修饰的注解作用在类、接口和枚举上
- ElementType.FIELD：允许作用在属性字段上
- ElementType.METHOD：允许作用在方法上
- ElementType.PARAMETER：允许作用在方法参数上
- ElementType.CONSTRUCTOR：允许作用在构造器上
- ElementType.LOCAL_VARIABLE：允许作用在本地局部变量上
- ElementType.ANNOTATION_TYPE：允许作用在注解上
- ElementType.PACKAGE：允许作用在包上

@Retention 用于指明当前注解的生命周期，它的基本定义如下：

```java
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.ANNOTATION_TYPE)
public @interface Retention {
    /**
     * Returns the retention policy.
     * @return the retention policy
     */
    RetentionPolicy value();
}
```

同样的，它也有一个 value 属性：

```
@Retention(value = RetentionPolicy.RUNTIME
```

这里的 RetentionPolicy 依然是一个枚举类型，它有以下几个枚举值可取：

- RetentionPolicy.SOURCE：当前注解编译期可见，不会写入 class 文件
- RetentionPolicy.CLASS：类加载阶段丢弃，会写入 class 文件
- RetentionPolicy.RUNTIME：永久保存，可以反射获取

@Retention 注解指定了被修饰的注解的生命周期，一种是只能在编译期可见，编译后会被丢弃，一种会被编译器编译进 class 文件中，无论是类或是方法，乃至字段，他们都是有属性表的，而 JAVA 虚拟机也定义了几种注解属性表用于存储注解信息，但是这种可见性不能带到方法区，类加载时会予以丢弃，最后一种则是永久存在的可见性。

剩下两种类型的注解我们日常用的不多，也比较简单，这里不再详细的进行介绍了，你只需要知道他们各自的作用即可。@Documented 注解修饰的注解，当我们执行 JavaDoc 文档打包时会被保存进 doc 文档，反之将在打包时丢弃。@Inherited 注解修饰的注解是具有可继承性的，也就说我们的注解修饰了一个类，而该类的子类将自动继承父类的该注解。

JAVA 的内置三大注解

除了上述四种元注解外，JDK 还为我们预定义了另外三种注解，它们是：

- @Override
- @Deprecated
- @SuppressWarnings

@Override 注解想必是大家很熟悉的了，它的定义如下：

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.SOURCE)
```

```
public @interface Override {
}
```

它没有任何的属性，所以并不能存储任何其他信息。它只能作用于方法之上，编译结束后将被丢弃。

所以你看，它就是一种典型的『标记式注解』，仅被编译器可知，编译器在对 java 文件进行编译成字节码的过程中，一旦检测到某个方法上被修饰了该注解，就会去匹对父类中是否具有一个同样方法签名的函数，如果不是，自然不能通过编译。

@Deprecated 的基本定义如下：

```
 */
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target(value={CONSTRUCTOR, FIELD, LOCAL_VARIABLE, METHOD, PACKAGE, PARAMETER, TYPE})
public @interface Deprecated {
}
```

依然是一种『标记式注解』，永久存在，可以修饰所有的类型，作用是，标记当前的类或者方法或者字段等已经不再被推荐使用了，可能下一次的 JDK 版本就会删除。

当然，编译器并不会强制要求你做什么，只是告诉你 JDK 已经不再推荐使用当前的方法或者类了，建议你使用某个替代者。

@SuppressWarnings 主要用来压制 java 的警告，它的基本定义如下：

```
 */
@Target({TYPE, FIELD, METHOD, PARAMETER, CONSTRUCTOR, LOCAL_VARIABLE})
@Retention(RetentionPolicy.SOURCE)
public @interface SuppressWarnings {
    /**
     * The set of warnings that are to be suppressed by the compiler in the
     * annotated element.  Duplicate names are permitted.  The second and
     * successive occurrences of a name are ignored.  The presence of
     * unrecognized warning names is <i>not</i> an error: Compilers must
     * ignore any warning names they do not recognize.  They are, however,
     * free to emit a warning if an annotation contains an unrecognized
     * warning name.
     *
     * <p> The string {@code "unchecked"} is used to suppress
     * unchecked warnings. Compiler vendors should document the
     * additional warning names they support in conjunction with this
     * annotation type. They are encouraged to cooperate to ensure
     * that the same names work across multiple compilers.
     * @return the set of warnings to be suppressed
     */
    String[] value();
}
```

它有一个 value 属性需要你主动的传值，这个 value 代表一个什么意思呢，这个 value 代表的就是需要被压制的警告类型。例如：

