# 课程目标

## 目标1：（基本）熟练使用scala编写Spark程序

## 目标2：（中级）动手编写一个简易Spark通信框架

## 目标3：（高级）为阅读Spark内核源码做准备

# 课程知识点：

1. scala基础语法

基本语法，

方法

函数

1. 集合框架

array list map set 元组 集合常用方法

1. 面向对象

对象 类 抽象类 特质 模式匹配

1. rpc通信 并发编程

Actor akka spark的通信机制

1. 高级语法

科里化，隐式转换 上下界 泛型 逆变斜变

# Scala概述

## 什么是Scala

10 先乘以10 再除以2 再加5

map(x=> x\*10).map(x=>x%2).map(x=>x+5)

函数式编程：

函数式编程是一种编程思想，主要的思想**把运算过程尽量写成一系列的函数调用**。

Scala是一种多范式的编程语言，其设计的初衷是要集成面向对象编程和函数式编程的各种特性。

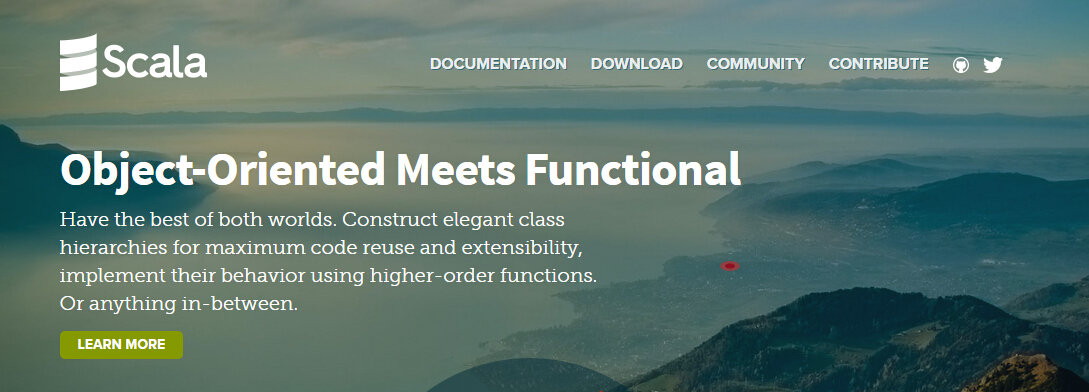
scala之父：Martin Odersky

helle.scala -🡪 .class 运行在jvm上

Scala运行于Java平台（Java虚拟机），并兼容现有的Java程序。

scala是对java的进一步封装，基于java来开发的。

也就是说，scala的代码最终会被编译为字节码文件，并运行在jvm上。



## 为什么要学Scala

1. 优雅：这是框架设计师第一个要考虑的问题，框架的用户是应用开发程序员，API是否优雅直接影响用户体验。
2. 速度快：Scala语言表达能力强，一行代码抵得上Java多行，开发速度快。

scala语言风格简洁，也很可能降低了可读性，所以学习及以后开发过程中，都需要有良好的**代码规范**。

3.**Spark的开发语言**，掌握好scala，就能更轻松的学好spark。

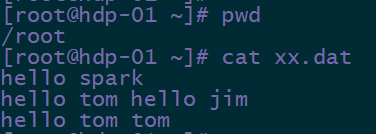
4.能融合到Hadoop生态圈：Hadoop现在是大数据事实标准，Spark并不是要取代Hadoop，而是要完善Hadoop生态。JVM语言大部分可能会想到Java，但Java做出来的API太丑，或者想实现一个优雅的API太费劲。



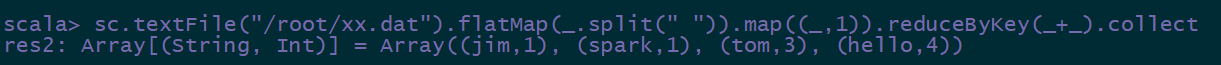
## Spark函数式编程初体验Spark-Shell之WordCount

hadoop中wordcount流程：

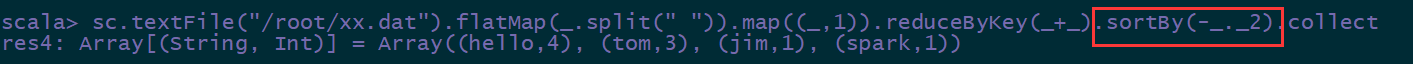
1. 读数据，切分
2. word,1
3. 拿到一组key相同的数据，计数



***Q1: 对上述文件内容使用Spark进行单词个数统计?***



***Q2: 对上述输出结果进行降序 ?***



**注：**上述代码，暂不需要练习

# Scala开发环境

## 安装JDK

因为Scala是运行在JVM平台上的，所以安装Scala之前要安装JDK

使用 # java -version 来验证

确保已安装jdk1.8+

## 安装Scala

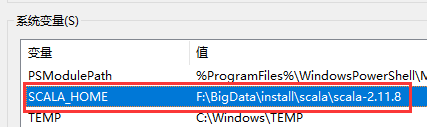
### Windows安装Scala编译器

访问Scala官网<http://scala-lang.org/download/2.11.8.html> 下载Scala编译器安装包，目前最新版本是2.12.x，但是目前大多数的框架都是用2.11.x编写开发的，Spark2.x使用的就是2.11.x，所以本课程使用2.11.x版本

安装方式：直接使用免安装版的，解压即可。



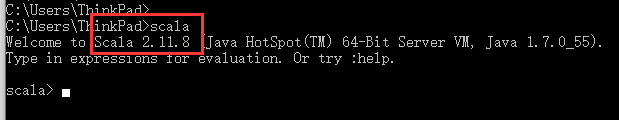
安装完成之后，配置环境变量SCALA\_HOME和PATH：

可以在cmd窗口下验证： 输入scala -version 查看scala版本



输入scala 可进入scala shell交互模式



输入：q 或：quit退出scala交互命令行。

该交互模式，有一个高大上的名称：REPL

Read Evaluate Print Loop

（读取-求值-打印-循环）

### Linux中安装Scala编译器

下载Scala地址<https://downloads.lightbend.com/scala/2.11.8/scala-2.11.8.tgz>

1,上传并解压Scala到指定目录

# tar -zxvf scala-2.11.8.tgz -C /usr/local/

2,创建一个软连接（可选项）

# ln -s 源文件目录 软连接目录

# ln -s /usr/local/scala-2.11.8 /usr/local/scala

3,配置环境变量，将scala加入到PATH中

# vi /etc/profile

|  |
| --- |
| export SCALA\_HOME=/usr/local/scala  export PATH=$PATH:$JAVA\_HOME/bin:$SCALA\_HOME/bin |

