测试和注解

一、断言

1. 断言和测试是检查代码行为符合预期的两种重要手段。
2. `Scala` 中，断言是对预定义方法`assert` （它定义在`Predef` 单例对象中）的调用：`assert(cond)` 。

   如果`cond` 条件不满足，则该表达式抛出`AssertionError` 异常。

   `assert` 还有一个版本：`assert(cond,explain)` 。如果`cond` 条件不满足，则抛出包含给定`explain` 的`AssertionError` 。其中`explain` 的类型为`Any` ，因此可以传入任何对象。`assert` 方法将调用`explain` 的`toString` 方法来获取一个字符串，放入到`AssertionError` 中。

   ```
   def sqrt(n:Double) = {
       assert(n>=0)                                  // 断言： n>=0
       assert(n>=0,"number must >=0: " + n)          // 提供了解释
       /* 详细实现  */
   }
   ```

3. 除了`assert` ，`Predef` 还提供了`ensuring` 方法。该方法可以用于任何结果类型，这得益于一个隐式类型转换。

   ```
   class C(val n:Double)
   {
       def scale(factor:Double) : C = {
           new C(this.n * factor) ensuring (_.n >=0 ) // 这里 ensuring 采用操作符语法
           /*
           等价于：
           val c = C(this.n * factor)          // 创建的对象
           val f = (x:C) => x.n >= 0           // 前提条件函数
           c.ensuring(f)                       // 返回 c 或者抛出异常
           */
       }
   }
   ```

   `ensuring` 方法接收一个函数参数，该函数称作前提条件函数。前提条件函数接受调用对象（即上面的`c` 对象）并返回布尔类型。

   `ensuring` 方法将调用对象传递给这个前提条件函数，如果前提条件函数返回`true` ，则`ensuring` 正常返回结果；如果返回`false` ，则`ensuring` 将抛出`AssertionError` 。
4. 断言可以通过`JVM` 的命令行参数`-ea` 打开，可以通过参数`-da` 关闭。打开时，断言就像是一个个小测试，用的是运行时得到的真实数据。

二、测试

1. 用 `Scala` 写测试有多种选择，包括 `Java` 工具如 `JUnit` 和 `TestNG` ，以及 `Scala` 工具如 `ScalaTest,specs2,ScalaCheck` 等。

2.1 ScalaTest

1. `ScalaTest` 是最灵活的 `Scala` 测试框架，可以很容易定制它来解决不同的问题。
2. `ScalaTest` 核心概念是套件 `suite`，即测试的集合。所谓的测试 `test` 可以是任何带有名字，可以被启动，要么成功、要么失败、要么被暂停、要么被取消的代码。

   在 `ScalaTest` 中，`Suite` 特质是核心组合单元。`Suite` 声明了一组“生命周期”方法，定义了运行测试的默认方式。你也可以重写这些方法来对测试的编写和运行进行定制。

   - `ScalaTest` 提供了风格特质 `style trait`，这些特质扩展自 `Suite` 并重写了生命周期方法来支持不同的测试风格。
   - `ScalaTest` 还提供了混入特质 `mixin trait` ，这些特质扩展自 `Suite` 并重写了生命周期方法来支持特定的测试需要。
   - 可以组合 `style trait` 和 `mixin trait` 来定义测试类，也可以通过编写 `Suite` 实例来定义测试套件。
3. `ScalaTest` 已经被集成到常见的构建工具（如 `sbt,maven` ）和 `IDE` （如 `IDEA,Eclipse` ）中。
4. 可以通过 `ScalaTest` 的 `Runner` 应用程序直接运行 `Suite`，或者在 `scala` 解释器中简单调用其 `execute` 方法。

