《Spark编程基础》

教材官网： http://dblab.xmu.edu.cn/post/spark/

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：2018年春季学期）

扫一扫访问教材官网

林子雨

厦门大学计算机科学系

E-mail: ziyulin@xmu.edu.cn

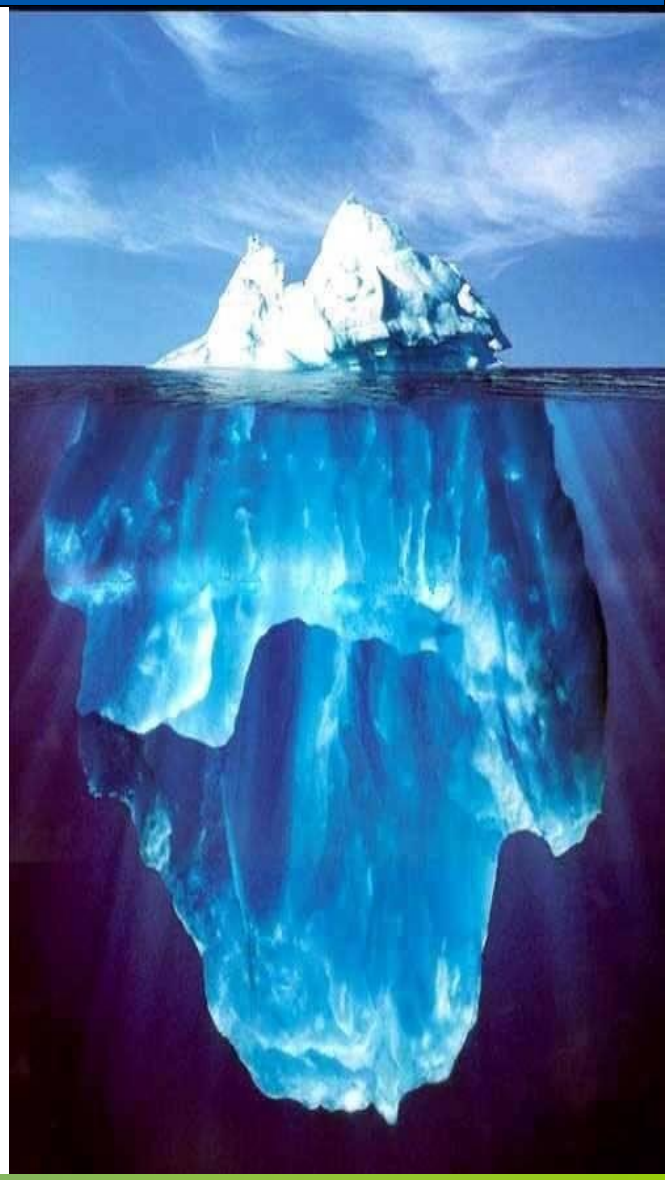
主页：**http://www.cs.xmu.edu.cn/linziyu**

课程配套授课视频

课程在线视频地址：http://dblab.xmu.edu.cn/post/10482/

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



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

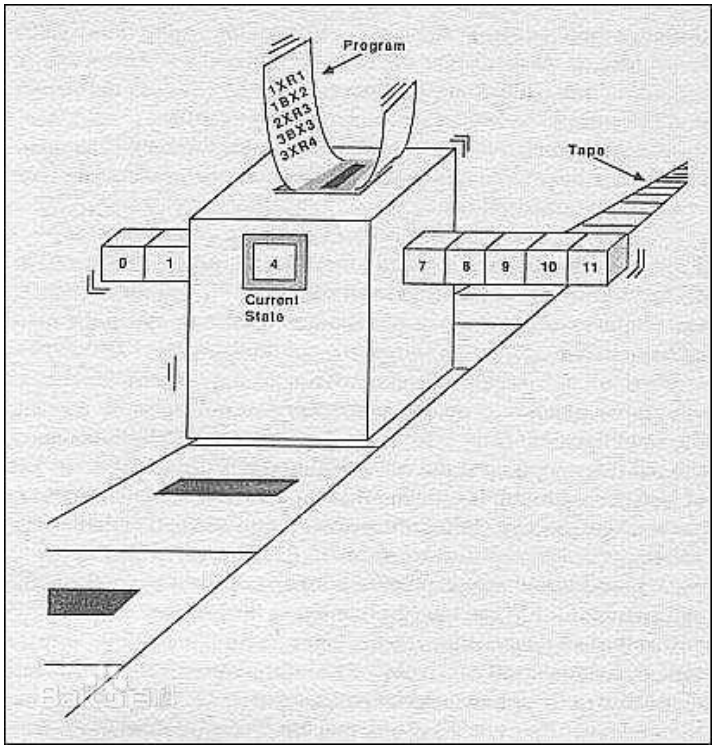
•λ演算被视为最小的通用程序设计语言

•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

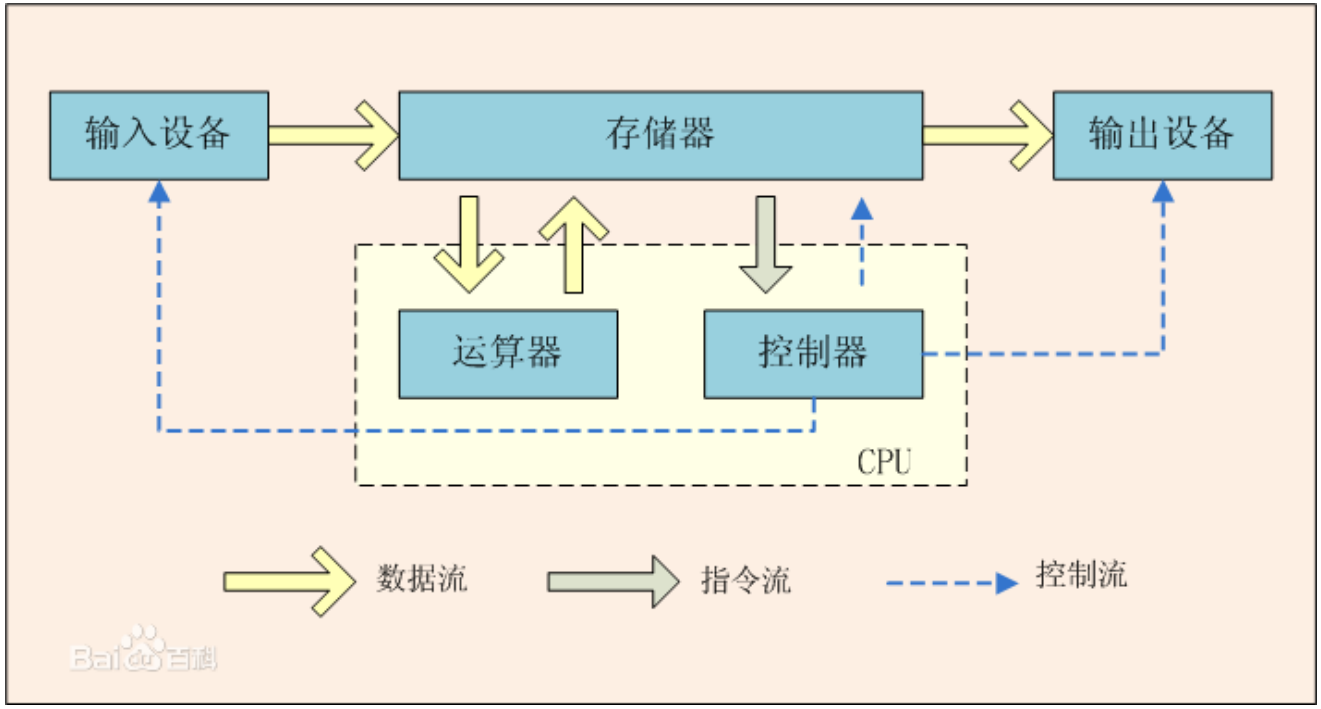
•英国数学家阿兰·图灵采用了完全不同的设计思路，提出了一种全新的抽象计算模型——图灵机



•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

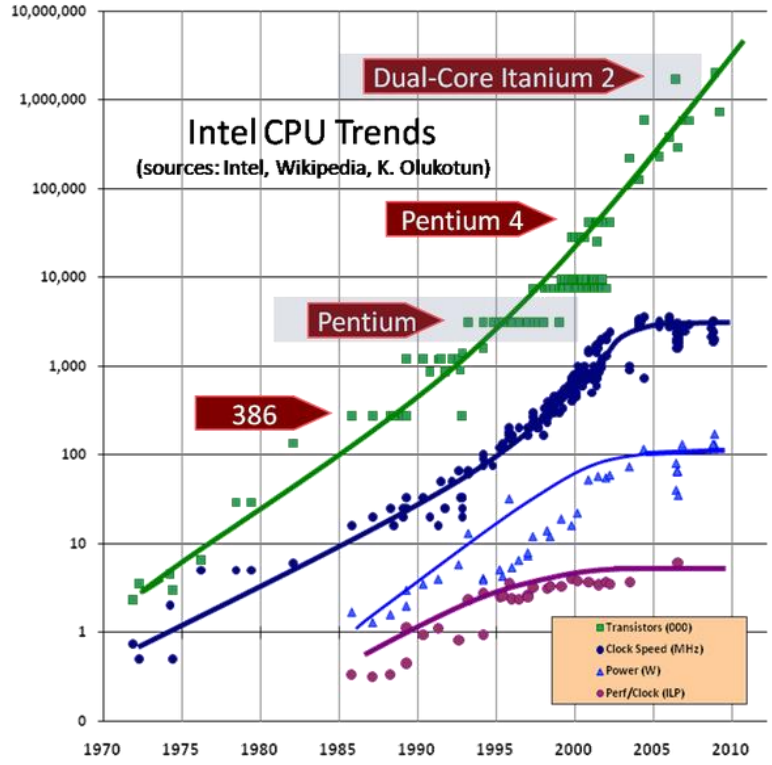
•函数式编程，又称泛函编程，它将计算机的计算视为数学上的函数计算

•函数编程语言最重要的基础是λ演算。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。

•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

Martin Odersky （ Scala之父）

詹姆斯·高斯林（ Java之父）

”If I were to pick a language to use today other than Java, it would be Scala.”
—James Gosling

2.1.4 Scala的安装

平台每年访问量超过100万次

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 Scala的安装

（1）Linux系统的安装

•Scala运行在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。本教程后续的Spark操作都是在Linux系统下进行的

•安装Linux系统，具体安装方法请参见教程官网的“实验指南”栏目的“Linux系统的安装”

