Effective Scala

Marius Eriksen, Twitter Inc.
marius@twitter.com (@marius)

[translated by hongjiang(@hongjiang)]

Table of Contents

其他语言

English 日本語 Русский

序言

Scala是Twitter使用的主要应用编程语言之一。很多我们的基础架构都是用scala写的，我们也有一些大的库支持我们使用。虽然非常有效，Scala也是一门大的语言,经验教会我们在实践中要非常小心。 它有什么陷阱？哪些特性我们应该拥抱，哪些应该避开？我们什么时候采用“纯函数式风格”，什么时候应该避免？换句话说：我们发现哪些可以高效的使用这门语言的地方？本指南试图把我们的经验提炼成短文，提供一系列最佳实践。我们使用scala主要创建一些大容量分布式系统服务——我们的建议也偏向于此——但这里的大多建议也应该自然的适用其他系统。这不是法律，但不当的使用应该被调整。

Scala提供很多工具使表达式可以很简洁。敲的少读的就少，读的少就能更快的读，因此简洁增强了代码的清晰。然而简洁也是一把钝器(blunt tool)也可能起到相反的效果：在考虑正确性之后，也要为读者着想。

首先，用scala编程，你不是在写java，haskell或python；scala程序不像这其中的任何一种。为了高效的使用语言，你必须用其术语表达你的问题。 强制把java程序转成scala程序是无用的，因为大多数情况下它会不如原来的。

这不是对scala的一篇介绍，我们假定读者熟悉这门语言。这儿有些学习scala的资源：

- Scala School
- Learning Scala
- Learning Scala in Small Bites

这是一篇“活的”文档，我们会更新它,以反映我们当前的最佳实践，但核心的思想不太可能会变： 永远重视可读性；写泛化的代码但不要在牺牲清晰度； 利用简单的语言特性的威力，但避免晦涩难懂（尤其是类型系统）。最重要的，总要意识到你所做的取舍。一门成熟的(sophisticated)语言需要复杂的实现，复杂性又产生了复杂性：推理，语义，特性之间的交互，以及与你合作者之间的理解。因此复杂性是为成熟所交的税——你必须确保效用超过它的成本。

玩的愉快。

格式化

代码格式化的规范 - 只要它们实用，并不重要。它的定义形式没有先天的好与坏，几乎每个人都有自己的偏好。然而，对于一致的应用采用同一格式化规则的总会增加可读性。已经熟悉某种特定风格的读者不必非要去掌握另一套当地习惯，或译解另一个角落里的语言语法。

这对scala来说也特别重要，因为它的语法高度的重叠。一个例子是方法调用：方法调用可以用“.”后边跟圆括号，或不使用“.”后边用空格加不带圆括号(针对空元或一元方法)方式调用。此外，不同风格的方法调用揭露了它们在语法上不同的分歧(ambiguities)。当然一致的应用慎重的选择一组格式化规则，对人和机器来说都会消除大量的歧义。

我们依着scala style guide 增加了以下规则：

空格

用两个空格缩紧。避免每行长度超过100列。在两个方法、类、对象定义之间使用一个空白行。

命名

对作用域较短的变量使用短名字：
    `is`, `js` 和 `ks`等可出现在循环中。
对作用域较长的变量使用长名字：
    外部APIs应该用长的，不需加以说明便可理解的名字。例如：`Future.collect` 而非 `Future.all`
使用通用的缩写，避开隐秘难懂的缩写：
    例如每个人都知道 `ok`,`err`, `defn`等缩写的意思，而`sfri`是不常用的。
不要重新绑定名字到不同的用处：
    `vals`(注：scala中的不可变类型)
避免用 `*s*`
    用`typ`替代 `` `type` ``
用*active*命名有副作用的操作：
    `user.activate()`而非 `user.setActive()`
对有返回值的方法用可描述的名字：
    `src.idDefined` 而非`src.defined`
*getters*不采用前缀`get`：
    用get是多余的: `site.count`而非`site.getCount`

不必重复名称在已经在*package*或*object*名称封装过的：

```
object User {
  def getUser(id: Int): Option[User]
}
object User {
  def get(id: Int): Option[User]
}
```

相对 get 方法 getUser 中的User是多余的，并不能提供额外的信息。

Imports

对*import*行按字母顺序排序：

这对视觉上的检查很方便，对自动操作也很简单。

当从一个包中引入多个时，用花括号：

```
import com.twitter.concurrent.{Broker, Offer}
```

当引入的超过*6*个，用通配符：

e.g.: import com.twitter.concurrent._

不要轻率的使用: 一些包导入了太多的名字

当引入集合的时候，用*import scala.collections.immutable(*不可变集合*)*或*scala.collections.mutable(*可变集合*)*限定名称可变和不可变集合有两个类名.限定名称让读者很明确知道使用的是哪个变量*(e.g. "mmutable.Map)"*

不要使用来自其它包的相对引用：

避免

```
import com.twitter
import concurrent
```

而应该用清晰的：

```
import com.twitter.concurrent
```

(译注，实际上面的import不能编译通过，第二个import应该为：import twitter.concurrent 即import一个包实际是定义了这个包的别名。)

将*import*放在文件的头部：

读者可以在一个地方参考所有的引用。

花括号

花括号用于创建复合表达式，复合表达式的返回值是最后一个表达式。避免对简单的表达式采用花括号；写成：

```
def square(x: Int) = x*x
```

而不是：

```
def square(x: Int) = {
  x * x
}
```

尽管它用在区分方法体的语句构成很诱人.第一种选择更少凌乱，更容易读。避免语句上的繁文缛节，除非需要阐明。

模式匹配

每当可应用的时候，直接在函数定义的地方使用模式匹配。例如，下面的写法match应该被折叠起来(collapse)

```
list map { item =>
  item match {
```

```
      case Some(x) => x
      case None => default
    }
  }
```

用下面的写法替代：

```
list map {
  case Some(x) => x
  case None => default
}
```

它很清晰的表达了 list中的元素都被映射，间接的方式让人不容易明白。

注释

使用ScalaDoc提供API文档。用下面的风格：

```
/**
 * ServiceBuilder builds services
 * ...
 */
```

而非标准的ScalaDoc风格：

```
/** ServiceBuilder builds services
 * ...
 */
```

不要诉诸于ASCII码艺术或其他可视化修饰。用文档记录APIs但不要添加不必要的注释。如果你发现你自己添加注释解释你的代码行为，先问问自己是否可以调整结构以让它明显的可以看出做了什么。相对于“it works, obviously” 更偏向于“obviously it works”

类型和泛型

类型系统的首要目的是检测程序错误，类型系统有效的提供了一个静态检测的有限形式，允许我们代码中明确某种类型的变量并且编译器可以验证。类型系统当然也提供了其他好处，但错误检测是他存在的理由(Raison d’Être)

