

2 1948

第 1 页 (共 13 页)

3. 一个 Monad 就是一种实现了 Monad typeclass 的数据类型。

当然，你可能会问那什么是 **typeclass** 呢？我想当你在看到实现二字的时候，就应该已经猜到了：

A typeclass is a sort of interface that defines some behavior. If a type is a part of a typeclass, that means that it supports and implements the behavior the typeclass describes. A lot of people coming from OOP get confused by typeclasses because they think they are like classes in object oriented languages. Well, they're not. You can think of them kind of as Java interfaces, only better.

是的，typeclass 就类似于 Java 中的接口，或者 Objective-C 中的协议。在 typeclass 中定义了一些函数，实现一个 typeclass 就是要实现这些函数，而所有实现了这个 typeclass 的数据类型都会拥有这些共同的行为。

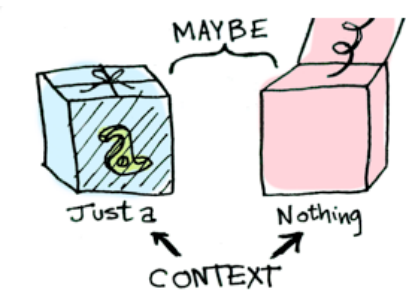
那 Functor、Applicative 和 Monad 三者之间有什么联系吗，为什么它们老是结伴出现呢？其实，Applicative 是增强型的 Functor，一种数据类型要成为 Applicative 的前提条件是它必须是 Functor；同样的，Monad 是增强型的 Applicative，一种数据类型要成为 Monad 的前提条件是它必须是 Applicative。注：这个联系，在我们看到 Applicative typeclass 和 Monad typeclass 的定义时，自然就会明了。

Maybe

在正式开始介绍 Functor、Applicative 和 Monad 的定义前，我想先介绍一种非常有意思的数据类型，**Maybe** 类型（可类比 Swift 中的 Optional）：

The Maybe type encapsulates an optional value. A value of type Maybe a either contains a value of type a (represented as Just a), or it is empty (represented as Nothing). Using Maybe is a good way to deal with errors or exceptional cases without resorting to drastic measures such as error.

Maybe 类型封装了一个可选值。一个 Maybe a 类型的值要么包含一个 a 类型的值（用 Just a 表示）；要么为空（用 Nothing 表示）。我们可以把 Maybe 看作一个盒子，这个盒子里面可能装着一个 a 类型的值，即 Just a；也可能是一个空盒子，即 Nothing。或者，你也可以把它理解成泛型，比如 Objective-C 中的 NSArray。不过，最正确的理解应该是把 Maybe 看作一个上下文，这个上下文表示某次计算可能成功也可能失败，成功时用 Just a 表示，a 为计算结果；失败时用 Nothing 表示，这就是 Maybe 类型存在的意义：



```
1 | data Maybe a = Nothing | Just a
```

下面，我们来直观地感受一下 Maybe 类型：

```
1 | ghci> Nothing
2 | Nothing
3 | ghci> Just 2
4 | Just 2
```

我们可以用盒子模型来理解一下，Nothing 就是一个空盒子；而 Just 2 则是一个装着 2 这个值的盒子：

很厉害的样子

想象力 评论了 iOS平台个人网银APP的安全测试报告...

路过。。。asdasdaa 评论了 因为A商，互联网的寒冬不再冷...

牛逼吹的不错。。lin\_chen 评论了 因为A商，互联网的寒冬不再冷...

太刁了！值得学习！！！mavericksding 评论了 我已经写了48年代码了，我感觉我还能写下去...

&#46;&#46;&#46;&#46;zhouxbbbj1 评论了 腾讯产品经理：8亿月活的腾讯QQ用户体验是怎么做的...

坦白的说，一种让开发人员过分的研究其语言特性和使用技巧的编程语言，并xiaojiachong 评论了 让我们来搞崩Cocoa吧（黑暗代码）...

mark&#46;&#46;&#46;&#46;XuanInitial 评论了 iOS开发——UI组件（个人整理）...

