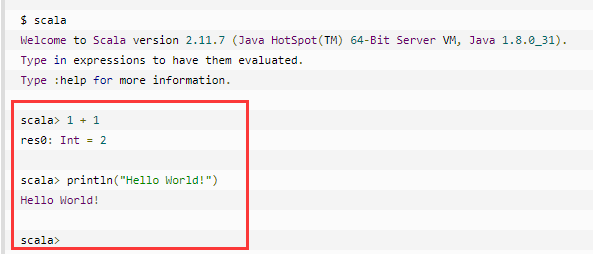
# 1 Scala

## 编程方式

### 交互式编程（Scala解释器）

（注意：虽然Scala类似于Python一样有自己的解释器，但Scala并不是解释型语言，只不过Scala解释器会马上将输入的Scala语言立即编译成二进制字节码，然后运行）

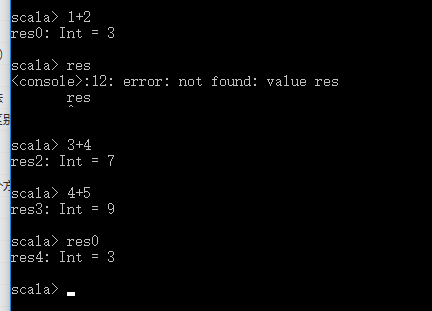
图1



#### res 默认表达式值的变量名

如果在解释器中没有显式指定Scala的表达式的值的变量名，默认的变量名就是res。

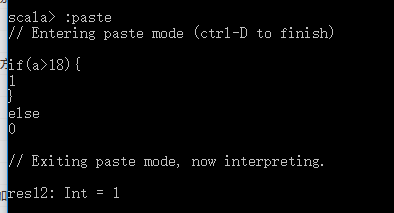
图1



#### :paste 和 ctrl+D

:paste和ctrl+D可以在解释器中输入多行语句。

图1



### 脚本编程

编写\*.scala脚本，然后scalac编译成\*.class，然后scala \*.class运行。

图1



## 1.2 变量

### 1.2.1 变量声明

在Scala中，用关键字var声明变量，用关键字val声明常量。

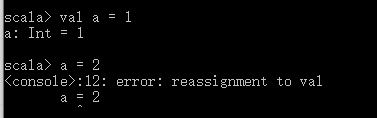
（注意：val声明的常量是不能被程序修改的，否则会编译时报错）

（注意：val类似于Java的常量，一旦声明它的值就不可以再改变）

（注意：var有点像JavaScript中的变量，是一般的变量，声明值后仍可以随时改变它的值）

（注意：在Spark中建议使用val常量，因为Spark中变量会经常在分布式系统（网络上）传输，val可以保证我们不需要担心这个变量的值在网络传输中被改变）

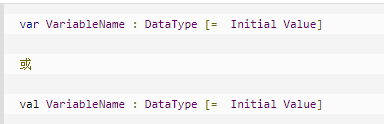
图1



### 1.2.2 变量类型声明

变量的类型在变量名之后，等于号之前声明。

图1

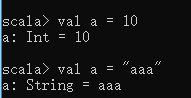


#### 1.2.2.1 类型推测

变量的类型不一定要声明，如果不声明，Scala会根据变量或常量的初始值推测出来。

（注意：如果既不声明变量类型，又没有初始值，则会编译报错）

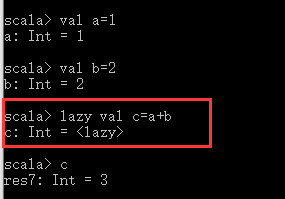
图1



## 1.2.3 lazy

用lazy修饰变量时，只有当变量第一次被使用到它的值时，才会去求值。

图1



## 1.3数据类型

原文链接：<https://www.cnblogs.com/qingyunzong/p/8858465.html>

Scala支持的数据类型如下：

图1

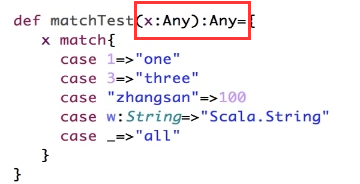


Unit：和其他语言中void等同，表示无值。

Nothing：是任何其他类型的子类型。

Any：是任何其他类的超类。

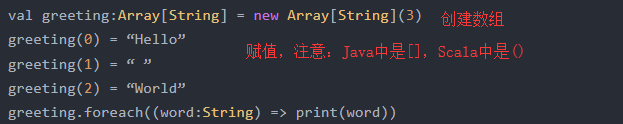
图1



### 1.3.1 数组（Array）

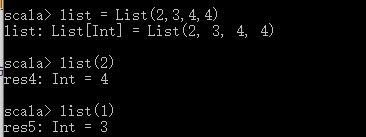
（注意：Array一旦创建后，长度就不可变）

图1



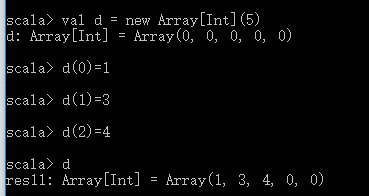
注意：List是有序的，且元素可以重复。

图2



（注意：Array一旦创建后，所有元素都有默认值）

图3



（注意：当创建时，每个元素的类型不一样时，Array的泛型是这些元素的公共父类型）

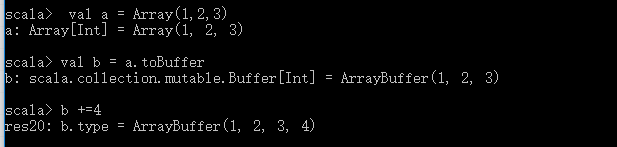
图4



#### 1.3.1.1 Array和Buffer的转换

toBuffer从Array转Buffer；toArray从Buffer转Array。

图1



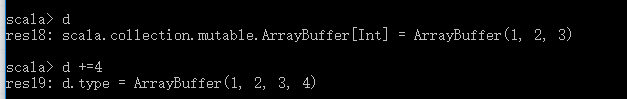
#### 1.3.1.2 ArrayBuffer

ArrayBuffer类似于Java的ArrayList，是长度可变的数组。

图1



图2（ArrayBuffer使用+=向里面添加元素）



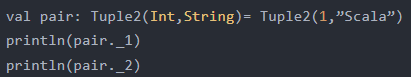
### 1.3.2 元组（Tuple）

元组可以包含不同类型的元素，然后用“.\_基于1的索引”访问元组的元素。

（注意：元组最多支持22个元素，超过22个元素会编译时报错）

（注意：Tuple的\_n访问是以1开始的，第一个元素的下标是1）

图1

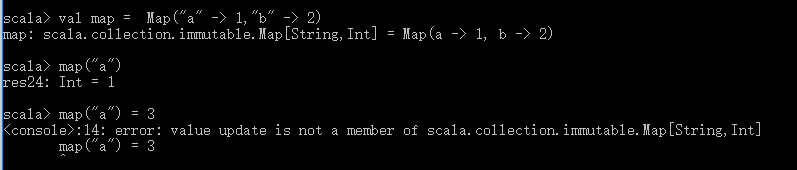


### 1.3.3 映射（map）

映射就是K/V的结构，每一个元素都是一个键值对。

（注意：这种immutable的Map是不可以修改的）

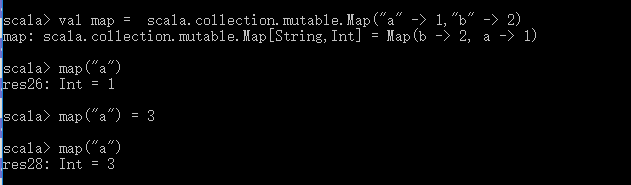
图1



#### 1.3.3.1 可变的map

mutable包里面的Map是可变的。

图1



### 1.3.4 Set

Set是没有重复的，无序的，不可变的元素的集合。

图1



#### 1.3.4.1 LinkedHashSet

LinkedHashSet是可变的，有序的。

