

iOS开发者必读资讯

阿里巴巴手淘客户端出品



随着 iOS14 系统的更新, iOS 系统的隐私保护上了一个新的台阶, 用户的隐私得到了更加好的保护; iOS14 系统出现了很多新的特性, Widget 可以让用户的桌面更加丰富, 定制型更加强; Clips 可以让用户在无需安装应用的情况下体验应用; Swift 语言进一步发展, 将进一步促进原生技术的发展。

—— 淘系技术部无线应用开发 彭玉堂 (巴格)



阿里云开发者“藏经阁”
海量免费电子书下载



关注「淘系技术」微信公众号
商业&技术&经验，全面分享
和 20W+程序员共同成长！

I 目录

详解 WWDC 20 SwiftUI 的重大改变及核心优势	4
iOS14 隐私适配及部分解决方案	25
Metal 新特性：大幅度提升 iOS 端性能	45
Swift 5.3 又更新了什么新奇爽快的语法？	67
Apple Widget：下一个顶级流量入口？	76
Swift 5.3 的进化：语法、标准库、调试能力大幅提升	89
WWDC：无线网络优化实践，带来哪些启发？	100
附录	117

详解 WWDC 20 SwiftUI 的重大改变及核心优势

作者 | 姜沂（倾寒）

出品 | 阿里巴巴新零售淘系技术

6月23日凌晨1点，苹果 WWDC20 开发者大会在线上以主题演讲的方式，在 Apple Park 进行直播。

23-26日，苹果公开了100多个面向开发者的视频，内容涵盖 Swift / SwiftUI、App Clips、Widgets、Privacy & Security 等方面。

对于开发者和程序员来说，我们有哪些新发现和新思考？

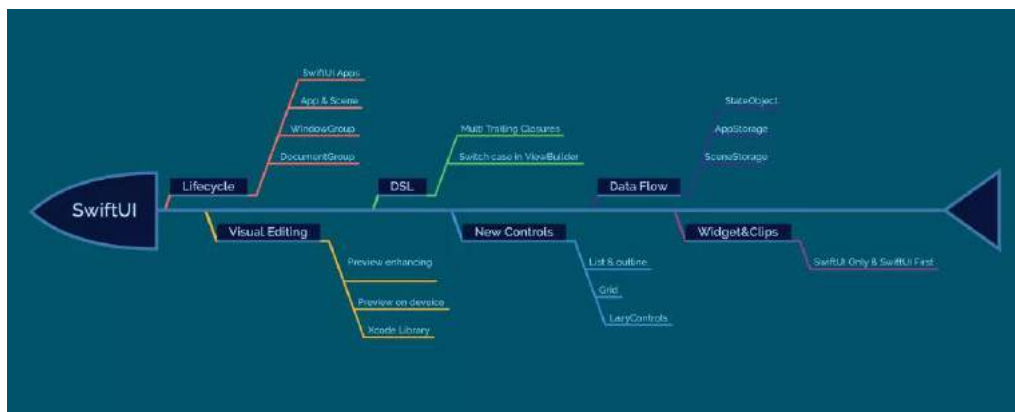
淘系技术客户端团队，将给大家带来一系列关于新系统背后的启发，欢迎交流讨论。

前言

SwiftUI 是苹果公司于 2019 年推出的 Apple Platform 的新一代声明式布局引擎，笔者于去年第一时间升级 Beta 尝鲜全家庭，并在短时间内迅速落地了基于 SwiftUI 的内部 APP，也分享了几篇关于 SwiftUI 的文章，但 SwiftUI 1.0 基本没有任何公司敢用在正式上线的主 APP 上，API 在 Beta 版本之间各种废弃，UI 样式经常不兼容，大列表性能差，彼时都标识着 SwiftUI 还称为一个 Toy Framework。

随着 WWDC 20 相关新特性和介绍视频的释出，都明确的宣告着 SwiftUI 元年已经到了，SwiftUI 已经成长为新时代的布局引擎。

以下从几个方面分享关于 SwiftUI 的重大改变及核心优势。



PS: 需要读者对 Swift 及 SwiftUI 1.0 有一定熟悉。

SwiftUI Apps

苹果在最近几年的动作中一直在搞 Apple Platform 统一的事情，从最近几年的 iPad 多任务 多窗口，到 Mac Catalyst 再到今年更进一步直接推出了 Apple silicon 芯片更是从硬件上做到了真正统一（话外音：你们在软件上玩的那些跨平台的都是小玩意，硬件才是王道）。

还提供了 Rosetta2 Universal2 帮助开发者基本无成本的迁移到新平台上。但是作为软件工程师还是要更多的关注软件生态的变化。首先了解下创建 APP 时的变化。

可以看到创建新工程时有了一套全新的模板基于 SwiftUI App Lifecycle 的跨平台项目。

代码也从原本的基于 UI/NS HostViewController 变成了基于 APP 的声明式描述，下面是代码的前后对比。

■ Before

```
class SceneDelegate: UIResponder, UIWindowSceneDelegate {

    var window: UIWindow?

    func scene(_ scene: UIScene, willConnectTo session: UISceneSession, options connectionOptions: UIScene.ConnectionOptions) {
```

```
let contentView = ContentView()

if let windowScene = scene as? UIWindowScene {

    let window = UIWindow(windowScene: windowScene)

    window.rootViewController = UIHostingController(rootView: contentView)

    self.window = window

    window.makeKeyAndVisible()

}

}
```

■ After

```
import SwiftUI

@main

struct MyApp: App {

    var body: some Scene {

        WindowGroup {

            ContentView()

        }

    }

}
```

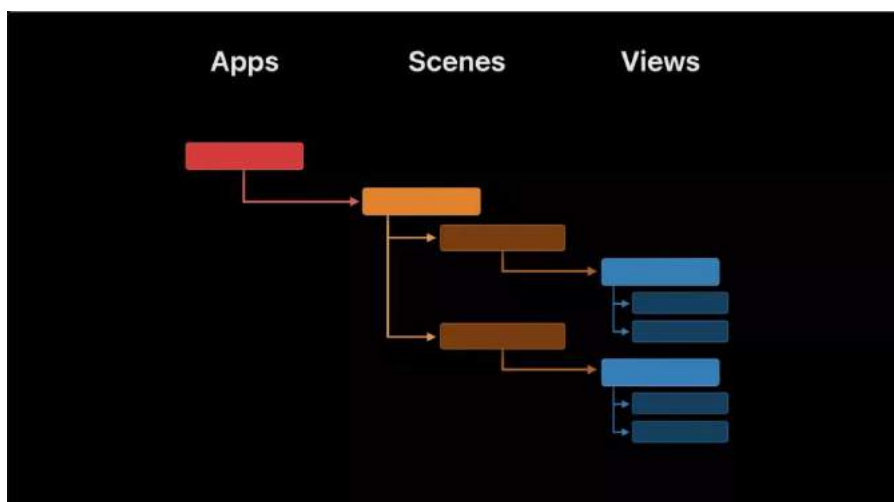
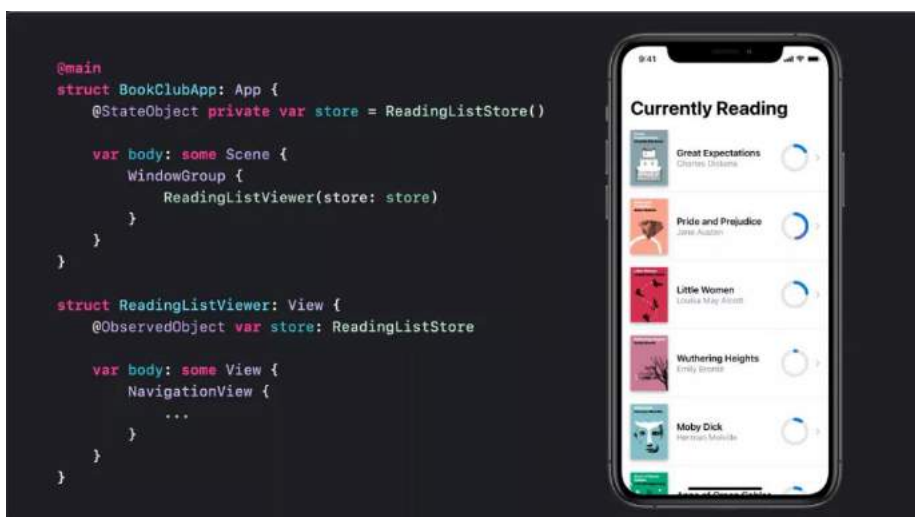
其中@main 是 Swift 5.1 新增的 Attribute 标记了应用程序的入口点，更多请参看 [SE-0281-main-attribute.md](#)

乍看好像只有代码精简了不少，很多人会认为这个简洁程度还不如 Flutter 的 main()
=> runApp(MyApp());.

但最重要的变化是这是第一次跨平台代码，完全无需引入任何 UIKit APPKit WatchKit 等相关 Framework，即可直接运行在不同平台上。这意味着我们后续在 UI 布局系统上可以逐渐摆脱对传统命令式 UI 编程的依赖。达到真正的平台无关。

SwiftUI 将整个原有的平台差异部分抽象为 App 和 Scene，对于一个 mac/iOS/iPad/watch/tv/..应用，来说 App 代表了整个应用，Scene 代表了与 Window 相关的多窗口，有些设备只有一个 Scene 有些则有多个，虽然不同的 OS 确实存在差异，但是在语义层面达到了一致。

其次一个没有历史包袱的 APP，也可以完整的从 Swift APP lifecycle 风格的模板开始，无需再和传统的 UIKit/AppKit 等混合。这也意味着可以达到 APP 完全 Declared and State-Driven。



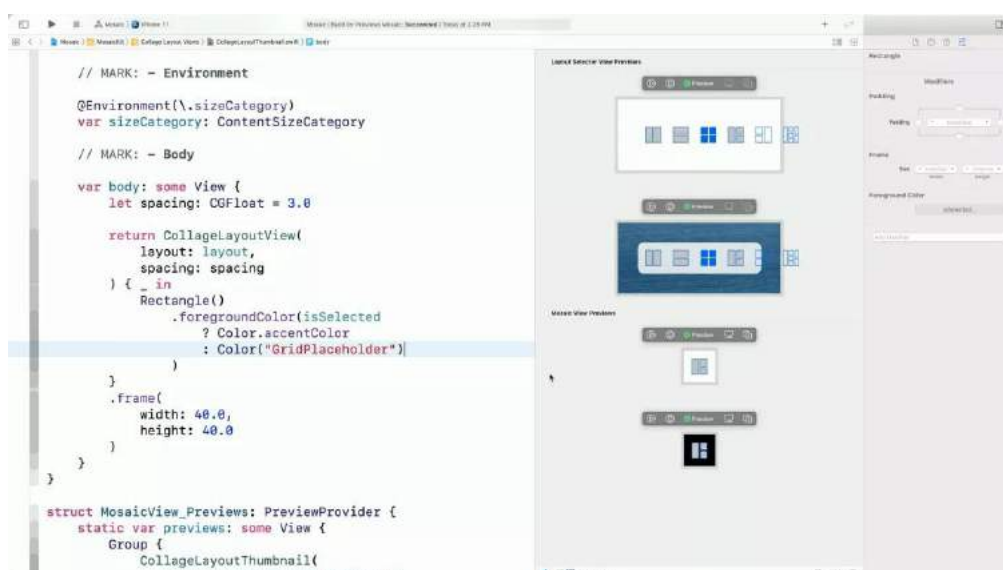
Visual Editing

Preview

在传统的利用 DSL 可视编程框架或者平台，诸如 Web Flutter 等技术，都是开发者编写好对应的代码，运行在对应的平台或者调试工具上。SwiftUI 作为苹果最重要的软件层战略框架，更是和 Xcode 深度结合，在运行之前就可以完整的预览你所编写的界面。

强大的 Preview 可以让你既可以从编写 DSL 到立即预览效果，也可从预览的 Canvas 画布中直接修改效果在代码编辑器中生成代码，这对于日常开发的效率有非常大的提高，尤其是在 UI 微调时，效果尤为突出。

Xcode12 可以在 Canvas 上同时预览多个不同设备环境的界面,也可以直接投射到真实的设备上预览。



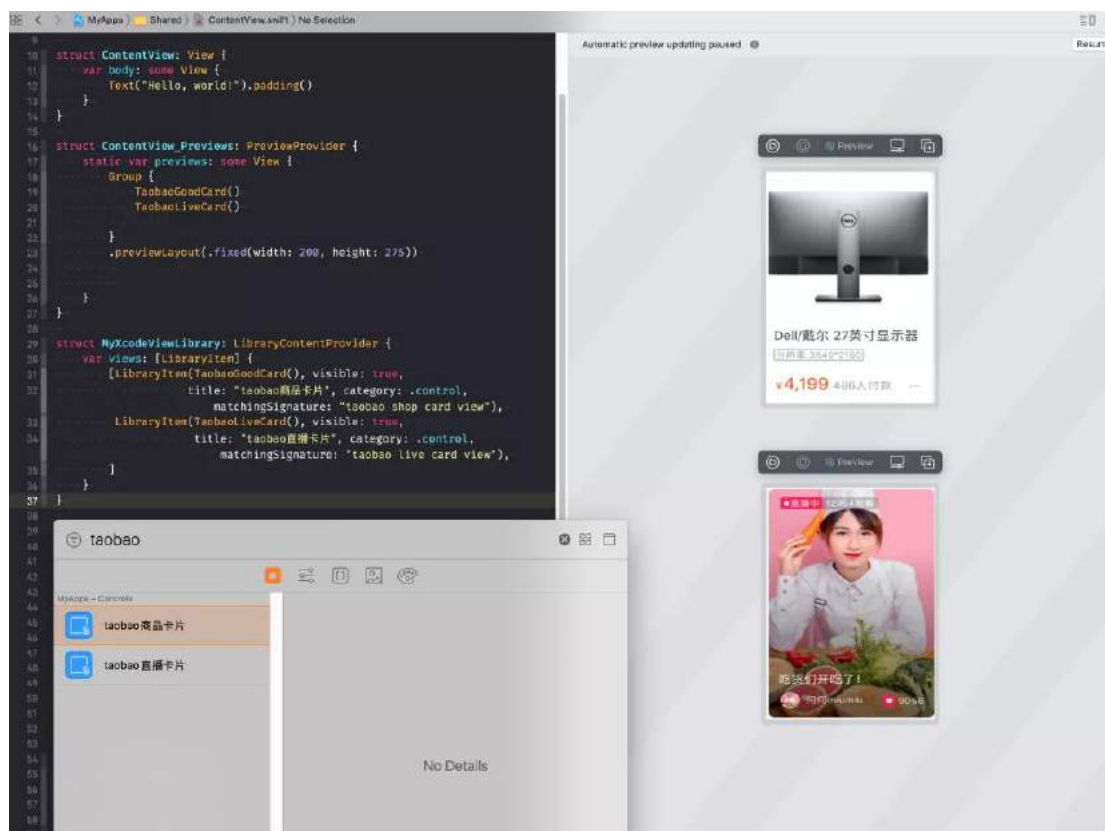
对于日常开发来说，编写一个 UI 界面通常依赖外部的网络/磁盘/其他数据，才能正常的构建，这也造成了 UI 开发虽然是开发中较为简单的一步，但同时也是最耗时的一步，有了预览功能，可以把很多繁琐的工作前置解决掉，对于研发效率会有非常大的提高。

Xcode Library

在编写真实项目中，一个公司的 APP UI 包含成百上千种风格的 View 组件，对于 UI 组件丰富的产品，如果一个新需求可以由现有的组件组合，那么需求交付的时间也会大大缩短。

但是对于一个大型的开发团队而言，一个开发同学是很难知道公司内到底有多少种组件库，而且即便知道有某种组件库，开发同学初期看到的也是代码，一般需要书写一定的 Demo 才可以用眼睛感知到这个组件到底是否是我想要的。

在 Xcode 12 中提供了更强大的工具，一个自定义组件，只需要遵守一个 `LibraryContentProvider` 协议就可被 Xcode 识别，可以像系统控件一样直接从 Xcode 里面识别并预览。对于一个大型团队来说，此功能可以大大提高找寻组件和查看组件样式的效率。



```
// Without trailing closure:

UIView.animate(withDuration: 0.3, animations: {

    self.view.alpha = 0

}, completion: { _ in

    self.view.removeFromSuperview()

})

// With trailing closure

UIView.animate(withDuration: 0.3, animations: {

    self.view.alpha = 0

}) { _ in

    self.view.removeFromSuperview()

}

// Multiple trailing closure arguments

UIView.animate(withDuration: 0.3) {

    self.view.alpha = 0

} completion: { _ in

    self.view.removeFromSuperview()

}
```

DSL

随着 Swift5.3 和 SwiftUI2.0 的推出, SwiftUI 在 DSL 上也更富有表现力, Swift 支持了多重尾闭包语法和在 ViewBuild 里面支持 Switch Case 语句。

Multiple Trailing Closures

虽然社区对多重尾闭包的讨论上一直存在争议问题,但最终 Swift5.3 还是接受并实现了,在普通命令式编程的地方使用会有一定的困惑性,但是在 SwiftUI 中 DSL 也更有声明式的味道。

```
// Without trailing closure:

UIView.animate(withDuration: 0.3, animations: {

    self.view.alpha = 0

}, completion: { _ in

    self.view.removeFromSuperview()

})

// With trailing closure

UIView.animate(withDuration: 0.3, animations: {

    self.view.alpha = 0

}) { _ in

    self.view.removeFromSuperview()

}

// Multiple trailing closure arguments

UIView.animate(withDuration: 0.3) {

    self.view.alpha = 0

} completion: { _ in

    self.view.removeFromSuperview()

}
```

Switch Case Support

在 SwiftUI 的 ViewBuilder DSL 体系中也支持了 Switch case 语法。

```
var body: some View {  
  
    switch c {  
  
        case .a:  
  
            return Text("A")  
  
        case .b:  
  
            return Text("B")  
  
        case .c:  
  
            return Text("C")  
  
    }  
  
}
```

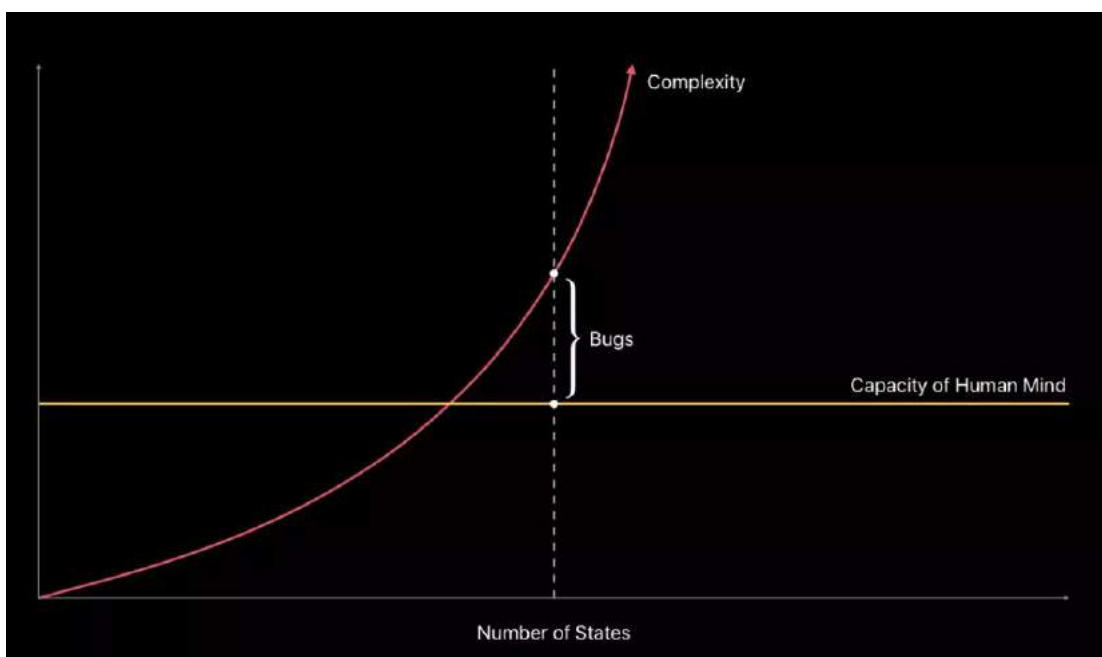
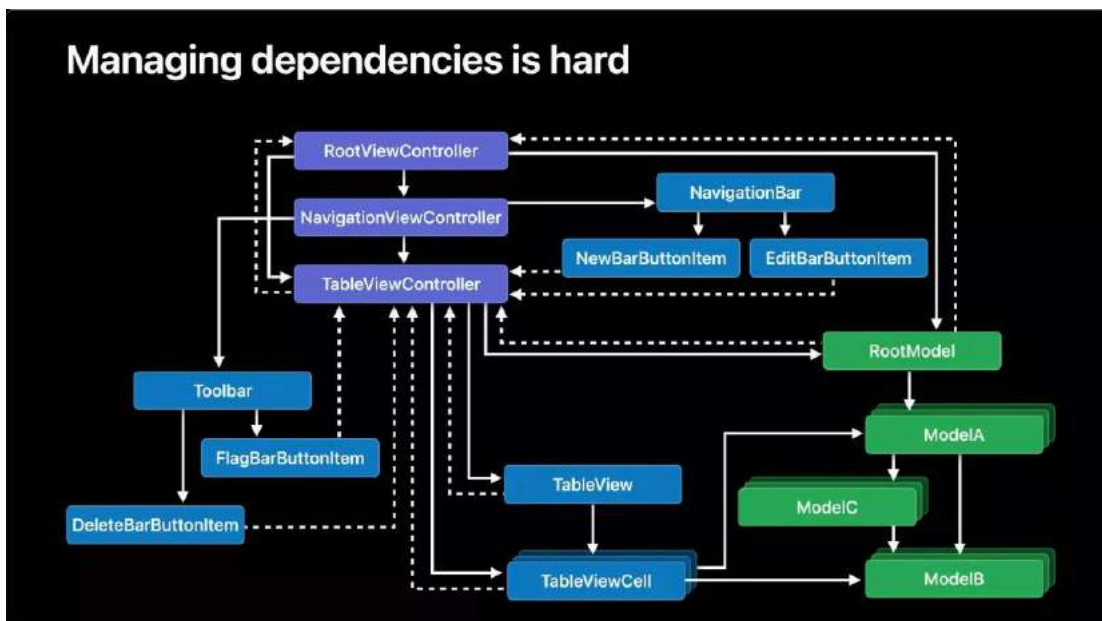
Data Flow

在使用传统命令式编程编写 UI 代码时，开发者需要手动处理 UIView 和 数据之间的依赖关系，每当一个 UIView 使用了外部的数据源，就表明了 UIView 对外部的数据产生了依赖，当一个数据产生变化时，如果意外的没有同步 UIView 的状态，那么 Bug 就产生了。

处理简单的依赖关系是可控的，但是在真实项目中，视图之间的依赖关系是非常复杂的，假设一个视图只有 4 种状态，组合起来就有 16 种，再加上时序的不同，情况就更加复杂。

人脑处理状态的复杂度是有限的，状态的复杂度一旦超过人脑的复杂度，就会产生大量的 Bug，并且修掉了这个产生了新的 Bug。

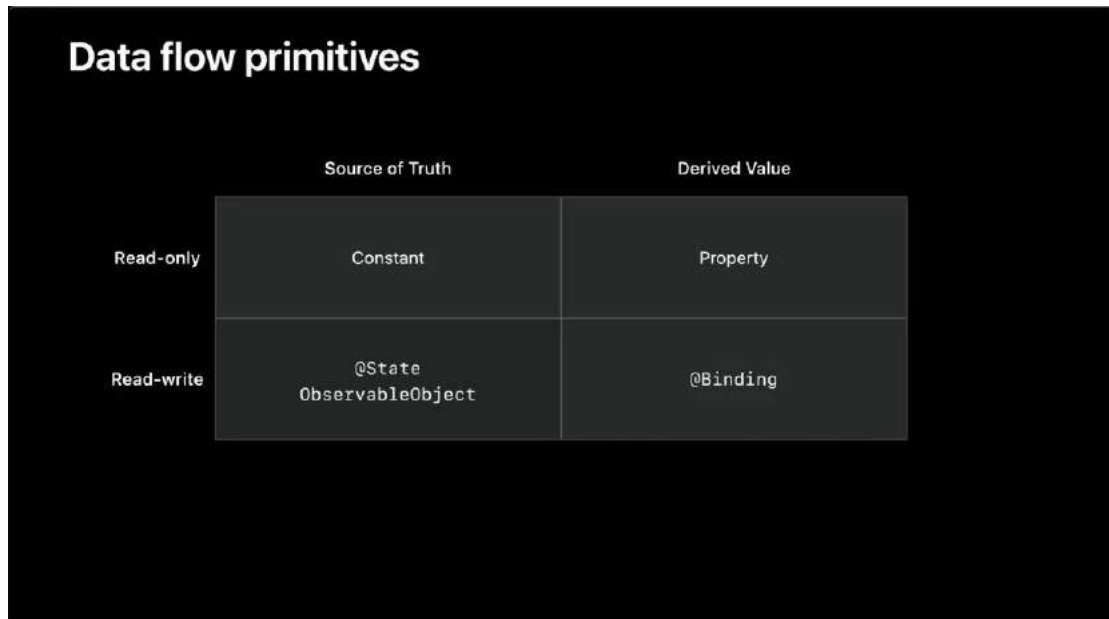
Managing dependencies is hard



那么 SwiftUI 是如何解决这个问题的？

SwiftUI 的框架提供了几个核心概念：

1. 统一的 body 属性, SwiftUI 自动从当前 App 状态集自动生成基于当前状态的快照 View。
2. 统一的数据流动原语。



关于 SwiftUI 中的 Data Flow 是如何消除视图和状态不一致的，请参考去年撰写的文档系列文章《深度解读 SwiftUI 背后那些事儿》。

今年 SwiftUI 2.0 新增的 StateObject 数据流原语让 SwiftUI 在重复创建 View 时避免重复创建 ObservableObject 从而提高 View 重建的性能。

SceneStorage 和 APPStorage 让一些可持久化的数据变得更加简单且具有语义化。

New Controls

前面提到的，新增的 DSL 语法 SwiftUI App Lifecycle，以及 Xcode Library Preview 其实本质上都是对去年 SwiftUI 1.0 锦上添花的新扩展。

真正重要的是今年新增的各类新控件，其中通过导出来自 Xcode11.5 和 Xcode12.0 beta 版本的 Swift 声明文件，可以观察到整个声明文件从原来的 10769 行增加到 20564 行。

新增了约 87 个 struct 16 个 protocol。有了这些丰富的组件才可以更好的构建我们的 APP。

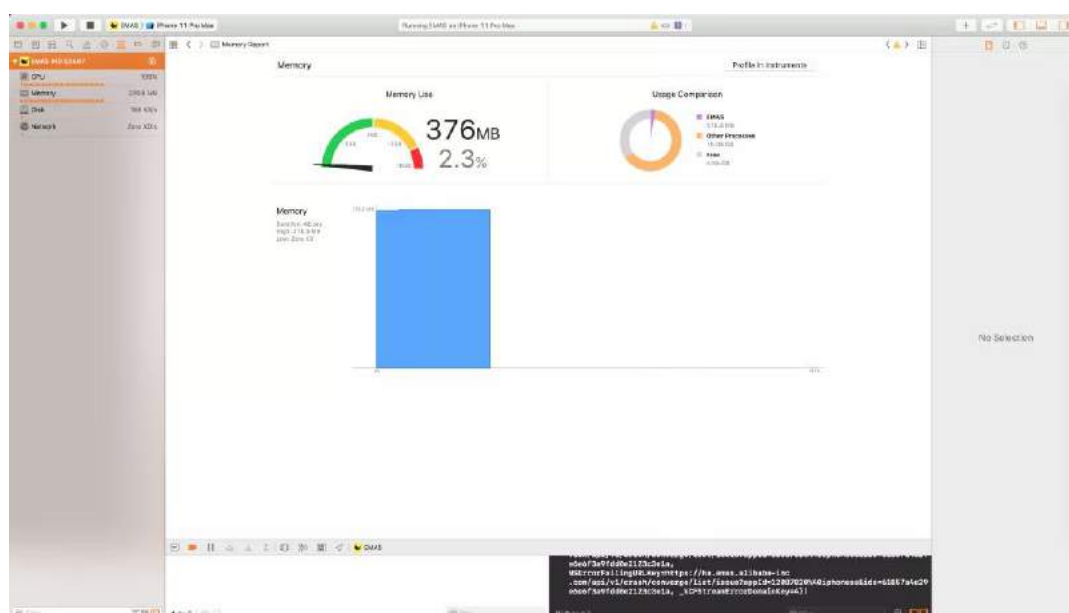
大列表组件

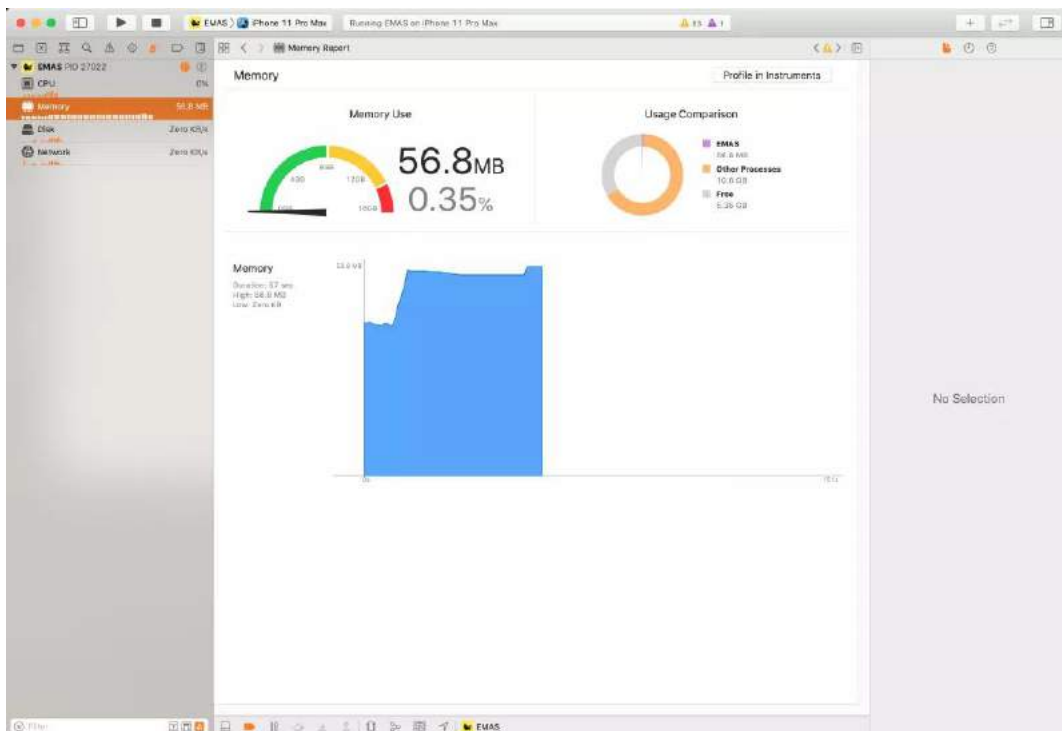
在任何一款 APP 中都会存在类似大列表组件，如淘宝 APP 里面的某家店铺里面商品列表流，首页的信息流，都是具有超长内容的列表页数据。对于长列表页来说，过长的 UI 页面会导致过多的内存占用，在用户的设备中，内存是最为重要的指标，对于目前国内的 APP 市场，低端手机仍然占据大量的市场，对于这些设备来说，一旦内存超标，APP 就很容易 OOM，这会导致用户体验非常差，在现有竞争关系激烈的市场环境下，体验差意味着会失去用户。

对于传统的命令式编程来说，我们可以主动控制 UITableViewCell 的重用，自建缓冲池等一系列手段去优化我们的 APP 内存占用，但是对于 SwiftUI 1.0 来说，系统提供的控件并没有有效的办法去让我们控制页面的渲染，对于大列表页面就容易出现内存占用过高的问题。

SwiftUI 2.0 推出了 LazyHStack 和 lazyVStack 加上 List 渲染模式默认就是 Lazy 的直接解决了最大的性能问题。

笔者以去年使用 SwiftUI 编写的 Emas App 为例，当列表页（并无大图）加载到 500 个时，APP 使用内存已经达到了将近 360MB。而只需要切换到 Xcode12 API 调整为到 LazyVStack 内存占用直接降低 300MB。





Widget and Clips

苹果与 WWDC 20 推出的 WidgetKit 支持的 API 是 SwiftUI Only, 虽然已经可以混合部分 UIKit 里面的 View, 但相信没有历史包袱 最低支持版本为 iOS14 的 Widget 没有人会选择笨重的命令式 API。

同理 Clips 也一样。这里因为篇幅原因就不做展开, 后续会有专门的文章分析相关技术。

Swift & SwiftUI 的机会在哪里?

