**Scala 2.8+ Handbook**

Scala的向后兼容性没有Java那么谨慎，2.8+相对于2.7有很大变化，本文档针对2.8+

**“*和别人分享你的知识，那才是永恒之道*”——somebody**

**前言：**

这只是份简单的笔记，本不应敝履自珍，但手痒难耐还是写点废话在前面。

书籍浩如烟海，技术方面的书也汗牛充栋，可惜我们的阅读速度和理解力、记忆力太有限，往往费力学懂的知识转眼就变得非常陌生，“博闻强志、过目不忘”者毕竟罕见。对于大部分人来说，即便昔日信手拈来的东西，时间久了也会毫无头绪。所以知识不在于曾经学过多少，而在于你记住并还能运用多少。

“好记性不如烂笔头”——近年来我慢慢习惯把费力学习的东西都做一个笔记，一是在学习的过程中加深印象，毕竟技术学习不同于欣赏娱乐大片和浏览娱乐新闻看个过眼烟云；二是便于学而“时习之”，书上的东西一般是针对不同技术背景的读者，有很多作者费力用墨之处对你来说纯属废话，而他一笔带过的地方恰恰让你困惑不已。一本读书笔记相当于你对书的“注解”。

Scala很好玩很有趣，但绝对不是一门简单易懂的编程语言（Java从1.0到7.0，一直做++；而Scala不光做++，也做--）。对于从Java或者其他FP走过来的人，Scala有很多“别扭”的用法，很难记清楚用正确。学Scala，最佳的做法是把它用到日常的应用开发中，不断加深记忆。但即便你准备这么做了，手头没有一份方便的备查材料，刚开始也会步履艰难。我在使用的过程中也有这个体会，所以才不怨其烦地把一些学来并尝试过的东西记在本文档中备查，以便之后能行云流水地“玩转”它。整个写笔记的过程，没有孤寂，而是沉浸在学习新知的兴奋和快意中，希望这种快乐也能放大传递给更多的人。

个人认为，对于一门编程语言使用中的查阅，大致有几个阶段：查教程（tutorial）——》查手册（handbook）——》查自己写的库。这个材料，不是严格的教程，或手册，而是介于这两者间。Scala目前已经出版了几本书，这些书从各自的角度解读Scala，比如Scala作者的“*Programming Scala*”，应该是最权威的Scala书。但好书不一定好读，一是太厚，二是无趣，不如自制的来得贴心。正如数学问题要用公式表达最清楚，编程问题得用**图表和代码**表示才最清楚，这二者也是本文中使用最多的表达方式，我尽量采用简短的代码来说明问题（简短代码也能说明很多事情，广受赞誉的Effective Java基本没有超过一页的程序代码）。能够熟练使用Java的程序员，参考本笔记，应该可以自如地开始着手写Scala程序。

若能给同样对Scala感兴趣的IT人一些帮助，本人必感欣慰。

——JamesQiu

[jamesqiu@sina.com](mailto:jamesqiu@sina.com)

[http://qiujj.com](http://qiujj.com/)

**目录**

[**Scala 2.8+ Handbook**](http://qiujj.com/static/Scala-Handbook.htm#_Toc306132177)

[1.           Scala有多cool](http://qiujj.com/static/Scala-Handbook.htm#_Toc306132178)

[1.1.        速度！](http://qiujj.com/static/Scala-Handbook.htm#_Toc306132179)

[1.2.        易用的数据结构](http://qiujj.com/static/Scala-Handbook.htm#_Toc306132180)

[1.3.        OOP+FP](http://qiujj.com/static/Scala-Handbook.htm#_Toc306132181)

[1.4.        动态+静态](http://qiujj.com/static/Scala-Handbook.htm#_Toc306132182)

[1.5.        DSL](http://qiujj.com/static/Scala-Handbook.htm#_Toc306132183)

[1.6.        够复杂](http://qiujj.com/static/Scala-Handbook.htm#_Toc306132184)

[1.7.        够有趣](http://qiujj.com/static/Scala-Handbook.htm#_Toc306132185)

[1.8.        开发社区](http://qiujj.com/static/Scala-Handbook.htm#_Toc306132186)

[2.           lang](http://qiujj.com/static/Scala-Handbook.htm#_Toc306132187)

[2.1.        和Java的异同](http://qiujj.com/static/Scala-Handbook.htm#_Toc306132188)

[2.2.        变量](http://qiujj.com/static/Scala-Handbook.htm#_Toc306132189)

[2.3.        基本类型](http://qiujj.com/static/Scala-Handbook.htm#_Toc306132190)

[2.4.        BigInt](http://qiujj.com/static/Scala-Handbook.htm#_Toc306132191)

[2.5.        字符串](http://qiujj.com/static/Scala-Handbook.htm#_Toc306132192)

[2.6.        Null, None, Nil, Nothing](http://qiujj.com/static/Scala-Handbook.htm#_Toc306132193)

[2.7.        ==和eq](http://qiujj.com/static/Scala-Handbook.htm#_Toc306132194)

[2.8.        Option[T]](http://qiujj.com/static/Scala-Handbook.htm#_Toc306132195)

[2.9.        区分<-,=>,->](http://qiujj.com/static/Scala-Handbook.htm#_Toc306132196)

[2.10.      match..case(switch)](http://qiujj.com/static/Scala-Handbook.htm#_Toc306132197)

[2.11.      try..catch..finally](http://qiujj.com/static/Scala-Handbook.htm#_Toc306132198)

[2.12.      require](http://qiujj.com/static/Scala-Handbook.htm#_Toc306132199)

[2.13.      main方法](http://qiujj.com/static/Scala-Handbook.htm#_Toc306132200)

[2.14.      package, import](http://qiujj.com/static/Scala-Handbook.htm#_Toc306132201)

[2.15.      if..else](http://qiujj.com/static/Scala-Handbook.htm#_Toc306132202)

[2.16.      循环操作](http://qiujj.com/static/Scala-Handbook.htm#_Toc306132203)

[2.17.      操作符重载](http://qiujj.com/static/Scala-Handbook.htm#_Toc306132204)

[2.18.      系统定义scala.\_](http://qiujj.com/static/Scala-Handbook.htm#_Toc306132205)

[2.19.      implicit隐式转换](http://qiujj.com/static/Scala-Handbook.htm#_Toc306132206)

[2.20.      type做alias](http://qiujj.com/static/Scala-Handbook.htm#_Toc306132207)

[2.21.      泛型](http://qiujj.com/static/Scala-Handbook.htm#_Toc306132208)

[2.22.      枚举Enum](http://qiujj.com/static/Scala-Handbook.htm#_Toc306132209)

[3.           FP](http://qiujj.com/static/Scala-Handbook.htm#_Toc306132210)

[3.1.        函数](http://qiujj.com/static/Scala-Handbook.htm#_Toc306132211)

[3.2.        函数式编程](http://qiujj.com/static/Scala-Handbook.htm#_Toc306132212)

[4.           OOP](http://qiujj.com/static/Scala-Handbook.htm#_Toc306132213)

[4.1.        类class](http://qiujj.com/static/Scala-Handbook.htm#_Toc306132214)

[4.2.        trait超级接口](http://qiujj.com/static/Scala-Handbook.htm#_Toc306132215)

[4.3.        协变和逆变(co-|contra-)variance](http://qiujj.com/static/Scala-Handbook.htm#_Toc306132216)

[5.           util包](http://qiujj.com/static/Scala-Handbook.htm#_Toc306132217)

[5.1.        架构](http://qiujj.com/static/Scala-Handbook.htm#_Toc306132218)

[5.2.        集合Array,List,Tuple](http://qiujj.com/static/Scala-Handbook.htm#_Toc306132219)

[5.3.        Map](http://qiujj.com/static/Scala-Handbook.htm#_Toc306132220)

[5.4.        Set](http://qiujj.com/static/Scala-Handbook.htm#_Toc306132221)

[5.5.        Iterator](http://qiujj.com/static/Scala-Handbook.htm#_Toc306132222)

[5.6.        Paralllel collection](http://qiujj.com/static/Scala-Handbook.htm#_Toc306132223)

[6.           io](http://qiujj.com/static/Scala-Handbook.htm#_Toc306132224)

[6.1.        文件I/O](http://qiujj.com/static/Scala-Handbook.htm#_Toc306132225)

[6.2.        网络I/O](http://qiujj.com/static/Scala-Handbook.htm#_Toc306132226)

[7.           actor](http://qiujj.com/static/Scala-Handbook.htm#_Toc306132227)

[7.1.        actor模型](http://qiujj.com/static/Scala-Handbook.htm#_Toc306132228)

[7.2.        多核计算](http://qiujj.com/static/Scala-Handbook.htm#_Toc306132229)

[7.3.        Actor用法](http://qiujj.com/static/Scala-Handbook.htm#_Toc306132230)

[7.4.        方式1：接受receive](http://qiujj.com/static/Scala-Handbook.htm#_Toc306132231)

[7.5.        方式2：接受react, loop](http://qiujj.com/static/Scala-Handbook.htm#_Toc306132232)

[7.6.        REPL接受消息](http://qiujj.com/static/Scala-Handbook.htm#_Toc306132233)

[7.7.        actor最佳实践](http://qiujj.com/static/Scala-Handbook.htm#_Toc306132234)

[7.8.        不同jvm间的消息访问](http://qiujj.com/static/Scala-Handbook.htm#_Toc306132235)

[7.9.        STM](http://qiujj.com/static/Scala-Handbook.htm#_Toc306132236)

[8.           misc](http://qiujj.com/static/Scala-Handbook.htm#_Toc306132237)

[8.1.        xml](http://qiujj.com/static/Scala-Handbook.htm#_Toc306132238)

[8.2.        json](http://qiujj.com/static/Scala-Handbook.htm#_Toc306132239)

[8.3.        Configgy](http://qiujj.com/static/Scala-Handbook.htm#_Toc306132240)

[8.4.        正则表达式regex](http://qiujj.com/static/Scala-Handbook.htm#_Toc306132241)

[8.5.        GUI](http://qiujj.com/static/Scala-Handbook.htm#_Toc306132242)

[9.           附录](http://qiujj.com/static/Scala-Handbook.htm#_Toc306132243)

[9.1.        stackoverflow的scala教程](http://qiujj.com/static/Scala-Handbook.htm#_Toc306132244)

[9.2.        ProjectEuler](http://qiujj.com/static/Scala-Handbook.htm#_Toc306132245)

[9.3.        Scala问答](http://qiujj.com/static/Scala-Handbook.htm#_Toc306132246)

[9.4.        rosettacode](http://qiujj.com/static/Scala-Handbook.htm#_Toc306132247)

[9.5.        编译、mvn、SBT](http://qiujj.com/static/Scala-Handbook.htm#_Toc306132248)

[9.6.        Scala水平划分](http://qiujj.com/static/Scala-Handbook.htm#_Toc306132249)

[9.7.        Scala适合的领域](http://qiujj.com/static/Scala-Handbook.htm#_Toc306132250)

[9.8.        Twitter的Scala School](http://qiujj.com/static/Scala-Handbook.htm#_Toc306132251)

1.   **Scala有多cool**

“Put productivity & creativity back in the hands of developers”

Scala is a tour de force of language design.

**1.1.**     **速度！**

**Java的运行速度**

——基于JVM，和Java运行速度相当。看看Ruby、Perl、Python对大项目运行效率的无奈，就知道有个好的编译器（Scalac）和运行时（JVM）是多么美好。

**Python/Ruby的编程速度**

——有更多的内建库和数据结构，编程就更快，Scala在完全继承Java和.NET的标准库的基础上，还扩展了更丰富有用的函数库。看看C++、D、Go等语言库的发展情况（不是匮乏就是混乱），就知道从头创建如Java、.NET这般庞大全面的类库并非易事；

类库和运行速度有关系吗？——很大程度上有，众多专家已经在类库中准备了充分优化的稳定算法，Scala对Java Collection算法进行直接包装或者直接调用，如果没有丰富的类库，你在项目周期内免不了摘抄一些不一定靠谱的算法和功能代码，这些代码极有可能在运行时给你带来麻烦。使用类库算法，不用担忧自造轮子的运行效率。

Scala是静态语言，Scalac和Javac是同一作者，编译成.class后运行于JVM平台，近20年那么多大公司投入进行的优化也不是白搭。对于大部分的应用来说，使用Scala不用再顾虑运行速度，它可能不是最快，但至少逼近Java，而不像Groovy、JRuby、Jython那般与Java有高达数十倍的效率差距。

**1.2.**     **易用的数据结构**

List-Map-Tuple及其丰富特性支持让你解决数据结构问题时游刃有余。

List(1,31,4,3,53,4,234) filter (10<) filter (100>)  // List(31, 53)

val (a,b) = List(1, 31,4,3,53,4,234) partition (10>) // a=List(1,4,3,4), b=List(31,53,234)

def info(p:Person) = (name, age, email) // info.\_1, info.\_2, info.\_3

**1.3.**     **OOP+FP**

  适当地选用OOP或者FP，能够使表达相对另一种更加清晰准确。

  实际可见的生产力在于：一个应用中的部分代码尤其是高知识凝聚的代码如数学函数和设计模式，一般来说不会自己编写，而是会来自于现成的Java库，或者其他语言，或者伪代码。我们可以很容易地把过程语言、面向对象语言、函数式语言中的代码“翻译”成Scala代码。试想如果我们要把Haskell或者Lisp的某个尾递归算法翻译成Java代码，还得多花点时间；而要把C++的代码翻译成Hashkell，同样也不简单。Scala的混血性给我们的实际使用提供了便利。

  语言特色能够塑造编程者的思维： C++也能使用抽象基类设计多重继承，但Java的接口引导你走得更远；Java也能设计类型安全的静态方法(final static)，但Scala鼓励你这样做并逐步从OOP到达FP的彼岸，而且来去自如。

**1.4.**     **动态+静态**

Scala虽然是一门彻头彻底的静态语言，但又具备了现代动态语言的很多方便和灵活：

  不需要冗余的类型声明

  可以在已有类上增加新方法（implicit转换和Dynamic trait）

  可以把不继承共同父类的不同类型对象传到方法中

  可以做静态语言的Refactoring

  不用象动态语言那样测试代码比业务代码还多

  代码自动完成（REPL和IDE）

  编译静态语言的性能

  Read-Eval-Print Loop交互解释器（注：Linux下的用户体验远好于Windows下）

**1.5.**     **DSL**

Scala可以把xml/html处理、数学公式表达、SQL查询等包装的更优雅、更合理，为使用者提供更好的API。这也使Scala的程序也更可读，从而更易于维护。

**1.6.**     **够复杂**

不同的思考模式：Java是先写后想，Scala是先想后写（其实FP大都如此）。

Scala相比于Java，可能达不到10倍的代码精简；但读Scala代码的效率一般只有Java的1/10——可见Java是一门没有多少特例的简单语言，而Scala则不然。

你不要指望把Scala作为初学者的第一门编程语言，这门语言甚至不是初级程序员能够掌控的——换句话说，能够读懂和写Scala代码，说明你是一个不折不扣的资深程序员，或者更准确一点，是资深Java程序员。

|  |  |
| --- | --- |
| http://qiujj.com/static/Scala-Handbook.files/image001.jpg | http://qiujj.com/static/Scala-Handbook.files/image002.jpg |
| http://qiujj.com/static/Scala-Handbook.files/image003.jpg | http://qiujj.com/static/Scala-Handbook.files/image004.jpg    http://qiujj.com/static/Scala-Handbook.files/image005.jpg http://qiujj.com/static/Scala-Handbook.files/image006.jpg |
| 简单的Java（Morse码，定型的玩具车） | 复杂的Scala（智能机，乐高积木） |

**1.7.**     **够有趣**

还看这句话：“Put **productivity & creativity** back in the hands of developers”。其实不仅限于Scala，对于所有的编程语言来说，一门语言是否“好玩”有趣，能否激起创作欲，才是最关键的，这比语言风格、运行速度、工具支持、社区文化都来得重要。

回想我使用过的语言，C、C++，只有在学习“图形学”课程做作业的时候给我“好玩”和编程过瘾的感觉；VB、Delphi、Lotus Script/Formular、JavaScript，重来没有给过我“好玩”的感觉，而Java是在之前很长一段时间内让我觉得最“好玩”的语言，用它编游戏、做模式识别的作业、做产品……，乐在其中。但是Java也许久没有再给我这种编程过瘾的感觉了。之前发现Groovy的时候，我以为又找到一门好玩的语言了，但我一段时间使用之后，发现不是我的菜（Perl、Python、Ruby也如此）；我不是说那些我不觉得好玩的语言不好，有其他很多人觉得他们非常“好玩”，并且用它们创建了无数杀手级的、伟大的、有用的程序。

有些人对一门语言会玩一辈子，就像Lisp、Haskell和Smalltalk的拥趸；而有些人会不断寻找下一个玩意儿，就像原来玩Java的一些人发现更好玩的Ruby和Python之后，倒戈狂喷Java，力挺后者；Groovy/Grails的玩家在很短的时间里面，写了无数的扩展和Plugin应用；学习Scala，能很多好玩的地方，能用它有激情地去写一些振奋人心的应用出来！

