

内容

[Functor](#)[Applicative](#)[Monad](#)[结论](#)

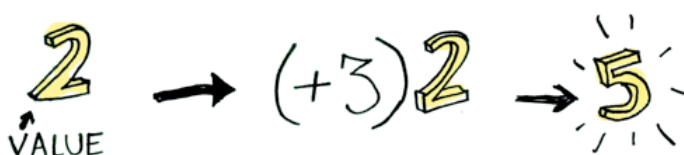
Functor, Applicative, 以及 Monad 的图片阐释

原作者写于 **APRIL 17, 2013**

这是个简单的值:

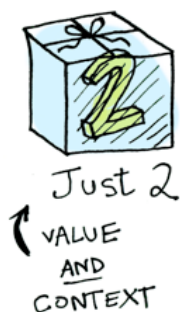


我们都知道怎么加一个函数应用到这个值上边:

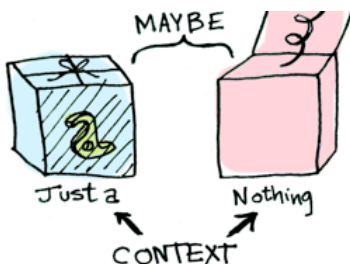


```
const add = (a, b) => a + b
const AddTwo = R.curry(add)(2)
```

很简单了. 我们来扩展一下, 让任意的值是在一个上下文当中. 现在的情况你可以想象一个可以把值放进去的盒子:



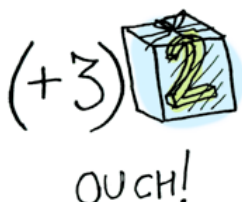
现在你把一个函数应用到这个值的时候, 根据其上下文你会得到不同的结果. 这就是 **Functor**, **Applicative**, **Monad**, **Arrow** 之类概念的基础. **Maybe** 数据类型定义了两种相关的上下文:



很快我们会看到对一个 `Just a` 和一个 `Nothing` 来说函数应用有何不同. 首先我们来说 **Functor**!

Functor

当一个值被封装在一个上下文里, 你就不能拿普通函数来应用:

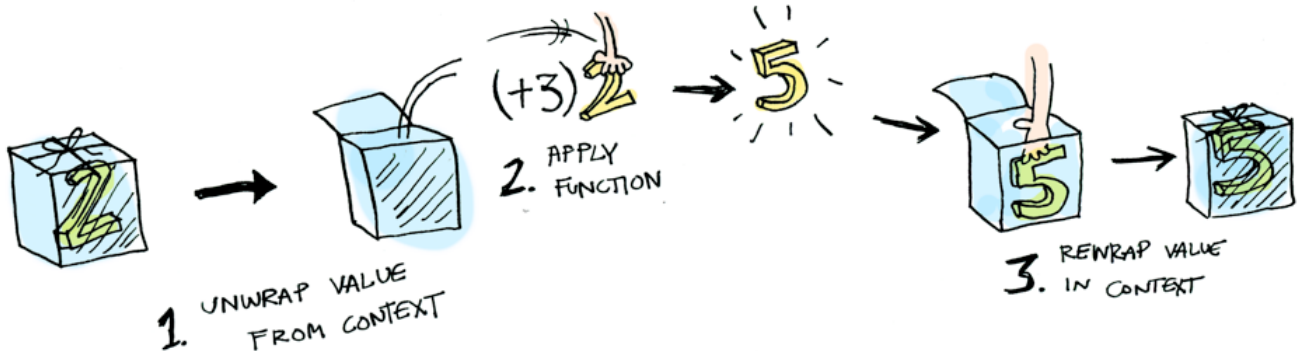


就在这里 `fmap` 出现了. `fmap` is from the street, `fmap` is hip to contexts. `fmap` 知道怎样将一个函数应用到一个带有上下文的值. 你可以对任何一个类型为 `Functor` 的类型使用 `fmap`.

