第 11 章

Scala 的层级

你已经在前一章看过了类继承的细节，是时候退回一步整体看看 Scala 的类层级了。Scala 里，每个类都继承自通用的名为 Any 的超类。因为所有的类都是 Any 的子类，所以定义在 Any 中的方法就是“共同的”方法：它们可以被任何对象调用。Scala 还在层级的底端定义了一些有趣的类，如 Null 和 Nothing，扮演通用的子类。例如，如同 Any 是所有其他类的超类，Nothing 是所有其他类的子类。本章中，我们将带你一览 Scala 的类层级。

11.1 Scala 的类层级

图 11.1（见下一页）展示了 Scala 的类层级的大纲。层级的顶端是 Any 类，定义了下列的方法：

```
final def ==(that: Any): Boolean
final def !=(that: Any): Boolean
def equals(that: Any): Boolean
def hashCode: Int
def toString: String
```

因为每个类都继承自 Any，所以 Scala 程序里的每个对象都能用==、!=或 equals 比较，用 hashCode 做散列（hash），以及用 toString 格式化。Any 类里的等号和不等号方法、==和!=，被声明为 final，因此它们不能在子类里重写。实际上，==总是与 equals 相同，!=总是与 equals 相反。因此独立的类可以通过重写 equals 方法改变==或!=的意义。我们会在本章后面展示一个例子。

根类 Any 有两个子类：AnyVal 和 AnyRef。AnyVal 是 Scala 里每个内建值类的父类。有 9 个这样的值类：Byte、Short、Char、Int、Long、Float、Double、Boolean 和 Unit。其中的前 8 个都对应到 Java 的基本类型，它们的值在运行时表示成 Java 的基本类型的值。Scala 里这些类的实例都写成字面量。例如，42 是 Int 的实例，'x'是 Char 的实例，false 是 Boolean 的实例。你不能使用 new 创造这些类的实例。这一点由一个“小技巧（trick）”保证，值类都被定义为既是抽象的又是 final 的。因此如果你写了：

```
scala> new Int
```

就会得到：

```
<console>:5: error: class Int is abstract; cannot be
instantiated
       new Int
       ^
```