相关帖子

Undefined symbols for architecture armv7:

IOS 游戏付费测试等待时没有进度显示

mac下通过system()命令用默认浏览器打开网址

^\_^求教大神如何判断视图是从哪个vc条转过来的

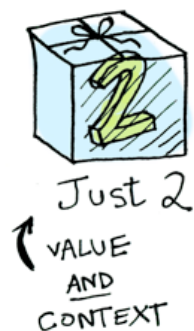
libMobileVLCKit播放流媒体时，只有音频，视频显示不出来

同一个页面内的多个UIActionSheet，如何实现他们的独立功能实现

关于加载webp的问题

iVersion版本动态检查版本更新，大家是如何测试这部分代码的呢

NSRunLoop 怎么停止。。



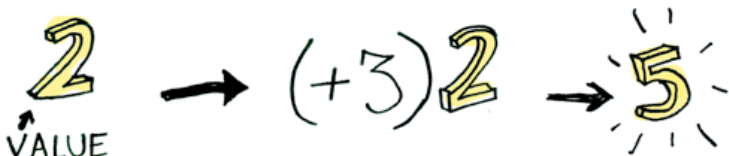
提前剧透：Maybe 类型实现了 Functor typeclass、Applicative typeclass 和 Monad typeclass，所以它同时是 Functor、Applicative 和 Monad，具体实现细节将在下面的章节进行介绍。

Functor

在正式开始介绍 Functor 前，我们先思考一个这样的问题，假如我们有一个值 2：

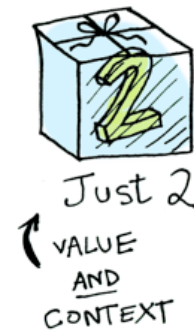


我们如何将函数 (+3) 应用到这个值上呢？我想上过小学的朋友应该都知道，这就是一个简单的加法运算：



```
1 | ghci> (+3) 2
2 | 5
```

分分钟搞定。那么问题来了，如果这个值 2 是在一个上下文中呢？比如 Maybe，此时，这个值 2 就变成了 Just 2：



这个时候，我们就不能直接将函数 (+3) 应用到 Just 2 了。那么，我们如何将一个函数应用到一个在上下文中的值呢？

微博



CocoaChina

加关注

【5个层面解构游戏的设计标准】决定游戏优劣的这个方法被称为游戏设计标准，通常由五个不同的部分所组成：核心体验，基础机制，奖惩系统，长期动机以及美学布局。这篇文章是对于游戏设计标准的解构，为那些不了解这个概念的开发者的做出解释。  
<http://t.cn/RUQ0PQH>



6分钟前 转发(1) | 评论

【极客斗士！“问题青年”与ISIS的一场“看不见的战争”】欧洲某处的一个昏暗杂乱的房间内，一名年轻人蜷缩在角落的沙发上，面前一台破旧电脑





是的，我想你应该已经猜到了，Functor 就是干这事的，欲知后事如何，请看下节分解。

### Functor typeclass

首先，我们来看一下 Functor typeclass 的定义：

```
1 | class Functor f where
2 |     fmap :: (a -> b) -> f a -> f b
```

在 Functor typeclass 中定义了一个函数 fmap，它将一个函数 (a -> b) 应用到一个在上下文中的值 f a，并返回另一个在相同上下文中的值 f b，这里的 f 是一个类型占位符，表示任意类型的 Functor。

注：fmap 函数可类比 Swift 中的 map 方法。

### Maybe Functor

我们知道 Maybe 类型就是一个 Functor，它实现了 Functor typeclass。我们将类型占位符 f 用具体类型 Maybe 代入可得：

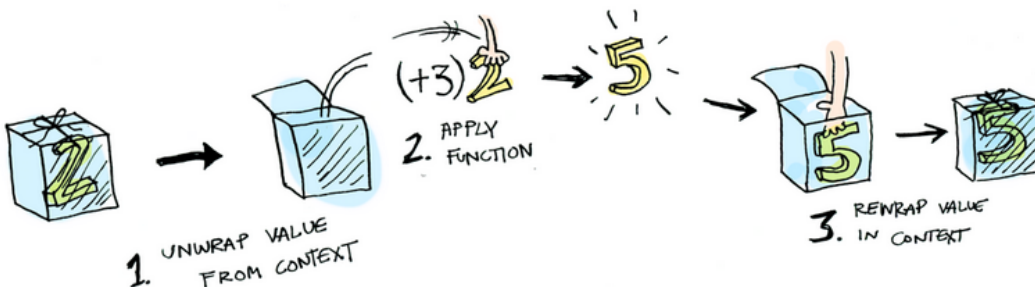
```
1 | class Functor Maybe where
2 |     fmap :: (a -> b) -> Maybe a -> Maybe b
```

