尚硅谷大数据技术之 Scala
扩展资料

官网：www.atguigu.com

ShangGuigu Technologies Co., Ltd.

尚硅谷技术有限公司

一　注解

注解就是标签。

标签是用来标记某些代码需要特殊处理的。

处理的手段可以在代码运行时操作，也可以在编译期操作。

1.1　什么可以被注解

1)　可以为类，方法，字段局部变量，参数，表达式，类型参数以及各种类型定义添加注解

```
@Entity class Student

@Test def play() {}

@BeanProperty var username = _

def doSomething(@NotNull message: String) {}

@BeanProperty @Id var username = _
```

2)　构造器注解，需要在主构造器之前，类名之后，且需要加括号，如果注解有参数，则写在注解括号里

```
class Student @Inject() (var username: String, var password: String)
```

3)　为表达式添加注解，在表达式后添加冒号

```
（map1.get(key): @unchecked) match {...}
```

4)　泛型添加注解

```
class Student[@specialized T]
```

5)　实际类型添加注解

```
String @cps[Unit]
```

1.2　注解参数

Java 注解可以有带名参数：

```
@Test(timeout = 100, expected = classOf[IOException])

// 如果参数名为 value,则该名称可以直接略去。

@Named("creds") var credentials: Credentials = _　// value 参数的值为 “creds”
```

```
// 注解不带参数，圆括号可以省去
@Entity class Credentials
```

Java 注解的参数类型只能是：

数值型的字面量

字符串

类字面量

Java 枚举

其他注解

上述类型的数组（但不能是数组的数组）

Scala 注解可以是任何类型，但只有少数几个 Scala 注解利用了这个增加的灵活性。

1.3 注解实现

你可以实现自己的注解，但是更多的是使用 Scala 和 Java 提供的注解。

注解必须扩展 Annotation 特质：

```
class unchecked extends annotation.Annotation
```

1.4 针对 Java 的注解

1) Java 修饰符：对于那些不是很常用的 Java 特性，Scala 使用注解，而不是修饰符关键字。

```
@volatile var done = false  // JVM 中将成为 volatile 的字段
@transient var recentLookups = new HashMap[String, String]  // 在 JVM 中将成为 transient 字段，该字段不会被序列化。
@strictfp def calculate(x: Double) = ...
@native def win32RegKeys(root: Int, path: String): Array[String]
```

2) 标记接口：Scala 用注解@cloneable 和@remote 而不是 Cloneable 和 Java.rmi.Remote “标记接口”来标记可被克隆的对象和远程的对象。

```
@cloneable class Employee
```

3) 受检异常：和 Scala 不同，Java 编译器会跟踪受检异常。如果你从 Java 代码中调用 Scala

的方法，其签名应包含那些可能被抛出的受检异常。用@throws 注解来生成正确的签名。

```
class Book {
@throws (classOf[IOException]) def read(filename: String) { ... }
...
}
Java 版本的方法签名：
void read(String fileName) throws IOException
// 如果没有@throws 注解，Java 代码将不能捕获该异常
try {//Java 代码
book.read("war-and-peace.txt");
} catch (IOException ex) {
...
}
```

即：Java 编译期需要在编译时就知道 read 方法可以抛 IOException 异常，否则 Java 会拒绝捕获该异常。

1.5 用于优化的注解

尾递归的优化

啥玩是尾递归？

尾递归：

```
def story(): Unit = {从前有座山，山上有座庙，庙里有个老和尚，一天老和尚对小和尚讲故事：story()}
```

注：进入下一个函数不再需要上一个函数的环境了，得出结果以后直接返回。

非尾递归：

```
def story(): Unit =  {从前有座山，山上有座庙，庙里有个老和尚，一天老和尚对小和尚讲故事：story()，小和尚听了，找了块豆腐撞死了}
```

注：下一个函数结束以后此函数还有后续，所以必须保存本身的环境以供处理返回值。

递归调用有时候能被转化成循环，这样能节约栈空间：

```
object Util {
def sum(xs: Seq[Int]): BigInt = {
  if (xs.isEmpty) 0 else xs.head + sum(xs.tail)
}
...
}
```

