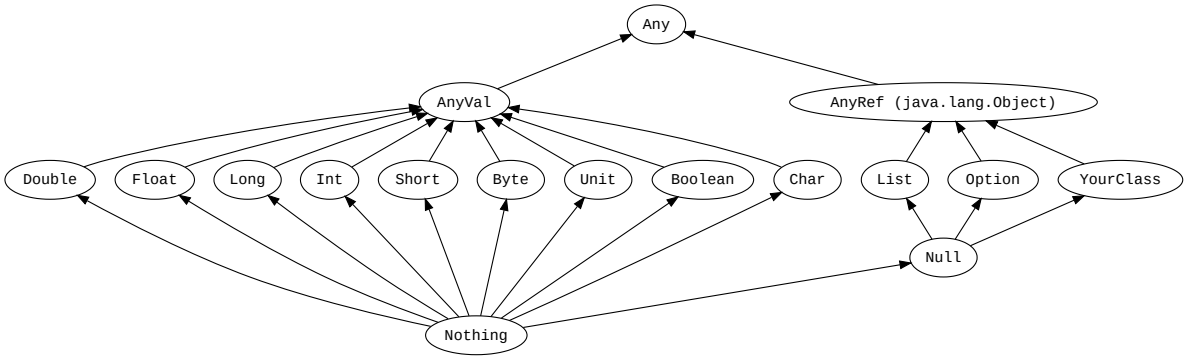
语法

统一类型



Scala类型层次结构

Any是所有类型的超类型，也称为顶级类 型。它定义了一些通用的方法如 `equals` 、 `hashCode` 和 `toString` 。 `Any` 有两个直接子类： `AnyVal` 和 `AnyRef` 。

`AnyVal` 代表值类型。有9个预定义的非空的值类型分别是： `Double` 、 `Float` 、 `Long` 、 `Int` 、 `Short` 、 `Byte` 、 `Char` 、 `Unit` 和 `Boolean` 。 `Unit` 是不带任何意义的值类型，它仅有一个实例可以像这样声明： `()` 。所有的函数必须有返回，所以说有时候 `Unit` 也是有用的返回类型。

`AnyRef` 代表引用类型。所有非值类型都被定义为引用类型。在Scala中，每个用户自定义的类型都是 `AnyRef` 的子类型。如果Scala被应用在Java的运行环境中， **`AnyRef` 相当于 `java.lang.Object`** 。

这里有一个例子，说明了字符串、整型、布尔值和函数都是对象，这一点和其他对象一样：

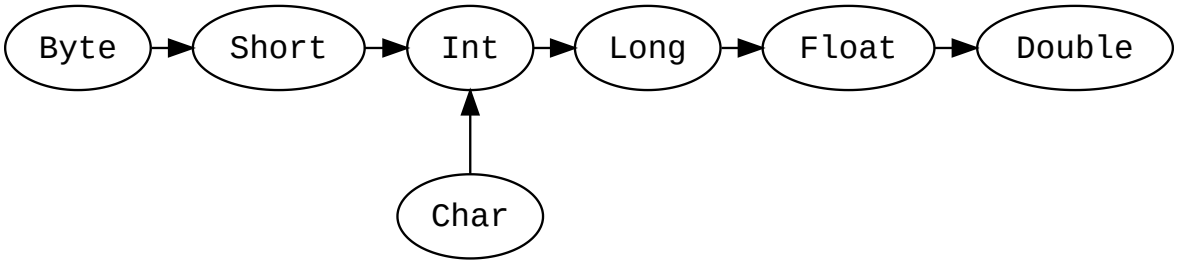
```
object Main extends App {
    val list: List[Any] = List(
        "a string",
        732, // 整数
        'c', // 字符
        true, // 布尔值
        () => "an anonymous function returning a string"
    )
    list.foreach(e => println(e))
}
```

输出：

```
a string
732
c
true
com.lymboy.scala.day01.Main$$$Lambda$2/1910163204@256216b3
```

类型转换

值类型可以按照下面的方向进行转换：



例如：

```
val x: Long = 987654321
val y: Float = x  // 9.8765434E8 (note that some precision is lost in this case)

val face: Char = '☺'
val number: Int = face  // 9786
```

转换是单向，下面这样写将不会通过编译。

```
val x: Long = 987654321
val y: Float = x  // 9.8765434E8
val z: Long = y  // Does not conform
```

你可以将一个类型转换为子类型，这点将在后面的文章介绍。

Nothing和Null

`Nothing`是所有类型的子类型，也称为底部类型。没有一个值是`Nothing`类型的。它的用途之一是给出非正常终止的信号，如抛出异常、程序退出或者一个无限循环（可以理解为它是一个不对值进行定义的表达式的类型，或者是一个不能正常返回的方法）。

`Null`是所有引用类型的子类型（即`AnyRef`的任意子类型）。它有一个单例值由关键字`null`所定义。`Null`主要是使得Scala满足和其他JVM语言的互操作性，但是几乎不应该在Scala代码中使用。我们将在后面的章节中介绍`null`的替代方案。

类

Scala中的类是用于创建对象的蓝图，其中包含了方法、常量、变量、类型、对象、特质、类，这些统称为成员。类型、对象和特质将在后面的文章中介绍。

类定义

一个最简的类的定义就是关键字`class`+标识符，类名首字母应大写。

```
class User
val user1 = new User
```

关键字`new`被用于创建类的实例。`User`由于没有定义任何构造器，因而只有一个不带任何参数的默认构造器。然而，你通常需要一个构造器和类体。下面是类定义的一个例子：

```
class Point(var x: Int, var y: Int) {
  def move(dx: Int, dy: Int): Unit = {
    x = x + dx
    y = y + dy
  }
  override def toString: String = s"($x, $y)"
}

val point1 = new Point(2, 3)
point1.x  // 2
println(point1)  // prints (2, 3)
```

`Point`类有4个成员：变量`x`和`y`，方法`move`和`toString`。与许多其他语言不同，主构造方法在类的签名中`(var x: Int, var y: Int)`。`move`方法带有2个参数，返回无任何意义的`Unit`类型值`()`。这一点与Java这类语言中的`void`相当。另外，`toString`方法不带任何参数但是返回一个`String`值。因为`toString`覆盖了AnyRef中的`toString`方法，所以用了`override`关键字标记。

构造器

构造器可以通过提供一个默认值来拥有可选参数：

```
object ConstructDemo extends App {
    class Point(var x: Int = 0, var y: Int = 0)

    val origin = new Point  // x, y都是1
    val point1 = new Point(1)
    println(point1.x)  // prints 1
}
```

在这个版本的`Point`类中，`x`和`y`拥有默认值`0`所以没有必传参数。然而，因为**构造器是从左往右读取参数**，所以如果仅仅要传个`y`的值，你需要**带名传参**。

```
class Point(var x: Int = 0, var y: Int = 0)
val point2 = new Point(y=2)
println(point2.y)  // prints 2
```

这样的做法在实践中有利于使得表达明确无误。

私有成员和Getter/Setter语法

成员默认是公有（`public`）的。使用`private`访问修饰符可以在类外部隐藏它们。

```
class Point {
  private var _x = 0
  private var _y = 0
  private val bound = 100

  def x = _x
  def x_= (newValue: Int): Unit = {
    if (newValue < bound) _x = newValue else printWarning
  }

  def y = _y
  def y_= (newValue: Int): Unit = {
    if (newValue < bound) _y = newValue else printWarning
  }

```

```
16      private def printWarning = println("WARNING: Out of bounds")
17    }
18
19    val point1 = new Point
20    point1.x = 99
21    point1.y = 101 // prints the warning
```

在这个版本的`Point`类中，数据存在私有变量`_x`和`_y`中。`def x`和`def y`方法用于访问私有数据。`def x_=`和`def y_=`是为了验证和给`_x`和`_y`赋值。注意下对于setter方法的特殊语法：这个方法在getter方法的后面加上`_=`，后面跟着参数。

主构造方法中带有`val`和`var`的参数是公有的。然而由于`val`是不可变的，所以不能像下面这样去使用。

```
1   class Point(val x: Int, val y: Int)
2   val point = new Point(1, 2)
3   point.x = 3  // <-- does not compile
```

不带`val`或`var`的参数是私有的，仅在类中可见。

```
1   class Point(x: Int, y: Int)
2   val point = new Point(1, 2)
3   point.x  // <-- does not compile
```

默认参数值

Scala具备给参数提供默认值的能力，这样调用者就可以忽略这些具有默认值的参数。

```
1   def log(message: String, level: String = "INFO") = println(s"$level: $message")
2
3   log("System starting")  // prints INFO: System starting
4   log("User not found", "WARNING")  // prints WARNING: User not found
```

上面的参数level有默认值，所以是可选的。最后一行中传入的参数`"WARNING"`重写了默认值`"INFO"`。在Java中，我们可以通过带有可选参数的重载方法达到同样的效果。不过，只要调用方忽略了一个参数，其他参数就必须要带名传入。

```
1   class Point(val x: Double = 0, val y: Double = 0)
2   val point1 = new Point(y = 1)
```

这里必须带名传入`y = 1`。

注意从Java代码中调用时，Scala中的默认参数则是必填的（非可选），如：

```
1   // Point.scala
2   class Point(val x: Double = 0, val y: Double = 0)
```

```
1   // Main.java
2   public class Main {
3       public static void main(String[] args) {
4           Point point = new Point(1);  // 编译不通过
5       }
6   }
```

命名参数

当调用方法时，实际参数可以通过其对应的形式参数的名称来标记：

```
def printName(first: String, last: String): Unit = {
  println(first + " " + last)
}

printName("John", "Smith")  // Prints "John Smith"
printName(first = "John", last = "Smith")  // Prints "John Smith"
printName(last = "Smith", first = "John")  // Prints "John Smith"
```

注意使用命名参数时，**顺序是可以重新排列的**。 但是，如果某些参数被命名了，而其他参数没有，则**未命名的参数要按照其方法签名中的参数顺序放在前面**。

```
printName(last = "Smith", "john") // 错误：参数在命名参数后
```

注意调用 Java 方法时不能使用命名参数。

Traits

Traits用于在类 Class之间共享程序接口 Interface和字段 Fields。 它们类似于Java 8的接口。 类和对象 Objects可以扩展Traits，但是Traits不能被实例化，因此Traits没有参数。

定义一个Traits

最简化的Traits就是关键字trait+标识符：

```
trait HairColor
```

Traits作为泛型类型和抽象方法非常有用。

```
trait Iterator[A] {
  def hasNext: Boolean
  def next(): A
}
```

