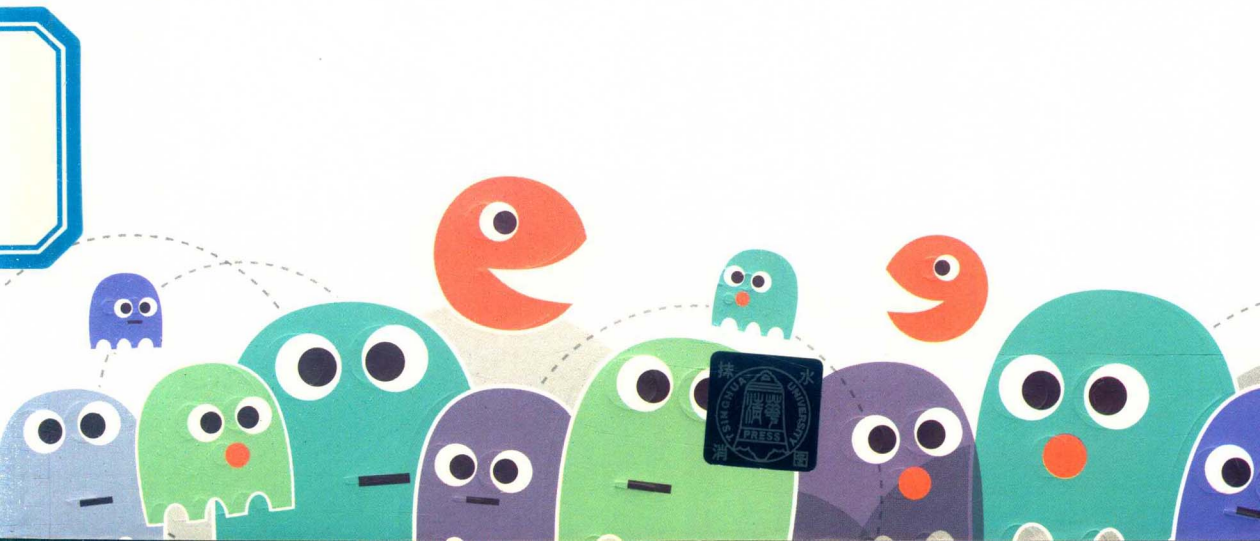
移动互联网开发技术丛书

iOS开发快速进阶与实战

黄新宇 著

清华大学出版社



移动互联网开发技术丛书

iOS开发快速进阶与实战

黄新宇 著

清华大学出版社
北京

内 容 简 介

本书偏向于iOS应用的实际开发，介绍了iOS开发过程中的技术实现方案和原理，包含基本知识、底层常用技术原理、开发技巧，以及技能扩展等各方面，其中大部分章节以实际项目开发中常见的问题为背景，内容阐述方式包括介绍原理、对比技术方案、实际应用、引导读者思维等，并在每一章最后部分归纳总结本章的重点内容。

本书既可以作为高等学校计算机软件技术课程的教材，也可以作为企业iOS开发人员的技术参考书。

图书在版编目(CIP)数据

iOS开发快速进阶与实战/黄新宇著. —北京：清华大学出版社，2018
(移动互联网开发技术丛书)
ISBN 978-7-302-50385-9

Ⅰ. ①i…　Ⅱ. ①黄…　Ⅲ. ①移动终端－应用程序－程序设计　Ⅳ. ①TN929.53

中国版本图书馆CIP数据核字(2018)第122958号

责任编辑：付弘宇　薛　阳
封面设计：刘　键
责任校对：李建庄
责任印制：丛怀宇

出版发行：清华大学出版社
网　　址：http://www.tup.com.cn，http://www.wqbook.com
地　　址：北京清华大学学研大厦A座　　**邮　　编**：100084
社 总 机：010-62770175　　**邮　　购**：010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质量反馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
课件下载：http://www.tup.com.cn,010-62795954
印 装 者：北京嘉实印刷有限公司
经　　销：全国新华书店
开　　本：185mm×260mm　　**印　　张**：12.25　　**字　　数**：294千字
版　　次：2018年8月第1版　　**印　　次**：2018年8月第1次印刷
印　　数：1～1500
定　　价：49.00元