上面的 sum 方法无法被优化，因为计算过程中最后一步是加法，而不是递归调用。调整后的代码：

```
def sum2(xs: Seq[Int], partial: BigInt): BigInt = {
if (xs.isEmpty) partial else sum2(xs.tail, xs.head + partial)
}
```

Scala 编译器会自动对 sum2 应用“尾递归”优化。如果你调用 sum(1 to 1000000) 将会发生一个栈溢出错误。不过 sum2(1 to 1000000, 0) 将会得到正确的结果。

尽管 Scala 编译器会尝试使用尾递归优化，但有时候某些不太明显的原因会造成它无法这样做。如果你想编译器无法进行优化时报错，则应该给你的方法加上@tailrec 注解。

注：对于消除递归，一个更加通用的机制叫做“蹦床”。蹦床的实现会将执行一个循环，不停的调用函数。每个函数都返回下一个将被调用的函数。尾递归在这里是一个特例，每个函数都返回它自己。Scala 有一个名为 TailCalls 的工具对象，帮助我们轻松实现蹦床：

```
import scala.util.control.TailCalls._
def evenLength(xs: Seq[Int]): TailRec[Boolean] = {
   if(xs.isEmpty) done(true) else tailcall(oddLength(xs.tail))
}

def oddLength(xs: Seq[Int]): TailRec[Boolean] = {
   if(xs.isEmpty) done(false) else tailcall(evenLength(xs.tail))
}
```

```
// 获得 TailRec 对象获取最终结果，可以用 result 方法
evenLength(1 to 1000000).result
```

二　类型参数

2.1　泛类型

类和特质都可以带类型参数，用方括号来定义类型参数，可以用类型参数来定义变量、方法参数和返回值。带有一个或多个类型参数的类是泛型的。如下 p1，如果实例化时没有指定泛型类型，则 scala 会自动根据构造参数的类型自动推断泛型的具体类型。

```
class Pair[T, S](val first: T, val second: S) {
    override def toString = "(" + first + "," + second + ")"
}
//从构造参数推断类型
val p1 = new Pair(42, "String")
//设置类型
val p2 = new Pair[Any, Any](42, "String")
```

2.2　泛型函数

函数或方法也可以有类型（泛型）参数。

```
// 从参数类型来推断类型
println(getMiddle(Array("Bob", "had", "a", "little", "brother")).getClass.getTypeName)
//指定类型，并保存为具体的函数。
val f = getMiddle[String] _
println(f(Array("Bob", "had", "a", "little", "brother")))
```

2.3　类型变量限定

在 Java 泛型里不表示某个泛型是另外一个泛型的子类型可以使用 extends 关键字，而在 scala 中使用符号“<:”，这种形式称之为泛型的上界。

```
class Pair1[T <: Comparable[T]](val first: T, val second: T) {
```

```
    def smaller = if (first.compareTo(second) < 0) first else second
}

object Main1 extends App{
    override def main(args: Array[String]): Unit = {
        val p = new Pair1("Fred", "Brooks")
        println(p.smaller)
    }
}
```

在 Java 泛型里表示某个泛型是另外一个泛型的父类型，使用 super 关键字，而在 scala 中，使用符号“>:”，这种形式称之为泛型的下界。

```
class Pair2[T](val first: T, val second: T) {
    def replaceFirst[R >: T](newFirst: R) = new Pair2[R](newFirst, second)
    override def toString = "(" + first + "," + second + ")"
}

object Main2 extends App{
    override def main(args: Array[String]): Unit = {
        val p = new Pair2("Nick", "Alice")
        println(p)
        println(p.replaceFirst("Joke"))
        println(p)
    }
}
```