扩展 `trait Iterator [A]` 需要一个类型 `A` 和实现方法 `hasNext` 和 `next` 。

使用Traits

使用 `extends` 关键字来扩展Traits。然后使用 `override` 关键字来实现trait里面的任何抽象成员：

```
 trait Iterator[A] {
   def hasNext: Boolean
   def next(): A
 }

 class IntIterator(to: Int) extends Iterator[Int] {
   private var current = 0
   override def hasNext: Boolean = current < to
   override def next(): Int =  {
     if (hasNext) {
       val t = current
       current += 1
```

```
        t
      } else 0
    }
  }

  val iterator = new IntIterator(10)
  iterator.next()  // returns 0
  iterator.next()  // returns 1
```

这个类 `IntIterator` 将参数 `to` 作为上限。它扩展了 `Iterator [Int]`，这意味着方法 `next` 必须返回一个Int。

子类型

凡是需要Traits的地方，都可以由该Traits的子类型来替换。

```
trait Pet {
    val name: String
    override def toString: String = s"(${name})"
}

class Dog(val name: String) extends Pet
class Cat(val name: String) extends Pet

var dog = new Dog("旺财")
var cat = new Cat("加菲")

val animals = ArrayBuffer.empty[Pet]
animals.addOne(dog)
animals.addOne(cat)

animals.foreach(println)
```

在这里 `trait Pet` 有一个抽象字段 `name` ， `name` 由Cat和Dog的构造函数中实现。最后一行，我们能调用 `pet.name` 的前提是**它必须在Traits Pet的子类型中得到了实现**。

元组

在 Scala 中，元组是一个可以容纳不同类型元素的类。 **元组是不可变的**。

当我们需要从函数返回多个值时，元组会派上用场。

元组可以创建如下：

```
val ingredient = ("Sugar" , 25):Tuple2[String, Int]
```

这将创建一个包含一个 String 元素和一个 Int 元素的元组。

Scala 中的元组包含一系列类：Tuple2，Tuple3等，直到 Tuple22。 因此，当我们创建一个包含 n 个元素（n 位于 2 和 22 之间）的元组时，Scala 基本上就是从上述的一组类中实例化 一个相对应的类，使用组成元素的类型进行参数化。 上例中， `ingredient` 的类型为 `Tuple2[String, Int]` 。

访问元素

使用下划线语法来访问元组中的元素。 'tuple._n' 取出了第 n 个元素（假设有足够多元素）。

```
println(ingredient._1) // Sugar
println(ingredient._2) // 25
```

解构元组数据

Scala 元组也支持解构。

```
val (name, quantity) = ingredient
println(name) // Sugar
println(quantity) // 25
```

元组解构也可用于模式匹配。

```
 val planetDistanceFromSun = List(("Mercury", 57.9), ("Venus", 108.2), ("Earth", 149.6 ), ("Mars", 227.9), ("Jupiter", 778.3))

 planetDistanceFromSun.foreach{ tuple => {
   tuple match {
       case ("Mercury", distance) => println(s"Mercury is $distance millions km far from Sun")
       case p if(p._1 == "Venus") => println(s"Venus is ${p._2} millions km far from Sun")
       case p if(p._1 == "Earth") => println(s"Blue planet is ${p._2} millions km far from Sun")
       case _ => println("Too far....")
     }
   }
 }
```

或者，在 'for' 表达式中。

```
val numPairs = List((2, 5), (3, -7), (20, 56))
for ((a, b) <- numPairs) {
    println(a * b)
}
```

类型 `Unit` 的值 `()` 在概念上与类型 `Tuple0` 的值 `()` 相同。 `Tuple0` 只能有一个值，因为它没有元素。

用户有时可能在元组和 case 类之间难以选择。 通常，如果元素具有更多含义，则首选 case 类。

组合类

当某个Traits被用于组合类时，被称为Mixins（可以理解为Java的`implement`）。

```
abstract class A {
  val message: String
}
class B extends A {
  val message = "I'm an instance of class B"
}
trait C extends A {
```

```
    def loudMessage = message.toUpperCase()
  }
  class D extends B with C

  val d = new D
  println(d.message)  // I'm an instance of class B
  println(d.loudMessage)  // I'M AN INSTANCE OF CLASS B
```

类 `D` 有一个父类 `B` 和一个 `mixin` C。一个类只能有一个父类但是可以有多个 `mixin`（分别使用关键字 `extend` 和 `with`）。`mixin` 和某个父类可能有相同的父类。

现在，让我们看一个更有趣的例子，其中使用了抽象类：

```
abstract class AbsIterator {
  type T
  def hasNext: Boolean
  def next(): T
}
```

该类中有一个抽象的类型 `T` 和标准的迭代器方法。

接下来，我们将实现一个具体的类（所有的抽象成员 `T`、`hasNext` 和 `next` 都会被实现）：

```
 class StringIterator(s: String) extends AbsIterator {
   type T = Char
   private var i = 0
   def hasNext = i < s.length
   def next() = {
     val ch = s charAt i
     i += 1
     ch
   }
 }
```

`StringIterator` 带有一个 `String` 类型参数的构造器，可用于对字符串进行迭代。（例如查看一个字符串是否包含某个字符）：

现在我们创建一个 `Traits`，也继承于 `AbsIterator`。

```
trait RichIterator extends AbsIterator {
  def foreach(f: T => Unit): Unit = while (hasNext) f(next())
}
```

该 `Traits` 实现了 `foreach` 方法——只要还有元素可以迭代（`while (hasNext)`），就会一直对下个元素(`next()`)调用传入的函数 `f: T => Unit`。因为 `RichIterator` 是个 `Traits`，可以不必实现 `AbsIterator` 中的抽象成员。

下面我们要把 `StringIterator` 和 `RichIterator` 中的功能组合成一个类。

```
object StringIteratorTest extends App {
  class RichStringIter extends StringIterator("Scala") with RichIterator
  val richStringIter = new RichStringIter
  richStringIter foreach println
}
```

新的类 `RichStringIter` 有一个父类 `StringIterator` 和一个 `mixin` `RichIterator`。如果是单一继承，我们将不会达到这样的灵活性。

高阶函数

高阶函数是指使用其他函数作为参数、或者返回一个函数作为结果的函数。在Scala中函数是“一等公民”，所以允许定义高阶函数。这里的术语可能有点让人困惑，我们约定，使用函数值作为参数，或者返回值为函数值的“函数”和“方法”，均称之为“高阶函数”。

最常见的一个例子是Scala集合类（collections）的高阶函数`map`

```
1   val salaries = Seq(20000, 70000, 40000)
2   val doubleSalary = (x: Int) => x * 2
3   val newSalaries = salaries.map(doubleSalary) // List(40000, 140000, 80000)
```

函数`doubleSalary`有一个整型参数`x`，返回`x * 2`。一般来说，在`=>`左边的元组是函数的参数列表，而右边表达式的值则为函数的返回值。在第3行，函数`doubleSalary`被应用在列表`salaries`中的每一个元素。

为了简化压缩代码，我们可以使用匿名函数，直接作为参数传递给`map`：

```
1   val salaries = Seq(20000, 70000, 40000)
2   val newSalaries = salaries.map(x => x * 2) // List(40000, 140000, 80000)
```

注意在上述示例中`x`没有被显式声明为Int类型，这是因为编译器能够根据map函数期望的类型推断出`x`的类型。对于上述代码，一种更惯用的写法为：

```
1   val salaries = Seq(20000, 70000, 40000)
2   val newSalaries = salaries.map(_ * 2)
```

既然Scala编译器已经知道了参数的类型（一个单独的Int），你可以只给出函数的右半部分，不过需要使用`_`代替参数名（在上一个例子中是`x`）

强制转换方法为函数

你同样可以传入一个对象方法作为高阶函数的参数，这是因为Scala编译器会将方法强制转换为一个函数。

```
1   case class WeeklyWeatherForecast(temperatures: Seq[Double]) {
2     private def convertCtoF(temp: Double) = temp * 1.8 + 32
3     def forecastInFahrenheit: Seq[Double] = temperatures.map(convertCtoF) // <--
    passing the method convertCtoF
4   }
```

在这个例子中，方法`convertCtoF`被传入`forecastInFahrenheit`。这是可以的，因为编译器强制将方法`convertCtoF`转成了函数`x => convertCtoF(x)`（注：`x`是编译器生成的变量名，保证在其作用域是唯一的）。

接收函数作为参数的函数

使用高阶函数的一个原因是减少冗余的代码。比方说需要写几个方法以通过不同方式来提升员工工资，若不使用高阶函数，代码可能像这样：

```
object SalaryRaiser {
  def smallPromotion(salaries: List[Double]): List[Double] =
    salaries.map(salary => salary * 1.1)

  def greatPromotion(salaries: List[Double]): List[Double] =
    salaries.map(salary => salary * math.log(salary))

  def hugePromotion(salaries: List[Double]): List[Double] =
    salaries.map(salary => salary * salary)
}
```

注意这三个方法的差异仅仅是提升的比例不同，为了简化代码，其实可以把重复的代码提到一个高阶函数中：

```
object SalaryRaiser {

  private def promotion(salaries: List[Double], promotionFunction: Double => Double): List[Double] =
    salaries.map(promotionFunction)

  def smallPromotion(salaries: List[Double]): List[Double] =
    promotion(salaries, salary => salary * 1.1)

  def bigPromotion(salaries: List[Double]): List[Double] =
    promotion(salaries, salary => salary * math.log(salary))

  def hugePromotion(salaries: List[Double]): List[Double] =
    promotion(salaries, salary => salary * salary)
}
```

新的方法`promotion`有两个参数，薪资列表和一个类型为`Double => Double`的函数（参数和返回值类型均为Double），返回薪资提升的结果。

返回函数的函数

有一些情况你希望生成一个函数， 比如：

```
def urlBuilder(ssl: Boolean, domainName: String): (String, String) => String = {
    val schema = if (ssl) "https://" else "http://"
    (endpoint: String, query: String) => s"$schema$domainName/$endpoint?$query"
}
val domainName = "www.example.com"
def getURL = urlBuilder(ssl = true, domainName)
val endpoint = "users"
val query = "id=1"
val url = getURL(endpoint, query) // "https://www.example.com/users?id=1": String
```

