理解 MONAD

唐巧@小猿搜题
http://weibo.com/tangqiaoboy

基础知识—：数组的MAP

```
let arr = [1, 2, 4]
// arr = [1, 2, 4]
```

```
let brr = arr.map {
    "No." + String($0)
}
// brr = ["No.1", "No.2", "No.4"]
```

基础知识二：数组的FLATMAP

```
let arr = [[1, 2, 3], [6, 5, 4]]
let brr = arr.flatMap {
        $0
}
// brr = [1, 2, 3, 6, 5, 4]
```

基础知识二：数组的FLATMAP

```
let arr: [Int?] = [1, 2, nil, 4, nil, 5]
let brr = arr.flatMap { $0 }
// brr = [1, 2, 4, 5]
```

基础知识三：OPTIONAL的MAP

```swift
let a1: Int? = 3
let b1 = a1.map{ $0 * 2 }
// b1 = 6
```

```swift
let a2: Int? = nil
let b2 = a2.map{ $0 * 2 }
// b2 = nil
```

基础知识四：OPTIONAL的FLATMAP

```
let s: String? = "abc"
let v = s.flatMap { (a: String) -> Int? in
    return Int(a)
}
```

基础知识五：类型转换

```
let s2: String? = nil
let s1: String? = "abc"
```

基础知识五：类型转换

```
public enum Optional<Wrapped> :
    _Reflectable, NilLiteralConvertible {
    case None
    case Some(Wrapped)

    @available(*, unavailable, renamed="Wrapped")
    public typealias T = Wrapped

    /// Construct a `nil` instance.
    @_transparent
    public init() { self = .None }

    /// Construct a non-`nil` instance that stores `some`.
    @_transparent
    public init(_ some: Wrapped) { self = .Some(some) }
}
```

MONAD是什么?

MONAD是什么?

链式调用的编程范式

链式调用的编程范式

数组的链式调用

```
let arr = [1, 3, 2]

let brr = arr.map {
    $0 * 2
} .map {
    "this is " + String($0)
} .map {
    $0.uppercaseString
}
```

Optional 的链式调用

```
let tq: Int? = 1
let b = tq.map {
    $0 * 2
}.map {
    "abc" + String($0)
}
```

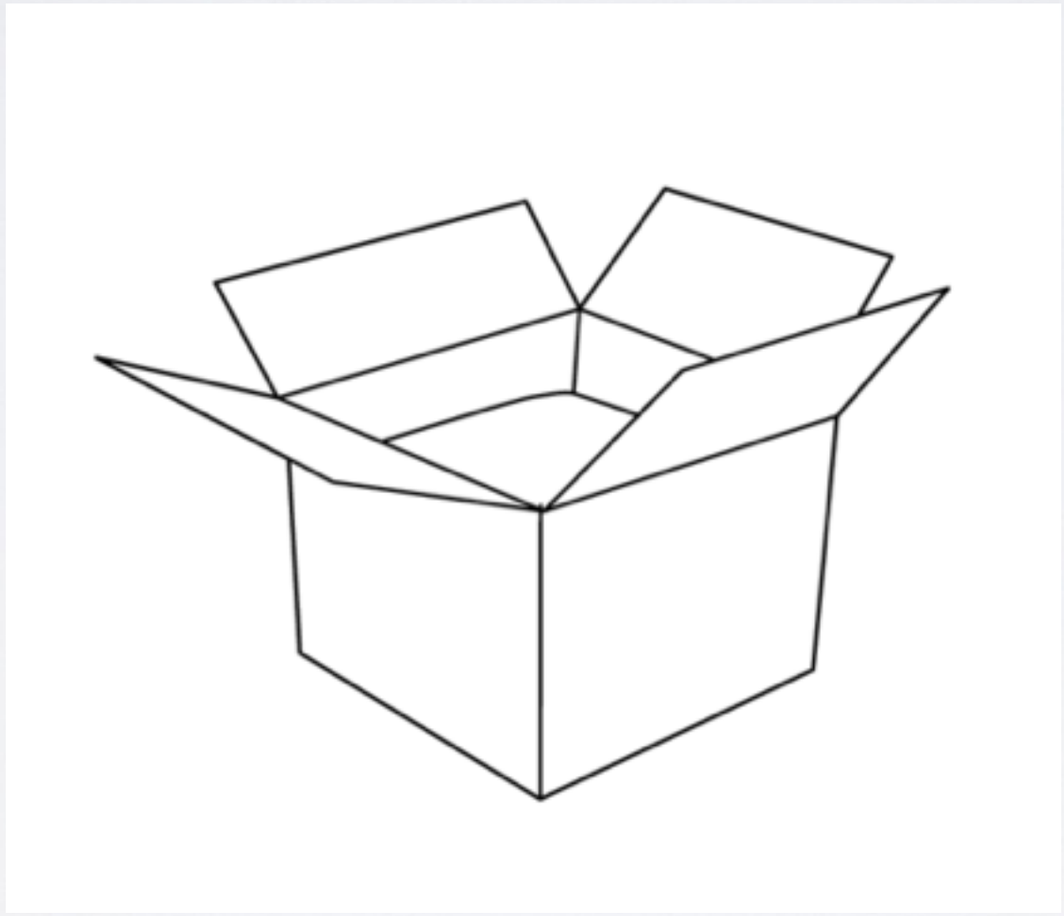
链式调用有什么好处?

```
    TTRequest *req1 = [TTRequest requestWithUrlString:@"url1"];
    [req1 startWithCompletionBlockWithSuccess:^(__kindof YTKBaseRequest *request) {
        TTRequest *req2 = [TTRequest requestWithUrlString:[NSString stringWithFormat:@"%@", req1.result]];
        [req2 startWithCompletionBlockWithSuccess:^(__kindof YTKBaseRequest *request) {
            TTRequest *req3 = [TTRequest requestWithUrlString:[NSString stringWithFormat:@"%@",
req2.result]];
            [req3 startWithCompletionBlockWithSuccess:^(__kindof YTKBaseRequest *request) {
                ;
            } failure:^(__kindof YTKBaseRequest *request) {
                [TTAlertUtils showAutoHideHint:@"网络错误" inView:self.view];
            }];
        } failure:^(__kindof YTKBaseRequest *request) {
            [TTAlertUtils showAutoHideHint:@"网络错误" inView:self.view];
        }];
    } failure:^(__kindof YTKBaseRequest *request) {
        [TTAlertUtils showAutoHideHint:@"网络错误" inView:self.view];
    }];
```

