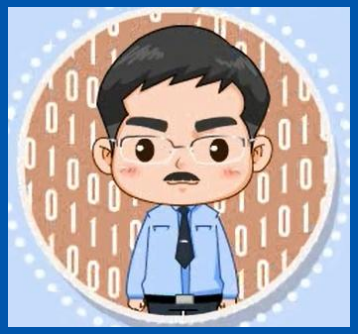


厦门大学研究生课程

《大数据处理技术Spark》

http://dblab.xmu.edu.cn/post/7659/

温馨提示：编辑幻灯片母版，可以修改每页PPT的厦大校徽和底部文字

第2章 Scala语言基础

（PPT版本号：2017年春季学期）

扫一扫访问班级主页

林子雨

厦门大学计算机科学系

E-mail: ziyulin@xmu.edu.cn

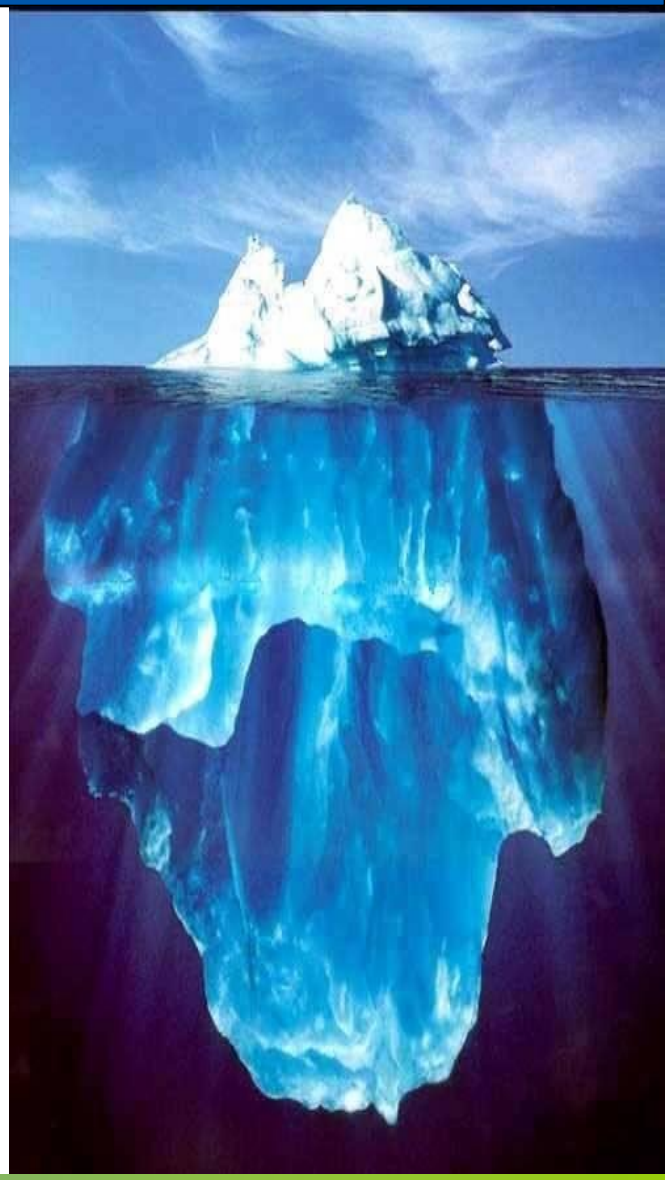
主页：**http://www.cs.xmu.edu.cn/linziyu**

提纲

- 2.1 Scala语言概述
- 2.2 Scala基础
- 2.3 面向对象编程基础
- 2.4 函数式编程基础

厦门大学林子雨

免费在线教程：http://dblab.xmu.edu.cn/blog/spark/



2.1 Scala语言概述

2.1.1 计算机的缘起
2.1.2 编程范式
2.1.3 Scala简介

2.1.1 计算机的缘起

•数学家阿隆佐•邱奇(Alonzo Church)设计了“λ演算”，这是一套用于研究函数定义、函数应用和递归的形式系统

•λ演算被视为最小的通用程序设计语言

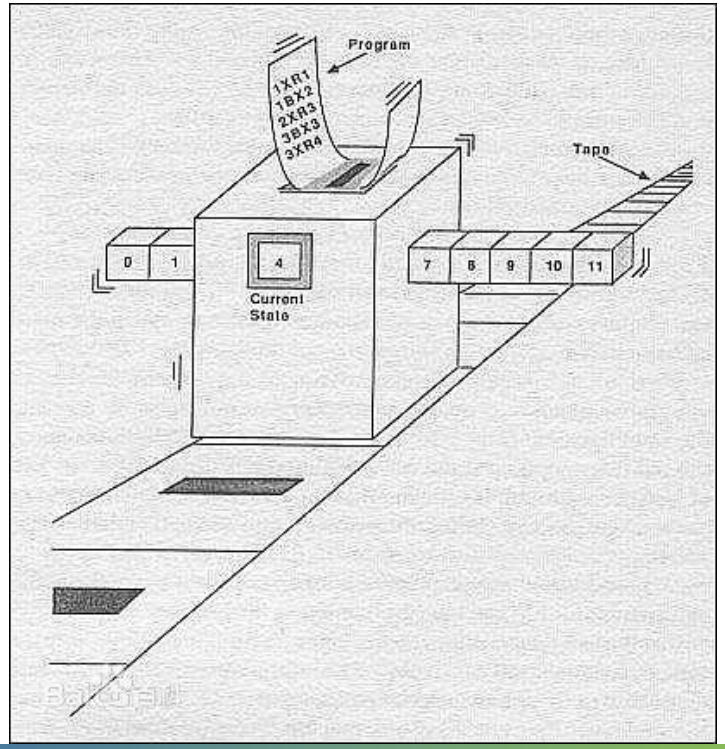
•λ演算的通用性就体现在，任何一个可计算函数都能用这种形式来表达和求值

•λ演算是一个数理逻辑形式系统，强调的是变换规则的运用，而非实现它们的具体机器

2.1.1 计算机的缘起

•英国数学家阿兰·图灵采用了完全不同的设计思路，提出了一种全新的抽象计算模型——图灵机

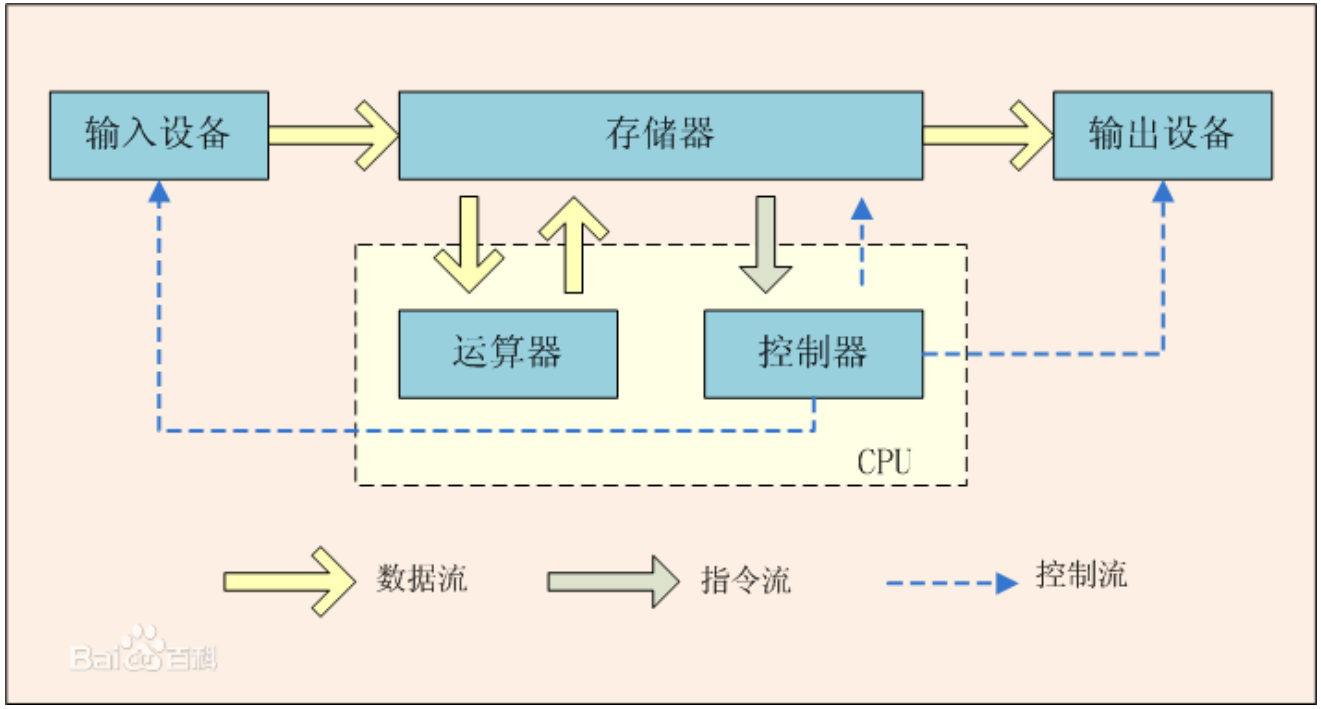
•图灵机是现代计算机的鼻祖。现有理论已经证明，λ演算和图灵机的计算能力是等价的



2.1.1 计算机的缘起

•冯·诺依曼（John Von Neumann）将图灵的理论物化成为实际的物理实体，成为了计算机体系结构的奠基者

•1945年6月，冯·诺依曼提出了在数字计算机内部的存储器中存放程序的概念，这是所有现代计算机的范式，被称为“冯·诺依曼结构”



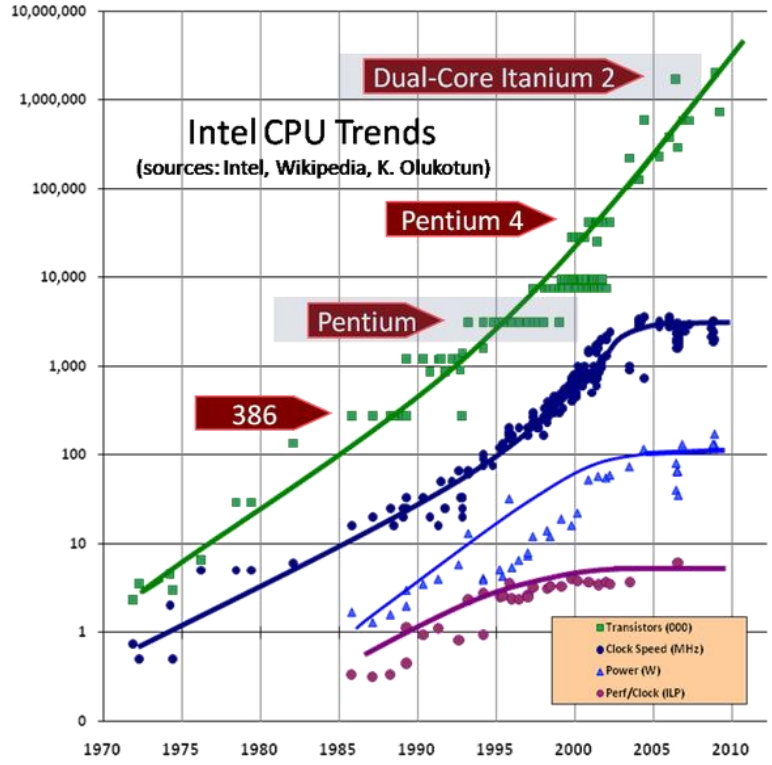
2.1.2 编程范式

- 编程范式是指计算机编程的基本风格或典范模式。常见的编程范式主要包括命令式编程和函数式编程。面向对象编程就属于命令式编程，比如C++、Java等
- 命令式语言是植根于冯·诺依曼体系的，一个命令式程序就是一个冯·诺依曼机的指令序列，给机器提供一条又一条的命令序列让其原封不动地执行
- 函数式编程，又称泛函编程，它将计算机的计算视为数学上的函数计算
- 函数编程语言最重要的基础是λ演算，λ演算对函数式编程特别是Lisp语言有着巨大的影响。典型的函数式语言包括Haskell、Erlang和Lisp等

2.1.2 编程范式

•一个很自然的问题是，既然已经有了命令式编程，为什么还需要函数式编程呢？

•为什么在C++、Java等命令式编程流行了很多年以后，近些年函数式编程会迅速升温呢？



•命令式编程涉及多线程之间的状态共享，需要锁机制实现并发控制

•函数式编程不会在多个线程之间共享状态，不需要用锁机制，可以更好并行处理，充分利用多核CPU并行处理能力

2.1.3 Scala简介

Scala是一门类Java的多范式语言，它整合了面向对象编程和函数式编程的最佳特性。具体来讲：

- Scala运行于Java虚拟机（JVM）之上，并且兼容现有的Java程序
- Scala是一门纯粹的面向对象的语言
- Scala也是一门函数式语言

2.1.4 Scala的安装和使用方法

•Scala运行于Java虚拟机（JVM）之上，因此只要安装有相应的Java虚拟机，所有的操作系统都可以运行Scala程序，包括Window、Linux、Unix、 Mac OS等。

