

Thinking in Scala

zhangyi

Published
with GitBook



目錄

1. 简介
2. 为何选择Scala
3. 函数式思想
4. Scala特性
 - i. Option与Null Object模式
5. 附录1：Scala编码规范

Thinking in Scala

记录我对Scala的思考，或许说零散的，但对于理解Scala会有极大裨益。

Scala有些特立独行，却又融合主要语言的风格，既是学院派，又烙着工程的印迹。它似乎想一统OO和FP的江山，因此在设计中有着诸多奇技淫巧，诸多妥协，诸多灵便的特性，复杂起来，会让人无所适从，强大起来，又让人爱不释手，赏心悦目。Scala于程序员而言，有些像唐末诗人李贺，奇诡飘忽，诗风偏向于晦涩朦胧而又精简，因此爱他之人膜拜到五体投地，厌他之人却又弃之如敝履。Scala遭人热捧，遭人冷眼，何尝不让人叹息其命运。

总算坚持住了。AKKA与Spark直接催生了Scala的大热，Twitter与Linkedin又在其中推波助澜，非但使得Scala站稳了脚跟，眼瞧着还有大步前进的可能。究竟如何，拭目以待。

为何选择Scala

Scala的亮点

简洁代码

Scala提供的脚本特性以及将函数作为一等公民的方式，使得它可以去掉不少在Java中显得冗余的代码，例如不必要的类定义，不必要的main函数声明。Scala提供的类型推断机制，也使得代码精简成为可能。Scala还有一个巧妙的设计，就是允许在定义类的同时定义该类的主构造函数。在大多数情况下，可以避免我们声明不必要的构造函数。

Scala还提供了一些非常有用的语法糖，如伴生对象，样例类，既简化了接口，也简化了我们需书写的代码。例如如下代码：

```
case class Person(name: String, age: Int)
val l = List(Person("Jack", 28), Person("Bruce", 30))
```

这里的List和Person都提供了伴生对象，避免再写冗余的new。这种方式对于DSL支持也是有帮助的。Person是一个样例类，虽然只有这么一行代码，蕴含的含义却非常丰富——它为Person提供了属性，属性对应的访问器，equals和hashCode方法，伴生对象，以及对模式匹配的支持。在Scala 2.11版本中，还突破了样例类属性个数的约束。由于样例类是不变的，也能实现trait，因而通常作为message而被广泛应用到系统中。例如在AKKA中，actor之间传递的消息都应该尽量定义为样例类。

支持OO与FP

将面向对象与函数式编程有机地结合，本身就是Martin Odersky以及Scala的目标。这二者的是非，我从来不予以置评。个人认为应针对不同场景，选择不同的设计思想。基于这样的思想，Scala成为我的所爱，也就是顺其自然的事情了。

演讲中，我主要提及了纯函数的定义，并介绍了应该如何设计没有副作用的纯函数。纯函数针对给定的输入，总是返回相同的输出，且没有任何副作用，就使得纯函数更容易推论（这意味着它更容易测试），更容易组合。从某种角度来讲，这样的设计指导思想与OO阵营中的CQS原则非常一致，只是重用的粒度不一样罢了。

我给出了Functional Programming in Scala一书中的例子。如下代码中的declareWinner函数并非纯函数：

```
object Game {
  def printWinner(p: Player): Unit =
    println(p.name + " is the winner!")

  def declareWinner(p1: Player, p2: Player): Unit =
    if (p1.score > p2.score)
      printWinner(p1)
    else printWinner(p2)
}
```

这里的printWinner要向控制台输出字符串，从而产生了副作用。（简单的判断标准是看函数的返回值是否为Unit）我们需要分离出专门返回winner的函数：

```
def winner(p1: Player, p2: Player): Player =
  if (p1.score > p2.score) p1 else p2
```

消除了副作用，函数的职责变得单一，我们就很容易对函数进行组合或重用了。除了可以打印winner之外，例如我们可以像

下面的代码那样获得List中最终的获胜者：

```
val players = List(Player("Sue", 7), Player("Bob", 8), Player("Joe", 4))
val finalWinner = players.reduceLeft(winner)
```

函数的抽象有时候需要脑洞大开，需要敏锐地去发现变化点与不变点，然后提炼出函数。例如，当我们定义了这样的List之后，比较sum与product的异同：

```
sealed trait MyList[+T]
case object Nil extends MyList[Nothing]
case class Cons[+T](h: T, t: MyList[T]) extends MyList[T]

object MyList {
  def sum(ints: MyList[Int]):Int = ints match {
    case Nil => 0
    case Cons(h, t) => h + sum(t)
  }

  def product(ds: MyList[Double]):Double = ds match {
    case Nil => 1.0
    case Cons(h, t) => h * product(t)
  }

  def apply[T](xs: T*):MyList[T] =
    if (xs.isEmpty) Nil
    else Cons(xs.head, apply(xs.tail: _*))
}
```

sum与product的相同之处都是针对List的元素进行运算，运算规律是计算两个元素，将结果与第三个元素进行计算，然后依次类推。这就是在函数式领域中非常常见的折叠（fold）计算：

```
def foldRight[A, B](l: MyList[A], z: B)(f: (A, B) => B):B = l match {
  case Nil => z
  case Cons(x, xs) => f(x, foldRight(xs, z)(f))
}
```

在引入了foldRight函数后，sum和product就可以重用foldRight了：

```
def sum(ints: MyList[Int]):Int = foldRight(ints, 0)(_ + _)
def product(ds: MyList[Double]):Double = foldRight(ds, 0.0)(_ * _)
```

在函数式编程的世界里，事实上大多数数据操作都可以抽象为filter，map，fold以及flatten几个操作。查看Scala的集合库，可以验证这个观点。虽然Scala集合提供了非常丰富的接口，但其实现基本上没有超出这四个操作的范围。

高阶函数

虽然Java 8引入了简洁的Lambda表达式，使得我们终于脱离了冗长而又多重嵌套的匿名类之苦，但就其本质，它实则还是接口，未能实现高阶函数，即未将函数视为一等公民，无法将函数作为方法参数或返回值。例如，在Java中，当我们需要定义一个能够接收lambda表达式的方法时，还需要声明形参为接口类型，Scala则省去了这个步骤：

```
def find(predicate: Person => Boolean)
```

结合Curry化，还可以对函数玩出如下的魔法：

```
def add(x: Int)(y: Int) = x + y
```

```
val addFor = add(2) _
val result = addFor(5)
```

表达式`add(2) _`返回的事实上是需要接受一个参数的函数，因此`addFor`变量的类型为函数。此时`result`的结果为7。

当然，从底层实现来看，Scala中的所有函数其实仍然是接口类型，可以说这种高阶函数仍然是语法糖。Scala之所以能让高阶函数显得如此自然，还在于它自己提供了基于JVM的编译器。

丰富的集合操作

虽然集合的多数操作都可以视为对`foreach`, `filter`, `map`, `fold`等操作的封装，但一个具有丰富API的集合库，却可以让开发人员更加高效。例如Twitter给出了如下的案例，要求从一组投票结果(语言, 票数)中统计不同程序语言的票数并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
```

这段代码首先将`Seq`按照语言类别进行分组。分组后得到一个`Map[String, Seq[(String, Int)]]`类型：

```
scala.collection.immutable.Map[String,Seq[(String, Int)]] = Map(scala -> List((scala,1), (scala,10), (scala,1)), java -> List((java,4)), python -> List((python,10)))
```

然后将这个类型转换为一个`Map`。转换时，通过`foldLeft`操作对前面`List`中`tuple`的`Int`值累加，所以得到的结果为：

```
scala.collection.immutable.Map[String,Int] = Map(scala -> 12, java -> 4, python -> 10)
```

之后，将`Map`转换为`Seq`，然后按照统计的数值降序排列，接着反转顺序即可。

显然，这些操作非常适用于数据处理场景。事实上，Spark的RDD也可以视为一种集合，提供了比Scala更加丰富的操作。此外，当我们需要编写这样的代码时，还可以在Scala提供的交互窗口下对算法进行`spike`，这是目前的Java所不具备的。

Stream

Stream与大数据集合操作的性能有关。由于函数式编程对不变性的要求，当我们操作集合时，都会产生一个新的集合，当集合元素较多时，会导致大量内存的消耗。例如如下的代码，除原来的集合外，还另外产生了三个临时的集合：

```
List(1,2,3,4).map(_ + 10).filter(_ % 2 == 0).map(_ * 3)
```

比较对集合的`while`操作，这是函数式操作的缺陷。虽可换以`while`来遍历集合，却又丢失了函数的高阶组合（`high-level composition`）优势。

解决之道就是采用`non-strictness`的集合。在Scala中，就是使用`stream`。关于这部分内容，我的同事崔鹏飞已有文章《[Scala中Stream的应用场景及其实现原理](#)》作了详细叙述。

并发与并行

Scala本身属于JVM语言，因此仍然支持Java的并发处理方式。若我们能遵循函数式编程思想，则建议有效运用Scala支持的并发特性。由于Scala在2.10版本中将原有的Actor取消，转而使用AKKA，所以我在演讲中并没有提及Actor。这是另外一个大的话题。

除了Actor，Scala中值得重视的并发特性就是Future与Promise。默认情况下，future和promise都是非阻塞的，通过提供回调的方式获得执行的结果。future提供了onComplete、onSuccess、onFailure回调。如下代码：

```
println("starting calculation ...")

val f = Future {
  sleep(Random.nextInt(500))
  42
}

println("before onComplete")
f.onComplete {
  case Success(value) => println(s"Got the callback, meaning = $value")
  case Failure(e) => e.printStackTrace()
}

// do the rest of your work
println("A ..."); sleep(100)
println("B ..."); sleep(100)
println("C ..."); sleep(100)
println("D ..."); sleep(100)
println("E ..."); sleep(100)
println("F ..."); sleep(100)

sleep(2000)
```

f的执行结果可能会在打印A到F的任何一个时间触发onComplete回调，以打印返回的结果。注意，这里的f是Future对象。

我们还可以利用for表达式组合多个future，AKKA中的ask模式也经常采用这种方式：