链式调用的编程范式

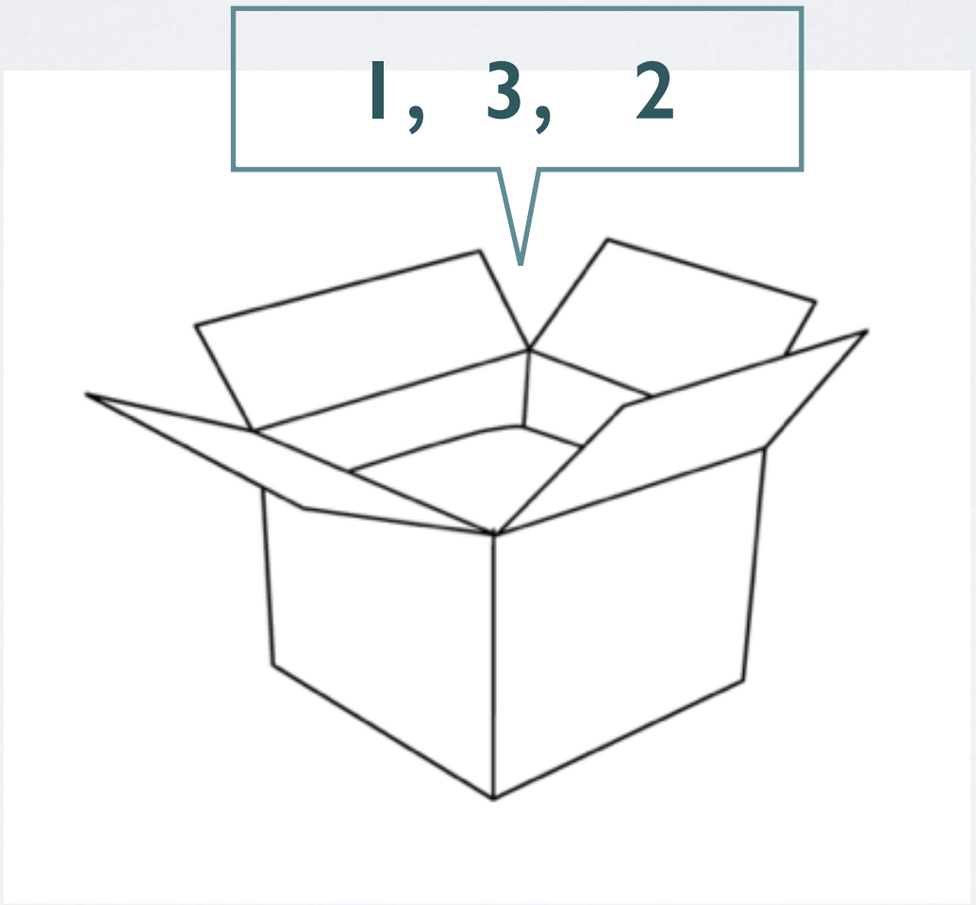
一种更 General 的设计模式

盒子：封装的数据



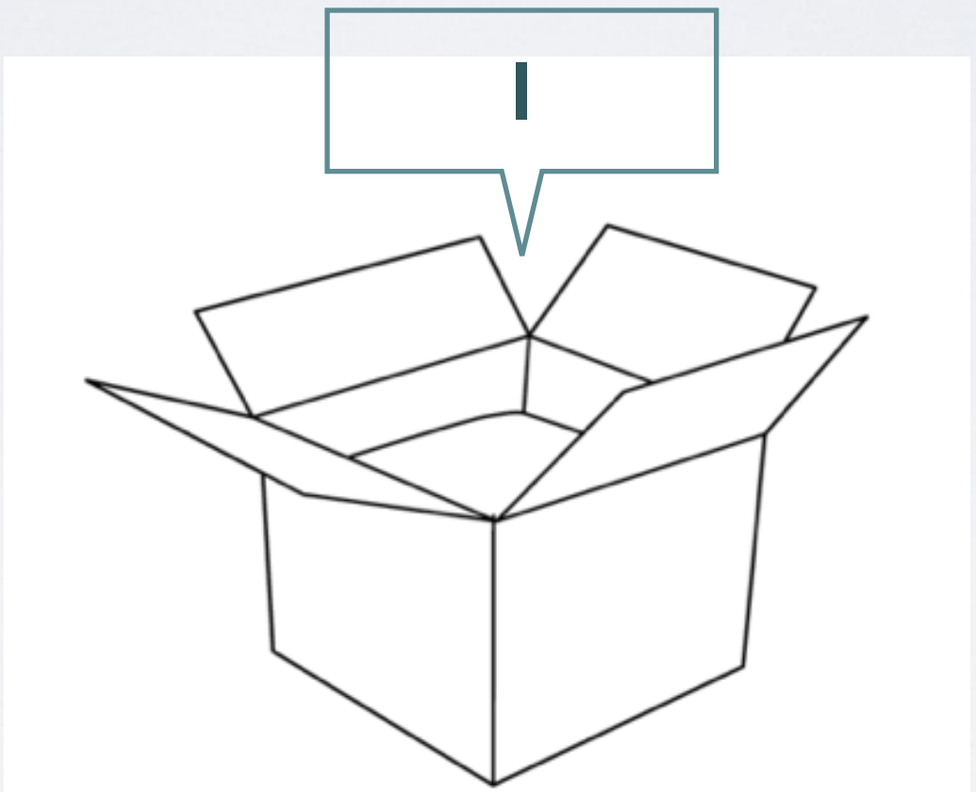
数组形式的盒子

```
let arr = [1, 3, 2]
```



OPTIONAL形式的盒子

```
let tq: Int? = 1
```

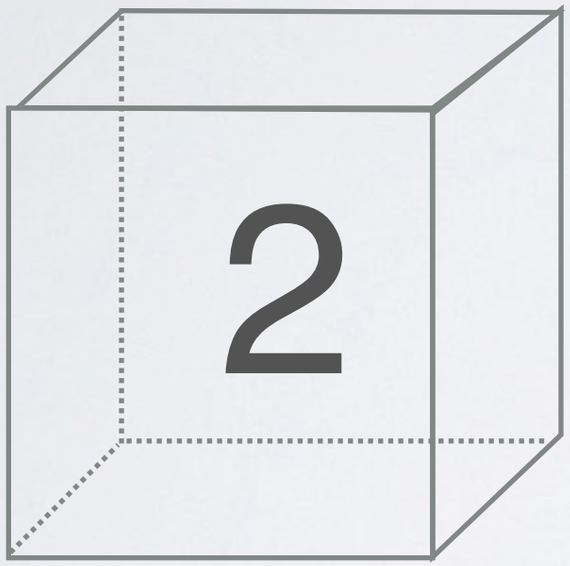


```
let arr = [1, 3, 2]
let tq: Int? = 1
```

```
enum Result<T> {
