Scala 编程

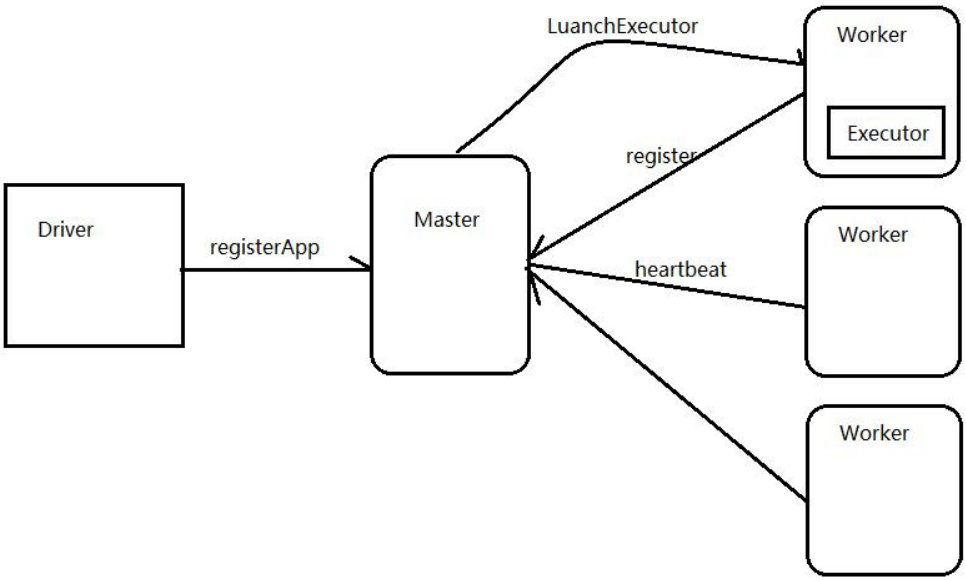
1. 课程目标

1.1. 目标 1：熟练使用 scala 编写 Spark 程序

```
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 2.0.2
      /_/

Using Scala version 2.11.8 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_144)
Type in expressions to have them evaluated.
Type :help for more information.

scala> sc.textFile("/words.txt").flatMap(_.split(" ")).map((_,1)).reduceByKey(_+_).collect
```

```
package itcast

import org.apache.spark.rdd.RDD
import org.apache.spark.{SparkConf, SparkContext}
object WordCount{

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

    val sparkConf: SparkConf = new SparkConf().setAppName("WordCount_Scala")
    val sc = new SparkContext(sparkConf)
    val dataRDD: RDD[String] = sc.textFile(args(0))
    val wordsRDD: RDD[String] = dataRDD.flatMap(_.split(" "))
    val wordAndOne: RDD[(String, Int)] = wordsRDD.map((_,1))
    val resultRDD: RDD[(String, Int)] = wordAndOne.reduceByKey(_+_)
    resultRDD.collect().foreach(println)

    sc.stop()

  }
}
```

1.2. 目标 2：动手编写一个简易版的 Spark 通信框架



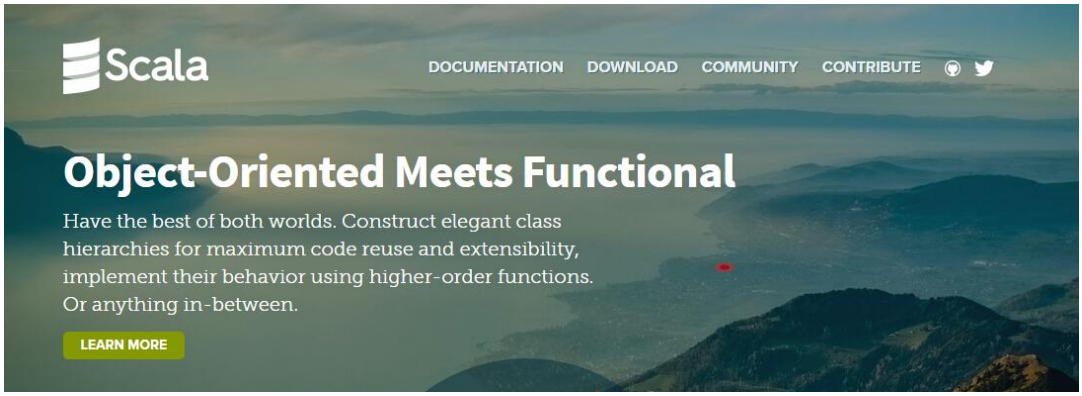
1.3. 目标 3：为阅读 Spark 内核源码做准备

```
private[deploy] object Master extends Logging {
  val SYSTEM_NAME = "sparkMaster"
  val ENDPOINT_NAME = "Master"

  def main(argStrings: Array[String]) {
    Utils.initDaemon(log)
    val conf = new SparkConf
    val args = new MasterArguments(argStrings, conf)
    val (rpcEnv, _, _) = startRpcEnvAndEndpoint(args.host, args.port, args.webUiPort, conf)
    rpcEnv.awaitTermination()
  }
```

2. Scala 概述

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。http://www.scala-lang.org



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. Scala 编译器安装

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。

3.2. 安装 Scala

3.2.1. Windows 安装 Scala 编译器

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，这里下载 scala-2.11.8.msi 后点击下一步就可以了（自动配置上环境变量）。也可以下载 scala-2.11.8.zip，解压后配置上环境变量就可以了。

3.2.2. Linux 安装 Scala 编译器

下载 Scala 地址 https://www.scala-lang.org/download/2.11.8.html

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

3.2.3. Scala 开发工具安装

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，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。

