MONADIC IN SCALA

屈鉴铭

WHY

SCALA HISTORY

2001

Martin Odersky 基于 Funnel 语言为原型, 开始设计一种针对 Web Service 的, 集函数式特性与面向对象特性为一体的程序语言

2004

Scala 的 Java 平台和 .Net 平台 1.0 版本先后发布

2006

Scala 2.0 版本问世. 其编译器由 Scala 编写. 之后, 被需求驱动持续发布多个包含新特性的子版本.

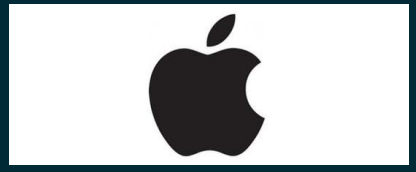
2012

Scala 2.10 版本, 支持了隐式转换, 宏与反射.

2014

Scala.js 0.1 版本发布.

WHO IS USING SCALA



WHY SHOULD I LEARN SCALA?

WRITE CONCISE AND CLEAR CODE

```
case class Person(firstName: String, lastName: String)
```

SCALABLE

Where the name `Scala` came from.

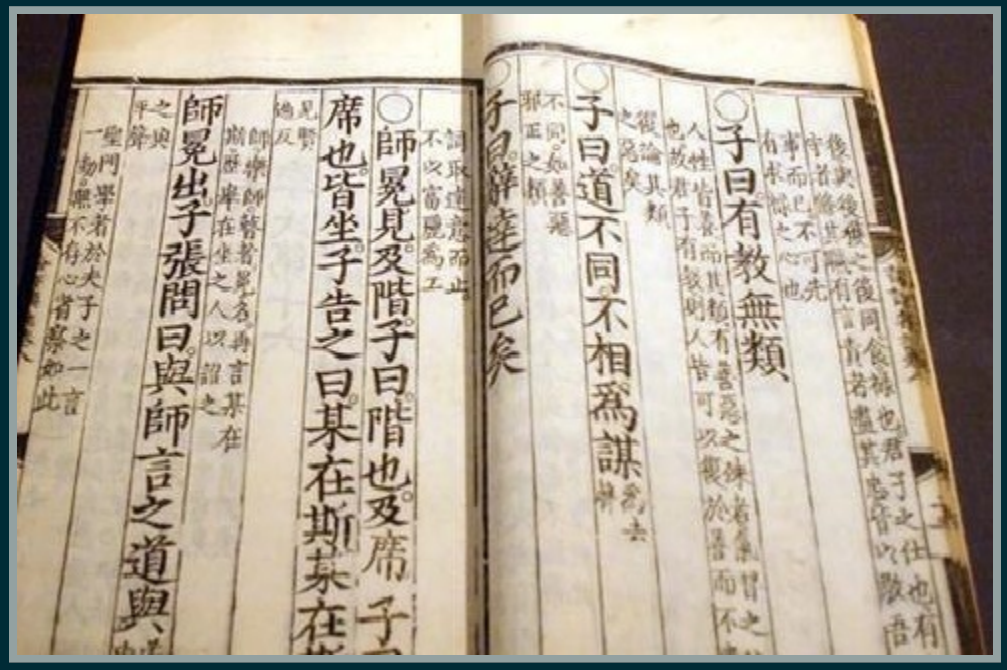
EASY TO MAINTAIN

1. IMMUTABLE

Find the lady (three-card monte)

2. (RELATIVELY) READABLE

Document OR Source Code?



