**范畴论完全装逼手册**

很多人都不明白什么是Monad，并不是因为不会用，不知觉可能就在用某种 monad。  
定义和使用起来其实不难，困惑的大多应该是后面的这堆理论– 范畴论。当然，我也没学过范畴论，只是略微看得懂写Haskell罢了。

我在书中[写过一章](https://book.douban.com/subject/26883736/)来解释，某人也尝试过很[写博客解释](http://www.ruanyifeng.com/blog/2017/02/fp-tutorial.html)，比如为了降低门槛用JS来，那Haskell/Scala的人出来喷你们前端这些不懂函数式的渣渣乱搞出来的东西根本就不是 monad。

我也画过一些图来解释，又会被嫌弃画风不好。但是，作为灵魂画师，我只  在乎画的灵魂是否能够给你一点启发。好吧，讲这么学术的东西，还是用dot来画吧，看起来好正规呢。

好了，安全带系好，我真的要开车了。为了  解释的不到位，就用 Haskell 好了（虽然haskell也没到鄙视链顶），其实也不难解释清楚 。

这里面很多很装逼的单词，它们都是 *斜体* ，就算没看懂，把这些词记住也足够。

## 第一部分：

### *Category*

一个  包含两个玩意

* 东西 O （Object）
* 两个东西的关系，箭头 ~> （  ）

一些属性

* 一定有一个叫 id 的箭头，也叫做 1
* 箭头可以

恩，就是这么简单

Figure 1: 有东西 a, b, c 和箭头 f, g 的 Category，其中 f . g 表示 compose f 和 g

这些玩意对应到 haskell 的 typeclass 大致就是这样

class Category (c :: \* -> \* -> \*) where

id :: c a a

(.) :: c y z -> c x y -> c x z

*注意到为什么我会箭头从右往左，你会发现这个方向跟 compose 的方向刚好一致*

如果这是你第一次见到 Haskell 代码，没有关系，语法真的很简单

* class 定义了一个 TypeClass， Category 是这个 TypeClass 的名字
* Type class 类似于定义类型的规范，规范为 where 后面那一坨
* 类型规范的对象是参数 (c:: \* -> \* -> \*) ， :: 后面是c的类型
* c 是 *higher kind* ，跟higher order function的定义差不多，它是接收类型，构造新类型的类型。这里的 c 接收一个类型，再接收一个类型，就可以返回个类型。
* id:: c a a 表示 c 范畴上的 a 到 a 的箭头
* . 的意思 c 范畴上，如果喂一个 y 到 z 的箭头，再喂一个 x 到 y 的箭头，那么就返回 x 到 z 的箭头。

?还没有高数抽象呢。

#### *Hask*

Haskell 类型系统范畴叫做 Hask

在 Hask 范畴上：

* 东西是类型
* 箭头是类型的变换，即 ->
* id 就是 id 函数的类型 a -> a
* compose 当然就是函数组合的类型

type Hask = (->)

instance Category (Hask:: \* -> \* -> \*) where

(f . g) x = f (g x)

我们看见新的关键字 instance ，这表示 Hask 是 Type class Category 的实例类型，也就是说我们可以Hask的个构造器去真的构造一个类型

比如：

(->) a a

就构造了一个从a类型到a类型的的类型

构造出来的这个类型可以作为 id 函数的类型

id :: (->) a a

#### *Duel*

每个 Category还有一个镜像，什么都一样，除了箭头是反的

两个范畴中间可以用叫 Functor 的东西来连接起来，简称 T。

Figure 2: Functor C D T, 从 C 到 D 范畴的Functor T

所以大部分把Functor/Monad比喻成盒子其实在定义上是错的，虽然这样比喻比较容易理解，在使用上问题也不大。但是，Functor只是从一个范畴到另一个范畴的映射关系而已。

* 范畴间 东西的 Functor 标记为 T(O)
* 范畴间 箭头的 Functor 标记为 T(~>)
* 任何范畴C上存在一个 T 把所有的 O 和 ~> 都映射到自己，标记为id functor 1C
  + 1C(O) = O
  + 1C(~>) = ~>

class (Category c, Category d) => Functor c d t where

fmap :: c a b -> d (t a) (t b)

Functor c d t 这表示从范畴 c 到范畴 d 的一个 Functor t

如果把范畴 c 和 d 都限制到 Hask 范畴

class Functor (->) (->) t where

fmap :: (->) a b -> (->) (t a) (t b)

-> 在 Haskell 中是中缀类型构造器，所以是可以写在中间的

这样就会变成我们熟悉的 Funtor 的 Typeclass（把Functor 的第一第二个参数去掉的话）

class Functor t where

fmap :: (a -> b) -> (t a -> t b)

而  就是这种连接相同范畴的 Functor，因为它从范畴 Hask 到达同样的范畴 Hask

这里的 fmap 就是 T(~>)，在 Hask 范畴上，所以是 T(->), 这个箭头是函数，所以也能表示成 T(f) 如果 f:: a -> b

当我们把一个Category看成一个object，functor看成箭头，那么我们又得到了一个Category，这种object是category的category我们叫它 – *Cat*

已经没法讲了，看 TODO 图吧

Functor 是范畴间的映射，而 Functor 在 Cat 范畴又是个箭头，所以，Functor间的映射，也就是 Cat 范畴上的 Functor，叫做 *自然变换*

Figure 3: Functor F和G，以及 F 到 G 的自然变化 η

所以范畴 c 上的函子 f 到 g 的自然变化就可以表示成

type Nat c f g = c (f a) (g a)

Hask 范畴上的自然变化就变成了

type NatHask f g = f a -> g a

有趣的是，自然转换也满足箭头的概念，可以当成 functor 范畴上的箭头，所以又可以定义出来一个 Functor Catergory

* 东西是函子
* 箭头是自然变换

要成为范畴，还有两点

* id 为 f a 到 f a 的自然变换
* 自然变换的组合

我们来梳理一下，已经不知道升了几个维度了，我们假设类型是第一维度

* 一维： Hask， 东西是类型，箭头是 ->
* 二维： Cat， 东西是 Hask， 箭头是 Functor
* 三维： Functor范畴， 东西是Functor， 箭头是自然变换