•2.1.4.1 安装Java
•2.1.4.2 安装Scala
•2.1.4.3 使用Scala解释器
•2.1.4.4 第1个Scala程序：HelloWorld

•具体可以参照厦门大学数据库实验室网站博客：
•http://dblab.xmu.edu.cn/blog/929-2/

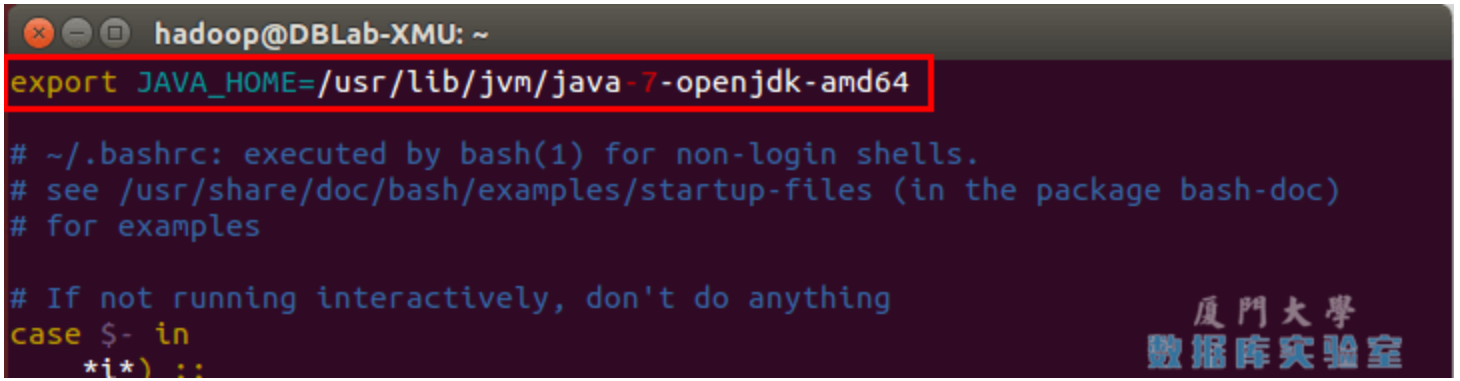
2.1.4.1 安装Java

直接通过命令安装 OpenJDK 7

```
$ sudo apt-get install openjdk-7-jre openjdk-7-jdk
```

配置 JAVA_HOME 环境变量

```
$ vim ~/.bashrc
```



使配置立即生效：

```
$ source ~/.bashrc     # 使变量设置生效
```

2.1.4.2 安装Scala

登录Scala官网，下载scala-2.11.8.tgz

```
$ sudo tar -zxf ~/下载/scala-2.11.8.tgz -C /usr/local   # 解压到/usr/local中
$ cd /usr/local/
$ sudo mv ./scala-2.11.8/ ./scala          # 将文件夹名改为scala
$ sudo chown -R hadoop ./scala        # 修改文件权限，用hadoop用户拥有对scala目录的权限
```

把scala命令添加到path环境变量中

```
$ vim ~/.bashrc
```

```
export PATH=$PATH:/usr/local/scala/bin
```

启动Scala解释器：

```
$ scala
```

```
scala>   //可以在命令提示符后面输入命令
```

2.1.4.3 使用Scala解释器

在Shell命令提示符界面中输入“scala”命令后，会进入scala命令行提示符状态：

```
scala>    //可以在命令提示符后面输入命令
```

```
scala> 8*2+5
res0: Int = 21
```

可以使用命令“:quit”退出Scala解释器，如下所示：

```
scala>:quit
```

2.1.4.4 第1个Scala程序：HelloWorld

```
$ cd /usr/local/scala/mycode
$ vim test.scala
```

```
object HelloWorld {
    def main(args: Array[String]){
        println("Hello, World!")
    }
}
```

```
$ scalac test.scala //编译的时候使用的是Scala文件名称
$ scala -classpath . HelloWorld  //执行的时候使用的是HelloWorld对象名称
```

注意，上面命令中一定要加入"-classpath .", 否则会出现“No such file or class on classpath: HelloWorld”

2.2 Scala基础

2.2.1 基本语法
2.2.2 控制结构
2.2.3 数据结构
2.2.4 面向对象编程基础
2.2.5 函数式编程基础

2.2.1 基本语法

2.2.1.1 声明值和变量
2.2.1.2 基本数据类型和操作
2.2.1.3 Range
2.2.1.4 控制台输入输出语句
2.2.1.5 读写文件
2.2.1.6 异常处理

2.2.1.1 声明值和变量

Scala有两种类型的变量：

•val：是不可变的，在声明时就必须被初始化，而且初始化以后就不能再赋值；

•var：是可变的，声明的时候需要进行初始化，初始化以后还可以再次对其赋值。

2.2.1.1 声明值和变量

```
scala> val myStr = "Hello World!"
myStr: String = Hello World!
```

```
scala> val myStr2 : String = "Hello World!"
myStr2: String = Hello World!
```

```
scala> val myStr3 : java.lang.String = "Hello World!"
myStr3: String = Hello World!
```

```
import java.lang._ //java.lang包里面所有的东西
```

```
scala> println(myStr)
Hello World!
```

```
scala> myStr = "Hello Scala!"
<console>:27: error: reassignment to val
         myStr = "Hello Scala!"
                 ^
```

2.2.1.1 声明值和变量

```
scala> var myPrice : Double = 9.9
myPrice: Double = 9.9
```

```
scala> myPrice = 10.6
myPrice: Double = 10.6
```

2.2.1.1 声明值和变量

小技巧：如何在**Scala**解释器中输入多行代码

```
scala> 2*9+4
res1: Int = 22
```

```
scala> val myStr4 =
     | "Hello World!"
myStr4: String = Hello World!
```

```
scala> val myStr5 =
     |
     |
You typed two blank lines. Starting a new command.
scala>
```

2.2.1.2 基本数据类型和操作

•Scala的数据类型包括：Byte、Char、Short、Int、Long、Float、Double和Boolean

•和Java不同的是，在Scala中，这些类型都是“类”，并且都是包scala的成员，比如，Int的全名是scala.Int。对于字符串，Scala用java.lang.String类来表示字符串

值类型	范围
Byte	8 位有符号补码整数（$-2^7 \sim 2^7-1$）
Short	16 位有符号补码整数（$-2^{15} \sim 2^{15}-1$）
Int	32 位有符号补码整数（$-2^{31} \sim 2^{31}-1$）
Long	64 位有符号补码整数（$-2^{63} \sim 2^{63}-1$）
Char	16 位无符号Unicode字符（$0 \sim 2^{16}-1$）
String	字符序列
Float	32 位 IEEE754 单精度浮点数
Double	64 位 IEEE754 单精度浮点数
Boolean	true 或 false

2.2.1.2 基本数据类型和操作

字面量（literal）

```
val i = 123  //123就是整数字面量
val i = 3.14 //3.14就是浮点数字面量
val i = true //true就是布尔型字面量
val i = 'A' //'A'就是字符字面量
val i = "Hello" //"Hello"就是字符串字面量
```

2.2.1.2 基本数据类型和操作

操作符：在Scala中，可以使用加(+)、减(-) 、乘(*) 、除(/) 、余数（%）等操作符，而且，这些操作符就是方法。例如，5 + 3和(5).+(3)是等价的，也就是说：

`a 方法 b` 等价于 `a.方法(b)`

前者是后者的简写形式，这里的+是方法名，是Int类中的一个方法。

```
scala> val sum1 = 5 + 3 //实际上调用了 (5).+(3)
sum1: Int = 8
scala> val sum2 = (5).+(3) //可以发现，写成方法调用的形式，和上面得到相同的结果
sum2: Int = 8
```

和Java不同，在Scala中并没有提供++和--操作符，当需要递增和递减时，可以采用如下方式表达：

```
scala> var i = 5;
i: Int = 5
scala> i += 1  //将i递增
scala> println(i)
6
```

2.2.1.2 基本数据类型和操作

富包装类

- 对于基本数据类型，除了以上提到的各种操作符外，Scala还提供了许多常用运算的方法，只是这些方法不是在基本类里面定义，还是被封装到一个对应的富包装类中
- 每个基本类型都有一个对应的富包装类，例如Int有一个RichInt类、String有一个RichString类，这些类位于包scala.runtime中
- 当对一个基本数据类型的对象调用其富包装类提供的方法，Scala会自动通过隐式转换将该对象转换为对应的富包装类型，然后再调用相应的方法。例如：3 max 5

2.2.1.3 Range

•在执行for循环时，我们经常会用到数值序列，比如，i的值从1循环到5，这时就可以采用Range来实现
•Range可以支持创建不同数据类型的数值序列，包括Int、Long、Float、Double、Char、BigInt和BigDecimal等

（1）创建一个从1到5的数值序列，包含区间终点5，步长为1

```
scala> 1 to 5
res0: scala.collection.immutable.Range.Inclusive = Range(1, 2, 3, 4, 5)
```

```
scala> 1.to(5)
res0: scala.collection.immutable.Range.Inclusive = Range(1, 2, 3, 4, 5)
```

2.2.1.3 Range

（2）创建一个从1到5的数值序列，不包含区间终点5，步长为1

```
scala> 1 until 5
res1: scala.collection.immutable.Range = Range(1, 2, 3, 4)
```

（3）创建一个从1到10的数值序列，包含区间终点10，步长为2

```
scala> 1 to 10 by 2
res2: scala.collection.immutable.Range = Range(1, 3, 5, 7, 9)
```

（4）创建一个Float类型的数值序列，从0.5f到5.9f，步长为0.3f

```
scala> 0.5f to 5.9f by 0.8f
res3: scala.collection.immutable.NumericRange[Float] = NumericRange(0.5, 1.3, 2.1, 2.899
9999, 3.6999998, 4.5, 5.3)
```

2.2.1.4控制台输入输出语句

•为了从控制台读写数据，可以使用以read为前缀的方法，包括：readInt、readDouble、readByte、readShort、readFloat、readLong、readChar readBoolean及readLine，分别对应9种基本数据类型，其中前8种方法没有参数，readLine可以不提供参数，也可以带一个字符串参数的提示

•所有这些函数都属于对象scala.io.StdIn的方法，使用前必须导入，或者直接用全称进行调用

2.2.1.4控制台输入输出语句

```
scala> import io.StdIn._
import io.StdIn._

scala> var i=readInt()
54
i: Int = 54

scala> var f=readFloat()
3.5e1
f: Float = 35.0

scala> var b=readBoolean()
true
b: Boolean = true

scala> var str=readLine("please input your name:")
please input your name:xiao zhang
str: String = xiao zhang

scala>
```

2.2.1.4控制台输入输出语句

为了向控制台输出信息，常用的两个函数是print()和println()，可以直接输出字符串或者其它数据类型

```
scala> var i=345
i: Int = 345

scala> print("i=");print(i)
i=345
scala> println("hello");println("world!")
hello
world!

scala>
```

2.2.1.4控制台输入输出语句

Scala还带有C语言风格的格式化字符串的printf()函数

```
scala> val i=36
i: Int = 36

scala> val f=56.5
f: Double = 56.5

scala> printf("My name is %s. I am  %d years old and %.1f Kg.\n","Xiao Ming",i,f)
My name is Xiao Ming. I am  36 years old and 56.5 Kg.

scala>
```

print()、println()和printf() 都在对象Predef中定义，该对象默认情况下被所有Scala程序引用，因此可以直接使用Predef对象提供的方法，而无需使用scala.Predef.的形式。

2.2.1.5读写文件

写入文件

Scala需要使用java.io.PrintWriter实现把数据写入到文件

```
scala> import java.io.PrintWriter
import java.io.PrintWriter //这行是Scala解释器执行上面语句后返回的结果
scala> val out = new PrintWriter("output.txt")
out: java.io.PrintWriter = java.io.PrintWriter@25641d39  //这行是Scala解释器执行上面语句后返回的结果
scala> for (i <- 1 to 5) out.println(i)
scala> out.close()
```

如果我们想把文件保存到一个指定的目录下，就需要给出文件路径

```
scala> import java.io.PrintWriter
import java.io.PrintWriter //这行是Scala解释器执行上面语句后返回的结果
scala> val out = new PrintWriter("/usr/local/scala/mycode/output.txt")
out: java.io.PrintWriter = java.io.PrintWriter@25641d39  //这行是Scala解释器执行上面语句后返回的结果
scala> for (i <- 1 to 5) out.println(i)
scala> out.close()
```

