EPFL 编程方法实验室

Scala 语言规范

版本：2.7

原作：马丁. 奥德赛 翻译：高德 赵炜

2010-7-20

目录

前言

Scala 是一门类 Java 的编程语言，它结合了面向对象编程和函数式编程。Scala 是纯面向对象的，每个值都是一个对象，对象的类型和行为由类定义，不同的类可以通过混入(mixin)的方式组合在一起。Scala 的设计目的是要和两种主流面向对象编程语言 Java 和 C#实现无缝互操作，这两种主流语言都非纯面向对象。

Scala 也是一门函数式变成语言，每个函数都是一个值，原生支持嵌套函数定义和高阶函数。Scala 也支持一种通用形式的模式匹配，模式匹配用来操作代数式类型，在很多函数式语言中都有实现。

Scala 被设计用来和 Java 无缝互操作（另一个修改的 Scala 实现可以工作在.NET 上）。Scala 类可以调用 Java 方法，创建 Java 对象，继承 Java 类和实现 Java 接口。这些都不需要额外的接口定义或者胶合代码。

Scala 始于 2001 年，由洛桑联邦理工学院(EPFL)的编程方法实验室研发。2003 年 11 月发布 1.0 版本，本书描述的是 2006 年 3 月发布的第二版，作为语言定义和一些核心库模块的参考手册，本书的目的不是教授 Scala 语言或它的概念，这些可以参考其他文档[Oa04, Ode06, OZ05b, OCRZ03, OZ05a]。

Scala 是很多人共同努力的结果。1.0 版的设计和实现由 Philippe Altherr, Vincent Cremet, Gilles Dubo chet, Burak Emir, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Sten man, Matthias Zenger 和本书作者完成；Iulian Dragos, Gilles Dubochet, Philipp Haller, Sean McDirmid, Lex Spoon 和 Geoffrey Washburn 加入了第二版语言和工具的研发。通过参与富有活力和灵感的讨论，并对本书的旧版提出意见，Gilad Bracha, Craig Cham bers, Erik Ernst, Matthias Felleisen, Shriram Krishnamurti, Gary Leavens, Sebastian Maneth, Erik Meijer, Klaus Ostermann, Didier Rémy, Mads Torgersen, 和 Philip Wadler 帮助形成了语言的设计。还有 Scala 邮件列表上的贡献者，他们给予了非常有用的回馈，帮助我们改进语言和工具。

1. 词法

Scala 程序使用的字符集是 Unicode 的基本多文种平面字符集；目前不支持 Unicode 中增补的字符。本章定义了 Scala 词法的两种模式：Scala 模式与 XML 模式。如果没有特别说明，以下对 Scala 符号的描述均指 Scala 模式，常量字符‘c’指 ASCII 段\u0000-\u007F。

在 Scala 模式中，十六进制 Unicode 转义字符会被对应的 Unicode 字符替换。

```
UnicodeEscape ::= \{\\}u{u} hexDigit hexDigit hexDigit hexDigit
hexDigit      ::= ‘0’|...|‘9’|‘A’|...|‘F’|...|‘a’|...|‘f’|
```

符号由下面几类字符构成(括号中是 Unicode 通用类别)：

1. 空白字符。\u0020|\u0009| \u000D|\u000A
2. 字母，包括小写 (Ll)，大写 (Lu)，词首字母大写 (Lt)，其他 (Lo)，数字 (Nl)，以及\u0024 ’$’ 和 \u005F ‘_’，这两个字母归类为大写字母
3. 数字 ‘0’|...|‘9’
4. 括号 ‘(’ | ‘)’ | ‘[’ | ‘]’ | ‘{’ | ‘}’。
5. 分隔符 ‘`’ | ‘’’ | ‘”’ | ‘.’ | ‘;’ | ‘,’。
6. 算符字符。由所有的没有包括在以上分类中的可打印 ASCII 字符\u0020-\u007F，数学符号(Sm)以及其他符号(So)构成

1.1. 标识符

语法：

```
op          ::=  opchar {opchar}
varid       ::=  lower idrest
plainid     ::=  upper idrest
             | varid
             | op
id          ::= plainid
             | ‘\’’ stringLit ‘\’’
idrest      ::=  {letter | digit}[‘_’ op]
```

有三种方法可以构造一个标识符。第一，首字符是字母，后续字符是任意字母和数字。这种标识符还可后接下划线’_’，然后是任意字母和数字。第二，首字符是算符字符，后续字符是任意算符字符。这两种形式是普通标识符。最后，标识符可以是由反引号’`’括起来的任意字符串(宿主系统可能会对字符串和合法性有些限制)。这种标识符可以由除了反引号的任意字符构成。

按惯例，标识符符合最长匹配原则。例如：

```
Big_bob++='def'
```

可以分解为三个标识符 big_bob,++=,和 def。变量标识符(varid，小写字母开头)和常量标识符(没有小写字母开头的限制)的模式匹配规则有所不同。

以下命名是保留字，不能用作词法标识符的语法类 id。

abstract	**case**	**catch**	**class**	**def**
do	**else**	**extends**	**false**	**final**
finally	**for**	**forSome**	**if**	**implicit**
import	**lazy**	**match**	**new**	**null**
object	**override**	**package**	**private**	**protected**
requires	**return**	**sealed**	**super**	**this**
throw	**trait**	**try**	**true**	**type**
val	**var**	**while**	**with**	**yield**
_	:	=	=>	<-	<:	<%	>:	#	@

Unicode 算符\u21D2 '⇒' 和 \u2190 '←'以及它们的 ASCII 对应'=>'也是保留字

示例 1.1.1 以下是一些标识符的示例

```
x          Object      maxIndex        p2p     empty_?
+          `yield` αρετη           _y      dot_product_*
__system   _MAX_LEN_
```

示例 1.1.2 反引号括起来的字符串是那些 Scala 中是保留字的 Java 中标识符的一个方法。例如，在 Scala 中 Thread.yield()是非法的，因为 **yield** 是保留字。但是可以这样调用：

```
Thread.`yield`()
```

1.2. 换行字符

语法：

```
semi    ::= ';' | nl{nl}
```

Scala 是一个基于行的语言。分号和换行均可作为语句的结束。如果换行满足以下三个条件则会被认为是一个特殊符号'nl'：

1. 换行之前的符号是一个语句的结束
2. 换行之后的符号是一个语句的开始
3. 符号处在一个允许多语句的区域中

可以作为语句结束的符号是：常量，标识符，保留字以及以下的分隔符：

this	**null**	**true**	**false**	**return**	**type**	<xml-start>
-	)	]	}			

可以作为语句开始的符号是除了以下分隔符及保留字之外的所有 Scala 符号：

```
catch  else   extends finally forSome match   requires
with   yield  ,   .   ;   :   _   =   =>  <-  <:  <%  >:  #   [   )
]   }
```

符号 **case** 只有在 **class** 或者 **object** 符号之前才可以作为语句开始。

多行语句许可的条件：

1．整个 Scala 源文件中，除了换行被禁止的嵌套区域

2．在匹配的{与}之间，除了换行被禁止的嵌套区域

多行语句在以下区域被禁止：

1．在匹配的(与)之间，除了换行被允许的嵌套区域。

2．在匹配的[与]之间，除了换行被允许的嵌套区域。

3．在 **case** 符号以及与其匹配的=>符号之间，除了换行被允许的嵌套区域。

4．XML 模式下的区域(§1.5)。

注意在 XML 中大括号{..}被转义，字符串并不是符号。因此当换行被允许时不要关闭区域。

一般地，即使连续的两个非换行符号中有多行，也只会插入一个 nl 符号。然而，如果两个符号被至少一个空行(行中没有可打印字符)分隔开，那么两个符号中就会插入两个 nl 符号。

Scala 语法(全文见附录 A)允许可选的 nl 符号，但分号不在此列。这样在某些位置换行并不会结束一个表达式或语句。这些位置如下所列：

以下位置允许多个换行符号(换了分号是不行地)：

- 在条件表达式(§6.16) 或 while 循环(§6.17)的条件及下一个表达式间
- For 循环(§6.19)中计数器及下一个表达式间
- 类型定义或声明中，在开始的 **type** 关键字之后

以下位置允许单个换行：

- 在一个是当前语句或表达式的合法继续的大括号”{”前
- 如果下行的第一个符号是一个表达式的开始(§6.12)，本行的中缀算符之后
- 在一个参数子句前(§4.6)
- 在一个标注(§11)之后

示例 1.2.1 以下是跨两行的四个合法语句。两行间的换行符号并未作为语句结束。

```
if(x > 0)
  x = x - 1

while(x > 0)
   x = x / 2

for(x <- 1 to 10)
   println(x)

type
IntList = List[Int]
```

示例 1.2.2 以下代码定义了一个匿名类

```
new Iterator[Int]
{
  private var x = 0
  def hasNext = true
  def next = { x += 1; x }
}
```

加一个换行后，同样的代码就成了一个对象创建和一个局部代码块

```
new Iterator[Int]

{
  private var x = 0
  def hasNext = true
  def next = { x += 1; x }
}
```

示例 1.2.3 以下代码定义了一个表达式：

```
x < 0 ||
x > 10
```

加一个换行后就成了两个表达式：

```
x < 0 ||

x > 10
```

示例 1.2.4 以下代码定义了一个单一的柯里化的函数：

```
def func(x: Int)
        (y: Int) = x + y
```

加一个换行后，同样的代码就成了一个抽象函数和一个非法语句

```
def func(x: Int)

        (y: Int) = x + y
```

示例 1.2.5 以下代码是一个加了标注的定义：

```
@serializable
protected class Data{...}
```

加一个换行后，同样的代码就成了一个属性标记和一个单独的语句(实际上是非法的)

```
@serializable

protected class Data{...}
```

1.3. 字面值

字面值包括整数，浮点数，字符，布尔值，记号，字符串。这些字面值的语法均和 Java 中的字面值一致。

语法：

```
Literal ::= ['-'] integerLiteral
        | ['-'] floatingPointLiteral
        | booleanLiteral
        | characterLiteral
        | stringLiteral
        | symbolLiteral
        | 'null'
```

1.3.1. 整型字面值

语法：

```
integerLiteral ::= (decimalNumeral | hexNumeral | octalNumeral)['L'|'l']
decimalNumeral ::= '0' | nonZeroDigit {digit}
hexNumeral     ::= '0''x' hexDigit {hexDigit}
octalNumeral   ::= '0' octalDigit {octalDigit}
digit          ::= '0' | nonZeroDigit
nonZeroDigit   ::= '1' | ... | '9'
octalDigit     ::= '0' | ... | '7'
```

整型字面值通常表示 Int 型，或者后面加上 L 或 l 表示 Long 型。Int 的值的范围是 -2^{31} 到 $2^{31}-1$ 间的整数，包含边界值。Long 的值的范围是 -2^{63} 到 $2^{63}-1$ 间的整数，包含边界值。整型字面值的值超出以上范围就会导致编译错误。

如果一个字面值在表达式中期望的类型 pt(§6.1)是 Byte, Short 或者 Char 中的一个，并且整数的值符合该类型的值的范围，那么这个数值就会被转为 pt 类型，这个字面值的类型也是 pt。数值范围如下所示：

Byte	-2^7 到 2^7-1
Short	-2^{15} 到 $2^{15}-1$
Char	0 到 $2^{16}-1$

示例 1.3.1 以下是一些整型字面值：

```
0      21      0xFFFFFFFF  0777L
```

1.3.2. 浮点型字面值

语法：

```
floatingPointLiteral  ::= digit { digit } '.' { digit } [ exponentPart ]
                          [ floatType ]
                        | '.' digit { digit } exponentPart [ floatType ]
                        |digit{digit}exponentPart [ floatType]
```

```
                        | digit { digit } [ exponentPart ] floatType
exponentPart            ::= ('E' | 'e')['+' | '-']digit{digit}
floatType               ::= 'F' | 'f' | 'D' | 'd'
```

如果浮点数字面值的后缀是 F 或者 f，那么这个字面值的类型是 Float，否则就是 Double。Float 类型包括所有 IEEE 754 32 位单精度二进制浮点数值，Double 类型包括所有 IEEE 754 64 位双精度二进制浮点数值。

如果程序中浮点数字面值后面跟一个字母开头的符号，那么这两者之间应当至少有一个空白字符。

示例 1.3.2 以下是一些浮点型字面值：

```
 0.0    1e30f       3.14159f        1.0e-100        .1
```

示例 1.3.3 短语'1.toString'将被解析为三个符号：'1'，'.'和'toString'。但是如果在句点后插入一个空格，短语'1. toString'就会被解析为一个浮点数'1.'和一个标识符'toString'。

1.3.3. 布尔型字面值

语法：

```
booleanLiteral  ::= 'true' | 'false'
```

布尔型字面值 **true** 和 **false** 是 Boolean 类型的成员

1.3.4. 字符型字面值

语法：

```
characterLiteral    ::= '\'' printableChar '\''
                      | '\'' charEscapeSeq '\''
```

字符型字面值就是单引号括起来的单个字符。字符可以是可打印 unicode 字符或者由一个转义序列(§1.3.6)描述的 unicode 字符。

示例 1.3.4 以下是一些字符型字面值：

```
 'a'    '\u0041'        '\n'        '\t'
```

注意'\u000A'不是一个合法的字符常数，因为在处理字面值前已经完成了 Unicode 转换，而 Unicode 字符\u000A(换行)不是一个可打印字符。可以使用转义序列'\n'或八进制转义'\12'来表示一个换行字符(§1.3.6)。

1.3.5. 字符串字面值

语法：

```
stringLiteral   ::= '\"' {stringElement} '\"'
stringElement   ::= printableCharNoDoubleQuote | charEscapeSeq
```

字符串字面值是由双引号括起来的字符序列。字符必须是可打印 unicode 字符或者转义序列(§1.3.6)。如果一个字符串字面值包括双引号，那么这个双引号必须用转义字符，比如：\"。字符串字面值的值是类 String 的一个实例。

示例 1.3.5 以下是一些字符串字面值

```
"Hello,\nWorld!"
"This string contains a \" character."
```

多行字符串字面值

语法：

```
stringLiteral   ::= '"""' multiLineChars '"""'
multiLineChars  ::= {['"'][''"'] charNoDoubleQuote}
```

多行字符串字面值是由三个双引号括起来的字符序列"""..."""。字符序列是除了三个双引号之外的任意字符序列。字符不一定必须是可打印的；换行或者其他控制字符也是可以的。Unicode 转义序列也可以，不过在(§1.3.6)中定义的转义序列不会被解析。

示例 1.3.6 以下是一个多行字符串字面值：

```
"""the present string
   spans three
   lines."""
```

以上语句会产生如下字符串：

```
the present string
   spans three
   lines.
```

Scala 库里包括一个工具方法 stripMargin，可以用来去掉多行字符串行首的空格。表达式：

```
"""the present string
   spans three
   lines.""".stripMargin
```

的值为：

```
the present string
spans three
lines.
```

stripMargin 方法定义在类 scala.runtime.RichString。由于有预定义的从 String 到 RichString 的隐式转换，因此这个方法可以应用到所有的字符串。

1.3.6. 转义序列

在字符串或字符字面值中可以有以下转义序列：

```
\b          \u0008:退格 BS
\t          \u0090:水平制表符 HT
```

```
\n          \u000a:换行 LF
\f          \u000c: 格式进纸 FF
\r          \u000d:回车 CR
\"          \u0022:双引号 "
\'          \u0027:单引号 '
\\          \u005c:反斜线 \
```

0 到 255 间的 Unicode 字符可以用一个八进制转义序列来表示，即反斜线'\'后跟最多三个八进制。

在字符或字符串中，反斜线和后面的字符序列不能构成一个合法的转义序列将会导致编译错误。

1.3.7. 记号字面值

语法：

```
symbolLiteral   ::= '' idrest
```

记号字面值'x 是表达式 scala.Symbol("x")的简写形式。Symbol 是一个 case 类(§5.3.2)，定义如下：

```
package scala
final case class Symbol private (name: String) {
 override def toString: String = "'" + name
}
```

Symbol 的伴随实例的 apply 方法中缓存了一个到 Symbol 的弱引用，因此同样的记号字面值是引用相等的。

1.4. 空白与注释

符号可由空白字符或注释分隔开。注释有两种格式：

单行注释是由//开始直到行尾的字符序列

多行注释是在/*和*/之间的字符序列。多行注释可以嵌套，但是必须合理的嵌套。因此像/* /* */这样的注释是非法的，因为有一个没有结束的注释。

1.5. XML 模式

为了允许包含 XML 片段字面值。当遇到左尖括号'<'时，在以下情况下词法分析就会从 Scala 模式切换到 XML 模式：'<'前必须是空白，左括号或者左大括号，'<'后必须跟一个 XML 命名。

语法：

```
(whitespace | '(' | '{' } '<' (XNameStart | '!' | '?')
XNameStart ::= '_' | BaseChar | Ideographic (和 W3C XML 一样, 但没有':')
```

扫描器遇到以下条件之一时将会从 XML 模式切换到 Scala 模式：

- 由'<'开始的 XML 表达式或模式已被成功解析。
- 解析器遇到一个内嵌的 Scala 表达式或模式强制扫描器返回正常模式，直到 Scala 表达式或模式被成功解析。在这种情况下，由于代码和 XML 片段可以嵌套，解析器将会用一个堆栈来储存嵌套的 XML 和 Scala 表达式。

注意在 XML 模式中不会生成 Scala 的符号，注释会解析为文本。

示例 1.5.1 以下定义了一个值，包括一个 XML 字面值，内嵌了两个 Scala 表达式

```
val b = <book>
          <title>The Scala Language Specification</title>
          <version>{scalaBook.version}</version>
          <authors>{scalaBook.authors.mkList("","", "\", "")}</authors>
        </book>
```

2. 标识符，命名和域

在 Scala 中，命名用来表示类型，值，方法以及类，这些统称为*实体*。命名在局部定义与声明(§4)，继承(§5.1.3)，import 子句，package 子句中存在，这些可以统称为*绑定*。

绑定有优先级，定义(局部或继承)有最高的优先级，然后是显式 import，然后是通配符 import，然后是包成员，是最低的优先级。

有两种不同的命名空间，一个是类型(§3)，一个是术语(§6)。同样的命名可以表示类型或术语，这要看命名应用所在的上下文。

绑定有一个域，在此域中用单一命名定义的实体可以用一个简单名称来访问。域可以嵌套。内部域中的绑定将会*遮盖*同一域中低优先级的绑定，或者外部域中低优先级或同优先级的绑定。

注意遮盖只是偏序关系。在下面情况中：

```
val x = 1;
{ import p.x;
  x}
```

x 的绑定并没有互相遮盖。因此第三行中对 x 的引用的含义将是不明确的。

对一个未限定的（类型或术语）标识符 x 的引用在以下条件下可以被单一绑定：

- 在同一命名空间中用命名 x 定义一个实体作为标识符
- 在此命名空间中遮盖所有的其他定义命名 x 的实体绑定

如果没有这样的绑定将会导致错误。如果 x 由一个 import 子句绑定，那么简单命名 x 将等价于由 import 子句映射所限定的命名。如果 x 由一个定义或声明绑定，那么 x 将指代由该绑定引入的实体。在此情况下，x 的类型即是引用的实体的类型。

示例 2.0.2 以下是包 P 和 Q 中两个名为 X 的对象的定义：

```
package P {
  object X { val x = 1; val y = 2 }
}

package Q {
  object X { val x = true; val y = "" }
}
```

以下程序示意了它们间不同的绑定及优先级。

```
package P {                          //'X'由 package 子句绑定
  import Console._                   //'println'由通配符 import 绑定
  object A {
```

```
  println("L4: "+X)                    //这里的'X'指'P.X'
  object B {
    import Q._{                       //'X'由通配符 import 绑定
    println("L7: "+X)                 //这里的'X'指'Q.X'
    import X._                        //'x'和'y'由通配符 import 绑定
    println("L8: "+x)                 //这里的'x'指'Q.X.x'
    object C {
      val x = 3                       //'x'由局部定义绑定
      println("L12: "+x)                   //这里的'x'指常数'3'
      { import Q.X._                      //'x'和'y'由通配符 import 绑定
//      println("L14: "+x)               //这里到'x'的引用指代不明确
        import X.y                      //'y'由显式 import 绑定
        println("L16: "+y)              //这里的'y'指'Q.X.y'
        { val x = "abc"                 //'x'由局部定义绑定
          import P.X._                  //'x'和'y'由通配符 import 绑定
//        println("L19: "+y)             //这里到'y'的引用指代不明确
          println("L20: "+x)             //这里的'x'指字符串"abc"
}}}}}}
```

一个到限定的(类型或术语)标识符 e.x 的引用指在同一个命名空间中 e 的类型 T 的一个名为 x 的成员作为标识符。如果 T 不是值类型(§3.2)将会导致错误。e.x 的类型就是引用的实体 T 的成员的类型。

3. 类型

语法：

```
Type                ::= InfixType ‘=>’ Type
                      | ‘(’[‘=>’ Type] ‘)’ ‘=>’ Type
                      | InfixType [ExistentialClause]
ExistentialClause   ::= ‘forSome’ ‘{‘ ExistentialDc
                          { semi ExistentialDcl} ‘}’
ExistentialDcl      ::= ‘type’ TypeDcl
                      | ‘val’ ValDcl
InfixType           ::= CompoundType {id [nl] CompoundType}
CompoundType        ::= AnnotType {‘with’ AnnotType}[Refinement]
                      | Refinement
AnnotType           ::= SimpleType {Annotation}
SimpleType          ::= SimpleType TypeArgs
                      | SimpleType ‘#’ id
                      | StableId
                      | Path ‘.’ ‘type’
                      | ‘(‘ Types [‘,’] ‘)’
TypeArgs            ::= ‘[‘ Types ‘]’
Types               ::= Type {‘,’ Type}
```

一阶类型和类型构造器(用类型的参数构造类型)是有区别的。一阶类型的一个子集是*值类型*，表示(一阶)值的集合。值类型可以是具体的或者抽象的。

每个具体的值类型可以用一个类类型来表示，比如指向某类[1](§5.3) 的类型指示器(§3.2.3)，或者表示类型交集(可能会加一个修饰(§3.2.7)来限制其成员的类型)的复合类型(§3.2.7)。类型参数(§4.4)和抽象类型绑定(§4.3)引入了抽象值类型。类型中的括号用来建组。

非值类型描述了那些不是值(§3.3)的标识符的属性。例如，一个类型构造器(§3.3.3)并不指明值的类型。然而，当一个类型构造器应用到正确的类型参数上时，就会产生一个可能是值类型的一阶类型。

在 Scala 中，非值类型被间接表述。例：写下一个方法签名来描述一个方法类型，虽然通过它可以得到对应的函数类型(§3.3.1)，但是它本身并不是一个真正的类型。类型构造器是另外一个例子，比如我们可以写 **type** Swap[m[_,_],a,b] = m[b,a]，但是并没有定义直接给出对应的匿名类型函数的语法。

[1] 我们假定对象和包都隐式地定义一个类(与对象或包同名，但是在用户程序中不可访问)

3.1. 路径

语法：

```
Path            ::= StableId
                  | [id '.'] this
StableId        ::= id
                  | Path '.' id
                  | [id '.'] 'super' [ClassQualifier] '.' id
ClassQualifier  ::= '[' id ']'
```

路径不是类型本身，但是它们可以是命名类型的一部分，这个功能是 Scala 类型系统的一个核心角色。

一个路径是下面定义中的一个：

- 空路径 ε（不能在程序中显式地写出来）
- C.**this**，C 是一个类的引用。当在 C 引用范围内时，路径 this 是 C.this 的简写。
- p.x，p 是一个路径，x 是 p 的一个稳定成员。稳定成员是由对象定义或者稳定类型的值定义引入的包或者成员(§3.6)。
- C.**super**.x 或 C.**super**[M].x，C 是一个类的引用，x 是 C 的超类或指定的父类 M 的稳定成员的引用。前缀 **super** 是 C.**super** 的简写，C 是包含引用范围的类的名称。

一个*稳定的标识符*是由标识符结束的一个路径。

3.2. 值类型

Scala 中的每个值都有一个以下格式的类型。

3.2.1. 单例类型

语法：

```
SimpleType  ::= Path '.' type
```

单例类型具有 p.**type** 的形式，p 是一个路径，指向一个期望与 scala.AnyRef 一致(§6.1)的值。类型指一组为 **null** 的或由 p 表示的值。

一个稳定类型指要么是一个单例类型，要么是特征 scala.Singleton 的子类型。

3.2.2. 类型映射

语法：

```
SimpleType  ::= SimpleType '#' id
```

类型映射 T#x 指类型 T 的类型成员 x。如果 x 指向抽象类型成员，那么 T 必须是一个稳定类型(§3.2.1)。

3.2.3. 类型指示

语法：

```
SimpleType  ::= StableId
```

类型指示指一个命名值类型。它可以是简单的或限定的。所有的这些类型指示都是类型映射的简写。

绑定在某类，对象或包 C 上的非限定的类型名 t 是 C.**this**.**type**#t 的简写，除非类型 t 是类型模式(§8.1.2)的一部分。后者中的 t 是 C#t 的简写。如果 t 没有被绑定在某类，对象或包上，那么 t 就是 ε.**type**#t 的简写。

一个限定的类型只是具有 p.t 的形式，p 是一个路径(§3.1)，t 是一个类型名。这个类型指示等价于类型映射 p.**type**#t。

示例 3.2.1 以下是一些类型指示以及扩展。我们假定一个本地类型参数 t，一个值 maintable 具有一个类型成员 Node，以及一个标准类 scala.Int

```
t                       ε.type#t
Int                     scala.type#Int
scala.Int               scala.type#Int
data.maintable.Node         data.maintable.type#Node
```

3.2.4. 参数化类型

语法：

```
SimpleType  ::= SimpleType TypeArgs
TypeArgs    ::= '[' Types ']'
```

参数化类型 $T[U_1,...,U_n]$ 包括类型指示 T 以及类型参数 $U_1,...,U_n$，n >=1。T 必须指向一个具有个参数类型 $a_1,...,a_n$ 的参数构造方法。

类型参数具有下界 $L_1,...,L_n$ 和上界 $U_1,...,U_n$。参数化类型必须保证每个参数与其边界一致：$\sigma L_i <: T_i <: \sigma U_i$，这里 σ 表示 $[a_1:=T_1,...,a_n:=T_n]$。

示例 3.2.2 以下是一些类型定义(部分)：

```
class TreeMap[A <: Comparable[A], B]{ ... }
class List[A] { ... }
class I extends Comparable[I] { ... }
```

以下是正确的参数化类型：

```
TreeMap[I, String]
List[I]
List[List[Boolean]]
```

示例 3.2.3 使用示例 3.2.2 中的类型定义，以下是错误的参数化类型：

```
TreeMap[I]                      //错误的参数个数
TreeMap[List[I], Boolean]       //类型参数越界
```

3.2.5. 元组类型

语法：

```
SimpleType  ::=  '(' Types [','] ')'
```

元组类型$(T_1,...,T_n)$是类 scala.Tuple*n*$[T_1,...,T_n]$(n>=2)的别名形式。此类型可以在结尾处有个额外的逗号，例：$(T_1,...,T_n,)$。

元组类是 case 类，其字段可以用选择器_1,...,_n 来访问。在对应的 Product 特征中有他们的抽象函数。这些元组类以及 Product 特征都是标准 Scala 类库的一部分，其形式如下：

```
case class Tuplen[+T1,...,+Tn](_1: T1,...,_n: Tn) extends
Productn[T1,...,Tn]{}
trait Productn[+T1,...,+Tn]{
  override def arity = n
  def _1: T1
  ...
  def _n: Tn
}
```

3.2.6. 标注类型

语法：

```
AnnotType   ::= SimpleType {Annotation}
```

标注类型 T $a_1,...,a_n$ 就是给类型 T 加上标注 $a_1,...,a_n$.

