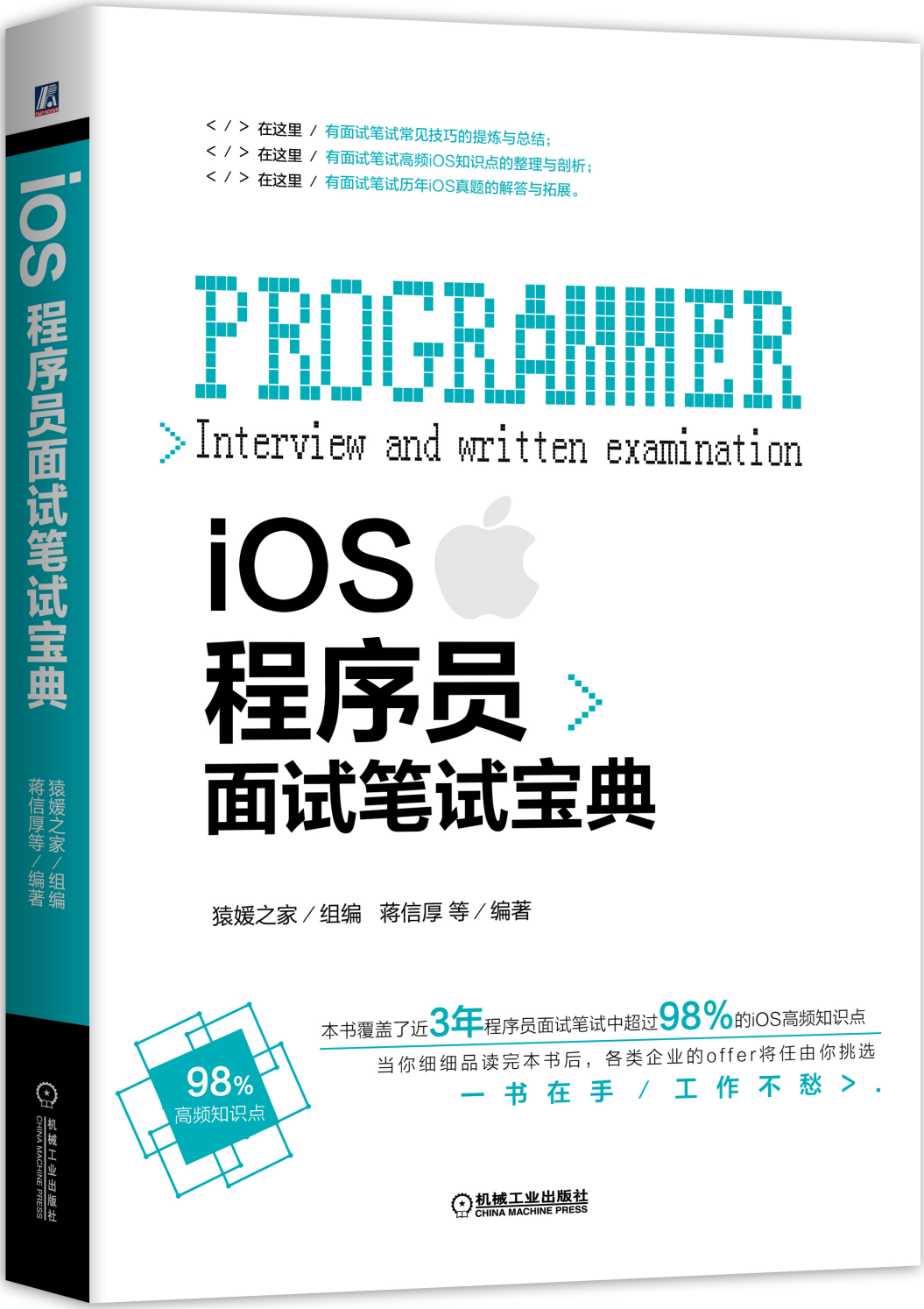
iOS 程序员面试笔试宝典
猿媛之家／组编
蒋信厚等／编著
机械工业出版社
CHINA MACHINE PRESS
< / > 在这里 / 有面试笔试常见技巧的提炼与总结；
< / > 在这里 / 有面试笔试高频iOS知识点的整理与剖析；
< / > 在这里 / 有面试笔试历年iOS真题的解答与拓展。
PROGRAMER
Interview and written examination
iOS 程序员 面试笔试宝典
猿媛之家／组编 蒋信厚 等／编著
98% 高频知识点
本书覆盖了近3年程序员面试笔试中超过98%的iOS高频知识点
当你细细品读完本书后，各类企业的offer将任由你挑选
一书在手 / 工作不愁 >.
机械工业出版社
CHINA MACHINE PRESS

京东、淘宝天猫等平台可购买纸质版全书啦，搜索关键词“iOS 程序员面试笔试宝典”可选购，或者扫描下面二维码直接在京东上购书！

iOS 程序员面试笔试宝典

猿媛之家　组　编
蒋信厚　等编著

机械工业出版社

前　言

本书是我读研期间开始着手起草，到交稿给出版社为止，整整耗时一年半。在此之前，我一直对 iOS 技术情有独钟，充满兴趣和学习热情。回想过去四五年的 iOS 学习历程，很庆幸当时的决心和之后的坚持，正是这种决心和坚持使我能收获一项自己最擅长的技能。

我从大二开始接触 iOS 开发，那时候 iOS 5 刚刚发布，iOS 技术刚火起来，而我已有的技术积累并不算多，所以学习难度非常大。当时，能够熟练开发 iOS 应用的同学真的是凤毛麟角，他们很让人敬佩，而这也更激发了我想学会 iOS 开发的欲望。为了学习 iOS 基础，我借阅了学校图书馆所有关于 Objective-C 的书，对于计算机基础还不扎实的我去自学 Objective-C 真的是很痛苦，各种 iOS 开发指南和开发案例的书也让我很吃力。好在那时候我做了一个正确的选择，用自己刚拿到的奖学金和攒下的钱毅然决然地买了一台低配的新款 Mac Pro 笔记本式计算机（就是 2013 年刚出视网膜屏幕且大大瘦身的那款，至今这台笔记本式计算机仍在我手中服役），这也是我能够长期保持学习热情以及后来深入 iOS 平台开发领域的敲门砖。

我最开始主要是拿别人的简单 DEMO 来学习，在别人写好的代码上改改、调调，我觉得这是入门最快的方法。在基本入门以后，我便开始系统地去看书、去验证，去实现自己的小想法，去尝试做一个小项目等。此外每当遇到问题，我都会去开发论坛交流，当时经常逛的网站有 Cocoa China、Stack Overflow 和 GitHub 等，在此过程中慢慢积累、慢慢武装自己。学习 iOS 的周期还是很漫长的，入门以后很长一段时间我并没有太大的提高，直到后来加入实际项目、参加公司实习以及能够研究一些优秀的开源代码之后，我才踏入进阶之路，同时也伴随着自己计算机专业水平的同步提高。事实上，我本科期间学习的汇编、编译原理、计算机组成原理、数据结构、数据库原理等，都一直在起着潜移默化的作用，而 iOS 技术的学习，则像是一个实践的平台，一个深入行业领域的路线。条条河流最终都是要汇入大海的，无论是一开始选择 iOS 开发、安卓开发，还是选择 Web 开发等路线，其最终目标都应该是借此打一口深深的井，钻下去然后慢慢扩散渗透，成为一名优秀的计算机行业专家，而不仅仅是一个初级平台开发者而已。

在编写整理本书期间，我从头到尾系统地梳理了自己的知识体系，不断地去验证、去挖掘重点、去剖析学习中最容易忽视的知识点，揭开我们学习中常常不愿意揭开的伤疤，然后认真地做出解析、敷上药膏。希望本书能够帮助更多的读者压缩这个学习过程的成本和周期，更快速地进入到更高的技术层面，更顺利地适应工作岗位，成为优秀的行业开发者。毕业后，我进入游戏行业，仍一直坚持 iOS 的使用和学习，这个长板对我尤其重要，是我平时想法和技术实践的主要移动平台。

本书技术部分将精选收录的题目进行了分类。第 1 章收录了 iOS 开发中的一些基础概念相关的问题；第 2 章和第 3 章分别归类了 Objective-C 语言从基础知识到中高级特性的问题，由浅入深地剖析了 Objective-C 语言开发各方面的核心问题；第 4 章收录的主要是有关官方

Cocoa Touch 框架的常见问题；第 5 章将 iOS 开发中的对象间通信机制相关的问题集中在一起，进行了总结和分析；第 6 章讨论了 iOS 中的一个重要话题：图层与动画，并结合问题进行了知识的总结和延伸；第 7 章总结了数据持久化有关的问题；第 8 章和第 9 章包含了 iOS 开发进阶之路的重中之重：内存管理和网络多线程编程；第 10 章收录了一些其他的重要的零碎话题，包括函数响应式编程、设计模式、第三方框架和程序调试问题等；第 11 章介绍了数据库相关知识；第 12 章介绍了操作系统相关知识。

对于书中的任何问题或困惑，读者都可以通过邮件联系我：yuancoder@foxmail.com。期待你的来信。

编　者

2018 年 5 月

目　录

面试笔试经验技巧篇

想找到一份程序员的工作，一点技术都没有显然是不行的，但是，只有技术也是不够的。面试笔试经验技巧篇主要提供 iOS 程序员面试笔试经验、面试笔试问题方法讨论等。通过本篇的学习，求职者必将获取到丰富的应试技巧与方法。

位也许已经“人满为患”或“名花有主”了，但企业对你兴趣不减，还是很希望你能成为企业的一员。面对这种提问，求职者应该迅速做出反应，如果认为对方是个不错的企业，你对新的岗位又有一定的把握，也可以先进单位再选岗位；如果对方情况一般，新岗位又不太适合自己，最好当面回答不行。

（17）你能来实习吗

对于实习这种敏感的问题，面试官一般是不会轻易提及的，除非是确实对求职者很感兴趣，相中求职者了。当求职者遇到这种情况时，一定要清楚面试官的意图，他希望求职者能够表态，如果确实可以去实习，一定及时地在面试官面前表达出来，这无疑可以给予自己更多的机会。

（18）你什么时候能到岗

当面试官问及到岗的时间时，表明面试官已经同意给 offer 了，此时只是为了确定求职者是否能够及时到岗并开始工作。如果确有难题千万不要遮遮掩掩，含糊其辞，一定要说清楚情况，诚实守信。

针对面试中存在的这种暗语，求职者在面试过程中，一定不要“很傻很天真”，要多留一个心眼，多推敲面试官的深意，仔细想想其中的“潜台词”，从而将面试官的那点“小伎俩”掌控。

经验技巧 14　名企 iOS 工程师行业访谈录

某知名互联网公司研发工程师访谈录

1．当前市场对于 iOS 程序员的需求如何？待遇如何？

就笔者所在的互联网公司来说，因为现在产品基本上是移动端先行，所以对 iOS 程序员的需求量还是挺大的，而 iOS 程序员的待遇基本与同级其他岗位（除算法岗外）无差别。

2．iOS 程序员未来的发展方向如何？

对于发展方向而言，我的个人感觉还是要看 iOS 程序员个人的成长路线以及以后的发展目标，iOS 这个行业的前景和市场需求目前还是比较光明的。

对于不同层次的 iOS 程序员成长路线和发展方向，我觉得可以分为以下几个方面的内容：

1）独立 App 开发。

2）业务能手，业务逻辑抽象。

3）SDK 功能组件开发。

4）跨端技术 Weex，React—Native 等。

5）底层研究、iOS 汇编、性能、网络、安全等研究。

6）端上机器学习（Core ML）和 AR（ARKit）这些新技术也很有发展前景。

总之，iOS 程序员并非要局限于 iOS 开发本身，条条大河终入海，iOS 程序员要以 iOS 开发为入口，深入下去，不断深入计算机领域，努力走在计算机科学技术发展的前端。

iOS 的行业前景主要依赖于 iOS、iPhone 本身的发展以及 App Store 的生态，目前看起来，iOS 的行业前景无须担心，它们都还处于上升期。大公司的 iOS 研发其实一直缺人，但是满足条件的开发者较少，由于现在行业新产品发布时 Web 版可以考虑不做，但是移动端是一定要有，所以，市场对 iOS 程序员的需求量还是很大的。

3．iOS 程序员有哪些可供选择的职业发展道路？

我认为可以大概划分以下两个路线：

1）一个是 UI 线，在大业务中专门负责业务页面搭建，沉淀 UI 组件。

2）一个是基础架构线，主要实现网络、高可用、App 架构等。

4．企业在招聘时，对 iOS 程序员通常有什么要求？iOS 程序员的日常工作是什么？

企业在招聘时，主要还是考察求职者对 iOS 开发基础知识的掌握情况，例如对 Objective-C 和 Swfit 语言的了解，对 App 运行机制的了解，对基础 Framework 以及业界知名的第三方框架的了解等，还有一部分较为重要的就是求职者的软素质，例如学习能力和沟通能力等。

iOS 程序员的日常工作大概可以分为以下几类：

1）最主要的还是业务页面的搭建，已有业务页面的维护。

2）基础组件（网络、UI 等）的编写维护，第三方组件的接入和升级。

3）iPhone 机型以及 iOS 系统适配。

5．要想成为一名出色的 iOS 程序员，需要掌握哪些必备的知识？有哪些好的书籍或是网站可供推荐学习？

一名出色的 iOS 程序员的必备知识基本与招聘要求是一致的：

1）Objective-C 和 Swift 的基础知识。

2）UIKit 和 Foundation 两个库的使用。

3）iOS App 以及 iOS 系统的运行机制。

对于学习书籍，由于我个人看得比较少，所以这里我就不推荐了，而我主要是通过以下几种方式来学习提升的：

1）苹果（https://developer.apple.com/documentation）和第三方库的文档。

2）优秀开发者的博客，例如喵神：https://onevcat.com/。

3）阅读 GitHub 优秀开源项目的源码。

某知名互联网公司研发专家访谈录

1．当前市场对于 iOS 程序员的需求如何？待遇如何？

因为 iOS 开发入门的门槛相对较低，虽然当前市场上入门级的 iOS 开发已经饱和了，但是对于 iOS 的中高端人才需求缺口仍然很大。而待遇方面的情况可以参考各类招聘网站的信息了，在此就不透露我个人以及我所在企业的薪酬体系了。

2．iOS 程序员未来的发展方向如何？

手机现在是人们生活中必不可少的工具之一，所以 iOS 程序员的发展前景非常乐观。现在互联网公司的主要业务都依赖于 App 进行操作和发展。此外，移动互联网已经深入到生活的方方面面，现在仍然有大量的公司业务只能在计算机端办理，非常不便利，这也是手机端业务的机遇和挑战。整体的市场需求对于移动开发是非常巨大的。