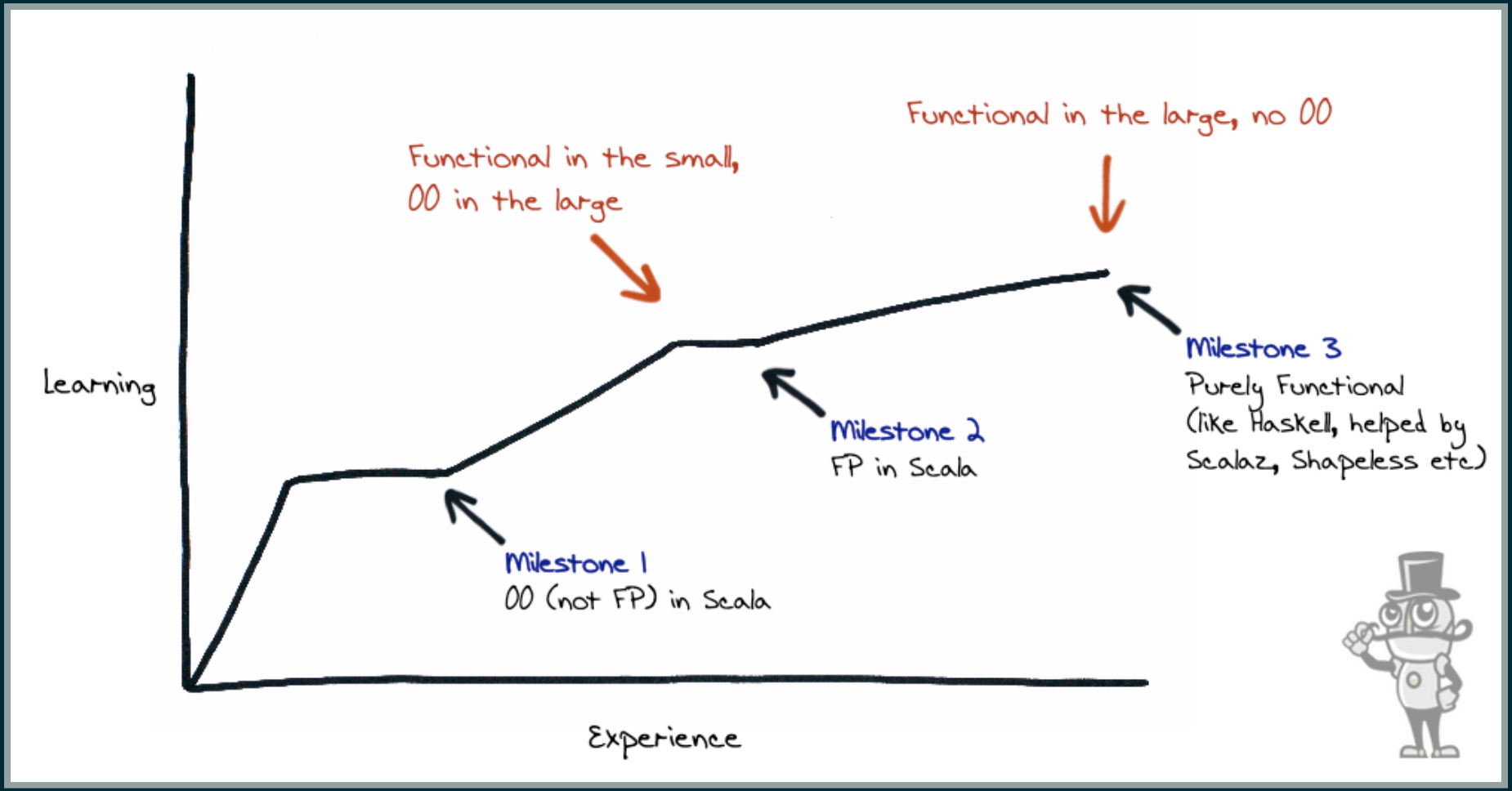
PLATFORM AND LIBRARIES SUPPORT

- All the advantages of running on JVM (Diagnostic, Tools, GC, etc.)
- All the libraries in Java ecosystem.
- Lots of awesome Scala libraries.

LISTEN TO US

LESS WELL

STEEP LEARNING CURVE



FLEXIBLE

- **Flexibility** comes at the price of **Simplicity**.
- So many **Concepts** & **Features**.

CONFLICT WITH JAVA LIBRARY

WORTH TO KNOW, WORTH TO LEARN

You might fall in love with it.

MONADIC PROGRAMMING

通过 Chainable 的语言风格以 **纯函数** 的形式来描述对数据的处理流程.

WHAT'S EFFECT?

```
var count = 0;
val mkEffect = (input1: Int) => {
  count = count + 1
  println(s"The current count is $count")
  val input2 = readLine()
  input1 + Integer.parseInt(input2)
}
```

没有 EFFECT 我们还需要 MONADIC PROGRAMMING 吗?

WHAT'S MONAD?

