SWIFTER

100 must know tips for Swift

2nd Edition

Wei Wang (@onevcat)

Table of Contents

Introduction

> Although all of us wish we could dive into Swift soon, I guess I should introduce the target of this book and who are the readers this book written for.

Why do I should read this book?

Quite a lot of learners of Swift - whether those who are totally new to Cocoa/Cocoa Touch, or used to write in Objective-C - are struggling in the same situation: how to improve their professional skill after getting started with Swift. Maybe it happens to be your situation as well. When you finished the last page of Apple's Swift tutorial, thinking you had mastered the new language, then created an Xcode project with Swift, and stopped at the very first line of code. You have to recall when to use Optional and when not to. You are not sure of how to express these familiar APIs in the new language. You are always getting trouble with compiling the code, without any clue for fixing them.

Don't worry, that is quite normal. The Apple's tutorial is written for showing the syntax to you. If you want to make Swift a powerful weapon in your daily life, you have to learn it deeper and use it more. The target of this book is introducing some innovative points for you, and improving your practice skill. These parts of knowledge are necessary for engineers who are using or wishing to use Swift as their next programming language.

What is contained in this book?

It is a collection of knowledge points and tips for Swift. I myself attended WWDC 14, and saw the birth of Swift with my own eyes. From the very first minute, I am learning Swift. By now, I concluded 100 tips for this language, and divided them into 3 sections, from the very basic ones to some high level ones. Each tip has unique content which should be understood by a senior developer.

This book is particularly suitable to be used as a reference and supplement of Swift official documentation. It would also become a fancy hand-book for developers. For the detail of what are contained in the book, you can refer to the Contents page of the book.

What is not contained in this book?

This should not be your FIRST book for learning Swift, and this book is no longer a tutorial for you to develop a simple calculator or note app. The main purpose of this book is clear - to explore those ignored details of Swift. Although we will not discuss the whole language in a systematic way, these points are utilized widely in developing. Based on this purpose, the chapters of the book are organized in a loose structure.

Generally speaking, if you are just looking for a beginner book of Swift, you may not probably choose this one. You can first read Apple's documentation on Swift, then have a look at this book later on. If you have been already an iOS developer for a while, or learned some Swift before and now wish to go further, this book is right for you.

How to read this book

100 is not a small number for tips. Fortunately, every part of this book is not so tightly with each other, that allows you to just open the book and pick any tip you like. Although I recommend to follow the order - because I paid specially attention not to refer the harder part in the earlier chapters, it is not a must-obeyed rule. And there are links to the referred chapters as well, you can jump through the book easily, and review the related chapters as you wish. If you are not interested in some chapters, just skip them first. You can pick these ones would help you mostly first, then go back to the skipped chapters later.

I suggest practicing the code in Xcode while reading. It could be a help to understand the intension of these sample code. Every sample code is not long, but prepared carefully. I hope you can "talk" to me by repeating these codes.

Code Sample

There are some code samples in most tips, mainly in Swift, and some in Objective-C as a reference. All code should be able to run in Swift 2 (which bundled in Xcode 7). Of course, the change is happening in Swift very rapidly now. Some code might be needed some modification to be compiled and run correctly in later Swift version. If you find it, please open an issue in the repository of this book, I will fix them as soon as possible.

