> *这篇文章是 Aditya Bhargava 所著《Functors, Applicatives, And Monads In Pictures》的中文译文，已联系原作者取得授权。另一版本的中文译文由 题叶 翻译，可在此处查看。*

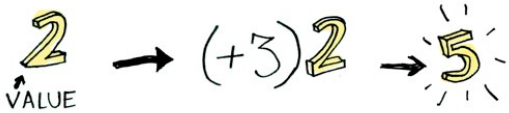
> *This Article is the Chinese translation for Functors, Applicatives, And Monads In Pictures (Written by Aditya Bhargava).*

- 英文原文写于 2013 年 4 月 17 日。

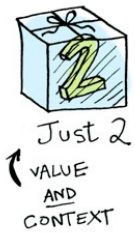
引文

下图是一个简单的值：

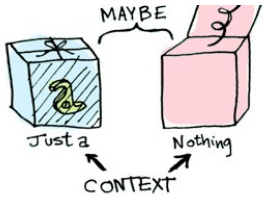
将一个函数应用到这个值上：



简直 Naive，让我们来扩展一个！假设一个值可以被放到上下文中。你可以把上下文想象成一个盒子，把值放入上下文的过程就如同把东西放到盒子里：



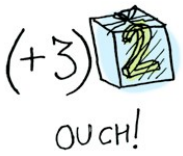
现在再把一个函数应用到这个值上，**根据不同的上下文**，我们将得到不同的结果。这就是 `Functor`、`Applicative`、`Monad`、`Arrows` 等概念的基础。就 `Maybe` 这一型别来说，它定义了两种相关联的上下文：



马上我们就会看到对 `Just a` 和 `Nothing` 应用一个函数的不同之处。在此之前，让我们先了解一下 `Functor`！

Functors

如果一个值被封装在上下文中，你会发现普通函数无法直接对其操作：



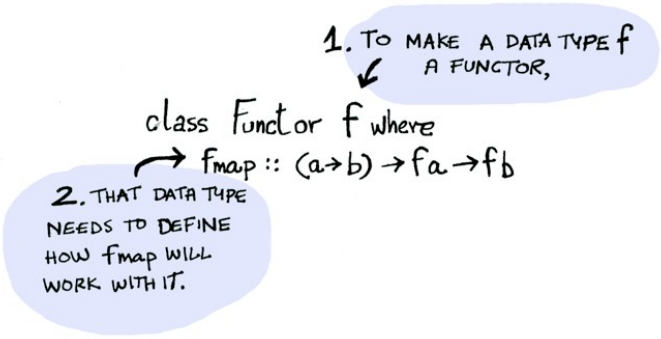
这时 `fmap` 就会发挥作用了！`fmap` 能够和上下文谈笑风生，它对普通函数和被上下文包装的值施了一点魔法，让它们能够愉快相处。举个例子，你想把函数 `(+3)` 应用到 `Just 2` 上，那么只需要加上 `fmap`：

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

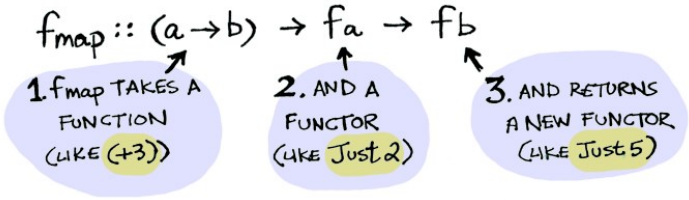
一颗赛艇！`fmap` 向我们展示了它的威力！但是 `fmap` 怎么知道如何应用一个函数呢？

什么是 Functor

`Functor` 是一个 类型类，这是它的定义：



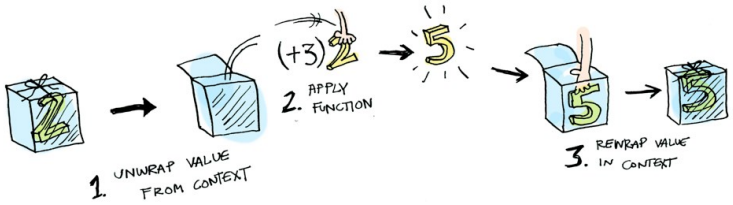
任何型别，只要能用 `fmap` 操作，就是一个 `Functor`。下面这张图展示了 `fmap` 各个参数的含义：



