

```
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

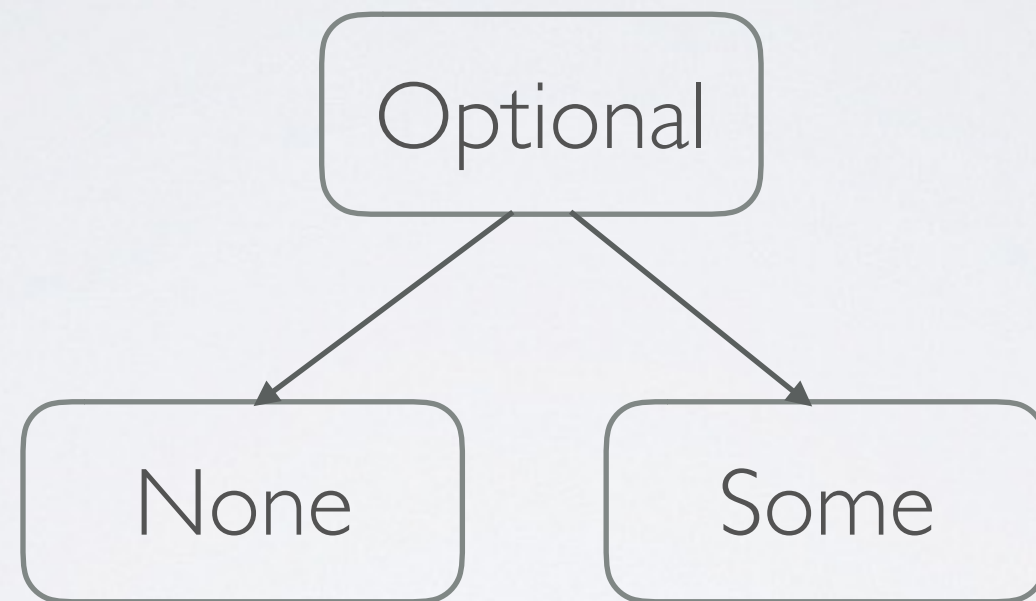
调用闭包f, 得到:  
Optional<Int>.None

