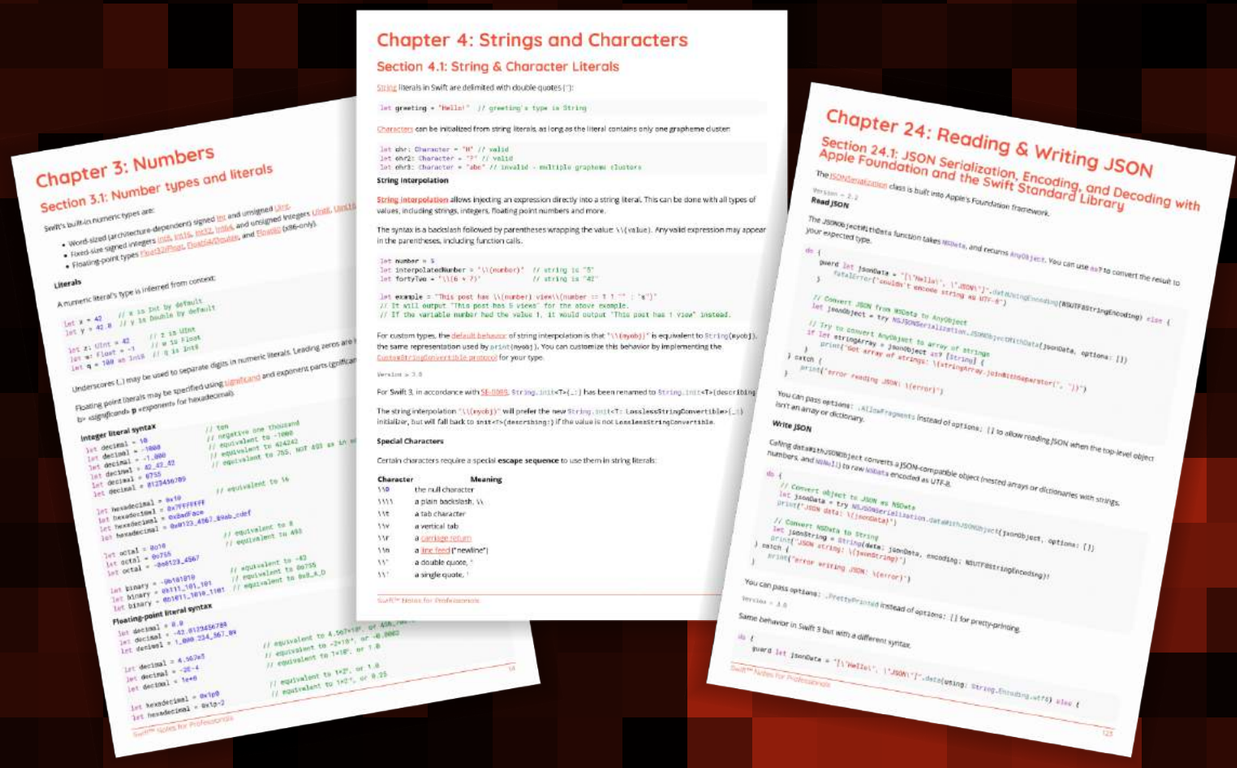
专业人士笔记

Swift™
专业人士笔记



200+ 页

专业提示和技巧

GoalKicker.com
免费编程书籍

免责声明

这是一本非官方的免费书籍，旨在教育用途，
与官方Swift™组织或公司无关。
所有商标和注册商标均为其各自所有者的财产

Swift™
Notes for Professionals

200+ pages

of professional hints and tricks

GoalKicker.com
Free Programming Books

Disclaimer

This is an unofficial free book created for educational purposes and is not affiliated with official Swift™ group(s) or company(s).
All trademarks and registered trademarks are the property of their respective owners

目录

Contents

关于

请随意免费与任何人分享此PDF，
本书的最新版本可从以下网址下载：
https://goalkicker.com/SwiftBook

本*Swift™ 专业笔记*一书汇编自Stack Overflow Documentation，内容由Stack Overflow的优秀人士撰写。
文本内容采用知识共享署名-相同方式共享许可发布，详见本书末尾对各章节贡献者的致谢。图片版权归各自所有者所有，除非另有说明。

本书为非官方免费书籍，旨在教育用途，与官方Swift™组织或公司及Stack Overflow无关。所有商标和注册商标均为其各自公司所有者所有。

本书所提供的信息不保证正确或准确，使用风险自负。

请将反馈和更正发送至web@petercv.com

About

Please feel free to share this PDF with anyone for free,
latest version of this book can be downloaded from:
https://goalkicker.com/SwiftBook

This *Swift™ Notes for Professionals* book is compiled from Stack Overflow Documentation, the content is written by the beautiful people at Stack Overflow. Text content is released under Creative Commons BY-SA, see credits at the end of this book whom contributed to the various chapters. Images may be copyright of their respective owners unless otherwise specified

This is an unofficial free book created for educational purposes and is not affiliated with official Swift™ group(s) or company(s) nor Stack Overflow. All trademarks and registered trademarks are the property of their respective company owners

The information presented in this book is not guaranteed to be correct nor accurate, use at your own risk

Please send feedback and corrections to web@petercv.com

第1章：Swift语言入门

Swift版本	Xcode版本	发布日期
开发开始时间（首次提交）	-	2010-07-17
1.0	Xcode 6	2014-06-02
1.1	Xcode 6.1	2014-10-16
1.2	Xcode 6.3	2015-02-09
2.0	Xcode 7	2015-06-08
2.1	Xcode 7.1	2015-09-23
开源发布	-	2015-12-03
2.2	Xcode 7.3	2016-03-21
2.3	Xcode 8	2016-09-13
3.0	Xcode 8	2016-09-13
3.1	Xcode 8.3	2017-03-27
4.0	Xcode 9	2017-11-19
4.1	Xcode 9.3	2018-03-29

第1.1节：你的第一个Swift程序

在名为hello.swift的文件中编写代码：

```
print("Hello, world!")
```

- 要一步编译并运行脚本，请在终端中使用swift（在包含该文件的目录下）：

> 要打开终端，请按 CTRL+ALT+T 在Linux上，或在macOS的Launchpad中找到它。要更改目录，输入cd目录名（或cd ..返回上一级目录）

```
$ swift hello.swift
Hello, world!
```

> 编译器（compiler）是一种计算机程序（或一组程序），它将用编程语言（源语言）编写的源代码转换为另一种计算机语言（目标语言），后者通常具有称为目标代码的二进制形式。（Wikipedia）

- 要单独编译和运行，请使用swiftc：

```
$ swiftc hello.swift
```

这将把你的代码编译成hello文件。要运行它，输入./，后跟文件名。

```
$ ./hello
Hello, world!
```

- 或者使用swift REPL（读-评估-打印循环），在命令行输入swift，然后在解释器中输入你的代码：

Chapter 1: Getting started with Swift Language

Swift Version	Xcode Version	Release Date
development began (first commit)	-	2010-07-17
1.0	Xcode 6	2014-06-02
1.1	Xcode 6.1	2014-10-16
1.2	Xcode 6.3	2015-02-09
2.0	Xcode 7	2015-06-08
2.1	Xcode 7.1	2015-09-23
open-source debut	-	2015-12-03
2.2	Xcode 7.3	2016-03-21
2.3	Xcode 8	2016-09-13
3.0	Xcode 8	2016-09-13
3.1	Xcode 8.3	2017-03-27
4.0	Xcode 9	2017-11-19
4.1	Xcode 9.3	2018-03-29

Section 1.1: Your first Swift program

Write your code in a file named `hello.swift`:

```
print("Hello, world!")
```

- To compile and run a script in one step, use `swift` from the terminal (in a directory where this file is located):

> To launch a terminal, press CTRL+ALT+T on *Linux*, or find it in Launchpad on *macOS*. To change directory, enter `cd`**`directory_name`** (or `cd ..` to go back)

```
$ swift hello.swift
Hello, world!
```

> A **compiler** is a computer program (or a set of programs) that transforms source code written in a programming language (the source language) into another computer language (the target language), with the latter often having a binary form known as object code. (Wikipedia)

- To compile and run separately, use `swiftc`:

```
$ swiftc hello.swift
```

This will compile your code into `hello` file. To run it, enter `./`, followed by a filename.

```
$ ./hello
Hello, world!
```

- Or use the swift REPL (Read-Eval-Print-Loop), by typing `swift` from the command line, then entering your code in the interpreter:

代码：

```
func greet(name: String, surname: String) {
    print("Greetings \(name) \(surname)")
}

let myName = "Homer"
let mySurname = "Simpson"

greet(name: myName, surname: mySurname)
```

> **让我们把这段大代码拆分成几部分：**
>
> - `func greet(name: String, surname: String) { // 函数体 }` - 创建一个函数，接收一个名字和一个姓氏。
> - `print("Greetings \(name) \(surname)")` - 这会在控制台打印“Greetings ”，然后是名字，接着是姓氏。基本上`\(变量名)`会打印该变量的值。
> - `let myName = "Homer"` 和 `let mySurname = "Simpson"` - 使用let创建常量（值不可更改的变量），名称分别为：myName、mySurname，值分别为："Homer"、"Simpson"。
> - `greet(name: myName, surname: mySurname)` - 调用之前创建的函数，传入常量myName和mySurname的值。

使用 REPL 运行：

```
$  swift
欢迎使用 Apple Swift。输入:help获取帮助。
  1> func greet(name: String, surname: String) {
  2.     print("Greetings \(name) \(surname)")
  3. }
  4>
  5> let myName = "Homer"
myName: String = "Homer"
  6> let mySurname = "Simpson"
mySurname: String = "Simpson"
  7> greet(name: myName, surname: mySurname)
Greetings Homer Simpson
  8> ^D
```

按下 CTRL + D 退出 REPL。

第1.2节：在Mac上使用Swift编写你的第一个程序（使用Playground）

在你的Mac上，通过此链接从Mac App Store下载并安装Xcode。

安装完成后，打开Xcode并选择开始使用Playground：

Code:

```
func greet(name: String, surname: String) {
    print("Greetings \(name) \(surname)")
}

let myName = "Homer"
let mySurname = "Simpson"

greet(name: myName, surname: mySurname)
```

> **Let's break this large code into pieces:**
>
> - `func greet(name: String, surname: String) { // function body }` - create a *function* that takes a `name` and a `surname`.
> - `print("Greetings \(name) \(surname)")` - This prints out to the console "Greetings ", then `name`, then `surname`. Basically `\(variable_name)` prints out that variable's value.
> - `let myName = "Homer"` and `let mySurname = "Simpson"` - create *constants* (variables which value you can't change) using `let` with names: `myName`, `mySurname` and values: `"Homer"`, `"Simpson"` respectively.
> - `greet(name: myName, surname: mySurname)` - calls a *function* that we created earlier supplying the values of *constants* `myName`, `mySurname`.

Running it using REPL:

```
$  swift
Welcome to Apple Swift. Type :help for assistance.
  1> func greet(name: String, surname: String) {
  2.     print("Greetings \(name) \(surname)")
  3. }
  4>
  5> let myName = "Homer"
myName: String = "Homer"
  6> let mySurname = "Simpson"
mySurname: String = "Simpson"
  7> greet(name: myName, surname: mySurname)
Greetings Homer Simpson
  8> ^D
```

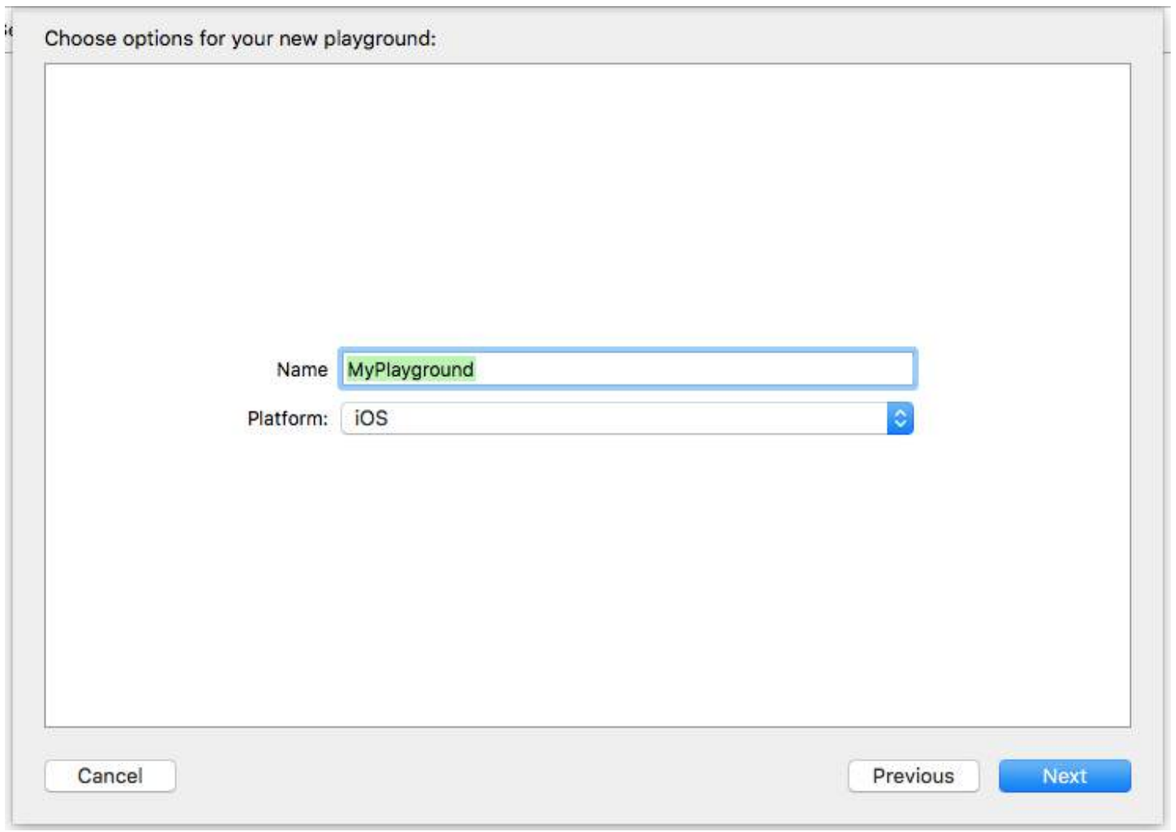
Press CTRL + D to quit from REPL.

Section 1.2: Your first program in Swift on a Mac (using a Playground)

From your Mac, download and install Xcode from the Mac App Store following this link.

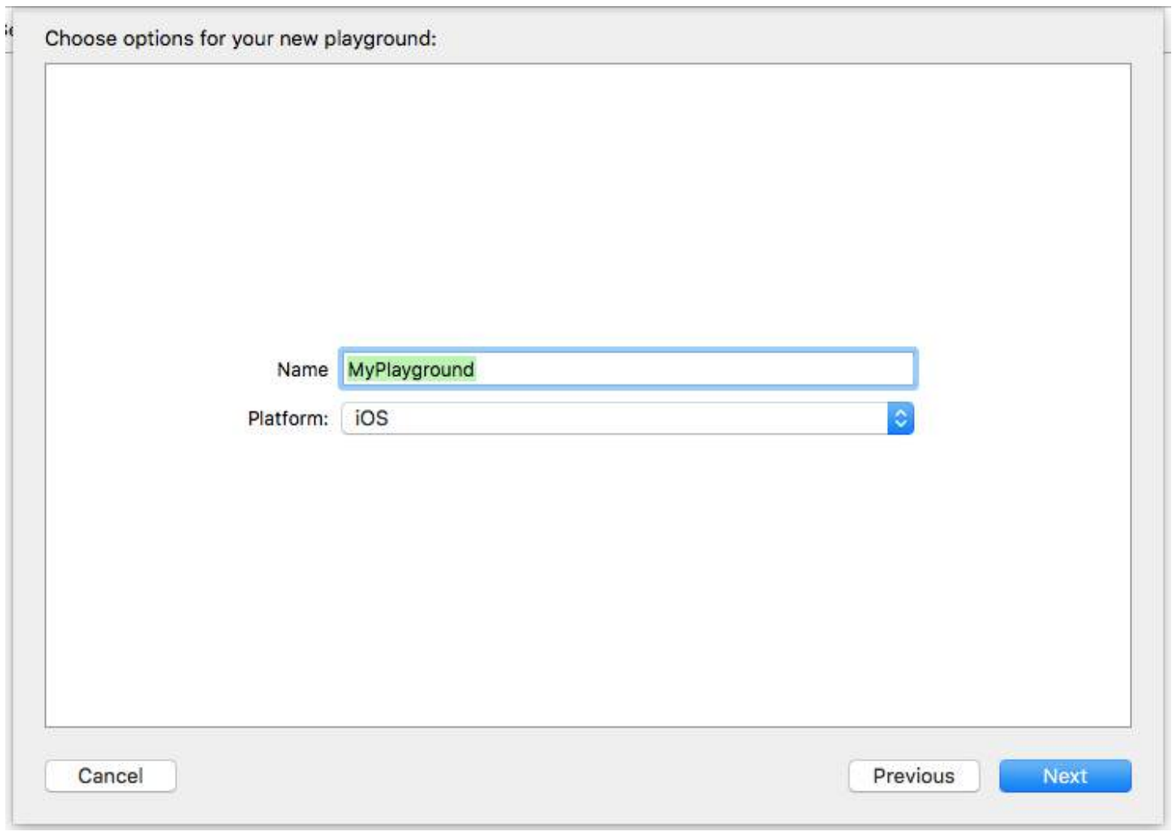
After the installation is complete, open Xcode and select **Get started with a Playground**:

在下一个面板中，你可以为你的Playground命名，或者保持默认的MyPlayground，然后点击下一步：

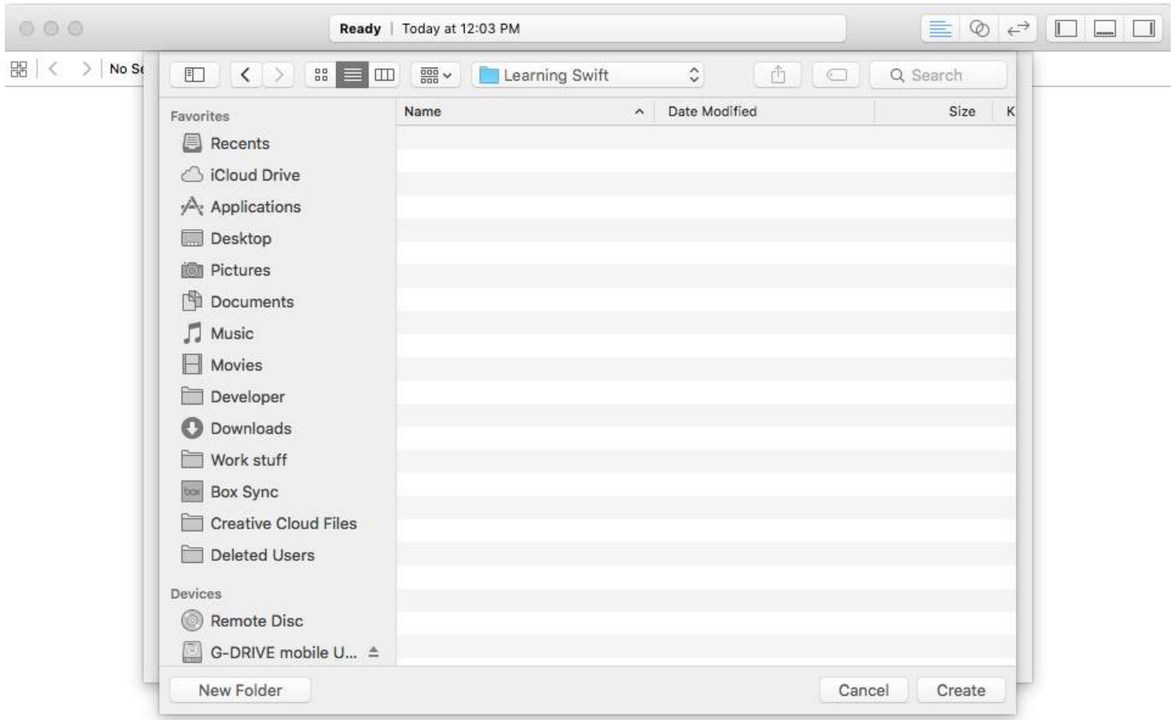


选择保存Playground的位置，然后点击创建：

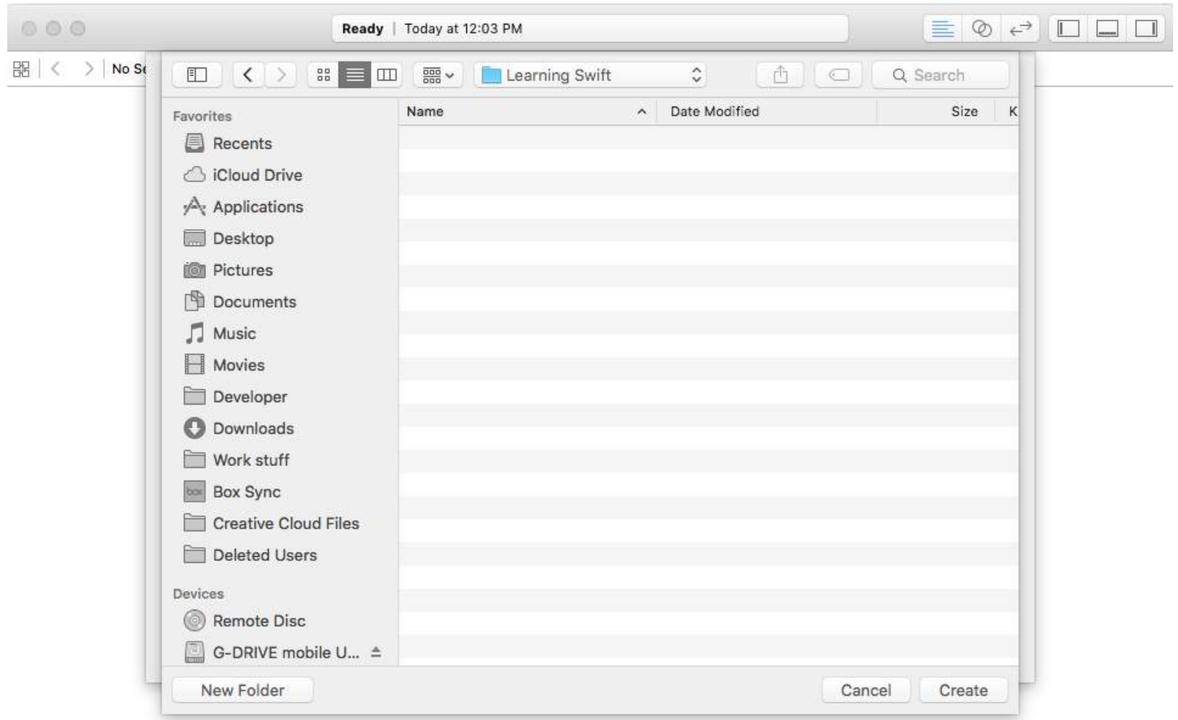
On the next panel, you can give your Playground a name or you can leave it `MyPlayground` and press **Next**:



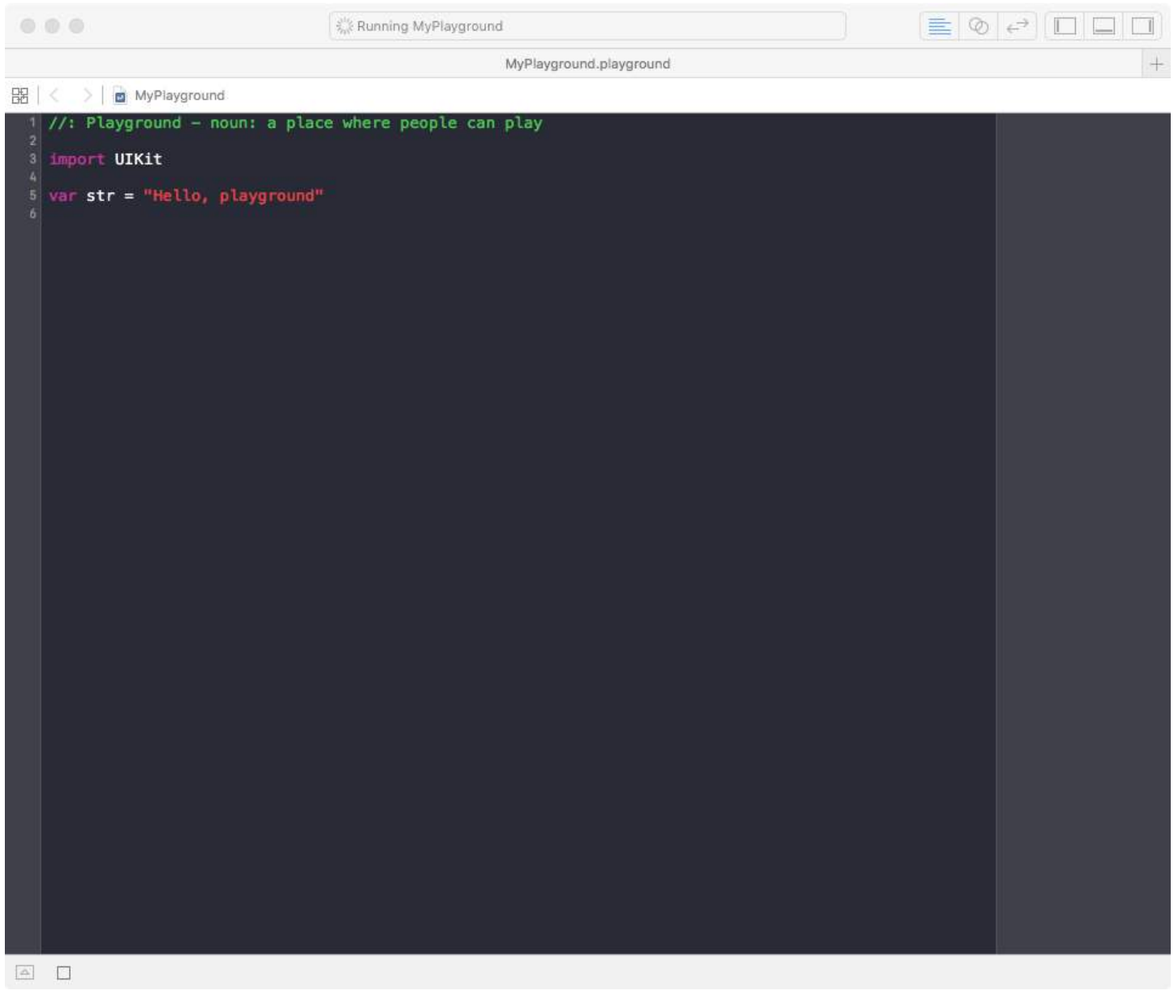
Select a location where to save the Playground and press **Create**:



Playground将打开，你的屏幕应类似如下：



The Playground will open and your screen should look something like this:

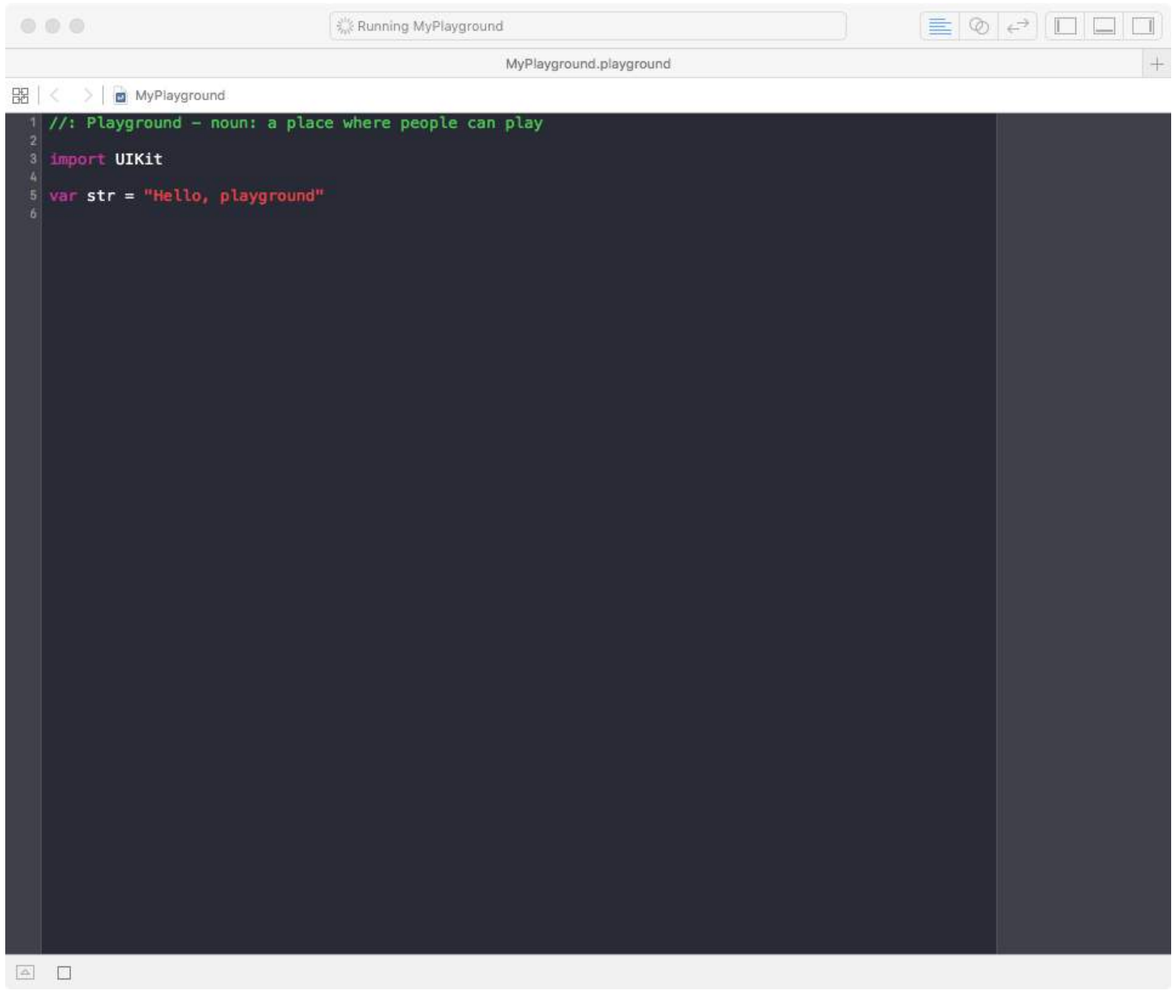


现在游乐场已经显示在屏幕上，按下 ⇧ + cmd + Y 显示调试区域。

最后删除Playground中的文本并输入：

```
print("Hello world")
```

你应该能在调试区域看到'Hello world'，并在右侧侧边栏看到"Hello world"：

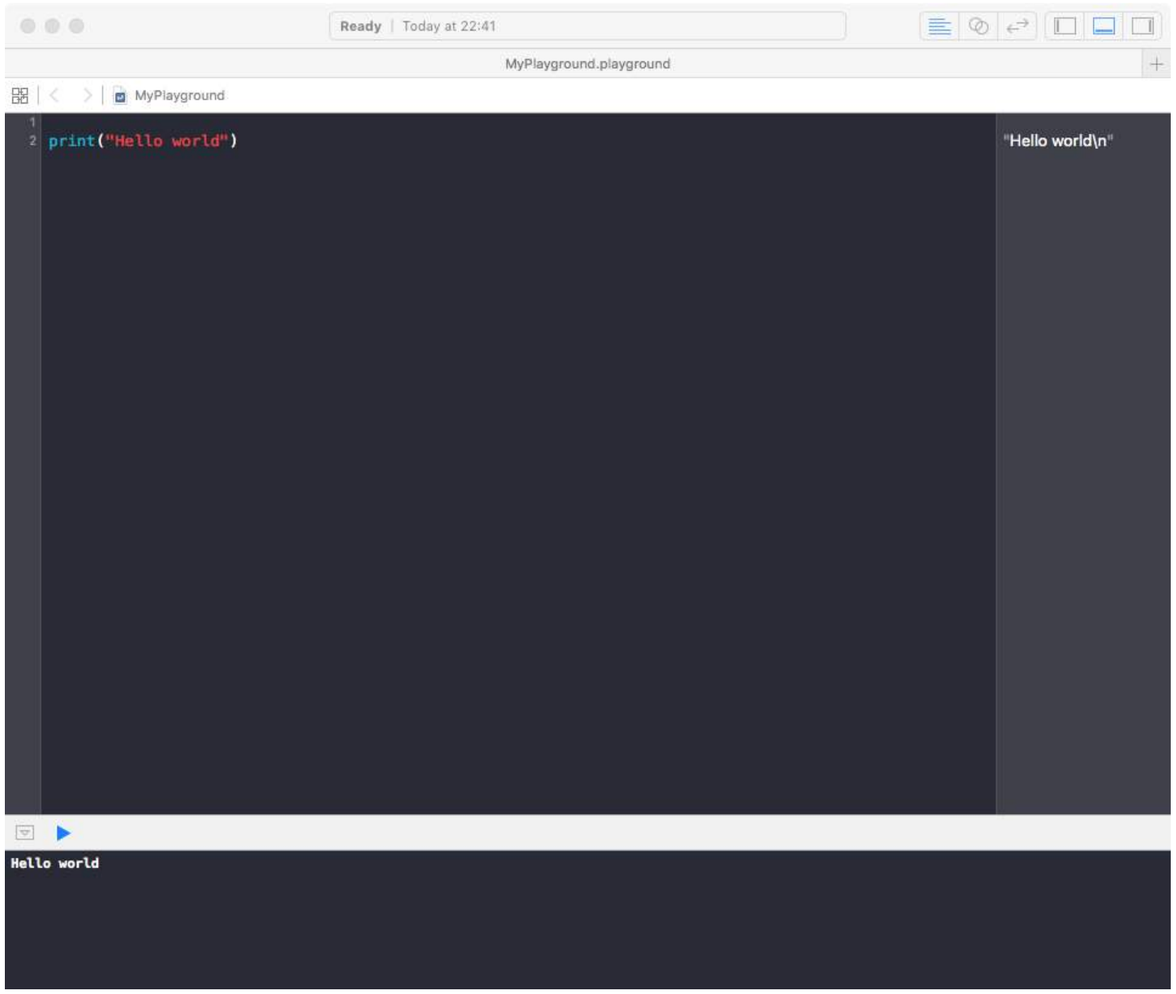


Now that the Playground is on the screen, press ⇧ + cmd + Y to show the **Debug Area**.

Finally delete the text inside Playground and type:

```
print("Hello world")
```

You should see 'Hello world' in the **Debug Area** and "Hello world\n" in the right **Sidebar**:

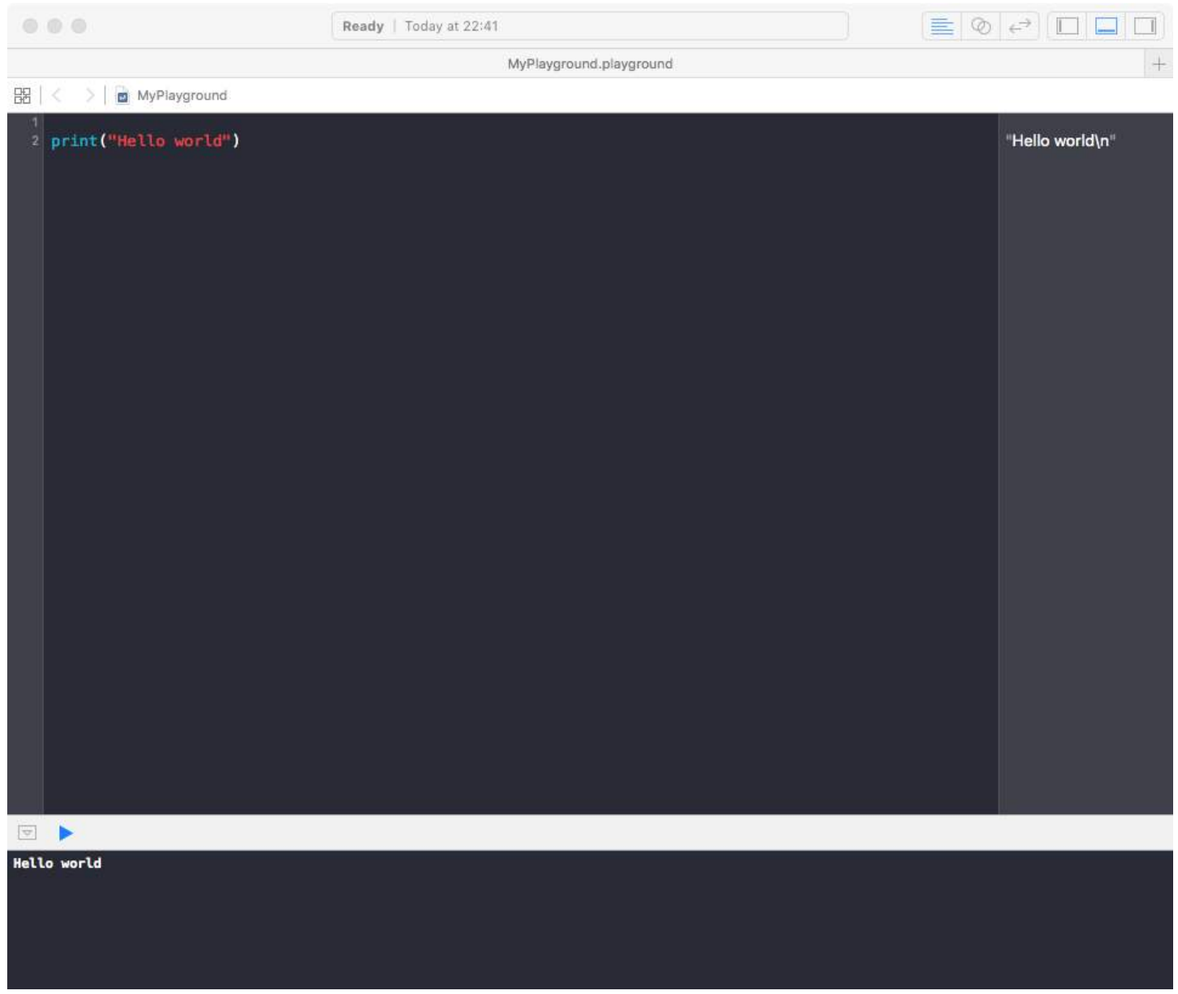


恭喜！你已经创建了你的第一个Swift程序！

第1.3节：在iPad上的Swift Playgrounds应用中创建你的第一个程序

Swift Playgrounds 应用是随时随地开始学习 Swift 编程的好方法。使用方法如下：

1- 从 App Store 下载Swift Playgrounds（适用于 iPad）。

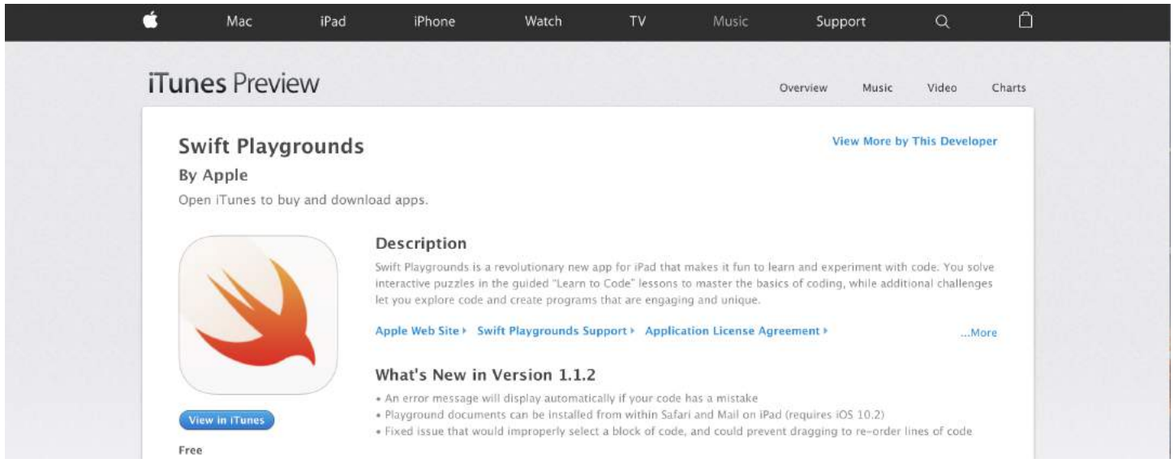


Congratulations! You've created your first program in Swift!

Section 1.3: Your first program in Swift Playgrounds app on iPad

Swift Playgrounds app is a great way to get started coding Swift on the go. To use it:

1- Download Swift Playgrounds for iPad from App Store.



2- 打开应用。

3- 在我的游乐场标签页，点击左上角的+，然后选择空白模板。

4- 输入你的代码。

5- 点击“运行我的代码”来执行代码。

6- 每行代码前会有一个小方块存储结果，点击它即可查看结果。

7- 若要逐步执行代码以便跟踪，点击“运行我的代码”旁边的按钮。

第1.4节：安装 Swift

首先，下载编译器和组件。

接下来，将 Swift 添加到你的路径中。在 macOS 上，可下载工具链的默认位置是 /Library/Developer/Toolchains。请在终端中运行以下命令：

```
export PATH=/Library/Developer/Toolchains/swift-latest.xctoolchain/usr/bin:"${PATH}"
```

在 Linux 上，您需要安装 clang：

```
$ sudo apt-get install clang
```

如果您将 Swift 工具链安装到了系统根目录以外的目录，您需要运行以下命令，使用您实际的 Swift 安装路径：

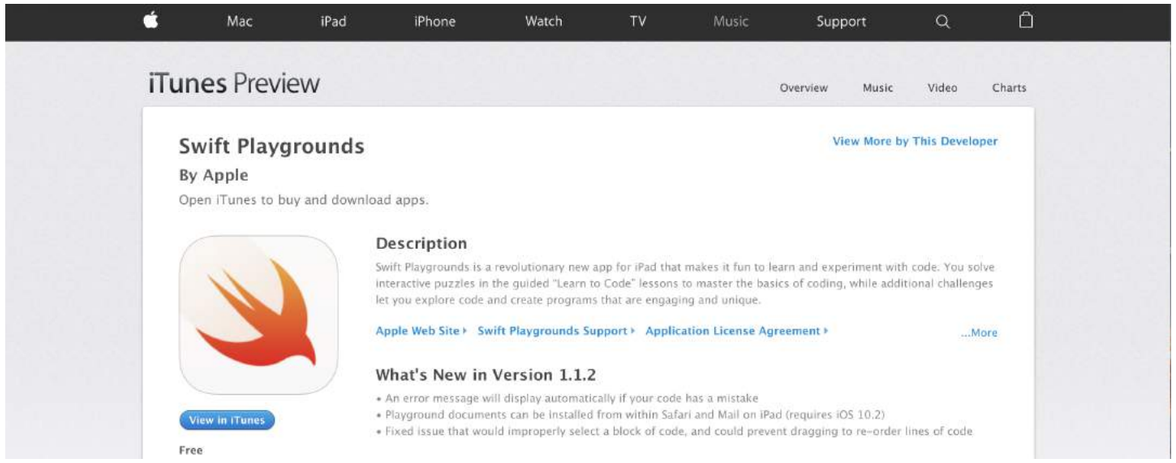
```
$ export PATH=/path/to/Swift/usr/bin:"${PATH}"
```

您可以通过运行此命令来验证您是否拥有当前版本的 Swift：

```
$ swift --version
```

第 1.5 节：可选值和 Optional 枚举

Optionals 类型，用于处理值的缺失。Optionals 表示“有一个值，且该值等于 x”或者“根本没有值”。



2- Open the app.

3- In the **My Playgrounds** tab, tap **+** on the top left corner and then select Blank template.

4- Enter your code.

5- Tap Run My Code to run your code.

6- At the front of each line, the result will be stored in a small square. Tap it to reveal the result.

7- To step slowly through code to trace it, tap the button next to Run My Code.

Section 1.4: Installing Swift

First, download the compiler and components.

Next, add Swift to your path. On macOS, the default location for the downloadable toolchain is /Library/Developer/Toolchains. Run the following command in Terminal:

```
export PATH=/Library/Developer/Toolchains/swift-latest.xctoolchain/usr/bin:"${PATH}"
```

On Linux, you will need to install clang:

```
$ sudo apt-get install clang
```

If you installed the Swift toolchain to a directory other than the system root, you will need to run the following command, using the actual path of your Swift installation:

```
$ export PATH=/path/to/Swift/usr/bin:"${PATH}"
```

You can verify you have the current version of Swift by running this command:

```
$ swift --version
```

Section 1.5: Optional Value and Optional enum

Optionals type, which handles the absence of a value. Optionals say either "there is a value, and it equals x" or "there isn't a value at all".

Optional 是一种独立的类型，实际上是 Swift 新增的超级枚举之一。它有两个可能的值，None 和 Some(T)，其中 T 是 Swift 中可用的正确数据类型的关联值。

让我们来看下面这段代码作为例子：

```
let x: String? = "Hello World"

if let y = x {
   print(y)
}
```

事实上，如果你在上面的代码中添加一条 print(x.dynamicType) 语句，你将在控制台看到：

```
Optional<String>
```

String? 实际上是 Optional 的语法糖，而 Optional 本身是一个独立的类型。

下面是 Optional 头文件的简化版本，你可以在 Xcode 中通过按住 Command 并点击代码中的 Optional 一词查看：

```
enum Optional<Wrapped> {

 /// 表示值的缺失。
 无案例

 /// 表示存在一个值，存储为 `Wrapped`。
 case some(Wrapped)
}
```

Optional 实际上是一个枚举，定义时关联一个泛型类型 Wrapped。它有两个情况：.none 表示值的缺失，和 .some 表示值的存在，该值作为其关联值存储，类型为Wrapped。

让我再说一遍：String? 不是一个 String，而是一个 Optional<String>。因为 Optional 是一个类型，所以它有自己的方法，例如 `map` 和 `flatMap`。

An Optional is a type on its own, actually one of Swift's new super-powered enums. It has two possible values, `None` and `Some(T)`, where T is an associated value of the correct data type available in Swift.

Let's have a look at this piece of code for example:

```
let x: String? = "Hello World"

if let y = x {
   print(y)
}
```

In fact if you add a **`print(x.dynamicType)`** statement in the code above you'll see this in the console:

```
Optional<String>
```

String? is actually syntactic sugar for Optional, and Optional is a type in its own right.

Here's a simplified version of the header of Optional, which you can see by command-clicking on the word Optional in your code from Xcode:

```
enum Optional<Wrapped> {

 /// The absence of a value.
 case none

 /// The presence of a value, stored as `Wrapped`.
 case some(Wrapped)
}
```

Optional is actually an enum, defined in relation to a generic type Wrapped. It has two cases: *`.none`* to represent the absence of a value, and *`.some`* to represent the presence of a value, which is stored as its associated value of type Wrapped.

Let me go through it again: `String`? is not a `String` but an `Optional<String>`.The fact that `Optional` is a type means that it has its own methods, for example `map` and `flatMap`.

第2章：变量与属性

第2.1节：创建变量

使用 var 声明一个新变量，后跟名称、类型和值：

```
var num: Int = 10
```

变量的值可以被修改：

```
num = 20 // num 现在等于20
```

除非它们用let定义：

```
let num: Int = 10 // num 不能改变
```

Swift 会推断变量类型，所以你不必总是声明变量类型：

```
let ten = 10 // num 是 Int
let pi = 3.14 // pi 是 Double
let floatPi: Float = 3.14 // floatPi 是 Float
```

变量名不限于字母和数字——它们还可以包含大多数其他 Unicode 字符，
尽管有一些限制

> 常量和变量名不能包含空白字符、数学符号、箭头、私用（或无效）Unicode 代码点，或线条和框线字符。它们也不能以数字开头
>
> 来源 developer.apple.com

```
var π: Double = 3.14159
var 🐶🐮: String = "Apples"
```

第2.2节：属性观察器

属性观察者响应属性值的变化。

```
var myProperty = 5 {
    willSet {
        print("将要设置为 \(newValue)。之前的值是 \(myProperty)")
    }
    didSet {
        print("已设置为 \(myProperty)。之前的值是 \(oldValue)")
    }
}
myProperty = 6
// 输出：将要设置为 6，之前的值是 5
// 输出：已设置为 6，之前的值是 5
```

- willSet 在 myProperty 被设置之前调用。新值可通过 newValue 获取，旧值仍然可通过 `myProperty` 获取。

Chapter 2: Variables & Properties

Section 2.1: Creating a Variable

Declare a new variable with `var`, followed by a name, type, and value:

```
var num: Int = 10
```

Variables can have their values changed:

```
num = 20 // num now equals 20
```

Unless they're defined with `let`:

```
let num: Int = 10 // num cannot change
```

Swift infers the type of variable, so you don't always have to declare variable type:

```
let ten = 10 // num is an Int
let pi = 3.14 // pi is a Double
let floatPi: Float = 3.14 // floatPi is a Float
```

Variable names aren't restricted to letters and numbers - they can also contain most other unicode characters, although there are some restrictions

> Constant and variable names cannot contain whitespace characters, mathematical symbols, arrows, private-use (or invalid) Unicode code points, or line- and box-drawing characters. Nor can they begin with a number
>
> **Source** developer.apple.com

```
var π: Double = 3.14159
var 🐶🐮: String = "Apples"
```

Section 2.2: Property Observers

Property observers respond to changes to a property's value.

```
var myProperty = 5 {
    willSet {
        print("Will set to \(newValue). It was previously \(myProperty)")
    }
    didSet {
        print("Did set to \(myProperty). It was previously \(oldValue)")
    }
}
myProperty = 6
// prints: Will set to 6, It was previously 5
// prints: Did set to 6. It was previously 5
```

- `willSet` is called **before** `myProperty` is set. The new value is available as `newValue`, and the old value is still available as `myProperty`.

- didSet 在 myProperty 被设置之后调用。旧值可通过 oldValue 获取，新值现在可通过 myProperty 获取。

> 注意：didSet 和 willSet 在以下情况下不会被调用：
>
> - 赋予初始值
> - 在变量自身的 didSet 或 willSet 中修改变量

- 可以声明 didSet 和 willSet 的 oldValue 和 newValue 参数名以提高可读性：

```
var myFontSize = 10 {
    willSet(newFontSize) {
        print("将字体设置为 \(newFontSize)，之前是 \(myFontSize)")
    }
    didSet(oldFontSize) {
        print("字体已设置为 \(myFontSize)，之前是 \(oldFontSize)")
    }
}
```

> **注意：** 虽然支持声明 setter 参数名，但应谨慎避免混淆名称：
>
> - willSet(oldValue) 和 didSet(newValue) 完全合法，但会极大地混淆代码的读者。

第2.3节：惰性存储属性

延迟存储属性的值在首次访问之前不会被计算。这对于节省内存非常有用，当变量的计算开销较大时尤其如此。你可以使用lazy来声明一个延迟属性：

```
lazy var veryExpensiveVariable = expensiveMethod()
```

通常它被赋值为一个闭包的返回值：

```
lazy var veryExpensiveString = { () -> String in
    var str = expensiveStrFetch()
str.expensiveManipulation(integer: arc4random_uniform(5))
    return str
}()
```

延迟存储属性必须用var声明。

第2.4节：属性基础

属性可以添加到类或结构体（技术上也可以添加到枚举，参见“计算属性”示例）。这些为类/结构体的实例添加关联的值：

```
class Dog {
    var name = ""
}
```

在上述情况下，Dog 的实例具有一个名为 name 的属性，类型为 String。该属性可以被访问和

- didSet is called **after** myProperty is set. The old value is available as oldValue, and the new value is now available as myProperty .

> **Note:** didSet and willSet will not be called in the following cases:
>
> - Assigning an initial value
> - Modifying the variable within its own didSet or willSet

- The parameter names for oldValue and newValue of didSet and willSet can also be declared to increase readability:

```
var myFontSize = 10 {
    willSet(newFontSize) {
        print("Will set font to \(newFontSize), it was \(myFontSize)")
    }
    didSet(oldFontSize) {
        print("Did set font to \(myFontSize), it was \(oldFontSize)")
    }
}
```

> **Caution:** While it is supported to declare setter parameter names, one should be cautious not to mix names up:
>
> - willSet(oldValue) and didSet(newValue) are entirely legal, but will considerably confuse readers of your code.

Section 2.3: Lazy Stored Properties

Lazy stored properties have values that are not calculated until first accessed. This is useful for memory saving when the variable's calculation is computationally expensive. You declare a lazy property with lazy:

```
lazy var veryExpensiveVariable = expensiveMethod()
```

Often it is assigned to a return value of a closure:

```
lazy var veryExpensiveString = { () -> String in
    var str = expensiveStrFetch()
    str.expensiveManipulation(integer: arc4random_uniform(5))
    return str
}()
```

Lazy stored properties must be declared with var.

Section 2.4: Property Basics

Properties can be added to a class or struct (technically enums too, see "Computed Properties" example). These add values that associate with instances of classes/structs:

```
class Dog {
    var name = ""
}
```

In the above case, instances of Dog have a property named name of type String. The property can be accessed and

在 Dog 的实例上进行修改：

```
let myDog = Dog()
myDog.name = "Doggy" // myDog 的名字现在是 "Doggy"
```

这类属性被视为 **存储属性**，因为它们在对象上存储数据并影响其内存。

第2.5节：计算属性

与存储属性不同，**计算属性**由 getter 和 setter 构成，在访问和设置时执行必要的代码。计算属性必须定义类型：

```
var pi = 3.14

class Circle {
    var radius = 0.0
    var circumference: Double {
        get {
            return pi * 半径 * 2
        }
        set {
radius = newValue / pi / 2
        }
    }
}

let circle = Circle()
circle.radius = 1
print(circle.circumference) // 输出 "6.28"
circle.circumference = 14
print(circle.radius) // 输出 "2.229..."
```

只读计算属性仍然用 var 声明：

```
var circumference: Double {
    get {
        return pi * 半径 * 2
    }
}
```

只读计算属性可以简写，省略 get：

```
var circumference: Double {
    return pi * radius * 2
}
```

第2.6节：局部变量和全局变量

局部变量是在函数、方法或闭包内定义的：

```
func printSomething() {
    let localString = "我是局部的！"
    print(localString)
}

func printSomethingAgain() {
```


modified on instances of `Dog`:

```
let myDog = Dog()
myDog.name = "Doggy" // myDog's name is now "Doggy"
```

These types of properties are considered **stored properties**, as they store something on an object and affect its memory.

Section 2.5: Computed Properties

Different from stored properties, **computed properties** are built with a getter and a setter, performing necessary code when accessed and set. Computed properties must define a type:

```
var pi = 3.14

class Circle {
    var radius = 0.0
    var circumference: Double {
        get {
            return pi * radius * 2
        }
        set {
            radius = newValue / pi / 2
        }
    }
}

let circle = Circle()
circle.radius = 1
print(circle.circumference) // Prints "6.28"
circle.circumference = 14
print(circle.radius) // Prints "2.229..."
```

A read-only computed property is still declared with a `var`:

```
var circumference: Double {
    get {
        return pi * radius * 2
    }
}
```

Read-only computed properties can be shortened to exclude `get`:

```
var circumference: Double {
    return pi * radius * 2
}
```

Section 2.6: Local and Global Variables

Local variables are defined within a function, method, or closure:

```
func printSomething() {
    let localString = "I'm local!"
    print(localString)
}

func printSomethingAgain() {
```

```
    print(localString) // 错误
}
```

全局变量是在函数、方法或闭包外定义的，且不在任何类型内定义（即在所有大括号之外）。它们可以在任何地方使用：

```
let globalString = "我是全局的！"
print(globalString)

func useGlobalString() {
    print(globalString) // 可用！
}

for i in 0..<2 {
    print(globalString) // 可用！
}

class GlobalStringUser {
    var computeGlobalString {
        return globalString // works!
    }
}
```

全局变量是延迟定义的（参见“延迟属性”示例）。

第2.7节：类型属性

类型属性是属于类型本身的属性，而不是实例的属性。它们可以是存储属性或计算属性。你可以用static来声明类型属性：

```
结构体狗 {
    static var noise = "汪！"
}

print(狗.noise) // 输出 "汪！"
```

在类中，你可以使用class关键字代替static，使其可被重写。但是，这只能应用于计算属性：

```
类动物 {
    class var noise: 字符串 {
        return "动物叫声！"
    }
}
class Pig: Animal {
    重写类变量 noise: String {
        返回 "Oink oink!"
    }
}
```

这通常用于单例模式。


```
    print(localString) // error
}
```

Global variables are defined outside of a function, method, or closure, and are not defined within a type (think outside of all brackets). They can be used anywhere:

```
let globalString = "I'm global!"
print(globalString)

func useGlobalString() {
    print(globalString) // works!
}

for i in 0..<2 {
    print(globalString) // works!
}

class GlobalStringUser {
    var computeGlobalString {
        return globalString // works!
    }
}
```

Global variables are defined lazily (see "Lazy Properties" example).

Section 2.7: Type Properties

Type properties are properties on the type itself, not on the instance. They can be both stored or computed properties. You declare a type property with `static`:

```
struct Dog {
    static var noise = "Bark!"
}

print(Dog.noise) // Prints "Bark!"
```

In a class, you can use the `class` keyword instead of `static` to make it overridable. However, you can only apply this on computed properties:

```
class Animal {
    class var noise: String {
        return "Animal noise!"
    }
}
class Pig: Animal {
    override class var noise: String {
        return "Oink oink!"
    }
}
```

This is used often with the singleton pattern.

第3章：数字

第3.1节：数字类型和字面量

Swift内置的数字类型有：

- 与字长相关的有符号Int和无符号UInt。
- 固定大小的有符号整数Int8、Int16、Int32、Int64，以及无符号整数UInt8、UInt16、UInt32、UInt64。
- 浮点类型Float32/Float、Float64/Double和Float80（仅限x86）。

字面量

数字字面量的类型由上下文推断：

```
let x = 42    // x 默认是 Int 类型
let y = 42.0  // y 默认是 Double 类型

let z: UInt = 42     // z 是 UInt 类型
let w: Float = -1    // w 是 Float 类型
let q = 100 as Int8  // q 是 Int8 类型
```

下划线（_）可用于分隔数字字面量中的数字。前导零将被忽略。

浮点字面量可以使用有效数字和指数部分来指定（十进制为有效数字» e «指数»；十六进制为有效数字» p «指数»）。

整数字面量语法

```
let decimal = 10             // 十
let decimal = -1000              // 负一千
let decimal = -1_000             // 等同于 -1000
let decimal = 42_42_42           // 等同于 424242
let decimal = 0755               // 等同于 755，不是某些其他语言中的 493
let decimal = 0123456789

let hexadecimal = 0x10         // 等同于 16
let hexadecimal = 0x7FFFFFFF
let 十六进制 = 0xBadFace
let 十六进制 = 0x0123_4567_89ab_cdef

let 八进制 = 0o10                // 等同于 8
let 八进制 = 0o755               // 等同于 493
let 八进制 = -0o0123_4567

let binary = -0b101010           // 等同于 -42
let binary = 0b111_101_101       // 等同于 0o755
let binary = 0b1011_1010_1101    // 等同于 0xB_A_D
```

浮点字面量语法

```
let decimal = 0.0
let decimal = -42.0123456789
let decimal = 1_000.234_567_89

let decimal = 4.567e5                // 等同于 4.567×10⁵，或 456_700.0
let decimal = -2E-4                  // 等同于 -2×10⁻⁴，或 -0.0002
let decimal = 1e+0                   // 等同于 1×10⁰，或 1.0

let hexadecimal = 0x1p0              // 等同于 1×2⁰，或 1.0
let hexadecimal = 0x1p-2             // 等同于 1×2⁻²，或 0.25
```


Chapter 3: Numbers

Section 3.1: Number types and literals

Swift's built-in numeric types are:

- Word-sized (architecture-dependent) signed Int and unsigned UInt.
- Fixed-size signed integers Int8, Int16, Int32, Int64, and unsigned integers UInt8, UInt16, UInt32, UInt64.
- Floating-point types Float32/Float, Float64/Double, and Float80 (x86-only).

Literals

A numeric literal's type is inferred from context:

```
let x = 42    // x is Int by default
let y = 42.0  // y is Double by default

let z: UInt = 42     // z is UInt
let w: Float = -1    // w is Float
let q = 100 as Int8  // q is Int8
```

Underscores (_) may be used to separate digits in numeric literals. Leading zeros are ignored.

Floating point literals may be specified using significand and exponent parts (gnificand» **e** «*exponent*» for decimal; b> «*significand*» **p** «*exponent*» for hexadecimal).

Integer literal syntax

```
let decimal = 10             // ten
let decimal = -1000              // negative one thousand
let decimal = -1_000             // equivalent to -1000
let decimal = 42_42_42           // equivalent to 424242
let decimal = 0755               // equivalent to 755, NOT 493 as in some other languages
let decimal = 0123456789

let hexadecimal = 0x10         // equivalent to 16
let hexadecimal = 0x7FFFFFFF
let hexadecimal = 0xBadFace
let hexadecimal = 0x0123_4567_89ab_cdef

let octal = 0o10                 // equivalent to 8
let octal = 0o755                // equivalent to 493
let octal = -0o0123_4567

let binary = -0b101010           // equivalent to -42
let binary = 0b111_101_101       // equivalent to 0o755
let binary = 0b1011_1010_1101    // equivalent to 0xB_A_D
```

Floating-point literal syntax

```
let decimal = 0.0
let decimal = -42.0123456789
let decimal = 1_000.234_567_89

let decimal = 4.567e5                // equivalent to 4.567×10⁵, or 456_700.0
let decimal = -2E-4                  // equivalent to -2×10⁻⁴, or -0.0002
let decimal = 1e+0                   // equivalent to 1×10⁰, or 1.0

let hexadecimal = 0x1p0              // equivalent to 1×2⁰, or 1.0
let hexadecimal = 0x1p-2             // equivalent to 1×2⁻², or 0.25
```

```
let hexadecimal = 0xFEEDp+3         // 等同于 65261×2³，或 522088.0
let hexadecimal = 0x1234.5P4        // 等同于 0x12345，或 74565.0
let hexadecimal = 0x123.45P8        // 等同于 0x12345，或 74565.0
let hexadecimal = 0x12.345P12       // 等同于 0x12345，或 74565.0
let hexadecimal = 0x1.2345P16       // 等同于 0x12345，或 74565.0
let hexadecimal = 0x0.12345P20      // 等同于 0x12345，或 74565.0
```

第3.2节：数字与字符串的相互转换

使用 String 初始化器将数字转换为字符串：

```
String(1635999)                             // 返回 "1635999"
String(1635999, radix: 10)                  // 返回 "1635999"
String(1635999, radix: 2)                   // 返回 "110001111011010011111"
String(1635999, radix: 16)                  // 返回 "18f69f"
String(1635999, radix: 16, uppercase: true) // 返回 "18F69F"
String(1635999, radix: 17)                  // 返回 "129gf4"
String(1635999, radix: 36)                  // 返回 "z2cf"
```

或者在简单情况下使用字符串插值：

```
let x = 42, y = 9001
"在 \(x) 和 \(y) 之间"  // 等同于 "在 42 和 9001 之间"
```

使用数字类型的初始化器将字符串转换为数字：

```
if let num = Int("42") { /* ... */ }               // num 是 42
if let num = Int("Z2cF") { /* ... */ }             // 返回 nil（不是数字）
if let num = Int("z2cf", radix: 36) { /* ... */ }  // num 是 1635999
if let num = Int("Z2cF", radix: 36) { /* ... */ }  // num 是 1635999
if let num = Int8("Z2cF", radix: 36) { /* ... */ } // 返回 nil（对于 Int8 来说太大）
```

第3.3节：四舍五入

round

将值四舍五入到最接近的整数，x.5 向上舍入（但注意 -x.5 向下舍入）。

```
round(3.000) // 3
round(3.001) // 3
round(3.499) // 3
round(3.500) // 4
round(3.999) // 4

round(-3.000) // -3
round(-3.001) // -3
round(-3.499) // -3
round(-3.500) // -4  *** 小心这里 ***
round(-3.999) // -4
```

ceil

将任何带小数的数字向上舍入到下一个更大的整数。

```
ceil(3.000) // 3
ceil(3.001) // 4
ceil(3.999) // 4
```



```
let hexadecimal = 0xFEEDp+3         // equivalent to 65261×2³, or 522088.0
let hexadecimal = 0x1234.5P4        // equivalent to 0x12345, or 74565.0
let hexadecimal = 0x123.45P8        // equivalent to 0x12345, or 74565.0
let hexadecimal = 0x12.345P12       // equivalent to 0x12345, or 74565.0
let hexadecimal = 0x1.2345P16       // equivalent to 0x12345, or 74565.0
let hexadecimal = 0x0.12345P20      // equivalent to 0x12345, or 74565.0
```

Section 3.2: Convert numbers to/from strings

Use String initializers for converting numbers into strings:

```
String(1635999)                             // returns "1635999"
String(1635999, radix: 10)                  // returns "1635999"
String(1635999, radix: 2)                   // returns "110001111011010011111"
String(1635999, radix: 16)                  // returns "18f69f"
String(1635999, radix: 16, uppercase: true) // returns "18F69F"
String(1635999, radix: 17)                  // returns "129gf4"
String(1635999, radix: 36)                  // returns "z2cf"
```

Or use string interpolation for simple cases:

```
let x = 42, y = 9001
"Between \(x) and \(y)"  // equivalent to "Between 42 and 9001"
```

Use initializers of numeric types to convert strings into numbers:

```
if let num = Int("42") { /* ... */ }               // num is 42
if let num = Int("Z2cF") { /* ... */ }             // returns nil (not a number)
if let num = Int("z2cf", radix: 36) { /* ... */ }  // num is 1635999
if let num = Int("Z2cF", radix: 36) { /* ... */ }  // num is 1635999
if let num = Int8("Z2cF", radix: 36) { /* ... */ } // returns nil (too large for Int8)
```

Section 3.3: Rounding

round

Rounds the value to the nearest whole number with x.5 rounding up (but note that -x.5 rounds down).

```
round(3.000) // 3
round(3.001) // 3
round(3.499) // 3
round(3.500) // 4
round(3.999) // 4

round(-3.000) // -3
round(-3.001) // -3
round(-3.499) // -3
round(-3.500) // -4  *** careful here ***
round(-3.999) // -4
```

ceil

Rounds any number with a decimal value up to the next larger whole number.

```
ceil(3.000) // 3
ceil(3.001) // 4
ceil(3.999) // 4
```

```
ceil(-3.000) // -3
ceil(-3.001) // -3
ceil(-3.999) // -3
```

floor

将任何带小数的数字向下取整到比它小的下一个整数。

```
floor(3.000) // 3
floor(3.001) // 3
floor(3.999) // 3

floor(-3.000) // -3
floor(-3.001) // -4
floor(-3.999) // -4
```

Int

将Double转换为Int，去掉任何小数部分。

```
Int(3.000) // 3
Int(3.001) // 3
Int(3.999) // 3

Int(-3.000) // -3
Int(-3.001) // -3
Int(-3.999) // -3
```

说明

- round、ceil 和 floor 支持64位和32位架构。

第3.4节：随机数生成

arc4random_uniform(someNumber: UInt32) -> UInt32

这会给你范围在0到someNumber - 1之间的随机整数。

UInt32的最大值是4,294,967,295（即2^32 - 1）。

示例：

- 抛硬币

```
let flip = arc4random_uniform(2) // 0 或 1
```

- 掷骰子

```
let roll = arc4random_uniform(6) + 1 // 1...6
```

- 十月的随机一天

```
let day = arc4random_uniform(31) + 1 // 1...31
```

- 1990年代的随机一年

```
let year = 1990 + arc4random_uniform(10)
```



```
ceil(-3.000) // -3
ceil(-3.001) // -3
ceil(-3.999) // -3
```

floor

Rounds any number with a decimal value down to the next smaller whole number.

```
floor(3.000) // 3
floor(3.001) // 3
floor(3.999) // 3

floor(-3.000) // -3
floor(-3.001) // -4
floor(-3.999) // -4
```

Int

Converts a `Double` to an `Int`, dropping any decimal value.

```
Int(3.000) // 3
Int(3.001) // 3
Int(3.999) // 3

Int(-3.000) // -3
Int(-3.001) // -3
Int(-3.999) // -3
```

Notes

- `round`, `ceil` and `floor` handle both 64 and 32 bit architecture.

Section 3.4: Random number generation

`arc4random_uniform(someNumber: UInt32) -> UInt32`

This gives you random integers in the range `0` to `someNumber - 1`.

The maximum value for `UInt32` is 4,294,967,295 (that is, `2^32 - 1`).

Examples:

- Coin flip

```
let flip = arc4random_uniform(2) // 0 or 1
```

- Dice roll

```
let roll = arc4random_uniform(6) + 1 // 1...6
```

- Random day in October

```
let day = arc4random_uniform(31) + 1 // 1...31
```

- Random year in the 1990s

```
let year = 1990 + arc4random_uniform(10)
```

通用形式：

```
let number = min + arc4random_uniform(max - min + 1)
```

其中 number、max 和 min 均为 UInt32 类型。

说明

- arc4random 存在轻微的模偏差，因此推荐使用 arc4random_uniform。
- 你可以将 UInt32 值转换为 Int，但要注意可能会超出范围。

第3.5节：将一种数值类型转换为另一种

```
func doSomething1(value: Double) { /* ... */ }
func doSomething2(value: UInt) { /* ... */ }

let x = 42                // x 是一个 Int
doSomething1(Double(x))   // 将 x 转换为 Double
doSomething2(UInt(x))     // 将 x 转换为 UInt
```

如果值溢出或下溢，整数初始化器会产生 runtime error：

```
Int8(-129.0) // 致命错误：浮点数值无法转换为 Int8，因为它小于 Int8.min

Int8(-129)    // 崩溃：EXC_BAD_INSTRUCTION / SIGILL
Int8(-128)    // 正常
Int8(-2)      // 正常
Int8(17)      // 正常
Int8(127)     // 正常
Int8(128)     // 崩溃：EXC_BAD_INSTRUCTION / SIGILLInt8(128.
0)  // 致命错误：浮点数值无法转换为Int8，因为它大于Int8.max
```

浮点转整数转换向零方向取整：

```
Int(-2.2)  // -2
Int(-1.9)  // -1
Int(-0.1)  //  0
Int(1.0)   //  1
Int(1.2)   //  1
Int(1.9)   //  1
Int(2.0)   //  2
```

整数转浮点转换可能有损失：

```
Int(Float(1_000_000_000_000_000_000))  // 999999984306749440
```

第3.6节：指数运算

在 Swift 中，我们可以使用内置的 pow() 方法对 Double 类型进行幂运算：

```
pow(BASE, EXPONENT)
```

在下面的代码中，底数（5）被设置为指数（2）的幂：

```
let number = pow(5.0, 2.0) // 等于 25
```


General form:

```
let number = min + arc4random_uniform(max - min + 1)
```

where `number`, `max`, and `min` are `UInt32`.

Notes

- There is a slight modulo bias with `arc4random` so `arc4random_uniform` is preferred.
- You can cast a `UInt32` value to an `Int` but just beware of going out of range.

Section 3.5: Convert one numeric type to another

```
func doSomething1(value: Double) { /* ... */ }
func doSomething2(value: UInt) { /* ... */ }

let x = 42                // x is an Int
doSomething1(Double(x))   // convert x to a Double
doSomething2(UInt(x))     // convert x to a UInt
```

Integer initializers produce a **runtime error** if the value overflows or underflows:

```
Int8(-129.0) // fatal error: floating point value cannot be converted to Int8 because it is less
than Int8.min
Int8(-129)   // crash: EXC_BAD_INSTRUCTION / SIGILL
Int8(-128)   // ok
Int8(-2)     // ok
Int8(17)     // ok
Int8(127)    // ok
Int8(128)    // crash: EXC_BAD_INSTRUCTION / SIGILL
Int8(128.0)  // fatal error: floating point value cannot be converted to Int8 because it is greater
than Int8.max
```

Float-to-integer conversion **rounds values towards zero**:

```
Int(-2.2)  // -2
Int(-1.9)  // -1
Int(-0.1)  //  0
Int(1.0)   //  1
Int(1.2)   //  1
Int(1.9)   //  1
Int(2.0)   //  2
```

Integer-to-float conversion may be **lossy**:

```
Int(Float(1_000_000_000_000_000_000))  // 999999984306749440
```

Section 3.6: Exponentiation

In Swift, we can **exponentiate** `Double`s with the built-in `pow()` method:

```
pow(BASE, EXPONENT)
```

In the code below, the base (5) is set to the power of the exponent (2) :

```
let number = pow(5.0, 2.0) // Equals 25
```

第4章：字符串和字符

第4.1节：字符串和字符字面量

Swift中的字符串字面量用双引号（" ）括起来：

```
let greeting = "Hello!"  // greeting的类型是String
```

字符可以从字符串字面量初始化，只要该字面量只包含一个字形簇：

```
let chr: Character = "H" // 有效
let chr2: Character = "  "// 有效
let chr3: Character = "abc" // 无效 - 多个字形簇
```

字符串插值

字符串插值允许将表达式直接嵌入字符串字面量中。这可以用于所有类型的值，包括字符串、整数、浮点数等。

语法是反斜杠后跟括号包裹的值：\\(value)。括号内可以出现任何有效表达式，包括函数调用。

```
let number = 5
let interpolatedNumber = "\\(number)"  // 字符串是 "5"
let fortyTwo = "\\(6 * 7)"             // 字符串是 "42"

let example = "这篇文章有 \\(number) 次浏览\\(number == 1 ? "" : "s")"
// 上述例子将输出 "这篇文章有 5 次浏览"。
// 如果变量 number 的值是 1，则会输出 "这篇文章有 1 次浏览"。
```

对于自定义类型，字符串插值的默认行为是"\\(myobj)"等同于String(myobj)，这与print(myobj)使用的表示相同。你可以通过实现CustomStringConvertible协议来自定义此行为。

版本 ≥ 3.0

对于 Swift 3，根据SE-0089，String.init<T>(_:)已重命名为String.init<T>(describing:)。

字符串插值"\\(myobj)"将优先使用新的String.init<T: LosslessStringConvertible>(_:)初始化器，但如果该值不是LosslessStringConvertible，则会回退到init<T>(describing:)。

特殊字符

某些字符需要特殊的转义序列才能在字符串字面量中使用：

字符	含义
\\0	空字符
\\\\	一个普通的反斜杠，\\
\	一个制表符
\\v	垂直制表符
\\r	回车符carriage return
\	换行符line feed（“换行”）
\\"	双引号，"
\\'	单引号，'

Chapter 4: Strings and Characters

Section 4.1: String & Character Literals

String literals in Swift are delimited with double quotes ("):

```
let greeting = "Hello!"  // greeting's type is String
```

Characters can be initialized from string literals, as long as the literal contains only one grapheme cluster:

```
let chr: Character = "H" // valid
let chr2: Character = "□" // valid
let chr3: Character = "abc" // invalid - multiple grapheme clusters
```

String Interpolation

String interpolation allows injecting an expression directly into a string literal. This can be done with all types of values, including strings, integers, floating point numbers and more.

The syntax is a backslash followed by parentheses wrapping the value: `\\(value)`. Any valid expression may appear in the parentheses, including function calls.

```
let number = 5
let interpolatedNumber = "\\(number)"  // string is "5"
let fortyTwo = "\\(6 * 7)"             // string is "42"

let example = "This post has \\(number) view\\(number == 1 ? "" : "s")"
// It will output "This post has 5 views" for the above example.
// If the variable number had the value 1, it would output "This post has 1 view" instead.
```

For custom types, the default behavior of string interpolation is that `"\\(myobj)"` is equivalent to `String(myobj)`, the same representation used by `print(myobj)`. You can customize this behavior by implementing the `CustomStringConvertible` protocol for your type.

Version ≥ 3.0

For Swift 3, in accordance with SE-0089, `String.init<T>(_:)` has been renamed to `String.init<T>(describing:)`.

The string interpolation `"\\(myobj)"` will prefer the new `String.init<T: LosslessStringConvertible>(_:)` initializer, but will fall back to `init<T>(describing:)` if the value is not `LosslessStringConvertible`.

Special Characters

Certain characters require a special **escape sequence** to use them in string literals:

Character	Meaning
\\0	the null character
\\\\	a plain backslash, \\
\\t	a tab character
\\v	a vertical tab
\\r	a carriage return
\\n	a line feed ("newline")
\\"	a double quote, "
\\'	a single quote, '

`\\u{n}` Unicode 代码点 n（十六进制）

示例：

```
let message = "然后他说，\\"我 \\u{1F496} 你！\\""

print(message) // 然后他说，"我 你！"
```

第4.2节：字符串连接

使用 + 运算符连接字符串以生成新字符串：

```
let name = "约翰"
let surname = "苹果籽"
let fullName = name + " " + surname  // fullName 是 "约翰 苹果籽"
```

使用 += 复合赋值运算符或方法追加到一个可变字符串：

```
let str2 = "那里"
var instruction = "look over"
instruction += " " + str2  // instruction 现在是 "look over there"

var instruction = "look over"
instruction.append(" " + str2)  // instruction 现在是 "look over there"
```

向可变字符串追加单个字符：

```
var greeting: String = "Hello"
let exclamationMark: Character = "!"
greeting.append(exclamationMark)
// 生成修改后的字符串 (greeting) = "Hello!"
```

向可变字符串追加多个字符

```
var alphabet:String = "my ABCs: "
alphabet.append(contentsOf: (0x61...0x7A).map(UnicodeScalar.init)
                                         .map(Character.init) )
// 生成修改后的字符串 (alphabet) = "my ABCs: abcdefghijklmnopqrstuvwxyz"
```

版本 ≥ 3.0

`appendContentsOf(_:)` 已重命名为 `append(_:)`。

使用 joinWithSeparator(_:)连接字符串序列以形成新字符串：

```
let words = ["apple", "orange", "banana"]
let str = words.joinWithSeparator(" & ")

print(str)   // "apple & orange & banana"
```

版本 ≥ 3.0

`joinWithSeparator(_)` 已重命名为 `joined(separator:)`。

分隔符separator默认是空字符串，因此 ["a", "b", "c"].joined() == "abc"。

`\\u{n}` the Unicode code point *n* (in hexadecimal)

Example:

```
let message = "Then he said, \\"I \\u{1F496} you!\\""

print(message) // Then he said, "I □ you!"
```

Section 4.2: Concatenate strings

Concatenate strings with the + operator to produce a new string:

```
let name = "John"
let surname = "Appleseed"
let fullName = name + " " + surname  // fullName is "John Appleseed"
```

Append to a **mutable** string using the += compound assignment operator, or using a method:

```
let str2 = "there"
var instruction = "look over"
instruction += " " + str2  // instruction is now "look over there"

var instruction = "look over"
instruction.append(" " + str2)  // instruction is now "look over there"
```

Append a single character to a mutable String:

```
var greeting: String = "Hello"
let exclamationMark: Character = "!"
greeting.append(exclamationMark)
// produces a modified String (greeting) = "Hello!"
```

Append multiple characters to a mutable String

```
var alphabet:String = "my ABCs: "
alphabet.append(contentsOf: (0x61...0x7A).map(UnicodeScalar.init)
                                         .map(Character.init) )
// produces a modified string (alphabet) = "my ABCs: abcdefghijklmnopqrstuvwxyz"
```

Version ≥ 3.0

`appendContentsOf(_:)` has been renamed to `append(_:)`.

Join a sequence of strings to form a new string using `joinWithSeparator(_:)`:

```
let words = ["apple", "orange", "banana"]
let str = words.joinWithSeparator(" & ")

print(str)   // "apple & orange & banana"
```

Version ≥ 3.0

`joinWithSeparator(_:)` has been renamed to `joined(separator:)`.

The `separator` is the empty string by default, so `["a", "b", "c"].joined() == "abc"`.

第4.3节：字符串编码与分解

Swift 的 String 由 Unicode 码点组成。它可以以多种不同方式进行分解和编码。

```
let str = "ที่ ' ①!"
```

字符串分解

字符串的 characters 是 Unicode extended grapheme clusters（扩展字形簇）：

```
Array(str.characters)  // ["n ̄ ' ", "", "①", "!"]
```

unicodeScalars 是组成字符串的 Unicode code points（注意 ที่ 是一个字形簇，但由3个码点组成 — 3607, 3637, 3656 — 因此结果数组的长度与 characters 不同）：

```
str.unicodeScalars.map{ $0.value }  // [3607, 3637, 3656, 128076, 9312, 33]
```

你可以将字符串编码并分解为UTF-8（一系列UInt8）或UTF-16（一系列UInt16）：

```
Array(str.utf8)   // [224, 184, 151, 224, 184, 181, 224, 185, 136, 240, 159, 145, 140, 226, 145,
160, 33]
Array(str.utf16)  // [3607, 3637, 3656, 55357, 56396, 9312, 33]
```

字符串长度和迭代

字符串的characters、unicodeScalars、utf8和utf16都是Collection，因此你可以获取它们的count并对它们进行迭代：

```
// 注意：这些操作不一定快速或廉价！

str.characters.count     // 4
str.unicodeScalars.count // 6
str.utf8.count           // 17
str.utf16.count          // 7
```

```
for c in str.characters { // ...
for u in str.unicodeScalars { // ...
for byte in str.utf8 { // ...
for byte in str.utf16 { // ...
```

第4.4节：检查和比较字符串

检查字符串是否为空：

```
if str.isEmpty {
    // 如果字符串为空则执行某些操作
}

// 如果字符串为空，则用备用字符串替换：
let result = str.isEmpty ? "fallback string" : str
```

检查两个字符串是否相等（在Unicode规范等价的意义上）：

```
"abc" == "def"           // false
"abc" == "ABC"           // false
"abc" == "abc"           // true

// “带重音符的拉丁小写字母A” == “拉丁小写字母A” + “组合重音符”
```


Section 4.3: String Encoding and Decomposition

A Swift String is made of Unicode code points. It can be decomposed and encoded in several different ways.

```
let str = "ที่□①!"
```

Decomposing Strings

A string's `characters` are Unicode extended grapheme clusters:

```
Array(str.characters)  // ["ที่", "□", "①", "!"]
```

The `unicodeScalars` are the Unicode code points that make up a string (notice that ที่ is one grapheme cluster, but 3 code points — 3607, 3637, 3656 — so the length of the resulting array is not the same as with `characters`):

```
str.unicodeScalars.map{ $0.value }  // [3607, 3637, 3656, 128076, 9312, 33]
```

You can encode and decompose strings as UTF-8 (a sequence of `UInt8`s) or UTF-16 (a sequence of `UInt16`s):

```
Array(str.utf8)   // [224, 184, 151, 224, 184, 181, 224, 185, 136, 240, 159, 145, 140, 226, 145,
160, 33]
Array(str.utf16)  // [3607, 3637, 3656, 55357, 56396, 9312, 33]
```

String Length and Iteration

A string's `characters`, `unicodeScalars`, `utf8`, and `utf16` are all Collections, so you can get their `count` and iterate over them:

```
// NOTE: These operations are NOT necessarily fast/cheap!

str.characters.count     // 4
str.unicodeScalars.count // 6
str.utf8.count           // 17
str.utf16.count          // 7
```

```
for c in str.characters { // ...
for u in str.unicodeScalars { // ...
for byte in str.utf8 { // ...
for byte in str.utf16 { // ...
```

Section 4.4: Examine and compare strings

Check whether a string is empty:

```
if str.isEmpty {
    // do something if the string is empty
}

// If the string is empty, replace it with a fallback:
let result = str.isEmpty ? "fallback string" : str
```

Check whether two strings are equal (in the sense of Unicode canonical equivalence):

```
"abc" == "def"           // false
"abc" == "ABC"           // false
"abc" == "abc"           // true

// "LATIN SMALL LETTER A WITH ACUTE" == "LATIN SMALL LETTER A" + "COMBINING ACUTE ACCENT"
```

```
"á" == "á"  // true
```

检查字符串是否以另一个字符串开始/结束：

```
"fortitude".hasPrefix("fort")      // true
"Swift Language".hasSuffix("age")  // true
```

第4.5节：字符串反转

版本 = 2.2

```
let aString = "这是一个测试字符串。"

// 首先，反转字符串的字符
let reversedCharacters = aString.characters.reverse()

// 然后使用 String() 初始化器转换回字符串
let reversedString = String(reversedCharacters)

print(reversedString) // "。gnirts tset a si sihT"
```

版本 = 3.0

```
let reversedCharacters = aString.characters.reversed()
let reversedString = String(reversedCharacters)
```

第4.6节：检查字符串是否包含来自定义集合的字符

信件

版本 = 3.0

```
let letters = CharacterSet.letters

let phrase = "Test case"
let range = phrase.rangeOfCharacter(from: letters)

// 如果未找到字母，range 将为 nil
if let test = range {
    print("找到字母")
}
else {
    print("未找到字母")
}
```

版本 = 2.2

```
let letters = NSCharacterSet.letterCharacterSet()

let phrase = "Test case"
let range = phrase.rangeOfCharacterFromSet(letters)

// 如果未找到字母，range 将为 nil
if let test = range {
    print("找到字母")
}
else {
   print("未找到字母")
}
```

新的 `CharacterSet` 结构体也桥接到 Objective-C 的 `NSCharacterSet` 类，定义了几个预定义的集合，如下：


```
"\u{e1}" == "a\u{301}"  // true
```

Check whether a string starts/ends with another string:

```
"fortitude".hasPrefix("fort")      // true
"Swift Language".hasSuffix("age")  // true
```

Section 4.5: Reversing Strings

Version = 2.2

```
let aString = "This is a test string."

// first, reverse the String's characters
let reversedCharacters = aString.characters.reverse()

// then convert back to a String with the String() initializer
let reversedString = String(reversedCharacters)

print(reversedString) // ".gnirts tset a si sihT"
```

Version = 3.0

```
let reversedCharacters = aString.characters.reversed()
let reversedString = String(reversedCharacters)
```

Section 4.6: Check if String contains Characters from a Defined Set

Letters

Version = 3.0

```
let letters = CharacterSet.letters

let phrase = "Test case"
let range = phrase.rangeOfCharacter(from: letters)

// range will be nil if no letters is found
if let test = range {
   print("letters found")
}
else {
   print("letters not found")
}
```

Version = 2.2

```
let letters = NSCharacterSet.letterCharacterSet()

let phrase = "Test case"
let range = phrase.rangeOfCharacterFromSet(letters)

// range will be nil if no letters is found
if let test = range {
   print("letters found")
}
else {
  print("letters not found")
}
```

The new `CharacterSet` struct that is also bridged to the Objective-C `NSCharacterSet` class define several predefined sets as:

- decimalDigits
- capitalizedLetters
- alphanumerics
- controlCharacters
- illegalCharacters
- 更多内容请参见NSCharacterSet参考资料。

你也可以定义你自己的字符集：

版本 = 3.0

```
let phrase = "Test case"
let charset = CharacterSet(charactersIn: "t")

if let _ = phrase.rangeOfCharacter(from: charset, options: .caseInsensitive) {
   print("yes")
}
else {
   print("no")
}
```

版本 = 2.2

```
let charset = NSCharacterSet(charactersInString: "t")

if let _ = phrase.rangeOfCharacterFromSet(charset, options: .CaseInsensitiveSearch, range: nil) {
   print("yes")
}
else {
    print("no")
}
```

你也可以包含范围：

版本 = 3.0

```
let phrase = "Test case"
let charset = CharacterSet(charactersIn: "t")

if let _ = phrase.rangeOfCharacter(from: charset, options: .caseInsensitive, range:
phrase.startIndex..<phrase.endIndex)) {
   print("yes")
}
else {
   print("no")
}
```

第4.7节：字符串迭代

版本 < 3.0

```
let string = "我的精彩字符串"
var index = string.startIndex

while index != string.endIndex {
    print(string[index])
index = index.successor()
}
```

注意：`endIndex` 位于字符串末尾之后（即 `string[string.endIndex]` 是错误的，但 `string[string.startIndex]` 是正确的）。另外，在空字符串（`""`）中，`string.startIndex == string.endIndex` 为 `true` 请务必检查空字符串，因为不能在空字符串上调用 startIndex.successor()。

- decimalDigits
- capitalizedLetters
- alphanumerics
- controlCharacters
- illegalCharacters
- and more you can find in the NSCharacterSet reference.

You also can define your own set of characters:

Version = 3.0

```
let phrase = "Test case"
let charset = CharacterSet(charactersIn: "t")

if let _ = phrase.rangeOfCharacter(from: charset, options: .caseInsensitive) {
   print("yes")
}
else {
   print("no")
}
```

Version = 2.2

```
let charset = NSCharacterSet(charactersInString: "t")

if let _ = phrase.rangeOfCharacterFromSet(charset, options: .CaseInsensitiveSearch, range: nil) {
   print("yes")
}
else {
    print("no")
}
```

You can also include range:

Version = 3.0

```
let phrase = "Test case"
let charset = CharacterSet(charactersIn: "t")

if let _ = phrase.rangeOfCharacter(from: charset, options: .caseInsensitive, range:
phrase.startIndex..<phrase.endIndex)) {
   print("yes")
}
else {
   print("no")
}
```

Section 4.7: String Iteration

Version < 3.0

```
let string = "My fantastic string"
var index = string.startIndex

while index != string.endIndex {
    print(string[index])
    index = index.successor()
}
```

Note: `endIndex` is after the end of the string (i.e. `string[string.endIndex]` is an error, but `string[string.startIndex]` is fine). Also, in an empty string (`""`), `string.startIndex == string.endIndex` is `true`. Be sure to check for empty strings, since you cannot call `startIndex.successor()` on an empty string.

版本 = 3.0

在 Swift 3 中，字符串索引不再有 `successor()`、`predecessor()`、`advancedBy(_:)`、`advancedBy(_:limit:)` 或 `distanceTo(_:)`方法。

相反，这些操作被移到了集合（collection）中，由集合负责递增和递减其索引。

可用的方法有 .index(after:)、.index(before:) 和 .index(_:, offsetBy:)。

```
let string = "我的精彩字符串"
var currentIndex = string.startIndex

while currentIndex != string.endIndex {
    print(string[currentIndex])
currentIndex = string.index(after: currentIndex)
}
```

注意：我们使用currentIndex作为变量名，以避免与.index方法混淆。

例如，如果你想反方向遍历：

版本 < 3.0

```
var index:String.Index? = string.endIndex.predecessor()

while index != nil {
    print(string[index!])
    if index != string.startIndex {
        index = index.predecessor()
    }
    else {
index = nil
    }
}
```

(或者你也可以先将字符串反转，但如果不需要遍历整个字符串，你可能更喜欢像这样的方法)

版本 = 3.0

```
var currentIndex: String.Index? = string.index(before: string.endIndex)

while currentIndex != nil {
    print(string[currentIndex!])
    if currentIndex != string.startIndex {
        currentIndex = string.index(before: currentIndex!)
    }
    else {
currentIndex = nil
    }
}
```

注意，Index 是一种对象类型，而不是 Int。你不能像下面这样访问字符串的字符：

```
let string = "My string"
string[2] // 不能这样做
string.characters[2] // 也不能这样做
```

但是你可以像下面这样获取特定索引：

Version = 3.0

In Swift 3, String indexes no longer have `successor()`, `predecessor()`, `advancedBy(_:)`, `advancedBy(_:limit:)`, or `distanceTo(_:)`.

Instead, those operations are moved to the collection, which is now responsible for incrementing and decrementing its indices.

Available methods are `.index(after:)`, `.index(before:)` and `.index(_:, offsetBy:)`.

```
let string = "My fantastic string"
var currentIndex = string.startIndex

while currentIndex != string.endIndex {
    print(string[currentIndex])
    currentIndex = string.index(after: currentIndex)
}
```

Note: we're using `currentIndex` as a variable name to avoid confusion with the `.index` method.

And, for example, if you want to go the other way:

Version < 3.0

```
var index:String.Index? = string.endIndex.predecessor()

while index != nil {
    print(string[index!])
    if index != string.startIndex {
        index = index.predecessor()
    }
    else {
        index = nil
    }
}
```

(Or you could just reverse the string first, but if you don't need to go all the way through the string you probably would prefer a method like this)

Version = 3.0

```
var currentIndex: String.Index? = string.index(before: string.endIndex)

while currentIndex != nil {
    print(string[currentIndex!])
    if currentIndex != string.startIndex {
        currentIndex = string.index(before: currentIndex!)
    }
    else {
        currentIndex = nil
    }
}
```

Note, `Index` is an object type, and not an `Int`. You cannot access a character of string as follows:

```
let string = "My string"
string[2] // can't do this
string.characters[2] // and also can't do this
```

But you can get a specific index as follows:

版本 < 3.0
```
index = string.startIndex.advanceBy(2)
```
版本 = 3.0
```
currentIndex = string.index(string.startIndex, offsetBy: 2)
```

并且可以像这样向后移动：

版本 < 3.0
```
index = string.endIndex.advancedBy(-2)
```
版本 = 3.0
```
currentIndex = string.index(string.endIndex, offsetBy: -2)
```

如果你可能会超出字符串的边界，或者你想指定一个限制，可以使用：

版本 < 3.0
```
index = string.startIndex.advanceBy(20, limit: string.endIndex)
```
版本 = 3.0
```
currentIndex = string.index(string.startIndex, offsetBy: 20, limitedBy: string.endIndex)
```

或者也可以直接遍历字符串中的字符，但这可能根据

上下文的不同而不太实用：

```
for c in string.characters {
    print(c)
}
```

第4.8节：将字符串拆分为数组

在Swift中，你可以通过在某个字符处切割字符串，轻松地将字符串分割成数组：

版本 = 3.0
```
let startDate = "23:51"

let startDateAsArray = startDate.components(separatedBy: ":") // ["23", "51"]`
```
版本 = 2.2
```
let startDate = "23:51"

let startArray = startDate.componentsSeparatedByString(":") // ["23", "51"]`
```

或者当分隔符不存在时：

版本 = 3.0
```
let myText = "MyText"

let myTextArray = myText.components(separatedBy: " ") // myTextArray 是 ["MyText"]
```
版本 = 2.2
```
let myText = "MyText"

let myTextArray = myText.componentsSeparatedByString(" ") // myTextArray 是 ["MyText"]
```

第4.9节：Unicode

设置值

直接使用Unicode

Version < 3.0
```
index = string.startIndex.advanceBy(2)
```
Version = 3.0
```
currentIndex = string.index(string.startIndex, offsetBy: 2)
```

And can go backwards like this:

Version < 3.0
```
index = string.endIndex.advancedBy(-2)
```
Version = 3.0
```
currentIndex = string.index(string.endIndex, offsetBy: -2)
```

If you might exceed the string's bounds, or you want to specify a limit you can use:

Version < 3.0
```
index = string.startIndex.advanceBy(20, limit: string.endIndex)
```
Version = 3.0
```
currentIndex = string.index(string.startIndex, offsetBy: 20, limitedBy: string.endIndex)
```

Alternatively one can just iterate through the characters in a string, but this might be less useful depending on the context:

```
for c in string.characters {
    print(c)
}
```

Section 4.8: Splitting a String into an Array

In Swift you can easily separate a String into an array by slicing it at a certain character:

Version = 3.0
```
let startDate = "23:51"

let startDateAsArray = startDate.components(separatedBy: ":") // ["23", "51"]`
```
Version = 2.2
```
let startDate = "23:51"

let startArray = startDate.componentsSeparatedByString(":") // ["23", "51"]`
```

Or when the separator isn't present:

Version = 3.0
```
let myText = "MyText"

let myTextArray = myText.components(separatedBy: " ") // myTextArray is ["MyText"]
```
Version = 2.2
```
let myText = "MyText"

let myTextArray = myText.componentsSeparatedByString(" ") // myTextArray is ["MyText"]
```

Section 4.9: Unicode

Setting values

Using Unicode directly

```
var str: String = "我想去北京, 莫斯科, मुंबई, القاهرة, 和 首尔市. "
var character: Character = ""
```

使用十六进制值

```
var str: String = "\\u{61}\\u{5927}\\u{1F34E}\\u{3C0}" // a大π
var character: Character = "\\u{65}\\u{301}" // é = "e" + 重音符号
```

注意Swift的`Character`可以由多个Unicode码点组成，但看起来是单个字符。这称为扩展字形簇（Extended Grapheme Cluster）。

转换

字符串 --> 十六进制

```
// 访问`str`的不同Unicode编码视图
str.utf8
str.utf16
str.unicodeScalars // UTF-32
```

十六进制 --> 字符串

```
let value0: UInt8 = 0x61
let value1: UInt16 = 0x5927
let value2: UInt32 = 0x1F34E

let string0 = String(UnicodeScalar(value0)) // a
let string1 = String(UnicodeScalar(value1)) // 大
let string2 = String(UnicodeScalar(value2)) //

// 将十六进制数组转换为字符串
let myHexArray = [0x43, 0x61, 0x74, 0x203C, 0x1F431] // 一个整数数组
var myString = ""
for hexValue in myHexArray {
    myString.append(UnicodeScalar(hexValue))
}
print(myString) // Cat!!
```

注意，对于 UTF-8 和 UTF-16，转换并不总是这么简单，因为像表情符号这样的字符不能用单个 UTF-16 值编码。它需要使用代理对。

第4.10节：将 Swift 字符串转换为数字类型

```
Int("123") // 返回 Int 类型的 123
Int("abcd") // 返回 nil
Int("10") // 返回 Int 类型的 10
Int("10", radix: 2) // 返回 Int 类型的 2
Double("1.5") // 返回 Double 类型的 1.5
Double("abcd") // 返回 nil
```

请注意，这样做会返回一个Optional值，使用前应相应地进行解包。

第4.11节：字符串与Data / NSData之间的转换

要将字符串转换为Data / NSData，或从Data / NSData转换为字符串，我们需要使用特定的编码对字符串进行编码。最


```
var str: String = "I want to visit 北京, Москва, म्ंबई, القاهرة, and 서울시. □"
var character: Character = "□"
```

Using hexadecimal values

```
var str: String = "\\u{61}\\u{5927}\\u{1F34E}\\u{3C0}" // a大□π
var character: Character = "\\u{65}\\u{301}" // é = "e" + accent mark
```

Note that the Swift `Character` can be composed of multiple Unicode code points, but appears to be a single character. This is called an Extended Grapheme Cluster.

Conversions

String --> Hex

```
// Accesses views of different Unicode encodings of `str`
str.utf8
str.utf16
str.unicodeScalars // UTF-32
```

Hex --> String

```
let value0: UInt8 = 0x61
let value1: UInt16 = 0x5927
let value2: UInt32 = 0x1F34E

let string0 = String(UnicodeScalar(value0)) // a
let string1 = String(UnicodeScalar(value1)) // 大
let string2 = String(UnicodeScalar(value2)) // □

// convert hex array to String
let myHexArray = [0x43, 0x61, 0x74, 0x203C, 0x1F431] // an Int array
var myString = ""
for hexValue in myHexArray {
    myString.append(UnicodeScalar(hexValue))
}
print(myString) // Cat‼□
```

Note that for UTF-8 and UTF-16 the conversion is not always this easy because things like emoji cannot be encoded with a single UTF-16 value. It takes a surrogate pair.

Section 4.10: Converting Swift string to a number type

```
Int("123") // Returns 123 of Int type
Int("abcd") // Returns nil
Int("10") // Returns 10 of Int type
Int("10", radix: 2) // Returns 2 of Int type
Double("1.5") // Returns 1.5 of Double type
Double("abcd") // Returns nil
```

Note that doing this returns an `Optional` value, which should be unwrapped accordingly before being used.

Section 4.11: Convert String to and from Data / NSData

To convert String to and from Data / NSData we need to encode this string with a specific encoding. The most

著名的是UTF-8，它是Unicode字符的8位表示，适合通过基于ASCII的系统进行传输或存储。以下是所有可用的String编码列表

字符串转Data/NSData

版本 = 3.0

```
let data = string.data(using: .utf8)
```

版本 = 2.2

```
let data = string.dataUsingEncoding(NSUTF8StringEncoding)
```

Data/NSData转字符串

版本 = 3.0

```
let string = String(data: data, encoding: .utf8)
```

版本 = 2.2

```
let string = String(data: data, encoding: NSUTF8StringEncoding)
```

第4.12节：字符串格式化

前导零

```
let number: Int = 7
let str1 = String(format: "%03d", number) // 007
let str2 = String(format: "%05d", number) // 00007
```

小数点后的数字

```
let number: Float = 3.14159
let str1 = String(format: "%.2f", number) // 3.14
let str2 = String(format: "%.4f", number) // 3.1416（四舍五入）
```

十进制转十六进制

```
let number: Int = 13627
let str1 = String(format: "%2X", number) // 353B
let str2 = String(format: "%2x", number) // 353b（注意小写的b）
```

或者可以使用执行相同操作的专用初始化器：

```
let number: Int = 13627
let str1 = String(number, radix: 16, uppercase: true) //353B
let str2 = String(number, radix: 16) // 353b
```

将十进制转换为任意进制的数字

```
let number: Int = 13627
let str1 = String(number, radix: 36) // aij
```

进制是 Int，范围在 [2, 36] 之间。

第4.13节：大写和小写字符串

将字符串中的所有字符转换为大写或小写：

版本 = 2.2

```
let text = "AaBbCc"
let uppercase = text.uppercaseString // "AABBCC"
```


famous one is UTF-8 which is an 8-bit representation of Unicode characters, suitable for transmission or storage by ASCII-based systems. Here is a list of all available String Encodings

String to Data/NSData

Version = 3.0

```
let data = string.data(using: .utf8)
```

Version = 2.2

```
let data = string.dataUsingEncoding(NSUTF8StringEncoding)
```

Data/NSData to String

Version = 3.0

```
let string = String(data: data, encoding: .utf8)
```

Version = 2.2

```
let string = String(data: data, encoding: NSUTF8StringEncoding)
```

Section 4.12: Formatting Strings

Leading Zeros

```
let number: Int = 7
let str1 = String(format: "%03d", number) // 007
let str2 = String(format: "%05d", number) // 00007
```

Numbers after Decimal

```
let number: Float = 3.14159
let str1 = String(format: "%.2f", number) // 3.14
let str2 = String(format: "%.4f", number) // 3.1416 (rounded)
```

Decimal to Hexadecimal

```
let number: Int = 13627
let str1 = String(format: "%2X", number) // 353B
let str2 = String(format: "%2x", number) // 353b (notice the lowercase b)
```

Alternatively one could use specialized initializer that does the same:

```
let number: Int = 13627
let str1 = String(number, radix: 16, uppercase: true) //353B
let str2 = String(number, radix: 16) // 353b
```

Decimal to a number with arbitrary radix

```
let number: Int = 13627
let str1 = String(number, radix: 36) // aij
```

Radix is Int in [2, 36].

Section 4.13: Uppercase and Lowercase Strings

To make all the characters in a String uppercase or lowercase:

Version = 2.2

```
let text = "AaBbCc"
let uppercase = text.uppercaseString // "AABBCC"
```

```
let lowercase = text.lowercaseString // "aabbcc"
```

版本 = 3.0

```
let text = "AaBbCc"
let uppercase = text.uppercased() // "AABBCC"
let lowercase = text.lowercased() // "aabbcc"
```

第4.14节：从字符串中移除不在集合中的字符

版本 = 2.2

```
func removeCharactersNotInSetFromText(text: String, set: Set<Character>) -> String {
   return String(text.characters.filter { set.contains( $0) })
}

let text = "Swift 3.0 Come Out"
var chars = Set([Character]("abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLKMNOPQRSTUVWXYZ".characters))
let newText = removeCharactersNotInSetFromText(text, set: chars) // "SwiftComeOut"
```

版本 = 3.0

```
func removeCharactersNotInSetFromText(text: String, set: Set<Character>) -> String {
  return String(text.characters.filter { set.contains( $0) })
}

let text = "Swift 3.0 Come Out"
var chars = Set([Character]("abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLKMNOPQRSTUVWXYZ".characters))
let newText = removeCharactersNotInSetFromText(text: text, set: chars)
```

第4.15节：统计字符串中某字符的出现次数

给定一个String和一个Character

```
let text = "Hello World"
let char: Character = "o"
```

我们可以使用以下方法统计Character在String中出现的次数

```
let sensitiveCount = text.characters.filter { $0 == char }.count // 区分大小写
let insensitiveCount = text.lowercaseString.characters.filter { $0 ==
Character(String(char).lowercaseString) } // 不区分大小写
```

第4.16节：移除首尾空白字符和换行符

版本 < 3.0

```
let someString = "  Swift语言  "let trimmedStri
ng =
someString.stringByTrimmingCharactersInSet(NSCharacterSet.whitespaceAndNewlineCharacterSet())
// "Swift语言"
```

方法stringByTrimmingCharactersInSet返回一个新字符串，该字符串通过移除原字符串两端包含在指定字符集中的字符生成。

我们也可以只移除空白字符或换行符。

只移除空白字符：

```
let trimmedWhiteSpace =
```



```
let lowercase = text.lowercaseString // "aabbcc"
```

Version = 3.0

```
let text = "AaBbCc"
let uppercase = text.uppercased() // "AABBCC"
let lowercase = text.lowercased() // "aabbcc"
```

Section 4.14: Remove characters from a string not defined in Set

Version = 2.2

```
func removeCharactersNotInSetFromText(text: String, set: Set<Character>) -> String {
   return String(text.characters.filter { set.contains( $0) })
}

let text = "Swift 3.0 Come Out"
var chars = Set([Character]("abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLKMNOPQRSTUVWXYZ".characters))
let newText = removeCharactersNotInSetFromText(text, set: chars) // "SwiftComeOut"
```

Version = 3.0

```
func removeCharactersNotInSetFromText(text: String, set: Set<Character>) -> String {
  return String(text.characters.filter { set.contains( $0) })
}

let text = "Swift 3.0 Come Out"
var chars = Set([Character]("abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLKMNOPQRSTUVWXYZ".characters))
let newText = removeCharactersNotInSetFromText(text: text, set: chars)
```

Section 4.15: Count occurrences of a Character into a String

Given a `String` and a `Character`

```
let text = "Hello World"
let char: Character = "o"
```

We can count the number of times the `Character` appears into the `String` using

```
let sensitiveCount = text.characters.filter { $0 == char }.count // case-sensitive
let insensitiveCount = text.lowercaseString.characters.filter { $0 ==
Character(String(char).lowercaseString) } // case-insensitive
```

Section 4.16: Remove leading and trailing WhiteSpace and NewLine

Version < 3.0

```
let someString = "  Swift Language  \n"
let trimmedString =
someString.stringByTrimmingCharactersInSet(NSCharacterSet.whitespaceAndNewlineCharacterSet())
// "Swift Language"
```

Method `stringByTrimmingCharactersInSet` returns a new string made by removing from both ends of the String characters contained in a given character set.

We can also just remove only whitespace or newline.

Removing only whitespace:

```
let trimmedWhiteSpace =
```

```
someString.stringByTrimmingCharactersInSet(NSCharacterSet.whitespaceCharacterSet())// "Swift语言  "
```

只移除换行符：

```
let trimmedNewLine =
someString.stringByTrimmingCharactersInSet(NSCharacterSet.newlineCharacterSet())
// "  Swift Language  "
```

版本 = 3.0

```
let someString = "  Swift Language  "let trim

medString = someString.trimmingCharacters(in: .whitespacesAndNewlines)// "Swift Language"


let trimmedWhiteSpace = someString.trimmingCharacters(in: .whitespaces)// "Swift Lan
guage  "

let trimmedNewLine = someString.trimmingCharacters(in: .newlines)
// "  Swift Language  "
```

注意：所有这些方法都属于Foundation。如果尚未通过其他库（如Cocoa或UIKit）导入Foundation，请使用import Foundation。


```
someString.stringByTrimmingCharactersInSet(NSCharacterSet.whitespaceCharacterSet())
// "Swift Language  \n"
```

Removing only newline:

```
let trimmedNewLine =
someString.stringByTrimmingCharactersInSet(NSCharacterSet.newlineCharacterSet())
// "  Swift Language  "
```

Version = 3.0

```
let someString = "  Swift Language  \n"

let trimmedString = someString.trimmingCharacters(in: .whitespacesAndNewlines)
// "Swift Language"

let trimmedWhiteSpace = someString.trimmingCharacters(in: .whitespaces)
// "Swift Language  \n"

let trimmedNewLine = someString.trimmingCharacters(in: .newlines)
// "  Swift Language  "
```

Note: all these methods belong to `Foundation`. Use `import Foundation` if Foundation isn't already imported via other libraries like Cocoa or UIKit.

