scalaz 介绍

Tiger Chan

http://www.cnblogs.com/tiger-xc

http://blog.csdn.net/tiger_xc

- scalaz - 为什么?
- functional programming 泛函编程 - 为什么?
- 多核cpu、多并发、并行运算、分布式计算、大数据… 世界变得更复杂
- 软件愈来愈复杂、臃肿，现有的方式方法再无法掌控软件开发过程；难以有效进行程序编写、理解意图、维护代码、测试功能、安全运行
- 把程序分解成一堆微细的、功能唯一的配件，然后按需要对这些配件进行各种组合形成更大的组件，然后再对组件进行各种组合。。。
  - code-reusable、reasonable、testable、maintainable
  and most importantly: **composable**
- **泛函编程模式Functional programming paradigm**

- 函数组合 functional composition

- 实现方式： Referencial Transparency RT等量替换定律
  -> pure function 纯函数
  -> pure code 纯代码
  -> immutable data 不可变数据结构
  -> no side-effects 无副作用

- 副作用：对外界事物（the world）的依赖，like:
  update program state，system I/O …

- 无副作用：延迟副作用的产生，抽取出（refactor）有副作用代码放到最后运行（at the end of the world）

- Functional programming 是什么?
- Programming using scala?
- 使用头等函数特性 using function as first-class-value
  + 对不可变结构进行操作 operate on immutable structure
  + 返回不可变数据 return immutable reference?
- not quite there yet，还没到站哪！
- last stop: **函数组合**，各个层面、各种组合
  (**functional composition** at all levels)
- 泛函编程模式是一种全新的编程概念，需要彻底的、革命性的、从概念上的设计思想改变
- 个人看法，在现有的codebase基础上逐渐引入FP是不可取的

我所期待看到的FP程序：

⊕ : compose

val mod1 = func1 ⊕ func3 ⊕ func8

val mod2 = func5 ⊕ func3

val mod32 = mod2 ⊕ func6 ⊕ func7

val modrw = frd ⊕ fwr

(mod1 ⊕ mod32).run modrw.run

```
def system: Config[Application] =
  (Applicative build dataStore)
  .and (workerPool) apply (new Application(_,_))
```

- scalaz, 为什么?
- write pure code >>> write code functionally
- 什么是Functional programming?
- lambda calculus，category theory ?
- F[A]，一个运算computation
- F=算法 context，A＝运算对象 value type
- 需要一套全新的运算方式：数据类型、数据结构、函数方法
- 泛函类型：List[A],Option[A],Monoid[A],Monad[F[_]]…
- 泛函方法：—— map[A,B](fa:F[A])(f: A => B): F[B]
  ———————————-ap[A,B](fa: F[A])(f: F[A => B]): F[B]
  ————————-flatMap[A,B](fa: F[A])(f: A => F[B]): F[B] ….

- scalaz, 是什么?
- library of typeclass
- typeclass: haskell 泛函类库
- 用scala实现的haskell typeclass库
- scala版的泛函组件库
- 支持scala的泛函编程

- typeclass 介绍

- typeclass: library of ad-hoc polymorphic functions
  即兴多态函数库

- polymorphism 多态： function generalization 函数概括施用：同一函数可以施用在不同类型的对象上 - 代码重复使用

- 多态实现方式：

  1、overloading 重载
  2、inherentance 继承
  3、pattern-matching 模式匹配
  4、type parameter 类参数 >>> ad-hoc polymorphism >>> typeclass

重载 overload

```scala
object overloading {
  case class Color(scheme: String)
  case class Person(name: String)
  def tell(color: Color) = s"I'm Color ${color.scheme}"
  def tell(person: Person)=  s"I'm ${person.name}"
}
import overloading._
tell(Color("RED"))        //> res0: String = I'm Color RED
tell(Person("John"))      //> res1: String = I'm John
```

继承 inheritance

```scala
object inheritance {
  trait AnyThing
  case class Color(scheme: String) extends AnyThing {
    def tell: String = s"I'm Color ${scheme}"
  }
  case class Person(name: String) extends AnyThing {
    def tell: String = s"I'm ${name}"
  }
}
import inheritance._
Color("RED").tell                    //> res0: String = I'm Color RED
Person("John").tell                  //> res1: String = I'm John
```

模式匹配 pattern-matching

```scala
object patternmatch {
  case class Color(scheme: String)
  case class Person(name: String)
  def tell(a: Any): String = a match {
     case Color(sch) =>  s"I'm Color ${sch}"
     case Person(nm) => s"I'm ${nm}"
     case i: Int => s"I'm a Integer with value $i"
  }
}
import patternmatch._
tell(3)                    //> res0: String = I'm a Integer with value 3
tell(Color("RED"))         //> res1: String = I'm Color RED
tell(Person("Jonh"))       //> res2: String = I'm Jonh
```

类参数 typeclass

```
trait Tellable[A] {
  def tell(a: A): String
}
object Tellable {
  implicit object StringTellable extends Tellable[String] {
     def tell(s: String): String =  s"I'm a string of chars $s"
  }
  implicit val intTellable = new Tellable[Int] {
     def tell(i: Int): String = s"I'm integer $i"
  }
}
def tellAll[A](a: A)(implicit Teller: Tellable[A]): String = Teller.tell(a)
tellAll("hello world!")                       //> res0: String = I'm a string of chars hello world!
tellAll(64)                                   //> res1: String = I'm integer 64
```

```
case class Color(scheme: String)

implicit val personTellable = new Tellable[Person] {
  def tell(p: Person): String = s"I'm $p.name"
}
tellAll(Person("John"))                       //> res3: String = I'm Person(John).name
```

```
implicit val myIntTellable = new  Tellable[Int] {
 def tell(i: Int): String = s"my integer value is $i"
}
tellAll(24)                                   //> res5: String = my integer value is 24
```

