关于原则：最重要的目标：每个元素都能够准确清晰的表达出它的含义。做出 API 设计、声明后要检查在上下文中是否足够清晰明白。

一、命名

#协议

（1）协议描述的是“做的事情”，命名为名词

```
protocol TableViewSectionProvider {
    func rowHeight(at row: Int) -> CGFloat
    var numberOfRows: Int { get }
    /* ... */
}
```

（2）协议描述的是“能力”，需添加后缀able或 ing 。（如 Equatable、 ProgressReporting）

```
protocol Loggable {
    func logCurrentState()
    /* ... */
}
```

```
protocol Equatable {
    func ==(lhs: Self, rhs: Self) -> bool {
        /* ... */
    }
}
```

（3）如果已经定义类，需要给类定义相关协议，则添加Protocol后缀

```
protocol InputTextViewProtocol {
    func sendTrackingEvent()
    func inputText() -> String
    /* ... */
}
```

#实现protocol：

如果确定protocol的实现不会被重写，建议用extension将protocol实现分离

推荐：

```
class MyViewController: UIViewController {
  // class stuff here
}

// MARK: - UITableViewDataSource

extension MyViewController: UITableViewDataSource {
  // table view data source methods
}

// MARK: - UIScrollViewDelegate

extension MyViewController: UIScrollViewDelegate {
  // scroll view delegate methods
}
```

不推荐：

```
class MyViewController: UIViewController, UITableViewDataSource, UIScrollViewDelegate {
```

```
    // all methods
}
```

2、Bool类型命名：用is最为前缀

```
var isString: Bool = true
```

3、枚举定义尽量简写，不要包括类型前缀

```
public enum UITableViewRowAnimation : Int {
    case fade
    case right // slide in from right (or out to right)
    case left
    case top
    case bottom
    case none // available in iOS 3.0
    case middle // available in iOS 3.2.  attempts to keep cell centered in the space it will/did occupy
    case automatic // available in iOS 5.0.  chooses an appropriate animation style for you
}

enum ZHCodeStandardEnum: Int {
    case code = 0 // code无特殊情况 字母小写
    case code1 = 1 // code1
    case code2 = 2  // code2
}
```

4、swift建议不要使用前缀

```
推荐
HomeViewController
Bundle

不推荐
NEHomeViewController
NSBundle
```

5、减少不必要的简写

```
推荐
let viewFrame = view.frame
let textField = ...
let table = ...
let controller = ...
let button = ...
let label =...

不推荐
let r = view.frame
let tf = ...
let tb = ...
let vc =...
let btn = ...
let lbl =...
```

6、变量命名应该能推断出该变量类型，如果不能推断，则需要以变量类型结尾

```
推荐
class TestClass: class {
    // UIKit的子类，后缀最好加上类型信息
    let coverImageView: UIImageView

    @IBOutlet weak var usernameTextField: UITextField!

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

     // UIViewContrller以ViewController结尾
     let fromViewController: UIViewController
}

不推荐
class TestClass: class {
     // image不是UIImageView类型
     let coverImage: UIImageView

     // or cover不能表明其是UIImageView类型
     var cover: UIImageView

     // String后缀多余
     let firstNameString: String

     // UIViewContrller不要缩写
     let fromVC: UIViewController
}
```

7、省略所有的冗余的参数标签

```
func min(_ number1: int, _ number2: int) {
    /* ... */
}

min(1, 2)
```

8、进行安全值类型转换的构造方法可以省略参数标签，非安全类型转换则需要添加参数标签以表示类型转换方法

```
extension UInt32 {
  /// 安全值类型转换，16位转32位，可省略参数标签
  init(_ value: Int16)

  /// 非安全类型转换，64位转32位，不可省略参数标签
  /// 截断显示
  init(truncating source: UInt64)

  /// 非安全类型转换，64位转32位，不可省略参数标签
   /// 显示最接近的近似值
   init(saturating valueToApproximate: UInt64)
}
```

9、当第一个参数构成整个语句的介词时（如，at, by, for, in, to, with 等），为第一个参数添加介词参数标签

```
推荐
```

```
// 添加介词标签havingLength
func removeBoxes(havingLength length: int) {
    /* ... */
}

x.removeBoxes(havingLength: 12)
```