```
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
  }
}
```

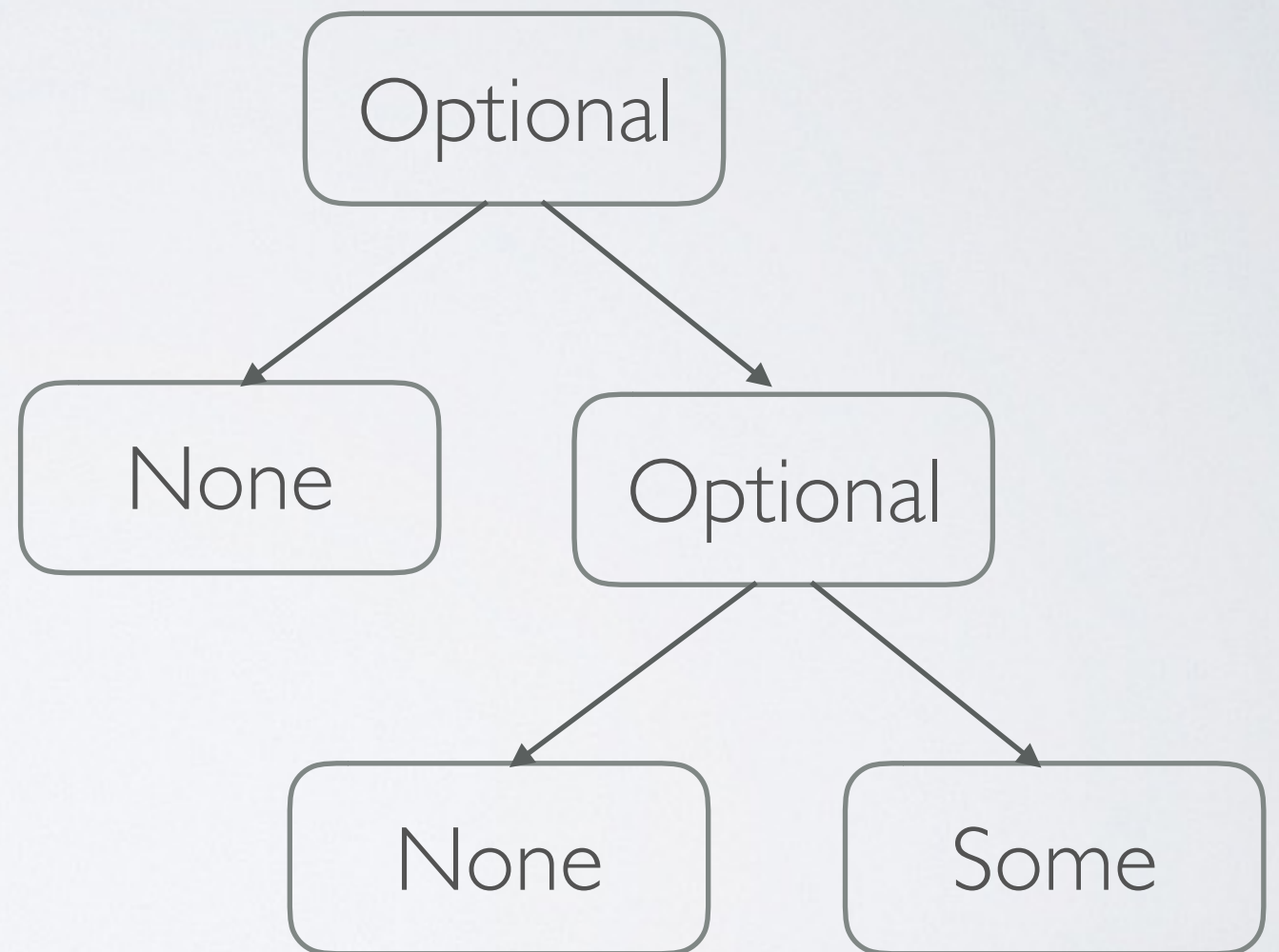
