1.课程目标

1.1. 目标 1：（基本）熟练使用 scala 编写 Spark 程序

1.2. 目标 2：（中级）动手编写一个简易 Spark 通信框架

1.3. 目标 3：（高级）为阅读 Spark 内核源码做准备

2.Scala 概述

2.1. 什么是 Scala

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**。

通过传递一系列的引用，（定义规则的时候，使用接口，具体调用的时候，传递实现类）

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。

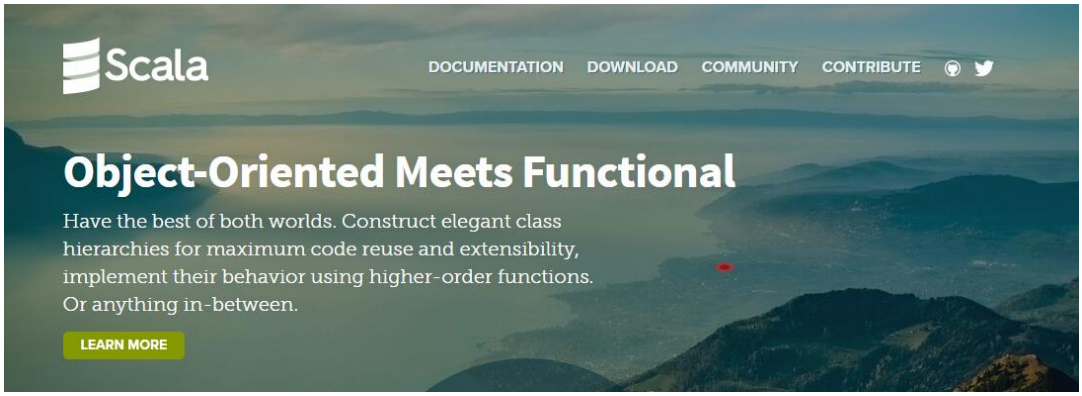
scala 之父：Martin Odersky

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。



2.2. 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。

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

```
[root@hdp-01 ~]# pwd
/root
[root@hdp-01 ~]# cat xx.dat
hello spark
hello tom hello jim
hello tom tom
```

Q1: 对上述文件内容使用 Spark 进行单词个数统计?

```
scala> sc.textFile("/root/xx.dat").flatMap(_.split(" ")).map((_,1)).reduceByKey(_+_).collect
res2: Array[(String, Int)] = Array((jim,1), (spark,1), (tom,3), (hello,4))
```

Q2: 对上述输出结果进行降序 ?

```
scala> sc.textFile("/root/xx.dat").flatMap(_.split(" ")).map((_,1)).reduceByKey(_+_).sortBy(-_._2).collect
res4: Array[(String, Int)] = Array((hello,4), (tom,3), (jim,1), (spark,1))
```

注：上述代码，暂不需要练习

3.Scala 开发环境

3.1. 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+

3.2. 安装 Scala

3.2.1. Windows 安装 Scala 编译器

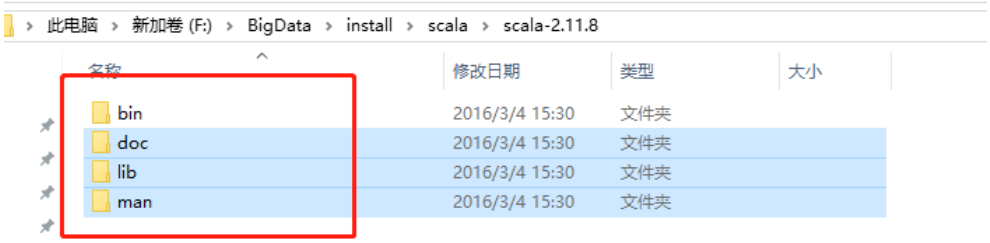
访问 Scala 官网 http://scala-lang.org/download/2.11.8.html 下载 Scala 编译器安装包，目前最新版本是 2.12.x，但是目前大多数的框架都是用 2.11.x 编写开发的，Spark2.x 使用的就是 2.11.x，所以本课程使用 2.11.x 版本

安装方式：直接使用免安装版的，解压即可。

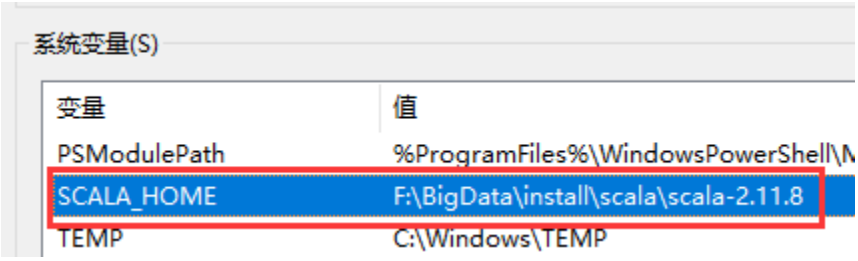
Archive	System	Size
scala-2.11.8.tgz	Mac OS X, Unix, Cygwin	27.35M
scala-2.11.8.msi	Windows (msi installer)	109.35M
scala-2.11.8.zip	Windows	27.40M
scala-2.11.8.deb	Debian	76.02M
scala-2.11.8.rpm	RPM package	108.16M
scala-docs-2.11.8.txz	API docs	46.00M
scala-docs-2.11.8.zip	API docs	84.21M
scala-sources-2.11.8.tar.gz	Sources	