感觉到达三维已经是极限了，尼玛还有完没完了，每升一个维度还要起这么多装逼的名字，再升维度就要一脸懵逼了呢。虽然维度不算太高，但是已经不能用简单的图来描述了，所以需要引入 String Diagram。

### String Diagram

String Diagram 的概念很简单，就是点变线线变点。

当有了自然变换之后，没法表示了呀，那原来的点和线都升一维度，变成线和面，这样，就腾出一个点来表示自然变换了。

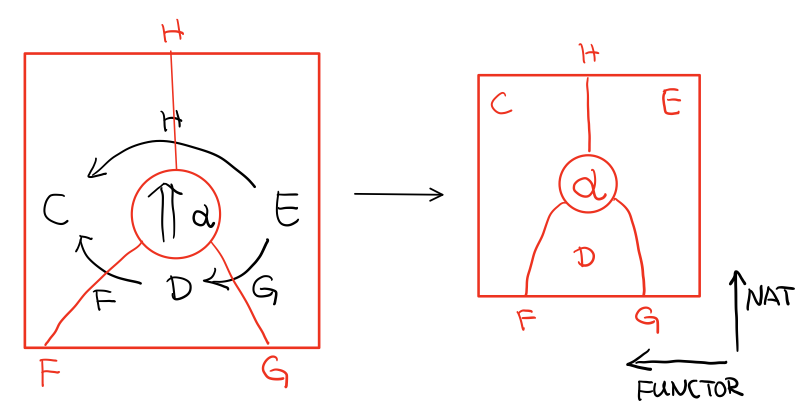


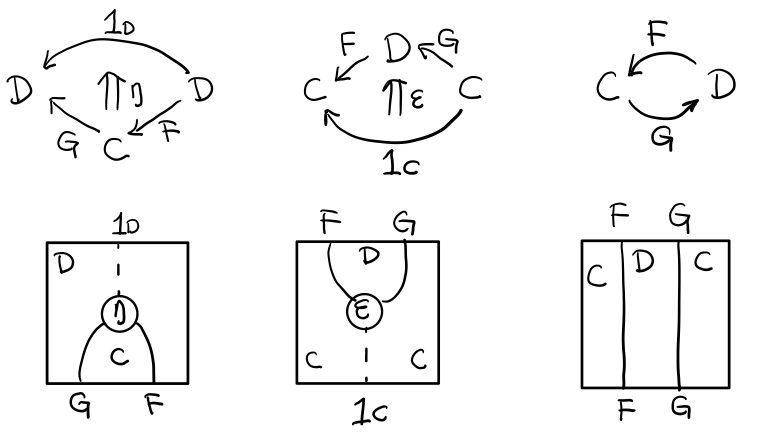
Figure 5: String Diagram：自然变换是点，Functor是线，范畴是面

compose的方向是从右往左，从下到上。

### *Adjunction Functor* 伴随函子

范畴C和D直接有来有回的函子，为什么要介绍这个，因为它直接可以推出 Monad

让我们来看看什么叫有来回。



其中：

* 一个范畴 C 可以通过函子 G 到 D，再通过函子 F 回到 C，那么 F 和 G 就是伴随函子。
* η 是 GF 到 1D 的自然变换
* ε 是 1C 到 FG 的自然变换