比如说, 想一下你想把 `(+3)` 应用到 `Just 2`. 用 `fmap`:

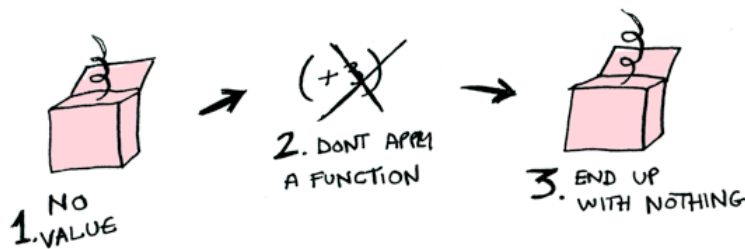
```
> fmap (+3) (Just 2)
Just 5
```

这是在幕后所发生的:



Bam! `fmap` 告诉了我们那是怎么做到的!

So then you're like, 好吧 `fmap`, 请应用 `(+3)` 到一个 `Nothing` ?



```
> fmap (+3) Nothing
Nothing
```

就像 *Matrix* 里的 *Morpheus*, `fmap` 就是知道要做什么; 你从 `Nothing` 开始, 那么你再由

`Nothing` 结束! `fmap` 是禅. So now you're all like, 准确说究竟什么是 **Functor**? 嗯, **Functor** 就是任何能用 `fmap` 操作的数据类型. 因此 `Maybe` 是个 **functor**. 而且我们很快会看到, `list` 也是 **functor**. 这样上下文存在就有意义了. 比如, 这是在没有 `Maybe` 的语言里你操作一个数据库记录的方法:

```
post = Post.find_by_id(1)
if post
  return post.title
else
  return nil
end
```

但用 **Haskell**:

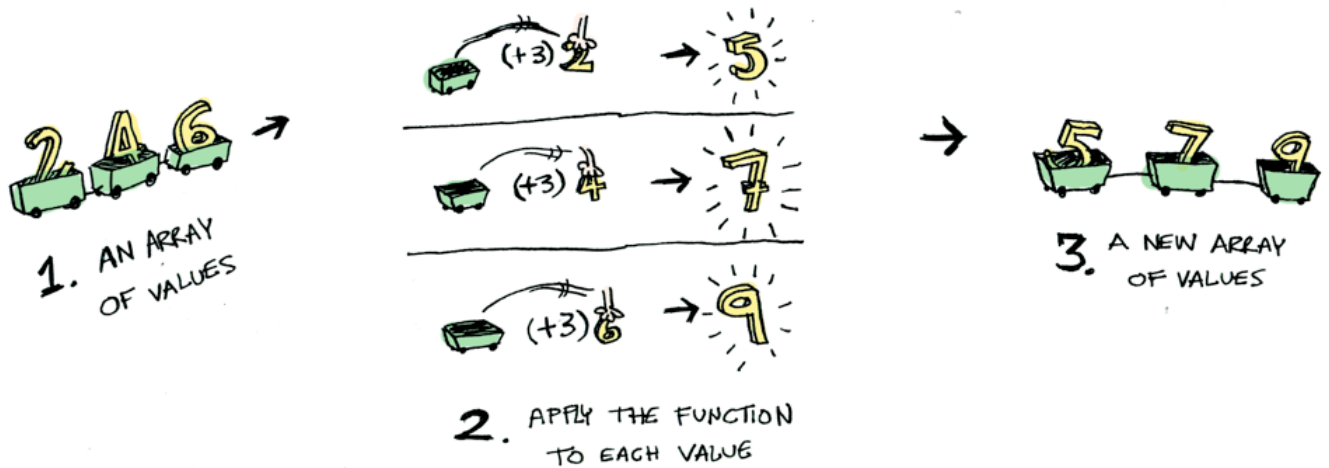
```
fmap (getPostTitle) (findPost 1)
```

如果 `findPost` 返回一条 `post`, 我们就通过 `getPostTitle` 得到了 `title`. 如果返回的是

`Nothing`, 我们加得到 `Nothing` ! 非常简洁, huh? `<$>` 是 `fmap` 的中缀表达式版本, 所以你会经常是看到:

```
getPostTitle <$> (findPost 1)
```

另一个例子: 当你把函数应用到 `list` 时发生了什么?