**1.8.**     **开发社区**

Scala开发/用户社区气氛良好，基本都是资深开发者以及有一定经验的用户，不会碰到太弱智的事（提问、争吵），除了语言和工具开源免费，最权威和最好的书也都是免费的（包括Lift社区）

2.   **lang**

**2.1.**     **和Java的异同**

**2.1.1.**  **语法**

|  |  |
| --- | --- |
| **Java++：增加的语法** | **Java--：删减的语法** |
| 纯OO | 静态成员 |
| 操作符重载 | 原生数据类型 |
| closure | break、continue |
| 使用trait进行mixin组合 | 接口 |
| existential type（\_） | 通配符List<?>, import pkg.\*; |
| 抽象类型 （type T） | 原始类型 class C1<T> {...} |
| 模式匹配 | enum枚举 |

注：

existential type——和Java互操作时进行对应

Iterator<? extends Component>  -->  Iterator[T] { type T <: Component }或者Iterator[\_]

**2.1.2.**  **库**

以下功能通过库的形式提供：

  assert

  enum

  property

  event

  actor

  resource control（自动释放）

def using[T <: { def close() }] (res:T)(block:T=>Unit) = {

  try { block(res) } finally { if(res!=null) res.close }}

using (new BufferedReader(new FileReader(path))) { f=> println(f.readLine) }

不用每次使用Java中的finally

val f = new BufferedReader(new FileReader(path)

try { println(f.readLine) } finally { if (f!=null) f.close }

  query

**2.2.**     **变量**

**2.2.1.**  **保留字**

abstract   case       catch      class      def

do         else       extends    false      final

finally    for        if         implicit   import

match      new        null       object        override

package    private    protected requires   return

sealed        super      this       throw      trait

try        true       type       val        var

while      with       yield

\_      :      =      =>     <-     <:     <%     >:     #      @

Scala调用Java的方法时，会碰到有Scala的保留字，如Thread.yield()

这在Scala中是非法的，专门有个解决办法，写成： Thread.`yield`()

注意：**没有**break和continue

**2.2.2.**  **变量标识**

这些标识在Java中是非法的，在Scala中是合法的，可以当作函数名使用，使接口更加DSL：

val empty\_? = true

val + = "hello"

val `yield` = 10

val \*\* = "power"

**2.2.3.**  **变量定义**

**2.2.3.1**     **val, var**

var 可变，可重新赋值，赋值为"\_"表示缺省值(0, false, null)，例如：

var d:Double = \_ // d = 0.0

var i:Int = \_ // i = 0

var s:String = \_ // s = null

var t:T = \_  // 泛型T对应的默认值

val 不可变，相当于const/final，但如果val为数组或者List，val的元素可以赋值；

val pi = 3. // 相当于3.0d

val pi = 3.f // 相当于3.0f

提示：向函数式风格推进的一个方式，就是尝试**不用**任何var来定义变量。

**2.2.3.2**     **花样定义**

和Python一样方便的赋值方式：

val x,y = 0 // 赋同一初始值

val (x,y) = (10, "hello") // 同时定义多个变量，注意：val x,y=10,"hello" 是**错误**的

更花：

val x::y = List(1,2,3,4)  // x = 1, y = List(2,3,4)

val List(a,b,c) = List(1,2,3) // a = 1, b = 2, c = 3

进一步花样：

val Array(a, b, \_, \_, c @ \_\*) = Array(1, 2, 3, 4, 5, 6, 7)  // 也可以用List，Seq

a // 1

b // 2

c // Array(5, 6, 7), \_\*匹配0个到多个

使用正则表达式定义：

val regex = "(\\d+)/(\\d+)/(\\d+)".r

val regex(year, month, day) = "2010/1/13"

// year: String = 2010

// month: String = 1

// day: String = 13

**2.2.3.3**     **lazy, val, def的区别**

|  |  |  |
| --- | --- | --- |
| val | 定义时就一次求值完成，保持不变 | val f = 10+20 // 30 |
| lazy | 定义时不求值，第一次使用时完成求值，保持不变 | lazy f = 10+20 // <lazy>  f // 30 |
| def | 定义时不求值，每次使用时都重新求值 | def f = 10+20 // 30  def t = System. currentTimeMillis // 每次不一样 |

scala> val f1 = System.currentTimeMillis

f1: Long = 1279682740376    // 马上求值

scala> f1

res94: Long = 1279682740376 // 之后保持不变

scala> lazy val f2 = System.currentTimeMillis

f2: Long = <lazy>  // 定义时不求值

scala> System.currentTimeMillis

res95: Long = 1279682764297

scala> f2

res96: Long = 1279682766545 // 第一次使用时求值，注意：6545 > 4297

scala> f2

res97: Long = 1279682766545 // 之后保持不变

scala> def f3 = System.currentTimeMillis

f3: Long

scala> f3

res98: Long = 1279682784478 // 每次求值

scala> f3

res99: Long = 1279682785352 // 每次求值

**2.3.**     **基本类型**

尽量使用大写形式： Int, Long, Double, Byte, Short, Char, Float, Double, Boolean

编译时Scala自动对应到Java原始类型，提高运行效率。Unit对应java的void

用 asInstanseOf[T]方法来强制转换类型：

def i = 10.asInstanceOf[Double] // i: Double = 10.0

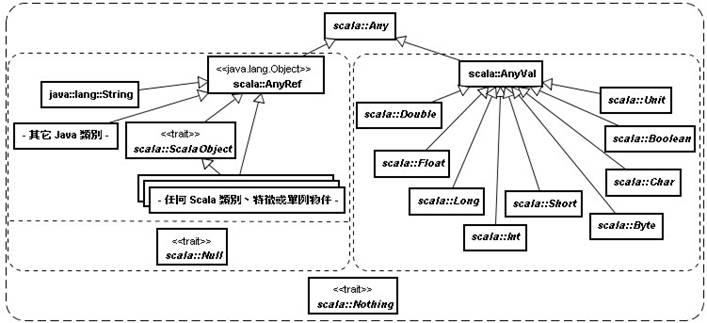
List('A','B','C').map(c=>(c+32).asInstanceOf[Char]) // List('a','b','c')

用isInstanceOf[T]方法来判断类型：

val b = 10.isInstanceOf[Int] // true

而在match ... case 中可以直接判断而不用此方法。

用Any统一了原生类型和引用类型。



**2.3.1.**  **Int**

-3 abs // 3

-3 max -2 // -2

-3 min -2 // -3

1.4 round // 1 四舍五入

1.6 round // 2 四舍五入

1.1 ceil // 2.0 天花板

1.1 floor // 1.0 地板

无++，--操作，但可以+=, -=, 如下：

var i = 0

i++  // 报错，无此操作

i+=1 // 1

i--  // 报错，无此操作

i-=1 // 0

def even(n:Int) = 0==(n & 1)

def odd(n:Int) = !even(n)

**2.3.2.**  **Char**

String可以转化为List[Char]

在String上做循环，其实就是对String中的每一个Char做操作，如：

"12345" map (toInt) // (49,50,51,52,53)

"jamesqiu" max // 'u'

"jamesqiu" min // 'a'

('a' to 'f') map (\_.toString\*3) // (aaa, bbb, ccc, ddd, eee, fff)

**2.4.**     **BigInt**

可以表示很大的整数：

BigInt(10000000000000000000000000) // 报错

BigInt("10000000000000000000000000") // scala.math.BigInt = 10000000000000000000000000

例如：

def  fac(n:Int):BigInt = if (n==0) 1 else fac(n-1)\*n

fac(1000)

或者写成：

def fac2(n:Int) = ((1:BigInt) to n).product

// res1: BigInt = 9332621544394415268169923885626670049071596826438......000000000000000000

**2.5.**     **字符串**

"..." 或者 """...""""

println("""|Welcome to Ultamix 3000.

           |Type "HELP" for help.""".stripMargin)

输出：

Welcome to Ultamix 3000.

Type "HELP" for help.

scala中，字符串除了可以+，也可以\*

"abc" \* 3 // "abcabcabc"

"abc" \* 0 // ""

例子：

"google".reverse // "elgoog"

"abc".reverse.reverse=="abc" // true

例子：

"Hello" map (\_.toUpper) // 相当于 "Hello".toUpperCase

**2.5.1.**  **类型转换**

"101".toInt // 101，无需 Integer.parseInt("101");

"3.14".toFloat // 3.14f

101.toString

3.14.toString

转换整个列表：

List("1","2","3") map (\_.toInt) // List(1,2,3)

或者

List("1","2","3") map Integer.parseInt // List(1,2,3)

**2.5.2.**  **StringBuilder**

val sb = new StringBuilder

sb += 'H'

sb ++= "ello"

sb.toString // "Hello"

sb clear // StringBuilder()

**2.5.3.**  **文本格式化**

使用java.text.MessageFormat.format:

**val** msg = java.text.MessageFormat.format(

    "At {1,time} on {1,date}, there was {2} on planet **{0}**.",

    "**Hoth**", **new** java.util.Date(), "a disturbance in the Force")

输出

At 17:50:34 on 2010-7-20, there was a disturbance in the Force on planet **Hoth**.

方法2：

"my name is %s, age is %d." format ("james", 30) // my name is james, age is 30.

注意：format还可以这么用

"%s-%d：%1$s is %2$d." format ("james", 30) // james-30：james is 30.

    "%2$d age's man %1$s: %2$d" format ("james", 30) // 30 age's man james: 30

**2.6.**     **Null, None, Nil, Nothing**

|  |  |
| --- | --- |
| Null | Trait，其唯一实例为null，是AnyRef的子类，\***不是**\* AnyVal的子类 |
| Nothing | Trait，所有类型（包括AnyRef和AnyVal）的子类，没有实例 |
| None | Option的两个子类之一，另一个是Some，用于安全的函数返回值 |
| Unit | 无返回值的函数的类型，和java的void对应 |
| Nil | 长度为0的List |

**2.7.**     **==和eq**

Scala的==很智能，他知道对于数值类型要调用Java中的==，ref类型要调用Java的equals()

"hello"=="Hello".toLowerCase()

在java中为false，在scala中为true

Scala的==总是内容对比

|  |  |
| --- | --- |
| 基本类型Int，Double， | 比值 |
| 其他类型 | 相当于A.equals(B) |

eq才是引用对比

例如：

val s1,s2 = "hello"

val s3 = new String("hello")

s1==s2 // true

s1 eq s2 // true

s1==s3 // true 值相同

s1 eq s3 // false 不是同一个引用

**2.8.**     **Option[T]**

**2.8.1.**  **概念**

  Option[T]可以是任意类型或者空，但一旦声明类型就不能改变；

  Option[T]可完美替代Java中的null，可以是Some[T]或者None；

  Option实现了map, flatMap, and filter 接口，允许在 'for'循环里使用它；

**函数返回值**能被统一处理了：

|  |  |
| --- | --- |
| **没有Option的日子** | **现在** |
| def find(id:Long):Person = ... | def find(id:Long):Option[Person] = ... |
| 返回Person或者null | 返回Some[Person]或者None |
| 返回null不特殊处理会抛：NullPointerExceptions | 返回值直接getOrElse或者列表操作 |
| 类比：Java的Stringx.split返回null | 类比：Java的Stringx.split返回new String[0] |
| **结论：函数永远不要返回null值，如果输入有问题或者抛异常，返回Option[T]** | |

**参数有效性检查**没有那么烦人了：

|  |  |
| --- | --- |
| **没有Option的日子** | **现在** |
| def blank(s:String) = if (s==null) false else {    s.toList.forall(\_.isWhitespace) } | def blank(s:String) =Option(s).toList.forall(  \_.forall(\_.isWhitespace)) |
| **结论：尽可能地不要浪费代码去检测输入，包装成Option[T]来统一处理** | |

**2.8.2.**  **使用**

Some(3).getOrElse(4) // 3

None.getOrElse(4) // 4

例如打印key=3的value：

写法1：

def p(map:Map[Int,Int]) = println(map(3))

p(Map(1->100,2->200)) // 抛异常

写法2：

def p(map:Map[Int,Int]) = println(map get 3 getOrElse "...")

p(Map(1->100,2->200)) // ...

p(Map(1->100,3->300)) // 300

**2.8.3.**  **例子**

例子1：

  def m(k:Int) = {

    Map((1,100),(2,200),(3,300)) get(k) match {

      case Some(v) =>

        k + ": " + v

      case None =>

"not found"

    }

  }

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

    println(m(1)) // 100

    println(m(2)) // 200

    println(m(3)) // 300

    println(m(4)) // "not found"

    println(m(-1)) // "not found"

  }

例子2：

val l = List(Some(100), None, Some(200), Some(120), None)

for (Some(s) <- l) yield s // List(100, 200, 120)

或

l flatMap (x=>x) // List(100, 200, 120)

例子3： Option结合flatMap

def toint(s:String) =

try { Some(Integer.parseInt(s)) } catch { case e:Exception => None }

List("123", "12a", "45") flatMap toint // List(123, 45)

List("123", "12a", "45") map toint // List(Some(123), None, Some(45))

**2.9.**     **区分<-,=>,->**

|  |  |  |
| --- | --- | --- |
| <- | for (i <- 0 until 100) | 用于for循环, 符号∈的象形 |
| => | List(1,2,3).map(x=> x\*x)  ((i:Int)=>i\*i)(5) // 25 | 用于匿名函数，相当于Ruby的 |x|,Groovy的{x-> x\*x}  也可用在import中定义别名：import javax.swing.{JFrame=>jf} |
| -> | Map(1->"a",2->"b") | 用于Map初始化, 也可以不用->而写成 Map((1,"a"),(2,"b")) |

**2.10.**    **match..case(switch)**

**2.10.1.**      **和switch..case的区别**

Java里面的写法：

switch(n) {

    case(1): ...; break;

    case(2): ...; break;

    default: ...;

}

变成Scala写法：

def m(n:String) =

n match {

    case "a" | "b" => ... // 这个比较好

    case "c" => ...

    case \_ => ...

}

每个case..=>结束不用写break了，\_相当于default

**2.10.2.**      **匹配数据类型**

match 可以很简单地匹配数据类型（不需要isInstanceOf[T]）：

def f(v:Any) = v match {

    case null => "null"

    case i:Int => i\*100

    case s:String => s

    case \_ => "others"

}

注意：上面case中的i、s都叫模式变量

f(null) // "null"

f(5) // 500

f("hello") // "hello"

f(3.14) // "others"

注意：自定义类型如果也要匹配，需要用[case class](http://qiujj.com/static/Scala-Handbook.htm#_case_class)

**2.10.3.**      **命令行参数解析例子**

/\*\* Basic command line parsing. \*/

**object** Main {

**var** verbose = **false** // 记录标识，以便能同时对-h和-v做出响应

**def** main(args: Array[String]) {

**for** (a <- args) a **match** {

**case** "-h" | "-help"    **=>**

        println("Usage: scala Main [-help|-verbose]")

**case** "-v" | "-verbose" **=>**

        verbose = **true**

**case** x **=>** // 这里x是临时变量

        println("Unknown option: '" + x + "'")

    }

**if** (verbose) println("How are you today?")

  }

}

**2.10.4.**      **使用case的递归函数**

写法1：

def fac(n:Int):Int = n match {

  case 0=>1

  case \_=>n\*fac(n-1)

}

写法2（使用[映射式函数](http://qiujj.com/static/Scala-Handbook.htm#_映射式定义)）：

def fac: Int=>Int = {

case 0=> 1

case n=> n\*fac(n-1)

}

写法3（使用尾递归）：

  def fac: (Int,Int)=>Int = {

    case (0,y) => y

    case (x,y) => fac(x-1, x\*y)

  }

fac(5,1) // 120

写法4（reduceLeft+!）：

def fac(n:Int) = 1 to n reduceLeft(\_\*\_)

implicit def foo(n:Int) = new { def ! = fac(n) }

5! // 120

写法5：(最简洁高效)

  def fac(n:Int) = (1:BigInt) to n product

  fac(5) // 120

**2.10.5.**      **变量匹配**

常量匹配很简单，即case后跟的都是常量；

变量匹配需要注意，case后跟的是match里面的临时变量，而不是其他变量名：

3 match {

case i => println("i=" + i) // 这里i是模式变量（临时变量），就是3

}

val a = 10

20 match { case a => 1 } // 1， a是模式变量，不是10

为了使用变量a，必须用`a`:

        20 match { case `a` => 1; case b => -1 } // -1，`a`是变量10

或者用大写的变量：

        val A = 10

        20 match { case A => 1; case b => -1 } // -1，大写A是变量10

**2.10.6.**      **case..if条件匹配**

写法1：

(1 to 20) foreach {                         

[case](http://scala-lang.org/) x [if](http://scala-lang.org/) (x % 15 == 0) => printf("%2d:15n\n",x)

[case](http://scala-lang.org/) x [if](http://scala-lang.org/) (x % 3 == 0)  => printf("%2d:3n\n",x)

[case](http://scala-lang.org/) x [if](http://scala-lang.org/) (x % 5 == 0)  => printf("%2d:5n\n",x)

[case](http://scala-lang.org/) x => printf("%2d\n",x)                         