第5章：布尔值

第5.1节：什么是Bool？

Bool是一种布尔类型，有两个可能的值：true和false。

```
let aTrueBool = true
let aFalseBool = false
```

布尔值用于控制流语句作为条件。 `if` 语句使用布尔条件来决定执行哪一段代码：

```
func test(_ someBoolean: Bool) {
    if someBoolean {
        print("是真的！")
    }
    else {
        print("是假的！")
    }
}
test(aTrueBool)  // 输出 "是真的！"
```

第5.2节：布尔值和内联条件表达式

处理布尔值的一种简洁方法是使用带有 `a ? b : c` 三元运算符的内联条件表达式，它是Swift的基本操作的一部分。

内联条件表达式由3个部分组成：

```
question ? answerIfTrue : answerIfFalse
```

其中 question 是一个布尔值，会被求值；如果 question 为真，则返回 answerIfTrue 的值，如果为假，则返回 answerIfFalse 的值。

上述表达式等同于：

```
if question {
    answerIfTrue
} else {
answerIfFalse
}
```

使用内联条件表达式可以根据布尔值返回一个值：

```
func isTurtle(_ value: Bool) {
     let color = value ? "green" : "red"
     print("这只动物是\(color)")
}

isTurtle(true) // 输出 '这只动物是绿色'
isTurtle(false) // 输出 '这只动物是红色'
```

你也可以根据布尔值调用方法：

```
func actionDark() {
```


Chapter 5: Booleans

Section 5.1: What is Bool?

Bool is a Boolean type with two possible values: `true` and `false`.

```
let aTrueBool = true
let aFalseBool = false
```

Bools are used in control-flow statements as conditions. The `if` statement uses a Boolean condition to determine which block of code to run:

```
func test(_ someBoolean: Bool) {
    if someBoolean {
        print("IT'S TRUE!")
    }
    else {
        print("IT'S FALSE!")
    }
}
test(aTrueBool)  // prints "IT'S TRUE!"
```

Section 5.2: Booleans and Inline Conditionals

A clean way to handle booleans is using an inline conditional with the a ? b : c ternary operator, which is part of Swift's Basic Operations.

The inline conditional is made up of 3 components:

```
question ? answerIfTrue : answerIfFalse
```

where question is a boolean that is evaluated and answerIfTrue is the value returned if the question is true, and answerIfFalse is the value returned if the question is false.

The expression above is the same as:

```
if question {
    answerIfTrue
} else {
    answerIfFalse
}
```

With inline conditionals you return a value based on a boolean:

```
func isTurtle(_ value: Bool) {
    let color = value ? "green" : "red"
    print("The animal is \(color)")
}

isTurtle(true) // outputs 'The animal is green'
isTurtle(false) // outputs 'The animal is red'
```

You can also call methods based on a boolean value:

```
func actionDark() {
```

```
    print("欢迎来到黑暗面")
}

func actionJedi() {
    print("欢迎加入绝地武士团")
}

func welcome(_ isJedi: Bool) {
    isJedi ? actionJedi() : actionDark()
}

welcome(true) // 输出 '欢迎加入绝地武士'
welcome(false) // 输出 '欢迎加入黑暗面'
```

内联条件允许简洁的一行布尔值判断

第5.3节：布尔逻辑运算符

或（||）运算符如果两个操作数中有一个为真，则返回真，否则返回假。例如，以下代码返回真，因为或运算符两边的表达式中至少有一个为真：

```
if (10 < 20) || (20 < 10) {
    print("表达式为真")
}
```

与（&&）运算符仅当两个操作数都为真时返回真。以下示例将返回假，因为两个操作数表达式中只有一个为真：

```
if (10 < 20) && (20 < 10) {
    print("表达式为真")
}
```

异或（^）运算符当且仅当两个操作数中有一个为真时返回真。例如，以下代码将返回真，因为只有一个操作数为真：

```
if (10 < 20) ^ (20 < 10) {
    print("表达式为真")
}
```

第5.4节：使用前缀!运算符取反布尔值

前缀操作符 ！返回其参数的逻辑否定。也就是说，!true 返回 false，!false 返回 true。

```
print(!true)  // 输出 "false"
print(!false) // 输出 "true"

func test(_ someBoolean: Bool) {
    if !someBoolean {
        print("someBoolean 是 false")
    }
}
```



```
    print("Welcome to the dark side")
}

func actionJedi() {
    print("Welcome to the Jedi order")
}

func welcome(_ isJedi: Bool) {
    isJedi ? actionJedi() : actionDark()
}

welcome(true) // outputs 'Welcome to the Jedi order'
welcome(false) // outputs 'Welcome to the dark side'
```

Inline conditionals allow for clean one-liner boolean evaluations

Section 5.3: Boolean Logical Operators

The OR (||) operator returns true if one of its two operands evaluates to true, otherwise it returns false. For example, the following code evaluates to true because at least one of the expressions either side of the OR operator is true:

```
if (10 < 20) || (20 < 10) {
    print("Expression is true")
}
```

The AND (&&) operator returns true only if both operands evaluate to be true. The following example will return false because only one of the two operand expressions evaluates to true:

```
if (10 < 20) && (20 < 10) {
    print("Expression is true")
}
```

The XOR (^) operator returns true if one and only one of the two operands evaluates to true. For example, the following code will return true since only one operator evaluates to be true:

```
if (10 < 20) ^ (20 < 10) {
    print("Expression is true")
}
```

Section 5.4: Negate a Bool with the prefix ! operator

The prefix ! operator returns the **logical negation** of its argument. That is, `!true` returns `false`, and `!false` returns `true`.

```
print(!true)  // prints "false"
print(!false) // prints "true"

func test(_ someBoolean: Bool) {
    if !someBoolean {
        print("someBoolean is false")
    }
}
```

第6章：数组

数组是一种有序的、支持随机访问的集合类型。数组是应用程序中最常用的数据类型之一。我们使用数组类型来存储单一类型的元素，即数组的 Element 类型。数组可以存储任何类型的元素——从整数到字符串再到类。

第6.1节：数组基础

数组是 Swift 标准库中的一种有序集合类型。它提供 O(1) 的随机访问和动态重新分配。数组是泛型类型，因此其包含的值的类型在编译时已知。

由于数组是值类型，其可变性由是否被标注为 var（可变）或 let（不可变）决定。

类型 [Int]（表示：包含 Int 的数组）是 Array<T> 的语法糖。

更多关于数组的内容请参见《Swift 编程语言》。

空数组

以下三个声明是等价的：

```
// 一个可变的字符串数组，初始为空。

var arrayOfStrings: [String] = []      // 类型注解 + 数组字面量
var arrayOfStrings = [String]()        // 调用 [String] 初始化器
var arrayOfStrings = Array<String>()   // 无语法糖
```

数组字面量

数组字面量用方括号括起逗号分隔的元素来表示：

```
// 创建一个不可变的 [Int] 类型数组，包含 2、4 和 7
let arrayOfInts = [2, 4, 7]
```

编译器通常可以根据字面量中的元素推断数组的类型，但显式的**类型注解**可以覆盖默认推断：

```
let arrayOfUInt8s: [UInt8] = [2, 4, 7]   // 变量上的类型注解
let arrayOfUInt8s = [2, 4, 7] as [UInt8] // 初始化表达式上的类型注解
let arrayOfUInt8s = [2 as UInt8, 4, 7]   // 对一个元素显式声明，其他元素推断
```

重复值的数组

```
// 一个不可变的 [String] 类型数组，包含 ["Example", "Example", "Example"]
let arrayOfStrings = Array(repeating: "Example",count: 3)
```

从其他序列创建数组

```
let dictionary = ["foo" : 4, "bar" : 6]

// 一个不可变的 [(String, Int)] 类型数组，包含 [("bar", 6), ("foo", 4)]
let arrayOfKeyValuePairs = Array(dictionary)
```

多维数组

在 Swift 中，多维数组是通过嵌套数组创建的：一个二维的 Int 数组是 [[Int]]（或者 Array<Array<Int>>）。

```
let array2x3 = [
    [1, 2, 3],
```


Chapter 6: Arrays

Array is an ordered, random-access collection type. Arrays are one of the most commonly used data types in an app. We use the Array type to hold elements of a single type, the array's Element type. An array can store any kind of elements---from integers to strings to classes.

Section 6.1: Basics of Arrays

`Array` is an ordered collection type in the Swift standard library. It provides O(1) random access and dynamic reallocation. Array is a generic type, so the type of values it contains are known at compile time.

As `Array` is a value type, its mutability is defined by whether it is annotated as a `var` (mutable) or `let` (immutable).

The type `[Int]` (meaning: an array containing `Int`s) is syntactic sugar for `Array<T>`.

Read more about arrays in The Swift Programming Language.

Empty arrays

The following three declarations are equivalent:

```
// A mutable array of Strings, initially empty.

var arrayOfStrings: [String] = []      // type annotation + array literal
var arrayOfStrings = [String]()        // invoking the [String] initializer
var arrayOfStrings = Array<String>()   // without syntactic sugar
```

Array literals

An array literal is written with square brackets surrounding comma-separated elements:

```
// Create an immutable array of type [Int] containing 2, 4, and 7
let arrayOfInts = [2, 4, 7]
```

The compiler can usually infer the type of an array based on the elements in the literal, but explicit **type annotations** can override the default:

```
let arrayOfUInt8s: [UInt8] = [2, 4, 7]   // type annotation on the variable
let arrayOfUInt8s = [2, 4, 7] as [UInt8] // type annotation on the initializer expression
let arrayOfUInt8s = [2 as UInt8, 4, 7]   // explicit for one element, inferred for the others
```

Arrays with repeated values

```
// An immutable array of type [String], containing ["Example", "Example", "Example"]
let arrayOfStrings = Array(repeating: "Example",count: 3)
```

Creating arrays from other sequences

```
let dictionary = ["foo" : 4, "bar" : 6]

// An immutable array of type [(String, Int)], containing [("bar", 6), ("foo", 4)]
let arrayOfKeyValuePairs = Array(dictionary)
```

Multi-dimensional arrays

In Swift, a multidimensional array is created by nesting arrays: a 2-dimensional array of `Int` is `[[Int]]` (or `Array<Array<Int>>`).

```
let array2x3 = [
    [1, 2, 3],
```

```
    [4, 5, 6]
]
// array2x3[0][1] 是 2，array2x3[1][2] 是 6。
```

要创建一个多维的重复值数组，请使用嵌套调用数组初始化器：

```
var array3x4x5 = Array(repeating: Array(repeating: Array(repeating: 0,count: 5),count: 4),count: 3)
```

第6.2节：使用flatMap(_:)从数组中提取指定类型的值

things数组包含Any类型的值。

```
let things: [Any] = [1, "Hello", 2, true, false, "World", 3]
```

我们可以提取指定类型的值，并创建一个该特定类型的新数组。假设我们想提取所有的Int类型，并以安全的方式将它们放入一个Int数组中。

```
let numbers = things.flatMap { $0 as? Int }
```

现在numbers被定义为[Int]。flatMap函数会丢弃所有nil元素，因此结果只包含以下值：

```
[1, 2, 3]
```

第6.3节：使用reduce(_:combine:)合并数组元素

reduce(_:combine:)可用于将序列的元素合并为单个值。它接受一个结果的初始值，以及一个对每个元素应用的闭包——该闭包将返回新的累积值。

例如，我们可以用它来对一个数字数组求和：

```
let numbers = [2, 5, 7, 8, 10, 4]

let sum = numbers.reduce(0) {accumulator, element in
    return accumulator + element
}

print(sum) // 36
```

我们传入了0作为初始值，因为这是求和的逻辑初始值。如果传入一个值N，结果的sum将是N + 36。传给reduce的闭包有两个参数。 accumulator是当前的累计值，它被赋值为闭包在每次迭代中返回的值。 element是当前迭代的元素。

如本例所示，我们传入了一个(Int, Int) -> Int的闭包给reduce，闭包简单地输出两个输入的相加–实际上我们可以直接传入+操作符，因为操作符在Swift中是函数：

```
let sum = numbers.reduce(0, combine: +)
```



```
    [4, 5, 6]
]
// array2x3[0][1] is 2, and array2x3[1][2] is 6.
```

To create a multidimensional array of repeated values, use nested calls of the array initializer:

```
var array3x4x5 = Array(repeating: Array(repeating: Array(repeating: 0,count: 5),count: 4),count: 3)
```

Section 6.2: Extracting values of a given type from an Array with flatMap(_:)

The `things` Array contains values of `Any` type.

```
let things: [Any] = [1, "Hello", 2, true, false, "World", 3]
```

We can extract values of a given type and create a new Array of that specific type. Let's say we want to extract all the `Int`(s) and put them into an `Int` Array in a safe way.

```
let numbers = things.flatMap { $0 as? Int }
```

Now `numbers` is defined as [`Int`]. The `flatMap` function discard all `nil` elements and the result thus contains only the following values:

```
[1, 2, 3]
```

Section 6.3: Combining an Array's elements with reduce(_:combine:)

`reduce(_:combine:)` can be used in order to combine the elements of a sequence into a single value. It takes an initial value for the result, as well as a closure to apply to each element – which will return the new accumulated value.

For example, we can use it to sum an array of numbers:

```
let numbers = [2, 5, 7, 8, 10, 4]

let sum = numbers.reduce(0) {accumulator, element in
    return accumulator + element
}

print(sum) // 36
```

We're passing `0` into the initial value, as that's the logical initial value for a summation. If we passed in a value of `N`, the resulting `sum` would be `N + 36`. The closure passed to `reduce` has two arguments. `accumulator` is the current accumulated value, which is assigned the value that the closure returns at each iteration. `element` is the current element in the iteration.

As in this example, we're passing an `(Int, Int) -> Int` closure to `reduce`, which is simply outputting the addition of the two inputs – we can actually pass in the `+` operator directly, as operators are functions in Swift:

```
let sum = numbers.reduce(0, combine: +)
```

第6.4节：使用flatMap(_:)扁平化数组转换的结果

除了可以通过过滤序列转换后的元素中的nil来创建数组外，
还有一个版本的flatMap(_:)期望转换闭包返回一个序列S。

```
extension SequenceType {
    public func flatMap<S : SequenceType>(transform: (Self.Generator.Element) throws -> S) rethrows
-> [S.Generator.Element]
}
```

转换后的每个序列将被连接，生成一个包含每个序列组合元素的数组-[S.Generator.Element]。

将字符串数组中的字符合并

例如，我们可以用它来获取一个质数字符串数组，并将它们的字符合并成一个单一的数组：

```
let primes = ["2", "3", "5", "7", "11", "13", "17", "19"]
let allCharacters = primes.flatMap { $0.characters }
// => "["2", "3", "5", "7", "1", "1", "1", "3", "1", "7", "1", "9"]"
```

分解上述示例：

1. primes 是一个 [String]（由于数组是一个序列，我们可以对其调用 flatMap(_:)）。
2. 转换闭包接收 primes 中的一个元素，即一个 String (Array<String>.Generator.Element)。
3. 然后闭包返回一个类型为 String.CharacterView 的序列。
4. 结果是一个数组，包含来自每个转换闭包调用的所有序列的组合元素-[String.CharacterView.Generator.Element]。

扁平化多维数组

由于flatMap(_:)会连接转换闭包调用返回的序列，因此它可以用来扁平化多维数组-例如将二维数组转换为一维数组，三维数组转换为二维数组等。

这可以通过在闭包中返回给定元素$0（一个嵌套数组）来简单实现：

```
// 一个类型为[[Int]]的二维数组
let array2D = [[1, 3], [4], [6, 8, 10], [11]]

// 一个类型为[Int]的一维数组
let flattenedArray = array2D.flatMap { $0 }

print(flattenedArray) // [1, 3, 4, 6, 8, 10, 11]
```

第6.5节：使用flatten()懒惰地扁平化多维数组

我们可以使用flatten()来懒惰地减少多维序列的嵌套。

例如，将二维数组懒惰地扁平化为一维数组：

```
// 一个二维数组，类型为 [[Int]]
let array2D = [[1, 3], [4], [6, 8, 10], [11]]
```


Section 6.4: Flattening the result of an Array transformation with flatMap(_:)

As well as being able to create an array by filtering out nil from the transformed elements of a sequence, there is also a version of flatMap(_:) that expects the transformation closure to return a sequence S.

```
extension SequenceType {
    public func flatMap<S : SequenceType>(transform: (Self.Generator.Element) throws -> S) rethrows
-> [S.Generator.Element]
}
```

Each sequence from the transformation will be concatenated, resulting in an array containing the combined elements of each sequence – [S.Generator.Element].

Combining the characters in an array of strings

For example, we can use it to take an array of prime strings and combine their characters into a single array:

```
let primes = ["2", "3", "5", "7", "11", "13", "17", "19"]
let allCharacters = primes.flatMap { $0.characters }
// => "["2", "3", "5", "7", "1", "1", "1", "3", "1", "7", "1", "9"]"
```

Breaking the above example down:

1. primes is a [String] (As an array is a sequence, we can call flatMap(_:) on it).
2. The transformation closure takes in one of the elements of primes, a String (Array<String>.Generator.Element).
3. The closure then returns a sequence of type String.CharacterView.
4. The result is then an array containing the combined elements of all the sequences from each of the transformation closure calls – [String.CharacterView.Generator.Element].

Flattening a multidimensional array

As flatMap(_:) will concatenate the sequences returned from the transformation closure calls, it can be used to flatten a multidimensional array – such as a 2D array into a 1D array, a 3D array into a 2D array etc.

This can simply be done by returning the given element $0 (a nested array) in the closure:

```
// A 2D array of type [[Int]]
let array2D = [[1, 3], [4], [6, 8, 10], [11]]

// A 1D array of type [Int]
let flattenedArray = array2D.flatMap { $0 }

print(flattenedArray) // [1, 3, 4, 6, 8, 10, 11]
```

Section 6.5: Lazily flattening a multidimensional Array with flatten()

We can use flatten() in order to lazily reduce the nesting of a multi-dimensional sequence.

For example, lazy flattening a 2D array into a 1D array:

```
// A 2D array of type [[Int]]
let array2D = [[1, 3], [4], [6, 8, 10], [11]]
```

```
// 一个 FlattenBidirectionalCollection<[[Int]]>
let lazilyFlattenedArray = array2D.flatten()

print(lazilyFlattenedArray.contains(4)) // true
```

在上述示例中，flatten() 将返回一个 FlattenBidirectionalCollection，它会惰性地应用数组的展平操作。因此 contains(_:) 只需要展平 array2D 的前两个嵌套数组-因为它在找到所需元素后会短路。

第6.6节：使用 flatMap(_:) 过滤数组中的 nil

你可以以类似于 map(_:) 的方式使用 flatMap(_:)，通过对序列元素应用转换来创建一个数组。

```
extension SequenceType {
    public func flatMap<T>(@noescape transform: (Self.Generator.Element) throws -> T?) rethrows ->
[T]
}
```

这个版本的 flatMap(_:) 的区别在于它期望转换闭包返回一个可选值 T? 对每个元素。然后它会安全地解包这些可选值，过滤掉 nil -最终得到一个 [T] 类型的数组。

例如，你可以用它将一个 [String] 转换成 [Int]，使用 Int 的可失败 String 初始化器，过滤掉无法转换的元素：

```
let strings = ["1", "foo", "3", "4", "bar", "6"]

let numbersThatCanBeConverted = strings.flatMap { Int($0) }

print(numbersThatCanBeConverted) // [1, 3, 4, 6]
```

你也可以利用 flatMap(_:) 过滤 nil 的能力，将一个可选值数组简单地转换成非可选值数组：

```
let optionalNumbers : [Int?] = [nil, 1, nil, 2, nil, 3]

let numbers = optionalNumbers.flatMap { $0 }

print(numbers) // [1, 2, 3]
```

第6.7节：使用范围下标访问数组

可以使用范围（Range）从数组中提取一系列连续的元素。

```
let words = ["Hey", "Hello", "Bonjour", "Welcome", "Hi", "Hola"]
let range = 2...4
let slice = words[range] // ["Bonjour", "Welcome", "Hi"]
```

使用范围下标访问数组返回一个ArraySlice。它是数组的一个子序列。

在我们的例子中，数组元素是字符串，因此返回的是ArraySlice<String>。

虽然ArraySlice遵循CollectionType协议，可以与sort、filter等方法一起使用，但它的目的不是用于


```
// A FlattenBidirectionalCollection<[[Int]]>
let lazilyFlattenedArray = array2D.flatten()

print(lazilyFlattenedArray.contains(4)) // true
```

In the above example, `flatten()` will return a `FlattenBidirectionalCollection`, which will lazily apply the flattening of the array. Therefore `contains(_:)` will only require the first two nested arrays of `array2D` to be flattened – as it will short-circuit upon finding the desired element.

Section 6.6: Filtering out nil from an Array transformation with flatMap(_:)

You can use `flatMap(_:)` in a similar manner to `map(_:)` in order to create an array by applying a transform to a sequence's elements.

```
extension SequenceType {
    public func flatMap<T>(@noescape transform: (Self.Generator.Element) throws -> T?) rethrows ->
[T]
}
```

The difference with this version of `flatMap(_:)` is that it expects the transform closure to return an Optional value `T?` for each of the elements. It will then safely unwrap each of these optional values, filtering out `nil` – resulting in an array of `[T]`.

For example, you can this in order to transform a `[String]` into a `[Int]` using `Int`'s failable `String` initializer, filtering out any elements that cannot be converted:

```
let strings = ["1", "foo", "3", "4", "bar", "6"]

let numbersThatCanBeConverted = strings.flatMap { Int($0) }

print(numbersThatCanBeConverted) // [1, 3, 4, 6]
```

You can also use `flatMap(_:)`'s ability to filter out `nil` in order to simply convert an array of optionals into an array of non-optionals:

```
let optionalNumbers : [Int?] = [nil, 1, nil, 2, nil, 3]

let numbers = optionalNumbers.flatMap { $0 }

print(numbers) // [1, 2, 3]
```

Section 6.7: Subscripting an Array with a Range

One can extract a series of consecutive elements from an Array using a Range.

```
let words = ["Hey", "Hello", "Bonjour", "Welcome", "Hi", "Hola"]
let range = 2...4
let slice = words[range] // ["Bonjour", "Welcome", "Hi"]
```

Subscripting an Array with a Range returns an `ArraySlice`. It's a subsequence of the Array.

In our example, we have an Array of Strings, so we get back `ArraySlice<String>`.

Although an ArraySlice conforms to `CollectionType` and can be used with `sort`, `filter`, etc, its purpose is not for

长期存储，而是用于临时计算：在完成操作后，应尽快将其转换回数组。

为此，使用Array()初始化器：

```
let result = Array(slice)
```

简单示例总结，无需中间步骤：

```
let words = ["Hey", "Hello", "Bonjour", "Welcome", "Hi", "Hola"]
let selectedWords = Array(words[2...4]) // ["Bonjour", "Welcome", "Hi"]
```

第6.8节：在不知道索引的情况下从数组中移除元素

通常，如果我们想从数组中移除一个元素，需要知道它的索引，这样才能使用remove(at:)函数轻松移除。

但是如果我们不知道索引，只知道要移除元素的值怎么办！

这里有一个简单的数组扩展，可以让我们在不知道索引的情况下轻松移除数组中的元素：

Swift3

```
extension Array where Element: Equatable {

    mutating func remove(_ element: Element) {
        _ = index(of: element).flatMap {
            self.remove(at: $0)
        }
    }
}
```

例如。

```
    var array = ["abc", "lmn", "pqr", "stu", "xyz"]
    array.remove("lmn")
    print("\(array)")    //["abc", "pqr", "stu", "xyz"]

    array.remove("nonexistent")
    print("\(array)")    //["abc", "pqr", "stu", "xyz"]
    //如果提供的元素值不存在，则不会执行任何操作！
```

另外，如果我们不小心这样做了：`array.remove(25)`即我们提供了不同数据类型的值，
编译器会抛出错误，提示——
`无法将值转换为预期的参数类型`

第6.9节：字符串数组排序

版本 = 3.0

最简单的方法是使用 sorted() ：

```
let words = ["Hello", "Bonjour", "Salute", "Ahola"]
let sortedWords = words.sorted()
print(sortedWords) // ["Ahola", "Bonjour", "Hello", "Salute"]
```


long-term storage but for transient computations: it should be converted back into an Array as soon as you've finished working with it.

For this, use the `Array()` initializer:

```
let result = Array(slice)
```

To sum up in a simple example without intermediary steps:

```
let words = ["Hey", "Hello", "Bonjour", "Welcome", "Hi", "Hola"]
let selectedWords = Array(words[2...4]) // ["Bonjour", "Welcome", "Hi"]
```

Section 6.8: Removing element from an array without knowing it's index

Generally, if we want to remove an element from an array, we need to know it's index so that we can remove it easily using `remove(at:)` function.

But what if we don't know the index but we know the value of element to be removed!

So here is the simple extension to an array which will allow us to remove an element from array easily without knowing it's index:

Swift3

```
extension Array where Element: Equatable {

    mutating func remove(_ element: Element) {
        _ = index(of: element).flatMap {
            self.remove(at: $0)
        }
    }
}
```

e.g.

```
    var array = ["abc", "lmn", "pqr", "stu", "xyz"]
    array.remove("lmn")
    print("\(array)")    //["abc", "pqr", "stu", "xyz"]

    array.remove("nonexistent")
    print("\(array)")    //["abc", "pqr", "stu", "xyz"]
    //if provided element value is not present, then it will do nothing!
```

Also if, by mistake, we did something like this: `array.remove(25)` i.e. we provided value with different data type, compiler will throw an error mentioning-
`cannot convert value to expected argument type`

Section 6.9: Sorting an Array of Strings

Version = 3.0

The most simple way is to use `sorted()`:

```
let words = ["Hello", "Bonjour", "Salute", "Ahola"]
let sortedWords = words.sorted()
print(sortedWords) // ["Ahola", "Bonjour", "Hello", "Salute"]
```

或 sort()

```
var mutableWords = ["Hello", "Bonjour", "Salute", "Ahola"]
mutableWords.sort()
print(mutableWords) // ["Ahola", "Bonjour", "Hello", "Salute"]
```

你可以传递一个闭包作为排序的参数：

```
let words = ["Hello", "Bonjour", "Salute", "Ahola"]
let sortedWords = words.sorted(isOrderedBefore: { $0 > $1 })
print(sortedWords) // ["Salute", "Hello", "Bonjour", "Ahola"]
```

使用尾随闭包的另一种语法：

```
let words = ["Hello", "Bonjour", "Salute", "Ahola"]
let sortedWords = words.sorted() { $0 > $1 }
print(sortedWords) // ["Salute", "Hello", "Bonjour", "Ahola"]
```

但是如果数组中的元素不一致，结果会出乎意料：

```
let words = ["Hello", "bonjour", "Salute", "ahola"]
let unexpected = words.sorted()
print(unexpected) // ["Hello", "Salute", "ahola", "bonjour"]
```

为了解决这个问题，可以对元素的小写版本进行排序：

```
let words = ["Hello", "bonjour", "Salute", "ahola"]
let sortedWords = words.sorted { $0.lowercased() < $1.lowercased() }
print(sortedWords) // ["ahola", "bonjour", "Hello", "Salute"]
```

或者import Foundation 并使用 NSString 的比较方法，如caseInsensitiveCompare：

```
let words = ["Hello", "bonjour", "Salute", "ahola"]
let sortedWords = words.sorted { $0.caseInsensitiveCompare($1) == .orderedAscending }
print(sortedWords) // ["ahola", "bonjour", "Hello", "Salute"]
```

或者，使用localizedCaseInsensitiveCompare，它可以处理变音符号。

要根据字符串中包含的数字值正确排序，请使用带有.numeric选项的compare：

```
let files = ["File-42.txt", "File-01.txt", "File-5.txt", "File-007.txt", "File-10.txt"]
let sortedFiles = files.sorted() { $0.compare($1, options: .numeric) == .orderedAscending }
print(sortedFiles) // ["File-01.txt", "File-5.txt", "File-007.txt", "File-10.txt", "File-42.txt"]
```

第6.10节：安全访问索引

通过向数组添加以下扩展，可以在不知道索引是否越界的情况下访问索引。

```
extension Array {
    下标 (安全索引: Int) -> 元素? {
        返回 indices ~= index ? self[index] : nil
    }
}
```

示例：

or sort()

```
var mutableWords = ["Hello", "Bonjour", "Salute", "Ahola"]
mutableWords.sort()
print(mutableWords) // ["Ahola", "Bonjour", "Hello", "Salute"]
```

You can pass a closure as an argument for sorting:

```
let words = ["Hello", "Bonjour", "Salute", "Ahola"]
let sortedWords = words.sorted(isOrderedBefore: { $0 > $1 })
print(sortedWords) // ["Salute", "Hello", "Bonjour", "Ahola"]
```

Alternative syntax with a trailing closure:

```
let words = ["Hello", "Bonjour", "Salute", "Ahola"]
let sortedWords = words.sorted() { $0 > $1 }
print(sortedWords) // ["Salute", "Hello", "Bonjour", "Ahola"]
```

But there will be unexpected results if the elements in the array are not consistent:

```
let words = ["Hello", "bonjour", "Salute", "ahola"]
let unexpected = words.sorted()
print(unexpected) // ["Hello", "Salute", "ahola", "bonjour"]
```

To address this issue, either sort on a lowercase version of the elements:

```
let words = ["Hello", "bonjour", "Salute", "ahola"]
let sortedWords = words.sorted { $0.lowercased() < $1.lowercased() }
print(sortedWords) // ["ahola", "bonjour", "Hello", "Salute"]
```

Or import Foundation and use NSString's comparison methods like caseInsensitiveCompare:

```
let words = ["Hello", "bonjour", "Salute", "ahola"]
let sortedWords = words.sorted { $0.caseInsensitiveCompare($1) == .orderedAscending }
print(sortedWords) // ["ahola", "bonjour", "Hello", "Salute"]
```

Alternatively, use localizedCaseInsensitiveCompare, which can manage diacritics.

To properly sort Strings by the *numeric* value they contain, use compare with the .numeric option:

```
let files = ["File-42.txt", "File-01.txt", "File-5.txt", "File-007.txt", "File-10.txt"]
let sortedFiles = files.sorted() { $0.compare($1, options: .numeric) == .orderedAscending }
print(sortedFiles) // ["File-01.txt", "File-5.txt", "File-007.txt", "File-10.txt", "File-42.txt"]
```

Section 6.10: Accessing indices safely

By adding the following extension to array indices can be accessed without knowing if the index is inside bounds.

```
extension Array {
    subscript (safe index: Int) -> Element? {
        return indices ~= index ? self[index] : nil
    }
}
```

example:

```
如果让 thirdValue = array[safe: 2] {
    打印(thirdValue)
}
```

第6.11节：过滤数组

你可以在SequenceType上使用filter(_:)方法来创建一个新数组，包含满足给定谓词的序列元素，该谓词可以作为闭包提供。

例如，从[Int]中过滤偶数：

```
让 numbers = [22, 41, 23, 30]

让 evenNumbers = numbers.filter { $0 % 2 == 0 }

打印(evenNumbers)  // [22, 30]
```

过滤一个[Person]数组，其中年龄小于30：

```
结构体 Person {
    变量 age：Int
}

让 people = [Person(age: 22), Person(age: 41), Person(age: 23), Person(age: 30)]

让 peopleYoungerThan30 = people.filter { $0.age < 30 }

print(peopleYoungerThan30) // [Person(age: 22), Person(age: 23)]
```

第6.12节：使用map(_:)转换数组元素

由于Array遵循SequenceType，我们可以使用map(_:)通过一个类型为(A) throws -> B的闭包将A数组转换为B数组。

例如，我们可以用它将一个Int数组转换为String数组，方法如下：

```
let numbers = [1, 2, 3, 4, 5]
let words = numbers.map { String($0) }
print(words) // ["1", "2", "3", "4", "5"]
```

map(_:)会遍历数组，对每个元素应用给定的闭包。该闭包的结果将用于填充一个包含转换后元素的新数组。

由于String有一个接收Int的初始化方法，我们也可以使用更简洁的语法：

```
let words = numbers.map(String.init)
```

一个map(_:)转换不一定要改变数组的类型——例如，它也可以用来将一个Int数组中的元素乘以二：

```
let numbers = [1, 2, 3, 4, 5]
let numbersTimes2 = numbers.map {$0 * 2}
print(numbersTimes2) // [2, 4, 6, 8, 10]
```



```
if let thirdValue = array[safe: 2] {
    print(thirdValue)
}
```

Section 6.11: Filtering an Array

You can use the `filter(_:)` method on `SequenceType` in order to create a new array containing the elements of the sequence that satisfy a given predicate, which can be provided as a closure.

For example, filtering even numbers from an `[Int]`:

```
let numbers = [22, 41, 23, 30]

let evenNumbers = numbers.filter { $0 % 2 == 0 }

print(evenNumbers)  // [22, 30]
```

Filtering a `[Person]`, where their age is less than 30:

```
struct Person {
    var age : Int
}

let people = [Person(age: 22), Person(age: 41), Person(age: 23), Person(age: 30)]

let peopleYoungerThan30 = people.filter { $0.age < 30 }

print(peopleYoungerThan30) // [Person(age: 22), Person(age: 23)]
```

Section 6.12: Transforming the elements of an Array with map(_:)

As `Array` conforms to `SequenceType`, we can use `map(_:)` to transform an array of `A` into an array of `B` using a closure of type `(A) throws -> B`.

For example, we could use it to transform an array of `Int`s into an array of `String`s like so:

```
let numbers = [1, 2, 3, 4, 5]
let words = numbers.map { String($0) }
print(words) // ["1", "2", "3", "4", "5"]
```

`map(_:)` will iterate through the array, applying the given closure to each element. The result of that closure will be used to populate a new array with the transformed elements.

Since `String` has an initialiser that receives an `Int` we can also use this clearer syntax:

```
let words = numbers.map(String.init)
```

A `map(_:)` transform need not change the type of the array – for example, it could also be used to multiply an array of `Int`s by two:

```
let numbers = [1, 2, 3, 4, 5]
let numbersTimes2 = numbers.map {$0 * 2}
print(numbersTimes2) // [2, 4, 6, 8, 10]
```

第6.13节：实用方法

确定数组是否为空或返回其大小

```
var exampleArray = [1,2,3,4,5]
exampleArray.isEmpty //false
exampleArray.count //5
```

反转数组 注意：结果不会直接作用于调用该方法的数组，必须将结果赋值给一个新的变量。

```
exampleArray = exampleArray.reverse()
//exampleArray = [9, 8, 7, 6, 5, 3, 2]
```

第6.14节：数组排序

```
var array = [3, 2, 1]
```

创建一个新的排序数组

由于Array遵循SequenceType协议，我们可以使用内置的排序方法生成一个排序后的新数组。

版本 = 2.1 版本 = 2.2

在Swift 2中，这是通过 sort()方法完成的。

```
let sorted = array.sort()  // [1, 2, 3]
```

版本 ≥ 3.0

从 Swift 3 开始，它被重命名为 sorted()。

```
let sorted = array.sorted()  // [1, 2, 3]
```

就地排序现有数组

由于 Array 遵循 MutableCollectionType 协议，我们可以就地对其元素进行排序。

版本 = 2.1 版本 = 2.2

在 Swift 2 中，这是通过 sortInPlace() 方法完成的。

```
array.sortInPlace() // [1, 2, 3]
```

版本 ≥ 3.0

从 Swift 3 开始，它被重命名为 sort()。

```
array.sort() // [1, 2, 3]
```

> 注意：为了使用上述方法，元素必须遵循 Comparable 协议。

使用自定义排序对数组进行排序

你也可以使用闭包来排序数组，以定义一个元素是否应该排在另一个元素之前–

Section 6.13: Useful Methods

Determine whether an array is empty or return its size

```
var exampleArray = [1,2,3,4,5]
exampleArray.isEmpty //false
exampleArray.count //5
```

Reverse an Array **Note: The result is not performed on the array the method is called on and must be put into its own variable.**

```
exampleArray = exampleArray.reverse()
//exampleArray = [9, 8, 7, 6, 5, 3, 2]
```

Section 6.14: Sorting an Array

```
var array = [3, 2, 1]
```

Creating a new sorted array

As `Array` conforms to `SequenceType`, we can generate a new array of the sorted elements using a built in sort method.

Version = 2.1 Version = 2.2

In Swift 2, this is done with the `sort()` method.

```
let sorted = array.sort()  // [1, 2, 3]
```

Version ≥ 3.0

As of Swift 3, it has been re-named to `sorted()`.

```
let sorted = array.sorted()  // [1, 2, 3]
```

Sorting an existing array in place

As `Array` conforms to `MutableCollectionType`, we can sort its elements in place.

Version = 2.1 Version = 2.2

In Swift 2, this is done using the `sortInPlace()` method.

```
array.sortInPlace() // [1, 2, 3]
```

Version ≥ 3.0

As of Swift 3, it has been renamed to `sort()`.

```
array.sort() // [1, 2, 3]
```

> Note: In order to use the above methods, the elements must conform to the `Comparable` protocol.

Sorting an array with a custom ordering

You may also sort an array using a closure to define whether one element should be ordered before another –

这不限于元素必须是Comparable的数组。例如，对于一个
地标（Landmark）来说，成为Comparable是没有意义的–但你仍然可以按高度或名称对地标数组进行排序。

```
结构体 Landmark {
    let name : String
    let metersTall : Int
}

var landmarks = [Landmark(name: "帝国大厦", metersTall: 443),
                 Landmark(name: "埃菲尔铁塔", metersTall: 300),
Landmark(name: "碎片大厦", metersTall: 310)]
```

版本 = 2.1 版本 = 2.2

```
// 按高度（升序）排序地标
landmarks.sortInPlace {$0.metersTall < $1.metersTall}

print(landmarks) // [Landmark(name: "埃菲尔铁塔", metersTall: 300), Landmark(name: "碎片大厦",
metersTall: 310), Landmark(name: "帝国大厦", metersTall: 443)]

// 创建一个按名称排序的新地标数组
let alphabeticalLandmarks = landmarks.sort {$0.name < $1.name}

print(alphabeticalLandmarks) // [Landmark(name: "埃菲尔铁塔", metersTall: 300), Landmark(name:
"帝国大厦", metersTall: 443), Landmark(name: "碎片大厦", metersTall: 310)]
```

版本 ≥ 3.0

```
// 按高度（升序）排序地标
landmarks.sort {$0.metersTall < $1.metersTall}

// 创建一个按名称排序的新地标数组
let alphabeticalLandmarks = landmarks.sorted {$0.name < $1.name}
```

> 注意：如果字符串不一致，字符串比较可能会产生意想不到的结果，详见字符串数组排序。

第6.15节：查找数组中的最小或最大元素

版本 = 2.1 版本 = 2.2

你可以使用minElement()和maxElement()方法来查找给定序列中的最小或最大元素。例如，对于一个数字数组：

```
let numbers = [2, 6, 1, 25, 13, 7, 9]

let minimumNumber = numbers.minElement() // Optional(1)
let maximumNumber = numbers.maxElement() // Optional(25)
```

版本 ≥ 3.0

从Swift 3开始，这些方法分别重命名为min()和max()：

```
let minimumNumber = numbers.min() // Optional(1)
let maximumNumber = numbers.max() // Optional(25)
```

这些方法返回的值是可选的，以反映数组可能为空的情况–如果为空，
将返回nil。

which isn't restricted to arrays where the elements must be `Comparable`. For example, it doesn't make sense for a `Landmark` to be `Comparable` – but you can still sort an array of landmarks by height or name.

```
struct Landmark {
    let name : String
    let metersTall : Int
}

var landmarks = [Landmark(name: "Empire State Building", metersTall: 443),
                 Landmark(name: "Eifell Tower", metersTall: 300),
                 Landmark(name: "The Shard", metersTall: 310)]
```

Version = 2.1 Version = 2.2

```
// sort landmarks by height (ascending)
landmarks.sortInPlace {$0.metersTall < $1.metersTall}

print(landmarks) // [Landmark(name: "Eifell Tower", metersTall: 300), Landmark(name: "The Shard",
metersTall: 310), Landmark(name: "Empire State Building", metersTall: 443)]

// create new array of landmarks sorted by name
let alphabeticalLandmarks = landmarks.sort {$0.name < $1.name}

print(alphabeticalLandmarks) // [Landmark(name: "Eifell Tower", metersTall: 300), Landmark(name:
"Empire State Building", metersTall: 443), Landmark(name: "The Shard", metersTall: 310)]
```

Version ≥ 3.0

```
// sort landmarks by height (ascending)
landmarks.sort {$0.metersTall < $1.metersTall}

// create new array of landmarks sorted by name
let alphabeticalLandmarks = landmarks.sorted {$0.name < $1.name}
```

> Note: String comparison can yield unexpected results if the strings are inconsistent, see Sorting an Array of Strings.

Section 6.15: Finding the minimum or maximum element of an Array

Version = 2.1 Version = 2.2

You can use the `minElement()` and `maxElement()` methods to find the minimum or maximum element in a given sequence. For example, with an array of numbers:

```
let numbers = [2, 6, 1, 25, 13, 7, 9]

let minimumNumber = numbers.minElement() // Optional(1)
let maximumNumber = numbers.maxElement() // Optional(25)
```

Version ≥ 3.0

As of Swift 3, the methods have been renamed to `min()` and `max()` respectively:

```
let minimumNumber = numbers.min() // Optional(1)
let maximumNumber = numbers.max() // Optional(25)
```

The returned values from these methods are Optional to reflect the fact that the array could be empty – if it is, `nil` will be returned.

> 注意：上述方法要求元素遵循Comparable协议。

使用自定义排序查找最小或最大元素

你也可以使用上述方法并传入自定义闭包，定义一个元素是否应该排在另一个元素之前，从而在元素不一定遵循Comparable协议的数组中找到最小或最大元素。

例如，对于一个向量数组：

```
struct Vector2 {
    let dx : Double
    let dy : Double

    var magnitude : Double {return sqrt(dx*dx+dy*dy)}
}

let vectors = [Vector2(dx: 3, dy: 2), Vector2(dx: 1, dy: 1), Vector2(dx: 2, dy: 2)]
```

版本 = 2.1 版本 = 2.2

```
// Vector2(dx: 1.0, dy: 1.0)
let lowestMagnitudeVec2 = vectors.minElement { $0.magnitude < $1.magnitude }

// Vector2(dx: 3.0, dy: 2.0)
let highestMagnitudeVec2 = vectors.maxElement { $0.magnitude < $1.magnitude }
```

版本 ≥ 3.0

```
let lowestMagnitudeVec2 = vectors.min { $0.magnitude < $1.magnitude }
let highestMagnitudeVec2 = vectors.max { $0.magnitude < $1.magnitude }
```

第6.16节：修改数组中的值

有多种方法可以向数组追加值

```
var exampleArray = [1,2,3,4,5]
exampleArray.append(6)
//exampleArray = [1, 2, 3, 4, 5, 6]
var sixOnwards = [7,8,9,10]
exampleArray += sixOnwards
//exampleArray = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

以及从数组中移除值

```
exampleArray.removeAtIndex(3)
//exampleArray = [1, 2, 3, 5, 6, 7, 8, 9, 10]
exampleArray.removeLast()
//exampleArray = [1, 2, 3, 5, 6, 7, 8, 9]
exampleArray.removeFirst()
//exampleArray = [2, 3, 5, 6, 7, 8, 9]
```

第6.17节：使用zip比较两个数组

zip函数接受两个类型为SequenceType的参数，并返回一个Zip2Sequence，其中每个元素包含来自第一个序列和第二个序列的一个值。

示例

> Note: The above methods require the elements to conform to the `Comparable` protocol.

Finding the minimum or maximum element with a custom ordering

You may also use the above methods with a custom closure, defining whether one element should be ordered before another, allowing you to find the minimum or maximum element in an array where the elements aren't necessarily `Comparable`.

For example, with an array of vectors:

```
struct Vector2 {
    let dx : Double
    let dy : Double

    var magnitude : Double {return sqrt(dx*dx+dy*dy)}
}

let vectors = [Vector2(dx: 3, dy: 2), Vector2(dx: 1, dy: 1), Vector2(dx: 2, dy: 2)]
```

Version = 2.1 Version = 2.2

```
// Vector2(dx: 1.0, dy: 1.0)
let lowestMagnitudeVec2 = vectors.minElement { $0.magnitude < $1.magnitude }

// Vector2(dx: 3.0, dy: 2.0)
let highestMagnitudeVec2 = vectors.maxElement { $0.magnitude < $1.magnitude }
```

Version ≥ 3.0

```
let lowestMagnitudeVec2 = vectors.min { $0.magnitude < $1.magnitude }
let highestMagnitudeVec2 = vectors.max { $0.magnitude < $1.magnitude }
```

Section 6.16: Modifying values in an array

There are multiple ways to append values onto an array

```
var exampleArray = [1,2,3,4,5]
exampleArray.append(6)
//exampleArray = [1, 2, 3, 4, 5, 6]
var sixOnwards = [7,8,9,10]
exampleArray += sixOnwards
//exampleArray = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

and remove values from an array

```
exampleArray.removeAtIndex(3)
//exampleArray = [1, 2, 3, 5, 6, 7, 8, 9, 10]
exampleArray.removeLast()
//exampleArray = [1, 2, 3, 5, 6, 7, 8, 9]
exampleArray.removeFirst()
//exampleArray = [2, 3, 5, 6, 7, 8, 9]
```

Section 6.17: Comparing 2 Arrays with zip

The `zip` function accepts 2 parameters of type `SequenceType` and returns a `Zip2Sequence` where each element contains a value from the first sequence and one from the second sequence.

Example

```
let nums = [1, 2, 3]
let animals = ["Dog", "Cat", "Tiger"]
let numsAndAnimals = zip(nums, animals)
```

numsAndAnimals现在包含以下值

sequence1	sequence1
1	"Dog"
2	"Cat"
3	"Tiger"

当你想对每个数组的第n个元素进行某种比较时，这非常有用。

示例

给定两个Int数组

```
let list0 = [0, 2, 4]
let list1 = [0, 4, 8]
```

你想检查list1中的每个值是否是list0中对应值的两倍。

```
let list1HasDoubleOfList0 = !zip(list0, list1).filter { $0 != (2 * $1)}.isEmpty
```

第6.18节：数组值分组

如果我们有如下结构体

```
struct Box {
    let name: String
    let thingsInside: Int
}
```

以及一个Box数组

```
let boxes = [
Box(name: "Box 0", thingsInside: 1),
    Box(name: "Box 1", thingsInside: 2),
    Box(name: "Box 2", thingsInside: 3),
    Box(name: "Box 3", thingsInside: 1),
    Box(name: "Box 4", thingsInside: 2),
    Box(name: "Box 5", thingsInside: 3),
    Box(name: "Box 6", thingsInside: 1)
]
```

我们可以根据thingsInside属性对盒子进行分组，以获得一个Dictionary，其中key是物品的数量，value是盒子数组。

```
let grouped = boxes.reduce([Int:[Box]]()) { (res, box) -> [Int:[Box]] in
    var res = res
res[box.thingsInside] = (res[box.thingsInside] ?? []) + [box]
    return res
}
```

现在 grouped 是一个 [Int:[Box]]，内容如下


```
let nums = [1, 2, 3]
let animals = ["Dog", "Cat", "Tiger"]
let numsAndAnimals = zip(nums, animals)
```

nomsAndAnimals now contains the following values

sequence1	sequence1
1	"Dog"
2	"Cat"
3	"Tiger"

This is useful when you want to perform some kind of comparation between the n-th element of each Array.

Example

Given 2 Arrays of `Int(s)`

```
let list0 = [0, 2, 4]
let list1 = [0, 4, 8]
```

you want to check whether each value into `list1` is the double of the related value in `list0`.

```
let list1HasDoubleOfList0 = !zip(list0, list1).filter { $0 != (2 * $1)}.isEmpty
```

Section 6.18: Grouping Array values

If we have a struct like this

```
struct Box {
    let name: String
    let thingsInside: Int
}
```

and an array of `Box(es)`

```
let boxes = [
    Box(name: "Box 0", thingsInside: 1),
    Box(name: "Box 1", thingsInside: 2),
    Box(name: "Box 2", thingsInside: 3),
    Box(name: "Box 3", thingsInside: 1),
    Box(name: "Box 4", thingsInside: 2),
    Box(name: "Box 5", thingsInside: 3),
    Box(name: "Box 6", thingsInside: 1)
]
```

we can group the boxes by the `thingsInside` property in order to get a `Dictionary` where the key is the number of things and the value is an array of boxes.

```
let grouped = boxes.reduce([Int:[Box]]()) { (res, box) -> [Int:[Box]] in
    var res = res
    res[box.thingsInside] = (res[box.thingsInside] ?? []) + [box]
    return res
}
```

Now grouped is a `[Int:[Box]]` and has the following content

```
[
    2: [Box(name: "Box 1", thingsInside: 2), Box(name: "Box 4", thingsInside: 2)],
    3: [Box(name: "Box 2", thingsInside: 3), Box(name: "Box 5", thingsInside: 3)],
    1: [Box(name: "Box 0", thingsInside: 1), Box(name: "Box 3", thingsInside: 1), Box(name: "Box
6", thingsInside: 1)]
]
```

第6.19节：值语义

复制数组会复制原数组中的所有项目。

更改新数组不会更改原数组。

```
var originalArray = ["Swift", "is", "great!"]
var newArray = originalArray
newArray[2] = "awesome!"
//originalArray = ["Swift", "is", "great!"]
//newArray = ["Swift", "is", "awesome!"]
```

复制的数组在被修改之前会与原数组共享同一块内存空间。因此，当复制的数组第一次被修改时，为其分配独立内存空间会带来性能开销。

第6.20节：访问数组值

以下示例将使用此数组演示如何访问值

```
var exampleArray:[Int] = [1,2,3,4,5]
//exampleArray = [1, 2, 3, 4, 5]
```

要访问已知索引处的值，请使用以下语法：

```
let exampleOne = exampleArray[2]
//exampleOne = 3
```

注意： 索引为2的值是数组中的第三个值。数组使用从零开始的索引，这意味着数组中的第一个元素索引为0。

```
let value0 = exampleArray[0]
let value1 = exampleArray[1]
let value2 = exampleArray[2]
let value3 = exampleArray[3]
let value4 = exampleArray[4]
//value0 = 1
//value1 = 2
//value2 = 3
//value3 = 4
//value4 = 5
```

使用过滤器访问数组的子集：

```
var filteredArray = exampleArray.filter({ $0 < 4 })
//filteredArray = [1, 2, 3]
```

过滤器可以有复杂的条件，比如只过滤偶数：

```
var evenArray = exampleArray.filter({ $0 % 2 == 0 })
```



```
[
    2: [Box(name: "Box 1", thingsInside: 2), Box(name: "Box 4", thingsInside: 2)],
    3: [Box(name: "Box 2", thingsInside: 3), Box(name: "Box 5", thingsInside: 3)],
    1: [Box(name: "Box 0", thingsInside: 1), Box(name: "Box 3", thingsInside: 1), Box(name: "Box
6", thingsInside: 1)]
]
```

Section 6.19: Value Semantics

Copying an array will copy all of the items inside the original array.

Changing the new array *will not change* the original array.

```
var originalArray = ["Swift", "is", "great!"]
var newArray = originalArray
newArray[2] = "awesome!"
//originalArray = ["Swift", "is", "great!"]
//newArray = ["Swift", "is", "awesome!"]
```

Copied arrays will share the same space in memory as the original until they are changed. As a result of this there is a performance hit when the copied array is given its own space in memory as it is changed for the first time.

Section 6.20: Accessing Array Values

The following examples will use this array to demonstrate accessing values

```
var exampleArray:[Int] = [1,2,3,4,5]
//exampleArray = [1, 2, 3, 4, 5]
```

To access a value at a known index use the following syntax:

```
let exampleOne = exampleArray[2]
//exampleOne = 3
```

Note: The value at *index two is the third value* in the `Array`. `Array`s use a *zero based index* which means the first element in the `Array` is at index 0.

```
let value0 = exampleArray[0]
let value1 = exampleArray[1]
let value2 = exampleArray[2]
let value3 = exampleArray[3]
let value4 = exampleArray[4]
//value0 = 1
//value1 = 2
//value2 = 3
//value3 = 4
//value4 = 5
```

Access a subset of an `Array` using filter:

```
var filteredArray = exampleArray.filter({ $0 < 4 })
//filteredArray = [1, 2, 3]
```

Filters can have complex conditions like filtering only even numbers:

```
var evenArray = exampleArray.filter({ $0 % 2 == 0 })
```

```
//evenArray = [2, 4]
```

也可以返回给定值的索引，如果未找到该值则返回 nil。

```
exampleArray.indexOf(3) // Optional(2)
```

有用于获取数组中第一个、最后一个、最大或最小值的方法。如果

数组为空，这些方法将返回 nil。

```
exampleArray.first // Optional(1)
exampleArray.last // Optional(5)
exampleArray.maxElement() // Optional(5)
exampleArray.minElement() // Optional(1)
```



```
//evenArray = [2, 4]
```

It is also possible to return the index of a given value, returning `nil` if the value wasn't found.

```
exampleArray.indexOf(3) // Optional(2)
```

There are methods for the first, last, maximum or minimum value in an `Array`. These methods will return `nil` if the `Array` is empty.

```
exampleArray.first // Optional(1)
exampleArray.last // Optional(5)
exampleArray.maxElement() // Optional(5)
exampleArray.minElement() // Optional(1)
```

第7章：元组

元组类型是用逗号分隔的类型列表，括在圆括号内。

这个类型列表也可以包含元素的名称，并使用这些名称来引用各个元素的值。

元素名称由一个标识符紧跟冒号（:）组成。

常见用法 -

我们可以将元组类型用作函数的返回类型，使函数能够返回包含多个值的单个元组

第7.1节：什么是元组？

元组将多个值组合成一个复合值。元组中的值可以是任意类型，且不必彼此相同。

元组通过将任意数量的值组合在一起来创建：

```
let tuple = ("one", 2, "three")

// 值通过从零开始的索引号读取
print(tuple.0) // one
print(tuple.1) // 2
print(tuple.2) // three
```

定义元组时也可以为单个值命名：

```
let namedTuple = (first: 1, middle: "dos", last: 3)

// 可以通过命名属性读取值
print(namedTuple.first)  // 1
print(namedTuple.middle) // dos

// 也可以通过索引号读取
print(namedTuple.2)      // 3
```

在作为变量使用时也可以命名，甚至可以包含可选值：

```
var numbers: (optionalFirst: Int?, middle: String, last: Int)?

// 后续操作
numbers = (nil, "dos", 3)

print(numbers.optionalFirst)// nil
print(numbers.middle)//"dos"
print(numbers.last)//3
```

第7.2节：分解为单个变量

元组可以通过以下语法分解为单个变量：

```
let myTuple = (name: "Some Name", age: 26)
let (first, second) = myTuple
```


Chapter 7: Tuples

A tuple type is a comma-separated list of types, enclosed in parentheses.

This list of types also can have name of the elements and use those names to refer to the values of the individual elements.

An element name consists of an identifier followed immediately by a colon (:).

Common use -

We can use a tuple type as the return type of a function to enable the function to return a single tuple containing multiple values

Section 7.1: What are Tuples?

Tuples group multiple values into a single compound value. The values within a tuple can be of any type and do not have to be of the same type as each other.

Tuples are created by grouping any amount of values:

```
let tuple = ("one", 2, "three")

// Values are read using index numbers starting at zero
print(tuple.0) // one
print(tuple.1) // 2
print(tuple.2) // three
```

Also individual values can be named when the tuple is defined:

```
let namedTuple = (first: 1, middle: "dos", last: 3)

// Values can be read with the named property
print(namedTuple.first)  // 1
print(namedTuple.middle) // dos

// And still with the index number
print(namedTuple.2)      // 3
```

They can also be named when being used as a variable and even have the ability to have optional values inside:

```
var numbers: (optionalFirst: Int?, middle: String, last: Int)?

//Later On
numbers = (nil, "dos", 3)

print(numbers.optionalFirst)// nil
print(numbers.middle)//"dos"
print(numbers.last)//3
```

Section 7.2: Decomposing into individual variables

Tuples can be decomposed into individual variables with the following syntax:

```
let myTuple = (name: "Some Name", age: 26)
let (first, second) = myTuple
```

```
print(first) // "Some Name"
print(second)  // 26
```

无论元组是否具有未命名属性，都可以使用此语法：

```
let unnamedTuple = ("uno", "dos")
let (one, two) = unnamedTuple
print(one) // "uno"
print(two) // "dos"
```

可以使用下划线（_）忽略特定属性：

```
let longTuple = ("ichi", "ni", "san")
let (_, _, third) = longTuple
print(third) // "san"
```

第7.3节：元组作为函数的返回值

函数可以返回元组：

```
func tupleReturner() -> (Int, String) {
    return (3, "Hello")
}

let myTuple = tupleReturner()
print(myTuple.0) // 3
print(myTuple.1) // "Hello"
```

如果你为参数命名，可以通过返回值使用这些名称：

```
func tupleReturner() -> (anInteger: Int, aString: String) {
    return (3, "Hello")
}

let myTuple = tupleReturner()
print(myTuple.anInteger) // 3
print(myTuple.aString) // "Hello"
```

第7.4节：使用typealias为元组类型命名

有时你可能想在代码的多个地方使用相同的元组类型。如果元组较复杂，这很快会变得混乱：

```
// 通过中心点和半径定义一个圆的元组
let unitCircle: (center: (x: CGFloat, y: CGFloat), radius: CGFloat) = ((0.0, 0.0), 1.0)

func doubleRadius(ofCircle circle: (center: (x: CGFloat, y: CGFloat), radius: CGFloat)) -> (center:
(x: CGFloat, y: CGFloat), radius: CGFloat) {
    return (circle.center, circle.radius * 2.0)
}
```

如果你在多个地方使用某种元组类型，可以使用 typealias 关键字为你的元组类型命名。

```
// 通过中心点和半径定义一个圆的元组
typealias Circle = (center: (x: CGFloat, y: CGFloat), radius: CGFloat)

let unitCircle: Circle = ((0.0, 0.0), 1)
```



```
print(first) // "Some Name"
print(second)  // 26
```

This syntax can be used regardless of if the tuple has unnamed properties:

```
let unnamedTuple = ("uno", "dos")
let (one, two) = unnamedTuple
print(one) // "uno"
print(two) // "dos"
```

Specific properties can be ignored by using underscore (_):

```
let longTuple = ("ichi", "ni", "san")
let (_, _, third) = longTuple
print(third) // "san"
```

Section 7.3: Tuples as the Return Value of Functions

Functions can return tuples:

```
func tupleReturner() -> (Int, String) {
    return (3, "Hello")
}

let myTuple = tupleReturner()
print(myTuple.0) // 3
print(myTuple.1) // "Hello"
```

If you assign parameter names, they can be used from the return value:

```
func tupleReturner() -> (anInteger: Int, aString: String) {
    return (3, "Hello")
}

let myTuple = tupleReturner()
print(myTuple.anInteger) // 3
print(myTuple.aString) // "Hello"
```

Section 7.4: Using a typealias to name your tuple type

Occasionally you may want to use the same tuple type in multiple places throughout your code. This can quickly get messy, especially if your tuple is complex:

```
// Define a circle tuple by its center point and radius
let unitCircle: (center: (x: CGFloat, y: CGFloat), radius: CGFloat) = ((0.0, 0.0), 1.0)

func doubleRadius(ofCircle circle: (center: (x: CGFloat, y: CGFloat), radius: CGFloat)) -> (center:
(x: CGFloat, y: CGFloat), radius: CGFloat) {
    return (circle.center, circle.radius * 2.0)
}
```

If you use a certain tuple type in more than one place, you can use the `typealias` keyword to name your tuple type.

```
// Define a circle tuple by its center point and radius
typealias Circle = (center: (x: CGFloat, y: CGFloat), radius: CGFloat)

let unitCircle: Circle = ((0.0, 0.0), 1)
```

```swift
func doubleRadius(ofCircle circle: Circle) -> Circle {
    // 别名元组也可以访问原始元组类型中的值标签。
    return (circle.center, circle.radius * 2.0)
}
```

如果你发现自己经常这样做，应该考虑使用 struct 代替。

第7.5节：交换值

元组对于在两个（或更多）变量之间交换值而不使用临时变量非常有用。

两个变量的示例

给定两个变量

```swift
var a = "Marty McFly"
var b = "Emmett Brown"
```

我们可以轻松交换数值

```swift
(a, b) = (b, a)
```

结果：

```swift
print(a) // "埃米特·布朗"
print(b) // "马蒂·麦克弗莱"
```

四变量示例

```swift
var a = 0
var b = 1
var c = 2
var d = 3

(a, b, c, d) = (d, c, b, a)

print(a, b, c, d) // 3, 2, 1, 0
```

第7.6节：元组作为Switch中的Case

在 switch 中使用元组

```swift
let switchTuple = (firstCase: true, secondCase: false)

switch switchTuple {
 case (true, false):
    // 执行某些操作
 case (true, true):
    // 执行某些操作
 case (false, true):
    // 执行某些操作
 case (false, false):
    // 执行某些操作
}
```

或者与枚举结合使用，例如使用尺寸类别（Size Classes）：


```swift
func doubleRadius(ofCircle circle: Circle) -> Circle {
    // Aliased tuples also have access to value labels in the original tuple type.
    return (circle.center, circle.radius * 2.0)
}
```

If you find yourself doing this too often, however, you should consider using a `struct` instead.

Section 7.5: Swapping values

Tuples are useful to swap values between 2 (or more) variables without using temporary variables.

Example with 2 variables

Given 2 variables

```swift
var a = "Marty McFly"
var b = "Emmett Brown"
```

we can easily swap the values

```swift
(a, b) = (b, a)
```

Result:

```swift
print(a) // "Emmett Brown"
print(b) // "Marty McFly"
```

Example with 4 variables

```swift
var a = 0
var b = 1
var c = 2
var d = 3

(a, b, c, d) = (d, c, b, a)

print(a, b, c, d) // 3, 2, 1, 0
```

Section 7.6: Tuples as Case in Switch

Use tuples in a switch

```swift
let switchTuple = (firstCase: true, secondCase: false)

switch switchTuple {
 case (true, false):
    // do something
 case (true, true):
    // do something
 case (false, true):
    // do something
 case (false, false):
    // do something
}
```

Or in combination with an Enum For example with Size Classes:

```swift
let switchTuple = (UIUserInterfaceSizeClass.Compact, UIUserInterfaceSizeClass.Regular)

switch switchTuple {
case (.Regular, .Compact):
    //语句
case (.Regular, .Regular):
    //语句
case (.Compact, .Regular):
    //语句
case (.Compact, .Compact):
    //语句
}
```



```swift
let switchTuple = (UIUserInterfaceSizeClass.Compact, UIUserInterfaceSizeClass.Regular)

switch switchTuple {
case (.Regular, .Compact):
    //statement
case (.Regular, .Regular):
    //statement
case (.Compact, .Regular):
    //statement
case (.Compact, .Compact):
    //statement
}
```

第8章：枚举

第8.1节：基本枚举

一个enum提供一组相关的值：

```
enum Direction {
    case up
    case 下
    case 左
    case 右
}

enum 方向 { case 上, 下, 左, 右 }
```

枚举值可以使用其全限定名称，但当类型名称可以推断时，可以省略：

```
let 方向 = 方向.上
let 方向: 方向 = 方向.上
let 方向: 方向 = .上

// func move(dir: 方向)...
move(方向.上)
move(.上)

obj.方向 = 方向.上
obj.方向 = .上
```

比较/提取枚举值最基本的方法是使用 switch 语句：

```
switch dir {
case .up:
    // 处理向上情况
case .down:
    // 处理向下情况
case .left:
    // 处理向左情况
case .right:
    // 处理向右情况
}
```

简单的枚举会自动成为Hashable、Equatable，并且支持字符串转换：

```
if dir == .down { ... }

let dirs: Set<Direction> = [.right, .left]

print(Direction.up)  // 输出 "up"
debugPrint(Direction.up)  // 输出 "Direction.up"
```

第8.2节：带关联值的枚举

枚举的case可以包含一个或多个负载（关联值）：

```
enum Action {
    case jump
    case kick
```


Chapter 8: Enums

Section 8.1: Basic enumerations

An enum provides a set of related values:

```
enum Direction {
    case up
    case down
    case left
    case right
}

enum Direction { case up, down, left, right }
```

Enum values can be used by their fully-qualified name, but you can omit the type name when it can be inferred:

```
let dir = Direction.up
let dir: Direction = Direction.up
let dir: Direction = .up

// func move(dir: Direction)...
move(Direction.up)
move(.up)

obj.dir = Direction.up
obj.dir = .up
```

The most fundamental way of comparing/extracting enum values is with a `switch` statement:

```
switch dir {
case .up:
    // handle the up case
case .down:
    // handle the down case
case .left:
    // handle the left case
case .right:
    // handle the right case
}
```

Simple enums are automatically Hashable, Equatable and have string conversions:

```
if dir == .down { ... }

let dirs: Set<Direction> = [.right, .left]

print(Direction.up)  // prints "up"
debugPrint(Direction.up)  // prints "Direction.up"
```

Section 8.2: Enums with associated values

Enum cases can contain one or more **payloads** (**associated values**):

```
enum Action {
    case jump
    case kick
```

```
    case move(距离: Float)  // “move”情况有一个关联的距离值
}
```

实例化枚举值时必须提供有效载荷：

```
performAction(.jump)
performAction(.kick)
performAction(.move(距离: 3.3))
performAction(.move(距离: 0.5))
```

switch 语句可以提取关联值：

```
switch action {
case .jump:
    ...
case .kick:
    ...
case .move(let 距离):  // 或者 case let .move(距离):
    print("移动中: \(距离)")
}
```

单个 case 提取可以使用 if case:

```
如果情况.移动(让 距离) = 动作 {
    打印("移动中: \(距离)")
}
```

守卫case语法可用于后续的值提取：

```
guard case .move(let distance) = action else {
    print("动作不是移动")
    return
}
```

带关联值的枚举默认不是Equatable。必须手动实现==操作符：

```
extension Action: Equatable { }

func ==(lhs: Action, rhs: Action) -> Bool {
    switch lhs {
    case .jump: if case .jump = rhs { return true }
    case .kick: if case .kick = rhs { return true }
    case .move(let lhsDistance): if case .move (let rhsDistance) = rhs { return lhsDistance ==
rhsDistance }
    }
    return false
}
```

第8.3节：间接负载

通常，枚举不能递归（因为它们需要无限存储空间）：

```
enum Tree<T> {
    case 叶子节点(T)
    case 分支(Tree<T>, Tree<T>)  // 错误：递归枚举 'Tree<T>' 未标记为 'indirect'
}
```



```
    case move(distance: Float)  // The "move" case has an associated distance
}
```

The payload must be provided when instantiating the enum value:

```
performAction(.jump)
performAction(.kick)
performAction(.move(distance: 3.3))
performAction(.move(distance: 0.5))
```

The `switch` statement can extract the associated value:

```
switch action {
case .jump:
    ...
case .kick:
    ...
case .move(let distance):  // or case let .move(distance):
    print("Moving: \(distance)")
}
```

A single case extraction can be done using `if case`:

```
if case .move(let distance) = action {
    print("Moving: \(distance)")
}
```

The `guard case` syntax can be used for later use extraction:

```
guard case .move(let distance) = action else {
    print("Action is not move")
    return
}
```

Enums with associated values are not `Equatable` by default. Implementation of the `==` operator must be done manually:

```
extension Action: Equatable { }

func ==(lhs: Action, rhs: Action) -> Bool {
    switch lhs {
    case .jump: if case .jump = rhs { return true }
    case .kick: if case .kick = rhs { return true }
    case .move(let lhsDistance): if case .move (let rhsDistance) = rhs { return lhsDistance ==
rhsDistance }
    }
    return false
}
```

Section 8.3: Indirect payloads

Normally, enums can't be recursive (because they would require infinite storage):

```
enum Tree<T> {
    case leaf(T)
    case branch(Tree<T>, Tree<T>)  // error: recursive enum 'Tree<T>' is not marked 'indirect'
}
```

关键字indirect 使枚举通过一层间接引用来存储其负载，而不是内联存储。
你可以在单个 case 上使用此关键字：

```
enum Tree<T> {
    case 叶子节点(T)
indirect case 分支(Tree<T>, Tree<T>)
}

let 树 = Tree.分支(.叶子节点(1), .分支(.叶子节点(2), .叶子节点(3)))
```

indirect 也可以作用于整个枚举，使任何 case 在必要时都变为间接存储：

```
indirect enum Tree<T> {
    case 叶子节点(T)
    case 分支(Tree<T>, Tree<T>)
}
```

第8.4节：原始值和哈希值

无负载的枚举可以具有任何字面量类型的原始值：

```
枚举 Rotation: Int {
    case up = 0
    case left = 90
    case upsideDown = 180
    case right = 270
}
```

没有指定具体类型的枚举不会暴露 rawValue 属性

```
enum Rotation {
    case up
    case 右
    case 下
    case 左
}

let foo = Rotation.up
foo.rawValue //错误
```

整数原始值默认从 0 开始并单调递增：

```
enum MetasyntacticVariable: Int {
    case foo  // rawValue 自动为 0
    case bar  // rawValue 自动为 1
    case baz = 7
    case quux  // rawValue 自动为 8
}
```

字符串原始值可以自动合成：

```
enum MarsMoon: String {
    case phobos  // rawValue 自动为 "phobos"
    case deimos  // rawValue 自动为 "deimos"
}
```

带原始值的枚举会自动遵循RawRepresentable协议。你可以通过枚举值的对应原始值

The **indirect** keyword makes the enum store its payload with a layer of indirection, rather than storing it inline.
You can use this keyword on a single case:

```
enum Tree<T> {
    case leaf(T)
    indirect case branch(Tree<T>, Tree<T>)
}

let tree = Tree.branch(.leaf(1), .branch(.leaf(2), .leaf(3)))
```

indirect also works on the whole enum, making any case indirect when necessary:

```
indirect enum Tree<T> {
    case leaf(T)
    case branch(Tree<T>, Tree<T>)
}
```

Section 8.4: Raw and Hash values

Enums without payloads can have *raw values* of any literal type:

```
enum Rotation: Int {
    case up = 0
    case left = 90
    case upsideDown = 180
    case right = 270
}
```

Enums without any specific type do not expose the rawValue property

```
enum Rotation {
    case up
    case right
    case down
    case left
}

let foo = Rotation.up
foo.rawValue //error
```

Integer raw values are assumed to start at 0 and increase monotonically:

```
enum MetasyntacticVariable: Int {
    case foo  // rawValue is automatically 0
    case bar  // rawValue is automatically 1
    case baz = 7
    case quux  // rawValue is automatically 8
}
```

String raw values can be synthesized automatically:

```
enum MarsMoon: String {
    case phobos  // rawValue is automatically "phobos"
    case deimos  // rawValue is automatically "deimos"
}
```

A raw-valued enum automatically conforms to RawRepresentable. You can get an enum value's corresponding raw

使用.rawValue获取：

```
func rotate(rotation: Rotation) {
    let degrees = rotation.rawValue
    …
}
```

你也可以使用init?(rawValue:)通过原始值创建枚举：

```
let rotation = Rotation(rawValue: 0)  // 返回 Rotation.Up
let otherRotation = Rotation(rawValue: 45)  // 返回 nil（没有 rawValue 为 45 的 Rotation）

if let moon = MarsMoon(rawValue: str) {
    print("火星有一颗名为 \(str) 的卫星")
} else {
    print("火星没有名为 \(str) 的卫星")
}
```

如果你想获取某个枚举的哈希值，可以访问它的 hashValue，哈希值会返回枚举的索引，索引从零开始。

```
let quux = MetasyntacticVariable(rawValue: 8)// rawValue 是 8
quux?.hashValue // hashValue 是 3
```

第8.5节：初始化器

枚举可以有自定义的初始化方法，这些方法可能比默认的init?(rawValue:)更有用。枚举也可以存储值。这对于存储它们初始化时的值并在以后检索该值非常有用。

```
enum 指南针方向 {
    case 北(Int)
    case 南(Int)
    case 东(Int)
    case 西(Int)

    init?(度数: Int) {
        switch 度数 {
        case 0...45:
            self = .北(度数)
        case 46...135:
            self = .东(度数)
        case 136...225:
            self = .南(度数)
        case 226...315:
            self = .西(度数)
        case 316...360:
            self = .北(度数)
        default:
            return nil
        }
    }

    var 值: Int = {
        switch self {
            case north(let degrees):
                return degrees
            case south(let degrees):
```


value with .rawValue:

```
func rotate(rotation: Rotation) {
    let degrees = rotation.rawValue
    ...
}
```

You can also create an enum *from* a raw value using init?(rawValue:):

```
let rotation = Rotation(rawValue: 0)  // returns Rotation.Up
let otherRotation = Rotation(rawValue: 45)  // returns nil (there is no Rotation with rawValue 45)

if let moon = MarsMoon(rawValue: str) {
    print("Mars has a moon named \(str)")
} else {
    print("Mars doesn't have a moon named \(str)")
}
```

If you wish to get the hash value of a specific enum you can access its hashValue, The hash value will return the index of the enum starting from zero.

```
let quux = MetasyntacticVariable(rawValue: 8)// rawValue is 8
quux?.hashValue //hashValue is 3
```

Section 8.5: Initializers

Enums can have custom init methods that can be more useful than the default init?(rawValue:). Enums can also store values as well. This can be useful for storing the values they where initialized with and retrieving that value later.

```
enum CompassDirection {
    case north(Int)
    case south(Int)
    case east(Int)
    case west(Int)

    init?(degrees: Int) {
        switch degrees {
        case 0...45:
            self = .north(degrees)
        case 46...135:
            self = .east(degrees)
        case 136...225:
            self = .south(degrees)
        case 226...315:
            self = .west(degrees)
        case 316...360:
            self = .north(degrees)
        default:
            return nil
        }
    }

    var value: Int = {
        switch self {
            case north(let degrees):
                return degrees
            case south(let degrees):
```

```
                return degrees
            case east(let degrees):
                return degrees
            case west(let degrees):
                return degrees
        }
    }
}
```

使用该初始化器我们可以这样做：

```
var direction = CompassDirection(degrees: 0) // 返回 CompassDirection.north
direction = CompassDirection(degrees: 90) // 返回 CompassDirection.east
print(direction.value) //打印 90
direction = CompassDirection(degrees: 500) // 返回 nil
```

第8.6节：枚举与类和结构体共享许多特性

Swift中的枚举比其他语言（如C）中的对应类型更强大。它们与类和结构体共享许多特性，例如定义初始化器、计算属性、实例方法、协议遵循和扩展。

```
protocol ChangesDirection {
    mutating func changeDirection()
}

enum Direction {

    // 枚举情况
    case 上, 下, 左, 右// 使用与另一

    个方向相反的情况初始化枚举实例

    init(oppositeTo otherDirection: Direction) {
        self = otherDirection.opposite
    }

    // 计算属性，返回相反的方向
    var opposite: Direction {
        switch self {
        case .上:
            return .下
        case .下:
            return .上
        case .左:
            return .右
        case .右:
            return .左
        }
    }
}

// Direction 的扩展，添加对 ChangesDirection 协议的遵循
extension Direction: ChangesDirection {
    mutating func changeDirection() {
        self = .左
    }
}
```



```
                return degrees
            case east(let degrees):
                return degrees
            case west(let degrees):
                return degrees
        }
    }
}
```

Using that initializer we can do this:

```
var direction = CompassDirection(degrees: 0) // Returns CompassDirection.north
direction = CompassDirection(degrees: 90) // Returns CompassDirection.east
print(direction.value) //prints 90
direction = CompassDirection(degrees: 500) // Returns nil
```

Section 8.6: Enumerations share many features with classes and structures

Enums in Swift are much more powerful than some of their counterparts in other languages, such as C. They share many features with classes and structs, such as defining initialisers, computed properties, instance methods, protocol conformances and extensions.

```
protocol ChangesDirection {
    mutating func changeDirection()
}

enum Direction {

    // enumeration cases
    case up, down, left, right

    // initialise the enum instance with a case
    // that's in the opposite direction to another
    init(oppositeTo otherDirection: Direction) {
        self = otherDirection.opposite
    }

    // computed property that returns the opposite direction
    var opposite: Direction {
        switch self {
        case .up:
            return .down
        case .down:
            return .up
        case .left:
            return .right
        case .right:
            return .left
        }
    }
}

// extension to Direction that adds conformance to the ChangesDirection protocol
extension Direction: ChangesDirection {
    mutating func changeDirection() {
        self = .left
    }
}
```

```
var dir = Direction(oppositeTo: .down) // Direction.up

dir.changeDirection() // Direction.left

let opposite = dir.opposite // Direction.right
```

第8.7节：嵌套枚举

你可以将枚举嵌套在另一个枚举中，这使你能够构建层次化的枚举，使结构更加有序和清晰。

```
enum Orchestra {
    enum Strings {
        case violin
        case viola
        case cello
        case doubleBasse
    }

    enum Keyboards {
        case piano
        case celesta
        case harp
    }

    enum 木管乐器 {
        case 长笛
        盒子 双簧管
        盒子 单簧管
        盒子 巴松管
        盒子 低音巴松管
    }
}
```

你可以这样使用它：

```
let 乐器1 = Orchestra.弦乐器.中提琴
let 乐器2 = Orchestra.键盘乐器.钢琴
```



```
var dir = Direction(oppositeTo: .down) // Direction.up

dir.changeDirection() // Direction.left

let opposite = dir.opposite // Direction.right
```

Section 8.7: Nested Enumerations

You can nest enumerations one inside an other, this allows you to structure hierarchical enums to be more organized and clear.

```
enum Orchestra {
    enum Strings {
        case violin
        case viola
        case cello
        case doubleBasse
    }

    enum Keyboards {
        case piano
        case celesta
        case harp
    }

    enum Woodwinds {
        case flute
        case oboe
        case clarinet
        case bassoon
        case contrabassoon
    }
}
```

And you can use it like that:

```
let instrment1 = Orchestra.Strings.viola
let instrment2 = Orchestra.Keyboards.piano
```

第9章：结构体

第9.1节：结构体是值类型

与通过引用传递的类不同，结构体是通过复制传递的：

```
first = "Hello"
second = first
first += " World!"
// first == "Hello World!"
// second == "Hello"
```

字符串是结构体，因此赋值时会被复制。

结构体也不能使用身份运算符进行比较：

```
window0 === window1 // 有效，因为 window 是类的实例
"hello" === "hello" // 错误：二元操作符 '===' 不能应用于两个 'String' 操作数
```

如果两个结构体实例比较相等，则认为它们是相同的。

这些将结构体与类区分开的特性共同构成了结构体作为值类型的本质。

第9.2节：访问结构体的成员

在 Swift 中，结构体使用简单的“点语法”来访问它们的成员。

例如：

```
struct DeliveryRange {
  var range: Double
  let center: Location
}
let storeLocation = Location(latitude: 44.9871,
                             longitude: -93.2758)
var pizzaRange = DeliveryRange(range: 200,
                               center: storeLocation)
```

你可以这样访问（打印）range：

```
    print(pizzaRange.range) // 200
```

你甚至可以使用点语法访问成员的成员：

```
print(pizzaRange.center.latitude) // 44.9871
```

类似于你可以使用点语法读取值，你也可以赋值。

```
pizzaRange.range = 250
```

第9.3节：结构体基础

```
struct Repository {
    let identifier: Int
```


Chapter 9: Structs

Section 9.1: Structs are value types

Unlike classes, which are passed by reference, structures are passed through copying:

```
first = "Hello"
second = first
first += " World!"
// first == "Hello World!"
// second == "Hello"
```

String is a structure, therefore it's copied on assignment.

Structures also cannot be compared using identity operator:

```
window0 === window1 // works because a window is a class instance
"hello" === "hello" // error: binary operator '===' cannot be applied to two 'String' operands
```

Every two structure instances are deemed identical if they compare equal.

Collectively, these traits that differentiate structures from classes are what makes structures value types.

Section 9.2: Accessing members of struct

In Swift, structures use a simple “dot syntax” to access their members.

For example:

```
struct DeliveryRange {
  var range: Double
  let center: Location
}
let storeLocation = Location(latitude: 44.9871,
                             longitude: -93.2758)
var pizzaRange = DeliveryRange(range: 200,
                               center: storeLocation)
```

You can access(print) the range like this:

```
    print(pizzaRange.range) // 200
```

You can even access members of members using dot syntax:

```
print(pizzaRange.center.latitude) // 44.9871
```

Similar to how you can read values with dot syntax, you can also assign them.

```
pizzaRange.range = 250
```

Section 9.3: Basics of Structs

```
struct Repository {
    let identifier: Int
```

```
    let name: String
    var description: String?
}
```

这定义了一个Repository结构体，包含三个存储属性，一个整数类型的identifier，一个字符串类型的name，以及一个可选的字符串类型的description。由于identifier和name是用let关键字声明的常量，初始化后不能修改。description是变量，修改它会更新结构体的值。

如果结构体没有定义任何自定义初始化器，结构体类型会自动获得一个成员逐一初始化器。即使结构体的存储属性没有默认值，也会获得成员逐一初始化器。

Repository包含三个存储属性，其中只有description有默认值（nil）。此外，它没有定义自己的初始化器，因此会自动获得一个成员逐一初始化器：

```
let newRepository = Repository(identifier: 0, name: "New Repository", description: "Brand New Repository")
```

第9.4节：修改结构体

修改结构体自身值的方法必须以mutating关键字作为前缀

```
struct Counter {
    private var value = 0

    mutating func next() {
        value += 1
    }
}
```

何时可以使用mutating方法

mutating方法仅适用于变量中的结构体值。

```
var counter = Counter()
counter.next()
```

何时不能使用mutating方法

另一方面，mutating方法不能用于常量中的结构体值

```
let counter = Counter()
counter.next()
// 错误：不能对不可变值使用可变成员：'counter'是一个'let'常量
```

第9.5节：结构体不能继承

与类不同，结构体不能继承：

```
class MyView: NSView { }  // 可用

struct MyInt: Int { } // 错误：不能从非协议类型'Int'继承
```

但是，结构体可以遵循协议：


```
    let name: String
    var description: String?
}
```

This defines a `Repository` struct with three stored properties, an integer `identifier`, a string `name`, and an optional string `description`. The `identifier` and `name` are constants, as they've been declared using the `let`-keyword. Once set during initialization, they cannot be modified. The description is a variable. Modifying it updates the value of the structure.

Structure types automatically receive a memberwise initializer if they do not define any of their own custom initializers. The structure receives a memberwise initializer even if it has stored properties that do not have default values.

`Repository` contains three stored properties of which only `description` has a default value (`nil`). Further it defines no initializers of its own, so it receives a memberwise initializer for free:

```
let newRepository = Repository(identifier: 0, name: "New Repository", description: "Brand New Repository")
```

Section 9.4: Mutating a Struct

A method of a struct that change the value of the struct itself must be prefixed with the `mutating` keyword

```
struct Counter {
    private var value = 0

    mutating func next() {
        value += 1
    }
}
```

When you can use mutating methods

The `mutating` methods are only available on struct values inside variables.

```
var counter = Counter()
counter.next()
```

When you can NOT use mutating methods

On the other hand, `mutating` methods are NOT available on struct values inside constants

```
let counter = Counter()
counter.next()
//  error: cannot use mutating member on immutable value: 'counter' is a 'let' constant
```

Section 9.5: Structs cannot inherit

Unlike classes, structures cannot inherit:

```
class MyView: NSView { }  // works

struct MyInt: Int { } // error: inheritance from non-protocol type 'Int'
```

Structures, however, can adopt protocols:

```
struct Vector: Hashable { ... }  // 可用
```



```
struct Vector: Hashable { ... }  // works
```

第10章：集合

第10.1节：声明集合

集合是无序的唯一值集合。唯一值必须是相同类型。

```
var colors = Set<String>()
```

你可以使用数组字面量语法声明一个带有值的集合。

```
var favoriteColors: Set<String> = ["Red", "Blue", "Green", "Blue"]
// {"Blue", "Green", "Red"}
```

第10.2节：对集合执行操作

两个集合中的公共值：

你可以使用intersect(_:)方法创建一个包含两个集合中所有公共值的新集合。

```
let favoriteColors: Set = ["Red", "Blue", "Green"]
let newColors: Set = ["Purple", "Orange", "Green"]

let intersect = favoriteColors.intersect(newColors) // a AND b
// intersect = {"Green"}
```

每个集合中的所有值：

你可以使用union(_:)方法创建一个包含每个集合中所有唯一值的新集合。

```
let union = favoriteColors.union(newColors) // a 或 b
// union = {"Red", "Purple", "Green", "Orange", "Blue"}
```

注意"Green"这个值在新集合中只出现了一次。

两个集合中都不存在的值：

你可以使用`exclusiveOr(_:)`方法来创建一个新集合，包含任一集合中独有的值，但不包括

两个集合共有的值。

```
let exclusiveOr = favoriteColors.exclusiveOr(newColors) // a XOR b
// exclusiveOr = {"Red", "Purple", "Orange", "Blue"}
```

注意"Green"这个值没有出现在新集合中，因为它存在于两个集合中。

不在某个集合中的值：

你可以使用subtract(_:)方法来创建一个新集合，包含不在特定集合中的值。

```
let subtract = favoriteColors.subtract(newColors) // a - (a AND b)
// subtract = {"Blue", "Red"}
```

注意"Green"这个值没有出现在新集合中，因为它也存在于第二个集合中。

Chapter 10: Sets

Section 10.1: Declaring Sets

Sets are unordered collections of unique values. Unique values must be of the same type.

```
var colors = Set<String>()
```

You can declare a set with values by using the array literal syntax.

```
var favoriteColors: Set<String> = ["Red", "Blue", "Green", "Blue"]
// {"Blue", "Green", "Red"}
```

Section 10.2: Performing operations on sets

Common values from both sets:

You can use the `intersect(_:)` method to create a new set containing all the values common to both sets.

```
let favoriteColors: Set = ["Red", "Blue", "Green"]
let newColors: Set = ["Purple", "Orange", "Green"]

let intersect = favoriteColors.intersect(newColors) // a AND b
// intersect = {"Green"}
```

All values from each set:

You can use the `union(_:)` method to create a new set containing all the unique values from each set.

```
let union = favoriteColors.union(newColors) // a OR b
// union = {"Red", "Purple", "Green", "Orange", "Blue"}
```

Notice how the value "Green" only appears once in the new set.

Values that don't exist in both sets:

You can use the `exclusiveOr(_:)` method to create a new set containing the unique values from either but not both sets.

```
let exclusiveOr = favoriteColors.exclusiveOr(newColors) // a XOR b
// exclusiveOr = {"Red", "Purple", "Orange", "Blue"}
```

Notice how the value "Green" doesn't appear in the new set, since it was in both sets.

Values that are not in a set:

You can use the `subtract(_:)` method to create a new set containing values that aren't in a specific set.

```
let subtract = favoriteColors.subtract(newColors) // a - (a AND b)
// subtract = {"Blue", "Red"}
```

Notice how the value "Green" doesn't appear in the new set, since it was also in the second set.

第10.3节：计数集合（CountedSet）

版本 = 3.0

Swift 3 引入了CountedSet类（它是NSCountedSet Objective-C 类的 Swift 版本）。

顾名思义，CountedSet 会跟踪一个值出现的次数。

```
let countedSet = CountedSet()
countedSet.add(1)
countedSet.add(1)
countedSet.add(1)
countedSet.add(2)

countedSet.count(for: 1) // 3
countedSet.count(for: 2) // 1
```

第10.4节：修改集合中的值

```
var favoriteColors: Set = ["Red", "Blue", "Green"]
//favoriteColors = {"Blue", "Green", "Red"}
```

你可以使用insert(_:)方法向集合中添加新项。

```
favoriteColors.insert("Orange")
//favoriteColors = {"Red", "Green", "Orange", "Blue"}
```

你可以使用remove(_:)方法从集合中移除某项。它返回一个可选值，包含被移除的值，如果值不在集合中则返回 nil。

```
let removedColor = favoriteColors.remove("Red")
//favoriteColors = {"Green", "Orange", "Blue"}
// removedColor = Optional("Red")

let anotherRemovedColor = favoriteColors.remove("Black")
// anotherRemovedColor = nil
```

第10.5节：检查集合是否包含某个值

```
var favoriteColors: Set = ["Red", "Blue", "Green"]
//favoriteColors = {"Blue", "Green", "Red"}
```

你可以使用contains(_:)方法来检查集合是否包含某个值。如果集合包含该值，则返回true。

```
if favoriteColors.contains("Blue") {
    print("谁不喜欢蓝色！")
}
// 输出 "谁不喜欢蓝色！"
```

第10.6节：向集合中添加自定义类型的值

为了定义你自己的类型的Set，你需要让你的类型遵循Hashable

```
struct Starship: Hashable {
```


Section 10.3: CountedSet

Version = 3.0

Swift 3 introduces the `CountedSet` class (it's the Swift version of the `NSCountedSet` Objective-C class).

CountedSet, as suggested by the name, keeps track of how many times a value is present.

```
let countedSet = CountedSet()
countedSet.add(1)
countedSet.add(1)
countedSet.add(1)
countedSet.add(2)

countedSet.count(for: 1) // 3
countedSet.count(for: 2) // 1
```

Section 10.4: Modifying values in a set

```
var favoriteColors: Set = ["Red", "Blue", "Green"]
//favoriteColors = {"Blue", "Green", "Red"}
```

You can use the `insert(_:)` method to add a new item into a set.

```
favoriteColors.insert("Orange")
//favoriteColors = {"Red", "Green", "Orange", "Blue"}
```

You can use the `remove(_:)` method to remove an item from a set. It returns optional containing value that was removed or nil if value was not in the set.

```
let removedColor = favoriteColors.remove("Red")
//favoriteColors = {"Green", "Orange", "Blue"}
// removedColor = Optional("Red")

let anotherRemovedColor = favoriteColors.remove("Black")
// anotherRemovedColor = nil
```

Section 10.5: Checking whether a set contains a value

```
var favoriteColors: Set = ["Red", "Blue", "Green"]
//favoriteColors = {"Blue", "Green", "Red"}
```

You can use the `contains(_:)` method to check whether a set contains a value. It will return true if the set contains that value.

```
if favoriteColors.contains("Blue") {
    print("Who doesn't like blue!")
}
// Prints "Who doesn't like blue!"
```

Section 10.6: Adding values of my own type to a Set

In order to define a `Set` of your own type you need to conform your type to `Hashable`

```
struct Starship: Hashable {
```

```
    let name: String
    var hashValue: Int { return name.hashValue }
}

func ==(left:星际飞船, right: 星际飞船) -> Bool {
    return left.name == right.name
}
```

现在你可以创建一个星际飞船的集合(Set)

```
let ships : Set<星际飞船> = [星际飞船(name:"Enterprise D"), 星际飞船(name:"Voyager"),
星际飞船(name:"Defiant") ]
```



```
    let name: String
    var hashValue: Int { return name.hashValue }
}

func ==(left:Starship, right: Starship) -> Bool {
    return left.name == right.name
}
```

Now you can create a `Set` of `Starship(s)`

```
let ships : Set<Starship> = [Starship(name:"Enterprise D"), Starship(name:"Voyager"),
Starship(name:"Defiant") ]
```

第11章：字典

第11.1节：声明字典

字典是键和值的无序集合。值对应唯一的键，且必须是相同的类型。

初始化字典的完整语法如下：

```
var books : Dictionary<Int, String> = Dictionary<Int, String>()
```

虽然有更简洁的初始化方式：

```
var books = [Int: String]()
// 或者
var books: [Int: String] = [:]
```

通过指定以逗号分隔的键和值列表来声明一个字典。类型可以根据键和值的类型推断得出。

```
var books: [Int: String] = [1: "Book 1", 2: "Book 2"]
//books = [2: "Book 2", 1: "Book 1"]
var otherBooks = [3: "Book 3", 4: "Book 4"]
//otherBooks = [3: "Book 3", 4: "Book 4"]
```

第11.2节：访问值

可以使用键来访问Dictionary中的值：

```
var books: [Int: String] = [1: "Book 1", 2: "Book 2"]
let bookName = books[1]
//bookName = "Book 1"
```

可以使用values属性遍历字典中的值：

```
for book in books.values {
    print("书名: \(book)")
}
//输出：书名：书 2
//输出：书名：书 1
```

同样，可以使用字典的keys属性来遍历其键：

```
for bookNumbers in books.keys {
    print("书号: \(bookNumber)")
}
// 输出：
// 书号: 1
// 书号: 2
```

要获取所有对应的key和value对（由于是字典，顺序不保证正确）

```
for (book,bookNumbers)in books{
print("\(book)  \(bookNumbers)")
}
```


Chapter 11: Dictionaries

Section 11.1: Declaring Dictionaries

Dictionaries are an unordered collection of keys and values. Values relate to unique keys and must be of the same type.

When initializing a Dictionary the full syntax is as follows:

```
var books : Dictionary<Int, String> = Dictionary<Int, String>()
```

Although a more concise way of initializing:

```
var books = [Int: String]()
// or
var books: [Int: String] = [:]
```

Declare a dictionary with keys and values by specifying them in a comma separated list. The types can be inferred from the types of keys and values.

```
var books: [Int: String] = [1: "Book 1", 2: "Book 2"]
//books = [2: "Book 2", 1: "Book 1"]
var otherBooks = [3: "Book 3", 4: "Book 4"]
//otherBooks = [3: "Book 3", 4: "Book 4"]
```

Section 11.2: Accessing Values

A value in a `Dictionary` can be accessed using its key:

```
var books: [Int: String] = [1: "Book 1", 2: "Book 2"]
let bookName = books[1]
//bookName = "Book 1"
```

The values of a dictionary can be iterated through using the `values` property:

```
for book in books.values {
    print("Book Title: \(book)")
}
//output: Book Title: Book 2
//output: Book Title: Book 1
```

Similarly, the keys of a dictionary can be iterated through using its `keys` property:

```
for bookNumbers in books.keys {
    print("Book number: \(bookNumber)")
}
// outputs:
// Book number: 1
// Book number: 2
```

To get all `key` and `value` pair corresponding to each other (you will not get in proper order since it is a Dictionary)

```
for (book,bookNumbers)in books{
print("\(book)  \(bookNumbers)")
}
```

```
// 输出：
// 2  书 2
// 1  书 1
```

注意，`Dictionary`与`Array`不同，天生是无序的——即在

迭代时不保证顺序。

如果想访问字典的多层级，可以使用重复的下标语法。

```
// 创建一个多级字典。
var myDictionary: [String:[Int:String]]! =
["玩具":[1:"汽车",2:"卡车"],"兴趣":[1:"科学",2:"数学"]]

print(myDictionary["玩具"][2]) // 输出 "卡车"
print(myDictionary["兴趣"][1]) // 输出 "科学"
```

第11.3节：使用键更改字典的值

```
var dict = ["名字": "约翰", "姓氏": "多伊"]
// 将键为 '名字' 的元素设置为 '简'
dict["名字"] = "简"
print(dict)
```

第11.4节：获取字典中的所有键

```
let myAllKeys = ["名字" : "基里特" , "姓氏" : "莫迪"]
let allKeys = Array(myAllKeys.keys)
print(allKeys)
```

第11.5节：修改字典

向字典添加键和值

```
var books = [Int: String]()
//books = [:]
books[5] = "Book 5"
//books = [5: "Book 5"]
books.updateValue("Book 6", forKey: 5)
//[5: "Book 6"]
```

updateValue 如果存在原始值则返回该值，否则返回 nil。

```
let previousValue = books.updateValue("Book 7", forKey: 5)
//books = [5: "Book 7"]
//previousValue = "Book 6"
```

使用类似语法移除值及其键

```
books[5] = nil
//books [:]
books[6] = "Deleting from Dictionaries"
//books = [6: "Deleting from Dictionaries"]
let removedBook = books.removeValueForKey(6)
//books = [:]
//removedValue = "从字典中删除"
```



```
// outputs:
// 2  Book 2
// 1  Book 1
```

Note that a `Dictionary`, unlike an `Array`, in inherently unordered-that is, there is no guarantee on the order during iteration.

If you want to access multiple levels of a Dictionary use a repeated subscript syntax.

```
// Create a multilevel dictionary.
var myDictionary: [String:[Int:String]]! =
["Toys":[1:"Car",2:"Truck"],"Interests":[1:"Science",2:"Math"]]

print(myDictionary["Toys"][2]) // Outputs "Truck"
print(myDictionary["Interests"][1]) // Outputs "Science"
```

Section 11.3: Change Value of Dictionary using Key

```
var dict = ["name": "John", "surname": "Doe"]
// Set the element with key: 'name' to 'Jane'
dict["name"] = "Jane"
print(dict)
```

Section 11.4: Get all keys in Dictionary

```
let myAllKeys = ["name" : "Kirit" , "surname" : "Modi"]
let allKeys = Array(myAllKeys.keys)
print(allKeys)
```

Section 11.5: Modifying Dictionaries

Add a key and value to a Dictionary

```
var books = [Int: String]()
//books = [:]
books[5] = "Book 5"
//books = [5: "Book 5"]
books.updateValue("Book 6", forKey: 5)
//[5: "Book 6"]
```

updateValue returns the original value if one exists or nil.

```
let previousValue = books.updateValue("Book 7", forKey: 5)
//books = [5: "Book 7"]
//previousValue = "Book 6"
```

Remove value and their keys with similar syntax

```
books[5] = nil
//books [:]
books[6] = "Deleting from Dictionaries"
//books = [6: "Deleting from Dictionaries"]
let removedBook = books.removeValueForKey(6)
//books = [:]
//removedValue = "Deleting from Dictionaries"
```

第11.6节：合并两个字典

```
扩展字典{
    函数 merge(dict: Dictionary<Key,Value>) -> Dictionary<Key,Value> {
        变量 mutableCopy = self
        对于(key, value) in dict {
            // 如果两个字典对同一个键都有值，则使用另一个字典的值。

mutableCopy[key] = value
        }
        返回 mutableCopy
    }
}
```


Section 11.6: Merge two dictionaries

```
extension Dictionary {
    func merge(dict: Dictionary<Key,Value>) -> Dictionary<Key,Value> {
        var mutableCopy = self
        for (key, value) in dict {
            // If both dictionaries have a value for same key, the value of the other dictionary is
used.
            mutableCopy[key] = value
        }
        return mutableCopy
    }
}
```

第12章：Switch

参数	详情
要测试的值	用于比较的变量

第12.1节：Switch语句和可选类型

当结果是可选类型时的一些示例情况。

```
var result: AnyObject? = someMethod()

switch result {
case nil:
     print("result 是空值")
case is String:
     print("result 是字符串")
case _ as Double:
     print("result 不是空值，且是Double类型的任意值")
case let myInt as Int where myInt > 0:
     print("\(myInt) 值不是空值，是一个大于0的整数")
case let a?:
     print("\(a) - 值已被解包")
}
```

第12.2节：基本使用

```
let number = 3
switch number {
case 1:
    print("One!")
case 2:
    print("Two!")
case 3:
    print("Three!")
default:
    print("Not One, Two or Three")
}
```

switch 语句也适用于除整数以外的数据类型。它们适用于任何数据类型。以下是一个针对字符串进行 switch 的示例：

```
let string = "Dog"
switch string {
case "Cat", "Dog":
  print("Animal is a house pet.")
default:
  print("Animal is not a house pet.")
}
```

这将打印以下内容：

```
动物是一只家养宠物。
```

第12.3节：匹配一个范围

switch语句中的单个case可以匹配一个值的范围。

Chapter 12: Switch

Parameter	Details
Value to test	The variable that to compare against

Section 12.1: Switch and Optionals

Some example cases when the result is an optional.

```
var result: AnyObject? = someMethod()

switch result {
case nil:
    print("result is nothing")
case is String:
    print("result is a String")
case _ as Double:
    print("result is not nil, any value that is a Double")
case let myInt as Int where myInt > 0:
    print("\(myInt) value is not nil but an int and greater than 0")
case let a?:
    print("\(a) - value is unwrapped")
}
```

Section 12.2: Basic Use

```
let number = 3
switch number {
case 1:
    print("One!")
case 2:
    print("Two!")
case 3:
    print("Three!")
default:
    print("Not One, Two or Three")
}
```

switch statements also work with data types other than integers. They work with any data type.Here's an example of switching on a string:

```
let string = "Dog"
switch string {
case "Cat", "Dog":
  print("Animal is a house pet.")
default:
  print("Animal is not a house pet.")
}
```

This will print the following:

```
Animal is a house pet.
```

Section 12.3: Matching a Range

A single case in a switch statement can match a range of values.

```
let number = 20
switch number {
case 0:
    print("零")
case 1..<10:
    print("介于一和十之间")
case 10..<20:
    print("介于十和二十之间")
case 20..<30:
    print("介于二十和三十之间")
default:
    print("大于三十或小于零")
}
```

第12.4节：部分匹配

Switch语句利用部分匹配。

```
let 坐标: (x: Int, y: Int, z: Int) = (3, 2, 5)

switch (坐标) {
case (0, 0, 0): // 1
   print("原点")
case (_, 0, 0): // 2
   print("在x轴上。")
case (0, _, 0): // 3
   print("在y轴上。")
case (0, 0, _): // 4
   print("在z轴上。")
default:        // 5
   print("空间中的某处")
}
```

1. 精确匹配值为 (0,0,0) 的情况。这是三维空间的原点。
2. 匹配 y=0，z=0，x 为任意值。这意味着坐标在 x 轴上。
3. 匹配 x=0，z=0，y 为任意值。这意味着坐标在 y 轴上。
4. 匹配 x=0，y=0，z 为任意值。这意味着坐标在 z 轴上。
5. 匹配其余的坐标。

注意：使用下划线表示你不关心该值。

如果你不想忽略该值，那么你可以在 switch 语句中使用它，像这样：

```
let 坐标: (x: Int, y: Int, z: Int) = (3, 2, 5)

switch (坐标) {
case (0, 0, 0):
   print("原点")
case (let x, 0, 0):
   print("x 轴上，x = \(x)")
case (0, let y, 0):
   print("y 轴上，y = \(y)")
case (0, 0, let z):
   print("z 轴上，z = \(z)")
case (let x, let y, let z):
   print("空间中某处，x = \(x), y = \(y), z = \(z)")
}
```

这里，轴的情况使用 let 语法提取相关的值。代码随后使用字符串插值来构建字符串并打印这些值。


```
let number = 20
switch number {
case 0:
    print("Zero")
case 1..<10:
    print("Between One and Ten")
case 10..<20:
    print("Between Ten and Twenty")
case 20..<30:
    print("Between Twenty and Thirty")
default:
    print("Greater than Thirty or less than Zero")
}
```

Section 12.4: Partial matching

Switch statement make use of partial matching.

```
let coordinates: (x: Int, y: Int, z: Int) = (3, 2, 5)

switch (coordinates) {
case (0, 0, 0): // 1
  print("Origin")
case (_, 0, 0): // 2
  print("On the x-axis.")
case (0, _, 0): // 3
  print("On the y-axis.")
case (0, 0, _): // 4
  print("On the z-axis.")
default:        // 5
  print("Somewhere in space")
}
```

1. Matches precisely the case where the value is (0,0,0). This is the origin of 3D space.
2. Matches y=0, z=0 and any value of x. This means the coordinate is on the x- axis.
3. Matches x=0, z=0 and any value of y. This means the coordinate is on they- axis.
4. Matches x=0, y=0 and any value of z. This means the coordinate is on the z- axis.
5. Matches the remainder of coordinates.

Note: using the underscore to mean that you don't care about the value.

If you don't want to ignore the value, then you can use it in your switch statement, like this:

```
let coordinates: (x: Int, y: Int, z: Int) = (3, 2, 5)

switch (coordinates) {
case (0, 0, 0):
  print("Origin")
case (let x, 0, 0):
  print("On the x-axis at x = \(x)")
case (0, let y, 0):
  print("On the y-axis at y = \(y)")
case (0, 0, let z):
  print("On the z-axis at z = \(z)")
case (let x, let y, let z):
  print("Somewhere in space at x = \(x), y = \(y), z = \(z)")
}
```

Here, the axis cases use the let syntax to pull out the pertinent values. The code then prints the values using string

插值。

注意：在此 switch 语句中不需要默认情况。这是因为最后一个 case 本质上就是默认情况—它匹配任何内容，因为元组的任何部分都没有限制。如果 switch 语句通过其所有 case 覆盖了所有可能的值，那么就不需要默认情况。

我们也可以使用 let-where 语法来匹配更复杂的情况。例如：

```
let 坐标: (x: Int, y: Int, z: Int) = (3, 2, 5)

switch (坐标) {
case (let x, let y, _) where y == x:
   print("沿着 y = x 线。")
case (let x, let y, _) where y == x * x:
   print("沿着 y = x² 线。")
default:
break
}
```

这里，我们匹配了“y 等于 x”和“y 等于 x 的平方”这两条线。

第12.5节：在 switch 中使用 where 语句

where 语句可以在 switch case 匹配中使用，以添加额外的条件来满足匹配要求。以下示例不仅检查范围，还检查数字是奇数还是偶数：

```
switch (temperature) {
        case 0...49 where temperature % 2 == 0:
           print("冷且为偶数")

        case 50...79 其中 temperature % 2 == 0:
           print("温暖且为偶数")

        case 80...110 其中 temperature % 2 == 0:
           print("热且为偶数")

        default:
           print("温度超出范围或为奇数")
}
```

第12.6节：匹配多个值

switch语句中的单个case可以匹配多个值。

```
let number = 3
switch number {
case 1, 2:
    print("One or Two!")
case 3:
    print("三！")
case 4, 5, 6:
    print("四、五或六！")
default:
    print("不是一、二、三、四、五或六")
}
```


interpolation to build the string.

Note: you don't need a default in this switch statement. This is because the final case is essentially the default—it matches anything, because there are no constraints on any part of the tuple. If the switch statement exhausts all possible values with its cases, then no default is necessary.

We can also use the let-where syntax to match more complex cases. For example:

```
let coordinates: (x: Int, y: Int, z: Int) = (3, 2, 5)

switch (coordinates) {
case (let x, let y, _) where y == x:
  print("Along the y = x line.")
case (let x, let y, _) where y == x * x:
  print("Along the y = x^2 line.")
default:
break
}
```

Here, We match the "y equals x" and "y equals x squared" lines.

Section 12.5: Using the where statement in a switch

The where statement may be used within a switch case match to add additional criteria required for a positive match. The following example checks not only for the range, but also if the number is odd or even:

```
switch (temperature) {
      case 0...49 where temperature % 2 == 0:
        print("Cold and even")

      case 50...79 where temperature % 2 == 0:
        print("Warm and even")

      case 80...110 where temperature % 2 == 0:
        print("Hot and even")

      default:
        print("Temperature out of range or odd")
}
```

Section 12.6: Matching Multiple Values

A single case in a switch statement can match on multiple values.

```
let number = 3
switch number {
case 1, 2:
    print("One or Two!")
case 3:
    print("Three!")
case 4, 5, 6:
    print("Four, Five or Six!")
default:
    print("Not One, Two, Three, Four, Five or Six")
}
```

第12.7节：Switch语句和枚举

Switch语句与枚举值配合得非常好

```
enum 车型 {
    case 标准, 快速, 非常快
}

let 车 = 车型.标准

switch 车 {
case .标准: print("标准")
case .快速: print("快速")
case .非常快: print("非常快")
}
```

> 由于我们为汽车的每个可能值都提供了一个案例，因此省略了default情况。

第12.8节：开关和元组

开关可以对元组进行切换：

```
public typealias mdyTuple = (month: Int, day: Int, year: Int)

let fredsBirthday =   (month: 4, day: 3, year: 1973)



switch theMDY
{
//你可以匹配一个字面量元组：
case (fredsBirthday):
message = "\(date) \(prefix) 弗雷德出生的那天"

//你可以匹配部分元素，忽略其他元素：
case (3, 15, _):
message = "当心三月的忌辰"

//你可以匹配字面量元组的部分元素，并将其他元素复制
//到一个常量中，在case体内使用：
case (bobsBirthday.month, bobsBirthday.day, let year) where year > bobsBirthday.year:
message = "\(date) \(prefix) Bob的 \(possessiveNumber(year - bobsBirthday.year))" +
    "生日"

//你可以将元组中的一个或多个元素复制到常量中，然后
//添加一个where子句以进一步限定条件：
case (susansBirthday.month, susansBirthday.day, let year)
  where year > susansBirthday.year:
message = "\(date) \(prefix) Susan的 " +
    "\(possessiveNumber(year - susansBirthday.year)) 生日"

//你可以将某些元素匹配到范围内：
case (5, 1...15, let year):
message = "\(date) \(prefix) 五月上半月，\(year)"
}
```


Section 12.7: Switch and Enums

The Switch statement works very well with Enum values

```
enum CarModel {
    case Standard, Fast, VeryFast
}

let car = CarModel.Standard

switch car {
case .Standard: print("Standard")
case .Fast: print("Fast")
case .VeryFast: print("VeryFast")
}
```

> Since we provided a case for each possible value of car, we omit the `default` case.