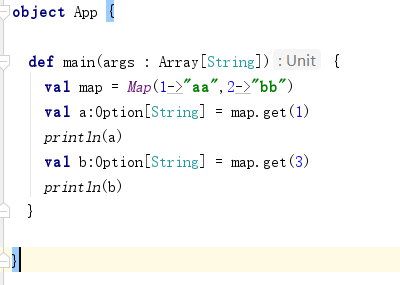
图1

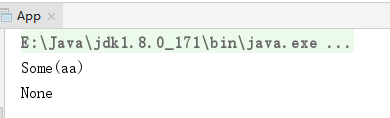


### 1.3.5 选项（Option）

Option（选项）表示一个值是可选的，可有可无值的。如果值存在，则Option[T]是一个Some(变量值)，如果值不存在，则Option[T]是对象None。

图1

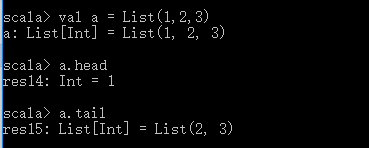




### 1.3.7 List

List是不可变的，head表示第一个元素，tail是一个List，表示除第一个元素以外的其他元素。

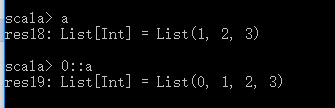
图1



#### 1.3.7.1 ::

a::b，a是一个head（一个元素），b是一个tail（一个List），合并在一起。

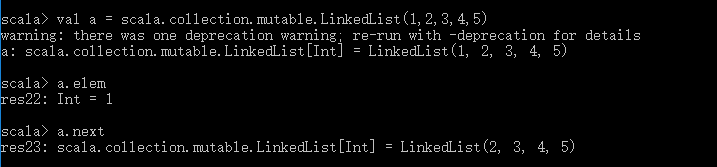
图1



#### 1.3.7.2 LinkedList

LinkedList是不可变的，elem表示第一个元素，next是一个LinkedList，表示除第一个元素以外的其他元素。

图1



## 1.4 方法和函数

Scala的方法是组成类的一部分，必须依赖类或类的对象来使用。

函数是在外面独立存在的，不需要依赖类或类的对象来使用。

### 1.4.1 无返回值的方法

如果方法无返回值，return type为Unit，相当于Java的void。

（注意：Scala的函数会返回函数最后一行语句的值，不需要return，Scala会自动猜测返回值的类型）

（注意：如果函数包含递归，必须显式指定函数的返回类型，因为对于递归Scala猜测返回值类型会失败）

图1

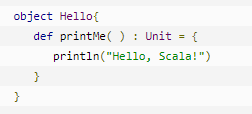


图2

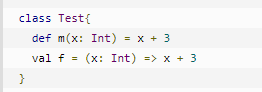


### 1.4.2 函数和方法的区别

方法是组成类的一部分。而函数是一个完整的对象，函数实际上是继承了Trait类的对象。

方法用def定义，函数用val定义。

图1



### 1.4.3 方法的简写

原文链接：<https://blog.csdn.net/u010916338/article/details/80454625>

图1（完整方法声明）

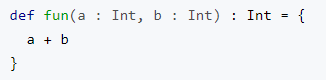


图2（没有参数，()可以省略）

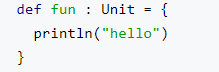


图3（没有返回值，Unit也可以省略）

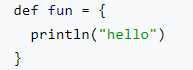


图4（方法体只有一行时，{}也可以省略）



### 1.4.4 =>

=>是创建匿名函数的语法糖。

A,B是参数的类型，T是返回值的类型。

图1（=>创建匿名函数的完整版）



图2（参数为空）



图3（返回值为空）



图4（\_表示参数，=>的左边可以整个省略）

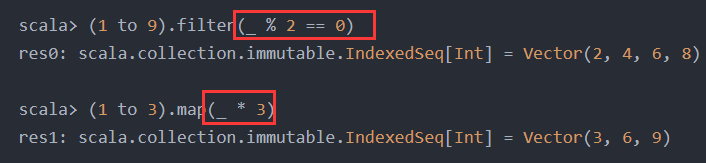
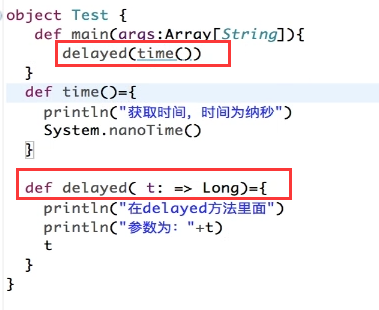


图5（=>左边是参数，右边是方法体）



### 1.4.5 方法作为另一个方法的参数

图1

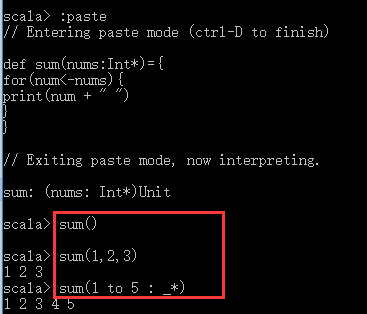


### 1.4.6 方法参数不定

（注意：参数数量不定时，0个也是允许的）

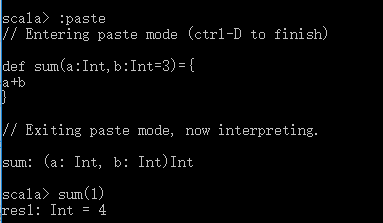
（注意：sum(1 to 5)会编译错误，因为1 to 5 是一个Range序列类型，而sum的参数是Int）

图1



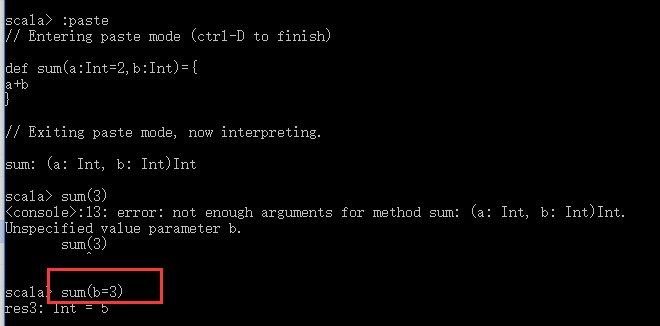
### 1.4.7 默认参数值

图1



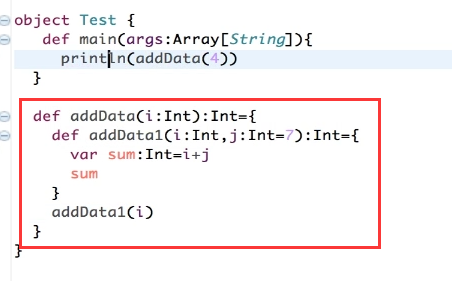
（注意：如果有默认值的参数在前面，则需要手动指定参数的名称）

图2



### 1.4.8 嵌套函数

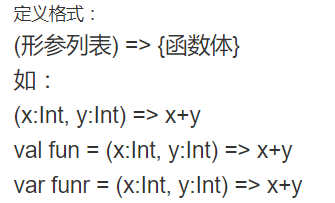
图1



### 1.4.9 匿名函数

匿名函数没有函数名，左边是形参列表（包括形参名字和形参类型），右边是函数体。

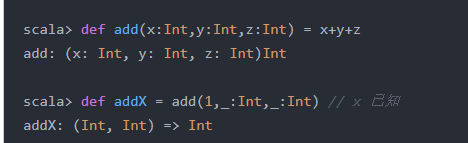
图1



### 1.4.10 偏应用函数

偏应用函数是指：首先有一个参数个数为n的函数add，而我们为它提供了少于n个参数，这个新函数addX称为偏应用函数。

图1



### 1.4.11 函数调用时不加参数

当调用函数时，如果不传入参数，可以省略()。

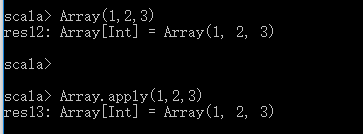
图1



### 1.4.12 apply函数

在Scala中的object中，可以声明apply函数。类名.apply()可以简写成类名()，在Scala中通常用类名()来调用apply()函数来创建类的实例对象，而不使用new 类名()的方式。