3.2.7. 复合类型

语法：

```
CompoundType    ::= AnnotType {'with' AnnotType} [Refinement]
                  | Refinement
Refinement      ::= [nl] '{' RefineStat {semi RefineStat} '}'
RefineStat      ::= Dcl
                  | 'type' TypeDef
                  |
```

复合类型 T_1 **with ... with** T_n {R}指一个拥有 $T_1,...,T_n$ 类型中的成员以及修饰{R}的对象。如果对象中有声明或定义覆盖了成分类型 $T_1,...,T_n$ 中的声明或定义，就会应用通常的覆盖规则(§5.1.4)；否则这个声明或定义就将是所谓的“结构化的”[2]。在一个结构化修饰的方法声明中，任何值参数的类型只是指修饰内部包含的类型参数或抽象类型。也就是它必须指代一个函数本身的类型参数，或者在修饰内部的一个类型定义。该

[2] 到一个结构化定义的成员的引用(方法调用或访问值或变量)可能产生比等价的非结构化成员慢的多的代码。

限制并不对函数的返回类型起作用。

如果没有修饰，那么默认就会添加一个空的修饰，例：T_1 **with ... with** T_n 即是 T_1 **with ... with** T_n {}的简写。

一个复合类型可以只有修饰{R}而没有前面的成分类型。这样的类型等价于AnyRef{R}。

示例 3.2.4 以下是如何声明以及使用参数类型包含结构化声明修饰的函数。

```
case class Bird (val name: String) extends Object {
  def fly(height: Int) = ...
 ...
}
case class Plane (val callsign: String) extends Object {
  def fly(height: Int) = ...
 ...
}
def takeoff(
     runway: Int,
     r: { val callsign: String; def fly(height: Int) }) = {
  tower.print(r.callsign + " requests take-off on runway " + runway)
  tower.read(r.callsign + " is clear for take-off")
  r.fly(1000)
}
val bird = new Bird("Polly the parrot"){ val callsign = name }
val a380 = new Plane("TZ-987")
takeoff(42, bird)
takeoff(89, a380)
```

虽然 Bird 和 Plane 没有除了 Object 之外的任何父类，用结构化声明修饰的函数 takeoff 的参数 r 却可以接受任何声明了值 callsign 以及函数 fly 的对象。

3.2.8. 中缀类型

语法：

```
InfixType   ::= CompoundType {id [nl] CompoundType}
```

中缀类型 T_1 op T_2 由一个中缀算符 op 应用到两个操作数 T_1 和 T_2 上得来。这个类型等价于类型应用 op[T_1, T_2]。中缀算符可以是除*之外的任意的标识符，因为*被保留作为重复参数类型的后缀(§4.6.2)。

所有类型的中缀算符拥有同样的优先级；因此必须用括号来改变顺序。类型算符的结合性(§6.12)由其形式来决定：由':'结尾的类型算符是右结合的，其他的是左结合的。

在一个连续的类型中缀运算 t_0 op_1 t_1 op_2...op_n t_n 里，所有的算符 op_1,...,op_n 必须具有相同的结合性。如果都是左结合的，该序列就被解析为 (...(t_0 op_1 t_1) op_2...) op_n t_n，否则会被解析为 t_0 op_1 (t_1 op_2 (...op_n t_n)..)。

3.2.9. 函数类型

语法：

```
Type    ::= InfixType '=>' Type
        | '(' ['=>' Type] ')' '=>' Type
```

类型 $(T_1,...,T_n)$ => U 表示那些参数类型为 $T_1,...,T_n$，并产生一个类型为 U 的结果的函数。如果只有一个参数类型则(T)=>U 可以简写为 T=>U。类型(=>T)=>U 表示以类型为 T 的传名(§4.6.1)参数并产生类型为 U 的结果。函数类型是右结合的，例：S=>T=>U 等价于 S=>(T=>U)。

函数类型是定义了 apply 函数的类类型的简写。比如 n 型函数类型$(T_1,...,T_n)$ => U 就是类 Function*n*[$T_1,...,T_n$,U]的简写。Scala 库中定义了 n 为 0 至 9 的这些类类型，如下所示：

```
package scala
trait Functionn[-T1,...,-Tn, +R] {
  def apply(x1: T1,...,xn: Tn): R
  override def toString = "<function>"
}
```

因此，函数类型与结果类型是协变(§4.5)的，与参数类型是逆变的。

传名函数类型(=>T)=>U 是类类型 ByNameFunction[T,U]的简写形式，定义如下：

```
package scala
trait ByNameFunction[-T, +R] {
  def apply(x: => T): R
  override def toString = "<function>"
}
```

3.2.10. 既存类型

语法：

```
Type                ::= InfixType ExistentialClauses
ExistentialClauses  ::= 'forSome' '{' ExistentialDcl
                            {semi ExistentialDcl} '}'
ExistentialDcl      ::= 'type' TypeDcl
                    | 'val' ValDcl
```

既存类型具有 T **forSome** {Q}的形式，Q 是一个类型声明的序列(§4.3)。设 $t_1[tps_1]>:L_1<:U_1,...,t_n[tps_n]>:L_n<:U_n$ 是 Q 中声明的类型(任何类型参数部分 $[tps_i]$都可以没有)。每个类型 t_i 的域都包含类型 T 和既存子句 Q。类型变量 t_i 就称为在类型 T **forSome** {Q}中被绑定。在 T 中但是没被绑定的类型变量就被称为在 T 中是*自由*的。

T **forSome** {Q}的类的实例就是类 σT，σ 是 $t_1,...,t_n$ 上的迭代，对于每一个 i，都有 $\sigma L_i<:\sigma t_i<:\sigma U_i$。既存类型 T **forSome**{Q}的值的集合就是所有其类型实例值的集

合的合集。

T **forSome** {Q}的斯科伦化是一个类实例 σT，σ 是[t'_1/t_1,..., t'_n/t_n上的迭代，每个 t'_i 是介于 σL_i 和 σU_i 间的新的抽象类型。

简化规则

既存类型遵循以下四个等效原则：

1. 既存类型中的多个 for 子句可以合并。例 T **forSome** {Q} **forSome**{Q'}等价于 T **forSome** {Q;Q'}
2. 未使用的限定可以去掉。例：T **forSome** {Q;Q'}，但是 Q'中定义的类型没有被 T 或 Q 引用，那么该表达式可等价为 T **forSome** {Q}
3. 空的限定可以丢弃。例：T **forSome**{}等价于 T。
4. 一个既存类型 T **forSome** {Q}，Q 中包含一个子句 **type** t[tps] >: L <: U 等价于类型 T' **forSome** {Q}，T'是将 T 中所有 t 的协变量替换为 U 并且将 T 中所有的 t 的逆变量替换为 L 的结果。

在值上的既存量化

为了语法上的方便，在既存类型上的绑定子句可以包括值声明 **val** x: T。既存类型 T **forSome** { Q; **val** x: S; Q'} 是 T' **forSome** { Q; **type** t <: S **with** Singleton; Q'}的简写形式，此处 t 是一个新的类型名，T'是将 T 中所有 x.**type** 用 t 代替的结果。

既存类型的占位符语法

语法：

```
WildcardType    ::= '_' TypeBounds
```

Scala 支持既存类型的占位符语法。通配符类型的形式为 _>:L<:U。两个边界均可忽略。如果下界>:L 被忽略则用>:scala.Nothing。如果上界<:U 被忽略则用<:scala.Any。通配符类型是既存限定类型变量的简写，既存的限定条件是内涵的。

通配符类型只能作为参数化类型的类型参量出现。设 T=p.c[targs,T,tags']是一个参数化类型，targs,targs'可以为空，T 是一个通配符类型_>:L<:U。那么 T 等价于以下既存类型：

```
p.c[tags,t,tags'] forSome { type t >:L<:U}
```

t 是一个新的类型变量。通配符类型可以作为中缀类型(§3.2.8)，函数类型(§3.2.9)或元组类型(§3.2.5)的一部分出现。它们的扩展也就是等价参数化类型的扩展

示例 3.2.5 假定以下类定义：

```
class Ref[T]
abstract class Outer { type T }
```

以下是一些既存类型的例子：

```
Ref[T] forSome { type T <: java.lang.Number }
Ref[x.T] forSome { val x: Outer }
Ref[x_type # T] forSome { type x_type <: Outer with Singleton }
```

列表中的后两个类型是等价的。使用通配符语法的另一种形式是：

```
Ref[_ <: java.lang.Number]
Ref[(_ <: Outer with Singleton)# T]
```

示例 3.2.6 类型 List[List[_]]等价于既存类型：

```
List[List[t] forSome { type t }]
```

示例 3.2.7 假定有协变类型：

```
class List[+T]
```

类型：

```
List[T] forSome { type T <: java.lang.Number }
```

应用上面的第四条简化规则，上式等价于：

```
List[java.lang.Number] forSome { type T <: java.lang.Number }
```

如果再应用上面的第二和第三条简化规则，上式可化为：

```
List[java.lang.Number]
```

3.2.11. Predef 中定义的原始类型

每个 Scala 程序都默认 import 一个 Predef 对象。该对象定义了一些原始类型做为类类型的别名。数值类型和布尔型有标准的 Scala 类。String 类型与宿主系统的 String 类型一致。在 Java 环境下，Predef 包括以下类型绑定：

```
type byte      = scala.Byte
type short     = scala.Short
type char      = scala.Char
type int       = scala.Int
type long      = scala.Long
type float     = scala.Float
type double    = scala.Double
type Boolean   = scala.Boolean
type String    = java.lang.String
```

3.3. 非值类型

以下类型并不表示值的集合，也并不显式地出现在程序中。它们只以已定义标识符的内部类型而引入。

3.3.1. 方法类型

方法类型在内部表示为(*Ts*)U，(Ts)是一个类型序列(T_1,...,T_n) n>=0，U 是一个(值或者方法)类型。这个类型表示一个命名的方法，其参数的类型是 T_1,...,T_n，返回结果的类型是 U。

方法类型是右结合的，(Ts_1)(Ts_2)U 被处理的方式是(Ts_1)((Ts_2)U)。

一个特例是没有参数的方法类型。可以写为=>T 的形式。无参数方法名称表达式将会在每次名称被引用时求值。

方法类型并不以值的类型的形式存在。如果方法名以值的方式被引用，其类型将会被自动转换为对应的函数类型(§6.25)。

示例 3.3.1 以下声明：

```
def a: Int
def b (x: Int): Boolean
def c (x: Int)(y: String, z: String): String
```

产生以下类型：

```
a: => Int
b: (Int) Boolean
c: (Int)(String, String) String
```

3.3.2. 多态方法类型

多态方法类型在内部表示为[tps]T，[tps]是类型参数部分[a_1 >: L_1 <: U_1,...,a_n >: L_n <: U_n]，n>=0，T 是一个(值或方法)类型。该类型表示一个以 S_1,...,S_n 为类型参量并产生类型为 T 的结果的命名方法，参数类型 S_1,...,S_n 与下界 L_1,...,L_n 和上界 U_1,...,U_n 一致(§3.2.4)。

示例 3.3.2 以下声明：

```
def empty[A]: List[A]
def union[A <: Comparable[A]] (x: Set[A], xs: Set[A]): Set[A]
```

产生如下类型：

```
empty: [A >: Nothing <: Any] List[A]
union: [A >: Nothing <: Comparable[A]] (x: Set[A], xs: Set[A]) Set[A]
```

3.3.3. 类型构造器

类型构造器在内部的表示方法类似于多态方法类型。[+/- a_1 >: L_1 <: U_1,...,+/-a_n >: L_n <: U_n] T 表示一个期望是类型构造器参数(§4.4)或有对应类型参数子句的抽象类型构造器绑定(§4.3)的类型。

示例 3.3.3 以下是类 Iterable[+X]的片段：

```
trait Iterable[+X] {
  def flatMap[newType[+X]<:Iterabe[X], S](f: X => newType[S]): newType[S]
}
```

从概念上来讲，类型构造器 Iterable 是匿名类型[+X] Iterable[X]的名称，在

flatMap 中传递给 newType 类型构造器参数。

3.4. 基本类型和成员定义

类成员的类型取决于成员被引用的方式。主要有三个概念：

1. 类型 T 的基本类型集合
2. 从前缀类型 S 中可见的类 C 中的类型 T
3. 类型 T 的成员绑定集合

以下是这三个概念的详细定义

1. 类 T 的基本类型集合定义如下
   - C 是类型 C 以及其父类型 $T_1,...,T_n$ 的基本类型，同时也是组合类型 T_1 **with ... with** T_n {R}的基本类型。
   - 类型别名的基本类型是别名的类型的基本类型
   - 抽象类型的基本类型是其上界的基本类型
   - 参数化类型 $C[T_1,...,T_n]$的基本类型是类型 C 的基本类型，C 的每一个类型参数 a_i 被对应的参数类型 T_i 代替
   - 单例类型 p.**type** 的基本类型是类型 p 的基本类型
   - 复合类型 T_1 **with ... with** T_n {R}的基本类型是所有 T_i 的基本类型的*缩减合并*。意思是设合集 Φ 为 T_i 的基本类型的合集，如果 Φ 包括同一个类的多个类型实例，比如 $S^i\#C[T^i_1,...,T^i_n](i\in I)$，那么所有的这些实例将会被一个与其他一致的实例代替。如果没有这样一个实例存在就会导致错误。如果存在这样一个缩减合并，那么该集合会产生类类型的集合，不同的类型是不同类的实例。
   - 类型选择 S#T 的基本类型如下确定：如果 T 是一个抽象类或别名，那么前面的子句就会被应用。否则 T 必须是一个定义在某个类 B 中的(可能还是参数化的)类类型。那么 S#T 的基本类型就是从前缀类型 S 中看到的 B 中 T 的基本类型。
   - 既存类型 T **forSome** {Q}的基本类型是所有 S **forSome** {Q}类型，S 是 T 的基本类型
2. 从前缀类型 S 中可见的类 C 中的类型 T 只在以下条件下起作用，前缀类型 S 有一个类型 C 的类型实例作为基本类型，即 $S'\#C[T_1,...,T_n]$。我们有以下定义
   - 如果 S = ε.**type**，那么从 S 看到的 C 中的 T 就是 T 本身
   - 否则，如果 S 是既存类型 S′ **forSome** {Q}，从 S′看 C 中的 T 将会是 T′，那么从 S 看 T 中的 C 将会是 T′ **forSome** {Q}
   - 否则，如果 T 是某类 D 的第 i 个类型参数，那么
     - 如果 S 有基本类型 $D[U_1,...,U_n]$，$[U_1,...,U_n]$是类型参数，那么从 S 中看到的 C 中的 T 就是 U_i
     - 否则，如果 C 定义在类 C′中，那么从 S 中看到的 C 中的 T 与在 S′中看到的 C′中的 T 是一样的
     - 否则，如果 C 不是定义在其他类中，那么从 S 中看到的 C 中的 T 就是 T 本身
   - 否则，如果 T 是某类 D 的单例类型 D.**this.type**，那么
     - 如果 D 是 C 的子类，S 的基本类型中有一个类 D 的类型实例，那么从 S 中看到的 C 中的 T 就是 S

- 否则，如果 C 定义在类 C′中，那么从 S 中看到的 C 中的 T 与 S′中看到的 C′中的 T 相同
- 否则，如果 C 不是定义在其他类中，那么从 S 中看到的 C 中的 T 就是 T 本身

● 如果 T 是其他类型，那么将在所有其类型组件中执行以上描述的映射

如果 T 是一个可能参数化的类类型，T 的类定义在某个类 D 中，S 是某前缀类型，那么“从 S 中看到 T”就是“从 S 中看到 D 中的 T 的简写”。

3. 类型 T 的成员绑定集合是(1)T 的基本类型中存在某类的类型实例和 C 中 d′的定义或声明（d 是将 d′中的类型 T′替换为从 T 中看到的 C 中的 T′得到的结果）等绑定(2)类型修饰的绑定(§3.2.7)(如果有的话)。

类型映射 S#t 的定义就是 S 中类型 t 的成员绑定 d_t。在此情况下，我们可以说 S#t 由 d_t 定义。

3.5. 类型间的关系

类型间有两种关系：

类型恒等	T ≡ U　T 和 U 可以在任何情况下互相替换
一致	T <: U 类型 T 与类型 U 一致

3.5.1. 类型恒等

类型间的恒等(≡)关系是最小的一致性[3]，具有以下特点：

- 如果 t 是一个类型别名的定义 **type** t = T，那么 t 就等价于 T
- 如果路径 p 有一个单例类型 q.**type** 那么 p.**type** ≡ q.**type**
- 如果 O 是一个对象的定义，p 是一个路径，仅包括包或者对象的选择器，并以 O 结束。那么 O.**this.type** ≡ p.**type**
- 两个复合类型(§3.2.7)相等的条件是：他们组件序列元素相等，并且顺序一致，而且修饰也相等。如果两个修饰绑定到同样的命名，并且两个修饰的每个声明的实体的修饰符，类型和边界也相等，那么这两个修饰相等。
- 两个方法类型(§3.3.1)相等的条件是：结果类型相等，参数数目一致，对应的参数类型一致。参数名称不必相等。
- 两个多态方法类型(§3.3.2)相等的条件是：同样数目的类型参数，如果将另一组类型参数重命名为当前一组，得到的类型以及对应的参数类型的上下界也相等。
- 两个既存类型(§3.2.10)相等的条件是：同样数目的量词，如果用另一组类型量词替换当前一组，定量类型以及对应的量词的上下界也相等。
- 两个类型构造器(§3.3.3)相等的条件是：同样数目的类型参数，如果用另一组类型参数替换当前一组，结果类型以及变化，对应的类型参数的上界和下界也是相等的。

3.5.2. 一致性

一致性关系(<:)是符合以下条件的最小的可传递关系：

- 一致性包含相等。如果 T ≡ U 那么 T <: U

[3] 一致性是上下文构成中封闭的等价关系

- 对于任意的值类型 T，有 scala.Nothing <: T <: scala.Any
- 对于任意的类型构造器 T(任意数目的类型参数)，有 scala.Nothing <: T <: scala.Any
- 对于任意的类类型 T，T <: scala.AnyRef，并且不是 T <: scala.NotNull，那么有 scala.Null <: T
- 类型变量或抽象类型 t 与其上界一致，其下界与 t 一致
- 类类型或者参数化类型与其任何基本类型一致
- 单例类型 p.**type** 与路径 p 的类型一致
- 单例类型 p.**type** 与类型 scala.Singleton 一致
- 类型映射 T#t 与 U#t 一致，如果 T 与 U 一致的话
- 参数化类型 $T[T_1,...,T_n]$ 与 $T[U_1,...,U_n]$ 一致的条件是(i=1,...,n)：
  - 如果 T 的第 i 个类型参数声明为协变量，那么 $T_i <: U_i$
  - 如果 T 的第 i 个类型参数声明为逆变量，那么 $U_i <: T_i$
  - 如果 T 的第 i 个类型参数既不是协变量也不是逆变量，那么 $U_i \equiv T_i$
- 复合类型 T_1 **with** ... **with** T_n {R}与其每个组件类型 T_i 一致
- 如果 T <: U_i(i=1,...,n)，对 R 中任一类型或值 x 的绑定 d，T 中存在一个包括 d 的 x 的成员绑定，那么 T 与复合类型 U_1 **with** ... **with** U_n {R}一致
- 如果既存类型 T **forSome** {Q} 的斯科伦化(§3.2.10)与 U 一致，那么该类型与 U 一致
- 如果 T 与 U **forSome** {Q}的某一个类型实例(§3.2.10)一致，那么 T 与既存类型 U **forSome** {Q}一致
- 如果 $T_i \equiv T'_i$(i=1,...,n)，U 与 U′一致，那么方法类型$(T_1,...,T_n)U$ 与 $(T'_1,...,T'_n)U'$一致。
- 多态类型$[a_1 >: L_1 <: U_1,...,a_n >: L_n <: U_n]T$ 与$[a_1 >: L'_1 <: U'_1, ..., a_n >: L'_n <: U'_n]$ T′一致的条件是：假设 $L'_1 <: a_1 <: U'_1$, ... ,$L'_n <: a_n <: U'_n$, T :< T′, $L_i <: L'_i$ $U'_I <: U_i$, i=1,...,n.
- 类型构造器 T 和 T′有类似的规则。可以用类型参数子句$[a_1,...,a_n]$和$[a'_1,...,a'_n]$来区分 T 和 T′。每个 a_i 和 a'_i 可能包括差异标注，高阶类型参数子句和边界。那么，T 与 T′一致的条件就是，任意列表$[t_1,...,t_n]$(包括声明的差异，边界和高阶类型参数子句)是 T′的有效的类型参数，同时也是 T 的有效的类型参数，并且有 $T[t_1,...,t_n] <: T'[t_1,...,t_n]$。这将导致：
  - a_i 的边界必须要比对应的 a'_i 的边界要弱。
  - a_i 的差异必须要与 a'_i 的差异一致，协变与协变一致，逆变与逆变一致，任意差异与无差异一致。
  - 这些限制要递归应用到 a_i 和 a'_i 对应的高阶类型参数子句上.

在以下某个条件下，类类型 C 的复合类型的声明或定义将包括另外一个类类型 C′的符合类型的同名声明：

- 如果 T<:T′，一个值声明或定义定义了一个类型为 T 的命名 x，包括一个值或者方法声明定义了类型为 T′的 x。
- 如果 T<:T′，一个方法声明或定义定义了一个类型为 T 的命名 x，包括一个方法声明定义了类型为 T′的 x
- 如果 T≡T′，类型别名 **type** $t[T_1,...,T_n]$ = T 包括一个类型别名 **type** t[T1,...,Tn] = T′。

- 如果 L′ <: L 且 U <: U′，类型声明 **type** t[T_1,...,T_n] >: L <: U 包括一个类型声明 **type** t[T_1,...,T_n] >: L′ <: U′。
- 如果 L <: t <: U 一个绑定到一个类型名称 t 的类型或类定义包括一个抽象声明 **type** t[T_1,...,T_n] >: L <: U。

<:关系在类型间组成了一种顺序，比如传递和自反。一个类型集合的最小上界与最大下界可以理解为与该顺序有关。

注意：类型集合的最小上界和最小下界不一定总是存在。例如，以下类定义：

```
class A[+T] {}
class B extends A[B]
class C extends A[C]
```

类型 A[Any]，A[A[Any]]，A[A[A[Any]]],...构成了一个 B 和 C 上界的降序序列。最小上界将会是这个序列的极限，这个极限不可能作为一个 Scala 类型存在。由于这种情况通常不大可能探测到，因此如果指定了一个最小上界或最大下界的类型，并且这个边界可能复杂度超过编译器设定的限制[4]的话就可能被 Scala 编译器拒绝。

最小上界和最大下界也可能不唯一。例如 A **with** B 和 B **with** A 都是 A 和 B 的最大下界。如果有多个最小上界或最大下界，Scala 编译器可能会自动选取其中某一个。

3.6. 易变类型

类型易变大致的意思是说一个类型参数或抽象类型实例没有非 **null** 值。在(§3.1)的解释中，易变类型的值成员不能出现在路径中。

易变类型符合下面 4 个分类中的一个：

复合类型 T_1 **with** ... **with** T_n {R}是易变的条件是以下三个中的一个.

1. T_2,...,T_n 是一个类型参数或抽象类型
2. T_1 是一个抽象类型，并且 R 或 T_j j>1 给复合类型提供了一个抽象成员
3. T_1,...,T_n 中有一个是单例类型

在这里类型 S 给类型 T 提供了一个抽象成员的意思是 S 有一个抽象成员同时也是 T 的成员。一个修饰 R 类型 T 提供一个抽象成员的意思是 R 包括一个抽象成员同时也是 T 的成员。

类型设计器是易变的意思是它是一个易变类型的别名，或者它制定了一个以易变类型为其上界的类型参数或抽象类型。

如果路径 p 是易变的，那么单例类型 p.**type** 就是易变的。

既存类型 T **forSome** {Q}是易变的条件是 T 是易变的。

3.7. 类型擦除

类型是通用的的含义是其包括类型参数或类型变量。类型擦除指的就是从(通用)类型到特定类型的映射。类型 T 的擦除类型的写法是|T|。擦除映射的定义如下。

- 别名类型的擦除就是其右侧的擦除
- 抽象类型的擦除就是其上界的擦除

[4] 现有的 Scala 编译器限定该边界的嵌套的参数化层次最多只能比操作数类型的最大嵌套层次深两层

- 参数化类型 scala.Array[T_1]的擦除是 scala.Array[|T_1|]
- 其他参数化类型 T[T_1,...,T_n]的擦除就|T|
- 单例类型 p.**type** 的擦除就是 p 的类型的擦除
- 类型映射 T#x 的擦除就是|T|#x
- 复合类型 T_1 **with** ...**with** T_n {R}的擦除就是|T_1|
- 既存类型 T **forSome** {Q}的擦除就是|T|

4. 基本声明与定义

语法：

```
Dcl             ::= 'val' ValDcl
                  | 'var' VarDcl
                  | 'def' FunDcl
                  | 'type' {nl} TypeDcl
PatVarDef       ::= 'val' PatDef
                  | 'var' VarDef
Def             ::= PatVarDef
                  | 'def' FunDef
                  | 'type' {nl} TypeDef
                  | TmplDef
```

声明引入命名并给其指定类型。声明可以是类定义(§5.1)或者复合类型(§3.2.7)中修饰定义的一部分。

定义引入命名用以表示术语或类型。定义可以是对象或类定义的一部分，或者只是局限在一个代码块中。声明和定义都会产生关联类型命名和类型定义与边界的绑定，将术语名称和类型联系起来。

声明或定义引入的命名的范围是包括该绑定的整个语句序列。。然而在代码块的前向引用中有一个限制：语句序列 $s_1...s_n$ 构成一个代码块，如果 s_i 中的一个简单命名引用一个定义在 s_j 中的实体，且 j>=i，那么 s_i 和 s_j 之间(包括这两者)的定义不能是值或者变量定义。

4.1. 值声明与定义

语法：

```
Dcl     ::= 'val' ValDcl
ValDcl  ::= ids ':' Type
Def     ::= 'val' PatDef
PatDef  ::= Pattern2 {',' Pattern2} [':' Type] '=' Expr
ids     ::= id {',' id}
```

值声明 **val** x: T 表示 x 是一个类型为 T 的值的命名。

值定义 **val** x: T = e 表示 x 是表达式 e 求值的结果。如果值的定义不是递归的，类型 T 可以忽略，默认就是表达式 e 的打包类型(§6.1)。如果给出了类型 T，那么 e 就被期望与其一致。

值定义的求值就是其右侧表达式 e 的求值，除非有一个 lazy 修饰符。值定义的效果是将 x 绑定到变为类型 T 的 e 的结果。lazy 型的值定义只在值第一次被访问时才对右侧表达式求值。

值定义可以在左侧有一个模式(§8.1)。如果 p 是除了简单命名或命名后跟冒号与类型的模式，那么值定义 **val** p = e 可扩展为以下形式：

1. 如果模式 p 具有绑定变量 $x_1,...,x_n$，n>1：

```
val $x = e match {case p => {x1,...,xn}}
val x1 = $x._1
...
val xn = $x._n
```

这里的$x 是一个新命名。

2. 如果 p 有一个唯一的绑定变量 x：

```
val x = e match { case p => x }
```

3. 如果 p 没有绑定变量：

```
e match { case p => ()}
```

示例 4.1.1 以下是一些值定义的例子：

```
val pi = 3.1414
val pi: Double = 3.1415        //与第一个等价
val Some(x) = f()          //模式定义
val x :: xs = mylist       //中缀模式定义
```

后两个定义具有以下扩展：

```
val x = f() match { case Some(x) = x }
val x$ = mylist match { case x :: xs => {x, xs} }
val x = x$._1
val xs = x$._2
```

声明或定义的值的命名不能以_=结束。

值声明 **val** $x_1,...,x_n$: T 是 **val** x1: T; ...; **val** xn: T 的简写形式。值定义 **val** $p_1,...,p_n$ = e 是 **val** p_1 = e; ...; **val** p_n = e 的简写形式。值定义 **val** $p_1,...,p_n$: T = e 是 **val** p_1:T = e; ...; **val** p_n:T = e;的简写形式。

4.2. 变量声明与定义

语法：

```
Dcl     ::= 'var' VarDcl
Def     ::= 'var' VarDef
VarDcl  ::= ids ':' Type
VarDef  ::= PatDef
          | ids ':' Type '=' '_'
```

变量声明 **var** x: T 等价于声明一个 getter 函数 x 和一个 setter 函数 x_=，如下所示：

```
def x: T
```

```
def x_= (y: T): Unit
```

包括变量声明的类的实现可以用变量定义来定义这些变量，也可以直接定义 getter 和 setter 函数。

变量定义 **var** x: T = e 引入了一个类型为 T 的可变变量，并用表达式 e 作为初始值。类型 T 可忽略，默认用 e 的类型。如果给定了 T，那么 e 被期望具有一致的类型(§6.1)。

变量定义可以在左侧有一个模式(§8.1)。如果 p 是除了简单命名或命名后跟冒号与类型的模式，那么变量定义 **var** p = e 具有和值定义一致的扩展模式，除了那些 p 中自由的命名是可变的变量，不是值。

任何声明或定义的变量的命名不能以_=结尾。

变量定义 **var** x: T = _只能以模板成员出现。该定义表示一个可变字段和一个默认初始值。该默认值取决于类型 T:

0	如果 T 是 Int 或其子类型
0L	如果 T 是 Long
0.0f	如果 T 是 Float
0.0d	如果 T 是 Double
false	如果 T 是 Boolean
{}	如果 T 是 Unit
null	所有其他类型

如果变量定义是模板成员，那么他们同时引入一个 getter 函数 x(返回当前赋给变量的值)和一个 setter 函数 x_=(改变当前赋给变量的值)。函数具有与变量声明相同的识别标识。模板具有 getter 和 setter 函数成员，初始的变量不能以模板成员的形式被直接访问。

示例 4.2.1 下面的例子显示了 Scala 中如何模拟属性。首先定义了一个类 TimeOfDayVar，具有可更新的整型字段表示时分秒。其实现包括一些测试，只允许合法值赋给这些字段。但是用户代码可以像访问其他变量那样直接访问这些字段。

```
class TimeOfDayVar {
  private var h: Int = 0
  private var m: Int = 0
  private var s: Int = 0

  def hours               = h
  def hours_= (h: Int)    = if (0 <= h && h < 24) this.h = h
                            else throw new DataError()
  def minutes             = m
  def minutes_= (m: Int)  = if(0<= m && m < 60) this.m = m
                            else throw new DataError()
  def seconds             = s
  def seconds_= (s: Int)  = if (0 <= s && s < 60) this.s = s
                            else throw new DataError()
}
val d = new TimeOfDayVar
d.hours = 8; d.minutes = 30; d.seconds = 0
```

```
d.hours = 24                    //抛出一个 DataError 异常。
```