3．iOS 程序员有哪些可供选择的职业发展道路？

成为在某一领域专精的优秀 iOS 专家。

4．企业在招聘时，对 iOS 程序员通常有何要求？iOS 程序员的日常工作是什么？

首先，iOS 开发需要扎实的计算机基础知识，包括基础的算法和数据结构，常用设计模式，网络通信协议，数据安全等；其次，要求 iOS 基础扎实，熟练使用常用的 UI 组件和网络组件。优秀的代码设计能力，避免开发中犯一些低级错误；了解各个常用框架的实现原理、

在学习 Objective-C 语言的基础上，也要尽快学习 Swift 语言。目前国内外企业都开始使用 Swift 语言作为开发语言，尤其是新的产品，基本都转向 Swift 语言了，但是对于 iOS 岗位的应聘者，依然是要求精通 Objective-C 语言。

经验技巧 17 React Native 和 Weex 重要吗

在传统开发中，当需要开发一款 App 的时候，往往需要在各个平台上，例如安卓平台、iOS 平台和 Web 平台，都开发一款对应的 App，我们将其称为“原生开发”。“原生开发”会给开发带来许多的问题，首先是开发人员增多和开发成本增加的问题，每个平台都需要有一名开发人员，而每增加一名开发人员就提高了开发成本；其次是还要保证不同平台之间功能的一致性，这给测试人员也带来了更多工作量；而最大的问题在于“原生开发”的周期长，复杂度高，这往往会造成产品难以在预期时间内完成。为了解决这种高成本的“原生开发”，诞生了两种代替原生开发的新技术：React Native 和 Weex。

什么是 React Native 开发？

React Native 是 Facebook 在 2015 年 3 月开源的一个跨平台 UI 框架，其理念是既拥有“原生开发”的用户体验，又保留 React（React 是 Facebook 2013 年开源的 Web 开发框架）的开发效率，这无疑迎合了业界的痛点。它的设计者 Occhino 不强求写一份 React Native 代码来同时支持多个应用平台，而是希望在不同的平台上通过编写 React Native 代码来支持各个平台，因此他提出了“Learn once，write anywhere”口号，并没有像 Java 设计的那样“wirte once，run anywhere”。React Native 底层的实现其实依赖于 JavaScript，通过 JavaScript 引擎来调用原生代码，从而实现页面的渲染和数据的绑定。React Native 不仅解决了跨平台问题，还解决了客户端动态更新困难的问题。React Native 使用热更新方式来动态更新应用，解决了客户端更新麻烦的问题，特别是 iOS 端，每次更新都需要重新发布一个版本。React Native 通过将基础模块和业务模块一起打包成一个 JS Bundle（JavaScript 资源包），然后将这个 JS Bundle 放到服务器上，客户端通过下载服务器上的 JS Bundle 来实现更新，避免了重新发布应用。在业务频繁变化的情况下，动态更新就变得非常有用。图 1 是 React Native 的设计框架。

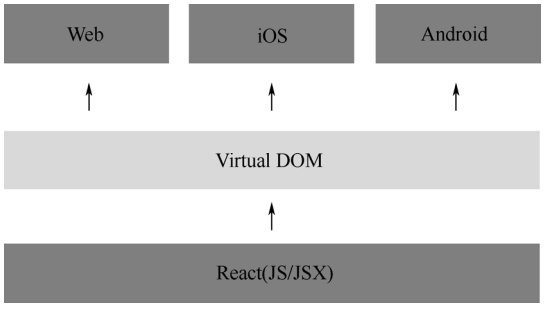

图 1　React Native 的设计框架

什么是 Weex 开发？

Weex 是阿里巴巴于 2016 年 6 月开源的一种用于构建跨平台应用的 UI 框架，其设计理念是希望客户端在具备动态发布能力的同时又不失良好的性能和用户体验。区别于 React

Native，为了彻底解决平台多样性带来的问题，Weex 大胆地提出了“write once，run everywhere”的口号，即写一份 Weex 代码可以在不同的平台上运行，它比 React Native 做得更彻底，更具革命性。Weex 又被称为 Vue Native，原因在于 Weex 是基于 Vue 来编写的，Vue 是目前非常流行的前端框架，它比 React 要简单、易用。Vue 使得 Weex 的开发更加简单，让更多的人能够快速上手，这也更迎合大众的口味。Weex 也是通过 JavaScript 引擎来调用原生代码实现页面的渲染和数据的绑定。和 React Native 一样，Weex 也是将代码打包成 JS Bundle 放到服务器上，客户端从服务器上下载 JS Bundle 来实现动态更新。不同的是，Weex 只打包业务模块，基础模块则留在了客户端的 Weex SDK 中，所以打包后的 JS Bundle 体积非常小，更加便于更新。图 2 是 Weex 的设计框架。

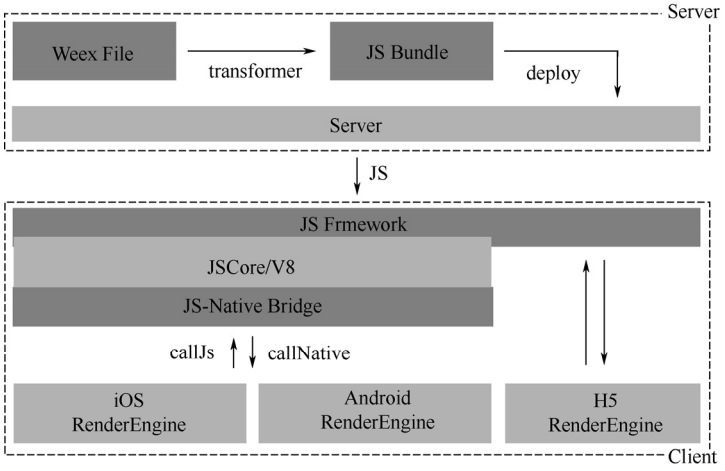

图 2　Weex 的设计框架

React Native 和 Weex 如何选择？

React Native 和 Weex 相同之处都是基于 JavaScript 渲染的，而 React Native 选择使用 React 编写前端界面，Weex 使用 Vue 编写前端界面。React Native 和 Weex 都能实现跨平台开发应用，从性能上来看，两者几乎一样，因为都是通过 JavaScript 调用原生的代码实现页面渲染，而 Weex 比 React Native 更加容易学习和使用，编写一份代码即可在多个平台上运行，还能够实现增量更新，这些是 Weex 的优势；而 React Native 的优势在于拥有更活跃的开源社区，版本换代更快，遇到的问题能够更快地得到解决。React Native 和 Weex 的共同的缺点是对原生组件的支持不够完善，许多原生组件的功能无法使用；自定义能力差，如很难实现自定义跳转效果；很难同时良好地支持各个平台，兼容性不够好；用户体验不够友好等。至于选择哪个框架或方案更好，这个很难说，因为这两个框架都还有很大的提升空间，我相信这种技术会是未来 App 开发的一个重要方向。没有最好的选择，只有更适合的选择，选择符合当前业务的方案才是最好的选择。

经验技巧 18　企业对 iOS 开发者的要求有哪些

通过各知名大公司对 iOS 开发工程师的招聘要求，可以看出对于非应届生基本的要求是

工作经验丰富，技能过硬，有一定工作成果等。针对应届毕业生，企业更看重学生的专业基础、知识面、工作潜力和可培养性、思维能力、编程能力以及数据结构和数学理论的理解掌握等等。总结企业对应聘者的要求主要包括但不限于以下几个方面：

（1）对语言的掌握程度

iOS 开发主要还是以 Objective-C 语言为主，虽然新出的 Swift 语言很流行，但经过多年的积累，尤其是大公司已经形成了一套完备的开发体系，Objective-C 语言还是多数公司已有 iOS 项目的主要语言，对于要求使用 Swift 语言的招聘需求暂时还较少。相对来说，Objective-C 作为苹果开发最原始的语言，学习难度较大，由于它是 C 语言的一个超集，对于有 C/C++语言基础的开发者学起来会容易得多，但对于 Objective-C 中的一些常用的语言特性和语法以及思想仍然需要努力去体会、去掌握。例如，Objective-C 中的协议（Protocol）和代理（Delegate）、类别（Category）、KVC 与 KVO、代码块（Block）、运行时特性（Runtime）、封装简化后的内存管理机制和多线程编程等，掌握后都会成为开发过程中的常用利器。当然对于初学 iOS 开发的人来说，还是推荐直接学习 Swift 语言，因为这是一门基于 C 语言和 Objective-C 语言优化后的新型语言，同时加入了顺应语言发展和期望的一些新特性，对于初学者来说比较友好、有趣，而且 iOS 8 以后是支持 Swift 和 Objective-C 混编的，也就是说 Swift 可以调用 Objective-C 的代码、使用已有 Objective-C 的第三方框架等，Objective-C 使用者也很容易转移到 Swift 中来。

（2）UI 界面的开发能力

包括基本的 UI 框架的熟练使用和灵活应用，AutoLayout 可视化界面的开发，更高要求是利用基本组件封装开发新的常用组件，甚至是对 UI 组件的优化等。另外，要理解界面和主线程的关系，熟知优化界面流畅性的一些经验和技巧，以及屏幕的动态适配等。

（3）设计模式和数据结构的掌握以及经验

这些直接决定了开发者的代码编写规范和质量，也会有关团队合作效率。要习惯和理解 iOS 开发中核心的 MVC 设计模式，并了解 MVC 设计模式的局限性，和因此而生的 MVVM 设计模式的结构及其优点。此外要熟知 Cocoa 框架中广泛存在的单例类及单例的实现方法，并不断实践深入理解其他在开发中经常接触的像观察者模式、代理模式、组合模式、工厂模式等设计模式的应用场景和它们的优势与作用。

（4）SDK 的接入和开发经验

iOS 开发中经常需要 SDK 的接入，如社交分享模块的接入，第三方登录验证接入等，熟悉常见的像 ShareSDK 等第三方高效接入平台的使用，熟悉第三方平台接入的一般流程和一些常见问题。自己的软件平台可能也会提供通用的 SDK 接口供其他厂家应用接入使用，这就需要懂得如何进行 SDK 库的封装和对外接口的设计开发。

（5）对网络编程的要求

包括基本的 HTTP 请求、数据安全性设计、数据的加解密，以及 TCP/IP 等协议的理解，甚至协议的设计实现等。了解 iOS 中基本的数据请求的实现方法，熟悉像 AFNetworking 等第三方网络编程框架的使用。

（6）对应用数据库（SQLite）的开发要求

了解数据库的基本原理和基本 SQL 语句的使用，熟悉 iOS 开发中数据持久化的常用手段，包括基本的偏好数据存储、数据归档与反归档、基本数据类型的本地存储，以及 SQLite 本地

数据库和 CoreData 的用法。

（7）对主流框架的熟悉程度

实际开发为了提高开发效率，会使用一些主流的第三方封装库，如 MJExtension、NFNetworking、MBPorgressHUB、MJRefresh 等，也经常会使用根据公司业务需要自行开发的框架组件。另外，要懂得如何用 CocoaPods 高效管理和使用大量的第三方库。

（8）对内存管理的理解和多线程编程的理解和使用

内存管理和多线程编程属于进阶阶段的内容，需要熟知 Objective-C 中的内存管理原则，熟悉引用计数的原理，了解如何避免内存泄漏与循环引用等问题的方法，以及如何检测内存问题的出现。学习多线程编程先要理解基本的概念，然后要熟悉 iOS 中的 3 种多线程实现方法，包括 NSThread、GCD 和 NSOperation，理解它们的区别和应用场景。

（9）Apple Store 发布经验

一款软件从开发完成到发布会有更多的挑战，尤其软件的质量优化、代码优化和 bug 清除等，如何提高软件的鲁棒性也是难点。另外，Apple 官方对上线软件作品的审核盐分严格，审核包括代码安全、资源格式、用户隐私、伦理道德、内容健康等各个方面，解决自身作品的各种问题使得软件顺利发布也是对开发者能力的很好验证。

（10）项目经历

项目是最容易考察一个应聘者能力水平的，因此应聘者要注意积累有深度的项目经历，同时要深入理解自己在项目中参与的部分，总结思考，有自己的理解和探索创新，包括实现思路、优化方法、遇到的难点、如何解决、项目中印象最深刻的部分以及最能体现自己水平的部分等。

如果有开源的 GitHub 项目，不错的技术博客等也是很大的加分项。

经验技巧 19　iOS 开发招聘有哪些要求

这里精选了部分知名企业的招聘要求，求职者可以通过参考企业的招聘需求了解企业开发中需要的开发者的能力和素质，了解 iOS 开发技术中的重点以及个人需要掌握的技术点，从而有所侧重地去弥补或加强自身的技术盲点或薄弱点，提高自身的核心竞争力。

表 2 是某知名互联网公司 1 对 iOS 开发工程师的招聘要求。

表 2　某知名互联网公司 1 对 iOS 开发工程师的招聘要求

学历要求	本科
岗位描述	1．独立完成 iPhone、iPad 客户端程序的开发 2．根据产品需求开发相关的移动产品 3．验证和修正测试中发现的问题
岗位要求	1．熟悉 Cocoa Touch、CoreData、iOS Runtime，精通 OS X、iOS 下的并行开发、网络、内存管理、GUI 开发 2．拥有很好的设计模式和思维，熟悉面向对象编程，图形界面开发 3．学习能力强，强烈的责任心，具有较强的沟通能力及团队合作精神 4．跨平台、多终端开发经验，encrypt/decrypt，HTTP client/server，graphics.优先 5．多年 iOS 客户端开发经验，熟悉 REST Application 的开发，有成功案例 6．对 iOS 的 UI 控件有优化经验者优先；有前端开发经验者优先 7．已在 App Store 发布过作品者优先

表 9 是某知名互联网公司 8 对 iOS 软件开发的招聘要求。

表 9　某知名互联网公司 8 对 iOS 软件开发的招聘要求

岗位要求	1．了解基本计算机理论，对数据结构、基本算法熟练掌握，具备基本的算法设计能力，有良好的编程习惯，代码风格和自己的技术想法，熟练掌握各种问题的搜索技巧 2．具备扎实的 C/C++语言基础，熟练掌握 Objective-C 编程。了解 SQL，熟悉网络编程、多线程、图形界面编程、熟悉 TCP、UDP、HTTP 协议 3．熟悉 iOS 事件机制，自定义和扩展 iOS UI 控件、对 Cocoa/UIKit 框架及 iOS SDK 有深入理解。熟练掌握常用开发框架以及 block，GCD 等常用技术，能独立进行应用模块的设计和实现。熟悉 XMPP，CoreData，AFNetWorking 或 ASIHTTPRequest，有越狱应用开发经验优先 4．掌握 iOS，Mac 或 UNIX 系统工作机制，精通 Xcode 工具系列，包括 Interface Builder 和 Instruments。能熟练使用 Mac 上的各种工具，能够通过工具或者脚本改善工作效率和质量 5．大专以上学历计算机相关专业，一年以上的 iOS 开发工作经验或有 App　Store 上架作品优先 6．丰富的 Apple 产品使用经验，熟悉 Apple 应用程序的设计理念 7．具备一定的沟通能力，能够清晰地表达自己的想法