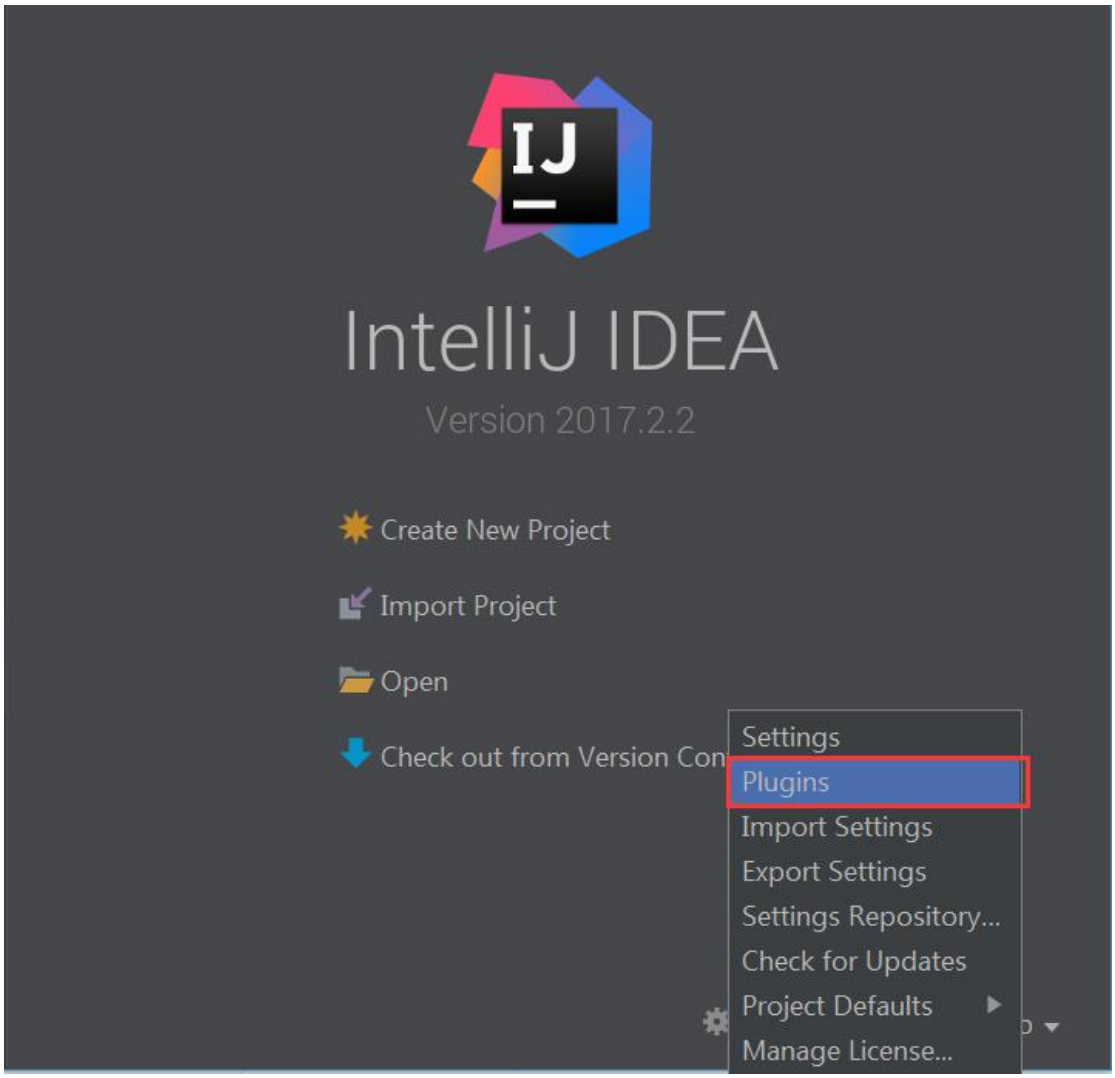
这里我们使用离线安装 Scala 插件：

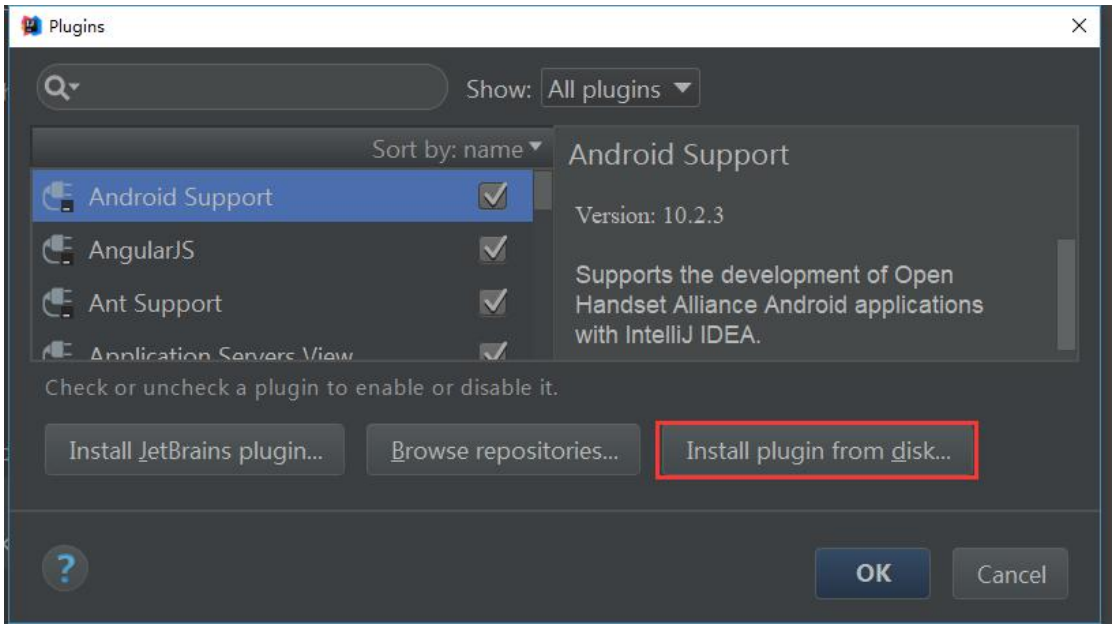
1. 安装 IDEA，点击下一步即可。

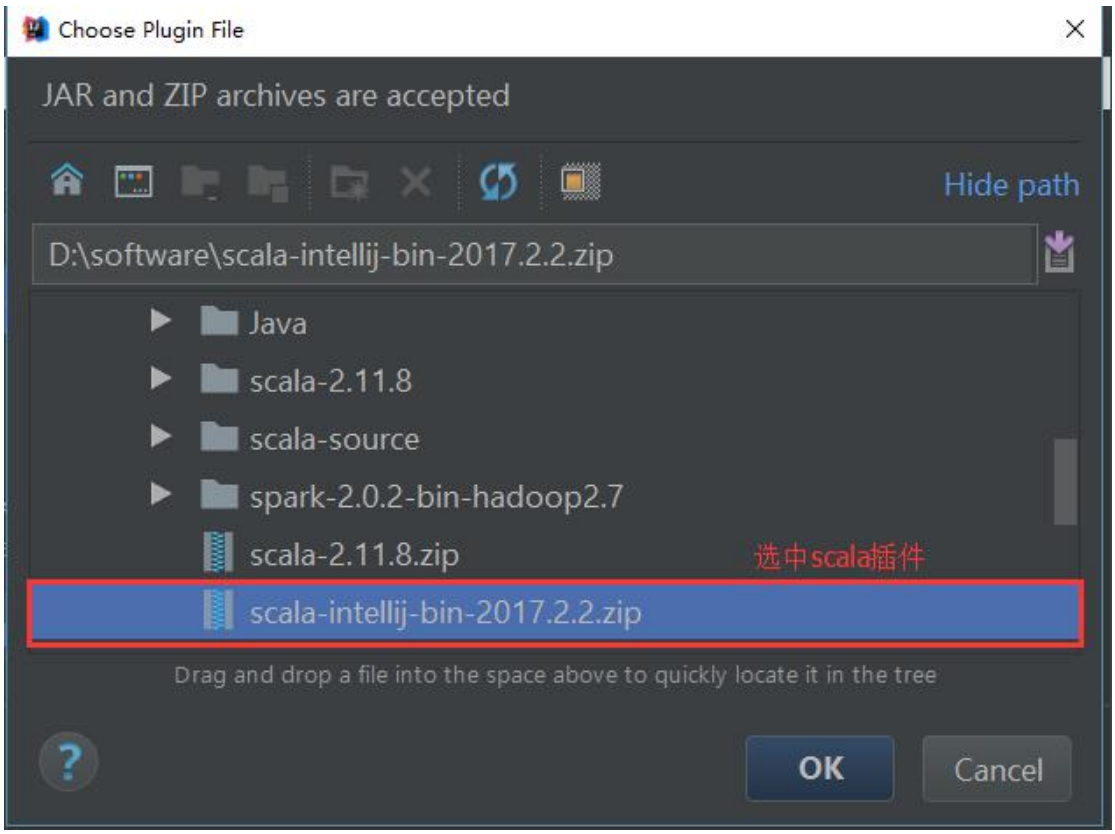
2. 下载 IEDA 的 scala 插件

插件地址： https://plugins.jetbrains.com/plugin/1347-scala

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


IntelliJ IDEA
Version 2017.2.2
Create New Project
Import Project
Open
Check out from Version Con
Settings
Plugins
Import Settings
Export Settings
Settings Repository...
Check for Updates
Project Defaults
Manage License...


Plugins
Show: All plugins
Sort by: name
Android Support
AngularJS
Ant Support
Application Servers View
Android Support
Version: 10.2.3
Supports the development of Open Handset Alliance Android applications with IntelliJ IDEA.
Check or uncheck a plugin to enable or disable it.
Install JetBrains plugin...
Browse repositories...
Install plugin from disk...
OK
Cancel



4. Scala 基础

4.1. 声明变量

```
package cn.itcast.scala
object VariableDemo {
  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
    val i = 1
    //使用 var 定义的变量是可变得，在 Scala 中鼓励使用 val
    var s = "hello"
    //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
    //变量名在前，类型在后
    val str: String = "itcast"
  }
}
```

4.2. 常用类型

Scala 和 Java 一样，有 7 种数值类型 Byte、Char、Short、Int、Long、Float、Double 类型和 1 个 Boolean 类型。

4.3. 条件表达式

Scala 的条件表达式比较简洁，定义变量时加上 if else 判断条件。例如：

```
package cn.itcast.scala

object ConditionDemo {
  def main(args: Array[String]) {
    val x = 1
    //判断 x 的值，将结果赋给 y
    val y = if (x > 0) 1 else -1
    //打印 y 的值
    println(y)

    //支持混合类型表达式
    val z = if (x > 1) 1 else "error"
    //打印 z 的值
    println(z)

    //如果缺失 else，相当于 if (x > 2) 1 else ()
    val m = if (x > 2) 1
    println(m)

    //在 scala 中每个表达式都有值，scala 中有个 Unit 类，用作不返回任何结果的方法的结果类型,相当于 Java 中的 void，Unit 只有一个实例值，写成()。
    val n = if (x > 2) 1 else ()
    println(n)

    //if 和 else if
    val k = if (x < 0) 0
    else if (x >= 1) 1 else -1
    println(k)
  }
}
```

4.4. 块表达式

定义变量时用 {} 包含一系列表达式，其中块的最后一个表达式的值就是块的值。

```
package cn.itcast.scala

object BlockExpressionDemo {
  def main(args: Array[String]) {
    val a = 10
    val b = 20
    //在 scala 中{}中包含一系列表达式，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 = {
                    val c=b-a
                    val d=b-c
                    d   //块中最后一个表达式的值
                }
                 //result 的值就是块表达式的结果
                  println(result)
        }
}
```

4.5. 循环

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)

```
package cn.itcast.scala
object ForDemo {
  def main(args: Array[String]) {
    //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 10)
      println(i)

    //for(i <- 数组)
    val arr = Array("a", "b", "c")
    for (i <- arr)
      println(i)
```

```
    //高级 for 循环
    //每个生成器都可以带一个条件，注意：if 前面没有分号
    for(i <- 1 to 3; j <- 1 to 3 if i != j)
      print((10 * i + j) + " ")
    println()

    //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)

  }

}
```

4.6. 调用方法和函数

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

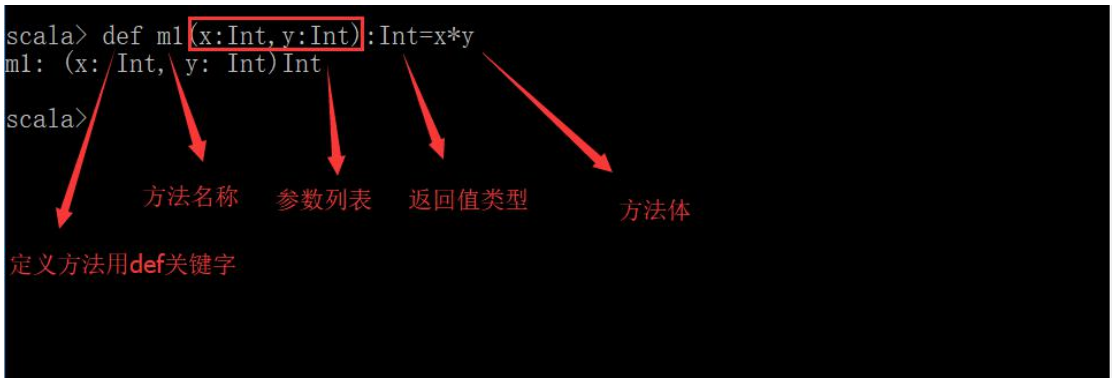
是如下方法调用的简写：

a.+(b)

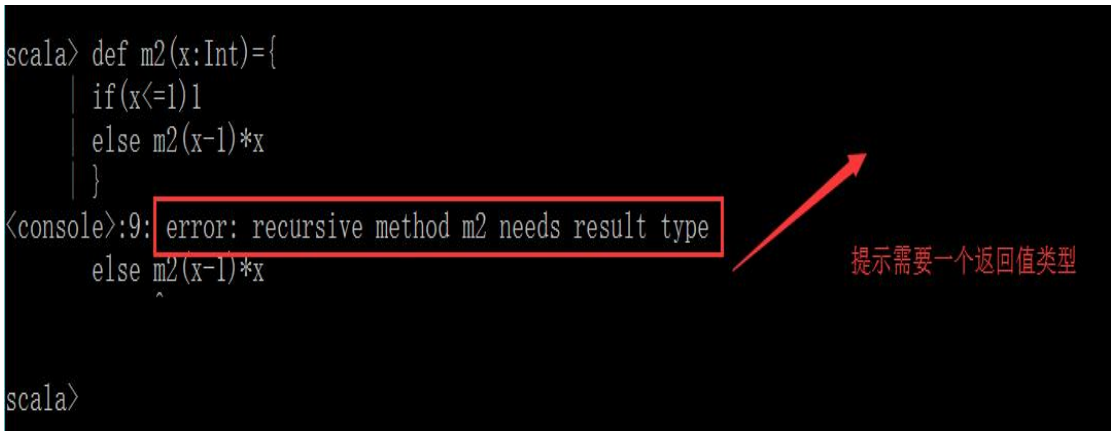
a 方法 b 可以写成 a.方法(b)

