Scala 进阶之路

1.课程目标

1.1. 目标 1：（初级）熟练使用 scala 编写 Spark 程序

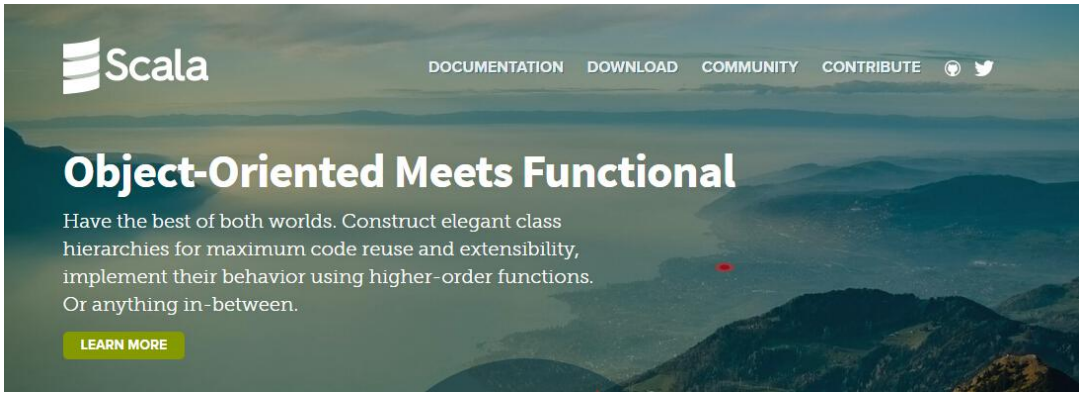
1.2. 目标 2：（中级）动手编写一个简易 Spark 通信框架

1.3. 目标 3：（高级）为阅读 Spark 内核源码做准备

2.Scala 介绍

2.1 什么是 Scala

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 运行于 Java 平台（Java 虚拟机），并兼容现有的 Java 程序。



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 是静态编译的，所以和 JRuby,Groovy 比起来速度会快很多。

3. 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。

3.开发环境准备

3.1 Scala SDK 安装

安全 Scala SDK 前，请确保已安装 JDK1.8+.

3.1.1 Window 下安装 Scala SDK

访问 Scala 官网 http://www.scala-lang.org/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，所以这里推荐 2.11.x 版本，下载 scala-2.11.8.msi 后点击下一步就可以了.

3.1.2 Linux 下安装 Scala SDK

下载 Scala 地址 http://downloads.typesafe.com/scala/2.11.8/scala-2.11.8.tgz 然后解压 Scala 到指定目录

tar -zxvf scala-2.11.8.tgz -C /usr/java

配置环境变量，将 scala 加入到 PATH 中

vi /etc/profile

export JAVA_HOME=/usr/java/jdk1.8.0_111

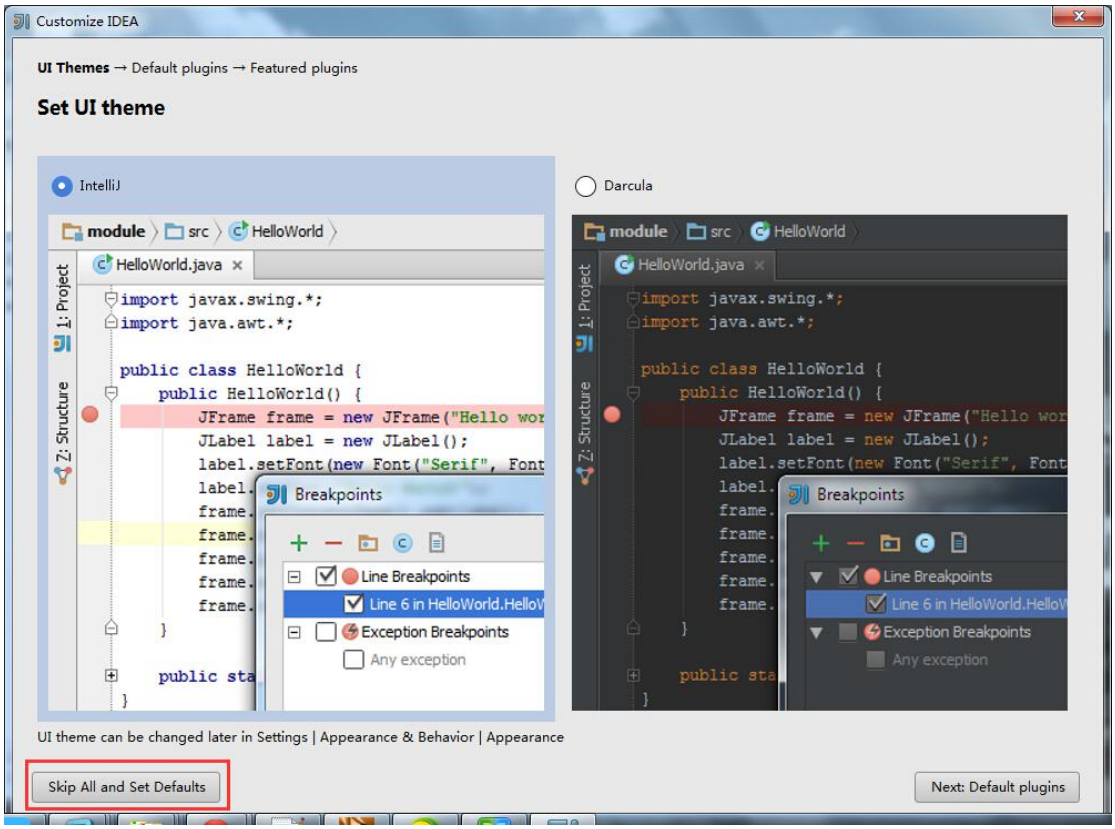
export PATH=$PATH:$JAVA_HOME/bin:/usr/java/scala-2.11.8/bin