变量声明 **var** $x_1,...,x_n$: T 是 **var** x_1: T; ...; **var** x_n: T 的简写形式。变量定义 **var** $x_1,...,x_n$ = e 是 **var** x_1 = e; ...; **var** x_n = e 的简写形式。变量定义 **var** $x_1,...,x_n$: T = e 是 **var** x_1:T = e; ...; **var** x_n:T = e 的简写形式。

4.3. 类型声明与类型别名

语法：

```
Dcl     ::= 'type' {nl} TypeDcl
TypeDcl ::= id [TypeParamClause] ['>:' Type] ['<:' Type]
Def     ::= type {nl} TypeDef
TypeDef ::= id [TypeParamClause] '=' Type
```

类型声明 **type** t[tps] >: L <: U 声明了 t 是一个下界为 L 上界为 U 抽象类型。如果类型参数子句[**type**]被忽略的话，t 是一个一阶类型的抽象，否则，t 就是一个类型构造器，接受类型参数子句中指明的参数。

如果一个类型声明是一个类型的成员声明，该类型的实现可以用任何符合条件 L<:T<:U 的类型 T 来实现 t。如果 L 与 U 不一致将会导致编译时错误。边界的某边或全部可被忽略。如果没有给出下界 L，类型 scala.Nothing 就会是默认的下界。如果没有给出上界 U，类型 scala.Any 就会是默认的上界。

类型构造器声明给与 t 有关的具体类型加上了额外的限制。除了边界 L 和 U，类型参数子句可以引入由类型构造器一致性(§3.5.2)控制的高阶边界和差异。

类型参数的域超越了边界 >:L <:U 和类型参数子句 tps 本身。(抽象类型构造器 tc 的)高阶类型参数子句拥有由类型参数声明 tc 限制的同样的域。

一些与嵌套域有关的例子，以下声明是完全等价的：**type** t[m[x] <: Bound[x], Bound[x]]，**type** t[m[x] <: Bound[x], Bound[Y]]和 **type** t[m[x] <: Bound[x], Bound[_]]，类型参数 m 的域限制为 m 的声明。以上所有情况中，t 是在两个类型构造器上抽象的抽象类型成员：m 是有一个类型参数并且是 Bound 的子类型的类型构造器，t 是第二个类型构造参数。t[MutableList, Iterable]是 t 的合法用法。

类型别名 **type** t = T 定义了 t 是类型 T 的别名命名。类型别名的左侧可以是一个类型参数子句，比如 **type** t[tps] = T。类型参数的域超过了右侧的 T 和类型参数子句本身。

定义(§4)与类型参数(§4.6)的域的规则使类型命名在其自身或右侧出现成为可能。然而如果类型别名递归引用到定义的类型构造器自身将会导致静态错误。也就是类型别名 **type** t[tps] = T 中的类型 T 不能直接或间接的引用到命名 t。如果抽象类型是其自身直接或间接的上界或下界也会导致错误。

示例 4.3.1 下面是合法的类型声明与定义：

```
type IntList = List[Integer]
type T <: Comparable[T]
type Two[A] = Tuple2[A, A]
type MyCollection[+X] <: Iterable[X]
```

以下是非法的：

```
type Abs = Comparable[Abs]     //递归类型别名

type S <: T                          //S, T 自我绑定
type T <: S

type T >: Comparable[T.That]   //无法从 T 选择
                               //T 是类型而不是值
type MyCollection <: Iterable  //类型构造器必须显式列出参数
```

如果类型别名 **type** t[tps] = S 指向类类型 S，命名 t 也可用作类型为 S 的对象的构造器。

示例 4.3.2 下面的 Predef 对象包括了一个定义，将 Pair 作为参数化类 Tuple2 的别名：

```
type Pair[+A, +B] = Tuple2[A, B]
```

因此，对于任意的两个类型 S 和 T，类型 Pair[S, T]等价于类型 Tuple2[S, T]。Pair 还可以用来作为 Tuple2 构造器的替换。并且由于 Tuple2 是一个 case 类，Pair 也是 case 类工厂 Tuple2 的别名，这在表达式或模式中均有效。因此以下都是 Pair 的合法使用。

```
val x: Pair[Int, String] = new Pair(1, "abc")
val y: Pair[String, Int] = x match {
  case Pair(i, s) => Pair(z + i, i * i)
}
```

4.4. 类型参数

语法：

```
TypeParamClause     ::= '[' VariantTypeParam {',' VariantTypeParam} ']'
VariantTypeParam    ::= ['+' | '-'] TypeParam
TypeParam           ::= (id | '_') [TypeParamClause] ['>:' Type]
                           ['<:' Type] ['>%' Type]
```

类型参数在类型定义，类定义和函数定义中出现。本节中我们只考虑有下界>:L 和上界<:U 的类型参数定义，视图边界<%U 将在 7.4 节中讨论。

一阶类型参数最一般的形式是+/- t >:L <:U。这里 L 和 U 是下界和上界，限制了参数的可能的类型参量。如果 L 与 U 不一致将会导致编译错误。+/-是差异，指一个可选前缀+或-。

在类型参数子句中的所有类型参数的名称必须两两不同。类型参数的作用域在每个类型参数子句中。因此类型参数作为自己边界的一部分或同一子句中其他类型参数的边界出现是合理的。然而，类型参数不能直接或间接的作为自己的边界。

类型构造参数给类型参数增加了一个嵌套的类型参数子句。最常见的类型构造器参数的形式是+/- t[tps] >: L <: U.

以上域的限制可归纳为嵌套类型参数子句，该子句声明了高阶的类型参数。高阶类型参数(类型参数 t 的类型参数)只在他们直接包围的参数子句(可能包括更深嵌套层次的子句)和 t 的边界中可见。因此他们的名字只需与其他可见参数两两不同。由于高阶类型参

数的命名往往是无关的，因此可以用'_'来指示它们，这种情况下其他地方均不可见。

示例 4.4.1 下面是一些合法的类型参数子句：

```
[S, T]
[Ex <: Throwable]
[A <: Comparable[B], B <: A]
[A, B >: A, C >: A <: B]
[M[X], N[X]]
[M[_], N[_]]   //和上一个等价
[M[X <: Bound[X]], Bound[_]]
[M[+X] <: Iterable[X]]
```

以下是一些非法的类型参数子句：

```
[A >: A]                          //非法，“A”做了自己的边界
[A <: B, B <: C, C <: A]   //非法，“A”做了自己的边界
[A, B, C >: A <: B]               //非法，“C”的下界“A”与上界“B”不一致
```

4.5. 差异标注

差异标注指示了参数化类型的实例在子类型(§3.5.2)上是如何不同的。“+”类型的差异指协变的依赖，“-”类型的差异指逆变的依赖，未标注指不变依赖。

差异标注限制了被标注类型变量在与类型参数绑定的类型或类中出现的方式。在类型定义 **type** T[tps] = S 或类型声明 **type** T[tps] >: L <: U 中，“+”标注的类型参数只能出现在协变的位置，“-”标注的类型参数只能出现在逆变的位置。类似地，对于类定义 **class** C[tps](ps) exntends T x: S => ...@, “+”标记的类型参数只能出现在类型自身 S 和模板 T 的协变位置，“-”标记的类型参数只能出现在逆变位置。

类型或模板中类型参数的协变位置定义如下：与协变相反的是逆变，非变与其自身相反。最高级的类型或模板总是在协变位置。差异位置按照下述方式变化：

- 方法参数的差异位置是参数子句差异位置的相对位置。
- 类型参数的差异位置是类型参数子句差异位置的相对位置
- 类型声明或类型参数的下界的差异位置是类型声明或参数差异位置的相对位置
- 类型别名 **type** T[tps] = S 右侧的 S 总是处于非变位置。
- 可变量的类型总是处于非变位置
- 类型选择 S#T 的前缀 S 总是处于非变位置
- 类型 S[…T…]的类型参量 T：如果对应的类型参数是非变的，那么 T 就在非变位置。如果对应的类型参数是逆变的，那么 T 的差异位置就是类型 S[…T…]的差异位置的相对位置。

到类的对象私有的值，变量或方法的引用的差异并未被检查。这些成员中类型参数可以出现在任意位置，并未限制其合法的差异标注。

示例 4.5.1 下面的差异标注是合法的：

```
abstract class P[+A, +B] {
  def fst: A; def snd: B
}
```

有了这个差异标注，类型 P 的子类型将对其参量自动协变。例如，

```
P[IOException, String] <: P[Throwable, AnyRef]
```

如果我们使 P 的元素可变，差异标注就非法了。

```
abstract class Q[+A, +B](x: A, y: B) {
  var fst: A = x            //*** 错误：非法差异：
  var snd: B = y            //'A', 'B'出现在非变位置。
}
```

如果可变变量是对象私有的，那么类定义就是合法的了：

```
abstract class R[+A, +B](x: A, y: B){
  private[this] var fst: A = x         //OK
  private[this] var snd: B = y         //OK
}
```

示例 4.5.2 下面的差异标注是非法的，因为 a 出现在 append 的参数的逆变位置：

```
abstract class Vector[+A] {
  def append(x: Vector[A]): Vector[A]
                  //**** 错误：非法的差异：
                  //'A'出现在逆变位置
}
```

这个问题可以通过对下界求值的方式将 append 的类型泛化来解决。

```
abstract class Vector[+A] {
  def append[B >: A](x: Vector[B]): Vector[B]
}
```

示例 4.5.3 下面是逆变类型参数有用的一个例子。

```
abstract class OutputChannel[-A] {
  def write(x: A): Unit
}
```

有了这个标注，OutputChannel[AnyRef]将和 OutputChannel[String]一致。也就是一个可以写任何对象的 channel 可以代替只能写 String 的 channel

4.6. 函数声明与定义

语法：

```
Dcl                ::= 'def' FunDcl
FunDcl             ::= FunSig : Type
Def                ::= 'def' FunDef
FunDef             ::= FunSig [':' Type] '=' Expr
FunSig             ::= id [FunTypeParamClause] ParamClauses
FunTypeParamClause ::= '[' TypeParam {',' TypeParam} ']'
ParamClauses       ::= {ParamClauses} [[nl] '(' 'implicit' Params ')']
ParamClause        ::= [nl] '(' [Params] ')'}
```

```
Params              ::= Param {‘,’ Param}
Param               ::= {Annotation} id [‘:’ ParamType]
ParamType           ::= Type
                    | ‘=>’ Type
                    | Type ‘*’
```

函数声明具有这样的形式：**def** f psig: T，f 是函数的名称，psig 是参数签名，T 是返回类型。函数定义 **def** f psig: T = e 还包括了函数体 e，例如一个表达式定义了函数的结果。参数签名由一个可选的类型参数子句[tps]，后跟零个或多个值参数子句$(ps_1)\ldots(ps_n)$构成。这样的声明或定义引入了一个值，该值具有一个(可能是多态的)方法类型，其参数类型与返回类型已给出。

已给出的函数体的类型被期望与函数声明的返回类型一致(§6.1)。如果函数定义不是递归的，那么返回类型则可省略，因为其可由函数体打包的类型推断出来。

类型参数子句 tps 由一个或多个类型声明(§4.3)构成，在其中引入了可能具有边界的类型参数。类型参数的域包括整个签名，也包括任何类型参数边界以及函数体(如果有的话)。

值参数子句 ps 由零个或多个规范类型绑定(如 x: T)构成，这些类型绑定绑定了值参数以及将它们与它们的类型联系起来。一个规范值参数命名 x 的范围是函数体(如果有的话)。所有的类型参数名及值参数名必须两两不同。

4.6.1. 叫名参数

语法：

```
ParamType       ::= ‘=>’ Type
```

值参数类型可以有前缀=>，例如：x: => T。这样一个参数的类型就是无参方法类型=>T。这表明对应的参数并没有在函数应用处求值，而是在函数中每次使用时才求值。也就是该参数以叫名的方式求值。

示例 4.6.1 声明：

```
def whileLoop (cond: => Boolean) (start: => Unit): Unit
```

表示 whileLoop 的所有参数都以叫名的方式求值。

4.6.2. 重复参数

语法：

```
ParamType   ::= Type ‘*’
```

参数段中的最后一个参数可以有后缀‘*’，例如：(…, x: T*)。方法中这样一个重复参数的类型就是序列类型 scala.Seq[T]。具有重复参数 T*的方法具有可变数目的类型为 T 的参数。也就是，如果方法 m 的类型$(T_1,\ldots,T_n,S*)U$应用到参数$(e_1,\ldots,e_k)$上，且有 k>=n，那么 m 就被认为在应用中具有类型$(T_1,\ldots,T_n,S,\ldots,S)U$，S 重复 k-n 次。这个规则的唯一例外是如果最后一个参数用_*类型标注的方式被标记为一个序列参量。如果以上的 m 应用到参数$(e_1,\ldots,e_n,e’:_*)$上，那么该应用中 m 的类型就被认为

$(T_1,\ldots,T_n,$scala.Seq[S]$)$。

示例 4.6.2 以下方法定义计算了可变数目的整形参数的和：

```
def sum(args: Int*) = {
  var result = 0
  for(arg <- args.elements) result += arg
  result
}
```

以下对该方法的应用可得出的结果为 0，1，6：

```
sum()
sum(1)
sum(1,2,3)
```

更进一步的，假设以下定义：

```
var xs = List(1,2,3)
```

以下对方法 sum 的应用是错误的：

```
sum(xs)        //  ***** error: expected: Int, found: List[Int]
```

相比较，以下应用是正确的，并产生结果 6：

```
sum(xs:_*)
```

4.6.3. 过程

语法：

```
FunDcl  ::= FunSig
FunDef  ::= FunSig[nl] '{'Block'}'
```

过程有特殊语法，例如，返回 Unit 值{}的函数。过程声明只是返回类型被忽略的函数声明。返回类型自动定义为 Unit 类型。例如 **def** f(ps)等价于 **def** f(ps):Unit。

过程定义是返回类型及等号被忽略的函数定义；其定义表达式必须是一个代码块。例如：**def** f(ps){stats}等价于 **def** f(ps):Unit={stats}.

示例 4.6.3 以下是名为 write 的过程的声明与定义：

```
trait Writer {
  def write(str: String)
}
object Terminal extends Writer{
  def write(str: String) {System.out.println(str)}
}
```

以上代码被内部自动完成为：

```
trait Writer {
  def write(str: String): Unit
```

```
}
object Terminal extends Writer{
  def write(str: String): Unit = {System.out.println(str)}
}
```

4.6.4. 方法返回类型推断

类成员定义 m 重载了基类 C 中的一些其他的函数 m′可以略去返回类型，即使是递归的也无所谓。因此被重载的函数 m′的返回类型 R′(被认为是 C 的成员) 在对 m 的每次调用中被认为是 m 的返回类型。在以上方式中，m 右侧的类型 R 可以被确定，并作为 m 的返回类型。注意到 R 可以与 R′不同，只要 R 与 R′一致即可。

示例 4.6.4 假定有以下定义：

```
trait I {
  def factorial(x: Int): Int
}
class C extends I {
  def factorial(x: Int) = if (x==0) 1 else x * factorial(x - 1)
}
```

这里忽略 C 中 factorial 的返回类型是没问题的，即使这是一个递归的方法。

4.7. Import 子句

语法：

```
Import          ::= 'import' ImportExpr {',' ImportExpr}
ImportExpr      ::= StableId '.' (id | '_' | ImportSelectors)
ImportSelectors ::= '{' { ImportSelector ','} (ImportSelector | '_') '}'
ImportSelector  ::= id ['=>' id | '=>' '_']
```

import 子句形式为 **import** p.I，p 是一个稳定标识符(§3.1)，I 是一个 import 表达式。import 表达式确定了 p 的成员中一些名称的集合，使这些名称不加限定即可用。最普通的 import 表达式的形式是一个 **import** 选择器的列表。

$\{x_1=>y_1,\ldots,x_n=>y_n,\_\}$

其中 n>=0，最后的通配符'_'可以没有。它使每个成员 $p.x_i$ 在未限定的名称 y_i 下可用。例如每个 **import** 选择器 $x_i=>y_i$ 将 $p.x_i$ 重命名为 y_i。如果存在最终的通配符，p 的除 $x_1,\ldots,x_n$ 之外的成员 z 也将在其自身未限定的名称下可用。

import 选择器对类型和术语成员起同样作用。例如，import 子句 **import** p.{x=>y}将术语 p.x 重命名为术语 y，并且将类型名 p.x 重命名为类型名 y。这两个名称中至少有一个引用 p 的一个成员。

如果 import 选择器的目标是通配符，**import** 选择器就会隐藏对源成员的访问。例如，**import** 选择器 x=>_将 x“重命名”为通配符号(作为用户程序中的名称不可访问)，因此也有效阻止了对 x 的非限制性的访问。这在同一个 import 选择器列表最后有一个通配符的情况下是有用的，此时将引入所有前面 import 选择器没有提及的成员。

由 import 子句所引入的绑定的域开始于 import 子句之后并扩展至封闭块，模板，包子句，或编译单元的末尾，具体决定于哪个先出现。

存在一些简化形式。**import** 选择器可以只是一个名字 x。这种情况下，x 以没有重命名的方式被引入，因此该 import 选择器等价于 x=>x。更进一步，也可以用一个标识符或通配符来替换整个的 import 选择器列表。import 子句 import p.x 等价于 import p.{x}，例如不用限定 p 的成员 x 即可用。import 子句 p._等价于 import p.{_}，例如不用限定 p 的所有成员 x 即可用(该处是 java 中 import p.*的同义语)。

一个 import 子句中的多个 import 表达式 import $p_1.I_1$,…,$p_n.I_n$ 被解释为一个 import 子句的序列 import $p_1.I_1$;…;**import** $p_n.I_n$。

示例 4.7.1 考虑以下对象定义：

```
object M{
  def z = 0, one = 1
  def add(x: Int, y: Int):Int = x + y
}
```

因此代码块

```
{import M.{one, z => zero, _}; add(zero, one)}
```

就等价于代码块

```
{M.add(M.z, M.one)}
```

5. 类与对象

语法：

```
TmplDef ::= [‘case’] ‘class’ ClassDef
          | [‘case’] ‘object’ ObjectDef
          | ‘trait’ TraitDef
```

类(§5.3)与对象(§5.4)都用模板来定义。

5.1. 模板

语法：

```
ClassTemplate   ::= [EarlyDefs] ClassParents [TemplateBody]
TraitTemplate   ::= [EarlyDefs] TraitParents [TemplateBody]
ClassParents    ::= Constr {‘with’ AnnotType}
TraitParents    ::= AnnotType {‘with’ AnnotType}
TemplateBody    ::= [nl] ‘{’[SelfType] TemplateStat{semi TemplateStat}‘}’
SelfType        ::= id [‘:’ Type] ‘=>’
                  | this ‘:’ Type ‘=>’
```

模板定义了对象的特征或对象的类或单个对象的类型签名，行为和初始状态。模板是实例创建表达式，类定义和对象定义的一部分。模板 sc **with** mt_1 **with** ... **with** mt_n {stats}包括一个构造器调用 sc，定义了模板的超类；以及特征引用 mt_1,...,mt_n (n>=0)，定义了模板的特征；和一个语句序列 stats，包括初始化代码和模板额外的成员定义。

每个特征引用 mt_i 必须表示一个特征(§5.3.3)，作为对比，超类构造器 sc 一般指向一个类而不是特征。可以写出由特征引用开始的一系列的父类，比如 mt_1 **with** ... **with** mt_n。在这种情况下父类列表被自动扩展以包括 mt_1 的超类，并作为第一个父类型。新的超类应当有至少一个无参数构造器。在以后内容中，我们将总是假定该自动扩展已经被执行，因此模板的第一个父类是一个正规的超类构造器，而不是一个特征引用。

每个类的父类列表也总是自动扩展至 scala.ScalaObject 特征做为最后一个混入类型。例如：

sc **with** mt_1 **with** ... **with** mt_n {stats}

将变为

mt_1 **with** ... **with** mt_n {stats} **with** ScalaObject {stats}

模板的父类列表必须格式正确。也就是由超类构造器 sc 指示的类必须是所有特征 $mt_1,...,mt_n$ 的超类的子类。换句话说，模板继承的非特征类在继承层级中构成了一个链，其起始为模板的超类。

模板的超类型的最低要求是类类型或复合类型(§3.2.7)由其所有的父类类型构成。

语句序列 stats 包括成员定义，定义了新成员或覆盖父类中的成员。如果模板构成了抽象类或特征的定义，语句部分 stats 还可以包括抽象成员的声明。如果模板构成了实体类的定义，stats 仍可以包括抽象类型成员的声明，但是不能包括抽象术语成员。更进一步，stats 还可以包括表达式；这些将以他们给定的顺序作为模板的初始化步骤的一部分来执行。

模板语句的序列可以有一个正式的参数定义和一个箭头作为前缀，例如 x=>或 x:T =>。如果给出了一个正式的参数，这个参数在模板体中可用作引用 this 的别名。如果给出了正式的参数的类型 T，这个定义就会以下面的方式影响类或对象的自类型 S：设 C 是定义了模板的类、特征或对象的类型，如果给定正式的自参数类型 T，S 就是 T 和 C 的最大下界。如果没有给出 T，S 就是 C。在模板内，this 的类型就会被假定为 S。

类或者对象的自类型必须与模板 t 继承的所有类的自类型一致。

自类型标注的第二种形式是 this: S=>。它规定了 this 的类型 S，但并没有为其引入别名。

示例 5.1.1 考虑以下的类定义

```
class Base extends Object{}
trait Mixin extends Base{}
object O extends Mixin{}
```

这种情况下，O 的定义可扩展为

```
object O extends Base with Mixin{}
```

继承自 Java 类型 模板可以有一个 Java 类作为其超类，或者 Java 接口作为其混入。

模板求值 考虑模板 sc **with** mt_1 **with** mt_n {stats}。如果这是特征(§5.3.3)的一个模板，那么其混入求值由语句序列 stats 的求值构成。

如果这不是一个特征的模板，那么其求值包括以下步骤：

- 首先，对超类的构造器 sc(§5.1.1)求值
- 然后，模板线性化中的所有基类，直到有 sc 表示的模板的超类将会做混入求值。混入求值的顺序是线性化中出现顺序的反序，比如紧挨 sc 前的类会被第一个求值。
- 最后对语句序列 stats 求值

5.1.1. 构造器调用

语法：

```
Constr  ::= AnnotType {'(' [Exprs [',']] ')'}
```

构造器调用定义了类型，成员以及由实例创建表达式或由类或对象定义继承的对象定义创建的对象的初始状态。构造器调用是一个函数应用 $x.c[targs](args_1)...(args_n)$，x 是一个稳定的标识符(§3.1)，c 是一个指向类或

定义别名类型的类型名，targs 是一个类型参量列表，$args_1,...,args_n$ 是参量列表，与该类的某个构造器的参数匹配。

前缀'x.'可以省略。类型参数列表只在类 c 需要类型参数时才给出。即使这样这也可以在使用本地类型推断(§6.25.4)可以合成参数列表时忽略。如果没有显式的给出参量，就会默认给一个空参量列表()。

构造器调用 $x.c[targs](args_1)...(args_n)$ 的执行包括以下几个步骤：

- 首先对前缀 x 求值
- 然后参量 $args_1,...,args_n$ 按照从左至右的顺序求值。
- 最后，对 c 指向的类的模板求值，初始化正在被创建的内容。

5.1.2. 类的线性化

通过类 C 可达的直接继承关系的传递闭包可达的类称为 C 的基类。由于混入的关系，基类的继承关系基本上构成一个直接非循环图。这个图的线性化如下定义：

定义 5.1.2 设类 C 有模板 C_1 **with** ... **with** C_n { stats }。C 的线性化 $\mathscr{L}(C)$ 定义如下：

$$\mathscr{L}(C) = C, \mathscr{L}(C_n) \vec{+} ... \vec{+} \mathscr{L}(C_1)$$

这里$\vec{+}$表示串联，算符右侧的元素替换算符左侧标识的元素

$$\{a,A\}\vec{+}B = a,(A\vec{+}B) \text{ 如果 } a \notin B$$
$$= (A\vec{+}B) \text{ 如果 } a \in B$$

示例 5.1.3 考虑以下的类定义

```
abstract class AbsIterator extends AnyRef { ... }
trait RichIterator extends AbsIterator { ... }
class StringIterator extends AbsIterator { ... }
class Iter extends StringIterator with RichIterator { ... }
```

那么类 Iter 的线性化就是：

```
{ Iter, RichIterator, StringIterator, AbsIterator, ScalaObject, AnyRef,
Any }
```

特征 ScalaObject 出现在列表里是因为每个 Scala 类(§5.1)都会添加它作为最后的混入。

注意一个类的线性化优化了继承关系：如果 C 是 D 的子类，那么 C 就会在任何 C 和 D 同时出现的线性化中出现在 D 前面。定义 5.1.2 也满足一个类的线性化总是包括其直接超类的线性化作为其后缀这个性质。例如，StringIterator 的线性化就是：

```
{ StringIterator, AbsIterator, ScalaObject, AnyRef, Any }
```

这就是其子类 Iter 的线性化的后缀。对混入的线性化却不是这样。例如，RichIterator 的线性化是：

```
{ RichIterator, AbsIterator, ScalaObject, AnyRef, Any}
```

这并不是 Iter 线性化的后缀。

5.1.3. 类成员

由模板 C_1 **with** ... **with** C_n { stats }定义的类 C 可以在语句序列 stats 中定义成员和继承所有父类的成员。Scala 采取了 Java 和 C#的方法静态重载的方便之处。因此一个类可以定义和/或集成多个同名方法。要确定类 C 定义的成员是否覆盖了父类的成员，或 C 中两个同时存在的重载的变量，Scala 使用了以下的成员匹配定义：

定义 5.1.4 成员定义 M 与成员定义 M′，匹配的条件是：首先他们绑定了同样的名称，然后符合下面中的一条：

1. M 和 M′不是方法定义
2. M 和 M′定义了同态的方法并具有等价的参数类型
3. M 定义了一个无参数方法，M′定义了一个具有空参数列表的方法，或反之亦然。
4. M 和 M′定义了多态的方法，具有同样数目的参数类型$\overline{T}$，$\overline{T}'$和同样数目的类型参数$\overline{t}$，$\overline{t}'$，并且$\overline{T}' = [\overline{t}'/\overline{t}]\,\overline{T}$。

成员定义有两类：实体定义与抽象定义。类 C 的成员要么直接定义(例如出现在 C 的语句序列 stats 中)或继承。有两条规则来确定类的成员集合，每个分类一条：

定义 5.1.5 类 C 的实体成员是某些类 $C_i \in \mathscr{L}(C)$ 中的实体定义 M，除非在前置的类 $C_j \in \mathscr{L}(C)$ (j<i)中有一个与 M 匹配的直接定义的实体成员 M′。

类 C 的抽象成员是某些类 $C_i \in \mathscr{L}(C)$ 中任意抽象定义的 M，除非 C 已经包括一个与 M 匹配的实体成员 M′，或者在前置类 $C_j \in \mathscr{L}(C)$ 且 j<i 中已经有一个与 M 匹配的直接定义的抽象成员 M′。

该定义也确定了类 C 及其父类(§5.1.4)中匹配的成员的重载关系。第一，实体定义总是覆盖抽象定义。第二，如果 M 和 M′都是实体的或抽象的，只有 M 定义的类在(C 的线性化中)M′所在类的前面出现时，M 才会重载 M′。

一个模板定义了两个匹配的成员将会导致错误。一个模板包括两个同名且具有同样擦除(§3.7)类型的成员(直接定义或继承)也将会导致错误。

示例 5.1.6 考虑以下特征定义

```
trait A { def f: Int }
trait B extends A{def f:Int = 1;def g:Int = 2;def h:Int=3}
trait C extends A{override def f:Int = 4; def g:Int}
trait D extends B with C{def h:Int}
```

特征 D 有一个直接定义的抽象成员 h。它从特征 C 继承了成员 f，从特征 B 继承了成员 g。

5.1.4. 覆盖

类 C 的成员 M 与(§5.1.3)中定义的 C 的基类的非私有成员 M′一致可定义为覆盖该成员。在此情况下覆盖成员 M 的绑定必须包含(§3.5.2)被覆盖的成员 M′的绑定。另外以下限制应用于 M 和 M′的修饰符：

- M′不能标记为 **final**。
- M 不能是 **private**(§5.2)。
- 如果 M 在某些封闭类或包 C 中标记为 **private**[C]，那么 M′必须在类或包 C′中标记为 **private**[C′]，且 C′等于 C 或 C′包含于 C。
- 如果 M 标记为 **protected**，那么 M′也必须是 **protected**。