产品编号：074360-01

前言

FOREWORD

成书背景

移动互联网经过近几年的快速发展，已日趋成熟稳定，在极大方便人们生活的同时，也正在悄悄改变人们的生活方式。互联网现已从 PC 端逐步划分出移动端的大群体，手机也不仅局限于其传统打电话发短信等基本功能，高速上网更满足当代人的生活需求。由上网衍生出来的手机功能包括即时通信、新闻资讯、视听娱乐、游戏、支付转账、生活工具等，这些都是移动互联网的代表领域，其中不乏有很多知名的应用为大众所知。

智能手机的发展也一直在接受移动互联网发展的检验，从发展之初的 Windows Mobile（Windows Phone 前身），到塞班、黑莓，再到现在的 iOS、安卓、Windows Phone，智能手机在发展之初无时无刻不经历着大风大浪，或许今年还是非常受欢迎的操作系统，明年市场份额就所剩无几。就目前而言，移动端操作系统主要分为 iOS 和安卓，这两者现今几乎占据了全部智能手机的市场份额，移动应用大部分只会考虑这两个方向。

iPhone 自问世之初，一直以惊艳闻名，在随后智能手机发展的过程中，也一直引领设计和硬件功能的创新，不仅于此，iPhone 的用户体验和操作系统的流畅度一直是被用户喜爱的主要原因。

开发 iPhone 应用称为 iOS 开发，不是对其操作系统的开发，而是开发基于 iOS 系统运行的应用程序。对于 iOS 开发，有 Objective-C 和 Swift 两种语言，但仅仅是在语法和编程方式上有较明显的差异，其主要实现方式往往没有太大的区别。

关于本书

本书是按照章节进行大致划分，内容之间没有依赖顺序关系，每一节的知识点都是相互独立的，读者可以根据自身情况进行选读和跳读。

本书内容主要包括三个方面：一是以开发中遇到的实际问题为例，列出的场景都是实际开发中常见的开发任务，在这些章节内容主要以实践为主，并附上了详尽的代码实现过程；第二个是偏向理论的内容，主要以面试题为基础进行深入的分析，旨在让读者不再死记硬背面试题，而是根据内容去理解这些理论的原理或实现过程；最后一个是技能进阶，针对问题的实现方式从不同角度给出实现方案，最后通过理论比较得出最优解，或者对于某些问

题提供比较巧妙的解决办法，开拓读者的思考方式。

笔者自 iOS 6 开始接触，虽然算不上是最早的一批开发者，却也总结了一些个人开发经验。本书内容是笔者自从事 iOS 开发以来的所有总结的整理。书中的内容主要以理论和实践为主，从提出问题到分析问题，再到解决问题，包括部分章节内容以使用场景带入，都是实际开发中所经常遇到的问题。整部书从准备到写成，持续了有近一年的时间，其实时间还是蛮紧张的。大部分的章节内容，从提出，到叙述，到举例，到论证，最后到总结是一个严谨的流程，不同于写个人博客。

书籍和博客虽然是优秀的知识传播媒介，但不足之处在于阅读时不一定能够理解作者真正想表达的意思，特别是对于技术开发这种实践性较强的情况。本书的内容花了很大的篇幅讲述了理论性的知识，示例代码作为其辅助说明的手段。或许读者能够在阅读时产生共鸣，因为可能遇到过相同的问题或者对于问题有相同的理解，但笔者建议读者能够更多地将章节内容以实践运用的方式来加深自我理解。另外，本书中的内容都是以知识点的形式，相对独立化，而在实际开发中又是另一回事，例如，需要考虑代码复用性，以及编程思想的运用，这些都需要读者对其熟练地使用，而不仅仅是了解。

基本上所有的开发者都有学习过其他开发者优秀的代码或文章，提升自我能力的前提是站在巨人的肩膀上，可以使自己少走很多弯路，同时也飞速提升了自我实力。因此笔者也希望能够以这本书给读者带来一些真正意义上的帮助。

本书中的示例代码都是在 Xcode 8. x 下运行，书中的示例代码仅考虑 iOS 8 以上，语言以 Objective-C 为主，部分内容涉及 Swift。

由于笔者能力有限，书中难免存在疏漏和不足之处，因此特地在 GitHub 上开了一个仓库，有任何意见和建议的读者，欢迎来这里提出，地址：https://github.com/huangxinyu1213/iOS-Advanced-book。