    case Success(T)
    case Failure(ErrorType)
}
```

所有可以被“打开”的数据

困境：封装的数据不能直接计算



+ 1 = ?

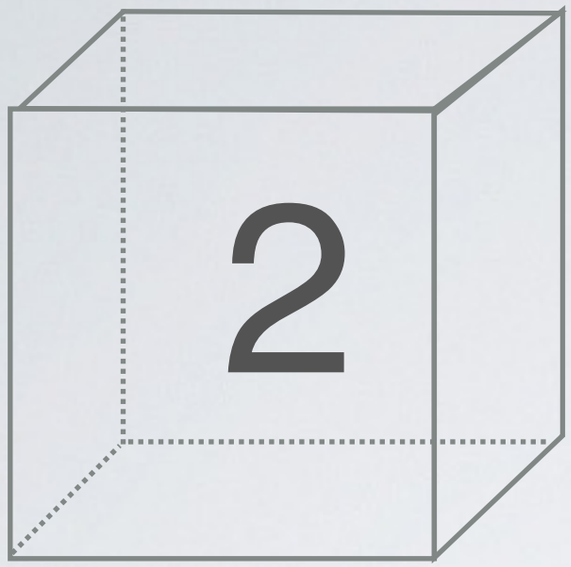
困境：封装的数据不能直接计算

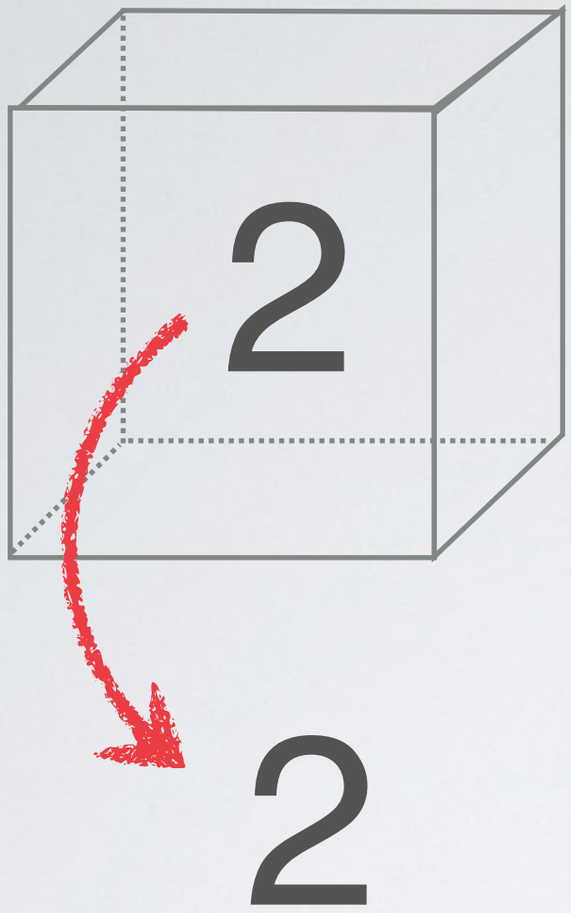
```
1
2 let a : Int? = 1
3 let b = a + 1
```

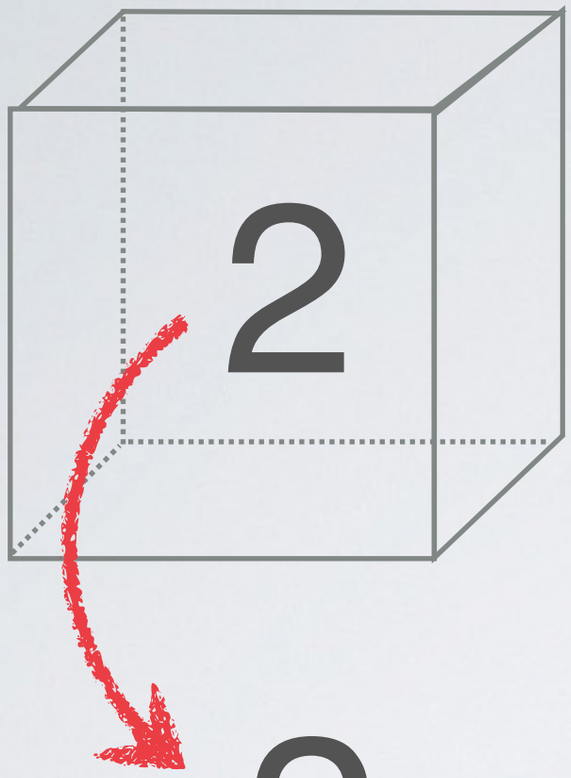
Value of optional type 'Int?' not unwrapped;