图1（使用类名()来创建Array类的实例对象）



### 1.4.13 函数能独立存在

在Java中，函数是不能独立存在的，必须被包含在class中；但在Scala中，函数是可以独立存在的。

图1

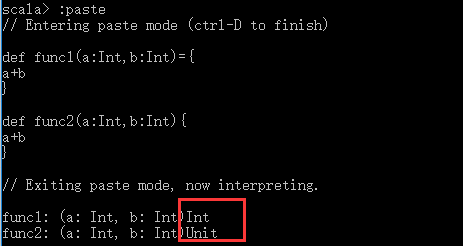


### 1.4.14 函数{}前有=和无=

{}前有=是有返回值的，无=是无返回值的。

Unit返回值的函数也称为过程。

图1



## 1.5 类和对象

类是抽象的，不占用内存；对象是具体的，占用内存。对象是类的具体实例。

图1（Scala中定义类）

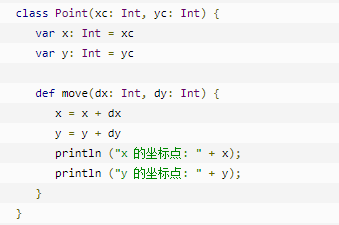
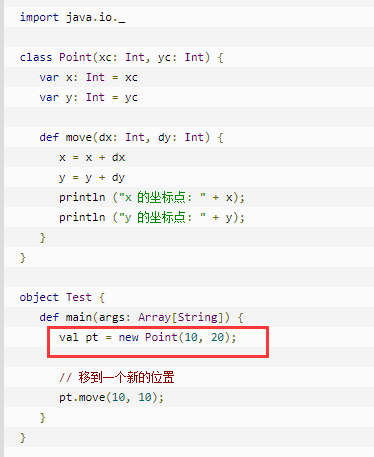


图2（Scala中定义对象）



### 1.5.1 object 单例对象

Scala中没有static，所以类的单例模式（只有一个实例的类）就是单例对象，相当于Java的静态类。通常用来放一些静态的field或method。

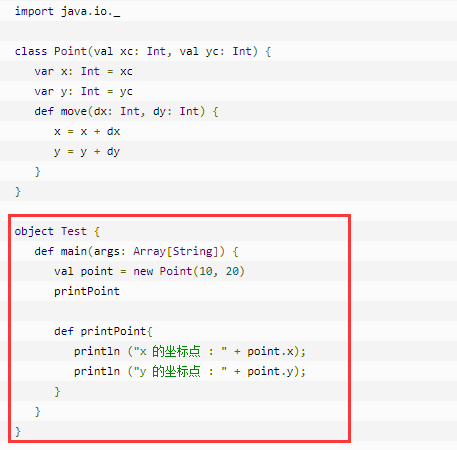
（注意：object也有构造方法，第一次调用object的方法时，就会调用object的constructor，以后调用object的方法时，不会再重复调用object的constructor）

（注意：object不能定义带参数的constructor）

（注意：object的constructor就是所有不在method里面的代码）

（注意：object的存在也是因为在Scala的class中没法定义static的成员）

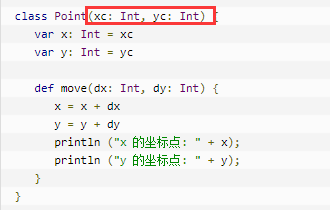
图1



### 1.5.2 类名后直接加参数

类似于Java的构造函数的形参，只不过可以写在类名后面。

图1



### 1.5.3 伴生类和伴生对象

当class和object共享一个名称时，class A是伴生类，object A是伴生对象，伴生对象和伴生类可以互相访问对方的私有成员。

（注意：伴生类和伴生对象必须放在同一个.scala文件中）

（注意：正常来说，伴生对象的私有成员是无法在外部访问的，但伴生类可以做到，所以是一大特色）

图1



#### 1.5.3.1 伴生对象的apply方法

Class()其实就是调用了这个类的伴生对象的apply方法，所以一般在伴生对象的apply方法中new Class的实例。

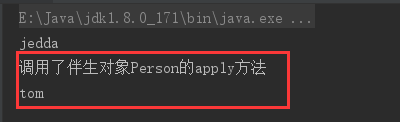
（注意：伴生对象的返回值默认就是该Class的实例对象）

图1



图2（实际上是调用了伴生对象的apply方法）





### 1.5.4 特征（Trait）

特征 类似于Java的接口，但它可以定义属性和方法的实现。

（在Scala中，不支持多继承class，但支持多继承Trait特征）

（Trait的方法可以有具体的实现）

（注意：类实现Trait的方法时，不需要加override关键字）

图1

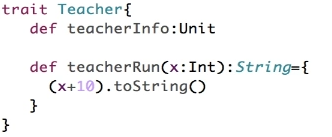
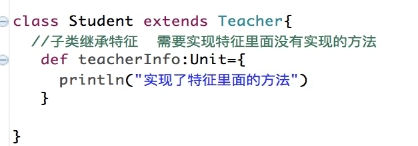


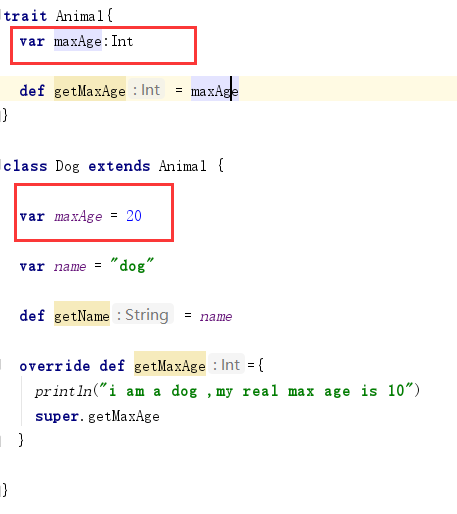
图2（实现了特征里面的方法）



#### 1.5.4.1 抽象field

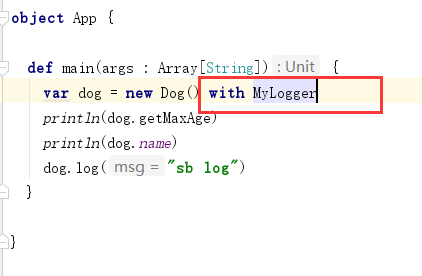
在trait中，可以定义抽象field，抽象field就是没有初始值的field。但子类继承该trait时，必须进行初始化，否则子类必须定义为抽象的。

