


Scala 编程

1. 课程目标.....	2
1.1. 目标 1：（初级）熟练使用 scala 编写 Spark 程序.....	2
1.2. 目标 2：（中级）动手编写一个简易 Spark 通信框架.....	3
1.3. 目标 3：（高级）为阅读 Spark 内核源码做准备.....	4
2. Scala 概述.....	4
2.1. 什么是 Scala.....	4
2.2. 为什么要学 Scala.....	4
3. Scala 编译器安装.....	5
3.1. 安装 JDK.....	5
3.2. 安装 Scala.....	5
3.2.1. Windows 安装 Scala 编译器.....	5
3.2.2. Linux 安装 Scala 编译器.....	5
3.2.3. Scala 开发工具安装.....	6
4. Scala 基础.....	9
4.1. 声明变量.....	9
4.2. 常用类型.....	9
4.3. 条件表达式.....	9
4.4. 块表达式.....	10
4.5. 循环.....	11
4.6. 调用方法和函数.....	12
4.7. 定义方法和函数.....	12
4.7.1. 定义方法.....	12
4.7.2. 定义函数.....	13
4.7.3. 方法和函数的区别.....	13
4.7.4. 将方法转换成函数（神奇的下划线）.....	14
5. 数组、映射、元组、集合.....	14
5.1. 数组.....	14
5.1.1. 定长数组和变长数组.....	14
5.1.2. 遍历数组.....	16
5.1.3. 数组转换.....	17
5.1.4. 数组常用算法.....	17
5.2. 映射.....	18
5.2.1. 构建映射.....	18
5.2.2. 获取和修改映射中的值.....	18
5.3. 元组.....	19
5.3.1. 创建元组.....	19
5.3.2. 获取元组中的值.....	20
5.3.3. 将对偶的集合转换成映射.....	20
5.3.4. 拉链操作.....	20

5.4. 集合.....	21
5.4.1. 序列.....	21
5.5. Set.....	22
5.6. Map.....	23
6. 类、对象、继承、特质.....	24
6.1. 类.....	24
6.1.1. 类的定义.....	24
6.1.2. 构造器.....	24
6.2. 对象.....	26
6.2.1. 单例对象.....	26
6.2.2. 伴生对象.....	27
6.2.3. apply 方法.....	27
6.2.4. 应用程序对象.....	28
6.3. 继承.....	28
6.3.1. 扩展类.....	28
6.3.2. 重写方法.....	28
6.3.3. 类型检查和转换.....	29
6.3.4. 超类的构造.....	29
7. 模式匹配和样例类.....	30
7.1. 匹配字符串.....	30
7.2. 匹配类型.....	30
7.3. 匹配数组、元组.....	31
7.4. 样例类.....	32
7.5. Option 类型.....	32
7.6. 偏函数.....	33