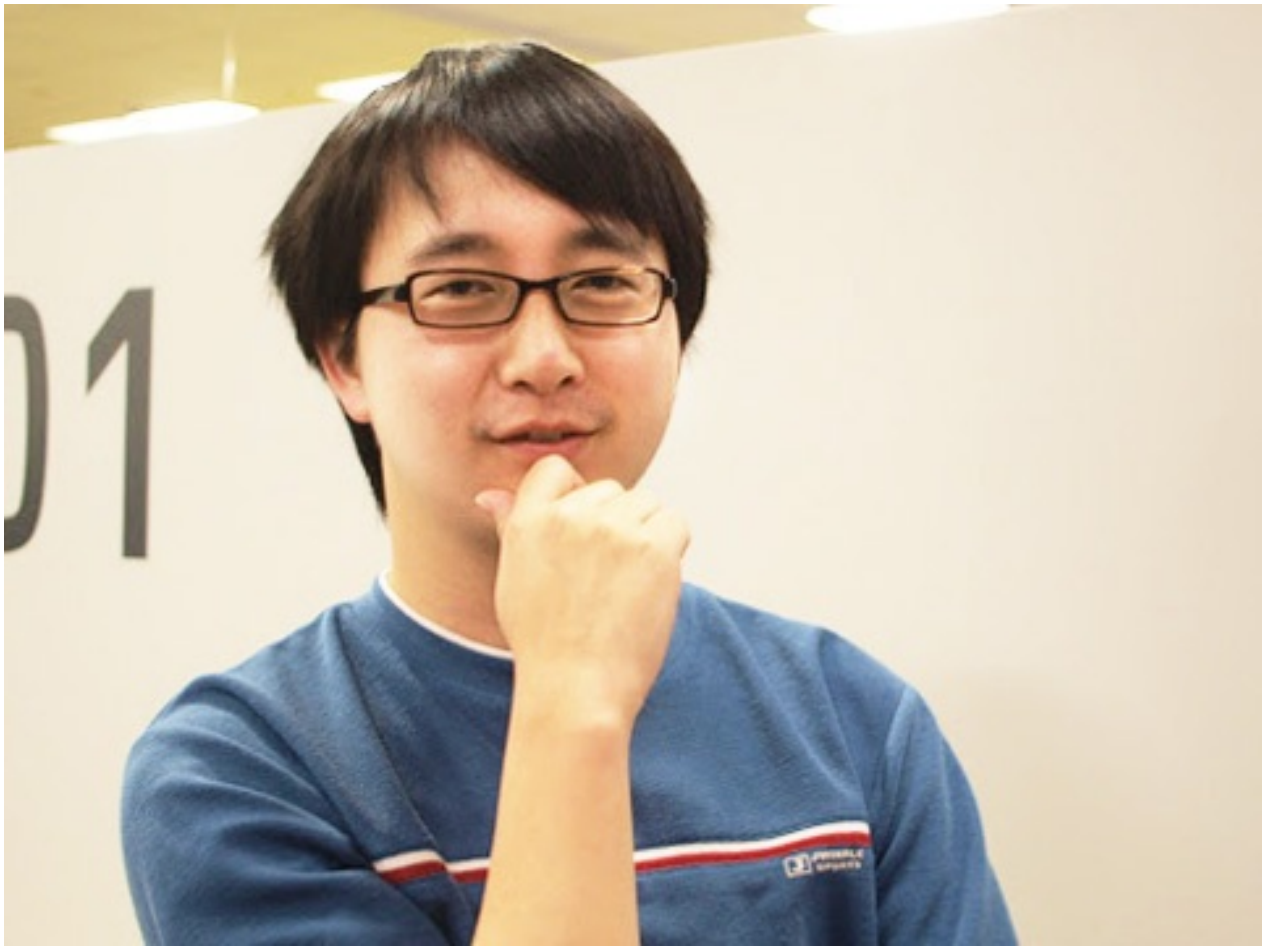
If not specially pointed out, these codes could be executed in both Playground and real project, and should share the same result. But there is also the situation of the code could be only tested in Playground OR in a project. This is always caused by the limitation of the platform, and I will mention it in the chapter if this happens.

Errata and Feedback

Swift is of great speed development and this book is under Swift 2 currently. As some new features introduced into Swift, there might be outdated content in the book. Despite the fact that I will improve this book, it might take some time after the upgrading of the language itself.

If you find any problem in reading this book, it would greatly help if you can report it in the issue page. I will confirm these feedback and fix the issues if necessary.

About the author



Wei Wang (onevcat) is an iOS developer from China. He got his master degree in Tsinghua University, which is one of the best universities in China. When he was an undergraduate he started to develop iOS app. Now he is a senior engineer with Cocoa and Objective-C experience, as well as Swift. He is writing a Chinese blog about iOS development in OneV's Den with thousands of subscribers. He loves to contribute to open source community and is the author of a famous Xcode plugin VVDocumenter as well.

Now Wei Wang is working in Japan for a mobile communication company. If you want to know more about him, please visit his personal page and follow him on Twitter.

Part I. New in Swift

Pattern Match

In the previous regular expression tip, we implemented a `=~` operator to do simple text match. Although there is no built-in support for regular expression, there is a similar feature in Swift. That is pattern match.