我们使用类型系统反映这一目标，但读者需要留心：正确的使用类型可以增加清晰度，而过份聪明只会迷乱。

Scala的强大类型系统是学术探索和实践共同来源(eg.Type level programming in Scala) 。但这是一个迷人的学术话题,这些技术很少在应用和正式产品代码中使用。它们应该避免。

返回类型注释

Scala允许返回类型是可以省略的，而注释提供了很好的文档：这对public方法特别重要。而一个方法不需要对外暴露并且它的返回值类型是显而易见的，则可以直接省略。

这对用混入(mixin)实例化对象时很重要，scala编译器为这些创造了单类。例如：

```
trait Service
```

```
def make() = new Service {
  def getId = 123
}
```

上面的make不需要不需要定义返回类型为Service；编译器会创建一个加工过的类型: Object with Service{def getId:Int}. (译注:with是scala里的mixin的语法)而不必用一个显式的注释：

```
def make(): Service = new Service{}
```

现在作者不必改变make方法的公开类型而随意的混入(mix in) 更多的特质(traits)，使向后兼容很容易实现。

变型

变型(Variance)发生在泛型与子类型化(subtyping)结合的时候。与容器类型的子类型化有关，它们定义了对所包含的类型如何子类型化。因为scala有声明点变型(declaration site variance)注释，公共库的作者——特别是集合——必须有丰富的注释器。这些注释对共享代码的可用性很重要，但滥用也会很危险。

不可变(invariants)是scala类型系统中高级部分，但也是必须的一面，应该使用广泛的(并且正确的)，它有助于子类型化的应用。

不可变*(Immutable)*集合应该是协变的*(covariant)*。方法接受的类型应该适当的降级(downgrade)：

```
trait Collection[+T] {
  def add[U >: T](other: U): Collection[U]
}
```

可变*(mutable)*集合应该是不可变的*(invariant)*. 协变对于可变集合是典型无效的。考虑：

```
trait HashSet[+T] {
  def add[U >: T](item: U)
}
```

下面的类型层级：

```
trait Mammal
trait Dog extends Mammal
trait Cat extends Mammal
```

如果我现在有一个狗(dog)的 HashSet：

```
val dogs: HashSet[Dog]
```

当它为一个哺乳动物的Set，增加一只猫(cat)

```
val mammals: HashSet[Mammal] = dogs
mammals.add(new Cat{})
```

这将不再是一个只存储狗(dog)的HashSet！

类型别名

使用类型别名当它们提供了便捷的命名或阐明意图时，但对于自解释类型不要使

用类型别名。比如

```
() => Int
```

比下面定义的别名IntMarker更清晰

```
type IntMaker = () => Int
IntMaker
```

但，下面的别名:

```
class ConcurrentPool[K, V] {
  type Queue = ConcurrentLinkedQueue[V]
  type Map   = ConcurrentHashMap[K, Queue]
  ...
}
```

有助于交流的目的并使得更加简短。

当使用类型别名的时候不要使用子类型化(subtyping)

```
trait SocketFactory extends (SocketAddress => Socket)
```

SocketFactory 是一个生产Socket的方法。使用一个类型别名更好：

```
type SocketFactory = SocketAddress => Socket
```

我们现在可以对 SocketFactory类型的值 提供函数字面量(function literals) ,也可以使用函数组合：

```
val addrToInet: SocketAddress => Long
val inetToSocket: Long => Socket

val factory: SocketFactory = addrToInet andThen inetToSocket
```

类型别名通过用 package object 将名字绑定在顶层:

```
package com.twitter
package object net {
  type SocketFactory = (SocketAddress) => Socket
}
```

注意类型别名不是新类型——他们等价于在语法上的用别名代替了原类型。

隐式转换

隐式转换是类型系统里一个强大的功能，但应当谨慎的使用。它们有复杂的解决规则为难你——通过简单的词法检查——领会实际发生了什么。在下面的场景使用隐式转换是OK的：

- 扩展或增加一个scala风格的集合
- 适配或扩展一个对象(pimp my library模式）（译注参见：http://www.artima.com/weblogs/viewpost.jsp?thread=179766)
- 通过提供约束证据来加强类型安全。(Use to enhance type safety by providing constraint evidence)
- To provide type evidence (typeclassing，不知怎么翻译，haskell中的概念，主要通过隐式转换来实现)
- For Manifests (注：Manifest[T]包含类型T的运行时信息)

如果你发现自己在用隐式转换，总要问问自己是否不使用这种方式也可以达到目的。

不要使用隐式转换对两个相似的数据类型做自动转换(例如，把list转换为stream);显示的做更好，因为类型有不同的语意，读者应该意识到这种含义。 译注： 1）一些单词的意义不同，但翻译为中文时可能用的相似的词语，比如mutable，Immutable 这两个翻译为可变和不可变，它们是指数据的可变与不可变。 variance, invariant 也翻译为 可变和不可变，（variance也翻译为“变型”），它们是指类型的可变与不可变。variance指支持协变或逆变的类型，invariant则相反。

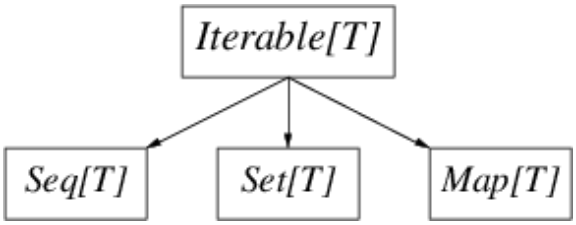
集合

Scala有一个非常通用，丰富，强大，可组合的集合库；集合是高阶的(high level)并暴露了一大套操作方法。很多集合的处理和转换可以被表达的简洁又可读，但粗心的用它的功能也导致相反的结果。每个scala程序员应该阅读 集合设计文档；通过它可以很好的洞察集合库，并了解设计动机。

总使用最简单的集合来满足你的需求

层级

集合库很大：除了精心设计的层级(Hierarchy)——根是 Traversable[T] —— 大多数集合都有不可变(immutable)和可变(mutable)两种变体。无论其复杂性，下面的图表包含了可变和不可变集合层级的重要差异。



Iterable[T] 是所有可遍历的集合，它提供了迭代的方法(foreach)。Seq[T] 是有序集合，Set[T]是数学上的集合(无序且不重复)，Map[T]是关联数组，也是无序的。

集合的使用

优先使用不可变集合.不可变集合适用于大多数情况，让程序易于理解和推断，因为它们是引用透明的(referentially transparent)因此缺省也是线程安全的。

