**Swift 代码规范指南**

**1. 代码格式**

1.1 使用四个空格进行缩进。

1.2 每行最多160个字符，这样可以避免一行过长。 (Xcode->Preferences->Text Editing->Page guide at column: 设置成160即可)

1.3 确保每个文件结尾都有空白行。

1.4 确保每行都不以空白字符作为结尾 （Xcode->Preferences->Text Editing->Automatically trim trailing whitespace + Including whitespace-only lines).

1.5 左大括号不用另起一行。

class SomeClass {

    func someMethod() {

        if x == y {

            /\* ... \*/

        } else if x == z {

            /\* ... \*/

        } else {

            /\* ... \*/

        }

    }

    /\* ... \*/

}

1.6 当在写一个变量类型，一个字典里的主键，一个函数的参数，遵从一个协议，或一个父类，不用在分号前添加空格。

// 指定类型

let pirateViewController: PirateViewController

// 字典语法(注意这里是向左对齐而不是分号对齐)

let ninjaDictionary: [String: AnyObject] = [

    "fightLikeDairyFarmer": false,

    "disgusting": true

]

// 声明函数

func myFunction<t, u: someprotocol where t.relatedtype == u>(firstArgument: U, secondArgument: T) {

    /\* ... \*/

}

// 调用函数

someFunction(someArgument: "Kitten")

// 父类

class PirateViewController: UIViewController {

    /\* ... \*/

}

// 协议

extension PirateViewController: UITableViewDataSource {

    /\* ... \*/

}</t, u: someprotocol where t.relatedtype == u>

1.7 基本来说，要在逗号后面加空格。

let myArray = [1, 2, 3, 4, 5]

1.8 二元运算符(+, ==, 或->)的前后都需要添加空格，左小括号后面和右小括号前面不需要空格。

let myValue = 20 + (30 / 2) \* 3

if 1 + 1 == 3 {

    fatalError("The universe is broken.")

}

func pancake() -> Pancake {

    /\* ... \*/

}

1.9  遵守Xcode内置的缩进格式( 如果已经遵守，按下CTRL-i 组合键文件格式没有变化)。当声明的一个函数需要跨多行时，推荐使用Xcode默认的格式，目前Xcode 版本是 7.3。

// Xcode针对跨多行函数声明缩进

func myFunctionWithManyParameters(parameterOne: String,

                                  parameterTwo: String,

                                  parameterThree: String) {

    // Xcode会自动缩进

    print("\(parameterOne) \(parameterTwo) \(parameterThree)")

}

// Xcode针对多行 if 语句的缩进

if myFirstVariable > (mySecondVariable + myThirdVariable)

    && myFourthVariable == .SomeEnumValue {

    // Xcode会自动缩进

    print("Hello, World!")

}

1.10 当调用的函数有多个参数时，每一个参数另起一行，并比函数名多一个缩进。

someFunctionWithManyArguments(

    firstArgument: "Hello, I am a string",

    secondArgument: resultFromSomeFunction()

    thirdArgument: someOtherLocalVariable)

1.11 当遇到需要处理的数组或字典内容较多需要多行显示时，需把 [ 和 ] 类似于方法体里的括号， 方法体里的闭包也要做类似处理。

someFunctionWithABunchOfArguments(

    someStringArgument: "hello I am a string",

    someArrayArgument: [

        "dadada daaaa daaaa dadada daaaa daaaa dadada daaaa daaaa",

        "string one is crazy - what is it thinking?"

    ],

    someDictionaryArgument: [

        "dictionary key 1": "some value 1, but also some more text here",

        "dictionary key 2": "some value 2"

    ],

    someClosure: { parameter1 in

        print(parameter1)

    })

1.12 应尽量避免出现多行断言，可使用本地变量或其他策略。

// 推荐

let firstCondition = x == firstReallyReallyLongPredicateFunction()

let secondCondition = y == secondReallyReallyLongPredicateFunction()

let thirdCondition = z == thirdReallyReallyLongPredicateFunction()

if firstCondition && secondCondition && thirdCondition {

    // 你要干什么

}

// 不推荐

if x == firstReallyReallyLongPredicateFunction()