#例外情况是，当后面所有参数构成独立短语时，则介词提前：

```
推荐
// 介词To提前
a.moveTo(x: b, y: c)

// 介词From提前
a.fadeFrom(red: b, green: c, blue: d)


不推荐
a.move(toX: b, y: c)
a.fade(fromRed: b, green: c, blue: d)
```

10、当第一个参数构成整个语句一部分时，省略第一个参数标签，否则需要添加第一个参数标签。其余情况下，给除第一个参数外的参数都添加标签

```
// 参数构成语句一部分，省略第一个参数标签
x.addSubview(y)

// 参数不构成语句一部分，不省略第一个参数标签
view.dismiss(animated: false)
```

#不要使用冗余的单词，特别是与参数及参数标签重复

```
推荐
func remove(_ member: Element) -> Element?


不推荐
func removeElement(_ member: Element) -> Element?
```

#命名出现缩写词，缩写词要么全部大写，要么全部小写，以首字母大小写为准

```
推荐
let urlRouterString = "https://xxxxx"
let htmlString = "xxxx"
class HTMLModel {
    /* ... */
}
struct URLRouter {
    /* ... */
}


不推荐
let uRLRouterString = "https://xxxxx"
let hTMLString = "xxxx"
```

```
class HtmlModel {
    /* ... */
}
struct UrlRouter {
    /* ... */
}
```

参数名要准确的表达出参数类型：

```
// 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!
}
```

11、数组和字典变量定义，定义时需要标明泛型类型，并使用更简洁的语法

```
推荐
var names: [String] = []
var lookup: [String: Int] = [:]

不推荐
var names = [String]()
var names: Array<String> = [String]() // 不够简洁

var lookup = [String: Int]()
 var lookup: Dictionary<String, Int> = [String: Int]() // 不够简洁
```

12、常量定义，建议尽可能定义在Type类型里面，避免污染全局命名空间

```
推荐
class TestTabelViewCell: UITableViewCell {
    static let kCellHeight = 80.0

    /* ... */
}
// uses
let cellHeight = TestTabelViewCell.kCellHeight


 不推荐
 let kCellHeight = 80.0
 class TestTabelViewCell: UITableViewCell {
     /* ... */
 }
 // uses
 let cellHeight = kCellHeight
```

14、当方法最后一个参数是Closure类型，调用时建议使用尾随闭包语法

```
推荐
UIView.animate(withDuration: 1.0) {
  self.myView.alpha = 0
}

不推荐
UIView.animate(withDuration: 1.0, animations: {
  self.myView.alpha = 0
})
```

15、 一般情况下，在逗号后面加一个空格

```
推荐
let testArray = [1, 2, 3, 4, 5]

不推荐
let testArray = [1,2,3,4,5]
```

16、基类属性及方法

```
// 私有属性及方法使用前缀 _
```

```
// let
private let _testA: String = "testA"
public let testB: Int = 2019

// var
private var _testC: String = "testC"
public var testD: String = "testD"

// 方法
private func _testFunction() {
    /* ... */
}
public func b_testFunction() {
    /* ... */
}
public func base_testFunction() {
    /* ... */
}
```

二、注释

#尽可能使用Xcode注释快捷键（⌘⌥/）[command + option + /]

```
推荐
/// <#Description#>
///
/// - Parameter testString: <#testString description#>
/// - Returns: <#return value description#>
func testFunction(testString: String?) -> String? {
    /* ... */
}


不推荐
// Comment
func testFunction(testString: String?) -> String? {
    /* ... */
}
```

#使用// MARK: -，按功能和协议/代理分组

```
/// MARK顺序没有强制要求，但System API & Public API一般分别放在第一块和第二块。

// MARK: - Public

// MARK: - Request

// MARK: - Action

// MARK: - Private

// MARK: - xxxDelegate


```

MARK： 用于方法或函数的注释

TODO：表示这里的代码还需继续处理

FIXME：表示这里修改了代码

```
// MARK: - 网络判断
extension IBCBaseViewController{
    /// 判断当前ViewController是否显示
    ///
    /// - Returns: 是否显示
    public func checkIsShow() -> Bool{
        // FIXME: 修改bug
        if self.isViewLoaded && self.view.window != nil{
            //当前viewController在显示
             return true
         }
         return false
     }

     /// 网络状态更改
     @objc public func networkStatusChange(){
         if checkIsShow(){
             shouldUpdateDatas()
         }else{
             // TODO: 未在当前页显示处理
         }
     }

     /// 需要更新数据
     @objc open func shouldUpdateDatas(){
         // TODO: 基础处理
     }
}
```