- 如果 M′不是一个抽象成员，那么 M 必须标记为 **override**
- 如果 M′在 C 中是不完整(§5.2)的，那么 M 必须标记为 **abstract override**
- 如果 M 和 M′都是实体值定义，那么他们必须都标记为 **lazy** 或者都不标记为 **lazy**。

对于无参数方法有个特例。如果一个无参数方法定义为 **def** f: T = ...或者 **def** f = ...覆盖了类型()T′中的一个空参数列表方法，那么 f 也被假定为具有一个空参数列表。

示例 5.1.7 考虑以下定义

```
trait Root { type T <: Root}
trait A extends Root { type T <: A}
trait B extends Root { type T <: B}
trait C extends A with B
```

那么类 C 的定义是错误的，因为 C 中 T 的绑定是 **type** T <: B，不能包含绑定 A 中的 **type** T <: A。该问题可通过在类 C 中添加 T 的覆盖定义来解决。

```
class C extends A with B { type T <: C}
```

5.1.5. 继承闭包

设 C 为类类型。C 的继承闭包就是以下类型的最小集合 φ：

- 如果 T 在 φ 中，那么语法上构成 T 的每个类型 T′也在 φ 中。
- 如果 T 是 φ 中的一个类类型，那么 T 的所有父类(§5.1)也在 φ 中。

如果类类型的继承闭包包含无穷个类型将会导致静态错误。(该限制对于使子类型可推断[KP07]是必要的)。

5.1.6. 前置定义

语法：

```
EarlyDefs   ::= '{' [EarlyDef {semi EarlyDef}] '}' 'with'
EarlyDef    ::= {Annotation} {Modifier} PatVarDef
```

模板开头可以是前置字段定义子句，该子句在子类型构造器被调用之前定义了字段的值。在以下模板中

```
{ val p1: T1 = e1
  ...
  val pn: Tn = en
} with sc with mt1 with mtn {stats}
```

初始模式定义 $p_1,...,p_n$ 被称为前置定义。他们定义了构成模板的字段。每个前置定义必须定义至少一个变量。

前置定义在模板被定义与赋类型参数以及在此之前的任意前置定义之前做类型检查与求值，参数可以是类的任意类型参数。在前置定义中在右侧任何对 this 的引用指模板之

外的 this 标识符。因此，前置定义不可能引用模板创建的对象，或者引用其字段与方法，除了同一段落中前面的前置定义之外。再者，对前面的前置定义的引用总是引用那里定义的值，并不牵涉到覆盖的定义。

换句话说，前置定义的代码块以包含一些值定义的局部代码块的形式求值。

前置定义在模板的父类构造器被调用之前以他们被定义的顺序求值。

示例 5.1.8 前置定义在特征中特别有用，它们没有通常的构造器参数。例如：

```
trait Greeting {
  val name: String
  val msg = "How are you, "+name
}
class C extends {
  val name = "Bob"
} with Greeting{
   println(msg)
}
```

在以上代码中，字段 name 在 Greeting 的构造器之前被初始化。类 Greeting 中的字段 msg 被正确初始化为"How are you, Bob"。

如果 name 不是在 C'的类主体中被初始化，而是在 Greeting 的构造器之后初始化。在此情况下，msg 会被初始化为"How are you, <null>"。

5.2. 修饰符

语法：

```
Modifier        ::= LocalModifier
                  | AccessModifier
                  | 'override'
LocalModifier   ::= 'abstract'
                  | 'final'
                  | 'sealed'
                  | 'implicit'
                  | 'lazy'
AccessModifier  ::= ('private' | 'protected') [AccessQualifier]
AccesQualifier  ::= '[' (id | 'this') ']'
```

成员定义前的修饰符会影响其标定的标识符的可见性及使用。如果给出了多个修饰符，其顺序没有关系，但是同一个修饰符不能重复出现。重复定义前的修饰符将应用于所有定义上面。控制修饰符的有效性与含义的规则如下：

- **private** 修饰符可以应用在模板的任何定义与声明上。这些成员只能被直接封闭的模板和其伴随模块及伴随类访问(示例 5.4.1)。他们不能被子类继承，也不能覆盖父类中的定义。

  此修饰符可由一个标识符 C 限定(如 **private**[C])，表示该类或包包含该定义。由该标识符标识的成员只能由包 C 或类 C 以及他们的伴随模块(§5.4)来访问。

这些成员也仅能由 C 内的模板访问。

限定的一个特殊形式是 **private**[**this**]。由该标识符标记的成员 M 只能从该成员定义的对象内访问。也就是选择 p.M 只有在前缀是 **this** 和包含该引用的类 O 的 O.**this** 时才合法。这也就是没有加限定的 **private** 的应用。

标记为没有限定的 private 的成员称为类私有，标记为 **private**[**this**]的成员称为对象私有。不管是类私有还是对象私有都可以称为私有成员，但是 **private**[C]不是，C 是一个标识符，在后者该成员称为限定私有。

类私有或对象私有成员不能是抽象的，并且不能再由 **protected**，**final** 或 **override** 修饰符限定。

- **protected** 标识符应用到类成员定义上。类的保护成员可以从以下位置访问：
  - 定义的类模板内
  - 所有以定义的类为基类的模板
  - 任何这些类的伴随模块

  protected 修饰符可以由一个标识符 C 来限定(例 **protected**[C])，C 必须是一个包含该定义的类或者包。由该修饰符标记的成员可以被包或者类 C 内的所有代码及伴随模块(§5.4)访问。

  一个 protected 标识符 x 可在选择 r.x 作为成员名称的条件是：
  - 访问是在定义该成员的模板内；如果给出了限定 C 的话，就是在包或者类 C 以及其伴随模块内，或者：
  - r 是保留字 **this** 和 **super** 中的一个，或者：
  - r 的类型与包含访问的类的类型实例一致

  限定的一个特殊形式是 **protected**[**this**]。被此修饰符标记的成员 M 只能从其定义的对象内访问。也就是选择 p.M 只有在前缀是 **this** 或者 O.**this** 的时候才合法(类 O 包含该引用)。这也就是未加限定的 protected 应用的方式。
- override 修饰符应用于类成员定义或声明。对于那些覆盖了父类中某些实体成员定义的成员定义与声明，该修饰符是必须的。如果给出了 override 修饰符，那么应当至少有一个被覆盖的成员定义或声明(可以是实体的或者抽象的)
- 当 **override** 和 **abstract** 一起出现时具有显著不同的意义。该修饰符组合仅用于特征的值成员。标记为 **abstract override** 的成员必须覆盖至少一个其他成员，所有被其覆盖的成员必须是不完整的。

  成员 M 是不完整的条件是：M 是抽象的(例如由一个声明定义)或者标记为 **abstract** 和 **override**，这样即使是被 M 覆盖的成员也成为不完整的。

  注意修饰符组合 **abstract override** 并不影响一个成员是不是实体或者抽象的概念。如果一个成员仅给出了一个声明，那么它就是抽象的；如果给出了完整定义，那么它就是实体的。
- abstract 修饰符可用于类定义。但没必要用于特征。对于具有不完整成员的类来说是必须的。抽象类不能通过构造器调用初始化(§6.10)，除非后跟覆盖了类中所有不完整成员的混入和/或修饰体。只有抽象类和特征可以有抽象术语成员。正如前文所述，abstract 修饰符可以和 **override** 连用，应用于类成员定义。
- **final** 修饰符应用于类成员定义和类定义。标记为 **final** 的类成员定义不能在子类中被覆盖。标记为 **final** 的类不能被模板继承。**final** 对于对象定义是多余的。标记为 **final** 的类或对象的成员隐含定义为 **final** 的，所以对它们来说 **final** 修饰符也是多余的。**final** 修饰符不能修饰不完整成员，并且在修饰符列表中不能与 **private** 或 **sealed** 组合。

- **sealed** 修饰符应用于类定义。标记为 **sealed** 的类不能被直接继承，除非继承的模板和该类定义于同一源文件。然而 **sealed** 的类的子类可以在任何地方继承。
- lazy 修饰符应用于值定义。标记为 **lazy** 的值只在其第一次被访问(可能永远也不发生)时初始化。试图在值初始化时访问该值可能导致循环行为。如果在初始化时有异常被抛出，该值则被认为没有被初始化，随后的访问将会继续尝试对其右侧表达式求值。

示例 5.2.1 以下代码列举了限定私有的用法：

```
package outerpkg.innerpkg
class Outer {
  class Inner {
    private[Outer] def f()
    private[innerpkg] def g()
    private[outerpkg] def h()
  }
}
```

在这里对方法 f 的访问可以出现在 OuterClass 的任何地方，但不能在其外面。对方法 g 的访问可以出现在包 outerpkg.innerpkg 的任何地方，和 Java 中的包私有方法类似。最后，对方法 h 的访问可以出现在包 outerpkg 的任何地方，包含其包括的所有包。

示例 5.2.2 阻止类的使用者去创建该类的新实例的一个常用方法是将该类声明为 **abstract** 和 **sealed**

```
object m {
  abstract sealed class C (x: Int) {
    def nextC = new C(x + 1) {}
  }
  val empty = new C(0){}
}
```

例如以上的代码用户只能通过调用 m.C 中的 nextC 方法来创建类 m.C 的实例。用户无法直接创建类 m.C 的对象。以下两行是错误的：

```
new m.C(0) //*** 错误：C 是抽象的，不能初始化
new m.C(0){}   //*** 错误：从 sealed 类非法继承
```

也可以通过将主要构造器标记为 **private** 来达成这一目的(参见示例 5.3.2)。

5.3. 类定义

语法：

```
TmplDef            ::= 'class' ClassDef
ClassDef           ::= id [TypeParamClauses] {Annotation}
ClassParamClauses  ::= {ClassParamClause}
                       [[nl] '(' implicit ClassParams ')']
ClassParamClause   ::= [nl] '(' [ClassParams] ')'
```

```
ClassParams          ::= ClassParam {‘,’ ClassParam}
ClassParam           ::= {Annotation} [{Modifier} (‘val’ | ‘var’)]
                         Id [‘:’ ParamType]
ClassTemplateOpt     ::= ‘extends’ ClassTemplate
                       | [[‘extends’] TemplateBody]
```

类定义最常见的形式是

class c[tps] as m(ps_1)...(ps_n) **extends** t (n>=0).

此处：

C 是要定义的类的名称

tps 是要定义的类的类型参数的非空列表。类型参数的作用域是整个类定义，包括类型参数段自身。用同一个名称来定义两个类型参数是非法的。类型参数段[tps]可以没有。具有类型参数段的类称为多态的，否则称为单态的。

as 是一个可为空的标注(§11) 序列。如果给出了标注，他们将应用于类的主构造器。

m 是访问修饰符(§5.2)，比如 **private** 或者 **protected**，可以有一个限定。如果给出了这样一个访问修饰符，它将应用于类的主构造器。

(ps_1)...(ps_n)是类的主构造器的正式值参数子句。正式值参数的作用域包含模板 t。然而一个正式值参数并不是任何父类或类型模板 t 成员的一部分。用同一个名称来定义两个正式值参数是非法的。如果没有给出正式参数段，则会假定有一个空的参数段()。

如果正式参数声明 x:T 前面有 **val** 或者 **var** 关键字，针对该参数的一个访问器(getter)定义(§4.2)将会被自动加入类中。getter 引入了类 c 的值成员 x，其定义是该参数的别名。如果引入关键字是 **var**，一个 setter 访问器 x_=(§4.2)也会被自动加入到类中。调用该 setter x_=(e)将会将参数的值变为 e 的求值结果。正式参数声明可以包含修饰符，并将会自动传递给访问器定义。一个有 **val** 或者 **var** 前缀的正式参数不能同时成为叫名参数(§4.6.1)。

t 是一个模板(§5.1)，具有以下形式：

sc **with** mt_1 **with** ... **with** mt_m (stats) (m>=0)

该模板定义了类的对象的基类，行为和初始状态。extends 子句 **extends** sc **with** mt_1 ... **with** mt_m 可以被忽略，默认为 **extends** scala.AnyRef。类体{stats}也可以没有，默认为空{}

这个类定义定义了一个类型 c[tps]和一个构造器，当该构造器应用于与类型 ps 一致的参数时将会通过对模板 t 求值来创建类型 c[tps]的实例。

示例 5.3.1 以下例子展示了类 C 的 **val** 和 **var** 参数

```
class C(x: Int, val y: String, var z: List[String])
var c = new C(1, “abc”, List())
c.z = c.y :: c.z
```

示例 5.3.2 下面的类只能从其伴随模块中创建

```
object Sensitive {
```

```
  def makeSensitive(credentials: Certificate): Sensitive =
    if (credentials == Admin) new Sensitive()
    else throw new SecurityViolationException
}
class Sensitive private (){
  ...
}
```

5.3.1. 构造器定义

语法：

```
FunDef          ::= 'this' ParamClause ParamClauses
                    ('=' ConstrExpr | [nl] ConstrBlock)
ConstrExpr      ::= SelfInvocation
                | ConstrBlock
ConstrBlock     ::= '{' SelfInvocation {semi BlockStat} '}'
SelfInvocation  ::= 'this' ArgumentExprs {ArgumentExprs}
```

一个类可以有除主构造器外的构造器。这些由形如 **def this**$(ps_1)...(ps_n)$ = e 之类的构造器定义所定义。这样的定义在类内引入了额外的构造器，并具有以正式参数列表 $ps_1,...,ps_n$ 形式的参数，其求值由构造器表达式 e 所定义。每个正式参数的作用域是构造器表达式 e。一个构造器表达式可以是一个构造器自调用 this$(args_1)$ ... $(args_n)$或者一个以构造器自调用开始的代码块。构造器自调用必须创建一个类的通用实例。例如，如果问题中的类具有名称 C 和类型参数[tps]，那么构造器自调用必须产生一个 C[tps]的实例；初始化正式类型参数是不允许的。

一个构造器定义中的签名及构造器自调用是有类型检查的，并在类内产生作用域的地方求值，可以加该类的任何类型参数以及该模板的任何前置定义(§5.1.6)。构造器的其他部分会被类型检查并以当前类中一个函数体的形式求值。

如果类 C 有辅助构造器，这些构造器与 C 的主构造器(§5.3)构成了重载的构造器定义。重载解析(§6.25.3)的通常规则应用于 C 的构造器调用，包括构造器表达式中的构造器自调用。然而，不同于其他方法，构造器从不继承。为了防止构造器调用的无限循环，限制了每个构造器自调用只能引用它前面定义的构造器(例如它只能引用前面的辅助构造器或类的主构造器)

示例 5.3.3 考虑以下类定义：

```
class LinkedList[A]() {
  var head = _
  var tail = null
  def isEmpty = tail != null
  def this(head: A) = { this(); this.head = head }
  def this(head: A, tail: List[A]) = { this(head); this.tail = tail}
}
```

这里定义了类 LinkedList 和三个构造器。第二个构造器创建了一个单值列表，第三个构造器创建了一个给出了 head 和 tail 的列表。

5.3.2. Case 类

语法

```
TmplDef ::= 'case' 'class' ClassDef
```

如果一个类定义有 **case** 前缀，那么该类就被称为 case 类

case 类中第一个参数段中的正式参数称为元素，对它们将作特殊处理。首先，该参数的值可以扩展为构造器模式的一个字段。其次，该参数默认添加 **val** 前缀，除非该参数已经有 **val** 或 **var** 修饰符。然后会针对该参数生成一个访问定义(§5.3)。

case 类定义 c[tps](ps$_1$)...(ps$_n$)有类参数 tps 和值参数 ps，会自动生成一个扩展对象(§8.1.7)，定义如下：

```
object c {
  def apply[tps](ps1)...(psn): c[tps] = new c[Ts](xs1)...(xsn)
  def unapply[tps](x: c[tps]) = scala.Some(x.xs11,...,x.xs1k)
}
```

这里，Ts 是类型参数段 tps 中定义的类型向量，每个 xs$_i$ 表示参数段 ps$_i$ 中的参数名称，xs$_{11}$,...,xs$_{1k}$ 表示第一个参数段 xs$_1$ 中所有的参数名。如果类中没有类型参数段，那么 apply 和 unapply 方法也就没有了。如果类 c 是 **abstract** 的，apply 的定义会被忽略。如果 c 的第一个参数段 ps$_1$ 以一个(§4.6.2)中的重复参数结尾，unapply 方法的名称会改为 unapplySeq。如果已经存在伴随对象 c，则不会创建新的对象，但是 apply 和 unapply 方法会添加进现有对象中。

每个 case 类都自动重载类 scala.AnyRef(§12.1)中的一些方法定义，除非 case 类本身已经给出了该方法的定义或在 case 类的某些基类中已经有与 AnyRef 中的方法不同的实体方法定义。特别是：

equals: (Any)Boolean 方法是结构相等的，两个实例相等的条件是他们都属于问题中的 case 类，且他们具有相同的构造器参数。

hashCode: Int 方法计算一个哈希码。如果数据结构成员的 hashCode 方法产生对应相等的哈希值，那么 case 类的 hashCode 产生的值也要相等。

toString: String 方法返回一个包含类名和其元素的字符串表示。

示例 5.3.4 以下是 lambda 演算的抽象语法定义

```
class Expr
case class Var     (x: String)             extends Expr
case class Apply   (f: Expr, e: Expr)      extends Expr
case class Lambda  (x: String, e: Expr)    extends Expr
```

此处定义了一个类 Expr 和 case 类 Var, Apply 和 Lambda。一个 lambda 表达式的传值参数计算器可以写为如下方式：

```
type Env = String => Value
case class Value(e: Expr, env: Env)

def eval(e: Expr, env: Env): Value = e match {
  case Var(x) =>
    env(x)
```

```
    case Apply(f, g) =>
     val Value(Lambda (x, e1), env1) = eval(f, env)
     val v = eval(g, env)
     eval (e1, (y => if (y == x) v else env1(y)))
    case Lambda(_,_) =>
     Value(e, env)
  }
```

可以通过在程序的其他地方扩展类型 Expr 来定义更多的 case 类，例如：

```
case class Number(x: Int) extends Expr
```

可以通过将基类 Expr 标记为 **sealed** 来移除扩展性；在此情况下，所有直接扩展 Expr 的类必须与 Expr 在同一源文件中

5.3.3. 特征

语法：

```
TmplDef            ::= 'trait' TraitDef
TraitDef           ::= id [TypeParamClause] TraitTemplateOpt
TraitTemplateOpt   ::= 'extends' TraitTemplate
                     | [['extends'] TemplateBody]
```

特征是那些要以混入的形式加入到其他类中的类。与通常的类不同，特征不能有构造器参数。且也不能有构造器参数传递给其父类。这些都是没必要的，因为特征在父类初始化后才进行初始化。

假定特征 D 定义了类型 C 的实例 x 的某些特点(例如 D 是 C 的一个基类)。那么 x 中 D 的实际超类型是 $\mathscr{L}(C)$ 中超越 D 的所有基类的复合类型。实际超类型给出了在特征中解析 **super** 的上下文(§6.5)。要注意到实际超类型依赖于特征所添加进的混入组合，当定义特征时是无法知道的。

如果 D 不是特征，那么它的实际超类型就是其最小合适超类型(实际在定义时可知)

示例 5.3.5 以下特征定义了与某些类型的对象可以比较的属性。包括一个抽象方法<和其他比较算符<=，>和>=的默认实现。

```
trait Comparable[T <: Comparable[T]] { self: T =>
  def < (that: T): Boolean
  def <=(that: T): Boolean = this < that || this == that
  def > (that: T): Boolean = that < this
  def >=(that: T): Boolean = that <= this
}
```

示例 5.3.6 考虑抽象类 Table 实现了由键类型 A 到值类型 B 的映射。该类有一个方法 set 来将一个新的键值对放入表中，和方法 get 来返回与给定键值匹配的可选值。最后，有和 get 方法类似的方法 apply，只是如果表中没有给定键的定义该方法将会返回一个给定的默认值。该类实现如下：

```
abstract class Table[A, B](defaultValue: B) {
```

```
  def get(key: A): Option[B]
  def set(key: A, value: B)
  def apply(key: A) = get(key) match {
   case Some(value) => value
   case None => defaultValue
  }
}
```

以下是 Table 类的实际定义。

```
class ListTable[A, B](defaultValue: B) extends Table[A,
B](defaultValue){
  private var elems: List[(A,B)]
  def get(key: A) = elems.find(._1.==(key)).map(._2)
  def set(key: A, value: B) = { elems = (key, value) :: elems }
}
```

以下是一个特征来防止对父类中 get 和 set 操作的并发访问：

```
trait SynchronizedTable[A, B] extends Table[A, B] {
  abstract override def get(key: A): B =
   synchronized { super.get(key) }
  abstract override def set(key: A, value: B) =
   synchronized { super.set(key, value) }
}
```

注意 SychronizedTable 并没有传递给父类 Table 参数，即使 Table 定义了正式参数。同样注意到 SynchronizedTable 的 get 和 set 方法中对 **super** 的调用静态地引用了父类 Table 中的抽象方法。这是合法的，因为该方法标记为 **abstract override** (§5.2)。

最后，以下混入组合创建了一个同步的列表，以字符串作为键，以整数作为值，并定义 0 为缺省值。

```
object MyTable extends ListTable[String, Int](0) with SynchronizedTable
```

对象 MyTable 从 SynchronizedTable 中继承了 get 和 set 方法。这些方法中对 **super** 的调用与对应的 ListTable 中的对应方法重新绑定，实际就是 MyTable 中 SynchronizedTable 中的实际超类型。

5.4. 对象定义

语法：

```
ObjectDef   ::= id ClassTemplate
```

对象定义定义了一个新类的单个对象。最常用的形式是 object m **extends** t。这里 m 是要定义的对象的名称，t 是一个具有以下形式的模板(§5.1)

```
sc with mt1 with ... with mtn { stats }
```

此处定义了 m 的基类，行为以及初始状态。extends 子句 **extends** sc **with** mt_1 **with ... with** mt_n 可忽略，默认是 **extends** scala.AnyRef。类体 {stats}也可被忽略，默认为空{}。

对象定义定义了与模板 t 一致的单个对象(或：模块)。它大概等同于以下的三个定义，定义了一个类并按需创建了该类的单个对象。

```
final class m$cls extends t
private var m$instance = null
final def m = {
  if (m$instance == null) m$instance = new m$cls
  m$instance
}
```

如果该定义是代码块的一部分则这里的 **final** 修饰符可忽略。名称 m$cls 和 m$instance 不可从用户程序中访问。

注意到对象定义的值是懒加载的。构造器 **new** m$cls 并不是在对象定义时求值，而是在程序执行是 m 第一次被去引用时(可能永远也不会发生)。试图再次对构造器求值来重新去引用将导致死循环或运行时错误。

然而以上讨论并不能应用于顶级对象。不能这样的原因是变量和方法定义不能出现在顶级。而顶级对象将被翻译为静态字段。

示例 5.4.1 Scala 中的类没有静态成员；然而可以通过对象定义来达到等价的效果，例如：

```
abstract class Point {
  val x: Double
  val y: Double
  def isOrigin = (x == 0.0 && y == 0.0)
}

object Point {
  val origin = new Point() { val x = 0.0; val y = 0.0 }
}
```

这里定义了一个类 Point 和一个包含成员 origin 的对象 Point。注意两次使用名称 Point 是合法的，因为类定义在类型命名空间中定义了名称 Point，而对象定义在术语命名空间中定义了 Point

Scala 编译器在解释一个具有静态成员的 Java 类时使用了这种技术。这样的一个类 C 可以在概念上认为是包括所有的 C 的实例成员的一个 Scala 类，和包括所有 C 的静态成员的一个 Scala 对象的一对组合。

通常来讲，一个类的伴随模块是和类具有同样名称的一个对象，并定义在同样的作用域和编译单元中。同样地，这个类可以称作该模块的伴随类。

6. 表达式

语法：

```
Expr        ::= (Bindings | id | '_') '=>' Expr
                | Expr1
Expr1       ::= 'if' '(' Expr ')' {nl} Expr [[semi] else Expr]
            | 'while' '(' Expr ')' {nl} Expr
            | 'try' '{' Block '}' [catch '{'
                CaseClauses '}'] ['finally' Expr]
            | 'do' Expr [semi] 'while' '(' Expr ')'
            | 'for' ('(' Enumerators ')' |
                '{' Enumerators '}') {nl} ['yield'] Expr
            | 'throw' Expr
            | 'return' Expr
            | [SimpleExpr '.'] id '=' Expr
            | SimpleExpr1 ArgumentExprs '=' Expr
            | PostfixExpr
            | PostfixExpr Ascription
            | PostfixExpr 'match' '{' CaseClauses '}'
PostfixExpr ::= InfixExpr [id [nl]]
InfixExpr   ::= PrefixExpr
            | InfixExpr id [nl] InfixExpr
PrefixExpr  ::= ['-' | '+' | '~' | '!'] SimpleExpr
SimpleExpr  ::= 'new' (ClassTemplate | TemplateBody)
            | BlockExpr
            | SimpleExpr1 ['_']
SimpleExpr1 ::= Literal
            | Path
            | '_'
            | '(' [Exprs [','] ']'
            | SimpleExpr '.' ids
            | SimpleExpr TypeArgs
            | SimpleExpr1 ArgumentExprs
            | XmlExpr
Exprs       ::= Expr {',' Expr}
BlockExpr   ::= '{' CaseClauses '}'
            | '{' Block '}'
```

```
Block       ::= {BlockStat semi} [ResultExpr]
ResultExpr  ::= Expr1
              | (Bindings | (id | '_') ':' CompoundType) '=>' Block
Ascription  ::= ':' InfixType
              | ':' Annotation {Annotation}
              | ':' '_' '*'
```

表达式由算符和操作数构成。以下将按照以上顺序的降序来讨论表达式的形式。

6.1. 表达式类型化

表达式的类型化往往和某些期望类型有关(可能是没有定义的)。当我们说“表达式 e 期望与类型 T 一致”时，我们的意思是(1)e 的期望类型是 T，(2)表达式 e 的类型必须与 T 一致。

以下斯科伦化规则通用于所有表达式：如果一个表达式的类型是既存类型 T，那么表达式的类型就假定是 T 的斯科伦化(§3.2.10)。

斯科伦化由类型打包反转。假定类型为 T 的表达式 e，且 $t_1[tps_1]>:L_1<:U_1,...,t_n[tps_n]>:L_n<:U_n$ 是由 e 的一部分(在 T 中是自由的)的斯科伦化所创建的所有类型变量。e 的打包类型是

T **forSome** { **type** $t_1[tps_1]>:L_1<:U_1$; ...; **type** $t_n[tps_n]>:L_n<:U_n$ }

6.2. 字面值

语法：

```
SimpleExpr  ::= Literal
```

字面值的类型化如(§1.3)中所述；它们的求值是立即可得的。

字面值的另外一个形式指明类。形式如下：

```
classOf[C]
```

这里 classOf 是在 scala.Predef(§12.5)中定义的一个方法，C 是一个类类型。该类字面值的值是类类型 C 的运行时表示。

6.3. Null 值

Null 值的类型是 scala.Null，且与所有引用类型兼容。它表示一个指向特殊”**null**”对象的引用值。该对象对类 scala.AnyRef 中的方法的实现如下：

- eq(x)和==(x)返回 **true** 的条件是 x 同样也是一个”**null**”对象
- ne(x)和!=(x)返回 **true** 的条件是 x 不是一个”**null**”对象
- isInstanceOf[T]总返回 **false**
- asInstanceOf[T]返回”**null**”对象的条件是 T 与 scala.AnyRef 一致，否则会抛出 NullPointerException。

“**null**”对象对任何其他成员的引用将导致抛出一个 NullPointerException。

6.4. 指示器

语法：

```
SimpleExpr  ::= Path
              | SimpleExpr '.' id
```

指示器指向一个命名术语。可以是一个简单命名或一个选择。

简单命名 x 指向(§2)中所表示的一个值。如果 x 由一个封闭类或对象 C 中的定义或声明绑定，那么它将等价于选择 C.**this**.x，C 指向包含 x 的类，即使类型名 C 在 x 出现时是是被遮盖的(§2)。

如果 r 是类型 T 的稳定标识符(§3.1)，则选择 r.x 静态指向 r 的在 T 中以命名 x 标识的术语成员 m。

对于其他表达式 e，e.x 以{ **val** y = e; y.x }的形式类型化，y 是一个新命名。代码块的类型化规则暗示在此情况下 x 的类型可能并不指向 e 的任何抽象类型成员。

指示器前缀的期望类型总是未定义的。指示器的类型是其指向的实体的类型 T，但以下情况除外：在需要稳定类型(§3.2.1)的上下文中路径(§3.1)p 的类型是单态类型 p.**type**。

需要稳定类型的上下文需要满足以下条件：

1. 路径 p 以一个选择前缀的形式出现，且并不表示一个常量，或
2. 期望类型 pt 是一个稳定类型，或
3. 期望类型 pt 是以一个稳定类型为下界的抽象类型，且 p 指向的实体的类型 T 与 pt 不一致，或
4. 路径 p 指向一个模块

选择 e.x 在限定表达式 e 第一次求值时求值，同时产生一个对象 r。选取的结果是 r 的成员要么由 m 定义或由重载 m 的定义所定义。如果该成员具有与 scala.NotNull 一致的类型，该成员的值必须初始化为与 null 所不同的值，否则将抛出 scala.UnitializedError。