同时根据同构的定义，G 与 F 是 *同构* 的。

同构指的是若是有

f :: a -> b

f':: b -> a

那么 f 与 f' 同构，因为 f . f' = id = f' . f

伴随函子的 FG 组合是 C 范畴的 id 函子 F . G = 1c

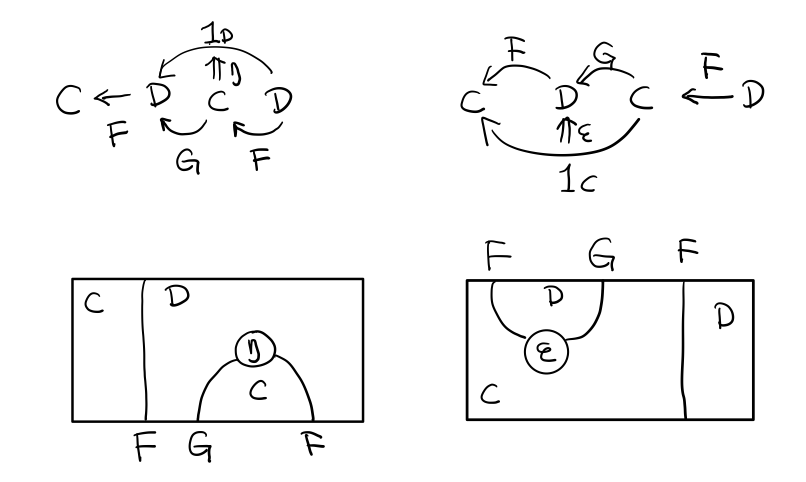


Figure 7: 伴随函子的两个Functor组合, 左侧为 F η, 右侧为 ε F

Functor 不仅横着可以组合，竖着(自然变换维度)也是可以组合的，因为自然变换是 Functor 范畴的箭头。

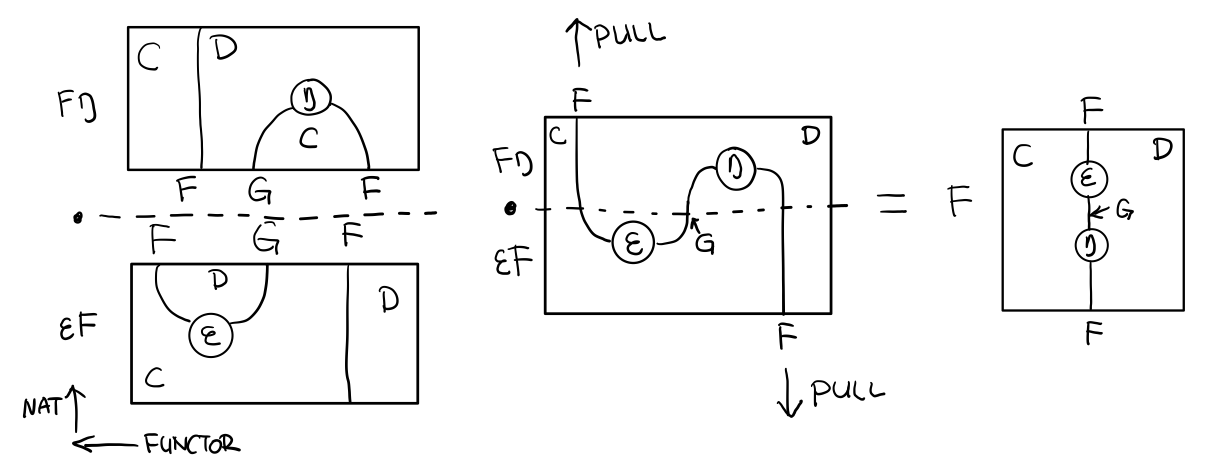


Figure 8: F η . ε F = F

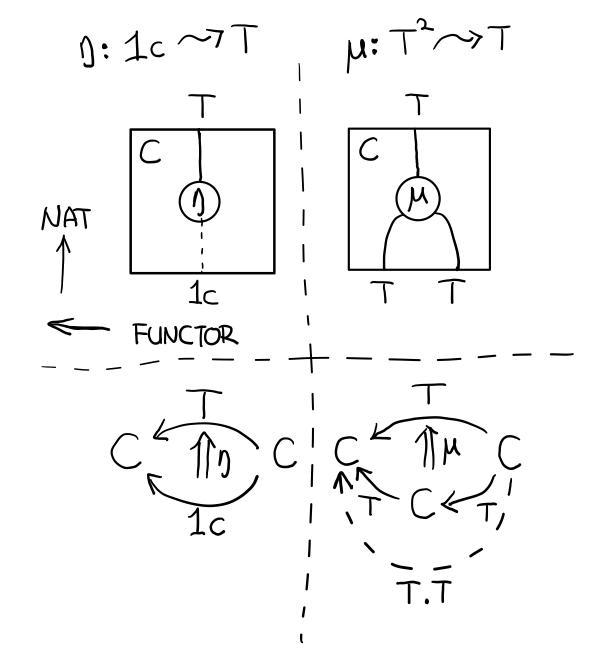
当到组合 F η . ε F 得到一个弯弯曲曲的 F 时，我们可以拽着F的两段一拉，就得到了直的 F。

String Diagram 神奇的地方是所有线都可以拉上下两端，这个技巧非常有用，在之后的单子推导还需要用到。

### 从伴随函子到

有了伴随函子，很容易推出单子，让我们先来看看什么是单子

* 首先，它是一个 endofunctor T
* 有一个从 ic 到 T 的自然变化 η (eta)
* 有一个从 T2 到 T 的自然变化 μ (mu)



class Endofunctor c t => Monad c t where

eta :: c a (t a)

mu :: c (t (t a)) (t a)

同样，把 c = Hask 替换进去，就得到更类似我们 Haskell 中 Monad 的定义

class Endofunctor m => Monad m where

eta :: a -> (m a)