使用可变集合时，明确的引用可变集合的命名空间。不要用使用import scala.collection.mutable._ 然后引用 Set ，应该用下面的方式替代：

```
import scala.collections.mutable
val set = mutable.Set()
```

这样更明确在使用一个可变集合。

使用集合类型缺省的构造函数。每当你需要一个有序的序列(不需要链表语义)，用 Seq() 等诸如此类的方法构造：

```
val seq = Seq(1, 2, 3)
val set = Set(1, 2, 3)
val map = Map(1 -> "one", 2 -> "two", 3 -> "three")
```

这种风格从语意上分离了集合与它的实现，让集合库使用更适当的类型：你需要Map，而不是必须一个红黑树(Red-Black Tree，注：红黑树TreeMap是Map的实现者)

此外，默认的构造函数通常使用专有的表达式，例如：Map() 将使用有3个成员的对象(专用的Map3类)来映射3个keys。

上面的推论是：在你自己的方法和构造函数里，适当的接受最宽泛的集合类型。通常可以归结为一个: Iterable, Seq, Set, 或 Map.如果你的方法需要一个 sequence，使用 Seq[T]，而不是List[T]

风格

函数式编程鼓励使用流水线转换将一个不可变的集合塑造为想要的结果。这常常会有非常简明的方案，但也容易迷糊读者——很难领悟作者的意图，或跟踪所有隐含的中间结果。例如，我们想要汇集不同的程序语言的投票从一组语言中(语言，票数)，按照得票的顺序显示：

```
val votes = Seq(("scala", 1), ("java", 4), ("scala", 10), ("scala", 1), ("python", 10))
val orderedVotes = votes
  .groupBy(_._1)
  .map { case (which, counts) =>
    (which, counts.foldLeft(0)(_ + _._2))
  }.toSeq
  .sortBy(_._2)
  .reverse
```

上面的代码简洁并且正确，但几乎每个读者都不好理解作者的原本意图。一个策略是声明中间结果和参数：

```
val votesByLang = votes groupBy { case (lang, _) => lang }
val sumByLang = votesByLang map { case (lang, counts) =>
  val countsOnly = counts map { case (_, count) => count }
  (lang, countsOnly.sum)
}
val orderedVotes = sumByLang.toSeq
  .sortBy { case (_, count) => count }
  .reverse
```

代码也同样简洁，但更清晰的表达了转换的发生(通过命名中间值)，和正在操作的数据的结构(通过命名参数)。如果你担心这种风格污染了命名空间，用大括号{}来将表达式分组:

```
val orderedVotes = {
  val votesByLang = ...
  ...
}
```

性能

高阶集合库（通常也伴随高阶构造)使推理性能更加困难：你越偏离直接指示计算机——即命令式风格——就越难准确预测一段代码的性能影响。然而推理正确性通常很容易；可读性也是加强的。在java运行时使用Scala使得情况更加复杂，Scala对你隐藏了装箱(boxing)/拆箱(unboxing)操作，可能引发严重的性能或内存空间问题。

在关注于低层次的细节之前,确保你使用的集合适合你。　确保你的数据结构没有不期望的渐进复杂度。各种scala集合的复杂性描述在这儿。

性能优化的第一条原则是理解你的应用为什么这么慢。不要使用空数据操作。在执行前分析[1]你的应用。关注的第一点是热循环(hot loops) 和大数据结构.过度关注优化通常是浪费精力。记住Knuth(高德纳)的格言：“过早优化是万恶之源”。

如果是需要更高性能或者空间效率的场景，通常更适合使用低级的集合。对大序列使用数组替代列表(List) (不可变Vector提供了一个指称透明的转换到数组的接口)，并考虑使用buffers替代直接序列的构造来提高性能。

*Java*集合

使用 scala.collection.JavaConverters 与java集合交互。有一系列的隐式的用于Java与Scala的转换。有助于读者，使用下面的方式确保转换是显式的：

```
import scala.collection.JavaConverters._

val list: java.util.List[Int] = Seq(1,2,3,4).asJava
val buffer: scala.collection.mutable.Buffer[Int] = list.asScala
```

并发

现代服务是高度并发的——　　服务器通常是在10–100秒内并列上千个同时操作——处理隐含的复杂性是创作健壮系统软件的中心主题。

*线程提供了一种表达并发的方式：它们给你独立的，堆共享的(heap-sharing)由操作系统调度的执行上下文。然而，在java里线程的创建是昂贵的，是一种必须托管的资源，通常借助于线程池。这对程序员创造了额外的复杂，也造成高度的耦合：很难从所使用的基础资源中分离应用逻辑。

这种复杂度尤其明显当创建高度分散(fan-out)的服务时：　每个输入请求导致一大批对另一层系统的请求。在这些系统中，线程池必须被托管以便根据每一层请求的比例来平衡：管理不善的线程池会渗入到另一个里(bleeds into another)。

一个健壮系统必须考虑超时和取消，两者都需要引入另一个“控制”线程，使问题更加复杂。注意若线程很廉价这些问题也将会被削弱：不再需要一个线程池，超时的线程将被丢弃，不再需要额外的资源管理。

因此，资源管理危害了模块。

Future

使用Future管理并发。它们将并发操作从资源管理里解耦出来：例如，Finagle(译注：twitter的一个框架)以有效的方式在少量线程上实现复用(multiplexes)。Scala有一个轻量级的闭包字面语法(literal syntax)，所以Futures引入了一些语法开销，它们成为很多程序员的老习惯(second nature)

Futures允许程序员用一种声明风格，可扩充的，有处理失败原则的，来表达并发计算。这些特性使我们相信它们尤其适合在函数式变成中用，这也是鼓励使用的风格。

*更愿意改造future为自己创建的。Future的转换(transformations)确保失败会传播，可以通过信号取消，对于程序员来说不必考虑java内存模型的含义。甚至一个仔细的程序员会写出下面的代码，顺序的发出10次RPC请求打印结果：

```
val p = new Promise[List[Result]]
var results: List[Result] = Nil
def collect() {
  doRpc() onSuccess { result =>
    results = result :: results
    if (results.length < 10)
      collect()
    else
      p.setValue(results)
  } onFailure { t =>
    p.setException(t)
  }
}

collect()
p onSuccess { results =>
  printf("Got results %s\n", results.mkString(", "))
}
```

程序员不得不确保RPC失败是可传播的，代码散布在控制流程中；糟糕的是，代码是错误的！ 没有声明results是volatile，我们不能确保results每次迭代会保持前一次值。Java内存模型是一个狡猾的野兽，幸好我们可以避开这些陷阱，通过用声明式风格(declarative style)：

```
def collect(results: List[Result] = Nil): Future[List[Result]] =
  doRpc() flatMap { result =>
    if (results.length < 9)
      collect(result :: results)
    else
      result :: results
  }

collect() onSuccess { results =>
  printf("Got results %s\n", results.mkString(", "))
}
```