2.1.4 Scala的安装

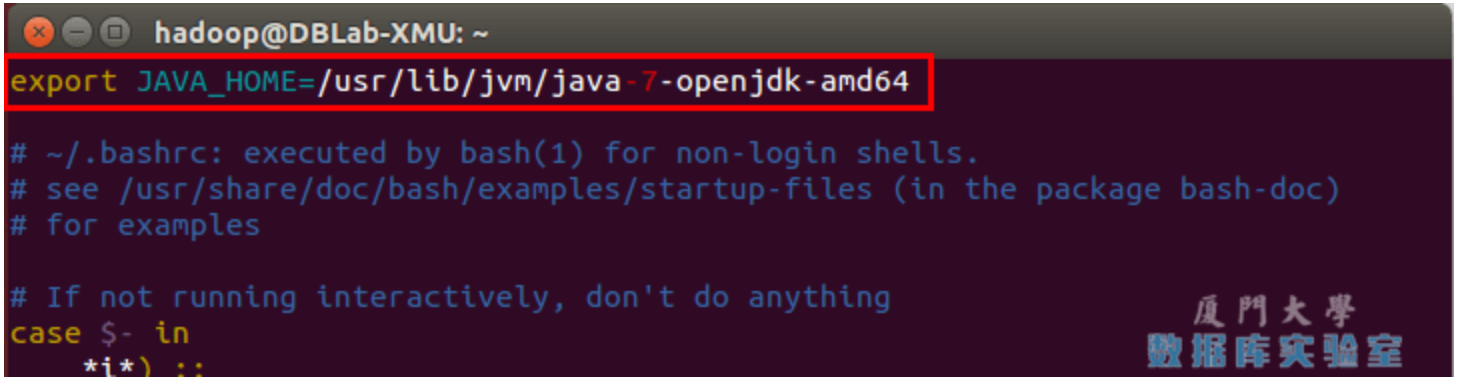
（2）在Linux系统中安装Java

第1种安装方式：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 Scala的安装

（2）在Linux系统中安装Java

第2种安装方式：直接通过命令安装 default-jdk

```
$ sudo apt-get install default-jre default-jdk
```

配置 JAVA_HOME 环境变量

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

在文件最前面添加如下单独一行（注意，等号“=”前后不能有空格），然后保存退出：

```
export JAVA_HOME=/usr/lib/jvm/default-java
```

使配置立即生效：

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

2.1.4 Scala的安装

（2）在Linux系统中安装Java

检验一下是否设置正确

```
$ echo $JAVA_HOME     # 检验变量值
$ java -version
$ $JAVA_HOME/bin/java -version  # 与直接执行 java -version 一样
```

如果设置正确的话，$JAVA_HOME/bin/java -version 会输出 java 的版本信息，且和 java -version 的输出结果一样

2.1.4 Scala的安装

（3）安装Scala

•本教程使用的Spark版本是2.1.0，其对应的Scala版本是2.11.8

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
```

2.1.5 HelloWorld

（1）通过HelloWorld程序了解Scala解释器的使用方法

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> 1+1
res0: Int = 2

scala> println("hello world")
hello world

scala> def show(x:Int){
     | println(s"I am $x")
     | }
show: (x: Int)Unit

scala>
```

可以使用命令“:quit”退出Scala解释器，如下所示：

```
scala> :quit
```

2.1.5 HelloWorld

（2）在Scala解释器中运行脚本文件

用“:load”命令导入脚本，一次运行多行程序：

使用文本编辑器（比如vim）创建一个代码文件Test.scala

```
//代码文件为/usr/local/scala/mycode/Test.scala
println("This is the first line")
println("This is the second line")
println("This is the third line")
```

在Scala REPL中执行如下命令运行该代码文件：

```
scala> :load  /usr/local/scala/mycode/Test.scala
Loading /usr/local/scala/mycode/Test.scala…
This is the first line
This is the second line
This is the third line
```

2.1.5 HelloWorld

（3）通过编译打包的方式运行Scala程序

```
//代码文件为/usr/local/scala/mycode/HelloWorld.scala
object HelloWorld {
  def main(args: Array[String]) {
  println("Hello, world!");
 }
}
```

使用scalac命令进行编译（编译的结果为Java字节码）

```
$ cd /usr/local/scala/mycode
$ scalac  HelloWorld.scala
```

使用scala或者java命令运行字节码文件

```
$ scala -classpath . HelloWorld
```

```
$ java  -classpath  .:/usr/local/scala/lib/scala-library.jar  HelloWorld
```

2.2 Scala基础

2.2.1基本数据类型和变量

2.2.2 输入输出

2.2.3 控制结构

2.2.4 数据结构

2.2.1基本数据类型和变量

1. 基本数据类型
2. 基本操作
3. 变量

2.2.1基本数据类型和变量

1.基本数据类型

•Scala的数据类型包括：Byte、Char、Short、Int、Long、Float、Double和Boolean（注意首字母大写）

•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基本数据类型和变量

2.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基本数据类型和变量

3.操作符

- 算术运算符：加(+)、减(-) 、乘(*) 、除(/) 、余数(%)；
- 关系运算符：大于(>)、小于(<)、等于(==)、不等于(!=)、大于等于(>=)、小于等于(<=)
- 逻辑运算符：逻辑与(&&)、逻辑或(||)、逻辑非(!)；
- 位运算符：按位与(&)、按位或(|)、按位异或(^)、按位取反(~)等
- 赋值运算符：=及其与其它运算符结合的扩展赋值运算符，例如+=、%=

操作符优先级：算术运算符 > 关系运算符 > 逻辑运算符 > 赋值运算符

2.2.1基本数据类型和变量

3.操作符

在Scala中，操作符就是方法
例如，5 + 3和(5).+(3)是等价的

`a 方法 b` 等价于 `a.方法(b)`

```
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
```

2.2.1基本数据类型和变量

3.操作符——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，而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基本数据类型和变量

4.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

基本语法：

```
val 变量名:数据类型 = 初始值
var 变量名:数据类型 = 初始值
```

4.变量

类型推断机制（type inference）：根据初始值自动推断变量的类型，使得定义变量时可以省略具体的数据类型及其前面的冒号

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

当然，我们也可以显式声明变量的类型：

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

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

myStr是val变量，因此，一旦初始化以后，就不能再次赋值

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

2.2.1基本数据类型和变量

4.变量

var变量初始化以后，可以再次赋值

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

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

注意：在REPL环境下，可以重复使用同一个变量名来定义变量，而且变量前的修饰符和其类型都可以不一致，REPL会以最新的一个定义为准

```
scala> val a = "Xiamen University"
a: String = Xiamen University
scala> var a = 50
a: Int = 50
```

2.2.2 输入输出

1. 控制台输入输出语句

2. 读写文件

2.2.2 输入输出

1. 控制台输入输出语句

•从控制台读入数据方法：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

2.2.2 输入输出

1. 控制台输入输出语句

从控制台读入数据方法

```
scala> import io.StdIn._
import io.StdIn._
scala> var i=readInt()
54
i: Int = 54
scala> var f=readFloat
1.618
f: Float = 1.618
scala> var b=readBoolean
true
b: Boolean = true
scala> var str=readLine("please input your name:")
please input your name:Li Lei
str: String = Li Lei
```

2.2.2 输入输出

1. 控制台输入输出语句

向控制台输出信息方法：

- print()和println()，可以直接输出字符串或者其它数据类型，其中println在末尾自动换行。

```
scala> val i=345
i: Int = 345
scala> print("i=");print(i)
//两条语句位于同一行，不能省略中间的分号
i=345
scala> println("hello");println("world")
hello
world
```

2.2.2 输入输出

1. 控制台输入输出语句

- C语言风格格式化字符串的printf()函数

```
scala> val i = 34
i: Int = 34
scala> val f=56.5
f: Double = 56.5
scala> printf("I am %d years old and weight %.1f Kg.","Li Lie",i,f)
I am 34 years old and weight 56.5 Kg.
```

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。

2.2.2 输入输出

1. 控制台输入输出语句

s字符串和f字符串：Scala提供的字符串插值机制，以方便在字符串字面量中直接嵌入变量的值。

基本语法：s " …$变量名… " 或 f " …$变量名%格式化字符… "

```
scala> val i=10
i: Int = 10
scala> val f=3.5
f: Double = 3.5452
scala> val s="hello"
s: String = hello
scala> println(s"$s:i=$i,f=$f")    //s插值字符串
hello:i=10,f=3.5452
scala> println(f"$s:i=$i%-4d,f=$f%.1f")   //f插值字符串
hello:i=10  ,f=3.5
```

2. 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，

PrintWriter类提供了print 和println两个写方法

```
scala> import java.io.PrintWriter
scala> val outputFile = new PrintWriter("test.txt")
scala> outputFile.println("Hello World")
scala> outputFile.print("Spark is good")
scala> outputFile.close()
```

2.2.2 输入输出

2. 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
```

2.2.3 控制结构

1. if条件表达式
2. while循环
3. for循环
4. 异常处理
5. 对循环的控制

2.2.3 控制结构

1. if条件表达式

```
if (表达式) {
        语句块1
}
else {
        语句块2
}
```

2.2.3 控制结构

```
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. while循环

```
while (表达式){
                    循环体
}
```

```
do{
                    循环体
}while (表达式)
```

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)
```

3. for循环

基本语法

```
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
```

2.2.3 控制结构

3. for循环

- “守卫(guard)”的表达式：过滤出一些满足条件的结果。

基本语法：

for （变量 <- 表达式 **if** 条件表达式） 语句块

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

```
2
4
```

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
```

2.2.3 控制结构

3. for循环

- **for推导式：** 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

for （变量 <- 表达式） **yield {**语句块**}**

```
scala> val r=for (i <- Array(1,2,3,4,5) if i%2==0) yield { println(i); i}
2
4
r: Array[Int] = Array(2,4)
```

4. 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
}
```

2.2.3 控制结构

5. 对循环的控制

•为了提前终止整个循环或者跳到下一个循环，Scala没有break和continue关键字

•Scala提供了一个Breaks类（位于包scala.util.control）。Breaks类有两个方法用于对循环结构进行控制，即breakable和break

```
breakable{
...
if(...) break
...
}
```

2.2.3 控制结构

5. 对循环的控制

```
//代码文件为/usr/local/scala/mycode/TestBreak.scala
import util.control.Breaks._ //导入Breaks类的所有方法
val array = Array(1,3,10,5,4)
breakable{
for(i<- array){
                if(i>5) break //跳出breakable，终止for循环，相当于Java中的break
println(i)
            }
}
// 上面的for语句将输出1，3

for(i<- array){
            breakable{
                        if(i>5) break
                       //跳出breakable，终止当次循环，相当于Java的continue
                        println(i)
            }
}// 上面的for语句将输出1，3，5，4
```

2.2.4 数据结构

1. 数组（Array）
2. 元组（Tuple）
3. 容器（Collection）
4. 序列（Sequence）
5. 集合（Set）
6. 映射（Map）
7. 迭代器（Iterator）

2.2.4 数据结构

1.数组（Array）

数组：一种可变的、可索引的、元素具有相同类型的数据集合

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

2.2.4 数据结构

1.数组（Array）

声明一个整型数组

```
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
```

2.2.4 数据结构

1.数组（Array）

声明一个字符串数组

```
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))
```

2.2.4 数据结构

1.数组（Array）

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")
```