目标读者

对于现在的编程来说，其编程语言变得越来越高级，使得开发门槛越来越低，开发者不必过多接触底层的实现，以及去写一些复杂的代码。就类似于 iOS 开发的 Objective-C 和 Swift 这两门语言一样，虽然是两种完全不同的开发语言，但开发者从 Objective-C 开发转到 Swift 开发其实并不是一件很难的事情，因为对 iOS 的 CocoaTouch 框架的使用，两门语言基本无异，这也从另一个角度可以证明，高级语言有很多的共同性，开发语言只是一种实现方式。

另一方面，市面上现在有很多对于 iOS 开发基础的图书和教程，包括苹果的官方文档，对于初学者来说，都是很好的入门资料。本书可能并不适合 iOS 初学者，因为本书并不打算从 iOS 的基础内容开始讲起，而更适合于一些有 iOS 开发基础的初中级开发工程师参考。

主要内容

本书内容不涉及 iOS 开发的基础知识，由一个个相互独立的知识点组成章节，主要是以进阶为目的，帮助开发者更高效地运用 iOS 开发技术。主要内容包括以下各章。

第 1 章：iOS 的类

类是面向对象的基础，iOS 的类不仅是实现面向对象，还有一些值得关注的特性和原理。

第 2 章：底层实现分析

iOS 开发中，系统为开发者提供了许多便捷和强大的基础功能，避免写一些过于复杂的底层实现代码，使编程人员只需要更加注重于代码和业务本身。

第 3 章：开发原理相关

主要介绍开发中常用技术以及实现的原理，同时也会对一些技术给出不同的实现方案，并做出比较，让开发者对一些原理知识能够有比较明确的认识。

第 4 章：线程安全——锁

线程安全是 iOS 开发中避免不了的话题，随着多线程的使用，对于资源的竞争以及数据的操作都可能存在风险，所以有必要在操作时保证线程安全。

第 5 章：排序算法

对算法的掌握是非常有必要的，不仅是在面试中经常会考算法，在实际应用中，算法的使用能更高效地处理数据。同时算法的思想也能更好地帮助我们理解计算机语言。

第 6 章：技能进阶与思考

本章内容更偏向于实际场景的应用和实现方式的思考，以及扩充开发者的知识宽度。

电子资源

笔者提供书中所有实例的源代码，读者可以从清华大学出版社网站 www. tup. com. cn 下载。本书使用与资源下载的相关问题请联系 fuhy@tup. tsinghua. edu. cn。

编 者

2018 年 5 月

目 录

CONTENTS

第1章

iOS的类

类是面向对象的基础，iOS 的类不仅是实现面向对象，还有一些值得关注的特性和原理。本章以 Objective-C 语言为基础，介绍 iOS 的一些基础知识。

本章内容：

- 创建并描述一个类
- 类属性和类方法
- 黑魔法

1.1 创建并描述一个类

开发中经常需要创建类文件，可以说在项目的开发过程中，很大一部分是由类组成的，要在 Objective-C 中创建一个类，需要继承 NSObject 或者其子类。NSObject 及其子类就是 Objective-C 语言中对对象的实现。

打开 NSObject 的定义，可以看到在其头文件中仅有一百行代码，即定义了对象及其基本方法。

```
@interface NSObject < NSObject > {
    Class isa OBJC_ISA_AVAILABILITY;
}
```

NSObject 其实是一个实现了同名协议的接口(Interface)，参数只有一个 isa，指向当前所属类。在 NSObject 协议中，不同的 NSObject 类，需要根据实际情况来重写部分方法。

在实际开发中，如何去继承一个 NSObject 类呢？看似简单，却有许多需要注意的地方。可以通过下面一个简单例子来说明。首先创建一个类，如图 1-1 所示。

简单地创建一个继承自 NSObject 的对象后，文件的结构目录上会多出两个 Person 的文件，一个是头文件(.h)，另一个是实现(.m)。头文件一般都是对这个类起到一个介绍的

图 1-1 创建一个类

作用，开发者可以直接通过头文件来获取该类的一些基本信息，包括类的属性和方法，从而不需要关心该类是如何实现的。而与之对应的就是其实现文件，相对于头文件，实现文件的代码量大多会比头文件多一些，针对头文件中介绍的属性和方法，实现文件应当提供具体的实现，从而保证这个类可以被很好地使用。打个比方，一个类就相当于是一家餐厅，而头文件就是它的菜单，食客（开发者）通过菜单就能了解到这家餐厅有哪些菜品食物可以点，而对应的这些菜品食物都是在餐厅的厨房进行具体的实现，厨房可以做一些菜单上没有的菜品，但是菜单上却不能有厨房做不了的菜品。

