What’s New in Swift 3

Apple integrated Swift 3 into Xcode 8 beta at WWDC and will release the final version later on this year. This is the language’s first version that is open source and works both on Mac OS X and Linux. If you have followed the [Swift Evolution](https://github.com/apple/swift-evolution) process since December last year and even already played with it in the [IBM sandbox](https://swiftlang.ng.bluemix.net/#/repl), you know that there are a lot of changes. It is quite sure if you compile your app in Xcode 8, it will break your code.

The changes of Swift 3 can be grouped into two major categories:

Removed features that have already been deprecated in Swift 2.2

Language modernisation issues

Let’s start with the removed ones, since they are easier to understand and you may have encountered them before as warnings in Xcode 7.3.

**++ and — Operators**

The increment and decrement operators are inherited from C and their functionality is straightforward – add or subtract 1 to a certain variable:

var i = 0; i++; ++I; i--; --i

However, things get complicated when it comes to deciding which one to choose. Each of them comes in two possible flavours: prefix and postfix – they are all functions under the hood and return values which you may use or discard thanks to [operator overloading](http://www.appcoda.com/operator-overloading-swift/).

This is overwhelming for beginners, so they have been removed – use the addition (+=) and subtraction (-=) assignment operators instead:

var i = 0; i+= 1; i-= 1

Of course, you can use the addition (+) and subtraction (-) operators as well – the compound assignment operators approach is shorter though:

i = i + 1; i = i - 1

Further reading: If you want to learn more about the motivation behind this change, check out [Chris Lattner’s proposal](https://github.com/apple/swift-evolution/blob/master/proposals/0004-remove-pre-post-inc-decrement.md) on the removal of ++ and — operators.

**C-style for Loop is History**

The most common usage of the increment and decrement operators is the C-style for loop. Removing the operators means that this feature is gone as well, since there is nothing you can do with it that you can’t do with the for-in control flow statement and ranges.

If you have some programming background, you probably write the for loop like this to print the numbers from 1 to 10:

for (i = 1; i <= 10; i++) {  print(i) }

In Swift 3, you’re no longer allowed to do that. And this is its Swift 3 counterpart – notice the closed range operator (…) in action:

for i in 1...10 {  print(i) }

Alternatively, you can also use the for-each loop with closures and shorthand arguments – read more about loops [here](https://cosminpupaza.wordpress.com/2015/12/04/for-vs-while-a-beginners-approach/).

(1...10).forEach {  print($0) }

Further reading: If you want to learn more about the motivation behind this change, check out [Erica Sadun’s proposal](https://github.com/apple/swift-evolution/blob/master/proposals/0007-remove-c-style-for-loops.md) on the removal of C-style for loop.

**Remove var from Function Parameters**

Function parameters are usually defined as constants, since you don’t need to modify them in its body. However, there are certain cases when declaring them as variables might come in handy. In Swift 2, you can mark a function parameter as variable with the var keyword. Once the parameter is marked as var, it creates a local copy of the value so you can modify its value in the body of the function.

As an example, the following function determines the greatest common divisor of two given numbers – if you skipped the Maths class back in high school, read more about it [here](https://en.wikipedia.org/wiki/Greatest_common_divisor):

func gcd(var a: Int, var b: Int) -> Int {

  if (a == b) { return a}

  repeat {

    if (a > b) {  a = a – b}

      else { b = b – a}

  } while (a != b)

  return a

}

The algorithm is simple: if both numbers are already equal, return one of them. Otherwise, compare them, subtract the smaller one from the bigger one and assign the result to the larger one until they become equal and return any one of them. As you can see, by marking a and b as variables, we can change their values in the function.

Swift 3 no longer allows developers to set function parameters as variables as Swift developers may get confused between var and inout. So the latest version of Swift simply removes var from function parameters.