2.2.1.5读写文件

读取文件

可以使用Scala.io.Source的getLines方法实现对文件中所有行的读取

```
scala> import scala.io.Source
import scala.io.Source //这行是Scala解释器执行上面语句后返回的结果
scala> val inputFile = Source.fromFile("output.txt")
inputFile: scala.io.BufferedSource = non-empty iterator  //这行是Scala解释器执行上面语句后返回的结果
scala> val lines = inputFile.getLines //返回的结果是一个迭代器
lines: Iterator[String] = non-empty iterator  //这行是Scala解释器执行上面语句后返回的结果
scala> for (line <- lines) println(line)
1
2
3
4
5
```

2.2.1.6 异常处理

Scala不支持Java中的“受检查异常”(checked exception)，将所有异常都当作“不受检异常”（或称为运行时异常）

Scala仍使用try-catch结构来捕获异常

```
import java.io.FileReader
import java.io.FileNotFoundException
import java.io.IOException
try {
  val f = new FileReader("input.txt")
        // 文件操作
} catch {
  case ex: FileNotFoundException =>
        // 文件不存在时的操作
  case ex: IOException =>
   // 发生I/O错误时的操作
} finally {
  file.close() // 确保关闭文件
}
```

2.2.2 控制结构

2.2.2.1 if条件表达式
2.2.2.2 while循环
2.2.2.3 for循环

2.2.2.1 if条件表达式

```
val x = 6
if (x>0) {println("This is a positive number")
} else {
    println("This is not a positive number")
}
```

```
val x = 3
if (x>0) {
    println("This is a positive number")
} else if (x==0) {
    println("This is a zero")
} else {
    println("This is a negative number")
}
```

有一点与Java不同的是，Scala中的if表达式的值可以赋值给变量

```
val x = 6
val a = if (x>0) 1 else -1
```

2.2.2.2 while循环

```
var i = 9
while (i > 0) {
    i -= 1
    printf("i is %d\n",i)
}
```

```
var i = 0
do {
    i += 1
    println(i)
}while (i<5)
```

2.2.2.3 for循环

Scala中的for循环语句格式如下：

```
for (变量<-表达式) 语句块
```

其中，“变量<-表达式”被称为“生成器（generator）”

```
for (i <- 1 to 5) println(i)
```

```
1
2
3
4
5
```

```
for (i <- 1 to 5 by 2) println(i)
```

```
1
3
5
```

2.2.2.3 for循环

- 不希望打印出所有的结果，过滤出一些满足制定条件的结果，需要使用到称为“守卫(guard)”的表达式
- 比如，只输出1到5之中的所有偶数，可以采用以下语句：

```
for (i <- 1 to 5 if i%2==0) println(i)
```

```
2
4
```

2.2.2.3 for循环

Scala也支持“多个生成器”的情形，可以用分号把它们隔开，比如：

```
for (i <- 1 to 5; j <- 1 to 3) println(i*j)
```

```
1
2
3
2
4
6
3
6
9
4
8
12
5
10
15
```

2.2.2.3 for循环

可以给每个生成器都添加一个“守卫”，如下：

```
for (i <- 1 to 5 if i%2==0; j <- 1 to 3 if j!=i) println(i*j)
```

```
2
6
4
8
12
```

2.2.2.3 for循环

for推导式

- Scala的for结构可以在每次执行的时候创造一个值，然后将包含了所有产生值的集合作为for循环表达式的结果返回，集合的类型由生成器中的集合类型确定
- 通过for循环遍历一个或多个集合，对集合中的元素进行“推导”，从而计算得到新的集合，用于后续的其他处理

for （变量 <- 表达式） **yield {**语句块**}**

```
scala> val r=for (i <- 1 to 5 if i%2==0) yield { println(i); i}
2
4
r: scala.collection.immutable.IndexedSeq[Int] = Vector(2, 4)
```

2.2.3 数据结构

2.2.3.1 容器（Collection）
2.2.3.2 列表（List）
2.2.3.3 集合（Set）
2.2.3.4 映射（Map）
2.2.3.5 迭代器（Iterator）
2.2.3.6 数组（Array）
2.2.3.7 元组（Tuple）

2.2.3.1 容器（collection）

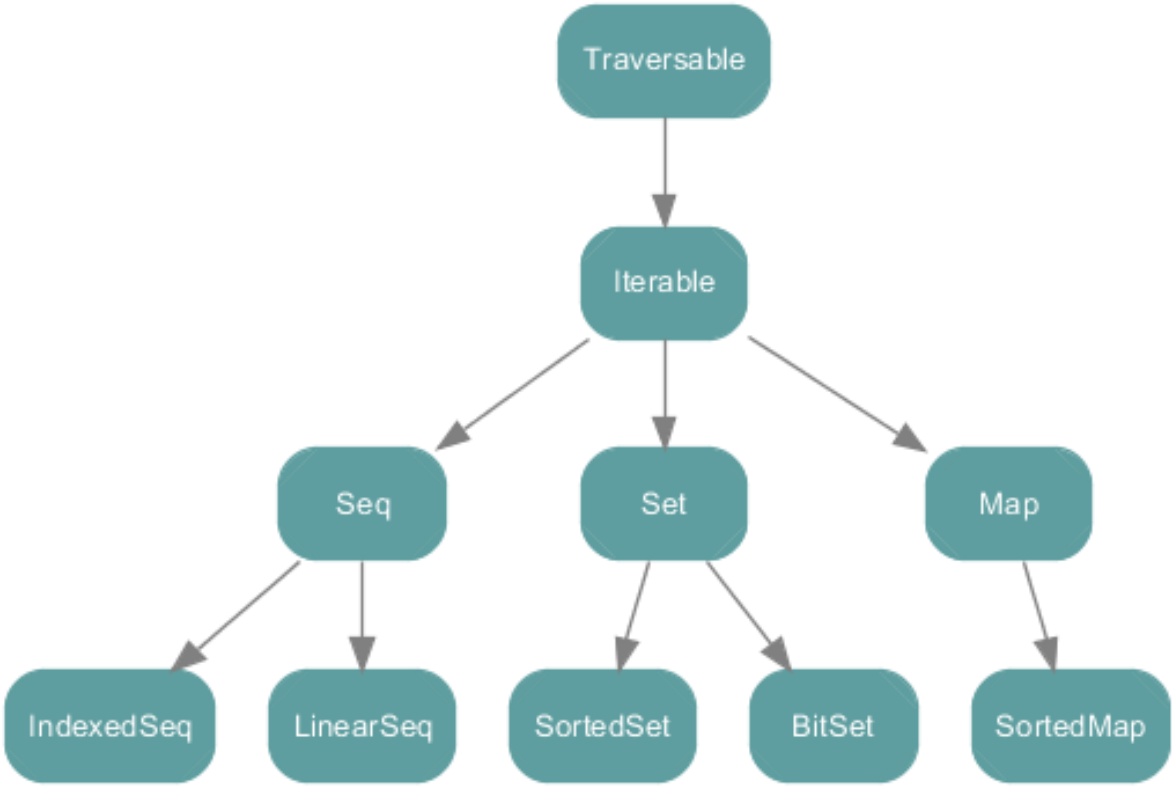
- Scala提供了一套丰富的容器（collection）库，包括列表（List）、数组（Array）、集合（Set）、映射（Map）等

- 根据容器中元素的组织方式和操作方式，可以区分为有序和无序、可变和不可变等不同的容器类别

- Scala用了三个包来组织容器类，分别是scala.collection 、scala.collection.mutable和scala.collection.immutable

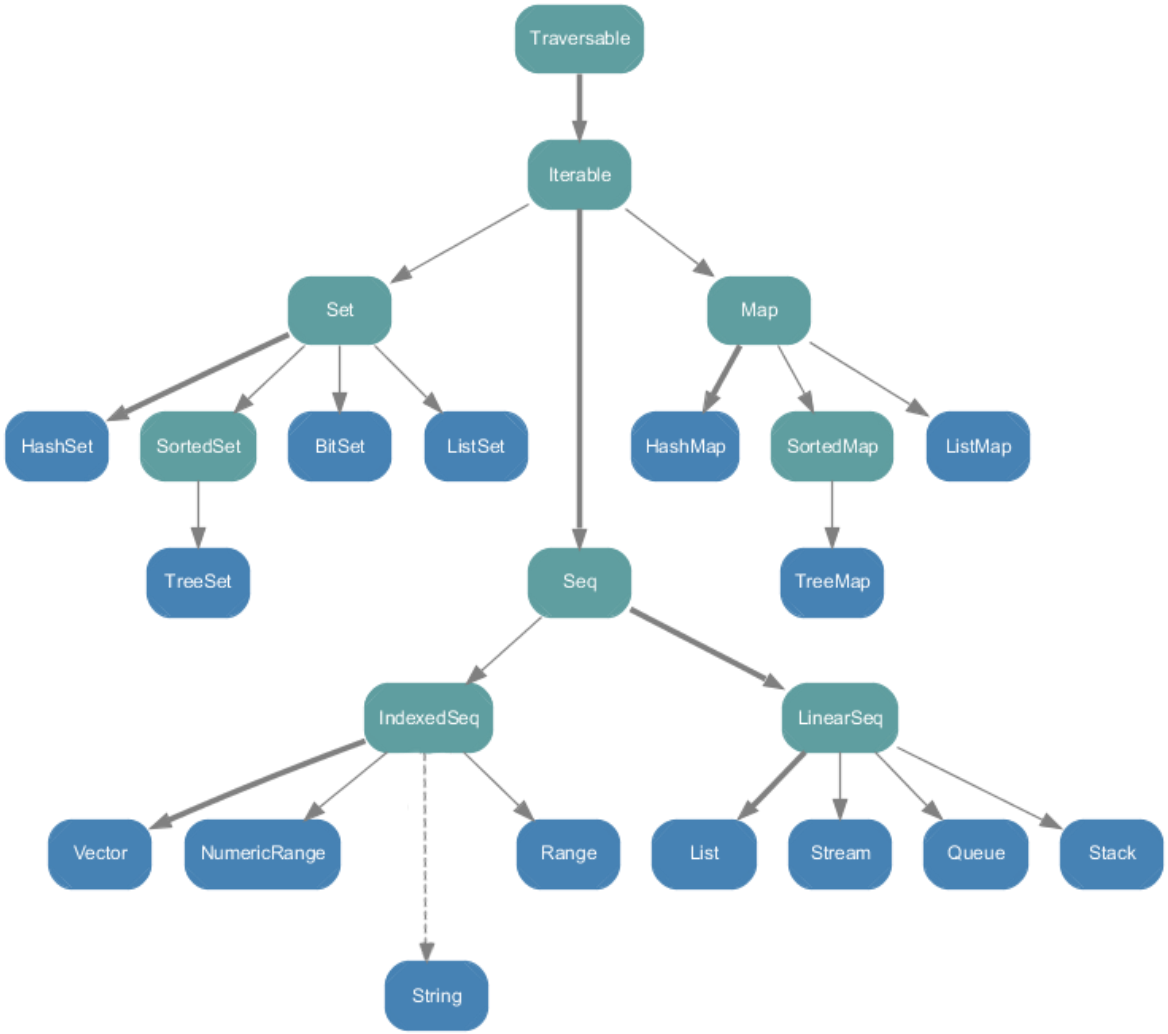
2.2.3.1 容器（collection）

下图显示了scala.collection包中所有的容器类。这些都是高级抽象类或特质。例如，所有容器类的基本特质(trait)是Traverable特质，它为所有的容器类定义了公用的foreach方法，用于对容器元素进行遍历操作