## Linux下运行第一个scala程序

### 代码编写：

# vim ScalaTest

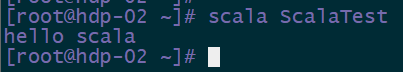
|  |
| --- |
| object ScalaTest {  def main(args: Array[String]) :Unit={  println("hello scala")  }  } |

### 代码编译：

# **scalac** ScalaTest

### 代码运行：

# **scala** ScalaTest



运行流程（类似于java）：

先编译（scalac），再执行(scala)

scala中，不强制要求源文件和类名一致。

## IDEA安装

目前Scala的开发工具主要有两种：Eclipse和IDEA，这两个开发工具都有相应的Scala插件，如果使用Eclipse，直接到官网下载即可

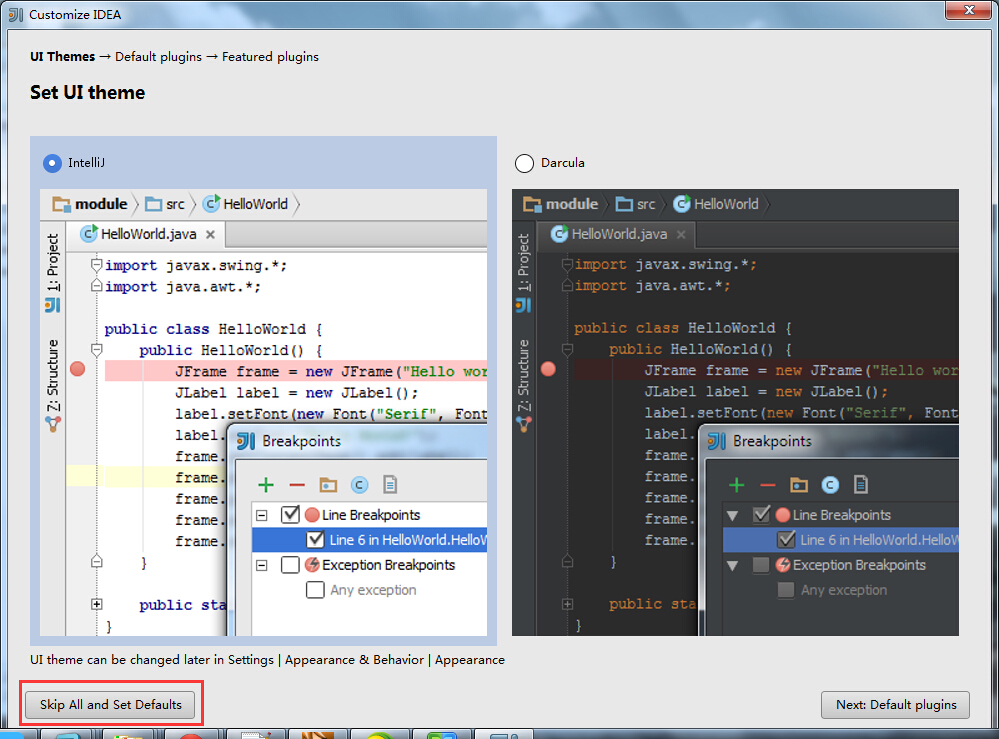
<http://scala-ide.org/download/sdk.html> **不推荐**使用该种方式

IDEA的Scala插件更优秀，有逼格的Spark攻城狮都选择IDEA（只需一次，就会爱上她）

下载地址：<http://www.jetbrains.com/idea/download/>

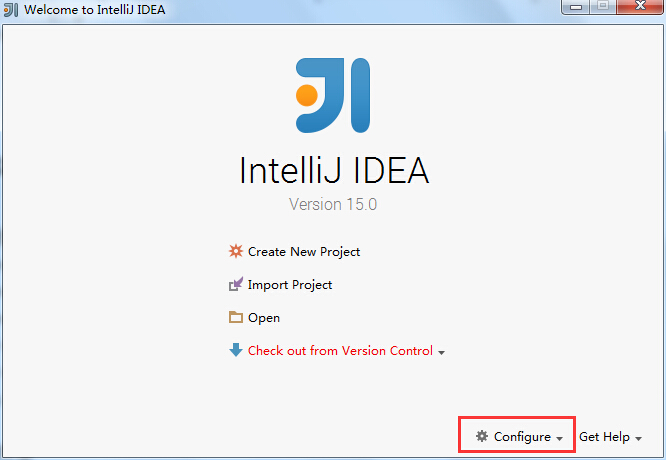
下载社区免费版，点击下一步安装即可，安装时如果有网络可以选择在线安装Scala插件。这里我们使用离线安装Scala插件：

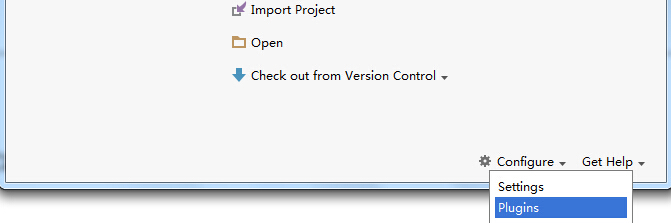
1. 安装IDEA，点击下一步即可。由于我们离线安装插件，所以点击Skip All and Set Default
2. 下载IEDA的scala插件，地址<http://plugins.jetbrains.com/?idea_ce>

