night_stalker 的 scala 杂记

笔记体。

杂记者，杂乱的记录也。

目 录

1. Scala

1.1 Scala 的杂记1：skah-la

发表时间: 2009-07-23

惊讶的发现念斯旮旯 ..... 不是 scale a .....

讲道理卖广告的多了，可惜没有实质意义，实作才是正确的方向 .....

请称呼我为 **hello world** 达人

```
args.foreach(println)
```

运行

```
scala s.scala hello world
```

打开 repl 练手，附感想。（进入是 scala，退出是:q ）

数据结构

（缺省是 immutable，某些单线程而且性能 critical 的某些极少数几乎遇不到的情况再考虑 import scala.collection.mutable.Xxx ）

```
// Array 没有 sort，谢谢 jonathan_zz 的指正
List(1)++List(4)
Array(1)++Array(4)
Array(61,62)+"abc"          // Array 可以和字符串相加产生字符串
                            // 但是 List 不行，另外 Array 不能被看做字符串 ...
(1::2::3::Nil).filter (3>)  // 这个我喜欢，和 haskell 差不多呦
(1::2::3::Nil).filter (_<2) // 不用 _ 就不行 - -
List(1,2):::List(3,4)       // trap：如果不小心少打一个冒号就傻蛋了，用++为妙

var m = Map("k"->"v")++Map("s"->"c")
m += ("u"->"v")
m("s")

var s = Set("doe","re")
```

```
s += "me"
s.contains("fa")
```

List（最佳实践：List 函数和递归可以完全取代 while 和 for 并且更强大）

```
l(2)
l.head;l.last
l.init;l.tail
l.count(3>_)
l.drop(2)
l.dropRight(2)

l.filter(_.length==4)
l.forall(_.endsWith("s")) // Ruby： l.all?
l.foreach(print)
l.map(_+"y")              // 深深喜欢上了下划线 ……
l.mkString(", ")          // 全世界都用 join 你偏要 mkString ……
l.reverse
l.sort(_(0).toLowerCase < _(1).toLowerCase)

l.isEmpty
l.length
l.exists(_=="yellow")
```

*** 枚举器可 toList

Tuple（(,) 的类型是 Tuple2，依此类推）

```
val t=(99,"bottles")
t._2 // Haskell： Snd t
```

让我很伤心的是 class 不能当 value 用。这时得用 object ……

泛型用 [] 而不是 <>，泛型好像可以用值参数？ 待验证。

类型，值得长篇大论大书特书的主题 ……

```
def f(x:Int, y:Int):Int = { ...
def f (x:Int) (y:Int) :Int = { ... // 看来还是上一行好点 。。。
def f (p: ty => boolean) (c:Int) :Array[ty] = ... // 看来上一行目的是 lambda 参数
def f:Unit = 3 // 返回 Unit 实例 【】
def f = 3      // 返回 Int  实例  3 -- 返回类型可以推
```

下面这几个用法都是不行的 ……

```
def f(x)=x        // × 为什么 haskell 能推 scala 不能推 ……
def f(x)=x+1      // ×
def f(x)=1        // ×
def f(x:Any)=1+x  // ×
```

Ruby 中检查一个类型响应 '+' 很简单： x.respond_to ? :+
Scala 用 **structual typing** 可以解决部分问题 ……（感谢 FX 的神奇代码）

```
def f(x:{def +(a:Int): Int}) = x + 1
class A{
  def +(i:Int)=i*2
}
f (new A)
```

def ，var ， val ， lazy val

FX 写道

```
val a = e的话，e是会求值的
lazy val a = e的话，e是等到第一次要使用a的值时才求值
def就是使用多少次都重新求值
var是变量
```

补充：

```
val a = 3
val a = 4  // ok
a = 2  // 重复赋值，出错
```

用得着区分 val 和 def 么 ..... 制造这么多词好郁闷 .....

lambda

```
val f = (_:Int) + 1
val f = (s:Int) => s + 1
```

def 非常微妙 .....

* 加不加括号的区别：

```
def f = {...}   // 调用不用加括号，但是不能当 lambda 用
def f() = {...} // 调用必须加括号，可以当 lambda 用
```

** 等不等的区别： 没等号相当于返回 Unit 类型。

*** 参数表是逗号分隔还是)(分割的区别： tuple 作参数和 curry 化的方法

**** curry 化还可以选择分组，相当赞的设计 ！

* 消灭 var 有好处 —— 引用透明

** 注意消灭了 var 还不够。因为某些 val 的内部状态可以改变。

*** val 可以重新定义，但不能改变值

Nil 和 null

```
scala> null
res14: Null = null
scala> Nil
res15: Nil.type = List()
```

curry，结合律的完美诠释

```
f(x,y) == f(x)(y) == (f(x))(y)
// 第一个 == 只对 lambda 有效，对 def 无效，def 是不是 curry 的就看参数表是 tuple 还是分离的括号系列
```

1.2 Scala 的杂记2：class 不是 object，你还得用 singleton object ……

发表时间: 2009-07-23

一点点编辑器友好的东西： \misc\scala-tool-support\vim

紧接上回

将 class 换成 object，就有了 singleton object。就如字面意义所说， singleton object 就是 singleton object ……

```
object ooxx{
  // ...
}
```

覆盖方法得加 override 关键字

```
class Complex(_re:Double, _im:Double) {
  def re = _re
  def im = _im
  override def toString() = "(" + _re + "," + _im + ")" // 不是 You can，是 You must ……
}
```

* 嗯 …… 父类是 abstract class 的时候 override 关键字变成可选了。
** abstract, private, extends 关键字用法(基本)同 java

case class 提供了打开 class 观察其内容的方法，远胜过繁琐低效的反射和 getter。

```
case class B() extends A  // 现在还不管，以后就要强制括号了
case class C(v:Int) extends A
```