表 10 是某知名互联网公司 9 对 iOS 软件工程师的招聘要求。

表 10　某知名互联网公司 9 对 iOS 软件工程师的招聘要求

岗位描述	1．参与公司产品开发以及加入设计团队优化 iOS 平台上软件的用户体验 2．使用最新的 iOS 编程技术实现传统原生用户界面的开发 3．创建可复用的和公司平台对接的 iOS 软件组件 4．分析优化 UI 界面和后端应用代码的效率和性能
岗位要求	1．计算机相关背景本科或研究生 2．有出色的面向对象软件开发经验 3．有使用 Objective-C 语言以及 Cocoa 等框架创建 iPhone 或 iPad 平台复杂应用的经验 4．有从用户界面到系统级别的移动应用开发能力 5．有大型复杂代码的理解和 debug 能力 6．有设计简洁可维护的 API 的经验 7．有多线程编程经验 8．有写单元测试以及可测试代码的经验 9．了解关于 iOS SDK 性能优化的工具和优化技术 10．出色的问题解决能力、辩证思考能力和交流技能

表 11 是某知名互联网公司 10 对 iOS 开发工程师的招聘要求。

表 11　某知名互联网公司 10 对 iOS 开发工程师的招聘要求

岗位描述	1．负责 iOS 客户端产品的开发和维护 2．根据项目任务计划独立按时完成软件高质量编码和测试工作 3．与团队成员进行有效沟通，进行技术风险评估，项目时间评估
岗位要求	1．三年及以上 iOS 项目开发经验，本科及以上学历，计算机相关专业优先 2．精通 Objective-C 语言；熟悉 C/C++语言者优先考虑 3．熟悉 iOS 框架以及各种特性，深刻理解常用设计模式 4．能主动研究前沿技术，将新技术转化为实际产品 5．有较好的学习能力和沟通能力，有创新能力和责任感，对移动端产品有浓厚的兴趣 6．熟悉团队协作模式，在项目中使用过 Git 做版本管理 7．有较好的学习能力和沟通能力，有创新能力和责任感

经验技巧 20　iOS 技术岗位面试精选

技术岗位的面试根据距离因素可能是现场面试、电话或者视频面试，面试时间从半个小时左右到一两个小时不等，多数是半小时左右。面试开始基本都是先自我介绍，主要确认一

下面试者本人基本信息和基本的表达能力，面试者要讲清楚自己的学历背景和工作技术背景，求职的来由和求职欲望等。

面试过程中首先是挑选简历中的内容进行深入提问，了解和核实简历信息的真实性，重点是简历上的项目经验。然后一般会问一两个简单的小算法，考验思维能力和灵活应变能力等。另外就是详细问 iOS 技术相关的问题了，侧重于基础，也就是本书中重点整理总结的内容。

面试最后，面试官都会问面试者有什么问题需要问的，有的话面试官会耐心给出回答，问完面试则结束。

以下这里从网上搜集了一些相对完整的面试经验，供参考体会，可以从中看出面试中的考察侧重点和考察方向等，以便读者有目的进行准备和完善自身知识体系。

某知名互联网公司一面（电话面试）

提前一天收到面试电话通知，第二天视频面试（实际是先笔试后面试）。

刚上来笔试写两个小算法：

1）判断 IP 地址合法性以及优化方法（正则表达式等）。

2）单链表逆置。

iOS 面试（全是 iOS 相关）：

1）cache 缓存机制。

2）隐式动画和显示动画的区别。

3）block 内部结构和原理。

4）tableView 的高度预估机制。

5）autorealease 的原理和应用场景。

6）KVC 和 KVO。

7）tableView 优化方法。

8）property 属性。

9）多线程 GCD 和 NSOperation 的比较。

10）NSCopy 协议。

11）pushviewcontroller 后 view 的释放时机（viewDidload）。

12）weak 的应用场景，和 assign 的区别；weak 的实现原理。

13）block 内部修改外部变量，_block 修饰符原理，闭包。

14）有没有看过哪些优秀第三方库的源码，例如 SDWebImage 中的缓存机制是如何实现的。

某知名互联网公司一面（电话面试）

1）开始先自我介绍。

2）根据简历了解个人研究方向的问题。

3）说一个自己最近印象深刻的项目，然后根据做的项目说一下里面的关键技术点，最后引导问了其中的安全问题，如何加密解密、数字签名、服务器配置等。

4）Objective-C 中的属性和实例变量的区别。

5）UIView 和 CALayer 的区别，还用过哪些 Layer 类（CATextLayer 等）？

6）Category 如何扩展，有什么好处？为什么不能扩展属性？

面试笔试技术攻克篇

面试笔试技术攻克篇主要针对近 3 年多家顶级 IT 企业的面试笔试真题而设计，这些企业涉及面非常广泛，面试笔试真题难易适中，覆盖面广，非常具有代表性与参考性。本篇对这些真题以及其背后的知识点进行了深度剖析，并且对部分真题进行了庖丁解牛式的分析与讲解，针对真题中涉及的部分重点、难点问题，本篇都进行了适当的扩展与延伸，力求对知识点的讲解清晰而不紊乱，全面而不啰唆，使得读者能够通过本书不仅获取到求职的知识，同时更有针对性地进行求职准备，最终能够收获一份满意的工作。

第 1 章　iOS 开发基础概念

iOS 开发中有一些非常重要的基础概念，这些概念既能体现开发者对于 iOS 技术的理解程度，也是对知识面的一种考察。因此，开发者在学习和实践的过程中对一些核心概念应该刨根问底，及时总结，从而做到融会贯通，同时要扩展相关的知识面。这里总结了一部分比较重要的知识点，读者应该据此夯实基础知识，多总结思考，深入理解。

1.1 概念

1.1.1 什么是“应用瘦身”

“应用瘦身”（App thinning）是美国苹果公司自 iOS 9 发布的新特性，它能对 Apple Store 和操作系统进行优化，它根据用户的具体设备型号，在保证应用特性完整的前提下，尽可能地压缩和减少应用程序安装包的体积，也就是尽可能减少应用程序对用户设备内存的占用，从而减小用户下载应用程序的负担。App thinning 的实现主要有以下 3 种方法：Slicing、Bitcode 和 On-Demand Resources。以下将对这 3 种方法进行介绍。

1．Slicing

在开发者将完整的应用安装包发布到 Apple Store 之后，Apple Store 会根据下载用户的目标设备型号创建相应的应用变体（variants of the app bundle）。这些变体只包含可执行的结构和资源等必要部分，而不需要让用户下载开发者提供的完整安装包。图 1-1 展示了从开发者使用 Xcode 开发完整应用并发布到 Apple Store 后被用户下载到不同设备上的流程。

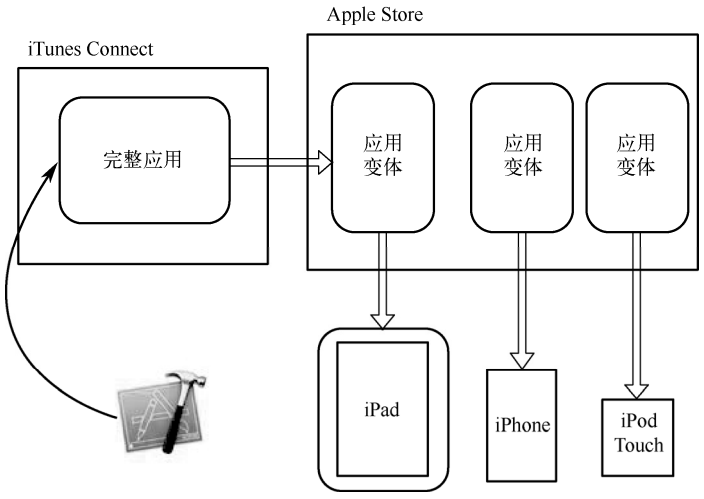

图 1-1　应用发布下载流程图

1.1.4 什么是响应者链

iOS 中的响应者链（Responder Chain）是用于确定事件响应者的一种机制，其中的事件主要指触摸事件（Touch Event），该机制和 UIKit 中的 UIResponder 类紧密相关。响应触摸事件的都是屏幕上的界面元素，而且必须是继承自 UIResponder 类的界面类（包括各种常见的视图类及其视图控制器类，如 UIView 和 UIViewController）才可以响应触摸事件。

一个事件响应者的完成主要经过两个过程：hitTest 方法命中视图和响应者链确定响应者。hitTest 方法首先从顶部 UIApplication 往下调用（从父类到子类），直到找到命中者，然后从命中者视图沿着响应者链往上传递寻找真正的响应者。

如图 1-4 所示界面结构，最顶部是一个 UIWindow 窗口，其下对应一个唯一的根视图，根视图上可以不断叠加嵌套各种子视图，构成一棵树。需要注意的是，父节点里面嵌套着子节点，即子节点的 frame 包含在父节点的 frame 内，但是子节点不一定是父节点的子类，它们是组合关系而非继承关系。

视图树从屏幕视角看上去可能是如图 1-5 所示的层次结构。

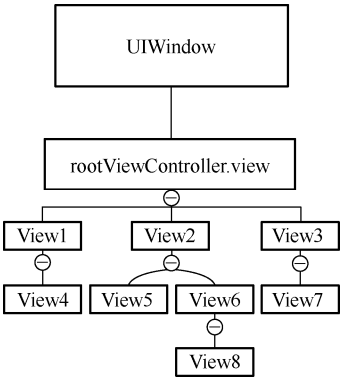

图 1-4　视图节点树形结构

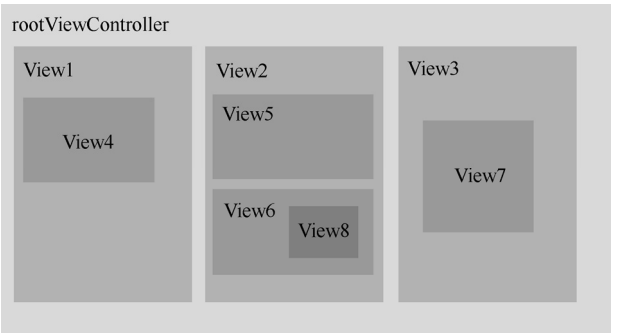

图 1-5　视图树的屏幕效果

1．命中测试

命中测试（hitTest）主要会用到视图类的 hitTest 函数和 pointInside 函数。其中，前者用于递归寻找命中者，后者则是检测当前视图是否被命中，即触摸点坐标是否在视图内部。当触摸事件发生后，系统会将触摸事件以 UIEvent 的方式加入到 UIApplication 的事件队列中，UIApplication 将事件分发给根部的 UIWindow 去处理，UIWindow 则开始调用 hitTest 方法进行迭代命中检测。

命中检测具体迭代的过程为：如果触摸点在当前视图内，那么递归对当前视图内部所有的子视图进行命中检测；如果不在当前视图内，那么返回 NO 停止迭代。这样最终会确定屏幕上最顶部的命中的视图元素，即命中者。

2．响应者链

通过命中测试找到命中者后，任务并没有完成，因为最终的命中者不一定是事件的响应者。所谓的响应就是开发中为事件绑定的一个触发函数，事件发生后执行响应函数里的代码，例如通过 addTarget 方法为按钮的单击事件绑定响应函数，在按钮被单击后能及时执行想要执

行的任务。

一个继承自 UIResponder 的视图要想能响应事件，还需要满足如下一些条件：

1）必须要有对应的视图控制器，因为按照 MVC 模式响应函数的逻辑代码要写在控制器内。另外，userInteractionEnabled 属性必须设置为 YES，否则会忽视事件不响应。

2）hidden 属性必须设置为 NO，隐藏的视图不可以响应事件，类似的 alpha 透明度属性的值不能过低，低于 0.01 接近透明也会影响响应。

3）最后要注意保证树形结构的正确性，子节点的 frame 一定都要在父节点的 frame 内。

响应者链的结构和上面的树形结构是对应的，是命中者节点所在的树的一条路径（加上视图节点对应的视图控制器）。命中者的下一个响应者是它的视图控制器（如果存在的话），如果命中者不满足条件则不能响应当前事件，此时会沿着响应者链往上寻找，看父节点能否响应，直到完成事件的响应。如果到了响应者链的顶端 UIWindow 事件依然没有被响应，那么将事件交给 UIApplication 结束响应循环，在这种情况下这个事件就没有实质的响应动作发生。响应者链过程如图 1-6 所示。

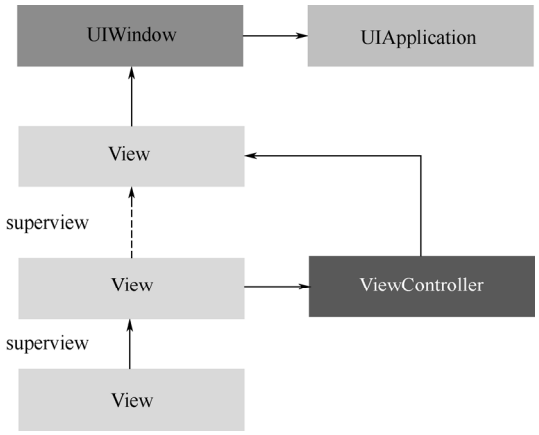

图 1-6　响应者链过程示意图

1.1.5　什么是“懒加载”

“懒加载”（Lazy loading）也被叫作“延迟加载”，它的核心思想是把对象的实例化尽量延迟，直到真正用到的时候才将其实例化，这样做的好处是可以减轻大量对象在实例化时对资源的消耗，而不是在程序初始化的时候就预先将对象实例化。另外，“懒加载”可以将对象的实例化代码从初始化方法中独立出来，从而提高代码的可读性，以便于代码能够更好地被组织。

最典型的一个应用“懒加载”的例子是在对象的 getter 方法中实例化对象的时候。例如 getter 方法被重写，使得在第一次调用 getter 方法时才实例化对象并将实例化的对象返回。判断是否是第一次调用 getter 方法可以通过判断对象是否为空来实现。“懒加载”的 getter 方法的实现模板如下：

```
/* getter*/
- (NSObject *)object {
    if (!_object) {
        _object = [[NSObject alloc] init];
    }
    return _object;
}
```

这种实现方法的缺点是使得 getter 方法产生副作用，也就是破坏了 getter 方法的纯洁性。因为按照约定和习惯，getter 方法就是作为接口简单地将需要的实例对象返回给外部，这里对

第 2 章　Objective-C 语言基础

Objective-C 是用来开发 OS X 和 iOS 软件的最原始语言，它是 C 语言的一个超集，具有面向对象的语言特性，同时它提供了强大的运行时动态语言特性。Objective-C 保留继承了 C 语言的语法、基本类型和流控制等，又在此基础上添加了定义类和方法的新的语法，具有动态类型、动态绑定和动态加载等动态特性，它将一些工作推迟到运行时，大大增强了编程的灵活性，也使其更加强大。