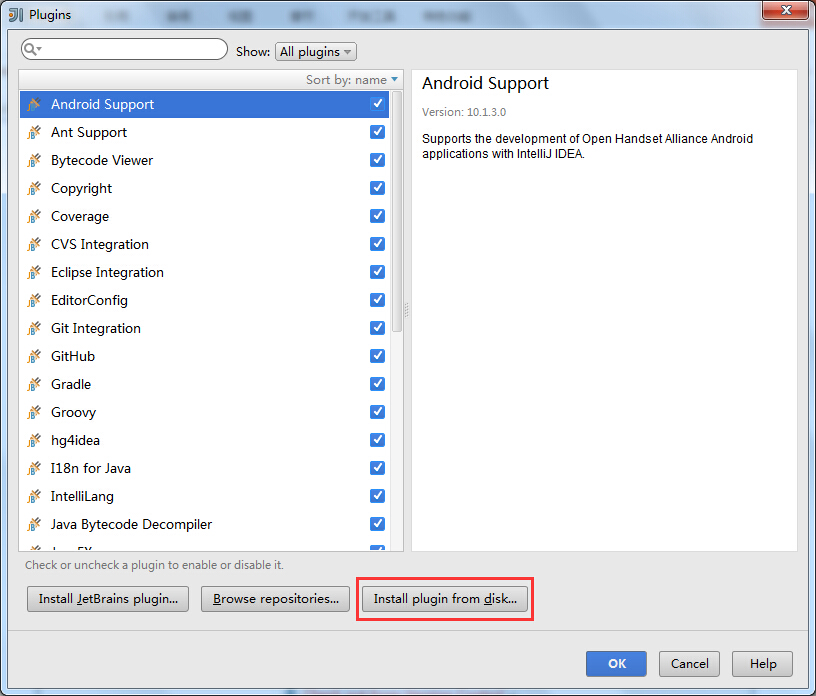


## Scala插件离线安装

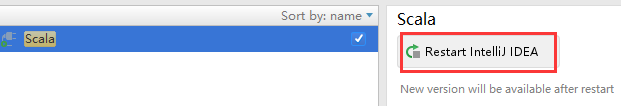
1. 安装Scala插件：Configure -> Plugins -> Install plugin from disk -> 选择Scala插件 -> OK -> 重启IDEA





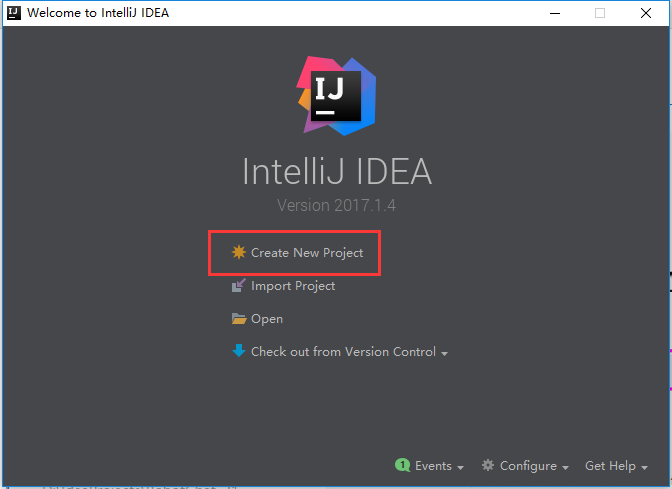


选择好插件后，重启IDEA：

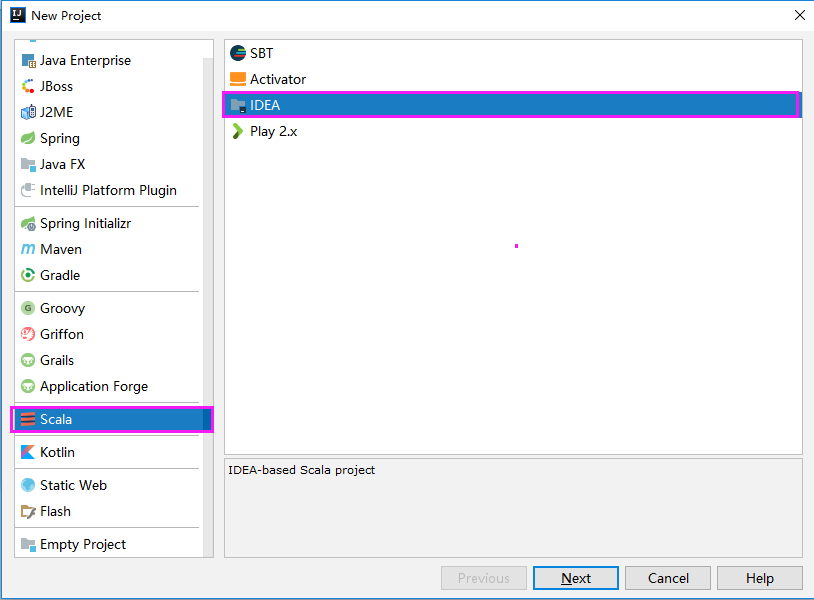


## IDEA创建Scala工程

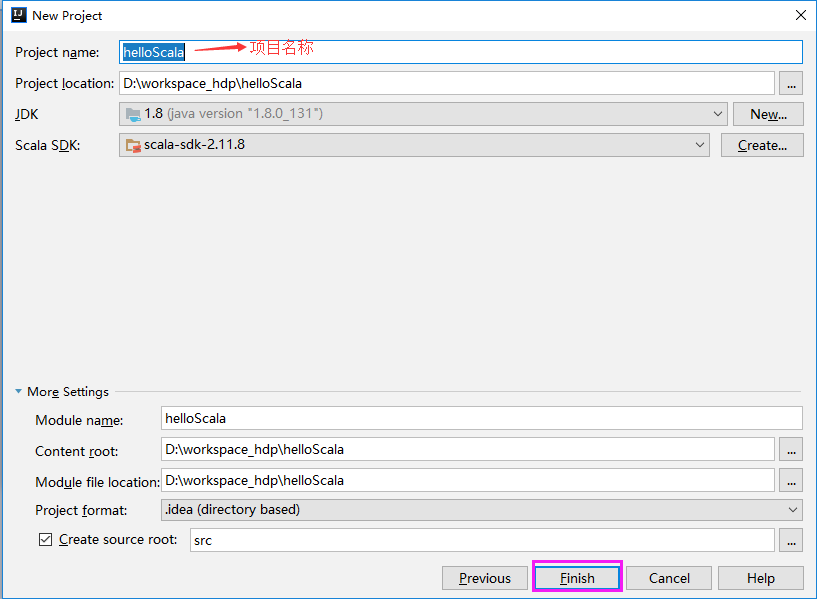
安装完成后, 双击打开IDEA, 创建一个新的项目(Create New Project)