    && y == secondReallyReallyLongPredicateFunction()

    && z == thirdReallyReallyLongPredicateFunction() {

    // 你要干什么

}

**2. 命名**

2.1 在Swift中不用如Objective-C式 一样添加前缀 (如使用 GuybrushThreepwoode 而不是 LIGuybrushThreepwood)。

2.2 使用帕斯卡拼写法（又名大骆驼拼写法，首字母大写）为类型命名 (如 struct, enum, class, typedef, associatedtype 等)。

2.3 使用小骆驼拼写法 (首字母小写) 为函数，方法，变量，常量，参数等命名。

2.4 首字母缩略词在命名中一般来说都是全部大写，例外的情形是如果首字母缩略词是一个命名的开始部分，而这个命名需要小写字母作为开头，这种情形下首字母缩略词全部小写。

// "HTML" 是变量名的开头, 需要全部小写 "html"

let htmlBodyContent: String = "

Hello, World!

"

// 推荐使用 ID 而不是 Id

let profileID: Int = 1

// 推荐使用 URLFinder 而不是 UrlFinder

class URLFinder {

    /\* ... \*/

}

2.5 使用前缀 k + 大骆驼命名法 为所有非单例的静态常量命名。

class MyClassName {

    // 基元常量使用 k 作为前缀

    static let kSomeConstantHeight: CGFloat = 80.0

    // 非基元常量也是用 k 作为前缀

    static let kDeleteButtonColor = UIColor.redColor()

    // 对于单例不要使用k作为前缀

    static let sharedInstance = MyClassName()

    /\* ... \*/

}

2.6 对于泛型和关联类型，可以使用单个大写字母，也可是遵从大骆驼命名方式并能描述泛型的单词。如果这个单词和要实现的协议或继承的父类有冲突，可以为相关类型或泛型名字添加 Type 作为后缀。

class SomeClass { /\* ... \*/ }

class SomeClass { /\* ... \*/ }

protocol Modelable {

    associatedtype Model

}

protocol Sequence {

    associatedtype IteratorType: Iterator

}

2.7 命名应该具有描述性 和 清晰的。

// 推荐

class RoundAnimatingButton: UIButton { /\* ... \*/ }

// 不推荐

class CustomButton: UIButton { /\* ... \*/ }

2.8 不要缩写，简写命名，或用单个字母命名。

// 推荐

class RoundAnimatingButton: UIButton {

    let animationDuration: NSTimeInterval

    func startAnimating() {

        let firstSubview = subviews.first

    }

}

// 不推荐

class RoundAnimating: UIButton {

    let aniDur: NSTimeInterval

    func srtAnmating() {

        let v = subviews.first

    }

}

2.9 如果原有命名不能明显表明类型，则属性命名内要包括类型信息。

// 推荐

class ConnectionTableViewCell: UITableViewCell {

    let personImageView: UIImageView

    let animationDuration: NSTimeInterval

    // 作为属性名的firstName，很明显是字符串类型，所以不用在命名里不用包含String

    let firstName: String

    // 虽然不推荐, 这里用 Controller 代替 ViewController 也可以。

    let popupController: UIViewController

    let popupViewController: UIViewController

    // 如果需要使用UIViewController的子类，如TableViewController, CollectionViewController, SplitViewController, 等，需要在命名里标名类型。

    let popupTableViewController: UITableViewController

    // 当使用outlets时, 确保命名中标注类型。

    @IBOutlet weak var submitButton: UIButton!

    @IBOutlet weak var emailTextField: UITextField!

    @IBOutlet weak var nameLabel: UILabel!

}

// 不推荐

class ConnectionTableViewCell: UITableViewCell {

    // 这个不是 UIImage, 不应该以Image 为结尾命名。

    // 建议使用 personImageView

    let personImage: UIImageView

    // 这个不是String，应该命名为 textLabel

    let text: UILabel

    // animation 不能清晰表达出时间间隔

    // 建议使用 animationDuration 或 animationTimeInterval

    let animation: NSTimeInterval

    // transition 不能清晰表达出是String

    // 建议使用 transitionText 或 transitionString

    let transition: String

    // 这个是ViewController，不是View

    let popupView: UIViewController

    // 由于不建议使用缩写，这里建议使用 ViewController替换 VC

    let popupVC: UIViewController

    // 技术上讲这个变量是 UIViewController, 但应该表达出这个变量是TableViewController

    let popupViewController: UITableViewController

    // 为了保持一致性，建议把类型放到变量的结尾，而不是开始，如submitButton

    @IBOutlet weak var btnSubmit: UIButton!