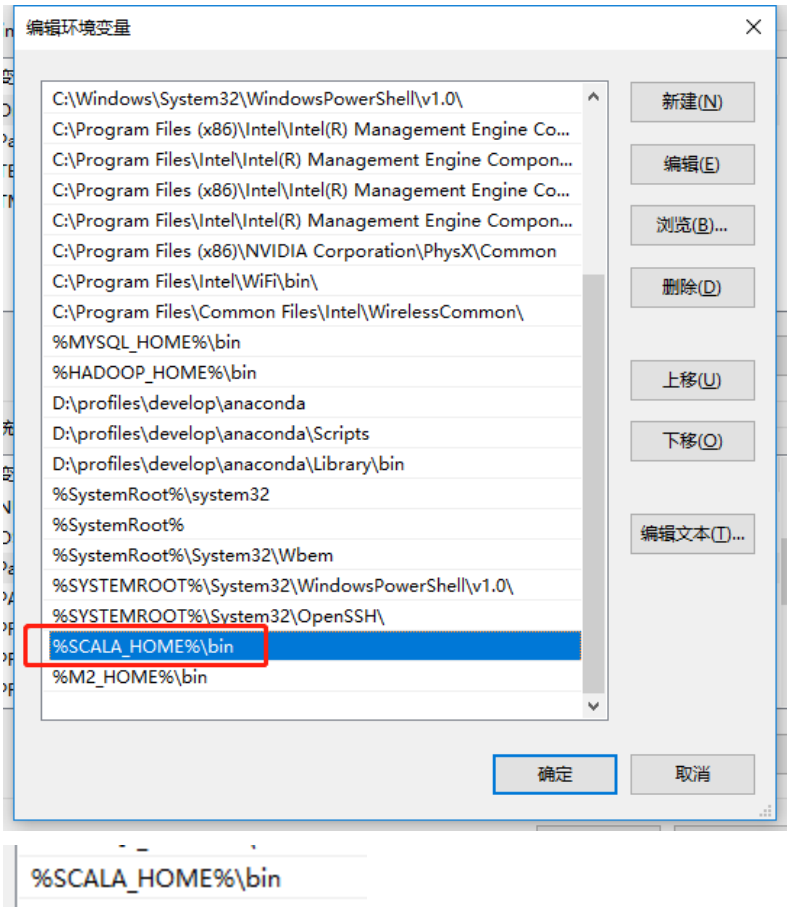
Archive	System	Size
scala-2.11.8.tgz	Mac OS X, Unix, Cygwin	27.35M
scala-2.11.8.msi	Windows (msi installer)	109.35M
scala-2.11.8.zip	Windows	27.40M

解压安装包：



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）

3.2.2. 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,配置环境变量，将 scala 加入到 PATH 中

vi /etc/profile

```
export SCALA_HOME=/usr/local/scala-2.11.8
export PATH=$PATH:$JAVA_HOME/bin:$SCALA_HOME/bin
```

3，重新的 source 环境变量，才能生效。

```
[root@hdp-01 ~]# source /etc/profile
```

4，验证

```
[root@hdp-01 ~]# scala -version
Scala code runner version 2.11.8 -- Copyright 2002-2016, LAMP/EPFL
```

3.3. Linux 下运行第一个 scala 程序

3.3.1. 代码编写：

vim ScalaTest

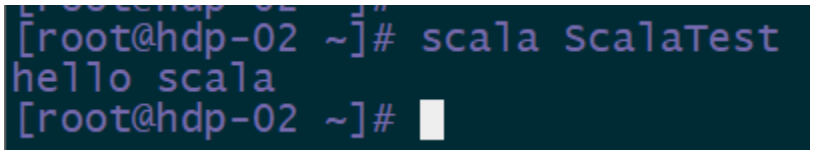
```
object ScalaTest {
      def main(args: Array[String]) :Unit={
            println("hello scala")
}
}
```

3.3.2. 代码编译：

scalac ScalaTest

3.3.3. 代码运行：

scala ScalaTest



运行流程（类似于 java）：

先编译（scalac），再执行(scala)

注意：scala 中，不要求源文件和类名一致。

3.4. IDEA 集成工具安装

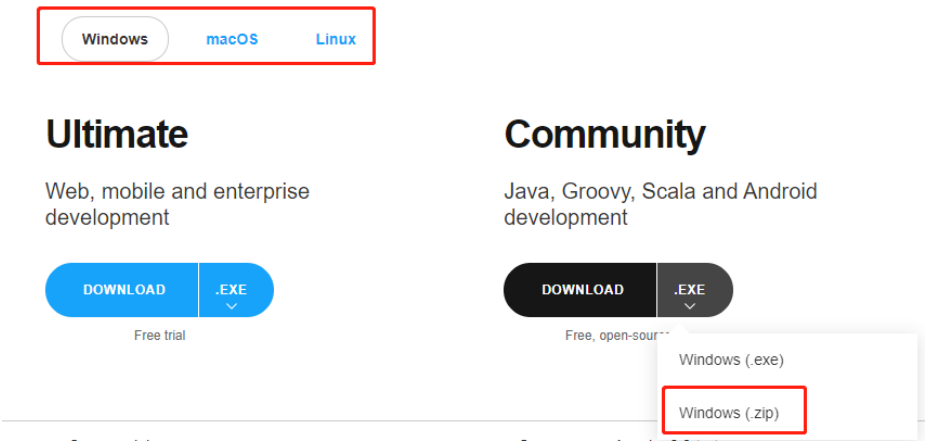
3.4.1. IDEA 安装（社区版）

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

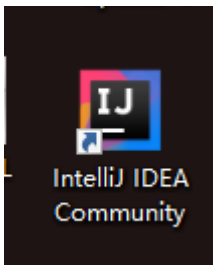
IDEA 的 Scala 插件更优秀，有逼格的 Spark 攻城狮都选择 IDEA（只需一次，就会爱上她）