困境：封装的数据不能直接计算

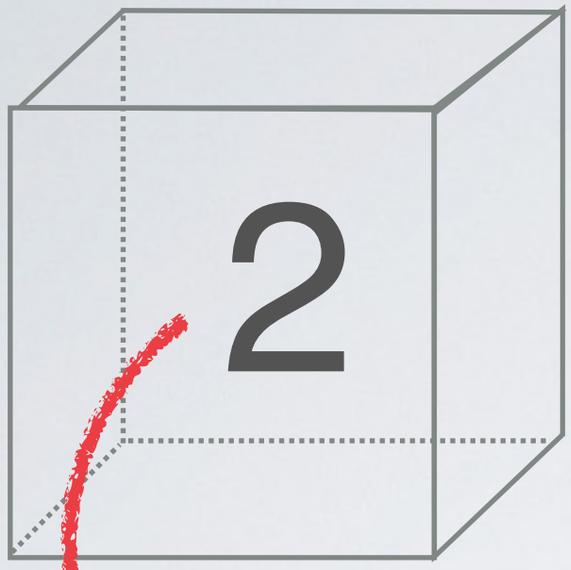
```
let a : Int? = 1
var b: Int?
if let a = a {
      b = a + 1
} else {
      b = nil
}
```


2

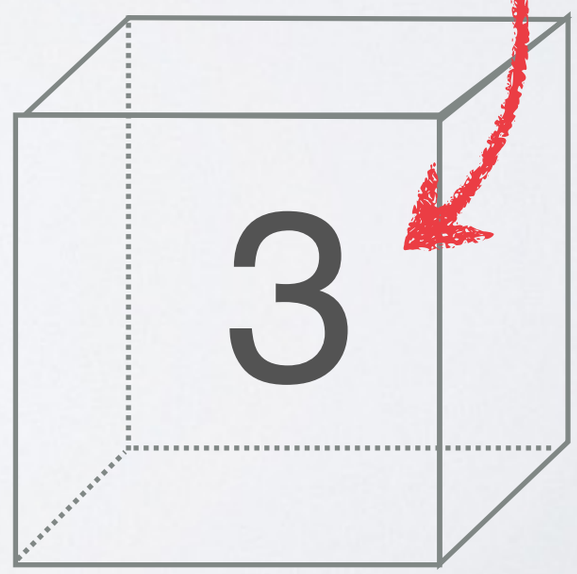

2
2

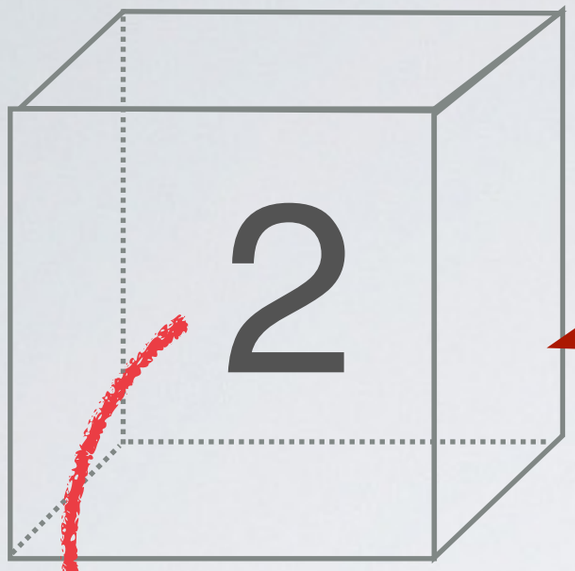


2 + 1 = 3



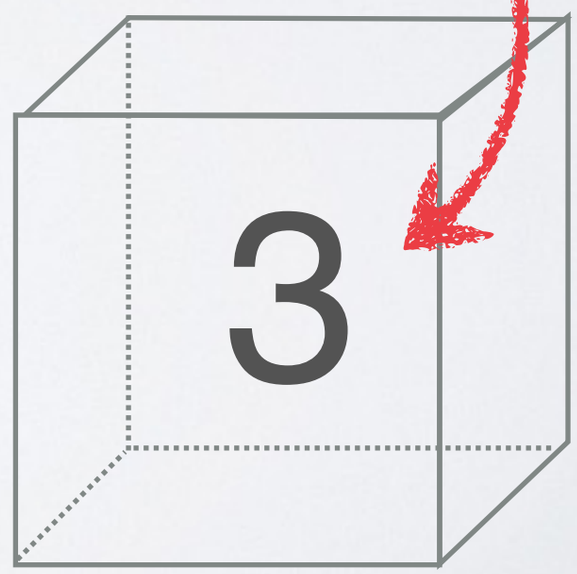
2 + 1 = 3

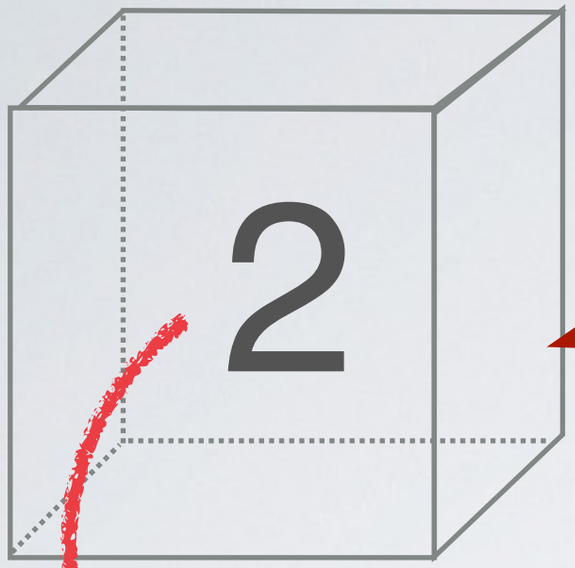




打开盒子

2 + 1 = 3

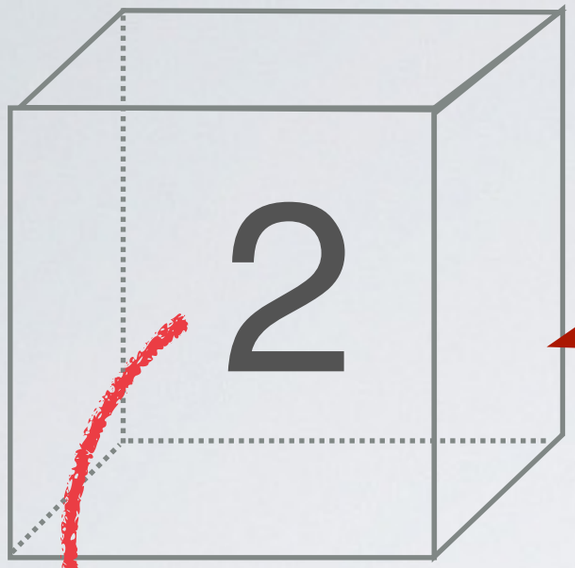




打开盒子

计算

2 + 1 = 3



打开盒子

计算

2 + 1 = 3

结果放到新盒子中

- 计算前需要打开盒子
- 计算之后再封装盒子

- 计算之前的打开能不能是自动的?
- 计算之后的封装能不能是自动的?