2.2.4 数据结构

1.数组（Array）

多维数组的创建：调用Array的ofDim方法

```
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

2.2.4 数据结构

2. 元组（Tuple）

元组是对多个不同类型对象的一种简单封装。定义元组最简单的方法就是把多个元素用逗号分开并用圆括号包围起来。使用下划线“_”加上从1开始的索引值，来访问元组的元素

```
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
```

如果需要在方法里返回多个不同类型的对象，Scala可以通过返回一个元组来实现

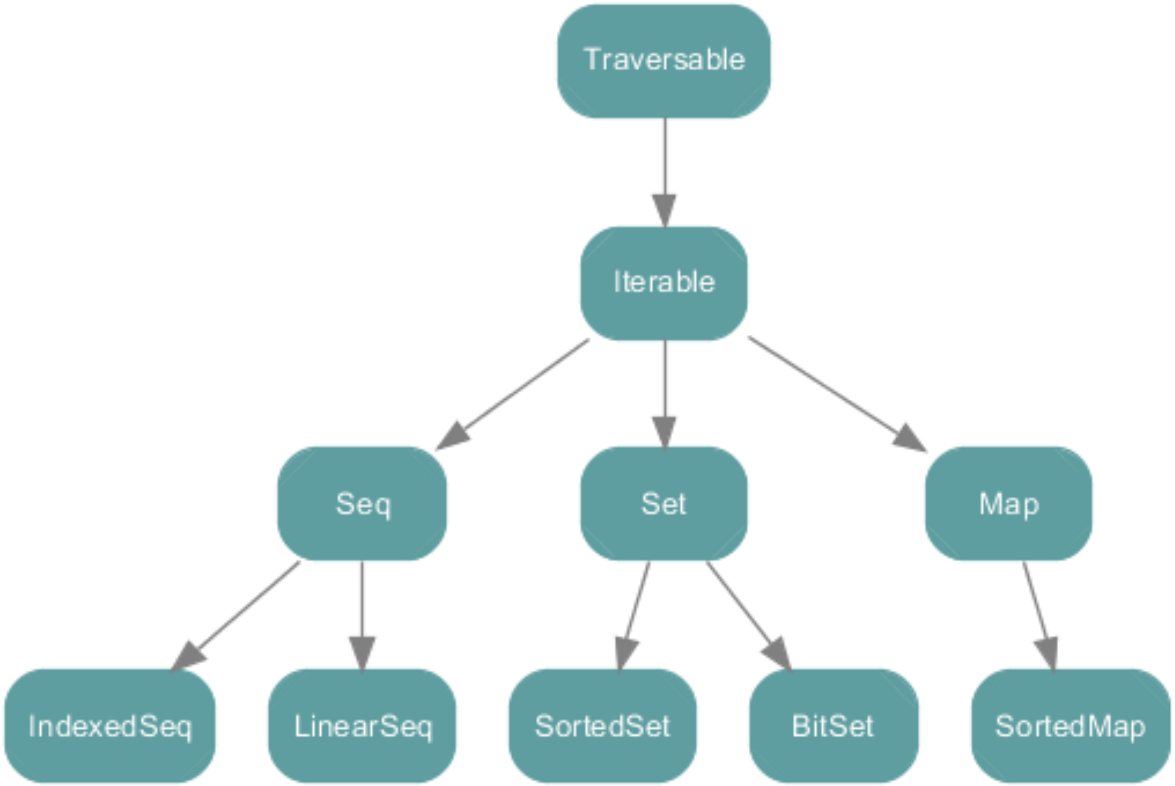
2.2.4 数据结构

3.容器（Collection）

- Scala提供了一套丰富的容器（collection）库，包括序列（Sequence）、集合（Set）、映射（Map）等
- Scala用了三个包来组织容器类，分别是scala.collection 、scala.collection.mutable和scala.collection.immutable
- scala.collection封装了可变容器和不可变容器的超类或特质，定义了可变容器和不可变容器的一些通用操作

3.容器（Collection）

scala.collection包中容器的宏观层次结构



2.2.4 数据结构

4.序列（Sequence）

序列（Sequence）：元素可以按照特定的顺序访问的容器。序列中每个元素均带有一个从0开始计数的固定索引位置

序列容器的根是collection.Seq特质。其具有两个子特质LinearSeq和IndexedSeq。LinearSeq序列具有高效的 head 和 tail 操作，而IndexedSeq序列具有高效的随机存储操作

实现了特质LinearSeq的常用序列有列表（List）和队列（Queue）。实现了特质IndexedSeq的常用序列有可变数组（ArrayBuffer）和向量（Vector）

2.2.4 数据结构

4.序列（Sequence）——列表（List）

•列表: 一种共享相同类型的不可变的对象序列。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")

2.2.4 数据结构

4.序列（Sequence）——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

注意：除了head、tail操作是常数时间O(1)，其它按索引访问的操作都需要从头开始遍历，因此是线性时间复杂度O(N)。

2.2.4 数据结构

4.序列（Sequence）——向量（Vector）

Vetor可以实现所有访问操作都是常数时间。

```
scala> val vec1=Vector(1,2)
vec1: scala.collection.immutable.Vector[Int] = Vector(1, 2, 3)
scala> val vec2 = 3 +: 4 +: vec1
vec2: scala.collection.immutable.Vector[Int] = Vector(3, 4, 1, 2)
scala> val vec3 = vec2 :+ 5
vec3: scala.collection.immutable.Vector[Int] = Vector(3, 4, 1, 2, 5)
scala> vec3(3)
res6: Int = 2
```

2.2.4 数据结构

4.序列（Sequence）——Range

- Range类：一种特殊的、带索引的不可变数字等差序列。其包含的值为从给定起点按一定步长增长(减小)到指定终点的所有数值
- 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

4.序列（Sequence）——Range

（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> val r=new Range(1,5,1)
```

```
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)
```

4.序列（Sequence）——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)
```

2.2.4 数据结构

5.集合（Set）

- 集合(set)：不重复元素的容器（collection）
- 列表（List）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

5.集合（Set）

•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"
```

2.2.4 数据结构

6.映射(Map)

•映射(Map)：一系列键值对的容器。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

•Scala 的映射包含了可变的和不可变的两种版本，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

2.2.4 数据结构

6.映射(Map)

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

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)
```

6.映射(Map)

可变的映射

```
import scala.collection.mutable.Map
val university2 = Map("XMU" -> "Xiamen University", "THU" -> "Tsinghua
University","PKU"->"Peking 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
```

2.2.4 数据结构

7.迭代器(Iterator)

•迭代器（Iterator）不是一个容器，而是提供了按顺序访问容器元素的数据结构

• 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())
}
```

建议：除next和hasnext方法外，在对一个迭代器调用了某个方法后，不要再次使用该迭代器

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 参数化类型

2.3.5 特质

2.3.6 模式匹配

2.3.7 包

2.3.1 类

1. 类的定义
2. 类成员的可见性
3. 方法的定义方式
4. 构造器

2.3.1 类

1.类的定义

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

字段用val或var关键字进行定义

方法定义：def 方法名(参数列表):返回结果类型={方法体}

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

2.3.1 类

1.类的定义

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

使用new关键字创建一个类的实例

```
val myCounter = new Counter
myCounter.value = 5 //访问字段
myCounter. increment(3) //调用方法
println(myCounter.current) //调用无参数方法时，可以省略方法名后的括号
```

2. 类成员的可见性

•Scala类中所有成员的默认可见性为公有，任何作用域内都能直接访问公有成员

•除了默认的公有可见性，Scala也提供private和protected，其中，private成员只对本类型和嵌套类型可见；protected成员对本类型和其继承类型都可见

2. 类成员的可见性

为了避免直接暴露public字段，建议将字段设置为private，对于private字段，Scala采用类似Java中的getter和setter方法，定义了两个成对的方法value和value_=进行读取和修改

```
//代码文件为/usr/local/scala/mycode/Counter.scala
class Counter {
    private var privateValue = 0
    def value = privateValue
    def value_=(newValue: Int) {
        if (newValue > 0) privateValue = newValue
        }
    def increment(step: Int): Unit = {value += step}
    def current():Int = {value}
}
```

2. 类成员的可见性

```
scala> :load /usr/local/scala/mycode/Counter.scala
Loading /usr/local/scala/mycode/Counter.scala…
defined class Counter
scala> val myCounter = new Counter
myCounter: Counter = Counter@f591271
scala> myCounter.value_=(3) //为privateValue设置新的值
scala> println(myCounter.value)//访问privateValue的当前值
3
```

Scala语法中有如下规范，当编译器看到以value和value_=这种成对形式出现的方法时，它允许用户去掉下划线_，而采用类似赋值表达式的形式

```
myCounter.value= 3 // 等效于myCounter.value_=(3)
```

3. 方法的定义方式

基本语法：def 方法名(参数列表):返回结果类型={方法体}

- 方法参数前不能加上val或var，所有的方法参数都是不可变类型
- 无参数的方法定义时可以省略括号，这时调用时也不能带有括号；如果定义时带有括号，则调用时可以带括号，也可以不带括号
- 方法名后面的圆括号()可以用大括号{}来代替
- 如果方法只有一个参数，可以省略点号（.）而采用中缀操作符调用方法
- 如果方法体只有一条语句，可以省略方法体两边的大括号

3. 方法的定义方式

```
//代码文件为/usr/local/scala/mycode/Counter1.scala
class Counter {
    var value = 0
    def increment(step:Int):Unit = {value += step}
    def current:Int= value
    def getValue():Int = value
}
```

2.3.1 类

```
//代码文件为/usr/local/scala/mycode/Counter1.scala
class Counter {
    var value = 0
    def increment(step:Int):Unit = {value += step}
    def current:Int= value
    def getValue():Int = value
}
```

3. 方法的定义方式

```
scala> :load /usr/local/scala/mycode/Counter1.scala
Loading /usr/local/scala/mycode/Counter1.scala…
defined class Counter
scala> val c=new Counter
c: Counter = Counter@30ab4b0e
scala> c increment 5 //中缀调用法
scala> c.getValue()     //getValue定义中有括号，可以带括号调用
res0: Int = 0
scala> c.getValue // getValue定义中有括号，也可不带括号调用
res1: Int = 0
scala> c.current() // current定义中没有括号，不可带括号调用
<console>:13: error: Int does not take parameters
      c.current()
              ^
scala> c.current  // current定义中没有括号，只能不带括号调用
res3: Int = 0
```

2.3.1 类

3. 方法的定义方式

- 当方法的返回结果类型可以从最后的表达式推断出时，可以省略结果类型
- 如果方法返回类型为Unit，可以同时省略返回结果类型和等号，但不能省略大括号

```
class Counter {
    var value = 0
    def increment(step:Int) { value += step }//赋值表达式的值为Unit类型
    def current()= value //根据value的类型自动推断出返回类型为Int型
}
```