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



下载社区免费版，一路 next 即可完成安装。

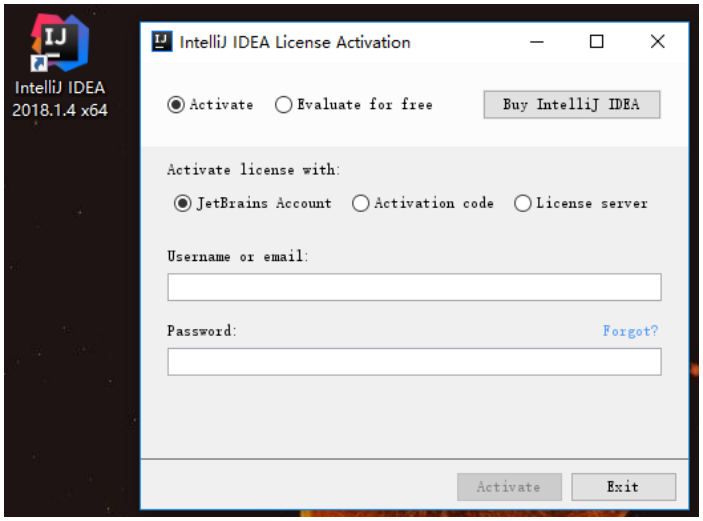
安装完成之后的界面：



3.4.2. IDEA 安装破解版

正常安装之后的图标。

双击打开之后的界面：

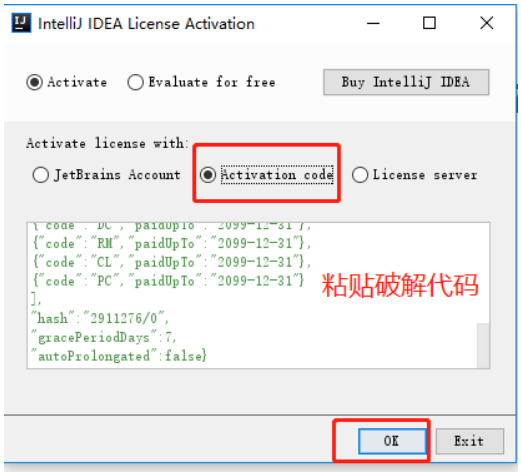


支持正版，土豪请直接购买！

以下破解步骤：

详细步骤参考相关破解文档。

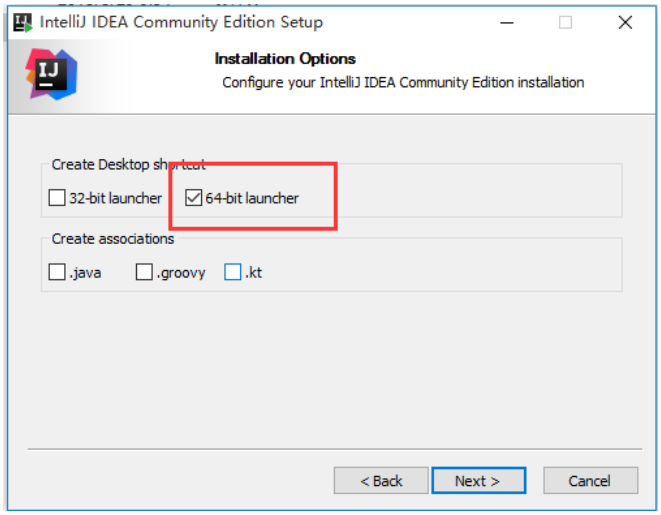
最后一步，就是粘贴破解代码重启即可正常使用。



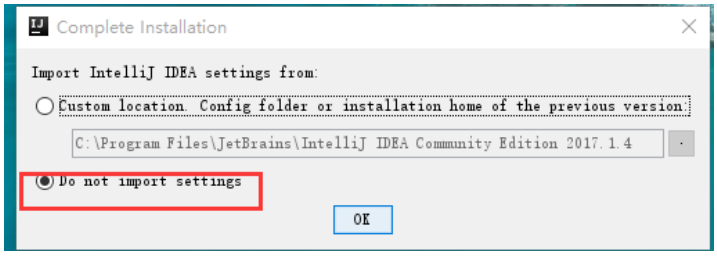
粘贴破解代码，重启即可正常使用。

3.4.3. 安装成功之后的几个选项

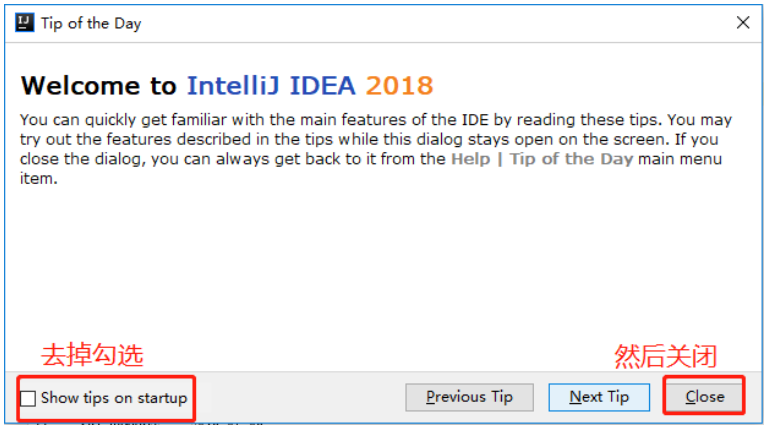
创建一个桌面快捷方式：



IDEA 启动时，如果没有 IDEA 相关的配置，不需要选择。



开启跳过：



3.5. scala 插件的安装

至此，IDEA 安装完成，如果需要正常用于开发 scala，则还需要安装 scala 的插件。

利用 IDEA 来进行 scala 开发，必须要有 scala 的插件。

安装时如果有网络可以选择在线安装 Scala 插件。

否则，可以直接从本地插件进行安装。

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

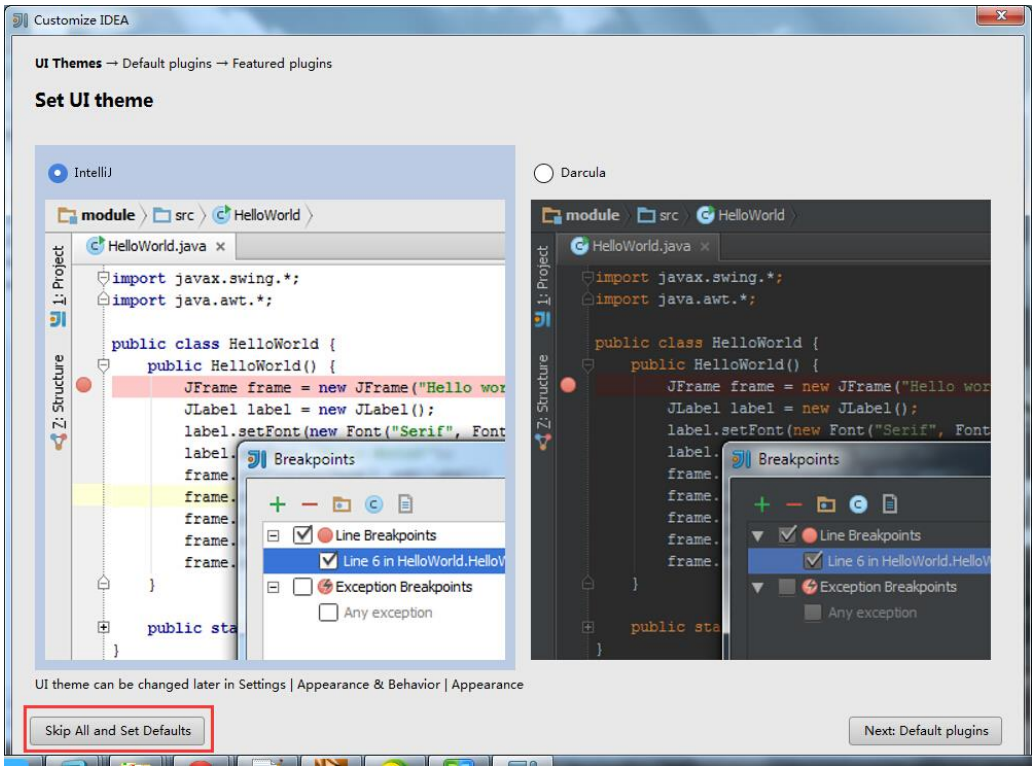
2.下载 IEDA 的 scala 插件，地址

http://plugins.jetbrains.com/?idea_ce

http://plugins.jetbrains.com/plugin/1347-scala

注意： 如果使用新版本的 IDEA，需要先下载和 IDEA 版本对应的 scala 插件

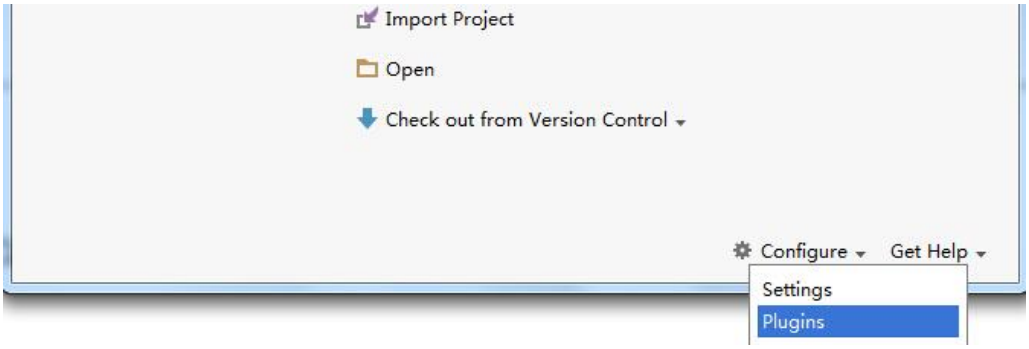