图1



#### 1.5.4.2 创建类时动态混入trait

图1



#### 1.5.4.3 trait调用链

当一个类继承了多个trait后，可以从右往左依次执行这些trait的同一个方法，只要这些方法的最后调用super.xxx(xxx)即可，表示调用上一个trait的相同方法。

（注意：这些tait要继承同一个类和trait）

图1

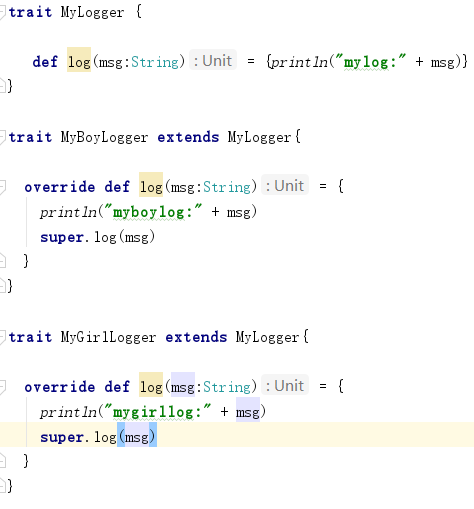
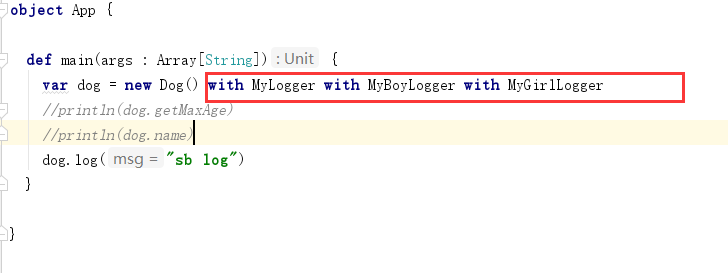
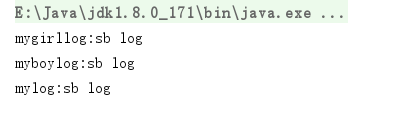


图2（从右往左依次调用）





#### 1.5.4.4 trait的构造方法

不在trait的方法里的代码就是trait的构造方法。

（注意：如果trait B继承了trait A，则构造trait B之前会先构造trait A）

（注意：如果一个类继承了traitB，trait C，则会从左到右去构造，先构造trait B，再构造trait C）

（注意：如果一个类继承了trait B，trait B只会构造一次）

（注意：trait是没有接收参数的构造方法的）

图1

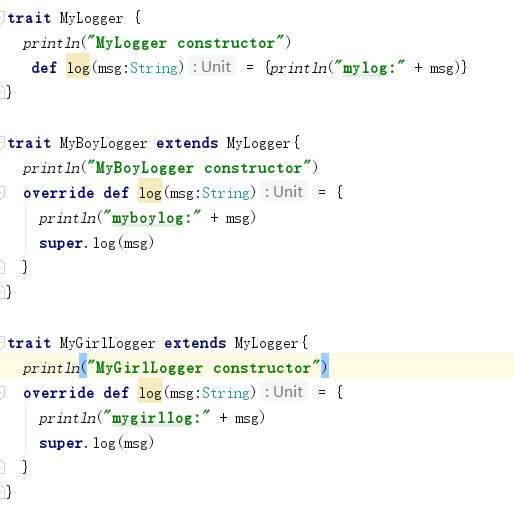
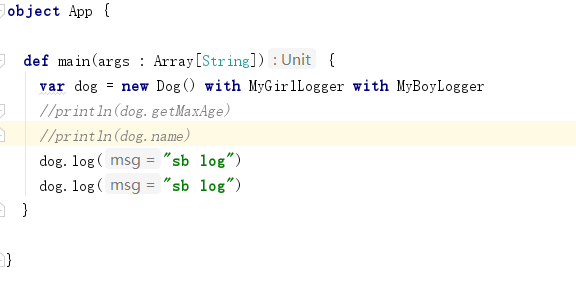
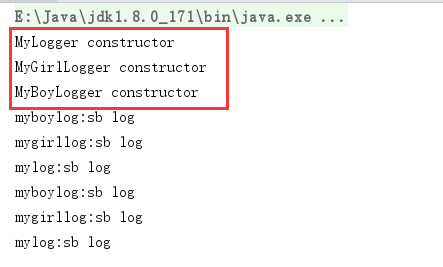


图2





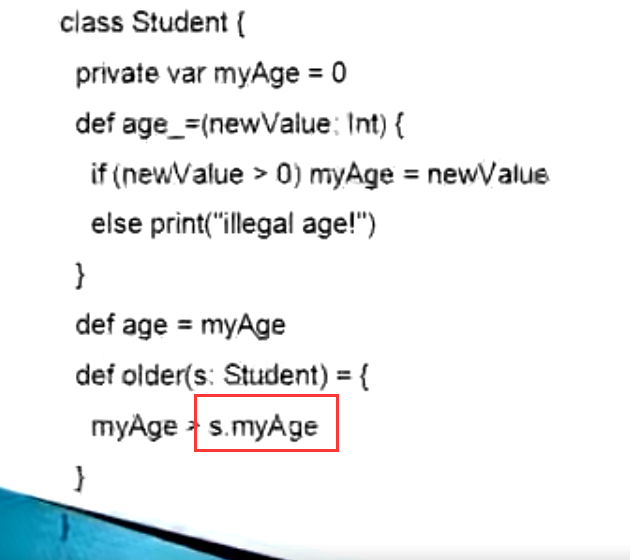
### 1.5.5 private修饰类成员

如果加了private修饰，则Scala自动生成的get，set方法都是private的。

（注意：如果希望不生成get，set方法，则使用private(this)修饰）

（注意：private只是类私有，对象student1是可以通过student2来访问对象student2的name和name\_方法的；但是private(this)是对象私有，压根不生成get，set方法，所以对象student1也无法访问对象student2的name和name\_方法）

图1（private是类私有，可以通过s来访问s的myAge的get方法）



### 1.5.6 var和val的类成员

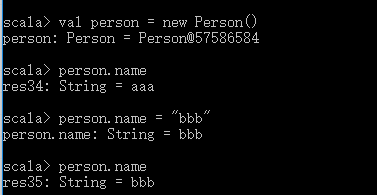
var的类成员，Scala会自动为它创建public的get和set方法。

val的类成员，Scala会自动为它创建public的get方法，不会生成set方法。

### 1.5.7 get和set方法

在Scala中，name成员的get和set方法不像Java那样叫getName和setName。get和set方法分别叫name和name\_= ，通过对象.name来调用name方法，通过对象.name=来调用name\_方法。