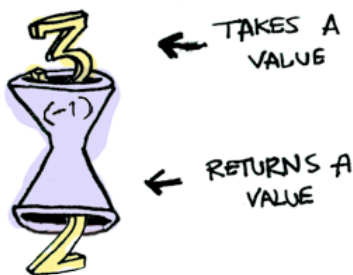


`List` 仅仅是另一种让 `fmap` 以不同方式应用函数的上下文!

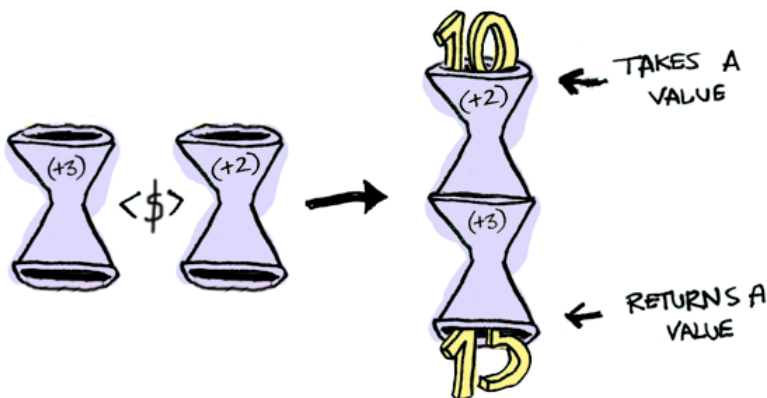
Okay, okay, 最后一个例子: 你把你一个函数应用到另一个函数时会发生什么?

```
fmap (+3) (+1)
```

这是个函数:



这是一个函数应用到另一个函数上:



结果就是又一个函数!

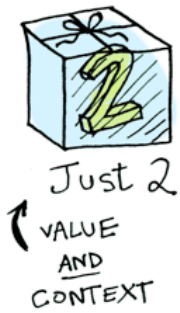
```
> import Control.Applicative
> let foo = (+3) <$> (+2)
> foo 10
15
```

这就是函数复合! 就是说, $f \text{ < \$ > } g == f . g$!

注意: 目前为止我们做的是将上下文当作是一个容纳值的盒子. But sometimes the box analogy wears a little thin. 特别要记住: 盒子是有效的记忆图像, 然而你并没有盒子. 有时你的“盒子”是个函数.

Applicative

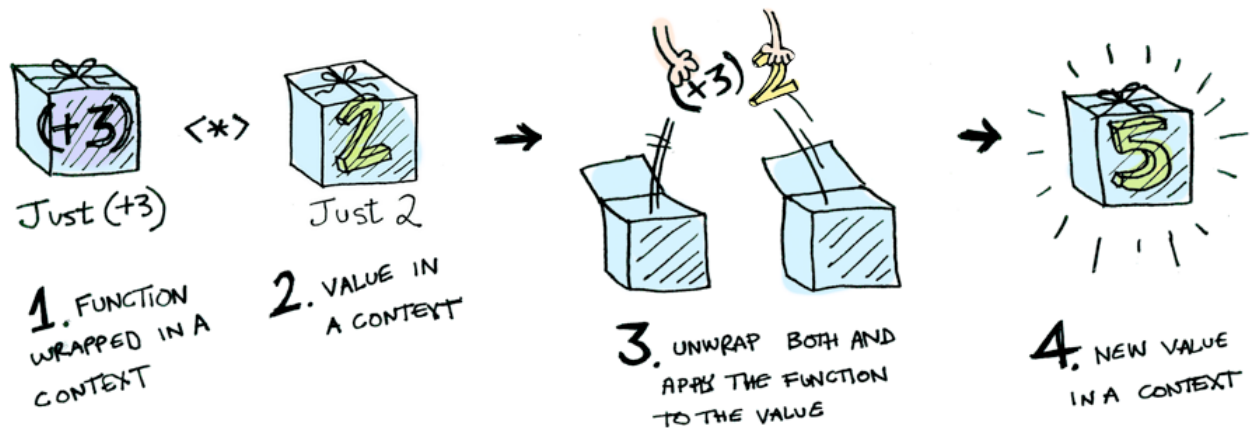
Applicative 把这带到了一个新的层次. 借助 **applicative**, 我们的 **values** 就被封装在了上下文里, 就像 **Functor**:



而我们的函数也被封装在了上下文里!