3.2 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 Scala 官网下载即可 http://scala-ide.org/download/sdk.html。

由于 IDEA 的 Scala 插件更优秀，大多数 Scala 程序员都选择 IDEA，可以到 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 Defaul
2.下载 IEDA 的 scala 插件，地址 http://plugins.jetbrains.com/?idea_ce

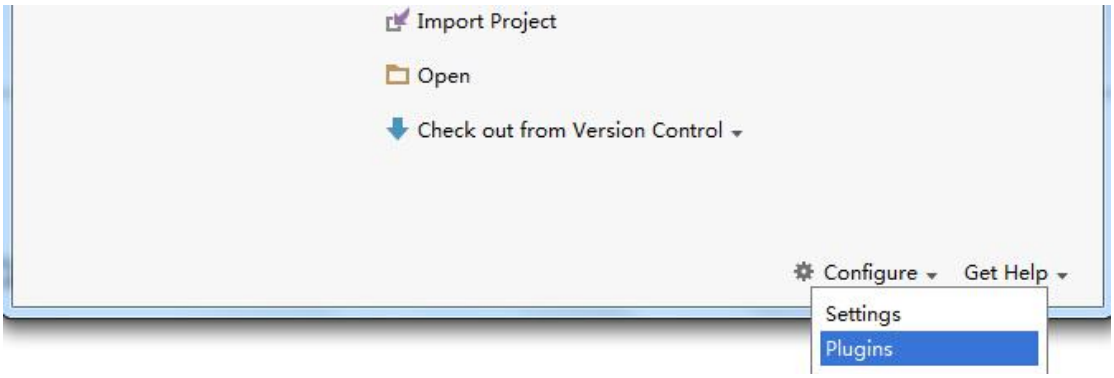