这就是MAP

```
let arr = [1, 3, 2]

let brr = arr.map {
        (element: Int) -> Int in
        return element * 2
}
```

这就是MAP

```
let arr = [1, 3, 2]

let brr = arr.map {
    (element: Int) -> Int in
    return element * 2
}
```

这就是MAP

```
let arr = [1, 3, 2]

let brr = arr.map {
    (element: Int) -> Int in
    return element * 2
}
```

自动将数组中的数据取出来，
算完之后再放到新数组中去

OPTIONAL 的 MAP

```
let a1: Int? = 3
let b1 = a1.map{ (e: Int) -> Int in
    return e * 2
}
```

```
let a1: Int? = 3
let b1 = a1.map{(e: Int) -> Int in
    return e * 2
}
```

回顾

- 什么是盒子?

- 什么是 map ?

Talk is cheap. Show me the code.

— Linus Torvalds

数组的MAP源码

```
public func map<T>(@noescape transform:
    (Generator.Element) throws -> T)
    rethrows -> [T] {
        let count: Int = numericCast(self.count)
        if count == 0 {
            return []
        }

        var result = ContiguousArray<T>()
        result.reserveCapacity(count)

        var i = self.startIndex

        for _ in 0..<count {
            result.append(try transform(self[i]))
            i = i.successor()
        }

        _expectEnd(i, self)
        return Array(result)
}
```

```
public func map<T>(@noescape transform:
    (Generator.Element) throws -> T)
    rethrows -> [T] {
        let count: Int = numericCast(self.count)
        if count == 0 {
            return []
        }

        var result = ContiguousArray<T>()
        result.reserveCapacity(count)

        var i = self.startIndex

        for _ in 0..<count {
            result.append(try transform(self[i]))
            i = i.successor()
        }

        _expectEnd(i, self)
        return Array(result)
}
```

打开盒子

```
public func map<T>(@noescape transform:
    (Generator.Element) throws -> T)
    rethrows -> [T] {
        let count: Int = numericCast(self.count)
        if count == 0 {
            return []
        }

        var result = ContiguousArray<T>()
        result.reserveCapacity(count)

        var i = self.startIndex

        for _ in 0..<count {
            result.append(try transform(self[i]))
            i = i.successor()
        }

        _expectEnd(i, self)
        return Array(result)
}
```

打开盒子

结果放到新盒子中

OPTIONAL的MAP源码

```
public func map<U>(@noescape f: (Wrapped) throws -> U)
    rethrows -> U? {
        switch self {
        case .Some(let y):
            return .Some(try f(y))
        case .None:
            return .None
        }
}
```

打开盒子

```
public func map<U>(@noescape f: (Wrapped) throws -> U)
    rethrows -> U? {
        switch self {
        case .Some(let y):
            return .Some(try f(y))
        case .None:
            return .None
        }
}
```

打开盒子

```
public func map<U>(@noescape f: (Wrapped) throws -> U)
    rethrows -> U? {
        switch self {
        case .Some(let y):
            return .Some(try f(y))
        case .None:
            return .None
        }
}
```

结果放到盒子中

为什么MAP不能解决所有问题?

为什么MAP不能解决所有问题?

计算之后的封装不一定能自动。

自动封装的问题

```
let tq: Int? = 1
let b = tq.map { (a: Int) -> Int? in
    if a % 2 == 0 {
        return a
    } else {
        return Optional<Int>.None
    }
}
if let _ = b {
    print("not nil")
}
```

自动封装的问题

```
let tq: Int? = 1
let b = tq.map { (a: Int) -> Int? in
    if a % 2 == 0 {
        return a
    } else {
        return Optional<Int>.None
    }
}
if let _ = b {
    print("not nil")
}
```

自动转换

对比源码

打开盒子

```
public func map<U>(@noescape f: (Wrapped) throws -> U)
    rethrows -> U? {
        switch self {
        case .Some(let y):
            return .Some(try f(y))
        case .None:
            return .None
        }
}
```

结果放到盒子中

```swift
let tq: Int? = 1
let b = tq.map { (a: Int) -> Int? in
    if a % 2 == 0 {
        return a
    } else {
        return Optional<Int>.None
    }
}
if let _ = b {
    print("not nil")
}
```

```swift
public func map<U>(@noescape f:
(Wrapped) throws -> U)
    rethrows -> U? {
        switch self {
        case .Some(let y):
            return .Some(try f(y))
        case .None:
            return .None
        }
}
```

```swift
let tq: Int? = 1
let b = tq.map { (a: Int) -> Int? in
    if a % 2 == 0 {
        return a
    } else {
        return Optional<Int>.None
    }
}
if let _ = b {
    print("not nil")
}
```

self 为 Some(1)

```swift
public func map<U>(@noescape f:
(Wrapped) throws -> U)
    rethrows -> U? {
        switch self {
        case .Some(let y):
            return .Some(try f(y))
        case .None:
            return .None
        }
}
```

```swift
let tq: Int? = 1
let b = tq.map { (a: Int) -> Int? in
    if a % 2 == 0 {
        return a
    } else {
        return Optional<Int>.None
    }
}
if let _ = b {
    print("not nil")
}
```

self 为 Some(1)

self 有值，y 为 1

```swift
public func map<U>(@noescape f:
(Wrapped) throws -> U)
    rethrows -> U? {
        switch self {
        case .Some(let y):
            return .Some(try f(y))
        case .None:
            return .None
        }
}
```

```swift
let tq: Int? = 1
let b = tq.map { (a: Int) -> Int? in
    if a % 2 == 0 {
        return a
    } else {
        return Optional<Int>.None
    }
}
if let _ = b {
    print("not nil")
}
```

self 为 Some(1)