VERSION	COMPATIBLE BUILDS	UPDATE DATE	
2018.1.8	181—182	Mar 26, 2018	DOWNLOAD
2018.1.7	181—182	Mar 21, 2018	DOWNLOAD
2017.3.15	173.1751—174	Mar 15, 2018	DOWNLOAD
2017.3.14	173.1751—174	Mar 15, 2018	DOWNLOAD
2018.1.6	181—182	Mar 13, 2018	DOWNLOAD
2018.1.4	181—182	Mar 05, 2018	DOWNLOAD
2018.1.3	181—182	Feb 20, 2018	DOWNLOAD
2018.1.1	181—182	Jan 18, 2018	DOWNLOAD
2017.3.11.1	173.1751—174	Jan 12, 2018	DOWNLOAD
2017.3.11	173.1751—174	Dec 15, 2017	DOWNLOAD

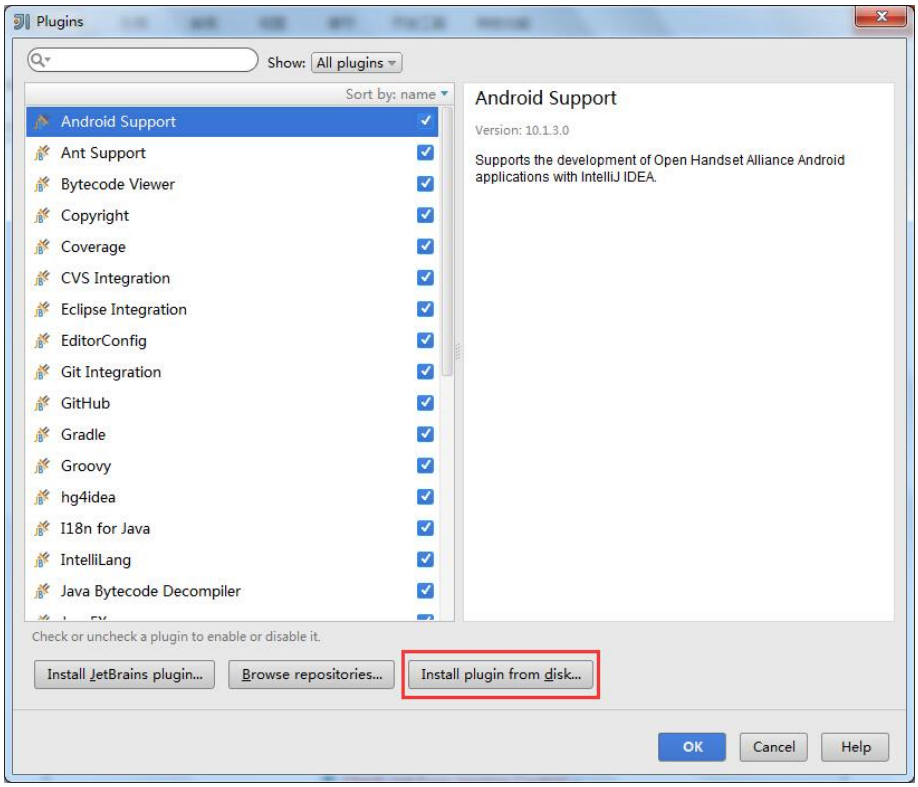


3.5.1. Scala 插件离线安装

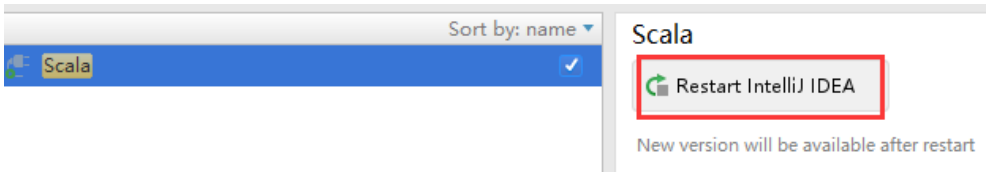
3.安装 Scala 插件：Configure -> Plugins -> Install plugin from disk -> 选择 Scala 插件 -> OK -> 重启 IDEA


Welcome to IntelliJ IDEA
IntelliJ IDEA
Version 15.0
Create New Project
Import Project
Open
Check out from Version Control
Configure
Get Help

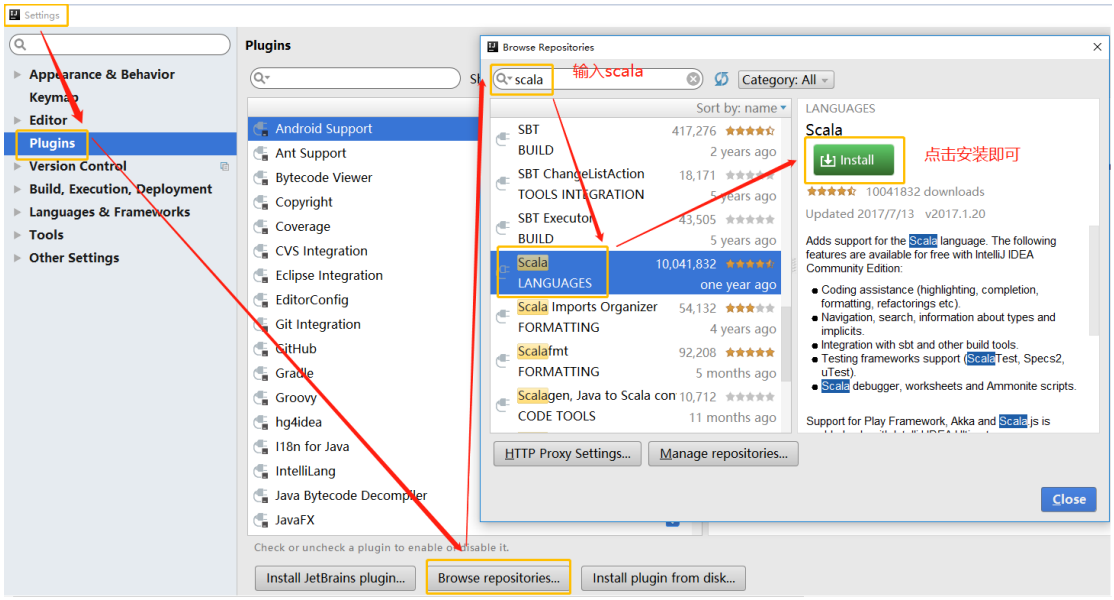

Import Project
Open
Check out from Version Control
Configure
Get Help
Settings
Plugins