* 生成实例： B() 和 new B 都可以。
** 对每个构造器参数都自动生成一个 getter 方法。
*** 对象含方法 equals() hashCode() toString()。

case 语法（case 语句之间可以用空格/分号/换行分隔）

```
o match {
  case a1 => 1
  case a2 => 2
  case _  => n // 下划线匹配任何东西
}
```

case class 和模式匹配范例（from scala tutor）

```
abstract class Tree
case class Sum (l:Tree, r:Tree) extends Tree
case class Var (n:String) extends Tree
case class Const (v:Int) extends Tree

type Env = String => Int // 类型别名，类似于 C 的 typedef

def eval (t:Tree, env:Env):Int = t match {
  case Sum (l, r) => eval (l, env) + eval (r, env)
  case Var (n)    => env (n)
  case Const (v)  => v
}

val t = Sum (Const (12), Var ("x"))
val env:Env = {case "x" => 3 ;case "y" => 5}
println ("" + eval (t, env))
```

* {case ... case ...} 是 (x => x match{case ... case ...}) 的糖。

traits 类似于 haskell 的 class，既可以看做类的类，也可以看做类的特性，还能看做抽象类来 extend。

范例（from scala tutor）：序关系（this that 真直白）

```
trait Ord {
  def < (that: Any): Boolean  // 仅声明，lazy 定义
  def <=(that: Any): Boolean = (this < that) || (this == that)
  def > (that: Any): Boolean = !(this <= that)
  def >=(that: Any): Boolean = !(this < that)
}
```

* 和 haskell 里面的 Ord class 没差别。可参看 Haskell 类型系统相关文章 和 如何放弃 OOP

** extend 了 Ord 的类，只需要定义 <，就拥有了包括 <= ，> ， >= 的所有方法。

*** trait 还可以当做 mixin （当有父类的时候）： A extends B with C

泛型（原文是个很酷的词： **Genericity**）

再次 copy and paste ……

```
// 定义
class Ref[T] {
private var contents: T = _  // 不初始化的东西是 abstract member，所以要用类型 T 对应的默认值（_）
def set(value: T) { contents = value }
def get: T = contents
}

// 使用
val v = new Ref[Int]
v.set 3
```

泛型和 trait 结合起来也会很 tricky，作者估计曾经是 C++ 的 TMP（模板元编程）的中毒者 ……

```
trait A[T] {...}
trait B[T <: A[T] {...} // <: 称作 plain bounds，< 体现了 subclass 这个情浓于水的血缘关系
```

* ruby 的继承语法也是用 < 呢

考虑一个农民工兄弟 java.io.File，它没有 extend 或者 with 过哪个 trait，怎么给他打上小标签，放到轰隆隆的工地上去呢？
view bounds 是对付那些我们没做过手脚的类型的利器。

```
trait Set[A <% T] // A 只要可以隐式转换到 T 就成立，不需要血缘关系
```

有上界 <:，也有下界 >:。上下界可以用于泛型参数，也可以用于 type 类型声明。

2 snippets

```
class B[T <: X >: Y]
class A extends B with C with D
```

下面这个**隐式参数**（implicit）的例子很有意思（implicit parameter有点像默认参数——如果指定了 implicit 关键字就无效，如果没指定，implicit 将生效并聪明地隐式转换到 implicit object，而且是灰常安全 —— 请称之为 checked 的弱类型！）

```
// 半群
abstract class SemiGroup[A] {
  def add(x: A, y: A): A
}
// 含幺半群
abstract class Monoid[A] extends SemiGroup[A] {
  def unit: A
}
// 当编译器遇到 implicit Monoid 时，暗示它：是这个东西哦，是这个东西哦
implicit object StringMonoid extends Monoid[String] {
  def add (x: String, y: String): String = x concat y
  def unit: String = ""
```

```
}
// 是那个东西哦，是那个东西哦
def sum[A](xs: List[A])(implicit m: Monoid[A]): A =
  if (xs.isEmpty) m.unit  // 这个 if else 的缩进表明作者吃过很多 banana
  else m.add (xs.head, sum (xs.tail))
// 使用
sum (List ("a","b","c"))
```

隐式转换可用于模拟 open class 的效果。几种隐式转换的用法总结：

```
// 方法 1：如上

// 方法 2：调用 new_method 时，自动用 ImConv 将 Klass 的对象转成子类再调用 ..
implicit def ImConv(o:Klass) = new Klass{
  def new_method ...
}

// 方法 2.5：有时不用或者不能子类化，new 一个 Object 就行了
implicit def ImConv(o:Klass) = new {
  def new_method ...
}

// 方法 3：令 Klass 可以转换到 WrapKlass
class WrapKlass(k:Klass) {
  def new_method ...
}
implicit def imConv(o:Klass) = new WrapKlass(Klass)
```

试了下 implicit class，出错：
`implicit' modifier can be used only for values, variables and methods.

方法 1、2 需要子类化，碰到 final class 就完蛋了。尝试一下 implicit object ss extends String{...} 出错：
illegal inheritance from final class String
想要对 final class 下手，就得包装而非子类化，即方法 2.5、3。

光暗示是不够的，**表白**（explicitly typed self references）也很重要！
http://www.scala-lang.org/node/124

```
class B extends C{
  self : A =>             // 本行表示 this 的类型是 A
  def abc = blabla(this)  // blabla 可能只接受 A 类型而不接受 B 类型
}
```

ETSR 的用处是：当 A 只声明了但还没定义而且下面的定义 B 也可以多态到 A 时，避免编译出错 ……

---

* 官网文档里这组 语言特性小文章 ，适合我们这些专吃蓝蓝路快餐的 ……

** 一点点代码风格的思考： like haskell，函数和参数之间留个空格大概会比较好？
see http://www.javaeye.com/problems/20678 （老庄大概会对馆里猿擅自把问题挪到问答频道感到很愤怒 …… ）。

