WWDC17
内参

作者／SwiftOldDriver

目录

博客：http://www.jianshu.com/u/839a8d8aa275

- What's New in LLVM

  作者：AloneMonkey

  博客：www.blogfshare.com

- Introducing Core NFC

  作者：WAMaker，丁香园初级 iOS 工程师

- 用自然语言处理优化你的 App

  作者：WAMaker

- 使用 Xcode 运行时工具帮助排查 Bug

  作者：PMST

  博客： http://www.jianshu.com/u/596f2ba91ce9

- 在 WKWebView 中自定义加载内容

  作者：@没故事的卓同学

  博客： http://www.jianshu.com/u/88a056103c02

- 使用 CloudKit 控制台构建更好的应用

  作者：sing_crystal

  博客：http://www.jianshu.com/u/7a2d2cc38444

- App 与不断发展的网络安全标准

  作者：@Joy_xx，网易 iOS 工程师

  博客：http://www.jianshu.com/u/9c51a213b02e

- iOS 11 里 App 终于可以密码自动填充了

  作者：四娘 @kemchenj

- iOS 11 定位技术中的一些新特性

  作者：tingxins

  博客： https://tingxins.com

- iOS 11 中 StoreKit 的改进

  作者：bestswifter

  博客： https://bestswifter.com

- 精通 UIKit UIGestureRecognizer System

  作者：Mr. Stein

  博客： www.shilei365.com

- Photos APIs 新特性

  作者：@JonyFang，iOS 工程师，产品设计爱好者

  博客： http://www.jonyfang.com/

前言

本书的内容主要介绍 WWDC 17 上提到的和 iOS 开发相关的开发技术，故**不包含 tvOS、watchOS、macOS** 等相关 session 的内容。

全书内容预计9月完成。

感谢泊学网和梁杰领导的 SwiftGG 对本书的支持。

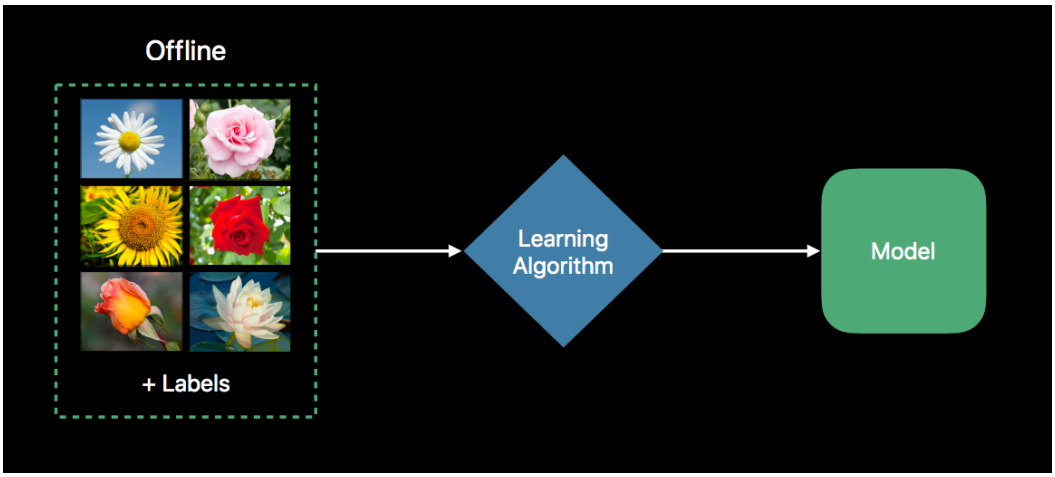
再一次，感谢所有购买这本书的读者，谢谢你们的付费支持。

对本书有问题可以微博私信我 @没故事的卓同学 或者发邮件 lacklock@gmail.com。

初探 iOS 上的人工智能：Core ML

Machine Learning 基本介绍

机器学习是一门人工智能的科学。它通过对经验、数据进行分析，来改进现有的计算机算法，优化现有的程序性能。其基本流程如下图：

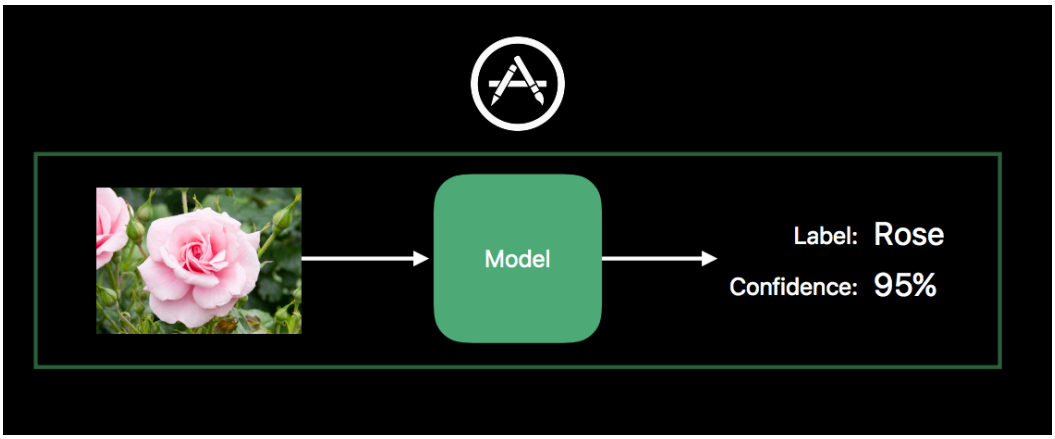


如图，机器学习有三个要素：

- 数据（Data）
- 学习算法（Learning Algorith）
- 模型（Model）

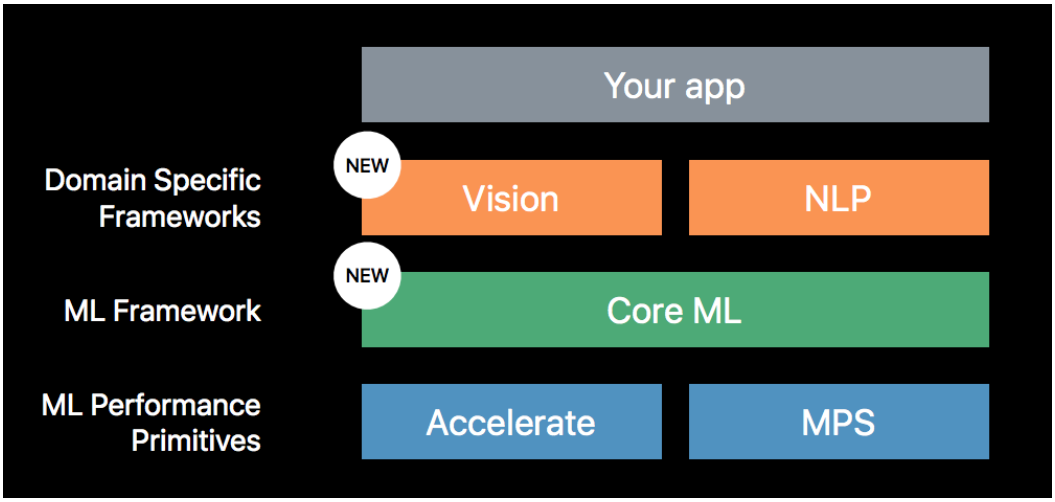
以图片分析 App 为例，这个场景下的数据、学习算法和模型分别对应：

- **数据**：各种花的图片。这些数据称为此次机器学习的**样本（Sample）**。每张图片中包含的信息，比如形状、色彩，称之为**特征（Feature）**。每张图片对应的花朵，比如梅花、玫瑰，称之为**标签（Label）**。这些图片中蕴含着某些规律，而我们人类正是依靠这些规律判断出图片对应的花朵（标签）。现在我们希望机器能够把这个规律挖掘出来。这样在面对新的**样本**时，机器能根据其**特征**，准确判断其对应**标签**，也就是判断图片对应的花朵。
- **学习算法**：机器学习的算法有很多，比如神经网络、逻辑回归、随机森林等等。作为 iOS 工程师我们无需深刻理解这些算法，只要知道输入数据后执行“学习算法”就可以得到模型。
- **模型**：即机器从样本数据中挖掘出来的规律。有了模型之后，面对新的数据，模型就可以做出相应判断，例如下图图片是郁金香还是玫瑰。

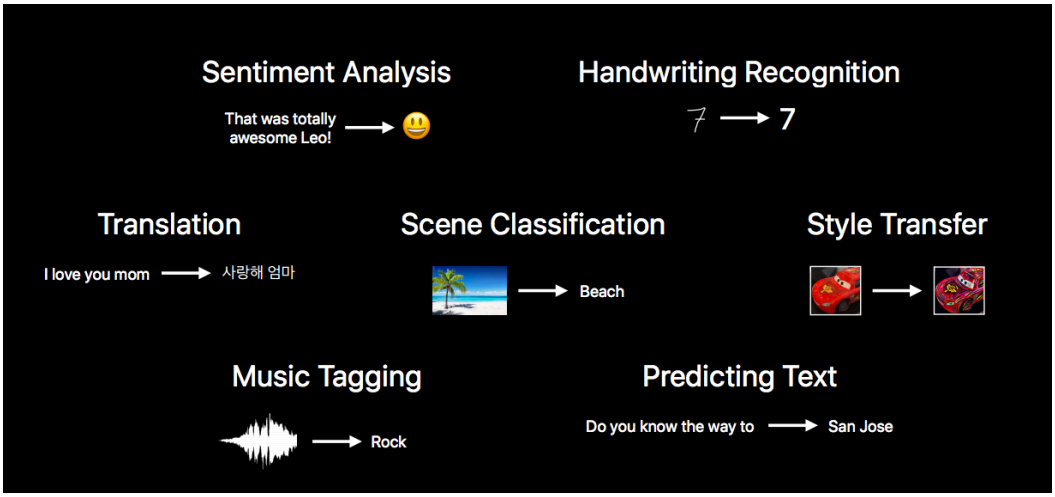


Core ML 到底是什么

Core ML 是苹果为其开发者准备的机器学习框架。支持 iOS, MacOS，tvOS，和 watchOS。它由4部分组成，其结构如下图：



- 最底层性能框架层：由 Acccelerate 和 Metal Performance Shaders 两个框架组成。前者用于图形学和数学上的大规模计算，后者用于优化与加速 GPU 和图形渲染。
- Core ML：支持导入机器学习的模型，并生成对应高级代码（Swift, Objective-C）的框架。
- 专用机器学习框架：基于 Core ML，针对特殊场景进行封装和优化的框架。如上图所示，**Vision** 框架和 **NLP** 框架都是 iOS 11 最新添加：前者用于图片分析，例如人脸识别；后者用于自然语义分析，例如上下文理解。原来 iOS 中的 **GamePlayKit** 框架也加入 Core ML，用于决策树（Decision Tree）模型的学习。
- 应用层：使用了这些框架之后构建起来的 App。应用场景（如下图）十分广泛：人脸识别，手写文字理解，类型转化，文字情感分析，自动翻译，等等。



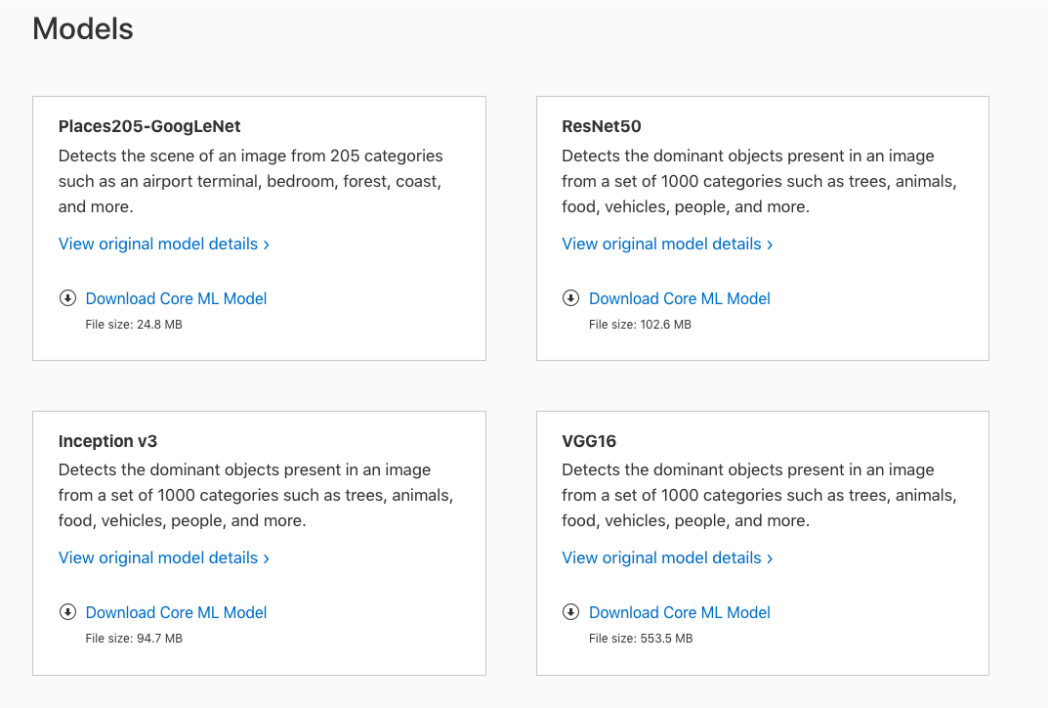
应用和代码详解

主要操作过程分三步：

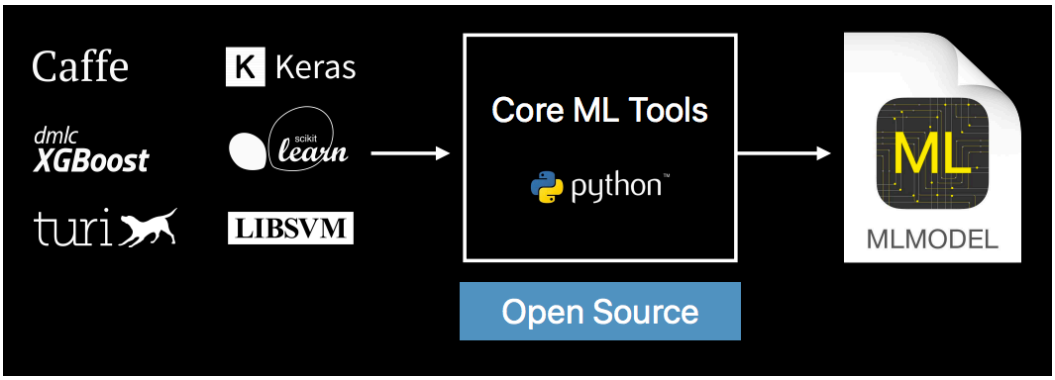
1. 得到 Core ML 的模型
2. 将模型导入项目中
3. 用生成的 Swift 接口进行编程

得到 Core ML 的模型

在苹果官网上你可以直接拿到训练好的模型。目前有4个：



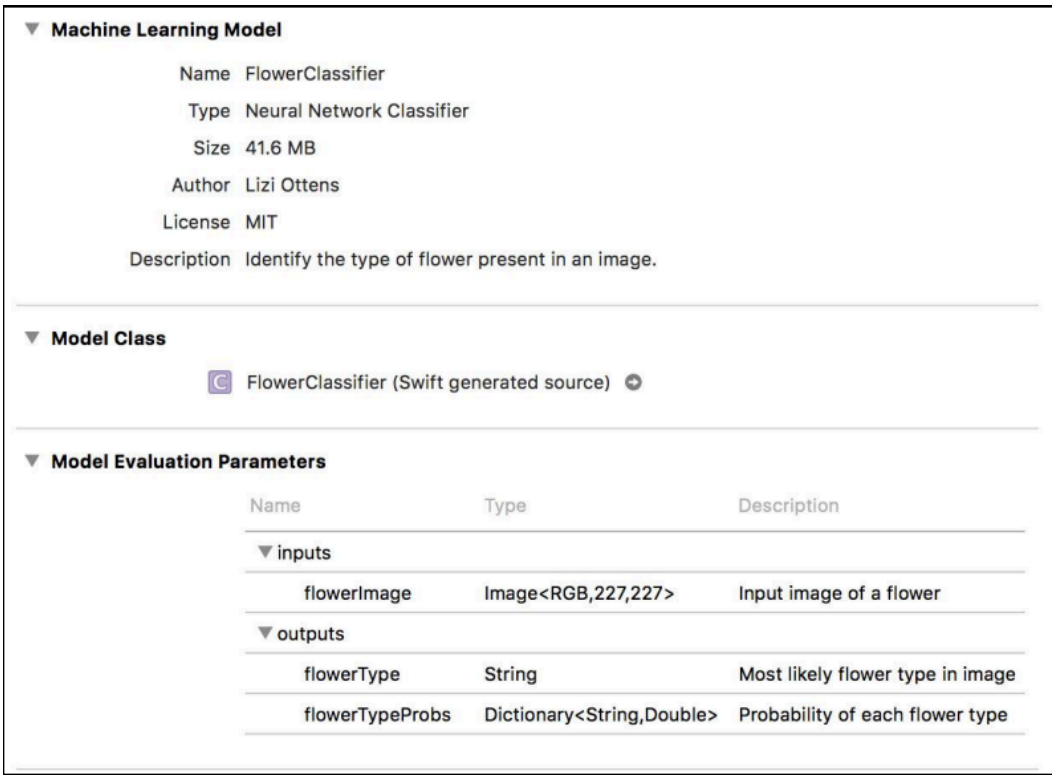
你也可以自己训练模型，或到第三方社区去下载别人训练好的模型。苹果提供了转换器（基于 Python，开源）用于把其它机器学习框架训练出来的模型转化为 Core ML 适配的模型。如下图所示：



如何使用 Core ML Tools 进行转化的原理和演示详见《Core ML in Depth》文章。

将模型导入项目中

直接拖拽即可。记得右侧 Target Membership 选项勾选自己的 App。你可以点击 mlmodel 文件观看它的属性。大小（Size）是 App 性能的一个重要指标，输入（Input）输出（Output）决定了如何使用这个模型。下图中输入是花朵图片，为 image 类型。输出有两个值，一个是最有可能的花朵，为 String 类型；另一个是所有可能的花朵类型即其对应的可能性，为 String 对应 Dobule 的 Dictionary 类型。



用生成的 Swift 接口进行编程

生成的接口如下：

```
// 输入
class FlowerClassifierInput {
    // 花朵图片
    var flowerImage: CVPixelBuffer
}

// 输出
class FlowerClassifierOutput {
    // 最有可能的花朵类型
    let flowerType: String

    // 所有可能的花朵类型即其对应的可能性
    let flowerTypeProbs: [String: Dobule]
}

// 模型
class FlowerClassifier {
    convenience init()

    // 通过输入产生输出
    func prediction(flowerImage: CVPixelBuffer) throws ->
FlowerClassifierOutput
}
```

在实际代码中调用接口进行应用：

```
// 调用 model
let flowerModel = FlowerClassifier()

// 利用 flower model 对输入进行分析
if let prediction = try? flowerModel.prediction(flowerImage: image) {
    // 得到分析的结构
    return prediction.flowerType
}
```

总结

借助 Core ML，苹果将复杂的学习算法和模型训练从机器学习中剥离出来，开发者无需理解其背后深奥的逻辑和计算，只需直接调用模型，在本地实时、安全的运用即可。同时为了兼顾扩展性，苹果对其他第三方机器学习框架和模型提供了 Core ML 转换接口，欲知原理如何，请看《再探 iOS 上的人工智能：Core ML》一文。

参考

视频地址：Introducing Core ML

ppt地址：Introducing Core ML

再探 iOS 上的人工智能：Core ML

总览

本文主要介绍 Core ML 基本知识，应用场景，实例展示，以及其转换工具的使用。

Core ML 回顾

正如《初探 iOS 上的人工智能：Core ML》文中所说，Core ML 是苹果推出的、使开发者可以使用机器学习模型进行开发的框架。它支持 MacOS，iOS，watchOS，tvOS 全平台。开发过程分为以下三步：

1. 通过其他平台或框架得到机器学习模型
2. 将模型导入 Xcode 中，Xcode 自动生成对应的 Swift 接口
3. 使用 Swift 接口进行编程

使用场景为：

- 情感分析
- 物体识别
- 个性化定制
- 类型转换
- 音乐标签
- 手势识别
- 自然语义识别

它支持的数据结构有以下几类：

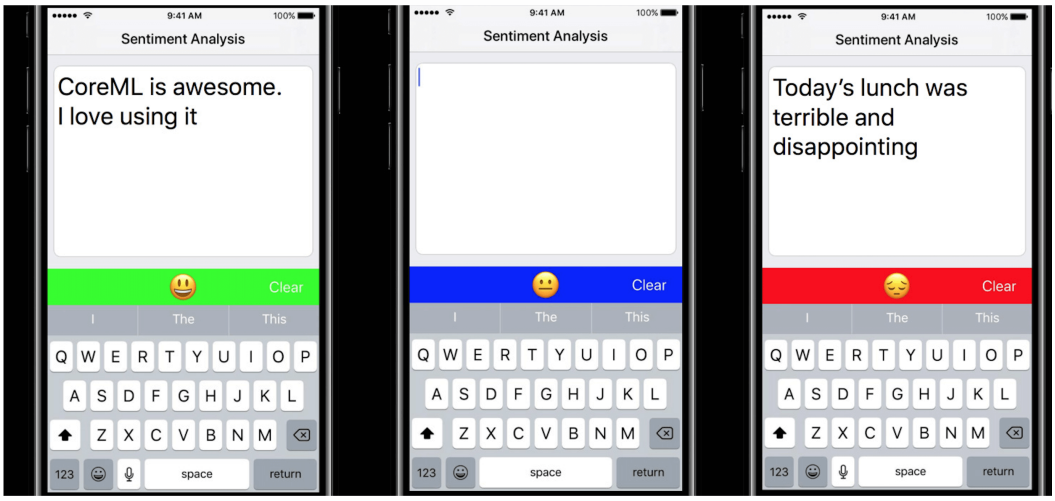
- 数值：浮点型（Dobule），整型（Int64）
- 类别：字符串（String），整型（Int64）
- 图片：像素缓冲（CVPixelBuffer）
- 数组：多维数组（MLMultiArray）
- 字典：字符串-浮点型（[String : Double]）,整型-浮点型（[Int64 : Double]）

下面展示两个示例应用：文字情感分析和实时物体识别。

示例 App

文字情感分析

所谓文字情感分析，就是根据输入文字内容，机器自动判定文字表达的是正面还是负面情绪。苹果示例的 App 如下：



当什么文字都没有的时候，App 反应的是正常情感；当文字内容为开心或赞美时，App 会变为笑脸；当文字内容为伤心或抱怨时，App 会变为苦瓜脸。整个过程为即时反映，几乎无延迟。

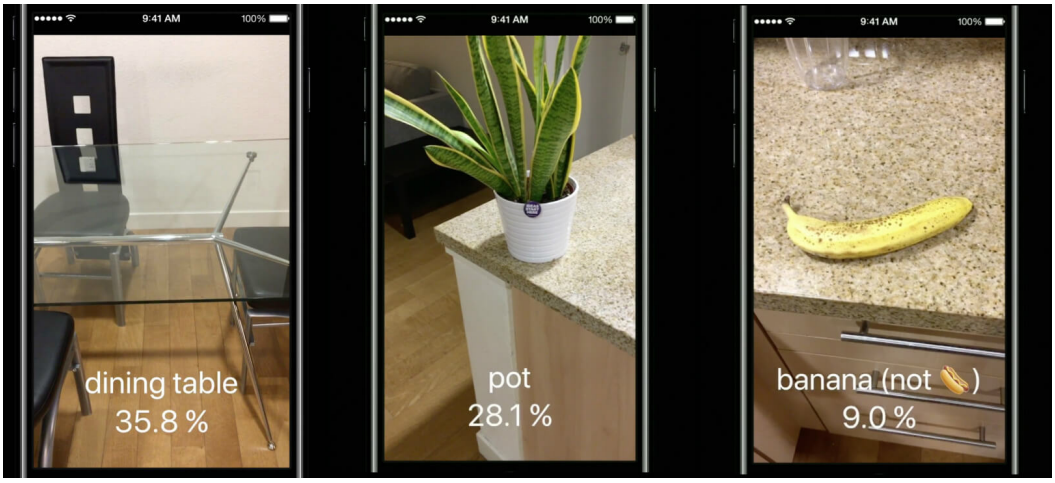
整个 App 的原理可以分为下四步

1. 用自然语义处理API统计输入文字的单词频率
2. 将单词频率输入到 Core ML 的模型中
3. Core ML 模型根据单词频率判断内容为正面或负面情绪
4. 根据情绪内容更新 UI

实时物体识别

文字情感分析 App 使用的是自然语义处理（**NLP**）的框架，而实时物体识别使用的是视觉（**Vision**）框架，他们两是基于 Core ML 的最核心的两大系统框架。

所谓实时物体识别，是指 iPhone 可以自动识别出摄像头所对准的物品，如下图所示。



当镜头对准餐桌时，机器自动识别出餐桌，并给出相似概率。类似 Google IO 大会上展示的 Google Lens。其原理是将所捕捉到的物体传给 Core ML 的模型，让模型实时推测出物品并更新 UI。具体的原理和代码参见《初探 iOS 上的人工智能：Core ML》，这里不再赘述。

Core ML 的支持和工具

现在 iOS 上可以运行的Core ML 的模型可以从苹果官网上下载，也可以通过第三方社区下载，Caffee，Keras，LIBSVM，scikit-learn，xgboot等开源机器学习框架训练出的模型皆可无缝转换为 Core ML 对应的模型。

同时，苹果开源并展示了如何使用它们自己的 Core ML 模型转换工具，它是用 Python 写成。你可以定制转换器和转换模型的参数。下面我们就来看看使用 Core ML 模型转换工具的步骤。

- **安装 Core ML 工具**

直接在终端中输入以下命令：

```
pip install -U coremltools
```

如果 Mac 上没有安装 Python，请先用命令行安装：

```
brew install python
```

- **转换训练好的模型**

假如你的模型是用 caffe 训练的，即现在有一个 .caffemodel 文件，以下步骤可以将其转化为苹果支持的 .mlmodel：

```
// 导入 core ml 工具库
import coremltools

// 利用 core ml 中对应的 caffee 转化器处理 .caffemodel 模型
coreml_model = coremltools.converters.caffe.convert('XXX.caffemodel')

// 将转化好的模型存储为 .mlmodel 文件
coreml_model.save('XXX.mlmodel')
```

确定转化好的模型能否工作(比如检测现有模型能否识别一张玫瑰图片)，可以直接运行：

```
myTestData = Image.open('rose.jpg')

XXX.mlmodel.predict('data': myTestData)
```

如果能正确输出结果（预测结果应含有 rose，其预测可能性较高），那么证明模型转换没有问题。

- **定制化转化的模型**

定制转化模型的参数，我们一般用 label.txt 文件来定义，直接传入转化中即可。

```
// 自定义模型的接口参数
labels = 'labels.txt'

// 将 labels 设为转换的模型参数
coreml_model = coremltools.converters.caffe.convert('XXX.caffemodel',
class_labels='labels')
```

定制转化的输入数据 data 为 image 类型：

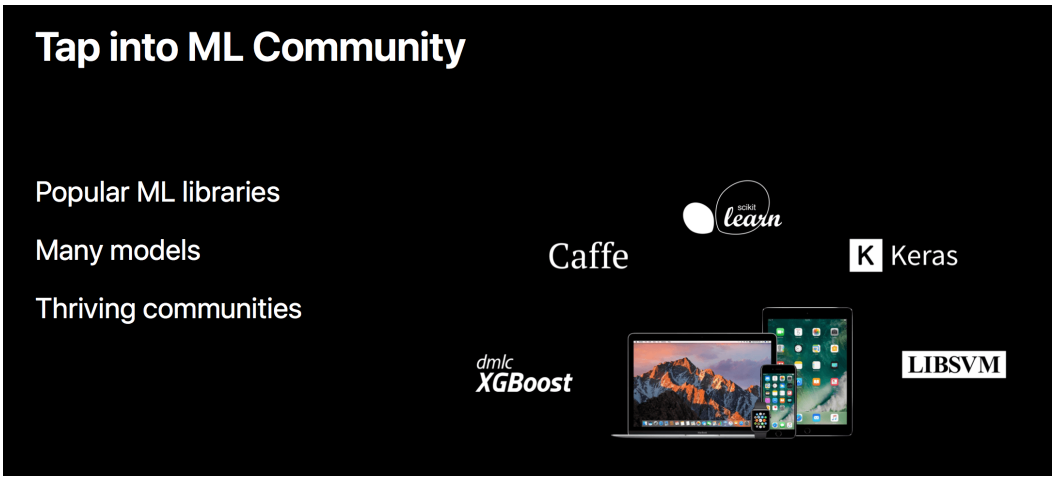
```
coreml_model = coremltools.converters.caffe.convert('XXX.caffemodel',
class_labels='labels', image_input_name = 'data')
```

指定转换模型的描述型参数（metadata）：

```
// 指定作者信息
coreml_model.author = 'Apple Papa'

// 指定许可证
coreml_model.license = 'MIT'

// 指定输入（'data'）描述
coreml_model.input_description['data'] = 'An image of flower'
```

其他参数也类似设置。

目前苹果因为版权问题，对于 Google 的 TensorFlow 和 Amazon 的 MXNet 两大框架支持不够，但是这两个框架训练出来的模型转换成 iOS 上对应的 Core ML 模型难度并不大，除了用苹果自己的转换工具外，网上也有很多开源工具可以做到。



参考

- 视频地址：Core ML in depth
- ppt地址：Core ML in depth

Introducing Drag and Drop

> 现在你可以在 iPad 进行拖放操作，将文字、图片和文件从一个 App 移动到另一个。这一功能针对 iPad 多点触控大显示屏量身设计，整个操作如同魔术。你可以轻触并移动处于屏幕上各个位置的几乎任何内容，甚至还能同时移动多个项目。

Drag and Drop 是什么

Drag and Drop 是一种 App 的数据交互方式，这种方式依赖于用户的拖拽手势。使用 Drag and Drop 可以实现 App 中的文本、图片等资源文件快速而形象地剪切和拷贝操作。

在官方文档中，Overview 对 Drag and Drop 共有四个短句介绍：

- 易用性（Easy to use）：Drag and Drop 利用 Multiple-Touch 特性，让用户在使用 iPad 端的 App 时以自然的方式交互数据。只需要点击并悬停在你想要的图像、文本或文件，然后拖拽至你希望的位置。在原生应用中，你可以拖拽联系人信息、事件提醒、地图的注释信息等等。
- Spring-loading：笔者不知道该如何翻译这个名字。但是很直观的感受就是和“弹性”和“载入”有关系。确实是这样，你可以在拖拽过程中把手指移动到按钮图标上，之后会产生类似点击的效果。这个效果解决了单手操作的点击事件问题。
- 多选特性（Multi-select）：iOS 11 引入了一个快捷方法来选取多个内容。当一个内容处于拖动状态时，可以用其他手指点击其他内容即可快速添加选中内容。
- 系统整合（Systemwide Integration）：Drag and Drop 已经整合到 iOS 中，你可以在这些位置使用：主屏幕、Dock、提醒、日历、消息、Spotlight、文件、Safari、联系人、iBooks、新闻、Notes、照片、地图、Keynote、Pages 以及 Numbers。并且提供了强大的 Drag and Drop API，可以在自己的 App 中实现。

Drag and Drop 的目标

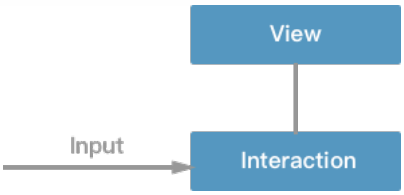
- The interface is live - 保持界面的实时性
- Deep integration with all of iOS - iOS 统一整合拖拽功能
- Great visual feedback - 更直观的视觉感受
- Hover to navigate - 悬停导航
- Items can be added - 可添加多内容
- Transfer drags between fingers - 手指间拖拽合作
- Multiple drag interactions - 多重拖拽交互合作

Drag and Drop 是 Multiple-Touch 技术的又一次实践。在这之前，MT 技术主要运用在游戏中，因为在日常工作的操作中几乎一个手指就可以完成。在 App 中的数据交互方面，传统的方法就是选中 → 长按复制 → 切换 App → 粘贴。这样的操作习惯就是直观且琐碎，步骤繁杂（这也是老罗托槽的地方）。

说起老罗发布的 One Step 技术，Apple 的 Drag and Drop 主要是在 iPad 端，而手机端在 WWDC 中并没有过多的演示。因为在 iOS 端拖拽显得过于鸡肋，屏幕太小拖拽的体验并没有体现出来。

使用 Drag

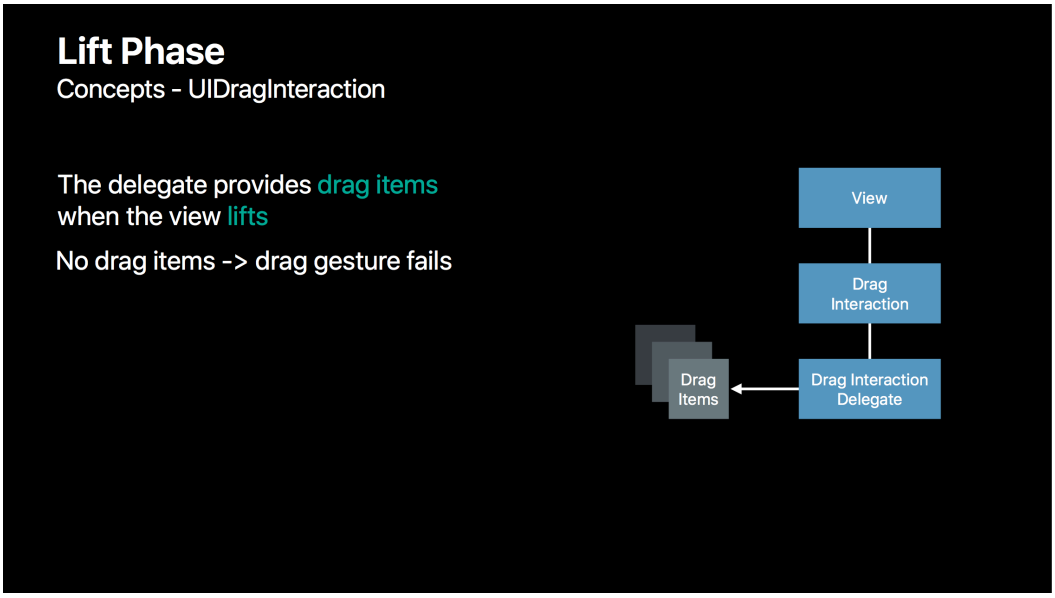
先来看一下 Drag 的简介和使用方法。Drag 的核心在于为 View 增加了Interaction 层，而这一层是用来输入 Drag 手势响应的。在 Session 203 中，Bruce Nilo 用这么一张图来形象深度的描述了其关系：



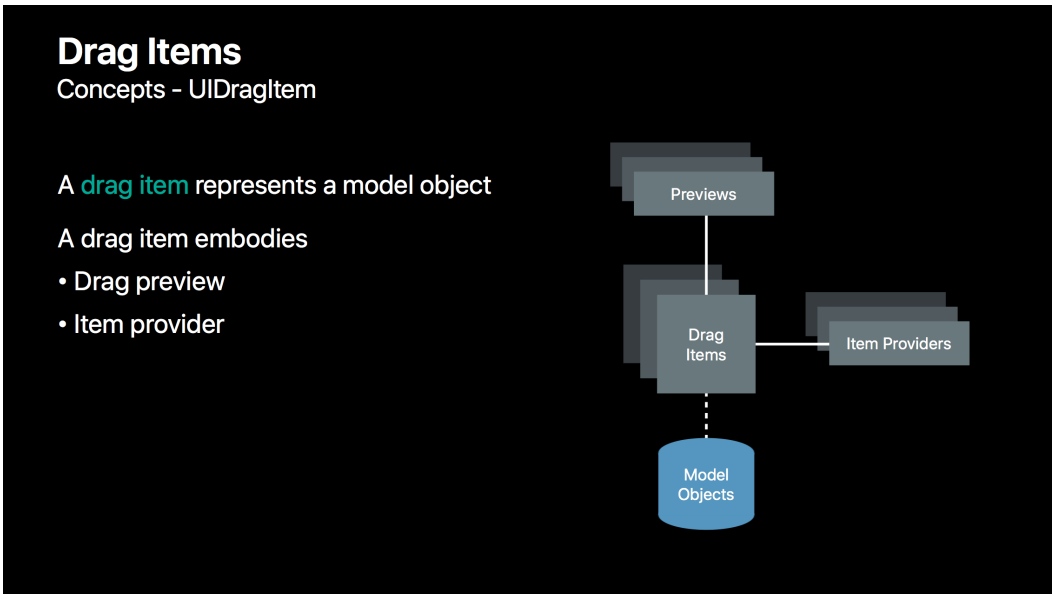
在 View 层上通过增加 Interaction 层来感知输入手势。这样做的好处是，使得在 iOS 平台上的 Drag and Drop 效果统一，并且不用重构项目即可增加 Drag and Drop 功能。

在功能实现上，Drag 采用 Cocoa 一贯的作风，使用代理来实现各个状态时的回调事件。

通过简单的代码就可以了解 `UIDragInteraction` 的使用方式。而对于多个内容（Item）的拖拽过程，Bruce Nilo 为我们引入了 Drag 中的一个新概念——**lift**。



Drag Interaction 协议方法提供了多内容的拖拽，如何抉择是哪些内容接受了 Drag 手势的影响，这需要将 View 进行 lift 操作后，就可以完成。让内容达到 lift 状态很简单，在其他内容处于 Drag 状态时，只需要轻按其他内容，即可添加。当没有拖拽手势响应的内容时，正如上 Keynote 所示，拖拽手势将立即失效。

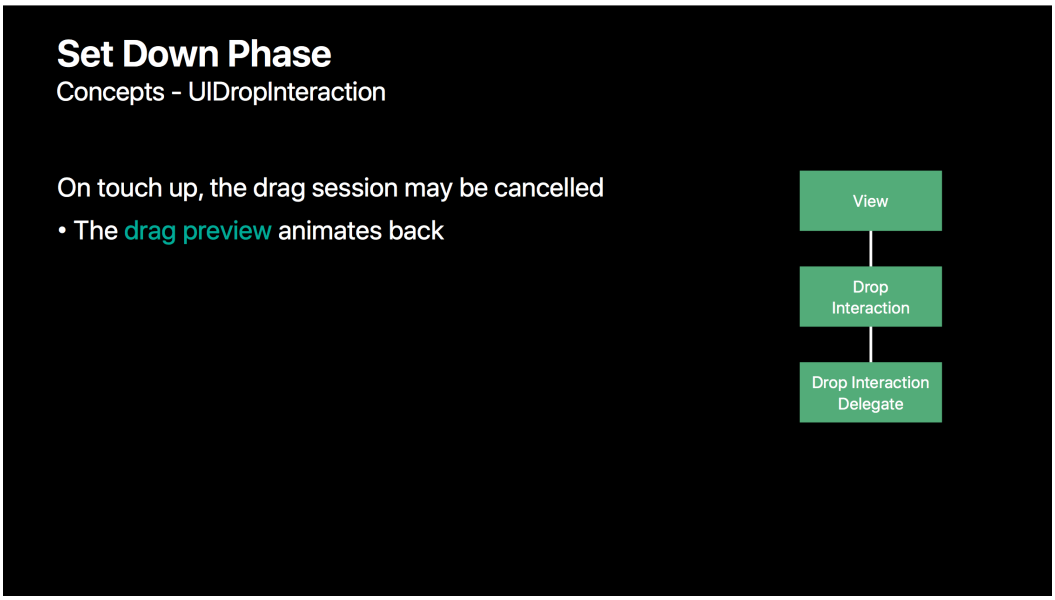


除了 lift，这里还引入了 Drag 内容（Drag Items）这么一个概念，什么是 Drag 内容呢？Drag 内容是为了完成 Drag 手势交互而附加到 View 上的模型对象。Drag 内容对象用于实现在屏幕上的拖动状态，并预览拖动的过程。

使用 Drop

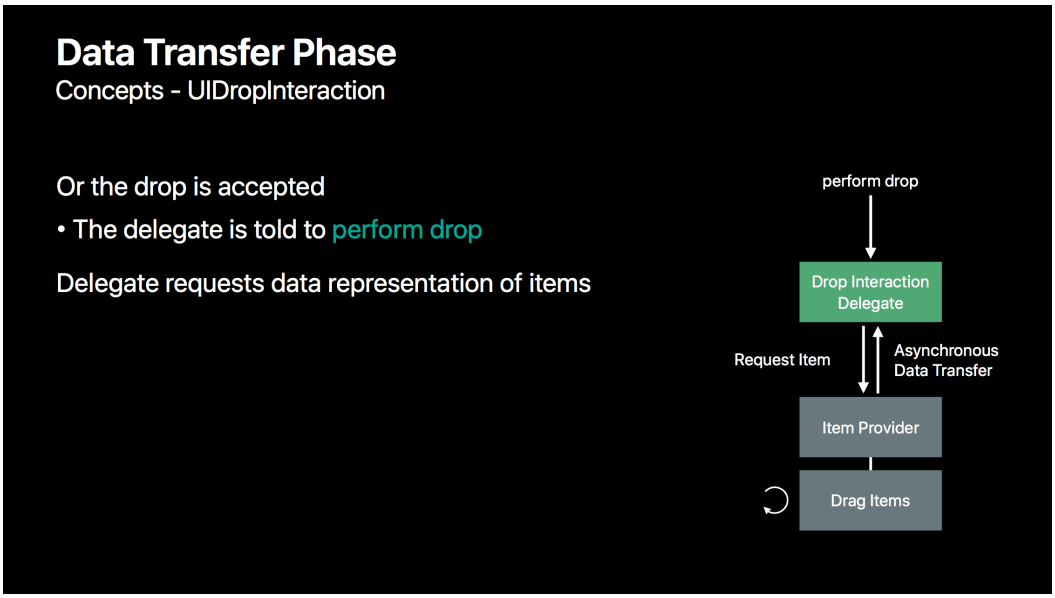
下面我们来说说 Drop 的启用方式。第一种方式是我们需要初始化一个 Paste Configuration。顾名思义，这个配置的对象和粘贴（paste）行为有关。这个配置从使用上来讲就是确定了 Drag 之后进行的 Drop 行为类型，可以是普通的 Drop 也可以是 Paste。想使用 Paste Configuration，需要实现一个 `pasteitemProviders` 的新方法。

我们发现，使用 Paste Configuration 来实现 Drop 过于局限，因为供我们选择的情况只有两种。对于更加复杂的方式，其实现方法同上文的 Drag 类似，就是利用 Drop Interactions 层并通过代理的方式控制行为。



当我们的手指离开屏幕的时候，数据就会传输到 Drag 手势的结束位置，即 Drop 手势的响应位置。这期间是一个什么过程呢?

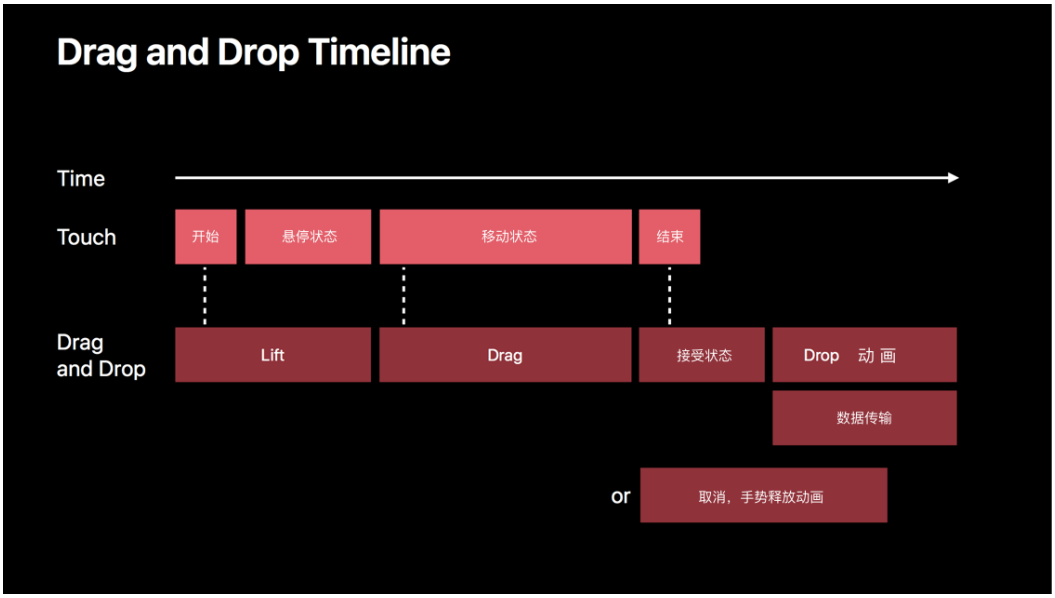
当手指拖拽 Drag 内容到可以 Drop 的位置时，由于这个位置上的 View 是设置过 Drop Interactions 的，所以就会更改提示图样，表示：“我对这些 Drag 内容感兴趣，可以接收”。在这个范围内的 Drop 会触发指定的 Interactions 代理方法的回调，回调方法中甚至可以拿到 Drag 内容并进行处理。而这一切的数据传输，都是以异步的形式。



从示意图中可以看出，所有的数据交互最终都由 `NSItemProvider` 进行获取和接收。在 iOS 8 之后，Apple 终于又想起来了这个曾经用于媒体数据分享的封装类。`NSItemProvider` 由于获取的是媒体资源，所以过程开销较长，十分容易阻塞线程。于是在 Drag and Drop 中为其开启异步执行的方法，增强体验。

Drag and Drop Timeline

下面我们来说一下 Drag and Drop 的 Timeline。整个的流程如 Session 203 中的图一样，我在这里把它修改成了中文描述：



描述一下上图表述的流程：当用户按下屏幕上的一个视图对象的时候，Lift 状态出发并相应 Lift Animation 效果。然后用户也许会将该对象拖拽到其他的 App 中，这时候进入了 Drag 状态。用户的手势释放后，Drop 状态启动，内部处理此位置是否可以接受该数据，根据代码实现的各种条件来确定 Drop Animation 以及数据的传输，或者无法接受数据，取消掉当前操作并执行 Drop Animation。

Drag and Drop 协议中的交互方法

Drag Interaction Delegate

Lift

在 Drag Interaction 中也包括了 Lift 的回调方法。可以说 Lift 过程是 Drag 过程的预备工作。Lift 过程包括了文件选择的动画、处于 Drag 时的显示状态等等。

```
func dragInteraction(_ interaction:UIDragInteraction, previewForLifting
item:UIDragItem, session:UIDragSession) -> UITargetedDragPreview?
```

这个回调方法是用来做什么的呢？说白了，就定制 Lift 的动画及显示效果。系统默认的效果是当按住一个内容后，选中内容以块状的形式“提升”，来提示你现在可以拖动这些内容。此时预览图将是选中的内容本身，文本即为文本、图片即为图片快照等等。倘若你不喜欢它，你可以将预览图进行修改即可。在会议中 Kurt Revis 给出了一个示例：

```
func dragInteraction(_ interaction:UIDragInteraction, previewForLifting
item:UIDragItem, session:UIDragSession) -> UITargetedDragPreview? {
    // 声明 imageView 实例
    let imageView = UIImageView(image: UIImage(named: "MyDragImage"))
    // 获取 interaction 层的 View
    let dragView = interaction.view!
    // 获取 dragView 的位置，从而确定手指的位置
    let dragPoint = session.location(in: dragView)
    // 使用视图与位置初始化成一个 Drag Preview Target
    let target = UIDragPreviewTarget(container: dragView, center: dragPoint)
    // UITargetedDragPreview 实例
    return UITargetedDragPreview(view: imageView,
parameters:UIDragPreviewParameters(), target: target)
}
```

`UITargetedDragPreview` 需要一个 target 参数是用来告知 `UITargetedDragPreview` 在哪个 superView 和位置上来展示 `Preview`。

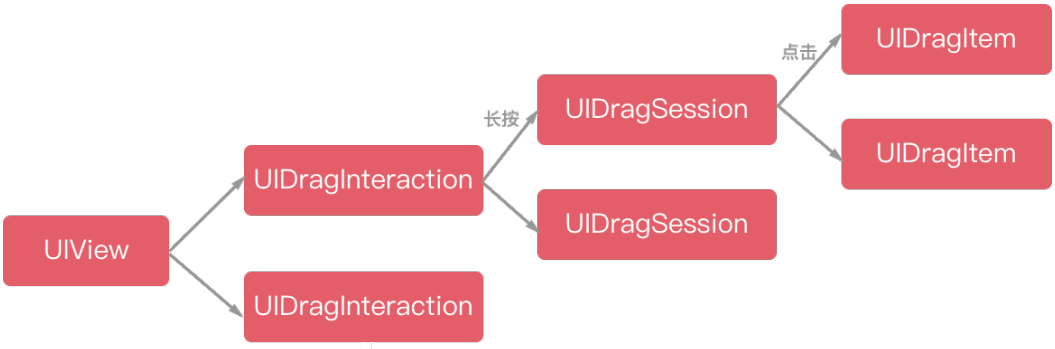
Drag

Lift 过程之后，开始启动 Drag 过程，回调方法将进入 `itemsForBeginning` 方法：

```
func dragInteraction(_ interaction: UIDragInteraction, itemsForBeginning
session: UIDragSession) -> [UIDragItem]
```

这个方法中设计到了三个类，`UIDragInteraction` 可以包含多个 `UIDragSession`，每个 `UIDragSession` 又可以含有多个 `UIDragItem`。`UIDragItem` 则是 Drop 时接收方所受到的对象。

其实对于一个 View 我们可以为其增加多个 `UIDragInteraction`，通过 `enable` 属性来觉得启用哪一个 `UIDragInteraction`。通过 Multiple-Touch 每次长按一个 View 的时候，则会建立一个新的 `UIDragSession`，但是如果单次点击，则会添加一个 `UIDragItem`。下图可以清晰的展示三者关系：



根据回调方法，我们拿到了 Drag 的一组内容。并且这组 `UIDragSession` 已经经历过 `UIDragItem` 的数据组装，这时候即将进入 Drop 状态之前，启动以下代理方法：

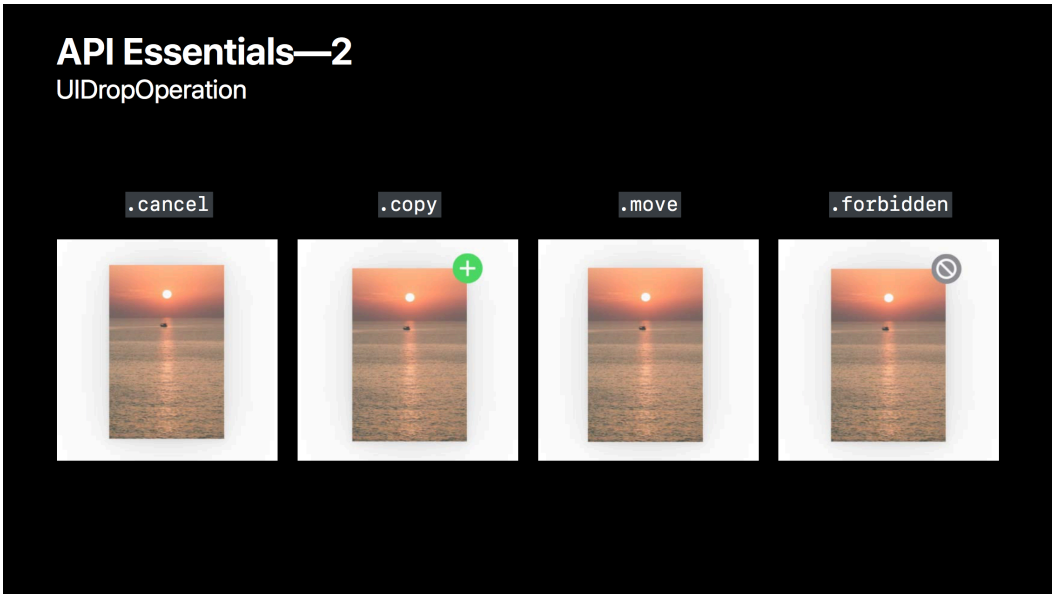
```
func dropInteraction(_ interaction: UIDropInteraction, sessionDidUpdate
session: UIDropSession) -> UIDropProposal
```

在 Drop 前对于数据的排查，主要是为 Drag 状态下如果数据非法或数据为空，提供了一个调用 Cancel 事件的入口，该状况下直接返回 `UIDropOperationCancel` 即可。

这里我们顺带提一下 `UIDropOperation` 的四种枚举类型：

- UIDropOperationCancel
- UIDropOperationCopy
- UIDropOperationMove
- UIDropOperationForbidden

在 Keynote 中已经描述的非常清楚了：



一些其他的代理方法

```
// 某个 UI 元素安装了 UIDragInteraction，单指长按时生成 UIDragSession，进入回调，索
取 UIDragItem
func dragInteraction(_ interaction: UIDragInteraction, itemsForBeginning
session: UIDragSession) -> [UIDragItem];

// 手指 A 长按某个 UI 元素后，手指 B 单击另外的 View，进入回调，允许添加更多的
UIDragItem 到当前 UIDragSession 中
func dragInteraction(_ interaction: UIDragInteraction,itemsForAddingTo
session: UIDragSession,withTouchAt point: CGPoint) -> [UIDragItem];

// 单指长按某个 View，Drag 开始，生成新的 UIDragSession，进入回调
func dragInteraction(_ interaction: UIDragInteraction,sessionWillBegin
session: UIDragSession);
```

Drop Interaction Delegate

Drop

在所有的获取数据和过滤数据过程完成之后进入 Drop 状态。首先先触发 `performDrop` 这个方法：

```
func dropInteraction(_ interaction: UIDropInteraction, performDrop session:
UIDropSession)
```

这个方法可以获取到当前 View 下的 `UIDropInteraction` 对象以及数据集合 `UIDropSession`。可以说是 Drop 状态时对于获取到数据的处理阶段。官方给了这么一个简单的例子：

```
func dropInteraction(_ interaction: UIDropInteraction, performDrop session:
UIDropSession) {
    for item in session.items {
        item.itemProvider.loadObject(ofClass: UIImage.self) { (object, error)
in
            if object != nil {
                DispatchQueue.main.async {
                    self.imageView.image = (object as! UIImage)
                }
            } else {
                // error 处理
            }
        }
    }
}
```

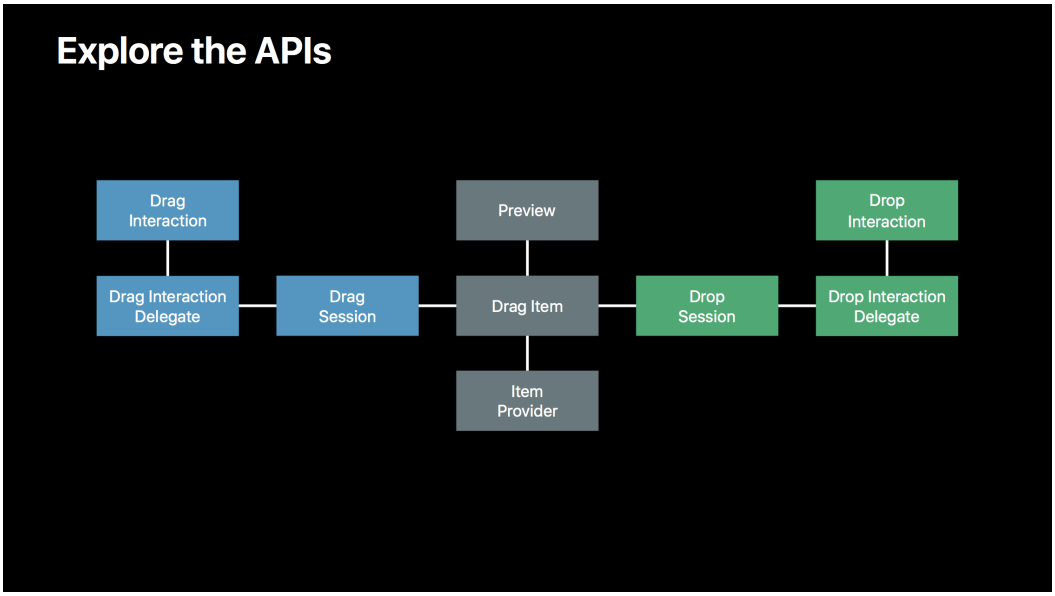
使用 `itemProvider` 来获取要更新的数据，从而替换 `imageView` 中的显示图片。

一些其他的代理方法

```
// Drag 的 View 元素进入 Drop 的区域
func dropInteraction(_ interaction: UIDropInteraction, sessionDidEnter
session: UIDropSession);

// Drag 的 View 元素在 Drop 区域内反复移动，多次进入
func dropInteraction(_ interaction: UIDropInteraction, sessionDidUpdate
session: UIDropSession) -> UIDropProposal;

// Drag 的 View 元素离开 Drop 的区域
func dropInteraction(_ interaction: UIDropInteraction, sessionDidExit
session: UIDropSession);
```

API 一览

最后我们来看一下 Drag and Drop 的 API 总结构：



所有的方法都是围绕着 Drag 内容展开的，然后根据内容按照 Drag 和 Drop 两个状态来设计 API 结构。在每个状态的过程方法中，Apple 使用了 Interaction Delegate 来设计每个过程的实现，将其方法委托给调用 Drag and Drop 控制逻辑的模块，以达到 Drag and Drop 的解耦。

总结

Drag and Drop 是一种 App 间数据交互的新体验，而这种体验对于 iPad 的大屏是十分友好的，但对于 iPhone 只能说体验一般。所以 Apple 也在 iPad 上主打这一功能。iOS 11 对于 iPad 的优化，可以看出苹果对于 iPad 来投入工作寄予希望，也是 iPad 逐渐的从一个娱乐平板变成办公设备的一个标致。

参考

- WWDC 2017 Session 213 - Mastering Drag and Drop
- WWDC 2017 Session 223 - Drag and Drop with Collection and Table View
- WWDC 2017 Session 227 - Data Delivery with Drag and Drop
- Drag and Drop 开发者官方文档

ARKit 介绍：iOS 上的增强现实

引言

ARKit 为开发 iPhone 和 iPad 增强现实（AR）app 提供了一个前沿平台。本文为你介绍 ARKit 框架，学习如何利用其强大的位置追踪和场景理解能力。ARKit 可以和 SceneKit 与 SpriteKit 无缝结合，或是与 Metal 2 配合直接控制渲染。

下面会为你讲解如何在 iOS 上创建完全自定义的增强现实体验，包括概念和实际的代码。很多开发者都迫不及待想拥抱增强现实，现在有了 ARKit，一切都变得相当简单 :)

增强现实

什么是增强现实?

增强现实就是创建一种在物理世界中放置虚拟物体的错觉。从 iPhone 或 iPad 的相机中看进虚拟世界，就像一面魔法透镜。

例子

先来看几个例子。Apple 已经让一部分开发者早先接触了 ARKit，这些都是他们的作品。让我们试着见微知著，看看不久的将来都会发生什么。

这是一家专注于“沉浸式讲故事”体验的公司，他们用 AR 讲述了《金发姑娘与三只熊》的故事。把一间卧室变成了一本虚拟的故事书，可以通过娓娓道来的文字推动故事进行，但更重要的是，可以让孩子从任意视角来探索故事场景。

这种级别的交互真的可以把虚拟场景变得更加活灵活现。

下一个例子是宜家，宜家使用 ARKit 来重新设计你的客厅。可以在物理物体旁边放上虚拟内容，为用户打开了一个充满无限可能性的新世界。

最后一个例子是游戏，Pokemon Go。

大名鼎鼎的 Pokemon Go 借助 ARKit 把小精灵的捕捉提升到了全新的 level。可以把虚拟内容固定在现实世界中，的确获得了比之前更加身临其境的体验。

小结

以上就是增强现实的四个例子，但还远远不止于此。有许许多多方法可以借助增强现实来提升用户体验。但增强现实需要很多领域的知识。从计算机视觉、传感器数据混合处理，到与硬件对话以获得相机校准和相机内部功能。Apple 想让这一切变得容易。所以 WWDC 2017 发布了 ARKit。

ARKit

- ARKit 是一个移动端 AR 平台，用于在 iOS 上开发增强现实 app。
- ARKit 提供了接口简单的高级 API，有一系列强大的功能。
- 但更重要的是，它也会在目前的数千万台 iOS 设备上推出。为了获得 ARKit 的完整功能，需要 A9 及以上芯片。其实也就是大部分运行 iOS 11 的设备，包括 iPhone 6S。

功能

那么 ARKit 都有哪些功能呢？其实 ARKit 可以被明确分为三层，第一层是追踪。

追踪（Tracking）

追踪是 ARKit 的核心功能，也就是可以实时追踪设备。

- 世界追踪（world tracking）可以提供设备在物理环境中的相对位置。
- 借助视觉惯性里程计Visual–Inertial Odometry(VIO)，可以提供设备所在位置的精确视图以及设备朝向，视觉惯性里程计使用了相机图像和设备的运动数据。
- 更重要的是不需要外设，不需要提前了解所处的环境，也不需要另外的传感器。

场景理解（Scene Understanding）

追踪上面一层是场景理解，即确定设备周围环境的属性或特征。它会提供诸如平面检测（plane detection）等功能。

- 平面检测能够确定物理环境中的表面或平面。例如地板或桌子。
- 为了放置虚拟物体，Apple 还提供了命中测试功能。此功能可获得与真实世界拓扑的相交点，以便在物理世界中放置虚拟物体。
- 最后，场景理解可以进行光线估算。光线估算用于正确光照你的虚拟几何体，使其与物理世界相匹配。
  结合使用上述功能，可以将虚拟内容无缝整合进物理环境。所以 ARKit 的最后一层就是**渲染**。

渲染（Rendering）

- Apple 让我们可以轻易整合任意渲染程序。他们提供的持续相机图像流、追踪信息以及场景理解都可以被导入任意渲染程序中。
- 对于使用 SceneKit 或 SpriteKit 的人，Apple 提供了自定义 AR view，替你完成了大部分的渲染。所以真的很容易上手。
- 同时对于做自定义渲染的人，Apple 通过 Xcode 提供了一个 metal 模板，可以把 ARKit 整合进你的自定义渲染器。

one more thing

Unity 和 UNREAL 会支持 ARKit 的全部功能。

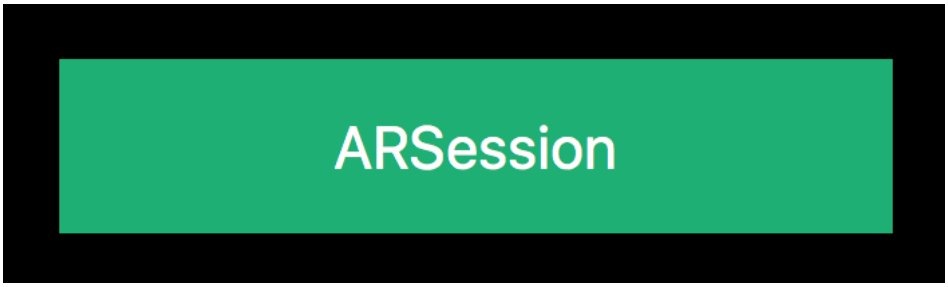
使用 ARKit

创建增强现实体验需要的所有**处理**都由 ARKit 框架负责。

选好处理程序后，只要使用 ARKit 来完场处理的部分即可。渲染增强现实场景所需的所有信息都由 ARKit 来提供。

除了**处理**，ARKit 还负责捕捉信息，这些信息用于构建增强现实。ARKit 会在幕后使用 AVFoundation 和 CoreMotion，从设备捕捉图像和运动数据以进行追踪，并为渲染程序提供相机图像。

所以如何使用 ARKit 呢?



ARKit 是基于 session 的 API。所以首先你要做创建一个简单的 ARSession。ARSession 对象用于控制所有**处理**流程，这些流程用于创建增强现实 app。

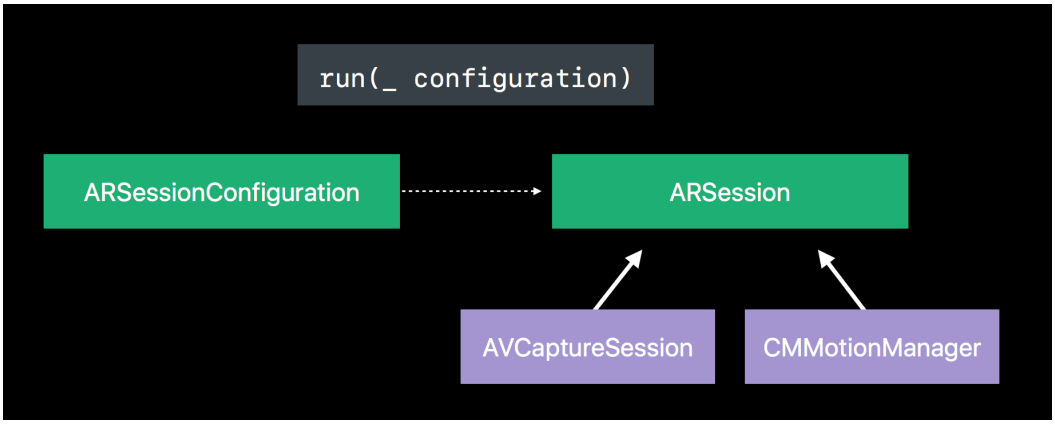
但首先需要确定增强现实 app 将会做哪种类型的追踪。所以，还要创建一个 ARSessionConfiguration。

ARSessionConfiguration 及其子类用于确定 session 将会运行什么样的追踪。只要把对应的属性设置为 enable 或 disable，就可以获得不同类型的场景理解，并让 ARSession 做不同的处理。

要运行 session，只要对 ARSession 调用 run 方法即可，带上所需的 configuration。

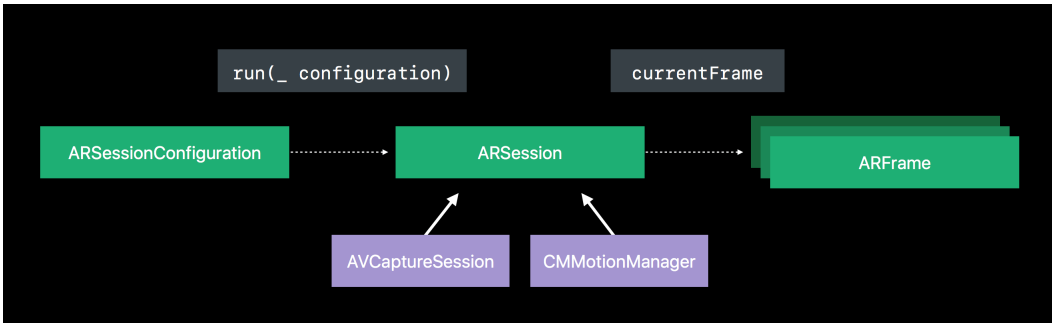
```
run(_ configuration)
```

然后处理流程就会立刻开始。同时底层也会开始捕捉信息。



所以幕后会自动创建 AVCaptureSession 和 CMMotionManager。它们用于获取图像数据和运动数据，这些数据会被用于追踪。

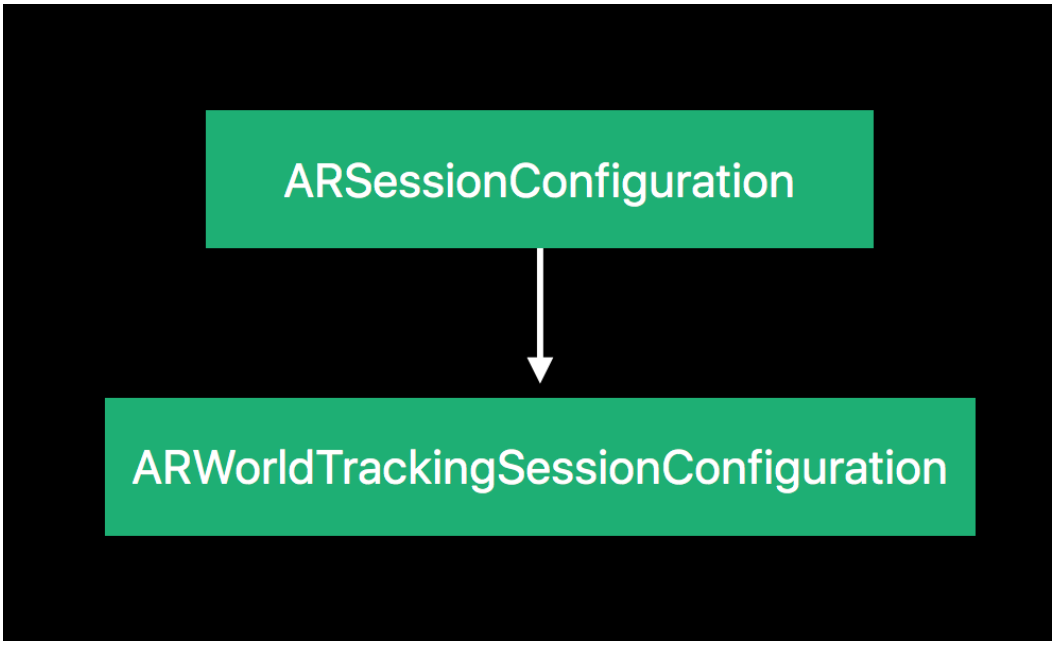
处理完成后，ARSession 会输出 ARFrames。



ARFrame 就是当前时刻的快照，包括 session 的所有状态，所有渲染增强现实场景所需的信息。要访问 ARFrame，只要获取 ARSession 的 currentFrame 属性。或者也可以把自己设置为 delegate，接收新的 ARFrame。

ARSessionConfiguration

下面详细讲解一下 ARSessionConfiguration。ARSessionConfiguration 用于确定 session 上将会运行哪种类型的追踪。所以它提供了不同的 configuration 类。基类是 ARSessionConfiguration，提供了三个追踪自由度，也就是设备角度。其子类 ARWorldTrackingSessionConfiguration 提供六个追踪自由度。



这个 World tracking 世界追踪是核心功能，不仅可以获得设备角度，还能获得设备的相对位置，此外还能获得有关场景的信息。因为有它，才能够进行场景理解，例如获得特征点以及在世界中的物理位置。要打开或关闭功能，只要设置 session configuration 类的属性即可。

session configuration 还可以告诉你可用性（availability）。如果你想知道当前设备是否支持直接追踪，只要检查 ARWorldTrackingSessionConfiguration 类的属性 isSupported 即可。

```
if ARWorldTrackingSessionConfiguration.isSupported {
    configuration = ARWorldTrackingSessionConfiguration()
}
else {
    configuration = ARSessionConfiguration()
}
```

如果支持的话就可以用 WorldTrackingSessionConfiguration，否则就降回只提供三个自由度的基类 ARSessionConfiguration。

这里要重点注意，由于基类没有如何场景理解功能，例如命中测试在某些设备上就不可用。所以 Apple 还提供了 UI required device capability，可以在 app 里设置，这样 app 就只会出现在受支持设备的 App Store 里。

ARSession

管理 AR 处理流程

刚刚说过，ARSession 是管理增强现实 app 所有**处理**流程的类。除了带 configuration 参数调用 run 之外，还可以调用 pause。pause 可以暂停 session 上所有处理流程。例如 view 不在前台了，就可以停止处理，以停止使用 CPU，暂停时追踪不会进行。要在暂停后恢复追踪，只要再次对 session 调用 run，参数即它自己的 configuration。最后，你可以多次调用 run 以在不同的 configuration 间切换。假设我想启用平面检测，就可以更改 configuration，再次对 session 调用 run，从而打开平面检测。session 会自动在两个 configuration 之间无缝转换，而不会丢失任何相机图像。

```
// 运行 session
session.run(configuration)

// 暂停 session
session.pause()

// 恢复 session
session.run(session.configuration)

// 改变 configuration
session.run(otherConfiguration)
```

重置追踪

除了 run 命令，还可以重置追踪。运行 run 命令时带上 options 参数即可重置追踪。

```
// 重置追踪
session.run(configuration, options: .resetTracking)
```

这样会重新初始化目前的所有追踪。相机位置也会再次从 0,0,0 开始。所以如果你想将应用重置为某个初始点，这个方法会很有用。

Session 更新

所以如何使用 ARSession 的处理结果呢？把自己设置为 delegate 就可以接收 session 更新。要获取最近一帧，就可以实现 session didUpdate Frame。要进行错误处理，就可以实现 session didFailWithError，此方法用于处理 fatal 错误，例如设备不支持世界追踪就会出现这样的错误，session 则会被暂停。

```
// 访问最近一帧
func session(_: ARSession, didUpdate: ARFrame)

// 处理 session 错误
func session(_: ARSession, didFailWithError: Error)
```

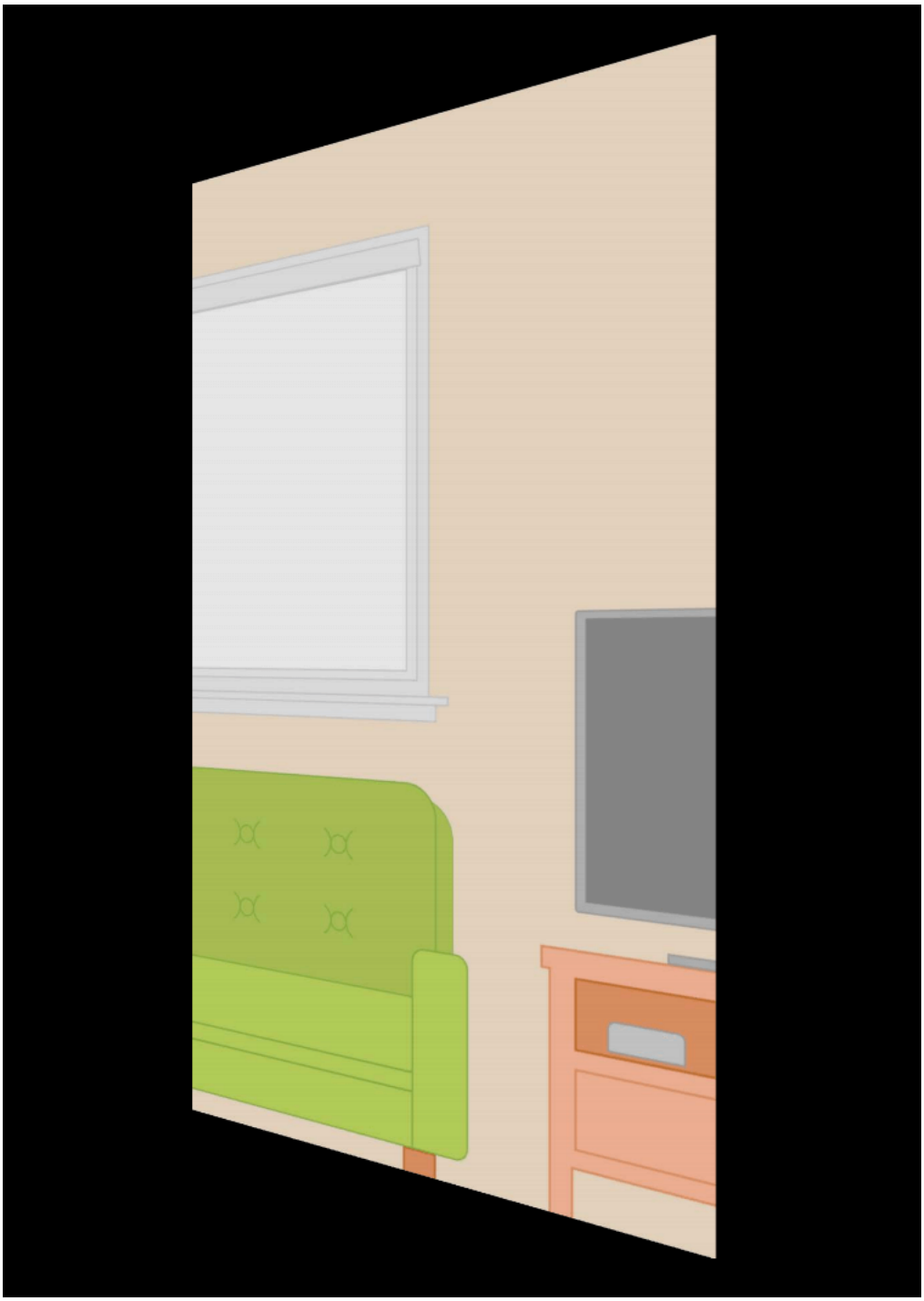
currentFrame

使用 ARSession 处理结果的另一种方式是通过 currentFrame 属性。

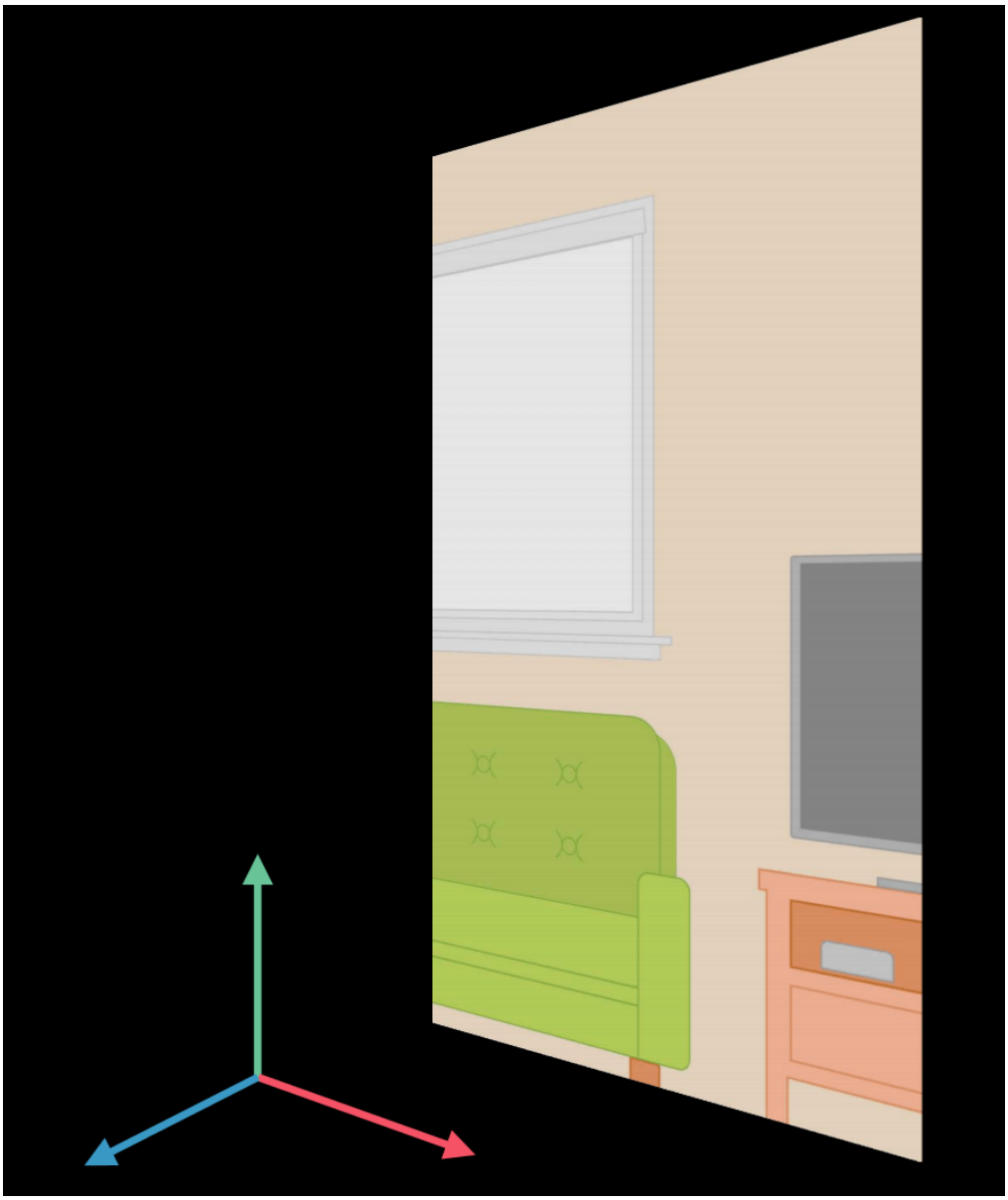
ARFrame

那么 ARFrame 都包含什么东西呢？渲染增强现实场景所需的所有信息，ARFrame 都有。

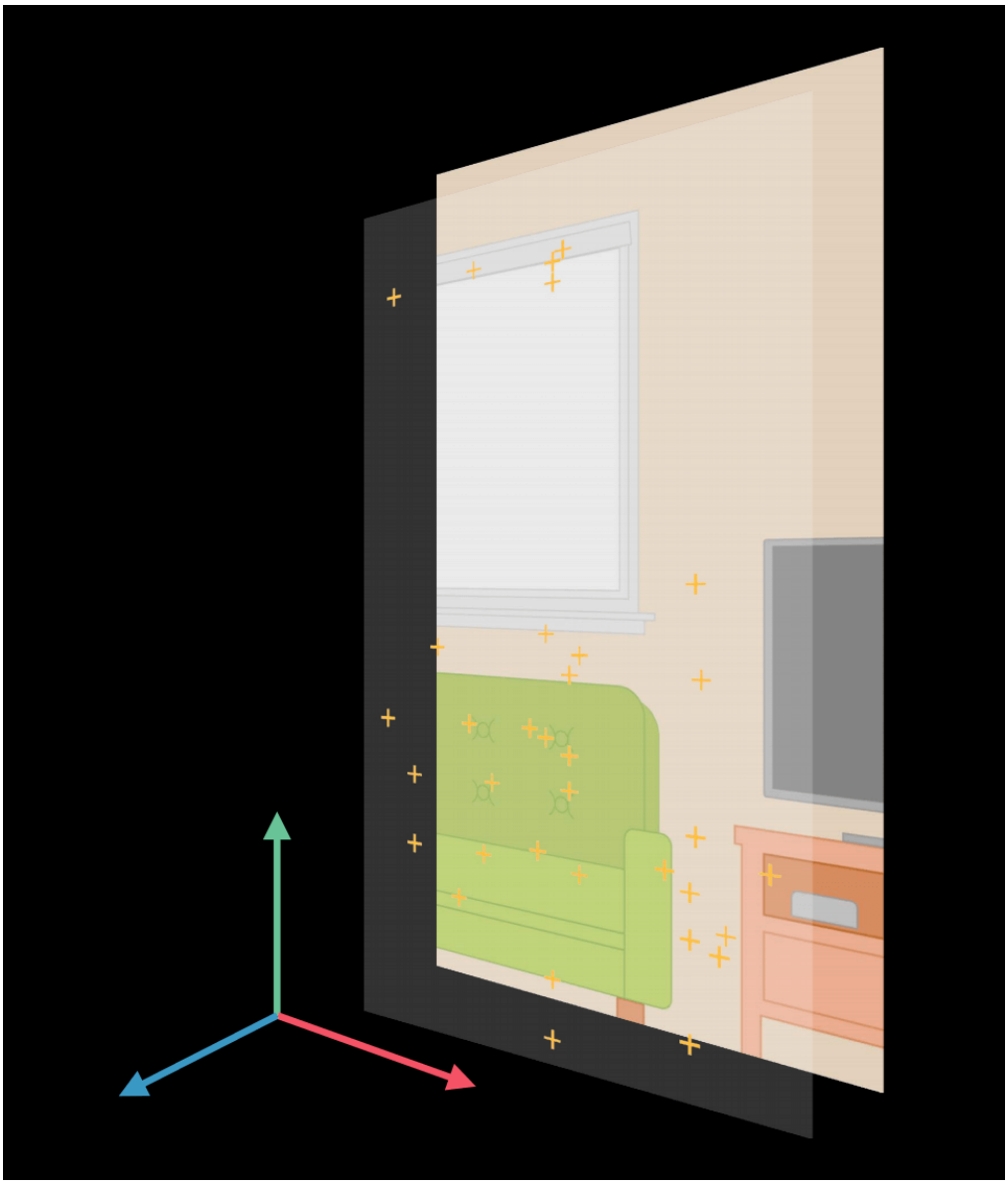
- ARFrame 首先会提供相机图像，用于渲染场景背景。



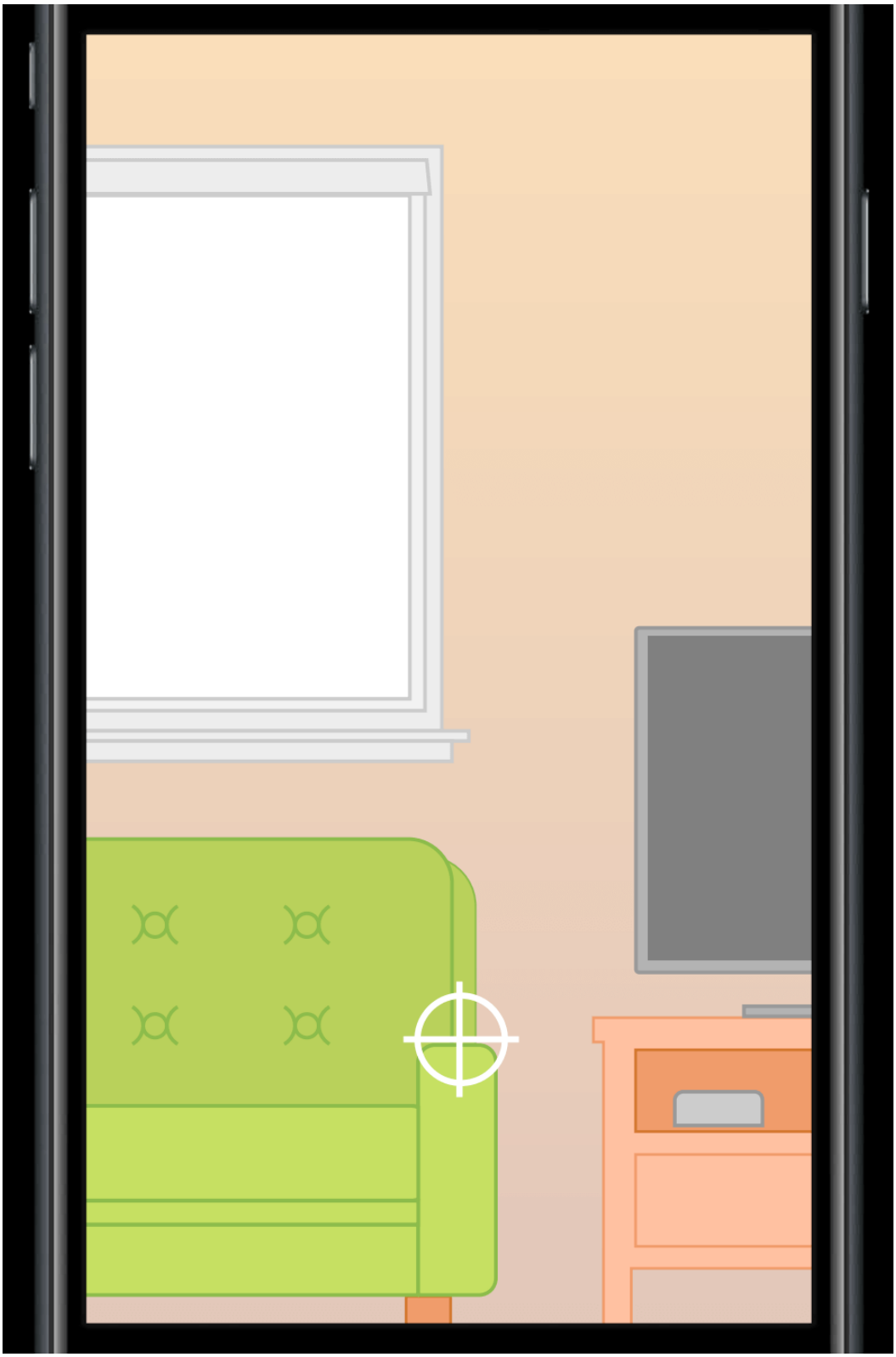
- 其次提供了追踪信息，如设备角度和位置，甚至是追踪状态。



- 最后它提供了场景理解，例如特征点、空间中的物理位置以及光线估算。ARKit 使用 **ARAnchor** 来表示空间中的物理位置。



ARAnchor



- ARAnchor 是空间中相对真实世界的位置和角度。
- ARAnchor 可以添加到场景中，或是从场景中移除。基本上来说，它们用于表示虚拟内容在物理环境中的锚定。所以如果要添加自定义 anchor，添加到 session 里就可以了。它会在 session 生命周期中一直存在。但如果你在运行诸如平面检测功能，ARAnchor 则会被自动添加到 session 中。
- 要响应被添加的 anchor，可以从 current ARFrame 中获得完整列表，此列表包含 session 正在追踪的所有 anchor。
- 或者也可以响应 delegate 方法，例如 add、update 以及 remove，session 中的 anchor 被添加、更新或移除时会通知。

小结

以上是四个主要类，用于创建增强现实体验。下面专门讨论一下追踪。

追踪

追踪就是要实时确定空间中的物理位置。这并不是一件简单的事。但增强现实必须要找到设备的位置和角度，这样才能正确渲染事物。下面看一个例子。

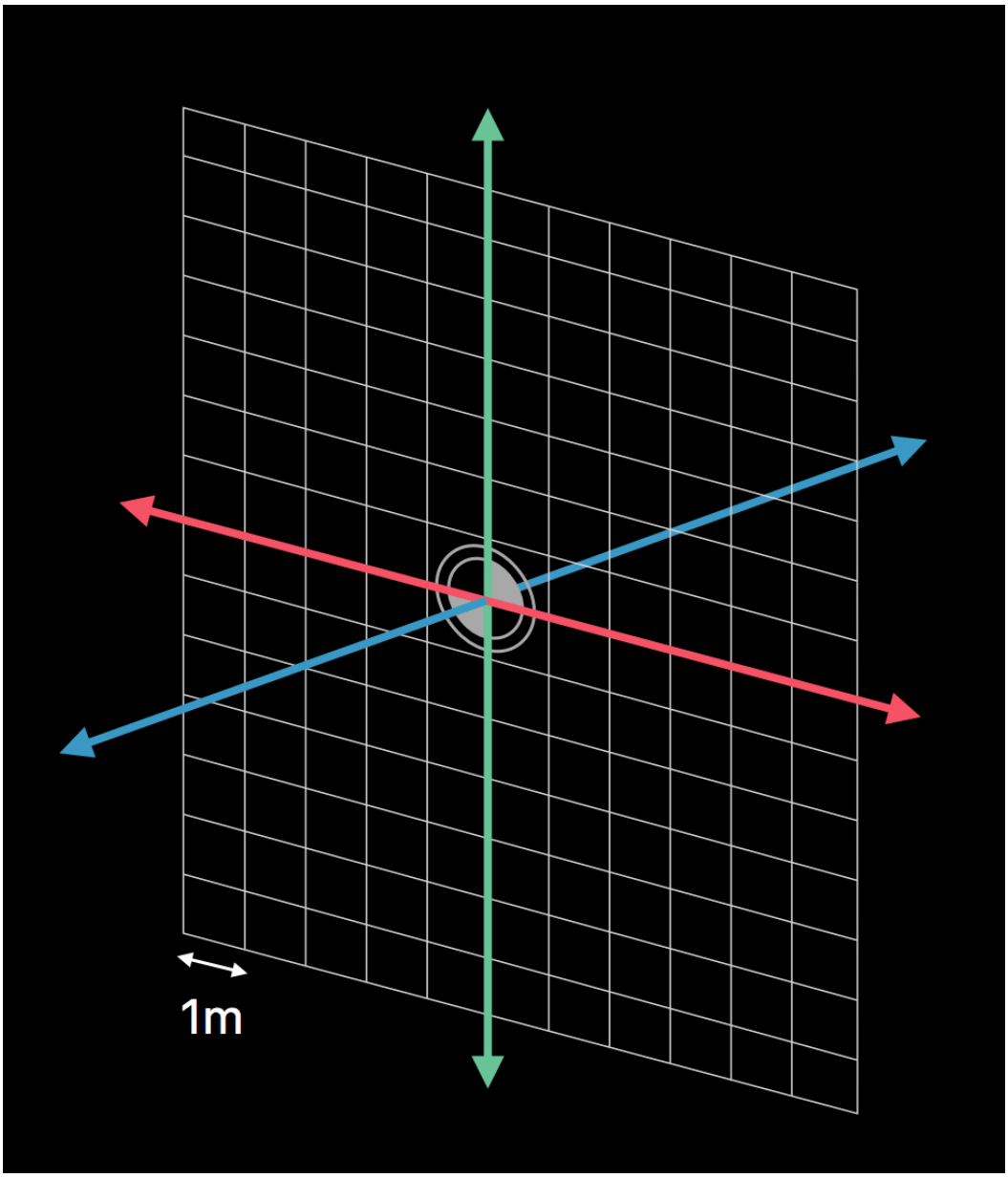
我在物理环境中放了一把虚拟椅子和一张虚拟桌子。如果我把设备转个角度，它们依然固定在空间中。但更重要的是，如果我在场景中走来走去，它们仍被固定在那里。

这是因为我们在不断更新投影的角度，也就用于渲染这个虚拟内容的投影矩阵，使其从任何角度看上去都是正确的。那具体要怎么做呢?