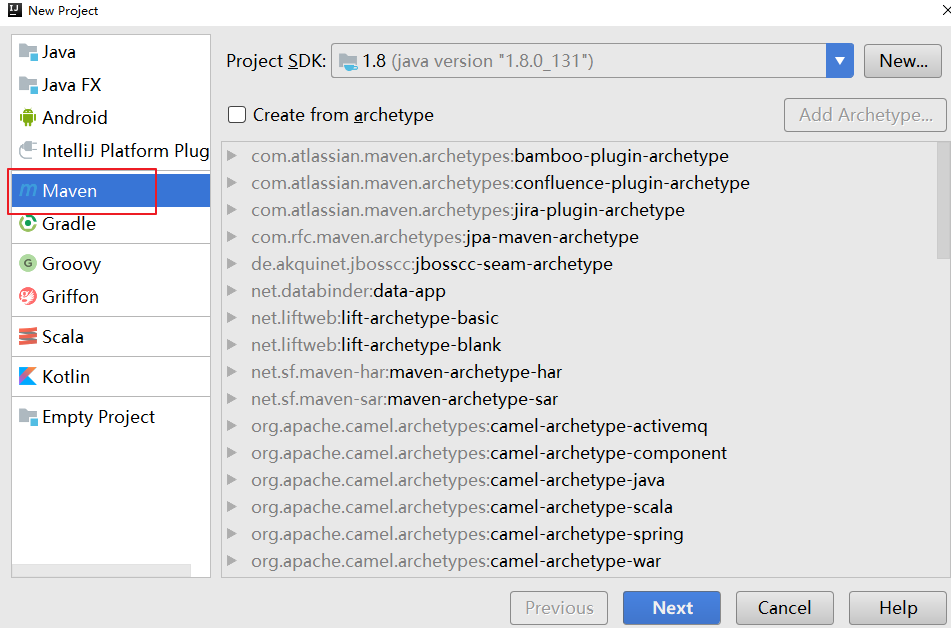
选中左侧的Scala -> IDEA -> Next

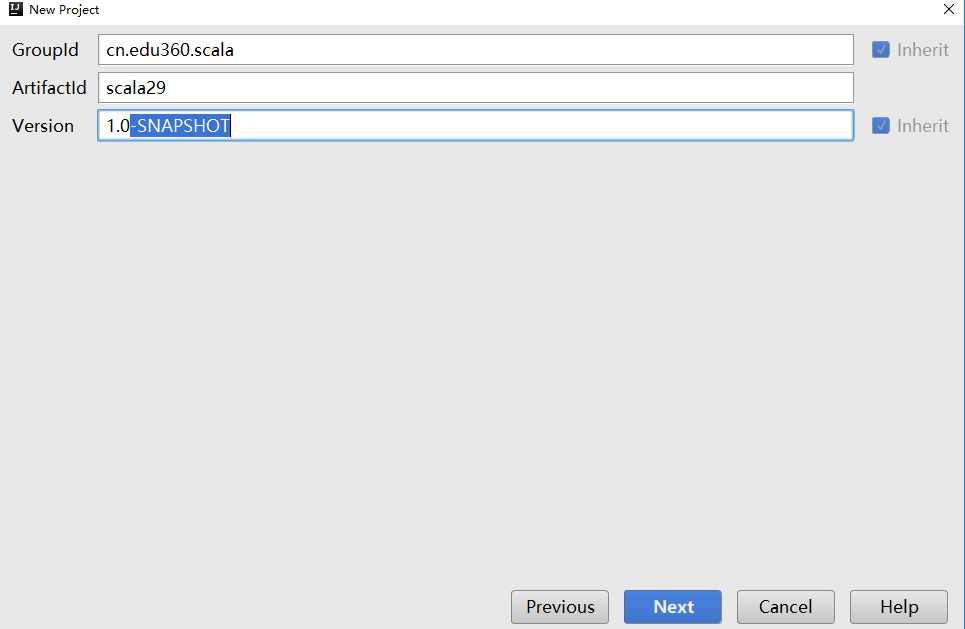


输入项目名称 -> 点击Finish完成即可



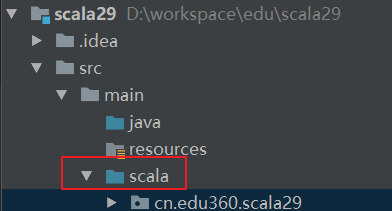
## IDEA创建maven工程





## maven工程设置：

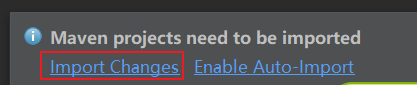
1，创建一个源码包：scala



2，在pom.xml文件中设置源码包：

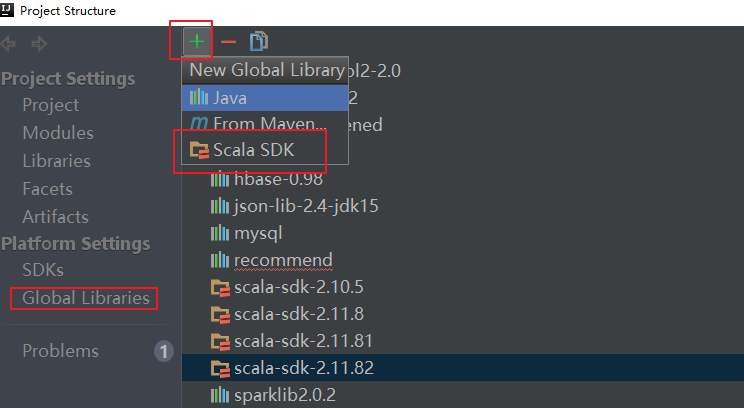


3，重新加载pom.xml文件



添加scalaDK：

ctrl+ alt + shift + s



1. 创建包，新建scala object
2. 编写代码
3. 运行

## IDEA中的第一个scala程序

new 一个 object

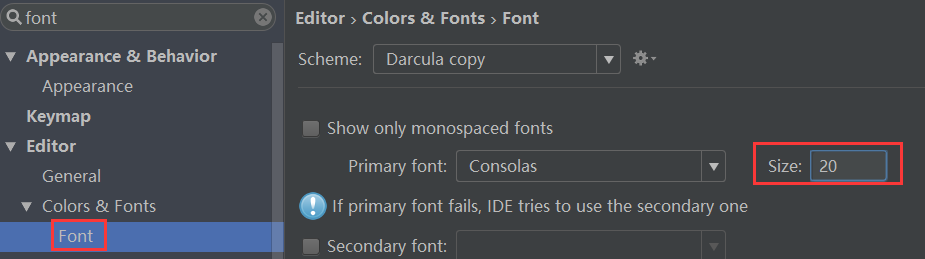
定义一个main方法

编写程序

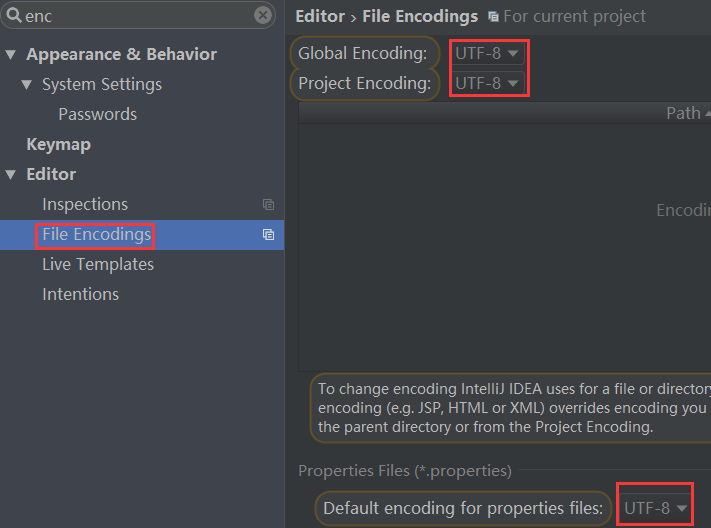
## IDEA常用配置：

Ctrl+Alt + s 进入到settings配置页面。

### 修改字体：

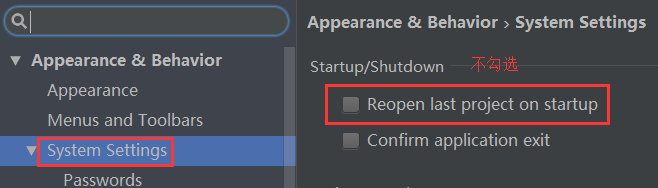


### 修改字符集：



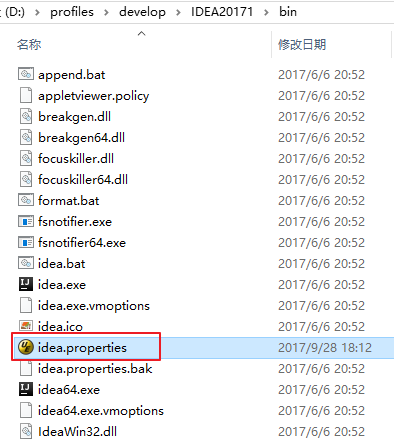