building my typeclass demo

```scala
def sum0(xa: List[Int]): Int = xa.foldLeft(0){_ + _}
sum0(List(1,2,3))         //> res0: Int = 6
```

```scala
object intAdder {
  def mzero = 0
  def madd(x: Int, y: Int): Int = x + y
}
def sum1(xa: List[Int]): Int =  xa.foldLeft(intAdder.mzero)
(intAdder.madd)
sum1(List(1,2,3))         //> res1: Int = 6
```

building my typeclass demo

```
trait Addable[A] {
  val mzero: A
  def madd(x: A, y: A): A
}
case class Crew(names: List[String])
object Addable {
  implicit object intAddable extends Addable[Int] {
    def mzero = 0
    def madd(x: Int, y: Int) = x + y
  }
  implicit object strAddable extends Addable[String] {
    val mzero = ""
    def madd(x: String, y: String) = x + y
  }
  implicit object crewAddable extends Addable[Crew] {
    val mzero = Crew(List())
    def madd(x: Crew, y: Crew): Crew = Crew(x.names ++ y.names)
  }
  def apply[A](implicit M: Addable[A]): Addable[A] = M
}
def sum2[A](xa: List[A])(implicit M: Addable[A]): A = xa.foldLeft(M.mzero)(M.madd)
sum2(List(1,2,3))                                                 //> res2: Int = 6
sum2(List("ab","c","def"))                                        //> res3: String = abcdef

sum2(List(Crew(List("john")), Crew(List(“susan","peter")))) //> res4: Crew = Crew(List(john, susan, peter))
Addable[Crew].mzero          //> res5: Crew = Crew(List())
Addable[Crew].madd(Crew(List("john")), Crew(List(“ada")))   //> res6: Crew = Crew(List(john, ada))
```

building my typeclass demo

```
trait FoldLeft[F[_]] {
  def foldLeft[A,B](fa: F[A])(b: B)(f: (B,A) => B): B
}
object FoldLeft {
  implicit object listFold extends FoldLeft[List] {
    def foldLeft[A,B](fa: List[A])(b: B)(f: (B,A) => B) = fa.foldLeft(b)(f)
  }
  def apply[F[_]](implicit F: FoldLeft[F]): FoldLeft[F] = F
}
FoldLeft[List].foldLeft(List(1,2,3))(0)(_ + _)    //> res7: Int = 6

def sum3[A: Addable,F[_]: FoldLeft](fa: F[A]): A = {
  val adder = implicitly[Addable[A]]
  val folder = implicitly[FoldLeft[F]]
  folder.foldLeft(fa)(adder.mzero)(adder.madd)
}
//> sum3: [A, F[_]](fa: F[A])(implicit evidence$1: Addable[A], implicit evidenc$2:FoldLeft[F])A
sum3(List(Crew(List("john")), Crew(List("susan","peter"))))
            //> res8: Crew = Crew(List(john, susan, peter))
```

method injection 方法注入

```
class AddableOp[A](a: A)(implicit M: Addable[A]) {
  def |+|(y: A) = M.madd(a,y)
}
implicit def toAddableOp[A: Addable](a: A): AddableOp[A] = new
AddableOp[A](a)
3 |+| 2                                    //> res9: Int = 5
("hello" |+| " ") |+| "world!"             //> res10: String = hello world!
```

```
implicit class addableOp[A: Addable](a: A) {
   def |+|(y: A) = implicitly[Addable[A]].madd(a,y)
}
3 |+| 2                                    //> res9: Int = 5
("hello" |+| " ") |+| "world!"             //> res10: String = hello world!
```

method injection 方法注入

```
trait AddableOp[A] {
  val M: Addable[A]
  val x: A
  def |%| = M.mzero
  def |+|(y: A) = M.madd(x,y)
}
implicit def toAddableOps[A: Addable](a: A): AddableOp[A] = new AddableOp[A] {
  val M = implicitly[Addable[A]]
  val x = a
}
3 |+| 2                                                  //> res9: Int = 5
("hello" |+| " ") |+| "world!"                           //> res10: String = hello world!
3.|%|                                                    //> res11: Int = 0
"hi".|%|                                                 //> res12: String = ""
```

```
final class FoldLeftOps[F[_],A](self: F[A])(implicit F: FoldLeft[F]) {
   def foldl[B](z: B)(f: (B,A) => B) = F.foldLeft(self)(z)(f)
}
implicit def toFoldLeftOps[F[_]: FoldLeft, A](v: F[A])/*(implicit F: FoldLeft[F])*/: FoldLeftOps[F,A] =
  new FoldLeftOps[F,A](v)                                //> toFoldLeftOps: [F[_], A](v: F[A])(implicit F: 
demo.worksheet.typeclasses.Fo
                                                         //| ldLeft[F])demo.worksheet.typeclasses.FoldLeftOps[F,A]
List(1,2,3).foldl(0)(_ + _)                              //> res13: Int = 6
```

Functional Computation 泛函运算方法

typeclass	运算对象	函数款式	运算结果
Functor : map[A,B]	(F[A])	(f: A => B):	F[B]
Applicative : ap[A,B]	(F[A])	(f: F[A => B]):	F[B]
Monad : flatMap[A,B]	(F[A])	(f: A => F[B]):	F[B]
Traverse : traverse[G:Applicative,A,B]	(F[A])	(f: A => G[B]):	G[F[B]]

Functor: A => B

```
def liftToStrong(name: String) = name.toUpperCase+”!"
List("china","usa","japan").map(liftToStrong).map(print)
                                     //> CHINA!USA!JAPAN!res0: List[Unit] = List((), (), ())
case class Record(id: Int, content: String)
case class Cache[A](data: A)
implicit object cacheFunctor extends Functor[Cache] {
  def map[A,B](ca: Cache[A])(f: A => B): Cache[B] = Cache(f(ca.data))
}
val data = Cache[Record](Record(1,"I'm cached data”))
def markRecord(r: Record) = Record(r.id + 1000, r.content + " updated!”)
def saveToDB(r: Record) = println("saving record “+r.id)
data.map(markRecord).map(saveToDB)   //> saving record 1001 res1: Cache[Unit] = Cache(())
val listOfcache = List(Cache(Record(1,"rec1")),Cache(Record(2,"rec2")))
val listCacheFunctor = Functor[List] compose Functor[Cache]
val mdata = listCacheFunctor.map(listCacheFunctor.map(listOfcache)(markRecord))(saveToDB)
//> saving record 1001 saving record 1002
//> mdata:List[Cache[Unit]] List(Cache(()),Cache(()))
```

