Swift简介

简介

1. Swift语言由苹果公司在2014年推出，用来撰写osx 和 iOS应用程序
2. 2014年，在Apple WWDC发布
3. 今年的秋季发布会 9月13日

历史

1. 2010 年 7 月LLVM 编译器的原作者暨苹果 开发者工具部门总监 [克里斯·拉特纳]（Chris Lattner）开始着手 Swift 编程语言的工作，还有一个 dogfooding 团队大力参与其中
2. 用了一年时间，完成基本架构，Appl开始组建一个小组
3. Swift大约经历4年开发，2014年6月发布1.0
4. 2015年06月09日，苹果通过WWDC大会上宣布Swift开源 2.0

版本

正式版Swift 4.2 Xcode9.4

优点

1. 苹果宣称Swift的特别是：快速，现化，安全，互动，而且明显优于Objective-C语言。
2. 可以使用现有的Cocoa和Cocoa Touch框架
3. Swift取消了Objective-C的指针及其他不安全访问的使用。
4. 舍弃了Objective-C早期的SmallTalk的语法，全面改为句点表示法。
5. 提供了类似Java的名字空间(namespace)、泛型(generic)、运算对象重载(operator owerloading)。
6. Swift被简单的形容为“没有C的Objective-C”(Objective-C without the C)。

大事件

1. 2014年6月发布Swift 1.0
2. 2015年 苹果同时推出Xcode6.2 Beta5 和 b.3Beta,在完善Swift1.1的同时，推出了Swift1.2测试版
3. 2015年6月2.0版本更新，10月2.1版本
4. 2016年3月 2.2版本
5. 2016年9月 3.0版本 10月3.0.1版本
6. 2017年3月 3.1版本
7. 2017年9月 4.0版本
8. 2018年3月 4.1版本 6月 4.2版本。

现状

- 苹果带头，文档只出swift
- 目前国内有些公司的新项目已经直接采用Swift开发
- 目前很多公司都在做Swift的人才储备

学习资源

- 苹果官方博客:https://developer.apple.com/swift/blog
- 苹果官方swift电子书:https://docs.swift.org/swift-book/LanguageGuide/TheBasics.html
- 实体书:

第一部分：起步

第一章: 常量和变量

变量和常量有数据类型。类型描述数据的本质，并为编译器提供数据处理方式的信息。根据 常量或变量的类型，编译器知道该保留多少内存，并且进行类型检查（ type checking）

1.1 常量

1. 指定一个常量名称和指定类型的值关联起来。常量的值不可以修改和改变。

2. 使用let 进行声明

```
let numberOfLights = "10"   //声明一个num为10的常量
```

1.2 变量

1. 指定一个变量名称和指定类型的值关联起来。变量的值可以进行修改和改变。
2. 使用var进行声明

```
var num = 10    //声明一个num为10的变量
num = 20        //将20赋值给num
```

一行中声明多个变量和常量,用逗号隔开

```
var x = 0.0,y = 0.0,z = 1
```

类型标注

1. 声明常量或者变量的时候，可以给指定类型

```
var name: String
name = "hello"
```

1.3 插值输出

1. 不能包含数学符号，箭头，保留的（或者非法的）关键字，连线与制表符
2. 也不能以数字开头

输出常量和变量

```
let 🐷 = "pig"
let 猪 = "pig"
print("猪的英文\(猪)")
```

计算

```
//: 自动推导:  根据右侧的数值推导数据类型
var x = 30
let y = 30
let z = x + y

//: swift是一个类型特别严格的语言，任何情况下都不会做隐式转换，两个变量要进行计算，必须是相同的类型
//推荐 option+click查看变量类型
```

1.4 练习

如果新增一个表示小镇出生率的变量，你会使用什么类型？给这个变量设置一个值，并更新print来使用这个值

第二部分: 基础知识

第二章: 条件语句

2.1 if/else

根据特定的逻辑条件执行代码,通常是"非此即彼"的关系。

```
var message: String

if people < 10000 {
    message = "\(people) is a small town"
}else{
    message = "\(people) is a big town"
}
```

比较运算符:

```
<   计算左边的值是否小于右边的值
<=   计算左边的值是否小于等于右边的值
> 计算左边的值是否大于右边的值
= 计算左边的值是否大于等于右边的值
== 计算左边的值是否等于右边的值
!= 计算左边的值是否不等于右边的值
=== 计算两个实例是否指向同一个引用
!== 计算两个实例是否不指向同一个引用
```

```
var hasPostOffice: Bool = true  //定一个变量表示邮局
if !hasPostOffice {
    print("no has postOffice")
}else{
   print("hasPostOffice \(hasPostOffice)")
}
```

! 是逻辑运算符，称为：“逻辑非”,取反操作。 真变为假 ，假变为真

逻辑运算符：

```
&& 逻辑与：当且仅当两者都为真时结果为真（否则结果为假）
|| 逻辑或：两者任意之一为真时结果为真（只有两者都为假时结果为假）
! 逻辑非：真变为假，假变为真
```

2.2 三目运算

```
和if/else很像，但是更加简洁: a?b:c .  三目运算可以这么念：a是否为真?  为真则执行b，否则执行c
```

```
message = hasPostOffice ? "有邮箱" : "没有邮箱"
print(message)
```

2.3 嵌套if

```
if/else语句写在另一个if/else语句的花括号里来实现。
```

```
if people < 10 {
     message = "\(people) is a small town"
}else{
    if people > 50 {
         message = "\(people) is a big town"
    }
}
```
注意：嵌套很常用，但是不建议嵌套太深,否则逻辑会混乱不清

2.4 else if

`else if`条件语句可以让你把多个条件语句串起来。它能让你检查多种情况，并根据哪个分

支为真来执行代码

```
if people < 10 {
     message = "\(people) is a small town"
}else if(people > 50){
    message = "\(people) is a big town"
}
```

2.5 练习

给判断小镇规模的代码再增加一个`else if`的判断，用来判断小镇人口规模是否非常大。 自己选择一个类型，相应的设置`message`变量

第三章：数

3.1 整数

Int8	Int16	Int32	Int64
UInt8	UInt16	UInt32	UInt64

使用时，我们只使用int代表有符号整数(既能表示正数又能表示负数)，UInt代表无符号整数(大于等于0的整数)，swift编译器会根据对应的目标平台的硬件自动转换为int或uint转换成对应的整数类型。

```
print("int max value is \(Int.max) \nint min value is \(Int.min)")
print("int max value is \(Int32.max) \nint min value is \(Int32.min)")
```

数值型字面量

> 整数可以被写作：

- 一个*十进制*数，没有前缀
- 一个*二进制*数，前缀是`0b`
- 一个*八进制*数，前缀是`0o`
- 一个*十六进制*数，前缀是`0x`

```
var intNum = 10                     //十进制10
var intNum1 = 0b0010                //二进制2
var intNum2 = 0o0010                //八进制8
var intNum3 = 0x0010                //十六进制16
```

类型别名

> 就是给现在的类型重新定义一个别名，使用 typealias

```
typealias cusInt = Int      //重新定义int的别名为 cusInt

var cus: cusInt = 10        //初始化一个变量cus 指定为cusInt类型
```

3.2 创建整数实例

```
let number1: Int = 10 //显式声明类型
let number2 = 10    //还是int类型，编译器自动推断出来的
let number3: uint = -1  //无符号类型，只能大于等0
```

3.3 浮点型

Swift有两种基本的浮点数类型： 32位浮点数Float和64位浮点数Double

Double:

- `Double`表示64位浮点数。当你需要存储很大或者很高精度的浮点数时请使用此类型。

Float:

- `Float`表示32位浮点数。精度要求不高的话可以使用此类型。

```
var one: Float = 1/3
var two: Double = 1/3
```

`Double`精确度很高，至少有15位数字，而`Float`只有6位数字,一般优先使用Double

浮点数操作

```
let d1 = 1.1 //隐式double声明
let d2: Double = 1.1
let f1: Float = 100.3
if d1 + 0.1 == 1.2 {
    print("== 1.2")
}else{
    print("计算机会存储一个非常接近1.2的近似值,\(d1+0.1)")
}
```

```
注意：永远不要使用浮点数直接进行运算
```

3.4 练习

```
-1的8位有符号二进制表示多少?
如果还是这个二进制串，但把它解释为一个8位无符号整数，值是多少?
```

第四章：switch语句

4.1 switch

if/else语句会根据我们关注的条件是否为真来执行代码。与之对应的是， switch语句关注 某个特殊的值，并将其与分支（ case）进行匹配；如果能匹配上， switch就会执行对应的 代码。

简单的逻辑判断：

```
let a: Int = 6
if a == 1 {
    print(a);
} else if a==2 {
    print(a);
}else if a==3 {
```

```
    print(a);
}else if a==4 {
    print(a);
}else if a==5 {
    print(a);
}else if a==6 {
    print(a);
}
```

使用switch替换：

```
switch a {
case 0:
     print(a);
case 1:
    print(a);
case 2:
    print(a);
case 3:
    print(a);
case 4:
    print(a);
    fallthrough
case 5:
    print(a);
default:
    print(a);
}
```

- case后面在swift中即使没有break也不会继续向下执行。 如果想要继续向下执行，使用

  ```
  fallthrough
  ```

- Swift中的switch语句不仅仅可以对数值型进行对比。

4.2 使用switch

```
var code = 404
var errorString: String
switch code {
case 400:
    errorString = "Bad request"
case 404:
    errorString = "Not found"
default:
    errorString = "None"
}
```