选择好插件后，重启 IDEA：

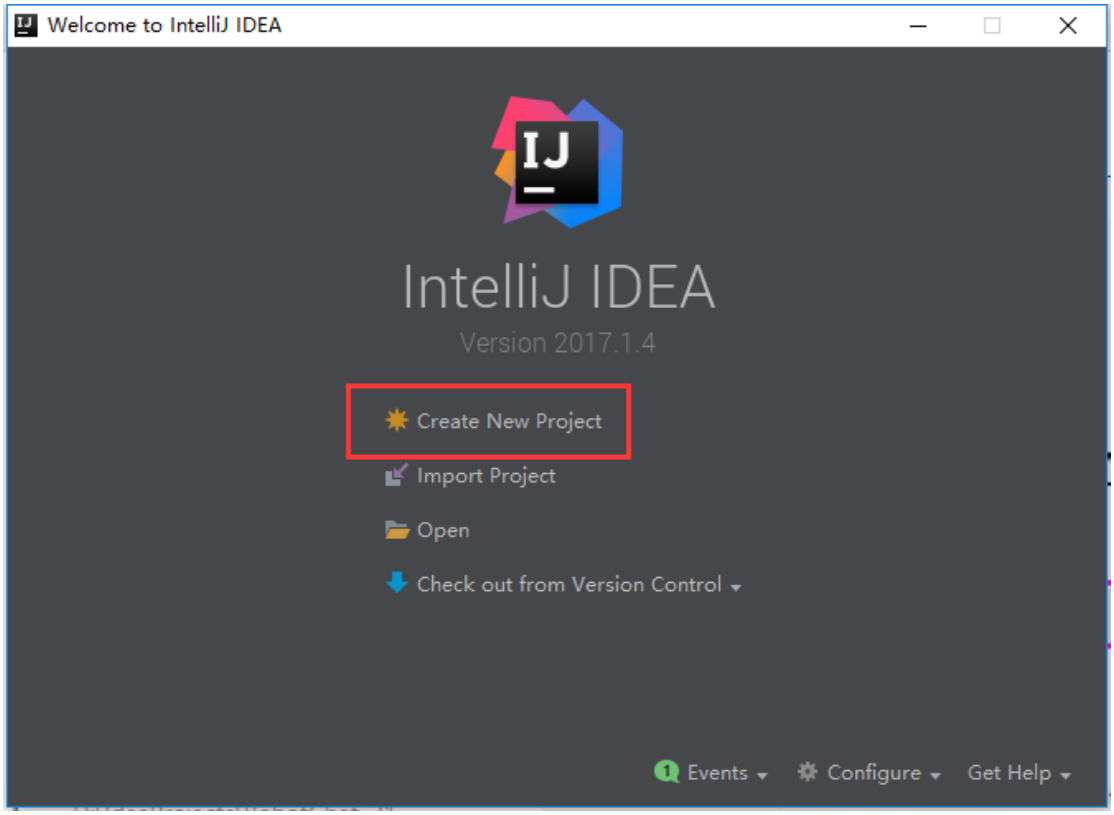


3.5.2. scala 插件在线安装

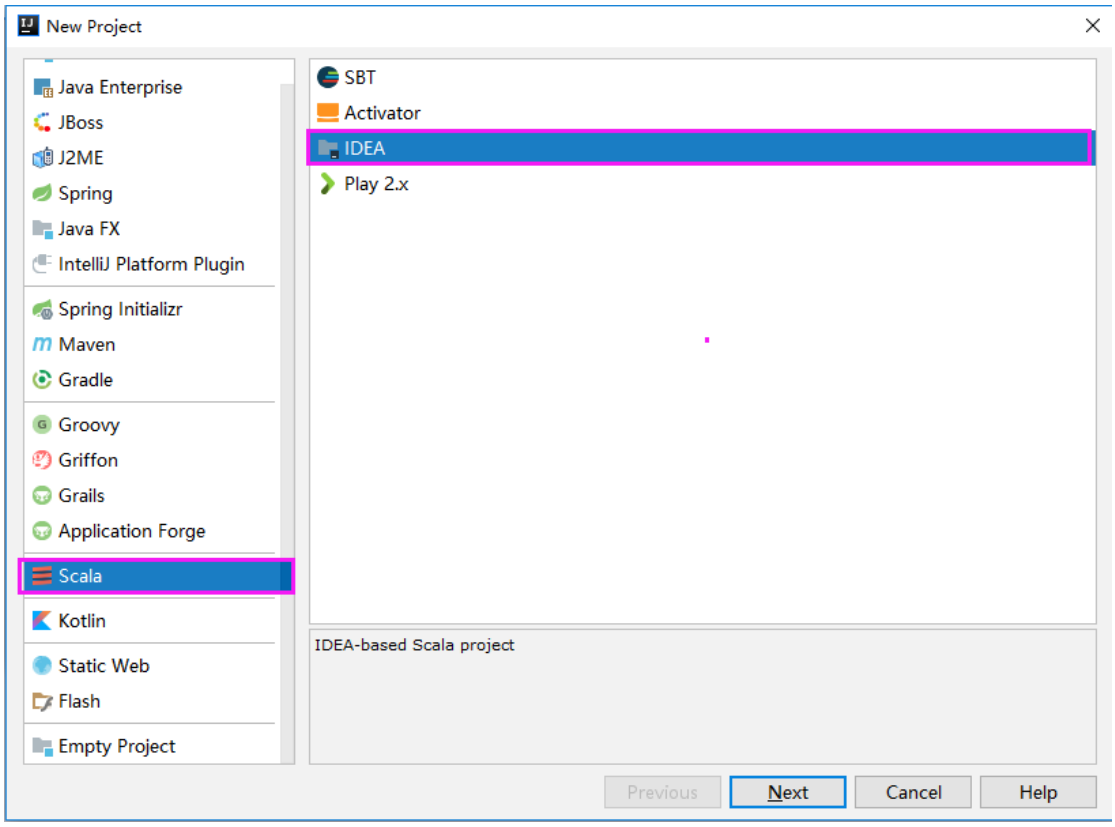


3.6. IDEA 创建 Scala 工程

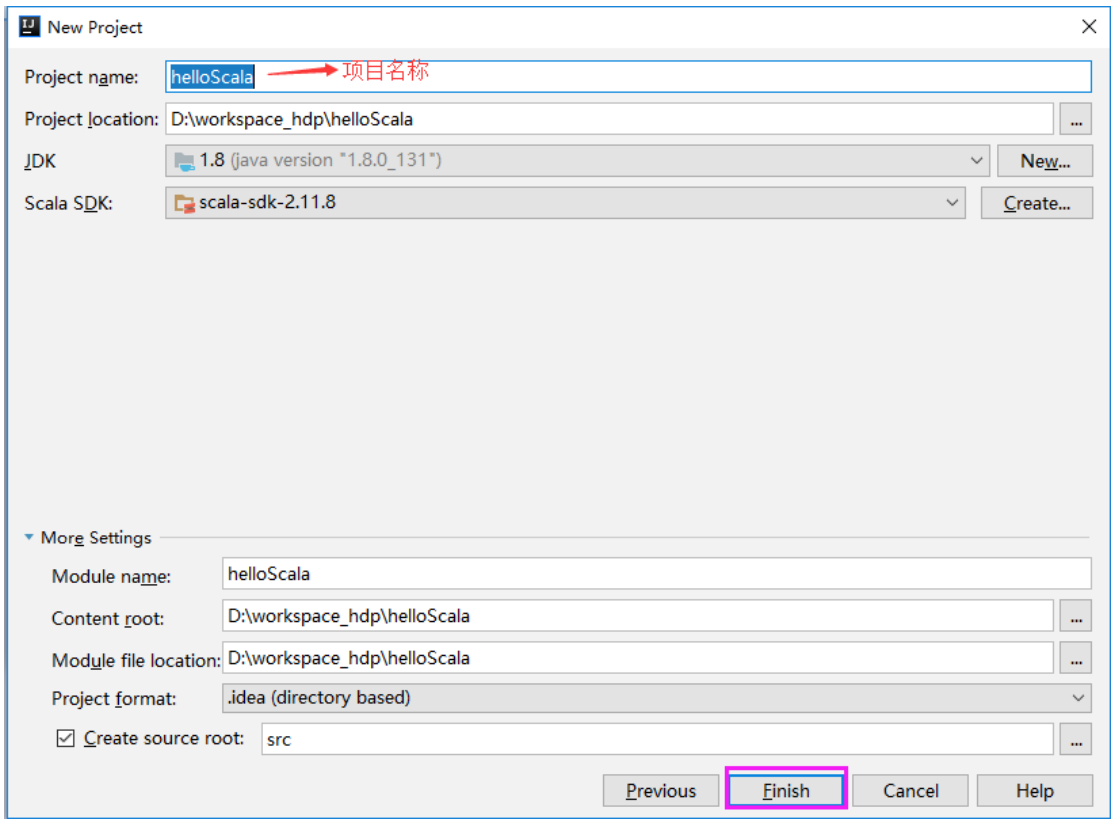
安装完成后，双击打开 IDEA，创建一个新的项目(Create New Project)