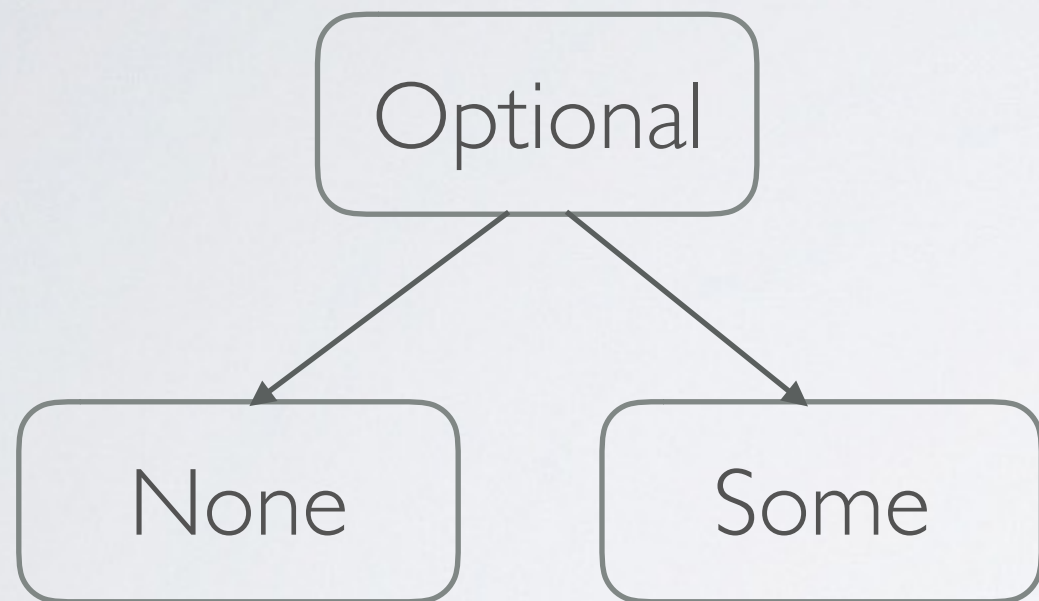
将Optional<Int>.None  
放入 .Some 中

产生多重Optional,  
if let 判断失效