笔者曾经在公司推动集团升级了基建, 支持了 Swift 开发环境也在淘宝落地了一些场景, 但是集团内一直有一些质疑的声音, 引入 Swift 到底有什么用?

SwiftUI 又是 N 年后才可以用上的小玩意, Objective-C 不够用吗? 现在笔者可以回答这些质疑的声音, Swift 未来的机会在 效率, 体验和苹果的技术红利。

效率

从研发效率上来说，Swift 对比 Objective-C 的精简程度不言而喻，笔者在淘宝 APP 上线的模块代码量下降了 40 %。

但更进一步，如果编写 UI 界面从 UIKit 转向了 SwiftUI 代码量直接少了不止一倍。更少的代码意味着更快的交付，在目前竞争激烈的市场会有更多的试错场景。

关于使用 UIKit 编写代码转向 SwiftUI 的代码量对比，读者可以参考开源 APP [MovieSwiftUI](#) 直观了解。

体验

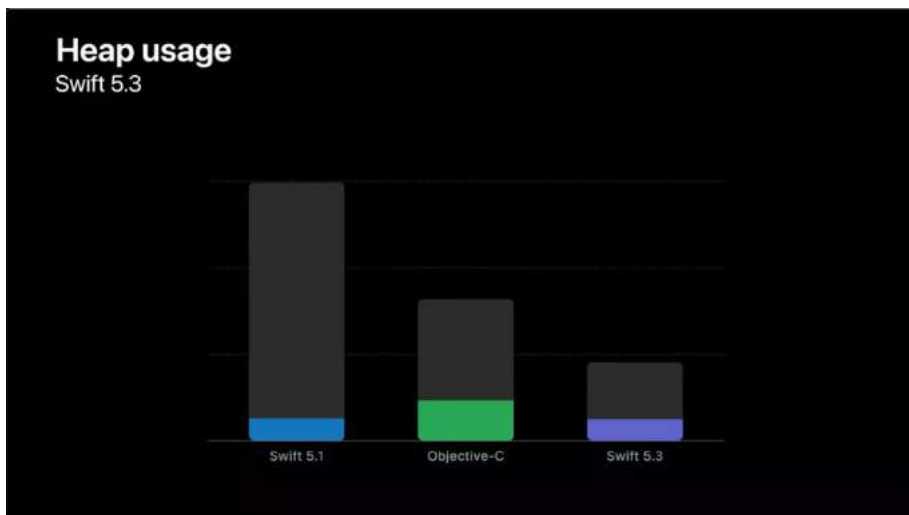
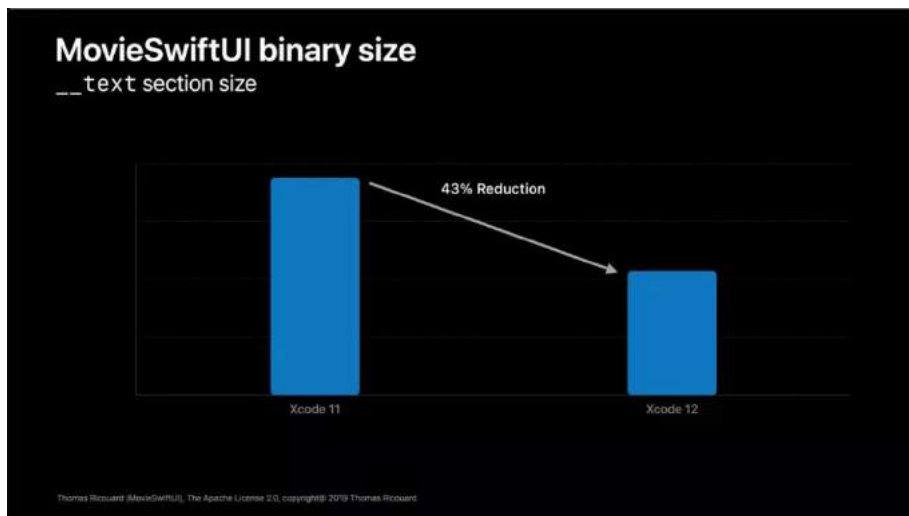
读者可能比较困惑对于切换语言和框架，对体验看上去没有任何帮助，但事实真是这样吗？

首先引入 Swift 后，由于 Swift 语言设计之初便对安全性列为最重要的目标，Swift 的引入会让代码尽可能的减少未定义的行为，减少 Crash 意味着 APP 的稳定性提高，体验自然更佳。

其次虽然 Swift 同样的语言出于对安全性考虑编译处理的指令会比 Objective-C 更多，但是如果 UI 部分都用 SwiftUI 来写呢？

更少的代码意味着更小的包大小，目前国内巨头 APP iOS 端 APP 包大小都朝着 200 MB 奔去，如果能减少更多的代码对包大小也可以在 200MB 的限制下承载更多而业务。对用户的体验也有较大的提升。

更进一步由于 Swift 选择使用值类型构建整个 APP，值类型的有点在于更扁平化的内联数据结构去分配内存，而不是使用更多间接指针引用，减少了大量不必要的堆内存消耗，意味着整体内存使用量的降低。对整个 APP 的稳定性也有较大的提高。



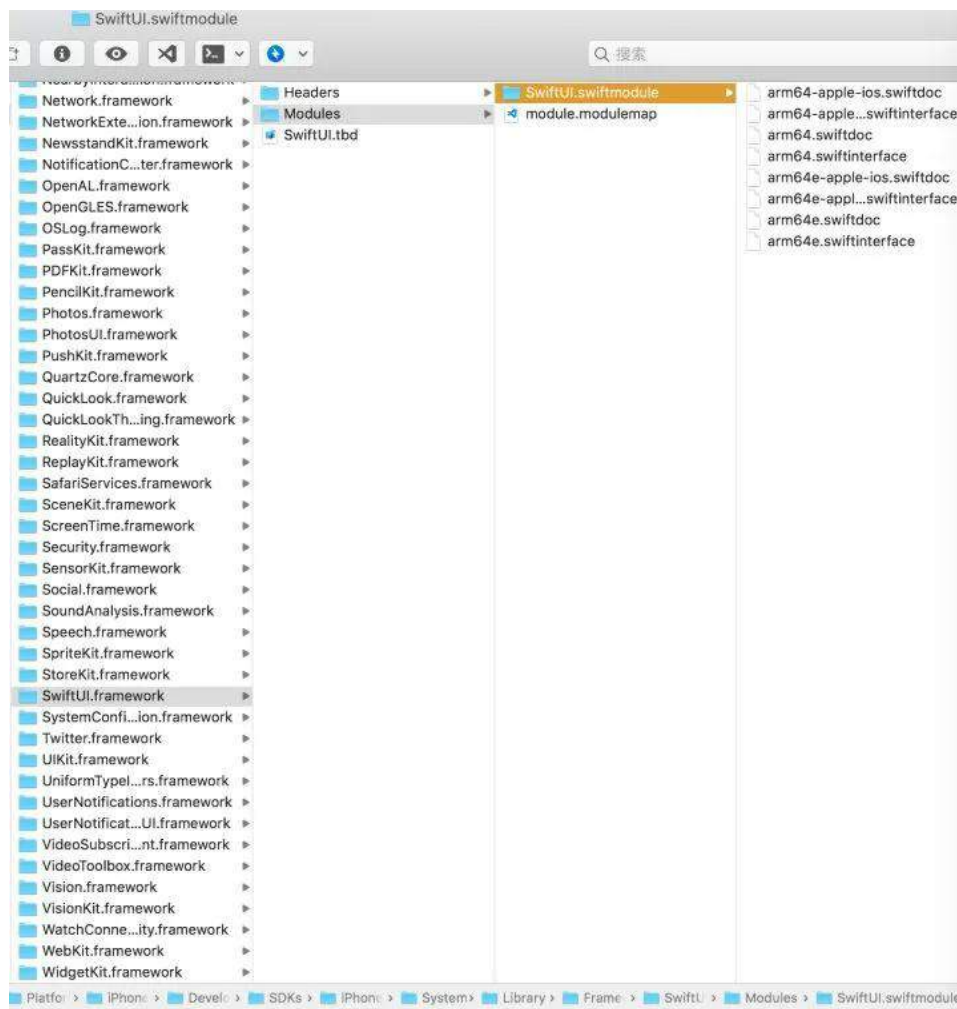
苹果的选择

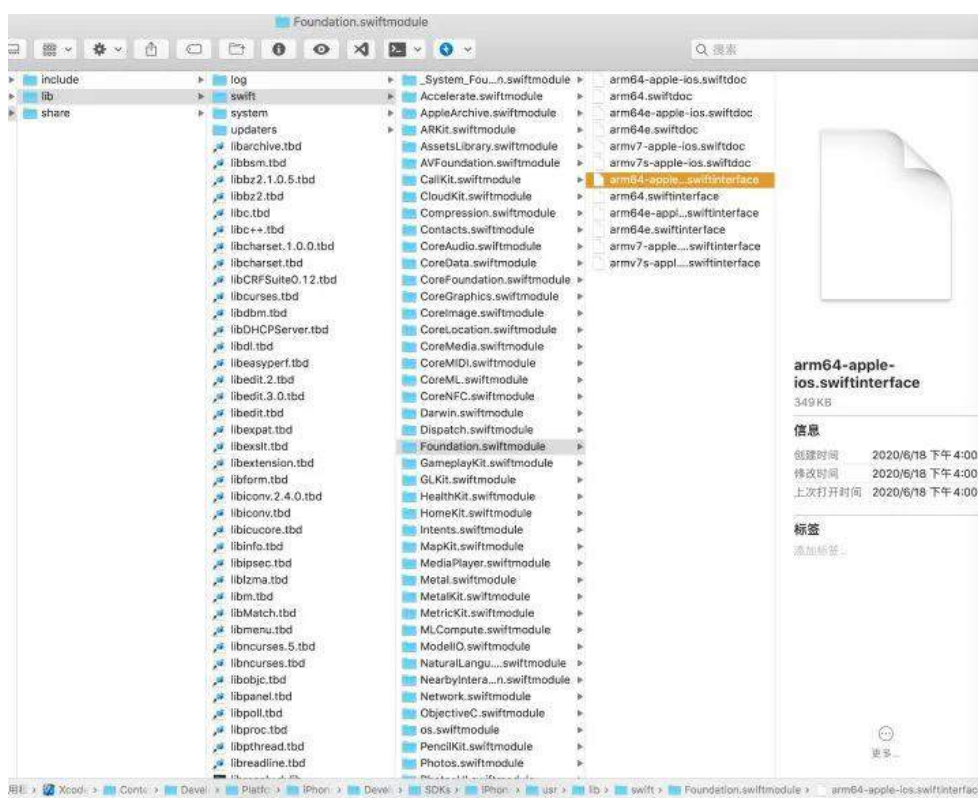
Swift 作为苹果的战略语言已经发展的越来越壮大，自 2019 年 Swift ABI 稳定后，苹果在 Swift 的投入越来越大。我们可以进入 [/Applications/Xcode-beta.app/Contents/Developer/Platforms/iPhoneOS.platform/Developer/SDKs/iPhoneOS.sdk/usr/lib/swift](#) , [/Applications/Xcode-beta.app/Contents/Developer/Platforms/iPhoneOS.platform/Developer/SDKs/iPhoneOS.sdk/System/Library/Frameworks](#) , <https://github.com/apple> 和 <https://github.com/swift-server> 看到，自 iOS 13 以来 苹果新增了约 10+ Pure Swift Library , 10+ Open Source Swift Library, 以及针对 144 个公开 Framework, 根据 Swift Style 重新设计了 57 个 Framework 的 API。

从以下数据：

1. 从 WWDC17 后 苹果已经不再使用 Objective-C 做 Sample Code 演示；
2. <https://developer.apple.com/>不再更新 Objective-C 相关的文档；
3. WidgetKit 是 SwiftUI only；
4. App Clips 10M 的包大小， SwiftUI 是最合适的框架；
5. 开源社区逐步放弃 Objective-C 如 Lottie；

可以判断，Swift 是未来 Apple 平台的唯一选择，越是有包袱的大厂 APP，从现在还不尽早储备，在未來越会寸步难行。





我们需要做些什么？

Swift

我们已经做了什么？

1. 一套支持 Swift 二进制的研发环境。
2. 300+ 支持了混编的淘系 SDK。
3. 手淘落地了 6 个模块。
4. 集团新增了约 20 个支持 Swift 的 APP。
5. 10 多场技术培训。
6. 169+ 语雀知识沉淀。
7. 300+ 工程师的集团 Swift 官方组织。

8. 2 个 技术创新产品。

经过去年横向组织大家共同的努力，我们已经已经支持了横向大基建。包括研发环境，工具支撑，沉淀了大量的文档，还有相关的技术课程。

■ 要朝什么方向去努力

目前集团对于 Swift 的呼声越来越高，我们大量的工程师希望的去使用 Swift 。目前首先要做的事情是依托 Swift 和 SPM 提升我们的开发体验，升级我们的中间件，使业务可以大量的用起来 Swift ，提高我们的研发效率和代码质量。

1. 升级基于 SPM 的新的包管理体系。
2. 升级老旧基础库，打磨新一代基建。
3. 引入新的 Swift 特有库 赋能业务。

SwiftUI

虽然前文提到了 SwiftUI 的众多优点，包括研发效率，体验的提高，但是在国内的环境中 SwiftUI 也有它致命的弊端，iOS 14 才可放心的使用。

只支持 Apple Platform，这和国内的要支持 Mobile Platform 从理念上冲突。

大型 APP 要解决的是如何部署到低版本操作系统上和安卓平台上，毕竟很多公司还在支持 iOS 9 对于升级到最低支持 iOS 14 好像还需要一个世纪那么漫长，而且国内的设备占比大头还是以 Android 居多 。

虽然可以看到 Swift 语言也在逐渐支持 Android 平台，但是也看到苹果对于安卓平台的 SwiftUI 并没有太大兴趣。

从体验上 Flutter 远不如 SwiftUI 这种亲儿子效果好， 但对于国内跨端欲望旺盛的市场来说 SwiftUI 还是比不过 Flutter， 不过既然 SwiftUI DSL 层已经基本固定，那么也有可能投入人力直接在低版本操作系统上实现一套自建的 SwiftUI 引擎，或者将 SwiftUI 引擎移植到安卓平台，比如对接 Flutter 或者直接对接 Android Native。

比起 Flutter 引入双端带来的包大小增量和体验不一致的情况，SwiftUI 保留 iOS 平台体验，只侵入一端的选择显然要更好一点。

不过短期内我们可以在 Clips 和 Widget 场景下开始使用 SwiftUI，毕竟 SwiftUI 快速的开发效率对和较低的包大小占用非常适合这样的场景，我们可以在业务场景中练兵储备我们的 SwiftUI，并积极在主 APP 中尝试。

参考

SwiftUI 背后那些事儿

https://mp.weixin.qq.com/s/ciiauLB__o-cXXfKn7IL1Q?spm=a2c6h.12873639.0.0.61617cf0CNuDWM

MovieSwiftUI

<https://github.com/Dimillian/MovieSwiftUI?spm=a2c6h.12873639.0.0.61617cf0dLMYPd>

SE-0281-main-attribute.md

<https://github.com/apple/swift-evolution/blob/master/proposals/0281-main-attribute.md?spm=a2c6h.12873639.0.0.61617cf0cvHdQx&file=0281-main-attribute.md>

Add custom views and modifiers to the Xcode Library

<https://developer.apple.com/videos/play/wwdc2020/10649/>

Structure your app for SwiftUI previews

<https://developer.apple.com/videos/play/wwdc2020/10149/>

Introduction to SwiftUI

<https://developer.apple.com/videos/play/wwdc2020/10119/>

What's new in SwiftUI

<https://developer.apple.com/videos/play/wwdc2020/10041/>

App essentials in SwiftUI

<https://developer.apple.com/videos/play/wwdc2020/10037/?spm=a2c6h.12873639.0.0.61617cf0K8NktJ>

Visually edit SwiftUI views

<https://developer.apple.com/videos/play/wwdc2020/10185/>

Stacks, Grids, and Outlines in SwiftUI

<https://developer.apple.com/videos/play/wwdc2020/10031/>

[Build document-based apps in SwiftUI](#)

<https://developer.apple.com/videos/play/wwdc2020/10039/>

[Data Essentials in SwiftUI](#)

<https://developer.apple.com/videos/play/wwdc2020/10040/>

[Build a SwiftUI view in Swift Playground](#)

<https://developer.apple.com/videos/play/wwdc2020/10643/>

[Build SwiftUI apps for tvOS](#)

<https://developer.apple.com/videos/play/wwdc2020/10042/>

[Build SwiftUI views for widgets](#)

<https://developer.apple.com/videos/play/wwdc2020/10033/>

[Build complications in SwiftUI](#)

<https://developer.apple.com/videos/play/wwdc2020/10048/>

[What's new in Swift](#)

<https://developer.apple.com/videos/play/wwdc2020/10170/>

[Swift packages: Resources and localization](#)

<https://developer.apple.com/videos/play/wwdc2020/10169/>

[Distribute binary frameworks as Swift packages](#)

<https://developer.apple.com/videos/play/wwdc2020/10147/>

[Explore logging in Swift](#)

<https://developer.apple.com/videos/play/wwdc2020/10168/>

[Create Swift Playgrounds content for iPad and Mac](#)

<https://developer.apple.com/videos/play/wwdc2020/10654/>

[Embrace Swift type inference](#)

<https://developer.apple.com/videos/play/wwdc2020/10165/?spm=a2c6h.12873639.0.0.61617cf0Syt53f>

[Explore numerical computing in Swift](#)

<https://developer.apple.com/videos/play/wwdc2020/10217/>

[Unsafe Swift](#)

<https://developer.apple.com/videos/play/wwdc2020/10648/>

[Safely manage pointers in Swift](#)

<https://developer.apple.com/videos/play/wwdc2020/10167/>

[Explore numerical computing in Swift](#)

<https://developer.apple.com/videos/play/wwdc2020/10217/?spm=ata.13261165.0.0.58fb628dEvvzNy>

[Explore Packages and Projects with Xcode Playgrounds](#)

<https://developer.apple.com/videos/play/wwdc2020/10096/?spm=ata.13261165.0.0.58fb628dEvvzNy>

[Use Swift on AWS Lambda with Xcode](#)

<https://developer.apple.com/videos/play/wwdc2020/10644/?spm=ata.13261165.0.0.58fb628dEvvzNy>

iOS14 隐私适配及部分解决方案

作者 | 盛兰雅（岚遥）

出品 | 阿里巴巴新零售淘系技术

在刚刚结束的线上 WWDC 2020 发布会上苹果向我们展示了新的 iOS14 系统。iOS14 的适配，很重要的一环就集中在用户隐私和安全方面。

在 iOS13 及以前，当用户首次访问应用程序时，会被要求开放大量权限，比如相册、定位、联系人，实际上该应用可能仅仅需要一个选择图片功能，却被要求开放整个照片库的权限，这确实是不合理的。对于相册，在 iOS14 中引入了 “LimitedPhotos Library” 的概念，用户可以授予应用访问其一部分的照片，对于应用来说，仅能读取到用户选择让应用来读取的照片，让我们看到了 Apple 对于用户隐私的尊重。这仅仅是一部分，在 iOS14 中，可以看到诸多类似的保护用户隐私的措施，也需要我们升级适配。

最近在调研 iOS14 的适配方案，本文主要分享一下 iOS14 上对于隐私授权的变更和部分适配方案，欢迎补充指正。

适配点

相册

iOS14 新增了 “Limited Photo Library Access” 模式，在授权弹窗中增加了 Select Photo 选项。用户可以在 App 请求调用相册时选择部分照片让 App 读取。从 App 的视角来看，你的相册里就只有这几张照片，App 无法得知其它照片的存在。



iOS14 中当用户选择

“PHAuthorizationStatusLimited” 时，如果未进行适配，有可能会在每次触发相册功能时都进行弹窗询问用户是否需要修改照片权限。

对于这种情况可通过在 Info.plist 中设置

“PHPhotoLibraryPreventAutomaticLimitedAccessAlert” 的值为 YES 来阻止该弹窗反复弹出，并且可通过下面这个 API 来主动控制何时弹出 PHPickerViewController 进行照片选择。

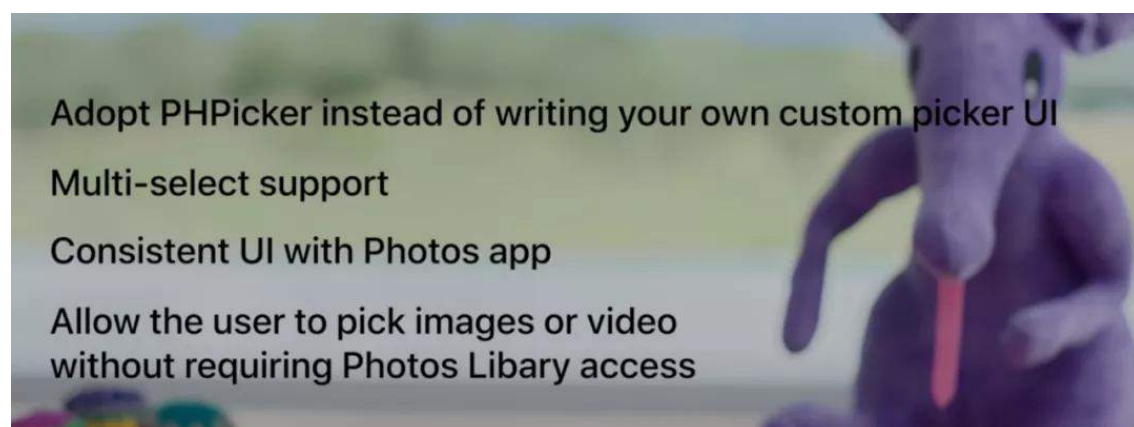
```
[[PHPhotoLibrary sharedPhotoLibrary] presentLimitedLibraryPickerFromViewController:self];
```

在 iOS14 中官方推荐使用 PHPicker 来替代原 API 进行图片选择。PHPicker 为独立进程，会在视图最顶层进行展示，应用内无法对其进行截图也无法直接访问到其内的数据。

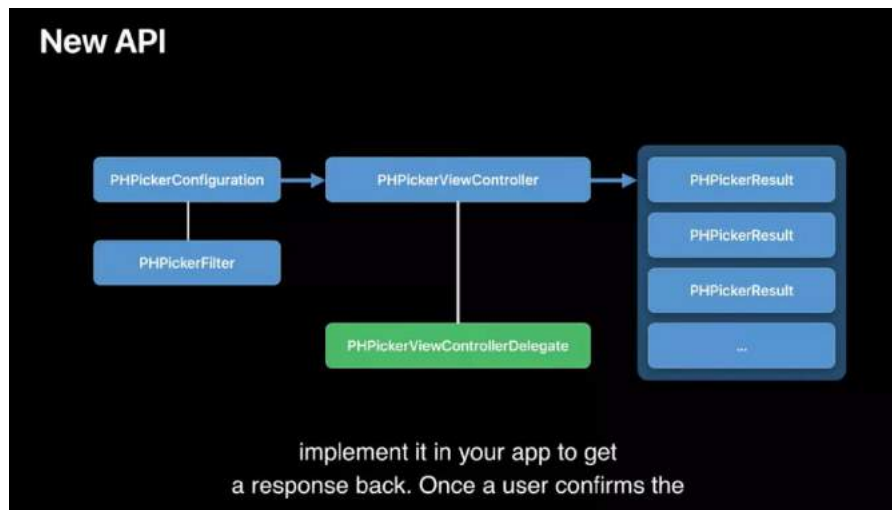
- UIImagePickerController -> PHPickerViewController， UIImagePickerController 功能受限，每次只能选择一张图片，将逐渐被废弃。

```
typedef NS_ENUM(NSUInteger, UIImagePickerControllerSourceType) {  
    UIImagePickerControllerSourceTypePhotoLibrary API_DEPRECATED("Will be removed in a future release, use PHPicker.", ios(2, API_TO_BE_DEPRECATED)),  
    UIImagePickerControllerSourceTypeCamera,  
    UIImagePickerControllerSourceTypeSavedPhotosAlbum API_DEPRECATED("Will be removed in a future release, use PHPicker.", ios(2, API_TO_BE_DEPRECATED)),  
    API_UNAVAILABLE(tvos);  
};
```

- PHPicker 支持多选，支持搜索，支持按 image, video, livePhotos 等进行选择。



新 API 及迁移 demo:



```

@interface ViewController () <PHPickerViewControllerDelegate>

@property (weak, nonatomic) IBOutlet UIImageView *imageView;

@property (nonatomic, strong) NSArray<NSItemProvider *> *itemProviders;

@end

@implementation ViewController

- (void)viewDidLoad {

    [super viewDidLoad];

    // Do any additional setup after loading the view.

}

- (IBAction)button:(id)sender {

    // 以下 API 仅为 iOS14 only

    PHPickerConfiguration *configuration = [[PHPickerConfiguration alloc] init];

    configuration.filter = [PHPickerFilter videosFilter]; // 可配置查询用户相册中文件的类型,
支持三种

    configuration.selectionLimit = 0; // 默认为1, 为0时表示可多选。

    PHPickerViewController *picker = [[PHPickerViewController alloc]
initWithConfiguration:configuration];
  
```

```
picker.delegate = self;

// picker vc, 在选完图片后需要在回调中手动 dismiss

[self presentViewController:picker animated:YES completion:^(

    });

}

#pragma mark - Delegate

- (void)picker:(PHPickerViewController *)picker didFinishPicking:(NSArray<PHPickerResult *>
*)results {

    [picker dismissViewControllerAnimated:YES completion:nil];

    if (!results || !results.count) {

        return;

    }

    NSItemProvider *itemProvider = results.firstObject.itemProvider;

    if ([itemProvider canLoadObjectOfClass:UIImage.class]) {

        __weak typeof(self) weakSelf = self;

        [itemProvider loadObjectOfClass:UIImage.class completionHandler:^(__kindof
id<NSItemProviderReading> _Nullable object, NSError * _Nullable error) {

            if ([object isKindOfClass:UIImage.class]) {

                __strong typeof(self) strongSelf = weakSelf;

                dispatch_async(dispatch_get_main_queue(), ^{

                    strongSelf.imageView.image = (UIImage *)object;

                });

            }

        }]);

    }

}
```

需要注意的是，在 limit Photo 模式下，AssetsLibrary 访问相册会失败；在 write Only 模式下，AssetLibrary 也会有显示问题。建议还在使用 AssetsLibrary 的同学尽快迁移到新 API。

授权相关：旧 API 废弃，增加 PHAccessLevel 参数。如果再使用以前的 API 来获取权限状态，

PHAuthorizationStatusLimited 状态下也会返回

PHAuthorizationStatusAuthorized

```
#pragma mark -
OS_EXPORT
@interface PHPhotoLibrary : NSObject

+ (PHPhotoLibrary *)sharedPhotoLibrary;

#pragma mark - Library access authorization status

/// Replaces \c +authorizationStatus to support add-only/read-write access level status
+ (PHAuthorizationStatus)authorizationStatusForAccessLevel:(PHAccessLevel)accessLevel API_AVAILABLE(macos(10.16),
ios(14), tvos(14));
+ (void)requestAuthorizationForAccessLevel:(PHAccessLevel)accessLevel handler:(void (^)(PHAuthorizationStatus
status))handler API_AVAILABLE(macos(10.16), ios(14), tvos(14));

/// Deprecated and replaced by authorizationStatusForAccessLevel, will return \c PHAuthorizationStatusAuthorized if the user has chosen limited photo
library access
+ (PHAuthorizationStatus)authorizationStatus API_DEPRECATED_WITH_REPLACEMENT("+authorizationStatusForAccessLevel:",
ios(8, API_TO_BE_DEPRECATED), macos(10.13, API_TO_BE_DEPRECATED), tvos(10, API_TO_BE_DEPRECATED));
+ (void)requestAuthorization:(void (^)(PHAuthorizationStatus status))handler
API_DEPRECATED_WITH_REPLACEMENT("+requestAuthorizationForAccessLevel:handler:", ios(8, API_TO_BE_DEPRECATED),
macos(10.13, API_TO_BE_DEPRECATED), tvos(10, API_TO_BE_DEPRECATED));
```

```
typedef NS_ENUM(NSInteger, PHAccessLevel) {

    PHAccessLevelAddOnly = 1, // 仅允许添加照片

    PHAccessLevelReadWrite = 2, // 允许访问照片, limitedLevel 必须为 readWrite

} API_AVAILABLE(macos(10.16), ios(14), tvos(14));

// 查询权限

PHAccessLevel level = PHAccessLevelReadWrite;

PHAuthorizationStatus status = [PHPhotoLibrary authorizationStatusForAccessLevel:level];

switch (status) {

    case PHAuthorizationStatusLimited:

        NSLog(@"limited");

        break;

    case PHAuthorizationStatusDenied:
```



```
        NSLog(@"denied");

        break;

    case PHAuthorizationStatusAuthorized:

        NSLog(@"authorized");

        break;

    default:

        break;

}

// 请求权限, 需注意 limited 权限尽在 accessLevel 为 readAndWrite 时生效

[PHPhotoLibrary requestAuthorizationForAccessLevel:level handler:^(PHAuthorizationStatus
status) {

    switch (status) {

        case PHAuthorizationStatusLimited:

            NSLog(@"limited");

            break;

        case PHAuthorizationStatusDenied:

            NSLog(@"denied");

            break;

        case PHAuthorizationStatusAuthorized:

            NSLog(@"authorized");

            break;

        default:

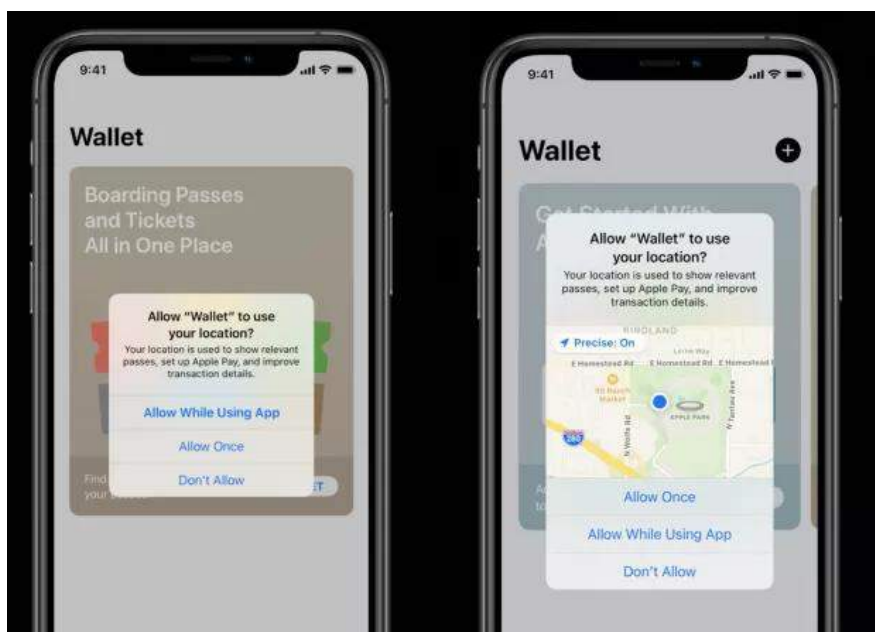
            break;

    }

}];
```

定位

在 iOS13 及以前, App 请求用户定位授权时为如下形态: 一旦用户同意应用获取定位信息, 当前应用就可以获取到用户的精确定位。



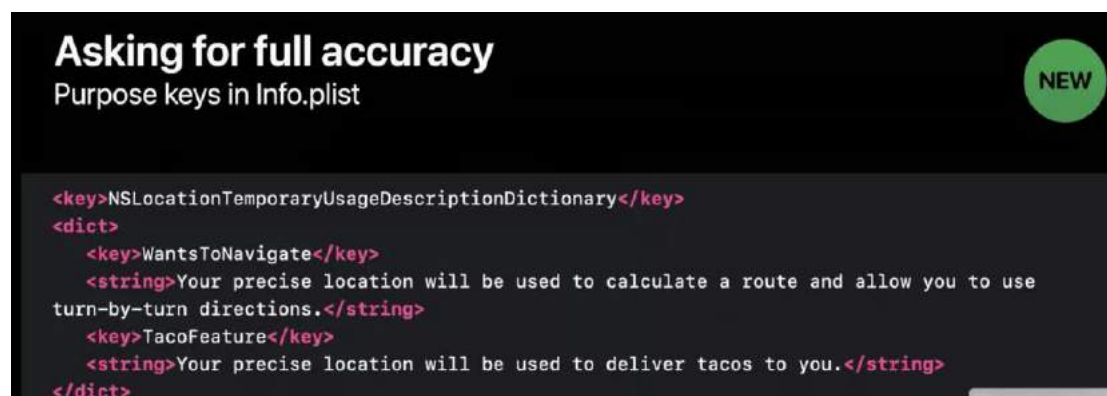
iOS14 新增用户大致位置选项可供用户选择, 原因是大多数 App 实际上并不需要获取用户到用户最准确的定位信息。iOS14 授权弹窗新增的 Precise 的开关默认会选中精确位置。用户通过这个开关可以进行更改, 当把这个值设为 On 时, 地图上会显示精确位置; 切换为 Off 时, 将显示用户的大致位置。