Applicative: F[A => B] >>> (F[A],F[B],F[C]…) => F[K]

```
trait Applicative[F[_]] extends Apply[F] { self =>
  def point[A](a: => A): F[A]
  override def map[A, B](fa: F[A])(f: A => B): F[B] =
    ap(fa)(point(f))
  override def apply2[A, B, C](fa: => F[A], fb: => F[B])(f: (A, B) => C): F[C] =
    ap2(fa, fb)(point(f))
```

```
trait Apply[F[_]] extends Functor[F] { self =>
  def ap[A,B]     (fa: => F[A])                              (f: => F[A => B]) :   F[B]
  def ap2[A,B,C]  (fa: => F[A], fb: => F[B])                 (f: F[(A,B) => C]):   F[C] =
    ap(fb)(ap(fa)(map(f)(_.curried)))
  def ap3[A,B,C,D](fa: => F[A], fb: => F[B], fc: => F[C])   (f: F[(A,B,C) => D]): F[D] =
    ap(fc)(ap2(fa,fb)(map(f)(f => ((a:A,b:B) => (c:C) => f(a,b,c)))))
  …
  def apply2[A, B, C]    (fa: => F[A], fb: => F[B])              (f: (A, B) => C):    F[C] =
    ap(fb)(map(fa)(f.curried))
  def apply3[A, B, C, D] (fa: => F[A], fb: => F[B], fc: => F[C])  (f: (A, B, C) => D): F[D] =
    apply2(tuple2(fa, fb), fc)((ab, c) => f(ab._1, ab._2, c))
  …
   def lift2[A, B, C]  (f: (A, B) => C):      (F[A], F[B]) => F[C] =
    apply2(_, _)(f)
  def lift3[A, B, C, D](f: (A, B, C) => D):   (F[A], F[B], F[C]) => F[D] =
    apply3(_, _, _)(f)
  …
```

Applicative: F[A => B] >>> (F[A],F[B],F[C]…) => F[K]

```
val getsLen: String => Int = s => s.length         //> getsLen  : String => Int = <function1>
getsLen("abcd")                                    //> res4: Int = 4
val liftedLen = getsLen.point[List]                //> liftedLen  : List[String => Int] = List(<function1>)
Apply[List].ap(List("abcd"))(liftedLen)            //> res5: List[Int] = List(4)
```

```
trait Config[A] { def get: A }
object Config {
  def apply[A](a: A) = new Config[A] { def get = a }
  implicit val configFunctor = new Functor[Config] { def map[A,B](ca: Config[A])(f: A => B) = Config(f(ca.get)) }
  implicit  val confApplicative = new Applicative[Config] {
      def point[A](a: => A) = Config(a)
      def ap[A,B](ca: => Config[A])(cfab: => Config[A => B]) =cfab map (_(ca.get)) }
}
```

```
def map_[A,B](ca: Config[A])(f: A => B): Config[B] = Apply[Config].ap(ca)(f.point[Config])
def incr(i: Int): Int = i + 1                            //> incr: (i: Int)Int
Apply[Config].ap(Config(3))((incr _).point[Config]).get  //> res6: Int = 4
^(Config(1),Config(2)){_ + _}.get                        //> res7: Int = 3
^^(Config(1),Config(2),Config(3)){_ + _ + _}.get         //> res8: Int = 6
(Config(1) |@| Config(2) |@| Config(3)){_ + _ + _}.get   //> res9: Int = 6
(Config("hello") |@| Config(" ") |@| Config("world")) {_ + _ + _}.get   //> res10: String = hello world
def greeting(hi: String, sp: String, w: String) = hi + sp + w
val configGreet = Apply[Config].lift3(greeting _)
    //> configGreet  : (Config[String],Config[String], Config[String]) => Config[String] = <function3>
configGreet(Config("hello"),Config(" "), Config("world!")).get         //> res11: String = hello world!
```

Applicative: F[A => B] >>> (F[A],F[B],F[C]…) => F[K]

```
case class WebLogForm(usr: String, id: String, pwd: String)
def getUsr: Config[String] = Config("usr")        //> getUsr: => Config[String]
def getId: Config[String] = Config("id")          //> getId: => Config[String]
def getPwd: Config[String] = Config("pwd")        //> getPwd: => Config[String]
^^(getUsr,getId,getPwd)(WebLogForm(_,_,_))
      //> res12: Config[WebLogForm] = $$anonfun$main$1$Config$3$$anon$4@359f7cdf
(getUsr |@| getId |@| getPwd)((a,b,c) => WebLogForm(a,b,c))
       //> res13:Config[WebLogForm] = $$anonfun$main$1$Config$3$$anon$4@1fa268de
```

```
import java.sql.DriverManager
val connection = java.sql.DriverManager.getConnection("src","usr","pwd")

val sqlConnection = Apply[Config].lift3(java.sql.DriverManager.getConnection)
//> sqlConnection: (Config[String], Config[String], Config[String]) =>
Config[java.sql.Connection] = <function3>

val conn = sqlConnection(Config("Source"),Config("User"),Config("Password"))
```

Monad: A => F[B]

```
trait Monad[F[_]] extends Applicative[F] with Bind[F] { self =>
  override def map[A,B](fa: F[A])(f: A => B) = bind(fa)(a => point(f(a)))
…
```

```
trait Bind[F[_]] extends Apply[F] { self =>
  def bind[A, B](fa: F[A])(f: A => F[B]): F[B]
  override def ap[A, B](fa: => F[A])(f: => F[A => B]): F[B] = {
    val fa0 = Need(fa)
    bind(f)(x => map(fa0.value)(x))
  }
  def join[A](ffa: F[F[A]]) = bind(ffa)(a => a)
…
```

```
final class BindOps[F[_],A] private[syntax](val self: F[A])(implicit val F: Bind[F]) extends
Ops[F[A]] {
  def flatMap[B](f: A => F[B]) = F.bind(self)(f)
  def >>=[B](f: A => F[B]) = F.bind(self)(f)
  def join[B](implicit ev: A <~< F[B]): F[B] = F.bind(self)(ev(_))
…
```

