05 注解

本节我们将介绍Java程序的一种特殊“注释”——注解（Annotation）。

使用注解

什么是注解（Annotation）？注解是放在Java源码的类、方法、字段、参数前的一种特殊“注释”：

```
// this is a component:
@Resource("hello")
public class Hello {
    @Inject
    int n;

    @PostConstruct
    public void hello(@Param String name) {
        System.out.println(name);
    }

    @Override
    public String toString() {
        return "Hello";
    }
}
```

注释会被编译器直接忽略，注解则可以被编译器打包进入class文件，因此，注解是一种用作标注的“元数据”。

注解的作用

从JVM的角度看，注解本身对代码逻辑没有任何影响，如何使用注解完全由工具决定。

Java的注解可以分为三类：

第一类是由编译器使用的注解，例如：

- `@Override`：让编译器检查该方法是否正确地实现了覆写；
- `@SuppressWarnings`：告诉编译器忽略此处代码产生的警告。

这类注解不会被编译进入`.class`文件，它们在编译后就被编译器扔掉了。

第二类是由工具处理`.class`文件使用的注解，比如有些工具会在加载class的时候，对class做动态修改，实现一些特殊的功能。这类注解会被编译进入`.class`文件，但加载结束后并不会存在于内存中。这类注解只被一些底层库使用，一般我们不必自己处理。

第三类是在程序运行期能够读取的注解，它们在加载后一直存在于JVM中，这也是最常用的注解。例如，一个配置了`@PostConstruct`的方法会在调用构造方法后自动被调用（这是Java代码读取该注解实现的功能，JVM并不会识别该注解）。

定义一个注解时，还可以定义配置参数。配置参数可以包括：

- 所有基本类型；
- String；
- 枚举类型；
- 基本类型、String以及枚举的数组。

因为配置参数必须是常量，所以，上述限制保证了注解在定义时就已经确定了每个参数的值。

注解的配置参数可以有默认值，缺少某个配置参数时将使用默认值。

此外，大部分注解会有一个名为`value`的配置参数，对此参数赋值，可以只写常量，相当于省略了value参数。

如果只写注解，相当于全部使用默认值。

举个栗子，对以下代码：

```
public class Hello {
    @Check(min=0, max=100, value=55)
    public int n;

    @Check(value=99)
    public int p;

    @Check(99) // @Check(value=99)
    public int x;

    @Check
    public int y;
}
```

`@Check`就是一个注解。第一个`@Check(min=0, max=100, value=55)`明确定义了三个参数，第二个`@Check(value=99)`只定义了一个`value`参数，它实际上和`@Check(99)`是完全一样的。最后一个`@Check`表示所有参数都使用默认值。

小结

- 注解（Annotation）是Java语言用于工具处理的标注；
- 注解可以配置参数，没有指定配置的参数使用默认值；
- 如果参数名称是`value`，且只有一个参数，那么可以省略参数名称。

定义注解

Java语言使用`@interface`语法来定义注解（`Annotation`），它的格式如下：

```
public @interface Report {
    int type() default 0;
    String level() default "info";
    String value() default "";
}
```

注解的参数类似无参数方法，可以用`default`设定一个默认值（强烈推荐）。最常用的参数应当命名为`value`。

元注解

有一些注解可以修饰其他注解，这些注解就称为元注解（meta annotation）。Java标准库已经定义了一些元注解，我们只需要使用元注解，通常不需要自己去编写元注解。

@Target

最常用的元注解是`@Target`。使用`@Target`可以定义`Annotation`能够被应用于源码的哪些位置：

- 类或接口：`ElementType.TYPE`；
- 字段：`ElementType.FIELD`；
- 方法：`ElementType.METHOD`；
- 构造方法：`ElementType.CONSTRUCTOR`；
- 方法参数：`ElementType.PARAMETER`。

例如，定义注解`@Report`可用在方法上，我们必须添加一个`@Target(ElementType.METHOD)`：

```
@Target(ElementType.METHOD)
public @interface Report {
    int type() default 0;
    String level() default "info";
    String value() default "";
}
```

定义注解`@Report`可用在方法或字段上，可以把`@Target`注解参数变为数组`{ ElementType.METHOD, ElementType.FIELD }`：

```
@Target({
    ElementType.METHOD,
    ElementType.FIELD
})
public @interface Report {
    // ...
}
```

