

详解 Swift 模式匹配

2015-10-27 👁 3222

作者：Benedikt Terhechte, [原文链接](#), 原文日期：2015-08-20

译者：mmoaay; 校对：numbbbbb; 定稿：千叶知风

在众多 Swift 提供给 Objective-C 程序员使用的新特性中，有个特性把自己伪装成一个无聊的老头，但是却在如何优雅的解决“僵尸金字塔”的问题上有着巨大的潜力。很显然我所说的这个特性就是 `switch` 语句，对于很多 Objective-C 程序员来说，除了用在 [Duff's Device](#) 上比较有趣之外，`switch` 语句非常笨拙，与多个 `if` 语句相比，它几乎没有任何优势。

不过 Swift 中的 `switch` 语句能做的就多了。在接下来的教程里，我会更加详细的讲解这些新特性的各种用途。我会忽略那些与 Objective-C 和 C 中 `switch` 语句相比没有任何优势的 C 写的模式都会导致编译不支持。



行解构，然后根据相应

我们之前已经了解过很多其他语言的模式匹配特性，这些语言包括 Haskell、Erlang、Scala 和 Prolog。这是一个福音，因为它允许我们观察那些语言如何利用模式匹配来解决问题。我们甚至可以通过观察它们的例子来找到最实用的那个。

一个交易引擎

假设华尔街找到你，他们需要一个新的运行在 iOS 设备上的交易平台。因为是交易平台，所以你需要给交易定义一个 `enum`。

第一步

```
enum Trades {  
    case Buy(stock: String, amount: Int, stockPrice: Float)  
    case Sell(stock: String, amount: Int, stockPrice: Float)  
}
```

同时还会提供如下的 API 给你来进行交易处理。注意销售订单的金额是如何变成负数的，而且他们还说股票的价格不重要，他们的引擎会在内部选择一个价格。

```
/**  
 - 参数 stock: 股票的名字  
 - 参数 amount: 金额，负数表示销售额，正数表示购买额  
*/  
func process(stock: String, _ amount: Int) {  
    print ("\(amount) of \(stock)")  
}
```



具备的强大处理能力：

115.5)

的信息。在这个例子中

往往比美好的想象要残

华尔街的人也意识到要处理这些问题你需要新的 API，所以他们给了你下面的两个：

```
func processSlow(stock: String, _ amount: Int, _ fee: Float) { print("slow") }  
func processFast(stock: String, _ amount: Int, _ fee: Float) { print("fast") }
```

交易类型

于是你回到绘图板重新增加了一个 `enum`。交易类型也是每个交易的一部分。

```
enum TraderType {  
    case SingleGuy  
    case Company  
}  
  
enum Trades {  
    case Buy(stock: String, amount: Int, stockPrice: Float, type: TraderType)  
    case Sell(stock: String, amount: Int, stockPrice: Float, type: TraderType)  
}
```

所以，如何更好地实现这一新机制呢？你可以用一个 `if / else` 分支来实现购买和销售，但是这会导致代码嵌套以至于很快代码就变的不清晰了——而且谁知道那些华尔街人会不会给你找新的麻烦。所以你应该把它定义为模式匹配的一个新要求：

```
let aTrade = Trades.Sell(stock: "GOOG", amount: 100, stockPrice: 666.0, type: TraderType)  
  
switch aTrade {
```



我们把
`amount`) 来进行简化，这

的问题（你真应该把项

就算是个人客户也得这

- 交易总额小于 1.000\$ 的购买订单通常处理更慢。

如果使用传统的 `if` 语句，这时代码就应该已经有点凌乱了，而 `switch` 就不会。Swift 为 `switch cases` 提供了保护机制，这种机制可以让你进一步的对可能匹配的 `case` 进行约束。