Monad: A => F[B]

```scala
trait Config[A] { def get: A }
object Config {
  def apply[A](a: A) = new Config[A] { def get = a }
  implicit val confMonad = new Monad[Config] {
    def point[A](a: => A) = Config(a)
    def bind[A,B](ca: Config[A])(f: A => Config[B]) = f(ca.get)
  }
}
```

```scala
List(List(1,2),List(10,20)).flatMap(x => x.map(a => a))       //> res14: List[Int] = List(1, 2, 10, 20)
(Config("hello") >>= { hi => Config(" ") >>= {
  sp => Config("World").map { world => hi + sp + world }}}).get  //> res15: String = hello World
(Config(3) >>= (a => Config(2).map (b => a + b))) get          //> res16: Int = 5
(for {
  a <- Config(3)
  b <- Config(2)
  c <- Config(a+b)
} yield (c)).get                                                 //> res17: Int = 5
```

```scala
    fa.flatMap(a => fb.flatMap(b => fc.flatMap(c => fd.map(...))))
    for {
       a <- (fa: F[A])
       b <- (fb: F[A])
       c <- (fc: F[A])
    } yield { ... }
```

Traverse: (F[A])(f: A => G[B]) >>> G[F[B]]

```
traverse[G:Applicative,A,B](F[A])(f: A => G[B]): G[F[B]]
sequence[G:Applicative,A](F[G[A]]): G[F[A]]
```

```
trait Book
trait Author
import concurrent._
def books(isbn: ISBN): Future[Book] = ???
                  //> books: (isbn: ISBN)scalaz.concurrent.Future[Book]
def listFutureBooks(isbns: List[ISBN]): List[Future[Book]] = isbns.map(books)
   //> listFutureBooks: (isbns: List[ISBN])List[scalaz.concurrent.Future[Book]]

def futureListBooks(isbns: List[ISBN]): Future[List[Book]] = isbns.traverse(books)
   //> futureListBooks: (isbns: List[ISBN])scalaz.concurrent.Future[List[Book]]

def futureListBooks_s(isbns: List[ISBN]): Future[List[Book]]  =
     listFutureBooks(authors).sequence
 //> futureListBooks_s: (isbns: List[ISBN])scalaz.concurrent.Future[List[Book]]
```

scala - library of typeclasses

- Functor、Applicative、Monad、Traverse
- Monoid
- State Monad
- Reader Monad
- Writer Monad
- Monad Transformer
- Lenses
- Free Monad、IO Monad and more …

Monad: Programming

```
Functor     :  map[A,B]     (F[A])  (f:    A => B) :  F[B]
Applicative:  ap[A,B]      (F[A])  (f: F[A => B]):  F[B]
Monad       :  flatMap[A,B] (F[A])  (f: A => F[B]):  F[B]
```

```
// a = e0
// b = e1(a)
// c = e2(a,b)        >>> imperative programming  行令编程
// d = e1(c)
```

```
def e0: Id[Int] = 3                              //> e0: => scalaz.Scalaz.Id[Int]
def e1(a: Int): Id[Int] = a + 2                  //> e1: (a: Int)scalaz.Scalaz.Id[Int]
def e2(a: Int, b: Int): Id[Int] = a + b          //> e2: (a: Int, b: Int)scalaz.Scalaz.Id[Int]
for {
 a <- e0
 b <- e1(a)          >>> imperative programming inside a monad  在算法内进行的行令编程
 c <- e2(a,b)
} yield c                                        //> res18: scalaz.Scalaz.Id[Int] = 8

def e0: Option[Int] = 3.some                     //> e0: => Option[Int]
def e1(a: Int): Option[Int] = (a + 2).some       //> e1: (a: Int)Option[Int]
def e2(a: Int, b: Int): Option[Int] = (a + b).some   //> e2: (a: Int, b: Int)Option[Int]
for {
 a <- e0
 b <- e1(a)          >>> imperative programming inside a monad  在算法内进行的行令编程
 c <- e2(a,b)
} yield c                                        //> res18: Option[Int] = Some(8)
```

Monad: Programming

```
def e0: Option[Int] = 3.some                            //> e0: => Option[Int]
def e1(a: Int): Option[Int] = (a + 2).some             //> e1: (a: Int)Option[Int]
def e2(a: Int, b: Int): Option[Int] = (a + b).some    //> e2: (a: Int, b: Int)Option[Int]
def e3: Option[Int] = none[Int]                        //> e3: => Option[Int]
for {
  a <- e0
  b <- e1(a)
  z <- e3          >>> monad with flow control effect 在流程控制算法内进行的行令编程
  c <- e2(a,b)
} yield c                                              //> res18: Option[Int] = None
```

Functional Program == Imperative program in a Monad

```
fa.flatMap(a => fb.flatMap(b => fc.flatMap(c => fd.map(...))))
for {
   a <- (fa: F[A])
   b <- (fb: F[A])
   c <- (fc: F[A])
} yield { ... }
```