6.5. This 和 Super

语法：

```
SimpleExpr  ::= [id '.'] 'this'
              | [id '.'] 'super' [ClassQualifier] '.' id
```

表达式 **this** 可以出现在作为模板或复合类型的语句部分中。它表示由最里层的模板或最靠近引用的复合类型所定义的对象。如果是一个复合类型，那么 **this** 的类型就是该复合类型。如果是一个实例创建表达式的模板，**this** 的类型就是该模板的类型。如果是一个有简单命名 C 的类或对象定义的模板，**this** 的类型与 C.**this** 的类型相同。

表达式 C.**this** 在具有简单命名 C 的封闭类或对象定义的语句部分中是合法的。它表示由最里层该定义所定义的对象。如果表达式的期望类型是一个稳定类型，或 C.**this** 以一个选择前缀的形式出现，那么它的类型就是 C.**this**.**type**，否则就是 C 自己的类型。

引用 **super**.m 静态地引用包含该引用的最里层模板的最小合理超类型的方法或类型 m。它求值的结果等价于 m 或重载 m 的该模板的实际超类型的成员 m′。被静态引用的成员 m 必须是类型或方法。如果是方法则必须是实体方法，如果是模板则必须是包含拥有重载了

m 且标记为 **abstract override** 的成员 m′的引用。

引用 C.**super**.m 静态地引用包含该引用的最里层命名为 C 的封闭类或对象的定义中最小合理超类型的方法或类型 m。该引用的求值为该类或对象的实际超类型中等价于 m 或重载了 m 的成员 m′。如果静态引用的成员 m 是一个方法，那么就必须是一个实体方法，或者最内层名为 C 的封闭类或对象定义必须有一个重载了 m 且标记为 **abstract override** 的成员 m′。

前缀 **super** 可以后跟特征限定[T]，比如 C.**super**[T].x。这叫做静态超引用。在此情况下该引用指向具有简单名称 T 的 C 的父特征中的类型或方法 x。该成员必须具有唯一定义。如果这是一个方法，则该方法必须是实体的。

示例 6.5.1 考虑以下类定义

```
class Root { def x = "Root" }
class A extends Root {
 override def x = "A";
 def superA = super.x;
}
trait B extends Root {
  override def x = "B";
 def superb = super.x;
}
class C extends Root with B {
  override def x = "C";
  def superC = super.x;
}
class D extends A with B {
  override def x = "D";
  def superD = super.x;
}
```

类 C 的线性化为{C, B, Root}，类 D 的线性化为{D, B, A, Root}。那么我们有：

```
(new A).superA == "Root"
(new C).superB == "Root"
(new C).superC == "B"
(new D).superA == "Root"
(new D).superB == "A"
(new D).superD == "B"
```

要注意到 superB 函数根据 B 与类 Root 或 A 混用将返回不同的值。

6.6. 函数应用

语法：

```
SimpleExpr      ::= SimpleExpr1 ArgumentExprs
ArgumentExprs   ::= '(' [Exprs [',']] ')'
                  | '(' [Exprs ','] PostfixExpr ':' '_' '*' ')'
```

```
                | [nl] BlockExpr
Exprs           ::= Expr {‘,’ Expr}
```

应用 f(e1,..,en)将函数 f 应用于参量表达式 e1,...,en。如果 f 具有方法类型(T1,...,Tn)U，则每个参量表达式 ei 的类型必须与对应的参数类型 Ti 一致。如果 f 具有值类型，该应用则等价于 f.apply(e1,..,en)，例如应用一个 f 定义的 apply 方法。

f(e1,...,en)的求值通常必须按照 f 和 e1,...,en 的顺序来进行。每个参量表达式将会被化为其对应的正式参数的类型。在此之后，该应用将会写回到函数的右侧，并用真实参量来替换正式参数。被重写的右侧的求值结果将最终转变为函数声明的结果类型(如果有的话)。

函数应用通常会在程序运行时堆中定位一个新帧。然而如果一个本地函数或者一个 final 方法的最后动作是调用其自身的话，该调用将在调用者的堆栈帧中执行。

对于具有无参方法类型=>T 的正式参数将会做特殊处理。该情况下，对应的实际参量表达式并不会在应用前求值。相反地，重写规则中每次在右侧使用正式参数时都将会重新对 e 求值。换句话说，对=>-参数的求值顺序是叫名的，而对于普通参数是传值的。同时 e 的打包类型(§6.1)必须与参数类型 T 一致。

应用中最后一个参数可以标记为序列参数，例如 e:_*。这样的一个参数必须与一个类型 S*的重复参数(§4.6.2)一致，也必须是唯一与该参数匹配的参量(比如正式参数与实际参量必须在数目上匹配)。更进一步的，对于与 S 一致的某些类型 T，e 的类型必须与 scala.Seq[T]一致。在此情况下，参数类型的最终形式是用其元素来替换序列 e。

示例 6.6.1 设有以下函数来计算参数的总和：

```
def sum(xs: Int*) = (0 /: xs)((x, y) => x + y)
```

那么

```
sum(1,2,3,4)
sum(List(1,2,3,4): _*)
```

都会得到结果 10.然而

```
sum(List(1,2,3,4))
```

却无法通过类型检查。

6.7. 方法值

语法：

```
SimpleExpr  ::= SimpleExpr1 ‘_’
```

表达式 e _在 e 是方法类型或 e 是一个叫名参数的情况下是正确的。如果 e 是一个具有参数的方法，e _ 表示通过 eta 展开(§6.25.5)得到的一个函数类型。如果 e 是一个无参方法或者有类型=>T 的叫名参数，e _ 表示类型为() => T 的函数，且将在应用于空参数列表()时对 e 求值。

示例 6.7.1 左列的方法值将对应等价于其右侧的匿名函数(§6.23)

```
Math.sin _              x => Math.sin(x)
```

```
Array.range _          (x1, x2) => Array.range(x1, x2)
List.map2 _            (x1, x2) => (x3) => List.map2(x1, x2)(x3)
List.map2(xs, ys)_     x=> List.map2(xs, ys)(x)
```

要注意到在方法名和其后跟下划线间必须要有空格，否则下划线将会被认为是方法名的一部分。

6.8. 类型应用

语法：

```
SimpleExpr  ::= SimpleExpr TypeArgs
```

类型应用 e[T_1,...,T_n]实例化了具有类型[a_1 >: L_1 <: U_1,...,a_n >: L_n <: U_n]S 和参量类型 T_1,...,T_n 的多态值 e。每个参量类型 T_i 必须符合对应的边界 L_i 和 U_i。也就是对于每个 i=1,...,n，我们必须有 pL_i <: T_i <: pU_i，这里 p 是[a_1:=T_1,...,a_n:=T_n]的指代。应用的类型是 pS。

如果函数部分 e 是某种值类型，该类型应用将等价于 e.apply[T_1,...,T_n]，比如由 e 定义的 apply 方法的应用。

如果本地类型推断(§6.25.4)可以通过实际函数参量类型和期望结果类型来得到一个多态函数的最佳类型参数，则类型应用可以忽略。

6.9. 元组

语法：

```
SimpleExpr  ::= '(' [Exprs [','']] ')'
```

元组表达式((e_1,...,e_n)是类实例创建 scala.Tuple$n$(e_1,...,e_n)的别名(n>=2)。该表达式后面还可以有个分号，例如(e_1,...,e_n,)。空元组()是类型 scala.Unit 的唯一值。

6.10. 实例创建表达式

语法：

```
SimpleExpr  ::= 'new' (ClassTemplate | TemplateBody)
```

一个简单的实例创建表达式具有形如 **new** c 的形式，c 是一个构造器调用(§5.1.1)。设 T 为 c 的类型，那么 T 必须表示 scala.AnyRef 的一个非抽象子类(的类型实例)。更进一步，表达式的*固实自类型*必须与 T 表示的类型的自类型一致(§5.1)。固实自类型通常为 T，一个特例是表达式 **new** c 在值定义的右侧出现

```
val x: S = new c
```

(类型标注：S 可能没有)。在此情况下，表达式的固实自类型是复合类型 T **with** x.**type**。

该表达式的求值方式是通过创建一个类型为 T 的新对象并以对 C 求值来初始化。表达

式的类型为 T。

对于某些类模板 t(§5.1)，一个常见的实例创建表达式具有 **new** t 的形式。这样的表达式等价于代码块

```
{ class a extends t; new a }
```

a 是*匿名类*的一个新名称。

创建结构化类型的值的快捷方式为：如果{D}是一个类体，则 **new** {D}就等价于通用实例创建表达式 **new** AnyRef{D}

示例 6.10.1 考虑以下结构化实例创建表达式

```
new { def getName() = "aaron" }
```

这是以下通用实例创建表达式的简写形式

```
new AnyRef{ def getName() = "aaron" }
```

后者则是以下代码块的简写：

```
{
 class anon$X extends AnyRef{ def getName() = "aaron" };
 new anon$X;
}
```

这里 anon$X 是某个新创建的名称。

6.11. 代码块

语法：

```
BlockExpr   ::= '{' Block '}'
Block       ::= {BlockStat semi} [ResultExpr]
```

代码块表达式{s_1;...;s_n;e}由一个代码块语句序列 s_1,...,s_n 和一个最终表达式 e 构成。语句序列中不能有两个定义或声明绑定到同一命名空间的同一命名上。最终表达式可忽略，默认为单元值()。

最终表达式 e 的期望类型是代码块的期望类型。所有前面的语句的期望类型是未定义的。

代码块 s_1;...;s_n;e 的类型是 T **forSome** {Q}，T 是 e 的类型，Q 包括在 T 中每个自由的和在语句 s1,...,sn 中局部定义值或类型命名的既存类型(§3.2.10)。我们说存在子句*绑定*了值或者类型命名。需要特别指出的：

- 一个本地定义的类型定义 **type** t[tps]=T 由存在子句 clause **type**[tps]>:T<:T 绑定
- 一个本地定义的值定义 **val** x:T=e 由存在子句 **val** x:T 绑定
- 一个本地定义的类定义 **class** c[tps] **extends** t 由存在子句 **type** c[tps]<:T 绑定，T 是最小类类型或修饰类型，且是类型 c[tps]的合适超类。
- 一个本地定义的对象定义 **object** x **extends** t 由存在子句 **val** x:T 绑定，T 是最小类类型或修饰类型，且是类型 x.**type** 的合适超类。

对代码块求值需要对其语句序列求值，然后对最终表达式 e 求值，该表达式定义了代

码块的结果。

示例 6.11.1 假定有类 Ref[T](x: T)，代码块

```
{ class C extends B {...} ; new Ref(new C) }
```

具有类型 Ref[_1] **forSome** { **type** _1 <: B }. 代码块

```
{ class C extends B {...} ; new C }
```

的类型仅是 B，因为(§3.2.10)中的规则有存在限定类型 _1 **forSome** { **type** _1 <: B }可简化为 B。

6.12. 前缀，中缀及后缀运算

语法：

```
PostfixExpr ::= InfixExpr [id [nl]]
InfixExpr   ::= PrefixExpr
              | InfixExpr id [nl] Inf2424ixExpr
PrefixExpr  ::= ['-' | '+' | '!' | '~'] SimpleExpr
```

表达式由算符和操作数构成。

6.12.1. 前缀运算

前缀运算 op e 由前缀算符 op(必须是'+', '-', '!'或'~'之一)。表达式 op e 等价于后缀方法应用 e.unary_op。

前缀算符不同于普通的函数应用，他们的操作数表达式不一定是原子的。例如，输入序列-sin(x)读取为-(sin(x))，函数应用 negate sin(x)将被解析为将中缀算符 sin 应用于操作数 negate 和(x)。

6.12.2. 后缀操作

后缀算符可以是任意标识符。后缀操作 e op 被解释为 e.op。

6.12.3. 中缀操作

中缀算符可以是任意标识符。中缀算符的优先级和相关性定义如下：

中缀算符的优先级由算符的第一个字符确定。字符按照优先级升序在下面列出，同一行中的字符具有同样的优先级。

```
(所有字母)
|
^
&
< >
= !
:
```

```
+ -
* / %
(所有其他特殊字符)
```

也就是说，由字母开头的算符具有最低的优先级，然后是由‘|’开头的算符，下同。

这个规则中有一个例外，就是赋值算符(§6.12.4)。赋值算符的优先级与简单赋值(=)相同。也就是比任何其他算符的优先级要低。

算符的相关性由算符的最后一个字符确定。由‘:’结尾的算符是右相关的。其他所有算符是左相关的。

算符的优先级和相关性确定了表达式部件结组的方式：

- 如果表达式中有多个中缀运算，那么具有高优先级的算符将比优先级低的绑定的更紧。
- 如果具有连贯的中缀运算 $e_0\ op_1\ e_1\ op_2...op_n\ e_n$，且算符 $op_1,...,op_n$ 具有同样的优先级，那么所有的这些算符将具有同样的相关性。如果所有算符都是左相关的，该序列将解析为 $(...(e_0\ op_1\ e_1)\ op_2...)\ op_n\ e_n$。否则，如果所有算符都是右相关的，则该序列将解析为 $e_0\ op_1(e_1\ op_2\ (...op_n\ e_n)...)$
- 后缀算符的优先级总是比中缀算符低。例如 $e_1\ op_1\ e_2\ op_2$ 总是等价于 $(e_1\ op_1\ e_2)\ op_2$。

左相关算符的右侧操作数可以由在括号中的几个参数组成，例如 $e\ op(e_1,...,e_n)$。该表达式将被解析为 $e.op(e_1,...,e_n)$。

左相关位运算 $e_1\ op\ e_2$ 解析为 $e_1.op(e_2)$。如果 op 是右相关的，同样的运算将被解析为 $\{\ \textbf{val}\ x=e_1;\ e_2.op(x)\ \}$，这里 x 是一个新的名称。

6.12.4. 赋值算符

赋值算符是一个由等号“=”结尾的算符记号((§1.1)中的语法类 op)，但具有以下条件的算符除外：

(1) 算符也由等号开始，或

(2) 算符是(<=), (>=), (!=)中的一个

赋值算符做特殊处理，如果没有其它有效的解释，则扩展为赋值。

我们考虑一个赋值算符，比如+=。在中缀运算 l += r 中，l 和 r 是表达式。该运算可以重新解释为负责赋值的运算

```
l = l + r
```

除非该运算的做的 l 只计算一次。

在以下两种条件下会发生再解析。

1. 左侧的 l 没有一个名为+=的成员，且不能由隐式转换(§6.25)转换为拥有成员+=的值。
2. 赋值运算 l = l + r 是类型正确的。特别此处暗含了 l 引用了一个变量或对象，且该变量或对象可以赋值，且可转变为一个具有名为+的成员的值。

6.13. 类型化的表达式

语法：

```
Expr1   ::= PostfixExpr ‘:’ CompoundType
```

类型化的表达式 e:T 具有类型 T。表达式 e 的类型被期望与 T 一致。表达式的结果就是 e 的值转化为类型 T。

示例 6.13.1 以下是合法与非法类型化的表达式的例子

```
1: Int          //合法，类型为 Int
1: Long         //合法，类型为 Long
//1: string     //*****非法
```

6.14. 标注表达式

语法：

```
Expr1   ::= PostfixExpr ‘:’ Annotation {Annotation}
```

标注表达式 e: @a_1 ... @a_n 将标注 a_1,...,a_n 附在表达式 e(§11)上。

6.15. 赋值

语法：

```
Expr1   ::= [SimpleExpr ‘.’] id ‘=’ Expr
          | SimpleExpr1 ArgumentExpr ‘=’ Expr
```

对一个简单变量的赋值 x = e 的解释依赖于 x 的定义。如果 x 是一个可变量，那么赋值将把当前 x 的值变为对表达式 e 求值所得的结果。e 的类型被期望与 x 的类型一致。如果 x 是某些模板中定义的无参数函数，且该模板中包括一个 setter 函数 x_=成员，那么赋值 x = e 就解释为对该 setter 函数的调用 x_=(e)。类似地，赋值 f.x = e 应用于一个无参函数 x 就解释为调用 f.x_=(e)。

赋值 f(args) = e 中=算符左侧的函数应用解释为 f.update(args, e)，例如对由 f 定义的 update 函数的调用。

示例 6.15.1 以下是矩阵乘法中的一些常用代码

```
def matmul(xss: Array[Array[Double]], yss: Array[Array[Double]]) = {
 val zss: Array[Array[Double]] = new Array(xss.length, yss(0).length)
 var I = 0
 while (I < xss.length) {
  var j = 0
  while (j < yss(0).length) {
   var acc = 0.0
   var k = 0
   while (k < yss.length){
    acc = acc + xss(i)(k) * yss(k)(j)
     k += 1
   }
   zss(i)(j) = acc
   j += 1
  }
```

```
    i += 1
  }
  zss
 }
```

去掉数据访问和赋值的语法糖，则是下面这个扩展的版本：

```
 def matmul(xss: Array[Array[Double]], yss: Array[Array[Double]]) = {
  val zss: Array[Array[Double]] = new Array(xss.length,
yss.apply(0).length)
  var I = 0
  while (I < xss.length) {
    var j = 0
    while (j < yss.apply(0).length) {
      var acc = 0.0
      var k = 0
      while (k < yss.length){
        acc = acc + xss.apply (i) .apply (k) * yss.apply (k) .apply (j)
        k += 1
      }
      zss.apply (i).update(j, acc)
      j += 1
    }
    i += 1
  }
  zss
 }
```

6.16. 条件表达式

语法：

```
Expr1   ::= 'if' '(' Expr ')' {nl} Expr [[semi] 'else' Expr]
```

条件表达式 **if** (e_1) e_2 **else** e_3 根据 e_1 的值来选择值 e_2 或 e_3。条件 e_1 期望与类型 Boolean 一致。Then 部分 e_2 和 else 部分 e_3 都期望与条件表达式的期望类型一致。条件表达式的类型是 e_2 和 e_3 的类型的最小上界。else 前的分号会被忽略。

条件表达式的求值中首先对 e_1 求值。如果值为 **true**，则返回 e_2 求值的结果，否则返回 e_3 求值的结果。

条件表达式的一种简单形式没有 else 部分。条件表达式 **if** (e_1) e_2 求值方式为 **if** (e_1) e_2 **else** ()。该表达式的类型是 Unit，且 then 部分 e_2 也期望与类型 Unit 一致。

6.17. While 循环表达式

语法：

```
Expr1       ::= 'while' '(' Expr ')' {nl} Expr
```

While 循环表达式 **while**(e_1)e_2 的类型化与求值方式类似于函数 whileLoop (e_1)(e_2)的应用，假定的函数 whileLoop 定义如下：

```
def whileLoop(cond: => Boolean)(body: => Unit): Unit =
    if (cond) { body ; whileLoop(cond)(body) } else {}
```

6.18. Do 循环表达式

语法：

```
Expr1       ::= 'do' Expr [semi] 'while' '(' Expr ')'
```

Do 循环表达式 **do** e_1 **while** (e_2)的类型化与求值方式类似于表达式 (e_1 ; **while** (e_2) e_1)。Do 循环表达式中 **while** 前的分号被忽略。

6.19. For 语句段

语法：

```
Expr1       ::= 'for' ('(' Enumerators ')' | '{' Enumerators
                    '}') {nl} ['yield'] Expr
Enumerators ::= Generator {semi Enumerator}
Enumerator  ::= Generator
            | Guard
            | 'val' Pattern1 '=' Expr
Generator   ::= Pattern1 '<-' Expr [Guard]
Guard       ::= 'if' PostfixExpr
```

for 语句段 **for** (enums) **yield** e 对于由枚举器 enums 产生的每个绑定求值表达式 e。一个枚举器序列总是由一个产生器开始；然后可跟其他产生器，值定义，或守卫。一个产生器 p <- e 从一个与模式 p 匹配的表达式 e 产生绑定。值定义 **val** p = e 将值名称 p(或模式 p 中的数个名称)绑定到表达式 e 的求值结果上。守卫 **if** e 包含一个布尔表达式，限制了枚举出来的绑定。产生器和守卫的精确含义通过翻译为四个方法的调用来定义：map filter flatMap 和 foreach。这些方法可以针对不同的携带类型具有不同的实现。

翻译框架如下。在第一步里，每个产生器 p <- e，对于 e 的类型被替换为如下形式，p 不是不可反驳的(§8.1)：

```
p <- e.filter { case p => true; case _ => false }
```

然后，以下规则将重复应用，直到所有的语句段都消耗完毕。

- for 语句段 **for** (p <- e) **yield** e′被翻译为 e.map { **case** p => e′ }
- for 语句段 **for** (p <- e) e′ 被翻译为 e.foreach { **case** p => e′ }
- for 语句段

  for (p <- e; p′ <- e′ ...) **yield** e′′,

  这里...是一个产生器或守卫序列(可能为空)，该语句段翻译为

  e.flatMap { **case** p => **for**(p′ <- e′ ...) **yield** e′′ }
- for 语句段

  for (p <- e; p′ <- e′ ...) e′′

这里... 是一个产生器或守卫序列(可能为空)，该语句段翻译为

```
e.foreach { case p => for (p′ <- e′ ...) e′′ }
```

- 后跟守卫 **if** g 的产生器 p <- e 翻译为单个产生器 p <- e.filter((x1,...,xn) => g)，这里 x1,...,xn 是 p 的自由变量。
- 后跟值定义 **val** p′ = e′的产生器 p <- e 翻译为以下值对产生器，这里的 x 和 x′是新名称：

```
val (p, p′) <-
  for (x@p <- e) yield { val x′@p′ = e′; (x, x′) }
```

示例 6.19.1 以下代码产生 1 到 n-1 间所有和为素数的数值对

```
for { i <- 1 until n
 j <- 1 until i
 if isPrime(i+j)
} yield (i, j)
```

该 for 语句段翻译为：

```
(1 until n)
   .flatMap {
       case i => (1 until i)
           .filter { j => isPrime(i+j) }
           .map { case j => (i, j) }
```

示例 6.19.2 for 语句段可以用来简明地描述向量和矩阵算法。比如以下就是一个函数来计算给定矩阵的转置：

```
def transpose[A](xss: Array[Array[A]]) = {
  for (i <- Array.range(0, xss(0).length)) yield
    for (xs <- xss) yield xs(i)
}
```

以下是一个函数，用来计算两个向量的无向量积：

```
def scalprod(xs: Array[Double], ys: Array[Double]) = {
  var acc = 0.0
  for ((x, y) <- xs zip ys) acc = acc + x * y
  acc
}
```

最后，这是一个求两个矩阵的积的函数。可以与示例 6.15.1 中的常见版本做一个比较

```
def matmul(xss: Array[Array[Double]], yss: Array[Array[Double]] = {
  val ysst = transpose(yss)
  for (xs <- xss) yield
    for (yst <- ysst) yield
      scalprod(xs, yst)
}
```

以上代码使用了类 scala.Array 中已有定义的成员 map, flatMap, filter 和 foreach。

6.20. Return 表达式

语法：

```
Expr1   ::= 'return' [Expr]
```

return 表达式 **return** e 必须出现在某些封闭的命名方法或函数体内。源程序中最里层的封闭命名方法或函数 f 必须有一个显式声明的结果类型，e 的类型必须与其一致。return 表达式求值表达式 e 并返回其值作为 f 的结果。任何 return 表达式之后的语句或表达式将忽略求值。return 表达式的类型是 scala.Nothing。表达式 e 可以没有。表达式 return 以 **return** ()的形式做类型检查和求值。

由编译器生成的 apply 方法，作为匿名函数的扩展，并不能作为源程序中的命名函数，因此不是 return 表达式的目标。

如果 return 表达式自身是匿名函数的一部分，可能在 return 表达式被求值前 f 的封闭实体就已经返回了。在此情况下会抛出 scala.runtime.NonLocalReturnException 异常。

6.21. Throw 表达式

语法：

```
Expr1   ::= 'throw' Expr
```

throw 表达式 **throw** e 对表达式 e 求值。该表达式的类型必须与 Throwable 一致。如果 e 求值的结果是异常的引用，则求值结束，抛出该异常。如果 e 求值的结果是 **null**，则求值结束并抛出 NullPointerException。如果此处有一个活动的 try 表达式(§6.22)，且要处理抛出的异常，则求值在该处理器中继续进行；否则执行 throw 的线程将被终止。throw 表达式的类型是 scala.Nothing。

6.22. Try 表达式

语法：

```
Expr1   ::= 'try' '{' Block '}' ['catch' '{' CaseClauses '}']
            ['finally' Expr]
```

Try 表达式具有 **try** { b } **catch** h 的形式，处理器 h 是能匹配以下匿名函数(§8.5)的模式

$$\{\ \textbf{case}\ p_1 \Rightarrow b_1\ \ldots\ \textbf{case}\ p_n \Rightarrow b_n\}$$

该表达式的求值方式是对代码块 b 求值。如果 b 的求值并没有导致抛出异常，则返回 b 的结果。否则处理器 h 将应用于抛出的异常。如果处理器包含一个 case 与抛出的异常匹配，则调用第一个该类 case。如果没有与抛出的异常匹配的 case，则异常被重新抛出。

设 pt 是 try 表达式的期望类型。代码块 b 被期望与 pt 一致。处理器 h 期望与类型

scala.PartialFunction[scala.Throwable, pt]一致。try 表达式的类型是 b 的类型与 h 的结果类型的最小上界。

try 表达式 **try**{ b } **finally** e 首先对代码块 b 求值。如果在求值中没有导致异常抛出，则对表达式 e 求值。如果在对 e 求值中有异常抛出，则 try 表达式的求值终止并抛出异常。如果在对 e 求值时没有异常抛出，则返回 b 的结果作为 try 表达式的结果。

如果在对 b 求值时有异常抛出，finally 代码块 e 同样也会被执行。如果在对 e 求值时有另外一个异常被抛出，则 try 表达式求值终止，同时抛出该异常。如果在对 e 求值时没有异常抛出，b 中抛出的异常在 e 的求值终止时被重新抛出。代码块 b 被期望与 try 表达式所期望的类型一致。finally 表达式 e 被期望与类型 Unit 一致。

Try 表达式 **try** { b } **catch** e_1 **finally** e_2 是 **try** { **try** { b } **catch** e_1 } **finally** e_2 的简写。

6.23. 匿名函数

语法：

```
Expr        ::= (Bindings | Id | '_') '=>' Expr
ResultExpr  ::= (Bindings | (Id | '_') ':' CompoundType) '=>' Block
Bindings    ::= '(' Binding {',' Binding} ')'
Binding     ::= (id | '_') [':' Type]
```

匿名函数$(x_1: T_1,...,x_n: T_n)$ => e 将类型为 T_i 的参数 x_i 映射为由表达式 e 给出的结果。每个正式参数 x_i 的作用域是 e。正式参数必须具有两两不同的名称。

如果匿名函数的期望类型具有 scala.Function*n*$[S_1,...,S_n, R]$的形式，则 e 的期望类型是 R，每个参数 x_i 的类型 T_i 可忽略，可假定 $T_i = S_i$。如果匿名函数的期望类型是某些其他类型，则所有正式参数的类型都必须显式的给出，且 e 的期望类型是未定义的。匿名函数的类型是 scala.Function*n*$[S_1,...,S_n, T]$，这里 T 是 e 的打包类型(§6.1)。T 必须等价于一个不引用任何正式参数 x_i 的类型。

匿名函数求值方式为实例创建表达式

```
new scala.Functionn[T1,...,Tn, T] {
  def apply(x1: T1,...,xn: Tn): T = e
}
```

在具有单个未类型化的正式参数时，(x) => e 可缩写为 x => e。如果匿名函数(x: T) => e 有单个类型化的参数作为一个代码块的结果表达式出现，则可缩写为 x: T => e。

一个正式参数也可以是由一个下划线 _ 表示的通配符。在此情况下可以随意选择该参数的一个新名称。

示例 6.23.1 匿名函数的例子

```
x => x                              //恒等函数
f => g => x => f(g(x))              //柯里化的函数组合
(x: Int, y: Int) => x + y           //求和函数
() => { count +=1; count }          //该函数参数列表为空
                                    //将一个非本地变量'count'加 1
                                    //并返回新的值
```

```
_ => 5                          //该函数忽略其所有参数并总是返回 5
```

匿名函数的占位符语法

语法：

```
SimpleExpr1 ::= '_'
```

一个表达式(语法上归类为 Expr)可以在合法的标识符处包含内嵌的下划线记号 _。这样一个表达式表示一个下划线后的位置的连续的参数的匿名函数。

定义下划线段具有形式为_:T 的表达式，T 是一个类型，或者形式为_，下划线并不是类型归属_:T 的表达式部分。

具有 Expr 句法归类的表达式 e 绑定到下划线段 u 的两个条件是：(1) e 合理包含 u，且(2)没有其他的具有 Expr 句法归类的表达式合理包含于 e 且其自身合理包含 u。

如果表达式 e 按照既定顺序绑定到下划线段 $u_1,...,u_n$，则等价于匿名函数 $(u'_1,...,u'_n)$=> e′，每个 u'_i 是将 u_i 中下划线替换为新的标识符的结果，e′则是将 e 中每个下划线段 u_i 替换为 u'_i 的结果。