Therefore, to write the same gcd function in Swift 3, it requires a different approach. You’ll need to save the values of the function parameters to local variables:

func gcd(a: Int, b: Int) -> Int {

  if (a == b) {  return a}

  var c = a,  var d = b

  repeat {

    if (c > d) { c = c – d}

       else {  d = d – c}

  } while (c != d)

  return c

}

If you want to learn more about the motivation behind the removal, you can check out the [original proposal](https://github.com/apple/swift-evolution/blob/master/proposals/0003-remove-var-parameters.md).

**Consistent Label Behaviour for Function Parameters**

Function parameter lists are tuples under the hood, so you can use them to call functions, as long as the tuple’s structure matches the function’s prototype. Take the gcd() function as an example. You can invoke the call like this:

gcd(8, b: 12)

Or you may even call the function like below:

let number = (8, b: 12); gcd(number)

As you can see, you do not need to specify the label of the first parameter in Swift 2. However, you have to specify the label of the second (and the rest of the parameters) when calling the function.

This syntax is confusing for beginners, so it is designed to standardize the label behaviour. In Swift 3, you call the function like this:

gcd(a: 8, b: 12)

You have to explicitly specify the label of the first parameter. If you don’t do that, Xcode 8 will show you an error.

Your first reaction to this change may be “OMG! I will have to make a lot of modifications to my existing code.” You’re right. That’s tons of changes. So Apple offers a way to suppress the first parameter label of a function call. You can add an underscore to the first parameter like this:

func gcd(\_ a: Int, b: Int) -> Int {…}

By doing this, you can invoke the function using the old way – without specifying the first label. This would make your code migration from Swift 2 to Swift 3 simpler.

Further reading: For the motivation and intent behind this change, you can check out [this proposal](https://github.com/apple/swift-evolution/blob/master/proposals/0046-first-label.md).

**Selectors as Strings No Longer Works**

Let’s create a button and make it do something when you tap on it – use playgrounds only, no Interface Builder allowed:

import UIKit

import XCPlayground

class Responder: NSObject { func tap() {  print("Button pressed")}}

let responder = Responder()

let button = UIButton(type: .System)

button.setTitle("Button", forState: .Normal)

button.addTarget(responder,action:"tap",

forControlEv ents: .TouchUpInside)

button.sizeToFit(); button.center = CGPoint(x: 50, y: 25)

let frame = CGRect(x: 0, y: 0, width: 100, height: 50)

let view = UIView(frame: frame), view.addSubview(button)

XCPlaygroundPage.currentPage.liveView = view

There’s quite a lot going on here, so let’s break it into steps:

Import the UIKit and XCPlayground frameworks – you need them to create the button and show it in the playground’s assistant editor.

Note: You should enable the assistant editor in Xcode to interact with the button: View -> Assistant Editor -> Show Assistant Editor.

Define the tap method which fires when the user presses the button and make a responder object for the button’s target – its base class is NSObject, because selectors only work with Objective-C methods.

Declare the button and set its properties.

Declare the view and its corresponding frame, add the button to it and display it in the playground’s assistant editor.

Take a look at the highlighted code. The button’s selector is a string. If you type it wrong, the code will compile but crash at runtime, since no corresponding method can be found.

To resolve the potential issue in compile time, Swift 3 replaces string selectors with the #selector() keyword. This enables the compiler to detect the problem early if you don’t get the method’s name right.

button.addTarget(responder, action: #selector(Responder.tap), for: .touchUpInside)

Further reading: For the motivation and intent behind this change, you can check out [Doug Gregor’s proposal](https://github.com/apple/swift-evolution/blob/master/proposals/0022-objc-selectors.md).

That’s it for the removed features. Now let’s move on to the language modernisation highlights.

**Key-paths as Strings**

This feature is similar to the previous one, but it applies to key-value coding (KVC) and key-value observing (KVO):

class Person: NSObject {

  var name: String = ""; init(name: String) {  self.name = name}}

let me = Person(name: "Cosmin"); me.valueForKeyPath("name")

You create a Person class that is key-value coding compliant, make my identity with the class designated initialiser and use the corresponding key-path to determine my name. Again, if you get it wrong, everything will blow up and I won’t be happy!