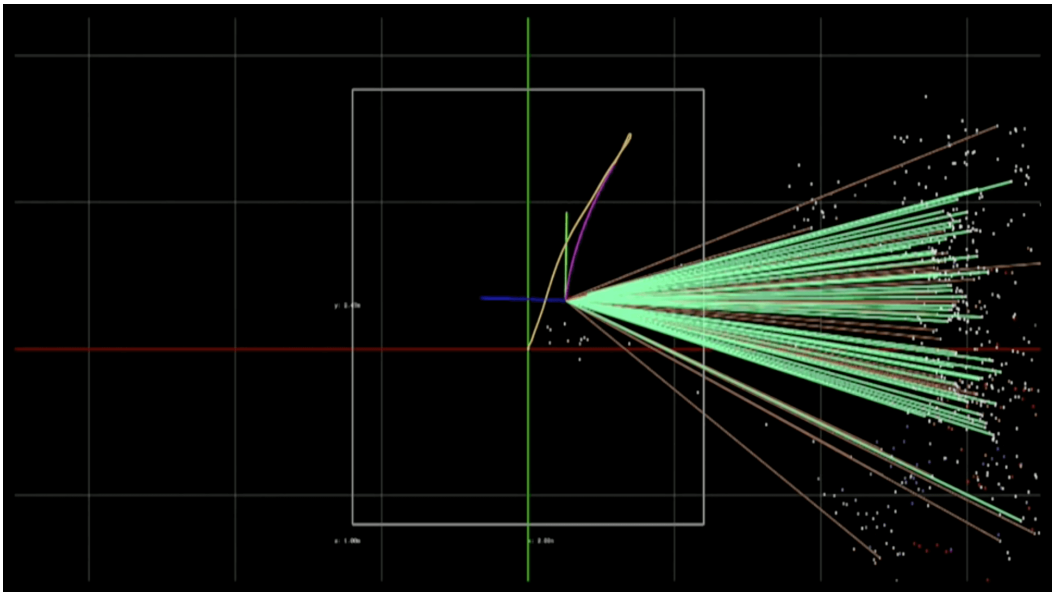
世界追踪

ARKit 提供了世界追踪功能。此技术使用了视觉惯性里程计以及相机图像和运动数据。

- 提供设备的旋转度以及相对位置。但更重要的是，它提供了真实世界比例。所以虚拟内容实际上会被缩放，然后渲染到物理场景中。
- 设备的运动数据计算出了物理移动距离，计算单位为米。
- 追踪给定的所有位置都是相对于 session 的起始位置的。
- 提供 3D 特征点。



世界追踪的工作原理



特征点就是相机图像中的一块块信息碎片，需要检测这些特征点。可以看到，坐标轴表示设备的位置和角度。当用户在世界中移动时，它会画出一条轨迹。这里的小点点就表示场景中已检测到的 3D 特征点。在场景中移动时可以对它们作三角测量，然后用它们去匹配特征，如果匹配之前的特征点则会画出一条线。使用所有这些信息以及运动数据，能够精确提供设备的角度和位置。

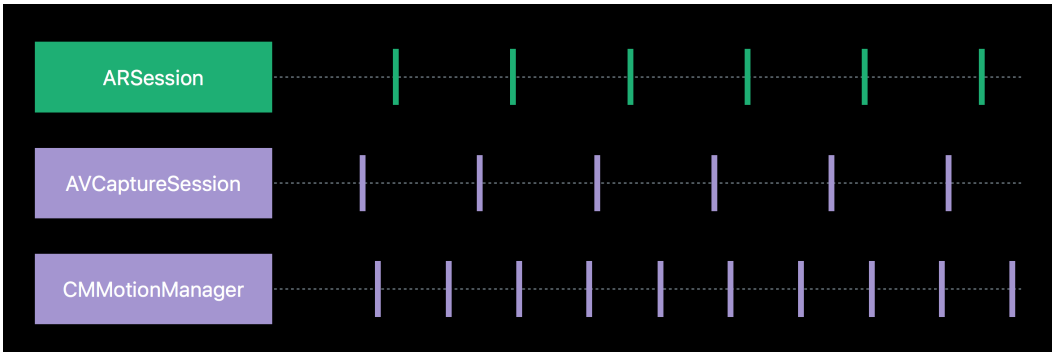
这看起来可能很难。但下面我们来看看如何用代码运行世界追踪。

世界追踪的代码实现

```
// 创建 session
let mySession = ARSession()

// 把自己设为 session delegate
mySession.delegate = self

// 创建 world tracking configuration
let configuration = ARWorldTrackingSessionConfiguration()

// 运行 session
mySession.run(configuration)
```

首先要创建一个 ARSession。之前说过，它会管理世界追踪中所有的处理流程。接下来，把自己设置为 session delegate，这样就可以接收帧的更新。然后创建 WorldTrackingSessionConfiguration，这一步就是在说，“我要用世界追踪。我希望 session 运行这个功能。”然后只要调用 run，处理流程就会立即开始。同时也会开始捕捉信息。

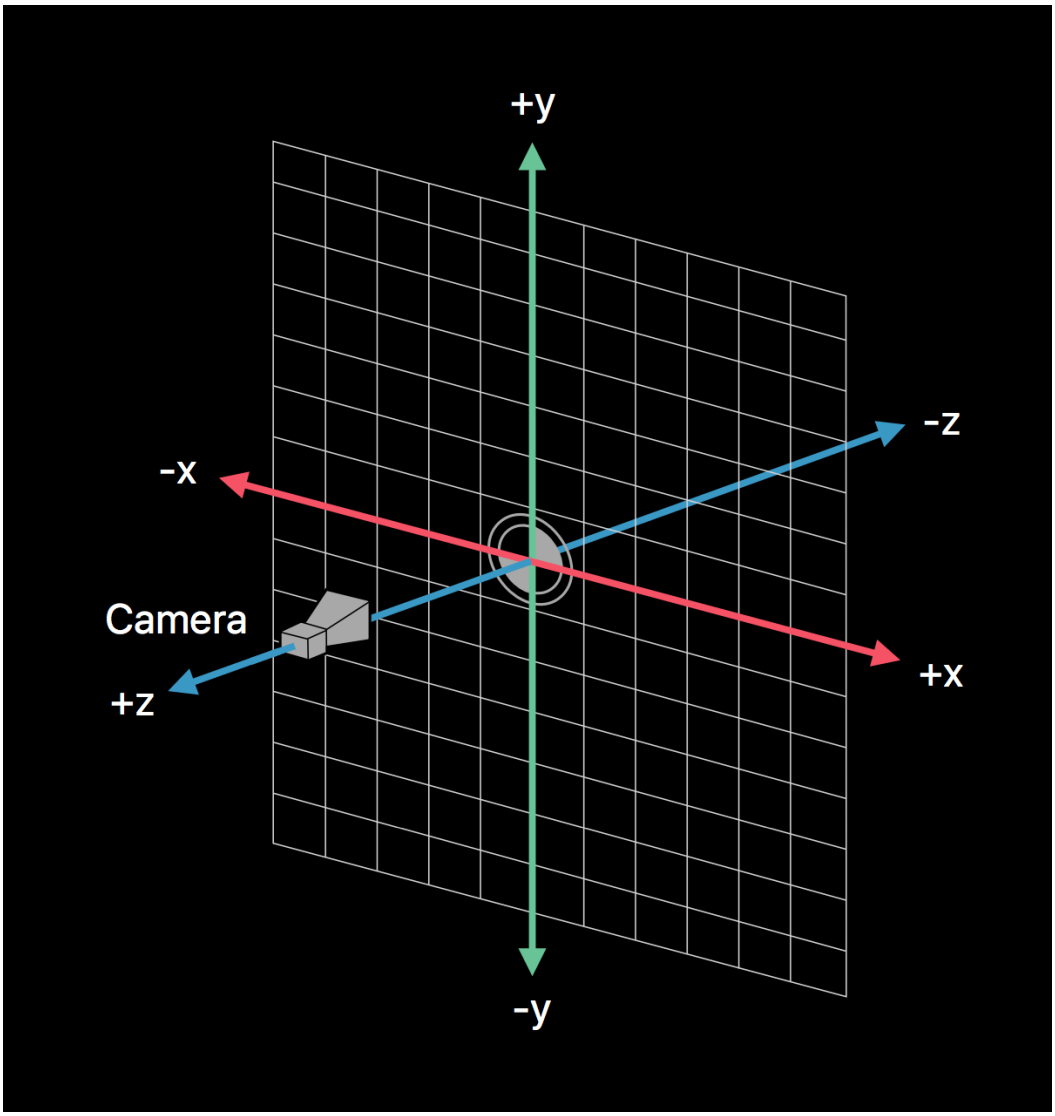
session 在幕后创建了一个 AVCaptureSession 以及一个 CMMotionManager，通过它们获得图像和运动数据。使用图像来检测场景中的特征点。在更高的频率下使用运动数据，随着时间推移计算其积分以获得设备的运动数据。同时使用两者，就能够进行传感器数据混合处理，从而提供精确的角度和位置，并以 ARFrame 形式返回。



ARCamera

每个 ARFrame 都会包含一个 ARCamera。ARCamera 对象表示虚拟摄像头。虚拟摄像头就代表了设备的角度和位置。

- ARCamera 提供了一个 transform。transform 是一个 4x4 矩阵。提供了物理设备相对于初始位置的变换。
- ARCamera 提供了追踪状态（tracking state），通知你如何使用 transform，这个在后面会讲。
- ARCamera 提供了相机内部功能（camera intrinsics）。包括焦距和主焦点，用于寻找投影矩阵。投影矩阵是 ARCamera 上的一个 convenience 方法，可用于渲染虚拟你的几何体。



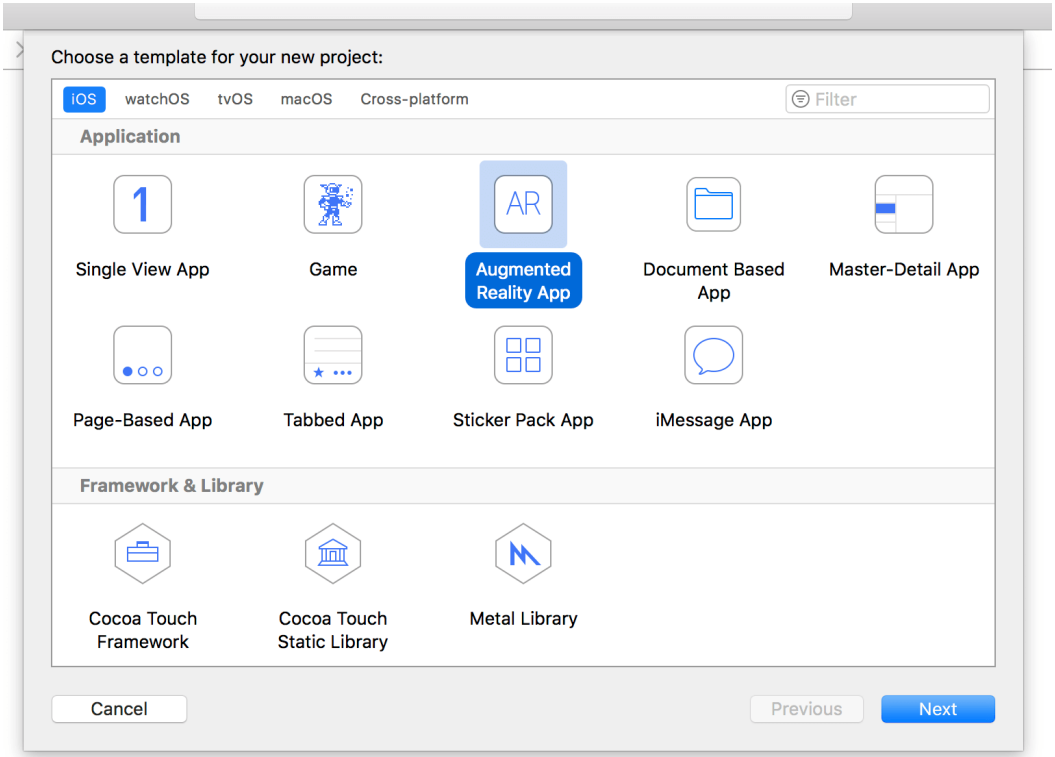
小结

以上就是 ARKit 提供的追踪功能。

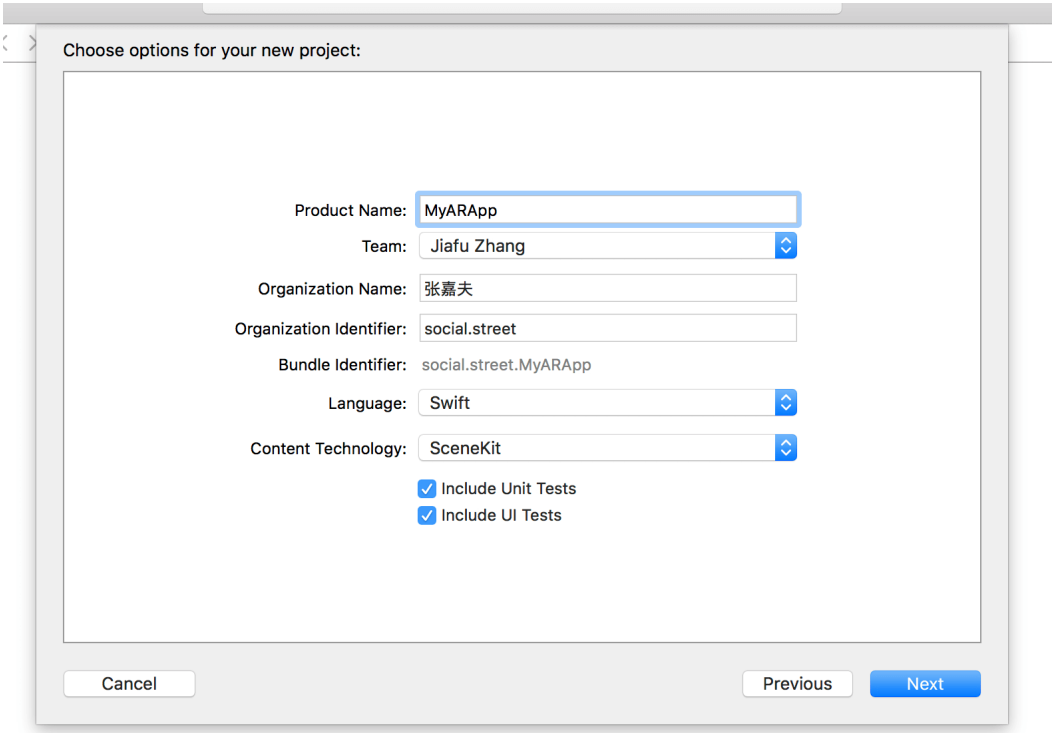
创建第一个 ARKit 应用

下面我们来看一个使用世界追踪的 demo，并创建第一个 ARKit 应用。

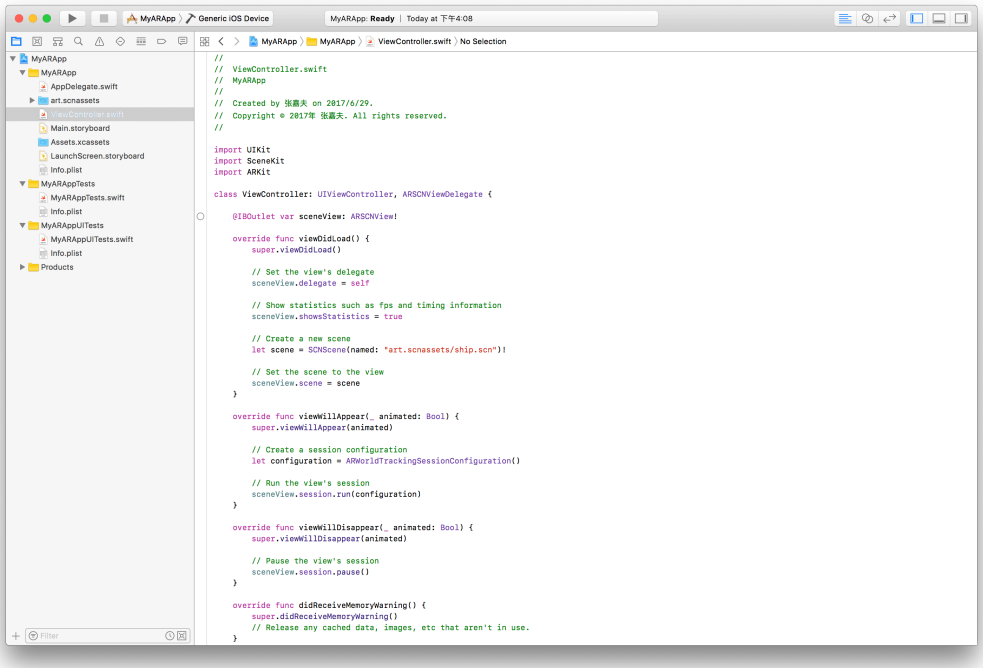
打开 Xcode 9 时会注意到有一张新的模板，用于创建增强现实 app。选择它，然后点 Next。



给定项目名 MyARApp，语言可以选择 Swift 或 Objective-C。这儿还有 Content Technology 选项。Content Technology 是用来渲染增强现实场景的。可以选择 SenceKit、SpriteKit 或 Metal。本例使用 SceneKit。



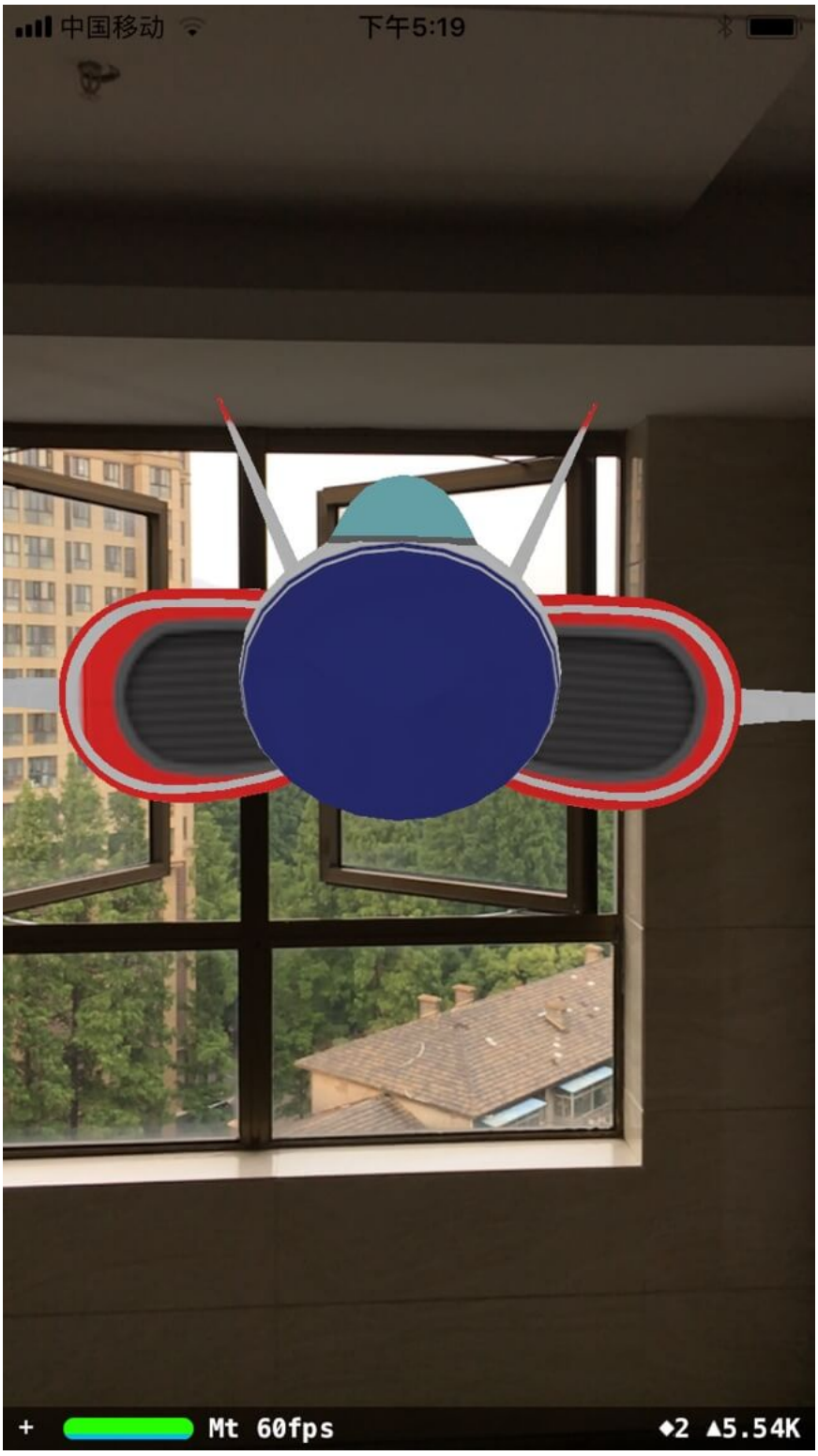
点击 Next 并创建 workspace。



这儿有一个 view controller。它有一个 ARSCNView。这个 ARSCNView 是一个自定义 AR 子类，替我们实现了大部分渲染工作。也就是说它会基于返回的 ARFrame 更新虚拟摄像头。ARSCNView 有一个 session 属性。可以看到给 sceneView 设置了一个 scene，这个 scene 将会是一艘飞船，会处于世界原点处 z 轴往前一点的位置。最重要的部分是对 session 调用 run，带有 WorldTrackingSessionConfiguration 参数。这样就会运行世界追踪，同时 view 会为我们更新虚拟摄像头。

尝试在设备上运行。安装后，会先弹出相机授权，必须使用相机进行追踪并渲染场景背景。

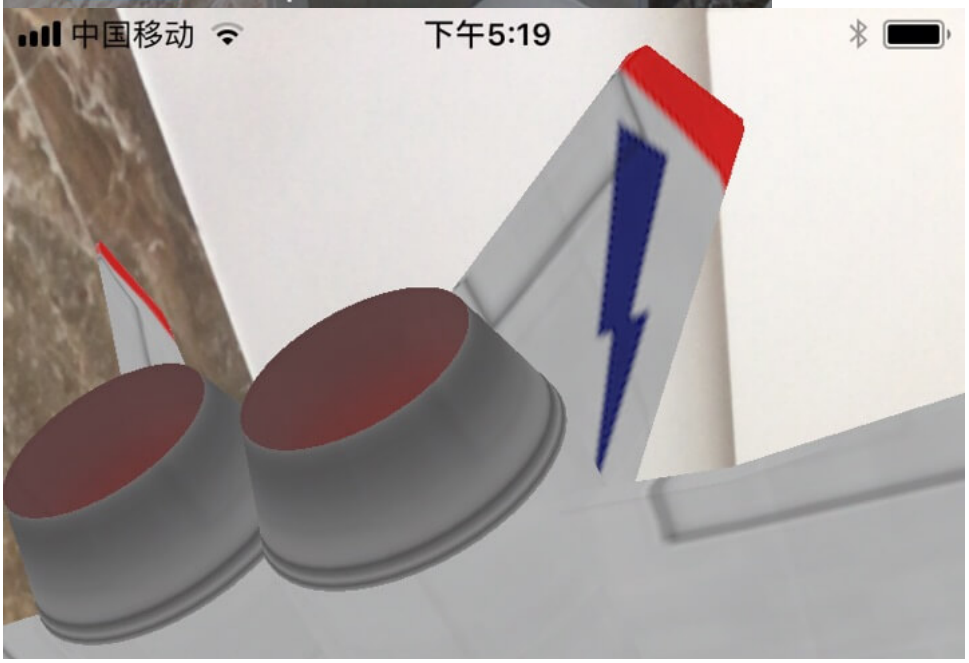
授权后就可以看到摄像头画面。正前方有一艘飞船。

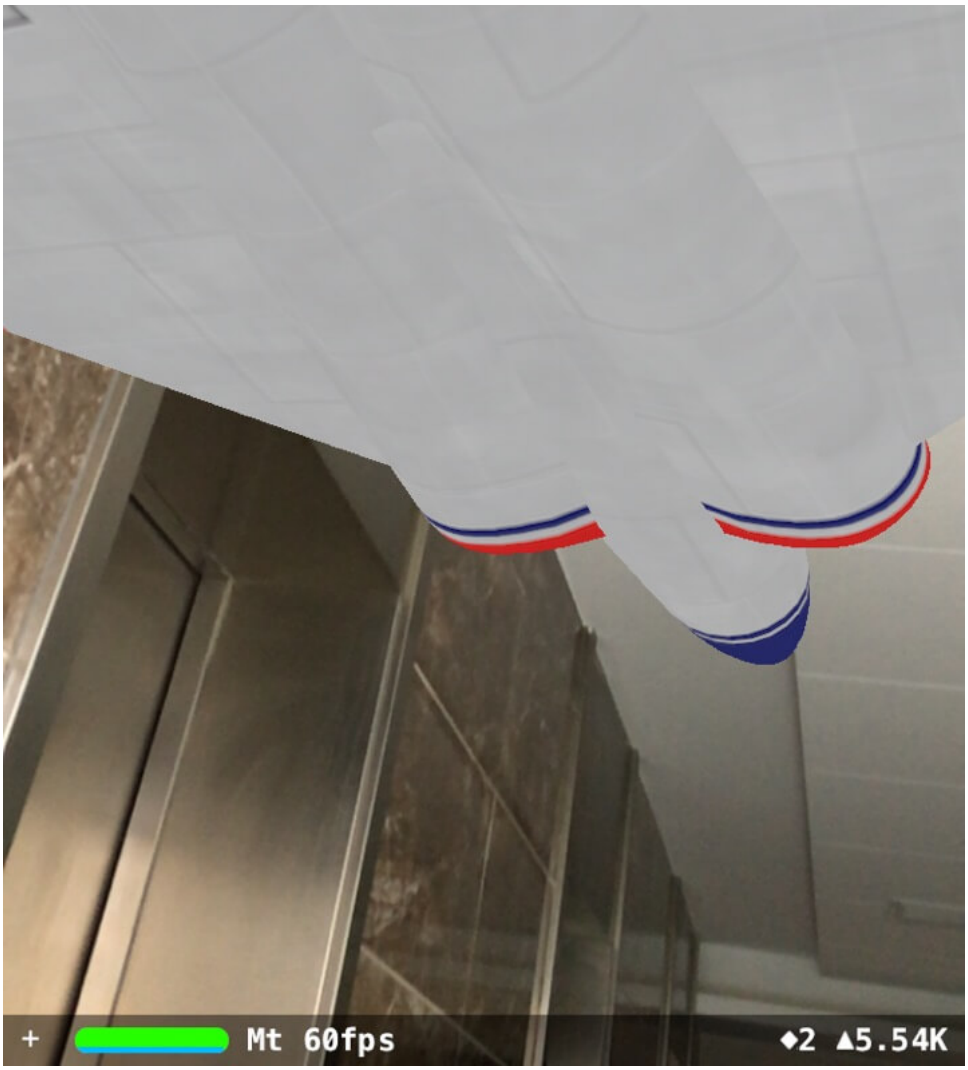


如果改变设备的角度，你会发现它被固定在空间中。

但更重要的是，如果你绕着飞船移动，就会发现它真的被固定在物理世界中了。


中国移动
下午5:19
Mt 60fps
◆2 ▲5.54K


中国移动
下午5:19



实现的原理就是同时使用设备的角度以及相对位置，更新虚拟摄像头，让它看在飞船上。

还不够好玩？来再给它加点料，尝试在点击屏幕时，为场景添加点东西。首先写一个 tap gesture recognizer，然后添加到 view 上，每次点击屏幕时，都会调用 handleTap 方法。

```
    override func viewDidLoad() {
        ...

        // Set the scene to the view
        sceneView.scene = scene

        let tapGesture = UITapGestureRecognizer(target: self, action:
#selector(ViewController.handleTap(gestureRecognizer:)))
        view.addGestureRecognizer(tapGesture)
    }
```

下面实现 handleTap 方法。

```
    @objc
    func handleTap(gestureRecognizer: UITapGestureRecognizer) {
        //使用 view 的快照来创建图片平面
        let imagePlane = SCNPlane(width: sceneView.bounds.width / 6000,
height: sceneView.bounds.height / 6000)
        imagePlane.firstMaterial?.diffuse.contents = sceneView.snapshot()
        imagePlane.firstMaterial?.lightingModel = .constant
    }
```

首先创建一个 SCNPlane，参数是 width 和 height。然后将 material 的 contents 设为 view 的快照（snapshot），这一步可能不是很直观。你猜会怎么样？其实就是把渲染后的 view 截图，包括摄像头画面背景以及前面放的虚拟几何体。然后将光线模型设为 constant，这样 ARKit 提供的光线估算就不会应用此图片上，因为它已经与环境匹配了。下一步要把它添加到场景中。

```
    @objc
    func handleTap(gestureRecognizer: UITapGestureRecognizer) {
        //使用 view 的快照来创建图片平面
        let imagePlane = SCNPlane(width: sceneView.bounds.width / 6000,
height: sceneView.bounds.height / 6000)
        imagePlane.firstMaterial?.diffuse.contents = sceneView.snapshot()
        imagePlane.firstMaterial?.lightingModel = .constant

        //创建 plane node 并添加到场景
        let planeNode = SCNNode(geometry: imagePlane)
        sceneView.scene.rootNode.addChildNode(planeNode)
    }
```

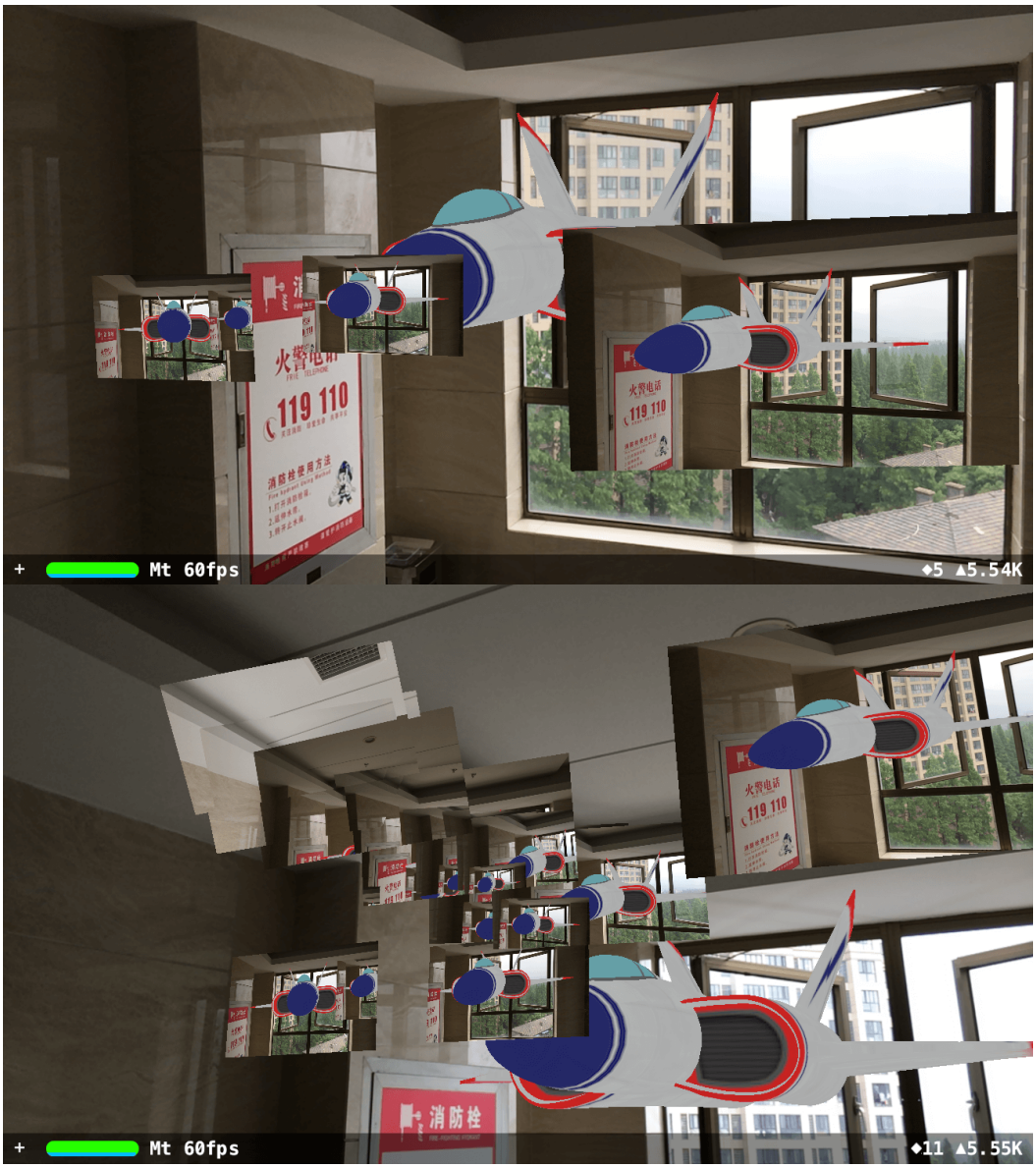
先创建一个 plane node，这个 SCNNode 封装了添加到场景中的几何体。每次触摸屏幕时，就会向场景中添加一个 image plane。但问题是，它总是会在 0, 0, 0 处。 所以怎么变得更好玩呢？我们有一个 current frame，其中包含了一个 ARCamera。我可以借助 camera 的 transform 来更新 plane node 的 transform，这样 plane node 就会处于摄像头当前在空间中的位置了。

```
    @objc
    func handleTap(gestureRecognizer: UITapGestureRecognizer) {
        guard let currentFrame = sceneView.session.currentFrame else {
            return
        }
        //使用 view 的快照来创建图片平面
        let imagePlane = SCNPlane(width: sceneView.bounds.width / 6000,
height: sceneView.bounds.height / 6000)
        imagePlane.firstMaterial?.diffuse.contents = sceneView.snapshot()
        imagePlane.firstMaterial?.lightingModel = .constant

        //创建 plane node 并添加到场景
        let planeNode = SCNNode(geometry: imagePlane)
        sceneView.scene.rootNode.addChildNode(planeNode)

        //将 node 的 transform 设为摄像头前 10cm
        var translation = matrix_identity_float4x4
        translation.columns.3.z = -0.1
        planeNode.simdTransform =
matrix_multiply(currentFrame.camera.transform, translation)
    }
```

首先从 sceneView session 中获得 current frame。下一步，用摄像头的 transform 更新 plane node 的 transform。这一步我先创建了转换矩阵，因为我不想把 image plane 就放在相机的位置，这样会挡住我的视线，所以要把它放在相机前面。所以这里的转换我用了 负z轴。缩放的单位都是米，所以使用 .1 来表示相机前方 10 厘米。将此矩阵和摄像头的 transform 相乘，并将结果应用到 plane node 上，这个 plane node 将会是一个 image plane，位于相机前方 10 厘米处。

现在来试试看会是什么样子。

摄像头场景运行后，可以看到依然有一艘飞船浮在空中。可以试着在任意地方点击屏幕，可以看到快照图片就浮在了空间里你点击的位置。



这只是 ARKit 的万千可能性之一，但的确是非常酷炫的体验。以上就是 ARKit 的使用。

追踪质量

刚刚的 demo 使用了 ARKit 的追踪功能，现在来讨论如何获得最佳质量的追踪结果。

- 追踪依赖于源源不断的传感器数据。这表示如果不再提供相机画面，追踪就会停止。
- 追踪在良好纹理的环境中会获得最佳工作状态。这表示场景从视觉上来说需要足够复杂，以便从相机画面中找到特征点。所以如果你对着一张白墙，或房间里光线不足，可能就无法找到特征点了，追踪功能就会受限。
- 追踪在静止场景中会获得最佳工作状态。所以如果相机里的大部分东西都在移动，视觉数据无法对应运动数据，就会导致漂移，这同样也会限制追踪状态。

为了应对这些情况，ARCamera 提供了 tracking state 属性。

tracking state 有三个可能值：Not Avaiable 不可用，Normal 正常，以及 Limited 受限。新的 session 会从 Not Avaiable 开始，表示摄像头的 transform 为空，即身份矩阵（identity matrix）。一般很快就会找到第一个追踪姿态（tracking pose），状态会从 Not Avaiable 变为 Normal，表示现在可以用摄像头的 transform 了。如果后面追踪受限，追踪状态会从 Normal 变为 Limited，而且会告诉你原因。例如用户面对一面白墙，或没有足够的光线，也就是特征不足。这时应该告知用户。所以，Apple 提供了一个 session delegate 方法供我们实现：

```
func session(_ session: ARSession, cameraDidChangeTrackingState camera:
ARCamera) {
    if case .limited(let reason) = camera.trackingState {
    // 告知用户追踪状态受限
    ...
    }
}
```

此时可以获得追踪状态，如果受限的话还会告诉你原因。应该把原因告知用户。因为只有他们才能真正修复追踪状态，要么开灯，要么别面对白墙。还有一种可能是传感器数据不可用。对于这种情况，应通过 session interruptions 来处理。

Session Interruptions

如果摄像头输入不可用，主要原因是 app 进入后台或在 iPad 上做多任务，session 也就无法获得相机画面。在这种情况下，追踪会不可用或停止，session 也会被终止。为了应对这种情况，Apple 为我们提供了方便的 delegate 方法：

```
func sessionWasInterrupted(_ session: ARSession) {
    showOverlay()
}

func sessionInterruptionEnded(_ session: ARSession) {
    hideOverlay()
    // 选择性重新开始整个体验
    ...
}
```

此时最好能将屏幕覆盖或模糊，以便告知用户当前体验已被暂停，也没有进行追踪。中断时一定要重点注意，由于没有进行追踪，设备的相对位置也就无法使用。如果用户移动了，当前的 anchor 或场景中的物理位置可能就无法像原来一样排布。对于这种情况，可能需要选择性重新开始整个体验。

以上就是追踪功能。下面讨论一下场景理解。

场景理解

场景理解的目标是找出环境中更多有关信息，以便在此环境中放置视觉对象，包括环境的 3D 拓扑以及光照情况等信息。

来看个例子，这是一张桌子。如果想在这张桌上放一个虚拟对象，首先要知道那儿有可以放东西的表面。

- 这一步可以通过平面检测实现。
- 下一步要找出放置虚拟对象的3D坐标系，这一步可以通过命中测试实现。也就是从设备发送光线，使之与现实世界相交，以便找出此坐标系。
- 为了以更真实的方式放置物体，需要估算光线以匹配环境中的光线。

下面依次讲解上面的三个步骤。

平面检测

- 平面检测可以提供相对于重力的水平面。包括地面以及类似桌子等平行平面。
- ARKit 会在后台聚合多个帧的信息，所以当用户绕着场景移动设备时，它会掌握更多有关平面的信息。
- 平面检测还能校准平面的边缘，即在平面所有检测到的部分四周套上一个矩形，并将其与主要区域对齐。所以从中也能得知物理平面的主要角度。
- 如果同一个物理平面检测到了多个虚拟平面，ARKit 会负责合并它们。组合后的平面会扩大至二者范围，因此后检测的那些平面就会从 session 中移除。

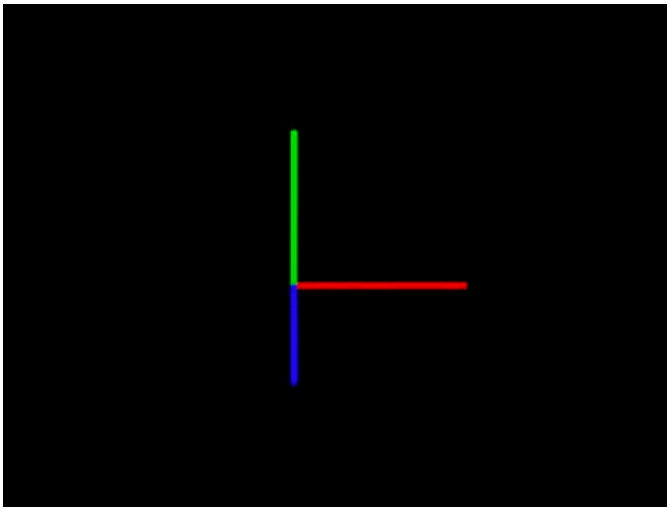
代码实现

```
// 启用 session 的平面检测


// 创建新的 world tracking configuration
let configuration = ARWorldTrackingSessionConfiguration()

// 启用平面检测
configuration.planeDetection = .horizontal

// 改变运行中 session 的 configuration
mySession.run(configuration)
```

首先创建一个 ARWorldTrackingSessionConfiguration。平面检测 planeDetection 是 ARWorldTrackingSessionConfiguration 的一个属性。要启用平面检测，只要设置 planeDetection 属性为 horizontal 即可。然后调用 ARSession 的 run 方法，用 configuration 作为参数，就会开始检测环境中的平面。如果想关掉平面检测，只要设置 plane detection 属性为 None，然后再次对 ARSession 调用 run 方法即可。session 中之前检测到的平面都会保留下来，也就是还存在于 ARFrames anchors 中。每当新的平面被检测到时，它们会以 ARPlaneAnchor 形式表示。

ARPlaneAnchor

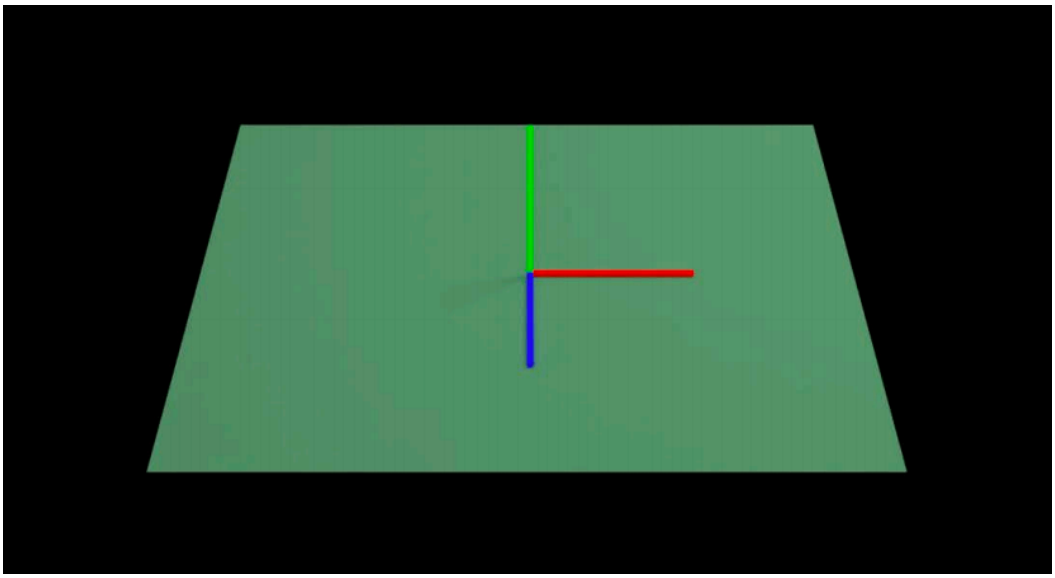
ARAnchor 用于表示真实世界中的位置和角度，而 ARPlaneAnchor 是它的子类。



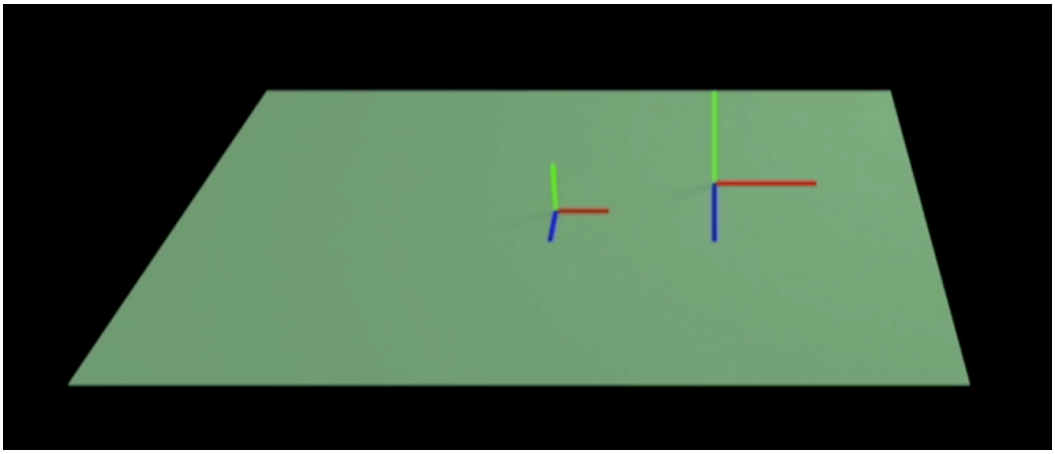
检测到新的 anchor 时，会调用 delegate 方法：

```
// 检测到新平面时调用
func session(_ session: ARSession, didAdd anchors: [ARAnchor]) {
    addPlaneGeometry(forAnchors: anchors)
}
```

此方法可以用于如视觉化平面。Extent 就是此平面的范围，以及一个相对的 center 属性。



如果用户绕着场景移动设备，对平面的了解会增多，所以范围 extent 可能也会相应改变。



此时会调用 delegate 方法：

```
// plane 的 transform 或 extent 变化时调用
func session(_ session: ARSession, didUpdate anchors: [ARAnchor]) {
    updatePlaneGeometry(forAnchors: anchors)
}
```

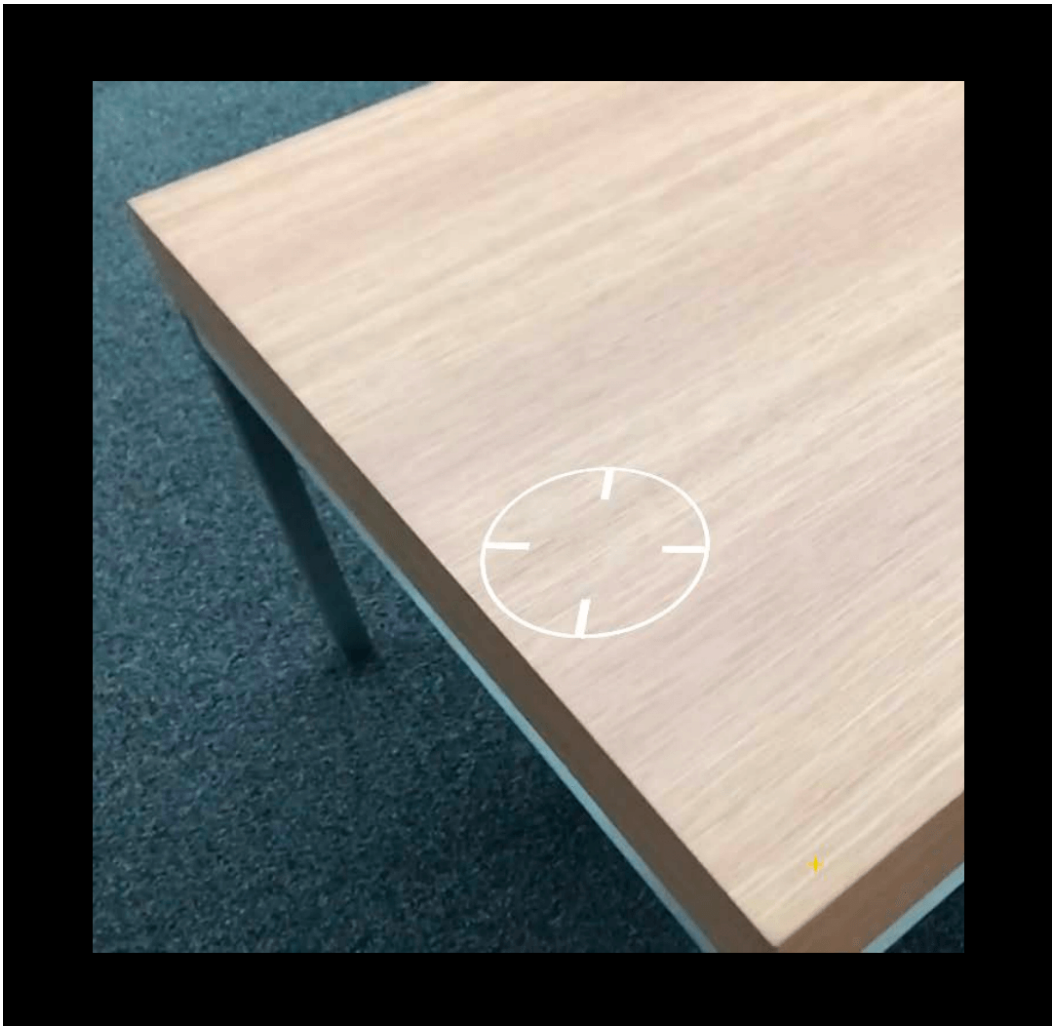
可以使用此方法更新视觉效果。注意 center 也会相应发生改变，因为平面会向某个方向扩张。从 session 中移除 anchor 时，会调用 delegate 方法：

```
// 合并时移除 plane 会调用
func session(_ session: ARSession, didRemove anchors: [ARAnchor]) {
    removePlaneGeometry(forAnchors: anchors)
}
```

如果 ARKit 合并了平面并移除了之前的小平面时会调用此方法，可以相应更新视觉效果。

现在我们知道了环境中都有哪些平面，下面看看如何实际地放点东西进去。这一步要使用命中测试。

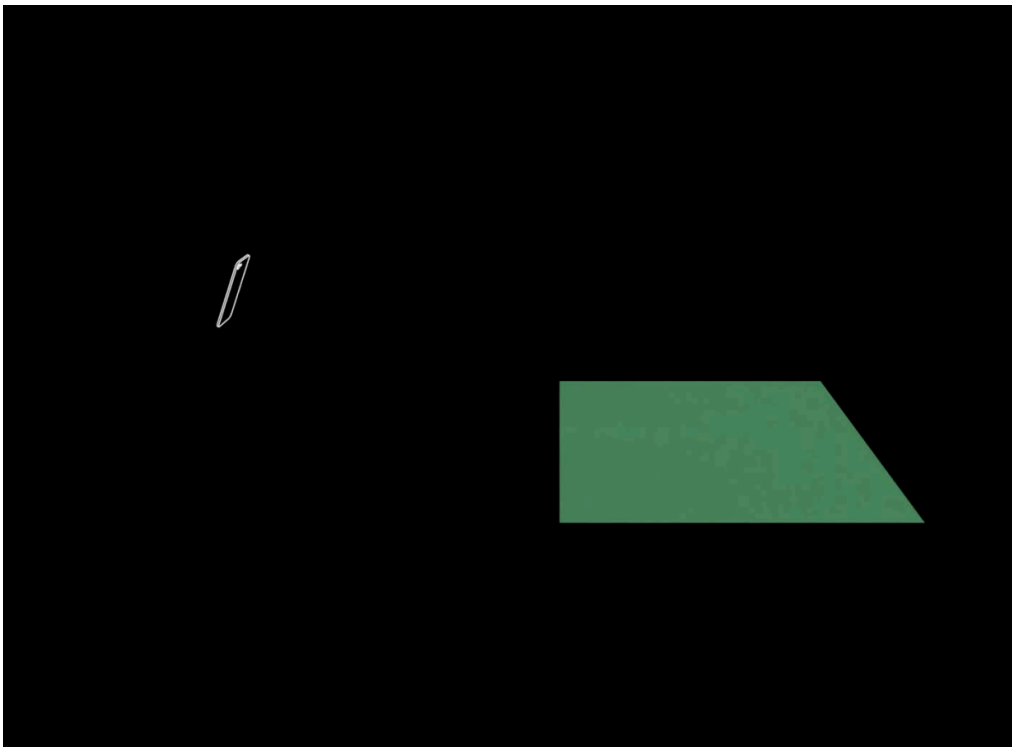
命中测试



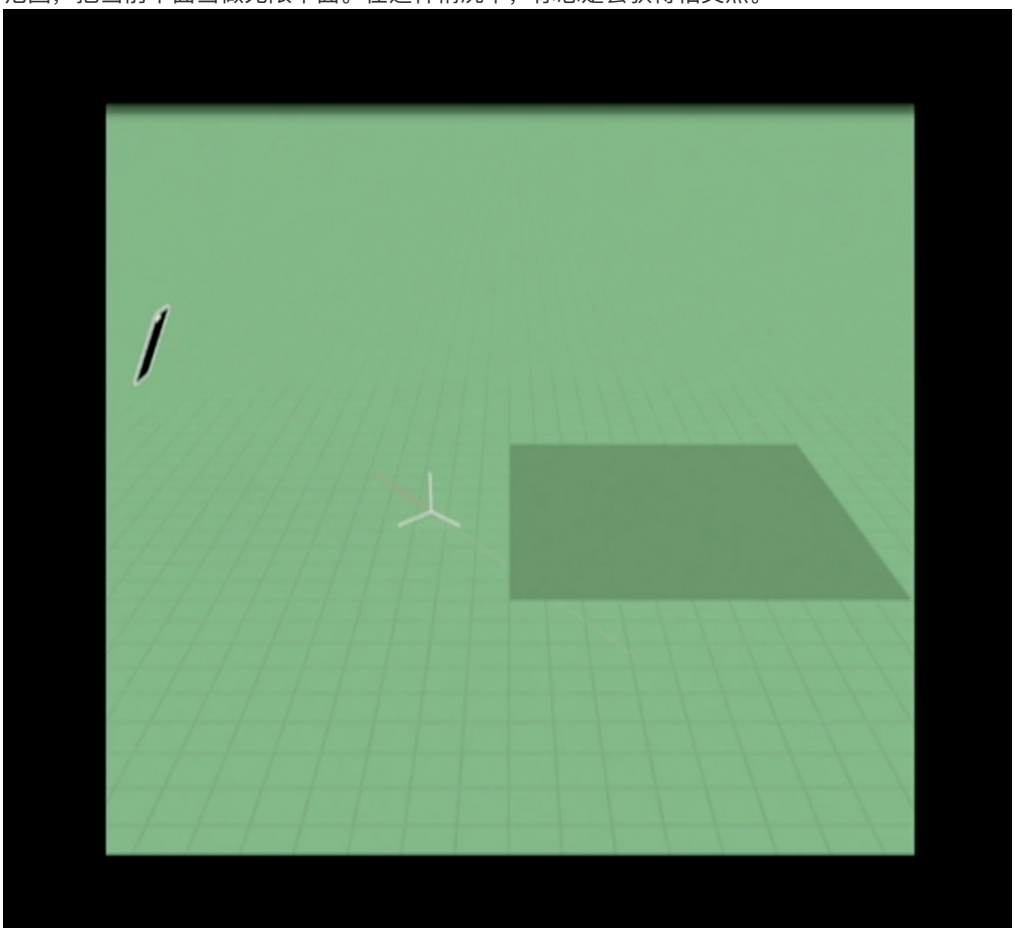
- 命中测试就是从设备发送一条光线，并与真实世界相交并找到相交点。
- ARKit 会使用所有能用的场景信息，包括所有检测到的平面以及3D特征点，这些信息都被 ARWorldTracking 用于确定位置。
- ARKit 然后发射光与所有能用的场景信息相交，然后用数组返回所有相交点，以距离升序排序。所以该数组的第一个元素就是离摄像头最近的相交点。
- 有不同的相交方式。可以通过 hit-test type 来定义。一共有四种方式，来具体看一看。

命中测试类型

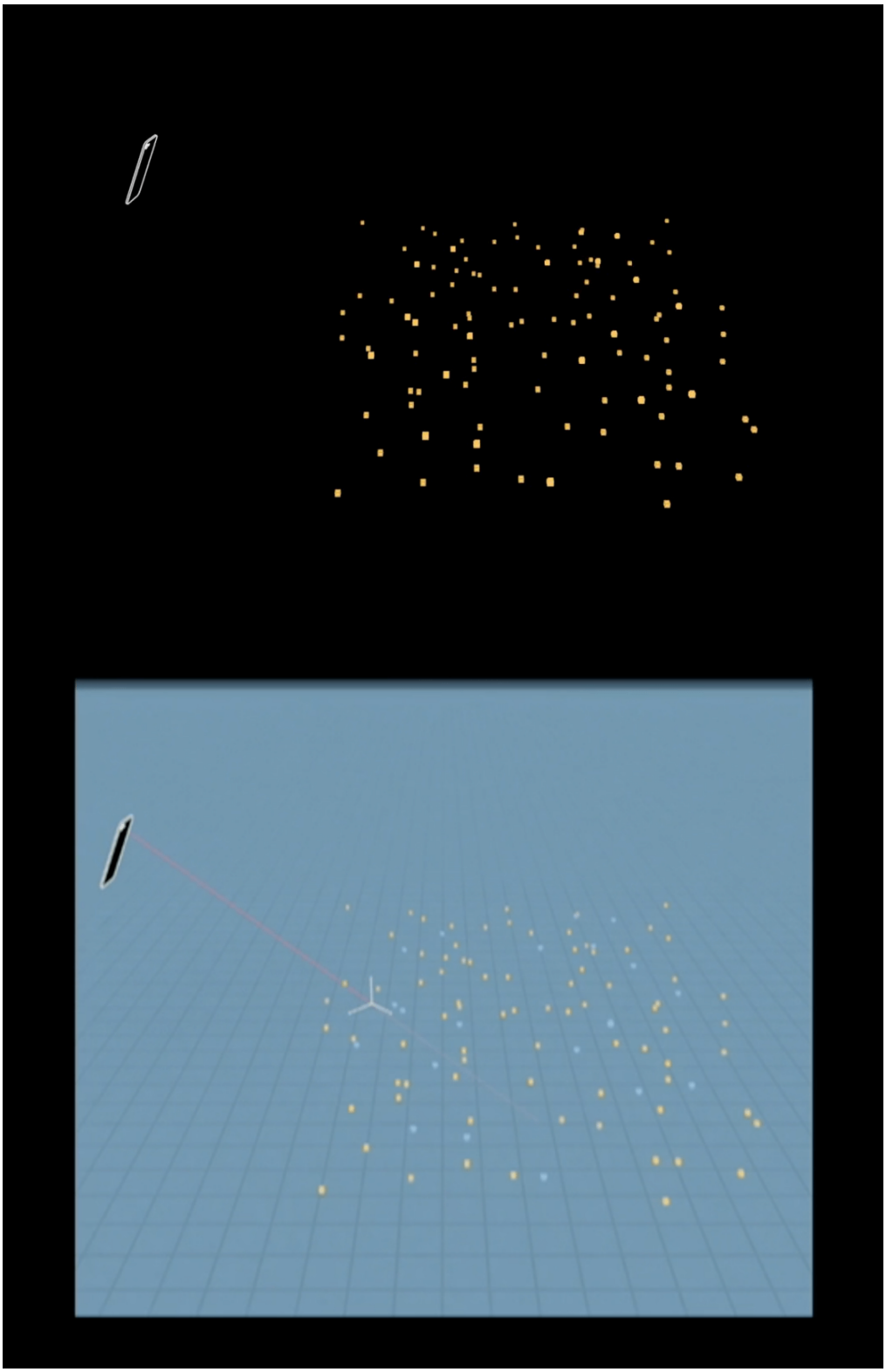
- Existing plane using extent: 如果在运行平面检测，并且 ARKit 已在环境中检测到了某个平面，就可以利用此平面。但你可以选择使用平面的范围或忽视它的范围。也就是说，例如你想让用户在某个平面上移动对象，就应该考虑范围，所以若光在范围内相交，就会产生一个相交点。如果这束光打到了范围的外面，就不会产生相交点。



- Existing plane: 但如果你只检测到了地面的一小部分，但希望来回移动家具，就可以选择忽略范围，把当前平面当做无限平面。在这种情况下，你总是会获得相交点。



- Estimated plane: 如果没有在运行平面检测或者还没有检测到某个平面，也可以根据目前的 3D 特征点来估算平面。在这种情况下，ARKit 会寻找环境中的处于共同平面的点并为它们安装一个平面。随后也返回与此平面的相交点。



- Feature point: 如果你想在某个很小的表面上放东西，但此表面无法生成平面，或者是某个非常不规则的环境，也可以选择直接和特征点相交。也就是说光线会与特征点产生相交点，并将距离最近的特征点其作为结果返回。

代码实现

```
// 根据命中测试添加 ARAnchor
let point = CGPoint(x: 0.5, y: 0.5) // 画面中心

// 对帧执行命中测试
let results = frame.hitTest(point, types: [.existingPlane,
.estimatedHorizontalPlane])

// 使用第一个结果
if let closestResult = results.first {

    // 用它创建 ARAnchor
    let anchor = ARAnchor(transform: closestResult.worldTransform)

    // 添加到 session
    session.add(anchor: anchor)

}
```

首先用 CGPoint 来定义从设备发射的光线，以标准画面空间坐标系表示。也就是说画面左上角是 (0,0)，右下角是 (1,1)。所以如果我们想让光从屏幕中心发射，就用 (0.5,0.5) 来定义 CGPoints。对于 SceneKit 或 SpriteKit，Apple 提供了自定义 overlay，只要发送这些坐标系中的 CGPoint 即可。所以可以通过 touch gesture 使用 UI tap 的结果作为输入来定义此光线。将此点用作 histTest 方法的参数，同时还有一个参数是命中测试类型。本例里用的是 existingPlane，表示会与ARKit 目前检测到的所有平面相交，同时还有 estimatedHorizontalPlane 可用作没有检测到平面时的备选方案。然后 ARKit 会返回结果数组。访问第一个结果，也就是离相机最近的相交点。用命中测试结果的 worldTransform 属性来创建一个新的 ARAnchor，并将其添加到 session 以便持续追踪。

如果将以上代码应用到下面的场景中，然后把手机对准桌子，就会返回屏幕中间与桌子的相交点。

然后在此位置放一个虚拟茶杯。

渲染引擎会默认背景画面有完美的光照条件。所以这增强现实看起来就像真的一样。但是如果你更暗的环境中，相机画面就会变暗，这是增强现实看上去就有一点出戏，好像在发光似的。

此时就需要调整虚拟对象的相对亮度。

因此就需要光线估算。

光线估算

- 光线估算需要使用摄像头画面，借助曝光信息来决定相对亮度。
- 对于光照良好的画面，默认为 1000 流明。对于更亮的环境，会得到更高的值。对于更暗的环境，会得到更低的值。如果是物理光源，也可以直接把这个值分配给 ambientIntensity 属性。
- 光线估算是默认启用的。可以通过 ARSessionConfiguration 的 isLightEstimationEnabled 属性进行配置：

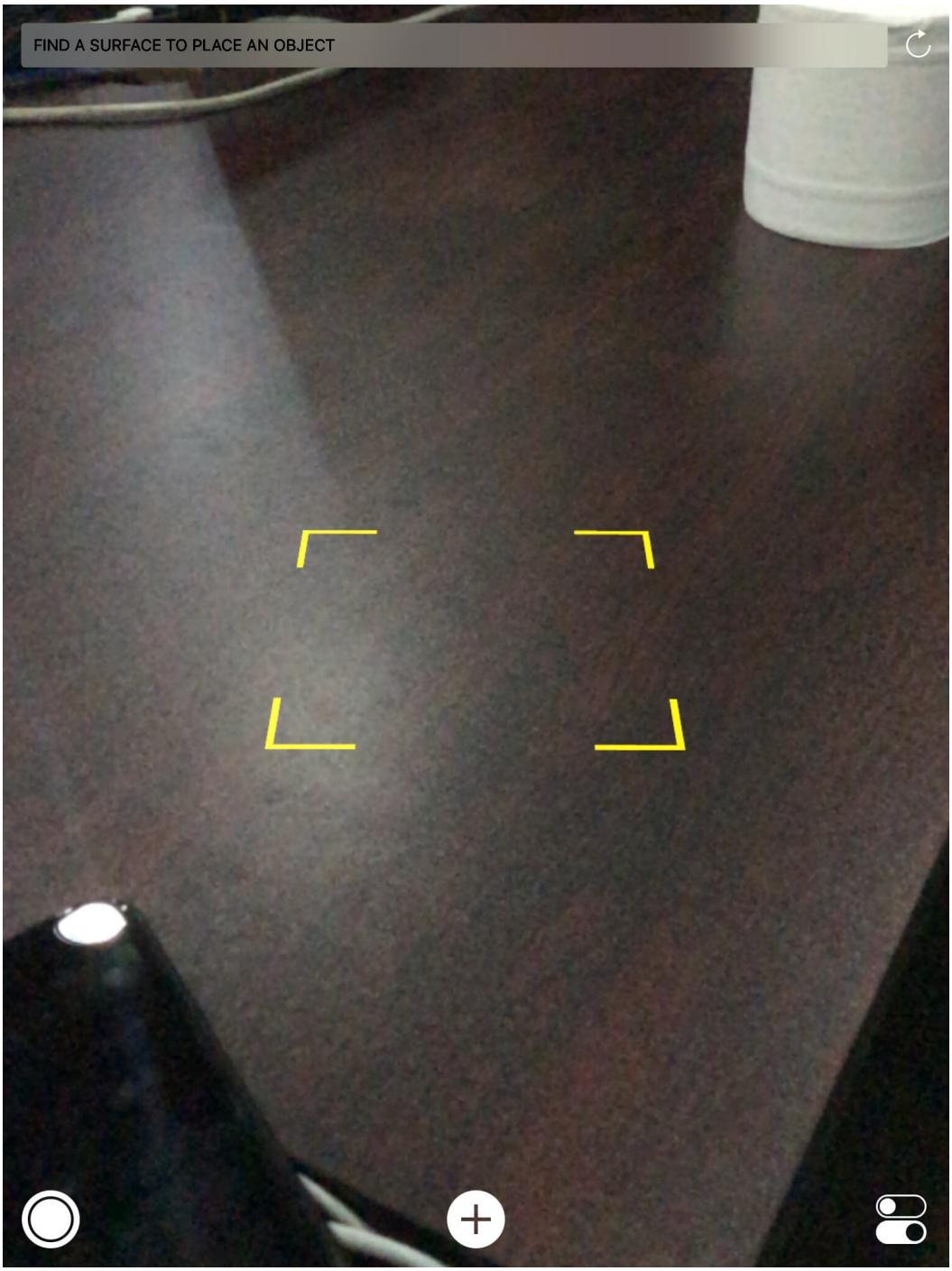
```
configuration.isLightEstimationEnabled = true
```

光线估算的结果以 ARFrame 的 lightEstimate 属性的 ambientIntensity 值表示：

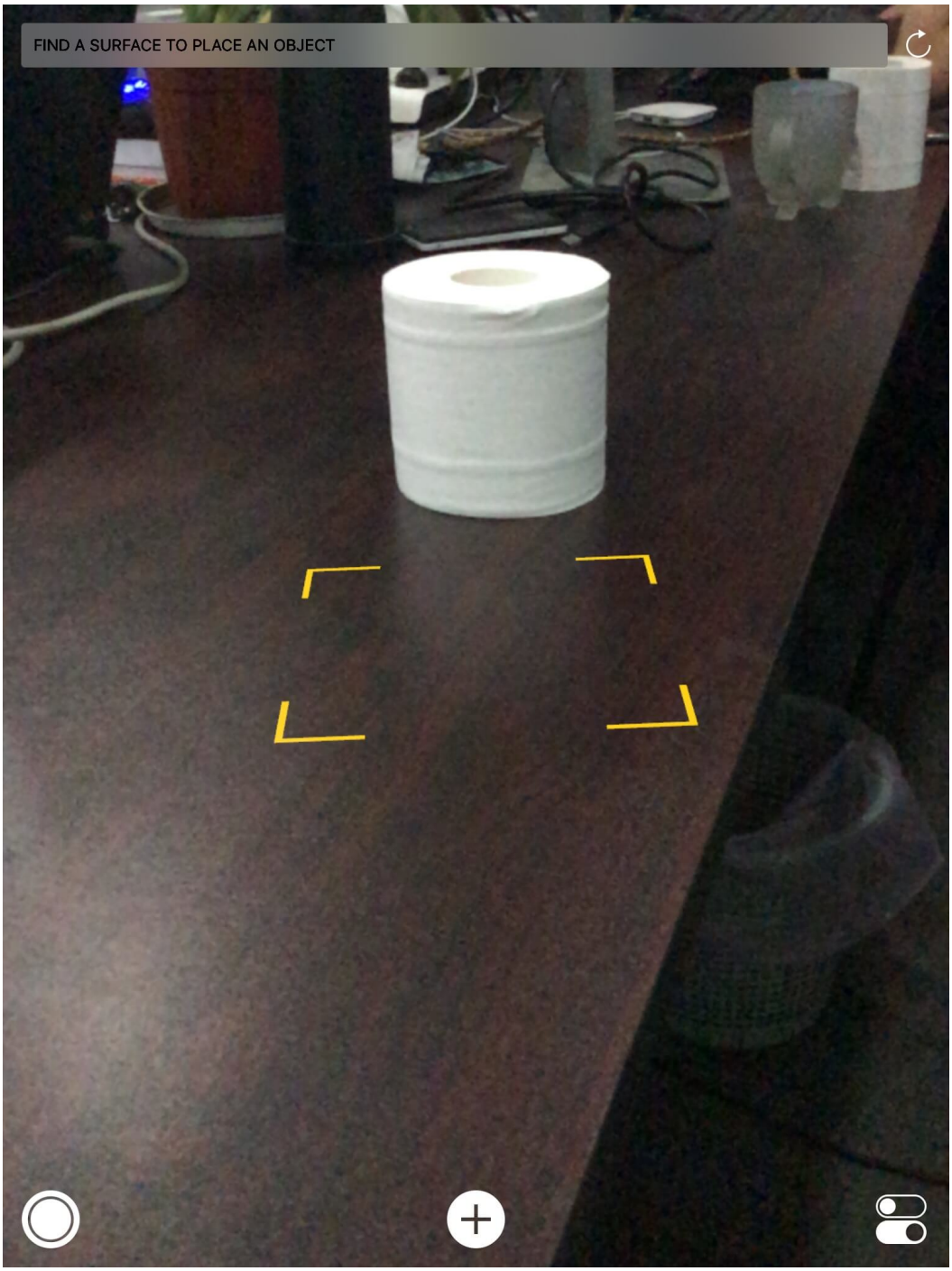
```
// 获取 ambient intensity 值
let intensity = frame.lightEstimate?.ambientIntensity
```

demo

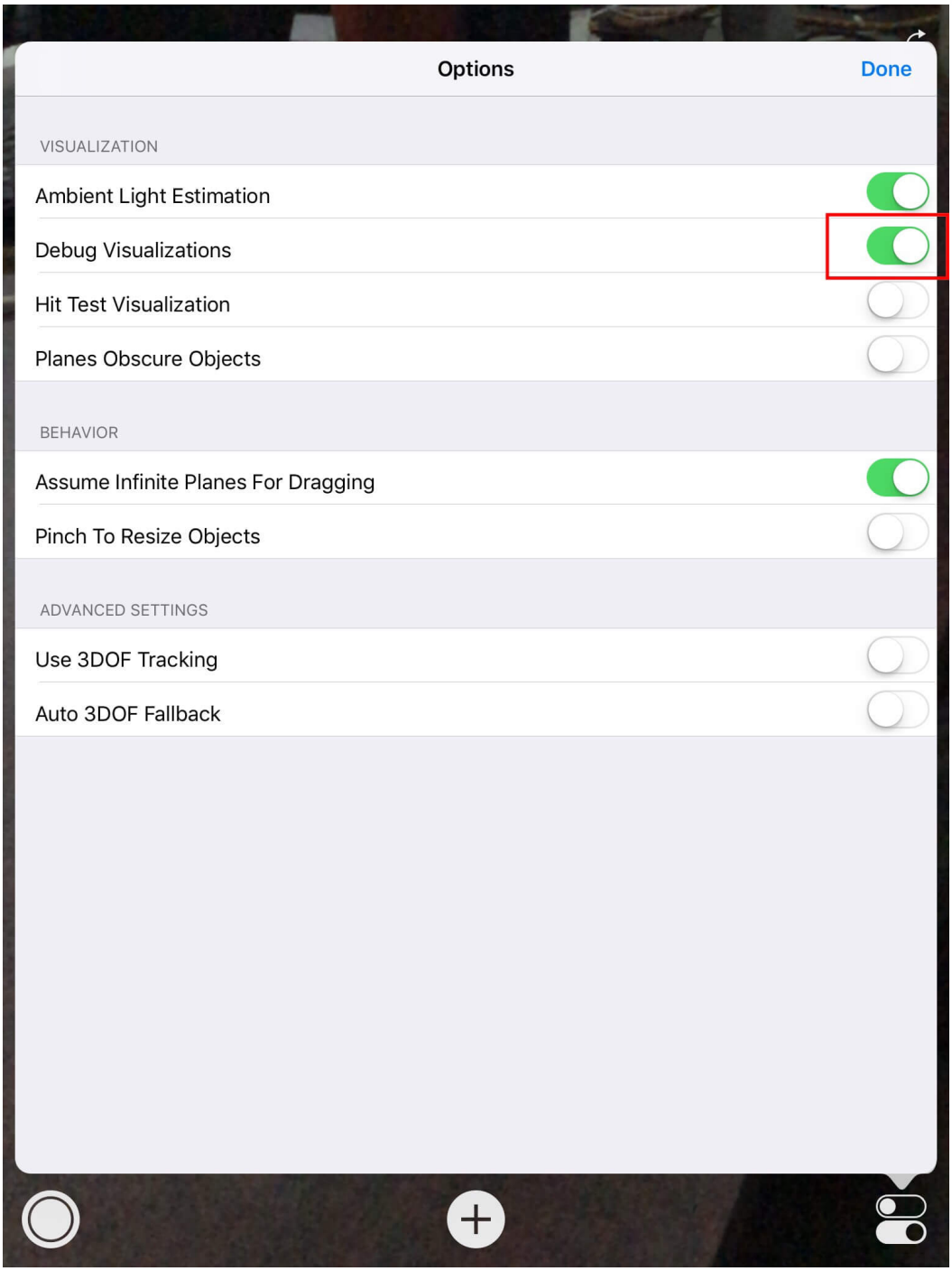
下面来实际看一个 demo，了解如何使用 ARKit 中的场景理解。这个 demo 是 ARKit 示例应用，可以从 Apple 的开发者网站下载。它利用场景理解功能，以便在环境中放置对象。打开之后，在桌子上来回移动，就可以看到一个对焦矩形。



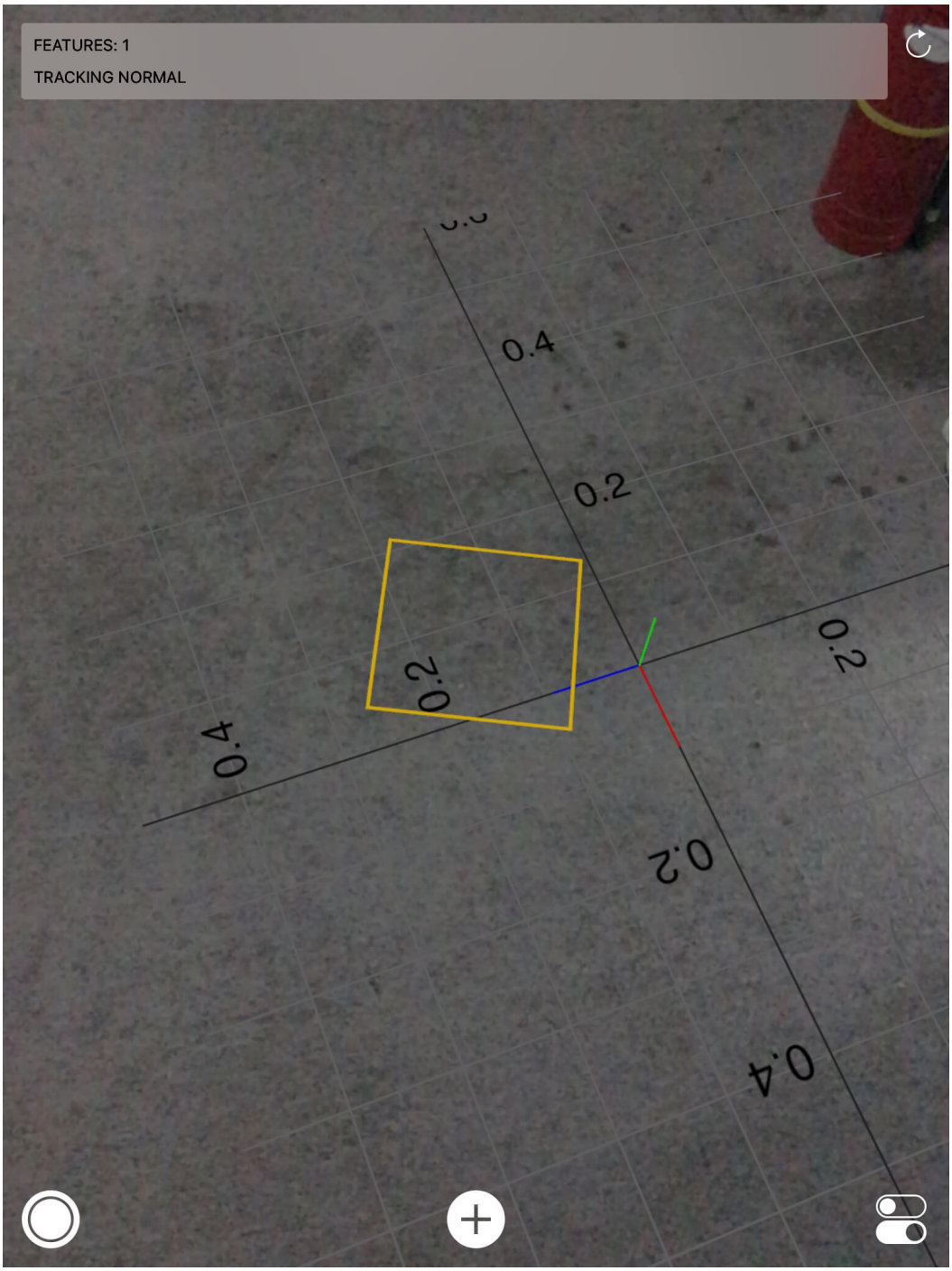
这一步就是在屏幕中心的位置做命中测试，以便找到相交点来放置物体。所以如果我对着桌子移动，这个矩形也会一起在桌子上滑动。



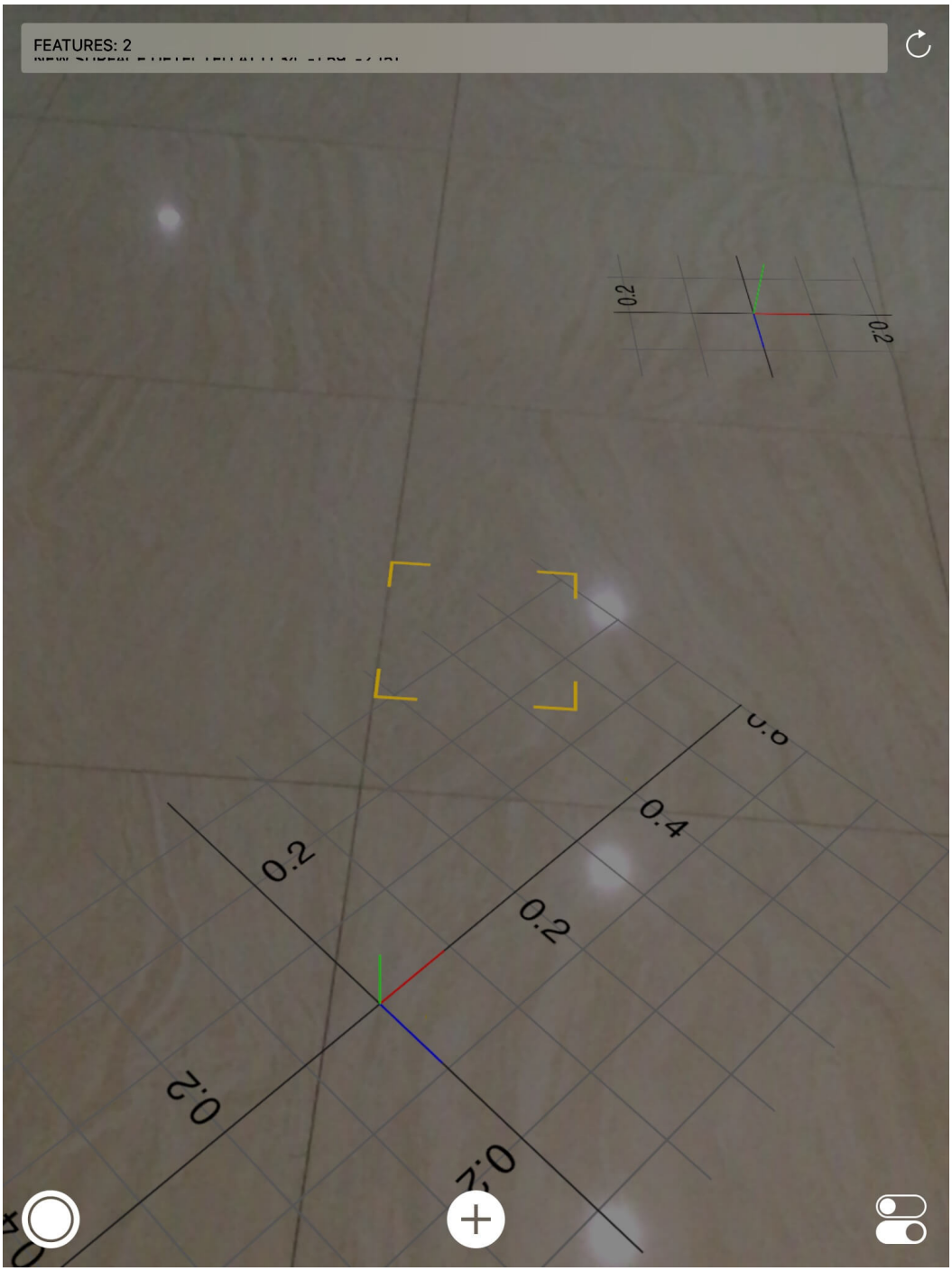
这一步还使用了平面检测，我们可以让平面检测的过程可视化，以便直观了解正在发生的事情。打开这里的 Debug 菜单并激活第二个选项，即 Debug Visualizations。



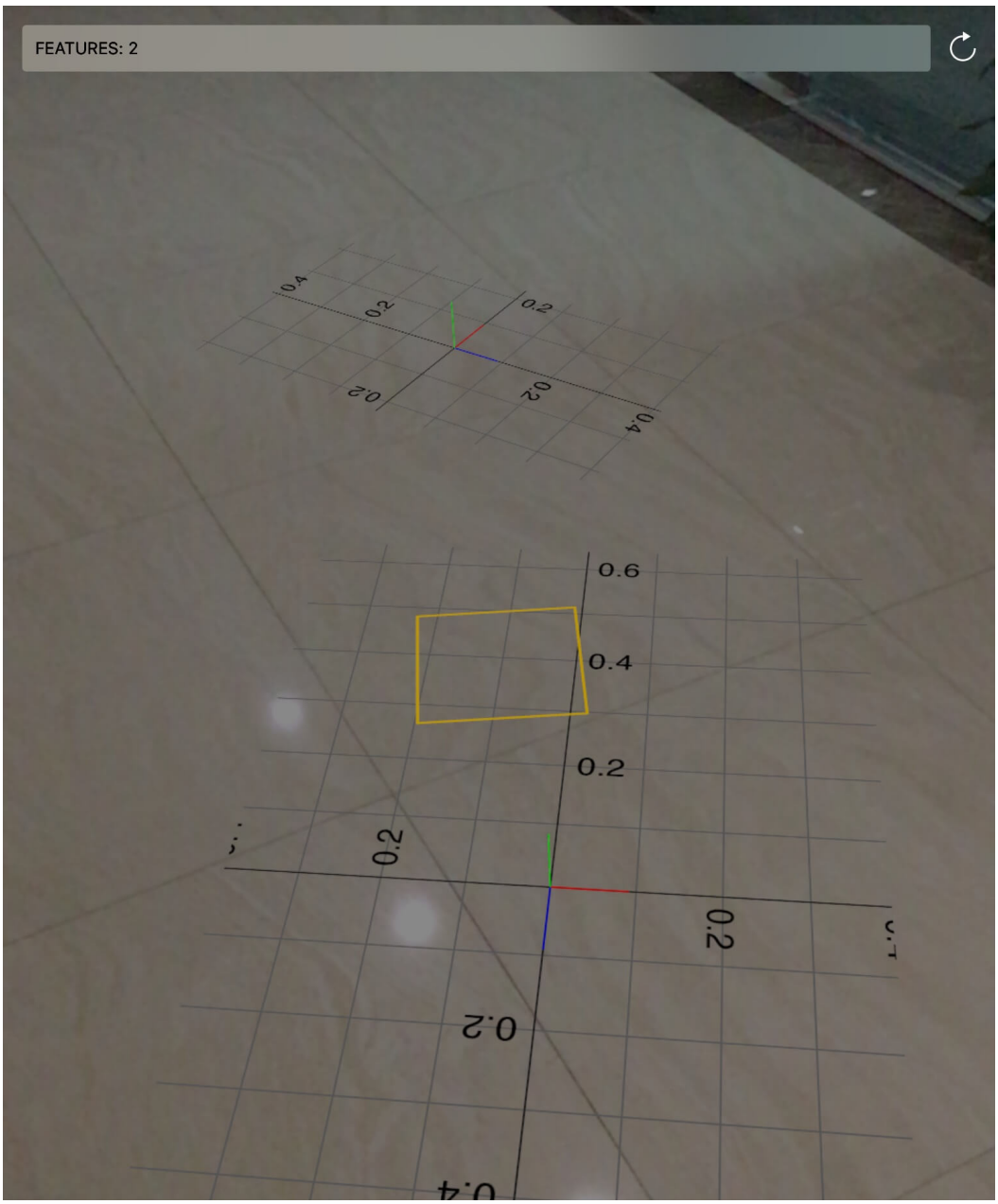
然后关闭这个菜单。就可以看到被检测到的平面了。

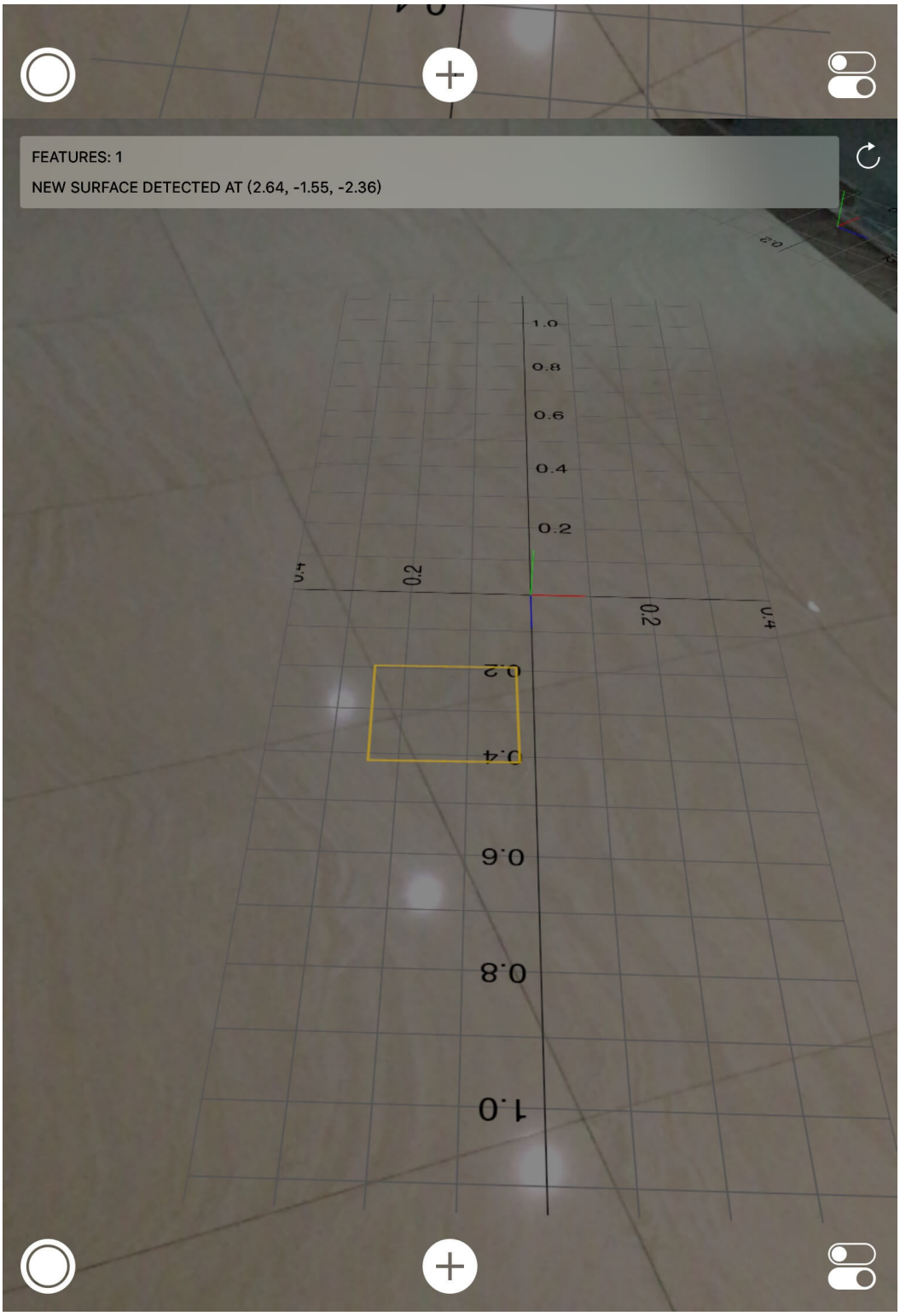


为了更好地理解新平面的检测过程，我们来重新检测一个平面。如果我指向一个新的平面，然后迅速指向这个平面的另一个位置，就会有两个平面被检测到：

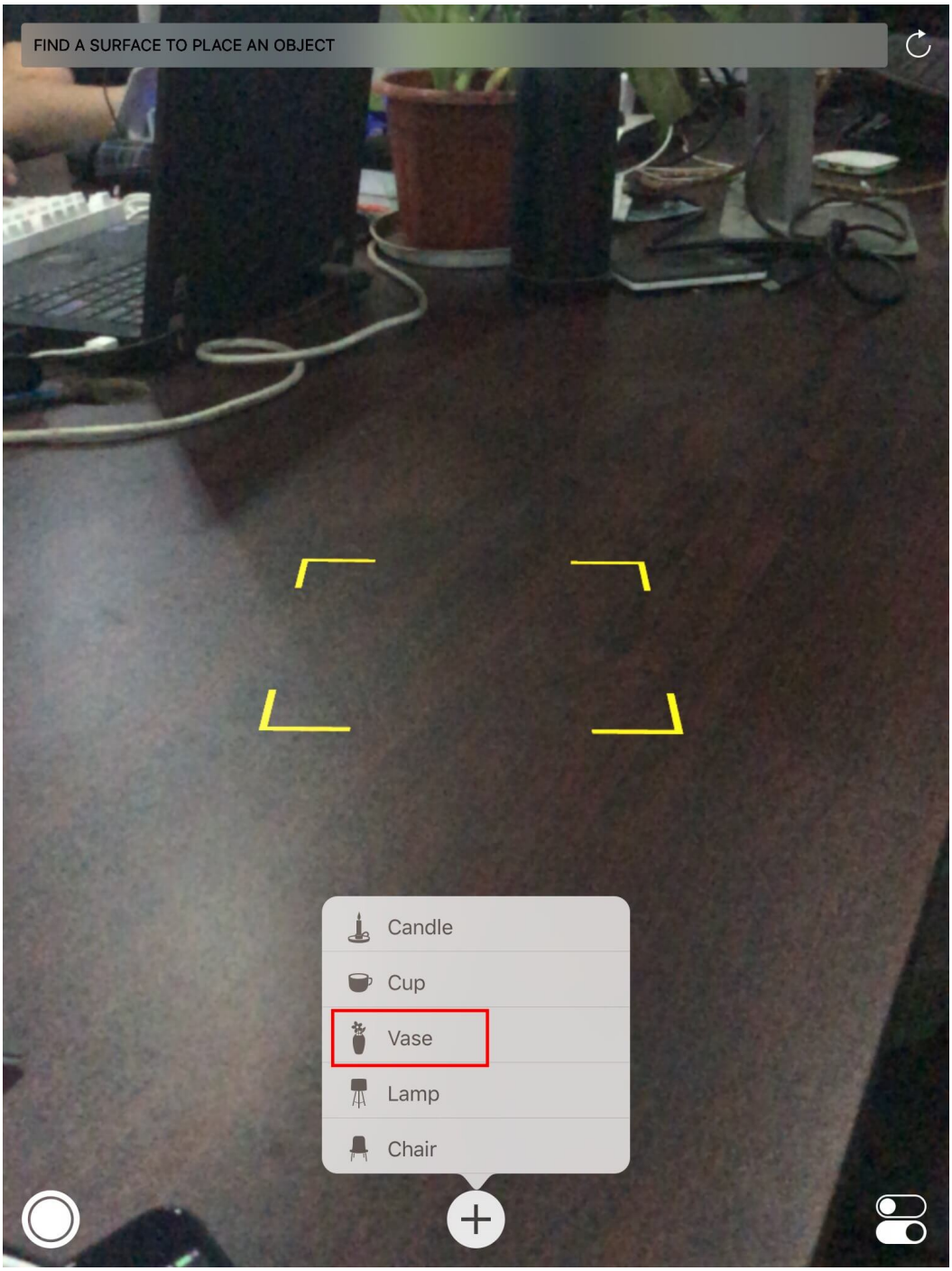


但如果我顺着这个平面移动，ARKit 会发现其实这里只有一个平面，所以这两个平面最终会合并为一个平面。


FEATURES: 2
0.6
0.4
0.2
0.2
0.2
0.2
0.2
0.4



下面，我们来实际放置一些物体。女朋友让我在办公桌上放一些鲜花，我不想让她失望，所以给这里来一个浪漫的花瓶。



点击对焦矩形，并选择添加“Vase”。

这里就是就是用屏幕中心进行命中测试并找到相交点，然后放置物体。重点要注意，这个花瓶是以真实世界的比例出现的，这是因为以下两点：1、世界追踪为我们提供了缩放比例；2、3D 模型是在真实世界坐标系中构建的。所以要为增强现实创建内容的话，一定要考虑第2点，花瓶不应该和一栋建筑物一样高，也不应该太小。

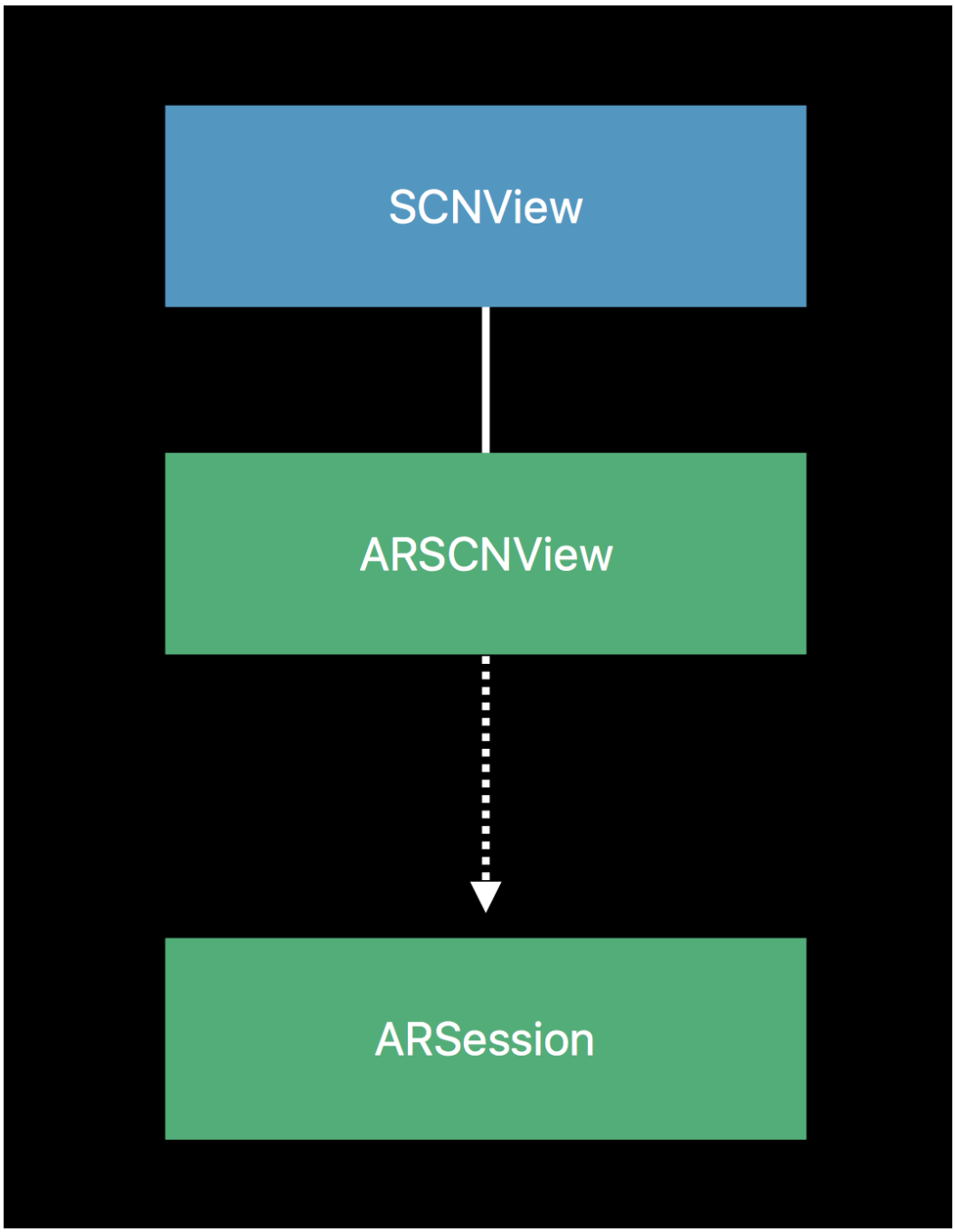
以上就是示例程序。可以从 Apple 的网站上下载并试着放入自己的内容。

渲染

下面我们来看看如何用 ARKit 进行渲染。渲染需要追踪、场景理解以及你的内容。要用 ARKit 渲染，需要处理 ARFrame 中提供的所有信息。

对于 SceneKit 和 SpriteKit，Apple 提供了可定制化的视图来为你处理 ARFrame 并进行渲染。而对于 Metal，可以创建自己的渲染引擎或将 ARKit 整合进目前的渲染引擎，Apple 提供了一张模板与它的使用介绍，使用此模板是一个很好的出发点。下面挨个讲解它们。

SceneKit



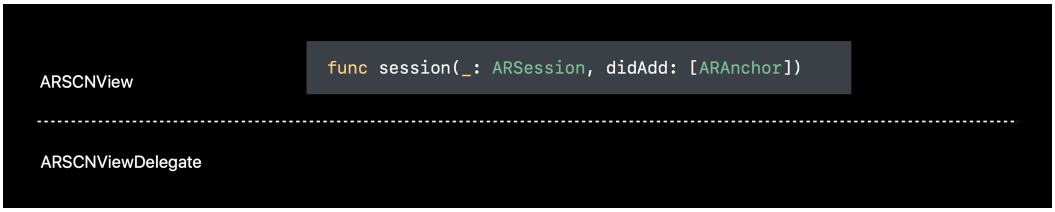
对于 SceneKit，Apple 提供了 ARSCNView，它是 SCNView 的子类，包含一个 ARSession，用于更新渲染。

ARSCNView

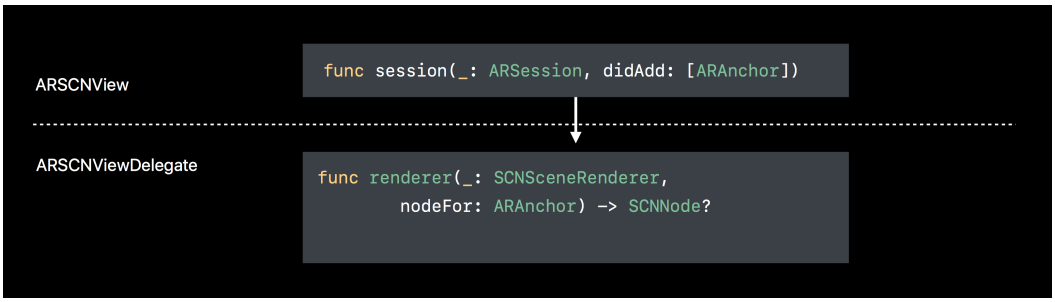
- ARSCNView 会绘制摄像头背景画面。
- ARSCNView 会根据 ARCamera 中的追踪 transform 更新 SCNCamera。场景保持不变，ARKit 只是控制 SCNCamera 在场景中移动，模拟用户在现实世界中来回移动设备。
- 使用光线估算时，ARSCNView 会自动在场景中放置一个 SCNLight 来更新光照。
- ARSCNView 会将 SCNNodees 映射到 ARAnchors，所以实际上不需要直接操作 ARAnchors，使用 SCNodees 即可。当一个新的 ARAnchor 被添加到 session 时，ARSCNView 也会创建一个 node。每次更新 ARAnchor 时，例如改变 transform，也会自动更新 nodes 的 transform。这一步是通过 ARSCNView delegate 来实现的。

ARSCNViewDelegate

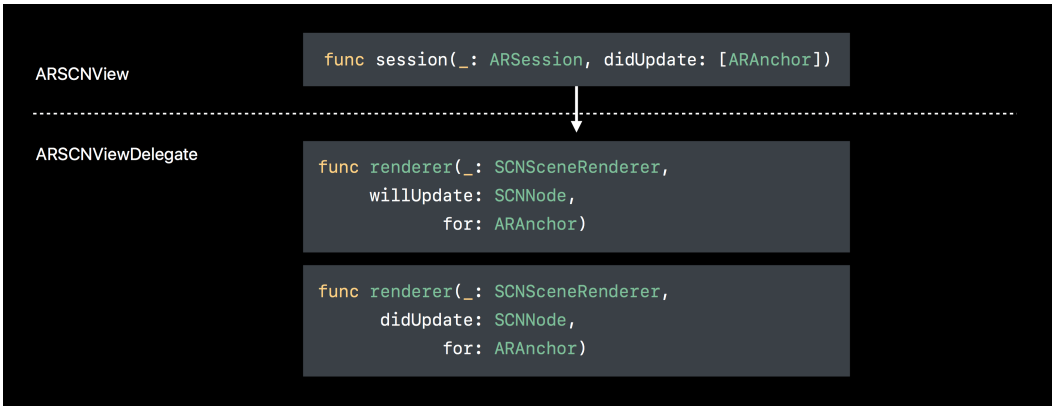
session 每次添加新的 anchor 时，ARSCNView 都会创建一个新的 SCNNode。如果想用自定义 node，可以实现 renderer nodeFor anchor 方法并返回自定义 node。



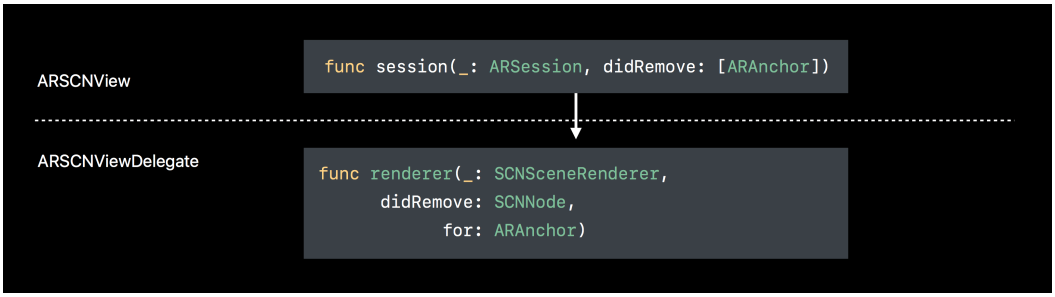
然后 SCNNode 会被添加到场景中，此时会接收另一个 delegate 方法：



node 被更新时同样也会接收 delegate 方法，例如 ARAnchor 的 transform 改变时，DSCNNode 的 transform 也会自动改变，此时会收到两个回调。transform 更新前一个，更新后一个。



从 session 中移除 ARAnchor 时，也会自动从场景中移除对应的 SCNNode，并调用 renderer didRemove node for anchor：



以上就是 ARKit 中的 SceneKit。下面来看 SpriteKit。

SpriteKit

对于 SpriteKit，Apple 提供了 ARSKView，它是 SKView 的子类。

ARSKView

- ARSKView 包含 ARSession，用于更新渲染。
- ARSKView 会绘制摄像头背景画面。
- ARSKView 会将 SKNodes 映射到 ARAnchors。

ARSKView 提供的 delegate 方法与 SceneKit 很相似。主要的区别是 SpriteKit 是 2D 渲染引擎，这表示不能简单地移动摄像头，其实这里 ARKit 是将 ARAnchor 的位置投影到 SpriteKit 视图上，然后把 Sprites 渲染为投影位置上的广告牌（billboard），也就是说 Sprites 总是会面对摄像头。

如果想更多了解相关内容，可以去看来自 SpriteKit 团队的 session，“Going beyond 2-D in SpriteKit”，会讲如何整合 ARKit 和 SpriteKit。

下面来看看如何借助 Metal 在 ARKit 中实现自定义渲染。

自定义渲染

处理流程

ARKit 的渲染主要要做四件事。

- 绘制摄像头背景画面。通常需要创建纹理并绘制在背景。
- 根据 ARCamera 更新虚拟摄像头。需要设置视图矩阵和投影矩阵。
- 根据光线估算更新场景中的光线。
- 如果基于场景理解放置了几何体，使用 ARAnchor 来正确设置 transform。

这些所需的信息都被包含在 ARFrame 中。有两种方式获取 ARFrame。

访问 ARFrame

第一种方式是通过 ARSession 上的 currentFrame 属性。

```
if let frame = mySession.currentFrame {
    if( frame.timestamp > _lastTimestamp ) {
        updateRenderer(frame) // 用此帧更新渲染程序
        _lastTimestamp = frame.timestamp
    }
}
```

如果有自己的渲染循环，你可以使用此方法来访问 currentFrame。同时还要利用 ARFrame 的 timestamp 属性以避免多次渲染同一帧。

另一种方式是使用 Session Delegate，每次计算新的一帧时，都会调用 session didUpdate frame 方法：

```
func session(_ session: ARSession, didUpdate frame: ARFrame) {
    // 用此帧更新渲染程序
    updateRenderer(frame)
}
```

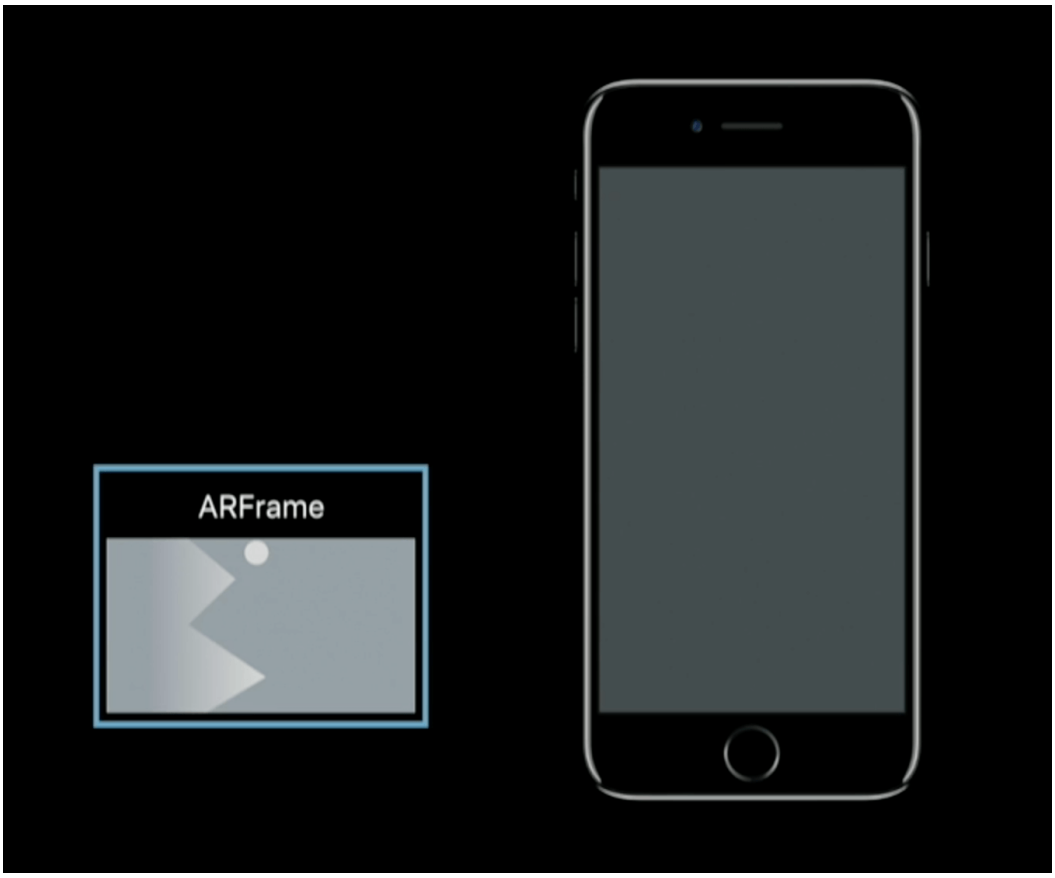
此时可以用它来更新渲染。此方法默认在主线程被调用，但你也可以提供自己的 dispatch queue 来调用它。下面来看看具体如何更新渲染。

```
func updateRenderer(_ frame: ARFrame) {

    // 绘制摄像头背景画面
    drawCameraImage(withPixelBuffer: frame.capturedImage)

    // 更新虚拟摄像头
    let viewMatrix = simd_inverse(frame.camera.transform)
    let projectionMatrix = frame.camera.projectionMatrix
    updateCamera(viewMatrix, projectionMatrix)

    // 更新光线
    updateLighting(frame.lightEstimate?.ambientIntensity)

    // 根据 anchors 更新几何体
    drawGeometry(forAnchors: frame.anchors)
}
```

首先要绘制摄像头背景画面。可以访问 AFFrame 上的 capturedImage 属性，这是 CVPixel Buffer。然后可以基于此 Pixel Buffer 生成 Metal texture，并在背景四边形中进行绘制。要注意由于这是通过 AV Foundation 暴露给我们的 Pixel Buffer，所以不应持有这些帧太多或太久，否则就会停止接收更新。下一步是根据 ARCamera 更新虚拟摄像头，因此需要确定视图矩阵以及投影矩阵。视图矩阵即 camera transform 的逆矩阵。Apple 在 ARCamera 上提供了一个 convenience 方法，以帮助我们生成投影矩阵。第三步是更新光线，访问 lightEstimate 属性并使用它的 ambientIntensity 值来更新光照模型。最后一步是遍历 anchors 和 anchors 的3D位置，并更新几何体的 transform，包括手动添加到 session 的 anchor 以及平面检测自动添加的 anchor。

绘制到 viewport

绘制摄像头画面时还有两点需要注意，一是 ARFrame 中的 captured iamge 总是以相同的角度提供。所以如果用户旋转了物理设备，画面可能对不上用户界面，此时需要应用 transform 来正确渲染。



二是摄像头画面的宽高比可能与设备不同。所以需要考虑这一点以便正确渲染屏幕中的的摄像头画面。

帮助方法

对于以上两点，Apple 提供了方便的帮助方法。ARFrame 有一个 displayTransform 方法：

```
// 给定 viewport 尺寸和角度并获得此帧的 display transform
let transform = frame.displayTransform(withViewportSize: viewportSize,
orientation: .portrait)
```

此方法可以获得从帧空间到视图空间的 transform。只要提供 view port 的尺寸以及界面角度，就会得到对应的 transform。在 Metal 那个例子里，使用此 transform 的逆矩阵来调整相机画面的纹理坐标。

```
// 给定 viewport 尺寸和角度并获得摄像头的投影矩阵
let projectionMatrix = camera.projectionMatrix(withViewportSize:
viewportSize,   orientation: .portrait,
zNear: 0.001, zFar: 1000)
```

同时还有投影矩阵方差，给它用户界面角度以及 view port 尺寸，并告知平面裁剪（clipping planes）限制，然后就可以用得到的投影矩阵在相机画面上正确绘制虚拟内容。以上就是关于 ARKit 的介绍。

总结

ARKit 是 high-level API，为在 iOS 上创建增强现实应用而设计。ARKit 提供 WorldTracking 功能，能够得到设备相对于起始位置的相对位置。为了在现实世界中放置物体，ARKit 还提供了场景理解功能。场景理解可以检测平面，也能够对真实世界进行命中测试来找到3D坐标系并在那里放置物体。同时为了提升增强内容的真实性，ARKit 提供了基于摄像头画面的光线估算。ARKit 还提供了与 SceneKit 和 SpriteKit 的定制化整合，如果想自己开发渲染引擎的话，同时还有一张 Metal 模板供你采用。

参考

- Introducing ARKit: Augmented Reality for iOS
- SceneKit: What's New
- Going beyond 2D with SpriteKit

终于 iOS 11 里，我们拥有了傻瓜化的交互式动画

回顾

我们先思考一个问题：iOS11 之前创建哪类动画最麻烦?