我们用flatMap顺序的操作，把我们处理中的结果预追加(prepend)到list中。这是一个通用的函数式编程的习语的Futures译本。这是正确的，不仅需要的“咒语”(boilerplate)可以减少，易出错的可能性也会减少，并且读起来更好。

*Future组合子(combinators)的使用。当操作多个futures时，Future.select,Future.join,和Future.collect应该被组合编写出通用模式。

集合

并发集合的主题充满着意见、微妙(subtleties)、教条、恐惧/不确定/怀疑(FUD)。在大多实际场景都不存在问题：总是先用最简单,最无聊，最标准的集合解决问题。在你知道不能使用synchronized前不要去用一个并发集合：JVM有着复杂老练的手段来制造同步欺骗(synchronization cheap)，所以它的效率能让你惊讶。

如果一个不可变(immutable)集合可行，就尽可能用不可变集合——它们是指称透明的(referentially transparent)，所以它们用在并发上下文的理由是简单。不可变集合的改变通常用更新引用到当前值(一个var单元或一个AtomicReference)。必须小心正确的应用：原子型的(atomics)必须重试(retried)，变量(var类型的)必须声明为volatile以保证它们发布(published)到它们的线程。

可变的并发集合有着复杂的语义，利用java内存模型的微妙的一面，所以在你使用前确定你理解它的含义——尤其对于发布更新(新公开方法)。同步的集合同样写

起来更好：像getOrElseUpdate操作不能够被并发集合正确的实现，创建复合(composite)集合尤其容易出错。

控制结构

函数式风格的程序倾向于需要更少的传统的控制结构，并且使用声明式风格写的程序读起来更好。这通常意味着打破你的逻辑，拆分到若干个小的方法或函数，用用匹配表达式(match expression)把他们粘在一起。函数式程序也倾向于更多面向表达式(expression-oriented)：条件分支是同一类型的值计算，for表达式(for-comprehension)，递归都是司空见惯的。

递归

*用递归术语来表达你的问题常常会是简化的，如果应用了尾递归优化(可以通过@tailrec注释检测)，编译器甚至会将你的代码转换为正常的循环。对比一个标准的命令式版本的堆排序(fix-down):

```
def fixDown(heap: Array[T], m: Int, n: Int): Unit = {
  var k: Int = m
  while (n >= 2*k) {
    var j = 2*k
    if (j < n && heap(j) < heap(j + 1))
      j += 1
    if (heap(k) >= heap(j))
      return
    else {
      swap(heap, k, j)
      k = j
    }
  }
}
```

每次进入while循环，我们工作在前一次迭代时污染过的状态。每个变量的值是那一分支所进入函数，当找到正确的位置时会在循环中return。 (敏锐的读者会在Dijkstra的"Go To声明是有害的"一文找到相似的观点)

考虑尾递归的实现[2]:

```
@tailrec
final def fixDown(heap: Array[T], i: Int, j: Int) {
  if (j < i*2) return

  val m = if (j == i*2 || heap(2*i) < heap(2*i+1)) 2*i else 2*i + 1
  if (heap(m) < heap(i)) {
    swap(heap, i, m)
    fixDown(heap, m, j)
  }
}
```

每次迭代都是一个明确定义的清白历史的变量，并且没有引用单元：到处都是不变的(invariants)。更容易实现，也容易阅读。也没有性能方面的惩罚：既然方法是尾递归，编译器会转换为标准的命令式的循环。

返回*(Return)*

并不是说命令式结构没有价值。在很多例子中它们很适合于提前终止计算而非对每个可能终止的点存在一个条件分支：的确在上面的fixDown函数，一个return用于

提前终止如果我们已经在堆的结尾。

Returns可以用于切断分支建立不变量(establish invariants)。这帮助了读者减少了嵌套，并且容易实现后续的代码的正确性。这尤其适用于卫语句(guard clauses)：

```
def compare(a: AnyRef, b: AnyRef): Int = {
  if (a eq b)
    return 0

  val d = System.identityHashCode(a) compare System.identityHashCode(b)
  if (d != 0)
    return d

  // slow path..
}
```

使用return增加了可读性

```
def suffix(i: Int) = {
  if      (i == 1) return "st"
  else if (i == 2) return "nd"
  else if (i == 3) return "rd"
  else             return "th"
}
```

上面是针对命令式语言的，在scala中鼓励省略return

```
def suffix(i: Int) =
  if      (i == 1) "st"
  else if (i == 2) "nd"
  else if (i == 3) "rd"
  else             "th"
```

但使用模式匹配更好：

```
def suffix(i: Int) = i match {
  case 1 => "st"
  case 2 => "nd"
  case 3 => "rd"
  case _ => "th"
}
```

注意，return会有隐性成本：当在闭包内部使用时。

```
seq foreach { elem =>
  if (elem.isLast)
    return

  // process...
}
```

在字节码层实现时会包含一个异常的捕获/声明(catching/throwing)对，用在频繁的执行的代码中，会有性能影响。

*for*循环和*for*推导

for对循环和聚集提供了简洁和自然的表达。 它在扁平化(flattening)很多序列时特别有用。for语法通过分配和派发闭包隐藏了底层的机制。这会导致意外的开销和语义；例如：

```
for (item <- container) {
  if (item != 2) return
}
```

如果容器延迟计算(delays computation)会引起运行时错误，使返回不在本地上下文(making the return nonlocal)

因为这些原因，常常更可取的是直接调用foreach, flatMap, map和filter —— 在清楚的时候使用for。

require*和断言*(assert)

要求(require)和断言(assert)都起到可执行文档的作用。两者都在类型系统不能表达所求要的不变量(invariants)的场景里有用。 assert用于代码假设的不变量(invariants) 例如：(译注，不变量 invariant 是指类型不可变，即不支持协变或逆变的类型变量)

```
val stream = getClass.getResourceAsStream("someclassdata")
assert(stream != null)
```

相反，require用于表达API契约：

```
def fib(n: Int) = {
  require(n > 0)
  ...
}
```

函数式编程

value-oriented 编程有很多优势，特别是用在与函数式编程结构相结合。这种风格强调通过状态变化来转换values，生成的代码是指称透明的(referentially transparent)，提供了更强的不变型(invariants)，因此容易实现。Case类(也被翻译为样本类)，模式匹配，解构绑定(destructuring bindings)，类型推断，轻量级的闭包和方法创建语法都是这一行的工具。