首先制作菜单，打开 Person.h 文件，内容如下。

```
#import <Foundation/Foundation.h>
@interface Person : NSObject
@end
```

导入 Foundation 框架。Foundation 框架是系统的基于 CoreFoundation 框架的封装，即常说的 Cocoa 框架，包含数据集合、日期、文件处理，KVC/KVO，Runloop 以及网络通信等一系列基本功能的类集合。

在头文件中导入框架尽量不要重复，如果 Person 类有一个 UIImage 的属性 headImage，则需要在头文件中加入 UIKit 框架。

```
#import <Foundation/Foundation.h>
#import <UIKit/UIKit.h>
```

此时，加入 UIKit 后，可以不再需要“#import < Foundation/Foundation.h >”了，因为 UIKit 是默认包含 Foundation 框架的，重复的包含是无意义的，并且会使代码看起来并不简洁明了。但是毕竟 Person 是一个模型类，并不是展示类，导入 UIKit 会给开发者造成困扰。通常来说，第一个 #import 的框架可以很明确地让开发者明白该类是属于哪一种类型。所以此时的做法应该为所需要的 UIImage 添加一个前向声明：

```
#import <Foundation/Foundation.h>
@class UIImage;
@interface Person : NSObject
@property (nonatomic, strong) UIImage *headImage;
@end
```

接下来，我们会想到这个 Person 类会需要一个 NSString 类型的 name 属性，而且是必需的，而刚才提到的 headImage，可能为 nil。在这种情况下，为了兼容 Swift 的可选值类型，需要在对应的 Property 中加上特定的修饰符：nullable 或 nonnull。这样在 Swift 中使用会桥接成更明确的方法。

```
#import <Foundation/Foundation.h>
@class UIImage;
NS_ASSUME_NONNULL_BEGIN
@interface Person : NSObject
@property (nonatomic, strong, nullable) UIImage *headImage;
@property (nonatomic, copy,nonnull) NSString *name;
@end
NS_ASSUME_NONNULL_END
```

如果一个类的属性很多，这样做则会很麻烦，所以可以利用系统提供的 NS_ASSUME_NONNULL_BEGIN、NS_ASSUME_NONNULL_END 这两个宏来将默认未注明 nullable 还是 nonnull 的类的属性都默认设置为 nonnull。

属性写完后，就需要写初始化方法。对于 Person 类，我们需要在初始化的时候传递一个 NSString 的 name，并且最好能给 headImage 赋上一个默认的头像。

```
// Person.h
#import <Foundation/Foundation.h>
@class UIImage;

NS_ASSUME_NONNULL_BEGIN

@interface Person : NSObject
@property (nonatomic, strong, nullable) UIImage *headImage;
@property (nonatomic, copy, nonnull) NSString *name;

- (instancetype)initWithName:(NSString *)name;

@end
```

```
NS_ASSUME_NONNULL_END

// Person.m
#import "Person.h"
#import <UIKit/UIImage.h>

@implementation Person

- (instancetype)initWithName:(NSString *)name {
    if (self = [super init]) {
        _name = name;
        _headImage = [UIImage imageNamed:@"default_head"];
    }
    return self;
}

@end
```

很简单，似乎并没有什么复杂的地方，可是当代码提交之后，协同开发的同事需要使用你的 Person 类创建实例时，却发现并没有默认的头像，这个时候你就会很疑惑，仔细一看同事的代码是这样写的：

```
Person *tom = [[Person alloc] init];
tom.name = @"Tom";
```

同事并没有使用你想让他使用的初始化方法，只是使用最基本的 alloc 和 init。没有使用我们提供的初始化方法就会未设置默认头像。这是协同开发中经常会遇到的问题。如何让同事也使用自己提供的方法，而不是使用其他的初始化方法呢？

```
// Person.h
#import <Foundation/Foundation.h>
@class UIImage;

NS_ASSUME_NONNULL_BEGIN

@interface Person : NSObject
@property (nonatomic, strong, nullable) UIImage *headImage;
@property (nonatomic, copy, nonnull) NSString *name;

- (instancetype)initWithName:(NSString *)name;
- (instancetype)init UNAVAILABLE_ATTRIBUTE;
+ (instancetype)new UNAVAILABLE_ATTRIBUTE;

@end

NS_ASSUME_NONNULL_END
```