Fortunately, this will not happen in Swift 3 anymore. Key-path strings have been replaced with the #keyPath() expression:

class Person: NSObject {

  var name: String = ""; init(name: String) {  self.name = name}}

let me = Person(name: "Cosmin")

me.value(forKeyPath: #keyPath(Person.name))

Further reading: For the motivation and intent behind this change, you can check out [David Hart’s proposal](https://github.com/apple/swift-evolution/blob/master/proposals/0062-objc-keypaths.md).

**Drop NS Prefix for Foundation Types**

The NS prefix goes way back for Foundation types – if you are in the mood for a history lesson, check [this](http://stackoverflow.com/questions/473758/what-does-the-ns-prefix-mean) out. A typical example is JSON parsing:

let file = NSBundle.mainBundle().pathForResource("tutorials", ofType: "json")

let url = NSURL(fileURLWithPath: file!)

let data = NSData(contentsOfURL: url)

let json = try! NSJSONSerialization.JSONObjectWithData(data!, options: []); print(json)

You use Foundation classes to connect to the file and extract the JSON data in an appropriate way: NSBundle -> NSURL -> NSData -> NSJSONSerialization.

The NS prefix has been dropped in Swift 3, so it all boils down to Bundle -> URL -> Data -> JSONSerialization:

let file = Bundle.main().pathForResource("tutorials", ofType: "json")

let url = URL(fileURLWithPath: file!)

let data = try! Data(contentsOf: url)

let json = try! JSONSerialization.jsonObject(with: data); print(json)

Further reading: For this naming convention change, you can check out [this proposal](https://github.com/apple/swift-evolution/blob/master/proposals/0086-drop-foundation-ns.md)written by Tony Parker and Philippe Hausler.

**M\_PI vs .pi**

Let’s compute the circumference and area of a circle with a given radius:

let r =  3.0; let circumference = 2 \* M\_PI \* r; let area = M\_PI \* r \* r

For old versions of Swift, you use M\_PI to refer to the pi constant. Swift 3 integrates the pi constant in the Float, Double and CGFloat types:

Float.pi; Double.pi; CGFloat.pi

The above piece of code will be written like this in Swift 3:

let r = 3.0; let circumference = 2 \* Double.pi \* r; let area = Double.pi \* r \* r

With type inference, you can even omit the type. Here is a shorter version:

let r = 3.0; let circumference = 2 \* Double.pi \* r; let area = Double.pi \* r \* r

**Grand Central Dispatch**

Grand Central Dispatch (CGD) is used for networking operations that don’t block the user interface on the main thread. It’s written in C and its API is overwhelming to beginners, even for trivial tasks such as creating an asynchronous queue and making it do something:

let queue = dispatch\_queue\_create("Swift 2.2", nil)

dispatch\_async(queue) {  print("Swift 2.2 queue")}

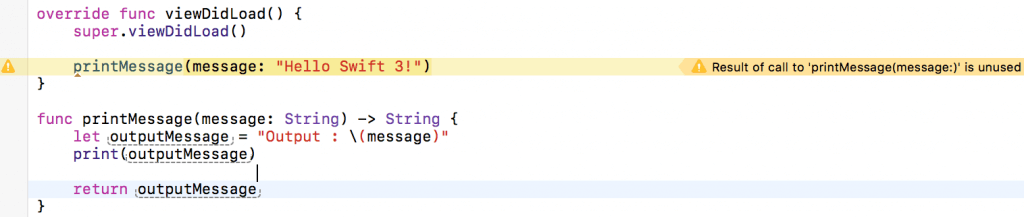
Swift 3 eliminates all the boilerplate code and redundant stuff by taking an object oriented approach:

let queue = DispatchQueue(label: "Swift 3"); queue.async {   print("Swift 3 queue")}

Further reading: For further information about this change, you can check out [this proposal](https://github.com/apple/swift-evolution/blob/master/proposals/0088-libdispatch-for-swift3.md) written by Matt Wright.