4.7. 定义方法和函数

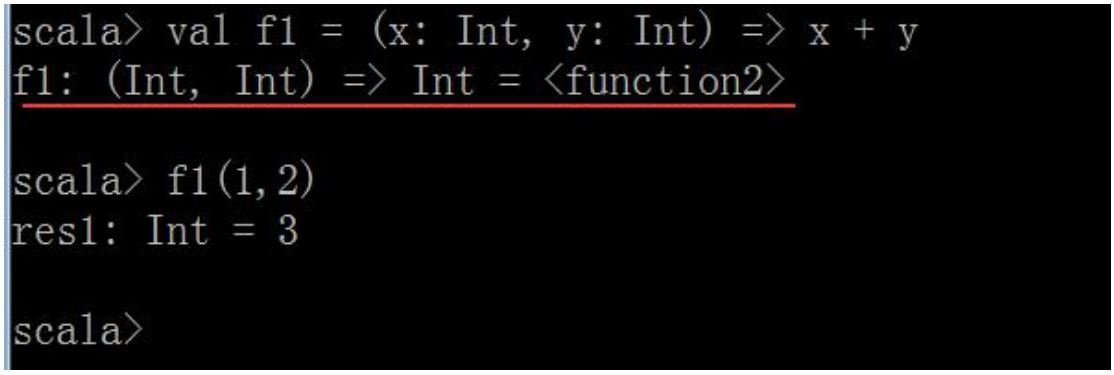
4.7.1. 定义方法



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



4.7.2. 定义函数



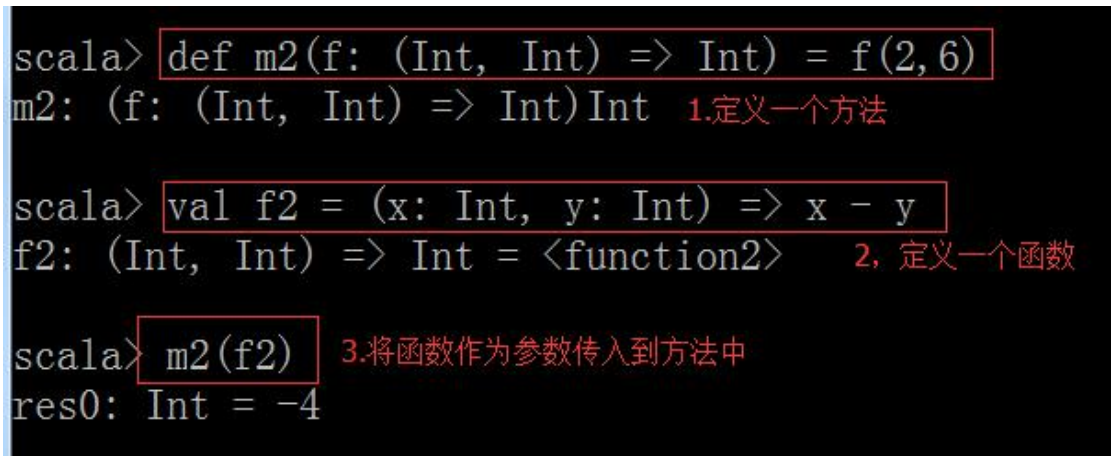
4.7.3. 方法和函数的区别

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，函数是一个对象，继承自 FuctionN。

函数对象有 apply、curried、toString、tupled 这些方法。而方法不具有这些特性。

如果想把方法转换成一个函数，可以用方法名跟上下划线的方式。

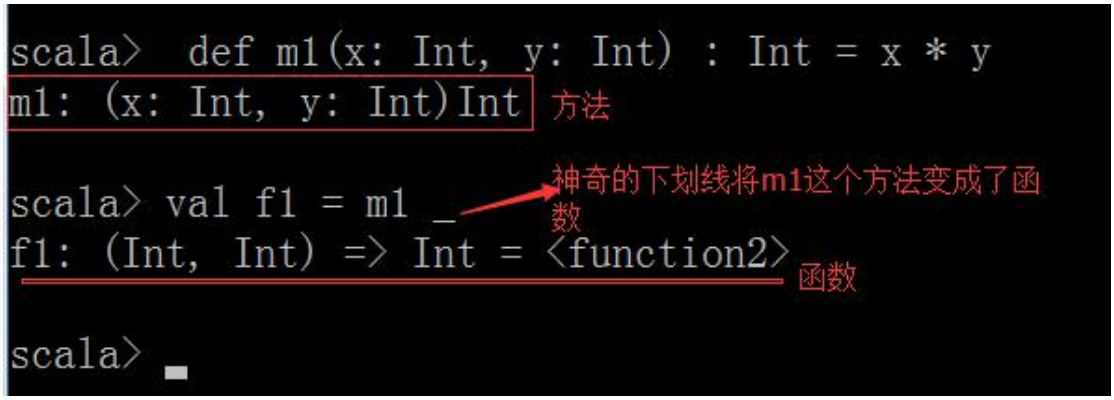
案例：首先定义一个方法，再定义一个函数，然后将函数传递到方法里面




```scala
package cn.itcast.scala

object MethodAndFunctionDemo {
  //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)
  }
}
```

4.7.4. 将方法转换成函数（神奇的下划线）

将方法转换成函数，只需要在方法的后面加上一个下划线



5. 数组、映射、元组、集合

5.1. 数组

5.1.1. 定长数组和变长数组

（1）定长数组定义格式：
val arr=new Array[T](数组长度)

（2）变长数组定义格式：
val arr = ArrayBuffer[T]()

注意需要导包：import scala.collection.mutable.ArrayBuffer

```
package cn.itcast.scala
import scala.collection.mutable.ArrayBuffer
object ArrayDemo {
  def main(args: Array[String]) {

    //初始化一个长度为 8 的定长数组，其所有元素均为 0
    val arr1 = new Array[Int](8)
    //直接打印定长数组，内容为数组的 hashcode 值
    println(arr1)
    //将数组转换成数组缓冲，就可以看到原数组中的内容了
    //toBuffer 会将数组转换长数组缓冲
```

```
    println(arr1.toBuffer)

    //注意：如果 new，相当于调用了数组的 apply 方法，直接为数组赋值
    //初始化一个长度为 1 的定长数组
    val arr2 = Array[Int](10)
    println(arr2.toBuffer)

    //定义一个长度为 3 的定长数组
    val arr3 = Array("hadoop", "storm", "spark")
    //使用()来访问元素
    println(arr3(2))

    //变长数组（数组缓冲）
    //如果想使用数组缓冲，需要导入 import scala.collection.mutable.ArrayBuffer
包
    val ab = ArrayBuffer[Int]()
    //向数组缓冲的尾部追加一个元素
    //+=尾部追加元素
    ab += 1
    //追加多个元素
    ab += (2, 3, 4, 5)
    //追加一个数组++=
    ab ++= Array(6, 7)
    //追加一个数组缓冲
    ab ++= ArrayBuffer(8,9)
    //打印数组缓冲 ab

    //在数组某个位置插入元素用 insert，从某下标插入
    ab.insert(0, -1, 0)
    //删除数组某个位置的元素用 remove  按照下标删除
    ab.remove(0)
    println(ab)

  }
}
```

5.1.2. 遍历数组

1.增强 for 循环
2.好用的 until 会生成脚标，0 until 10 包含 0 不包含 10

```
scala> 0 until 10
res0: scala.collection.immutable.Range = Range(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
```

```
package cn.itcast.scala
object ForArrayDemo {
  def main(args: Array[String]) {
    //初始化一个数组
    val arr = Array(1,2,3,4,5,6,7,8)
    //增强 for 循环
    for(i <- arr)
      println(i)

    //好用的 until 会生成一个 Range
    //reverse 是将前面生成的 Range 反转
    for(i <- (0 until arr.length).reverse)
      println(arr(i))
  }
}
```

5.1.3. 数组转换

yield 关键字将原始的数组进行转换会产生一个新的数组，原始的数组不变

```
scala> val arr = Array(1, 2, 3, 4, 5, 6, 7)  定义一个数组
arr: Array[Int] = Array(1, 2, 3, 4, 5, 6, 7)

scala> val res = for(e <- arr) yield e * 2  用yield关键字生成一个新的数组
res: Array[Int] = Array(2, 4, 6, 8, 10, 12, 14)

scala> arr.map(_ * 2)  map方法更加好用!
res1: Array[Int] = Array(2, 4, 6, 8, 10, 12, 14)
```

```
package cn.itcast.scala

object ArrayYieldDemo {
  def main(args: Array[String]) {
    //定义一个数组
    val arr = Array(1, 2, 3, 4, 5, 6, 7, 8, 9)
    //将偶数取出乘以 10 后再生成一个新的数组
    val res = for (e <- arr if e % 2 == 0) yield e * 10
    println(res.toBuffer)
```

```
    //更高级的写法,用着更爽
    //filter 是过滤，接收一个返回值为 boolean 的函数
    //map 相当于将数组中的每一个元素取出来，应用传进去的函数
    val r = arr.filter(_ % 2 == 0).map(_ * 10)
    println(r.toBuffer)

  }
}
```

5.1.4. 数组常用算法

在 Scala 中，数组上的某些方法对数组进行相应的操作非常方便！

```
scala> val arr = Array(2, 5, 1, 4, 3)
arr: Array[Int] = Array(2, 5, 1, 4, 3)

scala> arr.sum 求和
res8: Int = 15

scala> arr.max  求最大值
res9: Int = 5

scala> arr.sorted  排序
res10: Array[Int] = Array(1, 2, 3, 4, 5)
```

5.2. 映射

在 Scala 中，把哈希表这种数据结构叫做映射。

5.2.1. 构建映射

（1）构建映射格式

1、val map=Map(键 -> 值，键 -> 值....)

2、利用元组构建　val map=Map((键，值),(键，值),(键，值)....)

```
scala> val scores = Map("tom" -> 85, "jerry" -> 99, "kitty" -> 90)
scores: scala.collection.immutable.Map[String,Int] = Map(tom -> 85, jerry -> 99,
 kitty -> 90)    第一种创建Map的方式，用箭头

scala> val scores = Map(("tom", 85), ("jerry", 99), ("kitty", 90))
scores: scala.collection.immutable.Map[String,Int] = Map(tom -> 85, jerry -> 99,
 kitty -> 90)    第二种创建Map的方式，用元组
```

5.2.2. 获取和修改映射中的值

（1）获取映射中的值：

值=map(键)

```
scala> val scores = Map(("tom", 85), ("jerry", 99), ("kitty", 90))
scores: scala.collection.immutable.Map[String,Int] = Map(tom -> 85, jerry -> 99,
 kitty -> 90)

scala> scores("jerry")    获取映射中的值
res0: Int = 99
```

好用的 getOrElse

```
scala> scores.getOrElse("suke", 0)
res2: Int = 0 如果映射中有值，返回映射中的值，没有就返回默认值
```

注意：在 Scala 中，有两种 Map，一个是 immutable 包下的 Map，该 Map 中的内容不可变；另一个是 mutable 包下的 Map，该 Map 中的内容可变

例子：

```
scala> import scala.collection.mutable.Map 首先导入mutable包
import scala.collection.mutable.Map

scala> val scores = Map("tom" -> 80, "jerry" -> 99)  val定义的scores变量意味着变量的引用不变，但是Map中的内容可变
scores: scala.collection.mutable.Map[String,Int] = Map(tom -> 80, jerry -> 99)

scala> scores("tom") = 88    修改Map中的内容

scala> scores += ("kitty" -> 99, "suke" -> 60)    用+=向原来的Map中追加元素
res9: scores.type = Map(tom -> 88, kitty -> 99, jerry -> 99, suke -> 60)
```

注意：通常我们在创建一个集合是会用 val 这个关键字修饰一个变量（相当于 java 中的 final），那么就意味着该变量的引用不可变，该引用中的内容是不是可变，取决于这个引用指向的集合的类型

5.3. 元组

映射是 K/V 对偶的集合，对偶是元组的最简单形式，元组可以装着多个不同类型

的值。

5.3.1. 创建元组

（1）元组是不同类型的值的聚集；对偶是最简单的元组。
（2）元组表示通过将不同的值用小括号括起来，即表示元组。

创建元组格式：

val tuple=(元素,元素...)

```
scala> val t = ("hadoop", 3.14, 65535)
t: (String, Double, Int) = (hadoop,3.14,65535)

scala>
```

定义元组时用小括号将多个元素包起来，元素之间用逗号分隔
元素的类型可以不一样，元素个数可以任意多个

5.3.2. 获取元组中的值

(1) 获取元组中的值格式：

使用下划线加脚标 ，例如 t._1 t._2 t._3

注意：元组中的元素脚标是从 1 开始的

```
scala> val t=("hadoop",3.14,65535)
t: (String, Double, Int) = (hadoop,3.14,65535)

scala> t._1
res18: String = hadoop

scala> t._2
res19: Double = 3.14

scala> t._3
res20: Int = 65535
```

获取元组中的元素可以使用下划线加上脚标
但是需要注意的是元组中的元素脚标是从1开始的

5.3.3. 将对偶的集合转换成映射

将对偶的集合转换成映射：

调用其 toMap 方法

```
scala> val arr = Array(("tom", 88), ("jerry", 95))
arr: Array[(String, Int)] = Array((tom,88), (jerry,95))

scala> arr.toMap    toMap可以将对偶的集合转换成映射
res2: scala.collection.immutable.Map[String,Int] = Map(tom -> 88, jerry -> 95)
```