因此，对于 Maybe 类型来说，它要实现的函数 fmap 的功能就是将一个函数 (a -> b) 应用到一个在 Maybe 上下文中的值 Maybe a，并返回另一个在 Maybe 上下文中的值 Maybe b。接下来，我们一起来看一下 Maybe 类型实现 Functor typeclass 的具体细节：

```
1 | instance Functor Maybe where
2 |     fmap func (Just x) = Just (func x)
3 |     fmap func Nothing  = Nothing
```

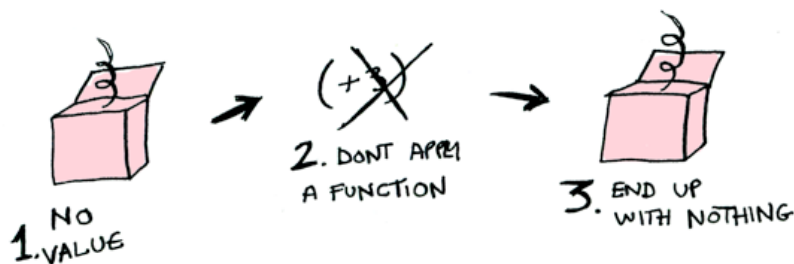
这里针对 Maybe 上下文的两种情况分别进行了处理：如果盒子中有值，即 Just x，那么就将 x 从盒子中取出，然后将函数 func 应用到 x，最后将结果放入一个相同类型的新盒子中；如果盒子为空，那么直接返回一个新的空盒子。

看到这里，我想你应该已经知道如何将一个函数应用到一个在上下文中的值了。比如前面提到的将函数 (+3) 应用到 Just 2：



```
1 | ghci> fmap (+3) (Just 2)
2 | Just 5
```

另外，值得一提的是，当我们将函数 (+3) 应用到一个空盒子，即 Nothing 时，我们将会得到一个新的空盒子：



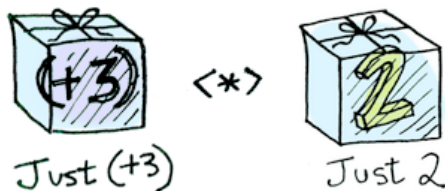
```
1 | ghci> fmap (+3) Nothing
2 | Nothing
```

### Applicative

现在，我们已经知道如何将函数 (+3) 应用到 Just 2 了。那么问题又来了，如果函数 (+3) 也在上下文中呢，比如 Maybe，此时，函数 (+3) 就变成了 Just (+3)：



那么，我们如何将一个在上下文中的函数应用到一个在上下文中的值呢？



这就是 Applicative 要干的事，详情请看下节内容。

### Applicative typeclass

同样的，我们先来看一下 Applicative typeclass 的定义：

```
1 | class Functor f => Applicative f where
2 |     pure :: a -> f a
3 |     () :: f (a -> b) -> f a -> f b
```

我们注意到，与 Functor typeclass 的定义不同的是，在 Applicative typeclass 的定义中多了一个类约束 Functor f，表示的意思是数据类型 f 要实现 Applicative typeclass 的前提条件是它必须要实现 Functor typeclass，也就是说它必须是一个 Functor。

在 Applicative typeclass 中定义了两个函数：

- pure：将一个值 a 放入上下文中；
- (<\*>)：将一个在上下文中的函数 f (a -> b) 应用到一个在上下文中的值 f a，并返回另一个在上下文中的值 f b。