4.3 访问元祖

```
let firstCode = 404
let secondCode = 200
let errorCodes = (firstCode,secondCode)
switch errorCodes {
case (404,404):
    print("404")
case (404,200):
    print("404,200")
default:
    print("none")
}
```

4.3 练习

查看喜爱安switch语句，控制台会打印什么内容?

```
let point = (x: 1, y: 4)
switch point {
case let q1 where (point.x > 0) && (point.y > 0):
    print("\(q1) is in quadrant 1")
case let q2 where (point.x < 0) && point.y > 0:
    print("\(q2) is in quadrant 2")
case let q3 where (point.x < 0) && point.y < 0:
    print("\(q3) is in quadrant 3")
case let q4 where (point.x > 0) && point.y < 0:
    print("\(q4) is in quadrant 4")
case (_, 0):
    print("\(point) sits on the x-axis")
case (0, _):
    print("\(point) sits on the y-axis")
```

```
default:
    print("Case not covered.")
}
```

第五章 循环

5.1 for-in 循环

你可以使用 `for-in` 循环来遍历一个集合中的所有元素

```
let names = ["小王","小红","张三","里斯"]
```

也可以通过遍历元祖访问每一个元祖

```
let newInfo = [("小王",20),("小红",20),("张三",15),("里斯",19)]
for item in newInfo{
    print(item)
}
```

遍历字典获取每个字典的value

```
let dict = ["name":"小王","age":"20"]
for item in dict {
    print(item.value)
}
```

`for-in` 循环还可以使用数字范围

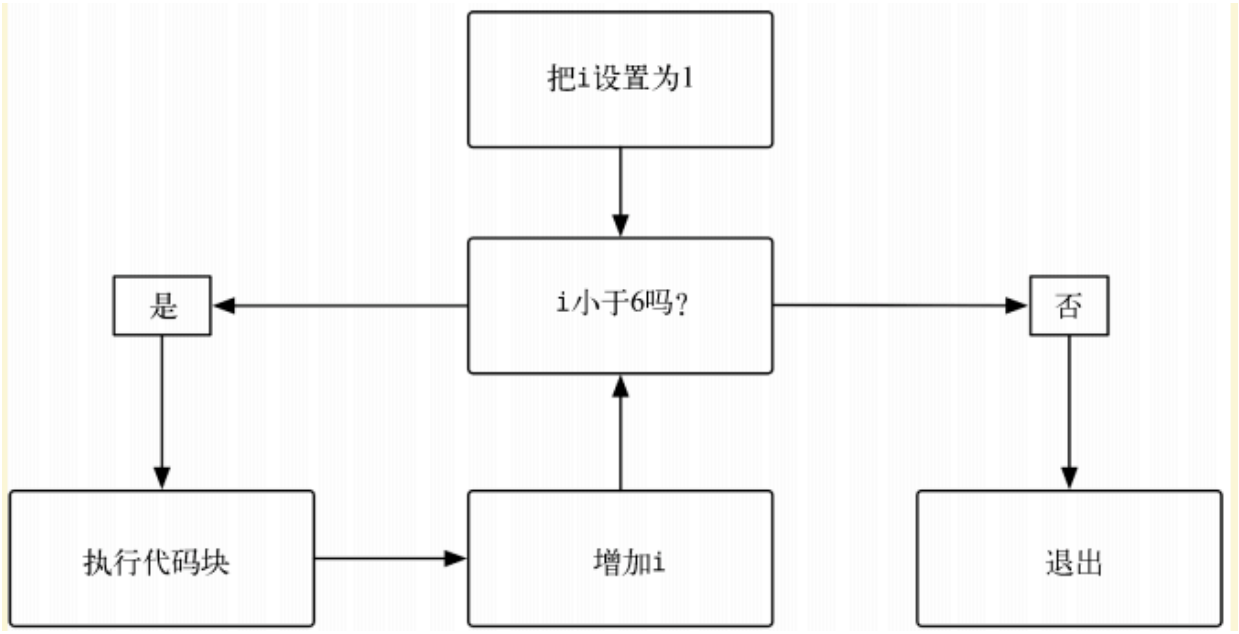
```
let newInfo = [("小王",20),("小红",20),("张三",15),("里斯",19)]
for item in newInfo[0...2] {
    print(item)
}
for index in 1...5 {
    print(index)
}
//自动推断类型
for index: Int  in 1..<5 {
    print(index)
}
```

5.2 While 循环

`while`循环会一直运行一段语句直到条件变成`false`。这类循环适合使用在第一次迭代前，迭代次数未知的情况下。Swift 提供两种`while`循环形式：

- `while`循环，每次在循环开始时计算条件是否符合；
- repeat-while循环，每次在循环结束时计算条件是否符合

```
var number: Int = 1
while number < 6 {
    number += 1
}
print(number)
//while循环最适用于循环的重复次数未知的情况。
```




```
var number: Int = 1
repeat{
    number += 1
} while number < 10
print(number)
```

5.3 重提控制转移语句

在switch中我们使用了fallthrough改变了代码的执行顺序，在循环语句中，也同样可以控制代码的执行顺序

```
//开枪游戏，默认能量保护罩为5层，武器系统有过热属性，当开枪次数大于100次时，就会过热，过热需要5秒的冷却时间，5秒后重置冷却。 当摧毁敌人500个时，游戏结束
```

```
var shields = 5                    //能量保护罩
var isHot = false                  //是否过热
var fireCount = 0                  //开枪次数
var destoryCount = 0                  //击毁个数
while shields > 0 {
    if destoryCount == 500 {
        print("game over!")
        break
    }
    if isHot {
        print("过热保护，冷却5秒")
        sleep(5)
        print("准备开火")
        sleep(1)
        isHot = false
        fireCount = 0
        continue
    }
    if fireCount > 100 {  //开枪次数大于100次，武器过热
        isHot = true
        continue
    }
    // 开枪
    print("Fire blasters!")
    fireCount += 1
    destoryCount += 1
}
```

元祖

元组（tuples）把多个值组合成一个复合值。元组内的值可以是任意类型，并不要求是相同类型。

```
let cusTup = ("张三",14,false)
cusTup.0
cusTup.1
cusTup.2
//注意：如果使用的数据结构不是临时性的，请选择类或者结构体
```

5.4 练习

一个除法练习游戏，对于给定区间内的每个值，如
果当前的数字能被3整除，就打印FIZZ。如果能被5整出，就打印BUZZ。如果能同时被3和5整除，
就打印 FIZZ BUZZ。如果不能被3也不能被5整除，就直接打印这个数字。

举个栗子，对于1~10的区间，玩Fizz Buzz游戏会得到这样的结果： 1，2，FIZZ，4，BUZZ，
FIZZ，
7，8，FIZZ，BUZZ.

在0~100的区间内进行循环，正确地为区间内的每个数字打印FIZZ、 BUZZ或者FIZZ BUZZ

第六章 可选类型

6.1 可选类型

可选类型让Swift语言更加安全。 一个可能为nil的实例应该被声明为可选类型。

使用可选类型(optionals)来处理值可以能缺失的情况

- 有值等于x
- 或者 没有值

//注意：C 和 Objective-C 以及java中并没有可选类型这个概念

Swift 的 `Int` 类型有一种构造器，作用是将一个 `String` 值转换成一个 `Int` 值。然而，并不是所有的字符串都可以转换成一个整数。字符串 `"123"` 可以被转换成数字 `123` ，但是字符串 `"hello, world"` 不行。

```
let possibleNumber = "123"
let convertedNumber = Int(possibleNumber)
```

```
let numString = nil
let numStrings: String? = nil
```

> 注意： `nil` 不能用于非可选的常量和变量。如果你的代码中有常量或者变量需要处理值缺失的情况，请把它们声明成对应的可选类型。

如果你声明一个可选常量或者变量但是没有赋值，它们会自动被设置为 `nil` ：

```
let numStrings: String?
```

> 注意：Swift的nil和Oc的nil并不一样，在oc中，nil是一个指向不存在对象的指针。 在Swift中，nill不是指针，它是一个确定的值，用来表示值缺失。任何类型的可选状态都可以为nil，不只是对象类型

```
//强制解析 !
var errorCoding: String?
errorCoding = "404"      ////如果没有赋值，打印会怎样?
if errorCoding != nil {  //增加条件语句
    let theError = errorCoding!
    print(theError)
}
```

6.2 可选绑定

可选绑定（optional binding）是一种固定模式，对于判断可选实例很有用。

如果有值，就赋值给一个临时的常量或者变量，并且使这个常量或变量在条件语句的第一个分支代码中可用。

```
if let tempValue = optional{
    //可以使用 tempValue临时变量
} else{
    //没有值，也就是说optional为nill
}
```

刚刚的列子可以这样改写:

```
if let tempValue = Int(numString) {
    print("临时常量=\(tempValue)")
}else{
    print("空值")
}
```

6.3 隐式解析可选类型

隐式解析可选类型和普通的可选类型类似，只不过它们不需要展开。

```
let string1: String? = "this is a string"
let string2: String = possibleString! // 需要感叹号来获取值


let string1: String! = "this is a string"
let string2: String = string1 // 不需要感叹号来获取值
```

6.4 练习

可选类型最好用来表示本来就可以为空的概念，不过为空不等同于零。比如，如果为银行账户建模，而用户给定账户没有余额，那么值0是不是比nil更合理?
看看以下例子，用什么类型最合适
1.一个人拥有的孩子数量： Int or Int?
2.一个人饲养的宠物名字： string or String?