}

写法2：

(1 to 20) map (x=> (x%3,x%5) match {

  case (0,0) => printf("%2d:15n\n",x)

  case (0,\_) => printf("%2d:3n\n",x)

  case (\_,0) => printf("%2d:5n\n",x)

  case (\_,\_) => printf("%2d\n",x)

})

**2.11.**    **try..catch..finally**

var f = openFile()

try {

  f = new FileReader("input.txt")

} catch {

  case ex: FileNotFoundException => // Handle missing file

  case ex: IOException => // Handle other I/O error

} finally {

  f.close()

}

**2.12.**    **require**

def f(n:Int) = { require(n!=0); 1.0/n }

def f(n:Int) = { require(n!=0, "*n can't be zero*"); 1.0/n }

f(0)

// java.lang.IllegalArgumentException: requirement failed: *n can't be zero*

**2.13.**    **main方法**

Scala的main方法(包括所有类似java的static方法)必须定义在一个object内：

object Test1 {

    def main(args: Array[String]) {

       println("hello world")

    }

}

编译：

fsc Test1.scala // 常驻内存编译服务器，第一次转载之后比scalac快

运行：

scala Test1.scala // 方式1，没有输出

scala -cp e:\scala\lib\scala-library.jar Test1 // 方式2，慢

java -cp e:\scala\lib\scala-library.jar Test1 // 方式3，快

如果文件不是utf8编码，执行需要使用.scala文件的编码，如：

scala -encoding gbk test.scala

**2.13.1.**      **Application**

不带命令行参数的简化main方法：

object app1 extends Application {

    println("hello world")

}

**2.14.**    **package, import**

**2.14.1.**      **import**

Scala的import可以只在局部作用域内生效；

可以格式 “import javax.swing.{JFrame=>jf}”来声明类型的别名。

jf.show()

  import javax.swing.\_

  import java.util.{List, Map}

  import java.util.\_, java.io.\_

Scala 缺省导入如下包：

  java.lang.\*

  scala.\*

  scala.Predef

由于Scala的package可以是相对路径下定义，有可能命名冲突，可以用：

import \_root\_.java.lang.Long

**2.14.2.**      **package**

package com.wr3 { // C# 和Ruby的方式，也可以改用Java的方式

    // import java.nio.\_ // "\*" 是scala的正常函数名，所以用\_

    class c1 {

       def m1() { println("c1.m1()") }

    }

    object o1 {

       def main(args: Array[String]) {

          println("o1.main()")

          new c1().m1()

        }

    }

}

编译：

fsc package.scala

运行：

java com.wr3.o1 // 方式1

scala com.wr3.o1 // 方式2

**2.14.3.**      **包对象**

Scala2.8+支持包对象（package object），除了和2.8之前一样可以有下级的object和class，还可以直接有下级变量和函数，例如：

-------------------------------- foo.scala

package p0

package object p1 {

    val a = 10

    def b = "hello " + a

    def main(args:Array[String]):Unit = printf("%s", p0.p1.b)

}

--------------------------------

p1就是一个包对象，a和b就是包p1直属的常量和函数,

$fsc foo.scala 命令产生如下class：

./p0/p1/package.class

调用：

scala p0.p1.package

**2.15.**    **if..else**

没有java的：

b = (x>y) ? 100 : -1

就用：

if (x>y) 100 else -1

**2.16.**    **循环操作**

|  |  |
| --- | --- |
| map | m->m |
| flatMap | m->n |
| indices | m->m |
| foreach | m->Unit |
| for (... if ...) yield | m->n |
| collect { case ... if ... => ... } | m->n |
| filter, filterNot | m->n |
| take | m->n |
| takeWhile | m->n |
| forall | m->1 (true|false) |
| reduceLeft, foldLeft | m->1 |
| scanLeft | m->m+1 |
| exists | m->1 (true|false) |
| find | m->1 (或者None) |
| count | m->1 |
| span, partition | m->2 |

**2.16.1.**      **for**

循环中的变量不用定义，如：

for(i<-1 to 10; j=i\*i) println(j)

for (s <- ss) foo(s)

for (i <- 0 to n) foo(i) // 包含n，即Range(0,1,2,...,n,n+1)

for (i <- 0 until n) foo(i)  // 不包含n，即Range(0,1,2,3,...,n)

例如：

for (n<-List(1,2,3,4) if n%2==1) yield n\*n  // List(1, 9)

for (n<-Array(1,2,3,4) if n%2==1) yield n\*n  // Array(1, 9)

注意：如果if后面不止一条语句，要用{..}包裹。

var s = 0; for (i <- 0 until 100) { s += i } // s = 4950

等价于不用for的写法：

List(1,2,3,4).filter(\_%2==1).map(n => n\*n)

如果for条件是多行，不能用(),要用{}

for(i<-0 to 5; j<-0 to 2) yield i+j

// Vector(0, 1, 2, 1, 2, 3, 2, 3, 4, 3, 4, 5, 4, 5 , 6, 5, 6, 7)

for{i<-0 to 5

j<-0 to 2} yield i+j

例子1：

// 边长21以内所有符合勾股弦的三角形：

[def](http://scala-lang.org/) triangle(n: Int) = [for](http://scala-lang.org/) {

  x <- 1 to 21

  y <- x to 21

  z <- y to 21

[if](http://scala-lang.org/) x \* x + y \* y == z \* z

} [yield](http://scala-lang.org/) (x, y, z)

结果：

// Vector((3,4,5), (5,12,13), (6,8,10), (8,15,17), (9,12,15), (12,16,20))

**2.16.1.**    **for .. yield**

把每次循环的结果“**移**”进一个集合（类型和循环内的一致）

for {子句} yield {循环体}

正确：

for (e<-List(1,2,3)) yield (e\*e) // List(1,4,9)

for {e<-List(1,2,3)} yield { e\*e } // List(1,4,9)

for {e<-List(1,2,3)} yield e\*e // List(1,4,9)

错误：

for (e<-List(1,2,3)) { yield e\*e } // 语法错误,yield不能在任何括号内

**2.16.2.**      **foreach**

List(1,2,3).foreach(println)

1

2

3

也可以写成：

(1 to 3).foreach(println)

或者

(1 until 4) foreach println

或者

Range(1,3) foreach println

注意：

  to包含，until不包含（最后的数）

  都可以写步长，如：

1 to (11,2) // 1,3,5,7,9,11 步长为2

    1 to 11 by 2

1 until (11,2) // 1,3,5,7,9

1 until 11 by 2

val r = (1 to 10 by 4) // (1,5,9), r.start=r.first=1; r.end=10, r.last=9

  也可以是BigInt

(1:BigInt) to 3

**2.16.3.**      **forall**

"所有都符合"——相当于 A1 && A2 && A3 && ... && Ai && ... && An

(1 to 3) forall (0<) // true

(-1 to 3) forall (0<) // false

又如：

def isPrime(n:Int) = 2 until n forall (n%\_!=0)

for (i<-1 to 100 if isPrime(i)) println(i)

(2 to 20) partition (isPrime \_) // (2,3,5,7,11,13,17,19), (4,6,8,9,10,12,14,15,16,18,20)

也可直接调用BigInt的内部方法：

(2 to 20) partition (BigInt(\_) isProbablePrime(10))

// 注：isProbablePrime(c)中c越大，是质数的概率越高，10对应概率：1 - 1/(2\*\*10) = 0.999

**2.16.4.**      **reduceLeft**

reduceLeft 方法首先应用于前两个元素，然后再应用于第一次应用的结果和接下去的一个元素，等等，直至整个列表。例如

计算阶乘：

def fac(n: Int) = 1 to n reduceLeft(\_\*\_)

fac(5) // 5\*4\*3\*2 = 120

相当于：

    ((((1\*2)\*3)\*4)\*5)

计算sum：

List(2,4,6).reduceLeft(\_+\_) // 12

相当于：

    ((2+4)+6)

取max：

List(1,4,9,6,7).reduceLeft( (x,y)=> if (x>y) x else y ) // 9

       或者简化为：

List(1,4,9,6,7).reduceLeft(\_ max \_) // 9

相当于：

    ((((1 max 4) max 9) max 6) max 7)

**2.16.5.**      **foldLeft scanLeft**

累加或累乘

def sum(L: List[Int]): Int = {

        var result = 0

        for (item <- L)

            result += item

        result

}

更scalable的写法：

def sum(L: Seq[Int]) = L.foldLeft(0)((a, b) => a + b)

def sum(L: Seq[Int]) = L.foldLeft(0)(\_ + \_)

def sum(L: List[Int]) = (0/:L){\_ + \_}

调用：

sum(List(1,3,5,7)) // 16

乘法：

def multiply(L: Seq[Int]) = L.foldLeft(1)(\_ \* \_)

multiply(Seq(1,2,3,4,5)) // 120

multiply(1 until 5+1) // 120

**2.16.6.**      **scanLeft**

List(1,2,3,4,5).scanLeft(0)(\_+\_) // (0,1,3,6,10,15)

相当于：

(0,(0+1),(0+1+2),(0+1+2+3),(0+1+2+3+4),(0+1+2+3+4+5))

List(1,2,3,4,5).scanLeft(1)(\_\*\_) // (1,2,6,24,120)

相当于

(1, 1\*1, 1\*1\*2, 1\*1\*2\*3, 1\*1\*2\*3\*4, 1\*1\*2\*3\*4\*5)

注：

  (z /: List(a, b, c))(op) 相当于 op(op(op(z, a), b), c)

简单来说：加法用0，乘法用1

  (List(a, b, c) :\ z) (op) equals op(a, op(b, op(c, z)))

**2.16.7.**      **take drop splitAt**

1 to 10 by 2 take 3 // Range(1, 3, 5)

1 to 10 by 2 drop 3 // Range(7, 9)

1 to 10 by 2 splitAt 2 // (Range(1, 3),Range(5, 7, 9))

例子：前10个质数

def prime(n:Int) = (! ((2 to math.sqrt(n).toInt) exists (i=> n%i==0)))

2 to 100 filter prime take 10

**2.16.8.**      **takeWhile, dropWhile, span**

while语句的缩写，

|  |  |
| --- | --- |
| takeWhile (...) | 等价于：while (...) { take } |
| dropWhile (...) | 等价于：while (...) { drop } |
| span (...) | 等价于：while (...) { take; drop } |

1 to 10 takeWhile (\_<5) // (1,2,3,4)

1 to 10 takeWhile (\_>5) // ()

10 to (1,-1) takeWhile(\_>6) // (10,9,8,7)

1 to 10 takeWhile (n=>n\*n<25) // (1, 2, 3, 4)

如果不想直接用集合元素做条件，可以定义var变量来判断：

例如，从1 to 10取前几个数字，要求累加不超过30：

var sum=0;

val rt = (1 to 10).takeWhile(e=> {sum=sum+e;sum<30}) // Range(1, 2, 3, 4, 5, 6, 7)

注意：takeWhile中的函数要返回Boolean，sum<30要放在最后；

1 to 10 dropWhile (\_<5) // (5,6,7,8,9,10)

1 to 10 dropWhile (n=>n\*n<25) // (5,6,7,8,9,10)

1 to 10 span (\_<5) // ((1,2,3,4),(5,6,7,8)

List(1,0,1,0) span (\_>0) // ((1), (0,1,0))

注意，partition是和span完全不同的操作

List(1,0,1,0) partition (\_>0) // ((1,1),(0,0))

**2.16.9.**      **break、continue**

Scala中**没有**break和continue语法！需要break得加辅助boolean变量，或者用库（continue没有）.

例子1：打印'a' to 'z'的前10个

var i=0; val rt = for(e<-('a' to 'z') if {i=i+1;i<=10}) printf("%d:%s\n",i,e)

或者：

('a' to 'z').slice(0,10).foreach(println)

例子2：1 to 100 和小于1000的数

var (n,sum)=(0,0); for(i<-0 to 100 if (sum+i<1000)) { n=i; sum+=i }

// n = 44, sum = 990

例子3：使用库来实现break

import [scala.util.control.Breaks.\_](http://lampsvn.epfl.ch/svn-repos/scala/scala/trunk/src/library/scala/util/control/Breaks.scala)

for(e<-1 to 10) { val e2 = e\*e; if (e2>10) break; println(e) }

**2.17.**    **操作符重载**

注意：其实Scala没有操作符，更谈不上操作符重载；+-/\*都是方法名，如1+2其实是(1).+(2)

object operator {

    class complex(val i:Int, val j:Int) { // val 是必须的

        def + (c2: complex) = {

            new complex (i+c2.i, j+c2.j)

        }

        override def toString() = { "(" + i + "," + j + ")" }

    }

    def main(args:Array[String]) = {

        val c1 = new complex(3, 10)

        val c2 = new complex(5, 70)

        printf("%s + %s = %s", c1, c2, c1+c2)

    }

}

编译：fsc operator.scala

运行：java operator // (3,10) + (5,70) = (8,80)

**2.18.**    **系统定义scala.\_**

scala.\_自动加载，只有发生类名冲突时才需要带上scala.包名。

|  |  |  |
| --- | --- | --- |
| scala.AnyValue | 所有基本类型的根 | Int,Char,Boolean,Double,Unit |
| scala.AnyRef | 所有引用类型的根 | 相当于java的java.lang.Object |
| scala.Null | 所有引用类型的子类 |  |
| scala.Nothing | 所有全部类型的子类 |  |
| scala.List | 不可变的List | scala特色的不可变List |
| scala.Int | scala中可以用int作为别名 | Double，Float等类似 |

**2.19.**    **implicit隐式转换**

用途：

  把一种object类型安全地自动转换成另一种object类型;

  不改动已有class设计即可添加新的方法;

**2.19.1.**      **类型转换**

implicit def foo(s:String):Int = Integer.parseInt(s) // 需要时把String->Int

def add(a:Int, b:Int) = a+b

add("100",8) // 108, 先把"100"隐式转换为100

**2.19.2.**      **例子：阶乘n!**

第一步：写函数

def factorial(x: Int) = 1 to n reduceLeft(\_\*\_)

第二步：定义 "!" 函数

    class m1(n: Int) {

        def ! = factorial(n)

    }

implicit def m2(n:Int) = new m1(n) // **隐式转换**，即**在需要时**把n转换为new m1(n)

注意：上面可以用匿名类简化为：

    implicit def m2(n:Int) = new { def ! = factorial(n) }

第三步：使用

    val n = 100

printf("%d! = %s\n", n, (n!))  // n! 相当于 new m1(n).!()

println(10!)

**2.19.3.**      **例子：cout**

import java.io.\_

class C1(p:PrintStream) {

def << (a:Any) = {

p.print(a)

p.flush

p

}

}

implicit def foo(p:PrintStream) = new C1(p)

val endl = '\n'

System.out<<"hello"<<" world"<<endl

**2.19.4.**      **例子：定义?:**

implicit def elvisOperator[T](alt: T) = new {

  def ?:[A >: T](pred: A) = if (pred == null) alt else pred

}

null ?: "" // ""

"abc" ?: "" // "abc"

10 ?: 0 // 10

(null ?: 0).asInstanceOf[Int] // 0

**2.19.5.**      **已有Object加新方法**

object NewMethod {

  // 定义新方法join()

**implicit def** foo1[T](list: List[T]) = new {

    def **join**(s:String) = list.mkString(s)

  }

  // 测试

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

    Console println List(1,2,3,4,5).join(" - ") // " 1 - 2 - 3 - 4 – 5"

  }

}

解释：

编译器发现List没有join(String)方法，就发查找代码中有没有定义在implicit def xx(List)内的join(String)方法，如果有就调用这个。

为Int增加乘方操作：

def pow(n:Int, m:Int):Int = if (m==0) 1 else n\*pow(n,m-1)

implicit def foo(n:Int) = new {

  def \*\*(m:Int) = pow(n,m)

}

2\*\*10 // 1024

例子2：定义如ruby的10.next

implicit def foo(n:Int) = new { def next = n+1 }

10.next // 11

**2.20.**    **type做alias**

相当于C语言的类型定义typedef，建立新的数据类型名（别名）；在一个函数中用到同名类时可以起不同的别名

例如：

type JDate = java.util.Date

type SDate = java.sql.Date

val d1 = new JDate()  // 相当于 val d = new java.util.Date()

val d2 = new SDate()  // 相当于 val d = new java.sql.Date()

注意：type也可以做泛型

**2.21.**    **泛型**

**2.21.1.**      **函数中的泛型：**

def foo[T](a:T) = println("value is " + a)

foo(10) // "value is 10"

foo(3.14) // "value is 3.14"

foo("hello") // "value is hello"

**2.21.2.**      **类中的泛型：**

class C1[T] {

private var v:T = \_   // 初始值为T类型的缺省值

def set(v1:T) = { v = v1 }

def get = v

}

new C1[Int].set(10).get // 10

new C1[String].set("hello").get // "hello"

**2.21.3.**      **泛型qsort例子**

def qsort[T <% Ordered[T]](a:List[T]): List[T] = if (a.size<=1) a else {

val m = a(a.size/2)

qsort(a.filter(m>)) ++ a.filter(m==) ++ qsort(a.filter(m<))

}

调用：

        val a = List(1, 3, 6, 2, 0, 9, 8, 7, 2)

        qsort[Int](a) // 注意2 a

        val b = List(1.0, 3.3, 6.2, 6.3, 0, 9.5, 8.7, 7.3, 2.2)

        qsort[Double](b) // 注意2 b

或者使用自带库：

List(1, 3, 6, 2, 0, 9, 8, 7, 2) sortWith(\_<\_)

List(1.0, 3.3, 6.2, 6.3, 0, 9.5, 8.7, 7.3, 2.2).sortWith(\_<\_)

**2.21.4.**      **泛型add例子**

// 采用implicit parameters

def add[T](x:T, y:T)(implicit n:Numeric[T]) = {

n.plus(x,y) // 或者用 import n.\_; x + y

}

add(1,2) // 3

add(1,2.14) // 3.14

再如，求max：

def max2[T](x:T, y:T)(implicit n:Numeric[T]) = {

  import n.\_

  if (x > y) x else y

}

max2(2, 3.14) // 3.14

**2.21.5.**      **泛型定义type**

**abstract** class C1 {

**type T**

    val e:T

}

**abstract** class C2 {

**type T**

    val list:List[T]

    def len = list.length

}

def m1(e1:Int) = new C1 {

**type T = Int**

    val e = e1

}

def m2(e1:List[Int]) = new C2 {

**type T = Int**

    val list = e1

}

Console println m1(10) // 10

Console println m2(List(1,2,3,4,5)).len // 5

注意：type也可以做数据类型的alias，类似C语言中的typedef

**2.21.6.**      **@specialized**

def test[@specialized(Int,Double) T](x:T)  = x match { case i:Int => "int"; case \_ => "other" }

没有@specialized之前，是编译成Object的代码；用了@specialized，变成IntTest(),DoubleTest(),...