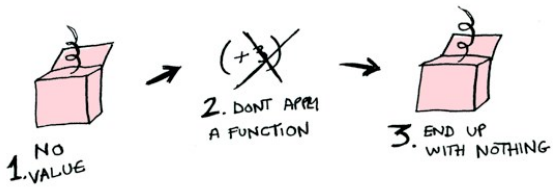
我们之所以能够执行 `fmap (+3) (Just 2)` ，是因为 `Maybe` 也是一个 `Functor` 。下面的定义指明了 `fmap` 在面对 `Just` 和 `Nothing` 时的处理方式：

```
instance Functor Maybe where
    fmap func (Just val) = Just (func val)
    fmap func Nothing = Nothing
```

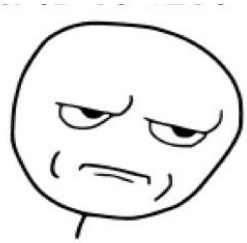
下图说明了执行 `fmap (+3) (Just 2)` 的整个过程：



假设你灵机一动，让 `fmap` 把 `(+3)` 应用到 `Nothing` 上：




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

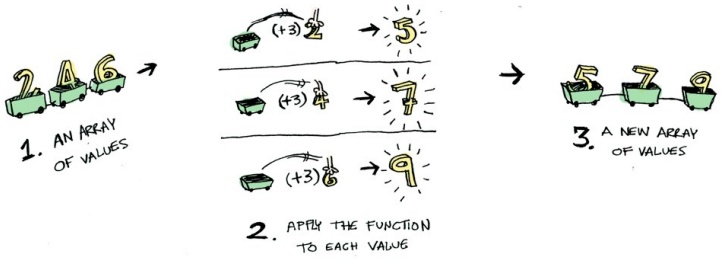


如墨菲斯（黑客帝国中的角色）一般执着，`fmap` 也很清楚自己该做什么：从 `Nothing` 开始就从 `Nothing` 结束！这也是 `Maybe` 类型存在的意义。我们通常使用类似如下的 Python 代码处理数据库：

```python
post = Post.find_by_id(1)
if post:
  return post.title
else:
  return None
```

用 `Haskell` 可以写成 `fmap (getPostTitle) (findPost 1)`。如果 `findPost` 返回了一篇文章，我们就可以通过 `getPostTitle` 获取其标题。如果 `findPost` 返回了 `Nothing`，我们当然也应该返回 `Nothing`！是不是很简洁？`<$>` 是 `fmap` 的中缀版本，所以写成 `getPostTitle <$> (findPost 1)` 也是允许的，并且这种写法更常见。

再来看一个例子：把一个函数应用到 `List` 上会发生什么呢？

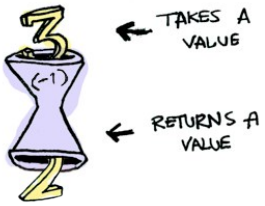


`List` 也是 `Functor`！这是它的定义：

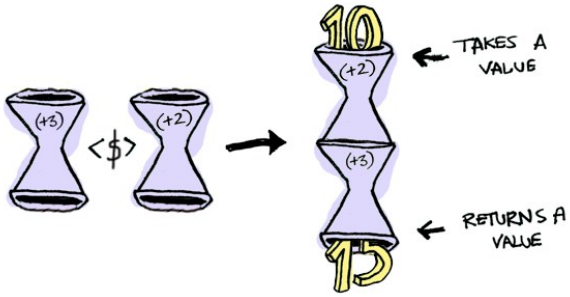
```python
instance Functor [] where
  fmap = map
```

我想你应该理解得差不多了，最后一个例子：如果把函数应用到另一个函数上呢，比如 `fmap (+3) (+1)` ？

这是一个函数：



将某个函数应用到另一个函数：



得到的结果是一个新函数！

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

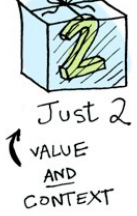
由此可见，函数同样是 `Functor`：

```python
instance Functor   ((->)   r) where
  fmap f g = f . g
```