选中左侧的 Scala -> IDEA -> Next



输入项目名称 -> 点击 Finish 完成即可



3.7. IDEA 中的第一个 scala 程序

new 一个 object

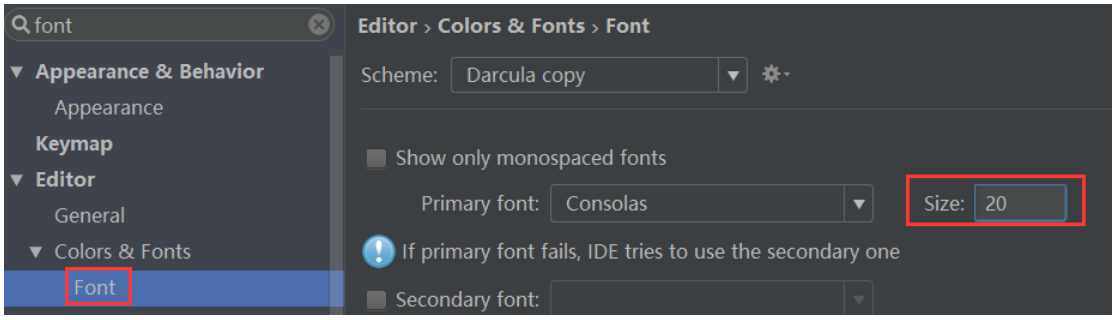
定义一个 main 方法

编写程序

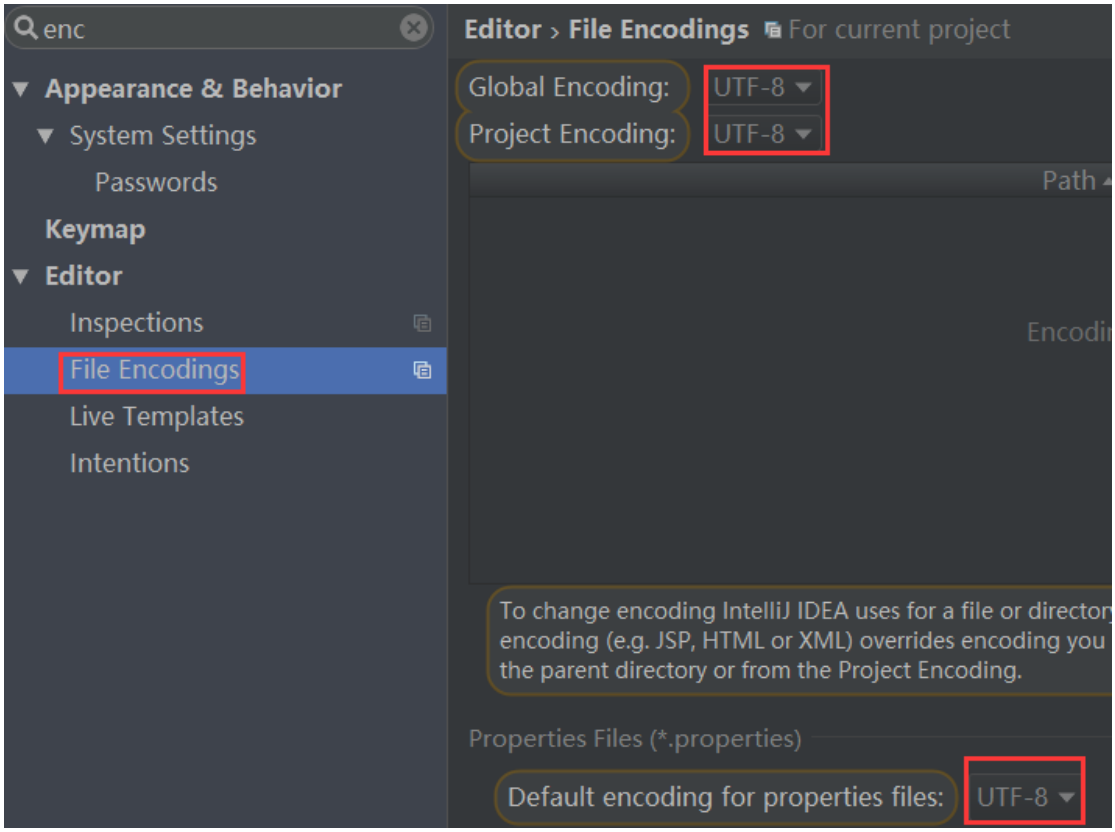
3.8. IDEA 常用配置：

Ctrl+Alt + s 进入到 settings 配置页面。

3.8.1. 修改字体：

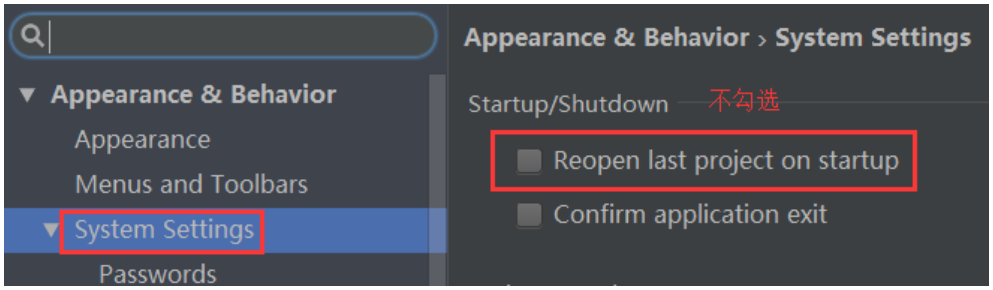


3.8.2. 修改字符集：



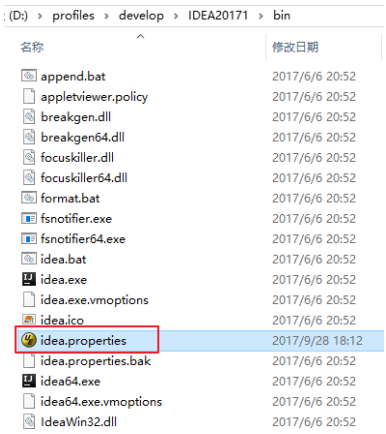
3.8.3. 去除自动进入上次项目

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：



可选：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
```

4.Scala 基础

4.1. 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

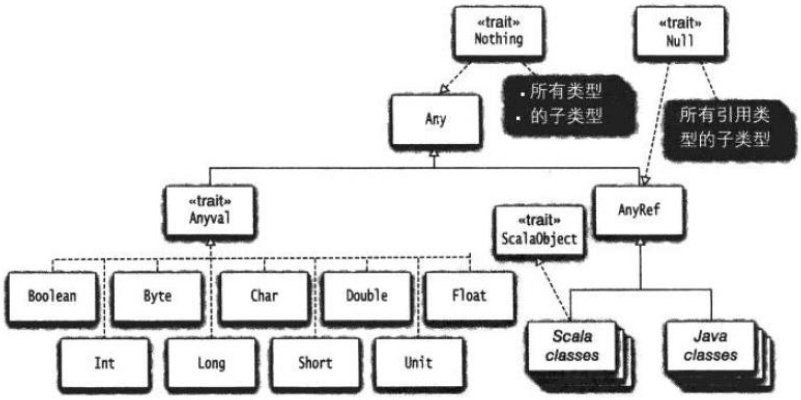


图8-1 Scala类的继承层级

4.2. 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

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。

```
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。

4.3. 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（））

3，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 ()

4，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
```

5，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{}

4.4. 循环

在 scala 中有 for 循环和 while 循环， for 循环最常用

4.4.1. 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

4.4.2. 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

4.4.3. 跳出 while 循环

scala 中没有 continue 和 break 关键字