1. 课程目标

1.1. 目标 1：（初级）熟练使用 scala 编写 Spark 程序

```
welcome to
 version 1.5.2

Using Scala version 2.10.4 (Java HotSpot(TM) 64-Bit Server VM, Java 1.7.0_45)
Type in expressions to have them evaluated.
Type :help for more information.
15/11/14 15:02:31 WARN MetricsSystem: Using default name DAGScheduler for source because spark.app.id is not set.
Spark context available as sc.
15/11/14 15:02:33 WARN Connection: BoneCP specified but not present in CLASSPATH (or one of dependencies)
15/11/14 15:02:34 WARN Connection: BoneCP specified but not present in CLASSPATH (or one of dependencies)
15/11/14 15:02:37 WARN ObjectStore: Version information not found in metastore. hive.metastore.schema.verification is not enabled so recording the schema version
15/11/14 15:02:37 WARN ObjectStore: Failed to get database default, returning NoSuchObjectException
15/11/14 15:02:39 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
15/11/14 15:02:40 WARN Connection: BoneCP specified but not present in CLASSPATH (or one of dependencies)
15/11/14 15:02:40 WARN Connection: BoneCP specified but not present in CLASSPATH (or one of dependencies)
SQL context available as sqlContext.

scala> sc.textFile("hdfs://node1.itcast.cn:9000/words.txt").flatMap(_.split(" ")).map(_._1).reduceByKey(_+_).saveAsTextFile("hdfs://node1.itcast.cn:9000/out")
```

```

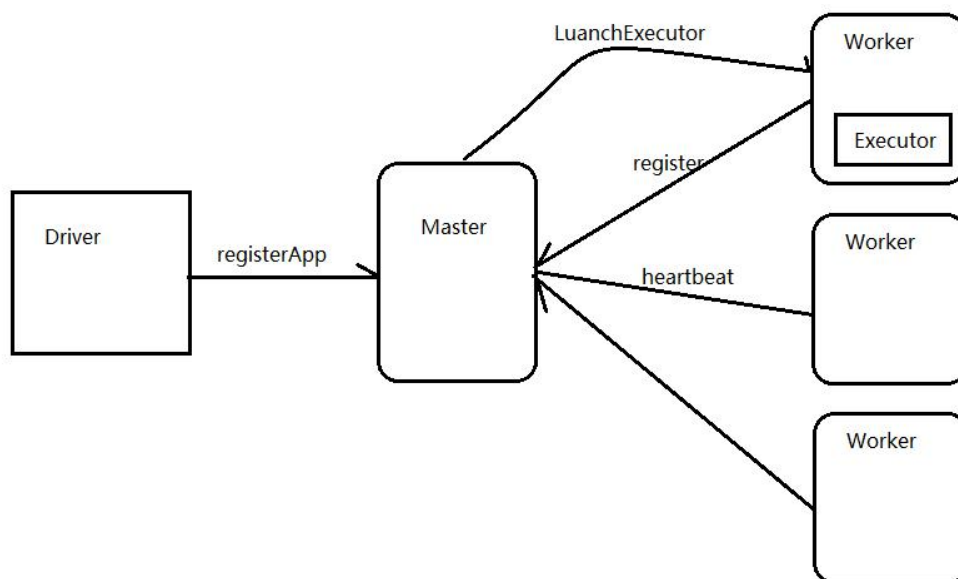
object UrlCount {

  val updateFunc = (iterator: Iterator[(String, Seq[Int], Option[Int])]) => {
    iterator.flatMap{case (x, y, z) => Some(y.sum + z.getOrElse(0)).map(n => (x, n))}
  }

  def main(args: Array[String]) {
    LoggerLevel.setStreamingLogLevels()
    val Array(zkQuorum, groupId, topics, numThreads, hdfs) = args
    val conf = new SparkConf().setAppName("UrlCount")
    val ssc = new StreamingContext(conf, Seconds(2))
    ssc.checkpoint(hdfs)
    val topicMap = topics.split(",").map(_._2).map(_._1).toMap
    val lines = KafkaUtils.createStream(ssc, zkQuorum, groupId, topicMap, StorageLevel.MEMORY_AND_DISK).map(_._2)
    val urls = lines.map(x => x.split(" ")[0, 1])
    val result = urls.updateStateByKey(updateFunc, new HashPartitioner(ssc.sparkContext.defaultParallelism), true)
    result.print()
    ssc.start()
    ssc.awaitTermination()
  }
}

```

1.2. 目标 2：（中级）动手编写一个简易 Spark 通信框架



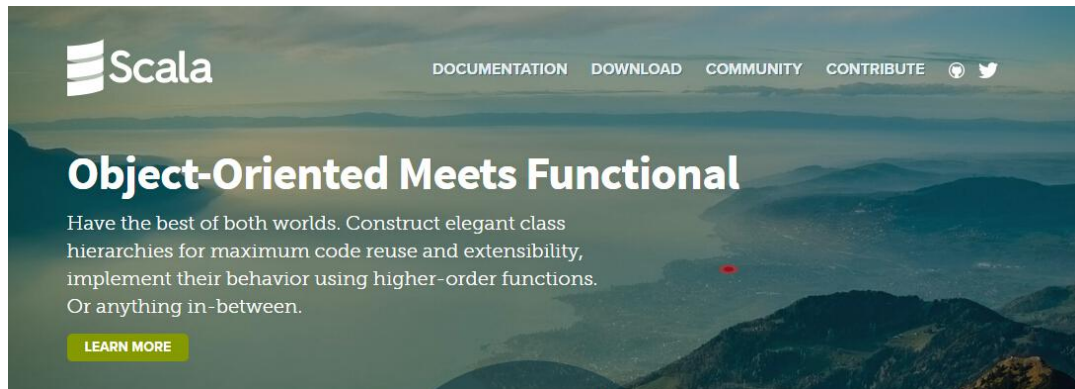
1.3. 目标 3：（高级）为阅读 Spark 内核源码做准备

```
861 private[spark] object Master extends Logging {  
862     val systemName = "sparkMaster"  
863     private val actorName = "Master"  
864  
865     def main(argStrings: Array[String]) {  
866         SignalLogger.register(log)  
867         val conf = new SparkConf  
868         val args = new MasterArguments(argStrings, conf)  
869         val (actorSystem, _, _) = startSystemAndActor(args.host, args.port, args.webUiPort, conf)  
870         actorSystem.awaitTermination()  
871     }  
872 }
```