注意urlBuilder的返回类型是`(String, String) => String`，这意味着返回的匿名函数有两个String参数，返回一个String。在这个例子中，返回的匿名函数是`(endpoint: String, query: String) => s"https://www.example.com/$endpoint?$query"`。

嵌套方法

在Scala中可以**嵌套定义方法**。例如以下对象提供了一个`factorial`方法来计算给定数值的阶乘：

```
def factorial(x: Int): Int = {
    def fact(x: Int, accumulator: Int): Int = {
        if (x <= 1) accumulator
        else fact(x - 1, x * accumulator)
    }
    fact(x, 1)
}

println("Factorial of 2: " + factorial(2))
println("Factorial of 3: " + factorial(3))
```

程序的输出为:

```
Factorial of 2: 2
Factorial of 3: 6
```

多参数列表（柯里化）

方法可以定义多个参数列表，当使用较少的参数列表调用多参数列表的方法时，会产生一个新的函数，该函数接收剩余的参数列表作为其参数。这被称为柯里化。

下面是一个例子，在Scala集合 `trait TraversableOnce` 定义了 `foldLeft`

```
def foldLeft[B](z: B)(op: (B, A) => B): B
```

`foldLeft` 从左到右，以此将一个二元运算 `op` 应用到初始值 `z` 和该迭代器（traversable)的所有元素上。以下是该函数的一个用例:

从初值0开始, 这里 `foldLeft` 将函数 `(m, n) => m + n` 依次应用到列表中的每一个元素和之前累积的值上。

```
val numbers = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
val res = numbers.foldLeft(0)((m, n) => m + n)
print(res) // 55
```

多参数列表有更复杂的调用语法，因此应该谨慎使用，建议的使用场景包括:

单一的函数参数

在某些情况下存在单一的函数参数时，例如上述例子 `foldLeft` 中的 `op`，多参数列表可以使得传递匿名函数作为参数的语法更为简洁。如果不使用多参数列表，代码可能像这样:

```
numbers.foldLeft(0, {(m: Int, n: Int) => m + n})
```

注意使用多参数列表时，我们还可以利用Scala的类型推断来让代码更加简洁（如下所示），而如果没有多参数列表，这是不可能的。

```
numbers.foldLeft(0)(_ + _)
```

像上述语句这样，我们可以给定多参数列表的一部分参数列表（如上述的 `z`）来形成一个新的函数（partially applied function），达到复用的目的，如下所示:

```
val numbers = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
val numberFunc = numbers.foldLeft(List[Int]())_

val squares = numberFunc((xs, x) => xs:+ x*x)
print(squares.toString()) // List(1, 4, 9, 16, 25, 36, 49, 64, 81, 100)

val cubes = numberFunc((xs, x) => xs:+ x*x*x)
print(cubes.toString())  // List(1, 8, 27, 64, 125, 216, 343, 512, 729, 1000)
```

最后，`foldLeft` 和 `foldRight` 可以按以下任意一种形式使用，

```
val numbers = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

numbers.foldLeft(0)((sum, item) => sum + item) // Generic Form
numbers.foldRight(0)((sum, item) => sum + item) // Generic Form

numbers.foldLeft(0)(_+_) // Curried Form
numbers.foldRight(0)(_+_) // Curried Form
```

隐式（IMPLICIT）参数

如果要指定参数列表中的某些参数为隐式（implicit），应该使用多参数列表。例如：

```
def execute(arg: Int)(implicit ec: ExecutionContext) = ???
```

CASE CLASSES

案例类（Case classes）和普通类差不多，只有几点关键差别，接下来的介绍将会涵盖这些差别。案例类非常适合用于不可变的数据。下一节将会介绍他们在模式匹配中的应用。

`case` 类似于Java中的 `final` 类，即不可变类。

定义一个case类

一个最简单的 `case` 类定义由关键字 `case class`，类名，参数列表（可为空）组成：

```
case class Book(isbn: String)

val frankenstein = Book("978-0486282114")
```

注意在实例化 `case` 类 `Book` 时，并没有使用关键字 `new`，这是因为 `case` 类有一个默认的 `apply` 方法来负责对象的创建。

当你创建包含参数的 `case` 类时，这些参数是公开（public）的 `val`

```
case class Message(sender: String, recipient: String, body: String)
val message1 = Message("guillaume@quebec.ca", "jorge@catalonia.es", "Ça va ?")

println(message1.sender)  // prints guillaume@quebec.ca
message1.sender = "travis@washington.us"  // this line does not compile
```

你不能给 `message1.sender` 重新赋值，因为它是一个 `val`（不可变）。在 `case` 类中使用 `var` 也是可以的，但并不推荐这样。

比较

`case` 类在比较的时候是按值比较而非按引用比较：

```
case class Message(sender: String, recipient: String, body: String)

val message2 = Message("jorge@catalonia.es", "guillaume@quebec.ca", "Com va?")
val message3 = Message("jorge@catalonia.es", "guillaume@quebec.ca", "Com va?")
val messagesAreTheSame = message2 == message3  // true
```

尽管 `message2` 和 `message3` 引用不同的对象，但是他们的值是相等的，所以 `message2 == message3` 为 `true`。

拷贝

你可以通过 `copy` 方法创建一个 `case` 类实例的浅拷贝，同时可以指定构造参数来做一些改变。

```
case class Message(sender: String, recipient: String, body: String)
val message4 = Message("julien@bretagne.fr", "travis@washington.us", "Me zo o komz gant ma amezeg")
val message5 = message4.copy(sender = message4.recipient, recipient = "claire@bourgogne.fr")
message5.sender  // travis@washington.us
message5.recipient // claire@bourgogne.fr
message5.body  // "Me zo o komz gant ma amezeg"
```

上述代码指定 `message4` 的 `recipient` 作为 `message5` 的 `sender`，指定 `message5` 的 `recipient` 为”claire@bourgogne.fr”，而 `message4` 的 `body` 则是直接拷贝作为 `message5` 的 `body` 了。

模式匹配

模式匹配是检查某个值（value）是否匹配某一个模式的机制，一个成功的匹配同时会将匹配值解构为其组成部分。它是Java中的 `switch` 语句的升级版，同样可以用于替代一系列的 if/else 语句。

语法

一个模式匹配语句包括一个待匹配的值，`match` 关键字，以及至少一个 `case` 语句。

```
import scala.util.Random

val x: Int = Random.nextInt(10)

x match {
  case 0 => "zero"
  case 1 => "one"
  case 2 => "two"
  case _ => "other"
}
```

上述代码中的 `val x` 是一个0到10之间的随机整数，将它放在 `match` 运算符的左侧对其进行模式匹配，`match` 的右侧是包含4条 `case` 的表达式，其中最后一个 `case _` 表示匹配其余所有情况，在这里就是其他可能的整型值。

`match` 表达式具有一个结果值

```
def matchTest(x: Int): String = x match {
  case 1 => "one"
  case 2 => "two"
  case _ => "other"
}
matchTest(3)  // other
matchTest(1)  // one
```

这个`match`表达式是String类型的，因为所有的情况（case）均返回String，所以`matchTest`函数的返回值是String类型。

case classes的匹配

`case`类非常适合用于模式匹配。

```
abstract class Notification
case class Email(sender: String, title: String, body: String) extends Notification
case class SMS(caller: String, message: String) extends Notification
case class VoiceRecording(contactName: String, link: String) extends Notification
```

`Notification`是一个虚基类，它有三个具体的子类`Email`，`SMS`和`VoiceRecording`，我们可以在这些案例类(Case Class)上像这样使用模式匹配：

```
def showNotification(notification: Notification): String = {
  notification match {
    case Email(sender, title, _) =>
      s"You got an email from $sender with title: $title"
    case SMS(number, message) =>
      s"You got an SMS from $number! Message: $message"
    case VoiceRecording(name, link) =>
      s"you received a Voice Recording from $name! Click the link to hear it: $link"
  }
}
val someSms = SMS("12345", "Are you there?")
val someVoiceRecording = VoiceRecording("Tom", "voicerecording.org/id/123")

println(showNotification(someSms))  // prints You got an SMS from 12345! Message: Are you there?

println(showNotification(someVoiceRecording))  // you received a Voice Recording from Tom! Click the link to hear it: voicerecording.org/id/123
```

`showNotification`函数接受一个抽象类`Notification`对象作为输入参数，然后匹配其具体类型。（也就是判断它是一个`Email`，`SMS`，还是`VoiceRecording`）。在`case Email(sender, title, _)`中，对象的`sender`和`title`属性在返回值中被使用，而`body`属性则被忽略，故使用`_`代替。

模式守卫（Pattern gaurds）

为了让匹配更加具体，可以使用模式守卫，也就是在模式后面加上`if <boolean expression>`。

```

def showImportantNotification(notification: Notification, importantPeopleInfo: Seq[String]): String = {
  notification match {
    case Email(sender, _, _) if importantPeopleInfo.contains(sender) =>
      "You got an email from special someone!"
```

```
      case SMS(number, _) if importantPeopleInfo.contains(number) =>
        "You got an SMS from special someone!"
      case other =>
        showNotification(other) // nothing special, delegate to our original showNotification function
    }
  }

  val importantPeopleInfo = Seq("867-5309", "jenny@gmail.com")

  val someSms = SMS("867-5309", "Are you there?")
  val someVoiceRecording = VoiceRecording("Tom", "voicerecording.org/id/123")
  val importantEmail = Email("jenny@gmail.com", "Drinks tonight?", "I'm free after 5!")
  val importantSms = SMS("867-5309", "I'm here! Where are you?")

  println(showImportantNotification(someSms, importantPeopleInfo))
  println(showImportantNotification(someVoiceRecording, importantPeopleInfo))
  println(showImportantNotification(importantEmail, importantPeopleInfo))
  println(showImportantNotification(importantSms, importantPeopleInfo))
```

在`case Email(sender, _, _) if importantPeopleInfo.contains(sender)`中，除了要求`notification`是`Email`类型外，还需要`sender`在重要人物列表`importantPeopleInfo`中，才会匹配到该模式。

仅匹配类型

也可以仅匹配类型，如下所示：

```
  abstract class Device
  case class Phone(model: String) extends Device {
    def screenOff = "Turning screen off"
  }
  case class Computer(model: String) extends Device {
    def screenSaverOn = "Turning screen saver on..."
  }

  def goIdle(device: Device) = device match {
    case p: Phone => p.screenOff
    case c: Computer => c.screenSaverOn
  }
```