示例 6.23.2 左侧的匿名函数使用了占位符语法。每个都等价于其右侧的匿名函数

```
_ + 1               x => x + 1
_ * _               (x1, x2) => x1 * x2
(_: Int) *2         (x: Int) => (x: Int) * 2
if (_) x else y     z => if (z) x else y
_.map(f)            x => x.map(f)
_.map(_ + 1)        x => x.map(y => y + 1)
```

6.24. 语句

语法：

```
BlockStat       ::= Import
                | ['implicit'] Def
                | {LocalModifier} TmplDef
                | Expr1
                |
TemplateStat    ::= Import
                | {Annotation} {Modifier} Def
                | {Annotation} {Modifier} Del
                | Expr
                |
```

语句是代码块和模板的一部分。语句可以是 import，定义或表达式，也可为空。在模板或类定义中使用的语句还可以是声明。作为语句来使用的表达式可以有任意的值类型。表达式语句 e 的求值是对表达式 e 求值然后丢弃求值的结果。

代码块语句可以是在代码块中绑定本地命名的定义。代码块本地定义中允许的修饰符是类或对象定义前的 **abstract, final, sealed**。

语句序列的求值是语句按照它们书写顺序的求值。

6.25. 隐式转换

隐式转换可以应用于那些类型与期望类型不一致的表达式或未应用的方法。在接下来的两小节中将给出可用的隐式转换。

如果 T 与 U 在应用 eta 扩展(§6.25.5)和视图应用(§7.3)后一致，则 T 与 U 是兼容的。

6.25.1. 值转换

以下的五个隐式转换可以应用到具有类型 T，且对某些期望类型 pt 做了类型检查的表达式 e。

重载解析 如果一个表达式表示某类的数个可能的成员，应用重载解析(§6.25.3)可以选定唯一的成员。

类型实例化 表达式 e 具有多态类型

$$[a_1 >: L_1 <: U_1,...,a_n >: L_n <: U_n]T$$

并不作为类型应用的函数部分，根据类型变量 a1,...,an 通过本地类型推断(§6.25.4)来确定实例类型 $T_1,...,T_n$，并隐式将 e 嵌入类型应用 $e[T_1,...,T_n]$(§6.8)的方式转换为 T 的类型实例。

数字字面值缩减 如果期望类型是 Byte，Short 或者 Char，且表达式 e 是一个符合该类型范围的整数字面值，则将被转换为该类型同样的字面值。

值丢弃 如果 e 具有值类型且期望类型是 Unit，则 e 通过嵌入术语{ e; () }的方式转换为期望类型。

视图应用 如果没有应用以上任何转换，且 e 的类型与期望类型 pt 不相似，则将通过视图(§7.3)尝试将 e 转换为期望的类型。

6.25.2. 方法转换

以下四个隐式转换可应用到那些无法应用到指定参数列表的方法上面。

求值 具有类型=> T 的无参数方法 m 总是通过对 m 绑定的表达式求值来转换到类型 T

隐式应用 如果该方法只接受隐式参数，则将通过规则§7.2 传递隐式参量。

eta 扩展 否则，如果方法不是一个构造器，且期望类型是一个函数类型(Ts′)=>T′，则 eta-扩展应用于表达式 e。

空应用 否则，如果 e 拥有方法类型()T，则将隐式应用于空参数列表，产生 e()。

6.25.3. 重载解析

如果一个标识符或选择 e 引用了数个类的成员，则将使用引用的上下文来推断唯一的成员。使用的方法将依赖于 e 是否被用作一个函数。设 A 是 e 引用的成员的集合。

首先假定 e 作为函数出现在应用中，比如 e(args)。如果在 A 中有且仅有一个可选成员是一个(可能是多态)方法类型，其元数与给出的参量数目匹配，则就会选定该可选成员。

否则，设 Ts 是通过用未定义类型来类型化每个参量所得到的类型向量。首先要确定的是可用的可选成员的集合。如果 Ts 中每个类型都与对应的可选成员中正式参数类型相似，且如果期望类型已定义，方法的结果类型与之兼容，则该可选项是可用的。对于一个多态方法，如果本地类型推断可以确定类型参量，则该实例化的方法是可用的，继而该多态方法也是可用的。

设 B 是可用的可选项的集合。如果 B 为空则导致错误。否则可以用以下"同样具体"和"更具体" 的定义来选出在 B 中最具体的可选项：

- 具有类型(Ts)U 的参数化的方法，如果有某些类型为 S 的其他成员，S 对于类型 Ts 的参量(ps)是可用的，则该方法与这些成员同样具体。
- 具有类型[a_1 >: L_1 <: U_1,...,a_n >: L_n <: U_n]T 的多态方法，如果有某些类型为 S 其他成员，如果假定对于 i=1,...,n，每个 a_i 都是一个抽象类型命名，其边界在 L_i 之上且在 U_i 之下，有 T 和 S 同样具体，则该方法和这些成员同样具体。
- 具有其他类型的成员总是与一个参数化的方法或一个多态方法同样具体。
- 给定具有类型 T 和 U 的两个没有参数化也不是多态方法类型的成员，类型为 T 的成员与类型为 U 的成员同样具体的条件是 T 的双重存在与 U 的双重存在相似。这里多态类型[a_1 >: L_1 <: U_1,...,a_n >: L_n <: U_n]T 的双重存在是 T **forSome** { **type** a_1 >: L_1 <: U_1,...,**type** a_n >: L_n <: U_n}。其他类型的双重存在是类型自身。

如果 A 与 B 同样具体，同时要么 B 与 A 不同样具体，要么 A 在 B 的一个子类中定义，则 A 比 B 更具体。

如果 B 中没有可选项比 B 中其他可选项更具体则将导致错误。

下面假定 e 以函数的形式在类型应用中出现，比如 e[targs]。那么我们将选择 A 中所有的与 targs 中的类型参量具有同样数目的参数类型的可选项。如果没有该类可选项将导致错误。如果有多个这样的可选项，则将对整个表达式 e[targs]重新应用重载解析。

最后我们假定 e 没有在应用或类型应用中做为函数出现。如果给出了期望类型，设 B 是 A 中与其兼容(§6.25)的该类可选项的集合。否则，设 B 为 A。在此情况下我们在 B 的所有可选项中选择最具体的可选项。如果 B 中没有可选项比 B 中其他所有的可选项更具体则将导致错误。

在所有情况下，如果最具体的可选项定义在类 C 中，且有另外一个可应用的可选项定义在 C 的子类中，则将导致错误。

示例 6.25.1 考虑以下定义：

```
class A extends B {}
def f(x: B, y: B) = ...
def f(x: A, y: B) = ...
val a: A
val b: B
```

则应用 f(b, b)指向 f 的第一个定义，应用 f(a, a)指向第二个。假设我们添加第三个重载定义

```
def f(x: B, y: A) = ...
```

则应用 f(a, a)将因模糊定义而被拒绝，因为不存在更具体的可应用签名。

6.25.4. 本地类型推断

本地类型推断推测将要传递给多态类型的表达式的类型参数。比如 e 有类型[a_1 >: L_1 <: U_1,...,a_n >: L_n <: U_n]T 且没有显式类型参数给出。

本地类型推断将此表达式转换为一个类型应用 e[T_1,...,T_n]。类型参量 T_1,...,T_n 的选择依赖于表达式出现处的上下文和期望类型 pt。这里有三种情况。

第一种情况：选择 如果表达式作为名为 x 的命名的前缀出现，则类型推断被延后至整个表达式 e.x。也就是如果 e.x 有类型 S，则现在处理的形式是有类型[a_1 >: L_1 <: U_1,...,a_n >: L_n <: U_n]S，且本地类型推断应用到在 e.x 出现处的上下文中推断类型参量 a_1,...,a_n。

第二种情况：值 如果表达式作为值出现，且没有被应用值参量，类型参量推断的方式是求解一个与表达式类型 t 和期望类型 pt 有关的限定系统。不失一般性，我们可以假定 T 是一个值类型；如果它是一个方法类型，我们可应用 eta 扩展(§6.25.5)将其变为函数类型。求解的意思是找到一个类型参数 a_i 的一个类型为 T_i 的代换 σ，且有：

- 遵守所有的类型参数边界，例如 σL_i <: σa_i 且 σa_i <: σU_i(i=1,...,n)
- 表达式类型与期望类型相似，例如 σT <: σpt。

如果没有这样的代换存在，则将导致编译时错误。如果存在数个这样的代换，则本地类型推断将会针对每个类型变量 a_i 的解空间选择一个最小或最大类型 T_i。如果类型参数 a_i 在表达式的类型 T 中以逆变的形式出现，则选择最大类型 T_i。在其他情况中选择最小类型 T_i，比如变量以协变，非变的方式在 T 中出现，或没有变量。这样的代换叫做类型为 T 的给定限定系统的最优解。

第三种情况：方法 如果表达式 e 在应用 e(d_1,...,d_n)中出现则应用该情况。此处 T 是一个方法类型(R_1,...,R_m)T′。不失一般性我们可以假定结果类型 T′是一个值类型；如果这是一个方法类型，我们可以应用 eta 扩展(§6.25.5)将其变为函数类型。首先使用两个代换方案计算参量表达式 d_j 的类型 S_j。每个参量表达式 d_j 首先用期望类型 R_j 类型化，

在这里类型参数 a_1,...,a_n 作为类型常量。如果失败的话，则将 R_j 中的每个类型参数 a_1,...,a_n 替换为未定义，得期望类型 R'_j，用此类型来类型化参量 d_j。

第二步，通过解一个与期望类型为 pt 的方法类型和参量类型 S_1,...,S_m 有关的限定系统来推断类型参量。求解该限定系统的意思是找到类型参数 a_i 的类型 T_i 的代换 σ，有：

- 遵守所有的类型参数边界，例如 σL_i <: σa_i 且 σa_i <: σU_i(i=1,...,n)
- 方法的结果类型 T′与期望类型一致，例如 σT′ <: σpt
- 每个参量类型与对应的正式参数类型一致，例如 σS_j<:σR_j(i=1,...,n)

如果不存在该代换则将导致编译时错误。如果存在数个解，则选择类型 T′的一个最优解。

期望类型 pt 的全部或部分可以是未定义的。在此一致性规则(§3.5.2)有所扩展，对于任意类型 T 以下两个语句总是正确的

```
undefined <: T 和 T <: undefined
```

对于给定类型变量，可能不存在最小解或最大解，这将导致编译时错误。由于<:是前

序的，因此一个类型的解集中可以有多个最优解。在此情况下 Scala 编译器将自由选取其中某一个。

示例 6.25.2 考虑以下两个方法

```
def cons[A](x: A, xs: List[A]): List[A]= x :: xs
def nil[B]: List[B] = Nil
```

以及定义：

```
val xs = cons(1, nil)
```

cons 的应用首先由一个未定义的期望类型进行类型化。该应用通过本地类型推断为 cons[Int](1, nil)来完成。这里使用了以下理由来推断类型参数 a 的类型参量 Int：

首先，参量表达式被类型化。第一个参量 1 的类型是 Int，第二个参量 nil 是自身多态的。首先尝试用期望类型 List[a]对 nil 做类型检查。这将得到限定系统

```
List[b?] <: List[a]
```

b?中的问号指这是限定系统中的一个变量。因为类 List 是协变的，该限定的最优解是：

```
b = scala.Nothing
```

第二步，在以下限定系统中求解 cons 的类型参数 a

```
Int <: a?
List[scala.Nothing] <: List[a?]
List[a?] <: undefined
```

该限定系统的最优解是

```
a = Int
```

所以 Int 就是 a 的类型推断的结果。

示例 6.25.3 考虑以下定义

```
val ys = cons(“abc”, xs)
```

xs 在前面定义了类型 List[Int]。在此情况下本地类型推断过程如下：

首先将参量表达式类型化。第一个参量“abc”的类型为 String。第二个参量 xs 首先被尝试用期望类型 List[a]类型化。这会失败，因为 List[Int]不是 List[a]的子类型。因此尝试第二种策略；将使用期望类型 List[undefined]来类型化 xs。这会成功得到参量类型 List[Int]。

第二步，在以下限定系统中求解 cons 的类型参数 a：

```
String <: a?
List[Int] <: List[a?]
List[a?] <: undefined
```

该限定系统的最优解是

```
a = scala.Any
```

所以 scala.Any 就是 a 的类型推断的结果。

6.25.5. Eta 扩展

Eta 扩展将一个方法类型的表达式变为一个等价的函数类型的表达式。由两步组成。

首先，标识出 e 的最大子表达式；比如 $e_1,...,e_m$。对于其中每项创建一个新命名 x_i。设 e′是将 e 中每个最大子表达式替换为对应的新命名 x_i 得到的表达式。然后，为方法的每个参量类型 T_i 创建一个新命名 y_i(i=1,...,n)。eta 扩展的结果是：

```
{ val x1 = e1;
 ...
 val xm = em;
 (y1:T1,...,yn:Tn) => e′(y1,...,yn)
}
```

如果 e 仅有一个叫名参数(例如有类型(=>T)U,T 和 U 为某些类型)，e 的 eta 扩展将产生一个类型为 ByNameFunction 的值，该类型定义如下：

```
trait ByNameFunction[-A, +B] extends AnyRef {
  def apply(x: => A): B
  override def toString = "<function>"
}
```

eta 扩展不适用于那些在一个参数段中既有叫名参数又有其他参数的方法。也不适用于那些有重复参数 x: T*(§4.6.2)的方法。

7. 隐含参数和视图

7.1. implicit 修饰符

语法：

```
LocalModifier   ::= 'implicit'
ParamClauses    ::= {ParamClause} [nl] '(' 'implicit' Params ')'
```

用 implicit 修饰符标记的模板成员和参数可以传递给隐含参数(§7.2)，且可以在隐式转换中使用，这种情况称为视图(§7.3)。implicit 修饰符不能用于所有的类型成员和顶级对象(§9.2)。

示例 7.1.1 以下代码定义了一个幺半群的抽象类以及两个实现，StringMonoid 和 IntMonoid。这两个实现标记为 **implicit**

```
abstract class Monoid[A] extends SemiGroup[A] {
  def unit: A
  def add(x: A, y: A): A
}
object Monoids {
  implicit object stringMonoid extends Monoid[String] {
    def add(x: String, y: String): String = x.concat(y)
    def unit: String = "";
  }
  implicit object intMonoid extends Monoid[Int] {
    def add(x: Int, y: Int): Int = x + y
    def unit: Int = 0
  }
}
```

7.2. 隐含参数

隐含参数列表(**implicit** $p_1,...,p_n$)将参数 $p_1,...,p_n$ 标记为隐含的。一个方法或构造器仅能有一个隐含参数列表，且必须是给出的参数列表的最后一个。

具有隐含参数列表的方法可以像正常方法一样应用到参量上。这种情况下 implicit 标识符没有作用。然而如果该方法没有隐含参数列表中的参量，对应的参量会自动提供。

有两种情况实体参量可以传递给类型为 T 隐含参数。首先，所有的标识符 x 可以在方法被调用的地方无需前缀就可以访问到，且该标识符表示一个隐含定义(§7.1)或隐含参数。一个可用的标识符可以是一个本地命名，或封闭模板的一个成员，或者通过 import

子句(§4.7)使其不用前缀即可访问。其次，在隐含参数的类型 T 的隐含作用域中的对象的 implicit 成员也是可用的。

类型 T 的隐含作用域由与隐含参数的类型相关联的类的所有伴随模块(§5.4)构成。类 C 与类型 T 相关联的条件是它是 T 的某部件的基类(§5.1.2)。类型 T 的部件指：

- 如果 T 是一个复合类型 T_1 **with** ... **with** T_n，则是 $T_1,...,T_n$ 的部件以及 T 自身的合集。
- 如果 T 是一个参数化的类型 S[T1,...,Tn]，则是 S 的部件以及 $T_1,...,T_n$ 的合集。
- 如果 T 是一个单例类型 p.**type**，则是 p 的类型的部件
- 如果 T 是一个类型投影 S#U，则是 S 的部件和 T 自身
- 其他情况下则只是 T 自身

如果有多个可用的参量与隐含参数的类型匹配，则将使用静态重载解析(§6.25.3)来选择一个最具体的。

示例 7.2.1 还是示例 7.1.1 中的那些类，现在有一个方法用幺半群的 add 和 unit 操作来计算一个元素列表的和。

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

这里的幺半群被标记为一个隐含参数，且可通过列表的类型来推断。考虑调用

```
sum(List(1,2,3))
```

在上下文中 stringMonoid 和 intMonoid 都是可见的。我们知道 sum 的正式类型参数 a 需要初始化为 Int。与隐含正式参数类型 Monoid[Int]匹配的唯一可用对象是 intMonoid，所以该对象会作为隐含参数被传递。

这里同样说明了隐含参数在所有类型参量被推断(§6.25.4)之后才被推断。

隐含方法自身同样可以有隐含参数。以下是模块 scala.List 中的一个例子，这里将列表注入到 scala.Ordered 类中，列表的元素类型也可以转化为这里的类型。

```
implicit def list2ordered[A](x: List[A])
  (implicit elem2ordered: A => Ordered[A]): Ordered[List[A]] =
  ...
```

假设再增加一个方法

```
implicit def int2ordered(x: Int): Ordered[Int]
```

该方法将整数注入到 Ordered 类中。现在我们可以在 ordered 列表上定义一个 sort 方法：

```
def sort[A](xs: List[A])(implicit a2ordered: A => Ordered[A]) = ...
```

以下我们将方法 sort 应用到整数列表的列表 yss: List[List[Int]]上：

```
sort(yss)
```

以上的调用将通过传递两个嵌套的隐含参量完成：

```
sort(yss)(xs: List[Int] => list2ordered[Int](xs)(int2ordered))
```

将隐含参量传递给隐含参量将有可能导致死循环。比如，试图定义以下方法，将任何类型都注入到 Ordered 类中：

```
implicit def magix[A](x: A)(implicit a2ordered: A =>
Ordered[A]): Ordered[A]): Ordered[A] = a2ordered(x)
```

现在，如果尝试将 sort 应用于没有另外注入到 Ordered 类中的参量 arg，则将得到无限扩展：

```
sort(arg)(x => magic(x)(x => magic(x)(x => ... )))
```

为了避免这类无限扩展发生，编译器将为当前搜索的隐含参量创建一个“开放隐含类型”堆栈。当一个类型 T 的隐含参量被搜索时，T 的“核心类型”将会被添加到堆栈中。这里 T 的核心类型是别名已展开，移除了顶级类型标注(§11)和修饰(§3.2.7)，且将顶级存在边界变量用他们的上界替换后的 T。在对隐含参数的搜索完成后，不管搜索成功与否，核心类型都会被从堆栈中删除。每当有一个核心类型添加到堆栈中，将会检查该类型没有影响到集合中的任何其他类型。

这里一个核心类型 T 影响到一个类型 U 的情况是 T 等价于 U，或 T 和 U 的顶级类型构造器有共有元素且 T 比 U 更复杂。

类型 T 的顶级类型构造器集合 ttcs(T)与类型的形式有关：

对于类型指示器，

ttcs(p.c) = {c}

对于参数化类型，

ttcs(p.c[targs]) = {c}

对于单例类型，

ttcs(p.**type**) = ttcss(T)，p 的类型是 T

对于复合类型，

ttcs(T_1 **with** ... **with** T_n) = ttcs(T_1) U ... U ttcs(T_n)

核心类型的复杂度 complexity(T)是一个依赖于类型的形式的整数：

对于类型指示器，

complexity(p.c) = 1 + complexity(p)

对于参数化类型，

complexity(p.c[targs]) = 1 + Σcomplexity(targs)

对于表示包 p 的单例类型，

complexity(p.**type**) = 0

对于其他单例类型，

complexity(p.**type**) = 1 + complexity(T)，p 的类型是 T

对于复合类型，

$$complexity(T_1 \textbf{ with } ... \textbf{ with } T_n) = \Sigma complexity(T_i)$$

示例 7.2.2 对于某些类型为 List[List[List[Int]]]的列表 xs，sort(xs)类型化的隐含参量类型搜索序列是

```
List[List[Int]] => Ordered[List[List[Int]]],
List[Int] => Ordered[List[Int]]
Int => Ordered[Int]
```

所有的类型都共享类型构造器 scala.Function1，但是每个新类型的复杂度要比之前的类型低。这就是代码的类型检查方式。

示例 7.2.3 设 ys 是某些不能转变为 Ordered 的类型的 List，例如：

```
val ys = List(new IllegalArgumentException, new ClassCastException, new
Error)
```

假设上下文中有以上定义的 magic。则隐含参量类型搜索的序列是

```
Throwable => Ordered[Throwable],
Throwable => Ordered[Throwable],
...
```

由于序列中的第二个类型等价于第一个，编译器将产生一个发散隐含扩展错误。

7.3. 视图

隐含参数和方法也可以定义隐式转换，称作视图。由类型 S 到类型 T 的视图由一个函数类型为 S=>T 或(=>S)=>T 的隐含值或一个可以转变为该类型的值定义。

视图在两种情况下应用。

1. 如果表达式 e 类型为 T，且 T 与表达式的期望类型不一致。这种情况下将会搜索一个隐含的 v，v 可以应用到 e 且结果类型与 pt 一致。搜索的形式类似于隐含参数，隐含作用域类似于 T => pt。如果找到了这样一个视图，则表达式 e 变为 v(e)。
2. 选择 e.m 中，e 的类型为 T，如果选择器 m 并不表示 T 的成员。这种情况下会搜索视图 v，v 可以应用到 e 且结果包含一个名为 m 的成员。搜索的形式类似于隐含参数，隐含作用域是 T。如果找到了这样一个视图，则选择 e.m 变为 v(e).m。

对于隐含参数，如果有多个可选者，则应用重载解析。

示例 7.3.1 类 scala.Ordered[A]有一个方法

```
def <= [B >: A](that: B)(implicit b2ordered: B => Ordered[B]): Boolean
```

设有类型为 List[Int]的两个列表 xs 和 ys，且§7.2 中定义的方法 list2ordered 和 int2ordered 在作用域中，那么操作

```
xs <= ys
```

是合法的，且扩展为

```
list2ordered(xs)(int2ordered).<=
  (ys)
```

```
  (xs => list2ordered(xs)(int2ordered))
```

list2ordered 的第一个应用将列表 xs 转变为类 Ordered 的一个实例，第二个则是传递给<=方法的隐含参数的一部分。

7.4. 视图边界

语法：

```
TypeParam   ::= (id | '_') [TypeParamClause]['>:' Type]
                ['<:' Type]
```

一个方法或非特征类的类型参数 A 可以有一个视图边界 A <% T。这里的类型参数可以实例化为任何类型 S 且 S 可通过应用一个视图变为边界 T。

处理包含这样一个类型参数的方法或类等价于一个具有视图参数的方法，如：

```
def f[A <% T](ps): R = ...
```

扩展为

```
def f[A](ps)(implicit v: A => T): R = ...
```

v 是隐含参数的新名称。特征没有构造器参数，不能应用该转换。因此特征的类型参数没有视图边界。

示例 7.4.1 示例 7.3.1 中提到的<=方法具体声明如下

```
def <= [B >: A <% Ordered[B]](that: B): Boolean
```

8. 模式匹配

8.1. 模式

语法：

```
Pattern         ::= Pattern1 { '|' Pattern1 }
Pattern1        ::= varid ':' TypePat
                | '_' ':' TypePat
                | Pattern2
Pattern2        ::= varid ['@' Pattern3]
                | Pattern3
Pattern3        ::= SimplePattern
                | SimplePattern {id [nl] SimplePattern}
SimplePattern   ::= '_'
                | varid
                | Literal
                | StableId
                | StableId '(' [Patterns [',']] ')'
                | StableId '('[Patterns ','][varid '@'] '_' '*' ')'
                | '(' [Patterns [',']] ')'
                | XmlPattern
Patterns        ::= Pattern {',' Patterns}
```

模式由常量，构造器，变量和类型测试组成。模式匹配测试一个或一组值是否符合给定模式，如果符合则将模式中的变量绑定到该值或该组值的相关部分。一个模式中的同一个变量名不能被多次绑定。

示例 8.1.1 一些模式的例子

1. 模式 ex:IOException 匹配所有类 IOException 的实例，将变量 ex 绑定到该实例。
2. 模式 Some(x) 匹配形如 Some(v)的值，绑定 x 到 Some 构造器的参量 v
3. 模式(x,_) 匹配值对，绑定 x 到第一个值，值对中的第二个部分被模式通配符_所匹配
4. 模式 x::y::xs 匹配长度>=2 的列表，绑定 x 到列表的第一个值，y 到第二个值，xs 绑定到其余部分
5. 模式 1|2|3 匹配 1 到 3 之间的整数

模式匹配总是在可以给出模式的期望类型的上下文中进行，且具有如下不同的类型：

8.1.1. 变量模式

语法：

```
SimplePattern   ::= '_'
                  | varid
```

变量模式 x 是一个简单的标识符，第一个字母必须小写，可以匹配任何值，并绑定变量名到该值。x 的类型为由外部给出的模式的期望类型，由该模式所处的外部表达式决定。变量模式的特殊情况是通配符_，每次出现都被作为一个新的变量使用。

8.1.2. 类型化模式

语法：

```
Pattern1    ::= varid ':' TypePat
              | '_' ':' TypePat
```

类型化模式 x:T 由模式变量 x 和类型模式 T 组成，匹配符合类型模式 T 的值(§8.2)，并绑定变量名到该值。

8.1.3. 字面值模式

语法：

```
SimplePattern   ::= Literal
```

字面值模式 L 匹配任何与字面值 L 相等(==)的值，L 的类型必须与模式的期望类型一致。

8.1.4. 稳定标识符模式

语法：

```
SimplePattern   ::= StableId
```

稳定标识符模式为一个稳定标识符 r(§3.1)，r 的类型要与模式的期望类型一致，该模式匹配所有满足 r==v 的值 v(§12.1)。

为避免与变量模式的语法重叠，稳定标识符模式可以不是小写字母开始的简单名字，但同一变量名若被反引号引用，将被作为稳定标识符模式处理。

示例 8.1.2 考虑以下函数定义：

```
def f(x: Int, y: Int) = x match {
  case y => ...
}
```

在这里，y 是一个变量模式，匹配任意的值(与 f 的参数 y 无关)。如下形式则可以得到稳定标识符模式：

```
def f(x: Int, y: Int) = x match {
```

```
  case `y` => ...
}
```

现在模式匹配到函数 f 的参数 y，只有当 f 的两个参数 x 和 y 相等，匹配才会成功。

8.1.5. 构造器模式

语法：

```
SimplePattern   ::= StableId '(' [Patterns [','']] ')'
```

构造器模式具有如 c(p_1,...,p_n)(n>=0)的形式，由稳定标识符 c 后跟模式 p_1,...,p_n 组成。c 或者是简单名字，或者是一个限定的名字标识一个 case 类(§5.3.2)。如果 c 标识的 case 类是单态的，它必须与模式的期望类型一致，x 的主构造器的正式参数类型类型(§5.3)就作为 p_1,...,p_n 的期望类型。如果该 case 类是多态的，其类型参数将被实例化，使得 c 的实例与模式的期望类型一致。构造器模式匹配所有由构造器调用 c(v_1,...,v_n)产生的对象，p_1,...,p_n 匹配到 v_1,...,v_n。

有一种构造器模式的特殊情况是 c 的正式参数类型结尾是一个重复参数，这将在后面(§8.1.8)讨论。

8.1.6. 元组模式

语法：

```
SimplePattern   ::= '(' [Patterns [','']] ')'
```

一个元组模式(p_1,...,p_n)其实是构造器模式 scala.Tuplen(p_1,...,p_n)的别名(n>=2)，也可以在末尾多加一个逗号:(p_1,...,p_n,)。空元组()是类型为 scala.Unit 的唯一值。

8.1.7. 提取模式

语法：

```
SimplePattern   ::= StableId '(' [Patterns [','']] ')'
```

提取模式具有与构造器模式相同的语法，但提取模式中的稳定标识符不是 case 类，而是这样一个对象，其拥有与模式相匹配的名为 unapply 或者 unapplySeq 的方法。

一个对象 x 的 unapply 方法只接受单一参数并且符合以下任一条件时，我们说它匹配模式 x(p_1,...,p_n)：

1. n = 0 且 unapply 返回布尔型值。匹配所有的值 v，如果 x.unapply(v)的值为 **true**。
2. n = 1 且 unapply 返回 Option[T]类型的值。这种情况下，p1 以期望类型 T 类型化。模式匹配所有的值 v，如果 x.unapply(v)的值为 Some(v1)且 p1 与 v1 匹配。
3. n > 1 且 unapply 返回 Option[(T_1,...,T_n)]。这种情况下，p_1,...,p_n 以期望类型 T_1,...,T_n 依次类型化。模式匹配所有的值 v，如果 x.unapply(v)的值为 Some((v_1,...,v_n))。

如果对象 x 的 unapplySeq 方法只接受单一参数并且返回形如 Option[S]的值，这里 S 是元素类型为 T 的 Seq[T]的子类型，我们说它匹配模式 $x(p_1,...,p_n)$，将在(§8.1.8)中进一步讨论。