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。



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，但是目前大多数的框架都是用 2.10.x 编写开发的，所以这里推荐 2.10.x 版本，下载 scala-2.10.6.msi 后点击下一步就可以了

3.2.2. Linux 安装 Scala 编译器

下载 Scala 地址 <http://downloads.typesafe.com/scala/2.10.6/scala-2.10.6.tgz> 然后解压 Scala 到指定目录

```
tar -zxvf scala-2.10.6.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/jdk1.7.0_45
```

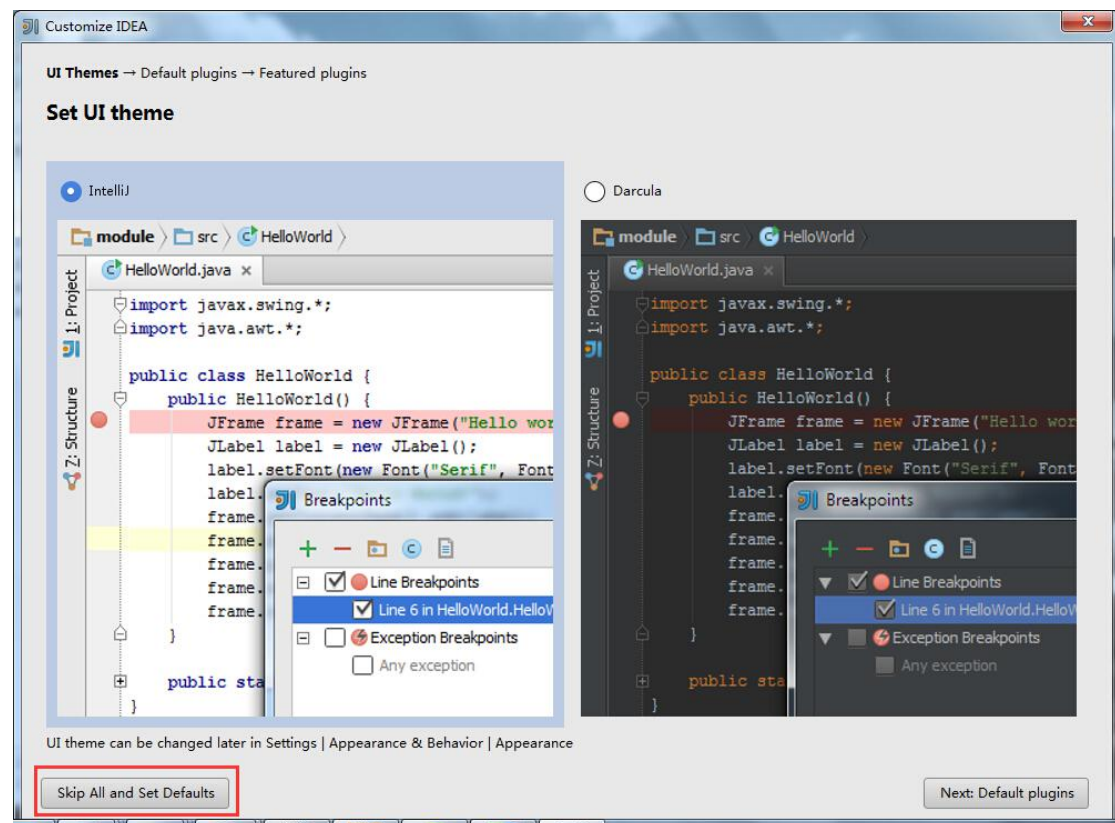
```
export PATH=$PATH:$JAVA_HOME/bin:/usr/java/scala-2.10.6/bin
```