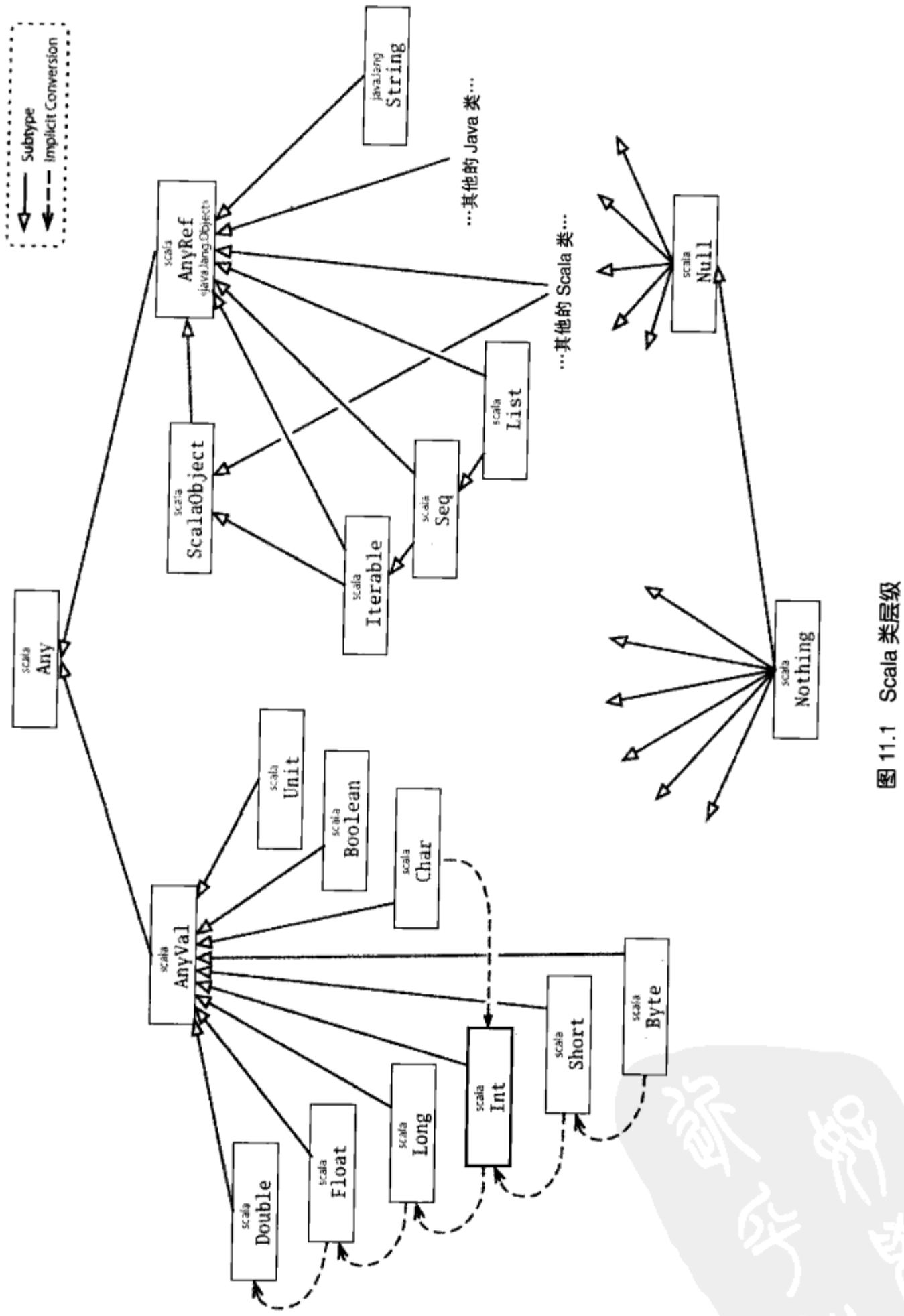

图 11.1 Scala 类层级

另一个值类，Unit，大约对应于 Java 的 void 类型；被用作不返回任何有趣结果的方法的结果类型。Unit 只有一个实例值，写成()，这在 7.2 节中讨论过。

正如第五章解释过的，值类以方法的形式支持通用的数学和布尔操作符。例如，Int 有名为+和*的方法，Boolean 有名为||和&&的方法。值类也从类 Any 继承所有的方法。你可以在解释器里测试：

```
scala> 42.toString
res1: java.lang.String = 42

scala> 42.hashCode
res2: Int = 42

scala> 42 equals 42
res3: Boolean = true
```


注意，值类的空间是平坦的；所有的值类都是 scala.AnyVal 的子类型，但是它们不是其他类的子类。但是不同的值类类型之间可以隐式地互相转换。例如，需要的时候，类 scala.Int 的实例可以自动放宽（通过隐式转换）到类 scala.Long 的实例。

正如 5.9 节中提到过的，隐式转换还用来为值类型添加更多的功能。例如，类型 Int 支持以下所有的操作：

```
scala> 42 max 43
res4: Int = 43

scala> 42 min 43
res5: Int = 42

scala> 1 until 5
res6: Range = Range(1, 2, 3, 4)

scala> 1 to 5
res7: Range.Inclusive = Range(1, 2, 3, 4, 5)

scala> 3.abs
res8: Int = 3

scala> (-3).abs
res9: Int = 3
```

这里解释其工作原理：方法 min、max、until、to 和 abs 都定义在类 scala.runtime.RichInt 里，并且有一个从类 Int 到 RichInt 的隐式转换。当你在 Int 上调用的方法没有定义在 Int 中，但定义在 RichInt 中时，就应用这个转换。类似的“支持类”和隐式转换也存在于其他的值类。我们将在第 21 章具体讨论隐式转换。

类 Any 的另一个子类是类 AnyRef。这个是 Scala 里所有引用类（reference class）的基类。正如前面提到的，在 Java 平台上 AnyRef 实际就是类 java.lang.Object 的别名。因此 Java 里写的类和 Scala 里写的都继承自 AnyRef[1]。你可以认为 java.lang.Object 是 Java 平台上实现 AnyRef 的方式。因此，尽管在 Java 平台上的 Scala 程序里 Object 和 AnyRef 的使用是可交换的，推荐的风格是在任何地方都只使用 AnyRef。

[1]注：使用 AnyRef 别名代替 java.lang.Object 名称的理由是 Scala 被设计成可以同时工作在 Java 和.Net 平台上。在.NET 平台上，AnyRef 是 System.Object 的别名。

Scala 类与 Java 类的不同在于它们还继承自一个名为 ScalaObject 的特别的记号特质[2]。是想要通过 ScalaObject 包含的 Scala 编译器定义和实现的方法让 Scala 程序的执行更高效。到现在为止，Scala object 只包含了一个方法，名为$tag，在内部使用以加速模式匹配。

11.2 原始类型是如何实现的

这些都是怎么实现的？实际上，Scala 以与 Java 同样的方式存储整数：把它当作 32 位的字。这对在 JVM 上的效率及与 Java 库的互操作性方面来说都很重要。标准的操作如加法或乘法都被实现为基本操作。然而，当整数需要被当作（Java）对象看待的时候，Scala 使用了“备份”类 java.lang.Integer。如在整数上调用 toString 方法或者把整数赋值给 Any 类型的变量时，就会这么做。需要的时候，Int 类型的整数能被透明转换为 java.lang.Integer 类型的“装箱整数（boxed integer）”。