Conceptually, regular expression is only a subset of pattern match. However, the pattern match support in Swift now is quite primary and simple. It can only handle equality match and range match. There is a pattern match operator in the standard library, which is not so widely known. It is `~=`. We can search it in the standard library and find the related APIs below:

```
func ~=<T : Equatable>(a: T, b: T) -> Bool

func ~=<T>(lhs: _OptionalNilComparisonType, rhs: T?) -> Bool

func ~=<I : IntervalType>(pattern: I, value: I.Bound) -> Bool
```

They can accept parameters of which types can be checked equality, types can be compared to `nil`, and types of a range and some specified value respectively. All methods return a `Bool` to indicate whether the match is successful or not. Remember something? It is not so obvious at first, but let us see some important use of `switch` in Swift:

1. Types can be checked equality:

```
let password = "akfuv(3"
switch password {
    case "akfuv(3": print("Passed")
    default:        print("Failed")
}
```

2. Types can be compared to optional value:

```
let num: Int? = nil
switch num {
    case nil: print("No value")
    default:  print("\(num!)")
}
```

3. Types of a range. Check some value in the range or not:

```
let x = 0.5
switch x {
    case -1.0...1.0: print("In range")
    default: print("Out of range")
}
```

That's right! `switch` statement is using `~=` to do a pattern match under the hood. The pattern is set as the left operand for `~=` by `case`. And the `switch` tells it what is waiting to be matched, as the right operand. This process is done by Swift implicitly. After knowing this, we can rely on the `switch` to make some fun things. By applying our customized pattern, sometimes we can write cleaner and more logical code. Here I will give an example of using the regular expression as the pattern in `switch`. So you can draw a basic idea of how to do.

Overriding the `~=` operator is always a good start point. Add a new version of `~=` and make it accepting an `NSRegularExpression` object as pattern, to match a `String` input:

```
func ~=(pattern: NSRegularExpression, input: String) -> Bool {
    return pattern.numberOfMatchesInString(input,
        options: nil,
        range: NSRange(location: 0, length: count(input))) > 0
}
```

For convenience, I also add an operator to convert a text string to `NSRegularExpression`. Of course you could use `StringLiteralConvertible` as well, but it is related to another tip and here I decide not to use it.

```
prefix operator ~/ {}

prefix func ~/(pattern: String) -> NSRegularExpression {
    return NSRegularExpression(pattern: pattern, options: nil, error: nil)!
}
```

It is so simple. Now we can use a regular expression in `case`, and let it match the string in `switch`:

```
let contact = ("http://onevcat.com", "onev@onevcat.com")

let mailRegex: NSRegularExpression
let siteRegex: NSRegularExpression

mailRegex =
    try ~/"^([a-z0-9_\\.-]+)@([\\da-z\\.-]+)\\.([a-z\\.]{2,6})$"
siteRegex =
    try ~/"^(https?:\\/\\/)?([\\da-z\\.-]+)\\.([a-z\\.]{2,6})([\\/\\w \\.-]*)*\\/?$"

switch contact {
    case (siteRegex, mailRegex): print("Valid website and email")
    case (_, mailRegex): print("Valid email only")
    case (siteRegex, _): print("Valid website only")
    default: print("Both invalid")
}

// Output:
// Valid website and email
```

II. From Objective-C/C to Swift

Selector

`@selector` is keyword in Objective-C. It could convert a method to a SEL type, which turns out behaving as a "function pointer". In the old Objective-C's age, selector is widely used, from setting the target-action to introspecting. In Objective-C, the way of generating a selector would be:

```
-(void) callMe {
    //...
}

-(void) callMeWithParam:(id)obj {
    //...
}

SEL someMethod = @selector(callMe);
SEL anotherMethod = @selector(callMeWithParam:);

// Or we can use NSSelectorFromString as well
// SEL someMethod = NSSelectorFromString(@"callMe");
// SEL anotherMethod = NSSelectorFromString(@"callMeWithParam:");
```

For writing less code, `@selector` is widely used. But if we need to decide which method to call at runtime, `NSSelectorFromString` would be preferred, so we could generate the method string dynamically and send the message with the name.