当不同类型对象需要调用不同方法时，仅匹配类型的模式非常有用，如上代码中`goIdle`函数对不同类型的`Device`有着不同的表现。一般使用类型的首字母作为`case`的标识符，例如上述代码中的`p`和`c`，这是一种惯例。

密封类

`trait`和类可以用`sealed`标记为密封的，这意味着其所有子类都必须与之定义在相同文件中，从而保证所有子类型都是已知的。

```
sealed abstract class Furniture
case class Couch() extends Furniture
case class Chair() extends Furniture

def findPlaceToSit(piece: Furniture): String = piece match {
  case a: Couch => "Lie on the couch"
  case b: Chair => "Sit on the chair"
}
```

这对于模式匹配很有用，因为我们不再需要一个匹配其他任意情况的`case`。

备注

Scala的模式匹配语句对于使用`case`类（case classes）表示的类型非常有用，同时也可以利用提取器对象（extractor objects）中的`unapply`方法来定义非`case`类对象的匹配。

单例对象

单例对象是一种特殊的类，有且只有一个实例。和惰性变量一样，单例对象是延迟创建的，当它第一次被使用时创建。

当对象定义于顶层时(即没有包含在其他类中)，单例对象只有一个实例。

当对象定义在一个类或方法中时，单例对象表现得和惰性变量一样。

定义一个单例对象

一个单例对象是就是一个值。单例对象的定义方式很像类，但是使用关键字`object`：

```
object Box
```

下面例子中的单例对象包含一个方法：

```
package logging

object Logger {
  def info(message: String): Unit = println(s"INFO: $message")
}
```

方法`info`可以在程序中的任何地方被引用。像这样创建功能性方法是单例对象的一种常见用法。

下面让我们来看看如何在另外一个包中使用`info`方法：

```
import logging.Logger.info

class Project(name: String, daysToComplete: Int)

class Test {
  val project1 = new Project("TPS Reports", 1)
  val project2 = new Project("Website redesign", 5)
  info("Created projects")  // Prints "INFO: Created projects"
}
```

因为 import 语句`import logging.Logger.info`，方法`info`在此处是可见的。

import语句要求被导入的标识具有一个“稳定路径”，一个单例对象由于全局唯一，所以具有稳定路径。

注意：如果一个 `object` 没定义在顶层而是定义在另一个类或者单例对象中，那么这个单例对象和其他类普通成员一样是“路径相关的”。这意味着有两种行为，`class Milk` 和 `class OrangeJuice`，一个类成员 `object NutritionInfo` “依赖”于包装它的实例，要么是牛奶要么是橙汁。`milk.NutritionInfo` 则完全不同于 `oj.NutritionInfo`。

伴生对象

当一个单例对象和某个类共享一个名称时，这个单例对象称为 ***伴生对象*** 。 同理，这个类被称为是这个单例对象的伴生类。类和它的伴生对象可以互相访问其私有成员。使用伴生对象来定义那些在伴生类中不依赖于实例化对象而存在的成员变量或者方法。

```
import scala.math._

case class Circle(radius: Double) {
  import Circle._
  def area: Double = calculateArea(radius)
}

object Circle {
  private def calculateArea(radius: Double): Double = Pi * pow(radius, 2.0)
}

val circle1 = Circle(5.0)

circle1.area
```

这里的 `class Circle` 有一个成员 `area` 是和具体的实例化对象相关的，单例对象 `object Circle` 包含一个方法 `calculateArea`，它在每一个实例化对象中都是可见的。

伴生对象也可以包含工厂方法：

```
class Email(val username: String, val domainName: String)

object Email {
  def fromString(emailString: String): Option[Email] = {
    emailString.split('@') match {
      case Array(a, b) => Some(new Email(a, b))
      case _ => None
    }
  }
}

val scalaCenterEmail = Email.fromString("scala.center@epfl.ch")
scalaCenterEmail match {
  case Some(email) => println(
    s"""Registered an email
       |Username: ${email.username}
       |Domain name: ${email.domainName}
     """)
  case None => println("Error: could not parse email")
}
```

伴生对象 `object Email` 包含有一个工厂方法 `fromString` 用来根据一个 String 创建 `Email` 实例。在这里我们返回的是 `Option[Email]` 以防有语法分析错误。

注意：**类和它的伴生对象必须定义在同一个源文件里**。如果需要在 REPL 里定义类和其伴生对象，需要将它们定义在同一行或者进入 `:paste` 模式。

Java 程序员的注意事项

在 Java 中 `static` 成员对应于 Scala 中的伴生对象的普通成员。

在 Java 代码中调用伴生对象时，伴生对象的成员会被定义成伴生类中的 `static` 成员。这称为 *静态转发* 。这种行为发生在当你自己没有定义一个伴生类时。

正则表达式模式

正则表达式是用来找出数据中的指定模式（或缺少该模式）的字符串。 `.r` 方法可使任意字符串变成一个正则表达式。

```
import scala.util.matching.Regex

val numberPattern: Regex = "[0-9]".r

numberPattern.findFirstMatchIn("awesomepassword") match {
  case Some(_) => println("Password OK")
  case None => println("Password must contain a number")
}
```

上例中， `numberPattern` 的类型是正则表达式类 `Regex` ，其作用是确保密码中包含一个数字。

你还可以使用括号来同时匹配多组正则表达式。

```
 import scala.util.matching.Regex

 val keyValPattern: Regex = "([0-9a-zA-Z-#() ]+): ([0-9a-zA-Z-#() ]+)".r

 val input: String =
   """background-color: #A03300;
     |background-image: url(img/header100.png);
     |background-position: top center;
     |background-repeat: repeat-x;
     |background-size: 2160px 108px;
     |margin: 0;
     |height: 108px;
     |width: 100%;""".stripMargin

 for (patternMatch <- keyValPattern.findAllMatchIn(input))
   println(s"key: ${patternMatch.group(1)} value: ${patternMatch.group(2)}")
```

上例解析出了一个字符串中的多个键和值，其中的每个匹配又有一组子匹配，结果如下：

```
key: background-color value: #A03300
key: background-image value: url(img
key: background-position value: top center
key: background-repeat value: repeat-x
key: background-size value: 2160px 108px
key: margin value: 0
key: height value: 108px
key: width value: 100
```

提取器对象

提取器对象是一个包含有 `unapply` 方法的单例对象。 `apply` 方法就像一个构造器，接受参数然后创建一个实例对象，反之 `unapply` 方法接受一个实例对象然后返回最初创建它所用的参数。提取器常用在模式匹配和偏函数中。

```
object CustomerID {

  def apply(name: String) = s"$name--${Random.nextLong}"

  def unapply(customerID: String): Option[String] = {
    val stringArray: Array[String] = customerID.split("--")
    if (stringArray.tail.nonEmpty) Some(stringArray.head) else None
  }
}

val customer1ID = CustomerID("Sukyoung")  // Sukyoung--23098234908
customer1ID match {
  case CustomerID(name) => println(name)  // prints Sukyoung
  case _ => println("Could not extract a CustomerID")
}
```

这里 `apply` 方法用 `name` 创建一个 `CustomerID` 字符串。而 `unapply` 方法正好相反，它返回 `name` 。当我们调用 `CustomerID("Sukyoung")` ，其实是调用了 `CustomerID.apply("Sukyoung")` 的简化语法。当我们调用 `case CustomerID(name) => println(name)` ，就是在调用提取器方法。

因为变量定义可以使用模式引入变量，提取器可以用来初始化这个变量，使用 unapply 方法来生成值。

```
val customer2ID = CustomerID("Nico")
val CustomerID(name) = customer2ID
println(name)  // prints Nico
```

上面的代码等价于 `val name = CustomerID.unapply(customer2ID).get` 。

```
val CustomerID(name2) = "--asdfasdfasdf"
```

如果没有匹配的值，会抛出 `scala.MatchError` :

```
val CustomerID(name3) = "-asdfasdfasdf"
```

`unapply` 方法的返回值应当符合下面的某一条:

- 如果只是用来判断真假，可以返回一个 `Boolean` 类型的值。例如 `case even()` 。
- 如果只是用来提取单个 T 类型的值，可以返回 `Option[T]` 。
- 如果你想要提取多个值，类型分别为 `T1,...,Tn` ，可以把它们放在一个可选的元组中 `Option[(T1,...,Tn)]` 。

有时，要提取的值的数量不是固定的，因此我们想根据输入来返回随机数量的值。这种情况下，你可以用 `unapplySeq` 方法来定义提取器，此方法返回 `Option[Seq[T]]` 。常见的例子有，用 `case List(x, y, z) =>` 来解构一个列表 `List` ，以及用一个正则表达式 `Regex` 来分解一个字符串 `String` ，例如 `case r(name, remainingFields @ _*) =>` 。

FOR 表达式

Scala 提供一个轻量级的标记方式用来表示 *序列推导* 。推导使用形式为 `for (enumerators) yield e` 的 for 表达式，此处 `enumerators` 指一组以分号分隔的枚举器。一个 *enumerator* 要么是一个产生新变量的生成器，要么是一个过滤器。for 表达式在枚举器产生的每一次绑定中都会计算 `e` 值，并在循环结束后返回这些值组成的序列。

看下例：

```
case class User(name: String, age: Int)

val userBase = List(User("Travis", 28),
                    User("Kelly", 33),
                    User("Jennifer", 44),
                    User("Dennis", 23))

val twentySomethings = for (user <- userBase if (user.age >= 20 && user.age < 30))
yield user.name // i.e. add this to a list

twentySomethings.foreach(name => println(name)) // prints Travis Dennis
```

这里 `n == 10` 和 `v == 10`。在第一次迭代时，`i == 0` 并且 `j == 0` 所以 `i + j != v` 因此没有返回值被生成。在 `i` 的值递增到 `1` 之前，`j` 的值又递增了 9 次。如果没有 `if` 语句过滤，上面的例子只会打印出如下的结果：

```
(0, 0) (0, 1) (0, 2) (0, 3) (0, 4) (0, 5) (0, 6) (0, 7) (0, 8) (0, 9) (1, 1) ...
```

注意 for 表达式并不局限于使用列表。任何数据类型只要支持 `withFilter`，`map`，和 `flatMap` 操作（不同数据类型可能支持不同的操作）都可以用来做序列推导。

你可以在使用 for 表达式时省略 `yield` 语句。此时会返回 `Unit`。当你想要执行一些副作用的时候这很有用。下面的例子输出和上面相同的结果，但是没有使用 `yield`：

```
def foo(n: Int, v: Int) =
   for (i <- 0 until n;
        j <- i until n if i + j == v)
   println(s"($i, $j)")

foo(10, 10)
```