*** 类型的其它 topics
协变、反变
forSome
Null 为可以成为 null 的类型
Nothing <: Null <: {ScalaObject, java类型} <: AnyRef <: Any
Nothing <: AnyVal <: Any
Type Erasure
Polymorphic Method Types

1.3 Scala 的杂记3：lift hello world

发表时间: 2009-07-25

lift 自从号称比 rails 快 4 倍后，最近又号称比 rails 快 6 倍，这是什么概念呢？
如果大家都不用 cache 不读数据库，scala 比 ruby 快 10 倍才对 ……
如果都用相同 cache 都连数据库，能快 20% 就恭喜了 —— 弄清楚 web 的瓶颈先。
再说部署惯例，apache 得比 lighttpd 慢多少 ……
这个宣传语太浮躁了。

lift 重启服务器耗时比 rails 慢很多，慢不止 6 倍 …… 修改代码后，不像 rails 那样能马上看到结果，沉重的硬伤啊！ 对快速开发影响太大了。

发完牢骚，lift 还是有它的优点的。
hello world 达人再次出动。
标准的复制自 wiki 的代码，不完全了解参数的意思 ……

```
mvn archetype:generate -U -DarchetypeGroupId=net.liftweb -DarchetypeArtifactId=lift-archetype-b
```

经过十几分钟到半小时后，helloworld 目录出现了
（可恶的 maven，各种下载 …… 幸好建好工程后可以把它换掉）

** 和 rails 比比： rails helloworld

转入 helloworld 目录，结构如下

```
src/
  main/
    resources/
    webapp/  # view 目录，其中的 html 用了 scala 的 schema，文件需要在 boot 配置
      index.html
    scala/
      bootstrap.liftweb/  # 我作了一点修改，用 . 分割的目录名减少嵌套
        boot.scala        # 启动类
      demo.helloworld/    # 同样做了修改
        comet/            # 这个是精髓，不过现在还没用到
```

```
        view/              # 骗人的目录 ……
        snippet/           # 相当于 rails 的 helper，scala 的 xml 语法，写起来挺好看
        model/
  test/   # 测试目录
target/   # 编译目标
pom.xml   # 月之女祭司
```

运行 mvn jetty:run， 再经过十几分钟到半小时的下载编译鼓捣，访问 localhost:8080 就能看到页面了，可喜可贺！

东西都下过一遍后，再次启动服务器需要的时间会大大减少。
lift 默认是不用连接数据库也能跑的，简单修改 Boot.scala ，包含数据库连接和一个新页面 hoho（hoho.html 请放置于 webapp 目录中。）

```
package bootstrap.liftweb

// 那些满屏 copy and paste import 的好好反省下吧！
import net.liftweb._
import util._,
       http._,
       sitemap._,
       sitemap.Loc._,
       mapper._

import Helpers._
import java.sql._

object connectionManager extends ConnectionManager {
  def newConnection(name : ConnectionIdentifier) = {
    try {
      Full(DriverManager.getConnection(
        "jdbc:mysql://localhost:3306/lift_hello_db",
        "root", "root"))
    } catch {
      case e : Exception => e.printStackTrace
```

```
      Empty
    }
  }
  def releaseConnection(conn : Connection) {
    conn.close
  }
}

class Boot {
  def boot {
    // where to search snippet
    LiftRules.addToPackages("demo.helloworld")

    // Build SiteMap
    val entries = List(
      Menu(Loc("home", List("index"), "go to home")),
      Menu(Loc("hoho", List("hoho"),  "go to hoho")))
    LiftRules.setSiteMap(SiteMap(entries:_*))

    // Connect DB
    DB.defineConnectionManager(DefaultConnectionIdentifier, connectionManager)
  }
}
```

代码参考了 tutor 的工程，构建时还得把 mysql 的驱动补上，明天继续干。

1.4 Scala 的杂记4：skcirT

发表时间: 2009-07-25

本篇收集一些想到或者碰到的技巧。

```
-----
```

更多的模式匹配

除了可以用 case class 进行匹配以外，还能利用 tuple 和 list 做匹配。

用 list 的情况相当于 case class 的构造函数匹配，同理对变参的 case class 都有效。

```
def ab = (1, 2)
val (a, b) = ab
```

```
def xxs = List(1, 2, 3)
val x::xs = xxs
val List(a,b,c) = xxs // 此时左右的个数要相等，否则会跳出一个 match error
```

或运算

```
case 1 | 2 | 3 => ...
```

用 _* 匹配 0 个到多个，产生的结果是 Seq[] 类型

```
val Array(a, b, _, _, c @ _*) = Array(1, 2, 3, 4, 5, 6, 7)
// a:Int = 1
// b:Int = 2
// c:Seq[Int] = Seq(5,6,7)
```

小心 trap：如果要匹配参数 y，必须用 `y`（用反单引号引起来）

```
def f(x:Int, y:Int) = x match {
  case 'y' => print("'y'")
  case y   => print("y")
}
f(1,1) // => 'y'
f(1,2) // => y
```

但是，如果这个量首字大写，就可以省略掉 `` ……

```
val Foo = 3
val Bar = 4
4 match {
  case Foo => "Foo"
  case Bar => "Bar"
}
// => "Bar"
```

使用模式匹配的函数例

```
val fac :Int => Int = {
  case 0 => 1
  case n => n * fac(n - 1)
}
```

xml 匹配

```
val <body>{ v }</body> = <body><oppai /></body>
//=> v: scala.xml.Node = <oppai></oppai>
```

还可以用 scala.xml.Elem 来匹配，其构造器是

```
case class Elem(val prefix: String,          // namespace prefix
                val label: String,           // (local) tag name
                val attributes: MetaData,
                val scope: NamespaceBinding, // namespace bindings
                val child: Node*) extends Node { ... }
```