Swift 是一种新型语言，用于开发 iOS、OS X、Watch OS 和 tvOS 应用，它充分继承了 C 语言和 Objective-C 语言的优点，同时又摆脱了 C 语言的兼容性问题。Swift 采用安全的编程模式，它添加了一些新的特性，使得编程更加简单、灵活和有趣。Objective-C 程序员可以很轻松地转移到 Swift 上来，因为两者具有诸多共性，同时 Swift 对于新手程序员也比较容易学习。

本章首先选取了一部分有关 Objective-C 语言特性的问题进行分析，然后分别整理了关于 Objective-C 语言中的数据类型的问题以及运算符和表达式相关的问题。通过这些问题的分析，读者可以了解到 Objective-C 语言基础特性的重点部分，加深对基础部分的深入理解。

2.1　Objective-C 语言基础特性

Objective-C 作为 C 语言的超集，在继承 C 语言特性的基础上增添了更多的特色用法和自己的语法，而这也使其成为一门专门用于苹果平台开发的独具特色的面向对象语言。

2.1.1　Objective-C 的优缺点有哪些

Objective-C 的优缺点见表 2-1。

表 2-1　Objective-C 的优缺点

优　点	缺　点
1）Objective-C 是 C 语言的超集，在 C 语言的基础上衍生了很多新的语言特性，封装得很完善而且方便使用，大大降低了编程复杂度，因此开发中使用起来会感觉方便高效	1）不支持命名空间（都是通过加一些像 NS 或者 UI 这样的命名前缀来达到用命名空间防止命名冲突的作用，但这样会使变量的命名更长）
2）Category（类别）的使用，可以快速扩展类的方法，同时使扩展的功能模块之间互不影响	2）不支持运算符重载
3）Posing（扮演）特性，[ParentClass poseAs: [ChildrenClass class]];该语言特性使得父类无须定义和初始化子类对象，即可通过父类扮演子类进行操作	3）不支持多重继承（C++语言通过 virtual 关键字防止二义性的出现，实现多重继承）
4）动态语言特性，动态类型、动态绑定和动态加载等，将类型确定、方法调用和资源加载等任务推迟到运行时，大大提高了编程灵活度	4）使用动态运行时类型，所有的方法都是通过消息传递机制方法调用，有其动态的优势，同时也使很多编译时的优化方法无法使用降低了性能，例如：内联方法等
5）指针：Objective-C 保留了 C 语言强大的指针特性	
6）Objective-C 与 C/C++可在.mm 文件中进行混合编程，灵活度更高	

6．Objective-C 和 Swift 语言中 UIImageView 的使用对比

（1）Objective-C

```
UIImageView *myImage = [[UIImageView alloc] initWithImage: [UIImage imageNamed:@”tiger.png”]];
[self.view addSubview:myImage];
myImage.center = CGPointMake(150, 200);
myImage.frame = CGRectMake(0, 0, 50, 25);
```

（2）Swift

```
let myImage = UIImageView(image: UIImage(named: "tiger.png"))
view.addSubview(myImage)
myImage2.frame = CGRect(x:0, y:0, width:50, height:25)
myImage2.center = CGPoint(x:150, y:200)
```

2.1.3 Foundation 对象与 Core Foundation 对象有什么区别

Foundation 对象是 Objective-C 对象，使用 Objective-C 语言实现；而 Core Foundation 对象是 C 对象，使用 C 语言实现。两者之间可以通过__bridge、__bridge_transfer、__bridge_retained 等关键字转换（桥接）。

Foundation 对象和 Core Foundation 对象更重要的区别是 ARC 下的内存管理问题。在非 ARC 下两者都需要开发者手动管理内存，没有区别。但在 ARC 下，系统只会自动管理 Foundation 对象的释放，而不支持对 Core Foundation 对象的管理。因此，在 ARC 下两者进行转换后，必须要确定转换后的对象是由开发者手动管理，还是由 ARC 系统继续管理，否则可能导致内存泄漏问题。

下面以 NSString 对象（Foundation 对象）和 CFStringRef 对象（Core Foundation 对象）为例，介绍两者的转换和内存管理权移交问题。

1）在非 ARC 下，NSString 对象和 CFStringRef 对象可以直接进行强制转换，都是手动管理内存，无须关心内存管理权的移交问题。

2）在 ARC 下，NSString 对象和 CFStringRef 对象在相互转换时，需要选择使用__bridge、__bridge_transfer 和__bridge_retained 来确定对象的管理权转移问题，三者的作用语义分别如下：

① __bridge 关键词最常用，它的含义是不改变对象的管理权所有者，本来由 ARC 管理的 Foundation 对象，转换成 Core Foundation 对象后依然由 ARC 管理；本来由开发者手动管理的 Core Foundation 对象转换成 Foundation 对象后继续由开发者手动管理。示例代码如下：

```
/* ARC 管理的 Foundation 对象 */
NSString *s1 = @"string";
/* 转换后依然由 ARC 管理释放 */
CFStringRef cfstring = (__bridge CFStringRef)s1;
/* 开发者手动管理的 Core Foundation 对象 */
CFStringRef s2 = CFStringCreateWithCString(NULL, "string", kCFStringEncodingASCII);
/* 转换后仍然需要开发者手动管理释放 */
NSString *fstring = (__bridge NSString*)s2;
```

② __bridge_transfer 用在将 Core Foundation 对象转换成 Foundation 对象时，用于进行内存管理权的移交，即本来需由开发者手动管理释放的 Core Foundation 对象在转换成 Foundation 对象后，交由 ARC 来管理对象的释放，开发者不用再关心对象的释放问题，因为不会发生内存泄漏。示例代码如下：

```
/* 开发者手动管理的 Core Foundation 对象 */
CFStringRef s2 = CFStringCreateWithCString(NULL, "string", kCFStringEncodingASCII);
/* 转换后改由 ARC 管理对象的释放，不用担心内存泄漏 */
NSString *fstring = (__bridge__transfer NSString*)s2;
// NSString *fstring = ( NSString*)CFBridgingRelease(s2); //另一种等效写法
```

③ __bridge_retained 用在将 Foundation 对象转换成 Core Foundation 对象时，进行 ARC 内存管理权的剥夺，即本来由 ARC 管理的 Foundation 对象在转换成 Core Foundation 对象后，ARC 不再继续管理该对象，需要开发者自己进行手动释放该对象，否则会发生内存泄漏。示例代码如下：

```
/* ARC 管理的 Foundation 对象 */
NSString *s1 = @"string";
/* 转换后 ARC 不再继续管理，需要手动释放 */
CFStringRef cfstring = (__bridge__retained CFStringRef)s1;
// CFStringRef cfstring = (CFStringRef)CFBridgingRetain(s1); //另一种等效写法
```

2.1.4 Objective–C 中的类方法和实例方法有什么本质区别和联系

在比较类方法和实例方法的区别之前，先要明确 Objective-C 中的类对象和实例对象的概念，开发中定义的类自身也是一个对象，称为类对象，保存该类的成员变量、属性列表和方法列表等。类对象经 alloc 和 init 实例化后成为实例对象。实例对象、类对象和元类的底层结构如图 2-1 所示。

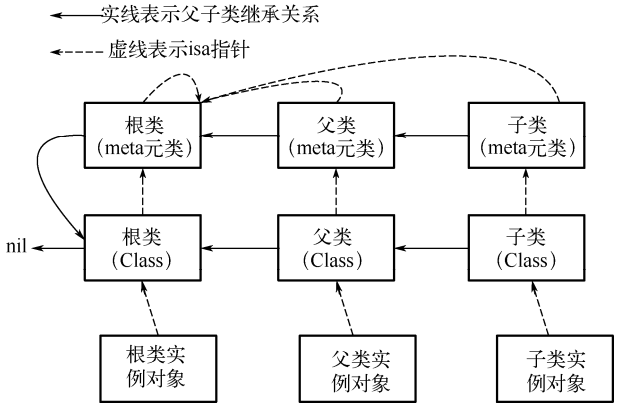

图 2-1 实例对象、类对象和元类的关系结构图

1）类方法属于类对象，用“+”号修饰，它类似于 C 语言中的静态方法，类方法列表定义在类对象的元类中，通过 isa 指针找到；实例方法属于实例对象，用“-”号修饰，实例方法列表定义在实例对象的类对象中，通过 isa 指针找到。

第 3 章　Objective–C 语言的高级特性

本章重点介绍一些 Objective–C 的高级特性，包括 Objective–C 对象的内存分布、开发中频繁用到的类别（Category）和扩展（Extension）的特性、Objective–C 语言的动态性和多态性、对方法进行动态修改的 Swizzling 魔法、类和对象的区别以及属性的本质和其他一些未分类的重要问题。理解本章的内容是深入理解 Objective–C 语言的直接通道，也是 iOS 开发进阶的必经之路。

3.1 Objective–C 中的属性

属性（property）是 Objective–C 中封装对象数据的一个重要特性。

在 Objective–C 中，对象通常情况下会把其所需要的数据保存为各种实例变量。实例变量一般通过“存取方法”（access method）进行访问。其中，“获取方法”（getter）用于读取变量值，而“设置方法”（setter）用于写入变量值。

Objective–C 中的属性特性为访问实例变量提供了一种简洁的抽象机制，编译器会自动创建出一套与属性相关的存取方法，用以访问给定类型中具有给定名称的变量，这个创建的过程叫作“自动合成”。除此之外，在编译阶段，编译器会自动向类中添加适当类型的实例变量，并且在属性名前面加下画线，以此作为实例变量的名字。示例代码如下：

```
@interface Person : NSObject
@property (nonatomic, strong) NSString *name;
@property (nonatomic, assign) NSInteger age;
@end
```

对于 Person 类的使用者来说，上面这段代码等效于下面这种写法：

```
@interface Person : NSObject
- (void)setName:(NSString *)name;
- (NSString *)name;
- (void)setAge:(NSInteger)age;
- (NSInteger)age;
@end
```

要访问属性时，可以使用“点语法”，编译器会将“点语法”自动转换为存取方法的调用，使用“点语法”的效果等同于调用存取方法。

需要强调的一点是，如果使用了属性但是又不想让编译器自动合成存取方法，那么有下面两种方法：

1）自己实现存取方法。如果只实现了其中一个存取方法，那么另外一个还是会由编译器来合成。

2）使用@dynamic 关键字。它会通知编译器：不要自动创建实现属性所用的实例变量，

也不要为其创建存取方法。这种方式一般与运行时动态添加方法配合使用。

深入理解 Objective-C 中的属性

属性把类对象中的实例变量及其读写方法统一封装起来，是对传统 C++中要重复为每个变量定义读写方法的一种封装优化，Objective-C 将这些实例变量封装为属性变量，系统可自动为属性生成 getter 和 setter 读写方法，同时仍然允许开发者利用读写语义属性参数(readwrite 等)、@synthesize 和@dynamic 关键词去选择性自定义读写方法或方法名。

1．传统 C++类实例变量的定义形式

原本的类实例变量定义形式如下，类(Class)是实例变量和方法的集合，变量的定义可以通过 public、private 和 protected 等语义关键词来修饰限定变量的定义域，实现对变量的封装。Objective-C 中仍然保留这种定义方法，其中关键词改为：@public、@private、@protected 以及@package 等，在头文件中的变量默认是@protected，在.m 文件中的变量默认是@private。

```
@interface Test : NSObject {
    @public                 // 声明为共有变量
    NSString *_name;
    @private                // 限制为私有变量，.m 实现文件中定义变量的默认类型
    NSString *_major;
    @protected              // 限制为子类访问变量，头文件中定义变量的默认类型
    NSString *_accupation;
    @package                // 包内变量，只能本框架内使用
    NSString *_company;
}
```

以上这种传统定义形式的缺点如下：

1）每个变量都要手动编写 getter 和 setter 方法，当变量很多时，类中会出现大量的这些读写方法的代码，同时这些读写方法的形式是相同的，因此会产生代码冗余。Objective-C 中属性变量的封装就是将这些方法的定义封装起来，减少大量形式重复的方法定义。

2）这种类变量定义的方式属于“硬编码”，即对象内部的变量定义和布局已经写死在了编译期，编译后不可再更改，否则会出错。因为上面所谓的“硬编码”，指的是类中的变量会被编译器定义为距对象初始指针地址的偏移量（offset），编译之后变量是通过地址偏移来寻找的，如果想要在类中插入新的变量，那么必须要重新编译计算每个变量的偏移量；否则顺序被打乱会读取错误的变量。例如图 3-1 所示的例子，在编译好的对象中变量的前面插入新的变量：

插入后_occupation 的偏移量变了，因为现在是第三个指针，这时候按照编译器的结果访问就会出错。

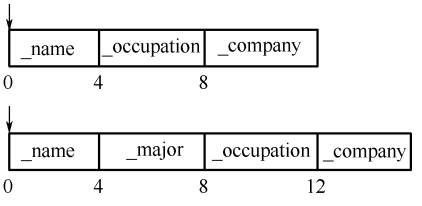

图 3-1 “硬编码”下变量内存分布

2．属性变量封装定义

（1）存取方法和变量名的自动合成

使用 Objective-C 的属性，编译器会自动按照 Objective-C 严格的存取方法命名规范自动生成对应的存取方法，通过存取方法就可以根据变量名去访问对应的变量，通过“点语法”访问变量其实就是调用了变量的存取方法（编译器会将“点语法”转换成存取方法的调用），也就是说通过属性定义的变量名成了存取方法名。此外，还会自动生成对应的实例变量名，由于自定

义的变量名与获取它的方法名是相同的，为了区分，实际的变量名在前面加下画线，另外虽然默认是加下画线的，但可以在实现文件中使用关键词@synthesize 自定义实际的变量名。下面例子中使用 property（属性）定义变量，编译器会自动生成对应的存取方法和加下画线的实例变量名，由于是编译期生成的方法，所以编译之前看不到。

```
@interface Test : NSObject
/* property 属性声明变量，编译期到时会自动生成获取方法：name 和设置方法：setName */
@property NSString *name;
@end
Test *test = [[Test alloc] init];
/** 通过点语法访问变量，等效于调用自动生成的存取方法访问变量：**/
/* 1.调用 test 的 setter 方法设置变量 */
test.name = @"sam";
/* 等效于： */
 [test setName:@"sam"];
/* 2.调用 test 的 getter 获取方法访问变量 */
NSString *s = test.name;
/* 等效于： */
NSString *s = [test name];
/* 3.name 成了存取方法的方法名，所以要想直接访问实例变量要使用自动生成的变量名，也就是
_name */
_name = @"albert";
s = _name;
```