第七章 数组

数组是值的有序集合。数组的每个位置都用索引标记。不像oc，Swift的Array类型可以支持任何类型的值:对象和非对象都可以。

7.1 创建数组

首先创建一个表示目标清单的数组。声明一个数组:

```
var bucketList: Array<String> = Array()
//换一种方式
var bucketList: [String] = [String]()
```

初始化数组

```
var bucketList: [String] = ["swim"]
//使用swift的推断特性
var bucketList = ["swim"]
```

7.2 访问和修改数组

```
bucketList.append("Fly")
bucketList.append("drive airport")
bucketList.insert("studySwift", at: 0)
bucketList.remove(at: 2)
```

获取数组个数

```
bucketList.count
```

使用下标

```
print(bucketList[0])
print(bucketList[0...2]) //区间下标打印出来是个数组
```

用循环从一个数组添加元素到另一个数组

```
var newItems = ["看电影","看书","打篮球"]
for item in newItems {
    bucketList.append(item)
}
```

7.3 数组相等

使用"=="号可以判断两个数组是否相等。

猜一下下面会打印什么?

```
var newItems = ["看电影","看书","打篮球"]
var otherItems = ["看书","看电影","打篮球"]

//判断两个数组是否相等
if otherItems == newItems {
    print("相等")
}else{
    print("不相等")
}
```

7.4 不可变数组

假设我们要记录用户每天吃的水果，生成一个报告。那么我们要把记录放在一个不可变数组中，毕竟吃过的水果是不可能再去修改的。

```
let eatFood = ["苹果","香蕉","哈密瓜","西瓜"]
//使用let 创建不可变数组，那么在试图改变它的时候编译器会报错
```

7.5 练习

1.var toDoList = ["Take", "Pay ","bills", "Cross off finished items"]

观察上面的数组，有一个Array类型的变量会告诉你toDoList是否有元素，利用文档找到它

2.把上面的数组写在playground中，用循环反转这个数组的元素顺序，然后把结果输出到控制台。

第八章 字典

上一章熟悉了Array类型，Array类型中的元素是有顺序的，每一个元素都有对应的一个索引，可以根据索引获取对应的元素。 但是顺序不总是很重要，有时候我们只想在容器中持有一组数据，在需要的时候获取信息，这就是字典(dictionary).

Dictonary使用键值对组织其内容。 字典的键映射到值。键在字典中是唯一的，不可重复的。

8.1 创建字典

```
var dict: Dictionary<String,Int> = [:]
var dict2 = Dictionary<String,Int>()
var dict3: [String:Int] = [:]
var dict4 = [String:Int]()
```

这四种方法得到的同样的结果。

[:]和（）语法有什么区别呢？ 本质上是一样的。 [:]使用字面量语法创建Dictionary类型的空实例。（）语法则使用Dictionary类型的默认初始化方法，这个方法会准备一个空的字典实例。

8.2 填充字典

利用字面量初始化一个字典

```
var person = ["name":"张三","班级":"高一8班","年龄":"20"]
```

8.3 访问和修改字典

根据key获取值

```
let name = person["name"]
//这里要注意一下name的类型，是可选类型，为什么?
```

修改字典中的值

```
person["age"] = "21"
let age = person["age"]
```

还有一种修改方式

```
person.updateValue("22", forKey: "age") //注意此方法中的返回值，为什么要返回这样的类型?
```

增加和删除

```
person["heigh"] = "180"
print(person)
```

两种删除方式

```
person.remove(at: person.index(forKey: "name")!)
person.removeValue(forKey: "name")
print(person)
```

8.4 练习

创建一个表示中国某省的字典，键表示省名（创建3个就行）,每个键会映射到一个数组，数组持有该省的5个邮编。最后输出字典的邮编类似下面:

```
["210000", "210002", "210004", "210008"]
```

第九章 集合

Set集合是Swift提供的第三种容器类型，Set不是很常用。

9.1 什么是集合

```
集合是一组互不相同的实例的无序组合。 不相同、无序。比如数组可以有[2,2,2,2,] 但是集合不行。
```

集合和字典有很多相似的地方，和字典一样，都是无序的。 字典的键必须唯一，集合也不允许有重复的值。

容器类型	有序	唯一	存储
数组	是	否	元素
字典	否	键	键值对
集合	否	元素	元素

9.2 创建集合

```
var sets = Set<String>()
var sets2: Set<String> = Set()
var sets3: Set = ["1","2"]
```

9.3 集合的操作

插入操作

```
sets2.insert("insertString")
sets2.update(with: "insert")
print(sets2)
//比较一下这两个方法有什么不同?
```

删除操作

```
sets2.remove("insert")
sets2.remove(at: sets2.index(of: "insertString")!)
```

9.4运用集合

想象一下，如果你在超市购物，遇到了你的朋友，他买了火锅底料，而你买了火锅食材，你们打算合并起来一起吃个火锅。

```
var food: Set = ["羊肉卷","牛肉卷","撒尿牛丸","雪花肥牛","精品肥牛","抖音网红面筋包","一米肥牛"]
var base: Set = ["麻辣牛油锅底","番茄西红柿锅底","三鲜菌菇锅底"]
let HotPot: Set = food.union(base)
print(HotPot)
```

如果在吃火锅的过程中，你又一个朋友来了，他手里也拿着食材，他因为忌口，所以只能吃你们共同的食材。

```
var food: Set = ["羊肉卷","牛肉卷","撒尿牛丸","雪花肥牛","精品肥牛","抖音网红面筋包","一米肥牛"]
var base: Set = ["麻辣牛油锅底","番茄西红柿锅底","三鲜菌菇锅底"]
var otherFood: Set = ["撒尿牛丸","牛肉卷","菠菜","金针菇","海带","老油条"]
var newFooed: Set = food.intersection(otherFood)
print(newFooed)
```

判断是否有相同元素

```
var food: Set = ["羊肉卷","牛肉卷","撒尿牛丸","雪花肥牛","精品肥牛","抖音网红面筋包","一米肥牛"]
var base: Set = ["麻辣牛油锅底","番茄西红柿锅底","三鲜菌菇锅底"]
var otherFood: Set = ["撒尿牛丸","牛肉卷","菠菜","金针菇","海带","老油条"]
var newFooed = otherFood.isDisjoint(with: food)
```

9.5 练习

观察以下代码，这段代码模拟了两个人去过的城市

```
let myCities = Set(["nanjing", "wuxi",
                    "changzhou", "shanghai", "beijing"])
let yourCities = Set(["nanjing", "shanghai", "beijing"])
```

找到set的一种方法，根据myCities是否包含所有yourCities中的城市来返回bool。提 示：这种关系使得myCities成为yourCities的超集 （ superset） 。

第十章 函数

函数(function)是一组有名字的代码，用来完成某个特定的任务。

10.1 一个基本的函数

```
func printGreeting(){
    print("Hello world")
}
printGreeting()
```

10.2 函数参数

```
//有一个参数 没有返回值的函数
func printPersonName(name: String){ //形参
    print("hello \(name)")
}
printPersonName(name: "张三")
```

printPersonName(name: String) 接受一个String类型的形参。 实参是调用者传递给函数形参的值。

如果你不想在调用函数时把形参暴露，你可以定义‘参数标签’

参数标签

```
func printPersonName2(yourName name: String){
    print("name is \(name)")
}
```

默认情况下函数使用参数名来作为它们的参数标签。如果你不希望为某个参数添加标签，你可以使用下划线来代替

```
//忽略参数标签
func printPersonName2(_ name: String, names2: String){
    print("name is \(name)")
}
```

如果我们的参数是不固定的，长度可变的，那么我们可以定义可变参数

10.3 可变参数

一个可变参数 可以接受0个或多个值。通过在变量类型后面加入(…)的方式来定义可变参数。

```
func printHelloToNames(names: String...){
    for name in names {
        print("hello\(name)")
    }
}
printHelloToNames(names: "张三","里斯","王五")
//可叫names的[String]类型的数组常量
```

> 注意：一个函数最多只能有一个可变参数

10.4 输入输出参数

> 函数参数默认是常量。 如果试图修改参数值会导致编译出错。那么就应该把这个参数定义为 ‘输入输出参数(In-Out Parameters)’

```
var error = "The request failed"
func appendErrorCode(code: Int, errorString: inout String ){
    if code == 400 {
        errorString += "bad request"
    }
    print(errorString)
}
appendErrorCode(code: 400, errorString: &error)
//&引用传递
```

10.5 函数类型

> 每个函数都有特定的 ‘函数类型’，函数类型是由 函数的参数类型和返回类型组成。

例如:

```
func addTwoInts(_ a: Int, _ b: Int) -> Int {
    return a + b
}
func multiplyTwoInts(_ a: Int, _ b: Int) -> Int {
    return a * b
}
```

下面的列子没有参数，也没有返回值

```
func printHelloWorld() {
    print("hello, world")
}
```

10.6 使用函数类型

> 在Swift中，使用函数类型就像使用其他类型一样，你可以定义一个类型为函数的常量或变量，并将适当的函数赋值给它:

```
func addTwoInts(a: Int, b: Int) -> Int{
    return a + b
}
var mathFunction: (Int, Int) -> Int = addTwoInts
```

> 这段代码可以解读为: "定义一个叫mathFunction 的变量，类型是 ‘一个有两个int型的参数并返回一个Int型的值的函数’，并让这个新变量指向addTwoInts函数"