编译后的文件尺寸扩充了，但性能也强了，不用box，unbox了

**2.22.**    **枚举Enum**

Scala没有在语言层面定义Enumeration，而是在库中实现：

例子1：

object Color extends Enumeration {

  type Color = Value

  val RED, GREEN, BLUE, WHITE, BLACK = Value

}

Color.RED // Color.Value = RED

import Color.\_

val colorful = Color.values filterNot (e=> e==WHITE || e==BLACK)

colorful foreach println // RED\nGREEN\nBLUE

例子2：

object Color extends Enumeration {

  val RED = Value("红色")

  val GREEN = Value("绿色")

  val BLUE = Value("蓝色")

  val WHITE = Value("黑")

  val BLASK = Value("白")

}

Color.RED // Color.Value = 红色

import Color.\_

val colorful = Color.values filterNot (List("黑","白") contains \_.toString)

colorful foreach println //红色\n绿色\n蓝色

3.   **FP**

**3.1.**     **函数**

函数的地位和一般的变量是同等的，可以作为函数的参数，可以作为返回值。

传入函数的任何输入是只读的，比如一个字符串，不会被改变，只会返回一个新的字符串。

Java里面的一个问题就是很多只用到一次的private方法，没有和使用它的方法紧密结合；Scala可以在函数里面定义函数，很好地解决了这个问题。

**3.1.1.**  **函数定义**

函数和方法一般用def定义；也可以用val定义匿名函数，或者定义函数别名。

def m0(x:Int) = x\*x

val m1 = (x:Int)=> x\*x // ()是必须的

val m2 = {x:Int=> x\*x} // 不用(), 用{}

m0(10) // 100

m1(10) // 100

m2(10) // 100

不需要返回值的函数，可以使用def f() {...}，永远返回Unit（即使使用了return）,即：

    def f() {...}  等价于 def f():Unit = {...}

例如：

def f() { return "hello world" }

f() // Unit，而不是 "hello world"

需要返回值的函数，用 def f() = {...} 或者 def f = {...}

def f() = { "hello world" }

f() // "hello world"

def f = { "hello world" } // f是匿名函数 {"hello world"}的别名

f // "hello world"

三种定义方式的区别：

|  |  |  |
| --- | --- | --- |
| def f() { return .. } | 调用：f, f()皆可 | 始终返回：Unit |
| def f() = ... | 调用：f, f()皆可 | 返回Unit或者值 |
| def f = ... | 调用：f | 返回Unit或者值 |

提示：函数式风格——尽量编写有返回值的函数; 尽量简短（函数体不使用{...}）

**3.1.2.**  **映射式定义**

一种特殊的定义：映射式定义（直接相当于数学中的映射关系）;

其实也可以看成是没有参数的函数，返回一个匿名函数；调用的时候是调用这个返回的匿名函数。

例子1：

def f:Int=>Double = { // 请看做 def f: (Int=>Double) = {...}

    case 1 => 0.1

    case 2 => 0.2

    case \_ => 0.0

}

f(1) // 0.1

f(3) // 0.0

例子2：

def m:Option[User]=>User = {

    case Some(x) => x

    case None => null

}

m(o).getOrElse("none...")

例子3：(多->1)

def m:(Int,Int)=>Int = \_+\_

m(2,3) // 5

例子4：

def m:Int=>Int = 30+  // 相当于30+\_,如果唯一的"\_"在最后,可以省略

m(5) // 35

**3.1.3.**  **特殊函数名 + - \* /**

方法名可以是\*：

def \*(x:Int, y:Int) = { x\*y }

\*(10,20) // = 200

1+2, 相当于1.+(2)

定义一元操作符（置前）可用unary\_。

*注：unary ：一元的，单一元素的, 单一构成的 。发音：【`ju: ne ri】*

-2，相当于：(2).unary\_- // -2

+2，相当于：(2).unary\_+ // 2

!true, 相当于：(true).unary\_! // false

~0，相当于 (0).unary\_~  // -1

**3.1.4.**  **缺省参数、命名参数**

注意：从Scala2.8开始支持。

请对比以下两种写法的可读性：

|  |  |
| --- | --- |
| **一般函数调用** | **命名函数调用** |
| sendEmail(  "jon.pretty@example.com",  List("recipient@example.com"),  "Test email",  b,  Nil, Nil, Nil) | sendEmail(      body = b,      subject = "Test email",      to = List("recipient@example.com"),      from = "jon.pretty@example.com",      attachments = List(file1, file2) ) |

定义和使用：

def join(a:List[String], s:String="-") = { a.mkString(s) }

join(List("a","b","c")) // a-b-c

join(List("a","b","c"), ":") // a:b:c

调整参数调用顺序：

join(s=":",a=List("a","b","c")) // a:b:c

**3.1.5.**  **函数调用**

def f(s: String = "default") = { s }

f // "hello world"

f() // "hello world"

  对象的无参数方法的调用，可以省略.和()

"hello world" toUpperCase // "HELLO WORLD"

  对象的1个参数方法的调用，可以省略.和()

"hello world" indexOf w // 6

"hello world" substring 5 // "world"

Console print 10 // 但不能写 print 10，只能print(10)，省略Console.

1 + 2 // 相当于 (1).+(2)

  对象的多个参数方法的调用,也可省略.但不能省略()：

"hello world" substring (0, 5) // "hello"

**注意：**

  不在class或者object中的函数不能如此调用：

def m(i:Int) = i\*i

m 10 // 错误

  但在class或者object中可以使用this调用：

object method {

    def m(i:Int) = i\*i

    def main(args: Array[String]) = {

        val ii = this m 15  // 等同于 m(15), this 不能省略

        println(ii)

    }

}

提示：这种调用方法最大的用途在于操作符重载，以及阅读性强的DSL。

例子：

    System exit 0

    Thread sleep 10

    Console println "hello world" // 省略Console则报错

    2 \*\* 10

    "-" \* 100

**3.1.6.**  **匿名函数**

形式：((命名参数列表)=>函数实现)(参数列表)

特殊地：

  无参数： (()=>函数实现)()

  有一个参数且在最后： (函数实现)(参数)

  无返回值： ((命名参数列表)=>Unit)(参数列表)

有参数的匿名函数（参数名称不能省）：

((i:Int)=> i\*i)(3) // 9

((i:Int, j:Int) => i+j)(3, 4) // 7

有一个参数且在最后的：

(10\*)(2) // 20, 相当于 ((x:Int)=>10\*x)(2)

(10+)(2) // 12, 相当于 ((x:Int)=>10+x)(2)

(List("a","b","c") mkString)("=") // a=b=c

无参数的匿名函数：

(()=> 10)() // 10

无参数无返回值：

(() => Unit)

( ()=> {println("hello"); 20\*10} )()

{ println("hello"); 20\*10 }

例子1：

def m = (i:Int)=> i\*i

m(3) // 9

List(1,2,3,4).map(m) // List(1, 4, 9, 16)

例子2：

def times3(m: ()=> Unit) = { m();m();m() }

times3 ( ()=> println("hello world") )

由于是无参数的匿名函数，可进一步简化：

def times3(m: =>Unit) = { m;m;m } // 参见“lazy参数”

times3 ( println("hello world") )

**3.1.7.**  **偏函数**

用下划线代替1+个参数的函数叫偏函数（partially applied function），如：

def sum(a:Int,b:Int,c:Int) = a+b+c

val p0 = sum \_ // 正确

val p1 = sum(10,\_,20) // 错误

val p2 = sum(10,\_:Int,20) // 正确

val p3 = sum(\_:Int,100,\_:Int)

p0(1,2,3) // 6

p2(100) // 130

p3(10,1) // 111

或者：

  (sum \_)(1 2 3) // 6

  (sum(1,\_:Int,3))(2) // 6

  (sum(\_:Int,2,\_:Int))(1,3) // 6

从上面可以看出，partial函数是一个正常函数中抽出来的一部分（参数不全），故称为partial函数。

**3.1.8.**  **“\_”匿名参数**

\_：匿名函数中的匿名参数，同Groovy的it一样，groovy： 3.times { print it }；但也有差别：

|  |  |  |
| --- | --- | --- |
|  | Groovy | Scala |
| 同 | print({it\*10}(5)) // 50 | ((\_:Int)\*10)(5) // 50  强类型语言，必须标明类型为Int |
| 不同 | def a = {it\*it}(5) // 25    两个it完全一样 | ((\_:Int)\*(\_:Int))(5,5) // 25  ((it:Int)=>it\*it)(5) // 25  不能用：((\_:Int)\*(\_:Int))(5)  两个\_不一样，表示2个不同参数 |

注意：多个下划线指代多个参数，而不是单个参数的重复使用。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。

Java：

*int* *add*(int i, int j) { *return* i+j; }

add(3,4)

返回值、函数名和return关键字可以省略，变为：

scala 写法1：

((i:Int, j:Int) => i+j)(3, 4) // 7

i,j变量名可以省略，变为：

scala写法2：

    ((\_:Int) + (\_:Int))(3,4) // 7

    注意：括号(\_:Int)括号是必须的

其他例子：

"Hello".exists(\_.isUpper) // true

例子2：

def sum(x:Int,y:Int,z:Int) = x+y+z

val sum1 = sum \_

val sum2 = sum(1000,\_:Int,100)

sum1(1,2,3) // 6

sum2(5) // 1105

如果\_在最后，还可以省略

1 to 5 foreach println

1 to 5 map (10\*)

1 to 5 map (\*10) // 错误！\_不在最后不能省

**3.1.9.**  **变长参数 \***

变长参数只能放在最后一个，否则就歧义了，

def sum(ns:Int\*, s:String) = ... // 错误

例子1：

def sum(ns: Int\*) = {

    var s = 0

    for (n<-ns) s += n

    s

}

sum(1,2,3,4) // 10

更函数化的写法：

def sum(ns:Int\*) = ns.reduceLeft(\_+\_)

sum(1,2,3,4) // 10

例子2：

def m(args:Any\*) = args foreach println

scala> m(3,3.14,"hello",List(1,2,3))

3

3.14

hello

List(1, 2, 3)

**3.1.10.**      **lazy参数**

就是调用时用到函数的该参数，每次都重新计算：

例子1：

|  |  |
| --- | --- |
| **一般参数** | **lazy参数** |
| def f1(x: Long) = {    val (a,b) = (x,x)    println("a="+a+",b="+b)  } | def f2(x: =>Long) = {    val (a,b) = (x,x)    println("a="+a+",b="+b)  } |
| f1(System.nanoTime)  // a=140857176658355,b=140857176658355 | f2(System.nanoTime)  // a=140880226786534,b=140880226787666 |

例子2： lazy参数是函数

|  |  |
| --- | --- |
| **一般参数** | **lazy参数** |
| def times3(m: Unit) = { m;m;m } | def times3(m: =>Unit) = { m;m;m } |
| times3 ( println("hello world") )  // 打印1次 | times3 ( println("hello world") )  // 打印3次 |

例子3：

|  |  |
| --- | --- |
| **一般参数** | **lazy参数** |
| def f(x:Int m: Unit) = { print(x\*x); m } | def f(x:Int, m: =>Unit) = { print(x\*x); m } |
| f(3,println("end"))  // 先执行m，输出："end\n9" | f(3,println("end"))  // lazy执行m，输出 “9end” |

例子4：自定义loop

def loop2 (cond: =>Boolean)(body: =>Unit): Unit =

  if (cond) { body; loop2(cond)(body) }

调用：

var i=0; loop2 (i<3) (i+=1) // i=3

或者

var i=0; loop2 {i<3} {i+=1} // i=3

或者我们更习惯的方式:

var i=0; loop2 (i<3) {i+=1}

例子5：自定义loop..until

def loop3 (body: =>Unit): foo = new foo(body)

class foo(body: =>Unit) {

  def until(cond: =>Boolean) { body; if(!cond) until(cond) }

}

//调用：

var i = 0

loop3 { i+=1; println(i) } until (i>3)

**3.2.**     **函数式编程**

**3.2.1.**  **风格**

首选使用val，用var之前要仔细考虑是否真正需要。

|  |  |
| --- | --- |
| **非函数式** | **函数式** |
| var s = ""  if (args.length>0) s = args[0] | val s = if (args.length>0) args[0] else "" |
| def gcd(a0:Int,b0:Int) = {    var a = a0; var b = b0    while(a!=0) {  val t = a  a = a % b  b = t    }    return b  } | 求a，b的最大公约数,所有的while循环基本都可以被替代：  def gcd(a:Int,b:Int) =  if (b==0) a else gcd(b, a % b) |

函数要短，可以试着所有的函数体都没有 {...} 包围，一旦出现，尽量分解

例如打印

1  2  3

2  4  6

3  6  9

|  |  |
| --- | --- |
| **非函数式** | **函数式** |
| for(i<-1 to 3; j<-1 to 3) {  print(i\*j + " ");  if (j==3) println  } | def f(n:Int) = 1 to 3 map (\_\*n) mkString " "  1 to 3 map f mkString "\n" |

**3.2.2.**  **函数作为参数**

采用映射式函数定义：

例子1：

def f(x:Int, y:Int, m:(Int, Int)=>Int) = m(x,y)

f(3,4, (x,y)=>x+y) // 7

f(3,4, (x,y)=>scala.math.sqrt(x\*x+y\*y)) // 5

例子2：

def f(x:Int, y:Int, m: =>Unit) = { println(x\*y); m } // 参见“lazy参数”

f(3,4, println("end"))

// 12

// end

例子3：

  def f(f2:Int=>Int) = f2(5)

  f(100+) // 105, 100+是匿名函数 ((x:Int)=>100+x)的简写

**3.2.3.**  **函数作为返回值**

def f(s:String):(Int => String) = { n:Int=> s \* n }

f("\*") // res32: (Int) => String = <function1>

res32(10) // "\*\*\*\*\*\*\*\*\*\*"

f("\*")(10) // 相当于先得到f("\*")的返回值函数，再用该返回值函数调用一个参数

**3.2.4.**  **Curry化**

例如：

def sum(a:Int, b:Int) = { a + b } // sum(1, 2) = 3

Curry化后：

def sum(a:Int)(b:Int) = { a + b } // sum(1)(2) = 3

或者：

def sum(a:Int) = { (b:Int)=> a + b } // sum(1)(2) = 3

// 调用方式二：val t1 = sum(10);  val t2 = t1(20)

**3.2.5.**  **递归**

使用递归的原因：

其一是很多数学关系、逻辑关系本身就是递归描述（Hanoi、Fib）的；

其二是函数式编程不鼓励用变量循环，而是用递归。

递归包含递归出口和递归体：

|  |  |
| --- | --- |
| 递归出口 | 即递归的终止条件  比如0!=1, fib(1)=fib(2)=1, index==1000 |
| 递归体 | 即和前面或后面的值之间的关系  比如n! = n\*(n-1)!, fib(n)=fib(n-2)+fib(n-1) |

例子1：幂运算

如计算2^100, scala.math.pow(2,2000)越界，可用递归

def pow(n:BigInt, m:BigInt):BigInt = if (m==0) 1 else pow(n,m-1)\*n

例子2：Hanoi汉诺塔

从a挪到c

|  |  |  |
| --- | --- | --- |
| **a(from)** | **b(via)** | **c(to)** |
| **=     --- 1**  **===**  **=====   --- n-1**  **=======  --- n** |  |  |
| **=======  --- n** | **=     --- 1**  **===**  **=====   --- n-1** |  |
|  | **=     --- 1**  **===**  **=====   --- n-1** | **=======  --- n** |
|  |  | **=     --- 1**  **===**  **=====   --- n-1**  **=======  --- n** |

def move(n:Int, a:Int, b:Int, c:Int) = {

  if (n==1) println("%s to %s" format (a, c)) else {

move(n-1, a,c,b); move(1,a,b,c); move(n-1,b,a,c) }}

例子3：最大公约数

12  20

    20  12 (=12%20)