泛型类

泛型类指可以接受类型参数的类。泛型类在集合类中被广泛使用。

定义一个泛型类

泛型类使用方括号 `[]` 来接受类型参数。一个惯例是使用字母 `A` 作为参数标识符，当然你可以使用任何参数名称。

```
class Stack[A] {
  private var elements: List[A] = Nil
  def push(x: A) { elements = x :: elements }
  def peek: A = elements.head
  def pop(): A = {
    val currentTop = peek
    elements = elements.tail
    currentTop
  }
}
```

上面的 `Stack` 类的实现中接受类型参数 `A`。 这表示其内部的列表， `var elements: List[A] = Nil`，只能够存储类型 `A` 的元素。方法 `def push` 只接受类型 `A` 的实例对象作为参数(注意: `elements = x :: elements` 将 `elements` 放到了一个将元素 `x` 添加到 `elements` 的头部而生成的新列表中)。

使用

要使用一个泛型类，将一个具体类型放到方括号中来代替 `A`。

```
val stack = new Stack[Int]
stack.push(1)
stack.push(2)
println(stack.pop)  // prints 2
println(stack.pop)  // prints 1
```

实例对象 `stack` 只能接受整型值。然而，如果类型参数有子类型，子类型可以被传入：

```
class Fruit
class Apple extends Fruit
class Banana extends Fruit

val stack = new Stack[Fruit]
val apple = new Apple
val banana = new Banana

stack.push(apple)
stack.push(banana)
```

类 `Apple` 和类 `Banana` 都继承自类 `Fruit`，所以我们可以把实例对象 `apple` 和 `banana` 压入栈 `Fruit` 中。

*注意：泛型类型的子类型是*不可传导*的。这表示如果我们有一个字母类型的栈 `Stack[Char]`，那它不能被用作一个整型的栈 `Stack[Int]`。否则就是不安全的，因为它将使我们能够在字母型的栈中插入真正的整型值。结论就是，只有当类型 `B = A` 时， `Stack[A]` 是 `Stack[B]` 的子类型才成立。因为此处可能会有很大的限制，Scala 提供了一种 类型参数注释机制 用以控制泛型类型的子类型的行为。*

型变

型变是复杂类型的子类型关系与其组件类型的子类型关系的相关性。 Scala支持 泛型类 的类型参数的型变注释，允许它们是协变的，逆变的，或在没有使用注释的情况下是不变的。 在类型系统中使用型变允许我们在复杂类型之间建立直观的连接，而缺乏型变则会限制类抽象的重用性。

```scala
class Foo[+A] // A covariant class
class Bar[-A] // A contravariant class
class Baz[A]  // An invariant class
```

协变

使用注释 `+A`，可以使一个泛型类的类型参数 `A` 成为协变。 对于某些类 `class List[+A]`，使 `A` 成为协变意味着对于两种类型 `A` 和 `B`，如果 `A` 是 `B` 的子类型，那么 `List[A]` 就是 `List[B]` 的子类型。 这允许我们使用泛型来创建非常有用和直观的子类型关系。

考虑以下简单的类结构：

```scala
abstract class Animal {
  def name: String
}
case class Cat(name: String) extends Animal
case class Dog(name: String) extends Animal
```

类型 `Cat` 和 `Dog` 都是 `Animal` 的子类型。 Scala 标准库有一个通用的不可变的类 `sealed abstract class List[+A]`，其中类型参数 `A` 是协变的。 这意味着 `List[Cat]` 是 `List[Animal]`， `List[Dog]` 也是 `List[Animal]`。 直观地说，猫的列表和狗的列表都是动物的列表是合理的，你应该能够用它们中的任何一个替换 `List[Animal]`。

在下例中，方法 `printAnimalNames` 将接受动物列表作为参数，并且逐行打印出它们的名称。 如果 `List[A]` 不是协变的，最后两个方法调用将不能编译，这将严重限制 `printAnimalNames` 方法的适用性。

```scala
object CovarianceTest extends App {
  def printAnimalNames(animals: List[Animal]): Unit = {
    animals.foreach { animal =>
      println(animal.name)
    }
  }

  val cats: List[Cat] = List(Cat("Whiskers"), Cat("Tom"))
  val dogs: List[Dog] = List(Dog("Fido"), Dog("Rex"))

  printAnimalNames(cats)
  // Whiskers
  // Tom

  printAnimalNames(dogs)
  // Fido
  // Rex
}
```

逆变

通过使用注释 `-A`，可以使一个泛型类的类型参数 `A` 成为逆变。 与协变类似，这会在类及其类型参数之间创建一个子类型关系，但其作用与协变完全相反。 也就是说，对于某个类 `class Writer[-A]`，使 `A` 逆变意味着对于两种类型 `A` 和 `B`，如果 `A` 是 `B` 的子类型，那么 `Writer[B]` 是 `Writer[A]` 的子类型。

考虑在下例中使用上面定义的类 `Cat`， `Dog` 和 `Animal`：

```
abstract class Printer[-A] {
  def print(value: A): Unit
}
```

这里 `Printer[A]` 是一个简单的类，用来打印出某种类型的 `A`。 让我们定义一些特定的子类：

```
class AnimalPrinter extends Printer[Animal] {
  def print(animal: Animal): Unit =
    println("The animal's name is: " + animal.name)
}

class CatPrinter extends Printer[Cat] {
  def print(cat: Cat): Unit =
    println("The cat's name is: " + cat.name)
}
```

如果 `Printer[Cat]` 知道如何在控制台打印出任意 `Cat`，并且 `Printer[Animal]` 知道如何在控制台打印出任意 `Animal`，那么 `Printer[Animal]` 也应该知道如何打印出 `Cat` 就是合理的。 反向关系不适用，因为 `Printer[Cat]` 并不知道如何在控制台打印出任意 `Animal`。 因此，如果我们愿意，我们应该能够用 `Printer[Animal]` 替换 `Printer[Cat]`，而使 `Printer[A]` 逆变允许我们做到这一点。

```
object ContravarianceTest extends App {
  val myCat: Cat = Cat("Boots")

  def printMyCat(printer: Printer[Cat]): Unit = {
    printer.print(myCat)
  }

  val catPrinter: Printer[Cat] = new CatPrinter
  val animalPrinter: Printer[Animal] = new AnimalPrinter

  printMyCat(catPrinter)
  printMyCat(animalPrinter)
}
```

这个程序的输出如下：

```
The cat's name is: Boots
The animal's name is: Boots
```

不变

默认情况下，Scala中的泛型类是不变的。 这意味着它们既不是协变的也不是逆变的。 在下例中，类 `Container` 是不变的。 `Container[Cat]` *不是* `Container[Animal]`，反之亦然。

```
class Container[A](value: A) {
  private var _value: A = value
  def getValue: A = _value
  def setValue(value: A): Unit = {
    _value = value
  }
}
```

可能看起来一个 `Container[Cat]` 自然也应该是一个 `Container[Animal]`，但允许一个可变的泛型类成为协变并不安全。 在这个例子中，`Container` 是不变的非常重要。 假设 `Container` 实际上是协变的，下面的情况可能会发生：

```
val catContainer: Container[Cat] = new Container(Cat("Felix"))
val animalContainer: Container[Animal] = catContainer
animalContainer.setValue(Dog("Spot"))
val cat: Cat = catContainer.getValue // 糟糕，我们最终会将一只狗作为值分配给一只猫
```

幸运的是，编译器在此之前就会阻止我们。

其他例子

另一个可以帮助理解型变的例子是 Scala 标准库中的 `trait Function1[-T, +R]`。 `Function1` 表示具有一个参数的函数，其中第一个类型参数 `T` 表示参数类型，第二个类型参数 `R` 表示返回类型。 `Function1` 在其参数类型上是逆变的，并且在其返回类型上是协变的。 对于这个例子，我们将使用文字符号 `A => B` 来表示 `Function1[A, B]`。

假设前面使用过的类似 `Cat`， `Dog`， `Animal` 的继承关系，加上以下内容：

```
abstract class SmallAnimal extends Animal
case class Mouse(name: String) extends SmallAnimal
```

假设我们正在处理接受动物类型的函数，并返回他们的食物类型。 如果我们想要一个 `Cat => SmallAnimal`（因为猫吃小动物），但是给它一个 `Animal => Mouse`，我们的程序仍然可以工作。 直观地看，一个 `Animal => Mouse` 的函数仍然会接受一个 `Cat` 作为参数，因为 `Cat` 即是一个 `Animal`，并且这个函数返回一个 `Mouse`，也是一个 `SmallAnimal`。 既然我们可以安全地，隐式地用后者代替前者，我们可以说 `Animal => Mouse` 是 `Cat => SmallAnimal` 的子类型。

与其他语言的比较

某些与 Scala 类似的语言以不同的方式支持型变。 例如，Scala 中的型变注释与 C# 中的非常相似，在定义类抽象时添加型变注释（声明点型变）。 但是在Java中，当类抽象被使用时（使用点型变），才会给出型变注释。

类型上界

在Scala中， 类型参数 和 抽象类型 都可以有一个类型边界约束。这种类型边界在限制类型变量实际取值的同时还能展露类型成员的更多信息。比如像 `T <: A` 这样声明的类型上界表示类型变量 `T` 应该是类型 `A` 的子类。下面的例子展示了类 `PetContainer` 的一个类型参数的类型上界。

```
abstract class Animal {
 def name: String
}

abstract class Pet extends Animal {}

class Cat extends Pet {
  override def name: String = "Cat"
}

class Dog extends Pet {
  override def name: String = "Dog"
}

class Lion extends Animal {
  override def name: String = "Lion"
}
```

```
18
19  class PetContainer[P <: Pet](p: P) {
20    def pet: P = p
21  }
22
23  val dogContainer = new PetContainer[Dog](new Dog)
24  val catContainer = new PetContainer[Cat](new Cat)
```

```
1  // this would not compile
2  val lionContainer = new PetContainer[Lion](new Lion)
```

类 `PetContainer` 接受一个必须是 `Pet` 子类的类型参数 `P`。因为 `Dog` 和 `Cat` 都是 `Pet` 的子类，所以可以构造 `PetContainer[Dog]` 和 `PetContainer[Cat]`。但在尝试构造 `PetContainer[Lion]` 的时候会得到下面的错误信息：

```
1  type arguments [Lion] do not conform to class PetContainer's type parameter bounds
   [P <: Pet]
```