在 Java 中，T 同时是 A 和 B 的子类型，称之为多界，形式如：<T extends A & B>。

在 Scala 中，对上界和下界不能有多个，但是可以使用混合类型，如：[T <: A with B]。

在 Java 中，不支持下界的多界形式。如:<T super A & B>这是不支持的。

在 Scala 中，对复合类型依然可以使用下界，如：[T >: A with B]。

2.5 视图界定

在 Scala 中，如果你想标记某一个泛型可以隐式的转换为另一个泛型，可以使用：[T <% Comparable[T]]，由于 Scala 的 Int 类型没有实现 Comparable 接口，所以我们需要将 Int 类型隐式的转换为 RichInt 类型，比如：

```
class Pair3[T <% Comparable[T]](val first: T, val second: T) {
  def smaller = if (first.compareTo(second) < 0) first else second
  override def toString = "(" + first + "," + second + ")"
}

object Main3 extends App {
  val p = new Pair3(4, 2)
  println(p.smaller)
}
```

2.6 上下文界定

视图界定 T <% V 要求必须存在一个从 T 到 V 的隐式转换。上下文界定的形式为 T:M，其中 M 是另一个泛型类，它要求必须存在一个类型为 M[T]的隐式值。

下面类定义要求必须存在一个类型为 Ordering[T]的隐式值，当你使用了一个使用了隐式值得方法时，传入该隐式参数。

```
class Pair4[T: Ordering](val first: T, val second: T) {
  def smaller(implicit ord: Ordering[T]) = {
    println(ord)
    if (ord.compare(first, second) < 0) first else second
  }
```

```
  override def toString = "(" + first + "," + second + ")"
}

object Main4 extends App{
  override def main(args: Array[String]): Unit = {
    val p4 = new Pair4(1, 2)
    println(p4.smaller)
  }
}
```

2.7 Manifest 上下文界定

Manifest 是 scala2.8 引入的一个特质，用于编译器在运行时也能获取泛型类型的信息。在 JVM 上，泛型参数类型 T 在运行时是被“擦拭”掉的，编译器把 T 当作 Object 来对待，所以 T 的具体信息是无法得到的；为了使得在运行时得到 T 的信息，scala 需要额外通过 Manifest 来存储 T 的信息，并作为参数用在方法的运行时上下文。

def test[T] (x:T, m:Manifest[T]) { ... }

有了 Manifest[T]这个记录 T 类型信息的参数 m，在运行时就可以根据 m 来更准确的判断 T 了。但如果每个方法都这么写，让方法的调用者要额外传入 m 参数，非常不友好，且对方法的设计是一道伤疤。好在 scala 中有隐式转换、隐式参数的功能，在这个地方可以用隐式参数来减轻调用者的麻烦。

```
def foo[T](x: List[T])(implicit m: Manifest[T]) = {
  println(m)
  if (m <:< manifest[String])
    println("Hey, this list is full of strings")
  else
    println("Non-stringy list")
}
```

```
foo(List("one", "two"))
foo(List(1, 2))
foo(List("one", 2))
```

隐式参数 m 是由编译器根据上下文自动传入的，比如上面是编译器根据 "one","two" 推断出 T 的类型是 String，从而隐式的传入了一个 Manifest[String]类型的对象参数，使得运行时可以根据这个参数做更多的事情。

不过上面的 foo 方法定义使用隐式参数的方式，仍显得啰嗦，于是 scala 里又引入了“上下文绑定”，

def foo[T](x: List[T]) (implicit m: Manifest[T])

可以简化为：

def foo[T:Manifest] (x: List[T])

在引入 Manifest 的时候，还引入了一个更弱一点的 ClassManifest，所谓的弱是指类型信息不如 Manifest 那么完整，主要针对高阶类型的情况