通过 UNAVAILABLE_ATTRIBUTE 修饰以在头文件中禁用其他的初始化方法，可

保证只有自己提供的初始化方法才是唯一途径。此时，如果要创建 Person 实例，只能通过-initWithName：初始化方法。

本节小结

(1) 创建类应尽量明确类型，必要时加上前向声明；

(2) 属性应当标明可选还是必选，在与 Swift 混编时会生成更加明确的 Swift 方法；

(3) 通过 UNAVAILABLE_ATTRIBUTE 禁用相关方法。

1.2 类方法的 self

引用 1.1 节中的内容，下面通过 alloc 和 initWithName 的方式来创建 Person 对象：

```
- (instancetype)initWithName:(NSString *)name {
    if (self = [super init]) {
        _name = name;
        _headImage = [UIImage imageNamed:@"default_head"];
    }
    return self;
}
```

虽然禁用了 init 方法，但是通过 super 调用父类的 init 并不会有影响。如果经常使用到，这个方法仍然显得有些麻烦，可以通过类方法来使创建过程更加简捷：

```
+ (instancetype)personWithName:(NSString *)name;
```

既然有了类方法来创建，我们也就不需要对外提供-initWithName:方法了，按照 1.1 节中的方法，将其标注为不可用。

```
// Person.h
#import <Foundation/Foundation.h>
@class UIImage;

NS_ASSUME_NONNULL_BEGIN

@interface Person : NSObject
@property (nonatomic, strong, nullable) UIImage *headImage;
@property (nonatomic, copy, nonnull) NSString *name;

+ (instancetype)personWithName:(NSString *)name;
- (instancetype)initWithName:(NSString *)name UNAVAILABLE_ATTRIBUTE;
- (instancetype)init UNAVAILABLE_ATTRIBUTE;
+ (instancetype)new UNAVAILABLE_ATTRIBUTE;

@end
NS_ASSUME_NONNULL_END
```

因为之前有-initWithName:的实现，我们希望在新加入的＋personWithName:方法中

可以直接调用，以下是错误代码。

```
// 这是错误代码!
+ (instancetype)personWithName:(NSString *)name {
    Person *p = [[Person alloc] initWithName:name];
    return p;
}
```

报错的原因很简单，就是因为我们禁用了 Person 类的-initWithName:方法，解决办法如下。

```
+ (instancetype)personWithName:(NSString *)name {
    Person *p = [[self alloc] initWithName:name];
    return p;
}
```

将 Person 换成 self 调用就不会报错，但据我们所知，在类方法中使用这两者似乎并没有什么区别，实际上 self 是表示当前类，而不一定是 Person 类。使用类方法创建实例，用[self alloc]和[Person alloc]在基本使用上是没有任何区别和影响的，但是并不代表二者没有区别，甚至会有使用上的误区，更甚者会导致意料之外的错误发生。

为了方便解释，下面再举一个类似的例子，创建一个 Animal 类，且为 Animal 类添加一个类方法类创建实例，并将默认的 init 设置为不可用。

```
// Animal.h
@interface Animal : NSObject

@property (nonatomic, copy) NSString *name;
- (instancetype)init NS_UNAVAILABLE;
+ (instancetype)animalWithName:(NSString *)name;

@end


// Animal.m
#import "Animal.h"

@implementation Animal

+ (instancetype)animalWithName:(NSString *)name {
    Animal *animal = [[self alloc] init];
    animal.name = name;
    return animal;
}
@end
```

此时，如果将＋animalWithName:方法中的[[self alloc] init]换成[[Animal alloc] init]，则会报错，错误很明显，是因为我们将 init 方法设置为不可使用，编译器在此时会自动

设置为该方法不可使用。那为什么用[[self alloc] init]就会不报错了呢？难道是编译器的bug？当然不是，请接着看。

我们要继承 Animal 来创建它的子类 Dog：

```
// Dog.h
#import "Animal.h"
@interface Dog : Animal
@end
```

