第 25 章

注解

注解是添加到程序源代码中的结构化信息。它们与注释很像，可以出现在程序的任何地方并附加到任意变量、方法、表达式或其他程序元素上。但与注释又不一样，它们具有结构性，因此更易于机器处理。

本章说明了 Scala 中如何使用注解。演示了它的一般语法及若干标准注解的使用方式。

但本章并不会说明如何编写新的注解处理工具，因为它不在本书的范围之内，第 29 章将演示一种，但这并非唯一的技巧。本章主要关注的是如何使用注解，因为这比定义新的注解处理器更为寻常。

25.1 为什么要有注解?

除了编译和运行之外，还可以对程序做许多事。例如：

1. 使用 Scaladoc 自动产生文档。
2. 漂亮地打印出符合你偏爱风格的代码。
3. 代码的常见错误检查，如：打开了文件却没（在全部逻辑分支中）关闭。
4. 实验性类型检查，例如副作用管理或所有权属性确认。

这类工具（程序）被称为元编程（meta-programming）工具，因为它们是把其他程序当做输入程序。注解通过让程序员把需要传递给工具的指令标示在他们的源代码中以提供对这些工具的支持。这种指令能让工具比在没有用户输入的情况下更为有效。例如，注解可以在以下方面增强前面所列的功能：

1. 文档生成器可以得到指示，把某些方法在文档中记为已经废除。
2. 排版打印器可以得到指示，跳过已被仔细手工格式化的程序部分。
3. 未关闭文件检查器可以得到指示，忽略已被人工验证为关闭的特定文件。
4. 副作用检查器可以得到指示，验证指定的方法是否有副作用。

所有的这些情况中，在理论上，编程语言提供插入额外信息的方式是可能的，而实际上，它们大多数都是直接从这种或那种语言中得到支持。然而对于一种语言来说实在有太多这样的工具能够直接支持所有的用例，以至于编译器已经完全忽略了所有的信息，毕竟只要代码能运行即可。

Scala 在这种问题上的哲学是，在核心语言中包含最少的、正交的支持以便于编写大量的元编程工

具。在这里，最少的支持就是注解系统。编译器只知道一个特性：注解，但并不把任何含义与注解个体联系在一起。每种元编程工具都可以以此为基础，定义并使用自己特定的注解。

25.2 注解语法

注解的典型使用方式如下：

```
@deprecated def bigMistake() = //...
```

注解为@deprecated 部分，它应用于 bigMistake[1]方法（省略——细节过于尴尬）的全体。这种情况下，方法被标记为 bigMistake 作者不希望你使用的东西。或许在代码的未来版本中 bigMistake 将被整体移除。

506

上面的例子里，是方法被加上了@deprecated 注解。注解还可以用在其他地方，它可以存在于任何种类的声明或定义之上，包括 val、var、def、class、object、trait，以及 type。并应用于跟随其后的声明或定义的整体：

```
@deprecated class QuickAndDirty {
  //...
}
```

注解还可以应用于表达式，就如在模式匹配中的@unchecked 注解（参见第 15 章）。这需要在表达式之后加上冒号（：），然后写上注解内容。依照语法构成来看，就好像注解被用作类型一样：

```
(e: @unchecked) match {
  // 非完备的样本（列表）
}
```

最后，注解还可以放在类型上。本章后续部分将会描述加注解的类型。

到目前为止，所见的注解都只是简单的 at 符号（@）跟着注解类。这种简单注解很普遍也很有用，但实际上注解具有更为丰富的一般形式：

$$@annot(exp_1, exp_2, ...)\ \{val\ name_1=const_1, ..., val\ name_n=const_n\}$$

annot 是指注解的类，这是所有的注解都必须包含的。exp 部分是注解的参数，类似于@deprecated 这样的注解是不需要任何参数的。你通常可以省略括号，但如果愿意的话也可以写成@deprecated()。而对于的确具有参数的注解来说，参数需要放在括号内，如@serial(1234)。

507

需要提供给注解的参数的具体形式取决于注解类。多数注解处理器只允许提供立即常量如 123 或 "hello"。不过只要类型检查可以通过，编译器本身支持任意的表达式。因此，有些注解类可以利用这一点以便让你做比如说引用作用域内的其他变量这样的事：

```
@cool val normal = "Hello"
@coolerThan(normal) val fonzy = "Heeyyy"
```

[1]译注：严重的错误。

一般语法内的 name = const 这种名值对形式可以用于更为复杂的、具有可选参数的注解。这些参数可选，并且可以按照任意次序指定。为了便于处理，等号右侧必须是常量。

25.3 标准注解

Scala 包含了若干个标准注解。它们是为了那些被广泛应用的，并足以将其放在语言规格说明书中的，但又不够基础到值得为其提供专属语法的特性而准备的。随着时间流逝，或许会有新的注解以同样的方式加入到标准中。