对于对用户位置敏感度不高的 App 来说, 这个似乎无影响, 但是对于强依赖精确位置的 App 适配工作就显得非常重要了。可以通过用户在“隐私设置”中设置来开启精确定位, 但是可能用户宁可放弃使用这个应用也不愿意开启。这个时候, iOS14 在 CLLocationManager 新增两个方法可用于向用户申请临时开启一次精确位置权限。

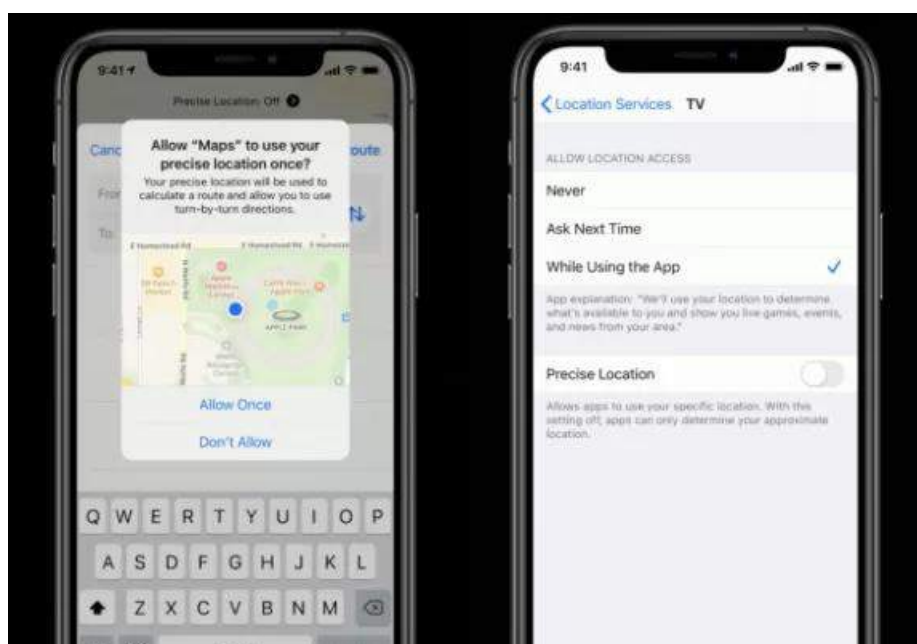
```
/*
 * callbacks occur. That is, it will be called on the runloop where the
 * CLLocationManager was originally initialized.
 */
- (void)requestTemporaryFullAccuracyAuthorizationWithPurposeKey:(NSString *)purposeKey completion:(void (^)(_Nullable NSError *
_Nullable))completion API_AVAILABLE(ios(14.0), macos(10.16), watchos(7.0), tvos(14.0));

/*
 * requestTemporaryFullAccuracyAuthorizationWithPurposeKey:
 *
 * Discussion:
 * This is a variant of requestTemporaryAccurateLocationAuthorizationWithPurposeKey:completion:
 * which doesn't take a completion block. This is equivalent to passing in a nil
 * completion block.
 */
- (void)requestTemporaryFullAccuracyAuthorizationWithPurposeKey:(NSString *)purposeKey API_AVAILABLE(ios(14.0), macos(10.16), watchos(7.0),
tvos(14.0));
```

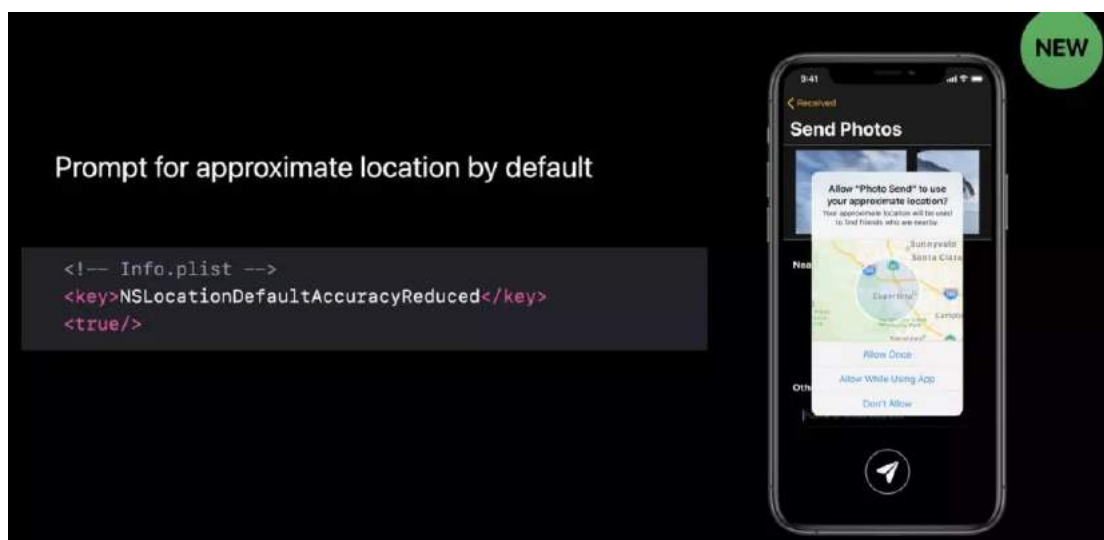
使用方式也很简单，需要首先在 Info.plist 中配置 “NSLocationTemporaryUsageDescriptionDictionary” 字典中需要配置 key 和 value 表明使用位置的原因，以及具体的描述。



在本例中，key 即为获取用户权限时传的 "purposeKey"，最终呈现给用户的就是左图，右图为当 App 主动关闭精确定位权限申请。



对于地理位置不敏感的 App 来说，iOS14 也可以通过直接在 info.plist 中添加 NSLocationDefaultAccuracyReduced 为 true 默认请求大概位置。



这样设置之后，即使用户想要为该 App 开启精确定位权限，也无法开启。



也可以直接通过 API 来根据不同的需求设置不同的定位精确度。



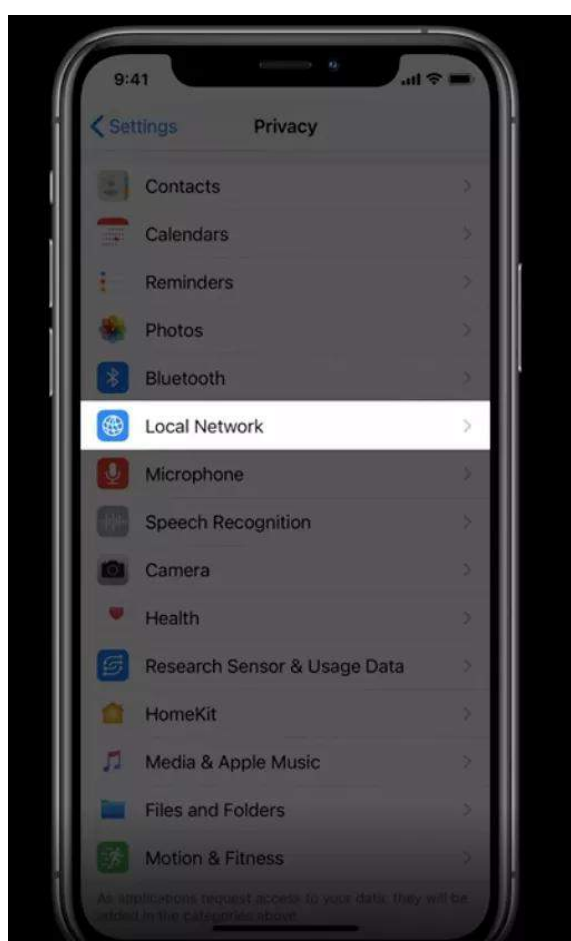
需要注意的是，当 App 在 Background 模式下，如果并未获得精确位置授权，那么 Beacon 及其他位置敏感功能都将受到限制。

Local Network

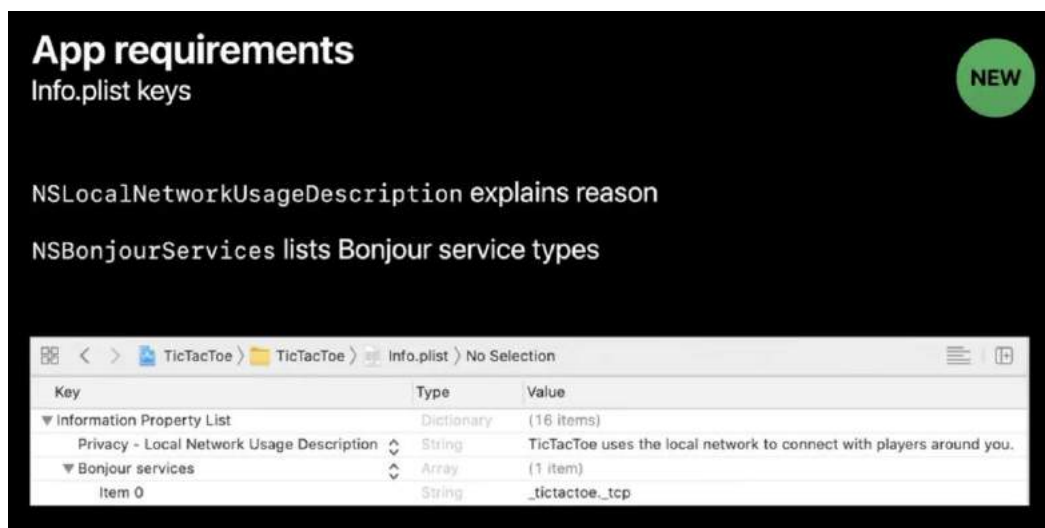
iOS14 当 App 要使用 Bonjour 服务时或者访问本地局域网，使用 mDNS 服务等，都需要授权，开发者需要在 Info.plist 中详细描述使用的为哪种服务以及用途。下图为需要无需申请权限与需要授权的服务：

 Unrestricted	 Requires Permission
TCP and UDP to Internet Hosts	TCP and UDP to Local Network Hosts
AirPrint, AirPlay, AirDrop	Bonjour Browse and Advertise
HomeKit	IP Multicast and Broadcast, ICMP

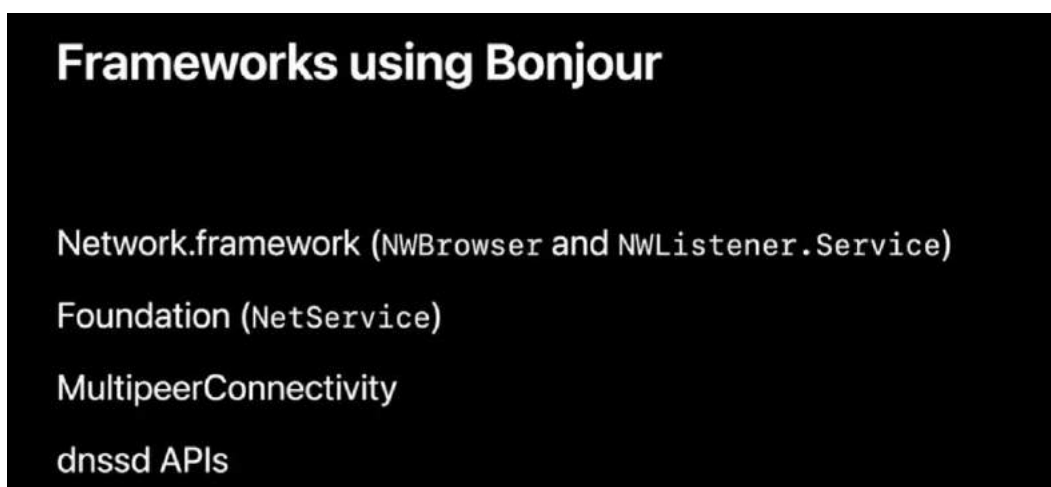
在 "隐私设置" 中也可以查看和修改具体有哪些 App 正在使用 LocalNetwork



如果应用中需要使用 LocalNetwork 需要在 Info.plist 中配置两个选项, 详细描述为什么需要使用该权限, 以及需要列出具体的使用 LocalNetwork 的服务列表。



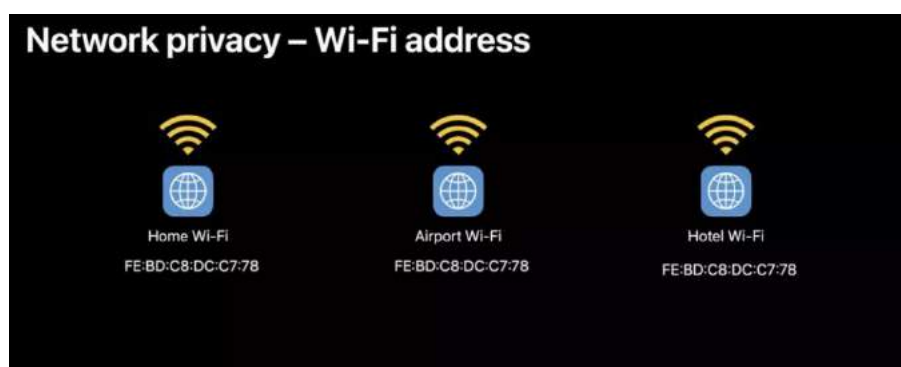
对于使用了下列包含 Bonjour 的 framework, 都需要更新描述.



Wi-Fi Address

iOS8 - iOS13, 用户在不同的网络间切换和接入时, mac 地址都不会改变, 这也就使得网络运营商还是可以通过 mac 地址对用户进行匹配和用户信息收集, 生成完整的用户信息。iOS14 提供 Wifi 加密服务, 每次接入不同的 WiFi 使用的 mac 地址都不同。每过 24 小时, mac 地址还会更新一次。需要关注是否有使用用户网络 mac 地址的服务。

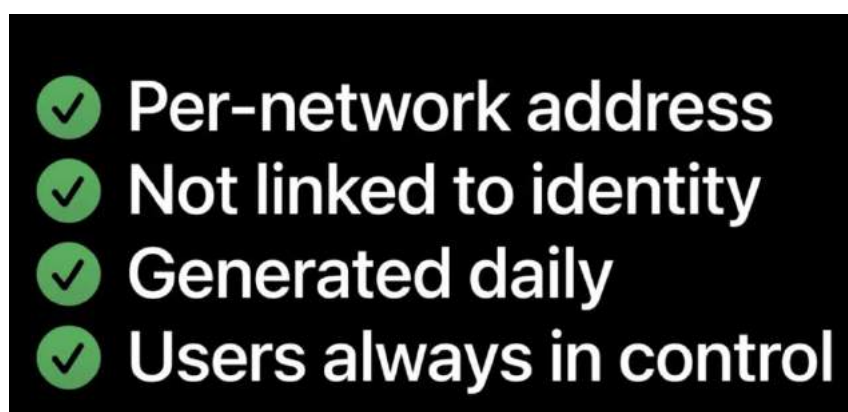
下图为 iOS13 及之前用户接入网络时 mac 地址并不会进行改变



下图为 iOS14 用户接入 Wi-Fi 时 mac 地址的变化情况



并且用户也可以自行选择是否开启 private Wi-Fi address



剪切板

在 iOS14 中，读取用户剪切板的数据会弹出提示。



弹出提示的原因是使用 UIPasteboard 访问用户数据，访问以下数据都会弹出 toast 提示。

```
5
7 @property(nullable, nonatomic, copy) NSString *string API_UNAVAILABLE(tvos) API_UNAVAILABLE(watchos);
8 @property(nullable, nonatomic, copy) NSArray<NSString *> *strings API_UNAVAILABLE(tvos) API_UNAVAILABLE(watchos);
9
10 @property(nullable, nonatomic, copy) NSURL *URL API_UNAVAILABLE(tvos) API_UNAVAILABLE(watchos);
11 @property(nullable, nonatomic, copy) NSArray<NSURL *> *URLs API_UNAVAILABLE(tvos) API_UNAVAILABLE(watchos);
12
13 @property(nullable, nonatomic, copy) UIImage *image API_UNAVAILABLE(tvos) API_UNAVAILABLE(watchos);
14 @property(nullable, nonatomic, copy) NSArray<UIImage *> *images API_UNAVAILABLE(tvos) API_UNAVAILABLE(watchos);
15
16 @property(nullable, nonatomic, copy) UIColor *color API_UNAVAILABLE(tvos) API_UNAVAILABLE(watchos);
17 @property(nullable, nonatomic, copy) NSArray<UIColor *> *colors API_UNAVAILABLE(tvos) API_UNAVAILABLE(watchos);
18
19 // Queries
```

兼容方案：如果应用访问剪切板仅仅用于判断是否为 URL 格式，则 iOS14 新增了两个 API 可以用于规避该提示。如果应用想直接访问剪切板的数据，暂时可能无法做到规避该提示。iOS14 新增两种 UIPasteboardDetectionPattern。

```
typedef NSString * UIPasteboardDetectionPattern NS_TYPED_ENUM API_AVAILABLE(ios(14.0));

/// NSString value, suitable for implementing "Paste and Go"
UIKIT_EXTERN UIPasteboardDetectionPattern const UIPasteboardDetectionPatternProbableWebURL API_AVAILABLE(ios(14.0));

/// NSString value, suitable for implementing "Paste and Search"
UIKIT_EXTERN UIPasteboardDetectionPattern const UIPasteboardDetectionPatternProbableWebSearch API_AVAILABLE(ios(14.0));
```

上面的两个 API 可用于规避提示，但只能用于判断剪切板中是否有 URL，并不是真正的访问剪贴板数据，也拿不到剪切板的真实数据。下面两个 API 可以获得具体的 URL 信息，但是会触发剪切板提示。并且实测当用户剪切板中包含多个 URL 时只会返回第一个。

```

// Detection
/// Detects patterns in the first pasteboard item.
///
/// @param patterns Detect only these patterns.
/// @param completionHandler Receives which patterns were detected, or an error.
- (void)detectPatternsForPatterns:(NSSet<UIPasteboardDetectionPattern> *)patterns
    completionHandler:(void (^)(NSSet<UIPasteboardDetectionPattern> * _Nullable,
                                NSError * _Nullable))completionHandler NS_REFINED_FOR_SWIFT API_AVAILABLE(ios(14.0));

/// Detects patterns in the specified pasteboard items.
///
/// @param patterns Detect only these patterns.
/// @param itemSet Specifies which pasteboard items by their position. Nil means all items.
/// @param completionHandler Receives which patterns were detected per item specified,
/// or an error.
- (void)detectPatternsForPatterns:(NSSet<UIPasteboardDetectionPattern> *)patterns
    inItemSet:(NSIndexSet * _Nullable)itemSet
    completionHandler:(void (^)(NSArray<NSSet<UIPasteboardDetectionPattern> * > * _Nullable,
                                NSError * _Nullable))completionHandler NS_REFINED_FOR_SWIFT API_AVAILABLE(ios(14.0));

/// Detects patterns and corresponding values in the first pasteboard item.
///
/// @param patterns Detect only these patterns.
/// @param completionHandler Receives which patterns and values were detected, or an error.
- (void)detectValuesForPatterns:(NSSet<UIPasteboardDetectionPattern> *)patterns
    completionHandler:(void (^)(NSDictionary<UIPasteboardDetectionPattern, id> * _Nullable,
                                NSError * _Nullable))completionHandler NS_REFINED_FOR_SWIFT API_AVAILABLE(ios(14.0));

/// Detects patterns and corresponding values in the specified pasteboard items.
///
/// @param patterns Detect only these patterns.
/// @param itemSet Specifies which pasteboard items by their position. Nil means all items.
/// @param completionHandler Receives which patterns and values were detected per item specified,
/// or an error.
- (void)detectValuesForPatterns:(NSSet<UIPasteboardDetectionPattern> *)patterns
    inItemSet:(NSIndexSet * _Nullable)itemSet
    completionHandler:(void (^)(NSArray<NSDictionary<UIPasteboardDetectionPattern, id> * > * _Nullable,
                                NSError * _Nullable))completionHandler NS_REFINED_FOR_SWIFT API_AVAILABLE(ios(14.0));

```

使用示例

```

NSSet *patterns = [[NSSet alloc] initWithObjects:UIPasteboardDetectionPatternProbableWebURL,
nil];

[[UIPasteboard generalPasteboard] detectPatternsForPatterns:patterns
completionHandler:^(NSSet<UIPasteboardDetectionPattern> * _Nullable result, NSError * _Nullable
error) {

    if (result && result.count) {

        // 当前剪切板中存在 URL

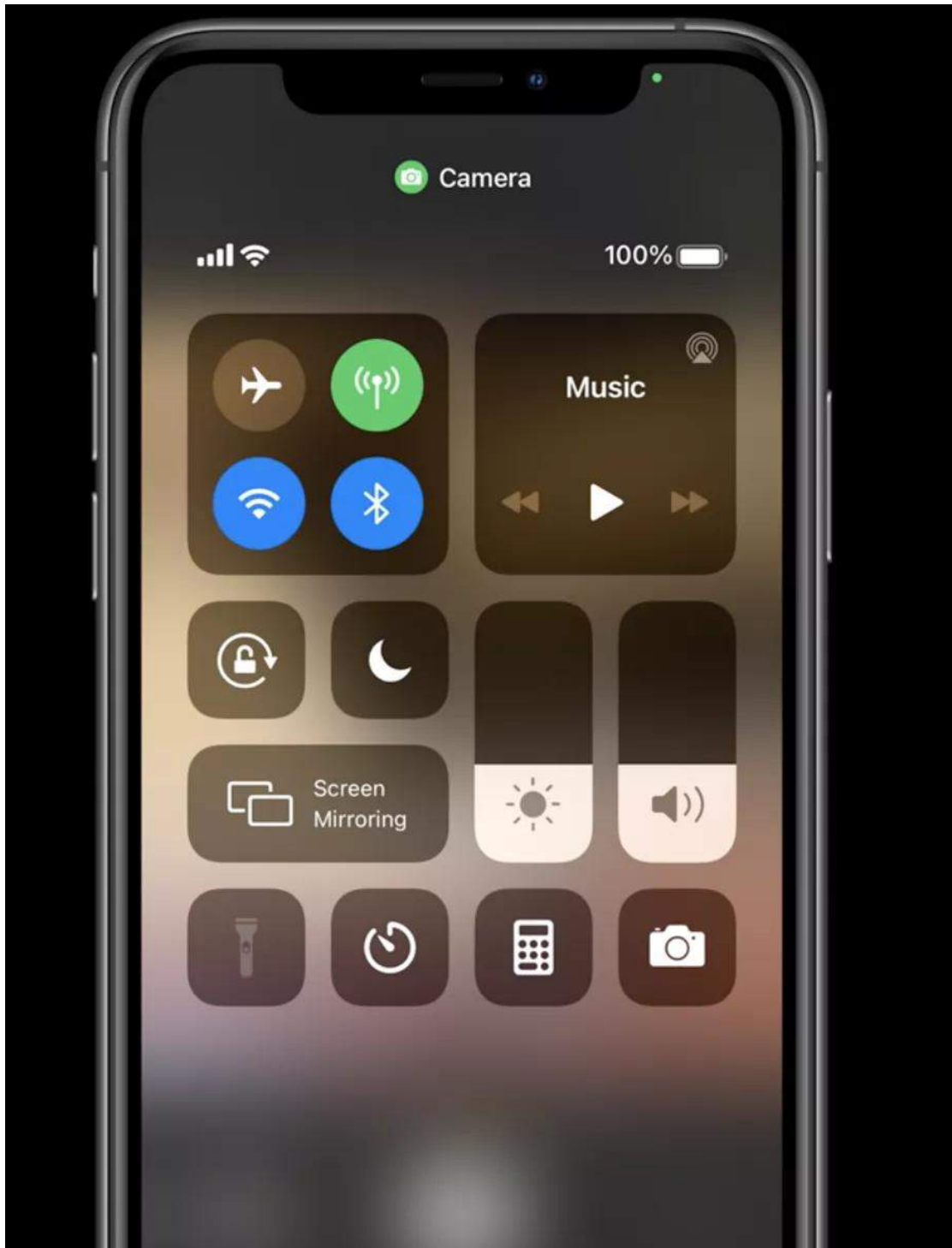
    }

}];

```

■ 相机和麦克风

iOS14 中 App 使用相机和麦克风时会有图标提示以及绿点和黄点提示，并且会显示当前是哪个 App 在使用此功能。我们无法控制是否显示该提示。



会触发录音小黄点的代码示例：

```
AVAudioRecorder *recorder = [[AVAudioRecorder alloc] initWithURL:recorderPath settings:nil  
error:nil];  
  
[recorder record];
```

触发相机小绿点的代码示例:

```
AVCaptureDeviceInput *videoInput = [[AVCaptureDeviceInput alloc]
initWithDevice:videoCaptureDevice error:nil];

AVCaptureSession *session = [[AVCaptureSession alloc] init];

if ([session canAddInput:videoInput]) {

    [session addInput:videoInput];

}

[session startRunning];
```

■ IDFA

IDFA 全称为 Identity for Advertisers，即广告标识符。用来标记用户，目前最广泛的用途是用于投放广告、个性化推荐等。

在 iOS13 及以前，系统会默认为用户开启允许追踪设置，我们可以简单的通过代码来获取到用户的 IDFA 标识符。

```
if ([[ASIdentifierManager sharedManager] isAdvertisingTrackingEnabled]) {

    NSString *idfaString = [[ASIdentifierManager sharedManager]
advertisingIdentifier].UUIDString;

    NSLog(@"%@", idfaString);

}
```

但是在 iOS14 中，这个判断用户是否允许被追踪的方法已经废弃。

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

NS_ASSUME_NONNULL_BEGIN

API_AVAILABLE(ios(6), macos(10.14), tvos(6))
@interface ASIdentifierManager : NSObject

+ (ASIdentifierManager *)sharedManager;

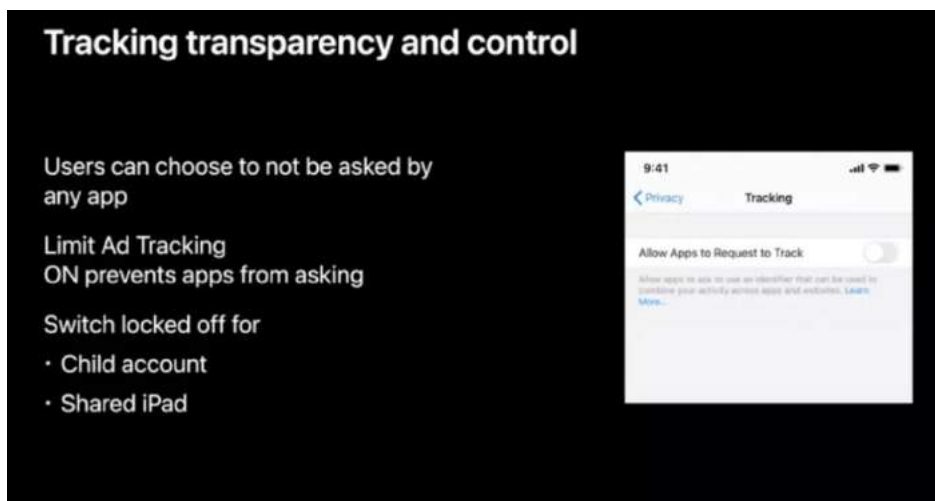
@property (nonatomic, readonly) NSUUID *advertisingIdentifier;
@property (nonatomic, readonly, getter=isAdvertisingTrackingEnabled) BOOL advertisingTrackingEnabled
    API_DEPRECATED("This has been replaced by functionality in AppTrackingTransparency's
    ATTrackingManager class.", ios(6, 14), macos(10.14, 10.16), tvos(6, 14));

- (void)clearAdvertisingIdentifier API_UNAVAILABLE(ios, macos, tvos);

@end

NS_ASSUME_NONNULL_END
```

iOS14 中，系统会默认为用户关闭广告追踪权限。



对于这种情况，我们需要去请求用户权限。首先需要在 Info.plist 中配置 " NSUserTrackingUsageDescription " 及描述文案,接着使用 AppTrackingTransparency 框架中的 ATTrackingManager 中的 requestTrackingAuthorizationWithCompletionHandler 请求用户权限，在用户授权后再去访问 IDFA 才能够获取到正确信息。

```
#import <AppTrackingTransparency/AppTrackingTransparency.h>

#import <AdSupport/AdSupport.h>

- (void)testIDFA {

    if (@available(iOS 14, *)) {

        [ATTrackingManager
requestTrackingAuthorizationWithCompletionHandler:^(ATTrackingManagerAuthorizationStatus
status) {

            if (status == ATTrackingManagerAuthorizationStatusAuthorized) {

                NSString *idfaString = [[ASIdentifierManager sharedManager]
advertisingIdentifier].UUIDString;

                }

            }

        }];