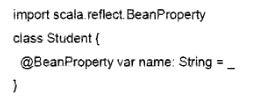
图1



### 1.5.8 @BeanProperty 修饰类成员

@BeanProperty除了会自动为类成员name生成name和name\_方法，还会生成Java风格的getName和setName方法，共4个方法。

图1

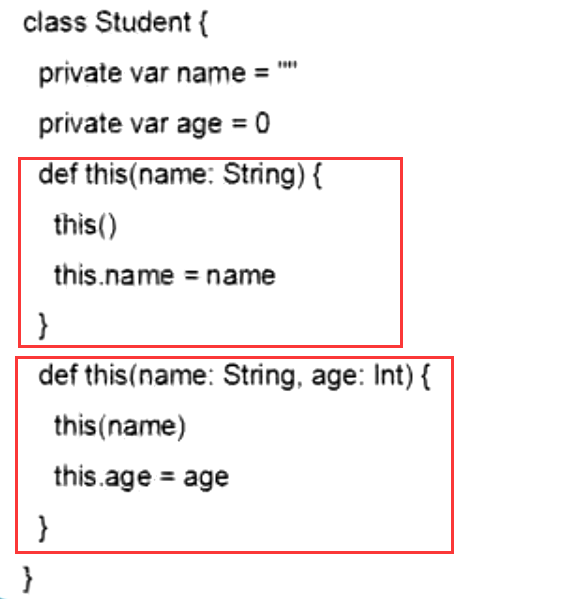


### 1.5.9 主构造方法和辅助构造方法

辅助构造函数之间可以相互调用，而且第一行必须调用主构造函数。

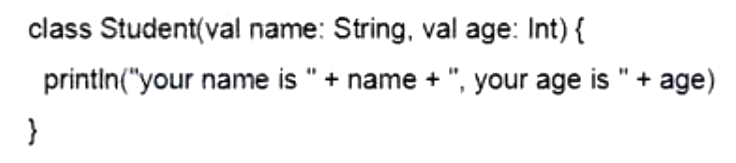
注意：（构造方法是不需要返回值的）

图1



（注意：在Scala中，主构造方法是和类名放在一起的）

图2



### 1.5.10 case class 样式类

case class是Scala中一种特殊的类，类似于Java中的Java Bean，但只提供field，没有method。并由Scala编译时自动提供getter和setter方法。

（注意：case class的field会当作主构造函数的参数）

（注意：case class的主构造函数的参数类型默认是val）

（注意：Scala自动为case class定义了伴生对象，并且在该伴生对象中定义了apply()方法，apply方法接收主构造函数中相同的参数）

图1

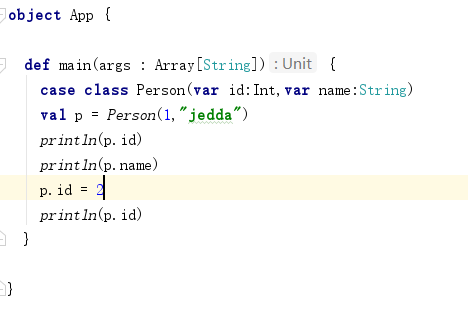
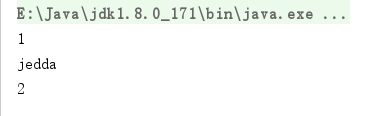


图2

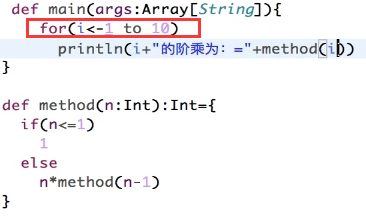


## 1.6 运算符

### 1.6.1 <-

左箭头<-是赋值的意思。

图1



## 1.7 Scala和Java的关系

Scala底层也是基于Java虚拟机（JVM）去运行的。

（注意：其实就是Scala源文件会编译成二进制字节码，然后将二进制字节码交给JVM去运行）

（注意：也因为Scala和Java都是基于JVM去运行的，所以Scala代码和Java代码可以相互调用）

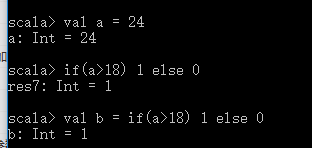
## 1.8 条件控制

### 1.8.1 if

#### 1.8.1.1 if表达式是有值的

在Scala的if语句中，if表达式是有值的，它的值就是if或else的最后一行语句返回的值。

图1



（注意：当if和else的最后一行语句的返回值的类型不一样时，会取2者的公共父类型）

图2



（注意：如果if表达式中没有else语句，则else语句默认的返回值是Unit类型，Unit类型只有一个实际值是()，相当于Java的void）

图3

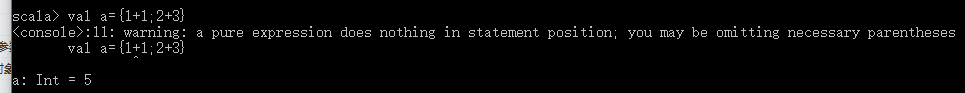
（Unit和Int的公共父类型是Any，可以类型是Any，但值是Unit类型唯一的实际值()）



### 1.8.2 {}块表达式

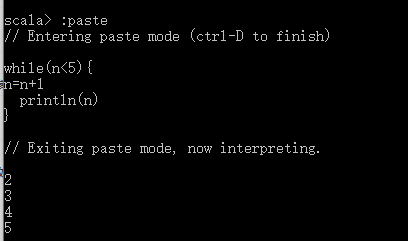
快表达式的值就是{}最后一个语句的值。

图1



### 1.8.3 while

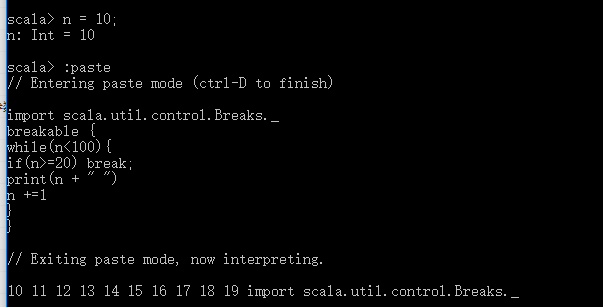
图1



### 1.8.4 跳出循环

在Scala中，没有像Java一样的break语句，但可以用Breaks的beak函数来实现。

图1

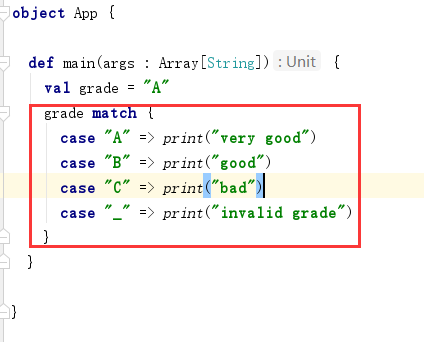


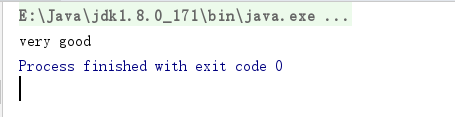
### 1.8.5 match case 模式匹配

Java里面的switch case只能匹配变量的值，而Scala的match case可以匹配变量的各种情况。

（注意：在match case中，只要其中一个case分支满足了，就不会再判断下一个case分支了，这和Java不同；但只要不满足，就会判断下一个case分支，如果需要退出要使用break）