（2）@synthesize 自定义变量名（特殊使用场景）

如果在 Test 的实现文件中使用@synthesize 关键字自定义实例变量名，那么就不能通过 _name 默认变量名来直接访问变量了，而是要使用自定义的名字，但实际为了规范和约定，@synthesize 自定义实例变量名的用法是不建议使用的（@synthesize 的原始用法是和@property 成对出现来自动合成指定属性变量的存取方法的）。

```
@implementation Test
/* 自定义实例变量名 */
@synthesize name = theName;
@end
```

现在要直接访问实例变量（不使用存取方法）就要通过自定义的变量名了：

```
/* 通过自定义的变量名访问，此时_name 已经不存在了 */
theName = @"albert";
s = theName;
```

注意一些旧版本的编译器@synthesize 和@property 是成对出现的，也就是说要手动使用 @synthesize 来合成相应的存取方法，否则不会自动合成（现在编译器默认会自动添加 @synthesize 自动合成存取方法）。

```
@synthesize name; // 旧版本手动指定要合成存取方法的变量
```

此时 set 方法名为：setName，变量名和 get 方法名都为 name，即 name 作为方法调用就

是方法名，作为变量直接取就是变量名。

```
name = @"Sam";                          // name 为变量名
NSString *oldName = [self name];        // name 为 get 方法名
```

（3）@dynamic 禁止存取方法自动合成

@dynamic 关键字是用来明确告诉编译器禁止自动合成属性变量的存取方法和加下画线的默认变量名。默认情况如果不用@dynamic 关键字，那么编译器就会自动合成那些没有定义的存取方法，而那些程序员已经定义了的存取方法不会再去合成，即程序员定义的存取方法优先级高。例如，如果此时程序员自定义了 setter 方法，那么编译器就会只自动合成 getter 方法，而不会再去合成已经定义了的 setter 方法。

```
@implementation Test
/* 禁止编译器自动生成存取方法 */
@dynamic name;
@end
```

此时，如果代码中依旧使用点方法，或者通过存取方法调用来访问 name，那么编译之前并没有异常，但编译之后由于在编译期编译器并没有自动合成存取方法，运行起来时，程序会在存取方法调用的位置处崩溃，因为调用了不存在的方法。

（4）不同属性特质修饰词的限制

通过在@property 后的括号内添加属性特质参数，也可以影响存取方法的生成。

```
@interface Test : NSObject
/* 括号内添加属性特质进行限制 */
@property(nonatomic, readwrite, copy) NSString *name;
@end
```

通常情况下，属性参数主要可以分为 3 类（见表 3-1）。

表 3-1　3 类主要的属性参数

原子性语义	atomic、nonatomic
读写语义	readwrite、readonly、getter、setter
内存管理语义	assign、weak、unsafe_unretained、retain、strong、copy

其中，最重要的是内存管理语义。要理解内存管理语义的作用和用法，首先要理解内存管理中的引用计数原理，也就是要理解 Objective-C 的内存管理机制。属性参数的内存管理语义是 Objective-C 中协助管理内存的很重要一部分。各种属性参数的含义和区别如下。

atomic、nonatomic：原子性和非原子性。原子性是数据库原理里面的一个概念，ACID 中的第一个。在多线程中同一个变量可能被多个线程访问甚至更改，进而造成数据污染，因此为了安全，Objective-C 中默认是 atomic，即会对 setter 方法加锁，相应的也会付出维护原子性（数据加锁解锁等）的系统资源代价。应用中如果不是特殊情况（多线程间的通信编程），那么一般还是用 nonatomic 来修饰变量的，不会对 setter 方法加锁，以提高多线程并发访问时的性能。

readonly、readwrite：readonly 表示变量只读，也就是它修饰的变量只有 get 方法而没有 set 方法；readwrite 既有 get 方法，也有 set 方法，可读亦可写。

getter = < getternname >，setter = < setternname >：可以选择性地在括号里直接指定存取方法的方法名，例如：

```
/* 更改默认的获取方法 name 为 getName */
@property(nonatomic, getter=getName, copy) NSString *name;
/* 之后要调用获取方法应使用上面指定的 */
s = [test getName];
```

assign：直接简单赋值，不会增加对象的引用计数，用于修饰非 Objective-C 类型，主要指基础数据类型（例如 NSInteger）和 C 数据类型（例如 int、float、double、char 等），或修饰对指针的弱引用。

weak：修饰弱引用，不增加引用对象的引用计数，主要可以用于避免循环引用，和 strong/retain 对应，功能上和 assign 一样简单，但不同的是用 weak 修饰的对象消失后会自动将指针置 nil，防止出现“悬挂指针”。

unsafe_unretained：这种修饰方式不常用，通过名字看出它是不安全的，为什么这么说呢？它和 weak 类似都是自己创建并持有对象，之后却不会继续被自己持有（引用计数没有+1，引用计数为 0 的时候会被自动释放，尽管 unsafe_unretained 和 weak 修饰的指针还指向那个对象）。不同的是，虽然在 ARC 中由编译器来自动管理内存，但 unsafe_unretained 修饰的变量并不会被编译器进行内存管理，也就是说既不是强引用也不是弱引用，生成的对象立刻就被释放掉了，出现了所谓的“悬挂指针”，所以不安全。

retain：常用于引用类型，是为了持有对象，声明强引用，将指针本来指向的旧的引用对象释放掉，然后将指针指向新的引用对象，同时将新对象的索引计数加 1。

strong：原理和 retain 类似，只不过在使用 ARC 自动引用计数时，用 strong 代替 retain。

copy：建立一个和新对象内容相同且索引计数为 1 的对象，指针指向这个对象，然后释放指针之前指向的旧对象。NSString 变量一般都用 copy 修饰，因为字符串常用于直接复制，而不是去引用某个字符串。

除了在属性变量前面加修饰词，开发中还会用到一些所有权修饰符，如__strong 和__weak。所有权修饰符和上面的修饰符有着对应关系，它们使用的目的和原理是一样的，可结合理解记忆。它们的对应关系如下：

1）__strong 修饰符对应于上面的 strong、retain 与 copy，强引用来持有对象，它和 C++语言中的智能指针 std::shared_ptr 类似，也是通过引用计数来持有实例对象。

2）__weak 修饰符对应于上面的 weak，同样它和 C++语言中的智能指针 std::weak_ptr 类似，也是用于防止循环引用问题。

3）__unsafe __unretained 修饰符对应于上面的 assign 和 unsafe_unretained，创建但不持有对象，可能导致指针悬挂。

常见面试笔试真题 1：Objective-C 中的属性和实例变量有哪些区别？

答案：从下面的代码中可以很明显地看出它们的区别：

```
@interface Test : NSObject {
```

```
@property (nonatomic, readonly, copy) NSString *name;
@end

/* Person.m 实现文件 */
/* continue 私有声明区域 */
@interface Person()
/* 让编译器再合成私有 setter 方法，其中 readwrite 可以省略，因为默认就是 readwrite */
@property (nonatomic, readwrite, copy) NSString *name;
@end

/* implementation 实现区域 */
@implementation Person
/* 测试 */
- (void)test {
    /* 下面两条语句等效，都是调用 setter 方法，但注意 setter 方法是私有的，只能在此处调用，在外部无法调用 */
    self.name = @"name";
    [self setName:@"name"];
}
@end
```

3.2 Objective-C 语言的多态性

3.2.1 什么叫多态

多态（Polymorphism）在面向对象语言中指同一个接口可以有多种不同的实现方式，Objective-C 中的多态则是不同对象对同一消息的不同响应方式，子类通过重写父类的方法来改变同一消息的实现，体现多态性。另外，C++中的多态主要是通过 virtual 关键字（虚函数、抽象类等）来实现，具体来说指允许父类的指针指向子类对象，使其成为一个更泛化、容纳度更高的父类对象，这样父对象就可以根据实际是哪种子类对象来调用父类同一个接口的不同子类实现。

举个简单例子来展示 Objective-C 的多态实现。假设有一个动物父类 Animal，其下有两个子类，一个是 Dog，一个是 Cat，父类有一个统一接口：shout，表示动物的叫声，父类对接口有一个默认实现，子类各自有自己的接口实现，继承关系如图 3-2 所示。

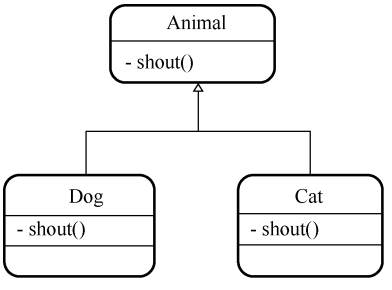

图 3-2 父子类继承关系

Animal 父类：

```
// Animal.h
@interface Animal : NSObject
/* 父类接口，动物叫声 */
- (void)shout;
@end
```

```
    NSLog(@"two parameters!");
}
```

多态调用：

```
[self test:1];                // output:one parameter!
[self test:1 andTwo:2]; // output:two parameter!
```

可以看出 Objective-C 可以通过参数个数实现函数重载，但是如果参数相同，那么无论参数和返回值类型相同与否，都无法编译通过。下面的定义是无法通过 Xcode 编译的。

```
- (void)test:(int)one;
- (int)test:(float)one; // Duplicate declaration of method 'test'
```

引申 1：Objective-C 的类可以多重继承吗？可以实现多个接口吗？重写一个类的方式用继承好还是类别好？为什么？

Objective-C 的类只支持单继承，不可以多重继承。

Objective-C 可以利用 protocol 代理协议实现多个接口，通过实现多个接口完成类似 C++ 的多重继承。在 Objective-C 中多态特性是通过 protocol（协议）或者 Category（类别）来实现的。protocol 定义的接口方法可以被多个类实现，Category 可以在不变动原类的情况下进行方法重写或者扩展。

重写一个类一般情况下使用类别更好，因为用 Category 去重写类的方法，仅对本 Category 有效，不会影响到其他类与原有类的关系。

引申 2：Cocoa 中有虚基类的概念吗？

Cocoa 中没有虚基类的概念。虚基类是 C++语言中为了解决多重继承二义性问题的，而 Objective-C 中只有单继承。

要实现类似 C++语言中的多继承，可以通过 protocal 来简单实现，因为一个类可以实现多个协议，类似于 Java 中一个类可以实现多个接口。

3.3 Objective-C 语言的动态性

Objective-C 语言的动态性主要体现在以下 3 个方面的内容。

1）动态类型（Dynamic typing）：运行时确定对象的类型。

2）动态绑定（Dynamic binding）：运行时确定对象的调用方法。

3）动态加载（Dynamic loading）：运行时加载需要的资源或者可执行代码。

1．动态类型

动态类型指对象指针类型的动态性，具体是指使用 id 类型将对象的类型推迟到运行时才确定，由赋给它的对象类型决定对象指针的类型。另外，类型确定推迟到运行时之后，可以通过 NSObject 的 isKindOfClass 方法动态判断对象最后的类型（动态类型识别）。也就是说，id 修饰的对象为动态类型对象，其他在编译器指明类型的为静态类型对象，通常如果不需要涉及多态还是要尽量使用静态类型（原因：错误可以在编译器提前查出，可读性好）。

2．动态绑定

动态绑定指方法确定的动态性，建立在动态类型的物质基础之上，具体指在 Objective-C 的消息分发机制支持下将要执行的方法推迟到运行时才确定，可以动态添加方法。也就是说，一个 Objective-C 对象是否调用某个方法不是在编译期决定的，方法的调用不和代码绑定在一起，而是到了运行时根据发出的具体消息而动态确定需要调用的代码。利用动态类型和动态绑定可以实现动态改变消息的执行者和消息的接收者，另外与动态绑定相关的还有基于消息传递机制的消息转发机制，主要处理应对一些接收者无法处理的消息，此时有机会将消息转发给其他对象处理。

动态绑定是基于动态类型的，在运行时对象的类型确定后，对象的属性和方法也就确定了，包括类中原来的属性和方法，以及运行时动态新加入的属性和方法。可以通过对象的 isa 指针找到方法列表，即可执行的消息列表。

3．动态加载

动态加载主要包括两个方面，一个是动态资源加载，另一个是代码模块的加载。这些资源在运行时根据需要有选择性地加入到程序中，是一种代码和资源的“懒加载”模式，可以降低内存开销，提高整个程序的性能，另外也大大提高了可扩展性。

例如：资源动态加载中的图片资源的屏幕适配，同一个图片对象可能需要准备几种不同分辨率的图片资源，程序会根据当前的机型动态选择加载对应分辨率的图片，像 iPhone 4 之前老机型使用的是@1x 的原始图片，而 Retina 显示屏出现之后每个像素点被分成了 4 个像素，因此同样尺寸的屏幕需要 4 倍分辨率（宽高各两倍）的@2x 图片，最新的针对 iPhone 6/6+以上的机型则需要@3x 分辨率的图片。例如，图 3-3 所示应用的 AppIcon，需要根据机型以及机型分辨率动态地选择加载某张具体的图片资源。

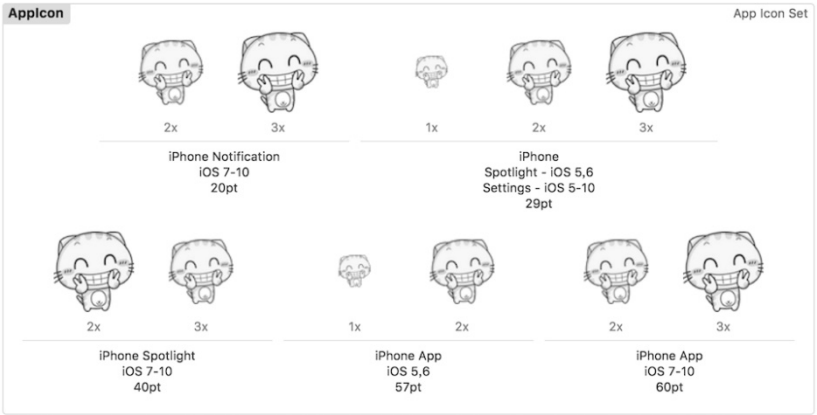

图 3-3　AppIcon

4．消息传递机制

在 Objective-C 中，方法的调用不再理解为对象调用其方法，而是要理解成对象接收消息，消息的发送采用“动态绑定”机制，具体会调用哪个方法直到运行时才能确定，确定后才会去执行绑定的代码。方法的调用实际就是告诉对象要干什么，给对象传递一个消息，对象为接收者（receiver），调用的方法及其参数就是消息（message），给一个对象传递消息的表达为：

```
/* 打印所有的变量名及其类型 */
for (int i = 0; i < count; i++) {
    const char *memberName = ivar_getName(members[i]);
    const char *memberType = ivar_getTypeEncoding(members[i]);
    NSLog(@"name:%s type:%s",memberName,memberType);
}
/* 访问私有属性和变量 */
Ivar varname = members[0];
Ivar varmajor = members[1];
object_setIvar(test, varname, @"albert");
object_setIvar(test, varmajor, @"mathmetics");
NSString *name = object_getIvar(test, varname);
NSString *major = object_getIvar(test, varmajor);
NSLog(@"pname:%@",name);
NSLog(@"pmajor:%@",major);
/* 要手动释放 */
free(members);
```