废弃

有些时候你可能会编写了宁愿没有写过的类或方法。尽管如此，一旦代码可用之后，其他人编写的代码就有可能会调用这些方法。因此，你不能简单的删除了事，因为这将导致其他人的代码编译失败。

废弃注解能够让你优雅地移除被证明出是错误的方法或类。如果你把方法或类标记为废弃，那么任何调用方法或类的人将收到废弃警告。他们最好能够注意到这个警告并更新他们的代码！经过了一段合适的时间之后，你感觉可以安全地假设所有理性的客户都已经停止了废弃类或方法的访问，就可以安全地删除这段代码。

要把方法标记为废弃，只要在方法之前加上@deprecated 即可。如：

```
@deprecated def bigMistake() =    //...
```

这个注解将在 Scala 代码访问该方法的时候让 Scala 编译器弹出废弃警告。

易变字段

并发编程不能很好地与共享的可变状态混用。为此，Scala 的并发重点关注于消息传递的支持，对共享的可变状态支持极少。细节参见第 30 章。

尽管如此，有些时候程序员仍然在并发编程中需要用到可变的状态。@volatile 注解对此提供了帮助。它可以通知编译器相关变量将被多个线程使用。实现这样的变量会使得它在读和写上变慢，但多线程的访问将表现得更具可预见性。

@volatile 关键字在不同平台上提供的保证也不同。不过，在 Java 平台上，它与你在 Java 代码中编写字段，并用 Java 的 volatile 修饰符对它做标记的行为是一致的。

二进制序列化

许多语言包含了二进制序列化的架构。序列化架构可以帮助你把对象变成字节流，反之亦然。如果你想要把对象保存到磁盘上或者通过网络发送，这会很有用。XML 有助于解决同一问题（参见第 26 章），但两者在速度、空间使用、灵活性以及可移植性方面有不同的侧重。

Scala 不具有自己的序列化框架。相反，你应该使用自己的基础平台上的框架。Scala 能够提供的是

为各种平台准备的三个注解。同样，Scala 的基于 Java 平台的编译器把这些注解以 Java 的方式作了解释（参见第 29 章）。

第一个注解指明类究竟是否可以序列化。多数类都可以序列化，但并非全部类。例如，socket 或 GUI 窗口句柄不能被序列化。默认情况下，类被认为是不可序列化的，你需要把`@serializable`注解添加到你希望序列化的类上。

第二个注解有助于处理随时间改变而改变的可序列化类。你可以通过添加如`@SerialVersionUID(1234)`这样的注解，把序号附加在类的当前版本上，这里 1234 应该用你选择的序号替代，而框架应该把这个数存在产生的字节流中。当你后来重载字节流并尝试把它转换为对象的时候，框架可以检查当前的类版本与字节流里的版本是否具有相同的版本号。如果想要对你的类做对于序列化不能相兼容的改变的话，你可以修改版本号，框架将自动拒绝加载类的旧实例。

509

最后，Scala 给不应被序列化的字段提供了`@transient`注解。如果你标记字段为`@transient`，那么框架就不应该在序列化该字段所在的对象时执行序列化。在对象被加载的时候，注解为`@transient`的字段也将根据其类型恢复为默认值。

自动的 get 和 set 方法

Scala 代码通常不需要为字段提供显式的 get 和 set 方法，因为 Scala 的字段访问和方法调用语法是混在一起的。但某些特定平台的框架需要 get 和 set 方法。为此，Scala 提供了`@scala.reflect.BeanProperty`注解。如果你把这个注解加到字段上，编译器就会为你自动产生 get 和 set 方法。如果你注解名为 crazy 的字段，那么 get 方法将被命名为 getCrazy，而 set 方法会被命名为 setCrazy。

产生的 get 和 set 方法只在完成编译之后才可用。因此，不能在同一时间编译的代码中调用注解字段的这些 get 和 set 方法。不过这在实际使用中不是问题，因为在 Scala 代码中你可以直接访问这些字段。这个特性只是为了支持那些需要有正规的 get 和 set 方法的架构，并且明显你不会在同一时间既编译框架又编译使用它的代码。

不检查

`@unchecked`注解由编译器在模式匹配的时候解释。它告诉编译器不要担心 match 表达式忽略了某些情况。细节参见 15.5 节。

25.4 小结

本章描述了你最应该了解的注解与平台无关的几方面知识。首先讲述了注解的语法，因为注解的使用远比定义新的注解更为寻常。其次演示了如何使用标准 Scala 编译器支持的若干注解，包括`@deprecated`、`@volatile`、`@serializable`、`@BeanProperty`及`@unchecked`。

510

第 29 章将提供更多特定于 Java 的注解信息。包括仅在 Java 中可用的注解，如何与基于 Java 的注解进行互操作，以及如何使用基于 Java 的机制在 Scala 中定义和处理注解。

511