你只需要对 `switch` 语句稍作修改就可以满足新的变化。

```
let aTrade = Trades.Buy(stock: "GOOG", amount: 1000, stockPrice: 666.0, type: TraderType)

switch aTrade {
case let .Buy(stock, amount, _, TraderType.SingleGuy):
    processSlow(stock, amount, 5.0)
case let .Sell(stock, amount, price, TraderType.SingleGuy)
    where price*Float(amount) > 1000000:
    processFast(stock, -1 * amount, 5.0)
case let .Sell(stock, amount, _, TraderType.SingleGuy):
    processSlow(stock, -1 * amount, 5.0)
case let .Buy(stock, amount, price, TraderType.Company)
    where price*Float(amount) < 1000:
    processSlow(stock, amount, 2.0)
case let .Buy(stock, amount, _, TraderType.Company):
    processFast(stock, amount, 2.0)
case let .Sell(stock, amount, _, TraderType.Company):
    processFast(stock, -1 * amount, 2.0)
}
```



也很好。

但方案还是有点繁琐；我想深入研究一下模式匹配。

那是什么？Swift 将这些模

， `guard` 和 `for`

的。这和
用这个值。有意思的是
它甚至可以匹配可选

值：

```
let p: String? = nil
```

```
switch p {  
case _?: print ("Has String")  
case nil: print ("No String")  
}
```

就像你在交易例子里面看到的一样，它也允许你忽略需要匹配的 `enum` 或者 `tuples` 中无用的数据：

```
switch (15, "example", 3.14) {  
    case (_, _, let pi): print ("pi: \(pi)")  
}
```

2. 标示模式

匹配一个具体的值。这个和 Objective-C 的 `switch` 实现是一样的：

```
switch 5 {  
    case 5: print("5")  
}
```

2.1 标示模式



在 `switch` 中。因为

多，但是我还是在这里

在这里，我们把 3 个值结合放到一个元组中（假想它们是通过调用不同的 API 得到的），然后一口气匹配它们，注意这个模式完成了三件事情：

1. 提取 `age`
2. 确保存在一个 `job`，就算我们不需要它
3. 确保 `payload` 的类型是 `NSDictionary`，尽管我们同样不需要访问它的具体值。

5. 枚举 Case 模式（Enumeration Case Pattern）

就如你在交易例子中所见，模式匹配对 Swift 的 `enum` 支持相当棒。这是因为 `enum cases` 就像密封、不可变且可解构的结构体。这非常像 `tuples`，你可以打开正好匹配上的某个单独 `case` 的内容然后只抽取出你需要的信息²。

假想你正在用函数式的风格写一个游戏，然后你需要定义一些实体。你可以使用 `structs` 但是你的实体的状态很少，你觉得这样有点矫枉过正。

```
enum Entities {  
    case Soldier(x: Int, y: Int)  
    case Tank(x: Int, y: Int)  
    case Player(x: Int, y: Int)  
}
```



的关键词：

它会做类型转换但是不

的话会把类型转换到左

下面是这两种关键词的例子：

```
let a: Any = 5
switch a {
    // 这会失败因为它的类型仍然是 `Any`
    // 错误: binary operator '+' cannot be applied to operands of type 'Any' and 'Int'
    case is Int: print (a + 1)
    // 有效并返回 '6'
    case let n as Int: print (n + 1)
    default: ()
}
```

注意 `is` 前没有 `pattern`。它直接和 `a` 做匹配。

7. 表达模式

表达模式非常强大。它可以把 `switch` 的值和实现了 `~=` 操作符的表达式进行匹配。而且对于这个操作符有默认的实现，比如对于范围匹配，你可以这样做：

```
switch 5 {
    case 0..10: print("In range 0-10")
}
```

然而，更有趣的可能是自己重写操作符，然后使你的自定义类型可以匹配。我们假定你想重写之前写的十斤游戏，而且你无论如何都要使用结构体



~= 操作符。

会出现类型检查错误。

```
bool {
```

```
}
```

一个可能解决上述类似问题的方案是给你的 `struct` 增加一个 `unapply` 方法然后再进行匹配：

```
extension Soldier {
    func unapply() -> (Int, Int, Int) {
        return (self.hp, self.x, self.y)
    }
}

func ~= (p: (Int, Int, Int), t: (Int, Int, Int)) -> Bool {
    return p.0 == t.0 && p.1 == t.1 && p.2 == t.2
}

let soldier = Soldier(hp: 99, x: 10, y: 10)
print(soldier.unapply() ~= (99, 10, 10))
```