```
public static void main(String[] args) {
    Date date = new Date(2018, 7, 11);
}
```

这么一段代码，程序启动时编译器会报一个警告。

> Warning:(8, 21) java: java.util.Date 中的 Date(int,int,int) 已过时

而如果我们不希望程序启动时，编译器检查代码中过时的方法，就可以使用 @SuppressWarnings 注解并给它的 value 属性传入一个参数值来压制编译器的检查。

```
@SuppressWarning(value = "deprecated")
public static void main(String[] args) {
    Date date = new Date(2018, 7, 11);
}
```

这样你就会发现，编译器不再检查 main 方法下是否有过时的方法调用，也就压制了编译器对于这种警告的检查。

当然，JAVA 中还有很多的警告类型，他们都会对应一个字符串，通过设置 value 属性的值即可压制对于这一类警告类型的检查。

自定义注解的相关内容就不再赘述了，比较简单，通过类似以下的语法即可自定义一个注解。

```
public @interface InnotationName{

}
```

当然，自定义注解的时候也可以选择性的使用元注解进行修饰，这样你可以更加具体的指定你的注解的生命周期、作用范围等信息。

注解与反射

上述内容我们介绍了注解使用上的细节，也简单提到，「注解的本质就是一个继承了 Annotation 接口的接口」，现在我们就来从虚拟机的层面看看，注解的本质到底是什么。

首先，我们自定义一个注解类型：

```
@Target(value = {ElementType.FIELD,ElementType.METHOD})
@Retention(value = RetentionPolicy.RUNTIME)
public @interface Hello {
    String value();
}
```

这里我们指定了 Hello 这个注解只能修饰字段和方法，并且该注解永久存活，以便我们反射获取。

之前我们说过，虚拟机规范定义了一系列和注解相关的属性表，也就是说，无论是字段、方法或是类本身，如果被注解修饰了，就可以被写进字节码文件。属性表有以下几种：

- RuntimeVisibleAnnotations：运行时可见的注解
- RuntimeInVisibleAnnotations：运行时不可见的注解
- RuntimeVisibleParameterAnnotations：运行时可见的方法参数注解
- RuntimeInVisibleParameterAnnotations：运行时不可见的方法参数注解
- AnnotationDefault：注解类元素的默认值

给大家看虚拟机的这几个注解相关的属性表的目的在于，让大家从整体上构建一个基本的印象，注解在字节码文件中是如何存储的。

所以，对于一个类或者接口来说，Class 类中提供了以下一些方法用于反射注解。

- getAnnotation：返回指定的注解
- isAnnotationPresent：判定当前元素是否被指定注解修饰
- getAnnotations：返回所有的注解
- getDeclaredAnnotation：返回本元素的指定注解
- getDeclaredAnnotations：返回本元素的所有注解，不包含父类继承而来的

方法、字段中相关反射注解的方法基本是类似的，这里不再赘述，我们下面看一个完整的例子。

首先，设置一个虚拟机启动参数，用于捕获 JDK 动态代理类。

> -Dsun.misc.ProxyGenerator.saveGeneratedFiles=true

然后 main 函数。

```
public class Test {
    @Hello("hello")
    public static void main(String[] args) throws NoSuchMethodException {
        Class cls = Test.class;
        Method method = cls.getMethod( name: "main", String[].class);
        Hello hello = method.getAnnotation(Hello.class);
    }
}
```

我们说过，注解本质上是继承了 Annotation 接口的接口，而当你通过反射，也就是我们这里的 getAnnotation 方法去获取一个注解类实例的时候，其实 JDK 是通过动态代理机制生成一个实现我们注解（接口）的代理类。

我们运行程序后，会看到输出目录里有这么一个代理类，反编译之后是这样的：

```
public final class $Proxy1 extends Proxy
    implements Hello
{

    public $Proxy1(InvocationHandler invocationhandler)
    {
        super(invocationhandler);
    }
```

```
public final String value()
{
    try
    {
        return (String)super.h.invoke(this, m3, null);
    }
    catch(Error _ex) { }
    catch(Throwable throwable)
    {
        throw new UndeclaredThrowableException(throwable);
    }
}
```

代理类实现接口 Hello 并重写其所有方法，包括 value 方法以及接口 Hello 从 Annotation 接口继承而来的方法。

而这个关键的 InvocationHandler 实例是谁?

AnnotationInvocationHandler 是 JAVA 中专门用于处理注解的 Handler， 这个类的设计也非常有意思。