答：交互式动画和自定义的timingFunction动画。

无code无真相。我们先来看看早先版本的动画接口是如何实现交互式动画和自定义timingFunciton的。

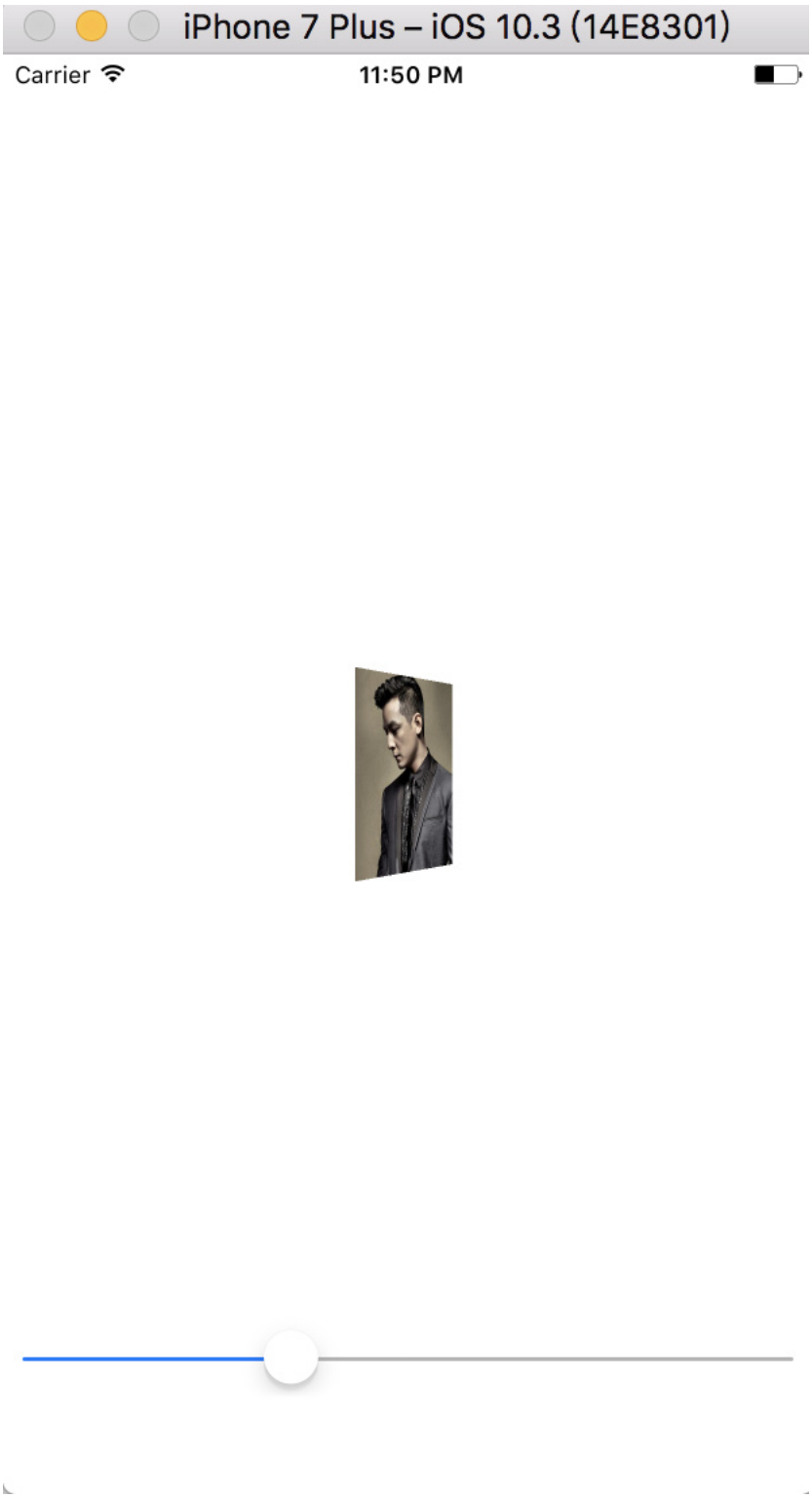
如何实现一个交互式动画?

大家知道，iOS里面动画的实现方式主要是两种，一种是UIViewAnimation和基于Layer层的CAAnimation。

两种动画的区别很多，当然，符合越底层的接口自由度越高的这个特点。CAAnimation的可定制性更强，但是在我看来，两种动画最主要的区别用一句话形容，就是.

`UIViewAnimation是开弓没有回头箭。CAAnimation是流星锤，可收可放。`

我们现在，就来实现一个用手势控制的动画。效果如图。



我们的目的是利用UISlider控制动画的进度，这个动画就是图片绕Y轴旋转。 代码如下。

```
class ViewController: UIViewController {

     let imageView = UIImageView.init(frame: CGRect.init(x: 0, y: 0, width:
100, height: 100))

    override func viewDidLoad() {
        super.viewDidLoad()

        imageView.image = UIImage.init(named: "wuyanzu.jpg")
        imageView.center = self.view.center
        imageView.layer.transform.m34 = -1.0/500
        self.view.addSubview(imageView)

        let basicAnimation = CABasicAnimation.init(keyPath:
"transform.rotation.y")
        basicAnimation.fromValue = 0
        basicAnimation.toValue = CGFloat.pi
        basicAnimation.duration = 1
        imageView.layer.add(basicAnimation, forKey: "rotate")
        imageView.layer.speed = 0
        // Do any additional setup after loading the view, typically from a
nib.
    }

    @IBAction func sliderValueChanged(sender:UISlider) {
        imageView.layer.timeOffset = CFTimeInterval(sender.value)
    }

}
```

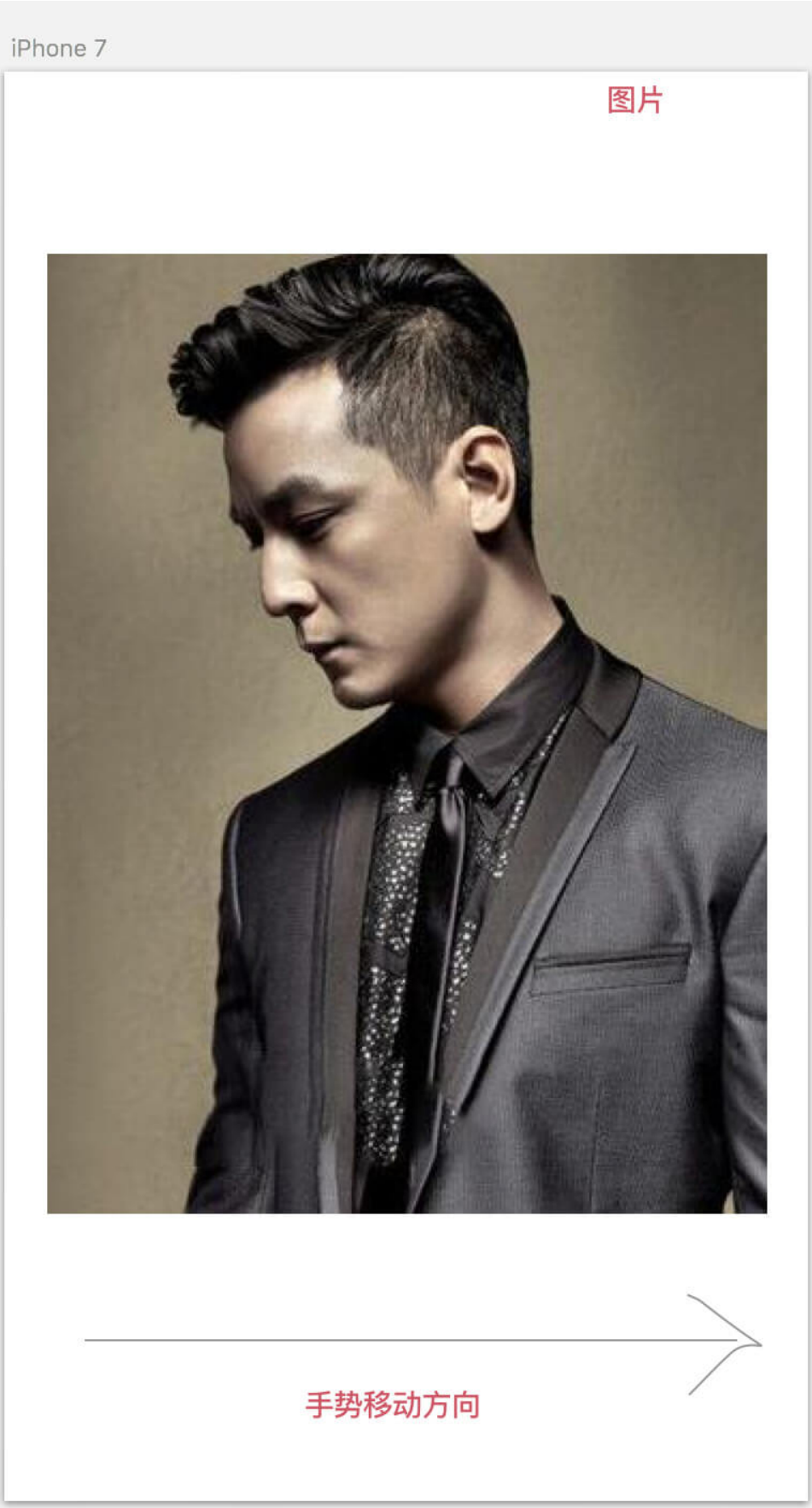
在iOS11之前，可交互动画的原理很简单。过程总结如下。

1. 将layer的speed设置为0，这样，动画就处于暂停状态
2. 利用timeOffset来控制整个动画的进度

再举个例子，如果这个动画不是利用UISlider控制旋转角度，而是利用PanGesture移动的距离来控制呢?

那么这种情况，你需要找到的就是手势的距离和Rotate动画timeOffset的一种关联。

我利用Sketch做了一个简陋的草图来模拟这种情况。



其实看完图片我们已经可以建立起手势移动距离和timeOffset的关联。 以横向移动为前提，那么`手指的x坐标/图片的width 总是 <= 1.0`,所以， 当旋转动画的总时长为1，那么动画的进度timeOffset就恰好等于x/imageView.width了。完美的关联了起来。

问题

我们也看到了这种处理方法的弊端。就是，实在太繁琐了。

所以，在今年的wwdc里，苹果为我们提供了一种非常方便的解决方案。

UIViewPropertyAnimator

其实在iOS10，苹果已经引入了另外一种基于View层的强大的动画框架，`UIViewPropertyAnimator`。

他提供了一个非常棒的方法来解决以前自定义timingFunction只能由CAAnimation来处理的问题。

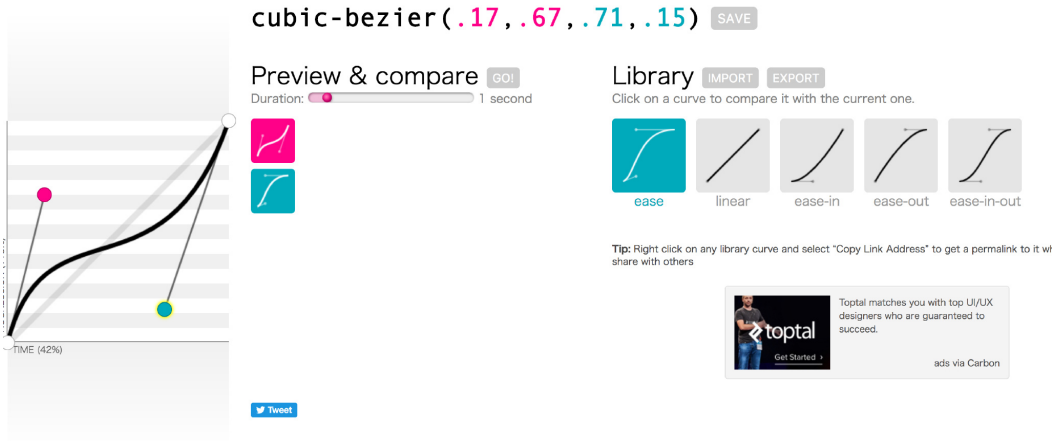
timingFunction

说到timingFunction，相信写过动画的人都非常清楚系统提供的几种。

- Liner （线性）
- EaseIn （先慢后快）
- EaseOut （先快后慢）
- EaseInEaseOut （慢进，加速，减速）

实际上这几种timingFunction只能说是勉强够用。当你想更细致调整动画速率的时候势必会使用自定义的贝塞尔曲线来控制动画速率。

比如在`http://cubic-bezier.com/`,我创建了一个自定义的曲线。



他的control point 分别是（0.17, 0.67, 0.71, 0.15） 那么，如果你想用这个贝塞尔曲线当做timingFunction,在iOS10之前你只能利用CABasicAnimatin来实现。

例如，第一个旋转动画自定义timingFunction是这样的。

```
basicAnimation.timingFunction = CAMediaTimingFunction.init(controlPoints: 0.17, 0.67, 0.71, 0.15)
```

想在View层自定义timingFunction？没门!

所幸，我们在iOS10的时候拥有了`UIViewPropertyAnimator`。

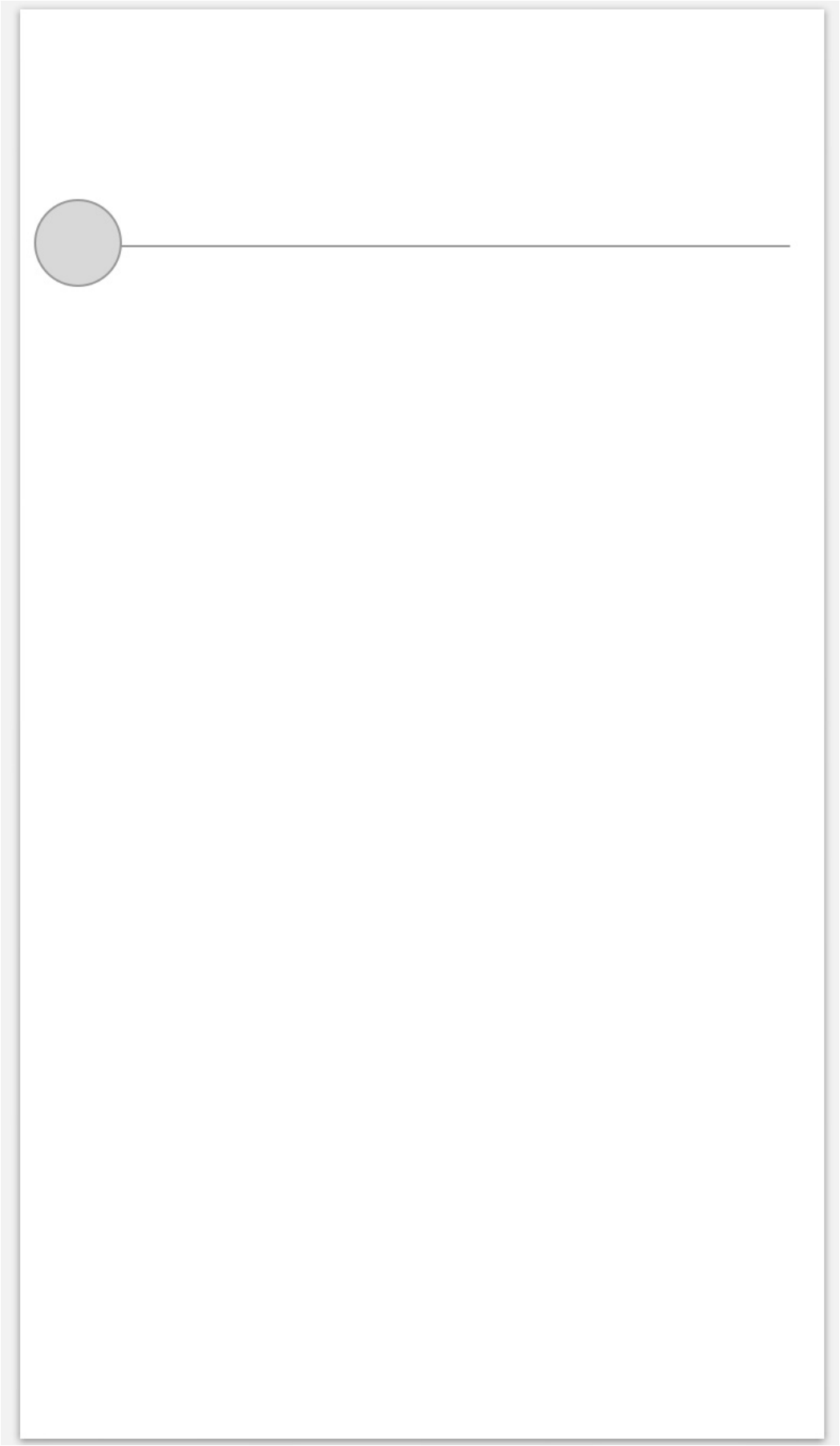
现在，我们如此简单的就创建了一个自定义动画速率的动画。

```
    let convenienceAnimator = UIViewPropertyAnimator.init(duration: 0.66,
controlPoint1: point1, controlPoint2: point2) {

            }
    convenienceAnimator.addCompletion({ (position) in
               if position == .end {

                }
     })
     convenienceAnimator.startAnimation()
```

iOS11中更强大的UIViewPropertyAnimator

session 230中，苹果着重介绍了我们梦寐以求的简单方便的交互式动画api。

举一个session 230中的例子来看一下新版本中如何实现交互式动画。



这里，我们需要用手势来控制动画的进度。这里，动画是让小球从左向右移动100的距离。

看看代码如何简单的将动画和手势关联起来。

```
var animator:UIViewPropertyAnimator!
var circle:UIImageView!

func handlePan(recognizer:UIPanGestureRecognizer) {
        switch recognizer.state {
        case .began:
            animator = UIViewPropertyAnimator.init(duration: 1, curve:
.easeOut, animations: {
                self.circle.frame = self.circle.frame.offsetBy(dx: 100, dy:
0)
            })
            animator.pauseAnimation()
        case .changed:
            let translation = recognizer.translation(in: self.circle)
            animator.fractionComplete = translation.x/100

        case .ended:
            animator.continueAnimation(withTimingParameters: nil,
durationFactor: 0)

        default:
            break
        }
    }
```

1. 手势开始的时候创建animator。然后暂停,在这里，动画暂停的本质同样是将Layer的speed设置为0。
2. 动画的完成率等同于手势移动的距离除以总距离。
3. 当手势结束的时候，我们调用了continueAnimation让动画继续执行到结束。其实这种需求比较少见，最常见的应该是当手势结束的时候让动画停留在这个阶段而不是继续进行动画。

在这里，我们改造一下这个动画，让它更符合我们的用户习惯。

首先，在手势事件的外部定义好这个animator。

```
 circle.backgroundColor = UIColor.red
        circle.layer.cornerRadius = 10
        circle.frame = CGRect.init(x: 10, y: 100, width: 20, height: 20)
        circle.isUserInteractionEnabled = true
        self.view.addSubview(circle)

        animator = UIViewPropertyAnimator.init(duration: 1, curve: .easeOut,
animations: {
            self.circle.frame = self.circle.frame.offsetBy(dx: 100, dy: 0)
        })
        animator.pauseAnimation()
```

然后，手势的事件代码如下。

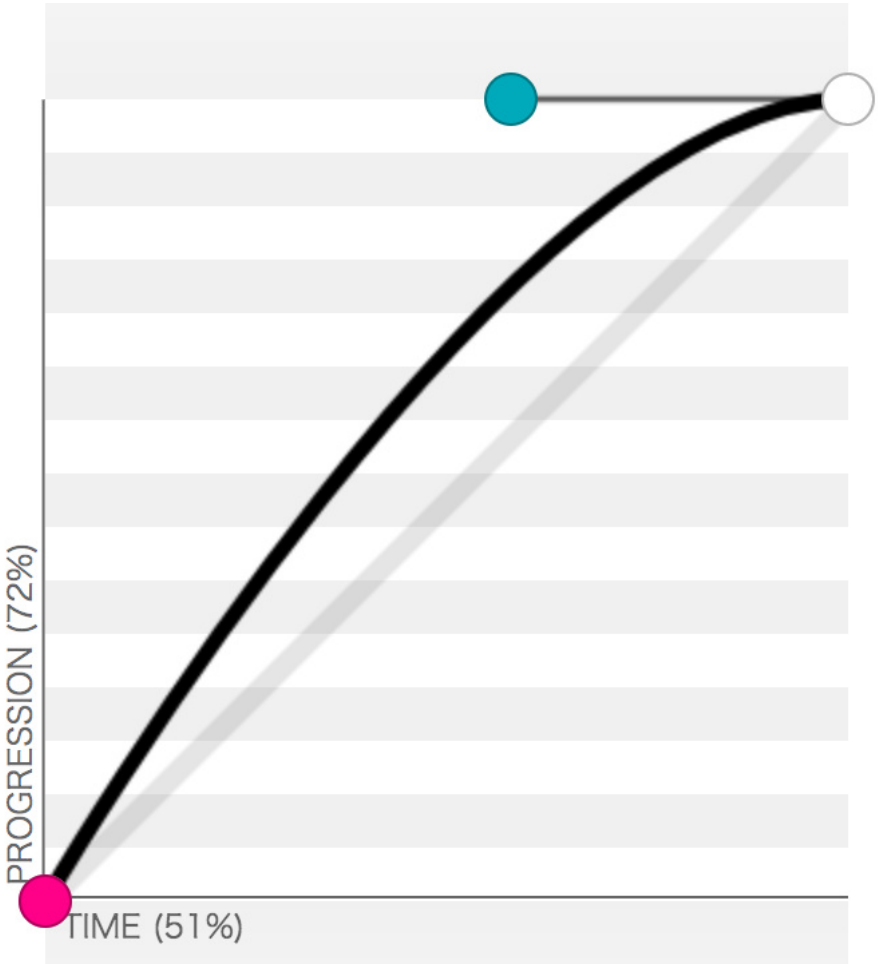
```
func handlePan(recognizer:UIPanGestureRecognizer) {
        switch recognizer.state {
        case .began:
            progress = animator.fractionComplete

        case .changed:
            let translation = recognizer.translation(in: self.circle)
            animator.fractionComplete = translation.x/100 + progress

        case .ended:
            break

        default:
            break
        }
    }
```

在这里，我们多了一个叫做progress的变量，这个变量的作用就是记录当前动画的进度，在每次手势变化的时候，让动画保持连贯性。不然，每一次动画都重新执行了。 建议同学们这里自己用代码试验一下效果。

出现了一些问题?

话说讲到这里，我不知道有没有同学会对一个非常重要的问题感到疑惑。 什么问题呢？ 就是**创建animator的时候的timingFunction是EaseOut，先快后慢，那么理论上应该是手势移动了一半，动画早就进行的超过了一半才对。**

因为EaseOut的动画曲线是这样的



注意看这张图的横纵坐标。 X坐标代表Time的进度，Y坐标代表动画的进度。 当X走到51%的时候，动画已经进行了72%。 在我们的场景中，这意味着，当手势移动了51个pixel的时候，circle这个view已经跑了72个pixel。

想想这会造成什么问题?

问题就是，用户在交互的时候完全摸不着头脑。

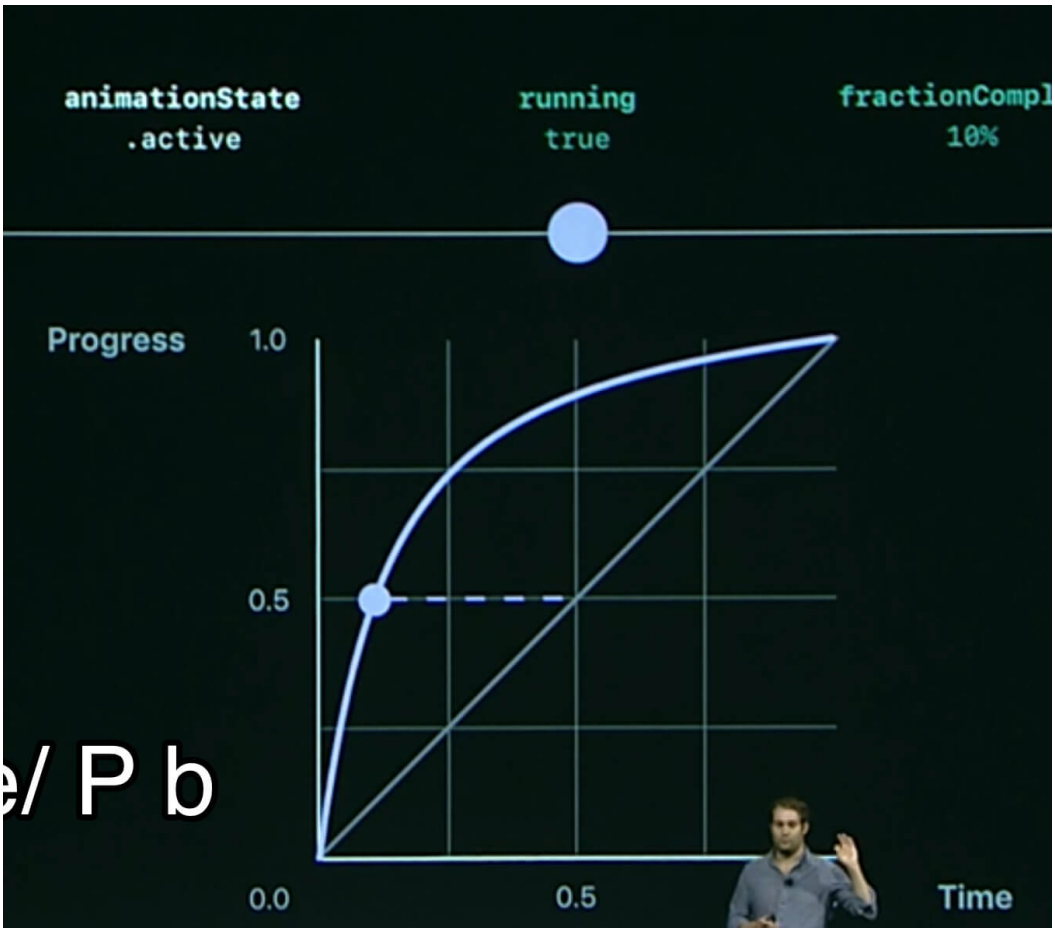
再举个形象的例子。

加入有一个UISlider控制一个Animator的进度，这个Animator是作用于View的透明度Alpha从1到0。

然后Animator的timingFunction是EaseOut，那么用户拖动UISlider的结果很可能是Slider还没滑动到底，这个View的alpha已经变成了0.

为了避免这种情况，当你的Animator是Interactive状态的时候，苹果会自动把你的timingFunction转变为`Linear`.

如图



那么如果你真的希望可交互式动画的timingFunction不是自动转变为Liner，能不能做到呢?

答案是可以的。 苹果在iOS11中为UIViewPropertyAnimator提供了一个Bool值`scrubsLinearly`，只要设置为No，那么动画就会按照你设置的timingFunction执行了。

第二个问题，动画执行完了怎么办?

其实在手势执行完毕的时候，调用`animator.continueAnimation(withTimingParameters: nil, durationFactor: 0)`会将动画执行完成，但是有一个问题是，动画一旦执行完成，动画的状态就会从Interactive转变为Active，也就是说，不可以再进行交互了。这时候，你需要把animator的pauseOnCompletion设置为false。那么动画就会一直保持Interactive状态了。

SpringAnimation

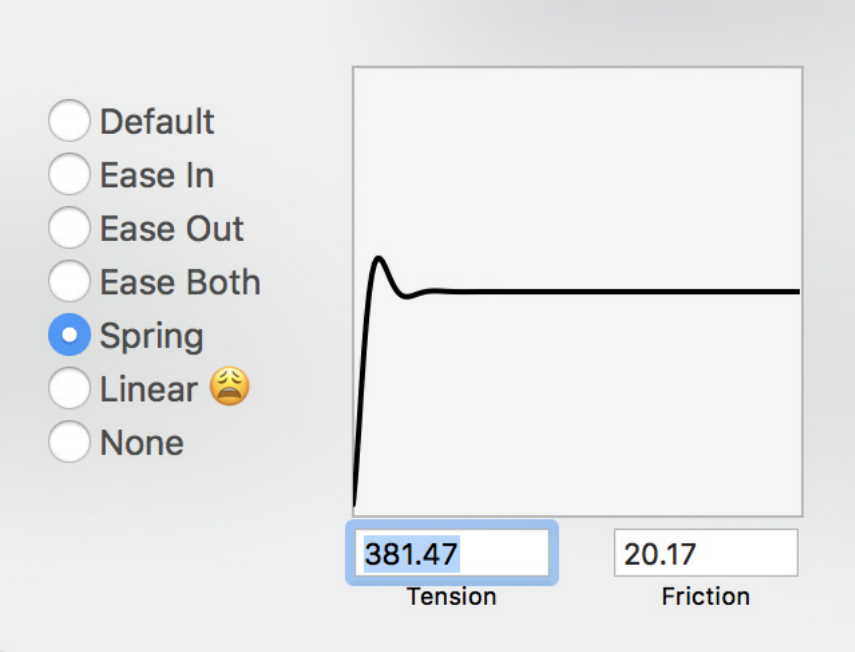
说实话，这期session中，让我比较失望的就是苹果对于SpringAnimation的支持只是简单地增加了一个under-damping的概念。并没有加入springAnimation中很重要的两个属性。

- Fricition
- Tension

为什么这两个属性非常重要。这里，我需要给大家介绍一个国外非常流行的app。**Principle**

他是国外做交互式prd的非常好用的一个app，我最近在做的一个app在做交互原型的时候大量的使用了这个app。

我们来看看这个app中对于spring动画的一些设置。



用damping这个参数调spring最大的问题就是.....无法当伸手党，直接拿来参数用。 所以，目前来说，最好用的SpringAnimation还是facebook得pop。

比如..... 一个pop伸手党的日常是这样的。

```
let alphaSpring = POPSpringAnimation.init(propertyNamed: kPOPViewAlpha)
        alphaSpring?.fromValue = 0.67
        alphaSpring?.toValue = 1
        alphaSpring?.dynamicsFriction = 20.17
        alphaSpring?.dynamicsTension = 381.47
        alphaSpring?.delegate = self
        alphaSpring?.name = "alpha"
        self.pop_add(alphaSpring, forKey: "alpha")
```

只能说，用pop好省心。

补充

cornerRadius终于可动画了。

提出两个问题

1. iOS11之前真的没有支持手势交互的api么?
2. 如果存在这样的api，那么这个api的原理是什么呢？是怎样实现无论是UIViewAnimation还是

CABasicAnimation都能无缝和手势关联的呢?

这是两个很有意思的问题，大家有空可以思考一下。

参考

> Session 230

Xcode 9 新增调试功能介绍

Xcode 9 支持了无线调试，大大的方便了一些 AR 等被有线调试限制的开发者。可见苹果对 AR 开发还是蛮重视的。

与此同时，增加了 View Controller 界面调试提高了原生开发的效率，还把 SpriteKit 和 SceneKit 提升到界面调试的一等公民。可见苹果对游戏开发也越来越重视。

下面我会总结一下在 **WWDC 2017** 上提到的关于 Xcode 9 新增的一些调试工具用法和介绍。

无线真机调试

苹果在 Xcode 9 引入了**无线调试**的功能。

主要便利了如下的几种开发者：

- AR，VR和相机应用开发者
- 运动感应和健身APP开发者
- 配件制造商

关于配件制造商，可能有些人一时想不到便利在哪里，其实主要是对于通过 Lightning 口的方式接入的配件来说，无线调试的引入，不再需要在配件上提供一个 Lightning 口用来连接手机和电脑。

支持的设备

- iPhone, iPad, or iPod Touch running iOS 11
- Apple TV running tvOS 11
- macOS 10.12.4+

支持的工具

- iOS/tvOS:
    - Xcode
    - Instruments
    - Accessibility
    - Inspector
    - Console(需要 macOS 10.13)
    - Configurator
- 只支持 tvOS : - Safari 上的 TVMLKit Web检查器
    - QuickTime 屏幕录制(需要 macOS 10.13)

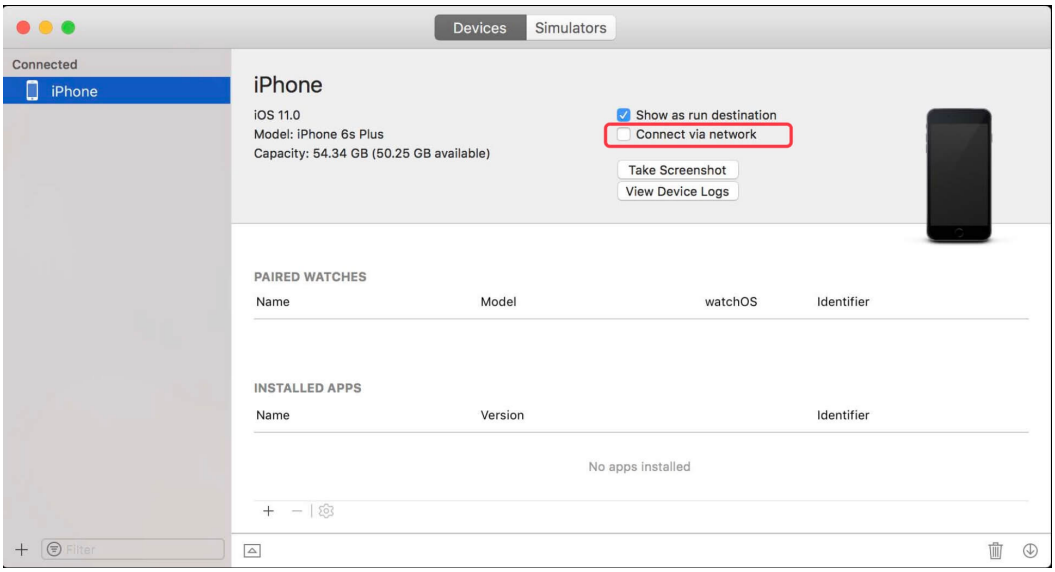
设备连接方式

除了通过 WIFI,通过 **网线口**、**无线连接**、**USB/Type-C** 等现在都可以进行调试了。

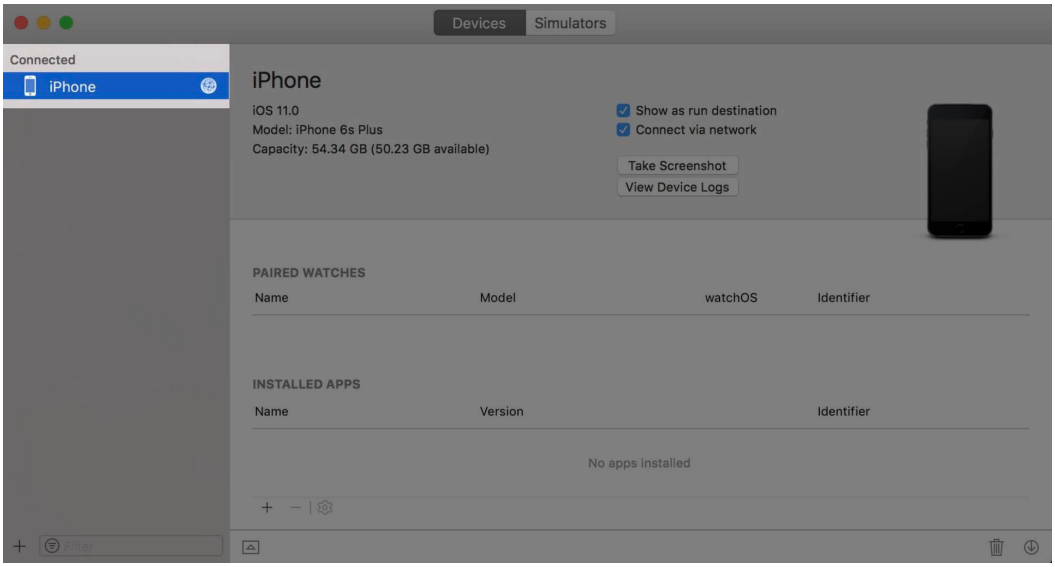
配置无线真机调试

首选，确保电脑和被调试设备处于同一个局域网下。

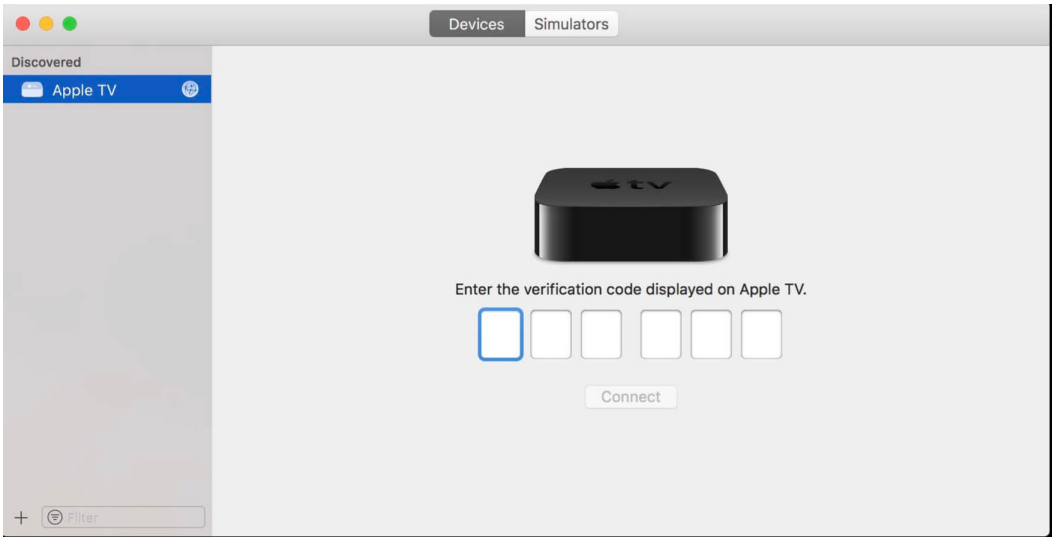
对于 iOS设备 来说，首先通过数据线连接到电脑，打开 Xcode -> Window -> Devices and Simulators



选中 Connect via network,出现如下所示界面即可。后续直接选择处于同一局域网下的手机就可以调试。

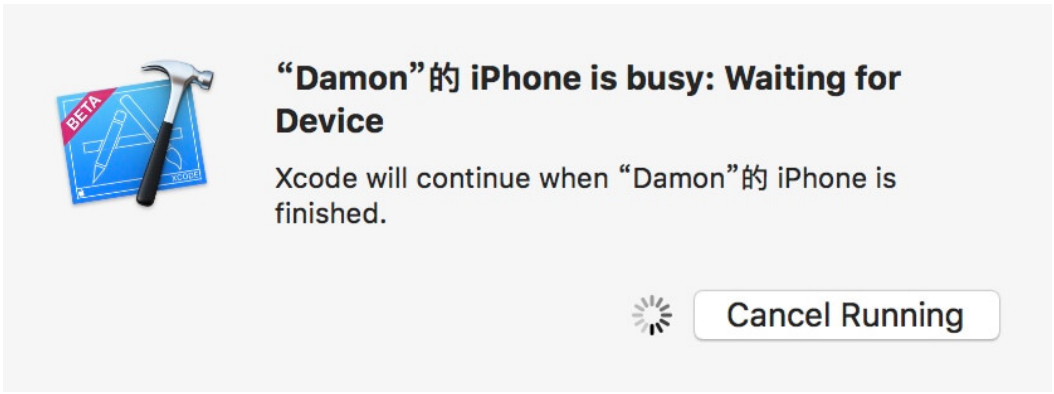


而对于 tvOS 设备来说，可以直接通过 WIFI 来进行 Connect via network 的设置。

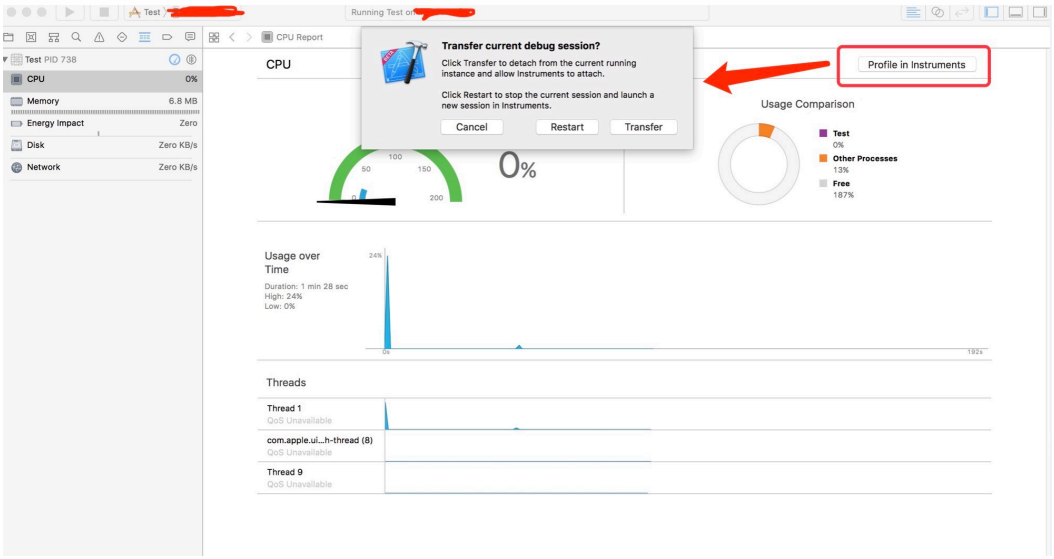


当我们设置好了 Connect via network，通常情况下已经可以无线真机调试了。但是对于一些**复杂的网络**，需要把手机的 IP 地址设置到 Connect via IP Address 中：

关于如何区分复杂网络:在设置好 Connect via network 之后，选择需要调试的真机，随便运行一个可以运行的程序，如果一直处于下图的 loading 界面，那么需要把手机的 IP 地址设置到 Connect via IP Address 中，才能进行无线真机调试。



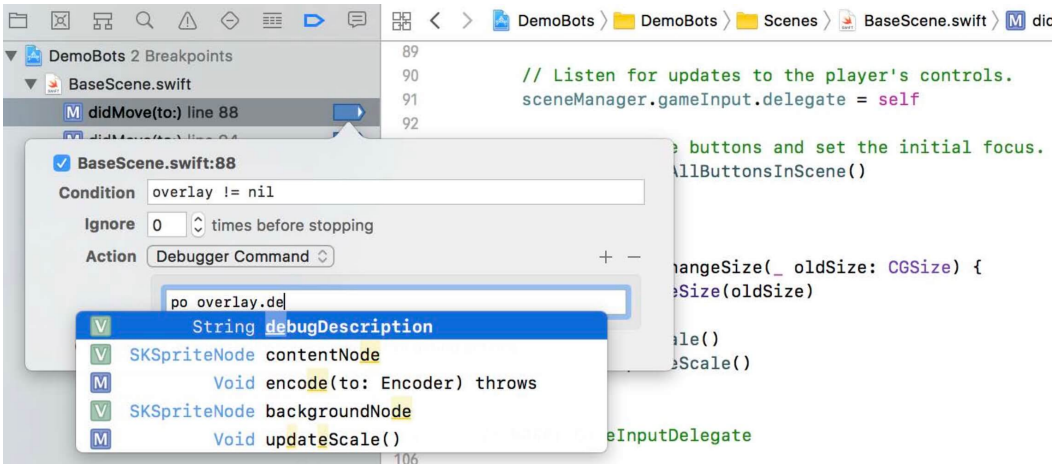
调试界面直接打开 Instruments



在演示 Xcode 9 的无线调试的时候，苹果工程师提到，在 Xcode 的线程调试界面，遇到性能问题时，现在可以直接打开 Instruments 并保证程序继续运行，继续调试。

断点调试优化

支持代码补全



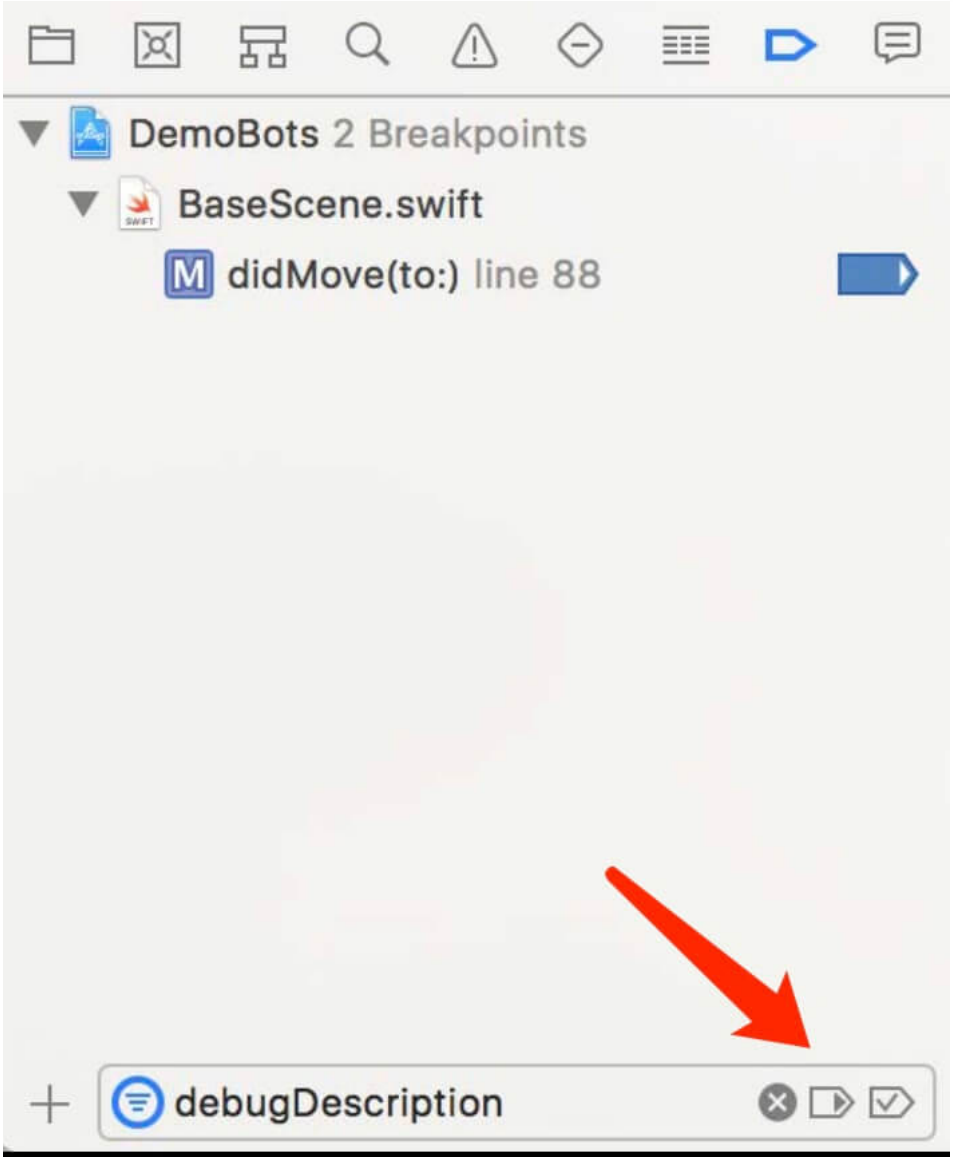
对于 condition 和 expression action 的 feild 来说都支持代码补全。

新增参数指示器

对于设置过参数的断点，在箭头处会有一个白色的三角形标记,用来区别设置过参数和没设置过参数的断点。

搜索加强

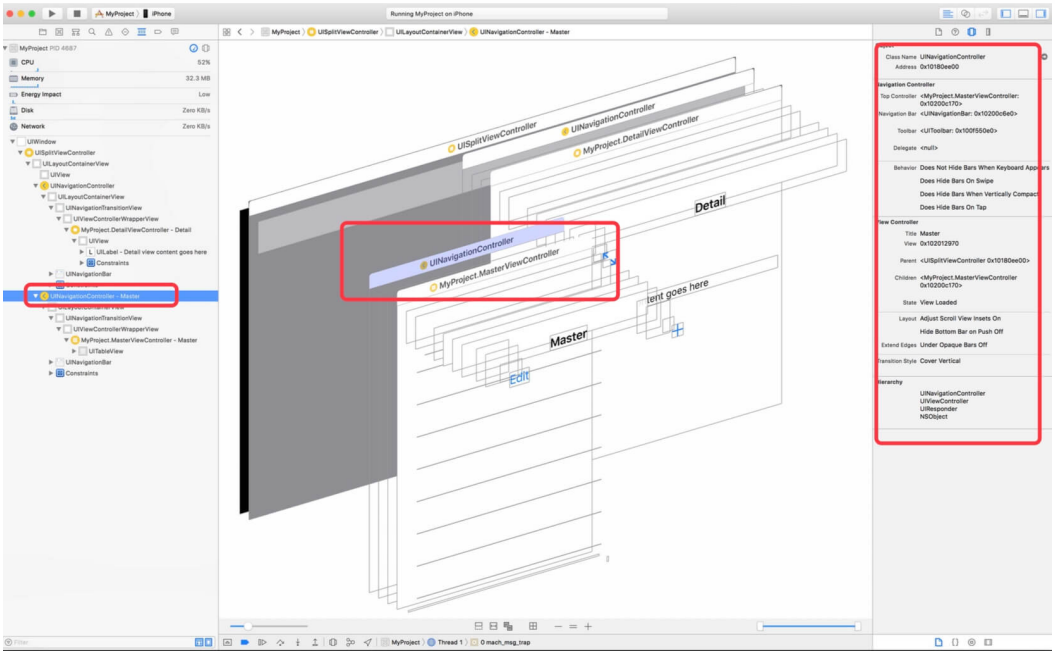
同时在过滤筛选界面，也新增了参数指示器的筛选选项。 而且目前也支持断点参数信息里面的文本筛选。



用户界面调试

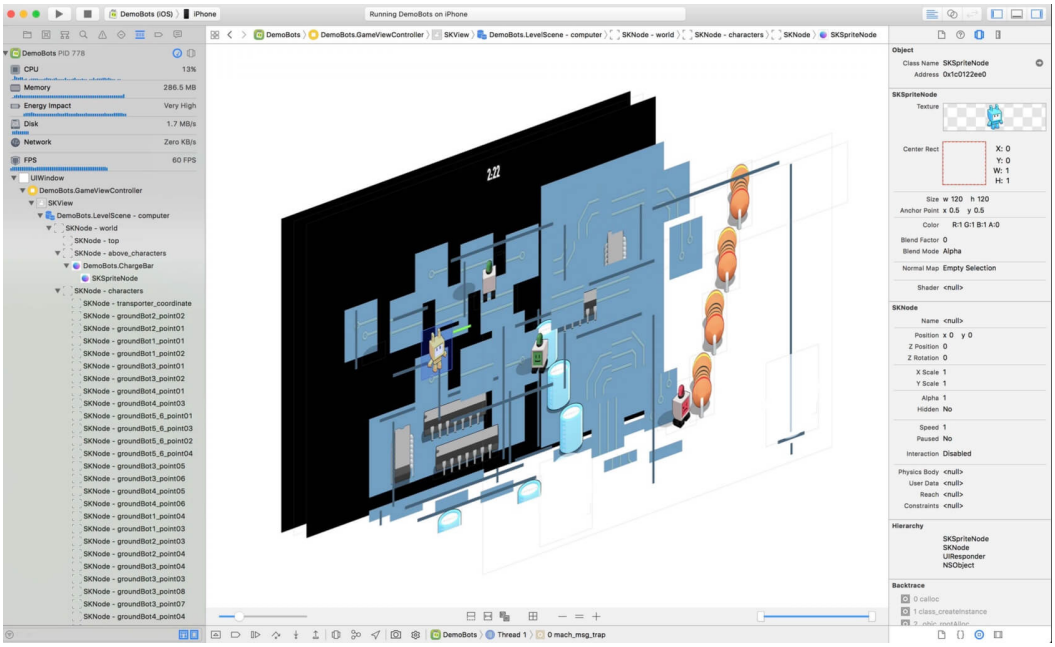
View Controller 调试

Xcode 9 中在调试的时候，将 ViewController 作为 View hierarchy 的一部分，也就是说 View Controller 现在支持 3D 调试。



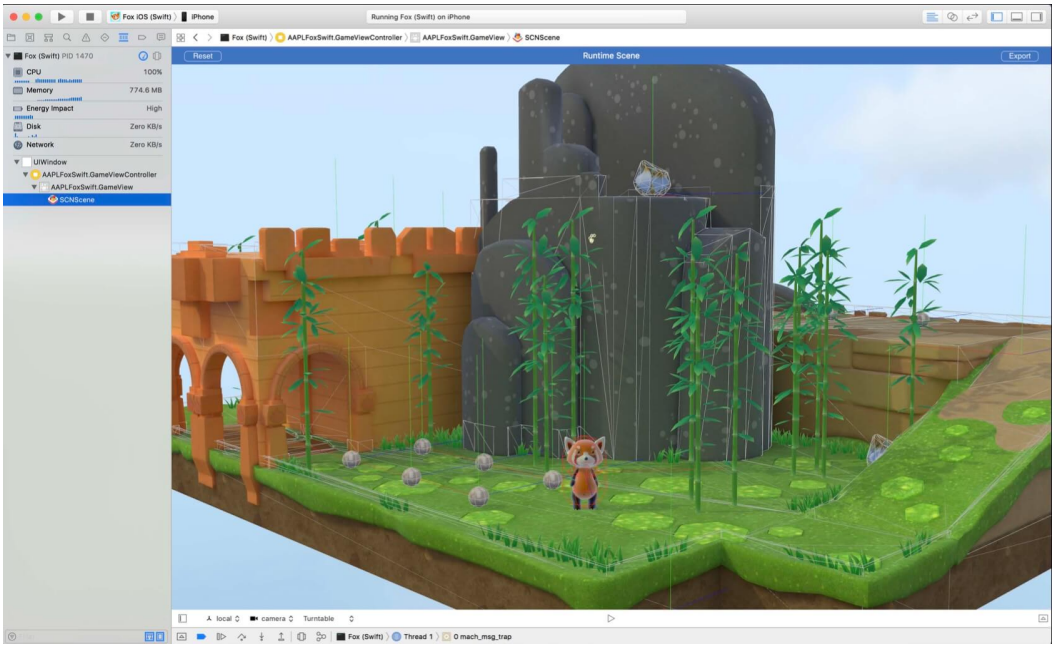
SpriteKit 调试

Xcode 9 支持了 SpriteKit 的界面调试，在此之前 Xcode 只支持 UIKit 和 AppKit 的截面调试。SpriteKit 成为了界面调试的一等公民。



SceneKit 调试

Xcode 9 支持了 SceneKit 的界面调试，在此之前 Xcode 只支持 UIKit 和 AppKit 的截面调试。SceneKit 也成为了界面调试的一等公民。



最低配置

- iOS 11
- tvOS 11
- macOS High Sierra

参考

> Session 404 视频
>
> Presentation Slides (PDF)

Drag and Drop 深入探究

大家对于 **Drag and Drop** 的如何使用已经在 「**Introducing Drag and Drop**」 中有所体验，本文章会在此深入讨论 **Drag and Drop** 的开发， 诸如 生命周期， 动画，预览，技巧。首先要明白 **Drag and Drop** 是一种在 Source App 和 Destination App 之间（这俩 App 可以是同一个）**传输数据的协议**， **Drag and Drop** 看起来很高大上，其实苹果已经将 API 极大的简化，按照拆分的思路， 你只需要关注用户输入事件，即可完成整个过程。 因为我们在实际操作的时候， 可以独立的来考虑 **Drag 和 Drop**，所以本文将会分两部分： **Drag Source** 和 **Drop Destination**。

Drag Source

设置 View 可被 Drag

设置并实现一个 `UIDragInteraction` 实例的 `UIDragInteractionDelegate` 代理方法 ，并将这个 `interaction` 添加给 `View` ，就实现了 `View` 可以被 `Drag` 功能（这种 `View` 上长按就会可以拖走）； `UITextView` 、 `UITableView`、 `UICollectionView` 这三个类有它们各自专属的方法来创建 `Drag Item` `interaction` 的代理一般可以是 `ViewController` 或 单独的类，但都需要遵循 `UIDragInteraction` 协议；如果代理是 `Controller` 这个操作一般在 `ViewDidLoad()` 或者在 View 的 `Init()` 方法内完成， `interaction` 初始化时必须设置好代理。类似这样：

```
func customEnableDragging(on view: UIView, dragInteractionDelegate:
UIDragInteractionDelegate) {
    let dragInteraction = UIDragInteraction(delegate:
dragInteractionDelegate)
    view.addInteraction(dragInteraction)
}
```

`Session` 包括两种 `UIDragSession` 和 `UIDropSession`，但 `Session` 都可以被理解为一根手指所干的事；比如一根手指拖拽着几张图片，那么在 `drag` 的区域，连带这些操作和拖拽的图片就是 `drag session` ，在 `drop` 区域这跟手指和他拖动的图片以及其他操作就是 `drop session`。但 `DragSession` 只包含一个 `localContext: Any?` 的自定义上下文标识符。 `Drop Session` 还包括 `progressIndicatorStyle: UIDropSessionProgressIndicatorStyle` 和方法 `func loadObjects(ofClass aClass: NSItemProviderReading.Type, completion: @escaping ([NSItemProviderReading]) -> Void) -> Progress`

为 Drag 过程提供数据

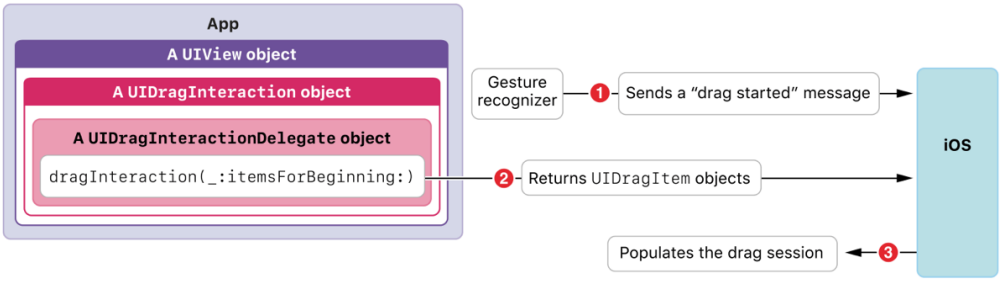
`Drag` 操作时需要为 `Drag` 的 `View` 提供 `dragItem` 所需数据，这个数据就是在 `Drag` 过程中**实际传输**的数据，比如字符串、图片、文件等；通过`dragInteraction(_:itemsForBeginning:)`方法来实现，这是 `UIDragInteractionDelegate` 的代理方法，并且这个方法只接受 `NSItemProvider`，也就是说你必须要把你的数据进行封装；比如你在拖拽一段文字，跨 App 传输，需要将需要传输的 App ，需要将对象封装成 `NSItemProvider` ， （如果传输的类对象是 `NSString`,`NSAttributedString`,`NSURL`,`UIColor`,`UIImage`，你不需关心 `NSItemProviderReading`与`NSItemProviderWriting`协议，也就是封装传输时候声明出数据的类型，以便数据接收方能够正确的取出和加载数据；如果是拖拽的数据是自定义的类，需要在数据的类中自行实现这个协议，来声明好 Uniform Type Identifiers , 有关具体可以参照 UTI Reference）
下面是传输一个字符串时候的实现：

```
func dragInteraction(_ interaction: UIDragInteraction, itemsForBeginning
session: UIDragSession) -> [UIDragItem] {
    // 因为 NSItemProvider 只支持类，所以你需要把 String Cast to NSString
    let stringItemProvider = NSItemProvider(object: "Hello World" as
NSString)
    return [
    UIDragItem(itemProvider: stringItemProvider)
    ]
}
```

如果一个拖拽的 `View` 所包含的内容是多张图片的组合或 文字和图片的组合，可以这么实现

```
func dragInteraction(_ interaction: UIDragInteraction, itemsForBeginning
session: UIDragSession) -> [UIDragItem] {
    return imageViews.map { (imageView) -> UIDragItem in
    let itemProvider = NSItemProvider(object: imageView.image!)
        let item = UIDragItem(itemProvider: itemProvider)
        item.localObject = imageView
        return item
    }
}
```

Drag 时整个的 iOS 的处理逻辑和生命周期



`Drag` 时，长按事件被识别为 `Drag` 手势，iOS 系统在受到 `drag` 手势的消息后，iOS 会初始化一个 `drag session` ，回调 `itemForBegining` 的代理方法，接下来 iOS 会把整个 Drag 的事务处理通过 `DragInteractionDelegate` 来处理。也就是说只有 `itemForBegining` 是必须实现的，其他方法都有默认实现

Drag 过程常用 UIDragInteractionDelegate 方法

接下来介绍一些 `DragInteractionDelegate` 的一些代理方法。 首先介绍一个概念 `Lift` ， `Lift` 指的是用户长按开始拖拽后，被拽的物体一般会跟随手指而移动，这些被拖拽的 `items` 跟随手指移动的状态就是 `Lift` ，就像被手指吸附抬起悬空一样；

开始

在 `Drag` 开始时，会调用 `itemForBegining` 代理方法，紧接着回调用 `previewForLifting` 方法来获取每个 `item` 的 `UITargetedDragPreview` ，也就是 `item` 被 `Lift` 后的预览样式，如果你希望使用默认的预览样式，不要实现这个代理方法；如果不需要任何抬起动画预览样式，返回 `nil` 。

添加

在 `Lift` 状态下，用第二根手指来点选其他可以 `Drag` 的 `View`，就可以将新的 `dragItem` 加入当前的 `Drag Session`，注意这是同一个 `drag session`，一般情况下`Drag and Drop` 过程中只需要一个`drag session`，但是也支持多个 `session`，这个需要自己去创建, 这里不赘述。***one drag interaction touch + one drag interaction delegate = one active drag session「一根手指的长按拖拽触摸 + 一个 drag interaction 代理 = 一个活跃的 drag session」***

```
func dragInteraction(_ interaction: UIDragInteraction, itemsForAddingTo
session: UIDragSession, withTouchAt point: CGPoint) -> [UIDragItem] {}
```

取消

在 Drag 过程中，用户的手指至少会有一根在屏幕上，如果用户所有的手指都离开了屏幕，iOS 会告知代理方法开始处理**「取消」**的相关代理方法 首先会调用方法来实现 `Cancel` 时候的动画，这个方法每个 `drag item` 都会单独调用一次，如果不实现，，如果使用系统默认动画，请不要实现这个方法，系统会有默认实现；

```
optional func dragInteraction(_ interaction: UIDragInteraction, item:
UIDragItem, willAnimateCancelWith animator: UIDragAnimating)
```

最后，所有的 `drag item` 都进行过 `cancel animation` 后，系统会调用以下方法结束对应的 `drag session`

```
optional func dragInteraction(_ interaction: UIDragInteraction, session:
UIDragSession, didEndWith operation: UIDropOperation)
```

所有代理方法的简介

Drag 的执行

```
func dragInteraction(_ interaction: UIDragInteraction, itemsForBeginning
session: UIDragSession) -> [UIDragItem]
//Drag 的执行最先回调的代理方法，必须实现，返回了 drag item

optional func dragInteraction(_ interaction: UIDragInteraction,
itemsForAddingTo session: UIDragSession, withTouchAt point: CGPoint) ->
[UIDragItem]
//用户在同一个 drag session 中添加 drag item 的代理方法，比如用户一根手指拖动着一张图片
时，点击其他图片，就会触发添加操作

optional func dragInteraction(_ interaction: UIDragInteraction,
sessionForAddingItems sessions: [UIDragSession], withTouchAt point: CGPoint)
-> UIDragSession?
//如果需要实现用户两根手指各自拖动各自的 drag item，实现这个方法，参数中包含每个 session
对应的触摸坐标
```

Drag 时的自定义动画

```
optional func dragInteraction(_ interaction: UIDragInteraction,
willAnimateLiftWith animator: UIDragAnimating, session: UIDragSession)
//长按之后使 drag item「悬空」时的动画效果，如果使用默认实现，请不要实现这个方法
optional func dragInteraction(_ interaction: UIDragInteraction, item:
UIDragItem, willAnimateCancelWith animator: UIDragAnimating)
//用户如果在拖动过程中手指离开屏幕，表示取消，此方法自定义「取消」的动画，注意每个 item 均
可自定义
```

监听 Drag 的进程

```
optional func dragInteraction(_ interaction: UIDragInteraction,
sessionWillBegin session: UIDragSession)
//监听drag session 开始时的回调方法，如果需要在 Drag 时候准备一些操作比如打点，可以在这
里实现

optional func dragInteraction(_ interaction: UIDragInteraction, session:
UIDragSession, willAdd items: [UIDragItem], for addingInteraction:
UIDragInteraction)
//session 中「将要添加」 drag item 时的监听回调方法

optional func dragInteraction(_ interaction: UIDragInteraction,
sessionDidMove session: UIDragSession)
//session 在移动时的监听回调方法，用户的一根手指往往就是一个 session ，在用户移动手指
时，会频繁调用这个方法

optional func dragInteraction(_ interaction: UIDragInteraction, session:
UIDragSession, willEndWith operation: UIDropOperation)
//session 「将要结束」时的监听回调方法

optional func dragInteraction(_ interaction: UIDragInteraction, session:
UIDragSession, didEndWith operation: UIDropOperation)
//session 「已经结束」时的监听回调方法

optional func dragInteraction(_ interaction: UIDragInteraction,
sessionDidTransferItems session: UIDragSession)
//如果用户执行了 Drop 操作，并且数据或对象异步「传输完毕」后会调用这个方法，需要注意是异
步，所以 UI 操作注意线程
```

Drag 时的自定义预览

```
optional func dragInteraction(_ interaction: UIDragInteraction,
previewForLifting item: UIDragItem, session: UIDragSession) ->
UITargetedDragPreview?
//被「悬空」的 drag item 的预览效果，如需使用默认的预览效果，返回 nil

optional func dragInteraction(_ interaction: UIDragInteraction,
previewForCancelling item: UIDragItem, withDefault defaultPreview:
UITargetedDragPreview) -> UITargetedDragPreview?
//用户取消 drag 时，取消时的预览效果，如需使用默认动画效果返回 nil

optional func dragInteraction(_ interaction: UIDragInteraction,
prefersFullSizePreviewsFor session: UIDragSession) -> Bool
//如果需要全尺寸、不压缩大小的预览时，返回 true 。一般在 drag 时使用100\*100 左右的预
览，上面的预览方法实现时已定义大小，默认 false
```

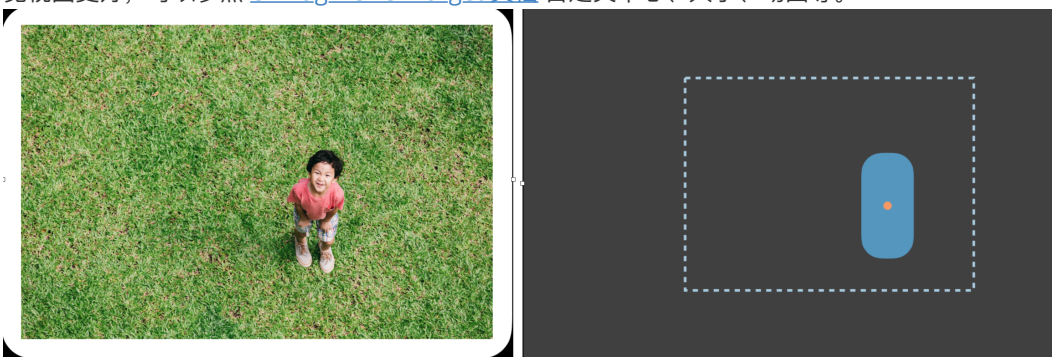
Drag 操作的限制

```
optional func dragInteraction(_ interaction: UIDragInteraction,
sessionIsRestrictedToDraggingApplication session: UIDragSession) -> Bool
//是限制 drag item 是否必须限制在 source App 内，不许拖到其他 App 内，如果需要这个限制
返回 true ，默认 false 且不需实现

optional func dragInteraction(_ interaction: UIDragInteraction,
sessionAllowsMoveOperation session: UIDragSession) -> Bool
//如果 drag 和 drop 的 App 为同一个时，可以设置 drag operation ，返回 true 时为
.move ，返回 false 时为 .copy ，默认 false 且不需实现
```

实践中的技巧和优化 Tips

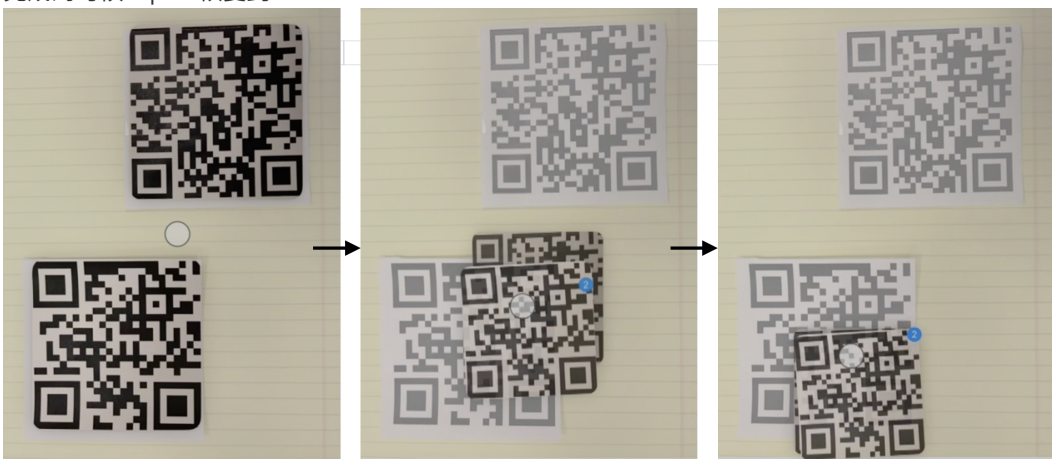
Lift 预览视图大小的优化

在一些特定的场合下，自定义 `Lift` 预览样式更好看，比如这张图片， 如果使用默认的 `Lift` 预览样式，就是右图中虚线的框，如果在拖动时候只显示这个小孩，体验将会更好，所以自定义的预览视图更好，可以参照 UIDragPreviewTarget 文档 自定义中心、大小、动画等。



Lift 时预览视图与动画的优化

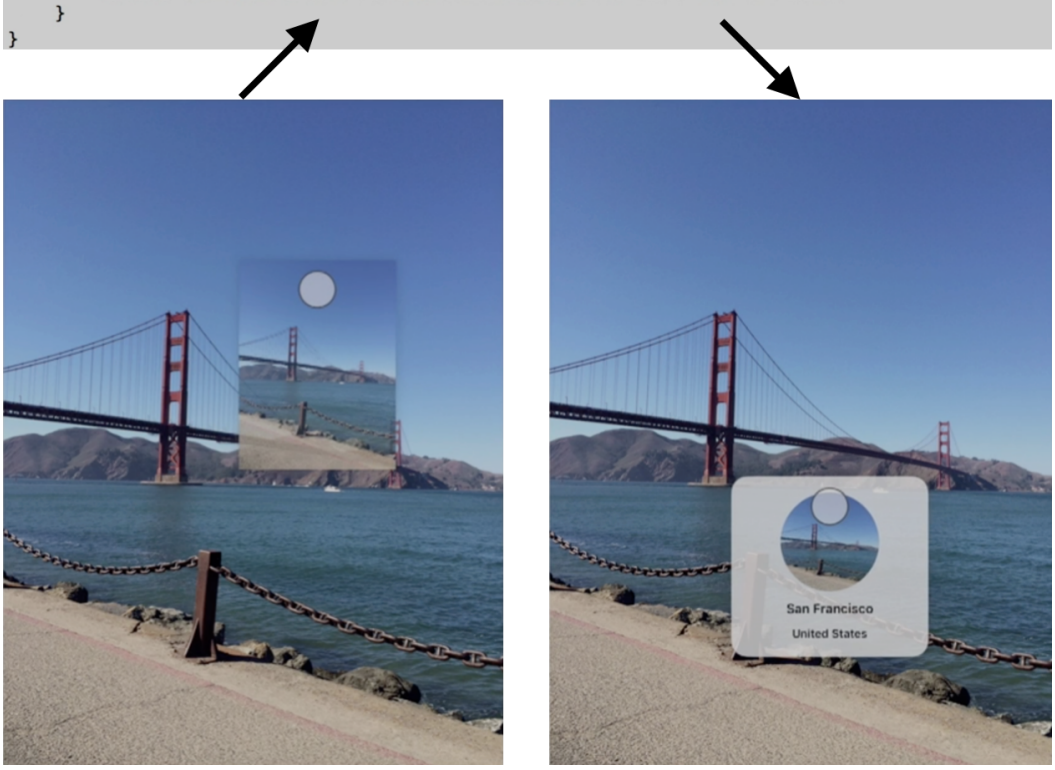
再比如这张两个二维码的图片，拖拽的时候可以获取到两个二维码，将两个二维码作为实际拖拽的内容，类似，相关 demo 的代码可以在 这里下载， 在这个 demo 中苹果的工程师介绍了很多 polish（润色）细节的实现，比如 drag 时候动画设置的技巧，Lift 的时候背景的 alpha 是0.5 ， 在完成的时候 alpha 恢复到 1；



实现更符合场景的预览视图

比如一个图片拖拽到地图中，地图 App 会显示照片拍摄地的 Pin，那么这个图片 `Lift` 后的预览就是如果是一张图片会给人造成困惑，如果是一个地点信息那就回更明晰：

```
func dragInteraction(_ interaction: UIDragInteraction, sessionWillBegin session:
    UIDragSession) {
    session.items[0].previewProvider = { () -> UIDragPreview in
        guard let image = self.image else { return UIDragPreview(view: self) }
        guard let mapItem = self.mapItem else { return UIDragPreview(view: self) }
        let previewView = LocationPlatterView(image, item: mapItem)
        let inflatedBounds = previewView.bounds.insetBy(dx: -20, dy: -20)
        let parameters = UIDragPreviewParameters()
        parameters.visiblePath = UIBezierPath(roundedRect: inflatedBounds, cornerRadius:
            20)
        return UIDragPreview(view: previewView, parameters: parameters)
    }
}
```



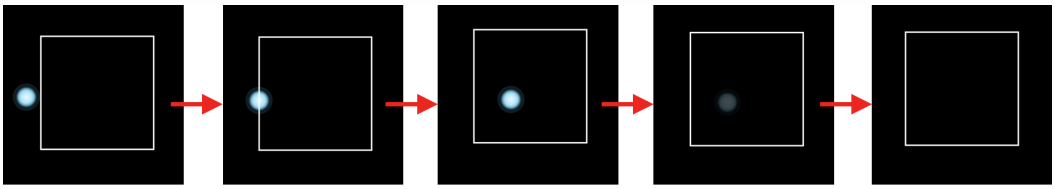
Drop Destination

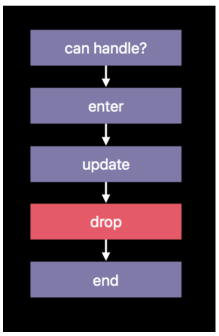
执行 Drop 功能所需的必要方法和操作

与 `Drag` 类似，如果想要一个 `View` 具有接受 `drag item` 的能力，需要先添加 `drop interaction`，像这样

```
func customEnableDropping(on view: UIView, dropInteractionDelegate:
UIDropInteractionDelegate) {
    let dropInteraction = UIDropInteraction(delegate:
dropInteractionDelegate)
    view.addInteraction(dropInteraction)
}
```

接下来结合用户的触摸与 `UIDropInteractionDelegate` 与 `Drop Session` 的生命周期，下图表示了用户操作与 代理方法之间的关系；图中的白色方框表示可以接受 `drag item` 的 `View` 的 `frame` 示意；蓝白色原点 `drag item`，每次移动也就是手指在屏幕上移动；原点淡出消失表示的是手指松开 执行 `drop`；没有原点意味着用户手指已经离开屏幕；因为 `NSItemProvider` 的数据读取是异步的，所以不影响 `session` 的结束






```
optional func dropInteraction(_ interaction: UIDropInteraction, canHandle session: UIDropSessio
n) -> Bool

optional func dropInteraction(_ interaction: UIDropInteraction, sessionDidEnter session: UIDrop
Session)

optional func dropInteraction(_ interaction: UIDropInteraction, sessionDidUpdate session: UIDro
pSession) -> UIDropProposal

optional func dropInteraction(_ interaction: UIDropInteraction, performDrop session: UIDropSess
ion)

optional func dropInteraction(_ interaction: UIDropInteraction, sessionDidEnd session: UIDropSe
ssion)
```

这个顺序代表的是 **drop destination App** 可以接受 `drag item`，并且用户执行了 `drop` 操作。用户的手指在 `drop` 区域移动时候， `update` 的操作会反复调用，你需要在这个方法里返回 `UIDropPrpposal`，这个对象会告诉 iOS 你的 App 是移动还是复制 `drag item` 的数据， 并且在移动期间 `sessionDidUpdate` 方法会被反复调用，来更新移动区域和 `drop proposal`，这个方法不允许阻塞，否则 App 可能崩溃。 取出时候的操作，比如取出一个 `UIImage` 对象

```
func dropInteraction(_ interaction: UIDropInteraction, performDrop session: UIDropSession) {
    session.progressIndicatorStyle = .none
    for item in session.items {
        // 保证传过来的数据或文件是可接受的类型
        guard item.itemProvider.canLoadObject(ofClass: UIImage.self) else { continue }
        let photoView = nextView()
        // 除了接受对象你也可以接受文件，或者自定义的数据，用于接收数据和对象的方法主要有以下三种，比如传输过来的是一个
            UIImage , NSItemProvider 可以帮你转换成 png 或者 jpg 文件数据，你可以选择接受文件而不是一个对象
        //loadObject(), loadDataRepresentation(), loadFileRepresentation()
        let progress = item.itemProvider.loadObject(ofClass: UIImage.self, completionHandler: { (image,
            _) in
            // 数据传输完成后会进入这里，但是当前的线程是 background，要操作 UI 时务必小心调度到主线程
            DispatchQueue.main.async {
                photoView.stopShowingProgress()
                photoView.image = image as? UIImage
            }
        })
        itemStates[ObjectIdentifier(item)] = (view: photoView, progress: progress)
    }
}
```

代理方法

`perform` 执行之后，类似于 `Drag` 的预览操作，会立即执行 `preview` 的代理方法， 注意 `performDrop` 的代理方法同一个 `Drop Session` 只执行一次，如果要使用默认的 `preview`，请返回 `nil`，使用方法如下

```
func dropInteraction(_ interaction: UIDropInteraction, previewForDropping
item: UIDragItem, withDefault defaultPreview: UITargetedDragPreview) ->
UITargetedDragPreview? {
     let target = UIDragPreviewTarget(container: container, center:
state.view.center)
     return UITargetedDragPreview(view: /*ur custom preview*/, parameters:
UIDragPreviewParameters(), target: target)
}
```

之后会执行完成 `Drop` 的动画代理方法，使用方法如下， `UIDragAnimating` 是一个 `protocol`，只有两个方法 `func addAnimations(() -> Void)` 和 `func addCompletion((UIViewAnimatingPosition) -> Void)`，意如其字面，一个添加动画，一个是添加动画完成后的回调

```
func dropInteraction(_ interaction: UIDropInteraction, item: UIDragItem,
willAnimateDropWith animator: UIDragAnimating) {
     animator.addCompletion { _ in
           view.alpha = 1
     }
}
```

但是如果 `Drop Session` 包含多个 `Drop item`，每个 `drop item` 都会调用自己的 `previewForDropping` 的预览代理方法，`willAnimateDropWith` 的代理方法也是 同样的道理。

所有 `UIDropInteractionDelegate` 的代理方法简介

Drop 的拦截与处理

```
optional func dropInteraction(_ interaction: UIDropInteraction, performDrop
session: UIDropSession)
//用户松开手指，触发 Drop 操作时的回调。必须实现

optional func dropInteraction(_ interaction: UIDropInteraction, canHandle
session: UIDropSession) -> Bool
//返回 true 或 false 来表明自己的 View 是否可以接受 drag item，默 false

optional func dropInteraction(_ interaction: UIDropInteraction, concludeDrop
session: UIDropSession)
//drop 操作结束时的回调，在 performDrop  和 动画结束之后
```

Drop 操作时的自定义动画

```
optional func dropInteraction(_ interaction: UIDropInteraction, item:
UIDragItem, willAnimateDropWith animator: UIDragAnimating)
//drop 操作时的自定义动画，在实际 Drop 时候触发，先 perform ，之后就是这个方法
```

监听 Drop 的过程与移动

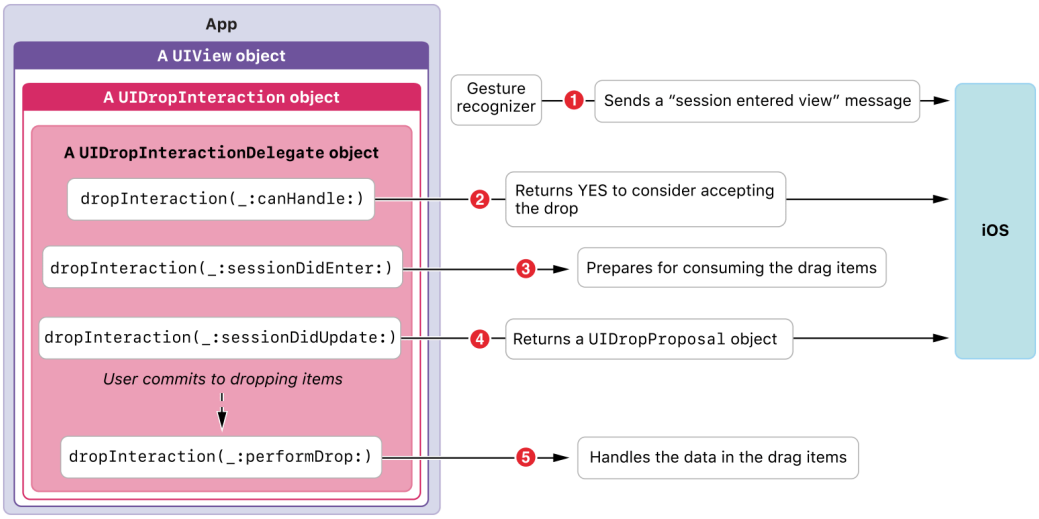
```
optional func dropInteraction(_ interaction: UIDropInteraction,
sessionDidEnter session: UIDropSession)
//用户手指拖拽的 items 「已经进入」了接受 drop item 的区域

optional func dropInteraction(_ interaction: UIDropInteraction,
sessionDidUpdate session: UIDropSession) -> UIDropProposal
//用户手指拖拽的 items 在接受 drop item 的区域内「移动了」

optional func dropInteraction(_ interaction: UIDropInteraction,
sessionDidExit session: UIDropSession)
//用户手指拖拽的 items 「已经离开」了接受 drop item 的区域

optional func dropInteraction(_ interaction: UIDropInteraction, sessionDidEnd
session: UIDropSession)
//session 结束了，用户取消和执行 Drop 最后的回调
```

Drop 操作载入时的预览

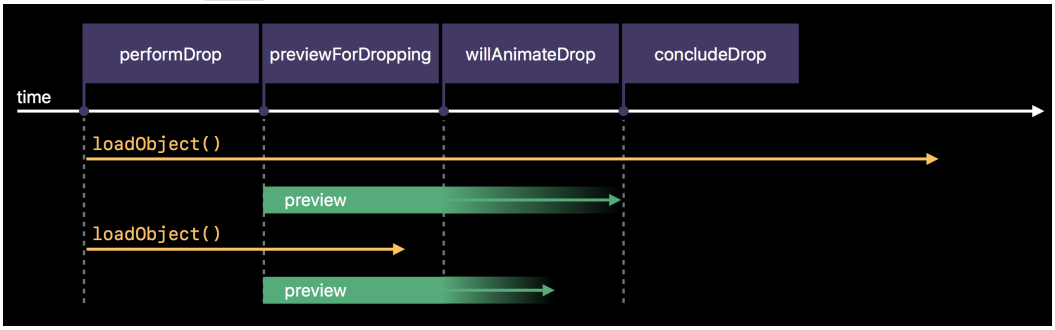
```
optional func dropInteraction(_ interaction: UIDropInteraction,
previewForDropping item: UIDragItem, withDefault defaultPreview:
UITargetedDragPreview) -> UITargetedDragPreview?
//drop 后，在 load过程中，预览视图，一般是一个加载的百分比视图
```

Drop 的生命周期

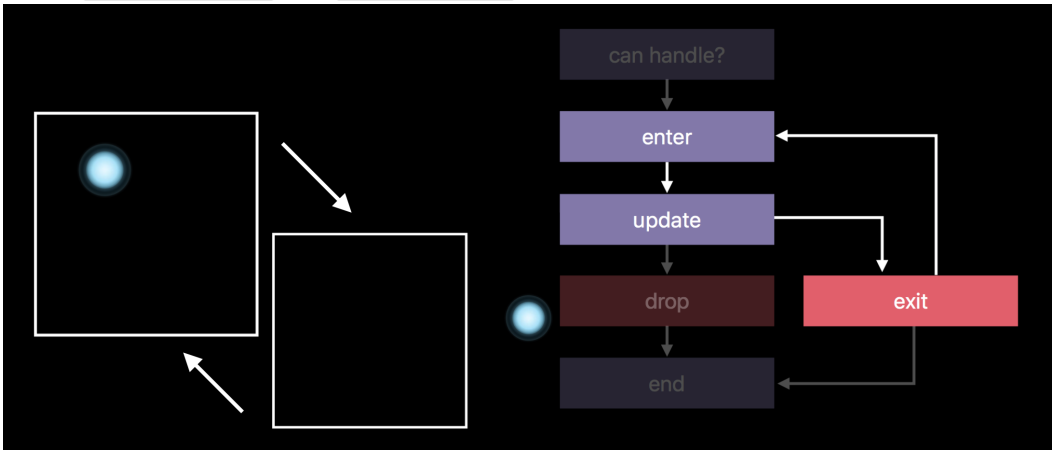
Drop 时整个 iOS 的处理流程



举例而言：一个 **Session** 包含 **3**个 **drop item** (比如一次拖了三个图片)，那么整个 **drop** 操作中包含 **1** 次 `prefrom drop` ，连续 **3** 次 `previewForDropping` ，连续 **3** 次 `willAnimateDropWith` ，最后是 `loadObject` 方法对象处理完成的 **3** 次**闭包回调**，每次时候闭包会传入传输完毕的对象。下图是整个 **Drop Load Object**的时间线中 `Preview` 与 `animator` 的周期示意图，可以帮助理解这段话。不同的对象 `load` 是完全异步的。



上面所讲的执行顺序是一个 Drop 操作顺利完成，如果用户把一个 Drop item 拖进视图又拖出去，会执行 `sessionDidExit` 然后 `sessionDidEnd` ， 此时的操作是

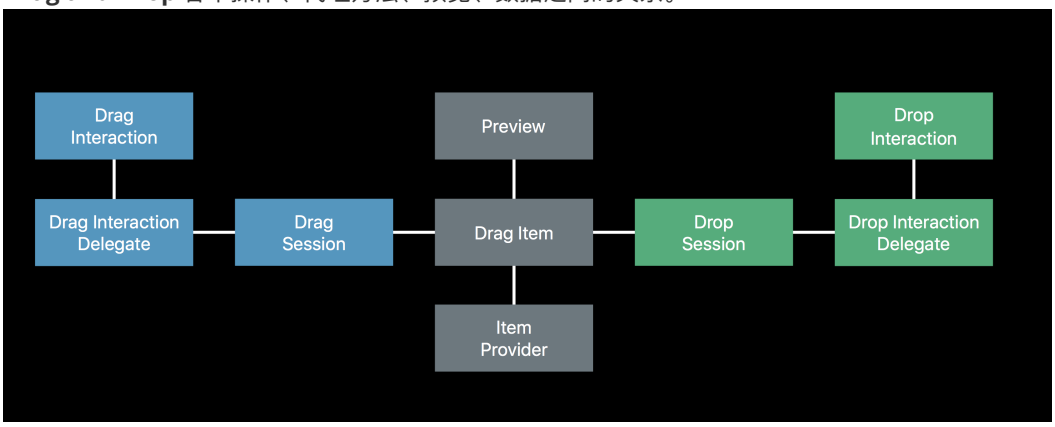


Tips

在慢速传输的情况下， `loadObject` 载入对象可能需要一些时间， `loadObject` 方法本身返回的是一个 `Progress` 对象，所以你可以利用这个 `Progress` 对象来立刻显示一个 `progressView` ，等待载入完毕时候用图像代替。主要保存这个 `progress` ，在慢速加载时，你可以利用这个 `progress` 对象的 `cancel()` 方法来取消加载，提前赋值其 `Progress.cancellationHandler` 闭包，会在取消时候回调。 在一个 `session` 中，如果 `items` 较少，你可以为每个 `item` 客制化 `preview` 和 `animation` ，如果同时家在很多 `items` ，尽量使用默认动画（用同一个 `animator` ）与预览

总结

当您看完此文，并且亲自动手来试试的时候，您也许就能看懂下面这幅图，这幅图反映了整个 **Drag and Drop** 各个操作、代理方法、预览、数据之间的关系。



开发者该知道的 HEVC 与 HEIF

1. HEVC 的基本介绍

HEVC 是一种**视频编码标准**。iOS 中上一代使用的编码标准是 H.264 (视频) / AAC(音频)。

HEVC(High Efficiency Video Coding) 是 AVC(Advanced Video Coding) 演进版本，当然你也可能听过它们一些不同的名字，不过是不同标准化组织的标准编号差异罢了，具体可以看下表：

-	ITU-T (国际电信联盟电信标准化部门)	ISO (国际标准化组织)
AVC	H.264	ISO/IEC: 14496 MPEG-4 Part 10
HEVC	H.265	ISO/IEC: 23008 MPEG-H Part 2

在 Apple 给出的数据中 HEVC 能比 H.264 提高 40% 的压缩率，而在实际使用中，以 iOS 自带相机拍摄 4K 视频为例，视频文件体积由 iOS 10（H.264 / AAC 编码）的 350 MB 每分钟下降到 iOS 11（HEVC / AAC 编码）的 170 MB 每分钟，对生成文件的空间占用率优化十分显著。

1.1 Apple 生态中的 HEVC 支持

Apple 现在支持的 HEVC Codec（编解码器）类型为 `kCMVideoCodecType_HEVC`，即 `hvc1`，对应的 Profiles 为 Main, Main Still Picture 及 Main 10 （即对应 HEVC Version 1 Profiles 中所有的内容），相应的**文件封装格式**（容器）还是我们熟悉 QuickTime Movie (.mov) 及 ISO MPEG-4 (.mp4) 。你可以在新的 iOS 11, tvOS 11 及 macOS 10.13 中使用它。

1.2 HEVC 的硬件要求

软编码 / 软解码 是指使用 CPU 来完成编解码运算，硬编码 / 硬解码 是指使用非 CPU 进行解码运算，常见的就是使用 GPU。相对而言，硬编码 / 硬解码 能够省电、减少设备发热，对于移动设备延长续航有很大的作用。当前如果要在 iOS 10 或者更早设备上解码 HEVC 资源，一般是使用 FFmpeg 这一类的软解码方案（但请注意它的 License 是 LGPL ）。

解码 Decode

新系统下所有 iOS / tvOS / macOS 设备均可执行 8-bit / 10-bit 的软解；在 iOS (tvOS) 上至少需要 A9 芯片来执行 8-bit / 10-bit 硬解码（也就是 iPhone 6s 之后的机型，猜测 9 月也会发布带 A9 / 4K 的 tvOS 新机型），而在 macOS 上至少需要酷睿 6 代来硬解码 8-bit 的资源，或者酷睿 7 代来硬解码 10-bit 的资源。

编码 Encode

新系统下，iOS 需要至少 A10 芯片来完成 8-bit 资源的硬编码（ iPhone 的相机现在仅支持 8-bit 视频的拍摄，有兴趣可以研究一下使用 `AVCaptureVideoDataOutput` 来接收相机输出时的 `videoSettings`）；在 macOS 上，至少需要酷睿 6 代来硬编码 8-bit 的资源，而 10-bit 的软编码则可以在所有的 Mac 上完成。

如果你想要查询当前设备是否支持硬件编码，可以使用下面这个新增的方法来查询。但是当前的测试版似乎还是存在 Bug，笔者使用了安装 iOS 11 beta 2 的 iPhone 7 Plus 及 iPad Pro 10.5 进行测试，均返回了 `false`，而按照 Apple 给出的资料，应当是可以支持硬解的🤷。

```
// 测试设备 iPhone 7 Plus / iOS 11 beta 2 / Xcode 9 beta 2 / Swift 4
let isSupport = VTIsHardwareDecodeSupported(kCMVideoCodecType_HEVC) // false
```

2. HEIF 基本介绍

HEIF(High Efficiency Image File) 是一种**图像文件封装格式**。不要惊讶，HEIF 的先行者是沙滩上的诺基亚，你可以在这里找到相关的信息。诺基亚称其为“Future Container Format for Images and Image Sequences.”。回想 Live Photo 这种形式也并不是由 Apple 首创，但却是由 Apple 推广开来，不禁唏嘘不已。

HEIF 相比 JEPG 能够提高 2 倍的压缩率，而且支持 HEVC 作为其压缩的编解码器；支持透明通道与深度；支持动画（比如动态 GIF , Live Photo）；支持图像序列（比如照片的长曝光）。