8.1.8. 模式序列

语法：

```
SimplePattern  ::= StableId '(' [Patterns ','] [varid '@'] '_' '*' ')'
```

模式序列 $p_1,...,p_n$ 在两种场合下出现。首先，出现在构造器模式 $c(q_1,...,q_m,p_1,...p_n)$中，其中 c 是一个 case 类，拥有 m+1 个主要构造参数，最后一个参数是类型为 S*的重复参数(§4.6.2)。其次，出现在提取模式 $x(p_1,...,p_n)$，如果对象 x 拥有一个返回 Seq[S]的 unapplySeq 方法，而没有匹配 $p_1,...,p_n$ 的 unapply 方法。所有这两种情况，模式的期望类型都是 S。

模式序列的最后一个模式可以是序列通配符_*。每一个模式 p_i 要么是期望类型 S，要么是通配符。当最后一个模式是通配符时，整个模式序列匹配到长度>=n-1 的值序列，否则只能匹配与模式 $p_1,...,p_n$ 匹配的元素构成的长度为 n 的值序列。

8.1.9. 中缀操作符模式

语法：

```
Pattern3    ::= SimplePattern {id [nl] SimplePattern}
```

中缀操作符模式 p op q 是构造器模式或提取模式 op(p,q) 在语法上的简化，操作符在模式中的优先级和结合性与在表达式中相同(§6.12)。

同样，中缀操作符模式 p op (q1,...,qn) 是构造器模式或提取模式 op(p,q1,...qn)的语法简化。

8.1.10. 模式选择

语法：

```
Pattern ::= Pattern1 { '|' Pattern1 }
```

模式选择 p1|...|pn 由一组备选模式 p_i 组成，所有的备选模式在类型上都要符合模式的期望类型，它们不能绑定到除通配符外的任何变量。当至少一个备选模式匹配值 v，则整个模式也匹配到 v。

8.1.11. XML 模式

将在 §10.2 中讨论。

8.1.12. 正则表达式模式

正则表达式模式在 Scala2.0 以后不再被支持，代之以一个大为简化的版本，该简化

版本涵盖非文本序列处理的大多数情况。当一个模式符合下列条件之一，我们称其为序列模式。

1. 模式的类型与类型 Seq[A]一致，期望类型 A
2. 模式为一个 case 类的构造器，拥有重复正式参数 A*(通配符*出现在最右边表示任意长度的序列)。可以用@绑定到 Seq[A]类型的变量。

8.1.13. 恒等模式

当一个模式 p 满足下列条件之一，我们称其为类型 T 的恒等模式

1. p 是变量模式
2. p 是类型化模式 x:T'，并且 T<:T'
3. p 是构造器模式 c(p_1,...,p_n)，T 是类 c 的一个实例，其主要构造器(§5.3)参量的类型为 T_1,...,T_n，而且每个 p_i 是 T_i 的恒等模式。

8.2. 类型模式

语法：

```
TypePat ::= Type
```

类型模式由类型、类型变量和通配符构成，具有以下可能的形式：

- 到类 C，类 p.C，或 T#C 的引用。匹配类 C 的所有非空实例。注意类的前缀(如果给出的话)是与决定类的实例无关的。例如模式 p.C(创建时有前缀路径 p)只匹配类 C 的实例。
  底层类型 scala.Nothing 和 scala.Null 不能用作类型模式，因为它们不能匹配任何东西。
- 单例类型 p.**type**，匹配路径 p 代表的值(调用类 AnyRef 的 eq 方法判断与 p 的同一性)
- 复合类型模式 T_1 **with** ... **with** T_n，其中每个 T_i 都是一个类型模式。匹配与每一个 T_i 都匹配的值。
- 参数化类型模式 T[a_1,...,a_n]，其中每个 a_i 或者是类型变量模式，或者是通配符_。匹配 a_1,...,a_n 任意实例所产生的类型 T。类型变量的绑定或别名在(§8.3)讨论。
- 参数化类型模式 scala.Array[T_1]，其中 T_1 也是一个类型模式。该模式匹配 scala.Array[U_1]的所有非空实例，条件是 T_1 匹配类型 U_1。

除了以上形式的类型，也存在其他的类型模式，但其类型会变为它们的类型擦除(§3.7)，导致 Scala 编译器产生警告信息"unchecked"，提示失去类型安全。

类型变量模式是一个小写字符开头的简单标识符，不能采用 scala 预定义的基本类型名称如 unit, boolean, byte, short, char, int, long, float 和 double。

8.3. 模式中的类型参数推断

类型参数推断处理指定类型模式或构造器模式中的被绑定的类型变量的边界，从而确定模式的期望类型。

类型化模式的类型参数推断：假定模式为 p:T'，将 T'中所有通配符用新的类型变量代替，

我们得到类型 T，其类型变量为 $a_1,...,a_n$，这些类型变量在模式中被绑定。并令模式的期望类型为 pt。

推断过程：首先为类型变量 $a_1,...,a_n$ 构造一个子类型约束集合 C_0。初始的约束集合反映了类型变量的边界，假定 $a_1,...,a_n$ 相关于类类型参数 $a_1',...,a_n'$，且具有下界 $L_1,...,L_n$ 和上界 $U_1,...,U_n$。集合 C_0 的约束条件如下：

$a_i <: \sigma U_i \ (i=1,...,n)$

$\sigma L_i <: a_i \ (i=1,...,n)$

σ 是替换 $[a_1':=a_1,...,a_n':=a_n]$

针对集合 C_0 进一步进行子类型约束，这一过程可能有两种情况：

1. 如果存在针对类型变量 $a_1,...,a_n$ 的变换 σ，使得 σT 与 pt 一致。可以确定一个针对 $a_1,...,a_n$ 的最弱子类型约束 C_1，使得 $C_0 \wedge C_1$ 意味着 T 符合 pt。
2. 如果不能用实例化类型变量的方法使 T 和 pt 一致，可以把 pt 的所有类型变量定义为模式中的某个方法的类型参数，令这些类型参数为 $b_1,...,b_m$，且 C_0' 是反映类型参数 b_i 的边界的子类型约束集。如果 T 是某 final 类的实例类型，那么有针对 $a_1,...,a_n$ 和 $b_1,...,b_m$ 的最弱子类型约束集 C_2，使得 $C_0 \wedge C_0' \wedge C_2$ 使 T 与 pt 一致；如果 T 不是 final 类的实例类型，同样有 $a_1,...,a_n$ 和 $b_1,...,b_m$ 的最弱子类型约束集 C_2，使得 $C_0 \wedge C_0' \wedge C_2$ 使有可能构造出类型 T'与 T 和 pt 都一致。如果没有符合该条件的约束集 C_2，就会产生静态错误。

最后一步，选择类型参数的边界以和前面建立的约束系统匹配，其处理过程因上一步的两种不同情况而异：

1. 对每个类型变量 a_i，我们让 $a_i>:L_i<:U_i$ (L_i 和 U_i 分别为类型下界和上界)即可满足 $C_0 \wedge C_1$。
2. 对类型变量 a_i 和 b_i，我们让 $a_i>:L_i<:U_i$ 和 $b_i>:L_i'<:U_i'$ 同时成立，即可满足 $C_0 \wedge C_0' \wedge C_2$。

在两种情况下，都引入了本地类型推断来降低整体边界推断的复杂性，类型的最小值和最大值使得类型集合的复杂度可以接受。

构造器模式的类型参数推断：假设构造器模式 $C(p_1,...,p_n)$ 中，类 C 具有类型参数 $a_1,...,a_n$，这些类型参数可以用与类型化模式 $(\_: C[a_1,...,a_n])$ 相同的方法推断出来。

示例 8.3.1 考虑以下程序片段：

```
val x: Any
x match {
 case y: List[a]=>...
}
```

这里，类型模式 List[a]与期望类型 Any 进行匹配，模式绑定了类型变量 a，List[a]的任何类型参数都使得 List[a]与 Any 一致，因此 a 是没有边界的抽象类型。a 的作用域就是 case 子句右边的代码。

另一方面，如果 x 的声明是这样的：

```
val x: List[List[String]]
```

这会产生约束 List[a] <: List[List[String]]。因为 List 类型具有协变性，该约束可简化为 a <: List[String]，这样可知 a 具有类型上界 List[String]。

示例 8.3.2 考虑以下程序片段：

```
val x: Any
x match {
  case y: List[String] => ...
}
```

在运行时，Scala 并不保存类型参量的信息，所以没有办法检查 x 是否是字符串列表。编译器将对模式 List[String]进行类型擦除(§3.7)，成为 List[_]。即只检查 x 的顶级运行时类是否与 List 一致，如果是，则模式将被匹配。这可能导致类型转换的异常，如果列表 x 的元素并非字符串的话。Scala 编译器会针对这种情况给出“unchecked”警告信息，提醒用户潜在的类型安全缺陷。

示例 8.3.3 考虑以下程序片段：

```
class Term[A]
class Number(val n: Int) extends Term[Int]
def f[B](t: Term[B]): B = t match {
  case y: Number => y.n
}
```

模式 y: Number 的期望类型是 Term[B]，但类型 Number 并不与 Term[B]一致，因此必须使用前面讨论过的规则 2，引入类型变量 b 来推断子类型约束。在这个例子里我们有约束 Number <: Term[B]，限定了 B=Int。因此 B 在 case 语句被当作一个抽象类型，其上下限均为 Int，由此，case 语句的右边部分 y.n 的类型为 Int，也就与函数声明的类型 Number 一致了。

8.4. 模式匹配表达式

语法：

```
Expr        ::= PostfixExpr 'match' '{' CaseClauses '}'
CaseClauses ::= CaseClause {CaseClause}
CaseClause  ::= 'case' Pattern [Guard] '=>' Block
```

模式匹配表达式形为

e **match** { **case** p_1 => b_1 . . . **case** p_n => b_n }

模式匹配表达式由一个选择器表达式 e 和一组 n 个 case 表达式组成(n>0)。每个 case 由一个(可能有守卫的)模式 p_i 和代码块 b_i 组成。每个模式 pi 可以由守卫语句(**if** e>)进一步限定（这里 e 为布尔型表达式)。p_i 中的模式变量的作用域包括守卫语句和对应的代码块 b_i。

令选择器表达式 e 的类型为 T，并且 $a_1,...a_m$ 是拥有模式匹配表达式内的方法的类型参数，每个 a_i 具有类型下界 L_i 和类型上界 U_i。每个模式 p(p ∈ {$p_1,...,p_n$})的类型有两种方法得到。首先，试图令 T 为 p 的期望类型，如果失败，将 T 的每个类型参数 a_i 用 undefined 取代，得到另一个期望类型 T'。如果这一步也失败的话，将会抛出一个编译错误。否则令 T_p 为表达式 p 的类型，决定一组类型下界 $L_1',...,L_m'$ 和类型上界 $U_1',...,U_m'$ 使得对所有 i，关系 $L_i<:L_i'$ 和 $U_i'<:U_i$ 成立，且满足下面的约束关系：

$L_1 <: a_1 <: U_1 \wedge . . . \wedge L_m <: a_m <: U_m => T_p <: T$

如果没有这样的类型边界，将抛出一个编译错误。否则 a_i 的上下界将为 L_i'和 U_i'，由此得到 p 开头的模式匹配子句的类型。

每一个代码块 b_i 的期望类型就是整个模式匹配表达式的期望类型，而模式匹配表达式的实际类型是所有代码块 b_i 的类型的最小公共上界。

当把模式匹配表达式应用于选择器时，选择器按顺序匹配所有模式，直到匹配成功。假设匹配到 case p_i=>b_i，整个表达式的结果就是代码块 b_i 的求值结果。而 p_i 所包含的所有模式变量将会绑定到选择器的对应部分。如果匹配失败，将会抛出一个 scala.MatchError 异常。

case 表达式中的模式可以有守卫后缀 **if** e，e 为布尔型表达式。匹配到该模式之后，对守卫表达式求值，值为真则匹配成功，否则继续匹配后继模式。

为了提高运行效率，编译器有可能打乱 case 表达式所给定的模式匹配次序。这时如果某些模式的守卫表达式包含副作用的话，就可能会对结果产生影响，但编译器能够确保只有模式被匹配到的时候，其守卫表达式才被求值。

如果选择器是 sealed 类(§5.2)的实例，编译器会给出警告，提醒给定待匹配的模式枚举不完全，运行时可能导致 scala.MatchError 异常。

示例 8.4.1 考虑以下算术运算符的定义：

```
abstract class Term[T]
case class Lit(x: Int) extends Term[Int]
case class Succ(t: Term[Int]) extends Term[Int]
case class IsZero(t: Term[Int]) extends Term[Boolean]
case class If[T](c: Term[Boolean],
                 t1: Term[T],
                     t2: Term[T]) extends Term[T]
```

以上定义包括数字字面值，加 1 运算，0 值测试和条件运算。每个运算符都有一个类型参数，表明运算的类型(整型或布尔型)。

下面是关于上述运算的类型安全的求值函数：

```
def eval[T](t: Term[T]): T = t match {
  case Lit(n)       => n
  case Succ(u)      => eval(u) + 1
  case IsZero(u)    => eval(u) == 0
  case If(c, u1, u2) => eval(if (eval(c)) u1 else u2)
}
```

类型参数可以通过模式匹配获得新的类型边界这一事实，是求值函数得以成立的关键。

例如：第二个 case 中，模式 Succ(u)类型参数 T 的类型为 Int，只有当 T 的类型上下界都是 Int 时，才符合选择器的期望类型。有了 Int <: T <: Int 这个假定，我们可以验证第二个的右侧的类型 Int 与期望类型 T 一致。

8.5. 模式匹配匿名函数

语法：

```
BlockExpr   ::= '{' CaseClauses '}'
```

匿名函数由一系列 case 组成：

```
{ case p1 => b1 . . . case pn => bn }
```

实际上是一个没有前缀 **match** 的表达式。表达式的期望类型必须部分被定义，要么是 scala.Function*k*[S_1,...,S_k, R](k>0)，要么是 scala.PartialFunction[S_1, R]，其中参数类型 S_1,...,S_k 必须明确，而结果类型可以待定。

如果期望类型是 scala.Function*k*[S_1,...,S_k, R]，则整个表达式与下面的匿名函数等价：

```
(x1:S1,...,xk:Sk ) => (x1,...,xk) match {
  case p1 => b1 ... case pn => bn
}
```

x_i 是新引入的变量名。如(§6.23)所示，上述匿名函数与下面的实例构造表达式等价(这里的 T 是所有 b_i 的最小共同类型上界)。

```
new scala.Functionk[S1,...,Sk, T] {
  def apply(x1 : S1,...,xk : Sk ): T = (x1,...,xk) match {
    case p1 => b1 ... case pn => bn
  }
}
```

如果期望类型是 scala.PartialFunction[S, R]，表达式与下列实例构造表达式等价：

```
new scala.PartialFunction[S , T ] {
  def apply(x : S): T = x match {
    case p1 => b1 ... case pn => bn
  }
  def isDefinedAt(x : S): Boolean = {
    case p1 => true ... case pn => true
    case _ => false
  }
}
```

x 是新引入的变量名，T 为所有 b_i 的最小共同类型上界。isDefinedAt 方法中，如果前面的模式 p_1,...p_n 已经有了变量模式或通配符模式，最后的缺省 case 将被忽略。

示例 8.5.1 这个函数使用 fold-left 操作符/:计算两个矢量的标积：

```
def scalarProduct(xs: Array[Double], ys: Array[Double]) =
  (0.0 /: (xs zip ys)) {
    case (a, (b, c)) => a + b * c
  }
```

case 子句与下列匿名函数等价：

```
(x, y) => (x, y) match {
  case (a, (b, c)) => a + b * c
```

```
}
```

9. 顶级定义

9.1. 编译单元

语法：

```
CompilationUnit ::= [‘package’ QualId semi] TopStatSeq
TopStatSeq      ::= TopStat {semi TopStat}
TopStat         ::= {Annotation} {Modifier} TmplDef
                  | Import
                  | Packaging
                  |
QualId          ::= id {‘.’ id}
```

编译单元由包，导入子句和类与对象定义(前面可以是包子句)构成。

有包子句开始的编译单元 **package** p; stats 等价于由单个包 **package** p {stats}构成的编译单元。

按照顺序隐式导入每个编译单元的内容有：包 java.lang，包 scala 以及对象 scala.Predef(§12.5)。该顺序中后引入的成员将隐藏先前引入的成员。

9.2. 打包

语法：

```
Packaging   ::= package QualId [nl] ‘{’ TopStatSeq ‘}’
```

包是一个定义了一些成员类，对象与包的特殊对象。与其他对象不同，包不是由定义引入的。包的成员集合是由打包确定的。

打包 **package** p { ds }将 ds 中所有的定义作为成员放到限定名为 p 的包中。包的成员称为顶级定义。如果 ds 中某定义标记为 **private**，则仅在包内成员中可见。

p 中或从 p 导入的选择 p.m 工作方式类似对象。然而不像其他对象，包不能用作值。包与模块或对象不能有相同的完整的限定名。

打包之外的顶级定义默认放到一个特别的空包中。这个包不能被命名，也不能被导入。然而空包的成员不用限定就对彼此可见。

9.3. 包引用

语法：

```
QualId  ::= id {‘.’ id}
```

到包的引用具有限定标识符的形式。和其他的引用类似，包的引用是相对的。也就是说，由命名 p 开始的包的引用将在定义了名为 p 的成员的最近的封闭域内查找。

特例是预定义的命名_root_，指最外层的根包，并包括所有顶层的包。

示例 9.3.1 考虑以下程序：

```
package b {
  class B
}

package a.b {
  class A {
    val x = new _root_.b.B
  }
}
```

这里引用_root_.b.B 指顶级包 b 中的类 B。如果忽略_root_前缀，命名 b 就解析为包 a.b，由于该包中没有名为 B 的类，所以这样会导致编译错误。

9.4. 程序

程序是顶级对象，具有一个类型为(Array[String])Unit 的方法成员 main。程序可以在命令行界面中执行。程序的命令行参量将以类型 Array[String]的形式传递给 main 方法。

程序的 main 方法可以直接在对象中定义，也可以继承。scala 库中定义了一个类，叫做 scala.Application，定义了一个空的继承的 main 方法。继承自该类的对象 m 就是一个程序，会执行对象 m 的初始化代码。

示例 9.4.1 以下示例将通过在模块 test.HelloWorld 中定义一个 main 方法来创建一个 hello world 程序。

```
package test
object HelloWorld {
  def main(args: Array[String]) { println("hello world") }
}
```

这个程序可以用以下命令来启动

```
scala test.HelloWorld
```

在 Java 环境中，可以用以下命令

```
java test.HelloWorld
```

HelloWorld 也可以继承 Application，无需定义 main 方法：

```
package test
object HelloWorld extends Application {
  println("Hello world")
}
```

10. XML 表达式与模式

作者：Burak Emir

在这一章里描述了 XML 表达式与模式的语法结构。该结构尽量遵循 XML 1.0 规范[W3C]，只是为了嵌入 Scala 代码片段而做了一些修改。

10.1. XML 表达式

XML 表达式是由以下方法产生的表达式，第一个元素的左尖括号'<'必须要在一个开始 XML 词法模式的位置(§1.5)。

语法：

```
XmlExpr ::= XmlContent {Element}
```

要符合 XML 规范的良好形式，比如开始标签和结束标签必须匹配，属性只定义一次，除非与实体解析有关的限制。

以下描述了 Scala 的可扩展标记语言，从设计上尽可能的与 W3C 的可扩展标记语言标准接近。只有在属性值和字符数据上有所改变。Scala 不支持声明，CDATA 段或处理指令。实体引用并不在运行时解析。

语法：

```
Element         ::= EmptyElemTag | Stag Content ETag
EmptyElemTat    ::= '<' Name {S Attribute} [S] '/>'
Stag            ::= '<' Name {S Attribute} [S] '>'
ETag            ::= '</' Name [S] '>'
Content         ::= XmlContent | Reference | ScalaExpr
XmlContent      ::= Element | CDSect | PI | Comment
```

如果 XML 表达式是一个单个元素，则其值是一个 XML 节点(scala.xml.Node 的子类的实例)的运行时表示。如果 XML 表达式包括一个以上的元素，则其值是一个 XML 节点的序列(scala.Seq[scala.xml.Node]的子类的实例)的运行时表示。

如果 XML 表达式是一个实体引用，CDATA 段，处理指令或标注，则由对应的 Scala 运行时类的实例来表示。

元素内容中首尾空格默认被删除，连续空白字符用单个空白字符\u0020 替换。可通过用一个编译选项来保留所有空白字符来改变该行为。

语法：

```
Attribute       ::= Name Eq AttValue
AttValue        ::= '"' {CharQ | CharRef} '"'
                  | '' {CharA | CharRef} ''
```

```
                | ScalaExpr
ScalaExpr       ::= Block
CharData        ::= {CharNoRef} without {CharNoRef} '{'CharB
                      {CharNoRef} and without {CharNoRef}']]>' {CharNoRef}
```

XML 表达式可以包含 Scala 表达式作为属性值或在节点内。在后者它们以左大括号'{'开始右大括号'}'结束的方式嵌入。在 CharData 产生的 XML 文本中表示单个左大括号'{'则需要将其重复一次。即'{{'表示 XML 文本'{'，并不会引入嵌入的 Scala 表达式。

语法：

```
BaseChar, Char, Comment, CombiningChar, Ideorgraphic, NameChar, S,
Reference   ::= "和 W3C XML 一样"
Char1       ::= Char without '<' | '&'
CharQ       ::= Char1 without '"'
CharA       ::= Char1 without '''
CharB       ::= Char1 without '{'
Name        ::= XNameStart {NameChar}
XNameStart  ::= '_' | BaseChar | Ideographic  (和 W3C XML 一样，但是没有 ':')
```

10.2. XML 模式

XML 模式是由以下方式产生的模式，元素模式的左尖括号'<'必须在开始 XML 模式词法的起始位置(§1.5)。

语法：

```
XmlPattern  ::= ElementPattern
```

此处应用 XML 规范的良好格式限制。

一个 XML 模式必须是单个元素模式。它必须准确匹配具有模式描述的同样结构的 XML 树的运行时表现。XML 模式可以包含 Scala 模式(§8.4)。

空格的处理与 XML 表达式中的一样。实体引用，CDATA 段，处理指令和标注的模式的运行时表示也一样。

元素中首尾空格默认被删除，连续空白字符用单个空白字符\u0020 替换。该行为可通过用一个编译选项来保留所有空白字符来改变。

语法：

```
ElemPattern ::= EmptyElemTagP | STagP ContentP ETagP
EmptyElemTagP   ::= '<' Name [S] '/>'
STagP           ::= '<' Name [S] '>'
ETagP           ::= '</' Name [S]'>'
ContentP        ::= [CharData] {(ElemPattern | ScalaPatterns) [CharData]}
ContentP1       ::= ElemPattern
                  | Reference
                  | CDSect
                  | PI
                  | Comment
```

```
                | ScalaPatterns
ScalaPatterns   ::= '{' Patterns '}'
```

11.用户定义的标注

语法：

```
Annotation      ::= '@' AnnotationExpr [nl]
AnnotationExpr  ::= Constr [[nl] '{' [NameValuePair
                      {',' NameValuePair}] '}']
NameValuePair   ::= val id '=' PrefixExpr
```

用户定义的标注将元信息与定义关联起来。一个简单的标注具有@c 或@c(a_1,...,a_n)的形式。这里 c 是类 C 的构造器，且该类必须与类 scala.Annotation 一致。构造器后可跟括号中可选的名/值对列表，例如{n_1=c_1,...,n_k=c_k}。该列表中的所有的值 c_i 必须为常量表达式，定义如下所示。

标注可以应用在定义或声明，类型，或表达式上。定义或声明的标注放在该定义的前面。类型的标注放在类型的后面。表达式的标注在表达式之后，中间有一个冒号。对一个实体可以应用多个标注。标注给出的顺序并没有意义。

示例：

```
@serializable class C { ... }      //类标注
@transient @ volatile var m: Int   //变量标注
String @local                          //类型标注
(e: @unchecked) match { ... }      //表达式标注
```

标注的含义依赖于其实现。在 Java 平台上，以下标注具有标准意义。

@transient

将一个字段标记为不保存；等价于 Java 中的 transient 修饰符

@volatile

将一个字段标记为可在程序控制外改变其值；等价于 Java 中的 volatile 修饰符

@serializable

将一个类标记为可序列化的；等价于在 Java 中继承接口 java.io.Serializable

@SerialVersionUID(<longlit>)

给一个类加上序列化版本标识符(一个长整型常量)。等价于 Java 中的以下字段定义：

```
private final static SerialVersionUID = <longlit>
```

@throws(<classlit>)

Java 编译器会通过分析一个方法或构造器是否会导致已检查异常来检查程序是否包含已检查异常的处理器。对于每个可能的已检查异常，方法或构造器的 throws 子句必须要说明异常的类或异常的某个父类。

@deprecated

将一个定义标记为不推荐的。对定义实体的访问将导致编译器发出一个不推荐警告。如果该代码自身属于一个标记为不推荐的定义，则该警告会被抑制。

@scala.reflect.BeanProperty

当作为一个定义或某变量 X 的前缀时，该标注将使 Java bean 形式的 getter 和 setter 方法 getX,setX 添加到该变量所在的类中。get 或 set 后变量的第一个字母将变为大写。当该标注加在一个不可变值定义 X 上时则只产生 getter。这些方法的构建是代码生成的一部分；因此，这些方法只有在对应的 class 文件生成后才可见。

@unchecked

当应用于一个 match 表达式的选择器时，该属性将抑制所有有关不完全的模式匹配的警告，这些匹配将会被忽略。比如以下方法定义不会产生警告。

```
def f(x: Option[Int]) = (x: @unchecked) match {
  case Some(y) => y
}
```

如果没有@unchecked 标注，Scala 编译器就会指出模式匹配是不完全的，并产生一个警告(由于 Option 是一个 sealed 类)。

@uncheckedStable

当应用于一个值声明或定义时，则允许定义的值出现在一个路径中，即使其类型是易变的。例如，以下定义是合法的：

```
type A { type T }
type B
@uncheckedStable val x: A with B    // 易变类型
val y: x.T                          // OK，因为'x'还是一个路径
```

如果没有该标注，指示器 x 就不能是一个路径，因为其类型 A with B 是易变的。继而引用 x.T 是格式错误的。

当应用于非可变类型的值声明或定义时，该标注没有作用。

其他标注将由平台或应用有关的工具解释。类 scala.Annotation 有两个子特征，用来表示这些标注是如何被持有的。继承自特征 scala.ClassfileAnnotation 的标注类的实例将被保存在生成的类文件中。继承自特征 scala.StaticAnnotation 的标注类的实例将在每个被标注的符号被访问的编译单元中，且对 Scala 类型检查器可见。标注类也可同时继承 scala.ClassfileAnnotation 和 scala.StaticAnnotation。如

果一个标注类没有继承 scala.ClassfileAnnotation 也没有继承 scala.StaticAnnotation，则其实例仅在编译器检查它们时在本地可见。

继承自 scala.ClassfileAnnotation 的类可能有更多的限制，这是为了保证它们可以映射到宿主环境中。特别是在 Java 和.NET 平台上，这些类必须是顶级的；比如他们不能包含在其他类或对象中。且在 Java 和.NET 中，所有的构造器参量都必须是常量表达式。

“常量表达式”的定义是与平台有关的，但是必须包含以下形式的表达式：

- 为值类的字面值，比如整数
- 字符串字面值
- 由 classOf 构建的类
- 平台上的某枚举类型的元素
- 字面值数组，形式为@Array(c_1,...,c_n)，所有的 c_i 都是常量表达式。

12.Scala 标准库

Scala 标准库包括包 scala 和一些类与模块。下面描述了这些类的一部分。

12.1. 根类

图 12 展示了 Scala 类的层次结构。层次结构的根是类 Any。Scala 执行环境中的每个类都直接或间接地继承自该类。类 Any 有两个直接子类：AnyRef 和 AnyVal。

子类 AnyRef 表示在宿主系统中表示为一个对象的所有值。所有用户定义的 Scala 类都直接或间接的继承自该类。更进一步，所有用户定义的 Scala 类也都继承自特征 scala.ScalaObject。由其他语言编写的类也都继承自 scala.AnyRef，但是并没有继承 scala.ScalaObject。