2.3.1 类

4. 构造器

•Scala类的定义主体就是类的构造器，称为主构造器。在类名之后用圆括号列出主构造器的参数列表

•主构造器的参数前可以使用val或var关键字，Scala内部将自动为这些参数创建私有字段，并提供对应的访问方法

4. 构造器

```
scala> class Counter(var name:String) //定义一个带字符串参数的简单类
defined class Counter
scala> var mycounter = new Counter("Runner")
mycounter: Counter = Counter@17fcc4f7
scala> println(mycounter.name) //调用读方法
Runner
scala> mycounter.name_=("Timer") //调用写方法
scala> mycounter.name = "Timer"// 更直观地调用写方法，和上句等效
mycounter.name: String = Timer
```

- 如果不希望将构造器参数成为类的字段，只需要省略关键字var或者val

4. 构造器

- Scala类可以包含零个或多个辅助构造器（auxiliary constructor）。辅助构造器使用this进行定义，this的返回类型为Unit
- 每个辅助构造器的第一个表达式必须是调用一个此前已经定义的辅助构造器或主构造器，调用的形式为“this(参数列表)”

4. 构造器

```
//代码文件为/usr/local/scala/mycode/Counter2.scala
class Counter {
    private var value = 0
    private var name = ""
    private var step = 1 //计算器的默认递进步长
    println("the main constructor")
    def this(name: String){ //第一个辅助构造器
        this() //调用主构造器
        this.name = name
        printf("the first auxiliary constructor,name:%s\n",name)
    }
    def this (name: String,step: Int){ //第二个辅助构造器
        this(name) //调用前一个辅助构造器
        this.step = step
       printf("the second auxiliary constructor,name:%s,step:%d\n",name,step)
    }
    def increment(step: Int): Unit = { value += step}
    def current(): Int = {value}
}
```

4. 构造器

```
scala> :load /usr/local/scala/mycode/Counter2.scala
Loading /usr/local/scala/mycode/Counter2.scala…
defined class Counter
scala> val c1=new Counter
the main constructor
c1: Counter = Counter@319c6b2

scala> val c2=new Counter("the 2nd Counter")
the main constructor
the first auxiliary constructor,name:the 2nd Counter
c2: Counter = Counter@4ed6c602

scala> val c3=new Counter("the 3rd Counter",2)
the main constructor
the first auxiliary constructor,name:the 3rd Counter
the second auxiliary constructor,name:the 3rd Counter,step:2
c3: Counter = Counter@64fab83b
```

2.3.2 对象

1. 单例对象
2. apply方法
3. update方法
4. unapply方法

2.3.2 对象

1.单例对象

- Scala采用单例对象（singleton object）来实现与Java静态成员同样的功能
- 使用object 关键字定义单例对象

```
//代码文件为/usr/local/scala/mycode/Person.scala
object Person {
    private var lastId = 0  //一个人的身份编号
    def newPersonId() = {
        lastId +=1
        lastId
    }
}
```

2.3.2 对象

1.单例对象

单例对象的使用与一个普通的类实例一样：

```
//代码文件为/usr/local/scala/mycode/Person.scala
object Person {
    private var lastId = 0  //一个人的身份编号
    def newPersonId() = {
        lastId +=1
        lastId
    }
}
```

```
scala> :load /usr/local/scala/mycode/Person.scala
Loading /usr/local/scala/mycode/Person.scala…
defined object Person
scala> printf("The first person id: %d.\n",Person.newPersonId())
The first person id: 1.
scala> printf("The second person id：%d.\n",Person.newPersonId())
The second person id：2.
scala> printf("The third person id: %d.\n",Person.newPersonId())
The third person id: 3.
```

2.3.2 对象

1.单例对象

伴生对象和孤立对象

- 当一个单例对象和它的同名类一起出现时，这时的单例对象被称为这个同名类的“伴生对象”（companion object）。相应的类被称为这个单例对象的“伴生类”
- 类和它的伴生对象必须存在于同一个文件中，可以相互访问私有成员
- 没有同名类的单例对象，被称为孤立对象（standalone object）。一般情况下，Scala程序的入口点main方法就是定义在一个孤立对象里

2.3.2 对象

1.单例对象

伴生对象和孤立对象

```
//代码文件为/usr/local/scala/mycode/Person1.scala
class Person(val name:String){
    private val id = Person.newPersonId() //调用了伴生对象中的方法
    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() = {
        lastId +=1
        lastId
    }
    def main(args: Array[String]) {
        val person1 = new Person("Lilei")
        val person2 = new Person("Hanmei")
        person1.info()
        person2.info()
    }
}
```

```
$ scalac /usr/local/scala/mycode/Person1.scala
$ scala –classpath . Person
The id of Lilei is 1.
The id of Hanmei is 2.
```

2.3.2 对象

2. apply方法

- 思考下行代码的执行过程：

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

- Scala自动调用Array类的伴生对象Array中的一个称为apply的方法，来创建一个Array对象myStrArr
- apply方法调用约定：用括号传递给类实例或单例对象名一个或多个参数时，Scala 会在相应的类或对象中查找方法名为apply且参数列表与传入的参数一致的方法，并用传入的参数来调用该apply方法

2. apply方法

例：类中的apply方法

```
//代码文件为/usr/local/scala/mycode/TestApplyClass.scala
class TestApplyClass {
    def apply(param: String){
                println("apply method called: " + param)
}
}
```

```
scala> :load /usr/local/scala/mycode/TestApplyClass.scala
Loading /usr/local/scala/mycode/TestApplyClass.scala…
defined class TestApplyClass
scala> val myObject = new TestApplyClass
myObject: TestApplyClass = TestApplyClass@11b352e9
scala> myObject("Hello Apply")// 自动调用类中定义的apply方法，等同于下句
apply method called: Hello Apply
scala> myObject.apply("Hello Apply")  //手动调用apply方法
apply method called: Hello Apply
```

2.3.2 对象

2. apply方法

伴生对象中的apply方法：将所有类的构造方法以apply方法的形式定义在伴生对象中，这样伴生对象就像生成类实例的工厂，而这些apply方法也被称为工厂方法

```
//代码文件为/usr/local/scala/mycode/MyTestApply.scala
class Car(name: String) {
    def info() {
        println("Car name is "+ name)
    }
}
object Car {
    def apply(name: String) = new Car(name) //调用伴生类Car的构造方法
}
object MyTestApply{
    def main (args: Array[String]) {
    val mycar = Car("BMW") //调用伴生对象中的apply方法
    mycar.info() //输出结果为“Car name is BMW”
    }
}
```

2.3.2 对象

2. apply方法

为什么要设计apply方法？

- 保持对象和函数之间使用的一致性
- 面向对象：“对象.方法” VS 数学：“函数(参数)”
- Scala中一切都是对象，包括函数也是对象。Scala中的函数既保留括号调用样式，也可以使用点号调用形式，其对应的方法名即为apply

```
scala> def add=(x:Int,y:Int)=>x+y  //add是一个函数
add: (Int, Int) => Int
scala> add(4,5)   //采用数学界的括号调用样式
res2: Int = 9
scala> add.apply(4,5) //add也是对象，采用点号形式调用apply方法
res3: Int = 9
```

2. apply方法

为什么要设计apply方法？

- **Scala的对象也可以看成函数，前提是该对象提供了apply方法**

```
//代码文件为/usr/local/scala/mycode/MyTestApply.scala
class Car(name: String) {
    def info() {
        println("Car name is "+ name)
    }
}
object Car {
    def apply(name: String) = new Car(name) //调用伴生类Car的构造方法
}
object MyTestApply{
    def main (args: Array[String]) {
    val mycar = Car("BMW") //调用伴生对象中的apply方法
    mycar.info() //输出结果为“Car name is BMW”
    }
}
```

2.3.2 对象

3. update方法

与apply方法类似的update方法也遵循相应的调用约定：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，并将括号里的参数和等号右边的值一起作为update方法的输入参数来执行调用

```
scala>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
scala>myStrArr(0) = "BigData" //实际上，调用了伴生类Array中的update方法，执行myStrArr.update(0,"BigData")
scala>myStrArr(1) = "Hadoop" //实际上，调用了伴生类Array中的update方法，执行myStrArr.update(1,"Hadoop")
scala>myStrArr(2) = "Spark" //实际上，调用了伴生类Array中的update方法，执行myStrArr.update(2,"Spark")
```

2.3.2 对象

4. unapply方法

•unapply方法用于对对象进行解构操作，与apply方法类似，该方法也会被自动调用

•可以认为unapply方法是apply方法的反向操作，apply方法接受构造参数变成对象，而unapply方法接受一个对象，从中提取值

2.3.2 对象

4. unapply方法

```
//代码文件为/usr/local/scala/mycode/TestUnapply.scala
class Car(val brand:String,val price:Int) {
    def info() {
        println("Car brand is "+ brand+" and price is "+price)
    }
}
object Car{
                def apply(brand:String,price:Int)= {
                        println("Debug:calling apply ... ")
                        new Car(brand,price)
                }
                def unapply(c:Car):Option[(String,Int)]={
                        println("Debug:calling unapply ... ")
                        Some((c.brand,c.price))
                }
}
object TestUnapply{
      def main (args: Array[String]) {
              var Car(carbrand,carprice) = Car("BMW",800000)
              println("brand: "+carbrand+" and carprice: "+carprice)
       }
}
```