Section 12.8: Switches and tuples

Switches can switch on tuples:

```
public typealias mdyTuple = (month: Int, day: Int, year: Int)

let fredsBirthday =   (month: 4, day: 3, year: 1973)



switch theMDY
{
//You can match on a literal tuple:
case (fredsBirthday):
  message = "\(date) \(prefix) the day Fred was born"

//You can match on some of the terms, and ignore others:
case (3, 15, _):
  message = "Beware the Ides of March"

//You can match on parts of a literal tuple, and copy other elements
//into a constant that you use in the body of the case:
case (bobsBirthday.month, bobsBirthday.day, let year) where year > bobsBirthday.year:
  message = "\(date) \(prefix) Bob's \(possessiveNumber(year - bobsBirthday.year))" +
    "birthday"

//You can copy one or more elements of the tuple into a constant and then
//add a where clause that further qualifies the case:
case (susansBirthday.month, susansBirthday.day, let year)
  where year > susansBirthday.year:
  message = "\(date) \(prefix) Susan's " +
    "\(possessiveNumber(year - susansBirthday.year)) birthday"

//You can match some elements to ranges:.
case (5, 1...15, let year):
  message = "\(date) \(prefix) in the first half of May, \(year)"
}
```

第12.9节：使用switch满足多个约束中的一个

你可以创建一个元组并像这样使用switch：

```
var str: String? = "hi"
var x: Int? = 5

switch (str, x) {
case (.Some,.Some):
    print("两者都有值")
case (.Some, nil):
    print("字符串有值")
case (nil, .Some):
    print("整数有值")
case (nil, nil):
    print("两者都没有值")
}
```

第12.10节：基于类的匹配——非常适合prepareForSegue

你也可以让switch语句基于你切换对象的类进行切换。

一个有用的例子是在prepareForSegue中。我过去是基于segue标识符进行切换，但那很脆弱。如果你后来更改了故事板并重命名了segue标识符，代码就会出错。或者，如果你使用segue跳转到同一个视图控制器类的多个实例（但不同的故事板场景），那么你就不能用segue标识符来判断目标的类。

Swift的switch语句来帮忙。

使用Swift的`case let var as Class`语法，像这样：

版本 < 3.0

```
override func prepareForSegue(segue: UIStoryboardSegue, sender: AnyObject?) {
  switch segue.destinationViewController {
    case let fooViewController as FooViewController:
fooViewController.delegate = self

    case let barViewController as BarViewController:
barViewController.data = data

    default:
      break
  }
}
```

版本 ≥ 3.0

在 Swift 3 中语法略有变化：

```
override func prepare(for segue: UIStoryboardSegue, sender: AnyObject?) {
  switch segue.destinationViewController {
    case let fooViewController as FooViewController:
fooViewController.delegate = self

    case let barViewController as BarViewController:
barViewController.data = data

    default:
```


Section 12.9: Satisfy one of multiple constraints using switch

You can create a tuple and use a switch like so:

```
var str: String? = "hi"
var x: Int? = 5

switch (str, x) {
case (.Some,.Some):
    print("Both have values")
case (.Some, nil):
    print("String has a value")
case (nil, .Some):
    print("Int has a value")
case (nil, nil):
    print("Neither have values")
}
```

Section 12.10: Matching based on class - great for prepareForSegue

You can also make a switch statement switch based on the **class** of the thing you're switching on.

An example where this is useful is in `prepareForSegue`. I used to switch based on the segue identifier, but that's fragile. if you change your storyboard later and rename the segue identifier, it breaks your code. Or, if you use segues to multiple instances same view controller class (but different storyboard scenes) then you can't use the segue identifier to figure out the class of the destination.

Swift switch statements to the rescue.

Use Swift `case let var as Class` syntax, like this:

Version < 3.0

```
override func prepareForSegue(segue: UIStoryboardSegue, sender: AnyObject?) {
  switch segue.destinationViewController {
    case let fooViewController as FooViewController:
      fooViewController.delegate = self

    case let barViewController as BarViewController:
      barViewController.data = data

    default:
      break
  }
}
```

Version ≥ 3.0

In Swift 3 the sytax has changed slightly:

```
override func prepare(for segue: UIStoryboardSegue, sender: AnyObject?) {
  switch segue.destinationViewController {
    case let fooViewController as FooViewController:
      fooViewController.delegate = self

    case let barViewController as BarViewController:
      barViewController.data = data

    default:
```

```
        break
    }
}
```

第12.11节：Switch穿透

值得注意的是，在 Swift 中，与其他常见语言不同，每个 case 语句末尾都有一个隐式的 break。如果想继续执行下一个 case（即执行多个 case），需要使用fallthrough语句。

```
switch(value) {
case 'one':
    // 执行操作一
    fallthrough
case 'two':
    // 这可以独立完成，也可以与第一种情况结合使用
default:
    // 默认操作
}
```

这对于流等情况非常有用。


```
        break
    }
}
```

Section 12.11: Switch fallthroughs

It is worth noting that in swift, unlike other languages people are familiar with, there is an implicit break at the end of each case statement. In order to follow through to the next case (i.e. have multiple cases execute) you need to use `fallthrough` statement.

```
switch(value) {
case 'one':
    // do operation one
    fallthrough
case 'two':
    // do this either independant, or in conjunction with first case
default:
    // default operation
}
```

this is useful for things like streams.

第13章：可选项

“ 可选值要么包含一个值，要么包含nil以表示值缺失”摘自：Apple Inc. “Swift编程语言（Swift 3.1版）” iBooks. https://itun.es/us/k5SW7.l基本的可选用例包括：用于常量（let），在循环中使用可选（if-let），在方法中安全解包可选值（guard-let），以及作为switch循环的一部分（case-let），如果为nil则默认一个值，使用合并运算符（??）

第13.1节：可选类型

可选项是一个泛型枚举类型，充当包装器。该包装器允许变量具有两种状态之一：用户定义类型的值或 nil，表示值的缺失。

这一能力在Swift中特别重要，因为该语言的设计目标之一是与苹果的框架良好配合。许多（大多数）苹果框架利用 nil，因其易用性及其在Objective-C编程模式和API设计中的重要性。

在Swift中，变量要想拥有 nil值，必须是可选类型。可选项可以通过在类型后附加 ！或者对变量类型使用?。例如，要使一个Int变为可选类型，可以使用

```
var numberOne: Int! = nil
var numberTwo: Int? = nil
```

？可选类型必须显式解包，如果你不确定变量在访问时是否有值，应使用可选类型。例如，将字符串转换为Int时，结果是一个可选的Int?，因为如果字符串不是有效数字，将返回nil

```
let str1 = "42"
let num1: Int? = Int(str1) // 42

let str2 = "Hello, World!"
let num2: Int? = Int(str2) // nil
```

！可选类型会自动解包，且只应在你确定变量在访问时一定有值时使用。例如，一个在viewDidLoad()中初始化的全局UIButton!变量

```
//myButton 直到调用 viewDidLoad 之前不会被访问，
//所以这里可以使用 ！可选类型
var myButton: UIButton!

override func viewDidLoad(){
    self.myButton = UIButton(frame: self.view.frame)
    self.myButton.backgroundColor = UIColor.redColor()
    self.view.addSubview(self.myButton)
}
```

第13.2节：解包可选类型

为了访问 Optional 的值，需要对其进行解包。

你可以使用可选绑定来有条件地解包一个 Optional，也可以使用!操作符来强制解包一个 Optional。

Chapter 13: Optionals

“ An optional value either contains a value or contains nil to indicate that a value is missing”

Excerpt From: Apple Inc. “The Swift Programming Language (Swift 3.1 Edition).” iBooks. https://itun.es/us/k5SW7.l

Basic optional use cases include: for a constant (let), use of an optional within a loop (if-let), safely unwrapping an optional value within a method (guard-let), and as part of switch loops (case-let), defaulting to a value if nil, using the coalesce operator (??)

Section 13.1: Types of Optionals

Optionals are a generic enum type that acts as a wrapper. This wrapper allows a variable to have one of two states: the value of the user-defined type or `nil`, which represents the absence of a value.

This ability is particularly important in Swift because one of the stated design objectives of the language is to work well with Apple's frameworks. Many (most) of Apple's frameworks utilize `nil` due to its ease of use and significance to programming patterns and API design within Objective-C.

In Swift, for a variable to have a `nil` value, it must be an optional. Optionals can be created by appending either a `!` or a `?` to the variable type. For example, to make an `Int` optional, you could use

```
var numberOne: Int! = nil
var numberTwo: Int? = nil
```

? optionals must be explicitly unwrapped, and should be used if you aren't certain whether or not the variable will have a value when you access it. For example, when turning a string into an `Int`, the result is an optional `Int?`, because nil will be returned if the string is not a valid number

```
let str1 = "42"
let num1: Int? = Int(str1) // 42

let str2 = "Hello, World!"
let num2: Int? = Int(str2) // nil
```

! optionals are automatically unwrapped, and should *only* be used when you are *certain* that the variable will have a value when you access it. For example, a global `UIButton!` variable that is initialized in `viewDidLoad()`

```
//myButton will not be accessed until viewDidLoad is called,
//so a ! optional can be used here
var myButton: UIButton!

override func viewDidLoad(){
    self.myButton = UIButton(frame: self.view.frame)
    self.myButton.backgroundColor = UIColor.redColor()
    self.view.addSubview(self.myButton)
}
```

Section 13.2: Unwrapping an Optional

In order to access the value of an Optional, it needs to be unwrapped.

You can *conditionally unwrap* an Optional using optional binding and *force unwrap* an Optional using the ! operator.

有条件解包实际上是在问“这个变量有值吗？”，而强制解包则表示“这个变量有值！”。

如果你强制解包一个值为nil的变量，程序会抛出在解包可选值时意外发现 nil异常并崩溃，因此需要谨慎考虑是否适合使用!。

```
var text: String? = nil
var unwrapped: String = text! //崩溃，提示“在解包可选值时意外发现 nil”
```

为了安全解包，可以使用if-let语句，如果被包裹的值为 nil，则不会抛出异常或崩溃：

```
var number: Int?
if let unwrappedNumber = number {       // number 是否已被赋值？
    print("number: \(unwrappedNumber)") // 不会执行到这一行
} else {
    print("number 未被赋值")
}
```

或者，使用 guard 语句：

```
var number: Int?
guard let unwrappedNumber = number else {
    return
}
print("number: \(unwrappedNumber)")
```

注意，变量 unwrappedNumber 的作用域在 if-let 语句内部，而在 guard 代码块外部。

你可以链式解包多个可选值，这主要在代码需要多个变量才能正确运行的情况下非常有用：

```
var firstName:String?
var lastName:String?

if let fn = firstName, let ln = lastName {
    print("\(fn) + \(ln)")//注意，条件只有在两个可选值都不为 nil 时才为真。

}
```

请注意，所有变量都必须被解包才能成功通过测试，否则你将无法确定哪些变量被解包了，哪些没有。

你可以在可选值被解包后立即链式使用条件语句。这意味着不需要嵌套的 if - else 语句！

```
var firstName:String? = "Bob"
var myBool:Bool? = false

if let fn = firstName, fn == "Bob", let bool = myBool, !bool {
    print("firstName 是 bob 且 myBool 为 false！")
}
```


Conditionally unwrapping effectively asks "Does this variable have a value?" while force unwrapping says "This variable has a value!".

If you force unwrap a variable that is `nil`, your program will throw an *unexpectedly found nil while unwrapping an optional* exception and crash, so you need to consider carefully if using `!` is appropriate.

```
var text: String? = nil
var unwrapped: String = text! //crashes with "unexpectedly found nil while unwrapping an Optional value"
```

For safe unwrapping, you can use an `if-let` statement, which will not throw an exception or crash if the wrapped value is `nil`:

```
var number: Int?
if let unwrappedNumber = number {       // Has `number` been assigned a value?
    print("number: \(unwrappedNumber)") // Will not enter this line
} else {
    print("number was not assigned a value")
}
```

Or, a guard statement:

```
var number: Int?
guard let unwrappedNumber = number else {
    return
}
print("number: \(unwrappedNumber)")
```

Note that the scope of the `unwrappedNumber` variable is inside the `if-let` statement and outside of the `guard` block.

You can chain unwrapping of many optionals, this is mainly useful in cases that your code requires more then variable to run correctly:

```
var firstName:String?
var lastName:String?

if let fn = firstName, let ln = lastName {
    print("\(fn) + \(ln)")//pay attention that the condition will be true only if both optionals are not nil.
}
```

Note that all the variables have to be unwrapped in order to pass successfully the test, otherwise you would have no way to determine which variables were unwrapped and which weren't.

You can chain conditional statements using your optionals immediately after it is unwrapped. This means no nested if - else statements!

```
var firstName:String? = "Bob"
var myBool:Bool? = false

if let fn = firstName, fn == "Bob", let bool = myBool, !bool {
    print("firstName is bob and myBool was false!")
}
```

第 13.3 节：Nil 合并运算符

你可以使用nil 合并运算符来解包一个非 nil 的值，否则提供一个不同的值：

```
func fallbackIfNil(str: String?) -> String {
    return str ?? "备用字符串"
}
print(fallbackIfNil("Hi")) // 输出 "Hi"
print(fallbackIfNil(nil)) // 输出 "备用字符串"
```

该运算符能够短路，意味着如果左操作数非 nil，则右操作数不会被计算：

```
func someExpensiveComputation() -> String { ... }

var foo : String? = "一个字符串"
let str = foo ?? someExpensiveComputation()
```

在这个例子中，由于foo非空，someExpensiveComputation()将不会被调用。

你也可以将多个空合并操作符连锁使用：

```
var foo : String?
var bar : String?

let baz = foo ?? bar ?? "fallback string"
```

在这个例子中，baz将被赋值为foo的解包值（如果非空），否则将被赋值为bar的解包值（如果非空），否则将被赋值为备用值。

第13.4节：可选链

你可以使用可选链来调用方法、访问属性或对可选值进行下标操作。这是通过在给定的可选变量和指定成员（方法、属性或下标）之间放置一个?来实现的。

```
结构体 Foo {
    函数 doSomething() {
        print("Hello World!")
    }
}

var foo : Foo? = Foo()

foo?.doSomething() // 当 foo 非空时打印 "Hello World!"
```

如果foo包含一个值，doSomething()将会被调用。如果foo是nil，那么不会发生任何错误–代码将静默失败并继续执行。

```
var foo : Foo? = nil

foo?.doSomething() // 因为 foo 是 nil，所以不会被调用
```

(这与在 Objective-C 中向 nil发送消息的行为类似)

Optional Chaining 被命名为此的原因是因为'optionality'会通过你调用/访问的成员传播。这意味着使用可选链调用的任何成员的返回值

Section 13.3: Nil Coalescing Operator

You can use the *nil coalescing operator* to unwrap a value if it is non-nil, otherwise provide a different value:

```
func fallbackIfNil(str: String?) -> String {
    return str ?? "Fallback String"
}
print(fallbackIfNil("Hi")) // Prints "Hi"
print(fallbackIfNil(nil)) // Prints "Fallback String"
```

This operator is able to short-circuit, meaning that if the left operand is non-nil, the right operand will not be evaluated:

```
func someExpensiveComputation() -> String { ... }

var foo : String? = "a string"
let str = foo ?? someExpensiveComputation()
```

In this example, as `foo` is non-nil, `someExpensiveComputation()` will not be called.

You can also chain multiple nil coalescing statements together:

```
var foo : String?
var bar : String?

let baz = foo ?? bar ?? "fallback string"
```

In this example `baz` will be assigned the unwrapped value of `foo` if it is non-nil, otherwise it will be assigned the unwrapped value of `bar` if it is non-nil, otherwise it will be assigned the fallback value.

Section 13.4: Optional Chaining

You can use Optional Chaining in order to call a method, access a property or subscript an optional. This is done by placing a ? between the given optional variable and the given member (method, property or subscript).

```
struct Foo {
    func doSomething() {
        print("Hello World!")
    }
}

var foo : Foo? = Foo()

foo?.doSomething() // prints "Hello World!" as foo is non-nil
```

If `foo` contains a value, `doSomething()` will be called on it. If `foo` is `nil`, then nothing bad will happen – the code will simply fail silently and continue executing.

```
var foo : Foo? = nil

foo?.doSomething() // will not be called as foo is nil
```

(This is similar behaviour to sending messages to `nil` in Objective-C)

The reason that Optional Chaining is named as such is because 'optionality' will be propagated through the members you call/access. What this means is that the return values of any members used with optional chaining

将是可选的，无论它们是否被声明为可选类型。

```
结构体 Foo {
    var bar : Int
    func doSomething() { ... }
}

let foo : Foo? = Foo(bar: 5)
print(foo?.bar) // Optional(5)
```

这里foo?.bar返回的是一个Int?，尽管bar是非可选的，因为foo本身是可选的。

随着可选性的传播，返回Void的方法在通过可选链调用时将返回Void?。这在判断方法是否被调用（从而判断可选值是否存在）时非常有用。

```
let foo : Foo? = Foo()

if foo?.doSomething() != nil {
    print("foo 非空，且 doSomething() 被调用了")
} else {
    print("foo 为空，因此 doSomething() 没有被调用")
}
```

这里我们将Void?的返回值与nil进行比较，以判断方法是否被调用（从而判断foo是否非空）。

第13.5节：概述 - 为什么使用可选类型？

在编程中，经常需要区分一个变量是否有值。对于引用类型，如C语言指针，可以使用特殊值null来表示变量无值。对于内置类型，如整数，则更为困难。可以使用一个指定值，如-1，但这依赖于对该值的解释，同时也剥夺了该“特殊”值的正常使用。

为了解决这个问题，Swift允许将任何变量声明为可选类型。通过在类型后加上?或!来表示（参见可选类型的种类）。

例如，

```
var possiblyInt: Int?
```

声明一个可能包含整数值也可能不包含整数值的变量。

特殊值 nil 表示当前没有为该变量赋值。

```
possiblyInt = 5       // 现在 PossiblyInt 是 5
possiblyInt = nil     // 现在 PossiblyInt 未被赋值
```

nil 也可以用来测试是否有赋值：

```
if possiblyInt != nil {
    print("possiblyInt 的值是 \(possiblyInt!)")
}
```

注意在 print 语句中使用了 ！来“解包”可选值。

作为可选值常见用法的一个例子，考虑一个从包含字符串中返回整数的函数

will be optional, regardless of whether they are typed as optional or not.

```
struct Foo {
    var bar : Int
    func doSomething() { ... }
}

let foo : Foo? = Foo(bar: 5)
print(foo?.bar) // Optional(5)
```

Here `foo?.bar` is returning an `Int?` even though `bar` is non-optional, as `foo` itself is optional.

As optionality is propagated, methods returning `Void` will return `Void?` when called with optional chaining. This can be useful in order to determine whether the method was called or not (and therefore if the optional has a value).

```
let foo : Foo? = Foo()

if foo?.doSomething() != nil {
    print("foo is non-nil, and doSomething() was called")
} else {
    print("foo is nil, therefore doSomething() wasn't called")
}
```

Here we're comparing the `Void?` return value with `nil` in order to determine whether the method was called (and therefore whether `foo` is non-nil).

Section 13.5: Overview - Why Optionals?

Often when programming it is necessary to make some distinction between a variable that has a value and one that does not. For reference types, such as C Pointers, a special value such as `null` can be used to indicate that the variable has no value. For intrinsic types, such as an integer, it is more difficult. A nominated value, such as -1 can be used, but this relies on interpretation of the value. It also eliminates that "special" value from normal use.

To address this, Swift allows any variable to be declared as an optional. This is indicated by the use of a ? or ! after the type (See Types of optionals)

For example,

```
var possiblyInt: Int?
```

declares a variable that may or may not contain an integer value.

The special value `nil` indicates that no value is currently assigned to this variable.

```
possiblyInt = 5       // PossiblyInt is now 5
possiblyInt = nil     // PossiblyInt is now unassigned
```

`nil` can also be used to test for an assigned value:

```
if possiblyInt != nil {
    print("possiblyInt has the value \(possiblyInt!)")
}
```

Note the use of ! in the print statement to *unwrap* the optional value.

As an example of a common use of optionals, consider a function that returns an integer from a string containing

数字；字符串可能包含非数字字符，甚至可能为空。

一个返回简单Int的函数如何表示失败？它不能通过返回任何特定值来表示失败，因为这会导致该值无法从字符串中解析。

```
var someInt
someInt = parseInt("not an integer") // 这个函数如何表示失败？
```

然而，在Swift中，该函数可以简单地返回一个optional Int。失败则通过返回值为nil来表示。

```
var someInt?
someInt = parseInt("not an integer")  // 如果解析失败，该函数返回nil
if someInt == nil {
     print("这不是一个有效的整数")
}
```

digits; It is possible that the string may contain non-digit characters, or may even be empty.

How can a function that returns a simple `Int` indicate failure? It cannot do so by returning any specific value as this would preclude that value from being parsed from the string.

```
var someInt
someInt = parseInt("not an integer") // How would this function indicate failure?
```

In Swift, however, that function can simply return an *optional* Int. Then failure is indicated by return value of `nil`.

```
var someInt?
someInt = parseInt("not an integer")  // This function returns nil if parsing fails
if someInt == nil {
    print("That isn't a valid integer")
}
```

第14章：条件语句

条件表达式，涉及if、else if和else等关键字，使Swift程序能够根据布尔条件（真或假）执行不同的操作。本节介绍Swift条件语句、布尔逻辑和三元表达式的使用。

第14.1节：可选绑定和“where”子句

在大多数表达式中，必须先对可选值进行解包。 if let 是一种可选绑定，如果可选值不为nil，则绑定成功：

```
let num: Int? = 10 // 或者：let num: Int? = nil

if let unwrappedNum = num {
    // num 的类型是 Int?；unwrappedNum 的类型是 Int
    print("num 不是 nil: \(unwrappedNum + 1)")
} else {
    print("num 是 nil")
}
```

你可以为新绑定的变量重用相同的名字，从而遮蔽原变量：

```
// num 最初的类型是 Int?
if let num = num {
    // 在此代码块内，num 的类型是 Int
}
```

版本 ≥ 1.2 版本 < 3.0

使用逗号（,）组合多个可选绑定：

```
if let 解包的数字 = num, let 解包的字符串 = str {
    // 使用解包的数字和解包的字符串执行操作
} else if let 解包的数字 = num {
    // 使用解包的数字执行操作
} else {
    // num 是 nil
}
```

在可选绑定后使用 where 子句应用进一步的约束：

```
if let 解包的数字 = num 且 解包的数字 % 2 == 0 { print("num 非 nil，且是偶数") }
```

如果你想尝试，可以交错任意数量的可选绑定和 `where` 子句：

```
if let num = num                            // num 必须非 nil
    where num % 2 == 1,                     // num 必须是奇数
    let str = str,                          // str 必须非 nil
    let firstChar = str.characters.first    // str 也必须非空
    where firstChar != "x"                  // 第一个字符不能是 "x"
{
    // 所有绑定和条件都成功！
}
```

版本 ≥ 3.0

在 Swift 3 中，where 子句已被替代（SE-0099）：只需使用另一个 ，来分隔可选绑定和布尔条件。

Chapter 14: Conditionals

Conditional expressions, involving keywords such as if, else if, and else, provide Swift programs with the ability to perform different actions depending on a Boolean condition: True or False. This section covers the use of Swift conditionals, Boolean logic, and ternary statements.

Section 14.1: Optional binding and "where" clauses

Optionals must be *unwrapped* before they can be used in most expressions. `if let` is an *optional binding*, which succeeds if the optional value was **not** `nil`:

```
let num: Int? = 10 // or: let num: Int? = nil

if let unwrappedNum = num {
    // num has type Int?; unwrappedNum has type Int
    print("num was not nil: \(unwrappedNum + 1)")
} else {
    print("num was nil")
}
```

You can reuse the **same name** for the newly bound variable, shadowing the original:

```
// num originally has type Int?
if let num = num {
    // num has type Int inside this block
}
```

Version ≥ 1.2 Version < 3.0

Combine multiple optional bindings with commas (,):

```
if let unwrappedNum = num, let unwrappedStr = str {
    // Do something with unwrappedNum & unwrappedStr
} else if let unwrappedNum = num {
    // Do something with unwrappedNum
} else {
    // num was nil
}
```

Apply further constraints after the optional binding using a `where` clause:

```
if let unwrappedNum = num e unwrappedNum % 2 == 0 { print("num is non-nil, and it's an even number") }
```

If you're feeling adventurous, interleave any number of optional bindings and `where` clauses:

```
if let num = num                            // num must be non-nil
    where num % 2 == 1,                     // num must be odd
    let str = str,                          // str must be non-nil
    let firstChar = str.characters.first    // str must also be non-empty
    where firstChar != "x"                  // the first character must not be "x"
{
    // all bindings & conditions succeeded!
}
```

Version ≥ 3.0

In Swift 3, `where` clauses have been replaced (SE-0099): simply use another , to separate optional bindings and boolean conditions.

if let 解包的数字 = num解包的数字 % 2 == 0 { print("num 非 nil，且是偶数") } if let num = num, // num 必须非 nil num % 2 == 1, // num 必须是奇数 let str = str, // str 必须非 nil let firstChar = str.characters.first, // str 也必须非空 firstChar != "x" // 第一个字符不能是 "x" { // 所有绑定和条件都成功！ }

第14.2节：使用 Guard

版本 ≥ 2.0

守卫（guard）检查一个条件，如果条件为假，则进入分支。守卫检查分支必须通过return、break或continue（如果适用）离开其包含的代码块；否则会导致编译错误。这样做的优点是，当写guard时，不可能让流程意外继续（而使用if则可能发生这种情况）。

使用守卫可以帮助保持嵌套层级较低，这通常会提高代码的可读性。

```
func printNum(num: Int) {
    guard num == 10 else {
        print("num 不是 10")
        return
    }
    print("num 是 10")
}
```

守卫还可以检查可选值中是否有值，然后在外层作用域中解包该值：

```
func printOptionalNum(num: Int?) {
    guard let unwrappedNum = num else {
        print("num 不存在")
        return
    }
    print(unwrappedNum)
}
```

守卫可以使用where关键字结合可选值解包和条件检查：

```
func printOptionalNum(num: Int?) {
guard let unwrappedNum = num, unwrappedNum == 10 else {
    print("num 不存在或不是 10")
    return
}
print(unwrappedNum)
}
```

第14.3节：基本条件语句：if语句

一个if语句检查一个Bool条件是否为true：

```
let num = 10

if num == 10 {
    // 只有当条件为真时，代码块内的代码才会执行。
    print("num 是 10")
}

let condition = num == 10    // condition 的类型是 Bool
if condition {
```


if let unwrappedNum = numwrappedNum % 2 == 0 { print("num is non-nil, and it's an even number") } if let num = num, // num must be non-nil num % 2 == 1, // num must be odd let str = str, // str must be non-nil let firstChar = str.characters.first, // str must also be non-empty firstChar != "x" // the first character must not be "x" { // all bindings & conditions succeeded! }

Section 14.2: Using Guard

Version ≥ 2.0

Guard checks for a condition, and if it is false, it enters the branch. Guard check branches must leave its enclosing block either via `return`, `break`, or `continue` (if applicable); failing to do so results in a compiler error. This has the advantage that when a `guard` is written it's not possible to let the flow continue accidentally (as would be possible with an `if`).

Using guards can help keep nesting levels low, which usually improves the readability of the code.

```
func printNum(num: Int) {
    guard num == 10 else {
        print("num is not 10")
        return
    }
    print("num is 10")
}
```

Guard can also check if there is a value in an optional, and then unwrap it in the outer scope:

```
func printOptionalNum(num: Int?) {
    guard let unwrappedNum = num else {
        print("num does not exist")
        return
    }
    print(unwrappedNum)
}
```

Guard can combine optional unwrapping and condition check using `where` keyword:

```
func printOptionalNum(num: Int?) {
guard let unwrappedNum = num, unwrappedNum == 10 else {
    print("num does not exist or is not 10")
    return
}
print(unwrappedNum)
}
```

Section 14.3: Basic conditionals: if-statements

An **`if` statement** checks whether a Bool condition is `true`:

```
let num = 10

if num == 10 {
    // Code inside this block only executes if the condition was true.
    print("num is 10")
}

let condition = num == 10   // condition's type is Bool
if condition {
```

```
    print("num 是 10")
}
```

if语句支持else if和else块，可以测试备选条件并提供备用方案：

```
let num = 10
if num < 10 {  // 如果第一个条件为真，执行以下代码。
    print("num 小于 10")
} else if num == 10 {  // 否则，检查下一个条件...
    print("num 是 10")
} else {  // 如果其他条件都不满足...
    print("所有其他条件都为假，所以 num 大于 10")
}
```

基本运算符如 && 和 || 可用于多个条件：

逻辑与运算符

```
let num = 10
let str = "Hi"
if num == 10 && str == "Hi" {
    print("num 是 10，且 str 是 \"Hi\"")
}
```

如果 num == 10 为假，第二个值将不会被计算。这称为短路求值。

逻辑或运算符

```
if num == 10 || str == "Hi" {
    print("num 是 10，或者 str 是 \"Hi\"")
}
```

如果 num == 10 为真，第二个值将不会被计算。

逻辑非运算符

```
if !str.isEmpty {
    print("str is not empty")
}
```

第14.4节：三元运算符

条件也可以使用三元运算符在一行内进行判断：

如果你想确定两个变量的最小值和最大值，可以使用if语句，如下所示：

```
let a = 5
let b = 10
let min: Int

if a < b {
    min = a
} else {
    min = b
}

let max: Int

if a > b {
    max = a
```



```
    print("num is 10")
}
```

`if` statements accept **`else if`** and **`else`** blocks, which can test alternate conditions and provide a fallback:

```
let num = 10
if num < 10 {  // Execute the following code if the first condition is true.
    print("num is less than 10")
} else if num == 10 {  // Or, if not, check the next condition...
    print("num is 10")
} else {  // If all else fails...
    print("all other conditions were false, so num is greater than 10")
}
```

Basic operators like `&&` and `||` can be used for multiple conditions:

The logical AND operator

```
let num = 10
let str = "Hi"
if num == 10 && str == "Hi" {
    print("num is 10, AND str is \"Hi\"")
}
```

If `num == 10` was false, the second value wouldn't be evaluated. This is known as short-circuit evaluation.

The logical OR operator

```
if num == 10 || str == "Hi" {
    print("num is 10, or str is \"Hi\")
}
```

If `num == 10` is true, the second value wouldn't be evaluated.

The logical NOT operator

```
if !str.isEmpty {
    print("str is not empty")
}
```

Section 14.4: Ternary operator

Conditions may also be evaluated in a single line using the ternary operator:

If you wanted to determine the minimum and maximum of two variables, you could use if statements, like so:

```
let a = 5
let b = 10
let min: Int

if a < b {
    min = a
} else {
    min = b
}

let max: Int

if a > b {
    max = a
```

```
} else {
    max = b
}
```

三元条件运算符根据条件的真假返回两个值中的一个。语法如下：这相当于表达式：

```
(<CONDITION>) ? <TRUE VALUE> : <FALSE VALUE>
```

上述代码可以使用三元条件运算符重写如下：

```
let a = 5
let b = 10
let min = a < b ? a : b
let max = a > b ? a : b
```

在第一个例子中，条件是 a < b。如果为真，赋值给 min 的结果将是 a；如果为假，结果将是 b 的值。

注意：因为找出两个数字中较大或较小的值是一个非常常见的操作，Swift 标准库提供了两个函数来实现此目的：max 和 min。

第14.5节：空合运算符

空合运算符 <OPTIONAL> ?? <DEFAULT VALUE> 如果 <OPTIONAL> 包含值，则解包该值；如果为 nil，则返回 <DEFAULT VALUE>。 <OPTIONAL> 始终是可选类型。 <DEFAULT VALUE> 必须与存储在 <OPTIONAL> 中的类型匹配。

空合运算符是下面使用三元运算符代码的简写：

```
a != nil ? a! : b
```

这可以通过下面的代码验证：

```
(a ?? b) == (a != nil ? a! : b) // 输出 true
```

举个例子

```
let defaultSpeed:String = "慢"
var userEnteredSpeed:String? = nil

print(userEnteredSpeed ?? defaultSpeed) // 输出 "慢"

userEnteredSpeed = "快"
print(userEnteredSpeed ?? defaultSpeed) // 输出 "快"
```



```
} else {
    max = b
}
```

The ternary conditional operator takes a condition and returns one of two values, depending on whether the condition was true or false. The syntax is as follows: This is equivalent of having the expression:

```
(<CONDITION>) ? <TRUE VALUE> : <FALSE VALUE>
```

The above code can be rewritten using ternary conditional operator as below:

```
let a = 5
let b = 10
let min = a < b ? a : b
let max = a > b ? a : b
```

In the first example, the condition is a < b. If this is true, the result assigned back to min will be of a; if it's false, the result will be the value of b.

Note: Because finding the greater or smaller of two numbers is such a common operation, the Swift standard library provides two functions for this purpose: max and min.

Section 14.5: Nil-Coalescing Operator

The nil-coalescing operator **<OPTIONAL>** ?? **<DEFAULT** VALUE**>** unwraps the **<OPTIONAL>** if it contains a value, or returns **<DEFAULT** VALUE**>** if is nil. **<OPTIONAL>** is always of an optional type. **<DEFAULT** VALUE**>** must match the type that is stored inside **<OPTIONAL>**.

The nil-coalescing operator is shorthand for the code below that uses a ternary operator:

```
a != nil ? a! : b
```

this can be verified by the code below:

```
(a ?? b) == (a != nil ? a! : b) // outputs true
```

Time For An Example

```
let defaultSpeed:String = "Slow"
var userEnteredSpeed:String? = nil

print(userEnteredSpeed ?? defaultSpeed) // outputs "Slow"

userEnteredSpeed = "Fast"
print(userEnteredSpeed ?? defaultSpeed) // outputs "Fast"
```

第15章：错误处理

第15.1节：错误处理基础

Swift中的函数可以返回值、抛出错误，或者两者兼有：

```
func reticulateSplines()                    // 无返回值且不抛出错误
func reticulateSplines()
```

nt // 总是返回一个值 func reticulateSplines() throws // 无返回值，但可能抛出错误 func reticulateSplines() throws -> *Int* // 可能返回值或抛出错误

任何符合ErrorType协议（包括NSError对象）的值都可以被抛出作为错误。
枚举提供了一种方便定义自定义错误的方式：

版本 ≥ 2.0 版本 ≤ 2.2

```
enum NetworkError: ErrorType {
    case Offline
    case ServerError(String)
}
```

版本 = 3.0

```
enum NetworkError: Error {
    // Swift 3 规定枚举案例应为 `lowerCamelCase`
    case offline
    case serverError(String)
}
```

错误表示程序执行期间的非致命失败，并通过专门的控制流结构 do/catch、throw 和 try 进行处理。

```
func fetchResource(resource: NSURL) throws -> String {
    if let (statusCode, responseString) = /* ...from elsewhere...*/ {
        if case 500..<600 = statusCode {
throw NetworkError.serverError(responseString)
        } else {
            return responseString
        }
    } 否则 {
抛出 NetworkError。offline
    }
}
```

错误可以通过 do/catch 捕获：

```
do {
    let response = try fetchResource(resURL)
    // 如果 fetchResource() 没有抛出错误，执行将继续到这里：
    print("收到响应：\\(response)")
    ...
} catch {
    // 如果抛出错误，我们可以在这里处理。
    print("哎呀，无法获取资源：\\(error)")
}
```

任何可能抛出错误的函数 必须 使用 try、try? 或 try! 调用：

```
// 错误：调用可能抛出错误但未使用 'try' 标记
```


Chapter 15: Error Handling

Section 15.1: Error handling basics

Functions in Swift may return values, **throw** errors, or both:

```
func reticulateSplines()                    // no return value and no error
func reticulateSplines()
```

nt // always returns a value func reticulateSplines() throws // no return value, but may throw an error func reticulateSplines() throws -> *Int* // may either return a value or throw an error

Any value which conforms to the ErrorType protocol (including NSError objects) can be thrown as an error. Enumerations provide a convenient way to define custom errors:

Version ≥ 2.0 Version ≤ 2.2

```
enum NetworkError: ErrorType {
    case Offline
    case ServerError(String)
}
```

Version = 3.0

```
enum NetworkError: Error {
    // Swift 3 dictates that enum cases should be `lowerCamelCase`
    case offline
    case serverError(String)
}
```

An error indicates a non-fatal failure during program execution, and is handled with the specialized control-flow constructs `do`/`catch`, `throw`, and `try`.

```
func fetchResource(resource: NSURL) throws -> String {
    if let (statusCode, responseString) = /* ...from elsewhere...*/ {
        if case 500..<600 = statusCode {
            throw NetworkError.serverError(responseString)
        } else {
            return responseString
        }
    } else {
        throw NetworkError.offline
    }
}
```

Errors can be caught with `do`/`catch`:

```
do {
    let response = try fetchResource(resURL)
    // If fetchResource() didn't throw an error, execution continues here:
    print("Got response: \\(response)")
    ...
} catch {
    // If an error is thrown, we can handle it here.
    print("Whoops, couldn't fetch resource: \\(error)")
}
```

Any function which can throw an error ***must*** be called using `try`, `try?`, or `try!`:

```
// error: call can throw but is not marked with 'try'
```

```
let response = fetchResource(resURL)

// “try” 在 do/catch 中使用，或在另一个可抛出函数中使用：
do {
    let response = try fetchResource(resURL)
} catch {
    // 处理错误
}

func foo() throws {
    // 如果抛出错误，继续将其传递给调用者。
    let response = try fetchResource(resURL)
}

// “try?” 将函数的返回值包装为可选类型（如果抛出错误则为 nil）。
if let response = try? fetchResource(resURL) {
    // 没有抛出错误
}

// “try!” 如果发生错误，运行时会导致程序崩溃。
let response = try! fetchResource(resURL)
```

第15.2节：捕获不同类型的错误

让我们为这个例子创建自己的错误类型。

版本 = 2.2

```
enum CustomError: ErrorType {
    case SomeError
    case AnotherError
}

func throwing() throws {
    throw CustomError.SomeError
}
```

版本 = 3.0

```
enum CustomError: Error {
    case someError
    case anotherError
}

func throwing() throws {
    throw CustomError.someError
}
```

Do-Catch 语法允许捕获抛出的错误，并且*自动*创建一个名为`error`的常量，该常量在catch块中可用：

```
do {
try throwing()
} catch {
    print(error)
}
```

你也可以自己声明一个变量：

```
do {
try throwing()
} catch let oops {
```



```
let response = fetchResource(resURL)

// "try" works within do/catch, or within another throwing function:
do {
    let response = try fetchResource(resURL)
} catch {
    // Handle the error
}

func foo() throws {
    // If an error is thrown, continue passing it up to the caller.
    let response = try fetchResource(resURL)
}

// "try?" wraps the function's return value in an Optional (nil if an error was thrown).
if let response = try? fetchResource(resURL) {
    // no error was thrown
}

// "try!" crashes the program at runtime if an error occurs.
let response = try! fetchResource(resURL)
```

Section 15.2: Catching different error types

Let's create our own error type for this example.

Version = 2.2

```
enum CustomError: ErrorType {
    case SomeError
    case AnotherError
}

func throwing() throws {
    throw CustomError.SomeError
}
```

Version = 3.0

```
enum CustomError: Error {
    case someError
    case anotherError
}

func throwing() throws {
    throw CustomError.someError
}
```

The Do-Catch syntax allows to catch a thrown error, and *automatically* creates a constant named `error` available in the `catch` block:

```
do {
    try throwing()
} catch {
    print(error)
}
```

You can also declare a variable yourself:

```
do {
    try throwing()
} catch let oops {
```

```
    print(oops)
}
```

也可以链式使用不同的catch语句。如果在Do代码块中可能抛出多种类型的错误，这样做很方便。

这里，Do-Catch会先尝试将错误转换为CustomError类型，如果自定义类型不匹配，则转换为NSError类型。

版本 = 2.2

```
do {
try somethingMayThrow()
} catch let custom as CustomError {
    print(custom)
} catch let error as NSError {
    print(error)
}
```

版本 = 3.0

在Swift 3中，不需要显式地向下转换为NSError。

```
do {
try somethingMayThrow()
} catch let custom as CustomError {
    print(custom)
} catch {
    print(error)
}
```

第15.3节：用于显式错误处理的Catch和Switch模式

```
class Plane {

    枚举 Emergency: ErrorType {
        case 无燃料
        case 引擎故障(原因: 字符串)
        case 机翼受损
    }

    变量 燃料公斤数: 整数

    //... 初始化及其他方法未显示

    函数 飞行() 抛出错误 {
        // ...
        如果 燃料公斤数 <= 0 {
            // 哎呀...
抛出 Emergency.无燃料
        }
    }

}
```

在客户端类中:

```
让 空军一号 = 飞机()
尝试 {
```



```
    print(oops)
}
```

It's also possible to chain different `catch` statements. This is convenient if several types of errors can be thrown in the Do block.

Here the Do-Catch will first attempt to cast the error as a `CustomError`, then as an `NSError` if the custom type was not matched.

Version = 2.2

```
do {
    try somethingMayThrow()
} catch let custom as CustomError {
    print(custom)
} catch let error as NSError {
    print(error)
}
```

Version = 3.0

In Swift 3, no need to explicitly downcast to NSError.

```
do {
    try somethingMayThrow()
} catch let custom as CustomError {
    print(custom)
} catch {
    print(error)
}
```

Section 15.3: Catch and Switch Pattern for Explicit Error Handling

```
class Plane {

    enum Emergency: ErrorType {
        case NoFuel
        case EngineFailure(reason: String)
        case DamagedWing
    }

    var fuelInKilograms: Int

    //... init and other methods not shown

    func fly() throws {
        // ...
        if fuelInKilograms <= 0 {
            // uh oh...
            throw Emergency.NoFuel
        }
    }

}
```

In the client class:

```
let airforceOne = Plane()
do {
```

```
尝试 空军一号.飞行()
} 捕获 让 紧急情况 作为 飞机.Emergency {
    切换 紧急情况 {
    case .无燃料:
        // 呼叫最近的机场进行紧急降落
    case .发动机故障(let 原因):
        print(原因) // 让机械师知道原因
    case .机翼受损:
        // 评估损坏情况并确定总统是否能继续飞行
    }
}
```

第15.4节：禁用错误传播

Swift的创建者非常注重使语言具有表现力，错误处理正是如此，具有表现力。如果你尝试调用一个可能抛出错误的函数，函数调用前需要加上try关键字。try关键字并非魔法。它所做的只是让开发者意识到该函数具有抛出错误的能力。

例如，以下代码使用了loadImage(atPath:)函数，该函数会加载给定路径的图像资源，或者在无法加载图像时抛出错误。在这种情况下，因为图像随应用程序一起发布，运行时不会抛出错误，所以禁用错误传播是合适的。

```
let photo = try! loadImage(atPath: "./Resources/John Appleseed.jpg")
```

第15.5节：创建带有本地化描述的自定义错误

创建自定义错误的枚举

```
enum RegistrationError: Error {
    case invalidEmail
    case invalidPassword
    case invalidPhoneNumber
}
```

创建RegistrationError的扩展以处理本地化描述。

```
extension RegistrationError: LocalizedError {
    public var errorDescription: String? {
        switch self {
        case .invalidEmail:
            return NSLocalizedString("无效电子邮件地址的描述", comment: "无效
电子邮件")
        case .invalidPassword:
            return NSLocalizedString("无效密码的描述", comment: "无效
密码")
        case .invalidPhoneNumber:
            return NSLocalizedString("无效电话号码的描述", comment: "无效电话号码")

        }
    }
}
```

处理错误：

```
let error: Error = RegistrationError.invalidEmail
```



```
    try airforceOne.fly()
} catch let emergency as Plane.Emergency {
    switch emergency {
    case .NoFuel:
        // call nearest airport for emergency landing
    case .EngineFailure(let reason):
        print(reason) // let the mechanic know the reason
    case .DamagedWing:
        // Assess the damage and determine if the president can make it
    }
}
```

Section 15.4: Disabling Error Propagation

The creators of Swift have put a lot of attention into making the language expressive and error handling is exactly that, expressive. If you try to invoke a function that can throw an error, the function call needs to be preceded by the try keyword. The try keyword isn't magical. All it does, is make the developer aware of the throwing ability of the function.

For example, the following code uses a loadImage(atPath:) function, which loads the image resource at a given path or throws an error if the image can't be loaded. In this case, because the image is shipped with the application, no error will be thrown at runtime, so it is appropriate to disable error propagation.

```
let photo = try! loadImage(atPath: "./Resources/John Appleseed.jpg")
```

Section 15.5: Create custom Error with localized description

Create `enum` of custom errors

```
enum RegistrationError: Error {
    case invalidEmail
    case invalidPassword
    case invalidPhoneNumber
}
```

Create `extension` of `RegistrationError` to handle the Localized description.

```
extension RegistrationError: LocalizedError {
    public var errorDescription: String? {
        switch self {
        case .invalidEmail:
            return NSLocalizedString("Description of invalid email address", comment: "Invalid
Email")
        case .invalidPassword:
            return NSLocalizedString("Description of invalid password", comment: "Invalid
Password")
        case .invalidPhoneNumber:
            return NSLocalizedString("Description of invalid phoneNumber", comment: "Invalid Phone
Number")
        }
    }
}
```

Handle error:

```
let error: Error = RegistrationError.invalidEmail
```

```
print(error.localizedDescription)
```



```
print(error.localizedDescription)
```

第16章：循环

第16.1节：for-in循环

for-in循环允许你遍历任何序列。

遍历范围

你可以遍历半开区间和闭区间：

```
for i in 0..<3 {
    print(i)
}

for i in 0...2 {
    print(i)
}

// 两者都会打印：
// 0
// 1
// 2
```

遍历数组或集合

```
let names = ["James", "Emily", "Miles"]

for name in names {
   print(name)
}

// James
// Emily
// Miles
```

版本 = 2.1 版本 = 2.2

如果你需要数组中每个元素的索引，可以在SequenceType上使用 enumerate() 方法。

```
for (index, name) in names.enumerate() {
    print("\(name) 的索引是 \(index)。")
}

// James 的索引是 0。
// Emily 的索引是 1。
// Miles 的索引是 2。
```

enumerate() 返回一个惰性序列，包含从 0 开始的连续Int元素对。因此对于数组，这些数字对应每个元素的索引–但对于其他类型的集合，情况可能不同。

版本 ≥ 3.0

在 Swift 3 中，enumerate() 已重命名为 enumerated()：

```
for (index, name) in names.enumerated() {
    print("名称 \(name) 的索引是 \(index)。")
}
```


Chapter 16: Loops

Section 16.1: For-in loop

The **for-in** loop allows you to iterate over any sequence.

Iterating over a range

You can iterate over both half-open and closed ranges:

```
for i in 0..<3 {
    print(i)
}

for i in 0...2 {
    print(i)
}

// Both print:
// 0
// 1
// 2
```

Iterating over an array or set

```
let names = ["James", "Emily", "Miles"]

for name in names {
   print(name)
}

// James
// Emily
// Miles
```

Version = 2.1 Version = 2.2

If you need the index for each element in the array, you can use the `enumerate()` method on `SequenceType`.

```
for (index, name) in names.enumerate() {
   print("The index of \(name) is \(index).")
}

// The index of James is 0.
// The index of Emily is 1.
// The index of Miles is 2.
```

`enumerate()` returns a lazy sequence containing pairs of elements with consecutive `Int`s, starting from 0. Therefore with arrays, these numbers will correspond to the given index of each element – however this may not be the case with other kinds of collections.

Version ≥ 3.0

In Swift 3, `enumerate()` has been renamed to `enumerated()`:

```
for (index, name) in names.enumerated() {
   print("The index of \(name) is \(index).")
}
```

遍历字典

```
let ages = ["詹姆斯": 29, "艾米丽": 24]

for (name, age) in ages {
     print(name, "是", age, "岁。")
}

// 艾米丽 24 岁。
// 詹姆斯 29 岁。
```

反向遍历

版本 = 2.1 版本 = 2.2

你可以对 SequenceType 使用 reverse() 方法，以便反向遍历任何序列：

```
for i in (0..<3).reverse() {
    print(i)
}

对于 i 在(0...2).reverse() {
    打印(i)
}

// 两者都会打印：
// 2
// 1
// 0

让 names = ["James", "Emily", "Miles"]

对于 name 在 names.reverse() {
    print(name)
}

// Miles
// Emily
// James
```

版本 ≥ 3.0

在 Swift 3 中，reverse() 已重命名为 reversed()：

```
对于 i 在(0..<3).reversed() {
    打印(i)
}
```

使用自定义步长遍历范围

版本 = 2.1 版本 = 2.2

通过在 Strideable 上使用 stride(_:_:) 方法，你可以使用自定义步长遍历范围：

```
for i in 4.stride(to: 0, by: -2) {
    print(i)
}

// 4
// 2

for i in 4.stride(through: 0, by: -2) {
    print(i)
}
```


Iterating over a dictionary

```
let ages = ["James": 29, "Emily": 24]

for (name, age) in ages {
    print(name, "is", age, "years old.")
}

// Emily is 24 years old.
// James is 29 years old.
```

Iterating in reverse

Version = 2.1 Version = 2.2

You can use the reverse() method on SequenceType in order to iterate over any sequence in reverse:

```
for i in (0..<3).reverse() {
    print(i)
}

for i in (0...2).reverse() {
    print(i)
}

// Both print:
// 2
// 1
// 0

let names = ["James", "Emily", "Miles"]

for name in names.reverse() {
    print(name)
}

// Miles
// Emily
// James
```

Version ≥ 3.0

In Swift 3, reverse() has been renamed to reversed():

```
for i in (0..<3).reversed() {
    print(i)
}
```

Iterating over ranges with custom stride

Version = 2.1 Version = 2.2

By using the stride(_:_:) methods on Strideable you can iterate over a range with a custom stride:

```
for i in 4.stride(to: 0, by: -2) {
    print(i)
}

// 4
// 2

for i in 4.stride(through: 0, by: -2) {
    print(i)
}
```

```
// 4
// 2
// 0
```

版本 = 1.2 版本 ≥ 3.0

在 Swift 3 中，stride(_:_:) 方法在 Stridable 上已被全局的 stride(_:_:_:) 函数取代：

```
for i in stride(from: 4, to: 0, by: -2) {
    print(i)
}

for i in stride(from: 4, through: 0, by: -2) {
    print(i)
}
```

第16.2节：重复-while循环

类似于 while 循环，只不过控制语句是在循环之后进行判断。因此，循环至少会执行一次。

```
var i: Int = 0

repeat {
   print(i)
   i += 1
} while i < 3

// 0
// 1
// 2
```

第16.3节：带过滤条件的for-in循环

1. **where子句**

通过添加一个where子句，你可以限制迭代只针对满足给定条件的元素。

```
for i in 0..<5 where i % 2 == 0 {
    print(i)
}

// 0
// 2
// 4


let names = ["James", "Emily", "Miles"]

for name in names where name.characters.contains("s") {
    print(name)
}

// James
// Miles
```

2. **case子句**

当你只需要遍历匹配某种模式的值时，这很有用：


```
// 4
// 2
// 0
```

Version = 1.2 Version ≥ 3.0

In Swift 3, the stride(_:_:) methods on Stridable have been replaced by the global stride(_:_:_:) functions:

```
for i in stride(from: 4, to: 0, by: -2) {
    print(i)
}

for i in stride(from: 4, through: 0, by: -2) {
    print(i)
}
```

Section 16.2: Repeat-while loop

Similar to the while loop, only the control statement is evaluated after the loop. Therefore, the loop will always execute at least once.

```
var i: Int = 0

repeat {
   print(i)
   i += 1
} while i < 3

// 0
// 1
// 2
```

Section 16.3: For-in loop with filtering

1. **where** clause

By adding a where clause you can restrict the iterations to ones that satisfy the given condition.

```
for i in 0..<5 where i % 2 == 0 {
    print(i)
}

// 0
// 2
// 4


let names = ["James", "Emily", "Miles"]

for name in names where name.characters.contains("s") {
    print(name)
}

// James
// Miles
```

2. **case** clause

It's useful when you need to iterate only through the values that match some pattern:

```swift
let points = [(5, 0), (31, 0), (5, 31)]
for case (_, 0) in points {
    print("点在x轴上")
}

//点在x轴上
//点在x轴上
```

你也可以通过在绑定常量后添加?标记来过滤可选值并在适当时解包它们：

```swift
let optionalNumbers = [31, 5, nil]
for case let number? in optionalNumbers {
    print(number)
}

//31
//5
```

第16.4节：序列类型的forEach块

符合SequenceType协议的类型可以在闭包中遍历其元素：

```
collection.forEach { print($0) }
```

同样也可以使用命名参数来完成：

```swift
collection.forEach { item in
    print(item)
}
```

*注意：这些代码块中不能使用控制流语句（如 break 或 continue）。可以调用 return，如果调用，将立即返回当前迭代的代码块（类似于 continue 的效果）。然后执行下一次迭代。

```swift
let arr = [1,2,3,4]

arr.forEach {

    // 3 和 4 的代码块仍然会被调用
    if $0 == 2 {
        return
    }
}
```

第16.5节：while 循环

只要条件为真，while 循环就会执行。

```swift
var count = 1

while count < 10 {
    print("这是第 \(count) 次循环")
    count += 1
}
```



```swift
let points = [(5, 0), (31, 0), (5, 31)]
for case (_, 0) in points {
    print("point on x-axis")
}

//point on x-axis
//point on x-axis
```

Also you can filter optional values and unwrap them if appropriate by adding ? mark after binding constant:

```swift
let optionalNumbers = [31, 5, nil]
for case let number? in optionalNumbers {
    print(number)
}

//31
//5
```

Section 16.4: Sequence Type forEach block

A type that conforms to the SequenceType protocol can iterate through it's elements within a closure:

```swift
collection.forEach { print($0) }
```

The same could also be done with a named parameter:

```swift
collection.forEach { item in
    print(item)
}
```

*Note: Control flow statements (such as break or continue) may not be used in these blocks. A return can be called, and if called, will immediately return the block for the current iteration (much like a continue would). The next iteration will then execute.

```swift
let arr = [1,2,3,4]

arr.forEach {

    // blocks for 3 and 4 will still be called
    if $0 == 2 {
        return
    }
}
```

Section 16.5: while loop

A `while` loop will execute as long as the condition is true.

```swift
var count = 1

while count < 10 {
    print("This is the \(count) run of the loop")
    count += 1
}
```

第16.6节：跳出循环

只要条件为真，循环就会执行，但你可以使用break关键字手动停止它。例如：

```
var peopleArray = ["John", "Nicole", "Thomas", "Richard", "Brian", "Novak", "Vick", "Amanda",
"Sonya"]
var positionOfNovak = 0

for person in peopleArray {
    if person == "Novak" { break }
    positionOfNovak += 1
}

print("Novak 是 peopleArray 中位置为 [\(positionOfNovak)] 的元素。")
//输出：Novak 是 peopleArray 中位置为 5 的元素。（正确）
```


Section 16.6: Breaking a loop

A loop will execute as long as its condition remains true, but you can stop it manually using the **`break`** keyword. For example:

```
var peopleArray = ["John", "Nicole", "Thomas", "Richard", "Brian", "Novak", "Vick", "Amanda",
"Sonya"]
var positionOfNovak = 0

for person in peopleArray {
    if person == "Novak" { break }
    positionOfNovak += 1
}

print("Novak is the element located on position [\(positionOfNovak)] in peopleArray.")
//prints out: Novak is the element located on position 5 in peopleArray. (which is true)
```

第17章：协议

协议是一种指定如何使用对象的方式。它们描述了类、结构体或枚举应提供的一组属性和方法，尽管协议对实现没有限制。

第17.1节：协议基础

关于协议

协议指定了符合该协议的 Swift 对象类型（类、结构体或枚举）所需的初始化器、属性、函数、下标和关联类型。在某些语言中，类似的需求规范被称为'接口'。

已声明和定义的协议本身就是一种类型，具有其声明需求的签名，有点类似于 Swift 函数基于其参数和返回值签名而成为一种类型的方式。

Swift 协议规范可以是可选的、明确要求的，和/或通过一种称为协议扩展的机制提供默认实现。一个希望遵循某个协议的 Swift 对象类型（类、结构体或枚举），只需声明其遵循该协议，即可通过扩展中为所有指定要求提供的实现，达到完全符合协议的要求。协议扩展的默认实现机制足以满足遵循协议的所有义务。

协议可以被其他协议继承。结合协议扩展，这意味着协议可以且应该被视为 Swift 的一个重要特性。

协议和扩展对于实现 Swift 更广泛的目标以及程序设计的灵活性和开发流程非常重要。Swift 协议和扩展功能的主要目的在于促进程序架构和开发中的组合式设计。这被称为协议导向编程。经验丰富的老程序员认为这优于面向对象编程设计。

协议定义了可以由任何结构体、类或枚举实现的接口：

```
protocol MyProtocol {
    init(value: Int)                        // 必需的初始化方法
    func doSomething() -> Bool              // 实例方法
    var message: String { get }             // 实例只读属性
    var value: Int { get set }              // 实例读写属性
    subscript(index: Int) -> Int { get }    // 实例下标
    static func instructions() -> String    // 静态方法
    static var max: Int { get }             // 静态只读属性
    static var total: Int { get set }       // 静态读写属性
}
```

协议中定义的属性必须标注为 { get } 或 { get set }。{ get } 表示该属性必须是可读的，因此可以实现为任何类型的属性。{ get set } 表示该属性必须既可读又可写。

结构体、类或枚举可以遵循一个协议：

```
struct MyStruct : MyProtocol {
    // 在此实现协议的要求
}
class MyClass : MyProtocol {
    // 在此实现协议的要求
}
```


Chapter 17: Protocols

Protocols are a way of specifying how to use an object. They describe a set of properties and methods which a class, structure, or enum should provide, although protocols pose no restrictions on the implementation.

Section 17.1: Protocol Basics

About Protocols

A Protocol specifies initialisers, properties, functions, subscripts and associated types required of a Swift object type (class, struct or enum) conforming to the protocol. In some languages similar ideas for requirement specifications of subsequent objects are known as 'interfaces'.

A declared and defined Protocol is a Type, in and of itself, with a signature of its stated requirements, somewhat similar to the manner in which Swift Functions are a Type based on their signature of parameters and returns.

Swift Protocol specifications can be optional, explicitly required and/or given default implementations via a facility known as Protocol Extensions. A Swift Object Type (class, struct or enum) desiring to conform to a Protocol that's fleshed out with Extensions for all its specified requirements needs only state its desire to conform to be in full conformance. The default implementations facility of Protocol Extensions can suffice to fulfil all obligations of conforming to a Protocol.

Protocols can be inherited by other Protocols. This, in conjunction with Protocol Extensions, means Protocols can and should be thought of as a significant feature of Swift.

Protocols and Extensions are important to realising Swift's broader objectives and approaches to program design flexibility and development processes. The primary stated purpose of Swift's Protocol and Extension capability is facilitation of compositional design in program architecture and development. This is referred to as Protocol Oriented Programming. Crusty old timers consider this superior to a focus on OOP design.

Protocols define interfaces which can be implemented by any struct, class, or enum:

```
protocol MyProtocol {
    init(value: Int)                        // required initializer
    func doSomething() -> Bool              // instance method
    var message: String { get }             // instance read-only property
    var value: Int { get set }              // read-write instance property
    subscript(index: Int) -> Int { get }    // instance subscript
    static func instructions() -> String    // static method
    static var max: Int { get }             // static read-only property
    static var total: Int { get set }       // read-write static property
}
```

Properties defined in protocols must either be annotated as `{ get }` or `{ get set }`. `{ get }` means that the property must be gettable, and therefore it can be implemented as *any* kind of property. `{ get set }` means that the property must be settable as well as gettable.

A struct, class, or enum may **conform to** a protocol:

```
struct MyStruct : MyProtocol {
    // Implement the protocol's requirements here
}
class MyClass : MyProtocol {
    // Implement the protocol's requirements here
}
```

```
enum MyEnum : MyProtocol {
    case caseA, caseB, caseC
    // 在这里实现协议的要求
}
```

协议还可以通过扩展为其任何要求定义一个默认实现：

```
extension MyProtocol {

    // doSomething() -> Bool 的默认实现
    // 如果遵循类型没有定义自己的实现，将使用此实现
    func doSomething() -> Bool {
        print("执行某事！")
        return true
    }
}
```

协议可以被用作类型，前提是它没有associatedtype要求：

```
func doStuff(object: MyProtocol) {
    // 对象上可用 MyProtocol 的所有要求
    print(object.message)
    print(object.doSomething())
}

let items : [MyProtocol] = [MyStruct(), MyClass(), MyEnum.caseA]
```

你也可以定义一个符合多个协议的抽象类型：

版本 ≥ 3.0

在 Swift 3 及更高版本中，这是通过用“&”符号分隔协议列表来实现的：

```
func doStuff(object: MyProtocol & AnotherProtocol) {
    // ...
}

let items : [MyProtocol & AnotherProtocol] = [MyStruct(), MyClass(), MyEnum.caseA]
```

版本 < 3.0

较旧版本的语法是protocol<...>，其中协议是用逗号分隔的列表，写在尖括号<>内。

```
protocol AnotherProtocol {
    func doSomethingElse()
}

func doStuff(object: protocol<MyProtocol, AnotherProtocol>) {

    // 对象上可以使用 MyProtocol 和 AnotherProtocol 的所有要求
    print(object.message)
object.doSomethingElse()
}

// MyStruct、MyClass 和 MyEnum 现在必须同时遵循 MyProtocol 和 AnotherProtocol
let items : [protocol<MyProtocol, AnotherProtocol>] = [MyStruct(), MyClass(), MyEnum.caseA]
```

现有类型可以通过扩展来遵循协议：


```
enum MyEnum : MyProtocol {
    case caseA, caseB, caseC
    // Implement the protocol's requirements here
}
```

A protocol may also define a **default implementation** for any of its requirements through an extension:

```
extension MyProtocol {

    // default implementation of doSomething() -> Bool
    // conforming types will use this implementation if they don't define their own
    func doSomething() -> Bool {
        print("do something!")
        return true
    }
}
```

A protocol can be **used as a type**, provided it doesn't have `associatedtype` requirements:

```
func doStuff(object: MyProtocol) {
    // All of MyProtocol's requirements are available on the object
    print(object.message)
    print(object.doSomething())
}

let items : [MyProtocol] = [MyStruct(), MyClass(), MyEnum.caseA]
```

You may also define an abstract type that conforms to **multiple** protocols:

Version ≥ 3.0

With Swift 3 or better, this is done by separating the list of protocols with an ampersand (`&`):

```
func doStuff(object: MyProtocol & AnotherProtocol) {
    // ...
}

let items : [MyProtocol & AnotherProtocol] = [MyStruct(), MyClass(), MyEnum.caseA]
```

Version < 3.0

Older versions have syntax `protocol<...>` where the protocols are a comma-separated list between the angle brackets `<>`.

```
protocol AnotherProtocol {
    func doSomethingElse()
}

func doStuff(object: protocol<MyProtocol, AnotherProtocol>) {

    // All of MyProtocol & AnotherProtocol's requirements are available on the object
    print(object.message)
    object.doSomethingElse()
}

// MyStruct, MyClass & MyEnum must now conform to both MyProtocol & AnotherProtocol
let items : [protocol<MyProtocol, AnotherProtocol>] = [MyStruct(), MyClass(), MyEnum.caseA]
```

Existing types can be **extended** to conform to a protocol:

```
extension String : MyProtocol {
    // 实现 String 尚未满足的任何要求
}
```

第17.2节：代理模式

代理（delegate）是 Cocoa 和 CocoaTouch 框架中常用的设计模式，其中一个类将实现某些功能的责任委托给另一个类。这遵循关注点分离的原则，框架类实现通用功能，而单独的代理实例实现具体的使用场景。

从对象通信的角度来看代理模式也是另一种理解方式。对象（Objects）经常需要相互通信，为此，一个对象需要遵循一个协议，才能成为另一个对象的代理。一旦完成此设置，另一个对象在发生有趣事件时会回调其代理。

例如，用户界面中的一个视图用于显示数据列表，应只负责数据如何显示的逻辑，而不负责决定显示哪些数据。

让我们深入一个更具体的例子。如果你有两个类，一个是父类，一个是子类：

```
class Parent { }
class Child { }
```

您想要通知父组件子组件的更改。

在 Swift 中，代理是通过protocol声明来实现的，因此我们将声明一个protocol，由代理（delegate）来实现。这里代理是parent对象。

```
protocol ChildDelegate: class {
    func childDidSomething()
}
```

子类需要声明一个属性来存储对代理的引用：

```
class Child {
    weak var delegate: ChildDelegate?
}
```

注意变量delegate是可选的，且协议ChildDelegate被标记为只能由类类型实现（没有这个，变量delegate无法声明为weak引用，从而避免循环引用。这意味着如果delegate变量在其他地方不再被引用，它将被释放）。这样父类只在需要且可用时注册代理。

此外，为了将代理标记为weak，我们必须通过在协议声明中添加class关键字，将ChildDelegate协议限制为引用类型。

在此示例中，当子类执行某些操作并需要通知其父类时，子类将调用：

```
delegate?.childDidSomething()
```

如果代理已被定义，代理将被通知子类已执行某些操作。

父类需要继承ChildDelegate协议才能响应其动作。这可以直接在父类中完成：


```
extension String : MyProtocol {
    // Implement any requirements which String doesn't already satisfy
}
```

Section 17.2: Delegate pattern

A *delegate* is a common design pattern used in Cocoa and CocoaTouch frameworks, where one class delegates responsibility for implementing some functionality to another. This follows a principle of separation of concerns, where the framework class implements generic functionality while a separate delegate instance implements the specific use case.

Another way to look into delegate pattern is in terms of object communication. `Objects` often needs to talks to each other and to do so an object need conform to a `protocol` in order to become a delegate of another Object. Once this setup has been done, the other object talks back to its delegates when interesting things happens.

For example, A view in userinterface to display a list of data should be responsible only for the logic of how data is displayed, not for deciding what data should be displayed.

Let's dive into a more concrete example. if you have two classes, a parent and a child:

```
class Parent { }
class Child { }
```

And you want to notify the parent of a change from the child.

In Swift, delegates are implemented using a `protocol` declaration and so we will declare a `protocol` which the `delegate` will implement. Here delegate is the `parent` object.

```
protocol ChildDelegate: class {
    func childDidSomething()
}
```

The child needs to declare a property to store the reference to the delegate:

```
class Child {
    weak var delegate: ChildDelegate?
}
```

Notice the variable `delegate` is an optional and the protocol `ChildDelegate` is marked to be only implemented by class type (without this the `delegate` variable can't be declared as a `weak` reference avoiding any retain cycle. This means that if the `delegate` variable is no longer referenced anywhere else, it will be released). This is so the parent class only registers the delegate when it is needed and available.

Also in order to mark our delegate as `weak` we must constrain our ChildDelegate protocol to reference types by adding `class` keyword in protocol declaration.

In this example, when the child does something and needs to notify its parent, the child will call:

```
delegate?.childDidSomething()
```

If the delegate has been defined, the delegate will be notified that the child has done something.

The parent class will need to extend the `ChildDelegate` protocol to be able to respond to its actions. This can be done directly on the parent class:

```
class Parent: ChildDelegate {
    ...

    func childDidSomething() {
        print("耶！")
    }
}
```

或者使用扩展：

```
extension Parent: ChildDelegate {
    func childDidSomething() {
        print("耶！")
    }
}
```

父类还需要告诉子类它是子类的代理：

```
// 在父类中
let child = Child()
child.delegate = self
```

默认情况下，Swiftprotocol不允许实现可选函数。只有当你的协议被标记为@objc属性并使用optional修饰符时，才可以指定可选函数。

例如，UITableView实现了iOS中表视图的通用行为，但用户必须实现两个代理类，分别是UITableViewDelegate和UITableViewDataSource，来定义具体单元格的外观和行为。

@objc public protocol UITableViewDelegate : NSObjectProtocol, UIScrollViewDelegate { // 显示自定义 optional public func tableView(tableView: UITableView, willDisplayCell cell: UITableViewCell, forRowAtIndexPath indexPath: NSIndexPath) optional public func tableView(tableView: UITableView, willDisplayHeaderView view: UIView，针对节 section: Int）可选的公共函数 tableView(tableView: UITableView, willDisplayFooterView view: UIView，针对节 section: Int）可选的公共函数 tableView(tableView: UITableView, didEndDisplayingCell cell: UITableViewCell，针对行索引路径 indexPath: NSIndexPath）... }

你可以通过更改类定义来实现此协议，例如：

```
class MyViewController : UIViewController, UITableViewDelegate
```

协议定义中未标记为`optional`的方法（本例中为`UITableViewDelegate`）必须实现。

第17.3节：关联类型要求

协议可以使用associatedtype关键字定义关联类型要求：

```
protocol Container {
```

associatedtype Element var count: Int { get } subscript(index: Int) -> Element { get set } }

带有关联类型要求的协议**只能用作泛型约束**：

// 这些是允许的，因为Container有关联类型要求： func displayValues(container: Container) { ... } class MyClass { let container: Container } // > 错误：协议 'Container' 只能用作


```
class Parent: ChildDelegate {
    ...

    func childDidSomething() {
        print("Yay!")
    }
}
```

Or using an extension:

```
extension Parent: ChildDelegate {
    func childDidSomething() {
        print("Yay!")
    }
}
```

The parent also needs to tell the child that it is the child's delegate:

```
// In the parent
let child = Child()
child.delegate = self
```

By default a Swift `protocol` does not allow an optional function be implemented. These can only be specified if your protocol is marked with the `@objc` attribute and the `optional` modifier.

For example `UITableView` implements the generic behavior of a table view in iOS, but the user must implement two delegate classes called `UITableViewDelegate` and `UITableViewDataSource` that implement how the specific cells look like and behave.

@objc public protocol UITableViewDelegate : NSObjectProtocol, UIScrollViewDelegate { // Display customization optional public func tableView(tableView: UITableView, willDisplayCell cell: UITableViewCell, forRowAtIndexPath indexPath: NSIndexPath) optional public func tableView(tableView: UITableView, willDisplayHeaderView view: UIView, forSection section: Int) optional public func tableView(tableView: UITableView, willDisplayFooterView view: UIView, forSection section: Int) optional public func tableView(tableView: UITableView, didEndDisplayingCell cell: UITableViewCell, forRowAtIndexPath indexPath: NSIndexPath) ... }

You can implement this protocol by changing your class definition, for example:

```
class MyViewController : UIViewController, UITableViewDelegate
```

Any methods not marked `optional` in the protocol definition (`UITableViewDelegate` in this case) must be implemented.

Section 17.3: Associated type requirements

Protocols may define **associated type requirements** using the `associatedtype` keyword:

```
protocol Container {
```

ciatedtype Element var count: Int { get } subscript(index: Int) -> Element { get set } }

Protocols with associated type requirements **can only be used as generic constraints**:

`// These are` allowed, because Container has associated type requirements: func displayValues(container: Container) { ... } class MyClass { let container: Container } // > error: protocol 'Container' can only be used as a

通用约束 // > 因为它有 Self 或关联类型要求 // 允许以下情况：func displayValues<T: Container>(container: T) { ... } class MyClass<T: Container> { let container: T }

符合协议的类型可以通过提供一个特定类型来隐式满足关联类型（associatedtype）要求，协议期望出现关联类型的位置：

```
struct ContainerOfOne<T>: Container {
    let count = 1           // 满足 count 要求
    var value: T

    // 通过将下标赋值/返回类型定义为 T，隐式满足下标关联类型要求

    // 因此 Swift 会推断 T == Element
    subscript(index: Int) ->
```

> { get { precondition(index == 0) return value } set { precondition(index == 0) value = newValue } } } let container = ContainerOfOne(value: "Hello")

(注意，为了使此示例更清晰，泛型占位类型命名为T——更合适的名称应为Element，这将覆盖协议中的associatedtype Element。编译器仍会推断泛型占位符Element用于满足associatedtype Element的要求。)

关联类型（associatedtype）也可以通过使用typealias显式满足：

```
struct ContainerOfOne<T>: Container {
```

别名 Element = T 下标(index: Int) -> **Element** { ... } // ... }

扩展也是如此：

```
// 将8位整数作为布尔值集合（每个位一个布尔值）暴露出来。
extension UInt8: Container {

    // 如上所述，此类型别名可以被推断
    typealias Element = Bool

    var count: Int { return 8 }
    subscript(index: Int) -> Bool {
        get {
            precondition(0 <= index && index < 8)
            return self & 1 << UInt8(index) != 0
        }
        set {
            precondition(0 <= index && index < 8)
            if newValue {
                self |= 1 << UInt8(index)
            } else {
                self &= ~(1 << UInt8(index))
            }
        }
    }
}
```

如果符合的类型已经满足要求，则无需实现：

```
extension Array: Container {}  // Array 满足所有要求，包括 Element
```


generic constraint // > because it has Self or associated type requirements // These are allowed: func displayValues**<T: Container>**(container: **T**) { ... } class MyClass**<T: Container>** { let container: **T** }

A type which conforms to the protocol may satisfy an `associatedtype` requirement implicitly, by providing a given type where the protocol expects the `associatedtype` to appear:

```
struct ContainerOfOne<T>: Container {
    let count = 1           // satisfy the count requirement
    var value: T

    // satisfy the subscript associatedtype requirement implicitly,
    // by defining the subscript assignment/return type as T
    // therefore Swift will infer that T == Element
    subscript(index: Int) ->
```

> { get { precondition(index == 0) return value } set { precondition(index == 0) value = newValue } } } let container = ContainerOfOne(value: "Hello")

(Note that to add clarity to this example, the generic placeholder type is named `T` – a more suitable name would be `Element`, which would shadow the protocol's `associatedtype Element`. The compiler will still infer that the generic placeholder `Element` is used to satisfy the `associatedtype Element` requirement.)

An `associatedtype` may also be satisfied explicitly through the use of a `typealias`:

```
struct ContainerOfOne<T>: Container {
```

alias Element = T subscript(index: Int) -> **Element** { ... } // ... }

The same goes for extensions:

```
// Expose an 8-bit integer as a collection of boolean values (one for each bit).
extension UInt8: Container {

    // as noted above, this typealias can be inferred
    typealias Element = Bool

    var count: Int { return 8 }
    subscript(index: Int) -> Bool {
        get {
            precondition(0 <= index && index < 8)
            return self & 1 << UInt8(index) != 0
        }
        set {
            precondition(0 <= index && index < 8)
            if newValue {
                self |= 1 << UInt8(index)
            } else {
                self &= ~(1 << UInt8(index))
            }
        }
    }
}
```

If the conforming type already satisfies the requirement, no implementation is needed:

```
extension Array: Container {}  // Array satisfies all requirements, including Element
```

第17.4节：仅限类的协议

协议可以通过在其继承列表中使用class关键字来指定只有类可以实现它。该关键字必须出现在继承列表中其他协议之前。

```
protocol ClassOnlyProtocol: s, SomeOtherProtocol { // 协议要求 }
```

如果非类类型尝试实现`ClassOnlyProtocol`，将会产生编译错误。

```
struct MyStruct: ClassOnlyProtocol {
    // 错误：非类类型'MyStruct'无法符合类协议'ClassOnlyProtocol'
}
```

其他协议可以继承ClassOnlyProtocol，但它们也将具有相同的仅限类的要求。

```
protocol MyProtocol: ClassOnlyProtocol {
    // ClassOnlyProtocol要求
    // MyProtocol要求
}

class MySecondClass: MyProtocol {
    // ClassOnlyProtocol要求
    // MyProtocol要求
}
```

仅限类协议的引用语义

使用仅限类的协议允许在符合类型未知时使用引用语义。

```
协议 Foo : 类 {
    变量 bar : 字符串 { 获取 设置 }
}

函数 takesAFoo(foo:Foo) {

    // 该赋值需要引用语义，
    // 因为 foo 在此作用域中是一个 let 常量。
foo.bar = "新值"
}
```

在此示例中，由于 `Foo` 是仅限类的协议，赋值给 `bar` 是有效的，因为编译器知道 `foo` 是一个类类型，因此具有引用语义。

如果 Foo 不是仅限类的协议，则会产生编译错误–因为符合协议的类型可能是一个值类型，这将需要使用 var 注解才能是可变的。

```
协议 Foo {
    变量 bar : 字符串 { 获取 设置 }
}

函数 takesAFoo(foo:Foo) {
foo.bar = "新值" // 错误：无法赋值属性：'foo' 是一个 'let' 常量
}
```

```
函数 takesAFoo(foo:Foo) {
```


Section 17.4: Class-Only Protocols

A protocol may specify that only a class can implement it through using the `class` keyword in its inheritance list. This keyword must appear before any other inherited protocols in this list.

```
protocol ClassOnlyProtocol: s, SomeOtherProtocol { // Protocol requirements }
```

If a non-class type tries to implement `ClassOnlyProtocol`, a compiler error will be generated.

```
struct MyStruct: ClassOnlyProtocol {
    // error: Non-class type 'MyStruct' cannot conform to class protocol 'ClassOnlyProtocol'
}
```

Other protocols may inherit from the `ClassOnlyProtocol`, but they will have the same class-only requirement.

```
protocol MyProtocol: ClassOnlyProtocol {
    // ClassOnlyProtocol Requirements
    // MyProtocol Requirements
}

class MySecondClass: MyProtocol {
    // ClassOnlyProtocol Requirements
    // MyProtocol Requirements
}
```

Reference semantics of class-only protocols

Using a class-only protocol allows for reference semantics when the conforming type is unknown.

```
protocol Foo : class {
    var bar : String { get set }
}

func takesAFoo(foo:Foo) {

    // this assignment requires reference semantics,
    // as foo is a let constant in this scope.
    foo.bar = "new value"
}
```

In this example, as `Foo` is a class-only protocol, the assignment to `bar` is valid as the compiler knows that `foo` is a class type, and therefore has reference semantics.

If `Foo` was not a class-only protocol, a compiler error would be yielded – as the conforming type could be a value type, which would require a `var` annotation in order to be mutable.

```
protocol Foo {
    var bar : String { get set }
}

func takesAFoo(foo:Foo) {
    foo.bar = "new value" // error: Cannot assign to property: 'foo' is a 'let' constant
}
```

```
func takesAFoo(foo:Foo) {
```

foo = foo // foo 的可变副本 foo.bar = "新值" // 无错误 - 同时满足引用和值语义 }

协议类型的弱变量

当对协议类型的变量应用weak修饰符时，该协议类型必须是仅限类的，因为weak只能应用于引用类型。

```
weak var weakReference : ClassOnlyProtocol?
```

第17.5节：针对特定符合类的协议扩展

你可以为特定类编写默认协议实现。

```
protocol MyProtocol {
    func doSomething()
}

extension MyProtocol where Self: UIViewController {
    func doSomething() {
        print("UIViewController 默认协议实现")
    }
}

class MyViewController: UIViewController, MyProtocol { }

let vc = MyViewController()
vc.doSomething() // 输出 "UIViewController 默认协议实现"
```

第17.6节：使用RawRepresentable协议（可扩展枚举）

```
// RawRepresentable 有一个关联类型 RawValue。
// 对于这个结构体，我们将通过实现类型为 String 的 rawValue 变量
// 让编译器推断类型
//
// 编译器推断 RawValue = String，无需 typealias
//
struct NotificationName: RawRepresentable {
    let rawValue: String

    static let dataFinished = NotificationNames(rawValue: "DataFinishedNotification")
}
```

这个结构体可以在其他地方扩展以添加更多案例

```
extension NotificationName {
    static let documentationLaunched = NotificationNames(rawValue:
"DocumentationLaunchedNotification")
}
```

接口可以围绕任何 RawRepresentable 类型或专门针对你的枚举结构体设计

```
func post(notification notification: NotificationName) -> Void {
    // 使用 notification.rawValue
}
```


foo = foo // mutable copy of foo foo.bar = "new value" // no error – satisfies both reference and value semantics }

Weak variables of protocol type

When applying the weak modifier to a variable of protocol type, that protocol type must be class-only, as `weak` can only be applied to reference types.

```
weak var weakReference : ClassOnlyProtocol?
```

Section 17.5: Protocol extension for a specific conforming class

You can write the **default protocol implementation** for a specific class.

```
protocol MyProtocol {
    func doSomething()
}

extension MyProtocol where Self: UIViewController {
    func doSomething() {
        print("UIViewController default protocol implementation")
    }
}

class MyViewController: UIViewController, MyProtocol { }

let vc = MyViewController()
vc.doSomething() // Prints "UIViewController default protocol implementation"
```

Section 17.6: Using the RawRepresentable protocol (Extensible Enum)

```
// RawRepresentable has an associatedType RawValue.
// For this struct, we will make the compiler infer the type
// by implementing the rawValue variable with a type of String
//
// Compiler infers RawValue = String without needing typealias
//
struct NotificationName: RawRepresentable {
    let rawValue: String

    static let dataFinished = NotificationNames(rawValue: "DataFinishedNotification")
}
```

This struct can be extended elsewhere to add cases

```
extension NotificationName {
    static let documentationLaunched = NotificationNames(rawValue:
"DocumentationLaunchedNotification")
}
```

And an interface can design around any RawRepresentable type or specifically your enum struct

```
func post(notification notification: NotificationName) -> Void {
    // use notification.rawValue
}
```

在调用处，您可以使用点语法简写来表示类型安全的 NotificationName

```
post(notification: .dataFinished)
```

使用通用的 RawRepresentable 函数

```
// RawRepresentable 有一个关联类型，
// 因此想要接受任何符合 RawRepresentable 类型的方法
// 需要是泛型的
func observe<T: RawRepresentable>(object: T) -> Void {
    // object.rawValue
}
```

第17.7节：实现 Hashable 协议

用于集合(Sets)和字典键(Dictionaries(key))的类型必须遵循Hashable协议，该协议继承自Equatable协议。

自定义类型遵循Hashable协议必须实现

- 一个计算属性 hashValue
- 定义一个等于运算符，即==或!=。

以下示例为自定义结构体实现了Hashable协议：

```
结构体 Cell {
    变量 行: 整数
    变量 列: 整数

    初始化(_ 行: 整数, _ 列: 整数) {
        self.行 = 行
        self.列 = 列
    }
}

扩展 Cell: 可哈希 {

    // 满足可哈希要求
    变量 哈希值: 整数 {
        获取 {
            返回 行.哈希值^列.哈希值
        }
    }

    // 满足可比较相等要求
    静态函数 ==(左: Cell, 右: Cell) -> 布尔 {
        返回 左.列 == 右.列 && 左.行 == 右.行
    }

}

// 现在我们可以将 Cell 作为字典的键
变量 字典 = [Cell : 字符串]()

字典[Cell(0, 0)] = "0, 0"
字典[Cell(1, 0)] = "1, 0"
字典[Cell(0, 1)] = "0, 1"

// 我们也可以创建 Cell 的集合
```


At call site, you can use dot syntax shorthand for the typesafe NotificationName

```
post(notification: .dataFinished)
```

Using generic RawRepresentable function

```
// RawRepresentable has an associate type, so the
// method that wants to accept any type conforming to
// RawRepresentable needs to be generic
func observe<T: RawRepresentable>(object: T) -> Void {
    // object.rawValue
}
```

Section 17.7: Implementing Hashable protocol

Types used in `Sets` and `Dictionaries(key)` must conform to `Hashable` protocol which inherits from `Equatable` protocol.

Custom type conforming to `Hashable` protocol must implement

- A calculated property `hashValue`
- Define one of the equality operators i.e. `==` or `!=`.

Following example implements `Hashable` protocol for a custom `struct`:

```
struct Cell {
    var row: Int
    var col: Int

    init(_ row: Int, _ col: Int) {
        self.row = row
        self.col = col
    }
}

extension Cell: Hashable {

    // Satisfy Hashable requirement
    var hashValue: Int {
        get {
            return row.hashValue^col.hashValue
        }
    }

    // Satisfy Equatable requirement
    static func ==(lhs: Cell, rhs: Cell) -> Bool {
        return lhs.col == rhs.col && lhs.row == rhs.row
    }

}

// Now we can make Cell as key of dictonary
var dict = [Cell : String]()

dict[Cell(0, 0)] = "0, 0"
dict[Cell(1, 0)] = "1, 0"
dict[Cell(0, 1)] = "0, 1"

// Also we can create Set of Cells
```

```
var set = Set<Cell>()

set.insert(Cell(0, 0))
set.insert(Cell(1, 0))
```

注意：自定义类型中不同的值不必拥有不同的哈希值，哈希冲突是可以接受的。如果哈希值相同，将使用等号运算符来判断实际是否相等。


```
var set = Set<Cell>()

set.insert(Cell(0, 0))
set.insert(Cell(1, 0))
```

Note: It is not necessary that different values in custom type have different hash values, collisions are acceptable. If hash values are equal, equality operator will be used to determine real equality.

第18章：函数

第18.1节：基本用法

函数可以声明为无参数或无返回值。唯一必需的信息是名称（本例中为hello）。

```
func hello()
{
    print("Hello World")
}
```

调用无参数函数时，写出函数名后跟一对空括号即可。

```
hello()
//输出: "Hello World"
```

第18.2节：带参数的函数

函数可以带参数，以便修改其功能。参数以逗号分隔的列表形式给出，定义了它们的类型和名称。

```
func magicNumber(number1: Int)
{
    print("\(number1) 是魔法数字")
}
```

注意： `\(number1)` 语法是基本的字符串插值，用于将整数插入字符串中。

带参数的函数通过指定函数名并提供与函数声明中使用的类型相同的输入值来调用。

```
magicNumber(5)
//输出: "5 是魔法数字
let example: Int = 10
magicNumber(example)
//输出: "10 是魔法数字"
```

任何 Int 类型的值都可以使用。

```
func magicNumber(number1: Int, number2: Int)
{
    print("\(number1 + number2) 是魔法数字")
}
```

当函数使用多个参数时，第一个参数的名称在调用时可以省略，但后续参数必须写出名称。

```
let ten: Int = 10
let five: Int = 5
magicNumber(ten,number2: five)
//输出: "15 是魔法数字"
```

使用外部参数名可以使函数调用更易读。

Chapter 18: Functions

Section 18.1: Basic Use

Functions can be declared without parameters or a return value. The only required information is a name (`hello` in this case).

```
func hello()
{
    print("Hello World")
}
```

Call a function with no parameters by writing its name followed by an empty pair of parenthesis.

```
hello()
//output: "Hello World"
```

Section 18.2: Functions with Parameters

Functions can take parameters so that their functionality can be modified. Parameters are given as a comma separated list with their types and names defined.

```
func magicNumber(number1: Int)
{
    print("\(number1) Is the magic number")
}
```

Note: The `\(number1)` syntax is basic String Interpolation and is used to insert the integer into the String.

Functions with parameters are called by specifying the function by name and supplying an input value of the type used in the function declaration.

```
magicNumber(5)
//output: "5 Is the magic number
let example: Int = 10
magicNumber(example)
//output: "10 Is the magic number"
```

Any value of type Int could have been used.

```
func magicNumber(number1: Int, number2: Int)
{
    print("\(number1 + number2) Is the magic number")
}
```

When a function uses multiple parameters the name of the first parameter is not required for the first but is on subsequent parameters.

```
let ten: Int = 10
let five: Int = 5
magicNumber(ten,number2: five)
//output: "15 Is the magic number"
```

Use external parameter names to make function calls more readable.

```
func magicNumber(one number1: Int, two number2: Int)
{
    print("\(number1 + number2) 是魔法数字")
}

let ten: Int = 10
let five: Int = 5
magicNumber(one: ten, two: five)
```

在函数声明中设置默认值可以让你在调用函数时不必传入任何参数。

```
func magicNumber(one number1: Int = 5, two number2: Int = 10)
{
    print("\(number1 + number2) 是魔法数字")
}

magicNumber()
//output: "15 Is the magic number"
```

第18.3节：下标

类、结构体和枚举可以定义下标，下标是访问集合、列表或序列成员元素的快捷方式。

示例

```
结构体 星期几 {

  变量 days = ["周日", "周一", "周二", "周三", "周四", "周五", "周六"]

  下标(索引: 整数) -> 字符串 {
    获取 {
      返回 days[索引]
    }
    set {
days[索引] = 新值
    }
  }
}
```

下标用法

```
变量 week = 星期几()
//你可以通过 array[index] 访问数组中索引为 index 的元素。
debugPrint(week[1])
debugPrint(week[0])
week[0] = "Sunday"
debugPrint(week[0])
```

下标选项：

下标可以接受任意数量的输入参数，这些输入参数可以是任意类型。下标
也可以返回任意类型。下标可以使用可变参数和变长参数，但不能使用输入输出
参数或提供默认参数值。

示例：


```
func magicNumber(one number1: Int, two number2: Int)
{
    print("\(number1 + number2) Is the magic number")
}

let ten: Int = 10
let five: Int = 5
magicNumber(one: ten, two: five)
```

Setting the default value in the function declaration allows you to call the function without giving any input values.

```
func magicNumber(one number1: Int = 5, two number2: Int = 10)
{
    print("\(number1 + number2) Is the magic number")
}

magicNumber()
//output: "15 Is the magic number"
```

Section 18.3: Subscripts

Classes, structures, and enumerations can define subscripts, which are shortcuts for accessing the member elements of a collection, list, or sequence.

Example

```
struct DaysOfWeek {

  var days = ["Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"]

  subscript(index: Int) -> String {
    get {
      return days[index]
    }
    set {
      days[index] = newValue
    }
  }
}
```

Subscript Usage

```
var week = DaysOfWeek()
//you access an element of an array at index by array[index].
debugPrint(week[1])
debugPrint(week[0])
week[0] = "Sunday"
debugPrint(week[0])
```

Subscripts Options:

Subscripts can take any number of input parameters, and these input parameters can be of any type. Subscripts can also return any type. Subscripts can use variable parameters and variadic parameters, but cannot use in-out parameters or provide default parameter values.

Example:

```
结构体 Food {

  枚举 MealTime {
    case 早餐, 午餐, 晚餐
  }

  变量 meals: [MealTime: String] = [:]

  下标 (type: MealTime) -> String? {
    获取 {
      返回 meals[type]
    }
    set {
meals[type] = newValue
    }
  }
}
```

用法

```
var diet = Food()
diet[.早餐] = "炒鸡蛋"
diet[.午餐] = "米饭"

debugPrint("我早餐吃了 \(diet[.早餐])")
```

第18.4节：方法

实例方法是属于Swift中某个类型（类、结构体、枚举或协议）实例的函数。类型方法则是在类型本身上调用的。

实例方法

实例方法通过在类型定义内部或扩展中使用func声明来定义。

```
class 计数器 {
    var 计数 = 0
    func 增加() {
        计数 += 1
    }
}
```

increment()实例方法是在计数器类的实例上调用的：

```
let 计数器 = 计数器()  // 创建计数器类的实例
计数器.增加()       // 在该实例上调用实例方法
```

类型方法

类型方法使用static func关键字定义。（对于类，class func定义的类型方法可以被子类重写。）

```
class SomeClass {
    class func someTypeMethod() {
        // 类型方法的实现写在这里
    }
}
```



```
struct Food {

  enum MealTime {
    case Breakfast, Lunch, Dinner
  }

  var meals: [MealTime: String] = [:]

  subscript (type: MealTime) -> String? {
    get {
      return meals[type]
    }
    set {
      meals[type] = newValue
    }
  }
}
```

Usage

```
var diet = Food()
diet[.Breakfast] = "Scrambled Eggs"
diet[.Lunch] = "Rice"

debugPrint("I had \(diet[.Breakfast]) for breakfast")
```

Section 18.4: Methods

Instance methods are functions that belong to instances of a type in Swift (a class, struct, enumeration, or protocol). **Type methods** are called on a type itself.

Instance Methods

Instance methods are defined with a `func` declaration inside the definition of the type, or in an extension.

```
class Counter {
    var count = 0
    func increment() {
        count += 1
    }
}
```

The `increment()` instance method is called on an instance of the `Counter` class:

```
let counter = Counter()  // create an instance of Counter class
counter.increment()      // call the instance method on this instance
```

Type Methods

Type methods are defined with the `static func` keywords. (For classes, `class func` defines a type method that can be overridden by subclasses.)

```
class SomeClass {
    class func someTypeMethod() {
        // type method implementation goes here
    }
}
```

```
SomeClass.someTypeMethod()  // 类型方法直接在SomeClass类型上调用
```

第18.5节：可变参数

有时，无法列出函数可能需要的参数数量。考虑一个sum函数：

```
func sum(_ a: Int, _ b: Int) -> Int {
    return a + b
}
```

这个函数可以很好地计算两个数字的和，但如果要计算三个数字的和，我们就得写另一个函数：

```
func sum(_ a: Int, _ b: Int, _ c: Int) -> Int {
    return a + b + c
}
```

如果一个函数有四个参数，则需要另一个，以此类推。Swift 使得定义一个带有可变数量参数的函数成为可能，方法是使用三个点的序列：...。例如，

```
func sum(_ numbers: Int...) -> Int {
    return numbers.reduce(0, combine: +)
}
```

注意可变参数 `numbers` 被合并成了一个类型为[Int]的单一Array。这在一般情况下都是如此，类型为T...的可变参数可以作为[T]访问。

现在可以这样调用这个函数：

```
let a = sum(1, 2) // a == 3
let b = sum(3, 4, 5, 6, 7) // b == 25
```

Swift中的可变参数不必出现在参数列表的末尾，但每个函数签名中只能有一个可变参数。

有时，给参数数量设置最小值会很方便。例如，计算 sum 时没有任何值是没有意义的。一个简单的强制方法是先设置一些非可变参数，然后再添加可变参数。为了确保 sum 至少需要两个参数调用，我们可以这样写

```
func sum(_ n1: Int, _ n2: Int, _ numbers: Int...) -> Int {
    return numbers.reduce(n1 + n2, combine: +)
}

sum(1, 2) // 合法
sum(3, 4, 5, 6, 7) // 合法
sum(1) // 不合法
sum() // 不合法
```

第18.6节：运算符是函数

运算符如+、-、??是一种使用符号而非字母命名的函数。它们的调用方式与普通函数不同：

- 前缀：>x


```
SomeClass.someTypeMethod()  // type method is called on the SomeClass type itself
```

Section 18.5: Variadic Parameters

Sometimes, it's not possible to list the number of parameters a function could need. Consider a `sum` function:

```
func sum(_ a: Int, _ b: Int) -> Int {
    return a + b
}
```

This works fine for finding the sum of two numbers, but for finding the sum of three we'd have to write another function:

```
func sum(_ a: Int, _ b: Int, _ c: Int) -> Int {
    return a + b + c
}
```

and one with four parameters would need another one, and so on. Swift makes it possible to define a function with a variable number of parameters using a sequence of three periods: `...`. For example,

```
func sum(_ numbers: Int...) -> Int {
    return numbers.reduce(0, combine: +)
}
```

Notice how the `numbers` parameter, which is variadic, is coalesced into a single `Array` of type `[Int]`. This is true in general, variadic parameters of type `T...` are accessible as a `[T]`.

This function can now be called like so:

```
let a = sum(1, 2) // a == 3
let b = sum(3, 4, 5, 6, 7) // b == 25
```

A variadic parameter in Swift doesn't have to come at the end of the parameter list, but there can only be one in each function signature.

Sometimes, it's convenient to put a minimum size on the number of parameters. For example, it doesn't really make sense to take the `sum` of no values. An easy way to enforce this is by putting some non-variadic required parameters and then adding the variadic parameter after. To make sure that `sum` can only be called with at least two parameters, we can write

```
func sum(_ n1: Int, _ n2: Int, _ numbers: Int...) -> Int {
    return numbers.reduce(n1 + n2, combine: +)
}

sum(1, 2) // ok
sum(3, 4, 5, 6, 7) // ok
sum(1) // not ok
sum() // not ok
```

Section 18.6: Operators are Functions

Operators such as `+`, `-`, `??` are a kind of function named using symbols rather than letters. They are invoked differently from functions:

- Prefix: >x

- 中缀：`x > y`
- 后缀：xb>

你可以在《Swift 编程语言》中阅读更多关于基本运算符和高级运算符的内容。

第18.7节：传递和返回函数

下面的函数返回另一个函数作为结果，该结果可以稍后赋值给变量并调用：

```
func jediTrainer () -> ((String, Int) -> String) {
   func train(name: String, times: Int) -> (String) {
      return "\(name) 已经被训练了 \(times) 次原力"
   }
   return train
}

let train = jediTrainer()
train("欧比旺", 3)
```

第18.8节：函数类型

每个函数都有其自身的函数类型，由参数类型和函数本身的返回类型组成。

例如，以下函数：

```
func sum(x: Int, y: Int) -> (result: Int) { return x + y }
```

具有以下函数类型：

```
(Int, Int) -> (Int)
```

因此，函数类型可以用作参数类型或作为嵌套函数的返回类型。

第18.9节：Inout参数

如果参数被标记为inout关键字，函数可以修改传入的参数。传递inout参数给函数时，调用者必须在传入的变量前加上&符号。

```
func updateFruit(fruit: inout Int) {
    fruit -= 1
}

var apples = 30 // 输出“有30个苹果”
print("There's \(apples) apples")

updateFruit(fruit: &apples)

print("现在有 \(apples) 个苹果") // 打印 "现在有 29 个苹果".
```

这允许将引用语义应用于通常具有值语义的类型。

第18.10节：抛出错误

如果你希望函数能够抛出错误，需要在包含参数的括号后添加throws关键字：

- Infix: `x > y`
- Postfix: xb>

You can read more about basic operators and advanced operators in The Swift Programming Language.

Section 18.7: Passing and returning functions

The following function is returning another function as its result which can be later assigned to a variable and called:

```
func jediTrainer () -> ((String, Int) -> String) {
  func train(name: String, times: Int) -> (String) {
    return "\(name) has been trained in the Force \(times) times"
  }
  return train
}

let train = jediTrainer()
train("Obi Wan", 3)
```

Section 18.8: Function types

Every function has its own function type, made up of the parameter types and the return type of the function itself. For example the following function:

```
func sum(x: Int, y: Int) -> (result: Int) { return x + y }
```

has a function type of:

```
(Int, Int) -> (Int)
```

Function types can thus be used as parameters types or as return types for nesting functions.

Section 18.9: Inout Parameters

Functions can modify the parameters passed to them if they are marked with the `inout` keyword. When passing an `inout` parameter to a function, the caller must add a & to the variable being passed.

```
func updateFruit(fruit: inout Int) {
    fruit -= 1
}

var apples = 30 // Prints "There's 30 apples"
print("There's \(apples) apples")

updateFruit(fruit: &apples)

print("There's now \(apples) apples") // Prints "There's 29 apples".
```

This allows reference semantics to be applied to types which would normally have value semantics.

Section 18.10: Throwing Errors

If you want a function to be able to throw errors, you need to add the `throws` keyword after the parentheses that hold the arguments:

```
func errorThrower()throws -> String {}
```

当你想抛出错误时，使用throw关键字：

```
func errorThrower()throws -> String {
  if true {
    return "True"
  } else {
    // 抛出错误
   throw Error.error
  }
}
```

如果你想调用一个可能抛出错误的函数，需要在do代码块中使用try关键字：

```
do {
try errorThrower()
}
```

有关 Swift 错误的更多内容：Errors

第18.11节：返回值

函数可以通过在参数列表后指定类型来返回值。

```
func 找斜边(a: Double, b: Double) -> Double
{
    return sqrt((a * a) + (b * b))
}

let c = 找斜边(3, b: 5)
//c = 5.830951894845301
```

函数也可以使用元组返回多个值。

```
func 数学(number: Int) -> (times2: Int, times3: Int)
{
    let two = number * 2
    let three = number * 3
    return (two, three)
}
let resultTuple = 数学(5)
//resultTuple = (10, 15)
```

第18.12节：尾随闭包语法

当函数的最后一个参数是闭包时

```
func loadData(id: String, completion:(result: String) -> ()) {
    // ...
completion(result:"这是结果数据")
}
```

函数可以使用尾随闭包语法调用

```
loadData("123") { result in
    print(result)
```



```
func errorThrower()throws -> String {}
```

When you want to throw an error, use the `throw` keyword:

```
func errorThrower()throws -> String {
  if true {
    return "True"
  } else {
    // Throwing an error
    throw Error.error
  }
}
```

If you want to call a function that can throw an error, you need to use the `try` keyword in a `do` block:

```
do {
  try errorThrower()
}
```

For more on Swift errors: Errors

Section 18.11: Returning Values

Functions can return values by specifying the type after the list of parameters.

```
func findHypotenuse(a: Double, b: Double) -> Double
{
    return sqrt((a * a) + (b * b))
}

let c = findHypotenuse(3, b: 5)
//c = 5.830951894845301
```

Functions can also return multiple values using tuples.

```
func maths(number: Int) -> (times2: Int, times3: Int)
{
    let two = number * 2
    let three = number * 3
    return (two, three)
}
let resultTuple = maths(5)
//resultTuple = (10, 15)
```

Section 18.12: Trailing Closure Syntax

When the last parameter of a function is a closure

```
func loadData(id: String, completion:(result: String) -> ()) {
    // ...
    completion(result:"This is the result data")
}
```

the function can be invoked using the Trailing Closure Syntax

```
loadData("123") { result in
    print(result)
```

```
}
```

第18.13节：带闭包的函数

使用接受并执行闭包的函数对于传递一段代码以在其他地方执行非常有用。我们可以先让函数接受一个可选闭包（在本例中）返回Void。

```
func closedFunc(block: (()->Void)? = nil) {
    print("刚开始")

    if let block = block {
       block()
    }
}
```

现在我们的函数已经定义好了，让我们调用它并传入一些代码：

```
closedFunc() { Void in
    print("Over already")
}
```

通过在函数调用中使用尾随闭包，我们可以传入代码（在本例中为print），以便在closedFunc()函数的某个时刻执行。

日志应打印：

> 刚开始
>
> 已经结束

一个更具体的用例可能包括在两个类之间执行代码：

```
class ViewController: UIViewController {

    override func viewDidLoad() {
        let _  = A.init(){Void in self.action(2)}
    }

    func action(i: Int) {
        print(i)
    }
}


class A: NSObject {
    var closure : ()?

    init(closure: (()->Void)? = nil) {
        // 注意这是在闭包之前执行的
        print("1")
        // 确保闭包不是 nil
        self.closure = closure?()
    }
}
```



```
}
```

Section 18.13: Functions With Closures

Using functions that take in and execute closures can be extremely useful for sending a block of code to be executed elsewhere. We can start by allowing our function to take in an optional closure that will (in this case) return `Void`.

```
func closedFunc(block: (()->Void)? = nil) {
    print("Just beginning")

    if let block = block {
        block()
    }
}
```

Now that our function has been defined, let's call it and pass in some code:

```
closedFunc() { Void in
    print("Over already")
}
```

By using a **trailing closure** with our function call, we can pass in code (in this case, `print`) to be executed at some point within our `closedFunc()` function.

The log should print:

> Just beginning
>
> Over already

A more specific use case of this could include the execution of code between two classes:

```
class ViewController: UIViewController {

    override func viewDidLoad() {
        let _  = A.init(){Void in self.action(2)}
    }

    func action(i: Int) {
        print(i)
    }
}


class A: NSObject {
    var closure : ()?

    init(closure: (()->Void)? = nil) {
        // Notice how this is executed before the  closure
        print("1")
        // Make sure closure isn't nil
        self.closure = closure?()
    }
}
```

日志应打印：

> 1
>
> 2

The log should print:

> 1
>
> 2

第19章：扩展

第19.1节：什么是扩展？

扩展用于扩展 Swift 中现有类型的功能。扩展可以添加下标、函数、初始化器和计算属性。它们还可以使类型遵循协议。

假设你想计算一个Int的阶乘。你可以在扩展中添加一个计算属性：

```
extension Int {
    var factorial: Int {
        return (1..<self+1).reduce(1, combine: *)
    }
}
```

然后您可以像访问原始 Int API 中包含的属性一样访问该属性。

```
let val1: Int = 10

val1.factorial  // 返回 3628800
```

第19.2节：变量和函数

扩展可以包含函数和计算型/常量型的获取变量。它们的格式是

```
extension ExtensionOf {
    // 新的函数和获取变量
}
```

要引用被扩展对象的实例，可以使用self，就像之前使用的一样

例如，创建一个对String的扩展，添加一个.length()函数，返回字符串的长度

```
extension String {
    func length() -> Int {
        return self.characters.count
    }
}

"Hello, World!".length() // 13
```

扩展还可以包含get变量。例如，给字符串添加一个.length变量，返回字符串的长度

```
extension String {
    var length: Int {
        get {
            return self.characters.count
        }
    }
}

"Hello, World!".length // 13
```


Chapter 19: Extensions

Section 19.1: What are Extensions?

Extensions are used to extend the functionality of existing types in Swift. Extensions can add subscripts, functions, initializers, and computed properties. They can also make types conform to protocols.

Suppose you want to be able to compute the factorial of an `Int`. You can add a computed property in an extension:

```
extension Int {
    var factorial: Int {
        return (1..<self+1).reduce(1, combine: *)
    }
}
```

Then you can access the property just as if it had been included in original Int API.

```
let val1: Int = 10

val1.factorial  // returns 3628800
```

Section 19.2: Variables and functions

Extensions can contain functions and computed/constant get variables. They are in the format

```
extension ExtensionOf {
    //new functions and get-variables
}
```

To reference the instance of the extended object, `self` can be used, just as it could be used

To create an extension of `String` that adds a `.length()` function which returns the length of the string, for example

```
extension String {
    func length() -> Int {
        return self.characters.count
    }
}

"Hello, World!".length() // 13
```

Extensions can also contain `get` variables. For example, adding a `.length` variable to the string which returns the length of the string

```
extension String {
    var length: Int {
        get {
            return self.characters.count
        }
    }
}

"Hello, World!".length // 13
```

第19.3节：扩展中的初始化器

扩展可以包含便利初始化器。例如，一个接受NSString的可失败初始化器，用于Int：

```
extension Int {
    init?(_ string: NSString) {
        self.init(string as String)  // 委托给现有的 Int.init(String) 初始化器
    }
}

let str1: NSString = "42"
Int(str1) // 42

let str2: NSString = "abc"
Int(str2) // nil
```

第19.4节：下标

扩展可以为现有类型添加新的下标。

此示例使用给定的索引获取字符串中的字符：

版本 = 2.2

```
extension String {
    下标(index: Int) -> Character {
        let newIndex = startIndex.advancedBy(index)
        return self[newIndex]
    }
}

var myString = "StackOverFlow"
print(myString[2]) // a
print(myString[3]) // c
```

版本 = 3.0

```
extension String {
    下标(offset: Int) -> Character {
        let newIndex = self.index(self.startIndex, offsetBy: offset)
        return self[newIndex]
    }
}

var myString = "StackOverFlow"
print(myString[2]) // a
print(myString[3]) // c
```

第19.5节：协议扩展

Swift 2.2的一个非常有用的功能是能够扩展协议。

它的工作方式非常类似于抽象类，针对你希望在所有实现某个协议的类中可用的功能（而无需继承自一个公共基类）。

```
protocol FooProtocol {
    func doSomething()
}

extension FooProtocol {
    func doSomething() {
```


Section 19.3: Initializers in Extensions

Extensions can contain convenience initializers. For example, a failable initializer for `Int` that accepts a `NSString`:

```
extension Int {
    init?(_ string: NSString) {
        self.init(string as String)  // delegate to the existing Int.init(String) initializer
    }
}

let str1: NSString = "42"
Int(str1) // 42

let str2: NSString = "abc"
Int(str2) // nil
```

Section 19.4: Subscripts

Extensions can add new subscripts to an existing type.