HEIF 的国际标准是 ISO/IEC 23008-12（June2015）。

HEIF 文件扩展名在不同内容下的差异：

Payload	Extension
HEVC	.heic
H.264	.avci
any codec	.heif

回想一下 iOS 11 的相册支持 GIF 播放了，真的是 Apple 听到大家的呼声了吗？另外 Live Photo 也不再是`照片.jpg` + `视频.mov` 的组合了。

2.1 HEIF 的硬件要求

新系统下所有的设备都能编解码新的图像格式。只不过要 A9 芯片或是酷睿 6 代以后的才支持硬解码。A10 芯片才支持硬编码。不过毕竟是图像，没有视频那么严格的实时性要求。

3. HEVC 与 HEIF 的兼容性

访问 HEIF 图像，可以说在新的操作系统与系统框架层面是全部支持的，用法与原来并没有什么两样，注意格式选择即可。

访问 HEVC 影片资源，同样也是全部支持，甚至连 HLS(HTTP Live Streaming) 协议都已经扩展完毕（毕竟自家的亲生儿子），想必视频站都会尽早跟进，毕竟相同画质能节约 40% 的带宽，省钱就是盈利啊。

传输时，对用户而言是比较无感的：对于生态外，总是会自动转码为 JPEG/H.264 来获得最好的兼容性（比如 Mail, Share Extensions），对于 Apple 自己生态内，会判定目标设备是否能够解码相应内容而选择是否转码（比如： PTP, AirDrop）。

例如：目前测试的情况，使用 iOS 11 beta 2 通过 AirDrop 传输新格式的照片/视频到 macOS 10.12.5 时，因为目标设备不兼容新的格式，iPhone 端会转码后发送 JPEG/H.264 内容；而当我把相同内容传输到 macOS 10.13 beta 2 时，则可以收到 HEIF/HEVC 内容。

4. 使用 HEVC

示例代码中有部分强制解包与 try! ，仅为简化代码，生产环境请严格使用 if let 与 try catch。

4.1 读取 Access

`PhotoKit` 可以返回 HEVC 资源：

```
// PHImageManager
manager.requestPlayerItem(forVideo: asset, options: nil) { (playerItem,
dictionary) in
    // use AVPlayerItem
}
manager.requestLivePhoto(for: asset, targetSize: size, contentMode: .default,
options: nil) { (livePhoto, dictionary) in
    // use PHLivePhoto
}
manager.requestExportSession(forVideo: asset, options: nil, exportPreset:
preset) { (session, dictionary) in
    // use AVAssetExportSession
}
manager.requestAVAsset(forVideo: asset, options: nil) { (asset, audioMix,
dictionary) in
    // use AVAsset
}
```

或者按需要直接返回二进制数据：

```
// PHAssetResourceManager
resourceManager.requestData(for: assetResource, options: nil,
dataReceivedHandler: { (data) in
    // use Data
}, { (error) in
    // handle Error
})
```

4.2 播放 Playback

这部分没有破坏性的 API 变化，唯一需要考虑的是不同设备的硬件差异导致的播放能力的不同。

AVAssetTrack 新增了一个 `isDecodable` 的实例属性，对于某些非实时解码操作会有帮助。

而对于要能够实时播放的内容，调用 `isPlayable` 查看本机的是否可以播放（毕竟大家都一定有过看幻灯片的经历）。

4.3 拍摄 Capture

4.3.1 拍摄 HEVC 格式的视频

Apple 给出的 Sample Code 是使用手机摄像头拍摄 4K 视频，并使用 `AVCaptureMovieFileOutput` 写入文件。唯一需要确定的就是本机是否包括 HEVC 的编解码器，并在输出设置中作出相应的选择。

```
let session = AVCaptureSession()
session.sessionPreset = .hd4K3840x2160

let camera = AVCaptureDevice.default(.builtInWideAngleCamera, for: nil,
position: .back)
let input = try! AVCaptureDeviceInput(device: camera!)
session.addInput(input)

let movieFileOutput = AVCaptureMovieFileOutput()
session.addOutput(movieFileOutput)

session.startRunning()
movieFileOutput.startRecording(to: url, recordingDelegate: self)

let connection = movieFileOutput.connection(with: .video)
if movieFileOutput.availableVideoCodecTypes.contains(.hevc) {
    outputSetings = [AVVideoCodecKey: AVVideoCodecType.hevc]
} else {
    outputSetings = [AVVideoCodecKey: AVVideoCodecType.h264]
}
movieFileOutput.setOutputSettings(outputSetings, for: connection!)
```

4.3.2 使用 HEVC 拍摄 Live Photo

使用 HEVC 拍摄 Live Photo 时也是类似的操作，并且可以享受这些增强功能：视频防抖、音频回放、30 FPS。

```
let photoSettings = AVCapturePhotoSettings()
photoSettings.livePhotoMovieFileURL = URL(fileURLWithPath: myFilePath)
if photoOutput.availableLivePhotoVideoCodecTypes.contains(.hevc) {
    photoSettings.livePhotoVideoCodecType = .hevc
}
photoOutput.capturePhoto(with: photoSettings, delegate: self)
```

4.3.3 使用 AVAssetWriter 来写入 HEVC 视频文件

同样，指定需要的视频编解码器即可。

```
// AVCaptureVideoDataOutput
// iOS 7
vdo.recommendedVideoSettingsForAssetWriter(writingTo: .mov)
// iOS 11
vdo.recommendedVideoSettings(forVideoCodecType: .hevc,
assetWriterOutputFileType: .mov)
```

4.4 输出 Export

4.4.1 使用 AVAssetExportSession 进行转码

当需要从 H.264 转码为 HEVC 格式时，使用新增的 Preset 即可。

```
AVAssetExportPresetHEVC1920x1080
AVAssetExportPresetHEVC3840x2160
AVAssetExportPresetHEVCHighestQuality
```

4.4.2 使用 AVAssetWriter 编码

为 `AVAssetWriterInput` 的输出设置指定编解码器为 HEVC：

```
settings = [AVVideoCodecKey: AVVideoCodecType.hevc]
```

当然，你也可以使用 `AVOutputSettingsAssistant` 来方便的为指定预设得到输出配置：

```
AVOutputSettingsPreset.hevc1920x1080
AVOutputSettingsPreset.hevc3840x2160
```

使用下面这个新增的方法，在输出设置中查询编码器支持的属性

```
let error = VTCopySupportedPropertyDictionaryForEncoder(
                        3840, 2160,
                        kCMVideoCodecType_HEVC,
                        encoderSpecification,
                        &encoderID, &properties)
if error == kVTCouldNotFindVideoEncoderErr {
    // no HEVC encoder
}
```

Encoder ID 对与一个编码器来说是唯一标识符。Properties 与 encoder ID 可以被用在输出配置中。

4.4.3 使用 VTCompressionSession 来编码采样数据

创建 HEVC 编码器与 H.264 编码器并没有显著区别，指定 Codec 为 `kCMVideoCodecType_HEVC` 即可。

```
let error = VTCompressionSessionCreate(
                        kCFAllocatorDefault,
                        3840, 2160,
                        kCMVideoCodecType_HEVC,
                        encoderSpecification as CFDictionary,
                        nil, nil, nil, nil, // using
VTCompressionSessionEncodeFrameWithOutputHandler
                        &session);
if error == kVTCouldNotFindVideoEncoderErr {
    // no HEVC encoder
}
```

对于 macOS 还有两个额外选项 `kVTVideoEncoderSpecification_EnableHardwareAcceleratedVideoEncoder` 及 `kVTVideoEncoderSpecification_RequireHardwareAcceleratedVideoEncoder`，可以在 `encoderSpecification` 中指定。

5. 使用 HEIF

5.1 HEVC 文件加载

与加载一张 JPEG 图像仅有扩展名的不同：

```
// Read a heic image from file
let inputURL = URL(fileURLWithPath: "/tmp/image.heic")
let source = CGImageSourceCreateWithURL(inputURL as CFURL, nil)
let imageProperties = CGImageSourceCopyPropertiesAtIndex(source, 0, nil) as?
[String: Any]
let image = CGImageSourceCreateImageAtIndex(source, 0, nil)
let options = [kCGImageSourceCreateThumbnailFromImageIfAbsent as String:
true, kCGImageSourceThumbnailMaxPixelSize as String: 320] as [String: Any]
let thumb = CGImageSourceCreateThumbnailAtIndex(source, 0, options as
CFDictionary)
```

5.2 HEIF 文件写入

与写入一张 JPEG 图像也仅有扩展名的区别：

```
// Writing a CGImage to a HEIC file
let url = URL(fileURLWithPath: "/tmp/output.heic")
guard let destination = CGImageDestinationCreateWithURL(url as CFURL,
                                AVFileType.heic as CFString, 1, nil)
    else {
        fatalError("unable to create CGImageDestination")
}

CGImageDestinationAddImage(imageDestination, image, nil)
CGImageDestinationFinalize(imageDestination)
```

5.3 编辑 HEIF 照片，并存为 JPEG 格式

```
// Editing a HEIF photo -- save as JPEG
func applyPhotoFilter(_ filterName: String, input: PHContentEditingInput,
output: PHContentEditingOutput, completion: () -> ()) {

    guard let inputImage = CIImage(contentsOf: input.fullSizeImageURL!)
        else { fatalError("can't load input image") }
    let outputImage = inputImage
        .applyingOrientation(input.fullSizeImageOrientation)
        .applyingFilter(filterName, withInputParameters: nil)

    // Write the edited image as a JPEG.
    do {
    try self.ciContext.writeJPEGRepresentation(of: outputImage,
            to: output.renderedContentURL, colorSpace:
inputImage.colorSpace!, options: [:])
    } catch let error { fatalError("can't apply filter to image: \(error)") }
    completion()
}
```

5.4 编辑 HEVC 视频，并使用 H.264 编码

```
// Editing an HEVC video -- save as H.264
func applyVideoFilter(_ filterName: String, input: PHContentEditingInput,
output: PHContentEditingOutput, completionHandler: @escaping () -> ()) {

    guard let avAsset = input.audiovisualAsset
        else { fatalError("can't get AV asset") }
    let composition = AVVideoComposition(asset: avAsset,
applyingCIFiltersWithHandler: { request in
        let img = request.sourceImage.applyingFilter(filterName,
withInputParameters: nil)
        request.finish(with: img, context: nil)
    })
    // Export the video composition to the output URL.
    guard let export = AVAssetExportSession(asset: avAsset, presetName:
AVAssetExportPresetHighestQuality)
        else { fatalError("can't set up AV export session") }
    export.outputFileType = AVFileType.mov
    export.outputURL = output.renderedContentURL
    export.videoComposition = composition
    export.exportAsynchronously(completionHandler: completionHandler)
}
```

5.5 AVCapturePhotoOutput 的使用

全新的 `AVCapturePhoto`，用来代替 `CMSampleBuffer` 的不足

- 比 CMSampleBuffer 更快
- 绝对的不可变
- 由专用的数据格式支持

AVCapturePhotoCaptureDelegate 新增的代理方法：

```
func photoOutput(_ output: AVCapturePhotoOutput,
                 didFinishProcessingPhoto photo: AVCapturePhoto,
                 error: Error?)
```

AVCapturePhotoCaptureDelegate 废弃的代理方法：

```
func photoOutput(_ output: AVCapturePhotoOutput,
                 didFinishProcessingPhoto photoSampleBuffer: CMSampleBuffer?,
                 previewPhoto: CMSampleBuffer?,
                 resolvedSettings: AVCaptureResolvedPhotoSettings,
                 bracketSettings: AVCaptureBracketedStillImageSettings?,
                 error: Error?)

func photoOutput(_ output: AVCapturePhotoOutput,
                 didFinishProcessingRawPhoto rawSampleBuffer: CMSampleBuffer?,
                 previewPhoto: CMSampleBuffer?,
                 resolvedSettings: AVCaptureResolvedPhotoSettings,
                 bracketSettings: AVCaptureBracketedStillImageSettings?,
                 error: Error?)
```

6. 番外：HEVC 视频流与 RTMP

做视频直播的童鞋如果是自己写的底层，也许会用到下面的内容。

产品经理 / 运营 / 老板一听说 HEVC 在相同画质下能减少 40% 带宽， 于是立马找到了你，表示 11 月能否完成 HEVC 的部署，然后转身走了，留下你一脸懵逼，然后赶紧打开了 Github / Stack Overflow。

客户端和桌面游览器（有 Flash 支持）一般是使用 RTMP 协议播放，而大部分移动游览器是使用 HLS 协议。HLS 是 Apple 的亲儿子，已经有了兼容的更新，服务端只要转码提供 AVC / HEVC 两份资源即可。

然后你很快就发现了问题，RTMP 协议的音视频消息，都是基于 FLV 格式的封装，但是 FLV 不支持 HEVC 啊！具体可以看 Adobe 的文档 。在 Adobe 正式发出扩展更新以前，如果你要使用 HEVC 就必须自己扩展。这部分**只要**和客户端 GG **协商**好就可以了，比如这里有一个案例。当然这些都不是今天的重点。

假设你已经完成了扩展，并通过 HEVC 编码器拿到了视频帧：

```
CMSampleBuffer 0x10430c690 retainCount: 9 allocator: 0x1b3c41dc0
    invalid = NO
    dataReady = YES
    makeDataReadyCallback = 0x0
    makeDataReadyRefcon = 0x0
    formatDescription = <CMVideoFormatDescription 0x1c4452c60 [0x1b3c41dc0]>
{
    mediaType:'vide'
    mediaSubType:'hvc1'
    mediaSpecific: {
        codecType: 'hvc1'       dimensions: 1920 x 1080
    }
    extensions: {<CFBasicHash 0x1c4469cc0 [0x1b3c41dc0]>{type = immutable
dict, count = 2,
entries =>
    0 : <CFString 0x1acc8c8b8 [0x1b3c41dc0]>{contents =
"SampleDescriptionExtensionAtoms"} = <CFBasicHash 0x1c4276d80 [0x1b3c41dc0]>
{type = immutable dict, count = 1,
entries =>
    0 : <CFString 0x1acc94d58 [0x1b3c41dc0]>{contents = "hvcC"} = <CFData
0x104321cd0 [0x1b3c41dc0]>{length = 563, capacity = 563, bytes =
0x010160000000b0000000000078f000fc ... 15e12e0930610789}
}

    2 : <CFString 0x1acc8c918 [0x1b3c41dc0]>{contents = "FormatName"} = HEVC
}
}
}
    sbufToTrackReadiness = 0x0
    numSamples = 1
    sampleTimingArray[1] = {
        {PTS = {40888716474703/1000000000 = 40888.716}, DTS = {INVALID},
duration = {INVALID}},
    }
    sampleSizeArray[1] = {
        sampleSize = 154297,
    }
    sampleAttachmentsArray[1] = {
        sample 0:
            DependsOnOthers = false
    }
    dataBuffer = 0x1c41069c0
```

重点关注 `CMVideoFormatDescription`，其中 `codecType: 'hvc1'` 确认已经是 HEVC 编码成功。

上传推流时，原来使用 H.264(AVC) 时候，提取 `SPS` 和 `PPS` 是使用 `avcC` 字段。而在 HEVC 中明显是用 `hvcC` 字段。直接在视频同步包中使用即可。

播放拉流时，由 ParameterSets 还原为 `CMVideoFormatDescription` 也增加了对应的函数：

```
@available(iOS 11.0, *)
public func CMVideoFormatDescriptionCreateFromHEVCParameterSets(_ allocator:
CFAllocator?, _ parameterSetCount: Int, _ parameterSetPointers:
UnsafePointer<UnsafePointer<UInt8>>, _ parameterSetSizes: UnsafePointer<Int>,
_ NALUnitHeaderLength: Int32, _ formatDescriptionOut:
UnsafeMutablePointer<CMFormatDescription?>) -> OSStatus
```

在 ISO/IEC 14496-15:2017 中你可以找到 hvcC 的数据结构，祝你好运。

```
aligned(8) class HEVCDecoderConfigurationRecord
{
    unsigned int(8) configurationVersion = 1;
    unsigned int(2) general_profile_space;
    unsigned int(1) general_tier_flag;
    unsigned int(5) general_profile_idc;
    unsigned int(32) general_profile_compatibility_flags;
    unsigned int(48) general_constraint_indicator_flags;
    unsigned int(8) general_level_idc;
    bit(4) reserved = '1111'b;
    unsigned int(12) min_spatial_segmentation_idc;
    bit(6) reserved = '111111'b;
    unsigned int(2) parallelismType;
    bit(6) reserved = '111111'b;
    unsigned int(2) chroma_format_idc;
    bit(5) reserved = '11111'b;
    unsigned int(3) bit_depth_luma_minus8;
    bit(5) reserved = '11111'b;
    unsigned int(3) bit_depth_chroma_minus8;
    bit(16) avgFrameRate;
    bit(2) constantFrameRate;
    bit(3) numTemporalLayers;
    bit(1) temporalIdNested;
    unsigned int(2) lengthSizeMinusOne;
    unsigned int(8) numOfArrays;
    for (j=0; j < numOfArrays; j++)
    {
        bit(1) array_completeness;
        unsigned int(1) reserved = 0;
        unsigned int(6) NAL_unit_type;
        unsigned int(16) numNalus;
        for (i=0; i< numNalus; i++)
        {
            unsigned int(16) nalUnitLength;
            bit(8*nalUnitLength) nalUnit;
        }
    }
}
```

到这里，你基本完成了，但是还是要考虑兼容性。iOS 11 并不是所有用户都会马上升级的；其次，A9 以下的设备软解码 HEVC 的性能如何？笔者手边没有设备，不好作出判断。H.264 短时间内还是不会从我们身边消失的，做好长期兼容的准备。

看到这里，你是否应该跟老板说暂时做不了呢？（完）

What's New in Foundation

新 API 概览

- `FileProvider` API 的增强
- 优化了查询剩余空间的 API
- 优化了 `NSString` 的 `Range` 与 `Swift.String` 的 `Range` 的转化
- 从 `NSXPCConnection` 剥离出 `NSProgress`
- iOS 新增温度感应通知的 API

Foundation 的优化

- **NSArray，NSDictionary，NSSet 以及他们的 mutable 类型新增 Copy-On-Write 的行为**。做这个最大的原因是为了更好地桥接到 Swift，当 `NSArray` 从 Objective-C 的 API 返回时，你就会得到一个 Swift 的值类型 `Array`，为了值语义，这些结构体会调用 `NSArray` 的 `copy` 去获取一个副本，如果刚好这个副本是一个 mutable 的类型，那对于性能影响可能会很大。现在我们可以通过 Copy-On-Write 去避免性能损耗，直到这个值实际被改变的时候才去进行 必要的操作。这样做让性能得到大幅提升。
- **Data 的 API 内联化**。Swift Foundation 里的 `Data` 结构体，也做了很多的性能优化。很赞的一件事情是我们可以让 Data 的部分行为直接内联到你的 app 里，例如索引到 data 里的每一个 byte，这对于性能提升的作用也很大。
- **更快的日期计算，更低的内存占用**。
- **提升了 NSNumber 桥街到 Swift 的性能**。这对于提高 Property List 的编码解码性能来说很重要。更多信息，请查看 Efficient Interactions with Frameworks。

Key Paths 与 Key Value Observation

KeyPath 对于 Cocoa 的使用非常重要，因为它可以通过类型的结构，去获取到任意一个实例的相应属性，而且这种方式远比 Block 更加简单和紧凑。

而它还没有完全融入到 Swift 的语言本身里，所以去年我们在 Swift 3 里新增了类型安全的 keyPath，这可以让编译器在编译期去检查 keyPath 的正确性。

让我们来回顾一下之前的写法，首先我们有一个 `Kid` 类型，然后我们通过 keyPath 使用 KVC 去获取和修改 `Kid` 的名字。

```
@objcMembers class Kid: NSObject {
    dynamic var nickname: String = ""
    dynamic var age: Double = 0.0
    dynamic var bestFriend: Kid? = nil
    dynamic var friends: [Kid] = []
}

let ben = Kid(nickname: "Benji", age: 5.5)

let kidsNameKeyPath = #keyPath(Kid.nickname)

let name = ben.valueForKeyPath(kidsNameKeyPath)
// valueForKeyPath(_: String) -> Any

ben.setValue("Ben", forKeyPath: kidsNameKeyPath)
// setValue(_: Any, forKeyPath: String)
```

`kidsNameKeyPath` 最终还是会编译成一个 `String`，它并没有携带任何类型信息，那这会导致什么结果：

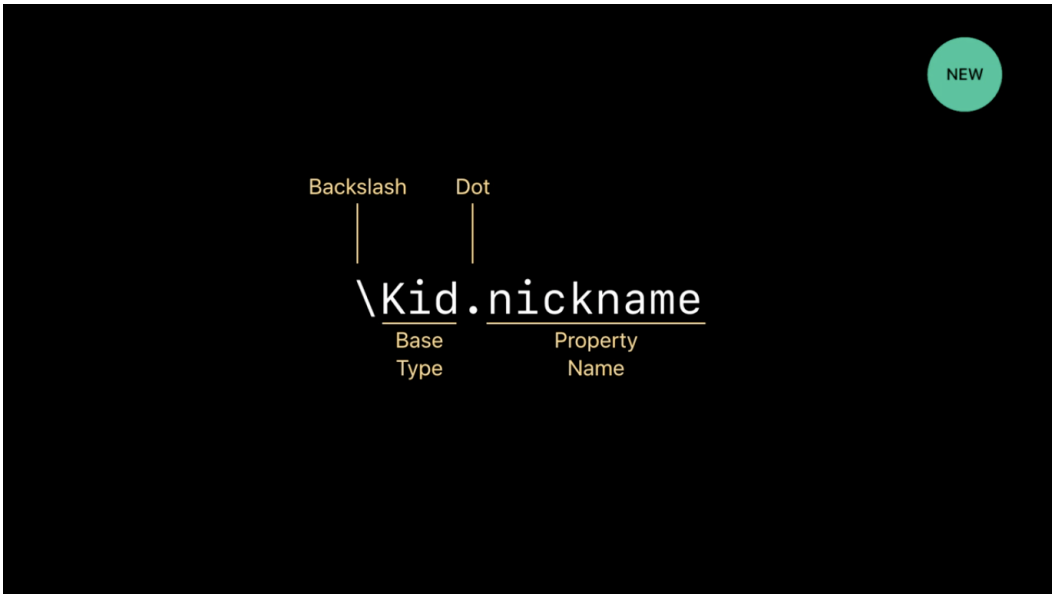
- 我们需要 Objective-C 的 runtime 才能让这个字符串变得有意义，而 Swift 的原生类型就不支持这种做法。
- 由于 keyPath 没有携带类型信息，所以 `valueForKeyPath(_: String) -> Any` 和 `setValue(_: Any, forKeyPath: String)` 就都做不到类型安全。

但这是 Swift，我们可以做得更好，那一个足够 Swifty 的 keyPath 应该是怎么样的：

- 可递归嵌套
- 类型安全
- 足够快
- 适用于所有 Swift 类型
- 所有平台都能够支持（不需要借助 Objective-C 的 runtime）

基础语法

最终的成果就是 Swift Evolution 的 SE-0161 Smart KeyPath 提案，声明的方式如下：



首先是**反斜杠**，以便 keyPath 与其它语法区分开来，接着是**基本类型**，再加一个**点**和**属性名**。

keyPath 支持递归声明，可以声明 property 的 property 的 property……

```
\Kid.nickname.characters.count
```

optional chaining 也可以直接使用

```
\Kid.bestFriend?.nickname
```

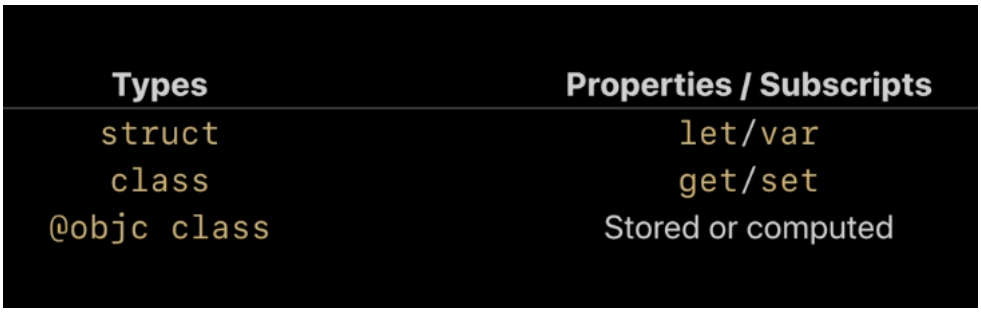
还可以直接使用 subscript

```
\Kid.friends[0]
```

通过类型推导可以在书写时省略掉不必要的元素

```
\Data.[.startIndex]
\.[.startIndex] // 与上面的表达式等价
```

这种 keyPath 为 Swift 所有类型提供了统一的语法格式：



通过 keyPath 读取数据的方式也很简单

```
let age = ben[keyPath: \Kid.age]
```

使用 keyPath 去修改数据的语法也一样

```
ben[keyPath: \Kid.nickname] = "Ben"
```

值类型的 KeyPath 使用

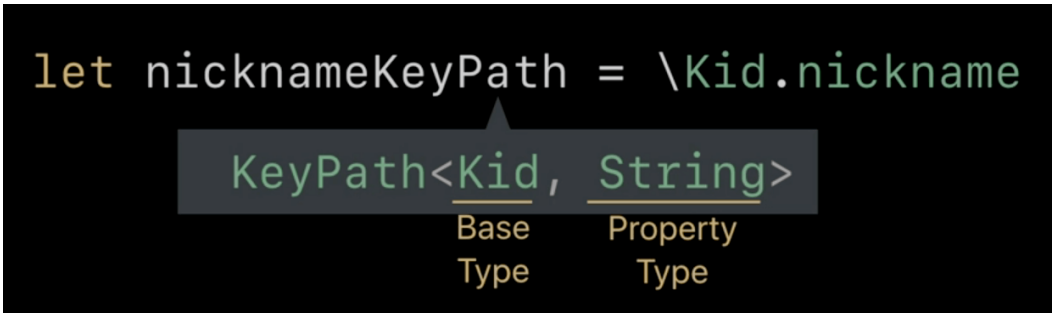
刚刚展示的都是引用类型的 keyPath 使用方法，那为了演示，我们首先定义一个 `BirthdayParty` 结构体。

```
// 使用 Swift 4 的 KeyPaths

struct BirthdayParty {
    let celebrant : Kid
    var theme     : String
    var attending : [Kid]
}

let bensParty = BirthdayParty(
    celebrant : ben,
    theme     : "Construction",
    attending : [])

let birthdayKid = bensParth[keyPath: \.celebrant]

bensParty[keyPath: \.theme] = "Pirate"
```

首先由于这里的 keyPath 很明显基础类型都是 `bensParty`，所以可以省略不写，这里我们看到的只是结果，那实际上到底发生了什么呢?

```
let nicknameKeyPath = \Kid.nickname
```

上面 keyPath 的表达式实际上会产生一个类型实例，那这个实例是什么类型呢？从 Xcode 里直接查看的话，我们可以看到类似于下图的内容：



`KeyPath` 类型的第一个泛型是这个 `KeyPath` 指向的基础类型，而第二个是 `KeyPath` **最终**指向的属性的类型。

同时 `KeyPath` 还支持拼接组合：

```
let particalpantPath = \BirthdayParty.attending
let kidAgeKeyPath = \Kid.age

let allKidsAgeKeyPath = participantPath.appending(kidAgeKeyPath)

let allKidsAge = miasBirthday[keyPath: allKidsAgeKeyPath]
```

当我们需要把 `KeyPath` 保存到数组里面的时候，如果它的基本类型都是一样的，但最终指向的属性类型不同的话，就可以使用 `KeyPath` 的父类 `PartialKeyPath` 去做一个统一的容器：

```
let titles = ["Theme", "Attending", "Birthday Kid"]

// 编译器自动推导为 [PartialKeyPath<BirthdayParty>]
let partyPaths = [
    \BirthdayParty.theme,
    \BirthdayParty.attending,
    \BirthdayParty.celebrant
]

for (title, partyPath) in zip(titles, partyPaths) {
    let partyValue = miasParty[keyPath: partyPath]
    print("\(title): \(partyValue)\n")
}
```

如果需要通过 KeyPath 去修改一个值类型的变量的时候，我们需要使用的类型是 `WritableKeyPath`。

```
let kidNicknameKeyPath: WritableKeyPath<Kid, String> = \Kid.nickname
ben[keyPath: kidNicknameKeyPath] = "Ben"
```

但我们使用 `WritaleKeyPath` 的时候，可能会出现这样的情况:

```
@objcMembers class Kid : NSObject {
    dynamic var age : Double = 0.0
    ...
}

struct BirthdayParty {
    var celebrant : Kid
    ...
}

extension BirthdayParty {
    func blowCandles(ageKeyPath: WritableKeyPath<BirthdayParty, Double>){
        let age = self[keyPath: ageKeyPath]
        self[keyPath: ageKeyPath] = age + 1 // 编译不通过
    }
}

bensParty.blowCandles(ageKeyPath: \.celebrant.age)
```

这里通过 `WritableKeyPath` 去修改 `bensParty` 的 `kid` 的属性，而 `kid` 是引用类型，这个修改实际上并不会改变 `bensParty` 的值。

当使用 `WritableKeyPath` 的基础类型为**值类型**的时候，通过这个 keyPath 去修改值，会被认为是修改了。

在这里解决方法很简单，改为使用 `ReferenceWritableKeyPath` 这个类型就可以正确地表达这样的含义了。

```
func blowCandles(ageKeyPath: ReferenceWritableKeyPath<BirthdayParty, Double>)
{ ... }
```

总结一下两种 keyPath 的区别

- **WritableKeyPath**。基于这种 keyPath 的修改意味着值的修改
- **ReferenceWritableKeyPath**。基于这种 keyPath 的修改意味着引用的修改

keyPath 会捕获值，而不像 block 那样

```
var index = 0
let whickKidKeyPath = \BirthdayParty.attendees[index]
let firstAttendeesAge = partyPersonAge(party, whichKidKeyPath)

index = 1
print(whichKidKeyPath) // \BirthdayParty.attendess[0]
```

keyPath 并不会捕获 `index` 的引用，而是直接捕获了 `index` 的值，所以 keyPath 不会随着 `index` 的修改而修改。

KVO 的语法得到了简化

observe 方法返回的是一个 `NSKeyValueObservation`，block 里参数分别是被观察的对象，便于我们操作的时候不容易产生引用循环，以及一个 `NSKeyValueObservedChange` 对象。

```
let observation = ben.observe(\.age) { (ben, chagne) in
    ...
}
```

`NSKeyValueObservation` 可以手动调用 `invalidate` 方法注销掉 KVO，它在被销毁的时候也会自动去调用 `invalidate`，正常情况下找个生命周期合适的对象保存好它就行了（例如 ViewController）。

然后 `NSKeyValueObservedChange` 封装了改动的信息，包括新值旧值，修改类型，不需要像以前那样需要手动从字典里取出来了。

```
public struct NSKeyValueObservedChange<Value> {

    // change 的类型
    public typealias Kind = NSKeyValueChange
    public let kind: NSKeyValueObservedChange.Kind

    // 新值和旧值
    public let newValue: Value?
    public let oldValue: Value?

    public let indexes: IndexSet?

    // `isPrior` 为 true 时，observation 会在实际修改之前触发
    // `isPrior` 的值由 `observe` 方法传入的参数决定
    public let isPrior: Bool
}
```

Encoding and Decoding

编码解码，其实就是格式化数据与 Swift 的数据结构相互转化的过程，而 Swift 的强类型特性与结构松散的格式化数据显得格格不入，特别是 JSON，Swift 团队认为只有让语言本身去处理这个问题才是最合适的，让编译器，标准库帮助一起完成这个过程。

现在 JSON 解析的操作非常简单，只要让我们需要解析的类型遵循 `Codable` 就可以了：

```
let jsonData = """
{
    "name"  : "Monalisa Octocat",
    "email" : "support@github.com",
    "date"  : "2011-04-14T16:00:49Z"
}
""".data(using: .utf8)!

// 1. 让我们需要解析的类型遵循 Codable 协议
struct Author : Codable {
    let name  : String
    let email : String
    let date  : Date
}

// 2. 创建一个解码器
let decoder = JSONDecoder()

// 3. 设置一下日期编码的形式
decoder.dateDecodingStrategy = .iso8601

// 4. 传入目标类型和数据进行解码
let author = try decoder.decode(Author.self, from: jsonData)
```

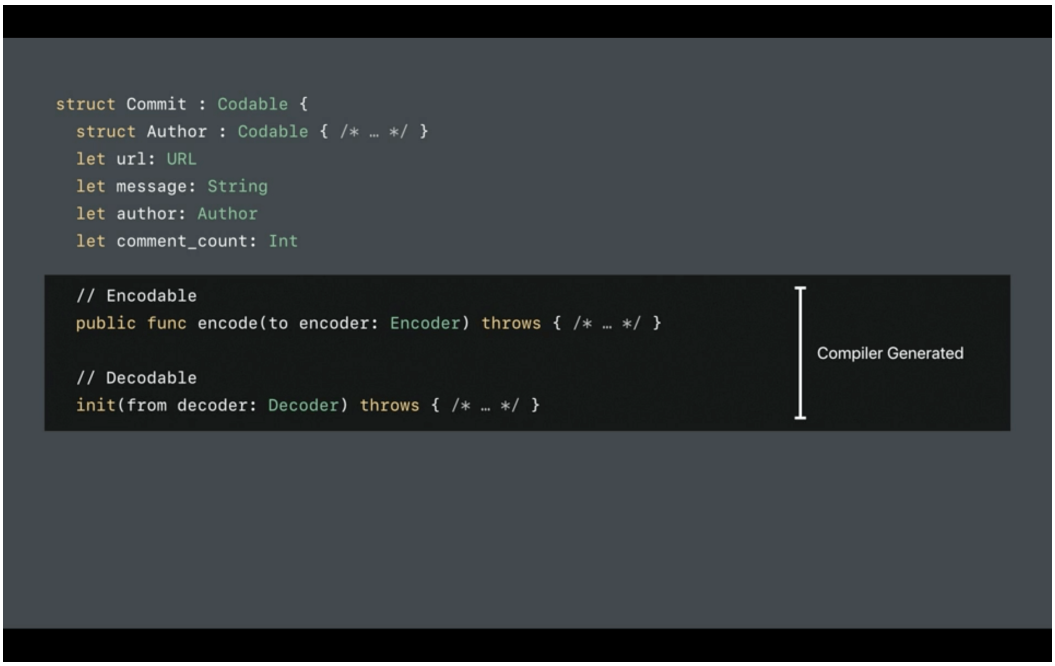
Codable

`Codable` 实际上包含了两个协议，`Encodable` 和 `Decodable`，实现如下：

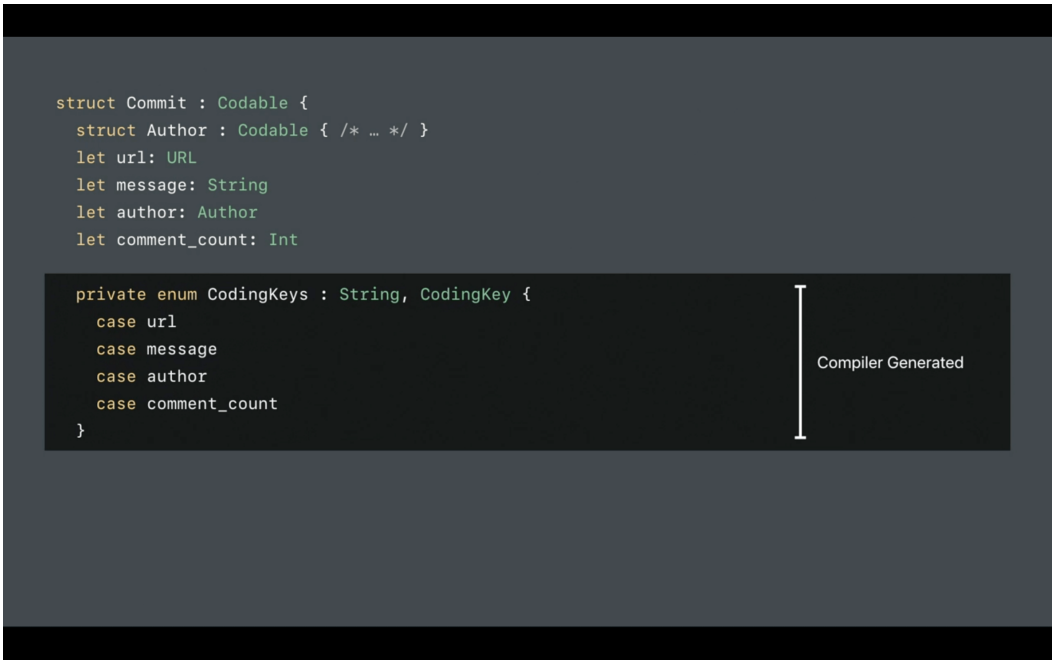
```
typealias Codable = Encodable & Decodable

public protocol Encodable {
    func encode(to encoder: Encoder) throws
}

public protocol Decodable {
    init(from decoder: Decoder) throws
}
```

`Codable` 包含了两个协议 `Encodable` 和 `Decodable`。

Coding 相关的这些 protocols 都借助了 protocol extension，让我们可以有自定义的实现，也可以直接使用默认的实现。



只要让我们自定义的类型遵循 `Codable`，编译器就会自动为我们插入 `encode(to:) throws` 和 `init(from:) throws` 的实现。这两个方法我们目前都不需要自定义，那么直接忽略掉就行了，但是我们发现 `comment_count` 并不符合 Swift 的命名规则，这个时候就需要来关注一下编译器为我们自动插入的另一段内容，一个 `private` 的 `CodingKeys` 枚举，遵循 `CodingKey` 协议（稍后我们会更深入地探讨这个协议）。



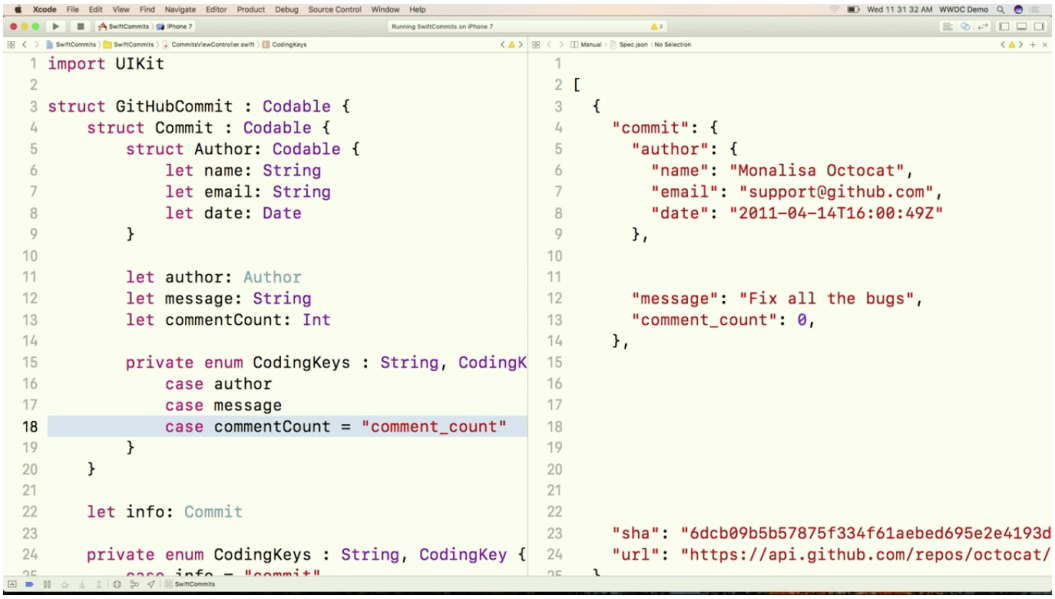
`CodingKeys` 里包含了四个 case，跟我们声明的四个属性名字一样，当我们需要自定义的时候，只需要修改相应的 case 即可。

```
struct Commit : Codable {
    struct Author : Codable { ... }
    let url           : URL
    let message       : String
    let author        : String
    let commentCount : Int

    private enum CodingKeys : String, CodingKey {
        case url
        case message
        case author
        case commentCount = "comment_count"
    }
}
```

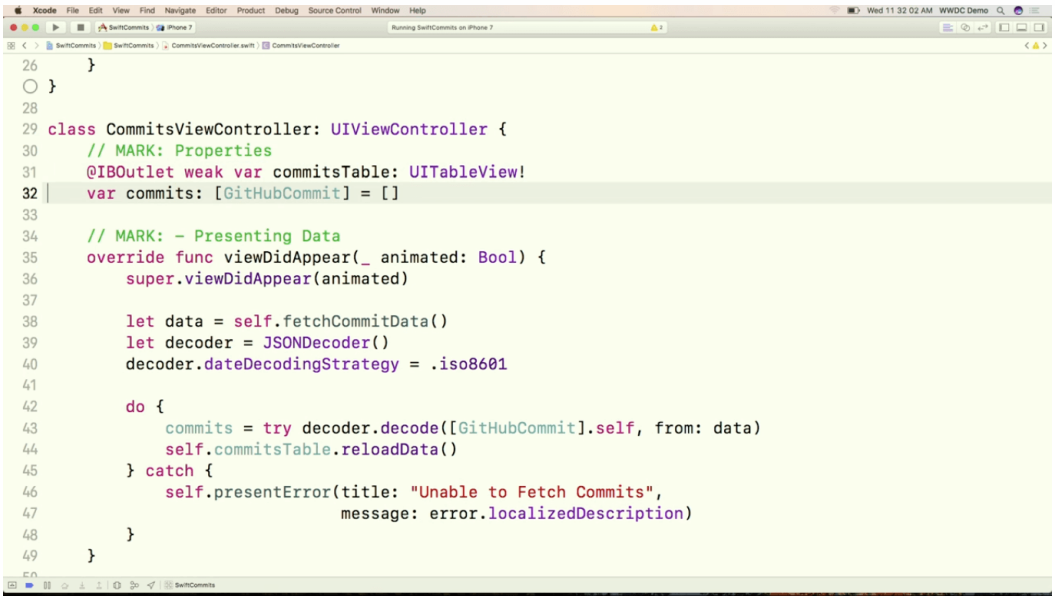
把属性名和 case 名都改为 `commentCount`，然后把 `commentCount` 的 `rawValue` 改为数据里对应的 key 就可以了。

如果类型的属性全部都遵循 `Codable` 的话，我们只要让自己的类型遵循 `Codable`，其它的交给编译器就可以了，而且标准库里面绝大部分基本数据类型现在都已经实现了 `Codable` 协议，所以基本上我们一句代码就能完成 JSON 解析的功能了。

实战例子



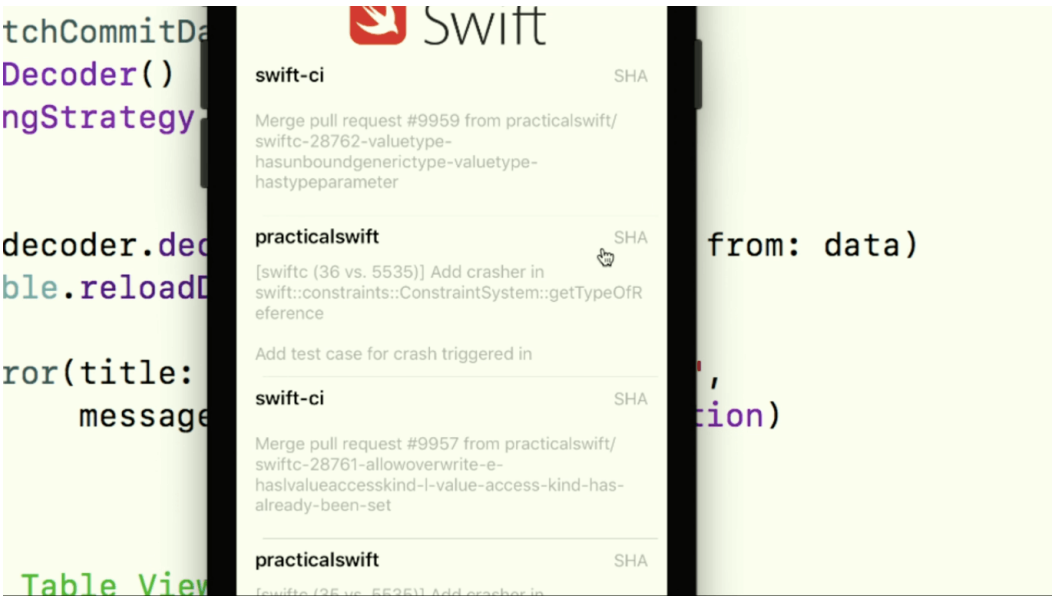
这里我们来看一个 App，这个 App 让我们能够查看 Swift repo 的提交。这里的 Commit 比起之前复杂了一点，右边是服务器返回的的 JSON 数据。



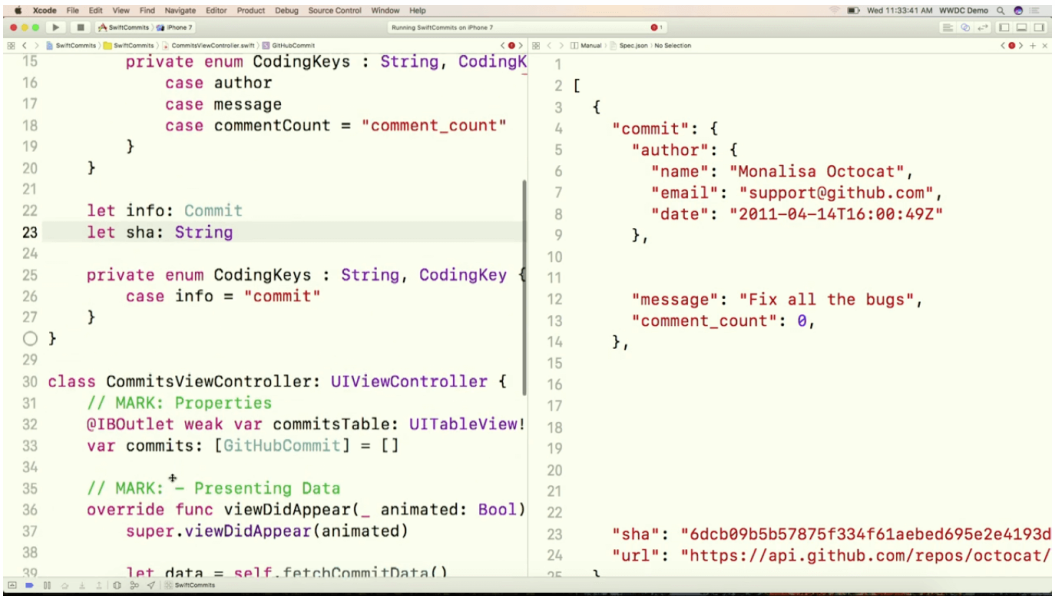
```swift
    }
}

class CommitsViewController: UIViewController {
    // MARK: Properties
    @IBOutlet weak var commitsTable: UITableView!
    var commits: [GitHubCommit] = []

    // MARK: - Presenting Data
    override func viewDidAppear(_ animated: Bool) {
        super.viewDidAppear(animated)

        let data = self.fetchCommitData()
        let decoder = JSONDecoder()
        decoder.dateDecodingStrategy = .iso8601

        do {
            commits = try decoder.decode([GitHubCommit].self, from: data)
            self.commitsTable.reloadData()
        } catch {
            self.presentError(title: "Unable to Fetch Commits",
                              message: error.localizedDescription)
        }
    }
```

App 的逻辑很简单，分成下面几步

1. 获取 Commit 的数据
2. 创建一个 Decoder
3. 设置 decoder 的日期解码格式
4. 由于 decode 方法会抛出错误，所以我们需要用一个 do catch 来包住它
    - 如果解码成功，我们就把数据填充到 commits 数组里，刷新 UI 数据
    - 如果解码出错，我们就把错误原因呈现出来



界面里特意留了一个让 SHA 值显示的空位，那先让我们把 Commit 的 SHA 值显示出来吧



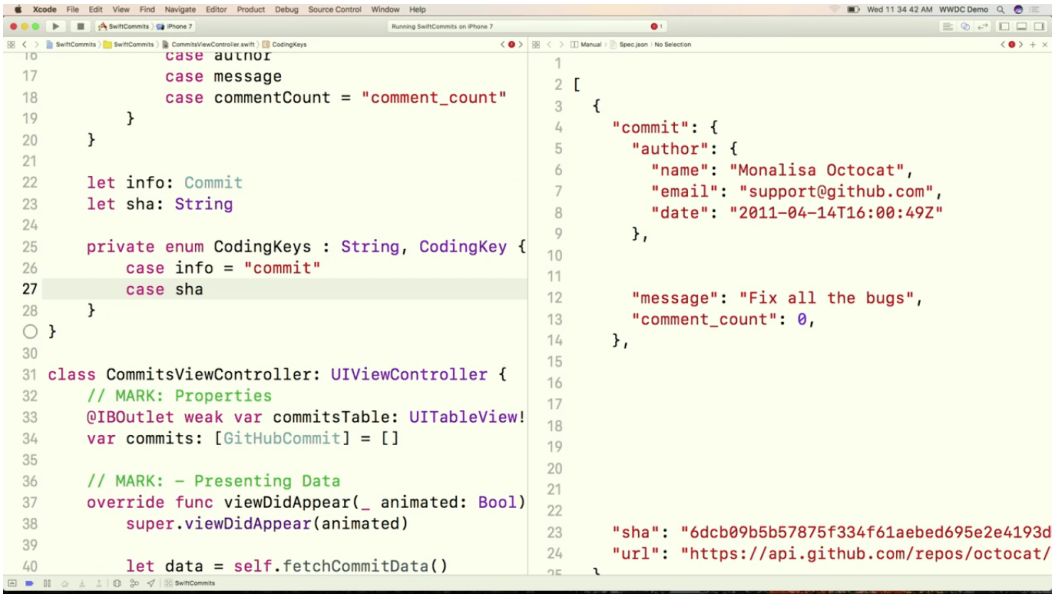
```
        private enum CodingKeys : String, CodingK
            case author
            case message
            case commentCount = "comment_count"
        }
    }

    let info: Commit
    let sha: String

    private enum CodingKeys : String, CodingKey {
        case info = "commit"
    }
}

class CommitsViewController: UIViewController {
    // MARK: Properties
    @IBOutlet weak var commitsTable: UITableView!
    var commits: [GitHubCommit] = []

    // MARK: - Presenting Data
    override func viewDidAppear(_ animated: Bool)
        super.viewDidAppear(animated)

        let data = self.fetchCommitData()
```

```
[
  {
    "commit": {
      "author": {
        "name": "Monalisa Octocat",
        "email": "support@github.com",
        "date": "2011-04-14T16:00:49Z"
      },

      "message": "Fix all the bugs",
      "comment_count": 0,
    },

    "sha": "6dcb09b5b57875f334f61aebed695e2e4193d
    "url": "https://api.github.com/repos/octocat/
```

首先我们给 Commit 添加一个 `sha` 属性，但是编译的时候我们会得到一个编译错误，原因是这样的，`GitHubCommit` 类型里的 `CodingKeys` 枚举可以用来控制自动生成的 `init(decoder: Decoder) throws` 方法的具体实现。

在这里我们把 `info` 属性映射到 JSON 数据里的 `commit` 字段，但是，编译器发现我们新增的 `sha` 属性在 `CodingKeys` 里没有相应的 case。如果属性有默认值，而且没有写入 `CodingKeys` 里的话，decode 跟 encode 方法就会自动忽略掉这个属性。

不过由于 `sha` 属性没有默认值，编译器尝试根据 `CodingKeys` 为我们生成 `init(decoder: Decoder) throws` 方法的时候，就会发现 `sha` 没有合理的值，编译器就会抛出一个错误。



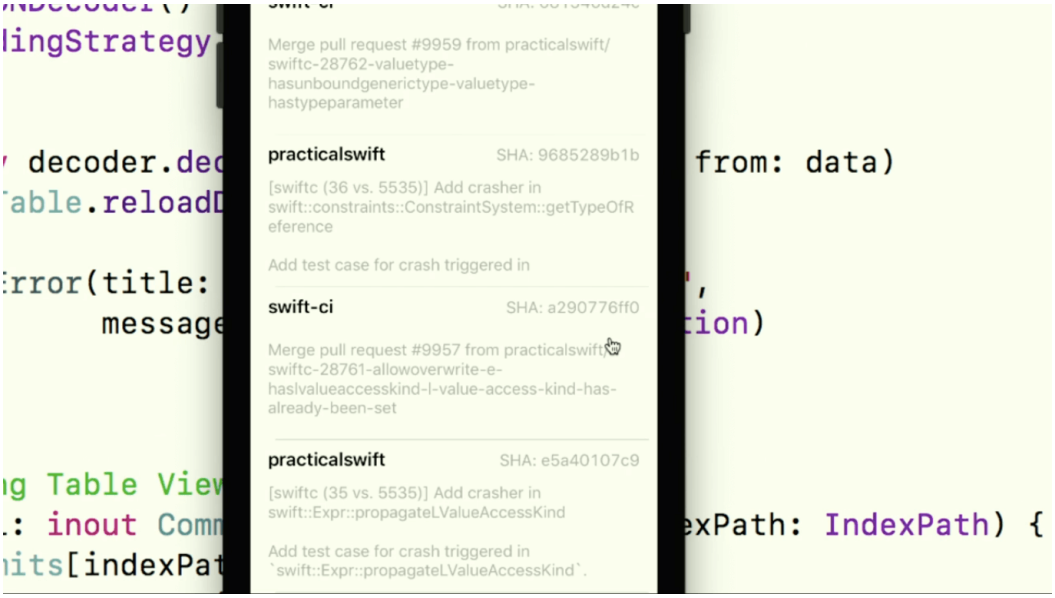
```
            case author
            case message
            case commentCount = "comment_count"
        }
    }

    let info: Commit
    let sha: String

    private enum CodingKeys : String, CodingKey {
        case info = "commit"
        case sha
    }
}

class CommitsViewController: UIViewController {
    // MARK: Properties
    @IBOutlet weak var commitsTable: UITableView!
    var commits: [GitHubCommit] = []

    // MARK: - Presenting Data
    override func viewDidAppear(_ animated: Bool)
        super.viewDidAppear(animated)

        let data = self.fetchCommitData()
```

```
[
  {
    "commit": {
      "author": {
        "name": "Monalisa Octocat",
        "email": "support@github.com",
        "date": "2011-04-14T16:00:49Z"
      },

      "message": "Fix all the bugs",
      "comment_count": 0,
    },

    "sha": "6dcb09b5b57875f334f61aebed695e2e4193d
    "url": "https://api.github.com/repos/octocat/
```

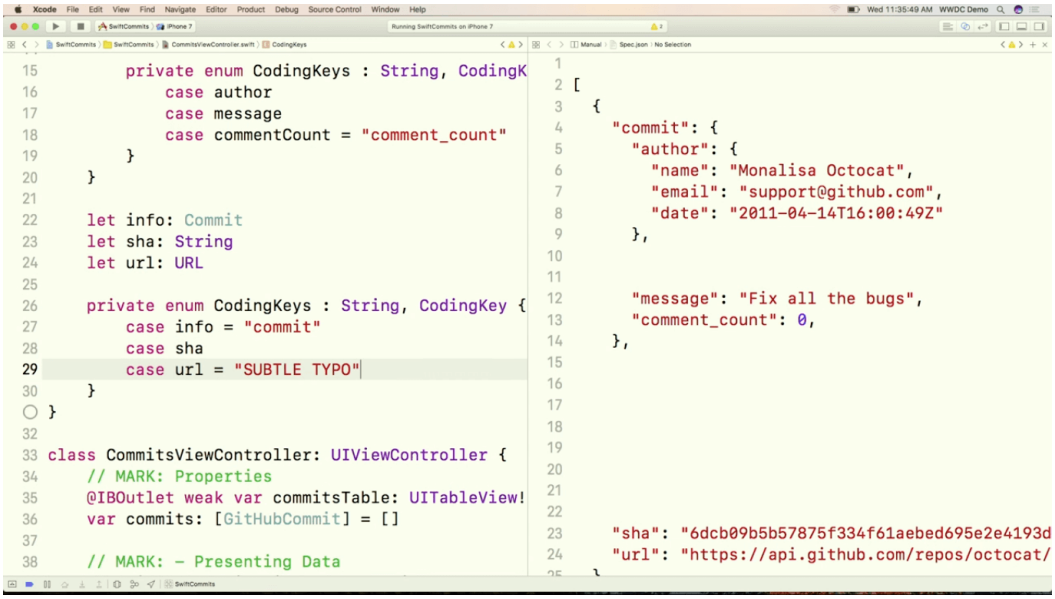
这里我们想让 `sha` 参与 Decode，所以直接加上相应的 case 就可以了.

```swift
override func viewDidAppear(_ animated: Bool) {
    super.viewDidAppear(animated)

    let data = self.fetchCommitData()
    let decoder = JSONDecoder()
    decoder.dateDecodingStrategy = .iso8601

    do {
        commits = try decoder.decode([GitHubCommit].self, from: data)
        self.commitsTable.reloadData()
    } catch {
        self.presentError(title: "Unable to Fetch Commits",
                          message: error.localizedDescription)
    }
}

// MARK: - Customizing Table View Cells
func setUpCell(_ cell: inout CommitsTableViewCell, at indexPath: IndexPath) {
    let commit = commits[indexPath.row]
    cell.authorLabel.text = commit.info.author.name
    cell.messageLabel.text = commit.info.message
    cell.shaLabel.text = "SHA: \(commit.sha.prefix(10))"
}
}
```

然后我们在 UI 里呈现出来，得到结果：



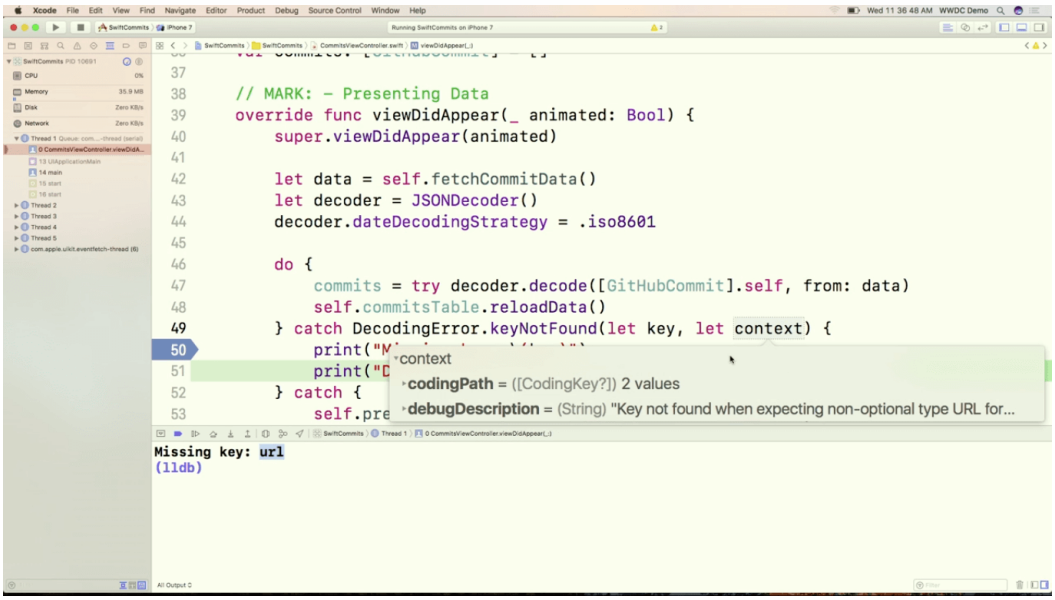
我们还有最后一个属性没有 decode，让我们完成它



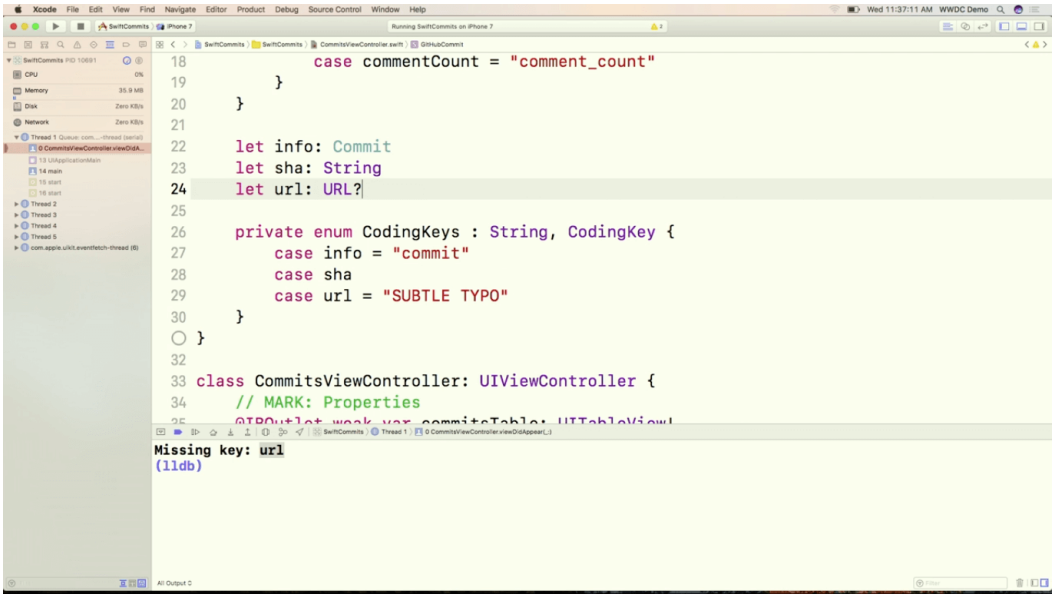
```swift
private enum CodingKeys : String, CodingKey {
    case author
    case message
    case commentCount = "comment_count"
}
}

let info: Commit
let sha: String
let url: URL

private enum CodingKeys : String, CodingKey {
    case info = "commit"
    case sha
    case url = "SUBTLE TYPO"
}
}

class CommitsViewController: UIViewController {
// MARK: Properties
@IBOutlet weak var commitsTable: UITableView!
var commits: [GitHubCommit] = []

// MARK: - Presenting Data
```

```json
[
  {
    "commit": {
      "author": {
        "name": "Monalisa Octocat",
        "email": "support@github.com",
        "date": "2011-04-14T16:00:49Z"
      },

      "message": "Fix all the bugs",
      "comment_count": 0,
    },

    "sha": "6dcb09b5b57875f334f61aebed695e2e4193d
    "url": "https://api.github.com/repos/octocat/
```

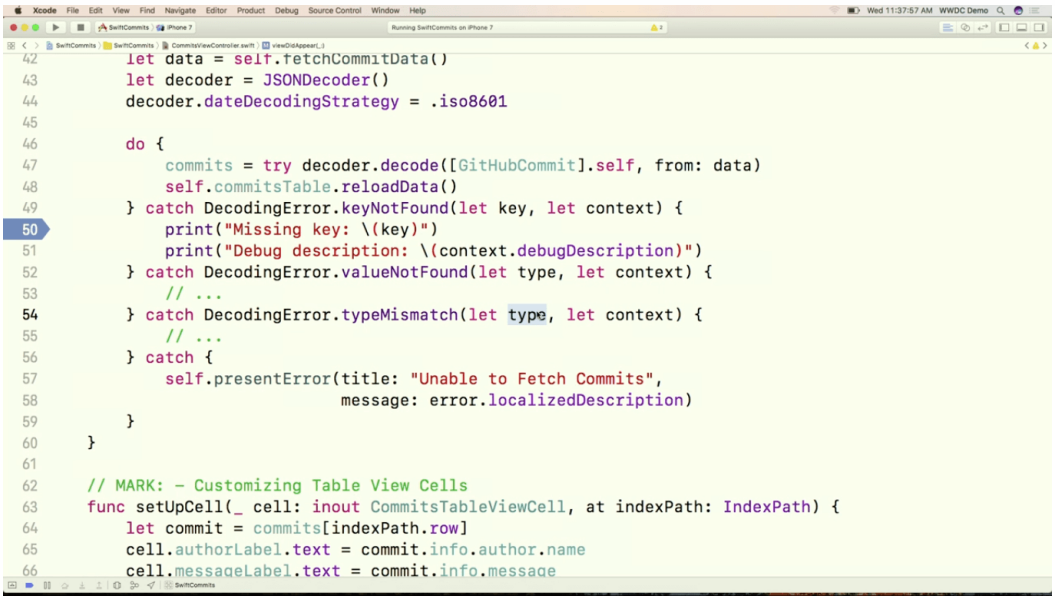
但在这里，我们尝试给 url 一个错误的 rawValue，很明显 JSON 里会找不到相应的 key，decode 的时候会抛出错误.



为了处理这个错误，我们尝试捕获 `DecodingError.keyNotFound` 的错误，这里的 key 是我们尝试访问但找不到的 key，而 `context` 会告诉我们哪里出错了，里面包含了两个很有用的信息，`codingPath` 会告诉我们是解析到 JSON 的哪个位置出现这个问题，而 `debugDescription` 里会解释具体出错的原因。



如果我们其实不是很介意 `url` 在 JSON 里有没有相应的字段的话，可以直接使用 optional，这样相当于给了 `url` 一个 nil 的默认值。



```
        let data = self.fetchCommitData()
        let decoder = JSONDecoder()
        decoder.dateDecodingStrategy = .iso8601

        do {
            commits = try decoder.decode([GitHubCommit].self, from: data)
            self.commitsTable.reloadData()
        } catch DecodingError.keyNotFound(let key, let context) {
            print("Missing key: \(key)")
            print("Debug description: \(context.debugDescription)")
        } catch DecodingError.valueNotFound(let type, let context) {
            // ...
        } catch DecodingError.typeMismatch(let type, let context) {
            // ...
        } catch {
            self.presentError(title: "Unable to Fetch Commits",
                              message: error.localizedDescription)
        }
    }

    // MARK: - Customizing Table View Cells
    func setUpCell(_ cell: inout CommitsTableViewCell, at indexPath: IndexPath) {
        let commit = commits[indexPath.row]
        cell.authorLabel.text = commit.info.author.name
        cell.messageLabel.text = commit.info.message
```

除了 `keyNotFound` 之后，还有 `valueNotFound` 找不到相应的值，或者 `typeMismatch` 值的类型不相符之类的错误。有了这些我们就可以很轻松地定位到错误的地方。

Codable 的哲学

在更进一步解释 `Codable` 之前，我想引出 `Codable` 的三个设计哲学思想。

内建错误处理机制

Swift 团队希望编码和解码能够有一套错误处理机制，就像我们刚刚在 demo 里看到的那样

- **处理不合法的输入时，应该是思考“什么时候处理”，而不应该是“该不该处理”**。
- **不应该导致崩溃，除非是开发者的错误（注意，不是失误）**。错误可能源于 API 改变，数据错误等等，所以决定在处理未知数据时不应该产生崩溃，但如果检测到是由于开发者的错误而导致的解码错误，就还是会 fatalError，而且带上具体的出错原因。
- **编码和解码都可能会产生错误**。对于其他错误，我们就使用错误处理机制去解决。

那就首先看一下 Coding 时会产生的 Error：

Encoding 编码

Encoding 很简单，就只有一种错误

- **Invalid Value**。不合法的值，例如 JSON 就不支持 `NaN`，`infinity`，尝试把这些值编码进 JSON 里就会产生这个错误。

Decoding 解码

Decoding 就比较复杂，总共有四种情况，前面三种我们在 demo 里都见过了，最后一种 Data corrupt 主要是用来囊括其它所有情况，前面我们已经看过它的具体用法了。

- **Missing key**：没有相应的 key。
- **Missing value**：没有相应的值。
- **Type mismatch**：类型错误。
- **Data corrupt**：数据错误。

数据的解码可以分为这么几个阶段：

1. **Bytes**：二进制数据。
2. **Structured bytes**：结构化数据。例如这里是 JSON 的话，就会检查这是不是一段合法的 JSON 数据，首先检查是不是一段字符串，然后格式是否正确，等等一系列的检查，如果不合法的话，JSON decoder 就会在这个阶段抛出错误。
3. **Typed data**：Swift 类型数据。这里会通过结构化数据编码出一个 Swift 的类型实例出来。

4. **Domain specific validation (可选)**：数据验证。前面的阶段我们只验证了有没有值，以及值的类型是否正确，那到了这一步，我们就可以验证值的合理性，例如年龄，我们就需要保证它必须在 0 到 100 的范围内。
5. **Graph-level validation (可选)**：整体验证。这个时候数据的验证就不局限于数据本身的合法性，而是跟实际情景关联起来，例如我们明明请求的是 Swift 的文档，却返回了 Objective-C 的文档给我们，作为文档本身，它是合理的，可以正常地编码成一个 Swift 类型数据，本身的数据的值也是合理的，但套入到当前的 context 里，它很明显就是一段错误的数据。

那让我们拿之前写过的 Commit 来讲解一下这个过程吧，之前我们讲了如何自定义 `CodingKeys` 来让属性和数据产生映射关系，现在让我们来自定义 `init(from:) throws` 方法吧

```
struct Commit : Codable {
    struct Author : Codable { ... }
    let url           : URL
    let message       : String
    let author        : Author
    let commentCount : Int
    private enum CodingKeys: String, CodingKey { ... }

    public init(from decoder: Decoder) throw {
        let container = try decoder.container(keyedBy: CodingKeys.self)

        url = try container.decode(URL.self, forKey: .url)
        message = try container.decode(String.self, forKey: .message)
        author = try container.decode(Author.self, forKey: .author)
        commentCount = try container.decode(Int.self, forKey: .commentCount)
    }
}
```

首先我们通过 decoder，传入 `CodingKeys` 获取到一个 `container`， `container` 会从结构化数据里取与 `CodingKeys` 对应的那一部分数据。

接着我们从 `container` 里取出我们所需的数据。这里假设我们需要保证 `url` 使用的是 https 协议，那我们怎么去验证它呢?

```
public init(from decoder: Decoder) throw {
    let container = try decoder.container(keyedBy: CodingKeys.self)

    url = try container.decode(URL.self, forKey: .url)
    guard url.scheme == "https" else {
        throw DecodingError.dataCorrupted(DecodingError.Context(
            codingPath : container.codingPath + [CodingKeys.url],
            debugDescription : "URL 需要是 https 协议的"))
    }

    message = try container.decode(String.self, forKey: .message)
    author = try container.decode(Author.self, forKey: .author)
    commentCount = try container.decode(Int.self, forKey: .commentCount)
}
```

这里很简单，使用 `url` 的 api，如果发现不是 https 的话就抛出错误。我们还注意到 `url` 在 json 里其实是字符串，而且 URL 可以使用这段字符串进行自我解码，如果这段字符串不合法的话，URL 就会抛出错误，然后由于 Swift 的错误处理机制，这个错误最终就会在 `url` 解析那一行抛出。

封装编码的细节

- **隐藏键值对**。我们发现，类型在编码的时候不让类型本身知道，结构化数据里使用了什么类型的键值对，很重要。因为这样可以让我们只关注需要的数据。
- **使用容器来封装数据**。Swift 提供了一个中间容器，结构化数据会先塞到容器里，然后再由我们

去从容器里取出数据去编码出一个类型数据，这样在编码类型数据的时候就不必了解数据本身到底是使用了 JSON 还是其它什么编码。

Swift 目前只提供了 3 种基础容器：

- **Keyed Containers**：保存了多个键值对，是大部分情况下的最优选择，因为它有良好的兼容性。
- **Unkeyed Containers**：用来保持有序的数据。
- **Single Value Containers**：只保存指向原始数据的指针，这是兼容性最好的一种容器，个人觉得主要是为了保留 `container` 的抽象，然后把这一部分数据转化的过程交给了程序员去完成。

现在让我们来看一下这里的 key 到底是什么？

```
public protocol CodingKey {
    var stringValue : String { get }
    var intValue    : Int?   { get }

    init?(stringValue: String)
    init?(intValue: Int)
}
```

这里的 `stringValue` 在应对类似于 JSON 这样的数据时就很有用，而 `intValue` 主要是对性能要求比较高的时候，使用 `intValue` 可以优化二进制层面数据，获取到更高的性能表现。

```
private enum CodingKeys : String, CodingKey {
     case url
     case author
     case commentCount = "comment_count"
}

CodingKeys.commentCount.stringValue // "comment_count"
CodingKeys.commentCount.intValue    // nil
```

而平时使用 `CodingKey` 协议，基本上就是遵守协议，然后让编译器去完成解析来的工作就可以了，在这我们可以看到 `stringValue` 基本上就是枚举 case 的名字，而 `intValue` 因为枚举的原始值是 `String` 所以为 nil。

而在我们自定义 case 名之后，`stringValue` 还是不变，保持 GitHub API 里的数据一致。

```
private enum CodingKeys : String, CodingKey {
    case url = 22
    case author = 100
    case comment_count
}

CodingKeys.commentCount.stringValue
// "comment_count"

CodingKeys.commentCount.intValue
// 101
```

如果你是在封装库的话，就更加推荐使用 Int 来作为原始值，会有更好的性能表现。

实例讲解

接下来我们通过再回到 Commit 的例子里，之前我们看了 `CodingKeys` 以及 `init(from:) throws` 的实现，现在来看一下 `encode(to:) throws`：

```
struct Commit : Codable {
    struct Author : Codable { ... }
    let url         : URL
    let message     : String
    let author      : Author
    let commentCount: Int
    private enum CodingKeys : String, CodingKey { ... }
    public init(from decoder: Decoder) throw { ... }

    public func encode(to encoder: Encoder) throws {
        var container = encoder.container(keyedBy: CodingKeys.self)
        try container.encode(url, forKey: .url)
        try container.encode(message, forKey: .message)
        try container.encode(author, forKey: .author)
        try container.encode(commentCount, forKey: .commentCount)
    }
}
```

这里没有做什么自定义的东西，跟编译器产生的实现其实是一样的，主要是为了让大家更了解这整个过程：

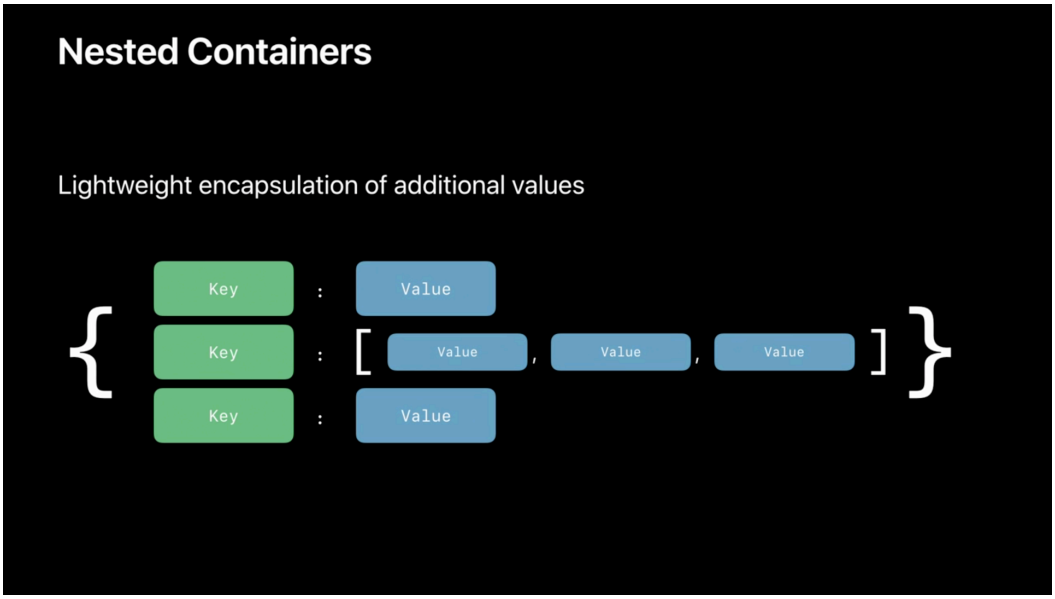
1. 传入 `CodingKeys.self` 构建 `container`
2. 将值编码到相应的 key 里

过程很简单，接下来展示什么时候该使用别的 Container：

```
struct Point2D: Encodable {
    var x: Double
    var y: Double

    public func encode(to encoder: Encoder) throws {
        var container = encoder.unkeyedContainer()
        try container.encode(x)
        try container.encode(y)
    }
}

// 编码后的数据 [ 1.5, 3.9 ]
```

这里是一个平面的坐标，数据简单，那 unkeyedContainer 就是个不错的选择。

Nested Containers



还有就是复合结构容器，支持上面三种基本容器的嵌套。

这样的设计主要是为了支持继承，让继承关系也能编码起来，而且也在每一层都把父类的信息都可以封装得很好。

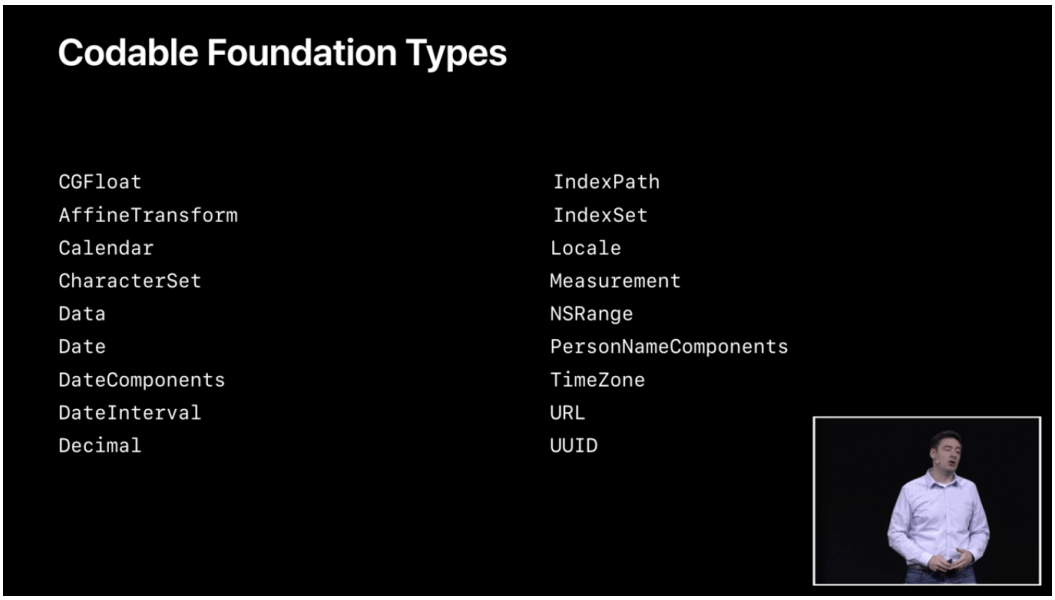
```
class Animal : Codable {
    var legCount : Int
    private enum CodingKeys : String, CodingKey {
        case legCount
    }
    required init(from decoder: Decoder) throws {
        let container = try decoder.container(keyedBy: CodingKeys.self)
        legCount = try container.decode(Int.self, forKey: .legCount)
    }
}

class Dog : Animal {
    var bestFriend : Kid
    private enum CodingKeys : String, CodingKey {
        case bestFriend
    }
    required init(from decoder: Decoder) throws {
        let container = try decoder.container(keyedBy: CodingKeys.self)
        bestFriend = try container.decode(Kid.self, forKey: .bestFriend)

        let superDecoder = try container.superDecoder()
        try super.init(from: superDecoder)
    }
}
```

这里我们有两个类，`Animal` 以及它的子类 `Dog`。这里由于 `CodingKeys` 是 `private` 的，所以父类和子类不会产生冲突。

前面初始化 `bestFriend` 的方法还是跟之前一样，然后只要 `container` 调用方法 `superDecoder` 就能获得父类所需的 decoder，传入 `super.init(from:)` 里就可以完成操作了。

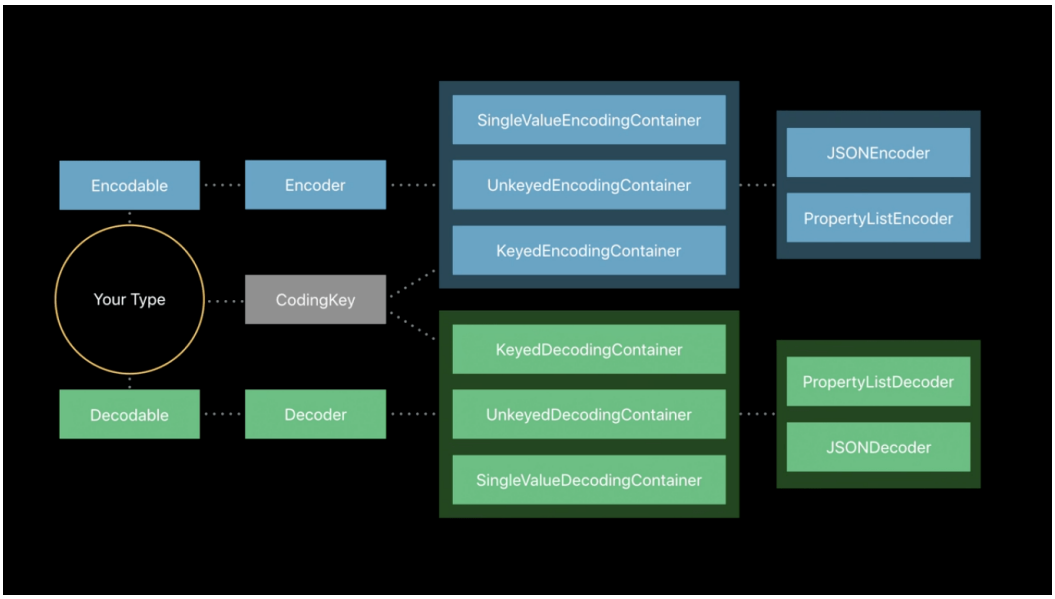
把类型的模样抽象出来

- **复用 Encodable 和 Decodable 的一种实现**。到现在我们发现，其实我们一直用的都是同一套实现，只是每个类型具体的属性不同而已，把类型的样式抽象出来之后，我们就可以一直复用同一套实现了。
- **就算类型发生改变，但生成 Codable 实现的实现并不需要跟着做改变**。我们其实是写了一套实现去生成具体的实现，做到了更加高级的抽象。
- **不同的格式，对应着不同的元类型和解码形式**。我们可以自定义特定类型的编码和解码形式。



受益于这套灵活的方案，Foundation 里这些奇奇怪怪的类型其实都是遵循 `Codable` 协议的。

总结



首先从我们的类型开始，遵循 `Encodable` / `Decodable` 协议，在初始化方法里获取到 `Encoder` / `Decoder`，然后我们就能获取到实际存储数据的 `Container`，而 `KeyedEncodingContainer` / `KeyedDecodingContainer` 再通过 `CodingKey` 插入/获取值，最后 `Container` 再借助 `Encoder` 获取到编码的具体格式。

iOS 11 中网络层的一些变化

本文将介绍 iOS11 下网络层（NSURLSession）的一些变化。这篇文章分4部分来总结了Session 707和709：

1. iOS11中网络层新出的功能
2. iOS11中网络层优化的功能
3. iOS11中网络层的最佳实践
4. 苹果对网络层未来的规划

因为这一次很多内容其实是苹果用新的技术来给网络层做的优化，所以`新功能`和`优化`很难界定。我这边采用的判断标准是：

1. 如果是需要工程师代码配合的优化，就算是`新功能`
2. 如果是不需要工程师代码配合的，哪怕是应用了新技术，我也把它归类为`优化`

---

1. iOS11中网络层新出的功能

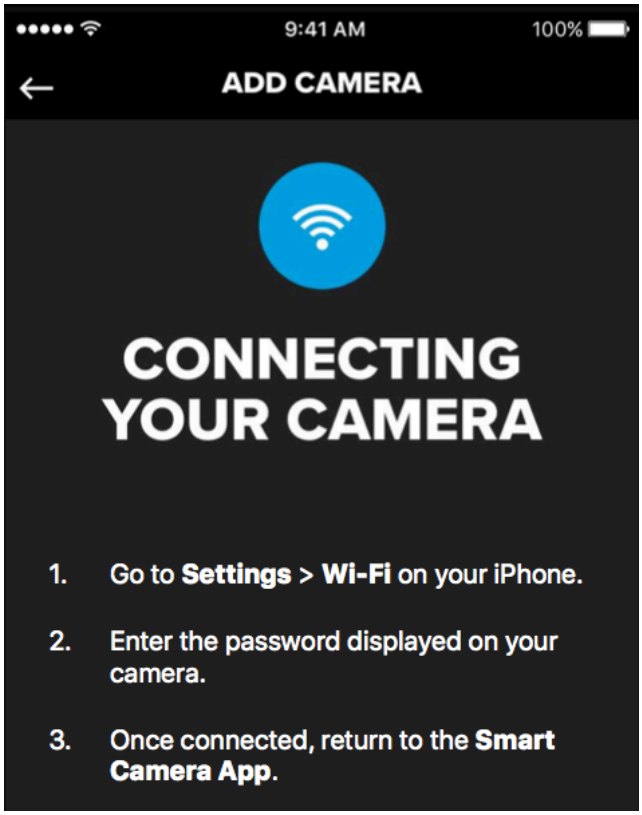
1.1 Network Extension Framework有新API

提供了两个新类：NEHotSpotConfiguration、NEDNSProxyProvider。

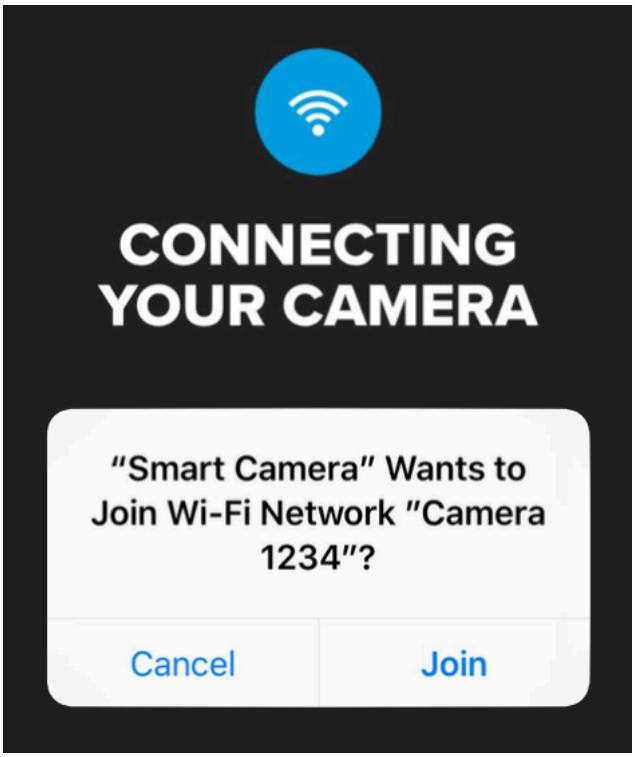
- NEHotSpotConfiguration

NEHotSpotConfiguration可以让你的智能设备在链接手机App之后，能够很方便地通过在手机App上的操作来实现热点的链接。

例如你买了一个网络摄像头，你想要连上摄像头的Wi-Fi热点去配置这个摄像头的话，以前要这么操作：



现在用NEHotSpotConfiguration就能很方便地搞定事情了：



当然，这套API也可以被拿来模拟各种网络环境，在测试App的时候很有用。

- NEDNSProxyProvider

NEDNSProxyProvider可以用来设置你的手机如何跟DNS做交互。你可以自己发DNS请求，也可以自己基于不同的协议去做DNS查询。例如DNS over TLS，DNS over HTTP。

1.2 可以进行多路多协议的网络操作（Multipath Protocols for Mobile Devices）

在之前的iOS版本中，网络如果要从Wi-Fi模式变成Cellular(2/3/4G)模式，那就是先断掉Wi-Fi然后再连上Cellular，然后数据包就从原来的Wi-Fi链路迁移到Cellular链路去发送。

在链路切换过程中链接肯定就断掉了，为了解决这个问题，苹果爸爸搞了Multipath Protocols for Mobile Devices（移动设备多路协议），这使得移动设备的TCP包可以在这两个（多个）链路上随意切换着发（同时开启两个流量链路），而不必断线重连。

效果就是：Wi-Fi和Cellular可以共存，相互辅助。有三个模式，前两个可选，第三个只能自己私底下玩：

- Handover Mode（高可靠模式）

这种模式下优先考虑的是链接的可靠性。只有在Wi-Fi信号不好的时候，流量才会走Cellular。如果Wi-Fi信号好，但是Wi-Fi很慢，这时候也不会切到Cellular链路。

这种模式在beta版中已经支持了。

- Interactive Mode（低延时模式）

这种模式下优先考虑的是链接的低延时。系统会看Wi-Fi快还是Cellular快。如果Cellular比Wi-Fi快，哪怕此时Wi-Fi信号很好，系统也会把流量切到Cellular链路。

这种模式在未来的beta版会支持。

- Aggregation Mode（混合模式）

在这种模式下，Wi-Fi和Cellular会同时起作用。如果Wi-Fi是1G带宽，Cellular也是1G带宽，那么你的设备就能享受2G带宽。

嗯，很好很强大。但你只能拿来玩，不能生产环境中使用。

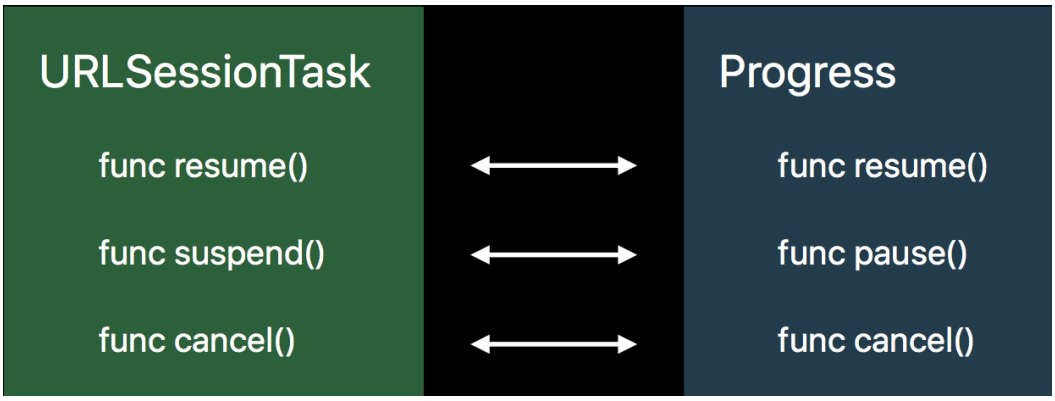
因为苹果爸爸不想让你用。其实他说的理由是希望开发者自己好好考虑1和2用哪个，因为如果3也能用的话，苹果爸爸知道你们就完全不会考虑1和2了，直接用3了。反正用户流量不是开发者掏钱。

这个模式只能够在系统设置里面自己开启来玩，不能像1和2那样可以让开发者在应用中通过NSURLSessionConfiguration.multipathServiceType自己选。

需要注意的是，Multipath Protocols for Mobile Devices这个功能同时也需要服务端支持MPTCP（Multipath TCP）才行，如果服务端不支持的话，光客户端支持没用。linux起了一个项目在做这个事情，项目地址：https://multipath-tcp.org。有兴趣的同学可以自己去看一下。

1.3 ProgressReporting协议

iOS11提供了ProgressReporting协议，并且NSURLSessionTask实现了这个协议，让你能够获得progress对象，这个progress对象可以以0～1.0的方式告诉你当前进度，而不用你自己去拿到已获得的数据量去除以需要获得的数据总量从而得出进度。因为有的时候你并不一定能够拿到数据总量。然后这个progress对象跟NSURLSessionTask的绑定是双向的：你调用progress对象的cancel、pause、resume也会使得task变为cancel、pause、resume，反之亦然。



1.4 URLSessionStreamTask 和 authentication proxies

你需要使用URLSessionStreamTask去替代之前的NSInputStream/NSOutputStream，它支持:

1. 使用host和port来进行TCP/IP连接
2. 支持基于STARTTLS的安全握手协议
3. 支持Navigation of authenticating HTTPS Proxies。事实上就是：如果你是通过proxy访问的网络，当proxy问你要证书的时候，iOS11会自动帮你从keychain里面找到证书给出去。

1.5 URLSession Adaptable Connectivity API

以前进行网络调用时，如果网络不通，那么系统就会报个错告诉你网络不通。这时候你要么轮询要么让用户手动retry，然后网络通了请求才能发送出去。

现在你可以这么做：如果请求发送的时候网络不通，那么这个请求就会等到网络通了的时候再发出去。于是你就不用轮询了，用户也不用手动retry了。

具体做法：把NSURLSessionConfiguration的waitsForConnectivity设置成YES，拿它去生成NSURLSession去使用就好了。

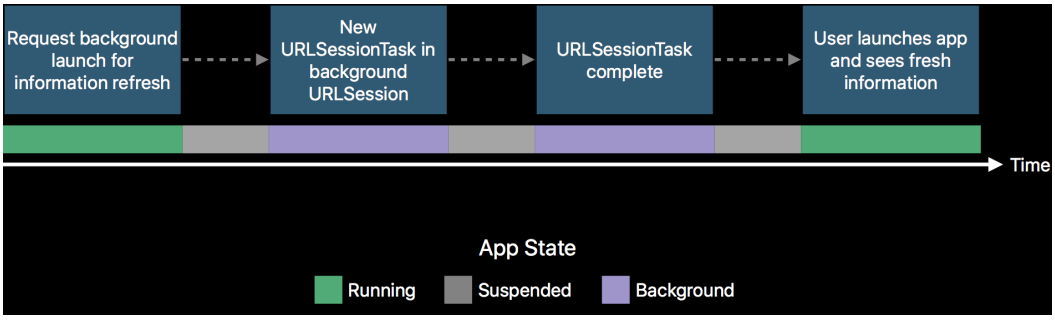
1.6 URLSessionTask Scheduling API

这其实算是一种优化，但还是需要工程师代码配合的，它主要解决了三个问题：

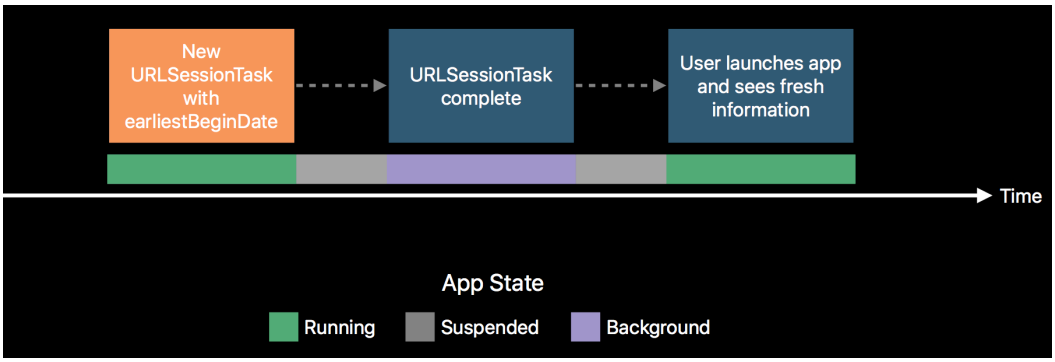
1. 没必要因为创建NSURLSessionTask而进行额外的后台加载
2. 当后台请求创建但还没发出去的时候，这个被创建的请求有可能因为上下文变化的原因导致这

个请求无意义
3. iOS系统并不知道什么时候去发起你的请求才是最合适的

原先你要做后台数据加载的时候，流程是这样的：



现在iOS11里，苹果把头两个步骤合并为一个步骤了：



所以当应用在前台的时候，你就可以创建这个NSURLSessionTask，iOS11里面就不会在后台额外launch一次去创建NSURLSessionTask。这就解决了问题1。

iOS11在NSURLSessionTask的delegate里面提供了一个新的方法：urlSession:task:willBeginDelayedRequest:completionHandler:。系统在发起请求之前会调一个这个回调，然后在这个completionHandler里面你告诉系统这个请求是否要发出去，是否要修改。从而解决了问题2。

iOS11也给了NSURLSessionTask一个property：earliestBeginDate。系统在earlistBeginDate之前是不会发起这个请求的。你给这个task设置一个earliestBeginDate，就解决了问题3。

最后小胖子又补充了一下，你可以通过设置NSURLSessionTask的countOfBytesClientExpectsToSend和countOfBytesClientExpectsToReceive来让系统更好地调度你的后台网络任务。

2. iOS11中网络层优化的功能

2.1 Explicit Congestion Notification（ECN，显式拥塞通知）

先说一下ECN的好处：

1. 可以最大化地使用网络带宽
2. 减少包重发的次数，从而降低延迟，可以提高用户体验

然后说说ECN到底是什么：

有`显式`那就有`隐式`。在以前，TCP/IP发现拥塞的方法是看有没有丢包情况，如果有，那它就认为当前网络有拥塞。于是TCP/IP就会降低发包速率，避免拥塞。这个过程我们可以理解为`隐式拥塞通知`。`显式拥塞通知`就是TCP/IP会收到打上拥塞标记的数据包，TCP/IP发现这个拥塞标记的话，就认为网络出现了拥塞，从而降低数据吞吐量，最终避免恶化网络拥塞现象。

因为通过丢包来发现网络拥塞（隐式拥塞通知）是一件非常消耗成本的事情。接收方在发现丢包之后，需要重新要求发送方来发送之前丢的包，这就额外占用了资源。所以大家就想能不能不用通过丢包的方式来表达网络拥塞的状态，从而让发送端降低发包速率，缓解拥塞。

所以就有了显式拥塞通知。在支持ECN的链路上，如果出现拥塞现象的话，链路不会丢包，而是会在包的header里打上一个标记。接收方拿到这个带着标记的包之后，就认为网络有拥塞现象了，于是就可以降低发包速率，从而缓解网络拥塞。

最后，ECN是需要SQM算法（Smart Queue Management，智能队列管理）支持的。

这个功能在iOS10.3的时候就已经有了。当链路支持显式拥塞通知的时候，iOS10.3上会有50%的链接使用显式拥塞通知来表达网络链路的拥塞状态。从iOS11开始，在支持显式拥塞通知的的链路上，100%的链接都会使用这个技术来表达网络链路的拥塞状态。

2.2 iOS11里的网络操作被移动到User Space去了

原来网络操作都是内核去处理的，现在由每个App各自去处理了。

其实这一段我没有听太懂，说是这么做能够减少更多的上下文切换，从而匀出更多的时间让CPU去处理UI方面的事情，但我感觉好像本质上没什么变化？后来群里AloneMonkey同学说：user到内核会有软中断，涉及到user space的上写文参数到kernel space的映射拷贝，会有比较大的消耗，所以一般会尽量减少这种切换。因此事实上这种优化能够降低不必要的开销，从而提高应用执行的效率。

这么一来的话，以前使用Network Kernel Extension（OS X下）的同学，就不要用了，将来会被deprecate掉。换成Network Extension Framework就好。

2.3 Brotli Compression

iOS11支持Brotli压缩算法（RFC7932）。需要在HTTPS下才能使用，HTTP请求里的Header的Content-Encoding的值是br就表示使用了Brotli压缩算法。这套压缩算法相比gzip的压缩效率提高了15%。苹果浏览器Safari使用了NSURLSession，所以Safari也支持了Brotli。

2.4 Public Suffix List Updates

iOS11更新了Public Suffix List。

补充一下，Public Suffix List能够带来的主要好处有三个：避免超级cookie导致的隐私泄漏、让你的地址栏上的public suffix部分可以高亮、可以更好地对历史URL进行排序。所谓的Public Suffix就是域名的后半部分：.com .co .uk（不完全举例）这些。这个列表告诉了客户端（往往是各大厂商的浏览器）如何去区分域名的边界。