### 去除自动进入上次项目

如果每次启动IDEA，直接进入到项目页面，没有引导页面，修改配置如下：



修改安装目录的idea.properties文件

该文件在idea的安装目录下：



把idea的缓存及配置移动到其他的盘中，去C盘化

|  |
| --- |
| # idea.config.path=${user.home}/.IdeaIC/config  idea.config.path=d:/profiles/develop/IDEACache/config  #---------------------------------------------------------------------  # Uncomment this option if you want to customize path to IDE system folder. Make sure you're using forward slashes.  #---------------------------------------------------------------------  # idea.system.path=${user.home}/.IdeaIC/system  idea.system.path=d:/profiles/develop/IDEACache/system |

# Scala基础

## 常用类型

Scala和java一样，

AnyVal

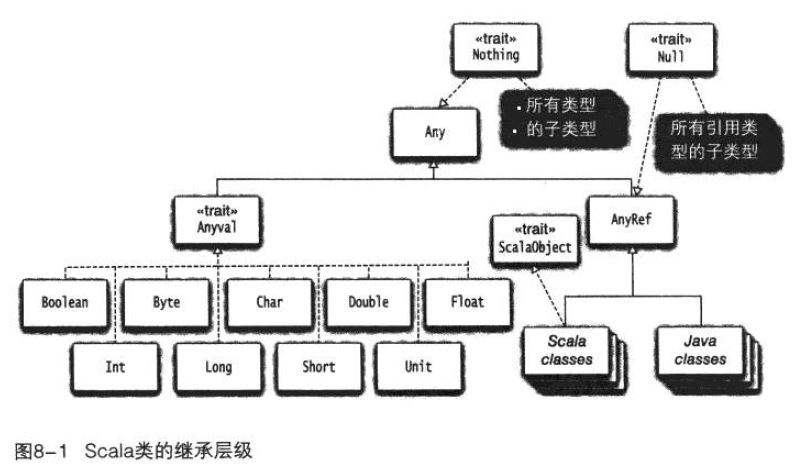
有7种**数值**类型：Byte、Char、Short、Int、Long、Float和Double（没有基本类型和包装类型的区分）