实际上`@Target`定义的`value`是`ElementType[]`数组，只有一个元素时，可以省略数组的写法。

@Retention

另一个重要的元注解`@Retention`定义了`Annotation`的生命周期：

- 仅编译期：`RetentionPolicy.SOURCE`；
- 仅class文件：`RetentionPolicy.CLASS`；
- 运行期：`RetentionPolicy.RUNTIME`。

如果`@Retention`不存在，则该`Annotation`默认为`CLASS`。因为通常我们自定义的`Annotation`都是`RUNTIME`，所以，务必要加上`@Retention(RetentionPolicy.RUNTIME)`这个元注解：

```
@Retention(RetentionPolicy.RUNTIME)
public @interface Report {
    int type() default 0;
    String level() default "info";
    String value() default "";
}
```

@Repeatable

使用`@Repeatable`这个元注解可以定义`Annotation`是否可重复。这个注解应用不是特别广泛。

```
@Repeatable
@Target(ElementType.TYPE)
public @interface Report {
    int type() default 0;
    String level() default "info";
    String value() default "";
}
```

经过`@Repeatable`修饰后，在某个类型声明处，就可以添加多个`@Report`注解：

```
@Report(type=1, level="debug")
@Report(type=2, level="warning")
public class Hello {
}
```

@Inherited

使用`@Inherited`定义子类是否可继承父类定义的`Annotation`。`@Inherited`仅针对`@Target(ElementType.TYPE)`类型的`annotation`有效，并且仅针对`class`的继承，对`interface`的继承无效：

```
@Inherited
@Target(ElementType.TYPE)
public @interface Report {
    int type() default 0;
    String level() default "info";
    String value() default "";
}
```

在使用的时候，如果一个类用到了`@Report`：

```
@Report(type=1)
public class Person {
}
```

则它的子类默认也定义了该注解：

```
public class Student extends Person {
}
```

如何定义Annotation

我们总结一下定义`Annotation`的步骤：

第一步，用`@interface`定义注解：

```
public @interface Report {
}
```

第二步，添加参数、默认值：

```
public @interface Report {
    int type() default 0;
    String level() default "info";
    String value() default "";
}
```

把最常用的参数定义为`value()`，推荐所有参数都尽量设置默认值。

第三步，用元注解配置注解：

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
public @interface Report {
    int type() default 0;
    String level() default "info";
    String value() default "";
}
```

其中，必须设置`@Target`和`@Retention`，`@Retention`一般设置为`RUNTIME`，因为我们自定义的注解通常要求在运行期读取。一般情况下，不必写`@Inherited`和`@Repeatable`。

小结

- Java使用`@interface`定义注解：
- 可定义多个参数和默认值，核心参数使用`value`名称；
- 必须设置`@Target`来指定`Annotation`可以应用的范围；
- 应当设置`@Retention(RetentionPolicy.RUNTIME)`便于运行期读取该`Annotation`。

处理注解

Java的注解本身对代码逻辑没有任何影响。根据`@Retention`的配置：

- `SOURCE`类型的注解在编译期就被丢掉了；
- `CLASS`类型的注解仅保存在class文件中，它们不会被加载进JVM；
- `RUNTIME`类型的注解会被加载进JVM，并且在运行期可以被程序读取。

如何使用注解完全由工具决定。`SOURCE`类型的注解主要由编译器使用，因此我们一般只使用，不编写。`CLASS`类型的注解主要由底层工具库使用，涉及到class的加载，一般我们很少用到。只有`RUNTIME`类型的注解不但要使用，还经常需要编写。

因此，我们只讨论如何读取`RUNTIME`类型的注解。

因为注解定义后也是一种`class`，所有的注解都继承自`java.lang.annotation.Annotation`，因此，读取注解，需要使用反射API。

Java提供的使用反射API读取`Annotation`的方法包括：

判断某个注解是否存在于`Class`、`Field`、`Method`或`Constructor`：

- `Class.isAnnotationPresent(Class)`
- `Field.isAnnotationPresent(Class)`
- `Method.isAnnotationPresent(Class)`
- `Constructor.isAnnotationPresent(Class)`

例如：