关于Public Suffix List Updates的具体介绍可以看https://publicsuffix.org。

3. iOS11中网络层的最佳实践

3.1 IPv6

苹果爸爸在说：IPv6各种好，大家快来用。如果你不支持IPv6，爸爸就不让你上架。

要支持IPv6的话，老老实实用NSURLSession或者CFNetwork就OK了。

不要做的事情：

1. 不用历史遗留的IPv4 API
2. 不要直接用IPv4的地址做链接，应该用域名去做请求
3. 发包前不要做各种检查：比如你在建立链接之前想看一下我当前这个设备是不是IPv4的地址，这种做法就不行
4. 不要直接使用socket去发起请求

之前我们team也有App上架被拒，原因说的是IPv6不通过。但事实上如果你非常确认以上这些不要做的事情你没做，那么很有可能就是苹果审核人他网络不好，连不到你的服务器（现在还是有一大部分应用的审核在美国）。这类错误苹果都会在邮件中说你不支持IPv6，所以不要被它迷惑了。

3.2 不要引入其他的网络库，要使用苹果自己的API

苹果并不是在说AFNetworking、Alamofire不能用。这些第三方库本质上还是基于NSURLSession，也就是苹果的API去开发的。所以用它们没问题。

苹果的意思是不希望你使用别的基于Socket开发的网络库，例如：ACE、Asio这些。因为苹果的NSURLSession针对自家设备的特点，结合各种网络条件，针对电量、临时 / 后台请求等做了一系列优化。若是你不用NSURLSession去做网络请求，那这些优化就都没了，苹果后续新版本给到的新功能也会用不上了。

3.3 注意timeoutIntervalForResource和timeoutIntervalForRequest的区别

timeoutIntervalForResource是表示数据没有在指定的时间里面加载完，默认值是7天。

timeoutIntervalForRequest是表示在下载过程中，如果某段时间之内一直都没有接收到数据，那么就认为超时。

举个例子就是，如果你要下一个10G的数据，timeoutIntervalForResource设置成7天的话，你的网速特别慢：0.1k/s，7天都没下载完，那就超时了。虽然整个过程中，你一直在源源不断地下载。

如果你要下一个10G的数据，timeoutIntervalForRequest设置为20秒的话，下的过程中有超过20s的时间段并没有数据过来，那么这时候就也算超时。

3.4 一般来说一个App就一个NSURLSession就够了

以前迁移NSURLConnection到NSURLSession的时候，会有人每次都创建新的NSURLSession，但事实上这是没必要的。各个并行的NSURLSessionTask可以共享同一个NSURLSession。真正会使用多个NSURLSession的情况，老头就举了个例子：如果你用safari去开隐私模式的窗口访问网络，那么每个窗口就是一个新的NSURLSession，从而避免数据泄漏。

最后，如果你使用了多个NSURLSession的话，记得清理就好，不清理的话苹果是会产生内存泄漏的。

3.5 NSURLSession的delegate方法和快手block方法不要同时使用

如果你用了block，那么delegate就不会回调了。这事情仅有两个特例是两个都回调的：taskIsWaitingForConnectivity和didReceiveAuthenticateChallenge。

4. 苹果对网络层未来的规划

4.1 TLS1.3

苹果要把网络库整体迁移到支持TLS1.2，年底TLS1.3的标准应该能出来。现在基于TLS1.3草稿的实现可以弄下来自己测试着玩了。

最新的TLS1.3草稿已经出到21了：draft-ietf-tls-tls13-21。

TLS1.3提供了加密的HTTP链接，也就是HTTPS。相对于TLS1.2来说，TLS1.3在安全和执行效率上都有提高。

在安全上，TLS 1.3放弃了很多原来TLS 1.2上的加密算法，毕竟都是上世纪90年代的算法了，现在那些算法安全性已经不高了。例如SHA-1和RC4就已经不用了。

在速度上，TLS 1.3主要是通过减少握手次数来实现速度提升的。TLS 1.2上的两轮握手在TLS 1.3上只需要一轮就可以建立连接了。

4.2 QUIC

Google搞了个QUIC，苹果在跟进。QUIC可以理解成UDP实现的TCP+TLS+HTTP/2集合体。主要是提高了数据传输效率和链接效率:

1. 极大降低了链接建立的时间
2. 增强了拥塞控制机制
3. 无阻塞的多路传输
4. 通过数据冗余传送来实现的错误控制机制（接收端会识别重复数据从而将其筛掉，最终使得错误率尽可能低）
5. 链接迁移（由于QUIC是UDP实现的，因此一个“链接”并不要求一定是端对端，可以几台设备同时处理一个“链接”上的数据）

目前QUIC的开发才刚刚开始，项目网站提供了玩具客户端和玩具服务端给大家玩：Playing with QUIC

总结

我个人比较喜欢的是设备支持了多路多协议的网络操作，这个功能可以极大地提高应用体验。不过也会有用户认为这个功能会导致额外的流量消耗，这就比较矛盾了。

我个人比较讨厌的是不允许delegate和block同时使用的这一项，相信大家也没少被这个事情坑过。因为delegate和block都有各自的适用场景，苹果这么一做，哪怕API调用时的业务场景是一样的，但只要适用场景不同，就也得创建多个NSURLSession了，否则使用起来就很别扭。我可以理解苹果这么做的目的是认为有些事件不适合做两次：delegate一次，block又一次。但这种做法其实并没有必要，应该由使用者来决定在delegate和block都有实现时，应该如何处理：是两个都调、只调block不调delegate、只调delegte不调block。

总的来说这一轮苹果对NSURLSession的改造都很中规中矩，即便是给的新功能，绝大多数也都是偏优化方向。

参考

- Session 707:Advances in Networking, Part 1
- Session 709:Advances in Networking, Part 2

浅谈 iOS 渲染与动画的艺术

本文的主题是渲染和动画。

本文将通过简单的例子来展现 iOS 系统中的各个渲染框架能做什么，适合做什么。之后会对 Core Animation 的图层原理以及一些日常开发中遇到的奇怪的现象背后的原因做一个简单的讲解。看完这一章节，相信读者会对 iOS 的渲染和动画会进一步的了解。

iOS 中的渲染框架

iOS 包含了多套渲染框架。从开发者们接触得最多的`UIKit`、`Core Animation`，到今年 iOS 11 最新推出的`Metal 2`、`Metal for VR`。它们各自的职责大不相同。作为开发者，我们或许并不能各个框架都精通，至少应该知道它们大概是用来做什么的，遇到具体需求的时候，才会有一个大致的方向。

UIKit & AppKit

作为开发者，对 UIKit 和 AppKit 应该是非常熟悉了，实际开发中大量应用到了它们。这不是本章节的重点，因此不再展开。

Core Animation

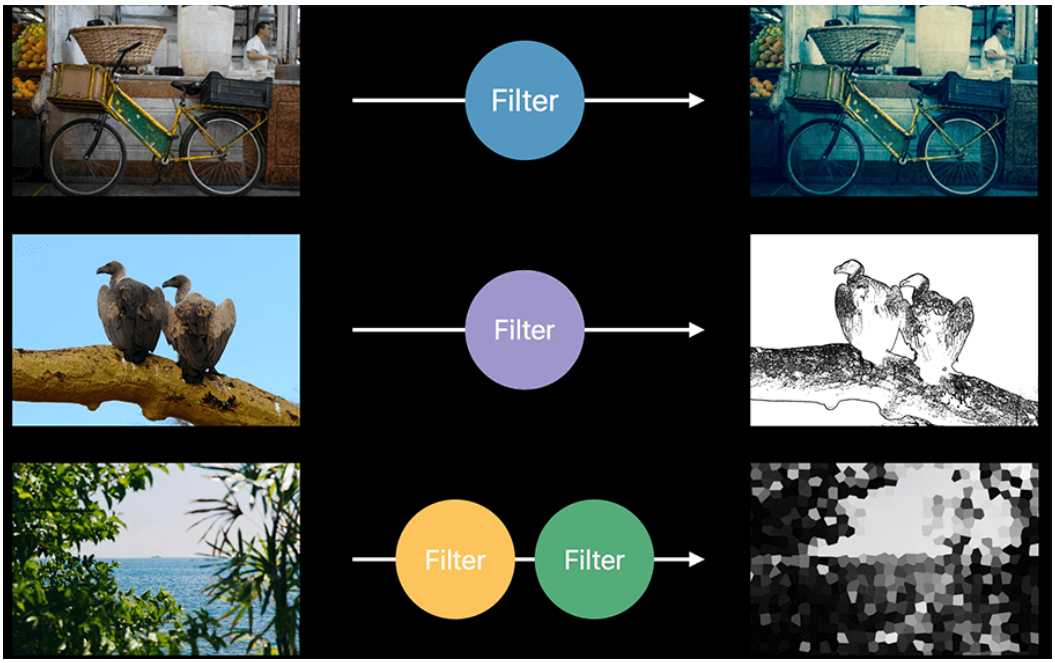
Core Animation 是常用的框架之一。它比 UIKit 和 AppKit 更底层。正如我们所知，`UIView`底下封装了一层`CALayer`树，Core Animation 层是真正的渲染层，我们之所以能在屏幕上看到内容，真正的渲染工作是在 Core Animation 层的。关于`CALayer`的内容，会在下文中展开。

Core Graphics

Core Graphics 也是常用的框架之一，相信开发者们都不陌生。它用于运行时绘制图像。开发者们可以通过 Core Graphics 绘制路径、颜色。当开发者需要在运行时创建图像时，可以使用 Core Graphics 去绘制。与之相对的是运行前创建图像，例如用 Photoshop 提前做好图片素材直接导入应用。相比之下，我们更需要 Core Graphics 去在运行时实时计算、绘制一系列图像帧来实现动画。

Core Image

Core Image 与 Core Graphics 恰恰相反，Core Image 用于在运行时创建图像，而 Core Image 是用来处理已经创建的图像的。Core Image 框架拥有一系列现成的图像过滤器，能对已存在的图像进行高效的处理。



它的优点在于十分高效。大部分情况下，它会在 GPU 中完成工作，但如果 GPU 忙，会使用 CPU 进行处理。如果设备支持 Metal，那么会使用 Metal 处理。这些操作会在底层完成，Apple 的工程师们已经帮助开发者们完成这些操作了。

Core Image 的过滤器支持管道式串联，在下文中会有一个具体的使用 Core Image 框架的例子。

SceneKit & SpriteKit

普通开发者可能会对 SceneKit 和 SpriteKit 感到陌生，SceneKit 主要用于某些 3D 场景需求，而 SpriteKit 更多的用于游戏开发。SceneKit 和 SpriteKit 都包含了粒子系统、物理引擎等，看上去似乎是专门为游戏准备的，但事实上在普通应用中开发者们一样也可以使用它们来完成一些比较炫酷的特效和物理模拟。

Metal

Metal 存在于以上渲染框架的最底层。Metal 对于大多数开发者来说都比较神秘，大多数开发者都没有直接使用过 Metal，但其实所有开发者都在间接地使用 Metal。Core Animation、Core Image、SceneKit、SpriteKit等等渲染框架都是构建于 Metal 之上的。

细心的开发者会发现，当在真机上调试 OpenGL 程序时，控制台会打印出启用 Metal 的日志。根据这点，笔者猜测，Apple 已经实现了一套机制将 OpenGL 命令无缝桥接到 Metal 上，由 Metal 担任真正于硬件交互的工作。

小结

以上是本文中提到的渲染框架。正如上文所说，开发者们并不一定要对这些框架样样精通，真正的目的是让开发者在遇到具体某个需求时能想起来似乎有某个框架是专门做这个任务的，能有个大致的方向。

下面有一些具体的例子能更形象地展现这些框架的功能。

一些例子

这里将有一些简单的 Demo，代码可以从 这里 下载。例子使用 Swift 4 编写，因此需要最新的 Xcode 9 编译。

UIKit Customization

此范例将`UISlider`的滑轨替换成了自定义的图片，效果如下图所示：

自定义图片经过`trackImage`方法，被重新渲染成了目标大小的图片：

```
func trackImage(_ image: UIImage, width: CGFloat, resizingMode:
UIImageResizingMode) -> UIImage {
    let capInsets = UIEdgeInsets(top: 0, left: 8, bottom: 0, right: 8)
    let bounds = CGRect(x: 0, y: 0, width: width, height: image.size.height)
    let renderer = UIGraphicsImageRenderer(bounds: bounds)
    return renderer.image { _ in
        image.resizableImage(withCapInsets: capInsets, resizingMode:
.stretch).draw(in: bounds)
    }.resizableImage(withCapInsets: capInsets, resizingMode: resizingMode)
}
```

之后通过设置`UISlider`的以下方法修改滑轨图片。

```
tintSlider.setMinimumTrackImage(tintMinTrackImage, for: .normal)
tintSlider.setMaximumTrackImage(tintMaxTrackImage, for: .normal)
temperatureSlider.setMinimumTrackImage(temperatureMinTrackImage, for:
.normal)
temperatureSlider.setMaximumTrackImage(temperatureMaxTrackImage, for:
.normal)
```

这是一个抛砖引玉的范例，我们接着看。

Core Image

此范例将通过 Core Image，对图片进行实时滤镜处理。

上文中提到，Core Image 拥有大量线程的图像过滤器用于图像处理，过滤器称为`CIFilter`。但事实上 CIFilter 并不是真正处理任务的对象，而真正处理任务的对象为`CIContext`。CIContext 是底层硬件渲染的入口，因此在使用 Core Image 框架时，必须先要创建它。

```
let ciContext = CIContext()
```

Core Image 框架接收的图像为`CIImage`对象，它可以通过`UIImage`或`CGImage`去构建。

```
let originalCIImage = CIImage(cgImage: originalCGImage)
```

关于 CIFilter，构造方法一定是值得吐槽的一点。它通过接收一个字符串类型的 name 参数和一个参数字典来构造对象，使得它无法脱离文档使用，期待在将来， CIFilter 的构造上能有进一步的改进。

```
public /*not inherited*/ init?(name: String, withInputParameters params:
[String : Any]?)
public /*not inherited*/ init?(name: String)
```

范例使用一个名为`CITemperatureAndTint`的过滤器，其参数接收一个`CIImage`原始图像和两个`CIVector`二维向量参数，它的两个维度是色温和色彩。通过界面上的滚动条，用户可以任意更改其中一个向量，以达到调整色彩的作用。

```
let neutralTemperature = CGFloat(temperatureSlider.value)
let neutralTint = CGFloat(tintSlider.value)
let neutral = CIVector(x: neutralTemperature, y: neutralTint)

let targetNeutralTemperature = CGFloat(6500)
let targetNeutralTint = CGFloat(0)
let targetNeutral = CIVector(x: targetNeutralTemperature, y:
targetNeutralTint)

let parameters = [ "inputImage": originalCIImage,
                   "inputNeutral": neutral,
                   "inputTargetNeutral": targetNeutral ]

let filter = CIFilter(name: "CITemperatureAndTint", withInputParameters:
parameters)
```

准备好了这一切以后，就可以进行真正的图像处理了：

```
let filteredImage = filter.outputImage
// CIContext 创建 CGImage 的过程会比较耗时，可以在放在子线程执行。
let filteredCGImage = self.ciContext.createCGImage(filteredImage, from:
filteredImage.extent)
self.imageView.image = UIImage(cgImage: filteredCGImage)
```

以上就是 Core Image 的图像滤镜处理流程。

由于图像处理是一项耗时的操作，如果你想你的应用对图像操作响应如飞，请参考接下来的`Core Graphics`。

Core Graphics

此范例将在运行时绘制一个张图像。

在去年 iOS 10 发布时，`UIGraphicsImageRenderer`诞生了，它能帮助我们更方便地使用绘图 API。

```
let renderer = UIGraphicsImageRenderer(size: size)
```

UIGraphicsImageRenderer 的 image 方法接收一个闭包，传入参数为`UIGraphicsImageRendererContext`。这就意味着， renderer 已经帮我们管理好了绘图上下文，我们只需在这个闭包内写入绘图内容即可。

```
let image = renderer.image { rendererContext in
    let cgContext = rendererContext.cgContext
    // 绘图
}
```

Core Graphics 的绘图十分形象。你可以想象手中有一根画笔，绘图时，调用`CGContext`的`move`方法将画笔移动到一个点，之后调用`addLine`方法画一条线到另一个点，就画好了一条线了。

```
public func move(to point: CGPoint)
public func addLine(to point: CGPoint)
```

对于范例中的五角星，我们只要画出封闭的五角星外轮廓，之后填充黄色即可。

```
let image = renderer.image { rendererContext in
    let cgContext = rendererContext.cgContext
    // 绘制轮廓
    ......
    //设置填充颜色
    cgContext.setFillColor(fillColor.cgColor)
    //填充
    cgContext.fillPath()
}
```

至此就是 Core Graphics 绘制图片的过程。

Core Animation

Core Animation 是本文的重点。在这里只是一个简单的例子，在下文中会有更深层的剖析和一个更加精彩的例子。

此范例将实现一个基于 Core Animation 的动画。点击中间的按钮，将会有一圈星星散开，逐渐变小，最后消失。

正如图中所示，点击按钮后会出现很多星星。其实每一个星星都是一个`CALayer`，通过控制每一个星星的动画，即可实现整体的效果。

获得其中的一个五角星图像：

```
let layer = CALayer()
// 设置五角星，通过上面例子中的 Core Graphics 绘制五角星。
layer.contents = ...
// 设置位置和大小
layer.position = ...
layer.bounds = ...
// 加入屏幕
button.layer.addSublayer(layer)
```

获得了目标 Layer 后，设置动画的最终变换状态：

```
// 设置终点位置
let finalPosition = ...
// 设置终点的平移量
var finalTransform = CATransform3DMakeTranslation(finalPosition.x -
initialPosition.x, finalPosition.y - initialPosition.y, 0)
// 设置旋转，这里的 spinRadians 值是4π，转两圈。但由于转两圈后的状态和不转的状态是一模一样的，因此这里不会有任何旋转。
finalTransform = CATransform3DRotate(finalTransform, spinRadians, 0, 0, 1)
// 设置缩放
finalTransform = CATransform3DScale(finalTransform, finalScale, finalScale,
finalScale)
```

完成了以上工作后，使用`CABasicAnimation`创建动画。

```
let animation = CABasicAnimation(keyPath: "transform")
// 当前变换状态
animation.fromValue = layer.transform
// 目标变换状态
animation.toValue = finalTransform
// 动画时长
animation.duration = animationDuration
// 动画时间函数，刚开始慢，最后变快。
animation.timingFunction = CAMediaTimingFunction(name:
kCAMediaTimingFunctionEaseOut)
layer.add(animation, forKey: "pew pew")
// 将变换设置为最终的状态，以防动画结束后回到原点。
layer.transform = finalTransform

// 在动画结束后移除Layer
DispatchQueue.main.asyncAfter(deadline: DispatchTime.now() +
animationDuration, execute: {
    layer.removeFromSuperlayer()
})
```

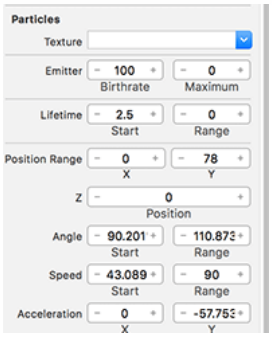
这样，一个五角星的动画就设定好了。关于例子中防止状态回弹的代码，读者不妨思考一下，为什么要这样写？五角星已经移动过去了，为何状态会回弹？五角星是真的移动过去了还是看上去移动过去了？这些问题与 Core Animation 的图层结构有关，将在下文中探讨。

SpriteKit

此范例实现了一个简单的粒子效果。

SpriteKit 是一款游戏引擎，但并不意味着只有游戏才能用，普通应用中一样也可以使用它来实现特定的效果。

场景是 SpriteKit 的一个大单位，称为`SKScene`。而粒子节点`SKEmitterNode`是实现粒子效果的载体，在 Xcode 中，粒子效果可以可视化定制，生成 sks 文件供程序调用。



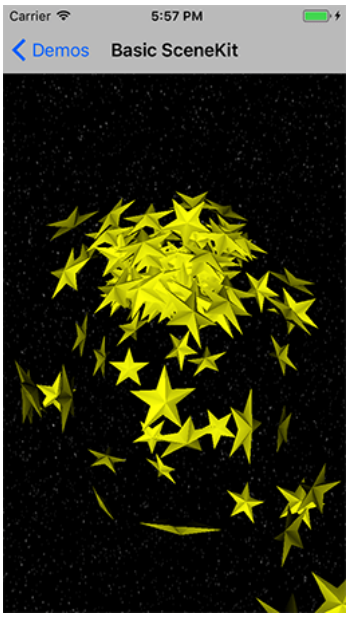
图中为 Xcode 自带的粒子效果工具，通过调整这些参数，开发者能实时预览粒子的变化。最终生成 sks 文件，就可以在程序中直接加载了。在设计粒子系统上，开发者并不需要写任何一句代码，就可以完成。

```
// 创建场景
scene = SKScene(size: UIScreen.main.bounds.size)
sceneView.presentScene(scene)
// 从sks文件加载粒子效果
emitter = SKEmitterNode(fileNamed: "star-particle-effect.sks")
// 生成粒子的贴图
emitter.particleTexture = SKTexture(image: starImage)
// 设置粒子的喷射点
emitter.position = CGPoint(x: scene.size.width / 2, y: scene.size.height / 2)
// 将节点加入场景，粒子系统会自动开始工作
sceneView.scene?.addChild(emitter)
```

对于这样的一个简单的粒子系统，只需要上面的几行代码即可实现。

SceneKit

此范例实现了一个三维粒子和物理模拟的粒子系统，粒子喷出的同时会因为重力旋转、掉落。



SceneKit 也是一款游戏引擎，也有场景的概念，称为`SCNScene`。不同的是，SceneKit 带来的是一个三维的场景。因此引入了摄像机`SCNCamera`的概念。通过设定摄像机不同的位置和不同观察角度，可以观察到三维世界里不同的现象。不过范例中并不需要操作摄像机。而这次的粒子并不是粒子系统自动创建的，而是通过手动生成`SCNNode`，赋予一定的初始速度，通过物理模拟喷射而出的。

在 SceneKit 中，视角变换、材质和光影只需要设定相应的值，具体的渲染过程已经不需要开发者去操心了。

设定摄像机：

```
// 创建场景
scene = SCNScene()
// 创建摄像机
let camera = SCNCamera()
let cameraNode = SCNNode()
cameraNode.camera = camera
cameraNode.position = SCNVector3(x: 0, y: 0, z: 10)
scene.rootNode.addChildNode(cameraNode)

// 加载物体资源
let asset = MDLAsset(url: Bundle.main.url(forResource: "star",
withExtension:"obj")!)
// 从资源加载物体几何体
if let mesh = asset[0] as? MDLMesh {
    starGeometry = SCNGeometry(mdlMesh: mesh)
}

// 创建材质
let material = SCNMaterial()
material.diffuse.contents = UIColor.yellow
starGeometry.materials = [material]

// 创建光源
let lightNode = SCNNode()
lightNode.light = SCNLight()
lightNode.light?.type = .directional
lightNode.position = SCNVector3(x: 10, y: 10, z: 10)
scene.rootNode.addChildNode(lightNode)
```

设定完场景的信息以后，视角变换、材质、光影渲染会由引擎完成，而开发者只需要致力于业务开发即可。

```
// 定时每秒喷射20个星星
starTimer = Timer.scheduledTimer(withTimeInterval: 0.05, repeats: true) { _
in
    // 从几何体生成星星节点
    let starNode = SCNNode(geometry: self.starGeometry)
    self.scene.rootNode.addChildNode(starNode)
    // 设置物理行为
    starNode.physicsBody = SCNPhysicsBody(type: .dynamic, shape:
SCNPhysicsShape(geometry: self.starGeometry, options: nil))
    // 设置初始喷射速度
    starNode.physicsBody?.velocity = SCNVector3(5 * (drand48() - 0.5), 5 + 2
* drand48(), 5 * (drand48() - 0.5))
    // 定时移除节点
    DispatchQueue.main.asyncAfter(deadline: DispatchTime.now() + 5) {
        starNode.removeFromParentNode()
    }
}
```

使用定时器定时喷射节点，而重力下落、旋转等也已经由引擎接管。至此，一个三维的粒子系统就已经完成了。

小结

通过几个简单的例子，向读者们展示了 iOS 系统中各个渲染框架的用途，相信大家对这些渲染框架已经有了基本的概念。接下来会针对日常开发中最常用的动画框架 Core Animation 做一个更为深入的讲解。

Core Animation

本章节将会对 Core Animation 的图层原理和一些日常开发中需要注意的地方做讲解。

图层原理

动画由一系列连续的帧组成，开发者需要手动编写每一帧的 UI 来实现动画。听上去很傻，我们从来都没那么做过。因为 Core Animation 已经为我们做了一部分工作，开发者只需要设定一些关键帧，就可以实现动画。

```
UIView.animate(withDuration: 0.25, animations: {
    self.view.frame.origin.y = 100;
})
```

这是一个简单的移动动画，在设定`self.view`会在 0.25 秒的时间内向下移动 100 个单位。当这段代码提交以后，view 会匀速向下移动，看上去`self.view.frame.origin.y`是匀速增加的，如果在动画结束以前去查询坐标的`y`值，从表面上来看，动画还在进行过程中，因此`y`值应该处于 0 ~ 100 中的某个值，但事实上会发现值已经是 100 了。属性在提交的那一刻就已经发生了变化，但为什么界面上没有立刻跳到目标帧？这个动画到底是如何实现的呢？

其实用户看到的动画层，并不是开发者所修改的那个层。这里涉及到两个层，模型层(Model)和展现层(Presentation)。

在提交动画的那一刻，展现层会对模型层做拷贝，覆盖在模型层上面。提交动画时模型层的值会立刻被修改，而展现层还是原来的样子。在动画进行过程中去查询属性，查询到的是模型层里的属性，它已经在提交动画时被修改了。而用户看到的动画，是 Core Animation 每一帧所计算的展现层。因此会出现查询到的属性和界面现象不符的情况。

当动画结束时，展现层就不复存在的，此时用户将再次看到模型层的界面。这也就解释了为什么`CAAnimation`在动画结束时画面会立刻跳回到原始画面，因为`CAAnimation`只提交动画，Core Animation 只是计算了一堆展现层覆盖在上面来展示动画并不修改模型层的属性值。对于这个问题，网上很多人提出了这样的解决方法：

```
animation.isRemovedOnCompletion = false
animation.fillMode = kCAFillModeForwards
```

它阻止了动画结束时展现层的移除，使得表面上看上去貌似是解决了问题，但模型层的属性仍然是原来的值。这会造成什么问题？很多“奇怪”的现象就出现了。例如移动了一个按钮，展现层上按钮是移动了，但按钮的模型层还在老地方，用户所看到的按钮是永远也点不中的。

这个问题正确的解决方案是什么？非常简单，在提交动画后，直接修改属性值。Core Animation 仍然会计算展现层覆盖在模型层之上，使用户看到的还是正常的动画。模型层实际上已经在目标位置了。动画结束时，展现层被移除，此时用户将看到已经在目标位置的模型层，整个动画过程也就结束了。

Frame & Transform

虽然`frame`和`bounds`属性在日常开发中用的很多，但不了解它们是如何计算的话，可能会遇上一些麻烦。例如以下一些列操作以后，会复现一个奇怪的现象。

- 设置了 layer 的 frame。

```
layer.frame = CGRect(x: 0, y: 0, width: 200, height: 100)
```

- 对 layer 进行缩放，缩小到原来的四分之一。

```
layer.transform = CGAffineTransform(scaleX: 0.25, y: 0.25)
```

- 重新设置 frame，layer 回到了之前的大小。

```
layer.frame = CGRect(x: 0, y: 0, width: 200, height: 100)
```

- 设置旋转角度，layer 突然放大了四倍，变成了 800 x 400。

```
layer.transform = CGAffineTransform(rotationAngle: 0.2)
```

为什么会出现这种现象?

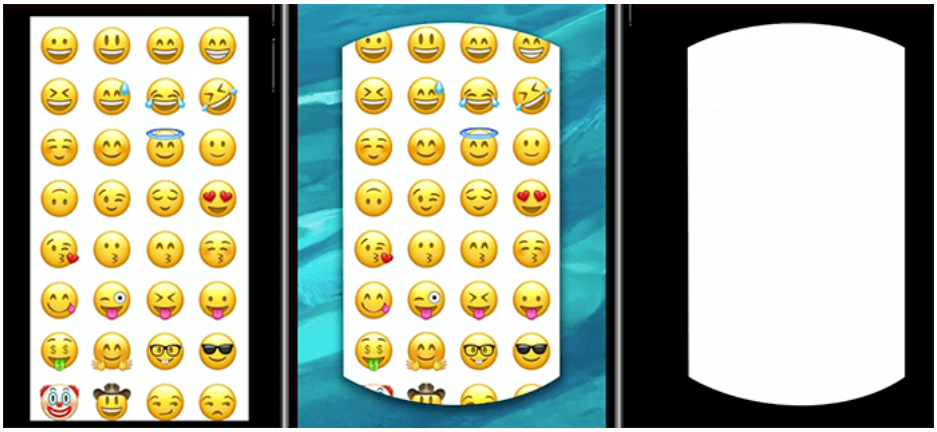
原因在于，`frame`是 layer 在外部的大小，`bounds`而是 layer 内部的大小。当 frame 被设置时，bounds 和 transform 会被作为依据进行计算。换句话说，在上面步骤中的第三步，重新设置 frame 时，transform 属性仍然是缩小四倍的效果，因此此时 frame 会变成目标大小的四倍。到此为止，在屏幕上看到的仍然是正确的大小，但在第四步，重新设置 transform 属性时，缩小四倍效果被替换成了旋转，因此此时 layer 会突然变成四倍大小。

因此，在使用 transform 属性的同时设置 frame 时，要格外小心，避免这类奇怪的现象的出现。

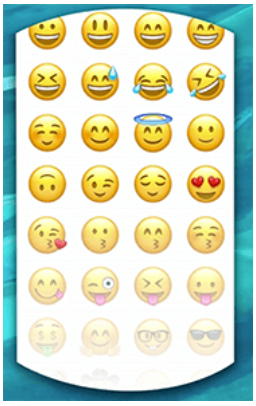
Mask & Shadow

Mask 和 Shadow 的不恰当使用，会带来严重的性能问题。

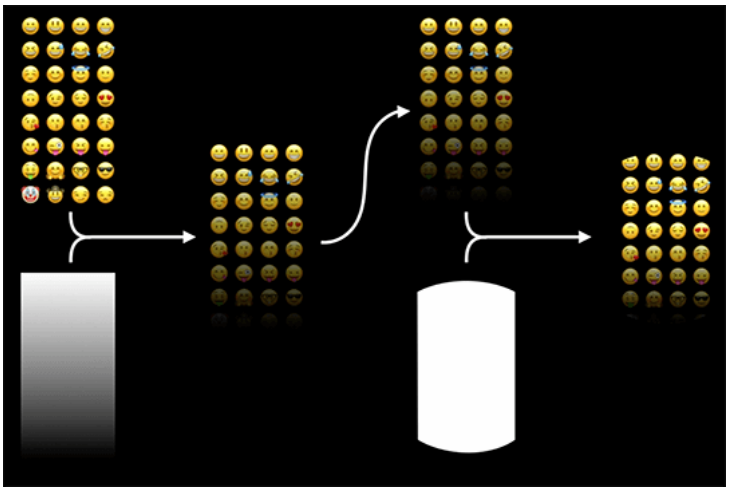
先来说说 Mask。Mask 即遮罩，这是一个常用的裁剪异形图像的方法。通过一个形状或图像，对另一个形状或图像做遮罩，取出交集或其他感兴趣的部分，例如下图中通过右图对左图进行裁剪，得到中图。



如果要得到下图的效果，你可能会想到使用两次遮罩。



如下图所示，使用半透明遮罩得到中图后，再使用裁剪遮罩得到最终结果。



这么做虽然可行，但是会带来严重的性能问题。每一帧的渲染需要重复多次离屏渲染，多次开辟新的空间，拷贝图像内存，是非常大的开销。因此这类情况下，在裁剪前覆盖一张半透明图片会比重复遮罩要快的多，重复遮罩并不是明智的选择。

再来说说 Shadow。

在绘制阴影以前，系统并不知道这个 layer 的轮廓是怎样的。因此需要将整个 layer 树都完整的绘制出来以后才能去绘制阴影。同样是上面图里的例子，这个 layer 包含了非常多的 emoji 表情，相当于包含了很多子 layer。默认情况下在绘制阴影以前，需要将每一个表情都绘制出来，然后再计算阴影。

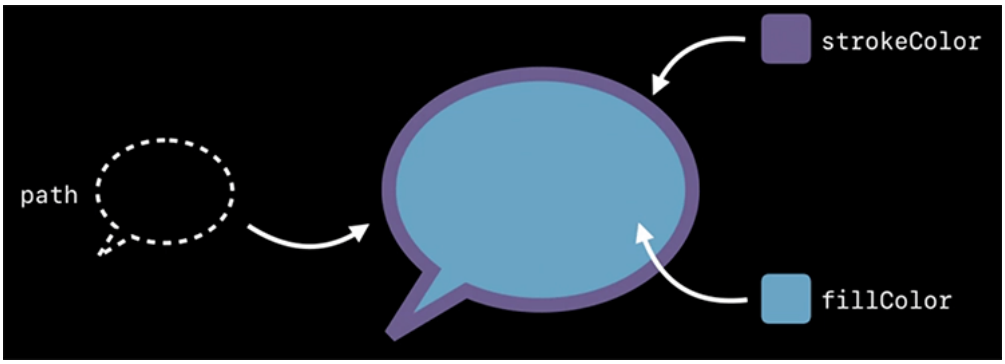
事实上那些 emoji 表情都在根 layer 内部，是不会对阴影造成任何影响的，因此绘制那些 layer 是没有必要的，传统的阴影绘制模型在这种情况下效率会非常低。解决方法是使用 Core Graphics 绘制阴影 path 进行阴影计算，这样做会非常高效，避免不必要的绘制。

Tips & Tricks

这里是几个可能被大家忽视却非常有用的东西。

CAShapeLayer

`CAShapeLayer`是 CALayer 的一个子类，它使用 Core Graphics Path 绘制 layer。



有意思的是，修改它的属性时是会有过渡动画的。以及它`strokeStart`和`strokeEnd`属性可以让开发者去定义它从哪里开始绘制，到哪里结束。还可以通过`lineDashPattern`和`lineDashPhase`属性设置边线样式。

CAShapeLayer 有非常多跟线条、路径有关的属性，可以用来完成许多有趣的效果。

CAGradientLayer

`CAGradientLayer`是 CALayer 的一个子类，用于绘制渐变的图层。

同样的，修改它的属性，也会产生过渡动画。它的`startPoint`和`endPoint`属性可以自定义绘制过程，产生类似金属高光反射的效果，如上图所示。

Layer Speed

Layer Speed 指的是 layer 在执行动画的速度。

在动画过程中，每个 layer 都使用自己的 layer speed。这就意味着，一个layer 动画，它的子 layer 可以使用不同的动画速度。且当速度为 0 时，此 layer 的动画将会暂停。这是一个非常强大的功能，使得开发者能完全控制动画进程。意味着 layer 动画，是可以打断，可以暂停的，甚至可以是回溯的。

这里有一个可回溯的动画例子：https://github.com/enums/LayerAnimationDemo

在界面任意位置上下拖动时，界面上的 layer 会慢慢展开，往回拖动则可以回溯到动画的任意时刻。查看代码会发现，其实际上是一个`CABaseAnimation`，但在视图加载时，layer 的速度被设置为了 0，此时动画停止：

```
containerLayer.speed = 0
```

在响应手势时设置 layer 的`timeOffset`属性使动画回溯到任意时间点。

```
containerView?.layer.timeOffset = currentContainerExpansion
```

通过控制 layer 的`speed`和`timeOffset`，实现动画进程的控制。但需要注意的是，这里的动画只显示在展现层，并不更改模型层的参数。在 iOS 的事件响应链中，碰撞测试是以模型层参数为准的，因此任何事件仍然会按照模型层的位置传递。如果给一个 UIButton 添加了类似动画，点击屏幕所显示的按钮的位置时未必能够命中按钮。在实际使用个过程中需要注意这点。

小结

通过几个简单的例子，介绍了 Core Animation 的一部分原理和日常使用过程中需要注意的点。图层原理可帮助读者更好的理解过渡动画的原理，避免一些由于对图层原理不熟悉而可能发生的麻烦。几个例子指出了日常开发中可能犯的错误，因此可能会造成界面混乱以及性能问题，在日常开发中需要多加注意。最后的 tips & Tricks 简单介绍了两个不常用的工具 layer，以及使用 layer speed 实现的可回溯动画。

结语

作为客户端开发者，用户体验是永恒的话题。上文所提到的渲染框架，是 Appler 工程师们给开发者们的馈赠。高效的 Core Graphics 和 Core Animation，炫酷的 SpriteKit，SceneKit。可以帮助开发者们快速开发动画和特效。

动画和渲染的学问远不止这些，本文的目的也是拓宽开发者们的视野，将渲染框架一一摆出来展示，让大家在遇到具体需求时能更合适的选择框架去做，在制作粒子特效时尽可能的选择 SpriteKit 和 SceneKit 去做，而不是一味地创建大量 UIView 去实现。看到这里，相信读者们都能够选择正确的渲染框架去做正确的事情了。

参考

- Building Visually Rich User Experiences - Video
- Building Visually Rich User Experiences - SampleCode
- Layer Speed Animation Demo

让你的 UI 适配 iOS 11 吧

> 注意：本篇涉及的内容基本都是 iOS 11 才有的新特性，所以使用的时候一定要记得做版本判断，当然这只是对于使用 Objective-C 开发的应用，因为 Swift 在编译期间就会各种提醒你。

各种 Bar 的新特性

`UINavigationBar`

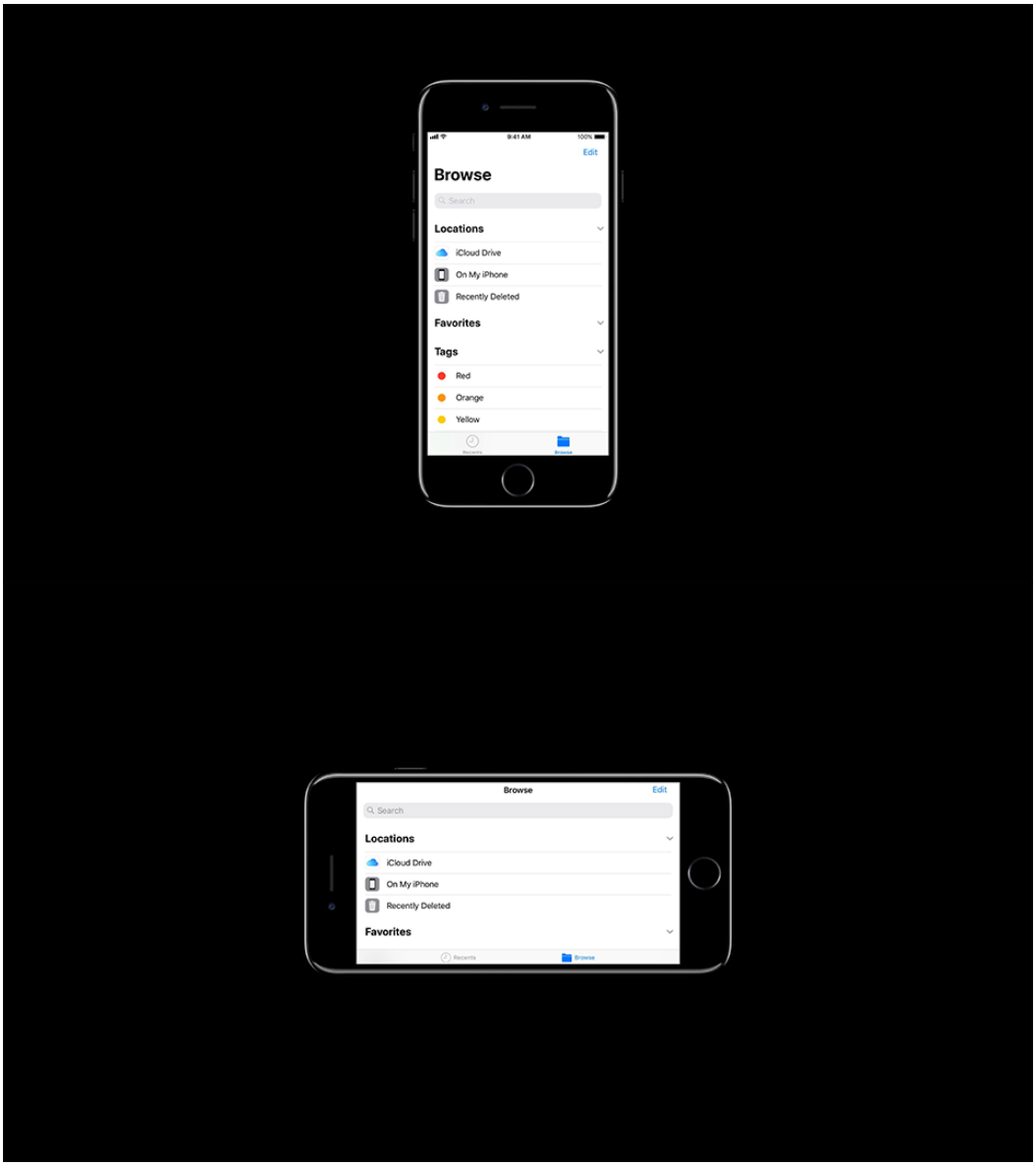
新特性

大标题模式

WWDC17 苹果发布了 iOS 11 系统，这一次，苹果在 UI 上又做了大调整，其中就包含了大标题。

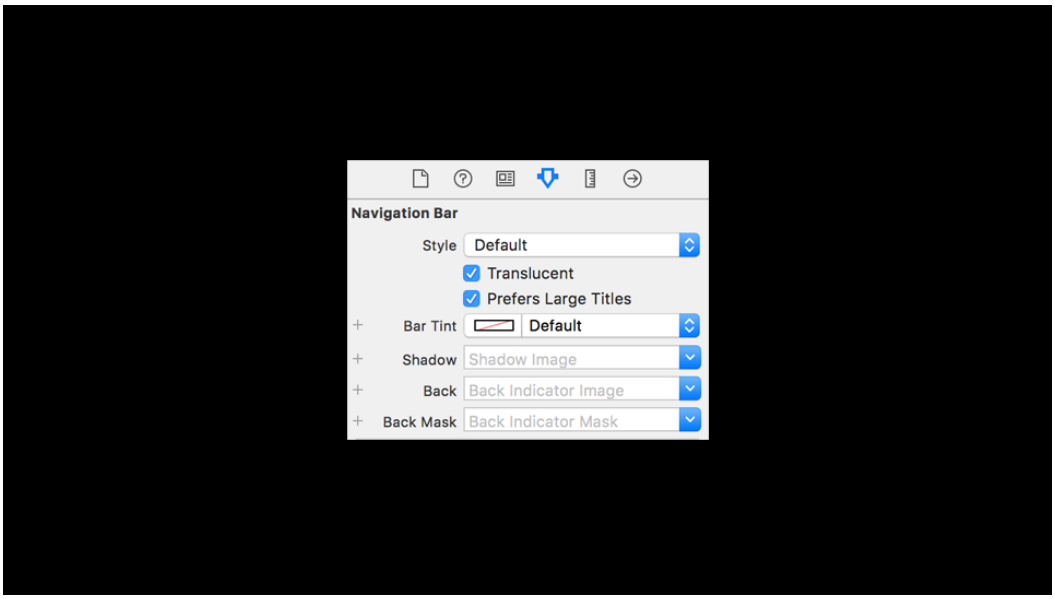
下面是来自官方的两张截图，这里有两个点需要注意：

- 截图中 `UITableView` 中的文本也比 iOS 10 的要大，但这并不是因为 `UINavigationBar` 的大标题模式而变大的，而是苹果根据大标题模式给出的 UI 设计规范。
- 横屏模式下并不会出现大标题，主要考虑横屏模式下垂直方向的显示区域太小。



当然，苹果也把这个特性开放给了开发者，所以现在我们就来看一下在 iOS 11 上如何开启这个效果，其实很简单，将 `UINavigationBar` 的 `prefersLargeTitles` 属性设置为 `true` 即可开启导航栏的大标题模式：

当然，也可以在 StoryBoard 中设置，开启方式如下图：



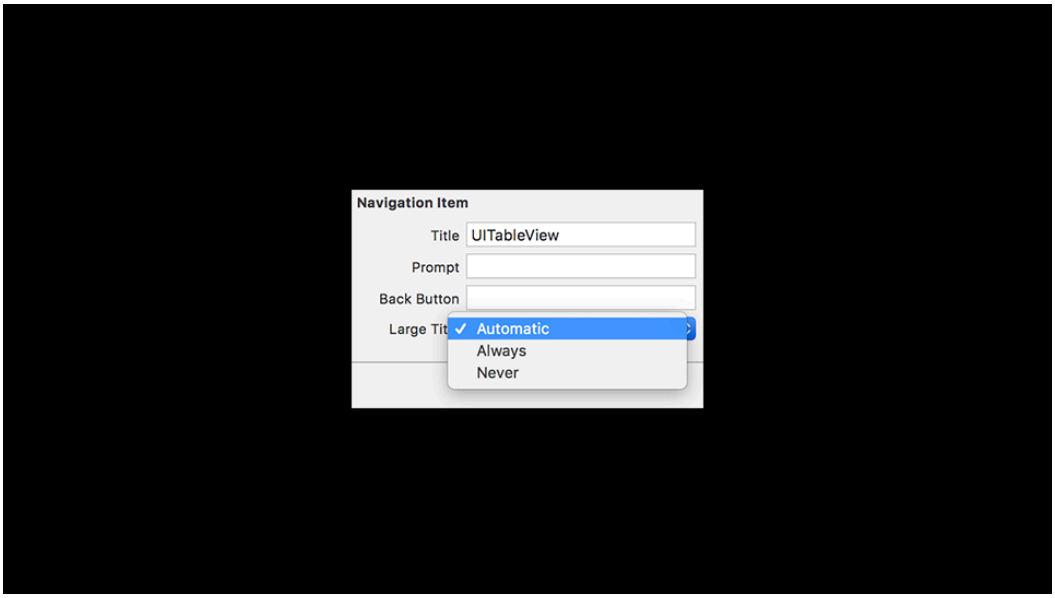
> 注意：因为每个 `UINavigationController` 只有一个 `UINavigationBar`，这也就意味着，如果将其 `UINavigationBar` 的 `prefersLargeTitles` 属性设置为 `true`，那么属于这个 `UINavigationController` 下的所有 `UIViewController` 都显示大标题。

既然 `prefersLargeTitles` 是控制某个 `UINavigationController` 下的所有 `UIViewController` 是否显示大标题的属性，苹果当然也会提供单独控制的属性，也就是 `UINavigationItem` 上的 `largeTitleDisplayMode`，我们知道，每个 `UIViewController` 都有独立的 `UINavigationItem`，这样我们就可以控制单个 `UIViewController` 的大标题模式了。

这个属性包含三种模式：

- .automatic：默认值。表示不做任何操作，保持 `UINavigationBar` 中上一个 `UINavigationItem` 的模式。
- .always：使用大标题。当我们进入这个页面时，显示的是大标题。
- .never：不使用大标题模式。当我们进入这个页面时，显示的是原来的小标题。

同样，这个属性也可以在 Storyboard 中设置：



在 `UINavigationItem` 上添加 `UISearchController`

大家先在脑海中过一下在 iOS 11 之前我们是怎么实现这个功能的。

是不是已经不想过了……好，我们来看看 iOS 11：

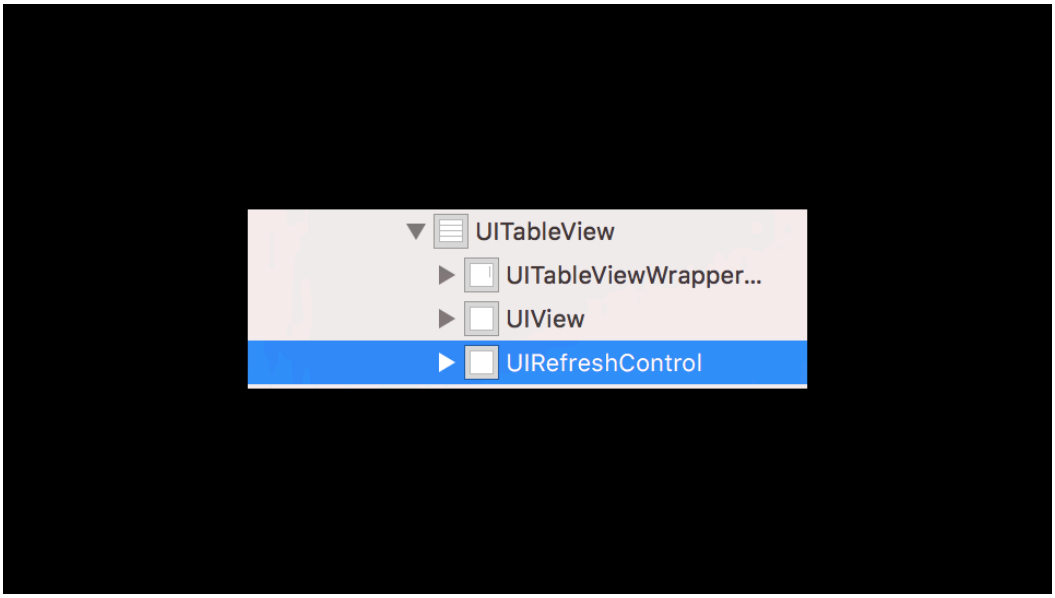
```
// 构建一个 UISearchController 对象
lazy var searchController = UISearchController()

// 设置

navigationItem.searchController = searchController
```

So easy！！！

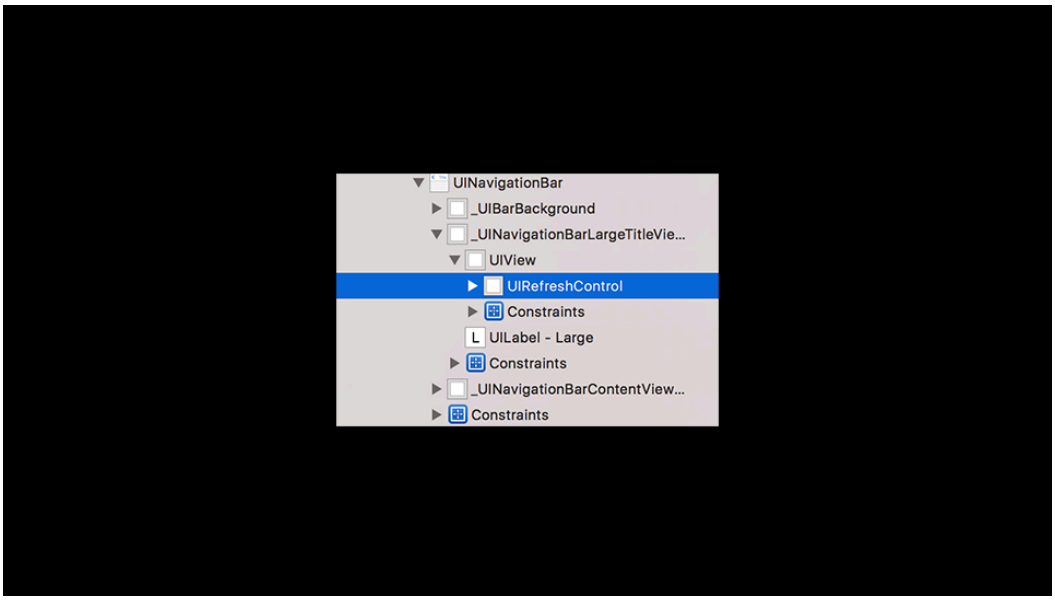
另外，iOS 11 的 `UINavigationItem` 还提供了 `hidesSearchBarWhenScrolling` 属性，设置为 `true` 就可以在滑动的时候隐藏 `UISearchBar`。

`UIRefreshControl` 的变化

在 iOS 11 之前，`UIRefreshControl` 会被添加到相应的控件上，如 `UITableView`：



而 iOS 11 之后，`UIRefreshControl` 被直接添加到了 `UINavigationBar` 上，如下图：



Rubber bending 效果

首先我们了解一下什么是 Rubber bending，通俗来说：就是当你在 `UIScrollView` 上下拉，然后放开时的回弹效果。

iOS 11 之后，因为大量的内容都被添加到 `UINavigationBar` 上，
如：`UIRefreshControl`、`UISearchController` 和大标题等，所以苹果在 `UINavigationBar` 上也引入了 Rubber banding 效果。

旧版本的兼容

既然 iOS 11 已经有了这些新特性，我们是否可以考虑在 iOS 11 之前也支持它们呢？通过实验，笔者分析出了几个关键变化点：

- 正常的小标题被隐藏了。
- 导航栏的高度由 64 变为 96。
- 大标题的标题字体为 34 。
- `UIRefreshControl` 和 `UISearchController` 都被添加到了 `UINavigationBar` 上。
- 当 `UINavigationBar` 下方的 View 为 `UIScrollView` 等可以滚动的控件时，如果对控件做下拉操作， `UINavigationBar` 上的大标题也会随之下拉，过程中文字海会又一个稍微放大的效果，当然还会有 `UIScrollView` 的 Rubber bending 效果。

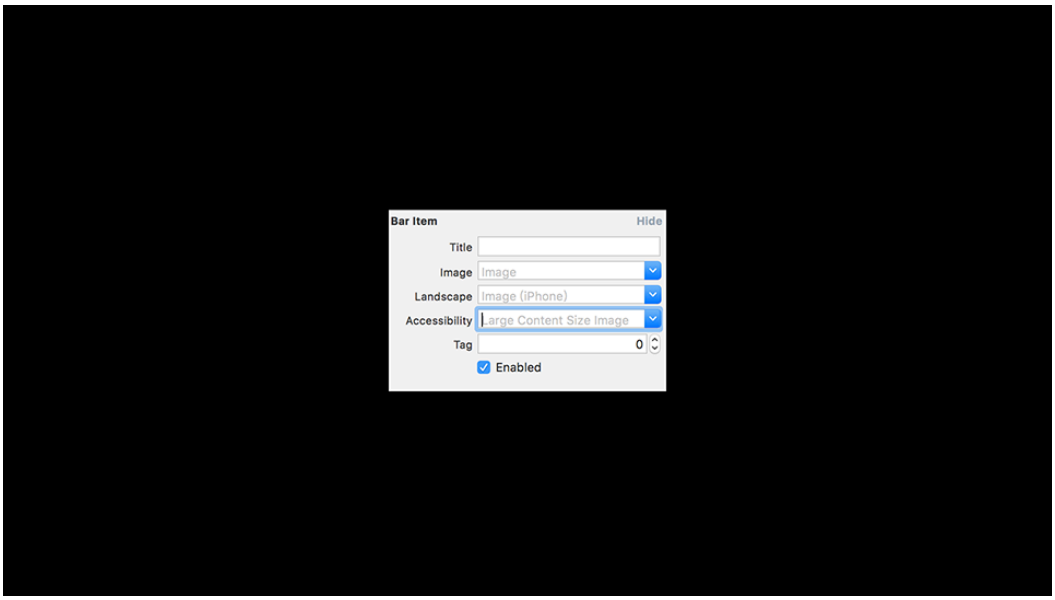
也就是说我们只要在 UI 上做这几点改动即可。另外考虑到这些特性是 iOS 11 特有的，我们可以考虑用 **Extension** 的方式扩展出这些特性。

`UITabBar` 的横屏模式

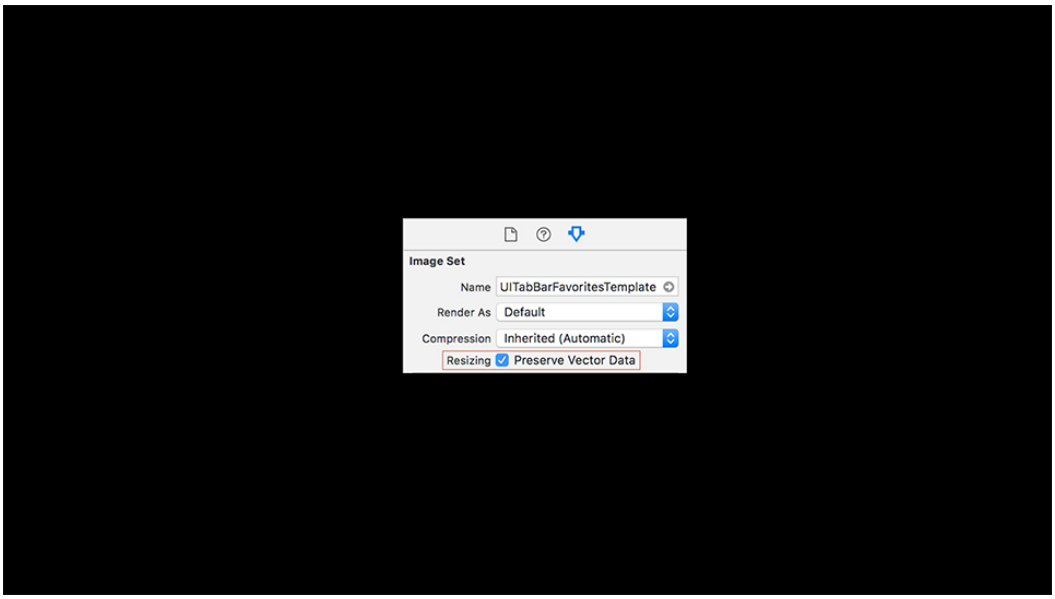
在 Landscape 模式和 iPad 下， `UITabBar` 也发生了一些改变，图标和标题是水平排列的，而且在 iPhone 上的图标还会更小，从而节省了 Landscape 模式下垂直方向的空间。同时我们还可以通过 `landscapeImagePhone` 这个属性来设置 iPhone Landscape 模式下 `UIBarItem` 的图标。

另外，如果设置了 `UIBarItem` 的 `largeContentSizeImage`，当开启**大字体**模式并长按这个 `UIBarItem` 时，会在屏幕中央出现它图标和标题的放大版。如下图：

当然这两个属性也可以在 Storyboard 中设置，如下图：



最后，如果你设置的图片是 .xcasset 文件中的 PDF 文件，可以直接勾选 Preserve Vector Data 这个选项，系统会自动设置 `largeContentSizeImage`，如下图：



> 这个功能其实是 Accessibility 内新增的特性，如果感兴趣的话可以观看 What's New in Accessibility

终于支持 AutoLayout 了

重磅消息！我们终于可以在 `UIToolBar` 和 `UINavigationBar` 上使用 AutoLayout 了。

这里有几个点需要我们注意：

- 开发者只需要设置自定义 View 内部的约束即可。
- `UINavigationBar` 会负责自定义 View 显示位置的控制。
- 开发者负责控制大小。控制大小的方式有三种：
  - 指定高度和宽度约束。
  - 实现 `intrinsicContentSize`。
  - 通过子视图把自定义 View 撑开。

什么是 `intrinsicContentSize`？

Intrinsic Content Size：固有大小。是苹果在 AutoLayout 中引入的一个概念，意思就是说我知道自己的大小，如果你没有为我指定大小，我就按照这个大小来。比如：大家都知道在使用 AutoLayout 的时候，`UILabel` 就不用指定尺寸大小，只需指定位置即可，就是因为，只要确定了文字内容，字体等信息，它自己就能计算出大小来。

> `UILabel`，`UIImageView`，`UIButton` 等组件及某些包含它们的系统组件都有 Intrinsic Content Size 属性。

Margin 的变化

什么是 Margin

首先我们需要了解一下什么是 Margin？官方对于 Margin 的解释其实挺拗口，其实通俗来说 Margin 就是边距。

原有的 Margin

我们先来看一下 iOS 11 之前就有的属性。

`layoutMargins`

这是 `UIView` 上的一个属性，类型是 `UIEdgeInsets`，`UIEdgeInsets` 这个类型大家应该就比较熟悉了，所以 `layoutMargins` 代表的就是 `UIView` 所有 subview 和其本身的边距了。如下图：

当然前提是这些 subview 是和 `layoutMarginsGuide` 做约束的。

`layoutMarginsGuide`

在 Autolayout 中作为约束的一个参考，如下图：

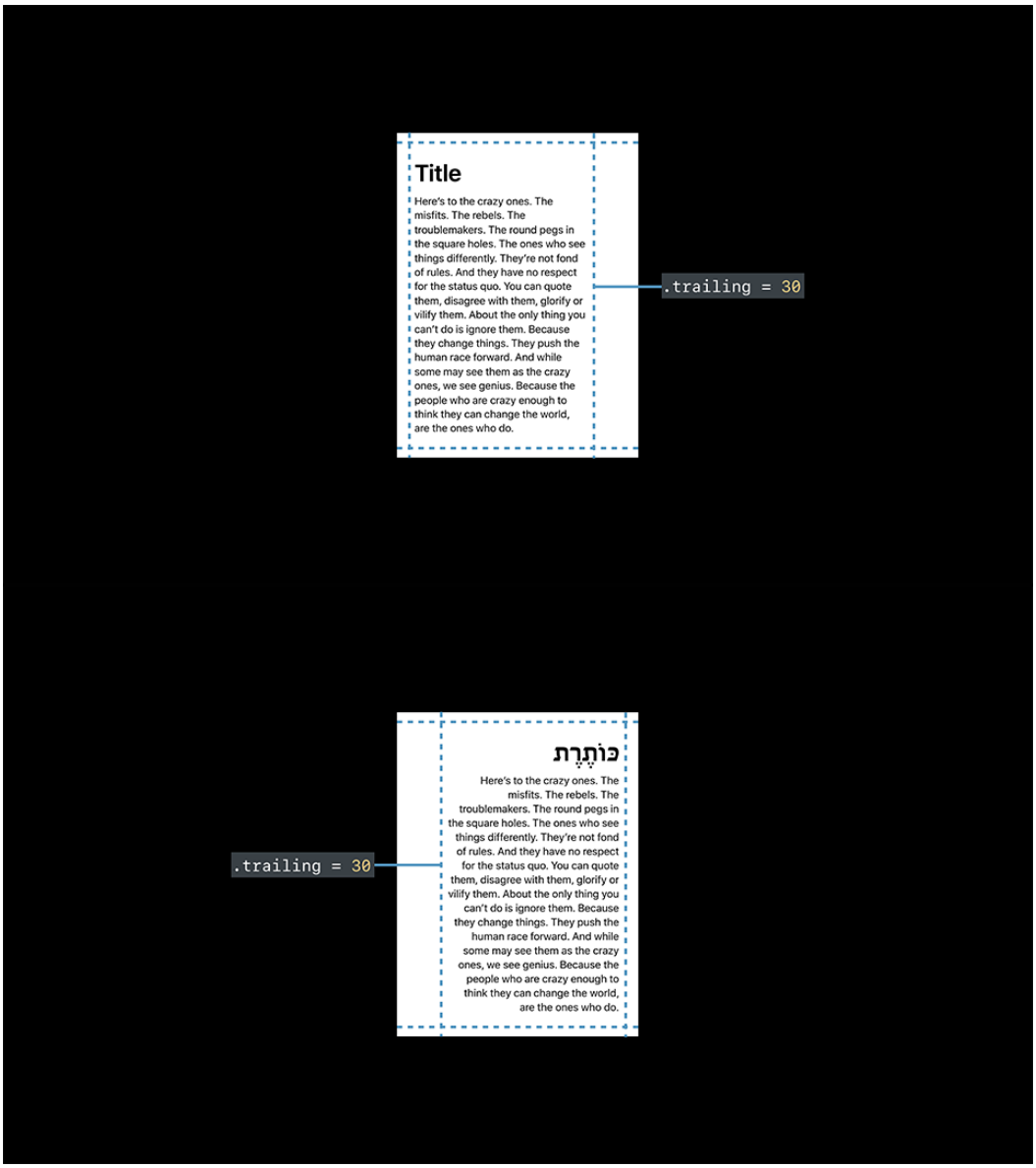
如果一个 `UIView` 所有的 subview 都是和 `layoutMarginsGuide` 做约束的话，当我们调节 `layoutMargins` 时，所有的 subview 都会受到影响。这里就有一个应用场景：如果你所有的 subview 都是基于 `layoutMarginsGuide` 来做约束的，当 UI 设计告诉你说他想把默认边距从 16 调整到 8。只需要一行简单的代码就可以搞定，如果没有的话，你就得一个一个约束去改了。What the fuck！！！

新增的 Margin

`directionalLayoutMargins`

iOS 11 引入了 `directionalLayoutMargins`，其实目的是为了多语言支持，对于大部分语言来说，都是从左到右的顺序，但是某些语言是从右到左的..... `directionalLayoutMargins` 其实就是用来解决这个问题的，当语言显示顺序是从左到右时，它的 .leading 和 .trailing 表示的是正常的左右，如果语言显示顺序是从右到左时，表示的就是右左了。另外要注意的是 `directionalLayoutMargins` 的类型是 `NSDirectionalEdgeInsets`。不过这个属性对于国内的大部分开发者来说，意义不大，除非是做一些阅读类的 App。

下面是从左到右和从右到左两种阅读顺序时设置 `directionalLayoutMargins.trailing` 为 30 的效果：



`systemMinimumLayoutMargins`

在 iOS 11 之前，`UIViewController` 的 view 默认有一个被锁定的 Margin，边距均为 16，而且我们无法对这个 Margin 做任何操作。iOS 11 之后，这个 Margin 终于被放出来了，即 `systemMinimumLayoutMargins`。当然，这个属性仍然是只读的，我们并不能直接修改这个属性，但是，苹果还新增了另外一个属性：`viewRespectsSystemMinimumLayoutMargins`。如果将其设置为 false，我们的和屏幕的边距就只由 `directionalLayoutMargins` 来控制了，这样一来，我们甚至可以将边距修改为 0，从而让我们的内容充满整个屏幕。

Safe area

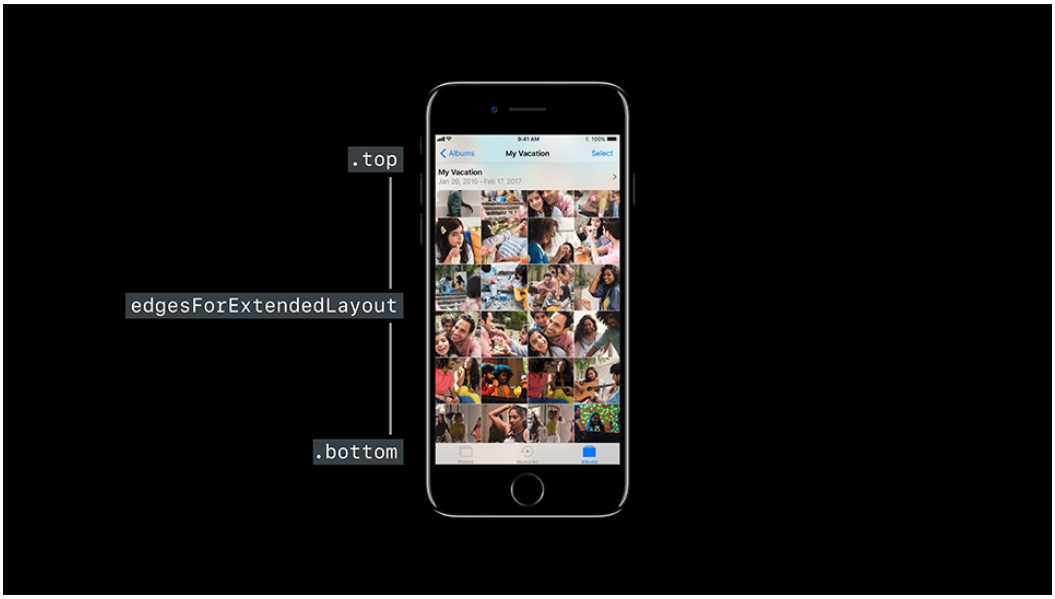
要了解 Safe area，首先我们要看一下 iOS 11 之前有的东西。

`edgesForExtendedLayout`

苹果在 iOS 7 的时候引入了半透明的 `UINavigationBar`、`UIToolBar` 和 `UITabBar`，并鼓励开发者把内容延伸到这些 Bar 的下面。而实现这个效果需要用到的属性就是 `UIViewController` 的 `edgesForExtendedLayout`。这个属性的类型为 `UIRectEdge`，包含了如下几个 case：

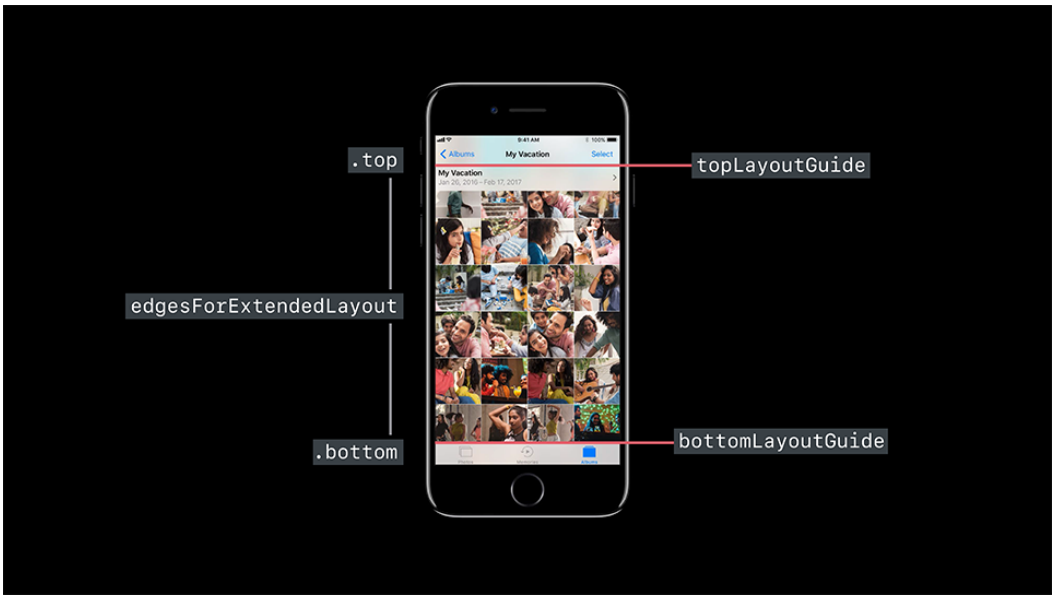
- `UIRectEdgeNone`：表示各个方向都不延伸内容。
- `UIRectEdgeTop`：表示向顶部延伸内容。
- `UIRectEdgeLeft`：表示向左侧延伸内容。
- `UIRectEdgeBottom`：表示向底部延伸内容。
- `UIRectEdgeRight`：表示向右侧延伸内容。
- `UIRectEdgeAll`：表示各个方向都延伸内容。

默认值是 `UIRectEdgeAll`。也就是各个方向都把内容延伸到 Bar 的下面。



`topLayoutGuide` 和 `bottomLayoutGuide`

然后系统会通过两个我们特别熟知的约束参考：`topLayoutGuide` 和 `bottomLayoutGuide` 来告诉我们顶部和底部 Bar 的大小。



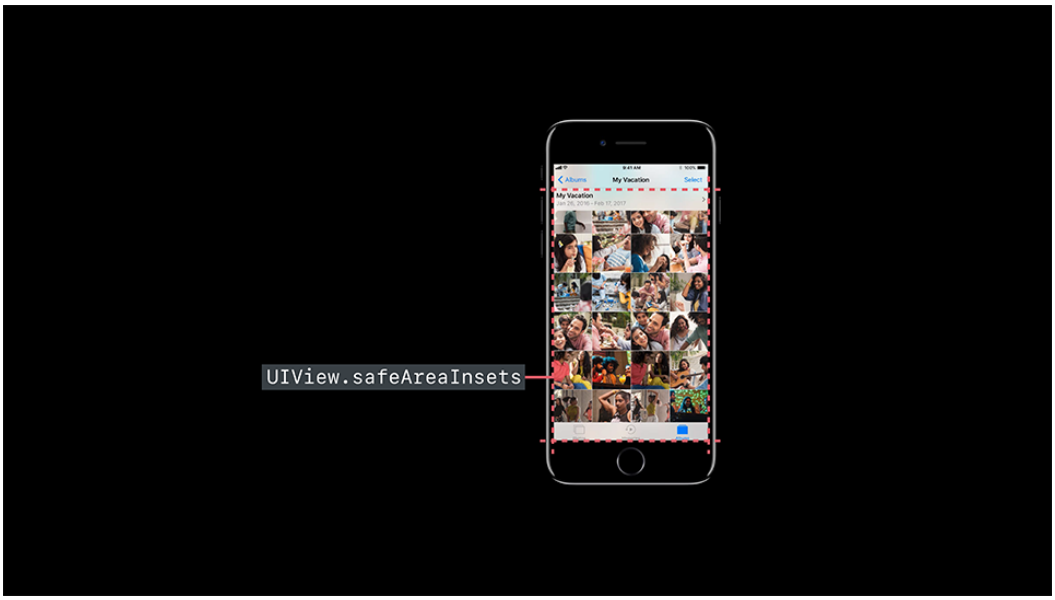
iOS 11 的变化

而 iOS 11 之后，这两个属性被 Safe area 所**代替**。官方对 Safe area 的解释如下：

- 描述不被父视图遮挡的视图区域。
- 通过 `safeAreaInsets` 和 `safeAreaLayoutGuide` 来使用。
- 在 tvOS 中结合 `UIScreen.overscanCompensationInsets` 来使用。

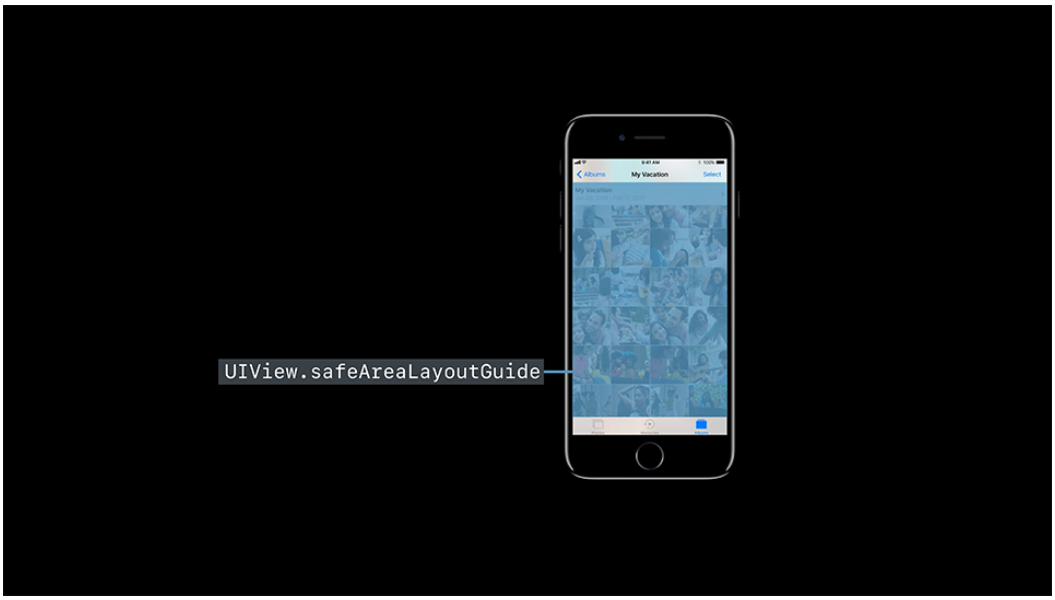
`safeAreaInsets`

`UIEdgeInsets` 类型，用于设置 Safe area 和 `UIView` 的边距。



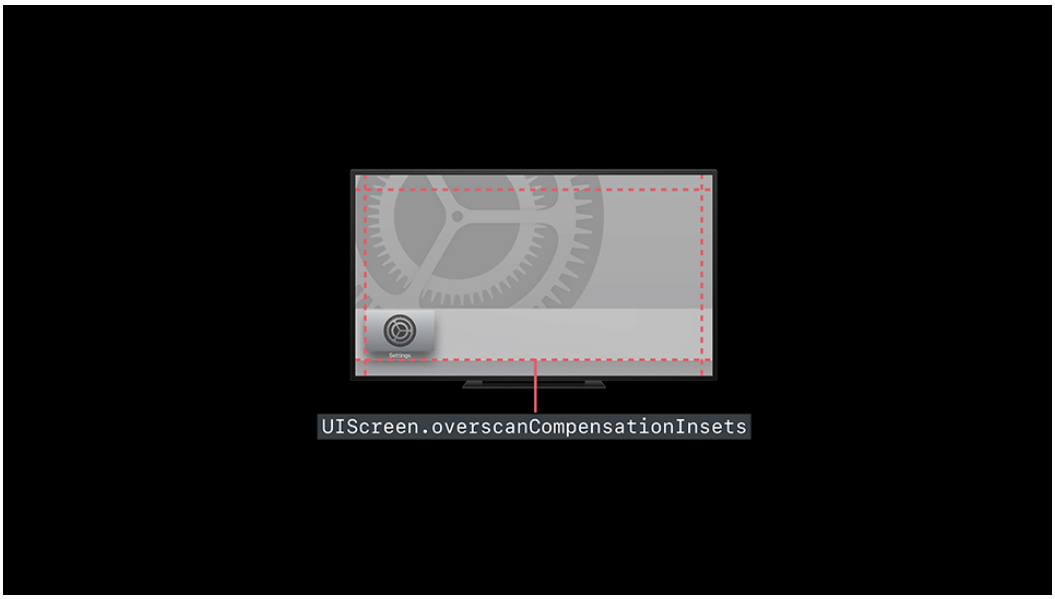
`safeAreaLayoutGuide`

Safe area 的约束参考，iOS 11 之后大家很快就会像熟悉 `topLayoutGuide` 和 `bottomLayoutGuide` 一样熟悉 `safeAreaLayoutGuide`。



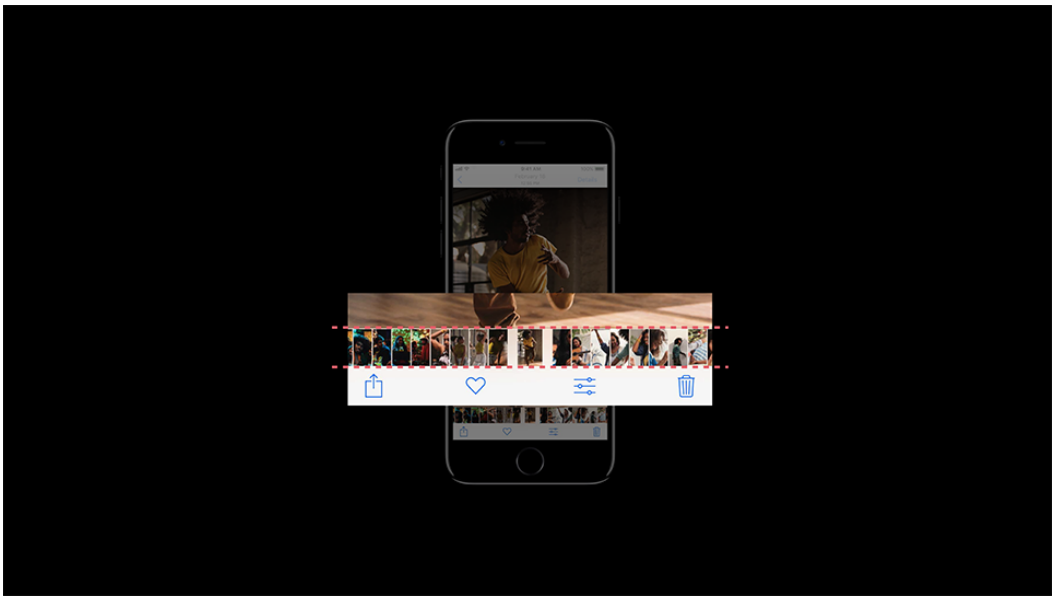
`UIScreen.overscanCompensationInsets`

`UIEdgeInsets` 类型，在 tvOS 中使用，可以让 `UIScreen` 中所有 `UIView` 的内容都显示在其规定的区域内，从而不会被电视的物理边框遮住。



`additionalSafeAreaInsets`

另外，iOS 11 还提供了一个属性：`additionalSafeAreaInsets`。当你在视图中添加了一个自己的 Bar，然后想让这个 Bar 和自带的 Bar 效果一样，就可以用这个属性来增大 `safeAreaInsets`，这样一来 Safe area 中的内容就也能延伸到这个 Bar 下面了。



Safe area 变化的回调方法

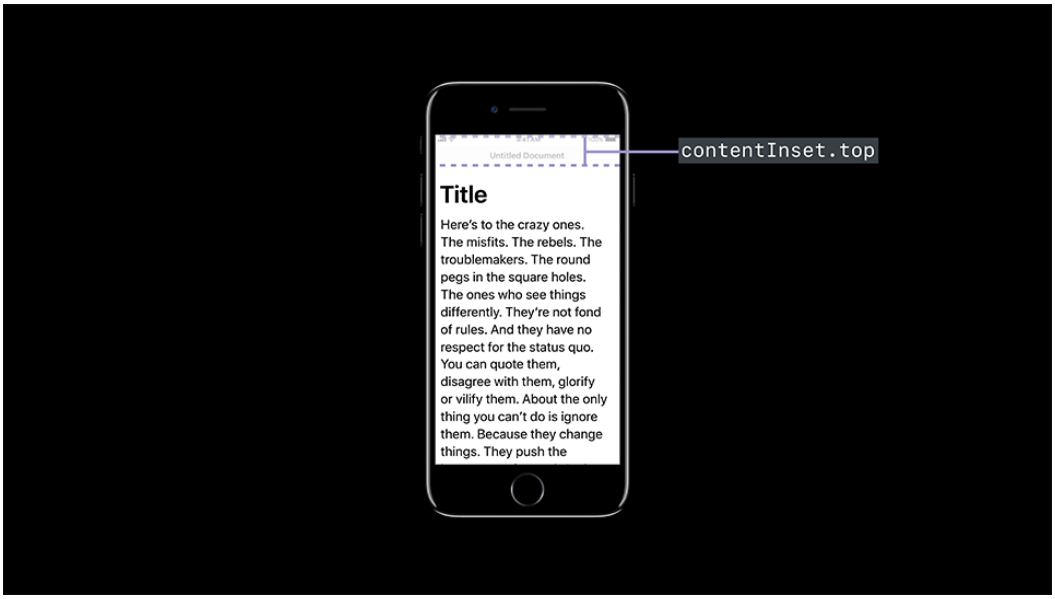
Safe area 发生变化的时候，iOS 11 还提供了两个回调方法来通知 `UIView` 和 `UIViewController`。

- safeAreaDidChange
- viewSafeAreaDidChange

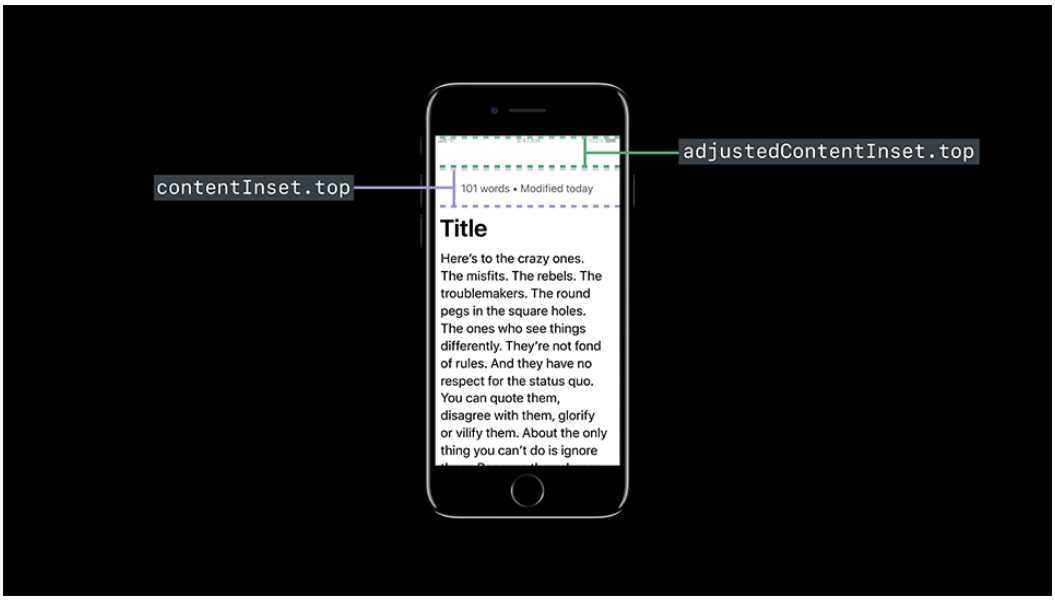
UIScrollView 的变化

`contentInset` 职能转移

在 iOS 11 之前，如果 `UINavigationController` 最顶部的 `UIViewController` 包含 `UIScrollView`，它会给这个 `UIScrollView` 的内容传递一个顶部边距。从而防止`UIScrollView` 顶部内容被遮住。



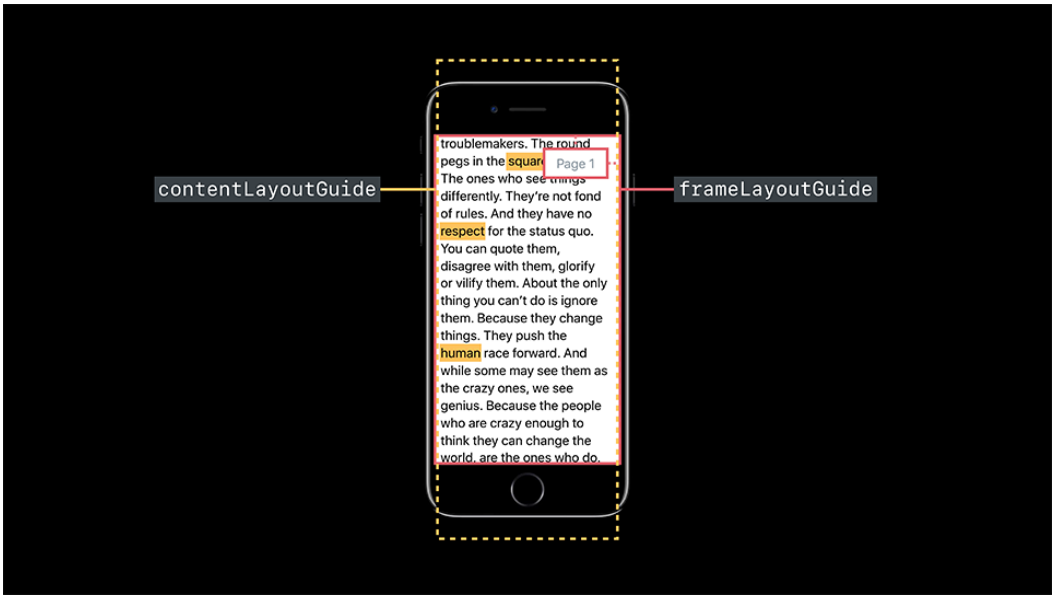
而 iOS 11 之后苹果不再用 `contentInset` 来填充滚动视图的内容。而是引入了另外一个属性 `adjustedContentInset` 来做 `contentInset` 原来做的事情。这也就意味着开发者可以用 `contentInset` 来做自己想做的一些事情，如下图：



> 注：这里可能会对一些第三方下拉刷新的库产生影响。

新增的 `frameLayoutGuide` 和 `contentLayoutGuide`

iOS 11 后 `UIScrollView` 新增了两个约束参考，一个是 `frameLayoutGuide`，如果某个 `UIView` 是和这个参考做约束，那它就不会随着 `UIScrollView` 的滚动而滚动，而是固定在指定的位置。另外一个是 `contentLayoutGuide`，这个参考和 iOS 11 之前一样，如果某个 `UIView` 是和这个参考做约束，那么它就会随着 `UIScrollView` 的滚动而滚动。



UITableView 的变化

AutoLayout 的支持

iOS 11以后，`UITableView` 的 Header、Footer 和 Cell 默认都使用 self-sizing，这也就意味着，我们不需要再关心 `UITableView` 中任何元素的高度，使用 Autolayout 的情况下，`UITableView` 会自动计算所有元素的高度。

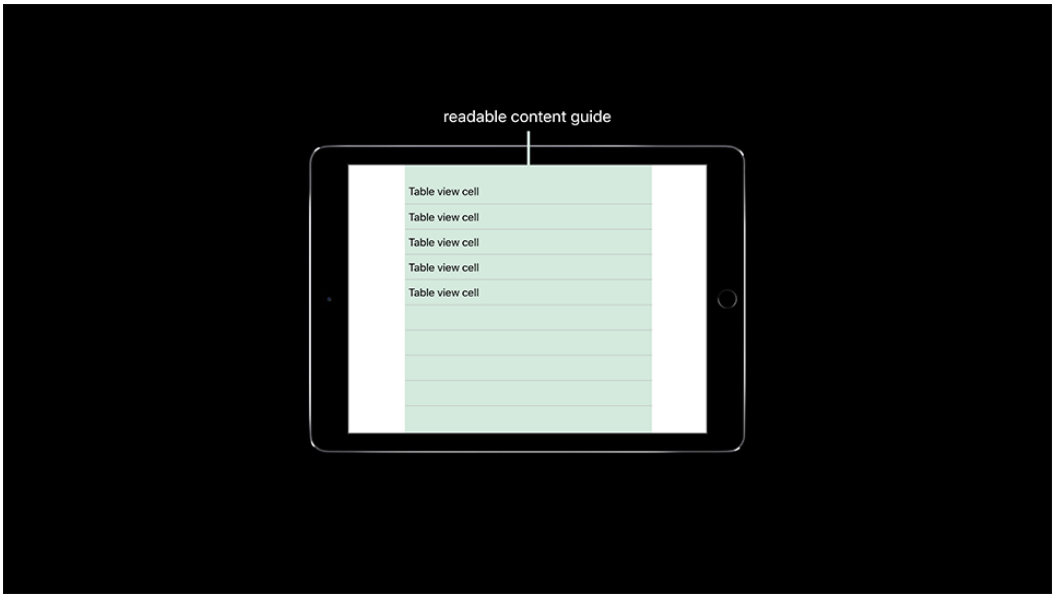
如果想禁用这个功能，使用下面的代码：

```
tableView.estimatedRowHeight = 0;
tableView.estimatedSectionHeaderHeight = 0;
tableView.estimatedSectionFooterHeight = 0;
```

Margin 和 Inset 的变化

`seperatorInset`

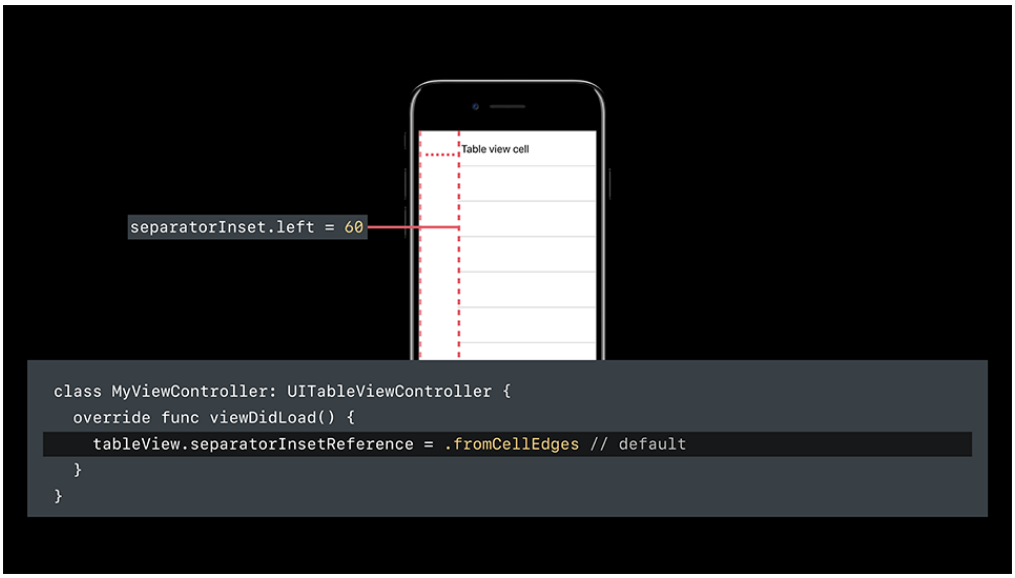
在 iOS 11 之前，`seperatorInset` 是根据 readable content guide 来改变的，这样在 iPad Landscape 模式下的时候，`UITableView` 的 Cell 就只能显示在一块固定的区域之内，并不能扩展到整个屏幕。



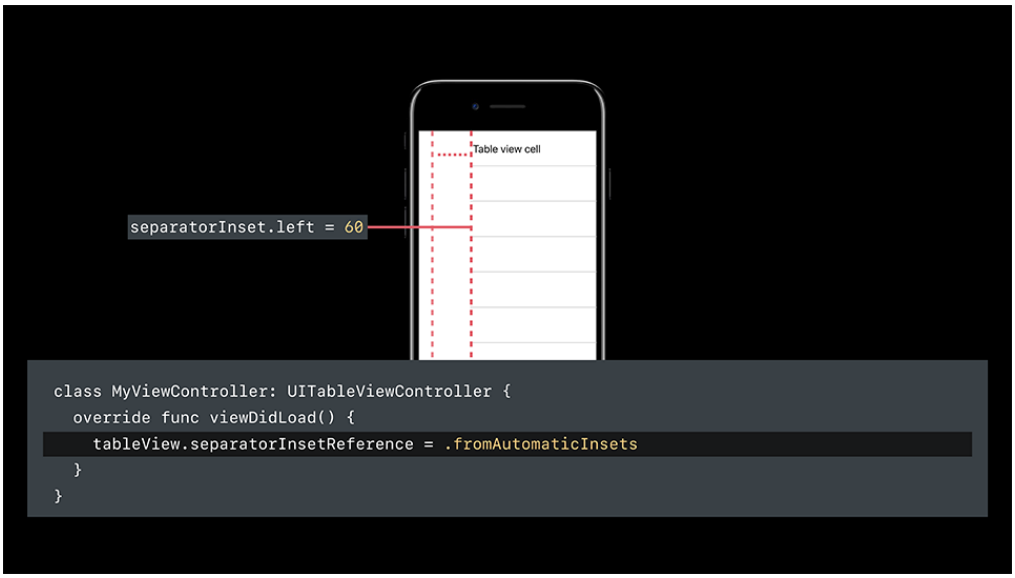
同时，`seperatorInset` 也是以 readable content guide 为基准产生作用，如下图，当我们设置 `seperatorInset.left` 为 30 时，变化是以 readable content guide 为基准产生的。

而 iOS 11 之后，`seperatorInset` 不再以 readable content guide 为基准产生作用，而是通过 `UITableview` 的 `seperatorInsetReference` 来决定其变化的基准。`seperatorInsetReference` 包含两种 case：

- .fromCellEdges：表示以 `UITableViewCell` 的边缘为基准。



- .fromAutomaticInsets：表示以 Safe area 的 insets 为基准。



如何配合 Safe area 使用

iOS 11 之后官方建议大家都根据 Safe area 来构建视图，为了让 `UITableview` 配合 Safe area 使用，这里有两点我们需要注意：

- 设置 `UITableview` 中 `seperatorInsetReference` 属性的值为 `.fromAutomaticInsets`。
- 所有的 Header 和 Footer 都要使用 `UITableViewHeaderFooterView`，包括 `UITableView` 的 Header 和 Footer，以及 Section 的 Header 和 Footer。

Swipe 操作

iOS 11 新增了如下几个特性：

- 支持滑动直接删除 Cell。
- 全新的 Swipe 样式和操作体验。
- Swipe 操作支持自定义图片。
- 左侧和右侧都支持 Swipe 操作。
- 提供回调方法可以让开发者取消 Swipe 操作。

结束语

iOS 11 新增的这些特性还是给开发者提供了极大的便利的。但是考虑到兼容旧版本的问题，能利用上的点并不是特别多，但是如果作为个人开发者，只考虑做一个单纯支持 iOS 11 的 App，这篇文章还是能提供一些帮助的。

另外，笔者把写文章时候的一些实践做了一个 Demo，开源在了 Github 上，配合学习效果会更好：https://github.com/mmoaay/WWDC17Session204。

优化输入体验的关键：keyboard技巧全介绍

这一节主要会讲如何构建更好的输入体验：

- 将键盘融入你的布局
- 使用 Input Accessory View
- 让你的 App 更好地应对多种语言输入
- 使用 traits 让文字补齐更加智能
- 支持实体键盘
- 创建自定义的输入控件
- Keyboard Extension 的一些建议和实践经验

让键盘融入你的 App

动态适应键盘

键盘的高度会跟随语言和设置变化，例如英文的键盘跟中文的九宫格键盘高度不同，第三方键盘的高度由于没有限制，所以任何高度都有可能。那我们该如何获取到键盘的高度呢，系统提供了六个通知：

- `UIKeyboardWillShow` 键盘即将出现
- `UIKeyboardDidShow` 键盘已出现
- `UIKeyboardWillHide` 键盘即将隐藏
- `UIKeyboardDidHide` 键盘已隐藏
- `UIKeyboardDidChangeFrame` 键盘的 frame 即将改变
- `UIKeyboardDidChangeFrame` 键盘的 frame 已改变

通过接收这些通知，我们可以获取到键盘 frame 修改前和修改后的值，但在深入探讨之前，让我们先来了解一种特殊状况。

在 iPad 里，用户可以随意调整键盘的位置，甚至是分成两个键盘，这个时候你不会想让键盘阻挡到 App 的内容，了解以下几点可以帮助你避免这样的情况发生。

- 隐藏键盘，或者是移动键盘位置都会发送 `Hide` 通知。
- Frame 修改的通知会在键盘位置移动后继续发送。
- 一般情况下，只要追踪最近一次发出的 `Hide` 和 `Show` 通知就可以了。

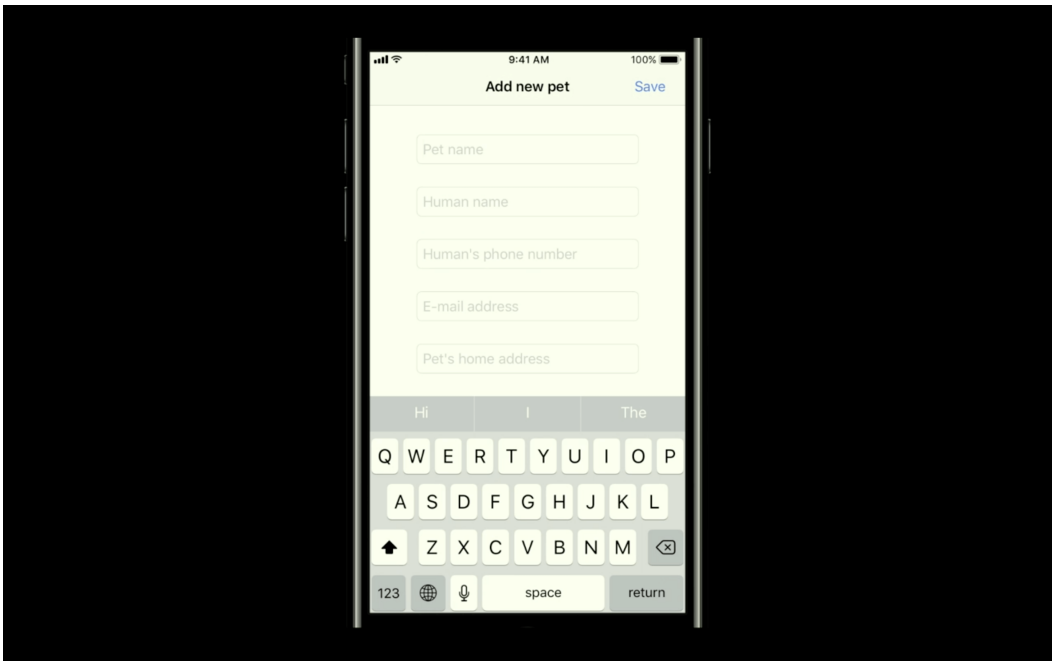
在了解完这个特例之后，让我们继续回到关于 Keyboard 的通知。

要记得键盘的 frame 总是以整个屏幕为参考系的，记得要把 frame 转化为我们使用的坐标系，然后获取我们的 view 和键盘的交集：

```
@objc func keyboardFrameChanged(_ notification: Notification) -> Void {
    if !keyboardIsHidden {
        guard
            let userInfo = notification.userInfo,
            let frame = userInfo[UIKeyboardFrameEndUserInfoKey] as? CGRect
        else {
            return
        }
        let convertedFrame = view.convert(frame, from:
UIScreen.main.coordinateSpace)
        let intersectedKeyboardHeight =
view.frame.intersection(convertedFrame).height
    }
}
```

不可滚动的布局

有时候我们需要跟不能滚动的布局打交道，例如登录页面的登录按钮，键盘出现的时候，我们希望它不被键盘挡到。



在这里我们有五个 textField，使用 layoutGuide 来控制它们之间的间隔，切换键盘的时候键盘就可以自动调整它们的间距，让五个 textField 全部显示出来，这是怎么做到的?