    } else {

        // 使用原方式访问 IDFA
```



```
}  
  
}
```

对于用户拒绝授权 UserTracking 的情况,可以考虑接入苹果的 SKAdNetwork 框架进行广告分析。感兴趣的同学可进一步了解: <https://developer.apple.com/documentation/storekit/skadnetwork>

■ 上传 AppStore

更加严格的隐私审核,可以让用户在下载 App 之前就知道此 App 将会需要哪些权限。目前苹果商店要求所有应用在上架时都必须提供一份隐私政策。如果引入了第三方收集用户信息等 SDK,都需要向苹果说明是这些信息的用途。

App Store transparency

Required to answer

- What data do you collect?
- How is the data used?
- Is the data linked to a particular user or device?
- Do you use this data to track users?



Data Linked to You

The following data may be collected and linked to your accounts, devices, or identity:



Financial Info



Location



Data Linked to You

The following data may be collected and linked to your accounts, devices, or identity:



Financial Info



Location



Contacts



Purchases



Browsing History



Identifiers



Data Used to Track You

The following data may be used to track you across apps and websites owned by other companies:



Contact Info



Location



Identifiers

总结

对于这次 iOS14 的隐私权限大升级和新尝试，体现了苹果对于用户隐私的尊重。

从用户角度来说，近年来越来越精准的广告投放让我们越来越感觉自己被”监视“着，此次升级后，我们有了更多保护自己隐私的方式以及避免广告骚扰的方法，苹果此举无疑会加大我们对其的好感度和信任感。但从另一个角度来说，对于 IDFA 的限制，可能会导致之前许多依靠广告投放收入的免费 App 难以继续维持生计，也可能也会导致免费 App 的数量有所降低。从开发者的角度来说，除了对 iOS14 隐私升级的积极适配外，也让我们感受到了 iOS14 中对于用户隐私的重视无疑会提高获取用户行为信息的成本。

冲击最大的应该就是广告行业，对于目前的推荐算法和用户拉新都会受到影响，如何在充分尊重用户隐私的前提下进行广告的精准投放对于开发者和广告商来说都是一个不小的机遇和挑战。

参考

- [WWDC 2020 Apple Developer](#)
- [Developer Documentation](#)

Metal 新特性：大幅度提升 iOS 端性能

作者 | 陈昱(岑彧)

出品 | 阿里巴巴新零售淘系技术

作为较早在客户端侧选择 Flutter 方案的技术团队，性能和用户体验一直是闲鱼技术团队在开发中比较关注的点。而 Metal 这样的直接操作 GPU 的底层接口无疑会给闲鱼技术团队突破性能瓶颈提供一些新的思路。

本文将详细阐述一下这次大会 Metal 相关的新特性，以及对于闲鱼技术和整个淘系技术来说，这些新特性带来了哪些技术启发与思考。

前言

Metal 是一个和 OpenGL ES 类似的面向底层的图形编程接口，通过使用相关的 api 可以直接操作 GPU，最早在 2014 年的 WWDC 的时候发布。Metal 是 iOS 平台独有的，意味着它不能像 OpenGL ES 那样支持跨平台，但是它能最大的挖掘苹果移动设备的 GPU 能力，进行复杂的运算，像 Unity 等游戏引擎都通过 Metal 对 3D 能力进行了优化，App Store 还有相应的运用 Metal 技术的游戏专题。

闲鱼团队是比较早在客户端侧选择 Flutter 方案的技术团队，当前的闲鱼工程里也是一个较为复杂的 Native-Flutter 混合工程。作为一个 2C 的应用，性能和用户体验一直是闲鱼技术团队在开发中比较关注的点。而 Metal 这样的直接操作 GPU 的底层接口无疑会给闲鱼技术团队突破性能瓶颈提供一些新的思路。

下面会详细阐述一下这次大会 Metal 相关的新特性，以及对于闲鱼技术和整个淘系技术来说，这些新特性带来了哪些技术启发与思考。

Metal 相关新特性

■ Harness Apple GPUs with Metal

这一章其实主要介绍的是 Apple GPU 的在图形渲染上的原理和工作流，是一些比较底层的硬件原理。当我们使用 Metal 进行 App 或者是游戏的构建的时候，Metal 会利用 GPU 的 tile-based deferred rendering (TBDR)架构给应用和游戏带来非常可观的性能提升。这一章主要就是介绍 GPU 的架构和能力，以及 TBDR 架构进行图像渲染的原理和流程。总之就是号召开发者们使用 Metal 来构建应用和游戏。因为这个 session 没有涉及到上层的软件开发，就不对视频的具体内容进行赘述了。详情可见：[Harness Apple GPUs with Metal](#)

■ Optimize Metal apps and games with GPU counters

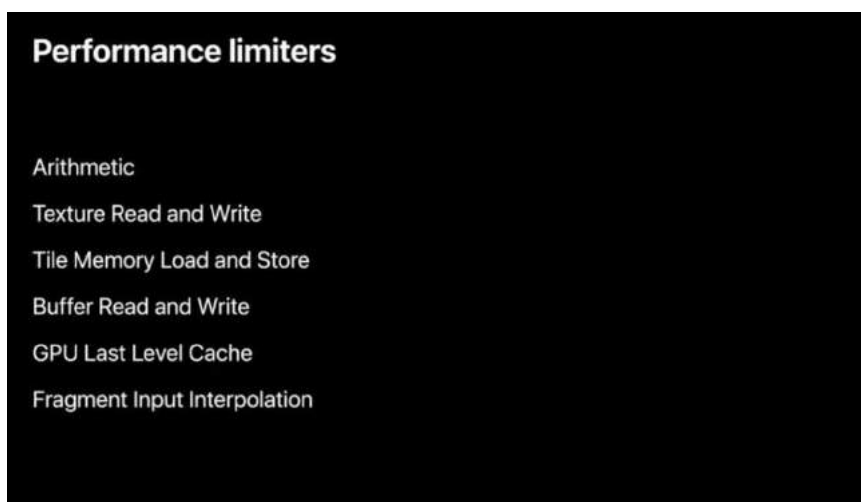
这一章主要介绍了 Xcode 中的 GPU 性能分析工具 Instrument，这个工具现在已经支持了 GPU 的性能分析。然后从多个方面分析了 GPU 的性能瓶颈，以及性能瓶颈出现时的优化点。总体来说就是通过性能分析工具来优化我们的 App 或者游戏，让整个画面更加流畅。整个章节主要分为五个部分：

总体介绍

这个环节主要是快速回顾了一下 Apple 的 GPU 的架构和渲染流程。然后因为很多渲染任务都需要在不同的硬件单元上进行，例如 ALU 和 TPU。他们对不同的吞吐量有着不同的度量。有很多 GPU 的性能指标需要被考虑，所以推出了 GPU 性能计数器。这个计数器可能测量到 GPU 的利用率，过高和过低都会造成我们的渲染性能瓶颈。关于计数器的具体使用，参考官方的 video 效果会更好：[Optimize Metal apps and games with GPU counters](#) (6: 37~9: 57)，主要使用了 Instrument 工具，关于工具的全面详细的使用可以参考 WWDC19 的 session video[Getting Started with Instruments](#)

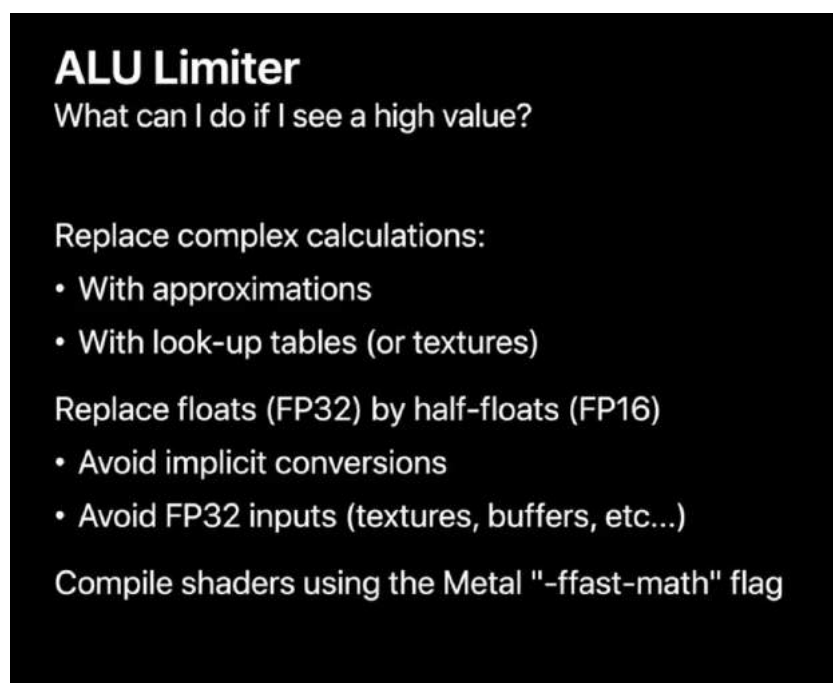
性能瓶颈分析

这一章主要介绍了造成 GPU 性能瓶颈的各个方面以及它们的优化点。主要分为六个方面，如下图所示：



1. Arithmetic (运算能力)

GPU 中通常通过 ALU (Arithmetic Logic Unit) 来处理各种运算，例如位操作，关系操作等。他是着色器核心的一部分。在这里一些复杂的操作或者是高精度的浮点运算都会造成一些性能瓶颈，所以给出以下建议来进行优化：

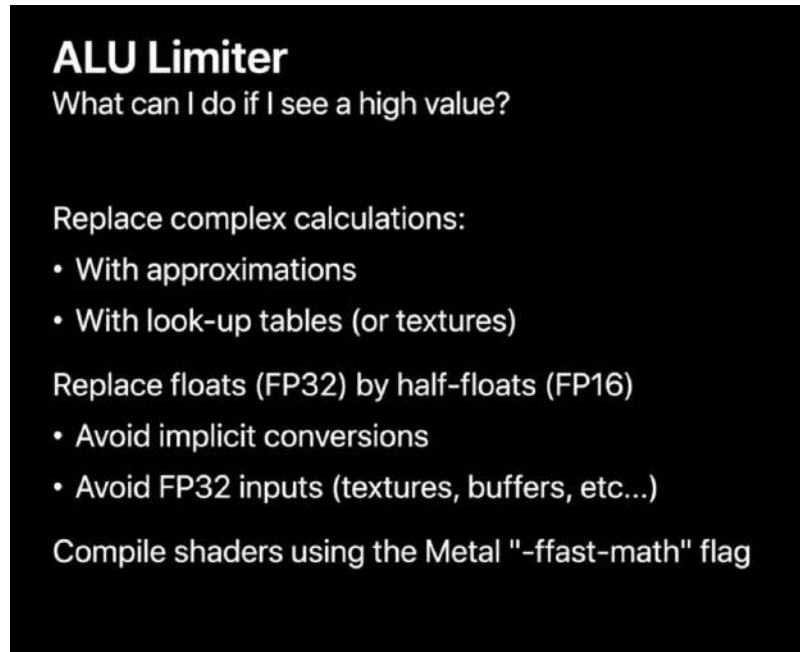


如上图所示，我们可以使用近似或者是查找表的方式来替换复杂的运算。此外，我们可以将全精度的浮点数替换为半精度的浮点数。尽量避免隐式转换，避免 32 位浮点数的输入。以及确保所有的着色器都使用 Metal 的 “-ffast-math” 来进行编译。

2. Texture Read and Write

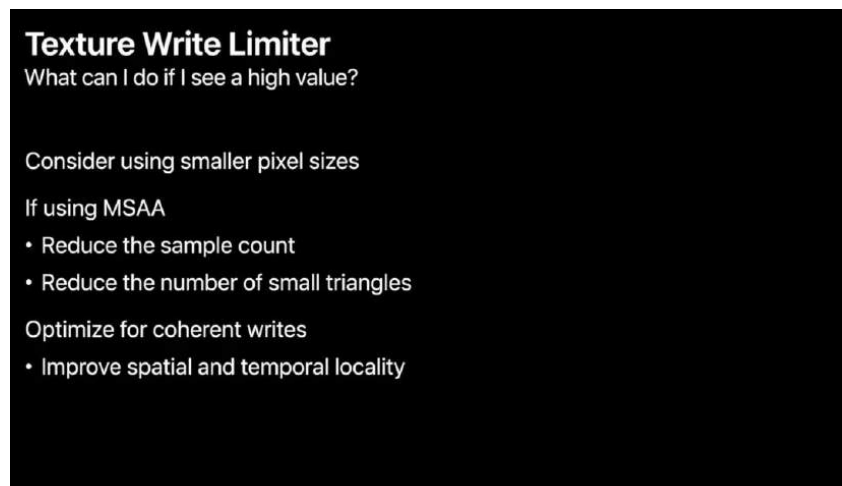
GPU 通过 Texture Processing Unit 来处理纹理的读写操作。当然在读写的过程中也会遇到一些性能瓶颈问题。这里从读和写两个部分分别来给出优化点：

- Read



如上图所示，我们可以尝试使用 mipmaps。此外，可以考虑更改过滤选项。例如，使用双线性代替三线性，降低像素大小。确保使用了纹理压缩，对 Asset 使用块压缩（如 ASTC），对运行时生成的纹理使用无损纹理压缩。

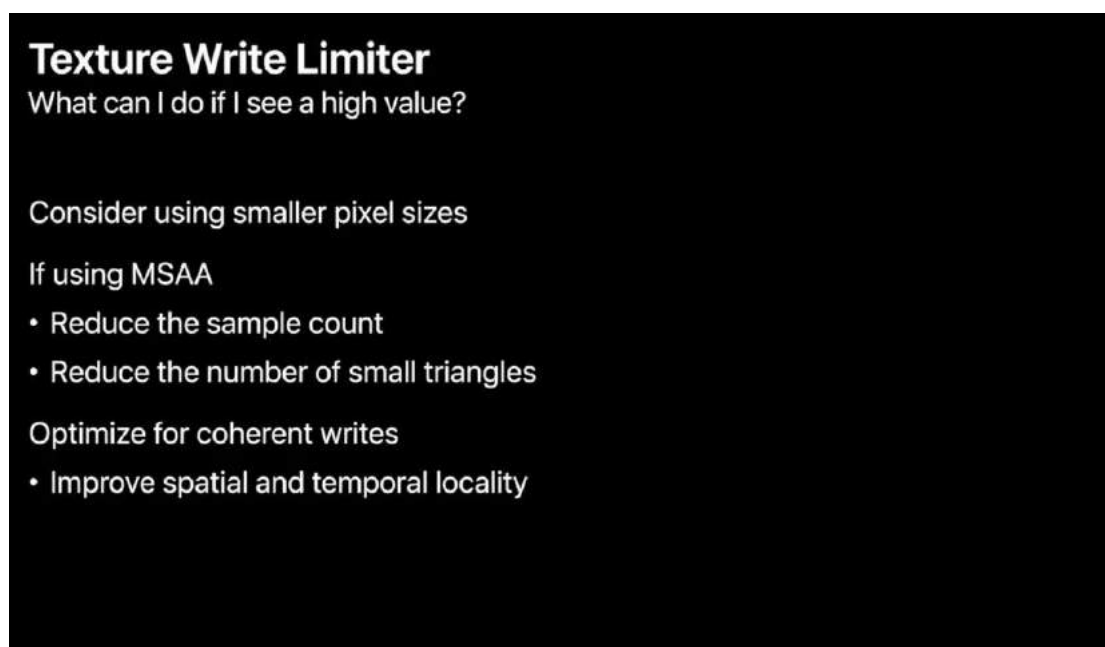
- Write



如上图所示，我们应该注意到像素的大小，以及每个像素中唯一 MSAA 样本的数量。此外，可以尝试一些优化一些逻辑写法。

Tile Memory Load and Store

图块内存是一组存储 Thread Group 和 ImageBlock 数据的高性能内存。当从 ImageBlock 或是 Threadgroup 读取或写入像素数据时，比如在使用 Tile 着色器时或者是计算分派时，可以访问到 Tile 内存。那当使用 GPU 性能计数器发现这个方面的性能瓶颈时，我们可以如下图所示进行优化。



考虑减少 threadgroup 的并行，或者是 SIMD/Quadgroup 操作。此外，确保将线程组的内存分配和访问对齐到 16 字节。最后，可以考虑重新排序内存访问模式。

Buffer Read and Write

在 Metal 中，缓冲区只被着色器核心访问。在这个地方发现了性能瓶颈。我们可以如下图所示进行优化：

Texture Write Limiter

What can I do if I see a high value?

Consider using smaller pixel sizes

If using MSAA

- Reduce the sample count
- Reduce the number of small triangles

Optimize for coherent writes

- Improve spatial and temporal locality

可以更大力度的压缩打包数据，例如使用例如 `packed_half3` 这样小的类型。此外，可以尝试向量化加载和存储。例如使用 SIMD 类型。避免寄存器溢出，以及可以使用纹理来平衡工作负载。

GPU Last Level Cache

如果在这个方面，我们的 GPU 性能计数器显示一个过高的值。我们可以如下图这样优化：

GPU Last Level Cache Limiter

What can I do if I see a high value?

If texture or buffer limiters show a high value

- Optimize those first
- Reduce size of working sets

If shaders use **device atomics**

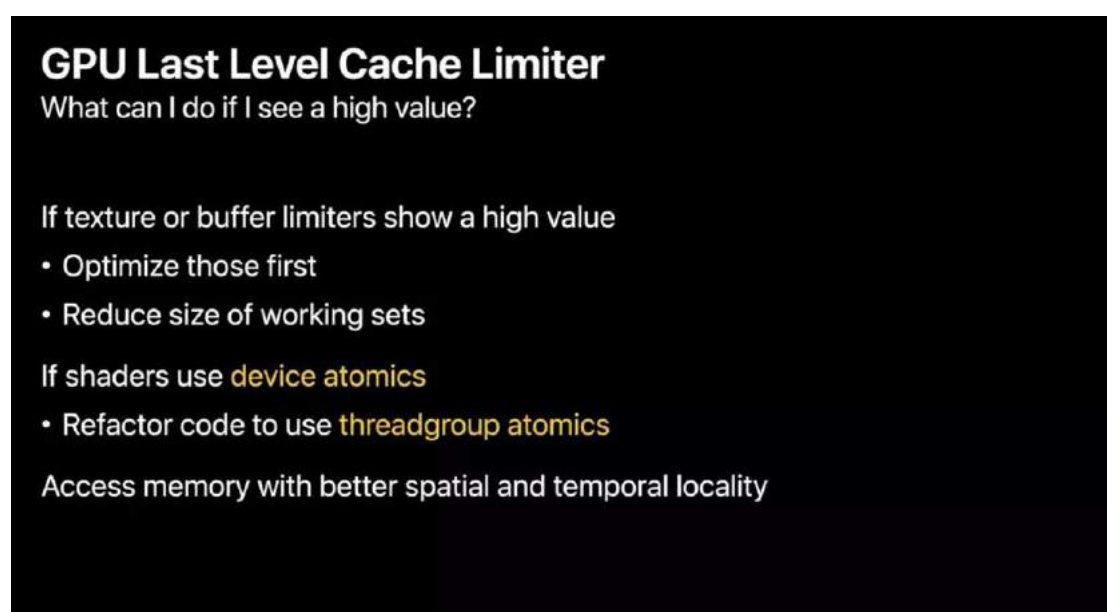
- Refactor code to use **threadgroup atomics**

Access memory with better spatial and temporal locality

如果纹理或者是缓存区也同样显示一个过高的值，我们可以把这个优化放到第一优先级。我们可以考虑减小工作集的大小。如果 Shader 正在使用 Device Atomics，我们可以尝试重构我们的代码来使用 Threadgroup Atomics。

Fragment Input Interpolation

分段输入插值。分段输入在渲染阶段由着色器核心进行插值。着色器核心有一个专用的分段输入插值器。这个是比较固定和高精度的功能。我们能优化的点不多，如下图所示：



尽可能的移除传递给分段着色器的顶点属性。

内存带宽

内存带宽也是影响我们 GPU 性能的一个重要因素。如果在 GPU 性能计数器的内存带宽模块看到一个很高的值。我们就应该如下图所示来进行优化：

Memory Bandwidth GPU counter

What can I do if I see a high value?

If texture or buffer limiters show a high value

- Optimize those first
- Reduce size of working sets

Only load data needed by the current render pass

Only store data needed by future render passes

Leverage texture compression

- Block-compression (**ASTC**) for assets
- Lossless compression for textures generated at run-time

如果纹理和缓存区也同样显示比较高的值，那优化优先级应该排到第一位。优化方案也是较少 Working Set 的大小。此外，我们应该只加载当前渲染过程需要的数据，只存储未来渲染过程需要的数据。然后就是确保使用纹理压缩。

Occupancy

Occupancy GPU counter

What can I do if I see a low value?

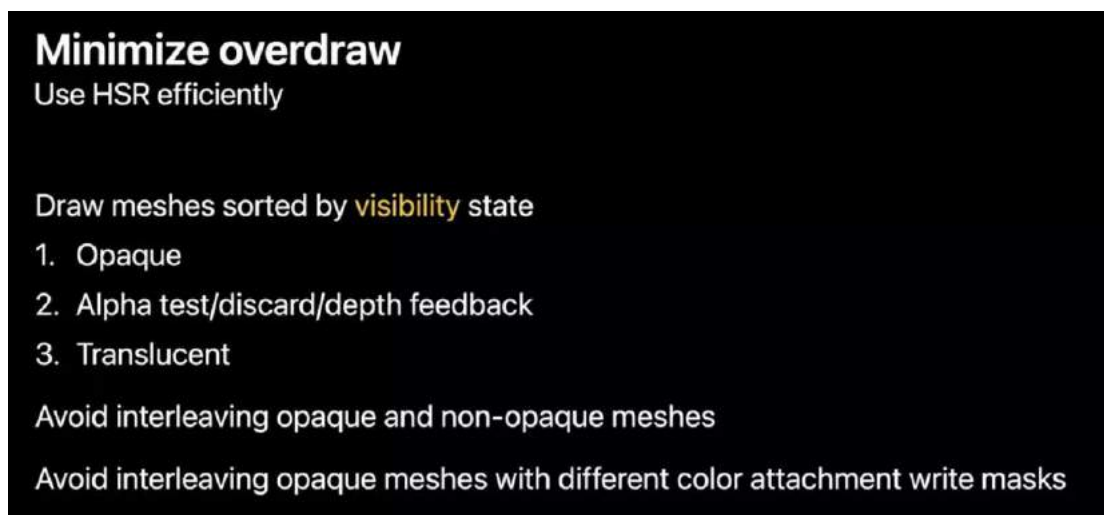
Correlate occupancy measurements with data from other counters or tools

When overall occupancy is low

- Shaders have exhausted some internal resources
- Threads finish executing faster than the GPU can create new ones
- Your app is rendering to a small area, or dispatching very small compute grids, such that the GPU runs out of threads to create

如果我们看到整体利用率比较低，这意味着 Shader 可能已经耗尽了一些内部资源，比如 tile 或者 threadgroup 内存。也可能是线程完成执行的速度比 GPU 创建新线程的速度快。

避免重复绘制



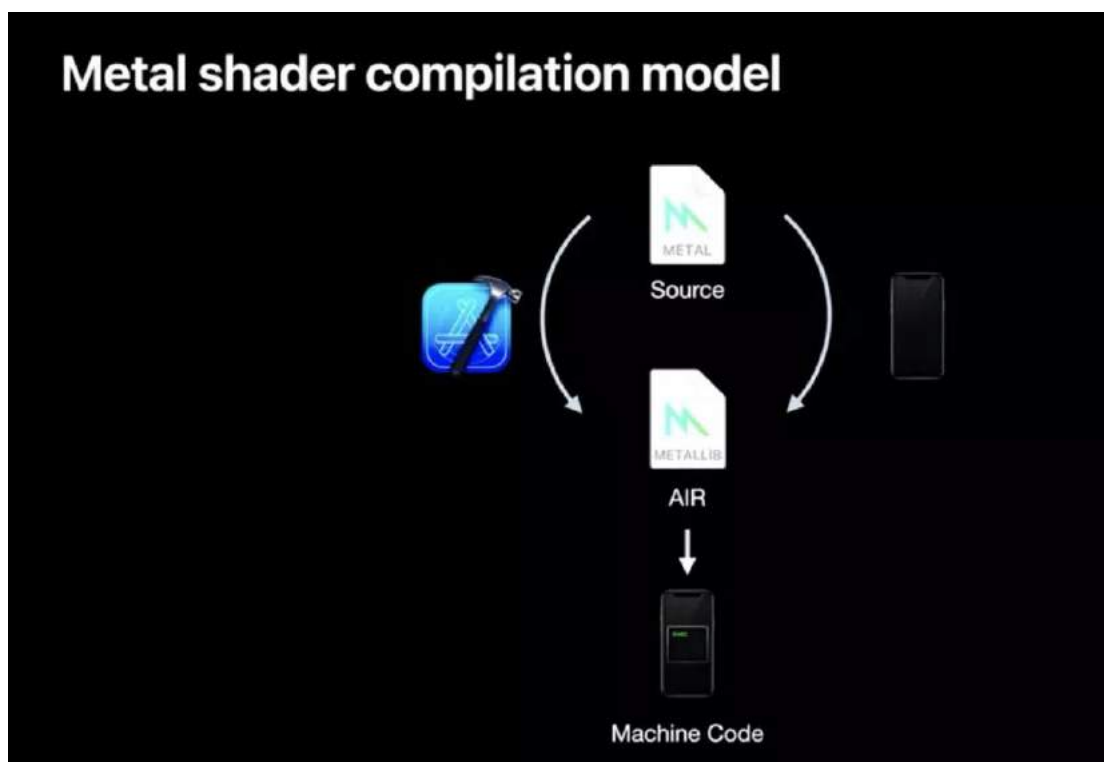
我们通过 GPU 计数器可以统计到重复绘制的区域，我们应该高效使用 HSR 来避免这样的重绘。我们可以如图所示的顺序来进行绘制。

Build GPU binaries with Metal

这一章主要给开发者们介绍了一种使用 Metal 的编程工作流，可以通过优化 Metal 的渲染编译模型来增强渲染管线，这个优化可以在应用程序启动，特别是首次启动时大大减少 PSO(管线状态对象)的加载时间。可以让我们的图形渲染更加的高效。整个章节主要分为四个部分：

Metal 的 Shader 编译模型概述

众所周知, Metal Shading Language 是 Apple 为开发者提供的 Shader 编程语言, Metal 会将编程语言编译成为一个叫做 AIR 的中间产物, 然后 AIR 会在设备上进一步编译, 生成每个 GPU 所需的特定的机器码。整个过程如下图所示：



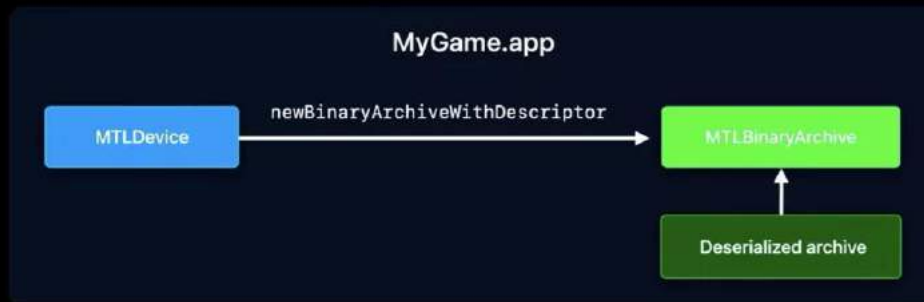
上述过程在每个管线的生命周期中都会发生，当前 Apple 为了加速管线的重新编译和重新创建流程，会缓存一些 Metal 的方法变体，但是这个过程还是会造成屏幕的加载耗时长。而且在当前的这个编译模型中，应用程序不能在不同的 PSO（管线状态对象）中重用之前生成的机器码子程序。

所以我们需要一种方法来减少这个整个管线编译（即源代码→AIR→GPU 二进制代码）的时间成本，还需要一种机制来支持不同 PSO 之间共享子程序和方法，这样就不需要将相同的代码多次编译或者是多次加载到内存中。这样开发者们就可以使用这套工具来优化 App 首次的启动体验。

■ Metal 二进制文件介绍

Metal 二进制文件就是解决上述需求的方法之一，现在开发者们可以直接使用 Metal 为二进制文件来控制 PSO 的缓存。开发者可以收集已编译的 PSO，然后将它们存储到设备中，甚至可以分发到其他兼容的设备中（同样的 GPU 和同样的操作系统），这种二进制文件可以看做一种 Asset。下面是一些例程和示意图：

Creating a Binary Archive



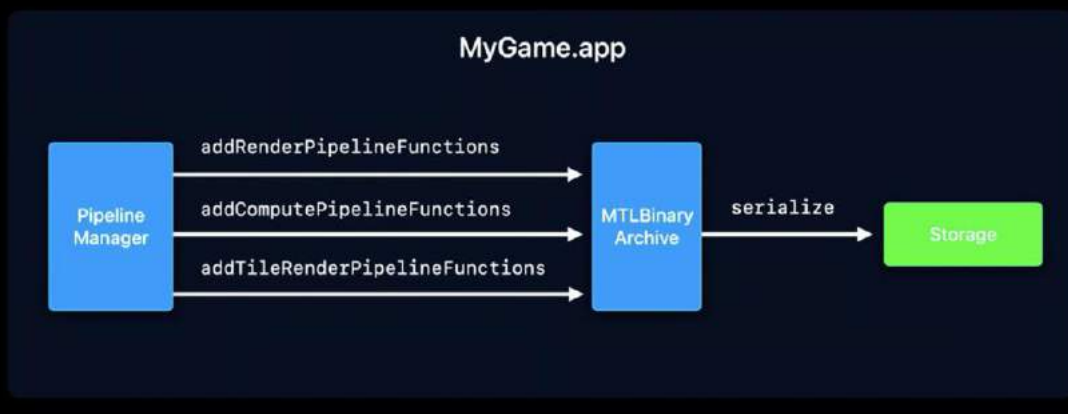
```
//创建一个空的二进制文件
```

```
let descriptor = MTLBinaryArchiveDescriptor()
```

```
descriptor.url = nil
```

```
let binaryArchive = try device.makeBinaryArchive(descriptor:descriptor)
```

Pipeline collection and harvesting

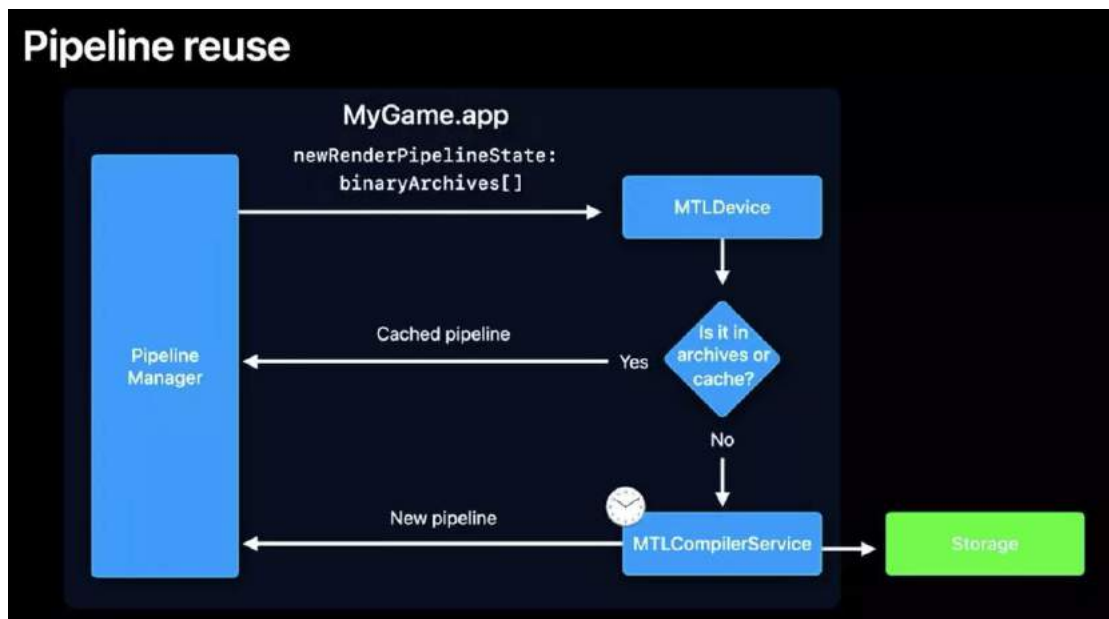


```

//Populating an archive
// Render pipelines
    try binaryArchive.addRenderPipelineFunctions(with: renderPipelineDescriptor)
// Compute pipelines
    try binaryArchive.addComputePipelineFunctions(with: computePipelineDescriptor)
// Tile render pipelines
    try binaryArchive.addTileRenderPipelineFunctions(with: tileRenderPipelineDescriptor)

```

Pipeline reuse



// 重用已编译的方法

// Reusing compiled functions to build a pipeline state object from a file

```

let renderPipelineDescriptor = MTLRenderPipelineDescriptor()
// ...
renderPipelineDescriptor.binaryArchives = [ binaryArchive ]

let renderPipeline = try device.makeRenderPipelineState(descriptor:
renderPipelineDescriptor)

```

//序列化

```

let documentsURL = FileManager.default.urls(for: .documentDirectory,
in: .userDomainMask).first!
let archiveURL = documentsURL.appendingPathComponent("binaryArchive.metallib")
try binaryArchive.serialize(to: NSURL.fileURL(withPath: archiveURL))

```

```
//反序列化
let documentsURL =
FileManager.default.urls(for: .documentDirectory,in: .userDomainMask).first!
    let serializeURL = documentsURL.appendingPathComponent("binaryArchive.metallib")
let descriptor = MTLBinaryArchiveDescriptor()
    descriptor.url = NSURL.fileURL(withPath: serializeURL)
    let binaryArchive = try device.makeBinaryArchive(descriptor: descriptor)
```

总的来说就是这个 Metal 二进制文件可以提供开发者手动管理管线缓存的方法，这样就可以从一个设备中获取这些文件并部署到其他兼容的设备上，在 iOS 环境下，极大地减少了第一次安装游戏或应用以及设备重启后的管道创建时间。可以优化应用的首次启动体验和冷启动体验。

■ Metal 对动态库的支持

动态库将允许开发者编写可重用的库代码，却可以减少重新编译程序的时间和内存成本，这个特性将会允许开发者将计算着色器和程序库动态链接。而且和二进制文件一样，动态库也是可序列化和可转移的。这也是解决上述需求的方案之一。

在 PSO 生成的时候，每个应用程序都需要为程序 library 生成机器码，而且使用相同的程序库编译多个管线会导致生成重复的机器码。由于大量的编译和内存的增加，这个可能会导致更长的管线加载时间。而动态库就可以解决这个问题。

Metal Dynamic Library 允许开发者以机器码的形式动态链接,加载和共享工具方法。代码可以在多个计算管线中重用，消除了重复编译和多个相同子程序的存储。而且这个 MTLDynamicLibrary 是可序列化的，可以作为应用程序的 Asset 使用。MTLDynamicLibrary 其实就是多个计算管线调用的导出方法的集合。

大致的工作流程如下：我们首先创建一个 MTLLibrary 作为我们指定的动态库，这个可以将我们的 metal 代码编译为 AIR。然后我们调用方法 makeDynamicLibrary，这个方法需要指定一个唯一的 installname，在管线创建时，linker 将会使用这个名字来加载动态库。这个方法可以将我们的动态库编译成为机器码。这就完成了动态库的创建。对于动态库的使用来说：通过设置 MTLCompileOptions 里的 libraries 参数，就可以完成动态库的加载和使用了。代码如下：

```
//使用动态库进行编译
let options = MTLCompileOptions()
options.libraries = [ utilityDylib ]
let library = try device.makeLibrary(source: kernelStr, options: options)
```

开发工具介绍

这个部分主要介绍了构建 Metal 二进制文件和构建动态库的具体工具和方法。以视频的形式可能会更好的表现，详情可见：[Build GPU binaries with Metal](#)（从 22: 51 开始）

Debug GPU-side errors in Metal

这一章主要介绍的是 GPU 侧的 bug，当前如果我们的应用程序出现了 GPU 侧的 bug，他的错误日志常常都不能让开发者很直观的定位到错误的代码范围和调用栈。所以在最新的 Xcode 中，增强了关于 GPU 侧的 debug 机制。可以像在代码侧发成的错误一样不但能定位到错误原因，还有错误的调用堆栈和各种信息都可以详细的查看到。让开发者能更好的修复代码造成的 GPU 侧的渲染错误。

Enhanced Command Buffer Errors

Current state of errors reporting

GPU Errors

```
Discarded (victim of GPU error/recovery) (IOAF code 5)
```

API Errors

```
-[MTLDebugComputeCommandEncoder setBuffer:offset:atIndex:]
'offset(200) must be < [buffer length](100).'
```

* frame #6:MPSRayTracing-[AAPLRenderer drawInMTKView:] at AAPLRenderer.mm:410:5

这是当前的错误日志上报, 我们可以看到 GPU 侧的错误日志不像 Api 的错误日志一样可以让开发者很快的定位到错误原因和错误的代码位置。

New Metal debugging tools		Enhanced command buffer errors and shader validation	
	Metal API		Metal shader code
Detect	✓		✓
Locate	✓		✓
Classify	✓		✓
Fix	😊		😊

而最新的 Metal debugging 工具就增强了这方面的能力, 让 Shader 的 code 也可以像 Api 代码一样提供错误定位和分类能力。

我们通过以下代码便可以启用增强版的 commandbuffer 错误机制