3.2.3. Scala 开发工具安装

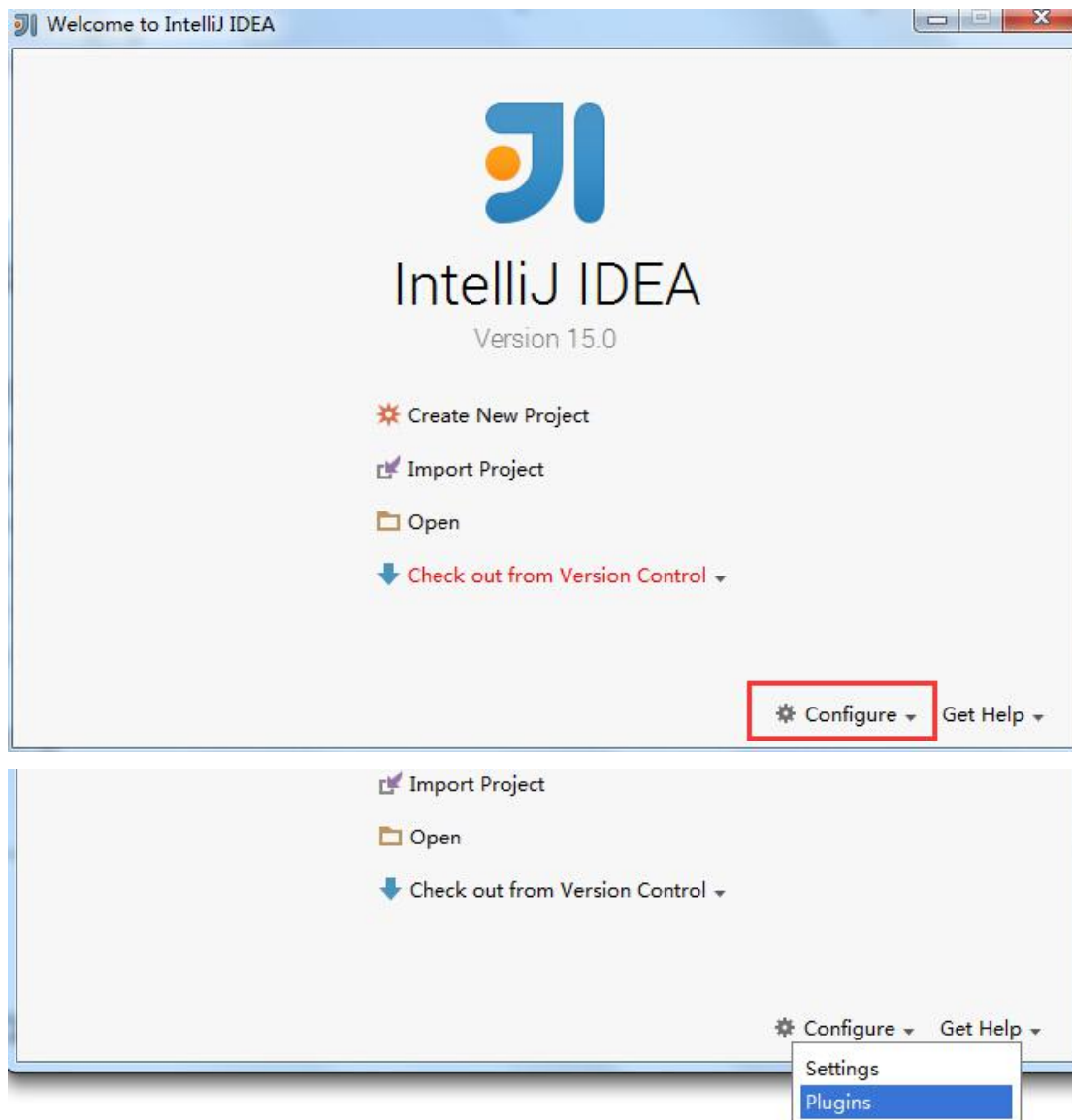
目前 Scala 的开发工具主要有两种：Eclipse 和 IDEA，这两个开发工具都有相应的 Scala 插件，如果使用 Eclipse，直接到 Scala 官网下载即可 <http://scala-ide.org/download/sdk.html>。