使用模式匹配做简单的反射

```
o match {
  case that:A => ...
  case that:B => ...
}
```

case 语句的 guard

```
case ... if ... => ...
```

match 语句如果没有匹配，就会弹出 match error: x

```
-----
```

内部 import

在某个局部作用域内 import 也是可以的，此 import 不影响作用域外面，推荐。

import 还有很多花样，示例：

```
import java.util.{Set,Map}
import java.io.File, http._
```

```
-----
```

运算符

运算符分为 多元中缀，一元前缀，一元后缀。
任何方法都可以当成中缀或者后缀运算符使用。
普通方法当成后缀运算符时，可能会有词法解析的问题，建议使用时在行末加分号，并且在对象和运算符之间留一个空格。

和 haskell 一样，前缀运算符仅限于 + - ~ !
定义前缀运算符时，需要在定义的方法前加 unary_ ，例如：

```
class A{
  def unary_+ = 3
}
+ new A // res4: Int = 3
```

```
-----
```

Automatic Type-Dependent Closure Construction

在 杂记 2 中写过，单独的一个 {} 是可以看成 (x => {}) 的，这样就能实现类似于 ruby 的吊尾 block 和实现新的控制结构了。
譬如实现一个 unless 结构
（ from http://www.scala-lang.org/node/138 ）：

```
// 类型 【=> Boolean】表示任意返回 Boolean 的 lambda，注意冒号后面要空一格
def unless(cond: => Boolean)(body: => Any) =
  if (!(cond)) body
```

```
-----
```

解释执行不同于编译执行

第一个不同的地方，参看 杂记 1 里 FX 的回帖

脚本中的最外围空间是可以运行代码的，譬如 杂记 1 中的 hello world

scala 用作脚本语言的话，可能占用资源有点问题 ……
http://www.codecommit.com/blog/scala/scala-as-a-scripting-language

脚本的运行速度也比较慢（启动比 ruby 慢多了 ……）

非脚本中，最外围空间是不能进行表达式求值的（error: expected class or object definition）
主程序的结构得写成

```
object Xxx extends Application {
  // stuffs here ...
}
```

或者

```
object Xxx {
  def main(args:Array[String]) {
    // stuffs here ...
  }
}
```

编译与运行

```
scalac s.scala
scala -classpath . Xxx
```

* 如果要设置编译目标目录： scalac -d tmp
** extends Application 可能有些问题，参见

http://blogs.sun.com/navi/en_US/entry/scala_puzzlers_part_1

Application 后面这个 block 非常神奇，猜想 Application 接收一个 => Unit 类型的构造参数，Application 定义的 main 便是调用此参数。
试着写一个：

```
abstract class Application2(mian: => Unit) {
    def main(args:Array[String]) {
        mian
    }
}
object s extends Application2 {
    printf("hi")
}
```

```
-----
```

val 等语句的限制

ruby 和 javascript 可以这样：

```
a = b = 3
```

但是 scala 的定义语句和赋值语句不一样，而且赋值语句不返回值（或者说返回 Unit）
想到的写法如下 ..... 还不如不用连等 .....

```
val a = {val b = 3; b}
var c = 0
var d = 0
c = {d = 3; d}
```

```
var a=0, b=0 // error - -
```

补充：原来应该这样写 - -，谢谢 EastSun 同学 ~

```
val a, b = 0
```

```
-----
```

更多的 xml

```
xml.XML loadString "<p></p>"
xml.XML loadFile "abc.xml"
```

完整的保存到一个文件用 XML.saveFull，其原型是

```
saveFull(filename : java.lang.String, node : Node, enc : java.lang.String, xmlDecl : Boolean, c
```

```
-----
```

注释也得小心

/* ... */ 起作用优先于 //

而 /* ... */ 可以嵌套，嵌套时 /* 和 */ 必须匹配

所以下面几个都会出错

1.

```
/* comment
  /*
*/
```

2.

```
/* comment
//*/
```

3.

```
/* comment
// */ comment
*/
```

正确的是

1.嵌套匹配：

```
/* comment
  /*
  */
*/
```

2.加个空格，以免第二行被认成 /* ：

```
/* comment
// */
```

3.注意注释起作用的顺序：

```
/* comment
*/ // comment
```

最后，scala 的 logo：

```
//                      __
//     ________ ___    / /  ___
//    / __/ __// _ | / /  / _ |
//  __\ \/ /__/ __ |/ /__/ __ |
// /____/\___/_/ |_/____/_/ | |
//                          | /
```

1.5 Scala 的杂记5：for comprehension 等

发表时间: 2009-07-28

for comprehension 是一种 list comprehension。通过指定条件（for），产生（yield）的结果是一个 List。

```
for(val i <- List.range(1,5); val j <- List(3,5); i < 3; j > 2)
  yield i+j
// => List[Int] = List(4, 6, 5, 7)
```

从其他语言理解：

```
sql 或者 linq :
for   相当于 from
yield 相当于 select
<-    相当于 in
```

```
haskell :
<-    相当于 ∈
for (ooo) yield (xxx) 相当于 [xxx | ooo]
```

有一点比较囧：for 条件如果是多行，就不能用圆括号而要用花括号 ……

```
for(val i <- List.range(1,5)
  val j <- List(3,5)) // error
  yield i+j
for{val i <- List.range(1,5)
  val j <- List(3,5)} // ok
  yield i+j
```

scala 编译器会把 for comprehension 翻译成 map、flatMap、filter 的表达方式。

```
// 如
for (val x <- e; x > 3) yield x+2
// 会翻译成
e.filter(_>3).map(_+2)
```

如果不需要结果，可以用 for-loop

```
for(...){...}
```

Stream - lazy list

eager 的计算会带来一些效率问题，譬如寻找 1000...10000(不含10000) 中的第一个可以被 13 整除的数:

```
List.range(1000, 10000).filter(_ % 13 == 0)(0)
```

这里会先生成一个 List，然后找出 List 里面所有可被 13 整除的数，返回一个新 List，最后取出新 List 的第一个元素 —— 多余工作太多了。

为了达到 lazy 的计算效果（只有最终结果依赖的内容会被计算），就要用到 Stream 了：

```
Stream.range(1000, 10000).filter(_ % 13 == 0)(0)
```

