## 通过反射访问和调用运行时类型的成员

1. 定义一个带成员变量的类和对象

case class *Purchase*(*name*: String, *orderNumber*: *Int*, *shipped*: *Boolean*) {  
 def helloWorld: *Unit* = *println*("hello world")  
}

val *p* = Purchase("Jeff Lebowski", 23819, *shipped* = false)

1. 获取一个ClassLoader mirror，通过这个mirror就可以获取到当前classloader中的所有类和类型。

val *m* = *ru*.runtimeMirror(getClass.getClassLoader)

1. 通过classLoader mirror调用reflect方法可获取到一个invoker mirror 类型为InstanceMirror。

val *im* = *m*.reflect(*p*)

1. 通过universe方法，获取目标的TermSymbol。

val *shippingTermSymb* = *ru*.typeOf[*Purchase*].decl(*ru*.TermName("shipped")).asTerm

1. 通过第三步获取到的invoker mirror传入第四步获取到的TermSymbol可以获取到一个invoker mirror，其类型为FieldMirror。

*ppingFieldMirror* = *im*.reflectField(*shippingTermSymb*)

1. 通过FieldMirror调用get或者set方法就能访问和调用成员变量

*ppingFieldMirror .set(true)*

*ppingFieldMirror .get*

# 编译时反射

通过编译时反射能实现元编程，即是让程序本身在编译时就能修改自己的编程技术。编译时反射是通过宏来实现的，通过宏，scala提供了在编译时执行方法和操作AST的能力。

编译时反射和运行时反射都是使用了一组API，在“scala.reflect.api”包中。这样就可以在宏和使用运行时反射的实现之间共享通用代码。

跟运行时反射相比，编译时反射除了一些特定的API以外，他们都共享很多共有概念像AST等。

# Symbols

Symbols的作用是在名称和实体之间共建一个联系，比如一个类或者一个方法。你在scala中定义的任何东西都能与它各自的symbol相联系。它是编译时反射和运行时反射共享的一个整体的抽象中心。

* Symbol是以层级关系构建的，叶子节点的symbol（比如一个方法有一个方法symbol）被更上一层叶子节点的的symbol（类Symbol）所拥有，最上一层的symbol（比如顶层package）的owner是一个特殊的NoSymbol的单例对象。
* TypeSymbol代表一个type，class或者一个类型参数。它有几个特殊的方法分别是isAbstractType，isContravariant，isCovariant。
* ClassSymbol是TypeSymbol的子类，它提供了一个类或者特质声明的所有信息，包括name, modifiers (isFinal, isPrivate, isProtected, isAbstractClass, etc.), baseClasses, and typeParams.
* TermSymbol代表val，var，def，对象声明和packages参数和value参数，它有两个子类分别是MethodSymbol和ModuleSymbol。
* 通过asClass或asMethod类似的惯用法可以把一个宽泛的Symbol转义成一个更具体的symbol，比如ru.typeOf[C[Int]].member(TermName("test"))获取一个宽泛的Symbol通过asMethod可以把这个Symbol转换成一个MethodSymbol。
* FreeSymbols分成两种FreeTermSymbol 和FreeTypeSymbol，表示这个symboll的信息是不全面的。这些符号是在具体化过程中的某些情况下生成的，每当具体化无法定位一个符号时(这意味着该符号在相应的类文件中不可用，例如，因为该符号引用了一个局部类)，它就会将其具体化为所谓的“free type”。

# Types

Type的实例代表了一个对应的symbol的类型的具体信息，这包含了它直接定义或者继承的的members（包括方法、fields、类型别名、抽象类型、嵌套类、特质等），它的Base Types，它的erasure等。同时也包含了一些验证type一致性和相等性的操作。

1. Types的初始化
   1. 通过typeof

大多数情况下首选这个方法，比如ru.typeOf[List[Int]]会返回一个包含List[Int]信息的type，

如果我们不知道类型的情况下像获取对应type的实例应该怎么办呢？通过定义一个带有上下文界type的方法就可以实现

def getType[T: TypeTag](obj: T) = typeOf[T]

这种写法等同

def getType [T](obj: T)(implicit evidence$1: sru.TypeTag[T])ru.Type

注：这种方法不适合于带type parameters的types，比如typeof[List[A]]其中A是个type parameter。这种情况需要使用weakTypeOf代替。

* 1. 通过标准type,比如通过可用的universe访问到的Int，Boolean，Any，Unit

比如val intTpe = universe.definitions.IntTpe

* 1. 使用工程方法typeRef和polyType（不推荐）

1. Types的通用操作
   1. 检查两个type之间是否存在子类型关系

使用<:<来检查两个type之间的关系

scala> **class** **A**; **class** **B** **extends** **A**

defined **class** **A** defined **class** **B**

scala> typeOf[**A**] <:< typeOf[**B**]

res0: **Boolean** = false

scala> typeOf[**B**] <:< typeOf[**A**]

res1: **Boolean** = true

注：检查两个type之间的弱一致性（weak conformance）使用weak\_<:<。

Scala的类型中存在如下的弱一致性

|  |
| --- |
| Byte <:w Short |
| Short <:w Int |
| Char <:w Int |
| Int <:w Long |
| Long <:w Float |
| Float <:w Double |