A SCARY DEFINITION:

Philip Wadler :

> *Monad 是自函子范畴上的一个含幺半群*

TYPE CONSTRUCTOR

Java 里的泛型是一阶类型构造器 (first-order type) :

```
class List<T> {}
```

在 Scala 里这样表示

```
class List[T] {}
```

在 Scala 里还支持高阶类型构造器

```
class List[F[_]] {}
```

MONAD: 一类物理特性和逻辑特性相同的数据结构的统称

MONAD 的物理特性

```
object Monad[F[_]] {
  def pure[A](a: => A): F[A]                                  // po
  def flatMap[A, B](fa: F[A])(f: A => F[B]): F[B] // bi
}
```

F 是一个 Monad, 则可以定义成员函数

```
class F[A] {
  def flatMap[A, B](f: A => F[B]): F[B] =
    Monad[F].flatMap[A, B](this)(f)
}
```

MONAD 的逻辑特性 (MONAD LAWS)

- Left identity (左同一律)

```
val f: A => F[B] = ???
val a: A = ???
Monad[F].pure[A](a).flatMap(f) === f(a)
```

- Right identity (右同一律)

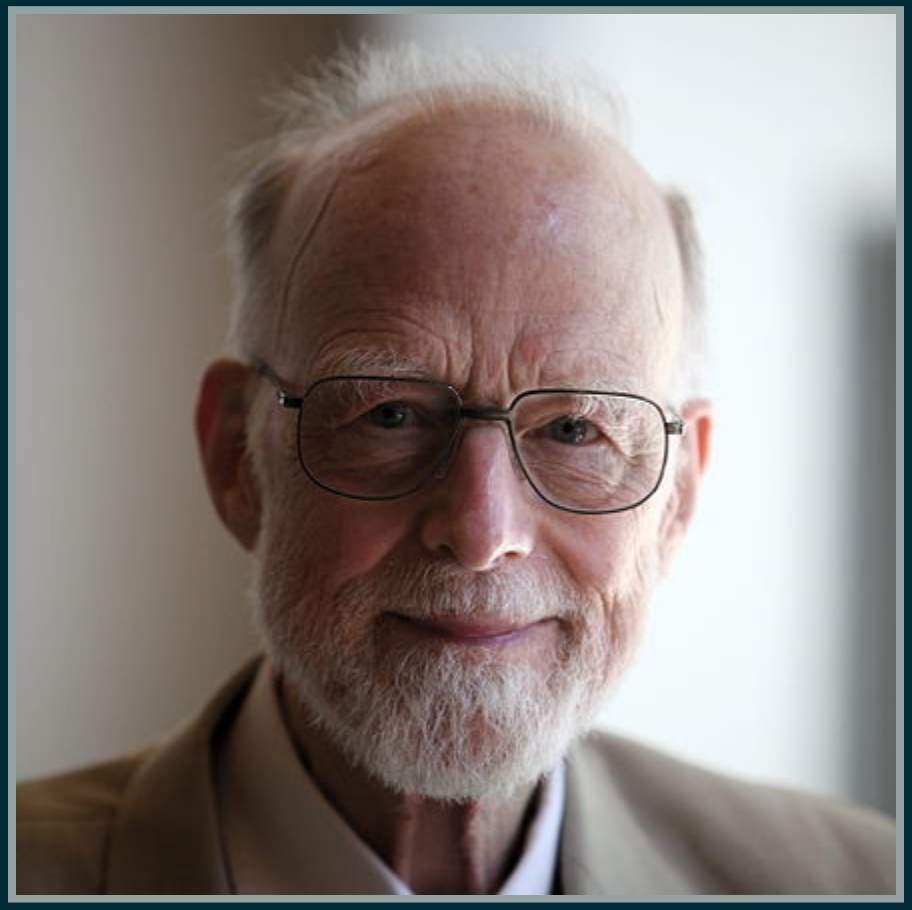
```
val m: F[A] = ???
m.flatMap(Monad[F].pure) === m
```

- Associativity (结合律)

```
val m: F[A] = ???
val f: A => F[B] = ???
val g: B => F[C] = ???
m.flatMap(f).flatMap(g) ===
  m.flatMap( a => f(a).flatMap(g) )
```

MONAD SAMPLES

A BILLION-DOLLAR MISTAKE



用户给出一个类型为 A 的值, 但也有可能什么都不给, 不用 null 该如何表示?