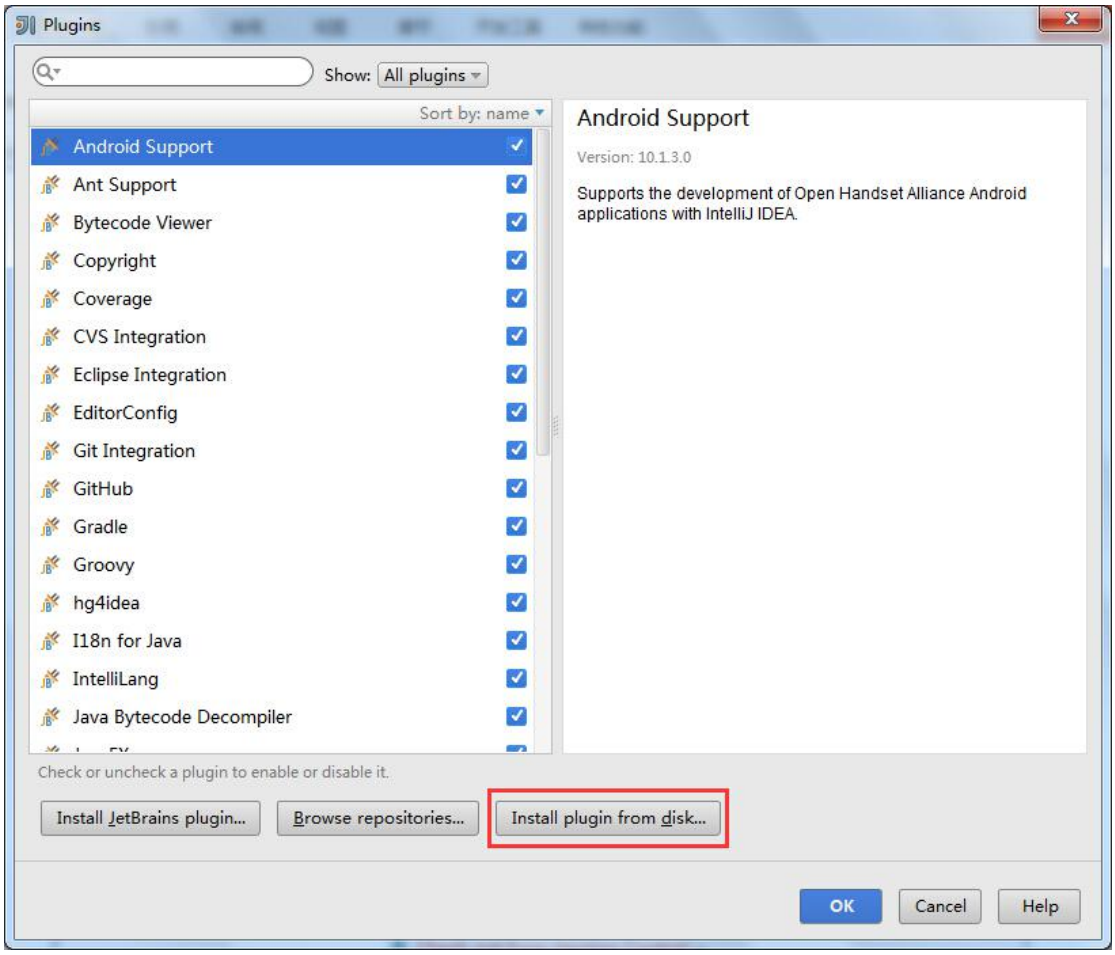
3.3 IDEA Scala 插件的离线安装

安装 Scala 插件：

Configure -> Plugins -> Install plugin from disk -> 选择 Scala 插件 -> OK -> 重启 IDEA

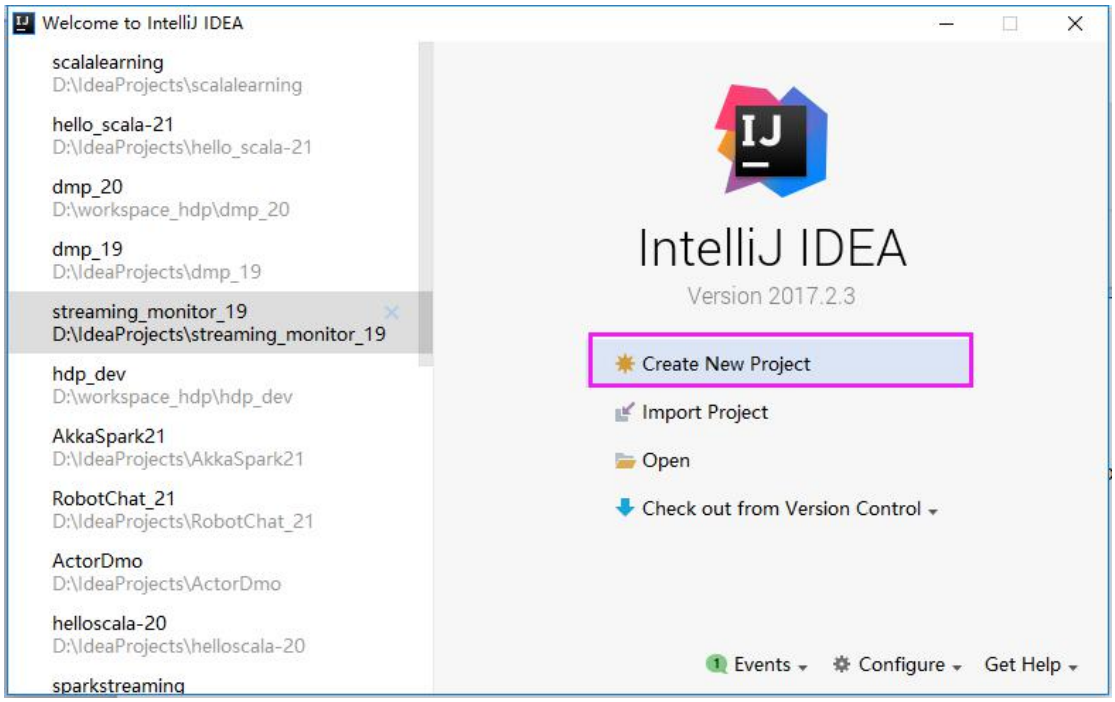

Welcome to IntelliJ IDEA
IntelliJ IDEA
Version 15.0
Create New Project
Import Project
Open
Check out from Version Control
Configure
Get Help