```
object Cloud {
  def runAlgorithm(times: Int): Future[Int] = Future {
    Thread.sleep(times)
    times
  }
}

object CloudApp extends App {
  val result1 = Cloud.runAlgorithm(10) //假设runAlgorithm需要耗费较长时间
  val result2 = Cloud.runAlgorithm(20)
  val result3 = Cloud.runAlgorithm(30)

  val result = for {
    r1 <- result1
    r2 <- result2
    r3 <- result3
  } yield (r1 + r2 + r3)

  result onSuccess {
    case result => println(s"total = $result")
  }

  Thread.sleep(2000)
}
```

这个例子会并行的执行三个操作，最终需要的时间取决于耗时最长的操作。注意，yield返回的仍然是一个future对象，它持有三个future结果的和。

promise相当于是future的工厂，只是比单纯地创建future具有更强的功能。这里不再详细介绍。

Scala提供了非常丰富的并行集合，它的核心抽象是splitter与combiner，前者负责分解，后者就像builder那样将拆分的集合再进行合并。在Scala中，几乎每个集合都对应定义了并行集合。多数情况下，可以调用集合的par方法来创建。

例如，我们需要抓取两个网站的内容并显示：

```
val urls = List("http://scala-lang.org",
               "http://agiledon.github.com")

def fromURL(url: String) = scala.io.Source.fromURL(url).getLines().mkString("\n")

val t = System.currentTimeMillis()
urls.par.map(fromURL(_))
println
println("time: " + (System.currentTimeMillis - t) + "ms")
```

如果没有添加par方法，程序就会顺序抓取两个网站内容，效率差不多会低一半。

那么，什么时候需要将集合转换为并行集合呢？这当然取决于集合大小。但这并没有所谓的标准值。因为影响执行效率的因素有很多，包括CPU的类型、核数、JVM的版本、集合元素的workload、特定操作、以及内存管理等。

并行集合会启动多个线程来执行，默认情况下，会根据cpu核数以及jvm的设置来确定。如果有兴趣，可以选择两台cpu核数不同的机器分别运行如下代码：

```
(1 to 10000).par.map(i => Thread.currentThread.getName).distinct.size
```

这段代码可以获得线程的数量。

我在演讲时，有人提问这种线程数量的灵活判断究竟取决于编译的机器，还是运行的机器？答案是和运行的机器有关。这事实上是由JVM的编译原理决定的。JVM的编译与纯粹的静态编译不同，Java和Scala编译器都是将源代码转换为JVM字节码，而在运行时，JVM会根据当前运行机器的硬件架构，将JVM字节码转换为机器码。这就是所谓的JIT（just-in-time）编译。

Scala还有很多优势，包括模式匹配、隐式转换、类型类、更好的泛型协变逆变等，当然这些特性也是造成Scala变得更复杂的起因。我们需要明智地判断，控制自己卖弄技巧的欲望，在代码可读性与高效精简之间取得合理的平衡。

题外话

说些题外话，当我推荐Scala时，提出质疑最多的往往不是Java程序员，而是负责团队的管理者，尤其是略懂技术或者曾经做过技术的管理者。他们会表示这样那样的担心，例如Scala的编译速度慢，调试困难，学习曲线高，诸如此类。

编译速度一直是Scala之殇，由于它相当于做了两次翻译，且需要对代码做一些优化，这个问题一时很难彻底根治。

调试困难被吐槽得较激烈，这是因为Scala的调试信息总是让人难以定位。虽然在2.9之后，似乎已有不少改进，但由于类型推断等特性的缘故，相较Java而言，打印的栈信息仍有词不达意之处。曲线救国的方式是多编写小的、职责单一的类（尤其是trait），尽量编写纯函数，以及提高测试覆盖率。此外，调试是否困难还与开发者自身对于Scala这门语言的熟悉程度有关，不能将罪过一味推诿给语言本身。

至于学习曲线高的问题，其实还在于我们对Scala的定位，即确定我们是开发应用还是开发库。此外，对于Scala提供的一些相对晦涩难用的语法，我们尽可以不用。ThoughtWorks技术雷达上将“Scala, the good parts”放到Adopt，而非整个Scala，寓意意味深长。

通常而言，OO转FP会显得相对困难，这是两种根本不同的思维范式。张无忌学太极剑时，学会的是忘记，只取其神，我们学FP，还得尝试忘记OO。自然，学到后来，其实还是万法归一。OO与FP仍然有许多相同的设计原则，例如单一职责，例如分而治之。

对于管理者而言，最关键的一点是明白Scala与Java的优劣对比，然后根据项目情况和团队情况，明智地进行技术决策。我们不能完全脱离上下文去说A优于B。世上哪有绝对呢？

日渐成熟的Scala技术栈

Scala社区的发展

然而，一门语言并不能孤立地存在，必须提供依附的平台，以及围绕它建立的生态圈。不如此，语言则不足以壮大。Ruby很优秀，但如果没有Ruby On Rails的推动，也很难发展到今天这个地步。Scala同样如此。反过来，当我们在使用一门语言时，也要选择符合这门语言的技术栈，在整个生态圈中找到适合具体场景的框架或工具。

当然，我们在使用Scala进行软件开发时，亦可以寻求庞大的Java社区支持；可是，如果选择调用Java开发的库，就会牺牲掉Scala给我们带来的福利。幸运的是，在如今，多数情况你已不必如此。伴随着Scala语言逐渐形成的Scala社区，已经开始慢慢形成相对完整的Scala技术栈。无论是企业开发、自动化测试或者大数据领域，这些框架或工具已经非常完整地呈现了Scala开发的生态系统。

快速了解Scala技术栈

若要了解Scala技术栈，并快速学习这些框架，一个好的方法是下载typesafe推出的Activator。它提供了相对富足的基于Scala以及Scala主流框架的开发模板，这其中实则还隐含了typesafe为Scala开发提供的最佳实践与指导。下图是Activator模板的截图：

Play Framework, AngularJS, WebJars, and RequireJS Seed

 mariussoutier  Source  playframework angularjs scala requirejs webjars

Starter application for Play Framework, AngularJS, RequireJS, and WebJars. Illustrates a more modular approach than the official Typesafe seed, and how to make your app production-ready.

Spray and Websocket interfaces to actors

 cuali  Source  akka spray tcp websocket



This template implements three different interfaces to its Akka actors: HTTP request, plain socket and websocket.

Config, Guice, Akka and Spray

 Eric Halpern  Source  scala spray akka json4s scalatest guice dependency-injection json4s scalatest scaladays2014

A modular scala framework for building and testing real spray REST services

Play Slick quickstart

 loicdescotte  Source  play scala starter slick

Activator template for Play Framework and the Slick database access library. This template helps building a classic Web app or a JSON API.

那么，是否有渠道可以整体地获知Scala技术栈到底包括哪些框架或工具，以及它们的特性与使用场景呢？感谢Lauris

Dzilums以及其他在Github的Contributors。在Lauris Dzilums的Github上，他建立了名为[awesome-scala](#)的Repository，搜罗了当下主要的基于Scala开发的框架与工具，涉及到的领域包括：

- Database
- Web Frameworks
- i18n
- Authentication
- Testing
- JSON Manipulation
- Serialization
- Science and Data Analysis
- Big Data
- Functional Reactive Programming
- Modularization and Dependency Injection
- Distributed Systems
- Extensions
- Android
- HTTP
- Semantic Web
- Metrics and Monitoring
- Sbt plugins

是否有“乱花渐欲迷人眼”的感觉？不是太少，而是太多！那就让我删繁就简，就我的经验介绍一些框架或工具，从持久化、分布式系统、HTTP、Web框架、大数据、测试这六方面入手，作一次蜻蜓点水般的俯瞰。

持久化

归根结底，对数据的持久化主要还是通过JDBC访问数据库。但是，我们需要更好的API接口，能更好地与Scala契合，又或者更自然的ORM。如果希望执行SQL语句来操作数据库，那么运用相对广泛的是框架ScalikeJDBC，它提供了非常简单的API接口，甚至提供了SQL的DSL语法。例如：

```
val alice: Option[Member] = withSQL {
  select.from(Member as m).where.eq(m.name, name)
}.map(rs => Member(rs)).single.apply()
```

如果希望使用ORM框架，Squeryl应该是很好的选择。该框架目前的版本为0.9.5，已经比较成熟了。Squeryl支持按惯例映射对象与关系表，相当于定义一个POSO（Plain Old Scala Object），从而减少框架的侵入。若映射违背了惯例，则可以利用框架定义的annotation如@Column定义映射。框架提供了org.squeryl.Table[T]来完成这种映射关系。

因为可以运用Scala的高阶函数、偏函数等特性，使得Squeryl的语法非常自然，例如根据条件对表进行更新：