This example gets the character inside a String using the given index:

Version = 2.2

```
extension String {
    subscript(index: Int) -> Character {
        let newIndex = startIndex.advancedBy(index)
        return self[newIndex]
    }
}

var myString = "StackOverFlow"
print(myString[2]) // a
print(myString[3]) // c
```

Version = 3.0

```
extension String {
    subscript(offset: Int) -> Character {
        let newIndex = self.index(self.startIndex, offsetBy: offset)
        return self[newIndex]
    }
}

var myString = "StackOverFlow"
print(myString[2]) // a
print(myString[3]) // c
```

Section 19.5: Protocol extensions

A very useful feature of Swift 2.2 is having the ability of extending protocols.

It works pretty much like abstract classes when regarding a functionality you want to be available in all the classes that implements some protocol (without having to inherit from a base common class).

```
protocol FooProtocol {
    func doSomething()
}

extension FooProtocol {
    func doSomething() {
```

```
            print("Hi")
        }
}

class Foo: FooProtocol {
        func myMethod() {
doSomething() // 仅通过实现协议，此方法即可使用
        }
}
```

这也可以通过泛型实现。

第19.6节：限制

可以使用where语句在泛型类型上编写更具限制性的方法。

```
extension Array where Element: StringLiteralConvertible {
  func toUpperCase() -> [String] {
        var result = [String]()
        for value in self {
result.append(String(value).uppercaseString)
        }
        return result
    }
}
```

使用示例

```
let array = ["a","b","c"]
let resultado = array.toUpperCase()
```

第19.7节：什么是扩展以及何时使用它们

扩展为现有的类、结构体、枚举或协议类型添加新功能。这包括
扩展那些你无法访问其原始源代码的类型的能力。

Swift中的扩展可以：

- 添加计算属性和计算类型属性
- 定义实例方法和类型方法
- 提供新的初始化方法
- 定义下标
- 定义和使用新的嵌套类型
- 使现有类型遵循某个协议

何时使用Swift扩展：

- Swift的附加功能
- UIKit / Foundation的附加功能
- 在不影响他人代码的情况下添加附加功能
- 将类拆分为：数据 / 功能 / 代理

何时不使用：

- 从另一个文件扩展你自己的类


```
        print("Hi")
    }
}

class Foo: FooProtocol {
    func myMethod() {
        doSomething() // By just implementing the protocol this method is available
    }
}
```

This is also possible using generics.

Section 19.6: Restrictions

It is possible to write a method on a generic type that is more restrictive using where sentence.

```
extension Array where Element: StringLiteralConvertible {
  func toUpperCase() -> [String] {
      var result = [String]()
      for value in self {
          result.append(String(value).uppercaseString)
      }
      return result
    }
}
```

Example of use

```
let array = ["a","b","c"]
let resultado = array.toUpperCase()
```

Section 19.7: What are extensions and when to use them

Extensions add new functionality to an existing class, structure, enumeration, or protocol type. This includes the ability to extend types for which you do not have access to the original source code.

Extensions in Swift can:

- Add computed properties and computed type properties
- Define instance methods and type methods
- Provide new initializers
- Define subscripts
- Define and use new nested types
- Make an existing type conform to a protocol

When to use Swift Extensions:

- Additional functionality to Swift
- Additional functionality to UIKit / Foundation
- Additional functionality without messing with other persons code
- Breakdown classes into: Data / Functionality / Delegate

When not to use:

- Extend your own classes from another file

简单示例：

```
extension Bool {
    public mutating func toggle() -> Bool {
        self = !self
        return self
    }
}

var myBool: Bool = true
print(myBool.toggle()) // false
```

来源

Simple example:

```
extension Bool {
    public mutating func toggle() -> Bool {
        self = !self
        return self
    }
}

var myBool: Bool = true
print(myBool.toggle()) // false
```

Source

第20章：类

第20.1节：定义类

你可以这样定义一个类：

```
class Dog {}
```

一个类也可以是另一个类的子类：

```
class Animal {}
class Dog: Animal {}
```

在这个例子中，Animal 也可以是一个协议，Dog 遵循该协议。

第20.2节：属性和方法

类可以定义实例可以使用的属性。在这个例子中，`Dog` 有两个属性：`name` 和 `dogYearAge`：

```
class Dog {
    var name = ""
    var dogYearAge = 0
}
```

你可以使用点语法访问属性：

```
let dog = Dog()
print(dog.name)
print(dog.dogYearAge)
```

类也可以定义可以在实例上调用的方法，它们的声明方式类似于普通函数，
只是写在类的内部：

```
class Dog {
    func bark() {
        print("Ruff!")
    }
}
```

调用方法也使用点语法：

```
dog.bark()
```

第20.3节：引用语义

类是引用类型，意味着多个变量可以引用同一个实例。

```
class Dog {
    var name = ""
}

let firstDog = Dog()
firstDog.name = "Fido"
```


Chapter 20: Classes

Section 20.1: Defining a Class

You define a class like this:

```
class Dog {}
```

A class can also be a subclass of another class:

```
class Animal {}
class Dog: Animal {}
```

In this example, `Animal` could also be a protocol that `Dog` conforms to.

Section 20.2: Properties and Methods

Classes can define properties that instances of the class can use. In this example, `Dog` has two properties: `name` and `dogYearAge`:

```
class Dog {
    var name = ""
    var dogYearAge = 0
}
```

You can access the properties with dot syntax:

```
let dog = Dog()
print(dog.name)
print(dog.dogYearAge)
```

Classes can also define methods that can be called on the instances, they are declared similar to normal functions, just inside the class:

```
class Dog {
    func bark() {
        print("Ruff!")
    }
}
```

Calling methods also uses dot syntax:

```
dog.bark()
```

Section 20.3: Reference Semantics

Classes are **reference types**, meaning that multiple variables can refer to the same instance.

```
class Dog {
    var name = ""
}

let firstDog = Dog()
firstDog.name = "Fido"
```

```
let otherDog = firstDog   // otherDog 指向同一个 Dog 实例
```

same Dog instance otherDog.name = "Rover" // 修改 otherDog **也会修改 firstDog**print(firstDog.name) // 打印 "Rover"

因为类是引用类型，即使类是常量，其变量属性仍然可以被修改。

```
class Dog {
    var name: String // name 是一个变量属性。
    let age: Int // age 是一个常量属性。
    init(name: String, age: Int) {
        self.name = name
        self.age = age
    }
}

let constantDog = Dog(name: "Rover", age: 5)// 该实例是一个常量。
var variableDog = Dog(name: "Spot", age 7)// 该实例是一个变量。

constantDog.name = "Fido" // 不是错误，因为 name 是变量属性。
constantDog.age = 6 // 错误，因为 age 是一个常量属性。
constantDog = Dog(name: "Fido", age: 6)
/* 最后一条是错误的，因为你改变的是实际引用，而不仅仅是引用指向的内容。 */


variableDog.name = "Ace" // 不是错误，因为 name 是一个变量属性。
variableDog.age = 8 // 错误，因为 age 是一个常量属性。
variableDog = Dog(name: "Ace", age: 8)
/* 最后一条不是错误，因为 variableDog 是一个变量实例，因此实际引用可以被改变。 */
```

使用 === 测试两个对象是否 identical（指向完全相同的实例）：

```
class Dog: Equatable {
    let name: String
    init(name: String) { self.name = name }
}

// 如果两只狗的名字相同，则认为它们相等。
func ==(lhs: Dog, rhs: Dog) -> Bool {
    return lhs.name == rhs.name
}

// 创建两个名字相同的 Dog 实例。
let spot1 = Dog(name: "Spot")
let spot2 = Dog(name: "Spot")

spot1 == spot2    // true，因为这两只狗是相等的
spot1 != spot2    // false

spot1 === spot2   // false，因为这两只狗是不同的实例
spot1 !== spot2   // true
```

第20.4节：类和多重继承

Swift 不支持多重继承。也就是说，你不能继承自多个类。

```
class Animal { ... }
class Pet { ... }
```



```
let otherDog = firstDog  // otherDog points to
```

same Dog instance otherDog.name = "Rover" // modifying otherDog **also modifies firstDog** print(firstDog.name) // prints "Rover"

Because classes are reference types, even if the class is a constant, its variable properties can still be modified.

```
class Dog {
    var name: String // name is a variable property.
    let age: Int // age is a constant property.
    init(name: String, age: Int) {
        self.name = name
        self.age = age
    }
}

let constantDog = Dog(name: "Rover", age: 5)// This instance is a constant.
var variableDog = Dog(name: "Spot", age 7)// This instance is a variable.

constantDog.name = "Fido" // Not an error because name is a variable property.
constantDog.age = 6 // Error because age is a constant property.
constantDog = Dog(name: "Fido", age: 6)
/* The last one is an error because you are changing the actual reference, not
just what the reference points to. */

variableDog.name = "Ace" // Not an error because name is a variable property.
variableDog.age = 8 // Error because age is a constant property.
variableDog = Dog(name: "Ace", age: 8)
/* The last one is not an error because variableDog is a variable instance and
therefore the actual reference can be changed. */
```

Test whether two objects are *identical* (point to the exact same instance) using ===:

```
class Dog: Equatable {
    let name: String
    init(name: String) { self.name = name }
}

// Consider two dogs equal if their names are equal.
func ==(lhs: Dog, rhs: Dog) -> Bool {
    return lhs.name == rhs.name
}

// Create two Dog instances which have the same name.
let spot1 = Dog(name: "Spot")
let spot2 = Dog(name: "Spot")

spot1 == spot2   // true, because the dogs are equal
spot1 != spot2   // false

spot1 === spot2  // false, because the dogs are different instances
spot1 !== spot2  // true
```

Section 20.4: Classes and Multiple Inheritance

Swift does not support multiple inheritance. That is, you cannot inherit from more than one class.

```
class Animal { ... }
class Pet { ... }
```

```
class Dog: Animal, Pet { ... } // 这将导致编译错误。
```

相反，建议在创建类型时使用组合。这可以通过使用
协议来实现。

第20.5节：deinit

```
class ClassA {

    var timer: NSTimer!

    init() {
        // 初始化计时器
    }

    deinit {
        // 代码
timer.invalidate()
    }
}
```



```
class Dog: Animal, Pet { ... } // This will result in a compiler error.
```

Instead you are encouraged to use composition when creating your types. This can be accomplished by using protocols.

Section 20.5: deinit

```
class ClassA {

    var timer: NSTimer!

    init() {
        // initialize timer
    }

    deinit {
        // code
        timer.invalidate()
    }
}
```

第21章：类型转换

第21.1节：向下转换

变量可以使用类型转换操作符as?和as!向子类型进行向下转换。

as 操作符尝试转换为子类型。
它可能会失败，因此返回一个可选值。

```
let value: Any = "John"

let name = value as? String
print(name) // 输出 Optional("John")

let age = value as? Double
print(age) // 输出 nil
```

as! 操作符强制类型转换。
它不返回可选值，但如果转换失败会崩溃。

```
let value: Any = "Paul"

let name = value as! String
print(name) // 输出 "Paul"

let age = value as! Double // 崩溃: "无法转换值..."
```

通常会将类型转换操作符与条件解包一起使用：

```
let value: Any = "George"

if let name = value as? String {
    print(name) // 输出 "George"
}

if let age = value as? Double {
    print(age) // 不会执行
}
```

第21.2节：Swift语言中的类型转换

类型转换

类型转换是一种检查实例类型的方法，或者将该实例视为其自身类层次结构中其他位置的不同超类或子类。

Swift中的类型转换通过is和as操作符实现。这两个操作符提供了一种简单且富有表现力的方式来检查值的类型或将值转换为不同的类型。

向下转换

某个类类型的常量或变量实际上可能指向其子类的实例。
当你认为情况如此时，可以尝试使用类型转换操作符（as? 或

Chapter 21: Type Casting

Section 21.1: Downcasting

A variable can be downcasted to a subtype using the *type cast operators* `as?`, and `as!`.

The `as?` operator *attempts* to cast to a subtype.
It can fail, therefore it returns an optional.

```
let value: Any = "John"

let name = value as? String
print(name) // prints Optional("John")

let age = value as? Double
print(age) // prints nil
```

The `as!` operator *forces* a cast.
It does not return an optional, but crashes if the cast fails.

```
let value: Any = "Paul"

let name = value as! String
print(name) // prints "Paul"

let age = value as! Double // crash: "Could not cast value..."
```

It is common to use type cast operators with conditional unwrapping:

```
let value: Any = "George"

if let name = value as? String {
    print(name) // prints "George"
}

if let age = value as? Double {
    print(age) // Not executed
}
```

Section 21.2: Type casting in Swift Language

Type Casting

Type casting is a way to check the type of an instance, or to treat that instance as a different superclass or subclass from somewhere else in its own class hierarchy.

Type casting in Swift is implemented with the is and as operators. These two operators provide a simple and expressive way to check the type of a value or cast a value to a different type.

Downcasting

A constant or variable of a certain class type may actually refer to an instance of a subclass behind the scenes. Where you believe this is the case, you can try to downcast to the subclass type with a type cast operator (as? or

as!).

由于向下类型转换可能失败，类型转换操作符有两种不同的形式。条件形式 as? 返回你尝试向下转换的类型的可选值。强制形式 as! 尝试进行向下转换，并将结果强制解包作为一个复合操作。

当你不确定向下转换是否会成功时，使用类型转换操作符的条件形式（as?）。这种形式的操作符总是返回一个可选值，如果无法进行向下转换，值将为 nil。
这使你能够检查向下转换是否成功。

只有当你确定向下转换一定会成功时，才使用类型转换操作符的强制形式（as!）。
如果你尝试向错误的类类型进行向下转换，这种形式的操作符会触发运行时错误。 了解更多。

字符串转整数和浮点数转换：

```
let numbers = "888.00"
let intValue = NSString(string: numbers).integerValue
print(intValue) // 输出 - 888


let numbers = "888.00"
let floatValue = NSString(string: numbers).floatValue
print(floatValue) // 输出 : 888.0
```

浮点数转字符串转换

```
let numbers = 888.00
let floatValue = String(numbers)
print(floatValue) // 输出 : 888.0

// 获取特定位数的小数值
let numbers = 888.00
let floatValue = String(format: "%.2f", numbers) // 这里 %.2f 表示小数点后保留两位数字我们可以根据需要使用
print(floatValue) // 输出 : "888.00"
```

整数转字符串值

```
let numbers = 888
let intValue = String(numbers)
print(intValue) // 输出 : "888"
```

浮点数转字符串值

```
let numbers = 888.00
let floatValue = String(numbers)
print(floatValue)
```

可选浮点数转字符串

```
let numbers: Any = 888.00
let floatValue = String(describing: numbers)
print(floatValue) // 输出 : 888.0
```

可选字符串转整数值

```
let hitCount = "100"
let data :AnyObject = hitCount
let score = Int(data as? String ?? "") ?? 0
print(score)
```

从 JSON 中向下转换值

as!).

Because downcasting can fail, the type cast operator comes in two different forms. The conditional form, as?, returns an optional value of the type you are trying to downcast to. The forced form, as!, attempts the downcast and force-unwraps the result as a single compound action.

Use the conditional form of the type cast operator (as?) when you are not sure if the downcast will succeed. This form of the operator will always return an optional value, and the value will be nil if the downcast was not possible. This enables you to check for a successful downcast.

Use the forced form of the type cast operator (as!) only when you are sure that the downcast will always succeed. This form of the operator will trigger a runtime error if you try to downcast to an incorrect class type. Know more.

String to Int & Float conversion : -

```
let numbers = "888.00"
let intValue = NSString(string: numbers).integerValue
print(intValue) // Output - 888


let numbers = "888.00"
let floatValue = NSString(string: numbers).floatValue
print(floatValue) // Output : 888.0
```

Float to String Conversion

```
let numbers = 888.00
let floatValue = String(numbers)
print(floatValue) // Output : 888.0

// Get Float value at particular decimal point
let numbers = 888.00
let floatValue = String(format: "%.2f", numbers) // Here %.2f will give 2 numbers after decimal points we can use as per our need
print(floatValue) // Output : "888.00"
```

Integer to String value

```
let numbers = 888
let intValue = String(numbers)
print(intValue) // Output : "888"
```

Float to String value

```
let numbers = 888.00
let floatValue = String(numbers)
print(floatValue)
```

Optional Float value to String

```
let numbers: Any = 888.00
let floatValue = String(describing: numbers)
print(floatValue) // Output : 888.0
```

Optional String to Int value

```
let hitCount = "100"
let data :AnyObject = hitCount
let score = Int(data as? String ?? "") ?? 0
print(score)
```

Downcasting values from JSON

```
    let json = ["name" : "john", "subjects": ["Maths", "Science", "English", "C Language"]] as
[String : Any]
    let name = json["name"] as? String ?? ""
    print(name) // 输出 : john
    let subjects = json["subjects"] as? [String] ?? []
    print(subjects) // 输出 : ["Maths", "Science", "English", "C Language"]
```

从可选的 JSON 中向下转换值

```
    let response: Any = ["name" : "john", "subjects": ["Maths", "Science", "English", "C
Language"]]
    let json = response as? [String: Any] ?? [:]
    let name = json["name"] as? String ?? ""
    print(name) // 输出 : john
    let subjects = json["subjects"] as? [String] ?? []
    print(subjects) // 输出 : ["Maths", "Science", "English", "C Language"]
```

管理带条件的 JSON 响应

```
    let response: Any = ["name" : "john", "subjects": ["Maths", "Science", "English", "C
Language"]] //可选响应

guard let json = response as? [String: Any] else {
        // 在此处理 nil 值
        print("空字典")
        // 在此执行操作
        return
    }
    let name = json["name"] as? String ?? ""
    print(name) // 输出 : john
    let subjects = json["subjects"] as? [String] ?? []
    print(subjects) // 输出 : ["Maths", "Science", "English", "C Language"]
```

管理带条件的 Nil 响应

```
    let response: Any? = nil
guard let json = response as? [String: Any] else {
        // 在此处理 nil 值
        print("空字典")
        // 在此执行操作
        return
    }
    let name = json["name"] as? String ?? ""
    print(name)
    let subjects = json["subjects"] as? [String] ?? []
    print(subjects)
```

输出 : 空字典

第21.3节：向上转型

as 操作符将会转换为超类型。由于它不会失败，因此不会返回可选类型。

```
let name = "Ringo"
let value = string as Any  // `value` 现在的类型是 `Any`
```

第21.4节：在函数上使用向下转换的示例


```
    let json = ["name" : "john", "subjects": ["Maths", "Science", "English", "C Language"]] as
[String : Any]
    let name = json["name"] as? String ?? ""
    print(name) // Output : john
    let subjects = json["subjects"] as? [String] ?? []
    print(subjects) // Output : ["Maths", "Science", "English", "C Language"]
```

Downcasting values from Optional JSON

```
    let response: Any = ["name" : "john", "subjects": ["Maths", "Science", "English", "C
Language"]]
    let json = response as? [String: Any] ?? [:]
    let name = json["name"] as? String ?? ""
    print(name) // Output : john
    let subjects = json["subjects"] as? [String] ?? []
    print(subjects) // Output : ["Maths", "Science", "English", "C Language"]
```

Manage JSON Response with conditions

```
    let response: Any = ["name" : "john", "subjects": ["Maths", "Science", "English", "C
Language"]] //Optional Response

    guard let json = response as? [String: Any] else {
        // Handle here nil value
        print("Empty Dictionary")
        // Do something here
        return
    }
    let name = json["name"] as? String ?? ""
    print(name) // Output : john
    let subjects = json["subjects"] as? [String] ?? []
    print(subjects) // Output : ["Maths", "Science", "English", "C Language"]
```

Manage Nil Response with condition

```
    let response: Any? = nil
    guard let json = response as? [String: Any] else {
        // Handle here nil value
        print("Empty Dictionary")
        // Do something here
        return
    }
    let name = json["name"] as? String ?? ""
    print(name)
    let subjects = json["subjects"] as? [String] ?? []
    print(subjects)
```

Output : `Empty Dictionary`

Section 21.3: Upcasting

The as operator will cast to a supertype. As it cannot fail, it does not return an optional.

```
let name = "Ringo"
let value = string as Any  // `value` is of type `Any` now
```

Section 21.4: Example of using a downcast on a function

涉及子类化的参数

向下转换可以用来在接受其超类参数的函数内部使用子类的代码和数据。

```
class Rat {
     var color = "white"
}

class PetRat: Rat {
     var name = "Spot"
}

func nameOfRat(: Rat) -> String {
    guard let petRat = ( as? PetRat) else {
          return "无名"
     }

     return petRat.name
}

let noName = Rat()
let spot = PetRat()

print(nameOfRat(noName))
print(nameOfRat(spot))
```

第21.5节：使用switch进行类型转换

switch语句也可以用来尝试转换成不同的类型：

```
func checkType(_ value: Any) -> String {
    switch value {

    // `is` 操作符可以用来检查类型
    case is Double:
         return "值是Double类型"

    // `as` 操作符将进行转换。在`switch`中不需要使用`as?`。
    case let string as String:
         return "值是字符串：\(string)"

    default:
         return "值是其他内容"
    }

}

checkType("字符串")  // "值是字符串: 字符串"
checkType(6.28)       // "值是双精度数"
checkType(UILabel()) // "值是其他内容"
```


parameter involving subclassing

A downcast can be used to make use of a subclass's code and data inside of a function taking a parameter of its superclass.

```
class Rat {
    var color = "white"
}

class PetRat: Rat {
    var name = "Spot"
}

func nameOfRat(☐: Rat) -> String {
    guard let petRat = (☐ as? PetRat) else {
        return "No name"
    }

    return petRat.name
}

let noName = Rat()
let spot = PetRat()

print(nameOfRat(noName))
print(nameOfRat(spot))
```

Section 21.5: Casting with switch

The `switch` statement can also be used to attempt casting into different types:

```
func checkType(_ value: Any) -> String {
    switch value {

    // The `is` operator can be used to check a type
    case is Double:
        return "value is a Double"

    // The `as` operator will cast. You do not need to use `as?` in a `switch`.
    case let string as String:
        return "value is the string: \(string)"

    default:
        return "value is something else"
    }

}

checkType("Cadena")  // "value is the string: Cadena"
checkType(6.28)      // "value is a Double"
checkType(UILabel()) // "value is something else"
```

第22章：泛型

第22.1节：泛型基础

泛型是类型的占位符，允许你编写可应用于多种类型的灵活代码。
使用泛型而非Any的优势在于它们仍然允许编译器强制执行严格的类型安全。

泛型占位符定义在尖括号内<>。

泛型函数

对于函数，这个占位符放在函数名之后：

```
/// 随机选择一个输入并返回它
func pickRandom
```

;T>(_ a:T, _ b:T) -> T { return arc4random_uniform(2) == 0 ? a : b }

在这种情况下，泛型占位符是T。当你调用该函数时，Swift可以为你推断T的类型（因为它只是实际类型的占位符）。

```
let randomOutput = pickRandom(5, 7) // 返回一个Int（要么是5，要么是7）
```

这里我们传递了两个整数给函数。因此Swift推断T == Int——因此函数签名被推断为(Int, Int) -> Int。

由于泛型提供了强类型安全——函数的参数和返回值必须是相同的类型。因此以下代码无法编译：

```
struct Foo {}

let foo = Foo()

let randomOutput = pickRandom(foo, 5) // 错误：无法将类型 'Int' 的值转换为预期的参数类型 'Foo'
```

泛型类型

为了在类、结构体或枚举中使用泛型，你可以在类型名称后定义泛型占位符。

class Bar;T> { var baz : T init(baz:T) { self.baz = baz } }

这个通用占位符在你使用类Bar时需要一个类型。在这种情况下，可以从初始化器init(baz:T)推断出来。

```
let bar = Bar(baz: "一个字符串") // bar的类型是Bar<String>
```

这里通用占位符T被推断为类型String，因此创建了一个Bar<String>实例。你也可以显式指定类型：

let bar = Bar;String>(baz: "一个字符串")

当与类型一起使用时，给定的通用占位符将在该实例的整个生命周期内保持其类型，初始化后不能更改。因此，当你访问属性baz时，它将始终是
该实例的String类型。

Chapter 22: Generics

Section 22.1: The Basics of Generics

Generics are placeholders for types, allowing you to write flexible code that can be applied across multiple types. The advantage of using generics over Any is that they still allow the compiler to enforce strong type-safety.

A generic placeholder is defined within angle brackets <**>**.

Generic Functions

For functions, this placeholder is placed after the function name:

```
/// Picks one of the inputs at random, and returns it
func pickRandom
```

;T>(_ a:**T**, _ b:**T**) -> **T** { return arc4random_uniform(2) == 0 ? a : b }

In this case, the generic placeholder is `T`. When you come to call the function, Swift can infer the type of `T` for you (as it simply acts as a placeholder for an actual type).

```
let randomOutput = pickRandom(5, 7) // returns an Int (that's either 5 or 7)
```

Here we're passing two integers to the function. Therefore Swift is inferring `T == Int` – thus the function signature is inferred to be `(Int, Int) -> Int`.

Because of the strong type safety that generics offer – both the arguments and return of the function must be the *same* type. Therefore the following will not compile:

```
struct Foo {}

let foo = Foo()

let randomOutput = pickRandom(foo, 5) // error: cannot convert value of type 'Int' to expected
argument type 'Foo'
```

Generic Types

In order to use generics with classes, structs or enums, you can define the generic placeholder after the type name.

class Bar;T> { var baz : **T** init(baz:**T**) { self.baz = baz } }

This generic placeholder will require a type when you come to use the class `Bar`. In this case, it can be inferred from the initialiser `init(baz:T)`.

```
let bar = Bar(baz: "a string") // bar's type is Bar<String>
```

Here the generic placeholder `T` is inferred to be of type `String`, thus creating a `Bar<String>` instance. You can also specify the type explicitly:

let bar = Bar;String>(baz: "a string")

When used with a type, the given generic placeholder will keep its type for the entire lifetime of the given instance, and cannot be changed after initialisation. Therefore when you access the property `baz`, it will always be of type `String` for this given instance.

```
let str = bar.baz // 类型为String
```

传递通用类型

当你传递通用类型时，在大多数情况下你必须明确你期望的通用占位符类型。例如，作为函数输入：

```
func takeABarInt(bar:Bar;Int>) { ... }
```

此函数只接受Bar<Int>。尝试传入泛型占位符类型不是Int的Bar实例将导致编译错误。

泛型占位符命名

泛型占位符名称不仅限于单个字母。如果某个占位符代表一个有意义的概念，应该给它一个描述性的名称。例如，Swift的Array有一个名为Element的泛型占位符，定义了给定Array实例的元素类型。

```
public struct Array;Element> : RandomAccessCollection, MutableCollection { ... }
```

第22.2节：约束泛型占位符类型

可以强制泛型类的类型参数实现某个协议，例如Equatable

```
class MyGenericClass<Type: Equatable>{

    var value: Type
    init(value: Type){
        self.value = value
    }

    func getValue() -> Type{
        return self.value
    }

    func valueEquals(anotherValue: Type) -> Bool{
        return self.value == anotherValue
    }
}
```

每当我们创建一个新的MyGenericClass时，类型参数必须实现Equatable协议
（确保类型参数可以使用==与同类型的另一个变量进行比较）

```
let myFloatGeneric = MyGenericClass<Double>(value: 2.71828) // 有效
let myStringGeneric = MyGenericClass<String>(value: "My String") // 有效

// "类型[Int]不符合协议'Equatable'"
let myInvalidGeneric = MyGenericClass<[Int]>(value: [2])

let myIntGeneric = MyGenericClass<Int>(value: 72)
print(myIntGeneric.valueEquals(72)) // true
print(myIntGeneric.valueEquals(-274)) // false

// "无法将类型'String'的值转换为预期的参数类型'Int'"
print(myIntGeneric.valueEquals("My String"))
```



```
let str = bar.baz // of type String
```

Passing Around Generic Types

When you come to pass around generic types, in most cases you have to be explicit about the generic placeholder type you expect. For example, as a function input:

```
func takeABarInt(bar:Bar;Int>) { ... }
```

This function will only accept a `Bar<Int>`. Attempting to pass in a `Bar` instance where the generic placeholder type is not `Int` will result in a compiler error.

Generic Placeholder Naming

Generic placeholder names are not just limited to single letters. If a given placeholder represents a meaningful concept, you should give it a descriptive name. For example, Swift's `Array` has a generic placeholder called `Element`, which defines the element type of a given `Array` instance.

```
public struct Array;Element> : RandomAccessCollection, MutableCollection { ... }
```

Section 22.2: Constraining Generic Placeholder Types

It is possible to force the type parameters of a generic class to implement a protocol, for example, `Equatable`

```
class MyGenericClass<Type: Equatable>{

    var value: Type
    init(value: Type){
        self.value = value
    }

    func getValue() -> Type{
        return self.value
    }

    func valueEquals(anotherValue: Type) -> Bool{
        return self.value == anotherValue
    }
}
```

Whenever we create a new `MyGenericClass`, the type parameter has to implement the `Equatable` protocol (ensuring the type parameter can be compared to another variable of the same type using ==)

```
let myFloatGeneric = MyGenericClass<Double>(value: 2.71828) // valid
let myStringGeneric = MyGenericClass<String>(value: "My String") // valid

// "Type [Int] does not conform to protocol 'Equatable'"
let myInvalidGeneric = MyGenericClass<[Int]>(value: [2])

let myIntGeneric = MyGenericClass<Int>(value: 72)
print(myIntGeneric.valueEquals(72)) // true
print(myIntGeneric.valueEquals(-274)) // false

// "Cannot convert value of type 'String' to expected argument type 'Int'"
print(myIntGeneric.valueEquals("My String"))
```

第22.3节：泛型类示例

带有类型参数Type的泛型类

```
class MyGenericClass<Type>{

    var value: Type
    init(value: Type){
        self.value = value
    }

    func getValue() -> Type{
        return self.value
    }

    func setValue(value: Type){
        self.value = value
    }
}
```

我们现在可以使用类型参数创建新对象

```
let myStringGeneric = MyGenericClass<String>(value: "My String Value")
let myIntGeneric = MyGenericClass<Int>(value: 42)

print(myStringGeneric.getValue()) // "My String Value"
print(myIntGeneric.getValue()) // 42

myStringGeneric.setValue("Another String")
myIntGeneric.setValue(1024)

print(myStringGeneric.getValue()) // "Another String"
print(myIntGeneric.getValue()) // 1024
```

泛型也可以使用多个类型参数创建

```
class AnotherGenericClass<TypeOne, TypeTwo, TypeThree>{

    var value1: TypeOne
    var value2: TypeTwo
    var value3: TypeThree
    init(value1: TypeOne, value2: TypeTwo, value3: TypeThree){
        self.value1 = value1
        self.value2 = value2
        self.value3 = value3
    }

    func getValueOne() -> TypeOne{return self.value1}
    func getValueTwo() -> TypeTwo{return self.value2}
    func getValueThree() -> TypeThree{return self.value3}
}
```

并以相同方式使用

```
        let myGeneric = AnotherGenericClass<String, Int, Double>(value1: "Value of pi", value2: 3, value3:
3.14159)

print(myGeneric.getValueOne() 是String) // true
print(myGeneric.getValueTwo() 是Int) // true
```


Section 22.3: Generic Class Examples

A generic class with the type parameter `Type`

```
class MyGenericClass<Type>{

    var value: Type
    init(value: Type){
        self.value = value
    }

    func getValue() -> Type{
        return self.value
    }

    func setValue(value: Type){
        self.value = value
    }
}
```

We can now create new objects using a type parameter

```
let myStringGeneric = MyGenericClass<String>(value: "My String Value")
let myIntGeneric = MyGenericClass<Int>(value: 42)

print(myStringGeneric.getValue()) // "My String Value"
print(myIntGeneric.getValue()) // 42

myStringGeneric.setValue("Another String")
myIntGeneric.setValue(1024)

print(myStringGeneric.getValue()) // "Another String"
print(myIntGeneric.getValue()) // 1024
```

Generics can also be created with multiple type parameters

```
class AnotherGenericClass<TypeOne, TypeTwo, TypeThree>{

    var value1: TypeOne
    var value2: TypeTwo
    var value3: TypeThree
    init(value1: TypeOne, value2: TypeTwo, value3: TypeThree){
        self.value1 = value1
        self.value2 = value2
        self.value3 = value3
    }

    func getValueOne() -> TypeOne{return self.value1}
    func getValueTwo() -> TypeTwo{return self.value2}
    func getValueThree() -> TypeThree{return self.value3}
}
```

And used in the same way

```
let myGeneric = AnotherGenericClass<String, Int, Double>(value1: "Value of pi", value2: 3, value3:
3.14159)

print(myGeneric.getValueOne() is String) // true
print(myGeneric.getValueTwo() is Int) // true
```

```
print(myGeneric.getValueThree() 是Double) // true
print(myGeneric.getValueTwo() 是String) // false

print(myGeneric.getValueOne()) // "Value of pi"
print(myGeneric.getValueTwo()) // 3
print(myGeneric.getValueThree()) // 3.14159
```

第22.4节：使用泛型简化数组函数

通过创建面向对象的移除函数来扩展数组功能的函数。

```
// 需要将扩展限制为可比较的元素。
// `Element` 是数组为其元素类型定义的泛型名称。
// 该限制还使我们可以访问 `index(of:_)`，该方法也在带有 `where Element: Equatable` 的
数组扩展中定义。
公共扩展数组其中元素:可比较{
    /// 从数组中移除给定的对象。
    变异函数 remove(_ element: Element) {
        如果 let index = self.index(of: element ) {
            self.remove(at: index)
        } 否则 {
            致命错误("移除错误，数组中不存在该元素：\"\(element)\"。")}

    }
}
```

用法

```
变量 myArray = [1,2,3]
打印(myArray)

// 输出 [1,2,3]
```

使用该函数可以在不需要索引的情况下移除元素。只需传入要移除的对象即可。

```
myArray.remove(2)
print(myArray)

// 输出 [1,3]
```

第22.5节：高级类型约束

可以使用where子句为泛型指定多个类型约束：

```
func doSomething<T where T: Comparable, T: Hashable>(first: T, second: T) {
    // 访问可哈希函数
guard first.hashValue == second.hashValue else {
        return
    }
    // 访问可比较函数
    if first == second {
        print("\(first) 和 \(second) 相等。")
    }
}
```

也可以将where子句写在参数列表之后：

```
func doSomething<T>(first: T, second: T) where T: Comparable, T: Hashable {
```



```
print(myGeneric.getValueThree() is Double) // true
print(myGeneric.getValueTwo() is String) // false

print(myGeneric.getValueOne()) // "Value of pi"
print(myGeneric.getValueTwo()) // 3
print(myGeneric.getValueThree()) // 3.14159
```

Section 22.4: Using Generics to Simplify Array Functions

A function that extends the functionality of the array by creating an object oriented remove function.

```
// Need to restrict the extension to elements that can be compared.
// The `Element` is the generics name defined by Array for its item types.
// This restriction also gives us access to `index(of:_)` which is also
// defined in an Array extension with `where Element: Equatable`.
public extension Array where Element: Equatable {
    /// Removes the given object from the array.
    mutating func remove(_ element: Element) {
        if let index = self.index(of: element ) {
            self.remove(at: index)
        } else {
            fatalError("Removal error, no such element:\"\(element)\" in array.\n")
        }
    }
}
```

Usage

```
var myArray = [1,2,3]
print(myArray)

// Prints [1,2,3]
```

Use the function to remove an element without need for an index. Just pass the object to remove.

```
myArray.remove(2)
print(myArray)

// Prints [1,3]
```

Section 22.5: Advanced Type Constraints

It's possible to specify several type constraints for generics using the `where` clause:

```
func doSomething<T where T: Comparable, T: Hashable>(first: T, second: T) {
    // Access hashable function
    guard first.hashValue == second.hashValue else {
        return
    }
    // Access comparable function
    if first == second {
        print("\(first) and \(second) are equal.")
    }
}
```

It's also valid to write the `where` clause after the argument list:

```
func doSomething<T>(first: T, second: T) where T: Comparable, T: Hashable {
```

```
    // 访问可哈希函数
    guard first.hashValue == second.hashValue else {
        return
    }
    // 访问可比较函数
    if first == second {
        print("\(first) 和 \(second) 相等。")
    }
}
```

扩展可以限制为满足条件的类型。该函数仅适用于满足类型条件的实例：

```
// “Element” 是由 “Array” 定义的泛型类型。对于此示例，
// 我们想添加一个函数，该函数要求 “Element” 可比较，
// 即：它需要遵循 Equatable 协议。
公共扩展数组其中元素:可比较{
    /// 从数组中移除给定的对象。
    变异函数 remove(_ element: Element) {
        // 我们也可以在这里使用 “self.index(of: element)”，因为 “index(of:_)”
        // 也在一个带有 “where Element: Equatable” 的扩展中定义。
        // 为了这个示例，显式使用 Equatable。
        如果 let index = self.index(where: { $0 == element }) {
            self.remove(at: index)
        } 否则 {
            致命错误("移除错误，数组中不存在该元素：\"\(element)\"。")}

    }
}
```

第 22.6 节：泛型类继承

泛型类可以被继承：

```
// 模型
类 MyFirstModel {
}

类 MySecondModel: MyFirstModel {
}

// 通用类
class MyFirstGenericClass<T: MyFirstModel> {

    func doSomethingWithModel(model: T) {
        // 在这里做一些事情
    }

}

class MySecondGenericClass<T: MySecondModel>: MyFirstGenericClass<T> {

    override func doSomethingWithModel(model: T) {
        super.doSomethingWithModel(model)

        // 在这里做更多事情
    }

}
```



```
    // Access hashable function
    guard first.hashValue == second.hashValue else {
        return
    }
    // Access comparable function
    if first == second {
        print("\(first) and \(second) are equal.")
    }
}
```

Extensions can be restricted to types that satisfy conditions. The function is only available to instances which satisfy the type conditions:

```
// "Element" is the generics type defined by "Array". For this example, we
// want to add a function that requires that "Element" can be compared, that
// is: it needs to adhere to the Equatable protocol.
public extension Array where Element: Equatable {
    /// Removes the given object from the array.
    mutating func remove(_ element: Element) {
        // We could also use "self.index(of: element)" here, as "index(of:_)"
        // is also defined in an extension with "where Element: Equatable".
        // For the sake of this example, explicitly make use of the Equatable.
        if let index = self.index(where: { $0 == element }) {
            self.remove(at: index)
        } else {
            fatalError("Removal error, no such element:\"\(element)\" in array.\n")
        }
    }
}
```

Section 22.6: Generic Class Inheritance

Generic classes can be inherited:

```
// Models
class MyFirstModel {
}

class MySecondModel: MyFirstModel {
}

// Generic classes
class MyFirstGenericClass<T: MyFirstModel> {

    func doSomethingWithModel(model: T) {
        // Do something here
    }

}

class MySecondGenericClass<T: MySecondModel>: MyFirstGenericClass<T> {

    override func doSomethingWithModel(model: T) {
        super.doSomethingWithModel(model)

        // Do more things here
    }

}
```

第22.7节：使用泛型增强类型安全性

让我们来看一个不使用泛型的例子

```
protocol JSONDecodable {
    static func from(_ json: [String: Any]) -> Any?
}
```

协议声明看起来没问题，除非你真正使用它。

```
let myTestObject = TestObject.from(myJson) as? TestObject
```

为什么你必须将结果强制转换为TestObject？因为协议声明中的返回类型是Any。

通过使用泛型，你可以避免这种可能导致运行时错误的问题（我们不希望出现错误！）

```
protocol JSONDecodable {
    associatedtype Element
    static func from(_ json: [String: Any]) -> Element?
}

struct TestObject: JSONDecodable {
    static func from(_ json: [String: Any]) -> TestObject? {
    }
}

let testObject = TestObject.from(myJson) // 现在 testObject 自动推断为 `TestObject?`
```


Section 22.7: Use generics to enhance type-safety

Let's take this example without using generics

```
protocol JSONDecodable {
    static func from(_ json: [String: Any]) -> Any?
}
```

The protocol declaration seems fine unless you actually use it.

```
let myTestObject = TestObject.from(myJson) as? TestObject
```

Why do you have to cast the result to `TestObject`? Because of the `Any` return type in the protocol declaration.

By using generics you can avoid this problem that can cause runtime errors (and we don't want to have them!)

```
protocol JSONDecodable {
    associatedtype Element
    static func from(_ json: [String: Any]) -> Element?
}

struct TestObject: JSONDecodable {
    static func from(_ json: [String: Any]) -> TestObject? {
    }
}

let testObject = TestObject.from(myJson) // testObject is now automatically `TestObject?`
```

第23章：OptionSet

第23.1节：OptionSet 协议

OptionSetType 是一个协议，设计用于表示位掩码类型，其中各个位表示集合的成员。
一组逻辑和/或函数强制执行正确的语法：

```
结构体 Features : OptionSet {
  让 rawValue : Int
  static let none = Features(rawValue: 0)
  static let feature0 = Features(rawValue: 1 << 0)
  static let feature1 = Features(rawValue: 1 << 1)
  static let feature2 = Features(rawValue: 1 << 2)
  static let feature3 = Features(rawValue: 1 << 3)
  static let feature4 = Features(rawValue: 1 << 4)
  static let feature5 = Features(rawValue: 1 << 5)
  static let all: Features = [feature0, feature1, feature2, feature3, feature4, feature5]
}

Features.feature1.rawValue //2
Features.all.rawValue //63

var options: Features = [.feature1, .feature2, .feature3]

options.contains(.feature1) //true
options.contains(.feature4) //false

options.insert(.feature4)
options.contains(.feature4) //true


var otherOptions : Features = [.feature1, .feature5]

options.contains(.feature5) //false

options.formUnion(otherOptions)
options.contains(.feature5) //true

options.remove(.feature5)
options.contains(.feature5) //false
```


Chapter 23: OptionSet

Section 23.1: OptionSet Protocol

OptionSetType is a protocol designed to represent bit mask types where individual bits represent members of the set. A set of logical and/or functions enforce the proper syntax:

```
struct Features : OptionSet {
  let rawValue : Int
  static let none = Features(rawValue: 0)
  static let feature0 = Features(rawValue: 1 << 0)
  static let feature1 = Features(rawValue: 1 << 1)
  static let feature2 = Features(rawValue: 1 << 2)
  static let feature3 = Features(rawValue: 1 << 3)
  static let feature4 = Features(rawValue: 1 << 4)
  static let feature5 = Features(rawValue: 1 << 5)
  static let all: Features = [feature0, feature1, feature2, feature3, feature4, feature5]
}

Features.feature1.rawValue //2
Features.all.rawValue //63

var options: Features = [.feature1, .feature2, .feature3]

options.contains(.feature1) //true
options.contains(.feature4) //false

options.insert(.feature4)
options.contains(.feature4) //true


var otherOptions : Features = [.feature1, .feature5]

options.contains(.feature5) //false

options.formUnion(otherOptions)
options.contains(.feature5) //true

options.remove(.feature5)
options.contains(.feature5) //false
```

第24章：读取与写入JSON

第24.1节：使用苹果Foundation和Swift标准库进行JSON序列化、编码和解码

JSONSerialization类内置于苹果的Foundation框架中。

版本 = 2.2

读取JSON

JSONObjectWithData函数接受NSData，并返回AnyObject。你可以使用as?将结果转换为你期望的类型。

```
do {
guard let jsonData = "[\"Hello\", \"JSON\"]".dataUsingEncoding(NSUTF8StringEncoding) else {
        fatalError("无法将字符串编码为UTF-8")
    }

    // 将JSON从NSData转换为AnyObject
    let jsonObject = try NSJSONSerialization.JSONObjectWithData(jsonData, options: [])

    // 尝试将AnyObject转换为字符串数组
    if let stringArray = jsonObject as? [String] {
        print("获得字符串数组: \(stringArray.joinWithSeparator(", "))")
    }
} catch {
    print("读取JSON时出错: \(error)")
}
```

你可以传入options: .AllowFragments代替options: []，以允许在顶层对象不是数组或字典时读取JSON。

写入 JSON

调用dataWithJSONObject将一个JSON兼容的对象（嵌套数组或字典，包含字符串、数字和NSNull）转换为以UTF-8编码的原始NSData。

```
do {
    // 将对象转换为JSON格式的NSData
    let jsonData = try NSJSONSerialization.dataWithJSONObject(jsonObject, options: [])
    print("JSON数据: \(jsonData)")

    // 将NSData转换为字符串
    let jsonString = String(data: jsonData, encoding: NSUTF8StringEncoding)!
    print("JSON字符串: \(jsonString)")
} catch {
    print("写入JSON时出错: \(error)")
}
```

你可以传入options: .PrettyPrinted来代替options: []以实现美化打印。

版本 = 3.0

Swift 3中行为相同，但语法不同。

```
do {
guard let jsonData = "[\"Hello\", \"JSON\"]".data(using: String.Encoding.utf8) else {
```


Chapter 24: Reading & Writing JSON

Section 24.1: JSON Serialization, Encoding, and Decoding with Apple Foundation and the Swift Standard Library

The JSONSerialization class is built into Apple's Foundation framework.

Version = 2.2

Read JSON

The `JSONObjectWithData` function takes `NSData`, and returns `AnyObject`. You can use `as?` to convert the result to your expected type.

```
do {
    guard let jsonData = "[\"Hello\", \"JSON\"]".dataUsingEncoding(NSUTF8StringEncoding) else {
        fatalError("couldn't encode string as UTF-8")
    }

    // Convert JSON from NSData to AnyObject
    let jsonObject = try NSJSONSerialization.JSONObjectWithData(jsonData, options: [])

    // Try to convert AnyObject to array of strings
    if let stringArray = jsonObject as? [String] {
        print("Got array of strings: \(stringArray.joinWithSeparator(", "))")
    }
} catch {
    print("error reading JSON: \(error)")
}
```

You can pass `options: .AllowFragments` instead of `options: []` to allow reading JSON when the top-level object isn't an array or dictionary.

Write JSON

Calling `dataWithJSONObject` converts a JSON-compatible object (nested arrays or dictionaries with strings, numbers, and `NSNull`) to raw `NSData` encoded as UTF-8.

```
do {
    // Convert object to JSON as NSData
    let jsonData = try NSJSONSerialization.dataWithJSONObject(jsonObject, options: [])
    print("JSON data: \(jsonData)")

    // Convert NSData to String
    let jsonString = String(data: jsonData, encoding: NSUTF8StringEncoding)!
    print("JSON string: \(jsonString)")
} catch {
    print("error writing JSON: \(error)")
}
```

You can pass `options: .PrettyPrinted` instead of `options: []` for pretty-printing.

Version = 3.0

Same behavior in Swift 3 but with a different syntax.

```
do {
    guard let jsonData = "[\"Hello\", \"JSON\"]".data(using: String.Encoding.utf8) else {
```

```
        fatalError("无法将字符串编码为UTF-8")
    }

    // 将JSON从NSData转换为AnyObject
    let jsonObject = try JSONSerialization.jsonObject(with: jsonData, options: [])

    // 尝试将AnyObject转换为字符串数组
    if let stringArray = jsonObject as? [String] {
        print("获取到字符串数组: \(stringArray.joined(separator: ", "))")
    }
} catch {
    print("读取JSON时出错: \(error)")
}

do {
    // 将对象转换为JSON格式的NSData
    let jsonData = try JSONSerialization.data(withJSONObject: jsonObject, options: [])
    print("JSON数据: \(jsonData)")

    // 将NSData转换为字符串
    let jsonString = String(data: jsonData, encoding: .utf8)!
    print("JSON字符串: \(jsonString)")
} catch {
    print("写入JSON时出错: \(error)")
}
```

> 注意：以下内容目前仅在Swift 4.0及更高版本中可用。

从Swift 4.0开始，Swift标准库包含了协议Encodable和Decodable，用于定义数据编码和解码的标准化方法。采用这些协议将允许Encoder和Decoder协议的实现对您的数据进行编码或解码，转换为或从外部表示形式（如JSON）。

遵循Codable协议结合了Encodable和Decodable协议。这现在是程序中处理JSON的推荐方式。

自动编码和解码

使类型可编码的最简单方法是将其属性声明为已经符合Codable的类型。这些类型包括标准库类型，如String、Int和Double；以及Foundation类型，如Date、Data和URL。如果一个类型的属性是可编码的，则该类型本身只需声明符合Codable协议即可自动符合该协议。

考虑以下示例，其中Book结构体符合Codable协议。

```
struct Book: Codable {
    let title: String
    let authors: [String]
    let publicationDate: Date
}
```

> 请注意，如果标准集合如Array和Dictionary包含可编码类型，则它们也符合Codable协议。

通过采用Codable，Book结构体现在可以使用Apple Foundation类JSONEncoder和JSONDecoder进行JSON的编码和解码，尽管Book本身没有专门处理JSON的代码。


```
        fatalError("couldn't encode string as UTF-8")
    }

    // Convert JSON from NSData to AnyObject
    let jsonObject = try JSONSerialization.jsonObject(with: jsonData, options: [])

    // Try to convert AnyObject to array of strings
    if let stringArray = jsonObject as? [String] {
        print("Got array of strings: \(stringArray.joined(separator: ", "))")
    }
} catch {
    print("error reading JSON: \(error)")
}

do {
    // Convert object to JSON as NSData
    let jsonData = try JSONSerialization.data(withJSONObject: jsonObject, options: [])
    print("JSON data: \(jsonData)")

    // Convert NSData to String
    let jsonString = String(data: jsonData, encoding: .utf8)!
    print("JSON string: \(jsonString)")
} catch {
    print("error writing JSON: \(error)")
}
```

> Note: The Following is currently available only in **Swift 4.0** and later.

As of Swift 4.0, the Swift standard library includes the protocols `Encodable` and `Decodable` to define a standardized approach to data encoding and decoding. Adopting these protocols will allow implementations of the `Encoder` and `Decoder` protocols take your data and encode or decode it to and from an external representation such as JSON. Conformance to the `Codable` protocol combines both the `Encodable` and `Decodable` protocols. This is now the recommended means to handle JSON in your program.

Encode and Decode Automatically

The easiest way to make a type codable is to declare its properties as types that are already `Codable`. These types include standard library types such as `String`, `Int`, and `Double`; and Foundation types such as `Date`, `Data`, and `URL`. If a type's properties are codable, the type itself will automatically conform to `Codable` by simply declaring the conformance.

Consider the following example, in which the `Book` structure conforms to `Codable`.

```
struct Book: Codable {
    let title: String
    let authors: [String]
    let publicationDate: Date
}
```

> Note that standard collections such as `Array` and `Dictionary` conform to `Codable` if they contain codable types.

By adopting `Codable`, the `Book` structure can now be encoded to and decoded from JSON using the Apple Foundation classes `JSONEncoder` and `JSONDecoder`, even though `Book` itself contains no code to specifically handle

也可以通过分别符合Encoder和Decoder协议来编写自定义的编码器和解码器。

编码为JSON数据

```
// 创建一个名为book的Book实例
let encoder = JSONEncoder()
let data = try! encoder.encode(book) // 生产代码中不要使用try!
print(data)
```

> 设置encoder.outputFormatting = .prettyPrinted以便更易阅读。## 从JSON数据解码

从JSON数据解码

```
// 从某个来源获取JSON字符串
let jsonData = jsonString.data(encoding: .utf8)!
let decoder = JSONDecoder()
let book = try! decoder.decode(Book.self, for: jsonData) // 生产代码中不要使用try!
print(book)
```

> 在上述示例中，Book.self告知解码器JSON应解码成的类型。

仅编码或仅解码

有时你可能不需要数据同时支持编码和解码，比如当你只需要从API读取JSON数据，或者你的程序只向API提交JSON数据时。

如果你只打算写入JSON数据，请让你的类型遵循Encodable协议。

```
struct Book: Encodable {
    let title: String
    let authors: [String]
    let publicationDate: Date
}
```

如果您只打算读取 JSON 数据，请使您的类型符合Decodable。

```
struct Book: Decodable {
    let title: String
    let authors: [String]
    let publicationDate: Date
}
```

使用自定义键名

API 经常使用除 Swift 标准驼峰命名法之外的命名约定，例如蛇形命名法。这在解码 JSON 时可能成为问题，因为默认情况下 JSON 键必须与类型的属性名完全匹配。为了解决这些情况，可以使用CodingKey协议为类型创建自定义键。

```
struct Book: Codable {
    // …
    enum CodingKeys: String, CodingKey {
        case title
        case authors
```


JSON. Custom encoders and decoders can be written, as well, by conforming to the `Encoder` and `Decoder` protocols, respectively.

Encode to JSON data

```
// Create an instance of Book called book
let encoder = JSONEncoder()
let data = try! encoder.encode(book) // Do not use try! in production code
print(data)
```

> Set `encoder.outputFormatting = .prettyPrinted` for easier reading. ## Decode from JSON data

Decode from JSON data

```
// Retrieve JSON string from some source
let jsonData = jsonString.data(encoding: .utf8)!
let decoder = JSONDecoder()
let book = try! decoder.decode(Book.self, for: jsonData) // Do not use try! in production code
print(book)
```

> In the above example, `Book.self` informs the decoder of the type to which the JSON should be decoded.

Encoding or Decoding Exclusively

Sometimes you may not need data to be both encodable and decodable, such as when you need only read JSON data from an API, or if your program only submits JSON data to an API.

If you intend only to write JSON data, conform your type to `Encodable`.

```
struct Book: Encodable {
    let title: String
    let authors: [String]
    let publicationDate: Date
}
```

If you intend only to read JSON data, conform your type to `Decodable`.

```
struct Book: Decodable {
    let title: String
    let authors: [String]
    let publicationDate: Date
}
```

Using Custom Key Names

APIs frequently use naming conventions other than the Swift-standard camel case, such as snake case. This can become an issue when it comes to decoding JSON, since by default the JSON keys must align exactly with your type's property names. To handle these scenarios you can create custom keys for your type using the `CodingKey` protocol.

```
struct Book: Codable {
    // ...
    enum CodingKeys: String, CodingKey {
        case title
        case authors
```

```
        case publicationDate = "publication_date"
    }
}
```

对于采用Codable协议的类型，CodingKeys 会自动生成，但通过在上例中创建我们自己的实现，允许解码器将本地驼峰命名的publicationDate与 API 传递的蛇形命名publication_date匹配。

第24.2节：SwiftyJSON

SwiftyJSON 是一个 Swift 框架，旨在消除普通 JSON 序列化中对可选链的需求。

您可以在这里下载：https://github.com/SwiftyJSON/SwiftyJSON

没有 SwiftyJSON，您的代码看起来会是这样，用来查找 JSON 对象中第一本书的名称：

```
if let jsonObject = try NSJSONSerialization.JSONObjectWithData(data, options: .AllowFragments) as?
[[String: AnyObject]],
let bookName = (jsonObject[0]["book"] as? [String: AnyObject])?["name"] as? String {
    //我们现在可以使用书名了
}
```

在 SwiftyJSON 中，这大大简化了：

```
let json = JSON(data: data)
if let bookName = json[0]["book"]["name"].string {
    //我们现在可以使用书名了
}
```

它省去了检查每个字段的需要，如果其中任何一个无效，它会返回 nil。

要使用 SwiftyJSON，请从 Git 仓库下载正确的版本——有一个 Swift 3 的分支。只需将 "SwiftyJSON.swift" 拖入您的项目并导入到您的类中：

```
import SwiftyJSON
```

您可以使用以下两个初始化方法创建您的 JSON 对象：

```
let jsonObject = JSON(data: dataObject)
```

或者

```
let jsonObject = JSON(jsonObject) //这可能是一个JSON格式的字符串，例如
```

要访问你的数据，使用下标：

```
let firstObjectInAnArray = jsonObject[0]
let nameOfFirstObject = jsonObject[0]["name"]
```

然后你可以将值解析为某种数据类型，这将返回一个可选值：

```
let nameOfFirstObject = jsonObject[0]["name"].string //这将返回名字的字符串

let nameOfFirstObject = jsonObject[0]["name"].double //这将返回空值
```



```
        case publicationDate = "publication_date"
    }
}
```

`CodingKeys` are generated automatically for types which adopt the `Codable` protocol, but by creating our own implementation in the example above we're allow our decoder to match the local camel case `publicationDate` with the snake case `publication_date` as it's delivered by the API.

Section 24.2: SwiftyJSON

SwiftyJSON is a Swift framework built to remove the need for optional chaining in normal JSON serialization.

You can download it here: https://github.com/SwiftyJSON/SwiftyJSON

Without SwiftyJSON, your code would look like this to find the name of the first book in a JSON object:

```
if let jsonObject = try NSJSONSerialization.JSONObjectWithData(data, options: .AllowFragments) as?
[[String: AnyObject]],
let bookName = (jsonObject[0]["book"] as? [String: AnyObject])?["name"] as? String {
    //We can now use the book name
}
```

In SwiftyJSON, this is hugely simplified:

```
let json = JSON(data: data)
if let bookName = json[0]["book"]["name"].string {
    //We can now use the book name
}
```

It removes the need to check every field, as it will return nil if any of them are invalid.

To use SwiftyJSON, download the correct version from the Git repository - there is a branch for Swift 3. Simply drag the "SwiftyJSON.swift" into your project and import into your class:

```
import SwiftyJSON
```

You can create your JSON object using the following two initializers:

```
let jsonObject = JSON(data: dataObject)
```

or

```
let jsonObject = JSON(jsonObject) //This could be a string in a JSON format for example
```

To access your data, use subscripts:

```
let firstObjectInAnArray = jsonObject[0]
let nameOfFirstObject = jsonObject[0]["name"]
```

You can then parse your value to a certain data type, which will return an optional value:

```
let nameOfFirstObject = jsonObject[0]["name"].string //This will return the name as a string

let nameOfFirstObject = jsonObject[0]["name"].double //This will return null
```

你也可以将路径编译成一个Swift数组：

```
let convolutedPath = jsonObject[0]["name"][2]["lastName"]["firstLetter"].string
```

等同于：

```
let convolutedPath = jsonObject[0, "name", 2, "lastName", "firstLetter"].string
```

SwiftyJSON 也具有打印自身错误的功能：

```
if let name = json[1337].string {
    //你可以使用该值——它是有效的
} 否则 {
    print(json[1337].error) // "Array[1337] 超出范围"——你不能使用该值
}
```

如果你需要写入你的 JSON 对象，可以再次使用下标：

```
var originalJSON:JSON = ["name": "Jack", "age": 18]
originalJSON["age"] = 25 //这将年龄改为 25
originalJSON["surname"] = "Smith" //这会创建一个名为 "surname" 的新字段并添加该值
it
```

如果你需要 JSON 的原始字符串，例如需要写入文件，可以获取原始值：

```
if let string = json.rawString() { //这是一个字符串对象
    //如果需要，可以将字符串写入文件
}

if let data = json.rawData() { //这是一个 NSData 对象
    //如果需要，可以将数据发送到你的服务器
}
```

第 24.3 节：弗雷迪

Freddy 是由Big Nerd Ranch维护的一个JSON解析库。它有三个主要优点：

1. 类型安全：帮助您以防止运行时崩溃的方式处理发送和接收 JSON。
2. 习惯用法：利用 Swift 的泛型、枚举和函数式特性，无需复杂的文档或神奇的自定义操作符。
3. 错误处理：为常见的 JSON 错误提供详细的错误信息。

示例 JSON 数据

让我们定义一些示例 JSON 数据以供这些示例使用。

```
{
  "success": true,
  "people": [
    {
      "name": "马特·马蒂亚斯",
      "age": 32,
      "spouse": true
    },
```


You can also compile your paths into a swift Array:

```
let convolutedPath = jsonObject[0]["name"][2]["lastName"]["firstLetter"].string
```

Is the same as:

```
let convolutedPath = jsonObject[0, "name", 2, "lastName", "firstLetter"].string
```

SwiftyJSON also has functionality to print its own errors:

```
if let name = json[1337].string {
    //You can use the value - it is valid
} else {
    print(json[1337].error) // "Array[1337] is out of bounds" - You cant use the value
}
```

If you need to write to your JSON object, you can use subscripts again:

```
var originalJSON:JSON = ["name": "Jack", "age": 18]
originalJSON["age"] = 25 //This changes the age to 25
originalJSON["surname"] = "Smith" //This creates a new field called "surname" and adds the value to
it
```

Should you need the original String for the JSON, for example if you need to write it to a file, you can get the raw value out:

```
if let string = json.rawString() { //This is a String object
    //Write the string to a file if you like
}

if let data = json.rawData() { //This is an NSData object
    //Send the data to your server if you like
}
```

Section 24.3: Freddy

Freddy is a JSON parsing library maintained by Big Nerd Ranch. It has three principal benefits:

1. Type Safety: Helps you work with sending and receiving JSON in a way that prevents runtime crashes.
2. Idiomatic: Takes advantage of Swift's generics, enumerations, and functional features, without complicated documentation or magical custom operators.
3. Error Handling: Provides informative error information for commonly occurring JSON errors.

Example JSON Data

Let's define some example JSON data for use with these examples.

```
{
  "success": true,
  "people": [
    {
      "name": "Matt Mathias",
      "age": 32,
      "spouse": true
    },
```

```
    {
      "name": "中士佩珀",
      "age": 25,
      "spouse": false
    }
  ],
  "jobs": [
    "教师",
    "法官"
  ],
  "states": {
    "乔治亚州": [
      30301,
      30302,
      30303
    ],
    "威斯康星州": [
      53000,
      53001
    ]
  }
}
```

```
let jsonString = "{\"success\": true, \"people\": [{\"name\": \"马特·马蒂亚斯\",\"age\":
32,\"spouse\": true},{\"name\": \"佩珀中士\",\"age\": 25,\"spouse\": false}],\"jobs\":
[\"教师\",\"法官\"],\"states\": {\"乔治亚州\": [30301,30302,30303],\"威斯康星州\":
[53000,53001]}}"
let jsonData = jsonString.dataUsingEncoding(NSUTF8StringEncoding)!
```

反序列化原始数据

要反序列化数据，我们初始化一个JSON对象，然后访问特定的键。

```
do {
    let json = try JSON(data: jsonData)
    let success = try json.bool("success")
} catch {
    // 处理错误
}
```

这里使用try是因为访问json中键"success"可能会失败——该键可能不存在，或者对应的值可能不是布尔类型。

我们还可以指定路径来访问嵌套在 JSON 结构中的元素。路径是由逗号分隔的键和索引列表，用于描述到感兴趣值的路径。

```
do {
    let json = try JSON(data: jsonData)
    let georgiaZipCodes = try json.array("states", "Georgia")
    let firstPersonName = try json.string("people", 0, "name")
} catch {
    // 处理错误
}
```

直接反序列化模型

JSON 可以直接解析为实现了 JSONDecodable 协议的模型类。

```
public struct Person {
    public let name: String
    public let age: Int
```



```
    {
      "name": "Sergeant Pepper",
      "age": 25,
      "spouse": false
    }
  ],
  "jobs": [
    "teacher",
    "judge"
  ],
  "states": {
    "Georgia": [
      30301,
      30302,
      30303
    ],
    "Wisconsin": [
      53000,
      53001
    ]
  }
}
```

```
let jsonString = "{\"success\": true, \"people\": [{\"name\": \"Matt Mathias\",\"age\":
32,\"spouse\": true},{\"name\": \"Sergeant Pepper\",\"age\": 25,\"spouse\": false}],\"jobs\":
[\"teacher\",\"judge\"],\"states\": {\"Georgia\": [30301,30302,30303],\"Wisconsin\":
[53000,53001]}}"
let jsonData = jsonString.dataUsingEncoding(NSUTF8StringEncoding)!
```

Deserializing Raw Data

To deserialize the data, we initialize a `JSON` object then access a particular key.

```
do {
    let json = try JSON(data: jsonData)
    let success = try json.bool("success")
} catch {
    // do something with the error
}
```

We `try` here because accessing the `json` for the key `"success"` could fail--it might not exist, or the value might not be a boolean.

We can also specify a path to access elements nested in the JSON structure. The path is a comma-separated list of keys and indices that describe the path to a value of interest.

```
do {
    let json = try JSON(data: jsonData)
    let georgiaZipCodes = try json.array("states", "Georgia")
    let firstPersonName = try json.string("people", 0, "name")
} catch {
    // do something with the error
}
```

Deserializing Models Directly

JSON can be directly parsed to a model class that implements the `JSONDecodable` protocol.

```
public struct Person {
    public let name: String
    public let age: Int
```

```
    public let spouse: Bool
}

extension Person: JSONDecodable {
    public init(json: JSON) throws {
        name = try json.string("name")
        age = try json.int("age")
        spouse = try json.bool("spouse")
    }
}

do {
    let json = try JSON(data: jsonData)
    let people = try json.arrayOf("people", type: Person.self)
} catch {
    // 处理错误
}
```

序列化原始数据

任何JSON值都可以直接序列化为NSData。

```
let success = JSON.Bool(false)
let data: NSData = try success.serialize()
```

直接序列化模型

任何实现了JSONEncodable协议的模型类都可以直接序列化为NSData。

```
extension Person: JSONEncodable {
    public func toJSON() -> JSON {
        return .Dictionary([
            "name": .String(name),
            "age": .Int(age),
            "spouse": .Bool(spouse)
        ])
    }
}

let newPerson = Person(name: "Glenn", age: 23, spouse: true)
let data: NSData = try newPerson.toJSON().serialize()
```

第24.4节：Swift 3中的JSON解析

这是我们将使用的名为animals.json的JSON文件

```
{
    "Sea Animals": [
    {
     "name": "鱼",
     "question": "鱼类有多少种？"    },
                 {
                 "name": "鲨鱼",
                 "question": "鲨鱼能活多久？"
                 },
                 {
                 "name": "鱿鱼",
                 "question": "鱿鱼有大脑吗？"
                 },
                 {
                 "name": "章鱼",
```



```
    public let spouse: Bool
}

extension Person: JSONDecodable {
    public init(json: JSON) throws {
        name = try json.string("name")
        age = try json.int("age")
        spouse = try json.bool("spouse")
    }
}

do {
    let json = try JSON(data: jsonData)
    let people = try json.arrayOf("people", type: Person.self)
} catch {
    // do something with the error
}
```

Serializing Raw Data

Any `JSON` value can be serialized directly to `NSData`.

```
let success = JSON.Bool(false)
let data: NSData = try success.serialize()
```

Serializing Models Directly

Any model class that implements the `JSONEncodable` protocol can be serialized directly to `NSData`.

```
extension Person: JSONEncodable {
    public func toJSON() -> JSON {
        return .Dictionary([
            "name": .String(name),
            "age": .Int(age),
            "spouse": .Bool(spouse)
        ])
    }
}

let newPerson = Person(name: "Glenn", age: 23, spouse: true)
let data: NSData = try newPerson.toJSON().serialize()
```

Section 24.4: JSON Parsing Swift 3

Here is the JSON File we will be using called `animals.json`

```
{
    "Sea Animals": [
    {
     "name": "Fish",
     "question": "How many species of fish are there?"    },
                 {
                 "name": "Sharks",
                 "question": "How long do sharks live?"
                 },
                 {
                 "name": "Squid",
                 "question": "Do squids have brains?"
                 },
                 {
                 "name": "Octopus",
```

```
                "question": "章鱼能长多大？"
                },
                {
                "name": "海星",
                "question": "海星能活多久？"
                }
                ],
    "哺乳动物": [
                {
      "名称": "狗",
      "问题": "狗的寿命有多长？"
    },
    {
      "名称": "大象",
      "问题": "小象的体重是多少？"
    },
    {
      "名称": "猫",
      "问题": "猫真的有九条命吗？"
    },
    {
      "名称": "老虎",
      "问题": "老虎生活在哪里？"
    },
    {
      "名称": "熊猫",
      "question": "熊猫吃什么？"
    }   ] }
```

在项目中导入你的 JSON 文件

你可以执行这个简单的函数来打印你的 JSON 文件

```
    func jsonParsingMethod() {
        //获取文件
        let filePath = Bundle.main.path(forResource: "animals", ofType: "json")
        let content = try! String(contentsOfFile: filePath!)

        let data: Data = content.data(using: String.Encoding.utf8)!
        let json: NSDictionary =  try! JSONSerialization.jsonObject(with: data as Data,
options:.mutableContainers) as! NSDictionary

//调用你想解析的文件部分
        if let results = json["mammals"] as? [[String: AnyObject]] {

            for res in results {
                //这将打印出文件中哺乳动物的名称。
                if let rates = res["name"] as? String {
                    print(rates)
                }
            }
        }
    }
```

如果你想把它放到表视图中，我会先用 NSObject 创建一个字典。

创建一个名为ParsingObject的新Swift文件，并创建你的字符串变量。

> 确保变量名与JSON文件中的名称相同


```
                "question": "How big do octopus get?"
                },
                {
                "name": "Star Fish",
                "question": "How long do star fish live?"
                }
                ],
    "mammals": [
                {
      "name": "Dog",
      "question": "How long do dogs live?"
    },
    {
      "name": "Elephant",
      "question": "How much do baby elephants weigh?"
    },
    {
      "name": "Cats",
      "question": "Do cats really have 9 lives?"
    },
    {
      "name": "Tigers",
      "question": "Where do tigers live?"
    },
    {
      "name": "Pandas",
      "question": "WHat do pandas eat?"
    }   ] }
```

Import your JSON File in your project

You can perform this simple function to print out your JSON file

```
    func jsonParsingMethod() {
        //get the file
        let filePath = Bundle.main.path(forResource: "animals", ofType: "json")
        let content = try! String(contentsOfFile: filePath!)

        let data: Data = content.data(using: String.Encoding.utf8)!
        let json: NSDictionary =  try! JSONSerialization.jsonObject(with: data as Data,
options:.mutableContainers) as! NSDictionary

//Call which part of the file you'd like to pare
        if let results = json["mammals"] as? [[String: AnyObject]] {

            for res in results {
                //this will print out the names of the mammals from out file.
                if let rates = res["name"] as? String {
                    print(rates)
                }
            }
        }
    }
```

If you want to put it in a table view, I would create a dictionary first with an NSObject.

Create a new swift file called `ParsingObject` and create your string variables.

> Make sure that the variable name is the same as the JSON File

例如，在我们的项目中有name和question，所以在新的swift文件中，我们将使用

```
var name: String?
var question: String?
```

将我们创建的NSObject初始化回ViewController.swift中 var array = ParsingObject 然后我们将执行之前相同的方法，只做少量修改。

```
    func jsonParsingMethod() {
        //获取文件
        let filePath = Bundle.main.path(forResource: "animals", ofType: "json")
        let content = try! String(contentsOfFile: filePath!)

        let data: Data = content.data(using: String.Encoding.utf8)!
        let json: NSDictionary =  try! JSONSerialization.jsonObject(with: data as Data,
options:.mutableContainers) as! NSDictionary

//这次我们获取海洋动物
        let results = json["Sea Animals"] as? [[String: AnyObject]]

//使用for循环获取所有内容
        for i in 0 ..< results!.count {

//获取值
            let dict = results?[i]
            let resultsArray = ParsingObject()

//将值追加到我们的NSObject文件中
          resultsArray.setValuesForKeys(dict!)
          array.append(resultsArray)

        }

    }
```

然后我们通过以下方式在表视图中显示它，

```
    func tableView(_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int {
        return array.count
    }

    func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) ->
UITableViewCell {
        let cell = tableView.dequeueReusableCell(withIdentifier: "Cell", for: indexPath)
//这里存储我们的值
        let object = array[indexPath.row]
cell.textLabel?.text = object.name
cell.detailTextLabel?.text = object.question
        return cell
    }
```

第24.5节：将简单的JSON解析为自定义对象

即使第三方库很好，协议也提供了一种简单的方式来解析JSON。你可以想象你有一个对象Todo，如下所示

```
struct Todo {
    let comment: String
```


. For example, in our project we have `name` and `question` so in our new swift file, we will use

```
var name: String?
var question: String?
```

Initialize the NSObject we made back into our ViewController.swift var array = ParsingObject Then we would perform the same method we had before with a minor modification.

```
    func jsonParsingMethod() {
        //get the file
        let filePath = Bundle.main.path(forResource: "animals", ofType: "json")
        let content = try! String(contentsOfFile: filePath!)

        let data: Data = content.data(using: String.Encoding.utf8)!
        let json: NSDictionary =  try! JSONSerialization.jsonObject(with: data as Data,
options:.mutableContainers) as! NSDictionary

//This time let's get Sea Animals
        let results = json["Sea Animals"] as? [[String: AnyObject]]

//Get all the stuff using a for-loop
        for i in 0 ..< results!.count {

//get the value
            let dict = results?[i]
            let resultsArray = ParsingObject()

//append the value to our NSObject file
            resultsArray.setValuesForKeys(dict!)
            array.append(resultsArray)

        }

    }
```

Then we show it in our tableview by doing this,

```
    func tableView(_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int {
        return array.count
    }

    func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) ->
UITableViewCell {
        let cell = tableView.dequeueReusableCell(withIdentifier: "Cell", for: indexPath)
//This is where our values are stored
        let object = array[indexPath.row]
        cell.textLabel?.text = object.name
        cell.detailTextLabel?.text = object.question
        return cell
    }
```

Section 24.5: Simple JSON parsing into custom objects

Even if third-party libraries are good, a simple way to parse the JSON is provided by protocols You can imagine you have got an object `Todo` as

```
struct Todo {
    let comment: String
```

```
}
```

每当你接收到JSON时，你可以像另一个示例中那样使用普通的NSData进行处理 NSJSONSerialization对象。

之后，使用一个简单的协议JSONDecodable

```
typealias JSONDictionary = [String:AnyObject]
protocol JSONDecodable {
    associatedtype Element
    static func from(json json: JSONDictionary) -> Element?
}
```

让你的Todo结构体遵循JSONDecodable协议即可实现功能

```
extension Todo: JSONDecodable {
    static func from(json json: JSONDictionary) -> Todo? {
        guard let comment = json["comment"] as? String else { return nil }
        return Todo(comment: comment)
    }
}
```

你可以用这个 JSON 代码试试：

```
{
    "todos": [
        {
            "comment" : "待办事项评论"
        }
    ]
}
```

当你从 API 获取到它后，可以像之前的示例那样将其序列化为一个 AnyObject 实例。之后，你可以检查该实例是否是一个 JSONDictionary 实例

```
guard let jsonDictionary = dictionary as? JSONDictionary else { return }
```

另一个需要检查的点，针对本例中特殊情况，因为 JSON 中有一个 Todo 数组，是 todos dictionary

```
guard let todosDictionary = jsonDictionary["todos"] as? [JSONDictionary] else { return }
```

既然你已经得到了字典数组，就可以使用 flatMap 将它们逐个转换成 Todo 对象（它会自动删除数组中的 nil 值）

```
let todos: [Todo] = todosDictionary.flatMap { Todo.from(json: $0) }
```

第24.6节：箭头

Arrow 是一个优雅的 Swift JSON 解析库。

它允许解析 JSON 并借助一个

<--

操作符：


```
}
```

Whenever you receive the JSON, you can handle the plain `NSData` as shown in the other example using `NSJSONSerialization` object.

After that, using a simple protocol `JSONDecodable`

```
typealias JSONDictionary = [String:AnyObject]
protocol JSONDecodable {
    associatedtype Element
    static func from(json json: JSONDictionary) -> Element?
}
```

And making your `Todo` struct conforming to `JSONDecodable` does the trick

```
extension Todo: JSONDecodable {
    static func from(json json: JSONDictionary) -> Todo? {
        guard let comment = json["comment"] as? String else { return nil }
        return Todo(comment: comment)
    }
}
```

You can try it with this json code:

```
{
    "todos": [
        {
            "comment" : "The todo comment"
        }
    ]
}
```

When you got it from the API, you can serialize it as the previous examples shown in an `AnyObject` instance. After that, you can check if the instance is a `JSONDictionary` instance

```
guard let jsonDictionary = dictionary as? JSONDictionary else { return }
```

The other thing to check, specific for this case because you have an array of `Todo` in the JSON, is the `todos` dictionary

```
guard let todosDictionary = jsonDictionary["todos"] as? [JSONDictionary] else { return }
```

Now that you got the array of dictionaries, you can convert each of them in a `Todo` object by using `flatMap` (it will automatically delete the `nil` values from the array)

```
let todos: [Todo] = todosDictionary.flatMap { Todo.from(json: $0) }
```

Section 24.6: Arrow

Arrow is an elegant JSON parsing library in Swift.

It allows to parse JSON and map it to custom model classes with help of an

<--

operator:

```
identifier <-- json["id"]
name <-- json["name"]
stats <-- json["stats"]
```

示例：

Swift 模型

```
struct Profile {
    var identifier = 0
    var name = ""
    var link: NSURL?
    var weekday: WeekDay = .Monday
    var stats = Stats()
    var phoneNumbers = [PhoneNumber]()
}
```

JSON 文件

```
{
    "id": 15678,
    "name": "约翰·多伊",
    "link": "https://apple.com/steve",
    "weekdayInt" : 3,
    "stats": {
        "numberOfFriends": 163,
        "numberOfFans": 10987
    },
    "phoneNumbers": [{
                      "label": "house",
                      "number": "9809876545"
                      }, {
                      "label": "cell",
                      "number": "0908070656"
                      }, {
                      "label": "work",
                      "number": "0916570656"
    }]
}
```

映射

```
扩展 个人资料：ArrowParsable {
    变异函数 反序列化(json: JSON) {
        标识符 <-- json["id"]
        链接 <-- json["link"]
        名称 <-- json["name"]
        星期几 <-- json["weekdayInt"]
        统计 <- json["stats"]
电话号码 <-- json["phoneNumbers"]
    }
}
```

用法

```
让 个人资料 = 个人资料()
个人资料.反序列化(json)
```

安装：


```
identifier <-- json["id"]
name <-- json["name"]
stats <-- json["stats"]
```

Example:

Swift model

```
struct Profile {
    var identifier = 0
    var name = ""
    var link: NSURL?
    var weekday: WeekDay = .Monday
    var stats = Stats()
    var phoneNumbers = [PhoneNumber]()
}
```

JSON file

```
{
    "id": 15678,
    "name": "John Doe",
    "link": "https://apple.com/steve",
    "weekdayInt" : 3,
    "stats": {
        "numberOfFriends": 163,
        "numberOfFans": 10987
    },
    "phoneNumbers": [{
                      "label": "house",
                      "number": "9809876545"
                      }, {
                      "label": "cell",
                      "number": "0908070656"
                      }, {
                      "label": "work",
                      "number": "0916570656"
    }]
}
```

Mapping

```
extension Profile: ArrowParsable {
    mutating func deserialize(json: JSON) {
        identifier <-- json["id"]
        link <-- json["link"]
        name <-- json["name"]
        weekday <-- json["weekdayInt"]
        stats <- json["stats"]
        phoneNumbers <-- json["phoneNumbers"]
    }
}
```

Usage

```
let profile = Profile()
profile.deserialize(json)
```

Installation:

卡萨奇

```
github "s4cha/Arrow"
```

CocoaPods

```
pod 'Arrow'
use_frameworks!
```

手动

只需将 Arrow.swift 复制并粘贴到你的 Xcode 项目中

https://github.com/s4cha/Arrow

作为框架

从GitHub 仓库下载 Arrow，并在示例项目中构建 Framework 目标。然后链接该框架。

Carthage

```
github "s4cha/Arrow"
```

CocoaPods

```
pod 'Arrow'
use_frameworks!
```

Manually

Simply Copy and Paste Arrow.swift in your Xcode Project

https://github.com/s4cha/Arrow

As A Framework

Download Arrow from the GitHub repository and build the Framework target on the example project. Then Link against this framework.

第25章：高级运算符

第25.1节：按位运算符

Swift 按位运算符允许你对数字的二进制形式进行操作。你可以通过在数字前加上0b来指定二进制字面量，例如0b110等同于二进制数110（十进制数6）。每个1或0都是数字中的一位（bit）。

按位非 ~：

```
var number: UInt8 = 0b01101100
let newNumber = ~number
// newNumber 等于 0b01101100
```

这里，每个位都会变成它的相反值。将数字显式声明为UInt8确保该数字是正数（这样我们在示例中就不必处理负数），并且它只有8位。如果0b01101100是更大的UInt，那么前导的0会被转换为1，并在取反时变得有意义：

```
var number: UInt16 = 0b01101100
// number 等于 0b0000000001101100
// 前面的 0 不重要
let newNumber = ~number
// newNumber 等于 0b1111111110010011
// 现在 1 是重要的
```

- 0 -> 1
- 1 -> 0

按位与 &:

```
var number = 0b0110
let newNumber = number & 0b1010
// newNumber 等于 0b0010
```

这里，只有当 & 运算符两边的二进制数在该位都为 1 时，该位才为 1。

- 0 & 0 -> 0
- 0 & 1 -> 0
- 1 & 1 -> 1

按位或 |:

```
var number = 0b0110
let newNumber = number | 0b1000
// newNumber 等于 0b1110
```

这里，只有当 | 运算符任一侧的二进制数在该位为 1 时，该位才为 1。

- 0 | 0 -> 0
- 0 | 1 -> 1
- 1 | 1 -> 1

按位异或（排他或）^：

Chapter 25: Advanced Operators

Section 25.1: Bitwise Operators

Swift Bitwise operators allow you to perform operations on the binary form of numbers. You can specify a binary literal by prefixing the number with `0b`, so for example `0b110` is equivalent to the binary number 110 (the decimal number 6). Each 1 or 0 is a bit in the number.

Bitwise NOT ~:

```
var number: UInt8 = 0b01101100
let newNumber = ~number
// newNumber is equal to 0b01101100
```

Here, each bit get changed to its opposite. Declaring the number as explicitly `UInt8` ensures that the number is positive (so that we don't have to deal with negatives in the example) and that it is only 8 bits. If `0b01101100` was a larger UInt, there would be leading 0s that would be converted to 1s and become significant upon inversion:

```
var number: UInt16 = 0b01101100
// number equals 0b0000000001101100
// the 0s are not significant
let newNumber = ~number
// newNumber equals 0b1111111110010011
// the 1s are now significant
```

- 0 -> 1
- 1 -> 0

Bitwise AND &:

```
var number = 0b0110
let newNumber = number & 0b1010
// newNumber is equal to 0b0010
```

Here, a given bit will be 1 if and only if the binary numbers on both sides of the & operator contained a 1 at that bit location.

- 0 & 0 -> 0
- 0 & 1 -> 0
- 1 & 1 -> 1

Bitwise OR |:

```
var number = 0b0110
let newNumber = number | 0b1000
// newNumber is equal to 0b1110
```

Here, a given bit will be 1 if and only if the binary number on at least one side of the | operator contained a 1 at that bit location.

- 0 | 0 -> 0
- 0 | 1 -> 1
- 1 | 1 -> 1

Bitwise XOR (Exclusive OR) ^:

```
var number = 0b0110
let newNumber = number ^ 0b1010
// newNumber 等于 0b1100
```

这里，只有当两个操作数在该位上的位不同，该位才为1。

- 0 ^ 0 -> 0
- 0 ^ 1 -> 1
- 1 ^ 1 -> 0

对于所有二进制操作，操作数的顺序对结果没有影响。

第25.2节：自定义运算符

Swift 支持创建自定义运算符。新运算符在全局级别使用`operator`关键字声明。

运算符的结构由三部分定义：操作数位置、优先级和结合性。

1. prefix、infix 和 postfix 修饰符用于开始自定义运算符声明。 prefix 和后缀修饰符声明运算符必须分别位于其作用的值的前面或后面。此类运算符是单目运算符，如8和3++**，因为它们只能作用于一个目标。中缀声明二元运算符，作用于其之间的两个值，例如2+3。

2. 优先级较高的操作符先被计算。默认的操作符优先级仅略高于?...:（Swift 2.x 中的值为 100）。标准 Swift 运算符的优先级可在此处找到。

3. 结合性定义了同一优先级运算符之间的运算顺序。左结合运算符从左到右计算（阅读顺序，像大多数运算符一样），而右结合运算符则从右到左计算。

版本 ≥ 3.0

从 Swift 3.0 开始，应该在**precedence group**中定义优先级和结合性，而不是在操作符本身中定义，这样多个操作符可以轻松共享相同的优先级，而无需引用难以理解的数字。标准优先级组的列表如下所示。

运算符根据计算代码返回值。该代码作为一个普通函数，参数指定输入类型，return关键字指定运算符返回的计算值。

这里是一个简单指数运算符的定义，因为标准Swift没有指数运算符。

```
import Foundation

中缀运算符 ** { 结合性 左 优先级 170 }

函数 ** (num: Double, power: Double) -> Double{
    return pow(num, power)
}
```

中缀操作符表示**操作符作用于两个值之间，例如9**2。由于该函数具有左结合性，3**3**2被计算为(3**3)**2。数字170的优先级高于所有标准Swift操作，这意味着尽管**具有左结合性，表达式3+2**4的计算结果为19。

版本 ≥ 3.0


```
var number = 0b0110
let newNumber = number ^ 0b1010
// newNumber is equal to 0b1100
```

Here, a given bit will be 1 if and only if the bits in that position of the two operands are different.

- 0 ^ 0 -> 0
- 0 ^ 1 -> 1
- 1 ^ 1 -> 0

For all binary operations, the order of the operands makes no difference on the result.

Section 25.2: Custom Operators

Swift supports the creation of custom operators. New operators are declared at a global level using the `operator` keyword.

The operator's structure is defined by three parts: operand placement, precedence, and associativity.

1. The `prefix`, `infix` and `postfix` modifiers are used to start an custom operator declaration. The `prefix` and `postfix` modifiers declare whether the operator must be before or after, respectively, the value on which it acts. Such operators are urnary, like `8` and `3++` **, since they can only act on one target. The `infix` declares a binary operator, which acts on the two values it is between, such as `2+3`.

2. Operators with higher **precedence** are calculated first. The default operator precedence is just higher than `?...:` (a value of 100 in Swift 2.x). The precedence of standard Swift operators can be found here.

3. **Associativity** defines the order of operations between operators of the same precedence. Left associative operators are calculated from left to right (reading order, like most operators), while right associative operators calculate from right to left.

Version ≥ 3.0

Starting from Swift 3.0, one would define the precedence and associativity in a **precedence group** instead of the operator itself, so that multiple operators can easily share the same precedence without referring to the cryptic numbers. The list of standard precedence groups is shown below.

Operators return values based on the calculation code. This code acts as a normal function, with parameters specifying the type of input and the `return` keyword specifying the calculated value that the operator returns.

Here is the definition of a simple exponential operator, since standard Swift does not have an exponential operator.

```
import Foundation

infix operator ** { associativity left precedence 170 }

func ** (num: Double, power: Double) -> Double{
    return pow(num, power)
}
```

The `infix` says that the `**` operator works in between two values, such as `9**2`. Because the function has left associativity, `3**3**2` is calculated as `(3**3)**2`. The precedence of `170` is higher than all standard Swift operations, meaning that `3+2**4` calculates to `19`, despite the left associativity of `**`.

Version ≥ 3.0

```
import Foundation

中缀操作符 **: BitwiseShiftPrecedence

func ** (num: Double, power: Double) -> Double {
    return pow(num, power)
}
```

在Swift 3.0中，我们可以使用内置的优先级组BitwiseShiftPrecedence来代替显式指定优先级和结合性，该优先级组赋予正确的值（与<<、>>相同）。

**: 自增和自减操作符已被弃用，并将在Swift 3中移除。

第25.3节：溢出操作符

溢出指的是当一个操作导致的数字超出该数字所能表示的位数范围时发生的情况。

由于二进制算术的工作方式，当数字超过其位数所能表示的最大值时，数字会溢出到该位数能表示的最小值，然后从那里继续递增。同样，当数字变得过小时，会下溢到该位数能表示的最大值，然后从那里继续递减。

由于这种行为通常不被期望且可能导致严重的安全问题，Swift的算术操作符+、-和*在操作可能导致溢出或下溢时会抛出错误。若明确允许溢出和下溢，请使用&+、&-和&*代替。

```
var almostTooLarge = Int.max
almostTooLarge + 1 // 不允许
almostTooLarge &+ 1 // 允许，但结果将是 Int.min 的值
```

第25.4节：交换律运算符

让我们添加一个自定义运算符来乘以 CGSize

```
func *(lhs: CGFloat, rhs: CGSize) -> CGSize{
    let height = lhs*rhs.height
    let width = lhs*rhs.width
    return CGSize(width: width, height: height)
}
```

现在这样可以用了

```
let sizeA = CGSize(height:100, width:200)
let sizeB = 1.1 * sizeA         //=> (height: 110, width: 220)
```

但如果我们尝试反过来做这个操作，就会报错

```
let sizeC = sizeB * 20          // 错误
```

但添加起来很简单：

```
func *(lhs: CGSize, rhs: CGFloat) -> CGSize{
    return rhs*lhs
```



```
import Foundation

infix operator **: BitwiseShiftPrecedence

func ** (num: Double, power: Double) -> Double {
    return pow(num, power)
}
```

Instead of specifying the precedence and associativity explicitly, on Swift 3.0 we could use the built-in precedence group BitwiseShiftPrecedence that gives the correct values (same as `<<`, `>>`).

**: The increment and decrement are deprecated and will be removed in Swift 3.

Section 25.3: Overflow Operators

Overflow refers to what happens when an operation would result in a number that is either larger or smaller than the designated amount of bits for that number may hold.

Due to the way binary arithmetic works, after a number becomes too large for its bits, the number overflows down to the smallest possible number (for the bit size) and then continues counting up from there. Similarly, when a number becomes too small, it underflows up to the largest possible number (for its bit size) and continues counting down from there.

Because this behavior is not often desired and can lead to serious security issues, the Swift arithmetic operators `+`, `-`, and `*` will throw errors when an operation would cause an overflow or underflow. To explicitly allow overflow and underflow, use `&+`, `&-`, and `&*` instead.

```
var almostTooLarge = Int.max
almostTooLarge + 1 // not allowed
almostTooLarge &+ 1 // allowed, but result will be the value of Int.min
```

Section 25.4: Commutative Operators

Let's add a custom operator to multiply a CGSize

```
func *(lhs: CGFloat, rhs: CGSize) -> CGSize{
    let height = lhs*rhs.height
    let width = lhs*rhs.width
    return CGSize(width: width, height: height)
}
```

Now this works

```
let sizeA = CGSize(height:100, width:200)
let sizeB = 1.1 * sizeA         //=> (height: 110, width: 220)
```

But if we try to do the operation in reverse, we get an error

```
let sizeC = sizeB * 20          // ERROR
```

But it's simple enough to add:

```
func *(lhs: CGSize, rhs: CGFloat) -> CGSize{
    return rhs*lhs
```

```
}
```

现在该运算符是交换律的。

```
let sizeA = CGSize(height:100, width:200)
let sizeB = sizeA * 1.1              //=> (height: 110, width: 220)
```

第25.5节：为字典重载 + 运算符

由于目前Swift中没有简单的方法来合并字典，重载+和+=
运算符以使用泛型添加此功能可能会很有用。

```
// 合并两个字典。如果两个字典都包含相同的键，使用右侧字典的值。

func +<K, V>(lhs: [K : V], rhs: [K : V]) -> [K : V] {
    var combined = lhs
    for (key, value) in rhs {
       combined[key] = value
    }
    return combined
}

// + 运算符的可变版本，允许字典"就地"追加。

func +=<K, V>(inout lhs: [K : V], rhs: [K : V]) {
    for (key, value) in rhs {
lhs[key] = value
    }
}
```

版本 ≥ 3.0

从 Swift 3 开始，inout 应该放在参数类型之前。

```
func +=<K, V>(lhs: inout [K : V], rhs: [K : V]) { ... }
```

示例用法：

```
let firstDict = ["hello" : "world"]
let secondDict = ["world" : "hello"]
var thirdDict = firstDict + secondDict // ["hello": "world", "world": "hello"]

thirdDict += ["hello":"bar", "baz":"qux"] // ["hello": "bar", "baz": "qux", "world": "hello"]
```

第25.6节：标准 Swift 运算符的优先级

优先级更高（绑定更紧）的运算符列在前面。

运算符	优先级组（≥3.0）	优先级	结合性
.		∞	左结合
?, !, ++, --, [], (), {}	（后缀）		
!, ~, +, -, ++, --	（前缀）		
~> （Swift ≤2.3）		255	左结合
<<, >>	按位移位优先级	160	无
, /, %, &, &	乘法优先级	150	左结合
+, -, |, ^, &+, &-	加法优先级	140	左结合


```
}
```

Now the operator is commutative.

```
let sizeA = CGSize(height:100, width:200)
let sizeB = sizeA * 1.1              //=> (height: 110, width: 220)
```

Section 25.5: Overloading + for Dictionaries

As there is currently no simple way of combining dictionaries in Swift, it can be useful to overload the + and += operators in order to add this functionality using generics.

```
// Combines two dictionaries together. If both dictionaries contain
// the same key, the value of the right hand side dictionary is used.
func +<K, V>(lhs: [K : V], rhs: [K : V]) -> [K : V] {
    var combined = lhs
    for (key, value) in rhs {
        combined[key] = value
    }
    return combined
}

// The mutable variant of the + overload, allowing a dictionary
// to be appended to 'in-place'.
func +=<K, V>(inout lhs: [K : V], rhs: [K : V]) {
    for (key, value) in rhs {
        lhs[key] = value
    }
}
```

Version ≥ 3.0

As of Swift 3, inout should be placed before the argument type.

```
func +=<K, V>(lhs: inout [K : V], rhs: [K : V]) { ... }
```

Example usage:

```
let firstDict = ["hello" : "world"]
let secondDict = ["world" : "hello"]
var thirdDict = firstDict + secondDict // ["hello": "world", "world": "hello"]

thirdDict += ["hello":"bar", "baz":"qux"] // ["hello": "bar", "baz": "qux", "world": "hello"]
```

Section 25.6: Precedence of standard Swift operators

Operators that bound tighter (higher precedence) are listed first.

Operators	Precedence group (≥3.0)	Precedence	Associativity
.		∞	left
?, !, ++, --, [], (), {}	(postfix)		
!, ~, +, -, ++, --	(prefix)		
~> (swift ≤2.3)		255	left
<<, >>	BitwiseShiftPrecedence	160	none
, /, %, &, &	MultiplicationPrecedence	150	left
+, -, |, ^, &+, &-	AdditionPrecedence	140	left

..., ..<	范围形成优先级	135	无
is, as, as?, as!	类型转换优先级	132	左结合
??	空合并优先级	131	右
<, <=, >, >=, ==, !=, ===, !==, ~=	比较优先级	130	无
&&	逻辑与优先级	120	左结合
||	逻辑或优先级	110	左结合
	默认优先级*		无
?...:	三元运算符优先级	100	右
=, +=, -=, *=, /=, %=, <<=, >>=, &=, |=, ^=	赋值优先级	90	右结合，赋值
->	函数箭头优先级		右

版本 ≥ 3.0

- 默认优先级组（DefaultPrecedence）高于三元运算符优先级（TernaryPrecedence），但与其余运算符无序。除了该组外，其余优先级是线性的。

- 该表也可以在Apple的API参考文档中找到
- 优先级组的实际定义可以在GitHub上的源代码中找到

..., ..<	RangeFormationPrecedence	135	none
is, as, as?, as!	CastingPrecedence	132	left
??	NilCoalescingPrecedence	131	right
<, <=, >, >=, ==, !=, ===, !==, ~=	ComparisonPrecedence	130	none
&&	LogicalConjunctionPrecedence	120	left
||	LogicalDisjunctionPrecedence	110	left
	DefaultPrecedence*		none
?... :	TernaryPrecedence	100	right
=, +=, -=, *=, /=, %=, <<=, >>=, &=, |=, ^=	AssignmentPrecedence	90	right, assignment
->	FunctionArrowPrecedence		right

Version ≥ 3.0

- The DefaultPrecedence precedence group is higher than TernaryPrecedence, but is unordered with the rest of the operators. Other than this group, the rest of the precedences are linear.

- This table can be also be found on Apple's API reference
- The actual definition of the precedence groups can be found in the source code on GitHub

第26章：方法交换（Method Swizzling）

第26.1节：扩展UIViewController并交换viewDidLoad方法

在 Objective-C 中，方法交换（method swizzling）是更改现有选择器实现的过程。这是可能的，因为选择器在调度表上被映射，调度表是指向函数或方法的指针表。

纯 Swift 方法不会被 Objective-C 运行时动态分发，但我们仍然可以在任何继承自NSObject的类上利用这些技巧。

在这里，我们将扩展UIViewController并交换viewDidLoad方法以添加一些自定义日志记录：

```
extension UIViewController {

    // 我们不能像在 Objective-C 中那样重写 load，因此改为重写 initialize
    public override static func initialize() {

        // 创建一个静态结构体用于我们的调度令牌，以确保内存中只有一个实例
        struct Static {
            static var token: dispatch_once_t = 0
        }

        // 将其包装在 dispatch_once 块中，确保只执行一次
        dispatch_once(&Static.token) {
            // 获取原始选择器和方法实现，并与我们的新方法交换

            let originalSelector = #selector(UIViewController.viewDidLoad)
            let swizzledSelector = #selector(UIViewController.myViewDidLoad)

            let originalMethod = class_getInstanceMethod(self, originalSelector)
            let swizzledMethod = class_getInstanceMethod(self, swizzledSelector)

            let didAddMethod = class_addMethod(self, originalSelector,
method_getImplementation(swizzledMethod), method_getTypeEncoding(swizzledMethod))

            // class_addMethod 如果使用不当或指针无效可能会失败，因此需要检查
确保我们成功将方法添加到查找表中
            if didAddMethod {
class_replaceMethod(self, swizzledSelector,
method_getImplementation(originalMethod), method_getTypeEncoding(originalMethod))
            } else {
method_exchangeImplementations(originalMethod, swizzledMethod);
            }
        }
    }

    // 我们的新 viewDidLoad 函数
    // 在此示例中，我们只是记录函数名，但这可以用来运行
任何自定义代码
    func myViewDidLoad() {
        // 这不是递归，因为我们在 initialize() 中交换了选择器。
        // 我们不能在扩展中调用 super。
        self.myViewDidLoad()
        print(#function) // 记录 myViewDidLoad()
    }
}
```


Chapter 26: Method Swizzling

Section 26.1: Extending UIViewController and Swizzling viewDidLoad

In Objective-C, method swizzling is the process of changing the implementation of an existing selector. This is possible due to the way Selectors are mapped on a dispatch table, or a table of pointers to functions or methods.

Pure Swift methods are not dynamically dispatched by the Objective-C runtime, but we can still take advantage of these tricks on any class that inherits from `NSObject`.

Here, we will extend `UIViewController` and swizzle `viewDidLoad` to add some custom logging:

```
extension UIViewController {

    // We cannot override load like we could in Objective-C, so override initialize instead
    public override static func initialize() {

        // Make a static struct for our dispatch token so only one exists in memory
        struct Static {
            static var token: dispatch_once_t = 0
        }

        // Wrap this in a dispatch_once block so it is only run once
        dispatch_once(&Static.token) {
            // Get the original selectors and method implementations, and swap them with our new
method
            let originalSelector = #selector(UIViewController.viewDidLoad)
            let swizzledSelector = #selector(UIViewController.myViewDidLoad)

            let originalMethod = class_getInstanceMethod(self, originalSelector)
            let swizzledMethod = class_getInstanceMethod(self, swizzledSelector)

            let didAddMethod = class_addMethod(self, originalSelector,
method_getImplementation(swizzledMethod), method_getTypeEncoding(swizzledMethod))

            // class_addMethod can fail if used incorrectly or with invalid pointers, so check to
make sure we were able to add the method to the lookup table successfully
            if didAddMethod {
                class_replaceMethod(self, swizzledSelector,
method_getImplementation(originalMethod), method_getTypeEncoding(originalMethod))
            } else {
                method_exchangeImplementations(originalMethod, swizzledMethod);
            }
        }
    }

    // Our new viewDidLoad function
    // In this example, we are just logging the name of the function, but this can be used to run
any custom code
    func myViewDidLoad() {
        // This is not recursive since we swapped the Selectors in initialize().
        // We cannot call super in an extension.
        self.myViewDidLoad()
        print(#function) // logs myViewDidLoad()
    }
}
```

第26.2节：Swift方法交换基础

让我们交换TestSwizzling类中methodOne()和methodTwo()的实现：

```
class TestSwizzling : NSObject {
    dynamic func methodOne()->Int{
        return 1
    }
}

extension TestSwizzling {

    // 在Objective-C中你会在load()中执行方法交换，
    // 但Swift中不允许使用此方法
    override class func initialize()
    {

        struct Inner {
            static let i: () = {

                let originalSelector = #selector(TestSwizzling.methodOne)
                let swizzledSelector = #selector(TestSwizzling.methodTwo)
                let originalMethod = class_getInstanceMethod(TestSwizzling.self, originalSelector);
                let swizzledMethod = class_getInstanceMethod(TestSwizzling.self, swizzledSelector)

method_exchangeImplementations(originalMethod, swizzledMethod)
            }
        }
        let _ = Inner.i
    }

    func methodTwo()->Int{
        // 经过方法交换后，这将不再是递归调用
        return methodTwo()+1
    }
}

var c = TestSwizzling()
print(c.methodOne())
print(c.methodTwo())
```

第26.3节：方法交换基础 - Objective-C

UIView的initWithFrame:方法的Objective-C方法交换示例

```
static IMP original_initWithFrame;

+ (void)swizzleMethods {
    static BOOL swizzled = NO;
    if (!swizzled) {
        swizzled = YES;

        Method initWithFrameMethod =
class_getInstanceMethod([UIView class], @selector(initWithFrame:));
        original_initWithFrame = method_setImplementation(
initWithFrame方法，(IMP)replacement_initWithFrame);
    }
}

static id replacement_initWithFrame(id self, SEL _cmd, CGRect rect) {
```


Section 26.2: Basics of Swift Swizzling

Let's swap the implementation of `methodOne()` and `methodTwo()` in our `TestSwizzling` class:

```
class TestSwizzling : NSObject {
    dynamic func methodOne()->Int{
        return 1
    }
}

extension TestSwizzling {

    //In Objective-C you'd perform the swizzling in load(),
    //but this method is not permitted in Swift
    override class func initialize()
    {

        struct Inner {
            static let i: () = {

                let originalSelector = #selector(TestSwizzling.methodOne)
                let swizzledSelector = #selector(TestSwizzling.methodTwo)
                let originalMethod = class_getInstanceMethod(TestSwizzling.self, originalSelector);
                let swizzledMethod = class_getInstanceMethod(TestSwizzling.self, swizzledSelector)

                method_exchangeImplementations(originalMethod, swizzledMethod)
            }
        }
        let _ = Inner.i
    }

    func methodTwo()->Int{
        // It will not be a recursive call anymore after the swizzling
        return methodTwo()+1
    }
}

var c = TestSwizzling()
print(c.methodOne())
print(c.methodTwo())
```

Section 26.3: Basics of Swizzling - Objective-C

Objective-C example of swizzling UIView's `initWithFrame:` method

```
static IMP original_initWithFrame;

+ (void)swizzleMethods {
    static BOOL swizzled = NO;
    if (!swizzled) {
        swizzled = YES;

        Method initWithFrameMethod =
            class_getInstanceMethod([UIView class], @selector(initWithFrame:));
        original_initWithFrame = method_setImplementation(
            initWithFrameMethod, (IMP)replacement_initWithFrame);
    }
}

static id replacement_initWithFrame(id self, SEL _cmd, CGRect rect) {
```

```
    // 这将替代UIView上原始的initWithFrame方法被调用
    // 在这里执行你需要的操作...

    // 额外提示：这是调用原始initWithFrame方法的方式
    UIView *view =
        ((id (*)(id, SEL, CGRect))original_initWithFrame)(self, _cmd, rect);

    return view;
}
```



```
    // This will be called instead of the original initWithFrame method on UIView
    // Do here whatever you need...

    // Bonus: This is how you would call the original initWithFrame method
    UIView *view =
        ((id (*)(id, SEL, CGRect))original_initWithFrame)(self, _cmd, rect);

    return view;
}
```

第27章：反射

第27.1节：Mirror的基本用法

创建作为Mirror对象的类

```
class Project {
    var title: String = ""
    var id: Int = 0
    var platform: String = ""
    var version: Int = 0
    var info: String?
}
```

创建一个实例，这个实例将真正成为镜像的主体。在这里你也可以为 Project 类的属性添加值。

```
let sampleProject = Project()
sampleProject.title = "MirrorMirror"
sampleProject.id = 199
sampleProject.platform = "iOS"
sampleProject.version = 2
sampleProject.info = "Reflection 测试应用"
```

下面的代码展示了 Mirror 实例的创建。mirror 的 children 属性是一个 AnyForwardCollection<Child> 其中 Child 是主题属性和值的类型别名元组。 Child 具有一个 label: 字符串 和 值：任何。

```
let projectMirror = Mirror(reflecting: sampleProject)
let properties = projectMirror.children

print(properties.count)        //5
print(properties.first?.label) //Optional("title")
print(properties.first!.value) //MirrorMirror
print()

for property in properties {
    print("\(property.label!):\(property.value)")
}
```

在Xcode的Playground或控制台中，上述for循环的输出。

```
title:MirrorMirror
id:199
platform:iOS
version:2
info:Optional("test app for Reflection")
```

在Xcode 8 beta 2的Playground中测试

第27.2节：获取类的属性类型和名称，无需实例化

使用Swift类Mirror可以提取name、value和type（Swift 3：type(of: value)，Swift 2：某个类的实例的属性的 value.dynamicType)

Chapter 27: Reflection

Section 27.1: Basic Usage for Mirror

Creating the class to be the subject of the Mirror

```
class Project {
    var title: String = ""
    var id: Int = 0
    var platform: String = ""
    var version: Int = 0
    var info: String?
}
```

Creating an instance that will actually be the subject of the mirror. Also here you can add values to the properties of the Project class.

```
let sampleProject = Project()
sampleProject.title = "MirrorMirror"
sampleProject.id = 199
sampleProject.platform = "iOS"
sampleProject.version = 2
sampleProject.info = "test app for Reflection"
```

The code below shows the creating of Mirror instance. The children property of the mirror is a `AnyForwardCollection<Child>` where `Child` is typealias tuple for subject's property and value. `Child` had a `label: String` and `value: Any`.

```
let projectMirror = Mirror(reflecting: sampleProject)
let properties = projectMirror.children

print(properties.count)        //5
print(properties.first?.label) //Optional("title")
print(properties.first!.value) //MirrorMirror
print()

for property in properties {
    print("\(property.label!):\(property.value)")
}
```

Output in Playground or Console in Xcode for the for loop above.

```
title:MirrorMirror
id:199
platform:iOS
version:2
info:Optional("test app for Reflection")
```

Tested in Playground on Xcode 8 beta 2

Section 27.2: Getting type and names of properties for a class without having to instantiate it

Using the Swift class `Mirror` works if you want to extract *name*, *value* and *type* (Swift 3: `type(of: value)`, Swift 2: `value.dynamicType`) of properties for an **instance** of a certain class.