**@discardableResult**

In Swift 3, Xcode will show you a warning if you do not use a return value of a function or method. Here is an example:



In the above code, the printMessage method returns the resulting message to the caller. However, the returned value is not used. It may be a potential issue, so the compiler in Swift 3 will give you a warning.

In case, it is not a must to handle the returned value. You can suppress the warning by adding @discardableResult to the method declaration:

override func viewDidLoad() {

    super.viewDidLoad(); ，  printMessage(message: "Hello Swift 3!")}

@discardableResult

func printMessage(message: String) -> String {

let outputMessage = "Output : \(message)"

print(outputMessage),  return outputMessage }

**Summary**

That’s all about Swift 3. The new version of Swift is a major release that makes the language even better. It contains a lot of fundamental changes that will definitely impact your existing Swift code. I hope this tutorial helps you better understand the changes, and hopefully, save you some time to migrate your Swift project.

All the code in this tutorial can be found in [this Playground project](https://github.com/appcoda/Swift3Playgrounds/blob/master/Swift%203.playground.zip?raw=true). I have tested it on Xcode 8 Beta. So make sure you use Xcode 8 to run the project.

If you have any questions or issues, please let me know. Happy coding!

**Swift 3中的新变化**

Apple在WWDC上已将Swift 3 整合进了Xcode 8 beta版中,并且将会在今年末发布最终的版本。这是Swift第一个开源的，并且可以同时在Mac OS X和Linux上使用的版本。假如你从去年12月开始就已经关注了Swift的更新过程和IBM sandbox项目，相信你应该会发现这其中有很大的变化。提醒一下，假如你尝试在Xcode 8中编译你的现有项目，你将会无法编译。

Swift 3的变化大致可以归纳为以下两个方面：

1. 移除了在swift 2.2已经废除的特性

2. 语言更加接近自然语言

移除的这些特性比较容易理解，并且在7.3中被处理成了警告，你之前很可能已经遇到过了，我们就从这里开始讲起。

++与—运算符

自增和自减运算符来源于C语言操作符，它们的作用很直观，是对一个变量进行加1或减一的操作

var i = 0; i++; ++I; i--; --i

然而，在我们决定使用哪种方法来实现时，会很难抉择。无论是自增还是自减方法，它们都有两种写法：前缀写法和后缀写法，它们的函数的底层实现都是存在返回值的，对于返回值的使用取决于你是否对运算符进行了重载。

这个会对初学者造成学习上的困难，因此我们移除了该使用方法，使用复合加法运算符（+=）和复合减法运算符（-=）来替代它们

var i = 0; i += 1; i -= 1

当然了，虽然复合运算符看起来更短，更便捷一些，但是你还是可以继续使用加法（+）和减法（-）运算符，

i = i + 1; i = i – 1

扩展阅读：如果你想要了解更多的这个改变背后的故事，请阅读Chris Lattner 对移除 ++和—操作符的看法。

C语言类型的for循环已经成为历史

自增与自减运算符用的最多的地方，还是在C语言风格的for循环当中，在移除了这些操作符的同时，也意味着C语言类型的for循环的特性也随之而去了，因为当你在使用for-in循环当时，循环控制语句和范围限制根本用不上这些操作符。

如果你有一定的编程经验，你大概已经写出了如下能够输出1到10的for循环了

for (i = 1; i <= 10; i++) { print(i)}

在Swift 3当中，已经不允许这样写了，它应该是这样(注意闭区间控制符的写法)

for i in 1...10 { print(i) }

或者，你可以使用for-each循环来实现（更多有关于循环的信息请看这）

(1...10).forEach { print($0)}

扩展阅读：如果你想了解更多的关于这个改变背后的故事，请阅读 Erica Sadun 对移除 C 风格循环的看法。

移除函数参数的用var修饰的标记

当我们不需要在函数体内对参数进行修改的话，通常会将参数声明为常量。然而在某些情况下，定义为变量会更加合适。在Swift 2当中，你可以利用var关键字将函数的参数标记为变量。一旦参数被定义为变量，它便会对该变量的值进行拷贝，因此你就可以在方法内部修改这些变量的值。