2种非数值类型： Boolean 和 Unit

**注意：**Unit表示无值，相当于java中的void。用作不返回任何结果或者结果为空的类型。Unit类型只有一个实例值，写成()。（小括号）

String是属于引用类型

AnyRef



## 声明变量

定义变量使用var 或者 val关键字

语法: var|val 变量名称 (: 数据类型) = 变量值

使用val定义的变量是不可变的，相当于java中用final修饰的变量

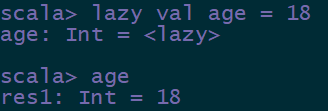
使用var定义的变量是可变的，推荐使用val

优先使用val，在循环的时候，会使用到var

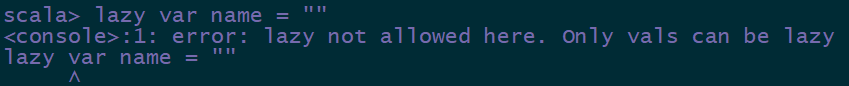
Scala编译器会自动推断变量的类型，必要的时候可以指定类型

lazy val 变量名

lazy关键字修饰的变量，是一个惰性变量，实现延迟加载（懒加载），在使用的时候才会加载。



lazy关键字不能修饰 var类型的变量



可以使用通配符（占位符）\_来指定变量

var name:String = \_

需要注意：

1. 使用占位符的时候，必须指定变量的类型
2. 变量只能使用var来修饰。
3. 使用占位符定义变量的时候，不能在main方法中定义。

|  |
| --- |
| **object** VariableTest {  **def** main(args: Array[String]) {  *// 使用val定义的变量值是不可变的，相当于java里用final修饰的变量*  *//变量名在前，类型在后* **val** name: String = “nvshen”  *// 使用var定义的变量是可变的，在Scala中鼓励使用val* **var** age = **18** *//Scala编译器会自动推断变量的类型，可以省略变量类型* **val** str = **"world"**  *// 声明多个变量*  **var** age，fv = **18**  **var** str: String = \_} } |

**可以同时声明多个变量，可以使用通配符声明变量：**

java中的通配符是\*,scala中的通配符是\_

定义一个变量，必须赋予初始值，如果没有初始值，可以使用\_占位符代替，但是变量必须指定类型。而且占位符变量不能定义在main方法内部。

## 条件表达式

**表达式都是有返回值的。**

条件表达式的值可以赋值给一个变量

支持混合类型的表达式。

Scala的条件表达式比较简洁，例如：

|  |
| --- |
| **object** ConditionTest {  **def** main(args: Array[String]) {  **val** x = 1  *//判断x的值，将结果赋给y* **val** y = **if** (x > 0) x **else** -1  *//打印y的值  println*(y)  *//如果缺失else，相当于if (x > 2) 1 else ()* **val** m = **if** (x > 2) 1  *println*(m)  *//在scala中每个表达式都有返回值，scala中有个Unit类，写做(),相当于Java中的void* **val** n = **if** (x > 2) 1 **else** ()  *println*(n)  *//支持混合类型表达式* **val** z = **if** (x > 1) 1 **else "error"** *//打印z的值  println*(z)  *混合类型会返回父类类型。*   *//if和else if* **val** k = **if** (x < 0) 0 **else if** (x >= 1) 1 **else** -1  *println*(k)  } } |

条件表达式总结：

1. 条件表达式，是有返回值的，可以使用变量接收条件表达式的值
2. 条件表达式的返回值是由谁来决定的？

由每一个分支，最后一行的值来决定的。

（比如最后一行是3>2 ,返回值是true；val a = 123 ,返回值是（））

1. 如果缺少某一个分支，默认的返回值类型是Unit，值是（）

if (age>10) age == if(age>10) age else ()

1. 在混合类型中，返回值的类型，一般情况下是所有分支返回值类型的一个父类(如果两个分支的数据类型可以转换，)

|  |
| --- |
| val result = if(x >10){  x // Int  }else {  99.9  }  result: Double |

1. 当每一个分支，只有一行内容的时候，就可以省略大括号，而且可以写在一行。推荐大家都写上{}

## 块表达式

{} 称为块表达式，块表达式中可以包含一系列的表达式，最后一个表达式的值就是块的值。

|  |
| --- |
| **object** BlockExpressionTest {  **def** main(args: Array[String]) {  **val** x = 0  *//在scala中{}中可包含一系列表达式，块中最后一个表达式的值就是块的值  //下面就是一个块表达式* **val** result = {  **if** (x < 0){  -1  } **else if**(x >= 1) {  1  } **else** {  **"error"** }  }  *// result的值就是块表达式的结果  println*(result)  } } |

## 循环

在scala中有for循环和while循环， for循环最常用

### for循环

语法结构：**for** (i **<-** 表达式/数组/集合)

java for循环方式：

// for(i=1;i<10;i++) // 传统for循环

// for(Int I :arr) // 增强for循环

|  |
| --- |
| **object** ForTest {  **def** main(args: Array[String]) {  *//for(i <- 数组)* **val** arr = *Array*(**"a"**, **"b"**, **"c"**)  *// 遍历打印数组中的每个元素*  **for** (i <- arr) *// 类似Java中的增强for*  *println*(i)  *// 通过角标获取数组中的元素*  **val** index = *Array*(**0,1,2**)  *// 遍历打印数组中的每个元素*  **for** (i <- index) *// 类似Java中的传统for*  *println*(arr(i)) *// 获取元素的方式是（），java中是[]*  *//for(i <- 表达式),表达式1 to 10返回一个Range（区间）*  *//每次循环将区间中的一个值赋给i* **for** (i <- 1 to 6)  *println*(i)  println(arr(i)) // **报错**，如果不加{}，只会把for后面的一行当做循环的内容。  **for** (i <- 1 to 6){  *println*(i)  println(arr(i))  }   **for**(i <- 1 until 6) { *// 0 until 6 => 会生成一个范围集合Range(0,1,2,3,4,5)  println*(*array*(i))  }  *// 打印数组中的偶数元素*  *// 在for循环中，通过添加守卫来实现输出满足特定条件的数据* **for**(e <- *arr* **if** e % 2 == 0) { *// for表达式中可以增加守卫  println*(e)  }   *//高级for循环  //每个生成器都可以带一个条件* **for**(i <- 1 to 3; j <- 1 to 3 **if** i != j){   *print*((10 \* i + j) + **" "**)  }    *//for推导式：如果for循环的循环体以yield开始，则该循环会构建出一个集合*  *//每次迭代生成集合中的一个值* **val** v = **for** (i <- 1 to 10) **yield** i \* 10  *println*(v)  } } |