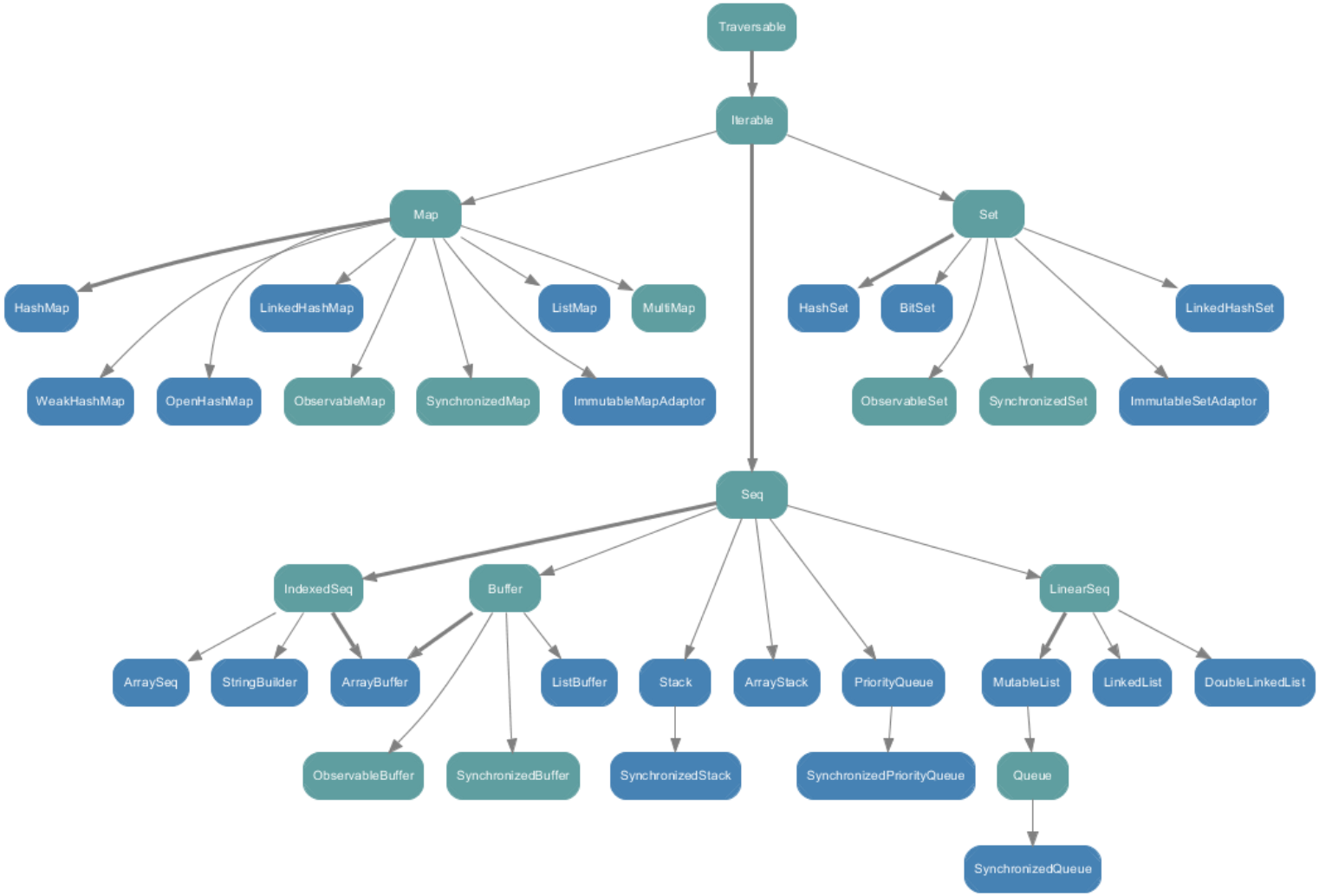
2.2.3.1 容器（collection）

下面的图表显示了scala.collection.immutable中的所有容器类



2.2.3.1 容器（collection）

下面的图表显示scala.collection.mutable中的所有容器类



2.2.3.2 列表（List）

- 列表是一种共享相同类型的不可变的对象序列。既然是一个不可变的集合， Scala的List定义在scala.collection.immutable包中
- 不同于Java的java.util.List，scala的List一旦被定义，其值就不能改变，因此声明List时必须初始化

```
var strList=List("BigData","Hadoop","Spark")
```

- 列表有头部和尾部的概念，可以分别使用head和tail方法来获取
- head返回的是列表第一个元素的值
- tail返回的是除第一个元素外的其它值构成的新列表，这体现出列表具有递归的链表结构
- strList.head将返回字符串”BigData”，strList.tail返回List ("Hadoop","Spark")

2.2.3.2 列表（List）

构造列表常用的方法是通过在已有列表前端增加元素，使用的操作符为::，例如：

val otherList="Apache"::strList

执行该语句后strList保持不变，而otherList将成为一个新的列表：

List("Apache","BigData","Hadoop","Spark")

Scala还定义了一个空列表对象Nil，借助Nil，可以将多个元素用操作符::串起来初始化一个列表

val intList = 1::2::3::Nil　与val intList = List(1,2,3)等效

2.2.3.3 集合（Set）

•集合(set)是不重复元素的容器（collection）。列表中的元素是按照插入的先后顺序来组织的，但是，“集合”中的元素并不会记录元素的插入顺序，而是以“哈希”方法对元素的值进行组织，所以，它允许你快速地找到某个元素

•集合包括可变集和不可变集，分别位于scala.collection.mutable包和scala.collection.immutable包，缺省情况下创建的是不可变集

```
var mySet = Set("Hadoop","Spark")
mySet += "Scala"
```

如果要声明一个可变集，则需要提前引入scala.collection.mutable.Set

```
import scala.collection.mutable.Set
val myMutableSet = Set("Database","BigData")
myMutableSet += "Cloud Computing"
```

2.2.3.4 映射（Map）

•映射(Map)是一系列键值对的容器。在一个映射中，键是唯一的，但值不一定是唯一的。可以根据键来对值进行快速的检索

•和集合一样，Scala 采用了类继承机制提供了可变的和不可变的两种版本的映射，分别定义在包scala.collection.mutable 和scala.collection.immutable 里。默认情况下，Scala中使用不可变的映射。如果想使用可变映射，必须明确地导入scala.collection.mutable.Map

```
val university = Map("XMU" -> "Xiamen University",
"THU" -> "Tsinghua University","PKU"->"Peking
University")
```

2.2.3.4 映射（Map）

如果要获取映射中的值，可以通过键来获取

```
println(university("XMU"))
```

对于这种访问方式，如果给定的键不存在，则会抛出异常，为此，访问前可以先调用contains方法确定键是否存在

```
val xmu = if (university.contains("XMU")) university("XMU") else 0
println(xmu)
```

2.2.3.4 映射（Map）

不可变映射，是无法更新映射中的元素的，也无法增加新的元素。如果要更新映射的元素，就需要定义一个可变的映射

```
import scala.collection.mutable.Map
val university2 = Map("XMU" -> "Xiamen University", "THU" -> "Tsinghua University","PKU"->"P
eking University")
university2("XMU") = "Ximan University" //更新已有元素的值
university2("FZU") = "Fuzhou University" //添加新元素
```

也可以使用+=操作来添加新的元素

```
university2 + = ("TJU"->"Tianjin University") //添加一个新元素
university2 + = ("SDU"->"Shandong University","WHU"->"Wuhan University") //同时添加两个新元素
```

2.2.3.4 映射（Map）

循环遍历映射

```
for ((k,v) <- 映射) 语句块
```

```
for ((k,v) <- university) printf("Code is : %s and name is: %s\n",k,v)
```

```
Code is : XMU and name is: Xiamen University
Code is : THU and name is: Tsinghua University
Code is : PKU and name is: Peking University
```

或者，也可以只遍历映射中的k或者v

```
for (k<-university.keys) println(k)
XMU //打印出的结果
THU //打印出的结果
PKU //打印出的结果
```

```
for (v<-university.values) println(v)
Xiamen University //打印出的结果
Tsinghua University //打印出的结果
Peking University  //打印出的结果
```

2.2.3.5 迭代器（Iterator）

•在Scala中，迭代器（Iterator）不是一个集合，但是，提供了访问集合的一种方法

•迭代器包含两个基本操作：next和hasNext。next可以返回迭代器的下一个元素，hasNext用于检测是否还有下一个元素

```
val iter = Iterator("Hadoop","Spark","Scala")
while (iter.hasNext) {
    println(iter.next())
}
```

```
val iter = Iterator("Hadoop","Spark","Scala")
for (elem <- iter) {
    println(elem)
}
```

2.2.3.5 迭代器（Iterator）

Iterable有两个方法返回迭代器：grouped和sliding。然而，这些迭代器返回的不是单个元素，而是原容器（collection）元素的全部子序列。这些最大的子序列作为参数传给这些方法。grouped方法返回元素的增量分块，sliding方法生成一个滑动元素的窗口。两者之间的差异通过REPL的作用能够清楚看出。

```
scala> val xs = List(1, 2, 3, 4, 5)
xs: List[Int] = List(1, 2, 3, 4, 5)
scala> val git = xs grouped 3
git: Iterator[List[Int]] = non-empty iterator
scala> git.next()
res3: List[Int] = List(1, 2, 3)
scala> git.next()
res4: List[Int] = List(4, 5)
scala> val sit = xs sliding 3
sit: Iterator[List[Int]] = non-empty iterator
scala> sit.next()
res5: List[Int] = List(1, 2, 3)
scala> sit.next()
res6: List[Int] = List(2, 3, 4)
scala> sit.next()
res7: List[Int] = List(3, 4, 5)
```

2.2.3.6 数组（Array）

数组是一种可变的、可索引的、元素具有相同类型的数据集合，它是各种高级语言中最常用的数据结构。Scala提供了参数化类型的通用数组类Array[T]，其中T可以是任意的Scala类型，可以通过显式指定类型或者通过隐式推断来实例化一个数组。

```
val intValueArr = new Array[Int](3)  //声明一个长度为3的整型数组，每个数组元素初始化为0
intValueArr(0) = 12 //给第1个数组元素赋值为12
intValueArr(1) = 45  //给第2个数组元素赋值为45
intValueArr(2) = 33 //给第3个数组元素赋值为33
```

```
val myStrArr = new Array[String](3) //声明一个长度为3的字符串数组，每个数组元素初始化为null
myStrArr(0) = "BigData"
myStrArr(1) = "Hadoop"
myStrArr(2) = "Spark"
for (i <- 0 to 2) println(myStrArr(i))
```

可以不给出数组类型，Scala会自动根据提供的初始化数据来推断出数组的类型

```
val intValueArr = Array(12,45,33)
val myStrArr = Array("BigData","Hadoop","Spark")
```

2.2.3.6 数组（Array）

Array提供了函数ofDim来定义二维和三维数组，用法如下：

```
val myMatrix = Array.ofDim[Int](3,4) //类型实际就是Array[Array[Int]]

val myCube = Array.ofDim[String](3,2,4) //类型实际是Array[Array[Array[Int]]]
```

可以使用多级圆括号来访问多维数组的元素，例如myMatrix(0)(1)返回第一行第二列的元素

2.2.3.6 数组（Array）

采用Array类型定义的数组属于定长数组，其数组长度在初始化后就不能改变。如果要定义变长数组，需要使用ArrayBuffer参数类型，其位于包scala.collection.mutable中。举例如下：

```
import  scala.collection.mutable.ArrayBuffer
val aMutableArr = ArrayBuffer(10,20,30)
aMutableArr += 40
aMutableArr.insert(2, 60,40)
 aMutableArr -= 40
 var temp=aMutableArr.remove(2)
```

2.2.3.7 元组（Tuple）

元组是不同类型的值的聚集。元组和列表不同，列表中各个元素必须是相同类型，而元组可以包含不同类型的元素

```
scala> val tuple = ("BigData",2015,45.0)
tuple: (String, Int, Double) = (BigData,2015,45.0)  //这行是Scala解释器返回的执行结果
scala> println(tuple._1)
BigData
scala> println(tuple._2)
2015
scala> println(tuple._3)
45.0
```

2.3 面向对象编程基础

2.3.1 类
2.3.2 对象
2.3.3 继承
2.3.4 特质
2.3.5 模式匹配

2.3.1 类

2.3.1.1 简单的类
2.3.1.2 给类增加字段和方法
2.3.1.3 创建对象
2.3.1.4 编译和执行
2.3.1.5 getter和setter方法
2.3.1.6 辅助构造器
2.3.1.7主构造器

2.3.1.1 简单的类

最简单的类的定义形式是：

```
class Counter{
    //这里定义类的字段和方法
}
```

可以使用new关键字来生成对象

```
new Counter //或者new Counter()
```

2.3.1.2 给类增加字段和方法

```
class Counter {
    private var value = 0
    def increment(): Unit = { value += 1}
    def current(): Int = {value}
}
```

Unit后面的等号和大括号后面，包含了该方法要执行的具体操作语句

如果大括号里面只有一行语句，那么也可以直接去掉大括号，写成下面的形式：

```
class Counter {
    private var value = 0
    def increment(): Unit = value += 1 //去掉了大括号
    def current(): Int = {value}  //作为对比，这里依然保留大括号
}
```

或者，还可以去掉返回值类型和等号，只保留大括号，如下：

```
class Counter {
    private var value = 0
    def increment() {value += 1} //去掉了返回值类型和等号，只保留大括号
    def current(): Int = {value} //作为对比，这里依然保留原来形式
}
```

2.3.1.3 创建对象

```
class Counter {
    private var value = 0
    def increment(): Unit = { value += 1}
    def current(): Int = {value}
}
```

下面我们新建对象，并调用其中的方法：

```
val myCounter = new Counter
myCounter.increment() //或者也可以不用圆括号，写成myCounter.increment
println(myCounter.current)
```

从上面代码可以看出，Scala在调用无参方法时，是可以省略方法名后面的圆括号的

2.3.1.4 编译和执行

新建一个TestCounter.scala代码文件

```
class Counter {
    private var value = 0
    def increment(): Unit = { value += 1}
    def current(): Int = {value}
}
val myCounter = new Counter
myCounter.increment()
println(myCounter.current)
```

在Linux Shell命令提示符下，使用scala命令执行这个代码文件：

```
$ scala TestCounter.scala
```

上面命令执行后，会在屏幕输出“1”