Free Monad: 函数数据化

```scala
trait Free[S[_],A]
case class Return[S[_],A](a: A) extends Free[S,A]
case class FlatMap[S[_],A,B](fa: Free[S,A], f: A => Free[S,B]) extends Free[S,B]
case class Suspend[S[_],A](s: S[A]) extends Free[S,A]

def liftF[S[_], A](value: S[A]): Free[S, A] = Suspend(value)
```

```scala
sealed trait MyFree[F[_],A]
object MyFree {
  final case class Return[F[_],A](a: A) extends MyFree[F,A]
  final case class Suspend[F[_],A](fa: F[A]) extends MyFree[F,A]
  final case class FlatMap[F[_],A,B](fa: MyFree[F,A], f: A => MyFree[F,B])
        extends MyFree[F,B]

  def point[F[_],A](a: A) = Return[F,A](a)
  def bind[F[_],A,B](fa: MyFree[F,A])(f: A => MyFree[F,B]): MyFree[F,B] =
      FlatMap(fa,f)
  def map[F[_],A,B](fa: MyFree[F,A])(f: A => B): MyFree[F,B] =
      bind(fa)(f andThen (Return(_)))
}
```

ADT-Algebraic Data Type 代数数据类型 >>> 程序语法

```
F[A] >>> ADT >>> 一项操作描述 >>> 语法
List[F[A]] >>> 一系列连续操作描述 >>> 一段程序功能描述 >>> AST-Agebraic Syntax Tree
```

```
trait PrintMsg[A]
case class PrintLine(msg: String) extends PrintMsg[Unit]
val printHello: List[PrintLine] = List(PrintLine("Hello"),PrintLine(" world!"))
               //> printHello  : List[PrintLine] = List(PrintLine(Hello), PrintLine( world!))
printHello.map(a => print(a.msg))            //> Hello world!res19: List[Unit] = List((), ())
```

```
sealed trait Interact[A]
case class Ask(prompt: String) extends Interact[String]
case class Tell(msg: String) extends Interact[Unit]

val prg = List(Ask(“What’s your first name?”),
               Ask(“What’s your last name?”),
               Tell(“Hello, ???”))

val prg = for {
   first <- Ask(“What’s your first name?”)
   last  <- Ask(“What’s your last name?”)
   _     <- Tell(s”Hello, $first $last!”)
}
```

AST - Monadic Program 功能描述 >>> 算式

```scala
sealed trait Dialog[A]
case class Ask(prompt: String) extends Dialog[String]
case class Tell(msg: String) extends Dialog[Unit]
implicit def liftToFree[A](da: Dialog[A]) = Free.liftF(da)

val ast = for {
  first <- Ask("What's your first name?")
  last <- Ask("what's your last name?")
  _ <- Tell(s"Hello $first $last!")
} yield()
```

```scala
val ast: Free[Dialog, Unit] =
 Ask("What's your first name?").bind(first =>
     Ask("What's your last name?”).bind(last =>
        Tell(s"Hello, $first $last!”).map(_ => ())))
```

```scala
val ast: Free[Dialog, Unit] =
 FlatMap(Ask("What's your first name?"), first =>
          FlatMap(Ask("What's your last name?"), last =>
                   FlatMap( Tell(s"Hello, $first $last!"), _ => Return(()))))
```

Interpreter - Transform & Run 编译运行：从功能描述到具体实现 >>> 算法

F[A] ~> G[A] ~> NaturalTransformation

```
sealed trait ~>[F[_],G[_]] { def apply[A](f: F[A]): G[A] }
```

```
object DialogConsole extends (Dialog ~> Id) {
  def apply[A](da: Dialog[A]): Id[A] = da match {
    case Ask(prompt) => println(prompt); readLine
    case Tell(msg) => println(msg)
} }
```

```
object Interact extends App {
  import Interacts._
  ast.foldMapRec(DialogConsole)
}
```

```
final def foldMapRec[M[_]](f: S ~> M)(implicit M: Applicative[M], B: BindRec[M]): M[A] =
  B.tailrecM[Free[S, A], A]{ _.step match {
      case Return(a) => M.point(\/-(a))
      case Suspend(t) => M.map(f(t))(\/.right)
      case b @ Gosub() => (b.a: @unchecked) match {
        case Suspend(t) => M.map(f(t))(a => -\/(b.f(a))) } }
  }(this)
…
 def tailrecM[A, B](f: A => Trampoline[A \/ B])(a: A): Trampoline[B] =
    f(a).flatMap(_.fold(tailrecM(f), point(_)))
…
 type Trampoline[A] = Free[Function0, A]
```

switch Interpreter - separation of concern 算式 / 算法的关注分离

```
type Input = String   type Output = String
emulating Console I/O  >>> Map[Output,Input] >>> Output1++Output2, Input(ask prompt as index)

final case class WriterT[F[_], W, A](run: F[(W, A)]) { self =>  … }
```

```
   type WF[A] = Map[String,String] => A
   type DialogTester[A] = WriterT[WF,List[String],A]
   def testerToWriter[A](f: Map[String,String] => (List[String],A)) =
      WriterT[WF,List[String],A](f)
   implicit val testerMonad = WriterT.writerTMonad[WF,List[String]]
   object DialogTester extends (Dialog ~> DialogTester) {
     def apply[A](da: Dialog[A]): DialogTester[A] = da match {
       case Ask(prompt) => testerToWriter {m => (List(),m(prompt))}
       case Tell(msg) => testerToWriter {m => (List(msg),()) }
     }
   }
```

```
object Interact extends App {
  import Interacts._
 // ast.foldMapRec(DialogConsole)
  val result = ast.foldMapRec(DialogTester).run(
      Map (
          "What's your first name ?" -> "Tiger",
          "what's your last name ?" -> "Chan"
      ))
   println(result._1)
}     // Hello Tiger Chan !
```

A Sample Program for Free

```
Functions (ADTs):
 Interact         >>> get user id; get password
 Login            >>> check password from authentication system developed by others
 Permission       >>> check permission from access system developed by others
```

```
val authScript = for {
  uid <- ask[T,String]("what's you id?",identity)
  idok <- checkId[T](uid)
  _ <- if (idok) tell[T](s"hi, $uid")
       else tell[T]("sorry, don't know you!")
  pwd <- if (idok) ask[T,String](s"what's your password?",identity)
         else Free.point[T,String]("")
  login <- if (idok) login[T](uid,pwd)
         else Free.point[T,Boolean](false)
    _ <- if (login) tell[T](s"congratulations, $uid")
         else tell[T](idok ? "sorry, no pass!" | "")
  acc <- if (login) ask[T,Int](s"what's your access code, $uid?",_.toInt)
         else Free.point[T,Int](0)
  perm <- if (login) hasPermission[T](uid,acc)
          else Free.point[T,Boolean](false)
  _ <- if (perm) tell[T](s"you may use the system, $uid")
         else tell[T]((idok && login)  ? "sorry, you are banned!" | "")

} yield ()
```