12  8 (=20%12)

        8  4 (=12%8)

           4  0 (=8%4)

def gcd(n:Int, m:Int):Int = if(m==0) n else gcd(m, n%m)

例子4：上台阶

上N级台阶，即可每次1步也可每次2步走，共有多少种不同走法

解法：迈出第一步有两种方法，第一步后就是N-1和N-2的走法了

def step(n:Int):Int = if (n<=2) n else step(n-1)+step(n-2)

推广：如果每次可走1步或2步或3步，则

def step(n:Int):Int = n match { case 1=>1; case 2=>2; case 3=>4;

case \_ =>step(n-1)+step(n-2)+step(n-3) }

**3.2.6.**  **尾-递归**

定义：函数尾（最后一条语句）是递归调用的函数。

tail-recursive会被优化成循环，所以没有堆栈溢出的问题。

线性递归的阶乘：

def nn1(n:Int):BigInt = if (n==0) 1 else nn1(n-1)\*n

println(nn1(1000)) // 4023...000

println(nn1(10000)) // 崩溃：（

尾递归的阶乘：

def nn2(n:Int, rt:BigInt):BigInt = if (n==0) rt else nn2(n-1, rt\*n)

println(nn2(1000,1)) // 40...8896

println(nn2(10000,1)) // 2846...000

def nn3(n:Int):BigInt = nn2(n,1)

对比：

|  |  |
| --- | --- |
| **线性递归** | **尾递归** |
| nn1(5)   {5 \* nn1(4) } {5 \* {4 \* nn1(3) }} {5 \* {4 \* {3 \* nn1(2) }}} {5 \* {4 \* {3 \* {2 \* nn1(1) }}}} {5 \* {4 \* {3 \* {2 \* 1}}}} {5 \* {4 \* {3 \* 2}}} {5 \* {4 \* 6}} {5 \* 24} 120 | nn2(5, 1) nn2(4, 1\*5=5) nn2(3, 4\*5=20) nn2(2, 3\*20=60) nn2(1, 2\*60=120) 120 |
| 不算，直到递归到一个确定的值后，又从这个具体值向后计算；更耗资源,每次重复的调用都使得调用链条不断加长. 系统使用栈进行数据保存和恢复 | 每递归一次就算出相应的结果。 |
| 不能优化成循环 | 可以优化成循环 |

例子：查找第10001个质数：

def prime(n:Int) = 2 to math.sqrt(n).toInt forall (n%\_!=0)

def next(p:Int) = p+1 to 2\*p find prime get

def f(p:Int, i:Int):Int = if (i==10001) p else f(next(p), i+1)

f(2,1) // 104743

或者使用Stream：

def f2(p:Int):Stream[Int] = p #:: f2(next(p))

f2(2).take(10001).last // 104743

例子：fib数列(1 1 2 3 5 8 13 21)

def fib2(n1:BigInt, n2:BigInt, i:Int, n:Int):BigInt =

  if (i==n) n1 else fib2(n2, n1+n2, i+1, n)

def fib(n:Int) = fib2(1,1,0,n)

0 to 10 map fib

fib(100-1) // 第100个fib数=354224848179261915075

4.   **OOP**

**4.1.**     **类class**

**4.1.1.**  **定义**

例子1：

class User {

var name = "anonymous"

var age:Int = \_

val country = "china"

def email = name + "@mail"

}

使用：

val u = new User

// var定义的属性可读可写

u.name = "qh"; u.age = 30

println(u.name + ", " + u.age) // "qh, 30"

// val 定义的属性只读不可写

u.country = "usa" // 报错

println(u.country) // "china"

// def 定义的是方法, 每次调用时重新计算

u.email // "qh@mail"

例子2：

// 定义

class Person(ln : String, fn : String, s : Person = null) {

  def lastName = ln;   // 用def定义后才是属性，ln，fn，s不可见

  def firstName = fn;

  def spouse = s;

  def introduction() : String =

    return ("Hi, " + firstName + " " + lastName) +

    (if (spouse != null) " and spouse, " + spouse.firstName + " " + spouse.lastName + "."

     else ".");

}

// 调用

new Person("aa","bb", new Person("cc","dd")).introduction();

**4.1.2.**  **构造方法**

class c1(x:String) // 等同于：class c1(private var x:String)

val o1 = new c1("aaa")

o1.x // 报错，因为是private的，定义成 class c1(var x:String) 才能这样用

例子1：

object construct1 {

class c1(name:String, age:Int) { // (1)直接在类定义处

    def this() { this("anonymous", 20) } // (2)用this定义

        def m1() = {  printf("%s=%d\n", name, age) }

    }

    def main(args:Array[String]) = {

new c1().m1()

new c1("qh", 30).m1()

}

}

编译：fsc construct1.scala

运行：java construct1

例子2：继承中的构造方法：

class c2(name:String, age:Int, female:Boolean=false)

    extends c1(name,age) {

    override def toString = { name + "," +  age + "," + female }

}

**4.1.3.**  **override**

不同于Java的使用 @Override，或者直接使用相同名字覆盖父类方法。

    override def toString = { name + "," +  age + "," + female }

如果是覆盖抽象方法，可以不用overriade关键字。

**4.1.4.**  **object单例对象**

如：

|  |  |
| --- | --- |
| Java | **Scala** |
| public class User {    private String name;    private User(String name) { this.name=name; }    public static User instance(String name) {  return new User(name)    }  } | object User {    var name:String = \_    def apply(name:String){this.name=name; this}    override def toString = "name: " + name  }  调用：  val u = User("qh") // "name: qh" |

**4.1.5.**  **静态方法**

Scala没有静态方法，类似静态方法的函数定义在object中：

object Stringx {

    def left(s0:String, s:String) = ...

}

直接调用Stringx.left(s0, s)，或者 Stringx left (s0, s)

定义在object中的implicit方法也能被直接调用：

例如：

--------- ImportSub.scala

object ImportSub {

  def fac(n: Int) = 1 to n reduceLeft (\_ \* \_)

  implicit def foo(n: Int) = new { def ! = fac(n) }

}

--------- ImportMain.scala

import ImportSub.\_

object ImportMain {

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

      println(5!) // 调用ImportSub中定义的implicit函数

  }

}

**4.1.6.**  **case class(条件类)**

例如：

case class Person(name:String, age:Int)

特殊之处：

  新建类实例不用new Person(..)，直接用Person("qh",20)

  自动定义好getXX方法，Person("qh",20).name // "qh"

  提供默认的toString(), Person("qh",20) // "Person(qh,20)"

  结合类继承可以通过模式匹配进行分解

例子1：

abstract class Person

case class Man(power:Int) extends Person

case class Woman(beauty:Int, from:String) extends Person

val w1 = Woman(100,"china")

val w2 = w1.copy(from="usa") // Woman(100,"usa")

def f(t:Person) = t match {

case Man(x) => "man's power:" + x

case Woman(x,y) => y + " beauty:" + x

}

f(Man(100)) // man's power:100

f(Woman(90, "china")) // china beauty:90

注：基本类型直接可以用[math case](http://qiujj.com/static/Scala-Handbook.htm#_match..case(switch))

例子2：可变的类状态

case class C1(var s: String, var ops: Int) {

  def >>= (f: (String=>String)) = {

    s = f(s)  // s改变

    ops += 1  // ops改变

    this // 返回自身，可以连续调用

  }

}

val C1(res, ops) = C1("ab", 0) >>= (\_ \* 3) >>= (\_ drop 3)

// res="ab"->"ababab"->"bab", ops=0-> 0+1+1->2

例子3：用case class代替tuple

val p = ("qh",20) // p.\_1 = "qh", p.\_2 = 20；好处是简洁，但无意义

case class person(name:String, age:Int)

val p = person("qh",20) // p.name = "qh", p.age = 20; 好处是有名字，自说明，可读性强

例子4：用case class来描述元数据

xml的版本：

<todo name = "housework">

<item priority = "high">Clean the hose</item>

<item priority = "medium">Wash the dishes</item>

<item priority = "medium">Buy more soap</item>

</todo>

Lisp的版本：

(todo "housework"

 (item (priority high) "Clean the house")

 (item (priority medium) "Wash the dishes")

 (item (priority medium) "Buy more soap"))

Scala的版本：

    case class item(priority:String, s:String)

    case class todo(name:String, items:List[item])

    todo (name="housework",

items=item("high","Clean the house")::

item("medium","Wash the dishes")::

item("medium","Buy more soap")::Nil)

**4.1.7.**  **case object(条件单例对象)**

比如定义一个标识类（而不是字符串）：

case object Start

case object Stop

**4.1.8.**  **枚举**

Java中：

   enum fruits { apple, banana, cherry }

在Scala中，则是：

sealed abstract class Fruits // sealed类似于java的final

case object Apple extends Fruits

case object Banana extends Fruits

case object Cherry extends Fruits

也可以是 case class

**4.1.9.**  **属性和Bean**

例子1（直接定义和使用属性）：

class c {

var name = "anonymous" // var定义的是r/w的属性

val age = 20  // val定义的是只r属性

}

val o = new c

o.name = "qh"

o.name // "qh"

o.age = 10 // 错误

o.age // 20

o.

例子2（定义get/set方法）：

class c2 {

@reflect.BeanProperty var name = "anonymous"

}

val o2 = new c2

o2.name = "qh" // 也可以直接存取

o2.name // "qh"

o2.setName("james") // 增加了set/get方法

o2.getName() // "james"

**4.1.10.**      **反射**

Scala没有太特别的反射机制，使用java的即可，不过Scala在match..case中可以匹配类型：

    case o:FooClass1 => ...

相关还有isInstanceOf[T], asInstanceOf[T]

例1（利用java的reflect）：

"hello".getClass.getMethods.map(\_.getName).toList.sortWith(\_<\_).mkString(", ")

例子2：

classOf[String] // 相当于java中的String.class

"aaa".isInstanceOf[String] // true

"aaa".asInstanceOf[String]

**4.2.**     **trait超级接口**

*注：trait  [treit] n.特征，特点，特性*

和Java的Interface类似，但可以定义实体方法，而非仅仅方法定义

trait可以看作有方法实现和字段的interface；代表一类事物的特性；

比如

Tom，可能是Engine 和Son两个trait的混合；

Sunny可能Sales、Son和Father三个trait的混合；

当在运行时往Son里面增加方法或者字段的时候，Tom和Sunny都得到增加的特性。

**4.2.1.**  **trait使用**

trait Runnable {

  def run(): Unit;

}

只是用一个接口，就用extends:

class c1 extends Runnable {...}

2个接口（或一个继承一个接口），用with而不是implements如下：

class c1 extends c0 with Runnable {

    def run(): Unit = {...}

}

一个类可以组合多个trait：

class c1 extends t1 with t2 with t3 {...}

**4.2.2.**  **mixin**

class Human

class Child

trait Dad {

    private var children:List[Child] = Nil

    def add(child:Child) = child :: children

}

class Man1(name:String) extends Human with Dad // 静态mixin

class Man2(name:String) extends Human // 先不具备Dad trait

val m1 = new Man1("qh")

m1.add(new Child)

val m2 = new Man2("小孩")

//    m2.add(new Child) // 报错

val m2$ = new Man2("james") with Dad // 动态mixin

m2$.add(new Child)

**4.3.**     **协变和逆变(co-|contra-)variance**

**4.3.1.**  **概念**

使用“+”“-”差异标记

Function[A, B]和Function[-A, +B]的区别图示：

|  |  |
| --- | --- |
| http://qiujj.com/static/Scala-Handbook.files/image008.jpg | http://qiujj.com/static/Scala-Handbook.files/image009.jpg |
| Function[A,B] | Function[-A,+B] |

|  |  |
| --- | --- |
| trait Queue[T] {} | 非变 |
| trait Queue[+T] {} | 协变  如果S extends A (S为子类型，A为父类型)，  则Queue[S]为子类型，Queue[A]为父类型  S <: A => Queue[S] <: Queue[A] |
| trait Queue[-T] {} | 逆变  如果S extends A (S为子类型，A为父类型)  则Queue[S]为父类型，Queue[A]为子类型，和协变互逆  S <: A => Queue[S] >: Queue[A] |

-A是A的子集，叫逆变

+B是B的超集，叫协变

**4.3.2.**  **类型上下界**

|  |  |
| --- | --- |
|  |  |
| <% | foo[T <% Ordered[T]](...)  关系较弱：T能够隐式转换为Ordered[T] |
| <: | foo[T <: Ordered[T]](...)  关系较强：T必须是Ordered[T]的子类型，即T的类型范围小于Ordered[T]，Ordered[T]为上界 |
| >: | foo[T >: A](...)  关系较强：T必须是A的父类型，即Tde类型范围大于A，A为下界 |

**4.3.3.**  **协变、逆变结合上下界**

例子1：

|  |  |
| --- | --- |
| trait c1[+T] {  def m[K >: T](x:K) = x } | trait c1[-T] {  def m[K <: T](x:K) = x } |
| object c2 extends c1[Int]  c2.m(3) // 3  c2.m(3.0) // 3.0  c2.m("abc") // "abc" | object c2 extends c1[Int]  c2.m(3) // 3  c2.m(3.0) // 报错  c2.m("abc") // 报错 |
|  |  |

例子2：

// 非变

case class T1[T](e:T)

val v1:T1[java.lang.Integer] = new T1(100)

val v2:T1[java.lang.Integer] = v1

v2.e // 100

val v3:T1[java.lang.Number] = v1 // 报错

// 协变

case class T1[+T](e:T)

val v1:T1[java.lang.Integer] = new T1(100)

val v2:T1[java.lang.Integer] = v1

v2.e // 100

val v3:T1[java.lang.Number] = v1 // 合法

v3.e // 100

val v4:T1[java.lang.Integer] = v3 //非法

// 逆变

class T1[-T](e:T)

val v1:T1[java.lang.Number] = new T1(100)

val v2:T1[java.lang.Number] = v1

val v3:T1[java.lang.Integer] = v1 // 合法

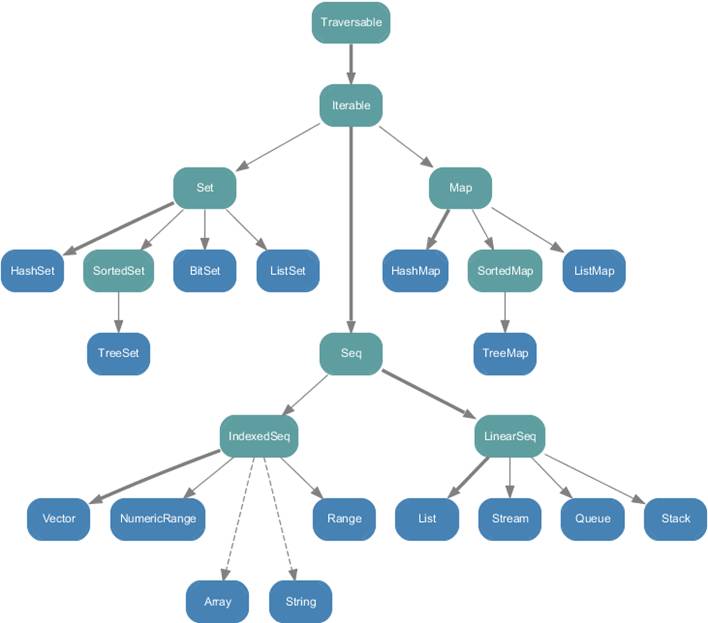
val v4:T1[java.lang.Number] = v3 // 非法

5.   **util包**

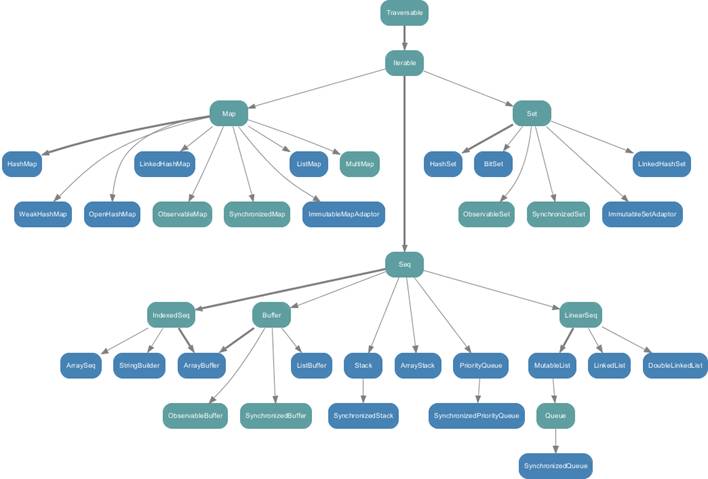
**5.1.**     **架构**

<http://www.scala-lang.org/docu/files/collections-api/collections.html>

scala.collection.immutable



scala.collection.mutable



|  |  |
| --- | --- |
| **不可变（collection.immutable.\_）** | **可变（collection.mutable.\_）** |
| Array | ArrayBuffer |
| List | ListBuffer |
| String | StringBuilder |
| / | LinkedList, DoubleLinkedList |
| List | MutableList |
| / | Queue |
| Array | ArraySeq |
| Stack | Stack |
| HashMap HashSet | HashMap HashSet |
|  | ArrayStack |