```
class AnnotationInvocationHandler implements InvocationHandler, Serializable {
    private static final long serialVersionUID = 6182022883658399397L;
    private final Class<? extends Annotation> type;
    private final Map<String, Object> memberValues;
    private transient volatile Method[] memberMethods = null;

    AnnotationInvocationHandler(Class<? extends Annotation> var1, Map<String, Object>
        Class[] var3 = var1.getInterfaces();
        if (var1.isAnnotation() && var3.length == 1 && var3[0] == Annotation.class) {
            this.type = var1;
            this.memberValues = var2;
        } else {
            throw new AnnotationFormatError("Attempt to create proxy for a non-annota
        }
    }
```

这里有一个 memberValues，它是一个 Map 键值对，键是我们注解属性名称，值就是该属性当初被赋上的值。

```
public Object invoke(Object var1, Method var2, Object[] var3) {
    String var4 = var2.getName();
    Class[] var5 = var2.getParameterTypes();
    if (var4.equals("equals") && var5.length == 1 && var5[0] == Object.class) {
        return this.equalsImpl(var3[0]);
    } else if (var5.length != 0) {
        throw new AssertionError( detailMessage: "Too many parameters for an annotatio
    } else {
        byte var7 = -1;
        switch(var4.hashCode()) {
        case -1776922004:
            if (var4.equals("toString")) {
                var7 = 0;
            }
            break;
        case 147696667:
            if (var4.equals("hashCode")) {
                var7 = 1;
            }
            break;
        case 1444986633:
            if (var4.equals("annotationType")) {
                var7 = 2;
            }
        }
```

```
            switch(var7) {
            case 0:
                return this.toStringImpl();
            case 1:
                return this.hashCodeImpl();
            case 2:
                return this.type;
            default:
                Object var6 = this.memberValues.get(var4);
                if (var6 == null) {
                    throw new IncompleteAnnotationException(this.type, var4);
                } else if (var6 instanceof ExceptionProxy) {
                    throw ((ExceptionProxy)var6).generateException();
                } else {
                    if (var6.getClass().isArray() && Array.getLength(var6) != 0)
                        var6 = this.cloneArray(var6);
                    }

                    return var6;
                }
            }
        }
    }
```

而这个 invoke 方法就很有意思了，大家注意看，我们的代理类代理了 Hello 接口中所有的方法，所以对于代理类中任何方法的调用都会被转到这里来。

var2 指向被调用的方法实例，而这里首先用变量 var4 获取该方法的简明名称，接着 switch 结构判断当前的调用方法是谁，如果是 Annotation 中的四大方法，将 var7 赋上特定的值。

如果当前调用的方法是 toString，equals，hashCode，annotationType 的话，AnnotationInvocationHandler 实例中已经预定义好了这些方法的实现，直接调用即可。

那么假如 var7 没有匹配上这四种方法，说明当前的方法调用的是自定义注解字节声明的方法，例如我们 Hello 注解的 value 方法。**这种情况下，将从我们的注解 map 中获取这个注解属性对应的值。**

其实，JAVA 中的注解设计个人觉得有点反人类，明明是属性的操作，非要用方法来实现。当然，如果你有不同的见解，欢迎留言探讨。

最后我们再总结一下整个反射注解的工作原理：

首先，我们通过键值对的形式可以为注解属性赋值，像这样：@Hello（value = "hello"）。

接着，你用注解修饰某个元素，编译器将在编译期扫描每个类或者方法上的注解，会做一个基本的检查，你的这个注解是否允许作用在当前位置，最后会将注解信息写入元素的属性表。

然后，当你进行反射的时候，虚拟机将所有生命周期在 RUNTIME 的注解取出来放到一个 map 中，并创建一个 AnnotationInvocationHandler 实例，把这个 map 传递给它。

最后，虚拟机将采用 JDK 动态代理机制生成一个目标注解的代理类，并初始化好处理器。

那么这样，一个注解的实例就创建出来了，它本质上就是一个代理类，你应当去理解好 AnnotationInvocationHandler 中 invoke 方法的实现逻辑，这是核心。一句话概括就是，**通过方法名返回注解属性值。**