2.3.3 继承

1. 抽象类
2. 扩展类
3. Scala的类层次结构
4. Option类

2.3.3 继承

1. 抽象类

如果一个类包含没有实现的成员，则必须使用abstract关键词进行修饰，定义为抽象类

```
abstract class Car(val name:String) {
    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
    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，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，否则编译会报错

2.3.3 继承

2. 扩展类

Scala只支持单一继承，而不支持多重继承。在类定义中使用extends关键字表示继承关系。定义子类时，需要注意：

- 重载父类的抽象成员（包括字段和方法）时，override关键字是可选的；而重载父类的非抽象成员时，override关键字是必选的
- 只能重载val类型的字段，而不能重载var类型的字段。因为var类型本身就是可变的，所以，可以直接修改它的值，无需重载

2.3.3 继承

2. 扩展类

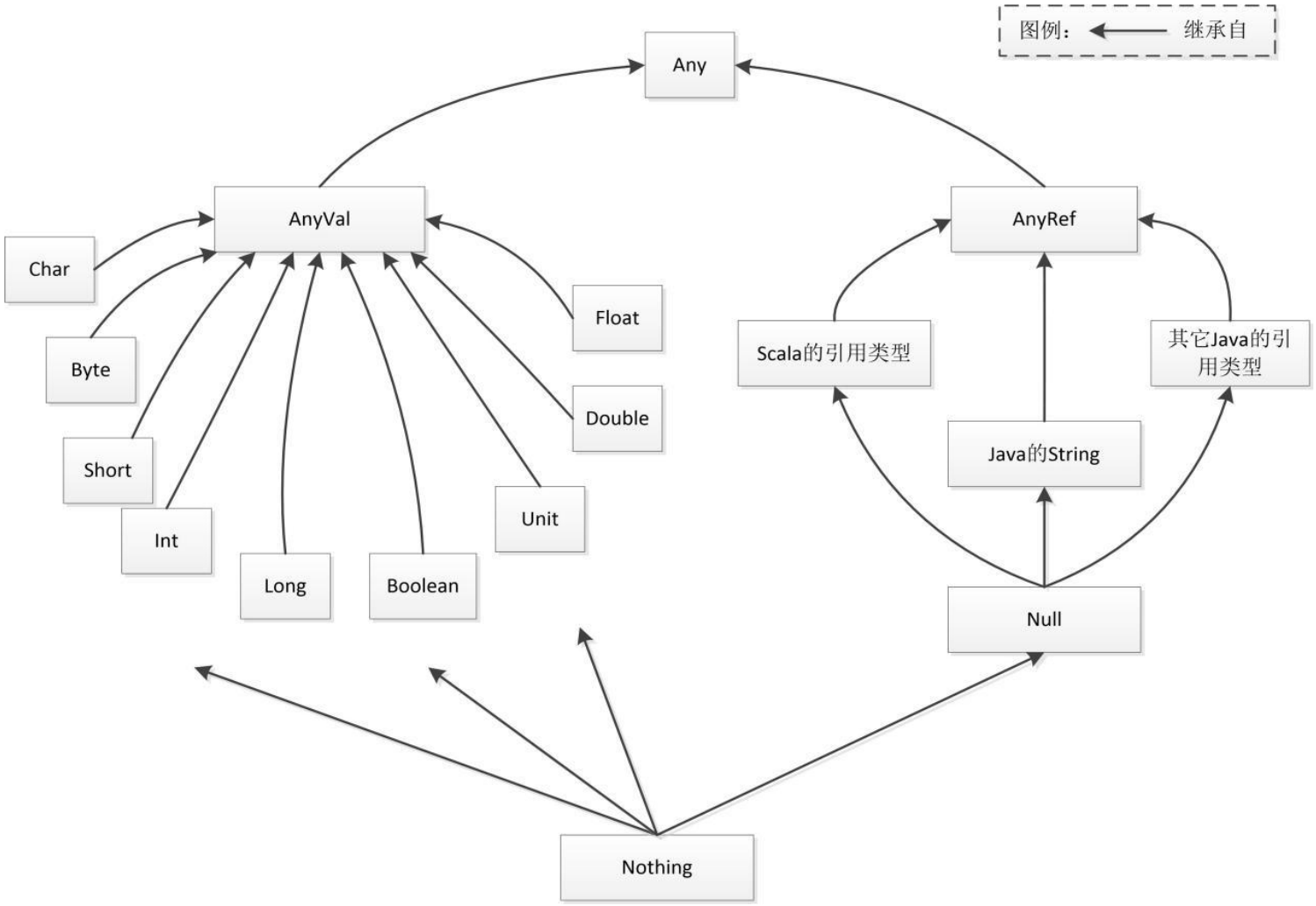
```
abstract class Car{
   val carBrand: String
     def info()
     def greeting() {println("Welcome to my car!")}
}
class BMWCar extends Car {
    override val carBrand = "BMW"
    def info() {printf("This is a %s car. It is expensive.\n", carBrand)}
    override def greeting() {println("Welcome to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

编译执行后，结果为：

```
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.
```

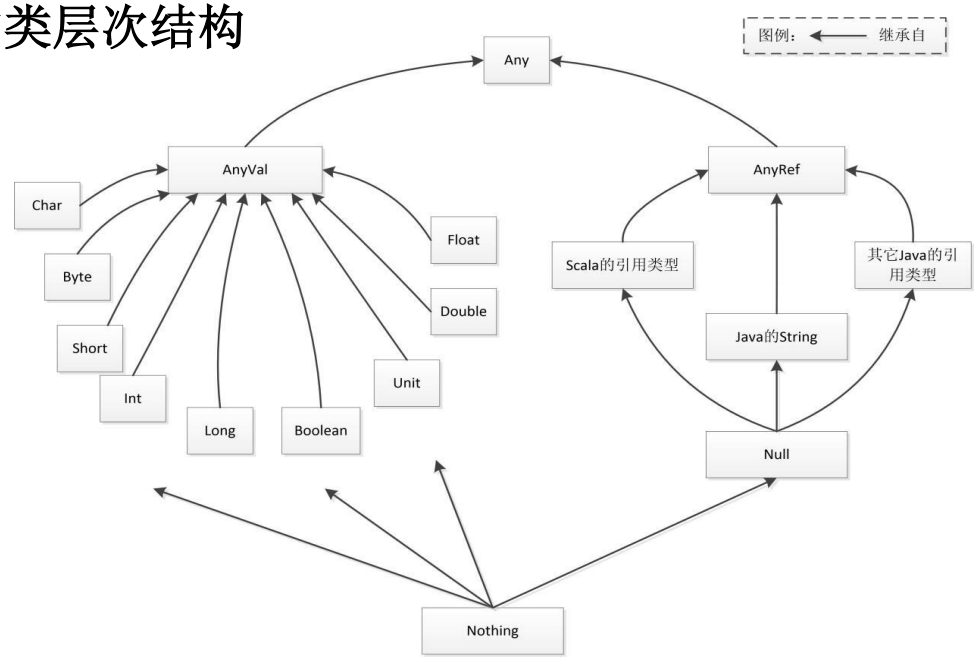
2.3.3 继承

3. Scala的类层次结构



2.3.3 继承

3. Scala的类层次结构



- Null是所有引用类型的子类，其唯一的实例为null，表示一个“空”对象，可以赋值给任何引用类型的变量，但不能赋值给值类型的变量
- Nothing是所有其它类型的子类，包括Null。Nothing没有实例，主要用于异常处理函数的返回类型

2.3.3 继承

4. Option类

- Scala提供null是为了实现在JVM与其它Java库的兼容性，但是，除非明确需要与Java库进行交互，否则，Scala建议尽量避免使用这种可能带来bug的null，而改用Option类
- Option是一个抽象类，有一个具体的子类Some和一个对象None，其中，前者表示有值的情形，后者表示没有值
- 当方法不确定是否有对象返回时，可以让返回值类型为Option[T]，其中，T为类型参数。对于这类方法，如果确实有T类型的对象需要返回，会将该对象包装成一个Some对象并返回；如果没有值需要返回，将返回None

2.3.3 继承

4. Option类

```
scala> case class Book(val name:String,val price:Double)
defined class Book
scala> val books=Map("hadoop"->Book("Hadoop",35.5),
     | "spark"->Book("Spark",55.5),
     | "hbase"->Book("Hbase",26.0)) //定义一个书名到书对象的映射
books: scala.collection.immutable.Map[String,Book] =…
scala> books.get("hadoop") //返回该键所对应值的Some对象
res0: Option[Book] = Some(Book(Hadoop,35.5))
scala> books.get("hive") // 不存在该键，返回None对象
res1: Option[Book] = None
scala> books.get("hadoop").get //Some对象的get方法返回其包装的对象
res2: Book = Book(Hadoop,35.5)
scala> books.get("hive").get // None对象的get方法会抛出异常
java.util.NoSuchElementException: None.get
  …
scala> books.get("hive").getOrElse(Book("Unknown name",0))
res4: Book = Book(Unknown name,0.0)
```

2.3.4 特质（trait）

1. 特质概述
2. 特质的定义
3. 把特质混入类中
4. 把多个特质混入类中

2.3.4 特质（trait）

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
- 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 特质（trait）

2.特质的定义

使用关键字trait定义特质

```
trait Flyable {
        var maxFlyHeight:Int  //抽象字段
        def fly() //抽象方法
        def breathe(){ //具体的方法
            println("I can breathe")
        }
}
```

- 特质既可以包含抽象成员，也可以包含非抽象成员。包含抽象成员时，不需要abstract关键字
- 特质可以使用extends继承其它的特质

2.3.4 特质（trait）

3. 把特质混入类中

可以使用extends或with关键字把特质混入类中
如果特质中包含抽象成员，则该类必须为这些抽象成员提供具体实现，除非该类被定义为抽象类

```
class Bird(flyHeight:Int) extends Flyable{
    var maxFlyHeight:Int = flyHeight  //重载特质的抽象字段
    def fly(){
        printf("I can fly at the height of %d.",maxFlyHeight)
     } //重载特质的抽象方法
}
```

2.3.4 特质（trait）

3. 把特质混入类中

把上面定义的特质Flyable和类Bird封装到一个代码文件Bird.scala中：

```
//代码文件为/usr/local/scala/mycode/Bird.scala
trait Flyable {
        var maxFlyHeight:Int  //抽象字段
        def fly() //抽象方法
        def breathe(){ //具体的方法
                println("I can breathe")
        }
 }
class Bird(flyHeight:Int) extends Flyable{
        var maxFlyHeight:Int = flyHeight  //重载特质的抽象字段
        def fly(){
             printf("I can fly at the height of %d",maxFlyHeight)
        } //重载特质的抽象方法
}
```

2.3.4 特质（trait）

3. 把特质混入类中

在Scala REPL中执行如下代码并观察效果：

```
scala> :load /usr/local/scala/mycode/Bird.scala
Loading /usr/local/scala/mycode/Bird.scala...
defined trait Flyable
defined class Bird

scala> val b=new Bird(100)
b: Bird = Bird@43a51d00

scala> b.fly()
I can fly at the height of 100
scala> b.breathe()
I can breathe
```

2.3.4 特质（trait）

3. 把特质混入类中

如果要混入多个特质，可以连续使用多个with

```
//代码文件为/usr/local/scala/mycode/Bird2.scala
trait Flyable {
      var maxFlyHeight:Int  //抽象字段
      def fly() //抽象方法
      def breathe(){ //具体的方法
           println("I can breathe")
      }
 }
trait HasLegs {
      val legs:Int   //抽象字段
      def move(){printf("I can walk with %d legs",legs)}
}
class Animal(val category:String){
      def info(){println("This is a "+category)}
}
class Bird(flyHeight:Int) extends Animal("Bird") with Flyable with HasLegs{
        var maxFlyHeight:Int = flyHeight //重载特质的抽象字段
        val legs=2 //重载特质的抽象字段
        def fly(){
             printf("I can fly at the height of %d",maxFlyHeight)
        }//重载特质的抽象方法
}
```

2.3.4 特质（trait）

3. 把特质混入类中

可以在Scala REPL中执行如下代码查看执行效果：

```
scala> :load /usr/local/scala/mycode/Bird2.scala
Loading /usr/local/scala/mycode/Bird2.scala...
defined trait Flyable
defined trait HasLegs
defined class Animal
defined class Bird

scala> val b=new Bird(108)
b: Bird = Bird@126675fd
scala> b.info
This is a Bird
scala> b.fly
I can fly at the height of 108
scala> b.move
I can walk with 2 legs
```

2.3.5 模式匹配

1. match语句
2. case类

2.3.5 模式匹配

1. match语句

最常见的模式匹配是match语句，match语句用在当需要从多个分支中进行选择的场景。

```
//代码文件为/usr/local/scala/mycode/TestMatch.scala
import scala.io.StdIn._
println("Please input the score:")
val grade=readChar()
grade match{
        case 'A' => println("85-100")
        case 'B' => println("70-84")
        case 'C' => println("60-69")
        case 'D' => println("<60")
        case _ => println("error input!")
}
```