```
//    while(true){
//      println("i love xxx ,")
//    }

    var i = 0
    while( i< 5 && i >= 0){
      println("song qing shu ")

      // 手动的让变量的值增加
      // scala  不支持 i++  只支持 i+= 1
      i += 1
    }

    i = 0
    // 先执行一次业务逻辑 再判断循环条件
    do{
      i+= 1
    }while(i< 5 && i> 0)

    val loop: Breaks = new Breaks
    // 借助breaks 类，来辅助退出循环
    loop.breakable{
      var k = 5
      while(k <= 10000){
        println(k)
        if(k == 100){
          // 调用类的break 方法
          loop.break()
        }
        k+=1
```

```
    }
  }

  var j = 100
  var flag = false
  while(true && !flag){
    // 如果满足条件，修改标识的值
    println("j="+j)
    if(j > 1000){
      flag = true
    }
    j+=1
  }
```

4.5. 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。

```
scala> arr.map(x=>x+100)
res8: Array[Int] = Array(101, 102, 103, 104, 105, 106, 107, 108, 109)

scala> arr.filter(x=>x%2==0)
res9: Array[Int] = Array(2, 4, 6, 8)

scala> arr.filter(x=>x%2==0).map(x=>x*10)
res10: Array[Int] = Array(20, 40, 60, 80)
```

4.6. 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：

a.+(b)

a 方法 b 可以写成 a.方法(b)

5.方法和函数（重难点）

方法：一段业务逻辑的综合。

5.1.定义方法

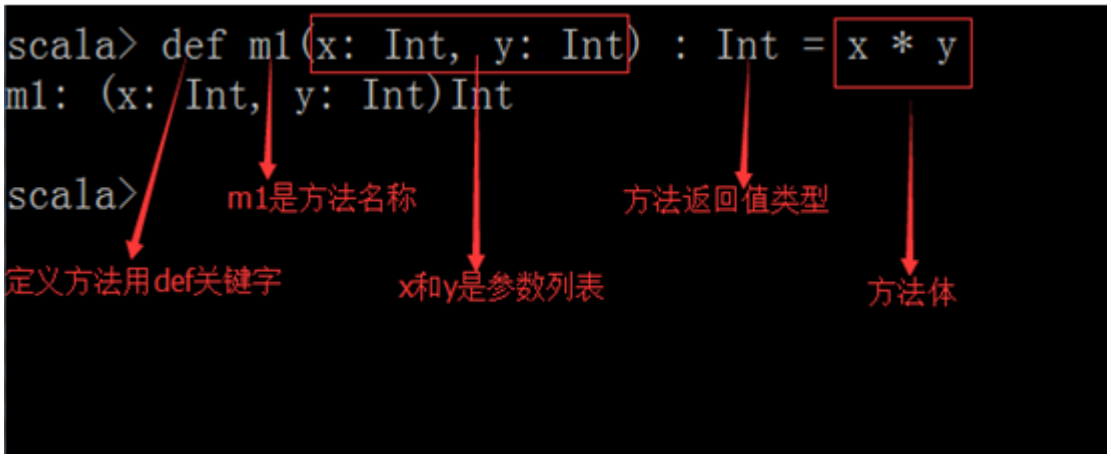
java 中的方法：

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
  *
```