```
update(songs)(s =>
  where(s.title === "Watermelon Man")
  set(s.title := "The Watermelon Man",
      s.year := s.year.~ + 1)
)
```

分布式系统

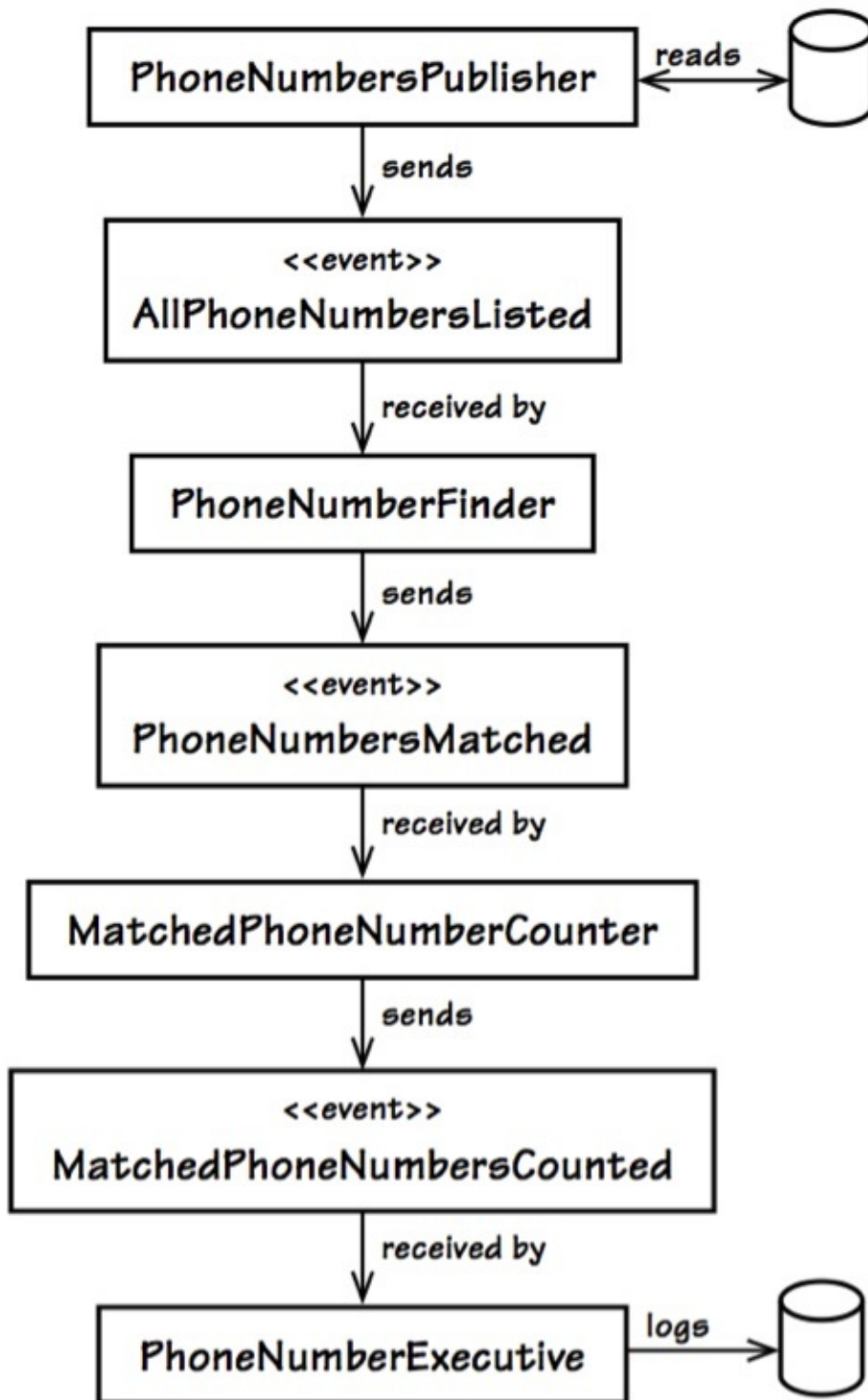
我放弃介绍诸如模块化管理以及依赖注入，是因为它们在Scala社区的价值不如Java社区大。例如，我们可以灵活地运用trait结合cake pattern就可以实现依赖注入的特性。因此，我直接跳过这些内容，来介绍影响更大的支持分布式系统的框架。

Finagle的血统高贵，来自过去的寒门，现在的高门大族Twitter。Twitter是较早使用Scala作为服务端开发的互联网公司，因

而积累了非常多的Scala经验，并基于这些经验推出了一些颇有影响力的框架。由于Twitter对可伸缩性、性能、并发的高要求，这些框架也极为关注这些质量属性。Finagle就是其中之一。它是一个扩展的RPC系统，以支持高并发服务器的搭建。我并没有真正在项目中使用过Finagle，大家可以到它的官方网站获得更多消息。

对于分布式的支持，绝对绕不开的框架还是AKKA。它产生的影响力如此之大，甚至使得Scala语言从2.10开始，就放弃了自己的Actor模型，转而将AKKA Actor收编为2.10版本的语言特性。许多框架在分布式处理方面也选择了使用AKKA，例如Spark、Spray。AKKA的Actor模型参考了Erlang语言，为每个Actor提供了一个专有的Mailbox，并将消息处理的实现细节做了良好的封装，使得并发编程变得更加容易。AKKA很好地统一了本地Actor与远程Actor，提供了几乎一致的API接口。AKKA也能够很好地支持消息的容错，除了提供一套完整的Monitoring机制外，还提供了对Dead Letter的处理。

AKKA天生支持EDA（Event-Driven Architecture）。当我们针对领域建模时，可以考虑针对事件进行建模。在AKKA中，这些事件模型可以被定义为Scala的case class，并作为消息传递给Actor。借用Vaughn Vernon在《实现领域驱动设计》中的例子，针对如下的事件流：



我们可以利用Akka简单地实现：

```

case class AllPhoneNumberListed(phoneNumbers: List[Int])
case class PhoneNumberMatched(phoneNumbers: List[Int])
case class AllPhoneNumberRead(fileName: String)

class PhoneNumbersPublisher(actor: ActorRef) extends ActorRef {
  def receive = {
    case ReadPhoneNumbers =>
      //list phone numbers

    actor ! AllPhoneNumberListed(List(1110, ))
  }
}

```

```

class PhoneNumberFinder(actor: ActorRef) extends ActorRef {
  def receive = {
    case AllPhoneNumberListed(numbers) =>
      //match

      actor ! PhoneNumberMatched()
  }
}

val finder = system.actorOf(Prop(new PhoneNumberFinder(...)))
val publisher = system.actorOf(Prop(new PhoneNumbersPublisher(finder)))

publisher ! ReadPhoneNumbers("callinfo.txt")

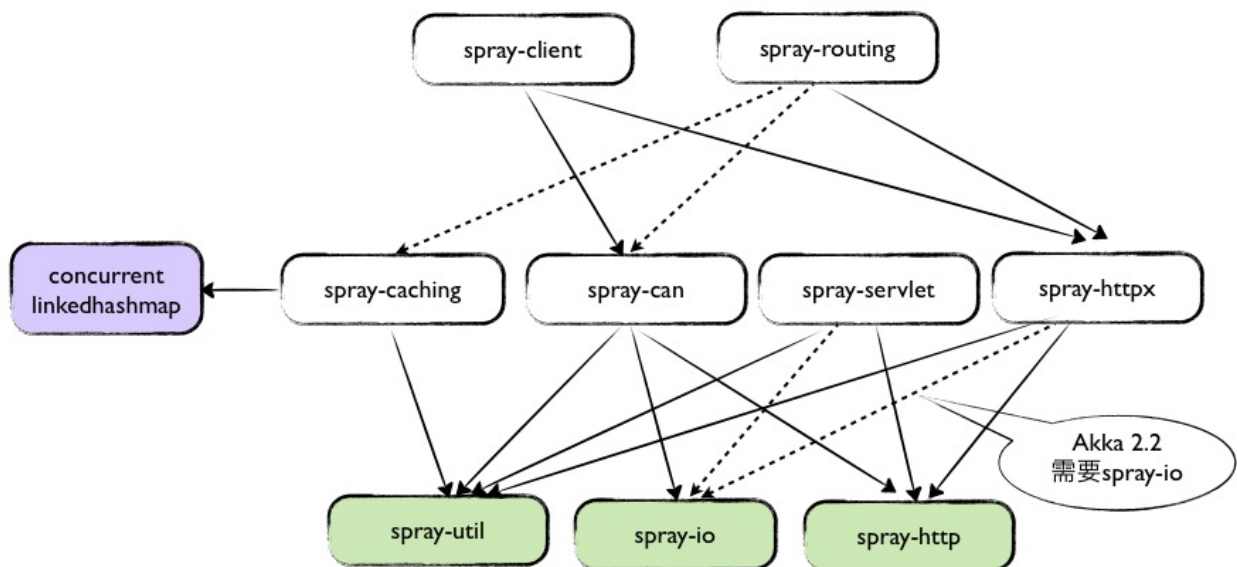
```

若需要处理的电话号码数据量大，我们可以很容易地将诸如PhoneNumbersPublisher、PhoneNumberFinder等Actors部署为Remote Actor。此时，仅仅需要更改客户端获得Actor的方式即可。

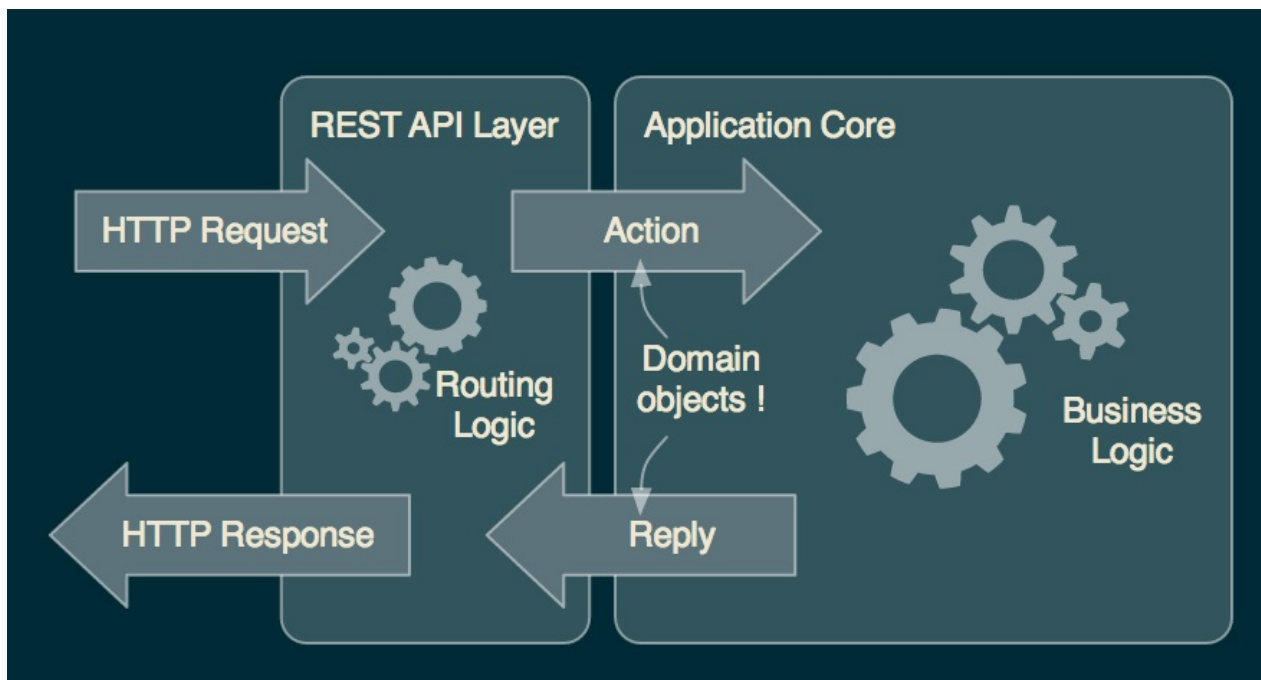
Twitter实现的Finagle是针对RPC通信，Akka则提供了内部的消息队列（MailBox），而由LinkedIn主持开发的Kafka则提供了支持高吞吐量的分布式消息队列中间件。这个顶着文学家帽子的消息队列，能够支持高效的Publisher-Subscriber模式进行消息处理，并以快速、稳定、可伸缩的特性很快引起了开发者的关注，并在一些框架中被列入候选的消息队列而提供支持，例如，Spark Streaming就支持Kafka作为流数据的Input Source。

HTTP

严格意义上讲，Spray并非单纯的HTTP框架，它还支持REST、JSON、Caching、Routing、IO等功能。Spray的模块及其之间的关系如下图所示：



我在项目中主要将Spray作为REST框架来使用，并结合AKKA来处理领域逻辑。Spray处理HTTP请求的架构如下图所示：



Spray提供了一套DSL风格的path语法，能够非常容易地编写支持各种HTTP动词的请求，例如：