2.3.1.4 编译和执行

也可以进入到Scala解释器下面去执行TestCounter.scala
首先启动Scala解释器，如下：

```
$ scala //在终端窗口中输入scala命令进入Scala解释器
```

进入scala命令提示符状态以后，可以在里面输入如下命令：

```
scala> :load /usr/local/scala/mycode/TestCounter.scala  //这是输入的命令
Loading /usr/local/scala/mycode/TestCounter.scala... //从这里开始是执行结果
defined class Counter
myCounter: Counter = Counter@17550481
1
```

完成上面操作以后，可以退出Scala解释器，回到Linux系统的Shell命令提示符状态，退出Scala解释器的命令如下：

```
scala> :quit
```

2.3.1.4 编译和执行

下面尝试一下，看看是否可以使用scalac命令对这个TestCounter.scala文件进行编译，如下：

```
$ scalac TestCounter.scala
```

执行上述scalac命令后，会出现一堆错误，无法编译。为什么呢？

原因：声明都没有被封装在对象中，因此，无法编译成JVM字节码

```
class Counter {
    private var value = 0
    def increment(): Unit = { value += 1}
    def current(): Int = {value}
}
val myCounter = new Counter
myCounter.increment()
println(myCounter.current)
```

2.3.1.4 编译和执行

在TestCounterJVM.scala中输入以下代码：

```
class Counter {
    private var value = 0
    def increment(): Unit = { value += 1}
    def current(): Int = {value}
}
object MyCounter{
    def main(args:Array[String]){
        val myCounter = new Counter
        myCounter.increment()
        println(myCounter.current)
    }
}
```

使用scalac命令编译这个代码文件，并用scala命令执行，如下：

```
$ scalac TestCounterJVM.scala
$ scala -classpath . MyCounter //MyCounter是包含main方法的对象名称，这里不能使用文件名称TestCounterJVM
```

上面命令执行后，会在屏幕输出“1”

2.3.1.4 编译和执行

现在我们对之前的类定义继续改进一下，让方法中带有参数。我们可以修改一下TestCounterJVM.scala文件：

```
class Counter {
    private var value = 0
    def increment(step: Int): Unit = { value += step}
    def current(): Int = {value}
}
object MyCounter{
    def main(args:Array[String]){
        val myCounter = new Counter
        myCounter.increment(5) //这里设置步长为5，每次增加5
        println(myCounter.current)
    }
}
```

编译执行这个文件，就可以得到执行结果是5。

2.3.1.5 getter和setter方法

•给类中的字段设置值以及读取值，在Java中是通过getter和setter方法实现的
•在Scala中，也提供了getter和setter方法的实现，但是并没有定义成getXxx和setXxx

继续修改TestCounterJVM.scala文件：

```
class Counter {
    var value = 0 //注意这里没有private修饰符，从而让这个变量对外部可见
    def increment(step: Int): Unit = { value += step}
    def current(): Int = {value}
}
object MyCounter{
    def main(args:Array[String]){
        val myCounter = new Counter
        println(myCounter.value)  //不是用getXxx获取字段的值
        myCounter.value = 3 //不是用setXxx设置字段的值
        myCounter.increment(1) //这里设置步长为1，每次增加1
        println(myCounter.current)
    }
}
```

编译执行这个文件，就可以得到两行执行结果，第一行是0，第二行是4。

2.3.1.5 getter和setter方法

•但是，在Java中，是不提倡设置这种公有(public)字段的，一般都是把value字段设置为private，然后提供getter和setter方法来获取和设置字段的值。那么，到了Scala中该怎么做呢？

•我们先把value字段声明为private，看看会出现什么效果，继续修改TestCounterJVM.scala文件：

```
class Counter {
    private var value = 0  //增加了private修饰符，成为私有字段
    def increment(step: Int): Unit = { value += step}
    def current(): Int = {value}
}
object MyCounter{
    def main(args:Array[String]){
        val myCounter = new Counter
        println(myCounter.value)  //不是用getXxx获取字段的值
        myCounter.value = 3 //不是用setXxx设置字段的值
        myCounter.increment(1) //这里设置步长为1，每次增加1
        println(myCounter.current)
    }
}
```

现在我们去用scalac命令编译上面的代码，就会报错，会出现“error:variable value in class Counter cannot be accessed in Counter”这样的错误信息。因为，value字段前面用了修饰符private，已经成为私有字段，外部是无法访问的。

2.3.1.5 getter和setter方法

value变成私有字段以后，Scala又没有提供getter和setter方法，怎么可以访问value字段呢？解决方案是，在Scala中，可以通过定义类似getter和setter的方法，分别叫做value和value_=，具体如下：

```
class Counter {
    private var privateValue = 0  //变成私有字段，并且修改字段名称
    def value = privateValue //定义一个方法，方法的名称就是原来我们想要的字段的名称
    def value_=(newValue: Int){
        if (newValue > 0) privateValue = newValue //只有提供的新值是正数，才允许修改
    }
    def increment(step: Int): Unit = { value += step}
    def current(): Int = {value}
}
object MyCounter{
    def main(args:Array[String]){
        val myCounter = new Counter
        println(myCounter.value)  //打印value的初始值
        myCounter.value = 3 //为value设置新的值
        println(myCounter.value)  //打印value的新值
        myCounter.increment(1) //这里设置步长为1，每次增加1
        println(myCounter.current)
    }
}
```

编译执行这个文件，就可以得到三行执行结果，第一行是0，第二行是3，第三行是4。

2.3.1.6 辅助构造器

- Scala构造器包含1个主构造器和若干个（0个或多个）辅助构造器
- 辅助构造器的名称为this，每个辅助构造器都必须调用一个此前已经定义的辅助构造器或主构造器
- 下面定义一个带有辅助构造器的类，我们对上面的Counter类定义进行修改：

```
class Counter {
    private var value = 0 //value用来存储计数器的起始值
    private var name = "" //表示计数器的名称
    private var mode = 1 //mode用来表示计数器类型（比如，1表示步数计数器，2表示时间计数器）
    def this(name: String){ //第一个辅助构造器
        this() //调用主构造器
        this.name = name
    }
    def this (name: String, mode: Int){ //第二个辅助构造器
        this(name) //调用前一个辅助构造器
        this.mode = mode
    }
    def increment(step: Int): Unit = { value += step}
    def current(): Int = {value}
    def info(): Unit = {printf("Name:%s and mode is %d\n",name,mode)}
}
```

（代码未完，剩余代码见下一页）

2.3.1.6 辅助构造器

（代码续上一页）

```
object MyCounter{
    def main(args:Array[String]){
        val myCounter1 = new Counter  //主构造器
        val myCounter2 = new Counter("Runner") //第一个辅助构造器，计数器的名称设置为Runner，用
来计算跑步步数
        val myCounter3 = new Counter("Timer",2) //第二个辅助构造器，计数器的名称设置为Timer，用
来计算秒数
        myCounter1.info  //显示计数器信息
        myCounter1.increment(1)     //设置步长
        printf("Current Value is: %d\n",myCounter1.current) //显示计数器当前值
        myCounter2.info  //显示计数器信息
        myCounter2.increment(2)     //设置步长
        printf("Current Value is: %d\n",myCounter2.current) //显示计数器当前值
        myCounter3.info  //显示计数器信息
        myCounter3.increment(3)     //设置步长
        printf("Current Value is: %d\n",myCounter3.current) //显示计数器当前值

    }
}
```

编译执行上述代码后，得到右边结果：

```
Name: and mode is 1
Current Value is: 1
Name:Runner and mode is 1
Current Value is: 2
Name:Timer and mode is 2
Current Value is: 3
```

2.3.1.7主构造器

•Scala的每个类都有主构造器。但是，Scala的主构造器和Java有着明显的不同，Scala的主构造器是整个类体，需要在类名称后面罗列出构造器所需的所有参数，这些参数被编译成字段，字段的值就是创建对象时传入的参数的值。
•对于上面给计数器设置name和mode的例子，刚才我们是使用辅助构造器来对name和mode的值进行设置，现在我们重新来一次，这次我们转而采用主构造器来设置name和mode的值。

```
class Counter(val name: String, val mode: Int) {
    private var value = 0 //value用来存储计数器的起始值
    def increment(step: Int): Unit = { value += step}
    def current(): Int = {value}
    def info(): Unit = {printf("Name:%s and mode is %d\n",name,mode)}
}
object MyCounter{
    def main(args:Array[String]){
        val myCounter = new Counter("Timer",2)
        myCounter.info  //显示计数器信息
        myCounter.increment(1)  //设置步长
        printf("Current Value is: %d\n",myCounter.current) //显示计数器当前值
    }
}
```

编译执行上述代码后，得到结果：

```
Name:Timer and mode is 2
Current Value is: 1
```

2.3.2 对象

2.3.2.1 单例对象
2.3.2.2 伴生对象
2.3.2.3 应用程序对象
2.3.2.4 apply方法和update方法

2.3.2.1 单例对象

Scala并没有提供Java那样的静态方法或静态字段，但是，可以采用object关键字实现单例对象，具备和Java静态方法同样的功能。

下面是单例对象的定义：

```
object Person {
    private var lastId = 0  //一个人的身份编号
    def newPersonId() = {
        lastId +=1
        lastId
    }
}
```

可以看出，单例对象的定义和类的定义很相似，明显的区分是，用object关键字，而不是用class关键字。

2.3.2.1 单例对象

把上述代码放入到一个test.scala文件中测试

```
object Person {
    private var lastId = 0  //一个人的身份编号
    def newPersonId() = {
        lastId +=1
        lastId
    }
}
printf("The first person id is %d.\n",Person.newPersonId())
printf("The second person id is %d.\n",Person.newPersonId())
printf("The third person id is %d.\n",Person.newPersonId())
```

在Shell命令提示符下输入scala命令运行上面代码：

```
$ scala test.scala //不能使用scalac编译，直接运行即可
```

执行后，屏幕上会显示以下结果：

```
The first person id is 1.
The second person id is 2.
The third person id is 3.
```

2.3.2.2 伴生对象

- 在Java中，我们经常需要用到同时包含实例方法和静态方法的类，在Scala中可以通过伴生对象来实现。
- 当单例对象与某个类具有相同的名称时，它被称为这个类的“伴生对象”。
- 类和它的伴生对象必须存在于同一个文件中，而且可以相互访问私有成员（字段和方法）。

2.3.2.2 伴生对象

删除并重新创建一个test.scala，在该文件中输入如下代码：

```
class Person {
    private val id = Person.newPersonId() //调用了伴生对象中的方法
    private var name = ""
    def this(name: String) {
        this()
        this.name = name
    }
    def info() { printf("The id of %s is %d.\n",name,id)}
}
object Person {
    private var lastId = 0  //一个人的身份编号
    private def newPersonId() = {
        lastId +=1
        lastId
    }
    def main(args: Array[String]){
        val person1 = new Person("Ziyu")
        val person2 = new Person("Minxing")
        person1.info()
        person2.info()
    }
}
```

运行结果：

```
The id of Ziyu is 1.
The id of Minxing is 2.
```

2.3.2.2 伴生对象

•从上面结果可以看出，伴生对象中定义的newPersonId()实际上就实现了Java中静态（static）方法的功能

•Scala源代码编译后都会变成JVM字节码，实际上，在编译上面的源代码文件以后，在Scala里面的class和object在Java层面都会被合二为一，class里面的成员成了实例成员，object成员成了static成员

2.3.2.2 伴生对象

为了验证这一点，我们可以一起测试一下。

删除并重新创建一个test.scala，在该文件中输入如下代码：

```
class Person {
    private val id = Person.newPersonId() //调用了伴生对象中的方法
    private var name = ""
    def this(name: String) {
        this()
        this.name = name
    }
    def info() { printf("The id of %s is %d.\n",name,id)}
}
object Person {
    private var lastId = 0  //一个人的身份编号
    def newPersonId() = { //注意，这里已经没有private
        lastId +=1
        lastId
    }
    def main(args: Array[String]){
        val person1 = new Person("Ziyu")
        val person2 = new Person("Minxing")
        person1.info()
        person2.info()
    }
}
```

这里做一点小小修改，那就是把object Person中的newPersonId()方法前面的private去掉

2.3.2.2 伴生对象

在Shell命令提示符状态下，输入以下命令编译并执行：

```
$ scalac test.scala
$ ls //显示当前目录下生成的编译结果文件
```

在目录下看到两个编译后得到的文件，即Person.class和Person$.class。经过编译后，伴生类和伴生对象在JVM中都被合并到了一起

忽略Person$.class，只看Person.class。请使用下面命令进行“反编译”：

```
$ javap Person
```

执行结果如下：

```
Compiled from "test.scala"
public class Person {
  public static void main(java.lang.String[]);
  public static int newPersonId();
  public void info();
  public Person();
  public Person(java.lang.String);
}
```