5.3.4. 拉链操作

1.使用 zip 命令可以将多个值绑定在一起

```
scala> val scores=Array(88,95,80)
scores: Array[Int] = Array(88, 95, 80)

scala> val names=Array("zhangsan","lisi","wangwu")
names: Array[String] = Array(zhangsan, lisi, wangwu)

scala> val scores=Array(88,95,80)
scores: Array[Int] = Array(88, 95, 80)   使用zip将对应的值进行捆绑

scala> names.zip(scores)
res8: Array[(String, Int)] = Array((zhangsan,88), (lisi,95), (wangwu,80))

scala>
```

注意：如果两个数组的元素个数不一致，拉链操作后生成的数组的长度为较小的那个数组的元素个数

2.如果其中一个元素的个数比较少，可以使用 zipAll 用默认的元素填充

```
scala> val names=Array("zhangsan","lisi","wangwu")
names: Array[String] = Array(zhangsan, lisi, wangwu)

scala> val scores=Array(88,95)
scores: Array[Int] = Array(88, 95)

scala> names.zipAll(scores,"zhaoliu",100)
res12: Array[(String, Int)] = Array((zhangsan,88), (lisi,95), (wangwu,100))
```

对names数组后进行元素填充

对scores数组后进行元素填充

5.4. 集合

Scala 的集合有三大类：序列 Seq、Set、映射 Map，所有的集合都扩展自 Iterable 特质，在 Scala 中集合有可变（mutable）和不可变（immutable）两种类型，immutable 类型的集合初始化后就不能改变了（注意与 val 修饰的变量进行区别）。

5.4.1. List

(1)不可变的序列 import scala.collection.immutable._

在 Scala 中列表要么为空（Nil 表示空列表） 要么是一个 head 元素加上一个 tail 列表。
9 :: List(5, 2) :: 操作符是将给定的头和尾创建一个新的列表
注意：:: 操作符是右结合的，如 9 :: 5 :: 2 :: Nil 相当于 9 :: (5 :: (2 :: Nil))

list 常用的操作符：

+: (elem: A): List[A] 在列表的头部添加一个元素

:: (x: A): List[A] 在列表的头部添加一个元素

:+ (elem: A): List[A] 在列表的尾部添加一个元素

++[B](that: GenTraversableOnce[B]): List[B] 从列表的尾部添加另外一个列表

::: (prefix: List[A]): List[A] 在列表的头部添加另外一个列表

```
val left = List(1,2,3)

val right = List(4,5,6)

//以下操作等价

left ++ right          // List(1,2,3,4,5,6)

right.:::(left)        // List(1,2,3,4,5,6)

//以下操作等价

0 +: left       //List(0,1,2,3)

left.+:(0)      //List(0,1,2,3)

//以下操作等价

left :+ 4       //List(1,2,3,4)

left.:+(4)      //List(1,2,3,4)

//以下操作等价

0 :: left          //List(0,1,2,3)

left.::(0)         //List(0,1,2,3)
```

例子:

```
package cn.itcast.collect
/**
 * 不可变List集合操作
 */
```

```
object ImmutListDemo {
  def main(args: Array[String]) {
    //创建一个不可变的集合
    val lst1 = List(1,2,3)
    //补充：另一种定义 list 方法
    val other_lst=2::Nil
    //获取集合的第一个元素
    val first=lst1.head
    //获取集合中除第一个元素外的其他元素集合，
    val tail=lst1.tail
    //补充：其中如果 List 中只有一个元素，那么它的 head 就是这个元素，它的 tail 就是 Nil;
    println(other_lst.head+"----"+other_lst.tail)
    //将 0 插入到 lst1 的前面生成一个新的 List
    val lst2 = 0 :: lst1
    val lst3 = lst1.::(0)
    val lst4 = 0 +: lst1
    val lst5 = lst1.+:(0)

//将一个元素添加到 lst1 的后面产生一个新的集合
    val lst6 = lst1 :+ 3
    val lst0 = List(4,5,6)
    //将 2 个 list 合并成一个新的 List
    val lst7 = lst1 ++ lst0
    //将 lst0 插入到 lst1 前面生成一个新的集合
    val lst8 = lst1 ++: lst0
    //将 lst0 插入到 lst1 前面生成一个新的集合
    val lst9 = lst1.:::(lst0)

    println(other_lst)
    println(lst1)
    println(first)
    println(tail)
    println(lst2)
    println(lst3)
    println(lst4)
    println(lst5)
    println(lst6)
    println(lst7)
    println(lst8)
    println(lst9)
  }
}
```

(2)可变的序列 import scala.collection.mutable._

```
ackage cn.itcast.collect
import scala.collection.mutable.ListBuffer

object MutListDemo extends App{
  //构建一个可变列表，初始有 3 个元素 1,2,3
  val lst0 = ListBuffer[Int](1,2,3)
  //创建一个空的可变列表
  val lst1 = new ListBuffer[Int]
  //向 lst1 中追加元素，注意：没有生成新的集合
  lst1 += 4
  lst1.append(5)

  //将 lst1 中的元素最近到 lst0 中， 注意：没有生成新的集合
  lst0 ++= lst1

  //将 lst0 和 lst1 合并成一个新的 ListBuffer 注意：生成了一个集合
  val lst2= lst0 ++ lst1

  //将元素追加到 lst0 的后面生成一个新的集合
  val lst3 = lst0 :+ 5

  //删除元素,注意：没有生成新的集合
  val lst4 = ListBuffer[Int](1,2,3,4,5)
  lst4 -= 5

  //删除一个集合列表,生成了一个新的集合
  val lst5=lst4--List(1,2)

  //把可变 list 转换成不可变的 list 直接加上 toList
  val lst6=lst5.toList

  //把可变 list 转变数组用 toArray
  val lst7=lst5.toArray

  println(lst0)
  println(lst1)
  println(lst2)
  println(lst3)
  println(lst4)
  println(lst5)
  println(lst6)
  println(lst7)
```

```
}
```

5.4.2. Set

(1)不可变的 Set　　import scala.collection.immutable._

Set 代表一个没有重复元素的集合；将重复元素加入 Set 是没有用的，而且 Set 是不保证插入顺序的，即 Set 中的元素是乱序的。

定义：val set=Set(元素,元素,.....)

```
//定义一个不可变的 Set 集合
scala> val set =Set(1,2,3,4,5,6,7)
set: scala.collection.immutable.Set[Int] = Set(5, 1, 6, 2, 7, 3, 4)

//元素个数
scala> set.size
res0: Int = 7

//取集合最小值
scala> set.min
res1: Int = 1

//取集合最大值
scala> set.max
res2: Int = 7

//将元素和 set1 合并生成一个新的 set，原有 set 不变
scala> set + 8
res3: scala.collection.immutable.Set[Int] = Set(5, 1, 6, 2, 7, 3, 8, 4)

scala> val set1=Set(7,8,9)
set1: scala.collection.immutable.Set[Int] = Set(7, 8, 9)

//两个集合的交集
scala> set & set1
res4: scala.collection.immutable.Set[Int] = Set(7)

//两个集合的并集
scala> set ++ set1
res5: scala.collection.immutable.Set[Int] = Set(5, 1, 6, 9, 2, 7, 3, 8, 4)

//在第一个 set 基础上去掉第二个 set 中存在的元素
```

```
scala> set -- set1
res6: scala.collection.immutable.Set[Int] = Set(5, 1, 6, 2, 3, 4)

//返回第一个不同于第二个 set 的元素集合
scala> set &~ set1
res7: scala.collection.immutable.Set[Int] = Set(5, 1, 6, 2, 3, 4)


//计算符合条件的元素个数
scala> set.count(_ >5)
res8: Int = 2

/返回第一个不同于第二个的元素集合
scala> set.diff(set1)
res9: scala.collection.immutable.Set[Int] = Set(5, 1, 6, 2, 3, 4)

/返回第一个不同于第二个的元素集合
scala> set1.diff(set)
res10: scala.collection.immutable.Set[Int] = Set(8, 9)

//取子 set(2,5 为元素位置，从 0 开始，包含头不包含尾)
scala> set.slice(2,5)
res11: scala.collection.immutable.Set[Int] = Set(6, 2, 7)

//迭代所有的子 set，取指定的个数组合
scala> set1.subsets(2).foreach(x=>println(x))
Set(7, 8)
Set(7, 9)
Set(8, 9)
```

(2)可变的 Set　import scala.collection.mutable._

```
//导入包
scala> import scala.collection.mutable
import scala.collection.mutable
//定义一个可变的 Set
scala> val set1=new HashSet[Int]()
set1: scala.collection.mutable.HashSet[Int] = Set()

//添加元素
scala> set1 += 1
```

```
res1: set1.type = Set(1)

//添加元素  add 等价于+=
scala> set1.add(2)
res2: Boolean = true
scala> set1
res3: scala.collection.mutable.HashSet[Int] = Set(1, 2)

//向集合中添加元素集合
scala> set1 ++=Set(1,4,5)
res5: set1.type = Set(1, 5, 2, 4)

//删除一个元素
scala> set1 -=5
res6: set1.type = Set(1, 2, 4)

//删除一个元素
scala> set1.remove(1)
res7: Boolean = true
scala> set1
res8: scala.collection.mutable.HashSet[Int] = Set(2, 4)
```

5.4.3. Map

(1)不可变的 Map import scala.collection.immutable._

定义 Map 集合

1.val map=Map(键 -> 值 , 键 -> 值...)

2.利用元组构建 val map=Map((键，值), (键，值) , (键，值)....)

展现形式：

```
val  map = Map(“zhangsan”->30,”lisi”->40)
val  map = Map((“zhangsan”,30),(“lisi”,40))
```

3.操作 map 集合

获取值： 值=map(键)

原则：通过先获取键，在获取键对应值。

4.遍历 map 集合

```
scala> val imap=Map("zhangsan" -> 20,"lisi" ->30)
imap: scala.collection.immutable.Map[String,Int] = Map(zhangsan -> 20, lisi -> 30)
//方法一：显示所有的 key
scala> imap.keys
```

```
res0: Iterable[String] = Set(zhangsan, lisi)

//方法二：显示所有的 key
scala> imap.keySet
res1: scala.collection.immutable.Set[String] = Set(zhangsan, lisi)

//通过 key 获取 value
scala> imap("lisi")
res2: Int = 30

//通过 key 获取 value  有 key 对应的值则返回，没有就返回默认值 0，
scala> imap.getOrElse("zhangsan",0)
res4: Int = 20

//没有对应的 key，返回默认 0
scala> imap.getOrElse("zhangsan1",0)
res5: Int = 0
//由于是不可变 map,故不能向其添加、删除、修改键值对
```