两个生成器： to until

1 to 10 生成 Range(1, 2, 3, 4, 5, 6, 7, 8, 9, 10) 左闭右闭区间 [ ]

1 until 10 生成 Range(1, 2, 3, 4, 5, 6, 7, 8, 9) 左闭右开区间 [ )

for循环总结：

循环的标识： **<-**

增强for循环: for(I <- arr)

普通for循环： to until

带守卫的for循环 （if 条件）

嵌套for循环 （for(i<- 1 to 3 ;j <- 1 to 3 if( I != j )))）

yield推导式 返回一个满足条件的数组

### while循环

Scala 的 while 循环和其它语言如 Java 功能一样，它含有一个条件，和一个循环体，只要条件满足，就一直执行循环体的代码。

语法结构：while(condition){ 循环体内容 }

|  |
| --- |
| **var** i = 0 **while**(i<5) {  *println*(i)  i += 1 *// i = i + 1* } |

Scala 也有 do-while 循环，它和 while 循环类似，只是检查条件是否满足在循环体执行之后检查。

|  |
| --- |
| i = 0 *// while 直接判断* **while**(i>0 && i<=5) {  *println*(i)  i += 1 } i = 0 *// do while 先执行一次循环，再进行判断* **do**{  *println*(i)  i += 1 }**while**(i>0 && i<=5) |

循环也有返回值，只不过都是Unit

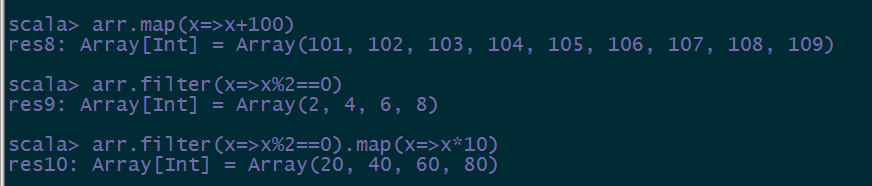
|  |
| --- |
| **val** res = **while**(i>0 && i<=5) {  *println*(i)  i += 1 } *println*(**s"res = $**res**"**) |

插值法

## 函数式编程再体验：

map：对集合或者数组中的每一个元素进行操作，该方法接收一个函数，具体的业务逻辑是自己定义的。

filter: 过滤，过滤出满足条件的元素。



## 调用方法（运算符重载为方法）

Scala中的+ - \* / %等操作符的作用与Java一样。只是有一点特别的：

这些操作符实际上是方法。操作符被重载为方法。

例如：

a + b

是如下方法调用的简写：

1. +(b)

a 方法 b可以写成 a.方法(b)

# 方法和函数（重难点）

方法：一段业务逻辑的综合。

## 定义方法

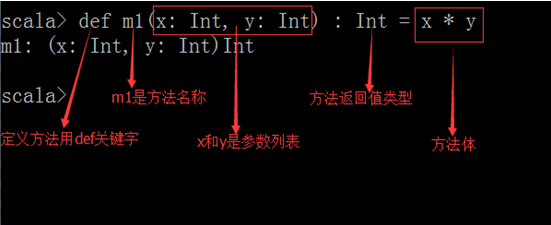
java中的方法：

public int add(int a,int b){

return a + b;

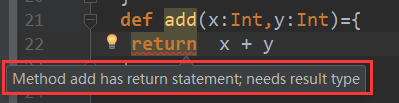
}

def methodName ([list of parameters]) : [return type] = {}



|  |
| --- |
| */\*\*  \* 方法的定义及调用  \*  \* 定义方法的格式为:  \* def methodName ([list of parameters]) : [return type] = {}  \* 如果不使用等号和方法体，则隐式声明抽象(abstract)方法。  \*/* **object** ScalaMethod **extends** App{   *// 定义个sum方法, 该方法有2个参数, 参数类型为整型, 方法的返回值为整型* **def** sum(a:Int, b: Int): Int = {  a + b  }  *// 定义有可变参数的方法,*  **def** sumAll(b: Int\*): Int = {  **var** v = 0  **for** (i<- b){  v += i  }  v *// 返回值*  }  *// 调用* **val** *result1* = *sum*(1, 5)  *println*(*result1*)  println(sumAll(1,11,13))   *// 该方法没有任何参数, 也没有返回值* **def** sayHello1 = *println*(**"Say BB1"**)  **def** sayHello2() = *println*(**"Say BB2"**)  *sayHello1 // 如果方法没有() 调用时不能加()  sayHello2 // 可是省略(), 也可以不省略* } |

如果有return关键字，必须要有返回值类型，否则报错如下：



方法总结：

1. 定义方法的关键字，def

格式： def 方法的名称（参数列表）：返回值类型 = {方法体内容}

1. 方法的返回值，最后一行的内容，如果是循环，那么返回值是Unit
2. 如果空参方法，定义的时候有（），调用的时候可以省略（），但是如果定义的时候没有（），调用方法的时候，不能加（）
3. 方法的返回值类型，可以省略，但是特殊情况下，必须加上：