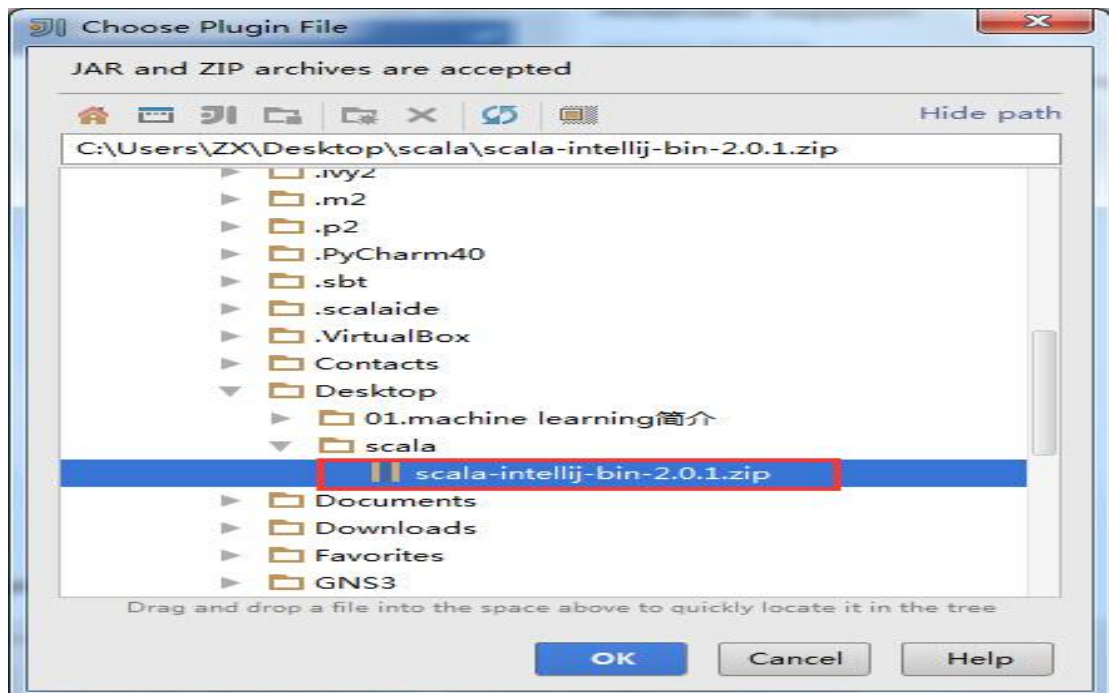
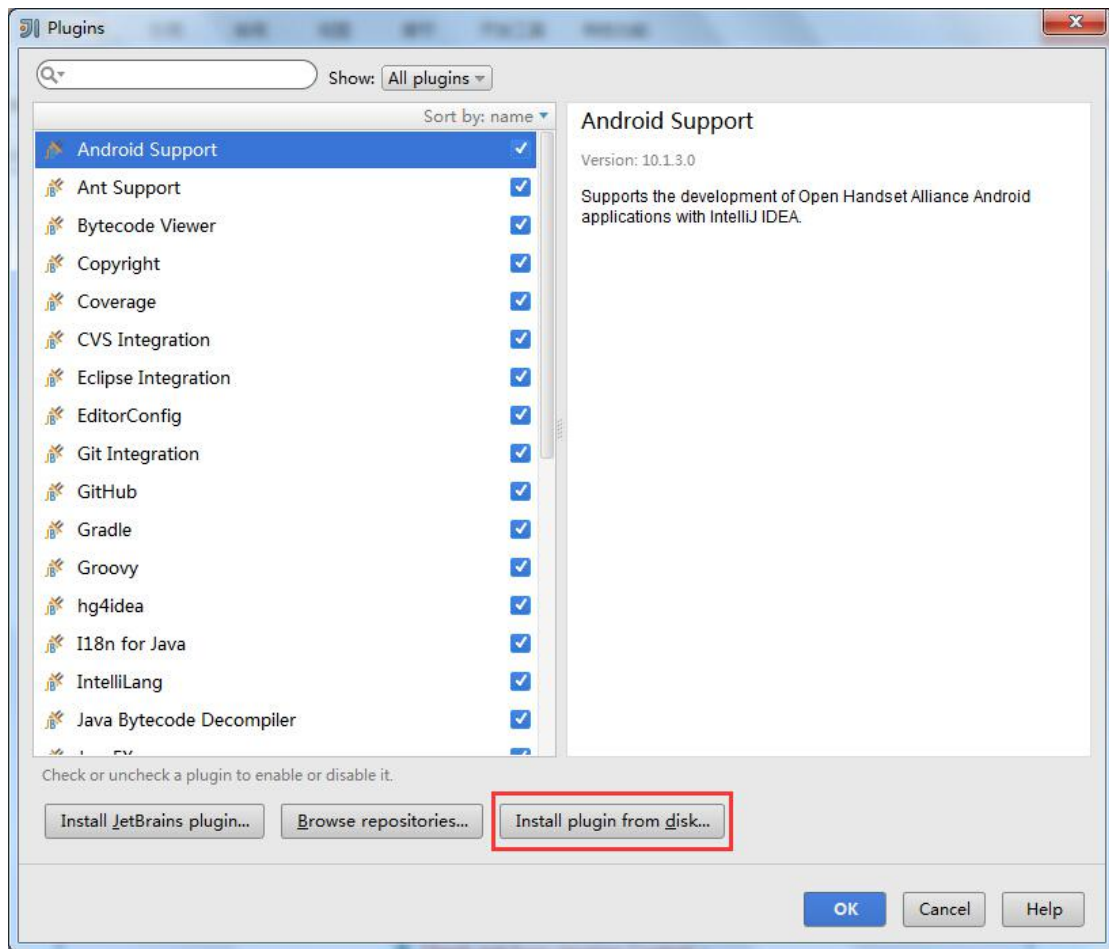
由于 IDEA 的 Scala 插件更优秀，大多数 Scala 程序员都选择 IDEA，可以到 <http://www.jetbrains.com/idea/download/> 下载社区免费版，点击下一步安装即可，安装时如果有网络可以选择在线安装 Scala 插件。这里我们使用离线安装 Scala 插件：