这是因为 `Lion` 并不是 `Pet` 的子类。

类型下界

类型上界 将类型限制为另一种类型的子类型，而 *类型下界* 将类型声明为另一种类型的超类型。 术语 `B >: A` 表示类型参数 `B` 或抽象类型 `B` 是类型 `A` 的超类型。 在大多数情况下， `A` 将是类的类型参数，而 `B` 将是方法的类型参数。

下面看一个适合用类型下界的例子：

```
1  trait Node[+B] {
2    def prepend(elem: B): Node[B]
3  }
4
5  case class ListNode[+B](h: B, t: Node[B]) extends Node[B] {
6    def prepend(elem: B): ListNode[B] = ListNode(elem, this)
7    def head: B = h
8    def tail: Node[B] = t
9  }
10
11 case class Nil[+B]() extends Node[B] {
12   def prepend(elem: B): ListNode[B] = ListNode(elem, this)
13 }
```

该程序实现了一个单链表。 `Nil` 表示空元素（即空列表）。 `class ListNode` 是一个节点，它包含一个类型为 `B` (`head`) 的元素和一个对列表其余部分的引用 (`tail`)。 `class Node` 及其子类型是协变的，因为我们定义了 `+B`。

但是，这个程序 *不能* 编译，因为方法 `prepend` 中的参数 `elem` 是 *协* 变的 `B` 类型。 这会出错，因为函数的参数类型是 *逆* 变的，而返回类型是 *协* 变的。

要解决这个问题，我们需要将方法 `prepend` 的参数 `elem` 的型变翻转。 我们通过引入一个新的类型参数 `U` 来实现这一点，该参数具有 `B` 作为类型下界。

```
trait Node[+B] {
  def prepend[U >: B](elem: U): Node[U]
}

case class ListNode[+B](h: B, t: Node[B]) extends Node[B] {
  def prepend[U >: B](elem: U): ListNode[U] = ListNode(elem, this)
  def head: B = h
  def tail: Node[B] = t
}

case class Nil[+B]() extends Node[B] {
  def prepend[U >: B](elem: U): ListNode[U] = ListNode(elem, this)
}
```

现在我们像下面这么做：

```
trait Bird
case class AfricanSwallow() extends Bird
case class EuropeanSwallow() extends Bird


val africanSwallowList= ListNode[AfricanSwallow](AfricanSwallow(), Nil())
val birdList: Node[Bird] = africanSwallowList
birdList.prepend(EuropeanSwallow())
```

可以为 `Node[Bird]` 赋值 `africanSwallowList`，然后再加入一个 `EuropeanSwallow`。

内部类

在Scala中，一个类可以作为另一个类的成员。 在一些类似 Java 的语言中，内部类是外部类的成员，而 Scala 正好相反，内部类是绑定到外部对象的。 假设我们希望编译器在编译时阻止我们混淆节点 nodes 与图形 graph 的关系，路径依赖类型提供了一种解决方案。

为了说明差异，我们简单描述了一个图形数据类型的实现：

```
class Graph {
  class Node {
    var connectedNodes: List[Node] = Nil
    def connectTo(node: Node) {
      if (!connectedNodes.exists(node.equals)) {
        connectedNodes = node :: connectedNodes
      }
    }
  }
  var nodes: List[Node] = Nil
  def newNode: Node = {
    val res = new Node
    nodes = res :: nodes
    res
  }
}
```

该程序将图形表示为节点列表 (`List[Node]`)。 每个节点都有一个用来存储与其相连的其他节点的列表 (`connectedNodes`)。 类 `Node` 是一个 *路径依赖类型* ，因为它嵌套在类 `Graph` 中。 因此，`connectedNodes` 中存储的所有节点必须使用同一个 `Graph` 的实例对象的 `newNode` 方法来创建。

```
val graph1: Graph = new Graph
val node1: graph1.Node = graph1.newNode
val node2: graph1.Node = graph1.newNode
val node3: graph1.Node = graph1.newNode
node1.connectTo(node2)
node3.connectTo(node1)
```

为清楚起见，我们已经明确地将 `node1`，`node2`，和 `node3` 的类型声明为 `graph1.Node`，但编译器其实可以自动推断出它。 这是因为当我们通过调用 `graph1.newNode` 来调用 `new Node` 时，该方法产生特定于实例 `graph1` 的 `Node` 类型的实例对象。

如果我们现在有两个图形，Scala 的类型系统不允许我们将一个图形中定义的节点与另一个图形的节点混合，因为另一个图形的节点具有不同的类型。 下例是一个非法的程序：

```
val graph1: Graph = new Graph
val node1: graph1.Node = graph1.newNode
val node2: graph1.Node = graph1.newNode
node1.connectTo(node2)      // legal
val graph2: Graph = new Graph
val node3: graph2.Node = graph2.newNode
node1.connectTo(node3)      // illegal!
```

类型 `graph1.Node` 与类型 `graph2.Node` 完全不同。 在 Java 中，上一个示例程序中的最后一行是正确的。 对于两个图形的节点，Java 将分配相同的类型 `Graph.Node`；即 `Node` 以类 `Graph` 为前缀。在Scala中也可以表示出这种类型，它写成了 `Graph#Node`。 如果我们希望能够连接不同图形的节点，我们必须通过以下方式更改图形类的初始实现的定义：

```
class Graph {
  class Node {
    var connectedNodes: List[Graph#Node] = Nil
    def connectTo(node: Graph#Node) {
      if (!connectedNodes.exists(node.equals)) {
        connectedNodes = node :: connectedNodes
      }
    }
  }
  var nodes: List[Node] = Nil
  def newNode: Node = {
    val res = new Node
    nodes = res :: nodes
    res
  }
}
```

抽象类型

特质和抽象类可以包含一个抽象类型成员，意味着实际类型可由具体实现来确定。例如：

```
trait Buffer {
  type T
  val element: T
}
```

这里定义的抽象类型 `T` 是用来描述成员 `element` 的类型的。通过抽象类来扩展这个特质后，就可以添加一个类型上边界来让抽象类型 `T` 变得更加具体。

```
abstract class SeqBuffer extends Buffer {
  type U
  type T <: Seq[U]
  def length = element.length
}
```

注意这里是如何借助另外一个抽象类型 `U` 来限定类型上边界的。通过声明类型 `T` 只可以是 `Seq[U]` 的子类（其中U是一个新的抽象类型），这个 `SeqBuffer` 类就限定了缓冲区中存储的元素类型只能是序列。

含有抽象类型成员的特质或类（**classes**）经常和匿名类的初始化一起使用。为了能够阐明问题，下面看一段程序，它处理一个涉及整型列表的序列缓冲区。

```
abstract class IntSeqBuffer extends SeqBuffer {
  type U = Int
}


def newIntSeqBuf(elem1: Int, elem2: Int): IntSeqBuffer =
  new IntSeqBuffer {
       type T = List[U]
       val element = List(elem1, elem2)
     }
val buf = newIntSeqBuf(7, 8)
println("length = " + buf.length)
println("content = " + buf.element)
```

这里的工厂方法 `newIntSeqBuf` 使用了 `IntSeqBuf` 的匿名类实现方式，其类型 `T` 被设置成了 `List[Int]`。

把抽象类型成员转成类的类型参数或者反过来，也是可行的。如下面这个版本只用了类的类型参数来转换上面的代码：

```
abstract class Buffer[+T] {
  val element: T
}
abstract class SeqBuffer[U, +T <: Seq[U]] extends Buffer[T] {
  def length = element.length
}

def newIntSeqBuf(e1: Int, e2: Int): SeqBuffer[Int, Seq[Int]] =
  new SeqBuffer[Int, List[Int]] {
    val element = List(e1, e2)
  }

val buf = newIntSeqBuf(7, 8)
println("length = " + buf.length)
println("content = " + buf.element)
```

需要注意的是为了隐藏从方法 `newIntSeqBuf` 返回的对象的具体序列实现的类型，这里的 **型变**（`+T <: Seq[U]`）是必不可少的。此外要说明的是，有些情况下用类型参数替换抽象类型是行不通的。

复合类型

有时需要表明一个对象的类型是其他几种类型的子类型。 在 Scala 中，这可以表示成 *复合类型*，即多个类型的交集。

假设我们有两个特质 `Cloneable` 和 `Resetable`：

```
trait Cloneable extends java.lang.Cloneable {
  override def clone(): Cloneable = {
    super.clone().asInstanceOf[Cloneable]
  }
}
trait Resetable {
  def reset: Unit
}
```

现在假设我们要编写一个方法 `cloneAndReset`，此方法接受一个对象，克隆它并重置原始对象：

```
def cloneAndReset(obj: ?): Cloneable = {
  val cloned = obj.clone()
  obj.reset
  cloned
}
```

这里出现一个问题，参数 `obj` 的类型是什么。 如果类型是 `Cloneable` 那么参数对象可以被克隆 `clone`，但不能重置 `reset`；如果类型是 `Resetable` 我们可以重置 `reset` 它，但却没有克隆 `clone` 操作。 为了避免在这种情况下进行类型转换，我们可以将 `obj` 的类型同时指定为 `Cloneable` 和 `Resetable`。 这种复合类型在 Scala 中写成：`Cloneable with Resetable`。

以下是更新后的方法：

```
def cloneAndReset(obj: Cloneable with Resetable): Cloneable = {
  //...
}
```

复合类型可以由多个对象类型构成，这些对象类型可以有单个细化，用于缩短已有对象成员的签名。 格式为：`A with B with C ... { refinement }`

关于使用细化的例子参考 组合类。

自类型

自类型用于声明一个特质必须混入其他特质，尽管该特质没有直接扩展其他特质。 这使得所依赖的成员可以在没有导入的情况下使用。

自类型是一种细化 `this` 或 `this` 别名之类型的方法。 语法看起来像普通函数语法，但是意义完全不一样。

要在特质中使用自类型，写一个标识符，跟上要混入的另一个特质，以及 `=>`（例如 `someIdentifier: SomeOtherTrait =>`）。

```
trait User {
  def username: String
}

trait Tweeter {
  this: User =>  // 重新赋予 this 的类型
  def tweet(tweetText: String) = println(s"$username: $tweetText")
}

class VerifiedTweeter(val username_ : String) extends Tweeter with User {  // 我们混入特质 User 因为 Tweeter 需要
```

```
    def username = s"real $username_"
}

val realBeyoncé = new VerifiedTweeter("Beyoncé")
realBeyoncé.tweet("Just spilled my glass of lemonade")  // 打印出 "real Beyoncé:
Just spilled my glass of lemonade"
```