这也是为什么如下的返回是一个double的原因

if (true) 1 else 1d

但Double和Int只存在弱一致性，不存在子类型关系。

* 1. 检查两个type的相等性

使用=:=来检查两个type类型的相等性

scala> getType(**List**(1,2,3)) =:= getType(**List**(1.0, 2.0, 3.0))

res1: **Boolean** = false

scala> getType(**List**(1,2,3)) =:= getType(**List**(9,8,7))

res2: **Boolean** = true

scala> type Histogram = List[Int]

defined type alias Histogram

scala> typeOf[Histogram] =:= getType(List(4,5,6))

res3: Boolean = true

scala> typeOf[Histogram] == getType(List(4,5,6))

res4: Boolean = false

* 1. 访问一个type的成员和内部type

通过一个已知的type，可以检查他的特定成员（members）或定义（declaration），type的成员包括所有的fields、methods、type aliases, abstract Types, nested classes/objects/traits。Type的定义类型的声明仅是在给定类型所代表的类/trait/对象定义中声明(而不是继承)的成员。

使用TermName从type中获取一个方法，使用TypeName获取type member

scala> typeOf[**List**[\_]].member(**TermName**("map"))

res0: scala.reflect.runtime.universe.**Symbol** = method map

scala> typeOf[**List**[\_]].member(**TypeName**("Self"))

res1: scala.reflect.runtime.universe.**Symbol** = **type** **Self**

**同时可以访问所有的方法，使用高阶函数可以列出我们感兴趣的信息。**

scala> typeOf[**List**[**Int**]].members.filter(\_.isPrivate).foreach(println \_)

# Trees

在反射中，通过APIs支持以下几种方法产生或者使用AST。而且除了pos（位置）、symbol（Symbol）和tpe（Type）需要在typeChekced阶段被赋值的字段之外，树是不可变的。

1. Scala的annotations，使用trees来代表他们的参数，通过Annotation.scalaArgs来使用。
2. Reify，它是一个特殊的方法接受一个表达式并返回对应的AST。
3. 使用宏的编译时反射
4. 使用工具箱的运行时编译都使用树作为它们的程序表示媒介。

### 树的种类

有三种主要的树类型：

1. TermTree的子类，比如方法的调用使用Apply node代表，对象的初始化使用New node代表。
2. TypTree的子类，代表那些显式出现在源代码中的types，比如List[Int]被parser成了AppliedTypeTree。
3. SymTree的子类，代表定义，比如class或者trait的定义使用ClassDef来代表，field或者参数的定义使用ValDef来代表。

### 树的遍历

1. 使用模式匹配

对于这样的定义

tree = **Apply**(**Select**(**Ident**(**TermName**("x")), **TermName**("$plus")), **List**(**Literal**(**Constant**(2))))

通过如下的模式匹配的方式可以提取内容

scala> **val** (fun, arg) = tree **match** {

| **case** **Apply**(fn, a :: **Nil**) => (fn, a)

| }

fun: scala.reflect.runtime.universe.**Tree** = x.$plus

arg: scala.reflect.runtime.universe.**Tree** = 2

或者更简洁一点

scala> **val** **Apply**(fun, arg :: **Nil**) = tree

fun: scala.reflect.runtime.universe.**Tree** = x.$plus

arg: scala.reflect.runtime.universe.**Tree** = 2

模式匹配来遍历树适合于简单情况，对于复杂的树的遍历使用Traverser来遍历。

1. 使用Traverser

Traverser会保证以深度优先的方法遍历树的每个节点，只需要简单的拓展Traverser基类，并重写traverse方法。比如：

scala> **object** **traverser** **extends** **Traverser** {

| **var** applies = **List**[**Apply**]()

| **override** **def** **traverse**(tree: **Tree**): **Unit** = tree **match** {

| **case** app @ **Apply**(fun, args) =>

| applies = app :: applies

| **super**.traverse(fun)

| **super**.traverseTrees(args)

| **case** \_ => **super**.traverse(tree)

| }

| }

这个函数将会收集所有的Apply的node，对一个AST调用traverser.traverse(tree)就可以收集所有Apply的node。

### 树的创建

1. 使用reify

它是一个特殊的方法接受一个表达式并返回对应的AST。

比如val tree = reify(println(2)).tree

通过showRaw方法我们就能得到println(2)的AST

**Apply**(**Select**(**Select**(**This**(**TypeName**("scala")), **TermName**("Predef")), **TermName**("println")), **List**(**Literal**(**Constant**(2))))

注：println(2)会被当成scala.Predef.println(2)

1. 在ToolBox中使用parse方法

ToolBox是打包在Scala-compiler.jar中的一个类库，它可以被用在类型检查，编译以及执行AST，同时它也可以把一个字符串parser成AST。

比如：

scala> **import** scala.reflect.runtime.universe.\_

**import** scala.reflect.runtime.universe.\_

scala> **import** scala.tools.reflect.**ToolBox**

**import** scala.tools.reflect.**ToolBox**

scala> **val** tb = runtimeMirror(getClass.getClassLoader).mkToolBox()

tb: scala.tools.reflect.**ToolBox**[scala.reflect.runtime.universe.**type**] = scala.tools.reflect.**ToolBoxFactory**$**ToolBoxImpl**@7bc979dd

scala> showRaw(tb.parse("println(2)"))

res2: **String** = **Apply**(**Ident**(**TermName**("println")), **List**(**Literal**(**Constant**(2))))

除了程序的框架结构AST，树也通过symbol的方式包含了关于程序的语义信息，这些语义信息在构建之初是没有的，在typeChecking阶段才会被添加进去。typeChecking在运行时反射使用ToolBox.typeCheck在编译时反射使用Context.typeCheck方法。

1. 手动创建一个树

# 注释（Annotations）

在scala中使用