函数的 `fmap` 其实就是函数复合。

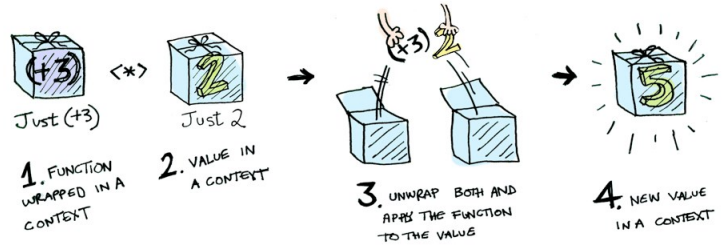
Applicative

`Applicative` 将 `Functor` 又提高了一个层次。与 `Functor` 类似，`Applicative` 中的值也被封装在上下文中：



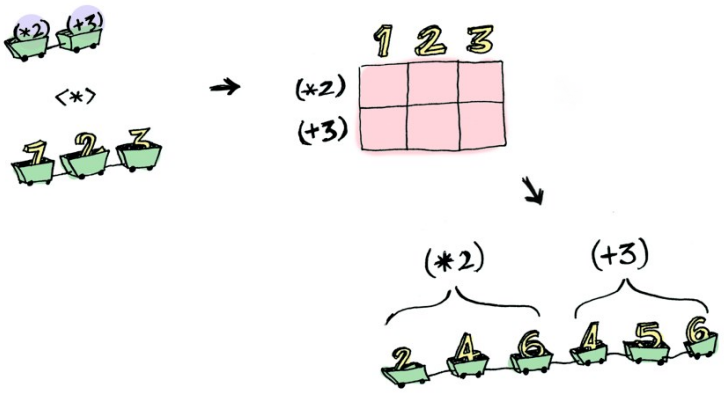
不同之处在于，现在函数也被封装到上下文中：

`Control.Applicative` 定义了 `<*>`，它知道如何将一个 *包装在上下文中的* 函数应用到 *包装在上下文的* 值上。举个例子，`Just (+3) <*> Just 2 == Just 5`：



使用 `<*>` 会产生很多有趣的情况，看看下面的 `List` 会发生什么：

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



下面是一些 `Applicative` 有而 `Functor` 不具备的功能。如何将一个接收两个参数的函数应用到两个被封装的值呢？

```
> (+) <$> (Just 5)
Just (+5)
> Just (+5) <$> (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` 丢到了一边。“我今天就教你们一点人生经验”，`Applicative` 如是说，“在装备了 `<$>` 和 `<*>` 后，我可以接受任何函数，之后我把对应的封装值喂给它们，最后我就得到了一个封装好的值！哈哈哈哈哈！”

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

对了，这种模式可以用 `liftA2` 简化：

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

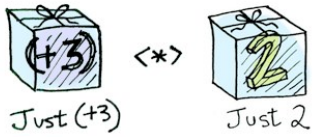
Monad

如何学习 `Monad` ：

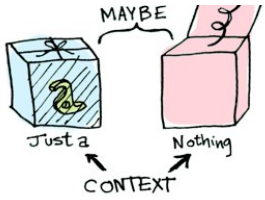
- 在计算机科学专业取得博士学位。
- 不学。因为这一节你根本用不到它！

`Functor` 将一个普通函数应用到被封装的值上：

`Applicative` 将一个封装的函数应用到封装值上：

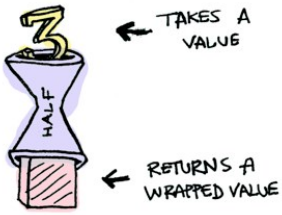


`Monad` 将一个 **“接受一个普通值并回传一个被封装的值”** 的函数应用到一个被封装的值上，这一任务由函数 `>>=` （读作 “bind”）完成。听起来似乎很拗口，让我们来看个例子吧，还是熟悉的 `Maybe` ：



假设 `half` 是只对偶数感兴趣的函数：

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



如果给 `half` 一个被封装的值会怎样？

这时我们需要用 `>>=` 把被封装的值挤到 `half` 中。看看 `>>=` 的照片：

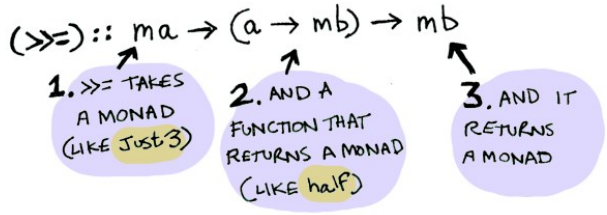
再看看它的效果：

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

这其中究竟发生了什么？`Monad` 是另一种类型类，这是它定义的一部分：

```
class Monad m where
  (>>=) :: m a -> (a -> m b) -> m b
```

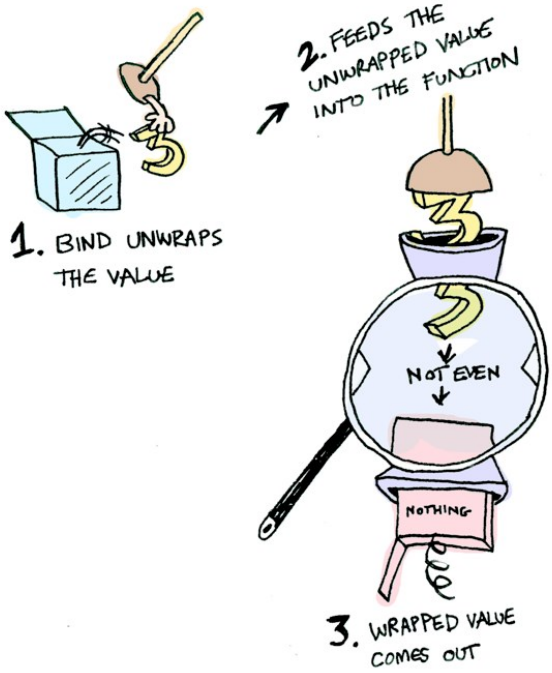
下图展示了 `>>=` 各个参数的意义：



下面的定义让 `Maybe` 成为了 `Monad` ：