    @IBOutlet weak var buttonSubmit: UIButton!

    // 在使用outlets 时，变量名内应包含类型名。

    // 这里建议使用 firstNameLabel

    @IBOutlet weak var firstName: UILabel!

}

2.10 当给函数参数命名时，要确保函数能理解每个参数的目的。

2.11 根据苹果接口设计指导文档, 如果协议描述的是协议做的事应该命名为名词(如Collection) ，如果描述的是行为，需添加后缀 able 或 ing (如Equatable 和 ProgressReporting)。 如果上述两者都不能满足需求，可以添加Protocol作为后缀，例子见下面。

// 这个协议描述的是协议能做的事，应该命名为名词。

protocol TableViewSectionProvider {

    func rowHeight(atRow row: Int) -> CGFloat

    var numberOfRows: Int { get }

    /\* ... \*/

}

// 这个协议表达的是行为, 以able最为后缀

protocol Loggable {

    func logCurrentState()

    /\* ... \*/

}

// 因为已经定义类InputTextView，如果依然需要定义相关协议，可以添加Protocol作为后缀。

protocol InputTextViewProtocol {

    func sendTrackingEvent()

    func inputText() -> String

    /\* ... \*/

}

**3. 代码风格**

3.1 综合

3.1.1 尽可能的多使用let，少使用var。

3.1.2 当需要遍历一个集合并变形成另一个集合时，推荐使用函数 map, filter 和 reduce。

// 推荐

let stringOfInts = [1, 2, 3].flatMap { String($0) }

// ["1", "2", "3"]

// 不推荐

var stringOfInts: [String] = []

for integer in [1, 2, 3] {

    stringOfInts.append(String(integer))

}

// 推荐

let evenNumbers = [4, 8, 15, 16, 23, 42].filter { $0 % 2 == 0 }

// [4, 8, 16, 42]

// 不推荐

var evenNumbers: [Int] = []

for integer in [4, 8, 15, 16, 23, 42] {

    if integer % 2 == 0 {

        evenNumbers(integer)

    }

}

3.1.3 如果变量类型可以依靠推断得出，不建议声明变量时指明类型。

3.1.4 如果一个函数有多个返回值，推荐使用 元组 而不是 inout 参数， 如果你见到一个元组多次，建议使用typealias ，而如果返回的元组有三个或多于三个以上的元素，建议使用结构体或类。

func pirateName() -> (firstName: String, lastName: String) {

    return ("Guybrush", "Threepwood")

}

let name = pirateName()

let firstName = name.firstName

let lastName = name.lastName

3.1.5 当使用委托和协议时，请注意避免出现循环引用，基本上是在定义属性的时候使用 weak 修饰。

3.1.6 在闭包里使用 self 的时候要注意出现循环引用，使用捕获列表可以避免这一点。

myFunctionWithClosure() { [weak self] (error) -> Void in

    // 方案 1

    self?.doSomething()

    // 或方案 2

    guard let strongSelf = self else {

        return

    }

    strongSelf.doSomething()

}

3.1.7 Switch 模块中不用显式使用break。

3.1.8 断言流程控制的时候不要使用小括号。

// 推荐

if x == y {

    /\* ... \*/

}

// 不推荐

if (x == y) {

    /\* ... \*/

}

3.1.9 在写枚举类型的时候，尽量简写。

// 推荐

imageView.setImageWithURL(url, type: .person)

// 不推荐

imageView.setImageWithURL(url, type: AsyncImageView.Type.person)

3.1.10 在使用类方法的时候不用简写，因为类方法不如 枚举 类型一样，可以根据轻易地推导出上下文。

// 推荐

imageView.backgroundColor = UIColor.whiteColor()