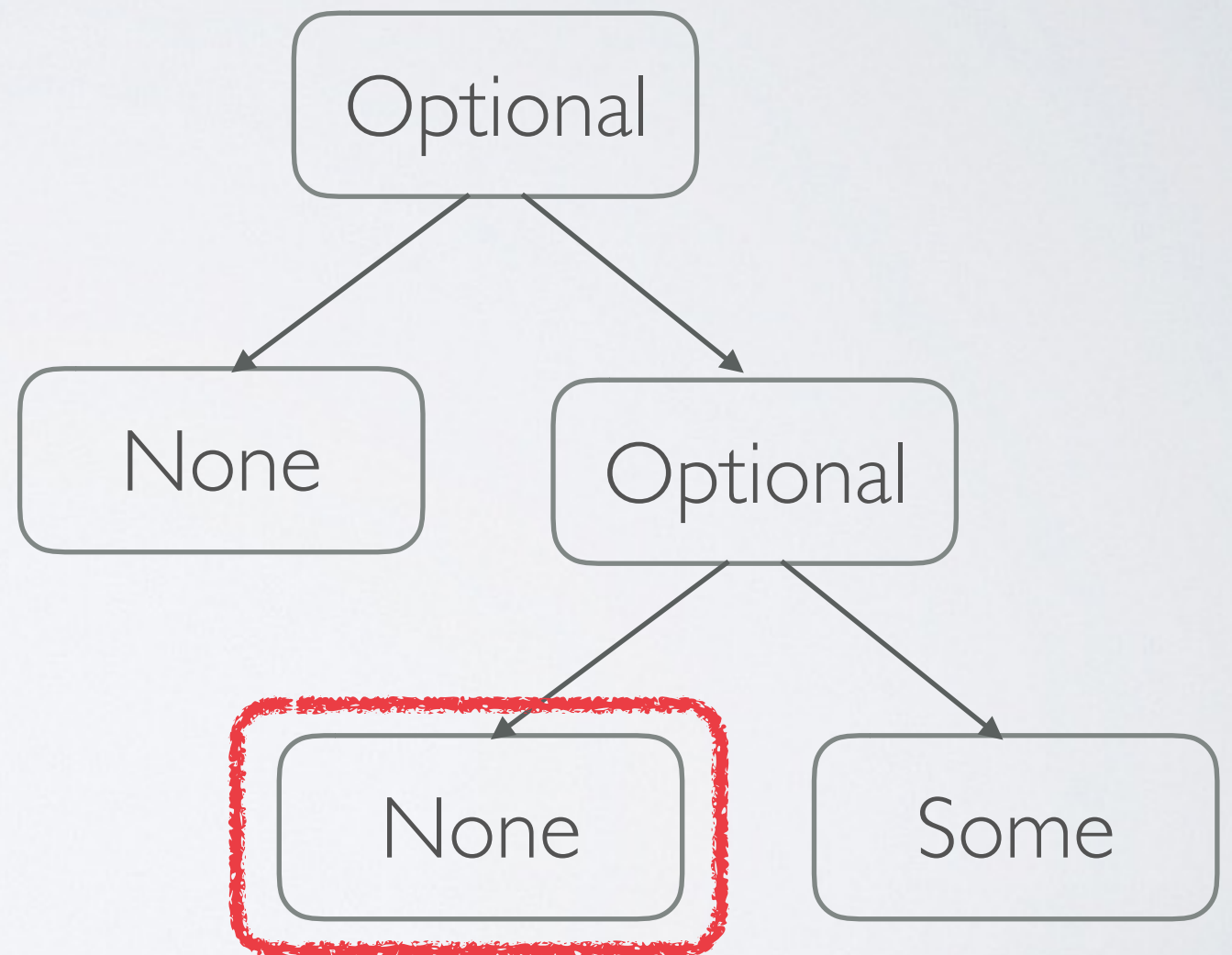
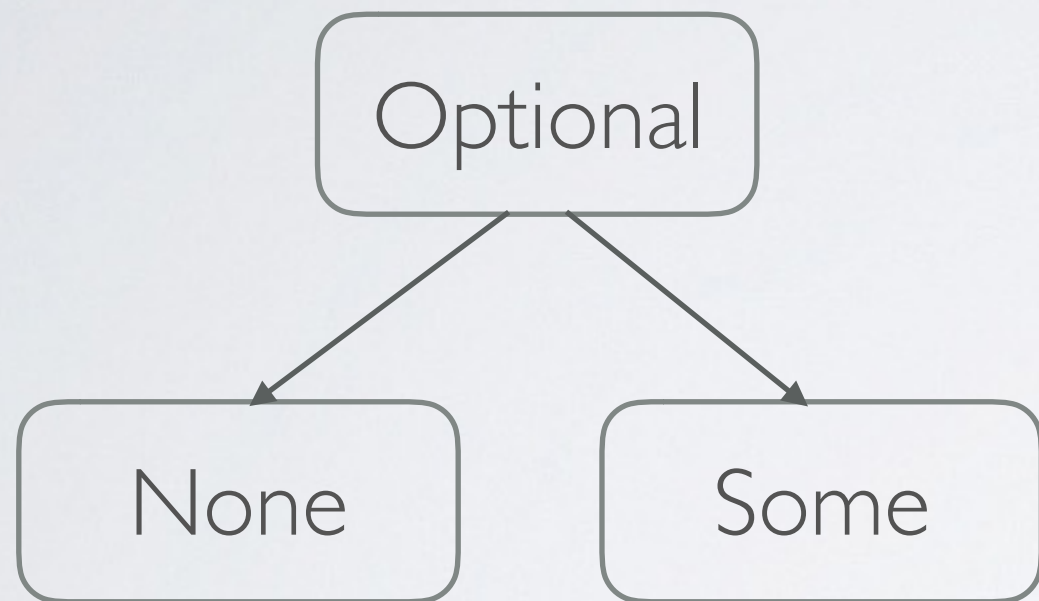
# 多重OPTIONAL



# 多重OPTIONAL



# 多重OPTIONAL