   ```
   // 在 scala 解释器中
   (new XXSuite).execute()
   ```

5. `ScalaTest` 的所有风格都被设计为鼓励编写专注的、带有描述性名字的测试。所有的风格都会生成规格说明书般的输出。
6. 示例：

   ```
   // 源码文件，位于包 model_a 中
   class C(val n:Double)
   {
       // 具体实现
   }

   // 测试文件
   import org.scalatest.FunSuite
   import model_a.C
   class CSuite extends FunSuite {
       test("C should have postive num"){
           val c = C(1.0)
           assert(c.n >=0)
       }
   }
   ```

   - `FunSuite` 中的 `Fun` 指的是函数 `function` 。
   - `test` 是定义在 `FunSuite` 中的一个方法，我们在 `CSuite` 的主构造方法中调用。

     调用时：

     - 圆括号中通过字符串给出测试的名称。

    - 通过花括号给出具体的测试代码。测试代码是一个以传名参数传入`test`的函数。`test`将这个函数登记下来，稍后执行。
7. 如果希望得到更详细的关于断言失败的信息，可以使用`ScalaTest`的`DiagrammedAssertions`，其错误消息会显式传入`assert`的表达式的一张示意图。
8. `ScalaTest`的`assert`方法并不在错误消息中区分实际结果和预期结果。如果你希望强调实际结果和预期结果，则使用`ScalaTest`的`assertResult`方法。如：

```
asseretResult(2){  // 预期结果：2
    c.n            // 实际结果
}
```

9. 如果要检查某个方法抛出某个预期的异常，则可以使用`ScalaTest`的`assertThrows`方法。

```
assertThrows[IllegalArgumentException]{
    C("Must be a double,but get a string")
}
```

    - 如果花括号中的代码未抛出异常，或者抛出了不同于预期的异常，则`assertThrows`将以`TestFailedException`异常终止。
    - 如果花括号中的代码以传入的异常类的实例异常终止（即：代码抛出了预期的异常），则`assertThrows`将正常返回。
    - 可以使用`intercept`，其机制与`assertThrows`相同，唯一区别在于：当代码抛出了预期的异常时，`intercept`将返回这个异常。注意：是返回，不是抛出。

```
val caught = intercept[ArithmeticException]{1/0}  // 返回了预期的异常
print(caught.getMessage)
```

2.2 BDD 风格

1. 行为驱动开发`BDD`测试风格的重点是：编写人类可读的关于代码预期行为的规格说明，同时给出验证代码具备指定行为的测试。

   `ScalaTest`包含了若干特质来支持这种风格的测试。
2. 在`FlatSpec`中，我们以规格子句`specifier clause`的形式编写测试。

```
import org.scalatest.FlatSpec
import org.scalatest.Matchers
import Element.elem   // 自定义类，即将测试它

class ElementSpec extends FlatSpec with Matchers{
    "An Element" should "has a given width" in {
        val ele = elem("name",2,3)
        ele.width should be (2)
    }
    it should "has a given height" in {
        val ele = elem("name",2,3)
        ele.height should be (3)
    }
    it should "throw an IAE if passed a negative width" in {
        an [IllegalArgumentException] should be thrownBy {
```

```
            ele("name",-2,3)
        }
    }
}
```

- 首先是以字符串表示的待测试的主体`subject`，如示例中的`"An Element"`。
- 然后是`should`（或者`must/can`）。
- 然后是一个描述该主体需要具备的行为的字符串。
- 然后是`in`。
- 最后是花括号包围的用于测试行为的代码。
- 在后续子句中，可以用`it`来指代最近给出的主体。

3. 当一个`FlatSpec`被执行时，它将每个规格子句作为`ScalaTest`测试运行。`FlatSpec`（以及`ScalaTest`的其它规则说明特质）在运行后将生成读起来像规格说明书的输出。
4. 通过混入`Matchers`特质，可以编写读上去更像自然语言的断言。`ScalaTest`在其`DSL`中提供了许多匹配器，并允许你用定制的失败消息定义新的`matcher`。
   - 上面示例中的匹配器包括`should be`和`an [...] should be thrownBy{...}`。
   - 如果相比`should`你更喜欢`must`，也可以选择`MustMatchers`。则匹配器可以为：