self 有值，y 为 1

调用闭包f，得到：
Optional<Int>.None

```swift
public func map<U>(@noescape f:
(Wrapped) throws -> U)
    rethrows -> U? {
        switch self {
        case .Some(let y):
            return .Some(try f(y))
        case .None:
            return .None
        }
}
```

```swift
let tq: Int? = 1
let b = tq.map { (a: Int) -> Int? in
    if a % 2 == 0 {
        return a
    } else {
        return Optional<Int>.None
    }
}
if let _ = b {
    print("not nil")
}
```

```swift
public func map<U>(@noescape f:
(Wrapped) throws -> U)
    rethrows -> U? {
        switch self {
        case .Some(let y):
            return .Some(try f(y))
        case .None:
            return .None
        }
}
```

self 为 Some(1)

self 有值，y 为 1

调用闭包f，得到：
Optional<Int>.None

将Optional<Int>.None
放入 .Some 中

```swift
let tq: Int? = 1
let b = tq.map { (a: Int) -> Int? in
    if a % 2 == 0 {
        return a
    } else {
        return Optional<Int>.None
    }
}
if let _ = b {
    print("not nil")
}
```

```swift
public func map<U>(@noescape f:
(Wrapped) throws -> U)
    rethrows -> U? {
        switch self {
        case .Some(let y):
            return .Some(try f(y))
        case .None:
            return .None
        }
}
```

self 为 Some(1)