所有这些听上去都类似 Java5 里的自动装箱，当然它们的确很像。不过有一个关键的差异，就是 Scala 里的装箱比 Java 里的更少见。尝试下面的 Java 代码：

```
// Java 代码
boolean isEqual(int x,int y) {
  return x == y;
}
System.out.println(isEqual(421,421));
```

你当然会得到 true。现在，把 isEqual 的参数类型变为 java.lang.Integer（或 Object，结果都一样）：

```
// Java 代码
boolean isEqual(Integer x,Integer y){
  return x==y;
}
System.out.println(isEqual(421,421));
```

209

你得到了 false！原因是数字 421 被装箱了两次，因此参数 x 和 y 是两个不同的对象。

因为在引用类型上==表示引用相等，而 Integer 是引用类型，所以结果是 false。这说明了 Java 不是纯粹面向对象语言的一个方面。我们能清楚观察到基本类型和引用类型之间的差别。

现在在 Scala 里尝试同样的实验：

```
scala> def isEqual(x:Int, y:Int) = x == y
isEqual:(Int,Int)Boolean

scala> isEqual(421,421)
res10:Boolean = true

scala> def isEqual(x:Any, y:Any) = x == y
isEqual:(Any,Any)Boolean

scala> isEqual(421,421)
res11:Boolean = true
```

[2]译注：special marker trait。

实际上 Scala 里的相等操作==被设计为对类型表达透明。对值类型来说，就是自然的（数学或布尔）相等。对于引用类型，==被视为继承自 Object 的 equals 方法的别名。这个方法被初始地定义为引用相等，但被许多子类重写以实现它们自然理念上的相等性。这也意味着在 Scala 里你永远也不会落入 Java 知名的关于字符串比较的陷阱。在 Scala 中，字符串比较以其应有的方式工作：

```
scala> val x = "abcd".substring(2)
x:java.lang.String = cd

scala> val y = "abcd".substring(2)
y:java.lang.String = cd

scala> x == y
res12:Boolean = true
```

Java 里，x 与 y 的比较结果将是 false。程序员在这种情况应该用 equals，不过它容易被忘记。

然而，有些情况你需要使用引用相等代替用户定义的相等。例如，某些时候效率是首要因素，你想要把某些类散列合并（hash cons）然后通过引用相等比较它们的实例[3]。为解决这类问题，AnyRef 类定义了附加的 eq 方法，它不能被重写并且实现为引用相等（也就是说，它表现得就像 Java 里对于引用类型的==那样）。同样也有一个 eq 的反义词，被称为 ne。例如：

```
scala> val x = new String("abc")
x:java.lang.String = abc

scala> val y = new String("abc")
y:java.lang.String = abc

scala> x == y
res13:Boolean = true

scala> x eq y
res14:Boolean = false

scala> x ne y
res15:Boolean = true
```

Scala 的相等性会在第 28 章中讨论。

11.3 底层类型

在图 11.1 类型层级的底部你看到了两个类 scala.Null 和 Scala.Nothing。它们是用统一的方式处理 Scala 面向对象类型系统的某些“边界情况”的特殊类型。

Null 类是 null 引用对象的类型，它是每个引用类（就是说，每个继承自 AnyRef 的类）的子类。Null 不兼容值类型。例如，你不能把 null 值赋给整数变量：

```
scala> val i: Int = null
<console>:4: error:type mismatch;
```

[3]注：类实例的散列合并是指把创建的所有实例缓存在弱集合中。然后，一旦需要类的新实例，首先检查缓存。如果缓存中已经有一个元素与要创建的相等，就可以重用存在的实例。这样安排的结果是，任何 equals() 方法判定相等的两个实例同样在引用相等性上一致。

```
found    : Null(null)
required: Int
```

Nothing 类型在 Scala 的类层级的最底端；它是任何其他类型的子类型。然而，根本没有这个类型的任何值。要一个没有值的类型有什么意思呢？7.4 节中讨论过，Nothing 的一个用处是它标明了不正常的终止。例如 Scala 的标准库中的 Predef 对象有一个 error 方法，如下定义：

```
def error(message: String): Nothing =
  throw new RuntimeException(message)
```

error 的返回类型是 Nothing，告诉用户方法不是正常返回的（代之以抛出异常）。因为 Nothing 是任何其他类型的子类，所以你可以非常灵活地使用像 error 这样的方法。例如：

```
def divide(x: Int, y: Int): Int =
  if(y != 0) x / y
  else error("can't divide by zero")
```

“那么”（then）状态分支，x / y，类型为 Int，而“否则”（else）分支，调用了 error，类型为 Nothing。因为 Nothing 是 Int 的子类型，所以整个状态语句的类型是 Int，正如需要的那样。

11.4 小结

本章中我们展示了在 Scala 类层级的顶端和底端的类。现在你对 Scala 里类继承的理解打下了良好的基础，做好了理解混入组合的准备。下一章，你会学到关于特质的内容。