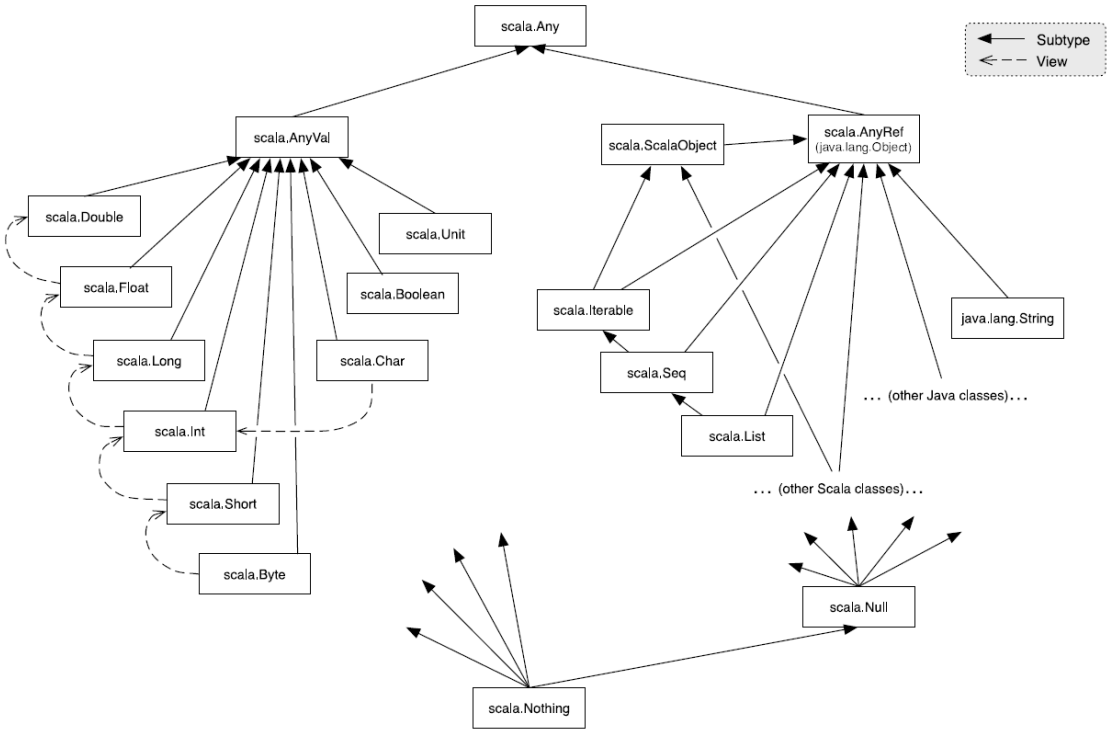

图 12.1 Scala 类层次结构

类 AnyVal 有固定数目的子类，它们描述了在宿主系统中没有作为对象来实现的那些值。

类 AnyRef 和 AnyVal 只需要提供在类 Any 中声明的成员，但是具体实现可以添加宿主有关的方法到这些类中（例如，一个实现可以有其自有的对象的根类）。

这些根类的签名描述如下。

```
package scala
/** 通用根类 */
abstract class Any {

  /** 定义相等，在这里是抽象的 */
  def equals(that: Any): Boolean

  /** 值间的语义相等 */
  final def == (that: Any): Boolean =
    if (null eq this) null eq that else this equals that

  /** 值间的语义不等 */
  final def != (that: Any): Boolean = !(this == that)

  /** Hash code, 这里是抽象的 */
  def hashCode: Int = ...

  /** 文本表示，这里是抽象的 */
  def toString: String =  ...

  /** 类型测试，需要按照下面的方式内联化 */
  def isInstanceOf[a]: Boolean = this match {
    case x: a => true
    case _ => false
  }

  /** 类型转换，需要按照下面的方式内联化 */
  def asInstanceOf[A] = this match {
    case x: A => x
    case _ => if (this eq null) this
                else throw new ClassCastException()
  }
}

/** 所有值类型的根类 */
final class AnyVal extends Any

/** 所有引用类型的根类 */
class AnyRef extends Any {
  def equals(that: Any): Boolean = this eq that
  final def eq(that: AnyRef): Boolean = ... //引用相等
  final def ne(that: AnyRef): Boolean = !(this eq that)
```

```
  def hashCode: Int = ...       //由内存地址计算得来
  def toString: String = ...    //由 hashcode 和类名得来
}

/** 应用于用户定义的 scala 类的混入类 */
trait ScalaObject extends AnyRef
```

对于测试 x.asInstanceOf[T]，如果 T 是一个数字值类型(§12.2)，则做特殊处理。在此情况下的转换是应用一个转换方法 x.toT(§12.2.1)。对于不是数值的值 x 该操作将导致 ClassCastException。

12.2. 值类

值类指在宿主系统中没有表现为对象的那些类。所有的值类都继承自 AnyVal。Scala 实现需要提供值类 Unit, Boolean, Double, Float, Long, Int, Char, Short 和 Byte(也可以提供更多的类)。这些类的特点定义如下。

12.2.1. 数字值类型

类 Double, Float, Long, Int, Char, Short 和 Byte 统称为数字值类型。类 Byte,Short 和 Char 称为子界类型。还有以下子界类型，Int 和 Long 称为整数类型，Float 和 Double 称为浮点类型。

数字值类型的级别为以下偏序：

```
    Byte - Short
                  \
                    Int - Long - Float - Double
                  /
            Char
```

在此排序中，Byte 和 Short 的级别最低，Double 级别最高。级别并不是指一致性(§3.5.2)关系；例如 Int 不是 Long 的子类型。然而，对象 Predef(§12.5)定义了从每个数字值类型到其高级数字值类型的视图(§7.3)。因此，如果在上下文(§6.25)中有需要，低级的类型可以隐式的转换为高级的类型。

给定两个数字值类型 S 和 T，它们的操作类型定义如下：如果 S 和 T 都是子界类型则 S 和 T 的操作类型是 Int。否则 S 和 T 的操作类型就是相对排名中较大的一个。给定两个数字值类型 v 和 w，它们的操作类型就是它们运行时的类型。

任何数字值类型 T 都支持以下方法。

- 比较方法，比如等于(==)，不等(!=)，小于(<)，大于(>)，小于等于(<=)，大于等于(>=)，每个方法都存在 7 个重载的可选项。每个可选项有一个数字值类型的参数。结果类型为 Boolean。操作形式为将接受者和参量变为其操作类型，并在此类型上做比较操作。
- 算数运算加(+)，减(-)，乘(*)，除(/)和求余(%)，每个方法都存在 7 个重载的可选项。每个可选项有一个类型为 U 的数字值类型的参数。结果类型是 T 和 U 的操作类型。操作形式为将接受者和参量变为其操作类型，并在此类型上做算数操作。
- 无参数算数方法正(+)和(-)的结果类型为 T。第一个返回其自身，第二个返回其

负值。

- 转换方法 toByte, toShort, toChar, toInt, toLong, toFloat, toDouble 将对应的对象变为目标类型，规则是 Java 的数值类型转换操作。转换可能截取数字值(比如从 Long 到 Int 或从 Int 到 Byte)或丢失精度(比如从 Double 到 Float 或 Long 和 Float 之间的转换)。

整型数字值也支持以下的操作：

- 位操作方法按位与(&)，按位或(|)和按位异或(^)，每个方法都存在 5 个重载的可选项。每个可选项都有一个整数值类型的参数。结果类型是 T 和 U 的操作类型。操作形式为将接受者和参量变为其操作类型，并在此类型上做对应的位运算操作。
- 无参数方法按位取反(~)。结果类型是接受者的类型 T 或 Int 中的较大者。操作方式是将接受者变为结果类型并将按位取反。
- 移位操作 包括左移(<<)，算数右移(>>)和无符号右移(>>>)。每个方法有两个重载的可选项，并有一个类型为 Int 或 Long 的参数 n。操作的结果类型就是接受者的类型 T 或 Int 中较大者。操作形式为将接受者变为结果类型并移动 n 位。

数字值类型同样也实现了类 Any 中的 equals, hashCode 和 toString 操作。

equals 方法测试参量是不是一个数字值类型。如果为 **true** 的话则执行对应类型的==操作。一个数值类型的等于方法可以认为定义如下：

```
def equals(other: Any): Boolean = other match {
  case that: Byte  => this == that
  case that: Short => this == that
  case that: Char  => this == that
  case that: Int       => this == that
  case that: Long  => this == that
  case that: Float => this == that
  case that: Double    => this == that
  case _ => false
}
```

hashCode 方法返回一个 hashcode 整数，对于相等的结果返回相等的数字值。对于 Int 类型和其他子界类型这一点必须得到保证。

接受者的 toString 方法将显示为一个整数或浮点数。

示例 12.2.1 数字值类型 Int 的签名如下所示：

```
package scala
abstract sealed class Int extends AnyVal {
  def == (that: Double): Boolean   // double 相等
  def == (that: Float): Boolean    // float 相等
  def == (that: Long): Boolean     // long 相等
  def == (that: Int): Boolean      // int 相等
  def == (that: Short): Boolean    // int 相等
  def == (that: Byte): Boolean     // int 相等
  def == (that: Char): Boolean     // int 相等
  /* ! =,  <, >, <=, >=的情况类似 */
```

```
  def + (that: Double): Double     // double 加
  def + (that: Float): Double      // float 加
  def + (that: Long): Long         // long 加
  def + (that: Int): Int           // int 加
  def + (that: Short): Int         // int 加
  def + (that: Byte): Int          // int 加
  def + (that: Char): Int          // int 加
  /* -, *, /, %的情况类似 */

  def & (that: Long): Long         // long 按位与
  def & (that: Int): Int           // int 按位与
  def & (that: Short): Int         // int 按位与
  def & (that: Byte): Int          // int 按位与
  def & (that: Char): Int          // int 按位与
  /* |, ^的情况类似 */

  def << (cnt: Int): Int           // int 左移
  def << (cnt: Long): Int          // long 左移
  /* >>, >>>的情况类似 */

  def unary_+ : Int                    // int 正
  def unary_- : Int                    // int 负
  def unary_~ : Int                    // int 按位取反

  def toByte: Byte                 // 变为 Byte
  def toShort: Short               // 变为 Short
  def toChar: Char                 // 变为 Char
  def toInt: Int                   // 变为 Int
  def toLong: Long                 // 变为 Long
  def toFloat: Float               // 变为 Float
  def toDouble: Double             // 变为 Double
}
```

12.2.2. Boolean 类

类 Boolean 只有两个值：**true** 和 **false**，以下类定义给出了其实现的操作。

```
package scala
abstract sealed class Boolean extends AnyVal {
  def && (p: => Boolean): Boolean =    // 布尔与
    if (this) p else false
  def || (p: => Boolean): Boolean =    // 布尔或
    if (this) true else p
  def & (x: Boolean): Boolean =        // 布尔严格与
    if (this) x else false
```

```
  def | (x: Boolean): Boolean =          // 布尔严格或
    if (this) true else x
  def == (x: Boolean): Boolean =         // 布尔相等
    if (this) x else x.unary_!
  def != (x: Boolean): Boolean           // 布尔不等
    if (this) x.unary_! else x
  def unary_!: Boolean                   // 布尔负
    if(this) false else true
}
```

该类也实现了 Any 类中的 equals，hashCode 和 toString 操作。

equals 方法在参量与接受者是同一布尔值时返回 **true**，否则 **false**。hashCode 方法在 **true** 上调用时返回 1，在 **false** 上调用返回 0。toString 方法将接受者变为一个字符串，比如"**true**"或者"**false**"。

12.2.3. Unit 类

类 Unit 只有一个值：()。且只实现了 Any 类中的三个方法 equals，hashCode 和 toString。

如果参量是值()则 equals 方法返回 **true**，否则 **false**。hashCode 方法返回一个固定的与实现有关的 hash-code 值。toString 方法返回"()"。

12.3. 标准引用类

本节描述了某些在 Scala 编译器中做特殊处理的标准 Scala 引用类 – 要么是 Scala 给它们提供了一些语法糖，要么 Scala 编译器为它们的操作产生了一些特殊代码。Scala 标准库中的其他类的文档在 Scala 库文档的 HTML 页中。

12.3.1. String 类

Scala 的 String 类一般派生自宿主系统的 String 类(可能有所不同)。对于 Scala 客户来说该类肯定支持一个方法

```
def + (that: Any): String
```

该方法将左侧的操作数与右侧操作数的字符形式连接在一起。

12.3.2. Tuple 类

Scala 定义了元组类 Tuple*n* n=2,...,9。定义如下。

```
package scala
case class Tuplen[+a_1, ..., +a_n](_1: a_1, ..., _n: a_n) {
  def toString = "(" ++_1 ++ "," ++ ... ++ "," ++ _n ++ ")"
}
```

隐式引入的 Predef 对象(§12.5)定义了命名 Pair 作为 Tuple2 的别名和 Triple 作为 Tuple3 的别名。

12.3.3. Function 类

Scala 定义了函数类 Function*n*, n=1,...,9。定义如下。

```
package scala
trait Function n[-a_1, ..., -a_n, +b] {
  def apply(x_1: a_1, ..., x_n: a_n): b
  def toString = "<function>"
}
```

Function1 的子类表示一个部分函数，在某些情况下在其领域中是没有定义的。对于 apply 方法，部分函数还有一个 isDefined 方法，来说明函数对于给定的参数是否定义：

```
class PartialFunction[-A, +B] extends Function1[A, B] {
  def isDefinedAt(x: A): Boolean
}
```

隐式引入的 Predef 对象(§12.5)定义了命名 Function 作为 Function1 的别名。

12.3.4. Array 类

通用数组类定义如下。

```
final class Array[A](len: Int) extends Seq[A] {
  def length: Int = len
  def apply(i: Int): A = ...
  def update(i: Int, x: A): Unit = ...
  def elements: Iterator[A] = ...
  def subArray(from: Int, end: Int): Array[A] = ...
  def filter(p: A => Boolean): Array[A] = ...
  def map[B](f: A => B): Array[B] = ...
  def flatMap[B](f: A => Array[B]): Array[B] = ...
}
```

如果 T 不是类型参数或抽象类型，类型 Array[T]表示宿主系统中的原生数组类型[]T。这种情况下 length 返回数组的长度，apply 表示下标，update 表示元素更新。由于 apply 和 update 操作(§6.25)的语法糖的存在，以下是 Scala 和 Java/C#中对数组 xs 操作的对应：

```
Scala              Java/C#
  xs.length      xs.length
  xs(i)          xs[i]
  xs(i) = e      xs[i] = e
```

数组也实现了序列特征 scala.Seq，定义了 elements 方法来返回一个包含数组中所有元素的 Iterator

因为在 Scala 中参数化类型的数组和宿主语言中数组的实现还是有差异的，在处理数组时需要注意一些细小的差别。解释如下。

首先，不像 Java 或 C#中的数组，Scala 中的数组不是协变的；也就是说，在 Scala 中 S<:T 并不能得出 Array[S] <: Array[T]。但是如果在宿主环境中可以将 S 变为 T 则可以将 S 的数组变为 T 的数组。

举例来说 Array[String]并不与 Array[Object]一致，即使 String 与 Object 一致。然而可以将类型为 Array[String]的表达式变为 Array[Object]。该转变将会成功，不会抛出 ClassCastException。例子如下：

```
val xs = new Array[String](2)
// val ys: Array[Object] = xs      // **** 错误：不兼容的类型
val ys: Array[Object] = xs.asInstanceOf[Array[Object]]    //OK
```

其次，对于有一个类型参数或抽象类型 T 作为元素类型的多态数组，其表现形式不同于[]T。然而 isInstanceOf 和 asInstanceOf 仍像数组数组使用单态数组的标准表现形式那样正常工作。

```
val ss = new Array[String](2)

def f[T](xs: Array[T]): Array[String] =
  if(xs.isInstanceOf[Array[String]]) xs.asInstanceOf[Array[String]]
  else throw new Error("not an instance")

f(ss)       // 返回 ss
```

多态数组的表现形式同样保证了多态数组的创建与期望一致。以下是一个 mkArray 方法的例子，给定定义了元素且类型为 T 的序列创建任意类型 T 的数组。

```
def mkArray[T](elems: Seq[T]): Array[T] = {
  val result = new Array[T](elems.length)
  val I = 0
  for (elem <- elems) {
    result(i) = elem
    I += 1
  }
}
```

注意在 Java 数组的类型擦除模型下，以上方法不能按照期望的方式工作-实际上它将总是返回一个 Object 数组。

再次，在 Java 环境中有一个方法 System.arraycopy，有两个对象为参数，指定起始坐标和长度，将元素从一个对象复制到另外一个对象，这两个对象的元素类型必须是兼容的。对于 Scala 的多态数组该方法不能正常工作，因为他们有不同的表现形式。作为代替应当使用 Array 类的伴随对象 Array.copy 方法。该伴随对象也定义了不同的数组构造方法，还定义了提取方法 unapplySeq(§8.1.7)，提供了数组上的模式匹配。

```
package scala
```

```
object Array {
  /** 从'src'复制元素到'dest' */
  def copy(src: AnyRef, srcPos: Int, dest: AnyRef,
               destPos: Int, length: Int): Unit = ...

  /** 将参数中所有数组合并为一个 */
  def concat[T](xs: Array[T]*): Array[T] = ...

  /** 创建一个连续的整数数组 */
  def range(start: Int, end: Int): Array[Int] = ...

  /** 用给定元素创建一个数组 */
  def apply[A <: AnyRef](xs: A*): Array[A] = ...

  /** 与上面类似 */
  def apply(xs: Boolean*)  : Array[Boolean]    = ...
  def apply(xs: Byte*)     : Array[Byte]       = ...
  def apply(xs: Short*)       : Array[Short]  = ...
  def apply(xs: Char*)     : Array[Char]       = ...
  def apply(xs: Int*)      : Array[Int]        = ...
  def apply(xs: Long*)     : Array[Long]       = ...
  def apply(xs: Float*)       : Array[Float]  = ...
  def apply(xs: Double*)   : Array[Double] = ...
  def apply(xs: Unit*)     : Array[Unit]       = ...

  /** 创建一个数组，包含一个元素的多个拷贝 */
  def make[A](n: Int, elem: A): Array[A] = ...

  /** 提供数组上的模式匹配 */
  def unapplySeq[A](x: Array[A]): Option[Seq[A]] = Some(x)
}
```

示例 12.3.1 以下方法复制给定的数组参量并返回初始数组和复制的数组：

```
def duplicate[T](xs: Array[T]) = {
  val ys = new Array[T](xs.length)
  Array.copy(xs, 0, ys, 0, xs.length)
  (xs, ys)
}
```

12.4. Node 类

```
package scala.xml
```

```
trait Node {

/** 该节点的标签 */
def label: String

/** 属性轴 */
def attribute: Map[String, String]

/** 子节点轴(该节点的所有子节点) */
def child: Seq[Node]

/** 下属节点轴(该节点的所有下属节点) */
def descendant: Seq[Node] = child.toList.flatMap{
    x => x::x.descentant.asInstanceOf[List[Node]]
}

/** 下属节点轴(该节点的所有下属节点) */
  def descendant_or_self: Seq[Node] = this::child.toList.flatMap{
    x => x::x.descendant.asInstanceOf[List[Node]]
  }

  override def equals(x: Any): Boolean = x match {
    case that:Node =>
      that.label == this.label &&
        that.attribute.sameElement(this.attribute) &&
          that.child.sameElement(this.child)
    case _ => false
  }

  /** XPath 形式的投影函数。返回该节点所有标签为'that'的子节点。保留它们在
*  文档中的顺序
   */
  def \(that: Symbol): NodeSeq = {
    new NodeSeq({
      that.name match {
        case "_" => child.toList
        case _ =>
          var res:List[Node] = Nil
          for (x <- child.elements if x.label == that.name) {
            res = x::res
          }
          res.reverse
      }
  })
```

```
  }

  /** XPath 形式的投影函数。返回该节点的'descendant_or_self'轴中所有
   *  标签为"that"的节点。保留它们在文档中的顺序。
  def \\(that: Symbol): NodeSeq = {
    new NodeSeq(
      that.name match {
        case "_" => this.descendant_or_self
        case _ => this.descendant_or_self.asInstanceOf[List[Node]]
        filter(x => x.label == that.name)
      })
  }

  /** 该 XML 节点的 hashcode */
  override def hashCode =
    Utility.hashCode(label, attribute.toList.hashCode, child)

  /** 该节点的字符串表现形式 */
  override def toString = Utility.toXML(this)
}
```

12.5. Predef 对象

Predef 对象为 Scala 程序定义了标准函数和类型别名。该对象总是被隐式引入，它所有的成员不用声明即可用。它在 JVM 环境中的定义与以下签名一致：

```
package scala
object Predef {

 // classOf ---------------------------------------------

 /** 返回类类型的运行时表示 */
 def classOf[T] = null
  // 这个不是真实的返回值，该方法将由编译器处理

 // 标准类型别名
 type byte     = scala.Byte
 type short    = scala.Short
 type char     = scala.Char
 type int      = scala.Int
 type long     = scala.Long
 type float    = scala.Float
 type double   = scala.Double
 type boolean  = scala.Boolean
 type unit     = scala.Unit
```

```
  type String   = java.lang.String
  type Class[T] = java.lang.Class[T]
  type Runnable = java.lang.Runnable

  type Throwable    = java.lang.Throwable
  type Exception    = java.lang.Exception
  type Error        = java.lang.Error

  type RuntimeException = java.lang.RuntimeException
  type NullPointerException = java.lang.NullPointerException
  type ClassCastException = java.lang.ClassCastException
  type IndexOutOfBoundsException =
        java.lang.IndexOutOfBoundsException
  type ArrayIndexOutOfBoundsException =
        java.lang.ArrayIndexOutOfBoundsException
  type StringIndexOutOfBoundsException =
        java.lang.StringIndexOutOfBoundsException
  type UnsupportedOperationException =
        java.lang.UnsupportedOperationException
  type IllegalArgumentException =
java.lang.IllegalArgumentException
  type NoSuchElementException = java.util.NoSuchElementException
  type NumberFormatException = java.lang.NumberFormatException

  // 杂项 -----------------------------------------------

  type Function[-A, +B] = Function1[A, B]

  type Map[A, B] = collection.immutable.Map[A, B]
  type Set[A] = collection.immutable.Set[A]

  val Map = collection.immutable.Map
  val Set = collection.immutable.Set

  // 错误与断言 -----------------------------------------

  def error(message: String): Nothing = throw new Error(message)

  def exit: Nothing = exit(0)

  def exit(status: Int): Nothing = {
    java.lang.System.exit(status)
    throw new Throwable()
```

```
}

def assert(assertion: Boolean) {
  if(!assertion)
   throw new java.lang.AssertionError("assertion failed")
}

def assert(assertion: Boolean, message: Any) {
  if(!assertion)
    throw new java.lang.ArrsertionError("assertion failed: " + message)
}

def assume(assumption: Boolean, message: Any) {
  if(!assumption)
    throw new IllegalArgumentException(message.toString)
}

// 元组 ----------------------------------------------

type Pair[+A, +B] = Tuple2[A, B]
object Pair {
  def apply[A, B](x: A, y: B) = Tuple2(x, y)
  def unapply[A, B](x: Tuple2[A, B]): Option[Tuple2[A, B] =
      Some(x)
}

type Triple[+A, +B, +C] = Tuple3[A, B, C]
object Trible {
  def apply[A, B, C](x: A, y: B, z: C) = Tuple3(x, y, z)
  def unapply[A, B, C](x: Tuple3[A, B, C]):
    Option[Tuple3[A, B, C]] = Some(x)
}

class ArrowAssoc[A](x: A){
  def -> [B](y: B): Tuple2[A, B] = Tuple2(x, y)
}

implicit def any2ArrowAssoc[A](x: A): ArrowAssoc[A] =
   new ArrowAssoc(x)

// 打印与读取 ----------------------------------

def print(x: Any) = Console.print(x)
def println() = Console.println()
```

```
  def println(x: Any) = Console.println(x)
  def printf(text: String, xs: Any*) = Console.printf(text, xs:_*)
  def format(test: String, xs: Any*) =
    Console.format(text, sx:_*)

  def readLine(): String = Console.readLine()
  def readLine(test: String, args: Any*) = Console.readLine(text, args)
  def readBoolean() = Console.readBoolean()
  def readByte() = Console.readByte()
  def readShort() = Console.readShort()
  def readChar() = Console.readChar()
  def readInt() = Console.readInt()
  def readLong() = Console.readLong()
  def readFloat() = Console.readFloat()
  def readDouble() = Console.readDouble()
  def readf(format: String) = Console.readf(format)
  def readf1(format: String) = Console.readf1(format)
  def readf2(format: String) = Console.readf2(format)
  def readf3(format: String) = Console.readf3(format)

  // "catch-all"隐式约定
  implicit def identity[A](x: A): A = x

  // Ordered 类中的视图

  implicit def int2ordered(x: Int): Ordered[Int] =
    new Ordered[Int] with Proxy {
      def self: Any = x
      def compare[B >: Int <% OrderedB]](y: B): Int = y match {
        case y1: Int =>
          if (x < y1) -1
          else if (x > y1) 1
          else 0
        case _ => -(y compare x)
      }
  }

  // 以下方法的实现和上一个类似：

  implicit def char2ordered(x: Char): Ordered[Char] = ...
  implicit def long2ordered(x: Long): Ordered[Long] = ...
  implicit def float2ordered(x: Float): Ordered[Float] = ...
  implicit def double2ordered(x: Double): Ordered[Double] = ...
  implicit def boolean2ordered(x: Boolean): Ordered[Boolean] = .
```

```
implicit def seq2ordered[A <% Ordered[A]](xs: Array[A]):
      Ordered[Seq[A]] = new Ordered[Seq[A]] with Proxy {
  def compare[B >: Seq[A] <% Ordered[B]](that: B): Int =
    that match {
      case that: Seq[A] =>
        var res = 0
        var these = this.elements
        var those = that.elements
        while (res == 0 && these.hasNext)
          res = if (!those.hasNext) 1 else these.next compare those.next
          case _ => - (that.compare xs)
      }
  }
}

implicit def string2ordered(x: String): Ordered[String] =
  new Ordered[String] with Proxy {
    def self: Any = x
    def compare [b >: String <% Ordered[b]](y: b): Int = y match {
      case y1: String => x compare y1
      case _ = - (y compare x)
    }
  }

implicit def tuple2ordered[a1 <% Ordered[a1],
a2 <% Ordered[a2]](x: Tuple2[a1, a2]): Ordered[Tuple2[a1, a2]] =
  new Ordered[Tuple2[a1, a2]] with Proxy {
    def self: Any = x
    def compare[T >: Tuple2[a1, a2] <% Ordered[T]](y: T): Int =
      y match {
        case y: Tuple2[a1, a2] =>
          val res = x._1 compare y._1
          if (res == 0) x._2 compare y._2
          else res
        case _ => - (y compare x)
      }
    }

// Tuple3 到 Tuple9 类似

// Seq 类中的视图

implicit def string2seq(str: String): Seq[Char] =
```

```
    new Seq[Char] {
      def length = str.length
      def elements = Iterator.fromString(str)
      def apply(n: Int) = str.charAt
      override def hashCode: Int = str.hashCode
      override def equals(y: Any): Boolean = (str == y)
      override protected def stringPrefix: String = "String"
    }

  //从原生类型到 Java 装箱类型的视图

  implicit def byte2Byte(x: Byte) = new java.lang.Byte(x)
  implicit def short2Short(x: Short) = new java.lang.Short(x)
  implicit def char2Character(x: Char) = new java.lang.Character(x)
  implicit def int2Integer(x: Int) = new java.lang.Integer(x)
  implicit def long2Long(x: Long) = new java.lang.Long(x)
  implicit def float2Float(x: Float) = new java.lang.Float(x)
  implicit def double2Double(x: Double) = new java.lang.Double(x)
  implicit def boolean2Boolean(x: Boolean) = new java.lang.Boolean(x)

  // 数字转换视图

  implicit def byte2short(x: Byte): Short = x.toShort
  implicit def byte2int(x: Byte): Int = x.toInt
  implicit def byte2long(x: Byte): Long = x.toLong
  implicit def byte2float(x: Byte): Float = x.toFloat
  implicit def byte2double(x: Byte): Double = x.toDouble

  implicit def short2int(x: Short): Int = x.toInt
  implicit def short2long(x: Short): Long = x.toLong
  implicit def short2float(x: Short): Float = x.toFloat
  implicit def short2double(x: Short): Double = x.toDouble

  implicit def char2int(x: Char): Int = x.toInt
  implicit def char2long(x: Char): Long = x.toLong
  implicit def char2float(x: Char): Float = x.toFloat
  implicit def char2double(x: Char): Double = x.toDouble

  implicit def int2long(x: Int): Long = x.toLong
  implicit def int2float(x: Int): Float = x.toFloat
  implicit def int2double(x: Int): Double = x.toDouble

  implicit def long2float(x: Long): Float = x.toFloat
  implicit def long2double(x: Long): Double = x.toDouble
```

```
 implicit def float2double(x: Float): Double = x.toDouble
}
```

后记

我觉得 Scala 是一门很好的语言，成功揉合了 OO 和 FP。希望该译本能给大家学习 Scala 语言带来帮助。

赵炜(wzhao1984@gmail.com)是我的同事。在翻译过程中提了不少建议，并全文翻译了第八章。同时来自 EPFL 编程方法实验室的 Antonio Cunei 也给了不少帮助和指导，并协助发布了此文档，在此一并表示感谢。

翻译的起始时间大约是 2009 年 8 月份，翻译过程总历时约一年，总耗时约几百个小时。

原文的参考书目(Bibliography)语法总结(Scala Syntax Summary)和更新历史(Change Log)部分未做翻译，也没有包含在本文档中，请参考原文获取相关信息。

由于水平所限，错误之处在所难免，如果不幸被您发现了，烦请发一封 email 至 gao_de@163.com，并注明页码章节，我将会在下个版本更新时一并更正。谢谢。

高德

2010 年 7 月 20 日
于 上海