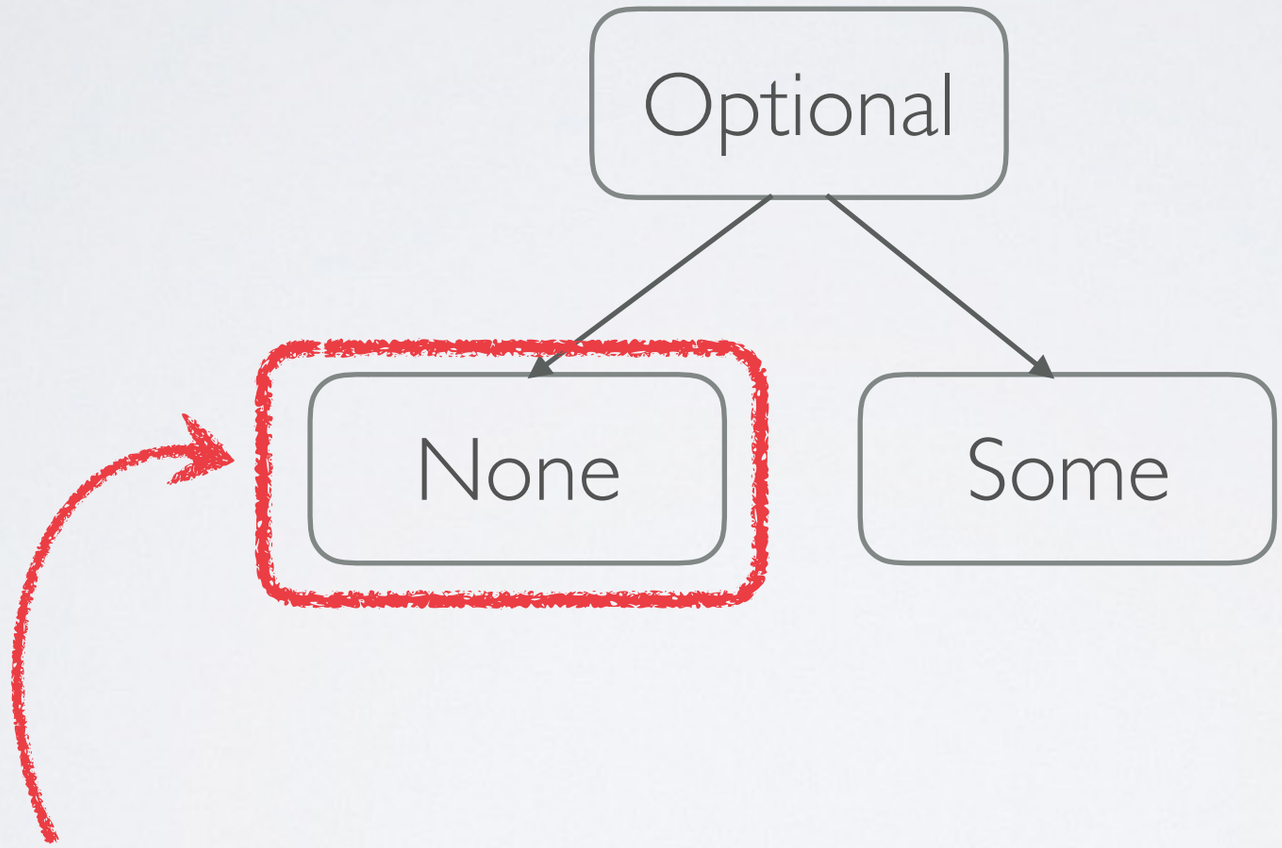
self 有值，y 为 1

调用闭包f，得到：
Optional<Int>.None

将Optional<Int>.None
放入 .Some 中

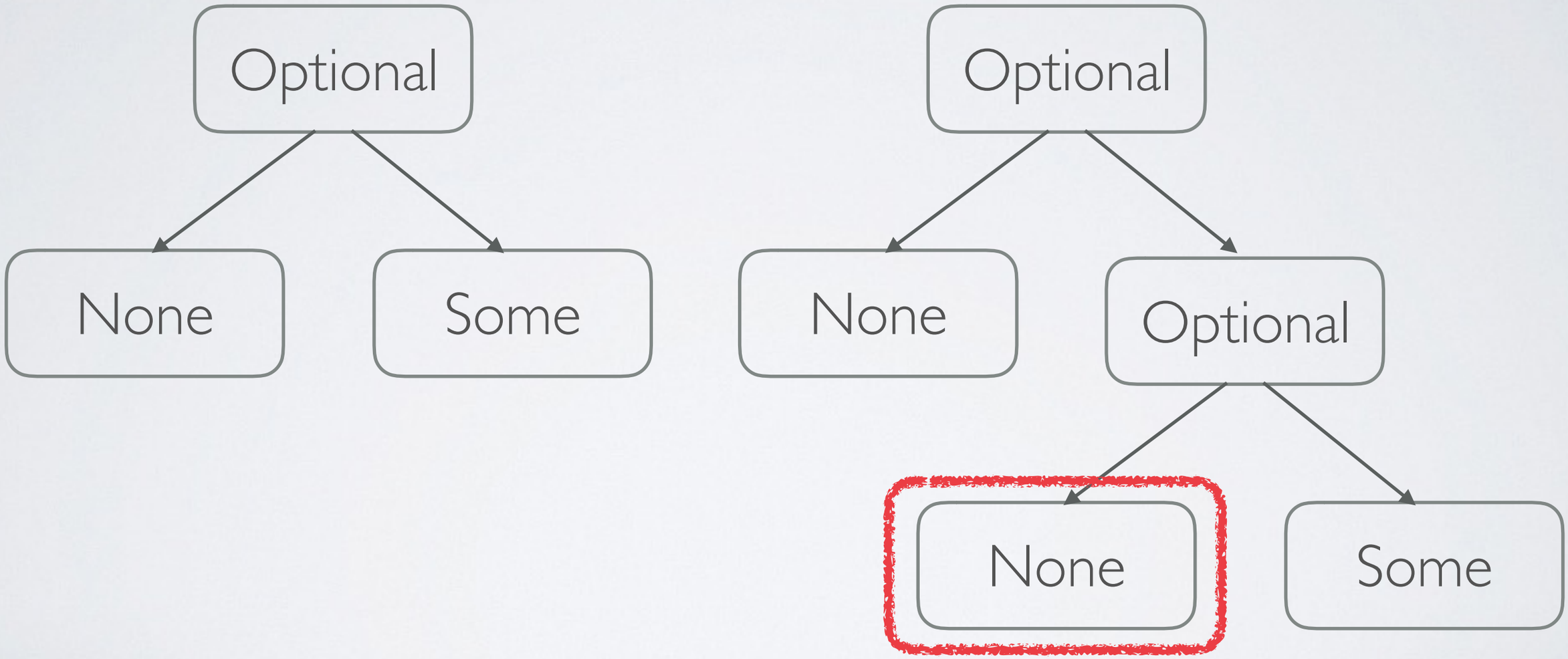
产生多重Optional，
if let 判断失效

多重OPTIONAL




```
Optional<Int>.None
```

多重OPTIONAL



```swift
let tq: Int? = 1
let b = tq.map { (a: Int) -> Int? in
    if a % 2 == 0 {
        return a
    } else {
        return Optional<Int>.None
    }
}
if let _ = b {
    print("not nil")
}
```

```swift
public func map<U>(@noescape f:
(Wrapped) throws -> U)
    rethrows -> U? {
        switch self {
        case .Some(let y):
            return .Some(try f(y))
        case .None:
            return .None
        }
}
```

self 为 Some(1)

self 有值，y 为 1

调用闭包f，得到：
Optional<Int>.None

将Optional<Int>.None
放入 .Some 中

产生多重Optional，
if let 判断失效

我们应该怎么改这段代码?

如果有一个朋友，把送你的礼物包了两层的盒子，你怎么得到这个礼物?

如果有一个朋友，把送你的礼物包了两层的盒子，你怎么得到这个礼物?

对！再打开一次不就行了。

```
8  let tq: Int? = 1                                          1
9  let b = tq.map { (a: Int) -> Int? in                      nil
10     if a % 2 == 0 {
11         return a
12     } else {
13         return Optional<Int>.None                         nil
14     }
15 }
16
17 let c: Int? = b!                                         nil
18
19 if let _ = c {
20     print("not nil")
21 } else {
22     print("nil")                                          "nil\n"
23 }
```

有没有那种每次MAP完帮我自动把两层盒子打开的函数?

有没有那种每次MAP完帮我自动把两层盒子打开的函数?

flatMap

将map改成flatMap

```
let tq: Int? = 1
let b = tq.flatMap { (a: Int) -> Int? in
    if a % 2 == 0 {
        return a    // return Some(a)
    } else {
        return nil  // return .None
    }
}
if let _ = b {
    print("not nil")
}
```

复习

- 计算之后不自动封装的模式，就是 monad。
- flatMap 就是一种 monad。

Talk is cheap. Show me the code.

— Linus Torvalds

OPTIONAL的FLATMAP

```
public func flatMap<U>(@noescape f: (Wrapped) throws -> U?)
    rethrows -> U? {
        switch self {
        case .Some(let y):
            return try f(y)
        case .None:
            return .None
        }
}
```

OPTIONAL的FLATMAP

打开盒子

```
public func flatMap<U>(@noescape f: (Wrapped) throws -> U?)
    rethrows -> U? {
        switch self {
        case .Some(let y):
            return try f(y)
        case .None:
            return .None
        }
}
```

OPTIONAL的FLATMAP

打开盒子

```
public func flatMap<U>(@noescape f: (Wrapped) throws -> U?)
    rethrows -> U? {
        switch self {
        case .Some(let y):
            return try f(y)
        case .None:
            return .None
        }
}
```

直接返回新盒子

对比一下

```
public func map<U>(@noescape f: (Wrapped) throws -> U)
    rethrows -> U? {
        switch self {
        case .Some(let y):
            return .Some(try f(y))
        case .None:
            return .None
        }
}
```

因为函数 f 返回的是值，
所以把它放到盒子中。

```
public func flatMap<U>(@noescape f: (Wrapped) throws -> U?)
    rethrows -> U? {
        switch self {
        case .Some(let y):
            return try f(y)
        case .None:
            return .None
        }
}
```

因为函数 f 已经返回的是盒子，
所以就不再把它放到新盒子中了。

数组的FLATMAP

```
public func flatMap<T>(
    @noescape transform: (${GElement}) throws -> T?
    ) rethrows -> [T] {
        var result: [T] = []
        for element in self {
            if let newElement = try transform(element) {
                result.append(newElement)
            }
        }
        return result
}
```

数组的FLATMAP

打开盒子

```
public func flatMap<T>(
    @noescape transform: (${GElement}) throws -> T?
    ) rethrows -> [T] {
        var result: [T] = []
        for element in self {
            if let newElement = try transform(element) {
                result.append(newElement)
            }
        }
        return result
}
```