(2)可变的 Map import scala.collection.mutable._

```
//导包
import scala.collection.mutable
//声明一个可变集合
scala> val user =mutable.HashMap("zhangsan"->50,"lisi" -> 100)
user: scala.collection.mutable.HashMap[String,Int] = Map(lisi -> 100, zhangsan ->
50)

//添加键值对
scala> user +=("wangwu" -> 30)
res0: user.type = Map(lisi -> 100, zhangsan -> 50, wangwu -> 30)

//添加多个键值对
scala> user += ("zhangsan0" -> 30,"lisi0" -> 20)
res1: user.type = Map(zhangsan0 -> 30, lisi -> 100, zhangsan -> 50, lisi0 -> 20,wangwu
-> 30)

//方法一：显示所有的 key
scala> user.keys
res2: Iterable[String] = Set(zhangsan0, lisi, zhangsan, lisi0, wangwu)

//方法二：显示所有的 key
scala> user.keySet
```

```
res3: scala.collection.Set[String] = Set(zhangsan0, lisi, zhangsan, lisi0, wangwu)

//通过 key 获取 value
scala> user("zhangsan")
res4: Int = 50

//通过 key 获取 value 有 key 对应的值则返回，没有就返回默认值 0，
scala> user.getOrElse("zhangsan",0)
res5: Int = 50

//没有对应的 key，返回默认 0
scala> user.getOrElse("zhangsan1",0)
res6: Int = 0

//更新键值对
scala> user("zhangsan") = 55
scala> user("zhangsan")
res8: Int = 55

//更新多个键值对
scala> user += ("zhangsan" -> 60, "lisi" -> 50)
res9: user.type = Map(zhangsan0 -> 30, lisi -> 50, zhangsan -> 60, lisi0 -> 20,wangwu
-> 30)

//删除 key
scala> user -=("zhangsan")
res14: user.type = Map(zhangsan0 -> 30, lisi -> 50, lisi0 -> 20, wangwu -> 30)

//删除 key
scala>user.remove("zhangsan0")

//遍历 map 方法一：通过 key 值
scala> for(x<- user.keys) println(x+" -> "+user(x))
lisi -> 50
lisi0 -> 20
wangwu -> 30

//遍历 map 方法二：模式匹配
scala> for((x,y) <- user) println(x+" -> "+y)
lisi -> 50
lisi0 -> 20
wangwu -> 30

//遍历 map 方法三：通过 foreach
```

```
scala>  user.foreach{case (x,y) => println(x+" -> "+y)}
lisi -> 50
lisi0 -> 20
wangwu -> 30
```

6. 类、对象、继承、特质

Scala 的类与 Java、C++的类比起来更简洁，学完之后你会更爱 Scala！！！

6.1. 类

6.1.1. 类的定义

```
package cn.itcast.class_demo

/**
* 在Scala 中，类并不用声明为public 类型的。
* Scala 源文件中可以包含多个类，所有这些类都具有共有可见性。
*/
class Person {
  //用 val 修饰的变量是可读属性，有 getter 但没有 setter（相当与 Java 中用 final 修饰的变量）
  val id="9527"

  //用 var 修饰的变量都既有 getter，又有 setter
  var age:Int=18

  //类私有字段，只能在类的内部使用或者伴生对象中访问
  private var name : String = "唐伯虎"

  //类私有字段，访问权限更加严格的，该字段在当前类中被访问
  //在伴生对象里面也不可以访问
  private[this] var pet = "小强"

}

//伴生对象（这个名字和类名相同，叫伴生对象）
object Person{
  def main(args: Array[String]): Unit = {
```

```
    val p=new Person

    //如果是下面的修改，发现下面有红线，说明 val 类型的不支持重新赋值，但是可以获取到值
    //p.id = "123"
    println(p.id)
    //打印 age
    println(p.age)
    //打印 name,伴生对象中可以在访问 private 变量
    println(p.name)
    //由于 pet 字段用 private[this]修饰，伴生对象中访问不到 pet 变量
    //p.pet(访问不到)

  }
}
```

6.1.2. 构造器

Scala 中的每个类都有主构造器，主构造器的参数直接放置类名后面，与类交织在一起。
注意：主构造器会执行类定义中的所有语句。

```
package cn.itcast.class_demo

/**
  *每个类都有主构造器，主构造器的参数直接放置类名后面，与类交织在一起
  */
class Student(val name:String,var age:Int) {
  //主构造器会执行类定义的所有语句
  println("执行主构造器")
  private  var gender="male"
  def this(name:String,age:Int,gender:String){
    //每个辅助构造器执行必须以主构造器或者其他辅助构造器的调用开始
    this(name,age)
    println("执行辅助构造器")
    this.gender=gender
  }
}

object Student {
  def main(args: Array[String]): Unit = {
    val s1=new Student("zhangsan",20)

    val s2=new Student("zhangsan",20,"female")
```

```
  }
}
```

6.2. Scala 面向对象编程之对象

6.2.1. Scala 中的 object

➢ object 相当于 class 的单个实例，通常在里面放一些静态的 field 或者 method；
在 Scala 中没有静态方法和静态字段，但是可以使用 object 这个语法结构来达到同样的目的。

object 作用：

1.存放工具方法和常量

2.高效共享单个不可变的实例

3. 单例模式

➢ 举例说明：

```
package cn.itcast.object_demo
import scala.collection.mutable.ArrayBuffer

class Session{}
object SessionFactory{
  //该部分相当于 java 中的静态块
   val session=new Session
  //在 object 中的方法相当于 java 中的静态方法
  def getSession(): Session ={
    session
  }
}
object SingletonDemo {
  def main(args: Array[String]) {
    //单例对象，不需要 new，用【单例对象名称.方法】调用对象中的方法
    val session1 = SessionFactory.getSession()
    println(session1)
    //单例对象，不需要 new，用【单例对象名称.变量】调用对象中成员变量
    val session2=SessionFactory.session
    println(session2)
  }
}
```

6.2.2. Scala 中的伴生对象

- 如果有一个 class 文件，还有一个与 class 同名的 object 文件，那么就称这个 object 是 class 的伴生对象，class 是 object 的伴生类；
- 伴生类和伴生对象必须存放在一个.scala 文件中；
- 伴生类和伴生对象的最大特点是，可以相互访问；
- 举例说明：

```
package cn.itcast.object_demo

//伴生类
class Dog {
  val id = 1
  private var name = "itcast"
  def printName(): Unit ={
    //在 Dog 类中可以访问伴生对象 Dog 的私有属性
    println(Dog.CONSTANT + name )
  }
}
//伴生对象
object Dog {
  //伴生对象中的私有属性
  private val CONSTANT = "汪汪汪 : "
  def main(args: Array[String]) {
    val p = new Dog
    //访问私有的字段 name
    p.name = "123"
    p.printName()
  }
}
//执行结果 汪汪汪 : 123
```

6.2.3. Scala 中的 apply 方法

- object 中非常重要的一个特殊方法，就是 apply 方法；
- apply 方法通常是在伴生对象中实现的，其目的是，通过伴生类的构造函数功能，来实现伴生对象的构造函数功能；
- 通常我们会在类的伴生对象中定义 apply 方法，当遇到类名(参数 1,...参数 n)时 apply 方法会被调用；
- 在创建伴生对象或伴生类的对象时，通常不会使用 new class/class() 的方式，而是直接使用 class()，隐式的调用伴生对象的 apply 方法，这样会让对象创建的更加简洁；
- 举例说明：

```
package cn.itcast.object_demo
/**
 *  Array 类的伴生对象中，就实现了可接收变长参数的 apply 方法，
 * 并通过创建一个 Array 类的实例化对象，实现了伴生对象的构造函数功能
 */
// 指定 T 泛型的数据类型，并使用变长参数 xs 接收传参，返回 Array[T] 数组
// 通过 new 关键字创建 xs.length 长的 Array 数组
// 其实就是调用 Array 伴生类的 constructor 进行 Array 对象的初始化
//  def apply[T: ClassTag](xs: T*): Array[T] = {
//    val array = new Array[T](xs.length)
//    var i = 0
//    for (x <- xs.iterator) { array(i) = x; i += 1 }
//    array
//  }

object ApplyDemo {
  def main(args: Array[String]) {
    //调用了 Array 伴生对象的 apply 方法
    //def apply(x: Int, xs: Int*): Array[Int]
    //arr1 中只有一个元素 5
    val arr1 = Array(5)
    //new 了一个长度为 5 的 array，数组里面包含 5 个 null
    var arr2 = new Array(5)
    println(arr1.toBuffer)
  }
}
```

6.2.4. Scala 中的 main 方法

- 同 Java 一样，如果要运行一个程序，必须要编写一个包含 main 方法的类；
- 在 Scala 中，也必须要有一个 main 方法，作为入口；
- Scala 中的 main 方法定义为 def main(args: Array[String])，而且必须定义在 object 中；
- 除了自己实现 main 方法之外，还可以继承 App Trait，然后，将需要写在 main 方法中运行的代码，直接作为 object 的 constructor 代码即可，而且还可以使用 args 接收传入的参数；
- 案例说明：

```
package cn.itcast.object_demo

//1.在 object 中定义 main 方法
object Main_Demo1 {
  def main(args: Array[String]) {
```

```
    if(args.length > 0){
      println("Hello, " + args(0))
    }else{
      println("Hello World!")
    }
  }
}
//2.使用继承 App Trait ,将需要写在 main 方法中运行的代码
// 直接作为 object 的 constructor 代码即可，
// 而且还可以使用 args 接收传入的参数。

object Main_Demo2 extends App{
    if(args.length > 0){
      println("Hello, " + args(0))
    }else{
      println("Hello World!")
    }
}
```

6.3. Scala 面向对象编程之继承

6.3.1. Scala 中继承(extends)的概念

- Scala 中，让子类继承父类，与 Java 一样，也是使用 extends 关键字；
- 继承就代表，子类可继承父类的 field 和 method ，然后子类还可以在自己的内部实现父类没有的，子类特有的 field 和 method，使用继承可以有效复用代码；
- 子类可以覆盖父类的 field 和 method，但是如果父类用 final 修饰，或者 field 和 method 用 final 修饰，则该类是无法被继承的，或者 field 和 method 是无法被覆盖的。
- private 修饰的 field 和 method 不可以被子类继承，只能在类的内部使用；
- field 必须要被定义成 val 的形式才能被继承，并且还要使用 override 关键字。因为 var 修饰的 field 是可变的，在子类中可直接引用被赋值，不需要被继承；即 val 修饰的才允许被继承，var 修饰的只允许被引用。继承就是改变、覆盖的意思。
- Java 中的访问控制权限，同样适用于 Scala

	类内部	本包	子类	外部包
public	√	√	√	√
protected	√	√	√	×
default	√	√	×	×
private	√	×	×	×

- 举例说明：

```
package cn.itcast.extends_demo

class Person {
  val name="super"
  def getName=this.name
}
class Student extends Person{
  //继承加上关键字
  override
  val name="sub"
  //子类可以定义自己的 field 和 method
  val score="A"
  def getScore=this.score
}
```

6.3.2. Scala 中 override 和 super 关键字

- Scala 中，如果子类要覆盖父类中的一个非抽象方法，必须要使用 override 关键字；子类可以覆盖父类的 val 修饰的 field，只要在子类中使用 override 关键字即可。
- override 关键字可以帮助开发者尽早的发现代码中的错误，比如， override 修饰的父类方法的方法名拼写错误。
- 此外，在子类覆盖父类方法后，如果在子类中要调用父类中被覆盖的方法，则必须要使用 super 关键字，显示的指出要调用的父类方法。
- 举例说明：

```
package cn.itcast.extends_demo

class Person1 {
  private val name = "leo"
  val age=50
  def getName = this.name
}
class Student extends Person1{
  private val score = "A"
  //子类可以覆盖父类的 val field,使用 override 关键字
  override
  val age=30
  def getScore = this.score
  //覆盖父类非抽象方法，必须要使用 override 关键字
  //同时调用父类的方法，使用 super 关键字
  override def getName = "your name is " + super.getName

}
```

6.3.3. Scala 中 isInstanceOf 和 asInstanceOf

如果实例化了子类的对象，但是将其赋予了父类类型的变量，在后续的过程中，又需要将父类类型的变量转换为子类类型的变量，应该如何做？

- 首先，需要使用 isInstanceOf 判断对象是否为指定类的对象，如果是的话，则可以使用 asInstanceOf 将对象转换为指定类型；
- 注意： p.isInstanceOf[XX] 判断 p 是否为 XX 对象的实例；p.asInstanceOf[XX] 把 p 转换成 XX 对象的实例
- 注意：如果没有用 isInstanceOf 先判断对象是否为指定类的实例，就直接用 asInstanceOf 转换，则可能会抛出异常；
- 注意：如果对象是 null，则 isInstanceOf 一定返回 false， asInstanceOf 一定返回 null；
- Scala 与 Java 类型检查和转换