```
// 启用增强版的 commandbuffer 错误机制
let desc = MTLCommandBufferDescriptor()
desc.errorOptions = .encoderExecutionStatus
let commandBuffer = commandQueue.makeCommandBuffer(descriptor: desc)
```

错误一共有五种状态:

Enhanced command buffer errors

What it can tell you

Possible error states

```
public enum MTLCommandEncoderErrorState {  
    completed  
    pending  
    faulted  
    affected  
    unknown  
}
```

我们也可以通过以下代码来打印 error:

```
//打印 commandbuffer 的错误  
if let error = commandBuffer.error as NSError? {  
  
    if let encoderInfos =  
        error.userInfo[MTLCommandBufferEncoderInfoErrorKey]  
        as? [MTLCommandBufferEncoderInfo] {  
  
        for info in encoderInfos {  
            print(info.label + info.debugSignposts.joined())  
            if info.errorState == .faulted {  
                print(info.label + " faulted!")  
            }  
        }  
    }  
}
```

开发者可以在开发时和测试时启用优化版的错误机制

Enhanced command buffer errors

When you should use it

Always on

- Development
- QA

■ Shader Validation

Shader validation

What it does

Catches Metal shader code issues

- Prevents & logs a GPU backtrace

Detect



Locate



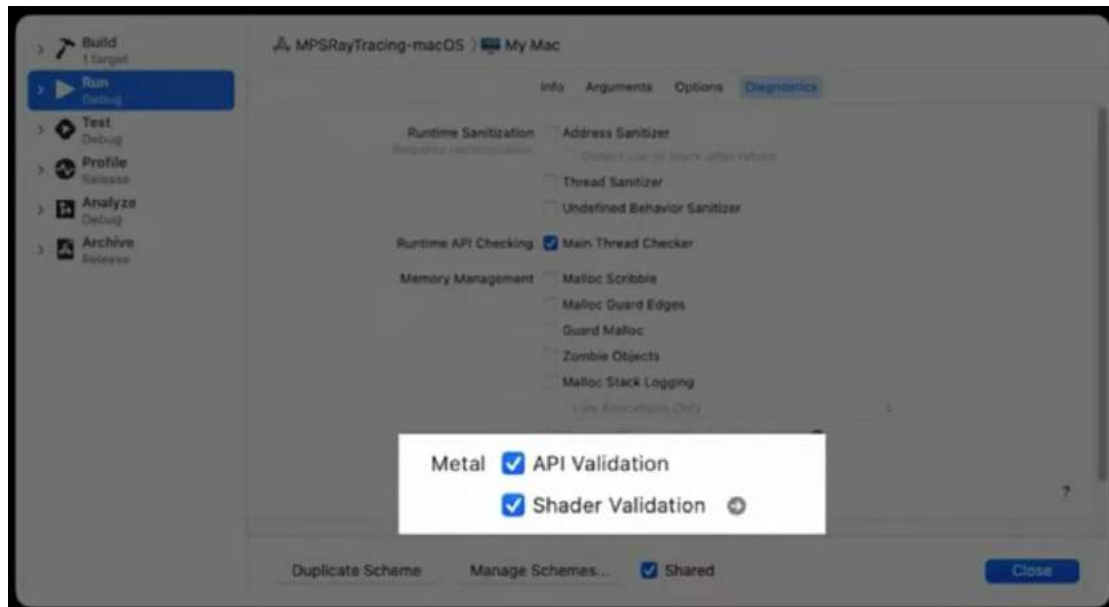
Classify



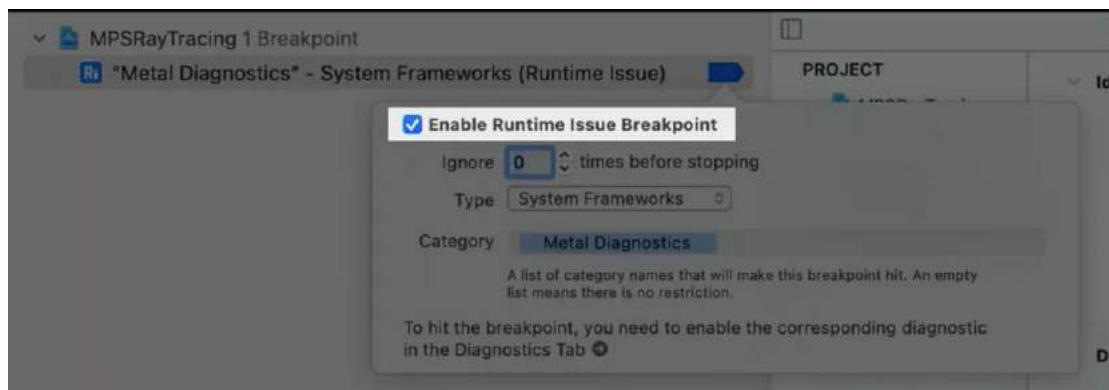
如上图所示，这个功能可以在 GPU 侧发生渲染错误时自动定位和 catch 到错误并定位到代码，以及获取回溯栈帧。

我们可以在 Xcode 中按照以下流程来开启这个功能：

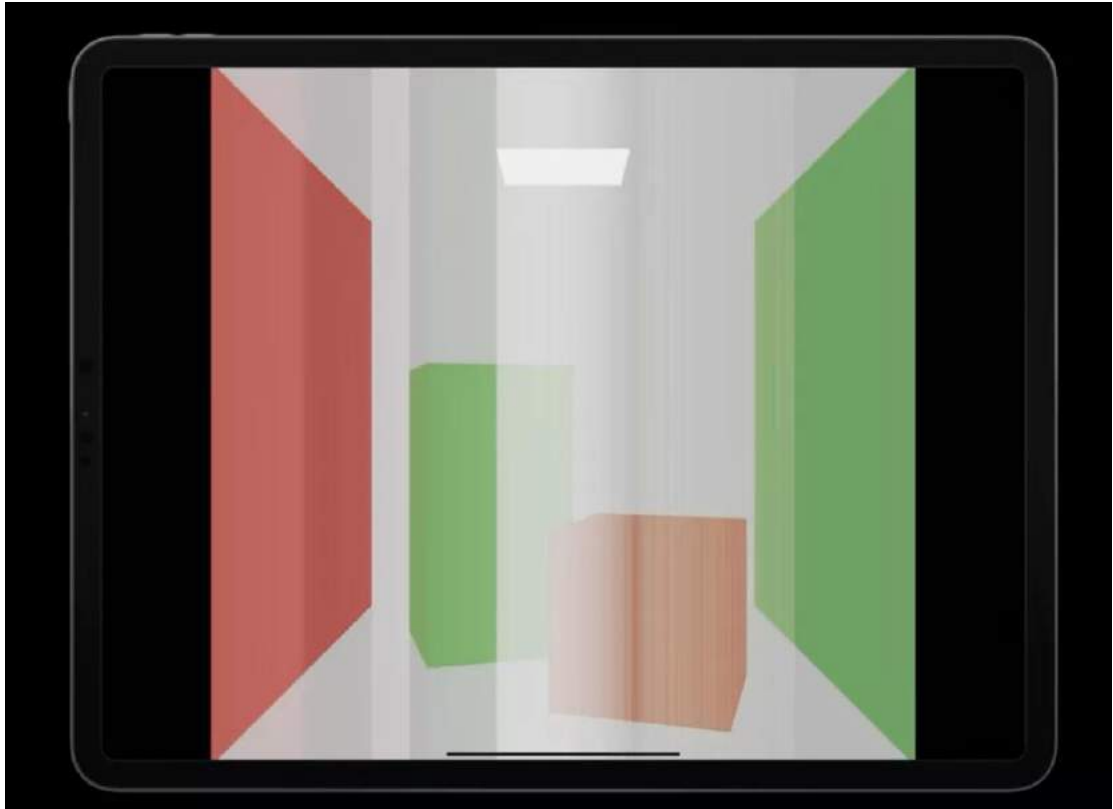
1. 开启 Metal 中的两个 Validation 选项



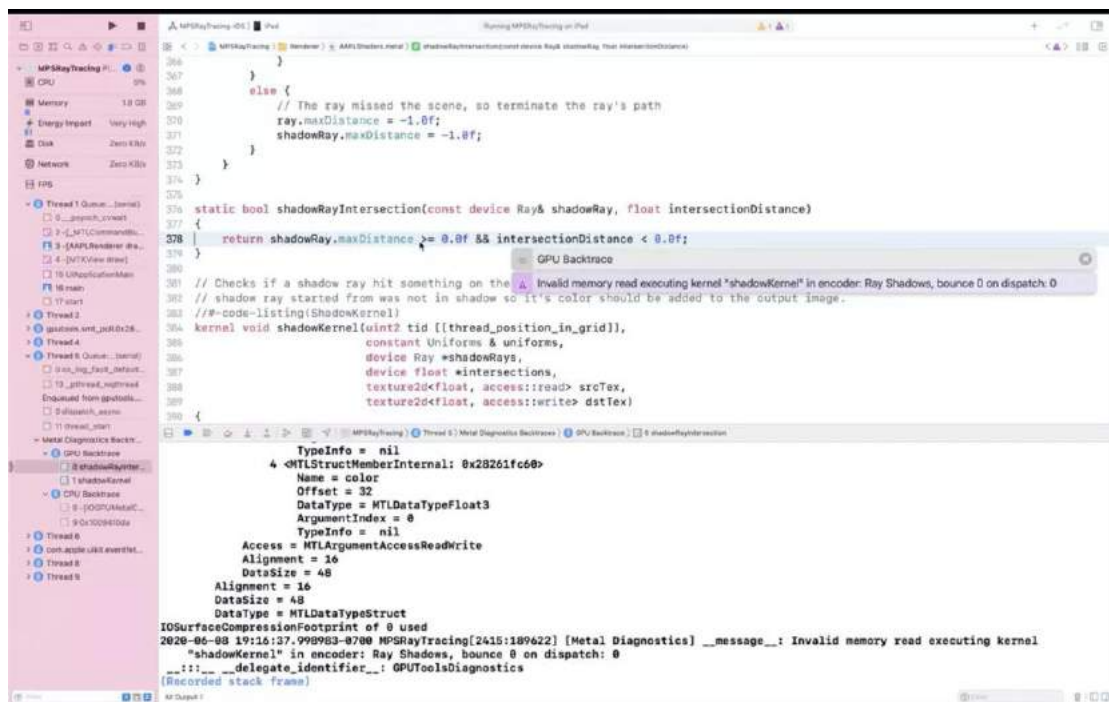
2. 开启 issue 自动断点开关并配置类型和分类等选项



Video 中用了一个 demo 来展示整个工作流，具体参见 [Debug GPU-side errors in Metal](#) (11: 25~14: 45) 大致流程如下图所示：



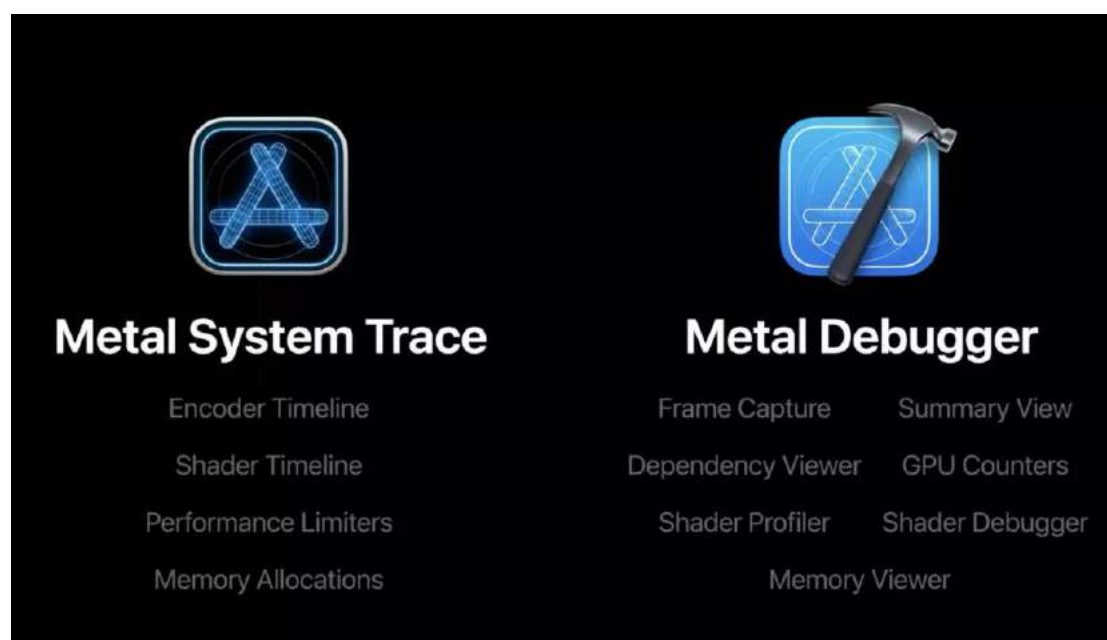
这是一个 Demo 应用程序，很明显它在渲染上出现了一些异常，但是因为是 GPU 侧的问题，所以开发者很难定位。但是通过上述的工作流开启 Shader Validation 之后。



Xcode 会自动断点到发生异常的地方，并展示出异常信息，这样就可以极大的提升开发者的错误修复效率。

Gain insights into your Metal app with Xcode 12

这一章主要讲的是 Xcode12 给 Metal App 提供了更多调试和分析的新工具。大致如下图所示：



主要分为两个部分：

■ Metal Debugger

这个工具可以让开发者在 App 运行时，获取到想分析和调试的任何一帧，然后再进入 Xcode 提供的各种分析界面，总体情况，依赖情况，内存，带宽，GPU，Shader 等各种具体的界面来对这一帧进行更加详细的分析和调试。整个过程使用视频的方式可能会更加高效，所以这里不会进行详细的赘述和分析。详情可以参见 [Gain insights into your Metal app with Xcode 12](#)

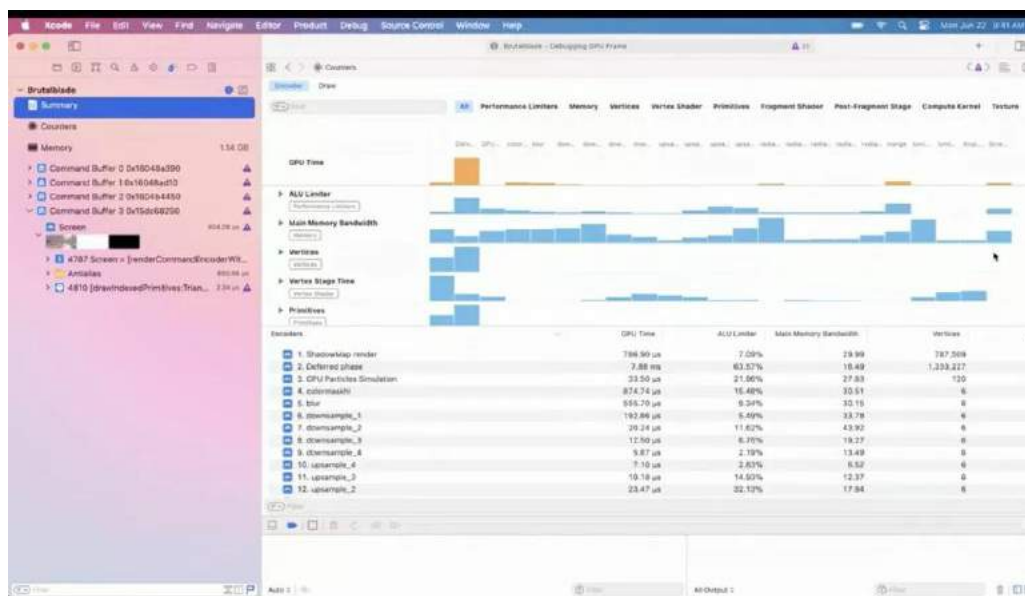
■ Metal System Trace

整个工具跟之前提到过的 Debugger 相比，他的功能主要是让开发者可以随着时间的推移来捕获应用程序的各种信息和特征，可以让开发者很好的调试一些例如终端，帧丢失，内存泄漏等问题。而 Debugger 主要是对某一帧进行调试和分析。

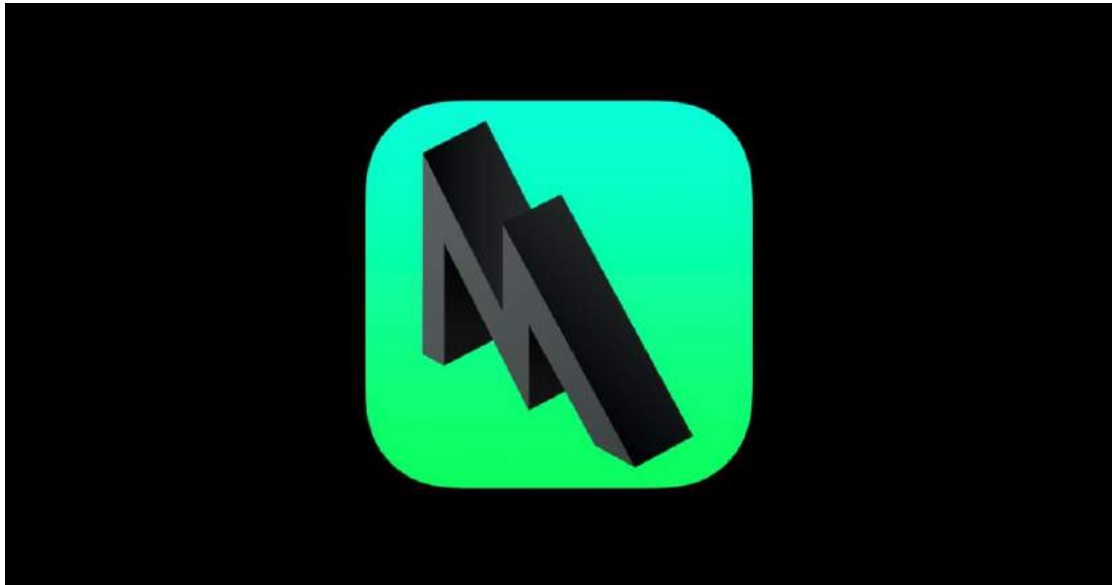
他提供了一个叫做编码时间线的工具，可以让开发者查看到 GPU 在应用运行中的运行各种命令缓冲的情况。然后提供了一个叫做着色器时间线的工具，可以让开发者查看到各种着色器在代码运行期间运行的过程。然后还有 GPU 计数器的工具，这个工具我们在前文进行了详细的分析，主要是用于解决 GPU 的绘制性能问题的工具。然后最后一个工具就是内存分配跟踪工具，可以让开发者查看到应用程序运行过程中各种内存的分配和释放，可以帮助开发者解决内存泄漏问题或者是降低应用内存占用。

技术启发与思考

WWDC 20 关于 Metal 的 Session 中，比较重要的就是官方提供了很多可供开发者进行 GPU 级别的调试工具以及性能分析工具。给比较成熟庞大而复杂的工程突破性能瓶颈，提供更加优秀的用户体验提供了一些思路。



闲鱼作为一个电商类 App，随着功能和增多和以及工程的复杂化，在所难免的会遇到性能瓶颈，而闲鱼团队当前面对挑战的方式是从工程级别来进行优化。从 Flutter 的角度来看，WWDC 20 对于 Metal 的调试工具和性能分析工具的完善，无疑提供了更多的优化思路。这为未来运行在 iOS 上的应用的调优和突破性能瓶颈带来了新的思路和可能性。



对于跨平台框架，Apple 有自家的 SwiftUI，这也是此次大会的重点项目。不过无论是 Flutter，还是 SwiftUI，大家最后对应用的性能瓶颈突破和优化一定是殊途同归的，也就是深入到 GPU 级别来进行开发和调试以及性能分析。对于未来的客户端开发人员，理解 GPU 和进行 GPU 级别的编程肯定是不可或缺的技能点之一。

参考

- <https://developer.apple.com/videos/play/wwdc2020/10602/?spm=ata.13261165.0.0.33874c47d0rPKo>
- <https://developer.apple.com/videos/play/wwdc2020/10605/?spm=ata.13261165.0.0.33874c47d0rPKo>

Swift 5.3 又更新了什么新奇爽快的语法？

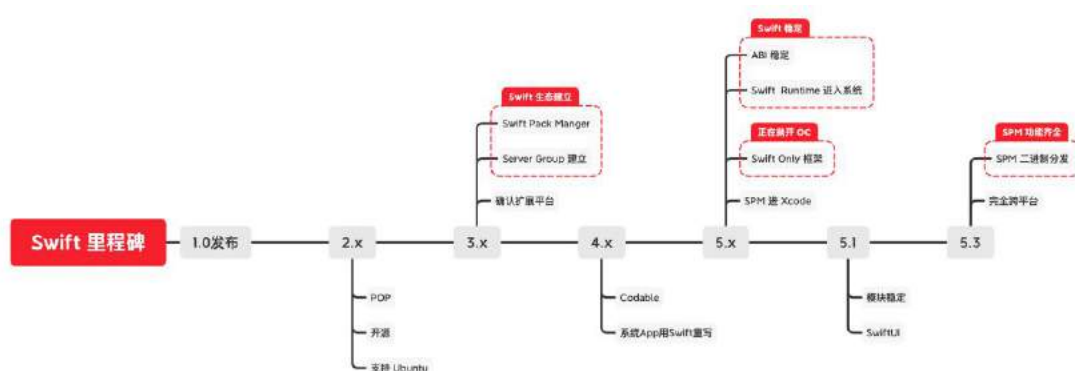
作者 | 蒋志远(星志)

出品 | 阿里巴巴新零售淘系技术

Swift 在 WWDC14 正式发布到 2019，经过 5 年的不断迭代，这其中经历了标准库变动，语法的增减。首先使用 Swift 作为开发语言的开发者们都苦不堪言，戏称《Swift 从入门到重学》，几乎每一年 Swift 都会迎来比较大的改动，甚至 API 都发生了变化。

WWDC 19 苹果发布了 Swift 5.0，苹果终于宣布 Swift 的 ABI 稳定。这标志着 Swift 这门语言已经趋于稳定，在 2019 至 2020 的迭代中，Swift 5.2 也做到了模块稳定，之前的大修大改已经不会出现了。

Swift 发展里程碑



有意思的是，在 WWDC16 中有一页 PPT 写下了 Goals for Swift 3：

Develop an open community
Portability to new platforms
Get the fundamentals right
Optimize for awesomeness

如今已经 2020 年,再回头看这些目标,Swift 5.3 几乎完全实现了。Swift 可谓一直不忘初心,朝着认为正确的方向不断的努力。

语法的不断成熟

每年 WWDC 比较期待的一个点就是看看 Swift 又加了一些什么新奇爽快的语法。我先列举一下 Swift 5.3 新增的语法:

- [SE-0249] KeyPath as Function
- [SE-0279] Multiple trailing closure
- [SE-0281] Type-based program entry point (@main)
- [SE-0266] Synthesized comparable conformation for enum types
- [SE-0269] Increased availability of implicit self in closure
- [SE-0276] Multi-pattern catch clauses
- [SE-0280] Enum cases as protocol witnesses
- [SE-0267] Where clauses on contextually generic declarations
- [SE-0270] Collection operations on noncontiguous elements
- [SE-0264] Standard library preview package
- [SE-0253] Callable values of user-defined nominal types
- [SE-0263] String initializer with access to uninitialized storage

上面列举的 12 个语法改动,有兴趣的同学可以到 [swift-evolution](#) 去查看详细的内容。其中比较有意思的几个改动就是 [SE-279], [SE-281]和[SE-0269],我们来细细品味一下。

■ Design for SwiftUI

[SE-279] Multiple trailing closure 多重尾闭包这个新特性算是一个非常大的改动。这个特性的出现影响了 Swift 非常重要的地方,ABI 设计。先来几个官方的例子:


```
// 1. Single closure argument -> trailing closure

UIView.animate(withDuration: 0.3) {

    self.view.alpha = 0

}

// 2. With trailing closure

UIView.animate(withDuration: 0.3, animations: {

    self.view.alpha = 0}) { _ in

    self.view.removeFromSuperview()

}

// 3. Multiple closure arguments -> no trailing closure

UIView.animate(withDuration: 0.3, animations: {

    self.view.alpha = 0}, completion: { _ in

    self.view.removeFromSuperview()

}))

// 4. Multiple trailing closure arguments

UIView.animate(withDuration: 0.3) {

    self.view.alpha = 0} completion: { _ in

    self.view.removeFromSuperview()

}
```

上面的代码中 2 的情况是经常出现的，尾闭包的含义就不够明确，以前最佳的方式应该为 3，但是两个闭包被套在一个小括号里，使得括号多重嵌套，可读性下降。为了解决这个问题，苹果使用了第 4 中方法——多重尾闭包。可以从代码看出可读性得到了很大的提升。

这一看似优雅的设计其实引入了很多语言的复杂度，有了这个特性你甚至可以写出类似下面的代码，看上去就像是创造了一个 if-else 语句，但其实它是一个函数。

```
func when<T>(  
    _ condition: @autoclosure () -> Bool,  
    then: () -> T,  
    `else`: () -> T  
)-> T {  
    condition() ? then() : `else`()  
}  
// using syntax  
when(2 < 3) {  
    print("then")  
} else: {  
    print("else")  
}  
// equivalent to:  
when(2 < 3, then: { print("then") }, else: { print("else") })
```

这就对 SDK 的开发者提出了更高的要求, 需要小心的使用这个特性来设计 API, 防止这个特性被滥用导致代码的可读性下降, 甚至造成歧义。苹果在 WWDC20 中也强调了这一点。

还有两个特性其实很简单, [SE-0281] 是 @main 来标记程序入口, [SE-0269] 是让开发者在特定的情况下不需要再写多余的 self.。

苹果推出这几个语言特性, 就是为了让战略性的项目 SwiftUI 的 API 更加简洁明了。

可以看下面的例子:

```
@main // <- @main  
  
struct MyApp: App {  
    var body: Scene {  
        WindowGroup {  
            Text("Hello World!")  
        }  
    }  
}  
  
} //-----  
  
Gauge(value: acidity, in: 3...10) {  
    Image(systemName: "drop.fill")
```

```
} currentValueLabel: { // <- multiple trailing closure
    Text("\(acidity, specifier: "%.1f")")
} minimumValueLabel: {
    Text("3")
} maximumValueLabel: {
    Text("10")
} //-----