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

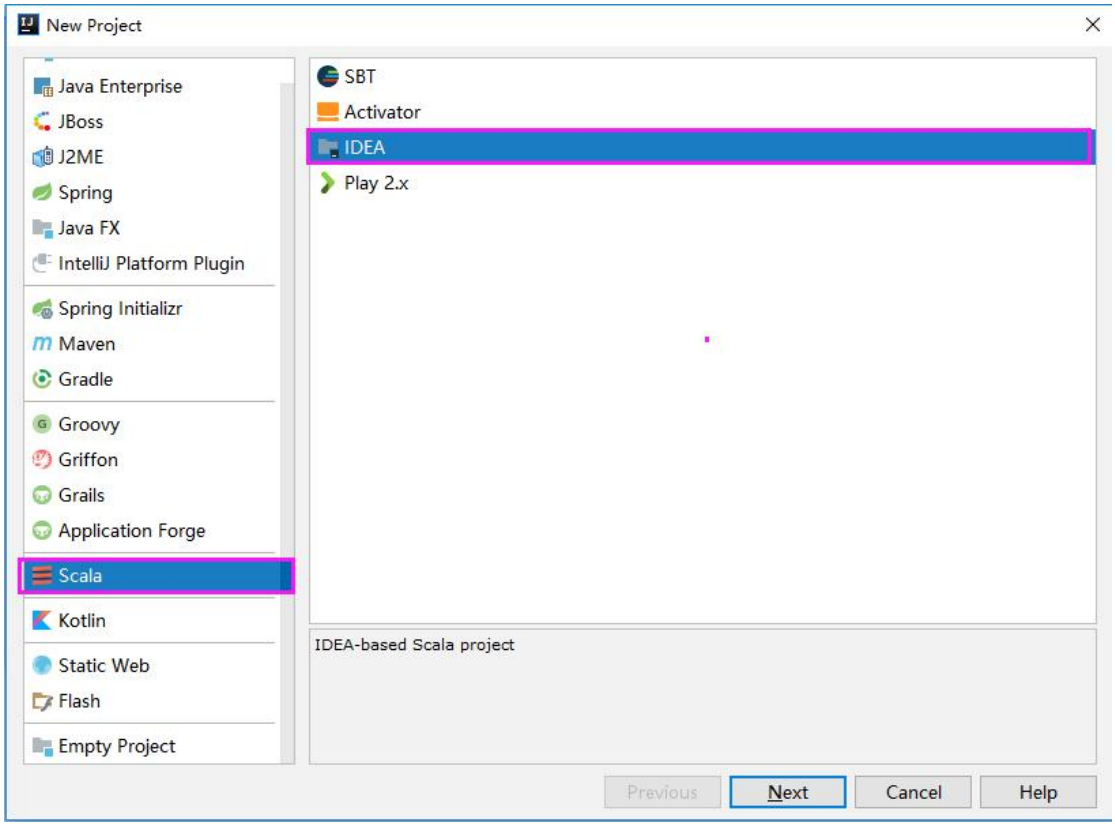


3.4 IDEA 创建 HelloScala 工程

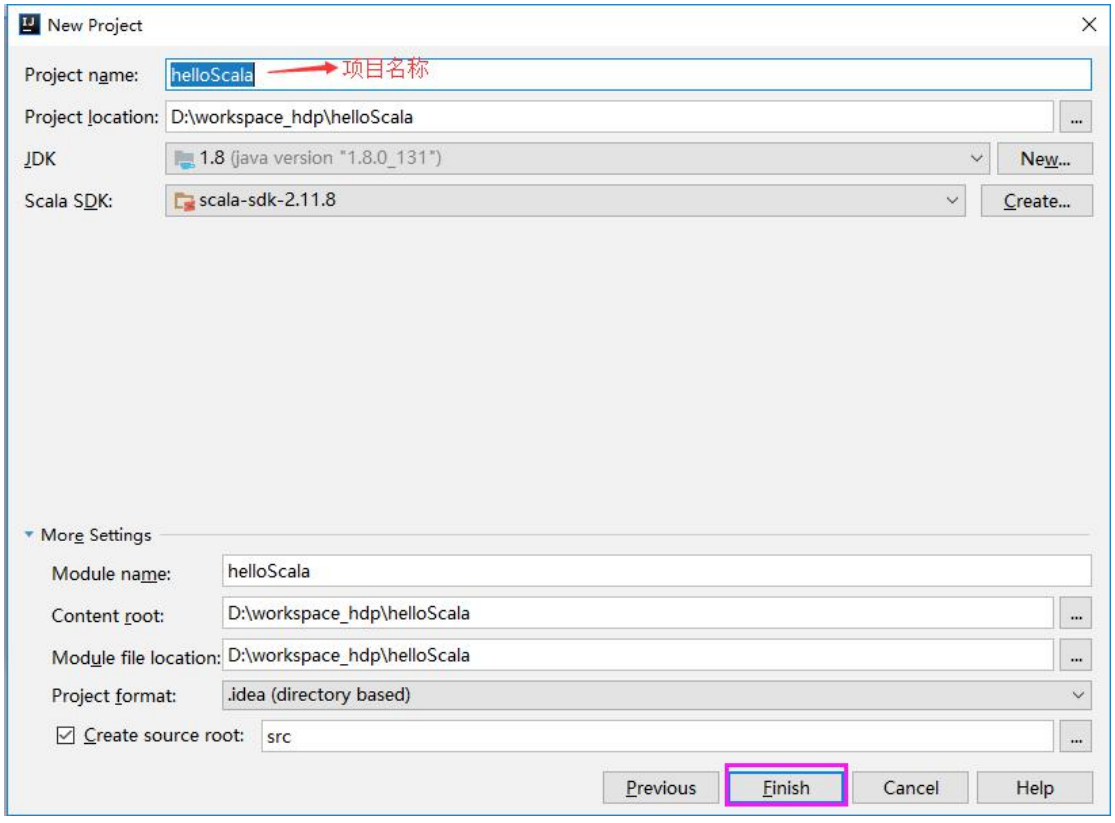
安装完成后，双击已打开 IDEA，创建一个新的项目(Create New Project)