***Case*类模拟代数数据类型**

Case类可模拟代数数据类型(ADT)编码：它们对大量的数据结构进行建模时有用，用强不变类型(invariants)提供了简洁的代码。尤其在结合模式匹配情况下。模式匹配实现了全面解析提供更强大的静态保护。 (译注：ADTs是Algebraic Data Type代数数据类型的缩写，关于这个概念见我的另一篇blog)

下面是用case类模拟代数数据类型的模式

```
sealed trait Tree[T]
case class Node[T](left: Tree[T], right: Tree[T]) extends Tree[T]
case class Leaf[T](value: T) extends Tree[T]
```

类型 Tree[T] 有两个构造函器：Node和Leaf。定义类型为sealed(封闭类)允许编译器彻底的分析(这是针对模式匹配的，参考Programming in Scala)因为构造器将不能从外部源文件中添加。

与模式匹配一同，这个建模使得代码简洁并且显然是正确的(obviously correct)

```
def findMin[T <: Ordered[T]](tree: Tree[T]) = tree match {
```

```
  case Node(left, right) => Seq(findMin(left), findMin(right)).min
  case Leaf(value) => value
}
```

一些递归结构，如树的组成是典型的ADTs(代数数据类型)应用，它们的用处领域更大。 disjoint,unions特别容易的用ADTs建模；这些频繁发生在状态机上(state machines)。

Options

Option类型是一个容器，空(None)或满(Some(value))二选一。它提供了使用null的另一种安全选择，应该尽可能的替代null。它是一个集合(最多只有一个元素)并用集合操所修饰，尽量用Option。

用 var username: Option[String] = None … username = Some(“foobar”)

代替

```
var username: String = null
...
username = "foobar"
```

前一种更安全：Option类型静态的强制username必须对空(emptyness)做检测。

对一个Option值做条件判断应该用foreach

```
if (opt.isDefined)
  operate(opt.get)
```

上面的代码应该用下面的方式替代：

```
opt foreach { value =>
  operate(value)}
```

风格看起来有些古怪，但提供了更多的安全和简洁。如果两种情况都有(Option的None或Some)，用模式匹配

```
opt match {
  case Some(value) => operate(value)
  case None => defaultAction()
}
```

如果都缺少缺省值，用getOrElse方法：

```
operate(opt getOrElse defaultValue)
```

不要过度使用Option： 如果有一个明确的缺省值——一个Null对象——直接用Null而不必用Option

Option还有一个方便的构造器用于包装空值(nullable value)

```
Option(getClass.getResourceAsStream("foo"))
```

得到一个 Option[InputStream] 假定空值(None)时getResourceAsStream会返回null

模式匹配

模式匹配(x match {...}) 在scala代码中无处不在：用于合并条件执行、解构(destructuring) 、在构造中造型。使用好模式匹配可以增加程序的明晰和安全。 使用模式匹配实现类型转换：

```
obj match {
  case str: String => ...
  case addr: SocketAddress => ...
```

模式匹配联合使用解构(destructuring)时效果最好（例如你要匹配case类）；下面的写法

```
animal match {
  case dog: Dog => "dog (%s)".format(dog.breed)
  case _ => animal.species
  }
```

应该被替代为：

```
animal match {
  case Dog(breed) => "dog (%s)".format(breed)
  case other => other.species
}
```

写自定义的抽取器(extractor)时必须有双重构造器，否则可能是不适合的。(译注：这句话没理解，可能作者是想说apply方法与unapply的对称性)

对条件执行不要用模式匹配，默认的方法更有意义。集合库的方法通常返回Options，避免：

```
val x = list match {
  case head :: _ => head
  case Nil => default
}
```

因为

```
val x = list.headOption getOrElse default
```

更短并且更能表达目的

偏函数

Scala提供了定义偏函数(PartialFunction)的语法快捷：

```
val pf: PartialFunction[Int, String] = {
  case i if i%2 == 0 => "even"
}
```

它们也可能由 orElse 组成:

```
val tf: (Int => String) = pf orElse { case _ => "odd"}

tf(1) == "odd"
tf(2) == "even"
```

偏函数出现在很多场景，并被有效的编码为 PartialFunction，例如 方法参数：

```
trait Publisher[T] {
  def subscribe(f: PartialFunction[T, Unit])
```

```
}

val publisher: Publisher[Int] = ..
publisher.subscribe {
  case i if isPrime(i) => println("found prime", i)
  case i if i%2 == 0 => count += 2
  /* ignore the rest */
}
```

或在返回一个Option的情况下：

```
// Attempt to classify the the throwable for logging.
type Classifier = Throwable => Option[java.util.logging.Level]
```

可以更好的用PartialFunction表达

```
type Classifier = PartialFunction[Throwable, java.util.Logging.Level]
```

因为它提供了更好的可组合性：

```
val classifier1: Classifier
val classifier2: Classifier

val classifier = classifier1 orElse classifier2 orElse { _ => java.util.Logging.Level.FINEST }
```

解构绑定

解构绑定与模式匹配有关,它们用了相同的机制，但可应用在当匹配只有一种选项的时候 (以免你接受异常的可能). 解构绑定特别适用于元组(tuple)和样本类(case class).

```
val tuple = ('a', 1)
val (char, digit) = tuple

val tweet = Tweet("just tweeting", Time.now)
val Tweet(text, timestamp) = tweet
```

惰性赋值

当使用lazy修饰一个val成员时，其赋值情况是在需要时才赋值的(by need) 因为scala中成员与方法是等价的（除了private[this]成员）

```
lazy val field = computation()
```

相当于下面的简写:

```
var _theField = None
def field = if (_theField.isDefined) _theField.get else {
  _theField = Some(computation())
  _theField.get
}
```

也就是说，它在需要时计算结果并会记住结果，在这种目的时使用lazy成员；但当语意上需要惰性赋值时(by semantics)，要避免使用惰性赋值，这种情况下，最好显式赋值因为它使得成本模型是明确的，并且副作用被严格的控制。

Lazy成员是线程安全的。

传名调用

方法参数可以指定为传名参数 (by-name) 意味着参数不是绑定到一个值，而是需要反复计算的。这一特性需要小心使用； 调用者按照传值(by-value)语法期待的话会感到惊讶。这一特性的动机是构造语法自然的 DSLs——使新的控制结构看起来更像本地语言特征。

只在下面的控制结构中使用传名调用， 调用者明显传递的是一段代码块(block)而非一个确定的计算结果。传名参数必须放在参数列表的最后一位。当使用传名调用时，确保方法名称让调用者明显的感知到方法参数是传名参数。

当你想要一个值被计算多次，特别是这个计算会引起副作用时，使用显式函数：

```
class SSLConnector(mkEngine: () => SSLEngine)
```