mu :: m m a -> m a

要推出单子的 η 变换，只需要让 FG = T

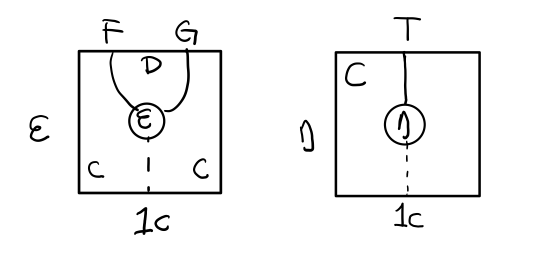


Figure 10: 伴随函子的 ε 就是单子的 η

同样的，当 FG = T, F η G 就可以变成 μ

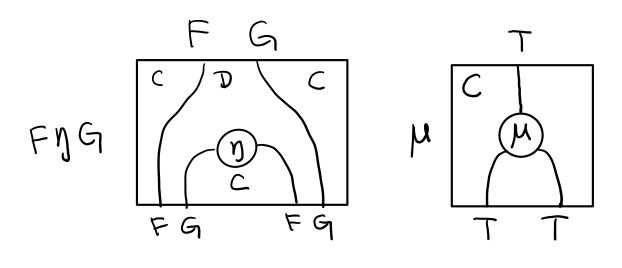


Figure 11: 伴随函子的 F η G 是函子的 μ

#### 三角等式

三角等式是指 μ . T η = T = μ . η T

要推出三角等式只需要组合 F η G 和 ε F G

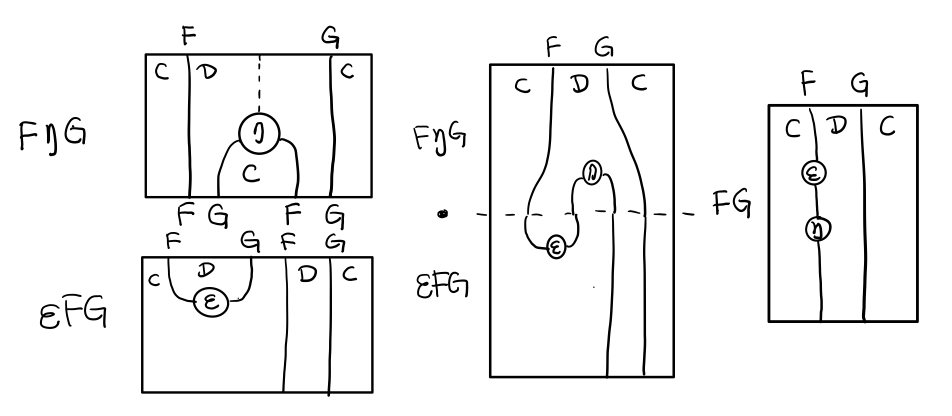


Figure 12: F η G . ε F G = F G

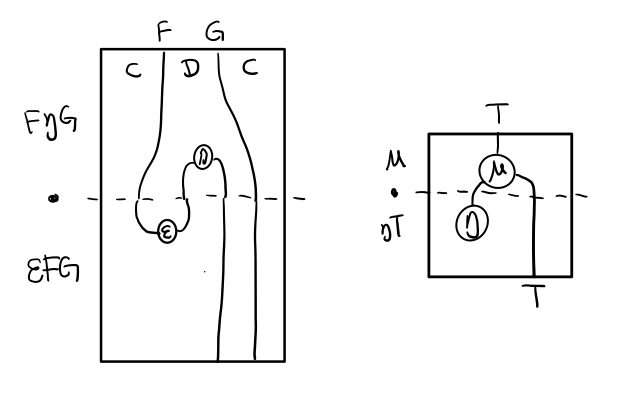


Figure 13: F η G . ε F G= F G 对应到Monad就是 μ . η T = T

换到代码上来说

class Endofunctor m => Monad m where

(mu . eta) m = m

同样的，左右翻转也成立

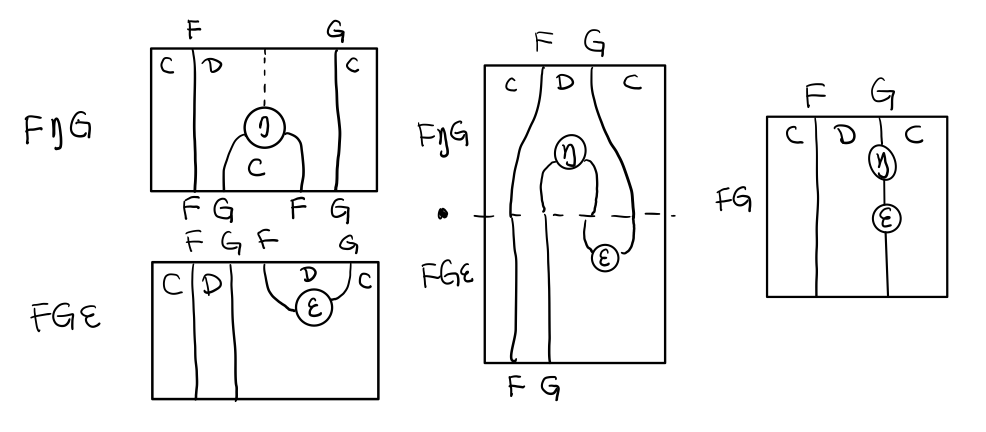


Figure 14: F η G . F G ε = F G

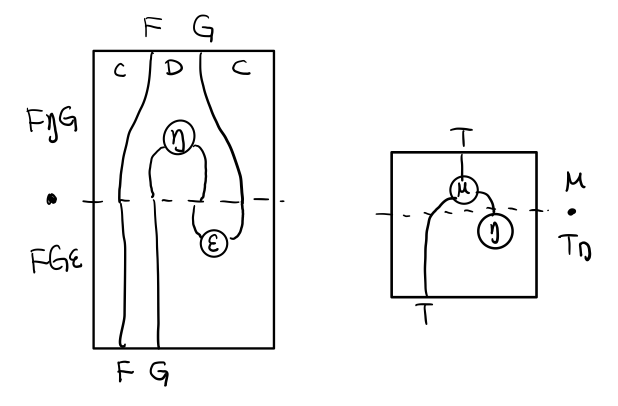


Figure 15: F η G . F G ε = F G 对应到 Monad是 μ . T η = T

T η 就是 fmap eta

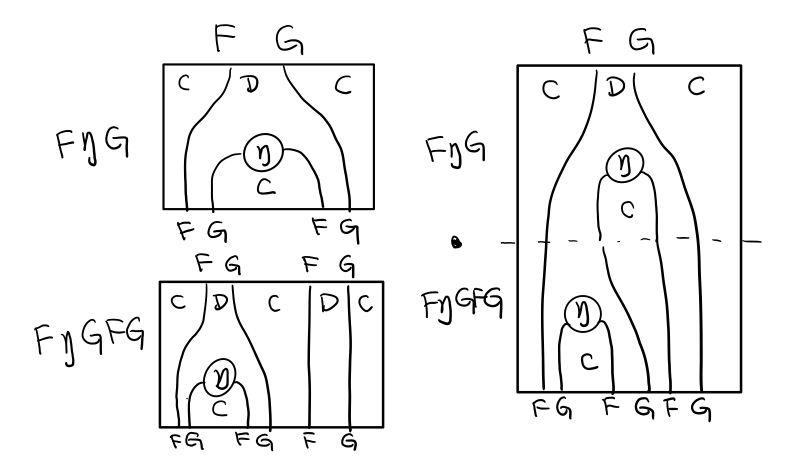
(mu . fmap eta) m = m

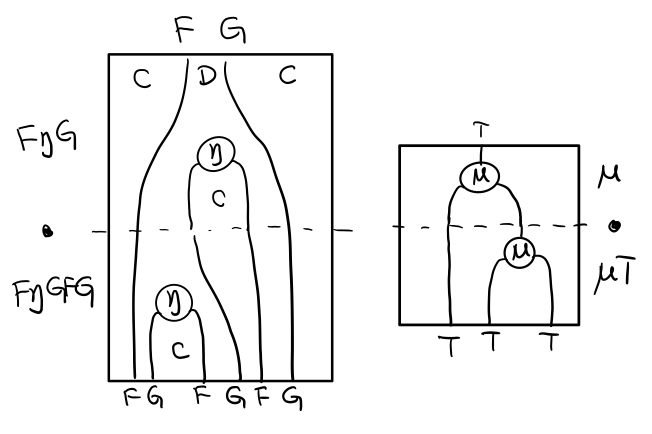
如果把 mu . fmap 写成 >>= , 就有了

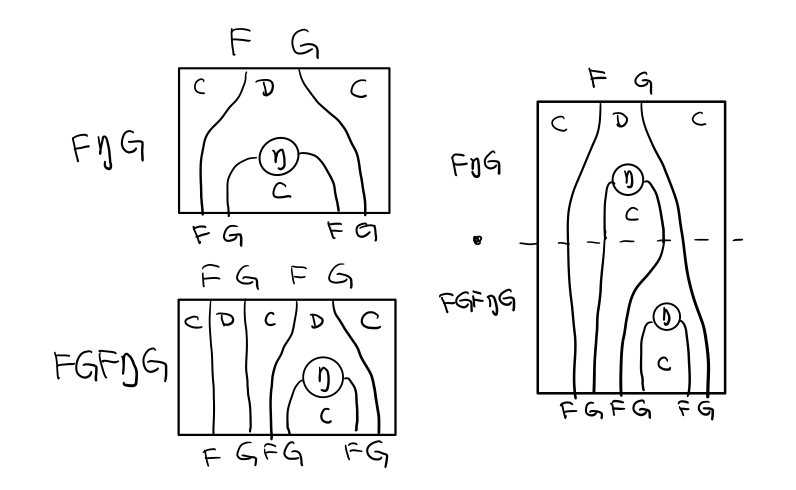
m >>= eta = m

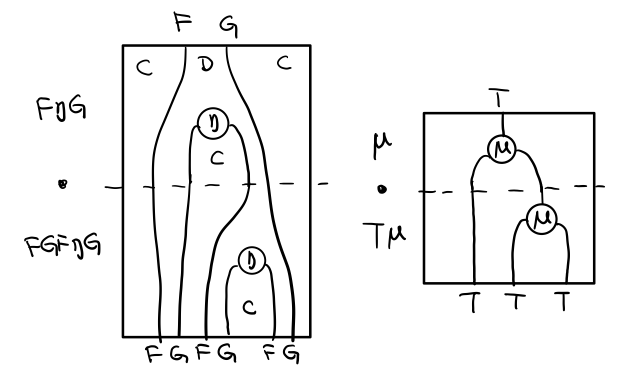
#### 结合律

单子另一大定律是结合律，让我们从伴随函子推起

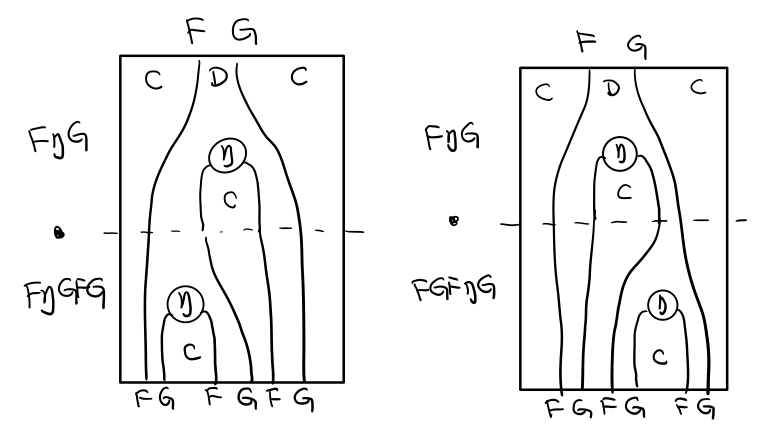
假设我们现在有函子 F η G 和 函子 F η G F G, compose 起来会变成 F η G . F η G F G 

用 F G = T ， F η G = μ 代换那么就得到了单子的 μ . μ T 

当组合 F η G 和 F G F μ G 后，会得到一个镜像的图 

对应到单子的 μ . T μ 

结合律是说 μ . μ T = μ . T μ , 即图左右翻转结果是相等的，为什么呢？看单子的String Diagram 不太好看出来，我们来看伴随函子

如果把左图的左边的 μ 往上挪一点，右边的 μ 往下挪一点，是不是跟右图就一样了 

结合律反映到代码中就是

mu . fmap mu = mu . mu

代码很难看出结合在哪里，因为正常的结合律应该是这样的 (1+2)+3 = 1+(2+3)，但是不想加法的维度不一样，这里说的是自然变换维度的结合，可以通过String Diagram 很清楚的看见结合的过程，即 μ 左边的两个T和先 μ 右边两个 T 是相等的。