使用:

```
print("结果: \(mathFunction(2,3))")
```

除了上面这种显式的写法，我们可以利用Swift的自动推断

```
let addMath = addTwoInts
//option+click 看看 addMath 的类型
```

10.7 函数类型作为参数

> 你可以使用 (Int, Int) -> Int 这样的函数类型作为另一个函数的参数类型。

例如:

```
func addTwoInts(a: Int, b: Int) -> Int{
    return a + b
}
let addMath = addTwoInts

func printMathResult(mathFuncation: (Int, Int) -> Int, a: Int, b: Int) ->
Void{
    print("Result \(mathFuncation(a,b))")
}
printMathResult(mathFuncation: addMath, a: 5, b: 5)
```

10.8 函数类型作为返回类型

> 同样，你也可以用函数类型作为另一个函数的返回类型。

例如:

```
func stepForward(input: Int) -> Int{
    return input + 1
}

func stepBackward(intput: Int) -> Int{
    return intput - 1
}

func chooseStepFunction(backward: Bool) -> (Int) ->Int {
    return backward ? stepBackward : stepForward
```

```
}

var backward: Bool = true
let funcType = chooseStepFunction(backward: backward)
print("result = \(funcType(2))")
```

10.9 嵌套函数

> 目前所说的函数都是全局函数，定义在全局域中。如果把函数定义在别的函数体中，称作嵌套函数

使用嵌套函数重写上面的方法

```
func chooseStepFunction2(backward: Bool) -> (Int) -> Int{
    func stepForward(input: Int) -> Int{
        return input + 1
    }
    func stepBackward(input: Int) -> Int{
        return input - 1
    }
    return backward ? stepBackward : stepForward
}

var curValue = 4
let functionType = chooseStepFunction2(backward: curValue > 0)
while curValue > 0{
    print("curvalue = \(curValue)")
    curValue = functionType(curValue)
}
```

> 嵌套函数默认情况下是对外不可见的，但是可以被它们的 外围函数调用。一个外围函数也可以返回它的某一个嵌套函数，使得这个函数可以在其他作用域中被使用。

第十一章 闭包

> 闭包是自包含的函数代码块，可以在代码中被传递和使用。

全局和嵌套函数实际上也是一种特殊的闭包，闭包采取以下三种形式之一:

- 全局函数是一个有名字但不会捕获任何值的闭包
- 嵌套函数是一个有名字并可以捕获其他封闭函数域内值的闭包。
- 闭包表达式是一个利用轻量级语法所写的可以捕获其上下文中变量或常量值的匿名闭包

Swift的闭包表达式在目前的版本中，主要优化如下:

- 利用上下文推断参数和返回值类型

- 隐式返回单表达式闭包。即单表达式可以省略return关键字
- 参数名称缩写
- 尾随闭包语法

###

11.1 闭包表达式

> 是一种利用简洁语法构建内联闭包的方式。

sorted方法，它会根据你所提供的排序闭包进行排序。

例如:

```
let names  = ["a","b","e","f","g","o","p"]
func backward(name1: String, name2: String) -> Bool {
    return name1 > name2        //name1 大于 name2 返回true,  大的在前小的在后
}
let namesSort = names.sorted(by: backward)
```

是不是太繁琐了？利用闭包表达可以更好的构造一个内联排序闭包

11.2 闭包表达式语法

```
{(parameters) -> returnType in
//引入的代码体

}
```

```
let namesSort2 = names.sorted(by: {(a1: String, a2: String) -> Bool in
    //引入的代码体
     return  a1 > a2
})
```

根据上下文推断，优化一下

```
let namesSort3 = names.sorted(by: {(a1,a2) in

    return a1 > a2
})
```

根据隐式返回单表达式闭包，再优化一下

```
let namesSort4 = names.sorted(by: {(a1,a2) in
    a1 > a2
})
```

参数名缩写

swift自动为内联闭包提供了参数名缩写功能，可以直接通过 $0,$1,$2来顺序调用闭包的参数。

```
let namesSort5 = names.sorted(by: {$0 > $1})
print(namesSort5)
```

11.3 尾随闭包

函数的参数可以是函数类型，也可以是闭包。 如果你需要将一个很长的闭包表达式作为最后一个参数传递给函数，可以使用尾随闭包来增强函数的可读性。在使用尾随闭包时，你不用写出它的参数标签

```
func someFunctionDemo(closure: () -> Void){
    //函数体部分。
    closure()
    print("22")
}

//不使用尾随闭包进行函数调用
someFunctionDemo(closure: {
   //闭包主体部分
    print("11");
});

//使用尾随闭包
someFunctionDemo {
    //闭包主体部分
    print("11");
};
```

那么之前的姓名排序可以改写成：

```
names.sorted { (a1, a2) -> Bool in
    a1 > a2
}
//作为尾随闭包，省略了参数名
```

简单的回调应用:

```
var  arrayDatas = ["数据1","数据2"]
func refreshData(block: () -> Void){
    //请求数据
    arrayDatas.append("数据3")
    block()
}
//请求数据
refreshData {
    //显示数据
    print(arrayDatas);
};
```

11.4 值捕获

> 闭包可以在其被定义的上下文中*捕获*常量或变量。即使定义这些常量和变量的原作用域已经不存在，闭包仍然可以在闭包函数体内引用和修改这些值。

最简单的嵌套函数

```
func makeIncrementer(forIncrement amount: Int) -> () -> Int {
    var runningTotal = 0
    func incrementer() -> Int {
        runningTotal += amount
        return runningTotal
    }
    return incrementer
}

let incrementByTen = makeIncrementer(forIncrement: 10)
incrementByTen()
// 返回的值为10
incrementByTen()
// 返回的值为20
incrementByTen()
// 返回的值为30
```

我们单独看嵌套函数

```
func incrementer() -> Int {
        runningTotal += amount
        return runningTotal
    }
//该函数没有任何参数，但是在函数体内访问了runningTotal 和 amount。 因为它从外围函数捕获了 这两个变量的引用。 捕获引用保证了这两个变量在调用完makeIncrementer后不会消失，并且保证了再次执行incrementer函数时，这两个变量依旧存在。
```

11.5 闭包时引用类型

> 上面的例子中，`incrementBySeven` 和 `incrementByTen` 都是常量，但是这些常量指向的闭包仍然可以增加其捕获的变量的值。这是因为函数和闭包都是*引用类型*。

无论你将函数或闭包赋值给一个常量还是变量，你实际上都是将常量或变量的值设置为对应函数或闭包的*引用*。

第十二章 枚举

> 枚举为一组相关的值定义了一个共同的类型.在c语言中，枚举为每个类型分配一组整形值，但是Swift会更加灵活，不必为每个枚举提供一个值。
>
> 此外，枚举成员可以指定任意类型的关联值存储到枚举成员中。

12.1 枚举语法

```
enum SomeEnumeration {
    //枚举定义放在这
}
```

> 定义一个枚举

```
enum TextAligement{
    case left
    case right
    case center
}
//大写字母开头,大驼峰式写法， 变量、函数、枚举成员开头使用小写字母，小驼峰写法
```

定义枚举的方式是在关键字'enum' 后声明一个枚举的名称。花括号中是定义的枚举。枚举体中至少包含一个case语句，表示枚举的可能值；

> 枚举的使用

```
var point: TextAligement = TextAligement.left
var point = TextAligement.left //利用了自动推断
point = .center
//注意：第一次创建枚举变量时必须要指定枚举名和值，后面可以省略。
```

11.2 比较枚举值

```
if point == TextAligement.center {
    print("current is center")
}
```

使用switch

```
switch point {
case TextAligement.left:
    print("left")
case TextAligement.right:
    print("right")
case TextAligement.center:
    print("center")
default:
    print("none")
}
```

12.3 原始值枚举

> 刚刚我们写过枚举的对比，发现枚举没有关联值，一般我们java或者oc ，定义枚举时每个枚举都有一个关联的整形值，默认从0开始。

重新定义一个枚举

```
enum NewTextAligenment: Int{
    case left
    case right
    case center
}
var newPoint = NewTextAligenment.left
if newPoint.rawValue == 0 {
    print("关联值==\(newPoint.rawValue)")
}
```

> 创建带有字符串原始值的枚举

```
enum NewTextAligenment: String{
    case left  = "居左"
    case right = "居右"
    case center = "居中"
}

var newPoint = NewTextAligenment.left
if newPoint.rawValue == "居左" {
    print("关联值==\(newPoint.rawValue)")
}
```

12.4 方法

方法是和类型关联的函数。 有些语言中(java,oc等)方法只能和类关联，但是在swift中，方法可以和枚举关联。

创建一个新的枚举代表电灯泡的状态：

```
enum LightStatus {
    case on
    case off
    func surfaceTemp(temp: Double) -> Double{
        switch self {
        case .on:
            return 150 + temp
        case .off:
            return  temp
        }
    }
}
```

上面创建了一个枚举

```
var status = LightStatus.on //创建一个实例
let temp = 77.0
var lightTemp = status.surfaceTemp(temp: temp) //调用实例方法
print("current LightTemp = \(lightTemp)")
```