struct ContentView : View {
    var landmark: Landmark

    var body: some View {
        Button(action: {
            landmark // <- no self.
        }) {
            Text("Button")}
    }
}
```

苹果也不是第一次为了 SwiftUI 改 Swift 的语法, [SE-0255] Implicit returns from single-expression functions 就是为了让 SwiftUI 的 body 不用写 return, 强化 DSL 风格。

苹果为了 SwiftUI, 推出了 `functionBuilder` 来实现 DSL 形式的代码; 为了 SwiftUI, 推出了许多让它更漂亮的语法特性; 为了 SwiftUI, 不惜为语言引入更多的复杂度。

■ 下限低, 上限高

Swift 的最初的设计方向就导致了它是一门下限很低上限很高, 入门容易精通难, 使用容易设计难的语言。它一系列的语法糖和语法设计, 包括类型推断系统, 都是让它成为十分亲民的语言, 而它的一系列语法特性, 让它在设计 API、SDK 实现的时候其复杂程度也不亚于其他语言。

Swift is a general-purpose, multi-paradigm, compiled programming language developed by Apple Inc. for iOS, iPadOS, macOS, watchOS, tvOS, and Linux.

----- Wikipedia

维基百科的后半部分可能有些局限了，但是前半部分说明了它是一门多编程范式的语言，Swift 可以支持 面向对象编程(OOP)，面向协议编程(OPP)，声明式编程(DP)，函数式编程(FP)，泛型编程(GP)。

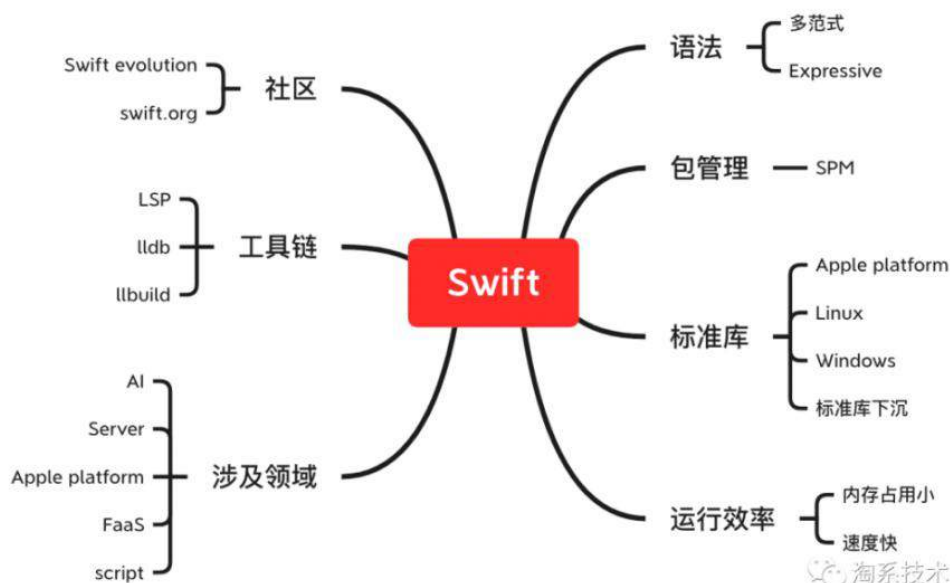
Objective-C 已经发展了这么多年，如此成熟，为什么现在苹果要开始抛开它？Objective-C 本来就是生于一个面向对象编程范式起飞的一年，与 C++ 一样为了拓展 C 命令式编程范式而诞生的语言。当时 C 语言虽然也可以实现 OOP，但是语法设计成为了限制。在当代计算机编程语言研究演进下，出现了很多编程范式的新理论，如函数式编程，元编程等等，同样 OC 可以通过它的方式来实现，但是语言的冗杂和 OOP 的设计已经不能更好的表达这些概念，就犹如纯 C 来表达 OOP 一样。

苹果推出 Swift 就是为了摆脱 OC 的束缚，让它能更好的践行现代的编程理论，所以才会诞生出 SwiftUI，才会有 Combine、map/filter/reduce 等这些库和 API。类型推断和元编程的理论也让 Swift 在保留强类型的环境下还能保持如此简洁优雅，可读性强的代码。而这些是 OC 无法做到的。尝试使用 OC 对这些现代概念做表达的 SDK 都会显得十分冗杂。

现在 Swift 语法已经不会再大变，有如此现代、安全、稳定、富有想象力的语言，有什么理由不真香呢？

土壤已经肥沃

我们先通过下面的脑图感受一下 Swift 更迭到 5.3 已经取得的成就。



现在 Swift 生态环境已经自成一派，有一套完整的工具链保证开发，有更独立的标准库让它可以自由迁移，包管理让丰富了它的 SDK，同时还具有 Native 语言的各种优势。在这么多年的发展中，它的能力已经触及到了 AI、Server、Mobile Device、FaaS，强大的标准库让它甚至可以当脚本语言使用。

Swift 如今已经不再孤立无援，开源让 Swift 变得生机勃勃，合理优雅的设计和开放的态度让全球的开发者们都在不断的完善它。

在 Swift 的大生态中，包管理工具是最值得一说的。一门语言它的能力，取决于它是否有强有力的 SDK，还有就是获取他们的途径。基础的 Swift 的能力都由工具链和标准库提供，而强有力的 SDK 最好的方式就是通过包管理工具快速获取。Swift Package Manager 就是 Swift 的军械库。

Swift Package Manager

看一下 Swift Package Manager(SPM) 的发展历程，SPM 从 Swift 3.0 时就发布，当时只支持 Git 远端仓库，支持源码发布。WWDC 19，Xcode 引入了一个新的架构 XCFramework，一个旨在能打包多个平台 framework 的文件结构。WWDC20 SPM 宣布正式支持发布二进制 framework。

WWDC20 SPM 走出的这一步标志着功能的完善。为什么这么说，因为在 iOS 届，解决了 Xcode 无包管理工具难题，让众多优秀的开源库能被快速获取的包管理工具 —— Cocoapods。在 SPM 出现之前，Cocoapods 的功能已经非常完善，打败了当时另外的一款包管理工具 Carthage。

Cocoapods 作为基于 Xcode 开发的第三方包管理工具，通过修改 Xcode 的工程信息来实现处理各个包的依赖问题。它支持依赖的二进制分发、源代码分发，基于 framework 良好的资源管理。现在许多公司的大型 App 也是用 Cocoapods 做模块化组件分离，通过二进制依赖的方式来提高打包速度。

然而比较可惜的是它工作流是脱离 Xcode 的，Podfile 更新等操作都是在工程之外。语言使用的是 Ruby，虽然使用了一些操作使 Podfile 变成了 DSL，但是对于开发者而言是有一定门槛的，特别是需要编辑发布 SDK 所使用的 podspec 时，对照文档，无代码提示的编写让写正确配置文件都成了难事。Cocoapods 还不得不设计出 podspec lint 来帮组开发者确认这个事情。

对于 Swift PM 这个亲儿子而言，就不存在这个问题，SPM 已经深入的集成到了 Xcode 中，所有的操作都可以利用 Xcode 完成。在编写 Swift Package 时，Xcode 现在已经具备将 Package.swift 作为工程项目打开，并且现在工程模板中已经包含了 Swift Package。强大的 Xcode 自动补全让编写 Package.swift 不再是一件难事。如果脱离 Xcode，利用 swift-lsp 结合现在热门的文本编辑器，如 VSCode 等都可以实现相应的功能。SPM 现在已经是独立于 Xcode 的存在，可以为 Swift 提供强有力的支持。

SPM 在功能层面已经不逊于 Cocoapods，在 WWDC20 时，SPM 已经支持以二进制库形式分发 Package，SPM 也可以管理包中的资源和本地化，基本的能力已经与 Cocoapods 差异不大，然而最关键的是，SPM 是纯 Swift，开源，还有苹果官方支持的包管理工具。

■ 新瓶酿酒酒更香

国外的不少 APP 已经迁移到了 Swift；三方开源库如 AFNetworking 已经用 Swift 重写为 Alamofire，Lottie 已经完全被 Swift 重写替代。苹果也推出了许多 Swift Only 的库。苹果也在利用 Swift 为 UIKit，GCD 等基础库不断提供更具有表现力的 API。

Uber 完成了迁移，收获了 Swift 极强的稳定性。Alamofire, SnapKit 用 Swift 重写，获得了更加具有表现力的 API，开发者更容易接受且喜爱。Reactive 系列的编程风格在 Swift 上大放异彩。

机会与展望

现在 Swift 发展了这么多年，遍地开花，Objective-C 已经有在逐渐被废弃的趋势。现在很有可能都很难招到认真学习过 OC 的应届生了。

Swift 在集团内部的发展，还有很多的工作要做，这其中充满了挑战。我们也在积极探索 Swift 在手淘的落地，取得了 Swift 5.1 能模块在手淘中正确运行起来的阶段性成就。但是比较遗憾的是还有很多模块是无法让 Swift 正确跑起来的，以现在的工具链来说甚至源码依赖调试都无法做到。

但是这都是暂时的，现在时机已经成熟，Swift 的语言特性，SPM，工具链，标准库都已经足够强大。未来，我们会尽快大力升级 Swift 的基建。让 Swift 的花在集团内开起。先定个小目标，希望 Swift 能成为集团 iOS 客户端开发的首选语言。

参考

[SwiftUI 背后那些事儿](#)

[WWDC20 What's new in Swift](#)

[WWDC19 What's new in Swift](#)

[WWDC18 What's new in Swift](#)

[WWDC17 What's new in Swift](#)

[WWDC16 What's new in Swift](#)

[WWDC20 Swift packages: Resources and localization](#)

[WWDC20 What's new in SwiftUI](#)

[Swift 5 时代的机遇与挑战到底在哪里？](#)

[Swift Evolution](#)

Apple Widget: 下一个顶级流量入口?

作者 | 王浙剑(柘剑)

出品 | 阿里巴巴新零售淘系技术

2020 年 6 月 22 日, 苹果召开了第一次线上的开发者大会 - WWDC20。这可谓是一次可以载入史册的发布会, 宣布了 ARM 架构 Mac 芯片、软硬件的生态统一、iOS 14 系统界面大改等一系列激动人心的消息。

当然, 最让我感兴趣的就是让 iOS 界面大改的 Widget 了。过去几年, iOS 的桌面交互体验可谓是一言难尽, Widget 的加入无疑是一次比较大的破局。在看发布会的时候, 我的脑海里就浮现出一个问题: “这会是下一个互联网公司竞争的流量入口吗?”

先不抛结论, 让我们先看一下 WWDC20 介绍了哪些新东西。

什么是 Widget?

Widget 不是一个小型的 App, 它是一种新的桌面内容展现形式, 主要是用于弥补主应用程序无法及时展示用户所关心的数据。如下图所示:



一个优秀的 Widget 需要有三个特点: 简单明了 (Glanceable)、恰当展示 (Relevant)、个性化定制 (Personalized)

■ 简单明了 (Glanceable)

Widget 不是一个小型的 App, 这句话被反复提起。一般用户每天进入主屏幕的次数超过 90 次, 但停留的总时长不过几分钟。通常来说用户只会在主屏幕上停留片刻时间, 就会跳转到其他地方, 所以并不需要任何复杂的交互设计来增强 Widget 的作用, 也不需要复杂的样式来丰富 Widget 的内容, 简单明了的内容才是 Widget 的关键。

和安卓的 Widget 不太一样, 苹果设计的 Widget 并不支持任何交互行为, 也不建议大家设计过于复杂的样式来呈现内容, 这也非常符合苹果对于主屏幕的改进一直保持克制的特点。

■ 恰当展示 (Relevant)

苹果期望 Widget 可以和正在执行或者考虑的事情紧密的结合。比如, 早上起床, 用户最关心天气怎么样, Widget 可以展示一下天气情况; 起床后, 用户就要了解一下一天的行程, Widget 可以展示一下 Reminders 中的内容; 等到一天忙完了, 准备睡觉的时候, 可以用 Widget 打开音乐稍微放松一下。为此, 苹果系统提供了一个叫智能叠放 (Smart Stacks) 的功能, 智能叠放是一个 Widgets 的集合。系统会根据每个人的习惯, 借助端智能的能力, 自动的显示准确的 Widget 在最顶部。



当然，苹果也考虑到了一些特殊的场景，比如 Widget Gallery 浏览时，提供了 Snapshot 的能力给到开发者可以定制展示样式，当加载内容的时候提供了 Placeholder UI API 而不是单调的 loading 加载框来避免过多的白屏的尴尬局面。这些设计的目的只有一个，苹果期望 Widget 可以在任何特定的场景都可以展示合理的样式。

■ 个性化定制 (Personalized)

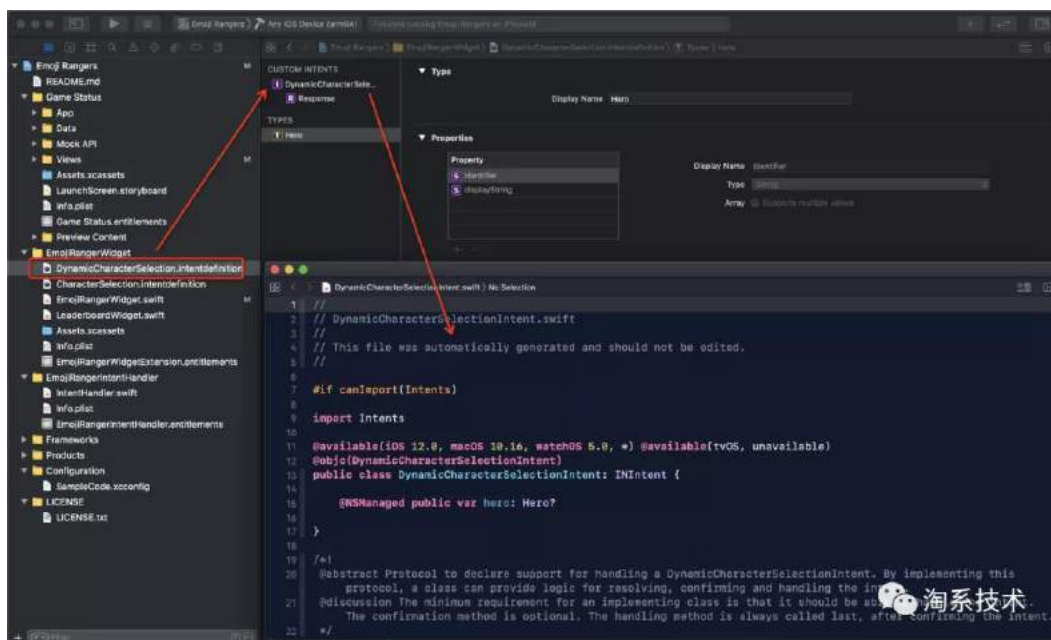
Widget 需要一定的定制能力，比如当我添加一个天气 Widget，我只需要关心杭州的天气怎么样。为了实现这个能力，苹果给 Widget 提供了 Configuration 的能力。顾名思义，就是可配置。一共有两种配置类型：



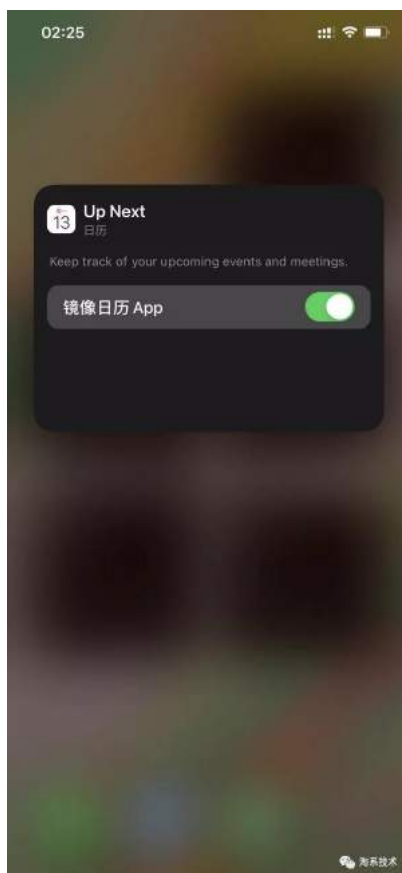
StaticConfiguration，也就是用户无需配置，展示的内容只和用户信息有关系。

IntentConfiguration，支持用户配置及用户意图的推测功能。

IntentConfiguration 的实现是基于 Intents.framework，开发过 SiriKit 和 Shortcuts 一定知道 Intents API 是用于了解用户意图的。其实就是一个智能的表单系统，开发者创建一个 SiriKit Intent Definition File 之后，只需要简单的配置，Xcode 会自动帮你生成对应的代码和类型。

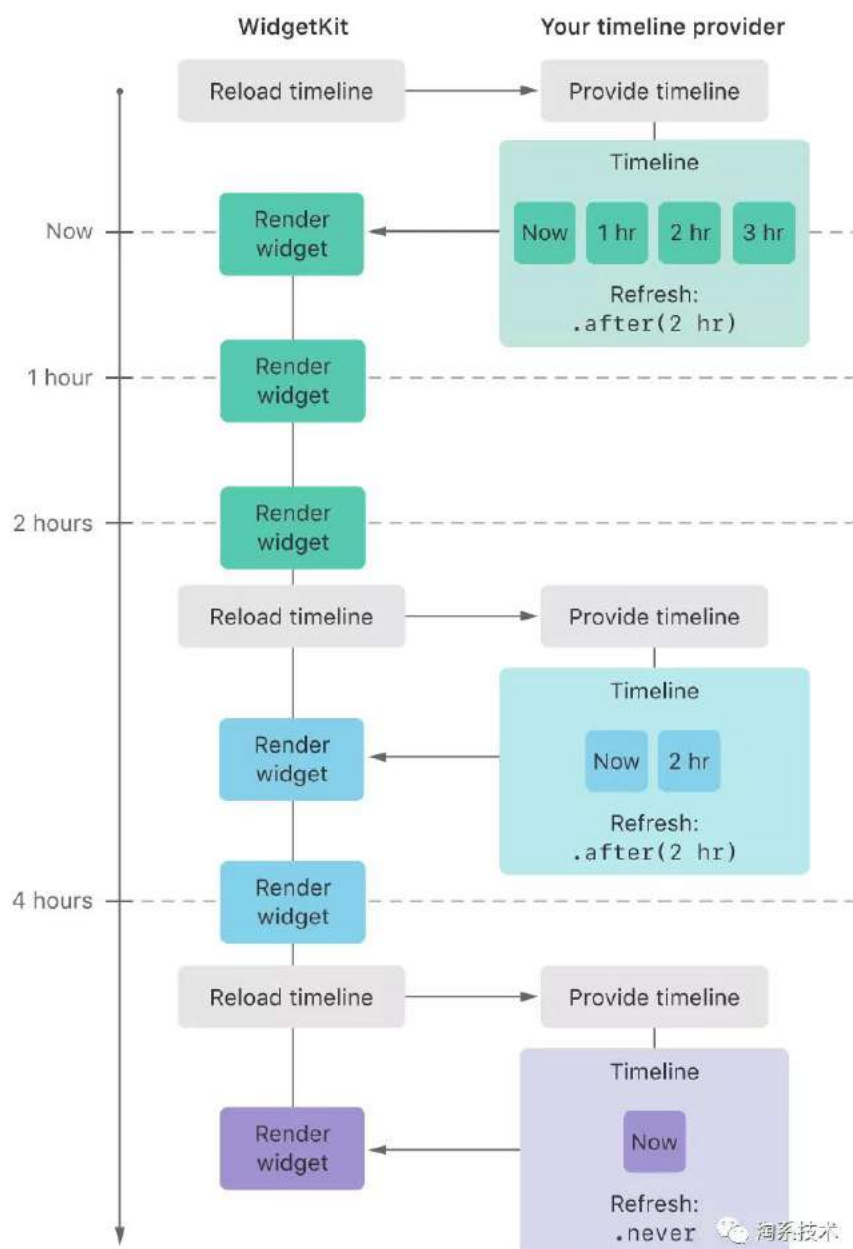


当开发者编写完配置之后，会借助 Intents.framework 的能力，在运行的时候直接绘制出一个配置页面（如下图所示），开发者并不需要关心如果编写这个页面。



Widget 的刷新方式

Widget 的刷新方式是很特别的，相当的克制。在展开讲刷新方式之前，要讲一个概念，叫 Timeline。顾名思义，就是时间线，下面的图就是一条 Timeline。



当系统的 **WidgetKit** 调用 **Reload Timeline API** 之后，会要求 **Widget Extension** 的 **Timeline Provider** 提供一组 **TimelineEntry** 和 **ReloadPolicy**，用来后续刷新页面。

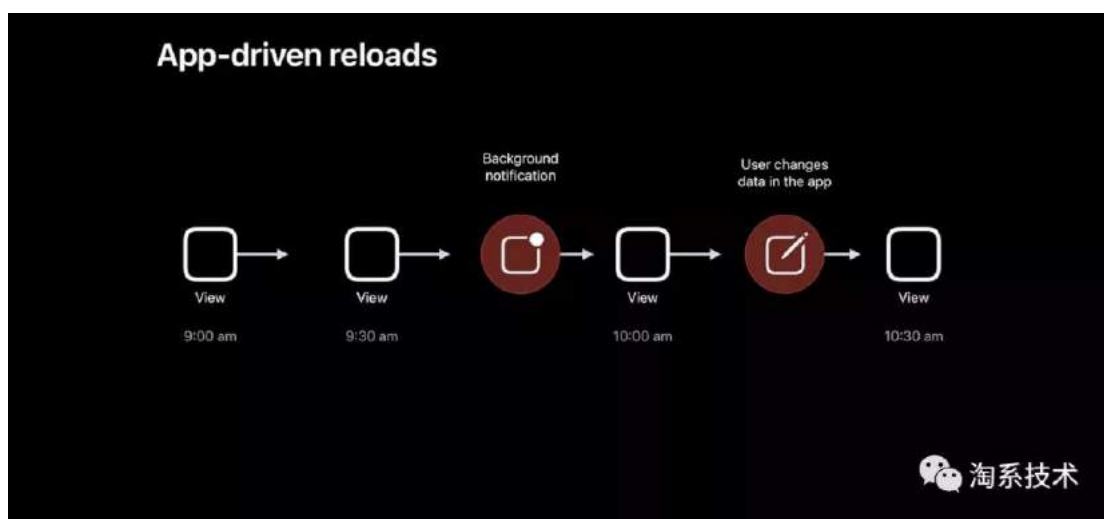
这里的概念比较多，我们一个一个来解释。

首先，Widget 的刷新完全由 WidgetCenter 控制。开发者无法通过任何 API 去主动刷新 Widget 的页面，只能告知 WidgetCenter，Timeline 需要刷新了。

系统提供了两种方式来驱动 Timeline 的 Reload。System Reloads 和 App-Driven Reloads。

System Reloads: 这个行为由系统主动发起，会调用一次 Reload Timeline 向 Widget 请求下一阶段刷新的数据。系统除了会按时发起 System Reloads 之外，还会借助端智能的能力，动态决策每个不同的 Timeline 的 System Reloads 的频次。例如被查看次数很大程度上直接决定了 System Reloads 的频率。当然还有一些由于设备环境变化触发的行为也会触发 System Reloads，比如设备时间进行了变更。

App-Driven Reloads: 指的是 App 请求 Widget 下一阶段刷新的数据。这里也要分两种场景，应用在前台运行和应用在后台运行。当应用在前台运行的时候，App 可以直接请求 WidgetCenter 的 API 来触发 Reload Timeline；而当应用处于后台时，后台推送（Background Notification）也可以触发 Reload Timeline。



注意，前面所提到的 Reload Timeline 并不是直接刷新 Widget，而是 WidgetCenter 重新向 Widget 请求下一阶段的数据。而 Timeline Provider 就是提供这个数据的对象。

而 Timeline Provider 提供的数据有两部分，一部分是 TimelineEntry，另外一部分是 ReloadPolicy。

TimelineEntry 是某个时间节点下 Widget 需要呈现的视图信息和时间点。

而 ReloadPolicy 则是接下来这段时间 Timeline 的刷新策略，一共有三种：

atEnd: 是指 Timeline 执行到最后一个时间片的时候再刷新。

atAfter: 是指在某个时间以后有规律的刷新。

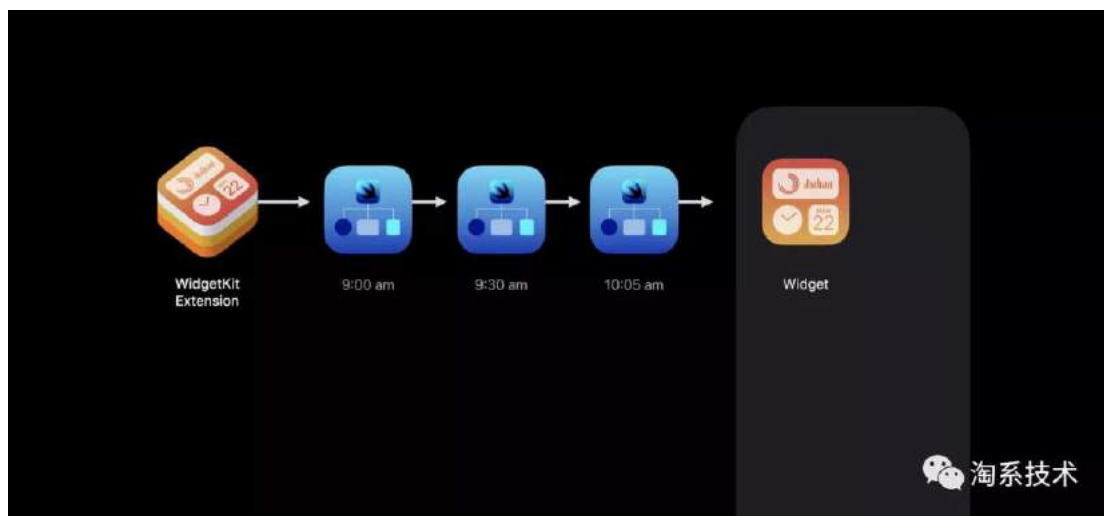
never: 是指以后不需要刷新了。什么时候需要重新刷新需要 App 重新告知 Widget。

当 Timeline Provider 提供完下一阶段的数据之后，就会停止运行。系统也会根据 entry 的信息，到点对 Widget 的展示内容进行刷新。值得一提的是，WidgetKit 会把 Timelines 所定义的 Entries 对应的 Views 结构信息缓存到磁盘，然后在刷新的时候才通过 JIT 的方式来渲染。这使得系统可以在极低电量开销下为众多 Widgets 处理 Timelines 信息。

简而言之，苹果对 Widget 的刷新相当的克制。开发者无法直接决定 Widget 刷新，只能提供刷新策略。具体的时间和节奏全部由系统来控制。苹果这么做，大概率是为了提高主屏幕的性能和减少电量开销上的考虑。

Widget 和 SwiftUI

Widget 只能用 SwiftUI 来进行开发，确切的说，Widget 的本质是一个随着时间线而更新的 SwiftUI 视图。



当我最开始知道这个限制的时候，说实话是相当震惊的。众所周知，SwiftUI 是一个去年才发布的新技术，而且最开始的时候 SwiftUI 是相当不稳定的，以至于苹果自己都是建议开发者暂时不要用到生产环境上，Widget 作为系统主屏幕的功能，强制使用这么新的技术，会不会太激进了？

显然是不会。苹果要求 Widget 只能使用 SwiftUI 主要是基于几点考虑：

- 1、SwiftUI 经过一年的发展，有了很大的提升，不仅可以使用 SwiftUI 来构建整个应用程序，而且在一些方面已经优于基于 UIKit 的开发方式了。具体的内容，大家可以看一下《详解 WWDC 20 SwiftUI 的重大改变及核心优势》
- 2、苹果正在布局跨平台，大统一的策略。Widget 作为系统的核心功能，使用 SwiftUI 是唯一的选择。SwiftUI 精美的 DSL 设计，使得开发者使用一套代码在 iOS、iPadOS、macOS、watchOS 和 tvOS 等多个平台展示不同的样式可以轻松的实现。（Widget 只会在 iOS、iPadOS 以及 macOS 上展示）
- 3、使用了 SwiftUI 使得 Dynamic Type 和 Dark Mode 等问题适配起来成本很低。
- 4、只有使用 SwiftUI 才能达到很多对于 Widget 的限制。倘若可以使用 UIKit 开发者可能有无数种办法绕过苹果的限制。比如开发无法使用 UIViewRepresentable 来桥接 UIKit，只要使用任何 UIKit 的元素会直接 Crash。
- 5、将 Swift 语言和 SwiftUI 的重要程度提升了一大截。

Widget 的展示形式

■ 一个 App 可以对应多个 Widget Extension

你可以使用 WidgetBundle 来进行组装。苹果并没有对 Widget Extension 有数量上的限制。所以为了避免大家开发过多的 Widget Extension 导致搜索起来麻烦, 在 Widget Gallery 中只能看到一个条目。



■ 一个 Widget Extension 一共只有三种尺寸

考虑到简单明了的特点以及手机屏幕的空间有限的问题。苹果只提供了三种样式可以选择, systemSmall (2 * 2 icon 区域)、systemMedium (2 * 4 icon 区域)、systemLarge (4 * 4 icon 区域)。



■ 同一种 Widget 可以被多次添加到主屏幕中

而且对于每一个 Widget 来说，都有其对应的独立 TimeLine，相互独立，互不干扰。



■ 开发者无法开发智能叠放 (Smart Stacks)

开发者无法开发一个 Widget 的集合。智能叠放 (Smart Stacks) 是一个系统特有的能力, 对于开发者来说, 唯一可以做的就是主动提供相关性信息。前文提到了 Timeline 的数据又一组 TimelineEntry 组成, 而每个 TimelineEntry 除了包含时间点和视图信息以外, 还可以包含一个 TimelineEntryRelevance 对象, 用来表示这个 entry 的相关性。

■ 不可交互, 只可点击

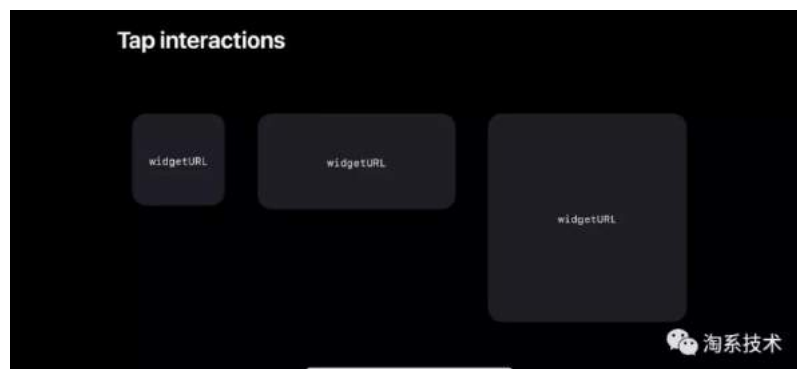
Widget 的 UI 是无状态的, 不支持滚动, 也不支持像 Switch 一样的互动元素。唯一开放的能力只有通过点击和 DeepLink 来唤起主 App。

苹果提供了两种 API 给到开发者, 第一种是 SwiftUI widgetURL API, 代码如下所示:

```
struct EmojiRangerWidgetEntryView: View {  
    var entry: Provider.Entry  
  
    var body: some View {  
        HStack(alignment: .top) {  
            AvatarView(entry.character)  
                .foregroundColor(.white)  
            Text(entry.character.bio)  
                .padding()  
                .foregroundColor(.white)  
        }  
        .padding()  
        .widgetURL(entry.character.url)  
    }  
}
```

淘系技术

而 widgetURL 的可点击区域如下:



对于 systemSmall 类型来说, 只支持 widgetURL 的方式, 但是 systemMedium 和 systemLarge 还可以使用 SwiftUI Link API, 代码如下所示:

```
var body: some View {
    VStack(spacing: 48) {
        ForEach(
            characters.sorted { $0.healthLevel > $1.healthLevel }, id: \.self) { character in
                Link(destination: character.url) {
                    HStack {
                        Avatar(character: character)
                        VStack(alignment: .leading) {
                            Text(character.name)
                                .font(.headline)
                                .foregroundColor(.white)
                            Text("Level \(character.level)")
                                .foregroundColor(.white)
                            HealthLevelShape(level: character.healthLevel)
                                .frame(height: 10)
                        }
                    }
                }
            }
    }
}
```

而 Link 的可点击区域如下:



同时, 为了性能和耗电量的考虑。Widget 不能展示视频和动态图像。所以期待通过动效吸引用户眼球的方式可以暂时息熄火了~

总结与展望

Widget 的出现, 让 iOS 系统的桌面有了破局, 一定会有很多产品都期待借助 Widget 来丰富自己产品的内容表达。

但是, Widget 设计的初衷是简单明了的在恰当的时机展示一些带有个性化定制的内容, 为了不让主屏幕的整体使用体验变得复杂, Widget 从技术上就做的很克制, 限制了很

多很多的能力。因此我认为 Widget 不会成为下一个互联网公司竞争的流量入口, 它会成为 App 提高用户体验的利器。

从技术角度看, SwiftUI Only 这种看似“激进”的策略其实也是一种信号, 其实也是在告诉大家苹果对于 Swift 以及 SwiftUI 的重视程度。

虽然, 从目前来看 Pure SwiftUI 的设计, 可以做的事情真的很少, 但是我也相信, 苹果会不断优化 Pure SwiftUI 的能力。让开发者可以以最低的开发成本, 适配更多的平台。

最后, 也期待大家可以好好研究一下 Widget, 结合自己的产品, 给到用户极致的用户体验。

参考

iOS 14 Preview

<https://www.apple.com.cn/ios/ios-14-preview/>

Widgets code-along

<https://developer.apple.com/news/?id=yv6so7ie>

Meet WidgetKit

<https://developer.apple.com/videos/play/wwdc2020/10028/>

What's new in SwiftUI

<https://developer.apple.com/videos/play/wwdc2020/10041/>

Add configuration and intelligence to your widgets

<https://developer.apple.com/videos/play/wwdc2020/10194/>

Build SwiftUI views for widgets

<https://developer.apple.com/videos/play/wwdc2020/10033/>

Creating a Widget Extension

<https://developer.apple.com/documentation/widgetkit/creating-a-widget-extension>

Building Widgets Using WidgetKit and SwiftUI

https://developer.apple.com/documentation/widgetkit/building_widgets_using_widgetkit_and_swiftui

Making a Configurable Widget

<https://developer.apple.com/documentation/widgetkit/making-a-configurable-widget>

Keeping a Widget Up To Date

<https://developer.apple.com/documentation/widgetkit/keeping-a-widget-up-to-date>

Swift 5.3 的进化：语法、标准库、调试能力大幅提升

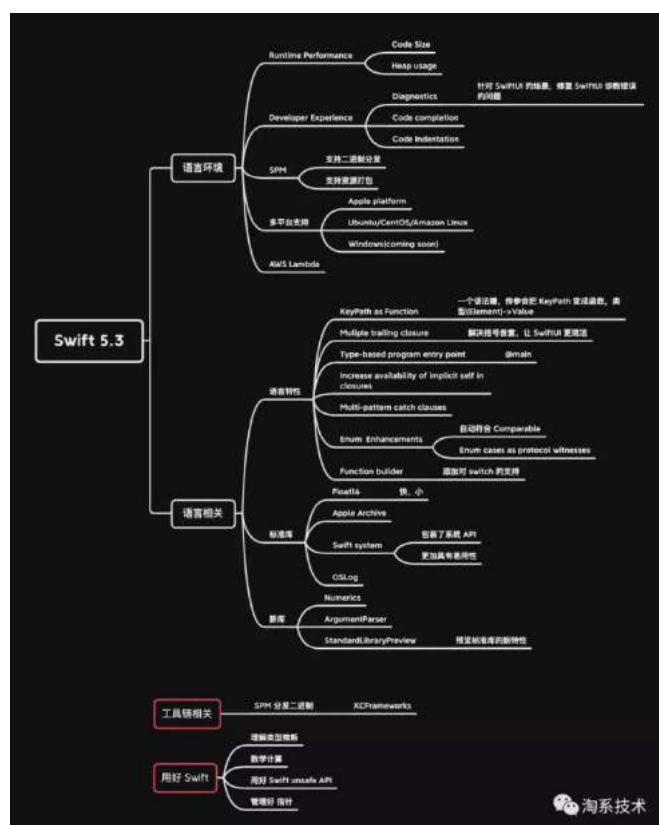
作者 | 蒋志远(星志)

出品 | 阿里巴巴新零售淘系技术

Swift 从 5.0 的 ABI 稳定到 5.1 的模块稳定，Swift 终于不是《Swift 入门到重学》了。本次 WWDC2020，Swift 5.3 正式发布，Swift 依旧朝着安全、高效、易读的方向持续发力，不断的在改进语法，增强代码的表达能力和易用性。因为 Swift 的模块稳定，SPM 现在也支持了二进制模块的分发，逐渐完善的社区生态也在不断拓宽 Swift 可以涉足的领域，而不仅仅是在 Apple 平台之上。

Swift 发展里程碑

下图展示了 WWDC2020 中 Swift 相关内容的脑图，希望可以帮助大家快速了解。



语言环境的完善和拓展

一门完善编程语言有三个最基本的要素：语法、标准库、调试能力。语法设计决定了语言的编程范式；标准库决定了语言的基本能力；调试能力决定了开发者的体验和语言的稳定性。

苹果在 Swift 的迭代过程中不断的强化这几点，我们可以来看看 Swift 又得到了哪些提升。

语法特性


Swift 的语法设计核心还是 OOP，但是这不妨碍 Swift 的语法在支持 POP 和函数式编程甚至 DSL 得到的强化。Swift 也因为 SDL 特性的加入，开始逐渐的适应声明式编程的方向发展，比如后文提到的 @main 等等。

Multiple trailing closure

```
// Multiple trailing closure syntax
UIView.animate(withDuration: 0.3) {
    self.view.alpha = 0
} completion: { _ in
    self.view.removeFromSuperview()
}
```

SE-0281

NEW

 淘系技术

这个改进解决了当函数最后几个参数为闭包的情况下，导致的括号嵌套的问题，API 更加简洁也更加具有表达性。SwiftUI 利用这个语言特性，也变得更加简洁易懂。

KeyPath as Function

```
KeyPath<Target, Value> --隐式转换--> (Target) -> Value
```

现在 KeyPath 可以当做函数来使用了。这个语法糖解决的问题当我们使用类似 map 一样的函数时，只需要取出对应数据模型中的某一个属性，为此我们不得不写类似 map { \$0.property } 的代码，有了这个语法糖，事情就可以简化成了 map(.property)。

■ Type-based program entry point (@main)

引入了新的修饰符 @main，可以标记在带有 public static func main() 函数实现的所有类型，无论 main 函数是从拓展来得还是继承来的。添加这个特性的意义在与维持声明式的语义，将声明式语义进行到底。YES！

■ Increase availability of implicit self in closures

以前我们写逃逸闭包时如果捕获了 self，我们需要在跟 self 有关的地方写上 self。以警示我们注意循环引用。