### Yoneda lemma /  米田引理

米田引理是说所有Functor f a 一定存在 embed 和 unembed，使得 f a 和 (a -> b) -> F b isomorphic 同构

haskell还要先打开 RankNTypes 的 feature

{-# LANGUAGE RankNTypes #-}

embed :: Functor f => f a -> (forall b . (a -> b) -> f b)

embed x f = fmap f x

unembed :: Functor f => (forall b . (a -> b) -> f b) -> f a

unembed f = f id

embed 可以把 functor f a 变成 (a -> b) -> f b

unembed 是反过来， (a -> b) -> f b 变成 f a

上个图就明白了

Figure 16: 也就是说，图中无论知道a->b 再加上任意一个 F x，都能推出另外一个 F

#### Rank N Type

* Monomorphic Rank 0 / 0级单态: t
* Polymorphic Rank 1 / 1级  多态: forall a. a -> t
* Polymorphic Rank 2 / 2级多态: (forall a. a -> t) -> t
* Polymorphic Rank 3 / 3级多态: ((forall a. a -> t) -> t) -> t

看rank几只要数左边 forall 的括号嵌套层数就好了

一级多态锁定全部类型变化中的类型a

二级多态可以分别确定 a -> t 这个函数的类型多态

比如

rank2 :: (forall a. a -> a) -> (Bool, Char)

rank2 f = (f True, f 'a')

* f 在 f True 时类型 Boolean -> Boolean 是符合 forall a. a->a 的
* 在 f 'a'= 时类型是 =Char -> Char 也符合 forall a. a->a

但是到 rank1 类型系统就懵逼了

rank1 :: forall a. (a -> a) -> (Bool, Char)

rank1 f = (f True, f 'a')

f 在 f True 是确定 a 是 Boolean，在rank1多态是时就确定了 a->a 的类型一定是Boolean -> Boolean

所以到 =f 'a'= 类型就挂了。

### *Kleisli Catergory*

Figure 17: 注意观察大火箭 <=< 的轨迹（不知道dot为什么会把这根线搞这么又弯又骚的） 和 >>= 。所以 Kleisli 其实就是斜着走的一个范畴，但是 >>= 把它硬生生掰  直了。

Functor 的 Catergory 叫做 Functor Catergory，因为有箭头 – 自然变换。Monad 也可以定义出来一个 Catergory（当然由于Monad是 Endofunctor，所以他也可以是 自函子范畴），叫做 Kleisli Catergory，那么 Kleisli 的箭头是什么？

我们看定义，Kleisli Catergory

1. 箭头是 Kleisli 箭头 a -> T b
2. 东西就是c范畴中的东西. 因为 a 和 b 都是 c 范畴上的， 由于T是自函子，所以 T b 也是 c 范畴的