**5.2.**     **集合Array,List,Tuple**

|  |  |  |  |
| --- | --- | --- | --- |
| Array | 长度固定 | 元素可变 | 确定长度，后赋值; |
| List | 长度固定 | 元素不可变 |  |
| Tuple | 长度固定 | 元素不可变 | 常用于有多个返回值的函数；或者多个变量的同时定义 |

Scala 2.8中，3者的元素都可以混合不同的类型（转化为Any类型）；

Scala 2.7中，Array、List都不能混合类型，只有Tuple可以；

**5.2.1.**  **定义和初始化**

**5.2.1.1**     **Array**

val list1 = new Array[String](0) // Array()

val list2 = new Array[String](3) // Array(null, null, null)

val list3:Array[String] = new Array(3) // // Array(null, null, null)

val list1 = Array("a","b","c","d") // 相当于Array.apply("a","b","c","d")

定义一个类型为Any的Array：

val aa = Array[Any](1, 2)

或：

val aa: Array[Any] = Array(1, 2)

或：

val aa: Array[\_] = Array(1, 2)

定义：

Array (1,3,5,7,9,11)

也可以用

Array[Int](1 to 11 by 2:\_\*)

Array对应的可变ArrayBuffer：

val ab = collection.mutable.ArrayBuffer[Int]()

ab += (1,3,5,7)

ab ++= List(9,11) // ArrayBuffer(1, 3, 5, 7, 9, 11)

ab toArray // Array (1, 3, 5, 7, 9, 11)

ab clear // ArrayBuffer()

**5.2.1.2**     **List**

val list:List[Int] = List(1,3,4,5,6) // 或者 List(1 to 6:\_\*)

val list1 = List("a","b","c","d") // 或者 List('a' to 'd':\_\*) map (\_.toString)

元素合并进List用::

val list2 = "a"::"b"::"c"::Nil // Nil是必须的

val list3 = "begin" :: list2 // list2不变，只能加在头，不能加在尾

多个List合并用++，也可以用:::(不如++)

val list4 = list2 ++ "end" ++ Nil

val list4 = list2 ::: "end" :: Nil // 相当于 list2 ::: List("end")

当 import java.util.\_ 之后会产生冲突，需要指明包

scala.List(1,2,3)

ListBuffer是可变的：

val lb = scala.collection.mutable.ListBuffer(1,2,3)

lb.append(4) // ListBuffer(1, 2, 3, 4)

建议定义方式：

val head::body = List(4,"a","b","c","d")

// head: Any = 4

// body: List[Any] = List(a, b, c, d)

val a::b::c = List(1,2,3)

// a: Int = 1

// b: Int = 2

// c: List[Int] = List(3)

定义固定长度的List：

List.fill(10)(2) // List(2, 2, 2, 2, 2, 2, 2, 2, 2, 2)

Array.fill(10)(2) // Array(2, 2, 2, 2, 2, 2, 2, 2, 2, 2)

又如：

List.fill(10)(scala.util.Random.nextPrintableChar)

// List(?, =, ^, L, p, <, \, 4, 0, !)

List.fill(10)(scala.util.Random.nextInt(101))

// List(80, 45, 26, 75, 24, 72, 96, 88, 86, 15)

List对应的可变ListBuffer：

val lb = collection.mutable.ListBuffer[Int]()

lb += (1,3,5,7)

lb ++= List(9,11) // ListBuffer(1, 3, 5, 7, 9, 11)

lb.toList // List(1, 3, 5, 7, 9, 11)

lb.clear // ListBuffer()

**5.2.1.3**     **Vector**

Scala2.8为了提高list的随机存取效率而引入的新集合类型（而list存取前部的元素快，越往后越慢）。

val v = Vector.empty

val v2 = 0 +: v :+ 10 :+ 20 // Vector(0, 10, 20), Vector 那一边始终有":"

v2(1) // 10

v2 updated (1,100) // Vector(0, 100, 20)

Seq的缺省实现是List：

Seq(1,2,3) // List(1, 2, 3)

IndexSeq的缺省实现是Vector:

IndexSeq(1,2,3) // Vector(1, 2, 3)

**5.2.1.4**     **Tuple**

val t1 = ("a","b","c")

var t2 = ("a", 123, 3.14, new Date())

val (a,b,c) = (2,4,6)

最简单的Tuple：

1->"hello world"

和下面的写法是等价的：

(1, "hello world")

**5.2.1.5**     **Range**

Range(0, 5) // (0,1,2,3,4)

等同于：

0 until 5

等同于：

0 to 4

两个Range相加：

('0' to '9') ++ ('A' to 'Z') // (0,1,..,9,A,B,...,Z)

Range和序列转换：

1 to 5 toList

相当与：

List(1 to 5:\_\*)

或者：

Vector(1 to 5: \_\*) // Vector(1,2,3,4,5)

**5.2.1.6**     **Stream**

Stream相当于lazy List，避免在中间过程中生成不必要的集合。

定义生成：

val st = 1 #:: 2 #:: 3 #:: Stream.empty // Stream(1, ?)

例子：fib数列的Stream版本简单易懂

def fib(a: Int, b: Int): Stream[Int] = a #:: fib(b,  a+b)

val fibs = fib(1, 1).take(7).toList // List(1, 1, 2, 3, 5, 8, 13)

fib数列的前后项比值趋于黄金分割：

def fn(n:Int) = fib(1,1)(n)

1 to 10 map (n=> 1.0\*fn(n)/fn(n+1)) // Vector(0.5, 0.666, ..., 0.618)

例子1：

Range        (1,50000000).filter (\_ % 13==0)(1) // 26, 但很慢，需要大量内存

Stream.range(1,50000000).filter(\_%13==0)(1) // 26，很快，只计算最终结果需要的内容

注意：

第一个版本在filter后生成一个中间collection，size=50000000/13;而后者不生成此中间collection，只计算到26即可。

例子2：

(1 to 100).map(i=> i\*3+7).filter(i=> (i%10)==0).sum // map和filter生成两个中间collection

(1 to 100).toStream.map(i=> i\*3+7).filter(i=> (i%10)==0).sum

**5.2.1.7**     **Stack Queue**

先进后出的堆栈：

val s = collection.immutable.Stack()

val s2 = s.push(10,20,30) // Stack(30, 20, 10)

s2.head // 30

s2.pop.pop // Stack(10)

对应的可变Stack：

val ms = collection.mutable.Stack()

ms.push(1,3,5).push(7) // Stack(7, 5, 3, 1)

ms.head // 7

ms.pop // 7, ms = Stack(5,3,1)

先进先出的队列：

val q = collection.immutable.Queue() // 也可指定类型 Queue[Int]()

val q2 = q.enqueue(0).enqueue(List(10,20,30)) // Queue(0, 10, 20, 30)

q2.dequeue.\_1 // 0

q2.dequeue.\_2 // Queue(10, 20, 30)

对应的可变Queue：

val mq = collection.mutable.Queue[Int]()

mq += (1,3,5)

mq ++= List(7,9) // Queue(1, 3, 5, 7, 9)

mq dequeue // 1, mq= Queue(3, 5, 7, 9)

mq clear // Queue()

**5.2.2.**  **使用(map, flatMap, filter, exists等)**

**5.2.2.1**     **map**

// 类型可以混合：

import java.util.\_

val list3 = Array("a", 123, 3.14, new Date())

List("a","b","c").map(s=> s.toUpperCase()) // 方式1

List("a","b","c").map(\_.toUpperCase())     // 方式2, 类似于Groovy的it

// = List(A, B, C)

**5.2.2.2**     **filter filterNot**

List(1,2,3,4,5).filter(\_%2==0) // List(2, 4)

也可以写成：

for (x<-List(1,2,3,4,5) if x%2==0) yield x

List(1,2,3,4,5).filterNot(\_%2==0) // List(1, 3, 5)

**5.2.2.3**     **partition span splitAt groupBy**

注：val (a,b) = List(1,2,3,4,5).partition(\_%2==0) // (List(2,4), List(1,3,5))

可把Collection分成：满足条件的一组，其他的另一组。

和partition相似的是span，但有不同：

List(1,9,2,4,5).span(\_<3)       // (List(1),List(9, 2, 4, 5))，碰到不符合就结束

List(1,9,2,4,5).partition(\_<3) // (List(1, 2),List(9, 4, 5))，扫描所有

List(1,3,5,7,9) splitAt 2 // (List(1, 3),List(5, 7, 9))

List(1,3,5,7,9) groupBy (5<) // Map((true,List(7, 9)), (false,List(1, 3, 5)))

**5.2.2.4**     **foreach**

打印：

Array("a","b","c","d").foreach(printf("[%s].",\_))

// [a].[b].[c].[d].

**5.2.2.5**     **exists**

// 集合中是否存在符合条件的元素

List(1,2,3,4,5).exists(\_%3==0) // true

**5.2.2.6**     **find**

返回序列中符合条件的第一个。

例子：查找整数的第一个因子（最小因子、质数）

def fac1(n:Int) = if (n>= -1 && n<=1) n else (2 to n.abs) find (n%\_==0) get

**5.2.2.7**     **sorted sortWith sortBy**

例子（排序）：

List(1,3,2,0,5,9,7).sorted //  List(0, 1, 2, 3, 5, 7, 9)

List(1,3,2,0,5,9,7).sortWith(\_>\_) // List(9, 7, 5, 3, 2, 1, 0)

List("abc", "cb", "defe", "z").sortBy(\_.size) // List(z, cb, abc, defe)

List((1,'c'), (1,'b'), (2,'a')) .sortBy(\_.\_2) // List((2,a), (1,b), (1,c))

**5.2.2.8**     **distinct**

例子：（去除List中的重复元素）

def uniq[T](l:List[T]) = l.distinct

uniq(List(1,2,3,2,1)) // List(1,2,3)

**5.2.2.9**     **flatMap**

flatMap的作用：把多层次的数据结构“平面化”，并去除空元素（如None）。

可用于：得到xml等树形结构的所有节点名称，去除None等

例子1a：（两个List做乘法）

List(1,2,3) \* List(10,20,30) = List(10, 20, 30, 20, 40, 60, 30, 60, 90)

val (a,b) = (List(1,2,3), List(10,20,30))

a flatMap (i=> b map (j=> i\*j))

等同于：

for (i<-a; i<-b) yield i\*j // 这个写法更清晰

例子1b：

如果不用flatMap而是用map，结果就是：

a map (i=> b map (j=> i\*j)) // List(List(10, 20, 30), List(20, 40, 60), List(30, 60, 90))

等同于：

for (i<-a) yield { for (j<-b) yield i\*j } // 不如上面的清晰

例子2：

List("abc","def") flatMap (\_.toList) // List(a, b, c, d, e, f)

而

List("abc","def") map (\_.toList) // List(List(a, b, c), List(d, e, f))

例子3：flatMap结合Option

def toint(s:String) =

try { Some(Integer.parseInt(s)) } catch { case e:Exception => None }

List("123", "12a", "45") flatMap toint // List(123, 45)

List("123", "12a", "45") map toint // List(Some(123), None, Some(45))

**5.2.2.10**   **indices，zipWithIndex**

得到indices：

val a = List(100,200,300)

a indices // (0,1,2)

a zipWithIndex // ((100,0), (200,1), (300,2))

(a indices) zip a // ((0,100), (1,200), (2,300))

截取一部分,相当于String的substring

List(100,200,300,400,500) slice (2,4) // (300,400), 取l(2), l(3)

**5.2.2.11**   **take drop splitAt**

List(1,3,5,7) take 2 // List(1,3)

List(1,3,5,7) drop 2 // List(5,7)

**5.2.2.12**   **count**

满足条件的元素数目：

例如1000内质数的个数：

def prime(n:Int) = if (n<2) false else 2 to math.sqrt(n).toInt forall (n%\_!=0)

1 to 1000 count prime  // 168

**5.2.2.13**   **updated patch**

对于immutable的数据结构，使用updated返回一个新的copy：

val v1 = List(1,2,3,4)

v1.updated(3,10) // List(1, 2, 3, 10), v1还是List(1, 2, 3, 4)

对于可变的数据结构，直接更改：

val mseq = scala.collection.mutable.ArraySeq(1, 2, 4, 6)

mseq(3) = 10 // mseq = ArraySeq(1, 2, 4, 10)

批量替换，返回新的copy：

val v1 = List(1,2,3,4,5)

val v2 = List(10,20,30)

v1 patch (0, v2, 3) // List(10,20,30,4,5), 但v1,v2不变

**5.2.2.14**   **reverse reverseMap**

1 to 5 reverse // Range(5, 4, 3, 2, 1)

"james".reverse.reverse // "james"

reverseMap就是revese + map

  1 to 5 reverseMap (10\*) // Vector(50, 40, 30, 20, 10)

相当于：

  (1 to 5 reverse) map (10\*)

**5.2.2.15**   **contains startsWith endWith**

  1 to 5 contains 3 // true, 后一个参数是1个元素

  1 to 5 containsSlice (2 to 4) // true, 后一个参数是1个集合

(1 to 5) startsWith (1 to 3) // true 后一个参数是1个集合

(1 to 5) endsWith (4 to 5)

(List(1,2,3) corresponds List(4,5,6)) (\_<\_) // true，长度相同且每个对应项符合判断条件

**5.2.2.16**   **集合运算**

List(1,2,3,4) intersect List(4,3,6) // 交集 = List(3, 4)

List(1,2,3,4) diff List(4,3,6) // A-B = List(1, 2)

List(1,2,3,4) union List(4,3,6) // A+B = List(1, 2, 3, 4, 4, 3, 6)

// 相当于

List(1,2,3,4) ++ List(4,3,6) // A+B = List(1, 2, 3, 4, 4, 3, 6)

**5.2.2.17**   **殊途同归**

例子：得到 (4, 16, 36, 64, 100)

写法1：

(1 to 10) filter (\_%2==0) map (x=>x\*x)

写法2：

for(x<-1 to 10 if x%2==0) yield x\*x

写法3：

(1 to 10) collect { case x if x%2==0 => x\*x }

**5.2.2.18**   **其他**

对其他语言去重感兴趣，可看看：

<http://rosettacode.org/wiki/Remove_duplicate_elements>

**5.2.3.**  **数组元素定位**

统一使用()，而不是[]，()就是apply()的简写，a(i)===a.apply(i)

// Array

val a = Array(100,200,300) // a(0)=100, a(1)=200, a(3)=300

a(0) // 100, 相当于a.apply(0)

a(0)=10 // Array(10, 200, 300)，相当于a.update(0, 10)

// List

val list = List("a","b","c")

// list(0)=="a", list(1)=="b", list(2)=="c"

// Tuple

val t1 = ("a","b","c") // t1.\_1="a", t1.\_2="b", t1.\_3="c"

**5.2.4.**  **view**

在某类型的集合对象上调用view方法，得到相同类型的集合，但所有的transform函数都是lazy的，从该view返回调用force方法。

对比：

val v = Vector(1 to 10:\_\*)

v map (1+) map (2\*) // Vector(4, 6, 8, 10, 12, 14, 16, 18, 20, 22)

以上过程得生成2个新的Vector，而：

val v = Vector(1 to 10:\_\*)

v.view map (1+) map (2\*) force

只在过程中生成1个新的Vector，相当于：

v map (x=>2\*(1+x))

又如：

((1 to 1000000000) view).take(3).force // Vector(1,2,3)

使用Stream：

Stream.range(1,1000000000).take(3).force //  Stream(1, 2, 3)

**5.2.5.**  **和Java集合间的转换（scalaj）**

方案一：Java的List<T>很容易通过List.toArray转换到Array，和Scala中的Array是等价的，可使用map、filter等。

方案二：使用第三方的scalaj扩展包（需自行下载设置classpath）

例子1：

val a1 = new java.util.ArrayList[Int]

a1.add(100); a1.add(200); a1.add(300)

// 自行转换

val a2 = a1.toArray

a2 map (e=>e.asInstanceOf[Int]) map(2\*) filter (300>)