程序的运行结果为：

```
2017-04-0212:07:22.105758 CommandLine[189:3242353] name:major type:@"NSString"
2017-04-02 12:07:22.105930 CommandLine[189:3242353] name:_name type:@"NSString"
2017-04-02 12:07:22.106156 CommandLine[189:3242353] pname:albert
2017-04-02 12:07:22.106221 CommandLine[189:3242353] pmajor:mathmetics
```

Runtime 暴力访问对象私有方法

和访问私有变量类似，这里使用 class_copyMethodList 运行时函数获取类对象中的方法列表，然后使用 performSelector 函数执行某个方法。这里只展示了实例方法的访问，类方法的访问也类似，另外还可以使用 class_addMethod 运行时函数往类对象中强行添加新的方法，具体见 Method Swizzling 部分。

```
#import "Test.h"
#import <objc/runtime.h>
...
Test *test = [[Test alloc] init];
/* 运行时暴力访问私有方法 */
/* 获取类对象方法列表 */
unsigned int count = 0;
Method *methods = class_copyMethodList([Test class], &count);
/* 获取第一个方法的方法名 */
SEL sel = method_getName(methods[0]);
/* 执行该方法 */
[test performSelector:sel];
```

3.4 Objective-C 中的类别与扩展机制

类别（Category）与类扩展（Extension）都是 Objective-C 中独有的用于灵活地扩展类对

3.4.4 为什么类别只能添加扩展方法而不能添加属性变量

这个问法有点疑惑性，实际问的是为什么向类别添加属性会失败。苹果公司并不是故意将类别设计成不允许开发者在里面添加属性，如果是有意，那么可能从设计上考虑保持类别特性的单纯，专门用来扩展功能，和继承的角色区别开，防止类别污染被扩展的类。

在类别中扩展属性不能成功的原因是无法在类别中取得属性的加下画线的实例变量名，导致无法手动实现实例变量的存取方法。在类别中定义了属性后，属性其实也成功添加到了类的属性列表中，但编译器只为其声明了存取方法，没有实现，同时又没有合成加下画线的实例变量名，导致无法访问实例变量也无法自己手动实现其存取方法，对于一个不能访问的属性，则失去了其存在的意义。

如果使用运行时的机制，那么开发者其实可以强行实现类别中属性的存取方法，实现在类别中扩展属性。这里在运行时，实现为 NSString 类扩展一个叫作 newString 的属性。示例代码如下：

```
/* NSString+Category.h */
#import <Foundation/Foundation.h>

@interface NSString (Category)
/*  在类别中扩展属性  */
@property (nonatomic, copy) NSString *newString;
@end
/* NSString+Category.m */
#import "NSString+Category.h"
#import <objc/runtime.h>

@implementation NSString (Category)
*  运行时强行实现 newString 的 getter 和 setter */
- (NSString *)newString {
    return objc_getAssociatedObject(self, @"newString");
}
- (void)setNewString:(NSString *)newString {
    objc_setAssociatedObject(self, @"newString", newString, OBJC_ASSOCIATION_COPY);
}
@end
/* main.m */
#import <Foundation/Foundation.h>
#import "NSString+Category.h"

int main(int argc, const char * argv[]) {
    NSString *string;
    /*  调用 newString 的 setter 方法  */
    string.newString = @"newString";
    /*  调用 newString 的 getter 方法  */
    NSLog(@"%@",string.newString);
    return 0;
}
```

3.5 Method Swizzling 魔法

Objective-C 的运行时特性提供了一种称作 Method Swizzling 的方法利器，利用它可以更加随心所欲地在运行时期间对编译器已经实现的方法再次动手脚，主要包括：交换类中某两个方法的实现、重新添加或替换某个方法的具体实现。

运行时的几种特殊类型：

1）Class。类名通过类的 class 类方法获得，例如：[UIViewController class]。

2）SEL。选择器也就是方法名，通过@selector(方法名:)获得，例如：@selector (buttonClicked:)。

3）Method。方法即运行时类中定义的方法，包括方法名（SEL）和方法实现（IMP）两部分，通过运行时方法 classgetInstanceMethod 或 classgetClassMethod 获得。

4）IMP。方法实现类型指方法的实现部分，通过运行时方法 classgetMethod Implementation 或 methodgetImplementation 获得。

3.5.1 Method Swizzling 的应用场景有哪些

Method Swizzling 主要用于在运行时对编译器编译好的方法再次进行编辑，应用场景主要有以下 4 种。

1．替换类中某两个类方法或实例方法的实现

替换函数：method_exchangeImplementations(method1, method2)。

这里随便定义一个 Test 类，类中定义两个实例方法和类方法并在.m 文件中实现，在运行时将两个实例方法的实现对调，以及将两个类方法的实现对调。注意运行时代码写在类的 load 方法内，该方法只会在该类第一次加载时调用一次，且编写运行时代码的地方需要引入运行时头文件：

```
#import <objc/runtime.h>
```

Test 类定义：

```
/* Test.h */
#import <Foundation/Foundation.h>
@interface Test : NSObject
/*定义两个公有实例方法*/
- (void)instanceMethod1;
- (void)instanceMethod2;
/*  定义两个公有类方法*/
+ (void)classMethod1;
+ (void)classMethod2;
@end
/* Test.m */
#import "Test.h"
#import <objc/runtime.h>
@implementation Test
/*实例方法的原实现*/
```

```
- (id)copyWithZone:(nullable NSZone *)zone {
    // 深拷贝或浅拷贝实现
}
```

NSMutableCopying 的协议方法为 mutableCopyWithZone：

```
- (id)mutableCopyWithZone:(nullable NSZone *)zone {
    // 深拷贝或浅拷贝实现
}
```

3.6.4 Objective-C 等同性中的字符串相等如何判断

有如下代码：

```
NSString *firstUserName = @"nick";
NSString *secondUserName = @"nick";
if (firstUserName   == secondUserName) {
    NSLog(@"areEqual");
}
else {
    NSLog(@"areNotEqual");
}
```

上述程序会输出“areEqual”。

看上去好像答案很明显，其实不然。用“==”比较两个指针相当于检查它们是否指向同一个对象（指针存储的是对象的地址）。两个指针只有指向同一个对象（两个指针存储相同的对象地址）时这两个指针才相等。两个指针指向的两个对象的值相等，但不是同一个对象，那么这两个指针还是不相等，所以这么分析上面的代码应该输出“areNotEqual”。

在上面的代码片中，firstUserName 和 secondUserName 是指向两个不同对象的指针，但为什么会输出“areEqual”呢？是因为 iOS 编译器对 string 对象的引用做了优化，也就是对字符串相同的对象做了复用而不是重新分配空间，所以上面代码虽然是分别定义的两个指针，但实际还是指向了同一个字符串对象，结果输出“areEqual”。

常见面试笔试真题 1：isEqual 和 isEqualToString 有什么区别？

答案：isEqual 是比较两个 NSObject 的方法，而 isEqualToString 是比较两个 NSString 的方法，明显 isEqualToString 只是专门用来比较字符串的，是 isEqual 的衍生方法。

isEqual 先是比较两个指针地址，如果地址相同，那么直接返回 YES；如果地址不同，那么再看两个指针指向的对象是否为空以及对象类型是否相同。如果有一个为空或者两者不是同类对象，那么直接返回 NO；如果都不为空且属于同类对象而且对象的属性也相等，那么返回 YES。

常见面试笔试真题 2：下面的代码会打印什么结果？

```
NSString *str = @"a123";
NSLog(@"%@", (str == @"a123") ? @"yes" : @"no");
```

答案：打印结果为 no。这里字符串 str 和相同内容的常量字符串比较，“==”比较的是它

们的指针，这样的写法临时的@“a123”是一个新的字符串，虽然字符串内容相同，但是没有被编译器优化，因此和 str 不是同一个字符串，地址不同。

常见面试笔试真题 3：下面可以比较两个 NSString *str1, *str2 异同的方法是（　　）。

A．if(str1 = str2) xxx;

B．if([str1 isEqualToString:str2]) xxx;

C．if(str1 && str2) xxx;

D．if([str1 length] == [str2 length]) xxx;

答案：B。

3.6.5　一个 Objective-C 对象如何进行内存布局（考虑有父类的情况）

所有父类的成员变量和自己的成员变量都会存放在该对象所对应的存储空间中。每一个对象内部都有一个 isa 指针，指向它的类对象，类对象中存放着本对象的实例方法列表（对象能够接收的消息列表，保存在它所对应的类对象中）和成员变量的列表，类对象内部也有一个 isa 指针指向元对象，元对象内部存放的是类方法列表，类对象内部还有一个 superclass 的指针，指向它的父类对象。

每个 Objective-C 对象都有相同的结构，如图 3-12 所示。

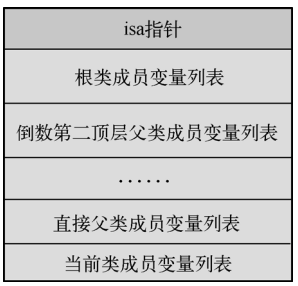

图 3-12　Objective-C 对象内存结构

常见面试笔试真题：一个 Objective-C 对象的 isa 指针指向什么？有什么作用？

答案：Objective-C 实例对象的 isa 指针是指向它的类对象的。Objective-C 中有 3 个层次的对象：实例对象（Instance）、类对象（Class）和元类（MetaClass）。Class 即自定义的类，是实例对象的类对象，而类对象又是其对应元类的实例对象。它们的关系如图 3-13 所示。

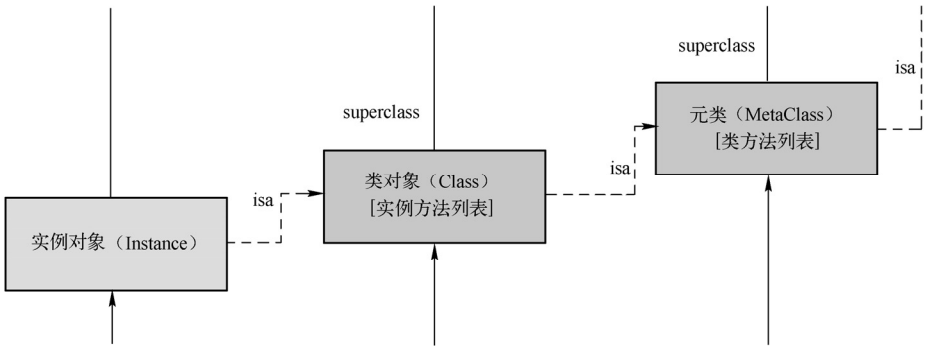

图 3-13　isa 指针结构

isa 指针的作用是通过它可以找到对应类对象或元类中的方法（对象可接收的方法列表）。例如，实例对象可以在其类对象中找到它的实例方法，Class 对象可以从元类中找到它的类方法。

第 4 章　Cocoa Touch 框架相关

第 1 章中讲到 Cocoa Touch 是官方提供给开发者的用来开发 iOS 应用的基础框架，主要包括 Foundation 框架和 UIKit 框架。这里整理了部分有关这个框架的一些常见问题，尤其是 UITableView 这个核心的组件。此部分的内容也是比较基础的知识。

4.1 UITableView

UITableView 可以说是 UIKit 中最重要的一个组件，不仅可以用来展示数据列表，还可以用做页面的布局和开发专用组件等。UITableView 的使用遵循 MVC 模式，从而实现了数据模型（NSObject）、视图（UIView）和控制器（UITableViewController）分离。UITableView 继承自 UIScrollView，可上下滑动，可以作为根视图组件，也可以作为子视图组件。

4.1.1 UITableViewCell 的复用原理是怎么样的

在 UITableViewController 中创建 UITableViewCell 时，initWithSytle:resuseIdentifier 中的 reuseIdentifier 有什么用？UITableViewCell 的复用原理是怎么样的？

reuseIdentifier 顾名思义是一个复用标识符，是一个自定义的独一无二的字符串，用来唯一地标记某种重复样式的 UITableViewCell。系统是通过 reuseIdentifier 来复用已经创建了的指定样式的 cell，iOS 中表格的 cell 通过复用机制来提高加载效率，因为多数情况下表格中的 cell 样式都是重复的，只是数据模型不同而已。因此，系统可以在保证创建一定数量的 cell 的前提下（覆盖整个 tableView），通过保存并重复使用已经创建的 cell 来提高加载效率和优化内存，避免不停地创建和销毁 cell 元素。

UITableViewCell 的复用原理其实很简单，可以通过下面一个简单的例子来理解。

开发时在 UITableViewController 类中写 cell 复用代码的基本模板如下：

```
        /* 可复用 cell 制作 */
        - (UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath:(NSIndexPath
*)indexPath {
            /* 定义 cell 重用的静态标志符 */
            static NSString *cell_id = @"cell_id_demo";
            /* 优先使用可复用的 cell */
            UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:cell_id];
            /* 如果要复用的 cell 还没有创建，那么创建一个供之后复用 */
            if (cell == nil) {
                /* 新创建 cell 并使用 cell_id 复用符标记 */
                cell = [[UITableViewCell alloc] initWithStyle:UITableViewCellStyleDefault reuseIdentifier:
cell_id];
            }
            /* 配置 cell 数据 */
            cell.textLabel.text = [NSString stringWithFormat:@"Cell%i", countNumber];
```

```
        /* 其他 cell 设置... */
        return cell;
    }
```

代码这样写的原因是通过调用当前 tableView 的 dequeueReusableCellWithIdentifier 方法判断指定的 reuseIdentifier 是否有可以重复使用的 cell。如果有，那么会返回可复用的 cell，cell 就绪之后便可以开始更新 cell 的数据；如果没有可以复用的 cell，那么返回 nil，然后会进入后面的 if 语句，此时创建新的 cell 并给它标记一个标识符 reuseIdentifier。注意上面的 if 语句并不是每次都创建一次新的 cell，然后全部重复利用新创建的那一个 cell，这是对 cell 复用机制的误解。事实是要创建足够数量的可覆盖整个 tableView 的 cell 之后才会开始复用之前的 cell（UITableView 中有一个 visiableCells 数组保存当前屏幕可见的 cell，还有一个 reusableTableCells 数组用来保存那些可复用的 cell）。下面通过测试来验证。