这样意图很明确，调用者不会感到惊奇。

flatMap

flatMap——结合了map 和 flatten —— 要特别小心，它有着难以琢磨的威力和强大的实用性。类似它的兄弟 map，它也是经常在非传统的集合中使用的，例如 Future , Option.它的行为由它的签名揭示；对于一些容器 Container[A]

```
flatMap[B](f: A => Container[B]): Container[B]
```

flatMap对集合中的每个元素调用了 函数 f 产生一个新的集合，将它们全部 flatten 后放入结果中

例如，获取两个字符的所有排列，相同的字符不能出现两次

```
val chars = 'a' to 'z'
val perms = chars flatMap { a =>
  chars flatMap { b =>
    if (a != b) Seq("%c%c".format(a, b))
    else Seq()
  }
}
```

等价于下面这段更简洁的 for-comprehension （基本就是针对上面的语法糖）

```
val perms = for {
  a <- chars
  b <- chars
  if a != b
} yield "%c%c".format(a, b)
```

flatMap在处理Options时频繁使用—— 它将options链合并为一个，

```
val host: Option[String] = ..
val port: Option[Int] = ..

val addr: Option[InetSocketAddress] =
  host flatMap { h =>
    port map { p =>
      new InetSocketAddress(h, p)
    }
  }
```

也可以使用更简洁的for来实现：

```
val addr: Option[InetSocketAddress] = for {
  h <- host
  p <- port
} yield new InetSocketAddress(h, p)
```

对flatMap在在Futures中的使用futures一节中有讨论。

面向对象的编程

scala的博大很大程度上在于它的对象系统。scala中所有的值都是对象，就这一意义而言scala是门纯粹的语言；基本类型和组合类型没有区别。scala也提供了mixin的特性允许更多正交的、细粒度的构造一些在编译时受益于静态类型检测的可被灵活组装的模块。　　mixin系统的背后动机之一是消除传统的依赖注入。这种“组件风格(component style)”编程的高潮是是the cake pattern.

依赖注入

在我们用来，发现scala本身删除了很多经典(构造函数)依赖注入的语法开销，我们更愿意就这样用:　它更清晰，依赖仍然通过类型编码，类构造语法是如此微不足道而变得轻而易举。有些无聊，简单，但有效。对模块化编程时使用依赖注入，特别是，组合优于继承—这使得程序更加模块化和可测试的。当遇到需要继承的情况，问问自己：在语言缺乏对继承支持的情况下如何构造程序？答案可能是令人信服的。

依赖注入典型的使用到 trait

```
trait TweetStream {
  def subscribe(f: Tweet => Unit)
}
class HosebirdStream extends TweetStream ...
class FileStream extends TweetStream ..

class TweetCounter(stream: TweetStream) {
  stream.subscribe { tweet => count += 1 }
}
```

这是常见的注入工厂 — 用于产生其他对象的对象。在这些例子中，更青睐用简单的函数而非专有的工厂类型。

```
class FilteredTweetCounter(mkStream: Filter => TweetStream) {
  mkStream(PublicTweets).subscribe { tweet => publicCount += 1 }
  mkStream(DMs).subscribe { tweet => dmCount += 1 }
}
```

Trait

依赖注入不妨碍使用公共接口，或在trait中实现公共代码。恰恰相反—正是因为以下原因而高度鼓励使用trait：一个具体的类可以实现多接口(traits)，公共的代码可以通过这些类复用。

保持traits简短并且是正交的：不要把分离的功能混在一个trait里，考虑将最小的相关的意图放在一起。例如，想象一下你要做一些IO的操作：

```
trait IOer {
```

```
  def write(bytes: Array[Byte])
  def read(n: Int): Array[Byte]
}
```

分离两个行为：

```
trait Reader {
  def read(n: Int): Array[Byte]
}
trait Writer {
  def write(bytes: Array[Byte])
}
```

可以将它们以混入(mix)的方式实现一个IOer : new Reader with Writer

接口最小化促使更好的正交和更清晰的模块化。

可见性

Scala有很丰富的可见性修饰。使用这些可见性修饰很重要，它们定义了哪些构成公开API。公开APIs应该限制，所以用户不会无意中依赖实现细节并限制了作者的能力来修改它们: 它们对于好的模块化设计是至关重要的。一般来说，扩展公开APIs比收缩公开的APIs容易的多。差劲的注释也能危害到你代码向后的二进制兼容性。

`private[this]`

一个类的成员标记为私有的，

```
private val x: Int = ...
```

它对这个类的所有实例来说都是可见的（但对其子类不可见）。大多情况，你想要的是 private[this] 。

```
private[this] val: Int = ..
```

这个修饰限制了它只对当前特定的实例可见。Scala编译器会把private[this]翻译为一个字段访问(因为访问仅限于静态定义的类)有时可增加性能优化。

单例类型

在Scala中创建单例类型是很常见的，例如：

```
def foo() = new Foo with Bar with Baz {
  ...
}
```

在这种情况下，可以通过声明返回类型来限制可见性：

```
def foo(): Foo with Bar = new Foo with Bar with Baz {
  ...
}
```

foo()方法的调用者会看到以有限视图的返回实例(Foo with Bar)

结构类型

不要在正常情况下使用结构类型。结构类型有着便利且强大的特性，但不幸的是在JVM上的实现不是很高效。　　然而——由于实现的怪癖——它提对执行反射(reflection)供了很好的速记。

```
val obj: AnyRef
obj.asInstanceOf[{def close()}].close()
```

垃圾回收

我们对生产模式的应用花了很多时间来调整垃圾回收。垃圾回收的关注点与java大致相似，尽管一些惯用的scala代码比起惯用的java代码会容易产生更多(短暂的)垃圾——函数式风格的副产品。Hotspot的分代垃圾收集通常使这不成问题，因为短暂的(short-lived)垃圾在大多情形下会被有效的释放掉。

在谈GC调优话题前，先看看这个由Attila举例说明我们在GC方面的一些经验的介绍。

scala固有的问题，你能够缓解GC的方法是产生更少的垃圾；但不要在没有数据的情况下行动。除非你做了某些明显的恶化。使用各种java的profiling工具——我们拥有的包括heapster和gcprof。

Java 兼容性

当我们写的scala代码被java调用时，我们要确保从java来用仍然习惯。这常常不需要额外的努力——class和trait明确的等价于java的中的对应类型 —— 但有时需要提供独立的Java Api。一种感受你的库中的java api好的方式是用java写单元测试(只是为了兼容性);这也确保了你的库中的java视图保持稳定，在这一点上不会随着时间因scala编译器的波动而影响。

包含部分实现的Trait不能直接被java使用： 改为 extends 一个抽象类

```
// 不能直接被java使用
trait Animal {
  def eat(other: Animal)
  def eatMany(animals: Seq[Animal) = animals foreach(eat(_))
}

// 改为这样:
abstract class JavaAnimal extends Animal
```