```

trait HttpServiceBase extends Directives with Json4sSupport {
  implicit val system: ActorSystem
  implicit def json4sFormats: Formats = DefaultFormats
  def route: Route
}

trait CustomerService extends HttpServiceBase {
  val route =
    path("customer" / "groups") {
      get {
        parameters('groupids.?) {
          (groupids) =>
            complete {
              groupids match {
                case Some(groupIds) =>
                  ViewUserGroup.queryUserGroup(groupIds.split(",").toList)
                case None => ViewUserGroup.queryUserGroup()
              }
            }
        }
      }
    } ~
    path("customers" / "vip" / "failureinfo") {
      post {
        entity(as[FailureVipCustomerRequest]) {
          request =>
            complete {
              VipCustomer.failureInfo(request)
            }
        }
      }
    }
}

```

我个人认为，在进行Web开发时，完全可以放弃Web框架，直接选择AngularJS结合Spray和AKKA，同样能够很好地满足Web开发需要。

Spray支持REST，且Spray自身提供了服务容器spray-can，因而允许Standalone的部署（当然也支持部署到Jetty和tomcat等应用服务器）。Spray对HTTP请求的内部处理机制实则是基于Akka-IO，通过IO这个Actor发出对HTTP的bind消息。例如：

```
IO(Http) ! Http.Bind(service, interface = "0.0.0.0", port = 8889)
```

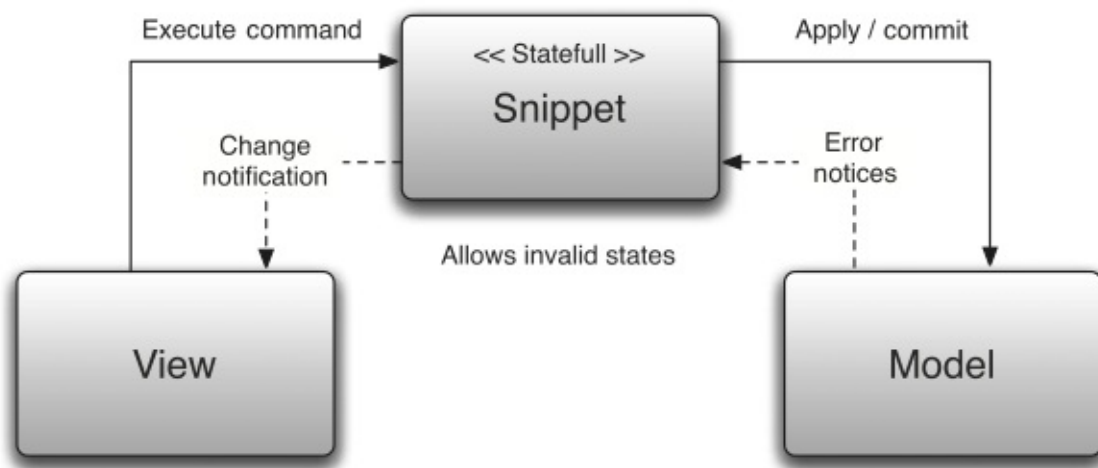
我们可以编写不同的Boot对象去绑定不同的主机Host以及端口。这些特性都使得Spray能够很好地支持当下较为流行的Micro Service架构风格。

Web框架

正如前面所说，当我们选择Spray作为REST框架时，完全可以选择诸如AngularJS或者Backbone之类的JavaScript框架开发Web客户端。客户端能够处理自己的逻辑，然后再以JSON格式发送请求给REST服务端。这时，我们将模型视为资源（Resource），视图完全在客户端。JS的控制器负责控制客户端的界面逻辑，服务端的控制器则负责处理业务逻辑，于是传统的MVC就变化为VC+R+C模式。这里的R指的是Resource，而服务端与客户端则通过JSON格式的Resource进行通信。

若硬要使用专有的Web框架，在Scala技术栈下，最为流行的就是Play Framework，这是一个标准的MVC框架。另外一个相对小众的Web框架是Lift。它与大多数Web框架如RoR、Struts、Django以及Spring MVC、Play不同，采用的并非MVC模式，而是使用了所谓的View First。它驱动开发者对内容生成与内容展现（Markup）形成“关注点分离”。

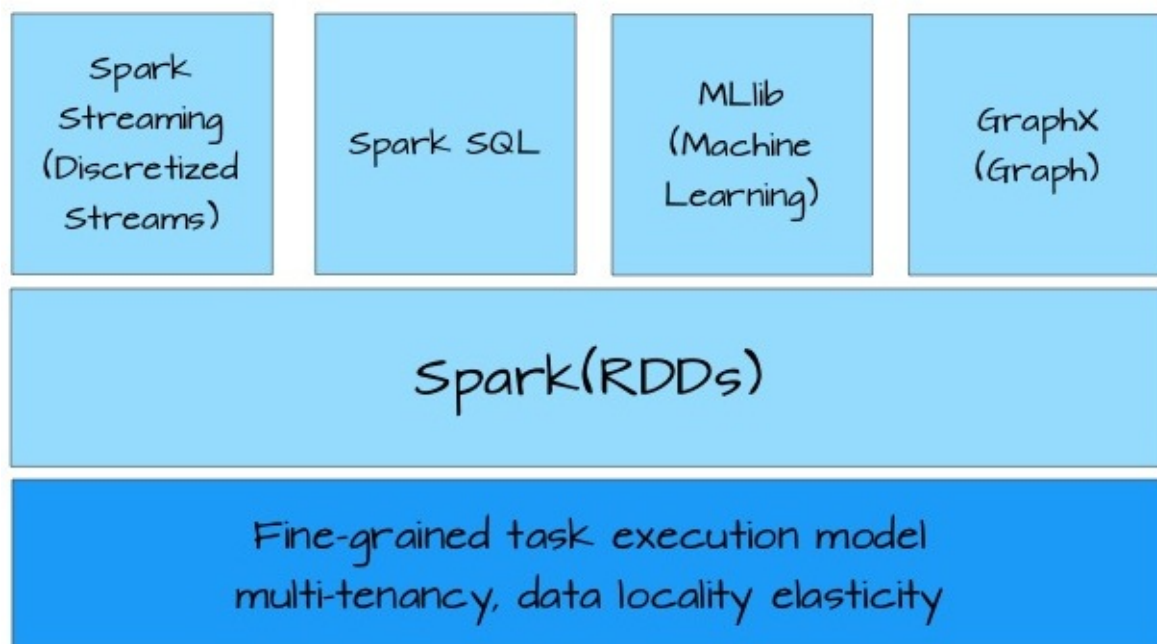
Lift将关注点重点放在View上，这是因为在一些Web应用中，可能存在多个页面对同一种Model的Action。倘若采用MVC中的Controller，会使得控制变得非常复杂。Lift提出了一种所谓view-snippet-model（简称为VSM）的模式。



View主要为响应页面请求的HTML内容，分为template views和generated views。Snippet的职责则用于生成动态内容，并在模型发生更改时，对Model和View进行协调。

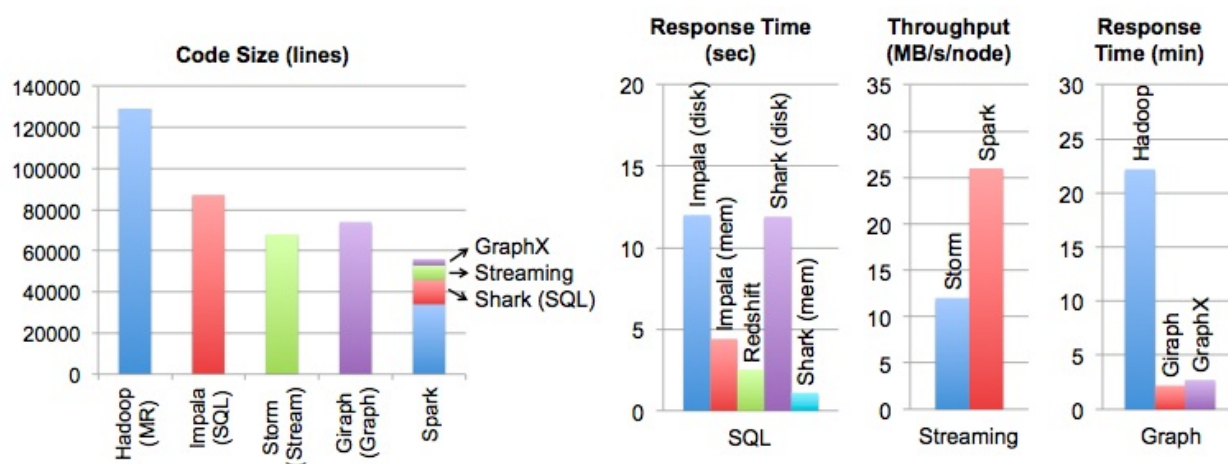
大数据

大数据框架最耀眼的新星非Spark莫属。与许多专有的大数据处理平台不同，Spark建立在统一抽象的RDD之上，使得它可以以基本一致的方式应对不同的大数据处理场景，包括MapReduce，Streaming，SQL，Machine Learning以及Graph等。这即Matei Zaharia所谓的“设计一个通用的编程抽象（Unified Programming Abstraction）”。



由于Spark具有先进的DAG执行引擎，支持cyclic data flow和内存计算。因此相比较Hadoop而言，性能更优。在内存中它的运行速度是Hadoop MapReduce的100倍，在磁盘上是10倍。

由于使用了Scala语言，通过高效利用Scala的语言特性，使得Spark的总代码量出奇地少，性能却在多数方面都具备一定的优势（只有在Streaming方面，逊色于Storm）。下图是针对Spark 0.9版本的BenchMark：



由于使用了Scala，使得语言的函数式特性得到了最棒的利用。事实上，函数式语言的诸多特性包括不变性、无副作用、组合子等，天生与数据处理匹配。于是，针对WordCount，我们可以如此简易地实现：