选中左侧的 Scala -> IDEA -> Next



输入项目名称 -> 点击 Finish 完成即可



4.基本语法

4.1 函数式编程体验 Spark-Shell 之 WordCount

```
-rw-------. 1 root root      1124 Sep  7 17:40 anaconda-ks.cfg
drwxr-xr-x. 5 root root      4096 Sep 10 14:21 apps
-rw-r--r--. 1 root root 429928937 May 25 17:37 hadoop-2.8.0.tar.gz
drwxr-xr-x. 3 root root      4096 Sep  7 14:30 hdp-data
drwxr-xr-x. 3 root root      4096 Sep  7 14:48 hdptmp
-rw-r--r--. 1 root root      3161 Sep  7 17:39 install.log.syslog
-rw-r--r--. 1 root root 203728858 Aug 30 19:30 spark-2.2.0-bin-hadoop2.7.tgz
-rw-r--r--. 1 root root        41 Sep 10 14:45 xx.dat
[root@hdp-nd-01 ~]# more xx.dat
hello laoyang
hello angelababy
hello tom
[root@hdp-nd-01 ~]#
```

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
res4: Array[(String, Int)] = Array((tom,1), (angelababy,1), (hello,3), (laoyang,1))
```

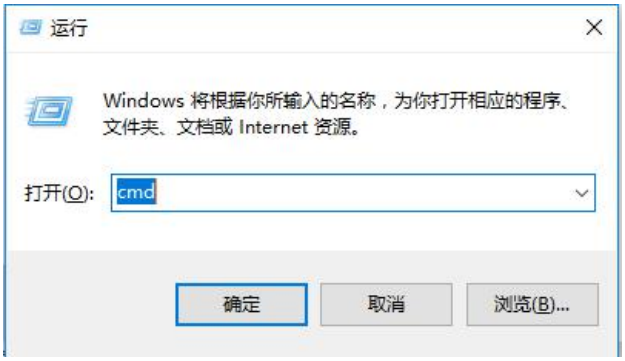
代码 → 输出结果

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
res13: Array[(String, Int)] = Array((hello,3), (tom,1), (angelababy,1), (laoyang,1))
```

4.2 Scala 交互模式(cmd 窗口介绍)

输入 cmd 命令，打开 windows 控制台:



输入 scala，进入 Scala shell 交互模式:

```
C:\WINDOWS\system32\cmd.exe - scala
Microsoft Windows [版本 10.0.15063]
(c) 2017 Microsoft Corporation。保留所有权利。

C:\Users\ThinkPad>scala
Welcome to Scala 2.11.8 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_131).
Type in expressions for evaluation. Or try :help.

scala>
```

在交互式窗口下，我们可以编写 Scala 的代码.

4.3 数据类型

Scala 和 Java 一样，有 7 种数值类型 Byte、Char、Short、Int、Long、Float 和 Double（无包装类型）和 Boolean、Unit 类型.