> **注：**这个页面布局的基本思路是，在 textField 之间添加空白的 view，然后将这些 view 使用 AutoLayout 指定为相同高度，最后让最下面的 view 的 bottom 对齐键盘的 top 就可以了。iOS 9 之后可以使用 layoutGuide 取代这些占位的 view，进一步提高性能。

首先我们创建一个自定义的 `keyboardGuide`，用来跟踪键盘高度的变化，然后给 `keyboardGuide` 一个高度的约束 `heightConstraint`，以便我们接下来再进行调整：

```
func setUpViews() {
    // ... view set up ...
    let keyboardGuide = UILayoutGuide()
    view.addLayoutGuide(keyboardGuide)
    heightConstraint = keyboardGuide.heightAnchor.constraint(equalToConstant:
kDefaultHeight)
    heightConstraint.isActive = true
    // ... view set up ...
}
```

接着我们把 `keyboardGuide` 跟 view 的 `safeAreaLayoutGuide` 绑定起来，然后将最下面的 layoutGuide（在这里是 `bottomSpacer`）跟 `keyboardGuide` 绑定起来：

```
func setUpViews() {
    // ...
    keyboardGuide.bottomAnchor.constraint(equalTo:
view.safeAreaLayoutGuide.bottomAnchor).isActive = true
    bottomSpacer.bottomAnchor.constraint(equalTo:
keyboardGuide.topAnchor).isActive = true
    // ...
}
```

最后，在接收到通知后，把 `heightConstraint` 的数值改成键盘遮挡的高度：

```
@objc func keyboardFrameChanged(_ notification: Notification) -> Void {
    if !keyboardIsHidden {
        // ... frame 坐标系转化 ...
        UIView.animate(withDuration: 0.2) {
            heightConstraint.constant = intersectedKeyboardHeight
            view.layoutIfNeeded()
        }
    }
}
```

可滚动的布局

平时更常见到的是可滚动的布局，为了让 ScrollView 里面的内容一直保持可见，我们需要调节 ScrollView 的 contentInset，跟之前做的很类似，只是稍微复杂了一点：

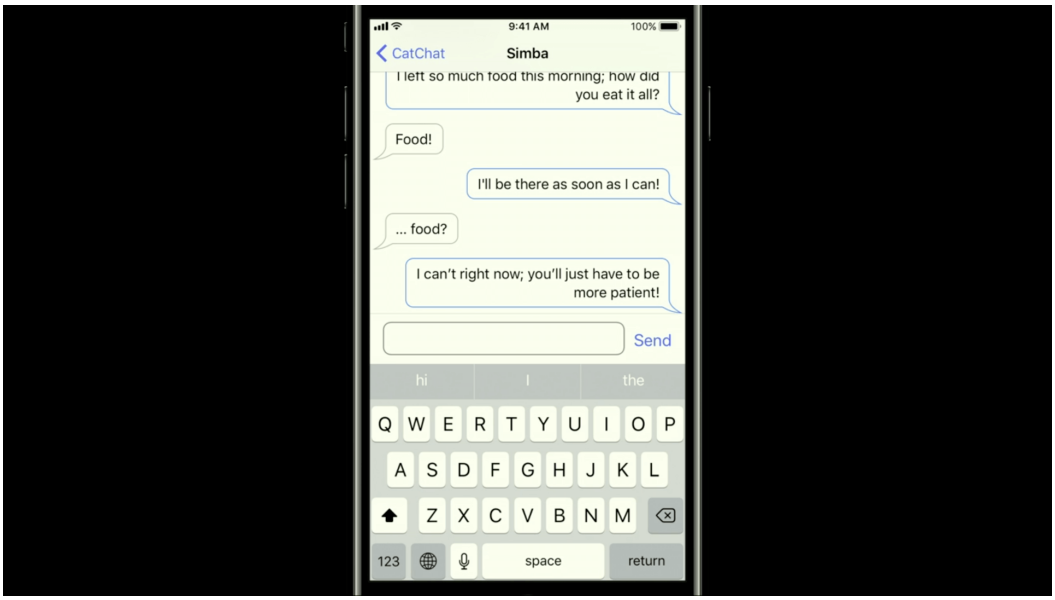
1. 保证键盘可见
2. frame 坐标系的转化
3. 给 contentInsets 设置一个合适的值
4. 如果有需要的话对于 ScrollView 的内容做处理

```
@objc func keyboardFrameChanged(_ notification: Notification) -> Void {
    if !keyboardIsHidden {
        // ... frame 坐标系转化 ...
        scrollView.contentInset.bottom = intersectedKeyboardHeight
        // ... 处理 ScrollView 的内容 ...
    }
}
```

如果使用 TableView 的话，那就更简单了，只要直接滚动到相应的 Row 就行了。

```
@objc func keyboardFrameChanged(_ notification: Notification) -> Void {
    let bottomRow = IndexPath(row: items.count - 1, section: 0)
    tableView.scrollToRow(at: bottomRow, at: .bottom, animated: true)
}
```

添加 Accessory View

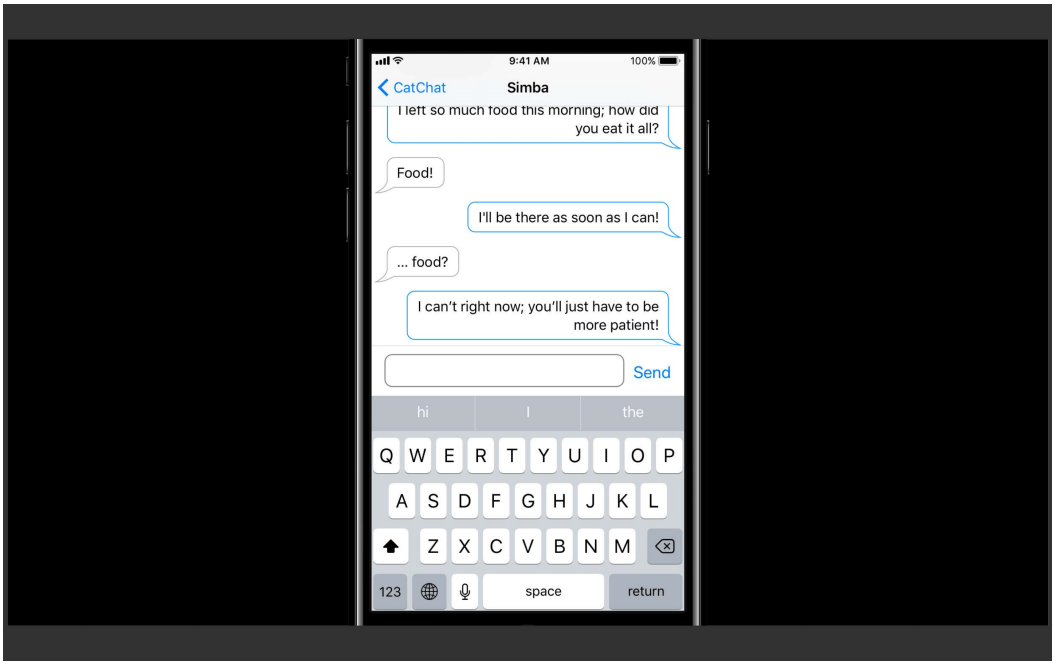


现在让我们来聊一下如果扩展输入框，添加一个 Accessory View，什么是 Accessory View？它是一个键盘上方的一个 view，在这里是 textField + button，也可以是一个 toolBar，还可以是图片，任何你想放上去的东西都可以，因为它就是一个普通的 view。

添加的方式很简单，重写 `UIViewController` 的 `canBecomeFirstResponder`，返回 `true`，并且重写 `UIView` 的 `inputAccessoryView` 或者 `UIViewController` 的 `inputAccessoryViewController`。

让 Accessory View 拥有动态高度

接着我们再来聊聊更多人在意的一个话题，动态高度的 Accessory View，例如 IM 软件里的输入框，我们希望可以根据输入的文本长度，把输入框撑开，以便让我们看到输入的整体内容。



如果用的是 UITextView 的话，可以通过设置 textContainer 的 `heightTracksTextView` 属性，并且禁止 textView 的滚动来实现。

```
expandingTextView.textContainer.heightTracksTextView =
trueexpandingTextView.isScrollEnabled = false
```

然后重写 `instrinsicContentSize`，计算内容高度，设置最小和最大高度，让内部的 textView 可以根据文本长度把我们自定义的 accessoryView 高度撑开。

> 注：`instrinsicContentSize` 是 AutoLayout 系统用来确定控件内容大小的一个属性，如果没有对尺寸有约束的话，就会直接使用 `instrinsicContentSize` 作为控件大小。

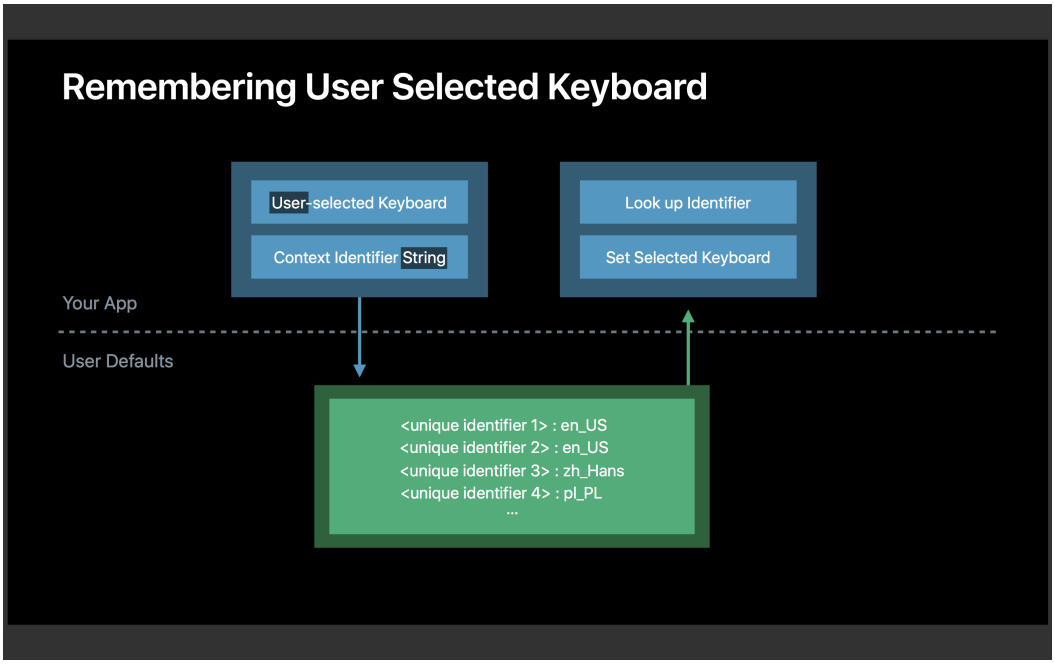
```
// 使用 intrinsicContentSize 来决定高度
override var intrinsicContentSize: CGSize {
    var newSize = self.bounds.size
    newSize.height = minHeight

    if expandingTextView.bounds.size.height > 0 {
       newSize.height = expandingTextView.bounds.size.height +
verticalPadding
    }
    if newSize.height > maxHeight {
       newSize.height = maxHeight
    }

    return newSize
}
```

使用上下文优化输入体验

让你的 App 更好地支持多语言输入

如果大家有外国友人或者是在国外生活的话，应该会遇到多语言输入的问题，跟家里的爸妈聊天的时候需要用中文输入法，而跟身边的同事聊天的时候会用到英文输入法，我们需要不停地来回切换键盘。

如果 App 能够知道每一段对话使用的输入法，或者说记录下来的话，输入体验就会好很多，实际上 iMessage 很早就做到了这一点，这是怎么做到的呢?



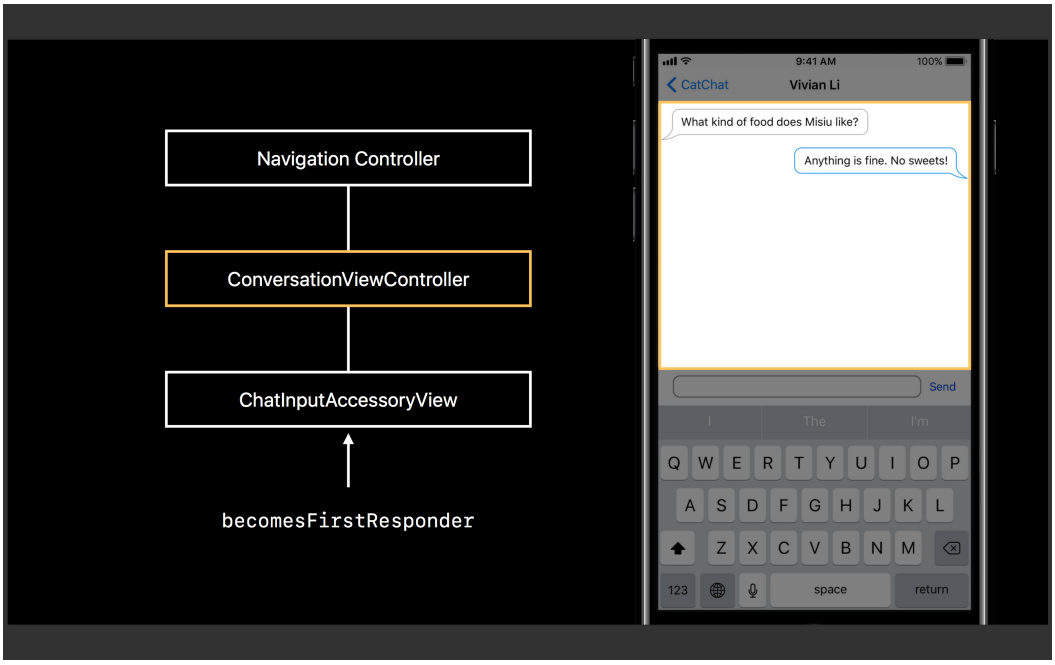
我们通过一个标识符，将输入法和输入控件关联到一起，这个标识符就是 UIResponder 的属性 `textInputContextIdentifier`。

当 textView 成为第一响应者时，系统就会去 UserDefaults 里使用 `textInputContextIdentifier` 查找有没有相应的输入法记录，有的话就会使用。

而每次键盘弹起或者切换的时候，系统也会在 UserDefaults 里以 `textInputContextIdentifier` 为 key 去记录这次使用的输入法。

听起来有点复杂，但要做的事情很简单，设置一个合理的 `textInputContextIdentifier` 就可以了，例如当前对话的用户的 id，当前对话群组的 id，都可以。

但我们该给哪一个 responder 设置 `textInputContextIdentifier` 呢?



每次键盘升起之前，我们自定义的 Accessory View 会成为 firstResponder，接着是 View Controller，Navigation Controller...

系统会沿着响应链去查找 `textInputContextIdentifier`，找到之后立刻唤起相应的输入法。

在这里我们把输入逻辑交给 ConversationViewController 处理：

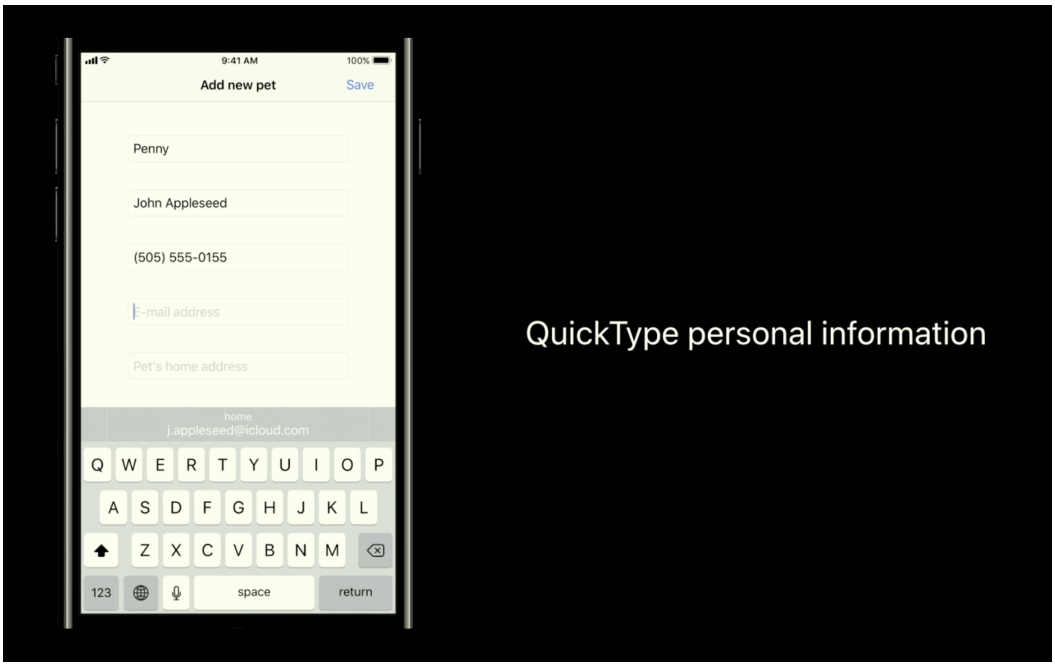
```
class ConversationViewController: UITableViewController, UITextViewDelegate {
    // ... other code ...
    override var textInputContextIdentifier: String? {
        // 返回一个唯一的标识符
        // 以便系统记录用户最近一次，在这段对话里使用的输入法
        // 这个标识符可以是任意的东西，例如这段对话的 id
        return self.conversation?.id
    }
    // ... other code ...
}
```

简单的一段代码就完成了输入法的自动记忆，重点是找到一个**合适**的 responder 设置一个**合理**的标识符。

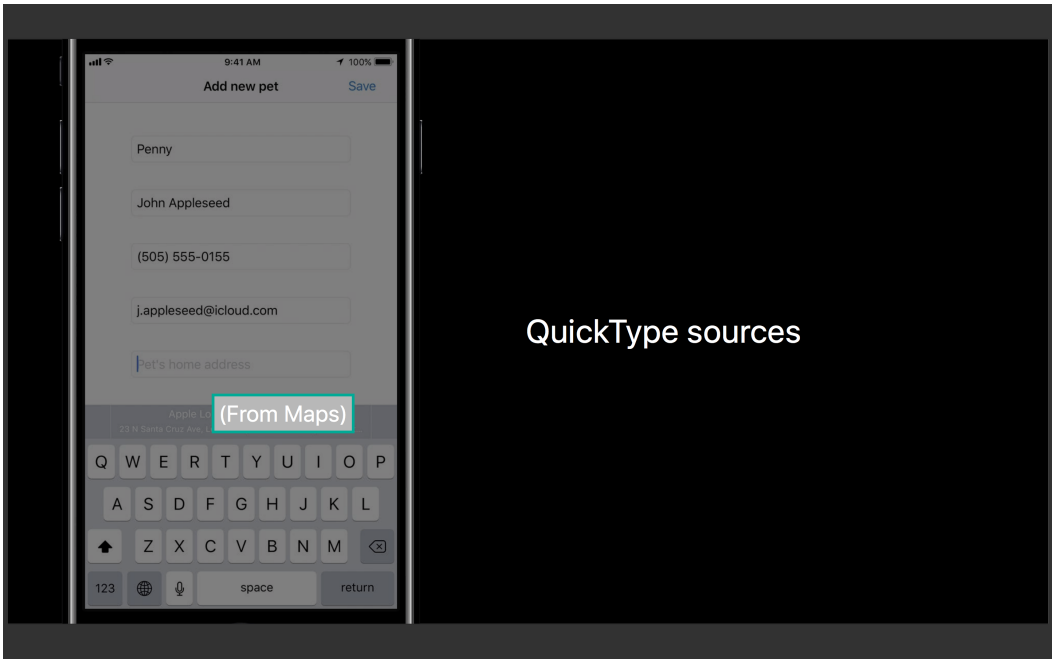
如果你不止想要记忆输入法，还想为某个特定的语言提供自动纠正之类的功能的话，可以通过 `textInputMode` 去获取输入法使用的语言。

更加注重上下文

在 iOS 4 的时候引入了 UIKeyboardType，让开发者可以根据上下文，在 App 里输入时弹出合适的键盘，去年我们又更近了一步，引入了 UITextContentTypes 让开发者可以提供上下文给系统，让系统可以更好地完成**输入预测**，**自动纠正**等等功能。



用户通过这些优化，甚至不需要打字，只需要点击 QuickType Bar 上系统提供的预测结果，就可以完成输入操作。

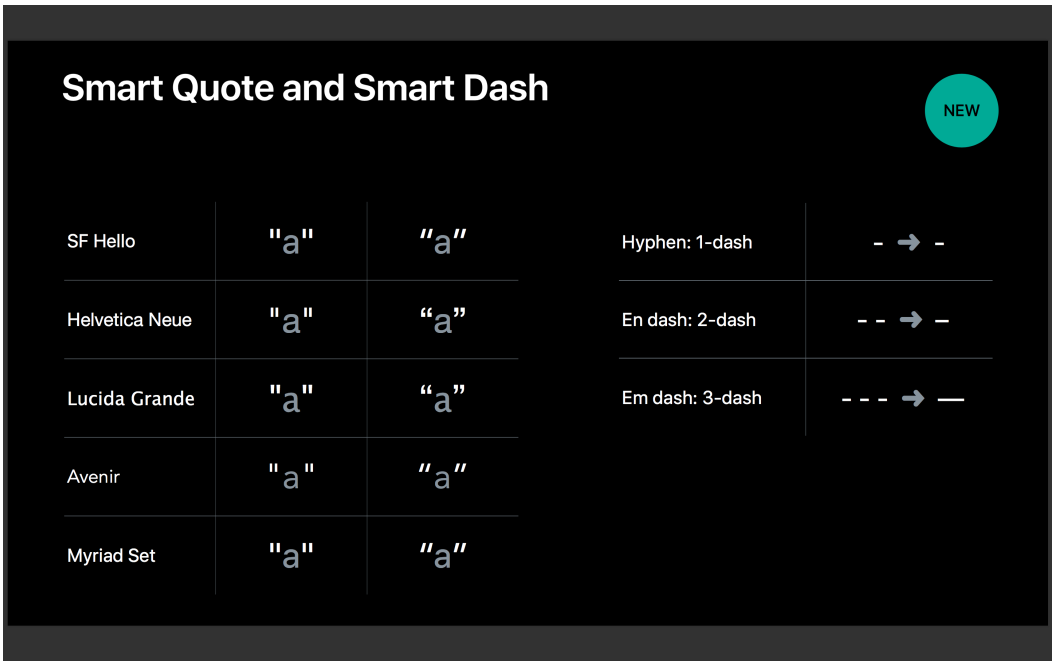


输入地址之前，如果我们打开原生的地图应用，搜索过某个地址的话，还可以在 QuickType Bar 看到这个地址，这是通过系统的 NSUserActivity 实现的，更多内容可以查看 [WWWDC 2016 Session 240 Increasing Usage of Your App With Proactive Suggestion]。

今年 iOS 11 还给 UITextContentType 新增了两个预设值，让用户可以在 App 里直接使用 Safari 里记录的登录信息，自动填充用户名和密码，更多内容可以查看本书另一篇文章《iOS 11 里 App 终于可以密码自动填充了》。

更智能的输入

更智能的引号和破折号



今年 iOS 会自动将双引号转化为符合语境的双引号，连续的两个 `-` 会转化为短破折号，三个会转化为长破折号。

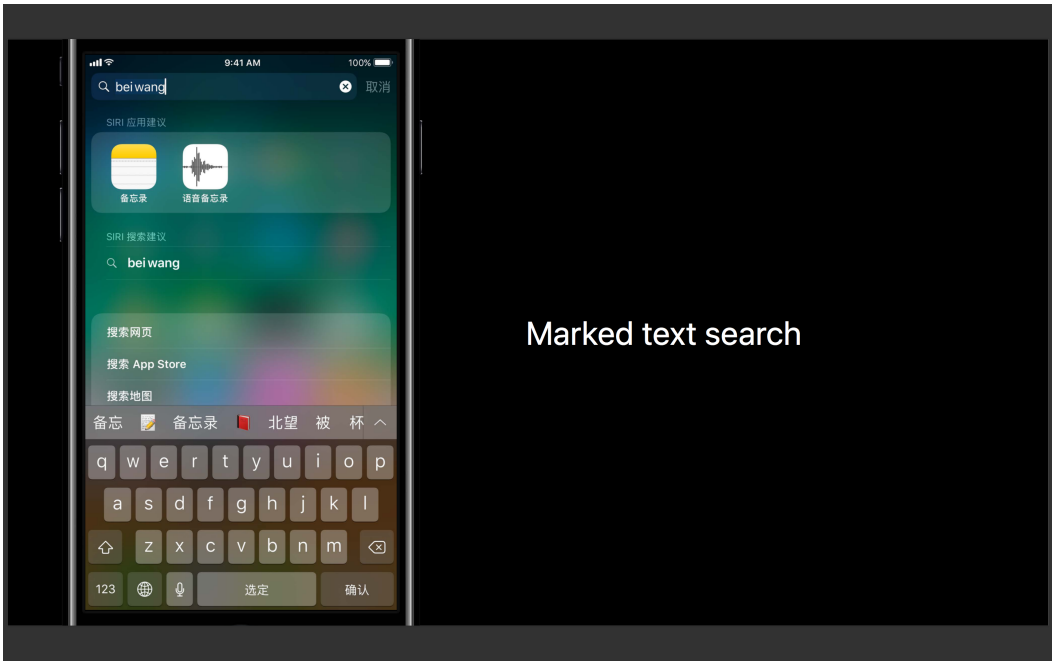
智能插入和删除

在手机上输入英文的时候，插入和删除永远都很烦人，除了增/删单词之后，你还总是需要重新移动光标去再处理单词前后的空格，今年 iOS 将会自动为用户处理前后空格的问题。

```
.default
.yes
.no
```

另外，以上提到的这些智能输入功能，默认都会开启（`.default`），但开发者也可以手动关闭一部分，例如输入快递单号的时候，就不需要自动纠正功能。我们只要通过 `UITextInputTraits` 去设置即可。

标记文本



提到文本输入，就不得不说标记文本，例如汉字，日文等等，总是需要先输入音符或者部首之类的信息，然后再从备选列表把要输入的字符挑选出来。

在 spotlight 里面，可以看到用户只输入了拼音，甚至都不需要选择具体的文字，搜索结果已经呈现了出来。

实现的原理也很简单，在用户还没有选择字符的情况下，我们可以通过 UITextInput 获取到系统预测的结果，提前进行搜索并呈现结果。

更多关于多语言输入的内容，可以查看 WWDC 2016 Session Internationalization Best Practices。

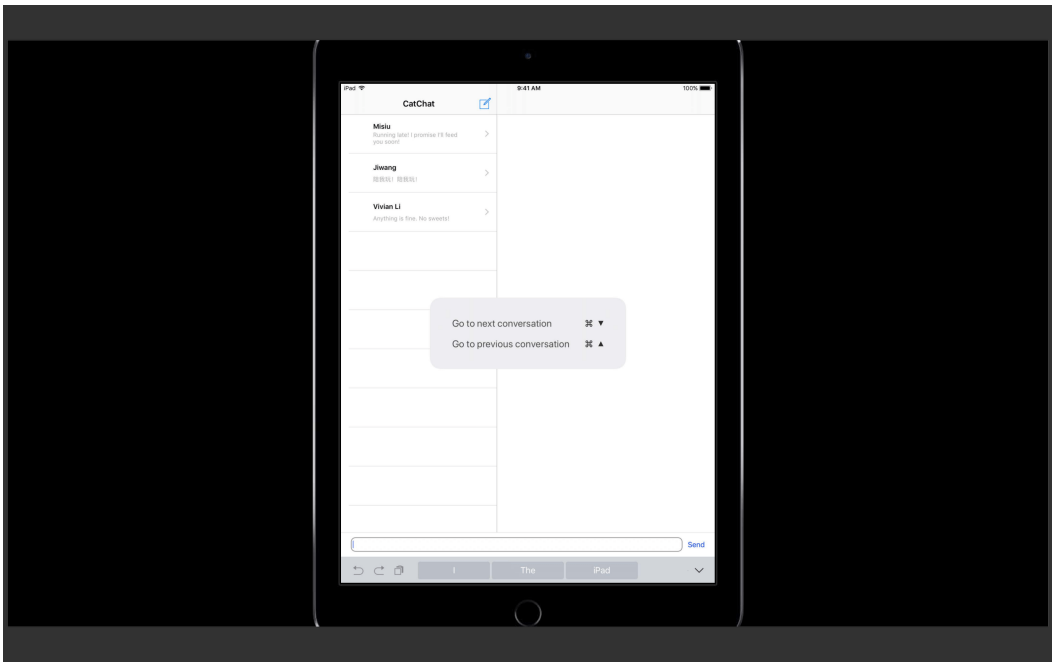
实体键盘输入优化

大家都会使用实体键盘，跟软键盘比起来，很大的一个区别就是快捷键，我们最常用到的可能是 `cmd + c`, `cmd + v` 之类的操作，iOS 今年也加入了快捷键的功能。

iPad 的 App 可以定义一套专用的快捷键，直接重写 `keyCommands` 属性就可以了：

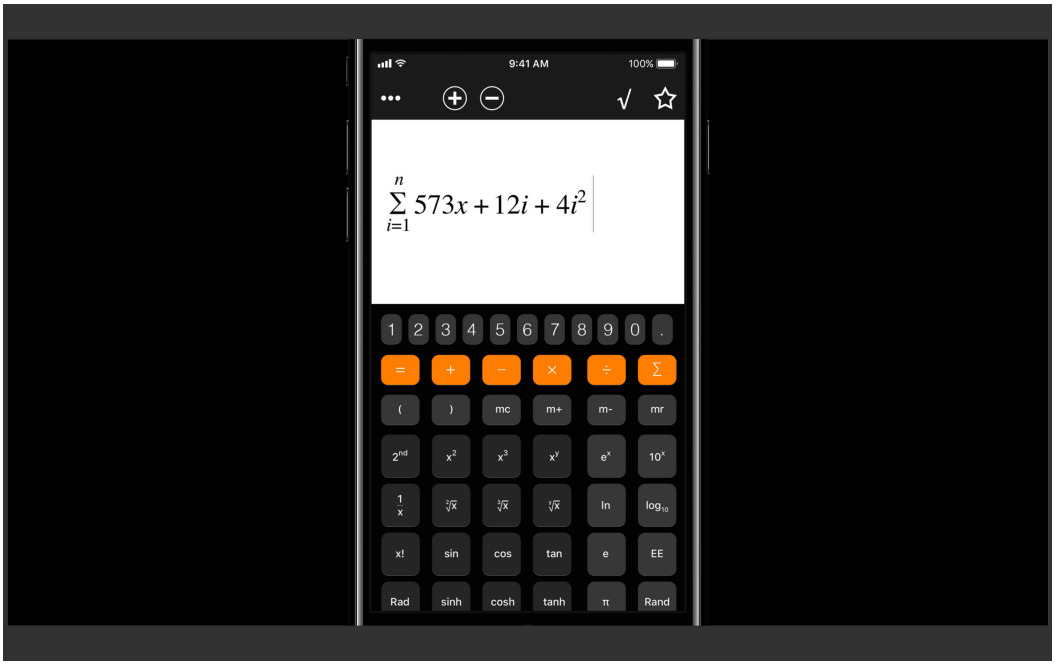
```
class ConversationViewController: UITableViewController, UITextViewDelegate {
    // ... some code ...
    override var keyCommands: [UIKeyCommand]? {
        return [
            // Command + 下箭头，切换到下一个对话
            UIKeyCommand(
                input: UIKeyInputDownArrow,
                modifierFlags: .command,
                action:
#selector(switchToConversationKeyCommandInvoked(sender:)),
                discoverabilityTitle:
NSLocalizedString("GO_TO_NEXT_CONVERSATION", comment: "")),
            // Command + 上箭头，切换到上一个对话
            UIKeyCommand(
                input: UIKeyInputUpArrow,
                modifierFlags: .command,
                action:
#selector(switchToConversationKeyCommandInvoked(sender:)),
                discoverabilityTitle:
NSLocalizedString("GO_TO_PREV_CONVERSATION", comment: ""))
        ]
    }
    //... some code ...
}
```

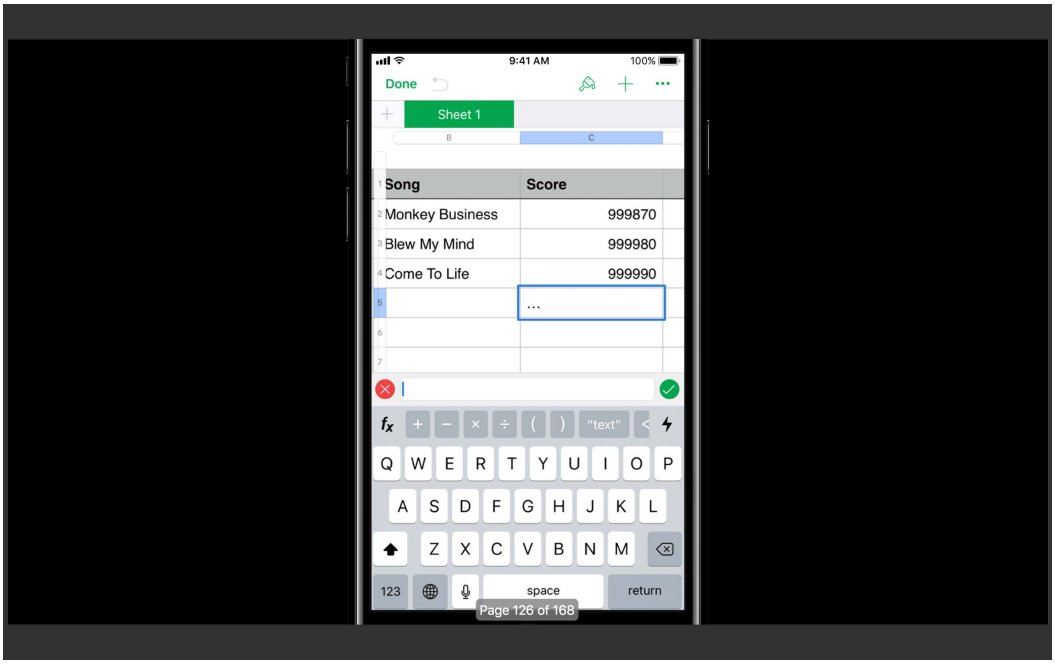
UIKeyCommand 的定义很简单，主键，辅助键（command，shift之类的），触发的动作，快捷键的名称。



定义好之后，我们就可以在快捷键提示的界面看到我们定义的快捷键了。

自定义输入控件


9:41 AM
100%
$\sum_{i=1}^{n} 573x + 12i + 4i^2$
1 2 3 4 5 6 7 8 9 0 .
= + − × ÷ Σ
( ) mc m+ m- mr
2nd x² x³ xʸ eˣ 10ˣ
1/x ∛x ∜x ln log10
x! sin cos tan e EE
Rad sinh cosh tanh π Rand

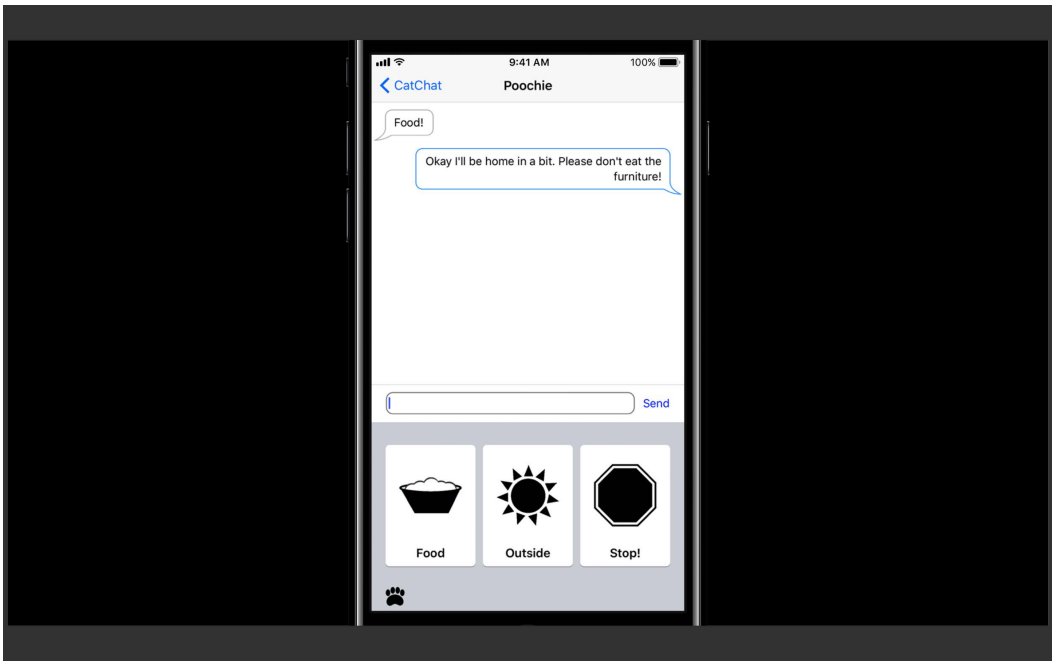

9:41 AM
100%
Done
Sheet 1
Song
Score
Monkey Business
999870
Blew My Mind
999980
Come To Life
999990
Page 126 of 168


The Shortest Route
learned up to now. (You might need your debugging skills, too!)
moveForward()
collectGem()
moveForward()
moveForward()
moveForward()
Run My Code
collectGem()
moveForward()
toggleSwitch()
turnLeft()
return

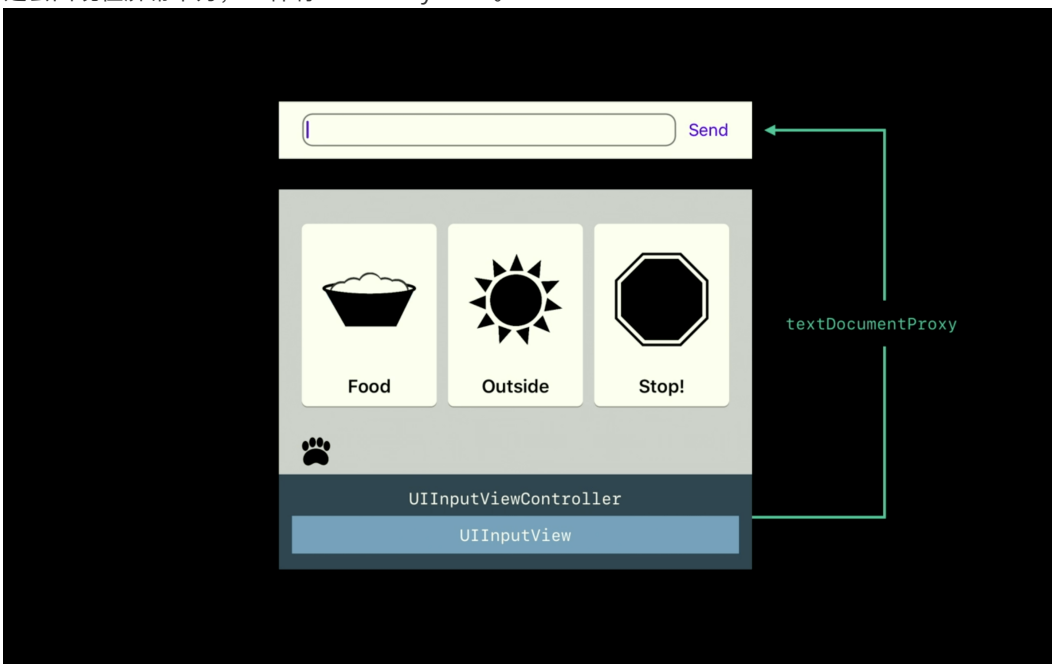
前面我们提到了 `UIKeyboardType`，让开发者可以选择键盘的类型，但这些原生的键盘类型也许满足不了你的输入需求，例如计算器需要各种数学符号，Swift Playground 需要自动补齐，表格软件里需要公式等等，这个时候我们就需要自定义键盘了。

为我们的宠物自定义一个 Input View

这里我们以一个宠物专用的 Input View 为例，讲解一下如何自定义 Input View，首先我们的宠物需要使用特定的语言，并且能表达的词汇很有限。



我们可以看到自定义的 Input View 有三个字符可以输入，喂食，散步和暂停，像普通的键盘一样也是会出现在屏幕下方，一样有 Accessory View。



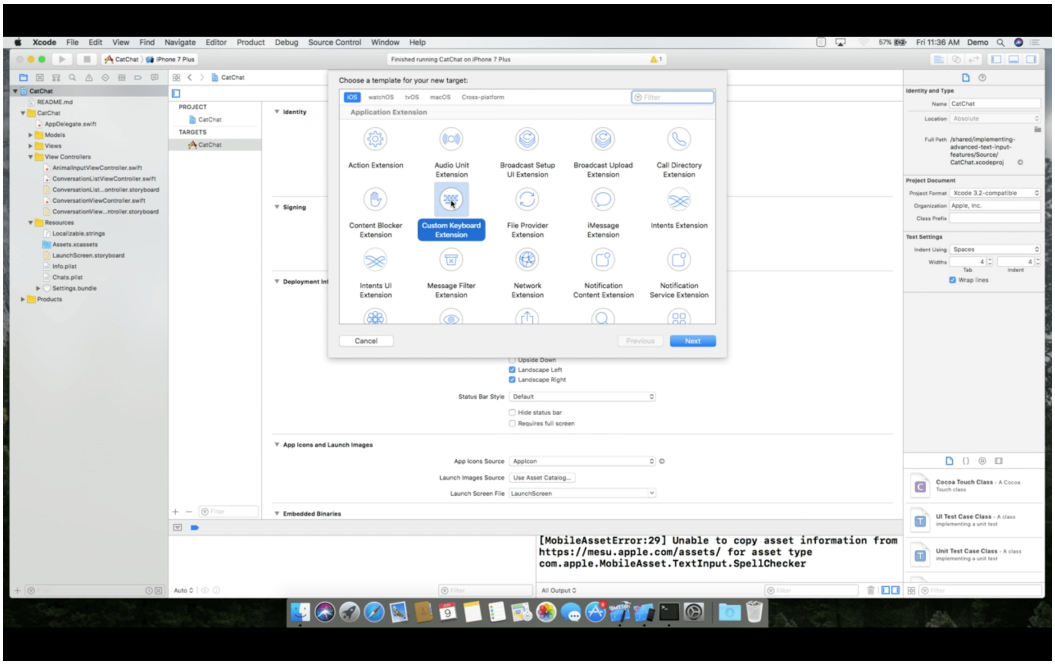
自定义 Input View 最简单的方式就是，继承 `UIInputViewController` 新建一个类，这样做有很多好处，例如跟系统键盘一样的灰色背景等等，更重要的是提供了一个 `textDocumentProxy` 去跟当前用户选择的 textField 进行交互，获得 `textContentType`，`inputTraits` 之类的上下文信息，`textDocumentProxy` 也可以往 textField 里插入文本，获取到插入点的位置，插入点前后的文本。

```swift
class ConversationViewController: UITableViewController, UITextViewDelegate {


    private let customInputView = AnimalInputView()

    override var canBecomeFirstResponder: Bool {
        return true
    }

    override var inputView: UIInputView? {
        // 返回我们自定义的 InputView 实例
        return customInputView
    }
    // ... 其它代码 ...
}
```

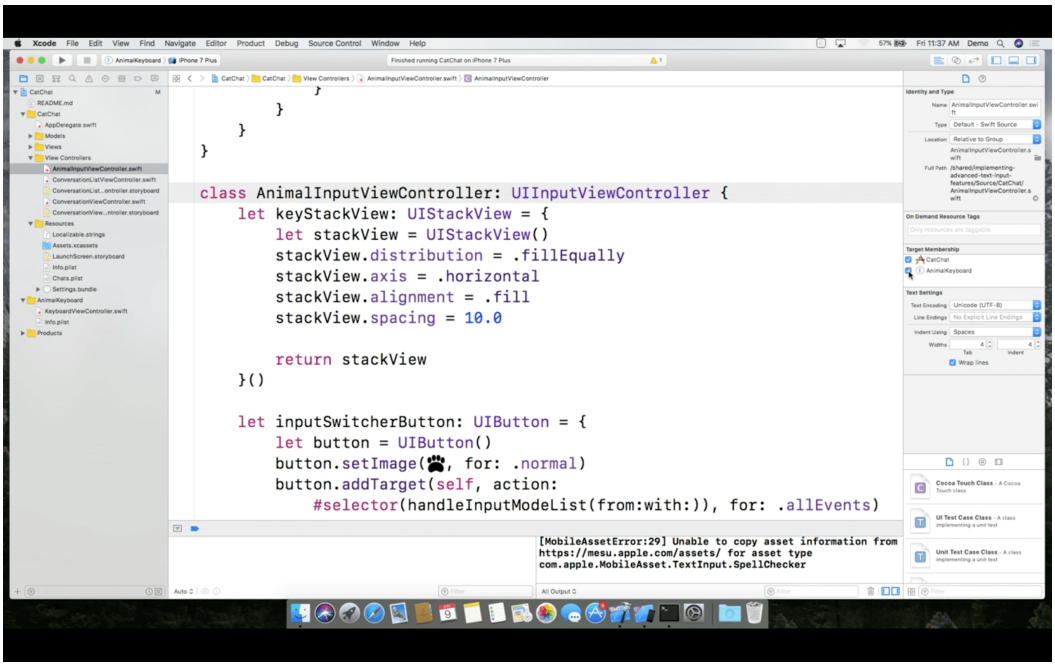
回到我们的宠物聊天软件里，`ConversationViewController` 首先需要重写 `canBecomeFirstResponder` 属性，让自己成为响应链的一环，接着重写 `inputView` 属性（或者 `inputViewController`），返回我们自定义的 inputView 就可以了。

Keyboard Extension

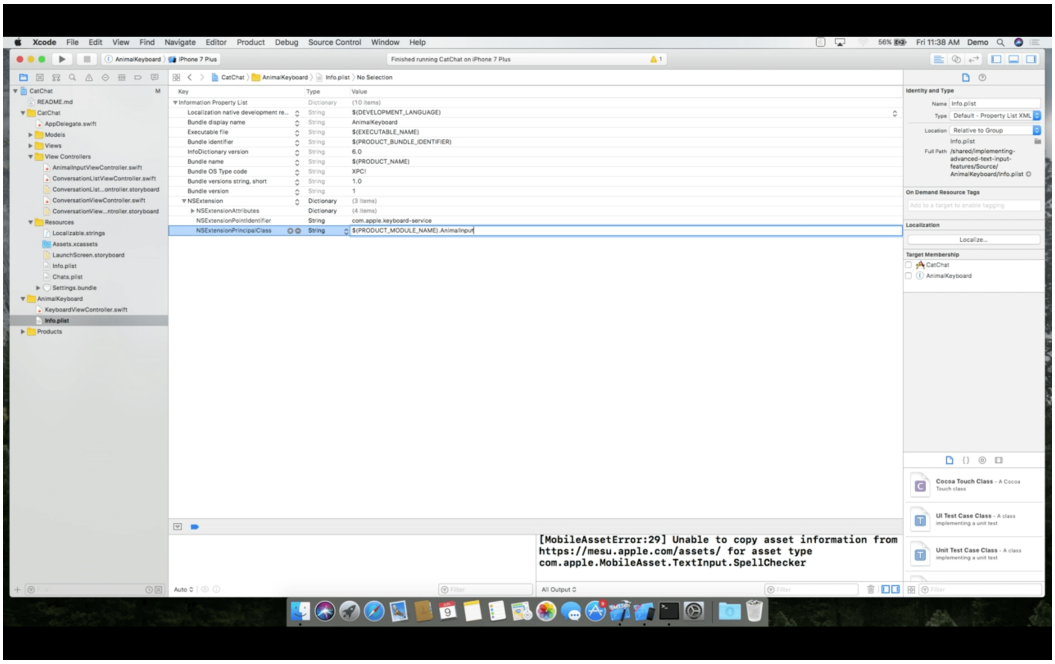
在 iOS 8 里引入了 Keyboard Extension，实际上把我们刚刚自定义的 inputView 转化为 Keyboard Extension 是一件非常简单的事情，这样就可以让用户在别的 App 里也能使用我们自定义的 inputView 了。



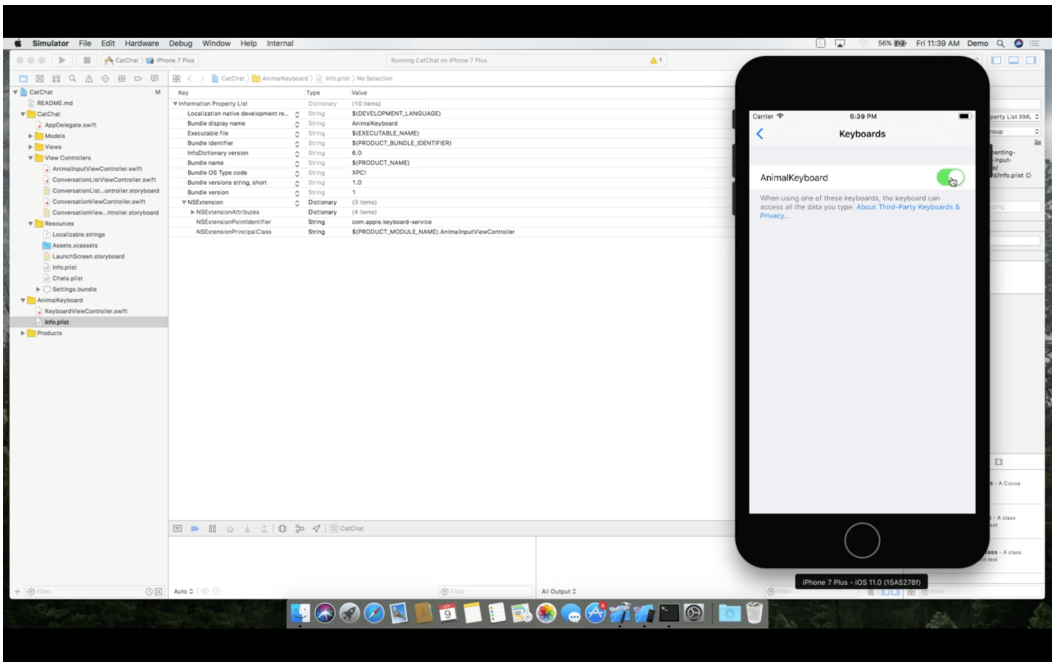
首先我们需要在项目里新建一个 Target，选择 Custom Keyboard Extension，系统会自动为我们创建一个 KeyboardViewController，直接把 CustomInputView 的逻辑搬到这里就可以了。



或者是复用我们在项目里定义好的 AnimalInputViewController 类型，像上图一样打开 AnimalInputViewController 文件，然后在右边的 Target Membership 里把刚刚新增的 Target 打上勾，让 AnimalInputViewController 也加入 AnimalKeyboard 的编译（直接在 Build Phase 里加入也可以）。



然后打开 AnimalKeyboard 目录下的 info.plist，如图将 NSExtensionPrincipalClass 改为 `${PRODUCT_MODULE_NAME}.AnimalInputViewController`（选择具体使用的 inputViewController 类型）。



最后，我们只要进入系统设置 -> PetChat（根据你的 App 名字不同而改变）-> Keyboards -> AnimalKeyboard（Keyboard Extension 的 target 名称），将键盘开关打开，然后在别的 App 里也可以使用自定义的键盘了。

iOS 11 新增的 API

- **选中的文字**。现在的 `inputViewController` 可以通过 `selectedText` 获取到当前选中的文字。
- **documentIdentifier**。`documentIdentifier` 的作用是来区分开每一个 textField，例如当用户切换 textField 时你可以把输入预测给清空。
- **请求获取最高访问的权限**。现在可以跟用户发起请求，要求获取最高访问权限，以便创建更好的智能输入体验，后面会更详细地介绍最高访问权限的内容。

第三方键盘 API

- **可以直接使用系统的输入法列表菜单**。而不是像之前那样只能切换到下个输入法。
- **个性化输入预测补齐**。这里的个性化是指，你可以获取到用户的通讯录资料，在优化输入预测时非常有用。
- **多语言支持**。前面提过的 `textInputContextIdentifier`，如果你的输入法也支持多语言的话，就可以通过这个去优化多语言输入的体验。

设计用户权限系统

- **隐私**。iOS 系统拥有一套隐私权限系统，让用户可以控制 App 可获取的隐私数据。* **使用用户信息**。例如将用户信息发送到云端进行分析，一定要保证有一套用户协议机制，在用户统一的情况下才可以使用用户的使用信息。
- **最高访问权限的请求**。Keyboard Extension 可以通过向用户请求最高访问权限，来获取更多用户信息以优化输入体验。

最高访问权限

这件事情的价值不在于最高访问权限的获取，很多功能都是原生支持的，不一定需要这么高的权限，权限的获取只是对于现有功能的补充，而不是必要的。

其中一个需要获取访问权限的场景是，keyboard 需要跟主 App 交互，例如访问数据库里的热门词条。或者是网络的请求，当前的地点，通讯录等等，需要注意的是通讯录本来就可以获取，只是在获取一些额外信息的时候需要申请权限，例如联系人备注之类的。

总结

- **如何让键盘更好地与 App 进行交互**
  - 动态适应键盘高度，调整 App 布局
  - 使用 Accessory View 来对于键盘进行功能补充
- **使用高级的智能输入功能，去优化用户体验**
  - 通过 `textInputContextIdentifier` 优化多语言输入
  - 使用 `textContentType` 让系统提供登录信息自动补齐的功能
  - 使用 `markedText` 获取输入结果预测
  - 为实体键盘创建快捷键
- **构建一个 Keyboard Extension 比你想象的更简单**
  - 自定义 inputView/inputViewController
  - 将自定义的 inputViewController 转化为 Keyboard Extension

高级开发应该掌握的自动布局技术

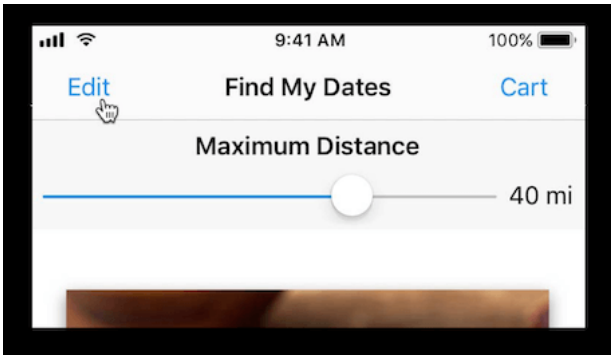
> 构建 app 时使用的自动布局技术，其实就是建立视图与视图之间关系。而约束是建立视图间关系的纽带，帮助我们的 app 可以适应各种尺寸的屏幕，在应对花样百出的布局需求时游刃有余。

前言

如果你以前从未使用过`Autolayout`，现在网上已经有很多很优秀的教程，包括往届 WWDC 中 sessions 视频资源都可供查看学习。在本文中将不再重复基本的使用方法，更多的去介绍一些更加复杂的场景中的应用，本文中技术结合实例使你更容易理解吸收。让我们一起来看看与`Autolayout`相关的六种技术与应用，这些内容都非常实用，在日常开发中一定会经常使用到，相信本文一定不会让你失望。

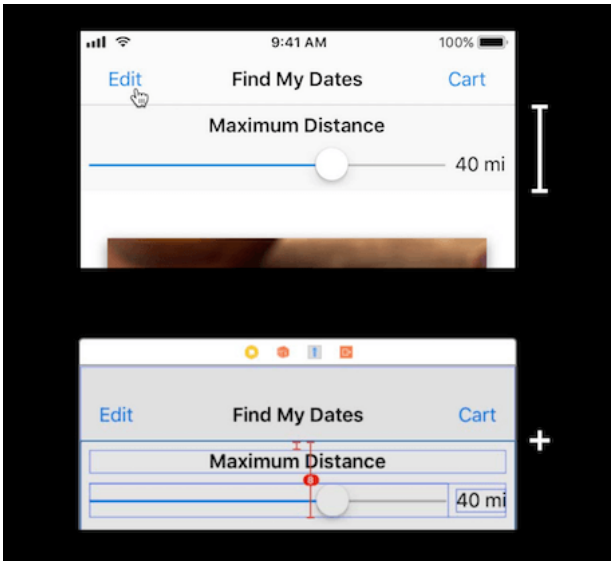
1. 运行时变换布局（Changing layout at runtime）

通常我们不仅可以在 app 中使用约束来对视图进行简单的定位，也可以组合使用以达到更复杂的效果。我们今天要讲的第一种技术点，便在运行时改变布局。如下图，在我们界面的顶部，有一个滑块区域。现在我们需要一个将滑块视图上移并且最终隐藏的功能。



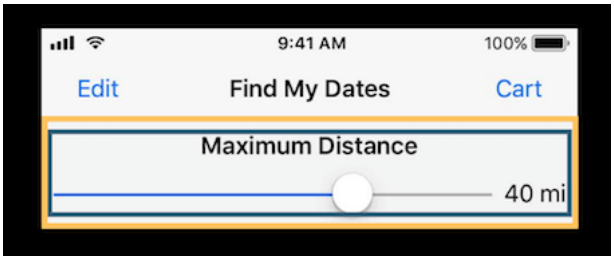
1.1 利用高度约束隐藏视图

通常我们希望约束在设置好之后不需要再次调整，尽量让结构清晰简单。现在我们来思考一下，从布局的角度使用最简单的方式实现这个功能，一般情况下，我们把这个区域视图高度缩短至0即可。但是如果我们真的添加上一个高度约束，并且设置为0。我们将在 Interface Builder 中发现一些警告。

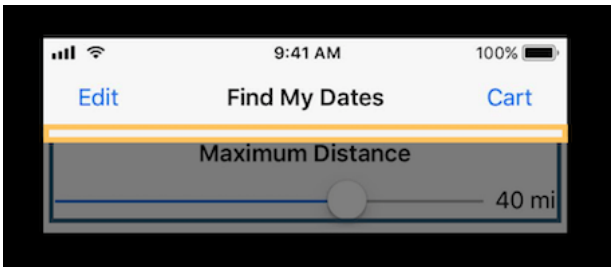


1.2 避免冲突

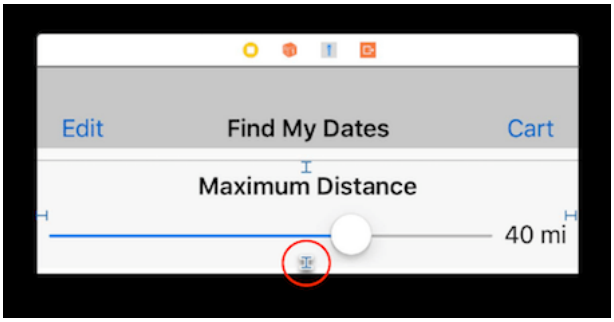
在图片中可以看到布局中的这些红线，这意味着我们设置的约束存在着一些冲突。之所以出现冲突，是因为我们设置的这些约束让布局引擎去做了一些不能同时并存的事情。而这个冲突出现是因为我们设置了高度为0的同时，无法保持足够的高度以满足该控件内部的内容显示。



为了解决这个问题，我们将 slider 和 label 所在的视图放进一个`warppingView`中，如图中橙色方框。在我们缩短`warppingView`的高度时，我们也要保证`warppingView`内部子视图的高度，并且满足子视图相关的约束，在启用 clips ToBounds 属性后，超出`warppingView`内部坐标系范围的内部控件在显示时将被裁剪掉。这样就达到了隐藏视图元素的效果，如下图效果，灰色区域将被裁剪不显示。



让我们来看看在 Xcode 中是如何做到的，我们需要在运行时控制`warppingView`的高度，所以我们将为`warppingView`手动创建一个高度约束`zeroHeightConstraint`，在运行时设置`zeroHeightConstraint`为0，并且在用户点击 Edit 按钮时，激活该约束。这样我们仍然会和之前一样出现冲突的情况，我们需要将滑块区域视图底部到`warppingView`底部边缘的约束禁用，避免了约束冲突，这样`warppingView`就可以正常缩短高度了。



1.3 实现代码

接下来看看完整代码，在我们控制器的子类中，我们持有3个属性：

- `warppingView`：外部容器视图
- `edgeConstraint`：底部边缘的约束
- `zeroHeightConstraint`：一个存储0高度约束的属性

```
@IBOutlet var warppingView: UIView!
@IBOutlet var edgeConstraint: NSLayoutConstraint!
var zeroHeightConstraint : NSLayoutConstraint!
```

我们创建了按钮点击事件，在响应按钮事件函数中，我们首先要保证`zeroHeightConstraint`已被创建。接着我们还希望这一个事件让视图可以在显示和隐藏间切换，所以我们要对一些约束做禁用和激活操作，做完这些就会得到我们想要的切换效果。

```
@IBAction func toggleDistanceControls(_ sender: Any) {
        if zeroHeightConstraint == nil {
            zeroHeightConstraint =
warppingView.heightAnchor.constraint(equalToConstant: 0)
        }

        let shouldShow = !edgeConstraint.isActive

        if shouldShow {
            zeroHeightConstraint.isActive = false
            edgeConstraint.isActive = true
        }else{
            edgeConstraint.isActive = false
            zeroHeightConstraint.isActive = true
        }
    }
```

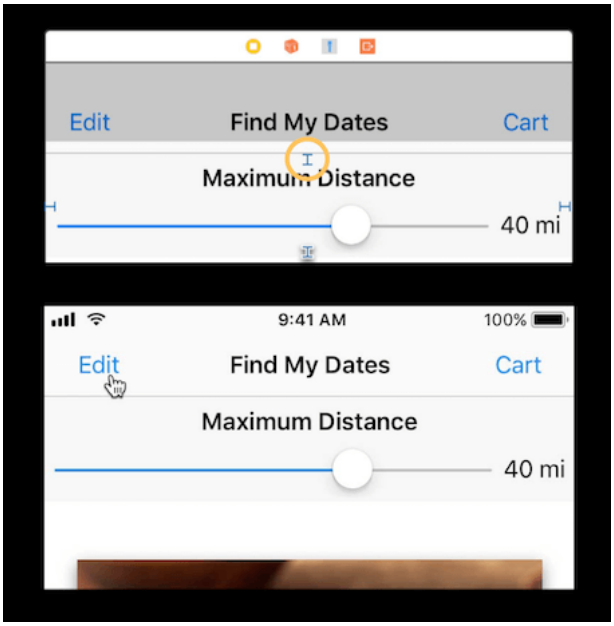
需要特别注意的是，**在激活一个约束前务必先禁用另外一个约束**。在这些简单的切换禁用和激活代码，遵守这一点让我们避免了冲突，如果约束中一旦存在冲突，控制台就会提醒我们：嘿，我检测到这些约束是互相冲突的😂。例如，我们激活了`zeroHeightConstraint`约束，而底部约束`edgeConstraint`还未被禁用，这个时候我们就会看到控制台打印出冲突信息。

1.4 加入动画

加入这些代码重新运行后你会发现我们的界面正确显示和隐藏了，但是我还想为这个过程加上动画，让用户可以看到视图切换过程能够提高用户体验。在这里我们使用`UIView animation block`来实现动画，`UIView animation`将捕捉并且动画化整个过程。

```
UIView.animate(withDuration: 0.25) {
    self.view.layoutIfNeeded()
}
```

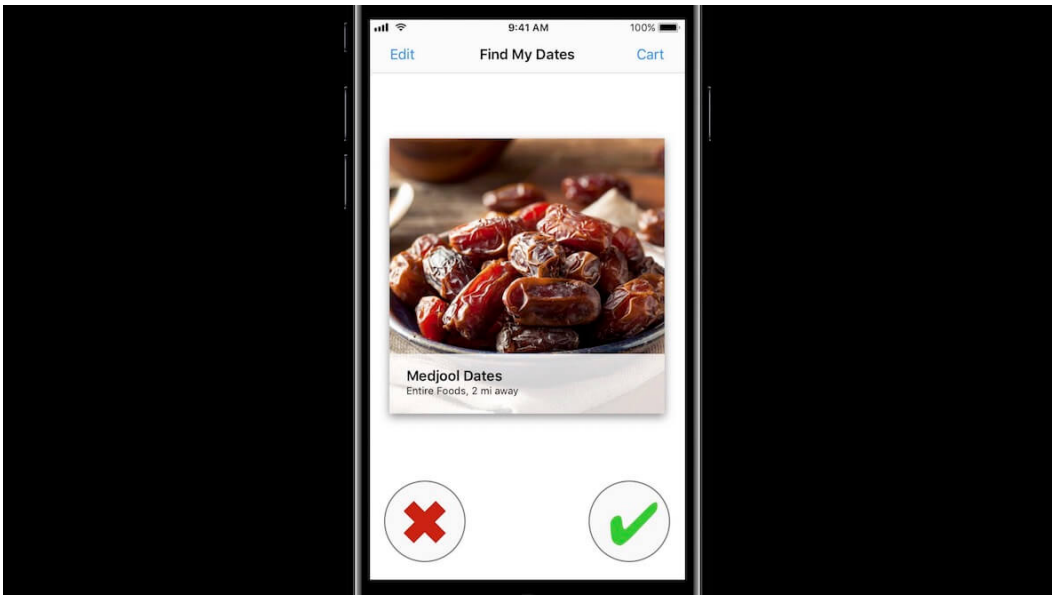
这里得到的动画效果，也并不是我想要的最终效果，我们还需要做最后一点调整，但是这不需要修改我们的代码，我们只需要将底部边缘的约束:`edgeConstraint`属性更换成连接到顶部边缘的约束，改成一个底部对齐的效果。整个动画效果发生改变，我确认这就是我需要的最终效果。



具体效果可以查看我们的Demo(非苹果官方)，通过上面这些内容我们可以知道，怎样通过运行时改变约束来动态调整我们 app 中的布局。

2. 跟踪触摸手势（Tracking touch）

现在我们来看看改变布局的另一种方法，**我保证它既简单又炫酷**。我们将用它来跟踪触摸手势。我们在我们下图的 app 的中央区域有一张卡片，我们希望卡片能随着触摸手势移动，随着靠近边缘的时候，会有一些旋转，一旦你的手离开屏幕，卡片就会弹回屏幕中间。



2.1 frame 饮水知源

通常一个控件在屏幕上的位置由它的 frame 决定，而 frame 又源起何处呢?

- **Layout engine owns frame**。当我们使用`Autolayout`并使用约束控制此视图时，布局引擎将会持有此视图的 frame 。
  - **Value derived from constraints**。frame 的值是从这些约束中计算出来的。
- **transform property offsets from frame**。还有另一个属性会影响视图在屏幕上的位置，那就是 transform ，在 transform 属性源起于 frame 。
- **CGAffineTransform = translation + rotation + scale**。通过`CGAffineTransform`，它可以帮助我们为视图加入平移，旋转和缩放等变换，在从约束中计算出 frame 之后，将其应用在 transform 中。

2.2 加入监听手势

再回到需求上，如果我们想要中间的卡片随着我的手势移动，那我们就要加入一个手势识别器，并且拖线连接到代码中，添加监听手势的方法，在该方法中我们可以访问手势识别器的各种属性。此外还将我们要移动的卡片也通过拖线创建了属性。

```
@IBOutlet weak var cardView: UIImageView!
@IBAction func panCard(_ sender: UIPanGestureRecognizer) {}
```

2.3 加入位移和旋转

接下来我们要通过手势识别器监听用户手势移动，得到位移结果后转换成 transform 应用在`cardView`上。在这里有`transform`函数帮我们进行了位移和轻微的旋转，这个时候，卡片将会随着你的手指移动伴随着轻微的旋转。

```
func transform(for translation: CGPoint) -> CGAffineTransform {
    let moveBy = CGAffineTransform(translationX:translation.x, y:
translation.y)
    let rotation = -sin(translation.x/(cardView.frame.width * 4.0))
    return moveBy.rotated(by: rotation)
}
```

2.4 位置还原

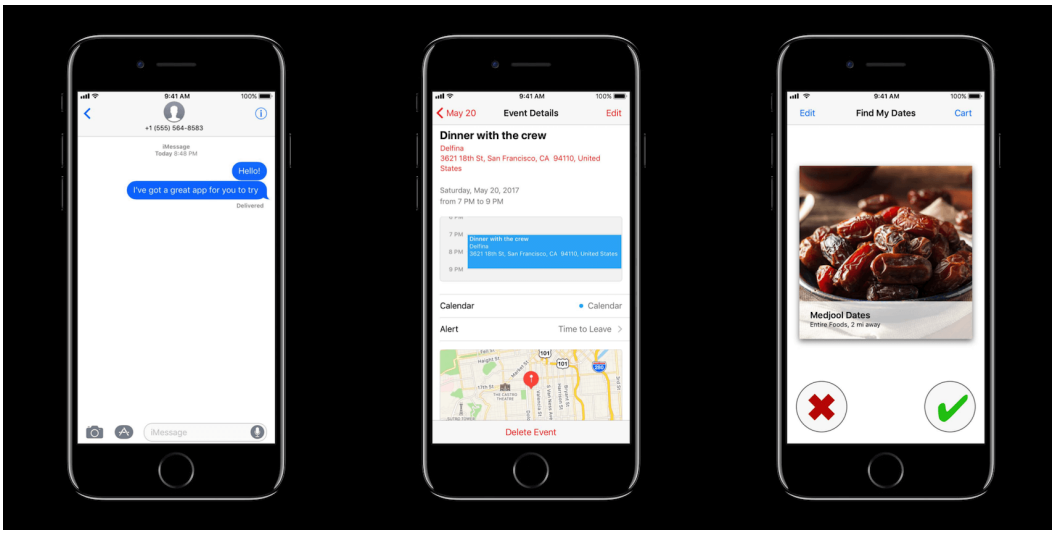
但是当我放开手指时，卡片停留在原位，没有回到屏幕中央，因为我们并没有去重置卡片的 transform 属性。当我再次触摸并移动，我们会看到它会回到原来的位置，这是因为我们开始了一个新的位移，新的位移关联的原来的 frame 。总之这不是我想要的效果，我希望在用户手指离开屏幕后，卡片能够立即回到屏幕中间的位置。我们可以通过手势识别器的状态来做到这一点。我们在`state`为`end`的时候，将重置 transform 并且加入弹簧动画。加入这部分代码运行 app ，在我松开卡片后它会弹回中间的位置。

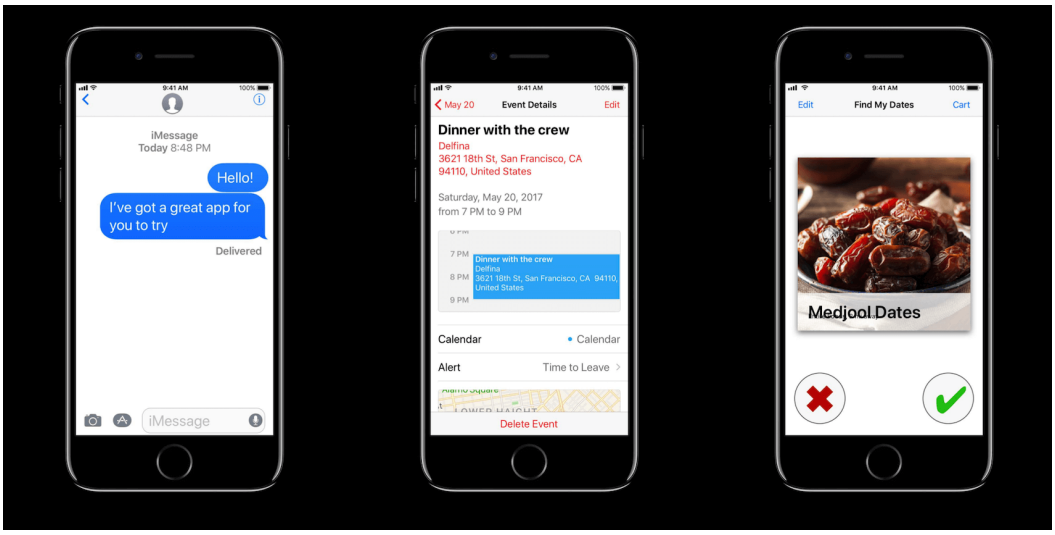
```
@IBAction func panCard(_ sender: UIPanGestureRecognizer) {
    switch sender.state {
    case .changed:
        let translation = sender.translation(in: view)
        cardView.transform = transform(for: translation)
    case .ended:
        UIView.animate(withDuration: 0.5, delay: 0, usingSpringWithDamping:
0.4,
initialSpringVelocity: 1.0, options: [], animations: {
            self.cardView.transform = .identity
        }, completion: nil)
    default:
        break;
    }
}
```

简单的几行代码，实现了一个很有意思的交互效果，在这些内容里面，我们可以看到 frame 它不仅是通过约束来计算出，也会受到 transform 的影响，视图的 frame 中蕴含多种属性的组合效果。

3. 动态字体（Dynamic type）

`Dynamic type`是 iOS 中提供了一组文本样式，文本样式包含了标题、副标题、正文等样式，而且用户可以控制这些样式字体大小的技术。在 iPhone 的短信消息中，如果用户喜欢大一点的字体，通过在设置中进行设置后，我们将看到消息界面会有所变化，字体变大了，消息气泡和输入文本也变大了。日历和其他一些地方也具备类似的功能。

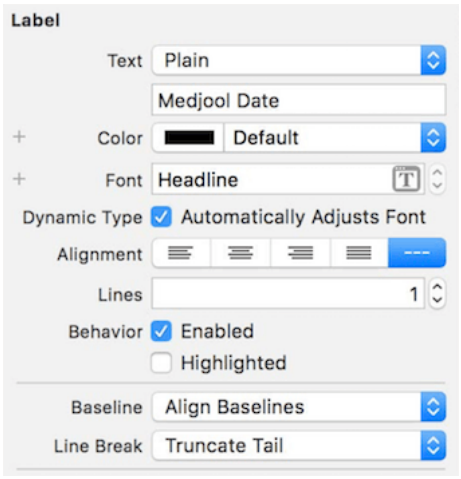




相信这个时候你一定会好奇，如何才能在我们自己的 app 实现这个功能呢？另外在调整字体大小时，如果不相应地调整我们的布局，容易造成视图重叠，对用户体验来说是非常不好的。幸运的是，`Autolayout`可以很轻松地帮我们搞定这个问题。

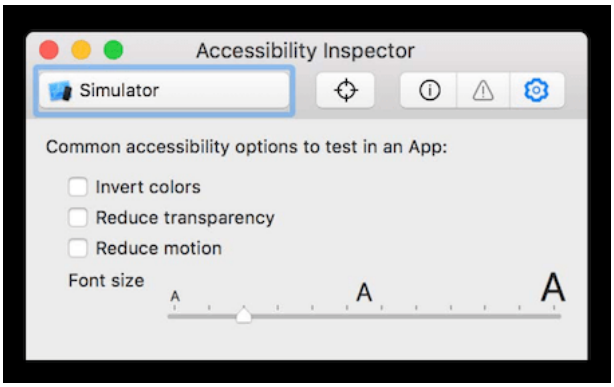
3.1 支持 Dynamic Type

所以赶紧让我们来看看是怎么实现的吧，打开 IB 界面，选中你要支持 Dynamic Type 的 label ，查看 label 的属性，勾选`automatically adjust font`，如果你眼睛够敏锐的话，你能看到上面出现了一个警告，原因是因为`automatically adjust font`属性生效，该属性要求 label 设置指定的文本样式。这里要将系统默认字体更换为`caption one`，该样式和默认字体12号大小相对应。



3.2 通过 Accessibility Inspector 改变字体大小

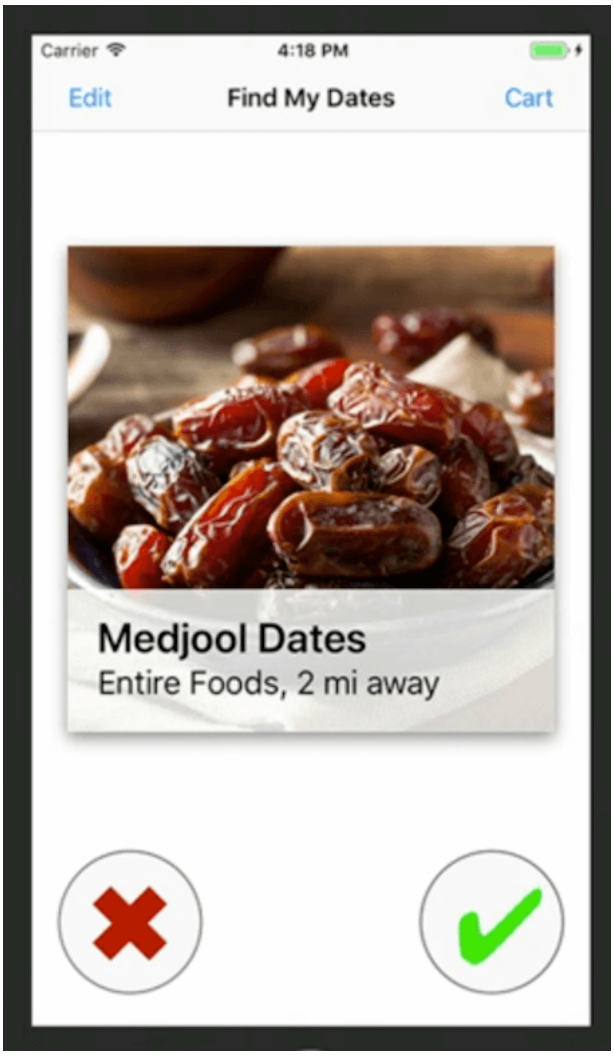
设置好这些再重新运行，你会发现和之前并没有什么不同，这是因为我们还没有改变文本样式的大小，我们可以在设置中调整字体大小，但是这种方式需要来回切换不够直观，所有我们用另外一种方法，点击顶部导航条`Xcode`->`Open Develop Tool`->`Accessibility Inspector`->`target`切换至模拟器->选择设置标签，就可以看到修改字体大小的滑块了。这个时候滑动滑块就能看到我们的 label 字体在实时地改变。如果我们连接了 iPhone ，我们也可以将`target`切换至我们的 iPhone 。



3.3 根据字体大小动态调整布局

在下图中可以看到，如果我们把字体调整到非常大的时候，我们的 label 就会发生重叠，接下来我们要解决这个问题。

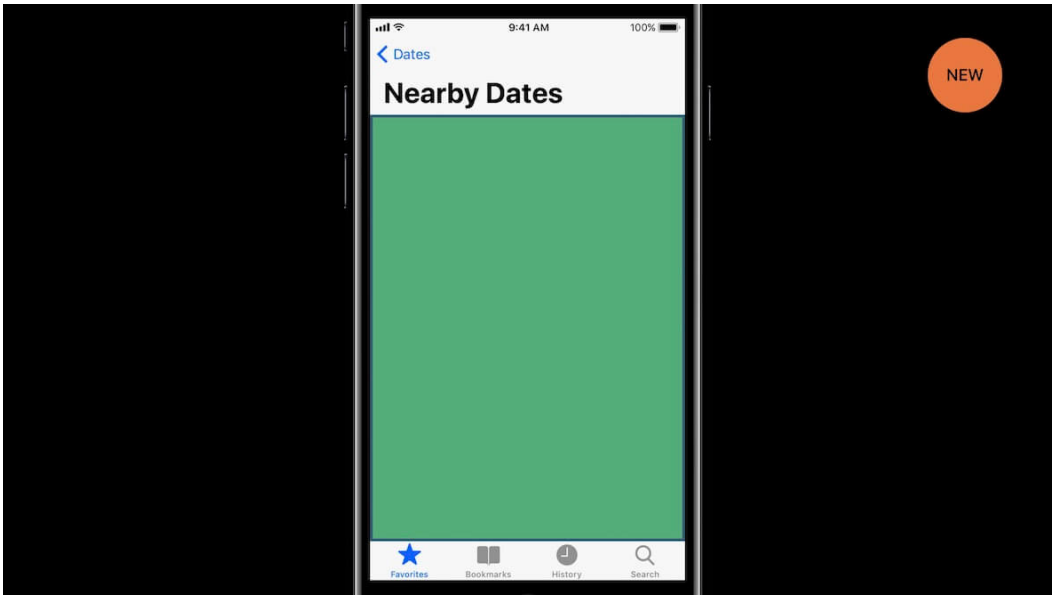
首先我们创建了一个文本区域，这个文本区域会随着字体变大而增高，所以我们只要将底部 label 被限制在底部，顶部 label 被限制在顶部，再在两者之间添加了一个垂直间距约束，使两个标签始终保持足够垂直间距，避免使用固定高度约束，这样 label 的高度会随着字体变大而增高，接着 label 又会将文本区域给撑高。这个时候可以打开`Accessibility Inspector`来测试改变我们 app 的字体大小，你会发现我们的文本区域会随着字体的变化而改变高度。



在这中间我们并不需要做很多处理，就能实现这样一个非常实用的功能，特别是当你有一些需要读者阅读文字的的需求，相信 Dynamic Type 能够帮助你，让你的 app 更加强大。

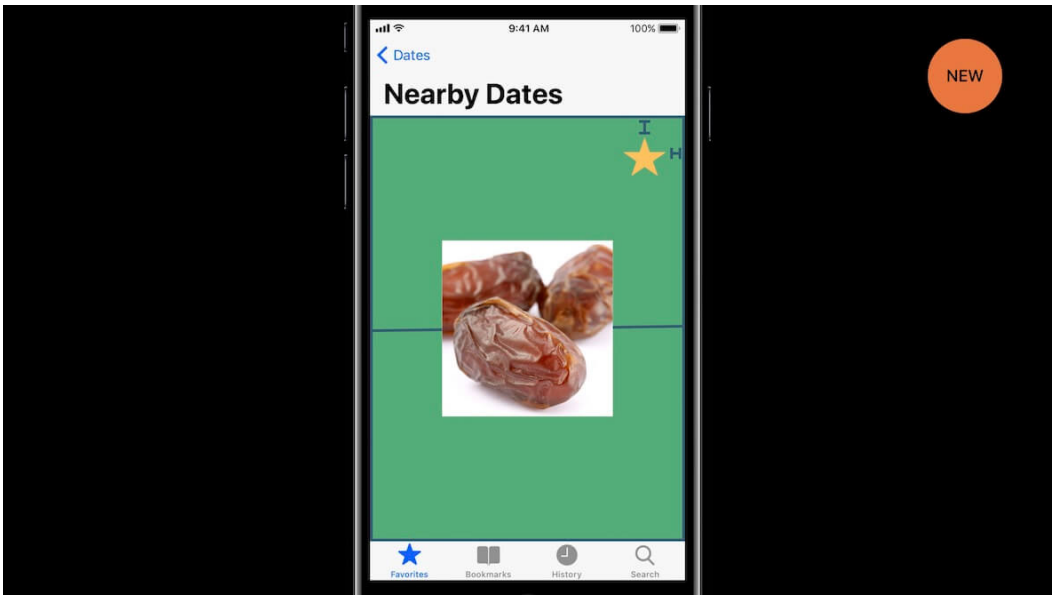
4. 安全区（Safe area）

接下来要介绍的内容在之后你可能会频繁使用到，所以一定要搬好小板凳认真看。当你新建了一个控制器，控制器有一个导航条和一个底部标签栏，如何保证你的内容主体不被导航条和标签栏遮挡？可能你已经听说过在 iOS 11 上有了新的 layout guide，称之为`Safe Area Layout Guide`。

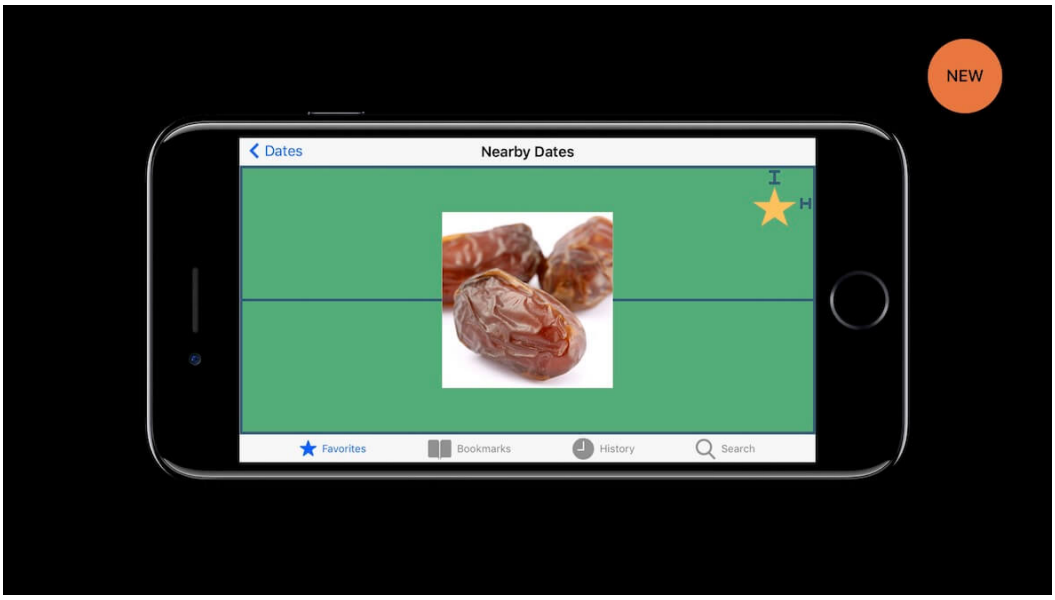


4.1 Safe Area Layout Guide 更易使用

这是`UIView`的新特性，它适用于自动布局，它是夹在导航条和标签栏之间的一个矩形，在这个矩形区域中你可以放心地为你的视图添加约束。在这之前，你可能不得不使用`UIViewController`的`Top Layer Guide`和`Bottom Layer Guide`，现在在`Safe Area`中这些都已经通通被丢弃了。

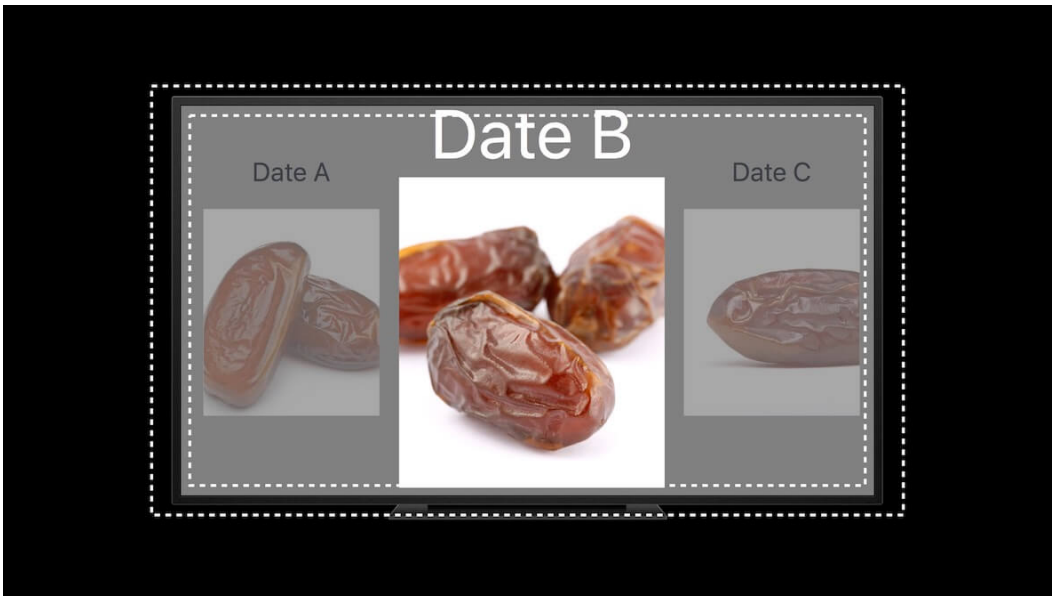


`Safe Area Layout Guide`使用起来更加简单，也更容易理解，如同字面意思，可以安全的让你的视图安全地呆在导航条和标签栏中间，不被遮挡，不管是尺寸的变化和屏幕旋转，它都会自动做相应地调整。

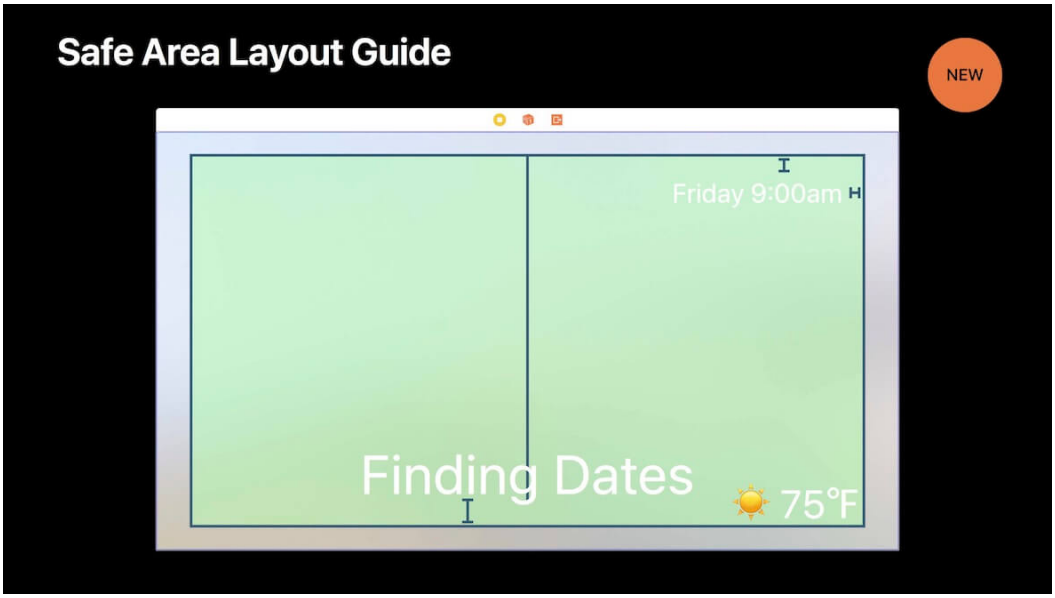


4.2 如何使用 Safe Area Layout Guide

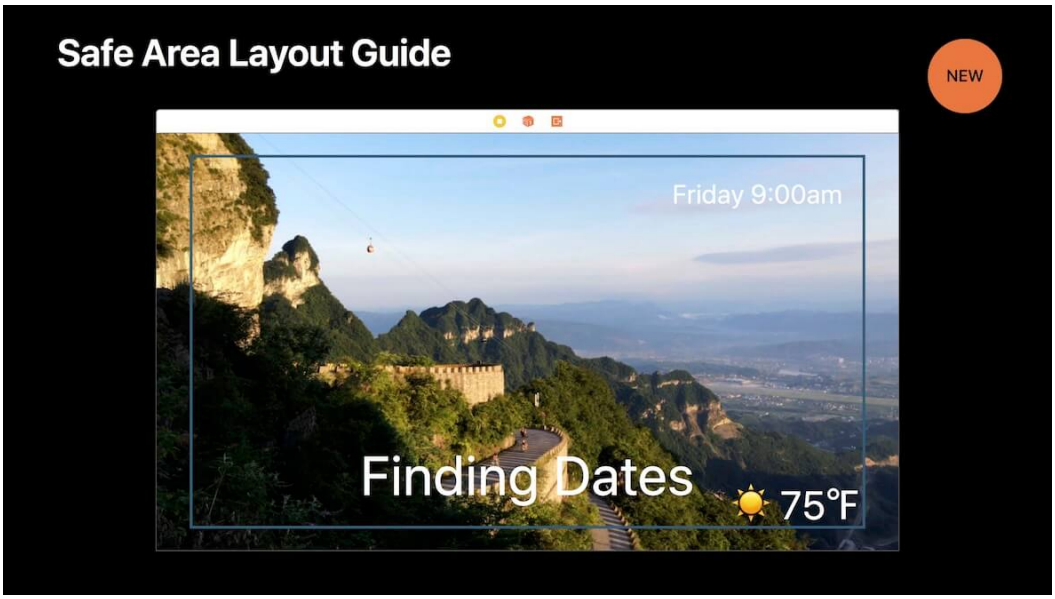
`Safe Area`也适用在 tvOS 上。如果要将你的 app 和内容放到 tvOS 上，你可能会遇到各种各样的尺寸的屏幕，在某些情况如下图，我们顶部的标题太靠近顶部边缘，可能因此被遮挡掉一部分。



这个时候我们要调整我们的内容，让它处于`Safe Area`之中。`Safe Area`代表`storyboard`中的这块浅绿色的区域，你只要将你视图中的约束设置到`Safe Area`中，那它就安全了。

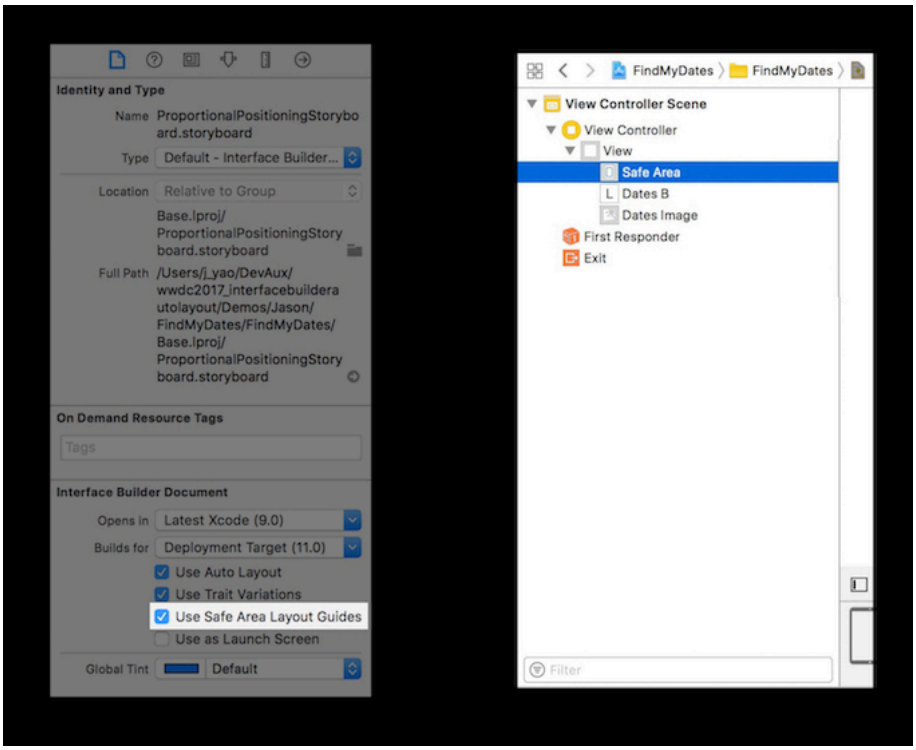


然后用一张的美丽背景图像填充剩余视图空间，妈妈在也不用担心我们的内容被导航条和标签栏挡住了。如下图，它们只会听话地呆在深色矩形方框内。



4.3 开启 Safe Area Layout Guide

开启`Safe Area Layout Guide`也十分简单，打开我们的`storyboard`，进入 file inspector 标签页，然后找到`Use Safe Area Layout Guides`并且勾选上。你会发现每个控制器中都会出现一个`Safe Area`视图，然后你就可以像其他视图一样，将约束连向它。



`Safe Area Layout Guide`是 UIView 的新特性，在之前的版本中顶部和底部的`Layout Guides`之间的矩形区域将与新的`Safe Area`相匹配，他们可以互相转换，如果在 iOS 11 的故事板中启用`Safe Area`，在你选中`afe Area Layout Guides`勾选框时 Xcode 将会自动升级你的约束。总而言之，在 Xcode 9 的故事板中使用`Safe Area Layout Guide`，将向下兼容 iOS 老版本的。

5. 比例定位（Proportional positioning）

接下来我们要谈一谈，关于如何将一个视图定位在其 superview 的布局技术，我们将其称之为 Proportional positioning，即按 **比例定位** 。在安卓的布局技术中也有类似的功能，它的应用面广泛且实用，相信未来的开发中一定会频繁使用到。

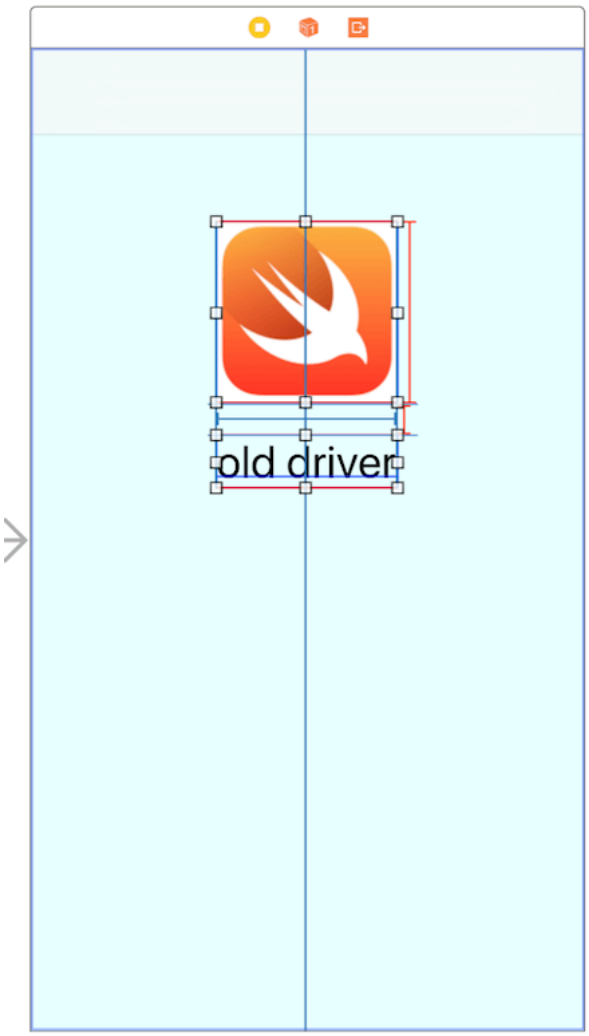
5.1 比例布局

假设现在我有一个需求，要将我们 app 中的卡片高度定位在其 superview 高度的70%。也许你会有几种方式可以实现上述需求，但是现在我要用一个最直接的方式来实现它，便是我现在要介绍得的使用 spacerview 的方法。