- 通配符_相当于Java中的default分支
- match结构中不需要break语句来跳出判断，Scala从前往后匹配到一个分支后，会自动跳出判断

2.3.5 模式匹配

1. match语句

case后面的表达式可以是任何类型的常量，而不要求是整数类型

```
//代码文件为/usr/local/scala/mycode/TestMatch1.scala
import scala.io.StdIn._
println("Please input a country:")
val country=readLine()
country match{
                    case "China" => println("中国")
                    case "America" => println("美国")
                    case "Japan" => println("日本")
                    case _ => println("我不认识!")
}
```

2.3.5 模式匹配

1. match语句

除了匹配特定的常量，还能匹配某种类型的所有值

```
//代码文件为/usr/local/scala/mycode/TestMatch2.scala
for (elem <- List(6,9,0.618,"Spark","Hadoop",'Hello)){
            val str  = elem match  {
                case i: Int => i + " is an int value."//匹配整型的值，并赋值给i
                case d: Double => d + " is a double value." //匹配浮点型的值
                case "Spark"=>"Spark is found." //匹配特定的字符串
                case s: String => s + " is a string value." //匹配其它字符串
                case _ =>"unexpected value："+ elem  //与以上都不匹配
            }
            println(str)
}
```

```
6 is an int value.
9 is an int value.
0.618 is a double value.
Spark is found.
Hadoop is a string value.
unexpected value：'Hello
```

2.3.5 模式匹配

1. match语句

可以在match表达式的case中使用守卫式（guard）添加一些过滤逻辑

```
//代码文件为/usr/local/scala/mycode/TestMatch3.scala
for (elem <- List(1,2,3,4)){
    elem match {
        case _ if (elem%2==0) => println(elem + " is even.")
        case _ => println(elem + " is odd.")
    }
}
```

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

2.3.5 模式匹配

2. case类

- case类是一种特殊的类，它们经过优化以被用于模式匹配
- 当定义一个类时，如果在class关键字前加上case关键字，则该类称为case类
- Scala为case类自动重载了许多实用的方法，包括toString、equals和hashcode方法
- Scala为每一个case类自动生成一个伴生对象，其包括模板代码
  - 1个apply方法，因此，实例化case类的时候无需使用new关键字
  - 1个unapply方法，该方法包含一个类型为伴生类的参数，返回的结果是Option类型，对应的类型参数是N元组，N是伴生类中主构造器参数的个数。Unapply方法用于对对象进行解构操作，在case类模式匹配中，该方法被自动调用，并将待匹配的对象作为参数传递给它

2.3.5 模式匹配

2. case类

例如，假设有如下定义的一个case类：

```
case class Car(brand: String, price: Int)
```

则编译器自动生成的伴生对象是：

```
object  Car{
        def apply(brand:String,price:Int)= new Car(brand,price)
        def unapply(c:Car):Option[(String,Int)]=Some((c.brand,c.price))
}
```

2.3.5 模式匹配

2. case类

```
//代码文件为/usr/local/scala/mycode/TestCase.scala
case class Car(brand: String, price: Int)
val myBYDCar = Car("BYD", 89000)
val myBMWCar = Car("BMW", 1200000)
val myBenzCar = 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 it?")
      }
 }
```

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

2.3.6 包

1. 包的定义
2. 引用包成员

2.3.6 包

1. 包的定义

- 为了解决程序中命名冲突问题，Scala也和Java一样采用包(package)来层次化、模块化地组织程序
- 包可以包含类、对象和特质的定义，但是不能包含函数或变量的定义

```
package  autodepartment
class MyClass
```

- 为了在任意位置访问MyClass类，需要使用autodepartment.MyClass

2.3.6 包

1. 包的定义

- 通过在关键字package后面加大括号，可以将程序的不同部分放在不同的包里。这样可以实现包的嵌套，相应的作用域也是嵌套的

```
package xmu {
      package autodepartment {
          class ControlCourse{
                        ...
          }
     }
     package csdepartment {
          class  OSCourse{
              val cc = new autodepartment.ControlCourse
          }
     }
}
```

2.3.6 包

2.引用包成员

- 可以用import 子句来引用包成员，这样可以简化包成员的访问方式

```
import xmu.autodepartment.ControlCourse
class MyClass{
        var myos=new ControlCourse
}
```

- 使用通配符下划线（_）引入类或对象的所有成员

```
import scala.io.StdIn._
var i=readInt()
var f=readFloat()
var str=readLine()
```

- Scala 隐式地添加了一些引用到每个程序前面，相当于每个Scala程序都隐式地以如下代码开始：

```
import java.lang._
import scala._
import Predef._
```

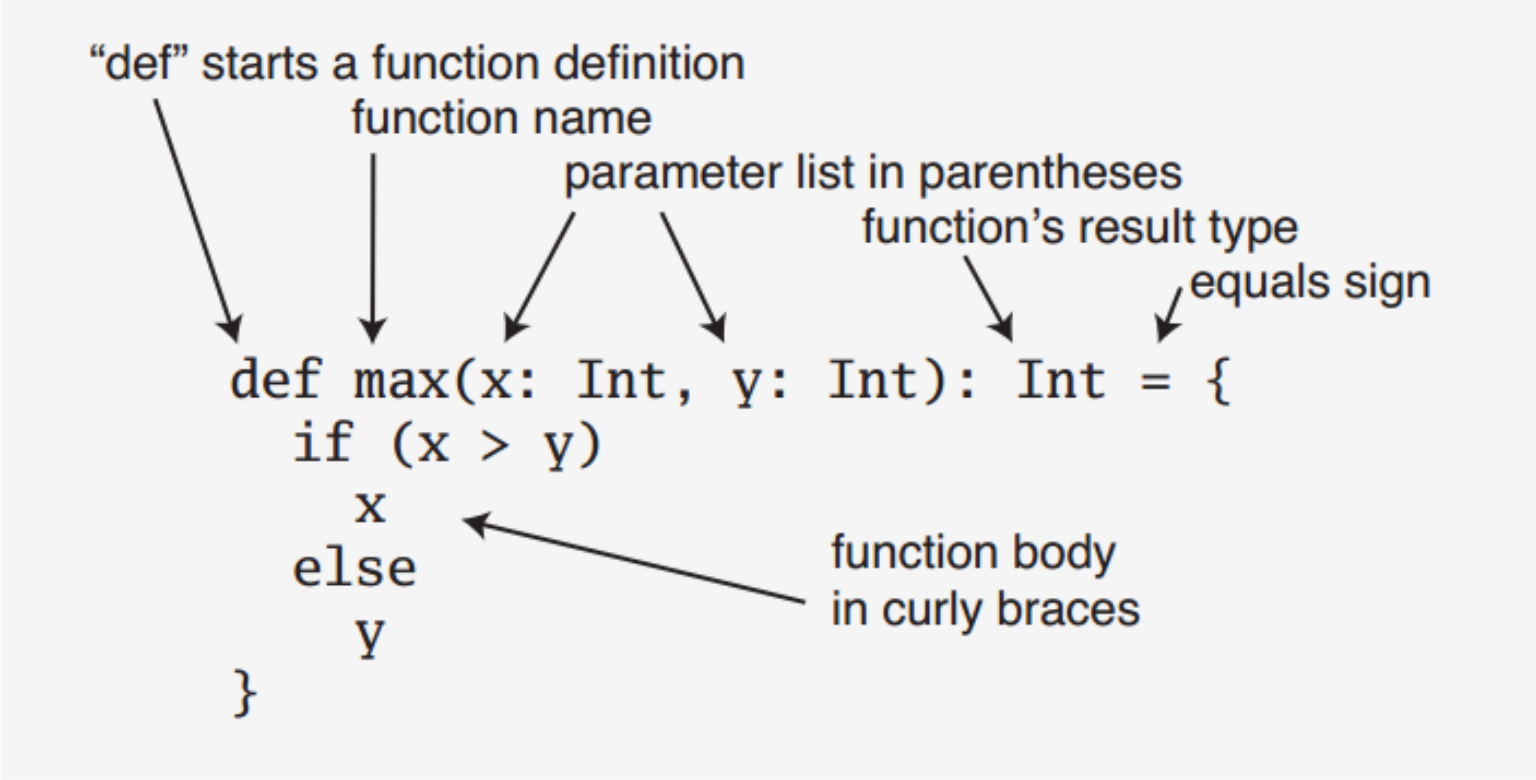
2.4 函数式编程基础

2.4.1 函数定义与使用
2.4.2 高阶函数
2.4.3 针对容器的操作
2.4.4 函数式编程实例WordCount

2.4.1 函数定义与使用

定义函数最通用的方法是作为某个类或者对象的成员，这种函数被称为方法，其定义的基本语法为

def 方法名(参数列表):结果类型={方法体}



2.4.1 函数定义与使用

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 函数定义与使用

- 函数字面量可以体现函数式编程的核心理念
- 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 函数定义与使用

下面一点点引导大家更好地理解函数的“类型”和“值”的概念
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 函数定义与使用

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 函数定义与使用

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
```

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 函数定义与使用

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“Lambda表达式”。“Lambda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
```

2.4.1 函数定义与使用

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
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
6
```

2.4.1 函数定义与使用

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 函数定义与使用

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

2.4.1 函数定义与使用

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 函数定义与使用

- 当函数的每个参数在函数字面量内仅出现一次，可以省略“=>”并用下划线“_”作为参数的占位符来简化函数字面量的表示，第一个下划线代表第一个参数，第二个下划线代表第二个参数，依此类推

```
scala> val counter = (_:Int) + 1 //有类型时括号不能省略，等效于“x:Int=>x+1”
counter: Int => Int = <function1>
scala> val add = (_:Int) + (_:Int) //等效于“(a:Int,b:Int)=>a+b”
add: (Int, Int) => Int = <function2>
scala> val m1=List(1,2,3)
m1: List[Int] = List(1, 2, 3)
scala>val m2=m1.map(_*2)//map接受一个函数作为参数，相当于
“m1.map(x=>x*2)”，参数的类型可以根据m1的元素类型推断出，所以可以省略。
m2: List[Int] = List(2, 4, 6)
```

2.4.2 高阶函数

- 高阶函数：当一个函数包含其它函数作为其参数或者返回结果为一个函数时，该函数被称为高阶函数

例：假设需要分别计算从一个整数到另一个整数的“连加和”、“平方和”以及“2的幂次和”

方案一：不采用高阶函数

```
def powerOfTwo(x: Int): Int = {if(x == 0) 1 else 2 * powerOfTwo(x-1)}
def sumInts(a: Int, b: Int): Int = {
                    if(a > b) 0 else a + sumInts(a + 1, b)
}
def sumSquares(a: Int, b: Int): Int = {
                    if(a > b) 0 else a*a + sumSquares(a + 1, b)
}
def sumPowersOfTwo(a: Int, b: Int): Int = {
                    if(a > b) 0 else powerOfTwo(a) + sumPowersOfTwo(a+1, b)
}
```

2.4.2 高阶函数

例：假设需要分别计算从一个整数到另一个整数的“连加和”、“平方和”以及“2的幂次和”

方案二：采用高阶函数