Stream 的方法大致和 List 相同，只是
用 Stream.cons 代替 ::
用 Stream.append 代替 :::

Stream 是 lazy 的，做 filter 或者 map 之类的事情时，内容不会马上被计算，当要取出值或者 toList 的时候才开始计算，需要的内容计算完毕以后，马上停止迭代。

进程控制

开启新进程和调用 shell 命令比较繁琐 ..... 一旦阻塞，按 ctrl+break 都打不断，只能把 shell 杀掉。

```
val rt = Runtime.getRuntime
val pr = rt exec "ruby -v"
// java -version 和 scala -v 和 scala -help 不是返回非 0 值就是无法 read ……
// 只好先用 ruby 试手了 ……
pr waitFor // 等待执行结束
var buf = new Array[Byte](4096)
val len = pr.getInputStream read buf        // 不定长的还得写个 loop ……
val buf2 = Array range (0,len) map (buf(_)) // Array 的方法比较少，头疼 ……
new String(buf2)
//=> 终于看到结果了 ……
```

* 交互式解释器里 resXX 是可以直接用的

** 参考（这段是 GPL 的还不能随便拿去赚钱 - - ）：
http://github.com/litan/scrisca/blob/master/src/main/scala/com/kogics/scrisca/proc/Process.scala

1.6 Scala 的杂记6：actors

发表时间: 2009-07-28

Scala 除了模式匹配和强大的下划线以外，另一个闪光点就是并发模型 Actor。

Actor 是一种响应消息的对象，处理消息的时候，它可以：

- 改变自身状态
- 产生新的 Actor
- 发送新的消息

此过程可能是异步的，也可能是同步的。
和 Thread 类有点相似吧？ scala 的 actor 的实现里，有个私有类 ActorProxy 就是包装线程对象的。但是 Actor 比线程更细小：譬如产生两个 Actor，可能只需要 1 个线程。事实上 Scala 的 Actor 在初始化时，创建一个含 4 个 Thread 的线程池，如果所有线程都阻塞了，线程池会增长。

Actor 的历史和 message passing 和 process calculi 有些关系。（见 wiki）

message passing，或者说 messaging

C++ 和 java 与其说是面向对象语言，不如说是面向类的语言。
Alan Kay 说过：messaging 比 OO 更本质。
C++ 式的成员方法更像是一种语法糖和设计上的 guideline：obj.method() 结果是要被翻译成 method(obj) 的。
而在 messaging 的系统中，中间多了一个消息分发。obj.method() 意味着向 obj 发送 method 消息。

消息分发的系统更灵活，更具扩展性（譬如一些程序中可以在运行时添加新的消息响应），C++ 和 Java 没有内建的 messaging，于是是人们就想出了类似 signal/slot，event/listener 这些概念 —— 它们本质上都是一个 messaging 的系统。
消息分发的另一个优点就是： 消息可以发给多个对象，对象的响应也可以是异步的。缺点是分发时会有一些额外的开销。

process calculus

Erlang 的 process 大概就是从 process 代数中来的。 process 代数中，P 和 Q 并行，可以表达为 P|Q，[待续补完 ……]

actor

actor 的概念表达成三元组的话，就是 (message, mailbox, behavior)。scala 中的实现：

- message 一般可以设计成 case class（方便进行匹配）的实例；
- actor 已经内建了 mailbox（消息队列）；

- behavior 通过 receive 或 react 代码块来设定。

官网 tutor： http://www.scala-lang.org/node/242

基础的用法就像这样：

```
import scala.actors.Actor
class A extends Actor {
  // 实现抽象方法 act: =>Unit
  def act {
    ...
    receive {
      case Msg1 => ... // 接收到消息时的处理

// 使用
val a = new A
a.start
// 发送消息
a ! Msg1
```

不过 object Actor 中已经定义了工厂方法 actor()，打开交互式解释器：

```
import scala.actors.Actor._  // 此处导入的是 object Actor 的方法，而不是 class Actor
val a = actor{  receive{
    case 1 => print("received 1")
}}

a ! 2 // 没有定义 2，所以没有响应
      // 虽然和 case 没有匹配，但是不会弹出 not match 的异常
a ! 1 // => received 1 完事了
a ! 1 // 无响应
```

用 actor() 方法创建的 actor 是立即开始的，从 scala 的源代码中可以找到：

```
def actor(body: => Unit): Actor = {
    val actor = new Actor {
        def act() = body
    }
    actor.start()
    actor
}
```

链式响应和循环响应

```
actor{  react{
    case 1 => react{
        case 2 => print("received 1 and 2"); reply(3)
    }
}}
actor{  loop{  react{
    case 1 => ...
}}}
```

带参数的消息可以用 case class 实现，如下面的 Move(x, y)

```
case class Move(x:Int, y:Int)
val a = actor{
  react{
    case Move(x, y) => ...
  }
}
```

- ! 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。

例子：

```
a !? Move(3, 4)        // 注意没有 reply 的话会程序会死掉 ……
a !? (1000, M(3, 4))   // 等待 1000 毫秒等待应答，如果没有 reply，就终止等待
a.!? (1000, M(3, 4))   // 上一行的非操作符写法
```

- react 和 receive 的 block 里面引用本 actor 时不能用 this —— 要用 self 方法。
- react 和 receive 区别： react 是 threadless 的，但是不返回值。receive 会返回 case ... => ... 块的值，但会阻塞 actor self 所在的线程。react 外面不能套 while(true)，必须用 loop。
- 带有超时的版本： receiveWithin 和 reactWithin
- 注意 mutable 和 immutable，尽量避免会带来死锁和不稳定问题的 mutable 状态。