//采用scalaj(<http://github.com/scalaj/scalaj-collection>)

import scalaj.collection.Imports.\_

val a3 = a1.asScala

// scala->java

List(1, 2, 3).asJava

Map(1 -> "a", 2 -> "b", 3 -> "c").asJava

Set(1, 2, 3).asJava

// scalaj还可以在java的collection上使用foreach (目前除foreach外，还不支持filter、map)

a1.foreach(println)

scalaj的简易文档如下：

// Java to Scala

|  |  |
| --- | --- |
| **Java类型** | **转换方法** |
| java.lang.Comparable[A] | #asScala: scala.math.Ordered[A] |
| java.util.Comparator[A] | #asScala: scala.math.Ordering[A] |
| java.util.Enumeration[A] | #asScala: scala.collection.Iterator[A]  #foreach(A => Unit): Unit |
| java.util.Iterator[A] | #asScala: scala.collection.Iterator[A]  #foreach(A => Unit): Unit |
| java.lang.Iterable[A] | #asScala: scala.collection.Iterable[A]  #foreach(A => Unit): Unit |
| java.util.List[A] | #asScala: scala.collection.Seq[A]  #asScalaMutable: scala.collection.mutable.Seq[A] |
| java.util.Set[A] | #asScala: scala.collection.Set[A]  #asScalaMutable: scala.collection.mutable.Set[A] |
| java.util.Map[A, B] | #asScala: scala.collection.Map[A, B]  #asScalaMutable: scala.collection.mutable.Map[A, B]  #foreach(((A, B)) => Unit): Unit |
| java.util.Dictionary[A, B] | #asScala: scala.collection.mutable.Map[A, B]  #foreach(((A, B)) => Unit): Unit |

// Scala to Java

|  |  |
| --- | --- |
| **Scala类型** | **转换方法** |
| scala.math.Ordered[A] | #asJava: java.util.Comparable[A] |
| scala.math.Ordering[A] | #asJava: java.util.Comparator[A] |
| scala.collection.Iterator[A] | #asJava: java.util.Iterator[A]  #asJavaEnumeration: java.util.Enumeration[A] |
| scala.collection.Iterable[A] | #asJava: java.lang.Iterable[A] |
| scala.collection.Seq[A] | #asJava: java.util.List[A] |
| scala.collection.mutable.Seq[A] | #asJava: java.util.List[A] |
| scala.collection.mutable.Buffer[A] | #asJava: java.util.List[A] |
| scala.collection.Set[A] | #asJava: java.util.Set[A] |
| scala.collection.mutable.Set[A] | #asJava: java.util.Set[A] |
| scala.collection.Map[A, B] | #asJava: java.util.Map[A, B] |
| scala.collection.mutable.Map[A, B] | #asJava: java.util.Map[A, B]  #asJavaDictionary: java.util.Dictionary[A, B] |

**5.3.**     **Map**

**5.3.1.**  **定义Map**

var m = Map[Int, Int]()

var m = Map(1->100, 2->200)

或者

var m = Map((1,100), (2,200))

相加：

val m = Map(1->100, 2->200) ++ Map(3->300)  // Map((1,100), (2,200), (3,300))

可以用zip()生成Map：

List(1,2,3).zip(List(100,200,300)).toMap // Map((1,100), (2,200), (3,300))

*注解：zip有“拉拉链”的意思，就是把两排链扣完全对应扣合在一起，非常形象。*

**5.3.2.**  **不可变Map(缺省)**

  定义：

val m2 = Map()

val m3 = Map(1->100, 2->200, 3->300)

指定类型：

val m1:Map[Int,String] = Map(1->"a",2->"b")

注：如果import java.util.\_后发生冲突，可指明：scala.collection.immutable.Map

保持循序的Map可以使用：

collection.immutable.ListMap

  读取元素：

// m3(1)=100, m3(2)=200, m3(3)=300

// m3.get(1)=Some(100), m3.get(3)=Some(300), m3.get(4)=None

val v = m3.get(4).getOrElse(-1) // -1

或者简化成：

m3.getOrElse(4, -1) // -1

  增加、删除、更新：

Map本身不可改变，即使定义为var，update也是返回一个新的不可变Map：

var m4 = Map(1->100)

val m5 = m4(1)=1000 // m4还是 1->100, m5：1->1000

m4 += (2->200) // m4指向新的(1->100,2->200), (1->100)应该被回收

另一种更新方式：

m4.updated(1,1000)

增加多个元素：

Map(1->100,2->200) + (3->300, 4->400) // Map((1,100), (2,200), (3,300), (4,400))

删除元素：

  Map(1->100,2->200,3->300) - (2,3) // Map((1,100))

Map(1->100,2->200,3->300) -- List(2,3) // Map((1,100))

  合并Mpa：

Map(1->100,2->200) ++ Map(3->300) // Map((1,100), (2,200), (3,300))

**5.3.3.**  **可变Map**

val map = scala.collection.mutable.Map[String, Any]()

map("k1")=100     // 增加元素，方法1

map += "k2"->"v2" // 增加元素，方法2

// map("k2")=="v2", map.get("k2")==Some("v2"), map.get("k3")==None

有则取之，无则加之：

val mm = collection.mutable.Map(1->100,2->200,3->300)

mm getOrElseUpdate (3,-1) // 300, mm不变

mm getOrElseUpdate (4,-1) // 300, mm= Map((2,200), **(4,-1)**, (1,100), (3,300))

删除：

val mm = collection.mutable.Map(1->100,2->200,3->300)

mm -= 1 // Map((2,200), (3,300))

mm -= (2,3) // Map()

mm += (1->100,2->200,3->300) // Map((2,200), (1,100), (3,300))

mm --= List(1,2) // Map((3,300))

mm remove 1 // Some(300), mm=Map()

mm += (1->100,2->200,3->300)

mm.retain((x,y) => x>1) // mm = Map((2,200), (3,300))

mm.clearn // mm = Map()

改变value：

mm transform ((x,y)=> 0) // mm = Map((2,0), (1,0), (3,0))

mm transform ((x,y)=> x\*10) // Map((2,20), (1,10), (3,30))

mm transform ((x,y)=> y+3) // Map((2,23), (1,13), (3,33))

**5.3.4.**  **Java的HashMap**

使用Java的HashMap：

val m1:java.util.Map[Int, String] = new java.util.HashMap

**5.3.5.**  **读取所有元素**

上面说过，Map(1->100,2->200,3->300) 和 Map((1,100),(2,200),(3,300))的写法是一样的，可见Map中的每一个entry都是一个Tuple，所以：

for(e<-map) println(e.\_1 + ": " + e.\_2)

或者

map.foreach(e=>println(e.\_1 + ": " + e.\_2))

或者（**最好**）

for ((k,v)<-map) println(k + ": " + v)

也可以进行filter、map操作：

map filter (e=>e.\_1>1) // Map((2,200), (3,300))

map filterKeys (\_>1) // Map((2,200), (3,300))

map.map(e=>(e.\_1\*10, e.\_2)) // Map(10->100,20->200,30->300)

map map (e=>e.\_2) // List(100, 200, 300)

相当于：

map.values.toList

按照key来取对应的value值：

2 to 100 flatMap map.get // (200,300) 只有key=2，3有值

**5.3.6.**  **多值Map**

结合Map和Tuple，很容易实现一个key对应的value是组合值的数据结构：

val m = Map(1->("james",20), 2->("qh",30), 3->("qiu", 40))

m(2).\_1 // "qh"

m(2).\_2 // 30

for( (k,(v1,v2)) <- m ) printf("%d: (%s,%d)\n", k, v1, v2)

**5.4.**     **Set**

注：BitSet（collection.immutable.BitSet）和Set类似，但操作更快

**5.4.1.**  **定义**

var s = Set(1,2,3,4,5) // scala.collection.immutable.Set

var s2 = Set[Int]() // scala.collection.immutable.Set[Int]

// 增加元素：

s2 += 1  // Set(1)

s2 += 3  // Set(1,3)

s2 += (2,4) // Set(1,3,2,4)

// 删除元素

Set(1,2,3) - 2 // Set(1,3)

Set(1,2,3) - (1,2) // Set(3)

Set(1,2,3).empty // Set() 全部删除

// 判断是否包含某元素

s(3) // true, 集合中有元素3

s(0) // false, 集合中没有元素0

// 合并

Set(1,2,3) ++ Set(2,3,4) // Set(1, 2, 3, 4)

Set(1,2,3) -- Set(2,3,4) // Set(1)

**5.4.2.**  **逻辑运算**

|  |  |
| --- | --- |
| **运算** | **例子** |
| 交集 | Set(1,2,3) & Set(2,3,4) // Set(2,3)  Set(1,2,3) intersect Set(2,3,4) |
| 并集 | Set(1,2,3) | Set(2,3,4) // Set(1,2,3,4)  Set(1,2,3) union Set(2,3,4) // Set(1,2,3,4) |
| 差集 | Set(1,2,3) &~ Set(2,3,4) // Set(1)  Set(1,2,3) diff Set(2  ,3,4) // Set(1) |

**5.4.3.**  **可变BitSet**

val bs = collection.mutable.BitSet()

bs += (1,3,5) // BitSet(1, 5, 3)

bs ++= List(7,9) // BitSet(1, 9, 7, 5, 3)

bs.clear // BitSet()

**5.5.**     **Iterator**

Iterator不属于集合类型，只是逐个存取集合中元素的方法：

val it = Iterator(1,3,5,7) // Iterator[Int] = non-empty iterator

it foreach println // 1 3 5 7

it foreach println // 无输出

三种常用的使用模式：

// 1、使用while

val it = Iterator(1,3,5,7)

while(it.hasNext) println(it.next)

// 2、使用for

for(e<- Iterator(1,3,5,7)) println(e)

// 3、使用foreach

Iterator(1,3,5,7) foreach println

Iterator也可以使用map的方法：

Iterator(1,3,5,7) map (10\*) toList // List(10, 30, 50, 70)

Iterator(1,3,5,7) dropWhile (5>) toList // List(5,7)

由于Iterator用一次后就消失了，如果要用两次，需要toList或者使用duplicate：

val (a,b) = Iterator(1,3,5,7) duplicate // a = b = non-empty iterator

又如：

val it = Iterator(1,3,5,7)

val (a,b) = it duplicate

// 在使用a、b前，不能使用it，否则a、b都不可用了。

a toList // List(1,3,5,7)

b toList // List(1,3,5,7)

// 此时it也不可用了

**5.6.**     **Paralllel collection**

Scala 2.9+引入：

(1 to 10).par foreach println

多运行几次，注意打印顺序会有不同

6.   **io**

**6.1.**     **文件I/O**

**6.1.1.**  **读文件**

scala特有的是scala.io.Source，例如：

import scala.io.\_

Source.fromFile("cn.scala","utf8").mkString

逐行读文件内容：

Source.fromFile(new java.io.File("cn.scala")).getLines().foreach(println)

**6.1.2.**  **写文件**

直接调用java的io：

import java.io.\_, java.nio.channels.\_, java.nio.\_

// 写文件

val f = new FileOutputStream("o.txt").getChannel

f write ByteBuffer.wrap("a little bit long ...".getBytes)

f close

或者：

var out = new java.io.FileWriter("./out.txt") // FileWriter("./out.txt", true) 为追加模式

out.write("hello\n")

out close

**6.1.3.**  **复制文件**

直接调用java的io：

val in  = new FileInputStream("in").getChannel

val out = new FileOutputStream("out").getChannel

in transferTo (0, in.size, out)

**6.1.4.**  **全目录扫描**

递归使用listFiles：

  import java.io.File

  class Dir(file:File) {

    // 目录则返回所有1级子文件；文件则返回empty

def child = new Iterable[File] {

  // Iterable接口必须实现elements方法

      def elements = if (file.isDirectory) file.listFiles.elements else Iterator.empty

    }

    // 递归扫描，组成列表

    def scan:Iterable[File] = {

      Seq.singleton(file) ++ child.flatMap(i=> new Dir(i).scan)

    }

  }

  // 定义一个隐式类型转换

  implicit def foo(f:File) = new Dir(f)

  val file = new File(".")

  Console println new File(".").getCanonicalPath

  val list = file.scan // File对象隐形转换成Dir对象

  for (f<-list) println(f)

**6.2.**     **网络I/O**

[import](http://scala-lang.org/) java.net.{URL, URLEncoder}

[import](http://scala-lang.org/) scala.io.Source.fromURL

fromURL(new URL("http://qh.appspot.com")).mkString

或者指定编码：

fromURL(new URL("http://qh.appspot.com"))(io.Codec.UTF8).mkString

7.   **actor**

[http://www.scala-lang.org/docu/files/actors-api/actors\_api\_guide.html#](http://www.scala-lang.org/docu/files/actors-api/actors_api_guide.html)

Scala中处理并发，有很多选择：

  actor消息模型，类似Erlang，首选，Lift和akka也实现了自己的actor模型。

  Thread、Runnable

  java.util.concurennt

  3rd并发框架如Netty，Mina

**7.1.**     **actor模型**

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

**7.2.**     **多核计算**

对比如下的算法：

|  |  |
| --- | --- |
| def perfect(n:Int) =  n==(1 until n filter (n%\_==0) sum)  val n = 33550336  // 串行计算  n to n+10 foreach (i=>println(perfect(i))) | def perfect(n:Int) =  n==(1 until n filter (n%\_==0) sum)  val n = 33550336  // 并行计算  class T1(n:Int) extends Thread {    override def run(){println(perfect(n))}}  n to n+10 foreach (i=>new T1(i).start) |
| 耗时：8297 | 耗时：5134 |
| http://qiujj.com/static/Scala-Handbook.files/image012.jpg | http://qiujj.com/static/Scala-Handbook.files/image013.jpg |
| 单线程串行计算，不能很好发挥多核优势 | 多线程并行计算，平均分配到多核，更快 |

上面是Java的写法，也可以用Scala的actor写法：

Scala写法1：

import actors.Actor,actors.Actor.\_

class A1 extends Actor {

def act { react { case n:Int=>println(perfect(n)) }}}

n to n+10 foreach (i=>{ (new A1).start ! i})

Scala写法2：

val aa = Array.fill(11)(actor { react { case n:Int=>println(perfect(n)) }})

n to n+10 foreach (i=>aa(i-n) ! i)

或者：

n to n+10 foreach (i=> actor { react { case n:Int=>println(perfect(n)) }} ! i)

**7.3.**     **Actor用法**

Scala会建立一个线程池共所有Actor来使用。receive模型是Actor从池中取一个线程一直使用；react模型是Actor从池中取一个线程用完给其他Actor用

**实现方式1：**

import scala.actors.\_

object Actor1 extends Actor { // 或者class

// 实现线程

def act() { react { case \_ =>println("ok"); exit} }

}

//发送消息：

Actor1.start ! 1001 // 必须调用start

**实现方式2：**

import scala.actors.Actor.\_

val a2 = actor { react { case \_ =>println("ok") } } // 马上启动

//发送消息：

a2 ! "message" // 不必调用start

提示：

|  |  |
| --- | --- |
| ! | 发送异步消息，没有返回值。 |
| !? | 发送同步消息，等待返回值。（会阻塞发送消息语句所在的线程） |
| !! | 发送异步消息，返回值是 Future[Any]。 |
| ? | 不带参数。查看 mailbox 中的下一条消息。 |

**7.4.**     **方式1：接受receive**

特点：要反复处理消息，receive外层用while(..)

import actors.Actor, actors.Actor.\_

val a1 = Actor.actor {

var work = true

while(work) {

receive { // 接受消息, 或者用receiveWith(1000)

case msg:String => println("a1: " + msg)

case x:Int => work = false; println("a1 stop: " + x)

}

}

}

a1 ! "hello" // "a1: hello"

a1 ! "world" // "a1: world"

a1 ! -1 // "a1 stop: -1"

a1 ! "no response :("

**7.5.**     **方式2：接受react, loop**

特点：

  从不返回

  要反复执行消息处理，react外层用loop，不能用while(..);

  通过复用线程，比receive更高效，应尽可能使用react

import actors.Actor, Actor.\_

val a1 = Actor.actor {

react {

        case x:Int => println("a1 stop: " + x)

        case msg:String => println("a1: " + msg); act()

    }

}

a1 ! "hello" // "a1: hello"

a1 ! "world" // "a1: world"

a1 ! -1 // "a1 stop: -1"

a1 ! "no response :("

如果不用退出的线程，可使用loop改写如下：

val a1 = Actor.actor {

  loop {

react {

case x:Int => println("a1 stop: " + x); exit()

        case msg:String => println("a1: " + msg)

    }

  }

}

**7.6.**     **REPL接受消息**

scala> self ! "hello"

scala> self.receive { case x => x }

scala> self.receiveWithin(1000) { case x => x }

**7.7.**     **actor最佳实践**

**7.7.1.**  **不阻塞actor**

actor不应由于处理某条消息而阻塞，可以调用helper-actor处理耗时操作(helper actor虽然是阻塞的，但由于不接受消息所以没问题)，以便actor接着处理下一条消息

-----------------------------------------

import actors.\_, actors.Actor.\_

val time = 1000

  // （1）原来阻塞的程序

  val mainActor1 = actor {

    loop { react {

        case n: Int => Thread.sleep(time)