12.5 关联值

> 到目前为止，用枚举做的事情都是一类：定义一些静态的成员值来枚举可能的值或状态。Swift还提供了一种强大的枚举：带关联值的成员。

- 关联值能让你把数据附在枚举实例上；
- 不同的成员可以有不同类型的关联值

例：定义一个枚举用来记录一些基本图形的尺寸。每种图形有不同的属性。

```
enum Shape {
    //正方形
    case square(side: Double)
    //长方形 长和宽
    case rectangle(width: Double, length: Double)
}

//定义一个边长为10的正方形
var shap: Shape = Shape.square(side: 10)
//定义一个行和宽的长方形
var rect = Shape.rectangle(width: 5, length: 10)
```

利用关联值计算面积

```
//在枚举中定义一个方法
 func area() -> Double{
        switch self {
        case .square(side: let value):
            return value * value
        case .rectangle(width: let width, length: let length):
            return width * length

        }
    }
```

```
//定义一个边长为10的正方形
var shap: Shape = Shape.square(side: 10)
print("正方形面积:\(shap.area())")
//定义一个行和宽的长方形
var rect = Shape.rectangle(width: 5, length: 10)
print("长方形面积:\(rect.area())")
```

12.6 练习

> 定义一个Car枚举,包含独轮车、自行车、电动车、小汽车、4轮卡车。

1.定义每种车的关联值。

2.在枚举中写一个方法，返回每种类型的轮子数量。

第十三章 结构体和类

> 结构体和类是构建应用的支柱。对象是类的实例，类是实例的抽象。

13.1 类和结构体对比

> 共同点：

- 都有属性
- 都有方法
- 定义下标操作来访问实例所包含的值
- 都有构造器用来初始化
- 通过扩展来增加功能
- 都可以实现协议提供某种标准功能

但是 与结构体相比，类还有如下的附加功能：

1. 继承,允许一个类继承另一个类的特征
2. 类型转换允许在运行时检查和解释一个类实例的类型
3. 析构器允许一个类实例释放任何其所被分配的资源
4. 引用计数允许对一个类的多次引用

> 注意：结构体总是通过被复制的方式在代码中传递，不使用引用计数。

13.2 结构体

> 结构体是把相关数据块组合在一起放在内存中的一种类型。

例如：创建一个结构体Town来为一个有怪兽袭击的问题小镇建模。

```
struct Town {
    var people = 5422
    var numberLights = 4

}

var myTown = Town()
print("people count is \(myTown.people) and numberlights count is \
(myTown.numberLights)")
```

让Town自己描述自己

```
struct Town {
    var people = 5422
    var numberLights = 4
    func description() -> Void {
        print("people count is \(myTown.people) and numberlights count is
\(myTown.numberLights)")
    }
}
var myTown = Town()
myTown.description()
```

如果要添加一个增加人口的方法，需要修改结构体的属性

如果结构体的方法如果要修改结构体的属性，需要使用mutating来标记此方法

```
mutating func changePeople(amout: Int){
        people += amout
    }
```

```
myTown.changePeople(amout: 2)
myTown.description()
```

###

第十四章 类

14.1 定义一个类

定义一个怪兽的类，有一些通用属性

```
class Monster {
    var town: Town?
    var name = "Monster"

    func attackTown() -> Void {
        if let tempTown = town {
            print("\(name) is attack town 有\(tempTown.people)人受到攻击")
        }else{
            print("\(name) 没有找到小镇")
        }
    }
}
```

初始化实例对象

```
var monster: Monster = Monster()
monster.town = myTown
monster.attackTown()
```

14.2 继承

面向对象三大特性：

类的一个主要特性是继承，这是结构体所没有的。继承就是指由一个类（父类）定义另一个类（子类）的关系。子类继承父类的属性和方法。

创建一个僵尸类，继承通用的怪兽类

```
class Zombie: Monster {
    var walkWithLimp = true
    override func attackTown() {
        town?.changePeople(amout: -10)
        super.attackTown()
    }
}

var zombie = Zombie()
zombie.town = myTown
zombie.attackTown()
```

- 上面代码继承了父类Monster,重写了父类attackTown()方法，注意：重写父类方法必须使用'override'关键字。

- 如果子类要调用父类的方法或者属性，必须使用super关键字。

14.3 禁止重写

> 有时候我们希望子类禁止重写父类的方法，可以使用关键字‘final’让方法或属性不可重写。

```
class Zombie: Monster {
    var walkWithLimp = true
    final  override func attackTown() {
        town?.changePeople(amout: -10)
        super.attackTown()
    }
}
//这样 Zombie的子类就无法重写 attackTown()方法
```

14.4 练习

1. 现在Zombie有个bug，如果Zombie实例骚扰了一个人口为0的镇子，那么人口会降到-10，没有意义。所以请修改Zombie类型的 attachtown方法，当人口大于等于10才去骚扰。
2. 再创建一个Monster类型的子类Vampire。重写terrorizeTown()方法，使得每个Vampire 实例袭击镇子后，都会给Vampire类型的吸血鬼奴仆（vampire thrall）数组加新的一员。这个 吸血鬼奴仆数组应该是空的。骚扰的镇子应该让镇子的人口减少1人。

第十五章 属性

> 属性分为两种：

1. 存储属性
2. 计算属性

存储属性可以有默认值，计算属性则根据已有的信息计算返回计算结果。

15.1 基本的存储属性

```
var str = "Hello, playground"

1.var 表示这个属性是变量
2.str默认值是‘Hello, playground"’
3.str是存储属性，其值可以被读写. 这里为什么说是可以被读写??
```

15.2 延迟存储属性

有时候我们不能马上给存储属性赋值，第一次被调用的时候才会计算其初始值的属性。在属性声明前使用 `lazy` 来标示一个延迟存储属性。

> 为什么延迟存储属性必须定义成var类型?

举个栗子:

```
class DataImporter {
      /* DataImporter 是一个负责将外部文件中的数据导入的类。 这个类的初始化会消耗不少时间。 */
    var fileName = "data.txt"
}

class DataManager {
    lazy var importer = DataImporter()
    var data = [String]() //string数组
}
let manager = DataManager()
manager.data.append("datas")
// DataImporter 实例的 importer 属性还没有被创建

print("\(manager.importer.fileName)") //这个时候才会被创建并且调用
```

15.3 计算属性

任何自定义的类、结构体和枚举都可以使用计算属性。计算属性不会像之前的属性那样存储值，而是提供一个 *读取方法*（getter）来获取属性的值，并可选的提供一个*写入方法（setter）* 设置属性的值。

举个栗子:

定义3个结构体来描述几何形状:

- Point封装了(x,y)坐标
- Size封装了一个width和一个height
- Rect表示一个有原点和尺寸的矩形

```
struct Point{
    var x = 0.0
```

```
    var y = 0.0
}

struct Size {
    var width = 0.0
    var height = 0.0
}

struct Rect {
    var origin = Point()
    var size = Size()
    var center: Point {
        get {
            let centerX = origin.x + (size.width/2)
            let centerY = origin.y + (size.height/2)
            return Point(x:centerX, y: centerY)
        }
        set(newCenter){
            origin.x = newCenter.x - size.width/2
            origin.y = newCenter.y - size.height/2
        }
    }
}

var square = Rect(origin: Point(x:0.0, y:0.0),size:Size(width:10.0,height:
10.0))
let initSquareCenter = square.center
print("初始值:\(initSquareCenter)")

square.center = Point(x: 15.0, y: 15.0)
print("当前正方形的偏移量\(square.origin.x),\(square.origin.y)")
```

- `Point` 封装了一个 `(x, y)` 的坐标
- `Size` 封装了一个 `width` 和一个 `height`
- `Rect` 表示一个有原点和尺寸的矩形

15.4 简化setter声明

如果计算属性的setter没有定义表示新值的参数名，则可以使用默认名称`newValue`

```
struct Rect {
    var origin = Point()
    var size = Size()
    var center: Point {
        get {
            let centerX = origin.x + (size.width/2)
            let centerY = origin.y + (size.height/2)
```

```
            return Point(x:centerX, y: centerY)
        }
//        set(newCenter){
//            origin.x = newCenter.x - size.width/2
//            origin.y = newCenter.y - size.height/2
//        }
        set {
            origin.x = newValue.x - size.width/2
            origin.y = newValue.y - size.height/2
        }
    }
}
```

15.5 只读计算属性

只有getter没有setter的计算属性就是只读计算属性。只读计算属性总是返回一个值。

```
struct Square{
    var width = 0.0,height = 0.0
    var area: Double {
        return width * height
    }
}

let squares = Square(width: 10.0, height: 10.0)
print("the area is \(squares.area)")
```

15.6 属性观察器

属性观察器 可以用来 监控和响应属性值的变化，每次属性被设置值的时候都会调用属性观察器。

可以为除了延迟存储属性之外的其他存储属性添加属性观察器，也可以通过重写属性的方式为继承的属性（包括存储属性和计算属性）添加属性观察器。你不必为非重写的计算属性添加属性观察器，因为可以通过它的 setter 直接监控和响应值的变化

可以为属性添加如下的一个或全部观察器：

- `willSet` 在新的值被设置之前调用
- `didSet` 在新的值被设置之后立即调用

`willSet` 观察器会将新的属性值作为常量参数传入，在 `willSet` 的实现代码中可以为这个参数指定一个名称，如果不指定则参数仍然可用，这时使用默认名称 `newValue` 表示。