```
def sum(f: Int => Int, a: Int, b: Int):Int = {
        if(a > b) 0 else f(a) + sum(f, a+1, b)
}
```

```
scala> sum(x=>x,1,5) //直接传入一个匿名函数
//且省略了参数x的类型，因为可以由sum的参数类型推断出来
res8: Int = 15
scala> sum(x=>x*x,1,5) //直接传入另一个匿名函数
res9: Int = 55
scala> sum(powerOfTwo,1,5) //传入一个已经定义好的方法
res10: Int = 62
```

2.4.3 针对容器的操作

1 遍历操作

2 映射操作

3 过滤操作

4 规约操作

5 拆分操作

2.4.3 针对容器的操作

1.遍历操作

- Scala容器的标准遍历方法foreach

```
def foreach[U](f: Elem => U) :Unit
```

```
scala> val list = List(1, 2, 3)
list: List[Int] = List(1, 2, 3)
scala> val f=(i:Int)=>println(i)
f: Int => Unit = <function1>
scala> list.foreach(f)
1
2
3
```

简化写法：“list foreach(i=>println(i))”或“list foreach println”

2.4.3 针对容器的操作

1.遍历操作

```
scala> val university = Map("XMU" ->"Xiamen University", "THU" ->"Tsinghua
University","PKU"->"Peking University")
university: scala.collection.mutable.Map[String,String] = ...
scala> university foreach{kv => println(kv._1+":"+kv._2)}
XMU:Xiamen University
THU:Tsinghua University
PKU:Peking University
```

简化写法：

```
university foreach{case (k,v) => println(k+":"+v)}
```

```
university foreach{x=>x match {case (k,v) => println(k+":"+v)}}
```

2.4.3 针对容器的操作

2 映射操作

- 映射是指通过对容器中的元素进行某些运算来生成一个新的容器。两个典型的映射操作是map方法和flatMap方法
- map方法（一对一映射）：将某个函数应用到集合中的每个元素，映射得到一个新的元素，map方法会返回一个与原容器类型大小都相同的新容器，只不过元素的类型可能不同

2.4.3 针对容器的操作

2 映射操作

```
scala> val books =List("Hadoop","Hive","HDFS")
books: List[String] = List(Hadoop, Hive, HDFS)
scala> books.map(s => s.toUpperCase)
//toUpperCase方法将一个字符串中的每个字母都变成大写字母
res56: List[String] = List(HADOOP, HIVE, HDFS)
scala> books.map(s => s.length) //将字符串映射到它的长度
res57: List[Int] = List(6, 4, 4) //新列表的元素类型为Int
```

2.4.3 针对容器的操作

2 映射操作

- flatMap方法（一对多映射）：将某个函数应用到容器中的元素时，对每个元素都会返回一个容器（而不是一个元素），然后，flatMap把生成的多个容器“拍扁”成为一个容器并返回。返回的容器与原容器类型相同，但大小可能不同，其中元素的类型也可能不同

```
scala> books flatMap (s => s.toList)
res58: List[Char] = List(H, a, d, o, o, p, H, i, v, e, H, D, F, S)
```

2.4.3 针对容器的操作

3 过滤操作

- 过滤：遍历一个容器，从中获取满足指定条件的元素，返回一个新的容器

- filter方法：接受一个返回布尔值的函数f作为参数，并将f作用到每个元素上，将f返回真值的元素组成一个新容器返回

2.4.3 针对容器的操作

3 过滤操作

```
scala> val university = Map("XMU" ->"Xiamen University", "THU" -
>"Tsinghua University","PKU"->"Peking University","XMUT"->"Xiamen
University of Technology")
university: scala.collection.immutable.Map[String,String] = ...

//过滤出值中包含“Xiamen”的元素，contains为String的方法
scala> val xmus = university filter {kv => kv._2 contains "Xiamen"}
universityOfXiamen: scala.collection.immutable.Map[String,String] =
Map(XMU -> Xiamen University, XMUT -> Xiamen University of
Technology)

scala> val l=List(1,2,3,4,5,6) filter {_%2==0}
//使用了占位符语法，过滤能被2整除的元素
l: List[Int] = List(2, 4, 6)
```

2.4.3 针对容器的操作

3 过滤操作

- filterNot方法过滤出不符合条件的元素；exists方法判断是否存在满足给定条件的元素；find方法返回第一个满足条件的元素

```
scala> val t=List("Spark","Hadoop","Hbase")
t: List[String] = List(Spark, Hadoop, Hbase)
scala> t exists {_ startsWith "H"} //startsWith为String的函数
res3: Boolean = true
scala> t find {_ startsWith "Hb"}
res4: Option[String] = Some(Hbase) //find的返回值用Option类进行了包装
scala> t find {_ startsWith "Hp"}
res5: Option[String] = None
```

2.4.3 针对容器的操作

4 规约操作

- 规约操作是对容器元素进行两两运算，将其“规约”为一个值
- reduce方法：接受一个二元函数f作为参数，首先将f作用在某两个元素上并返回一个值，然后再将f作用在上一个返回值和容器的下一个元素上，再返回一个值，依此类推，最后容器中的所有值会被规约为一个值

2.4.3 针对容器的操作

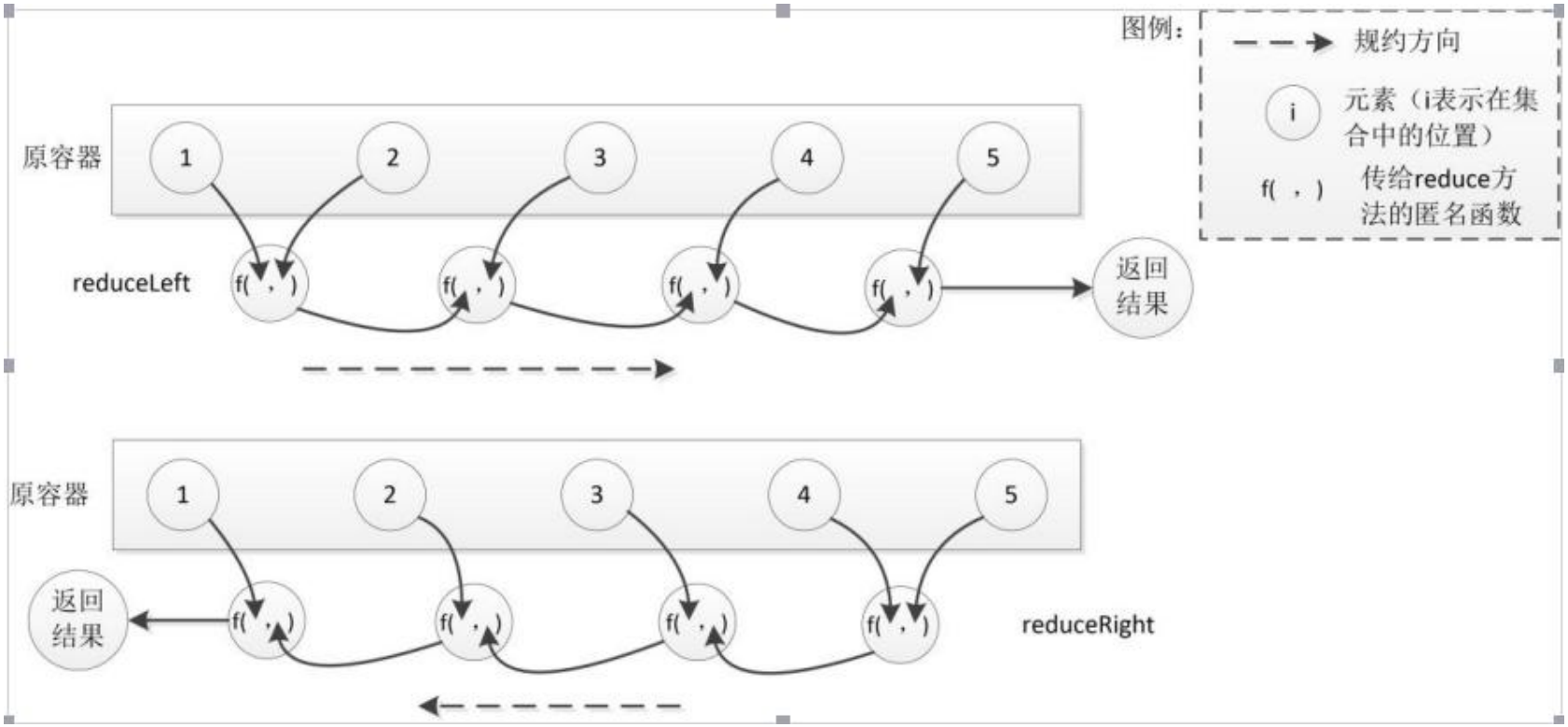
4 规约操作