Twitter标准库

Twitter最重要的标准库是 Util 和 Finagle。Util 可以理解为Scala和Java的标准库扩展，提供了标准库中没有的功能或更合适的实现。Finagle 是我们的RPC系统，核心分布式系统组件。

Future

Futures已经在并发一节中简单讨论过。有一个中心机制来协调异步处理，渗透在我们代码库中，包括核心的Finagle。Futures允许组合并发事件，简化了高并发操作。有助于它们在JVM上高效的实现。

Twitter的future是异步的，所以基本上任何操作(阻塞操作)可以suspend它的线程的

执行；网络IO和磁盘IO是就是例子——必须由系统处理，它为结果提供future。Finagle为网络IO提供了这样一种系统。

Futue清晰简单：它们持有一个尚未完成运算结果的 promise 。它们是一个简单的容器——一个占位符。一次计算当然可能会失败，这种状况必须被编码：一个Future可以有3种状态： pending, failed, completed。

> 闲话：组合***(composition)***
>
> 让我们重新审视我们所说的组合：将简单的组件合成一个更复杂的。函数组合的一个权威的例子：给定函数 f 和 g，组合函数 (g□f)(x) = g(f(x)) ——结果先对 x使用f函数，然后在使用g函数——用scala来写：
>
> ```
> val f = (i: Int) => i.toString
> val g = (s: String) => s+s+s
> val h = g compose f  // : Int => String
>
> scala> h(123)
> res0: java.lang.String = 123123123
> ```
>
> 复合函数h，是个新的函数，由之前定义的f和g函数合成。

Future是一种集合类型——它是个包含0或1个元素的容器——你可以发现他们有标准的集合方法（eg：map, filter, foreach）。因为Future的值是延迟的，结果应用这些方法中的任何一种必然也延迟；在

```
val result: Future[Int]
val resultStr: Future[String] = result map { i => i.toString }
```

函数 { i => i.toString } 不会被调用，直到int值可用；转换集合的resultStr在可用之前也一直是待定状态。

List可以被 flatten：

```
val listOfList: List[List[Int]] = ..
val list: List[Int] = listOfList.flatten
```

这对future也是有意义的：

```
val futureOfFuture: Future[Future[Int]] = ..
val future: Future[Int] = futureOfFuture.flatten
```

因为future是延迟的，flatten的实现——立即返回——不得不返回一个等待外部future (Future[Future[Int]) 完成的future (Future[Future[Int]]).如果外部future失败，内部flattened future也将失败。

Future (类似List) 也定义了flatMap；Future[A] 定义方法flatMap的签名

```
flatMap[B](f: A => Future[B]): Future[B]
```

如同组合 map 和 flatten，我们可以这样实现：

```
def flatMap[B](f: A => Future[B]): Future[B] = {
  val mapped: Future[Future[B]] = this map f
  val flattened: Future[B] = mapped.flatten
  flattened
}
```

这是一种有威力的组合。使用flatMap我们可以定义一个 Future 是两个Future序列的结果。第二个future 的计算基于第一个的结果。想象我们需要2次RPC调用来验证一个用户身份，我们可以用下

面的方式组合操作：

```
def getUser(id: Int): Future[User]
def authenticate(user: User): Future[Boolean]

def isIdAuthed(id: Int): Future[Boolean] =
  getUser(id) flatMap { user => authenticate(user) }
```

这种组合类型的一个额外的好处是错误处理是内置的：如果getUser(..)或authenticate(..)没有额外的错误处理代码 future 从 isAuthred(..)返回时将会失败。

风格

Future回调方法(respond, onSuccess, onFailure, ensure) 返回一个新的链到它parent的Future,。这个Future被保证只有在它parent完成后才完成，使用模式如下：

```
acquireResource()
future onSuccess { value =>
  computeSomething(value)
} ensure {
  freeResource()
}
```

freeResource() 被保证只有在 computeSomething之后才执行， 允许模拟 try-finally 模式。

使用 onSuccess替代 foreach —— 它与 onFailure 方法对称，命名的意图更明确，并且也允许chaining。

永远避免直接创建Promise实例： 几乎每一个任务都可以通过使用预定义的组合子完成。这些组合子确保错误和取消是可传播的，通常鼓励的数据流风格的编程，不再需要同步和volatility声明。

用尾递归风格编写的代码不再遭受堆栈空间泄漏，允许以数据流风格高效的实现循环：

```
case class Node(parent: Option[Node], ...)
def getNode(id: Int): Future[Node] = ...

def getHierarchy(id: Int, nodes: List[Node] = Nil): Future[Node] =
  getNode(id) flatMap {
    case n@Node(Some(parent), ..) => getHierarchy(parent, n :: nodes)
    case n => Future.value((n :: nodes).reverse)
  }
```

Future定义很多有用的方法： 使用 Future.value() 和 Future.exception() 来创建未满意(pre-satisfied) 的future。Future.collect(), Future.join() 和 Future.select() 提供了组合子将多个future合成一个(例如：scatter-gather操作的gather部分)。

Cancellation

Future实现了一种弱形式的取消。调用Future#cancel 不会直接终止运算，而是发送某个级别的可被任何处理查询的触发信号，最终满足这个future。Cancellation信号流向相反的方向：一个由消费者设置的cancellation信号，传播到它的消费者。生产者使用 Promise的onCancellation来监听信号并执行相应的动作。

这意味这取消语意上依赖生产者，没有默认的实现。取消只是一个提示。

Local

Util的Local提供了一个位于特定的future派发树(dispatch tree)的引用单元(cell)。设定一个local的值，使这个值可以用于被同一个线程的Future 延迟的任何计算。有一些类似的线程locals，除了它们的范围不是一个java线程，而是一个 future 线程树。在

```
trait User {
  def name: String
  def incrCost(points: Int)
}
val user = new Local[User]

...
```

```
user() = currentUser
rpc() ensure {
  user().incrCost(10)
}
```

在 ensure块中的 user() 将在回调被添加的时候引用 user local的值。

就thread locals来说，Locals非常的方便，但要避免：确信通过显式传递数据时问题不能被充分的解决，甚至有些繁重的。

Locals有效的被核心库使用在非常常见的问题上——线程通过RPC跟踪，传播监视器，为future的回调创建stack traces——任何其他解决方法都使得用户负担过度。Locals在几乎任何其他情况下都不适合。

Offer/Broker

并发系统由于需要协调访问数据和资源而变得复杂。Actor提出一种简化的策略：每一个actor是一个连续的进程(process),保持自己的状态和资源,数据通过消息的方式与其它actor共享。 共享数据需要actor之间通信。