// 不推荐

imageView.backgroundColor = .whiteColor()

3.1.11 不建议使用用self.修饰除非需要。

3.1.12 在新写一个方法的时候，需要衡量这个方法是否将来会被重写，如果不是，请用 final 关键词修饰，这样阻止方法被重写。一般来说，final 方法可以优化编译速度，在合适的时候可以大胆使用它。但需要注意的是，在一个公开发布的代码库中使用 final 和本地项目中使用 final 的影响差别很大的。

3.1.13 在使用一些语句如 else，catch等紧随代码块的关键词的时候，确保代码块和关键词在同一行。下面 if/else 和 do/catch 的例子.

if someBoolean {

    // 你想要什么

} else {

    // 你不想做什么

}

do {

    let fileContents = try readFile("filename.txt")

} catch {

    print(error)

}

3.2 访问控制修饰符

3.2.1 如果需要把访问修饰符放到第一个位置。

// 推荐

private static let kMyPrivateNumber: Int

// 不推荐

static private let kMyPrivateNumber: Int

3.2.2 访问修饰符不应单独另起一行，应和访问修饰符描述的对象保持在同一行。

// 推荐

public class Pirate {

    /\* ... \*/

}

// 不推荐

public

class Pirate {

    /\* ... \*/

}

3.2.3  默认的访问控制修饰符是 internal, 如果需要使用internal 可以省略不写。

3.2.4 当一个变量需要被单元测试 访问时，需要声明为 internal 类型来使用@testable import {ModuleName}。 如果一个变量实际上是private 类型，而因为单元测试需要被声明为 internal 类型，确定添加合适的注释文档来解释为什么这么做。这里添加注释推荐使用 - warning: 标记语法。

/\*\*

 这个变量是private 名字

 - warning: 定义为 internal 而不是 private 为了 `@testable`.

 \*/

let pirateName = "LeChuck"

3.3 自定义操作符

不推荐使用自定义操作符，如果需要创建函数来替代。

在重写操作符之前，请慎重考虑是否有充分的理由一定要在全局范围内创建新的操作符，而不是使用其他策略。

你可以重载现有的操作符来支持新的类型(特别是 ==)，但是新定义的必须保留操作符的原来含义，比如 == 必须用来测试是否相等并返回布尔值。

3.4 Switch 语句 和 枚举

3.4.1 在使用 Switch 语句时，如果选项是有限集合时，不要使用default，相反地，把一些不用的选项放到底部，并用 break 关键词 阻止其执行。

3.4.2 因为Swift 中的 switch 选项默认是包含break的， 如果不需要不用使用 break 关键词。

3.4.3 case 语句 应和 switch 语句左对齐，并在 标准的 default 上面。

3.4.4 当定义的选项有关联值时，确保关联值有恰当的名称，而不只是类型。(如. 使用 case Hunger(hungerLevel: Int) 而不是 case Hunger(Int)).

enum Problem {

    case attitude

    case hair

    case hunger(hungerLevel: Int)

}

func handleProblem(problem: Problem) {

    switch problem {

    case .attitude:

        print("At least I don't have a hair problem.")

    case .hair:

        print("Your barber didn't know when to stop.")

    case .hunger(let hungerLevel):

        print("The hunger level is \(hungerLevel).")

    }

}

3.4.5 推荐尽可能使用fall through。

3.4.6 如果default 的选项不应该触发，可以抛出错误 或 断言类似的做法。

func handleDigit(digit: Int) throws {

    case 0, 1, 2, 3, 4, 5, 6, 7, 8, 9:

        print("Yes, \(digit) is a digit!")

    default:

        throw Error(message: "The given number was not a digit.")

}

3.5 可选类型

3.5.1 唯一使用隐式拆包可选型（implicitly unwrapped optionals）的场景是结合@IBOutlets，在其他场景使用 非可选类型 和 常规可选类型，即使有的场景你确定有的变量使用的时候永远不会为 nil， 但这样做可以保持一致性和程序更加健壮。

3.5.2 不要使用 as! 或 try!。

3.5.3 如果对于一个变量你不打算声明为可选类型，但当需要检查变量值是否为 nil，推荐用当前值和 nil 直接比较，而不推荐使用 if let 语法。

// 推荐

if someOptional != nil {

    // 你要做什么

}

// 不推荐

if let \_ = someOptional {

    // 你要做什么

}

3.5.4 不要使用 unowned，unowned 和 weak 变量基本上等价，并且都是隐式拆包( unowned 在引用计数上有少许性能优化)，由于不推荐使用隐式拆包，也不推荐使用unowned 变量。

// 推荐

weak var parentViewController: UIViewController?