但是这相当繁琐而且没有利用好模式匹配背后的大量魔法般的效果。

在这篇博文之前的版本中我写过 `~=` 不适用于协议，但是我错了。我记得我在一个 `Playground` 中试过。而这个例子（由 [reddit](#) 上的 [latrodectus](#) 友情提供）是完全可用的：

```
protocol Entity {
    var value: Int {get}
}
```



多表达模式的细节，看

前，还需要讨论最后一

件事情。

fallthrough, break 和标签

下面的内容和模式匹配没有直接关系，仅仅是和 `switch` 关键词有关，所以我就简单说了。和 C/C++/Objective-C 不一样的是：`switch cases` 不会自动进入下一个 `case`，这也是为什么 Swift 不需要给每个 `case` 都写上 `break`。你可以选择使用 `fallthrough` 关键词来实现传统的自动进入下一个 `case` 的行为。

```
switch 5 {
    case 5:
        print("Is 5")
        fallthrough
    default:
        print("Is a number")
}
// 会在命令行输出: "Is 5" "Is a number"
```

另外，你可以使用 `break` 来提前跳出 `switch` 语句。既然不会默认进入下一个 `case`，为什么还需要这么做呢？比如你知道在一个 `case` 中有一个必须的要求是不满足的，这样你就不能继续执行这个 `case` 了：

```
let userType = "system"
let userID = 10
switch (userType, userID) {
```



再继续调用
但是如果多个这样的情
嵌套代码。

如果你想跳出循环，而不是
定义一个 `labels`，然

我们已经讨论过 `switch` 语句的一些用法。现在，让我们来看一些（多少有点）有趣的真实案例。

真实案例

可选值

对可选值进行解包的方式有很多种，模式匹配就是其中一种。可能到现在这种方法你已经用得非常频繁了，但还是给一个简短的例子吧：

```
var result: String? = secretMethod()
switch result {
case .None:
    println("is nothing")
case let a:
    println("\(a) is a value")
}
```

如果是 Swift 2.0 的话，这会更简单：

```
var result: String? = secretMethod()
switch result {
case nil:
    print("is nothing")
}
```



因为它是 `optional` 或者是一个确定的值。把这个值绑定到一个变量，状态被非常明显的区分

对类型检查。然而，当（[反射一文](#)中），那你 `String` 和 `NSNumber`

`int: 20`), `NSNumber(int: 40`

而， `switch` 语句可

```
for x in u {
    switch x {
```

```
case _ as NSString:
    print("string")
case _ as NSNumber:
    print("number")
default:
    print("Unknown types")
}
}
```

按范围做分级

现在你正在给你当地的高校写分级的 iOS 应用。老师想要输入一个 0 到 100 的数值，然后得到一个相应的等级字符（A-F）。模式匹配现在要来拯救你了：

```
let aGrade = 84

switch aGrade {
case 90...100: print("A")
case 80...90:  print("B")
case 70...80:  print("C")
case 60...70:  print("D")
case 0...60:   print("F")
}
```



频率。我们的目标就是
的不包含其频率的所有

```
print(res)
```

然而，因为 `flatMap` 只能返回非空元素，所以这个解决方案还有很大的改进空间。首先，我们可以放弃使用 `e.1` 而利用元组来做适当的解构（你猜对了）。然后我们只需要调用一次 `flatMap`，这样可以减少先 `filter` 后 `map` 所带来的不必要的性能开销。

```
let res = wordFreqs.flatMap { (e) -> String? in
    switch e {
    case let (s, t) where t > 3: return s
    default: return nil
    }
}
print(res)
```

遍历目录

假想你需要遍历一个文件树然后查找以下内容：

- 所有 customer1 和 customer2 创建的“psd”文件
- 所有 customer2 创建的“blend”文件
- 所有用户创建的“jpeg”文件

```
guard let enumerator = NSFileManager.defaultManager().enumeratorAtPath("/customers/2014,
```



nil)