scala 在 2.10 里却用 TypeTag 替代了 Manifest，用 ClassTag 替代了 ClassManifest，原因是在路径依赖类型中，Manifest 存在问题：

```
scala> class Foo{class Bar}
defined class Foo

scala> def m(f: Foo)(b: f.Bar)(implicit ev: Manifest[f.Bar]) = ev
warning: there were 2 deprecation warnings; re-run with -deprecation for details
m: (f: Foo)(b: f.Bar)(implicit ev: Manifest[f.Bar])Manifest[f.Bar]

scala> val f1 = new Foo;val b1 = new f1.Bar
f1: Foo = Foo@681e731c
b1: f1.Bar = Foo$Bar@271768ab

scala> val f2 = new Foo;val b2 = new f2.Bar
```

```
f2: Foo = Foo@3e50039c

b2: f2.Bar = Foo$Bar@771d16b9


scala> val ev1 = m(f1)(b1)

warning: there were 2 deprecation warnings; re-run with -deprecation for details

ev1: Manifest[f1.Bar] = Foo@681e731c.type#Foo$Bar


scala> val ev2 = m(f2)(b2)

warning: there were 2 deprecation warnings; re-run with -deprecation for details

ev2: Manifest[f2.Bar] = Foo@3e50039c.type#Foo$Bar


scala> ev1 == ev2 // they should be different, thus the result is wrong

res28: Boolean = true
```

了解之后，我们总结一下，TypeTag 到底有啥用呢？看下面的例子：

请留意：

=:=，意思为：type equality

<:< ,意思为：subtype relation

类型判断不要用 == 或 !=

```
class Animal{}

class Dog extends Animal{}


object MainFoo extends App{

  override def main(args: Array[String]): Unit = {

    val list1 = List(1, 2, 3)

    val list2 = List("1", "2", "3")

    val list3 = List("1", "2", 3)
```

```
    def test1(x: List[Any]) = {
      x match {
        case list: List[Int] => "Int list"
        case list: List[String] => "String list"
        case list: List[Any] => "Any list"
      }
    }
    println(test1(list1))
    println(test1(list2))
    println(test1(list3))

    import scala.reflect.runtime.universe._
    def test2[A : TypeTag](x: List[A]) = typeOf[A] match {
      case t if t =:= typeOf[String] => "String List"
      case t if t <:< typeOf[Animal] => "Dog List"
      case t if t =:= typeOf[Int] => "Int List"
    }

    println(test2(List("string")))
    println(test2(List(new Dog)))
    println(test2(List(1, 2)))
  }
}
```

2.8 多重界定

不能同时有多个上界或下界，变通的方式是使用复合类型

T <: A with B

T >: A with B

可以同时有上界和下界，如

T >: A <: B

这种情况下界必须写在前边，上界写在后边，位置不能反。同时 A 要符合 B 的子类型，A 与 B 不能是两个无关的类型。

可以同时有多个 view bounds

T <% A <% B

这种情况要求必须同时存在 T=>A 的隐式转换，和 T=>B 的隐式转换。

```
class A{}
class B{}
implicit def string2A(s:String) = new A
implicit def string2B(s:String) = new B
def foo2[ T <% A <% B](x:T)    = println("foo2 OK")
foo2("test")
```

可以同时有多个上下文界定

T : A : B

这种情况要求必须同时存在 C[T]类型的隐式值，和 D[T]类型的隐式值。

```
class C[T];
class D[T];
implicit val c = new C[Int]
implicit val d = new D[Int]
def foo3[ T : C : D ](i:T) = println("foo3 OK")
foo3(2)
```

2.9 类型约束

类型约束，提供了限定类型的另一种方式，一共有 3 中关系声明：

T =:= U 意思为：T 类型是否等于 U 类型

T <:< U 意思为：T 类型是否为 U 或 U 的子类型

T <%< U 意思为：T 类型是否被隐式（视图）转换为 U

如果想使用上面的约束，需要添加“隐式类型证明参数”比如：

```
class Pair5[T] (val first: T, val second: T)(implicit ev: T <:< Comparable[T]){}
```

使用举例：

```
import java.io.File


class Pair6[T](val first: T, val second: T) {
  def smaller(implicit ev: T <:< Ordered[T]) = {
    if(first < second) first else second
  }
}


object Main6 extends App{
  override def main(args: Array[String]): Unit = {
    //构造 Pair6[File]时，注意此时是不会报错的
    val p6 = new Pair6[File](new File(""), new File(""))
    //这就报错了
    p6.smaller
  }
}
```