看到图上的  和 μ 了没？

f :: b -> T c

fmap f :: T b -> T^2 c

mu :: T^2 c -> T c

紫色的箭头连起来（compose）就是 T f'，所以,

tb >>= f = mu . fmap f tb

大火箭则是蓝色箭头的组合

(f <=< g) = mu . T f . g = mu . fmap f . g

而且大火箭就是 Kleisli 范畴的 compose

(<=<) :: Monad T => (b -> T c) -> (a -> T b) -> (a -> T c)

### Summary

第一部分理论部分都讲完了， 如果你读到这里还没有被这些的概念搞daze，接下来可以看看它到底跟我们编程有鸟关系呢？第二部分将介绍这些概念产生的一些实用的monad

dentity 与 runIdentity 是 isomorphic （同构，忘了的话回去翻第一部分）

Identity :: a -> Identity a

runIdentity :: Identity a -> a

所以 runIdentity . Identity = id ，所以他们是同构的。

左边的 Identity 是类型构造器, 接收类型 a 返回 Identity a 类型

如果 a 是 Int，那么就得到一个 Identity Int 类型。

右边的 Identity 是数据构造器，也就是构造值，比如 Identity 1 会构造出一个值，其类型为 Identity Int

大括号比较诡异，可以想象成给 a 一个 key，同过这个 key 可以把 a 取出来，比如

runIdentity (Identity 1)

会返回 1

Identity 可以实现 Functor 和 Monad，就得到 Identity functor 和 Identity monad

instance Functor Identity where

fmap f (Identity a) = Identity (f a)

instance Monad Identity where

return a = Identity a

Identity a >>= f = f a

可以看到 Identity 即是构造器，也是解构器，在模式匹配是可以 destructure 值。例如上面Functor 实现中的 fmap f (Identity a) , 假如fmap的是 Identity 1, 那么这个模式匹配到 (Identity a) 时会把 1 放到 a 的位置。

Identity 看起来什么也没有干，就跟 identity 函数一样，但是在后面讲到 State monad时你会发现他的价值。

### Maybe

这是一个超级简单的 Monad，首先，需要定义这个一个

data Maybe a = Just a | Nothing

Haskell中定义一个ADT十分简单，不像Scala那么啰嗦。左边是类型构造器，右边有数据构造器，你会发现有一根竖线 | ， 它分隔着两个构造器

* Just
* Nothing

其中 a （一定要小写）可以是任意类型

所以 Just 1 会得到一个 Num a => Mabye a 类型（意思就是 Maybe a 但是 a 的类型约束为 Num ）， Nothing 也会得到一个 Maybe a 只不过 a 没有类型约束。

总之我们有了构造器可以构造出 Maybe 类型，而这个类型能做的事情，就要取决它实现了哪些 class 的 instance 了。比如它可以是一个 Functor

instance Functor Maybe where

fmap f (Just a) = Just (f a)

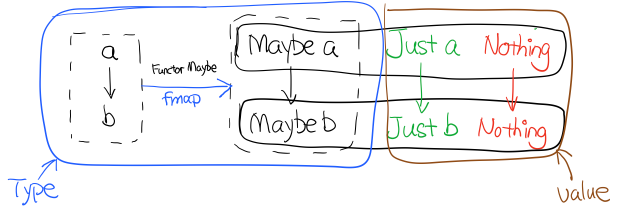


Figure 1: fmap :: (a -> b) -> f a -> f b

然后，还实现 Monad

instance Monad Maybe where

return a = Just a

(Just a) >>= f = f a

Nothing >>= f = Nothing

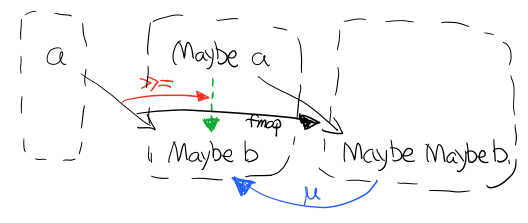


Figure 2: 还记得第一部分提到的 Kleisli 范畴吗？

Maybe 有用在于能合适的处理  的返回值。偏函数相对于，是指只能对部分输入返回输出的函数。

比如一个取数组某一位上的值的函数，就是偏函数，因为假设你想取第4位的值，但不是所有数组长度都大于4，就会有获取不了的尴尬情况。

[1,2,3] !! 4

如果使用 Maybe 把偏函数处理不了的输入都返回成 Nothing，这样结果依然保持 Maybe 类型，不影响后面的计算。

### Either

Either 的定义也很简单

data Either a b = Left a | Right b

#### Product & Coproduct

看过第一部分应该还能记得有一个东西叫 Duel，所以见到如果范畴上有 Coproduct 那么肯定在duel范畴上会有同样的东西叫 Product。

那么我们先来看看什么是 Coproduct

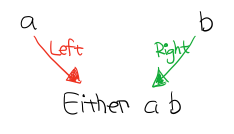
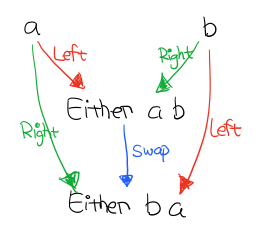


Figure 3: Coproduct

像这样，能通过两个箭头到达同一个东西，就是 Coproduct。这里箭头 Left 能让 a 到Either a b ， 箭头 Right 也能让 b 到达 Either a b

有意思的是还肯定存在一个 Coproduct 和 箭头，使得下图成立 

箭头反过来，就是 Product, 比如 Tuple

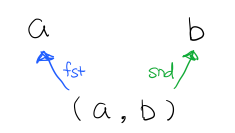


Figure 4: Product

Tuple 的 fst 箭头能让 (a, b) 到达 a 对象，而箭头 snd 能让其到达 b 对象。

#### Either Monad

确切的说，Either 不是 monad， Either a 才是。还记得 monad 的 class 定义吗？

class Endofunctor m => Monad m where

eta :: a -> (m a)

mu :: m m a -> m a

所以 m 必须是个 Endofunctor，也就是要满足Functor

class Functor t where

fmap :: (a -> b) -> (t a -> t b)

t a 的 kind 是 \*，所以 t 必须是 kind \* -> \* 也就是说，m 必须是接收一个类型参数的类型构造器

而 Either 的 kind 是 \* -> \* -> \*, Either a 才是 \* -> \*

所以只能定义 Either a 的 Monad

instance Monad (Either a) where

Left l >>= \_ = Left l

Right r >>= k = k r

很明显的，>>= 任何函数到 都不会改变，只有 >>= 右边才能产生新的计算。

### Reader

Reader 的作用是给一个计算喂数据。

在描述计算的时候，并不需要关心输入时什么，只需要 asks 就可以拿到输入值

而真正的输入，会在运行计算时给予。

跟 Identity 一样，我们用 newtype 来定义一个同构的 Reader 类型

newtype Reader e a = Reader { runReader :: (e -> a) }

其中

* e 是输入
* a 是结果
* 构造 Reader 类型需要确定 输入的类型 e 与输出的类型 a
* runReader 的类型是 runReader:: (Reader e a) -> (e -> a)