Offer/Broker 建立于这3个重要的方式上：1，通信通道(Brokers)是first class——即发送消息需要通过Brokers，而非直接到actor。2, Offer/Broker 是一种同步机制：通信会话是同步的。 这意味我们可以用 Broker做为协调机制：当进程a发送一条信息给进程b；a和b都要对系统状态达成一致。3, 最后，通信可以选择性的执行：一个进程可以提出几个不同的通信，其中的一个将被获取。

为了以一种通用的方式支持选择性通信（以及其他组合）,我们需要将通信的描述和执行解耦。这正是Offer做的——它是一个持久数据用于描述一次通信；为了执行这个通信（offer执行），我们通过它的sync()方法同步

```
trait Offer[T] {
  def sync(): Future[T]
}
```

返回 Future[T] 当通信被获取的时候生成交换值

Broker通过offer协调值的交换——它是通信的通道：

```
trait Broker[T] {
  def send(msg: T): Offer[Unit]
  val recv: Offer[T]
}
```

所以，当创建2个offer

```
val b: Broker[Int]
val sendOf = b.send(1)
val recvOf = b.recv
```

sendOf和recvOf都同步

```
// In process 1:
sendOf.sync()

// In process 2:
recvOf.sync()
```

两个offer都获取并且值1被交换。

可选择通信被通过Offer.choose绑定的多个offer执行。

```
def choose[T](ofs: Offer[T]*): Offer[T]
```

生成一个新的offer，当同步的获取特定的ofs——第一个可用。当多个都立即可用时，随机获取一个。

Offer对象有些一次性的Offers用于与来自Broker的Offer构建。

```
Offer.timeout(duration): Offer[Unit]
```

offer在给定时间后激活。Offer.never将用于不会有效，Offer.const(value)在给定值后立即有效。

通过选择性通信来组合是非常游泳的。例如，在一个send操作中使用超时：

```
Offer.choose(
  Offer.timeout(10.seconds),
  broker.send("my value")
).sync()
```

人们可能会比较 Offer/Broker 与SynchronousQueue，他们细节上不同而非重要的方面。Offer可以被组合，而queue不能。例如，考虑一组queues，描述为 Brokers:

```
val q0 = new Broker[Int]
val q1 = new Broker[Int]
val q2 = new Broker[Int]
```

现在让我们为reading创建一个合并的queue

```
val anyq: Offer[Int] = Offer.choose(q0.recv, q1.recv, q2.recv)
```

anyq是一个将从第一个可用的queue中读取的offer。注意 anyq 仍是同步的——我们仍然拥有底层队列的语义。这类组合是不可能用queue的。

例子：一个简单的连接池

连接池在网络应用中很常见，并且它们的实现常常需要技巧——例如，通常需要超时机制在从池中获取一个连接的时候，因为不同的客户端有不同的延迟需求。池的简单原则：维护一个连接队列，满足那些进入的等待者。使用传统的同步原语，这通常需要2个队列(queues)：一个用于等待者(当没有连接可用时)，一个用于连接(当没有等待者时)。

使用 Offer/Brokers ，可以表达的非常自然：

```
class Pool(conns: Seq[Conn]) {
  private[this] val waiters = new Broker[Conn]
  private[this] val returnConn = new Broker[Conn]

  val get: Offer[Conn] = waiters.recv
  def put(c: Conn) { returnConn ! c }

  private[this] def loop(connq: Queue[Conn]) {
    Offer.choose(
      if (connq.isEmpty) Offer.never else {
        val (head, rest) = connq.dequeue
        waiters.send(head) { _ => loop(rest) }
      },
      returnConn.recv { c => loop(connq enqueue c) }
    ).sync()
  }

  loop(Queue.empty ++ conns)
}
```

loop总是提供一个归还的连接，但只有queue非空的时候才会send。 使用持久化队列(persistent queue)更进一步简化逻辑。与连接池的接口也是通过Offer，所以调用者如果愿意设置timeout，他们可以通过利用组合子(combinators)来做：

```
val conn: Future[Option[Conn]] = Offer.choose(
  pool.get { conn => Some(conn) },
  Offer.timeout(1.second) { _ => None }
).sync()
```

实现timeout不需要额外的记账(bookkeeping)；这是因为Offer的语义：如果Offer.timeout被选择，不会再有offer从池中获得——连接池和它的调用者在各自waiter的broker上不必同时同意接受和发送。

埃拉托色尼筛子(Sieve of Eratosthenes 译注：一种用于筛选素数的算法)

在构造并发程序作为一组顺序的同步通信进程，它通常很有用——有时程序被大大的简化了。Offer和Broker提供了一组工具来让它简单并一致。确实，它们的应用超越了我们可能认为是经典并发性问题——并发编程(使有Offer/Broker辅助)是一种有用的构建工具，正如子例程(subroutines)，类，和模块都是——来自CSP的重要思想。

这里有一个埃拉托色尼筛子可以对一个整数流(stream of integers)构造为连续的应用过滤器 。首先，我们需要一个整数的源(source of integers)：

```
def integers(from: Int): Offer[Int] = {
  val b = new Broker[Int]
  def gen(n: Int): Unit = b.send(n).sync() ensure gen(n + 1)
  gen(from)
  b.recv
}
```

integers(n) 方法简单的提供了从n开始的所有连续的整数。然后我们需要一个过滤器：

```
def filter(in: Offer[Int], prime: Int): Offer[Int] = {
  val b = new Broker[Int]
  def loop() {
    in.sync() onSuccess { i =>
      if (i % prime != 0)
        b.send(i).sync() ensure loop()
      else
        loop()
    }
  }
  loop()

  b.recv
}

def sieve = {
  val b = new Broker[Int]
  def loop(of: Offer[Int]) {
    for (prime <- of.sync(); _ <- b.send(prime).sync())
      loop(filter(of, prime))
  }
  loop(integers(2))
  b.recv
}
```

loop() 工作很简单：从of中读取下一个质数(prime)，然后对of应用过滤器排除这个质数。loop不断的递归，持续的质数被过滤，于是我们得到了筛选结果。我们现在打印前10000个质数：

```
val primes = sieve
0 until 10000 foreach { _ =>
  println(primes.sync())
}
```

除了构造简单，组件正交，这种做法也给你一种流式筛子(streaming sieve)：你不需要先计算出你感兴趣的质数集合，进一步提高了模块化。

致谢

本课程由Twitter公司Scala社区贡献——我希望我是个忠实的记录者。

Blake Matheny, Nick Kallen, Steve Gury, 和Raghavendra Prabhu提供了很多有用的指导和许多优秀的建议。

---

1. Yourkit 是一个很好的profiler [back]
2. From Finagle’s heap balancer [back]