如何简洁清楚地展示 UITableViewCell 的复用机制呢？下面的例子是创建最基本的文本 cell，并创建一个 cell 创建计数器，每次新创建 cell 计数器加 1 然后显示在 cell 上，如果是复用的 cell，那么会显示是复用的哪一个 cell，测试代码如下：

```
    /* 分区个数设置为 1 */
    - (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView {
        return 1;
    }
    / * 创建 20 个 cell，保证覆盖并超出整个 tableView */
    - (NSInteger)tableView:(UITableView *)tableView numberOfRowsInSection:(NSInteger)section {
        return 20;
    }
    /* cell 复用机制测试 */
    - (UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath:(NSIndexPath
*)indexPath {
        /* 定义 cell 重用的静态标志符 */
        static NSString *cell_id = @"cell_id_demo";
        /* 计数用 */
        static int countNumber = 1;
        /* 优先使用可复用的 cell */
        UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:cell_id];
        /* 如果要复用的 cell 还没有创建，那么创建一个供之后复用 */
        if (cell == nil) {
            /* 新创建 cell 并使用 cell_id 复用符标记 */
            cell = [[UITableViewCell alloc] initWithStyle:UITableViewCellStyleDefault reuseIdentifier:
cell_id];
            /* 计数器标记新创建的 cell */
            cell.textLabel.text = [NSString stringWithFormat:@"Cell%i", countNumber];
            /* 计数器递增 */
            countNumber++;
        }
        return cell;
    }
```

运行在 iPhone 5S 设备上（UITableViewController 作为跟控制器，tableView 覆盖整个屏幕），

20 个 cell 显示结果依次为（见图 4-1 和图 4-2）：

Cell1、Cell2、Cell3、Cell4、Cell5、Cell6、Cell7、Cell8、Cell9、Cell10、Cell11、Cell12、Cell13、Cell14、Cell1、Cell2、Cell3、Cell4、Cell5、Cell6。

图 4-1　cell 刚好填满屏幕

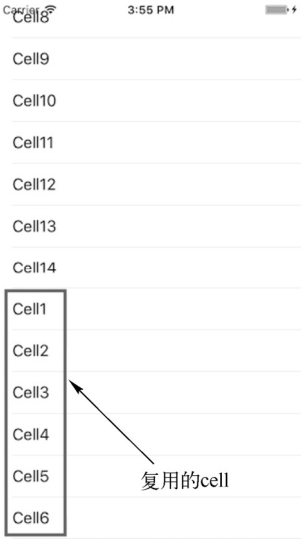

图 4-2　滑动复用的 cell

可以看出一共创建了 14 个 cell，其中整个屏幕可显示 13 个 cell，系统多创建一个的原因是保证在表格滑动显示半个 cell 时仍然能覆盖整个 tableView。之后的 6 个 cell 就是复用了开始创建的那 6 个 cell 了。这样 UITableViewCell 复用的基本机制就很清楚了，还会有 reloadData 或者 reloadRowsAtIndex 等刷新表格数据的情况，可能会伴随新的 cell 创建和可复用 cell 的更新，但也是建立在基本复用机制的基础之上的。

常见面试笔试真题：UITableView 中 cell 的复用是由几个数组实现的？（　　）

A．1　　B．2　　C．3　　D．3 或 4

答案：B。

UITableView 中 cell 的复用是由两个数组实现的，一个是 visiableCells 数组，保存当前屏幕可见的 cell；另一个 reusableTableCells 数组用来保存那些可复用的 cell。所以答案为 B。

4.1.2　能否在一个视图控制器中嵌入两个 tableView 控制器

可以，相当于视图和视图控制器的嵌套，视图可以添加子视图，视图控制器也可以添加子控制器。这种情况有时会被用到而且很重要，但是有一点容易被忽视，就是将子视图添加到了父视图却忘记将对应的控制器作为子控制器添加到父控制器，导致子视图能显示但是不能响应（没有对接好控制器）。例如在当前视图上放一个小尺寸的表格组件，也就是在 UIViewController 上添加一个 UITableViewController 子控制器及其子 view。

```
/* 假设有 3 个视图控制器，一个作为父控制器，两个作为子控制器 */
UIViewController *superVC = [[UIViewController alloc]init];
```

第 5 章　iOS 开发中的对象间通信机制

在 iOS 开发过程中，尤其是在一些相对复杂的工程中，对象与对象之间的通信是必不可少的，选择合适有效的机制来实现对象之间通信是十分重要的。在进行对象之间的通信时，一方面要保证通信的安全性、有效性；另一方面要降低类之间的耦合性。对象间通信机制在高质量的程序开发中起着关键的作用。

iOS 中对象间通信机制主要有基于 Protocol（协议）的代理模式、基于 KVO（链值观察）和通知中心的观察者模式、消息推送、对象间直接引用传值和 block 代码块通信等，在实际的应用中，通常在不同的情形下会选择不同的实现机制。

5.1 iOS 中的 Protocol 和 Delegate

Protocol（协议）类似于 Java 语言中的接口，它是一个自定义方法的集合，让遵守这个协议的类去实现为了达到某种功能的这些方法，与 Java 语言中的接口不同的是协议中可以使用 @optional 来选择性实现某个方法。

Delegate（代理）是一种设计模式，是一个概念，只不过在 Objective-C 中通过 Protocol 来进行实现，指的是一个对象在某些特定时刻通知其他类的对象去做一些任务，但不需要获取到那些对象的指针，两者共同来完成一件事，实现不同对象之间的通信。代理模式大大减小了对象之间的耦合度，它使得代码逻辑更加清晰有序，而且，由于降低了框架复杂度，所以便于代码的维护扩展。另外，消息的传递过程可以有参数回调，它类似于 Java 语言的回调监听机制，从而大大提高了编程的灵活性。

通过协议实现代理模式的示例如下，在下例中，DemoViewController 定义代理源，触发代理事件，并通知遵循该代理的 MyViewController 类去做一些事情。协议声明如下：

```
#import <UIKit/UIKit.h>
/* 定义协议（协议可以定义在外部单独的头文件中） */
@protocol DemoDelegate <NSObject>
/* 必须实现的方法，默认是@required */
@required
- (void) selectedCell:(NSInteger)index;
/* 非必须实现的方法 */
@optional
- (void) optionalFunc;
@end

@interface DemoTableViewController : UITableViewController
/* 定义代理，委托其他类来帮助本类完成一些其他任务，本类通过下面定义的 delegate 来通知其他实现上面协议的其他类 */
@property (nonatomic, weak) id<DemoDelegate>delegate;
@end
```

第 6 章　iOS 中的图层与动画

CoreAnimation，中文翻译为核心动画，是 iOS 与 OS X 平台上负责图像渲染与动画的基础框架，也是一组非常强大的动画处理 API。使用它能做出非常炫丽的动画效果，而且往往是事半功倍，也就是说，使用少量的代码就可以实现非常强大的功能。CoreAnimation 拥有一组直接作用在 CALayer 上的动画处理 API，CoreAnimation 的动画执行过程都是在后台操作的，不会阻塞主线程。通过 CoreAnimation 提供的接口，可以方便完成自己所想要的动画。

6.1 图层

6.1.1 UIView 和 CALayer 的区别与联系是什么

1．UIView 和 CALayer 是什么

CALayer 是动画中经常使用的一个类，它包含在 QuartzCore 框架中。CALayer 类在概念上和 UIView 类似，同样是一些被层级关系树管理的矩形块，也可以包含一些内容（像图片、文本或者背景色），管理子图层的位置。它们有一些方法和属性用来做动画和变换。使用 CoreAnimation 开发动画的本质就是将 CALayer 中的内容转化为位图供硬件操作。

CALayer 是一个比 UIView 更底层的图形类，是对底层图形 API（OpenGL ES）一层层封装后得到的一个类，用于展示一些可见的图形元素，保留了一些基本的图形化操作，但同时由于相对高度的封装，使得操作使用变得很简单。CALayer 用于管理图形元素，甚至可以制作动画，它保留了一些几何属性，如位置、尺寸、图形变换等。一般的 CALayer 是作为 UIView 背后的支持角色，在创建了一个 UIView 的同时也存在一个相应的 CALayer。UIView 作为 CALayer 的代理角色去实现一些功能，例如常见的为 UIView 制作一个圆角，就会用到 UIView 背后的 layer 操作：

```
view.layer.cornerRadius = 10;
```

CALayer 可以通过 UIView 很方便地展示操作 UI 元素，但是 CALayer 自身单独也可以展示和操作可见元素，且灵活度更高，它自身有一些可见可设置的属性，如背景色、边框、阴影等。

另外，UIView 简单来说是一个可以在里面渲染可见内容的矩形框，它里面的内容可以和用户进行交互，UIView 可以对交互事件进行处理。除了其背后 CALayer 的图形操作支持，UIView 自身也有像设置背景色等最基本的属性设置。

2．UIView 和 CALayer 的联系

UIView 和 CALayer 的主要联系上面已经提到，CALayer 在 UIView 背后提供更加丰富灵活的图形操作，UIView 作为 CALayer 的代理更加快速地帮 CALayer 显示一些常用的 UI 元素并提供交互。

另外，UIView 类是所有视图的基类，CALayer 是图层类。事实上，UIView 和 CAlayer 是平行的层级关系。每一个 UIView 都有一个 CALayer 实例的图层属性，视图的责任就是创建并管理图层，以确保当子视图在层级关系中被添加或者被移除的时候，与它们相关联的图层也同样在层级关系树中有相同的操作。

3．UIView 和 CALayer 的区别

1）CALayer 无法响应用户事件。UIView 和 CALayer 的最明显区别在于它们的可交互性，即 UIView 可以响应用户事件，而 CALayer 不可以，原因可以从这两个类的继承关系上看出（见图 6-1）。UIView 是继承自 UIResponder 的，决定了 UIView 类及其子类能够通过响应链（iOS 通过视图层级关系来传递触摸事件）接收并响应用户事件。而 CALayer 直接继承于 NSObject 类，所以它不清楚具体的响应链，也就无法响应用户事件。

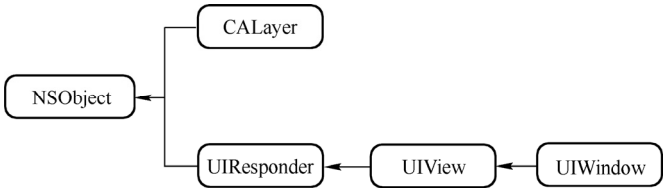

图 6-1　CALayer 和 UIView 继承关系

2）分工不同。UIView 类侧重于对显示内容的管理和整体布局，而 CALayer 侧重于显示内容的绘制、显示和动画。

3）所属框架不同。UIView 类是属于 UIKit.framework 框架的，UIKit 框架主要就是用来构建用户界面的。CALayer 类是属于 QuartzCore.framework 框架的，而且 CALayer 是作为一个低级的，可以承载绘制内容的底层对象出现在该框架的。

常见面试笔试真题 1：为什么 iOS 中提供 UIView 和 CALayer 这两个平行的层级结构呢？

答案：iOS 中提供 UIView 和 CALayer 这两个平行层级结构主要是为了做到职责分离，实现视图的绘制、显示和布局解耦，避免很多重复的代码。在 iOS 和 Mac OS 两个平台上，事件和用户交互有很多地方并不相同，毕竟基于多点触控的用户界面和基于鼠标键盘有着本质的区别，这就是为什么 iOS 有 UIKit 和 UIView，但是 Mac OS 有 Appkit 和 NSView 的原因。它们功能虽然相似，但是在实现上有着显著的区别。创建两个层次结构就能够在 iOS 与 Mac OS 之间共享代码，从而使得开发更加便捷。

常见面试笔试真题 2：UIWindow 是什么？有什么特点和作用？

答案：从图 6-1 中的继承关系会发现 UIWindow 居然是 UIView 的子类，因为 UIWindow 在应用中是作为根视图来承载 UIView 元素的，也就是说根父视图是子视图的子类，有点违背直觉。

但事实就是这样，UIWindow 提供一个区域（一般就是整个屏幕）来显示 UIView，并且将事件分发给 UIView。一个应用一般只有一个 UIWindow，但特殊情况也会创建子 UIWindow，例如实现一个始终漂浮在顶层的悬浮窗，就可以使用一个 UIWindow 来实现。

6.1.2　什么是 Layer 层对象

Layer 层对象是用来展示可见内容的一种数据对象，常在视图中用来渲染视图内容。一般

的层对象在界面中可以实现一些复杂的动画或者其他类型的一些复杂特效。

常见的几个其自身具有绘制功能的专用 Layer 有：CATextLayer、CAShapeLayer、CAGradientLayer，这里给出使用示例。其他还有用于 3D 图形变换的 CATransformLayer，实现滚动视图的 CAScrollLayer，专门播放视频的 AVPlayerLayer 和制作粒子特效的 CAEmitterLayer 等。它们都是继承自 CALayer 的，和 CALayer 一样都来自 QuartzCore.framework 框架。

1．CATextLayer

这个类是用来实现更加灵活的文字布局和渲染的，它几乎包含了 UILabel 的所有特性并在此基础上增加了很多更强大的功能，包括字体、尺寸、前景色和下画线等文字效果，同时 CATextLayer 的渲染效率明显高于 UILabel。

通过 CALayer 来实现一个 UILabel 的示例代码如下（效果见图 6-2）：

```
- (void)viewDidLoad {
    [super viewDidLoad];
    /* 创建一个字符承载视图 */
    UIView *textView = [[UIView alloc] initWithFrame:CGRectMake(50, 50, 200, 50)];
    CATextLayer *text = [CATextLayer layer];
    text.frame = textView.frame;
    text.string = @"CAText";
    /* 文字前景色和背景色 */
    text.foregroundColor = [UIColor whiteColor].CGColor;
    text.backgroundColor = [UIColor blackColor].CGColor;
    /* 文字超出视图边界裁剪 */
    text.wrapped = YES;
    /* 文字字体 */
    text.font = (__bridge CFTypeRef)[UIFont systemFontOfSize:30].fontName;
    /* 文字居中 */
    text.alignmentMode = kCAAlignmentCenter;
    /* 适应屏幕 Retina 分辨率，防止像素化导致模糊 */
    text.contentsScale = [[UIScreen mainScreen] scale];
    [textView.layer addSublayer:text];
    [self.view addSubview:textView];
}
```

图 6-2　CATextLayer 效果

2．CAShapeLayer

这个类是用来专门绘制矢量图形的图形子类，例如可以指定线宽和颜色等利用 CGPath 绘制图形路径，可以实现图形的 3D 变换效果，渲染效率比 Core Graphics 快很多，而且可以在超出视图边界之外绘制，即不会被边界裁减掉。

这里展示使用 CAShapeLayer 绘制一个圆形的实例代码如下（效果见图 6-3）：

```
-(void)drawRect:(CGRect)rect{
        UIBezierPath *path = [UIBezierPath bezierPathWithArcCenter:CGPointMake(100,100)
radius:100 startAngle:0 endAngle:M_PI_2 clockwise:YES];
        path.lineWidth = 5;
        path.lineCapStyle = kCGLineCapRound;
        path.lineJoinStyle = kCGLineCapRound;
        [path stroke];
}
```