下面是一个求两个数最大公约数的例子（如果你还想回到高中数学课堂上再学习一遍的话，请点击这里）

func gcd(var a: Int, var b: Int) -> Int {

if (a == b) { return a }

repeat {

if (a > b) { a = a – b }

else { b = b – a}

} while (a != b)

return a

}

这个算法的逻辑十分简单：如果这两个数已经相等，则返回其中的一个值，否则，进行大小比较，用大的数减去小的数，将运算的结果复制给较大的数，然后再将两个数进行比较，直到两个数相等为止，最后返回其中一个值。正如你所看到的，通过将a和b标记为变量，我们才可以在方法内改变它们的值。

在Swift 3当中已经不允许开发者再将参数设置为变量了，因为开发者也许会对于选择var和inout产生困惑。因此在最新的版本当中，已经移除了对函数的参数标记var的特性。

因此，如果你还想在Swift 3当中书写上面的gcd函数的时候，你就要选择其他的办法了。你可以在函数内部声明一个临时变量来存储函数参数的值。

扩展阅读：如果你想要了解更多的关于该改变背后故事，你可以去参考决定移除var的想法

函数形式参数的一致性

函数在参数列表底层的实现方法中实际上就是元组，只要元组的结构和函数参数列表的结构相同，你便可以通过调用函数来使用它们。就拿上面的gcd（）函数来作为例子，你可以就像如下的方法来调用它:

gcd(8, b: 12)

或者你也可以通过如下的方式来调用:

let number = (8, b: 12), gcd(number)

正如你所见，在Swift 2当中，当调用函数的时候，你不需要为第一个参数带上形式参数。但是，从第二个参数开始，你就必须要带上形式参数了。

这个语法会给初学者造成很大的困扰，因此，它需要统一规范形式参数标签。在Swift 3当中，你需要向下面这样调用函数：

gcd(a: 8, b: 12)

即使是第一个参数，你也必须为其带上标签。如果你不这样做的话，在Xcode 8中会直接报错。

你对于这个修改的第一个反应可能会是“我的天，这得在我的项目中改动多少代码啊。”是的，没有错，你所面对的就是成吨的改变。还好，苹果提供了另外一个解决的方案，便是在第一个参数前加一个下划线,就像下面这样：

func gcd(\_ a: Int, b: Int) -> Int {…}

通过这样做，你可以继续像以前一样不用为第一个参数带上标签的方法调用函数。使用这个方法，可以使你从Swift 2转换到 Swift 3稍微轻松一些。

扩展阅读：如果你想了解关于这项改动背后更多的故事，你可以去参考this proposal。

Selectors中不再允许使用String

让我们去创建一个button并为它添加点击事件，让它去做一些事情——使用playgrounds就行，不需要使用带有界面的：

import UIKit

import XCPlayground

class Responder: NSObject {

func tap() { print("Button pressed") }}

let responder = Responder()

let button = UIButton(type: .System)

button.setTitle("Button", forState: .Normal)

button.addTarget(responder, action: "tap", forControlEvents: .TouchUpInside)

button.sizeToFit(); button.center = CGPoint(x: 50, y: 25)

let frame = CGRect(x: 0, y: 0, width: 100, height: 50)

let view = UIView(frame: frame); view.addSubview(button)

XCPlaygroundPage.currentPage.liveView = view

这其中确实有很多的步骤，让我们来一步一步分析：

1. 导入UIKit和XCPlayground框架，你需要使用它们来创建一个button并在playground的assistant editor中展示

注意：你需要在Xcode的菜单栏中点击View -> Assistant Editor -> Show Assistant Editor.