同样，`didSet` 观察器会将旧的属性值作为参数传入，可以为该参数命名或者使用默认参数名 `oldValue`。如果在 `didSet` 方法中再次对该属性赋值，那么新值会覆盖旧的值。

```
struct Square{
    var width = 0.0{
        willSet{
            print("老数据\(width),新的数据\(newValue)")
        }
        didSet{
            print("老数据\(oldValue),新数据\(width))") //在设置完数据后
        }
    }
    var height = 0.0
    var area: Double {
        return width * height
    }
}

var squares = Square(width: 10.0, height: 10.0)
squares.width = 2.0
print("the area is \(squares.area)")
```

15.7 练习

1. 定义一个小镇，镇长很忙，不必关注每次出生和搬迁。只有当人口数减少时才打印人口变化。
2. 新建一个叫作Mayor的结构体类型。Town类型应该有一个属性叫mayor,持有一个Mayor类型的实例。每次人口变化就通知monyor，如果小镇人口减少，就让mayor实例打印如下信息:"听到最近发生的这起悲剧，我深感悲痛"(提示:你需要给Mayor类型定义一个新的实例方法来完成这个练习。)

第十六章 初始化

初始化就是设置类型实例的操作，包括给每个存储属性初始值，以及一些其他准备工作。

到目前为止，我们创建类型的方法都差不多。属性要么有默认值，要么是按需计算。

控制类型实例的创建过程是常见的需求，比如，如果能让实例的属性立即得到正确的值是最好的。之前我们都是给实例的存储属性一个默认值，并在创建后修改，这种方法不够优雅。

16.1 构造器

使用关键字init表示。虽然初始化方法是类型的方法，但是前面没有func关键字。

```
struct CustomType{
    init(someValue: value) {
        //初始化代码
    }
}
```

结构体、枚举和类都是用这种通用语法，没什么区别。

16.2 默认构造器

```
struct Town {
    var people: Int = 0
    var numberLights: Int = 0
    func description() -> Void {
        print("people count is \(myTown.people) and numberlights count is
\(myTown.numberLights)")
    }
    mutating func changePeople(amout: Int){
        people += amout
    }
}
//这里的初始化方法其实是使用了Swift编译器提供的‘空初始化方法’，为实例的属性指定了默认值
var myTown = Town()
```

16.3 结构体的逐一成员构造器

除了上面提到的默认构造器，如果结构体没有提供自定义的构造器，它们将自动获得一个逐一成员构造器，即使结构体的存储型属性没有默认值。

```
var myTown2 = Town(people: 5, numberLights: 0)
```

初始化实例，并为实例属性指定默认值。

16.4 自定义初始化方法

```
struct Town {
    var people: Int
    var numberLights: Int
```

```
    init(people: Int, lights: Int) {
        self.people = people
        self.numberLights = lights

    }
    func description() -> Void {
        print("people count is \(self.people) and numberlights count is \
(self.numberLights)")
    }
    mutating func changePeople(amout: Int){
        people += amout
    }
}

var towm = Town(people: 5, lights: 5)
towm.description()
```

16.5 类的继承和构造过程

类里面的所有存储型属性——包括所有继承自父类的属性——都必须在构造过程中设置初始值。

Swift 为类类型提供了两种构造器来确保实例中所有存储型属性都能获得初始值，它们分别是*指定构造器*和*便利构造器*。

每一个类都必须拥有至少一个指定构造器。

*便利构造器*是类中比较次要的、辅助型的构造器

举个栗子:

```
//类的指定构造器的写法和值类型简单构造器一样:
init(parameters) {
    statements
}
```

```
//便利构造器需要在init关键字之前加上 convenience关键字
convenience init(parameters) {
    statements
}
```

16.6 类的构造器代理规则

为了简化指定构造器和便利构造器之间的调用关系，Swift 采用以下三条规则来限制构造器之间的代理调用：

规则 1

指定构造器必须调用其直接父类的的指定构造器。

规则 2

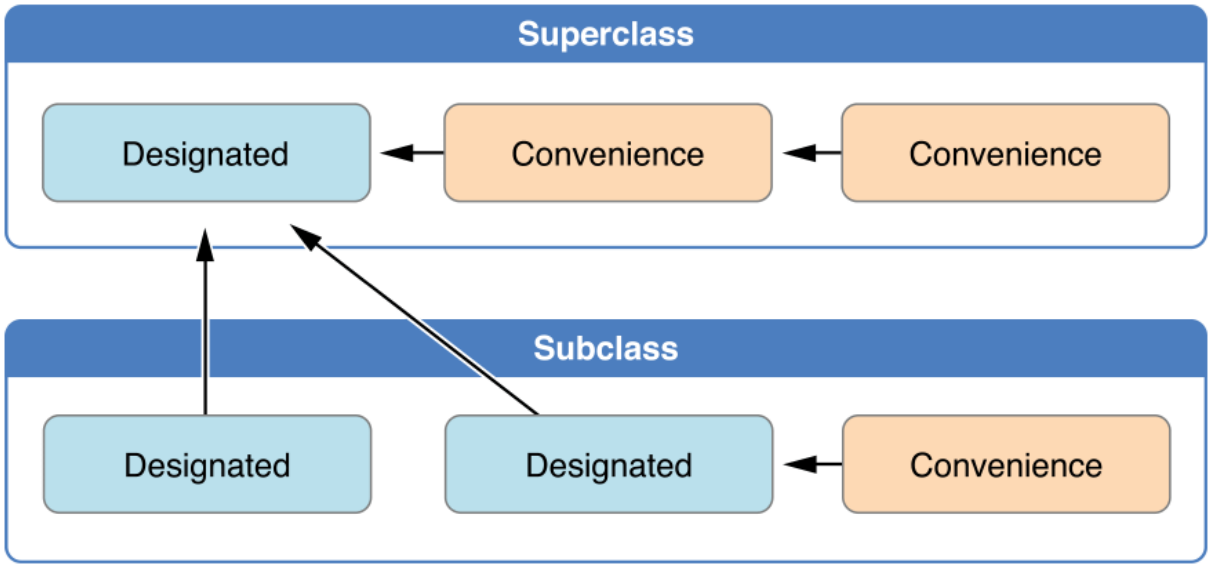
便利构造器必须调用*同*类中定义的其它构造器。

规则 3

便利构造器必须最终导致一个指定构造器被调用。

简而言之：

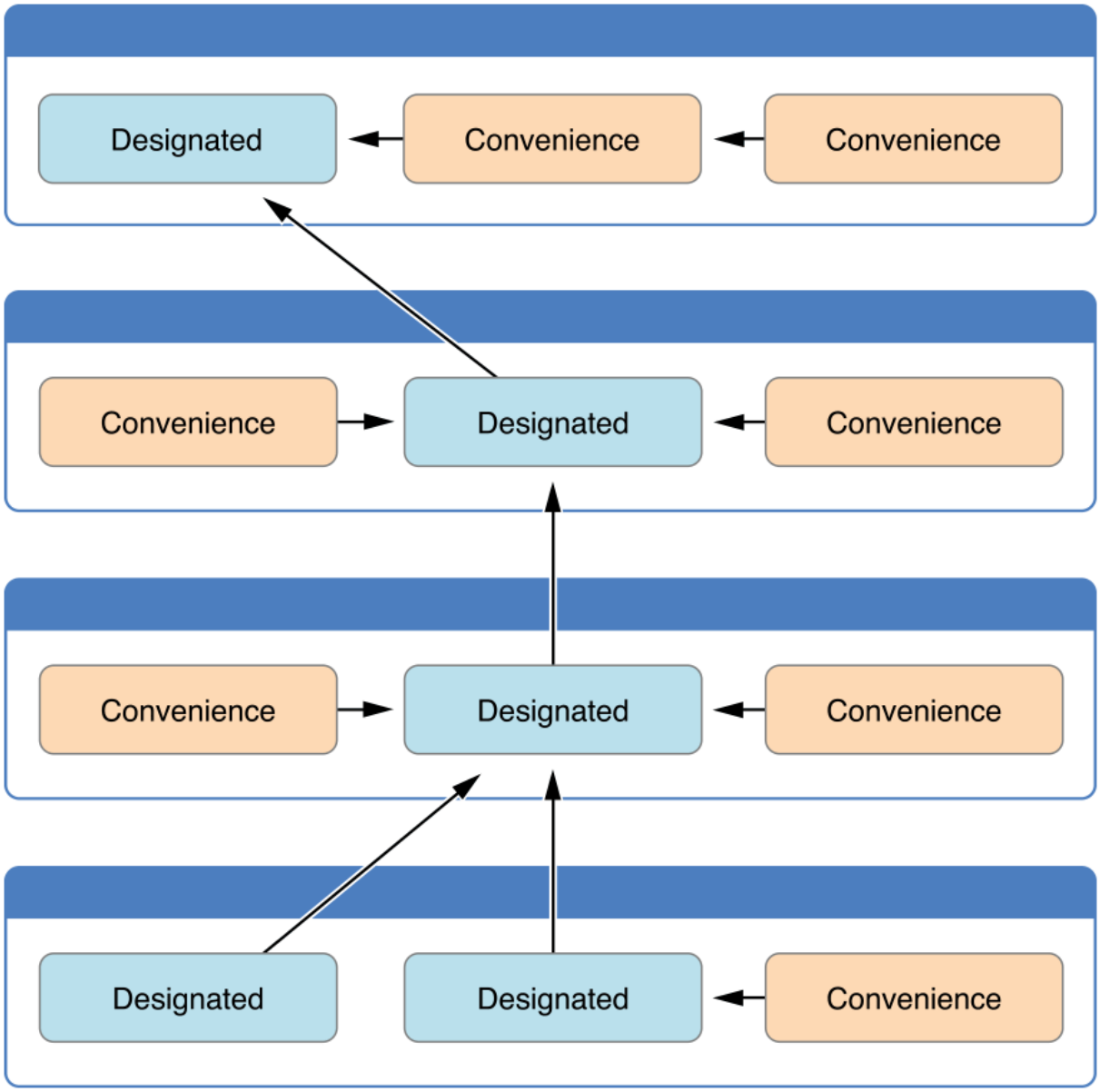
- 指定构造器*必须*总是*向上*代理
- 便利构造器*必须*总是*横向*代理



如图所示，父类包含一个指定初始化器和两个便捷初始化器。一个便捷初始化器调用另一个便捷初始化器，而后者又调用了指定初始化器。这满足了上边的规则2和规则3。父类本身没有其他父类，所以规则1不适用。