Functor ADT

```scala
sealed trait Interact[+A]
case class Ask[A](prompt: String, onInput: String => A) extends Interact[A]
case class Tell[A](msg: String, next: A) extends Interact[A]
sealed trait InteractInstances {
  object InteractFunctor extends Functor[Interact] {
    def map[A,B](ia: Interact[A])(f: A => B): Interact[B] = ia match {
      case Ask(prompt,input) => Ask(prompt, input andThen f)
      case Tell(msg,next) => Tell(msg, f(next))
    }
  }
}
sealed trait InteractFunctions {
  def ask[G[_],A](p: String, f: String => A)(implicit I: Inject[Interact,G]): Free[G,A] =
    Free.liftF(I.inj(Ask(p,f)))
  def tell[G[_]](m: String)(implicit I: Inject[Interact,G]): Free[G,Unit] =
    Free.liftF(I.inj(Tell(m,Free.pure(()))))
}
```

```scala
val interactScript = for {
  first <- ask("what's your first name?",identity)
  last <- ask("your last name?",_.toUpperCase())
  _ <- tell(s"hello, $first $last")
} yield ()
```

```scala
object InteractConsole extends (Interact ~> Id) {
  def apply[A](ia: Interact[A]): Id[A] = ia match {
    case Ask(p,onInput) => println(p); onInput(readLine)
    case Tell(m,n) => println(m); n
  }
}
```

```scala
interactScript.foldMapRec(InteractConsole)
```

Inject ADT 注入语法 >>> 扩大语句集

```
val prg: Free[???,Boolean] = for {
  uid <- ask("your id:")
  pwd <- ask("password:")
  ok <- login(uid,pwd)
} yield ok
```

```
> Multi-effect Monad >>> Multi-ADT AST >>> 多种语法支持
> Tailored Monad e.g ReaderWriterState Monad
> Monad Transformer (F[A] and F[B]), Coproduct (F[A] or F[B])
```

```
case class Coproduct[F[_],G[_],A](run: Either[F[A],G[A]])
sealed trait Inject[F[_],G[_]] {
  def inj[A](sub: F[A]): G[A]
  def prj[A](sup: G[A]): Option[F[A]]
}
object Inject {
  implicit def injRefl[F[_]] = new Inject[F,F] { … }
  implicit def injLeft[F[_],G[_]] =
    new Inject[F,({type λ[α] = Coproduct[F,G,α]})#λ] { … }
  implicit def injRight[F[_],G[_],H[_]](implicit I: Inject[F,G]) =
    new Inject[F,({type f[x] = Coproduct[H,G,x]})#f] { … }
}
```

Lift and Inject into Coproduct

```scala
sealed trait InteractFunctions {
  def ask[G[_],A](p: String, f: String => A)(implicit I: Inject[Interact,G]): Free[G,A] =
    Free.liftF(I.inj(Ask(p,f)))
  def tell[G[_]](m: String)(implicit I: Inject[Interact,G]): Free[G,Unit] =
    Free.liftF(I.inj(Tell(m,Free.pure(())))) }

sealed trait LoginFunctions {
  def checkId[G[_]](uid: String)(implicit I: Inject[UserLogin,G]): Free[G,Boolean] =
    Free.liftF(I.inj(CheckId(uid)))
  def login[G[_]](uid: String, pswd: String)(implicit I: Inject[UserLogin, G]): Free[G,Boolean] =
    Free.liftF(I.inj(Login(uid,pswd))) }
```

```scala
type InteractLogin[A] = Coproduct[Interact,UserLogin,A]
val loginScript = for {
  uid <- ask[InteractLogin,String]("what's you id?",identity)
  idok <- checkId[InteractLogin](uid)
  _ <- if (idok) tell[InteractLogin](s"hi, $uid") else tell[InteractLogin]("sorry, don't know you!")
  pwd <- if (idok) ask[InteractLogin,String](s"what's your password?",identity)
         else Free.point[InteractLogin,String]("")
  login <- if (idok) login[InteractLogin](uid,pwd)
         else Free.point[InteractLogin,Boolean](false)
  _ <- if (login) tell[InteractLogin](s"congratulations, $uid")
         else tell[InteractLogin](idok ? "sorry, no pass!" | "")
} yield login
```

Dependency Injection Using Reader

```
type Reader[E, A] = ReaderT[Id, E, A]
type ReaderT[F[_], E, A] = Kleisli[F, E, A]
final case class Kleisli[M[_], A, B](run: A => M[B]) { self => … }
object Reader {
  def apply[E, A](f: E => A): Reader[E, A] = Kleisli[Id, E, A](f)
}
case class Reader[A,B](run: A => B) { … }
```

```
object Dependencies {
  trait UserControl {
    val pswdMap: Map[String,String]
    def validateId(uid: String): Boolean
    def validatePassword(uid: String, pswd: String): Boolean
  }
  trait AccessControl {
    val accMap: Map[String, Int]
    def grandAccess(uid: String, acc: Int): Boolean
  }
  trait Authenticator extends UserControl with AccessControl
}
```