OPTION[A]

```
trait Option[+A]
case class Some[A](value: A) extends Option[A]
object None extends Option[Nothing]

def root(i: Int): Option[Int] = ???
def square(i: Int): Option[Int] = ???
Some(4).flatMap(root) === root(4) // 左同一律
Some(4).flatMap(Some.apply) === Some(4) // 右同一律
Some(4).flatMap(root).flatMap(square) ===  //结合律
  Some(4).flatMap(four => root(four).flatMap(square))
```

怎么用?

考虑有两个值,

- 第一个值时, 结果为空
- 第一个值不为空, 第二个值为空时, 结果为空
- 第一个和第二个值都不为空时, 结果为两个值的和

OPTION 的用法实例

```
val value1: Option[Int] = ???
val value2: Option[Int] = ???

value1.flatMap { v1 =>
  value2.flatMap { v2 =>
    v1 + v2
  }
}
```

更 scala 的写法是

```
for {
v1 <- value1
v2 <- value2
} yield v1 + v2
```

EITHER (DISJUNCTION, XOR)

```
trait Either[+A, +B]
class Left[A](value: A) extends Either[A, Nothing]
class Right[B](value: B) extends Either[Nothing, B]
```

考虑场景, 可能出错的多个有序处理过程,

- 所有过程不出错, 则完成处理, 并输出结果
- 任何一个过程出错, 则中断后续处理, 并返回错误

EITHER 的用法实例

```
def userInputName: Either[Error, String] = ???

def findInvoiceFromDatabaseBy(name: String):
  Either[Error, Invoice] = ???

def getJsonFromInvoice(invoice: Invoice):
  Either[Error, Json] = ???

val json = for {
  name <- userInputName
  invoice <- findInvoiceFromDatabaseBy(name)
  json <- getJsonFromInvoice(invoice)
} yield json
```

READER

```scala
class Reader[A, B](run: A => B)

def ask[B]: Reader[B, B] =
  Reader[B, B](identity[B])

def pure[A, B](b: B): Reader[A, B] =
  Reader[A, B](_ => b)
```