There is no `@selector` keyword anymore in Swift. Instead of it, from Swift 2.2, we could get a selector by using `#selector` and pass the function name which exposed to Objective-C to it. Similarly, the original `SEL` type is also replaced by a `Selector` struct. The two examples above could be rewritten as below:

```
func callMe() {
    //...
}

func callMeWithParam(obj: AnyObject!) {
    //...
}

let someMethod = #selector(callMe)
let anotherMethod = #selector(callMeWithParam(_:))
```

Same as Objective-C, you have to add the colon (`:`) following `callMeWithParam` to construct the full method name. Method with multiple parameters is similar, like this:

```
func turnByAngle(theAngle: Int, speed: Float) {
    //...
}

let method = #selector(turnByAngle(_:speed:))
```

Besides of that, since `Selector` type conforms the `StringLiteralConvertible` protocol, we could even use a string to assign a selector, without using its init method explicitly. It is semantic in some situation, for example, setting a call back method when adding a notification observer:

```
NSNotificationCenter.defaultCenter().addObserver(self,
        selector: "callMe", name: "CallMeNotification", object: nil)
```

We should pay special attention on the fact of selector being a concept of Objective-C runtime. If the method of a selector is only exposed in Swift, but not Objective-C (or in other words, it is a private method of Swift), you might get an "unrecognized selector" exception when sending method to this selector:

> ## This is wrong code
>
> ```swift private func callMe() { //... }

NSTimer.scheduledTimerWithTimeInterval(1, target: self, selector:#selector(callMe), userInfo: nil, repeats: true)

```
One solution would be adding `@objc` keyword before `private`, so the runtime could get to know

```swift
@objc private func callMe() {
    //...
}

NSTimer.scheduledTimerWithTimeInterval(1, target: self,
             selector:#selector(callMe), userInfo: nil, repeats: true)
```

Lastly, if the method name is unique in the scope it belongs to, we could use the name alone without full signature/ Compared to the full signature, it will be a bit easier for us to write the short version:

```
let someMethod = #selector(callMe)
let anotherMethod = #selector(callMeWithParam)
let method = #selector(turnByAngle)
```

However, if there are two or more functions with the same name in the same scope, Swift will not be happy, even the function signatures are different between them:
```

```
func commonFunc() {

}

func commonFunc(input: Int) -> Int {
    return input
}

let method = #selector(commonFunc)
// Compile error, there is ambiguity in `commonFunc`
```

We have to cast them to corresponded types to make it compile:

```
let method1 = #selector(commonFunc as ()->())
let method2 = #selector(commonFunc as Int->Int)
```

III. Swift In Practice

Printable and DebugPrintable

> Output will not be involved when implementing these two protocols in Playground and Swift REPL.
>
> If you need to verify the code in this tip, you need to do it in a project.

When defining and implementing a type, a common and progressive way in Swift is defining a simple type first, and then adding features by conforming protocols by extension. This follows a good design concept and contributes to improving scalability. In Objective-C, we can use the combination of protocol and category to do similar things. It is even simpler in Swift.

`Printable` and `DebugPrintable` protocols are good example for it. For a regular object, when we use `print` on it, only the type of this object would be printed. If we need some more useful information, we can extend this type to make it conform `Printable` protocol. Consider we have a calendar app, in which some meeting appointments are stored. The model type contains the date, position and attendee name:

```
struct Meeting {
    var date: NSDate
    var place: String
    var attendeeName: String
}

let meeting = Meeting(date: NSDate(timeIntervalSinceNow: 86400),
                      place: "Room B1",
                  attendeeName: "John")
print(meeting)
// Output:
// YourModuleName.Meeting
```

It is a meaningless output. What we need would be a formatted output, something like this:

```
print("Meeting with \(meeting.attendeeName) in \(meeting.place) at \(meeting.date)")
// Output:
// Meeting with John in Room B1 at 2014-08-25 11:05:28 +0000
```

Much better! But it is not acceptable as well if we need to write the log statement every time. Proper way to do this is using `Printable` protocol, defining a template string for every `print` calling. Compared change the `Meeting` struct, we prefer an `extension` on it, to avoid messing up the core model with the helper methods:

```
extension Meeting: Printable {
    var description: String {
        return "Meeting with \(meeting.attendeeName) in \(meeting.place) at \(meeting.date)"
    }
}
```

Now, we can just simply pass the `meeting` to `print` to get the formatted output:

```
print(meeting)
// Output:
// Meeting with John in Room B1 at 2014-08-25 11:05:28 +0000
```

`DebugPrintable` is similar to `Printable`, except it will only called in a debugger logout. For these types conforming to `DebugPrintable`, we can use a printing command, such as `po meeting`, to print the `debugDescription` string defined in its `DebugPrintable`.

Acknowledgements

- Swift official documentation
- Apple Developer Forum
- swift tag on Stackoverflow
- NSHipster
- NSBlog Friday FAQ
- Airspeed Velocity

Release Note

2.0.0 (2016/5/18)

Compatible for Swift 2.2.

1.0.0 (2014/9/19)

First Edition