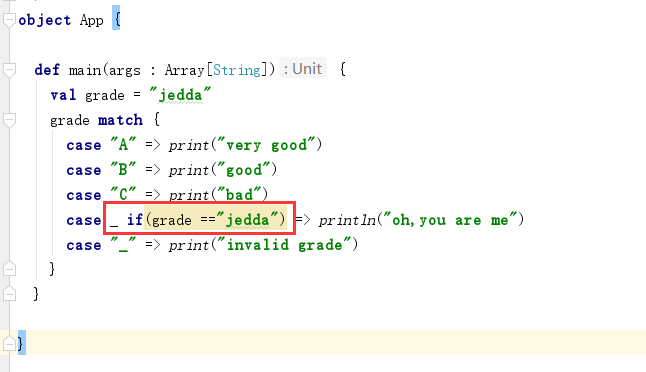
图1





#### 1.8.5.1 case分支中使用if条件判断

图1

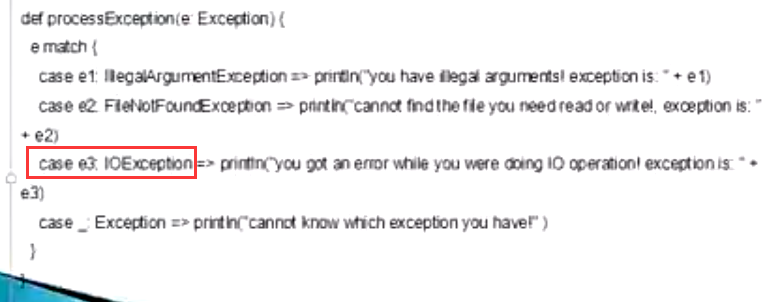




#### 1.8.5.2 case分支中判断变量类型

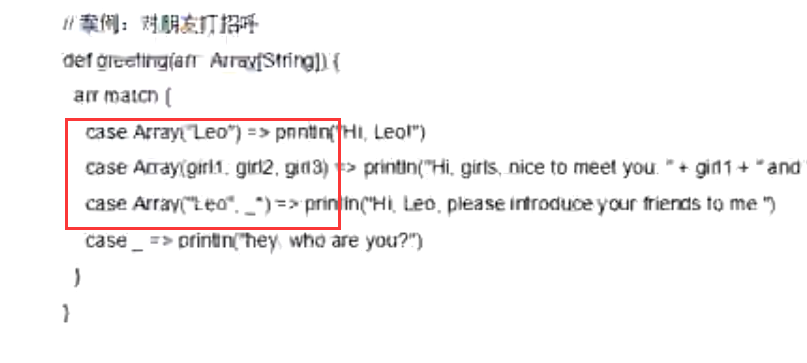
注意：判断变量类型时，case后面跟的是变量名，而不是变量值。

图1



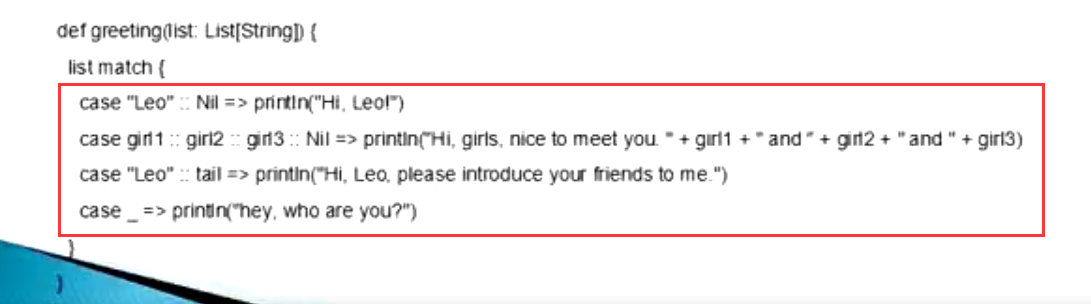
#### 1.8.5.3 case分支中对Array进行操作

图1



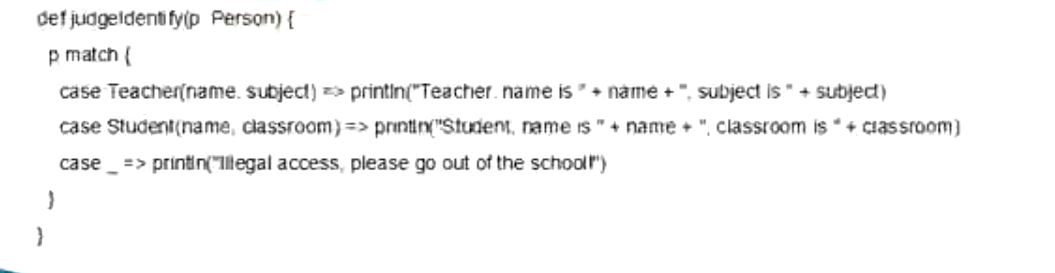
#### 1.8.5.4 case分支中对List进行操作

图1



#### 1.8.5.5 case分支中判断对象所属class

图1

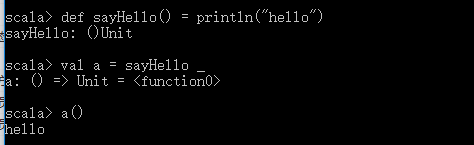


## 1.9 函数式编程

### 1.9.1 函数赋值给一个变量，用变量来调用函数

要想用变量来调用函数，赋值时必须加\_ 。

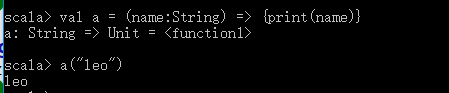
图1



### 1.9.2 将匿名函数赋值给一个变量

匿名函数用(xxx) => {xxx}的格式定义，然后赋值给一个变量。

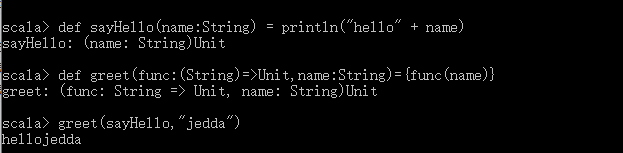
图1



### 1.9.3 高阶函数：将函数作为入参传给另一个函数

高阶函数的入参是函数的位置要用(xxx)=>xxx指定函数的类型。

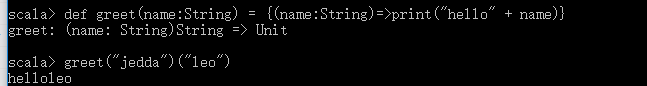
图1



### 1.9.4 高阶函数：将函数作为另一个函数的返回值

返回值是函数的位置要用(xxx)=>xxx指定函数的类型。

图1



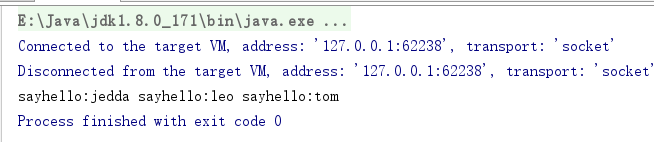
### 1.9.5 \_

如果该函数只有1个函数，则调用该函数时可以用\_替代。

图1



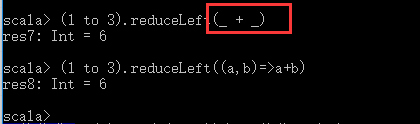
图2



如果用\_声明函数，可以只写函数的函数体。

（注意：其实这里的redeuceLeft就是一个高阶函数，它接收另一个函数作为参数，需要指定函数的类型。但用\_来声明函数时，可以只写函数的函数体）

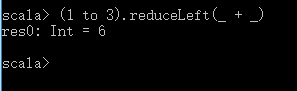
图3



### 1.9.6 \_ \_

2个下划线时，从集合中取出2个元素，执行func。然后再和集合中取出一个元素，和该结果执行func。

图1



### 1.9.7 currying函数

currying函数将一个2参数的函数变成2个1参数的函数，第一个函数取原函数的第一个参数，第二个函数取原函数的第二个参数。

图1

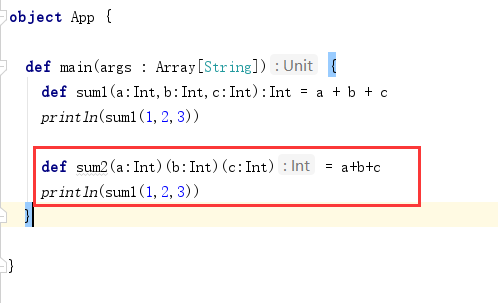
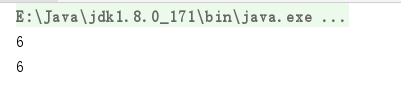