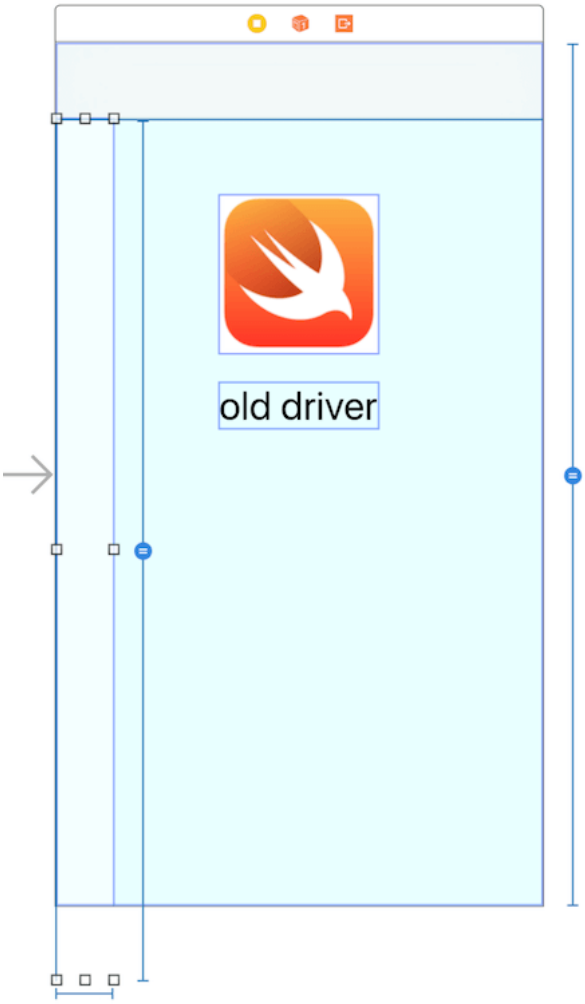
从对象库拖出一个视图，只是一个普通的`UIView`。为它添加约束后设置隐藏，这样就不会渲染它，让它做一个安静的美男子，这样它就成为你需要定位的视图的参照物。而且这种技术也可以组合使用，灵活搭配，这里有另一个例子，我有一个场景，要遵守1/5，2/5和2/5的比例，然后他们以这些比例填充满整个屏幕。下面让我们来看看如何做到的。

5.2 构建 SpacerView

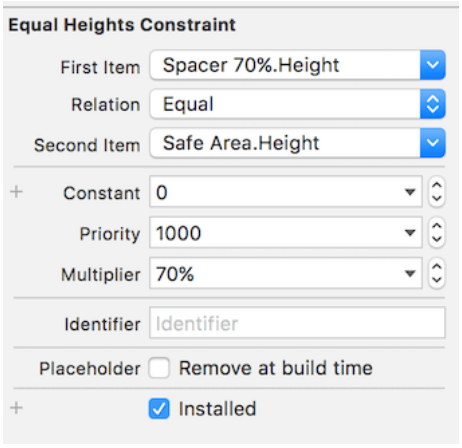
如下图中我们已经有一个基本的布局。我有一个 label 和一个 image ,已经添加了基本约束。 当我选中它们，如果你仔细看，你会注意到它左边和右边的约束是蓝色的，但顶部和底部是红色的。这意味着我们还需要添加一些约束来定位。无论何时在 Interface Builder 画布中看到红色，那只可能是两种情况，要么你的约束太少，位置是不确定的，或者设置了太多的约束，其中一部分是冲突的。



我知道是因为我没有对垂直方向位置进行固定。所以接下来我们要通过创建 spacerview 来实现。拖一个`UIView`出来。首先我们将其隐藏，这样不会浪费性能进行绘制，我们为其添加好上方、左侧以及宽度的约束后，我们还没有设置其高度约束，我们要为其设置等高约束，使其高度与 superview 高度相，如下图效果。

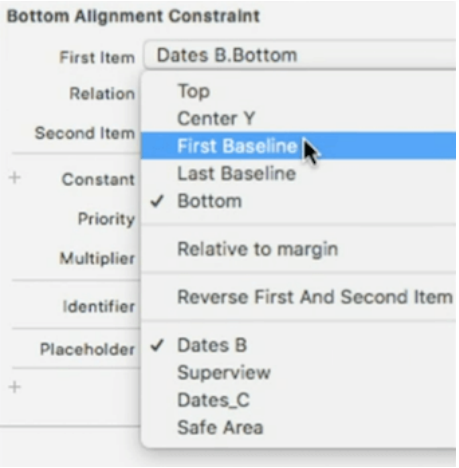


接着让我们查看等高约束的属性，修改比例为70%，设置成功后你会发现`spacerview`的高度已经缩减了，下一步设置 Second Item 即比例参考对象视图为 `Safe Area`，这样我们的`spacerview`就已经设置好了。

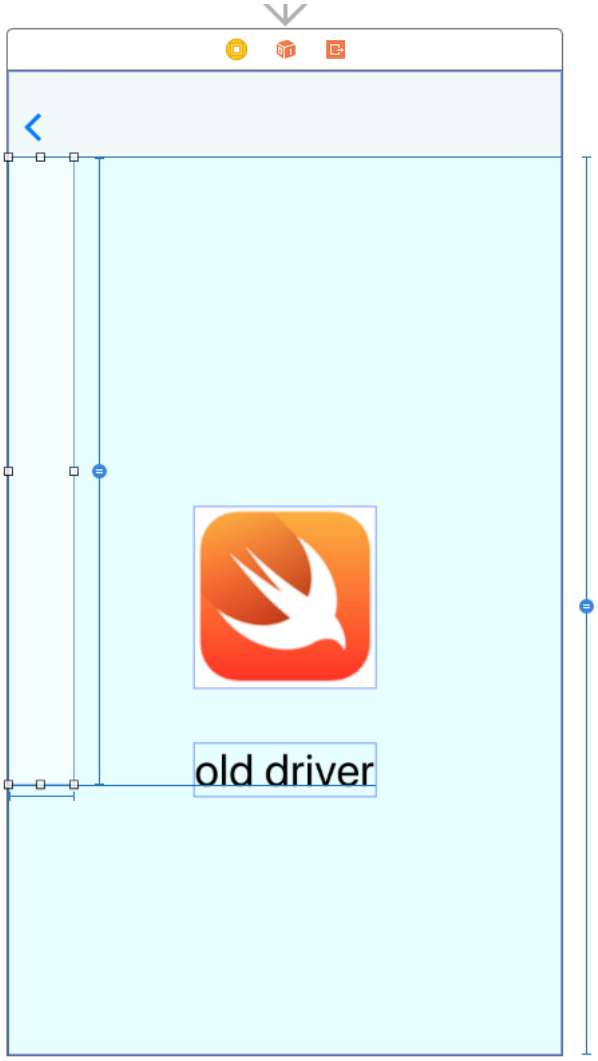


5.3 对齐到 Baseline

现在我们要将我们卡片视图底部与`spacerview`底部对齐，所以我们添加了底部对齐的约束，如果我想要是`spacerview`与我们的卡片文案的`baseline`对齐怎么办？选中约束后转到属性检查器，选择 FirstItem 选项，选中`First Baseline`即可。



重新运行后得到了我想要的效果。

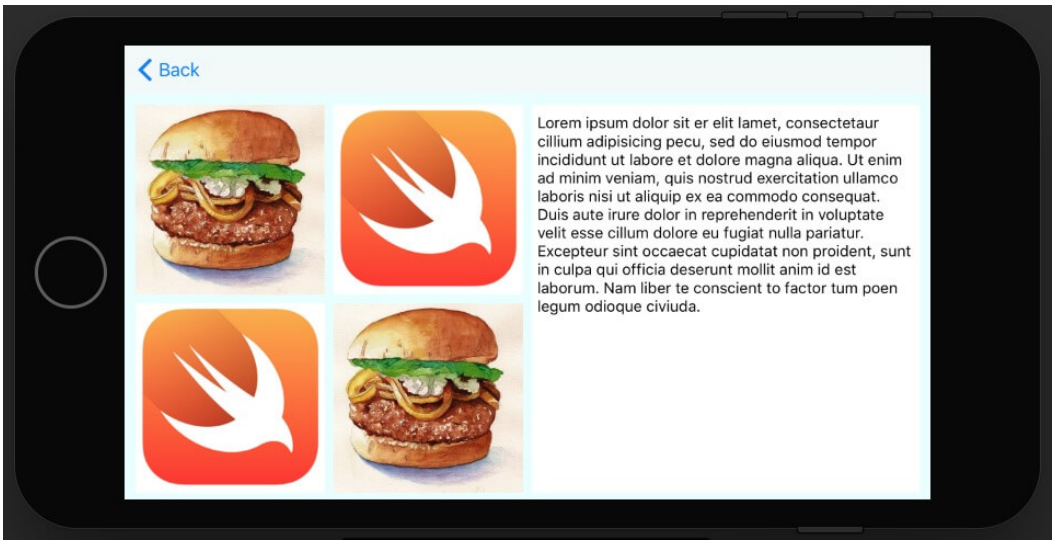


所以当你需要在 Interface Builder 中使用这个比例定位技术时，使用`spacerview`能够帮助你达到期望。**但是一定要将这些视图标记为隐藏**，使它们不会被渲染，但又能协助你进行布局，使你能够定位你的内容。如果你是用编程方式进行布局，可以使用`UILayout Guide`来完成，你可以将其用作等效于`spacerviews`。

6. Stack view 自适应布局（Stack view adaptive layout）

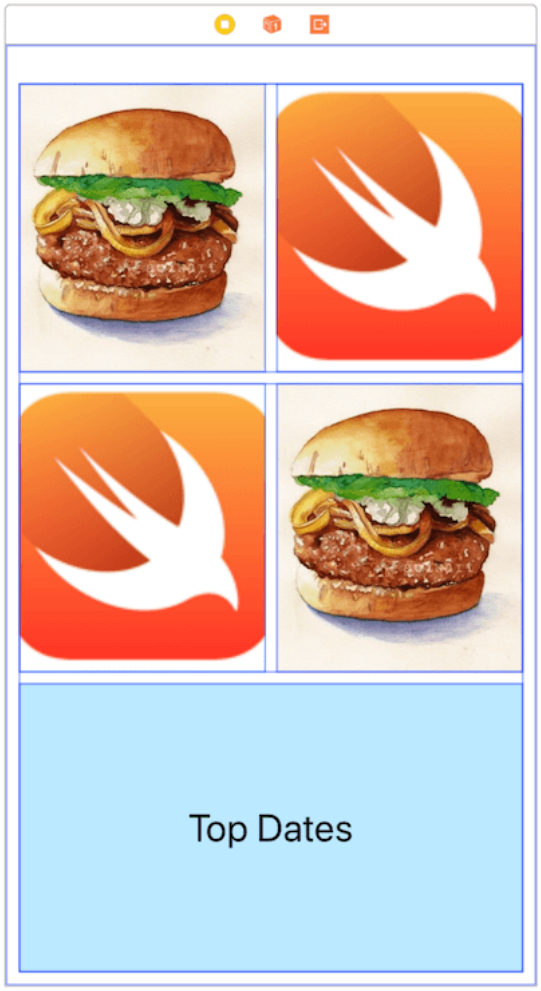
让我们一起来看看最后一种我们要布局的视图，如下图，你能看到 app 中展示了一个自适应布局的页面，上方是一个4x4的网格排列，底部有一个 label 。

当我旋转手机的时候，出现了一些不一样的东西。它仍然会显示一个4x4的网格，但它有一个文本视图出现在右边的位置。

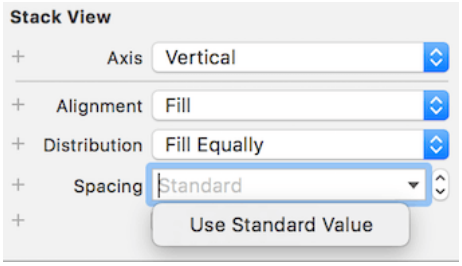


6.1 竖屏布局

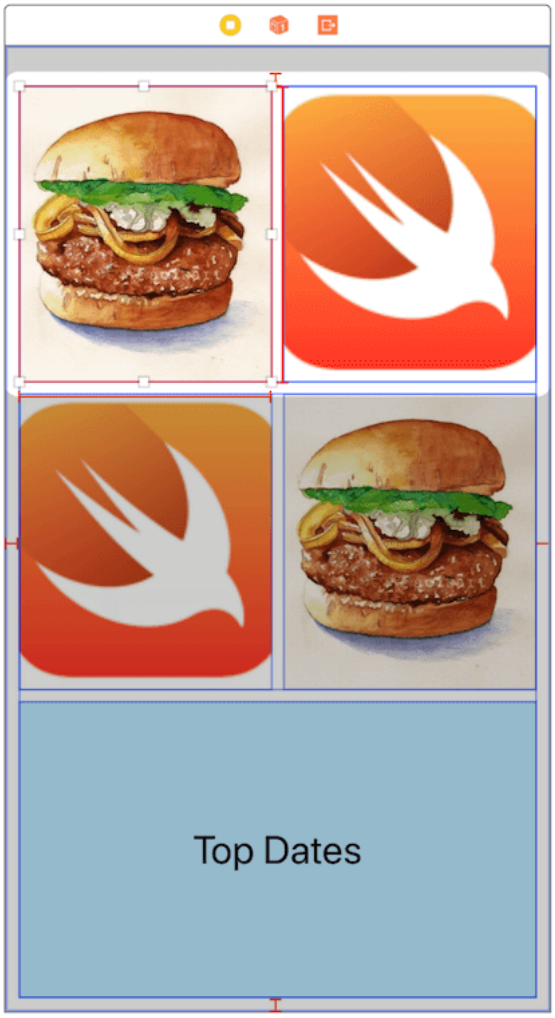
这一切是怎么做到的呢？让我们来看看如何对 Interface Builder 中的 stackview 进行自适应布局。首先看下图中最外层是一个垂直的 stackview ，从上到下分成三行，第一行和第二行都是包含两张图片的水平 stackview ，第三行是一个 label 。就如你看到的，他们高度是相等的，我们可以通过 **Alignment**，**Distribution**，和 **Spacing** 等属性来进行调整，以达到你想要的布局。 stackview 有一个非常赞的地方，就是它能帮你管理被包含的视图的约束，这样你只要添加很少的约束。



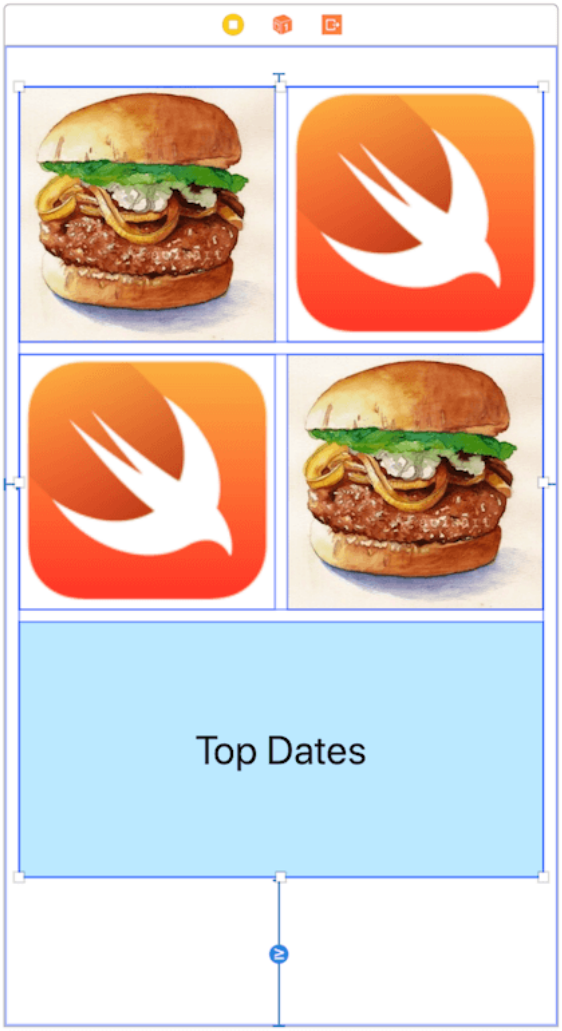
接下来让我们选中所有的 stackview ，在`Distribution`选项中选择`fill equally`实现平均分布，在`Spacing`选项中我们可以手动输入我们想要的间距，另外系统也提供了标准间距选项给我们，点击输入框右边的倒三角就会出现一个`Use Standard Value`选项，直接选中即可。



下一步我要确保这些图像是正方形的，我们直接选中第一张图片，为其添加一个宽高比为1:1的约束，添加后你会发现出现一些冲突，这是因为在满足填充满整个屏幕和三行平均分布的同时，无法保证图片比例达到1:1。

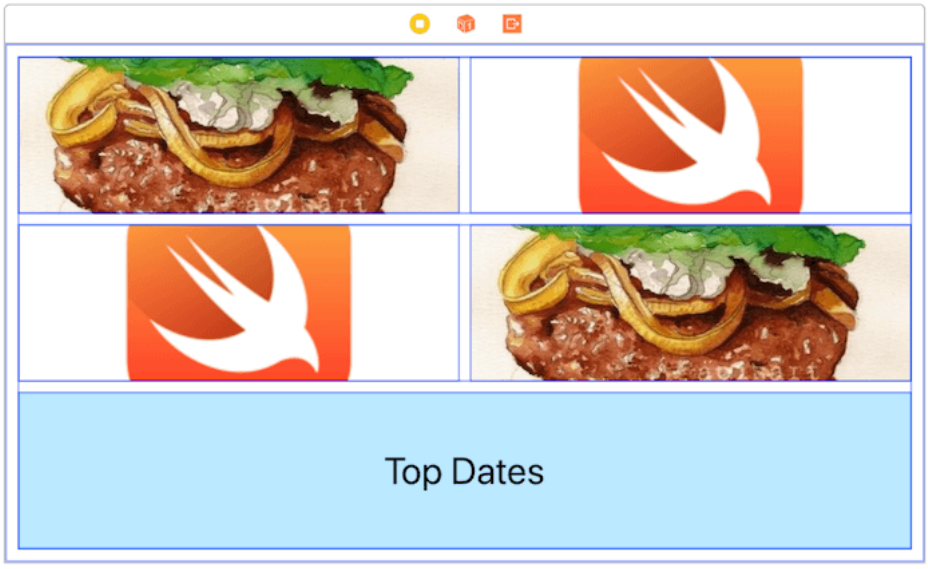


所以我们要做一些改变，我们将 stackview 到底部的固定约束修改成大于等于，这样出现的冲突就解决了，也达到了我想要的效果。

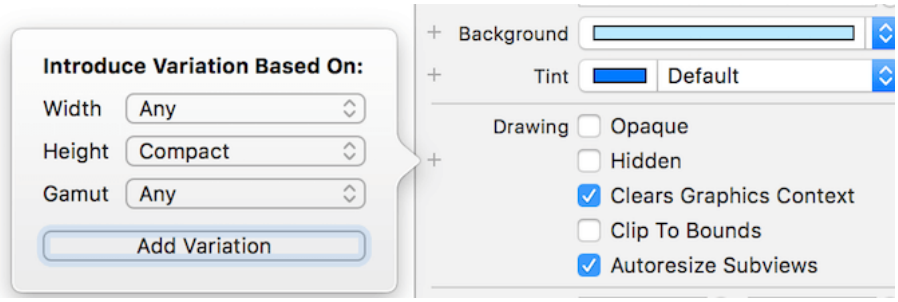


6.2 横屏布局

当我们将设备旋转到横屏状态，我们预期的效果是在右边有一个 textview ，而底部并没有 label ，为了更接近我们预期效果我们需要把底部的 label 先隐藏。我们要如何才能做到在竖屏中显示，在横屏中隐藏呢?

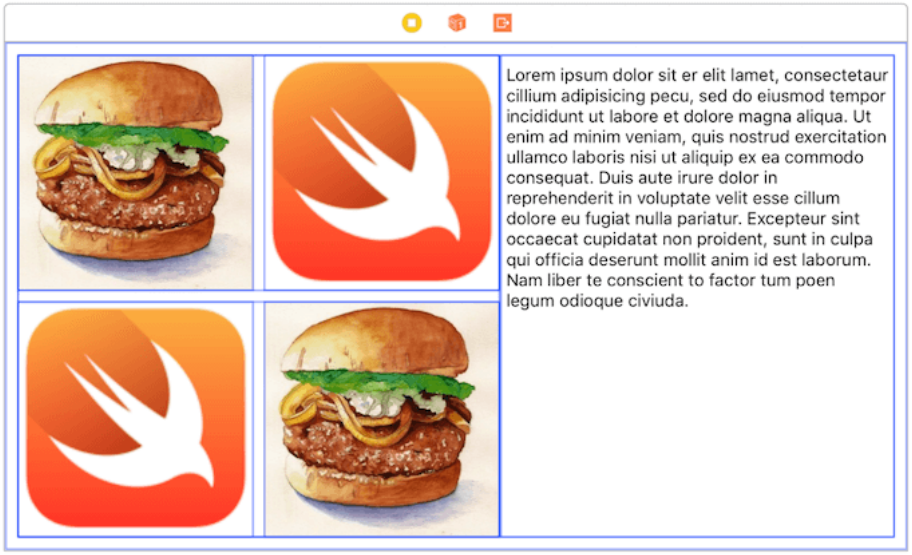


在全新的 Xcode9 中的隐藏属性，可以为不同`size class`分别设置显示或隐藏。转到 label 的`hidden`属性，你会发现勾选按钮左边有一个加号，它让这一切变得轻松简单。点击后在弹出的界面中`Width`选择`any`，在横屏的时候，`Height`选择`compact`，因为在横屏的时候它的高度是紧凑的。

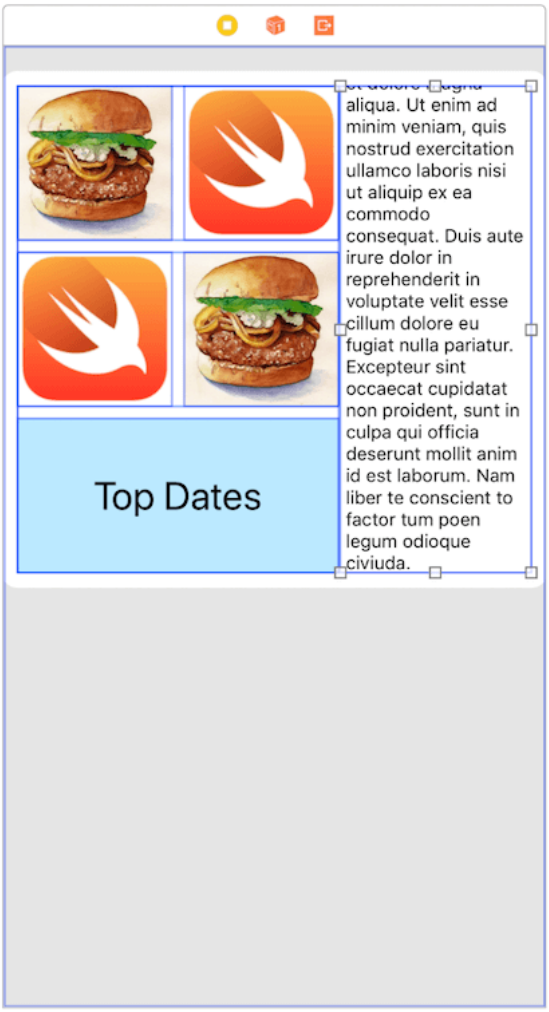


做完这些点击`add variation`，并且在`hidden`属性下面找到刚刚设置的隐藏属性并且勾选中它，你会发现 label 被隐藏了，如果你切换成竖屏，又会显示出来。

接下来继续添加一个 textview ，为了做到这点，我们要在最外层套一个水平排列的 stackview ，然后将 textview 加入 stackview 中，并且为新建的 stackview 添加约束，其中底部约束就如同之前设置为大于等于。这样就得到了我们横屏中需要的效果。



当我们切换到竖屏时 textview 仍然显示了，我们要在竖屏时隐藏它，就像之前一样转到隐藏菜单，并添加一个变量，`Width`选择`any`，`Height`选择`Regular`，然后将其标记为隐藏，这样在竖屏时, textview 就不再显示了，就此达到了我们期望的效果。



在使用 stackview 的时候，我们可以使用`Alignment`，`Distribution`，`Spacing`这些属性帮助我们布局。还有嵌套使用，只需要加入很少的约束，我们仅仅需要在你对宽高比例有要求的时候，通过宽高比例约束来获得我们想要的比例。令人惊喜的是，Xcode 9 中的隐藏属性是可分级的，它非常适合与 stackview 搭配使用，而且随着`size class`的变化的隐藏属性是向下兼容的。

总结

到这里我们已经看完了`Autolayout`相关的的六种技术，在你构建 app 的时候有了更多的布局手段，这些技术能够使你的界面看起来非常美观，结构清晰，并且自适应布局，在日常开发中会经常使用到。我已经迫不及待地想看到更多人使用这些技术了。

Demo

GitHub：FindMyDates

参考

- 视频地址：WWDC 2017 Session 412 - Auto Layout Techniques in Interface Builder
- PDF地址：WWDC 2017 Session 412 - Auto Layout Techniques in Interface Builder

What's New in LLVM

Xcode 9中LLVM编译器主要增加了如下几个方面的新功能。

- OC/C/C++ Api可用性检查
- 静态分析检查
- 新的警告类型
- C++重构
- C++17的新特性
- 链接时间优化

OC/C/C++ Api可用性检查

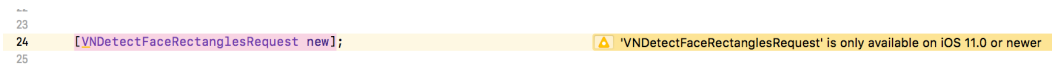
OC Api可用性检查

当使用新的api时，需要注意在老的系统上面运行可能会导致崩溃的问题，所以需要判断API运行的可用性。在`Swift`中，可以使用`#availabl`来判断，现在`OC`也可以了。

比如`Base SDK`为`iOS 11.0`，`iOS Deployment Target`为`iOS 9.0`,那么在使用`iOS 11`新增的Api时，比如:

```
[VNDetectFaceRectanglesRequest new]
```

就会提示警告:



那么可以通过运行时判断系统版本:

```
if(@available(iOS 11, *)){
    [VNDetectFaceRectanglesRequest new];
}else{

}
```

同样在编写SDK或者对外的接口的时候，某些方法或者类也只能在某个以上的版本才能使用，这个时候可以在方法和类上加上对应的可用版本号。

```
@interface MyAlbumController : UIViewController

-(void)showFaces API_AVAILABLE(ios(11.0));

@end
```

```
API_AVAILABLE(ios(11.0))
@interface MyAlbumController : UIViewController

-(void)showFaces;

@end
```

这样在调用者使用的时候，如果`iOS Deployment Target`低于指定的版本就会出现警告。

C/C++ Api可用性检查

在`C/C++`也可以使用`__builtin_available`在运行时检测可用性。

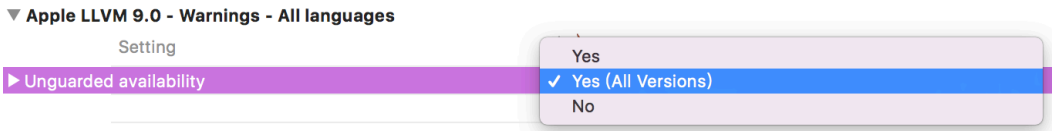
```
if(__builtin_available(iOS 11, macOS 10.13, *)){
    CFNewAPIOniOS11();
}
```

或者定义对外接口的可用版本:

```
void myFunctionForiOS11OrNewer(int i) API_AVAILABLE(ios(11.0), macos(10.13));
```

```
class API_AVAILABLE(ios(11.0),macos(10.13)) MyClassForiOS11OrNewer{

};
```

在编译选项中可以设置是否检测、检测所有版本。



新的静态分析检查

新版增加了三个新的检测:

- NSNumber和CFNumberRef中的比检测
- dispatch_once实例唯一变量检测
- NSMutable copy属性检测

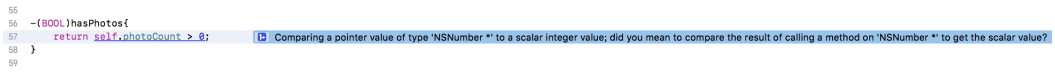
NSNumber和CFNumberRef中的比较

如果在程序中，写出下面的代码:

```
@property NSNumber *photoCount;

-(BOOL)hasPhotos{
    return self.photoCount > 0;
}
```

其实这个时候语义就不太明确，是要比较`photoCount`是不是为nil，还是比较里面的数字是不是大于0。

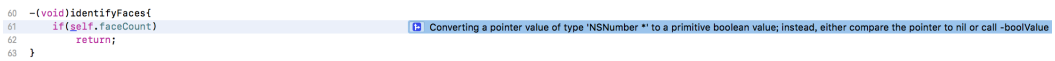
所以使用`Xcode`的分析后就会报错警告:



这是改成如下代码，整数和整数的比较。

```
-(BOOL)hasPhotos{
    return self.photoCount.integerValue > 0;
}
```

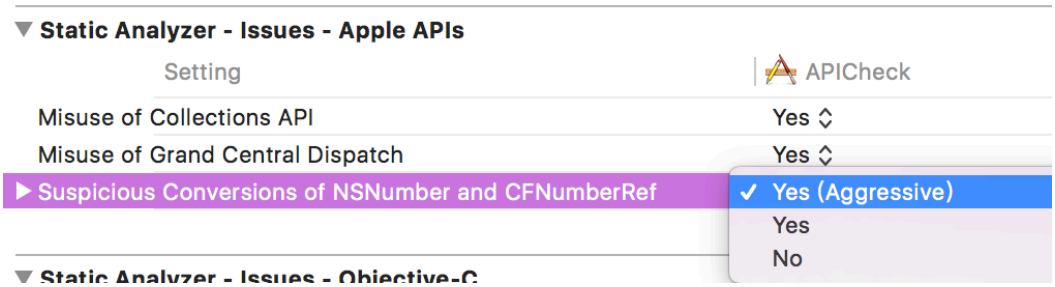
同样，如下的代码也会有同样的问题。



需要明确程序的目的，比如这里是想判断`faceCount`是否为nil。

```
-(void)identifyFaces{
    if(self.faceCount != nil)
        return;
}
```

这可以在编译设置里面控制。



dispatch_once

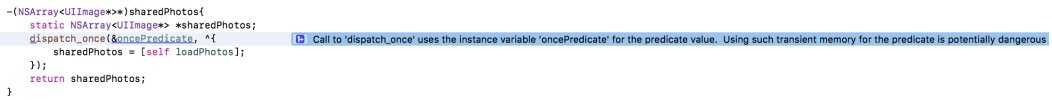
在代码里面经常会用到`dispatch_once`保证某段代码只执行一次，来初始化获取唯一的实例。

```
+(NSArray<UIImage*>*)sharedPhotos{
    static NSArray<UIImage*> *sharedPhotos;
    static dispatch_once_t oncePredicate;
    dispatch_once(&oncePredicate, ^{
        sharedPhotos = [self loadPhotos];
    });
    return sharedPhotos;
}
```

这里很重要的一点就是`oncePredicate`必须是全局的或者静态变量，如果下是下面这种就不行了。

```
@implementation Album{
    dispatch_once_t oncePredicate;
}

dispatch_once(&oncePredicate, ^{
    self.photos = [self loadPhotos];
});
```

因为变量`oncePredicate`是可变的，获取到的地址也会不一样，所以Xcode会报如下警告:



也可以通过锁来保证同一个时间只有一个线程执行里面的代码。

```
@implementation Album{
    NSLock *photosLock;
}

[photosLock lock];
if(self.photos == nil){
    self.photos = [self loadPhotos];
}
[photosLock unlock];
```

copy

如果把`NSMutableArray`设为`copy`，如下:

```
@property (copy) NSMutableArray<UIImage*> *photos;

-(void)replaceWithStockPhoto:(UIImage*)stockPhoto{
    self.photos = [NSMutableArray<UIImage*> new];
    [self.photos addObject:stockPhoto];
}
```

当调用`replaceWithStockPhoto:`的时候，就会报错:

```
-[__NSArray0 addObject:]: unrecognized selector sent to instance
0x610000009ad0
```

因为给`photos`赋值的时候会调用`copy`变成不可变的`NSArray`，所以再去调用`addObject`的时候就会报错。

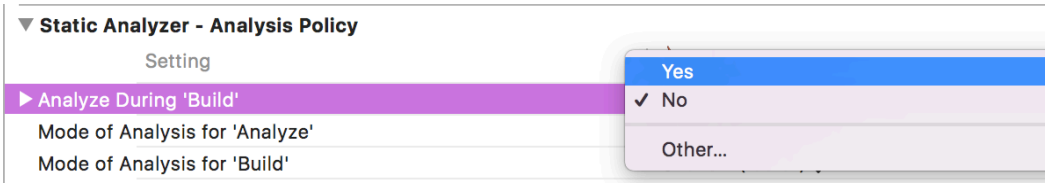
Xcode静态分析也会提醒:

当然可以重写`setPhotos:`:

```
@synthesize photos = _photos;

-(void)setPhotos:(NSMutableArray<UIImage *> *)photos{
    _photos = [photos mutableCopy];
}
```

不过最好还是不要用`copy`。

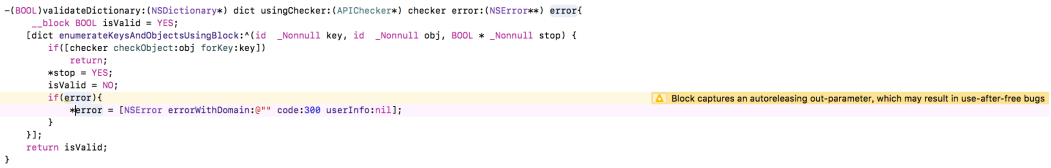
在`Xcode`编译设置中还可以开启`Build`的时候做静态分析:



新的警告类型

Block里面初始化外部参数是不安全的

看下面一段代码:



如果这样在`block`里面去初始化`error`就会出现上面的警告，因为`enumerateKeysAndObjectsUsingBlock`这个函数里面加了`@autoreleasepool`，会导致`error`被释放，外部再使用就会报错。

第一种解决方法是强引用`error`。

```
-(BOOL)validateDictionary:(NSDictionary*) dict usingChecker:(APIChecker*)
checker error:(NSError*__strong*) error{

}
```

另外一种解决方案，是在局部申明一个`NSError`对象指针，指向初始化出来的对象。

```
-(BOOL)validateDictionary:(NSDictionary*) dict usingChecker:(APIChecker*)
checker error:(NSError**) error{
    __block BOOL isValid = YES;
    __block NSError *strongError = nil;
    [dict enumerateKeysAndObjectsUsingBlock:^(id  _Nonnull key, id  _Nonnull
obj, BOOL * _Nonnull stop) {
        if([checker checkObject:obj forKey:key])
            return;
        *stop = YES;
        isValid = NO;
        strongError = [NSError errorWithDomain:@"" code:300 userInfo:nil];
    }];
    if(error){
        *error = strongError;
    }
    return isValid;
}
```

无参数方法定义

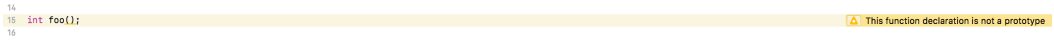
通过以下定义个方法:

```
int foo();
```

这样定义后，然后在程序中带参数去调用也只是警告。

```
foo();
foo(1,2,3);
foo(&x, y);
```

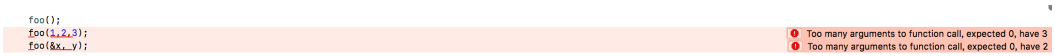
当然定义方法时也会有警告。



这个时候需要指定参数为`void`。

```
int foo(void)
```

发现调用的地方就报错了。

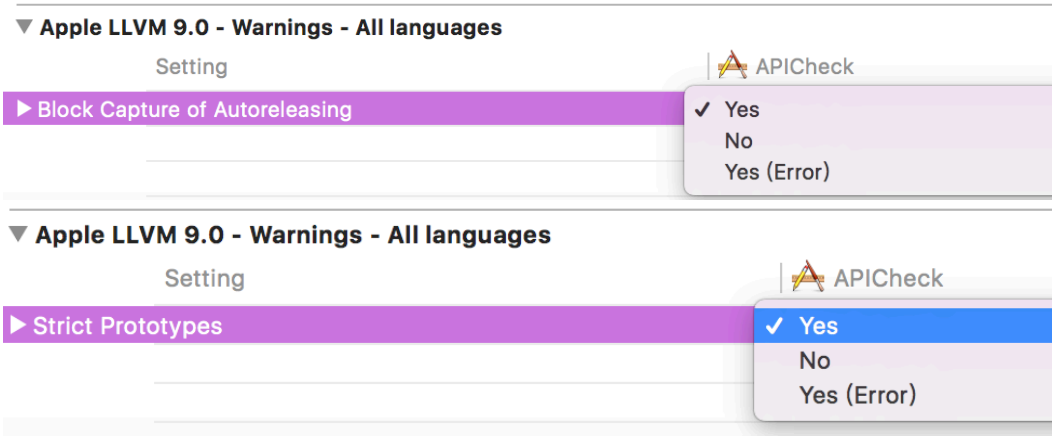


同样定义`block`也是如此:

需要指定参数为`void`:

```
int takesBlock(int (^block)(void));
-(void)takesBlock:(int (^)(void))block;
```

都可以在编译设置里面设置开启或转成error。



C++重构

这一部分讲的使用Xcode提供的`rename`,`extract`功能对LLVM源码上的一些方法名字修改，方法抽取，方法增加等等，这里就不做详细讲解了。

C++17的新特性

使用auto [...]代替std::tuple

```
std::tuple<int, double, char> compute();
void run() {
 int a; double b; char c;
 std::tie(a, b, c) = compute();
 ...
}
```

上面的代码可以简化为:

```
std::tuple<int, double, char> compute();
void run() {
 auto [a, b, c] = compute();
 ...
}
```

自动把a,b,c的类型绑定上。

还支持结构体的绑定:

```
struct Point { double x; double y; double z; };
Point computeMidpoint(Point p1, Point p2);
...
auto [x, y, z] = computeMidpoint(src, dest);
```

if条件中初始化

```
if(auto slash = path.rfind('/'); slash != std::string::npos)
  return {path.substr(0, slash), path.substr(slash + 1)};
```

上面直接在`if`条件中初始化局部变量`slash`，然后再用于条件中，如果想要在后面继续使用`slash`就会报错。

```
if (auto slash = path.rfind('/'); slash != std::string::npos)
  return {path.substr(0, slash), path.substr(slash + 1)};

// Much later, in unrelated code...
if (slash != std::string::npos)                    Use of undeclared identifier 'slash'
  launchTheSpaceship();
```

高级迭代

```
template <class Iterator> Iterator advance(Iterator it, long n) {
 if constexpr (is_random_access_iterator_v<Iterator>)
 return it + n;
 while (n > 0) { ++it; --n; }
 while (n < 0) { --it; ++n; }
 return it;
}
...
auto fifth_node = advance(list.begin(), 5);
auto fifth_char = advance(string.begin(), 5);
```

新增`advance`获取指定位置的值。

新增string_view

先看个例子:

```
#include <string>
std::pair<std::string, std::string> split(const std::string &path) {
 if (auto slash = path.rfind('/'); slash != std::string::npos)
 return {path.substr(0, slash), path.substr(slash + 1)};
 return {std::string{}, path};
}
```

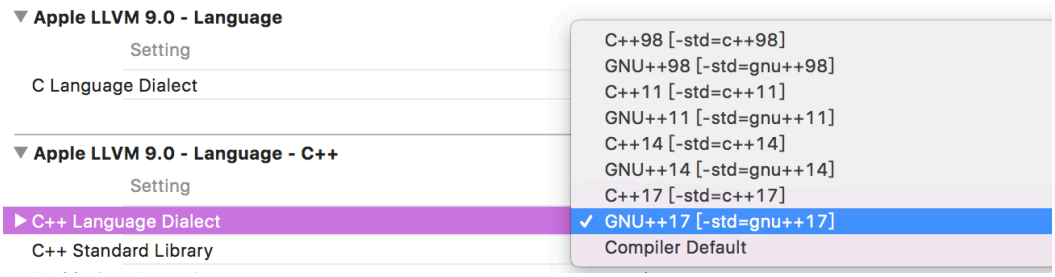
使用`std::string`会导致子字符串被拷贝，浪费内存，所以新加了`std::string_view`，它是对`const char *`和`size_t`的封装，有着丰富的api，和`std::string`用起来一样。但它只是引用原字符串做为子字符串，而不会拷贝。

再看下面的代码:

```
std::pair<std::string_view, std::string_view> split(std::string_view path);
...
auto [directory, filename] = split(pwd + "/" + path);
auto dot = filename.rfind('.');
```

因为`pwd + "/" + path`返回的是一个临时的变量，所以可能会在使用的时候被销毁，这个在编码的时候是需要注意的。

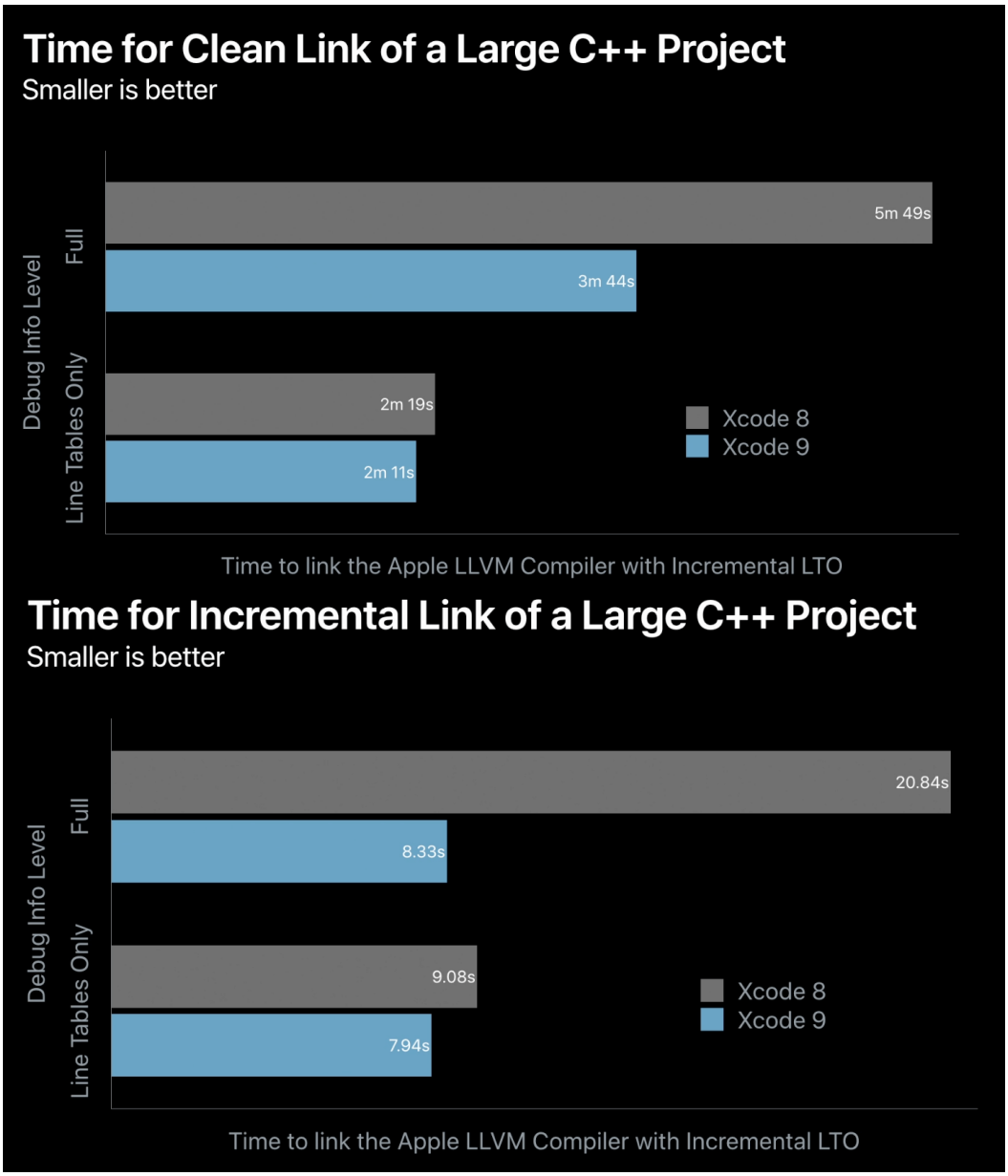
可以在编译设置里面选择使用`C++ 17`:



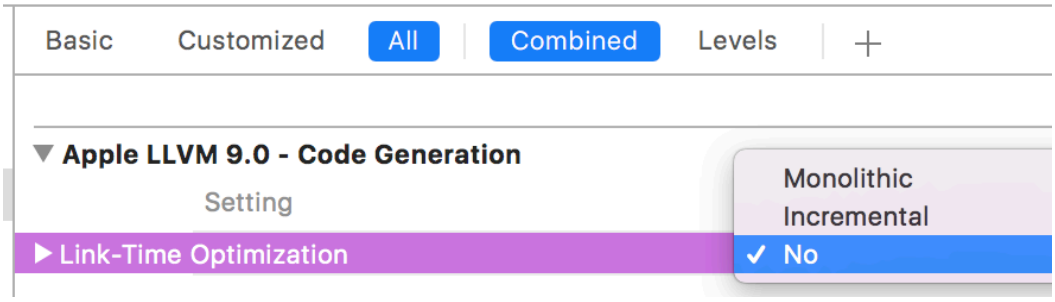
- C++17 没有语言扩展
- GUN++17 有语言扩展

链接时间优化

这部分没有细说，只有数据对比。



编译设置里面可以开启:



参考

视频地址: What's New in LLVM Video

ppt地址: [What's New in LLVM Keynote](https://devstreaming-cdn.apple.com/videos/wwdc/2017/411a7o9phe4uekm/411/411_whats_new_in_llvm.pdf)

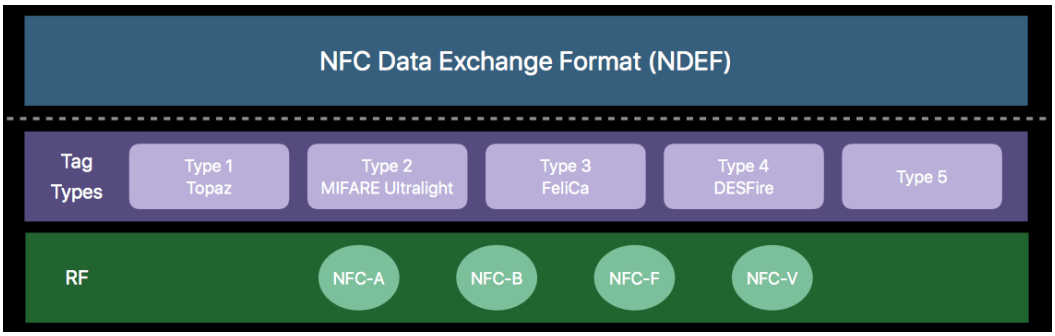
Introducing Core NFC

苹果在iOS 11上面向iPhone 7及以上设备开放了NFC读取的框架——Core NFC。

关于 NFC（Near Field Communication）

近场通讯，简称 NFC ，是一套能够让两个设备在近距离（几厘米）未接触的情况下进行无线数据通信的协议。它包含了一系列不同标准和协议，这些标准在不同的应用场景中相互作用。

最常见的是 NFC 设备是具有各种样式，尺寸和功能的 NFC 标签（tag）。标签最常见的5个类型标记为 Type 1 到 Type 5 ，每个类型对应一个标准。NDEF（NFC Data Exchange Format）为 NFC 读卡器提供了一种标准的通信数据格式。



Core NFC

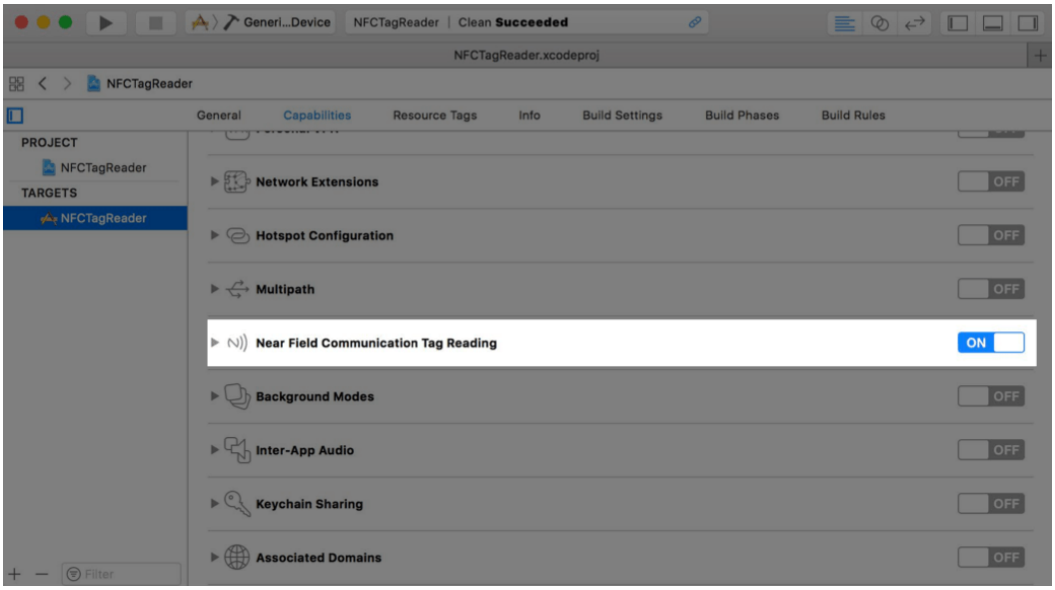
Core NFC 的使用首先需要满足这些条件：

- 包含 NDEF 格式数据的标签
- 标签属于类型 1 至类型 5 中的一种
- 硬件要求： iPhone 7 及 iPhone 7 Plus 以上

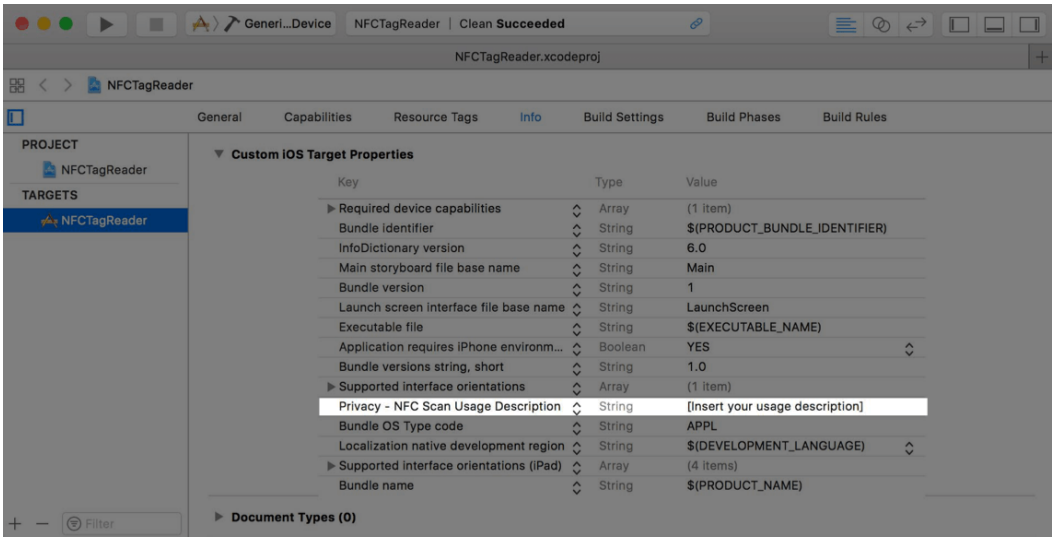
权限

要使用 Core NFC 首先需要申请权限：

- 将 Capabilities 的 Near Field Communication Tag Reading 设置打开（在第一个 Xcode 9 beta 版本中并不包含这个设置，需要去https://developer.apple.com/account/ios/certificate/ 主动申请）

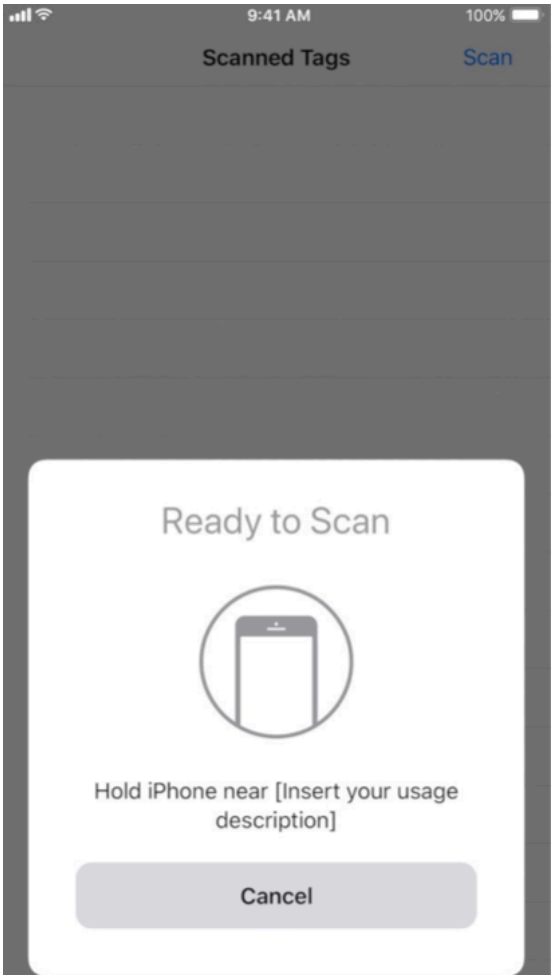


- 在 Info.plist 中加上 "Privacy - NFC Scan Usage Description"

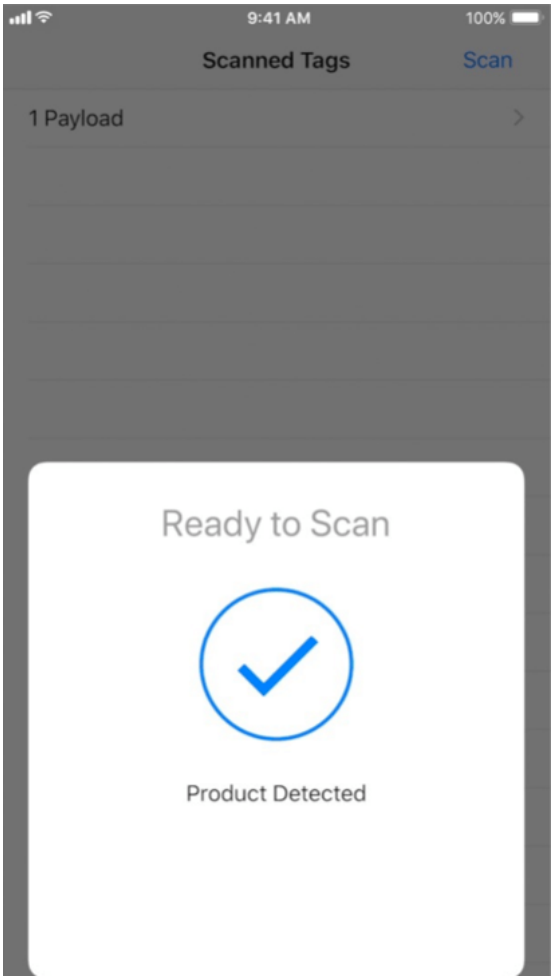


重要信息

- Core NFC 基于会话（session）
- 只能在前台状态下执行
- 每个会话支持最长60s的扫描时间
- 可支持单标签或多标签的读取：
  - 单标签：会话会在标签成功读取后立即结束；
  - 多标签：会话会在用户主动取消或达到60s后自动结束
- 在 Info.plist 中填写的描述会被展示在应用最上层，如下图



- 当读取成功后界面会发生改变



样例代码

使用 Core NFC 来读取标签仅需3步（不由得想到把大象放进冰箱的梗= =）

1. 实现 NFCNDEFReaderSessionDelegate 协议
2. 创建 NFCNDEFReaderSession 实例
3. 启动 session ，处理代理回调

```
import CoreNFC

class MessageTableViewController: UITableViewController,
NFCNDEFReaderSessionDelegate {

    // MARK: - NFCNDEFReaderSessionDelegate

    func readerSession(_ session: NFCNDEFReaderSession, didDetectNDEFs
messages: [NFCNDEFMessage]) {
        // 对读取到的 NFCNDEFMessage 对象进行处理
    }

    func readerSession(_ session: NFCNDEFReaderSession,
didInvalidateWithError error: Error) {
        // 根据返回的 error 来确定失败的原因，由于当前会话会在方法结束时被销毁，如果需要
继续读取标签需要重新创建会话实例。
    }

}
```

```
@IBAction func beginScanning(_ sender: Any) {
    // 如果要进行多标签读取，需要将 invalidateAfterFirstRead 设置为 false
    let session = NFCNDEFReaderSession(delegate: self, queue: nil,
invalidateAfterFirstRead: true)
    session.begin()
}
```

NFCNDEFMessage 对象包含了一组 NFCNDEFPayload 类型的对象

```
class NFCNDEFMessage: NSObject, CVarArg, Hashable, Equatable, NSSecureCoding
{
    var records: [NFCNDEFPayload]
}
```

每个 NFCNDEFPayload 对象由 标识（identifier），内容（payload），类型（type）和类型格式（typeNameFormat）组成

```
class NFCNDEFPayload: NSObject, CVarArg, Hashable, Equatable, NSSecureCoding
{
    var identifier: Data
    var payload: Data
    var type: Data
    var typeNameFormat: NFCTypeNameFormat
}
```

```
enum NFCTypeNameFormat: UInt8 {
    case absoluteURI
    case empty
    case media
    case nfcExternal
    case nfcWellKnown
    case unchanged
    case unknown
}
```

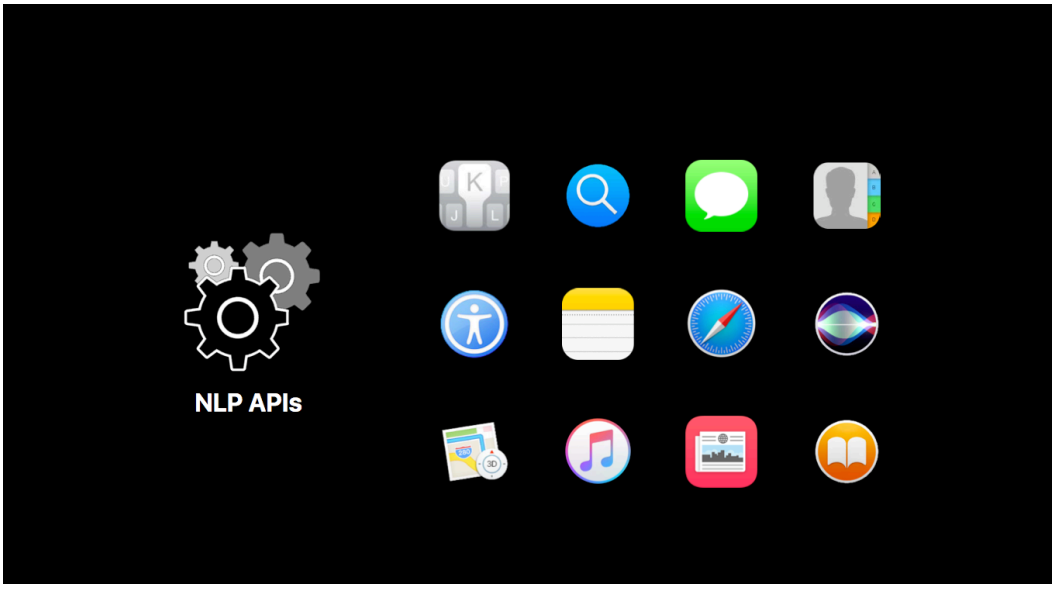
参考

- 视频地址：https://developer.apple.com/videos/play/wwdc2017/718/
- Slides地址：https://devstreaming-cdn.apple.com/videos/wwdc/2017/718jes6q3aac0f1a/718/718_introducing_core_nfc.pdf
- 样例代码：https://developer.apple.com/sample-code/wwdc/2017/CoreNFC-Tag-Reading.zip
- 官方文档：https://developer.apple.com/documentation/corenfc

用自然语言处理优化你的 App

自然语言处理（Natural Language Processing，以下简称 NLP）是人工智能和语言学的分支学科。**NLP** 框架基于 **Core ML** 针对文字输入（typed），手写识别（recognized handwriting）以及语音输入转换（transcribed speech）等应用场景进行了封装及优化 。

NLP API 已经在苹果自家的应用中使用，横跨多个平台，驱动多款应用。



NLP APIs 基石

- **语言识别（language identification）**：

  在处理文本时首先需要知道目标语言是什么。

- **文本切分（tokenization）**：

  将大段的文本切分成有意义的小段，切分的颗粒度通过标注单元（tagging units）指定 。

  如果你想要将一段文本以句子为单位进行切分，你需要设置标注单元为 **sentence** 。单纯的以 '.' 号作为句子的分隔符，下面这种情况会被错误的分成3句。

  当涉及到像中文这样复杂的语言时，要获得有意义的部分则需要将标注单元设置为 **word** 。

- **词性（part of speech）**：

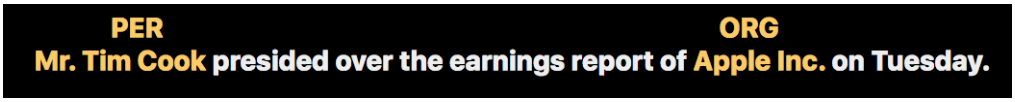
  对给定文本中的每一个词标注词性。

- **词行还原（lemmatization）**：

  将不同形态的词转为词根，对于处理像俄语，土耳其语这样拥有非常多词性复杂变化的语言会非常有帮助。

- **实体识别（named entity recognition）**：

  用于识别人名，地名，组织名等实体。



如何使用

NSLinguisticTagger (NS_CLASS_AVAILABLE(10_7, 5_0))

在 iOS 11 中，苹果对它进行了升级，以支持文本分割，实体识别，词性还原等多种新特性。

- 标注单元（tagging units）：在原有 word 的基础上新增了 sentence，paragraph，document

  ```
  public enum NSLinguisticTaggerUnit: Int {
      case word
      case sentence
      case paragraph
      case document
  }
  ```

  提供了新的 API 用来查询对应单元（unit）和语言所支持的 NSLinguisticTagScheme 。

  ```
  class func availableTagSchemes(for unit: NSLinguisticTaggerUnit, language:
  String) -> [NSLinguisticTagScheme]
  ```

- 语言识别（dominantLanguage）
- 针对 Swift 4 增加了标签和标记方案的命名类型
- 更好的性能
- 更高的准确率
- 更多的语言支持

NSLinguisticTagScheme

初始化 NSLinguisticTagger 时需要设定特征标记方案（Linguistic tag scheme）。

```
init(tagSchemes: [NSLinguisticTagScheme], options: Int)
```

特征标记方案（Linguistic tag scheme）	适用单元（Applicable linguistic units）	可能的返回值（Possible return values）
tokenType	word	将短语在大粒度上分成词语、标点符号、空格并返回
lexicalClass	word	将短语根据类型分为话语部分、标点符号、空格并返回
nameType	word	将短语根据是否为命名实体分类并返回
nameTypeOrLexicalClass	word	返回满足 nameType 或 lexicalClass 的部分
lemma	word	在词根可知时返回词根
language	word, sentence, paragraph, document	BCP-47 language tag
script	word, sentence, paragraph, document	ISO 15924 script code

NSLinguisticTagger Options

用来确定需要排除的符号或需要合并的短语

```
public struct Options : OptionSet {

    public init(rawValue: UInt)

    // 省略单词
    public static var omitWords: NSLinguisticTagger.Options { get }

    // 省略标点符号
    public static var omitPunctuation: NSLinguisticTagger.Options { get }

    // 省略空格
    public static var omitWhitespace: NSLinguisticTagger.Options { get }

    // 省略其他
    public static var omitOther: NSLinguisticTagger.Options { get }

    // 针对 NSLinguisticTagSchemeNameType。
    // 默认一个名字中的每个短语都被分成不同的实例，
    // 很多情况下需要将类似 "San Francisco" 这样的名字当作一个短语而不是两个短语来看待，
    // 传入这个属性即可实现这个功能。
    public static var joinNames: NSLinguisticTagger.Options { get }
}
```

API Demo

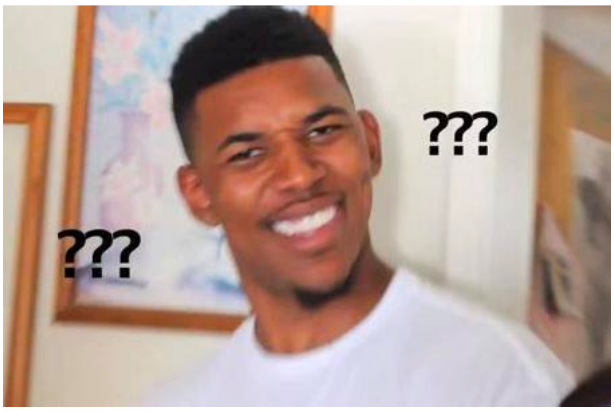
P.S.：以下代码都是在 Xcode 9.0 beta 2 测试的。

- 语言识别

```
import Foundation

let tagger = NSLinguisticTagger(tagSchemes: [.language], options: 0)
tagger.string = "Die Kleinen haben friedlich zusammen gespielt."
print(tagger.dominantLanguage ?? "language not found")

/**
 * 控制台输出：
 * de
 * 注：de 是德语的语言代码
 */
```

然而，下面这种情况，加 '.' 和不加会被识别成了两种语言= =

```
import Foundation

var tagger = NSLinguisticTagger(tagSchemes: [.language], options: 0)
tagger.string = "I love Swift"
print(tagger.dominantLanguage ?? "language not found")

tagger.string = "I love Swift."
print(tagger.dominantLanguage ?? "language not found")

/**
 * 控制台输出：
 * en
 * nb
 */
```



- 文本分割

```
import Foundation

let tagger = NSLinguisticTagger(tagSchemes: [.tokenType], options: 0)
let text = "Mr. Tim Cook presided over the earnings report of Apple Inc.
on Tuesday."
tagger.string = text
let range = NSRange(location: 0, length: text.utf16.count)
let options: NSLinguisticTagger.Options = [.omitPunctuation,
.omitWhitespace]

// 以分词为例：unit 设置为 word
// 注意：enumerateTags 方法中的 scheme 参数必须在 tagger 初始化时设定的
tagSchemes 中
tagger.enumerateTags(in: range, unit: .word, scheme: .tokenType, options:
options) { (tag, tokenRange, stop) in
    let token = (text as NSString).substring(with: tokenRange)
    print(token)
}

/**
 * 控制台输出：
 * Mr.
 * Tim
 * Cook
 * presided
 * over
 * the
 * earnings
 * report
 * of
 * Apple
 * Inc.
 * on
 * Tuesday
 */
```

```
// 以分句为例：unit 设置为 sentence
tagger.enumerateTags(in: range, unit: .sentence, scheme: .tokenType,
options: options) { (tag, tokenRange, stop) in
    let token = (text as NSString).substring(with: tokenRange)
    print(token)
}

/**
 * 控制台输出：
 * Mr. Tim Cook presided over the earnings report of Apple Inc. on
Tuesday.
 */
```

- 词性

```swift
import Foundation

// 这里仅对特定语言获取 tagSchemes 做一个示范
let schemes = NSLinguisticTagger.availableTagSchemes(forLanguage: "en")
let tagger = NSLinguisticTagger(tagSchemes: schemes, options: 0)
let text = "Tim Cook is the CEO of Apple Inc. which is located in
Cupertino, California."
tagger.string = text
let range = NSRange(location: 0, length: text.utf16.count)

// 注意 options 中的 joinName
let options: NSLinguisticTagger.Options = [.omitPunctuation,
.omitWhitespace, .joinNames]
tagger.enumerateTags(in: range, unit: .word, scheme:
.nameTypeOrLexicalClass, options: options) { (tag, tokenRange, stop) in
    guard let tag = tag else { return }
    let token = (text as NSString).substring(with: tokenRange)
    print(token + ": " + tag.rawValue)
}

/**
 * 控制台输出:
 * Tim Cook: PersonalName
 * is: Verb
 * the: Determiner
 * CEO: Noun
 * of: Preposition
 * Apple Inc.: OrganizationName
 * which: Pronoun
 * is: Verb
 * located: Verb
 * in: Preposition
 * Cupertino: PlaceName
 * California: PlaceName
 */
```

- 词形还原

```
import Foundation

let tagger = NSLinguisticTagger(tagSchemes: [.lemma], options: 0)
let text = "Great hikes make great pics! Wonderful afternoon in Marin
County."
tagger.string = text
let range = NSRange(location: 0, length: text.utf16.count)
let options: NSLinguisticTagger.Options = [.omitPunctuation,
.omitWhitespace]
tagger.enumerateTags(in: range, unit: .word, scheme: .lemma, options:
options) { (tag, tokenRange, stop) in
    let token = (text as NSString).substring(with: tokenRange)
    guard let lemma = tag?.rawValue, lemma != token else {
        return
    }
    print(token + ": " + lemma)
}

/**
 * 控制台输出:
 * Great: great
 * hikes: hike
 * pics: pic
 * Wonderful: wonderful
 * County: county
 */
```

- 实体识别（named entity recognition）

```
import Foundation

// 这里仅对特定语言获取 tagSchemes 做一个示范
let schemes = NSLinguisticTagger.availableTagSchemes(forLanguage: "en")
let tagger = NSLinguisticTagger(tagSchemes: schemes, options: 0)
let text = "Tim Cook is the CEO of Apple Inc. which is located in
Cupertino, California."
tagger.string = text
let range = NSRange(location: 0, length: text.utf16.count)
let options: NSLinguisticTagger.Options = [.omitPunctuation,
.omitWhitespace, .joinNames]

// 用来对 tag 的输出做区分
let tags: [NSLinguisticTag] = [.personalName, .placeName,
.organizationName]

tagger.enumerateTags(in: range, unit: .word, scheme:
.nameTypeOrLexicalClass, options: options) { (tag, tokenRange, stop) in
    guard let tag = tag, tags.contains(tag) else { return }
    let token = (text as NSString).substring(with: tokenRange)
    print(token + ": " + tag.rawValue)
}

/**
 * 控制台输出:
 * Tim Cook: PersonalName
 * Apple Inc.: OrganizationName
 * Cupertino: PlaceName
 * California: PlaceName
 */
```

总结

优势

- 全平台体验一致
- 隐私安全
  - 基于设备的机器学习（On-device machine learning）
  - 用户数据存于本地（User data stays on-device）
- 卓越的性能
- 更高的准确性

语言支持

功能	支持语言
语言识别	29种脚本，52种语言
文本切分	所有 iOS / macOS 的系统语言
词形还原	英语，法语，意大利语，德语，西班牙语，葡萄牙语，俄语，土耳其语
词性	同上
实体识别	同上

调试技巧

- 模型和对应语言的键盘绑定，安装了某一语言的键盘后模型会被下载到设备上。如果出现获取到的 tag 是 NSLinguisticTagOtherWord，很可能是模型没有被下载
- 尽可能指定语言

参考

视频地址：https://developer.apple.com/videos/play/wwdc2017/208/

Slides 地址：https://devstreaming-cdn.apple.com/videos/wwdc/2017/208tpmh7cwo17vk/208/208_natural_language_processing_and_your_apps.pdf

NSHipster NSLinguisticTagger： http://nshipster.cn/nslinguistictagger/

Winnow（demo 1）：https://developer.apple.com/sample-code/wwdc/2017/Enhancing-App-Experiences-with-NLP.zip

Whisk（demo 2）：https://developer.apple.com/sample-code/wwdc/2017/NLP-Whisk.zip

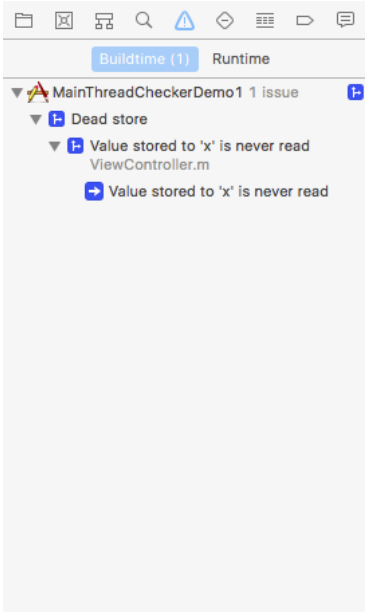
使用 Xcode 运行时工具帮助排查 Bug

1. 介绍

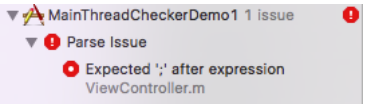
> iOS 应用程序主流开发 IDE 包括：苹果自家的 Xcode，JetBrains 的 AppCode 以及刚刚登陆 MacOS 的宇宙第一 IDE Visual Studio。其中 Xcode 被吐槽较多，功能少，时不时地转菊花，甚至给你 crash 一下，不过我们也可以看到这几年苹果开发团队在工具这块的努力，比如去年加入的 `Memory Graph` 功能——觉得非常好用，这里给出卓同学的博客文章Xcode8调试黑科技：Memory Graph实战解决闭包引用循环问题。而今年同样又带来了更多运行时工具帮助我们排查和解决问题，闲话少说，开始我们的正题：

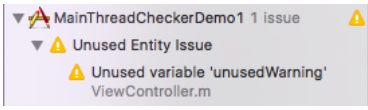
日常开发中，我们所写的程序到可执行文件需要经过四个步骤：预处理，**编译**，汇编和链接，其中编译过程又可分为：1. 词法分析，将代码的字符序列分割成一系列的词法单元(token)；2. 语法分析，生成抽象语法树(AST)；3. **语义分析，检查表达式是否合法有效**；4. 源代码优化器，用于生成与机器无关的中间代码表示(IR)；5. 最后是代码生成器和目标代码优化器，用于生成汇编代码以及做一些优化处理。

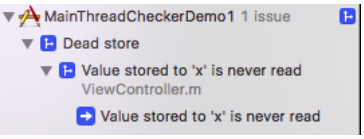
其中编译过程中所能分析的语义是**静态语义**，是在编译期间就已经可以确定的语义，比如除数不能为 0；而与之相对的**动态语义**则是在运行时才能确定的语义。前者 Xcode 会将发现的问题集中显示在 `Issue navigator` 的 `Buildtime` 一栏中：



我们会经常遇到错误、警告、静态分析问题以及单元测试失败信息：







那么程序运行时的动态语义分析问题我们又该如何定位和排查呢？ 这正是 Session 406 所要呈现的！而运行时问题符号标识想必你也不陌生了。

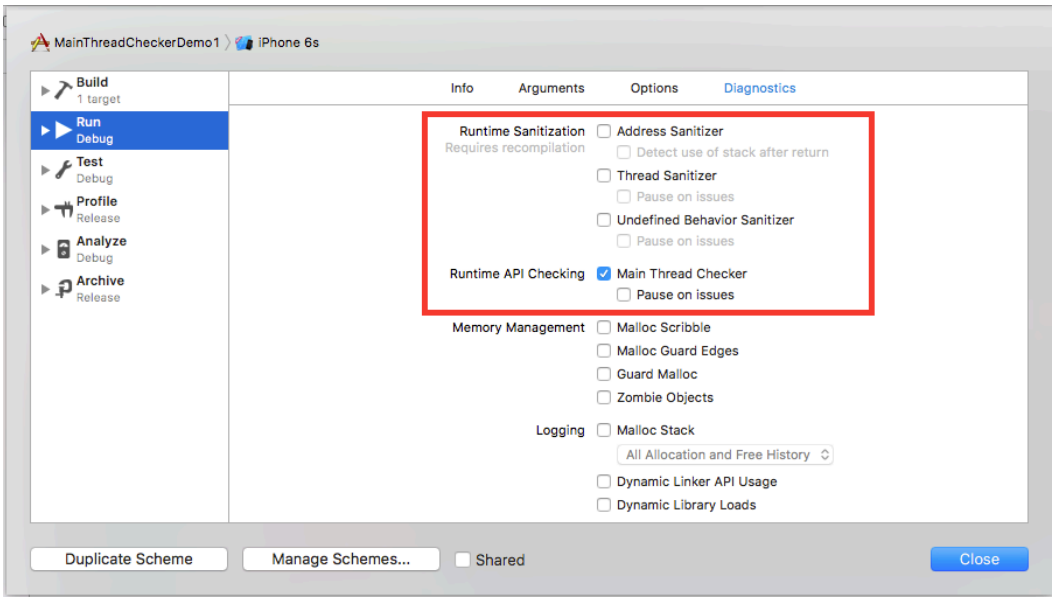
2. Runtime Checking ——运行时检查工具

> 测试环境要求：Xcode 9.0 beta 及以上

首先通过下面两种方式打开 `Edit Scheme` 面板：

- 点击项目 `Target` 选中 `Edit Scheme`；
- Xcode 顶部菜单栏中按照 `Product->Scheme->Edit Scheme` 路径进入面板

此时你应该看到如下界面：



图形中红色矩形框标识了 4 种运行时检测工具，分别是：

- Main Thread Checker 默认开启，新功能
- Address Sanitizer
- Thread Sanitizer
- Undefined Behavior Sanitizer — 新功能

2.1 Main Thread Checker

2.1.1 功能概述

众所周知，某些 API 当且仅在主线程中被调用使用，例如 AppKit 和 UIKit 两个框架操作；而其他诸如文件下载、图片处理等操作则可以放置到后台线程中执行，如下：

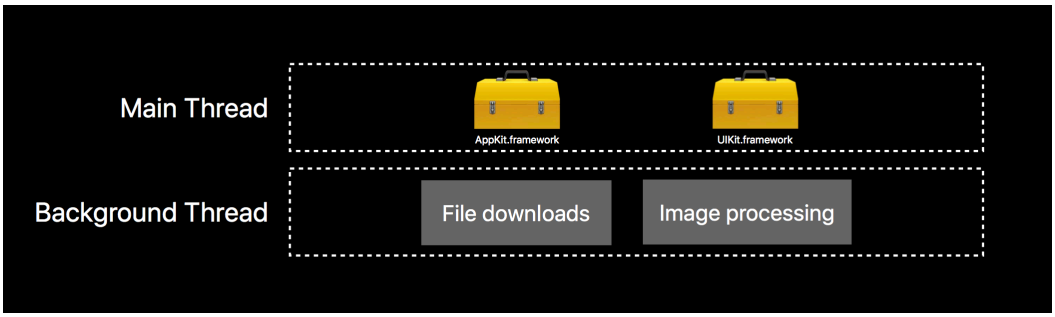
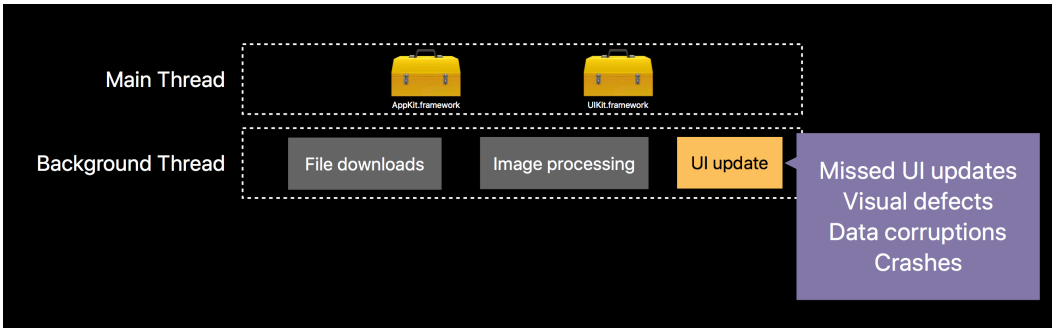


Image Processing 图片处理在后台线程中执行完毕，会更新 `ImageView` 中的图片内容，若更新操作发生在 `Background Thread` 中，`Main Thread Checker` 会检测并给出警告，如下：



正确做法是将 UI 更新操作放置到主线程中执行。

2.1.2 Demo 演示

> Demo 下载地址：MainThreadCheckerDemo
>
> 演示功能：检测 UI 更新操作是否发生在主线程。

打开 Demo 工程，首先确保 `Edit Scheme` 面板中开启了 `Runtime API Checking`；工程相当简单，视图中放置了一个 `Button` 和一个 `Label`，点击按钮将繁重的任务加入到全局队列中（并发队列）去执行，由 GCD 来负责派发任务到某个线程（非主线程），延迟5秒后更新 `Label` 的显示文字，但是此时 UI 更新操作并未处于主线程中，这显然是不正确的！是时候运行工程来看看 `Main Thread Checker` 是否能够帮助我们排查出这个问题：

```
- (IBAction)changeText:(UIButton *)sender {
    dispatch_queue_t queue = dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);

    dispatch_async(queue, ^{
        // 执行繁重的计算工作 这里用延迟模拟
        [NSThread sleepForTimeInterval:5];

        // 更新UI
        self.descriptionLabel.text = @"计算完毕";
    });

}
```

点击按钮等待 5 秒，Xcode 聪明地定位到问题代码，并给出了运行时问题：

```
- (IBAction)changeText:(UIButton *)sender {
    dispatch_queue_t queue = dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);

    dispatch_async(queue, ^{
        // 执行繁重的计算工作 这里用延迟模拟
        [NSThread sleepForTimeInterval:5];

        // 更新UI
        self.descriptionLabel.text = @"计算完毕";   -[UILabel setText:] must be called from main thread only
    });

}
```

2.1.3 易发生错误的地方

- 网络回调

- 创建和销毁 `NSView`、`UIView` 等对象
- 设计异步接口时

重点讨论如何设计异步接口，思考下面这个异步接口有什么问题?

```
DeepThought.asyncComputeAnswer(to: theQuestion) { reply in ...
    // 任务执行完毕 回调允许你做一些事情
}
```

该接口对传入的 `theQuestion` 问题进行异步计算，为了不阻塞主线程，将计算放置到某个线程中异步计算，等到计算完毕则回调让你处理一些事情，而此时处于什么线程并未指定！倘若你放置一些 UI 更新行为显然是不正确的。因此我们需要对接口做出些许改动，由调用方来指定回调闭包到哪个队列执行：

```
DeepThought.asyncComputeAnswer(to: theQuestion, completionQueue: queue) {
reply in
    // 这里指定了回调闭包的队列 如果是UI更新行为，指定 main queue 即可
}
```

2.1.4 小结

- 检测违反 API 线程规则的行为
- AppKit 和 UIKit 接口必须在主线程中调用
- 适用于 Swift 、C 语言
- 不需要重新编译
- 默认开启

> 更多有关 GCD 知识，可阅读 Grand Central Dispatch(GCD) 深入浅出 一文。

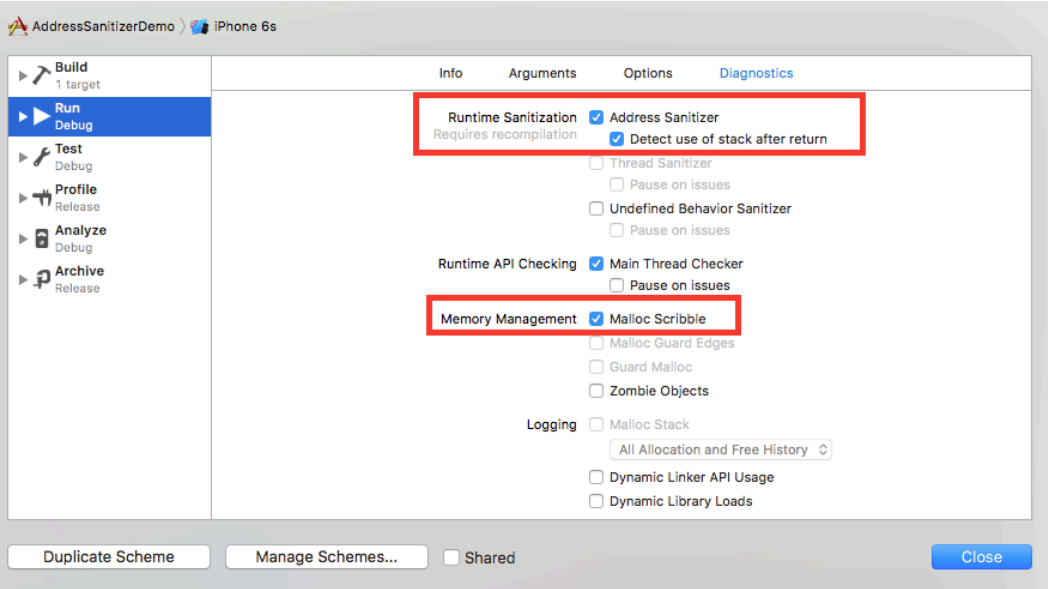
2.2 Address Sanitizer —— 检测内存问题

我们将通过几个示例来介绍 Address Sanitizer 工具的强大之处，当然你若是想深入了解这个工具，不妨看看 WWDC 2015 `Advanced Debugging and the address Sanitizer` 这个 Session，希望对你有帮助。

2.2.1 检测 use-after-free 问题

> Demo 下载地址: AddressSanitizerDemo
>
> 演示功能：使用已经释放内存的对象或指针，Xcode 会给出错误信息。

打开 Demo 工程，首先确保 `Edit Scheme` 面板中勾选了`Address Sanitizer`，`Detect use of stack after return` 以及内存管理中的 `Malloc Scribble`，如下：



查看工程 `AppDelegate.m` 代码：

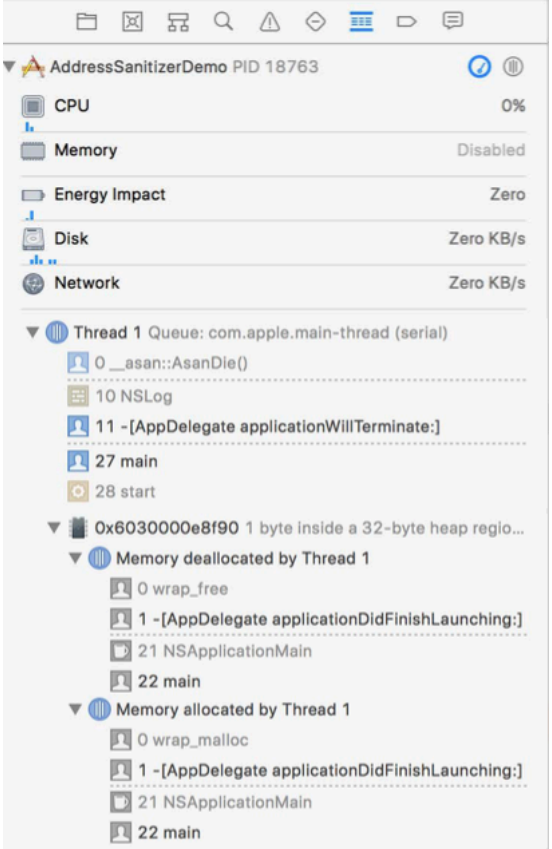
```
@interface AppDelegate ()

@end

@implementation AppDelegate
/// 1
char * buffer;

- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions {
    /// 2
    buffer = malloc(32);
    snprintf(buffer, 32, "Hello, World!");
    NSLog(@"%s",buffer);
    free(buffer);
    return YES;
}

- (void)applicationWillTerminate:(UIApplication *)application {
    /// 3
    NSLog(@"%s",buffer);
}
@end
```

1. 声明一个 `char *` 类型指针 `buffer`，此时并未指向任何有效内存块；
2. 使用 `malloc` 方法为 `buffer` 分配一块大小为 32 字节的内存块，并初始化内容为*Hello, World!*；接着使用 `NSLog` 打印字符串；最后释放掉 `buffer` 指针指向的内存；
3. 在应用程序即将终止前打印 `buffer` 内存 —— 此时 `buffer` 早已经释放。

点击运行程序，看到终端输出 `Hello, World!`；杀掉程序（同时按下 `shift+command+H` 两次），观察 Xcode 给出的错误信息：

```objc
- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:
    (NSDictionary *)launchOptions {
    buffer = malloc(32);
    snprintf(buffer, 32, "Hello, World!");
    NSLog(@"%s",buffer);
    free(buffer);
    return YES;
}

- (void)applicationWillTerminate:(UIApplication *)application {
    NSLog(@"%s",buffer);                    Thread 1: Use of deallocated memory
}
```

观察左侧 Debug navigator 面板，居然输出 `buffer` 在何处创建以及在何处被释放！这个功能实在强大，帮助我们排查和定位内存问题。



2.2.2 检测 use-after-scope 问题

首先思考下面这段代码存在哪些问题：

```
int *integer_pointer = NULL; // 1
if (is_some_condition_true()) {
    int value = calculate_value(); //2
    integer_pointer = &value; // 3
}
*integer_pointer = 42; // 4
```

1. 首先定义一个 `int` 类型的指针 `integer_pointer`；
2. 在 `if` 作用域中我们定义了 `value` 接收 `calculate_value` 函数返回的计算结果；
3. 将指针 `integer_pointer` 指向 `value` 的内存地址；
4. 修改 `integer_pointer` 指向内存块的值。

乍看之下没有任何问题，点击运行 2.2.1 中的示例：

Xcode 抛出了 **Use of out-of-scope stack memory** 错误，问题原因主要在于变量 `value` 的作用域是 `if` 语句括号内，一旦出了大括号，`value`声明周期宣告结束，栈上分配的内存会被收回，此时 `integer_pointer`指针指向的内存不复存在！

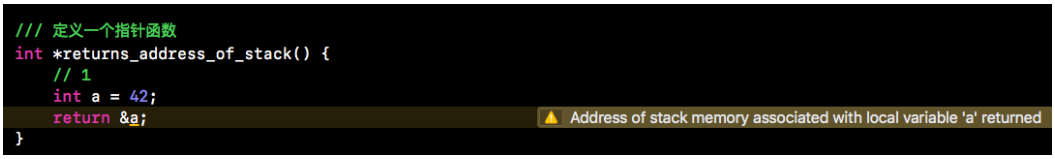
2.2.3 检测 use-after-return 问题

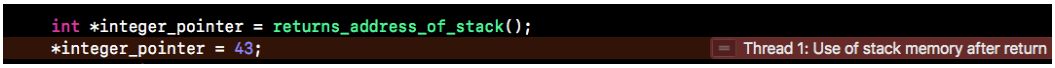
C 语言面试题中经常涉及，请看如下代码：

```
/// 定义一个指针函数
int *returns_address_of_stack() {
    // 1
    int a = 42;
    return &a;
}
// 2
int *integer_pointer = returns_address_of_stack();
*integer_pointer = 43;
```

1. 函数中定义变量 `a`，然后返回 `a` 的地址
2. 使用 `int *` 类型的 `integer_pointer` 指针接收函数返回值，并设置指针指向内存块的值。

注释掉 2.2.2 代码，运行`detect use-after-return issues` 代码片段：





注意编译之前，Xcode 就已经抛出警告了，若一意孤行运行程序我们将得到 `Use of stack memory after return` 错误信息。问题原因一目了然，变量`a`的作用域局限于函数体，一旦跳出函数则生命周期结束，栈上分配的内存块就会被收回，即使返回了之前变量 `a` 的内存地址，其实也是无意义的。

2.2.4 Swift 中的 Address Sanitizer

Swift 作为一门安全语言，相对来说问题会少很多，但同样存在上述问题，如下：

```
let string = "Hello, World!"
var firstBytePointer:UnsafePointer<Char>  // 1
...
string.withCString { pointerToCString in
    firstBytePointer = pointerToCString // 2
}
...

let firstByte = firstBytePointer.pointee // 3
print(firstByte)
```

1. 定义一个 `UnsafePointer<Char>` 类型的变量，等同于 `const Char *`，即指针可变，指针指向的内存值不可修改
2. Swift 提供给调用者操作对象指针的接口，其中 `pointerToCString` 指向 `string` 字符串内存

首地址，注意它的作用域局限于大括号里，一旦跳出，变量生命周期结束

3. 使用 `firstBytePointer.pointee` 获取 `firstBytePointer` 指针指向内存区域的值，等同于 * 解引操作。

有了之前的学习经验，这里代码问题一目了然，是 `Use of deallocated memory` 的问题。

正确的使用姿势为：

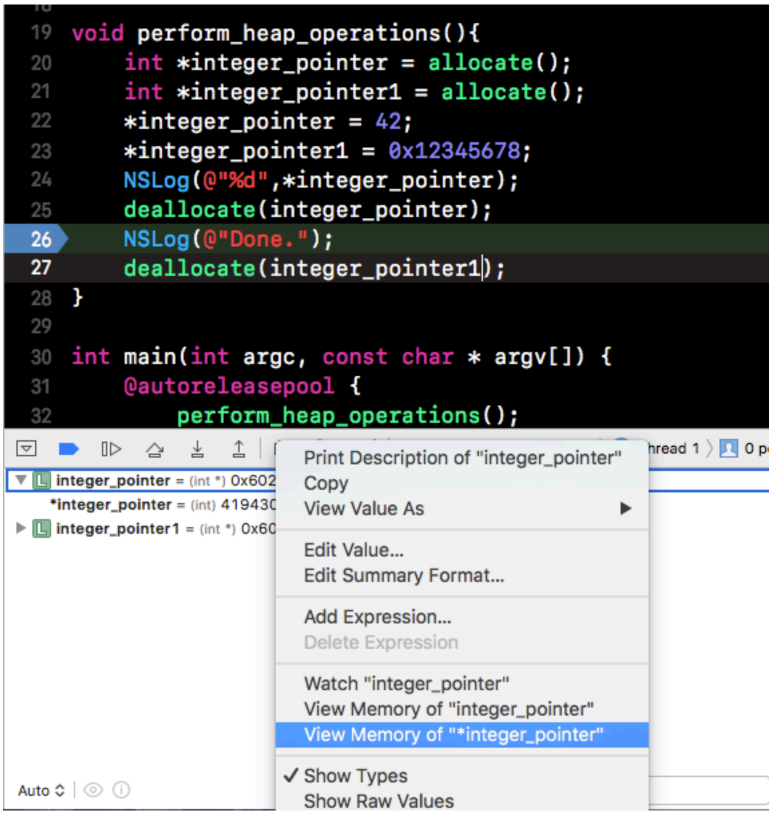
```
let string = "Hello, World!"

string.withCString { pointerToCString in
    let firstByte = pointerToCString.pointee
    print(firstByte)
}
```

2.2.5 更多调试小技巧

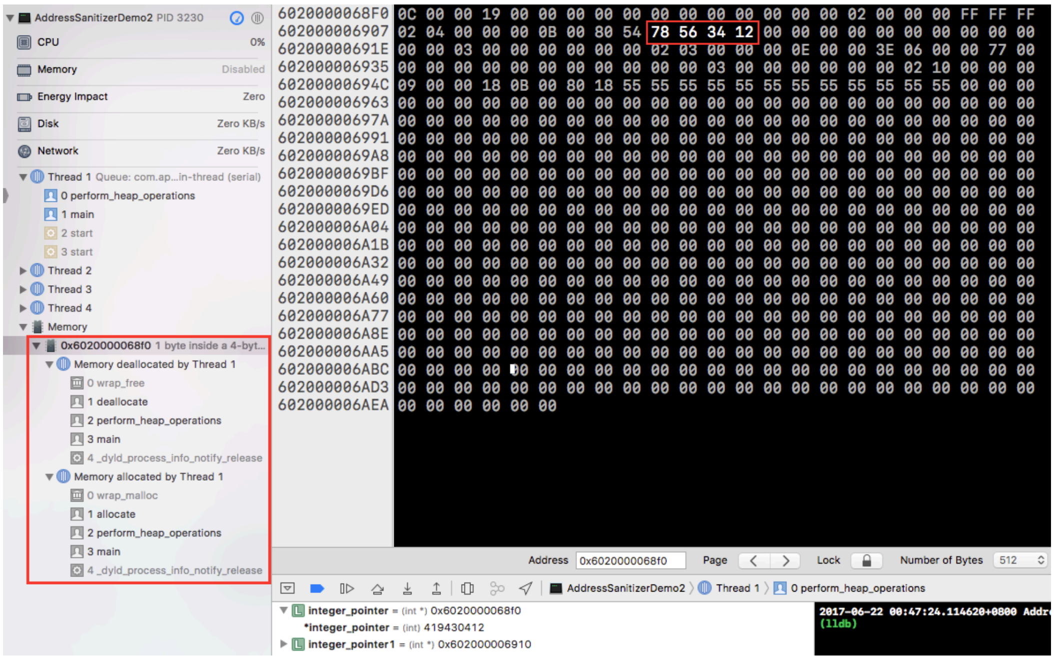
接下来我们将通过另外一个小 Demo 来学习查看 `allocation` 和 `deallocation` 的调用栈，点击下载 AddressSanitizerDemo2，现在打开工程查看代码：

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

int *allocate(){
    return malloc(sizeof(int));
}

void deallocate(int *p) {
    free(p);
}

void perform_heap_operations(){
    int *integer_pointer = allocate();
    int *integer_pointer1 = allocate();
    *integer_pointer = 42;
    *integer_pointer1 = 0x12345678;
    NSLog(@"%d",*integer_pointer);
    deallocate(integer_pointer);
    NSLog(@"Done.");
    deallocate(integer_pointer1);
}

int main(int argc, const char * argv[]) {
    @autoreleasepool {
        perform_heap_operations();
    }
    return 0;
}
```

在第 24 行设置断点，点击运行等待程序执行到 `NSLog(@"Done.");`

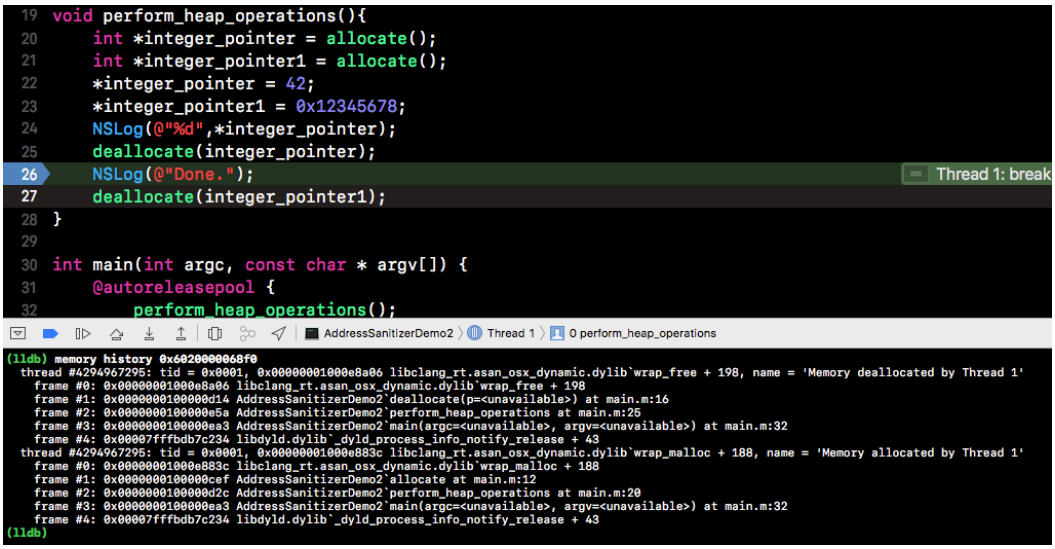


按照上图查看 `integer_pointer` 指针指向内存区域内容。当然你也可以使用 `shift+command+m` 快捷键呼出查看内存面板，然后手动输入内存地址。现在你看到的内存面板如下所示：



注意左侧红色方块显示了 `integer_pointer` 分配内存和释放内存的调用时机；上部红色方框标出的高亮部分为有效的内存区域——代码中为 `integer_pointer1` 指向的内存区域值，由于其内存释放在断点之后，因此这里未释放，值显示为 `0x12345678`；而其他灰色的内存区域表示未被使用，无效区域。

接下来我们使用 `lldb` 来输出内存使用历史情况，在终端键入 `memory history 0x6020000068f0` 命令，输出如下：



输出内容其实和左侧红框标出内容是一致的。

2.3 Thread Sanitizer —— 检测多线程问题

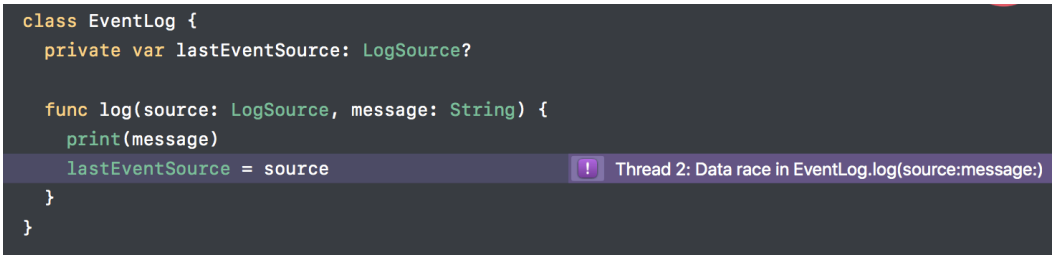
2.3.1 检测直接访问内存引发的数据竞争

多线程技术在日常应用中具有许多优势，诸如资源利用率高，程序响应快以及设计简单，但问题随之接踵而来，首当其冲的就是数据竞争(**Data Race**)，试想某个可变的共享数据，被多个线程同时访问修改，这将导致计算结果不符合预期，甚至导致内存损坏以及程序崩溃，同样问题在 Swift 中也无法避免。

接下来我们通过代码来一一讲述可能存在的问题以及解决方式：