Interpret Coproduct 编译语句集

```scala
type AuthReader[A] = Reader[Authenticator,A]
object InteractLogin extends (Interact ~> AuthReader) {
  def apply[A](ia: Interact[A]): AuthReader[A] = ia match {
    case Ask(p,onInput) => println(p); Reader {m => onInput(readLine)}
    case Tell(msg,n) => println(msg); Reader {m => n}
  }
}
object LoginConsole extends (UserLogin ~> AuthReader) {
  def apply[A](ua: UserLogin[A]): AuthReader[A] = ua match {
    case CheckId(uid) => Reader {m => m.validateId(uid)}
    case Login(uid,pwd) => Reader {m => m.validatePassword(uid, pwd)}
  }
}
```

```scala
//选择F或H其中一种语法
def or[F[_],H[_],G[_]](f: F~>G, h: H~>G) =
    new (({type l[x] = Coproduct[F,H,x]})#l ~> G) {
      def apply[A](ca: Coproduct[F,H,A]):G[A] = ca.run match {
        case -\/(fg) => f(fg)
        case \/-(hg) => h(hg)
      }
  }
```

Running Coproduct Program 运算多语法程序时注入依赖

```scala
object AuthControl extends Authenticator {
  val pswdMap = Map (
    "Tiger" -> "1234",
    "John" -> "0000"
  )
 override def validateId(uid: String) =
   pswdMap.getOrElse(uid,"???") /== "???"
 override def validatePassword(uid: String, pswd: String) =
    pswdMap.getOrElse(uid, pswd+"!") === pswd

 val accMap = Map (
   "Tiger" -> 8,
   "John" -> 0
 )
 override def grandAccess(uid: String, acc: Int) =
   accMap.getOrElse(uid, -1) > acc
}


loginScript.foldMapRec(or(InteractLogin,LoginConsole)).run(AuthControl)
```

A 3 ADT Coproduct Example 由三种语法组成的程序 － ADT

```scala
sealed trait Interact[+A]
case class Ask[A](prompt: String, onInput: String => A) extends Interact[A]
case class Tell[A](msg: String, next: A) extends Interact[A]

sealed trait InteractInstances {
  object InteractFunctor extends Functor[Interact] {
    def map[A,B](ia: Interact[A])(f: A => B): Interact[B] = ia match {
      case Ask(prompt,input) => Ask(prompt, input andThen f)
      case Tell(msg,next) => Tell(msg, f(next))
    }
  }
}
```

```scala
sealed trait UserLogin[+A]  // None Functor 高阶类
case class CheckId(uid: String) extends UserLogin[Boolean]
case class Login(uid: String, pswd: String) extends UserLogin[Boolean]
```

```scala
sealed trait Permission[+A]
case class HasPermission(uid: String, acc: Int) extends Permission[Boolean]
```

A 3 ADT Coproduct Example 由三种语法组成的程序 – lifting

```scala
sealed trait InteractFunctions {
  def ask[G[_],A](p: String, f: String => A)(implicit I: Inject[Interact,G]): Free[G,A] =
    Free.liftF(I.inj(Ask(p,f)))
  def tell[G[_]](m: String)(implicit I: Inject[Interact,G]): Free[G,Unit] =
    Free.liftF(I.inj(Tell(m,Free.pure(()))))
}
object Interacts extends InteractInstances with InteractFunctions
```

```scala
sealed trait LoginFunctions {
  def checkId[G[_]](uid: String)(implicit I: Inject[UserLogin,G]): Free[G,Boolean] =
    Free.liftF(I.inj(CheckId(uid)))
  def login[G[_]](uid: String, pswd: String)(implicit I: Inject[UserLogin, G]): Free[G,Boolean] =
    Free.liftF(I.inj(Login(uid,pswd)))
}
object Logins extends LoginFunctions
```

```scala
sealed trait PermissionFunctions {
  def hasPermission[G[_]](uid: String, acc: Int)(implicit I: Inject[Permission,G]): Free[G,Boolean] =
    Free.liftF(I.inj(HasPermission(uid,acc)))
}
object Permissions extends PermissionFunctions
```

A 3 ADT Coproduct Example 由三种语法组成的程序 － AST

```scala
  type InteractLoginPermission[A] = Coproduct[Permission,InteractLogin,A]
  type T[A] = InteractLoginPermission[A]
  val authScript = for {
    uid <- ask[T,String]("what's you id?",identity)
    idok <- checkId[T](uid)
    _ <- if (idok) tell[T](s"hi, $uid")
         else tell[T]("sorry, don't know you!")
    pwd <- if (idok) ask[T,String](s"what's your password?",identity)
           else Free.point[T,String]("")
    login <- if (idok) login[T](uid,pwd)
           else Free.point[T,Boolean](false)
      _ <- if (login) tell[T](s"congratulations, $uid")
           else tell[T](idok ? "sorry, no pass!" | "")
    acc <- if (login) ask[T,Int](s"what's your access code, $uid?",_.toInt)
           else Free.point[T,Int](0)
    perm <- if (login) hasPermission[T](uid,acc)
            else Free.point[T,Boolean](false)
    _ <- if (perm) tell[T](s"you may use the system, $uid")
           else tell[T]((idok && login)  ? "sorry, you are banned!" | "")

  } yield ()
}
```

A 3 ADT Coproduct Example 由三种语法组成的程序 – Interpret & Run

```scala
object AuthControl extends Authenticator {
  val pswdMap = Map ( "Tiger" -> "1234","John" -> "0000")
  override def validateId(uid: String) =  pswdMap.getOrElse(uid,"???") /== "???"
  override def validatePassword(uid: String, pswd: String) = pswdMap.getOrElse(uid, pswd+"!") === pswd

  val accMap = Map ("Tiger" -> 8,"John" -> 0)
  override def grandAccess(uid: String, acc: Int) =
   accMap.getOrElse(uid, -1) > acc
}

authScript.foldMapRec(among3(InteractLogin,LoginConsole,PermConsole)).run(AuthControl)
```

```scala
def among3[F[_],H[_],K[_],G[_]](f: F~>G, h: H~>G, k: K~>G) = {
  type FH[A] = Coproduct[F,H,A]
  type KFH[A] = Coproduct[K,FH,A]
  new (({type l[x] = Coproduct[K,FH,x]})#l ~> G) {
    def apply[A](kfh: KFH[A]): G[A] = kfh.run match {
      case -\/(kg) => k(kg)
      case \/-(cfh) => cfh.run match {
        case -\/(fg) => f(fg)
        case \/-(hg) => h(hg)
      }
    }
  }
}
```