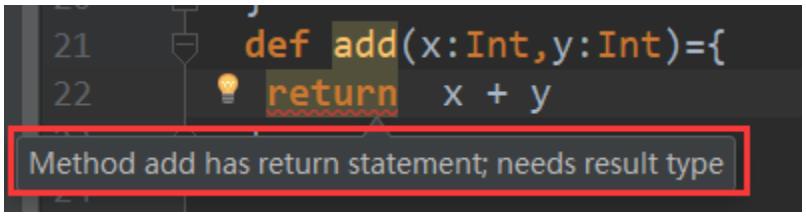
```
    *  定义方法的格式为:
    * def methodName ([list of parameters]) : [return type] = {}
    *  如果不使用等号和方法体，则隐式声明抽象(abstract)方法。
    */
object ScalaMethod extends App{

      // 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}

2，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

3，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（）

4，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

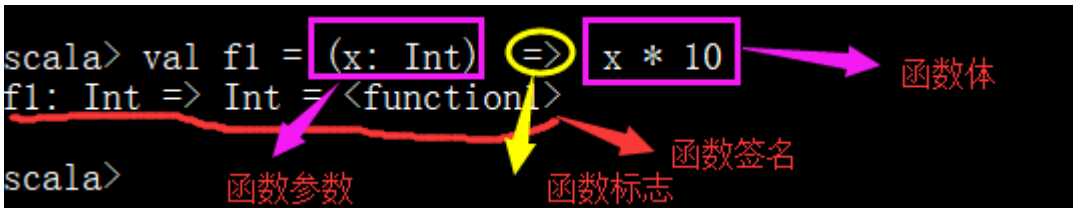
4.2，递归调用的方法。

5，方法不能最为最终的表达式存在，（空参的方法调用除外）

5.2. 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

```
scala> val f2:(Int, Int) => Int = (x, y) => x * y
f2: (Int, Int) => Int = <function2>

scala>
```

红色表示函数的参数，2个Int类型参数

```
scala>
```

蓝色标识函数的体的返回值为Int类型

<function2> 表示该函数有2个参数参数

```
scala> f2(9,9)
res1: Int = 81
```

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

5.3. 方法和函数的区别

方法本质上，就是一种特殊的函数。

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：

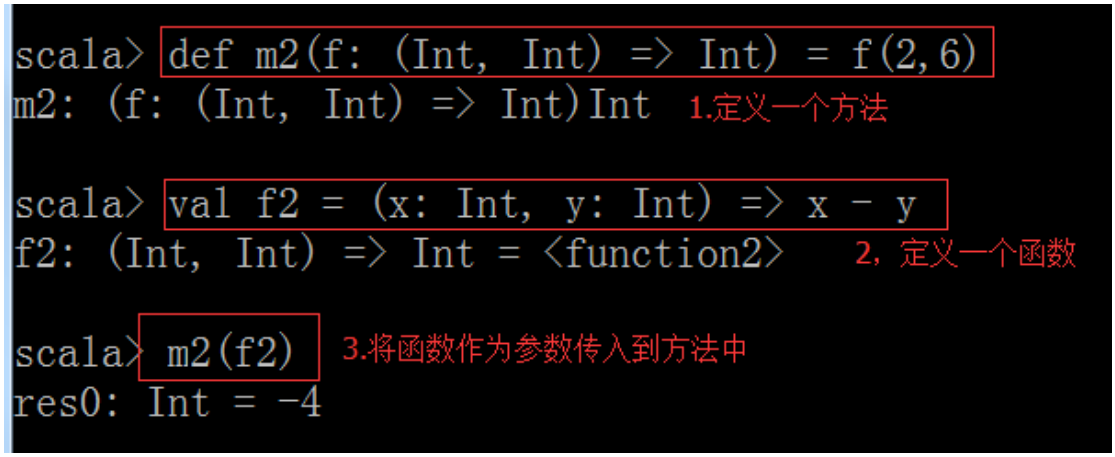
1，方法用 def 关键字定义，函数的标识 =>

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)
  }
}
```

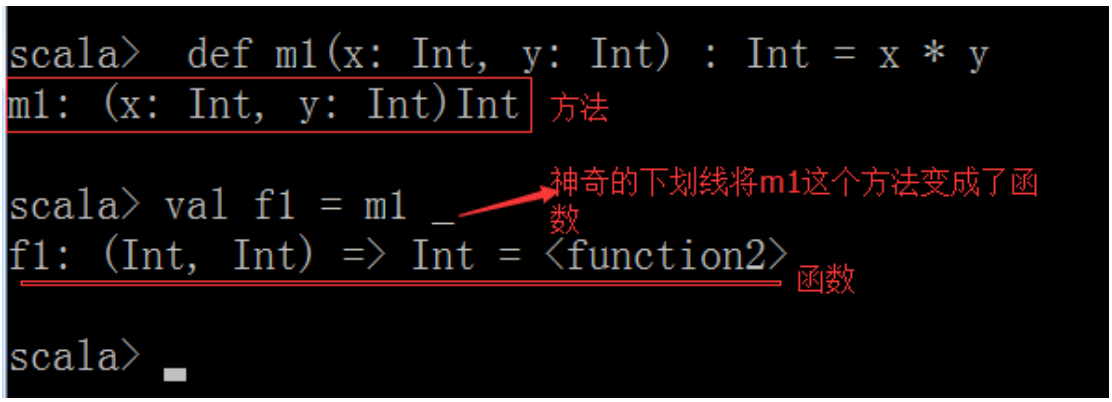
```
// 定义一个普通方法
def max(x:Int,y:Int) = if(x>y)x else y

// 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))
}
```

5.4. 将方法转换成函数（神奇的下划线）



5.5.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。

5，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。