如果你的类继承自NSObject，你可以使用方法class_copyPropertyList结合property_getAttributes来查找类的属性的名称和类型——无需拥有该类的实例。我在Github上创建了一个项目用于此，但这里是代码：

```
func getTypesOfProperties(in clazz: NSObject.Type) -> Dictionary<String, Any>? {
    var count = UInt32()
guard let properties = class_copyPropertyList(clazz, &count) else { return nil }
    var types: Dictionary<String, Any> = [:]
    for i in 0..<Int(count) {
guard let property: objc_property_t = properties[i], let name = getNameOf(property:
property) else { continue }
        let type = getTypeOf(property: property)
       types[name] = type
    }
    free(properties)
    return types
}

func getTypeOf(property: objc_property_t) -> Any {
    guard let attributesAsNSString: NSString = NSString(utf8String:
property_getAttributes(property)) 否则 { 返回 Any.self }
    让 attributes = attributesAsNSString 作为 String
    让 slices = attributes.components(以 "\""分割)
   保证 slices.数量 > 1 否则 { 返回 getPrimitiveDataType(withAttributes: attributes) }
    让 objectClassName = slices[1]
    让 objectClass = NSClassFromString(objectClassName) 作为! NSObject.类型
    返回 objectClass
}

   函数 getPrimitiveDataType(withAttributes attributes: String) -> Any {
       保证 让 letter = attributes.子字符串(从: 1, 到: 2), 让 type =
primitiveDataTypes[letter] 否则 { 返回 Any.self }
       返回 type
    }
```

其中 primitiveDataTypes 是一个字典，将属性字符串中的字母映射到一个值类型：

```
让 primitiveDataTypes: Dictionary<String, Any> = [
    "c" : Int8.self,
    "s" : Int16.self,
    "i" : Int32.self,
    "q" : Int.self, //也包括: Int64, NSInteger，仅在64位平台上有效
    "S" : UInt16.self,
    "I" : UInt32.self,
    "Q" : UInt.self, // 仅在64位平台上为UInt64，且为真
    "B" : Bool.self,
    "d" : Double.self,
    "f" : Float.self,
    "{" : Decimal.self
]

   func getNameOf(property: objc_property_t) -> String? {
       guard let name: NSString = NSString(utf8String: property_getName(property)) else { return
nil }
       return name as String
    }
```

它可以提取所有继承自NSObject的类类型属性的NSObject.Type，例如NSDate（Swift3中为：Date）、NSString（Swift3中为String?）和NSNumber，但它存储为类型Any（如你所见，该方法返回的字典中值的类型）。这是由于值类型（如Int、Int32、

If you class inherits from `NSObject`, you can use the method `class_copyPropertyList` together with `property_getAttributes` to find out the *name* and *types* of properties for a class - **without having an instance of it**. I created a project on Github for this, but here is the code:

```
func getTypesOfProperties(in clazz: NSObject.Type) -> Dictionary<String, Any>? {
    var count = UInt32()
    guard let properties = class_copyPropertyList(clazz, &count) else { return nil }
    var types: Dictionary<String, Any> = [:]
    for i in 0..<Int(count) {
        guard let property: objc_property_t = properties[i], let name = getNameOf(property:
property) else { continue }
        let type = getTypeOf(property: property)
        types[name] = type
    }
    free(properties)
    return types
}

func getTypeOf(property: objc_property_t) -> Any {
    guard let attributesAsNSString: NSString = NSString(utf8String:
property_getAttributes(property)) else { return Any.self }
    let attributes = attributesAsNSString as String
    let slices = attributes.components(separatedBy: "\"")
    guard slices.count > 1 else { return getPrimitiveDataType(withAttributes: attributes) }
    let objectClassName = slices[1]
    let objectClass = NSClassFromString(objectClassName) as! NSObject.Type
    return objectClass
}

   func getPrimitiveDataType(withAttributes attributes: String) -> Any {
       guard let letter = attributes.substring(from: 1, to: 2), let type =
primitiveDataTypes[letter] else { return Any.self }
       return type
    }
```

Where `primitiveDataTypes` is a Dictionary mapping a letter in the attribute string to a value type:

```
let primitiveDataTypes: Dictionary<String, Any> = [
    "c" : Int8.self,
    "s" : Int16.self,
    "i" : Int32.self,
    "q" : Int.self, //also: Int64, NSInteger, only true on 64 bit platforms
    "S" : UInt16.self,
    "I" : UInt32.self,
    "Q" : UInt.self, //also UInt64, only true on 64 bit platforms
    "B" : Bool.self,
    "d" : Double.self,
    "f" : Float.self,
    "{" : Decimal.self
]

   func getNameOf(property: objc_property_t) -> String? {
       guard let name: NSString = NSString(utf8String: property_getName(property)) else { return
nil }
       return name as String
    }
```

It can extract the `NSObject.Type` of all properties which class type inherits from `NSObject` such as `NSDate` (Swift3: `Date`), `NSString`(Swift3: `String`?) and `NSNumber`, however it is store in the type `Any` (as you can see as the type of the value of the Dictionary returned by the method). This is due to the limitations of `value types` such as Int, Int32,

Bool）存在限制。由于这些类型不继承自NSObject，调用例如Int的.self——Int.self不会返回NSObject.Type，而是返回类型Any。因此该方法返回Dictionary<String，Any>?而非Dictionary<String，NSObject.Type>?。

你可以这样使用该方法：

```
class Book: NSObject {
    let title: String
    let author: String?
    let numberOfPages: Int
    let released: Date
    let isPocket: Bool

    init(title: String, author: String?, numberOfPages: Int, released: Date, isPocket: Bool) {
        self.title = title
        self.author = author
        self.numberOfPages = numberOfPages
        self.released = released
        self.isPocket = isPocket
    }
}

guard let types = getTypesOfProperties(in: Book.self) else { return }
for (name, type) in types {
    print("'\(name)' has type '\(type)'")
}
// Prints:
// 'title' has type 'NSString'
// 'numberOfPages' has type 'Int'
// 'author' has type 'NSString'
// 'released' has type 'NSDate'
// 'isPocket' has type 'Bool'
```

你也可以尝试将Any转换为NSObject.Type，这对于所有继承自NSObject的属性都会成功，然后你可以使用标准的==运算符来检查类型：

```
func checkPropertiesOfBook() {
guard let types = getTypesOfProperties(in: Book.self) else { return }
    for (name, type) in types {
        if let objectType = type as? NSObject.Type {
            if objectType == NSDate.self {
                print("属性名为 '\(name)' 的类型是 'NSDate'")
            } else if objectType == NSString.self {
                print("属性名为 '\(name)' 的类型是 'NSString'")
            }
        }
    }
}
```

如果你声明了这个自定义的 == 操作符：

```
func ==(rhs: Any, lhs: Any) -> Bool {
    let rhsType: String = "\(rhs)"
    let lhsType: String = "\(lhs)"
    let same = rhsType == lhsType
    return same
}
```

你甚至可以这样检查值类型的类型：

Bool. Since those types do not inherit from NSObject, calling `.self` on e.g. an Int - `Int.self` does not return NSObject.Type, but rather the type `Any`. Thus the method returns `Dictionary<String, Any>?` and not `Dictionary<String, NSObject.Type>?`.

You can use this method like this:

```
class Book: NSObject {
    let title: String
    let author: String?
    let numberOfPages: Int
    let released: Date
    let isPocket: Bool

    init(title: String, author: String?, numberOfPages: Int, released: Date, isPocket: Bool) {
        self.title = title
        self.author = author
        self.numberOfPages = numberOfPages
        self.released = released
        self.isPocket = isPocket
    }
}

guard let types = getTypesOfProperties(in: Book.self) else { return }
for (name, type) in types {
    print("'\(name)' has type '\(type)'")
}
// Prints:
// 'title' has type 'NSString'
// 'numberOfPages' has type 'Int'
// 'author' has type 'NSString'
// 'released' has type 'NSDate'
// 'isPocket' has type 'Bool'
```

You can also try to cast the `Any` to `NSObject.Type`, which will succeed for all properties inheriting from `NSObject`, then you can check the type using standard == operator:

```
func checkPropertiesOfBook() {
    guard let types = getTypesOfProperties(in: Book.self) else { return }
    for (name, type) in types {
        if let objectType = type as? NSObject.Type {
            if objectType == NSDate.self {
                print("Property named '\(name)' has type 'NSDate'")
            } else if objectType == NSString.self {
                print("Property named '\(name)' has type 'NSString'")
            }
        }
    }
}
```

If you declare this custom == operator:

```
func ==(rhs: Any, lhs: Any) -> Bool {
    let rhsType: String = "\(rhs)"
    let lhsType: String = "\(lhs)"
    let same = rhsType == lhsType
    return same
}
```

You can then even check the type of `value types` like this:

```swift
func checkPropertiesOfBook() {
guard let types = getTypesOfProperties(in: Book.self) else { return }
    for (name, type) in types {
        if type == Int.self {
            print("属性名为 '\(name)' 的类型是 'Int'")
        } else if type == Bool.self {
            print("属性名为 '\(name)' 的类型是 'Bool'")
        }
    }
}
```

限制 当值类型是可选项时，此解决方案无效。如果你在NSObject子类中声明了如下属性：var myOptionalInt: Int?，上述代码将找不到该属性，因为方法 class_copyPropertyList 不包含可选值类型。


```swift
func checkPropertiesOfBook() {
    guard let types = getTypesOfProperties(in: Book.self) else { return }
    for (name, type) in types {
        if type == Int.self {
            print("Property named '\(name)' has type 'Int'")
        } else if type == Bool.self {
            print("Property named '\(name)' has type 'Bool'")
        }
    }
}
```

LIMITATIONS This solution does not work when `value types` are optionals. If you have declared a property in you NSObject subclass like this: `var myOptionalInt: Int?`, the code above won't find that property because the method `class_copyPropertyList` does not contain optional value types.

第28章：访问控制

第28.1节：使用结构体的基本示例

版本 ≥ 3.0

在Swift 3中有多种访问级别。此示例使用了除open之外的所有级别：

```
public struct 车 {

    public let 品牌: String
    let 型号: String //可选关键字：默认是“internal”
    private let 全名: String
fileprivate var 其他名称: String

    public init(_ 品牌: String, 型号: String) {
        self.品牌 = 品牌
        self.型号 = 型号
        self.fullName = "\(make)\(model)"
        self.otherName = "\(model) - \(make)"
    }
}
```

假设myCar 是这样初始化的：

```
let myCar = Car("Apple", model: "iCar")
```

Car.make（公开）

```
print(myCar.make)
```

这条打印语句在任何地方都能运行，包括导入了Car的目标。

Car.model（内部）

```
print(myCar.model)
```

如果代码和Car在同一个目标中，这条语句可以编译通过。

Car.otherName（文件私有）

```
print(myCar.otherName)
```

这只有在代码与Car在同一个文件中时才有效。

Car.fullName（私有）

```
print(myCar.fullName)
```

这在 Swift 3 中不起作用。private属性只能在同一个struct/class内访问。

```
public struct 车 {

    public let make: String       //public
    let model: String             //internal
    private let fullName: String! //private

    public init(_ make: String, model model: String) {
        self.make = make
```


Chapter 28: Access Control

Section 28.1: Basic Example using a Struct

Version ≥ 3.0

In Swift 3 there are multiple access-levels. This example uses them all except for `open`:

```
public struct Car {

    public let make: String
    let model: String //Optional keyword: will automatically be "internal"
    private let fullName: String
    fileprivate var otherName: String

    public init(_ make: String, model: String) {
        self.make = make
        self.model = model
        self.fullName = "\(make)\(model)"
        self.otherName = "\(model) - \(make)"
    }
}
```

Assume `myCar` was initialized like this:

```
let myCar = Car("Apple", model: "iCar")
```

Car.make (public)

```
print(myCar.make)
```

This print will work everywhere, including targets that import `Car`.

Car.model (internal)

```
print(myCar.model)
```

This will compile if the code is in the same target as `Car`.

Car.otherName (fileprivate)

```
print(myCar.otherName)
```

This will only work if the code is *in the same file* as `Car`.

Car.fullName (private)

```
print(myCar.fullName)
```

This won't work in Swift 3. `private` properties can only be accessed within the same `struct`/`class`.

```
public struct Car {

    public let make: String       //public
    let model: String             //internal
    private let fullName: String! //private

    public init(_ make: String, model model: String) {
        self.make = make
```

```
        self.型号 = 型号
        self.fullName = "\(make)\(model)"
    }
}
```

如果实体有多个相关的访问级别，Swift 会选择最低的访问级别。如果一个私有变量存在于一个公共类中，该变量仍然被视为私有。

第28.2节：子类示例

```
public class SuperClass {
    private func secretMethod() {}
}

internal class SubClass: SuperClass {
    override internal func secretMethod() {
        super.secretMethod()
    }
}
```

第28.3节：Getter和Setter示例

```
结构体 Square {
    私有(设置) 变量 area = 0

    变量 side: 整数 = 0 {
        属性观察器 didSet {
area = side*side
        }
    }
}

公共结构体 Square {
    公共私有(设置) 变量 area = 0
    公共变量 side: 整数 = 0 {
        didSet {
area = side*side
        }
    }
    公共初始化() {}
}
```



```
        self.model = model
        self.fullName = "\(make)\(model)"
    }
}
```

If the entity has multiple associated access levels, Swift looks for the lowest level of access. If a private variable exists in a public class, the variable will still be considered private.

Section 28.2: Subclassing Example

```
public class SuperClass {
    private func secretMethod() {}
}

internal class SubClass: SuperClass {
    override internal func secretMethod() {
        super.secretMethod()
    }
}
```

Section 28.3: Getters and Setters Example

```
struct Square {
    private(set) var area = 0

    var side: Int = 0 {
        didSet {
            area = side*side
        }
    }
}

public struct Square {
    public private(set) var area = 0
    public var side: Int = 0 {
        didSet {
            area = side*side
        }
    }
    public init() {}
}
```

第29章：闭包

第29.1节：闭包基础

闭包（也称为代码块或lambda表达式）是可以在程序中存储和传递的代码片段。

```
让 sayHi = { 打印("Hello") }
// sayHi的类型是"() -> ()"，也称为"() -> Void"

sayHi()  // 输出"Hello"
```

像其他函数一样，闭包可以接受参数并返回结果或抛出错误：

```
let addInts = { (x: Int, y: Int) -> Int in
    return x + y
}
// addInts 的类型是 "(Int, Int) -> Int"

let result = addInts(1, 2)  // result 是 3

let divideInts = { (x: Int, y: Int) throws -> Int in
    if y == 0 {
throw MyErrors.DivisionByZero
    }
    return x / y
}
// divideInts 的类型是 "(Int, Int) throws -> Int"
```

闭包可以捕获它们作用域中的值：

```
// 这个函数返回另一个返回整数的函数
func makeProducer(x: Int) -> (() -> Int) {
    let closure = { x }  // x 被闭包捕获
    return closure
}

// 这两个函数调用使用完全相同的代码，
// 但每个闭包捕获了不同的值。
let three = makeProducer(3)
let four = makeProducer(4)
three()  // 返回 3
four()  // 返回4
```

闭包可以直接传递给函数：

```
let squares = (1...10).map({ $0 * $0 })  // 返回 [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
let squares = (1...10).map { $0 * $0 }

NSURLSession.sharedSession().dataTaskWithURL(myURL,
    completionHandler: { (data: NSData?, response: NSURLResponse?, error: NSError?) in
        if let data = data {
            print("请求成功，数据: \(data)")
        } else {
            print("请求失败: \(error)")
        }
    }).resume()
```


Chapter 29: Closures

Section 29.1: Closure basics

Closures (also known as **blocks** or **lambdas**) are pieces of code which can be stored and passed around within your program.

```
let sayHi = { print("Hello") }
// The type of sayHi is "() -> ()", aka "() -> Void"

sayHi()  // prints "Hello"
```

Like other functions, closures can accept arguments and return results or throw errors:

```
let addInts = { (x: Int, y: Int) -> Int in
    return x + y
}
// The type of addInts is "(Int, Int) -> Int"

let result = addInts(1, 2)  // result is 3

let divideInts = { (x: Int, y: Int) throws -> Int in
    if y == 0 {
        throw MyErrors.DivisionByZero
    }
    return x / y
}
// The type of divideInts is "(Int, Int) throws -> Int"
```

Closures can **capture** values from their scope:

```
// This function returns another function which returns an integer
func makeProducer(x: Int) -> (() -> Int) {
    let closure = { x }  // x is captured by the closure
    return closure
}

// These two function calls use the exact same code,
// but each closure has captured different values.
let three = makeProducer(3)
let four = makeProducer(4)
three()  // returns 3
four()  // returns 4
```

Closures can be passed directly into functions:

```
let squares = (1...10).map({ $0 * $0 })  // returns [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
let squares = (1...10).map { $0 * $0 }

NSURLSession.sharedSession().dataTaskWithURL(myURL,
    completionHandler: { (data: NSData?, response: NSURLResponse?, error: NSError?) in
        if let data = data {
            print("Request succeeded, data: \(data)")
        } else {
            print("Request failed: \(error)")
        }
    }).resume()
```

第29.2节：语法变体

基本的闭包语法是

> *{ [捕获列表] (参数) throws-ness -> 返回类型 in 函数体 }.*

这些部分中的许多可以省略，因此有几种等效的方式来编写简单的闭包：

```
let addOne = { [] (x: Int) -> Int in return x + 1 }
let addOne = { [] (x: Int) -> Int in x + 1 }
let addOne = { (x: Int) -> Int in x + 1 }
let addOne = { x -> Int in x + 1 }
let addOne = { x in x + 1 }
let addOne = { $0 + 1 }

let addOneOrThrow = { [] (x: Int) throws -> Int in return x + 1 }
let addOneOrThrow = { [] (x: Int) throws -> Int in x + 1 }
let addOneOrThrow = { (x: Int) throws -> Int in x + 1 }
let addOneOrThrow = { x throws -> Int in x + 1 }
let addOneOrThrow = { x throws in x + 1 }
```

- 如果捕获列表为空，可以省略。
- 如果参数类型可以推断，则不需要类型注解。
- 如果返回类型可以推断，则不需要指定返回类型。
- 参数不必命名；可以用$0、$1、$2等来引用。
- 如果闭包只包含一个表达式，且该表达式的值需要被返回，则可以省略return关键字。

- 如果闭包被推断为会抛出错误，或者写在期望抛出错误闭包的上下文中，或者不抛出错误，则throws可以省略。

```
// 闭包的类型未知，因此我们必须指定x和y的类型。
// 输出类型被推断为Int，因为Int的+运算符返回Int。
let addInts = { (x: Int, y: Int) in x + y }

// 闭包的类型已指定，因此我们可以省略参数的类型注解。
let addInts: (Int, Int) -> Int = { x, y in x + y }
let addInts: (Int, Int) -> Int = { $0 + $1 }
```

第29.3节：将闭包传递给函数

函数可以接受闭包（或其他函数）作为参数：

```
func foo(value: Double, block: () -> Void) { ... }
func foo(value: Double, block: Int -> Int) { ... }
func foo(value: Double, block: (Int, Int) -> String) { ... }
```

尾随闭包语法

如果函数的最后一个参数是闭包，闭包的大括号{/}可以写在函数调用之后：

```
foo(3.5, block: { print("Hello") })

foo(3.5) { print("Hello") }

dispatch_async(dispatch_get_main_queue(), {
```


Section 29.2: Syntax variations

The basic closure syntax is

> { [*capture list*] (*parameters*) *throws-ness* `->` *return type* `in` *body* }.

Many of these parts can be omitted, so there are several equivalent ways to write simple closures:

```
let addOne = { [] (x: Int) -> Int in return x + 1 }
let addOne = { [] (x: Int) -> Int in x + 1 }
let addOne = { (x: Int) -> Int in x + 1 }
let addOne = { x -> Int in x + 1 }
let addOne = { x in x + 1 }
let addOne = { $0 + 1 }

let addOneOrThrow = { [] (x: Int) throws -> Int in return x + 1 }
let addOneOrThrow = { [] (x: Int) throws -> Int in x + 1 }
let addOneOrThrow = { (x: Int) throws -> Int in x + 1 }
let addOneOrThrow = { x throws -> Int in x + 1 }
let addOneOrThrow = { x throws in x + 1 }
```

- The capture list can be omitted if it's empty.
- Parameters don't need type annotations if their types can be inferred.
- The return type doesn't need to be specified if it can be inferred.
- Parameters don't have to be named; instead they can be referred to with `$0`, `$1`, `$2`, etc.
- If the closure contains a single expression, whose value is to be returned, the `return` keyword can be omitted.
- If the closure is inferred to throw an error, is written in a context which expects a throwing closure, or doesn't throw an error, `throws` can be omitted.

```
// The closure's type is unknown, so we have to specify the type of x and y.
// The output type is inferred to be Int, because the + operator for Ints returns Int.
let addInts = { (x: Int, y: Int) in x + y }

// The closure's type is specified, so we can omit the parameters' type annotations.
let addInts: (Int, Int) -> Int = { x, y in x + y }
let addInts: (Int, Int) -> Int = { $0 + $1 }
```

Section 29.3: Passing closures into functions

Functions may accept closures (or other functions) as parameters:

```
func foo(value: Double, block: () -> Void) { ... }
func foo(value: Double, block: Int -> Int) { ... }
func foo(value: Double, block: (Int, Int) -> String) { ... }
```

Trailing closure syntax

If a function's last parameter is a closure, the closure braces {/} may be written **after** the function invocation:

```
foo(3.5, block: { print("Hello") })

foo(3.5) { print("Hello") }

dispatch_async(dispatch_get_main_queue(), {
```

```
    print("来自主队列的问候")
})

dispatch_async(dispatch_get_main_queue()) {
    print("来自主队列的问候")
}
```

如果函数的唯一参数是闭包，使用尾随闭包语法调用时，也可以省略一对圆括号()：

```
func bar(block: () -> Void) { ... }
```

```
bar() { print("Hello") }

bar { print("Hello") }
```

@noescape 参数

标记为@noescape的闭包参数保证在函数调用返回之前执行，因此在闭包体内不需要使用self.

func executeNow(scape block: () -> Void) { // 由于 `block` 是 @noescape，存储到外部变量是非法的 // 我们只能在这里调用它。 block() } func executeLater(block: () -> Void) { dispatch_async(dispatch_get_main_queue()) { // 未来某个时间... block() } }

```
class MyClass {
    var x = 0
    func showExamples() {
        // 错误：闭包中引用属性 'x' 需要显式使用 'self.' 来明确捕获语义

executeLater { x = 1 }

executeLater { self.x = 2 }  // 正确，闭包显式捕获了 self

        // 这里不需要 “self.”，因为 executeNow() 接受的是 @noescape 闭包。
executeNow { x = 3 }

        // 再次说明，不需要使用 self.，因为 map() 使用了 @noescape。
        [1, 2, 3].map { $0 + x }
    }
}
```

Swift 3 注意事项：

请注意，在 Swift 3 中，不再需要将闭包标记为 @noescape。闭包默认不会逃逸。在 Swift 3 中，不是将闭包标记为非逃逸，而是使用“@escaping”关键字标记作为函数参数的逃逸闭包。

throws 和 rethrows

闭包和其他函数一样，可能会抛出错误：

func executeNowOrIgnoreError(block: () ws -> Void) { do { try block() } catch { print("error: \(error)") } }

函数当然也可以将错误传递给调用者：

func executeNowOrThrow(block: () ws -> Void) throws { try block() }


```
    print("Hello from the main queue")
})

dispatch_async(dispatch_get_main_queue()) {
    print("Hello from the main queue")
}
```

If a function's only argument is a closure, you may also omit the pair of parentheses () when calling it with the trailing closure syntax:

```
func bar(block: () -> Void) { ... }
```

```
bar() { print("Hello") }

bar { print("Hello") }
```

@noescape parameters

Closure parameters marked @noescape are guaranteed to execute before the function call returns, so using self. is not required inside the closure body:

func executeNow(scape block: () -> Void) { // Since `block` is @noescape, it's illegal to store it to an external variable. // We can only call it right here. block() } func executeLater(block: () -> Void) { dispatch_async(dispatch_get_main_queue()) { // Some time in the future... block() } }

```
class MyClass {
    var x = 0
    func showExamples() {
        // error: reference to property 'x' in closure requires explicit 'self.' to make capture
semantics explicit
        executeLater { x = 1 }

        executeLater { self.x = 2 }  // ok, the closure explicitly captures self

        // Here "self." is not required, because executeNow() takes a @noescape block.
        executeNow { x = 3 }

        // Again, self. is not required, because map() uses @noescape.
        [1, 2, 3].map { $0 + x }
    }
}
```

Swift 3 note:

Note that in Swift 3, you no longer mark blocks as @noescape. Blocks are now **not** escaping by default. In Swift 3, instead of marking a closure as non-escaping, you mark a function parameter that is an escaping closure as escaping using the "@escaping" keyword.

throws and rethrows

Closures, like other functions, may throw errors:

func executeNowOrIgnoreError(block: () ws -> Void) { do { try block() } catch { print("error: \(error)") } }

The function may, of course, pass the error along to its caller:

func executeNowOrThrow(block: () ws -> Void) **throws** { try block() }

但是，如果传入的闭包 不抛出错误，调用者仍然需要处理一个会抛出错误的函数：

```
// 这很烦人，因为这需要使用“try”，而“print()”是不会抛出错误的！
尝试 executeNowOrThrow { print("只是打印，没有错误！") }
```

解决方案是rethrows，表示该函数只有在其闭包参数抛出错误时才会抛出错误：

`func executeNowOrRethrow(block: () ws -> Void) ***rethrows*** { try block() } // 这里不需要“try”，因为 block 不会抛出错误。 executeNowOrRethrow { print("此闭包不会抛出错误") } // 该闭包 **可能**抛出错误，因此需要“try”。 try executeNowOrRethrow { throw MyError.Example }`

许多标准库函数使用rethrows，包括map()、filter()和indexOf()。

第29.4节：捕获、强引用/弱引用和保留循环

```
class MyClass {
    func sayHi() { print("Hello") }
    deinit { print("Goodbye") }
}
```

当闭包捕获引用类型（类实例）时，默认持有强引用：

```
let closure: () -> Void
do {
    let obj = MyClass()
    // 捕获对`obj`的强引用：只要闭包本身存活，对象就会被保持活跃

closure = { obj.sayHi() }
closure()  // 对象仍然存在；打印 "Hello"
} // obj 超出作用域
closure()  // 对象仍然存在；打印 "Hello"
```

闭包的 capture list 可以用来指定弱引用或无主引用：

```
let closure: () -> Void
do {
    let obj = MyClass()
    // 捕获对 `obj` 的弱引用：闭包不会保持对象存活；
    // 对象在闭包内部变为可选类型。
closure = { [weak obj] in obj?.sayHi() }
    closure()  // 对象仍然存在；打印 "Hello"
} // obj 超出作用域并被释放；打印 "Goodbye"
closure()  // 闭包内部的 `obj` 为 nil；不会打印任何内容。
```

```
let closure: () -> Void
do {
    let obj = MyClass()
    // 捕获对 `obj` 的无主引用：闭包不会保持对象存活；
    // 假设对象在闭包存活期间始终可访问。
closure = { [unowned obj] in obj.sayHi() }
    closure()  // 对象仍然存在；打印 "Hello"
} // obj 超出作用域并被释放；打印 "Goodbye"
closure()  // 崩溃！访问已释放的 obj。
```

更多信息请参见内存管理主题，以及《Swift 编程语言》的 自动引用计数 部分。

However, if the block passed in *doesn't* throw, the caller is still stuck with a throwing function:

```
// It's annoying that this requires "try", because "print()" can't throw!
try executeNowOrThrow { print("Just printing, no errors here!") }
```

The solution is **`rethrows`**, which designates that the function can only throw **if its closure parameter throws**:

`func executeNowOrRethrow(block: () ws -> Void) ***rethrows*** { try block() } // "try" is not required here, because the block can't throw an error. executeNowOrRethrow { print("No errors are thrown from this closure") } // This block **can** throw an error, so "try" is required. try executeNowOrRethrow { throw MyError.Example }`

Many standard library functions use `rethrows`, including `map()`, `filter()`, and `indexOf()`.

Section 29.4: Captures, strong/weak references, and retain cycles

```
class MyClass {
    func sayHi() { print("Hello") }
    deinit { print("Goodbye") }
}
```

When a closure captures a reference type (a class instance), it holds a strong reference by default:

```
let closure: () -> Void
do {
    let obj = MyClass()
    // Captures a strong reference to `obj`: the object will be kept alive
    // as long as the closure itself is alive.
    closure = { obj.sayHi() }
    closure()  // The object is still alive; prints "Hello"
} // obj goes out of scope
closure()  // The object is still alive; prints "Hello"
```

The closure's **capture list** can be used to specify a weak or unowned reference:

```
let closure: () -> Void
do {
    let obj = MyClass()
    // Captures a weak reference to `obj`: the closure will not keep the object alive;
    // the object becomes optional inside the closure.
    closure = { [weak obj] in obj?.sayHi() }
    closure()  // The object is still alive; prints "Hello"
} // obj goes out of scope and is deallocated; prints "Goodbye"
closure()  // `obj` is nil from inside the closure; this does not print anything.
```

```
let closure: () -> Void
do {
    let obj = MyClass()
    // Captures an unowned reference to `obj`: the closure will not keep the object alive;
    // the object is always assumed to be accessible while the closure is alive.
    closure = { [unowned obj] in obj.sayHi() }
    closure()  // The object is still alive; prints "Hello"
} // obj goes out of scope and is deallocated; prints "Goodbye"
closure()  // crash! obj is being accessed after it's deallocated.
```

For more information, see the Memory Management topic, and the Automatic Reference Counting section of The Swift Programming Language.

保留循环

如果一个对象持有一个闭包，而该闭包又持有对该对象的强引用，这就是一个保留循环。除非打破循环，否则存储对象和闭包的内存将泄漏（永远不会被回收）。

```
class 游戏 {
    var score = 0
    let controller: GCController
    init(controller: GCController) {
        self.controller = controller

        // 错误示范：该闭包强引用了 self，而 self 又强引用了 controller
        // （从而闭包），形成了循环引用。
        self.controller.controllerPausedHandler = {
            let curScore = self.score
            print("暂停按钮被按下；当前分数：\(curScore)")
        }

        // 解决方案：使用 `weak self` 来打破循环引用。
        self.controller.controllerPausedHandler = { [weak self] in
            guard let strongSelf = self else { return }
            let curScore = strongSelf.score
            print("暂停按钮被按下；当前分数：\(curScore)")
        }
    }
}
```

第29.5节：使用闭包进行异步编程

闭包经常用于异步任务，例如从网站获取数据时。

版本 < 3.0

```
func getData(urlString: String, callback: (result: NSData?) -> Void) {

    // 将URL字符串转换为NSURLRequest。
guard let url = NSURL(string: urlString) else { return }
    let request = NSURLRequest(URL: url)

    // 异步从给定的URL获取数据。
    let task = NSURLSession.sharedSession().dataTaskWithRequest(request) {(data: NSData?, response:
NSURLResponse?, error: NSError?) in

        // 我们现在已经得到了来自网站的NSData响应。
        // 我们可以通过使用作为参数传入此函数的回调函数
        // 将数据“传出”此函数。

callback(result: data)
    }

task.resume()
}
```

此函数是异步的，因此不会阻塞调用它的线程（如果在GUI应用程序的主线程上调用，不会导致界面冻结）。

版本 < 3.0

```
print("1. 正在调用 getData")

getData("http://www.example.com") {(result: NSData?) -> Void in
```


Retain cycles

If an object holds onto a closure, which also holds a strong reference to the object, this is a **retain cycle**. Unless the cycle is broken, the memory storing the object and closure will be leaked (never reclaimed).

```
class Game {
    var score = 0
    let controller: GCController
    init(controller: GCController) {
        self.controller = controller

        // BAD: the block captures self strongly, but self holds the controller
        // (and thus the block) strongly, which is a cycle.
        self.controller.controllerPausedHandler = {
            let curScore = self.score
            print("Pause button pressed; current score: \(curScore)")
        }

        // SOLUTION: use `weak self` to break the cycle.
        self.controller.controllerPausedHandler = { [weak self] in
            guard let strongSelf = self else { return }
            let curScore = strongSelf.score
            print("Pause button pressed; current score: \(curScore)")
        }
    }
}
```

Section 29.5: Using closures for asynchronous coding

Closures are often used for asynchronous tasks, for example when fetching data from a website.

Version < 3.0

```
func getData(urlString: String, callback: (result: NSData?) -> Void) {

    // Turn the URL string into an NSURLRequest.
    guard let url = NSURL(string: urlString) else { return }
    let request = NSURLRequest(URL: url)

    // Asynchronously fetch data from the given URL.
    let task = NSURLSession.sharedSession().dataTaskWithRequest(request) {(data: NSData?, response:
NSURLResponse?, error: NSError?) in

        // We now have the NSData response from the website.
        // We can get it "out" of the function by using the callback
        // that was passed to this function as a parameter.

        callback(result: data)
    }

    task.resume()
}
```

This function is asynchronous, so will not block the thread it is being called on (it won't freeze the interface if called on the main thread of your GUI application).

Version < 3.0

```
print("1. Going to call getData")

getData("http://www.example.com") {(result: NSData?) -> Void in
```

```
    // 当从 http://www.example.com 获取数据后调用。
    print("2. 已获取数据")
}

print("3. 已调用 getData")
```

因为任务是异步的，输出通常会是这样：

```
"1. 准备调用 getData"
"3. 已调用 getData"
"2. 已获取数据"
```

因为闭包内的代码print("2. 已获取数据")，只有在从 URL 获取到数据后才会被调用。

第29.6节：闭包和类型别名

闭包可以用typealias来定义。如果同一个闭包签名在多个地方使用，这提供了一个方便的类型占位符。例如，常见的网络请求回调或用户界面事件处理程序非常适合用类型别名“命名”。

public typealias ureType = (x: Int, y: Int) -> Int

然后你可以使用该类型别名定义一个函数：

```
public func closureFunction(closure ureType) { let z = closure(1, 2) } closureFunction() { (x: Int, y: Int) -> Int in return x + y }
```



```
    // Called when the data from http://www.example.com has been fetched.
    print("2. Fetched data")
}

print("3. Called getData")
```

Because the task is asynchronous, the output will usually look like this:

```
"1. Going to call getData"
"3. Called getData"
"2. Fetched data"
```

Because the code inside of the closure, `print("2. Fetched data")`, will not be called until the data from the URL is fetched.

Section 29.6: Closures and Type Alias

A closure can be defined with a `typealias`. This provides a convenient type placeholder if the same closure signature is used in multiple places. For example, common network request callbacks or user interface event handlers make great candidates for being "named" with a type alias.

```
public typealias ureType = (x: Int, y: Int) -> Int
```

You can then define a function using the typealias:

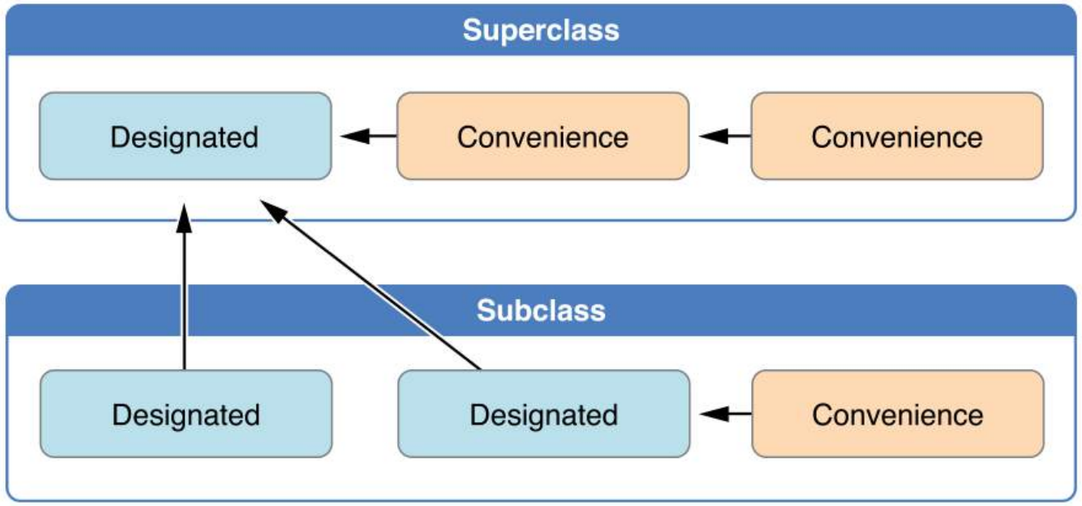
```
public func closureFunction(closure: ureType) { let z = closure(1, 2) } closureFunction() { (x: Int, y: Int) -> Int in return x + y }
```

第30章：初始化器

第30.1节：便利初始化器

Swift类支持多种初始化方式。根据苹果的规范，必须遵守以下3条规则：

1. 指定初始化器必须调用其直接父类的指定初始化器。



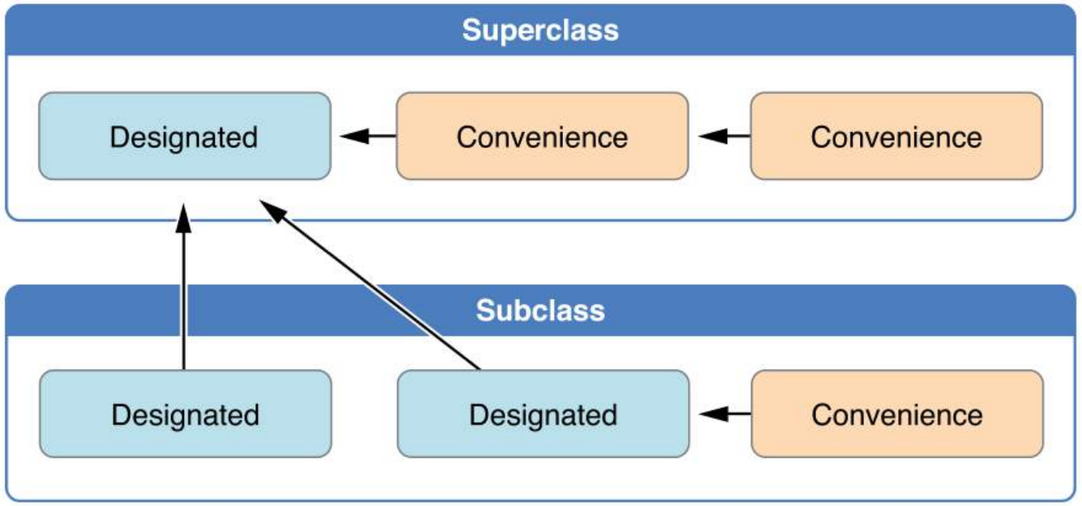
2. 便利初始化器必须调用同一类中的另一个初始化器。
3. 便利初始化器最终必须调用指定初始化器。

Chapter 30: Initializers

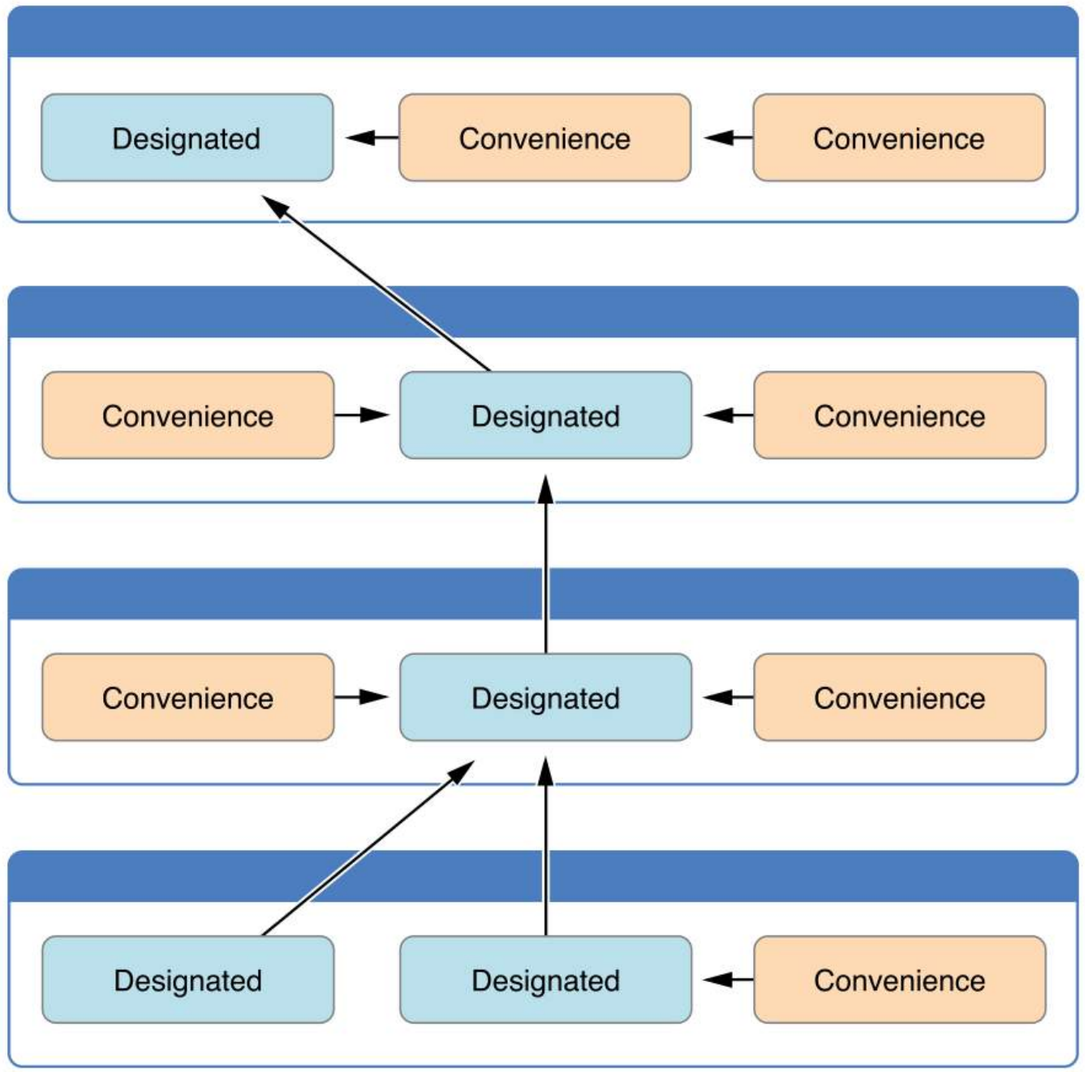
Section 30.1: Convenience init

Swift classes supports having multiple ways of being initialized. Following Apple's specs this 3 rules must be respected:

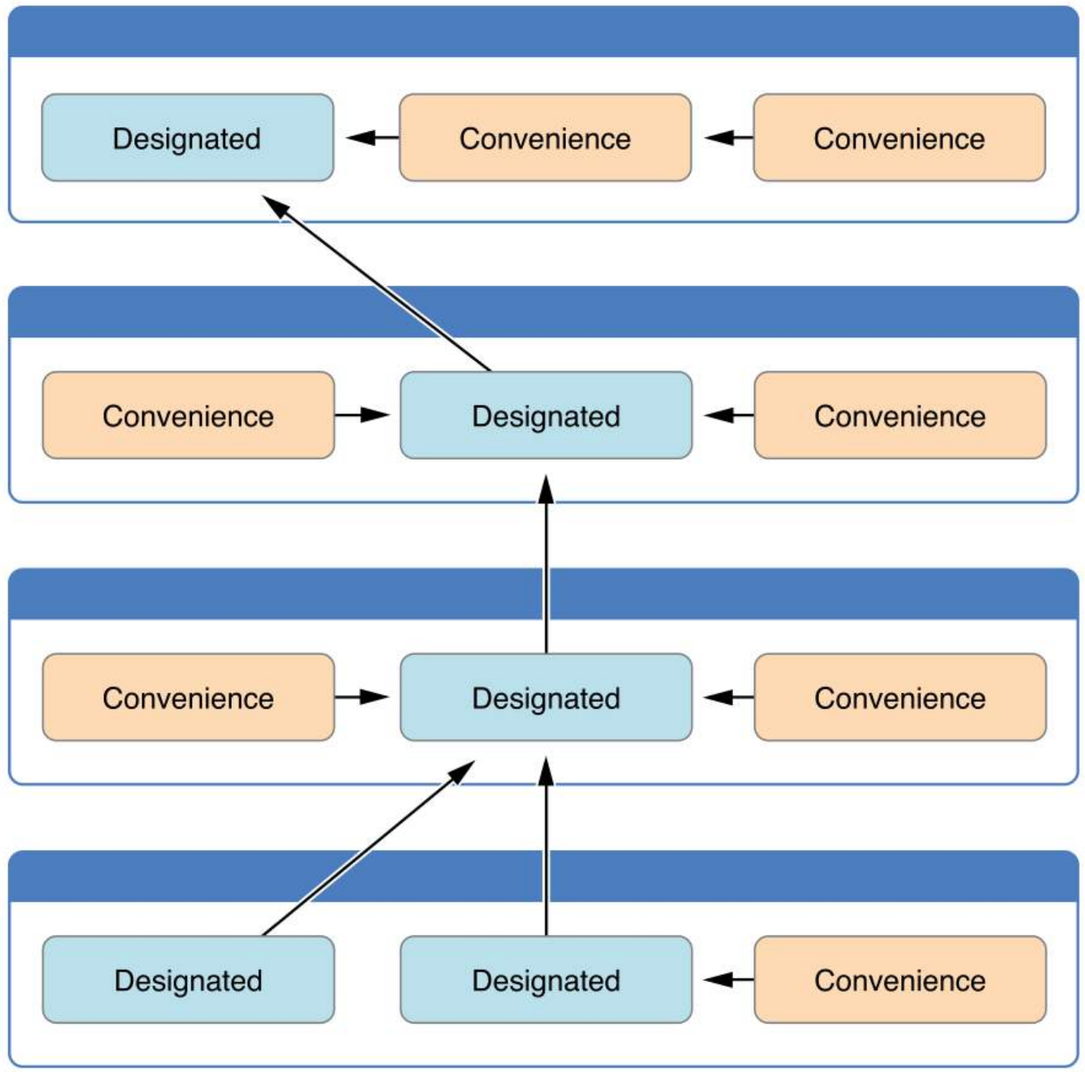
1. A designated initializer must call a designated initializer from its immediate superclass.



2. A convenience initializer must call another initializer from the same class.
3. A convenience initializer must ultimately call a designated initializer.




```
class Foo {

    var someString: String
    var someValue: Int
    var someBool: Bool

    // 指定初始化方法
    init(someString: String, someValue: Int, someBool: Bool)
    {
        self.someString = someString
        self.someValue = someValue
        self.someBool = someBool
    }

    // 便利初始化方法必须调用同一类中的另一个初始化方法。
    convenience init()
    {
        self.init(otherString: "")
    }
```


```
class Foo {

    var someString: String
    var someValue: Int
    var someBool: Bool

    // Designated Initializer
    init(someString: String, someValue: Int, someBool: Bool)
    {
        self.someString = someString
        self.someValue = someValue
        self.someBool = someBool
    }

    // A convenience initializer must call another initializer from the same class.
    convenience init()
    {
        self.init(otherString: "")
    }
```

```
    // 便利初始化方法最终必须调用指定初始化方法。
convenience init(otherString: String)
    {
        self.init(someString: otherString, someValue:  0, someBool: false)
    }
}


类 Baz: Foo
{
    变量 someFloat: Float

    // 指定初始化方法
    初始化(someFloat: Float)
    {
        self.someFloat = someFloat

        // 指定初始化方法必须调用其直接
父类的指定初始化方法。
        super.init(someString: "", someValue: 0, someBool: false)
    }

    // 便利初始化方法必须调用同一类中的另一个初始化方法。
    convenience init()
    {
        self.init(someFloat: 0)
    }
}
```

指定初始化方法

```
让 c = Foo(someString: "Some string", someValue: 10, someBool: true)
```

便利初始化()

```
让 a = Foo()
```

Convenience init(otherString: String)

```
let b = Foo(otherString: "Some string")
```

指定初始化器（将调用父类的指定初始化器）

```
let d = Baz(someFloat: 3)
```

便利初始化()

```
let e = Baz()
```

图片来源：The_Swift_Programming_Language

第30.2节：设置默认属性值

你可以使用初始化器来设置默认属性值：

```
struct Example {
    var upvotes: Int
    init() {
upvotes = 42
    }
```



```
    // A convenience initializer must ultimately call a designated initializer.
convenience init(otherString: String)
    {
        self.init(someString: otherString, someValue:  0, someBool: false)
    }
}


class Baz: Foo
{
    var someFloat: Float

    // Designed initializer
    init(someFloat: Float)
    {
        self.someFloat = someFloat

        // A designated initializer must call a designated initializer from its immediate
superclass.
        super.init(someString: "", someValue: 0, someBool: false)
    }

    // A convenience initializer must call another initializer from the same class.
    convenience init()
    {
        self.init(someFloat: 0)
    }
}
```

Designated Initializer

```
let c = Foo(someString: "Some string", someValue: 10, someBool: true)
```

Convenience init()

```
let a = Foo()
```

Convenience init(otherString: String)

```
let b = Foo(otherString: "Some string")
```

Designated Initializer (will call the superclass Designated Initializer)

```
let d = Baz(someFloat: 3)
```

Convenience init()

```
let e = Baz()
```

Image source: The Swift Programming Language

Section 30.2: Setting default property values

You can use an initializer to set default property values:

```
struct Example {
    var upvotes: Int
    init() {
        upvotes = 42
    }
```

```
}
let myExample = Example() // 调用初始化器
print(myExample.upvotes) // 输出：42
```

或者，将默认属性值作为属性声明的一部分指定：

```
结构体 示例 {
    变量 赞成票 = 42 // 这里推断出类型为 'Int'
}
```

类和结构体 必须 在实例创建时为所有存储属性设置合适的初始值。
此示例无法编译，因为初始化器未为 反对票 提供初始值：

```
结构体 示例 {
    变量 赞成票: 整数
    变量 反对票: 整数
    初始化() {
赞成票 = 0
    } // 错误：初始化器返回时未初始化所有存储属性
}
```

第30.3节：使用参数自定义初始化

```
结构体 公制距离 {
    变量 距离（米）: 双精度浮点数

    init(从厘米: Double) {
        distanceInMeters = centimeters / 100
    }
    init(fromKilometers 公里数: Double) {
        distanceInMeters = 公里数 * 1000
    }
}

let myDistance = MetricDistance(fromCentimeters: 42)
// myDistance.distanceInMeters 是 0.42
let myOtherDistance = MetricDistance(fromKilometers: 42)
// myOtherDistance.distanceInMeters 是 42000
```

注意你不能省略参数标签：

```
let myBadDistance = MetricDistance(42) // 错误：参数标签与任何可用的重载不匹配
```

为了允许省略参数标签，使用下划线 _ 作为标签：

```
struct MetricDistance {
    var distanceInMeters: Double
    init(_ 米数: Double) {
        distanceInMeters = 米数
    }
}
let myDistance = MetricDistance(42) // distanceInMeters = 42
```

如果你的参数标签与一个或多个属性同名，使用 self 明确设置属性值：

```
struct Color {
```



```
}
let myExample = Example() // call the initializer
print(myExample.upvotes) // prints: 42
```

Or, specify default property values as a part of the property's declaration:

```
struct Example {
    var upvotes = 42 // the type 'Int' is inferred here
}
```

Classes and structs **must** set all stored properties to an appropriate initial value by the time an instance is created. This example will not compile, because the initializer did not give an initial value for `downvotes`:

```
struct Example {
    var upvotes: Int
    var downvotes: Int
    init() {
        upvotes = 0
    } // error: Return from initializer without initializing all stored properties
}
```

Section 30.3: Customizing initialization with parameters

```
struct MetricDistance {
    var distanceInMeters: Double

    init(fromCentimeters centimeters: Double) {
        distanceInMeters = centimeters / 100
    }
    init(fromKilometers kilos: Double) {
        distanceInMeters = kilos * 1000
    }
}

let myDistance = MetricDistance(fromCentimeters: 42)
// myDistance.distanceInMeters is 0.42
let myOtherDistance = MetricDistance(fromKilometers: 42)
// myOtherDistance.distanceInMeters is 42000
```

Note that you cannot omit the parameter labels:

```
let myBadDistance = MetricDistance(42) // error: argument labels do not match any available
overloads
```

In order to allow omission of parameter labels, use an underscore _ as the label:

```
struct MetricDistance {
    var distanceInMeters: Double
    init(_ meters: Double) {
        distanceInMeters = meters
    }
}
let myDistance = MetricDistance(42) // distanceInMeters = 42
```

If your argument labels share names with one or more properties, use `self` to explicitly set the property values:

```
struct Color {
```

```
    var 红色, 绿色, 蓝色: Double
    init(红色: Double, 绿色: Double, 蓝色: Double) {
        self.红色 = 红色
        self.绿色 = 绿色
        self.蓝色 = 蓝色
    }
}
```

第30.4节：可抛出初始化器

使用错误处理使结构体（或类）初始化器成为可抛出初始化器：

错误处理枚举示例：

```
enum 验证错误: Error {
    case 无效
}
```

你可以使用错误处理枚举来检查结构体（或类）的参数是否符合预期要求

```
struct 用户 {
    let 名字: String

    init(名字: String?) throws {

guard let 名字 = 名字 else {
          验证错误.无效
        }

        self.name = name
    }
}
```

现在，你可以通过以下方式使用可抛出初始化器：

```
do {
    let user = try User(name: "Sample name")

    // 成功
}
catch ValidationError.invalid {
      // 处理错误
}
```



```
    var red, green, blue: Double
    init(red: Double, green: Double, blue: Double) {
        self.red = red
        self.green = green
        self.blue = blue
    }
}
```

Section 30.4: Throwable Initilizer

Using Error Handling to make Struct(or class) initializer as throwable initializer:

Example Error Handling enum:

```
enum ValidationError: Error {
    case invalid
}
```

You can use Error Handling enum to check the parameter for the Struct(or class) meet expected requirement

```
struct User {
    let name: String

    init(name: String?) throws {

        guard let name = name else {
           ValidationError.invalid
        }

        self.name = name
    }
}
```

Now, you can use throwable initializer by:

```
do {
   let user = try User(name: "Sample name")

   // success
}
catch ValidationError.invalid {
     // handle error
}
```

第31章：关联对象

第31.1节：使用关联对象实现协议扩展中的属性

在Swift中，协议扩展不能拥有真正的属性。

然而，实际上你可以使用“关联对象”技术。结果几乎和“真实”的
属性完全一样。

以下是向协议扩展添加“关联对象”的具体技术：

基本上，你使用Objective-C的“objc_getAssociatedObject”和_set调用。

基本调用如下：

```
获取 {
   return objc_getAssociatedObject(self, & _Handle) as! YourType
   }
set {
objc_setAssociatedObject(self, & _Handle, newValue, .OBJC_ASSOCIATION_RETAIN)
    }
```

这是一个完整的示例。两个关键点是：

1. 在协议中，必须使用“: class”以避免变异问题。
2. 在扩展中，必须使用“where Self:UIViewController”（或其他适当的类）来指定确认类型。

所以，对于示例属性“p”：

```
import Foundation
import UIKit
import ObjectiveC           // 不要忘记这一点

var _Handle: UInt8 = 42     // 它可以是任意值

protocol Able: class {
    var click:UIView? { get set }
    var x:CGFloat? { get set }
    // 注意这里 >> 不要 << 声明 p
}

extension Able where Self:UIViewController {

    var p:YourType { // YourType 可能是一个枚举类型
      get {
           return objc_getAssociatedObject(self, & _Handle) as! YourType
           // 但是，见下文
           }
      set {
objc_setAssociatedObject(self, & _Handle, newValue, .OBJC_ASSOCIATION_RETAIN)
           // 通常，你会想在这里运行某种“setter”方法...
__setter()
           }
    }
```


Chapter 31: Associated Objects

Section 31.1: Property, in a protocol extension, achieved using associated object

In Swift, protocol extensions cannot have true properties.

However, in practice you can use the "associated object" technique. The result is almost exactly like a "real" property.

Here is the exact technique for adding an "associated object" to a protocol extension:

Fundamentally, you use the objective-c "objc_getAssociatedObject" and _set calls.

The basic calls are:

```
get {
   return objc_getAssociatedObject(self, & _Handle) as! YourType
   }
set {
   objc_setAssociatedObject(self, & _Handle, newValue, .OBJC_ASSOCIATION_RETAIN)
    }
```

Here's a full example. The two critical points are:

1. In the protocol, you must use ": class" to avoid the mutation problem.
2. In the extension, you must use "where Self:UIViewController" (or whatever appropriate class) to give the confirming type.

So, for an example property "p":

```
import Foundation
import UIKit
import ObjectiveC           // don't forget this

var _Handle: UInt8 = 42     // it can be any value

protocol Able: class {
    var click:UIView? { get set }
    var x:CGFloat? { get set }
    // note that you >> do not << declare p here
}

extension Able where Self:UIViewController {

    var p:YourType { // YourType might be, say, an Enum
      get {
           return objc_getAssociatedObject(self, & _Handle) as! YourType
           // HOWEVER, SEE BELOW
           }
      set {
           objc_setAssociatedObject(self, & _Handle, newValue, .OBJC_ASSOCIATION_RETAIN)
           // often, you'll want to run some sort of "setter" here...
           __setter()
           }
    }
```

```
    func __setter() { something = p.blah() }

    func someOtherExtensionFunction() { p.blah() }
    // 在其他扩展函数中使用“p”是可以的，
    // 并且你可以在遵循协议的类中任何地方使用 p
}
```

在任何遵循协议的类中，你现在已经“添加”了属性“p”：

你可以像使用遵循类中的任何普通属性一样使用“p”。示例：

```
class Clock:UIViewController, Able {
    var u:Int = 0

    func blah() {
     u = …
      ... = u
      // 像使用任何普通属性一样使用 "p"
      p = ...
      … = p
    }

    override func viewDidLoad() {
      super.viewDidLoad()
pm = .none // "p" 必须在 Clock 的某处被“初始化”
    }
}
```

注意。你必须初始化伪属性。

Xcode 不会强制你在遵循类中初始化 "p"。

你必须初始化 "p"，这点很重要，可能是在遵循类的 viewDidLoad 中进行初始化。

值得记住的是，p 实际上只是一个计算属性。p 实际上只是两个函数，带有语法糖。p 并不存在“变量”本身：编译器不会以任何方式“为 p 分配内存”。因此，期望 Xcode 强制“初始化 p”是没有意义的。

确切地说，你必须记住“第一次使用 p，就像是在初始化它一样”。
（同样，这很可能是在你的 viewDidLoad 代码中。）

关于 getter 本身。

请注意，如果在为“p”设置值之前调用getter，程序将会崩溃。

为避免这种情况，可以考虑如下代码：

```
    获取 {
        let g = objc_getAssociatedObject(self, &_Handle)
        if (g == nil) {
objc_setAssociatedObject(self, &_Handle, _默认初始值_, .OBJC_ASSOCIATION)
            return _默认初始值_
        }
        return objc_getAssociatedObject(self, &_Handle) as! YourType
        }
```

再强调一次。Xcode 不会强制你在遵循的类中初始化p。你必须在遵循类的


```
    func __setter() { something = p.blah() }

    func someOtherExtensionFunction() { p.blah() }
    // it's ok to use "p" inside other extension functions,
    // and you can use p anywhere in the conforming class
}
```

In any conforming class, you have now "added" the property "p":

You can use "p" just as you would use any ordinary property in the conforming class. Example:

```
class Clock:UIViewController, Able {
    var u:Int = 0

    func blah() {
      u = ...
      ... = u
      // use "p" as you would any normal property
      p = ...
      ... = p
    }

    override func viewDidLoad() {
      super.viewDidLoad()
      pm = .none // "p" MUST be "initialized" somewhere in Clock
    }
}
```

Note. You MUST initialize the pseudo-property.

Xcode **will *not* enforce** you initializing "p" in the conforming class.

It is essential that you initialize "p", perhaps in viewDidLoad of the confirming class.

It is worth remembering that p is actually just a ***computed property***. p is actually just two functions, with syntactic sugar. There is no p "variable" anywhere: the compiler does not "assign some memory for p" in any sense. For this reason, it is meaningless to expect Xcode to enforce "initializing p".

Indeed, to speak more accurately, you must remember to "use p for the first time, as if you were initializing it". (Again, that would very likely be in your viewDidLoad code.)

Regarding the getter as such.

Note that it **will crash** if the getter is called before a value for "p" is set.

To avoid that, consider code such as:

```
    get {
        let g = objc_getAssociatedObject(self, &_Handle)
        if (g == nil) {
            objc_setAssociatedObject(self, &_Handle, _default initial value_, .OBJC_ASSOCIATION)
            return _default initial value_
        }
        return objc_getAssociatedObject(self, &_Handle) as! YourType
        }
```

To repeat. Xcode **will *not* enforce** you initializing p in the conforming class. It is essential that you initialize p, say in

viewDidLoad 方法中初始化p。

简化代码……

你可能想使用这两个全局函数：

```
func _aoGet(_ ss: Any!, _ handlePointer: UnsafeRawPointer!, _ safeValue: Any!)->Any! {
    let g = objc_getAssociatedObject(ss, handlePointer)
    if (g == nil) {
objc_setAssociatedObject(ss, handlePointer, safeValue, .OBJC_ASSOCIATION_RETAIN)
        return safeValue
    }
    return objc_getAssociatedObject(ss, handlePointer)
}

func _aoSet(_ ss: Any!, _ handlePointer: UnsafeRawPointer!, _ val: Any!) {
    objc_setAssociatedObject(ss, handlePointer, val, .OBJC_ASSOCIATION_RETAIN)
}
```

请注意，它们除了减少输入和使代码更易读之外，实际上没有任何其他作用。（它们本质上是宏或内联函数。）

你的代码变成了：

```
protocol PMable: class {
    var click:UILabel? { get set } // 这里是普通属性
}

var _pHandle: UInt8 = 321

extension PMable 其中 Self:UIViewController {

    var p:P {
        获取 {
            return _aoGet(self, &_pHandle, P() ) as! P
        }
        set {
_aoSet(self, &_pHandle, newValue)
          __pmSetter()
        }
    }

    func __pmSetter() {
        click!.text = String(p)
    }

    func someFunction() {
p.blah()
    }
}
```

（在 _aoGet 的示例中，P 是可初始化的：你可以用 ""、0 或任何默认值代替 P()。）

viewDidLoad of the conforming class.

Making the code simpler…

You may wish to use these two global functions:

```
func _aoGet(_ ss: Any!, _ handlePointer: UnsafeRawPointer!, _ safeValue: Any!)->Any! {
    let g = objc_getAssociatedObject(ss, handlePointer)
    if (g == nil) {
        objc_setAssociatedObject(ss, handlePointer, safeValue, .OBJC_ASSOCIATION_RETAIN)
        return safeValue
    }
    return objc_getAssociatedObject(ss, handlePointer)
}

func _aoSet(_ ss: Any!, _ handlePointer: UnsafeRawPointer!, _ val: Any!) {
    objc_setAssociatedObject(ss, handlePointer, val, .OBJC_ASSOCIATION_RETAIN)
}
```

Note that they do nothing, whatsoever, other than save typing and make the code more readable. (They are essentially macros or inline functions.)

Your code then becomes:

```
protocol PMable: class {
    var click:UILabel? { get set } // ordinary properties here
}

var _pHandle: UInt8 = 321

extension PMable where Self:UIViewController {

    var p:P {
        get {
            return _aoGet(self, &_pHandle, P() ) as! P
        }
        set {
            _aoSet(self, &_pHandle, newValue)
            __pmSetter()
        }
    }

    func __pmSetter() {
        click!.text = String(p)
    }

    func someFunction() {
        p.blah()
    }
}
```

(In the example at _aoGet, P is initalizable: instead of P() you could use "", 0, or any default value.)

第32章：并发

第32.1节：获取 Grand Central Dispatch（GCD）队列

Grand Central Dispatch 基于"调度队列"的概念。调度队列按传入的顺序执行你指定的任务。
调度队列有三种类型：

- 串行调度队列（也称为私有调度队列）一次执行一个任务，按顺序执行。它们经常用于同步访问资源。
- 并发调度队列（也称为全局调度队列）可以同时执行一个或多个任务。
- 主调度队列在主线程上执行任务。

访问主队列的方法：

版本 = 3.0
```
let mainQueue = DispatchQueue.main
```
版本 < 3.0
```
let mainQueue = dispatch_get_main_queue()
```

系统提供了并发的全局调度队列（对你的应用程序全局可用），具有不同的优先级。你可以使用 Swift 3 中的Dispatch Queue类访问这些队列：

版本 = 3.0
```
let globalConcurrentQueue = DispatchQueue.global(qos: .default)
```

等同于

```
let globalConcurrentQueue = DispatchQueue.global()
```
版本 < 3.0
```
let globalConcurrentQueue = dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0)
```

在 iOS 8 及更高版本中，可以传递的服务质量（Quality of Service）值包括.userInteractive、.userInitiated、.default、.utility 和 .background。这些取代了 DISPATCH_QUEUE_PRIORITY_ 常量。

您也可以创建具有不同优先级的队列：

版本 = 3.0
```
let myConcurrentQueue = DispatchQueue(label: "my-concurrent-queue", qos: .userInitiated,
attributes: [.concurrent], autoreleaseFrequency: .workItem, target: nil)
let mySerialQueue = DispatchQueue(label: "my-serial-queue", qos: .background, attributes: [],
autoreleaseFrequency: .workItem, target: nil)
```
版本 < 3.0
```
let myConcurrentQueue = dispatch_queue_create("my-concurrent-queue", DISPATCH_QUEUE_CONCURRENT)
let mySerialQueue = dispatch_queue_create("my-serial-queue", DISPATCH_QUEUE_SERIAL)
```

在 Swift 3 中，使用此初始化器创建的队列默认是串行的，传递 .workItem 作为自动释放频率确保为每个工作项创建并清空一个自动释放池。还有 .never，表示您将自行管理自动释放池，或者 .inherit，表示继承环境中的设置。在大多数情况下，除非进行极端自定义，否则您可能不会使用 .never。

第 32.2 节：并发循环

GCD 提供了一种执行循环的机制，使得循环相互之间并发执行。

Chapter 32: Concurrency

Section 32.1: Obtaining a Grand Central Dispatch (GCD) queue

Grand Central Dispatch works on the concept of "Dispatch Queues". A dispatch queue executes tasks you designate in the order which they are passed. There are three types of dispatch queues:

- **Serial Dispatch Queues** (aka private dispatch queues) execute one task at a time, in order. They are frequently used to synchronize access to a resource.
- **Concurrent Dispatch Queues** (aka global dispatch queues) execute one or more tasks concurrently.
- The **Main Dispatch Queue** executes tasks on the main thread.

To access the main queue:

Version = 3.0
```
let mainQueue = DispatchQueue.main
```
Version < 3.0
```
let mainQueue = dispatch_get_main_queue()
```

The system provides *concurrent* global dispatch queues (global to your application), with varying priorities. You can access these queues using the DispatchQueue class in Swift 3:

Version = 3.0
```
let globalConcurrentQueue = DispatchQueue.global(qos: .default)
```

equivalent to

```
let globalConcurrentQueue = DispatchQueue.global()
```
Version < 3.0
```
let globalConcurrentQueue = dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0)
```

In iOS 8 or later, the possible quality of service values which may be passed are .userInteractive, .userInitiated, .default, .utility, and .background. These replace the DISPATCH_QUEUE_PRIORITY_ constants.

You can also create your own queues with varying priorities:

Version = 3.0
```
let myConcurrentQueue = DispatchQueue(label: "my-concurrent-queue", qos: .userInitiated,
attributes: [.concurrent], autoreleaseFrequency: .workItem, target: nil)
let mySerialQueue = DispatchQueue(label: "my-serial-queue", qos: .background, attributes: [],
autoreleaseFrequency: .workItem, target: nil)
```
Version < 3.0
```
let myConcurrentQueue = dispatch_queue_create("my-concurrent-queue", DISPATCH_QUEUE_CONCURRENT)
let mySerialQueue = dispatch_queue_create("my-serial-queue", DISPATCH_QUEUE_SERIAL)
```

In Swift 3, queues created with this initializer are serial by default, and passing .workItem for autorelease frequency ensures an autorelease pool is created and drained for each work item. There is also .never, which means you will be managing your own autorelease pools yourself, or .inherit which inherits the setting from the environment. In most cases you probably won't use .never except in cases of extreme customization.

Section 32.2: Concurrent Loops

GCD provides mechanism for performing a loop, whereby the loops happen concurrently with respect to each

这在执行一系列计算量大的计算时非常有用。

考虑以下循环：

```
for index in 0 ..< iterations {
    // 在这里执行一些计算量大的操作
}
```

你可以使用concurrentPerform（在 Swift 3 中）或dispatch_apply（在 Swift 2 中）并发执行这些计算：

版本 = 3.0

```
DispatchQueue.concurrentPerform(iterations: iterations) { index in
    // 在这里执行一些计算量大的操作
}
```

版本 < 3.0

```
dispatch_apply(iterations, queue) { index in
    // 在这里执行一些计算量大的操作
}
```

循环闭包将针对从0到但不包括iterations的每个index调用。这些迭代将相互并发运行，因此它们运行的顺序不保证。实际同时并发运行的迭代数量通常由设备的能力决定（例如设备有多少个核心）。

几个特别注意事项：

- concurrentPerform/dispatch_apply可能会使循环相互并发运行，但这整个过程是同步发生的，针对调用它的线程而言。因此，不要从主线程调用它，因为这会阻塞主线程直到循环完成。

- 由于这些循环相互并发运行，你需要负责确保结果的线程安全。例如，如果用这些计算量大的结果更新某个字典，务必自己同步这些更新操作。

- 注意，运行并发循环会有一定的开销。因此，如果循环内执行的计算不够复杂，你可能会发现使用并发循环带来的性能提升会被同步这些并发线程的开销抵消，甚至完全抵消。

  因此，你需要自行判断每次循环迭代中应执行的工作量。如果计算过于简单，可以采用“步进”方式增加每次循环的工作量。例如，与其执行一百万次简单计算的并发循环，不如执行100次循环，每次循环做1万次计算。这样每个线程执行的工作量足够，管理这些并发循环的开销就会变得不那么显著。

第32.3节：在大中央调度（GCD）队列中运行任务

版本 = 3.0

要在调度队列上运行任务，请使用同步(sync)、异步(async)和延迟(after)方法。

要异步地将任务分派到队列：

other. This is very useful when performing a series of computationally expensive calculations.

Consider this loop:

```
for index in 0 ..< iterations {
    // Do something computationally expensive here
}
```

You can perform those calculations concurrently using `concurrentPerform` (in Swift 3) or `dispatch_apply` (in Swift 2):

Version = 3.0

```
DispatchQueue.concurrentPerform(iterations: iterations) { index in
    // Do something computationally expensive here
}
```

Version < 3.0

```
dispatch_apply(iterations, queue) { index in
    // Do something computationally expensive here
}
```

The loop closure will be invoked for each `index` from `0` to, but not including, `iterations`. These iterations will be run concurrently with respect to each other, and thus the order that they run is not guaranteed. The actual number of iterations that happen concurrently at any given time is generally dictated by the capabilities of the device in question (e.g. how many cores does the device have).

A couple of special considerations:

- The `concurrentPerform`/`dispatch_apply` may run the loops concurrently with respect to each other, but this all happens synchronously with respect to the thread from which you call it. So, do not call this from the main thread, as this will block that thread until the loop is done.

- Because these loops happen concurrently with respect to each other, you are responsible for ensuring the thread-safety of the results. For example, if updating some dictionary with the results of these computationally expensive calculations, make sure to synchronize those updates yourself.

- Note, there is some overhead associated in running concurrent loops. Thus, if the calculations being performed inside the loop are not sufficiently computationally intensive, you may find that any performance gained by using concurrent loops may be diminished, if not be completely offset, by the overhead associated with the synchronizing all of these concurrent threads.

  So, you are responsible determining the correct amount of work to be performed in each iteration of the loop. If the calculations are too simple, you may employ "striding" to include more work per loop. For example, rather than doing a concurrent loop with 1 million trivial calculations, you may do 100 iterations in your loop, doing 10,000 calculations per loop. That way there is enough work being performed on each thread, so the overhead associated with managing these concurrent loops becomes less significant.

Section 32.3: Running tasks in a Grand Central Dispatch (GCD) queue

Version = 3.0

To run tasks on a dispatch queue, use the `sync`, `async`, and `after` methods.

To dispatch a task to a queue asynchronously:

```
let queue = DispatchQueue(label: "myQueueName")

queue.async {
    //执行某些操作

DispatchQueue.main.async {
        //这将在主线程中调用
        //任何UI更新都应放在这里
    }
}
// ... 这里的代码会立即执行，早于任务完成
```

要同步地将任务分派到队列：

```
queue.sync {
    // 执行某些任务
}
// ... 代码将在任务完成后执行
```

在一定秒数后将任务派发到队列：

```
queue.asyncAfter(deadline: .now() + 3) {
    // 这将在3秒后于后台线程执行
}
// ... 这里的代码会立即执行，早于任务完成
```

> 注意： 任何用户界面的更新都应在主线程调用！确保将UI更新的代码放入DispatchQueue.main.async { ... }

版本 = 2.0

队列类型：

```
let mainQueue = dispatch_get_main_queue()
let highQueue = dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_HIGH, 0)
let backgroundQueue = dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_BACKGROUND, 0)
```

要异步地将任务分派到队列：

```
dispatch_async(queue) {
    // 你的代码将异步运行。代码被排队并在未来某个时间点执行。

}
// async块之后的代码将立即执行
```

要同步地将任务分派到队列：

```
dispatch_sync(queue) {
    // 你的同步代码
}
// 同步代码块之后的代码将等待同步任务完成
```

要在时间间隔后调度任务（使用NSEC_PER_SEC将秒转换为纳秒）：

```
dispatch_after(dispatch_time(DISPATCH_TIME_NOW, Int64(2.5 * Double(NSEC_PER_SEC))),
```



```
let queue = DispatchQueue(label: "myQueueName")

queue.async {
    //do something

    DispatchQueue.main.async {
        //this will be called in main thread
        //any UI updates should be placed here
    }
}
// ... code here will execute immediately, before the task finished
```

To dispatch a task to a queue synchronously:

```
queue.sync {
    // Do some task
}
// ... code here will not execute until the task is finished
```

To dispatch a task to a queue after a certain number of seconds:

```
queue.asyncAfter(deadline: .now() + 3) {
    //this will be executed in a background-thread after 3 seconds
}
// ... code here will execute immediately, before the task finished
```

> **NOTE:** Any updates of the user-interface should be called on the main thread! Make sure, that you put the code for UI updates inside `DispatchQueue.main.async { ... }`

Version = 2.0

Types of queue:

```
let mainQueue = dispatch_get_main_queue()
let highQueue = dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_HIGH, 0)
let backgroundQueue = dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_BACKGROUND, 0)
```

To dispatch a task to a queue asynchronously:

```
dispatch_async(queue) {
    // Your code run run asynchronously. Code is queued and executed
    // at some point in the future.
}
// Code after the async block will execute immediately
```

To dispatch a task to a queue synchronously:

```
dispatch_sync(queue) {
    // Your sync code
}
// Code after the sync block will wait until the sync task finished
```

To dispatch a task to after a time interval (use `NSEC_PER_SEC` to convert seconds to nanoseconds):

```
dispatch_after(dispatch_time(DISPATCH_TIME_NOW, Int64(2.5 * Double(NSEC_PER_SEC))),
```

```
dispatch_get_main_queue()) {
    // 2.5秒后执行的代码
}
```

异步执行任务然后更新UI：

```
dispatch_async(queue) {
    // 你的耗时代码
    dispatch_async(dispatch_get_main_queue()) {
        // 更新UI的代码
    }
}
```

> 注意：任何用户界面的更新都应在主线程调用！确保将UI更新代码放在 dispatch_async(dispatch_get_main_queue()) { ... } 内

第32.4节：在OperationQueue中运行任务

你可以将OperationQueue看作是一条等待执行的任务队列。与GCD中的调度队列不同，操作队列不是先进先出（FIFO）。相反，只要有足够的系统资源允许，它们会在任务准备好执行时立即执行。

获取主OperationQueue：

版本 ≥ 3.0

```
let mainQueue = OperationQueue.main
```

创建自定义OperationQueue：

版本 ≥ 3.0

```
let queue = OperationQueue()
queue.name = "我的队列"
queue.qualityOfService = .default
```

服务质量指定工作的重要性，或者用户多大程度上依赖任务的即时结果。

向OperationQueue添加一个Operation：

版本 ≥ 3.0

```
// 某个Operation子类的实例
let operation = BlockOperation {
    // 在这里执行任务
}

queue.addOperation(operation)
```

向OperationQueue添加一个块：

版本 ≥ 3.0

```
myQueue.addOperation {
    // 一些任务
}
```



```
dispatch_get_main_queue()) {
    // Code to be performed in 2.5 seconds here
}
```

To execute a task asynchronously and than update the UI:

```
dispatch_async(queue) {
    // Your time consuming code here
    dispatch_async(dispatch_get_main_queue()) {
        // Update the UI code
    }
}
```

> **NOTE:** Any updates of the user-interface should be called on the main thread! Make sure, that you put the code for UI updates inside `dispatch_async(dispatch_get_main_queue()) { ... }`

Section 32.4: Running Tasks in an OperationQueue

You can think of an `OperationQueue` as a line of tasks waiting to be executed. Unlike dispatch queues in GCD, operation queues are not FIFO (first-in-first-out). Instead, they execute tasks as soon as they are ready to be executed, as long as there are enough system resources to allow for it.

Get the main `OperationQueue`:

Version ≥ 3.0

```
let mainQueue = OperationQueue.main
```

Create a custom `OperationQueue`:

Version ≥ 3.0

```
let queue = OperationQueue()
queue.name = "My Queue"
queue.qualityOfService = .default
```

Quality of Service specifies the importance of the work, or how much the user is likely to be counting on immediate results from the task.

Add an `Operation` to an `OperationQueue`:

Version ≥ 3.0

```
// An instance of some Operation subclass
let operation = BlockOperation {
    // perform task here
}

queue.addOperation(operation)
```

Add a block to an `OperationQueue`:

Version ≥ 3.0

```
myQueue.addOperation {
    // some task
}
```

向OperationQueue添加多个Operation：

版本 ≥ 3.0

```
let operations = [Operation]()
// 用Operations填充数组

myQueue.addOperation(operations)
```

调整队列中可同时运行的Operation数量：

```
myQueue.maxConcurrentOperationCount = 3 // 允许同时执行3个操作

// 根据当前系统状况设置并发操作数
myQueue.maxConcurrentOperationCount = NSOperationQueueDefaultMaxConcurrentOperationCount
```

挂起队列将阻止其开始执行任何现有的、未启动的操作或任何新添加到队列中的操作。恢复该队列的方法是将`isSuspended`设置回`false`：

版本 ≥ 3.0

```
myQueue.isSuspended = true

// 重新启用执行
myQueue.isSuspended = false
```

挂起OperationQueue不会停止或取消已经在执行的操作。应仅尝试挂起你创建的队列，而不是全局队列或主队列。

第32.5节：创建高级操作

Foundation框架提供了Operation类型，它表示一个高级对象，封装了可以在队列上执行的一部分工作。队列不仅协调这些操作的执行，还可以建立操作之间的依赖关系，创建可取消的操作，限制操作队列使用的并发度等。

当所有依赖操作执行完成后，操作（Operation）变为准备执行状态。此时，isReady 属性会变为 true。

创建一个简单的非并发 Operation 子类：

版本 ≥ 3.0

```
class MyOperation: Operation {

    init(<parameters>) {
        // 在这里进行任何初始化工作
    }

    override func main() {
        // 执行任务
    }

}
```

版本 ≤ 2.3

```
class MyOperation: NSOperation {

    init(<parameters>) {
        // 在这里进行任何初始化工作
    }
```


Add multiple `Operations` to an `OperationQueue`:

Version ≥ 3.0

```
let operations = [Operation]()
// Fill array with Operations

myQueue.addOperation(operations)
```

Adjust how many `Operations` may be run concurrently within the queue:

```
myQueue.maxConcurrentOperationCount = 3 // 3 operations may execute at once

// Sets number of concurrent operations based on current system conditions
myQueue.maxConcurrentOperationCount = NSOperationQueueDefaultMaxConcurrentOperationCount
```

Suspending a queue will prevent it from starting the execution of any existing, unstarted operations or of any new operations added to the queue. The way to resume this queue is to set the `isSuspended` back to `false`:

Version ≥ 3.0

```
myQueue.isSuspended = true

// Re-enable execution
myQueue.isSuspended = false
```

Suspending an `OperationQueue` does not stop or cancel operations that are already executing. One should only attempt suspending a queue that you created, not global queues or the main queue.

Section 32.5: Creating High-Level Operations

The Foundation framework provides the `Operation` type, which represents a high-level object that encapsulates a portion of work that may be executed on a queue. Not only does the queue coordinate the performance of those operations, but you can also establish dependencies between operations, create cancelable operations, constrain the degree of concurrency employed by the operation queue, etc.

`Operations` become ready to execute when all of its dependencies are finished executing. The `isReady` property then changes to `true`.

Create a simple non-concurrent `Operation` subclass:

Version ≥ 3.0

```
class MyOperation: Operation {

    init(<parameters>) {
        // Do any setup work here
    }

    override func main() {
        // Perform the task
    }

}
```

Version ≤ 2.3

```
class MyOperation: NSOperation {

    init(<parameters>) {
        // Do any setup work here
    }
```

```
    override func main() {
        // 执行任务
    }

}
```

向 OperationQueue 添加操作：

版本 ≥ 1.0

```
myQueue.addOperation(operation)
```

这将在队列上并发执行该操作。

管理对Operation的依赖关系。

依赖关系定义了必须在该Operation被视为准备好执行之前，在队列上先执行的其他Operation。

版本 ≥ 1.0

```
operation2.addDependency(operation1)

operation2.removeDependency(operation1)
```

在没有队列的情况下运行一个Operation：

版本 ≥ 1.0

```
operation.start()
```

依赖关系将被忽略。如果这是一个并发操作，任务仍可能并发执行，前提是其start方法将工作卸载到后台队列。

并发操作。

如果Operation要执行的任务本身是异步的（例如URLSession数据任务），你应该将该Operation实现为并发操作。在这种情况下，你的isAsynchronous实现应返回true，通常你会有一个start方法执行一些设置，然后调用其main方法，实际执行任务。

当实现一个异步Operation开始时，必须实现isExecuting、isFinished方法以及KVO。因此，当执行开始时，isExecuting属性变为true。当一个Operation完成其任务时，isExecuting被设置为false，isFinished被设置为true。如果操作被取消，isCancelled和isFinished都会变为true。所有这些属性都是键值可观察的。

取消一个Operation。

调用cancel只是将isCancelled属性更改为true。为了响应来自你自己的Operation子类中的取消，你应该至少在main方法中定期检查isCancelled的值并做出相应处理。

版本 ≥ 1.0

```
operation.cancel()
```



```
    override func main() {
        // Perform the task
    }

}
```

Add an operation to an `OperationQueue`:

Version ≥ 1.0

```
myQueue.addOperation(operation)
```

This will execute the operation concurrently on the queue.

Manage dependencies on an `Operation`.

Dependencies define other `Operation`s that must execute on a queue before that `Operation` is considered ready to execute.

Version ≥ 1.0

```
operation2.addDependency(operation1)

operation2.removeDependency(operation1)
```

Run an `Operation` without a queue:

Version ≥ 1.0

```
operation.start()
```

Dependencies will be ignored. If this is a concurrent operation, the task may still be executed concurrently if its `start` method offloads work to background queues.

Concurrent Operations.

If the task that an `Operation` is to perform is, itself, asynchronous, (e.g. a `URLSession` data task), you should implement the `Operation` as a concurrent operation. In this case, your `isAsynchronous` implementation should return `true`, you'd generally have `start` method that performs some setup, then calls its `main` method which actually executes the task.

When implementing an asynchronous `Operation` begins you must implement `isExecuting`, `isFinished` methods and KVO. So, when execution starts, `isExecuting` property changes to `true`. When an `Operation` finishes its task, `isExecuting` is set to `false`, and `isFinished` is set to `true`. If the operation it is cancelled both `isCancelled` and `isFinished` change to `true`. All of these properties are key-value observable.

Cancel an `Operation`.

Calling `cancel` simply changes the `isCancelled` property to `true`. To respond to cancellation from within your own `Operation` subclass, you should check the value of `isCancelled` at least periodically within `main` and respond appropriately.

Version ≥ 1.0

```
operation.cancel()
```

第33章：协议导向编程入门

第33.1节：将协议作为一等类型使用

协议导向编程可以作为Swift的核心设计模式。

不同类型能够遵循相同的协议，值类型甚至可以遵循多个协议，甚至提供默认的方法实现。

最初定义的协议可以表示常用的属性和/或方法，类型可以是特定的也可以是通用的。

```
protocol ItemData {

    var title: String { get }
    var description: String { get }
    var thumbnailURL: NSURL { get }
    var created: NSDate { get }
    var updated: NSDate { get }

}

协议 DisplayItem {

    函数 hasBeenUpdated() -> Bool
    函数 getFormattedTitle() -> String
    函数 getFormattedDescription() -> String

}

协议 GetAPIItemDataOperation {

    静态函数 get(url: NSURL, completed: ([ItemData]) -> Void)
}
```

可以为 get 方法创建一个默认实现，但如果需要，遵循该协议的类型可以重写该实现。

```
扩展 GetAPIItemDataOperation {

    静态函数 get(url: NSURL, completed: ([ItemData]) -> Void) {

        让 date = NSDate(
timeIntervalSinceNow: NSDate().timeIntervalSince1970
            + 5000)

        // 从URL获取数据
        let urlData: [String: AnyObject] = [
            "title": "红色科迈罗",
            "desc": "一辆快速的红色汽车。",
            "thumb":"http://cars.images.com/red-camaro.png",
            "created": NSDate(), "updated": date]

        // 在此示例中使用了强制解包
        // 实际使用中绝不应使用强制解包
        // 应使用条件解包（guard 或 if/let）
```


Chapter 33: Getting Started with Protocol Oriented Programming

Section 33.1: Using protocols as first class types

Protocol oriented programing can be used as a core Swift design pattern.

Different types are able to conform to the same protocol, value types can even conform to multiple protocols and even provide default method implementation.

Initially protocols are defined that can represent commonly used properties and/or methods with either specific or generic types.

```
protocol ItemData {

    var title: String { get }
    var description: String { get }
    var thumbnailURL: NSURL { get }
    var created: NSDate { get }
    var updated: NSDate { get }

}

protocol DisplayItem {

    func hasBeenUpdated() -> Bool
    func getFormattedTitle() -> String
    func getFormattedDescription() -> String

}

protocol GetAPIItemDataOperation {

    static func get(url: NSURL, completed: ([ItemData]) -> Void)
}
```

A default implementation for the get method can be created, though if desired conforming types may override the implementation.

```
extension GetAPIItemDataOperation {

    static func get(url: NSURL, completed: ([ItemData]) -> Void) {

        let date = NSDate(
        timeIntervalSinceNow: NSDate().timeIntervalSince1970
            + 5000)

        // get data from url
        let urlData: [String: AnyObject] = [
            "title": "Red Camaro",
            "desc": "A fast red car.",
            "thumb":"http://cars.images.com/red-camaro.png",
            "created": NSDate(), "updated": date]

        // in this example forced unwrapping is used
        // forced unwrapping should never be used in practice
        // instead conditional unwrapping should be used (guard or if/let)
```

```
        let item = Item(
title: urlData["title"] as! String,
            description: urlData["desc"] as! String,
            thumbnailURL: NSURL(string: urlData["thumb"] as! String)!,
            created: urlData["created"] as! NSDate,
updated: urlData["updated"] as! NSDate)

        completed([item])

    }
}

struct ItemOperation: GetAPIItemDataOperation { }
```

一个符合ItemData协议的值类型，该值类型也能够符合其他协议。

```
struct Item: ItemData {

    let 标题: String
    let 描述: String
    let 缩略图URL: NSURL
    let 创建时间: NSDate
    let 更新时间: NSDate

}
```

这里将 item 结构体扩展为符合显示项（DisplayItem）。

```
extension Item: DisplayItem {

    func hasBeenUpdated() -> Bool {
        return updated.timeIntervalSince1970 >
            created.timeIntervalSince1970
    }

    func getFormattedTitle() -> String {
        return title.stringByTrimmingCharactersInSet(
            .whitespaceAndNewlineCharacterSet())
    }

    func getFormattedDescription() -> String {
        return description.stringByTrimmingCharactersInSet(
            .whitespaceAndNewlineCharacterSet())
    }
}
```

使用静态 get 方法的示例调用位置。

```
ItemOperation.get(NSURL()) { (itemData) in

    // 可能通知视图有新数据
    // 或解析数据以获取用户请求的信息，等等。
    dispatch_async(dispatch_get_main_queue(), {

        // self.items = itemData
    })

}
```

不同的使用场景将需要不同的实现。这里的主要思想是展示协议作为设计中关注的重点，在不同类型中的一致性。


```
        let item = Item(
            title: urlData["title"] as! String,
            description: urlData["desc"] as! String,
            thumbnailURL: NSURL(string: urlData["thumb"] as! String)!,
            created: urlData["created"] as! NSDate,
            updated: urlData["updated"] as! NSDate)

        completed([item])

    }
}

struct ItemOperation: GetAPIItemDataOperation { }
```

A value type that conforms to the ItemData protocol, this value type is also able to conform to other protocols.

```
struct Item: ItemData {

    let title: String
    let description: String
    let thumbnailURL: NSURL
    let created: NSDate
    let updated: NSDate

}
```

Here the item struct is extended to conform to a display item.

```
extension Item: DisplayItem {

    func hasBeenUpdated() -> Bool {
        return updated.timeIntervalSince1970 >
            created.timeIntervalSince1970
    }

    func getFormattedTitle() -> String {
        return title.stringByTrimmingCharactersInSet(
            .whitespaceAndNewlineCharacterSet())
    }

    func getFormattedDescription() -> String {
        return description.stringByTrimmingCharactersInSet(
            .whitespaceAndNewlineCharacterSet())
    }
}
```

An example call site for using the static get method.

```
ItemOperation.get(NSURL()) { (itemData) in

    // perhaps inform a view of new data
    // or parse the data for user requested info, etc.
    dispatch_async(dispatch_get_main_queue(), {

        // self.items = itemData
    })

}
```

Different use cases will require different implementations. The main idea here is to show conformance from

在这个例子中，API 数据可能有条件地保存到 Core Data 实体中。

```
// 默认的 Core Data 创建的类 + 扩展
class LocalItem: NSManagedObject { }

extension LocalItem {

    @NSManaged var title: String
    @NSManaged var itemDescription: String
    @NSManaged var thumbnailURLStr: String
    @NSManaged var createdAt: NSDate
    @NSManaged var updatedAt: NSDate
}
```

这里，基于 Core Data 的类也可以遵循 DisplayItem 协议。

```
extension LocalItem: DisplayItem {

    func hasBeenUpdated() -> Bool {
        return updatedAt.timeIntervalSince1970 >
            createdAt.timeIntervalSince1970
    }

    func getFormattedTitle() -> String {
        return title.stringByTrimmingCharactersInSet(
            .whitespaceAndNewlineCharacterSet())
    }

    func getFormattedDescription() -> String {
        return itemDescription.stringByTrimmingCharactersInSet(
            .whitespaceAndNewlineCharacterSet())
    }
}

// 在使用中，核心数据结果可以
// 有条件地转换为协议
class MyController: UIViewController {

    override func viewDidLoad() {

        let fr: NSFetchRequest = NSFetchRequest(
        entityName: "Items")

        let context = NSManagedObjectContext(
        concurrencyType: .MainQueueConcurrencyType)

        do {

            let items: AnyObject = try context.executeFetchRequest(fr)
            if let displayItems = items as? [DisplayItem] {

                print(displayItems)
            }

        } catch let error as NSError {
            print(error.localizedDescription)
        }

    }
```


varying types where the protocol is the main point of the focus in the design. In this example perhaps the API data is conditionally saved to a Core Data entity.

```
// the default core data created classes + extension
class LocalItem: NSManagedObject { }

extension LocalItem {

    @NSManaged var title: String
    @NSManaged var itemDescription: String
    @NSManaged var thumbnailURLStr: String
    @NSManaged var createdAt: NSDate
    @NSManaged var updatedAt: NSDate
}
```

Here the Core Data backed class can also conform to the DisplayItem protocol.

```
extension LocalItem: DisplayItem {

    func hasBeenUpdated() -> Bool {
        return updatedAt.timeIntervalSince1970 >
            createdAt.timeIntervalSince1970
    }

    func getFormattedTitle() -> String {
        return title.stringByTrimmingCharactersInSet(
            .whitespaceAndNewlineCharacterSet())
    }

    func getFormattedDescription() -> String {
        return itemDescription.stringByTrimmingCharactersInSet(
            .whitespaceAndNewlineCharacterSet())
    }
}

// In use, the core data results can be
// conditionally casts as a protocol
class MyController: UIViewController {

    override func viewDidLoad() {

        let fr: NSFetchRequest = NSFetchRequest(
        entityName: "Items")

        let context = NSManagedObjectContext(
        concurrencyType: .MainQueueConcurrencyType)

        do {

            let items: AnyObject = try context.executeFetchRequest(fr)
            if let displayItems = items as? [DisplayItem] {

                print(displayItems)
            }

        } catch let error as NSError {
            print(error.localizedDescription)
        }

    }
```

```
}
```

第33.2节：利用面向协议编程进行单元测试

面向协议编程是一个有用的工具，可以轻松地为我们的代码编写更好的单元测试。

假设我们想测试一个依赖于ViewModel类的UIViewController。

生产代码所需的步骤如下：

1. 定义一个协议，公开 ViewModel 类的公共接口，包含 UIViewController 所需的所有属性和方法。
2. 实现符合该协议的真实 ViewModel 类。
3. 使用依赖注入技术，让视图控制器使用我们想要的实现，传递的是协议类型而非具体实例。

```
protocol ViewModelType {
   var title : String {get}
   func confirm()
}

class ViewModel : ViewModelType {
   let title : String

   init(title: String) {
       self.title = title
   }
   func confirm() { … }
}

class ViewController : UIViewController {
   // 我们将 viewModel 属性声明为符合协议的对象
   // 这样可以无缝替换实现。
   var viewModel : ViewModelType!
   @IBOutlet 变量 titleLabel : UILabel!

   override func viewDidLoad() {
       super.viewDidLoad()
titleLabel.text = viewModel.title
   }

   @IBAction func didTapOnButton(sender: UIButton) {
       viewModel.confirm()
   }
}

// 通过依赖注入（DI）设置视图控制器并分配视图模型。
// 视图控制器不知道视图模型的具体类，
// 只依赖协议中声明的接口。
let viewController = //… 实例化视图控制器
viewController.viewModel = ViewModel(title: "MyTitle")
```

然后，在单元测试中：

1. 实现一个符合相同协议的模拟（mock）视图模型
2. 使用依赖注入将其传递给被测试的UIViewController，而非真实实例。
3. 测试！


```
}
```

Section 33.2: Leveraging Protocol Oriented Programming for Unit Testing

Protocol Oriented Programming is a useful tool in order to easily write better unit tests for our code.

Let's say we want to test a UIViewController that relies on a ViewModel class.

The needed steps on the production code are:

1. Define a protocol that exposes the public interface of the class ViewModel, with all the properties and methods needed by the UIViewController.
2. Implement the real ViewModel class, conforming to that protocol.
3. Use a dependency injection technique to let the view controller use the implementation we want, passing it as the protocol and not the concrete instance.

```
protocol ViewModelType {
   var title : String {get}
   func confirm()
}

class ViewModel : ViewModelType {
   let title : String

   init(title: String) {
       self.title = title
   }
   func confirm() { ... }
}

class ViewController : UIViewController {
   // We declare the viewModel property as an object conforming to the protocol
   // so we can swap the implementations without any friction.
   var viewModel : ViewModelType!
   @IBOutlet var titleLabel : UILabel!

   override func viewDidLoad() {
       super.viewDidLoad()
       titleLabel.text = viewModel.title
   }

   @IBAction func didTapOnButton(sender: UIButton) {
       viewModel.confirm()
   }
}

// With DI we setup the view controller and assign the view model.
// The view controller doesn't know the concrete class of the view model,
// but just relies on the declared interface on the protocol.
let viewController = //... Instantiate view controller
viewController.viewModel = ViewModel(title: "MyTitle")
```

Then, on unit test:

1. Implement a mock ViewModel that conforms to the same protocol
2. Pass it to the UIViewController under test using dependency injection, instead of the real instance.
3. Test!

```
class FakeViewModel : ViewModelType {
   let title : String = "FakeTitle"

   var didConfirm = false
   func confirm() {
didConfirm = true
   }
}

class ViewControllerTest : XCTestCase {
    var sut : ViewController!
    var viewModel : FakeViewModel!

    override func setUp() {
        super.setUp()

viewModel = FakeViewModel()
        sut = // … 初始化视图控制器
       sut.viewModel = viewModel

XCTAssertNotNil(self.sut.view) // 需要触发视图加载
    }

    func testTitleLabel() {
XCTAssertEqual(self.sut.titleLabel.text, "FakeTitle")
    }

    func testTapOnButton() {
        sut.didTapOnButton(UIButton())
        XCTAssertTrue(self.viewModel.didConfirm)
    }
}
```



```
class FakeViewModel : ViewModelType {
   let title : String = "FakeTitle"

   var didConfirm = false
   func confirm() {
       didConfirm = true
   }
}

class ViewControllerTest : XCTestCase {
    var sut : ViewController!
    var viewModel : FakeViewModel!

    override func setUp() {
        super.setUp()

        viewModel = FakeViewModel()
        sut = // ... initialization for view controller
        sut.viewModel = viewModel

        XCTAssertNotNil(self.sut.view) // Needed to trigger view loading
    }

    func testTitleLabel() {
        XCTAssertEqual(self.sut.titleLabel.text, "FakeTitle")
    }

    func testTapOnButton() {
        sut.didTapOnButton(UIButton())
        XCTAssertTrue(self.viewModel.didConfirm)
    }
}
```

第34章：Swift中的函数式编程

第34.1节：从人员列表中提取姓名列表

给定一个Person结构体

```
struct Person {
    let name: String
    let birthYear: Int?
}
```

以及一个Person数组

```
let persons = [
Person(name: "Walter White", birthYear: 1959),
    Person(name: "Jesse Pinkman", birthYear: 1984),
    Person(name: "Skyler White", birthYear: 1970),
    Person(name: "Saul Goodman", birthYear: nil)
]
```

我们可以获取一个包含每个Person的name属性的String数组。

```
let names = persons.map { $0.name }
// ["Walter White", "Jesse Pinkman", "Skyler White", "Saul Goodman"]
```

第34.2节：遍历

```
let numbers = [3, 1, 4, 1, 5]
// 非函数式
for (index, element) in numbers.enumerate() {
    print(index, element)
}

// functional
numbers.enumerate().map { (index, element) in
    print((index, element))
}
```

第34.3节：过滤

通过选择流中满足某个条件的元素来创建一个流称为过滤

```
var newReleases = [
    [
        "id": 70111470,
        "title": "虎胆龙威",
        "boxart": "http://cdn-0.nflximg.com/images/2891/DieHard.jpg",
        "uri": "http://api.netflix.com/catalog/titles/movies/70111470",
        "rating": 4.0,
        "bookmark": []
    ],
    [
        "id": 654356453,
        "title": "坏男孩",
```


Chapter 34: Functional Programming in Swift

Section 34.1: Extracting a list of names from a list of Person(s)

Given a `Person` struct

```
struct Person {
    let name: String
    let birthYear: Int?
}
```

and an Array of `Person(s)`

```
let persons = [
    Person(name: "Walter White", birthYear: 1959),
    Person(name: "Jesse Pinkman", birthYear: 1984),
    Person(name: "Skyler White", birthYear: 1970),
    Person(name: "Saul Goodman", birthYear: nil)
]
```

we can retrieve an array of `String` containing the `name` property of each Person.

```
let names = persons.map { $0.name }
// ["Walter White", "Jesse Pinkman", "Skyler White", "Saul Goodman"]
```

Section 34.2: Traversing

```
let numbers = [3, 1, 4, 1, 5]
// non-functional
for (index, element) in numbers.enumerate() {
    print(index, element)
}

// functional
numbers.enumerate().map { (index, element) in
    print((index, element))
}
```

Section 34.3: Filtering

Create a stream by selecting the elements from a stream that pass a certain condition is called **filtering**

```
var newReleases = [
    [
        "id": 70111470,
        "title": "Die Hard",
        "boxart": "http://cdn-0.nflximg.com/images/2891/DieHard.jpg",
        "uri": "http://api.netflix.com/catalog/titles/movies/70111470",
        "rating": 4.0,
        "bookmark": []
    ],
    [
        "id": 654356453,
        "title": "Bad Boys",
```

```
        "boxart": "http://cdn-0.nflximg.com/images/2891/BadBoys.jpg",
        "uri": "http://api.netflix.com/catalog/titles/movies/70111470",
        "rating": 5.0,
        "bookmark": [[ "id": 432534, "time": 65876586 ]]
    ],
    [
        "id": 65432445,
        "title": "密室",
        "boxart": "http://cdn-0.nflximg.com/images/2891/TheChamber.jpg",
        "uri": "http://api.netflix.com/catalog/titles/movies/70111470",
        "rating": 4.0,
        "bookmark": []
    ],
    [
        "id": 675465,
        "title": "骨折",
        "boxart": "http://cdn-0.nflximg.com/images/2891/Fracture.jpg",
        "uri": "http://api.netflix.com/catalog/titles/movies/70111470",
        "rating": 5.0,
        "bookmark": [[ "id": 432534, "time": 65876586 ]]
    ]
]

var videos1 = [[String: AnyObject]]()
/**
*  使用 map 进行过滤
 */
newReleases.map { e in
    if e["rating"] as! Float == 5.0 {
videos1.append(["id": e["id"] as! Int, "title": e["title"] as! String])
    }
}

print(videos1)

var videos2 = [[String: AnyObject]]()
/**
*  使用 filter 和链式调用进行过滤
 */
newReleases
.filter{ e in
e["rating"] as! Float == 5.0
    }
.map { e in
videos2.append(["id": e["id"] as! Int, "title": e["title"] as! String])
}

print(videos2)
```

第34.4节：使用Filter过滤结构体

你经常可能想要过滤结构体和其他复杂数据类型。在结构体数组中搜索包含特定值的条目是一个非常常见的任务，使用Swift的函数式编程特性可以轻松实现。而且，代码非常简洁。

```
struct 画家 {
    enum 类型 { case 印象派, 表现主义, 超现实主义, 抽象派, 波普艺术 }
    var 名字: String
    var 姓氏: String
    var 类型: 类型
```



```
        "boxart": "http://cdn-0.nflximg.com/images/2891/BadBoys.jpg",
        "uri": "http://api.netflix.com/catalog/titles/movies/70111470",
        "rating": 5.0,
        "bookmark": [[ "id": 432534, "time": 65876586 ]]
    ],
    [
        "id": 65432445,
        "title": "The Chamber",
        "boxart": "http://cdn-0.nflximg.com/images/2891/TheChamber.jpg",
        "uri": "http://api.netflix.com/catalog/titles/movies/70111470",
        "rating": 4.0,
        "bookmark": []
    ],
    [
        "id": 675465,
        "title": "Fracture",
        "boxart": "http://cdn-0.nflximg.com/images/2891/Fracture.jpg",
        "uri": "http://api.netflix.com/catalog/titles/movies/70111470",
        "rating": 5.0,
        "bookmark": [[ "id": 432534, "time": 65876586 ]]
    ]
]

var videos1 = [[String: AnyObject]]()
/**
 *  Filtering using map
 */
newReleases.map { e in
    if e["rating"] as! Float == 5.0 {
        videos1.append(["id": e["id"] as! Int, "title": e["title"] as! String])
    }
}

print(videos1)

var videos2 = [[String: AnyObject]]()
/**
 *  Filtering using filter and chaining
 */
newReleases
    .filter{ e in
        e["rating"] as! Float == 5.0
    }
    .map { e in
    videos2.append(["id": e["id"] as! Int, "title": e["title"] as! String])
}

print(videos2)
```

Section 34.4: Using Filter with Structs

Frequently you may want to filter structures and other complex data types. Searching an array of structs for entries that contain a particular value is a very common task, and easily achieved in Swift using functional programming features. What's more, the code is extremely succinct.

```
struct Painter {
    enum Type { case Impressionist, Expressionist, Surrealist, Abstract, Pop }
    var firstName: String
    var lastName: String
    var type: Type
```

```
}

let 画家们 = [
Painter(firstName: "克洛德", lastName: "莫奈", type: .印象派),
    Painter(firstName: "埃德加", lastName: "德加", type: .印象派),
    Painter(firstName: "埃贡", lastName: "席勒", type: .表现主义),
    Painter(firstName: "乔治", lastName: "格罗斯", type: .表现主义),
    Painter(firstName: "马克", lastName: "罗斯科", type: .抽象派),
画家(名字: "杰克逊", 姓氏: "波洛克", 类型: .抽象),
    画家(名字: "巴勃罗", 姓氏: "毕加索", 类型: .超现实主义),
    画家(名字: "安迪", 姓氏: "沃霍尔", 类型: .波普)
]

// 列出表现主义画家
dump(painters.filter({$0.type == .表现主义}))

// 统计表现主义画家数量
dump(painters.filter({$0.type == .表现主义}).count)
// 输出 "2"

// 结合 filter 和 map 进行更复杂的操作，例如列出所有
// 非印象派和非表现主义画家的姓氏
dump(painters.filter({$0.type != .印象派 && $0.type != .表现主义})
    .map({$0.lastName}).joinWithSeparator(", "))
// 输出 "罗斯科, 波洛克, 毕加索, 沃霍尔"
```

第34.5节：投影

将函数应用于集合/流并创建新的集合/流称为投影。

```
/// 投影
var newReleases = [
    [
        "id": 70111470,
        "title": "虎胆龙威",
        "boxart": "http://cdn-0.nflximg.com/images/2891/DieHard.jpg",
        "uri": "http://api.netflix.com/catalog/titles/movies/70111470",
        "rating": [4.0],
        "bookmark": []
    ],
    [
        "id": 654356453,
        "title": "坏男孩",
        "boxart": "http://cdn-0.nflximg.com/images/2891/BadBoys.jpg",
        "uri": "http://api.netflix.com/catalog/titles/movies/70111470",
        "rating": [5.0],
        "bookmark": [[ "id": 432534, "time": 65876586 ]]
    ],
    [
        "id": 65432445,
        "title": "密室",
        "boxart": "http://cdn-0.nflximg.com/images/2891/TheChamber.jpg",
        "uri": "http://api.netflix.com/catalog/titles/movies/70111470",
        "rating": [4.0],
        "bookmark": []
    ],
    [
        "id": 675465,
        "title": "骨折",
        "boxart": "http://cdn-0.nflximg.com/images/2891/Fracture.jpg",
        "uri": "http://api.netflix.com/catalog/titles/movies/70111470",
```



```
}

let painters = [
    Painter(firstName: "Claude", lastName: "Monet", type: .Impressionist),
    Painter(firstName: "Edgar", lastName: "Degas", type: .Impressionist),
    Painter(firstName: "Egon", lastName: "Schiele", type: .Expressionist),
    Painter(firstName: "George", lastName: "Grosz", type: .Expressionist),
    Painter(firstName: "Mark", lastName: "Rothko", type: .Abstract),
    Painter(firstName: "Jackson", lastName: "Pollock", type: .Abstract),
    Painter(firstName: "Pablo", lastName: "Picasso", type: .Surrealist),
    Painter(firstName: "Andy", lastName: "Warhol", type: .Pop)
]

// list the expressionists
dump(painters.filter({$0.type == .Expressionist}))

// count the expressionists
dump(painters.filter({$0.type == .Expressionist}).count)
// prints "2"

// combine filter and map for more complex operations, for example listing all
// non-impressionist and non-expressionists by surname
dump(painters.filter({$0.type != .Impressionist && $0.type != .Expressionist})
    .map({$0.lastName}).joinWithSeparator(", "))
// prints "Rothko, Pollock, Picasso, Warhol"
```

Section 34.5: Projecting

Applying a function to a collection/stream and creating a new collection/stream is called a **projection**.

```
/// Projection
var newReleases = [
    [
        "id": 70111470,
        "title": "Die Hard",
        "boxart": "http://cdn-0.nflximg.com/images/2891/DieHard.jpg",
        "uri": "http://api.netflix.com/catalog/titles/movies/70111470",
        "rating": [4.0],
        "bookmark": []
    ],
    [
        "id": 654356453,
        "title": "Bad Boys",
        "boxart": "http://cdn-0.nflximg.com/images/2891/BadBoys.jpg",
        "uri": "http://api.netflix.com/catalog/titles/movies/70111470",
        "rating": [5.0],
        "bookmark": [[ "id": 432534, "time": 65876586 ]]
    ],
    [
        "id": 65432445,
        "title": "The Chamber",
        "boxart": "http://cdn-0.nflximg.com/images/2891/TheChamber.jpg",
        "uri": "http://api.netflix.com/catalog/titles/movies/70111470",
        "rating": [4.0],
        "bookmark": []
    ],
    [
        "id": 675465,
        "title": "Fracture",
        "boxart": "http://cdn-0.nflximg.com/images/2891/Fracture.jpg",
        "uri": "http://api.netflix.com/catalog/titles/movies/70111470",
```

```
        "rating": [5.0],
        "bookmark": [[ "id": 432534, "time": 65876586 ]]
    ]
]


var videoAndTitlePairs = [[String: AnyObject]]()
newReleases.map { e in
videoAndTitlePairs.append(["id": e["id"] as! Int, "title": e["title"] as! String])
}

print(videoAndTitlePairs)
```



```
        "rating": [5.0],
        "bookmark": [[ "id": 432534, "time": 65876586 ]]
    ]
]


var videoAndTitlePairs = [[String: AnyObject]]()
newReleases.map { e in
    videoAndTitlePairs.append(["id": e["id"] as! Int, "title": e["title"] as! String])
}

print(videoAndTitlePairs)
```

第35章：Swift中的函数作为一等公民

作为一等公民的函数意味着它可以享有与对象相同的特权。它可以被赋值给变量，作为参数传递给函数，或者用作返回类型。

第35.1节：将函数赋值给变量

```
结构体 数学
{
    内部函数 执行操作(inputArray: [整数], operation: (整数)-> 整数)-> [整数]
    {
        变量 处理数组 = [整数]()

        对于 项目 在 输入数组
        {
处理数组.添加(operation(项目))
        }

        返回 处理数组
    }


    内部函数 执行复杂操作(valueOne: 整数)-> ((整数)-> 整数)
    {
        return
            ({
                返回 valueOne + $0
            })
    }

}


let arrayToBeProcessed = [1,3,5,7,9,11,8,6,4,2,100]

let math = Mathematics()

func add2(item: Int)-> Int
{
    return (item + 2)
}

// 将函数赋值给变量，然后作为参数传递给函数
let add2ToMe = add2
print(math.performOperation(inputArray: arrayToBeProcessed, operation: add2ToMe))
```

输出：

```
[3, 5, 7, 9, 11, 13, 10, 8, 6, 4, 102]
```

同样，上述功能也可以使用闭包实现

```
// 将闭包赋值给变量，然后作为参数传递给函数
let add2 = {(item: Int)-> Int in return item + 2}
print(math.performOperation(inputArray: arrayToBeProcessed, operation: add2))
```


Chapter 35: Function as first class citizens in Swift

Functions as First-class members means, it can enjoy privileges just like Objects does. It can be assigned to a variable, passed on to a function as parameter or can be used as return type.