也就是说在描述完一个 Reader 的计算后，使用 runReader 可以得到一个 e -> a 的函数，使用这个函数，就可以接收输入，通过构造好的计算，算出结果 a 返回。

那么，让我们来实现 Reader 的 monad instance，就可以描述一个可以 ask 的计算了。

instance Monad (Reader e) where

return a = Reader $ \\_ -> a

(Reader g) >>= f = Reader $ \e -> runReader (f (g e)) e

跟Either一样，我们只能定义 Reader e 的 monad instance。

注意这里的

* f 类型是 (a -> Reader e a)
* g 其实就是是 destructure 出来的 runReader，也就是 e -> a
* 所以 (g e) 返回 a
* f (g e) 就是 Reader e a
* 再 run 一把最后得到 a

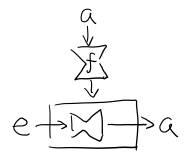


Figure 5: f 函数，接收 a 返回一个 从 e 到 a 的 Reader

让我们来看看如何使用 Reader

import Control.Monad.Reader

data Environment = Env

{ fistName :: String

, lastName :: String

} deriving (Show)

helloworld :: Reader Environment String

helloworld = do

f <- asks firstName

l <- asks lastName

return "Hello " ++ f ++ l

runHelloworld :: String

runHelloworld = runReader helloworld $ Env "Jichao" "Ouyang"

这段代码很简单，helloworld 负责打招呼，也就是在名字前面加个 "Hello"，而跟谁打招呼，这个函数并不关心，而单纯的是向 Environment  就好。

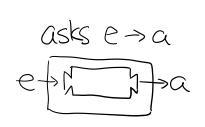
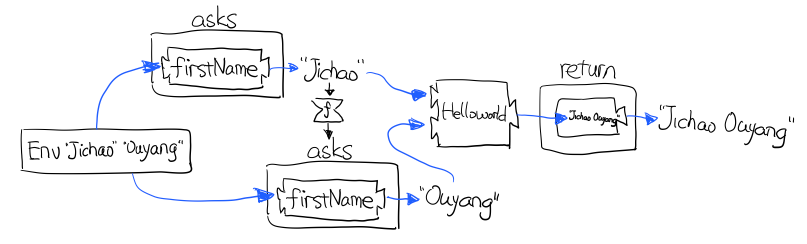


Figure 6: asks 可以将 e -> a 的函数变换成 Reader e a

在运行时，可以提供给 Reader 的输入 Env fistname lastname。 

#### do notation

这可能是你第一次见到 do 和 <-. 如果不是，随意跳过这节。

* do 中所有 <- 的右边都是 Reader Environment String 类型
* do 中的 return 返回类型也必须为 Reader Environment String
* asks firstName 返回的是 Reader Environment String 类型， <- 可以理解成吧 monad Reader Environment 的内容放到左边的 f, 所以 f 的类型是 String。

看起来像命令式的语句，其实只是 >>= 的语法糖，但是明显用do可读性要高很多。

helloworld = (asks firstName) >>=

\f -> (asks lastName) >>=

\l -> return "Hello " ++ f ++ l

### Writer

除了返回值，计算会需要产生一些额外的数据，比如 log

此时就需要一个 Writter，其返回值会是一个这样 (result, log) 的 tuple

限制是 log 的类型必须是个

example :: Writer String String

example = do

tell "How are you?"

tell "I'm fine thank you, and you?"

return "Hehe Da~"

output :: (String, String)

output = runWriter example

*-- ("Hehe Da~", "How are you?I'm fine thank you, and you?")*

Writer 的定义更简单

newtype Writer l a = Writer { runWriter :: (a,l) }

里面只是一个 tuple 而已

* w 是 log
* a 是 返回值

看看如何实现 Writer monad

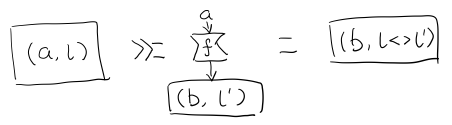
instance (Monoid w) => Monad (Writer w) where

return a = Writer (a,mempty)

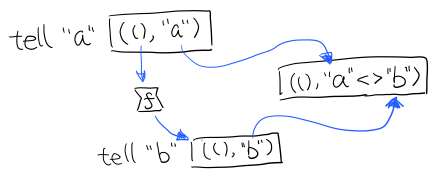
(Writer (a,l)) >>= f = let (a',l') = runWriter $ f a in

Writer (a',l `mappend` l')

* return 不会有任何 log，l 是 monoid 的 mempty
* f 的类型为 a -> Writer l a
* runWriter $ f a 返回 (a, l)



所以在 >>= 时，我们先把 f a 返回的 Writer run了，然后把两次 log mappend 起来。



### State

跟名字就看得出来 State monad 是为了处理状态。虽然函数式编程不应该有状态，不然会引用透明性。但是，state monad并不是在计算过程中修改状态，而是通过描述这种变化，然后需要时在运行返回最终结果。这一点跟 Reader 和 Writer 这两个看起来是副作用的 IO 是一样的。

先看下 State 类型的定义

newtype State s a = State { runState :: s -> (a, s) }

可以看到 State 只包含一个 从旧状态 s 到新状态 s 和返回值 a 的 Tuple 的函数。

通过实现 Monad，State 就可以实现命令式编程中的变量的功能。

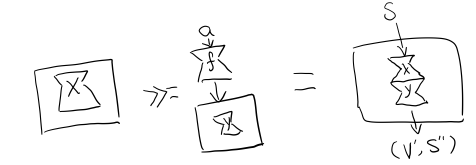
instance Monad (State s) where

return a = State $ \s -> (a,s)