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



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





4. Scala 基础

4.1. 声明变量

```
package cn.itcast.scala

/**
 * Created by ZX on 2015/11/6.
 */
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 和 Double（无包装类型）和一个 Boolean 类型

4.3. 条件表达式

Scala 的条件表达式比较简洁，例如：

```
package cn.itcast.scala

/**
 * Created by ZX on 2015/11/7.
 */
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, 写做 (), 相当于 Java 中的 void
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

```

package cn.itcast.scala

/**
 * Created by ZX on 2015/11/7.
 */
object BlockExpressionDemo {
  def main(args: Array[String]) {
    val x = 0
    //在 scala 中 {} 中块包含一系列表达式, 块中最后一个表达式的值就是块的值
    //下面就是一个块表达式
    val result = {
      if (x < 0) {
        -1
      } else if (x >= 1) {

```

```

    1
  } else {
    "error"
  }
}
//result 的值就是块表达式的结果
println(result)
}
}

```

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

/**
 * Created by ZX on 2015/11/7.
 */
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

```
scala> def m1(x: Int, y: Int) : Int = x * y  
m1: (x: Int, y: Int)Int  
scala>
```

定义方法用def关键字

m1是方法名称

x和y是参数列表

方法返回值类型

方法体

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

```
scala> val f1 = (x: Int, y: Int) => x + y
f1: (Int, Int) => Int = <function2>

scala> f1(1, 2)
res1: Int = 3

scala>
```

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

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> def m2(f: (Int, Int) => Int) = f(2, 6)
m2: (f: (Int, Int) => Int)Int 1.定义一个方法

scala> val f2 = (x: Int, y: Int) => x - y
f2: (Int, Int) => Int = <function2> 2. 定义一个函数

scala> m2(f2) 3.将函数作为参数传入到方法中
res0: Int = -4
```

```
package cn.itcast.scala
```

```
/**
```

```
 * Created by ZX on 2015/11/11.
```

```
 */
```

```
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）

```

scala> def m1(x: Int, y: Int) : Int = x * y
m1: (x: Int, y: Int)Int 方法

scala> val f1 = m1 _
f1: (Int, Int) => Int = <function2> 函数
      ^
      |
      | 神奇的下划线将m1这个方法变成了函数

```

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

```
package cn.itcast.scala
```



```

import scala.collection.mutable.ArrayBuffer

/**
 * Created by ZX on 2015/11/11.
 */
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 会将数组转换成长数组缓冲
        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
    }
}

```

```
ab.insert(0, -1, 0)
//删除数组某个位置的元素用 remove
ab.remove(8, 2)
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

/**
 * Created by ZX on 2015/11/12.
 */
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

/**
 * Created by ZX on 2015/11/12.
 */
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

```
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

```
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，
scores: scala.collection.mutable.Map[String,Int] = Map(tom -> 80, jerry -> 99) 但是Map中的内容可变

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