```
// 判断@Report是否存在于Person类:
Person.class.isAnnotationPresent(Report.class);
```

使用反射API读取Annotation：

- `Class.getAnnotation(Class)`
- `Field.getAnnotation(Class)`
- `Method.getAnnotation(Class)`

- `Constructor.getAnnotation(Class)`

例如：

```
// 获取Person定义的@Report注解:
Report report = Person.class.getAnnotation(Report.class);
int type = report.type();
String level = report.level();
```

使用反射API读取`Annotation`有两种方法。方法一是先判断`Annotation`是否存在，如果存在，就直接读取：

```
Class cls = Person.class;
if (cls.isAnnotationPresent(Report.class)) {
    Report report = cls.getAnnotation(Report.class);
    // ...
}
```

第二种方法是直接读取`Annotation`，如果`Annotation`不存在，将返回`null`：

```
Class cls = Person.class;
Report report = cls.getAnnotation(Report.class);
if (report != null) {
   ...
}
```

读取方法、字段和构造方法的`Annotation`和Class类似。但要读取方法参数的`Annotation`就比较麻烦一点，因为方法参数本身可以看成一个数组，而每个参数又可以定义多个注解，所以，一次获取方法参数的所有注解就必须用一个二维数组来表示。例如，对于以下方法定义的注解：

```
public void hello(@NotNull @Range(max=5) String name,
@NotNull String prefix) {
}
```

要读取方法参数的注解，我们先用反射获取`Method`实例，然后读取方法参数的所有注解：

```
// 获取Method实例:
Method m = ...
// 获取所有参数的Annotation:
Annotation[][] annos = m.getParameterAnnotations();
// 第一个参数（索引为0）的所有Annotation:
Annotation[] annosOfName = annos[0];
for (Annotation anno : annosOfName) {
    if (anno instanceof Range) { // @Range注解
        Range r = (Range) anno;
    }
    if (anno instanceof NotNull) { // @NotNull注解
```

```
            NotNull n = (NotNull) anno;
        }
    }
```

使用注解

注解如何使用，完全由程序自己决定。例如，JUnit是一个测试框架，它会自动运行所有标记为`@Test`的方法。

我们来看一个`@Range`注解，我们希望用它来定义一个`String`字段的规则：字段长度满足`@Range`的参数定义：

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.FIELD)
public @interface Range {
    int min() default 0;
    int max() default 255;
}
```

在某个JavaBean中，我们可以使用该注解：

```
public class Person {
    @Range(min=1, max=20)
    public String name;

    @Range(max=10)
    public String city;
}
```

但是，定义了注解，本身对程序逻辑没有任何影响。我们必须自己编写代码来使用注解。这里，我们编写一个`Person`实例的检查方法，它可以检查`Person`实例的`String`字段长度是否满足`@Range`的定义：

```
void check(Person person) throws IllegalArgumentException,
ReflectiveOperationException {
    // 遍历所有Field:
    for (Field field : person.getClass().getFields()) {
        // 获取Field定义的@Range:
        Range range = field.getAnnotation(Range.class);
        // 如果@Range存在:
        if (range != null) {
            // 获取Field的值:
            Object value = field.get(person);
            // 如果值是String:
            if (value instanceof String) {
                String s = (String) value;
                // 判断值是否满足@Range的min/max:
                if (s.length() < range.min() || s.length()
> range.max()) {
```

```
                    throw new
IllegalArgumentException("Invalid field: " +
field.getName());
                }
            }
        }
    }
}
```

这样一来，我们通过`@Range`注解，配合`check()`方法，就可以完成`Person`实例的检查。注意检查逻辑完全是我们自己编写的，JVM不会自动给注解添加任何额外的逻辑。

练习

使用`@Range`注解来检查Java Bean的字段。如果字段类型是`String`，就检查`String`的长度，如果字段是`int`，就检查`int`的范围。

下载练习：annotation-range-check （推荐使用IDE练习插件快速下载）

小结

- 可以在运行期通过反射读取`RUNTIME`类型的注解，注意千万不要漏写`@Retention(RetentionPolicy.RUNTIME)`，否则运行期无法读取到该注解。
- 可以通过程序处理注解来实现相应的功能：
    - 对JavaBean的属性值按规则进行检查；
    - JUnit会自动运行`@Test`标记的测试方法。