```
file = spark.textFile("hdfs://...")

file.flatMap(line => line.split(" "))
    .map(word => (word, 1))
    .reduceByKey(_ + _)
```

要是使用Hadoop，就没有这么方便了。幸运的是，Twitter的一个开源框架scalding提供了对Hadoop MapReduce的抽象与包装。它使得我们可以按照Scala的方式执行MapReduce的Job：

```
class WordCountJob(args : Args) extends Job(args) {
  TextLine( args("input") )
```



```

.flatMap('line -> 'word) { line : String => tokenize(line) }
.groupBy('word) { _.size }
.write( Tsv( args("output") ) )

// Split a piece of text into individual words.
def tokenize(text : String) : Array[String] = {
  // Lowercase each word and remove punctuation.
  text.toLowerCase.replaceAll("[^a-zA-Z0-9\\s]", "").split("\\s+")
}
}

```

测试

虽然我们可以使用诸如JUnit、TestNG为Scala项目开发编写单元测试，使用Cucumber之类的BDD框架编写验收测试。但在多数情况下，我们更倾向于选择使用ScalaTest或者Specs2。在一些Java开发项目中，我们也开始尝试使用ScalaTest来编写验收测试，乃至单元测试。

若要我选择ScalaTest或Specs2，我更倾向于ScalaTest，这是因为ScalaTest支持的风格更具备多样性，可以满足各种不同的需求，例如传统的JUnit风格、函数式风格以及Spec方式。我的一篇博客《ScalaTest的测试风格》详细介绍了各自的语法。

一个被广泛使用的测试工具是Gatling，它是基于Scala、AKKA以及Netty开发的性能测试与压力测试工具。我的同事刘冉在InfoQ发表的文章《新一代服务器性能测试工具Gatling》对Gatling进行了详细深入的介绍。

ScalaMeter也是一款很不错的性能测试工具。我们可以像编写ScalaTest测试那样的风格来编写ScalaMeter性能测试用例，并能够快捷地生成性能测试数据。这些功能都非常有助于我们针对代码或软件产品进行BenchMark测试。我们曾经用ScalaMeter来编写针对Scala集合的性能测试，例如比较Vector、ArrayBuffer、ListBuffer以及List等集合的相关操作，以便于我们更好地使用Scala集合。以下代码展示了如何使用ScalaMeter编写性能测试：

```

import org.scalameter.api._

object RangeBenchmark
extends PerformanceTest.Microbenchmark {
  val ranges = for {
    size <- Gen.range("size")(300000, 1500000, 3000000)
  } yield 0 until size

  measure method "map" in {
    using(ranges) curve("Range") in {
      _.map(_ + 1)
    }
  }
}

```

根据场景选择框架或工具

比起Java庞大的社区，以及它提供的浩如烟海般的技术栈，Scala技术栈差不多可以说是沧海一粟。然而，麻雀虽小却五脏俱全，何况Scala以及Scala技术栈仍然走在迈向成熟的道路上。对于Scala程序员而言，因为项目的不同，未必能涉猎所有技术栈，而且针对不同的方面，也有多个选择。在选择这些框架或工具时，应根据实际的场景做出判断。为稳妥起见，最好能运用技术矩阵地方式对多个方案进行设计权衡与决策。

我们也不能固步自封，视Java社区而不顾。毕竟那些Java框架已经经历了千锤百炼，并有许多成功的案例作为佐证。关注Scala技术栈，却又不局限自己的视野，量力而为，选择合适的技术方案，才是设计与开发的正道。

函数式思想

函数的抽象能力

我在阅读或编写具有函数式风格的代码时，常常为函数式思想非凡的抽象能力所惊叹。作为一直以来持有OO信仰的程序员而言，对于“抽象”并不陌生。我甚至将面向对象思想的精髓定义为两个单词：职责（Responsibility）与抽象（Abstraction）。只要职责分配合理，设计就是良好的；若能再加上合理的抽象，程序会变得更精简且可扩展。如果你熟悉GoF的设计模式，你几乎可以从每个模式中读出“抽象”的意义来。

然而，无论如何，面向对象思想构筑的其实是一个名词的世界，这在很大程度上局限了它的世界观，它只能以实体（Entity）为核心。虽然我们仍然可以针对实体提炼共同特征，但这些特征若为行为，却无法单独存在，这是面向对象思想的硬伤。

如果说面向对象思想是物质世界的哲学观，则函数式思想展现的就是纯粹的数学思维了。函数作为一等公民，它不代表任何物质（对象），而仅仅代表一种转换行为。是的，任何一个函数都可以视为一种“转换(transform)”。这是对行为的最高抽象，代表了类型（type）[注意，是类型（type），而不是类（class）]之间的某种动作。函数可以是极为原子的操作，也可以是多个原子函数的组合，或者在组合之上再封装一层语义更清晰的函数表现。

理解了函数的转换本质，我们就必须学会在具体行为中“洞见”这种转换本质。这种“洞见”可以理解为解构分析，就好似我们在甄别化石的年代时，利用核分析技术去计算碳14同位素原子数量一般。我们解构出来的“原子”函数往往具有非凡的抽象能力。例如，我们针对集合的sum与product操作，可以解构出原子的fold函数。虽然从行为特征看，sum为求和，product为求积，但从抽象层面看，都是从一个初始值开始，依次对集合元素进行运算。而运算本身，又是抽象的另一个转换操作，从而引入了高阶函数的概念。若要让fold不止局限于某一种具体类型，则可以引入函数式语言的类型系统。fold可以根据折叠的方向分为foldRight与foldLeft。foldRight（或foldr）的函数定义如下：

```
//scala语言
def fold[A, B](l: MyList[A], z: B)(f: (A, B) => B): B = l match {
  case Nil => z
  case Cons(x, xs) => f(x, fold(xs, z)(f))
}
```

```
--haskell语言
foldr f zero (x:xs) = f x (foldr f zero xs)
foldr _ zero []     = zero
```

《深入理解Scala》一书在讲解Scala的Option时，给出了一个有趣的案例，其中揭示的抽象思想与fold有异曲同工之妙。这个案例讲解了如何用多个可能未初始化的变量构造另一个变量，Option正适合处理这种情况，我在博客《并非Null Object这么简单》中介绍了Option的本质，这里不再赘述。这个例子是希望通过数据库配置信息创建连接。由于配置信息可能有误，创建的连接可能为null，因而使用Option的api会更加健壮：

```
def createConnection(conn_url: Option[String],
                     conn_user: Option[String],
                     conn_pw: Option[String]): Option[Connection] =
  for {
    url <- conn_url
    user <- conn_user
    pw <- conn_pw
  } yield DriverManager.getConnection(url, user, pw)
```

现在，我们将这个函数无限抽象化，那就是要去掉一些复杂而冗余的具象信息，就好像过滤掉让人眼花缭乱的缤纷颜色，仅仅呈现最朴素的黑白二色一般。首先，我们抹掉“创建连接”的特征，然后再抹掉类型信息。我们可以看到createConnection实则是对DriverManager.getConnection的转换，经此转换后，若要创建连接，就可以传入三个Option[String]类型的参数，获得

Option[Connection]类型的结果。然后再去掉具体的String类型，就可以抽象出如下的“转换”操作：

```
(A, B, C): => D      转换为      (Option[A], Option[B], Option[C]) => Option[D]
```

注意，这个转换操作是函数到函数的转换。

书中找到了一个正确的概念来恰如其分地描述这一“转换”操作，即为lift（提升）：

```
def lift[A, B, C, D](f: Function3[A, B, C, D]): Function3[Option[A], Option[B], Option[C], Option[D]] =
  (oa: Option[A], ob: Option[B], oc: Option[C]) =>
    for (a <- oa; b <- ob; c <- oc) yield f(a, b, c)
```

Function3事实上是Scala中对(A, B, C) => D函数的封装。相对而言，我更喜欢高阶函数的形式：