。同样，模式匹配的代

Fibonacci

同样，来看一下使用模式匹配实现的 fibonacci 算法有多优美³

```
func fibonacci(i: Int) -> Int {
    switch(i) {
    case let n where n <= 0: return 0
    case 0, 1: return 1
    case let n: return fibonacci(n - 1) + fibonacci(n - 2)
    }
}

print(fibonacci(8))
```

当然，如果是大数的话，程序栈会爆掉。

传统的 API 和值提取

通常情况下，当你从外部源取数据的时候，比如一个库，或者一个 API，它不仅是一种很好的做法，而且通常在解析数据之前需要检查数据的一致性。你需要确保所有的 `key` 都是存在的、或者数据的类型都正确、或者数组的长度满足要求。如果不这么做就会因为 bug（有的 `key` 不存在）而导致 app 崩溃（索引不存在的数组项）。而传统的做法通常是嵌套 `if` 语句。



——如管理员或者邮政
设计和增长，API 的使用

`key` 可能是不存在

ame, middlename,

息只包含如下信
。如果没有指定

57,

```
switch (item["type"], item["department"], item["age"], item["name"]) {
    case let (sys as String, dep as String, age as Int, name as [String]) where
        age < 1980 &&
```

```
sys == "system":
    createSystemUser(name.count == 2 ? name.last! : name.first!, dep: dep ?? "Corp")
default:()
}

// 返回 ("voldemort", "Dark Arts")
```

注意这段代码做了一个很危险的假设：就是如果 `name` 数组元素的个数不是 2 个的话，那么它一定包含 4 个元素。如果这种假设不成立，我们获得了包含 0 个元素的数组，这段代码就会崩溃。

除了这一点，模式匹配向你展示了它是在如何在只有一个 `case` 的情况下帮助你编写干净的代码和简化值的提取的。

同样来看看我们是怎么写紧跟在 `case` 之后 `let` 的，这样一来就不必在每一次赋值的时候都重复写它。

模式和其他关键词

Swift 的文档指出不是所有的模式都可以在 `if`、`for` 或者 `guard` 语句中使用。然而，这个文档似乎不是最新的。所有 7 种模式对这三个关键词都有效。

写了一个例子。

子：

```
print(a)
return true
default: return false
```

```
    }  
}  
let u: Any = "a"  
let b: Any = 5  
print(valueTupleType((5, u)))  
print(valueTupleType((5, b)))  
// 5, 5, "if 5", 5, true, false
```

我们可以带着这个想法详细地看一看每一个关键词。

使用 for case

到了 Swift 2.0 后，模式匹配变得更加重要，因为它被扩展到不仅可以支持 `switch`，还可以支持其他的关键词。比如，让我们写一个简单的只返回非空元素的数组函数：

```
func nonnil<T>(array: [T?]) -> [T] {  
    var result: [T] = []  
    for case let x? in array {  
        result.append(x)  
    }  
    return result  
}  
  
print(nonnil(["a", nil, "b", "c", nil]))
```

关键词 `case` 可以被 `for` 循环使用，就像 `switch` 中的 `case` 一样。下面是另外一个例子。还记得我们之前说的游戏么？经过第一次重构之后，现在我们的实体系统看起来是这样的：

```
enum Entity {  
    enum EntityType {  
        case Soldier  
        case Player  
    }  
    case Entry(type: EntityType, x: Int, y: Int, hp: Int)  
}
```

真棒！这可以让我们用更少的代码绘制出所有的项目：

```
for case let Entity.Entry(t, x, y, _) in gameEntities()  
where x > 0 && y > 0 {  
    drawEntity(t, x, y)  
}
```

我们用一行就解析出了所有必需的属性，然后确保我们不会在 0 以下的范围绘制，最后我们调用渲染方法（`drawEntity`）。

为了知道选手是否在游戏中胜出，我们想要知道是否有至少一个士兵的血量是大于 0 的。

```
func gameOver() -> Bool {
```

```
for case Entity.Entry(.Soldier, _, _, let hp) in gameEntities()
where hp > 0 {return false}
return true
}
print(gameOver())
```