从结果可以看出，经过编译后，伴生类Person中的成员和伴生对象Person中的成员都被合并到一起，并且，伴生对象中的方法newPersonId()，成为静态方法

2.3.2.3 应用程序对象

每个Scala应用程序都必须从一个对象的main方法开始

重新创建一个test.scala，在该文件中输入如下代码：

```
object HelloWorld {
    def main(args: Array[String]){
        println("Hello, World!")
    }
}
```

为了运行上述代码，我们现在可以使用两种不同的方法。

第一种方法:直接使用scala命令运行得到结果。

```
$ scala test.scala
```

第二种方法：先编译再执行

```
$ scalac test.scala //编译的时候使用的是Scala文件名称
$ scala -classpath . HelloWorld  //执行的时候使用的是HelloWorld对象名称
```

2.3.2.4 apply方法和update方法

我们经常会用到对象的apply方法和update方法，虽然我们表面上并没有察觉，但是，实际上，在Scala中，apply方法和update方法都会遵循相关的约定被调用，约定如下：

- 用括号传递给变量(对象)一个或多个参数时，Scala 会把它转换成对apply方法的调用
- 当对带有括号并包括一到若干参数的对象进行赋值时，编译器将调用对象的update方法，在调用时，是把括号里的参数和等号右边的对象一起作为update方法的输入参数来执行调用

2.3.2.4 apply方法和update方法

下面我们测试一下apply方法是否被调用。删除并重新创建test.scala文件，输入以下代码：

```
class TestApplyClass {

    def apply(param: String): String = {

        println("apply method called, parameter is: " + param)

        "Hello World!"
    }
}
val myObject = new TestApplyClass
println(myObject("param1"))
```

在Linux系统的Shell命令提示符下运行scala命令：

```
$ scala test.scala
```

运行后会得到以下结果：

```
apply method is called, parameter is:param1
Hello World!
```

2.3.2.4 apply方法和update方法

上面是类中定义了apply方法，下面看一个在单例对象中定义apply方法的例子：

```
object TestApplySingleObject {
    def apply(param1: String, param2: String): String = {
        println("apply method called")
        param1 + " and " + param2
    }
}

val group = TestApplySingleObject("Zhangfei", "Liubei")
println(group)
```

把上面代码放入到test.scala文件中测试执行后，可以得到如下结果：

```
apply method called
Zhangfei and Liubei
```

可以看出，在执行TestApplySingleObject("Zhangfei", "Liubei")时调用了apply方法，并且把“Zhangfei and Liubei”作为返回值，赋值给group变量，因此，println(group)语句会打印出“Zhangfei and Liubei”。

2.3.2.4 apply方法和update方法

下面我们测试一个伴生类和伴生对象中的apply方法实例。删除并重新创建test.scala文件，输入以下代码：

```
class TestApplyClassAndObject {
}
class ApplyTest{
    def apply() = println("apply method in class is called!")
    def greetingOfClass: Unit ={
        println("Greeting method in class is called.")
    }
}
object ApplyTest{
     def apply() = {
          println("apply method in object is called")
        new ApplyTest()
     }
}
object  TestApplyClassAndObject{
     def main (args: Array[String]) {
        val a = ApplyTest() //这里会调用伴生对象中的apply方法
        a.greetingOfClass
        a() // 这里会调用伴生类中的apply方法
    }
}
```

执行结果如下：

```
apply method in object is called
Greeting method in class is called.
apply method in class is called!
```

2.3.2.4 apply方法和update方法

首先使用scalac编译命令对test.scala进行编译，然后，使用scala命令运行，具体如下：

```
$ cd /usr/local/scala/mycode
$ scalac test.scala
$ scala  -classpath . TestApplyClassAndObject //这里的TestApplyClassAndObject是在test.scala文件中object关键字后面的TestApplyClassAndObject
```

上述代码执行后得到以下结果：

```
apply method in object is called
Greeting method in class is called.
apply method in class is called!
```

从上面代码可以看出，当我们执行val a = ApplyTest()时，会导致apply方法的调用并返回该方法调用的值，也就是ApplyTest的实例化对象。当执行a()时，又会导致调用伴生类的apply方法，如果我们愿意，就可以在伴生类的apply方法中写入一些处理逻辑，这样就可以把传入的参数赋值给实例化对象的变量。

2.3.2.4 apply方法和update方法

下面看一个apply方法的例子。由于Scala中的Array对象定义了apply方法，因此，我们就可以采用如下方式初始化一个数组：

```
val myStrArr = Array("BigData","Hadoop","Spark")
```

也就是说，不需要new关键字，不用构造器，直接给对象传递3个参数，Scala就会转换成对apply方法的调用，也就是调用Array类的伴生对象Array的apply方法，完成数组的初始化。

2.3.2.4 apply方法和update方法

实际上，update方法也是类似的，比如：

```
val myStrArr = new Array[String](3) //声明一个长度为3的字符串数组，每个数组元素初始化为null
myStrArr(0) = "BigData" //实际上，调用了伴生类Array中的update方法，执行myStrArr.update(0,"BigDat
a")
myStrArr(1) = "Hadoop" //实际上，调用了伴生类Array中的update方法，执行myStrArr.update(1,"Hadoo
p")
myStrArr(2) = "Spark" //实际上，调用了伴生类Array中的update方法，执行myStrArr.update(2,"Spark")
```

从上面可以看出，在进行元组赋值的时候，之所以没有采用Java中的方括号myStrArr[0]，而是采用圆括号的形式，myStrArr(0)，是因为存在上述的update方法的机制。

2.3.3 继承

2.3.3.1 Scala与Java在继承方面的区别
2.3.3.2 抽象类
2.3.3.3 扩展类

2.3.3.1 Scala与Java在继承方面的区别

Scala中的继承与Java有着显著的不同：
（1）重写一个非抽象方法必须使用override修饰符。
（2）只有主构造器可以调用超类的主构造器。
（3）在子类中重写超类的抽象方法时，不需要使用override关键字。
（4）可以重写超类中的字段。

Scala和Java一样，不允许类从多个超类继承

2.3.3.2 抽象类

以汽车为例子，首先我们创建一个抽象类，让这个抽象类被其他类继承。

```
abstract class Car{    //是抽象类，不能直接被实例化
   val carBrand: String //字段没有初始化值，就是一个抽象字段
     def info() //抽象方法，不需要使用abstract关键字
     def greeting() {println("Welcome to my car!")}
}
```

关于上面的定义，说明几点：
（1）定义一个抽象类，需要使用关键字abstract。
（2）定义一个抽象类的抽象方法，也不需要关键字abstract，只要把方法体空着，不写方法体就可以。
（3）抽象类中定义的字段，只要没有给出初始化值，就表示是一个抽象字段，但是，抽象字段必须要声明类型，比如：val carBrand: String，就把carBrand声明为字符串类型，这个时候，不能省略类型，否则编译会报错。

2.3.3.3 扩展类

抽象类不能直接被实例化，所以，下面我们定义几个扩展类，它们都是扩展了Car类，或者说继承自Car类。

```
class BMWCar extends Car {
    override val carBrand = "BMW"  //重写超类字段，需要使用override关键字，否则编译会报错
    def info() {printf("This is a %s car. It is on sale", carBrand)} //重写超类的抽象方法时，不需要使用override关键字，不过，如果加上override编译也不错报错
    override def greeting() {println("Welcome to my BMW car!")} //重写超类的非抽象方法，必须使用override关键字
}

class BYDCar extends Car {
    override val carBrand = "BYD" //重写超类字段，需要使用override关键字，否则编译会报错
    def info() {printf("This is a %s car. It is cheap.", carBrand)} //重写超类的抽象方法时，不需要使用override关键字，不过，如果加上override编译也不错报错
    override def greeting() {println("Welcome to my BYD car!")} //重写超类的非抽象方法，必须使用override关键字
}
```

2.3.3.3 扩展类

下面，我们把上述代码放入一个完整的代码文件test.scala，编译运行。

```
abstract class Car{
   val carBrand: String
     def info()
     def greeting() {println("Welcome to my car!")}
}
class BMWCar extends Car {
    override val carBrand = "BMW"
    def info() {printf("This is a %s car. It is expensive.\n", carBrand)}
    override def greeting() {println("Welcome to my BMW car!")}
}

class BYDCar extends Car {
    override val carBrand = "BYD"
    def info() {printf("This is a %s car. It is cheap.\n", carBrand)}
    override def greeting() {println("Welcome to my BYD car!")}
}

object MyCar {
    def main(args: Array[String]){
        val myCar1 = new BMWCar()
        val myCar2 = new BYDCar()
        myCar1.greeting()
        myCar1.info()
        myCar2.greeting()
        myCar2.info()
    }
}
```

执行后，屏幕上会显示以下结果：

```
Welcome to my BMW car!
This is a BMW caar. It is expensive.
Welcome to my BYD car!
This is a BYD caar. It is cheap.
```

2.3.3.3 扩展类

在Shell命令提示符下输入scala命令运行上面代码：

```
$ scalac test.scala //编译
$ scala -classpath . MyCar //运行，这里的MyCar是上面代码文件中object后面的MyCar
```

执行后，屏幕上会显示以下结果：

```
Welcome to my BMW car!
This is a BMW caar. It is expensive.
Welcome to my BYD car!
This is a BYD caar. It is cheap.
```

2.3.4 特质（trait）

2.3.4.1 特质概述
2.3.4.2 特质的定义
2.3.4.3 把特质混入类中
2.3.4.4 特质可以包含具体实现
2.3.4.5 把多个特质混入类中

2.3.4.1 特质概述

- Java中提供了接口，允许一个类实现任意数量的接口
- 在Scala中没有接口的概念，而是提供了“特质(trait)”，它不仅实现了接口的功能，还具备了很多其他的特性
- Scala的特质，是代码重用的基本单元，可以同时拥有抽象方法和具体方法
- Scala中，一个类只能继承自一个超类，却可以实现多个特质，从而重用特质中的方法和字段，实现了多重继承

2.3.4.2 特质的定义

特质的定义和类的定义非常相似，有区别的是，特质定义使用关键字trait。

```
trait CarId{
  var id: Int
    def currentId(): Int      //定义了一个抽象方法
}
```

上面定义了一个特质，里面包含一个抽象字段id和抽象方法currentId。注意，抽象方法不需要使用abstract关键字，特质中没有方法体的方法，默认就是抽象方法。

2.3.4.3 把特质混入类中

特质定义好以后，就可以使用extends或with关键字把特质混入类中。

```
class BYDCarId extends CarId{ //使用extends关键字
  override var id = 10000 //BYD汽车编号从10000开始
    def currentId(): Int = {id += 1; id} //返回汽车编号
}
class BMWCarId extends CarId{ //使用extends关键字
  override var id = 20000 //BMW汽车编号从20000开始
    def currentId(): Int = {id += 1; id} //返回汽车编号
}
```

2.3.4.3 把特质混入类中

下面，我们把上述代码放入一个完整的代码文件test.scala，编译运行。

```
trait CarId{
  var id: Int
    def currentId(): Int     //定义了一个抽象方法
}
class BYDCarId extends CarId{ //使用extends关键字
   override var id = 10000 //BYD汽车编号从10000开始
     def currentId(): Int = {id += 1; id} //返回汽车编号
 }
 class BMWCarId extends CarId{ //使用extends关键字
   override var id = 20000 //BMW汽车编号从10000开始
     def currentId(): Int = {id += 1; id} //返回汽车编号
 }
 object MyCar {
    def main(args: Array[String]){
        val myCarId1 = new BYDCarId()
        val myCarId2 = new BMWCarId()
        printf("My first CarId is %d.\n",myCarId1.currentId)
        printf("My second CarId is %d.\n",myCarId2.currentId)
    }
}
```

执行结果：

```
My first CarId is 10001.
My second CarId is 20001.
```

2.3.4.4 特质可以包含具体实现

上面的实例中，特质只包含了抽象字段和抽象方法，相当于实现了类似Java接口的功能。实际上，特质也可以包含具体实现，也就是说，特质中的字段和方法不一定要是抽象的。

```
trait CarGreeting{
  def greeting(msg: String) {println(msg)}
}
```

2.3.4.5 把多个特质混入类中

上面已经定义了两个特质CarId和CarGreeting。可以把两个特质都混入到类中。

```
trait CarId{
  var id: Int
    def currentId(): Int     //定义了一个抽象方法
}
trait CarGreeting{
  def greeting(msg: String) {println(msg)}
}

class BYDCarId extends CarId with CarGreeting{ //使用extends关键字混入第1个特质，后面可以反复使用with关键字混入更多特质
   override var id = 10000 //BYD汽车编号从10000开始
     def currentId(): Int = {id += 1; id} //返回汽车编号
 }
 class BMWCarId extends CarId with CarGreeting{ //使用extends关键字混入第1个特质，后面可以反复使用with关键字混入更多特质
   override var id = 20000 //BMW汽车编号从10000开始
     def currentId(): Int = {id += 1; id} //返回汽车编号
 }
 object MyCar {
    def main(args: Array[String]){
        val myCarId1 = new BYDCarId()
        val myCarId2 = new BMWCarId()
        myCarId1.greeting("Welcome my first car.")
        printf("My first CarId is %d.\n",myCarId1.currentId)
        myCarId2.greeting("Welcome my second car.")
        printf("My second CarId is %d.\n",myCarId2.currentId)
    }
}
```