Yeah. Let that sink in. **Applicative** 并不是开玩笑. `Control.Applicative` 定义了 `<*>`, 这个函数知道怎样把封装在上下文里的函数应用到封装在上下文里的值:

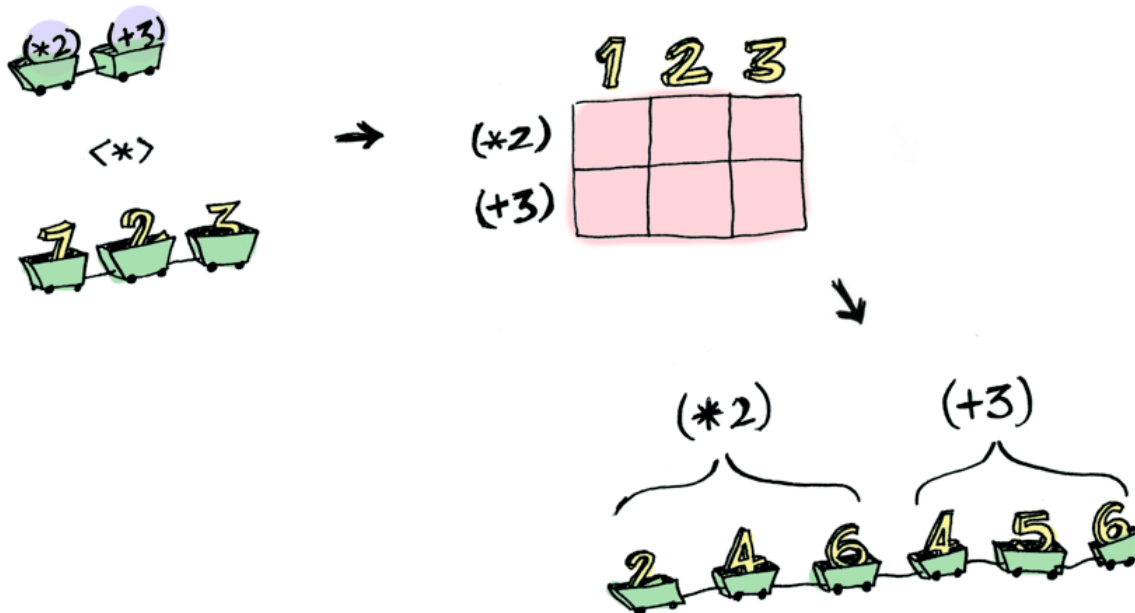


也就是:

```
Just (+3) <*> Just 2 == Just 5
```

使用 `<*>` 能带来一些有趣的情形. 比如:

```
> [(+2), (+3)] <*> [1, 2, 3]
[2, 4, 6, 4, 5, 6]
```



这里有一些是你能用 **Applicative** 做, 而无法用 **Functor** 做到的. 你怎么才能把需要两个参数的函数应用到两个封装的值上呢?

```
> (+) <$> (Just 5)
Just (+5)
> Just (+5) <$> (Just 4)
ERROR ??? WHAT DOES THIS EVEN MEAN WHY IS THE FUNCTION WRAPPED IN A JUST
```

Applicative:

```
> (+) <$> (Just 5)
Just (+5)
> Just (+5) <*> (Just 3)
Just 8
```

Applicative 把 **Functor** 推到了一边. “大腕儿用得起任意个参数的函数,” 他说. “用 $<$>$ 和 $<*>$ 武装之后, 我可以接受需要任何个未封装的值的函数. 然后我传进一些封装过的值, 再我就得到一个封装的值的输出! AHAHAHAHAH!”

```
> (*) <$> Just 5 <*> Just 3
Just 15
```



一 *applicative* 看着一 *functor* 应用一函数

还有啦! 有一个叫做 `liftA2` 的函数也做一样的事:

```
> liftA2 (*) (Just 5) (Just 3)
Just 15
```

Monad

如何学习 **Monad**:

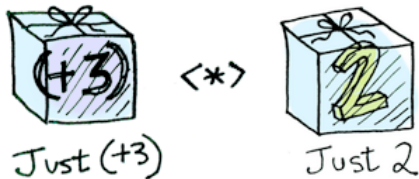
1. 拿个计算机科学的 PhD.
2. 把她抛在一边, 因为这个章节里你用不到她!