2.10 型变

术语：

英文	中文	示例
Variance	型变	Function[-T, +R]
Nonvariant	不变	Array[A]
Covariant	协变	Supplier[+A]
Contravariant	逆变	Consumer[-A]

Immutable	不可变	String
Mutable	可变	StringBuilder

其中，Mutable 常常意味着 Nonvariant，但是 Noncovariant 与 Mutable 分别表示两个不同的范畴。

即：可变的，一般意味着“不可型变”，但是“不可协变”和可变的，分别表示两个不同范畴。

型变(Variance)拥有三种基本形态：协变(Covariant)，逆变(Contravariant)，不变(Nonconviant)，可以形式化地描述为：

一般地，假设类型 C[T]持有类型参数 T；给定两个类型 A 和 B，如果满足 A <: B，则 C[A]与 C[B]之间存在三种关系：

如果 C[A] <: C[B]，那么 C 是协变的(Covariant);

如果 C[A] :> C[B]，那么 C 是逆变的(Contravariant);

否则，C 是不变的(Nonvariant)。

Scala 的类型参数使用+标识“协变”，-标识“逆变”，而不带任何标识的表示“不变”(Nonvariable)：

```
trait C[+A]     // C is covariant
trait C[-A]     // C is contravariant
trait C[A]      // C is nonvariant
```

如何判断一个类型是否有型变能力：

一般地，“不可变的”(Immutable)类型意味着“型变”(Variant)，而“可变的”(Mutable)意味着“不变”(Nonvariant)。

其中，对于不可变的(Immutable)类型 C[T]

如果它是一个生产者，其类型参数应该是协变的，即 C[+T]；

如果它是一个消费者，其类型参数应该是逆变的，即 C[-T]。

三　文件和正则表达式

3.1　读取行

```
import scala.io.Source


object FileSyllabus {
   def main(args: Array[String]): Unit = {
      //文件读取
      val file1 = Source.fromFile("C:\\Users\\61661\\Desktop\\scala 笔记.txt")
      val lines = file1.getLines
      for (line <- lines) {
         println(line)
      }
      file1.close
   }
}
```

注：记得 close

1) 文件内容转数组：

```
val array= file1.getLines.toArray
```

2) 文件内容转字符串：

```
val iterator = file1.mkString
```

3.2 读取字符

由于 Source.fromFile 直接返回的就是 Iterator[Char]，所以可以直接对其进行迭代，按照字符访问里边每一个元素。

```
Source.fromFile("C:\\Users\\61661\\Desktop\\scala 笔记.txt", "UTF-8")
for(ch <- file2){
   println(ch)
}
file2.close
```

3.3 读取词法单元和数字

如果想将以某个字符或某个正则表达式分开的字符成组读取，可以这么做：

```
val file3 = Source.fromFile("D:\\BigData 课堂笔记\\尚硅谷 BigData 笔记\\尚硅谷大数据技术之Scala\\2.资料\\info.csv")
val tokens = file3.mkString.split(",")
println(tokens.mkString(" "))
file3.close
```

3.4 读取网络资源、文件写入、控制台操作

1) 读取网络资源

```
val webFile = Source.fromURL("http://www.baidu.com")
webFile.foreach(print)
webFile.close()
```

2) 写入数据到文件

```
import java.io.{File, PrintWriter}
val writer = new PrintWriter(new File("嘿嘿嘿.txt"))
for (i <- 1 to 100)
    writer.println(i)
writer.close()
```

3) 控制台操作

```
//控制台交互--老 API
print("请输入内容:")
val consoleLine1 = Console.readLine()
println("刚才输入的内容是:" + consoleLine1)

//控制台交互--新 API
print("请输入内容(新 API):")
```

```
val consoleLine2 = StdIn.readLine()
println("刚才输入的内容是:" + consoleLine2)
```