Section 35.1: Assigning function to a variable

```
struct Mathematics
{
    internal func performOperation(inputArray: [Int], operation: (Int)-> Int)-> [Int]
    {
        var processedArray = [Int]()

        for item in inputArray
        {
            processedArray.append(operation(item))
        }

        return processedArray
    }


    internal func performComplexOperation(valueOne: Int)-> ((Int)-> Int)
    {
        return
            ({
                return valueOne + $0
            })
    }

}


let arrayToBeProcessed = [1,3,5,7,9,11,8,6,4,2,100]

let math = Mathematics()

func add2(item: Int)-> Int
{
    return (item + 2)
}

// assigning the function to a variable and then passing it to a function as param
let add2ToMe = add2
print(math.performOperation(inputArray: arrayToBeProcessed, operation: add2ToMe))
```

Output:

```
[3, 5, 7, 9, 11, 13, 10, 8, 6, 4, 102]
```

Similarly the above could be achieved using a `closure`

```
// assigning the closure to a variable and then passing it to a function as param
let add2 = {(item: Int)-> Int in return item + 2}
print(math.performOperation(inputArray: arrayToBeProcessed, operation: add2))
```

第35.2节：将函数作为参数传递给另一个函数，从而创建高阶函数

```
func multiply2(item: Int)-> Int
{
    return (item + 2)
}


let multiply2ToMe = multiply2

// 直接将函数作为参数传递给函数
print(math.performOperation(inputArray: arrayToBeProcessed, operation: multiply2ToMe))
```

输出：

```
[3, 5, 7, 9, 11, 13, 10, 8, 6, 4, 102]
```

同样，上述功能也可以使用闭包实现

```
// 直接将闭包作为参数传递给函数
print(math.performOperation(inputArray: arrayToBeProcessed, operation: { $0 * 2 }))
```

第35.3节：函数作为另一个函数的返回类型

```
// 函数作为返回类型
print(math.performComplexOperation(valueOne: 4)(5))
```

输出：

9

Section 35.2: Passing function as an argument to another function, thus creating a Higher-Order Function

```
func multiply2(item: Int)-> Int
{
    return (item + 2)
}


let multiply2ToMe = multiply2

// passing the function directly to the function as param
print(math.performOperation(inputArray: arrayToBeProcessed, operation: multiply2ToMe))
```

Output:

```
[3, 5, 7, 9, 11, 13, 10, 8, 6, 4, 102]
```

Similarly the above could be achieved using a `closure`

```
// passing the closure directly to the function as param
print(math.performOperation(inputArray: arrayToBeProcessed, operation: { $0 * 2 }))
```

Section 35.3: Function as return type from another function

```
// function as return type
print(math.performComplexOperation(valueOne: 4)(5))
```

Output:

9

第36章：代码块

摘自Swift文档

当闭包作为参数传递给函数，但在函数返回后才被调用时，称该闭包逃逸了函数。当你声明一个以闭包作为参数的函数时，可以在参数类型前写@escaping，表示该闭包允许逃逸。

第36.1节：非逃逸闭包

在 Swift 1 和 2 中，闭包参数默认是逃逸的。如果你知道闭包不会逃逸函数体，可以用 @noescape 属性标记该参数。

在 Swift 3 中，情况正好相反：闭包参数默认是非逃逸的。如果你打算让它逃逸函数，则必须用 @escaping 属性标记。

```
类 ClassOne {
   // @noescape 默认应用于此处
   func methodOne(completion: () -> Void) {
      //
   }
}

类 ClassTwo {
   let obj = ClassOne()
   var greeting = "Hello, World!"

   func methodTwo() {
     obj.methodOne() {
        // 需要使用 self.greeting
        print(greeting)
     }
   }
}
```

第 36.2 节：逃逸闭包

摘自Swift文档

> @escaping

> 将此属性应用于方法或函数声明中参数的类型，以表示该参数的值可以被存储以供稍后执行。这意味着该值可以超出调用的生命周期。带有 escaping 类型属性的函数类型参数需要显式使用 self。
>
> 用于属性或方法。

```
class ClassThree {

    var closure: (() -> ())?

    func doSomething(completion: @escaping () -> ()) {
       closure = finishBlock
    }
```


Chapter 36: Blocks

From Swift Documentarion

A closure is said to escape a function when the closure is passed as an argument to the function, but is called after the function returns. When you declare a function that takes a closure as one of its parameters, you can write @escaping before the parameter's type to indicate that the closure is allowed to escape.

Section 36.1: Non-escaping closure

In Swift 1 and 2, closure parameters were escaping by default. If you knew your closure wouldn't escape the function body, you could mark the parameter with the @noescape attribute.

In Swift 3, it's the other way around: closure parameters are non-escaping by default. If you intend for it to escape the function, you have to mark it with the @escaping attribute.

```
class ClassOne {
  // @noescape is applied here as default
  func methodOne(completion: () -> Void) {
    //
  }
}

class ClassTwo {
  let obj = ClassOne()
  var greeting = "Hello, World!"

  func methodTwo() {
    obj.methodOne() {
      // self.greeting is required
      print(greeting)
    }
  }
}
```

Section 36.2: Escaping closure

From Swift Documentarion

> @escaping

> Apply this attribute to a parameter's type in a method or function declaration to indicate that the parameter's value can be stored for later execution. This means that the value is allowed to outlive the lifetime of the call. Function type parameters with the escaping type attribute require explicit use of self. for properties or methods.

```
class ClassThree {

    var closure: (() -> ())?

    func doSomething(completion: @escaping () -> ()) {
       closure = finishBlock
    }
```

```
}
```

在上述示例中，completion闭包被保存到closure中，并且实际上会在函数调用结束后继续存在。因此，编译器会强制将completion闭包标记为@escaping。


```
}
```

In the above example the completion block is saved to closure and will literally live beyond the function call. So complier will force to mark completion block as @escaping.

第37章：defer语句

第37.1节：何时使用defer语句

defer语句由一段代码块组成，该代码块将在函数返回时执行，应用于清理工作。

由于Swift的guard语句鼓励早期返回的编程风格，可能存在多条返回路径。defer语句提供了清理代码，从而无需在每个返回点重复编写清理逻辑。

它还可以节省调试和性能分析时的时间，因为可以避免因忘记清理而导致的内存泄漏和未关闭的资源。

它可以用于在函数结束时释放缓冲区：

```
函数 doSomething() {
    let data = UnsafeMutablePointer<UInt8>(allocatingCapacity: 42)
    // 该指针在函数返回时不会被释放
    // 因此我们添加了一个 defer 语句
    defer {
data.deallocateCapacity(42)
    }
    // 它将在函数返回时执行。

guard condition else {
        return /* 将执行 defer 块 */
    }

}   // defer 块也会在函数结束时执行。
```

它也可以用于在函数结束时关闭资源：

```
func write(data: UnsafePointer<UInt8>, dataLength: Int) throws {
    var stream:NSOutputStream = getOutputStream()
defer {
        stream.close()
    }

    let written = stream.write(data, maxLength: dataLength)
    guard written >= 0 else {
throw stream.streamError! /* 将执行 defer 块 */
    }

}   // 函数结束时也会执行 defer 块
```

第37.2节：何时不使用 defer 语句

使用 defer 语句时，确保代码保持可读性且执行顺序清晰。例如，以下 defer 语句的使用使得执行顺序和代码功能难以理解。

```
后缀函数 ++ (inout value: Int) -> Int {
    defer { value += 1 } // 不要这样做！
    return value
}
```


Chapter 37: The Defer Statement

Section 37.1: When to use a defer statement

A `defer` statement consists of a block of code, which will be executed when a function returns and should be used for cleanup.

As Swift's `guard` statements encourage a style of early return, many possible paths for a return may exist. A `defer` statement provides cleanup code, which then does not need to be repeated every time.

It can also save time during debugging and profiling, as memory leaks and unused open resources due to forgotten cleanup can be avoided.

It can be used to deallocate a buffer at the end of a function:

```
func doSomething() {
    let data = UnsafeMutablePointer<UInt8>(allocatingCapacity: 42)
    // this pointer would not be released when the function returns
    // so we add a defer-statement
    defer {
        data.deallocateCapacity(42)
    }
    // it will be executed when the function returns.

    guard condition else {
        return /* will execute defer-block */
    }

}   // The defer-block will also be executed on the end of the function.
```

It can also be used to close resources at the end of a function:

```
func write(data: UnsafePointer<UInt8>, dataLength: Int) throws {
    var stream:NSOutputStream = getOutputStream()
    defer {
        stream.close()
    }

    let written = stream.write(data, maxLength: dataLength)
    guard written >= 0 else {
        throw stream.streamError! /* will execute defer-block */
    }

}    // the defer-block will also be executed on the end of the function
```

Section 37.2: When NOT to use a defer statement

When using a defer-statement, make sure the code remains readable and the execution order remains clear. For example, the following use of the defer-statement makes the execution order and the function of the code hard to comprehend.

```
postfix func ++ (inout value: Int) -> Int {
    defer { value += 1 } // do NOT do this!
    return value
}
```

第38章：风格规范

第38.1节：流畅用法

使用自然语言
函数调用应接近自然英语语言。

示例：

```
list.insert(元素, 在: 索引)
```

而不是

```
list.insert(元素, 位置: 索引)
```

工厂方法命名
工厂方法应以前缀 `make` 开头。

示例：

```
factory.makeObject()
```

初始化器和工厂方法中的参数命名
第一个参数的名称不应包含在工厂方法或初始化器的命名中。

示例：

```
factory.makeObject(键: 值)
```

而不是：

```
factory.makeObject(具有属性: value)
```

根据副作用命名

- 具有副作用的函数（变异函数）应使用动词或以form-为前缀的名词命名。
- 无副作用的函数（非变异函数）应使用名词或带有后缀

-ing或-ed的动词命名。

示例：变异函数：

```
print(value)
array.sort()                // 原地排序
list.add(value)             // 变异列表
set.formUnion(anotherSet)   // set 现在是 set 和 anotherSet 的并集
```

非变异函数：

```
let sortedArray = array.sorted()     // 非原地排序
let union = set.union(anotherSet)    // union 现在是 set 和另一个集合的并集
```

布尔函数或变量
涉及布尔值的语句应被视为断言。

Chapter 38: Style Conventions

Section 38.1: Fluent Usage

Using natural language
Functions calls should be close to natural English language.

Example:

```
list.insert(element, at: index)
```

instead of

```
list.insert(element, position: index)
```

Naming Factory Methods
Factory methods should begin with the prefix `make`.

Example:

```
factory.makeObject()
```

Naming Parameters in Initializers and Factory Methods
The name of the first argument should not be involved in naming a factory method or initializer.

Example:

```
factory.makeObject(key: value)
```

Instead of:

```
factory.makeObject(havingProperty: value)
```

Naming according to side effects

- Functions with side effects (mutating functions) should be named using verbs or nouns prefixed with `form-` .
- Functions without side effects (nonmutating functions) should be named using nouns or verbs with the suffix `-ing` or `-ed`.

Example: Mutating functions:

```
print(value)
array.sort()                // in place sorting
list.add(value)             // mutates list
set.formUnion(anotherSet)   // set is now the union of set and anotherSet
```

Nonmutating functions:

```
let sortedArray = array.sorted()     // out of place sorting
let union = set.union(anotherSet)    // union is now the union of set and another set
```

Boolean functions or variables
Statements involving booleans should read as assertions.

示例：

```
set.isEmpty
line.intersects(anotherLine)
```

命名协议

- 描述某物是什么的协议应使用名词命名。
- 描述能力的协议应以-able、-ible或-ing作为后缀。

示例：

```
Collection        // 描述某物是一个集合
ProgressReporting // 描述某物具有报告进度的能力
Equatable         // 描述某物具有与某物相等的能力
```

类型和属性
类型、变量和属性应读作名词。

示例：

```
let factory = ...
let list = [1, 2, 3, 4]
```

第38.2节：清晰的用法

避免歧义
类、结构体、函数和变量的名称应避免歧义。

示例：

```
extension List {
    public mutating func remove(at position: Index) -> Element {
        // 实现代码
    }
}
```

对该函数的调用将如下所示：

```
list.remove(at: 42)
```

通过这种方式，避免了歧义。如果函数调用只是list.remove(42)，则不清楚是要移除值等于42的元素，还是移除索引为42的元素。

避免冗余
函数名称不应包含冗余信息。

一个不好的例子是：

```
extension List {
    public mutating func removeElement(element: Element) -> Element? {
        // 实现代码
    }
}
```


Example:

```
set.isEmpty
line.intersects(anotherLine)
```

Naming Protocols

- Protocols describing what something is should be named using nouns.
- Protocols describing capabilities should have `-able`, `-ible` or `-ing` as suffix.

Example:

```
Collection        // describes that something is a collection
ProgressReporting // describes that something has the capability of reporting progress
Equatable         // describes that something has the capability of being equal to something
```

Types and Properties
Types, variables and properties should read as nouns.

Example:

```
let factory = ...
let list = [1, 2, 3, 4]
```

Section 38.2: Clear Usage

Avoid Ambiguity
The name of classes, structures, functions and variables should avoid ambiguity.

Example:

```
extension List {
    public mutating func remove(at position: Index) -> Element {
        // implementation
    }
}
```

The function call to this function will then look like this:

```
list.remove(at: 42)
```

This way, ambiguity is avoided. If the function call would be just `list.remove(42)` it would be unclear, if an Element equal to 42 would be removed or if the Element at Index 42 would be removed.

Avoid Redundancy
The name of functions should not contain redundant information.

A bad example would be:

```
extension List {
    public mutating func removeElement(element: Element) -> Element? {
        // implementation
    }
}
```

调用该函数可能看起来像`list.removeElement(someObject)`。变量`someObject`已经表明，一个元素被移除了。函数签名最好改成这样：

```
extension List {
    public mutating func remove(_ member: Element) -> Element? {
        // 实现代码
    }
}
```

调用该函数看起来像这样：list.remove(someObject)。

根据变量的角色命名

变量应根据其角色命名（例如供应者、问候语），而不是根据其类型（例如工厂、字符串等）命名

协议名称与变量名称之间的高耦合

如果类型名称在大多数情况下描述其角色（例如迭代器），则该类型应以后缀

`Type`命名。（例如 IteratorType）

使用弱类型参数时提供更多细节

如果对象的类型不能清楚地表明其在函数调用中的用途，则函数应以一个前置名词命名，用于描述每个弱类型参数的用途。

示例：

```
func addObserver(_ observer: NSObject, forKeyPath path: String)
```

调用看起来应为 `object.addObserver(self, forKeyPath: path)`，而不是

```
func add(_ observer: NSObject, for keyPath: String)
```

调用看起来应为 object.add(self, for: path)

第38.3节：大小写

类型与协议

类型和协议名称应以大写字母开头。

示例：

```
protocol Collection {}
struct String {}
class UIView {}
struct Int {}
enum Color {}
```

其他所有...

变量、常量、函数和枚举案例应以小写字母开头。

示例：

```
let greeting = "Hello"
let height = 42.0

enum Color {
```


A call to the function may look like `list.removeElement(someObject)`. The variable `someObject` already indicates, that an Element is removed. It would be better for the function signature to look like this:

```
extension List {
    public mutating func remove(_ member: Element) -> Element? {
        // implementation
    }
}
```

The call to this function looks like this: `list.remove(someObject)`.

Naming variables according to their role

Variables should be named by their role (e.g. supplier, greeting) instead of their type (e.g. factory, string, etc..)

High coupling between Protocol Name and Variable Names

If the name of the type describes its role in most cases (e.g. Iterator), the type should be named with the suffix `Type`. (e.g. IteratorType)

Provide additional details when using weakly typed parameters

If the type of an object does not indicate its usage in a function call clearly, the function should be named with a preceding noun for every weakly typed parameter, describing its usage.

Example:

```
func addObserver(_ observer: NSObject, forKeyPath path: String)
```

to which a call would look like `object.addObserver(self, forKeyPath: path)`

instead of

```
func add(_ observer: NSObject, for keyPath: String)
```

to which a call would look like `object.add(self, for: path)`

Section 38.3: Capitalization

Types & Protocols

Type and protocol names should start with an uppercase letter.

Example:

```
protocol Collection {}
struct String {}
class UIView {}
struct Int {}
enum Color {}
```

Everything else...

Variables, constants, functions and enumeration cases should start with a lowercase letter.

Example:

```
let greeting = "Hello"
let height = 42.0

enum Color {
```

```
    case red
    case green
    case blue
}

func print(_ string: String) {
    …
}
```

驼峰命名法：

所有命名应使用适当的驼峰命名法。类型/协议名称使用大驼峰命名法，其他所有名称使用小驼峰命名法。

大驼峰命名法：

```
protocol IteratorType { … }
```

小驼峰命名法：

```
let inputView = …
```

缩写

除非是常用缩写（例如 URL、ID），否则应避免使用缩写。如果使用缩写，所有字母应保持相同大小写。

示例：

```
let userID: UserID = …
let urlString: URLString = …
```



```
    case red
    case green
    case blue
}

func print(_ string: String) {
    ...
}
```

Camel Case:

All naming should use the appropriate camel case. Upper camel case for type/protocol names and lower camel case for everything else.

Upper Camel Case:

```
protocol IteratorType { ... }
```

Lower Camel Case:

```
let inputView = ...
```

Abbreviations

Abbreviations should be avoided unless commonly used (e.g. URL, ID). If an abbreviation is used, all letters should have the same case.

Example:

```
let userID: UserID = ...
let urlString: URLString = ...
```

第39章：Swift中的NSRegularExpression

第39.1节：扩展String以进行简单的模式匹配

```
extension String {
    func matchesPattern(pattern: String) -> Bool {
        do {
            let regex = try NSRegularExpression(pattern: pattern,
                                                options: NSRegularExpressionOptions(rawValue: 0))
            let range: NSRange = NSMakeRange(0, self.characters.count)
            let matches = regex.matchesInString(self, options: NSMatchingOptions(), range: range)
            return matches.count > 0
        } catch _ {
            返回 false
        }
    }
}

// 非常基础的示例 - 检查特定字符串
dump("Pinkman".matchesPattern("(White|Pinkman|Goodman|Schrader|Fring)"))

// 使用字符组检查发音相似的印象派画家
dump("Monet".matchesPattern("(M[oa]net)"))
dump("Manet".matchesPattern("(M[oa]net)"))
dump("Money".matchesPattern("(M[oa]net)"))      // 错误

// 检查姓氏是否在列表中
dump("Skyler White".matchesPattern("\\\\w+ (White|Pinkman|Goodman|Schrader|Fring)"))

// 检查字符串是否像英国股票代码
dump("VOD.L".matchesPattern("[A-Z]{2,3}\\\\.L"))
dump("BP.L".matchesPattern("[A-Z]{2,3}\\\\.L"))

// 检查整个字符串是否为可打印的ASCII字符
dump("tab\formatted text".matchesPattern("^[\\u{0020}-\\u{007e}]*$"))

// Unicode示例：检查字符串是否包含扑克牌花色
dump("♠".matchesPattern("[\\u{2660}-\\u{2667}]"))
dump("♡".matchesPattern("[\\u{2660}-\\u{2667}]"))
dump(" .matchesPattern("[\\u{2660}-\\u{2667}]"))     // 错误

// 注意：正则表达式需要Unicode转义字符
dump("♣".matchesPattern("♣"))            // 不起作用
```

下面是另一个示例，基于上述内容做一些有用的事情，这些事情很难通过其他方法实现，非常适合用正则表达式解决。

```
// 英国邮政编码的格式验证。
// 这只是简单检查格式是否像有效的英国邮政编码，不应对误报失败。

private func isPostcodeValid(postcode: String) -> Bool {
    return postcode.matchesPattern("^[A-Z]{1,2}([0-9][A-Z]|[0-9]{1,2})\\\\s[0-9][A-Z]{2}")
}

// 有效的模式（来自 https://en.wikipedia.org/wiki/Postcodes_in_the_United_Kingdom#Validation）
// 将返回 true
dump(isPostcodeValid("EC1A 1BB"))
dump(isPostcodeValid("W1A 0AX"))
dump(isPostcodeValid("M1 1AE"))
```


Chapter 39: NSRegularExpression in Swift

Section 39.1: Extending String to do simple pattern matching

```
extension String {
    func matchesPattern(pattern: String) -> Bool {
        do {
            let regex = try NSRegularExpression(pattern: pattern,
                                                options: NSRegularExpressionOptions(rawValue: 0))
            let range: NSRange = NSMakeRange(0, self.characters.count)
            let matches = regex.matchesInString(self, options: NSMatchingOptions(), range: range)
            return matches.count > 0
        } catch _ {
            return false
        }
    }
}

// very basic examples - check for specific strings
dump("Pinkman".matchesPattern("(White|Pinkman|Goodman|Schrader|Fring)"))

// using character groups to check for similar-sounding impressionist painters
dump("Monet".matchesPattern("(M[oa]net)"))
dump("Manet".matchesPattern("(M[oa]net)"))
dump("Money".matchesPattern("(M[oa]net)"))     // false

// check surname is in list
dump("Skyler White".matchesPattern("\\\\w+ (White|Pinkman|Goodman|Schrader|Fring)"))

// check if string looks like a UK stock ticker
dump("VOD.L".matchesPattern("[A-Z]{2,3}\\\\.L"))
dump("BP.L".matchesPattern("[A-Z]{2,3}\\\\.L"))

// check entire string is printable ASCII characters
dump("tab\\tformatted text".matchesPattern("^[\\u{0020}-\\u{007e}]*$"))

// Unicode example: check if string contains a playing card suit
dump("♠□".matchesPattern("[\\u{2660}-\\u{2667}]"))
dump("♡".matchesPattern("[\\u{2660}-\\u{2667}]"))
dump("□".matchesPattern("[\\u{2660}-\\u{2667}]"))    // false

// NOTE: regex needs Unicode-escaped characters
dump("♣□".matchesPattern("♣□"))           // does NOT work
```

Below is another example which builds on the above to do something useful, which can't easily be done by any other method and lends itself well to a regex solution.

```
// Pattern validation for a UK postcode.
// This simply checks that the format looks like a valid UK postcode and should not fail on false
positives.
private func isPostcodeValid(postcode: String) -> Bool {
    return postcode.matchesPattern("^[A-Z]{1,2}([0-9][A-Z]|[0-9]{1,2})\\\\s[0-9][A-Z]{2}")
}

// valid patterns (from https://en.wikipedia.org/wiki/Postcodes_in_the_United_Kingdom#Validation)
// will return true
dump(isPostcodeValid("EC1A 1BB"))
dump(isPostcodeValid("W1A 0AX"))
dump(isPostcodeValid("M1 1AE"))
```

```
dump(isPostcodeValid("B33 8TH"))
dump(isPostcodeValid("CR2 6XH"))
dump(isPostcodeValid("DN55 1PT"))

// 一些无效的模式
// 将返回 false
dump(isPostcodeValid("EC12A 1BB"))
dump(isPostcodeValid("CRB1 6XH"))
dump(isPostcodeValid("CR 6XH"))
```

第39.2节：基本用法

在 Swift 中实现正则表达式时需要考虑几个方面。

```
let letters = "abcdefg"
let pattern = "[a,b,c]"
let regEx = try NSRegularExpression(pattern: pattern, options: [])
let nsString = letters as NSString
let matches = regEx.matches(in: letters, options: [], range: NSMakeRange(0, nsString.length))
let output = matches.map {nsString.substring(with: $0.range)}
//output = ["a", "b", "c"]
```

为了获得支持所有字符类型的准确范围长度，输入字符串必须转换为NSString。

为了安全地匹配模式，应将匹配操作放在do catch块中以处理失败情况

```
let numbers = "121314"
let pattern = "1[2,3]"
do {
    let regEx = try NSRegularExpression(pattern: pattern, options: [])
    let nsString = numbers as NSString
    let matches = regEx.matches(in: numbers, options: [], range: NSMakeRange(0, nsString.length))
    let output = matches.map {nsString.substring(with: $0.range)}
    output
} catch let error as NSError {
    print("匹配失败")
}
//output = ["12", "13"]
```

正则表达式功能通常放在扩展或辅助工具中以分离关注点。

第39.3节：替换子字符串

模式可以用来替换输入字符串的一部分。

下面的示例将分币符号替换为美元符号。

```
var money = "¢¥€£$¥€£¢"
let pattern = "¢"
do {
    let regEx = try NSRegularExpression (pattern: pattern, options: [])
    let nsString = money as NSString
    let range = NSMakeRange(0, nsString.length)
    let correct$ = regEx.stringByReplacingMatches(in: money, options: .withTransparentBounds,
range: range, withTemplate: "$")
} catch let error as NSError {
    print("匹配失败")
```



```
dump(isPostcodeValid("B33 8TH"))
dump(isPostcodeValid("CR2 6XH"))
dump(isPostcodeValid("DN55 1PT"))

// some invalid patterns
// will return false
dump(isPostcodeValid("EC12A 1BB"))
dump(isPostcodeValid("CRB1 6XH"))
dump(isPostcodeValid("CR 6XH"))
```

Section 39.2: Basic Usage

There are several considerations when implementing Regular Expressions in Swift.

```
let letters = "abcdefg"
let pattern = "[a,b,c]"
let regEx = try NSRegularExpression(pattern: pattern, options: [])
let nsString = letters as NSString
let matches = regEx.matches(in: letters, options: [], range: NSMakeRange(0, nsString.length))
let output = matches.map {nsString.substring(with: $0.range)}
//output = ["a", "b", "c"]
```

In order to get an accurate range length that supports all character types the input string must be converted to a NSString.

For safety matching against a pattern should be enclosed in a do catch block to handle failure

```
let numbers = "121314"
let pattern = "1[2,3]"
do {
    let regEx = try NSRegularExpression(pattern: pattern, options: [])
    let nsString = numbers as NSString
    let matches = regEx.matches(in: numbers, options: [], range: NSMakeRange(0, nsString.length))
    let output = matches.map {nsString.substring(with: $0.range)}
    output
} catch let error as NSError {
    print("Matching failed")
}
//output = ["12", "13"]
```

Regular expression functionality is often put in an extension or helper to separate concerns.

Section 39.3: Replacing Substrings

Patterns can be used to replace part of an input string.

The example below replaces the cent symbol with the dollar symbol.

```
var money = "¢¥€£$¥€£¢"
let pattern = "¢"
do {
    let regEx = try NSRegularExpression (pattern: pattern, options: [])
    let nsString = money as NSString
    let range = NSMakeRange(0, nsString.length)
    let correct$ = regEx.stringByReplacingMatches(in: money, options: .withTransparentBounds,
range: range, withTemplate: "$")
} catch let error as NSError {
    print("Matching failed")
```

```
}
//correct$ = "$¥€£$¥€£$"
```

第39.4节：特殊字符

要匹配特殊字符，应使用双反斜杠 \. 变为 \\。

需要转义的字符包括

```
(){}[]/\+*$>.|^?
```

下面的示例获取三种类型的开括号

```
let specials = "(){}[]"
let pattern = "(\\(|\\{|\\[)"
do {
    let regEx = try NSRegularExpression(pattern: pattern, options: [])
    let nsString = specials as NSString
    let matches = regEx.matches(in: specials, options: [], range: NSMakeRange(0, nsString.length))
    let output = matches.map {nsString.substring(with: $0.range)}
} catch let error as NSError {
    print("匹配失败")
}
//output = ["(", "{", "["]
```

第39.5节：验证

正则表达式可以通过计算匹配次数来验证输入。

```
var validDate = false

let numbers = "35/12/2016"
let usPattern =  "^(0[1-9]|1[012])[-/.](0[1-9]|[12][0-9]|3[01])[-/.](19|20)\\d\\d$"
let ukPattern = "^(0[1-9]|[12][0-9]|3[01])[-/](0[1-9]|1[012])[-/](19|20)\\d\\d$"
do {
    let regEx = try NSRegularExpression(pattern: ukPattern, options: [])
    let nsString = numbers as NSString
    let matches = regEx.matches(in: numbers, options: [], range: NSMakeRange(0, nsString.length))

    if matches.count > 0 {
validDate = true
    }

    validDate

} catch let error as NSError {
    print("匹配失败")
}
//output = false
```

第39.6节：用于邮件验证的NSRegularExpression

```
func isValidEmail(email: String) -> Bool {

    let emailRegEx = "[A-Z0-9a-z._%+-]+@[A-Za-z0-9.-]+\\.[A-Za-z]{2,4}"

    let emailTest = NSPredicate(format:"SELF MATCHES %@", emailRegEx)
    return emailTest.evaluate(with: email)
```



```
}
//correct$ = "$¥€£$¥€£$"
```

Section 39.4: Special Characters

To match special characters Double Backslash should be used `\. becomes \\.`

Characters you'll have to escape include

```
(){}[]/\+*$>.|^?
```

The below example get three kinds of opening brackets

```
let specials = "(){}[]"
let pattern = "(\\(|\\{|\\[)"
do {
    let regEx = try NSRegularExpression(pattern: pattern, options: [])
    let nsString = specials as NSString
    let matches = regEx.matches(in: specials, options: [], range: NSMakeRange(0, nsString.length))
    let output = matches.map {nsString.substring(with: $0.range)}
} catch let error as NSError {
    print("Matching failed")
}
//output = ["(", "{", "["]
```

Section 39.5: Validation

Regular expressions can be used to validate inputs by counting the number of matches.

```
var validDate = false

let numbers = "35/12/2016"
let usPattern =  "^(0[1-9]|1[012])[-/.](0[1-9]|[12][0-9]|3[01])[-/.](19|20)\\d\\d$"
let ukPattern = "^(0[1-9]|[12][0-9]|3[01])[-/](0[1-9]|1[012])[-/](19|20)\\d\\d$"
do {
    let regEx = try NSRegularExpression(pattern: ukPattern, options: [])
    let nsString = numbers as NSString
    let matches = regEx.matches(in: numbers, options: [], range: NSMakeRange(0, nsString.length))

    if matches.count > 0 {
        validDate = true
    }

    validDate

} catch let error as NSError {
    print("Matching failed")
}
//output = false
```

Section 39.6: NSRegularExpression for mail validation

```
func isValidEmail(email: String) -> Bool {

    let emailRegEx = "[A-Z0-9a-z._%+-]+@[A-Za-z0-9.-]+\\.[A-Za-z]{2,4}"

    let emailTest = NSPredicate(format:"SELF MATCHES %@", emailRegEx)
    return emailTest.evaluate(with: email)
```

```
}
```

或者你也可以像这样使用 String 扩展：

```
extension String
{
    func isValidEmail() -> Bool {

        let emailRegEx = "[A-Z0-9a-z._%+-]+@[A-Za-z0-9.-]+\\.[A-Za-z]{2,4}"

        let emailTest = NSPredicate(format:"SELF MATCHES %@", emailRegEx)
        return emailTest.evaluate(with: self)
    }
}
```



```
}
```

or you could use String extension like this:

```
extension String
{
    func isValidEmail() -> Bool {

        let emailRegEx = "[A-Z0-9a-z._%+-]+@[A-Za-z0-9.-]+\\.[A-Za-z]{2,4}"

        let emailTest = NSPredicate(format:"SELF MATCHES %@", emailRegEx)
        return emailTest.evaluate(with: self)
    }
}
```

第40章：RxSwift

第40.1节：释放

订阅创建后，正确管理其释放非常重要。

文档告诉我们

> 如果序列在有限时间内终止，不调用 dispose 或不使用 addDisposableTo(disposeBag)
> 不会导致永久的资源泄漏。然而，这些资源会一直被使用，直到序列
> 完成，要么通过生成完所有元素，要么返回错误。

有两种释放资源的方法。

1. 使用disposeBag和addDisposableTo操作符。
2. 使用takeUntil操作符。

在第一种情况下，你需要手动将订阅传递给DisposeBag对象，这样可以正确清理所有占用的内存。

```
let bag = DisposeBag()
Observable.just(1).subscribeNext {
    print($0)
}.addDisposableTo(bag)
```

实际上你不需要在每个创建的类中都创建DisposeBag，只需查看*RxSwift Community*的项目NSObject+Rx。使用该框架，上述代码可以重写为：

```
Observable.just(1).subscribeNext {
    print($0)
}.addDisposableTo(rx_disposeBag)
```

在第二种情况下，如果订阅时间与self对象的生命周期一致，可以使用takeUntil(rx_deallocated)来实现自动释放：

```
let _ = sequence
    .takeUntil(rx_deallocated)
    .subscribe {
        print($0)
    }
```

第40.2节：RxSwift基础

FRP，即函数响应式编程，有一些你需要了解的基本术语。

每一条数据都可以表示为Observable，它是一个异步数据流。FRP的强大之处在于将同步和异步事件表示为流，Observable，并提供相同的接口来操作它们。

通常Observable包含若干（或无）事件，这些事件包含数据——.Next事件，然后它可以成功终止（.Success）或因错误终止（.Error）。

让我们来看下面的弹珠图：

Chapter 40: RxSwift

Section 40.1: Disposing

After the subscription was created, it is important to manage its correct deallocation.

The docs told us that

> If a sequence terminates in finite time, not calling dispose or not using addDisposableTo(disposeBag)
> won't cause any permanent resource leaks. However, those resources will be used until the sequence
> completes, either by finishing production of elements or returning an error.

There are two ways of deallocate resources.

1. Using `disposeBags` and `addDisposableTo` operator.
2. Using `takeUntil` operator.

In the first case you manually pass the subscription to the `DisposeBag` object, which correctly clears all taken memory.

```
let bag = DisposeBag()
Observable.just(1).subscribeNext {
    print($0)
}.addDisposableTo(bag)
```

You don't actually need to create `DisposeBags` in every class that you create, just take a look at *RxSwift Community*'s project named NSObject+Rx. Using the framework the code above can be rewritten as follows:

```
Observable.just(1).subscribeNext {
    print($0)
}.addDisposableTo(rx_disposeBag)
```

In the second case, if the subscription time coincides with the `self` object lifetime, it is possible to implement disposing using `takeUntil(rx_deallocated)`:

```
let _ = sequence
    .takeUntil(rx_deallocated)
    .subscribe {
        print($0)
    }
```

Section 40.2: RxSwift basics

FRP, or Functional Reactive Programming, has some basic terms which you need to know.

Every piece of data can be represented as `Observable`, which is an asynchronous data stream. The power of FRP is in representation synchronous and asynchronous events as streams, `Observables`, and providing the same interface to work with it.

Usually `Observable` holds several (or none) events that holds the date - `.Next` events, and then it can be terminated successfully (`.Success`) or with an error (`.Error`).

Let's take a look at following marble diagram:

```
--(1)--(2)--(3)|-->
```

在这个例子中，有一个Int值的流。随着时间推移，发生了三个.Next事件，然后流成功终止。

```
--X->
```

上图显示了一个没有发出数据且由.Error事件终止Observable的情况。

在继续之前，这里有一些有用的资源：

1. RxSwift。查看示例，阅读文档并开始入门。
2. RxSwift Slack 群组 有几个用于教育和问题解决的频道。
3. 尝试使用RxMarbles，了解各个操作符的作用，以及哪个在你的情况下最有用。
4. 看看这个示例，自己探索代码。

第40.3节：创建可观察对象

RxSwift 提供了多种创建Observable的方法，我们来看一下：

```
import RxSwift

let intObservable = Observable.just(123) // Observable<Int>
let stringObservable = Observable.just("RxSwift") // Observable<String>
let doubleObservable = Observable.just(3.14) // Observable<Double>
```

所以，可观察对象已经创建。它们只包含一个值，然后以成功结束。然而，创建后没有任何发生。为什么？

使用Observable有两个步骤：你先**观察**某些东西以*创建*一个流，然后你**订阅**该流或将其**绑定**到某物以*交互*。

```
Observable.just(12).subscribe {
    print($0)
}
```

控制台将打印：

```
.Next(12)
.Completed()
```

如果我只对处理发生在.Next事件中的数据感兴趣，我会使用subscribeNext操作符：

```
Observable.just(12).subscribeNext {
    print($0) // 现在打印“12”
}
```

如果我想创建一个包含多个值的observable，我会使用不同的操作符：

```
Observable.of(1,2,3,4,5).subscribeNext {
    print($0)
}
// 1
// 2
```



```
--(1)--(2)--(3)|-->
```

In this example there is a stream of Int values. As time moves forward, three .Next events occurred, and then the stream terminated successfully.

```
--X->
```

The diagram above shows a case where no data was emitted and .Error event terminates the Observable.

Before we move on, there are some useful resources:

1. RxSwift. Look at examples, read docs and getting started.
2. RxSwift Slack room has a few channels for education problem solving.
3. Play around with RxMarbles to know what operator does, and which is the most useful in your case.
4. Take a look on this example, explore the code by yourself.

Section 40.3: Creating observables

RxSwift offers many ways to create an Observable, let's take a look:

```
import RxSwift

let intObservale = Observable.just(123) // Observable<Int>
let stringObservale = Observable.just("RxSwift") // Observable<String>
let doubleObservale = Observable.just(3.14) // Observable<Double>
```

So, the observables are created. They holds just one value and then terminates with success. Nevertheless, nothing happening after it was created. Why?

There are two steps in working with Observables: you **observe** something to *create* a stream and then you **subscribe** to the stream or **bind** it to something to *interact* with it.

```
Observable.just(12).subscribe {
    print($0)
}
```

The console will print:

```
.Next(12)
.Completed()
```

And if I interested only in working with data, which take place in .Next events, I'd use subscribeNext operator:

```
Observable.just(12).subscribeNext {
    print($0) // prints "12" now
}
```

If I want create an observable of many values, I use different operators:

```
Observable.of(1,2,3,4,5).subscribeNext {
    print($0)
}
// 1
// 2
```

```
// 3
// 4
// 5

// 我也可以将现有的数据类型表示为Observables：
[1,2,3,4,5].asObservable().subscribeNext {
    print($0)
}
// 结果与之前相同。
```

最后，也许我想要一个Observable来执行一些工作。例如，将网络
操作封装成Observable<SomeResultType>是很方便的。让我们看看如何实现这一点：

```
Observable.create { observer in     // 创建一个Observable ...
MyNetworkService.doSomeWorkWithCompletion { (result, error) in
        if let e = error {
observer.onError(e)     // ..包含一个错误
        } else {
observer.onNext(result) // ..或者发出数据
            observer.onCompleted()  // ..并成功终止。
        }
    }
    return NopDisposable.instance // 这里你可以手动释放任何资源
                                  //如果这个observable被释放的话。
}
```

第40.4节：绑定

```
Observable.combineLatest(firstName.rx_text, lastName.rx_text) { $0 + " " + $1 }
.map { "问候，\($0)" }
.绑定到(greetingLabel.rx_text)
```

使用combineLatest操作符，每当两个Observable中的任意一个发出一个元素时，结合每个Observable最近发出的元素。因此，我们以这种方式结合两个UITextField的结果，创建一个新的消息，文本为"Greetings, \($0)"，使用字符串插值，随后绑定到UILabel的文本。

我们可以非常简单地将数据绑定到任何UITableView和UICollectionView：

```
viewModel
.rows
.bindTo(resultsTableView.rx_itemsWithCellIdentifier("WikipediaSearchCell", cellType:
WikipediaSearchCell.self)) { (_, viewModel, cell)
    cell.title = viewModel.title
cell.url = viewModel.url
}
.addDisposableTo(disposeBag)
```

这是对cellForRowAtIndexPath数据源方法的Rx封装。同时，Rx还负责实现numberOfRowsAtIndexPath方法，传统上这是一个必需实现的方法，但在这里你不必实现它，它已经被处理好了。

第40.5节：RxCocoa与ControlEvents

RxSwift 不仅提供了控制数据的方法，还能以响应式的方式表示用户操作。

RxCocoa 包含了你所需的一切。它将大多数 UI 组件的属性封装成Observable，但实际上并非完全如此。有一些升级版的Observable，称为ControlEvent（表示事件）和


```
// 3
// 4
// 5

// I can represent existing data types as Observables also:
[1,2,3,4,5].asObservable().subscribeNext {
    print($0)
}
// result is the same as before.
```

And finally, maybe I want an Observable that does some work. For example, it is convenient to wrap a network operation into Observable<SomeResultType>. Let's take a look of do one can achieve this:

```
Observable.create { observer in    // create an Observable ...
    MyNetworkService.doSomeWorkWithCompletion { (result, error) in
        if let e = error {
            observer.onError(e)    // ..that either holds an error
        } else {
            observer.onNext(result) // ..or emits the data
            observer.onCompleted()  // ..and terminates successfully.
        }
    }
    return NopDisposable.instance // here you can manually free any resources
                                  //in case if this observable is being disposed.
}
```

Section 40.4: Bindings

```
Observable.combineLatest(firstName.rx_text, lastName.rx_text) { $0 + " " + $1 }
.map { "Greetings, \($0)" }
.bindTo(greetingLabel.rx_text)
```

Using the combineLatest operator every time an item is emitted by either of two Observables, combine the latest item emitted by each Observable. So in this way we combine the result of the two UITextField's creating a new message with the text "Greetings, \($0)" using string interpolation to later bind to the text of a UILabel.

We can bind data to any UITableView and UICollectionView in an very easy way:

```
viewModel
.rows
.bindTo(resultsTableView.rx_itemsWithCellIdentifier("WikipediaSearchCell", cellType:
WikipediaSearchCell.self)) { (_, viewModel, cell) in
    cell.title = viewModel.title
    cell.url = viewModel.url
}
.addDisposableTo(disposeBag)
```

That's an Rx wrapper around the cellForRowAtIndexPath data source method. And also Rx takes care of the implementation of the numberOfRowsAtIndexPath, which is a required method in a traditional sense, but you don't have to implement it here, it's taken care of.

Section 40.5: RxCocoa and ControlEvents

RxSwift provides not only the ways to control your data, but to represent user actions in a reactive way also.

RxCocoa contains everything you need. It wraps most of the UI components' properties into Observables, but not really. There are some upgraded Observables called ControlEvents (which represent events) and

ControlProperties（表示属性，惊喜！）。这些东西在底层持有Observable流，但也有一些细节：

- 它永远不会失败，因此没有错误。
- 当控件被释放时，它会Complete序列。
- 它在主线程（MainScheduler.instance）上发送事件。

基本上，你可以像平常一样使用它们：

```
button.rx_tap.subscribeNext { _ in    // 控件事件
    print("用户点击了按钮！")
}.addDisposableTo(bag)

textField.rx_text.subscribeNext { text in // 控件属性
    print("文本框内容是：\(text)")
}.addDisposableTo(bag)
// 注意 ControlProperty 在订阅时会生成 .Next 事件
// 在这种情况下，日志将显示
// “文本字段包含：”
// 在应用程序的最开始。
```

这非常重要：只要你使用Rx，就忘掉@IBAction那些东西，所有你需要的都可以一次性绑定和配置。例如，你的视图控制器的viewDidLoad方法是描述UI组件如何工作的好地方。

好的，另一个例子：假设我们有一个文本字段、一个按钮和一个标签。我们想要**验证文本**在文本字段中当我们**点击**按钮时，并且**显示**结果在标签中。是的，看起来又是一个验证邮箱的任务，对吧？

首先，我们获取button.rx_tap ControlEvent：

```
----()----------------------()----->
```

这里空括号表示用户点击。接下来，我们用withLatestFrom操作符获取文本字段中写的内容（看看这里，想象上面的流代表用户点击，下面的流代表文本字段中的文本）。

```
button.rx_tap.withLatestFrom(textField.rx_text)

----("")-------------------("123")--->
//  ^ 点击    ^ 我输入了123     ^ 点击
```

很好，我们有一个字符串流用于验证，只有在需要验证时才发出。

任何Observable都有类似的操作符，比如map或filter，我们将使用map来验证文本。自己创建validateEmail函数，使用你想要的任何正则表达式。

```
button.rx_tap                                // ControlEvent<Void>
        .withLatestFrom(textField.rx_text)   // Observable<String>
        .map(validateEmail)                  // Observable<Bool>
        .map { (isCorrect) in
            return isCorrect ? "邮箱正确" : "请输入正确的邮箱"
        }                                    // Observable<String>
        .bindTo(label.rx_text)
        .addDisposableTo(bag)
```

完成！如果你需要更多自定义逻辑（比如在出错时显示错误视图，成功时跳转到另一个

ControlProperties (which represent properties, surprise!). These things holds Observable streams under the hood, but also have some nuances:

- It never fails, so no errors.
- It will Complete sequence on control being deallocated.
- It delivers events on the main thread (MainScheduler.instance).

Basically, you can work with them as usual:

```
button.rx_tap.subscribeNext { _ in   // control event
    print("User tapped the button!")
}.addDisposableTo(bag)

textField.rx_text.subscribeNext { text in // control property
    print("The textfield contains: \(text)")
}.addDisposableTo(bag)
// notice that ControlProperty generates .Next event on subscription
// In this case, the log will display
// "The textfield contains: "
// at the very start of the app.
```

This is very important to use: as long as you use Rx, forget about the @IBAction stuff, everything you need you can bind and configure at once. For example, viewDidLoad method of your view controller is a good candidate to describe how the UI-components work.

Ok, another example: suppose we have a textfield, a button, and a label. We want to **validate text** in the textfield when we **tap** the button, and **display** the results in the label. Yep, seems like an another validate-email task, huh?

First of all, we grab the button.rx_tap ControlEvent:

```
----()----------------------()----->
```

Here empty parenthesis show user taps. Next, we take what's written in the textField with withLatestFrom operator (take a look at it here, imagine that upper stream represents user taps, bottom one represents text in the text field).

```
button.rx_tap.withLatestFrom(textField.rx_text)

----("")-------------------("123")--->
//  ^ tap    ^ i wrote 123     ^ tap
```

Nice, we have a stream of strings to validate, emitted only when we need to validate.

Any Observable has such familiar operators as map or filter, we'll take map to validate the text. Create validateEmail function yourself, use any regex you want.

```
button.rx_tap                                // ControlEvent<Void>
        .withLatestFrom(textField.rx_text)   // Observable<String>
        .map(validateEmail)                  // Observable<Bool>
        .map { (isCorrect) in
            return isCorrect ? "Email is correct" : "Input the correct one, please"
        }                                    // Observable<String>
        .bindTo(label.rx_text)
        .addDisposableTo(bag)
```

Done! If you need more custom logic (like showing error views in case of error, making a transition to another

界面……），只需订阅最终的Bool流并在那里编写即可。

screen on success…), just subscribe to the final `Bool` stream and write it there.

第41章：Swift包管理器

第41.1节：创建和使用简单的Swift包

要创建Swift包，打开终端然后创建一个空文件夹：

mkdir AwesomeProject cd AwesomeProject

并初始化一个Git仓库：

git init

然后创建软件包本身。可以手动创建软件包结构，但有一种使用命令行界面（CLI）命令的简单方法。

如果你想创建一个可执行文件：

swift package init --type executable

将生成多个文件。其中，main.swift将作为你的应用程序的入口点。

如果你想创建一个库：

swift package init --type library

生成的AwesomeProject.swift文件将作为该库的主文件。

在这两种情况下，你都可以在Sources文件夹中添加其他Swift文件（适用常规的访问控制规则）。

Package.swift文件本身将自动填充以下内容：

```
import PackageDescription

let package = Package(
    name: "AwesomeProject"
)
```

包的版本管理通过Git标签完成：

git tag '1.0.0'

一旦推送到远程或本地 Git 仓库，您的包将可供其他项目使用。

您的包现在已准备好进行编译：

swift build

编译后的项目将位于.build/debug文件夹中。

您自己的包也可以解析对其他包的依赖。例如，如果您想在自己的项目中包含"SomeOtherPackage"，请修改您的*Package.swift*文件以包含该依赖：

```
import PackageDescription

let package = Package(
    name: "AwesomeProject",
    targets: [],
```


Chapter 41: Swift Package Manager

Section 41.1: Creation and usage of a simple Swift package

To create a Swift Package, open a Terminal then create an empty folder:

mkdir AwesomeProject cd AwesomeProject

And init a Git repository:

git init

Then create the package itself. One could create the package structure manually but there's a simple way using the CLI command.

If you want to make an executable:

swift package init --type executable

Several files will be generated. Among them, *main.swift* will be the entry point for your application.

If you want to make a library:

swift package init --type library

The generated *AwesomeProject.swift* file will be used as the main file for this library.

In both cases you can add other Swift files in the *Sources* folder (usual rules for access control apply).

The *Package.swift* file itself will be automatically populated with this content:

```
import PackageDescription

let package = Package(
    name: "AwesomeProject"
)
```

Versioning the package is done with Git tags:

git tag '1.0.0'

Once pushed to a remote or local Git repository, your package will be available to other projects.

Your package is now ready to be compiled:

swift build

The compiled project will be available in the *.build/debug* folder.

Your own package can also resolve dependencies to other packages. For example, if you want to include "SomeOtherPackage" in your own project, change your *Package.swift* file to include the dependency:

```
import PackageDescription

let package = Package(
    name: "AwesomeProject",
    targets: [],
```

```
    dependencies: [
.Package(url: "https://github.com/someUser/SomeOtherPackage.git",
                 majorVersion: 1),
    ]
)
```

然后重新构建你的项目：Swift 包管理器将自动解析、下载并构建依赖项。


```
    dependencies: [
        .Package(url: "https://github.com/someUser/SomeOtherPackage.git",
                 majorVersion: 1),
    ]
)
```

Then build your project again: the Swift Package Manager will automatically resolve, download and build the dependencies.

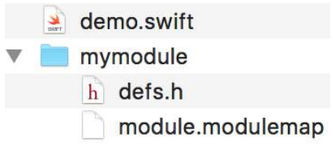
第42章：与C和Objective-

C协作

第42.1节：使用模块映射导入C头文件

一个模块映射可以简单地导入mymodule，通过配置它读取C头文件并使其表现为Swift函数。

在名为mymodule的目录中放置一个名为module.modulemap的文件：



在模块映射文件中：

```
// mymodule/module.modulemap
module mymodule {
header "defs.h"
}
```

然后导入该模块：

```
// demo.swift
import mymodule
print("空颜色: \(Color())")
```

使用-I 目录标志告诉 swiftc 去哪里查找模块：

```
swiftc -I . demo.swift    # "-I ." 意味着 "在当前目录中搜索模块"
```

有关模块映射语法的更多信息，请参阅 Clang 文档关于模块映射。

第42.2节：从Swift代码中使用Objective-C类

如果 MyFramework 在其公共头文件（以及伞形头文件）中包含Objective-C类，那么 import MyFramework 就足以在Swift中使用它们。

桥接头文件

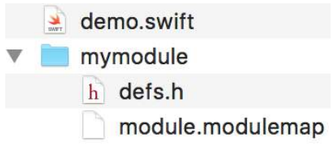
桥接头文件 使额外的Objective-C和C声明对Swift代码可见。添加项目文件时，Xcode 可能会自动提供创建桥接头文件的选项：

Chapter 42: Working with C and Objective-C

Section 42.1: Use a module map to import C headers

A **module map** can simply `import mymodule` by configuring it to read C header files and make them appear as Swift functions.

Place a file named `module.modulemap` inside a directory named `mymodule`:



Inside the module map file:

```
// mymodule/module.modulemap
module mymodule {
    header "defs.h"
}
```

Then `import` the module:

```
// demo.swift
import mymodule
print("Empty color: \(Color())")
```

Use the `-I` ctory flag to tell `swiftc` where to find the module:

```
swiftc -I . demo.swift    # "-I ." means "search for modules in the current directory"
```

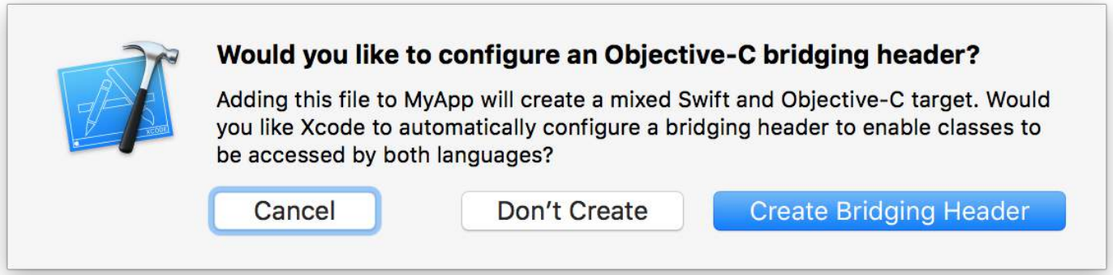
For more information about the module map syntax, see the Clang documentation about module maps.

Section 42.2: Using Objective-C classes from Swift code

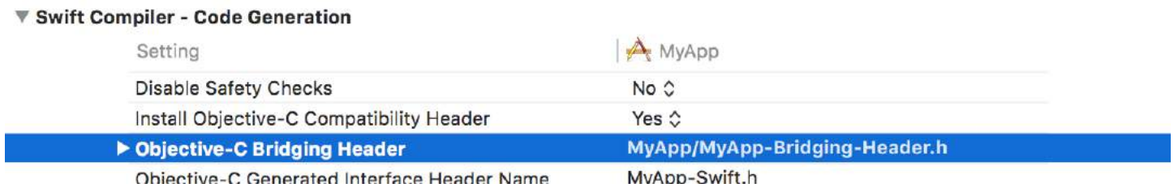
If MyFramework contains Objective-C classes in its public headers (and the umbrella header), then **`import MyFramework`** is all that's necessary to use them from Swift.

Bridging headers

A **bridging header** makes additional Objective-C and C declarations visible to Swift code. When adding project files, Xcode may offer to create a bridging header automatically:



要手动创建一个，请修改Objective-C Bridging Header构建设置：

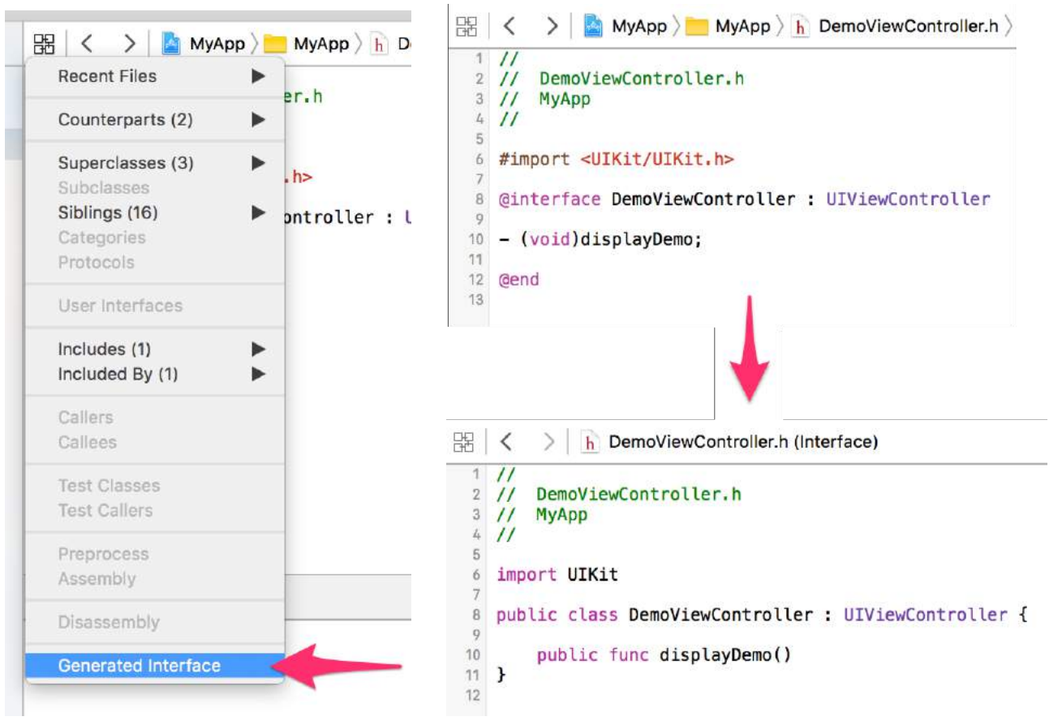


在桥接头文件中，导入代码中需要使用的文件：

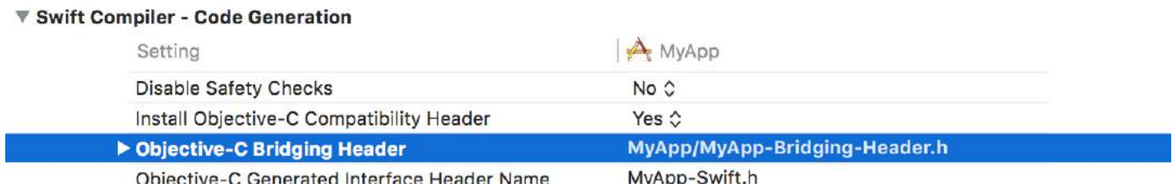
```
// MyApp-Bridging-Header.h
#import "MyClass.h"  // 允许此模块中的代码使用 MyClass
```

生成的接口

点击相关项目按钮（或按⌃1），然后选择生成的接口以查看将从 Objective-C 头文件生成的 Swift 接口。



To create one manually, modify the **Objective-C Bridging Header** build setting:

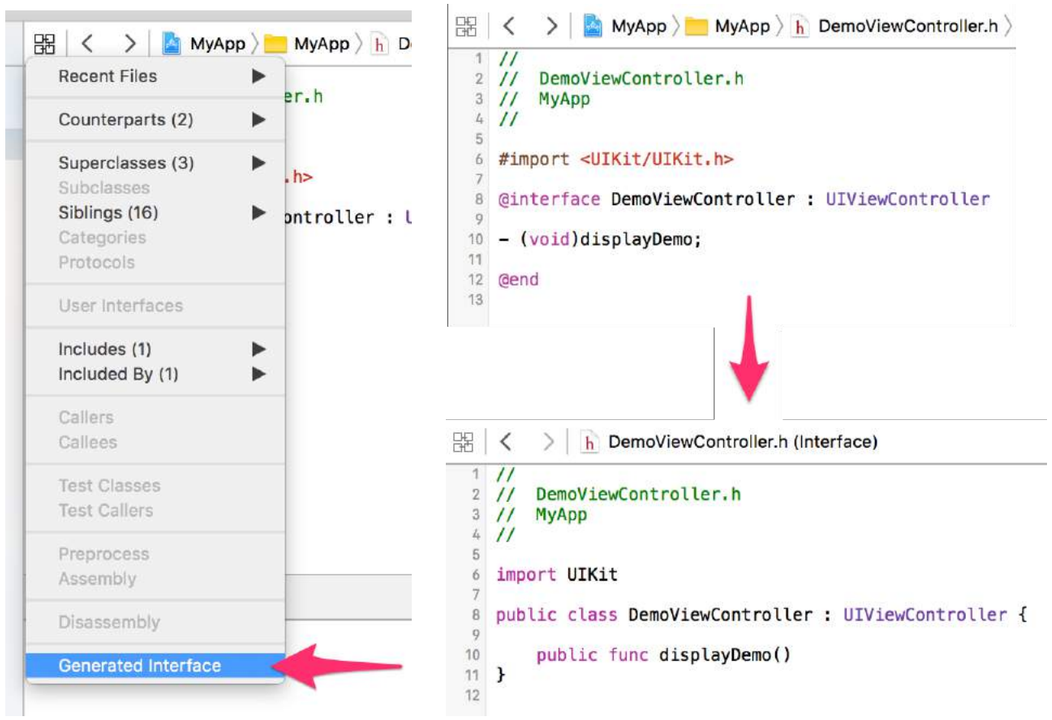


Inside the bridging header, import whichever files are necessary to use from code:

```
// MyApp-Bridging-Header.h
#import "MyClass.h"  // allows code in this module to use MyClass
```

Generated Interface

Click the Related Items button (or press ⌃1), then select **Generated Interface** to see the Swift interface that will be generated from an Objective-C header.



第42.3节：为 swiftc 指定桥接头文件

-import-objc-header 标志指定了 swiftc 要导入的头文件：

```
// defs.h
struct Color {
    int red, green, blue;
};

#define MAX_VALUE 255
```

```
// demo.swift
extension Color: CustomStringConvertible {  // 对C结构体的扩展
    public var description: String {
        return "Color(red: \(red), green: \(green), blue: \(blue))"
    }
}
print("MAX_VALUE is: \(MAX_VALUE)")  // C宏变成常量
let color = Color(red: 0xCA, green: 0xCA, blue: 0xD0)  // C结构体初始化器
print("The color is \(color)")
```

$ swiftc demo.swift -import-objc-header defs.h && ./demo MAX_VALUE is: 255 The color is Color(red: 202, green: 202, blue: 208)

第42.4节：使用C标准库

Swift的C互操作性允许你使用来自C标准库的函数和类型。

在Linux上，C标准库通过Glibc模块暴露；在苹果平台上称为Darwin。

```
#if os(macOS) || os(iOS) || os(tvOS) || os(watchOS)
import Darwin
#elseif os(Linux)
import Glibc
#endif

// 使用 open()、read() 以及其他 libc 功能
```

第42.5节：Objective-C 与 Swift 之间的细粒度互操作

当一个 API 被标记为NS_REFINED_FOR_SWIFT时，导入到 Swift 时会加上两个下划线前缀（__）：

```
@interface MyClass : NSObject
- (NSInteger)indexOfObject:(id)obj
```

EFINED_FOR_SWIFT ; @end

生成的接口如下所示：

```
public class MyClass : NSObject {
    public func
```

dexOfObject (obj: AnyObject) -> Int }

现在你可以用更“Swift 风格”的扩展替换该 API。在这种情况下，我们可以使用可选返回值，过滤掉NSNotFound：

Section 42.3: Specify a bridging header to swiftc

The `-import-objc-header` flag specifies a header for `swiftc` to import:

```
// defs.h
struct Color {
    int red, green, blue;
};

#define MAX_VALUE 255
```

```
// demo.swift
extension Color: CustomStringConvertible {  // extension on a C struct
    public var description: String {
        return "Color(red: \(red), green: \(green), blue: \(blue))"
    }
}
print("MAX_VALUE is: \(MAX_VALUE)")  // C macro becomes a constant
let color = Color(red: 0xCA, green: 0xCA, blue: 0xD0)  // C struct initializer
print("The color is \(color)")
```

$ swiftc demo.swift **-import-objc-header defs.h** && ./demo MAX_VALUE is: 255 The color is Color(red: 202, green: 202, blue: 208)

Section 42.4: Use the C standard library

Swift's C interoperability allows you to use functions and types from the C standard library.

On Linux, the C standard library is exposed via the `Glibc` module; on Apple platforms it's called `Darwin`.

```
#if os(macOS) || os(iOS) || os(tvOS) || os(watchOS)
import Darwin
#elseif os(Linux)
import Glibc
#endif

// use open(), read(), and other libc features
```

Section 42.5: Fine-grained interoperation between Objective-C and Swift

When an API is marked with `NS_REFINED_FOR_SWIFT`, it will be prefixed with two underscores (`__`) when imported to Swift:

```
@interface MyClass : NSObject
- (NSInteger)indexOfObject:(id)obj
```

EFINED_FOR_SWIFT; @end

The generated interface looks like this:

```
public class MyClass : NSObject {
    public func
```

dexOfObject(obj: AnyObject) -> Int }

Now you can **replace the API** with a more "Swifty" extension. In this case, we can use an optional return value, filtering out NSNotFound:

```
extension MyClass {
    // 与其在对象不存在时返回 NSNotFound，
    // 这个“改进”的 API 返回 nil。
    func
```

xOfObject(obj: AnyObject) -> Int? { let idx = __indexOfObject(obj) if idx == NSNotFound { return nil } return idx } } // Swift 代码，使用“if let”方式： let myobj = MyClass() if let idx = myobj.indexOfObject(something) { // 使用 idx 做某些操作 }

在大多数情况下，你可能想限制 Objective-C 函数的参数是否可以为 nil。 这通过使用 _Nonnull 关键字来实现，该关键字限定任何指针或 block 引用：

```
void
doStuff(const void *const _Nonnull data, void (^_Nonnull completion)())
{
    // 复杂的异步代码
}
```

写好后，编译器在我们尝试从 Swift 代码向该函数传递 nil 时会报错：

```
doStuff(
    nil,  // 错误：nil 与预期的参数类型 'UnsafeRawPointer' 不兼容
    nil)  // 错误：nil 与预期的参数类型 '() -> Void' 不兼容
```

与_Nonnull相反的是_Nullable，表示在此参数中可以接受传入 nil。_Nullable也是默认值；然而，显式指定它可以使代码更具自我说明性和未来兼容性。

为了进一步帮助编译器优化你的代码，你可能还想指定该块是否为逃逸（escaping）：

```
void
callNow(__attribute__((noescape)) void (^_Nonnull f)())
{
    // f不会被存储在任何地方
}
```

通过这个属性，我们承诺不会保存该块的引用，也不会在函数执行结束后调用该块。

第42.6节：从Objective-C代码中使用Swift类

在同一模块内

在名为“MyModule”的模块内，Xcode会生成一个名为MyModule-Swift.h的头文件，用于向Objective-C公开公共的Swift类。导入此头文件以使用Swift类：

```
// MySwiftClass.swift 在 MyApp 中
import Foundation

// 该类必须是`public`才能被看到，除非该目标也有桥接头文件
```

ic class MySwiftClass: NSObject { // ... }

```
// MyViewController.m 在 MyApp

#import "MyViewController.h"
```

ort "MyApp-Swift.h" // 导入生成的接口 **#import <MyFramework/MyFramework-Swift.h>** // 或者使用尖括号用于框架目标 @implementation MyViewController - (void)demo { [[MySwiftClass alloc] init]; // 使用 Swift 类 } @end


```
extension MyClass {
    // Rather than returning NSNotFound if the object doesn't exist,
    // this "refined" API returns nil.
    func
```

xOfObject(obj: AnyObject) -> **Int?** { let idx = __indexOfObject(obj) if idx == NSNotFound { return nil } return idx } } // Swift code, using "if let" as it should be: let myobj = MyClass() if let idx = myobj.indexOfObject(something) { // do something with idx }

In most cases you might want to restrict whether or not an argument to an Objective-C function could be `nil`. This is done using `_Nonnull` keyword, which qualifies any pointer or block reference:

```
void
doStuff(const void *const _Nonnull data, void (^_Nonnull completion)())
{
    // complex asynchronous code
}
```

With that written, the compiler shall emit an error whenever we try to pass `nil` to that function from our Swift code:

```
doStuff(
    nil,  // error: nil is not compatible with expected argument type 'UnsafeRawPointer'
    nil)  // error: nil is not compatible with expected argument type '() -> Void'
```

The opposite of `_Nonnull` is `_Nullable`, which means that it is acceptable to pass `nil` in this argument. `_Nullable` is also the default; however, specifying it explicitly allows for more self-documented and future-proof code.

To further help the compiler with optimising your code, you also might want to specify if the block is escaping:

```
void
callNow(__attribute__((noescape)) void (^_Nonnull f)())
{
    // f is not stored anywhere
}
```

With this attribute we promise not to save the block reference and not to call the block after the function has finished execution.

Section 42.6: Using Swift classes from Objective-C code

In the same module

Inside a module named "**MyModule**", Xcode generates a header named **`MyModule-Swift.h`** which exposes public Swift classes to Objective-C. Import this header in order to use the Swift classes:

```
// MySwiftClass.swift in MyApp
import Foundation

// The class must be `public` to be visible, unless this target also has a bridging header
```

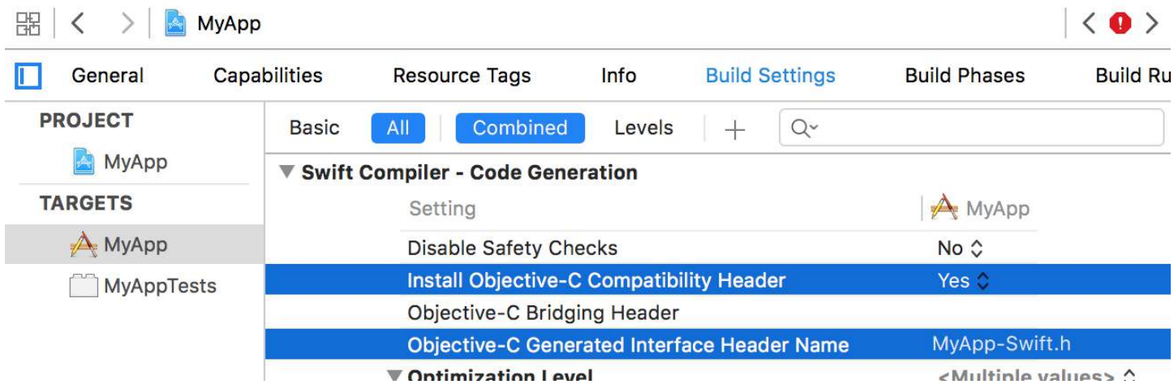
ic class MySwiftClass: NSObject { // ... }

```
// MyViewController.m in MyApp

#import "MyViewController.h"
```

ort "MyApp-Swift.h" // import the generated interface **#import <MyFramework/MyFramework-Swift.h>** // or use angle brackets for a framework target @implementation MyViewController - (void)demo { [[MySwiftClass alloc] init]; // use the Swift class } @end

相关的构建设置：

- Objective-C 生成的接口头文件名称：控制生成的 Obj-C 头文件的名称。
- **安装 Objective-C 兼容性头文件**: 是否将 -Swift.h 头文件设为公共头文件（针对框架目标）。

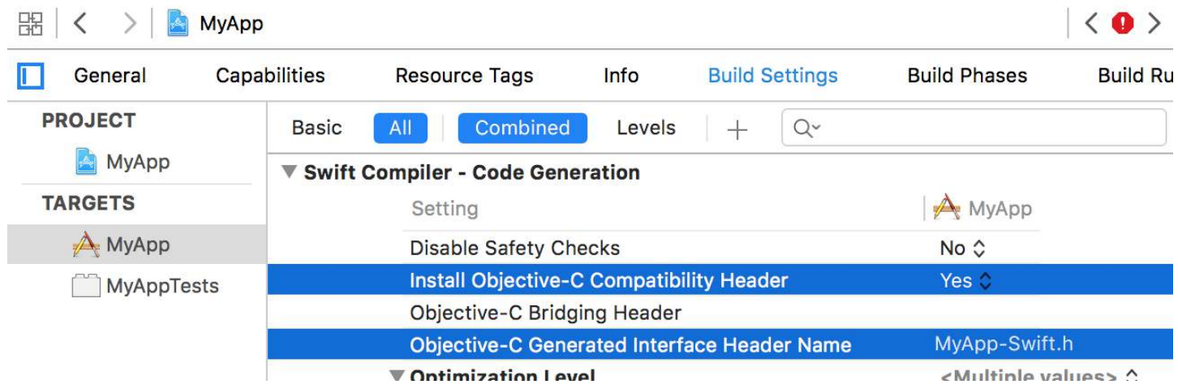


在另一个模块中

使用 `@import MyFramework;` 导入整个模块，包括 Swift 类的 Obj-C 接口（如果上述构建设置已启用）。

Relevant build settings:

- **Objective-C Generated Interface Header Name**: controls the name of the generated Obj-C header.
- **Install Objective-C Compatibility Header**: whether the -Swift.h header should be a public header (for framework targets).



In another module

Using `@import MyFramework;` imports the whole module, including Obj-C interfaces to Swift classes (if the aforementioned build setting is enabled).

第43章：文档标记

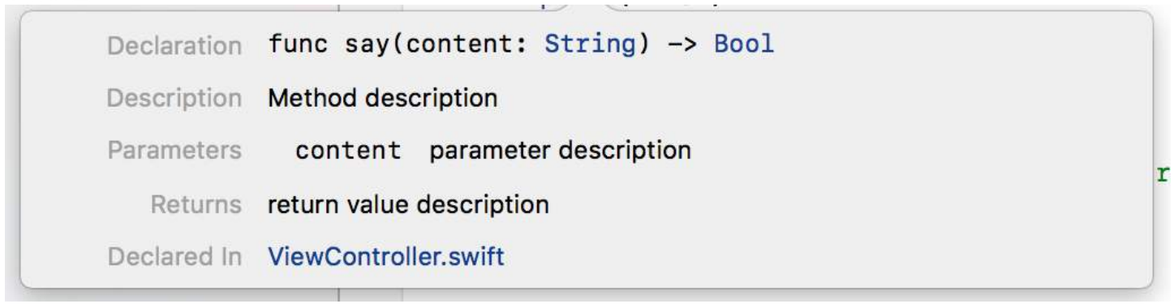
第43.1节：类文档

这是一个基本的类文档示例：

```
/// 类描述
class Student {

    // 成员描述
    var name: String

    /// 方法描述
    ///
    /// - 参数 content: 参数描述
    ///
    /// - 返回值: 返回值描述
    func say(content: String) -> Bool {
        print("\(self.name) 说 \(content)")
        return true
    }
}
```

请注意，使用Xcode 8，您可以生成文档代码片段，方法是 `command`+`option`+`/`.

这将返回：



第43.2节：文档风格

```
/**
将用户添加到被分配任务的人员列表中。

- 参数名称：要添加的名称
                        - 返回值：一个布尔值（true/false），表示用户是否成功添加到人员列表中。
*/
func addMeToList(name: String) -> Bool {

    // 执行某些操作....


    return true
}
```


Chapter 43: Documentation markup

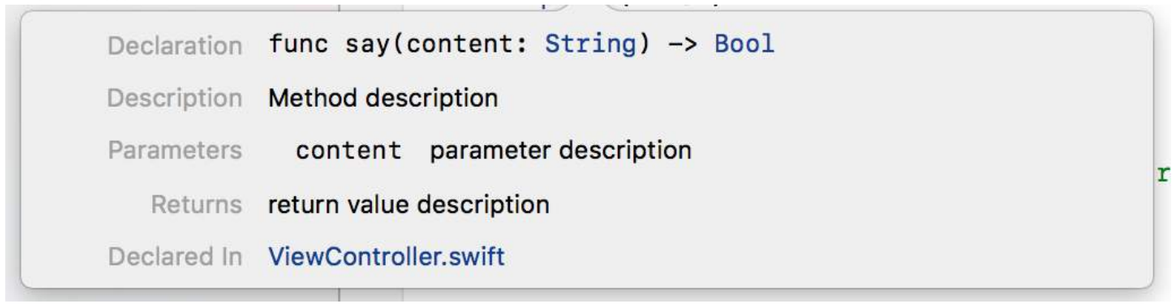
Section 43.1: Class documentation

Here is a basic class documentation example:

```
/// Class description
class Student {

    // Member description
    var name: String

    /// Method description
    ///
    /// - parameter content:   parameter description
    ///
    /// - returns: return value description
    func say(content: String) -> Bool {
        print("\(self.name) say \(content)")
        return true
    }
}
```

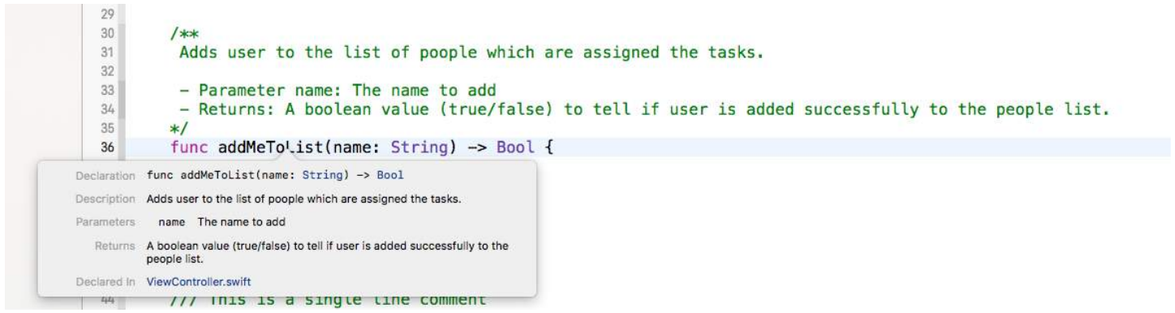
Note that with *Xcode 8*, you can generate the documentation snippet with `command`+`option`+`/`.

This will return:



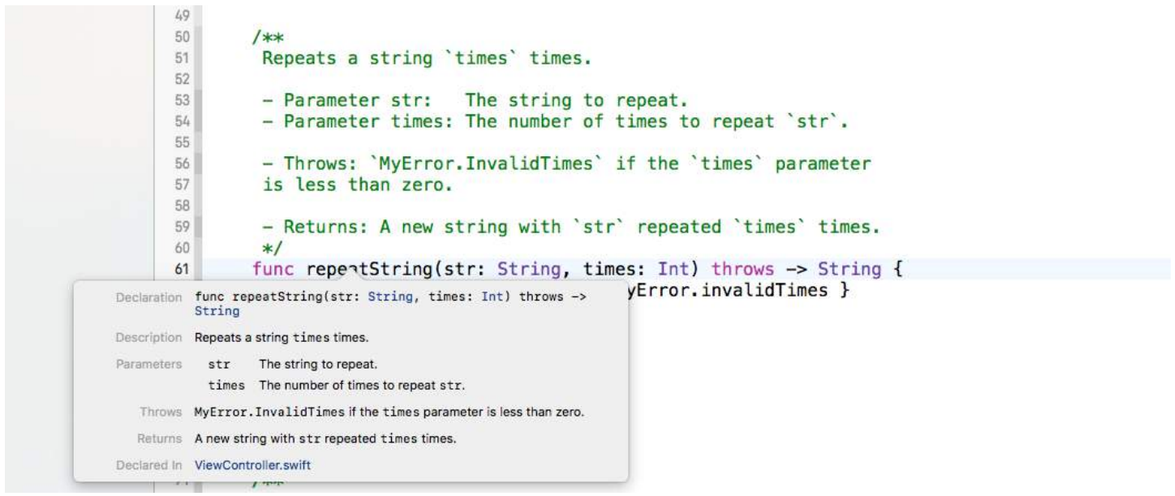
Section 43.2: Documentation styles

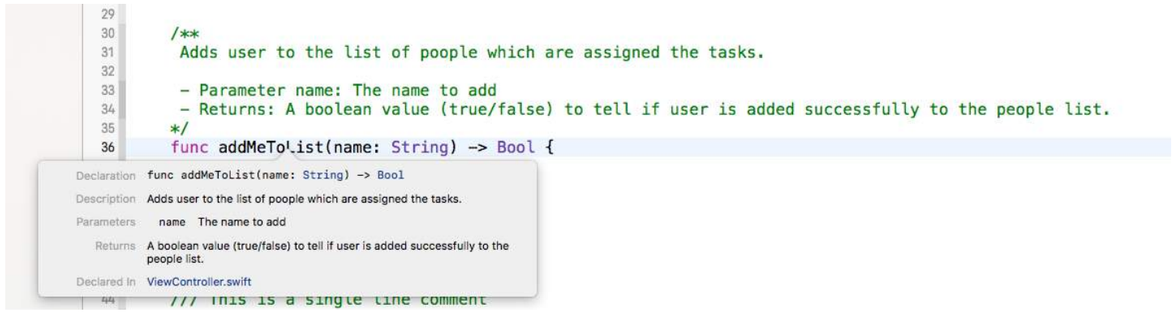
```
/**
 Adds user to the list of poople which are assigned the tasks.

 - Parameter name: The name to add
 - Returns: A boolean value (true/false) to tell if user is added successfully to the people list.
*/
func addMeToList(name: String) -> Bool {

    // Do something....


    return true
}
```




```
/// 这是一个单行注释
func singleLineComment() {

}
```

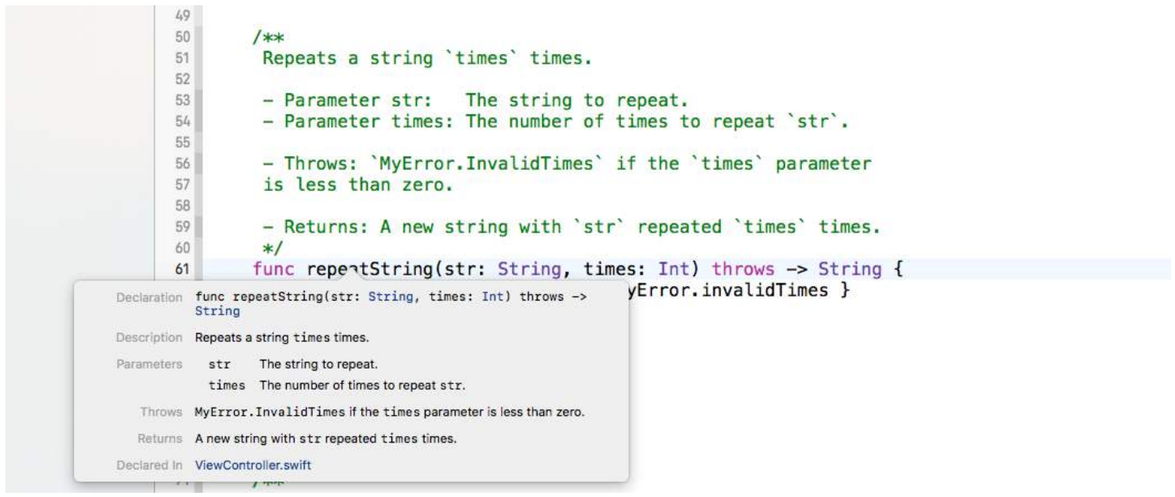

```
/**
重复字符串 `times` 次。

- 参数 str：要重复的字符串。
- 参数 times：重复 `str` 的次数。

- 抛出：如果 `times` 参数小于零，则抛出 `MyError.InvalidTimes`。


- 返回：一个新的字符串，包含 `str` 重复 `times` 次的结果。
*/
func repeatString(str: String, times: Int) throws -> String {
    guard times >= 0 else { throw MyError.invalidTimes }
    return "Hello, world"
}
```



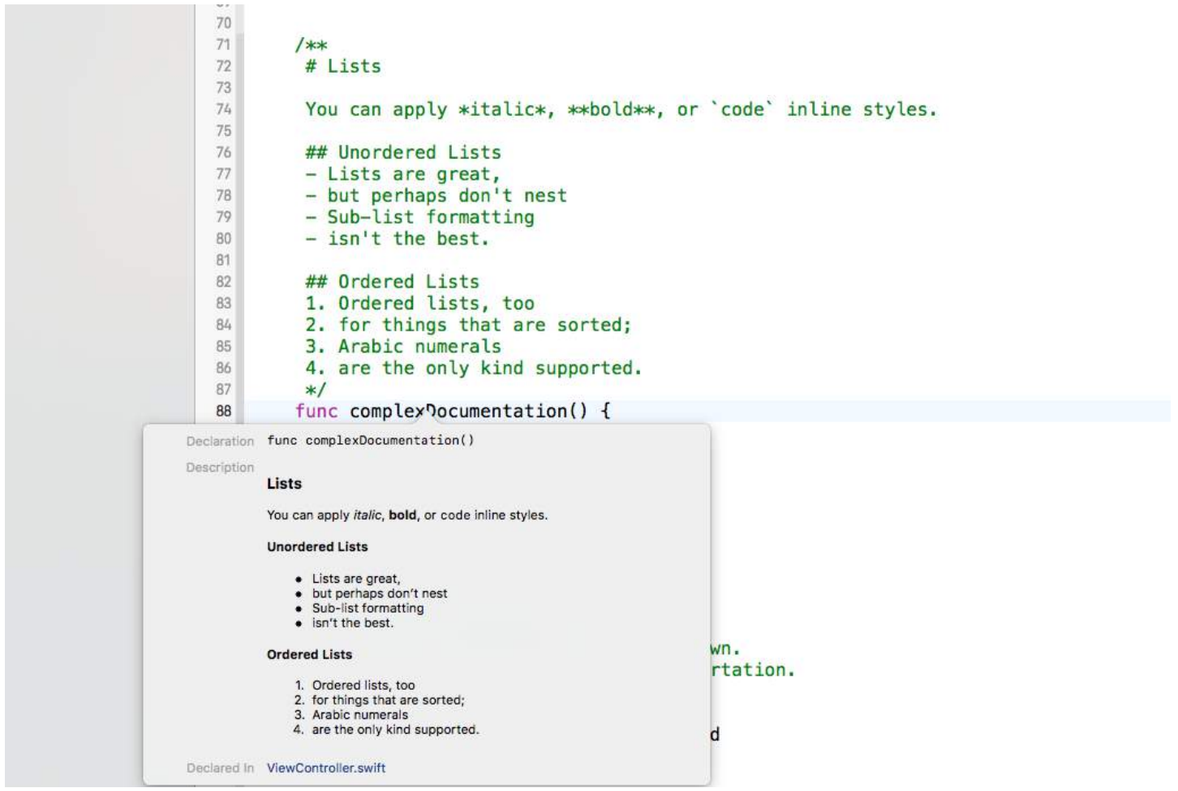



```
/// This is a single line comment
func singleLineComment() {

}
```


```
/**
 Repeats a string `times` times.

 - Parameter str:   The string to repeat.
 - Parameter times: The number of times to repeat `str`.

 - Throws: `MyError.InvalidTimes` if the `times` parameter
 is less than zero.

 - Returns: A new string with `str` repeated `times` times.
*/
func repeatString(str: String, times: Int) throws -> String {
    guard times >= 0 else { throw MyError.invalidTimes }
    return "Hello, world"
}
```



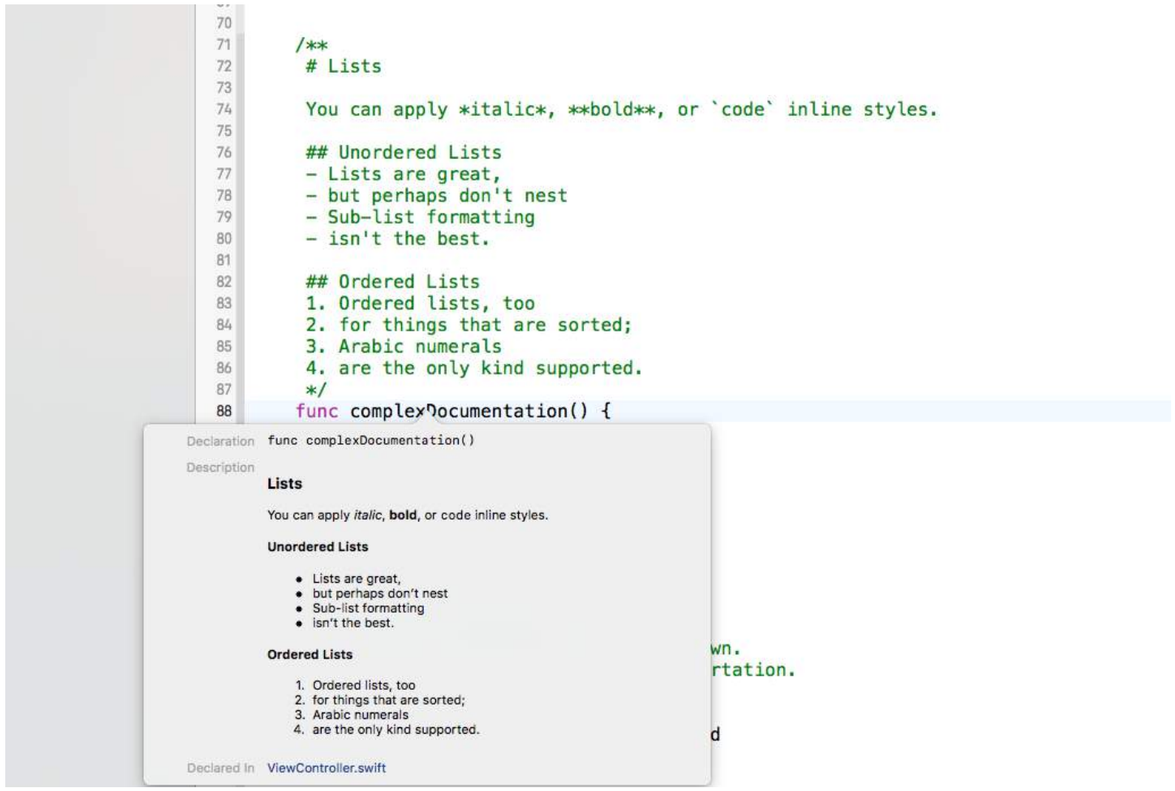
```
/**
# 列表

你可以应用*斜体*、**加粗**或`代码`内联样式。

## 无序列表
 - 列表很棒，
 - 但也许不要嵌套
 - 子列表格式
 - 不是最好的。

## 有序列表
 1. 有序列表也是如此
 2. 用于排序的内容；
 3. 阿拉伯数字
4. 是唯一支持的类型。
*/
func complexDocumentation() {

}
```




```
/**
框架和结构风格。

- 道路：用于街道或小径。
- 旅行：用于长途旅行。
- 巡航：用于城镇周边的休闲出行。
- 混合：用于通用交通。
*/
enum Style {
    案例 公路，旅游，巡航，混合
```



```
/**
 # Lists

 You can apply *italic*, **bold**, or `code` inline styles.

 ## Unordered Lists
 - Lists are great,
 - but perhaps don't nest
 - Sub-list formatting
 - isn't the best.

 ## Ordered Lists
 1. Ordered lists, too
 2. for things that are sorted;
 3. Arabic numerals
 4. are the only kind supported.
*/
func complexDocumentation() {

}
```




```
/**
 Frame and construction style.

 - Road: For streets or trails.
 - Touring: For long journeys.
 - Cruiser: For casual trips around town.
 - Hybrid: For general-purpose transportation.
*/
enum Style {
    case Road, Touring, Cruiser, Hybrid
```

```
}
```

```
    /**
     Frame and construction style.

     - Road: For streets or trails.
     - Touring: For long journeys.
     - Cruiser: For casual trips around town.
     - Hybrid: For general-purpose transportation.
     */
    enum Style {
```

Declaration `enum Style`

Description Frame and construction style.

- Road: For streets or trails.
- Touring: For long journeys.
- Cruiser: For casual trips around town.
- Hybrid: For general-purpose transportation.

Declared In ViewController.swift

}

```
    /**
     Frame and construction style.

     - Road: For streets or trails.
     - Touring: For long journeys.
     - Cruiser: For casual trips around town.
     - Hybrid: For general-purpose transportation.
     */
    enum Style {
```

Declaration `enum Style`

Description Frame and construction style.

- Road: For streets or trails.
- Touring: For long journeys.
- Cruiser: For casual trips around town.
- Hybrid: For general-purpose transportation.

Declared In ViewController.swift

第44章：类型别名

第44.1节：带参数闭包的类型别名

```
类型别名 SuccessHandler = (NSURLSessionDataTask, AnyObject?) -> Void
```

这段代码创建了一个名为SuccessHandler的类型别名，就像var string = ""创建了一个名为string的变量一样。

现在每当你使用SuccessHandler时，例如：

```
func example(_ handler: SuccessHandler) {}
```

你实际上是在写：

```
func example(_ handler: (NSURLSessionDataTask, AnyObject?) -> Void) {}
```

第44.2节：空闭包的类型别名

```
typealias Handler = () -> Void
typealias Handler = () -> ()
```

该代码块创建了一个类型别名，表示一个从Void到Void的函数（不接受参数且无返回值）。

以下是一个使用示例：

```
var func: Handler?

func = {}
```

第44.3节：其他类型的类型别名

```
typealias Number = NSNumber
```

你也可以使用类型别名为某个类型起另一个名字，以便更容易记忆，或使代码更优雅。

元组的类型别名

```
typealias PersonTuple = (name: String, age: Int, address: String)
```

这可以用作：

```
func getPerson(for name: String) -> PersonTuple {
    //从数据库获取，等等
    return ("name", 45, "address")
}
```


Chapter 44: Typealias

Section 44.1: typealias for closures with parameters

```
typealias SuccessHandler = (NSURLSessionDataTask, AnyObject?) -> Void
```

This code block creates a type alias named `SuccessHandler`, just in the same way `var string = ""` creates a variable with the name `string`.

Now whenever you use `SuccessHandler`, for example:

```
func example(_ handler: SuccessHandler) {}
```

You are essentilly writing:

```
func example(_ handler: (NSURLSessionDataTask, AnyObject?) -> Void) {}
```

Section 44.2: typealias for empty closures

```
typealias Handler = () -> Void
typealias Handler = () -> ()
```

This block creates a type alias that works as a Void to Void function (takes in no parameters and returns nothing).

Here is a usage example:

```
var func: Handler?

func = {}
```

Section 44.3: typealias for other types

```
typealias Number = NSNumber
```

You can also use a type alias to give a type another name to make it easier to remember, or make your code more elegant.

typealias for Tuples

```
typealias PersonTuple = (name: String, age: Int, address: String)
```

And this can be used as:

```
func getPerson(for name: String) -> PersonTuple {
    //fetch from db, etc
    return ("name", 45, "address")
}
```

第45章：依赖注入

第45.1节：使用视图控制器的依赖注入

依赖注入简介

一个应用程序由许多相互协作的对象组成。对象通常依赖其他对象来执行某些任务。当一个对象负责引用它自己的依赖时，会导致代码高度耦合、难以测试和难以修改。

依赖注入是一种实现控制反转以解决依赖关系的软件设计模式。
注入是将依赖传递给将使用它的依赖对象。这允许将客户端的依赖与客户端的行为分离，从而使应用程序松散耦合。

不要与上述定义混淆——依赖注入仅仅意味着给对象它的实例变量。

就是这么简单，但它带来了很多好处：

- 更容易测试你的代码（使用自动化测试，如单元测试和UI测试）当与面
- 向协议编程结合使用时，可以轻松更改某个类的实现——更容易重构

- 它使代码更加模块化和可重用

依赖注入（DI）在应用程序中最常用的三种实现方式是：

1. 初始化器注入
2. 属性注入
3. 使用第三方依赖注入框架（如 Swinject、Cleanse、Dip 或 Typhoon）

这里有一篇有趣的文章，里面包含了更多关于依赖注入的相关文章链接，如果你想深入了解依赖注入和控制反转原则，值得一看。

让我们展示如何在视图控制器中使用依赖注入——这是普通iOS开发者的日常任务。

无依赖注入示例

我们将有两个视图控制器：LoginViewController 和 TimelineViewController。LoginViewController 用于登录，登录成功后，它将切换到 TimelineViewController。两个视图控制器都依赖于 FirebaseNetworkService。

登录视图控制器

```
类 登录视图控制器: UIViewController {

    变量 networkService = FirebaseNetworkService()

    重写函数 viewDidLoad() {
        super.viewDidLoad()
    }
}
```

时间线视图控制器

```
类 时间线视图控制器: UIViewController {
```


Chapter 45: Dependency Injection

Section 45.1: Dependency Injection with View Controllers

Dependenct Injection Intro

An application is composed of many objects that collaborate with each other. Objects usually depend on other objects to perform some task. When an object is responsible for referencing its own dependencies it leads to a highly coupled, hard-to-test and hard-to-change code.

Dependency injection is a software design pattern that implements inversion of control for resolving dependencies. An injection is passing of dependency to a dependent object that would use it. This allows a separation of client's dependencies from the client's behaviour, which allows the application to be loosely coupled.

Not to be confused with the above definition - a dependency injection simply means giving an object its instance variables.

It's that simple, but it provides a lot of benefits:

- easier to test your code (using automated tests like unit and UI tests)
- when used in tandem with protocol-oriented programming it makes it easy to change the implementation of a certain class - easier to refactor
- it makes the code more modular and reusable

There are three most commonly used ways Dependency Injection (DI) can be implemented in an application:

1. Initializer injection
2. Property injection
3. Using third party DI frameworks (like Swinject, Cleanse, Dip or Typhoon)

There is an interesting article with links to more articles about Dependency Injection so check it out if you want to dig deeper into DI and Inversion of Control principle.

Let's show how to use DI with View Controllers - an every day task for an average iOS developer.

Example Without DI

We'll have two View Controllers: **LoginViewController** and **TimelineViewController**. LoginViewController is used to login and upon successful loign, it will switch to the TimelineViewController. Both view controllers are dependent on the **FirebaseNetworkService**.

LoginViewController

```
class LoginViewController: UIViewController {

    var networkService = FirebaseNetworkService()

    override func viewDidLoad() {
        super.viewDidLoad()
    }
}
```

TimelineViewController

```
class TimelineViewController: UIViewController {
```

```
    变量 networkService = FirebaseNetworkService()

    重写函数 viewDidLoad() {
        super.viewDidLoad()
    }

    @IBAction 函数 logoutButtonPressed(_ sender: UIButton) {
        networkService.logutCurrentUser()
    }
}
```

Firebase网络服务

```
class FirebaseNetworkService {

    func loginUser(username: String, passwordHash: String) {
        // 实现细节对本例不重要
    }

    func logutCurrentUser() {
        // 实现细节对本例不重要
    }
}
```

这个例子非常简单，但假设你有10或15个不同的视图控制器，其中一些也依赖于FirebaseNetworkService。在某个时刻，你想将Firebase作为后端服务替换为你公司的内部后端服务。为此，你必须逐个查看每个视图控制器，并将FirebaseNetworkService替换为CompanyNetworkService。如果CompanyNetworkService中的某些方法发生了变化，你将有大量工作要做。
CompanyNetworkService中的方法发生了变化，你将有大量工作要做。

单元测试和UI测试不在本例范围内，但如果你想对紧密耦合依赖的视图控制器进行单元测试，将会非常困难。

让我们重写这个例子，将网络服务注入到我们的视图控制器中。

依赖注入示例

为了充分利用依赖注入，让我们在协议中定义网络服务的功能。
这样，依赖网络服务的视图控制器甚至不需要了解其真实的实现。

```
协议 NetworkService {
    函数 loginUser(username: String, passwordHash: String)
    函数 logutCurrentUser()
}
```

添加 NetworkService 协议的实现：

```
类 FirebaseNetworkServiceImpl: NetworkService {
    函数 loginUser(username: String, passwordHash: String) {
        // Firebase 实现
    }

    函数 logutCurrentUser() {
        // Firebase 实现
    }
}
```



```
    var networkService = FirebaseNetworkService()

    override func viewDidLoad() {
        super.viewDidLoad()
    }

    @IBAction func logoutButtonPressed(_ sender: UIButton) {
        networkService.logutCurrentUser()
    }
}
```

FirebaseNetworkService

```
class FirebaseNetworkService {

    func loginUser(username: String, passwordHash: String) {
        // Implementation not important for this example
    }

    func logutCurrentUser() {
        // Implementation not important for this example
    }
}
```

This example is very simple, but let's assume you have 10 or 15 different view controller and some of them are also dependent on the FirebaseNetworkService. At some moment you want to change Firebase as your backend service with your company's in-house backend service. To do that you'll have to go through every view controller and change FirebaseNetworkService with CompanyNetworkService. And if some of the methods in the CompanyNetworkService have changed, you'll have a lot of work to do.

Unit and UI testing is not the scope of this example, but if you wanted to unit test view controllers with tightly coupled dependencies, you would have a really hard time doing so.

Let's rewrite this example and inject Network Service to our view controllers.

Example with Dependancy Injection

To make the best out of the Dependency Injection, let's define the functionality of the Network Service in a protocol. This way, view controllers dependent on a network service won't even have to know about the real implementation of it.

```
protocol NetworkService {
    func loginUser(username: String, passwordHash: String)
    func logutCurrentUser()
}
```

Add an implementation of the NetworkService protocol:

```
class FirebaseNetworkServiceImpl: NetworkService {
    func loginUser(username: String, passwordHash: String) {
        // Firebase implementation
    }

    func logutCurrentUser() {
        // Firebase implementation
    }
}
```

让我们修改 LoginViewController 和 TimelineViewController，使用新的 NetworkService 协议替代 FirebaseNetworkService。

登录视图控制器

```
类 登录视图控制器: UIViewController {

    // 这里不需要初始化，因为 NetworkService 协议的实现
    // 会被注入
    变量 networkService: NetworkService?

    override func viewDidLoad() {
        super.viewDidLoad()
    }
}
```

时间线视图控制器

```
类 时间线视图控制器: UIViewController {

    变量 networkService: NetworkService?

    override func viewDidLoad() {
        super.viewDidLoad()
    }

    @IBAction 函数 logoutButtonPressed(_ sender: UIButton) {
       networkService?.logutCurrentUser()
    }
}
```

现在，问题是：我们如何在登录视图控制器（LoginViewController）和时间线视图控制器（TimelineViewController）中注入正确的NetworkService实现？

由于登录视图控制器是起始视图控制器，并且每次应用启动时都会显示，我们可以在AppDelegate中注入所有依赖。

```
func application(_ application: UIApplication, didFinishLaunchingWithOptions launchOptions:
[UIApplicationLaunchOptionsKey: Any]?) -> Bool {
    // 这段逻辑会根据你的项目结构不同，或者起始视图控制器是否是导航控制器或标签栏控制器而有所不
    同
    if let loginVC = window?.rootViewController as? LoginViewController {
loginVC.networkService = FirebaseNetworkServiceImpl()
    }
    return true
}
```

在AppDelegate中，我们只是获取第一个视图控制器（LoginViewController）的引用，并使用属性注入方法注入NetworkService的实现。

接下来要做的任务是在时间线视图控制器（TimelineViewController）中注入NetworkService的实现。最简单的方法是在LoginViewController切换到TimelineViewController时进行注入。

我们将在LoginViewController的prepareForSegue方法中添加注入代码（如果你使用不同的方式在视图控制器之间导航，请将注入代码放在相应位置）。

我们的LoginViewController类现在看起来是这样的：

Let's change LoginViewController and TimelineViewController to use new NetworkService protocol instead of FirebaseNetworkService.

LoginViewController

```
class LoginViewController: UIViewController {

    // No need to initialize it here since an implementation
    // of the NetworkService protocol will be injected
    var networkService: NetworkService?

    override func viewDidLoad() {
        super.viewDidLoad()
    }
}
```

TimelineViewController

```
class TimelineViewController: UIViewController {

    var networkService: NetworkService?

    override func viewDidLoad() {
        super.viewDidLoad()
    }

    @IBAction func logoutButtonPressed(_ sender: UIButton) {
        networkService?.logutCurrentUser()
    }
}
```

Now, the question is: How do we inject the correct NetworkService implementation in the LoginViewController and TimelineViewController?

Since LoginViewController is the starting view controller and will show every time the application starts, we can inject all dependencies in the **AppDelegate**.

```
func application(_ application: UIApplication, didFinishLaunchingWithOptions launchOptions:
[UIApplicationLaunchOptionsKey: Any]?) -> Bool {
    // This logic will be different based on your project's structure or whether
    // you have a navigation controller or tab bar controller for your starting view controller
    if let loginVC = window?.rootViewController as? LoginViewController {
        loginVC.networkService = FirebaseNetworkServiceImpl()
    }
    return true
}
```

In the AppDelegate we are simply taking the reference to the first view controller (LoginViewController) and injecting the NetworkService implementation using the property injection method.

Now, the next task is to inject the NetworkService implementation in the TimelineViewController. The easiest way is to do that when LoginViewController is transitioning to the TimlineViewController.

We'll add the injection code in the prepareForSegue method in the LoginViewController (if you are using a different approach to navigate through view controllers, place the injection code there).

Our LoginViewController class looks like this now:

```
class LoginViewController: UIViewController {
    // 这里不需要初始化，因为NetworkService协议的实现会被注入

    变量 networkService: NetworkService?

    override func viewDidLoad() {
        super.viewDidLoad()
    }

    override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
        if segue.identifier == "TimelineViewController" {
            if let timelineVC = segue.destination as? TimelineViewController {
                // 注入 NetworkService 实现
timelineVC.networkService = networkService
            }
        }
    }
}
```

我们完成了，就这么简单。

现在想象一下，我们想将 NetworkService 的实现从 Firebase 切换到我们公司自定义的后端实现。我们只需要做的是：

添加新的 NetworkService 实现类：

```
class CompanyNetworkServiceImpl: NetworkService {
    func loginUser(username: String, passwordHash: String) {
        // 公司 API 实现
    }

    func logutCurrentUser() {
        // 公司 API 实现
    }
}
```

在 AppDelegate 中将 FirebaseNetworkServiceImpl 替换为新的实现：

```
func application(_ application: UIApplication, didFinishLaunchingWithOptions launchOptions:
[UIApplicationLaunchOptionsKey: Any]?) -> Bool {
        // 这段逻辑会根据你的项目结构不同，或者你的起始视图控制器是导航控制器还是标签栏控制器而不同

        if let loginVC = window?.rootViewController as? LoginViewController {
loginVC.networkService = CompanyNetworkServiceImpl()
        }
        return true
    }
```

就是这样，我们已经切换了 NetworkService 协议的整个底层实现，甚至没有触碰到 LoginViewController 或 TimelineViewController。

由于这是一个简单的示例，你现在可能看不到所有的好处，但如果你尝试在项目中使用依赖注入（DI），你会看到它的好处，并且会一直使用依赖注入。

第45.2节：依赖注入类型

本示例将演示如何在 Swift 中使用依赖注入（DI）设计模式，使用以下方法：


```
class LoginViewController: UIViewController {
    // No need to initialize it here since an implementation
    // of the NetworkService protocol will be injected
    var networkService: NetworkService?

    override func viewDidLoad() {
        super.viewDidLoad()
    }

    override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
        if segue.identifier == "TimelineViewController" {
            if let timelineVC = segue.destination as? TimelineViewController {
                // Injecting the NetworkService implementation
                timelineVC.networkService = networkService
            }
        }
    }
}
```

We are done and it's that easy.

Now imagine we want to switch our NetworkService implementation from Firebase to our custom company's backend implementation. All we would have to do is:

Add new NetworkService implementation class:

```
class CompanyNetworkServiceImpl: NetworkService {
    func loginUser(username: String, passwordHash: String) {
        // Company API implementation
    }

    func logutCurrentUser() {
        // Company API implementation
    }
}
```

Switch the FirebaseNetworkServiceImpl with the new implementation in the AppDelegate:

```
func application(_ application: UIApplication, didFinishLaunchingWithOptions launchOptions:
[UIApplicationLaunchOptionsKey: Any]?) -> Bool {
        // This logic will be different based on your project's structure or whether
        // you have a navigation controller or tab bar controller for your starting view controller
        if let loginVC = window?.rootViewController as? LoginViewController {
            loginVC.networkService = CompanyNetworkServiceImpl()
        }
        return true
    }
```

That's it, we have switched the whole underlaying implementation of the NetworkService protocol without even touching LoginViewController or TimelineViewController.