好的是 `Soldier` 的匹配是 `for` 查询的一部分。这感觉有点像 `SQL` 而不是命令循环编程。同时，这也可以让编译器更清晰的知道我们的意图，从而就有了打通调度增强这条路的可能性。另外一个很好的体验就是我们不需要完成的拼写出

`Entity.EntityType.Soldier`。就算我们像上面一样只写 `.Soldier`，Swift 也能明白我们的意图。

使用 guard case

另外一个支持模式匹配的关键词就是新引入的 `guard` 关键词。它允许你像 `if let` 一样把 `optionals` 绑定到本地范围，而且不需要任何嵌套：

```
func example(a: String?) {
    guard let a = a else { return }
    print(a)
}
example("yes")
```

`guard let case` 允许你做一些类似模式匹配所介绍的事情。让我们再来看一下士兵的例子。在玩家的血量变满之前，我们需要计算需要增加的血量。士兵不能涨血，所以对于士兵实体而言，我们始终返回 0。

```
let MAX_HP = 100

func healthHP(entity: Entity) -> Int {
    guard case let Entity.Entry(.Player, _, _, hp) = entity
    where hp < MAX_HP
    else { return 0 }
    return MAX_HP - hp
}

print("Soldier", healthHP(Entity.Entry(type: .Soldier, x: 10, y: 10, hp: 79)))
print("Player", healthHP(Entity.Entry(type: .Player, x: 10, y: 10, hp: 57)))

// 输出：
"Soldier 0"
"Player 43"
```

这是把我们目前讨论的各种机制用到极致的一个例子。

- 它非常清晰，没有牵扯到任何嵌套
- 状态的逻辑和初始化是在 `func` 之前处理的，这样可以提高代码的可读性
- 非常简洁

这也是 `switch` 和 `for` 的完美结合，可以把复杂的逻辑结构封装成易读的格式。当然，它不会让逻辑变得更容易理解，但是至少会以更清晰的方式展现给你。特别是使用 `enums` 的时候。

使用 if case

`if case` 的作用和 `guard case` 相反。它是一种非常棒的在分支中打开和匹配数据的方式。结合之前 `guard` 的例子。很显然，我们需要一个 `move` 函数，这个函数允许我们表示一个实体在朝一个方向移动。因为我们的实体是 `enums`，所以我们需要返回一个更新过的实体。

```
func move(entity: Entity, xd: Int, yd: Int) -> Entity {
    if case Entity.Entry(let t, let x, let y, let hp) = entity
        where (x + xd) < 1000 &&
            (y + yd) < 1000 {
        return Entity.Entry(type: t, x: (x + xd), y: (y + yd), hp: hp)
    }
    return entity
}

print(move(Entity.Entry(type: .Soldier, x: 10, y: 10, hp: 79), xd: 30, yd: 500))
// 输出: Entry(main.Entity.EntityType.Soldier, 40, 510, 79)
```

限制

一些限制已经在文章中说过，比如有关 `Expression Patterns` 的问题，看起来它似乎不能匹配 `tuples`（那样的话就真的很方便了）。在 Scala 和 Clojure 中，模式匹配在集合上同样可用，所以你可以匹配它的头部、尾部和部分等。[4](#)。这在 Swift 中是不支持的（尽管 [Austin Zheng](#) 在我之前链接的博客里差不多实现了这一点）

另外一种不可用的情况是（这一点 Scala 同样做得很好）对类或者结构体进行解构。Swift 允许我们定义一个 `unapply` 方法，这个方法做的事情大体和 `init` 相反。实现这个方法，然后就可以让类型检查器对类进行匹配。而在 Swift 中，它看起来就像下面一样：

```
struct Imaginary {
    let x: Int
    let y: Int
    func unapply() -> (Int, Int) {
        // 实现这个方法之后，理论上来说实现了解构变量所需的所有细节
        return (self.x, self.y)
    }
}

// 然后这个就会自动 unapply 然后再进行匹配
guard case let Imaginary(x, y) = anImaginaryObject else { break }
```