```
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

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

scala> val r1 = t._1
r1: String = hadoop

scala> val r2 = t._2
r2: Double = 3.14

scala>
```

获取元组中的元素可以使用下划线加脚标
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

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

scala> arr.toMap
res2: scala.collection.immutable.Map[String,Int] = Map(tom -> 88, jerry -> 95)
```

toMap可以将对偶的集合转换成映射

5.3.4. 拉链操作

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 ns = names.zip(scores)
ns: Array[(String, Int)] = Array((tom,88), (jerry,95), (kitty,80))

scala> ns.toMap
res4: scala.collection.immutable.Map[String,Int] = Map(tom -> 88, jerry -> 95, kitty -> 80)
```

使用zip将对应的值绑定到一起

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

5.4. 集合

Scala 的集合有三大类：序列 **Seq**、集 **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. 序列

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))`

```
package cn.itcast.collect

object ImmutListDemo {

  def main(args: Array[String]) {
    //创建一个不可变的集合
    val lst1 = List(1, 2, 3)
    //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 lst1 插入到 lst0 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(lst9)
  }
}
```

可变的序列 `import scala.collection.mutable._`

```
package 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 中的元素追加到 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
}
```

5.5.Set

不可变的 Set

```
package cn.itcast.collect
import scala.collection.immutable.HashSet

object ImmutSetDemo extends App{
  val set1 = new HashSet[Int]()
  //将元素和 set1 合并生成一个新的 set，原有 set 不变
  val set2 = set1 + 4
  //set 中元素不能重复
  val set3 = set1 ++ Set(5, 6, 7)
  val set0 = Set(1,3,4) ++ set1
  println(set0.getClass)
}
```

可变的 Set

```
package cn.itcast.collect
import scala.collection.mutable

object MutSetDemo extends App{
  //创建一个可变的 HashSet
  val set1 = new mutable.HashSet[Int]()
  //向 HashSet 中添加元素
  set1 += 2
  //add 等价于 +=
  set1.add(4)
  set1 ++= Set(1, 3, 5)
  println(set1)
  //删除一个元素
  set1 -= 5
  set1.remove(2)
  println(set1)
}
```

5.6. Map

```
package cn.itcast.collect
import scala.collection.mutable

object MutMapDemo extends App{
  val map1 = new mutable.HashMap[String, Int]()
  //向 map 中添加数据
  map1("spark") = 1
  map1 += (("hadoop", 2))
  map1.put("storm", 3)
  println(map1)

  //从 map 中移除元素
  map1 -= "spark"
  map1.remove("hadoop")
  println(map1)
}
```

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

```
//在 Scala 中，类并不用声明为 public。  
//Scala 源文件中可以包含多个类，所有这些类都具有公有可见性。  
class Person {  
  //用 val 修饰的变量是只读属性，有 getter 但没有 setter  
  //（相当与 Java 中用 final 修饰的变量）  
  val id = "9527"  
  
  //用 var 修饰的变量既有 getter 又有 setter  
  var age: Int = 18  
  
  //类私有字段，只能在类的内部使用  
  private var name: String = "唐伯虎"  
  
  //对象私有字段，访问权限更加严格的，Person 类的方法只能访问到当前对象的字段  
  private[this] val pet = "小强"  
}
```

6.1.2. 构造器

注意：主构造器会执行类定义中的所有语句

```
/**  
 *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, val age: Int){  
  //主构造器会执行类定义中的所有语句  
  println("执行主构造器")  
  
  try {  
    println("读取文件")  
    throw new IOException("io exception")  
  } catch {  

```

```

    case e: NullPointerException => println("打印异常 Exception : " + e)
    case e: IOException => println("打印异常 Exception : " + e)
  } finally {
    println("执行 finally 部分")
  }

  private var gender = "male"

  //用 this 关键字定义辅助构造器
  def this(name: String, age: Int, gender: String) {
    //每个辅助构造器必须以主构造器或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
  }
}

```

```

/**
 *构造器参数可以不带 val 或 var，如果不带 val 或 var 的参数至少被一个方法所使用，
 *那么它将会被提升为字段
 */
//在类名后面加 private 就变成了私有的
class Queen private(val name: String, prop: Array[String], private var age: Int = 18) {

  println(prop.size)