As this is a simple example, you might not see all the benefits right now, but if you try to use DI in your projects, you'll see the benefits and will always use Dependency Injection.

Section 45.2: Dependency Injection Types

This example will demonstrate how to use Dependency Injection (**DI**) design pattern in Swift using these methods:

1. 初始化器注入（正确的术语是构造函数注入，但由于 Swift 有初始化器，所以称为初始化器注入）
2. **属性注入**
3. **方法注入**

无依赖注入的示例设置

```
    protocol Engine {
        func startEngine()
        func stopEngine()
    }

    class TrainEngine: Engine {
        func startEngine() {
            print("Engine started")
        }

        func stopEngine() {
            print("引擎已停止")
        }
    }

    protocol TrainCar {
    var numberOfSeats: Int { get }
    func attachCar(attach: Bool)
}

class RestaurantCar: TrainCar {
    var numberOfSeats: Int {
        get {
            return 30
        }
    }
    func attachCar(attach: Bool) {
        print("连接车厢")
    }
}

class PassengerCar: TrainCar {
    var numberOfSeats: Int {
        get {
            return 50
        }
    }
    func attachCar(attach: Bool) {
        print("连接车厢")
    }
}

class Train {
    let engine: Engine?
    var mainCar: TrainCar?
}
```

初始化器依赖注入

顾名思义，所有依赖项都是通过类的初始化器注入的。为了通过初始化器注入依赖项，我们将在Train类中添加初始化器。

Train类现在看起来是这样的：

```
class Train {
```


1. **Initializer Injection** (the proper term is Constructor Injection, but since Swift has initializers it's called initializer injection)
2. **Property Injection**
3. **Method Injection**

Example Setup without DI

```
    protocol Engine {
        func startEngine()
        func stopEngine()
    }

    class TrainEngine: Engine {
        func startEngine() {
            print("Engine started")
        }

        func stopEngine() {
            print("Engine stopped")
        }
    }

    protocol TrainCar {
    var numberOfSeats: Int { get }
    func attachCar(attach: Bool)
}

class RestaurantCar: TrainCar {
    var numberOfSeats: Int {
        get {
            return 30
        }
    }
    func attachCar(attach: Bool) {
        print("Attach car")
    }
}

class PassengerCar: TrainCar {
    var numberOfSeats: Int {
        get {
            return 50
        }
    }
    func attachCar(attach: Bool) {
        print("Attach car")
    }
}

class Train {
    let engine: Engine?
    var mainCar: TrainCar?
}
```

Initializer Dependency Injection

As the name says, all dependencies are injected through the class initializer. To inject dependencies through the initializer, we'll add the initializer to the `Train` class.

Train class now looks like this:

```
class Train {
```

```
    let engine: Engine?
    var mainCar: TrainCar?

    init(engine: Engine) {
        self.engine = engine
    }
}
```

当我们想要创建Train类的实例时，将使用初始化器注入特定的Engine实现：

```
let train = Train(engine: TrainEngine())
```

注意： 与属性注入相比，初始化器注入的主要优点是我们可以将变量设置为私有变量，甚至使用let关键字将其设为常量（正如我们在示例中所做的）。这样我们可以确保没有人能够访问或更改它。

属性依赖注入

使用属性进行依赖注入比使用初始化器更简单。让我们向已经创建的火车对象注入一个乘用车（PassengerCar）依赖，使用属性依赖注入：

```
train.mainCar = PassengerCar()
```

我们的火车的mainCar现在是一个PassengerCar实例。

方法依赖注入

这种类型的依赖注入与前两种稍有不同，因为它不会影响整个对象，而只会注入一个依赖，用于一个特定方法的作用域内。当依赖仅在单个方法中使用时，通常不适合让整个对象依赖它。让我们给Train类添加一个新方法：

```
func reparkCar(trainCar: TrainCar) {
    trainCar.attachCar(attach: true)
    engine?.startEngine()
engine?.stopEngine()
    trainCar.attachCar(attach: false)
}
```

现在，如果我们调用新的Train类方法，就会使用方法依赖注入注入TrainCar。

```
train.reparkCar(trainCar: RestaurantCar())
```



```
    let engine: Engine?
    var mainCar: TrainCar?

    init(engine: Engine) {
        self.engine = engine
    }
}
```

When we want to create an instance of the Train class we'll use initializer to inject a specific Engine implementation:

```
let train = Train(engine: TrainEngine())
```

NOTE: The main advantage of the initializer injection versus the property injection is that we can set the variable as private variable or even make it a constant with the `let` keyword (as we did in our example). This way we can make sure that no one can access it or change it.

Properties Dependency Injection

DI using properties is even simpler that using an initializer. Let's inject a PassengerCar dependency to the train object we already created using the properties DI:

```
train.mainCar = PassengerCar()
```

That's it. Our train's `mainCar` is now a `PassengerCar` instance.

Method Dependency Injection

This type of dependency injection is a little different that the previous two because it won't affect the whole object, but it will only inject a dependency to be used in the scope of one specific method. When a dependency is only used in a single method, it's usually not good to make the whole object dependent on it. Let's add a new method to the Train class:

```
func reparkCar(trainCar: TrainCar) {
    trainCar.attachCar(attach: true)
    engine?.startEngine()
    engine?.stopEngine()
    trainCar.attachCar(attach: false)
}
```

Now, if we call the new Train's class method, we'll inject the `TrainCar` using the method dependency injection.

```
train.reparkCar(trainCar: RestaurantCar())
```

第46章：磁盘空间缓存

使用URLSession和FileManager缓存视频、图片和音频

第46.1节：读取

```
let url = "https://path-to-media"
guard let documentsUrl = FileManager.default.urls(for: .documentDirectory, in:
.userDomainMask).first,
      let searchQuery = url.absoluteString.components(separatedBy: "/").last else {
  return nil
}

do {
  let directoryContents = try FileManager.default.contentsOfDirectory(at: documentsUrl,
includingPropertiesForKeys: nil, options: [])
  let cachedFiles = directoryContents.filter { $0.absoluteString.contains(searchQuery) }

  // 对通过url找到的文件进行操作
} catch {
  // 未能找到任何文件
}
```

第46.2节：保存

```
let url = "https://path-to-media"
let request = URLRequest(url: url)
let downloadTask = URLSession.shared.downloadTask(with: request) { (location, response, error) in
  guard let location = location,
        let response = response,
        let documentsPath = NSSearchPathForDirectoriesInDomains(.documentDirectory,
.userDomainMask, true).first else {
    return
  }
  let documentsDirectoryUrl = URL(fileURLWithPath: documentsPath)
  let documentUrl = documentsDirectoryUrl.appendingPathComponent(response.suggestedFilename)
  let _ = try? FileManager.default.moveItem(at: location, to: documentUrl)

  // documentUrl 是我们刚刚下载并保存到 FileManager 的本地 URL
}.resume()
```


Chapter 46: Caching on disk space

Caching videos, images and audios using `URLSession` and `FileManager`

Section 46.1: Reading

```
let url = "https://path-to-media"
guard let documentsUrl = FileManager.default.urls(for: .documentDirectory, in:
.userDomainMask).first,
      let searchQuery = url.absoluteString.components(separatedBy: "/").last else {
  return nil
}

do {
  let directoryContents = try FileManager.default.contentsOfDirectory(at: documentsUrl,
includingPropertiesForKeys: nil, options: [])
  let cachedFiles = directoryContents.filter { $0.absoluteString.contains(searchQuery) }

  // do something with the files found by the url
} catch {
  // Could not find any files
}
```

Section 46.2: Saving

```
let url = "https://path-to-media"
let request = URLRequest(url: url)
let downloadTask = URLSession.shared.downloadTask(with: request) { (location, response, error) in
  guard let location = location,
        let response = response,
        let documentsPath = NSSearchPathForDirectoriesInDomains(.documentDirectory,
.userDomainMask, true).first else {
    return
  }
  let documentsDirectoryUrl = URL(fileURLWithPath: documentsPath)
  let documentUrl = documentsDirectoryUrl.appendingPathComponent(response.suggestedFilename)
  let _ = try? FileManager.default.moveItem(at: location, to: documentUrl)

  // documentUrl is the local URL which we just downloaded and saved to the FileManager
}.resume()
```

第47章：使用 Swift 的算法

算法是计算的基础。在不同情况下选择使用哪种算法，区分了普通程序员和优秀程序员。基于此，以下是一些基本算法的定义和代码示例。

第47.1节：排序

冒泡排序

这是一种简单的排序算法，它反复遍历待排序的列表，比较每对相邻的元素，如果顺序错误则交换它们。遍历列表的过程会重复进行，直到不再需要交换。虽然算法简单，但对于大多数问题来说太慢且不实用。它的时间复杂度是 $O(n^2)$，但被认为比插入排序更慢。

```
extension Array where Element: Comparable {

func bubbleSort() -> Array<Element> {

    //检查简单情况
    guard self.count > 1 else {
        return self
    }

    //变异副本
    var output: Array<Element> = self

    for primaryIndex in 0..<self.count {
        let passes = (output.count - 1) - primaryIndex

        //"半开区间"操作符
        for secondaryIndex in 0..<passes {
            let key = output[secondaryIndex]

            //比较 / 交换位置
            if (key > output[secondaryIndex + 1]) {
                swap(&output[secondaryIndex], &output[secondaryIndex + 1])
            }
        }
    }

    return output
}

}
```

插入排序

插入排序是计算机科学中较为基础的算法之一。插入排序通过遍历集合并根据元素的值对其进行定位来对元素进行排序。集合被分为已排序和未排序两部分，重复此过程直到所有元素排序完成。插入排序的时间复杂度为O(n2)。你可以将其放入扩展中，如下面的示例，或者为其创建一个方法。

```
extension Array where Element: Comparable {

func insertionSort() -> Array<Element> {
```


Chapter 47: Algorithms with Swift

Algorithms are a backbone to computing. Making a choice of which algorithm to use in which situation distinguishes an average from good programmer. With that in mind, here are definitions and code examples of some of the basic algorithms out there.

Section 47.1: Sorting

Bubble Sort

This is a simple sorting algorithm that repeatedly steps through the list to be sorted, compares each pair of adjacent items and swaps them if they are in the wrong order. The pass through the list is repeated until no swaps are needed. Although the algorithm is simple, it is too slow and impractical for most problems. It has complexity of O(n2) but it is considered slower than insertion sort.

```
extension Array where Element: Comparable {

func bubbleSort() -> Array<Element> {

    //check for trivial case
    guard self.count > 1 else {
        return self
    }

    //mutated copy
    var output: Array<Element> = self

    for primaryIndex in 0..<self.count {
        let passes = (output.count - 1) - primaryIndex

        //"half-open" range operator
        for secondaryIndex in 0..<passes {
            let key = output[secondaryIndex]

            //compare / swap positions
            if (key > output[secondaryIndex + 1]) {
                swap(&output[secondaryIndex], &output[secondaryIndex + 1])
            }
        }
    }

    return output
}

}
```

Insertion sort

Insertion sort is one of the more basic algorithms in computer science. The insertion sort ranks elements by iterating through a collection and positions elements based on their value. The set is divided into sorted and unsorted halves and repeats until all elements are sorted. Insertion sort has complexity of O(n2). You can put it in an extension, like in an example below, or you can create a method for it.

```
extension Array where Element: Comparable {

func insertionSort() -> Array<Element> {
```

```
    //检查简单情况
    guard self.count > 1 else {
        return self
    }

    //变异副本
    var output: Array<Element> = self

    for primaryindex in 0..<output.count {

        let key = output[primaryindex]
        var secondaryindex = primaryindex

        while secondaryindex > -1 {
            if key < output[secondaryindex] {

                //移动到正确的位置
output.remove(at: secondaryindex + 1)
                output.insert(key, at: secondaryindex)
            }
secondaryindex -= 1
        }
    }

    return output
}
}
```

选择排序

选择排序以其简单性著称。它从数组的第一个元素开始，将其值保存为最小值（或最大值，取决于排序顺序）。然后遍历数组，遇到比当前最小值更小的值时就替换该最小值。该最小值随后被放置在数组的最左侧，接着从下一个索引开始重复此过程，直到数组末尾。选择排序的时间复杂度为O(n²)，但它被认为比其对应的排序算法更慢——选择排序。

func selectionSort() -> Array { //检查简单情况 guard self.count > 1 else { return self }

```
//变异副本
var output: Array<Element> = self

for primaryindex in 0..<output.count {
    var minimum = primaryindex
    var secondaryindex = primaryindex + 1

    while secondaryindex < output.count {
        //将最低值存为minimum
        if output[minimum] > output[secondaryindex] {
            minimum = secondaryindex
        }
secondaryindex += 1
    }

    //将minimum值与数组迭代位置交换
    if primaryindex != minimum {
        swap(&output[primaryindex], &output[minimum])
    }
}

return output
```



```
    //check for trivial case
    guard self.count > 1 else {
        return self
    }

    //mutated copy
    var output: Array<Element> = self

    for primaryindex in 0..<output.count {

        let key = output[primaryindex]
        var secondaryindex = primaryindex

        while secondaryindex > -1 {
            if key < output[secondaryindex] {

                //move to correct position
                output.remove(at: secondaryindex + 1)
                output.insert(key, at: secondaryindex)
            }
            secondaryindex -= 1
        }
    }

    return output
}
}
```

Selection sort

Selection sort is noted for its simplicity. It starts with the first element in the array, saving it's value as a minimum value (or maximum, depending on sorting order). It then itterates through the array, and replaces the min value with any other value lesser then min it finds on the way. That min value is then placed at the leftmost part of the array and the process is repeated, from the next index, until the end of the array. Selection sort has complexity of O(n2) but it is considered slower than it's counterpart - Selection sort.

func selectionSort() -> Array { //check for trivial case guard self.count > 1 else { return self }

```
//mutated copy
var output: Array<Element> = self

for primaryindex in 0..<output.count {
    var minimum = primaryindex
    var secondaryindex = primaryindex + 1

    while secondaryindex < output.count {
        //store lowest value as minimum
        if output[minimum] > output[secondaryindex] {
            minimum = secondaryindex
        }
        secondaryindex += 1
    }

    //swap minimum value with array iteration
    if primaryindex != minimum {
        swap(&output[primaryindex], &output[minimum])
    }
}

return output
```

```
}
```

快速排序 - O(n log n) 时间复杂度

快速排序是高级算法之一。它具有O(n log n)的时间复杂度，并采用分治策略。这种组合带来了高级的算法性能。
快速排序首先将一个大数组分成两个较小的子数组：低元素和高元素。然后快速排序可以递归地对这些子数组进行排序。

步骤如下：

从数组中选择一个元素，称为枢轴（pivot）。

重新排列数组，使所有小于基准值的元素位于基准值之前，而所有大于基准值的元素位于其之后（等于基准值的元素可以放在任一侧）。完成此划分后，基准值处于其最终位置。这称为划分操作。

递归地将上述步骤应用于较小值元素的子数组和较大值元素的子数组。

变异函数 quickSort() -> Array {

```
函数 qSort(start startIndex: Int, _ pivot: Int) {

    如果 (startIndex < pivot) {
        令 iPivot = qPartition(start: startIndex, pivot)
        qSort(start: startIndex, iPivot - 1)
        qSort(start: iPivot + 1, pivot)
    }
}
qSort(start: 0, self.endIndex - 1)
返回 self
}

变异函数 qPartition(start startIndex: Int, _ pivot: Int) -> Int {

变量 wallIndex: Int = startIndex

//比较范围与枢轴
对于 currentIndex in wallIndex..<pivot {

    如果 self[currentIndex] <= self[pivot] {
        如果 wallIndex != currentIndex {
            交换(&self[currentIndex], &self[wallIndex])
        }

        //前进墙索引
        wallIndex += 1
    }
}
    //将枢轴移动到最终位置
    if wallIndex != pivot {
        swap(&self[wallIndex], &self[pivot])
    }
    return wallIndex
}
```



```
}
```

Quick Sort - O(n log n) complexity time

Quicksort is one of the advanced algorithms. It features a time complexity of O(n log n) and applies a divide & conquer strategy. This combination results in advanced algorithmic performance. Quicksort first divides a large array into two smaller sub-arrays: the low elements and the high elements. Quicksort can then recursively sort the sub-arrays.

The steps are:

Pick an element, called a pivot, from the array.

Reorder the array so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the partition operation.

Recursively apply the above steps to the sub-array of elements with smaller values and separately to the sub-array of elements with greater values.

mutating func quickSort() -> Array {

```
func qSort(start startIndex: Int, _ pivot: Int) {

    if (startIndex < pivot) {
        let iPivot = qPartition(start: startIndex, pivot)
        qSort(start: startIndex, iPivot - 1)
        qSort(start: iPivot + 1, pivot)
    }
}
qSort(start: 0, self.endIndex - 1)
return self
}

mutating func qPartition(start startIndex: Int, _ pivot: Int) -> Int {

var wallIndex: Int = startIndex

//compare range with pivot
for currentIndex in wallIndex..<pivot {

    if self[currentIndex] <= self[pivot] {
        if wallIndex != currentIndex {
            swap(&self[currentIndex], &self[wallIndex])
        }

        //advance wall
        wallIndex += 1
    }
}
    //move pivot to final position
    if wallIndex != pivot {
        swap(&self[wallIndex], &self[pivot])
    }
    return wallIndex
}
```

第47.2节：插入排序

插入排序是计算机科学中较为基础的算法之一。插入排序通过遍历集合并根据元素的值对其进行定位来对元素进行排序。集合被分为已排序和未排序两部分，重复此过程直到所有元素排序完成。插入排序的时间复杂度为O(n2)。你可以将其放入扩展中，如下面的示例，或者为其创建一个方法。

```
extension Array where Element: Comparable {

func insertionSort() -> Array<Element> {

    //检查简单情况
    guard self.count > 1 else {
        return self
    }

    //变异副本
    var output: Array<Element> = self

    for primaryindex in 0..<output.count {

        let key = output[primaryindex]
        var secondaryindex = primaryindex

        while secondaryindex > -1 {
            if key < output[secondaryindex] {

                //移动到正确的位置
output.remove(at: secondaryindex + 1)
                output.insert(key, at: secondaryindex)
            }
secondaryindex -= 1
        }
    }

    return output
}
}
```

第47.3节：选择排序

选择排序以其简单性著称。它从数组的第一个元素开始，将其值保存为最小值（或最大值，取决于排序顺序）。然后遍历数组，遇到比当前最小值更小的值时就替换该最小值。该最小值随后被放置在数组的最左侧，接着从下一个索引开始重复此过程，直到数组末尾。选择排序的时间复杂度为O(n²)，但它被认为比其对应的排序算法更慢——选择排序。

```
func selectionSort() -> Array<Element> {
    //检查简单情况
guard self.count > 1 else {
        return self
    }

    //变异副本
    var output: Array<Element> = self

    for primaryindex in 0..<output.count {
        var minimum = primaryindex
```


Section 47.2: Insertion Sort

Insertion sort is one of the more basic algorithms in computer science. The insertion sort ranks elements by iterating through a collection and positions elements based on their value. The set is divided into sorted and unsorted halves and repeats until all elements are sorted. Insertion sort has complexity of O(n2). You can put it in an extension, like in an example below, or you can create a method for it.

```
extension Array where Element: Comparable {

func insertionSort() -> Array<Element> {

    //check for trivial case
    guard self.count > 1 else {
        return self
    }

    //mutated copy
    var output: Array<Element> = self

    for primaryindex in 0..<output.count {

        let key = output[primaryindex]
        var secondaryindex = primaryindex

        while secondaryindex > -1 {
            if key < output[secondaryindex] {

                //move to correct position
                output.remove(at: secondaryindex + 1)
                output.insert(key, at: secondaryindex)
            }
            secondaryindex -= 1
        }
    }

    return output
}
}
```

Section 47.3: Selection sort

Selection sort is noted for its simplicity. It starts with the first element in the array, saving it's value as a minimum value (or maximum, depending on sorting order). It then itterates through the array, and replaces the min value with any other value lesser then min it finds on the way. That min value is then placed at the leftmost part of the array and the process is repeated, from the next index, until the end of the array. Selection sort has complexity of O(n2) but it is considered slower than it's counterpart - Selection sort.

```
func selectionSort() -> Array<Element> {
    //check for trivial case
    guard self.count > 1 else {
        return self
    }

    //mutated copy
    var output: Array<Element> = self

    for primaryindex in 0..<output.count {
        var minimum = primaryindex
```

```
        var secondaryindex = primaryindex + 1

        while secondaryindex < output.count {
            //将最低值存为最小值
            if output[minimum] > output[secondaryindex] {
                minimum = secondaryindex
            }
secondaryindex += 1
        }

        //将minimum值与数组迭代位置交换
        if primaryindex != minimum {
            swap(&output[primaryindex], &output[minimum])
        }
    }

    return output
}
```

第47.4节：渐近分析

由于我们有许多不同的算法可供选择，当我们想要对数组进行排序时，我们需要知道哪一个算法能够完成它的任务。因此，我们需要某种方法来衡量算法的速度和可靠性。这就是
渐近分析开始发挥作用。渐近分析是描述算法效率随输入规模（n）增长的过程。在计算机科学中，渐近通常以一种通用格式表示，称为大O符号（Big O Notation）。

- 线性时间 O(n)：当数组中的每个元素都必须被评估以使函数达到其目标时，这意味着随着元素数量的增加，函数的效率会降低。这样的函数被称为线性时间运行，因为其速度依赖于输入规模。

- 多项式时间 $O(n^2)$：如果函数的复杂度呈指数增长（意味着对于n个元素的数组，函数的复杂度是n的平方），则该函数在多项式时间内运行。这通常是带有嵌套循环的函数。两个嵌套循环导致$O(n^2)$复杂度，三个嵌套循环导致$O(n^3)$复杂度，依此类推……

- 对数时间 O(log n)：当输入规模（n）增长时，对数时间函数的复杂度增长最小。这类函数是每个程序员都追求的。

第47.5节：快速排序 - O(n log n)复杂度时间

快速排序是高级算法之一。它具有O(n log n)的时间复杂度，并采用分治策略。这种组合带来了高级的算法性能。
快速排序首先将一个大数组分成两个较小的子数组：低元素和高元素。然后快速排序可以递归地对这些子数组进行排序。

步骤如下：

1. 从数组中选择一个元素，称为枢轴（pivot）。
2. 重新排列数组，使所有值小于枢轴的元素位于枢轴之前，所有值大于枢轴的元素位于枢轴之后（相等的值可以放在任一侧）。完成此划分后，枢轴处于其最终位置。这称为划分操作。
3. 递归地对值较小的子数组和值较大的子数组分别应用上述步骤。

```
mutating func quickSort() -> Array<Element> {
```



```
        var secondaryindex = primaryindex + 1

        while secondaryindex < output.count {
            //store lowest value as minimum
            if output[minimum] > output[secondaryindex] {
                minimum = secondaryindex
            }
            secondaryindex += 1
        }

        //swap minimum value with array iteration
        if primaryindex != minimum {
            swap(&output[primaryindex], &output[minimum])
        }
    }

    return output
}
```

Section 47.4: Asymptotic analysis

Since we have many different algorithms to choose from, when we want to sort an array, we need to know which one will do it's job. So we need some method of measuring algoritm's speed and reliability. That's where Asymptotic analysis kicks in. Asymptotic analysis is the process of describing the efficiency of algorithms as their input size (n) grows. In computer science, asymptotics are usually expressed in a common format known as Big O Notation.

- **Linear time O(n)**: When each item in the array has to be evaluated in order for a function to achieve it's goal, that means that the function becomes less efficent as number of elements is increasing. *A function like this is said to run in linear time because its speed is dependent on its input size.*
- **Polynominal time O(n2)**: If complexity of a function grows exponentialy (meaning that for n elements of an array complexity of a function is n squared) that function operates in polynominal time. These are usually functions with nested loops. Two nested loops result in O(n2) complexity, three nested loops result in O(n3) complexity, and so on...
- **Logarithmic time O(log n):** Logarithmic time functions's complexity is minimized when the size of its inputs (n) grows. These are the type of functions every programmer strives for.

Section 47.5: Quick Sort - O(n log n) complexity time

Quicksort is one of the advanced algorithms. It features a time complexity of O(n log n) and applies a divide & conquer strategy. This combination results in advanced algorithmic performance. Quicksort first divides a large array into two smaller sub-arrays: the low elements and the high elements. Quicksort can then recursively sort the sub-arrays.

The steps are:

1. Pick an element, called a pivot, from the array.
2. Reorder the array so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the partition operation.
3. Recursively apply the above steps to the sub-array of elements with smaller values and separately to the sub-array of elements with greater values.

```
mutating func quickSort() -> Array<Element> {
```

```
函数 qSort(start startIndex: Int, _ pivot: Int) {

    如果 (startIndex < pivot) {
        令 iPivot = qPartition(start: startIndex, pivot)
        qSort(start: startIndex, iPivot - 1)
        qSort(start: iPivot + 1, pivot)
    }
}
qSort(start: 0, self.endIndex - 1)
返回 self

}

mutating func qPartition(start startIndex: Int, _ pivot: Int) -> Int {

var wallIndex: Int = startIndex

//比较范围与枢轴
for currentIndex in wallIndex..<pivot {

    如果 self[currentIndex] <= self[pivot] {
        如果 wallIndex != currentIndex {
            交换(&self[currentIndex], &self[wallIndex])
        }

        //前进墙索引
        wallIndex += 1
    }
}

    //将枢轴移动到最终位置
    if wallIndex != pivot {
        swap(&self[wallIndex], &self[pivot])
    }
    return wallIndex
}
```

第47.6节：图，字典树，栈

图

在计算机科学中，图是一种抽象数据类型，旨在实现数学中的无向图和有向图概念。图数据结构由有限（且可能可变）的一组顶点或节点或点组成，连同一组无序顶点对（用于无向图）或有序顶点对（用于有向图）。这些顶点对称为无向图的边、弧或线，有向图的箭头、有向边、有向弧或有向线。顶点可以是图结构的一部分，也可以是由整数索引或引用表示的外部实体。图数据结构还可以为每条边关联一些边值，例如符号标签或数值属性（成本、容量、长度等）。

(Wikipedia, source)

```
//
//  GraphFactory.swift
//  SwiftStructures
//
//  由Wayne Bishop于2014年6月7日创建。
//  版权所有 (c) 2014 Arbutus Software Inc. 保留所有权利。
//
```



```
func qSort(start startIndex: Int, _ pivot: Int) {

    if (startIndex < pivot) {
        let iPivot = qPartition(start: startIndex, pivot)
        qSort(start: startIndex, iPivot - 1)
        qSort(start: iPivot + 1, pivot)
    }
}
qSort(start: 0, self.endIndex - 1)
return self

}

mutating func qPartition(start startIndex: Int, _ pivot: Int) -> Int {

var wallIndex: Int = startIndex

//compare range with pivot
for currentIndex in wallIndex..<pivot {

    if self[currentIndex] <= self[pivot] {
        if wallIndex != currentIndex {
            swap(&self[currentIndex], &self[wallIndex])
        }

        //advance wall
        wallIndex += 1
    }
}

    //move pivot to final position
    if wallIndex != pivot {
        swap(&self[wallIndex], &self[pivot])
    }
    return wallIndex
}
```

Section 47.6: Graph, Trie, Stack

Graph

In computer science, a graph is an abstract data type that is meant to implement the undirected graph and directed graph concepts from mathematics. A graph data structure consists of a finite (and possibly mutable) set of vertices or nodes or points, together with a set of unordered pairs of these vertices for an undirected graph or a set of ordered pairs for a directed graph. These pairs are known as edges, arcs, or lines for an undirected graph and as arrows, directed edges, directed arcs, or directed lines for a directed graph. The vertices may be part of the graph structure, or may be external entities represented by integer indices or references. A graph data structure may also associate to each edge some edge value, such as a symbolic label or a numeric attribute (cost, capacity, length, etc.).
(Wikipedia, source)

```
//
//  GraphFactory.swift
//  SwiftStructures
//
//  Created by Wayne Bishop on 6/7/14.
//  Copyright (c) 2014 Arbutus Software Inc. All rights reserved.
//
```

```
import Foundation


public class SwiftGraph {


    //声明一个默认的有向图画布
    private var canvas: Array<Vertex>
    public var isDirected: Bool


    init() {
canvas = Array<Vertex>()
        isDirected = true
    }


    //创建一个新的顶点
    func addVertex(key: String) -> Vertex {


        //设置键值
        let childVertex: Vertex = Vertex()
        childVertex.key = key


        //将顶点添加到图画布中
       canvas.append(childVertex)


        return childVertex
    }



//向源顶点添加边
func addEdge(source: Vertex, neighbor: Vertex, weight: Int) {


    //创建一条新边
    let newEdge = Edge()


    //设置默认属性
   newEdge.neighbor = neighbor
   newEdge.weight = weight
   source.neighbors.append(newEdge)


    print("顶点的邻居: \(source.key as String!) 是 \(neighbor.key as String!)..")


    //检查无向图的条件
    if isDirected == false {


        //创建一条新的反向边
        let reverseEdge = Edge()


        //设置反向边的属性
     reverseEdge.neighbor = source
```



```
import Foundation


public class SwiftGraph {


    //declare a default directed graph canvas
    private var canvas: Array<Vertex>
    public var isDirected: Bool


    init() {
        canvas = Array<Vertex>()
        isDirected = true
    }


    //create a new vertex
    func addVertex(key: String) -> Vertex {


        //set the key
        let childVertex: Vertex = Vertex()
        childVertex.key = key


        //add the vertex to the graph canvas
        canvas.append(childVertex)


        return childVertex
    }



//add edge to source vertex
func addEdge(source: Vertex, neighbor: Vertex, weight: Int) {


    //create a new edge
    let newEdge = Edge()


    //establish the default properties
    newEdge.neighbor = neighbor
    newEdge.weight = weight
    source.neighbors.append(newEdge)


    print("The neighbor of vertex: \(source.key as String!) is \(neighbor.key as String!)..")


    //check condition for an undirected graph
    if isDirected == false {


        //create a new reversed edge
        let reverseEdge = Edge()


        //establish the reversed properties
        reverseEdge.neighbor = source
```

```
reverseEdge.weight = weight
            neighbor.neighbors.append(reverseEdge)

            print("顶点的邻居: \(neighbor.key as String!) 是 \(source.key as
String!)..")

        }


    }




    /* 给定最短路径，反转路径序列。
类似于反转链表的过程。 */

    func reversePath(_ head: Path!, source: Vertex) -> Path! {


guard head != nil else {
             return head
        }

        //变异的副本
        var output = head


        var current: Path! = output
        var prev: Path!
        var next: Path!


        while(current != nil) {
            next = current.previous
            current.previous = prev
          prev = current
current = next
        }


        //将源路径附加到序列中
        let sourcePath: Path = Path()

sourcePath.destination = source
        sourcePath.previous = prev
sourcePath.total = nil

       output = sourcePath


        return output

    }




    //处理Dijkstra最短路径算法
    func processDijkstra(_ source: 顶点, destination: 顶点) -> 路径? {
```



```
            reverseEdge.weight = weight
            neighbor.neighbors.append(reverseEdge)

            print("The neighbor of vertex: \(neighbor.key as String!) is \(source.key as
String!)..")

        }


    }




    /* reverse the sequence of paths given the shortest path.
       process analagous to reversing a linked list. */

    func reversePath(_ head: Path!, source: Vertex) -> Path! {


        guard head != nil else {
            return head
        }

        //mutated copy
        var output = head


        var current: Path! = output
        var prev: Path!
        var next: Path!


        while(current != nil) {
            next = current.previous
            current.previous = prev
            prev = current
            current = next
        }


        //append the source path to the sequence
        let sourcePath: Path = Path()

        sourcePath.destination = source
        sourcePath.previous = prev
        sourcePath.total = nil

        output = sourcePath


        return output

    }




    //process Dijkstra's shortest path algorthim
    func processDijkstra(_ source: Vertex, destination: Vertex) -> Path? {
```

```
        var frontier: Array<路径> = Array<路径>()
        var finalPaths: Array<路径> = Array<路径>()


        //使用源点的边来创建前沿
        for e in source.neighbors {

            let newPath: 路径 = 路径()


newPath.destination = e.neighbor
            newPath.previous = nil
newPath.total = e.weight


            //将新路径添加到前沿
           frontier.append(newPath)

        }


        //构建最佳路径
        var bestPath: 路径 = 路径()


        while frontier.count != 0 {

            //使用贪心算法支持路径变化
            bestPath = Path()
            var pathIndex: Int = 0


            for x in 0..<frontier.count {

                let itemPath: Path = frontier[x]

                if  (bestPath.total == nil) || (itemPath.total < bestPath.total) {
                 bestPath = itemPath
                    pathIndex = x
                }

            }



            //enumerate the bestPath edges
            for e in bestPath.destination.neighbors {

                let newPath: 路径 = 路径()

newPath.destination = e.neighbor
                newPath.previous = bestPath
                newPath.total = bestPath.total + e.weight


                //add the new path to the frontier
                frontier.append(newPath)

            }
```



```
        var frontier: Array<Path> = Array<Path>()
        var finalPaths: Array<Path> = Array<Path>()


        //use source edges to create the frontier
        for e in source.neighbors {

            let newPath: Path = Path()


            newPath.destination = e.neighbor
            newPath.previous = nil
            newPath.total = e.weight


            //add the new path to the frontier
            frontier.append(newPath)

        }


        //construct the best path
        var bestPath: Path = Path()


        while frontier.count != 0 {

            //support path changes using the greedy approach
            bestPath = Path()
            var pathIndex: Int = 0


            for x in 0..<frontier.count {

                let itemPath: Path = frontier[x]

                if  (bestPath.total == nil) || (itemPath.total < bestPath.total) {
                    bestPath = itemPath
                    pathIndex = x
                }

            }



            //enumerate the bestPath edges
            for e in bestPath.destination.neighbors {

                let newPath: Path = Path()

                newPath.destination = e.neighbor
                newPath.previous = bestPath
                newPath.total = bestPath.total + e.weight


                //add the new path to the frontier
                frontier.append(newPath)

            }
```

```
        //preserve the bestPath
        finalPaths.append(bestPath)


        //从边界中移除bestPath
        //frontier.removeAtIndex(pathIndex) - Swift2
      frontier.remove(at: pathIndex)


} //结束while循环


//将最短路径设为可选类型
var shortestPath: Path! = Path()


for itemPath in finalPaths {

    if (itemPath.destination.key == destination.key) {

        if  (shortestPath.total == nil) || (itemPath.total < shortestPath.total) {
         shortestPath = itemPath
        }

    }

}


return shortestPath

    }


    ///Dijkstra最短路径算法的优化版本
    func processDijkstraWithHeap(_ source: Vertex, destination: Vertex) -> Path! {


        let frontier: PathHeap = PathHeap()
        let finalPaths: PathHeap = PathHeap()


        //使用源点的边来创建前沿
        for e in source.neighbors {

            let newPath: 路径 = 路径()


newPath.destination = e.neighbor
            newPath.previous = nil
newPath.total = e.weight


            //将新路径添加到边界
          frontier.enQueue(newPath)

        }
```



```
        //preserve the bestPath
        finalPaths.append(bestPath)


        //remove the bestPath from the frontier
        //frontier.removeAtIndex(pathIndex) - Swift2
        frontier.remove(at: pathIndex)


} //end while


//establish the shortest path as an optional
var shortestPath: Path! = Path()


for itemPath in finalPaths {

    if (itemPath.destination.key == destination.key) {

        if  (shortestPath.total == nil) || (itemPath.total < shortestPath.total) {
            shortestPath = itemPath
        }

    }

}


return shortestPath

    }


    ///an optimized version of Dijkstra's shortest path algorthim
    func processDijkstraWithHeap(_ source: Vertex, destination: Vertex) -> Path! {


        let frontier: PathHeap = PathHeap()
        let finalPaths: PathHeap = PathHeap()


        //use source edges to create the frontier
        for e in source.neighbors {

            let newPath: Path = Path()


            newPath.destination = e.neighbor
            newPath.previous = nil
            newPath.total = e.weight


            //add the new path to the frontier
            frontier.enQueue(newPath)

        }
```

```
        //构建最佳路径
        var bestPath: 路径 = 路径()


        while frontier.count != 0 {

            //使用贪心算法获取最佳路径
            bestPath = Path()
bestPath = frontier.peek()


            //enumerate the bestPath edges
            for e in bestPath.destination.neighbors {

                let newPath: 路径 = 路径()

newPath.destination = e.neighbor
                newPath.previous = bestPath
                newPath.total = bestPath.total + e.weight


                //将新路径添加到边界队列
              frontier.enQueue(newPath)

            }


            //保留与目标匹配的最佳路径
            if (bestPath.destination.key == destination.key) {
              finalPaths.enQueue(bestPath)
            }


              //从边界队列中移除最佳路径
            frontier.deQueue()


        } //结束while循环


        //从堆中获取最短路径
        var shortestPath: Path! = Path()
shortestPath = finalPaths.peek()


        return shortestPath

    }

    //标记：遍历算法

    //带有输入输出闭包函数的广度优先遍历
    func traverse(_ startingv: Vertex, formula: (_ node: inout Vertex) -> ()) {


        //建立一个新的队列
        let graphQueue: Queue<Vertex> = Queue<Vertex>()
```



```
        //construct the best path
        var bestPath: Path = Path()


        while frontier.count != 0 {

            //use the greedy approach to obtain the best path
            bestPath = Path()
            bestPath = frontier.peek()


            //enumerate the bestPath edges
            for e in bestPath.destination.neighbors {

                let newPath: Path = Path()

                newPath.destination = e.neighbor
                newPath.previous = bestPath
                newPath.total = bestPath.total + e.weight


                //add the new path to the frontier
                frontier.enQueue(newPath)

            }


            //preserve the bestPaths that match destination
            if (bestPath.destination.key == destination.key) {
                finalPaths.enQueue(bestPath)
            }


                //remove the bestPath from the frontier
                frontier.deQueue()


        } //end while


        //obtain the shortest path from the heap
        var shortestPath: Path! = Path()
        shortestPath = finalPaths.peek()


        return shortestPath

    }


    //MARK: traversal algorithms


    //bfs traversal with inout closure function
    func traverse(_ startingv: Vertex, formula: (_ node: inout Vertex) -> ()) {


        //establish a new queue
        let graphQueue: Queue<Vertex> = Queue<Vertex>()
```

```
        //将起始顶点入队
        graphQueue.enQueue(startingv)


        while !graphQueue.isEmpty() {

            //遍历下一个队列中的顶点
            var vitem: Vertex = graphQueue.deQueue() as Vertex!


            //将未访问的顶点加入队列
            for e in vitem.neighbors {
                if e.neighbor.visited == false {
                    print("将顶点: \(e.neighbor.key!) 加入队列..")
                 graphQueue.enQueue(e.neighbor)
                }
            }


            /*
备注：这演示了如何调用带有inout参数的闭包。
通过引用传递，无需返回值。
            */

            //调用公式
            formula(&vitem)


        } //结束while循环


        print("图遍历完成..")


    }



    //广度优先搜索
    func traverse(_ startingv: Vertex) {


        //建立一个新的队列
        let graphQueue: Queue<Vertex> = Queue<Vertex>()


        //将起始顶点入队
        graphQueue.enQueue(startingv)


        while !graphQueue.isEmpty() {

            //遍历下一个队列中的顶点
            let vitem = graphQueue.deQueue() as Vertex!

guard vitem != nil else {
                return
            }

            //将未访问的顶点加入队列
            for e in vitem!.neighbors {
```



```
        //queue a starting vertex
        graphQueue.enQueue(startingv)


        while !graphQueue.isEmpty() {

            //traverse the next queued vertex
            var vitem: Vertex = graphQueue.deQueue() as Vertex!


            //add unvisited vertices to the queue
            for e in vitem.neighbors {
                if e.neighbor.visited == false {
                    print("adding vertex: \(e.neighbor.key!) to queue..")
                    graphQueue.enQueue(e.neighbor)
                }
            }


            /*
            notes: this demonstrates how to invoke a closure with an inout parameter.
            By passing by reference no return value is required.
            */

            //invoke formula
            formula(&vitem)


        } //end while


        print("graph traversal complete..")


    }



    //breadth first search
    func traverse(_ startingv: Vertex) {


        //establish a new queue
        let graphQueue: Queue<Vertex> = Queue<Vertex>()


        //queue a starting vertex
        graphQueue.enQueue(startingv)


        while !graphQueue.isEmpty() {

            //traverse the next queued vertex
            let vitem = graphQueue.deQueue() as Vertex!

            guard vitem != nil else {
                return
            }

            //add unvisited vertices to the queue
            for e in vitem!.neighbors {
```

```
                if e.neighbor.visited == false {
                    print("将顶点: \(e.neighbor.key!) 加入队列..")
                 graphQueue.enQueue(e.neighbor)
                }
            }

vitem!.visited = true
            print("遍历顶点: \(vitem!.key!)..")

        } //结束while循环

        print("图遍历完成..")

    } //函数结束

    //使用带尾随闭包的广度优先搜索更新所有值
    func update(startingv: Vertex, formula:((Vertex) -> Bool)) {

        //建立一个新的队列
        let graphQueue: Queue<Vertex> = Queue<Vertex>()

        //将起始顶点入队
       graphQueue.enQueue(startingv)


        while !graphQueue.isEmpty() {

            //遍历下一个队列中的顶点
            let vitem = graphQueue.deQueue() as Vertex!

guard vitem != nil else {
                    return
            }

            //将未访问的顶点加入队列
            for e in vitem!.neighbors {
                if e.neighbor.visited == false {
                    print("将顶点: \(e.neighbor.key!) 加入队列..")
                 graphQueue.enQueue(e.neighbor)
                }
            }


            //应用公式..
            if formula(vitem!) == false {
                print("公式无法更新: \(vitem!.key)")
            }
            else {
                print("遍历顶点: \(vitem!.key!)..")
            }

vitem!.visited = true
```



```
                if e.neighbor.visited == false {
                    print("adding vertex: \(e.neighbor.key!) to queue..")
                    graphQueue.enQueue(e.neighbor)
                }
            }

            vitem!.visited = true
            print("traversed vertex: \(vitem!.key!)..")

        } //end while

        print("graph traversal complete..")

    } //end function

    //use bfs with trailing closure to update all values
    func update(startingv: Vertex, formula:((Vertex) -> Bool)) {

        //establish a new queue
        let graphQueue: Queue<Vertex> = Queue<Vertex>()

        //queue a starting vertex
        graphQueue.enQueue(startingv)


        while !graphQueue.isEmpty() {

            //traverse the next queued vertex
            let vitem = graphQueue.deQueue() as Vertex!

            guard vitem != nil else {
                return
            }

            //add unvisited vertices to the queue
            for e in vitem!.neighbors {
                if e.neighbor.visited == false {
                    print("adding vertex: \(e.neighbor.key!) to queue..")
                    graphQueue.enQueue(e.neighbor)
                }
            }


            //apply formula..
            if formula(vitem!) == false {
                print("formula unable to update: \(vitem!.key)")
            }
            else {
                print("traversed vertex: \(vitem!.key!)..")
            }

            vitem!.visited = true
```

```
        } //结束while循环

        print("图遍历完成..")

    }



}
```

Trie（字典树）

在计算机科学中，Trie，也称为数字树，有时称为基数树或前缀树（因为它们可以通过前缀进行搜索），是一种搜索树——一种有序树数据结构，用于存储动态集合或关联数组，其中键通常是字符串。（维基百科，source）

```
//
//  Trie.swift
//  SwiftStructures
//
// 由韦恩·毕晓普于2014年10月14日创建。
//  版权所有 (c) 2014 Arbutus Software Inc. 保留所有权利。
//
import Foundation


public class Trie {

    private var root: TrieNode!


    init(){
root = TrieNode()
    }



    //构建字典内容的树层次结构
    func append(word keyword: String) {


        //简单情况
guard keyword.length > 0 else {
            return
        }


        var current: TrieNode = root


        while keyword.length != current.level {

            var childToUse: TrieNode!
            let searchKey = keyword.子字符串(到: current.级别 + 1)


            //打印("current 有 \(current.children.count) 个子节点..")
```



```
        } //end while

        print("graph traversal complete..")

    }



}
```

Trie

In computer science, a trie, also called digital tree and sometimes radix tree or prefix tree (as they can be searched by prefixes), is a kind of search tree—an ordered tree data structure that is used to store a dynamic set or associative array where the keys are usually strings. (Wikipedia, source)

```
//
//  Trie.swift
//  SwiftStructures
//
//  Created by Wayne Bishop on 10/14/14.
//  Copyright (c) 2014 Arbutus Software Inc. All rights reserved.
//
import Foundation


public class Trie {

    private var root: TrieNode!


    init(){
        root = TrieNode()
    }



    //builds a tree hierarchy of dictionary content
    func append(word keyword: String) {


        //trivial case
        guard keyword.length > 0 else {
            return
        }


        var current: TrieNode = root


        while keyword.length != current.level {

            var childToUse: TrieNode!
            let searchKey = keyword.substring(to: current.level + 1)


            //print("current has \(current.children.count) children..")
```

```
            //遍历子节点
            for child in current.子节点 {

                if (child.键 == searchKey) {
                    childToUse = child
                    break
                }

            }


            //新节点
            if childToUse == nil {

childToUse = TrieNode()
                childToUse.键 = searchKey
                childToUse.级别 = current.级别 + 1
                current.子节点.添加(childToUse)
            }


            current = childToUse


        } //结束while循环


        //单词结束检查
        if (keyword.长度 == current.级别) {
            current.是结束 = true
            打印("已到达单词末尾！")
            返回
        }


    } //函数结束



    //根据前缀查找单词
    func search(forWord keyword: String) -> Array<String>! {


        //简单情况
guard 关键字。长度 > 0 否则 {
            返回 nil
        }


        变量 current: TrieNode = root
        变量 wordList = Array<String>()


        while keyword.length != current.level {

            var childToUse: TrieNode!
            let searchKey = keyword.子字符串(到: current.级别 + 1)
```



```
            //iterate through child nodes
            for child in current.children {

                if (child.key == searchKey) {
                    childToUse = child
                    break
                }

            }


            //new node
            if childToUse == nil {

                childToUse = TrieNode()
                childToUse.key = searchKey
                childToUse.level = current.level + 1
                current.children.append(childToUse)
            }


            current = childToUse


        } //end while


        //final end of word check
        if (keyword.length == current.level) {
            current.isFinal = true
            print("end of word reached!")
            return
        }



    } //end function



    //find words based on the prefix
    func search(forWord keyword: String) -> Array<String>! {


        //trivial case
        guard keyword.length > 0 else {
            return nil
        }


        var current: TrieNode = root
        var wordList = Array<String>()


        while keyword.length != current.level {

            var childToUse: TrieNode!
            let searchKey = keyword.substring(to: current.level + 1)
```

```
        //打印("查找前缀: \(searchKey)..")


        //遍历所有子节点
        对于 child 在 current.children 中 {

            如果 (child.key == searchKey) {
                childToUse = child
                current = childToUse
                跳出循环
            }

        }


        如果 childToUse == nil {
           返回 nil
        }


    } //结束while循环



    //获取关键字及其所有后代
    如果 ((current.key == keyword) && (current.isFinal)) {
      wordList.append(current.key)
    }


    //仅包含是单词的子节点
    对于 child 在 current.children 中 {

            如果 (child.isFinal == true) {
              wordList.append(child.key)
            }

        }


        return wordList


    } //函数结束


}
```

(GitHub, source)

栈

在计算机科学中，栈是一种抽象数据类型，用作元素的集合，具有两个主要操作：push，将元素添加到集合中；pop，移除最近添加且尚未被移除的元素。元素从栈中弹出的顺序产生了其另一名称——LIFO（后进先出）。此外，peek操作可以在不修改栈的情况下访问栈顶元素。

(Wikipedia, source)

请参见下方的许可信息及原始代码来源（github）


```
        //print("looking for prefix: \(searchKey)..")


        //iterate through any child nodes
        for child in current.children {

            if (child.key == searchKey) {
                childToUse = child
                current = childToUse
                break
            }

        }


        if childToUse == nil {
           return nil
        }


    } //end while



    //retrieve the keyword and any descendants
    if ((current.key == keyword) && (current.isFinal)) {
        wordList.append(current.key)
    }


    //include only children that are words
    for child in current.children {

            if (child.isFinal == true) {
                wordList.append(child.key)
            }

        }


        return wordList


    } //end function


}
```

(GitHub, source)

Stack

In computer science, a stack is an abstract data type that serves as a collection of elements, with two principal operations: push, which adds an element to the collection, and pop, which removes the most recently added element that was not yet removed. The order in which elements come off a stack gives rise to its alternative name, LIFO (for last in, first out). Additionally, a peek operation may give access to the top without modifying the stack. (Wikipedia, source)

See license info below and original code source at (github)

```
//
//  Stack.swift
//  SwiftStructures
//
//  由Wayne Bishop于2014年8月1日创建。
//  版权所有 (c) 2014 Arbutus Software Inc. 保留所有权利。
//
import Foundation


class Stack<T> {

    private var top: Node<T>

    init() {
top = Node<T>()
    }


    //项目数量 - O(n)
    var count: Int {


        //返回简单情况
       guard top.key != nil else {
          return 0
        }


        var current = top
        var x: Int = 1


        //遍历列表
        while current.next != nil {
          current = current.next!
x += 1
        }

        return x

    }


    //向栈中添加项目
    func push(withKey key: T) {


        //返回简单情况
       guard top.key != nil else {
           top.key = key
             return
        }


        //创建新项目
        let childToUse = Node<T>()
       childToUse.key = key


        //将新创建的项目设置为顶部
       childToUse.next = top
```



```
//
//  Stack.swift
//  SwiftStructures
//
//  Created by Wayne Bishop on 8/1/14.
//  Copyright (c) 2014 Arbutus Software Inc. All rights reserved.
//
import Foundation


class Stack<T> {

    private var top: Node<T>

    init() {
        top = Node<T>()
    }


    //the number of items - O(n)
    var count: Int {


        //return trivial case
        guard top.key != nil else {
          return 0
        }


        var current = top
        var x: Int = 1


        //cycle through list
        while current.next != nil {
            current = current.next!
            x += 1
        }

        return x

    }


    //add item to the stack
    func push(withKey key: T) {


        //return trivial case
        guard top.key != nil else {
            top.key = key
            return
        }


        //create new item
        let childToUse = Node<T>()
        childToUse.key = key


        //set new created item at top
        childToUse.next = top
```

```
        top = childToUse

    }


    //从栈中移除元素
    func pop() {

        if self.count > 1 {
            top = top.next
        }
        else {
top.key = nil
        }

    }


    //获取栈顶元素
    func peek() -> T! {


        //确定实例
        if let topitem = top.key {
            return topitem
        }

        else {
            return nil
        }

    }


    //检查是否为空
    func isEmpty() -> Bool {

        if self.count == 0 {
            return true
        }

        else {
            return false
        }

    }


}
```

> MIT 许可证（MIT）
>
> 版权所有（c）2015，韦恩·毕晓普（Wayne Bishop）和阿布图斯软件公司（Arbutus Software Inc.）。
>
> 特此免费授予任何获得本软件及相关文档文件（“软件”）副本的人无限制地使用本软件的权利，包括但不限于使用、复制、修改、合并、发布、分发、再许可及/或销售软件副本，并允许向其提供软件的人这样做，须遵守以下条件：


```
        top = childToUse

    }


    //remove item from the stack
    func pop() {

        if self.count > 1 {
            top = top.next
        }
        else {
            top.key = nil
        }

    }


    //retrieve the top most item
    func peek() -> T! {


        //determine instance
        if let topitem = top.key {
            return topitem
        }

        else {
            return nil
        }

    }


    //check for value
    func isEmpty() -> Bool {

        if self.count == 0 {
            return true
        }

        else {
            return false
        }

    }


}
```

> The MIT License (MIT)
>
> Copyright (c) 2015, Wayne Bishop & Arbutus Software Inc.
>
> Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

上述版权声明和本许可声明应包含在软件的所有副本或重要部分中。

本软件按“原样”提供，不附带任何形式的明示或暗示保证，包括但不限于对适销性、特定用途适用性及非侵权的保证。在任何情况下，作者或版权持有人均不对因本软件或本软件的使用或其他交易而产生的任何索赔、损害或其他责任承担责任，无论是在合同诉讼、侵权或其他方面。

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

第48章：Swift 高级函数

高级函数如map、flatMap、filter和reduce用于操作各种集合类型，如数组（Array）和字典（Dictionary）。高级函数通常需要较少的代码，并且可以链式调用，以简洁的方式构建复杂逻辑。

第48.1节：扁平化多维数组

要将多维数组扁平化为一维数组，可以使用flatMap高级函数。另一个用例是忽略数组中的nil值并映射值。让我们通过示例来查看：

假设我们有一个多维城市数组，想要按升序排序城市名称列表。在这种情况下，我们可以使用flatMap函数，如下所示：

```
let arrStateName = [["阿拉斯加", "爱荷华", "密苏里", "新墨西哥"], ["纽约", "得克萨斯",
"华盛顿", "马里兰"], ["新泽西", "弗吉尼亚", "佛罗里达", "科罗拉多"]]
```

从多维数组准备一维列表，

```
let arrFlatStateList = arrStateName.flatMap({ $0 }) // ["阿拉斯加", "爱荷华", "密苏里", "新墨西哥",
"纽约", "得克萨斯", "华盛顿", "马里兰", "新泽西", "弗吉尼亚", "佛罗里达", "科罗拉多"]
```

对于排序数组值，我们可以使用链式操作或对扁平数组进行排序。下面的示例展示了链式操作，

```
// Swift 2.3 语法
let arrSortedStateList = arrStateName.flatMap({ $0 }).sort(<) // ["阿拉斯加",    "科罗拉多",
"佛罗里达", "爱荷华", "马里兰", "密苏里", "新泽西", "新墨西哥", "纽约", "得克萨斯",
"弗吉尼亚", "华盛顿"]

// Swift 3 语法
let arrSortedStateList = arrStateName.flatMap({ $0 }).sorted(by: <) // ["阿拉斯加", "科罗拉多",
"佛罗里达", "爱荷华", "马里兰", "密苏里", "新泽西", "新墨西哥", "纽约", "得克萨斯",
"弗吉尼亚", "华盛顿"]
```

第48.2节：高级函数介绍

让我们通过一个场景更好地理解高级函数，

```
结构体 User {
    变量 name: 字符串
    变量 age: 整数
    变量 country: 字符串?
}

//用户信息
let user1 = User(name: "约翰", age: 24, country: "美国")
let user2 = User(name: "陈", age: 20, country: nil)
let user3 = User(name: "摩根", age: 30, country: nil)
let user4 = User(name: "瑞秋", age: 20, country: "英国")
let user5 = User(name: "凯蒂", age: 23, country: "美国")
let user6 = User(name: "大卫", age: 35, country: "美国")
let user7 = User(name: "鲍勃",age: 22, country: nil)

//用户数组列表
```


Chapter 48: Swift Advance functions

Advance functions like `map`, `flatMap`, `filter`, and `reduce` are used to operate on various collection types like Array and Dictionary. Advance functions typically require little code and can be chained together in order to build up complex logic in a concise way.

Section 48.1: Flatten multidimensional array

To flatten multidimensional array into single dimension, flatMap advance functions is used. Other use case is to neglect nil value from array & mapping values. Let's check with example:

Suppose We have an multidimensional array of cities & we want to sorted city name list in ascending order. In that case we can use flatMap function like:

```
let arrStateName = [["Alaska", "Iowa", "Missouri", "New Mexico"], ["New York", "Texas",
"Washington", "Maryland"], ["New Jersey", "Virginia", "Florida", "Colorado"]]
```

Preparing a single dimensional list from multidimensional array,

```
let arrFlatStateList = arrStateName.flatMap({ $0 }) // ["Alaska", "Iowa", "Missouri", "New Mexico",
"New York", "Texas", "Washington", "Maryland", "New Jersey", "Virginia", "Florida", "Colorado"]
```

For sorting array values, we can use chaining operation or sort flatten array. Here below example showing chaining operation,

```
// Swift 2.3 syntax
let arrSortedStateList = arrStateName.flatMap({ $0 }).sort(<) // ["Alaska",     "Colorado",
"Florida", "Iowa", "Maryland", "Missouri", "New Jersey", "New Mexico", "New York", "Texas",
"Virginia", "Washington"]

// Swift 3 syntax
let arrSortedStateList = arrStateName.flatMap({ $0 }).sorted(by: <) // ["Alaska", "Colorado",
"Florida", "Iowa", "Maryland", "Missouri", "New Jersey", "New Mexico", "New York", "Texas",
"Virginia", "Washington"]
```

Section 48.2: Introduction with advance functions

Let's take an scenario to understand advance function in better way,

```
struct User {
    var name: String
    var age: Int
    var country: String?
}

//User's information
let user1 = User(name: "John", age: 24, country: "USA")
let user2 = User(name: "Chan", age: 20, country: nil)
let user3 = User(name: "Morgan", age: 30, country: nil)
let user4 = User(name: "Rachel", age: 20, country: "UK")
let user5 = User(name: "Katie", age: 23, country: "USA")
let user6 = User(name: "David", age: 35, country: "USA")
let user7 = User(name: "Bob",age: 22, country: nil)

//User's array list
```

```
let arrUser = [user1, user2, user3, user4, user5, user6, user7]
```

映射函数：

使用 map 遍历集合，并对集合中的每个元素执行相同的操作。map 函数返回一个数组，包含对每个元素应用映射或转换函数后的结果。

```
//从数组中获取所有用户的名字
let arrUserName = arrUser.map({ $0.name }) // ["约翰", "陈", "摩根", "瑞秋", "凯蒂",
"大卫", "鲍勃"]
```

扁平映射函数：

最简单的用法如其名，用于扁平化一个集合的集合。

```
// 获取所有用户的国家名称并忽略空值。
let arrCountry = arrUser.flatMap({ $0.country }) // ["USA", "UK", "USA", "USA"]
```

过滤函数：

使用 filter 遍历集合，并返回一个只包含满足包含条件的元素的数组。

```
// 从用户列表数组中过滤出美国用户。
let arrUSAUsers = arrUser.filter({ $0.country == "USA" }) // [user1, user5, user6]

// 链式调用方法获取居住在美国的用户的姓名
let arrUserList = arrUser.filter({ $0.country == "USA" }).map({ $0.name }) // ["John", "Katie",
"David"]
```

归约（Reduce）：

使用 reduce 将集合中的所有项合并，创建一个新的单一值。

Swift 2.3：

```
//获取用户的总年龄。
let arrUserAge = arrUser.map({ $0.age }).reduce(0, combine: { $0 + $1 }) //174

//准备所有用户年龄数组
let strUserName = arrUserName.reduce("", combine: { $0 == "" ? $1 : $0 + ", " + $1 }) // John,
Chan, Morgan, Rachel, Katie, David, Bob
```

Swift 3:

```
//获取用户的总年龄。
let arrUserAge = arrUser.map({ $0.age }).reduce(0, { $0 + $1 }) //174

//准备所有用户年龄数组
let strUserName = arrUserName.reduce("", { $0 == "" ? $1 : $0 + ", " + $1 }) // John, Chan, Morgan,
Rachel, Katie, David, Bob
```



```
let arrUser = [user1, user2, user3, user4, user5, user6, user7]
```

Map Function:

Use map to loop over a collection and apply the same operation to each element in the collection. The map function returns an array containing the results of applying a mapping or transform function to each item.

```
//Fetch all the user's name from array
let arrUserName = arrUser.map({ $0.name }) // ["John", "Chan", "Morgan", "Rachel", "Katie",
"David", "Bob"]
```

Flat-Map Function:

The simplest use is as the name suggests to flatten a collection of collections.

```
// Fetch all user country name & ignore nil value.
let arrCountry = arrUser.flatMap({ $0.country }) // ["USA", "UK", "USA", "USA"]
```

Filter Function:

Use filter to loop over a collection and return an Array containing only those elements that match an include condition.

```
// Filtering USA user from the array user list.
let arrUSAUsers = arrUser.filter({ $0.country == "USA" }) // [user1, user5, user6]

// User chaining methods to fetch user's name who live in USA
let arrUserList = arrUser.filter({ $0.country == "USA" }).map({ $0.name }) // ["John", "Katie",
"David"]
```

Reduce:

Use reduce to combine all items in a collection to create a single new value.

Swift 2.3:

```
//Fetch user's total age.
let arrUserAge = arrUser.map({ $0.age }).reduce(0, combine: { $0 + $1 }) //174

//Prepare all user name string with separated by comma
let strUserName = arrUserName.reduce("", combine: { $0 == "" ? $1 : $0 + ", " + $1 }) // John,
Chan, Morgan, Rachel, Katie, David, Bob
```

Swift 3:

```
//Fetch user's total age.
let arrUserAge = arrUser.map({ $0.age }).reduce(0, { $0 + $1 }) //174

//Prepare all user name string with separated by comma
let strUserName = arrUserName.reduce("", { $0 == "" ? $1 : $0 + ", " + $1 }) // John, Chan, Morgan,
Rachel, Katie, David, Bob
```

第49章：完成处理器

几乎所有应用程序都使用异步函数来防止代码阻塞主线程。

第49.1节：无输入参数的完成处理器

```
func sampleWithCompletion(completion:@escaping (()-> ())){
    let delayInSeconds = 1.0
DispatchQueue.main.asyncAfter(deadline: DispatchTime.now() + delayInSeconds) {

        completion()

    }
}

//调用函数
sampleWithCompletion {
    print("一秒后")
}
```

第49.2节：带输入参数的完成处理器

```
枚举 ReadResult{
    案例 成功
    案例 失败
    案例 待定
}

结构体 OutpuData {
    变量 data = Data()
    变量 result: ReadResult
    变量 error: Error?
}

函数 readData(from url: String, completion: @escaping (OutpuData) ->  Void) {
    变量 _data = OutpuData(data: Data(), result: .待定, error: nil)
DispatchQueue.global().async {
        让 url=URL(string: url)
        尝试 {
            让 rawData = try Data(contentsOf: url!)
            _data.result = .成功
          _data.data = rawData
completion(_data)
        }
捕获 let error {
            _data.result = .失败
          _data.error = error
            completion(_data)
        }

    }
}

readData(from: "https://raw.githubusercontent.com/trev/bearcal/master/sample-data-large.json") {
(output) in
    切换 输出。结果 {
    情况 .成功:
        跳出
    情况 .失败:
```


Chapter 49: Completion Handler

Virtually all apps are using asynchronous functions to keep the code from blocking the main thread.

Section 49.1: Completion handler with no input argument

```
func sampleWithCompletion(completion:@escaping (()-> ())){
    let delayInSeconds = 1.0
    DispatchQueue.main.asyncAfter(deadline: DispatchTime.now() + delayInSeconds) {

        completion()

    }
}

//Call the function
sampleWithCompletion {
    print("after one second")
}
```

Section 49.2: Completion handler with input argument

```
enum ReadResult{
    case Successful
    case Failed
    case Pending
}

struct OutpuData {
    var data = Data()
    var result: ReadResult
    var error: Error?
}

func readData(from url: String, completion: @escaping (OutpuData) ->  Void) {
    var _data = OutpuData(data: Data(), result: .Pending, error: nil)
    DispatchQueue.global().async {
        let url=URL(string: url)
        do {
            let rawData = try Data(contentsOf: url!)
            _data.result = .Successful
            _data.data = rawData
            completion(_data)
        }
        catch let error {
            _data.result = .Failed
            _data.error = error
            completion(_data)
        }

    }
}

readData(from: "https://raw.githubusercontent.com/trev/bearcal/master/sample-data-large.json") {
(output) in
    switch output.result {
    case .Successful:
        break
    case .Failed:
```

```
            跳出
        情况 .等待中:
            跳出

        }
}
```



```
            break
        case .Pending:
            break

        }
}
```

第50章：Kitura的Swift HTTP服务器

使用Kitura的Swift服务器

Kitura 是一个用Swift编写的Web框架，专为Web服务创建。它非常容易设置HTTP请求。环境方面，需要安装了XCode的OS X，或者运行Swift 3.0的Linux系统。

第50.1节：Hello world应用程序

配置

首先，创建一个名为 Package.swift 的文件。这个文件告诉 Swift 编译器库的位置。在这个 Hello World 示例中，我们使用的是 GitHub 仓库。我们需要Kitura和HeliumLogger。将以下代码放入 Package.swift 文件中。它指定了项目名称为*kitura-helloworld*，同时也指定了依赖的 URL。

```
import PackageDescription
let package = Package(
    name: "kitura-helloworld",
       dependencies: [
               .Package(url: "https://github.com/IBM-Swift/HeliumLogger.git", majorVersion: 1, minor:
6),
                .Package(url: "https://github.com/IBM-Swift/Kitura.git", majorVersion: 1, minor: 6) ] )
```

接下来，创建一个名为 Sources 的文件夹。在里面创建一个名为 main.swift 的文件。这是我们实现该应用所有逻辑的文件。将以下代码输入到这个 main 文件中。

导入库并启用日志记录

```
import Kitura
import Foundation
import HeliumLogger

HeliumLogger.use()
```

添加路由器。路由器指定 HTTP 请求的路径、类型等。这里我们添加了一个 GET 请求处理器，它打印*Hello world*，然后添加了一个 POST 请求处理器，从请求中读取纯文本并将其返回。

```
let router = Router()

router.get("/get") {
    request, response, next in
    response.send("Hello, World!")
   next()
}

router.post("/post") {
    request, response, next in
    var string: String?
    do{
string = try request.readString()

    } catch let error {
string = error.localizedDescription
    }
response.send("Value \(string!) received.")
   next()
```


Chapter 50: Swift HTTP server by Kitura

Swift server with Kitura

Kitura is a web framework written in swift that is created for web services. It's very easy to set up for HTTP requests. For environment, it needs either OS X with XCode installed, or Linux running swift 3.0.

Section 50.1: Hello world application

Configuration

First, create a file called Package.swift. This is the file that tells swift compiler where the libraries are located. In this hello world example, we are using GitHub repos. We need Kitura and HeliumLogger. Put the following code inside Package.swift. It specified the name of the project as *kitura-helloworld* and also the dependency urls.

```
import PackageDescription
let package = Package(
    name: "kitura-helloworld",
        dependencies: [
            .Package(url: "https://github.com/IBM-Swift/HeliumLogger.git", majorVersion: 1, minor:
6),
            .Package(url: "https://github.com/IBM-Swift/Kitura.git", majorVersion: 1, minor: 6) ] )
```

Next, create a folder called Sources. Inside, create a file called main.swift. This is the file that we implement all the logic for this application. Enter the following code into this main file.

Import libraries and enable logging

```
import Kitura
import Foundation
import HeliumLogger

HeliumLogger.use()
```

Adding a router. Router specifies a path, type, etc of the HTTP request. Here we are adding a GET request handler which prints *Hello world*, and then a post request that reads plain text from the request and then send it back.

```
let router = Router()

router.get("/get") {
    request, response, next in
    response.send("Hello, World!")
    next()
}

router.post("/post") {
    request, response, next in
    var string: String?
    do{
        string = try request.readString()

    } catch let error {
        string = error.localizedDescription
    }
    response.send("Value \(string!) received.")
    next()
```

```
}
```

指定一个端口来运行服务

```
let port = 8080
```

将路由器和端口绑定在一起，并将它们作为HTTP服务添加

```
Kitura.addHTTPServer(onPort: port, with: router)
Kitura.run()
```

执行

导航到包含 Package.swift 文件和 Resources 文件夹的根目录。运行以下命令。Swift 编译器将自动下载 Package.swift 中提到的资源到 Packages 文件夹，然后将这些资源与 main.swift 一起编译

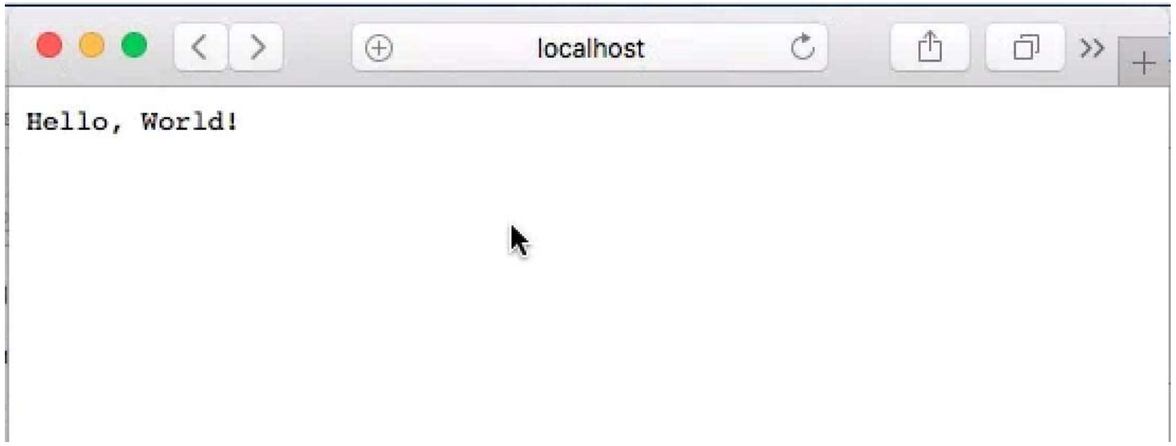
```
swift build
```

构建完成后，可执行文件将放置在此位置。双击此可执行文件以启动服务器。

```
.build/debug/kitura-helloworld
```

验证

打开浏览器，输入localhost:8080/get作为网址并回车。应该会显示 hello world 页面。



打开 HTTP 请求应用，向localhost:8080/post发送纯文本。响应字符串将正确显示输入的文本。


```
}
```

Specify a port to run the service

```
let port = 8080
```

Bind the router and port together and add them as HTTP service

```
Kitura.addHTTPServer(onPort: port, with: router)
Kitura.run()
```

Execute

Navigate to the root folder with Package.swift file and Resources folder. Run the following command. Swift compiler will automatically download the mentioned resources in Package.swift into Packages folder, and then compile these resources with main.swift

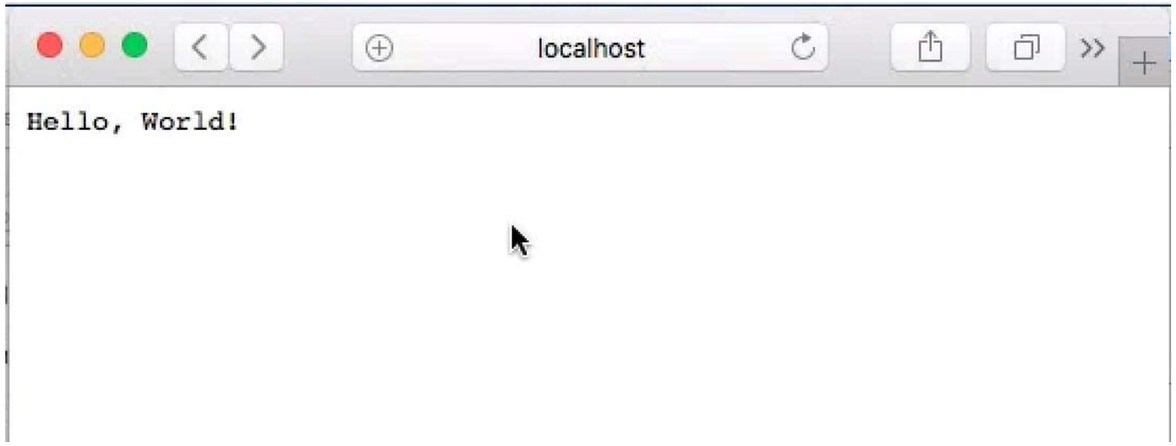
```
swift build
```

When the build is finished, executable will be placed at this location. Double click this executable to start the server.

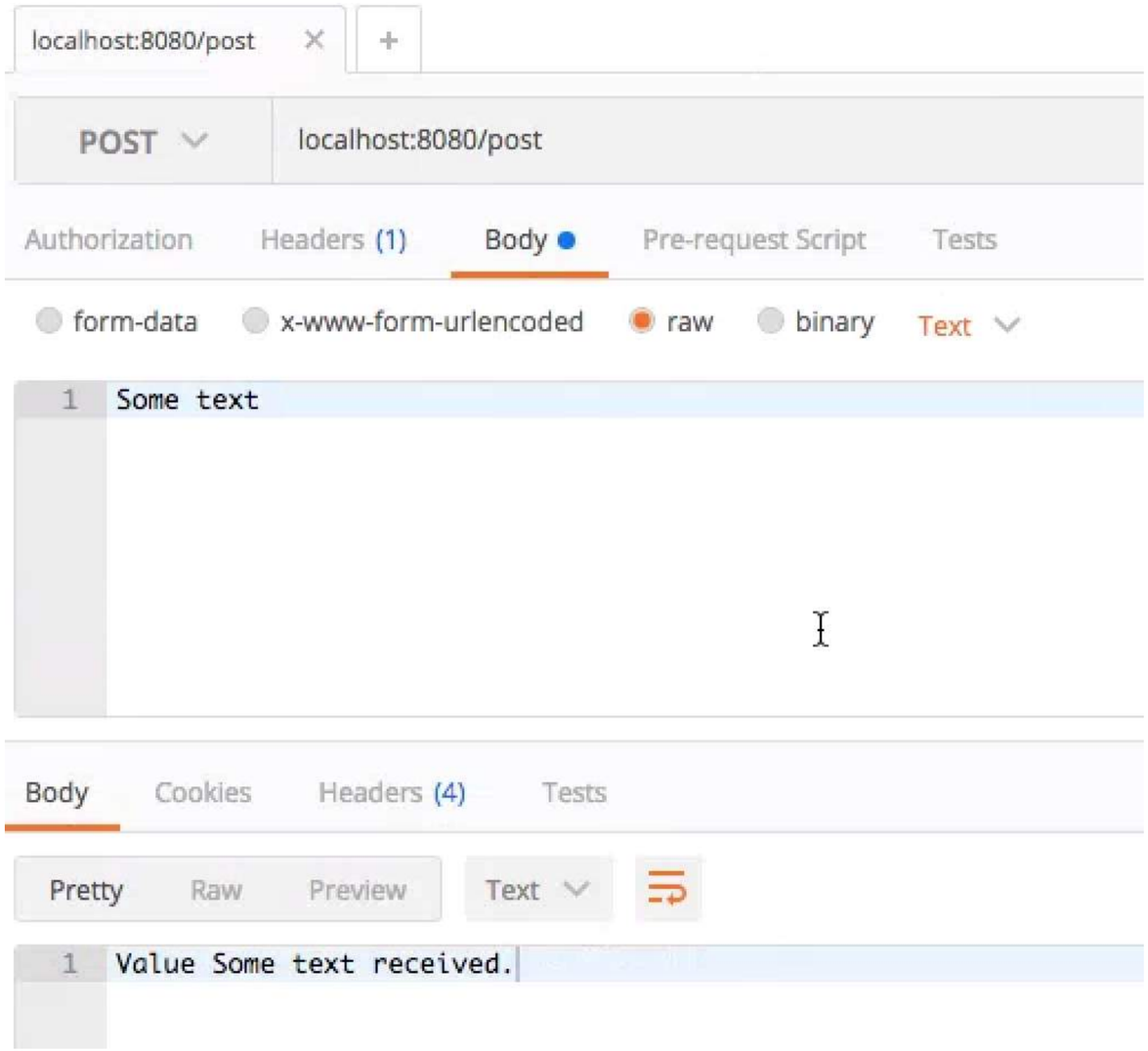
```
.build/debug/kitura-helloworld
```

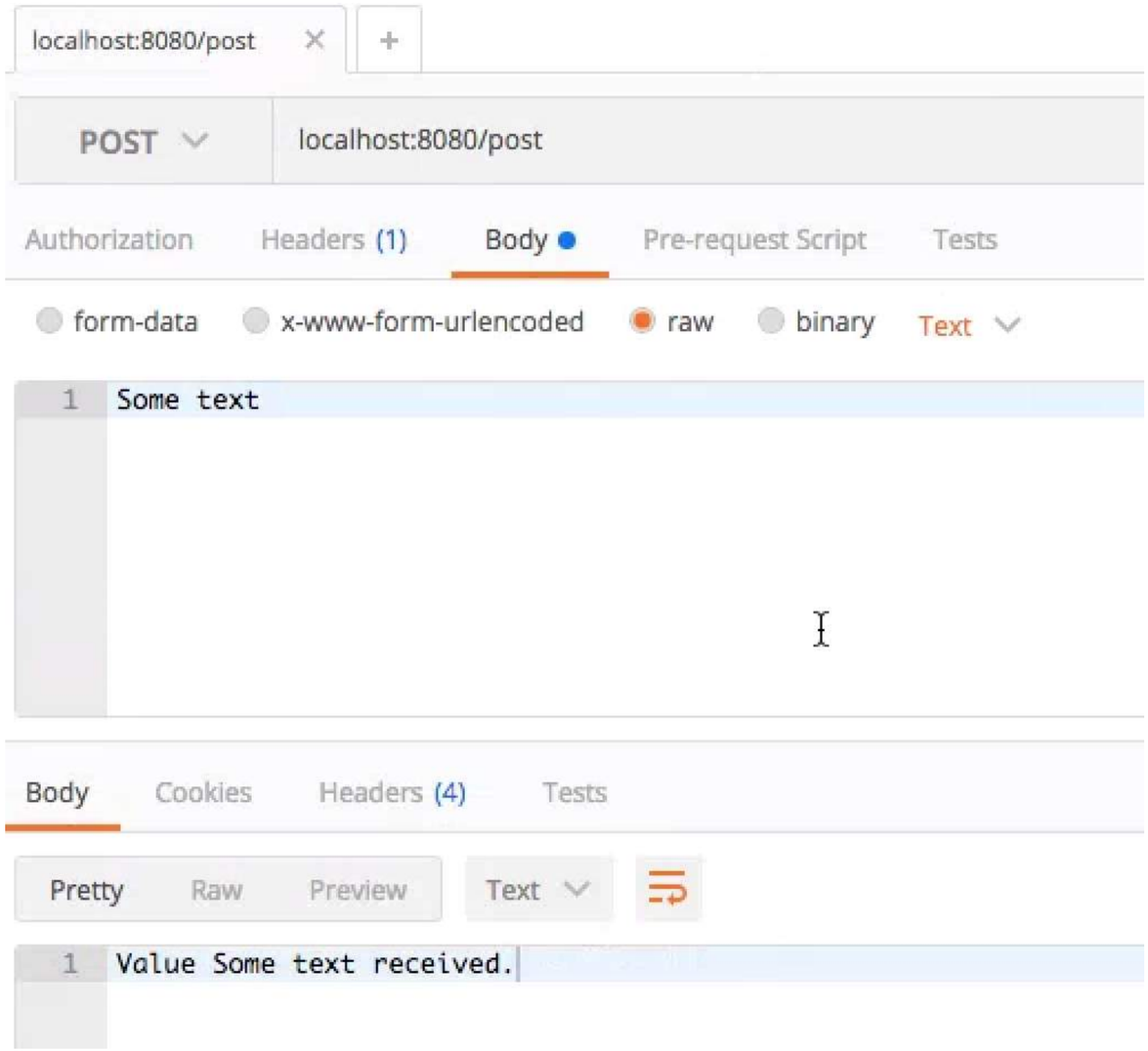
Validate

Open a browser, type in `localhost:8080/get` as url and hit enter. The hello world page should come out.



Open a HTTP request app, post plain text to `localhost:8080/post`. The respond string will show the entered text correctly.


localhost:8080/post
POST
localhost:8080/post
Authorization
Headers (1)
Body
Pre-request Script
Tests
form-data
x-www-form-urlencoded
raw
binary
Text
1
Some text
Body
Cookies
Headers (4)
Tests
Pretty
Raw
Preview
Text
1
Value Some text received.


localhost:8080/post
POST
localhost:8080/post
Authorization
Headers (1)
Body
Pre-request Script
Tests
form-data
x-www-form-urlencoded
raw
binary
Text
1
Some text
Body
Cookies
Headers (4)
Tests
Pretty
Raw
Preview
Text
1
Value Some text received.

第51章：从字符串生成首字母 UIImage

这是一个将生成某人首字母 UIImage 的类。哈利·波特将生成一个 HP 的图像。

第51.1节：InitialsImageFactory

```
class InitialsImageFactory: NSObject {

class func imageWith(name: String?) -> UIImage? {

let frame = CGRect(x: 0, y: 0, width: 50, height: 50)
let nameLabel = UILabel(frame: frame)
nameLabel.textAlignment = .center
nameLabel.backgroundColor = .lightGray
nameLabel.textColor = .white
nameLabel.font = UIFont.boldSystemFont(ofSize: 20)
var initials = ""

if let initialsArray = name?.components(separatedBy: " ") {

  if let firstWord = initialsArray.first {
    if let firstLetter = firstWord.characters.first {
      initials += String(firstLetter).capitalized
    }

  }
  if initialsArray.count > 1, let lastWord = initialsArray.last {
    if let lastLetter = lastWord.characters.first {
initials += String(lastLetter).capitalized
    }

  }
} else {
  return nil
}

nameLabel.text = initials
UIGraphicsBeginImageContext(frame.size)
if let currentContext = UIGraphicsGetCurrentContext() {
  nameLabel.layer.render(in: currentContext)
  let nameImage = UIGraphicsGetImageFromCurrentImageContext()
  return nameImage
}
return nil
}

}
```


Chapter 51: Generate UIImage of Initials from String

This is a class that will generate a UIImage of a person's initials. Harry Potter would generate an image of HP.

Section 51.1: InitialsImageFactory

```
class InitialsImageFactory: NSObject {

class func imageWith(name: String?) -> UIImage? {

let frame = CGRect(x: 0, y: 0, width: 50, height: 50)
let nameLabel = UILabel(frame: frame)
nameLabel.textAlignment = .center
nameLabel.backgroundColor = .lightGray
nameLabel.textColor = .white
nameLabel.font = UIFont.boldSystemFont(ofSize: 20)
var initials = ""

if let initialsArray = name?.components(separatedBy: " ") {

  if let firstWord = initialsArray.first {
    if let firstLetter = firstWord.characters.first {
      initials += String(firstLetter).capitalized
    }

  }
  if initialsArray.count > 1, let lastWord = initialsArray.last {
    if let lastLetter = lastWord.characters.first {
      initials += String(lastLetter).capitalized
    }

  }
} else {
  return nil
}

nameLabel.text = initials
UIGraphicsBeginImageContext(frame.size)
if let currentContext = UIGraphicsGetCurrentContext() {
  nameLabel.layer.render(in: currentContext)
  let nameImage = UIGraphicsGetImageFromCurrentImageContext()
  return nameImage
}
return nil
}

}
```

第52章：设计模式 - 创建型

设计模式是软件开发中经常出现问题的通用解决方案。以下是结构化和设计代码的标准化最佳实践模板，以及这些设计模式适用的常见场景示例。

创建型设计模式 抽象对象的实例化过程，使系统更独立于创建、组合和表示的过程。

第52.1节：单例模式

单例模式是一种常用的设计模式，包含一个类的单个实例，该实例在整个程序中共享。

在下面的示例中，我们创建了一个static属性，该属性持有Foo类的一个实例。请记住，static属性在类的所有对象之间共享，且不能被子类重写。

```
public class Foo
{
    static let shared = Foo()

    // 用于防止类被直接实例化
    private init() {}

    func doSomething()
    {
        print("做某事")
    }
}
```

用法：

```
Foo.shared.doSomething()
```

务必记住private初始化方法：

> 这确保你的单例真正唯一，并防止外部对象通过访问控制自行创建类的实例。由于Swift中所有对象都带有默认的公共初始化方法，你需要重写init并将其设为私有。KrakenDev

第52.2节：建造者模式

> 建造者模式是一种对象创建软件设计模式。与旨在实现多态性的抽象工厂模式和工厂方法模式不同，建造者模式的目的是解决望远镜构造函数反模式。望远镜构造函数反模式是指随着对象构造函数参数组合的增加，构造函数列表呈指数增长。建造者模式不是使用大量构造函数，而是使用另一个对象——建造者，逐步接收每个初始化参数，然后一次性返回构建完成的对象。
>
> *-维基百科*

Chapter 52: Design Patterns - Creational

Design patterns are general solutions to problems that frequently occur in software development. The following are templates of standardized best practices in structuring and designing code, as well as examples of common contexts in which these design patterns would be appropriate.

Creational design patterns abstract the instantiation of objects to make a system more independent of the process of creation, composition, and representation.

Section 52.1: Singleton

Singletons are a frequently used design pattern which consists of a single instance of a class that is shared throughout a program.

In the following example, we create a `static` property that holds an instance of the `Foo` class. Remember that a `static` property is shared between all objects of a class and can't be overwritten by subclassing.

```
public class Foo
{
    static let shared = Foo()

    // Used for preventing the class from being instantiated directly
    private init() {}

    func doSomething()
    {
        print("Do something")
    }
}
```

Usage:

```
Foo.shared.doSomething()
```

Be sure to remember the `private` initializer:

> This makes sure your singletons are truly unique and prevents outside objects from creating their own instances of your class through virtue of access control. Since all objects come with a default public initializer in Swift, you need to override your init and make it private. KrakenDev

Section 52.2: Builder Pattern

> The builder pattern is an **object creation software design pattern**. Unlike the abstract factory pattern and the factory method pattern whose intention is to enable polymorphism, the intention of the builder pattern is to find a solution to the telescoping constructor anti-pattern. The telescoping constructor anti-pattern occurs when the increase of object constructor parameter combination leads to an exponential list of constructors. Instead of using numerous constructors, the builder pattern uses another object, a builder, that receives each initialization parameter step by step and then returns the resulting constructed object at once.
>
> *-Wikipedia*

建造者模式的主要目标是从对象创建时设置默认配置。它是将要构建的对象与所有其他相关构建对象之间的中介。

示例：

为了更清楚地说明，让我们来看一个汽车制造者的例子。

假设我们有一个汽车类，包含许多选项来创建对象，例如：

- 颜色。
- 座位数。
- 车轮数。
- 类型。
- 变速箱类型。
- 发动机。
- 安全气囊可用性。

```
import UIKit

enum CarType {
    case

运动型多用途车（Sportage），
    轿车
}

enum 齿轮类型 {
    case

手动,
    自动
}

struct 电机 {
    var 编号: String
    var 名称: String
    var 型号: String
    var 气缸数: UInt8
}

class 汽车: CustomStringConvertible {
    var 颜色: UIColor
    var 座位数: UInt8
    var 轮子数: UInt8
    var 类型: 汽车类型
    var 齿轮类型: 齿轮类型
    var 电机: 电机
    var shouldHasAirbags: Bool

    var description: String {
        return "颜色: \(color)座位数: \(numberOfSeats)轮子数:
\(numberOfWheels) 类型: \(gearType)发动机: \(motor)安全气囊是否配备: \(shouldHasAirbags)"}


            init(color: UIColor, numberOfSeats: UInt8, numberOfWheels: UInt8, type: CarType, gearType:
GearType, motor: Motor, shouldHasAirbags: Bool) {

        self.color = color
        self.numberOfSeats = numberOfSeats
        self.numberOfWheels = numberOfWheels
```


The main goal of the builder pattern is to setup a default configuration for an object from its creation. It is an intermediary between the object will be built and all other objects related to building it.

Example:

To make it more clear, let's take a look at a *Car Builder* example.

Consider that we have a *Car* class contains many options to create an object, such as:

- Color.
- Number of seats.
- Number of wheels.
- Type.
- Gear type.
- Motor.
- Airbag availability.

```
import UIKit

enum CarType {
    case

    sportage,
    saloon
}

enum GearType {
    case

    manual,
    automatic
}

struct Motor {
    var id: String
    var name: String
    var model: String
    var numberOfCylinders: UInt8
}

class Car: CustomStringConvertible {
    var color: UIColor
    var numberOfSeats: UInt8
    var numberOfWheels: UInt8
    var type: CarType
    var gearType: GearType
    var motor: Motor
    var shouldHasAirbags: Bool

    var description: String {
        return "color: \(color)\nNumber of seats: \(numberOfSeats)\nNumber of Wheels:
\(numberOfWheels)\n Type: \(gearType)\nMotor: \(motor)\nAirbag Availability: \(shouldHasAirbags)"
    }

    init(color: UIColor, numberOfSeats: UInt8, numberOfWheels: UInt8, type: CarType, gearType:
GearType, motor: Motor, shouldHasAirbags: Bool) {

        self.color = color
        self.numberOfSeats = numberOfSeats
        self.numberOfWheels = numberOfWheels
```

```
        self.type = type
        self.gearType = gearType
        self.motor = motor
        self.shouldHasAirbags = shouldHasAirbags

    }
}
```

创建一个汽车对象：

```
let aCar = Car(color: UIColor.black,
               numberOfSeats: 4,
               numberOfWheels: 4,
               type: .saloon,
gearType: .automatic,
               motor: Motor(id: "101", name: "Super Motor",
                            model: "c4", numberOfCylinders: 6),
               shouldHasAirbags: true)

print(aCar)

/* 打印
颜色: UIExtendedGrayColorSpace 0 1
 座位数: 4
车轮数: 4
类型: automatic
发动机: Motor(id: "101", name: "Super Motor", model: "c4", numberOfCylinders: 6)
 安全气囊可用性: true
*/
```

创建汽车对象时出现的问题是，汽车需要许多配置数据才能创建。

对于应用建造者模式，初始化参数应具有默认值，且在需要时可以更改。

CarBuilder 类：

```
class CarBuilder {
    var color: UIColor = UIColor.black
    var numberOfSeats: UInt8 = 5
    var numberOfWheels: UInt8 = 4
    var type: CarType = .saloon
    var gearType: GearType = .automatic
    var motor: Motor = Motor(id: "111", name: "Default Motor",
                             model: "T9", numberOfCylinders: 4)
    var shouldHasAirbags: Bool = false

    func buildCar() -> Car {
        return Car(color: color, numberOfSeats: numberOfSeats, numberOfWheels: numberOfWheels,
type: type, gearType: gearType, motor: motor, shouldHasAirbags: shouldHasAirbags)
    }
}
```

CarBuilder 类定义了可以更改的属性，以编辑所创建汽车对象的值。

让我们使用CarBuilder来制造新车：

```
var builder = CarBuilder()
// 目前，构建器创建的汽车是默认配置。
```



```
        self.type = type
        self.gearType = gearType
        self.motor = motor
        self.shouldHasAirbags = shouldHasAirbags

    }
}
```

Creating a car object:

```
let aCar = Car(color: UIColor.black,
               numberOfSeats: 4,
               numberOfWheels: 4,
               type: .saloon,
               gearType: .automatic,
               motor: Motor(id: "101", name: "Super Motor",
                            model: "c4", numberOfCylinders: 6),
               shouldHasAirbags: true)

print(aCar)

/* Printing
 color: UIExtendedGrayColorSpace 0 1
 Number of seats: 4
 Number of Wheels: 4
 Type: automatic
 Motor: Motor(id: "101", name: "Super Motor", model: "c4", numberOfCylinders: 6)
 Airbag Availability: true
*/
```

The **problem** arises when creating a car object is that the car requires many configuration data to be created.

For applying the Builder Pattern, the initializer parameters should have default values *which are changeable if needed*.

CarBuilder class:

```
class CarBuilder {
    var color: UIColor = UIColor.black
    var numberOfSeats: UInt8 = 5
    var numberOfWheels: UInt8 = 4
    var type: CarType = .saloon
    var gearType: GearType = .automatic
    var motor: Motor = Motor(id: "111", name: "Default Motor",
                             model: "T9", numberOfCylinders: 4)
    var shouldHasAirbags: Bool = false

    func buildCar() -> Car {
        return Car(color: color, numberOfSeats: numberOfSeats, numberOfWheels: numberOfWheels,
type: type, gearType: gearType, motor: motor, shouldHasAirbags: shouldHasAirbags)
    }
}
```

The `CarBuilder` class defines properties that could be changed to to edit the values of the created car object.

Let's build new cars by using the `CarBuilder`:

```
var builder = CarBuilder()
// currently, the builder creates cars with default configuration.
```

```
let defaultCar = builder.buildCar()
//print(defaultCar.description)
/* 打印
颜色: UIExtendedGrayColorSpace 0 1
 座位数: 5
车轮数: 4
类型: automatic
发动机: Motor(id: "111", name: "默认发动机", model: "T9", numberOfCylinders: 4)
 安全气囊可用性: false
*/

builder.shouldHasAirbags = true
// 现在，构建器创建的汽车是默认配置，
// 但对安全气囊的可用性做了小修改

let safeCar = builder.buildCar()
print(safeCar.description)
/* 打印
颜色: UIExtendedGrayColorSpace 0 1
 座位数: 5
车轮数: 4
类型: automatic
发动机: Motor(id: "111", name: "默认发动机", model: "T9", numberOfCylinders: 4)
 安全气囊可用性: true
 */

builder.color = UIColor.purple
// 现在，构建器创建具有默认配置的汽车
// 并带有一些额外功能：安全气囊可用且颜色为紫色

let femaleCar = builder.buildCar()
print(femaleCar)
/* 输出
颜色：UIExtendedSRGBColorSpace 0.5 0 0.5 1
 座位数：5
车轮数: 4
类型: automatic
发动机: Motor(id: "111", name: "默认发动机", model: "T9", numberOfCylinders: 4)
 安全气囊可用性: true
*/
```

应用建造者模式的好处是通过设置默认值，轻松创建应包含大量配置的对象，同时也方便更改这些默认值。

进一步扩展：

作为良好实践，所有需要默认值的属性应放在一个独立协议中，该协议应由类本身及其构建器实现。

回到我们的示例，创建一个名为CarBluePrint的新协议：

```
import UIKit

enum CarType {
    case

运动型多用途车（Sportage），
    轿车
}

enum GearType {
```



```
let defaultCar = builder.buildCar()
//print(defaultCar.description)
/* prints
 color: UIExtendedGrayColorSpace 0 1
 Number of seats: 5
 Number of Wheels: 4
 Type: automatic
 Motor: Motor(id: "111", name: "Default Motor", model: "T9", numberOfCylinders: 4)
 Airbag Availability: false
*/

builder.shouldHasAirbags = true
// now, the builder creates cars with default configuration,
// but with a small edit on making the airbags available

let safeCar = builder.buildCar()
print(safeCar.description)
/* prints
 color: UIExtendedGrayColorSpace 0 1
 Number of seats: 5
 Number of Wheels: 4
 Type: automatic
 Motor: Motor(id: "111", name: "Default Motor", model: "T9", numberOfCylinders: 4)
 Airbag Availability: true
 */

builder.color = UIColor.purple
// now, the builder creates cars with default configuration
// with some extra features: the airbags are available and the color is purple

let femaleCar = builder.buildCar()
print(femaleCar)
/* prints
 color: UIExtendedSRGBColorSpace 0.5 0 0.5 1
 Number of seats: 5
 Number of Wheels: 4
 Type: automatic
 Motor: Motor(id: "111", name: "Default Motor", model: "T9", numberOfCylinders: 4)
 Airbag Availability: true
*/
```

The **benefit** of applying the Builder Pattern is the ease of creating objects that should contain much of configurations by setting default values, also, the ease of changing these default values.

Take it Further:

As a good practice, all properties that need default values should be in a *separated protocol*, which should be implemented by the class itself and its builder.

Backing to our example, let's create a new protocol called `CarBluePrint`:

```
import UIKit

enum CarType {
    case

    sportage,
    saloon
}

enum GearType {
```

```
    案例

手动,
    自动
}

struct 电机 {
    var 编号: String
    var 名称: String
    var 型号: String
    var 气缸数: UInt8
}

协议 CarBluePrint {
    变量 颜色: UIColor { 获取 设置 }
    变量 座位数: UInt8 { 获取 设置 }
    变量 轮子数: UInt8 { 获取 设置 }
    变量 类型: CarType { 获取 设置 }
    变量 变速箱类型: GearType { 获取 设置 }
    变量 发动机: Motor { 获取 设置 }
    变量 应该有安全气囊: Bool { 获取 设置 }
}

类 Car: CustomStringConvertible, CarBluePrint {
    变量 颜色: UIColor
    var 座位数: UInt8
    var 轮子数: UInt8
    var 类型: 汽车类型
    var 齿轮类型: 齿轮类型
    var 电机: 电机
    var shouldHasAirbags: Bool

    var description: String {
        return "颜色: \(color)座位数: \(numberOfSeats)轮子数:
\(numberOfWheels) 类型: \(gearType)发动机: \(motor)安全气囊是否配备: \(shouldHasAirbags)"}


            init(color: UIColor, numberOfSeats: UInt8, numberOfWheels: UInt8, type: CarType, gearType:
GearType, motor: Motor, shouldHasAirbags: Bool) {

        self.color = color
        self.座位数 = 座位数
        self.轮子数 = 轮子数
        self.类型 = 类型
        self.变速箱类型 = 变速箱类型
        self.发动机 = 发动机
        self.shouldHasAirbags = shouldHasAirbags

    }
}

类 CarBuilder: CarBluePrint {
    变量 颜色: UIColor = UIColor.黑色
    变量 座位数: UInt8 = 5
    var numberOfWheels: UInt8 = 4
    var type: CarType = .saloon
    var gearType: GearType = .automatic
    var motor: Motor = Motor(id: "111", name: "Default Motor",
                             model: "T9", numberOfCylinders: 4)
    var shouldHasAirbags: Bool = false

    func buildCar() -> Car {
```



```
    case

    manual,
    automatic
}

struct Motor {
    var id: String
    var name: String
    var model: String
    var numberOfCylinders: UInt8
}

protocol CarBluePrint {
    var color: UIColor { get set }
    var numberOfSeats: UInt8 { get set }
    var numberOfWheels: UInt8 { get set }
    var type: CarType { get set }
    var gearType: GearType { get set }
    var motor: Motor { get set }
    var shouldHasAirbags: Bool { get set }
}

class Car: CustomStringConvertible, CarBluePrint {
    var color: UIColor
    var numberOfSeats: UInt8
    var numberOfWheels: UInt8
    var type: CarType
    var gearType: GearType
    var motor: Motor
    var shouldHasAirbags: Bool

    var description: String {
        return "color: \(color)\nNumber of seats: \(numberOfSeats)\nNumber of Wheels:
\(numberOfWheels)\n Type: \(gearType)\nMotor: \(motor)\nAirbag Availability: \(shouldHasAirbags)"
    }

    init(color: UIColor, numberOfSeats: UInt8, numberOfWheels: UInt8, type: CarType, gearType:
GearType, motor: Motor, shouldHasAirbags: Bool) {

        self.color = color
        self.numberOfSeats = numberOfSeats
        self.numberOfWheels = numberOfWheels
        self.type = type
        self.gearType = gearType
        self.motor = motor
        self.shouldHasAirbags = shouldHasAirbags

    }
}

class CarBuilder: CarBluePrint {
    var color: UIColor = UIColor.black
    var numberOfSeats: UInt8 = 5
    var numberOfWheels: UInt8 = 4
    var type: CarType = .saloon
    var gearType: GearType = .automatic
    var motor: Motor = Motor(id: "111", name: "Default Motor",
                             model: "T9", numberOfCylinders: 4)
    var shouldHasAirbags: Bool = false

    func buildCar() -> Car {
```

```
        return Car(color: color, numberOfSeats: numberOfSeats, numberOfWheels: numberOfWheels,
type: type, gearType: gearType, motor: motor, shouldHasAirbags: shouldHasAirbags)
    }
}
```

将需要默认值的属性声明到协议中的好处是强制实现任何新增的属性；当一个类遵循协议时，必须声明它的所有属性/方法。

假设需要在创建汽车的蓝图中添加一个名为“电池名称”的新功能：

```
协议 CarBluePrint {
    变量 颜色: UIColor { 获取 设置 }
    变量 座位数: UInt8 { 获取 设置 }
    变量 轮子数: UInt8 { 获取 设置 }
    变量 类型: CarType { 获取 设置 }
    变量 变速箱类型: GearType { 获取 设置 }
    变量 发动机: Motor { 获取 设置 }
    var shouldHasAirbags: Bool { get set }

    // 添加新属性
    var batteryName: String { get set }
}
```

添加新属性后，注意会出现两个编译时错误，提示遵循 `CarBluePrint` 协议需要声明 'batteryName' 属性。这保证了 `CarBuilder` 会声明并为 `batteryName` 属性设置默认值。

在向 CarBluePrint 协议添加 batteryName 新属性后，`Car` 和 `CarBuilder` 类的实现应为：

```
类 Car: CustomStringConvertible, CarBluePrint {
    变量 颜色: UIColor
    var 座位数: UInt8
    var 轮子数: UInt8
    var 类型: 汽车类型
    var 齿轮类型: 齿轮类型
    var 电机: 电机
    var shouldHasAirbags: Bool
    var batteryName: String

    var description: String {
        return "颜色: \(color)座位数: \(numberOfSeats)轮子数:
\(numberOfWheels)类型: \(gearType)电机: \(motor)安全气囊可用性:
\(shouldHasAirbags)电池名称: \(batteryName)"}


            init(color: UIColor, numberOfSeats: UInt8, numberOfWheels: UInt8, type: CarType, gearType:
GearType, motor: Motor, shouldHasAirbags: Bool, batteryName: String) {

        self.color = color
        self.座位数 = 座位数
        self.轮子数 = 轮子数
        self.类型 = 类型
        self.变速箱类型 = 变速箱类型
        self.发动机 = 发动机
        self.shouldHasAirbags = shouldHasAirbags
        self.batteryName = batteryName
    }
```



```
        return Car(color: color, numberOfSeats: numberOfSeats, numberOfWheels: numberOfWheels,
type: type, gearType: gearType, motor: motor, shouldHasAirbags: shouldHasAirbags)
    }
}
```

The benefit of declaring the properties that need default value into a protocol is the forcing to implement any new added property; When a class conforms to a protocol, it has to declare all its properties/methods.

Consider that there is a required new feature that should be added to the blueprint of creating a car called "battery name":

```
protocol CarBluePrint {
    var color: UIColor { get set }
    var numberOfSeats: UInt8 { get set }
    var numberOfWheels: UInt8 { get set }
    var type: CarType { get set }
    var gearType: GearType { get set }
    var motor: Motor { get set }
    var shouldHasAirbags: Bool { get set }

    // adding the new property
    var batteryName: String { get set }
}
```

After adding the new property, note that two compile-time errors will arise, notifying that conforming to `CarBluePrint` protocol requires to declare 'batteryName' property. That guarantees that `CarBuilder` will declare and set a default value for `batteryName` property.

After adding `batteryName` new property to `CarBluePrint` protocol, the implementation of both `Car` and `CarBuilder` classes should be:

```
class Car: CustomStringConvertible, CarBluePrint {
    var color: UIColor
    var numberOfSeats: UInt8
    var numberOfWheels: UInt8
    var type: CarType
    var gearType: GearType
    var motor: Motor
    var shouldHasAirbags: Bool
    var batteryName: String

    var description: String {
        return "color: \(color)\nNumber of seats: \(numberOfSeats)\nNumber of Wheels:
\(numberOfWheels)\nType: \(gearType)\nMotor: \(motor)\nAirbag Availability:
\(shouldHasAirbags)\nBattery Name: \(batteryName)"
    }

    init(color: UIColor, numberOfSeats: UInt8, numberOfWheels: UInt8, type: CarType, gearType:
GearType, motor: Motor, shouldHasAirbags: Bool, batteryName: String) {

        self.color = color
        self.numberOfSeats = numberOfSeats
        self.numberOfWheels = numberOfWheels
        self.type = type
        self.gearType = gearType
        self.motor = motor
        self.shouldHasAirbags = shouldHasAirbags
        self.batteryName = batteryName
    }
```

```
}

class CarBuilder: CarBluePrint {
    var color: UIColor = UIColor.red
    var numberOfSeats: UInt8 = 5
    var numberOfWheels: UInt8 = 4
    var type: CarType = .saloon
    var gearType: GearType = .automatic
    var motor: Motor = Motor(id: "111", name: "Default Motor",
                             model: "T9", numberOfCylinders: 4)
    var shouldHasAirbags: Bool = false
    var batteryName: String = "默认电池名称"

    func buildCar() -> Car {
        return Car(color: color, numberOfSeats: numberOfSeats, numberOfWheels: numberOfWheels,
type: type, gearType: gearType, motor: motor, shouldHasAirbags: shouldHasAirbags, batteryName:
batteryName)
    }
}
```

再次，使用CarBuilder来构建新车：

```
var builder = CarBuilder()

let defaultCar = builder.buildCar()
print(defaultCar)
/* 输出
颜色: UIExtendedSRGBColorSpace 1 0 0 1
 座位数: 5
车轮数: 4
类型: automatic
发动机: Motor(id: "111", name: "默认发动机", model: "T9", numberOfCylinders: 4)
 安全气囊可用性: false
电池名称: 默认电池名称
*/

builder.batteryName = "新电池名称"

let editedBatteryCar = builder.buildCar()
print(editedBatteryCar)
/*
颜色: UIExtendedSRGBColorSpace 1 0 0 1
 座位数: 5
车轮数: 4
类型: automatic
发动机: Motor(id: "111", name: "默认发动机", model: "T9", numberOfCylinders: 4)
 安全气囊可用性: false
电池名称: 新电池名称
 */
```

第52.3节：工厂方法

> 在基于类的编程中，工厂方法模式是一种创建型模式，它使用工厂方法来解决创建对象的问题，而无需指定将要创建的对象的确切类。维基百科参考

```
协议 SenderProtocol
{
```



```
}

class CarBuilder: CarBluePrint {
    var color: UIColor = UIColor.red
    var numberOfSeats: UInt8 = 5
    var numberOfWheels: UInt8 = 4
    var type: CarType = .saloon
    var gearType: GearType = .automatic
    var motor: Motor = Motor(id: "111", name: "Default Motor",
                             model: "T9", numberOfCylinders: 4)
    var shouldHasAirbags: Bool = false
    var batteryName: String = "Default Battery Name"

    func buildCar() -> Car {
        return Car(color: color, numberOfSeats: numberOfSeats, numberOfWheels: numberOfWheels,
type: type, gearType: gearType, motor: motor, shouldHasAirbags: shouldHasAirbags, batteryName:
batteryName)
    }
}
```

Again, let's build new cars by using the `CarBuilder`:

```
var builder = CarBuilder()

let defaultCar = builder.buildCar()
print(defaultCar)
/* prints
 color: UIExtendedSRGBColorSpace 1 0 0 1
 Number of seats: 5
 Number of Wheels: 4
 Type: automatic
 Motor: Motor(id: "111", name: "Default Motor", model: "T9", numberOfCylinders: 4)
 Airbag Availability: false
 Battery Name: Default Battery Name
*/

builder.batteryName = "New Battery Name"

let editedBatteryCar = builder.buildCar()
print(editedBatteryCar)
/*
 color: UIExtendedSRGBColorSpace 1 0 0 1
 Number of seats: 5
 Number of Wheels: 4
 Type: automatic
 Motor: Motor(id: "111", name: "Default Motor", model: "T9", numberOfCylinders: 4)
 Airbag Availability: false
 Battery Name: New Battery Name
 */
```

Section 52.3: Factory Method

> In class-based programming, the factory method pattern is a creational pattern that uses factory methods to deal with the problem of creating objects without having to specify the exact class of the object that will be created. Wikipedia reference

```
protocol SenderProtocol
{
```

```
    函数 send(package: AnyObject)
}

类 Fedex: SenderProtocol
{
    函数 send(package: AnyObject)
    {
        打印("Fedex 递送")
    }
}

类 RegularPriorityMail: SenderProtocol
{
    函数 send(package: AnyObject)
    {
        打印("普通优先邮件递送")
    }
}

// 这是我们的工厂
class DeliverFactory
{
    // 它将负责返回处理任务的合适实例
    static func makeSender(isLate isLate: Bool) -> SenderProtocol
    {
        return isLate ? Fedex() : RegularPriorityMail()
    }
}

// 用法：
let package = ["Item 1", "Item 2"]

// Fedex 类将处理配送
DeliverFactory.makeSender(isLate:true).send(package)

// Regular Priority Mail 类将处理配送
DeliverFactory.makeSender(isLate:false).send(package)
```

通过这样做，我们不依赖于类的具体实现，使得 sender() 对使用者来说完全透明。

在这种情况下，我们只需要知道有一个发送者会处理配送，并且暴露一个名为 send()的方法。还有其他几个优点：减少类之间的耦合，更容易测试，更容易添加新行为而无需更改使用它的代码。

> 在面向对象设计中，接口提供了抽象层，便于对代码进行概念性解释，并创建一个屏障以防止依赖。
> 维基百科参考

第52.4节：观察者模式

> 观察者模式是指一个称为主题的对象维护其依赖对象列表，称为观察者，并在状态发生变化时自动通知它们，通常通过调用它们的某个方法。
> 它主要用于实现分布式事件处理系统。观察者模式也是熟悉的模型-视图-控制器（MVC）架构模式的关键部分。
> 维基百科参考


```
    func send(package: AnyObject)
}

class Fedex: SenderProtocol
{
    func send(package: AnyObject)
    {
        print("Fedex deliver")
    }
}

class RegularPriorityMail: SenderProtocol
{
    func send(package: AnyObject)
    {
        print("Regular Priority Mail deliver")
    }
}

// This is our Factory
class DeliverFactory
{
    // It will be responsable for returning the proper instance that will handle the task
    static func makeSender(isLate isLate: Bool) -> SenderProtocol
    {
        return isLate ? Fedex() : RegularPriorityMail()
    }
}

// Usage:
let package = ["Item 1", "Item 2"]

// Fedex class will handle the delivery
DeliverFactory.makeSender(isLate:true).send(package)

// Regular Priority Mail class will handle the delivery
DeliverFactory.makeSender(isLate:false).send(package)
```

By doing that we don't depend on the real implementation of the class, making the `sender()` completely transparent to who is consuming it.