```
// Increased availability of implicit self in closures

UIView.animate(withDuration: 0.3) {
    self.recordingView?.alpha = 0.0
    self.textView.alpha = 1.0
} completion: { _ in
    self.activeTransitionCount -= 1
    if !self.isRecording && self.activeTransitionCount == 0 {
        self.recordingView?.removeFromSuperview()
        self.recordingView = nil
    }
}
```

淘系技术

如果在闭包的捕获列表中显示声明捕获 self，在闭包中对 self 相关的访问可以省略。如果是在不可变的函数中访问，self 可以直接省略（为了 SwiftUI）。

```
// Increased availability of implicit self in closures

UIView.animate(withDuration: 0.3) { [self] in
    recordingView?.alpha = 0.0
    textView.alpha = 1.0
} completion: { [self] _ in
    activeTransitionCount -= 1
    if !isRecording && activeTransitionCount == 0 {
        recordingView?.removeFromSuperview()
        recordingView = nil
    }
}
```

SE-0269 NEW
淘系技术

其实这一点改进有用但是覆盖面并不是很广，因为在实际的应用中，我们都是尽量先弱引用 self 后再强引用 self，来保证 self 的可访问性。在如下场景就得不到此项优化：

```
requestDataAsync { [weak self] in

    guard let self = self else {return}

    self.property // ❌
```

```
property // ☒ 需要显式声明 self  
}
```

Multi-pattern catch clauses

这种写法可以自动实现错误匹配，进入到对应的错误处理中，而不需要使用 switch，增强语言的可读性。

```
// Multi-pattern catch clauses  
  
import System  
  
do {  
    fd = try FileDescriptor.open(  
        path, .readOnly, options: .create, permissions: .ownerReadWrite)  
} catch Errno.noSuchFileOrDirectory, Errno.notDirectory {  
    // create directory structure first!  
} catch {  
    print(error)  
}
```

SE-0276 NEW

淘系技术

Enum enhancements

自动符合 Comparable

编译器现在可以自动为你生成 Comparable 的相关方法，实现 Enum 的比较。

```
// Synthesized comparable conformance for enums  
  
enum MessageStatus: Hashable, Comparable {  
    case draft  
    case saved  
    case failedToSend  
    case sent  
    case delivered  
    case read  
  
    var wasSent: Bool {  
        self >= .sent  
    }  
}
```

SE-0266 NEW

淘系技术

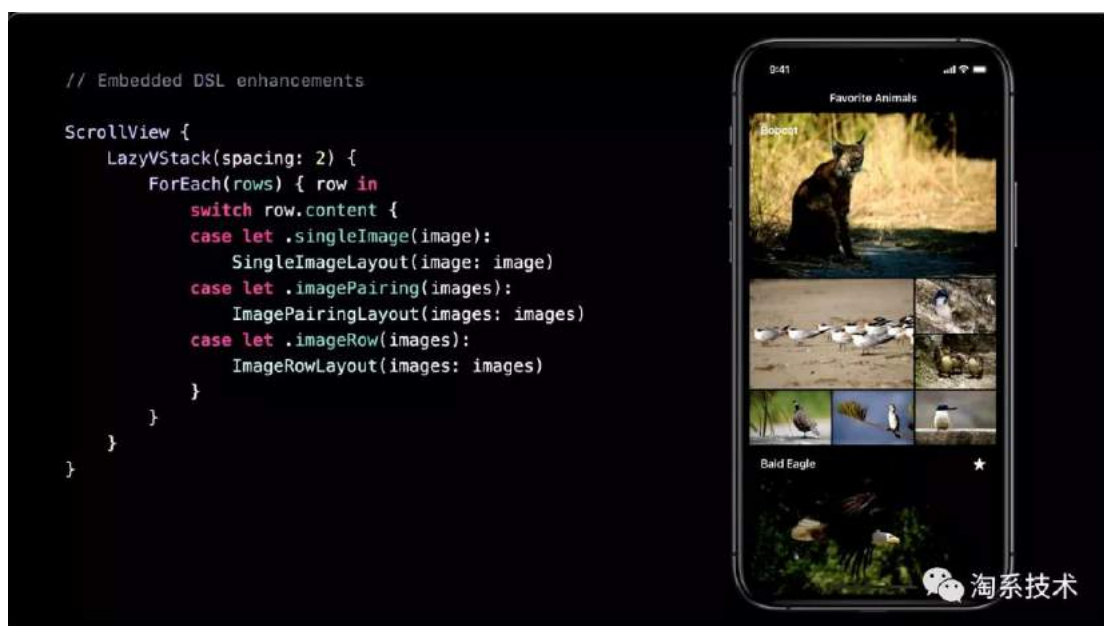
Enum case 可以用来适配 protocol

这个特性看引用场景需要吧，官方给了一个比较好的例子。具体相关内容可以参考 S E-0280。



DSL 新增对 switch 的支持

现在可以在 SwiftUI 等 Swift DSL 中使用 switch-case 来进行模式匹配，之前只有对 if-else 的支持。某种程度上也是为 SwiftUI 而生的能力。



标准库

其实这里说标准库是广义的标准库，其中包括了开发者随语言分发的标准库，如标准 I/O 库等，运行时环境，编译环境，一方库等等。苹果今年在这部分下了功夫，因为标准库、各种各样的一方三方库才是展现一门语言能力的地方，否则再好的语法也不会有人用，语言终究还是工具，能解决问题才是关键。

Swift 5.3 在代码尺寸和运行时都有不小的提升。Swift Package Manager(SPM) 增加对二进制和资源的支持，深度集成 Xcode，亲儿子的优势逐渐显示了出来。Swift 本来设计的初衷本来就是一门 General Purpose 的语言，今年苹果正式宣布 Swift 支持了 Apple platform, Ubuntu/CentOS/Amazon Linux，不久的将来也会支持 Windows，正式成为一门优秀的跨平台语言。

■ 标准库更新

- Float16 的支持，具有更好的运算性能
- Apple Archive 一种功能类似 zip 的压缩文件，苹果就是用这种格式来更新系统
- Swift System 对操作系统基础 API 的包装，让 API 更健壮易用
- OSLog 推荐使用的 logging system

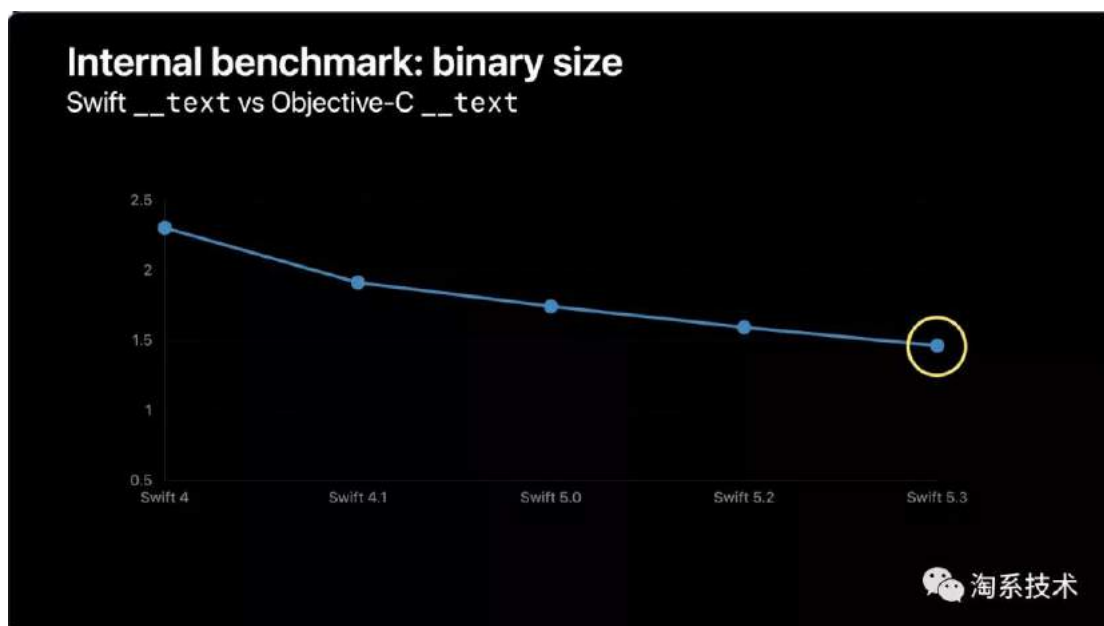
从这些更新可以看出，苹果对 Swift 的底层操作非常关心，Swift System 的出现让使用 Swift 底层开发者脱离 OS C API 的折磨，获得更一致更健壮的代码体验。

■ 新的一方库

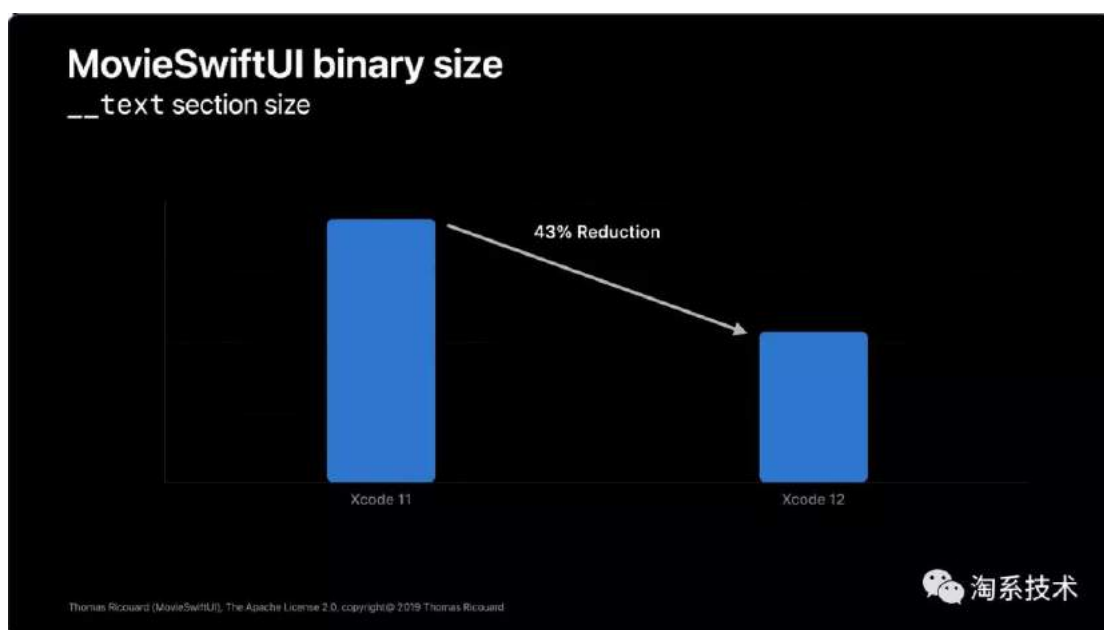
- Numerics 解决 Swift 中数学计算相关问题
- ArgumentParser 为 Swift 编写脚本提供了强力的工具
- StandardLibraryPreview 预览标准库中的新功能

■ Code Size and Runtime

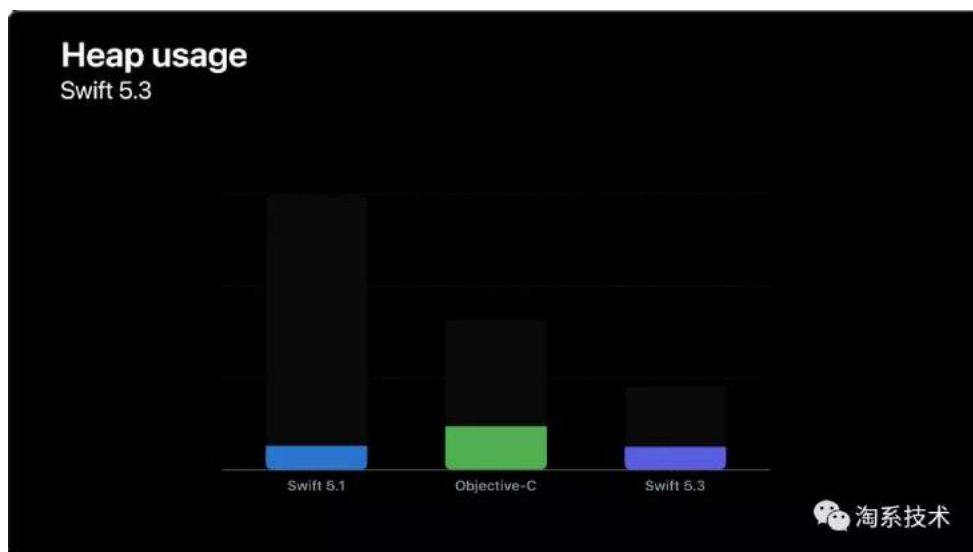
在使用 UIKit 的情况下，Swift 4.1 时生成的二进制代码已经从 OC 的两倍还多，但是在 Swift 5.3 中，这个差距已经缩小到小于 1.5 倍了。也就是说之前大量使用 Swift app 会自动得到二进制大小的优化，而对于担忧 Swift 会造成包大小问题的 App，现在已经不算是很大的问题了。因为只需要付出可以接受的代价，就能获得 Swift 带来的安全性和开发体验。



如果 App 使用了纯 SwiftUI, 二进制代码甚至可以缩小 43% 之多。可见苹果优化 Swift 的功底之深, 而且这些优化, 只需要从新编译一次即可享受, 何乐而不为。



因为 Swift 更紧凑的值类型, 运行时的内存, 分配相同的对象所需的空间自然比 OC 更小。Swift 5.3 相较 5.1, 运行时的必要额外信息存储要少非常多, 甚至做到了比 OC 还要少, 大大减小了 Swift 的运行时内存。这对低内存的设备是非常有帮助的, 同时, 更少的系统内存意味着更多的用户内存。而这一切, 只需要重新编译即可。



Swift 底层的不断优化也让其成为一门高效的语言，降低运行时的要求，就可以提升其应用的场景。

Swift Package Manager

SPM 作为 Swift 生态非常重要的一环，也迎来了更新。

1. SPM 支持二进制包分发
2. SPM 支持了资源的打包

这两点更新已经表明了 SPM 的能力已经足够完善了。目前具有一定规模 App 的内部模块都开始使用 Cocoapods 做二进制组件化的集成，这样可以明确对代码解耦，提高打包的效率。在这样的背景之下，SPM 对这两点关键特性的支持已经可以覆盖住大型 App 需求了，而且 SPM 不单单只跟 Swift 玩，C Family 它都可以支持。

在 SPM 与 Cocoapods 的对比中，亲儿子 SPM 跟 Xcode 深入整合，Xcode 可以直接打开编辑 swift package，Xcode 因为 SPM 设计了对应的操作界面，降低了开发和使用的门槛。成熟的工具链也让联调 Swift Package 轻而易举。而 Cocoapods 由社区维护，每一次 Xcode 更新其响应也不算很及时，在针对大型 App 时因为 Podfile 与 podspec 的分离导致了不一致，使用 ruby 还有一定的门槛。

现在也许是拥抱 SPM 的好时机。

■ 跨平台

现在官方支持的操作系统列表如下：

1. Apple platform
2. Ubuntu 16.04, 18.04, 20.04
3. CentOS 8
4. Amazon Linux 2
5. Windows (coming soon)

真正做到的跨平台，并且 Swift 官方支持 AWS Lambda。AWS Lambda Runtime 已经开源，支持了 AWS 的 FaaS 编程，进一步的拓宽了 Swift 涉足的领域。

调试能力(开发者体验)

调试和开发者体验也是一门语言非常重要的一环，因为没有人会写出没有错误的代码，检查错误的能力和工具对一门语言来说十分重要。苹果也十分注重这一方面，在开发者体验上下足了功夫。

■ 更加智能的诊断信息

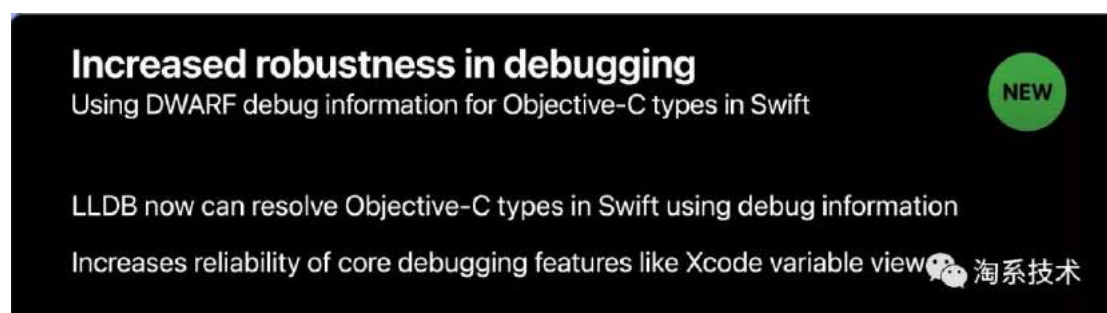
刚开始使用 Swift 的开发者可能经常会对 Xcode 的报错信息不知所云，在引入 SwiftUI 后，这个问题尤为明显。笔者第一次编写 SwiftUI 时，只要 body 中某个地方出错，报出来的错误都是不正确的，只能通过肉眼检查和推断才能明白自己的错误，十分痛苦。

现在苹果重制了诊断能力，现在 Swift 的错误诊断比之前准确了许多，错误没有乱报并且错误提示也变得很好理解，特别是 SwiftUI，很容易知道错在哪了。在 Swift 中，编译通过就是对正确性的一个很好的证明，除非你用不安全的方式让编译器闭嘴。

自动补全

经过强化的类型推断系统，也增强了 Swift 的代码补全能力。这个估计升级到 Xcode 12 就可以顺利体验了。同时代码缩进能力也得到了加强。

LLDB



积极拥抱 Swift

看了这么多年 WWDC，每次看时大家应该都有一种心态
只支持最新版本，我们才支持 iOS X（低版本），这些东西跟我没关系
感觉 Swift 真香，但是现实只让我使用 OC（叹口气）
什么语言不是用，OC 这么多年肯定够用了

危机

然而如果不及时做出改变，保持能用就行，在前进的路上，背上的担子就会越来越重。
当发现快走不动时，又回过头来看 WWDC，就会发现，原来解决方案很久以前就已经给出来了，只是当时不觉得是个问题，这不支持那不合适，但是现在想拿出来使用的时候，面对背上那一团团的乱码，却又束手无策。

做出改变是痛苦的，但是当以前觉得痒就挠挠就解决了的事情，逐渐变成现在的痛点时，要做出改变也许会更痛苦。

Swift 的出现就是为了替代并且超越 Objective-C 的语言，虽然说苹果因为历史原因还在使用 OC，但是种种迹象表明，苹果正在做积极的工作，逐渐通过 Swift 降低 OC 在整个系统的比重。

社区也在积极的转变，许多著名的第三方库都已经迁移至 Swift，OC 版本已经不再维护，例如 Lottie 已经在 Swift 版本上出现了 OC 版本不存在的特性，并且 OC 版本不再维护。这种现象慢慢会越来越多。

■ 改变

我们也在积极探索 Swift 在手淘的落地，取得了 Swift 5.1 能模块在手淘中正确运行起来的阶段性成就。

现在时机已经成熟，语言特性，SPM，工具链，标准库都已经足够强大，是时候做出改变了。

Swift 虽然看起来很简单，但其实它是一种下限低，上限高的语言，集团内部的 Swift 环境，需要大家来一起维护。我们未来也要加强 Swift 语言相关的培训，让开发者真正理解 Swift，上手 Swift，成为一名 Swifter 而不是 OSwifter。

参考

[SwiftUI 背后那些事儿](#)

[WWDC20 What's new in Swift](#)

[WWDC19 What's new in Swift](#)

[WWDC18 What's new in Swift](#)

[WWDC17 What's new in Swift](#)

[WWDC16 What's new in Swift](#)

[WWDC20 Swift packages: Resources and localization](#)

[WWDC20 What's new in SwiftUI](#)

[Swift 5 时代的机遇与挑战到底在哪里？](#)

[Swift Evolution](#)

WWDC：无线网络优化实践，带来哪些启发？

作者 | 徐杰(无宸)

出品 | 阿里巴巴新零售淘系技术

网络技术作为互联网应用赖以存在的技术基础，速度与安全永远是其核心使命，本次 WWDC 的网络类 topic 涵盖内容基本还是围绕这两个点来展开。本次 WWDC 网络类 session 在基础网络技术上譬如新协议、新算法方面着墨并不多；也未提出新的类似 NSURL Session / Network.framework 之类的新网络组件。站在应用视角，本次 WWDC 网络类 session 可分为两大类：

- 无线网络体验优化实践在系统层面的标准化；
- 本地网络应用的权限管控增强。

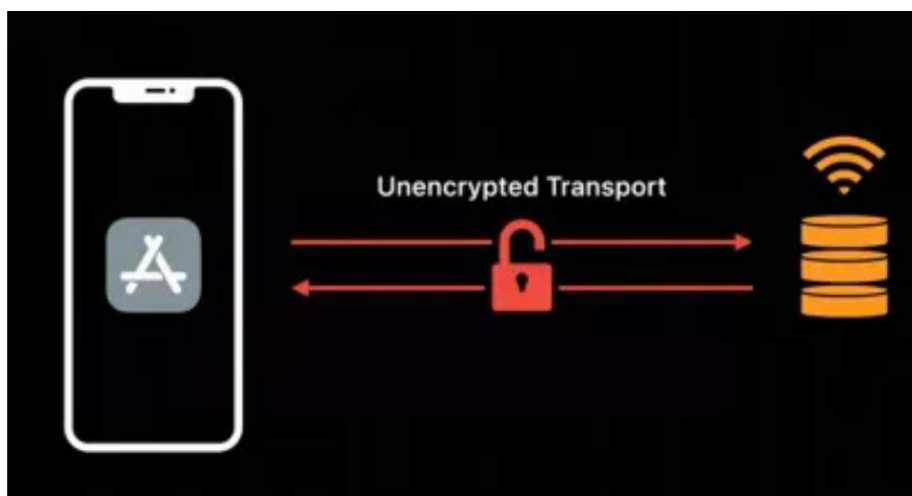
在第一类议题中，我们看到很多已经在手淘中的类似实践，或标准或自研，说明手淘在网络技术的开发与应用上还是较为深入和前沿的，基本走在全球业界前列。根据我们手淘的业务特点，笔者重点关注第一类 session，并简单探讨该新技术可以给我们带来什么样启发和变化。

使用加密 DNS

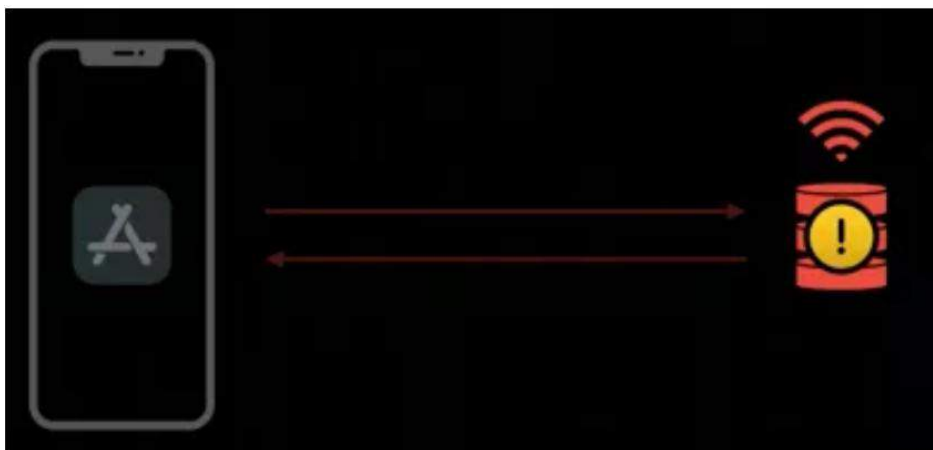
DNS 解析是网络的连接的第一步，这里提到的“加密 DNS”是什么、它解决什么问题？

■ 解决什么问题

一是传统 Local DNS 的查询与回复均基于非加密 UDP，我们常见的 DNS 劫持问题



二是 Local DNS Server 本身不可信，或者本地 Local DNS 服务不可用。



其实针对 DNS 解析过程中以上两个问题，在实践中早就有了解决方案，就是 HTTPDNS，各大云厂商也都有成熟产品售卖，那苹果这里的加密 DNS 与我们的现有 HTTPDNS 有何不同呢？

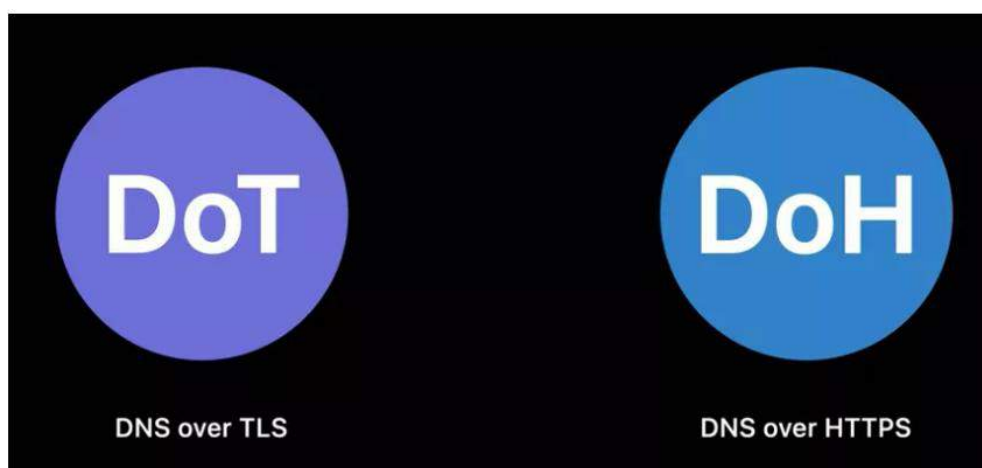
现有 HTTPDNS 有两个很大的问题：

- 一是对业务的侵入性，即如果某个网络连接需要使用 HTTPDNS 的能力，首先他需要集成服务商提供的 SDK，引入相应的 Class，然后修改网络连接的阶段的代码；
- 二是面临各种技术坑，比如 302 场景的 IP 直连处理、WebView 下 IP 直连如何处理 Cookie、以及 iOS 上的老大难的 SNI 问题等，这些都需要业务开发者付出极大的努力和尝试。

iOS 14 上的 Encrypted DNS 功能很好的解决了现有 HTTPDNS 的存在的问题。

■ 规范与标准

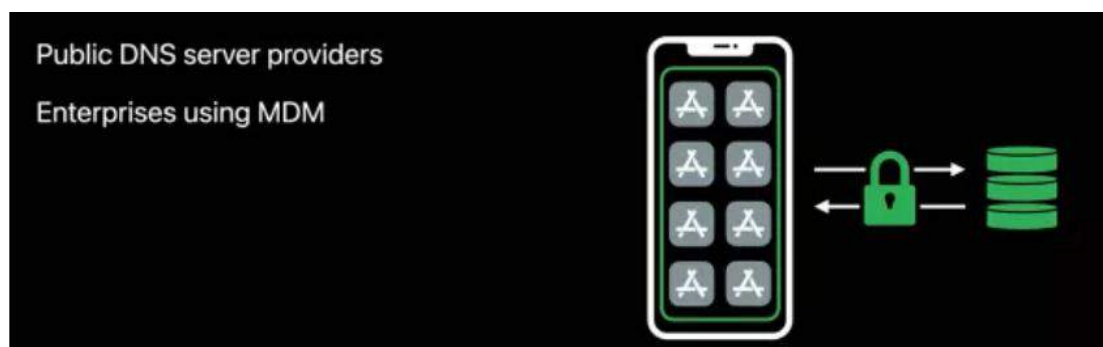
iOS 14 开始系统原生支持两种标准规范的 Encrypted DNS，分别是 DNS over TLS 与 DNS over HTTPS。



具体协议标准可以参见：rfc7858 (DoT) 、 rfc8484 (DoH)

■ 如何实现

iOS 14 提供了两种设置加密 DNS 的方法。第一种方式是选择一个 DNS 服务器作为系统全局所有 App 默认的 DNS 解析器，如果你提供的是一个公共 DNS 服务器，你可以使用 NEDNSSettingsManager API 编写一个 NetworkExtension App 完成系统全局加密 DNS 设置。或者如果你使用 MDM(Mobile Device Management)管理设备的企业设置；你可以推送一个包含 DNSSettings paload 的 profile 文件完成加密 DNS 设置。



使用 NetworkExtension 设置系统域全局 DNS 服务器示例代码：

```
// Create a DNS configuration

import NetworkExtension

NEDNSSettingsManager.shared().loadFromPreferences { loadError in
    if let loadError = loadError {
        // ...handle error...
        return
    }
    let dohSettings = NEDNSOverHTTPSSettings(servers: [ "2001:db8::2" ])
    dohSettings.serverURL = URL(string: "https://dnsserver.example.net/dns-query")
    NEDNSSettingsManager.shared().dnsSettings = dohSettings
    NEDNSSettingsManager.shared().saveToPreferences { saveError in
        if let saveError = saveError {
            // ...handle error...
            return
        }
    }
}
```

上述代码首先通过 NEDNSSettingsManager 加载配置，加载成功后，创建一个基于 DoH 协议的 NEDNSOverHTTPSSettings 实例，对其配置 DNS IP 地址和域名，DNS IP 地址是可选配置的。然后将 NEDNSOverHTTPSSettings 实例配置到 NEDNSSettingsManager 共享实例的 dnsSettings 属性，最后保存配置。

一条 DNS 配置包括 DNS 服务器地址、DoT/DoH 协议、一组网络规则。网络规则确保 DNS 设置兼容不同的网络。因为公共 DNS 服务器是无法解析本地网络的私有域名，比如只有企业 WiFi 网络内的 DNS 服务器可以解析员工访问的私有域名，这类情况就需要手动指定网络规则兼容企业 WiFi 网。

网络规则可以针对不同网络类型定义行为，比如蜂窝网、WiFi、或者具体的 WiFi SSID。在匹配的网络下，你可以禁用配置的全局 DNS 设置，或者对私有域名不使用 DNS 设置。

而在一些情况下，兼容性会自动处理。比如强制门户网络(captive portal)，手机在连接上某个 WiFi 的时候，自动弹出一个页面输入账号密码才能连接网络。这种情况下系统域全局 DNS 配置会做例外处理。相类似的，对于 VPN 网络，在 VPN 隧道内的解析将使用 VPN 的 DNS 设置，而不使用系统域 DNS 配置。

网络规则设置示例代码：

```
// Apply network rules

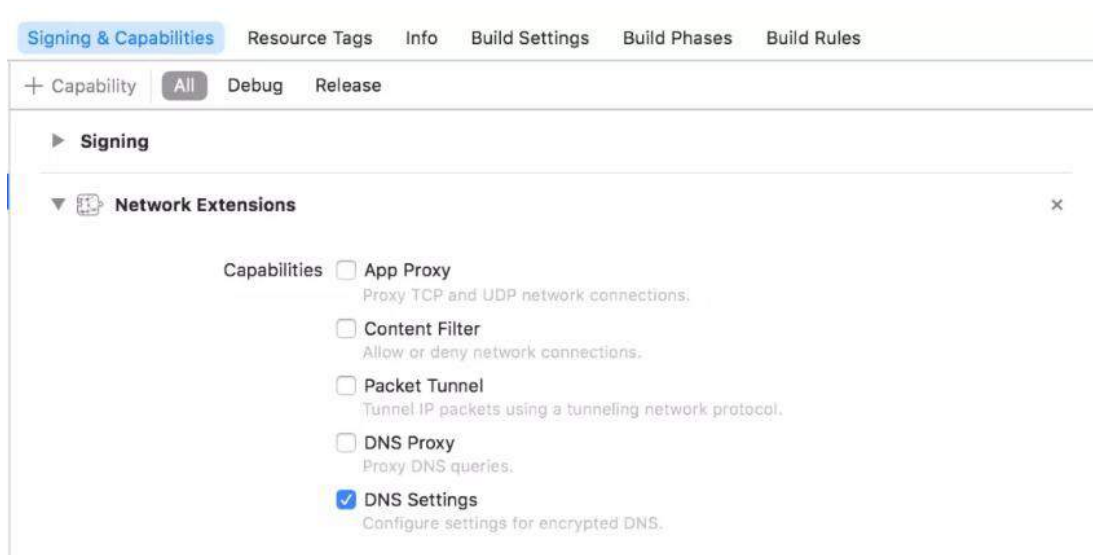
let workWiFi = NEOnDemandRuleEvaluateConnection()
workWiFi.interfaceTypeMatch = .wifi
workWiFi.ssidMatch = ["MyWorkWiFi"]
workWiFi.connectionRules =
    [ NEEvaluateConnectionRule(matchDomains: ["enterprise.example.net"],
                                andAction: .neverConnect) ]

let disableOnCell = NEOnDemandRuleDisconnect()
disableOnCell.interfaceTypeMatch = .cellular