注：<\*> 函数的发音我也不知道，如果有同学知道的话还请告之，谢谢。

## Maybe Applicative

同样的，我们将类型占位符 `f` 用具体类型 `Maybe` 代入，可得：

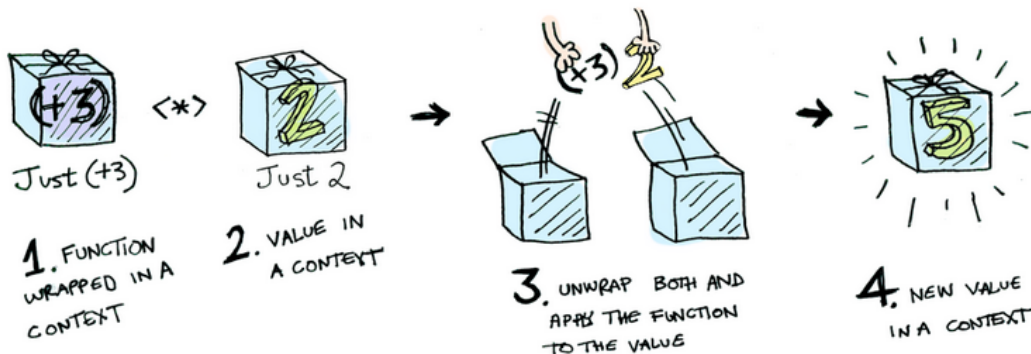
```
1 class Functor Maybe => Applicative Maybe where
2   pure :: a -> Maybe a
3   () :: Maybe (a -> b) -> Maybe a -> Maybe b
```

因此，对于 `Maybe` 类型来说，它要实现的 `pure` 函数的功能就是将一个值 `a` 放入 `Maybe` 上下文中。而 `<*>` 函数的功能则是将一个在 `Maybe` 上下文中的函数 `Maybe (a -> b)` 应用到一个在 `Maybe` 上下文中的值 `Maybe a`，并返回另一个在 `Maybe` 上下文中的值 `Maybe b`。接下来，我们一起来看一下 `Maybe` 类型实现 `Applicative` typeclass 的具体细节：

```
1 instance Applicative Maybe where
2   pure = Just
3   Nothing _ = Nothing
4   (Just func) something = fmap func something
```

`pure` 函数的实现非常简单，直接等于 `Just` 即可。而对于 `<*>` 函数的实现，我们同样需要针对 `Maybe` 上下文的两种情况分别进行处理：当装函数的盒子为空时，直接返回一个新的空盒子；当装函数的盒子不为空时，即 `Just func`，则取出 `func`，使用 `fmap` 函数直接将 `func` 应用到那个在上下文中的值，这个正是我们前面说的 `Functor` 的功能。

好了，我们接下来看一下将 `Just (+3)` 应用到 `Just 2` 的具体过程：



```
1 ghci> Just (+3) Just 2
2 Just 5
```

同样的，当我们将一个空盒子，即 `Nothing` 应用到 `Just 2` 的时候，我们将得到一个新的空盒子：

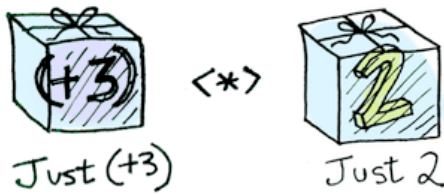
```
1 ghci> Nothing Just 2
2 Nothing
```

## Monad

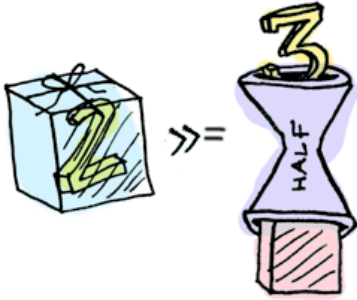
截至目前，我们已经知道了 `Functor` 的作用就是应用一个函数到一个上下文中的值：



而 `Applicative` 的作用则是应用一个上下文中的函数到一个上下文中的值：



那么 Monad 又会是什么呢？其实，Monad 的作用跟 Functor 类似，也是应用一个函数到一个上下文中的值。不同之处在于，Functor 应用的是一个接收一个普通值并且返回一个普通值的函数，而 Monad 应用的是一个接收一个普通值但是返回一个在上下文中的值的函数：



#### Monad typeclass

同样的，我们先来看一下 Monad typeclass 的定义：

```
1 class Applicative m => Monad m where
2   return :: a -> m a
3   (>>=) :: m a -> (a -> m b) -> m b
4   (>>) :: m a -> m b -> m b
5   x >> y = x >>= \_ -> y
6   fail :: String -> m a
7   fail msg = error msg
```