                         println(n)

        case s => println(s) } }

  }

  1 to 5 foreach { mainActor1 ! \_ } // 5秒钟后打印完数字

  // （2）改写由helper actor去阻塞的程序

  val mainActor2: Actor = actor {

    loop { react {

        case n: Int => actor { Thread.sleep(time); mainActor2 ! "wakeup" }

                        println(n)

        case s => println(s) } }

  }

  1 to 5 foreach { mainActor2 ! \_ } // 马上打印数字; 1秒钟后打印5个wakeup

-----------------------------------------

**7.7.2.**  **actor之间用且仅用消息来通讯**

actor模型让我们写多线程程序时只用关注各个独立的单线程程序（actor），他们之间通过消息来通讯。例如，如果BadActor中有一个GoodActor的引用:

class BadActor(a:GoodActor) extends Actor {...}

那在BadActor中即可以通过该引用来直接调用GoodActor的方法，也可以通过“!”来传递消息。选择后者！因为一旦BadActor通过引用读取GoodActor实例的私有数据，而这些数据可能正被其他线程改写值，结果就避免不了“共享数据-锁”模型中的麻烦事：即必须保证BadActor线程读取GoodActor的私有数据时，GoodActor线程在这块成为“共享数据”的操作上加锁。GoodActor只要有了共享数据，就必须来加锁防范竞用冲突和死锁，你又得从actor模型退回到“共享数据-锁”模型（注：actor对消息是顺序处理的，本来不用考虑共享数据）。

**7.7.3.**  **采用不可变消息**

Scala的actor模型让每个actor的act方法内部接近于单线程环境，你不用当心act方法里面的操作是否线程安全。在act方法中你可以尽情使用非同步、可变对象，因为每个act方法被有效限制在单个线程中，这也是actor模型被称为“share-nothing” 模型（零共享模型）的原因，其数据的作用范围被限制在单个线程中。不过一旦对象内的数据被用于多个actor之间进行消息传递。这时你就必须考虑消息对象是否线程安全。

保证消息对象线程安全的最好方法就是保证只使用不可变对象作为消息对象。**消息类中只定义val字段，且只能指向不可变对象**。定义这种不可变消息类的简单方法就是使用case class， 并保证其所有的val字段都是不可变的。Scala API中提供了很多不可变对象可用，例如基本类型、String、Tuple、List，不可变Set、不可变Map等。

如果你发现确实需要把一个可变对象obj1发送给其他actor，也因该是发送一份拷贝对象obj1.clone过去，而不是把obj1直接发过去。例如，数据对象Array是可变且未做同步的，所以Array只应该由一个actor同时存取，如果需要发送数组arr，就发送arr.clone（arr中的元素也应该是不可变对象），或者直接发送一个不可变对象arr.toList更好。

总结：大部分时候使用不可变对象很方便，不可变对象是并行系统的曙光，它们是易使用、低风险的线程安全对象。当你将来要设计一个和并行相关的程序时，无论是否使用actor，都应该尽量使用不可变的数据结构。

**7.7.4.**  **让消息自说明**

对每一种消息创建一个对应的case class，而不是使用上面的tuple数据结构。虽然这种包装在很多情况下并非必须，但该做法能使actor程序易于理解，例如：

// 不易理解，因为传递的是个一般的字符串，很难指出那个actor来响应这个消息

lookerUpper ! ("www.scala-lang.org", self)

// 改为如下，则指出只有react能处理LoopupIP的actor来处理：

case class LookupIP(hostname: String, requester: Actor)

lookerUpper ! LookupIP("www.scala-lang.org", self)

**7.8.**     **不同jvm间的消息访问**

服务器端：

object ActorServer extends Application {

    import actors.Actor, actors.Actor.\_, actors.remote.RemoteActor

    Actor.actor { // 创建并启动一个 actor

      // 当前 actor 监听的端口： 3000

      RemoteActor.alive(3000)

      // 在 3000 端口注册本 actor，取名为 server1。

      // 第一个参数为 actor 的标识，它以单引号开头，是 Scala 中的 Symbol 量，

      // Symbol 量和字符串相似，但 Symbol 相等是基于字符串比较的。

      // self 指代当前 actor （注意此处不能用 this）

      RemoteActor.register('server1, Actor.self)

      // 收到消息后的响应

      loop {

        Actor.react {case msg =>

          println("server1 get: " + msg)

        }

      }

    }

}

客户端：

object ActorClient extends Application {

    import actors.Actor, actors.remote.Node, actors.remote.RemoteActor

    Actor.actor {

      // 取得一个节点（ip:port 唯一标识一个节点）

      // Node 是个 case class，所以不需要 new

      val node = Node("127.0.0.1", 3000)

      // 取得节点对应的 actor 代理对象

      val remoteActor = RemoteActor.select(node, 'server1)

      // 现在 remoteActor 就和普通的 actor 一样，可以向它发送消息了！

      println("-- begin to send message")

      remoteActor ! "ActorClient的消息"

      println("-- end")

    }

}

**7.9.**     **STM**

<http://nbronson.github.com/scala-stm/>

a lightweight **Software Transactional Memory** for Scala, inspired by the STMs in Haskell and Clojure.

Cheat-Sheet:

**import** **scala.concurrent.stm.\_**

**val** x **=** **Ref**(0) // allocate a Ref[Int]

**val** y **=** **Ref**.make[String]() // type-specific default

**val** z **=** x.single // Ref.View[Int]

atomic { **implicit** txn **=>**

**val** i **=** x() // read

  y() **=** "x was " + i // write

**val** eq **=** atomic { **implicit** txn **=>** // nested atomic

    x() == z() // both Ref and Ref.View can be used inside atomic

  }

  assert(eq)

  y.set(y.get + ", long-form access")

}

// only Ref.View can be used outside atomic

println("y was '" + y.single() + "'")

println("z was " + z())

atomic { **implicit** txn **=>**

  y() **=** y() + ", first alternative"

**if** (x getWith { **\_** > 0 }) // read via a function

retry // try alternatives or block

} orAtomic { **implicit** txn **=>**

  y() **=** y() + ", second alternative"

}

**val** prev **=** z.swap(10) // atomic swap

**val** success **=** z.compareAndSet(10, 11) // atomic compare-and-set

z.transform { **\_** max 20 } // atomic transformation

**val** pre **=** y.single.getAndTransform { \_.toUpperCase }

**val** post **=** y.single.transformAndGet { \_.filterNot { **\_** == ' ' } }

8.   **misc**

**8.1.**     **xml**

**8.1.1.**  **生成**

Scala原生支持xml，就如同Java支持String一样，这就让生成xml和xhtml很简单优雅：

val name = "james"

val age = 10

val html = <html>name={name}, age="{age}"</html> toString

// <html>name=james, age=&quot;10&quot;</html>

又如：

val html = <html><head><title>{myTitle}</title></head><body>{"hello world"}</boty></html>

更复杂的例子：

val x = <r>{(1 to 5).map(i => <e>{i}</e>)}</r>

// <r><e>1</e><e>2</e><e>3</e><e>4</e><e>5</e></r>

val x0 = <users><user name="qh"/></users>

val <users>{u}</users> = x0  // u: scala.xml.Node = <user name="qh"></user>

**8.1.2.**  **xml文件**

xml.XML loadString "<p></p>"

xml.XML loadFile "abc.xml"

xml.XML.saveFull("foo.xml", node, "UTF-8", •

xmlDecl: Boolean, doctype : DocType)

**8.1.3.**  **读取：**

val x = <r>{(1 to 5).map(i => <e>{i}</e>)}</r>

// <r><e>1</e><e>2</e><e>3</e><e>4</e><e>5</e></r>

(x \ "e") map (\_.text.toInt) // List(1, 2, 3, 4, 5)

val x0 = <users>

<user name="qh"><age>20</age></user>

<user name="james"><age>30</age></user>

</users>

(x0 \ "user") // <user name="qh"><age>20</age></user>, <user name="james"><age>30</age></user>)

(x0 \ "user" \ "age") // (<age>20</age>, <age>30</age>)

(x0 \ "age")  // 直接下级: ()

(x0 \\ "age") // 所有下级：(<age>20</age>, <age>30</age>)

(x0 \ "\_") 所有

**8.1.4.**  **访问属性**

val x = <uu><u name="qh" /><u name="james" /><u name="qiu" /></uu>

(x \ "u" \\ "**@**name") foreach println // "qh\njames\nqiu"

例子：

val data =

<shopping>  
  <item name="bread" quantity="3" price="2.50"/>  
  <item name="milk" quantity="2" price="3.50"/>  
</shopping>  
  
val res = for (item <- data \ "item" ;   
                 price = (item \ "@price").text.toDouble ;   
                 qty = (item \ "@quantity").text.toInt)  
           yield (price \* qty)  
  
printf("$%.2f\n", res.sum)

**8.1.5.**  **格式化输出**

val pp = new xml.PrettyPrinter(80, 4)  // 行宽 80，缩进为 4

pp formatNodes <b><a/></b>

结果是字符串

<b>

    <a></a>

</b>

**8.2.**     **json**

Scala-json

**8.3.**     **Configgy**

<http://www.lag.net/configgy/>

简单配置及logging：

----------------------------

log {

  filename = "/var/log/pingd.log"

  roll = "daily"

  level = "debug"

}

hostname = "pingd.example.com"

port = 3000

----------------------------

val conf = net.lag.configgy.Configgy.configure("/etc/pingd.conf").config

val hostname = conf.getString("hostname", "localhost")

val port = conf.getInt("port", 3000)

**8.4.**     **正则表达式regex**

例子1：完全匹配

//将字符串转为scala.util.matching.Regex

val pattern = "^[a-z0-9.\_%\\-+]+@(?:[a-z0-9\\-]+\\.)+[a-z]{2,4}$"

val emailRegex = pattern.r // 或者 new scala.util.matching.Regex(pattern)

//emailRegex.pattern=>java.util.regex.Pattern 使用java的Pattern

emailRegex.pattern.matcher("tt@16.cn").matches // true

例子2：查找匹配部分

val p = "[0-9]+".r

p.findAllIn("2 ad 12ab ab21 23").toList // List(2, 12, 21, 23)

p.findFirstMatchIn("abc123xyz").get // scala.util.matching.Regex.Match = 123

更多例子如下：

**定义：**

[val](http://scala-lang.org/) r1 = "(\\d+) lines".r  // syntactic sugar

[val](http://scala-lang.org/) r2 = """(\d+) lines""".r  // using triple-quotes to preserve backslashes

或者

import scala.util.matching.Regex

[val](http://scala-lang.org/) r3 = [new](http://scala-lang.org/) Regex("(\\d+) lines")  // standard

[val](http://scala-lang.org/) r4 = [new](http://scala-lang.org/) Regex("""(\d+) lines""", "lines") // with named groups

**search和replace（java.lang.String的方法）：**

"99 lines" matches "(\\d+) lines" // true

"99 Lines" matches "(\\d+) lines" // false

"99 lines" replace ("99", "98") // "98 lines"

"99 lines lines" replaceAll ("l", "L") // 99 Lines Lines

**search（regex的方法）：**

"\\d+".r findFirstIn "99 lines" // Some(99)

"\\w+".r findAllIn "99 lines" // iterator(长度为2)

"\\s+".r findPrefixOf "99 lines" // None

"\\s+".r findPrefixOf "  99 lines" // Some(  )

[val](http://scala-lang.org/) r4 = [new](http://scala-lang.org/) Regex("""(\d+) lines""", "g1") // with named groups

r4 findFirstMatchIn "99 lines-a" // Some(99 lines)

r4 findPrefixMatchOf "99 lines-a" // Some(99 lines)

[val](http://scala-lang.org/) b = (r4 findFirstMatchIn "99 lines").get.group("g1") // "99"

**match（regex的方法）：**

[val](http://scala-lang.org/) r1 = "(\\d+) lines".r

[val](http://scala-lang.org/) r4 = [new](http://scala-lang.org/) Regex("""(\d+) lines""", "g1")

[val](http://scala-lang.org/) Some(b) = r4 findPrefixOf "99 lines" // "99 lines"

[for](http://scala-lang.org/) {

  line <- """|99 lines-a

             |99 lines

             |pass

             |98 lines-c""".stripMargin.lines

} line [match](http://scala-lang.org/) {

[case](http://scala-lang.org/) r1(n) => println("Has " + n + " Lines.") // "Has 99 Lines."

[case](http://scala-lang.org/) \_ =>

}

[for](http://scala-lang.org/) (matched <- "(\\w+)".r findAllIn "99 lines" matchData)

 println("Matched from " + matched.start + " to " + matched.end)

输出：

Matched from 0 to 2

Matched from 3 to 8

**replace（regex的方法）：**

[val](http://scala-lang.org/) r2 = """(\d+) lines""".r  // using triple-quotes to preserve backslashes

r2 replaceFirstIn ("99 lines-a", "aaa") // "aaa-a"

r2 replaceAllIn ("99 lines-a, 98 lines-b", "bbb") // "bbb-a, bbb-b"

其他：使用正则表达式定义变量

val regex = "(\\d+)/(\\d+)/(\\d+)".r

val regex(year, month, day) = "2010/1/13"

// year: String = 2010

// month: String = 1

// day: String = 13

**8.5.**     **GUI**

**8.5.1.**  **java方式**

[import](http://scala-lang.org/) javax.swing.JFrame

[var](http://scala-lang.org/) jf = [new](http://scala-lang.org/) JFrame("Hello!")

jf.setSize(800, 600)

jf.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE)

jf.setVisible([true](http://scala-lang.org/))

**8.5.2.**  **scala方式**

import swing.\_, swing.Swing.\_

object Swing1 extends SimpleGUIApplication { // scala2.7

  def top = new MainFrame { // 必须实现top方法

    title = "窗口1"

    preferredSize = (400, 300)

    val label = new Label("hello world cn中文")

    contents = new BorderPanel {

      layout(label) = BorderPanel.Position.Center

    }

  }

}

9.   **附录**

**9.1.**     **stackoverflow的scala教程**

解答一些实际问题：

<http://stackoverflow.com/tags/scala/info>

**9.2.**     **ProjectEuler**

数学+编程答题：

<http://projecteuler.net/index.php?section=problems>

对上面问题的Haskell解答参考：

<http://www.haskell.org/haskellwiki/Euler_problems>

对上面问题的Python解答参考：

<http://www.s-anand.net/euler.html>

对上面问题的Scala解答参考：

<http://pavelfatin.com/scala-for-project-euler/>

**9.3.**     **Scala问答**

<http://stackoverflow.com/questions/tagged/scala+list>

**9.4.**     **rosettacode**

[**http://rosettacode.org/wiki/Category:Scala**](http://rosettacode.org/wiki/Category:Scala)**（各种语言解题汇编）**

**9.5.**     **编译、mvn、SBT**

见“[Maven 学习笔记](file:///F:\data\__Apache\Maven\Maven%20%E5%AD%A6%E4%B9%A0%E7%AC%94%E8%AE%B0.doc)”中的4、SBT

Scala中类似于ant和maven的东西，配置文件使用.scala,交互模式下使用，可以自定编译和测试。

和vim/emacs/textmate等结合进行增量开发。

结合mvn和sbt，可以利用mvn eclipse:eclipse来产生.project和.classpath，利用sbt的不间断编译运行，步骤：

  运行maven命令（看mvnh.bat）创建目录./prj1/结构：

mvn archetype:create -DgroupId=jamesqiu -DartifactId=prj1 -DpackageName=test

                             -------            ----------        ----------

                      项目在仓库中的唯一标识   跟目录名及jar名        包名

  运行sbt

问及name就填和mvn的packageName一样（“test”）

问及organization就填和mvn的groupId一样（“jamesqiu”）

问及scala version就填2.8.0

  运行mvn eclipse:eclipse生成eclipse项目文件

  运行sbt

sbt ~compile 进行增量编译；

sbt ~run 进行增量编译运行，有多个可执行类时需要手工选择。

**9.6.**     **Scala水平划分**

来源：<http://www.scala-lang.org/node/8610> （Martin Odersky）

|  |  |
| --- | --- |
| **应用开发者** | **库开发者** |
| Level A1: Beginning application programmer           Java-like statements and expressions: standard operators, method calls, conditionals, loops, try/catch   * class, object, def, val, var, import, package * Infix notation for method calls * Simple closures * Collections with map, filter, etc            for-expressions |  |
| Level A2: Intermediate application programmer           Pattern matching           Trait composition           Recursion, in particular tail recursion           XML literals | Level L1: Junior library designer           Type parameters           Traits           Lazy vals           Control abstraction, currying           By-name parameters |
| Level A3: Expert application programmer           Folds, i.e. methods such as foldLeft, foldRight           Streams and other lazy data structures           Actors           Combinator parsers | Level L2: Senior library designer           Variance annotations           Existential types (e.g., to interface with Java wildcards)           Self type annotations and the cake pattern for dependency injection           Structural types (aka static duck typing)           Defining map/flatmap/withFilter for new kinds of for-expressions           Extractors |
|  | Level L3: Expert library designer           Early initializers           Abstract types           Implicit definitions           Higher-kinded types |

**9.7.**     **Scala适合的领域**

<http://goodstuff.im/scala-use-is-less-good-than-java-use-for-at-l> by David Pollak

小结：

  Scala更适合real-time, distributed, concurrent的应用，而不是ORM、CRUD应用（尽管有Squeryl）；

  大部分的应用还是ORM和CRUD，与其推销Scala到5万个项目里只成功20%，还不如推荐到5千个合适项目里成功80%；

  你的公司已高端人才为主，推荐Scala；以堆人力为主，还是Java吧；

**9.8.**     **Twitter的Scala School**

<http://twitter.github.com/scala_school/>

训练java expert快速成为scala expert