Monads add a new twist.

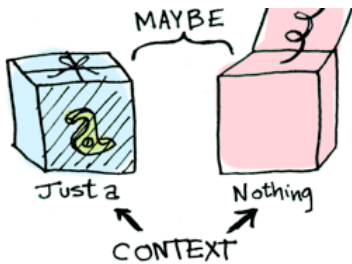
Functor 应用函数到封装过的值:



Applicative 应用封装过的函数到封装过的值:



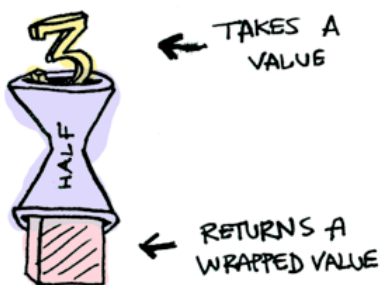
Monads 应用会返回封装过的值的函数到封装过的值. Monad 有个 `>=>` (念做 “bind”) 来做这个. 一起看个例子. Good ol' `Maybe` is a monad:



Just a monad hanging out

假定 `half` 是仅对偶数可用的函数:

```
half x = if even x
         then Just (x `div` 2)
         else Nothing
```



我们给它传入一个封装过的值会怎样?



我们要用到 `>=>`, 用来强推我们封装过的值到函数里去. 这是 `>=>` 的照片:



它怎么起作用的:

```
> Just 3 >= half
Nothing
> Just 4 >= half
Just 2
> Nothing >= half
Nothing
```

其中发生了什么?



如果你传进一个 `Nothing` 就更简单了:



酷! 我们来看另一个例子: 那个 `IO monad`:



明确的三个函数. `getLine` 获取用户输入而不接收参数:



```
getLine :: IO String
```

`readFile` 接收一个字符串 (文件名) 再返回文件的内容:


```
readFile :: FilePath -> IO String
```

`putStrLn` 接收一个字符串打印:

```
putStrLn :: String -> IO ()
```

这三个函数接收一个常规的值 (或者不接收值) 返回一个封装过的值. 我们可以用 `>>=` 把一切串联起来!

```
getLine >>= readFile >>= putStrLn
```

Aw yeah! 我们不需要在取消封装和重新封装 `IO monad` 的值上浪费时间. `>>=` 为我们做了那些工作! Haskell 还为 `monad` 提供了语法糖, 叫做 `do` 表达式:

```
foo = do
  filename <- getLine
  contents <- readFile filename
  putStrLn contents
```

结论

functor: 通过 `fmap` 或者 `<$>` 应用是函数到封装过的值

applicative: 通过 `<*>` 或者 `liftA` 应用封装过的函数到封装过的值

monads: 通过 `>>=` 或者 `liftM` 应用会返回封装过的值的函数到封装过的值

所以, 亲爱的朋友 (我想在这点上我们是朋友), 我想我们都一致认为 `monad` 是简单的而且是个高明的观念(tm). Now that you've wet your whistle on this guide, why not pull a Mel Gibson and grab the whole bottle. 看下 LYAH 上关于 `Monad` 的 [章节](#). 很多东西我其实掩饰了因为 Miran 深入这方面做得很棒.

4 Comments adit.io

1


 Login

Recommend 1

Tweet

Share

Sort by Best



Join the discussion...

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

- CicholGricenchos • 3 years ago


其实一切都是为了延时求值。。抛开延时求值，Functor和Just没有实际的区别，Functor就是把操作叠加到一个函数上，在用的时候再应用到Just上

1 ^ | v • Reply • Share ›
- peng • a year ago

居然还专门为中文做了一个板块，看来国内的程序员真的好多

^ | v • Reply • Share ›
- brzhang • 3 years ago

感谢，太形象了。

^ | v • Reply • Share ›
-  agent • 6 years ago

很形象，但过于形象的解释往往也容易不得要领，流于表面现象

^ | v • Reply • Share ›