```
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

```
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
  }
}
```

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。



7		
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
    } else {
        return nil
    }
}
if let _ = b {
    print("not nil")
}
```



# 将map改成flatMap

```
let tq: Int? = 1
let b = tq.flatMap { (a: Int) -> Int? in
  if a % 2 == 0 {
    return a
  } else {
    return nil
  }
}
if let _ = b {
  print("not nil")
}
```

Int 和 nil 怎么转换成 Int? 的?

# 看源码

```
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) }
```

```
}
```

# 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

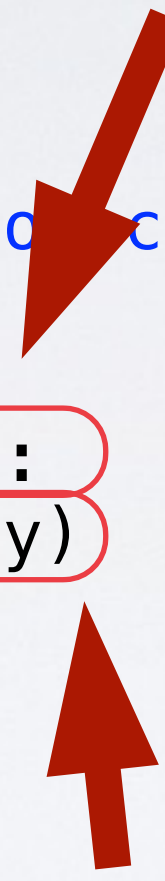


```
public func flatMap<U>(@noescape f: (Wrapped) throws -> U?)
    rethrows -> U? {
    switch self {
    case .Some(let y):
        return try f(y)
    case .None:
        return .None
    }
}
```

# 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
    }
}
```

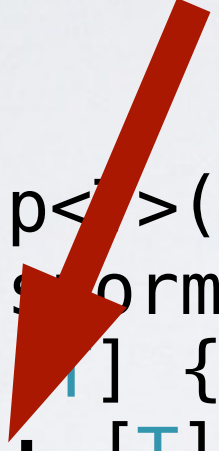
```
public func flatMap<U>(@noescape f: (Wrapped) throws -> U?)
    rethrows -> U? {
    switch self {
    case .Some(let y):
        return try f(y)
    case .None:
        return .None
    }
}
```

# 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  
}
```

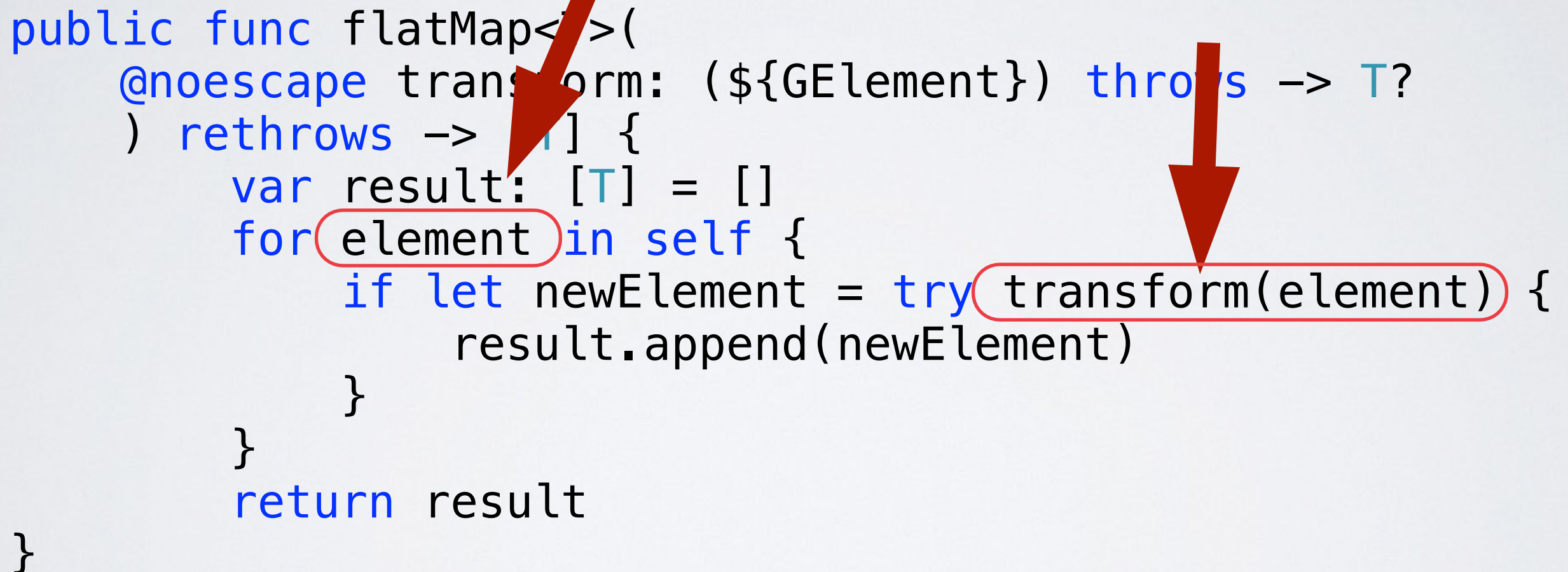
# 数组的FLATMAP

打开盒子

transform函数

返回的结果是另一个盒子