哇，这什么鬼，完全看不懂啊，太复杂了。兄台莫急，且听我细说。在 Monad typeclass 中定义了四个函数，分别是 return、(>>=)、(>>) 和 fail，且后两个函数 (>>) 和 fail 给出了默认实现，而在绝大多数情况下，我们都不需要去重写它们。因此，去掉这两个函数后，Monad typeclass 的定义可简化为：

```
1 class Applicative m => Monad m where
2   return :: a -> m a
3   (>>=) :: m a -> (a -> m b) -> m b
```

怎么样？现在看上去就好多了吧。跟 Applicative typeclass 的定义一样，在 Monad typeclass 的定义中也有一个类约束 Applicative m，表示的意思是一种数据类型 m 要成为 Monad 的前提条件是它必须是 Applicative。另外，其实 return 函数的功能与 Applicative 中的 pure 函数的功能是一样的，只不过换了一个名字而已，它们的作用都是将一个值 a 放入上下文中。而 (>>=) 函数的功能则是应用一个（接收一个普通值 a 但是返回一个在上下文中的值 m b 的）函数 (a -> m b) 到一个上下文中的值 m a，并返回另一个在相同上下文中的值 m b。

注：>>= 函数的发音为 bind，学习 ReactiveCocoa 的同学要注意啦。另外，>>= 函数可类比 Swift 中的 flatMap 方法。

#### Maybe Monad

同样的，我们将类型占位符 m 用具体类型 Maybe 代入，可得：

```
1 class Applicative Maybe => Monad Maybe where
2   return :: a -> Maybe a
```

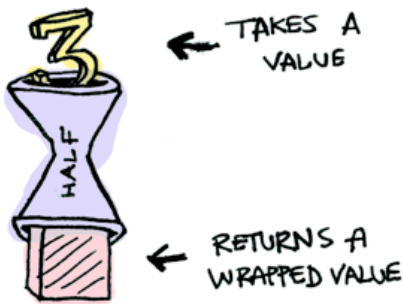
```
3 | (>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
```

相信你用盒子模型已经能够轻松地理解上面两个函数了，因此不再赘述。接下来，我们一起来看一下 Maybe 类型实现 Monad typeclass 的具体细节：

```
1 | instance Monad Maybe where
2 |     return x = Just x
3 |     Nothing >>= func = Nothing
4 |     Just x >>= func = func x
```

正如前面所说，return 函数的实现跟 pure 函数一样，直接等于 Just 函数即可，功能就是将一个值 x 放入 Maybe 盒子中，变成 Just x。同样的，对于 (>>=) 函数的实现，我们需要针对 Maybe 上下文的两种情况分别进行处理，当盒子为空时，直接返回一个新的空盒子；当盒子不为空时，即 Just x，则取出 x，直接将 func 函数应用到 x，而我们知道 func x 的结果就是一个在上下文中的值。

下面，我们一起来看一个具体的例子。我们先定义一个 half 函数，这个函数接收一个数字 x 作为参数，如果 x 是偶数，则将 x 除以 2，并将结果放入 Maybe 盒子中；如果 x 不是偶数，则返回一个空盒子：



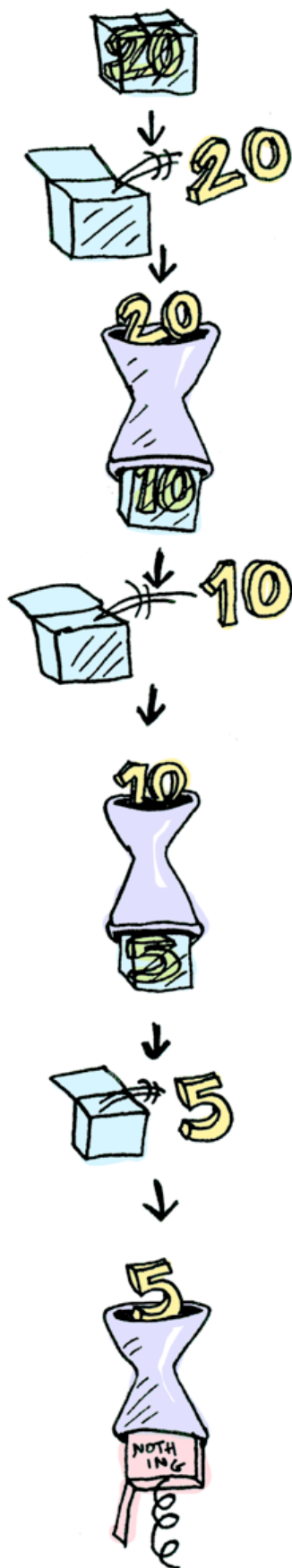
```
1 | half x = if even x
2 |         then Just (x `div` 2)
3 |         else Nothing
```