这个图中的子类有两个指定初始化器和一个便捷初始化器。便捷初始化器必须调用这两个指定初始化器之一，因为它只能从同一个类中调用初始化器。这满足了上边的规则2和规则3。两个指定初始化器又必须从父类调用一个指定初始化器，这满足了上边所说的规则1。

下图展示了四个类之间更复杂的层级结构。它演示了指定初始化器是如何在此层级结构中充当”管道”作用，在类的初始化链上简化了类之间的内部关系：



16.7 两段式初始化

Swift 的类初始化是一个两段式过程。在第一个阶段，每一个存储属性被引入类为分配了一个初始值。一旦每个存储属性的初始状态被确定，第二个阶段就开始了，每个类都有机会在新的实例准备使用之前来定制它的存储属性。

两段式初始化过程的使用让初始化更加安全，同时在每个类的层级结构给与了完备的灵活性。两段式初始化过程可以防止属性值在初始化之前被访问，还可以防止属性值被另一个初始化器意外地赋予不同的值。

安全检查 1

指定初始化器必须保证在向上委托给父类初始化器之前，其所在类引入的所有属性都要初始化完成。

如上所述，一个对象的内存只有在其所有储存型属性确定之后才能完全初始化。为了满足这一规则，指定初始化器必须保证它自己的属性在它上交委托之前先完成初始化。

安全检查 2

指定初始化器必须先向上委托父类初始化器，然后才能为继承的属性设置新值。如果不这样做，指定初始化器赋予的新值将被父类中的初始化器所覆盖。

安全检查 3

便捷初始化器必须先委托同类中的其它初始化器，然后再为任意属性赋新值（包括同类里定义的属性）。如果没这么做，便捷构初始化器赋予的新值将被自己类中其它指定初始化器所覆盖。

安全检查 4

初始化器在第一阶段初始化完成之前，不能调用任何实例方法、不能读取任何实例属性的值，也不能引用self 作为值。

直到第一阶段结束类实例才完全合法。属性只能被读取，方法也只能被调用，直到第一阶段结束的时候，这个类实例才被看做是合法的。

以下是两段初始化过程，基于上述四种检查的流程：

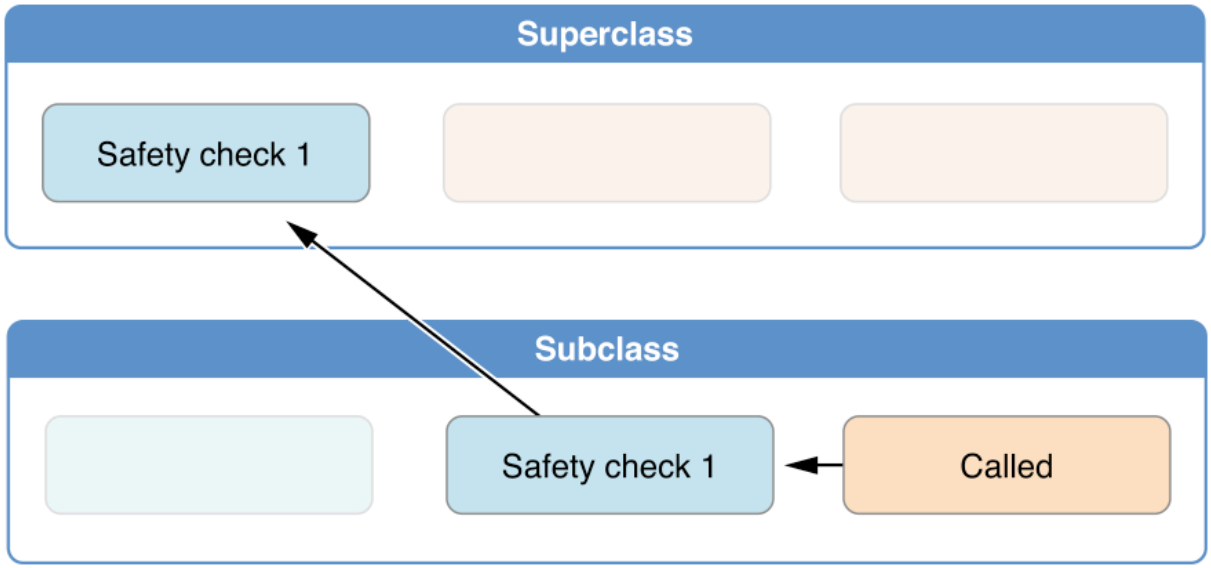
阶段 1

- 指定或便捷初始化器在类中被调用；
- 为这个类的新实例分配内存。内存还没有被初始化；
- 这个类的指定初始化器确保所有由此类引入的存储属性都有一个值。现在这些存储属性的内存被初始化了；
- 指定初始化器上交父类的初始化器为其存储属性执行相同的任务；
- 这个调用父类初始化器的过程将沿着初始化器链一直向上进行，直到到达初始化器链的最顶部；
- 一旦达了初始化器链的最顶部，在链顶部的类确保所有的存储属性都有一个值，此实例的内存被认为完全初始化了，此时第一阶段完成。

阶段 2

- 从顶部初始化器往下，链中的每一个指定初始化器都有机会进一步定制实例。初始化器现在能够访问 self 并且可以修改它的属性，调用它的实例方法等等；
- 最终，链中任何便捷初始化器都有机会定制实例以及使用 slef 。

下面展示了第一阶段假定的子类和父类之间的初始化调用关系：



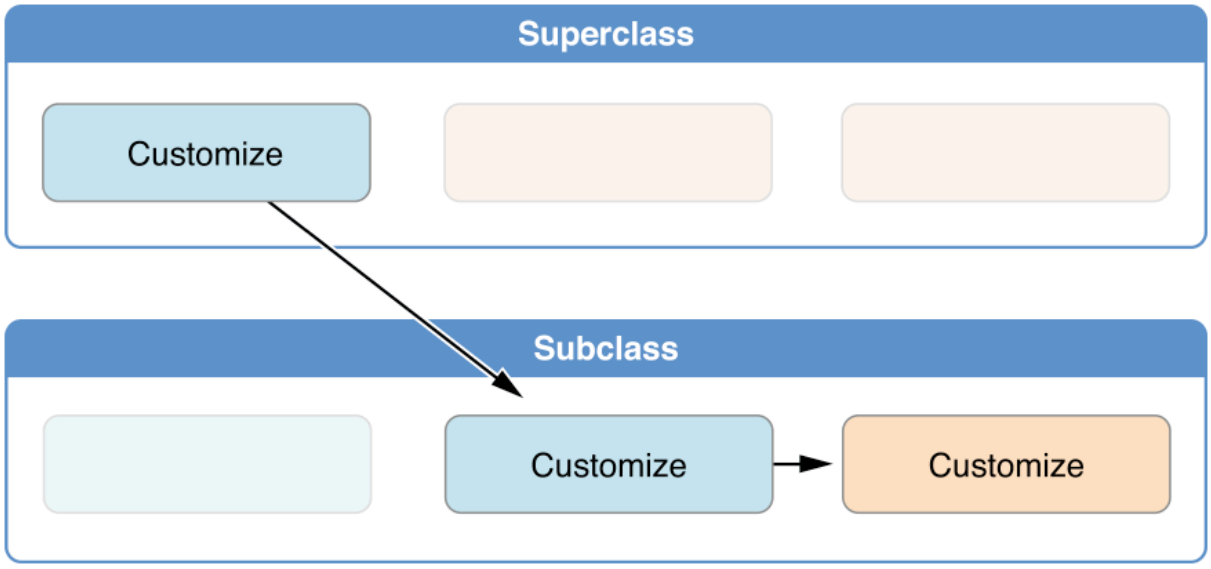
在这个栗子中，初始化过程从一个子类的便捷初始化器开始。这个便捷初始化器还不能修改任何属性。它委托给了同一类里的指定初始化器。

指定初始化器确保所有的子类属性都有值，如安全检查1。然后它调用父类的指定初始化器来沿着初始化器链一直往上完成父类的初始化过程。

父类的指定初始化器确保所有的父类属性都有值。由于没有更多的父类来初始化，也就不需要更多的委托。

一旦父类中所有属性都有初始值，它的内存就被认为完全初始化了，第阶段完成。

下图是相同的初始化过程在第二阶段的样子：



现在父类的指定初始化器有机会来定制更多实例(尽管没有这种必要)。

一旦父类的指定初始化器完成了调用，子类的指定初始化器就可以执行额外的定制(同样，尽管没有这种必要)。

最后，一旦子类的指定初始化器完成，最初调用的便捷初始化器将会执行额外的定制操作。

16.8 初始化器的继承和重写

不像在 Objective-C 中的子类，Swift 的子类不会默认继承父类的初始化器。Swift 的这种机制防止父类的简单初始化器被一个更专用的子类继承并被用来创建一个没有完全或错误初始化的新实例的情况发生。只有在特定情况下才会继承父类的初始化器.