```
def lift[A, B, C, D](f: (A, B, C) => D): (Option[A], Option[B], Option[C]) => Option[D] =
  (oa: Option[A], ob: Option[B], oc: Option[C]) =>
    for (a <- oa; b <- ob; c <- oc) yield f(a, b, c)
```

lift函数是宽泛的抽象，之前的DriverManager.getConnection()函数则为一个具体的被转换对象。它可以作为参数传入到lift函数中：

```
val createConnection1 = lift(DriverManager.getConnection)
```

lift函数返回的实则是一个函数，它本质上等同于之前定义的createConnection()函数。由于lift抹掉了具体的类型信息，使得它不仅可以将getConnection提升为具有Option的函数，还能针对所有形如(A, B, C) => D格式的函数。让我们来自定义一个combine函数：

```
def combine(prefix: String, number: Int, suffix: String): String =
  s"$prefix - $number - $suffix"

val optionCombine = lift(combine)
```

区分combine函数与optionCombine函数的执行结果：

```
scala> val optionCombine = lift(combine)
optionCombine: (Option[String], Option[Int], Option[String]) => Option[String] = <function3>

scala> combine("@", 20, "@")
res18: String = @ - 20 - @

scala> combine("@", 20, null)
res19: String = @ - 20 - null

scala> optionCombine(Some("@"), Some(20), Some("@"))
res20: Option[String] = Some(@ - 20 - @)

scala> optionCombine(Some("@"), None, Some("@"))
res21: Option[String] = None
```

△ lift的执行结果

诸如fold或lift这样的终极抽象在函数式语言的api中可谓俯拾皆是，如针对集合的monad操作filter, flatMap, map，又例如函数

组合的操作sequence, andThen等。我们还可以结合转换语义为这种基本转换命名,使得代码更加简略可读。例如针对如下的三个函数定义:

```
def intDouble(rng: RNG): ((Int,Double), RNG)
def doubleInt(rng: RNG): ((Double,Int), RNG)
def double3(rng: RNG): ((Double,Double,Double), RNG)
```

我们可以抽象出RNG => (A, RNG)的通用模式,然后从语义上将其命名为Rand,那么,在scala中可以利用type关键字为这种转换定义别名:

```
type Rand[+A] = RNG => (A, RNG)
```

当我们将函数作为基本的抽象单元后,再对面向对象思想做一次回眸,会发现OO中的多数设计原则与设计模式,都可以简化为函数。Scott Wlaschin在Functional Design Patterns的演讲中给出了非常形象的对比:

OO pattern/principle

- Single Responsibility Principle
- Open/Closed principle
- Dependency Inversion Principle
- Interface Segregation Principle
- Factory pattern
- Strategy pattern
- Decorator pattern
- Visitor pattern

FP pattern/principle

- Functions
- Functions
- Functions, also
- Functions
- Yes, functions
- Oh my, functions again!
- Functions
- Functions 😊

Seriously, FP patterns are different

△ OO和FP的模式与原则

显然,函数才是最为纯粹的抽象。正所谓“大道至简”,有时候,简单可能就意味着一切。

Scala特性

Option与Null Object模式

在大多数程序语言中，我们都需要与Null打交道，并且纠缠于对它的检查中。一不小心让它给溜出来，就可能像打开潘多拉的盒子一般，给程序世界带来灾难。说起来，在我们人类世界中，Null到底算什么“东西”呢？语义上讲，它就是一场空，即所谓“虚无”。这个世界并没有任何物质可以代表“虚无”，因而它仅存于我们的精神层面。说虚无存在其实是一种悖论，因为存在其实是虚无的反面。若从程序本质上讲，Null代表一种状态，指一个对象（或变量），虽获声明却未真正诞生，甚至可能永远不会诞生。而一旦诞生，Null就被抹去了，回归了正确的状态。

站在OO的角度来讲，既然Everything is object，自然可以将Null同样视为Object——这近似于前面提到的悖论，既然是Null，为何又是Object呢？换言之，在对象世界里，其实没有什么不存在，所谓“不存在”仍然是一种“存在”。这么说容易让人变糊涂，就好像我们搞不清楚“我是谁”。所以，我宁肯采用Martin Fowler的说法，将Null Object视为一种Special Case，即Null其实是一种特例。

视Null为一种特例，即可用OO的特化来表达。当某个对象可能存在Null这种状态时，都可以将这种状态表示为一种特化的类，它不再代表Null，而是代表“什么都不做”。凡是返回Null的地方，都替换为这个Null Object，用以表达这种Null其实仅仅是一种特例。于是乎，我们像抹杀异教徒一般抹去了“虚无”的存在。（当虚无被抹去，是什么样的存在？）

然而，若在程序语言中实现自己的Null Object，固然可以在一定程度上消除对Null的检查，却存在一些约束：

- 对于String之类的类型，无法定义NullString子类；
- 每次都需要自己去定义子类来表示Null；
- 必须约束团队不能返回Null；

Google的Guava框架为了解决这一问题，引入了Optional：

```
public abstract class Optional<T> implements Serializable {
    public static <T> Optional<T> absent() {
        return (Optional<T>) Absent.INSTANCE;
    }
    public static <T> Optional<T> of(T reference) {
        return new Present<T>(checkNotNull(reference));
    }
    public static <T> Optional<T> fromNullable(@Nullable T nullableReference) {
        return (nullableReference == null)
            ? Optional.<T>absent()
            : new Present<T>(nullableReference);
    }
    public abstract boolean isPresent();
    public abstract T get();
    public abstract T or(T defaultValue);
    public abstract <V> Optional<V> transform(Function<? super T, V> function);
}
```

于是，我们可以这样来使用Optional:

```
public final Optional<E> first() {
    Iterator<E> iterator = iterable.iterator();
    return iterator.hasNext()
        ? Optional.of(iterator.next())
        : Optional.<E>absent();
}
```

first()方法返回的是一个Optional类型。这是Guava中操作集合的一个方法。当我们要获得第一个元素时，可以调用该方法：

```
List<Person> persons = newArrayList();
String name = from(persons).first().transform(new Function<Person, String>() {
```

```
@Override
public String apply(Person input) {
    return input.getName();
}
}).or("not found");
assertThat(name, is("not found"));
```

不知是巧合，还是一种借鉴，Java 8同样定义了Optional用以处理这种情况。前面的代码在Java 8下可以改写为：

```
List<Person> persons = new ArrayList();
String name = persons.stream().findFirst().map(p -> p.getName()).orElse("not found");
assertThat(name, is("not found"));
```

其实在Scala的早期版本，已经提供了Option[T]类型。前面的代码若用scala编写，就变成：

```
case class Person(name: String, age: Int)
val persons = List[Person]()
persons.headOption.map(p => p.name).getOrElse("not found")
```

这样的设计方式，还是Null Object模式吗？让我们回到Null的本原状态，思考为什么会产生Null？首先，Null代表一种异常状态，即在某种未可知的情形下，可能返回Null；正常情况下，返回的则是非Null的对象。Null与非Null，代表一种未知与不确定性。哈姆雷特纠结于“To be, or not to be, this is a question”，但在程序世界里，可以抽象为一个集合来表达这种非此即彼的状况。

从函数式编程的角度来讲，我们可以将这样的集合设计为一个Monad。根据DSL in Action一书中对Monad的介绍，一个Monad由以下三部分定义：

- 一个抽象M[A],其中M是类型构造函数。在Scala语言中可以写成class M[A]，或者case class M[A]，有或者trait M[A]
- 一个unit方法（unit v）。对应Scala中的函数new M(v)或者M(v)的调用。
- 一个bind方法，起到将运算排成序列的作用。在Scala中通过flatMap组合子来实现。bind f m对应的Scala语句是m.flatMap f。

同时，Monad还必须满足以下三条规则：

- 右单位元(identity)。即对于任意Monad m，有m.flatMap unit => m。对于Option，unit就是Option伴生对象定义的apply()方法。若m为Some("Scala")，则m.flatMap {x => Option(x)}，其结果还是m。
- 左单位元(unit)。即对于任意Monad m，有unit(v).flatMap f => f(v)。

假设我们定义一个函数f：

```
def f(v: String) = Option(v)
```

则Option("Scala").flatMap {x => f(x)}的结果就等于f("scala")。

- 结合律。即对于任意Monad m，有m.flatMap g.flatMap h => m.flatMap {x => g(x).flatMap h}。

无论是Scala中的Option[A]，还是Java 8中的Optional[T]，都是一个Monad。此时的Null不再是特例，而是抽象Option[A]对称的两个元素中的其中一个，在Scala中，即Option[T]中的Some[T]或None。它们俩面貌相同，却是一对性格迥异的双生子。

在设计为Monad后，就可以利用Monad提供的bind功能，完成多个函数的组合。组合时，并不需要考虑返回为None的情况。Monad能保证在前一个函数返回空值时，后续函数不会被调用。让我们来看一个案例。例如，我们需要根据某个key从会话中获得对应的值，然后再将该值作为参数去查询符合条件的特定Customer。在Scala中，可以将这两个步骤定义为函数，返回结果分别为Option[String]与Option[Customer]：

```
def params(key: String): Option[String]
def queryCustomer(refId: String): Option[Customer]
val customer =
  (
    for {
      r <- params("customerId")
      c <- queryCustomer(r)
    } yield c
  ) getOrElse error("Not Found")
```

这段代码用到了Scala的for comprehension，它实则是对flatMap的一种包装。尤其当嵌套多个flatMap时，使用for comprehension会更加直观可读。翻译为flatMap，则为：

