Scala 编码规范---by zhangyi(教练)

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 = maxListImpParm(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

OBJ

    // 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 +: l

}

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)

//suggesion
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 Futrue[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

完毕后被安全的释放。

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：被测试类名 + 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 RecordGenerator {
```

```
    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