接下来，我们使用 (>>=) 函数将 half 函数应用到 Just 20，假设得到结果 y；然后继续使用 (>>=) 函数将 half 函数应用到上一步的结果 y，以此类推，看看会得到什么样的结果：

```
1 | ghci> Just 20 >>= half
2 | Just 10
3 | ghci> Just 10 >>= half
4 | Just 5
5 | ghci> Just 5 >>= half
6 | Nothing
```

看到上面的运算过程，不知道你有没有看出点什么端倪呢？上一步的输出作为下一步的输入，并且只要你愿意的话，这个过程可以无限地进行下去。我想你可能已经想到了，是的，就是链式操作。所有的操作链接起来就像是一条生产线，每一步的操作都是对输入进行加工，然后产生输出，整个操作过程可以看作是对最初的原材料 Just 20 进行加工并最终生产出成品 Nothing 的过程：

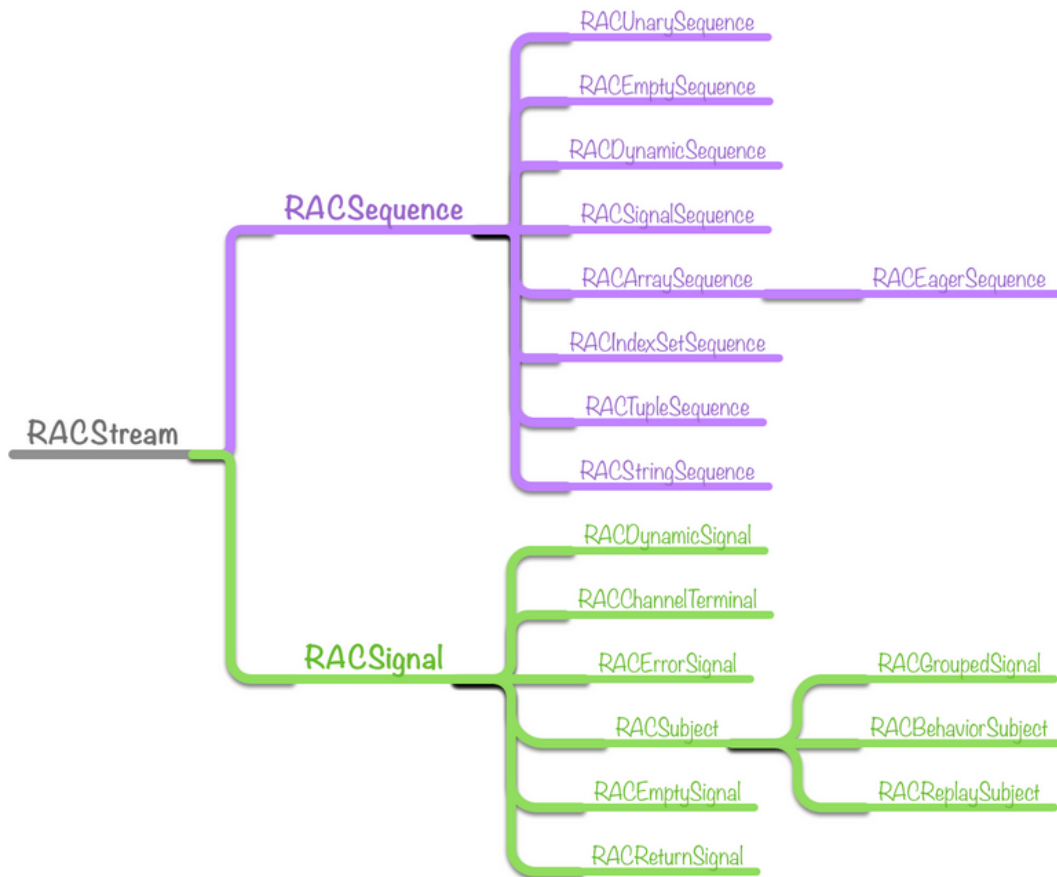




动处理上下文，而我们只需要关心真正的值就可以了。

## ReactiveCocoa

现在，我们已经知道 Monad 是什么了，它就是一种实现了 Monad typeclass 的数据类型。那么它有什么具体的应用呢？你总不能让我们都来做理论研究吧。既然如此，那我们就只好祭出 Objective-C 中的神器，ReactiveCocoa，它就是根据 Monad 的概念搭建起来的。下面是 RACStream 的继承结构图：



RACStream 是 ReactiveCocoa 中最核心的类，它就是一个 Monad：

```

1  /// An abstract class representing any stream of values.
2  ///
3  /// This class represents a monad, upon which many stream-based operations can
4  /// be built.
5  ///
6  /// When subclassing RACStream, only the methods in the main @interface body ne
7  /// to be overridden.
8  @interface RACStream : NSObject
9  /// Lifts `value` into the stream monad.
10 ///
11 /// Returns a stream containing only the given value.
12 + (instancetype) return:(id) value;
13 /// Lazily binds a block to the values in the receiver.
14 ///
15 /// This should only be used if you need to terminate the bind early, or close
16 /// over some state. -flattenMap: is more appropriate for all other cases.
17 ///
18 /// block - A block returning a RACStreamBindBlock. This block will be invoked
19 ///         each time the bound stream is re-evaluated. This block must not be
20 ///         nil or return nil.
21 ///