更新

08/21/2015 结合 [Reddit](#) 上 foBrowsing 的有用反馈

- 增加 `guard case let`
- 增加简化版的 `let` 语法（如：`let (x, y)` 替代 `(let x, let y)`）

08/22/2015 似乎有一些东西我没测试好。我列举的一些限制实际上是可用的，另外一个 [Reddit](#) 上的评论者（[latrodectus](#)）提出了一些非常有用的指正。

- 将之前的修正为：所有的模式对三个关键词都适用，然后增加了一些要点案例
- 关于协议和表达式模式无效这个限制，其实没有的
- 增加“模式可用性”章节

08/24/2015

- 增加 `if case` 样例，重命名了一些章节。
- 修复了一些文本拼写错误。尤其我不小心写道：`_` 不能匹配 `nil`。那当然是不对的，`_` 可以匹配所有的东西。（感谢 [obecker](#)）

09/18/2015

- 添加了日语翻译的链接

1.可以把它当做 `shell` 里面的 `*` 通配符

2.我不清楚编译器是否在对这点进行了优化，但理论上来说，它应该能计算出所需数据的正确位置，然后忽略 `enum` 的其他情况并内联这个地址

3.当然，不是 [Haskell](#)实现的对手：

```
fib 0 = 0
```

```
fib 1 = 1
```

```
fib n = fib (n-1) + fib (n-2)
```

```
4.比如：switch [1, 2, 4, 3] {  
case [, 2, , 3]:  
}
```

本文由 [SwiftGG](#) 翻译组翻译，已经获得作者翻译授权，最新文章请访问 <http://swift.gg>。

APPVENTURE  Swift 进阶

上一篇

[关于 Swift 的 5 个误区](#)

下一篇

[Swift 中的范围和区间](#)



Issue Page

Error: Comments Not Initialized

Write

Preview

Login with GitHub

Leave a comment

Styling with Markdown is supported

Comment

Powered by [Gitment](#)

分类

APPVENTURE¹³

Andyy Hope⁴

AppCoda³⁹

Big O Note-Taking²

Coding Explorer Blog²

Crunchy Development²⁴

Erica Sadun⁶⁷

IOSCREATOR²⁹

Jacob Banded-Storch²

Jameson

Quave¹

JamesonQuave.com¹⁸

Jesse Squires¹

KHANLOU¹⁶

Mike Ash⁵

Natasha The
Robot⁴⁸

Ole
Begemann³⁰

Open Source
Swift¹¹

Raj Kandathi⁶

Reinder de
Vries¹

Russ Bishop⁷

Soroush
Khanlou²

Swift and
Painless¹¹

Swift 入门¹

Swift 进阶³

Think and
Build²

Thomas
Hanning²¹

Thoughtbot²

Tomasz Szulc⁸

Wooji Juice¹

alisoftware¹

alloc-init⁶

iAchieved.it²²

iOS¹

iOS 开发³

khanlou.com¹

medium.com¹¹

mikeash.com⁸

radex.io³

swiftandpainless¹

uraimo¹⁵

原创文章⁶

投稿⁷

直播资源¹

社区问答¹⁹

标签

Swift 进阶¹⁶⁶

Swift 入门¹²⁸

iOS 开发⁷⁴

Swift⁶⁷

Swift 跨平台

11

Swift 开源信

息 ¹¹

Swift 3 ⁷

WatchOS 2 ⁷

iOS 入门 ⁶

Apple TV 开

发 ⁵

iOS 9 ⁴

Xcode ⁴

IOSCREATOR ⁴

Jesse

Squires ³

Swift 2 ³

社区问答 ³

Swift 进化 ²

SwiftyDB ¹

Objective-C ¹

推送通知 ¹

友情链接

C4iOS 教程

SwiftGG直播

T 沙龙

Code Build

Me

//TODO:

chiba

Perfect

Freeze

小锅的 swift
之路

Prayer 的博客

画渣程序猿
mmoaay

小铁匠的 swift
之路

ppppppmst 的
简书博客

CMB 的博客

BridgeQ

walkingway 的
博客

靛青K

JackAlan

SwiftConChina

Swift 中国

泊学

BearyChat

PHP-Z 论坛

官方文档

又拍云赞助图
床

 RSS 订阅

微信公众号