进一步的参考

* 避免 mutable state
http://www.hpl.hp.com/techreports/2009/HPL-2009-148.pdf

* scala api
http://scala-tools.org/scaladocs/scala-library/2.7.1/

* sbaz 是 scala 的包管理系统。通过 sbaz 安装本地文档：

```
sbaz install scala-devel-docs
```

在 %scala_home%\doc\scala-devel-docs\api 可以查看 api
在 %scala_home%\doc\scala-devel-docs\examples\actors 可以找到很多使用 actor 的例子

* channel 如何工作
http://en.wikipedia.org/wiki/Actor_model_and_process_calculi

* remote actor
http://youshottheinvisibleswordsman.co.uk/2009/04/01/remoteactor-in-scala/

* 比较详尽的理论描述（1986, Agha, 扫描版）：
http://dspace.mit.edu/bitstream/1721.1/6952/2/AITR-844.pdf

1.7 Scala 的杂记7：lift 基础

发表时间: 2009-07-29

续杂记 3

主要是 Exploring Lift 的笔记。

启动原理

web.xml 的配置（为了少打几个字，用了 yaml 表示 ……）

```
web-app:
  filter:
    filter-name: LiftFilter
    display-name: LiftFilter
    description: LiftFilter
    filter-class: net.liftweb.http.LiftFilter
  filter-mapping:
    filter-name: LiftFilter
    url-pattern: /*
```

假设你仅仅需要实现一个高性能的简单服务，根本用不着 lift，直接写一个 servlet filter，里面各种 actor 处理并发问题，在 web.xml 里面配上即可 ……

标准 import

```
import xml._

import net.liftweb.http._
import S._

import net.liftweb.util._
import Helpers._
```

net.liftweb.http 的成员

```
S          用于访问 request, response, cookie, session 等
SHtml      定义 HTML 生成函数
LiftRules  全局设置
```

bootstrap.liftweb.Boot 默认启动类
最简单的启动类就像这样：

```
class Boot {
  def boot = {
    LiftRules.addToPackages("your.package")
    // 相当于添加了以下几个包：
    //   your.package.view
    //   your.package.comet
    //   your.package.snippet
  }
}
```

lift app 的启动过程:
1: url 重写
2: 执行匹配 url 的分发函数
3: 如何找到 view
  a: 检查 LiftRules.viewDispatch RulesSeq
  b: 如果 a 没匹配，找 template
  c: 如果 b 没匹配，找 class

---

template

template 是 html、xhml、htm 或者没扩展名的文件。命名规则：

```
<path to template>[_<language>][.<suffix>]
```

其中 language 是按照请求的 accept-language 的规则寻找的

url /dir/tmpl.xhtml 仅对应同名同位置文件
url /dir/tmpl 对应 /dir/tmpl.xhtml, /dir/tmpl_es-ES.xhtml, ...

注意:
1: 摆在 template 根目录的文件不会起作用，如果需要，得手动配置。
2: /template-hidden 目录中的文件会自动隐藏（所有 -hidden 结尾的目录都会）
3: 就算 template 的扩展名是 html，都要写成严格的 xml，否则会 parse error(+﹏+)~
4: template 是动态编译的，修改了不用重启服务器

---

snippet 是 template 中的 lift tag，产生一个 xml 片段。
<lift:Hello /> == <lift:Hello.render />
== <lift:snippet type="Hello" /> == <lift:snippet type="Hello:render" />
* 比较长的那种写法，大概是在类名和 surround 等既定标签冲突时用的。

snippet 遵守递归的，从外到内渲染的规则（用 eager_eval 属性可以改变渲染顺序）

```
<lift:As.snip>
  这里可以访问 As 的 bind,如 <A:name />
  <lift:Bs.snip>
    这里可以访问 As 和 Bs 的 bind,如 <A:name /><B:name />
  </lift:Bs.snip>
</lift:As.snip>
```

相应的 snippet 定义：方法签名必须是 public (xml.NodeSeq) => xml.NodeSeq

```
class As {
  def snip (xhtml : NodeSeq) : NodeSeq =
    bind("A", xhtml, "name" -> Text("The A snippet"))
```

```
}
class Bs {
  def snip (xhtml : NodeSeq) : NodeSeq =
    bind("B", xhtml, "name" -> Text("The B snippet"))
  def snip2 (n: NodeSeq):NodeSeq = <oppai></oppai>
}
// 似乎 bind 和返回 xml 的没法放在同一个 snippet 中 ……
```

---

view 也可以拿来当 template（view 和 template 的关系就像 servlet 和 jsp 之间的关系一样）
lift 不是 MVC 框架，可以选择用 view 当 controller，用 template 当 view 来实现 MVC 结构 …… 不过 …… 不折腾了吧 ……
示例来自 exploring lift

```
class ExpenseView extends LiftView {
  override def dispatch = {
    case "enumerate" => doEnumerate _
  }
  def doEnumerate () : NodeSeq = {
    ...
    <lift:surround with="default" at="content">
      { expenseItems.toTable }
    </lift:surround>
  }
}
```

标签小汇

snippet:
<lift:snippet form="GET/POST" type="Class:method" multipart="true/false" />

<lift:Class.method form="..." multipart="..." />
<lift:Class form="..." multipart="..." />

surround:

<lift:surround with="template_name" at="binding">
children
</lift:surround>

* 默认 with = "/templates-hidden/default"
* surround 中的 <head> 标签会被 merge 进外头的 <head> 中。

bind:

<lift:bind name="binding" />

* 对应 surround

embed:

<lift:embed what="template_name" />

* 在一个 template 中包含另一个 template，可用于 Ajax。如果是 js 命令嵌入的 template，子 template 中的 js 不会执行。

comet:

<lift:comet type="ClassName" name="optional" />

* 在页面中加入一个 comet actor。name 代表这个 actor 的名字。

---

stateful snippet 是函数式的状态 snippet，比 session 更细粒度。stateful snippet 作用之一是实现多页表单。

```
class MySnippet extends StatefulSnippet {
  // dispatch 被用来分发请求，它是 var，所以处理请求时可以改变它
  val dispatch: DispatchIt = { case "forms" => first _ } // MySnippet.forms

  // 定义一些存储状态的变量
  var s = 0

  // 每个状态的处理
  def first (x: NodeSeq): NodeSeq = {
    dispatch = { case "forms" => second _ }
```

```
    s = s + 1
    <p>first. s = {s}</p>
  }
  def second (x: NodeSeq): NodeSeq = {
    dispatch = { case "forms" => third _ }
    s = s * 2
    <p>second. s = {s}</p>
  }
  // ... and so on
}
```

使用：

```
<lift:MySnippet.forms />
```

---

eager_eval ：让 tag 内容先运行，再运行 snippet 本身。尤其在 embed 的情况下有用（例子摘自 exploring lift）。

设 formTemplate 定义为

```
<hello:name />
<hello:time />
```

那么

```
<lift:Hello.world eager_eval="true">
  <lift:embed what="formTemplate" />
</lift:Hello.world>
```

第一步，eager eval 了内含的 embed 标签

```
<lift:Hello.world eager_eval="true">
  <hello:name />
  <hello:time />
</lift:Hello.world>
```

第二步，snippet 可以对 hello:name 和 hello:time 标签求值。

如果不是 eager_eval，最后就产生两个不能求值的标签了 ..... embed 和求值顺序尤其要注意，官网示例程序 PocketChange 里就有个页面求值顺序错了，结果显示了一堆 lift 标签 .....

---

url 重写

前面提到 LiftRules 可以配置几乎一切东西，自然包括 url 重写。请求 url 会先按 "/" 分隔，被解析成 List[String]，然后进入重写规则中。例子应该比较 self-explain 了 .....

```
LiftRules.rewrite.append {
  case RewriteRequest(
    ParsePath(List("account",acctName),_,_,_),_,_) =>
      RewriteResponse(List("viewAcct"), Map("name" -> acctName))
}
```

包括子域名的重写详解：
http://blog.getintheloop.eu/2009/5/3/url-rewriting-with-the-lift-framework

1.8 Scala 的杂记8：反射、bean 和 IO

发表时间: 2009-07-30

反射

scala 本身没有多少反射成分，基本只能用 java 的反射类，有点繁琐 ……

前面提到，简单的类型检查，用匹配即可:

```
case v: MyClass => ...
```

对于引用类型（AnyRef），obj.getClass 得到的是 java.utils.Class 类型的对象。
注意： Class.forName 对 scala 类无效；如果要从类型 Ty 出发获得 class 对象，用全局函数 classOf[Ty] ——对值类型也一样。

Class 的一些方法：

```
val k = obj getClass
k newInstance
k getDeclaredConstructors
k getDeclaredFields  //=> Array[java.lang.reflect.Field]
k getDeclaredMethods //=> Array[java.lang.reflect.Method]
                     // 有个默认方法 SomeClass.$tag
                     // val 成员对应 private 的 field 和 public getter
                     // var 成员对应 private 的 field 和 public (get/set)ter
k get<Constructors|Fields|Methods>
k isInstance obj     //=> true
```

在交互式 shell 中探索 Method：

```
import java.lang.reflect._
classOf[Method].getMethods.map(m => println(m.getName))
```

对于任意类型（Any 包括 AnyRef 和 AnyVal，AnyVal 包括 Boolean, Byte, Char, Double, Long, Float, Int, Short, Unit），有以下方法

```
isInstanceOf[T]
asInstanceOf[T]
```

进一步参考：JAM 的源码（JAM 是一个从 java 源文件或者类出发，用 velocity 生成 ActionScript 代码的工具）可以处理 .class 文件的类型信息
http://annogen.codehaus.org/JAM

Bean

考虑这个简单的 scala 类：

```
class A { var p = "p" }
```

生成的 java 类包含一个 private 字段及两个方法： p() 及 p_$eq()，如果是 val，则 setter 方法 p_$eq 是 private 的。

虽然这两个方法比 getXX setXX 好看一些 …… 但是如果想要和各种使用 bean 的东西（如 hibernate —— oh no ……）交互，可以用一个 annotation 生成 getP() 和 setP()：

```
class A {
  @reflect.BeanProperty
  var p = "p"
}
```

另一个方式是用 BeanInfo 注解 class， 注解的类里面，val 对应 reader，var 对应 r/w， def 对应 method。注意不能在 scala 中用 getP() 和 setP() ……

scala 的 reader 和 writer 的标准签名比 get/set 好一些 ……

```
p(): Ty
p_=(that: Ty): Unit // 和 ruby 一样的 xxx= 方法真的很难么 ……
```

但问题是 getter 不能和对应的变量重名 ……

IO

scala 专有的 IO 方法只有 scala.io.BufferedSource.fromInputStream：将 InputStream 转换成 Iterable。

日常 IO 与 java 基本无异 ……

```
import java.io._
import java.nio.channels._
import java.nio._
// 写文件
val f = new FileOutputStream("o.txt").getChannel
f write ByteBuffer.wrap("a little bit long ...".getBytes)
f close
// 复制文件
val in  = new FileInputStream("in").getChannel
val out = new FileOutputStream("out").getChannel
in transferTo (0, in.size, out)
```

关于 java IO 的一点备忘

* FileOutputStream, FileInputStream, RandomAccessFile 都有 getChannel 方法

** ByteBuffer

```
val buf = ByteBuffer allocate 4096
buf clear; in  read  buf // read  前要 clear
buf flip;  out write buf // write 前要 flip
```