注意:Unit 表示无值，和其他语言中 void 等同。用作不返回任何结果的方法的结果类型。Unit 只有一个实例值，写成()。

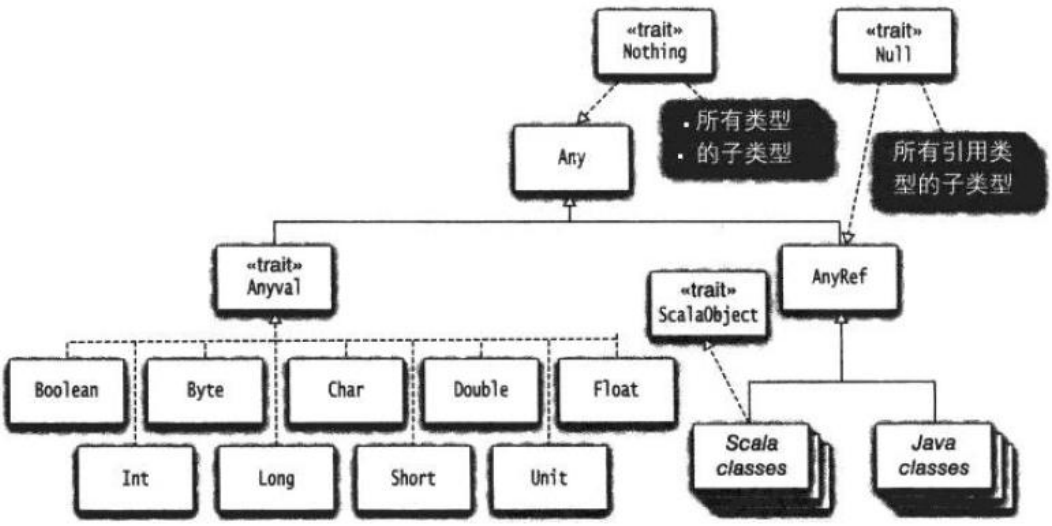


图8-1 Scala类的继承层级

4.4 变量的定义

```
object 变量定义 extends App {
    /**
      * 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 修饰的变量，值不能为修改，相当于 java 中 final 修饰的变量
    val name = "小牛人"

    // 使用 var 修饰的变量，值可以修改
    var age = 18

    // 定义变量时，可以指定数据类型，也可以不指定，不指定时编译器会自动推测变量的数据类型
    val nickName: String = "牛牛"
}
```

4.5 字符串的格式化输出

```
/**
  * Scala 中的格式化输出
  */
object ScalaPrint extends App{

    val name = "小牛学堂"
    val pirce  = 998.88d
    val url  = "www.edu360.com"

    // 普通输出
    println("name=" + name, "pirce=" + pirce, "url=" + url)

    // 文字'f'插值器允许创建一个格式化的字符串，类似于 C 语言中的 printf。
    // 在使用'f'插值器时，所有变量引用都应该是 printf 样式格式说明符，如%d，%i，%f 等。
    // 这里$name%s 打印 String 变量 James 和$height%2.2f 打印浮点值 1.90。
    println(f"$name%s 学费 $pirce%1.2f, 网址是$url")      // 该行输出有换行
    printf("%s 学费 %1.2f, 网址是%s", name, pirce, url)   // 该行输出没有换行


    // 's'允许在处理字符串时直接使用变量。
    //  在 println 语句中将 String 变量($name)附加到普通字符串中。
    println(s"name=$name, pirce=$pirce, url=$url")

    // 字符串插入器还可以处理任意表达式。
    // 使用's'字符串插入器处理具有任意表达式(${1 + 1})的字符串(1 + 1)的以下代码片段。
任何表达式都可以嵌入到${}中。
    println(s"1 + 1 = ${1 + 1}") // output: 1 + 1 = 2
}
```

4.6 条件表达式

```
/**
  * Scala if 条件表达式
  */
object ScalaIf extends App{

    //if 语句的使用
    val faceValue: Int = 99
    val res1 = if(faceValue > 90 ) "帅的不要不要的" else "瘪犊子玩意儿"
    println(res1)

    // 8 > 8 不成立，且代码没有 else 分支，res2 是什么呢
    val i = 8
    val res2 = if (i > 8) i
    println(res2)

    val res3 = if (i > 8) i else "前面是整型这里是字符串，那个 res3 是个啥?"
    println(res3)

    // if ... else if ... else 代码较多时可以使用代码块{}
    val score = 76
    val res4 = {
        if(score > 60 && score < 70) "及格"
        else if(score >=70 && score < 80) "良好"
        else "优秀"
    }
}
```

4.7 循环语句/yeild 关键字

在 scala 中有 for 循环和 while 循环，用 for 循环比较多

for 循环语法结构：for (i <- 表达式/数组/集合)

```
/**
  * Scala for 循环
  */
object ScalaFor extends App {

    // 定义个数组，元素为 1 到 6
    val array = Array(1, 2, 3, 4, 5, 6)
    // 遍历打印数组中的每个元素
    for(ele <- array) { // 类似 Java 中的增强 for
```

```
        println(ele)
    }

    // 通过角标获取数组中的元素
    // 定义一个 0 到 5 的角标范围
    for(i <- 0 to 5) { // 0 to 5 => 会生成一个范围集合 Range(0,1,2,3,4,5)
        println(array(i))
    }

    for(i <- 0 until 6) { // 0 until 6 => 会生成一个范围集合 Range(0,1,2,3,4,5)
        println(array(i))
    }

    // to 和 until 的区别就是 to 包含为前后都为闭区间, until 为前闭后开区间

    // 打印数组中的偶数元素
    for(e <- array if e % 2 == 0) { // for 表达式中可以增加守卫
        println(e)
    }

    // 观察下面代码输出结果
    for(i <- 1 to 3; j <- 1 to 3 if i != j) {
        println((10 * i + j) + " ")
    }

    for(e <- array if e % 2 == 0) yeild e
}
```