3.5 序列化

```
@SerialVersionUID(1L) class Person extends Serializable{
    override def toString = name + "," + age

    val name = "Nick"
    val age = 20

}

object PersonMain extends App{
    override def main(args: Array[String]): Unit = {

        import java.io.{FileOutputStream, FileInputStream, ObjectOutputStream,
ObjectInputStream}
        val nick = new Person
        val out = new ObjectOutputStream(new FileOutputStream("Nick.obj"))
        out.writeObject(nick)
        out.close()

        val in = new ObjectInputStream(new FileInputStream("Nick.obj"))
        val saveNick = in.readObject()
        in.close()
        println(saveNick)
    }
}
```

3.6 进程控制

我们可以使用 scala 来操作 shell，scala 提供了 scala.sys.process 包提供了用于 shell 程序交互的工具。

1) 执行 shell

```
import sys.process._
"ls -al /"!;
"ls -al /"!!
```

注：!和!!的区别在于：process 包中有一个将字符串隐式转换成 ProcessBuild 对象的功能，感叹号就是执行这个对象，单感叹号的意思就是程序执行成功返回 0，执行失败返回非 0，如果双感叹号，则结果以字符串的形式返回。

2) 管道符

```
import sys.process._
"ls -al /" #| "grep etc" !
```

3) 将 shell 的执行结果重定向到文件

```
import sys.process._
"ls -al /" #| "grep etc" !;
"ls -al /" #>> new File("output.txt") !;
```

注：注意，每一个感叹号后边，有分号结束

scala 进程还可以提供：

p #&& q 操作，即 p 任务执行成功后，则执行 q 任务。

p #|| q 操作，即 p 任务执行不成功，则执行 q 任务。

既然这么强大，那么 crontab + scala + shell，就完全不需要使用 oozie 了。

3.7 正则表达式

我们可以通过正则表达式匹配一个句子中所有符合匹配的内容，并输出：

```
import scala.util.matching.Regex
val pattern1 = new Regex("(S|s)cala")
```

```
val pattern2 = "(S|s)cala".r
val str = "Scala is scalable and cool"
println((pattern2 findAllIn str).mkString(","))
```

四　高级类型

4.1　类型与类的区别

在 Java 里，一直到 jdk1.5 之前，我们说一个对象的类型(type)，都与它的 class 是一一映射的，通过获取它们的 class 对象，比如 String.class, int.class, obj.getClass() 等，就可以判断它们的类型(type)是不是一致的。

而到了 jdk1.5 之后，因为引入了泛型的概念，类型系统变得复杂了，并且因为 jvm 选择了在运行时采用类型擦拭的做法(兼容性考虑)，类型已经不能单纯的用 class 来区分了，比如 List<String> 和 List<Integer> 的 class 都是 Class<List>，然而两者类型(type)却是不同的。泛型类型的信息要通过反射的技巧来获取，同时 java 里增加了 Type 接口来表达更泛的类型，这样对于 List<String>这样由类型构造器和类型参数组成的类型，可以通过 Type 来描述；它和 List<Integer> 类型的对应的 Type 对象是完全不同的。

在 Scala 里，类型系统又比 java 复杂很多，泛型从一开始就存在，还支持高阶的概念(后续会讲述)。所以它没有直接用 Java 里的 Type 接口，而是自己提供了一个 scala.reflect.runtime.universe.Type(2.10 后)

在 scala 里获取类型信息是比较便捷的：

```
class A{}
object TypeSyllabus {
  def main(args: Array[String]): Unit = {
    import scala.reflect.runtime.universe._
    println(typeOf[A])
  }
}
```

同样 scala 里获取类(Class)信息也很便捷，类似：

```
class A{}
object TypeSyllabus {
  def main(args: Array[String]): Unit = {
    import scala.reflect.runtime.universe._
    println(typeOf[A])
    println(classOf[A])
  }
}
```

注：注意，typeOf 和 classOf 方法接收的都是类型符号(symbol)，并不是对象实例。

4.2 classOf 与 getClass 的区别

获取 Class 时的两个方法：classOf 和 getClass

```
scala> class  A
scala> val a = new A
scala> a.getClass
res2: Class[_ <: A] = class A
scala> classOf[A]
res3: Class[A] = class A
```