6.1.4 iOS 中如何实现为 UIImageView 添加圆角

圆角指按照一定圆角半径平滑矩形视图的 4 个角的效果，如图 6-6 所示。

图 6-6 圆角效果

按照背后的渲染方式，实现 UIView 及其子类的圆角效果有两种方法：一种是直接设置 layer 的圆角属性，为“离屏渲染”；另一种是自定义圆角绘制方法，实现“当前屏幕渲染”。

当前屏幕渲染（On-screen Rendering）指 GPU 直接在当前显示的屏幕缓冲区中进行图形渲染，不需要提前另开缓冲区也就不需要缓冲区的切换，因此性能较高。

离屏渲染（Off-screen Rendering）：简单来说就是提前另开一个缓冲区进行图形渲染，由于显示需要和当前屏幕缓冲区进行切换，所以很耗费性能。通常圆角、遮罩、不透明度、阴影、渐变、光栅化和抗锯齿等设置都会触发离屏渲染。

（1）“离屏渲染”实现圆角

iOS 中圆角效果实现的最简单、最直接的方式是直接修改 View 的 layer 层参数，这样会触发“离屏渲染”，性能不高。

```
/* 设置圆角半径 */
view.layer.cornerRadius = 5;
/* 将边界以外的区域遮盖住 */
view.layer.masksToBounds = YES;
```

（2）“当前屏幕渲染”实现圆角

直接在当前屏幕渲染绘制，提高性能。

为 UIImage 类扩展一个实例方法：

```
/* On-screen-renderring 绘制 UIImage 矩形圆角 */
- (UIImage *)imageWithCornerRadius:(CGFloat)radius ofSize:(CGSize)size{
    /* 当前 UIImage 的可见绘制区域 */
    CGRect rect = (CGRect){0.f,0.f,size};
    /* 创建基于位图的上下文 */
```

```
        UIGraphicsBeginImageContextWithOptions(size,NO,UIScreen.mainScreen.scale);
        /* 在当前位图上下文添加圆角绘制路径 */
        CGContextAddPath(UIGraphicsGetCurrentContext(),[UIBezierPath
bezierPathWithRoundedRect:rect cornerRadius:radius].CGPath);
        /* 当前绘制路径和原绘制路径相交得到最终裁剪绘制路径 */
        CGContextClip(UIGraphicsGetCurrentContext());
        /* 绘制 */
        [self drawInRect:rect];
        /* 取得裁剪后的 image */
        UIImage *image = UIGraphicsGetImageFromCurrentImageContext();
        /* 关闭当前位图上下文 */
        UIGraphicsEndImageContext();
        return image;
    }
```

使用时，让实例化的 UIImage 对象调用上面的实例方法即可。

```
    UIImageView *imageView = [[UIImageView alloc] initWithFrame:CGRectMake(10, 10, 100, 100)];
    /* 创建并初始化 UIImage */
    UIImage *image = [UIImage imageNamed:@"icon"];
    /* 添加圆角矩形 */
    image = [image imageWithCornerRadius:50 ofSize:imageView.frame.size];
    [imageView setImage:image];
```

引申：考虑到圆角渲染的性能问题，实际开发中会使用一个小技巧来实现圆角效果，即直接制作一个颜色和背景色相同的、中间镂空的图片遮在头像上面，简单实现遮罩效果，从而避免了圆角渲染造成的性能问题。

6.1.5 contentsScale 属性有什么作用

图层的 contentsScale 属性属于支持高分辨率屏幕（如 Retina 屏幕）机制的一部分，它定义了图层 content 中图像的像素尺寸与视图大小的比例。它也被用来判断在绘制图层时允许为 content 属性创建的空间大小，以及需要显示的图片的拉伸度。

默认情况下，contentsScale 的值是 1.0，也就是说图层的绘制系统将会以每个点对应一个像素来绘制图片。如果将其设置为 2.0，那么会以每个点对应两个像素来绘制图片，此即所谓的 Retina 屏幕。

在开发中，有时会直接为图层的 content 设置图片，这时可以设置 contentsScale 为合适的值，以防止图片在 Retina 屏幕上显示不正确（像素化或模糊），代码如下：

```
    layer.contentsScale = [UIScreen mainScreen].scale;
```

常见面试笔试真题：iOS 中点与像素有什么关系？

答案：1）点是 iOS 中标准的坐标体系。它就是 iOS 中的虚拟像素，也被称为逻辑像素。在标准设备中，一个点就是一个像素，但是在 Ratina 屏幕上，一个点等于 2×2 个像素。iOS 用点作为屏幕的坐标测算体系就是为了在 Retina 设备和普通设备上能有一致的视觉效果。

2）像素是图片分辨率的尺寸单位。物理像素坐标并不会用于屏幕布局，但是仍然和图片有相对关系。UIImage 是一个屏幕分辨率解决方法，它是用点来度量大小的。但是，一些底

6.1.10 QuartzCore 和 Core Graphics 有什么区别

Core Graphics 是 iOS 系统中的底层绘图框架。平时使用最频繁的 point、size、rect 等视图属性都被定义在这个框架中。Core Graphics 框架所包含的 API 都是以 CG 开头，它提供的都是 C 语言的函数接口。

QuartzCore 框架从名称中感觉不是很清晰，但是从其头文件可以发现，它其实就是 CoreAnimation。也就是说，QuartzCore 专指 CoreAnimation 用到的动画相关的库、API 和类。以下是 QuartzCore 框架的头文件：

```
#ifndef QUARTZCORE_H
#define QUARTZCORE_H
#include <QuartzCore/CoreAnimation.h>
#endif /* QUARTZCORE_H */
```

Core Graphics 和 QuartzCore 都是跨 iOS 和 Mac OS 平台的，这点区别于 UIKit 框架（UIKit 框架只适用 iOS 平台）。其次 Quartz Core 中大量使用了 Core Graphics 中的类，因为动画的产生必然要用到图形库中的东西。

6.2 动画

6.2.1 UIView 动画原理是什么？以 UIView 类的 animateWithDuration 方法为例

iOS 4.0 以后提供了几个基于 block 块的动画方法（UIView 类的+animateWithDuration 类方法），替代之前的+beginAnimations:context:和+commitAnimations 方法，用于快速实现一些简单常用的 UI 动画效果。这种方法创建的动画为隐式动画，开发者只需要指定 UI 元素的一些属性的目标值，即可得到属性从当前值平滑过渡到目标值的动画效果，同时可以指定动画的持续时间。

可以设置 UI 隐式动画的常用属性主要有 frame、bounds、center、transform、alpha、backgroundColor、contentStretch 等。

例如，实现一个让视图平滑移动到指定位置的动画代码如下：

```
[UIView animateWithDuration:0.3 animations:^{
    /* 设置 center 目标属性值 */
    view.center = CGPointMake(200, 200);
} completion:^(BOOL finished) {
    /* 动画执行结束回调 */
}];
```

动画持续时间为 0.3s。在 animations 代码块中定义目标属性值。动画结束后会回调 completion 代码块，可在动画结束后继续开始新的动画，实现简单的连续动画效果。

6.2.2 什么是隐式动画和显式动画

隐式动画是 UIKit 动画的基础，是 iOS 中创建动态 UI 界面的最直接的一种方式。开发者

通过直接设定 UI 元素的一些可见属性的目标值，如 frame、bounds、center、transform、alpha、backgroundColor、contentStretch 等，即可自动生成属性变化的过渡动画。例如，设置视图的目标位置为 P1，即可自动生成从视图当前位置移动到 P1 的平滑动画。隐式动画是一种默认动画，动画是线性的，可以满足基本的需求，但对于一些复杂的动画，如让视图沿曲线移动，隐式动画就无能为力了，需要定义显式动画来实现。

例如，假设视图 view 位于屏幕外，通过执行下面的隐式动画，view 可以平滑地移入当前视图中央，动画时间为 0.5s，动画结束时则回调可以紧接着进行其他的操作，如继续进行下一个动画等。

```
/* 隐式动画，视图平滑移入当前视图中央 */
[UIView animateWithDuration:0.5 animations:^{
    view.center = self.view.center;
} completion:^(BOOL finished) {
    // 动画结束回调...
}];
```

显式动画不像隐式动画那样默认从一个初始状态线性变化到目标状态，而是需要显式地定义完整的动画流程，这样略微复杂的同时会更加灵活，可以实现更加复杂的动画效果。例如，隐式动画只能实现直线平移效果，而显式动画可以显式地定义任意的曲线路径，让视图沿着曲线移动。简单来说，显式动画就是要显式地定义动画对象，设置动画对象的各个状态和值，然后将动画对象应用到视图上，即可呈现动画的效果。

例如，下面定义一个在 x 和 y 轴方向上（二维平面）不断缩放的动画对象，动画使视图层先放大 1.2 倍，动画结束后回到初始状态，如此循环。应用的时候将该动画对象添加到对应视图的 layer 层上即可。

```
/* 定义基本动画对象(缩放动画) */
CABasicAnimation *animation = [CABasicAnimation animationWithKeyPath:@"transformPath"];
/* 设置动画目标状态, xy 平面放大 1.2 倍*/
CATransform3D scaleTransform = CATransform3DMakeScale(1.2, 1.2, 1);
animation.toValue = [NSValue valueWithCATransform3D:scaleTransform];
/* 动画持续时间 0.5s */
animation.duration = 0.5;
/* 动画不断循环重复 */
animation.repeatCount = HUGE_VALF;
/* 自动逆动画 */
animation.autoreverses = YES;
/* 动画结束移除之前动画对视图的影响，回到初始状态 */
animation.removedOnCompletion = NO;
animation.fillMode = kCAFillModeForwards;
/* 应用动画 */
[view.layer addAnimation: animation forKey:@"animationScaleKey"];
```

6.2.3 隐式动画的实现原理是什么？如何禁用图层的隐式动画

在 iOS 程序开发中，有时仅仅改变了图层或者视图的一个动画属性（例如

第 9 章　iOS 中的网络和多线程编程

在移动互联网时代，几乎所有的应用程序都需要使用到网络请求，只有通过网络和外界进行数据交换、数据更新，应用程序才能保持新鲜与活力。网络编程是实时更新应用程序数据的最常用手段之一。而为了编写高效的网络请求模块，开发者必须能够灵活运用多线程的各种操作。

9.1 iOS 网络编程与多线程基础

iOS 中的多线程编程主要可以分为 3 个层次：NSThread、GCD 和 NSOperation。另外，由于 Objective-C 兼容 C 语言，所以仍然可以使用 C 语言的 POSIX 接口来实现多线程，但需要引入相应的头文件：#include <pthread.h>。

1．NSThread

NSThread 是封装程度最小、最轻量级的多线程编程接口，它使用更灵活，但要手动管理线程的生命周期、线程同步和线程加锁等，开销较大。

NSThread 的使用比较简单，可以动态创建并初始化 NSThread 对象，对其进行设置然后启动；也可以通过 NSThread 的静态方法快速创建并启动新线程；此外 NSObject 基类对象还提供了隐式快速创建 NSThread 线程的 performSelector 系列扩展工具类方法，和一些静态工具接口来控制当前线程以及获取当前线程的一些信息。

下面以在一个 UIViewController 中为例展示 NSThread 的使用方法。代码如下：

```
- (void)viewDidLoad {
    [super viewDidLoad];
    /** NSThread 静态工具方法 **/
    /* 1 是否开启了多线程 */
    BOOL isMultiThreaded = [NSThread isMultiThreaded];
    /* 2 获取当前线程 */
    NSThread *currentThread = [NSThread currentThread];
    /* 3 获取主线程 */
    NSThread *mainThread = [NSThread mainThread];
    /* 4 睡眠当前线程 */
    /* 4.1 线程睡眠 5s */
    [NSThread sleepForTimeInterval:5];
    /* 4.2 线程睡眠到指定时间，效果同上 */
    [NSThread sleepUntilDate:[NSDate dateWithTimeIntervalSinceNow:5]];
    /* 5 退出当前线程，注意不要在主线程调用，防止主线程被 kill 掉 */
    //[NSThread exit];

    /* NSThread 线程对象的基本创建，target 为入口方法所在的对象，selector 为线程入口方法 */
    /* 1 线程实例对象创建与设置 */
    NSThread *newThread= [[NSThread alloc] initWithTarget:self selector:@selector(run) object:nil];
```

```
        /* 设置线程优先级 threadPriority(0~1.0)，该属性即将被抛弃，将使用 qualityOfService 代替 */
        // newThread.threadPriority = 1.0;
        newThread.qualityOfService = NSQualityOfServiceUserInteractive;
        /* 开启线程 */
        [newThread start];
        /* 2 静态方法快速创建并开启新线程 */
        [NSThread detachNewThreadSelector:@selector(run) toTarget:self withObject:nil];
        [NSThread detachNewThreadWithBlock:^{
            NSLog(@"block run...");
        }];

        /** NSObejct 基类隐式创建线程的一些静态工具方法 **/
        /* 1 在当前线程上执行方法，延迟 2s */
        [self performSelector:@selector(run) withObject:nil afterDelay:2.0];
        /* 2 在指定线程上执行方法，不等待当前线程 */
        [self performSelector:@selector(run) onThread:newThread withObject:nil waitUntilDone:NO];
        /* 3 后台异步执行方法 */
        [self performSelectorInBackground:@selector(run) withObject:nil];
        /* 4 在主线程上执行方法 */
        [self performSelectorOnMainThread:@selector(run) withObject:nil waitUntilDone:NO];
    }
    - (void)run {
        NSLog(@"run...");
    }
```

2．GCD

GCD（Grand Central Dispatch），又叫大中央调度，它对线程操作进行了封装，加入了很多新的特性，内部进行了效率优化，提供了简洁的 C 语言接口，使用更加简单高效，也是苹果公司推荐的方式。

这里对 GCD 进行简单提炼，整理出如下必会内容，可以帮助读者快速掌握使用。

1）同步 dispatch_sync 与异步 dispatch_async 任务派发。

2）串行队列与并发队列 dispatch_queue_t。

3）dispatch_once_t 只执行一次。

4）dispatch_after 延后执行。

5）dispatch_group_t 组调度。

（1）串行与并发（Serial 和 Concurrent）

这个概念在创建操作队列的时候有宏定义参数，用来指定创建的是串行队列还是并行队列。

串行指队列内任务一个接一个地执行，任务之间要依次等待不可重合，且添加的任务按照先进先出（FIFO）的顺序执行，但并不是指这就是单线程，只是同一个串行队列内的任务需要依次等待排队执行避免出现竞态条件，仍然可以创建多个串行队列并行地执行任务。也就是说，串行队列内是串行的，串行队列之间仍然是可以并行的，同一个串行队列内的任务的执行顺序是确定的（FIFO），且可以创建任意多个串行队列。

并行指同一个队列先后添加的多个任务可以同时并列执行，任务之间不会相互等待，且这些任务的执行顺序和执行过程不可预测。

京东、淘宝天猫等平台可购买纸质版全书啦，搜索关键词“iOS 程序员面试笔试宝典”可选购，或者扫描下面二维码直接在京东上购书！