```
instance Monad Maybe where
    Nothing >>= func = Nothing
    Just val >>= func  = func val
```

来看看执行 `Just 3 >>= half` 时发生了什么：



如果传入 `Nothing` 就更容易了：

这些调用过程还可以被连起来，比如执行 `Just 20 >>= half >>= half >>= half` 会得到 `Nothing` ：

流弊！现在我们知道， `Maybe` 既是 `Functor` ，又是 `Applicative` ，还是 `Monad` 。

再来看另一个例子： `IO Monad` 。

介绍三个函数先。

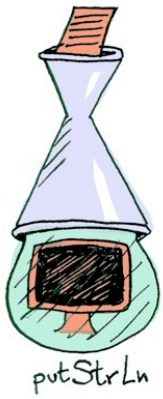
- `getLine` 不接受参数并获取用户输入（ `getLine :: IO String` ）：

getLine

- `readFile` 接受一个字符串（文件路径）并返回文件的内容（`readFile :: FilePath -> IO String`，`FilePath` 是 `String` 的别名）：

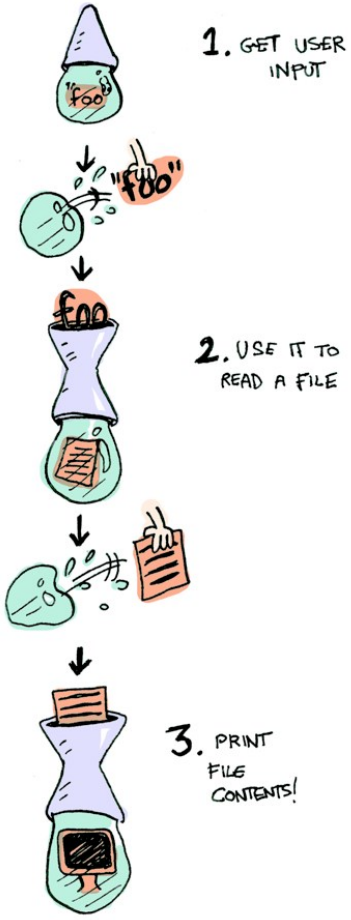
readFile

- `putStrLn` 接受一个字符串并打印它（`putStrLn :: String -> IO ()`）：



putStrLn

这三个函数都接受一个正常的值（或者不接受值）并且回传一个被封装在 `IO Monad` 中的值。我们可以用 `>>=` 把它们串起来！




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

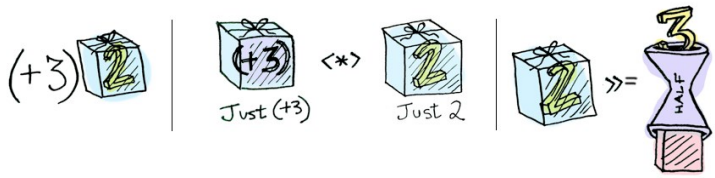
Haskell 还为我们提供了 `do` ，它是 `Monad` 的语法糖：

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

总结

1. 实现了 `Functor` 类型类的数据类型被称为 functor。
2. 实现了 `Applicative` 类型类的数据类型被称为 applicative。
3. 实现了 `Monad` 类型类的数据类型被称为 monad。
4. `Maybe` 实现了这三种类型类，所以它同时是 functor、applicative 和 monad。

它们三个之间的区别是什么呢？



Functor　　Applicative　　Monad

- `functors` ：使用 `fmap` 或 `<$>` 把一个普通函数应用到被封装的值上
- `applicatives` ：使用 `<*>` 或 `liftA` 把一个被封装的函数应用到被封装的值上
- `monads` ：使用 `>>=` 或 `liftM` 把一个接受普通值、回传封装值的函数应用到一个被封装的值上

亲爱的朋友（我觉得我们算是朋友了），现在你是否觉得 monad 是一个简单并聪明的概念呢？既然你已经读完了这篇 “科普文”，不如进一步了解一下 monad：LYAH 编写的 Monad 章节 中包含了许多我在本文中忽略的信息，他写的非常棒，我就不在此赘述了。

更多与 Monad 相关的图文介绍，请看 三种实用 monad。

英文原文链接： *Functors, Applicatives, And Monads In Pictures* （Written by Aditya Bhargava）