数组的FLATMAP

打开盒子

transform函数
返回的结果是另一个盒子

```
public func flatMap<T>(
    @noescape transform: (${GElement}) throws -> T?
    ) rethrows -> [T] {
        var result: [T] = []
        for element in self {
            if let newElement = try transform(element) {
                result.append(newElement)
            }
        }
        return result
}
```

数组的FLATMAP

打开盒子

transform函数
返回的结果是另一个盒子

```
public func flatMap<T>(
    @noescape transform: (${GElement}) throws -> T?
    ) rethrows -> [T] {
        var result: [T] = []
        for element in self {
            if let newElement = try transform(element) {
                result.append(newElement)
            }
        }
        return result
}
```

盒子被打开，然后放到另一个盒子中

数组的FLATMAP（二）

```
public func flatMap<S : SequenceType>(
    transform: (${GElement}) throws -> S
    ) rethrows -> [S.${GElement}] {
        var result: [S.${GElement}] = []
        for element in self {
            result.appendContentsOf(try transform(element))
        }
        return result
}
```

数组的FLATMAP（二）

打开盒子

```
public func flatMap<S : SequenceType>(
    transform: (${GElement}) throws -> S
    ) rethrows -> [S.${GElement}] {
        var result: [S.${GElement}] = []
        for element in self {
            result.appendContentsOf(try transform(element))
        }
        return result
}
```

数组的FLATMAP（二）

打开盒子

transform函数
返回的结果是另一个盒子

```
public func flatMap<S : SequenceType>(
    transform: (${GElement}) throws -> S
    ) rethrows -> [S.${GElement}] {
        var result: [S.${GElement}] = []
        for element in self {
            result.appendContentsOf(try transform(element))
        }
        return result
}
```

数组的FLATMAP（二）

打开盒子

transform函数
返回的结果是另一个盒子

```
public func flatMap<S : SequenceType>(
    transform: (${GElement}) throws -> S
    ) rethrows -> [S.${GElement}] {
        var result: [S.${GElement}] = []
        for element in self {
            result.appendContentsOf(try transform(element))
        }
        return result
}
```

盒子被打开，然后放到另一个盒子中

回顾

- Monad：对一种封装过的值，使用 flatMap 函数。
- Functor：对一种封装过的值，使用 map 函数。

回顾

- flatMap：
  - 对自己解包，然后应用到一个闭包 F 上。
  - 这个闭包 F：接受一个「未封装的值」，返回一个盒子。
- map：
  - 对自己解包，然后应用到一个闭包 F 上。
  - 这个闭包 F：接受一个「未封装的值」，返回一个「未封装的值」。

函数是一等公民

- 如果把函数放进盒子里呢?

APPLICATIVE

```
extension Optional {
    func apply<U>(f: (T -> U)?) -> U? {
        switch f {
        case .Some(let someF): return self.map(someF)
        case .None: return .None
        }
    }
}
```

APPLICATIVE

```
extension Optional {
    func apply<U>(f: (T -> U)?) -> U? {
        switch f {
        case .Some(let someF): return self.map(someF)
        case .None: return .None
        }
    }
}
```

APPLICATIVE

```
extension Array {
    func apply<U>(fs: [Element -> U]) -> [U] {
        var result = [U]()
        for f in fs {
            for element in self.map(f) {
                result.append(element)
            }
        }
        return result
    }
}
```

APPLICATIVE

```
extension Array {
    func apply<U>(fs: [Element -> U]) -> [U] {
        var result = [U]()
        for f in fs {
            for element in self.map(f) {
                result.append(element)
            }
        }
        return result
    }
}
```

其它例子

- ReactiveCocoa
- Promise

REACTIVECOCOA

```
extension SignalType {

    public func flatMap<U>(strategy: FlattenStrategy,
        transform: Value -> SignalProducer<U, Error>)
        -> Signal<U, Error> {
            return map(transform).flatten(strategy)
    }

    public func flatMap<U>(strategy: FlattenStrategy,
        transform: Value -> Signal<U, Error>)
        -> Signal<U, Error> {
            return map(transform).flatten(strategy)
    }
}
```

REACTIVECOCOA

transform函数
返回的结果是另一个盒子

```
extension SignalType {

    public func flatMap<U>(strategy: FlattenStrategy,
        transform: Value -> SignalProducer<U, Error>)
        -> Signal<U, Error> {
            return map(transform).flatten(strategy)
    }

    public func flatMap<U>(strategy: FlattenStrategy,
        transform: Value -> Signal<U, Error>)
        -> Signal<U, Error> {
            return map(transform).flatten(strategy)
    }
}
```

transform函数
返回的结果是另一个盒子

PROMISE

```
- (void)setupApi {
    TTRequest *req1 = [TTRequest requestWithUrlString:@"url1"];
    req1.promise.then(^(id res) {
        return [TTRequest requestWithUrlString:[NSString stringWithFormat:@"%@", res]].promise;
    }).then(^(id res1, id res2){
        return [TTRequest requestWithUrlString:[NSString stringWithFormat:@"%@", res1]].promise;
    }).catch(^{
        [TTAlertUtils showSimpleAlertView:@"网络错误"];
    });
}
```

PROMISE

闭包返回的是一个新的promise对象（盒子）

```
- (void)setupApi {
    TTRequest *req1 = [TTRequest requestWithUrlString:@"url1"];
    req1.promise.then(^(id res) {
        return [TTRequest requestWithUrlString:[NSString stringWithFormat:@"%@", res[]].promise;
    }).then(^(id res1, id res2){
        return [TTRequest requestWithUrlString:[NSString stringWithFormat:@"%@", res1]].promise;
    }).catch(^{
        [TTAlertUtils showSimpleAlertView:@"网络错误"];
    });
}
```

闭包接受的参数是盒子里的对象

总结

- Monad 是一种编程范式
- Monad 基于封装后的数据（盒子）
- 数组、Optional、Enum 都是封装后的数据（盒子）的具体表现形式
- Monad 可以支持链式调用

反思

- Monad 到底有多大用?
- Promise 为什么没有在 iOS 开发中流行?
- 学习成本、沟通成本、收益权衡

THANKS