     ```
     result must be >= 0
     map must contain key 'c'
     ```

5. `BDD`的一个重要思想是：测试可以在软件功能制定者、软件功能实现者、软件功能测试者这三者之间架起一道沟通的桥梁。

   `ScalaTest`的`FeatureSpec`就是专门为此设计的。其设计目标是引导关于软件需求的对话：必须指明具体的功能`feature`、然后用场景`scenario`来描述这些功能。

```
import org.scalatest._

class TVSetSpec extends FeatureSpec with GivenWhenThen {
    features("TV power button") {
        scenario("User presses power button when TV is off"){
            Given(" a TV set that is switched off")
            When("the power button is pressed")
            Then("The TV should switch on")
            pending
        }
    }
}
```

   - `Given,When,Then`方法由`GivenWhenThen`特质提供，能帮助我们将对话聚焦在每个独立场景的具体细节上。
   - 最后的`pending`调用表明测试和实际行为都还没有实现：这仅仅是规格说明。

     一旦所有的测试和给定的行为都实现了，这些测试就会通过。此时我们说需求已经满足了。

2.3 specs2

1. `specs2`测试框架是`Eric Torreborre`用`Scala`编写的开源工具，也支持`TDD`风格的测试，但是语法不同。

```
import org.specs2._
import Element.elem   // 自定义类，即将测试它

object ElementSpec extends Specification{
    "An Element" should "has a given width" in {
        val ele = elem("name",2,3)
        ele.width must be_==(2)
    }
    it should "has a given height" in {
        val ele = elem("name",2,3)
        ele.height must be_==(2)
    }
    it should "throw an IAE if passed a negative width" in {
        ele('name',-2,3) must throwA[IllegalArgumentException]
    }
}
```

2. 和 `ScalaTest` 一样，`specs2` 也提供了匹配器 `DSL`。如上例中的 `must be_==` 和 `must throwA`。
3. 可以单独使用 `specs2`，不过它也被集成到 `ScalaTest` 和 `JUnit` 中，因此也可以用这些工具来运行 `specs2` 测试。

2.4 ScalaCheck

1. `Scala` 另一个有用的测试工具是 `ScalaCheck`，这是由 `Richard Nilsson` 编写的开源框架。

   `ScalaCheck` 让你能够指定被测试的代码必须满足的性质。对每个性质，`ScalaCheck` 都会生成数据并执行断言来检查代码是否满足该性质。

```
import org.scalatest.WordSpec
import org.scalatest.prop.PropertyChecks
import org.scalatest.MustMatchers._
import Element.elem   // 自定义类，即将测试它

class ElementSpec extends WordSpec with PropertyChecks {
    "An Element" must "has a given width" in {
        forAll {
            (w:Int) => whenever (w>0){ elem("name",w,3).width must euqual
(2)}
        }
    }

}
```

   - `PropertyChecks` 特质提供了若干 `forAll` 方法，让你可以将基于性质的测试跟传统的基于断言或基于匹配器的测试混合在一起。
   - `whenever` 的意思是：只要 `w>0` 为 `true`，则右边代码块中的表达式必须为 `true`。
   - 通过这一小段代码，`ScalaCheck` 就会帮我们生成数百条 `w` 可能的取值并对每一个进行测试，尝试找出不满足该性质的值。

     如果对每个值，该性质都满足，则测试通过。否则测试将以 `TestFailedException` 终止，该异常将会包含关于测试失败的信息。

2.5 执行测试

1. 每一个测试框架都提供了某种组织和运行测试的机制。
2. 在 `ScalaTest` 中，我们通过将 `Suite` 嵌套在别的 `Suite` 当中来组织大型的测试套件。当 `Suite` 被执行时，它将执行嵌套的 `Suite` 和其它测试。
   - 可以手动或者自动嵌套测试套件。
     - 手动方式：在你的 `Suite` 中重写 `nestedSuite` 方法，或者将你希望嵌套的 `Suite` 作为参数传递给 `Suites` 类的构造方法。
     - 自动方式：将包名提供给 `ScalaTest` 的 `Runner`，它会自动发现 `Suite` 套件，并将它们嵌套在一个根 `Suite` 里，并执行这个根 `Suite` 。
   - 可以通过命令行调用 `ScalaTest` 的 `Runner` 应用程序，也可以通过构建工具如 `sbt,maven,ant` 来调用。