如果你想自定义子类来实现一个或多个和父类相同的初始化器，你可以在子类中为那些初始化器提供定制的实现。

当你写的子类初始化器匹配父类*指定初始化器*的时候，你实际上可以重写那个初始化器。因此，在子类的初始化器定义之前你必须写 override 修饰符。如同默认初始化器所描述的那样，即使是自动提供的默认初始化器你也可以重写。

> 当重写父类指定初始化器时，你必须写 override 修饰符，就算你子类初始化器的实现是一个便捷初始化器。

下面的栗子定义了一个名为 Vehicle 的基类。基类声明了一个名为 numberOfWheels 的存储属性，类型为Int 的默认值 0 。 numberOfWheels 属性通过一个名为 description 的计算属性来创建一个 String 类型的车辆特征字符串描述：

```
class Vehicle {
    var  numberOfwheels = 0
    var description: String {
        return "\(numberOfwheels) wheels"
    }
}
```

Vehicle 类只为它的存储属性提供了默认值，并且没有提供自定义的初始化器。因此，如同默认初始化器中描述的那样，它会自动收到一个默认初始化器。默认初始化器(如果可用的话)总是类的指定初始化器，也可以用来创建一个新的 Vehicle 实例， numberOfWheels 默认为 0：

```
let vehicle = Vehicle()
print("Vehicle:\(vehicle.description)")
```

下面的例子定义了一个名为 Bicycle 继承自 Vehicle 的子类：

```
class Bicycle: Vehicle{
    override init() {
        super.init()
        numberOfwheels = 2
    }
}
//子类 Bicycle 定义了一个自定义初始化器 init() 。这个指定初始化器和 Bicycle 的父类的指定初始化器相匹配，所以 Bicycle 中的指定初始化器需要带上 override 修饰符。
```

```
let bicycle = Bicycle()
print("bicycle:\(bicycle.description)")
```

> 注意：子类可以在初始化时修改继承的变量属性，但是不能修改继承过来的常量属性

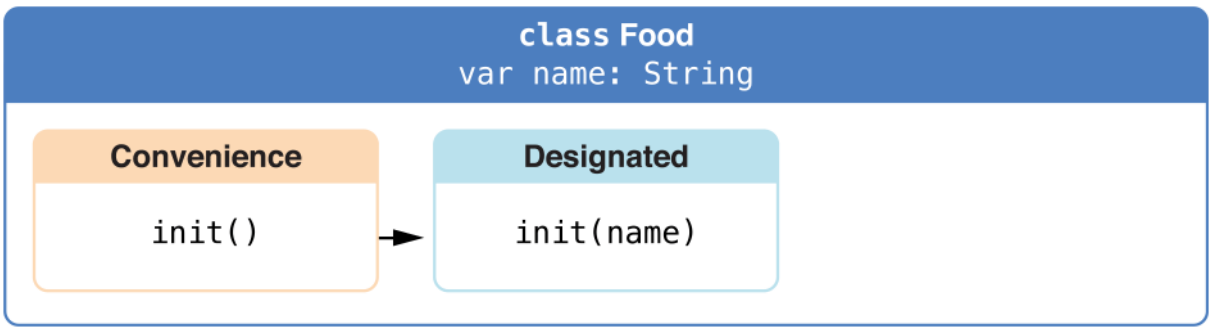
16.9 初始化器的自动继承

子类默认不会继承父类初始化器。总之，在特定的情况下父类初始化器*是*可以被自动继承的

假设你为你子类引入的任何新的属性都提供了默认值，请遵守以下2个规则：

- 如果你的子类没有定义任何指定初始化器，它会自动继承父类所有的指定初始化器。
- 如果你的子类提供了*所有*父类指定初始化器的实现——要么是通过规则1继承来的，要么通过在定义中提供自定义实现的——那么它自动继承所有的父类便捷初始化器。

```
class Food {
    var name: String
    init(name: String) {
        print("父类init(name: string)")
        self.name = name
    }
    convenience init() {
        print("父类init()")
        self.init(name: "[Unnamed]]")
    }
}
```




```
let nameMeat = Food(name: "Bacon")
```

Food 类的 init(name: String) 初始化器是一个*指定初始化器*，因为它确保 Food 类新实例的所有存储属性都被完全初始化。 Food 类没有父类，所以 init(name: String) 初始化器不用调用 super.init() 来完成它的初始化。

Food 类也提供了没有实际参数的*便捷*初始化器 init() 。 init() 初始化器通过委托调用同一类中定义的指定初始化器 init(name: String) 并给参数 name 传值 [Unnamed] 来实现提供默认的名称占位符

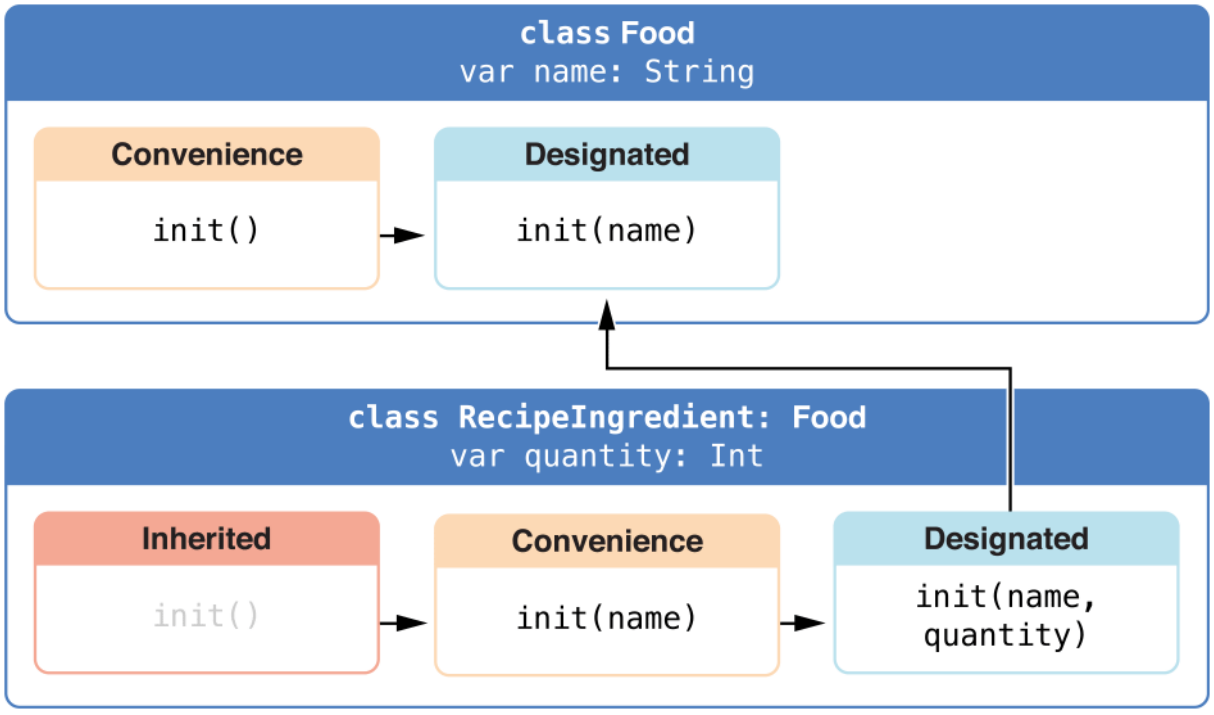
```
let mysterMeat = Food()
```

Food的子类RecipeIngredient调味剂，它引入了 Int 类型的数量属性 quantity

```
class RecipeIngredient: Food{
    var quantity: Int
    init(name: String, quantity: Int) {
        print("子类init(name, quantity)")
        self.quantity = quantity
        super.init(name: name)
    }

    override convenience init(name: String){
        print("子类init(name: string)")
        self.init(name: name, quantity: 1)
    }
}
```

下面的图标展示了 RecipeIngredient 了的初始化链:



RecipeIngredient 类只有一个指定初始化器， init(name: String, quantity: Int) ，它可以用来填充新的RecipeIngredient 实例中所有的属性。这个初始化器一开始先将传入的 quantity 实际参数赋值给 quantity 属性，这个属性也是唯一一个通过 RecipeIngredient 引入的新属性。随后，初始化器将向上委托给父类 Food 的 init(name: String) 初始化器。这个过程满足上边所述的**两段式初始化**的安全检查

1。

RecipeIngredient 类提供的 init(name: String) 便捷初始化器接收与 Food 中指定初始化器 init(name:String) 相同的形式参数，它必须使用override 修饰符

尽管 RecipeIngredient 提供了 init(name: String) 初始化器作为一个便捷初始化器，然而 RecipeIngredient 类为所有的父类指定初始化器提供了实现。因此， RecipeIngredient 类也自动继承了父类所有的便捷初始化器。

```
let rec = RecipeIngredient()
/*
父类init()
子类init(name: string)
子类init(name, quantity)
父类init(name: string)
*/
```

```
let rec2 = RecipeIngredient(name: "boom")
/*
子类init(name: string)
子类init(name, quantity)
父类init(name: string)

*/
```

第十七章 值类型与引用类型

第十八章 协议

第十九章 扩展