// 不推荐

weak var parentViewController: UIViewController!

unowned var parentViewController: UIViewController

3.5.5 当拆包取值时，使用和被拆包取值变量相同的名称。

guard let myVariable = myVariable else {

    return

}

3.6 协议

在实现协议的时候，有两种方式来组织你的代码:

使用 // MARK: 注释来分割协议实现和其他代码。

使用 extension 在 类/结构体已有代码外，但在同一个文件内。

请注意 extension 内的代码不能被子类重写，这也意味着测试很难进行。 如果这是经常发生的情况，为了代码一致性最好统一使用第一种办法。否则使用第二种办法，其可以代码分割更清晰。

使用而第二种方法的时候，使用  // MARK:  依然可以让代码在 Xcode 可读性更强。

3.7 属性

3.7.1 对于只读属性，计算后(Computed)属性, 提供 getter 而不是 get {}。

var computedProperty: String {

    if someBool {

        return "I'm a mighty pirate!"

    }

    return "I'm selling these fine leather jackets."

}

3.7.2 对于属性相关方法 get {}, set {}, willSet, 和 didSet, 确保缩进相关代码块。

3.7.3 对于willSet/didSet 和 set 中的旧值和新值虽然可以自定义名称，但推荐使用默认标准名称 newValue/oldValue。

var computedProperty: String {

    get {

        if someBool {

            return "I'm a mighty pirate!"

        }

        return "I'm selling these fine leather jackets."

    }

    set {

        computedProperty = newValue

    }

    willSet {

        print("will set to \(newValue)")

    }

    didSet {

        print("did set from \(oldValue) to \(newValue)")

    }

}

3.7.4 在创建类常量的时候，使用 static 关键词修饰。

class MyTableViewCell: UITableViewCell {

    static let kReuseIdentifier = String(MyTableViewCell)

    static let kCellHeight: CGFloat = 80.0

}

3.7.5 声明单例属性可以通过下面方式进行：

class PirateManager {

    static let sharedInstance = PirateManager()

    /\* ... \*/

}

3.8 闭包

3.8.1 如果参数的类型很明显，可以在函数名里可以省略参数类型, 但明确声明类型也是允许的。 代码的可读性有时候是添加详细的信息，而有时候部分重复，根据你的判断力做出选择吧，但前后要保持一致性。

// 省略类型

doSomethingWithClosure() { response in

    print(response)

}

// 明确指出类型

doSomethingWithClosure() { response: NSURLResponse in

    print(response)

}

// map 语句使用简写

[1, 2, 3].flatMap { String($0) }

3.8.2 如果使用捕捉列表 或 有具体的非 Void返回类型，参数列表应该在小括号内， 否则小括号可以省略。

// 因为使用捕捉列表，小括号不能省略。

doSomethingWithClosure() { [weak self] (response: NSURLResponse) in

    self?.handleResponse(response)

}

// 因为返回类型，小括号不能省略。

doSomethingWithClosure() { (response: NSURLResponse) -> String in

    return String(response)

}

3.8.3 如果闭包是变量类型，不需把变量值放在括号中，除非需要，如变量类型是可选类型(Optional?)， 或当前闭包在另一个闭包内。确保闭包里的所以参数放在小括号中，这样()表示没有参数，Void 表示不需要返回值。

let completionBlock: (success: Bool) -> Void = {

    print("Success? \(success)")

}

let completionBlock: () -> Void = {

    print("Completed!")