  //prop 被下面的方法使用后，prop 就变成了不可变对象私有字段，等同于 private[this] val prop
  //如果没有被方法使用该参数将不被保存为字段，仅仅是一个可以被主构造器中的代码访问的普通参数
  def description = name + " is " + age + " years old with " + prop.toBuffer
}

object Queen {
  def main(args: Array[String]) {
    //私有的构造器，只有在其伴生对象中使用
    val q = new Queen("hatano", Array("蜡烛", "皮鞭"), 20)
    println(q.description())
  }
}

```

6.2. 对象

6.2.1. 单例对象

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

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

```
package cn.itcast.scala

import scala.collection.mutable.ArrayBuffer

/**
 * Created by ZX on 2015/11/14.
 */
object SingletonDemo {
  def main(args: Array[String]) {
    //单例对象，不需要 new，用【类名.方法】调用对象中的方法
    val session = SessionFactory.getSession()
    println(session)
  }
}

object SessionFactory{
  //该部分相当于 java 中的静态块
  var counts = 5
  val sessions = new ArrayBuffer[Session]()
  while(counts > 0){
    sessions += new Session
    counts -= 1
  }

  //在 object 中的方法相当于 java 中的静态方法
  def getSession(): Session = {
    sessions.remove(0)
  }
}

class Session{
}
```


6.2.2. 伴生对象

在 Scala 的类中，与类名相同的对象叫做**伴生对象**，类和伴生对象之间可以相互访问私有的方法和属性

```
package cn.itcast.scala

/**
 * Created by ZX on 2015/11/14.
 */
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()
  }
}
```

6.2.3. apply 方法

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

```
package cn.itcast.scala
```

```
/**
```

```

* Created by ZX on 2015/11/14.
*/
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)
    println(arr1.toBuffer)

    //new 了一个长度为 5 的 array, 数组里面包含 5 个 null
    var arr2 = new Array(5)
  }
}

```

6.2.4. 应用程序对象

Scala 程序都必须从一个对象的 main 方法开始，可以通过扩展 App 特质，不写 main 方法。

```

package cn.itcast.scala

/**
 * Created by ZX on 2015/11/14.
 */
object AppObjectDemo extends App{
  //不用写 main 方法
  println("I love you Scala")
}

```

6.3. 继承

6.3.1. 扩展类

在 Scala 中扩展类的方式和 Java 一样都是使用 extends 关键字

6.3.2. 重写方法

在 Scala 中重写一个非抽象的方法必须使用 override 修饰符

6.3.3. 类型检查和转换

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

6.3.4. 超类的构造

```
package cn.itcast.scala

/**
 * Created by ZX on 2015/11/10.
 */
object ClazzDemo {
  def main(args: Array[String]) {
    //val h = new Human
    //println(h.fight)
  }
}

trait Flyable{
  def fly(): Unit ={
    println("I can fly")
  }

  def fight(): String
}

abstract class Animal {
  def run(): Int
  val name: String
}

class Human extends Animal with Flyable{

  val name = "abc"

  //打印几次"ABC"?
  val t1, t2, (a, b, c) = {
    println("ABC")
    (1, 2, 3)
  }
}
```

```

println(a)
println(tl._1)

//在 Scala 中重写一个非抽象方法必须用 override 修饰
override def fight(): String = {
    "fight with 棒子"
}
//在子类中重写超类的抽象方法时，不需要使用 override 关键字，写了也可以
def run(): Int = {
    1
}
}

```

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("YoshizawaAkiho", "YuiHatano", "AoiSola")
    val name = arr(Random.nextInt(arr.length))
    name match {
        case "YoshizawaAkiho" => println("吉泽老师...")
        case "YuiHatano" => println("波多老师...")
        case _ => println("真不知道你们在说什么...")
    }
}

```

7.2. 匹配类型

```

package cn.itcast.cases

```

```
import scala.util.Random

object CaseDemo01 extends App{
  //val v = if(x >= 5) 1 else if(x < 2) 2.0 else "hello"
  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

```
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 = (2, 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 中样例类是一中特殊的类，可用于模式匹配。`case class` 是多例的，后面要跟构造参数，`case object` 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")//前面需要加上 s, $id 直接取 id 的值
    }
    case HeartBeat(time) => {
      println(time)
    }
    case CheckTimeOutTask => {
      println("check")
    }
  }
}

```

7.5. Option 类型

在 Scala 中 Option 类型样例类用来表示可能存在或也可能不存在的值(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` 代表参数类型，`B` 代表返回类型，常用作输入模式匹配

```

package cn.itcast.cases

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"))
  }
}

```