1. 定义按钮的点击事件，使用户在点击按钮的时候，能够触发点击事件——这需要继承基类NSObject，因为selectors只对于Objective-C的方法有效
2. 声明按钮并设置它的属性
3. 声明view并给定其合适的frame，然后将button添加上去并在playground的assistant editor上去展示它

让我们看下给按钮添加点击事件的代码，

 button.addTarget(responder, action: "tap", forControlEvents: .TouchUpInside)

按钮的selector是一个字符串，假如你的字符串书写错误了，程序依然会通过编译，但是会在运行时崩溃，因为并没有找到相关的方法来调用。

为了解决在编译时的潜在问题，在Swift 3当中将字符串类型的selectors替换为了#selector（）关键字，这样便可以在编译时就可以让编译器去判断是否获取到正确的方法名，而不用等到运行时。

button.addTarget(responder, action: #selector(Responder.tap), for: .touchUpInside)

扩展阅读：如果想了解更多的关于这个改变的背后的故事，你可以去参考Doug Gregor’s proposal

以上就是关于在Swift 3中移除的特性的全部内容。现在，让我们来看看语言现代化的亮点吧。

String的Key-path的用法

这个特性和之前的那一个很相似，但是这个是用在键值编码（KVC）和键值观察（KVO）上的：

class Person: NSObject {

var name: String = ""; init(name: String) { self.name = name}}

let me = Person(name: "Cosmin"); me.valueForKeyPath("name")

首先你要创建一个类。例如Person类，这一步是实现KVC的首要条件，然后再使用构造方法创建出一个me对象，并利用KVC对其属性name进行修改。假如在书写过程中出现了错误，那么一切准备工作将算是白做了🙁。

幸运得是，这种情况将不会在Swift 3中出现了，String中的Key-path方法由具有同样作用的#keypath来代替了：

class Person: NSObject {

var name: String = ""; init(name: String) { self.name = name}}

let me = Person(name: "Cosmin")

me.value(forKeyPath: #keyPath(Person.name))

扩展阅读:如果你想了解更多的关于这个改变背后的故事，你可以去参考David Hart’s proposal.

移除了Foundation框架中的NS前缀

关于NS前缀已经从Swift 3中移除了，如果你还对之前的写法感兴趣，你可以来参考这里，下面是一个典型的解析JSON的例子

let file = NSBundle.mainBundle().pathForResource("tutorials", ofType: "json")

let url = NSURL(fileURLWithPath: file!)

let data = NSData(contentsOfURL: url)

let json = try! NSJSONSerialization.JSONObjectWithData(data!, options: []),

print(json)

上述代码使用了Foundation框架中的某些类来完成了对JSON数据的解析：NSBundle -> NSURL -> NSData -> NSJSONSerialization.

在Swift 3中，NS前缀已经被移除了，因此，总的来说，现在的解析过程已经变成了：Bundle -> URL -> Data -> JSONSerialization:

let file = Bundle.main().pathForResource("tutorials", ofType: "json")

let url = URL(fileURLWithPath: file!)

let data = try! Data(contentsOf: url)

let json = try! JSONSerialization.jsonObject(with: data); print(json)

扩展阅读：如果你想了解更多的关于这个改变背后的故事，你可以去参考Tony Parker和Philippe Hausler对这件事的看法。

M\_PI和pi\_

让我们来看一个知道已知半径来求圆周长的例子：

let r = 3.0, let circumference = 2 \* M\_PI \* r, let area = M\_PI \* r \* r

在旧版本的Swift中，你会用使用M\_PI\_来表示π这个常量，在Swift 3中π分别有Float，Double，CGFloat三种表现形式：

Float.pi, Double.pi, CGFloat.pi

在以上代码在swift 3中，应该这样

let r = 3.0; let circumference = 2 \* Double.pi \* r

let area = Double.pi \* r \* r

根据类型推断，我们可以将类型的前缀移除，使用更为精简的方式：

let r = 3.0; let circumference = 2 \* .pi \* r; let area = .pi \* r \* r

Grand Central Dispatch(GCD)