}

let completionBlock: (() -> Void)? = nil

3.9 数组

3.9.1 基本上不要通过下标直接访问数组内容，如果可能使用如 .first 或 .last, 因为这些方法是非强制类型并不会崩溃。 推荐尽可能使用 for item in items 而不是 for i in 0..<="" p="">

3.9.2 不要使用 += 或 + 操作符给数组添加新元素，使用性能较好的.append() 或.appendContentsOf()  ，如果需要声明数组基于其他的数组并保持不可变类型， 使用 let myNewArray = [arr1, arr2].flatten()，而不是let myNewArray = arr1 + arr2 。

3.10 错误处理

假设一个函数 myFunction 返回类型声明为 String，但是总有可能函数会遇到error，有一种解决方案是返回类型声明为 String?, 当遇到错误的时候返回 nil。

例子:

func readFile(withFilename filename: String) -> String? {

    guard let file = openFile(filename) else {

        return nil

    }

    let fileContents = file.read()

    file.close()

    return fileContents

}

func printSomeFile() {

    let filename = "somefile.txt"

    guard let fileContents = readFile(filename) else {

        print("不能打开 \(filename).")

        return

    }

    print(fileContents)

}

实际上如果预知失败的原因，我们应该使用Swift 中的 try/catch 。

定义 错误对象 结构体如下:

struct Error: ErrorType {

    public let file: StaticString

    public let function: StaticString

    public let line: UInt

    public let message: String

    public init(message: String, file: StaticString = #file, function: StaticString = #function, line: UInt = #line) {

        self.file = file

        self.function = function

        self.line = line

        self.message = message

    }

}

使用案例:

func readFile(withFilename filename: String) throws -> String {

    guard let file = openFile(filename) else {

        throw Error(message: “打不开的文件名称 \(filename).")

    }

    let fileContents = file.read()

    file.close()

    return fileContents

}

func printSomeFile() {

    do {

        let fileContents = try readFile(filename)

        print(fileContents)

    } catch {

        print(error)

    }

}

其实项目中还是有一些场景更适合声明为可选类型，而不是错误捕捉和处理，比如在获取远端数据过程中遇到错误，nil作为返回结果是合理的，也就是声明返回可选类型比错误处理更合理。

整体上说，如果一个方法有可能失败，并且使用可选类型作为返回类型会导致错误原因湮没，不妨考虑抛出错误而不是吃掉它。

3.11 使用 guard 语句

3.11.1 总体上，我们推荐使用提前返回的策略，而不是 if 语句的嵌套。使用 guard 语句可以改善代码的可读性。

// 推荐

func eatDoughnut(atIndex index: Int) {

    guard index >= 0 && index < doughnuts else {

        // 如果 index 超出允许范围，提前返回。

        return

    }

    let doughnut = doughnuts[index]

    eat(doughnut)

}

// 不推荐

func eatDoughnuts(atIndex index: Int) {

    if index >= 0 && index < donuts.count {

        let doughnut = doughnuts[index]

        eat(doughnut)

    }

}

3.11.2 在解析可选类型时，推荐使用 guard 语句，而不是 if 语句，因为 guard 语句可以减少不必要的嵌套缩进。

// 推荐

guard let monkeyIsland = monkeyIsland else {

    return

}

bookVacation(onIsland: monkeyIsland)

bragAboutVacation(onIsland: monkeyIsland)

// 不推荐

if let monkeyIsland = monkeyIsland {

    bookVacation(onIsland: monkeyIsland)

    bragAboutVacation(onIsland: monkeyIsland)

}

// 禁止

if monkeyIsland == nil {

    return

}

bookVacation(onIsland: monkeyIsland!)

bragAboutVacation(onIsland: monkeyIsland!)

3.11.3 当解析可选类型需要决定在 if 语句 和 guard 语句之间做选择时，最重要的判断标准是是否让代码可读性更强，实际项目中会面临更多的情景，如依赖 2 个不同的布尔值，复杂的逻辑语句会涉及多次比较等，大体上说，根据你的判断力让代码保持一致性和更强可读性， 如果你不确定 if 语句 和 guard 语句哪一个可读性更强，建议使用 guard 。