执行结果如下：

```
Welcome my first car.
My first CarId is 10001.
Welcome my second car.
My second CarId is 20001.
```

2.3.5 模式匹配

2.3.5.1 简单匹配
2.3.5.2 类型模式
2.3.5.3 "守卫(guard)"语句
2.3.5.4 for表达式中的模式
2.3.5.5 case类的匹配
2.3.5.6 Option类型

2.3.5.1 简单匹配

Scala的模式匹配最常用于match语句中。下面是一个简单的整型值的匹配实例。

```
val colorNum = 1
val colorStr = colorNum match {
    case 1 => "red"
    case 2 => "green"
    case 3 => "yellow"
    case _ => "Not Allowed"
}
println(colorStr)
```

另外，在模式匹配的case语句中，还可以使用变量。

```
val colorNum = 4
val colorStr = colorNum match {
    case 1 => "red"
    case 2 => "green"
    case 3 => "yellow"
    case unexpected => unexpected + " is Not Allowed"
}
println(colorStr)
```

执行结果：`4 is Not Allowed`

2.3.5.2 类型模式

Scala可以对表达式的类型进行匹配。

```
for (elem <- List(9,12.3,"Spark","Hadoop",'Hello)){
    val str  = elem match{
        case i: Int => i + " is an int value."
        case d: Double => d + " is a double value."
        case "Spark"=> "Spark is found."
        case s: String => s + " is a string value."
        case _ => "This is an unexpected value."
    }
println(str)
}
```

执行结果：

```
9 is an int value.
12.3 is a double value.
Spark is found.
Hadoop is a string value.
This is an unexpected value.
```

2.3.5.3 "守卫(guard)"语句

可以在模式匹配中添加一些必要的处理逻辑。

```
for (elem <- List(1,2,3,4)){
      elem match {
          case _ if (elem %2 == 0) => println(elem + " is even.")
            case _ => println(elem + " is odd.")
    }
}
```

执行结果：

```
1 is odd.
2 is even.
3 is odd.
4 is even.
```

2.3.5.4 for表达式中的模式

以我们之前举过的映射为例子，我们创建的映射如下：

```
val university = Map("XMU" -> "Xiamen University", "THU" -> "Tsinghua University","PKU"->"Peking University")
```

循环遍历映射的基本格式是：

```
for ((k,v) <- 映射) 语句块
```

对于遍历过程得到的每个值，都会被绑定到k和v两个变量上

```
for ((k,v) <- university) printf("Code is : %s and name is: %s\n",k,v)
```

执行结果：

```
Code is : XMU and name is: Xiamen University
Code is : THU and name is: Tsinghua University
Code is : PKU and name is: Peking University
```

2.3.5.5 case类的匹配

case类是一种特殊的类，它们经过优化以被用于模式匹配。

```
case class Car(brand: String, price: Int)
val myBYDCar = new Car("BYD", 89000)
val myBMWCar = new Car("BMW", 1200000)
val myBenzCar = new Car("Benz", 1500000)
for (car <- List(myBYDCar, myBMWCar, myBenzCar)) {
    car match{
        case Car("BYD", 89000) => println("Hello, BYD!")
        case Car("BMW", 1200000) => println("Hello, BMW!")
        case Car(brand, price) => println("Brand:"+ brand +", Price:"+price+", do you want i
t?")
    }
}
```

执行结果：

```
Hello, BYD!
Hello, BMW!
Brand: Benz, Price:1500000, do you want it?
```

2.3.5.6 Option类型

•标准类库中的Option类型用case类来表示那种可能存在、也可能不存在的值。

•一般而言，对于每种语言来说，都会有一个关键字来表示一个对象引用的是“无”，在Java中使用的是null。Scala融合了函数式编程风格，因此，当预计到变量或者函数返回值可能不会引用任何值的时候，建议你使用Option类型。

•Option类包含一个子类Some，当存在可以被引用的值的时候，就可以使用Some来包含这个值，例如Some("Hadoop")。而None则被声明为一个对象，而不是一个类，表示没有值。

2.3.5.6 Option类型

下面我们给出一个实例。

```
//首先我们创建一个映射
scala> val books=Map("hadoop"->5,"spark"->10,"hbase"->7)
books: scala.collection.immutable.Map[String,Int] = Map(hadoop -> 5, spark -> 10, hbase
-> 7)

//下面我们从映射中取出键为"hadoop"对应的值，这个键是存在的，可以取到值，并且取到的值会被包含在S
ome中返回
scala> books.get("hadoop")
res0: Option[Int] = Some(5)

//下面我们从映射中取出键为"hive"对应的值，这个键是不存在的，所以取到的值是None对象
scala> books.get("hive")
res1: Option[Int] = None
```

2.3.5.6 Option类型

Option类型还提供了getOrElse方法，这个方法在这个Option是Some的实例时返回对应的值，而在是None的实例时返回传入的参数。例如：

```
scala> val sales=books.get("hive")
sales: Option[Int] = None

scala> sales.getOrElse("No Such Book")
res3: Any = No Such Book

scala> println(sales.getOrElse("No Such Book"))
No Such Book
```

可以看出，当我们采用getOrElse方法时，如果我们取的"hive"没有对应的值，我们就可以显示我们指定的“No Such Book”，而不是显示None。

2.3.5.6 Option类型

在Scala中，使用Option的情形是非常频繁的。在Scala里，经常会用到Option[T]类型，其中的T可以是Sting或Int或其他各种数据类型。
Option[T]实际上就是一个容器，我们可以把它看做是一个集合，只不过这个集合中要么只包含一个元素（被包装在Some中返回），要么就不存在元素（返回None）。
既然是一个集合，我们当然可以对它使用map、foreach或者filter等方法。比如：

```
scala> books.get("hive").foreach(println)
```

可以发现，上述代码执行后，屏幕上什么都没有显示，因为，foreach遍历遇到None的时候，什么也不做，自然不会执行println操作。

2.4 函数式编程基础

2.4.1 函数定义和高阶函数
2.4.2 针对集合的操作
2.4.3 函数式编程实例WordCount

2.4.1 函数定义和高阶函数

2.4.1.1 函数字面量
2.4.1.2 函数的类型和值
2.4.1.3 匿名函数、Lamda表达式与闭包
2.4.1.4 占位符语法

2.4.1.1 函数字面量

字面量包括整数字面量、浮点数字面量、布尔型字面量、字符字面量、字符串字面量、符号字面量、函数字面量和元组字面量。

```
val i = 123  //123就是整数字面量
val i = 3.14 //3.14就是浮点数字面量
val i = true //true就是布尔型字面量
val i = 'A' //'A'就是字符字面量
val i = "Hello" //"Hello"就是字符串字面量
```

除了函数字面量我们会比较陌生以外，其他几种字面量都很容易理解。

2.4.1.1 函数字面量

•函数字面量可以体现函数式编程的核心理念。
•在非函数式编程语言里，函数的定义包含了“函数类型”和“值”两种层面的内容。
•但是，在函数式编程中，函数是“头等公民”，可以像任何其他数据类型一样被传递和操作，也就是说，函数的使用方式和其他数据类型的使用方式完全一致了。
•这时，我们就可以像定义变量那样去定义一个函数，由此导致的结果是，函数也会和其他变量一样，开始有“值”。
•就像变量的“类型”和“值”是分开的两个概念一样，函数式编程中，函数的“类型”和“值”也成为两个分开的概念，函数的“值”，就是“函数字面量”。

2.4.1.2 函数的类型和值

下面我们一点点引导大家更好地理解函数的“类型”和“值”的概念。
我们现在定义一个大家比较熟悉的传统类型的函数，定义的语法和我们之前介绍过的定义“类中的方法”类似（实际上，定义函数最常用的方法是作为某个对象的成员，这种函数被称为方法）：

```
def counter(value: Int): Int = { value += 1}
```

上面定义个这个函数的“类型”如下：

```
(Int) => Int
```

实际上，只有多个参数时（不同参数之间用逗号隔开），圆括号才是必须的，当参数只有一个时，圆括号可以省略，如下：

```
Int => Int
```

上面就得到了函数的“类型”

2.4.1.2 函数的类型和值

下面看看如何得到函数的“值”

实际上，我们只要把函数定义中的类型声明部分去除，剩下的就是函数的“值”，如下：

```
(value) => {value += 1} //只有一条语句时，大括号可以省略
```

注意：上面就是函数的“值”，需要注意的是，采用“=>”而不是“=”，这是Scala的语法要求。

2.4.1.2 函数的类型和值

现在，我们再按照大家比较熟悉的定义变量的方式，采用Scala语法来定义一个函数。
声明一个变量时，我们采用的形式是：

```
val num: Int = 5 //当然，Int类型声明也可以省略，因为Scala具有自动推断类型的功能
```

照葫芦画瓢，我们也可以按照上面类似的形式来定义Scala中的函数：

```
val counter: Int => Int = { (value) => value += 1 }
```

从上面可以看出，在Scala中，函数已经是“头等公民”，单独剥离出来了“值”的概念，一个函数“值”就是函数字面量。这样，我们只要在某个需要声明函数的地方声明一个函数类型，在调用的时候传一个对应的函数字面量即可，和使用普通变量一模一样。

2.4.1.3 匿名函数、Lamda表达式与闭包

我们不需要给每个函数命名，这时就可以使用匿名函数，如下：

```
(num: Int) => num * 2
```

上面这种匿名函数的定义形式，我们经常称为“Lamda表达式”。“Lamda表达式”的形式如下：

```
(参数) => 表达式 //如果参数只有一个，参数的圆括号可以省略
```

我们可以直接把匿名函数存放到变量中，下面是在Scala解释器中的执行过程：

```
scala> val myNumFunc: Int=>Int = (num: Int) => num * 2 //这行是我们输入的命令，把匿名函数定义为一个值，赋值给myNumFunc变量
myNumFunc: Int => Int = <function1>  //这行是执行返回的结果
scala> println(myNumFunc(3)) //myNumFunc函数调用的时候，需要给出参数的值，这里传入3，得到乘法结果是6
6
```

2.4.1.3 匿名函数、Lamda表达式与闭包

实际上，Scala具有类型推断机制，可以自动推断变量类型，比如下面两条语句都是可以的：

```
val number: Int = 5
val number =5 //省略Int类型声明
```

所以，上面的定义中，我们可以myNumFunc的类型声明，也就是去掉“Int=>Int”，在Scala解释器中的执行过程如下：

```
scala> val myNumFunc = (num: Int) => num * 2
myNumFunc: Int => Int = <function1>
scala> println(myNumFunc(3))
6
```

2.4.1.3 匿名函数、Lamda表达式与闭包

下面我们再尝试一下，是否可以继续省略num的类型声明，在Scala解释器中的执行过程如下：

```
scala> val myNumFunc= (num) => num * 2
<console>:12: error: missing parameter type
       val myNumFunc= (num) => num * 2
                      ^
```

可以看出，解释器会报错，因为，全部省略以后，实际上，解释器也无法推断出类型

下面我们尝试一下，省略num的类型声明，但是，给出myNumFunc的类型声明，在Scala解释器中的执行过程如下：

```
scala> val myNumFunc: Int=>Int = (num) => num * 2
myNumFunc: Int => Int = <function1>
```

不会报错，因为，给出了myNumFunc的类型为“Int=>Int”以后，解释器可以推断出num类型为Int类型。

2.4.1.3 匿名函数、Lamda表达式与闭包

闭包是一个函数，一种比较特殊的函数，它和普通的函数有很大区别

普通函数：`val addMore=(x:Int)=>x>0`

闭包：`val addMore=(x:Int)=>x+more`

闭包反映了一个从开放到封闭的过程

```
scala> var more = 1
more: Int = 1
scala> val addMore = (x: Int) => x + more
addMore: (Int) => Int = < function>
scala> addMore(10)
res19: Int = 11
```

- 每次addMore函数被调用时都会创建一个新闭包
- 每个闭包都会访问闭包创建时活跃的more变量

```
scala> var more = 1
more: Int = 1
scala> val addMore = (x: Int) => x + more
addMore: (Int) => Int = < function>
scala> addMore(10)
res19: Int = 11
scala> more =9
more: Int = 9
scala> addMore(10)
res20: Int = 19
```

2.4.1.4 占位符语法

为了让函数字面量更加简洁，我们可以使用下划线作为一个或多个参数的占位符，只要每个参数在函数字面量内仅出现一次。

```
scala> val numList = List(-3, -5, 1, 6, 9)
numList: List[Int] = List(-3, -5, 1, 6, 9)
scala> numList.filter(x => x > 0 )
res1: List[Int] = List(1, 6, 9)
scala> numList.filter(_ > 0)
res2: List[Int] = List(1, 6, 9)
```