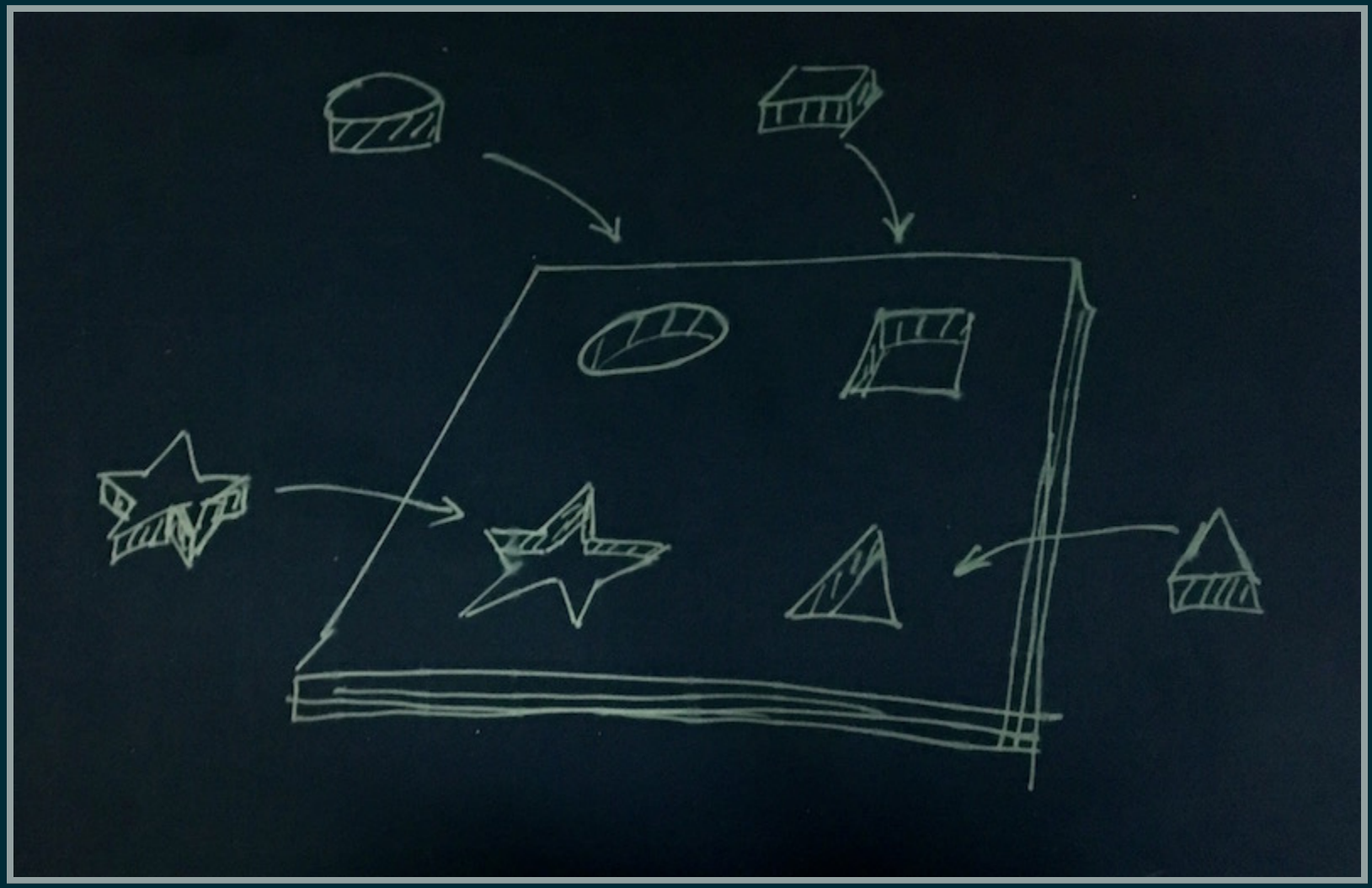
考虑, 我们需要一个从某处获取的配置文件, 并以此为基础做后续处理

READER 的用法实例

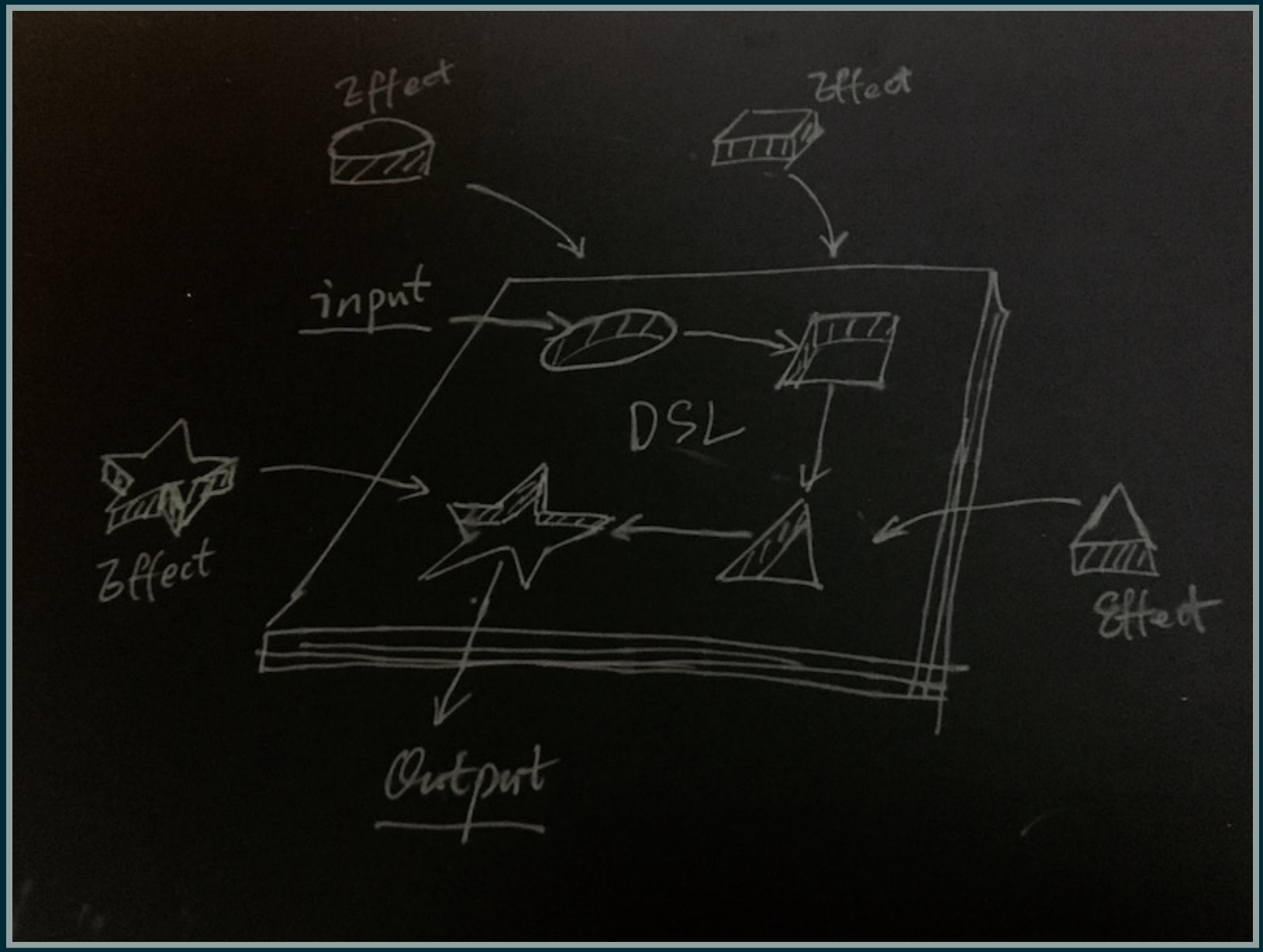
```
def getApiEndpoint: Reader[Config, URI] =
  ask[Config].flatMap(config =>
    pure[Config, URI](config.endpoint)
)
def getCountFromApi(uri: URI): Reader[Config, Int] =
  pure[Config, Int](syncGet(uri+"/count"))

val count = getApiEndpoint.flatMap { uri =>
  getCountFromApi(uri)
}.run(config)
```

FREE & INTERPRETER PATTERN



FREE & INTERPRETER PATTERN



好处:

- 处理逻辑的解耦
- 高可重用

COMPOSABLE MONAD

MONAD TRANSFORMER

```
def getApiEndpoint:
  Reader[Config, Either[Error, URI]] = ???

def getCountFromApi(uri: URI):
  Reader[Config, Either[Error, Int]] = ???

for {
  eitherUri <- getApiEndpoint
  eitherCount <- eitherUri match {
    case Left(error) =>
      pure[Config, Either[Error, URI]](eitherUri)
    case Right(uri) =>
      getCountFromApi(uri)
  }
} yield eitherCount
```

EITHERT