上面显示了两者的不同，getClass 方法得到的是 Class[A]的某个子类，而 classOf[A] 得到是正确的 Class[A]，但是去比较的话，这两个类型是 equals 为 true 的。

这种细微的差别，体现在类型赋值时，因为 java 里的 Class[T]是不支持协变的，所以无法把一个 Class[_ <: A] 赋值给一个 Class[A]。

4.3 单例类型

4.4 类型投影

在 scala 里，内部类型(排除定义在 object 内部的)，想要表达所有的外部类 A 实例路径下的 B 类型，即对 a1.B 和 a2.B 及所有的 an.B 类型找一个共同的父类型，这就是类型投影，用 A#B 的形式表示。

A#B

/ \

/ \

a1.B a2.B

我们回头来对比一下 scala 里的类型投影与 java 里的内部类型的概念，java 里的内部类型在写法上是 Outter.Inner 它其实等同于 scala 里的投影类型 Outter#Inner，java 里没有路径依赖类型的概念，比较简化。

4.5 类型别名

可以通过 type 关键字来创建一个简单的别名，类型别名必须被嵌套在类或者对象中，不能出现在 scala 文件的顶层：

```
class Document {
  import scala.collection.mutable._
  type Index = HashMap[String, (Int, Int)]
def play(x: Index): Unit ={}
}
```

4.6 结构类型

结构类型是指一组关于抽象方法、字段和类型的规格说明，你可以对任何具备 play 方法的类的实例调用 play 方法，这种方式比定义特质更加灵活，是通过反射进行调用的：

```
class Structure {
  def play() = println("play 方法调用了")
}

object HelloStructure {
  def main(args: Array[String]): Unit = {
    type X = {def play(): Unit} //type 关键字是把 = 后面的内容命名为别名。
```

```
    def init(res: X) = res.play //本地方法

    init(new {
      def play() = println("Played")
    })

    init(new {
      def play() = println("Play 再一次")
    })

    object A {
      def play() {
        println("A object play")
      }
    }

    init(A)
    val structure = new Structure
    init(structure)
  }
}
```

总结：

结构类型，简单来说，就是只要是传入的类型，符合之前定义的结构的，都可以调用。

4.7 复合类型

class A extends B with C with D with E

应做类似如下形式解读：

class A extends (B with C with D with E)

T1 with T2 with T3 …

这种形式的类型称为复合类型(compound type)或者也叫交集类型(intersection type)。

也可以通过 type 的方式声明符合类型：

```
type X = X1 with X2
```

4.8 中置类型

中置类型是一个带有两个类型参数的类型，以中置语法表示，比如可以将 Map[String, Int] 表示为：

```
val scores: String Map Int = Map("Fred" -> 42)
```

4.9 自身类型

self => 这句相当于给 this 起了一个别名为 self：

```
class A {
  self =>  //this 别名
  val x=2
  def foo = self.x + this.x
}
```

self 不是关键字，可以用除了 this 外的任何名字命名(除关键字)。就上面的代码，在 A 内部，可以用 this 指代当前对象，也可以用 self 指代，两者是等价的。

它的一个场景是用在有内部类的情况下：

```
class Outer { outer =>
  val v1 = "here"
  class Inner {
    val v1 = "there"
      println(outer.v1) // 用 outer 表示外部类，相当于 Outer.this
  }
}
```

对于 this 别名 self =>这种写法形式，是自身类型(self type)的一种特殊方式。self 在不声明类型的情况下，只是 this 的别名，所以不允许用 this 做 this 的别名。

4.10 运行时反射

scala 编译器会将 scala 代码编译成 JVM 字节码，编译过程中会擦除 scala 特有的一些类型信息，在 scala-2.10 以前，只能在 scala 中利用 java 的反射机制，但是通过 java 反射机制得到的是只是擦除后的类型信息，并不包括 scala 的一些特定类型信息。从 scala-2.10 起，scala 实现了自己的反射机制，我们可以通过 scala 的反射机制得到 scala 的类型信息。