```
params("customerId").flatMap{
  r => queryCustomer(r).map {
    c => c
  }
} getOrElse error("Not Found")
```

当我最初看到Guava设计的Optional[T]时，我以为是Null Object模式的体现。显然，它的功能要超出Null Object的范畴。但它也并非Monad，在前面给出的定义中，我们可以看到Guava的Optional[T]仅提供了map（即定义中的transform）功能，而没有提供更基本的flatMap操作。具有函数式编程功能的Scala与Java 8加强了这一功能，利用Monad强化了程序对Null的处理。

Scala编码规范

这是我去年在一个Scala项目中结合一些参考资料和项目实践整理的一份编码规范，基于的Scala版本为2.10，但同时也适用于2.11版本。参考资料见文后。整个编码规范分为如下六个部分：

1. 格式与命名
2. 语法特性
3. 编码风格
4. 高效编码
5. 编码模式
6. 测试

格式与命名

1) 代码格式 用两个空格缩进。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。

2) 优先考虑使用val，而非var。

3) 当引入多个包时，使用花括号：

```
import jxl.write.{WritableCell, Number, Label}
```

当引入的包超过6个时，应使用通配符_：

```
import org.scalatest.events._
```

4) 若方法暴露为接口，则返回类型应该显式声明。例如：

```
def execute(conn: Connection): Boolean = {  
  executeCommand(conn, sqlStatement) match {  
    case Right(result) => result  
    case Left(_) => false  
  }  
}
```

5) 集合的命名规范 xs, ys, as, bs等作为某种Sequence对象的名称；x, y, z, a, b作为sequence元素的名称。h作为head的名称，t作为tail的名称。

6) 避免对简单的表达式采用花括号；

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

7) 泛型类型参数的命名虽然没有限制，但建议遵循如下规则：A代表一个简单的类型，例如List[A] B, C, D用于第2、第3、第4等类型。例如：class List[A] { def mapB: List[B] = ... } N代表数值类型

注意：在Java中，通常以K、V代表Map的key与value，但是在Scala中，更倾向于使用A、B代表Map的key与value。

语法特性

- 1) 定义隐式类时，应该将构造函数的参数声明为val。
- 2) 使用for表达式；如果需要条件表达式，应将条件表达式写到for comprehension中：

```
//not good
for (file <- files) {
  if (hasSoundFileExtension(file) && !soundFileIsLong(file)) {
    soundFiles += file
  }
}

//better
for {
  file <- files
  if hasSoundFileExtension(file)
  if !soundFileIsLong(file)
} yield file
```

通常情况下，我们应优先考虑filter, map, flatMap等操作，而非for comprehension：

```
//best
files.filter(hasSourceFileExtension).filterNot(soundFileIsLong)
```

- 3) 避免使用isInstanceOf，而是使用模式匹配，尤其是在处理比较复杂的类型判断时，使用模式匹配的可读性更好。

```
//avoid
if (x.isInstanceOf[Foo]) { do something ...

//suggest
def isPerson(x: Any): Boolean = x match {
  case p: Person => true
  case _ => false
}
```

- 4) 以下情况使用abstract class，而不是trait：

- 想要创建一个需要构造函数参数的基类
- 代码可能会被Java代码调用

- 5) 如果希望trait只能被某个类（及其子类）extend，应该使用self type：

```
trait MyTrait {
  this: BaseType =>
}
```

如果希望对扩展trait的类做更多限制，可以在self type后增加更多对trait的混入：

```
trait WarpCore {
  this: Starship with WarpCoreEjector with FireExtinguisher =>
}

// this works
class Enterprise extends Starship
```

```

with WarpCore
with WarpCoreEjector
with FireExtinguisher

// won't compile
class Enterprise extends Starship
  with WarpCore
  with WarpCoreEjector

```

如果要限制扩展trait的类必须定义相关的方法，可以在self type中定义方法，这称之为structural type（类似动态语言的鸭子类型）：

```

trait WarpCore {
  this: {
    def ejectWarpCore(password: String): Boolean
    def startWarpCore: Unit
  } =>
}

class Starship
class Enterprise extends Starship with WarpCore {
  def ejectWarpCore(password: String): Boolean = {
    if (password == "password") { println("core ejected"); true } else false }
  def startWarpCore { println("core started") }
}

```

6) 对于较长的类型名称，在特定上下文中，以不影响阅读性和表达设计意图为前提，建议使用类型别名，它可以帮助程序变得更简短。例如：

```

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

```

7) 如果要使用隐式参数，应尽量使用自定义类型作为隐式参数的类型，而避免过于宽泛的类型，如String, Int, Boolean等。

```

//suggestion
def maxOfList[T](elements: List[T])
  (implicit orderer: T => Ordered[T]): T =
  elements match {
    case List() =>
      throw new IllegalArgumentException("empty list!")
    case List(x) => x
    case x :: rest =>
      val maxRest = maxListImpParam(rest)(orderer)
      if (orderer(x) > maxRest) x
      else maxRest
  }

//avoid
def maxOfListPoorStyle[T](elements: List[T])
  (implicit orderer: (T, T) => Boolean): T

```

8) 对于异常的处理，Scala除了提供Java风格的try...catch...finally之外，还提供了allCatch.opt、Try...Success...Failure以及Either...Right...Left等风格的处理方式。其中，Try是2.10提供的语法。根据不同的场景选择不同风格：

优先选择Try风格。Try很好地支持模式匹配，它兼具Option与Either的特点，因而既提供了集合的语义，又支持模式匹配，又提供了getOrElse()方法。同时，它还可以组合多个Try，并支持运用for combination。

```

val z = for {
  a <- Try(x.toInt)
  b <- Try(y.toInt)
}

```

```

} yield a * b
val answer = z.getOrElse(0) * 2

```

如果希望清楚的表现非此即彼的特性，应考虑使用Either。注意，约定成俗下，我们习惯将正确的结果放在Either的右边（Right既表示右边，又表示正确）

如果希望将异常情况处理为None，则应考虑使用allCatch.opt。

```

import scala.util.control.Exception._

def readTextFile(f: String): Option[List[String]] =
  allCatch.opt(Source.fromFile(f).getLines.toList)

```

如果希望在执行后释放资源，从而需要使用finally时，考虑try...catch...finally，或者结合try...catch...finally与Either。

```

private def executeQuery(conn: Connection, sql: String): Either[SQLException, ResultSet] = {
  var stmt: Statement = null
  var rs: ResultSet = null
  try {
    stmt = conn.createStatement()
    rs = stmt.executeQuery(sql)
    Right(rs)
  } catch {
    case e: SQLException => {
      e.printStackTrace()
      Left(e)
    }
  } finally {
    try {
      if (rs != null) rs.close()
      if (stmt != null) stmt.close()
    } catch {
      case e: SQLException => e.printStackTrace()
    }
  }
}

```

为避免重复，还应考虑引入Load Pattern。

编码风格

1) 尽可能直接在函数定义的地方使用模式匹配。例如，在下面的写法中，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。此时，传入map的函数实则为partial function。

2) 避免使用null，而应该使用Option的None。

```
import java.io._

object CopyBytes extends App {
  var in = None: Option[FileInputStream]
  var out = None: Option[FileOutputStream]
  try {
    in = Some(new FileInputStream("/tmp/Test.class"))
    out = Some(new FileOutputStream("/tmp/Test.class.copy"))
    var c = 0
    while ({c = in.get.read; c != -1}) {
      out.get.write(c)
    }
  } catch {
    case e: IOException => e.printStackTrace
  } finally {
    println("entered finally ...")
    if (in.isDefined) in.get.close
    if (out.isDefined) out.get.close
  }
}
```

方法的返回值也要避免返回Null。应考虑返回Option，Either，或者Try。例如：

```
import scala.util.{Try, Success, Failure}

def readTextFile(filename: String): Try[List[String]] = {
  Try(io.Source.fromFile(filename).getLines.toList)
}

val filename = "/etc/passwd"
readTextFile(filename) match {
  case Success(lines) => lines.foreach(println)
  case Failure(f) => println(f)
}
```

3) 若在Class中需要定义常量，应将其定义为val，并将其放在该类的伴生对象中：

```
class Pizza (var crustSize: Int, var crustType: String) {
  def this(crustSize: Int) {
    this(crustSize, Pizza.DEFAULT_CRUST_TYPE)
  }

  def this(crustType: String) {
    this(Pizza.DEFAULT_CRUST_SIZE, crustType)
  }

  def this() {
    this(Pizza.DEFAULT_CRUST_SIZE, Pizza.DEFAULT_CRUST_TYPE)
  }
  override def toString = s"A $crustSize inch pizza with a $crustType crust"
}

object Pizza {
  val DEFAULT_CRUST_SIZE = 12
  val DEFAULT_CRUST_TYPE = "THIN"
}
```

4) 合理为构造函数或方法提供默认值。例如：

```
class Socket (val timeout: Int = 10000)
```

5) 如果需要返回多个值时，应返回tuple。

```
def getStockInfo = {
  //
  ("NFLX", 100.00, 101.00)
}
```

6) 作为访问器的方法，如果没有副作用，在声明时建议定义为没有括号。

例如，Scala集合库提供的scala.collection.immutable.Queue中，dequeue方法没有副作用，声明时就没有括号：

```
import scala.collection.immutable.Queue

val q = Queue(1, 2, 3, 4)
val value = q.dequeue
```

7) 将包的公有代码（常量、枚举、类型定义、隐式转换等）放到package object中。

```
package com.agiledon.myapp

package object model {
  // field
  val MAGIC_NUM = 42 182 | Chapter 6: Objects

  // method
  def echo(a: Any) { println(a) }

  // enumeration
  object Margin extends Enumeration {
    type Margin = Value
    val TOP, BOTTOM, LEFT, RIGHT = Value
  }

  // type definition
  type MutableMap[K, V] = scala.collection.mutable.Map[K, V]
  val MutableMap = scala.collection.mutable.Map
}
```

8) 建议将package object放到与包对象命名空间一致的目录下，并命名为package.scala。以model为例，package.scala文件应放在：+- com +- agiledon +- myapp +- model +- package.scala

9) 若有多个样例类属于同一类型，应共同继承自一个sealed trait。