4.1,方法有return关键字

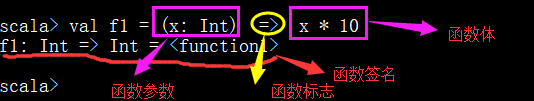
4.2，递归调用的方法。

5，方法不能最为最终的表达式存在，（空参的方法调用除外）

## 定义函数

和方法类似，基本能实现相同的功能

val| var 函数名称=（函数的参数列表） **=> 函数体**



函数可以作为最终的表达式存在，返回的内容就是函数的签名

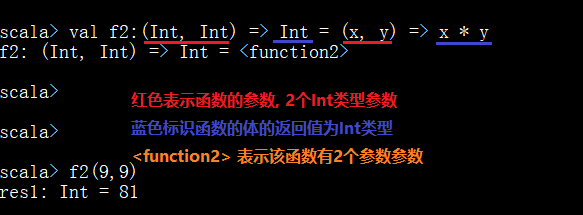
签名（函数的名称，函数的参数，函数的返回值类型）

**这种定义方式不需要指定返回值类型，编译器会自动推断**

第二种定义方式：

复杂全面的定义

val | var 函数名称：（输入参数类型）=> 返回值类型 = （参数的引用）=> 函数体



定义一个无参的函数

不同于方法，没有参数的函数定义，也必须加**()**

val f2:()=>Unit =() => println(123) val f2 =() => println(123) 返回值类型为Unit

val f2:()=>Int =() => 123 val f2=() => 123 返回值类型为Int

今日总结：

1. scala的环境，IDEA的环境搞定
2. 数据类型，变量 Any— AnyVal/AnyRef --- xxxxx
3. 条件表达式，块表达式

条件表达式都是有返回值的，返回值是由谁决定的，

混合的表达式中，返回值类型是由各个子分支的返回值的父类决定的

1. 循环 for while

for循**环是重点** **<-**

两个生成器 to until

for可以加守卫

使用yield推导式

嵌套for循环

1. 方法的定义及使用

**def** 方法名称（参数列表）：返回值类型={方法体}

1. 函数的定义和使用（\*\*\*\*）

val 函数名称=（x:Int,y:Int） => x + y

val 函数名称：（参数类型） => 返回值类型 = （参数引用） => 函数体内容

匿名函数

函数的签名信息（ <function2>）

## 方法和函数的区别

在函数式编程语言中，函数是“头等公民”，**它可以像任何其他数据类型一样被传递和操作**

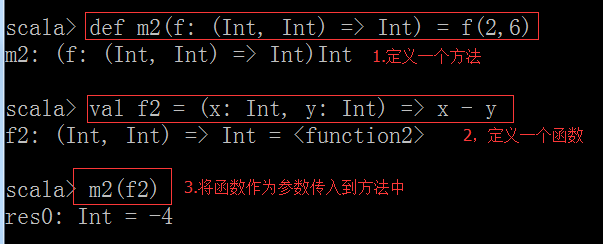
函数可以当成值来进行传递，方法的参数和返回值都可以是函数。

函数和变量，类，对象， 是一个级别的。

区别和联系：

1. 方法用def关键字定义，函数的标识 =>
2. 方法不能作为最终的表达式存在，但是函数可以，返回函数的签名信息
3. 方法和函数调用的时候都需要显示的传入参数
4. 函数可以作为方法的参数，和返回值类型。

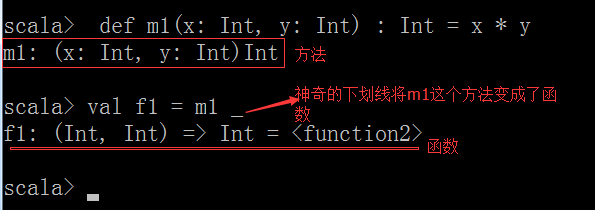
案例：首先定义一个方法，再定义一个函数，然后将函数传递到方法里面



|  |
| --- |
| **object** MethodAndFunctionTest {  *//定义一个方法  //方法m2参数要求是一个函数，函数的参数必须是两个Int类型*  *//返回值类型也是Int类型* **def** m1(f: (Int, Int) => Int) : Int = {  f(2, 6)  }   *//定义一个函数f1，参数是两个Int类型，返回值是一个Int类型* **val** *f1* = (x: Int, y: Int) => x + y  *//再定义一个函数f2* **val** *f2* = (m: Int, n: Int) => m \* n   *//main方法* **def** main(args: Array[String]) {   *//调用m1方法，并传入f1函数* **val** r1 = *m1*(*f1*)  *println*(r1)   *//调用m1方法，并传入f2函数* **val** r2 = *m1*(*f2*)  *println*(r2)  } } |

|  |
| --- |
| *// 定义一个普通方法* **def** max(x:Int,y:Int) = **if**(x>y)x **else** y  *// 定义一个方法，参数是一个函数，参数只需要函数签名，在调用的时候具体再传入函数体* **def** max1(f:(Int,Int)=>Int) = f(20,10) **def** max2(f:(Int,Int) => Int,x:Int,y:Int)= f(x,y)  *// 定义一个方法，方法返回值是函数* **def** max3()= (x:Int,y:Int)=> **if** (x>y) x **else** y **def** main(args: Array[String]): Unit = {   *println*(*max*(10,20))  *println*(*max1*((x:Int,y:Int)=>**if**(x>y) x **else** y))  *println*(*max2*((x:Int,y:Int)=>**if**(x>y) x **else** y,10,20))  *println*(*max3*()(10,20)) } |

## 将方法转换成函数（神奇的下划线）



## 方法和函数的总结（重点）：

1，方法的定义 使用**def** 关键字 函数的定义 使用 **=>**

函数有两种定义方式

val f = (x:Int,y:Int) => x + y

val f:(Int,Int) => Int = (x,y) => if(x>y) x else y

2，方法不能作为最终的表达式存在，但是函数可以作为最终的表达式存在，返回函数签名

函数签名：函数的名称，参数类型，返回值类型 函数的别名（函数参数个数）

3，方法和函数的调用，方法和函数可以相互调用。

方法名称（参数列表） 函数名称（参数列表）

无参的方法可以省略（），无参的函数不能省略括号

4，函数是一等公民，可以作为方法的参数和返回值类型。

**函数作为方法的参数**： 定义方法的时候，函数参数规范（函数的名称，参数类型，返回值类型），然后在调用方法的时候，传递的是一个已存在的函数，或者是一个匿名函数。都要求传入的函数，必须满足定义的规范，使用匿名函数的时候，可以省略参数类型（（x,y）=>x\*y）

函数最为方法的返回值类型，当调用方法的时候，返回函数，传入参数进行函数的调用。

1. 方法可以转化为函数，通过在方法名称 \_ ,另外当方法的参数是一个函数的时候，满足条件的方法会被自动转换为函数，进行传递。

|  |
| --- |
| def method1(f: (Int, Int) => Int): Int = {  | // 可以调用函数  | f(1, 100)  | }  def max(x:Int,y:Int) = x \* y  method1(max \_)  method1(max) // 方法自动转换为函数 |

方法和函数应该怎么使用？

一般情况下，优先使用函数

实际上，还是定义方法使用的更多，函数会作为方法的参数。

最最常用的：调用方法的时候传递函数。