#swift 注释是支持 mark down 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 {
    /* ... */
}
```

12、对外接口不兼容时，使用@available(iOS x.0, *)标明接口适配起始系统版本号

```swift
@available(iOS x.0, *)
class myClass {

}

@available(iOS x.0, *)
func myFunction() {

}
```

三、关于闭包

在Closures中使用self时避免循环引用

推荐

```swift
resource.request().onComplete { [weak self] response in
    guard let strongSelf = self else {
        return
    }
    let model = strongSelf.updateModel(response)
    strongSelf.updateUI(model)
}
```

不推荐

```swift
// 不推荐使用unowned
// might crash if self is released before response returns
resource.request().onComplete { [unowned self] response in
    let model = self.updateModel(response)
    self.updateUI(model)
}
```

不推荐

```swift
// deallocate could happen between updating the model and updating UI
resource.request().onComplete { [weak self] response in
    let model = self?.updateModel(response)
    self?.updateUI(model)
}
```

Golden Path，最短路径原则

推荐

```swift
func login(with username: String?, password: String?) throws -> LoginError {
  guard let username = contextusername else {
    throw .noUsername
  }
  guard let password = password else {
    throw .noPassword
  }

  /* login code */
}
```

不推荐

```
func login(with username: String?, password: String?) throws -> LoginError {
  if let username = username {
    if let password = inputDatapassword {
        /* login code */
    } else {
        throw .noPassword
    }
  } else {
    throw .noUsername
  }
}
```

单例

```
class TestManager {
    static let shared = TestManager()

    /* ... */
}
```

四、缩进和换行

1 使用四个空格进行缩进。

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)

3 确保每个文件结尾都有空白行。

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).

5 左大括号不用另起一行。

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

```
// 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!")
}
```

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)
```

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)
     })
```

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
}
```

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 {
    /* ... */
}
```

五、操作符

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 {
    /* ... */
}
```

六、其他

1 多使用let，少使用var
2 少用!去强制解包
3 可选类型拆包取值时，使用if let判断
4 不要使用 as! 或 try!
5 数组访问尽可能使用 .first 或 .last, 推荐使用 for item in items 而不是 for i in 0..
6 如果变量能够推断出类型，则不建议声明变量时指明类型
7 如果变量能够推断出类型，则不建议声明变量时指明类型
8 switch case选项不需要使用break关键词
9 访问控制
10 对于私有访问，如果在文件内不能被修改，则标记为private；如果在文件内可修改，则标记为fileprivate
11 对于公有访问，如果不希望在外面继承或者override，则标记为public，否则标明为open
12 访问控制权限关键字应该写在最前面，除了@IBOutlet、IBAction、@discardableResult、static 等关键字在最前面
13 如调用者可以不使用方法的返回值，则需要使用@discardableResult标明
14 使用==和!=判断内容上是否一致
15 使用===和!==判断class类型对象是否同一个引用，而不是用 ==和!=
16 Runtime兼容
17 Swift语言本身对Runtime并不支持，需要在属性或者方法前添加dynamic修饰符才能获取动态型，继承自NSObject的类其继承的父类的方法也具有动态型，子类的属性和方法也需要加dynamic才能获取动态性

七、Objective-C兼容

1 Swift接口不对Objective-C兼容，在编译器或者Coding中就会出现错误
2 暴漏给Objective-C的任何接口，需要添加@objc关键字，如果定义的类继承自NSObject则不需要添加
3 如果方法参数或者返回值为空，则需要标明为可选类型

八、2019.10.8补充

关于空白：

- 用 tab，而非 空格
- 文件结束时留一空行
- 用足够的空行把代码分割成合理的块
- 不要在一行结尾留下空白
- 千万别在空行留下缩进
- 一块代码中间不要留空白

关于返回：

```
if n.isNumber {
```

```
    /* ... */
} else {
    return
}

用这个：
guard n.isNumber else {
    return
}
/* ... */
```

关于解包：

```
推荐：
if let foo = foo {
    // Use unwrapped `foo` value in here
} else {
    // If appropriate, handle the case where the optional is nil
}
或者使用可选链，比如：
// Call the function if `foo` is not nil. If `foo` is nil, ignore we ever tried to make the call
foo?.callSomethingIfFooIsNotNil()

不推荐：
foo!.callSomethingIfFooIsNotNil()
```