```
sealed trait Message
case class GetCustomers extends Message
case class GetOrders extends Message
```

注：这里的sealed，表示trait的所有实现都必须声明在定义trait的文件中。

10) 考虑使用renaming clause来简化代码。例如，替换被频繁使用的长名称方法：

```
import System.out.{println => p}

p("hallo scala")
p("input")
```

11) 在遍历Map对象或者Tuple的List时，且需要访问map的key和value值时，优先考虑采用Partial Function，而非使用_1和_2的形式。例如：

```
val dollar = Map("China" -> "CNY", "US" -> "DOL")

//perfer
dollar.foreach {
  case (country, currency) => println(s"$country -> $currency")
}

//avoid
dollar.foreach ( x => println(s"$x._1 -> $x._2") )
```

或者，考虑使用for comprehension：

```
for ((country, currency) <- dollar) println(s"$country -> $currency")
```

12) 遍历集合对象时，如果需要获得并操作集合对象的下标，不要使用如下方式：

```
val l = List("zero", "one", "two", "three")

for (i <- 0 until l.length) yield (i, l(i))
```

而应该使用zipWithIndex方法：

```
for ((number, index) <- l.zipWithIndex) yield (index, number)
```

或者：

```
l.zipWithIndex.map(x => (x._2, x._1))
```

当然，如果需要将索引值放在Tuple的第二个元素，就更方便了。直接使用zipWithIndex即可。

zipWithIndex的索引初始值为0，如果想指定索引的初始值，可以使用zip：

```
l.zip(Stream from 1)
```

13) 应尽量定义小粒度的trait，然后再以混入的方式继承多个trait。例如ScalaTest中的FlatSpec：

```
class FlatSpec extends FlatSpecLike ...

trait FlatSpecLike extends Suite with ShouldVerb with MustVerb with CanVerb with Informing ...
```

小粒度的trait既有利于重用，同时还有利于对业务逻辑进行单元测试，尤其是当一部分逻辑需要依赖外部环境时，可以运用“关注点分离”的原则，将不依赖于外部环境的逻辑分离到单独的trait中。

14) 优先使用不可变集合。如果确定要使用可变集合，应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。

15) 在自己定义的方法和构造函数里，应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]。这样可以分离集合与它的实现，从而达成更好的可扩展性。

16) 应谨慎使用流水线转换的形式。当流水线转换的逻辑比较复杂时，应充分考虑代码的可读性，准确地表达开发者的意图，而不过分追求函数式编程的流水线转换风格。例如，我们想要从一组投票结果(语言，票数)中统计不同程序语言的票数并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)。

17) 对于 Options 对象，如果 getOrElse 能够表达业务逻辑，就应避免对其使用模式匹配。许多集合的操作都提供了返回 Options 的方法。例如 headOption 等。

```
val x = list.headOption getOrElse 0
```

这要比模式匹配更清楚：

```
val x = list match
  case head::_ => head
  case Nil: => 0
```

18) 当需要对两个或两个以上的集合进行操作时，应优先考虑使用 for 表达式，而非 map, flatMap 等操作。此时，for comprehension 会更简洁易读。例如，获取两个字符的所有排列，相同的字符不能出现两次。使用 flatMap 的代码为：

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

使用 for comprehension 会更易懂：

```
val perms = for {
```



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

高效编码

1) 应尽量避免让trait去extend一个class。因为这种做法可能会导致间接的继承多个类，从而产生编译错误。同时，还会导致继承体系的复杂度。

```
class StarfleetComponent
trait StarfleetWarpCore extends StarfleetComponent
class Starship extends StarfleetComponent with StarfleetWarpCore
class RomulanStuff

// won't compile
class Warbird extends RomulanStuff with StarfleetWarpCore
```

2) 选择使用Seq时，若需要索引下标功能，优先考虑选择Vector，若需要Mutable的集合，则选择ArrayBuffer；若要选择Linear集合，优先选择List，若需要Mutable的集合，则选择ListBuffer。

3) 如果需要快速、通用、不变、带顺序的集合，应优先考虑使用Vector。Vector很好地平衡了快速的随机选择和快速的随机更新（函数式）操作。Vector是Scala集合库中最灵活的高效集合。一个原则是：当你对选择集合类型犹疑不定时，就应选择使用Vector。

需要注意的是：当我们创建了一个IndexSeq时，Scala实际上会创建Vector对象：

```
scala> val x = IndexedSeq(1,2,3)
x: IndexedSeq[Int] = Vector(1, 2, 3)
```

4) 如果需要选择通用的可变集合，应优先考虑使用ArrayBuffer。尤其面对一个大的集合，且新元素总是要添加到集合末尾时，就可以选择ArrayBuffer。如果使用的可变集合特性更近似于List这样的线性集合，则考虑使用ListBuffer。

5) 如果需要将大量数据添加到集合中，建议选择使用List的prepend操作，将这些数据添加到List头部，最后做一次reverse操作。例如：

```
var l = List[Int]()
(1 to max).foreach {
  i => i += 1
}
l.reverse
```

6) 当一个类的某个字段在获取值时需要耗费资源，并且，该字段的值并非一开始就需要使用。则应将该字段声明为lazy。

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

7) 在使用Future进行并发处理时，应使用回调的方式，而非阻塞：

```
//avoid
val f = Future {
  //executing long time
}

val result = Await.result(f, 5 second)
```

```
//suggestion
val f = Future {
    //executing long time
}
f.onComplete {
    case Success(result) => //handle result
    case Failure(e) => e.printStackTrace
}
```

8) 若有多个操作需要并行进行同步操作，可以选择使用par集合。例如：

```
val urls = List("http://scala-lang.org",
    "http://agiledon.github.com")

def fromURL(url: String) = scala.io.Source.fromURL(url)
    .getLines().mkString("\n")

val t = System.currentTimeMillis()
urls.par.map(fromURL(_))
println("time: " + (System.currentTimeMillis - t) + "ms")
```

9) 若有多个操作需要并行进行异步操作，则采用for comprehension对future进行join方式的执行。例如，假设Cloud.runAlgorithm()方法返回一个Future[Int]，可以同时执行多个runAlgorithm方法：

```
val result1 = Cloud.runAlgorithm(10)
val result2 = Cloud.runAlgorithm(20)
val result3 = Cloud.runAlgorithm(30)

val result = for {
    r1 <- result1
    r2 <- result2
    r3 <- result3
} yield (r1 + r2 + r3)

result onSuccess {
    case result => println(s"total = $result")
}
```

编码模式

1) Loan Pattern: 确保打开的资源（如文件、数据库连接）能够在操作完后被安全的释放。

Loan Pattern的通用格式如下：

```
def using[A](r : Resource)(f : Resource => A) : A =
    try {
        f(r)
    } finally {
        r.dispose()
    }
```

这个格式针对Resource类型进行操作。还有一种做法是：只要实现了close方法，都可以运用Loan Pattern：

```
def using[A <: def close():Unit, B](resource: A)(f: A => B): B =
    try {
        f(resource)
    } finally {
        resource.close()
    }
```

以FileSource为例：

```
using(io.Source.fromFile("example.txt")) {
  source => {
    for (line <- source.getLines) {
      println(line)
    }
  }
}
```

2) Cake Pattern: 利用self type实现依赖注入

例如，对于DbAccessor而言，需要提供不同的DbConnectionFactory来创建连接，从而访问不同的Data Source。

```
trait DbConnectionFactory {
  def createDbConnection: Connection
}

trait SybaseDbConnectionFactory extends DbConnectionFactory...
trait MySQLDbConnectionFactory extends DbConnectionFactory...
```

运用Cake Pattern，DbAccessor的定义应该为：

```
trait DbAccessor {
  this: DbConnectionFactory =>

  //...
}
```

由于DbAccessor使用了self type，因此可以在DbAccessor中调用DbConnectionFactory的方法createDbConnection()。客户端在创建DbAccessor时，可以根据需要选择混入的DbConnectionFactory：

```
val sybaseDbAccessor = new DbAccessor with SybaseDbConnectionFactory
```

当然，也可以定义object：

```
object SybaseDbAccessor extends DbAccessor with SybaseDbConnectionFactory
object MySQLDbAccessor extends DbAccessor with MySQLDbConnectionFactory
```

测试

1) 测试类应该与被测试类处于同一包下。如果使用Spec2或ScalaTest的FlatSpec等，则测试类的命名应该为：被测类名 + Spec；若使用JUnit等框架，则测试类的命名为：被测类名 + Test

2) 测试含有具体实现的trait时，可以让被测试类直接继承Trait。例如：

```
trait RecordsGenerator {
  def generateRecords(table: List[List[String]]): List[Record] {
    //...
  }
}

class RecordsGeneratorSpec extends FlatSpec with ShouldMatcher with RecordsGenerator {
```

```

val table = List(List("abc", "def"), List("aaa", "bbb"))
it should "generate records" in {
    val records = generateRecords(table)
    records.size should be(2)
}

```

3) 若要对文件进行测试，可以用字符串假装文件：

```

type CsvLine = String
def formatCsv(source: Source): List[CsvLine] = {
    source.getLines(_.replace(", ", "|"))
}

```

formatCsv需要接受一个文件源，例如Source.fromFile("testdata.txt")。但在测试时，可以通过Source.fromString方法来生成formatCsv需要接收的Source对象：

```

it should "format csv lines" in {
    val lines = Source.fromString("abc, def, hgi\n1, 2, 3\none, two, three")
    val result = formatCsv(lines)
    result.mkString("\n") should be("abc|def|hgi\n1|2|3\none|two|three")
}

```

参考资料：

1. Scala Style Guide
2. [Programming in Scala](#), Martin Odersky
3. [Scala Cookbook](#), Alvin Alexander
4. [Effective Scala](#), Twitter