// if 语句更有可读性

if operationFailed {

    return

}

// guard 语句这里有更好的可读性

guard isSuccessful else {

    return

}

// 双重否定不易被理解 - 不要这么做

guard !operationFailed else {

    return

}

3.11.4  如果需要在2个状态间做出选择，建议使用if 语句，而不是使用 guard 语句。

// 推荐

if isFriendly {

    print("你好, 远路来的朋友！")

} else {

    print(“穷小子，哪儿来的？")

}

// 不推荐

guard isFriendly else {

    print("穷小子，哪儿来的？")

    return

}

print("你好, 远路来的朋友！")

3.11.5  你只应该在在失败情形下退出当前上下文的场景下使用 guard 语句，下面的例子可以解释 if 语句有时候比 guard 语句更合适 – 我们有两个不相关的条件，不应该相互阻塞。

if let monkeyIsland = monkeyIsland {

    bookVacation(onIsland: monkeyIsland)

}

if let woodchuck = woodchuck where canChuckWood(woodchuck) {

    woodchuck.chuckWood()

}

3.11.6 我们会经常遇到使用 guard 语句拆包多个可选值，如果所有拆包失败的错误处理都一致可以把拆包组合到一起 (如 return, break, continue,throw 等).

// 组合在一起因为可能立即返回

guard let thingOne = thingOne,

    let thingTwo = thingTwo,

    let thingThree = thingThree else {

    return

}

// 使用独立的语句 因为每个场景返回不同的错误

guard let thingOne = thingOne else {

    throw Error(message: "Unwrapping thingOne failed.")

}

guard let thingTwo = thingTwo else {

    throw Error(message: "Unwrapping thingTwo failed.")

}

guard let thingThree = thingThree else {

    throw Error(message: "Unwrapping thingThree failed.")

}

**4. 文档/注释**

4.1 文档

如果一个函数比 O(1) 复杂度高，你需要考虑为函数添加注释，因为函数签名(方法名和参数列表) 并不是那么的一目了然，这里推荐比较流行的插件 VVDocumenter. 不论出于何种原因，如果有任何奇淫巧计不易理解的代码，都需要添加注释，对于复杂的 类/结构体/枚举/协议/属性 都需要添加注释。所有公开的 函数/类/变量/枚举/协议/属性/常数 也都需要添加文档，特别是 函数声明(包括名称和参数列表) 不是那么清晰的时候。

写文档时，确保参照苹果文档中提及的标记语法合集。

在注释文档完成后，你应检查格式是否正确。

规则:

4.1.1 一行不要超过160个字符 (和代码长度限制雷同).

4.1.2 即使文档注释只有一行，也要使用模块化格式 (/\*\* \*/).

4.1.3 注释模块中的空行不要使用 \* 来占位。

4.1.4 确定使用新的 – parameter 格式，而不是就得 Use the new -:param: 格式，另外注意 parameter 是小写的。

4.1.5 如果需要给一个方法的 参数/返回值/抛出异常 添加注释，务必给所有的添加注释，即使会看起来有部分重复，否则注释会看起来不完整，有时候如果只有一个参数值得添加注释，可以在方法注释里重点描述。

4.1.6 对于负责的类，在描述类的使用方法时可以添加一些合适的例子，请注意Swift注释是支持 MarkDown 语法的。

/\*\*

 ## 功能列表

 这个类提供下一下很赞的功能，如下:

 - 功能 1

 - 功能 2

 - 功能 3

 ## 例子

 这是一个代码块使用四个空格作为缩进的例子。

     let myAwesomeThing = MyAwesomeClass()

     myAwesomeThing.makeMoney()

 ## 警告

 使用的时候总注意以下几点

 1. 第一点

 2. 第二点

 3. 第三点

 \*/

class MyAwesomeClass {

    /\* ... \*/

}

4.1.8 在写文档注释时，尽量保持简洁。

4.2 其他注释原则

4.2.1  // 后面要保留空格。

4.2.2 注释必须要另起一行。

4.2.3 使用注释 // MARK: - xoxo 时, 下面一行保留为空行。

class Pirate {

    // MARK: - 实例属性

    private let pirateName: String

    // MARK: - 初始化

    init() {

        /\* ... \*/

    }

}

英文原文：[Swift Style Guide](https://github.com/linkedin/swift-style-guide#1-code-formatting)