In this case all we need to know is that a sender will handle the deliver and exposes a method called `send()`. There are several other advantages: reduce classes coupling, easier to test, easier to add new behaviours without having to change who is consuming it.

> Within object-oriented design, interfaces provide layers of abstraction that facilitate conceptual explanation of the code and create a barrier preventing dependencies.Wikipedia reference

Section 52.4: Observer

> The observer pattern is where an object, called the subject, maintains a list of its dependents, called observers, and notifies them automatically of any state changes, usually by calling one of their methods.
> It is mainly used to implement distributed event handling systems. The Observer pattern is also a key part in the familiar model–view–controller (MVC) architectural pattern.Wikipedia reference

基本上，观察者模式用于当你有一个对象可以通知观察者某些行为或状态变化时。

首先，让我们创建一个全局引用（类外部）用于通知中心：

```
let notifCentre = NotificationCenter.default
```

很好，现在我们可以在任何地方调用它。接下来我们想要注册一个类作为观察者.....

```
notifCentre.addObserver(self, selector: #selector(self.myFunc), name: "myNotification", object:
nil)
```

这将该类添加为“readForMyFunc”的观察者。它还表明当收到该通知时应调用函数myFunc。该函数应写在同一个类中：

```
func myFunc(){
    print("已收到通知")
}
```

> 该模式的一个优点是你可以添加多个类作为观察者，从而在一次通知后执行多项操作。

现在几乎可以在代码中的任何地方通过以下代码行简单地发送（或发布，如果你愿意）通知：

```
notifCentre.post(name: "myNotification", object: nil)
```

> 你也可以通过通知传递一个字典形式的信息

```
let myInfo = "pass this on"
notifCentre.post(name: "myNotification", object: ["moreInfo":myInfo])
```

但你需要在你的函数中添加一个通知处理：

```
func myFunc(_ notification: Notification){
    let userInfo = (notification as NSNotification).userInfo as! [String: AnyObject]
    let passedInfo = userInfo["moreInfo"]
    print("通知 \(moreInfo) 已被接收")
    //打印 - 通知 pass this on 已被接收
}
```

第52.5节：责任链模式

> 在面向对象设计中，责任链模式是一种设计模式，由命令对象的源头和一系列处理对象组成。每个处理对象包含定义它能处理的命令对象类型的逻辑；其余的命令对象会被传递给链中的下一个处理对象。该链还支持在末端添加新的处理对象。 维基百科

设置组成责任链的类。

首先我们为所有的处理对象创建一个接口。

Basically the observer pattern is used when you have an object which can notify observers of certain behaviors or state changes.

First lets create a global reference (outside of a class) for the Notification Centre :

```
let notifCentre = NotificationCenter.default
```

Great now we can call this from anywhere. We would then want to register a class as an observer...

```
notifCentre.addObserver(self, selector: #selector(self.myFunc), name: "myNotification", object:
nil)
```

This adds the class as an observer for "readForMyFunc". It also indicates that the function myFunc should be called when that notification is received. This function should be written in the same class:

```
func myFunc(){
    print("The notification has been received")
}
```

> One of the advantages to this pattern is that you can add many classes as observers and thus perform many actions after one notification.

The notification can now simply be sent(or posted if you prefer) from almost anywhere in the code with the line:

```
notifCentre.post(name: "myNotification", object: nil)
```

> You can also pass information with the notification as a Dictionary

```
let myInfo = "pass this on"
notifCentre.post(name: "myNotification", object: ["moreInfo":myInfo])
```

But then you need to add a notification to your function:

```
func myFunc(_ notification: Notification){
    let userInfo = (notification as NSNotification).userInfo as! [String: AnyObject]
    let passedInfo = userInfo["moreInfo"]
    print("The notification \(moreInfo) has been received")
    //prints - The notification pass this on has been received
}
```

Section 52.5: Chain of responsibility

> In object-oriented design, the chain-of-responsibility pattern is a design pattern consisting of a source of `command` objects and a series of `processing` objects. Each `processing` object contains logic that defines the types of command objects that it can handle; the rest are passed to the next `processing` object in the chain. A mechanism also exists for adding new `processing` objects to the end of this chain. Wikipedia

Setting up the classes that made up the chain of responsibility.

First we create an interface for all the `processing` objects.

```
protocol PurchasePower {
var allowable : Float { get }
  var role : String { get }
  var successor : PurchasePower? { get set }
}

extension PurchasePower {
  func process(request : PurchaseRequest){
    if request.amount < self.allowable {
      print(self.role + " 将批准 $ \(request.amount) 用于 \(request.purpose)")
    } else if successor != nil {
successor?.process(request: request)
    }
  }
}
```

然后我们创建命令对象。

```
结构体 PurchaseRequest {
  变量 amount : 浮点数
  变量 purpose : 字符串
}
```

最后，创建组成责任链的对象。

```
class ManagerPower : PurchasePower {
  var allowable: Float = 20
  var role : String = "经理"
  var successor: PurchasePower?
}

class DirectorPower : PurchasePower {
  var allowable: Float = 100
  var role = "主管"
  var successor: PurchasePower?
}

class PresidentPower : PurchasePower {
  var allowable: Float = 5000
  var role = "总裁"
  var successor: PurchasePower?
}
```

初始化并将它们链接起来：

```
let manager = ManagerPower()
let director = DirectorPower()
let president = PresidentPower()

manager.successor = director
director.successor = president
```

> 这里用于链接对象的机制是属性访问

创建运行请求：

```
manager.process(request: PurchaseRequest(amount: 2, purpose: "买一支笔"))  // 经理将会
```



```
protocol PurchasePower {
var allowable : Float { get }
  var role : String { get }
  var successor : PurchasePower? { get set }
}

extension PurchasePower {
  func process(request : PurchaseRequest){
    if request.amount < self.allowable {
      print(self.role + " will approve $ \(request.amount) for \(request.purpose)")
    } else if successor != nil {
      successor?.process(request: request)
    }
  }
}
```

Then we create the command object.

```
struct PurchaseRequest {
  var amount : Float
  var purpose : String
}
```

Finally, creating objects that made up the chain of responsibility.

```
class ManagerPower : PurchasePower {
  var allowable: Float = 20
  var role : String = "Manager"
  var successor: PurchasePower?
}

class DirectorPower : PurchasePower {
  var allowable: Float = 100
  var role = "Director"
  var successor: PurchasePower?
}

class PresidentPower : PurchasePower {
  var allowable: Float = 5000
  var role = "President"
  var successor: PurchasePower?
}
```

Initiate and chaining it together :

```
let manager = ManagerPower()
let director = DirectorPower()
let president = PresidentPower()

manager.successor = director
director.successor = president
```

> The mechanism for chaining up objects here is property access

Creating request to run it :

```
manager.process(request: PurchaseRequest(amount: 2, purpose: "buying a pen"))  // Manager will
```

```
批准 2.0 美元购买一支笔
manager.process(request: PurchaseRequest(amount: 90, purpose: "买一台打印机")) // 主管将会
批准 90.0 美元购买一台打印机

manager.process(request: PurchaseRequest(amount: 2000, purpose: "投资股票")) // 总裁将会
批准 2000.0 美元投资股票
```

第52.6节：迭代器

> 在计算机编程中，迭代器是一个对象，使程序员能够遍历容器，
> 特别是列表。 维基百科

```
struct Turtle {
  let name: String
}

struct Turtles {
  let turtles: [Turtle]
}

struct TurtlesIterator: IteratorProtocol {
  private var current = 0
  private let turtles: [Turtle]

  init(turtles: [Turtle]) {
    self.turtles = turtles
  }

  mutating func next() -> Turtle? {
    defer { current += 1 }
    return turtles.count > current ? turtles[current] : nil
  }
}

extension Turtles: Sequence {
  func makeIterator() -> TurtlesIterator {
    return TurtlesIterator(turtles: turtles)
  }
}
```

使用示例为

```
let ninjaTurtles = Turtles(turtles: [Turtle(name: "Leo"),
                                     Turtle(name: "Mickey"),
                                     Turtle(name: "Raph"),
                                     Turtle(name: "Doney")])
print("Splinter and")
for turtle in ninjaTurtles {
  print("伟大的：\(turtle)")
}
```



```
approve $ 2.0 for buying a pen
manager.process(request: PurchaseRequest(amount: 90, purpose: "buying a printer")) // Director will
approve $ 90.0 for buying a printer

manager.process(request: PurchaseRequest(amount: 2000, purpose: "invest in stock")) // President
will approve $ 2000.0 for invest in stock
```

Section 52.6: Iterator

> In computer programming an iterator is an object that enables a programmer to traverse a container,
> particularly lists. Wikipedia

```
struct Turtle {
  let name: String
}

struct Turtles {
  let turtles: [Turtle]
}

struct TurtlesIterator: IteratorProtocol {
  private var current = 0
  private let turtles: [Turtle]

  init(turtles: [Turtle]) {
    self.turtles = turtles
  }

  mutating func next() -> Turtle? {
    defer { current += 1 }
    return turtles.count > current ? turtles[current] : nil
  }
}

extension Turtles: Sequence {
  func makeIterator() -> TurtlesIterator {
    return TurtlesIterator(turtles: turtles)
  }
}
```

And usage example would be

```
let ninjaTurtles = Turtles(turtles: [Turtle(name: "Leo"),
                                     Turtle(name: "Mickey"),
                                     Turtle(name: "Raph"),
                                     Turtle(name: "Doney")])
print("Splinter and")
for turtle in ninjaTurtles {
  print("The great: \(turtle)")
}
```

第53章：设计模式 - 结构型

设计模式是软件开发中经常出现问题的通用解决方案。以下是结构化和设计代码的标准化最佳实践模板，以及这些设计模式适用的常见场景示例。

结构型设计模式 关注类和对象的组合，以创建接口并实现更强大的功能。

第53.1节：适配器

适配器 用于将给定类的接口，称为适配者（Adaptee），转换为另一种接口，称为目标（Target）。对目标的操作由客户端（Client）调用，这些操作由适配器进行适配并传递给适配者。

在 Swift 中，适配器通常可以通过协议来实现。在以下示例中，能够与目标（Target）通信的客户端通过使用适配器，获得了执行适配者（Adaptee）类功能的能力。

```
// 客户端无法直接访问的功能
class 适配者 {
    func foo() {
        // …
    }
}

// 客户端可以直接访问的目标功能
protocol 目标功能 {
    func fooBar() {}
}

// 适配器用于将对目标的请求传递给适配者的请求
extension 适配者: 目标功能 {
    func fooBar() {
       foo()
    }
}
```

单向适配器的示例流程：客户端 -> 目标 -> 适配器 -> 适配者适配器也可以是双向的，这被

称为双向适配器。当两个不同的客户端需要以不同方式查看同一个对象时，双向适配器可能会很有用。

第53.2节：外观模式

外观（Facade）为子系统接口提供了统一的高级接口。这允许更简单、更安全地访问子系统的更通用功能。

以下是一个用于在 UserDefaults 中设置和检索对象的外观模式示例。

```
枚举 Defaults {

    静态函数 set(_ object: Any, forKey defaultName: String) {
        让 defaults: UserDefaults = UserDefaults.standard
defaults.set(object, forKey:defaultName)
        defaults.synchronize()
```


Chapter 53: Design Patterns - Structural

Design patterns are general solutions to problems that frequently occur in software development. The following are templates of standardized best practices in structuring and designing code, as well as examples of common contexts in which these design patterns would be appropriate.

Structural design patterns focus on the composition of classes and objects to create interfaces and achieve greater functionality.

Section 53.1: Adapter

Adapters are used to convert the interface of a given class, known as an **Adaptee**, into another interface, called the **Target**. Operations on the Target are called by a **Client**, and those operations are *adapted* by the Adapter and passed on to the Adaptee.

In Swift, Adapters can often be formed through the use of protocols. In the following example, a Client able to communicate with the Target is provided with the ability to perform functions of the Adaptee class through the use of an adapter.

```
// The functionality to which a Client has no direct access
class Adaptee {
    func foo() {
        // ...
    }
}

// Target's functionality, to which a Client does have direct access
protocol TargetFunctionality {
    func fooBar() {}
}

// Adapter used to pass the request on the Target to a request on the Adaptee
extension Adaptee: TargetFunctionality {
    func fooBar() {
        foo()
    }
}
```

Example flow of a one-way adapter: `Client -> Target -> Adapter -> Adaptee`

Adapters can also be bi-directional, and these are known as **two-way adapters**. A two-way adapter may be useful when two different clients need to view an object differently.

Section 53.2: Facade

A **Facade** provides a unified, high-level interface to subsystem interfaces. This allows for simpler, safer access to the more general facilities of a subsystem.

The following is an example of a Facade used to set and retrieve objects in UserDefaults.

```
enum Defaults {

    static func set(_ object: Any, forKey defaultName: String) {
        let defaults: UserDefaults = UserDefaults.standard
        defaults.set(object, forKey:defaultName)
        defaults.synchronize()
```

```
    }

    静态函数 object(forKey key: String) -> AnyObject! {
        让 defaults: UserDefaults = UserDefaults.standard
        返回 defaults.object(forKey: key) 作为 AnyObject!
    }

}
```

用法可能如下所示。

```
Defaults.set("Beyond all recognition.", forKey:"fooBar")
Defaults.object(forKey: "fooBar")
```

访问共享实例和同步 UserDefaults 的复杂性对客户端是隐藏的，
并且该接口可以从程序的任何地方访问。


```
    }

    static func object(forKey key: String) -> AnyObject! {
        let defaults: UserDefaults = UserDefaults.standard
        return defaults.object(forKey: key) as AnyObject!
    }

}
```

Usage might look like the following.

```
Defaults.set("Beyond all recognition.", forKey:"fooBar")
Defaults.object(forKey: "fooBar")
```

The complexities of accessing the shared instance and synchronizing UserDefaults are hidden from the client, and this interface can be accessed from anywhere in the program.

第54章：（不安全的）缓冲区指针

“缓冲区指针用于对内存区域的低级访问。例如，你可以使用缓冲区指针来高效处理和在应用程序与服务之间传递数据。”

摘自：苹果公司“使用 Swift 与 Cocoa 和 Objective-C（Swift 3.1 版）”iBooks。
https://itun.es/us/utTW7.l

您负责通过缓冲指针管理所操作内存的生命周期，以避免内存泄漏
或未定义行为。

第54.1节：UnsafeMutablePointer

```
结构体 UnsafeMutablePointer<Pointee>
```

> 用于访问和操作特定类型数据的指针。

您使用 UnsafeMutablePointer 类型的实例来访问内存中特定类型的数据。指针可以访问的数据类型是指针的 Pointee 类型。UnsafeMutablePointer 不提供自动内存管理或对齐保证。您负责通过不安全指针管理所操作内存的生命周期，以避免内存泄漏或未定义行为。

您手动管理的内存可以是未绑定类型的，也可以绑定到特定类型。您使用 UnsafeMutablePointer 类型来访问和管理已绑定到特定类型的内存。（来源）

```
import Foundation

let arr = [1,5,7,8]

let pointer = UnsafeMutablePointer<[Int]>.allocate(capacity: 4)
pointer.initialize(to: arr)

let x = pointer.pointee[3]

print(x)

指针。反初始化()
指针。释放(容量：4)

类 A {
  变量 x: 字符串?

  便利初始化 (_ x: 字符串) {
    self.初始化()
    self.x = x
  }

  函数 描述() -> 字符串 {
    返回 x ?? ""
  }
}


让 arr2 = [A("OK"), A("OK 2")]
让 指针2 = UnsafeMutablePointer<[A]>.分配(容量：2)
```


Chapter 54: (Unsafe) Buffer Pointers

“A buffer pointer is used for low-level access to a region of memory. For example, you can use a buffer pointer for efficent processing and communication of data between apps and services.”

Excerpt From: Apple Inc. “Using Swift with Cocoa and Objective-C (Swift 3.1 Edition).” iBooks.
https://itun.es/us/utTW7.l

You are responsible for handling the life cycle of any memory you work with through buffer pointers, to avoid leaks or undefined behavior.

Section 54.1: UnsafeMutablePointer

```
struct UnsafeMutablePointer<Pointee>
```

> A pointer for accessing and manipulating data of a specific type.

You use instances of the UnsafeMutablePointer type to access data of a specific type in memory. The type of data that a pointer can access is the pointer's Pointee type. UnsafeMutablePointer provides no automated memory management or alignment guarantees. You are responsible for handling the life cycle of any memory you work with through unsafe pointers to avoid leaks or undefined behavior.

Memory that you manually manage can be either untyped or bound to a specific type. You use the UnsafeMutablePointer type to access and manage memory that has been bound to a specific type. (Source)

```
import Foundation

let arr = [1,5,7,8]

let pointer = UnsafeMutablePointer<[Int]>.allocate(capacity: 4)
pointer.initialize(to: arr)

let x = pointer.pointee[3]

print(x)

pointer.deinitialize()
pointer.deallocate(capacity: 4)

class A {
  var x: String?

  convenience init (_ x: String) {
    self.init()
    self.x = x
  }

  func description() -> String {
    return x ?? ""
  }
}


let arr2 = [A("OK"), A("OK 2")]
let pointer2 = UnsafeMutablePointer<[A]>.allocate(capacity: 2)
```

```
指针2.初始化(为：arr2)

指针2.指向的值
让 y = 指针2.指向的值[1]

打印(y)

指针2.反初始化()
指针2.释放(容量：2)
```

转换为 Swift 3.0，来源于原始源代码_____

第54.2节：缓冲指针的实际用例

解析 Swift 库方法中不安全指针的使用；

```
public init?(validatingUTF8 cString: UnsafePointer<CChar>)
```

目的：

通过复制并验证给定指针引用的以空字符结尾的 UTF-8 数据来创建一个新的字符串。

此初始化器不会尝试修复格式错误的 UTF-8 代码单元序列。如果发现任何错误，初始化器的结果为nil。以下示例使用指向两个不同CChar数组内容的指针调用此初始化器——第一个数组包含格式正确的 UTF-8 代码单元序列，第二个数组末尾包含格式错误的序列。

> **来源，苹果公司，Swift 3 头文件**（访问头文件：在 Playground 中，按 Cmd+点击 Swift 词语）
> 在以下代码行：

```
import Swift
```

```
let validUTF8: [CChar] = [67, 97, 102, -61, -87, 0]
     validUTF8.withUnsafeBufferPointer { ptr in
         let s = String(validatingUTF8: ptr.baseAddress!)
         print(s as Any)
     }
     // 打印 "Optional(Café)"

     let invalidUTF8: [CChar] = [67, 97, 102, -61, 0]
     invalidUTF8.withUnsafeBufferPointer { ptr in
      let s = String(validatingUTF8: ptr.baseAddress!)
      print(s as Any)
     }
// 输出 "nil"
```

> (来源，苹果公司，Swift 头文件示例)


```
pointer2.initialize(to: arr2)

pointer2.pointee
let y = pointer2.pointee[1]

print(y)

pointer2.deinitialize()
pointer2.deallocate(capacity: 2)
```

Converted to Swift 3.0 from original source

Section 54.2: Practical Use-Case for Buffer Pointers

Deconstructing the use of an unsafe pointer in the Swift library method;

```
public init?(validatingUTF8 cString: UnsafePointer<CChar>)
```

Purpose:

Creates a new string by copying and validating the null-terminated UTF-8 data referenced by the given pointer.

This initializer does not try to repair ill-formed UTF-8 code unit sequences. If any are found, the result of the initializer is `nil`. The following example calls this initializer with pointers to the contents of two different `CChar` arrays---the first with well-formed UTF-8 code unit sequences and the second with an ill-formed sequence at the end.

> **Source**, **Apple Inc., Swift 3 header file** (For header access: In Playground, Cmd+Click on the word Swift) in the line of code:

```
import Swift
```

```
let validUTF8: [CChar] = [67, 97, 102, -61, -87, 0]
     validUTF8.withUnsafeBufferPointer { ptr in
         let s = String(validatingUTF8: ptr.baseAddress!)
         print(s as Any)
     }
     // Prints "Optional(Café)"

     let invalidUTF8: [CChar] = [67, 97, 102, -61, 0]
     invalidUTF8.withUnsafeBufferPointer { ptr in
      let s = String(validatingUTF8: ptr.baseAddress!)
      print(s as Any)
     }
// Prints "nil"
```

> (Source, Apple Inc., Swift Header File Example)

第55章：加密哈希

第55.1节：使用 MD5、SHA1、SHA224、SHA256、SHA384、SHA512 的 HMAC（Swift 3）

> 这些函数将使用八种加密哈希算法之一对字符串或数据输入进行哈希。

name 参数指定哈希函数名称，类型为字符串。支持的函数有 MD5、SHA1、SHA224、SHA256、SHA384 和 SHA512

此示例需要 Common Crypto
项目中必须有桥接头文件：
#import <CommonCrypto/CommonCrypto.h>
将 Security.framework 添加到项目中。

这些函数接受一个哈希名称、要哈希的消息、一个密钥并返回摘要：

hashName：哈希函数名称，类型为字符串 message：消息，类型为Data key：密钥，类型为Data 返回值：摘要，类型为Data

```
func hmac(hashName:String, message:Data, key:Data) -> Data? {
    let algos = ["SHA1":   (kCCHmacAlgSHA1,   CC_SHA1_DIGEST_LENGTH),
                 "MD5":    (kCCHmacAlgMD5,    CC_MD5_DIGEST_LENGTH),
                 "SHA224": (kCCHmacAlgSHA224, CC_SHA224_DIGEST_LENGTH),
                 "SHA256": (kCCHmacAlgSHA256, CC_SHA256_DIGEST_LENGTH),
                 "SHA384": (kCCHmacAlgSHA384, CC_SHA384_DIGEST_LENGTH),
                 "SHA512": (kCCHmacAlgSHA512, CC_SHA512_DIGEST_LENGTH)]
    guard let (hashAlgorithm, length) = algos[hashName]  else { return nil }
    var macData = Data(count: Int(length))

macData.withUnsafeMutableBytes {macBytes in
        message.withUnsafeBytes {messageBytes in
            key.withUnsafeBytes {keyBytes in
              CCHmac(CCHmacAlgorithm(hashAlgorithm),
                        keyBytes,      key.count,
messageBytes, message.count,
                     macBytes)
            }
        }
    }
    return macData
}
```

hashName：哈希函数名称，类型为字符串 message：消息，类型为字符串 key：密钥，类型为字符串 返回值：摘要，类型为Data

```
func hmac(hashName:String, message:String, key:String) -> Data? {
    let messageData = message.data(using:.utf8)!
    let keyData = key.data(using:.utf8)!
    return hmac(hashName:hashName, message:messageData, key:keyData)
}
```

hashName：哈希函数的名称，类型为字符串 message：消息，类型为字符串 key：密钥，类型为数据 returns：摘要，类型为数据

```
func hmac(hashName:String, message:String, key:Data) -> Data? {
    let messageData = message.data(using:.utf8)!
    return hmac(hashName:hashName, message:messageData, key:key)
}
```


Chapter 55: Cryptographic Hashing

Section 55.1: HMAC with MD5, SHA1, SHA224, SHA256, SHA384, SHA512 (Swift 3)

> These functions will hash either String or Data input with one of eight cryptographic hash algorithms.

The name parameter specifies the hash function name as a String Supported functions are MD5, SHA1, SHA224, SHA256, SHA384 and SHA512

This example requires Common Crypto
It is necessary to have a bridging header to the project:
`#import <CommonCrypto/CommonCrypto.h>`
Add the Security.framework to the project.

These functions takes a hash name, message to be hashed, a key and return a digest:

hashName: name of a hash function as String message: message as Data key: key as Data returns: digest as Data

```
func hmac(hashName:String, message:Data, key:Data) -> Data? {
    let algos = ["SHA1":   (kCCHmacAlgSHA1,   CC_SHA1_DIGEST_LENGTH),
                 "MD5":    (kCCHmacAlgMD5,    CC_MD5_DIGEST_LENGTH),
                 "SHA224": (kCCHmacAlgSHA224, CC_SHA224_DIGEST_LENGTH),
                 "SHA256": (kCCHmacAlgSHA256, CC_SHA256_DIGEST_LENGTH),
                 "SHA384": (kCCHmacAlgSHA384, CC_SHA384_DIGEST_LENGTH),
                 "SHA512": (kCCHmacAlgSHA512, CC_SHA512_DIGEST_LENGTH)]
    guard let (hashAlgorithm, length) = algos[hashName]  else { return nil }
    var macData = Data(count: Int(length))

    macData.withUnsafeMutableBytes {macBytes in
        message.withUnsafeBytes {messageBytes in
            key.withUnsafeBytes {keyBytes in
                CCHmac(CCHmacAlgorithm(hashAlgorithm),
                       keyBytes,     key.count,
                       messageBytes, message.count,
                       macBytes)
            }
        }
    }
    return macData
}
```

hashName: name of a hash function as String message: message as String key: key as String returns: digest as Data

```
func hmac(hashName:String, message:String, key:String) -> Data? {
    let messageData = message.data(using:.utf8)!
    let keyData = key.data(using:.utf8)!
    return hmac(hashName:hashName, message:messageData, key:keyData)
}
```

hashName: name of a hash function as String message: message as String key: key as Data returns: digest as Data

```
func hmac(hashName:String, message:String, key:Data) -> Data? {
    let messageData = message.data(using:.utf8)!
    return hmac(hashName:hashName, message:messageData, key:key)
}
```

// 示例

```
let clearString = "clearData0123456"
let keyString   = "keyData8901234562"
let clearData   = clearString.data(using:.utf8)!
let keyData     = keyString.data(using:.utf8)!
print("clearString: \(clearString)")
print("keyString:   \(keyString)")
print("clearData: \(clearData as NSData)")
print("keyData:   \(keyData as NSData)")

let hmacData1 = hmac(hashName:"SHA1", message:clearData, key:keyData)
print("hmacData1: \(hmacData1! as NSData)")

let hmacData2 = hmac(hashName:"SHA1", message:clearString, key:keyString)
print("hmacData2: \(hmacData2! as NSData)")

let hmacData3 = hmac(hashName:"SHA1", message:clearString, key:keyData)
print("hmacData3: \(hmacData3! as NSData)")
```

输出：

```
clearString: clearData0123456
keyString:   keyData8901234562
clearData: <636c6561 72446174 61303132 33343536>
keyData:   <6b657944 61746138 39303132 33343536 32>

hmacData1:
hmacData2:
hmacData3:
```

第55.2节：MD2、MD4、MD5、SHA1、SHA224、SHA256、SHA384、SHA512（Swift 3）

> 这些函数将使用八种加密哈希算法之一对字符串或数据输入进行哈希。

name参数指定哈希函数名称，类型为字符串
支持的函数有MD2、MD4、MD5、SHA1、SHA224、SHA256、SHA384和SHA512

此示例需要Common Crypto
项目中需要有一个桥接头文件：
#import <CommonCrypto/CommonCrypto.h>
将 Security.framework 添加到项目中。

此函数接受哈希名称和要哈希的数据，返回一个Data类型的结果：

name：哈希函数名称，类型为字符串 data：要哈希的数据 返回值：哈希结果，类型为Data

```
func hash(name:String, data:Data) -> Data? {
    let algos = ["MD2":    (CC_MD2,    CC_MD2_DIGEST_LENGTH),
                 "MD4":    (CC_MD4,    CC_MD4_DIGEST_LENGTH),
                 "MD5":    (CC_MD5,    CC_MD5_DIGEST_LENGTH),
                 "SHA1":   (CC_SHA1,   CC_SHA1_DIGEST_LENGTH),
                 "SHA224": (CC_SHA224, CC_SHA224_DIGEST_LENGTH),
                 "SHA256": (CC_SHA256, CC_SHA256_DIGEST_LENGTH),
                 "SHA384": (CC_SHA384, CC_SHA384_DIGEST_LENGTH),
```


// Examples

```
let clearString = "clearData0123456"
let keyString   = "keyData8901234562"
let clearData   = clearString.data(using:.utf8)!
let keyData     = keyString.data(using:.utf8)!
print("clearString: \(clearString)")
print("keyString:   \(keyString)")
print("clearData: \(clearData as NSData)")
print("keyData:   \(keyData as NSData)")

let hmacData1 = hmac(hashName:"SHA1", message:clearData, key:keyData)
print("hmacData1: \(hmacData1! as NSData)")

let hmacData2 = hmac(hashName:"SHA1", message:clearString, key:keyString)
print("hmacData2: \(hmacData2! as NSData)")

let hmacData3 = hmac(hashName:"SHA1", message:clearString, key:keyData)
print("hmacData3: \(hmacData3! as NSData)")
```

Output:

```
clearString: clearData0123456
keyString:   keyData8901234562
clearData: <636c6561 72446174 61303132 33343536>
keyData:   <6b657944 61746138 39303132 33343536 32>

hmacData1:
hmacData2:
hmacData3:
```

Section 55.2: MD2, MD4, MD5, SHA1, SHA224, SHA256, SHA384, SHA512 (Swift 3)

> These functions will hash either String or Data input with one of eight cryptographic hash algorithms.

The name parameter specifies the hash function name as a String
Supported functions are MD2, MD4, MD5, SHA1, SHA224, SHA256, SHA384 and SHA512

This example requires Common Crypto
It is necessary to have a bridging header to the project:
#import <CommonCrypto/CommonCrypto.h>
Add the Security.framework to the project.

This function takes a hash name and Data to be hashed and returns a Data:

name: A name of a hash function as a String data: The Data to be hashed returns: the hashed result as Data

```
func hash(name:String, data:Data) -> Data? {
    let algos = ["MD2":    (CC_MD2,    CC_MD2_DIGEST_LENGTH),
                 "MD4":    (CC_MD4,    CC_MD4_DIGEST_LENGTH),
                 "MD5":    (CC_MD5,    CC_MD5_DIGEST_LENGTH),
                 "SHA1":   (CC_SHA1,   CC_SHA1_DIGEST_LENGTH),
                 "SHA224": (CC_SHA224, CC_SHA224_DIGEST_LENGTH),
                 "SHA256": (CC_SHA256, CC_SHA256_DIGEST_LENGTH),
                 "SHA384": (CC_SHA384, CC_SHA384_DIGEST_LENGTH),
```

```
                "SHA512": (CC_SHA512, CC_SHA512_DIGEST_LENGTH)]
    guard let (hashAlgorithm, length) = algos[name]  else { return nil }
    var hashData = Data(count: Int(length))

_ = hashData.withUnsafeMutableBytes {digestBytes in
        data.withUnsafeBytes {messageBytes in
hashAlgorithm(messageBytes, CC_LONG(data.count), digestBytes)
        }
    }
    return hashData
}
```

此函数接受一个哈希名称和要哈希的字符串，并返回一个Data：

name：哈希函数的名称，类型为String string：要哈希的字符串 返回值：哈希结果，类型为Data

```
func hash(name:String, string:String) -> Data? {
    let data = string.data(using:.utf8)!
    return hash(name:name, data:data)
}
```

示例：

```
let clearString = "clearData0123456"
let clearData   = clearString.data(using:.utf8)!
print("clearString: \(clearString)")
print("clearData: \(clearData as NSData)")

let hashSHA256 = hash(name:"SHA256", string:clearString)
print("hashSHA256: \(hashSHA256! as NSData)")

let hashMD5 = hash(name:"MD5", data:clearData)
print("hashMD5: \(hashMD5! as NSData)")
```

输出：

```
clearString: clearData0123456
clearData: <636c6561 72446174 61303132 33343536>

hashSHA256: <aabc766b 6b357564 e41f4f91 2d494bcc bfa16924 b574abbd ba9e3e9d a0c8920a>
hashMD5: <4df665f7 b94aea69 695b0e7b baf9e9d6>
```



```
                "SHA512": (CC_SHA512, CC_SHA512_DIGEST_LENGTH)]
    guard let (hashAlgorithm, length) = algos[name]  else { return nil }
    var hashData = Data(count: Int(length))

    _ = hashData.withUnsafeMutableBytes {digestBytes in
        data.withUnsafeBytes {messageBytes in
            hashAlgorithm(messageBytes, CC_LONG(data.count), digestBytes)
        }
    }
    return hashData
}
```

This function takes a hash name and String to be hashed and returns a Data:

name: A name of a hash function as a String string: The String to be hashed returns: the hashed result as Data

```
func hash(name:String, string:String) -> Data? {
    let data = string.data(using:.utf8)!
    return hash(name:name, data:data)
}
```

Examples:

```
let clearString = "clearData0123456"
let clearData   = clearString.data(using:.utf8)!
print("clearString: \(clearString)")
print("clearData: \(clearData as NSData)")

let hashSHA256 = hash(name:"SHA256", string:clearString)
print("hashSHA256: \(hashSHA256! as NSData)")

let hashMD5 = hash(name:"MD5", data:clearData)
print("hashMD5: \(hashMD5! as NSData)")
```

Output:

```
clearString: clearData0123456
clearData: <636c6561 72446174 61303132 33343536>

hashSHA256: <aabc766b 6b357564 e41f4f91 2d494bcc bfa16924 b574abbd ba9e3e9d a0c8920a>
hashMD5: <4df665f7 b94aea69 695b0e7b baf9e9d6>
```

第56章：AES加密

第56.1节：使用随机IV的CBC模式AES加密（Swift 3.0）

IV被加在加密数据的前面

aesCBC128Encrypt 会生成一个随机IV并加在加密代码的前面。
aesCBC128Decrypt 会在解密时使用加在前面的IV。

输入的数据和密钥都是Data对象。如果需要Base64等编码形式，请在调用方法中进行转换。

密钥长度应为128位（16字节）、192位（24字节）或256位（32字节）。如果使用其他密钥长度，将抛出错误。

默认使用PKCS#7填充。

此示例需要 Common Crypto
项目中必须有桥接头文件：
#import <CommonCrypto/CommonCrypto.h>
将Security.framework添加到项目中。

这是示例代码，不是生产代码。

```
enum AESError: 错误 {
    case KeyError((字符串, 整数))
    case IVError((字符串, 整数))
    case CryptorError((字符串, 整数))
}

// iv 被加密数据前缀
func aesCBCEncrypt(data:Data, keyData:Data) throws -> Data {
    let keyLength = keyData.count
    let validKeyLengths = [kCCKeySizeAES128, kCCKeySizeAES192, kCCKeySizeAES256]
    if (validKeyLengths.contains(keyLength) == false) {
throw AESError.KeyError(("无效的密钥长度", keyLength))
    }

    let ivSize = kCCBlockSizeAES128;
    let cryptLength = size_t(ivSize + data.count + kCCBlockSizeAES128)
    var cryptData = Data(count:cryptLength)

    let status = cryptData.withUnsafeMutableBytes {ivBytes in
       SecRandomCopyBytes(kSecRandomDefault, kCCBlockSizeAES128, ivBytes)
    }
    if (status != 0) {
throw AESError.IVError(("IV 生成失败", Int(status)))
    }

    var numBytesEncrypted :size_t = 0
    let options   = CCOptions(kCCOptionPKCS7Padding)

    let cryptStatus = cryptData.withUnsafeMutableBytes {cryptBytes in
        data.withUnsafeBytes {dataBytes in
            keyData.withUnsafeBytes {keyBytes in
              CCCrypt(CCOperation(kCCEncrypt),
                      CCAlgorithm(kCCAlgorithmAES),
```


Chapter 56: AES encryption

Section 56.1: AES encryption in CBC mode with a random IV (Swift 3.0)

The iv is prefixed to the encrypted data

`aesCBC128Encrypt` will create a random IV and prefixed to the encrypted code.
`aesCBC128Decrypt` will use the prefixed IV during decryption.

Inputs are the data and key are Data objects. If an encoded form such as Base64 if required convert to and/or from in the calling method.

The key should be exactly 128-bits (16-bytes), 192-bits (24-bytes) or 256-bits (32-bytes) in length. If another key size is used an error will be thrown.

PKCS#7 padding is set by default.

This example requires Common Crypto
It is necessary to have a bridging header to the project:
`#import <CommonCrypto/CommonCrypto.h>`
Add the `Security.framework` to the project.

This is example, not production code.

```
enum AESError: Error {
    case KeyError((String, Int))
    case IVError((String, Int))
    case CryptorError((String, Int))
}

// The iv is prefixed to the encrypted data
func aesCBCEncrypt(data:Data, keyData:Data) throws -> Data {
    let keyLength = keyData.count
    let validKeyLengths = [kCCKeySizeAES128, kCCKeySizeAES192, kCCKeySizeAES256]
    if (validKeyLengths.contains(keyLength) == false) {
        throw AESError.KeyError(("Invalid key length", keyLength))
    }

    let ivSize = kCCBlockSizeAES128;
    let cryptLength = size_t(ivSize + data.count + kCCBlockSizeAES128)
    var cryptData = Data(count:cryptLength)

    let status = cryptData.withUnsafeMutableBytes {ivBytes in
        SecRandomCopyBytes(kSecRandomDefault, kCCBlockSizeAES128, ivBytes)
    }
    if (status != 0) {
        throw AESError.IVError(("IV generation failed", Int(status)))
    }

    var numBytesEncrypted :size_t = 0
    let options   = CCOptions(kCCOptionPKCS7Padding)

    let cryptStatus = cryptData.withUnsafeMutableBytes {cryptBytes in
        data.withUnsafeBytes {dataBytes in
            keyData.withUnsafeBytes {keyBytes in
                CCCrypt(CCOperation(kCCEncrypt),
                        CCAlgorithm(kCCAlgorithmAES),
```

```
options,
                        keyBytes, keyLength,
                        cryptBytes,
dataBytes, data.count,
                        cryptBytes+kCCBlockSizeAES128, cryptLength,
                            &numBytesEncrypted)
                }
            }
    }

    if UInt32(cryptStatus) == UInt32(kCCSuccess) {
        cryptData.count = numBytesEncrypted + ivSize
    }
    else {
throw AESError.CryptorError(("加密失败", Int(cryptStatus)))
    }

    return cryptData;
}

// iv 被加在加密数据之前
func aesCBCDecrypt(data:Data, keyData:Data) throws -> Data? {
    let keyLength = keyData.count
    let validKeyLengths = [kCCKeySizeAES128, kCCKeySizeAES192, kCCKeySizeAES256]
    if (validKeyLengths.contains(keyLength) == false) {
throw AESError.KeyError(("无效的密钥长度", keyLength))
    }

    let ivSize = kCCBlockSizeAES128;
    let clearLength = size_t(data.count - ivSize)
    var clearData = Data(count:clearLength)

    var numBytesDecrypted :size_t = 0
    let options   = CCOptions(kCCOptionPKCS7Padding)

    let cryptStatus = clearData.withUnsafeMutableBytes {cryptBytes in
        data.withUnsafeBytes {dataBytes in
            keyData.withUnsafeBytes {keyBytes in
              CCCrypt(CCOperation(kCCDecrypt),
                        CCAlgorithm(kCCAlgorithmAES128),
                        options,
keyBytes, keyLength,
                        dataBytes,
dataBytes+kCCBlockSizeAES128, clearLength,
                        cryptBytes, clearLength,
                            &numBytesDecrypted)
                }
            }
    }

    if UInt32(cryptStatus) == UInt32(kCCSuccess) {
        clearData.count = numBytesDecrypted
    }
    else {
throw AESError.CryptorError(("Decryption failed", Int(cryptStatus)))
    }

    return clearData;
}
```

示例用法：


```
                        options,
                        keyBytes, keyLength,
                        cryptBytes,
                        dataBytes, data.count,
                        cryptBytes+kCCBlockSizeAES128, cryptLength,
                        &numBytesEncrypted)
                }
            }
    }

    if UInt32(cryptStatus) == UInt32(kCCSuccess) {
        cryptData.count = numBytesEncrypted + ivSize
    }
    else {
        throw AESError.CryptorError(("Encryption failed", Int(cryptStatus)))
    }

    return cryptData;
}

// The iv is prefixed to the encrypted data
func aesCBCDecrypt(data:Data, keyData:Data) throws -> Data? {
    let keyLength = keyData.count
    let validKeyLengths = [kCCKeySizeAES128, kCCKeySizeAES192, kCCKeySizeAES256]
    if (validKeyLengths.contains(keyLength) == false) {
        throw AESError.KeyError(("Invalid key length", keyLength))
    }

    let ivSize = kCCBlockSizeAES128;
    let clearLength = size_t(data.count - ivSize)
    var clearData = Data(count:clearLength)

    var numBytesDecrypted :size_t = 0
    let options   = CCOptions(kCCOptionPKCS7Padding)

    let cryptStatus = clearData.withUnsafeMutableBytes {cryptBytes in
        data.withUnsafeBytes {dataBytes in
            keyData.withUnsafeBytes {keyBytes in
                CCCrypt(CCOperation(kCCDecrypt),
                        CCAlgorithm(kCCAlgorithmAES128),
                        options,
                        keyBytes, keyLength,
                        dataBytes,
                        dataBytes+kCCBlockSizeAES128, clearLength,
                        cryptBytes, clearLength,
                        &numBytesDecrypted)
            }
        }
    }

    if UInt32(cryptStatus) == UInt32(kCCSuccess) {
        clearData.count = numBytesDecrypted
    }
    else {
        throw AESError.CryptorError(("Decryption failed", Int(cryptStatus)))
    }

    return clearData;
}
```

Example usage:

```
let clearData = "clearData0123456".data(using:String.Encoding.utf8)!
let keyData   = "keyData890123456".data(using:String.Encoding.utf8)!
print("clearData:   \(clearData as NSData)")
print("keyData:     \(keyData as NSData)")

var cryptData :Data?
do {
cryptData = try aesCBCEncrypt(data:clearData, keyData:keyData)
    print("cryptData:   \(cryptData! as NSData)")
}
catch (let status) {
    print("Error aesCBCEncrypt: \(status)")
}

let decryptData :Data?
do {
    let decryptData = try aesCBCDecrypt(data:cryptData!, keyData:keyData)
    print("decryptData: \(decryptData! as NSData)")
}
catch (let status) {
    print("Error aesCBCDecrypt: \(status)")
}
```

示例输出：

```
clearData:    <636c6561 72446174 61303132 33343536>
keyData:      <6b657944 61746138 39303132 33343536>
cryptData:    <92c57393 f454d959 5a4d158f 6e1cd3e7 77986ee9 b2970f49 2bafcf1a 8ee9d51a bde49c31
d7780256 71837a61 60fa4be0>
decryptData: <636c6561 72446174 61303132 33343536>
```

注意事项：
CBC模式示例代码的一个典型问题是将随机初始化向量（IV）的生成和共享留给用户处理。此示例包含了IV的生成，将其作为前缀附加在加密数据之前，并在解密时使用该前缀IV。这使得普通用户无需关注CBC模式所必需的细节。

为了安全，加密数据也应具备认证功能，但此示例代码未提供该功能，以保持代码简洁并便于在其他平台上的互操作性。

同样缺失的是从密码派生密钥的过程，建议如果使用文本密码作为密钥材料，应使用PBKDF2。

有关健壮的、适用于多平台的生产级加密代码，请参见RNCryptor。

> 已更新为使用throw/catch机制，并根据提供的密钥支持多种密钥长度。

第56.2节：使用随机IV的CBC模式AES加密（Swift 2.3）

IV被加在加密数据的前面

aesCBC128Encrypt将创建一个随机IV并将其作为前缀附加到加密代码中。aesCBC128Decrypt将在解密时使用该前缀IV。

输入的数据和密钥均为Data对象。如果需要编码形式（如Base64），请进行相应的转换。


```
let clearData = "clearData0123456".data(using:String.Encoding.utf8)!
let keyData   = "keyData890123456".data(using:String.Encoding.utf8)!
print("clearData:   \(clearData as NSData)")
print("keyData:     \(keyData as NSData)")

var cryptData :Data?
do {
    cryptData = try aesCBCEncrypt(data:clearData, keyData:keyData)
    print("cryptData:   \(cryptData! as NSData)")
}
catch (let status) {
    print("Error aesCBCEncrypt: \(status)")
}

let decryptData :Data?
do {
    let decryptData = try aesCBCDecrypt(data:cryptData!, keyData:keyData)
    print("decryptData: \(decryptData! as NSData)")
}
catch (let status) {
    print("Error aesCBCDecrypt: \(status)")
}
```

Example Output:

```
clearData:   <636c6561 72446174 61303132 33343536>
keyData:     <6b657944 61746138 39303132 33343536>
cryptData:   <92c57393 f454d959 5a4d158f 6e1cd3e7 77986ee9 b2970f49 2bafcf1a 8ee9d51a bde49c31
d7780256 71837a61 60fa4be0>
decryptData: <636c6561 72446174 61303132 33343536>
```

Notes:
One typical problem with CBC mode example code is that it leaves the creation and sharing of the random IV to the user. This example includes generation of the IV, prefixed the encrypted data and uses the prefixed IV during decryption. This frees the casual user from the details that are necessary for CBC mode.

For security the encrypted data also should have authentication, this example code does not provide that in order to be small and allow better interoperability for other platforms.

Also missing is key derivation of the key from a password, it is suggested that PBKDF2 be used is text passwords are used as keying material.

For robust production ready multi-platform encryption code see RNCryptor.

> Updated to use throw/catch and multiple key sizes based on the provided key.

Section 56.2: AES encryption in CBC mode with a random IV (Swift 2.3)

The iv is prefixed to the encrypted data

aesCBC128Encrypt will create a random IV and prefixed to the encrypted code. aesCBC128Decrypt will use the prefixed IV during decryption.

Inputs are the data and key are Data objects. If an encoded form such as Base64 if required convert to and/or from

在调用方法中。

密钥应为精确的128位（16字节）。有关其他密钥大小，请参见Swift 3.0示例。

默认设置为PKCS#7填充。

此示例需要Common Crypto，项目中必须有桥接头文件：#import <CommonCrypto/CommonCrypto.h>，并将Security.framework添加到项目中。

有关说明，请参见Swift 3示例。

这是示例代码，不是生产代码。

```
func aesCBC128Encrypt(data data:[UInt8], keyData:[UInt8]) -> [UInt8]? {
    let keyLength   = size_t(kCCKeySizeAES128)
    let ivLength    = size_t(kCCBlockSizeAES128)
    let cryptDataLength = size_t(data.count + kCCBlockSizeAES128)
    var cryptData = [UInt8](count:ivLength + cryptDataLength, repeatedValue:0)

    let status = SecRandomCopyBytes(kSecRandomDefault, Int(ivLength),
UnsafeMutablePointer<UInt8>(cryptData));
    if (status != 0) {
        print("IV错误，错误码: \(status)")
        return nil
    }

    var numBytesEncrypted :size_t = 0
    let cryptStatus = CCCrypt(CCOperation(kCCEncrypt),
                          CCAlgorithm(kCCAlgorithmAES128),
                          CCOptions(kCCOptionPKCS7Padding),
                          keyData, keyLength,
cryptData,
                                data, data.count,
                                &cryptData + ivLength, cryptDataLength,
                                &numBytesEncrypted)

    if UInt32(cryptStatus) == UInt32(kCCSuccess) {
        cryptData.removeRange(numBytesEncrypted+ivLength..<cryptData.count)
    }
    else {
        print("Error: \(cryptStatus)")
        return nil;
    }

    return cryptData;
}

func aesCBC128Decrypt(data data:[UInt8], keyData:[UInt8]) -> [UInt8]? {
    let clearLength = size_t(data.count)
    var clearData   = [UInt8](count:clearLength, repeatedValue:0)

    let keyLength   = size_t(kCCKeySizeAES128)
    let ivLength    = size_t(kCCBlockSizeAES128)

    var numBytesDecrypted :size_t = 0
    let cryptStatus = CCCrypt(CCOperation(kCCDecrypt),
                          CCAlgorithm(kCCAlgorithmAES128),
                          CCOptions(kCCOptionPKCS7Padding),
                          keyData, keyLength,
                               data,
UnsafePointer<UInt8>(data) + ivLength, data.count - ivLength,
```


in the calling method.

The key should be exactly 128-bits (16-bytes). For other key sizes see the Swift 3.0 example.

PKCS#7 padding is set by default.

This example requires Common Crypto It is necessary to have a bridging header to the project: #import <CommonCrypto/CommonCrypto.h> Add the Security.framework to the project.

See Swift 3 example for notes.

This is example, not production code.

```
func aesCBC128Encrypt(data data:[UInt8], keyData:[UInt8]) -> [UInt8]? {
    let keyLength   = size_t(kCCKeySizeAES128)
    let ivLength    = size_t(kCCBlockSizeAES128)
    let cryptDataLength = size_t(data.count + kCCBlockSizeAES128)
    var cryptData = [UInt8](count:ivLength + cryptDataLength, repeatedValue:0)

    let status = SecRandomCopyBytes(kSecRandomDefault, Int(ivLength),
UnsafeMutablePointer<UInt8>(cryptData));
    if (status != 0) {
        print("IV Error, errno: \(status)")
        return nil
    }

    var numBytesEncrypted :size_t = 0
    let cryptStatus = CCCrypt(CCOperation(kCCEncrypt),
                              CCAlgorithm(kCCAlgorithmAES128),
                              CCOptions(kCCOptionPKCS7Padding),
                              keyData, keyLength,
                              cryptData,
                              data, data.count,
                              &cryptData + ivLength, cryptDataLength,
                              &numBytesEncrypted)

    if UInt32(cryptStatus) == UInt32(kCCSuccess) {
        cryptData.removeRange(numBytesEncrypted+ivLength..<cryptData.count)
    }
    else {
        print("Error: \(cryptStatus)")
        return nil;
    }

    return cryptData;
}

func aesCBC128Decrypt(data data:[UInt8], keyData:[UInt8]) -> [UInt8]? {
    let clearLength = size_t(data.count)
    var clearData   = [UInt8](count:clearLength, repeatedValue:0)

    let keyLength   = size_t(kCCKeySizeAES128)
    let ivLength    = size_t(kCCBlockSizeAES128)

    var numBytesDecrypted :size_t = 0
    let cryptStatus = CCCrypt(CCOperation(kCCDecrypt),
                              CCAlgorithm(kCCAlgorithmAES128),
                              CCOptions(kCCOptionPKCS7Padding),
                              keyData, keyLength,
                              data,
                              UnsafePointer<UInt8>(data) + ivLength, data.count - ivLength,
```

```
                                &clearData, clearLength,
                                &numBytesDecrypted)

    if UInt32(cryptStatus) == UInt32(kCCSuccess) {
        clearData.removeRange(numBytesDecrypted..<clearLength)

    } 否则 {
        print("Error: \(cryptStatus)")
        return nil;
    }

    return clearData;
}
```

示例用法：

```
let clearData = toData("clearData0123456")
let keyData   = toData("keyData890123456")

print("clearData:   \(toHex(clearData))")
print("keyData:     \(toHex(keyData))")
let cryptData = aesCBC128Encrypt(data:clearData, keyData:keyData)!
print("cryptData:   \(toHex(cryptData))")
let decryptData = aesCBC128Decrypt(data:cryptData, keyData:keyData)!
print("decryptData: \(toHex(decryptData))")
```

示例输出：

```
clearData:    <636c6561 72446174 61303132 33343536>
keyData:      <6b657944 61746138 39303132 33343536>
cryptData:    <9fce4323 830e3734 93dd93bf e464f72a a653a3a5 2c40d5ea e90c1017 958750a7 ff094c53
6a81b458 b1fbd6d4 1f583298>
decryptData: <636c6561 72446174 61303132 33343536>
```

第56.3节：使用PKCS7填充的ECB模式AES加密

摘自苹果文档关于IV，

> 如果使用ECB模式或选择了流密码算法，则忽略此参数。

```
func AESEncryption(key: String) -> String? {

        let keyData: NSData! = (key as NSString).data(using: String.Encoding.utf8.rawValue) as
NSData!

        let data: NSData! = (self as NSString).data(using: String.Encoding.utf8.rawValue) as
NSData!

        let cryptData    = NSMutableData(length: Int(data.length) + kCCBlockSizeAES128)!

        let keyLength              = size_t(kCCKeySizeAES128)
        let operation: CCOperation = UInt32(kCCEncrypt)
        let algoritm:  CCAlgorithm = UInt32(kCCAlgorithmAES128)
        let options:   CCOptions   = UInt32(kCCOptionECBMode + kCCOptionPKCS7Padding)

        var numBytesEncrypted :size_t = 0
```



```
                                &clearData, clearLength,
                                &numBytesDecrypted)

    if UInt32(cryptStatus) == UInt32(kCCSuccess) {
        clearData.removeRange(numBytesDecrypted..<clearLength)

    } else {
        print("Error: \(cryptStatus)")
        return nil;
    }

    return clearData;
}
```

Example usage:

```
let clearData = toData("clearData0123456")
let keyData   = toData("keyData890123456")

print("clearData:   \(toHex(clearData))")
print("keyData:     \(toHex(keyData))")
let cryptData = aesCBC128Encrypt(data:clearData, keyData:keyData)!
print("cryptData:   \(toHex(cryptData))")
let decryptData = aesCBC128Decrypt(data:cryptData, keyData:keyData)!
print("decryptData: \(toHex(decryptData))")
```

Example Output:

```
clearData:    <636c6561 72446174 61303132 33343536>
keyData:      <6b657944 61746138 39303132 33343536>
cryptData:    <9fce4323 830e3734 93dd93bf e464f72a a653a3a5 2c40d5ea e90c1017 958750a7 ff094c53
6a81b458 b1fbd6d4 1f583298>
decryptData: <636c6561 72446174 61303132 33343536>
```

Section 56.3: AES encryption in ECB mode with PKCS7 padding

From Apple documentation for IV,

> This parameter is ignored if ECB mode is used or if a stream cipher algorithm is selected.

```
func AESEncryption(key: String) -> String? {

        let keyData: NSData! = (key as NSString).data(using: String.Encoding.utf8.rawValue) as
NSData!

        let data: NSData! = (self as NSString).data(using: String.Encoding.utf8.rawValue) as
NSData!

        let cryptData    = NSMutableData(length: Int(data.length) + kCCBlockSizeAES128)!

        let keyLength              = size_t(kCCKeySizeAES128)
        let operation: CCOperation = UInt32(kCCEncrypt)
        let algoritm:  CCAlgorithm = UInt32(kCCAlgorithmAES128)
        let options:   CCOptions   = UInt32(kCCOptionECBMode + kCCOptionPKCS7Padding)

        var numBytesEncrypted :size_t = 0
```

```
        let cryptStatus = CCCrypt(operation,
                                  algoritm,
                                  options,
                                        keyData.bytes, keyLength,
                                        nil,
data.bytes, data.length,
                                        cryptData.mutableBytes, cryptData.length,
                                        &numBytesEncrypted)

        if UInt32(cryptStatus) == UInt32(kCCSuccess) {
            cryptData.length = Int(numBytesEncrypted)

            var bytes = [UInt8](repeating: 0, count: cryptData.length)
            cryptData.getBytes(&bytes, length: cryptData.length)

            var hexString = ""
            for byte in bytes {
hexString += String(format:"%02x", UInt8(byte))
            }

            return hexString
        }

        return nil
    }
```



```
        let cryptStatus = CCCrypt(operation,
                                  algoritm,
                                  options,
                                  keyData.bytes, keyLength,
                                  nil,
                                  data.bytes, data.length,
                                  cryptData.mutableBytes, cryptData.length,
                                  &numBytesEncrypted)

        if UInt32(cryptStatus) == UInt32(kCCSuccess) {
            cryptData.length = Int(numBytesEncrypted)

            var bytes = [UInt8](repeating: 0, count: cryptData.length)
            cryptData.getBytes(&bytes, length: cryptData.length)

            var hexString = ""
            for byte in bytes {
                hexString += String(format:"%02x", UInt8(byte))
            }

            return hexString
        }

        return nil
    }
```

第57章：PBKDF2密钥派生

第57.1节：基于密码的密钥派生2（Swift 3）

基于密码的密钥派生既可用于从密码文本派生加密密钥，也可用于保存密码以进行身份验证。

可以使用多种哈希算法，包括本示例代码提供的SHA1、SHA256、SHA512。

rounds参数用于使计算变慢，从而使攻击者每次尝试都必须花费大量时间。典型的延迟值在100毫秒到500毫秒之间，如果性能不可接受，可以使用较短的值。

此示例需要 Common Crypto
项目中必须有桥接头文件：
`#import <CommonCrypto/CommonCrypto.h>`
将Security.framework添加到项目中。

参数：

```
password      密码 字符串
salt          盐 数据
keyByteCount  生成的密钥字节数
rounds        迭代轮数

返回值        派生密钥
```

```
func pbkdf2SHA1(password: String, salt: Data, keyByteCount: Int, rounds: Int) -> Data? {
    return pbkdf2(hash:CCPBKDFAlgorithm(kCCPRFHmacAlgSHA1), password:password, salt:salt,
keyByteCount:keyByteCount, rounds:rounds)
}

func pbkdf2SHA256(密码: String, 盐: Data, 密钥字节数: Int, 轮数: Int) -> Data? {
    return pbkdf2(hash:CCPBKDFAlgorithm(kCCPRFHmacAlgSHA256), password:密码, salt:盐,
keyByteCount:密钥字节数, rounds:轮数)
}

func pbkdf2SHA512(密码: String, 盐: Data, 密钥字节数: Int, 轮数: Int) -> Data? {
    return pbkdf2(hash:CCPBKDFAlgorithm(kCCPRFHmacAlgSHA512), password:密码, salt:盐,
keyByteCount:密钥字节数, rounds:轮数)
}

func pbkdf2(hash :CCPBKDFAlgorithm, 密码: String, 盐: Data, 密钥字节数: Int, 轮数: Int)
-> Data? {
    let 密码数据 = 密码.data(using:String.Encoding.utf8)!
    var 派生密钥数据 = Data(repeating:0, count:密钥字节数)

    let 派生状态 = 派生密钥数据.withUnsafeMutableBytes {派生密钥字节 in
       盐.withUnsafeBytes { 盐字节 in

            CCKeyDerivationPBKDF(
CCPBKDFAlgorithm(kCCPBKDF2),
               密码, 密码数据.count,
               盐字节, 盐.count,
hash,
                  UInt32(轮数),
```


Chapter 57: PBKDF2 Key Derivation

Section 57.1: Password Based Key Derivation 2 (Swift 3)

Password Based Key Derivation can be used both for deriving an encryption key from password text and saving a password for authentication purposes.

There are several hash algorithms that can be used including SHA1, SHA256, SHA512 which are provided by this example code.

The rounds parameter is used to make the calculation slow so that an attacker will have to spend substantial time on each attempt. Typical delay values fall in the 100ms to 500ms, shorter values can be used if there is unacceptable performance.

This example requires Common Crypto
It is necessary to have a bridging header to the project:
`#import <CommonCrypto/CommonCrypto.h>`
Add the `Security.framework` to the project.

Parameters:

```
password      password String
salt          salt Data
keyByteCount  number of key bytes to generate
rounds        Iteration rounds

returns       Derived key
```

```
func pbkdf2SHA1(password: String, salt: Data, keyByteCount: Int, rounds: Int) -> Data? {
    return pbkdf2(hash:CCPBKDFAlgorithm(kCCPRFHmacAlgSHA1), password:password, salt:salt,
keyByteCount:keyByteCount, rounds:rounds)
}

func pbkdf2SHA256(password: String, salt: Data, keyByteCount: Int, rounds: Int) -> Data? {
    return pbkdf2(hash:CCPBKDFAlgorithm(kCCPRFHmacAlgSHA256), password:password, salt:salt,
keyByteCount:keyByteCount, rounds:rounds)
}

func pbkdf2SHA512(password: String, salt: Data, keyByteCount: Int, rounds: Int) -> Data? {
    return pbkdf2(hash:CCPBKDFAlgorithm(kCCPRFHmacAlgSHA512), password:password, salt:salt,
keyByteCount:keyByteCount, rounds:rounds)
}

func pbkdf2(hash :CCPBKDFAlgorithm, password: String, salt: Data, keyByteCount: Int, rounds: Int)
-> Data? {
    let passwordData = password.data(using:String.Encoding.utf8)!
    var derivedKeyData = Data(repeating:0, count:keyByteCount)

    let derivationStatus = derivedKeyData.withUnsafeMutableBytes {derivedKeyBytes in
        salt.withUnsafeBytes { saltBytes in

            CCKeyDerivationPBKDF(
                CCPBKDFAlgorithm(kCCPBKDF2),
                password, passwordData.count,
                saltBytes, salt.count,
                hash,
                UInt32(rounds),
```

```
派生密钥字节，派生密钥数据.count)
        }
    }
    if (派生状态 != 0) {
        print("错误: \(派生状态)")
        return nil;
    }

    return 派生密钥数据
}
```

示例用法：

```
let password     = "password"
//let salt       = "saltData".data(using: String.Encoding.utf8)!
let salt         = Data(bytes: [0x73, 0x61, 0x6c, 0x74, 0x44, 0x61, 0x74, 0x61])
let keyByteCount = 16
let rounds       = 100000

let derivedKey = pbkdf2SHA1(password:password, salt:salt, keyByteCount:keyByteCount, rounds:rounds)
print("derivedKey (SHA1): \(derivedKey! as NSData)")
```

示例输出：

```
derivedKey (SHA1): <6b9d4fa3 0385d128 f6d196ee 3f1d6dbf>
```

第57.2节：基于密码的密钥派生2（Swift 2.3）

有关用法信息和说明，请参见Swift 3示例

```
func pbkdf2SHA1(password: String, salt: [UInt8], keyCount: Int, rounds: Int) -> [UInt8]? {
    return pbkdf2(CCPBKDFAlgorithm(kCCPRFHmacAlgSHA1), password:password, salt:salt,
keyCount:keyCount, rounds:UInt32(rounds))
}

func pbkdf2SHA256(password: String, salt: [UInt8], keyCount: Int, rounds: Int) -> [UInt8]? {
    return pbkdf2(CCPBKDFAlgorithm(kCCPRFHmacAlgSHA256), password:password, salt:salt,
keyCount:keyCount, rounds:UInt32(rounds))
}

func pbkdf2SHA512(password: String, salt: [UInt8], keyCount: Int, rounds: Int) -> [UInt8]? {
    return pbkdf2(CCPBKDFAlgorithm(kCCPRFHmacAlgSHA512), password:password, salt:salt,
keyCount:keyCount, rounds:UInt32(rounds))
}

func pbkdf2(hash :CCPBKDFAlgorithm, password: String, salt: [UInt8], keyCount: Int, rounds:
UInt32!) -> [UInt8]! {
    let derivedKey   = [UInt8](count:keyCount, repeatedValue:0)
    let passwordData = password.dataUsingEncoding(NSUTF8StringEncoding)!

    let derivationStatus = CCKeyDerivationPBKDF(
       CCPBKDFAlgorithm(kCCPBKDF2),
        UnsafePointer<Int8>(passwordData.bytes), passwordData.length,
        UnsafePointer<UInt8>(salt), salt.count,
CCPseudoRandomAlgorithm(hash),
       rounds,
UnsafeMutablePointer<UInt8>(derivedKey),
        derivedKey.count)
```



```
                derivedKeyBytes, derivedKeyData.count)
        }
    }
    if (derivationStatus != 0) {
        print("Error: \(derivationStatus)")
        return nil;
    }

    return derivedKeyData
}
```

Example usage:

```
let password     = "password"
//let salt       = "saltData".data(using: String.Encoding.utf8)!
let salt         = Data(bytes: [0x73, 0x61, 0x6c, 0x74, 0x44, 0x61, 0x74, 0x61])
let keyByteCount = 16
let rounds       = 100000

let derivedKey = pbkdf2SHA1(password:password, salt:salt, keyByteCount:keyByteCount, rounds:rounds)
print("derivedKey (SHA1): \(derivedKey! as NSData)")
```

Example Output:

```
derivedKey (SHA1): <6b9d4fa3 0385d128 f6d196ee 3f1d6dbf>
```

Section 57.2: Password Based Key Derivation 2 (Swift 2.3)

See Swift 3 example for usage information and notes

```
func pbkdf2SHA1(password: String, salt: [UInt8], keyCount: Int, rounds: Int) -> [UInt8]? {
    return pbkdf2(CCPBKDFAlgorithm(kCCPRFHmacAlgSHA1), password:password, salt:salt,
keyCount:keyCount, rounds:UInt32(rounds))
}

func pbkdf2SHA256(password: String, salt: [UInt8], keyCount: Int, rounds: Int) -> [UInt8]? {
    return pbkdf2(CCPBKDFAlgorithm(kCCPRFHmacAlgSHA256), password:password, salt:salt,
keyCount:keyCount, rounds:UInt32(rounds))
}

func pbkdf2SHA512(password: String, salt: [UInt8], keyCount: Int, rounds: Int) -> [UInt8]? {
    return pbkdf2(CCPBKDFAlgorithm(kCCPRFHmacAlgSHA512), password:password, salt:salt,
keyCount:keyCount, rounds:UInt32(rounds))
}

func pbkdf2(hash :CCPBKDFAlgorithm, password: String, salt: [UInt8], keyCount: Int, rounds:
UInt32!) -> [UInt8]! {
    let derivedKey   = [UInt8](count:keyCount, repeatedValue:0)
    let passwordData = password.dataUsingEncoding(NSUTF8StringEncoding)!

    let derivationStatus = CCKeyDerivationPBKDF(
        CCPBKDFAlgorithm(kCCPBKDF2),
        UnsafePointer<Int8>(passwordData.bytes), passwordData.length,
        UnsafePointer<UInt8>(salt), salt.count,
        CCPseudoRandomAlgorithm(hash),
        rounds,
        UnsafeMutablePointer<UInt8>(derivedKey),
        derivedKey.count)
```

```
    if (派生状态 != 0) {
        print("错误: \(派生状态)")
        return nil;
    }

    return derivedKey
}
```

示例用法：

```
let password = "password"
// let salt  = [UInt8]("saltData".utf8)
let salt     = [UInt8]([0x73, 0x61, 0x6c, 0x74, 0x44, 0x61, 0x74, 0x61])
let rounds   = 100_000
let keyCount = 16

let derivedKey = pbkdf2SHA1(password, salt:salt, keyCount:keyCount, rounds:rounds)
print("derivedKey (SHA1):   \(NSData(bytes:derivedKey!, length:derivedKey!.count))")
```

示例输出：

```
derivedKey (SHA1):   <6b9d4fa3 0385d128 f6d196ee 3f1d6dbf>
```

第57.3节：基于密码的密钥派生校准（Swift 2.3）

有关用法信息和说明，请参见Swift 3示例

```
func pbkdf2SHA1Calibrate(password:String, salt:[UInt8], msec:Int) -> UInt32 {
    let actualRoundCount: UInt32 = CCCalibratePBKDF(
CCPBKDFAlgorithm(kCCPBKDF2),
        password.utf8.count,
salt.count,
CCPseudoRandomAlgorithm(kCCPRFHmacAlgSHA1),
       kCCKeySizeAES256,
        UInt32(msec));
    return actualRoundCount
}
```

示例用法：

```
let saltData       = [UInt8]([0x73, 0x61, 0x6c, 0x74, 0x44, 0x61, 0x74, 0x61])
let passwordString = "password"
let delayMsec      = 100

let rounds = pbkdf2SHA1Calibrate(passwordString, salt:saltData, msec:delayMsec)
print("For \(delayMsec) msec delay, rounds: \(rounds)")
```

示例输出：

> 对于 100 毫秒延迟，轮数：94339

第 57.4 节：基于密码的密钥派生校准


```
    if (derivationStatus != 0) {
        print("Error: \(derivationStatus)")
        return nil;
    }

    return derivedKey
}
```

Example usage:

```
let password = "password"
// let salt  = [UInt8]("saltData".utf8)
let salt     = [UInt8]([0x73, 0x61, 0x6c, 0x74, 0x44, 0x61, 0x74, 0x61])
let rounds   = 100_000
let keyCount = 16

let derivedKey = pbkdf2SHA1(password, salt:salt, keyCount:keyCount, rounds:rounds)
print("derivedKey (SHA1):   \(NSData(bytes:derivedKey!, length:derivedKey!.count))")
```

Example Output:

```
derivedKey (SHA1):   <6b9d4fa3 0385d128 f6d196ee 3f1d6dbf>
```

Section 57.3: Password Based Key Derivation Calibration (Swift 2.3)

See Swift 3 example for usage information and notes

```
func pbkdf2SHA1Calibrate(password:String, salt:[UInt8], msec:Int) -> UInt32 {
    let actualRoundCount: UInt32 = CCCalibratePBKDF(
        CCPBKDFAlgorithm(kCCPBKDF2),
        password.utf8.count,
        salt.count,
        CCPseudoRandomAlgorithm(kCCPRFHmacAlgSHA1),
        kCCKeySizeAES256,
        UInt32(msec));
    return actualRoundCount
}
```

Example usage:

```
let saltData       = [UInt8]([0x73, 0x61, 0x6c, 0x74, 0x44, 0x61, 0x74, 0x61])
let passwordString = "password"
let delayMsec      = 100

let rounds = pbkdf2SHA1Calibrate(passwordString, salt:saltData, msec:delayMsec)
print("For \(delayMsec) msec delay, rounds: \(rounds)")
```

Example Output:

> For 100 msec delay, rounds: 94339

Section 57.4: Password Based Key Derivation Calibration

(Swift 3)

确定当前平台上用于特定延迟的伪随机函数（PRF）轮数。

若干参数默认设置为代表性数值，这些数值不应实质性影响轮数。

```
password 示例密码。
salt     示例盐值。
msec     我们希望实现的密钥派生目标持续时间。

返回 用于期望处理时间的迭代次数。


func pbkdf2SHA1Calibrate(password: String, salt: Data, msec: Int) -> UInt32 {
    let actualRoundCount: UInt32 = CCCalibratePBKDF(
CCPBKDFAlgorithm(kCCPBKDF2),
         password.utf8.count,
salt.count,
CCPseudoRandomAlgorithm(kCCPRFHmacAlgSHA1),
        kCCKeySizeAES256,
          UInt32(msec));
    return actualRoundCount
}
```

示例用法：

```
let saltData       = Data(bytes: [0x73, 0x61, 0x6c, 0x74, 0x44, 0x61, 0x74, 0x61])
let passwordString = "password"
let delayMsec      = 100

let rounds = pbkdf2SHA1Calibrate(password:passwordString, salt:saltData, msec:delayMsec)
print("For \(delayMsec) msec delay, rounds: \(rounds)")
```

示例输出：

```
对于 100 毫秒延迟，迭代次数：93457
```


(Swift 3)

Determine the number of PRF rounds to use for a specific delay on the current platform.

Several parameters are defaulted to representative values that should not materially affect the round count.

```
password Sample password.
salt     Sample salt.
msec     Targeted duration we want to achieve for a key derivation.

returns  The number of iterations to use for the desired processing time.


func pbkdf2SHA1Calibrate(password: String, salt: Data, msec: Int) -> UInt32 {
    let actualRoundCount: UInt32 = CCCalibratePBKDF(
        CCPBKDFAlgorithm(kCCPBKDF2),
        password.utf8.count,
        salt.count,
        CCPseudoRandomAlgorithm(kCCPRFHmacAlgSHA1),
        kCCKeySizeAES256,
        UInt32(msec));
    return actualRoundCount
}
```

Example usage:

```
let saltData       = Data(bytes: [0x73, 0x61, 0x6c, 0x74, 0x44, 0x61, 0x74, 0x61])
let passwordString = "password"
let delayMsec      = 100

let rounds = pbkdf2SHA1Calibrate(password:passwordString, salt:saltData, msec:delayMsec)
print("For \(delayMsec) msec delay, rounds: \(rounds)")
```

Example Output:

```
For 100 msec delay, rounds: 93457
```

第58章：Swift中的日志记录

第58.1节：dump

dump 通过反射（镜像）打印对象的内容。

数组的详细视图：

```
let names = ["Joe", "Jane", "Jim", "Joyce"]
dump(names)
```

打印：

```
▿ 4 个元素
- [0]: 乔
- [1]: 简
- [2]: 吉姆
- [3]: 乔伊斯
```

对于字典：

```
let attributes = ["foo": 10, "bar": 33, "baz": 42]
dump(attributes)
```

打印：

```
▿ 3 个键/值对
▿ [0]: (2 个元素)
- .0: bar
- .1: 33
▿ [1]: (2 个元素)
- .0: baz
- .1: 42
▿ [2]: (2 个元素)
- .0: foo
- .1: 10
```

dump 被声明为 dump(_:name:indent:maxDepth:maxItems:)。

第一个参数没有标签。

还有其他可用参数，比如 name 用于设置被检查对象的标签：

```
dump(attributes, name: "mirroring")
```

打印：

```
▿ mirroring: 3 个键/值对
▿ [0]: (2 个元素)
```


Chapter 58: Logging in Swift

Section 58.1: dump

`dump` prints the contents of an object via reflection (mirroring).

Detailed view of an array:

```
let names = ["Joe", "Jane", "Jim", "Joyce"]
dump(names)
```

Prints:

```
▿ 4 elements
- [0]: Joe
- [1]: Jane
- [2]: Jim
- [3]: Joyce
```

For a dictionary:

```
let attributes = ["foo": 10, "bar": 33, "baz": 42]
dump(attributes)
```

Prints:

```
▿ 3 key/value pairs
▿ [0]: (2 elements)
- .0: bar
- .1: 33
▿ [1]: (2 elements)
- .0: baz
- .1: 42
▿ [2]: (2 elements)
- .0: foo
- .1: 10
```

`dump` is declared as `dump(_:name:indent:maxDepth:maxItems:)`.

The first parameter has no label.

There's other parameters available, like `name` to set a label for the object being inspected:

```
dump(attributes, name: "mirroring")
```

Prints:

```
▿ mirroring: 3 key/value pairs
▿ [0]: (2 elements)
```

- .0: bar
- .1: 33
▿ [1]: (2 个元素)
- .0: baz
- .1: 42
▿ [2]: (2 个元素)
- .0: foo
- .1: 10

你也可以选择只打印一定数量的项目，使用 maxItems:，解析对象到一定深度，使用 maxDepth:，或者更改打印对象的缩进，使用 indent:。

第58.2节：调试打印

调试打印显示最适合调试的实例表示。

```
print("Hello")
debugPrint("Hello")

let dict = ["foo": 1, "bar": 2]

print(dict)
debugPrint(dict)
```

输出结果

```
>>> Hello
>>> "Hello"
>>> [foo: 1, bar: 2]
>>> ["foo": 1, "bar": 2]
```

这些额外信息可能非常重要，例如：

```
let wordArray = ["foo", "bar", "food, bars"]

print(wordArray)
debugPrint(wordArray)
```

输出结果

```
>>> [foo, bar, food, bars]
>>> ["foo", "bar", "food, bars"]
```

请注意，在第一个输出中，数组中似乎有4个元素，而不是3个。出于这样的原因，调试时更推荐使用debugPrint

更新类的调试和打印值

```
struct Foo: Printable, DebugPrintable {
    var description: String {return "对象的清晰描述"}
    var debugDescription: String {return "有助于调试的信息"}
}

var foo = Foo()

print(foo)
```


- .0: bar
- .1: 33
▿ [1]: (2 elements)
- .0: baz
- .1: 42
▿ [2]: (2 elements)
- .0: foo
- .1: 10

You can also choose to print only a certain number of items with `maxItems:`, to parse the object up to a certain depth with `maxDepth:`, and to change the indentation of printed objects with `indent:`.

Section 58.2: Debug Print

Debug Print shows the instance representation that is most suitable for debugging.

```
print("Hello")
debugPrint("Hello")

let dict = ["foo": 1, "bar": 2]

print(dict)
debugPrint(dict)
```

Yields

```
>>> Hello
>>> "Hello"
>>> [foo: 1, bar: 2]
>>> ["foo": 1, "bar": 2]
```

This extra information can be very important, for example:

```
let wordArray = ["foo", "bar", "food, bars"]

print(wordArray)
debugPrint(wordArray)
```

Yields

```
>>> [foo, bar, food, bars]
>>> ["foo", "bar", "food, bars"]
```

Notice how in the first output it appears that there are 4 elements in the array as opposed to 3. For reasons like this, it is preferable when debugging to use `debugPrint`

Updating a classes debug and print values

```
struct Foo: Printable, DebugPrintable {
    var description: String {return "Clear description of the object"}
    var debugDescription: String {return "Helpful message for debugging"}
}

var foo = Foo()

print(foo)
```

```
debugPrint(foo)

>>> 对象的清晰描述
>>> 有助于调试的信息
```

第58.3节：print() 与 dump()

我们许多人开始调试时会用简单的print()。假设我们有这样一个类：

```
class Abc {
    let a = "aa"
    let b = "bb"
}
```

我们有一个Abc的实例，如下所示：

```
let abc = Abc()
```

当我们对变量运行print()时，输出是

```
App.Abc
```

而 dump() 输出的是

```
App.Abc #0
-a: "aa"
-b: "bb"
```

如所见，dump() 输出整个类层次结构，而print()仅输出类名。

因此，dump() 对于UI调试特别有用

```
let view = UIView(frame: CGRect(x: 0, y: 0, width: 100, height: 100))
```

使用dump(view)我们得到：

```
- <UIView: 0x108a0cde0; frame = (0 0; 100 100); layer = <CALayer: 0x159340cb0>> #0
    - super: UIResponder
       - NSObject
```

使用print(view)我们得到：

```
<UIView: 0x108a0cde0; frame = (0 0; 100 100); layer = <CALayer: 0x159340cb0>>
```

使用dump()可以获得更多关于类的信息，因此在调试类本身时更有用。

第58.4节：print与NSLog

在Swift中，我们可以使用print()和NSLog()函数在Xcode控制台打印内容。

但print()和NSLog()函数有很多区别，例如：

1 时间戳：NSLog() 会打印时间戳以及我们传递给它的字符串，但 print() 不会打印时间戳。

例如。


```
debugPrint(foo)

>>> Clear description of the object
>>> Helpful message for debugging
```

Section 58.3: print() vs dump()

Many of us start debugging with simple `print()`. Let's say we have such a class:

```
class Abc {
    let a = "aa"
    let b = "bb"
}
```

and we have an instance of `Abc` as so:

```
let abc = Abc()
```

When we run the `print()` on the variable, the output is

```
App.Abc
```

while `dump()` outputs

```
App.Abc #0
- a: "aa"
- b: "bb"
```

As seen, `dump()` outputs the whole class hierarchy, while `print()` simply outputs the class name.

Therefore, `dump()` is especially useful for UI debugging

```
let view = UIView(frame: CGRect(x: 0, y: 0, width: 100, height: 100))
```

With `dump(view)` we get:

```
- <UIView: 0x108a0cde0; frame = (0 0; 100 100); layer = <CALayer: 0x159340cb0>> #0
   - super: UIResponder
      - NSObject
```

While `print(view)` we get:

```
<UIView: 0x108a0cde0; frame = (0 0; 100 100); layer = <CALayer: 0x159340cb0>>
```

There is more info on the class with `dump()`, and so it is more useful in debugging the class itself.

Section 58.4: print vs NSLog

In swift we can use both `print()` and `NSLog()` functions to print something on Xcode console.

But there are lot of differences in `print()` and `NSLog()` functions, such as:

1 TimeStamp: `NSLog()` will print timestamp along with the string we passed to it, but `print()` will not print timestamp.
e.g.

```
let array = [1, 2, 3, 4, 5]
print(array)
NSLog(array.description)
```

输出：

> [1, 2, 3, 4, 5]
> 2017-05-31 13:14:38.582 ProjetName[2286:7473287] [1, 2, 3, 4, 5]

它还会打印ProjectName以及时间戳。

2 仅限字符串：NSLog()只接受字符串作为输入，但print()可以打印传入的任何类型的输入。

例如。

```
let array = [1, 2, 3, 4, 5]
print(array) //打印 [1, 2, 3, 4, 5]
NSLog(array) //错误：无法将类型[Int]的值转换为预期的参数类型'String'
```

3 性能：NSLog() 函数相比于 print() 函数非常 慢。

4 同步：NSLog() 处理多线程环境中的同时使用，并且打印输出时不会重叠。但 print() 不会处理这种情况，打印输出时会混乱。

5 设备控制台：NSLog() 也会输出到设备控制台，我们可以通过将设备连接到 Xcode 来查看该输出。 print() 不会打印输出到设备控制台。


```
let array = [1, 2, 3, 4, 5]
print(array)
NSLog(array.description)
```

Output:

> [1, 2, 3, 4, 5]
> 2017-05-31 13:14:38.582 ProjetName[2286:7473287] [1, 2, 3, 4, 5]

It'll also print **ProjectName** along with timestamp.

2 Only String: `NSLog()` only takes String as an input, but `print()` can print any type of input passed to it.
e.g.

```
let array = [1, 2, 3, 4, 5]
print(array) //prints [1, 2, 3, 4, 5]
NSLog(array) //error: Cannot convert value of type [Int] to expected argument type 'String'
```

3 Performance: `NSLog()` function is very **slow** compare to `print()` function.

4 Synchronization: `NSLog()` handles simultaneous usage from multi-threading environment and prints output without overlapping it. But `print()` will not handle such cases and jumbles while prating output.

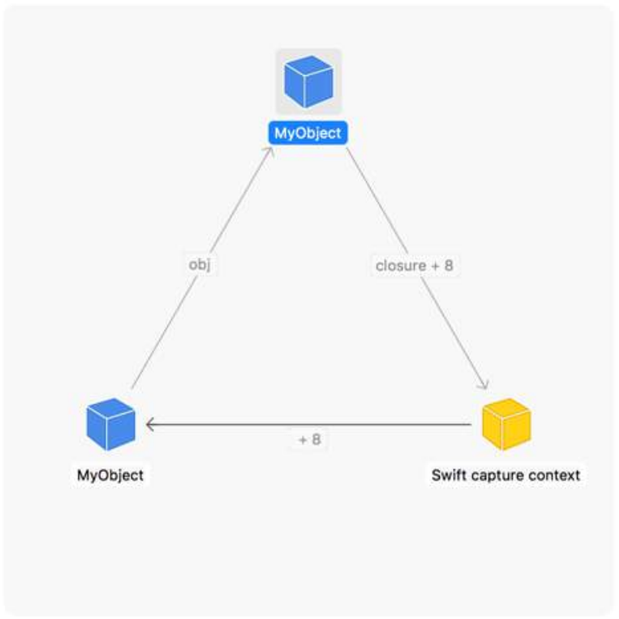
5 Device Console: `NSLog()` outputs on device console also, we can see this output by connecting our device to Xcode. `print()` will not print output to device's console.

第59章：内存管理

本主题概述了 Swift 运行时如何以及何时为应用程序数据结构分配内存，以及何时回收这些内存。默认情况下，类实例所使用的内存通过引用计数进行管理。结构体总是通过复制传递。若想退出内置的内存管理方案，可以使用 [Unmanaged][1] 结构体。[1]: https://developer.apple.com/reference/swift/unmanaged

第59.1节：引用循环和弱引用

“引用循环”（或“保留循环”）之所以如此命名，是因为它表示“对象图”中的一个“循环”：



每个箭头表示一个对象“保留”另一个对象（强引用）。除非打破循环，否则这些对象的内存将“永远不会被释放”。

当两个类的实例相互引用时，会产生保留循环：

```
class A { var b: B? = nil }
class B { var a: A? = nil }

let a = A()
let b = B()

a.b = b  // a 保留 b
b.a = a  // b 保留 a -- 一个引用循环
```

这两种情况都会一直存在直到程序终止。这就是保留循环。

弱引用

为了避免保留循环，创建引用时使用关键字weak或unowned来打破保留循环。

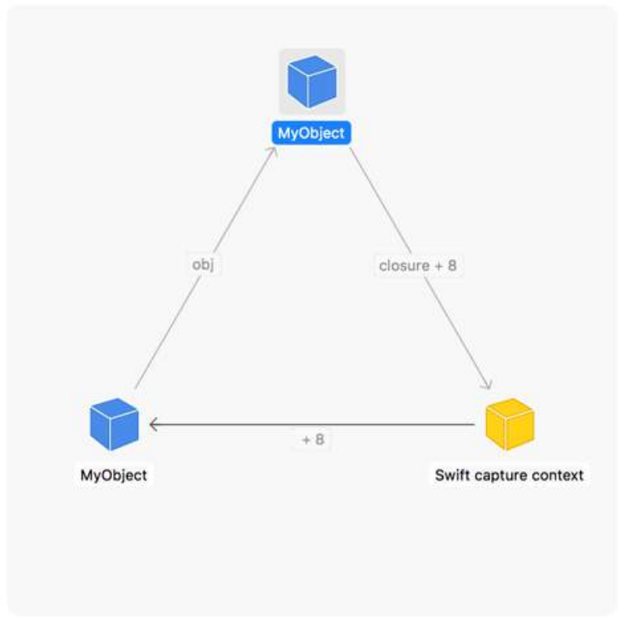
class B { var a: A? = nil }

Chapter 59: Memory Management

This topic outlines how and when the Swift runtime shall allocate memory for application data structures, and when that memory shall be reclaimed. By default, the memory backing class instances is managed through reference counting. The structures are always passed through copying. To opt out of the built-in memory management scheme, one could use [Unmanaged][1] structure. [1]: https://developer.apple.com/reference/swift/unmanaged

Section 59.1: Reference Cycles and Weak References

A *reference cycle* (or *retain cycle*) is so named because it indicates a cycle in the object graph:



Each arrow indicates one object retaining another (a strong reference). Unless the cycle is broken, the memory for these objects will **never be freed**.

A retain cycle is created when two instances of classes reference each other:

```
class A { var b: B? = nil }
class B { var a: A? = nil }

let a = A()
let b = B()

a.b = b  // a retains b
b.a = a  // b retains a -- a reference cycle
```

Both instances they will live on until the program terminates. This is a retain cycle.

Weak References

To avoid retain cycles, use the keyword `weak` or `unowned` when creating references to break retain cycles.

`class B { var a: A? = nil }`

弱引用或无主引用不会增加实例的引用计数。这些引用不会
导致保留循环。当被引用的对象被释放时，弱引用**会变为**`nil`。

```
a.b = b  // a 保留 b
b.a = a  // b 持有 a 的弱引用 -- 不是引用循环
```

在使用闭包时，也可以在捕获列表中使用weak和unowned。

第59.2节：手动内存管理

在与 C API 交互时，可能需要绕过 Swift 的引用计数器。实现这一点的方法是使用
unmanaged 对象。

如果需要向 C 函数提供类型转换的指针，请使用 Unmanaged 结构的 toOpaque 方法获取原始指针，使用 fromOpaque 恢复原始实例：

```
setupDisplayLink() {
  let pointerToSelf: UnsafeRawPointer = Unmanaged.passUnretained(self).toOpaque()
  CVDisplayLinkSetOutputCallback(self.displayLink, self.redraw, pointerToSelf)
}

func redraw(pointerToSelf: UnsafeRawPointer, /* args omitted */) {
  let recoveredSelf = Unmanaged<Self>.fromOpaque(pointerToSelf).takeUnretainedValue()
  recoveredSelf.doRedraw()
}
```

注意，使用 passUnretained 及其对应方法时，必须像使用 unowned

引用一样采取所有预防措施。

为了与传统的 Objective-C API 交互，可能需要手动影响某个对象的引用计数。为此，Unmanaged 提供了相应的 retain 和 release 方法。不过，更推荐使用 passRetained和 takeRetainedValue，这两个方法会在返回结果前执行保留操作：

```
func preferredFilenameExtension(for uti: String) -> String! {
  let result = UTTypeCopyPreferredTagWithClass(uti, kUTTagClassFilenameExtension)
  guard result != nil else { return nil }

  return result!.takeRetainedValue() as String
}
```

这些方案应始终作为最后手段，优先使用语言本身的 API。

Weak or unowned references will not increase the reference count of an instance. These references don't contribute to retain cycles. The weak reference **becomes** `nil` when the object it references is deallocated.

```
a.b = b  // a retains b
b.a = a  // b holds a weak reference to a -- not a reference cycle
```

When working with closures, you can also use `weak` and `unowned` in capture lists.

Section 59.2: Manual Memory Management

When interfacing with C APIs, one might want to back off Swift reference counter. Doing so is achieved with unmanaged objects.

If you need to supply a type-punned pointer to a C function, use `toOpaque` method of the `Unmanaged` structure to obtain a raw pointer, and `fromOpaque` to recover the original instance:

```
setupDisplayLink() {
  let pointerToSelf: UnsafeRawPointer = Unmanaged.passUnretained(self).toOpaque()
  CVDisplayLinkSetOutputCallback(self.displayLink, self.redraw, pointerToSelf)
}

func redraw(pointerToSelf: UnsafeRawPointer, /* args omitted */) {
  let recoveredSelf = Unmanaged<Self>.fromOpaque(pointerToSelf).takeUnretainedValue()
  recoveredSelf.doRedraw()
}
```

Note that, if using `passUnretained` and counterparts, it's necessary to take all precautions as with `unowned` references.

To interact with legacy Objective-C APIs, one might want to manually affect reference count of a certain object. For that `Unmanaged` has respective methods `retain` and `release`. Nonetheless, it is more desired to use `passRetained` and `takeRetainedValue`, which perform retaining before returning the result:

```
func preferredFilenameExtension(for uti: String) -> String! {
  let result = UTTypeCopyPreferredTagWithClass(uti, kUTTagClassFilenameExtension)
  guard result != nil else { return nil }

  return result!.takeRetainedValue() as String
}
```

These solutions should always be the last resort, and language-native APIs sould always be preferred.

第60章：性能

第60.1节：分配性能

在Swift中，内存管理是通过自动引用计数（Automatic Reference Counting）自动完成的。（参见内存管理）分配是为对象在内存中预留空间的过程，在Swift中理解这种分配的性能需要对堆（heap）和栈（stack）有所了解。堆是大多数对象存放的内存位置，你可以把它想象成一个储物棚。另一方面，栈是导致当前执行的函数调用栈。（因此，栈跟踪是一种调用栈中函数的打印输出。）

从栈上分配和释放是非常高效的操作，然而相比之下，堆分配的开销较大。在设计性能时，你应当牢记这一点。

类：

```
class MyClass {

    let myProperty: String

}
```

Swift中的类是引用类型，因此会发生几件事。首先，实际对象会被分配到堆上。然后，任何对该对象的引用必须被添加到栈上。这使得类的分配成本更高。

结构体：

```
struct MyStruct {

    let myProperty: Int

}
```

由于结构体是值类型，因此在传递时会被复制，它们被分配在栈上。这使得结构体比类更高效，然而，如果你需要身份标识和/或引用语义，结构体无法提供这些功能。

关于包含字符串和类属性的结构体的警告

虽然结构体通常比类更轻量，但你应该注意包含类属性的结构体：

```
struct MyStruct {

    let myProperty: MyClass

}
```

这里，由于引用计数和其他因素，性能现在更接近类的表现。此外，如果结构体中有多个属性是类，性能影响可能比结构体本身是类时更为负面。

另外，虽然字符串是结构体，但它们内部将字符存储在堆上，因此比大多数结构体更耗费资源。

Chapter 60: Performance

Section 60.1: Allocation Performance

In Swift, memory management is done for you automatically using Automatic Reference Counting. (See Memory Management) Allocation is the process of reserving a spot in memory for an object, and in Swift understanding the performance of such requires some understanding of the **heap** and the **stack**. The heap is a memory location where most objects get placed, and you may think of it as a storage shed. The stack, on the other hand, is a call stack of functions that have led to the current execution. (Hence, a stack trace is a sort of printout of the functions on the call stack.)

Allocating and deallocating from the stack is a very efficient operation, however in comparison heap allocation is costly. When designing for performance, you should keep this in mind.

Classes:

```
class MyClass {

    let myProperty: String

}
```

Classes in Swift are reference types and therefore several things happen. First, the actual object will be allocated onto the heap. Then, any references to that object must be added to the stack. This makes classes a more expensive object for allocation.

Structs:

```
struct MyStruct {

    let myProperty: Int

}
```

Because structs are value types and therefore copied when passed around, they are allocated on the stack. This makes structs more efficient than classes, however, if you do need a notion of identity and/or reference semantics, a struct cannot provide you with those things.

Warning about structs with Strings and properties that are classes

While structs are generally cheeper than classes, you should be careful about structs with properties that are classes:

```
struct MyStruct {

    let myProperty: MyClass

}
```

Here, due to reference counting and other factors, the performance is now more similar to a class. Further, if more than one property in the struct is a class, the performance impact may be even more negative than if the struct were a class instead.

Also, while Strings are structs, they internally store their characters on the heap, so are more expensive than most

结构体。

structs.

鸣谢

非常感谢所有来自Stack Overflow Documentation的人员帮助提供此内容，更多更改可发送至web@petercv.com以发布或更新新内容

4444	第23章
阿卜杜勒·亚辛	第43章
已采纳答案	第7、17、42和59章
亚当·巴登	第58章
Adda_25	第32章
艾哈迈德·F	第52章
阿吉特·R·纳亚克	第18章
阿杰特怀特韦	第12章
AK1	第9、12和14章
亚历山德罗	第23章
亚历山德罗·奥鲁	第33章
亚历克斯·波波夫	第8章
亚历山大·奥尔费鲁克	第40章
AMAN77	第52章
阿南德·尼姆杰	第21章
安德烈亚·安东尼奥尼	第4章
安德烈亚斯	第5章
安德烈·戈尔杰夫	第22章
安迪·伊巴涅斯	第18章
andyvn22	第21章
antonio081014	第4章
阿斯德鲁巴尔	第27章和第28章
AstroCB	第4章
atxe	第14章
阿维	第8章
avismara	第25章
斧头	第4章
巴特ł奥米耶·塞曼ń奇克	第44章
本·特伦格罗夫	第6章
brduca	第14、19、30和52章
凯勒布·克莱维特	第4、7、13、16、18和44章
克里斯托弗·厄兹贝克	第2章
科里·威尔海特	第17章
ctietze	第29章
西里尔·伊瓦尔·加西亚	第24章
D31	第16章
达塔特拉亚	第6章、第44章和第58章
达利娅·普拉斯尼卡尔	第10章和第17章
丹·哈比布	第58章
暗尘	第6章、第14章和第22章
大卫	第19章
迪奥戈·安图内斯	第8章、第9章和第11章
邓肯·C	第12章和第29章
梯队	第34章和第39章
egor.zhdan	第4章和第15章
elprl	第12章
埃斯卡鲁斯	第19章

Credits

Thank you greatly to all the people from Stack Overflow Documentation who helped provide this content, more changes can be sent to web@petercv.com for new content to be published or updated

4444	Chapter 23
Abdul Yasin	Chapter 43
Accepted Answer	Chapters 7, 17, 42 and 59
Adam Bardon	Chapter 58
Adda_25	Chapter 32
Ahmad F	Chapter 52
Ajith R Nayak	Chapter 18
Ajwhiteway	Chapter 12
AK1	Chapters 9, 12 and 14
Alessandro	Chapter 23
Alessandro Orrù	Chapter 33
Alex Popov	Chapter 8
Alexander Olferuk	Chapter 40
AMAN77	Chapter 52
Anand Nimje	Chapter 21
Andrea Antonioni	Chapter 4
Andreas	Chapter 5
Andrey Gordeev	Chapter 22
Andy Ibanez	Chapter 18
andyvn22	Chapter 21
antonio081014	Chapter 4
Asdrubal	Chapters 27 and 28
AstroCB	Chapter 4
atxe	Chapter 14
Avi	Chapter 8
avismara	Chapter 25
Axe	Chapter 4
Bartłomiej Semańczyk	Chapter 44
Ben Trengrove	Chapter 6
brduca	Chapters 14, 19, 30 and 52
Caleb Kleveter	Chapters 4, 7, 13, 16, 18 and 44
Christopher Oezbek	Chapter 2
Cory Wilhite	Chapter 17
ctietze	Chapter 29
Cyril Ivar Garcia	Chapter 24
D31	Chapter 16
D4ttatraya	Chapters 6, 44 and 58
Dalija Prasnikar	Chapters 10 and 17
DanHabib	Chapter 58
DarkDust	Chapters 6, 14 and 22
David	Chapter 19
Diogo Antunes	Chapters 8, 9 and 11
Duncan C	Chapters 12 and 29
Echelon	Chapters 34 and 39
egor.zhdan	Chapters 4 and 15
elprl	Chapter 12
Esqarrouth	Chapter 19

esthepiking	第1章和第17章
宁方明	第50章
法蒂	第31章
费尔杜尔	第4章
费利克斯SFD	第2、28、30和32章
费伦茨·基什	第1章
弗雷德·福斯特	第16章和第33章
fredpi	第9章
格伦·R·费舍尔	第22章和第24章
godisgood4	第21章
戈文德·赖	第4章
吉列尔梅·托雷斯·卡斯特罗	第4章
哈迪·努拉拉	第39章和第47章
哈米什	第4、6、8、13、14、16、17、22、25和29章
哈里克里希南·P	第4章
他遇见了	第4章
伊恩·拉赫曼	第24、52和53章
我相信	第17章
伊丹	第16章
意图	第25章
iOS开发中心	第11章
杰克·乔利	第24章
[illegible]	第4章、第25章、第26章、第31章和第32章
杰森·伯恩	第15章
杰夫·刘易斯	第16章
吉姆	第1章
琼	第12章
乔乔德莫	第2章、第4章、第7章、第13章、第19章、第22章和第29章
乔什·布朗	第9章
JPetric	第45章
jtbandes	第1、3、4、5、6、8、14、15、17、18、19、20、24、29和42章
juanjo	第8和13章
kabiroberai	第4和33章
kennytm	第25章
Kevin	第5、6和9章
基里特·莫迪	第11章
科特	第18章
考希克	第11章
库马尔·维韦克·米特拉	第35章
凯尔·金	第4章
洛佩	第4章
洛普赛	第4、21和29章
迷失在海上的约书亚	第24章
卢卡·安杰莱蒂	第1、4、6、7、9、10、12、16、18和34章
卢卡·安焦洛尼	第1章
卢卡·达尔贝尔蒂	第22章和第24章
卢克	第2章
卢克·赛德沃克	第4和32章
马哈茂德·亚当	第4章
马库斯·罗塞尔	第9章
马克	第42章
马丁·德利尔	第43章
马特	第20、36和56章

esthepiking	Chapters 1 and 17
Fangming Ning	Chapter 50
Fattie	Chapter 31
Feldur	Chapter 4
FelixSFD	Chapters 2, 28, 30 and 32
Ferenc Kiss	Chapter 1
Fred Faust	Chapters 16 and 33
fredpi	Chapter 9
Glenn R. Fisher	Chapters 22 and 24
godisgood4	Chapter 21
Govind Rai	Chapter 4
Guilherme Torres Castro	Chapter 4
Hady Nourallah	Chapters 39 and 47
Hamish	Chapters 4, 6, 8, 13, 14, 16, 17, 22, 25 and 29
HariKrishnan.P	Chapter 4
HeMet	Chapter 4
Ian Rahman	Chapters 24, 52 and 53
iBelieve	Chapter 17
Idan	Chapter 16
Intentss	Chapter 25
iOSDevCenter	Chapter 11
Jack Chorley	Chapter 24
JAL	Chapters 4, 25, 26, 31 and 32
Jason Bourne	Chapter 15
Jeff Lewis	Chapter 16
Jim	Chapter 1
joan	Chapter 12
Jojodmo	Chapters 2, 4, 7, 13, 19, 22 and 29
Josh Brown	Chapter 9
JPetric	Chapter 45
jtbandes	Chapters 1, 3, 4, 5, 6, 8, 14, 15, 17, 18, 19, 20, 24, 29 and 42
juanjo	Chapters 8 and 13
kabiroberai	Chapters 4 and 33
kennytm	Chapter 25
Kevin	Chapters 5, 6 and 9
Kirit Modi	Chapter 11
Kote	Chapter 18
Koushik	Chapter 11
Kumar Vivek Mitra	Chapter 35
Kyle KIM	Chapter 4
Lope	Chapter 4
LopSae	Chapters 4, 21 and 29
lostAtSeaJoshua	Chapter 24
Luca Angeletti	Chapters 1, 4, 6, 7, 9, 10, 12, 16, 18 and 34
Luca Angioloni	Chapter 1
Luca D'Alberti	Chapters 22 and 24
Luke	Chapter 2
LukeSideWalker	Chapters 4 and 32
Mahmoud Adam	Chapter 4
Marcus Rossel	Chapter 9
Mark	Chapter 42
Martin Delille	Chapter 43
Matt	Chapters 20, 36 and 56

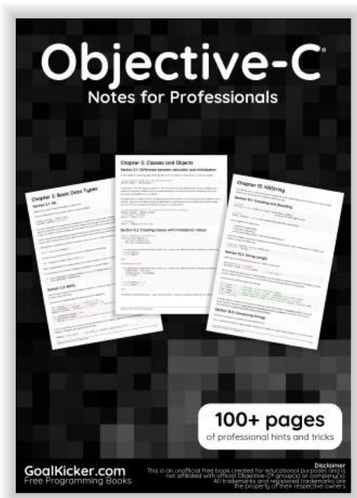
matt.baranowski	第17章
马修·西曼	第4、8、17、25、29、32和60章
马克斯·德西亚托夫	第4章
maxkonovalov	第24章
迈萨姆	第49章
梅胡尔·索吉特拉	第15章
米歇尔·阿泽维多	第13章
莫里茨	第4、6、7、13、15、19、24、41和58章
森谷	第6章
Xcoder先生	第16章
M_G	第32章
内特·库克	第4章
内森·凯勒特	第7、8和13章
尼克·波德拉茨	第8和21章
尼古拉·鲁赫	第4和8章
诺姆	第26章
努尔	第18章
奥列格·达努	第17章
orccrusher99	第25章
pableiros	第5章和第6章
Palle	第32章、第37章、第38章和第59章
Panda	第4章
Paulw11	第13章
pixatlazaki	第4章
拉胡尔	第17章
Rick Pasveer	第7章
罗布	第32章
罗布·纳皮尔	第9章
罗纳德·马丁	第7章
RubberDucky4444	第51章
瑞安·H.	第20章和第38章
saagarjha	第6章、第11章和第18章
萨加尔·图马尔	第48章
萨琼	第27章
圣诞老人	第2、12、30和32章
大脚怪	第11章
sdasdadas	第4章
SeanRobinson159	第8和17章
赛义德·帕尔萨·内沙伊	第1章
shannoga	第8和13章
吕诗静	第34章
shim	第4章
SKOOP	第13章和第17章
solidcell	第4章
Sơn_Đỗ_Đình_Thy	第52章
SteBra	第47章
史蒂夫·莫泽	第10章
苏尼特·蒂皮尔内尼	第11章和第22章
苏尼尔·普拉贾帕蒂	第1章
苏尼尔·夏尔马	第4章
苏拉格奇	第3章和第4章
坦纳	第1章和第13章
泰勒·斯威夫特	第4章

matt.baranowski	Chapter 17
Matthew Seaman	Chapters 4, 8, 17, 25, 29, 32 and 60
Max Desiatov	Chapter 4
maxkonovalov	Chapter 24
Maysam	Chapter 49
Mehul Sojitra	Chapter 15
Michaël Azevedo	Chapter 13
Moritz	Chapters 4, 6, 7, 13, 15, 19, 24, 41 and 58
Moriya	Chapter 6
Mr. Xcoder	Chapter 16
M_G	Chapter 32
Nate Cook	Chapter 4
Nathan Kellert	Chapters 7, 8 and 13
Nick Podratz	Chapters 8 and 21
Nikolai Ruhe	Chapters 4 and 8
Noam	Chapter 26
noor	Chapter 18
Oleg Danu	Chapter 17
orccrusher99	Chapter 25
pableiros	Chapters 5 and 6
Palle	Chapters 32, 37, 38 and 59
Panda	Chapter 4
Paulw11	Chapter 13
pixatlazaki	Chapter 4
Rahul	Chapter 17
Rick Pasveer	Chapter 7
Rob	Chapter 32
Rob Napier	Chapter 9
Ronald Martin	Chapter 7
RubberDucky4444	Chapter 51
Ryan H.	Chapters 20 and 38
saagarjha	Chapters 6, 11 and 18
Sagar Thummar	Chapter 48
Sajjon	Chapter 27
Santa Claus	Chapters 2, 12, 30 and 32
sasquatch	Chapter 11
sdasdadas	Chapter 4
SeanRobinson159	Chapters 8 and 17
Seyyed Parsa Neshaei	Chapter 1
shannoga	Chapters 8 and 13
Shijing Lv	Chapter 34
shim	Chapter 4
SKOOP	Chapters 13 and 17
solidcell	Chapter 4
Sơn Đỗ Đình Thy	Chapter 52
SteBra	Chapter 47
Steve Moser	Chapter 10
Suneet Tipirneni	Chapters 11 and 22
Sunil Prajapati	Chapter 1
Sunil Sharma	Chapter 4
Suragch	Chapters 3 and 4
Tanner	Chapters 1 and 13
taylor swift	Chapter 4

ThaNerd	第14章
胸廓	第52章
ThrowingSpoon	第6、11、12、18和39章
蒂莫西·拉舍尔	第29章
tktsubota	第2、7、13、14和20章
汤姆·马格努森	第15章
tomahh	第6章
托米·C.	第47章和第54章
torinpitchers	第14章
翁贝托·雷蒙迪	第26章
撤销	第17章
user3480295	第4和8章
user5389107	第4章
vacawama	第6章
维克多·西格勒	第4章、第17章和第40章
维克多·加达尔特	第46章
维努普里亚·阿里瓦扎甘	第4章和第30章
弗拉基米尔·努尔	第6章
WMios	第4章
xoudini	第16章
扎克	第20章
zaph	第55章、第56章和第57章
ZGski	第3章和第18章
□ R N	第1章

ThaNerd	Chapter 14
Thorax	Chapter 52
ThrowingSpoon	Chapters 6, 11, 12, 18 and 39
Timothy Rascher	Chapter 29
tktsubota	Chapters 2, 7, 13, 14 and 20
Tom Magnusson	Chapter 15
tomahh	Chapter 6
Tommie C.	Chapters 47 and 54
torinpitchers	Chapter 14
Umberto Raimondi	Chapter 26
Undo	Chapter 17
user3480295	Chapters 4 and 8
user5389107	Chapter 4
vacawama	Chapter 6
Victor Sigler	Chapters 4, 17 and 40
Viktor Gardart	Chapter 46
Vinupriya Arivazhagan	Chapters 4 and 30
Vladimir Nul	Chapter 6
WMios	Chapter 4
xoudini	Chapter 16
Zack	Chapter 20
zaph	Chapters 55, 56 and 57
ZGski	Chapters 3 and 18
□□R□□□□N	Chapter 1

你可能也喜欢



You may also like