从上面运行结果可以看出，下面两个函数字面量是等价的。

```
x => x>0
```

```
_ > 0
```

2.4.1.4 占位符语法

有时你把下划线当作参数的占位符时，编译器有可能没有足够的信息推断缺失的参数类型。例如，假设你只是写_ + _：

```
scala> val f = _ + _
<console>:4: error: missing parameter type for expanded
function ((x$1, x$2) => x$1.$plus(x$2))
  val f = _ + _
```

这种情况下，你可以运用 冒号指定类型，如下：

```
scala> val f = (_: Int) + (_: Int)
f: (Int, Int) => Int = <function>
scala> f(5, 10)
res11: Int = 15
```

请留心 _ + _将扩展成带两个参数的函数字面量。这也是仅当每个参数在函数字面量中最多出现一次的情况下你才能运用 这种短格式的原由 。多个下划线指代多个参数，而不是单个参数的重复运用 。第一个下划线代表第一个参数，第二个下划线代表第二个，第三个……，如此类推。

2.4.2 针对集合的操作

2.4.2.1 遍历操作
2.4.2.2 map操作和flatMap操作
2.4.2.3 filter操作
2.4.2.4 reduce操作
2.4.2.5 fold操作

2.4.2.1 遍历操作

列表的遍历

可以使用for循环进行遍历：

```
val list = List(1, 2, 3, 4, 5)
for (elem <- list) println(elem)
```

也可以使用foreach进行遍历：

```
val list = List(1, 2, 3, 4, 5)
list.foreach(elem => println(elem)) //本行语句甚至可以简写为list.foreach(println)，或者写成：lis
t foreach println
```

2.4.2.1 遍历操作

映射的遍历

循环遍历映射，是经常需要用到的操作，基本格式是：

```
for ((k,v) <- 映射) 语句块
```

创建一个映射

```
val university = Map("XMU" -> "Xiamen University", "THU" -> "Tsinghua University","PKU"->"Peking University")
```

循环遍历映射

```
for ((k,v) <- university) printf("Code is : %s and name is: %s\n",k,v)
```

执行结果：

```
Code is : XMU and name is: Xiamen University
Code is : THU and name is: Tsinghua University
Code is : PKU and name is: Peking University
```

2.4.2.1 遍历操作

映射的遍历

也可以使用foreach来实现对映射的遍历

```
scala> university foreach {case(k,v) => println(k+":"+v)} //或者写成：university.foreac
h({case (k,v) => println(k+":"+v)})
XMU:Xiamen University
THU:Tsinghua University
PKU:Peking University
```

也可以尝试使用下面形式来遍历

```
scala> university foreach {kv => println(kv._1+":"+kv._2)}
XMU:Xiamen University
THU:Tsinghua University
PKU:Peking University
```

2.4.2.2 map操作和flatMap操作

map操作

map操作是针对集合的典型变换操作，它将某个函数应用到集合中的每个元素，并产生一个结果集合。

```
scala> val books = List("Hadoop", "Hive", "HDFS")
books: List[String] = List(Hadoop, Hive, HDFS)
scala> books.map(s => s.toUpperCase)
res0: List[String] = List(HADOOP, HIVE, HDFS)
```

2.4.2.2 map操作和flatMap操作

flatMap操作

flatMap是map的一种扩展。在flatMap中，我们会传入一个函数，该函数对每个输入都会返回一个集合（而不是一个元素），然后，flatMap把生成的多个集合“拍扁”成为一个集合。

```
scala> val books = List("Hadoop","Hive","HDFS")
books: List[String] = List(Hadoop, Hive, HDFS)
scala> books flatMap (s => s.toList)
res0: List[Char] = List(H, a, o, o, p, H, i, v, e, H, D, F, S)
```

上面的flatMap执行时，会把books中的每个元素都调用toList，生成List[Char]，最终，多个Char的集合被“拍扁”成一个集合。

2.4.2.3 filter操作

遍历一个集合并从中获取满足指定条件的元素组成一个新的集合。Scala中可以通过filter操作来实现。

首先创建一个映射：

```
val university = Map("XMU" -> "Xiamen University", "THU" -> "Tsinghua University","PKU"->"Peking University","XMUT"->"Xiamen University of Technology")
```

采用filter操作过滤得到那些学校名称中包含“Xiamen”的元素

```
val universityOfXiamen = university filter {kv => kv._2 contains "Xiamen"}
```

采用filter操作过滤得到那些学校名称中以字母“P”开头的元素：

```
val universityOfP = university filter {kv => kv._2 startsWith "P"}
```

2.4.2.4 reduce操作

使用reduce这种二元操作对集合中的元素进行归约

reduce包含reduceLeft和reduceRight两种操作，前者从集合的头部开始操作，后者从集合的尾部开始操作。

```
scala> val list = List(1,2,3,4,5)
list: List[Int] = List(1, 2, 3, 4, 5)
scala> list.reduceLeft(_ + _)
res21: Int = 15
scala> list.reduceRight(_ + _)
res22: Int = 15
```

reduceLeft(_ + _)整个加法操作的执行顺序如下：

```
1+2 = 3
3+3 = 6
6+4 = 10
10+5 = 15
```

reduceRight(_ + _)表示从列表尾部开始，对两两元素进行求和操作，顺序如下：

```
4+5 = 9
3+9 = 12
2+12 = 14
1+14 = 15
```

直接使用reduce，而不用reduceLeft和reduceRight，这时，默认采用的是reduceLeft

2.4.2.5 fold操作

折叠(fold)操作和reduce（归约）操作比较类似。fold操作需要从一个初始的“种子”值开始，并以该值作为上下文，处理集合中的每个元素。

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

scala> list.fold(10)(_*_)
res0: Int = 1200
```

fold有两个变体：foldLeft()和foldRight()，其中，foldLeft()，第一个参数为累计值，集合遍历的方向是从左到右。foldRight()，第二个参数为累计值，集合遍历的方向是从右到左。对于fold()自身而言，遍历的顺序是未定义的，不过，一般都是从左到右遍历。

2.4.3 函数式编程实例WordCount

```
import java.io.File
import scala.io.Source
object WordCount {
  def main(args: Array[String]): Unit = {
    val dirfile=new File("/usr/local/scala/mycode/wordcount")
    val files=dirfile.listFiles
    for(file <- files) println(file)
    val listFiles=files.toList
    val wordsMap=scala.collection.mutable.Map[String,Int]()
    listFiles.foreach( file =>Source.fromFile(file).getLines().foreach(line=>line.split("
").
                  foreach(
                      word=>{
                        if (wordsMap.contains(word)) {
                          wordsMap(word)+=1
                        }else {
                          wordsMap+=(word->1)
                        }
                      }
                  )
            )

    )
    println(wordsMap)
    for((key,value)<-wordsMap) println(key+": "+value)
  }
}
```

附录：主讲教师林子雨简介

主讲教师：林子雨

单位：厦门大学计算机科学系
E-mail: ziyulin@xmu.edu.cn
个人网页：http://www.cs.xmu.edu.cn/linziyu
数据库实验室网站：http://dblab.xmu.edu.cn

扫一扫访问个人主页

林子雨，男，1978年出生，博士（毕业于北京大学），现为厦门大学计算机科学系助理教授（讲师），曾任厦门大学信息科学与技术学院院长助理、晋江市发展和改革局副局长。中国计算机学会数据库专业委员会委员，中国计算机学会信息系统专业委员会委员，荣获“2016中国大数据创新百人”称号。中国高校首个“数字教师”提出者和建设者，厦门大学数据库实验室负责人，厦门大学云计算与大数据研究中心主要建设者和骨干成员，2013年度厦门大学奖教金获得者。主要研究方向为数据库、数据仓库、数据挖掘、大数据、云计算和物联网，并以第一作者身份在《软件学报》《计算机学报》和《计算机研究与发展》等国家重点期刊以及国际学术会议上发表多篇学术论文。作为项目负责人主持的科研项目包括1项国家自然科学青年基金项目(No.61303004)、1项福建省自然科学青年基金项目(No.2013J05099)和1项中央高校基本科研业务费项目(No.2011121049)，同时，作为课题负责人完成了国家发改委城市信息化重大课题、国家物联网重大应用示范工程区域试点泉州市工作方案、2015泉州市互联网经济调研等课题。中国高校首个“数字教师”提出者和建设者，2009年至今，“数字教师”大平台累计向网络免费发布超过100万字高价值的研究和教学资料，累计网络访问量超过100万次。打造了中国高校大数据教学知名品牌，编著出版了中国高校第一本系统介绍大数据知识的专业教材《大数据技术原理与应用》，并成为京东、当当网等网店畅销书籍；建设了国内高校首个大数据课程公共服务平台，为教师教学和学生学习大数据课程提供全方位、一站式服务，年访问量超过50万次。具有丰富的政府和企业信息化培训经验，厦门大学管理学院EDP中心、浙江大学管理学院EDP中心、厦门大学继续教育学院、泉州市科技培训中心特邀培训讲师，曾给中国移动通信集团公司、福州马尾区政府、福建龙岩卷烟厂、福建省物联网科学研究院、石狮市物流协会、厦门市物流协会、浙江省中小企业家、四川泸州企业家、江苏沛县企业家等开展信息化培训，累计培训人数达3000人以上。

附录：林子雨编著《Spark入门教程》

厦门大学林子雨编著《Spark入门教程》
教程内容包括Scala语言、Spark简介、安装、运行架构、RDD的设计与运行原理、部署模式、RDD编程、键值对RDD、数据读写、Spark SQL、Spark Streaming、MLlib等

免费在线教程：http://dblab.xmu.edu.cn/blog/spark/

附录：《大数据技术原理与应用》教材

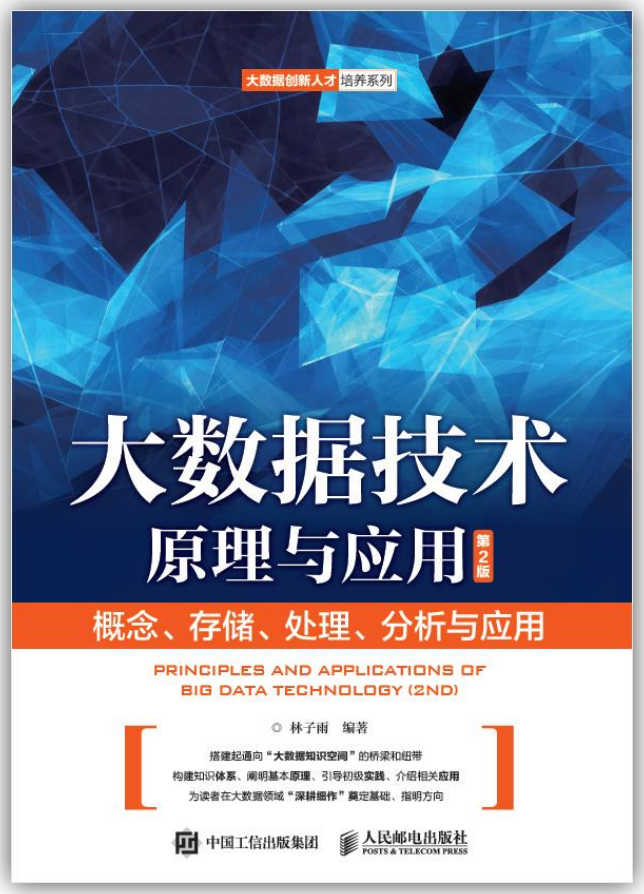
扫一扫访问教材官网

《大数据技术原理与应用——概念、存储、处理、分析与应用（第2版）》，由厦门大学计算机科学系林子雨老师编著，是中国高校第一本系统介绍大数据知识的专业教材。

全书共有15章，系统地论述了大数据的基本概念、大数据处理架构Hadoop、分布式文件系统HDFS、分布式数据 库HBase、NoSQL数据库、云数据库、分布式并行编程模型MapReduce、Spark、流计算、图计算、数据可视化以及大数据在互联网、生物医学和物流等各个领域的应用。在Hadoop、HDFS、HBase、MapReduce和Spark等重要章节，安排了入门级的实践操作，让读者更好地学习和掌握大数据关键技术。

本书可以作为高等院校计算机专业、信息管理等相关专业的大数据课程教材，也可供相关技术人员参考、学习、培训之用。

欢迎访问《大数据技术原理与应用——概念、存储、处理、分析与应用（第2版）》教材官方网站：
http://dblab.xmu.edu.cn/post/bigdata



附录：中国高校大数据课程公共服务平台

中国高校大数据课程

公 共 服 务 平 台

http://dblab.xmu.edu.cn/post/bigdata-teaching-platform/

扫一扫访问平台主页

扫一扫观看3分钟FLASH动画宣传片


Thank You!