因为我们在特质 `trait Tweeter` 中定义了 `this: User =>`，现在变量 `username` 可以在 `tweet` 方法内使用。 这也意味着，由于 `VerifiedTweeter` 继承了 `Tweeter`，它还必须混入 `User`（使用 `with User`）。

隐式参数

方法可以具有 *隐式* 参数列表，由参数列表开头的 *implicit* 关键字标记。 如果参数列表中的参数没有像往常一样传递， Scala 将查看它是否可以获得正确类型的隐式值，如果可以，则自动传递。

Scala 将查找这些参数的位置分为两类：

- Scala 在调用包含有隐式参数块的方法时，将首先查找可以直接访问的隐式定义和隐式参数 (无前缀)。
- 然后，它在所有伴生对象中查找与隐式候选类型相关的有隐式标记的成员。

在下面的例子中，我们定义了一个方法 `sum`，它使用 Monoid 类的 `add` 和 `unit` 方法计算一个列表中元素的总和。 请注意，隐式值不能是顶级值。

```
abstract class Monoid[A] {
  def add(x: A, y: A): A
  def unit: A
}

object ImplicitTest {
  implicit val stringMonoid: Monoid[String] = new Monoid[String] {
    def add(x: String, y: String): String = x concat y
    def unit: String = ""
  }

  implicit val intMonoid: Monoid[Int] = new Monoid[Int] {
    def add(x: Int, y: Int): Int = x + y
    def unit: Int = 0
  }

  def sum[A](xs: List[A])(implicit m: Monoid[A]): A =
    if (xs.isEmpty) m.unit
    else m.add(xs.head, sum(xs.tail))

  def main(args: Array[String]): Unit = {
    println(sum(List(1, 2, 3)))       // uses IntMonoid implicitly
    println(sum(List("a", "b", "c"))) // uses StringMonoid implicitly
  }
}
```

类 `Monoid` 定义了一个名为 `add` 的操作，它将一对 `A` 类型的值相加并返回一个 `A`，以及一个名为 `unit` 的操作，用来创建一个（特定的） `A` 类型的值。

为了说明隐式参数如何工作，我们首先分别为字符串和整数定义 Monoid 实例， `StringMonoid` 和 `IntMonoid`。 `implicit` 关键字表示可以隐式使用相应的对象。

方法 `sum` 接受一个 `List[A]`，并返回一个 `A` 的值，它从 `unit` 中取初始的 `A` 值，并使用 `add` 方法依次将列表中的下一个 `A` 值相加。在这里将参数 `m` 定义为隐式意味着，如果 Scala 可以找到隐式 `Monoid[A]` 用于隐式参数 `m`，我们在调用 `sum` 方法时只需要传入 `xs` 参数。

在 `main` 方法中我们调用了 `sum` 方法两次，并且只传入参数 `xs`。 Scala 会在上例的上下文范围内寻找隐式值。 第一次调用 `sum` 方法的时候传入了一个 `List[Int]` 作为 `xs` 的值，这意味着此处类型 `A` 是 `Int`。 隐式参数列表 `m` 被省略了，因此 Scala 将查找类型为 `Monoid[Int]` 的隐式值。 第一查找规则如下

> Scala 在调用包含有隐式参数块的方法时，将首先查找可以直接访问的隐式定义和隐式参数 (无前缀)。

`intMonoid` 是一个隐式定义，可以在 `main` 中直接访问。 并且它的类型也正确，因此它会被自动传递给 `sum` 方法。

第二次调用 `sum` 方法的时候传入一个 `List[String]`，这意味着此处类型 `A` 是 `String`。 与查找 `Int` 型的隐式参数时类似，但这次会找到 `stringMonoid`，并自动将其作为 `m` 传入。

该程序将输出

```
6
abc
```

隐式转换

一个从类型 `S` 到类型 `T` 的隐式转换由一个函数类型 `S => T` 的隐式值来定义，或者由一个可转换成所需值的隐式方法来定义。

隐式转换在两种情况下会用到:

- 如果一个表达式 `e` 的类型为 `S`， 并且类型 `S` 不符合表达式的期望类型 `T`。
- 在一个类型为 `S` 的实例对象 `e` 中调用 `e.m`， 如果被调用的 `m` 并没有在类型 `S` 中声明。

在第一种情况下，搜索转换 `c`，它适用于 `e`，并且结果类型为 `T`。 在第二种情况下，搜索转换 `c`，它适用于 `e`，其结果包含名为 `m` 的成员。

如果一个隐式方法 `List[A] => Ordered[List[A]]`，以及一个隐式方法 `Int => Ordered[Int]` 在上下文范围内，那么对下面两个类型为 `List[Int]` 的列表的操作是合法的:

```
List(1, 2, 3) <= List(4, 5)
```

在 `scala.Predef.intWrapper` 已经自动提供了一个隐式方法 `Int => Ordered[Int]`。下面提供了一个隐式方法 `List[A] => Ordered[List[A]]` 的例子。

```
import scala.language.implicitConversions

implicit def list2ordered[A](x: List[A])
    (implicit elem2ordered: A => Ordered[A]): Ordered[List[A]] =
  new Ordered[List[A]] {
    //replace with a more useful implementation
    def compare(that: List[A]): Int = 1
  }
```

自动导入的对象 `scala.Predef` 声明了几个预定义类型 (例如 `Pair`) 和方法 (例如 `assert`)，同时也声明了一些隐式转换。

例如，当调用一个接受 `java.lang.Integer` 作为参数的 Java 方法时，你完全可以传入一个 `scala.Int`。那是因为 Predef 包含了以下的隐式转换：

```
import scala.language.implicitConversions

implicit def int2Integer(x: Int) =
  java.lang.Integer.valueOf(x)
```

因为如果不加选择地使用隐式转换可能会导致陷阱，编译器会在编译隐式转换定义时发出警告。

要关闭警告，执行以下任一操作：

- 将 `scala.language.implicitConversions` 导入到隐式转换定义的上下文范围内
- 启用编译器选项 `-language:implicitConversions`

在编译器应用隐式转换时不会发出警告。

多态方法

Scala 中的方法可以按类型和值进行参数化。 语法和泛型类类似。 类型参数括在方括号中，而值参数括在圆括号中。

看下面的例子：

```
def listOfDuplicates[A](x: A, length: Int): List[A] = {
  if (length < 1)
    Nil
  else
    x :: listOfDuplicates(x, length - 1)
}
println(listOfDuplicates[Int](3, 4))  // List(3, 3, 3, 3)
println(listOfDuplicates("La", 8))  // List(La, La, La, La, La, La, La, La)
```

方法 `listOfDuplicates` 具有类型参数 `A` 和值参数 `x` 和 `length`。 值 `x` 是 `A` 类型。 如果 `length < 1`，我们返回一个空列表。 否则我们将 `x` 添加到递归调用返回的重复列表中。 （注意，`::` 表示将左侧的元素添加到右侧的列表中。）

上例中第一次调用方法时，我们显式地提供了类型参数 `[Int]`。 因此第一个参数必须是 `Int` 类型，并且返回类型为 `List[Int]`。

上例中第二次调用方法，表明并不总是需要显式提供类型参数。 编译器通常可以根据上下文或值参数的类型来推断。 在这个例子中，`"La"` 是一个 `String`，因此编译器知道 `A` 必须是 `String`。

类型推断

Scala 编译器通常可以推断出表达式的类型，因此你不必显式地声明它。

省略类型

```
val businessName = "Montreux Jazz Café"
```

编译器可以发现 `businessName` 是 String 类型。 它的工作原理和方法类似：

```
def squareOf(x: Int) = x * x
```

编译器可以推断出方法的返回类型为 `Int`，因此不需要明确地声明返回类型。

对于递归方法，编译器无法推断出结果类型。 下面这个程序就是由于这个原因而编译失败：

```
1   def fac(n: Int) = if (n == 0) 1 else n * fac(n - 1)
```

当调用 多态方法 或实例化 泛型类 时，也不必明确指定类型参数。 Scala 编译器将从上下文和实际方法的类型/构造函数参数的类型推断出缺失的类型参数。

看下面两个例子：

```
1   case class MyPair[A, B](x: A, y: B)
2   val p = MyPair(1, "scala") // type: MyPair[Int, String]
3
4   def id[T](x: T) = x
5   val q = id(1)              // type: Int
```

编译器使用传给 `MyPair` 参数的类型来推断出 `A` 和 `B` 的类型。对于 `x` 的类型同样如此。

参数

编译器从不推断方法形式参数的类型。 但是，在某些情况下，当函数作为参数传递时，编译器可以推断出匿名函数形式参数的类型。

```
1   Seq(1, 3, 4).map(x => x * 2)  // List(2, 6, 8)
```

方法 map 的形式参数是 `f: A => B`。 因为我们把整数放在 `Seq` 中，编译器知道 `A` 是 `Int` 类型(即 `x` 是一个整数)。 因此，编译器可以从 `x * 2` 推断出 `B` 是 `Int` 类型。

何时不要依赖类型推断

通常认为，公开可访问的 API 成员应该具有显示类型声明以增加可读性。 因此，我们建议你将代码中向用户公开的任何 API 明确指定类型。

此外，类型推断有时会推断出太具体的类型。 假设我们这么写：

```
1   var obj = null
```

我们就不能进行重新赋值：

```
1   obj = new AnyRef
```

它不能编译，因为 `obj` 推断出的类型是 `Null`。 由于该类型的唯一值是 `null`，因此无法分配其他的值。

运算符

在Scala中，运算符即是方法。 任何具有单个参数的方法都可以用作 *中缀运算符* 。 例如，可以使用点号调用 `+`：

```
1   10.+(1)
```

而中缀运算符则更易读：

```
1   10 + 1
```

定义和使用运算符

你可以使用任何合法标识符作为运算符。 包括像 `add` 这样的名字或像 `+` 这样的符号。

```
case class Vec(x: Double, y: Double) {
  def +(that: Vec) = Vec(this.x + that.x, this.y + that.y)
}

val vector1 = Vec(1.0, 1.0)
val vector2 = Vec(2.0, 2.0)

val vector3 = vector1 + vector2
vector3.x  // 3.0
vector3.y  // 3.0
```