Scala	Java
obj.isInstanceOf[C]	obj instanceof C
obj.asInstanceOf[C]	(C)obj
classOf[C]	C.class

- 举例说明：

```
package cn.itcast.extends_demo

class Person3 {}
class Student3 extends Person3
object Student3{
    def main (args: Array[String] ) {
    val p: Person3 = new Student3
    var s: Student3 = null
    //如果对象是 null，则 isInstanceOf 一定返回 false
    println (s.isInstanceOf[Student3])
    // 判断 p 是否为 Student3 对象的实例
  if (p.isInstanceOf[Student3] ) {
    //把 p 转换成 Student3 对象的实例
      s = p.asInstanceOf[Student3]
  }
  println (s.isInstanceOf[Student3] )
  }
}
```

6.3.4. Scala 中 getClass 和 classOf

- isInstanceOf 只能判断出对象是否为指定类以及其子类的对象，而不能精确的判断出，对象就是指定类的对象；
- 如果要求精确地判断出对象就是指定类的对象，那么就只能使用 getClass 和 classOf 了；
- p.getClass 可以精确地获取对象的类，classOf[XX] 可以精确的获取类，然后使用 == 操作符即可判断；
- 举例说明：

```
package cn.itcast.extends_demo

class Person4 {}
class Student4 extends Person4
object Student4{
  def main(args: Array[String]) {
    val p:Person4=new Student4
    //判断 p 是否为 Person4 类的实例
    println(p.isInstanceOf[Person4])//true
    //判断 p 的类型是否为 Person4 类
    println(p.getClass == classOf[Person4])//false
    //判断 p 的类型是否为 Student4 类
    println(p.getClass == classOf[Student4])//true
  }
}
```

6.3.5. Scala 中使用模式匹配进行类型判断

- 在实际的开发中，比如 spark 源码中，大量的地方使用了模式匹配的语法进行类型的判断，这种方式更加地简洁明了，而且代码的可维护性和可扩展性也非常高；
- 使用模式匹配，功能性上来说，与 isInstanceOf 的作用一样，主要判断是否为该类或其子类的对象即可，不是精准判断。
- 等同于 Java 中的 switch case 语法；
- 举例说明：

```
package cn.itcast.extends_demo

class Person5 {}
class Student5 extends Person5
object Student5{
  def main(args: Array[String]) {
```

```
    val p:Person5=new Student5
    p match {
      // 匹配是否为 Person 类或其子类对象
      case per:Person5 => println("This is a Person5's Object!")
      // 匹配所有剩余情况
      case _  =>println("Unknown type!")
    }
  }
}
```

6.3.6. Scala 中 protected

- 跟 Java 一样，Scala 中同样可使用 protected 关键字来修饰 field 和 method。在子类中，可直接访问父类的 field 和 method，而不需要使用 super 关键字；
- 还可以使用 protected[this] 关键字， 访问权限的保护范围：只允许在当前子类中访问父类的 field 和 method，不允许通过其他子类对象访问父类的 field 和 method。

- 举例说明：

```
package cn.itcast.extends_demo

class Person6{
  protected var name:String="tom"
  protected[this] var hobby:String ="game"
  protected def sayBye=println("再见...")
}
class Student6 extends Person6{
  //父类使用 protected 关键字来修饰 field 可以直接访问
  def  sayHello =println("Hello "+name)
  //父类使用 protected 关键字来修饰 method 可以直接访问
  def  sayByeBye=sayBye
  def makeFriends(s:Student6)={
    println("My hobby is "+hobby+", your hobby is UnKnown")
  }
}
object Student6{
  def main(args: Array[String]) {
    val s:Student6=new Student6
    s.sayHello
    s.makeFriends(s)
    s.sayByeBye
  }
}
```

6.3.7. Scala 中调用父类的 constructor

- Scala 中，每个类都可以有一个主 constructor 和任意多个辅助 constructor，而且每个辅助 constructor 的第一行都必须调用其他辅助 constructor 或者主 constructor 代码；因此子类的辅助 constructor 是一定不可能直接调用父类的 constructor 的；
- 只能在子类的主 constructor 中调用父类的 constructor。
- 如果父类的构造函数已经定义过的 field，比如 name 和 age，子类再使用时，就不要用 val 或 var 来修饰了，否则会被认为，子类要覆盖父类的 field，且要求一定要使用 override 关键字。
- 举例说明：

```
package cn.itcast.extends_demo

class Person7(val name:String,val age:Int){
  var score :Double=0.0
  var address:String="beijing"
  def this(name:String,score:Double)={
    //每个辅助 constructor 的第一行都必须调用其他辅助 constructor 或者主 constructor 代码
    //主 constructor 代码
      this(name,30)
      this.score=score
  }
  //其他辅助 constructor
  def this(name:String,address:String)={
      this(name,100.0)
      this.address=address
  }
}
class Student7(name:String,score:Double) extends Person7(name,score)
```

6.3.8. Scala 中匿名内部类

- 在 Scala 中，匿名内部类是非常常见的，而且功能强大。Spark 的源码中大量的使用了匿名内部类；
- 匿名内部类，就是定义一个没有名称的子类，并直接创建其对象，然后将对象的引用赋予一个变量，即匿名内部类的实例化对象。然后将该对象传递给其他函数使用。

- 举例说明：

```
package cn.itcast.extends_demo

class Person8(val name:String) {
  def sayHello="Hello ,I'm "+name
}
class GreetDemo{
  //接受 Person8 参数，并规定 Person8 类只含有一个返回 String 的 sayHello 方法
  def greeting(p:Person8{
      def sayHello:String})={
          println(p.sayHello)
  }
}
object GreetDemo {
  def main(args: Array[String]) {
    //创建 Person8 的匿名子类对象
    val p=new Person8("tom")
    val g=new GreetDemo
    g.greeting(p)
  }
}
```

6.3.9. Scala 中抽象类

- 如果在父类中，有某些方法无法立即实现，而需要依赖不同的子类来覆盖，重写实现不同的方法。此时，可以将父类中的这些方法编写成只含有方法签名，不含方法体的形式，这种形式就叫做抽象方法；
- 一个类中，如果含有一个抽象方法或抽象 field，就必须使用 abstract 将类声明为抽象类，该类是不可以被实例化的；
- 在子类中覆盖抽象类的抽象方法时，可以不加 override 关键字；
- 举例说明：

```
package cn.itcast.extends_demo

abstract class Person9(val name:String) {
  //必须指出返回类型，不然默认返回为 Unit
  def sayHello:String
  def sayBye:String
}
class Student9(name:String) extends Person9(name){
  //必须指出返回类型，不然默认
  def sayHello: String = "Hello,"+name
  def sayBye: String ="Bye,"+name
```

```
}
object Student9{
  def main(args: Array[String]) {
    val s = new Student9("tom")
    println(s.sayHello)
    println(s.sayBye)
  }
}
```

6.3.10. Scala 中抽象 field

- 如果在父类中，定义了 field，但是没有给出初始值，则此 field 为抽象 field；
- 举例说明：

```
package cn.itcast.extends_demo

abstract class Person10 (val name:String){
    //抽象 fields
    val age:Int
}
class Student10(name: String) extends Person10(name) {
   val age: Int = 50
}
```

6.4. Scala 中面向对象编程之 trait

6.4.1. 将 trait 作为接口使用

- Scala 中的 trait 是一种特殊的概念；
- 首先先将 trait 作为接口使用，此时的 trait 就与 Java 中的接口 (interface)非常类似；
- 在 trait 中可以定义抽象方法，就像抽象类中的抽象方法一样，只要不给出方法的方法体即可；
- 类可以使用 extends 关键字继承 trait，注意，这里不是 implement，而是 extends ，在 Scala 中没有 implement 的概念，无论继承类还是 trait，统一都是 extends；
- 类继承后，必须实现其中的抽象方法，实现时，不需要使用 override 关键字；
- Scala 不支持对类进行多继承，但是支持多重继承 trait，使用 with 关键字即可。
- 举例说明：

```
package cn.itcast.triat

trait HelloTrait {
  def sayHello(): Unit
}
trait MakeFriendsTrait {
  def makeFriends(c: Children): Unit
}
//多重继承 trait
class Children(val name: String) extends HelloTrait with MakeFriendsTrait with Cloneable
with Serializable{
  def sayHello() =println("Hello, " + this.name)
  def makeFriends(c: Children) = println("Hello, my name is " + this.name + ", your name
is " + c.name)
}
object Children{
  def main(args: Array[String]) {
    val c1=new Children("tom")
    val c2=new Children("jim")
    c1.sayHello()//Hello, tom
    c1.makeFriends(c2)//Hello, my name is tom, your name is jim
  }
}
```

6.4.2. 在 trait 中定义具体的方法

- Scala 中的 trait 不仅可以定义抽象方法，还可以定义具体的方法，此时 trait 更像是包含了通用方法的工具，可以认为 trait 还包含了类的功能。
- 举例说明：

```
package cn.itcast.triat
/**
 * 比如 trait 中可以包含很多子类都通用的方法，例如打印日志或其他工具方法等等。
 * spark 就使用trait 定义了通用的日志打印方法：
 */
trait Logger {
  def log(message: String): Unit = println(message)
}
class PersonForLog(val name: String) extends Logger {
  def makeFriends(other: PersonForLog) = {
    println("Hello, " + other.name + "! My name is " + this.name + ", I miss you!!")
    this.log("makeFriends method is invoked with parameter PersonForLog[name = " +
```

```
other.name + "]")
  }
}
object PersonForLog{
  def main(args: Array[String]) {
    val p1=new PersonForLog("jack")
    val p2=new PersonForLog("rose")
    p1.makeFriends(p2)
    //Hello, rose! My name is jack, I miss you!!
    //makeFriens method is invoked with parameter PersonForLog[name = rose]
  }
}
```

6.4.3. 在 trait 中定义具体 field

- Scala 中的 trait 可以定义具体的 field，此时继承 trait 的子类就自动获得了 trait 中定义的 field；
- 但是这种获取 field 的方式与继承 class 的是不同的。如果是继承 class 获取的 field，实际上还是定义在父类中的；而继承 trait 获取的 field，就直接被添加到子类中了。
- 举例说明：

```
package cn.itcast.triat

trait PersonForField {
  val  age:Int=50
}

//继承 trait 获取的 field 直接被添加到子类中
class StudentForField(val name: String) extends PersonForField {
  def sayHello = println("Hi, I'm " + this.name + ", my  age  is "+ age)
}

object StudentForField{
  def main(args: Array[String]) {
    val s=new StudentForField("tom")
    s.sayHello
  }
}
```

6.4.4. 在 trait 中定义抽象 field

- Scala 中的 trait 也能定义抽象 field， 而 trait 中的具体方法也能基于抽象 field 编写；
- 继承 trait 的类，则必须覆盖抽象 field，提供具体的值；
- 举例说明：

```
package cn.itcast.triat

trait SayHelloTrait {
  val msg:String
  def sayHello(name: String) = println(msg + ", " + name)
}

class PersonForAbstractField(val name: String) extends SayHelloTrait {
  //必须覆盖抽象 field
  val msg = "Hello"
  def makeFriends(other: PersonForAbstractField) = {
    this.sayHello(other.name)
    println("I'm " + this.name + ", I want to make friends with you!!")
  }
}
object PersonForAbstractField{
  def main(args: Array[String]) {
    val p1=new PersonForAbstractField("Tom")
    val p2=new PersonForAbstractField("Rose")
    p1.makeFriends(p2)
  }
}
```