Grand Central Dispatch(GCD)常被用于网络请求当中，因为它可以解决因主线程阻塞，UI不能j及时刷新的问题。它是用C语言来实现的，而且它的API对于初学者来说也很难理解，甚至创建一个基本的异步队列就要像下面这样写：

let queue = dispatch\_queue\_create("Swift 2.2", nil)

dispatch\_async(queue) {  print("Swift 2.2 queue")}

在Swift 3中移除了这些冗余的代码，并使得写法更加趋于面向对象

let queue =DispatchQueue(label:"Swift3")

queue.async { print("Swift 3 queue")}

扩展阅读：如果你想了解更多的关于这个改变背后的故事，你可以去参考Matt Wright对这个改变的看法

更加Swift化的Core Graphics

Core Graphics是一个十分强大的绘图框架，但是和GCD一样，同样的也是C语言风格的API：

let frame = CGRect(x: 0, y: 0, width: 100, height: 50)

class View: UIView {

override func drawRect(rect: CGRect) {

let context = UIGraphicsGetCurrentContext()

let blue = UIColor.blueColor().CGColor

CGContextSetFillColorWithColor(context, blue)

let red = UIColor.redColor().CGColor

CGContextSetStrokeColorWithColor(context, red)

CGContextSetLineWidth(context, 10)

CGContextAddRect(context, frame)

CGContextDrawPath(context, .FillStroke)}}

let aView = View(frame: frame)

在上面的代码当中，你创建了一个view的frame，然后创建了一个继承于UIView的类，并重写了drawRect（）方法来重绘view的内容。

在Swift 3中，具有完全不同的实现方法—-首先对当前的上下文进行解包，之后便可以进行你所需要的重绘操作了：

let frame = CGRect(x: 0, y: 0, width: 100, height: 50)

class View: UIView {

override func draw(\_ rect: CGRect) {

guard let context = UIGraphicsGetCurrentContext() else {return}

let blue = UIColor.blue().cgColor, context.setFillColor(blue)

let red = UIColor.red().cgColor; context.setStrokeColor(red)

context.setLineWidth(10)

context.addRect(frame)

context.draw Path(using: .fillStroke)}}

let aView = View(frame: frame)

注意：当在view调用drawRect（）方法之前，context为nil，所以你就要使用guard语句来进行处理，更多内容请点击这里。

@discardableResult

在Swift 3中，你如果没有接受方法或函数的返回值，Xcode将会报警告。如下：



在上面的代码当中，printMessage方法会返回给调用者一条信息。但是，这个返回值并没有被使用。这可能会存在潜在的问题，所以编译器会在Swift 3给一个警告。

在这种情况下，并不一定非得要接受返回值，并对返回值进行处理。你可以使用@discardableResult关键字对方法进行声明，以此来消除警告：

override func viewDidLoad() {

super.viewDidLoad(); printMessage(message: "Hello Swift 3!")}

@discardableResult

func printMessage(message: String) -> String {

let outputMessage = "Output : \(message)"

print(outputMessage); return outputMessage}

总结

这些便是Swift 3中所有的改动。新版本Swift的发布，使得Swift这门语言变得越来越好，越来越优雅。同样的，它可能也包含了许多对你现有代码造成影响的改动。希望这篇教程可以帮助你更好地理解这些改变，同时也希望为你迁移swift项目节省大量的时间。

教程当中的所有代码你都可以在this Playground project中找到。所有代码我也已在Xcode 8测试版中进行了测试。所以，请确保你的项目的运行环境为Xcode 8.

如果你还有其他的问题和困惑。欢迎你来告诉我。Happy coding!