```
public func flatMap<T>(  
    @noescape transform: ({GEElement}) 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  
}
```

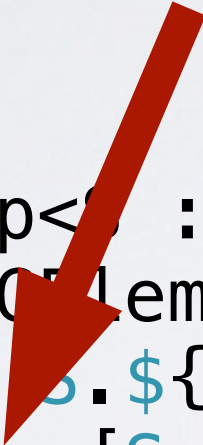
盒子被打开，然后放到另一个盒子中

# 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



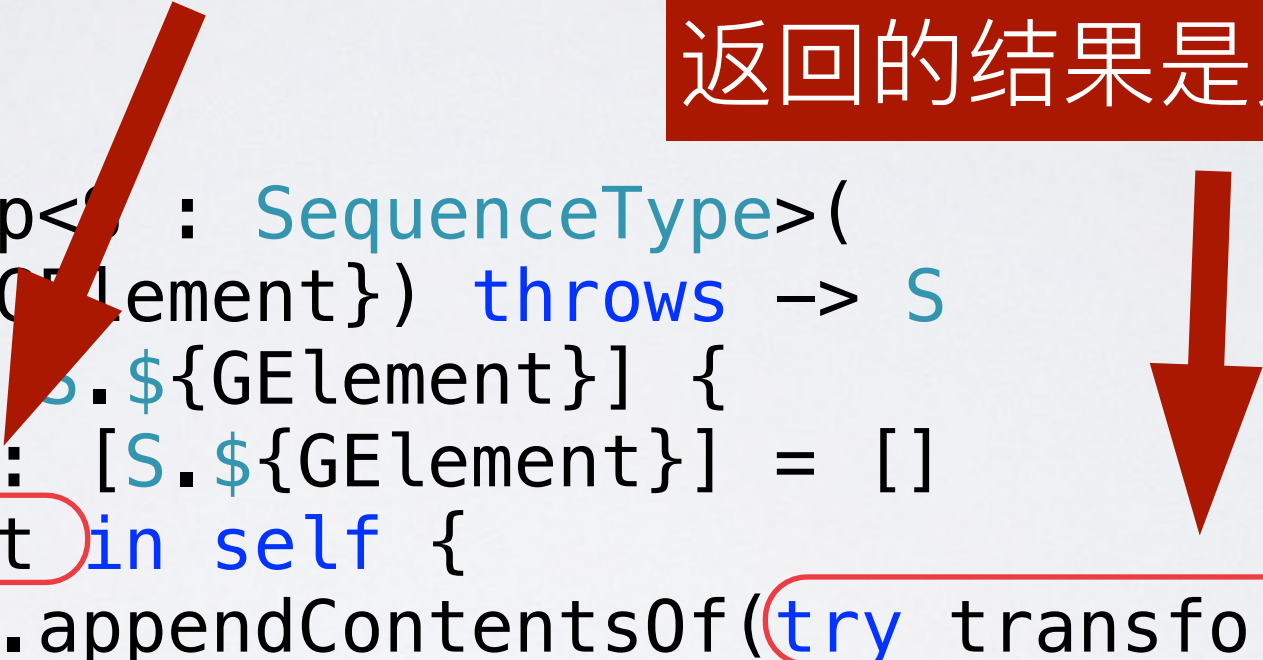
```
public func flatMap<S : SequenceType>(  
    transform: (${OElement}) throws -> S  
) rethrows -> S.${GElement} {  
    var result: [S.${GElement}] = []  
    for element in self {  
        result.appendContentsOf(try transform(element))  
    }  
    return result  
}
```

# 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: (${OElement}) throws -> S  
) rethrows -> S.${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上, 这个闭包: 接受一个「未封装的值」, 返回一个「封装后的值」。
- map: 对自己解包, 然后应用到一个闭包上, 这个闭包: 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 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)  
    }  
}
```



# PROMISE

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

# 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?
- 脱离实际业务需求的技术方案都是耍流氓。

Q & A