     通过命令行调用 `Runner` 最简单的方式是通过 `org.scalatest.run` 。该应用程序预期一个完整的测试类名。

     如：

     ```
     scalac -cp scalatest.jar TVSetSpec.scala            # 编译测试类
     scala -cp scalatest.jar org.scalatest.run TVSetSpec # 执行测试
     ```

     - 通过 `cp` 参数将 `ScalaTest` 的 `JAR` 文件包含在类路径中。
     - `org.scalatest.run` 是完整的应用程序类名。 `Scala` 将会运行这个 `app` ，并传入剩下的命令行参数。
     - `TVSetSpec` 这个参数指定了要执行的套件。

三、注解

1. 注解是添加到程序源代码中的结构化信息。
   - 和注释一样，注解可以出现在程序中的任何位置，附加到任意变量、方法、表达式或其它程序元素上。
   - 和注释不同，注解有结构，因此更容易机器处理。

   这里我们主要介绍如何使用注解，而不是如何编写新的注解。编写新的注解不是我们关注的重点。因为使用注解要比编写新注解常用的多。
2. 一个典型的注解示例：

   ```
   @deprecated def f() = ...
   ```

   注解可以用于各种声明、定义上，包括 `val, var, def, class, object, trait, type` 。注解对于跟在它后面的整个声明或定义有效。

   注解也可以用于表达式，做法是在表达式后面写一个冒号 `:` 再写注解。从语法角度来看，注解像是被用在了类型上：

   ```
   (e: @unchecked) match{
     //...
   }
   ```

3. 目前为止所给的注解都是 `@` 加上注解类名的方式，不过注解也有更丰富的一般格式：

```
@annot(exp1,exp2,...)
```

其中：

- `annot` 是注解类名，所有的注解都必须包含。
- `exp` 部分是给注解的入参。对于 `@deprecated` 这样的注解而言，它们并不需要入参，因此通常可以省略圆括号。但是你也可以这样写：`@deprecated()` 。

  对于确实需要入参的注解，需要将入参写到圆括号中，例如 `@serial(1234)` 。

你提供给注解的入参的形式取决于特定的注解类。大多数注解处理器只允许你提供直接常量，如 `123` 或者 `"hello"`。 不过编译器本身（对于注解而言）是支持任意表达式的，只要它们能够通过类型检查。如：

```
@cool val normal = "hello"
@coolerThan(normal) val fonzy = "world"
```

`scala` 在内部将注解表示为仅仅是对某个注解类的构造方法的调用（想象下如果将 `@` 替换为 `new`）。这意味着编译器可以很自然地支持注解的带名字的参数和默认参数，因为 `scala` 已经支持方法和构造方法调用的带名字参数和默认参数。

不能直接把注解当做另一个注解的入参，因为注解并不是合法的表达式。在这种情况下，必须用 `new` 或者 `@` ：

```
import annotation._
class strategy(arg: Annotation) extends Annotation
class delayed extends Annotation
@strategy(new delayed) def f()=...
```

4. `scala` 包含了若干标准注解，它们是为一些非常常用的功能服务的，因此就被放在了语言规范中。不过还没有达到足够基础的程度，因此并没有自己的语法。
   - `deprecated`：可以将方法、类标记为 `deprecated`，这样任何人调用了这个方法或类都会得到一个 `deprecation` 警告。当经过一段时间之后，就可以假定使用方不再访问这个方法或类，因此可以安全地移除这个方法或类。