此时，如果使用下面这句代码来创建，仍然会报错。原因是 init 方法在父类 Animal 方法中被禁用，所以在子类中仍然不可使用。

```
Dog *dog = [[Dog alloc] init];
```

假设在+animalWithName:方法中用的是 Animal，那么用它来创建 Dog 实例肯定是不对的，因为当前类是 Dog，如果再用 Animal 来创建肯定是错误的，等于是创建了一个 Animal 实例来赋给 Dog 对象。如果用 self 来创建的话则没有此问题，因为对于 Dog 类来说，self 即指的是 Dog，self 代表具体当前是哪一个类，因此返回结果是 Dog 实例，这是我们所希望的，所以这也是为什么应该在 Animal 类中+animalWithName:方法中使用[self alloc]而不是[Animal alloc]。

```
Dog *dog = [Dog animalWithName:@"Hachiko"];
```

在此基础上，假设一种很新奇的场景，因为 Animal 对我们来说是很概念性、很抽象的类，所以我们提供了+animalWithName:方法，而禁止-init 方法，但 Dog 类是非常具体的，我们希望可以保留-init 方法来为 Dog 类提供另外一种初始化方法，当然-init 方法仍然是对 Animal 不可使用的。可以将-init 方法在 Dog 的头文件中再声明一次，表示对该方法在此处可以破例使用。

```
// Dog.h
#import "Animal.h"
@interface Dog : Animal

- (instancetype)init;

@end
```

此时，就可以在任意处使用-init 方法来为 Dog 创建实例，当然-init 方法对于 Animal 类来说仍然是不可用的。

然而，事情并没有我们想的那样容易，虽然在这种场景下的问题已经解决了，我们又有了另外一个新的问题，编译器报了一个没有实现-init 方法的警告！虽然我们在 Dog 类中设置-init 方法再次可用了，而且我们知道，-init 方法在 NSObject 基类中是有默认实现的，此时此处报警告从主观意识上来说有一些不合情理，但是毕竟是负责任的编译器，为了防止我们会有这样的隐患，可以手动去除警告。

```
// Dog.m
#import "Dog.h"
#pragma clang diagnostic push
#pragma clang diagnostic ignored "-Wincomplete-implementation"
@implementation Dog
@end
#pragma clang diagnostic pop
```

本节小结

(1) 类方法中的 self 指的是当前类，而不是固定的某个类，还可能是这个类的子类；

(2) 对于父类禁用的方法，需要在子类头文件中再次声明才可以使用，并需要在类实现文件中去除编译器警告。

1.3 类属性

我们知道，在实例方法中，self 指的是类实例，而在类方法中，self 指的是类，而不是类实例，一般情况下也是可以直接将 self 换成类名来调用。类实例是由类创建的，那类是怎么来的呢？图 1-2 是类关系图，可以看出，类都是来自于一个叫作 MetaClass 的类。

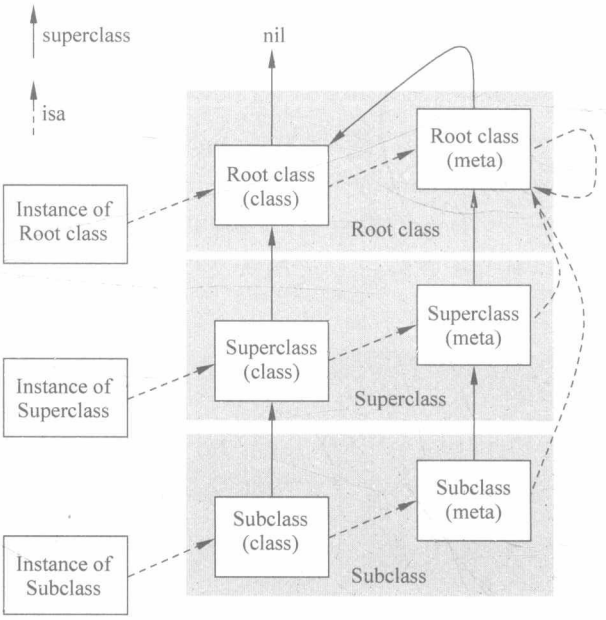


图 1-2 类的关系

类可以看作是其元类的实例。与此同时，我们再看一下 objc 对象对应的结构体：

```
struct objc_class {
    Class isa OBJC_ISA_AVAILABILITY;

#if !__OBJC2__
```