(State x) >>= f = State $ \s -> let (v,s') = x s in

runState (f v) s'

return 很简单，就不用解释了。



x 类型是 s -> (a, s) ,所以 x s 之后会返回 结果和状态。也就是运行当前 State，把结果 v 传给函数 f，返回的 State 再接着上次状态运行。

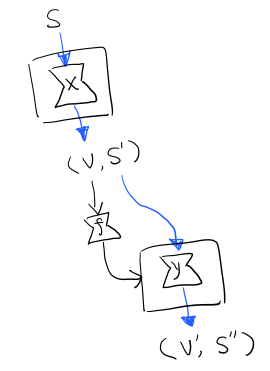


Figure 10: State x >>= f 后runState的数据流（啊啊啊，画歪了，感觉需要脉动一下）

使用起来也很方便，State 提供 get put moidfy 三个方便的函数可以生成修改状态的State monad

import Control.Monad.Trans.State.Strict

test :: State Int Int

test = do

a <- get

modify (+1)

b <- get

return (a + b)

main = print $ show $ runState test 3

*-- (7, 4)*

### Validation

如果你有注意到，前面的 Either 可以用在处理错误和正确的路径分支，但是问题是错误只发生一次。

*Validation 没有在标准库中，但是我觉得好有用啊，你可以在 ekmett 的 [github](https://github.com/ekmett/either) 中找到源码*

想象一下这种场景，用户提交一个表单，我们需要对每一个field进行验证，如果有错误，需要把错误的哪几个field的错误消息返回。显然如果使用 Either 来做，只能返回第一个field的错误信息，后面的计算都会被跳过。

针对这种情况， Validation 更适合

data Validation e a = Failure e | Success a

ADT定义看起来跟 Either 是一样的，不同的是  Failure 是

monoid 首先得是 {{{(半群,Semigroup)}}} ，然后再 含幺。

class Semigroup a where

(<>) :: a -> a -> a

(<>) = mappend

半群非常简单，只要是可以 <> (mappend) 的类型就是了。

含幺只需要有一个 mempty 的 幺元就行

class Monoid a where

mempty :: a

mappend :: a -> a -> a

比如 List 就是 Semigroup

instance Semigroup [a] where

(<>) = (++)

也是 Monoid

instance Monoid [a] where

mempty = []

mappend = (++)

Monoid 的 <> 满足：

* mempty <> a = a
* a <> b <> c = a <> (b <> c)

#### 回到 Validation

现在让 Failure e 满足 Monoid，就可以 mappend 错误信息了。

instance Semigroup e => Semigroup (Validation e a) where

Failure e1 <> Failure e2 = Failure (e1 <> e2)

Failure \_ <> Success a2 = Success a2

Success a1 <> Failure \_ = Success a1

Success a1 <> Success \_ = Success a1

下来，我们用一个简单的例子来看看 Validation 与 Either 有什么区别。

假设我们有一个form，需要输入姓名与电话，验证需要姓名是非空而电话是11位数字。

首先，我们需要有一个函数去创建包含姓名和电话的model

data Info = Info {name: String, phone: String} deriving Show

然后我们需要验证函数

notEmpty :: String -> String -> Validation [String] String

notEmpty desc "" = Failure [desc <> " cannot be empty!"]

notEmpty \_ field = Success field

notEmpty 检查字符是否为空，如果是空返回 Failure 包含错误信息，若是非空则返回 Success 包含 field

同样的可以创建 11位数字的验证函数

phoneNumberLength :: String -> String -> Validation [String] String

phoneNumberLength desc field | (length field) == 11 = Success field

| otherwise = Failure [desc <> "'s length is not 11"]

实现 Validation 的 Applicative instance，这样就可以把函数调用lift成带有验证的 Applicative

instance Semigroup e => Applicative (Validation e) where

pure = Success

Failure e1 <\*> Failure e2 = Failure e1 <> Failure e2

Failure e1 <\*> Success \_ = Failure e1

Success \_ <\*> Failure e2 = Failure e2

Success f <\*> Success a = Success (f a)

* 失败应用到失败会 concat 起来
* 失败跟应用或被成功应用还是失败
* 只有成功应用到成功才能成功，这很符合验证的逻辑，一旦验证中发生任何错误，都应该返回失败。

createInfo :: String -> String -> Validation [String] Info

createInfo name phone = Info <$> notEmpty "name" name <\*> phoneNumberLength "phone" phone

现在我们就可以使用带validation的 createInfo 来安全的创建 Info 了

createInfo "jichao" "12345678910" *-- Success Info "jichao" "12345678910"*

createInfo "" "123" *-- Failure ["name cannot be empty!", "phone's length is not 11"]*

### Cont

Cont 是  的 monad，也就是说，它是包含 cps 计算 monad。

先看一下什么是 CPS，比如有一个加法

add :: Int -> Int -> Int

add = (+)

但是如果你想在算法加法后，能够继续进行一个其他的计算，那么就可以写一个 cps版本的加法

addCPS :: Int -> Int -> (Int -> r) -> r

addCPS a b k = k (a + b)

非常简单，现在我们可以看看为什么需要一个 Cont monad 来包住 CPS 计算，首先，来看 ADT 定义

newtype Cont r a = Cont { runCont :: ((a -> r) -> r) }

又是一个同构的类型，Cont 构造器只需要一个 runCount，也就是让他能继续计算的一个函数。

完了之后来把之前的 addCPS 改成 Cont

add :: Int -> Int -> Cont k Int

add a b = return (a + b)

注意到 addCPS 接收到 a 和 b 之后返回的类型是 (Int -> r) -> r ，而 Cont 版本的add 返回 Cont k Int

明显构造 Cont k Int 也正是需要 (Int -> r) -> r ，所以 Cont 就是算了 k 的抽象了。

instance Monad (Cont r) where

return a = Cont $ \k -> k a

(Cont c) >>= f = Cont $ \k -> c (\a -> runCont (f a) k)

### Summary

第二部分食用部分也讲完了， 不知是否以及大致了解了monad的尿性各种基本玩法呢？通过这些常用的基本的 monad instance，解决命令式编程中的一些简单问题应该是够了。

不过，接下来还有更变态的猫，就先叫她  猫呢好了。