22 /// Returns a new stream which represents the combined result of all lazy
23 /// applications of `block`.
24 - (instancetype) bind:(RACStreamBindBlock (^)(void)) block;
25 @end
  
```

我们可以看到，在 RACStream 中定义了两个看上去非常眼熟的方法：

1. `+(instancetype)return:(id)value;`;
2. `-(instancetype)bind:(RACStreamBindBlock (^)(void))block;`。

其中，`return:` 方法的功能就是将一个值 `value` 放入 RACStream 上下文中；而 `bind:` 方法的功能则是将一个 RACStreamBindBlock 类型的 `block` 应用到一个在 RACStream 上下文中的值（receiver），并返回另一个在 RACStream 上下文中的值。注，RACStreamBindBlock 类型的 `block` 就是一个接收一个普通值 `value` 但是返回一个在 RACStream 上下文中的值的“函数”：

```
1  /// A block which accepts a value from a RACStream and returns a new instance
2  /// of the same stream class.
3  ///
4  /// Setting `stop` to `YES` will cause the bind to terminate after the returned
5  /// value. Returning `nil` will result in immediate termination.
6  typedef RACStream * (^RACStreamBindBlock)(id value, BOOL *stop);
```

接下来，为了加深理解，我们一起来对比一下 Monad typeclass 的定义：

```
1  class Applicative m => Monad m where
2      return :: a -> m a
3      (>=>) :: m a -> (a -> m b) -> m b
```

同样的，我们将类型占位符 `m` 用 RACStream 代入，可得：

```
1  class Applicative RACStream => Monad RACStream where
2      return :: a -> RACStream a
3      (>=>) :: RACStream a -> (a -> RACStream b) -> RACStream b
```

其中，`return :: a -> RACStream a` 就对应 `+(instancetype)return:(id)value;`，而 `(>=>) :: RACStream a -> (a -> RACStream b) -> RACStream b` 则对应 `-(instancetype)bind:(RACStreamBindBlock (^)(void))block;`。注：我们前面已经提到过了，`>=>` 函数的发音就是 `bind`。因此，ReactiveCocoa 便有了下面的玩法：

```
1  RACSignal *signal2 = [[[signal1
2      bind:block1]
3      bind:block2]
4      bind:block3];
```