图2



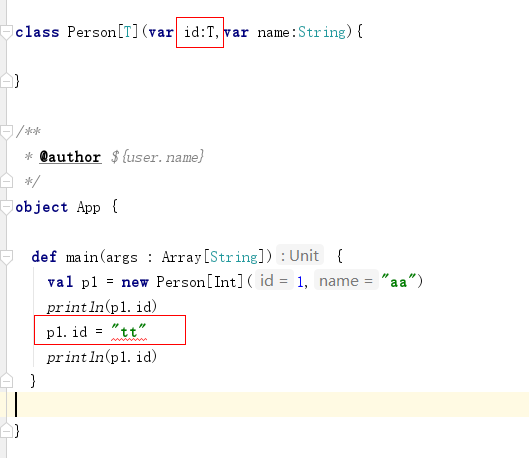
## 1.10 类型参数（泛型）

### 1.10.1 泛型类

在类的声明中，定义泛型，这样在类内部的field和method就可以使用这些泛型。

（注意：优点是对field和method进行的类型限制，防止field和method被传入不期望的类型）

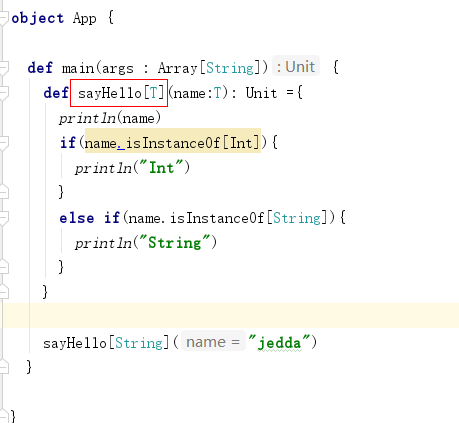
图1

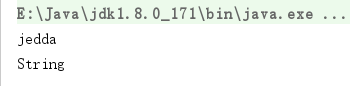


### 1.10.2 泛型函数

在声明函数时，定义泛型，这样在函数内部，比如函数的参数，函数的局部变量，函数的返回值等就可以做泛型限制。

图1





### 1.10.3 上边界

泛型如果没有边界，则泛型可以是任意类型，没有任何限制。上边界是限制了泛型必须是某个类的子类，从而让程序能正常运行，因为不受限制的类中有些类是没有程序里面定义的方法的。

图1（没有上边界的情况）

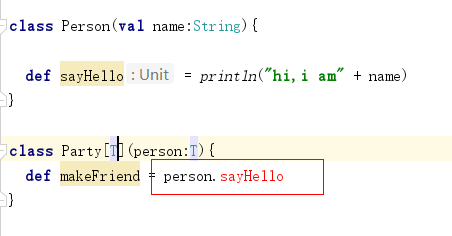
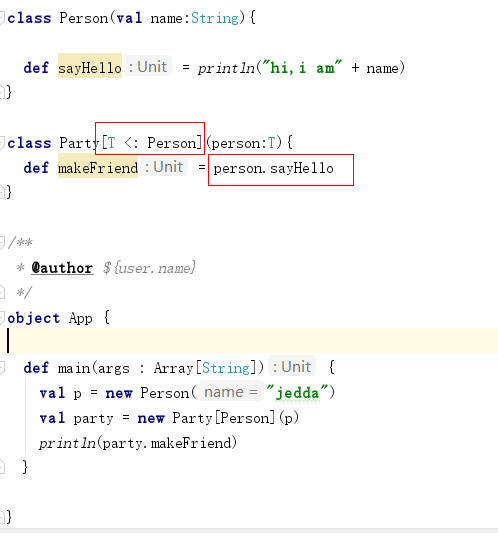


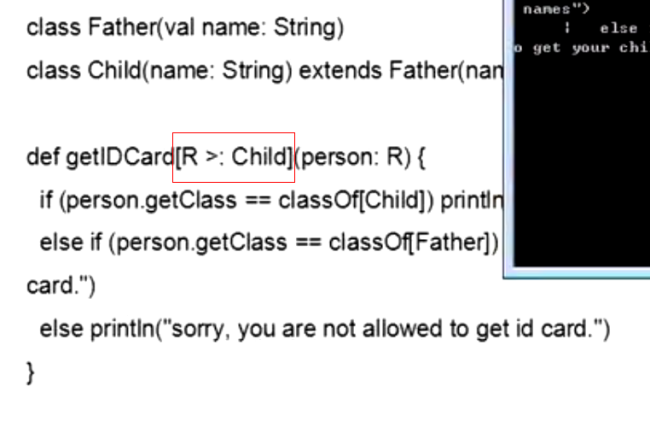
图2（有上边界）



### 1.10.4 下边界

下边界限制了泛型必须是某个类的父类。

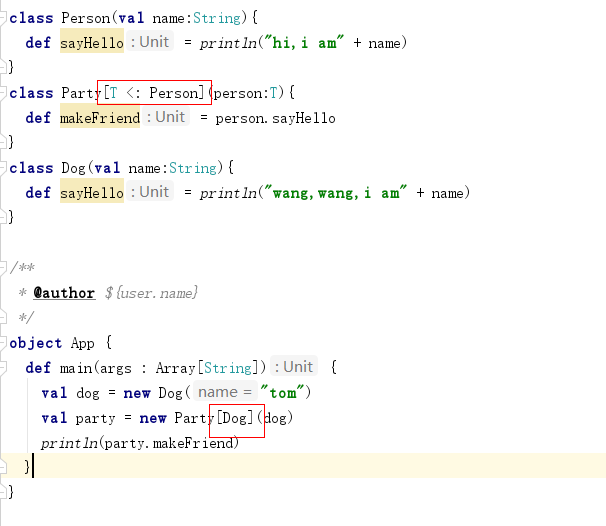
图1



### 1.10.5 View Bounds

上下边界Bounds会定义一个泛型的范围，在这个范围以外的类型无法使用。但View Bounds可以打破这个限制，配合implicit的隐式类型转换，可以使上下边界以外的类型能够使用。

图1（未使用View Bounds）



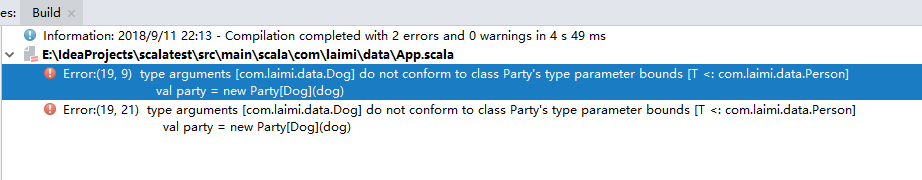
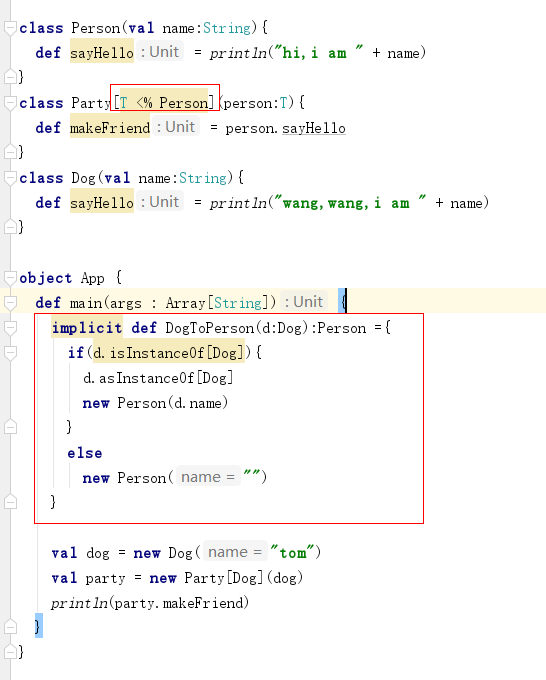
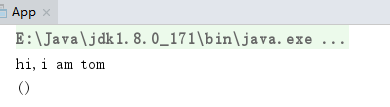