     可以简单地在方法/类之前写上 `@deprecated`。也可以提供一个字符串作为入参，此时这个字符串将在编译时随警告一起提示出来。通常这个字符串可以告诉大家解释该方法/类已经过时，应该如何升级。
   - `volatile`：可以将变量标记为 `volatile`，从而告诉编译器这个变量会被多个线程使用。这样的变量实现的效果使得读写更慢，但是从多个线程访问时的行为更可预期。

     事实上 `scala` 鼓励使用不可变的对象以及函数式编程，因此比较少地使用共享的可变状态。因此 `@volatile` 在 `scala` 中应用较少。
   - 序列化：序列化框架可以帮助我们将对象转化为字节流，或者将字节流还原为对象。当你希望将对象保存到磁盘或者将对象通过网络发送时很有帮助。

     `scala` 并没有自己的序列化框架，而是使用底层平台提供的框架。`scala` 能做的是提供三个可被不同框架使用的注解。针对 `Java` 平台的 `scala` 编译器会以 `java` 的方式来解释这些注解。
     - `serializable`：该注解用于表示某个类是否支持序列化。大多数类都是可序列化的，但是有些类不支持，如套接字或 `GUI` 窗口的句柄就不能被序列化。

默认情况下，系统不会认为类是可以序列化的，因此你需要给你认为可以序列化的类添加 `@serializable` 注解。

- `SerialVersionUID(1234)`：该注解用于表示版本可变的序列化。有些类，随着时间推移它可能发生变化（比如一个新的修改）。因此可以通过添加 `SerialVersionUID(1234)` 这样的注解来对某个类的当前版本带上一个序列号，其中 `1234` 可以替换为你想要的序列号。

  序列化框架将会把这个序列号保存在生成的字节流中。当稍后你从字节流中反序列化出对象时，框架可以检查对应类的当前版本是否和字节流中的版本一致。框架会自动拒绝载入老版本的对象。

- `transient`：该注解用于标记那些完全不应该被序列化的字段。

  如果你将某个字段标记为 `@transient`，那么就算包含该字段的对象被序列化了，序列化框架也不会保存该字段。当从字节流重新载入对象时，注解为 `@transient` 的这个字段将会被恢复成对应类型的默认值。

- `scala.reflect.BeanProperty`：该注解会为字段自动生成 `get` 和 `set` 方法。

  实际上 `scala` 代码通常不需要显式给出字段的 `get` 和 `set` 方法，因为 `scala` 混合了字段访问和方法调用的语法。不过有一些特定的框架可能希望你提供 `get` 或 `set` 方法。

  此时可以使用 `@scala.reflect.BeanProperty` 注解，该注解作用在字段上可以为字段自动生成 `get` 和 `set` 方法。如果该字段名字叫 `xyz`，则 `get` 方法自动命名为 `getXyz`，`set` 方法自动命名为 `setXyz`。

  注意：生成的 `get` 和 `set` 方法仅在编译后可用。因此，你不能在编写代码的时候调用这些 `get` 和 `set` 方法。但在实际应用中这不是问题，因为在 `scala` 中你可以直接访问这些字段。

- `tailrec`：该注解用于对尾递归方法进行尾递归优化。

  如果尾递归优化因为某些原因无法执行优化，那么你将会得到一个警告，并告诉你为什么无法优化。

- `unchecked`：该注解用在处理模式匹配的时候，告诉编译器不要担心 `match` 表达式可能看上去漏了某些 `case`。

- `native`：该注解告诉编译器某个方法的实现是由运行时而非 `scala` 代码提供的。编译器会在输出中开启合适的标记，将由开发者利用诸如 `java` 本地接口 `JNI` 的机制来提供实现。

  当使用 `@native` 注解时，必须提供方法体，不过这个方法体并不会被包含在输出当中。如，以下是声明一个由运行时提供的 `f` 方法：

```
@native
def f() = {}
```