4.8 运算符/运算符重载

Scala 中的+ - * / %等操作符的作用与 Java 一样，位操作符 & | ^ >> <<也一样。只是有一点特别的：这些操作符实际上是方法。例如：

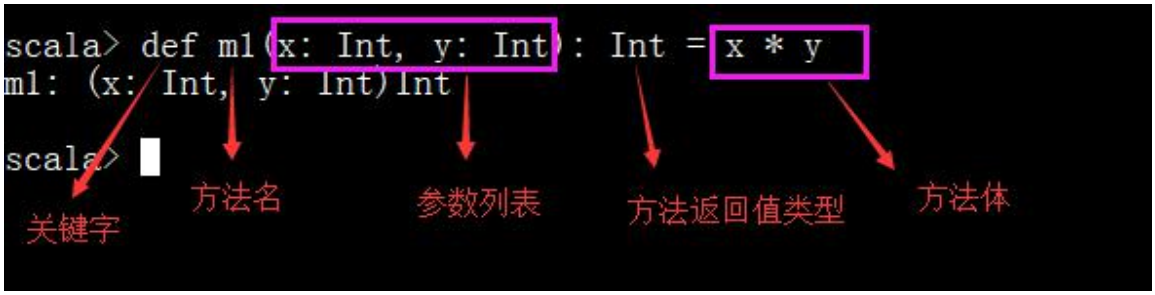
a + b

是如下方法调用的简写：

a.+(b)

a 方法 b 可以写成 a.方法(b)

4.9 方法的定义与调用



方法的返回值类型可以不写，编译器可以自动推断出来，但是对于递归函数，必须指定返回类型.

```
/**
  * 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，改方法有 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
    }
    // 调用
    val result1 = sum(1, 5)
    println(result1)

    // 改方法没有任何参数，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
}
```

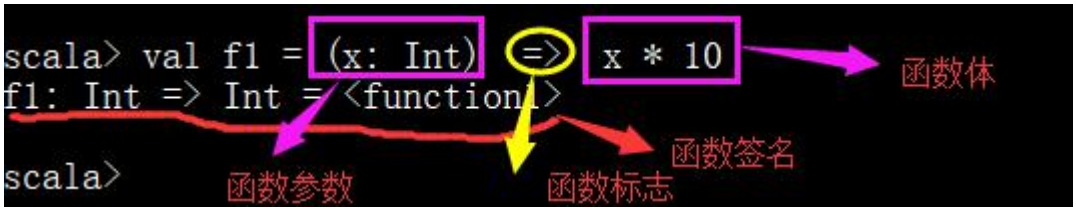
方法可转换为函数:

```
scala> def m(a: Int, b: Int): Int = a * b
m: (a: Int, b: Int)Int

scala> val fm = m_
<console>:11: error: not found: value m_
       val fm = m_


scala> val fm = m _
fm: (Int, Int) => Int = <function2>

scala> m(2,3)
res4: Int = 6

scala> fm(2,3)
res5: Int = 6

scala>
```

方法名空格 _

4.10 函数的定义与调用

函数定义方式一:



调用: f1(2)，其中 f1 为函数的引用，也可以叫做函数名. function1 表示一个参数的函数.

函数定义方式二:

```
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
```

下面为没有任何参数的函数，函数的返回值为 Int 类型.

```
scala> val f4:() => Int = () => 1
f4: () => Int = <function0>
```

4.11 传值调用与传名调用

通常，函数的参数是传值参数；也就是说，参数的值在传递给函数之前确定。
其实，在 Scala 中，我们方法或者函数的参数可以是一个表达式，也就是将一个代码逻辑传递给了某个方法或者函数.

```
/**
  * scala 的
  *     传名调用(call-by-name)
  *     传值调用(call-by-value)
  */
object ScalaCallName extends App{

    def currentTime(): Long ={
        println("打印系统当前时间,单位纳秒")
        System.nanoTime()
    }

    /**
      * 该方法的参数为一个无参的函数，并且函数的返回值为 Long
      */
    def delayed(f: => Long): Unit = {
        println("delayed ================")
        println("time = " + f)
    }

    def delayed1(time: Long) = {
        println("delayed1 ================")
        println("time1 = " + time)
    }

    // 调用方式一
    delayed(currentTime)

    println("------------华丽丽的分割线----------")

    // 调用方式二
    val time = currentTime()
    delayed1(time)
}
```

4.12 可变参数函数

```
// 定义一个可变参数的方法
def methodManyParams(a: String*) = {
    for (p <- a) {
        println(p)
    }
}
// 调用
methodManyParams("中华", "人民", "共和国")
```

4.13 默认参数值函数

```
// 带默认的参数列表
def add(a: Int = 1, b: Int = 7): Int = {
    println(s"a + b = ${a + b}" )
    a + b
}
// 调用
add(2)  //  带有默认值 a 的参数，调用时可以不传
add(b=9, a=2) // 调用时，可以指定具体的参数值
add(b=18) // 调用如果执行修改某一个具体的参数值的话，可以指定参数名称
```