6.4.5. 在实例对象指定混入某个 trait

- 可在创建类的对象时，为该对象指定混入某个 trait，且只有混入了 trait 的对象才具有 trait 中的方法，而其他该类的对象则没有；
- 在创建对象时，使用 with 关键字指定混入某个 trait；
- 举例说明：

```
package cn.itcast.triat
```

```
trait LoggedTrait {
  // 该方法为实现的具体方法
  def log(msg: String) = {}
}
trait MyLogger extends LoggedTrait{
  // 覆盖 log() 方法
override def log(msg: String) = println("log: " + msg)
}

class PersonForMixTraitMethod(val name: String) extends LoggedTrait {
  def sayHello = {
    println("Hi, I'm " + this.name)
    log("sayHello method is invoked!")
  }
}
object PersonForMixTraitMethod{
  def main(args: Array[String]) {
    val tom= new PersonForMixTraitMethod("Tom").sayHello //结果为：Hi, I'm Tom
    // 使用 with 关键字，指定混入 MyLogger trait
    val rose = new PersonForMixTraitMethod("Rose") with MyLogger
    rose.sayHello
// 结果为:      Hi, I'm Rose
// 结果为:      log: sayHello method is invoked!
  }
}
```

6.4.6. trait 调用链

- Scala 中支持让类继承多个 trait 后，可依次调用多个 trait 中的同一个方法，只要让多个 trait 中的同一个方法，在最后都依次执行 super 关键字即可；
- 类中调用多个 trait 中都有的这个方法时，首先会从最右边的 trait 的方法开始执行，然后依次往左执行，形成一个调用链条；
- 这种特性非常强大，其实就是设计模式中责任链模式的一种具体实现；
- 案例说明：

```
package cn.itcast.triat

trait HandlerTrait {
  def handle(data: String) = {println("last one")}
}
trait DataValidHandlerTrait extends HandlerTrait {
```

```
  override def handle(data: String) = {
            println("check data: " + data)
            super.handle(data)
}
}
trait SignatureValidHandlerTrait extends HandlerTrait {
  override def handle(data: String) = {
         println("check signature: " + data)
         super.handle(data)
  }
}
class PersonForRespLine(val name: String) extends SignatureValidHandlerTrait with
DataValidHandlerTrait {
  def sayHello = {
       println("Hello, " + this.name)
       this.handle(this.name)
  }
}
object PersonForRespLine{
  def main(args: Array[String]) {
     val p=new PersonForRespLine("tom")
      p.sayHello
      //执行结果:
//    Hello, tom
//    check data: tom
//    check signature: tom
//    last one
  }
}
```

6.4.7. 混合使用 trait 的具体方法和抽象方法

- 在 trait 中，可以混合使用具体方法和抽象方法；
- 可以让具体方法依赖于抽象方法，而抽象方法则可放到继承 trait 的子类中去实现；
- 这种 trait 特性，其实就是设计模式中的模板设计模式的体现；
- 举例说明：

```
package cn.itcast.triat

trait ValidTrait {
 //抽象方法
  def getName: String
//具体方法，具体方法的返回值依赖于抽象方法

 def valid: Boolean = {"Tom".equals(this.getName)
  }
}
class PersonForValid(val name: String) extends ValidTrait {
  def getName: String = this.name
}

object PersonForValid{
  def main(args: Array[String]): Unit = {
    val person = new PersonForValid("Rose")
    println(person.valid)
  }
}
```

6.4.8. trait 的构造机制

- 在 Scala 中，trait 也是有构造代码的，即在 trait 中，不包含在任何方法中的代码；
- 继承了 trait 的子类，其构造机制如下：
- 父类的构造函数先执行， class 类必须放在最左边；多个 trait 从左向右依次执行；构造 trait 时，先构造父 trait，如果多个 trait 继承同一个父 trait，则父 trait 只会构造一次；所有 trait 构造完毕之后，子类的构造函数最后执行。
- 举例说明：

```
package cn.itcast.triat

class Person_One {
  println("Person's constructor!")
}
trait Logger_One {
  println("Logger's constructor!")
}
```

```
trait MyLogger_One extends Logger_One {
  println("MyLogger's constructor!")
}
trait TimeLogger_One extends Logger_One {
  println("TimeLogger's contructor!")
}
class Student_One extends Person_One with MyLogger_One with TimeLogger_One {
  println("Student's constructor!")
  }
object exe_one {
  def main(args: Array[String]): Unit = {
    val student = new Student_One
    //执行结果为:
//      Person's constructor!
//      Logger's constructor!
//      MyLogger's constructor!
//      TimeLogger's contructor!
//      Student's constructor!
  }
}
```

6.4.9. trait 继承 class

- 在 Scala 中 trait 也可以继承 class，此时这个 class 就会成为所有继承该 trait 的子类的超级父类。
- 举例说明：

```
package cn.itcast.triat

class MyUtil {
  def printMsg(msg: String) = println(msg)
}
trait Logger_Two extends MyUtil {
  def log(msg: String) = this.printMsg("log: " + msg)
}
class Person_Three(val name: String) extends Logger_Two {
    def sayHello {
        this.log("Hi, I'm " + this.name)
        this.printMsg("Hello, I'm " + this.name)
  }
}
object Person_Three{
```

```
  def main(args: Array[String]) {
      val p=new Person_Three("Tom")
      p.sayHello
    //执行结果：
//      log: Hi, I'm Tom
//      Hello, I'm Tom
  }
}
```

7. 模式匹配和样例类

Scala 有一个十分强大的模式匹配机制，可以应用到很多场合：如 switch 语句、类型检查等。并且 Scala 还提供了样例类，对模式匹配进行了优化，可以快速进行匹配。

7.1. 匹配字符串

```
package cn.itcast.cases
import scala.util.Random

object CaseDemo01 extends App{
  val arr = Array("hadoop", "zookeeper", "spark")
  val name = arr(Random.nextInt(arr.length))
  name match {
    case "hadoop"    => println("大数据分布式存储和计算框架...")
    case "zookeeper" => println("大数据分布式协调服务框架...")
    case "spark" => println("大数据分布式内存计算框架...")
    case _ => println("我不认识你...")
  }
}
```

7.2. 匹配类型

```
package cn.itcast.cases
```

```
import scala.util.Random

object CaseDemo01 extends App{
  val arr = Array("hello", 1, 2.0, CaseDemo)
  val v = arr(Random.nextInt(4))
  println(v)
  v match {
    case x: Int => println("Int " + x)
    case y: Double if(y >= 0) => println("Double "+ y)
    case z: String => println("String " + z)
    case _ => throw new Exception("not match exception")
  }
}
```

注意： case y: Double if(y >= 0) => ...

模式匹配的时候还可以添加守卫条件。如不符合守卫条件，将掉入 case _中。

7.3. 匹配数组、元组、集合

```
package cn.itcast.cases

object CaseDemo03 extends App{

  val arr = Array(1, 3, 5)
  arr match {
    case Array(1, x, y) => println(x + " " + y)
    case Array(0) => println("only 0")
    case Array(0, _*) => println("0 ...")
    case _ => println("something else")
  }

  val lst = List(3, -1)
  lst match {
    case 0 :: Nil => println("only 0")
    case x :: y :: Nil => println(s"x: $x y: $y")
    case 0 :: tail => println("0 ...")
    case _ => println("something else")
  }

  val tup = (1, 3, 7)
  tup match {
    case (1, x, y) => println(s"1, $x , $y")
```

```
    case (_, z, 5) => println(z)
    case  _ => println("else")
  }
}
```

注意：在 Scala 中列表要么为空（Nil 表示空列表）要么是一个 head 元素加上一个 tail 列表。
9 :: List(5, 2) :: 操作符是将给定的头和尾创建一个新的列表
注意：:: 操作符是右结合的，如 9 :: 5 :: 2 :: Nil 相当于 9 :: (5 :: (2 :: Nil))

7.4. 样例类

在 Scala 中样例类是一种特殊的类，可用于模式匹配。

定义形式：

case class 类型，是多例的，后面要跟构造参数。

case object 类型，是单例的。

```
package cn.itcast.cases
import scala.util.Random

case class SubmitTask(id: String, name: String)
case class HeartBeat(time: Long)
case object CheckTimeOutTask

object CaseDemo04 extends App{
  val arr = Array(CheckTimeOutTask, HeartBeat(12333), SubmitTask("0001", "task-0001"))

  arr(Random.nextInt(arr.length)) match {
    case SubmitTask(id, name) => {
      println(s"$id, $name")
    }
    case HeartBeat(time) => {
      println(time)
    }
    case CheckTimeOutTask => {
      println("check")
    }
  }
}
```

7.5. Option 类型

在 Scala 中 Option 类型用样例类来表示可能存在或者可能不存在的值(Option 的子类有 Some 和 None)。Some 包装了某个值，None 表示没有值

```
package cn.itcast.cases

object OptionDemo {
  def main(args: Array[String]) {
    val map = Map("a" -> 1, "b" -> 2)
    val v = map.get("b") match {
      case Some(i) => i
      case None => 0
    }
    println(v)
    //更好的方式
    val v1 = map.getOrElse("c", 0)
    println(v1)
  }
}
```

7.6. 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 代表输入参数类型，B 代表返回结果类型，常用作输入模式匹配，偏函数最大的特点就是它只接受和处理其参数定义域的一个子集。

```
package cn.itcast.cases

object PartialFuncDemo  {

  val 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"))
  }
}
```

8. Scala 中的协变、逆变、非变

8.1. 协变、逆变、非变介绍

协变和逆变主要是用来解决参数化类型的泛化问题。Scala 的协变与逆变是非常有特色的，完全解决了 Java 中泛型的一大缺憾：举例来说，Java 中，如果有 A 是 B 的子类，但 Card[A] 却不是 Card[B] 的子类；而 Scala 中，只要灵活使用协变与逆变，就可以解决此类 Java 泛型问题；

由于参数化类型的参数（参数类型）是可变的，当两个参数化类型的参数是继承关系（可泛化），那被参数化的类型是否也可以泛化呢？Java 中这种情况下是不可泛化的，然而 Scala 提供了三个选择，即协变（“+”）、逆变（“-”）和非变。

下面说一下三种情况的含义，首先假设有参数化特征 Queue，那它可以有如下三种定义。

(1) trait Queue[T] {}

这是非变情况。这种情况下，当类型 B 是类型 A 的子类型，则 Queue[B]与 Queue[A]没有任何从属关系，这种情况是和 Java 一样的。

(2) trait Queue[+T] {}

这是协变情况。这种情况下，当类型 B 是类型 A 的子类型，则 Queue[B]也可以认为是 Queue[A]的子类型，即 Queue[B]可以泛化为 Queue[A]。也就是被参数化类型的泛化方向与参数类型的方向是一致的，所以称为协变。

(3) trait Queue[-T] {}

这是逆变情况。这种情况下，当类型 B 是类型 A 的子类型，则 Queue[A]反过来可以认为是 Queue[B]的子类型。也就是被参数化类型的泛化方向与参数类型的方向是相反的，所以称为逆变。

8.2. 协变、逆变、非变总结

- C[+T]：如果 A 是 B 的子类，那么 C[A]是 C[B]的子类。
- C[-T]：如果 A 是 B 的子类，那么 C[B]是 C[A]的子类。
- C[T]： 无论 A 和 B 是什么关系，C[A]和 C[B]没有从属关系。