```
scala> val list =List(1,2,3,4,5)
list: List[Int] = List(1, 2, 3, 4, 5)
scala>  list.reduce(_ + _) //将列表元素累加，使用了占位符语法
res16: Int = 15
scala>  list.reduce(_ * _) //将列表元素连乘
res17: Int = 120
scala> list map (_.toString) reduce ((x,y)=>s"f($x,$y)")
res5: String = f(f(f(f(1,2),3),4),5) //f表示传入reduce的二元函数
```

2.4.3 针对容器的操作

4 规约操作

- reduceLeft和reduceRight：前者从左到右进行遍历，后者从右到左进行遍历



2.4.3 针对容器的操作

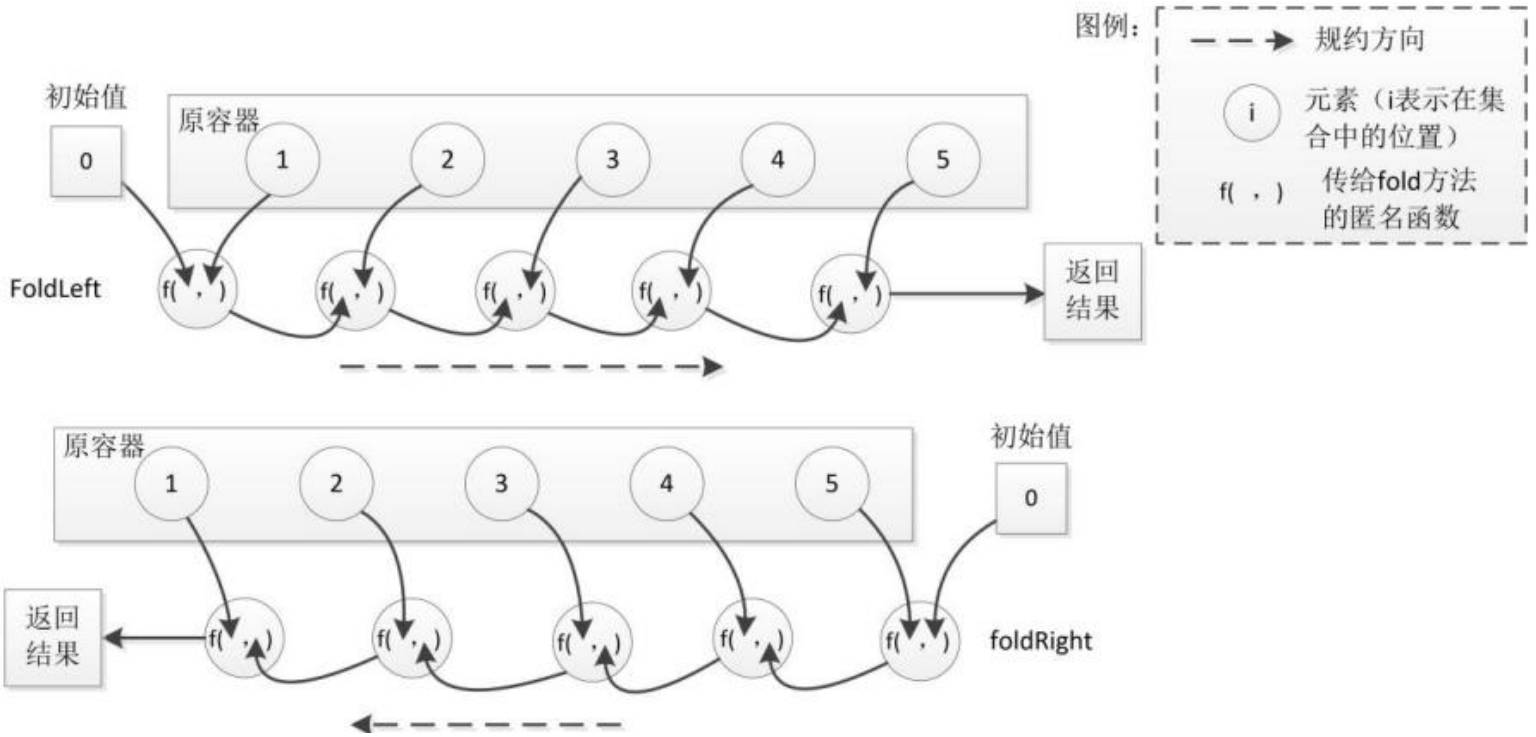
4 规约操作

```
scala> val list = List(1,2,3,4,5)
list: List[Int] = List(1, 2, 3, 4, 5)
scala> list reduceLeft {_-_}
res24: Int = -13
scala> list reduceRight {_-_}
res25: Int = 3
scala> val s = list map (_.toString)  //将整型列表转换成字符串列表
s: List[String] = List(1, 2, 3, 4, 5)
scala> s reduceLeft {(accu,x)=>s"($accu-$x)"}
res28: String = ((((1-2)-3)-4)-5)//list reduceLeft{_-_}的计算过程
scala> s reduceRight {(x,accu)=>s"($x-$accu)"}
res30: String = (1-(2-(3-(4-5))))//list reduceRight{_-_}的计算过程
```

2.4.3 针对容器的操作

4 规约操作

- fold方法：一个双参数列表的函数，从提供的初始值开始规约。第一个参数列表接受一个规约的初始值，第二个参数列表接受与reduce中一样的二元函数参数
- foldLeft和foldRight：前者从左到右进行遍历，后者从右到左进行遍历



2.4.3 针对容器的操作

4 规约操作

```
scala> val list =List(1,2,3,4,5)
list: List[Int] = List(1, 2, 3, 4, 5)
scala> list.fold(10)(_*_)
res32: Int = 1200
scala> (list fold 10)(_*_) //fold的中缀调用写法
res33: Int = 1200
scala> (list foldLeft 10)(_-_)//计算顺序(((((10-1)-2)-3)-4)-5)
res34: Int = -5
scala> (list foldRight 10)(_-_) //计算顺序(1-(2-(3-(4-(5-10)))))
res35: Int = -7
scala> val em = List.empty
em: List[Nothing] = List()
scala> em.fold(10)(_-_)//对空容器fold的结果为初始值，对空容器调用reduce
会报错
res36: Int = 10
```

2.4.3 针对容器的操作

5 拆分操作

- 拆分操作是把一个容器里的元素按一定的规则分割成多个子容器。常用的拆分方法有partition、groupedBy、grouped和sliding
- partition方法：接受一个布尔函数对容器元素进行遍历，以二元组的形式返回满足条件和不满足条件的两个集合
- groupedBy方法：接受一个返回U类型的函数对容器元素进行遍历，将返回值相同的元素作为一个子容器，并与该相同的值构成一个键值对，最后返回的是一个映射
- grouped和sliding方法：接受一个整型参数n，将容器拆分为多个与原容器类型相同的子容器，并返回由这些子容器构成的迭代器。其中，grouped按从左到右的方式将容器划分为多个大小为n的子容器（最后一个的大小可能小于n）；sliding使用一个长度为n的滑动窗口，从左到右将容器截取为多个大小为n的子容器

2.4.3 针对容器的操作

5 拆分操作

```
scala> val xs = List(1,2,3,4,5)
xs: List[Int] = List(1, 2, 3, 4, 5)
scala> val part = xs.partition(_<3)
part: (List[Int], List[Int]) = (List(1, 2),List(3, 4, 5))
scala> val gby = xs.groupBy(x=>x%3) //按被3整除的余数进行划分
gby: scala.collection.immutable.Map[Int,List[Int]] = Map(2 -> List(2, 5), 1 -> List(1, 4), 0 -> List(3))
scala> gby(2) //获取键值为2（余数为2）的子容器
res11: List[Int] = List(2, 5)
scala> val ged = xs.grouped(3) //拆分为大小为3个子容器
ged: Iterator[List[Int]] = non-empty iterator
scala> ged.next //第一个子容器
res3: List[Int] = List(1, 2, 3)
scala> ged.next //第二个子容器，里面只剩下两个元素
res5: List[Int] = List(4, 5)
scala> ged.hasNext //迭代器已经遍历完了
res6: Boolean = false
scala> val sl = xs.sliding(3)//滑动拆分为大小为3个子容器
sl: Iterator[List[Int]] = non-empty iterator
scala> sl.next //第一个子容器
res7: List[Int] = List(1, 2, 3)
scala> sl.next //第二个子容器
res8: List[Int] = List(2, 3, 4)
scala> sl.next//第三个子容器
res9: List[Int] = List(3, 4, 5)
scala> sl.hasNext //迭代器已经遍历完了
res10: Boolean = false
```

2.4.4 函数式编程实例WordCount

```
1	import java.io.File
2	import scala.io.Source
3	import collection.mutable.Map
4	object WordCount {
5		def main(args: Array[String]) {
6			val dirfile=new File("testfiles")
7			val files  = dirfile.listFiles
8			val results = Map.empty[String,Int]
9			for(file <-files) {
10				val data= Source.fromFile(file)
11				val strs =data.getLines.flatMap{s =>s.split(" ")}
12				strs foreach { word =>
13					if (results.contains(word))
14					results(word)+=1 else  results(word)=1
15					}
16				}
17			results foreach{case (k,v) => println(s"$k:$v")}
18		}
19	}
```

附录A：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。国内高校首个“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年度和2017年度厦门大学教学类奖教金获得者，荣获2017年福建省精品在线开放课程和2017年厦门大学高等教育成果二等奖。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)，主持的教改课题包括1项2016年福建省教改课题和1项2016年教育部产学协作育人项目，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500万字高价值的研究和教学资料，累计网络访问量超过500万次。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100万次。

附录B：大数据学习路线图

大数据学习路线图访问地址：http://dblab.xmu.edu.cn/post/10164/

附录C：《大数据技术原理与应用》教材

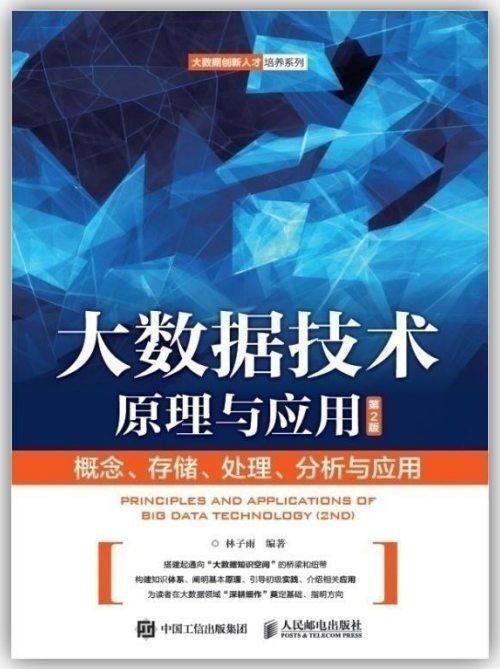
《大数据技术原理与应用——概念、存储、处理、分析与应用（第2版）》，由厦门大学计算机科学系林子雨博士编著，是国内高校第一本系统介绍大数据知识的专业教材。人民邮电出版社 ISBN:978-7-115-44330-4 定价：49.80元

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和MapReduce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》教材官方网站：
http://dblab.xmu.edu.cn/post/bigdata

扫一扫访问教材官网



附录D：《大数据基础编程、实验和案例教程》

本书是与《大数据技术原理与应用（第2版）》教材配套的唯一指定实验指导书

大数据教材

1+1黄金组合

厦门大学林子雨编著

配套实验指导书

清华大学出版社 ISBN:978-7-302-47209-4 定价：59元

- 步步引导，循序渐进，详尽的安装指南为顺利搭建大数据实验环境铺平道路
- 深入浅出，去粗取精，丰富的代码实例帮助快速掌握大数据基础编程方法
- 精心设计，巧妙融合，五套大数据实验题目促进理论与编程知识的消化和吸收
- 结合理论，联系实际，大数据课程综合实验案例精彩呈现大数据分析全流程

附录E：《Spark编程基础》

《Spark编程基础》

厦门大学 林子雨，赖永炫，陶继平 编著

披荆斩棘，在大数据丛林中开辟学习捷径
填沟削坎，为快速学习Spark技术铺平道路
深入浅出，有效降低Spark技术学习门槛
资源全面，构建全方位一站式在线服务体系

人民邮电出版社出版发行，ISBN:978-7-115-47598-5
教材官网：http://dblab.xmu.edu.cn/post/spark/

本书以Scala作为开发Spark应用程序的编程语言，系统介绍了Spark编程的基础知识。全书共8章，内容包括大数据技术概述、Scala语言基础、Spark的设计与运行原理、Spark环境搭建和使用方法、RDD编程、Spark SQL、Spark Streaming、Spark MLlib等。本书每个章节都安排了入门级的编程实践操作，以便读者更好地学习和掌握Spark编程方法。本书官网免费提供了全套的在线教学资源，包括讲义PPT、习题、源代码、软件、数据集、授课视频、上机实验指南等。

附录F：高校大数据课程公共服务平台

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

Department of Computer Science, Xiamen University, 2018