4.14 高阶函数

```
// 高阶函数：将其他函数作为参数或其结果是函数的函数

// 定义一个方法，参数为带一个整型参数返回值为整型的函数 f 和 一个整型参数 v，返回值为一个函数
def apply(f: Int => String, v: Int) = f(v)

// 定义一个方法，参数为一个整型参数，返回值为 String
def layout(x: Int) = "[" + x.toString() + "]"

// 调用
println(apply(layout, 10))
```

4.15 部分参数应用函数

如果函数传递所有预期的参数，则表示已完全应用它。 如果只传递几个参数并不是全部参数，那么将返回部分应用的函数。这样就可以方便地绑定一些参数，其余的参数可稍后填写补上。

```
// 定义个输出的方法，参数为 date，message
def log(date: Date, message: String) = {
    println(s"$date, $message")
}

val date = new Date()

// 调用 log 的时候，传递了一个具体的时间参数，message 为待定参数
// logBoundDate 成了一个新的函数，只有 log 的部分参数(message)
val logBoundDate: (String) => Unit = log(date, _: String)

// 调用 logBoundDate 的时候，只需要传递待传的 message 参数即可
logBoundDate("fuck jerry ")
logBoundDate("fuck 涛涛 ")
```

上述，log()方法有两个参数：date 和 message。 我们想要多次调用该方法，具有相同的日期值，但不同的消息值。可以通过将参数部分地应用到 log()方法来消除将日期传递给每个调用的干扰。为此，首先将值绑定到 date 参数，并将第二个参数绑定到其位置。 结果是存储在变量中的部分应用函数。

4.16 柯里化(Currying)

柯里化(Currying)指的是将原来接受两个参数的函数变成新的接受一个参数的函数的过程。新的函数返回一个以原有第二个参数为参数的函数。

```
// 我们看下这个方法的定义，求 2 个数的和
def add(x: Int, y: Int) = x + y
// 那么我们应用的时候，应该是这样用：add(1,2)
// 现在我们把这个函数变一下形：
def add(x:Int)(y:Int) = x + y
// 那么我们应用的时候，应该是这样用：add(1)(2),最后结果都一样是 3，这种方式（过程）就叫柯里化。经过柯里化之后，函数的通用性有所降低，但是适用性有所提高。

// 分析下其演变过程
def add(x: Int) = (y: Int) => x + y

// (y: Int) => x + y 为一个匿名函数，也就意味着 add 方法的返回值为一个匿名函数
// 那么现在的调用过程为
```

```
val result = add(2)
val sum1 = result(3)
val sum2 = result(8)
```

4.17 偏函数

被包在花括号内没有 match 的一组 case 语句是一个偏函数，它是 PartialFunction[A, B]的一个实例，A 代表参数类型，B 代表返回类型，常用作输入模式匹配。

```
object PartialFuncDemo  {

  def func1: PartialFunction[String, Int] = {
    case "one" => 1
    case "two" => 2
    case _ => -1
  }

  def func2(num: String) : Int = num match {
    case "one" => 1
    case "two" => 2
    case _ => -1
  }

  def main(args: Array[String]) {
    println(func1("one"))
    println(func2("one"))
  }

}
```

4.18 数组的定义

```
// 数组的定义，定义一个固定长度的数组，长度可变，内容可变
var x:Array[String] = new Array[String](3)
// 或者
var y = new Array[String](3)

// 定义定长数组，长度不可变，内容可变
val z = Array(1,2,3)
// 修改第 0 个元素的内容
z(0) = 100
```

4.19 map|flatten|flatMap|foreach 方法的使用

```
// 定义一个数组
val array = Array[Int](2,4,6,9,3)
// map 方法是将 array 数组中的每个元素进行某种映射操作，(x: Int) => x * 2 为一个匿名函数，x 就是 array 中的每个元素
val y = array map((x: Int) => x * 2)
// 或者这样写，编译器会自动推测 x 的数据类型
val z = array.map(x => x*2)
// 亦或者，_ 表示入参，表示数组中的每个元素值
val x = array.map(_ * 2)

println(x.toBuffer)


println("--------骚气的分割线--------")

// 定义一个数组
val words = Array("hello tom hello jim hello jerry", "hello Hatano")

// 将数组中的每个元素进行分割
// Array(Array(hello, tom, hello, jim, hello, jerry), Array(hello, Hatano))
val splitWords: Array[Array[String]] = words.map(wd => wd.split(" "))

// 此时数组中的每个元素进过 split 之后变成了 Array，flatten 是对 splitWords 里面的元素进行扁平化操作
// Array(hello, tom, hello, jim, hello, jerry, hello, Hatano)
val flattenWords = splitWords.flatten


// 上述的 2 步操作，可以等价于 flatMap， 意味先 map 操作后进行 flatten 操作
val result: Array[String] = words.flatMap(wd => wd.split(" "))

// 遍历数组，打印每个元素
result.foreach(println)
```