*** MappedByteBuffer: 内存映射文件

```
val mbuf = someFileChannel map (FileChannel.MapMode.READ_WRITE, 0, 100100100)
```

1.9 Maven in Yaml

发表时间: 2009-07-31

醒醒吧，maven。
yaml 万岁！

原文需要翻 wall，就转了 ……
http://www.coderoshi.com/2007/08/maven-less-ugly.html

简单的 yaml 到 xml 的转换小工具，运行需要 ruby。

以下代码与原文略有不同，存为 y2x.rb。

引用

```
require 'yaml'

def convert(hash, tabs="  ")
  hash.each do |k,v|
    k = k.strip
    if v.class == Hash
      puts "#{tabs}<#{k}>"
      convert(v, tabs + "  ")
      puts "#{tabs}</#{k}>"
    elsif v.class == Array
      puts "#{tabs}<#{k}>"
      single_name = k =~ /ies$/ ? k.sub(/ies$/, 'y') : k.chop
      v.each do |i|
        if i.class == Hash
          puts "#{tabs}  <#{single_name}>"
          convert(i, tabs + "    ")
          puts "#{tabs}  </#{single_name}>"
        else
          puts "#{tabs}  <#{single_name}>#{i}</#{single_name}>"
        end
      end
```

```
      puts "#{tabs}</#{k}>"
    else
      puts "#{tabs}<#{k}>#{v}</#{k}>"
    end
  end
end

puts <<-PROJ
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_
  <modelVersion>4.0.0</modelVersion>
PROJ
convert YAML.load_file(ARGV[0])
puts "</project>"
```

用法：

```
ruby y2x.rb hello.yaml > pom.xml
```

例：给 lift hello world 改写的 hello.yaml

```
groupId: demo.helloworld
artifactId: helloworld
version: 1.0-SNAPSHOT
packaging: war
name: helloworld

repositories:
  - { id: scala-tools.org, name: scala-tools, url: "http://scala-tools.org/repo-releases" }

dependencies:
  - { groupId: org.scala-lang, artifactId: scala-library, version: 2.7.5 }
  - { groupId: org.scala-lang, artifactId: scala-compiler, version: 2.7.5, scope: test }
```

```
  - { groupId: net.liftweb, artifactId: lift-util, version: 1.0 }
  - { groupId: net.liftweb, artifactId: lift-webkit, version: 1.0 }
  - { groupId: net.liftweb, artifactId: lift-mapper, version: 1.0 }
  - { groupId: javax.servlet, artifactId: servlet-api, version: 2.5, scope: provided }
  - { groupId: junit, artifactId: junit, version: 4.5, scope: test }
  - { groupId: org.mortbay.jetty, artifactId: jetty, version: 6.1.6, scope: test }

build:
  sourceDirectory: src/main/scala
  testSourceDirectory: src/test/scala
  plugins:
    - groupId: org.scala-tools
      artifactId: maven-scala-plugin
      executions:
        - goals: [compile, testCompile]
      configuration:
        scalaVersion: 2.7.5
    - groupId: org.mortbay.jetty
      artifactId: maven-jetty-plugin
      configuration:
        contextPath: /
        scanIntervalSeconds: 5
    - groupId: net.sf.alchim
      artifactId: yuicompressor-maven-plugin
      executions:
        - goals: [compress]
      configuration:
        nosuffix: true
```

和生成的 xml 对比一下：

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.
  <modelVersion>4.0.0</modelVersion>
```

```
<groupId>demo.helloworld</groupId>
<artifactId>helloworld</artifactId>
<version>1.0-SNAPSHOT</version>
<packaging>war</packaging>
<name>helloworld</name>
<inceptionYear>2008</inceptionYear>
<repositories>
  <repository>
    <id>scala-tools.org</id>
    <name>scala-tools</name>
    <url>http://scala-tools.org/repo-releases</url>
  </repository>
</repositories>
<dependencies>
  <dependency>
    <groupId>org.scala-lang</groupId>
    <artifactId>scala-library</artifactId>
    <version>2.7.5</version>
  </dependency>
  <dependency>
    <groupId>org.scala-lang</groupId>
    <artifactId>scala-compiler</artifactId>
    <version>2.7.5</version>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>net.liftweb</groupId>
    <artifactId>lift-util</artifactId>
    <version>1.0</version>
  </dependency>
  <dependency>
    <groupId>net.liftweb</groupId>
    <artifactId>lift-webkit</artifactId>
    <version>1.0</version>
  </dependency>
  <dependency>
    <groupId>net.liftweb</groupId>
```

```
    <artifactId>lift-mapper</artifactId>
    <version>1.0</version>
  </dependency>
  <dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>servlet-api</artifactId>
    <version>2.5</version>
    <scope>provided</scope>
  </dependency>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.5</version>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>org.mortbay.jetty</groupId>
    <artifactId>jetty</artifactId>
    <version>6.1.6</version>
    <scope>test</scope>
  </dependency>
</dependencies>
<build>
  <sourceDirectory>src/main/scala</sourceDirectory>
  <testSourceDirectory>src/test/scala</testSourceDirectory>
  <plugins>
    <plugin>
      <groupId>org.scala-tools</groupId>
      <artifactId>maven-scala-plugin</artifactId>
      <executions>
        <execution>
          <goals>
            <goal>compile</goal>
            <goal>testCompile</goal>
          </goals>
        </execution>
      </executions>
```

```
        <configuration>
          <scalaVersion>2.7.5</scalaVersion>
        </configuration>
      </plugin>
      <plugin>
        <groupId>org.mortbay.jetty</groupId>
        <artifactId>maven-jetty-plugin</artifactId>
        <configuration>
          <contextPath>/</contextPath>
          <scanIntervalSeconds>5</scanIntervalSeconds>
        </configuration>
      </plugin>
      <plugin>
        <groupId>net.sf.alchim</groupId>
        <artifactId>yuicompressor-maven-plugin</artifactId>
        <executions>
          <execution>
            <goals>
              <goal>compress</goal>
            </goals>
          </execution>
        </executions>
        <configuration>
          <nosuffix>true</nosuffix>
        </configuration>
      </plugin>
    </plugins>
  </build>
</project>
```

ps: 看看我的文档里的 .m2 目录，又大了 ……