8.3. 案例

```
package cn.itcast.scala.enhance.covariance

class Super
class Sub extends Super
//协变
class Temp1[+A](title: String)
//逆变
class Temp2[-A](title: String)
//非变
class Temp3[A](title: String)

object Covariance_demo{
  def main(args: Array[String]) {
    //支持协变 Temp1[Sub]还是 Temp1[Super]的子类
    val t1: Temp1[Super] = new Temp1[Sub]("hello scala!!!")
    //支持逆变 Temp1[Super]是 Temp1[Sub]的子类
    val t2: Temp2[Sub] = new Temp2[Super]("hello scala!!!")
    //支持非变 Temp3[Super]与 Temp3[Sub]没有从属关系，如下代码会报错
    //val t3: Temp3[Sub] = new Temp3[Super]("hello scala!!!")
    //val t4: Temp3[Super] = new Temp3[Sub]("hello scala!!!")
    println(t1.toString)
    println(t2.toString)
  }
}
```

9. Scala 中的上下界

9.1. 上界、下界介绍

在指定泛型类型时，有时需要界定泛型类型的范围，而不是接收任意类型。比如，要求某个泛型类型，必须是某个类的子类，这样在程序中就可以放心的调用父类的方法，程序才能正常的使用与运行。此时，就可以使用上下边界 Bounds 的特性；

Scala 的上下边界特性允许泛型类型是某个类的子类，或者是某个类的父类；

(1) U >: T

这是类型下界的定义，也就是 U 必须是类型 T 的父类(或本身，自己也可以认为是自己的父类)。

(2) S <: T

这是类型上界的定义，也就是 S 必须是类型 T 的子类（或本身，自己也可以认为是自己的子类)。

10. Scala Actor 并发编程

10.1. 课程目标

10.1.1. 目标一：熟悉 Scala Actor 并发编程

10.1.2. 目标二：为学习 Akka 做准备

注：Scala Actor 是 scala 2.10.x 版本及以前版本的 Actor。
Scala 在 2.11.x 版本中将 Akka 加入其中，作为其默认的 Actor，老版本的 Actor 已经废弃。

10.2. 什么是 Scala Actor

10.2.1. 概念

Scala 中的 Actor 能够实现并行编程的强大功能，它是基于事件模型的并发机制，Scala 是运用消息的发送、接收来实现高并发的。

Actor 可以看作是一个个独立的实体，他们之间是毫无关联的。但是，他们可以通过消息来通信。一个 Actor 收到其他 Actor 的信息后，它可以根据需要作出各种相应。消息的类型可以是任意的，消息的内容也可以是任意的。

10.2.2. java 并发编程与 Scala Actor 编程的区别

Java内置线程模型	Scala actor模型
“共享数据-锁”模型（share data and lock）	share nothing
每个object有一个monitor，监视多线程对共享数据的访问	不共享数据，actor之间通过message通讯
加锁的代码段用synchronized标识	
死锁问题	
每个线程内部是顺序执行的	每个actor内部是顺序执行的

对于 Java，我们都知道它的多线程实现需要对共享资源（变量、对象等）使用 synchronized 关键字进行代码块同步、对象锁互斥等等。而且，常常一大块的 try…catch 语句块中加上 wait 方法、notify 方法、notifyAll 方法是让人很头疼的。原因就在于 Java 中多数使用的是可变状态的对象资源，对这些资源进行共享来实现多线程编程的话，控制好资源竞争与防止对象状态被意外修改是非常重要的，而对象状态的不变性也是较难以保证的。

与 **Java** 的基于共享数据和锁的线程模型不同，**Scala** 的 actor 包则提供了另外一种不共享任何数据、依赖消息传递的模型,从而进行并发编程。

10.2.3. Actor 的执行顺序

1、首先调用 start()方法启动 Actor

2、调用 start()方法后其 act()方法会被执行

3、向 Actor 发送消息

4、act 方法执行完成之后，程序会调用 exit 方法

10.2.4. 发送消息的方式

!	发送异步消息，没有返回值。
!?	发送同步消息，等待返回值。
!!	发送异步消息，返回值是 Future[Any]。

注意：Future 表示一个异步操作的结果状态，可能还没有实际完成的异步任务的结果

Any 是所有类的超类，Future[Any]的泛型是异步操作结果的类型。

10.3. Actor 实战

10.3.1. 第一个例子

怎么实现 actor 并发编程：

1、定义一个 class 或者是 object 继承 Actor 特质，注意导包 import scala.actors.Actor

2、重写对应的 act 方法

3、调用 Actor 的 start 方法执行 Actor

4、当 act 方法执行完成，整个程序运行结束

```
package cn.itcast.actor
import scala.actors.Actor

object Actor1 extends Actor{
  //重写 act 方法
  def act(){
    for(i <- 1 to 10){
      println("actor-1 " + i)
      Thread.sleep(2000)
    }
  }
}

object Actor2 extends Actor{
  //重写 act 方法
  def act(){
    for(i <- 1 to 10){
      println("actor-2 " + i)
      Thread.sleep(2000)
    }
```

```
  }
}

object ActorTest extends App{
  //启动 Actor
  Actor1.start()
  Actor2.start()
}
```

说明：上面分别调用了两个单例对象的 start()方法，他们的 act()方法会被执行，相同与在 java 中开启了两个线程，线程的 run()方法会被执行

注意：这两个 Actor 是并行执行的，act()方法中的 for 循环执行完成后 actor 程序就退出了

10.3.2. 第二个例子

怎么实现 actor 发送、接受消息

1、定义一个 class 或者是 object 继承 Actor 特质，注意导包 import scala.actors.Actor

2、重写对应的 act 方法

3、调用 Actor 的 start 方法执行 Actor

4、通过不同发送消息的方式对 actor 发送消息

5、act 方法中通过 receive 方法接受消息并进行相应的处理

6、act 方法执行完成之后，程序退出

```
package cn.itcast.actor
import scala.actors.Actor
class MyActor extends Actor {

  override def act(): Unit = {
      receive {
        case "start" => {
          println("starting ...")
        }
      }
    }
  }
}
```

```
object MyActor {
  def main(args: Array[String]) {
    val actor = new MyActor
    actor.start()
    actor ! "start"

    println("消息发送完成！")
  }
}
```

10.3.3. 第三个例子

怎么实现 actor 可以不断地接受消息：

在 act 方法中可以使用 while(true)的方式，不断的接受消息。

```
package cn.itcast.actor
import scala.actors.Actor
class MyActor1 extends Actor {

  override def act(): Unit = {
    while (true) {
      receive {
        case "start" => {
          println("starting ...")
        }
        case "stop" => {
          println("stopping ...")
        }
      }
    }
  }
}

object MyActor1 {
  def main(args: Array[String]) {
    val actor = new MyActor1
    actor.start()
    actor ! "start"
```

```
    actor ! "stop"
  }
}
```

说明：在 act()方法中加入了 while（true）循环，就可以不停的接收消息
注意：发送 start 消息和 stop 的消息是异步的，但是 Actor 接收到消息执行的过程是同步的按顺序执行

10.3.4. 第四个例子

使用 react 方法代替 receive 方法去接受消息

好处：react 方式会复用线程，避免频繁的线程创建、销毁和切换。比 receive 更高效

注意： react 如果要反复执行消息处理，react 外层要用 loop，不能用 while

```
package cn.itcast.actor
import scala.actors.Actor
class YourActor extends Actor {
  override def act(): Unit = {
    loop {
      react {
        case "start" => {
          println("starting ...")
        }
        case "stop" => {
          println("stopping ...")

        }
      }
    }
  }
}


object YourActor {
  def main(args: Array[String]) {
    val actor = new YourActor
    actor.start()
    actor ! "start"
    actor ! "stop"
    println("消息发送完成！")
  }
```

```
}
```

10.3.5. 第五个例子

结合 case class 样例类发送消息和接受消息

1、将消息封装在一个样例类中
2、通过匹配不同的样例类去执行不同的操作
3、Actor 可以返回消息给发送方。通过 sender 方法向当前消息发送方返回消息

```
package cn.itcast.actor
import scala.actors.Actor


case class SyncMessage(id:Int,msg:String)//同步消息
case class AsyncMessage(id:Int,msg:String)//异步消息
case class ReplyMessage(id:Int,msg:String)//返回结果消息

class MsgActor extends Actor{
  override def act(): Unit ={
    loop{
      react{
        case "start"=>{println("starting....")}

        case SyncMessage(id,msg)=>{
          println(s"id:$id, SyncMessage: $msg")
          Thread.sleep(2000)
          sender !ReplyMessage(1,"finished...")
        }
        case AsyncMessage(id,msg)=>{
          println(s"id:$id,AsyncMessage: $msg")
         // Thread.sleep(2000)
          sender !ReplyMessage(3,"finished...")
          Thread.sleep(2000)
        }

      }
    }
  }
```

```
}

object MainActor {
  def main(args: Array[String]): Unit = {
     val mActor=new MsgActor
        mActor.start()
        mActor!"start"

        //同步消息 有返回值
     val reply1= mActor!?SyncMessage(1,"我是同步消息")
      println(reply1)
      println("==============================")
        //异步无返回消息
     val reply2=mActor!AsyncMessage(2,"我是异步无返回消息")

      println("==============================")
        //异步有返回消息
    val reply3=mActor!!AsyncMessage(3,"我是异步有返回消息")
    //Future 的 apply()方法会构建一个异步操作且在未来某一个时刻返回一
个值
      val result=reply3.apply()
      println(result)

  }
}
```

10.3.6. 练习实战

需求：

用 actor 并发编程写一个单机版的 WordCount，将多个文件作为输入，计算完成后将多个任务汇总，得到最终的结果。

大致的思想步骤：

1、通过 loop +react 方式去不断的接受消息

2、利用 case class 样例类去匹配对应的操作

3、其中 scala 中提供了文件读取的接口 Source,通过调用其 fromFile 方法去获取文件内容

4、将每个文件的单词数量进行局部汇总，存放在一个 ListBuffer 中

5、最后将 ListBuffer 中的结果进行全局汇总。

```
package cn.itcast.actor
import java.io.File
import scala.actors.{Actor, Future}
import scala.collection.mutable
import scala.io.Source

case class SubmitTask(fileName: String)
case class ResultTask(result: Map[String, Int])
class Task extends Actor {
  override def act(): Unit = {
    loop {
      react {
        case SubmitTask(fileName) => {
          val contents = Source.fromFile(new File(fileName)).mkString
          val arr = contents.split("\r\n")
          val result = arr.flatMap(_.split(" ")).map((_, 1)).groupBy(_._1).mapValues(_.length)
          //val result = arr.flatMap(_.split(" ")).map((_,
1)).groupBy(_._1).mapValues(_.foldLeft(0)(_ + _._2))
          sender ! ResultTask(result)
        }
      }
    }
  }
}

object WorkCount {
  def main(args: Array[String]) {
    val files = Array("d://aaa.txt", "d://bbb.txt","d://ccc.txt")
    val replaySet = new mutable.HashSet[Future[Any]]
    val resultList = new mutable.ListBuffer[ResultTask]
    for(f <- files) {
      val t = new Task
      val replay = t.start() !! SubmitTask(f)
      replaySet += replay
    }
    while(replaySet.size > 0){
      val toCumpute = replaySet.filter(_.isSet)
      for(r <- toCumpute){
        val result = r.apply()
```

```
            resultList += result.asInstanceOf[ResultTask]
            replaySet.remove(r)
          }

        }
        val finalResult = resultList.map(_.result).flatten.groupBy(_._1).mapValues(x =>
x.foldLeft(0)(_ + _._2))
        println(finalResult)
      }
}
```