图2

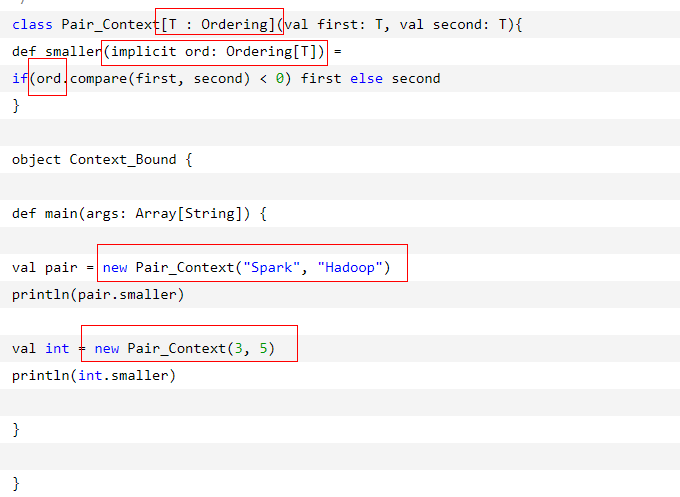




### 1.10.6 Context Bound

泛型需要以“T:泛型”的格式声明，然后类中必须有一个隐式值，且这个隐式值的类型必须是“泛型[T]”。然后这个隐式值就可以在函数内部使用。

图1



### 1.10.7 协变和逆变

在Java中，如果B是A的子类，但Card[B]并不是Card[A]的子类，这里的Card是一个普通的类。

协变：+T。比如：Card[+T]，则Card[T的子类]都可以匹配。

逆变：-T。比如：Card[-T]，则Card[T的父类]都可以匹配。

## 1.11 implicit关键字

### 1.11.1 隐式类型转换

当编译器看到类型X却需要类型Y，则会在当前作用域找，看是否定义了从类型X到类型Y的隐式转换函数。

（注意：通常建议隐式转换函数的名字是：one2one）

（注意：隐式转换函数需要用implicit关键字修饰）

（注意：Scala会去3个地方寻找隐式转换函数，1：伴生类或伴生对象中找 2.从当前作用域找 3.import中找）

图1

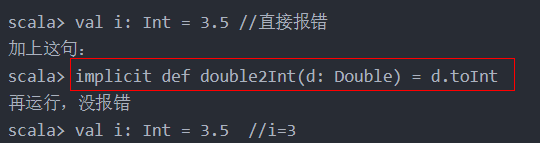
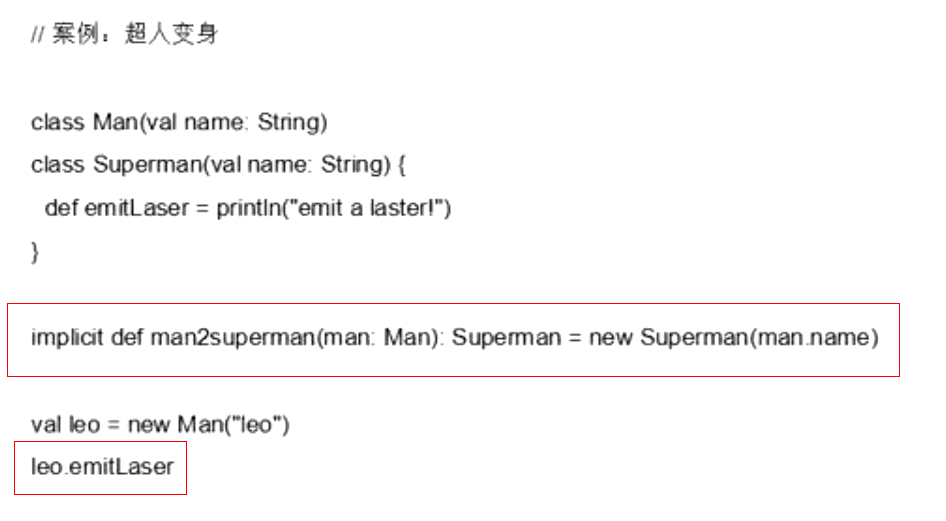


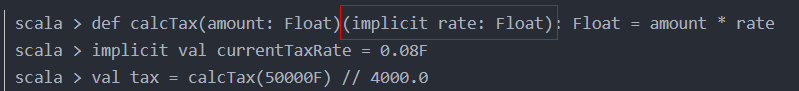
图2（超人变身的例子）



### 1.11.2 隐式参数

一个方法可以把最后一个参数列表定义为隐式，这样当包含隐式参数的方法被调用时，会从上下文找合适的隐式变量，并将该隐式变量的隐式值自动填充到隐式参数中。

图1



## 1.12 Actor

Scala的Actor类似于Java的多线程。

（注意：Spark使用的是分布式多线程框架Akka，Akka实现了类似于Actor的模型，核心概念也是Actor。）

Scala提供了Actor trait，只要重写act方法就行。使用start()启动actor，使用!向actor发送消息，actor使用receive接收消息。

（注意：发送消息默认是异步的，如果要同步发送则使用!?发送，同步会阻塞）

图1

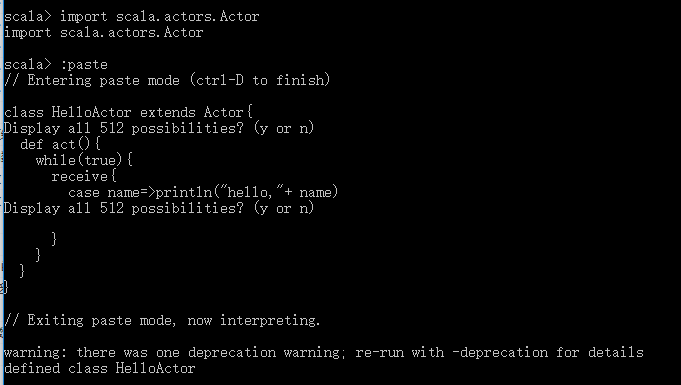






图2（!向Actor发送消息）



### 1.12.1 向Actor发送case class类型的消息

图1

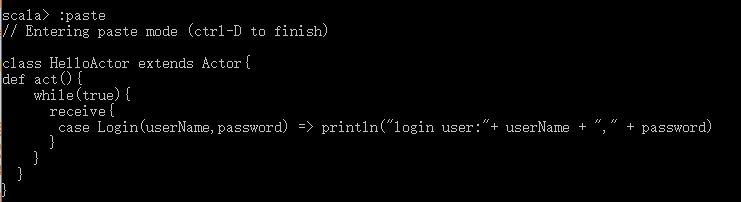
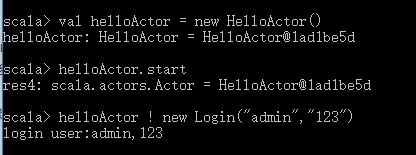


图2



### 1.12.2 Actor和Actor互相通信

Actor和Actor互相通信时，需要持有对方Actor的引用。