类 Vec 有一个方法 `+`，我们用它来使 `vector1` 和 `vector2` 相加。 使用圆括号，你可以使用易读的语法来构建复杂表达式。 这是 `MyBool` 类的定义，其中有方法 `and` 和 `or`：

```
case class MyBool(x: Boolean) {
  def and(that: MyBool): MyBool = if (x) that else this
  def or(that: MyBool): MyBool = if (x) this else that
  def negate: MyBool = MyBool(!x)
}
```

现在可以使用 `and` 和 `or` 作为中缀运算符：

```
def not(x: MyBool) = x.negate
def xor(x: MyBool, y: MyBool) = (x or y) and not(x and y)
```

这有助于让方法 `xor` 的定义更具可读性。

优先级

当一个表达式使用多个运算符时，将根据运算符的第一个字符来评估优先级：

```
(characters not shown below)
* / %
+ -
:
= !
< >
&
^
|
(all letters)
```

这也适用于你自定义的方法。 例如，以下表达式：

```
a + b ^? c ?^ d less a ==> b | c
```

等价于

```
((a + b) ^? (c ?^ d)) less ((a ==> b) | c)
```

`?^` 具有最高优先级，因为它以字符 `?` 开头。 `+` 具有第二高的优先级，然后依次是 `==>`， `^?`，`|`， 和 `less`。

传名参数

传名参数 仅在被使用时触发实际参数的求值运算。 它们与 *传值参数* 正好相反。 要将一个参数变为传名参数，只需在它的类型前加上 `=>`。

```
def calculate(input: => Int) = input * 37
```

传名参数的优点是，如果它们在函数体中未被使用，则不会对它们进行求值。 另一方面，传值参数的优点是它们仅被计算一次。 以下是我们如何实现一个 while 循环的例子：

```
def whileLoop(condition: => Boolean)(body: => Unit): Unit =
  if (condition) {
    body
    whileLoop(condition)(body)
  }

var i = 2

whileLoop (i > 0) {
  println(i)
  i -= 1
}  // prints 2 1
```

方法 `whileLoop` 使用多个参数列表来分别获取循环条件和循环体。 如果 `condition` 为 true，则执行 `body`，然后对 whileLoop 进行递归调用。 如果 `condition` 为 false，则永远不会计算 body，因为我们在 `body` 的类型前加上了 `=>`。

现在当我们传递 `i > 0` 作为我们的 `condition` 并且 `println(i); i-= 1` 作为 `body` 时，它表现得像许多语言中的标准 while 循环。

如果参数是计算密集型或长时间运行的代码块，如获取 URL，这种延迟计算参数直到它被使用时才计算的能力可以帮助提高性能。

注解

注解将元信息与定义相关联。 例如，方法之前的注解 `@deprecated` 会导致编译器在该方法被使用时打印警告信息。

```
object DeprecationDemo extends App {
  @deprecated("deprecation message", "release # which deprecates method")
  def hello = "hola"

  hello
}
```

这个程序可以编译，但编译器将打印一个警告信息: “there was one deprecation warning”。

注解作用于其后的第一个定义或声明。 在定义和声明之前可以有多个注解。 这些注解的顺序并不重要。

确保编码正确性的注解

如果不满足条件，某些注解实际上会导致编译失败。 例如，注解 `@tailrec` 确保方法是 尾递归。 尾递归可以保持内存需求不变。 以下是它在计算阶乘的方法中的用法：

```
import scala.annotation.tailrec

def factorial(x: Int): Int = {

  @tailrec
  def factorialHelper(x: Int, accumulator: Int): Int = {
    if (x == 1) accumulator else factorialHelper(x - 1, accumulator * x)
  }
  factorialHelper(x, 1)
}
```

方法 `factorialHelper` 使用注解 `@tailrec` 确保方法确实是尾递归的。 如果我们将方法 `factorialHelper` 的实现改为以下内容，它将编译失败：

```
import scala.annotation.tailrec

def factorial(x: Int): Int = {
  @tailrec
  def factorialHelper(x: Int): Int = {
    if (x == 1) 1 else x * factorialHelper(x - 1)
  }
  factorialHelper(x)
}
```

我们将得到一个错误信息 “Recursive call not in tail position”.

影响代码生成的注解

像 `@inline` 这样的注解会影响生成的代码(即你的 jar 文件可能与你没有使用注解时有不同的字节)。 内联表示在调用点插入被调用方法体中的代码。 生成的字节码更长，但有希望能运行得更快。 使用注解 `@inline` 并不能确保方法内联，当且仅当满足某些生成代码大小的启发式算法时，它才会触发编译器执行此操作。

Java 注解

在编写与 Java 互操作的 Scala 代码时，注解语法中存在一些差异需要注意。 **注意：** 确保你在开启 `-target:jvm-1.8` 选项时使用 Java 注解。

Java 注解有用户自定义元数据的形式，参考 **annotations** 。 注解的一个关键特性是它们依赖于指定 name-value 对来初始化它们的元素。 例如，如果我们需要一个注解来跟踪某个类的来源，我们可以将其定义为

```
@interface Source {
  public String URL();
  public String mail();
}
```

并且按如下方式使用它

```
@Source(URL = "https://coders.com/",
        mail = "support@coders.com")
public class MyClass extends HisClass ...
```

Scala 中的注解应用看起来像构造函数调用，要实例化 Java 注解，必须使用命名参数：

```
@Source(URL = "https://coders.com/",
        mail = "support@coders.com")
class MyScalaClass ...
```

如果注解只包含一个元素(没有默认值)，则此语法非常繁琐，因此，按照惯例，如果将元素名称指定为 `value`，则可以使用类似构造函数的语法在 Java 中应用它：

```
@interface SourceURL {
    public String value();
    public String mail() default "";
}
```

然后按如下方式使用

```
@SourceURL("https://coders.com/")
public class MyClass extends HisClass ...
```

在这种情况下， Scala 提供了相同的可能性

```
@SourceURL("https://coders.com/")
class MyScalaClass ...
```

`mail` 元素在定义时设有默认值，因此我们不需要显式地为它提供值。 但是，如果我们需要显示地提供值，我们则不能在 Java 中混合使用这两种方式：

```
@SourceURL(value = "https://coders.com/",
           mail = "support@coders.com")
public class MyClass extends HisClass ...
```

Scala 在这方面提供了更大的灵活性

```
@SourceURL("https://coders.com/",
           mail = "support@coders.com")
class MyScalaClass ...
```

包和导入

包和导入

Scala 使用包来创建命名空间，从而允许你创建模块化程序。

创建包

通过在 Scala 文件的头部声明一个或多个包名称来创建包。

```
package users

class User
```

一个惯例是将包命名为与包含 Scala 文件的目录名相同。 但是，Scala 并未对文件布局作任何限制。 在一个 sbt 工程中， `package users` 的目录结构可能如下所示：

```
- ExampleProject
  - build.sbt
  - project
  - src
    - main
      - scala
        - users
          User.scala
          UserProfile.scala
          UserPreferences.scala
    - test
```

注意 `users` 目录是包含在 `scala` 目录中的，该包中包含有多个 Scala 文件。 包中的每个 Scala 文件都可以具有相同的包声明。 声明包的另一种方式是使用大括号：

```
package users {
  package administrators {
    class NormalUser
  }
  package normalusers {
    class NormalUser
  }
}
```

如你所见，这允许包嵌套并提供了对范围和封装的更好控制。

包名称应全部为小写，如果代码是在拥有独立网站的组织内开发的，则应采用以下的约定格式：`<top-level-domain>.<domain-name>.<project-name>`。 例如，如果 Google 有一个名为 `SelfDrivingCar` 的项目，则包名称将如下所示：

```
package com.google.selfdrivingcar.camera

class Lens
```

这可以对应于以下目录结构：
`SelfDrivingCar/src/main/scala/com/google/selfdrivingcar/camera/Lens.scala`

导入

`import` 语句用于导入其他包中的成员（类，特质，函数等）。 使用相同包的成员不需要 `import` 语句。 导入语句可以有选择性：

```
import users._  // 导入包 users 中的所有成员
import users.User  // 导入类 User
import users.{User, UserPreferences}  // 仅导入选择的成员
import users.{UserPreferences => UPrefs}  // 导入类并且设置别名
```

Scala 不同于 Java 的一点是 Scala 可以在任何地方使用导入：

```
def sqrtplus1(x: Int) = {
  import scala.math.sqrt
  sqrt(x) + 1.0
}
```

如果存在命名冲突并且你需要从项目的根目录导入，请在包名称前加上 `_root_`：

```
package accounts

import _root_.users._
```

注意：包 `scala` 和 `java.lang` 以及 `object Predef` 是默认导入的。

包对象

Scala 提供包对象作为在整个包中方便的共享使用的容器。

包对象中可以定义任何内容，而不仅仅是变量和方法。 例如，包对象经常用于保存包级作用域的类型别名和隐式转换。 包对象甚至可以继承 Scala 的类和特质。

按照惯例，包对象的代码通常放在名为 `package.scala` 的源文件中。

每个包都允许有一个包对象。 在包对象中的任何定义都被认为是包自身的成员。

看下例。 假设有一个类 `Fruit` 和三个 `Fruit` 对象在包 `gardening.fruits` 中；

```
// in file gardening/fruits/Fruit.scala
package gardening.fruits

case class Fruit(name: String, color: String)
object Apple extends Fruit("Apple", "green")
object Plum extends Fruit("Plum", "blue")
object Banana extends Fruit("Banana", "yellow")
```

现在假设你要将变量 `planted` 和方法 `showFruit` 直接放入包 `gardening` 中。 下面是具体做法：

```
// in file gardening/fruits/package.scala
package gardening
package object fruits {
  val planted = List(Apple, Plum, Banana)
  def showFruit(fruit: Fruit): Unit = {
    println(s"${fruit.name}s are ${fruit.color}")
  }
}
```

作为一个使用范例，下例中的对象 `PrintPlanted` 用导入类 `Fruit` 相同的方式来导入 `planted` 和 `showFruit`，在导入包 `gardening.fruits` 时使用通配符：

```
// in file PrintPlanted.scala
import gardening.fruits._
object PrintPlanted {
  def main(args: Array[String]): Unit = {
    for (fruit <- planted) {
      showFruit(fruit)
    }
  }
}
```

包对象与其他对象类似，这意味着你可以使用继承来构建它们。 例如，一个包对象可能会混入多个特质：

```
package object fruits extends FruitAliases with FruitHelpers {
  // helpers and variables follows here
}
```