给定类型或者对象实例，通过 scala 运行时反射，可以做到：1）获取运行时类型信息；2）通过类型信息实例化新对象；3）访问或调用对象的方法和属性等。

4.10.1 获取运行时类型信息

scala 运行时类型信息是保存在 TypeTag 对象中，编译器在编译过程中将类型信息保存到 TypeTag 中，并将其携带到运行期。我们可以通过 typeTag 方法获取 TypeTag 类型信息。

```
import scala.reflect.runtime.universe._
val typeTagList = typeTag[List[Int]]//得到了包装 Type 对象的 TypeTag 对象
println(typeTagList)
或者使用：
typeOf[List[Int]]//直接得到了 Type 对象
```

注：Type 对象是没有被类型擦除的

我们可以通过 typeTag 得到里面的 type，再通过 type 得到里面封装的各种内容：

```
import scala.reflect.runtime.universe._
val typeTagList = typeTag[List[Int]]
println(typeTagList)
println(typeTagList.tpe)
println(typeTagList.tpe.decls.take(10))
```

4.10.2 运行时类型实例化

我们已经知道通过 Type 对象可以获取未擦除的详尽的类型信息，下面我们通过 Type 对象中

的信息找到构造方法并实例化类型的一个对象：

```
class Person(name:String, age: Int) {
  def myPrint() = {
    println(name + "," + age)
  }
}

object PersonMain extends App{
  override def main(args: Array[String]): Unit = {
    //得到 JavaUniverse 用于反射
    val ru = scala.reflect.runtime.universe
    //得到一个 JavaMirror，一会用于反射 Person.class
    val mirror = ru.runtimeMirror(getClass.getClassLoader)
    //得到 Person 类的 Type 对象后，得到 type 的特征值并转为 ClassSymbol 对象
    val classPerson = ru.typeOf[Person].typeSymbol.asClass
    //得到 classMirror 对象
    val classMirror = mirror.reflectClass(classPerson)
    //得到构造器 Method
    val constructor = ru.typeOf[Person].decl(ru.termNames.CONSTRUCTOR).asMethod
    //得到 MethodMirror
    val methodMirror = classMirror.reflectConstructor(constructor)
    //实例化该对象
    val p = methodMirror("Mike", 1)
    println(p)
  }
}
```

4.10.3 运行时类成员的访问

```
class Person(name:String, age: Int) {
    def myPrint() = {
        println(name + "," + age)
    }
}

object PersonMain extends App{
    override def main(args: Array[String]): Unit = {
        //获取 Environment 和 universe
        val ru = scala.reflect.runtime.universe
        //获取对应的 Mirrors,这里是运行时的
        val mirror = ru.runtimeMirror(getClass.getClassLoader)
        //得到 Person 类的 Type 对象后，得到 type 的特征值并转为 ClassSymbol 对象
        val classPerson = ru.typeOf[Person].typeSymbol.asClass
        //用 Mirrors 去 reflect 对应的类,返回一个 Mirrors 的实例,而该 Mirrors 装载着对应类的信
息
        val classMirror = mirror.reflectClass(classPerson)
        //得到构造器 Method
        val constructor = ru.typeOf[Person].decl(ru.termNames.CONSTRUCTOR).asMethod
        //得到 MethodMirror
        val methodMirror = classMirror.reflectConstructor(constructor)
        //实例化该对象
        val p = methodMirror("Mike", 1)
        println(p)



        //反射方法并调用
```

```
        val instanceMirror = mirror.reflect(p)
        //得到 Method 的 Mirror
        val myPrintMethod = ru.typeOf[Person].decl(ru.TermName("myPrint")).asMethod
        //通过 Method 的 Mirror 索取方法
        val myPrint = instanceMirror.reflectMethod(myPrintMethod)
        //运行 myPrint 方法
        myPrint()

        //得到属性 Field 的 Mirror
        val nameField = ru.typeOf[Person].decl(ru.TermName("name")).asTerm
        val name = instanceMirror.reflectField(nameField)
        println(name.get)

    }
}
```