Monad 就像是 ReactiveCocoa 中的太极，太极生两仪，两仪生四象，四象生八卦。至此，我们已经知道了 ReactiveCocoa 中最核心的原理，而更多关于 ReactiveCocoa 的内容我们将在后续的源码解析中再进行介绍，敬请期待。

## 总结

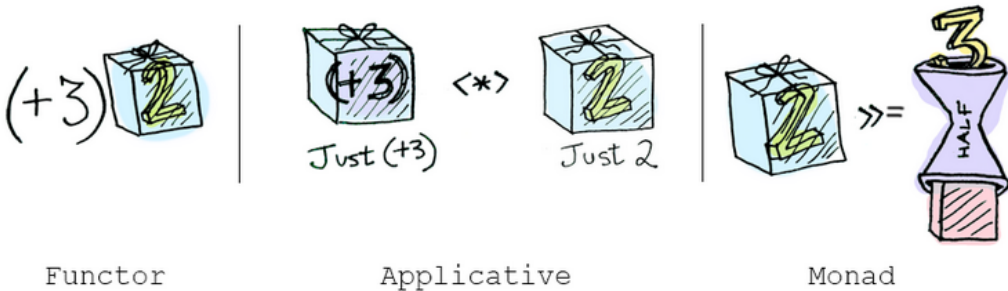
Functor、Applicative 和 Monad 是什么：

1. 一个 Functor 就是一种实现了 Functor typeclass 的数据类型；
2. 一个 Applicative 就是一种实现了 Applicative typeclass 的数据类型；
3. 一个 Monad 就是一种实现了 Monad typeclass 的数据类型。

Functor、Applicative 和 Monad 三者之间的联系：

1. Applicative 是增强型的 Functor，一种数据类型要成为 Applicative 的前提条件是它必须是 Functor；
2. Monad 是增强型的 Applicative，一种数据类型要成为 Monad 的前提条件是它必须是 Applicative。

Functor、Applicative 和 Monad 三者之间的区别：



1. Functor：使用 fmap 应用一个函数到一个上下文中的值；
2. Applicative：使用 <\*> 应用一个上下文中的函数到一个上下文中的值；
3. Monad：使用 >=> 应用一个接收一个普通值但是返回一个在上下文中的值的函数到一个上下文中的值。

此外，我们还介绍了一种非常有意思的数据类型 Maybe，它实现了 Functor typeclass、Applicative typeclass 和 Monad typeclass，所以它同时是 Functor、Applicative 和 Monad。

以上就是本文的全部内容，希望可以对你有所帮助，Good luck！

#### 参考链接

- <http://learnyouahaskell.com/chapters>
- <https://downloads.haskell.org/~ghc/latest/docs/html/libraries>
- [http://adit.io/posts/2013-04-17-functors,applicatives,and\\_monads\\_in\\_pictures.html](http://adit.io/posts/2013-04-17-functors,applicatives,and_monads_in_pictures.html)



微信扫一扫  
订阅每日移动开发及APP推广热点资讯  
公众号：CocoaChina

我要投稿

收藏文章

分享到：

上一篇：源码推荐(11.09)：仿QQ抽屉效果，Swift写的简单的引导页

下一篇：源码推荐(11.10)：一个类似于安卓ViewPager的开源库，数据的持久化

#### 相关资讯

Swift中的仿函数(Functor)和Monad模型

ReactiveCocoa 用 RACSignal 替代 Delegate

论坛源码推荐（10.27）：Swift/Objective-C行为驱动开发

Swift 的 函数式编程

What will your legacy be?

Find out more >




UNIVERSITY OF  
BATH

我来说两句



您还没有登录！请 [登录](#) 或 [注册](#)


所有评论 (2)


- 


yuedong56

2015-11-11 09:06:38

什么鬼？没看出跟iOS有啥关系来

 0


 0


回复
- 

a23234836

2015-11-11 03:09:01

写的一渣，烂翻译组织

 0

 0

回复

[关于我们](#) [商务合作](#) [联系我们](#) [合作伙伴](#)

北京触控科技有限公司版权所有

©2015 Chukong Technologies, Inc.

京ICP备 11006519号 京ICP证 100954号 京公网安备11010502020289  京网文[2012]0426-138号