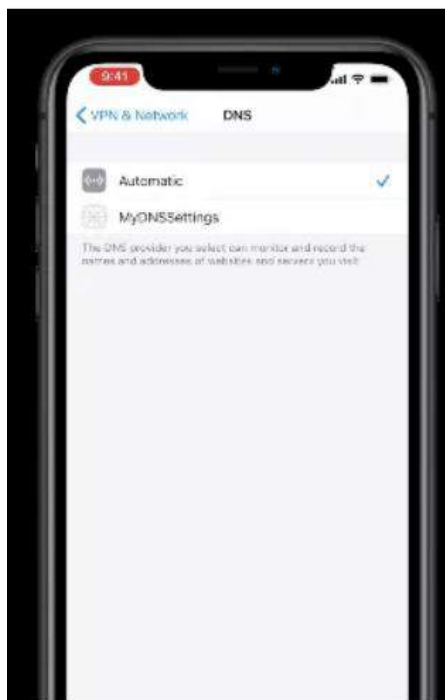
let enableByDefault = NEOnDemandRuleConnect()

NEDNSSettingsManager.shared().onDemandRules = [
    workWiFi,
    disableOnCell,
    enableByDefault
]
```

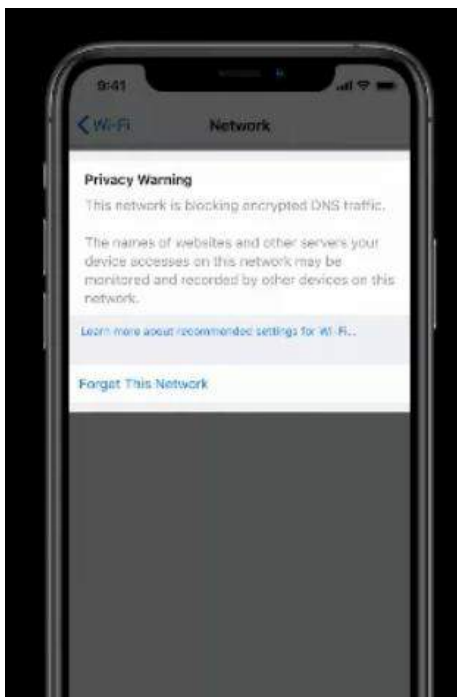
上述代码设置了三个网络规则，第一个规则表示 DNS 配置应该在 SSID="MyWorkWiFi"的 WiFi 网络生效，但对私有企业域名 enterprise.example.net 不开启。第二个规则表示规则在蜂窝网下应该被禁止使用；第三个 NEOnDemandRuleConnect 表示 DNS 配置应该默认开启；因为配置 DNS 是系统支持的，所以在编写 NetworkExtension App 时不需要实现 Extension 程序，只需要在 Network Extensions 中勾选 DNS Settings 选项。



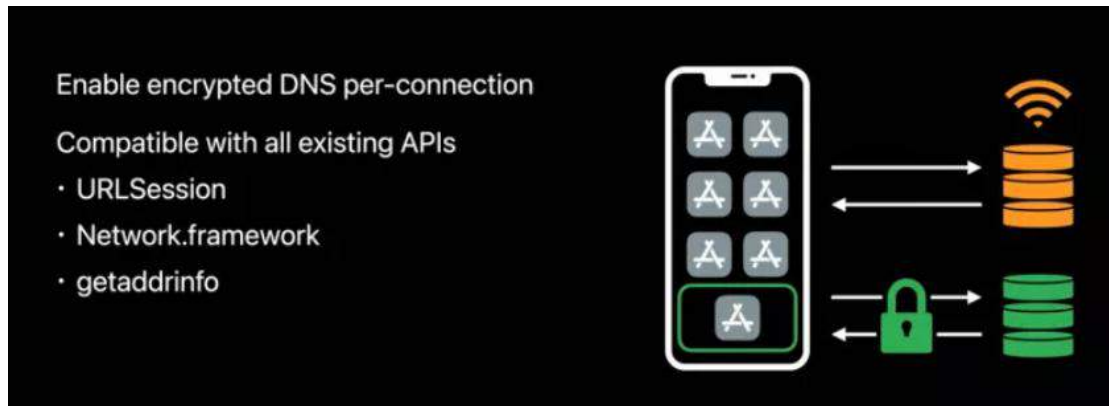
运行 NetworkExtension App，DNS 配置将会被安装到系统，为了让 DNS 配置生效，需要前往设置->通用->VPN & Network->DNS 手动启用。



一些网络可能会通过策略阻止你使用加密的 DNS 服务器。这些网络尝试分析 DNS 查询请求来过滤流量。对于此类网络，系统会被标记隐私警告提示，在该网络下的网络连接将会失败。



第二种方式是针对单个 App 的所有连接或部分连接启用加密 DNS。



如果你只想为你的 App 使用加密 DNS，而非涉及整个系统域。你可以适配 Network framework 的 `PrivacyContext`，对你的整个 App 开启加密 DNS，或者仅对某一连接开启。不管使用的是 `URLSessionTask`，或 `Network framework` 连接或 `getaddrinfo` 的 `POSIX API`，这种方式都有效。

对单个连接使用加密 DNS 示例代码：

```
// Use encrypted DNS with NWConnection

import Network

let privacyContext = NWParameters.PrivacyContext(description: "EncryptedDNS")
if let url = URL(string: "https://dnsserver.example.net/dns-query") {
    let address = NWEndpoint.hostPort(host: "2001:db8::2", port: 443)
    privacyContext.requireEncryptedNameResolution(true,
        fallbackResolver: .https(url, serverAddresses: [ address ]))
}

let tlsParams = NWParameters.tls
tlsParams.setPrivacyContext(privacyContext)

let conn = NWConnection(host: "www.apple.com", port: 443, using: tlsParams)
conn.start(queue: .main)
```

验证请求是否使用加密 DNS：

```
import Network

conn.requestEstablishmentReport(queue: .main) { report in
    if let report = report {
        for resolution in report.resolutions {
            switch resolution.dnsProtocol {
            case .https, .tls:
                print("Used encrypted DNS!")
            case .udp, .tcp:
                print("Used unencrypted DNS")
            default:
                // Ignore unknown protocols
            }
        }
    }
}
```

如果你想在 App 范围内使用加密 DNS，你可以配置默认的 PrivacyContext；App 内发起的每个 DNS 解析都会使用这个配置。不管是 URLSessionTask，还是类似 getaddrinfo 的底层 API。

```
import Network

if let url = URL(string: "https://dnsserver.example.net/dns-query") {
    let address = NWEndpoint.hostPort(host: "2001:db8::2", port: 443)
    NWParameters.PrivacyContext.default.requireEncryptedNameResolution(true,
        fallbackResolver: .https(url, serverAddresses: [ address ]))
}

let task = URLSession.shared.dataTask(with: ...)
task.resume()

getaddrinfo(...)
```

■ 现有实践对比及启发

• 与现有实践对比

上文已经提到过 HTTPDNS 产品，手淘的域名解析，采用的是类似于 HTTPDNS 原理，但更加复杂的调度方案。但是相比于这种系统层面的标准化方案，解决了现有实践中的两大难题：

- 对业务的侵入性：业务必须修改现有网络连接的阶段的代码；
- 交互的标准兼容性不足：IP 直连下的 302 问题、Cookie 问题、SNI 问题等。

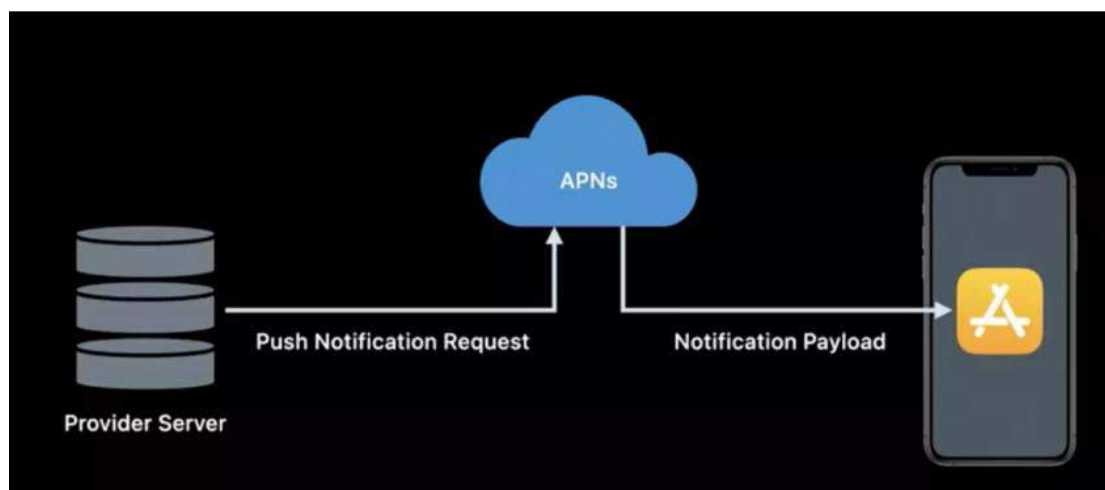
- 带来的变化

一是在应用内部,我们可以对业务透明提供提供全局的域名调度或者域名兜底解析能力,不再是只有用到特定组件或 SDK 才可以。二是苹果放开系统级别 DNS 接管后,用户设备上的服务相比原 Local DNS 的“中立性”如何管控?对特定业务是否甚至会造成恶化?如果对外部应用的这种“中立性”疑问或担心确实存在,则这种系统标准化的加密 DNS 对手淘这种大型应用则是必选项。

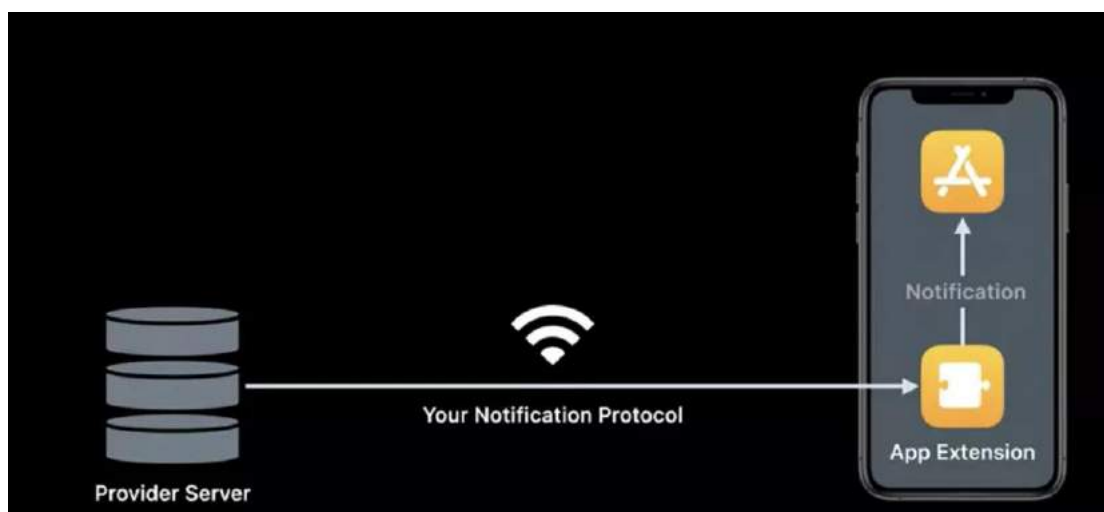
受限网络中推送

■ 解决什么问题

当向 iOS 设备推送消息时,业务服务器需要将消息先发送给 APNS 服务器,APNS 服务器再将消息转换为通知 payload 推送给目标设备。如果设备所在的 WiFi 网络没有连接互联网或者当前网络受限,比如在游艇、医院、野营地这些地方,设备没有与 APNS 服务器建立有效连接,APNS 消息投递将会失败。



对于一些 App 来说,接收推送消息是 App 的一项非常重要的功能,即使在没有互联网连接的情况下也需要持续稳定工作。为了满足这一需求,iOS 14 中增加了本地推送连接(Local Push Connectivity)API,这是 NetworkExtension 家族中新的 API。本地推送连接 API 允许开发者创建自己的推送连接服务,通过开发一个 App Extension,在指定的 WiFi 网络下可以直接与本地业务服务器通信。



上图中，App Extension 主要负责保持与本地业务服务器之间的连接，以及接收业务服务器发来的通知。因为没有了 APNS，开发者需要为业务服务器与 App Extension 定义自己的通信协议。主 App 需要配置具体在哪个 WiFi 网络下使用本地推送连接。当 App 加入到指定的 WiFi 网络，系统会拉起 App Extension，并在后台持续运行。当 App 断开与指定 WiFi 网络的连接，系统将停止 App Extension 运行。

■ 如何实现

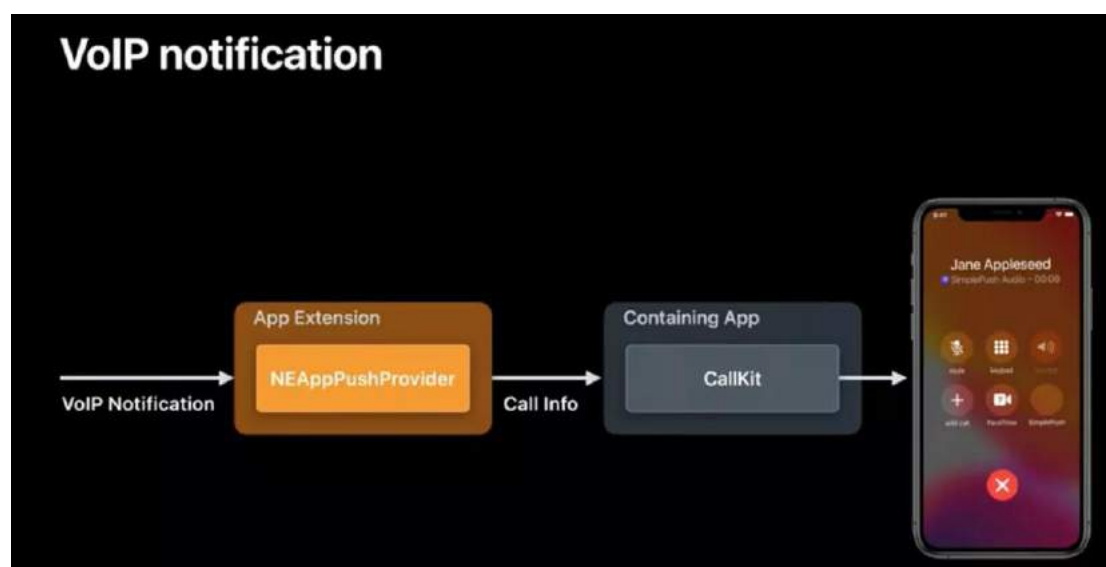
本地推送连接对那些推送功能非常重要，而设备所在网络受限的场景非常适合。而对于常规的推送需求，依然推荐使用 PushKit 或 UserNotification API 处理 APNS 推送消息。每台设备和 APNS 服务器之间只建立一条 APNS 连接，设备上所有 App 都公用这一条连接，所以 APNS 非常省电。

APNS 与本地推送连接对比：

Push Notifications	Local Push Connectivity
Requires APNs connectivity	Requires local server connectivity
Any network	Specific Wi-Fi networks
Apple APNs protocol	Custom protocol

新的本地推送连接 API 由两个类组成: NEAppPushManager 和 NEAppPushProvider。NEAppPushManager 在主 App 中使用, 主 App 使用 NEAppPushManager 创建一个配置, 配置中指定具体哪个 WiFi 网络下运行本地推送连接。你可以使用 NEAppPushManager 加载/移除/保存配置。NEAppPushProvider 则在 App Extension 使用。在 App Extension 中实现一个 NEAppPushProvider 的子类, 子类中需要覆盖生命周期管理方法, 这些方法在 App Extension 运行和停止时被调用。

App Extension 主要处理两类推送, 一类是常规的推送通知, 一类是 VoIP 呼叫通知。如果是常规的推送通知, App Extension 收到消息后, 可以使用 UserNotification API 构造一个本地推送显示推送信息。如果是 VoIP 呼叫通知, App Extension 使用 NEAppPushProvider 类将呼叫信息报告给系统。如果此时主 App 不在运行, 系统将唤醒主 App, 并将消息投递给它, 最后主 App 再使用 CallKit API 显示呼叫界面。



下面是在主 App 中使用 NEAppPushManager 的示例代码:

```
// Create Configuration

import NetworkExtension

let manager = NEAppPushManager()
manager.matchSSIDs = [ "Cruise Ship Wi-Fi", "Cruise Ship Staff Wi-Fi" ]
manager.providerBundleIdentifier = "com.myexample.SimplePush.Provider"
manager.providerConfiguration = [ "host": "cruiseship.example.com" ]
manager.isEnabled = true

manager.saveToPreferences { (error) in
    if let error = error {
        // Handle error
        return
    }
    // Report success
}
```

上述代码创建了一个 NEAppPushManager 实例, 并配置实例的各个属性值。matchSSIDs 表示在指定的 WiFi 网络下才启用本地推送连接。providerBundleIdentifier 表示 App Extension 的包名, providerConfiguration 是传给 App Extension 的一些配置, 在 App Extension 内可以通过 NEAppPushProvider 的 providerConfiguration 属性获取。isEnabled 表示使用这个配置开启本地推送连接。最后调用 saveToPreferences 方法持久化配置。下面是 App Extension 实现 NEAppPushProvider 子类的示例代码:

```
// Manage App Extension life cycle and report VoIP call

class SimplePushProvider: NEAppPushProvider {

    override func start(completionHandler: @escaping (Error?) -> Void) {
        // Connect to your provider server
        completionHandler(nil)
    }

    override func stop(with reason: NEProviderStopReason,
        completionHandler: @escaping () -> Void) {
        // Disconnect your provider server
        completionHandler()
    }

    func handleIncomingVoIPCall(callInfo: [AnyHashable : Any]) {
        reportIncomingCall(userInfo: callInfo)
    }
}
```

系统调用 start(completionHandler:) 方法启动 App Extension, 在这个方法内 App Extension 与本地业务服务器建立连接。当 App Extension 停止运行, 系统调用 stop(with:) 方法, 在这个方法内 App Extension 断开与业务服务器的连接。handleIncomingVoIPCall(callInfo:) 方法在收到 VoIP 呼叫时被调用, 在方法内 AppExtension 调用 reportIncomingCall(userInfo:) 将该事件上报给系统。随后系统将会唤醒主 App, 并将呼叫信息传递给主 App。主 App 处理系统传入的呼叫信息示例代码:

```
// Handling incoming VoIP call in the containing app

class AppDelegate: UIResponder, UIApplicationDelegate, NEAppPushDelegate {
    func application(_ application: UIApplication,
                    didFinishLaunchingWithOptions launchOptions:
                        [UIApplication.LaunchOptionsKey: Any]?) -> Bool {
        NEAppPushManager.loadAllFromPreferences { (managers, error) in
            // Handle non-nil error
            for manager in managers {
                manager.delegate = self
            }
        }
        return true
    }

    func appPushManager(_ manager: NEAppPushManager,
                        didReceiveIncomingCallWithUserInfo userInfo: [AnyHashable: Any] = [:]) {
        // Report incoming call to CallKit and let it display call UI
    }
}
```

以上代码在 AppDelegate 的 didFinishLaunchingWithOptions 方法内使用 NEAppPushManager 加载所有配置，并设置每个 NEAppPushManager 示例的代理属性。系统会在主线程中将接收到呼叫信息通过 didReceiveIncomingCallWithUserInfo 方法投递给主 App。主 App 必须通过 CallKit API 将呼入信息上报给 CallKit 展示呼叫界面。

■ 价值场景

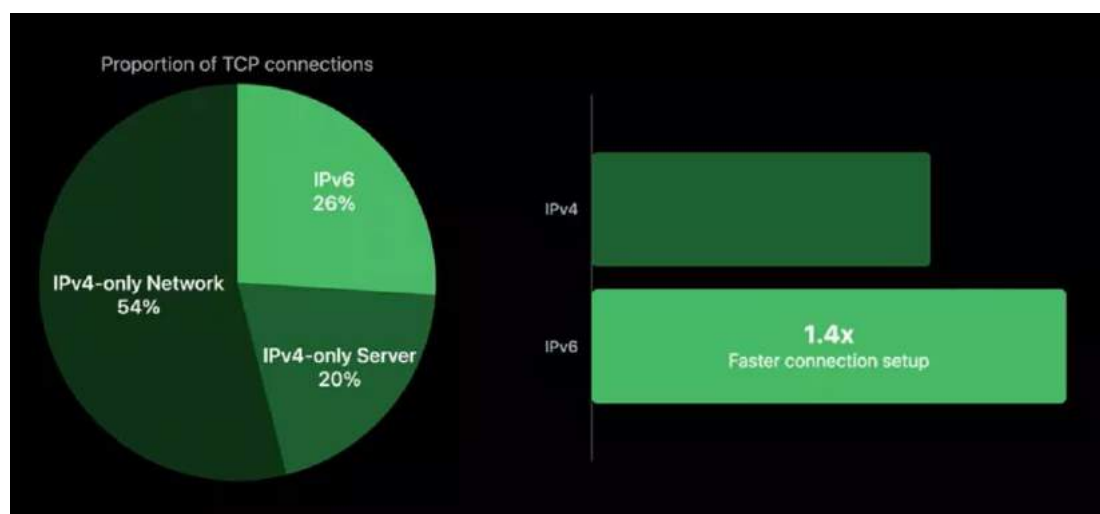
如果只考虑推送本身，对于手淘或者大部分消费类应用来说，笔者认为价值并不大，因为此类 APP 不可能在一个封闭的本地网络里去部署资源来提供服务能力。这里一个可应用的点在于：设备一旦进入特定网络环境，触发 App Extension，进而唤起主 App，应用可在后台完成一定事务。因为 iOS App 一直缺乏后台服务能力，这种特定网络环境的触发唤醒，极大的补充了这一能力。

现代网络技术的应用

苹果在这次 WWDC 中，把一些较新的网络技术，对应用的体验提升，做了一个简单综述，包括 IPv6、HTTP/2、TLS1.3、MTCP、以及 HTTP/3。这些技术在手淘基本都有涉及，有些是已经是大规模部署、有些是正在逐步推进中。对各个应用来说，如果已经在应用这些技术了，则云端均尽可能标准化，便于进一步推广和复用。这里简单的对苹果的综述做一个搬运：

IPv6

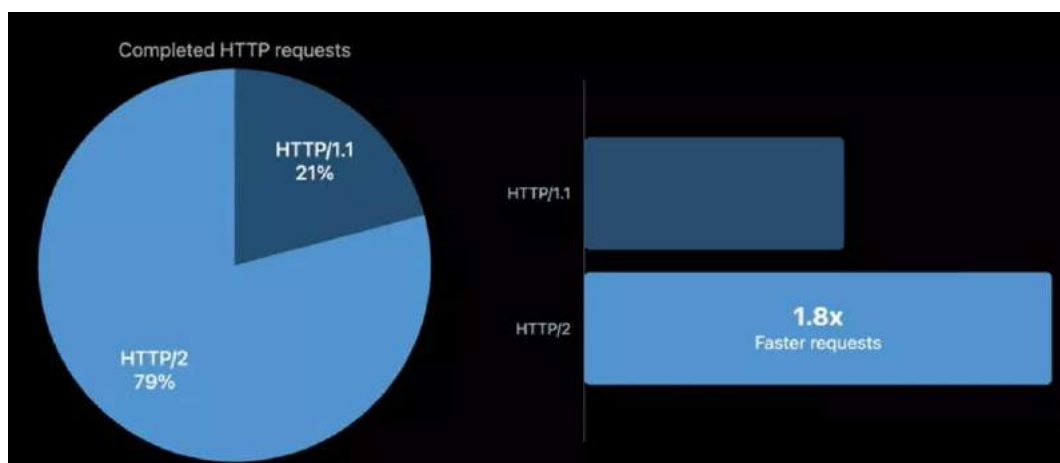
苹果根据最新统计,苹果全球设备 TCP 连接占比中,IPv6 占比 26%,IPv4 占比 74%,其中 74%的占比中有 20%是因为服务端没有开启 IPv6 支持。在建连时间方面,由于减少了 NAT 使用,提高了路由效率,IPv6 的建连时间比 IPv4 快 1.4 倍。开发者只需使用 `URLSession` 和 `Network.framework` API, IPv6 网络适配将自动支持。



以上是苹果的数据。阿里巴巴集团从 18 年开始大力推进 IPv6 的建设,目前我们在 IPv6 整体应用规模上在业界是属于头部大厂。但根据我们应用的实际效果数据,以及业界友商的应用数据,性能提升并不明显。以及工信部的 IPv6 建设目标来看,性能提升也不是 IPv6 建设的目标,只要达到 IPv4 同等水平即可。IPv6 的意义就在解决 IPv4 地址空间枯竭的问题,更多的在规模、安全,而不是性能提升。就企业而言,例如可以降低 IPv4 地址的购买费用;就国家而言,IPv6 突破了 IPv4 中国境内无 DNS 根结点的风险。IPv6 目前阶段在国内尚处于发展中,基础运营商覆盖、家用网络接入设备支持、应用服务的支持,正在快速发展中。

HTTP/2

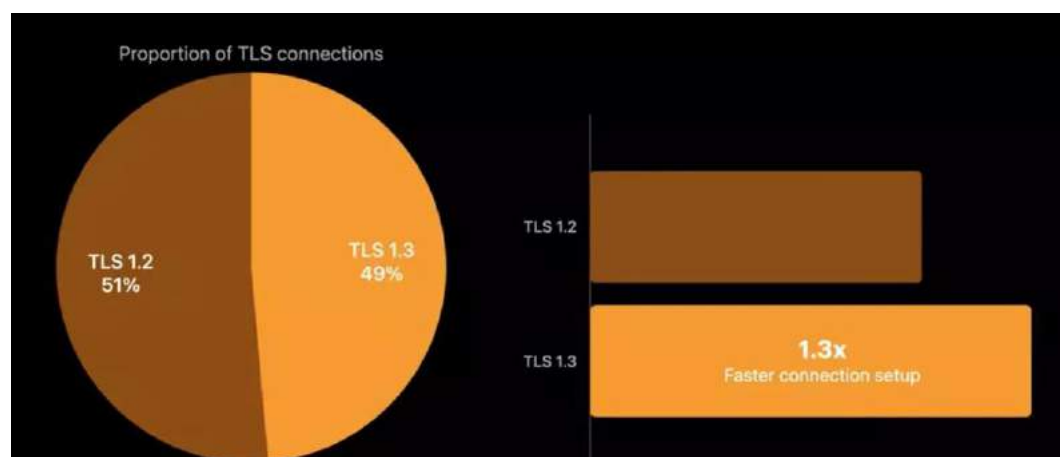
HTTP/2 的多路复用特性使得对同一服务器的多个请求复用到单个连接上,不必等待前一个请求响应结束才能发送下一个请求,不仅节省了时间也提升了性能。头部压缩特性提升了带宽利用率,通过简化消息内容,从而降低消息的大小。根据最新统计,在 Safari 中 HTTP2 Web 流量占比 79%,HTTP/2 比 HTTP/1.1 快 1.8 倍。如果服务端支持 HTTP/2, `URLSession` 将默认使用 HTTP/2。



在手淘业务中, 全面应用 HTTP2 已经有三四年之久, 其使用规模比苹果统计的结果要高的多, 取得了巨大的体验提升收获。手淘的核心流量分为两类: API 请求 MTOP 于图片 CDN 请求, 这两类的 HTTP2 的流量占比约超过 98%, 基本实现核心流量全部长连化。但当前手淘的 HTTP2 技术在设备支持之前即大规模应用, 协议栈完全自研, 为进一步提升建联速度, 又做了一些预置证书的优化。下一步将逐步使基础网络依赖标准化平台能力, 降低对私有协议栈的依赖, 可降低包大小以及提升产品复用体验。

■ TLS1.3

TLS1.3 通过减少一次握手减少了建连时间, 通过形式化验证(Formal Verification)与减少被错误配置的可能性, 提高了通信安全。从 iOS 13.4 开始, TLS1.3 默认在 URLSession 和 Network.framework 开启。根据最新统计, 在最新的 iOS 系统上, 大约 49% 的连接使用 TLS1.3。使用 TLS1.3 比使用 TLS1.2 建连时间快 1.3 倍。如果服务端支持 TLS1.3, URLSession 将默认使用 TLS1.3。



目前手淘的 HTTP/2 的 TLS version 仍然还是基于 TLS1.2，不过为了解决低版本 TLS 的性能之殇，手淘网络通过预置证书与自定义 SSL 握手流程，创新自研了 slight-ssl 协议，实现了 0rtt 的效果。TLS 1.3 标准在系统层面的全面支持，对应用来说是一个明确的技术趋势信号，我们的网络协议需尽快标准化，对网络管道两端来说，客户端应用层网络接口需全面升级到 NSURLSession 或 Network.framework，实现对系统标准能力的应用；云端也许全面支持 TLS1.3，可不依赖端侧 SDK 即可提供更新安全、高效、标准的网连接。

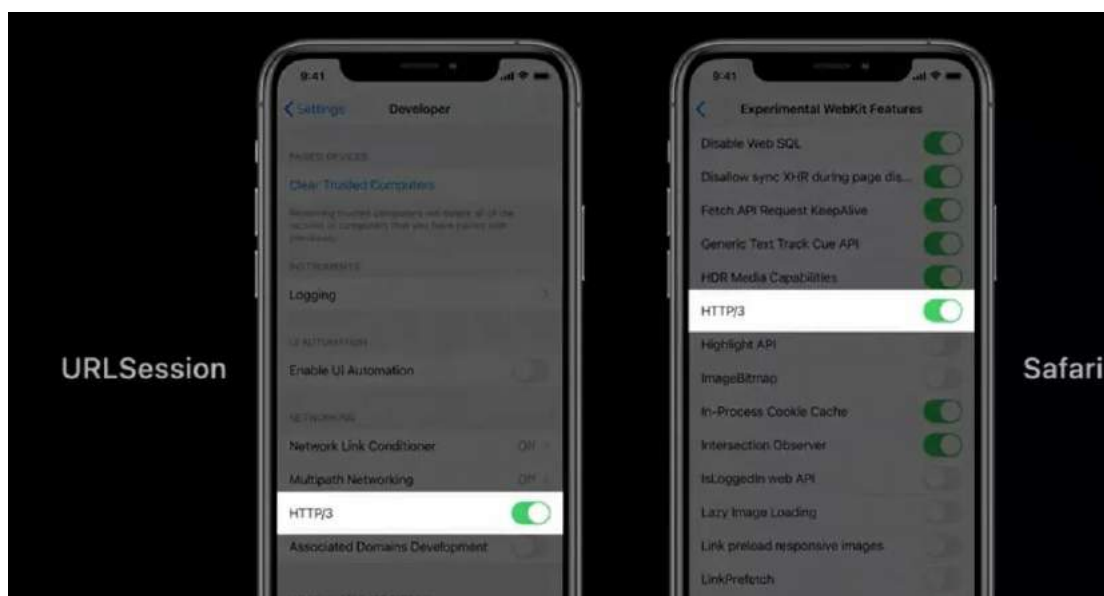
■ MultiPath TCP

MultiPath TCP 允许在一条 TCP 链路中建立多个子通道。当设备切换网络时，单个 TCP 连接依然可以继续使用。苹果的 Apple Music 与 Siri 服务都使用了 MultiPath TCP。Apple Music 在使用 MultiPath TCP 之后，音乐播放卡顿次数减少了 13%，卡顿持续时间减少了 22%。开启 MultiPath TCP 需要客户端和服务端都支持才能生效，服务端支持 MultiPath TCP 可参考：<http://multipath-tcp.org/>

其实手淘在这方面也有类似的优化尝试：多网卡：同时通过 Wi-Fi 与蜂窝网连接目标服务器，提升数据传输速度。其技术原理与 MTCP 不一样，但也是想在上层起到类似作用：通过多路连接，提升数据交换带宽。业界也有类似的产品，例如华为的 LinkTurbo。

■ HTTP/3

HTTP/3 是下一代 HTTP 协议，它是构建在新的 QUIC 传输协议之上，QUIC 协议内建了对 TLS1.3 的支持，并提供了与 HTTP/2 一样的多路复用功能，并进一步减少了队头阻塞的发生，让单个请求或相应的丢失不影响到其他请求。使用 QUIC 的 HTTP/3 还具有较高的保真度信息，以提供改进的拥塞控制和丢包恢复。同时也包括内建的移动性支持，这样网络切换不会导致正在进行的操作失败，可以无缝在不同网络之间切换。不过 HTTP/3 目前还处于草案阶段，iOS 14 和 MacOS Big Sur 包括了一个对使用 URLSession 的 HTTP/3 的实验预览支持，这个功能可以在开发者设置中开启。同时 Safari 的 HTTP/3 支持也可在开发者设置中开启。



在手淘中，我们的 QUIC 应用应该会早于苹果系统先行支持，目前已经在灰度中。

| 附录

淘系技术部——客户端团队， 正在进行社招招聘， 岗位有 iOS Android 客户端开发工程师、欢迎大家加入我们。

简历投递：yutang.pyt@alibaba-inc.com



阿里云开发者“藏经阁”
海量免费电子书下载



关注「淘系技术」微信公众号
商业&技术&经验，全面分享
和 20W+程序员共同成长！