A Simple I/O Monad

```
no meaningful program without side-effect such as I/O
we have to make effectful functions composable
```

```scala
trait MyIO[+A] { self =>
  def run: A
  def map[B](f: A => B): MyIO[B] = new MyIO[B] { def run = f(self.run) }
  def flatMap[B](f: A => MyIO[B]): MyIO[B] = new MyIO[B] { def run = f(self.run).run }
}
object MyIO {
  def apply[A](a: A) = new MyIO[A] { def run = a }
  implicit val ioMonad = new Monad[MyIO] {
     def point[A](a: => A) = new MyIO[A] { def run = a }
     def bind[A,B](ma: MyIO[A])(f: A => MyIO[B]): MyIO[B] = ma flatMap f
  } }
```

```scala
def ask(prompt: String): MyIO[String] = MyIO { println(prompt); readLine }
def tell(msg: String): MyIO[Unit] = MyIO { println(msg) }

 val prg: MyIO[Unit] = for {
   first <- ask("What's your first name?")
   last <- ask("What's your last name?")
   _ <- tell(s"Hello $first $last!")
 } yield()

 pre.run
```

A Simple I/O Monad - Composing Functions

```
def efctfun1: MyIO[Unit] = ???
def efctfun2: MyIO[Unit] = ???
def efctfun3: MyIO[Unit] = ???
def efctfun4: MyIO[Unit] = ???
def efctfun5: MyIO[Unit] = ???
```

```
val prg13: MyIO[Unit] = for {
  x <- efctfun1
  y <- efctfun3
} yield()
val prg135: MyIO[Unit] = for {
  _ <- prg13
  x <- efctfun5
} yield()
val cmpScript = for {
  _ <- efctfun1
  _ <- prg135
  _ <- efctfun4
  _ <-prg13
} yield()

cmpScript.run
```

scalaz IO Monad

```
sealed abstract class IO[A] {
  private[effect] def apply(rw: Tower[IvoryTower]): Trampoline[(Tower[IvoryTower], A)]
  def unsafePerformIO(): A = apply(ivoryTower).run._2
…
```

```
val hello = print("Hello").point[IO]    //> hello:IO[Unit] = scalaz.effect.IO$$anon$6@145eaa29
val world = IO { println(" world!") }  //> world:IO[Unit] = scalaz.effect.IO$$anon$6@57c758ac
val howareyou = io {rw => return_(rw -> println("how are you!"))}
                   //> howareyou  : scalaz.effect.IO[Unit] = scalaz.effect.IO$$anon$6@a9cd3b1
val greet = hello |+| world |+| howareyou  //> greet:IO[Unit] = scalaz.effect.IO$$anon
greet.unsafePerformIO()                    //> Hello world! how are you!
```

```
def div(dvdn: Int, dvsor: Int): IO[Int] = IO(dvdn / dvsor)
val ioprg: IO[Int] = for {
  _ <- putLn("enter dividend:")
  dvdn <- readLn
  _ <- putLn("enter divisor:")
  dvsor <- readLn
  quot <- div(dvdn.toInt, dvsor.toInt)
  _ <- putLn(s"the result:$quot")
} yield quot
ioprg.unsafePerformIO()
```

scalaz IO Monad

Considerations:
1、Stack safe runner
2、Flow control
3、Error handling
4、Logging

```scala
def div(dvdn: Int, dvsor: Int): IO[Int] = IO(dvdn / dvsor)

val ioprg: IO[Int] = for {
   _ <- putLn("enter dividend:")
   dvdn <- readLn
   _ <- putLn("enter divisor:")
   dvsor <- readLn
   quot <- div(dvdn.toInt, dvsor.toInt)
   _ <- putLn(s"the result:$quot")
} yield quot

ioprg.unsafePerformIO()
```

scalaz IO Monad - Flow Control

```
Multiple effects achieved by MonadTransformer
final case class OptionT[F[_], A](run: F[Option[A]]) { … }
OptionT[IO,Int] >>> IO[Option[Int]]
```

```
val optionIOprg: OptionT[IO,Int] = for {
  _ <- putLn("enter dividend:").liftM[OptionT]
  dvdn <- readLn.liftM[OptionT]
  _ <- putLn("enter divisor:").liftM[OptionT]
  dvsor <- readLn.liftM[OptionT]
  a <- if (dvsor.toInt == 0 ) OptionT(IO(None: Option[String]))
          else IO(0).liftM[OptionT]
  quot <- div(dvdn.toInt, dvsor.toInt).liftM[OptionT]
  _ <- putLn(s"the result:$quot").liftM[OptionT]
} yield quot

optionIOprg.unsafePerformIO()
```

scalaz IO Monad - Error Handling & Logging

```
Logging achieved by MonadTransformer WriterT
final case class WriterT[F[_], W, A](run: F[(W, A)]) { … }
WriterT[IO,List[String],Int] >>> IO[Writer[List[String],Int]]
```

```
type WriterTIO[F[_],A] = WriterT[F,List[String],A]
val writerIOprg: WriterT[IO,List[String],Int] = for {
  _ <- putLn("enter dividend:").liftM[WriterTIO]
  dvdn <- readLn.liftM[WriterTIO]
  _ <- WriterT.writerT((List(s"received dividend $dvdn;"),dvdn).point[IO])
  _ <- putLn("enter divisor:").liftM[WriterTIO]
  dvsor <- readLn.liftM[WriterTIO]
  _ <- WriterT.writerT(IO(List(s"received divisor $dvsor, ready to divide ..."),dvdn))
  quot <- div(dvdn.toInt, dvsor.toInt).except(e =>
                                IO({println(e.getMessage());-99})).liftM[WriterTIO]
  _ <- if (quot < 0) WriterT.writerT((List(s"divide by zero Error!!!"),-99).point[IO])
           else putLn(s"the result:$quot").liftM[WriterTIO]
} yield (quot)

 optionIOprg.unsafePerformIO()
```

Thank you !

谢谢！