```
def getApiEndpoint:
  EitherT[Reader[Config, ?], Error, URI] = ???

def getCountFromApi(uri: URI):
  EitherT[Reader[Config, ?], Error, Int] = ???

for {
  uri <- getApiEndpoint
  count <- getCountFromApi(uri)
} yield count
```

WHAT ABOUT MORE?

- Eff

```
def getApiEndpoint[
  R : Reader[Config, ?] MemberIn ?]: Eff[R, URI] =
  send[Reader[Config, ?], R, URI](???)

def getCountFromApi[
  R : Either[Error, ?] MemberIn ?](uri: URI): Eff[R, In
  send[Either[Error, ?], R, Int](???)

val count: Eff[R: Reader[Config, ?] MemberIn ?
               : Either[Error, ?] MemberIn ?] = for {
  uri <- getApiEndpoint
  count <- getCountFromApi(uri)
} yield count
```

MONADIC IN PRODUCTION

FP LIBRARY

- cats
- scalaz

MONADIC FRAMEWORK

Play, Akka, Unfiltered, Scalatra

OTHER LIBRARY

https://github.com/lauris/awesome-scala

MONADIC IN OTHER LANGUAGES

- JS: monet.js, lodash/fp
- Ruby: monads

SHOULD WE USE IT?

THANK YOU

Bye.