```
class EventLog {
    private var lastEventSource: LogSource?
    func log(source: LogSource, message: String) {
        print(message)
        lastEventSource = source // 1 等同于 setter 方法
    }
}

// 线程 1
eventLog.log(source: networkingSubsystem, message: "Download finished")

// 线程 2
eventLog.log(source: databaseSubsystem, message: "Query complete")
```

线程一和线程二同时调用 evenLog 的 `log` 方法，打印日志信息并设置 `lastEventSource` 变量新值，由于两者都希望修改它，形成了一种竞争关系，即 `Data Race`。



为了消除竞争关系，解决方法是将任务逐一加入到自定义队列中，按照 FIFO 顺序执行，更改后的代码如下：

```swift
class EventLog {
    private var lastEventSource: LogSource?
    private var queue = DispatchQueue(label: "com.example.EventLog.queue")

    func log(source: LogSource, message: String) {
        // 异步方式将任务加入队列中，不阻塞当前进程。
        queue.async {
            print(message)
            lastEventSource = source // 1 等同于 setter 方法
        }
    }
}
```

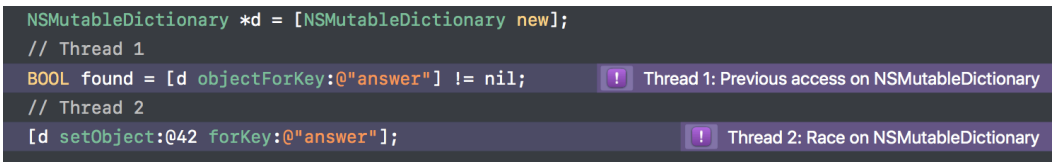
> GCD(Grand Central Dispatch) 应该成为我们使用多线程的首选，它具有轻量级，便利且易上手等优势。

2.3.2 检测访问集合引发的数据竞争

上文所述的 Data Race 发生在多线程同时访问修改同一块内存区域，而接下来我们聊一聊 Xcode 9 中 Thread Sanitizer 新增功能，主要面向集合(Collection)的 Data Race 检查，请看如下代码：

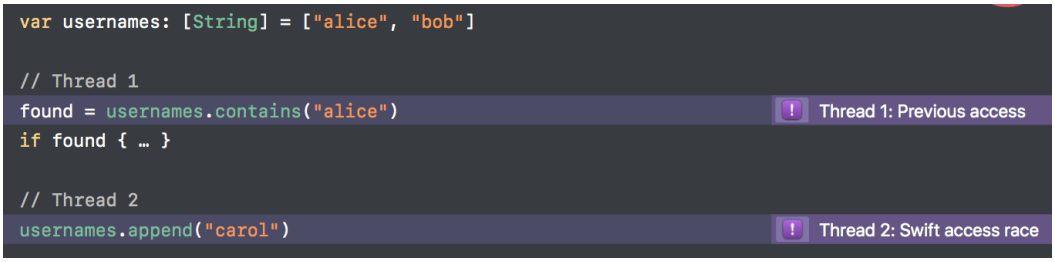
```objc
NSMutableDictionary *d = [NSMutableDictionary new];
// 线程 1 通过键取值
BOOL found = [d objectForKey:@"answer"] != nil;
// 线程 2 设值
[d setObject:@42 forKey:@"answer"];
```

执行代码，Xcode 会给出代码存在的问题：



现在 Xcode 9 不但能检测出多线程**同时**且**直接**访问内存导致的数据竞争，而且对于那些间接访问的数据竞争也能给出警示。包括 Objective-C 中的 `NSMutableArray` 和 `NSMutableDictionary`，以及 Swift 中的 `Array` 和 `Dictionary`。

例如下面代码中线程 1 调用 `contains` 方法判断是否包含了 “alice”，这里间接访问了数组内容；而随后线程 2 调用 `append` 方法修改数组内容，结果可能导致程序崩溃，想必大家也应该踩过这样的坑吧。



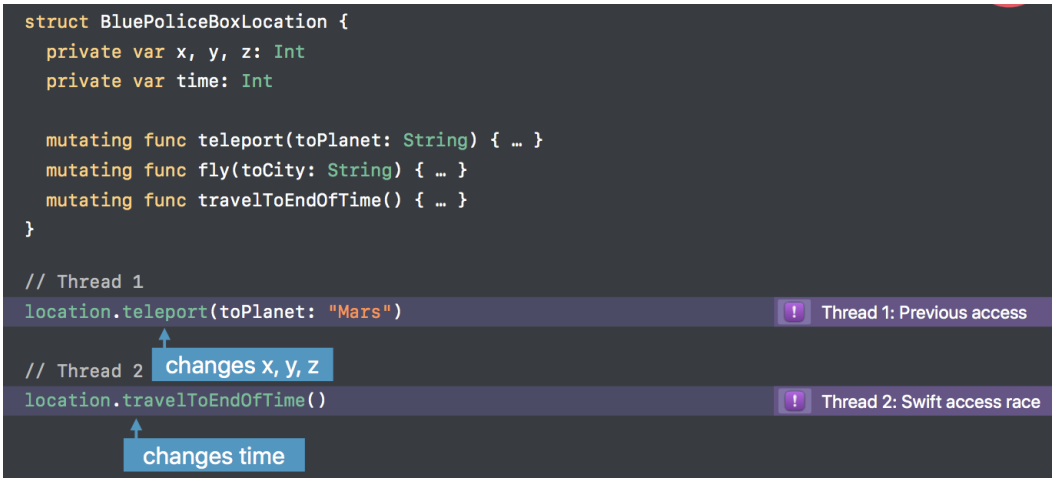
现在借助 gcd 来修改代码：

```swift
var usernames: [String] = ["alice", "bob"]
var queue = DispatchQueue(label: "com.example.usernames.queue")

// 线程 1
queue.sync {
    found = usernames.contains("alice")
}
if found { ... }

// 线程 2
queue.async {
    usernames.append("carol")
}
```

2.3.3 Swift 中的数据竞争

一谈起 Swift 中的结构体(structure)和类(class)，不免又引发一场激烈的讨论。当然本节只谈论 Swift 中结构体的数据竞争与类的数据竞争的不同点，不涉及孰优孰劣的问题。

- 结构体中的可变方法(mutating methods) 要求独占整个结构体，换言之当某个线程调用结构体实例的可变方法执行操作 —— 可能只修改结构体中的某个变量，而此时其他线程正在调用结构体实例的另一个可变方法 —— 修改结构体中其他变量，此刻也视为数据竞争。
- 类中的方法则没有这么“霸道”，它只独占访问的存储属性，即某个线程调用 `method1` 方法修改属性 X 的值，此刻其他线程调用 `method2` 方法修改属性 Y 的值，则不构成数据竞争，当然倘若 `method2` 方法中修改了 属性 X 的值，即视为数据竞争。



可以看到线程 1 和 线程 2 调用结构体的可变方法，两者修改的变量互不干涉，但仍然给出了数据竞争的提示。这里又该如何解决呢？大多数童鞋可能第一时间想到的解决方案是：新增实例一个 `queue` 队列，每个方法中用 `queue.sync{}` 来包裹之前的代码。

很遗憾，这不是一个好主意！正确的做法是使用一个类容器来包裹结构体，代码如下：

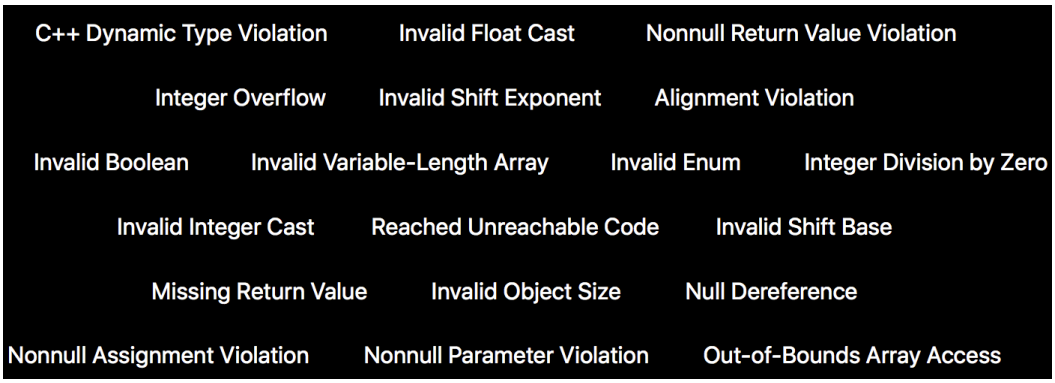
```
struct BluePoliceBoxLocation { ... }

class BluePoliceBox {
    private var location: BluePoliceBoxLocation
    private var queue: DispatchQueue = ...

    func goOnRescueMission() {
        queue.sync {
          location.teleport(toPlanet:"Mars")
          ...
        }
    }

    func goToWrongPlaceAgain() {
        queue.sync {
          ...
        }
    }
}
```

本节中，我们借助 GCD 的串行队列，以同步的方式访问资源，防止资源竞争问题发生，当然 Thread Sanitizer 真的非常强大，将代码中潜在的资源竞争问题暴露无遗。

2.4 Undefined Behavior Sanitizer

首先我们简单了解下什么是 **Undefined Behavior Sanitizer** ，有什么特点？1. 它同样是一个运行时 bug 检测工具；2. 帮助检查 C 系语言中不安全的结构；3. 它能够与其他一个或多个运行时工具配合使用。

它的涉及点真的很广，见下图：



本节主要讲述三个方面：整数溢出问题(*Integer Overflow*)、内存对齐问题(*Alignment Violation*)以及非空返回值问题(*Nonnull Return Value Violation*)。

2.4.1 整数溢出问题(*Integer Overflow*)

整数溢出问题一直是个头疼的问题，早前 Xcode 并不会因为整数而抛出错误或给出提示信息，因此你可能会得到一些莫名其妙的数据，例如：

```
int x = 2147483647;
x += 1;
NSLog(@"x value is %d",x);// 输出-2147483648
```

由于 `x` 是一个有符号整数，其值范围是 0x00000000 — 0x7FFFFFFF，如果执行加一操作，则发生溢出错误，但旧的 Xcode 并未给出任何错误。

实际上，倘若开发过程中我们能够细心一些，上面代码的问题还是能够发现的。但是涉及到一些复杂的算法，且仅当达到某种条件时才溢出时，就加大了排查难度。OK，现在使用 Xcode 9 的新功能，请确保在 `Edit Scheme` 中勾选了 **Undefined Behavior Sanitizer** 选项，点击运行你应该看到如下提示：

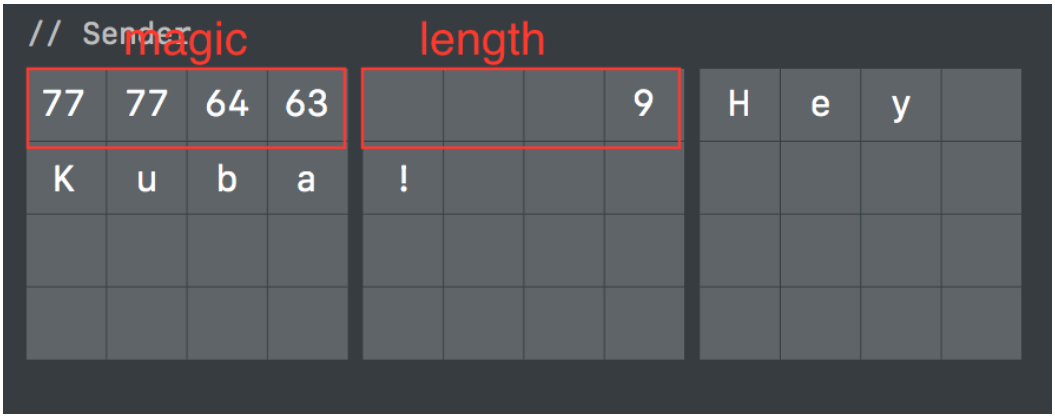
> 请牢记：**(INT_MAX + 1) > INT_MAX**

2.4.2 内存对齐问题(*Alignment Violation*)

内存对齐问题相对于其他问题可能受关注较少，但我们仍然需要对其有一定了解，本小节将通过一个自定义网络协议的数据包结构展开讨论。

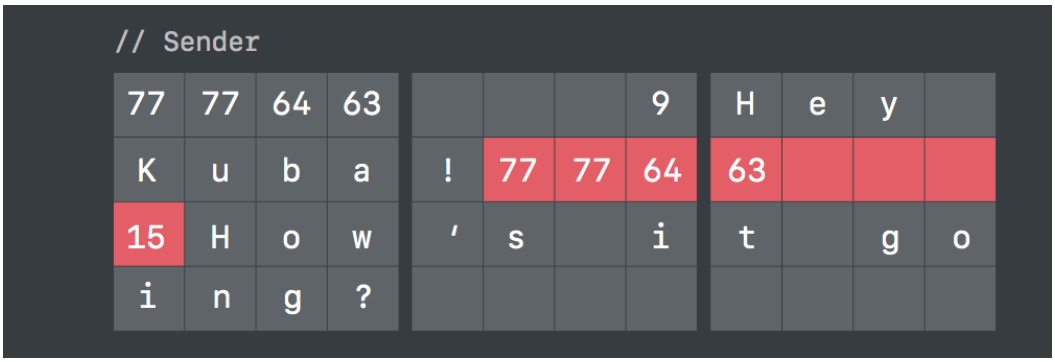
```
struct Packet {
  int magic;     // 标识字段
  int payloadLength;// 传输有效数据长度
  char payload[]; // 数据
}
```

`Packet` 结构体非常简单，各个字段作用都已经注释，此外请注意 `magic` 和 `payloadLength` 都是 `int` 类型，它们在内存中分别占4个字节，而实际传输数据根据 `payloadLength` 来指定，下面来看一个简单数据包在内存中的布局：



网络字节序按从高到低的顺序存储，在网络上使用统一的网络字节顺序，可以避免兼容性问题，换句话说所见即所得，`magic` 值为 `0x4D4D403F`，`length` 值为`0x00000009`，随后跟着的就是有效传输的数据了，这里是“Hey Kuba!”共 9 个字符。

我们发现叹号"!"后面还空着 3 个字节，假设它后面又紧跟着一个数据包，那么会造成如下情况：



这样子我们使用偏移量来访问第二个数据包时，Xcode 会给出运行时警告：

```
typedef struct Packet {
    int magic;
    int payloadLength;
    char payload[];
}Packet;

int main(int argc, const char * argv[]) {
    @autoreleasepool {
        char byteStream[] = {0x77,0x77,0x64,0x63,0x00,0x00,0x00,0x09,'H','e','y',0x00,'K','u','b','a','!',
            0x77,0x77,0x64,0x63,0x00,0x00,0x00,0x0E,'H','o','w',',','s',' ','i','t',','
            ','g','o','i','n','g','?'};

        Packet *p = (Packet *)(byteStream + 17);
        if (p->magic != 0x12233333) { 2 Member access within misaligned address 0x7fff5fbff631 for type 'Packet' (aka 'struct Pac...
            NSLog(@"出错了");
        }
    }
    return 0;
}
```

> 调试终端给出了运行时错误：结构体 `Packet` 成员访问时出现内存未对齐问题，要求是 4 字节对齐。即我们希望第二个包的 `77 77 ...` 从下一个四字节开始。

为了解决该问题，我们需要使用 `__attribute__((packed))` 来标识这个结构体，如下：

```
struct Packet {
    int magic;
    int payloadLength;
    char payload[];
}__attribute__((packed));

typedef struct Packet Packet;

int main(int argc, const char * argv[]) {
    @autoreleasepool {
        char byteStream[] = {0x77,0x77,0x64,0x63,0x00,0x00,0x00,0x09,'H','e','y',0x00,'K','u','b','a','!',
            0x77,0x77,0x64,0x63,0x00,0x00,0x00,0x0E,'H','o','w',',','s',' ','i','t',','
            ','g','o','i','n','g','?'};

        Packet *p = (Packet *)(byteStream + 17);
        if (p->magic != 0x12233333) {
            NSLog(@"出错了");
        }
    }
    return 0;
}
```

现在我们访问第二个 `magic` 字段方式也需要作出改变，这里借助 `memcpy` 来实现：

```
int magic;
memcpy(&magic, byteStream + offsetof(struct Packet, magic), sizeof(int));
if (magic != ...)
```

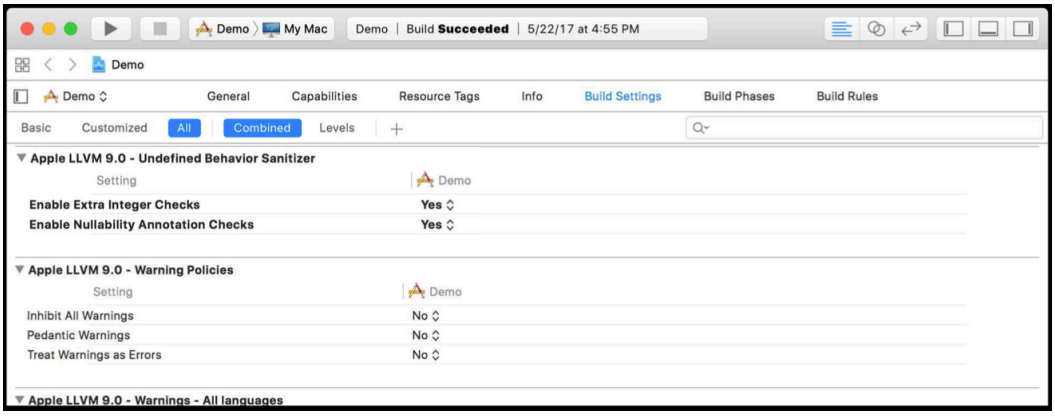
2.4.3 非空返回值问题(*Nonnull Return Value Violation*)

OC 开发中我们会标识某个函数的返回结果为 `nonnull` 关键字，但实际上函数存在返回 `nil` 的情况，这对于 Swift 和 OC 混编项目来说，存在潜在的崩溃危险，因此必须对此检查是非常有必要的！

先看下如下例子：

```
@implementation SolarSystem
+ (nonnull NSDictionary *)planetMoons {
  return @{@"Earth": @[@"Moon"],
        @"Mars":@[@"Phobos",@"Deimos"],
        //...
        @"Pluto":@[@"Charon", @"Hydra", @"Nix", @"Kerberos", @"Styx"]
  };
}
- (nonull NSArray *)moonsOfPlanet:(nonnull NSString *)planet {
  return [[self class] planetMoons][planet];
}
@end

// 调用
NSMutableArray *biggestMoons = [NSMutableArray new];
[biggestMoons addObject:[solarSystem moonsOfPlanet:@"Pltuo"][0]]; // 注意这里 Pluto 拼成了 Pltuo
```

关于 `Extra Integer Checks` 以及 `Nullability Annotation Checks` 你需要前往 **Build Settings** 设置选项为 **YES**:



现在运行代码，Xcode 将对非空返回情况进行检查：

3. 总结

随着 Xcode 9 加入更多运行时工具，定位和排查问题更加简单和精准，此外 Xcode 9 的提示信息也有所改变，描述更加清晰易读，好感度顺便提升。

本文是我对 session 406 的简单总结，希望大家喜欢。

4. 参考

> Session 406 视频
>
> Address Sanitizer 官方文档
>
> Thread Sanitizer 官方文档
>
> Undefined Behavior Sanitizer 官方文档
>
> Presentation Slides (PDF)

在 WKWebView 中自定义加载内容

WKWebView 在 iOS 11 中主要增加了三个新功能：

- 管理 cookie
- 过滤网页内容
- 拦截自定义的资源请求

管理 cookie

通过 `WKHTTPCookieStore` 可以对 cookie 进行以下操作：

- 添加、删除自有的 cookie
- 访问在 WKWebView 中可见的所有 cookie
- 包括只支持 HTTP 请求的 cookie
- 观察 cookie 的存储变化

获取 WKHTTPCookieStore

通过`websiteDataStore`获取只读的`cookieStore`：

```
let cookieStore = webView.configuration.websiteDataStore.httpCookieStore;
```

添加、删除 cookie

调用`setCookie`方法添加：

```
let cookie = HTTPCookie(properties: [ HTTPCookiePropertyKey.domain:
"canineschool.org", HTTPCookiePropertyKey.path: "/",
HTTPCookiePropertyKey.secure: true,
HTTPCookiePropertyKey.name: "LoginSessionID", HTTPCookiePropertyKey.value:
"5bd9d8cabc46041579a311230539b8d1"])

cookieStore.setCookie(cookie!) {
    webView.load(loggedInURLRequest)
}
```

删除则调用`delete`方法。

因为 cookie 的设置由独立的线程设置，所以设置成功后 WebKit 会有一个回调。

获取所有 cookie

通过`getAllCookies`方法获取：

```
cookieStore.getAllCookies() { (cookies) in
    for cookie in cookies {
        // Find the login cookie
    }
}
```

过滤网页内容

现在 WKWebView 也支持 Safari Extension 推出的 Content Blocker 功能。主要能够实现以下的功能：

- 阻止加载请求

- 设置某些内容不可见
- 把不安全的 HTTP 请求替换成 HTTPS 请求

使用方式是提供一个拦截规则（JSON 格式）给 WebView，规则包括触发规则 trigger 和对应的 action：

```
[{
"trigger": {
"url-filter": ".*" },
"action": {
"type": "make-https"}
}]
```

具体触发规则和对应的操作可以参照这篇文章：WKWebView内容过滤规则详解。

WebKit 会把拦截规则编译成高效的二进制码。使用方法如下：

```
WKContentRuleListStore.default().compileContentRuleList(
   forIdentifier: "ContentBlockingRules",
  encodedContentRuleList: jsonString) { (contentRuleList, error) in
    if let error = error {
         return
    }
  let configuration = WKWebViewConfiguration()
  configuration.userContentController.add(contentRuleList!)
}
```

拦截自定义的资源请求

WKWebView 现在支持拦截自定义 scheme 的请求。比如 https://www.apple.com , file:///Users/Alex/demo.txt，https、file 就是 scheme。如果我们要拦截 WK 中的请求，就要先把页面里的内容的 scheme 配错成非 HTTP 标准的 scheme。比如配置成 apple-local://img/banner.png。然后我们再注册”apple-local“的 scheme handler 就可以自己拦截处理这个请求了。

在注册时同时需要传入实现 `WKURLSchemeHandler` 协议的 handler 实例：

```
class MyCustomSchemeHandler : NSObject, WKURLSchemeHandler {

    func webView(_ webView: WKWebView, start urlSchemeTask: WKURLSchemeTask)
{
    }

    func webView(_ webView: WKWebView, stop urlSchemeTask: WKURLSchemeTask) {
    }
}

let configuration = WKWebViewConfiguration()
configuration.setURLSchemeHandler(MyCustomSchemeHandler(), forURLScheme:
“apple-local”)
```

自定义返回数据

通过 `WKURLSchemeHandler` 的回调函数中的参数 `WKURLSchemeTask` 来实现自定义的返回数据。

`WKURLSchemeTask` 的定义如下：

```
protocol WKURLSchemeTask : NSObjectProtocol {
    var request: URLRequest

    func didReceive(_ response: URLResponse)
    func didReceive(_ data: Data)
    func didFinish()
    func didFailWithError(_ error: Error)
}
```

返回数据的操作方式和 URLProtocol 非常相似，调用 `didReceive` 返回数据，最后调用 `didFinish` 表示处理结束：

```
func webView(_ webView: WKWebView, start urlSchemeTask: WKURLSchemeTask) {
    let resourceData = createHTMLResourceData()
    let response = ...

    urlSchemeTask.didReceive(response)
    urlSchemeTask.didReceive(resourceData)
    urlSchemeTask.didFinish()
}
```

参考

Customized Loading in WKWebView

使用 CloudKit 控制台构建更好的应用

引言

自 CloudKit 首次在 iOS 8 发布以来，苹果公司每年都会推出不少更新。CloudKit 为开发者提供了后端服务，让开发者专注于 App 前端的开发，节省开发者的成本。

本文为你介绍如何借助全新的 CloudKit 控制台开发更优秀的 App。

这次，苹果公司彻底整修了 CloudKit 控制台，新功能贯穿整个开发周期：开发、测试、操作、支持 CloudKit 服务等等，获取数据变动记录，修改分享关系，获取实时的服务器日志，包括所有用户的事件，甚至是推送通知，更方便进行代码调试和为用户提供技术支持。

Baas - Backend as a Service

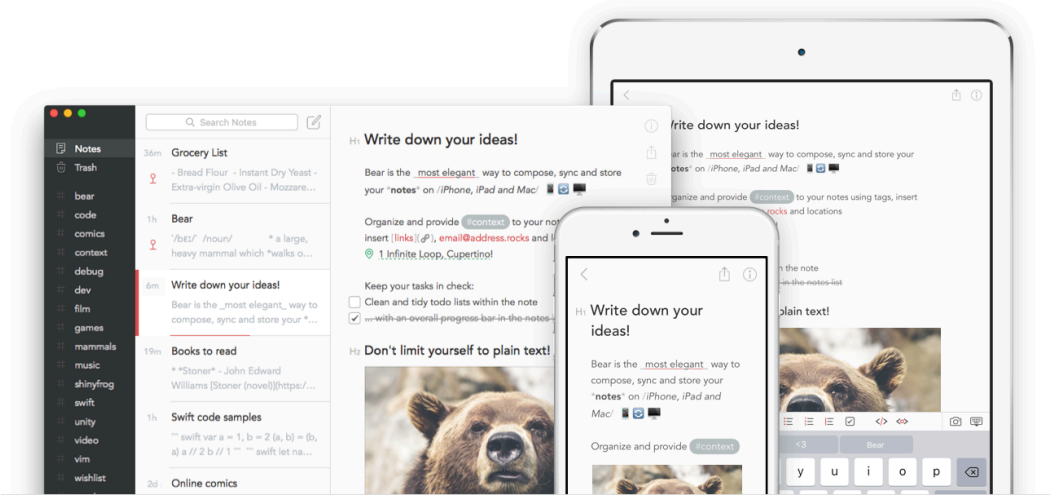
国内目前比较出名的 BaaS 服务商是 LeanCloud。

Parse 虽然在 2017 年 1 月 28 日停止了服务，不过很多公司提供了 Parse 托管服务，例如：SashiDo.io、Qursky、Back4App。

苹果借助 CloudKit进军 Baas 领域，还推出了 CloudKit JS 框架，借助该框架，开发的 web 应用可以获取服务器上的数据。

例子

苹果自己的产品：Photos, iCloud Drive, Notes, News, 和 Safari 等。这些 App 能够多终端实时同步数据。当然了，也可以只在某个设备中存储，不进行多终端同步，需要额外进行设置。



Bear 的多终端实时同步技术，使用的就是 CloudKit，将用户的数据存储在 Private Database里，属于 Pro 付费功能，如果你的 iCloud 5G 免费空间已经用完，购买了 Bear 的该功能，也无法实时同步，不过数据不会丢失，会被存储在本地。也就是说，Bear 有自己的本地数据持久化方案，但同时，也支持 CloudKit 存储数据以及多终端实时同步功能。

小结

CloudKit 使用用户的 iCloud 存储空间，也就是说，私有数据是存储在用户的 iCloud 中，占用用户的 iCloud 存储空间额度。而对于公共空间，每个 App 的免费空间和额度很多，足够开发者使用了。CloudKit 费用计算器，可以计算 App 的费用。能为开发者节省不少开支。

在用户隐私和数据安全方面无需开发者担心，数据都存储在 iCloud 上，有苹果罩着呢。不过开发者无法看到用户的私有数据（Private Database），只能看到共享的（Shared Database）和公开的（Public Database）数据。

Cloudkit 的最大的优势就是够在苹果所有的系统上使用，多终端数据同步。

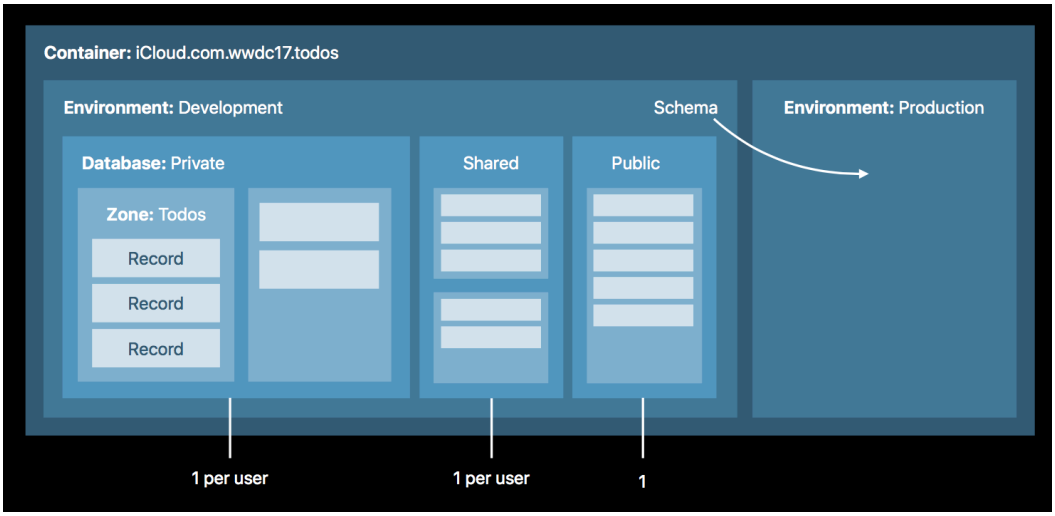
更新 CloudKit 控制台的初心

- 在 App 开发的整个生命周期中都提供了所需的工具。
- 能够在控制台直接查看、体验所有的 API。
- 能够在控制台直接查看所有用户和系统的事件。
- 更好的理解集合行为（aggregate behavior）以及背后的通信机制。

接下来，在演示控制台的操作之前，先介绍以下两个方面，之后再进行演示：

1. 快速浏览一下 CloudKit 的一些核心概念。
2. 展示一个基于 CloudKit 的示例 iOS App：Todo List App，以便于接下来演示如何使用控制台进行测试和开发。

CloudKit 核心概念



上图是 CloudKit 结构化数据关系图，为开发者提供了面向对象的、海量的、无需创建数据表结构即存即用的存储能力。

下表是关系型数据库、MongoDB 和 LeanCloud 、CloudKit 的对应术语：

RDBMS	MongoDB	LeanCloud	CloudKit
Database	Database	Application	Container
Table	Collection	Class	Zone
Row	Document	Object	Record
Index	Index	Index	Index
JOIN	Embedded, Reference	Embedded Object, Pointer	Reference

从上面的表格中科院发现，CloudKit 的 Environment 和 Database，没有出现在对比表格中，会在下面解释。

Record

键值对形式存储一条 Record，值的类型有：String，Int，Double，Asset（二进制文件）。每条 Record，都有自己的 Zone。

Zone

Zone 里有很多 Record。CloudKit 有默认的 Zone，如果不够，也可以创建更多的 Zone。每个 Zone，都有自己的 Database。

Database

Cloudkit 提供了 3 种类型的 Database，各自的数据查看权限不同，分别为：Private Database（私有数据库），Shared Database（共享数据库） 和 Public Database（公开数据库）。

每个用户都有自己的私有数据库，该数据库中的数据只有用户自己可以查看，其他人包括开发者也没有权限查看。

去年苹果公司推出了 CloudKit Sharing 功能，iOS 自带的 App：便签 App 中，可以分享某条便签给其他人，就是使用了该功能。被分享的数据就存储在共享数据库中。

最后是公共数据库，每个人都可以在该数据库中进行读写。

看到这里可能会有疑问，如果我看不到私有数据库里的数据，那么如何开发？如何调试？如何解决 bug 呢?

实际上，开发的时候，可以在设备的 iCloud 上登陆自己的苹果开发者账号，这样就可以在控制台中看到自己的数据了。开发者环境还是生产环境，都可以看到自己开发者账号的数据。就是说，登陆 CloudKit 控制台的账号，要和测试设备中的 iCloud 账号一致才行。

环境和数据架构

以上所有的数据和数据库，都在某个环境（Environment）中：Development（开发环境） 和 Production（生产环境）。在开发的时候，使用开发环境，定义数据模式、Record 的类型，要存储的数据的类型，以及对应的 index，完成数据库架构（Schema）。

App 上架后，使用的是生产环境。所以记得将开发环境部署到生产环境里。

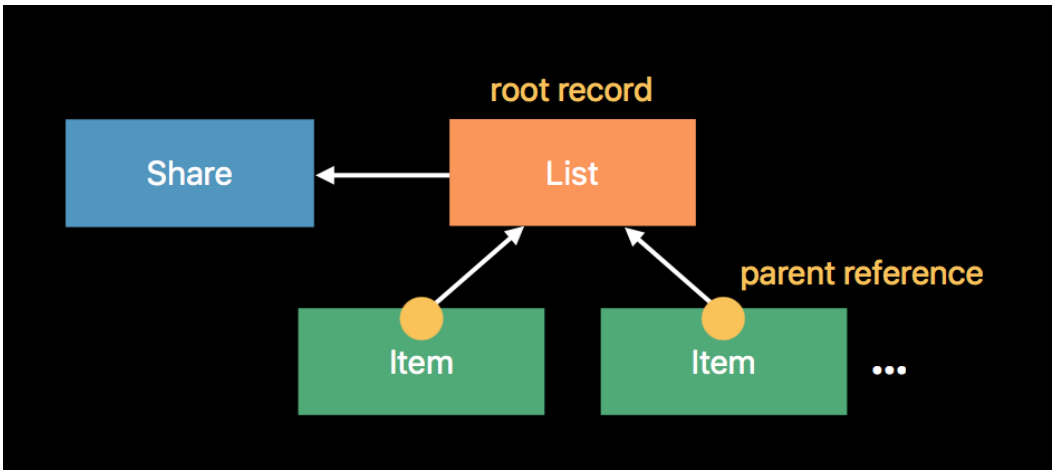
在生产环境的中，用户们有各自的的私有数据库、共享数据库和一个公开数据库，完全和开发环境隔离的，你在开发环境中做的操作，不会影响生产环境，除非你手动将新的数据模型部署到生产环境中。

Container

这算是最高层级的容器了，一个 App 对应一个 Container。不过我谷歌了一下，如果你想让多个 App 都访问同一个 Container，也是可行的（尚未实测）。想让一个 App 访问多个 Container，也可行（尚未实测）！所以，具体看你需求了。一般简单的小 App，用一个 Container 就绰绰有余了。

示例：Todo List App

Data model 数据建模



从上图中可以看出，该 App 的数据模型相对简单，用户可以创建代办清单，每个清单中可以创建要代办的事项（Item）。要用 CloudKit，Record 是必须的基本要素。在这里，创建一个名为 List 的 Record。每条 List Record 都有对应的 Items，从属关系为 Reference。

同时，还希望该 App 可以将每个代办清单分享给其他人，这里使用 CloudKit Sharing 接口。可以定义谁可以看这条记录，他们的读写权限是什么。如果想了解更多关于 CloudKit Sharing 的知识点，请参考 WWDC 2016 的 What's New with CloudKit 。

获取数据变更的 API

下面说一下如何使用 API、如何和服务器进行通信。

简单说一下该 App 的需求：

1. 用户在 iPhone 上创建代办清单时，该用户的其他安装该 App 的设备也能同步信息；
2. 如果用户将某条数据在某些人之间进行了分享，其中一人编辑后，能看到编辑后的信息；
3. 使用 CoreData 进行本地的数据持久化，将所有的数据本地存储，以便能快速读写，在没有网络时也能使用该 App；
4. 每次数据有了变动，且在有网的情况下，这些数据可以返回到服务器上。

推送变动通知 - `CKDatabaseSubscription`

在服务器上为用户的私有数据库创建 CKDatabaseSubscription，只要私有数据库的数据发生了改变，服务器就给用户的所有设备推送通知。由于我们使用了分享功能，所以还需要订阅共享数据库中的数据变动。只要服务器上的数据有了变动，就会推送通知。

- `CKDatabaseSubscription` (私有数据库)
- `CKDatabaseSubscription` (共享数据库)

获取变动信息 - `CKFetchDatabaseChangesOperation`

通过上一步操作，目前设备能收到变动通知了，那么收到变动通知后，我们得需要知道具体是哪些数据发生了变化，这就需要用到 `CKFetchDatabaseChangesOperation`。需要知道是哪些数据库以及数据库中的哪些 Zone 发生了数据变动：

- `CKFetchDatabaseChangesOperation` （查询数据库中的数据变动）
- `CKFetchRecordZoneChangesOperation` （查询 Zone 中的数据变动）

如果想了解更多信息，请参考 WWDC 2016 的 CloudKit Best Practices。由于该操作太常用了，还专门在文档中添加了一节：Maintaining a Local Cache of CloudKit Records，提供了流程介绍和代码示例。

Demo - 在 CloudKit 控制台获取数据变更信息

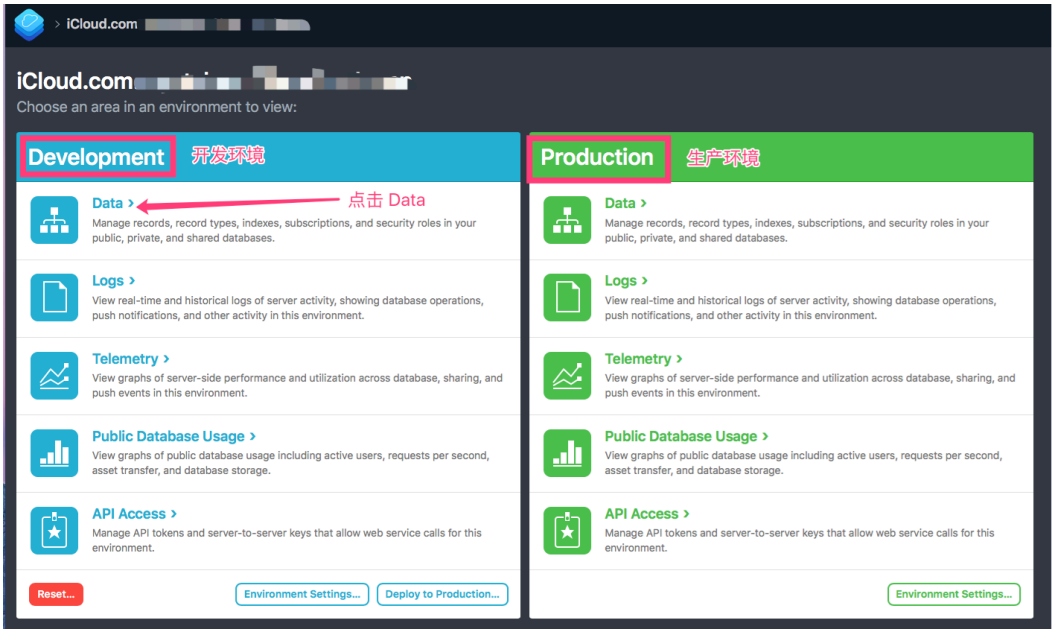
之前获取数据变动的操作（也就是上面刚刚讲过的这个点），只能通过代码实现，现在也能通过控制台来查看了。现在，开始体验大整修后的控制台吧~

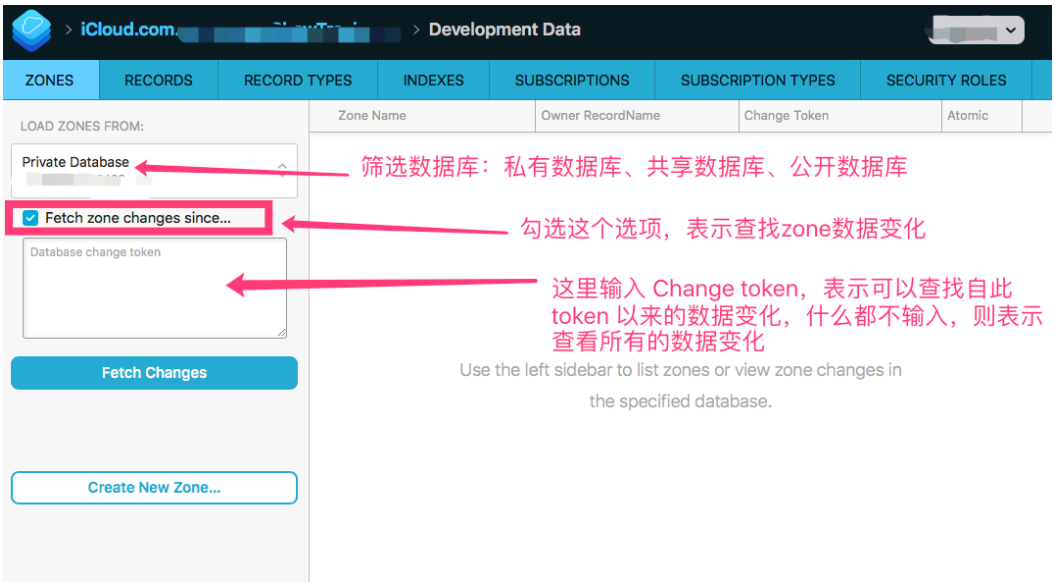
控制台首页

在首页，可以看到：

1. 自己所属的开发团队（可以同时属于多个开发团队）；
2. 对于每个开发团队，可以看到正在开发的项目，也就是 Container。之前提及过，一般情况下，一个 App 对应一个 Container。
3. 能看出哪些 Container 处于开发环境，哪些 Container 已经部署到了生产环境。
4. 搜索框，可以过滤检索团队和项目。

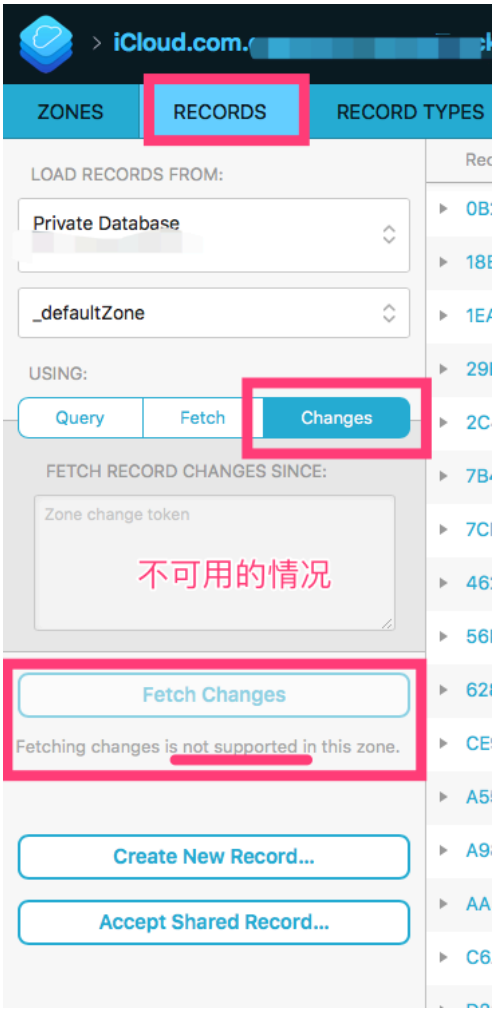
点击该 App 的 Container 后可以看到两个分类：开发环境和生产环境，点击生产环境的 Data 模块后，进入名为 Zones 的 Tab 页。





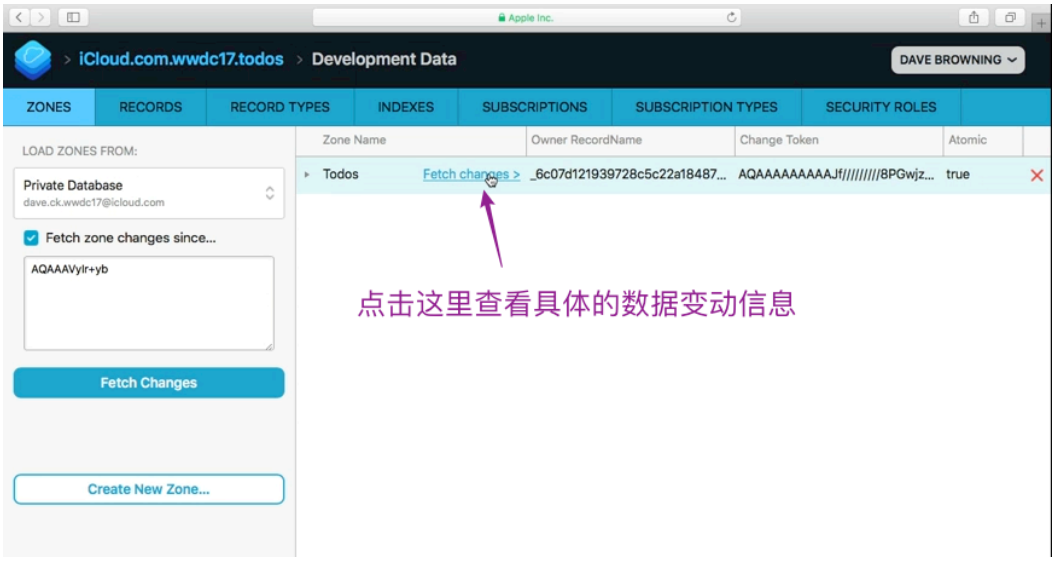
按上图所示，选择私有数据库（也可以选择其他的数据库），勾选 Fetch zone changes since... 选项，这样就可以查看变动了。

如果这里你查不到数据，看一下设备的 iCloud 账号是不是你的开发者账号，如果是同一个账号，却查不到变化。那么可以点击名为 RECORDS 的 Tab 页，如下图：

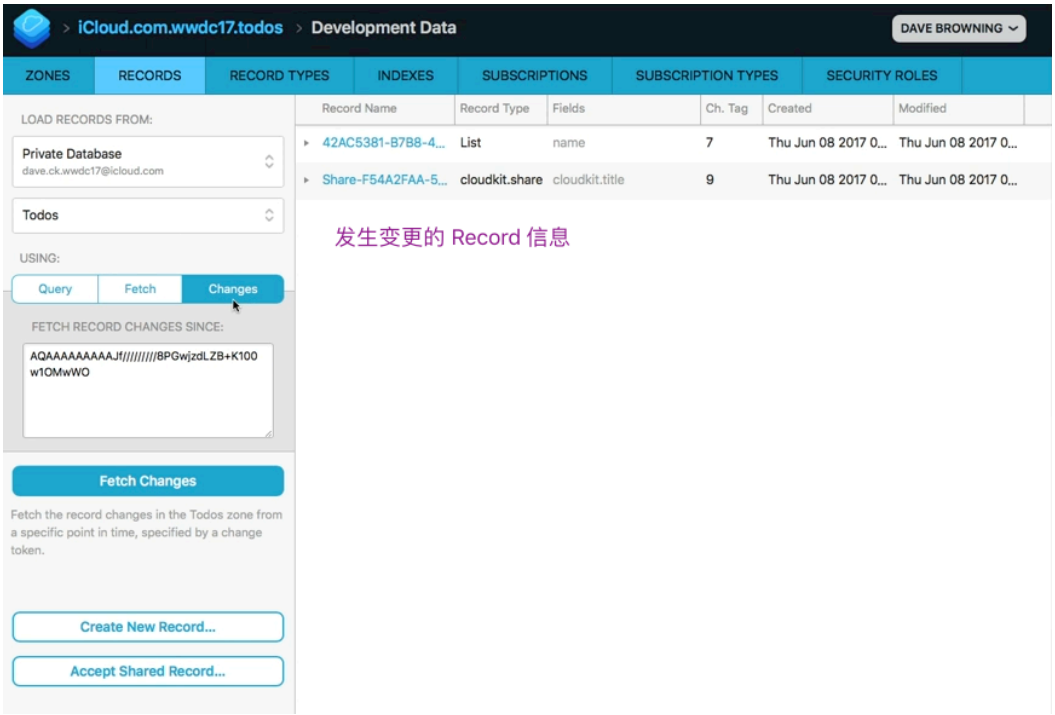


这里不可用，说明你未支持查询变动，检查一下上面刚刚提及的两个获取数据变更的 API 是否已经实现。

如果操作无误，差点查询变更，会出现如下图所示的结果（需要你先在设备上创建或者变更一条数据，才能在控制台中看到）：

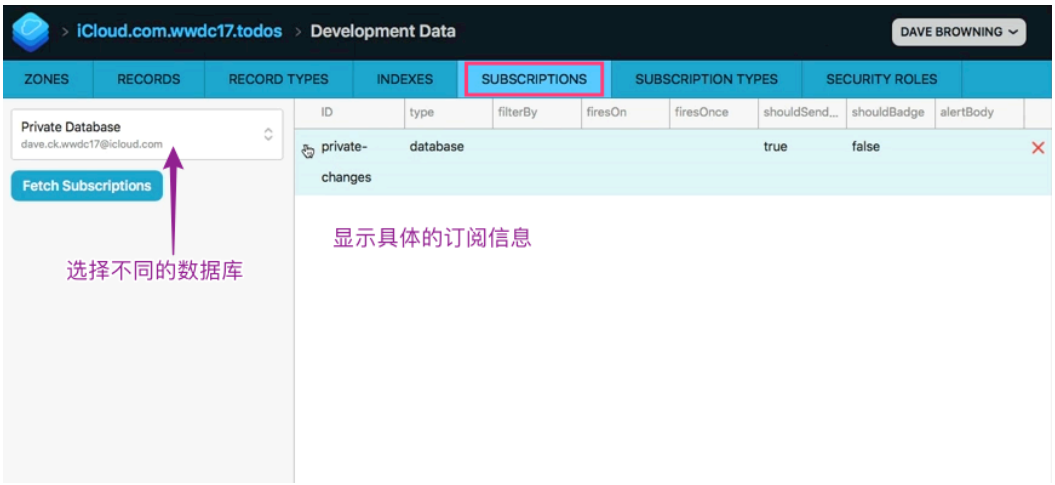


点击之后，可以看到：



尝试着点击列出的 Record，在右边弹出的详细信息框中，尝试改一些东西，设备上回立即同步变更。

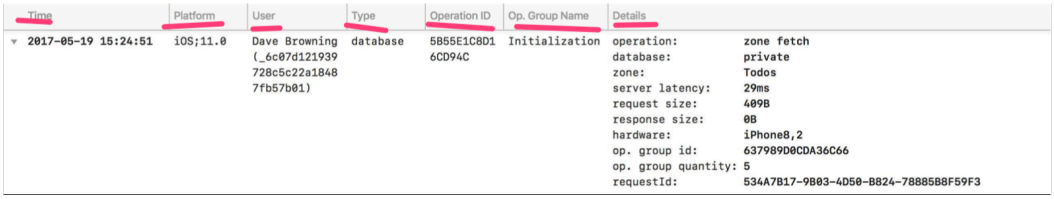
Demo - 在 CloudKit 控制台获取订阅和推送通知



如上图所示，这是查看订阅（subscription）的 Tab 页。

服务器的事件日志

上面又是查询数据变更（fetch changes），又是推送通知（push notification），还有订阅，还要设计多个设备和平台。要是又一个地方，可以看到这些所有的事件就好了，所以，这次控制台新增了事件日志功能（Log Event）。



如上图所示，可以查看所有的日志记录。

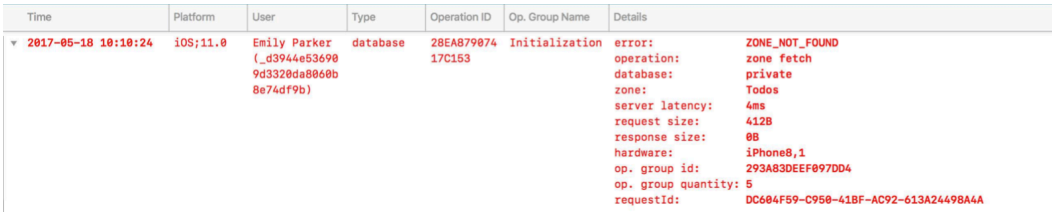
Log Event 的组成要素：

1. Time - 该事件发生的时间。
2. Platform - 平台信息，如 MacOS、iOS、tvOS，watchOS，web。
3. User - 如果开发团队中人，可以看到名字，如果是普通用户，则只能看到 CloudKit 的用户名 ID，你看不到用户的 Apple ID，这样可以保护用户的隐私。
4. Type - 如：database, push, sharing。
5. Operation ID - 为每次操作自动生成的 ID。
6. Op. Group Name - （Operation Group Name）之后会详细讲述。
7. Details - 所有的详细信息。

恩，这个日志的信息量非常大，也是为了帮助开发者调试 bug，更好地对用户进行技术支持。

错误日志：

错误日志的字体颜色是红色的，以区别于一般的日志。可以查看具体的错误类型。



Operation Group

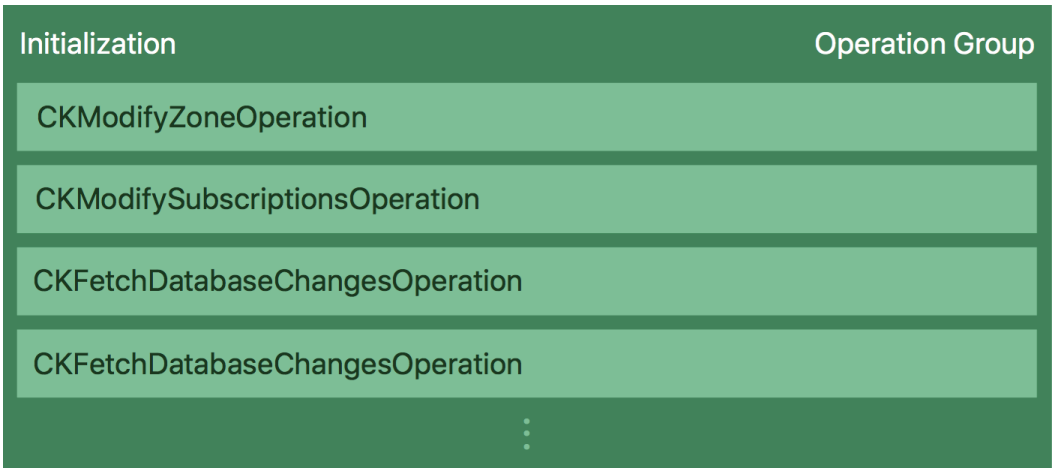
之前多次提到的 Operation Group 到底是什么呢?

Operation Group 是基于应用的一组操作逻辑。英文字面的意思是操作组合。

这里听起来怎么有点 Workflow 的感觉？似曾相识。

该功能在 iOS 11 中发布，低于该版本无法使用。

举个例子，以上面的代办清单 app 为例，详见下图：



在第一次启动 App 时，需要做上图中的这四个事情，那么这四个事情，就可以放在一起，组一个 Operation Group。

苹果是如何使用 Opration Group 的？请见下图：

自动备份、手动备份、设置相册、缩略图下载、获取电影数据流、iCloud 同步等等，都使用了操作组合。

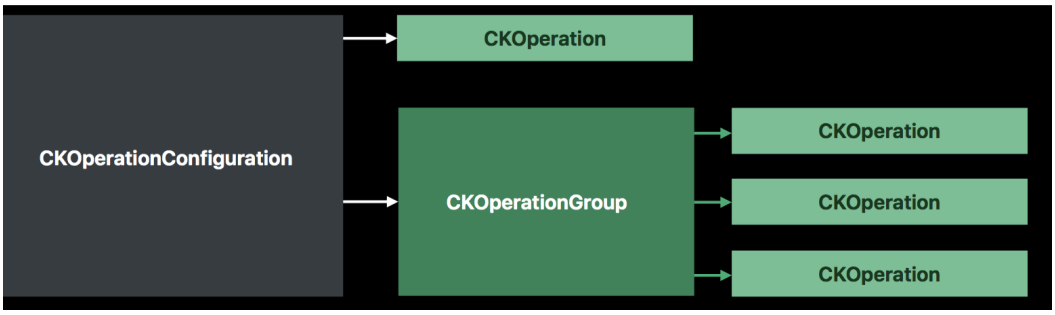
下面说一下操作组合的 API，从视频的 22:40 开始，到 25:10 结束。

具体代码如下。

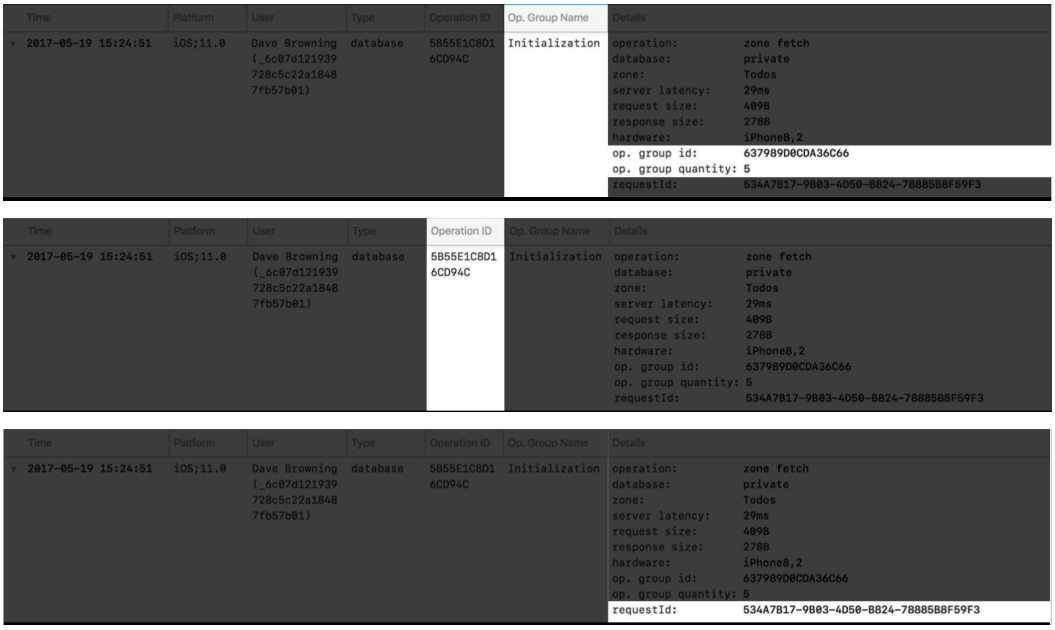
```
open class CKOperationGroup : NSObject, NSSecureCoding {

     // 该 ID 由系统提供
     open var operationGroupID: String { get }

    // 对组合操作进行设置
    @NSCopying open var defaultConfiguration: CKOperationConfiguration?

    // 该操作组合的名字
    open var name: String?
    // 数量。比如在备份文件时，设计要备份多少份文件；在相册中下载缩略图时，要下载多少缩略图。
    open var quantity: Int

    // 告知客户端，你估计大概有多少数据需要在客户端和服务器之间传输，主要类型，是一个枚举，列出来大概的几种情况
    // 不需要十分精确，估算一个大概的传输数据即可
    open var expectedSendSize: CKOperationGroupTransferSize // 枚举
    open var expectedReceiveSize: CKOperationGroupTransferSize

}

open class CKOperation : Operation {
     // 实现某个操作组合
     open var group: CKOperationGroup?
}


// 组合操作的设置
open class CKOperationConfiguration : NSObject {
    open var container: CKContainer?
    open var qualityOfService: QualityOfService
    open var allowsCellularAccess: Bool
    open var isLongLived: Bool
    open var timeoutIntervalForRequest: TimeInterval
    open var timeoutIntervalForResource: TimeInterval

}
```

`CKOperationGroupTransferSize` 枚举：

```
public enum CKOperationGroupTransferSize : Int {

    case kilobytes
    case megabytes
    case gigabytes
    case tensOfMegabytes
    case tensOfGigabytes
    case hundredsOfMegabytes
    case hundredsOfGigabytes
    case unknown

}
```

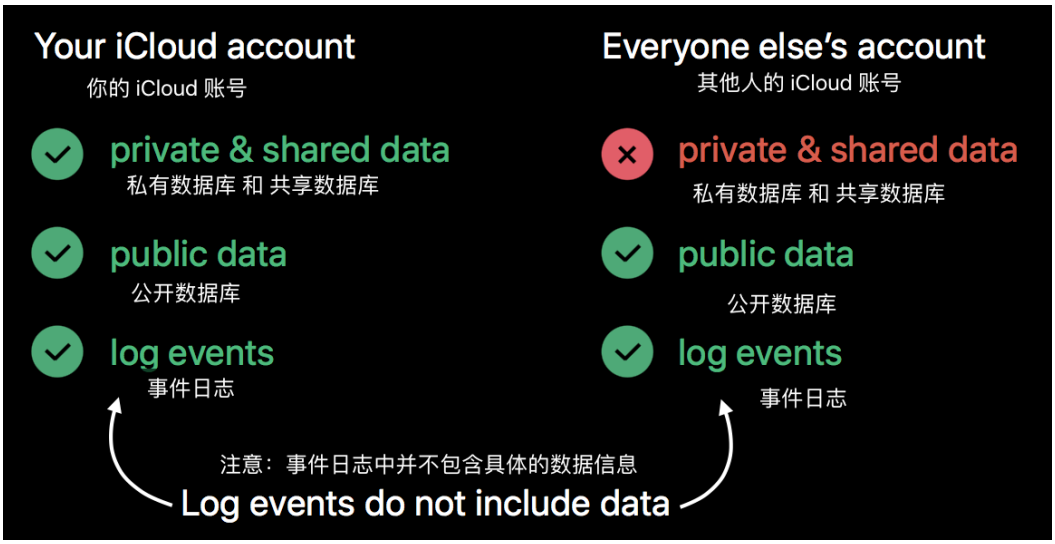


Operation Group 在事件日志中的展示请见下图：

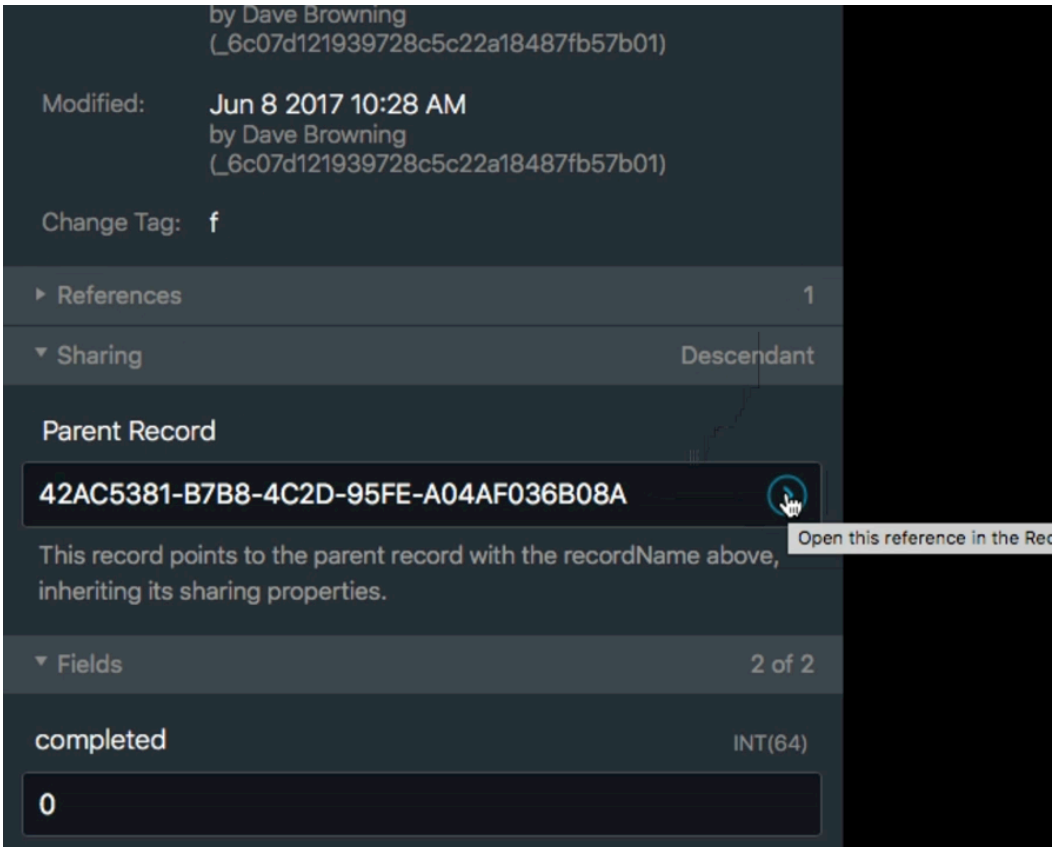


用户隐私

苹果非常注重用户隐私，但是如何在保护隐私和进行开发调试中取得平衡呢？毕竟要是真的看不到私有数据库的一些数据，想调试解决 Bug 还是很难吧？

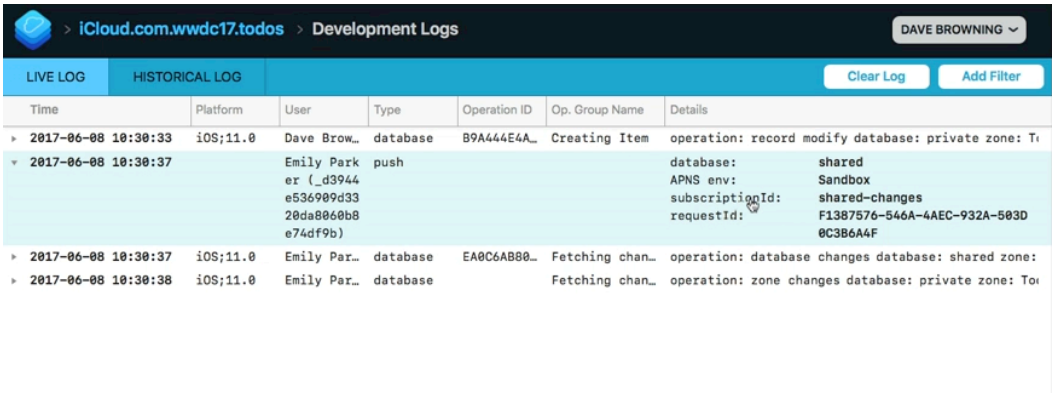


Demo - CloudKit 控制台中的分享

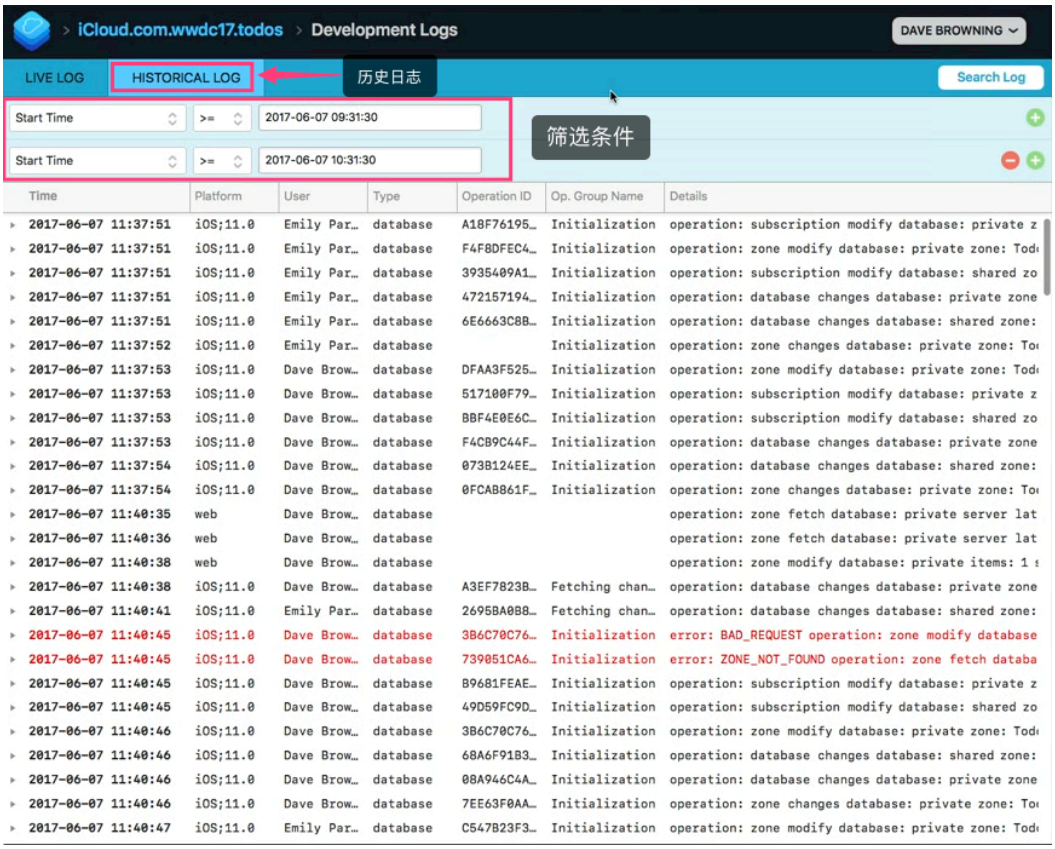


这里可以看到该条数据是否为分享的数据，更方便的查看 Reference 关联的各个数据。

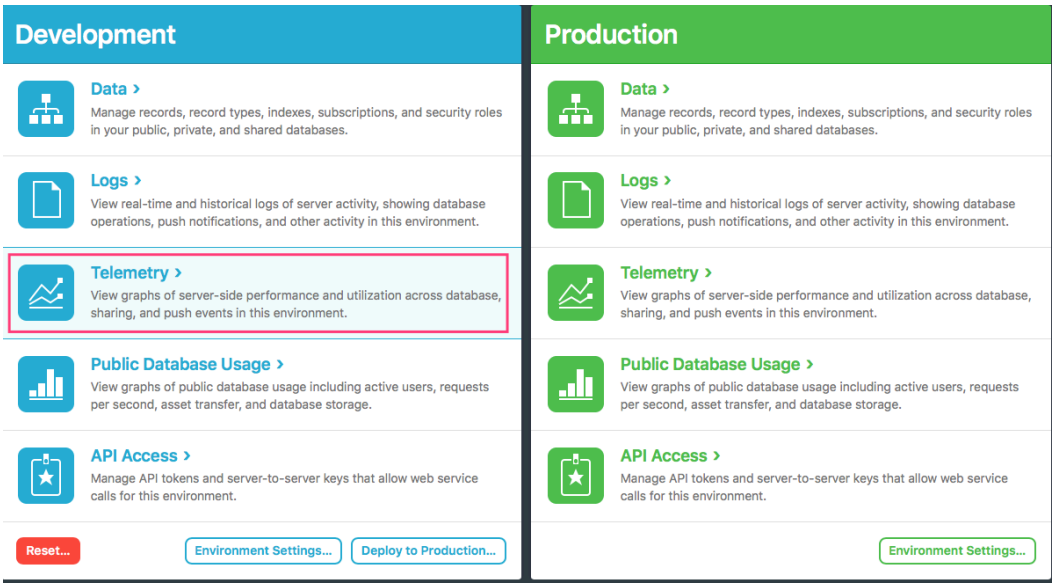
那么在事件日志里，是如何记录显示分享功能的信息呢？请见下图：



这里再补充一个功能，如下图所示，还能查看历史日志。

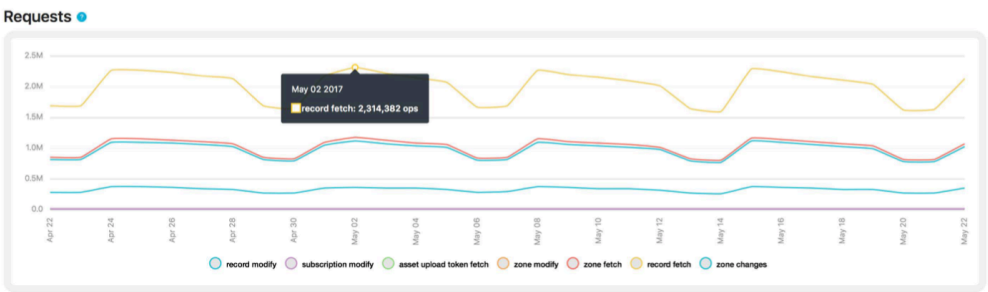


Demo - CloudKit 控制台中的 Telemetry

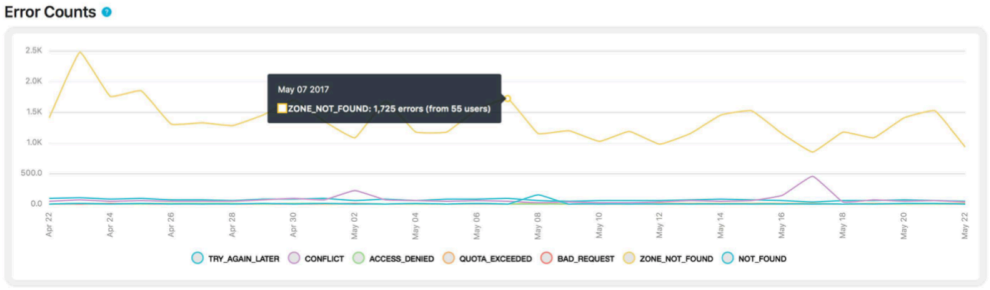


数据统计分析功能比之前进步多了，可以进行多种查询和筛选：

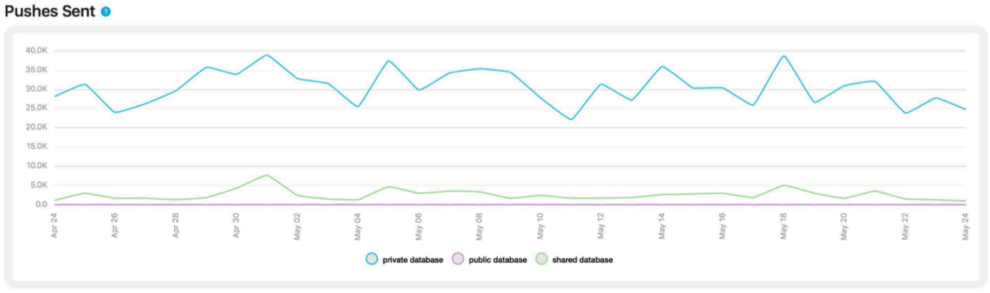
1. 统计检测与数据变动相关的信息；



2. 检测错误；



3. 核实推送信息；



总结

本 Session 主要讲述了2个事情：

1. 全新的 CloudKit 控制台
2. 全新的 CKOperationGroup API，操作组合 API。

如果你对 CloudKit 有什么意见和建议，欢迎你的反馈邮件：cloudkit@apple.com!

相关视频

WWDC 2017

- Advances in Networking, Part 1
- Advances in Networking, Part 2

WWDC 2016

- CloudKit Best Practices
- What's New with CloudKit

WWDC 2015

- CloudKit JS and Web Services
- CloudKit Tips and Tricks
- What's New in CloudKit

更多资料

- CloudKit
- CloudKit Catalog
- CloudKit JS
- iCloud and CloudKit for Developers
- Maintaining a Local Cache of CloudKit Records
- HD Video
- SD Video
- Presentation Slides（PDF）

水平有限，如有疏漏不妥之处，还请不吝赐教。联系信息：sing8796185@163.com

App 与不断发展的网络安全标准

现在，我们越来越关注用户的隐私和信息安全，但是随着时间的推移，计算机变得越来越高效，那些年代久远的网络协议和加密算法，在新的硬件和攻击方式下，变得岌岌可危。

针对网络安全，我们要怎样做?

- 时刻关注 App 的安全，保证 App 的及时更新
- 了解业界的发展变化，跟进最新的标准、学术研究和工业界的实践
- 确保集成的第三方库可以得到及时的更新，避免安全隐患
- OS 移除安全隐患
- 使用 ATS（App Transport Security），ATS 会强制在 App 中使用最佳实践
- 相比被攻击后的严重后果，前期投入一定的维护成本是值得的

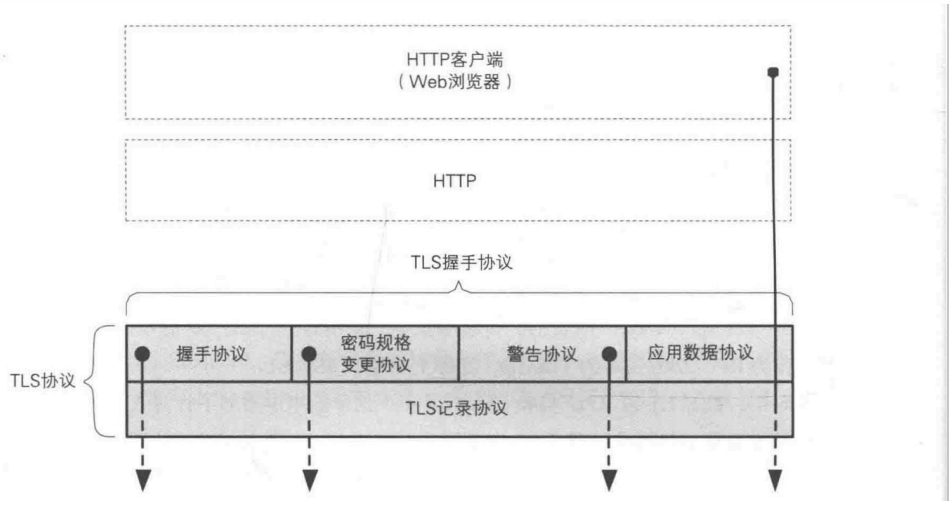
常见的网络攻击

这里总结了一些常见的网络攻击，它们针对的对象有所不同，有的是想窃取数据，有的是想伪装身份，它们在图中以颜色进行区分。我将会通过一些实践来告诉你，如何去避免这些攻击，具体来说就是关于加密（encryption）、密码散列（cryptographic hashes）、公共密钥（public keys）、网络协议（protocols）、证书吊销检查（revocation）等内容的一些实践。

TLS 协议

现在大多数的 App 网络都已经切换到 HTTPS 了，HTTPS 其实就是在 HTTP 下加入了 TLS 层，TLS 是保证网络安全的基础，它可以对传输内容加密、进行身份验证、避免传输内容被篡改。

TLS 协议是由 TLS 握手协议和 TLS 记录协议两部分组成，TLS 记录协议负责对传输内容加密，TLS 握手协议又分为四个子协议，负责协商加密算法、共享密钥、传达警告等工作。



TLS 协议中涉及到多种不同的密码技术，比如在握手协议中使用到了非对称加密、散列函数、数字签名技术，在 TLS 记录协议中使用到了对称加密、散列函数等技术。具体的这些技术，在下面会详细介绍，并且说明哪些技术已经过时了，需要更换更安全的加密算法来保证安全。

密码技术介绍

本文中，涉及到一些加密算法，根据密钥的使用方式，可以分为对称加密和非对称加密：

- **对称加密：**加密和解密的过程中使用的是同一种密钥。常用的加密算法有DES、AES等。
- **非对称加密：**又叫做公钥加密，在加密和解密的过程中使用不同的密钥。常用的加密算法有RSA、ECC等。

除了上述两种加密算法，还有一些用来确保信息完整性和身份认证的技术：

- **密码散列算法：**它不是一种加密算法，而是用来确保信息未被篡改。密码散列函数会根据信息的内容计算出一个散列值，这个散列值就被用来检查信息的完整性。
- **数字签名：**信息的发送者用私钥加密，接收者使用公钥解密。用私钥加密的密文只能由对应的公钥来解密，所以可以将这个过程叫做数字签名，用来身份认证。
- **证书：**经过 CA 机构数字签名后的公钥证书，用于身份认证。

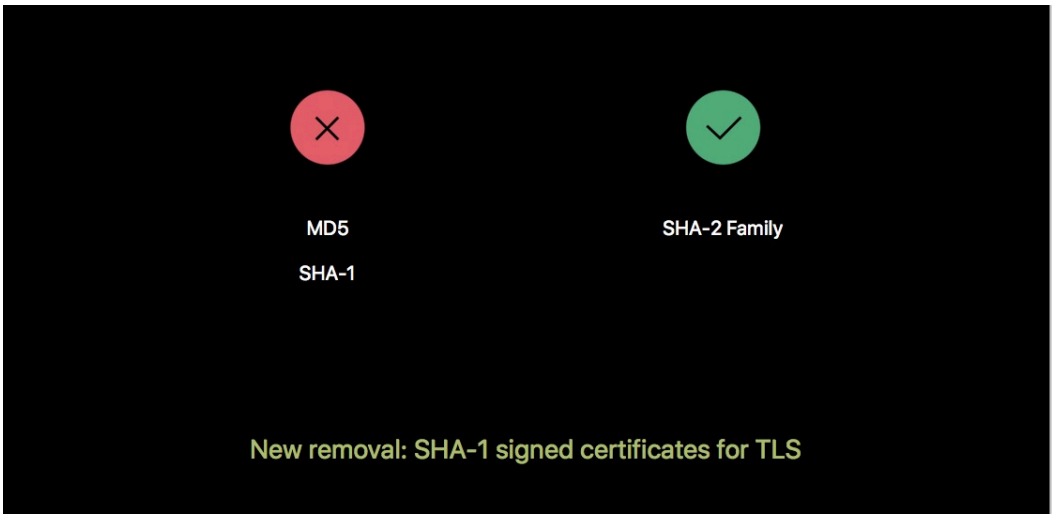
加密

加密是我们众所周知的用来防止信息被窃听的手段，但是一些加密算法已经非常不安全，很容易被攻击者攻破。特别是 RC4， 目前可以在短短三天之内就可以攻破它。而且，Triple-DES 和 AES 加密算法在 BEAST 和 Lucky 13 这些攻击面前，也无法保证安全性。苹果正在计划在所有平台上弃用 Triple-DES 和 RC4，同时也不推荐使用 AES-CBC 算法，但是还没给出具体的日期。Apple 推荐我们使用 AES-GCM 或者 ChaCha20/Poly1305 算法。

密码散列

密码散列会有一个输入值和一个输出值，输入的称为消息，输出的称为散列值。密码散列会根据输入的内容计算一个散列值出来，这个散列值可以用来判断信息是否完整。但是其中一些密码散列已经不安全了，黑客可以通过碰撞攻击的方式来篡改数据。碰撞攻击是通过大量的尝试，来找到不同的输入值，但是输出值是一样的情况，进而篡改数据。因为篡改后的散列值并没有发生变化，所以你无法判断数据是否被修改过。

MD-5 和 SHA-1 已经不安全了，Apple 已经在前几年在移除了 MD-5 算法的签名证书。SHA-1 在今年的早些时候，也遭遇了攻击。所以，Apple 将也不再信任 SHA-1 算法的签名证书。为了保证绝对的安全，开发者应该使用 SHA-2 算法。



公钥密码

一般情况下，经过公钥加密的内容，只能通过私钥打开加密内容。但是，768 位的 RSA 在2009 年被攻破，Apple 在 2016 年春天已经移除了对 1024 位以下 RSA 签名的证书的支持。由此来看，对 1024 位 RSA 加密的攻击也快要来到了，所以 Apple 也宣布将不再对 2048 位以下 RSA 签名的证书信任。你应该使用 2048 位以上的 RSA 加密，或者其他 Apple 平台信任的椭圆曲线密码。

协议

如果开发者正在使用HTTP协议，那就代表着你传输的内容，任何监听者都可以获取到。同时，一些老化的 TLS 版本，也是不安全的，比如 SSL 3.0、TLS1.1 和 TLS1.0。应该避免使用这些协议，开发者应该使用基于 TLS1.2 的 HTTPS 协议。

同时，Apple 宣布发布 TLS 1.3 Beta 版，后续会详细讲。

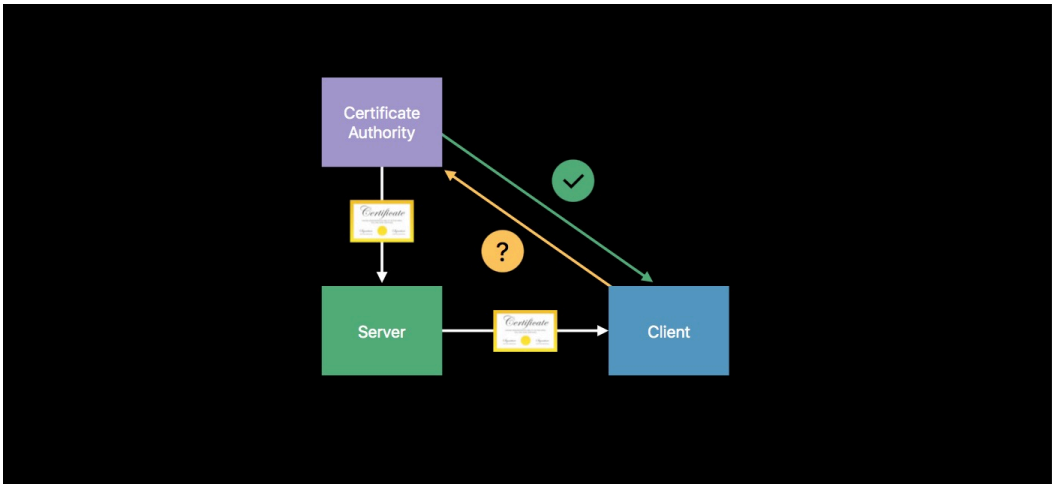
吊销

当私钥泄漏或者或者其他一些情况出现时，我们可能需要吊销证书。客户端在收到证书的时候，需要确认证书的吊销情况。

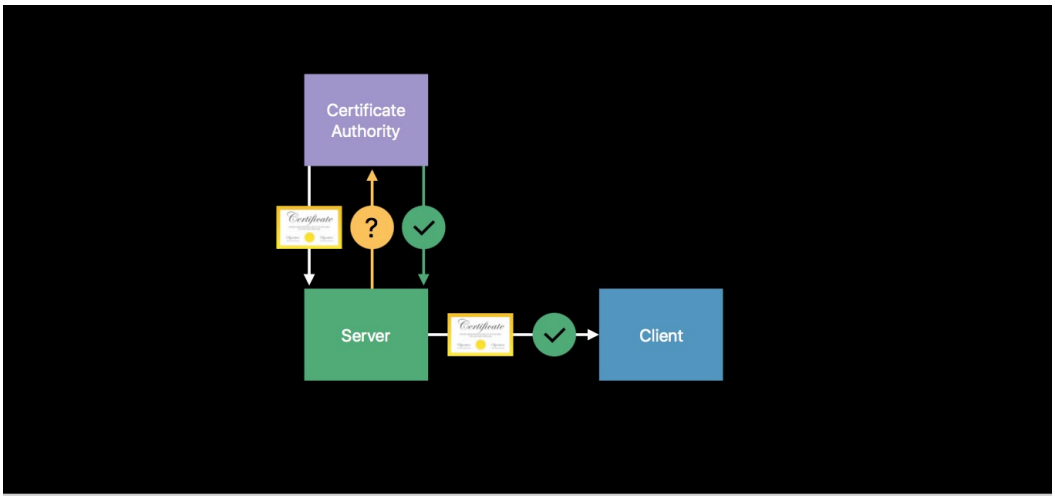
Apple平台默认是不进行吊销检查的。目前存在的吊销机制有以下几种：

- 证书吊销列表（Certificate Revocation List，CRL），这是一个包含吊销证书序列号的列表，证书颁发机构维护着这样的列表。它的缺点在于，CRL 会越来越大，实时查询会越来越慢，同时需要不断请求 CA 的CRL，具有一定的延迟性。
- 在线证书状态协议（Online Certificate Status Protocol，OCSP），OCSP 是这样工作的。首先，服务端会从 CA 申请一个证书，服务端会把证书发送给客户端作为验证，客户端为了验证服务端的身份会向 CA 发送一个请求，来获取该证书的吊销信息，CA 会返回该证书的状态给客户端。客户端根据返回结果来判断服务端的证书是否正确。可以看出来，OCSP 有明显的缺陷，客户端需要向 CA 发送额外的网络请求，这导致了它存在一些性能问题。并且 OCSP 是明文传输，

会存在安全隐患。基于这两个原因，Apple默认不支持OCSP验证。如果开发者想启用 OCSP 查询，需要在 App 中集成其他API。

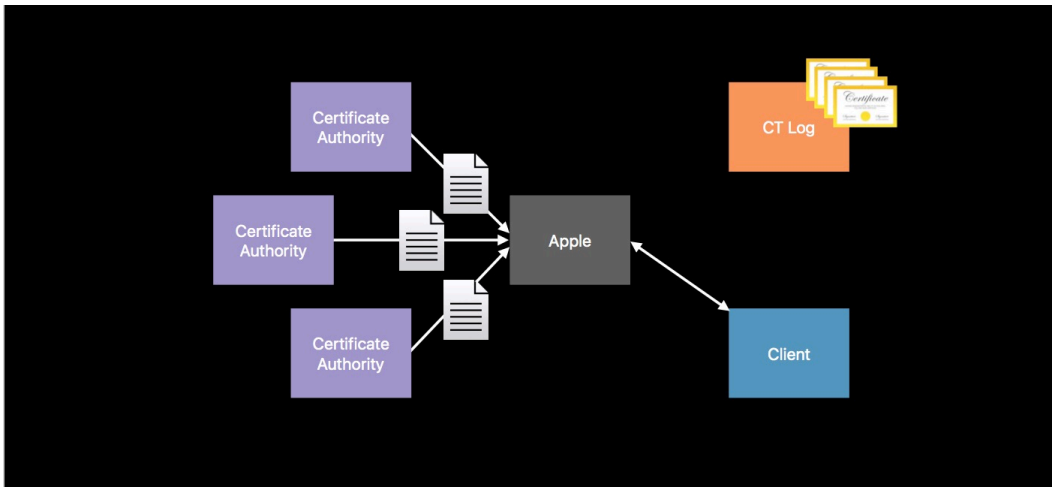


- OCSP Stapling，它弥补了 OCSP 的缺陷，服务端在从 CA 请求证书的时候，也会从 CA 请求到证书的吊销信息，然后将吊销信息和证书一起发送给客户端，客户端可以同时对证书和吊销信息进行校验。



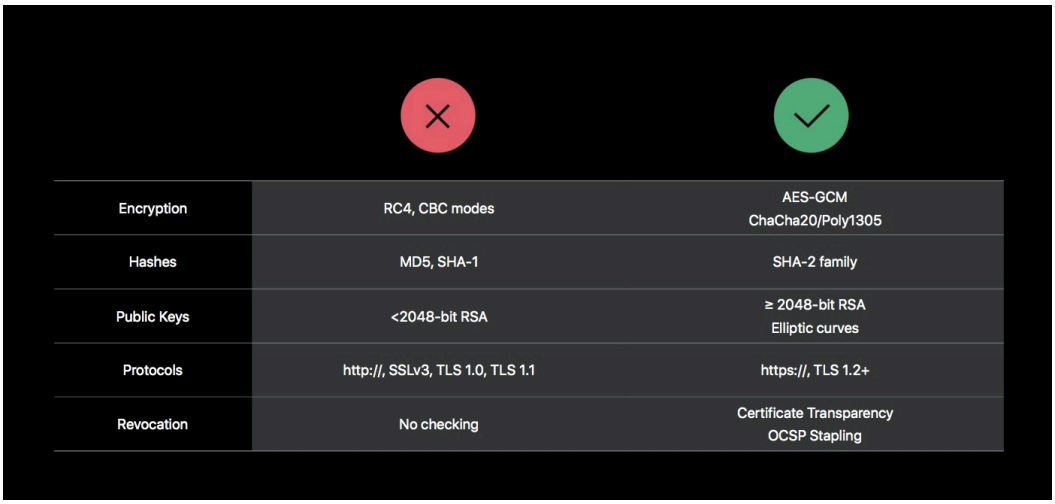
尽管 Apple 鼓励开发者在服务端采用 OCSP Stapling 这种方式，但是普及程度远远不够。Apple 同时宣布加强在所有平台的证书吊销验证。Apple 会从 Certificate Transparency log 中获取信赖的证书列表，开发者和CA都可以向 Certificate Transparency log 提交证书，提交的证书会受到监控和审计。然后 Apple 会从证书颁发机构查询证书的吊销状态，把所有吊销证书的信息捆绑到一起，每隔一段时间自动下发给客户端设备，这样客户端就可以周期性的验证证书的吊销状态了。

在进行 TLS 会话时，如果发现证书在吊销列表内，那么客户端则执行一次 OCSP 检查，去校验证书是否真的已经被吊销。如果证书没有在吊销列表中，则不进行 OCSP 检查。这样的话，客户端一般就不需要向 CA 发送额外的网络请求，避免了性能问题。



回顾

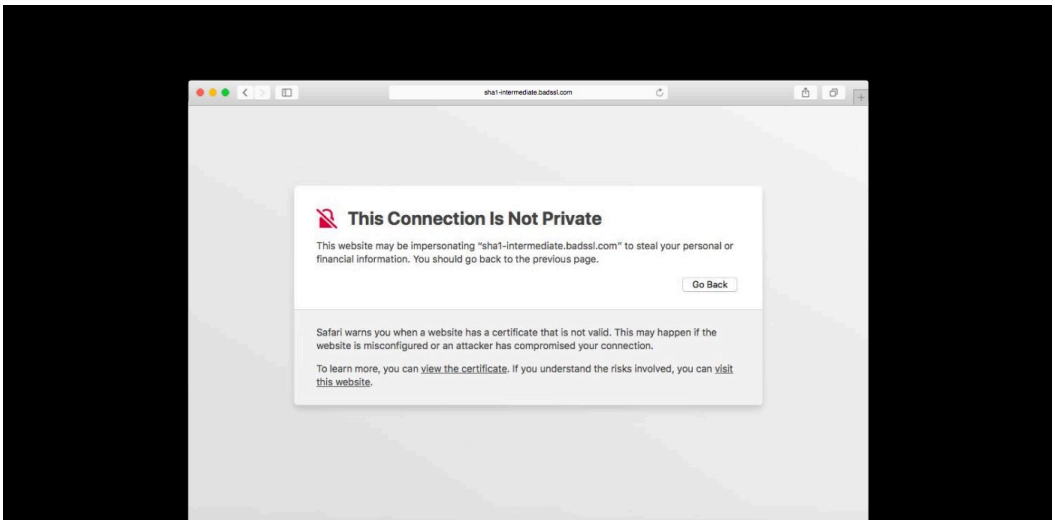
Apple 关于加密、散列函数、网络协议、证书吊销校验和公钥密码方面的修改如图所示：



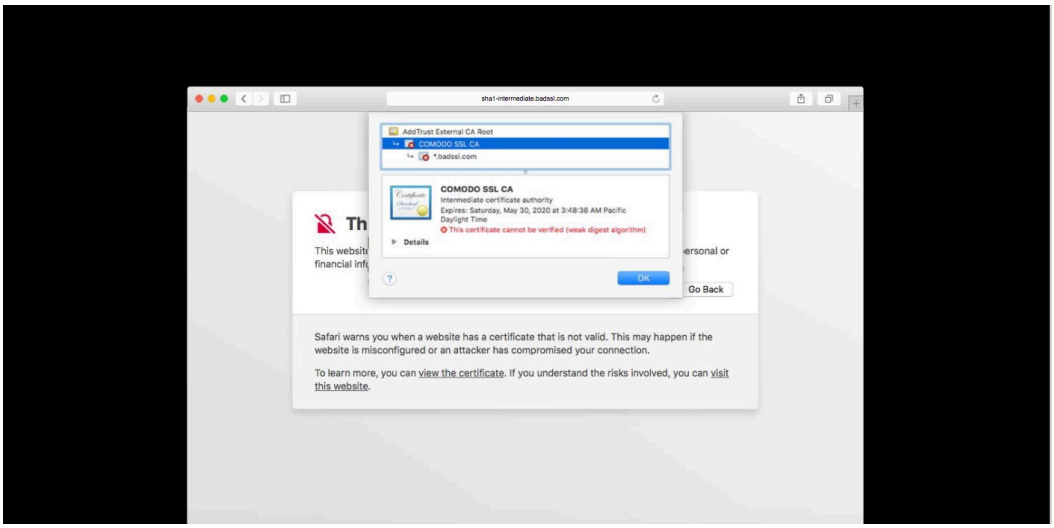
Encryption	RC4, CBC modes	AES-GCM ChaCha20/Poly1305
Hashes	MD5, SHA-1	SHA-2 family
Public Keys	<2048-bit RSA	≥ 2048-bit RSA Elliptic curves
Protocols	http://, SSLv3, TLS 1.0, TLS 1.1	https://, TLS 1.2+
Revocation	No checking	Certificate Transparency OCSP Stapling

新的证书报错界面

在 Safari 中重新设计了新的证书报错界面，如果你用了不符合要求的证书，那么将会看到如下界面。



可以通过证书查看器，进一步查看证书错误的详细情况。



ATS 与 TLS 1.3

ATS 在 iOS 9 的时候推出，是为了保证开发者使用加密的网络来传输数据。但是 Apple 发现全部转为 ATS 的过渡期要比预期长，所以 Apple 扩大了对 ATS 的豁免的支持。

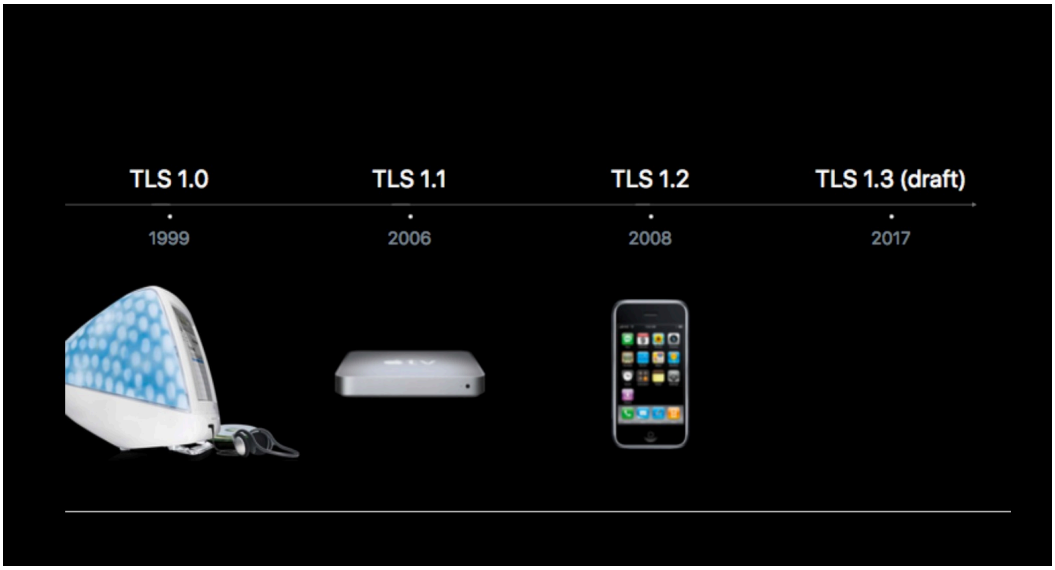
去年 Apple 发现目前的 ATS 豁免并不能满足所有开发者的需求，所以就开放了对AVFoundation、WebView 和 Webkit 的单独豁免支持。豁免可以限制在一个单一域名内和整个 App 内，同时也支持对本地网络（原始 IP 地址和不合格的主机名）的豁免。

虽然目前扩大了对豁免的支持，但是 Apple 依然相信使用 ATS 才是正确的选择，依然会积极推荐 ATS 的使用。

TLS 发展历程

TLS 的前身其实就是 SSL（Secure Sockets Layer），Netscape 是最早发布 SSL 的公司，后续由于互联网标准化组织的接手，将 SSL 标准化，并在 1999 年将其称为 TLS（Transport Layer Security）。后续又在 2006 年 4 月对 TLS 1.0 进行了更新，这个版本与 1.0 差异不大。2008 年发布了 TLS 1.2，也就是目前 Apple 推荐在 HTTPS 网络上使用的协议。

TLS 1.3 是正在制定的 TLS 新标准，目前还是草案版本。



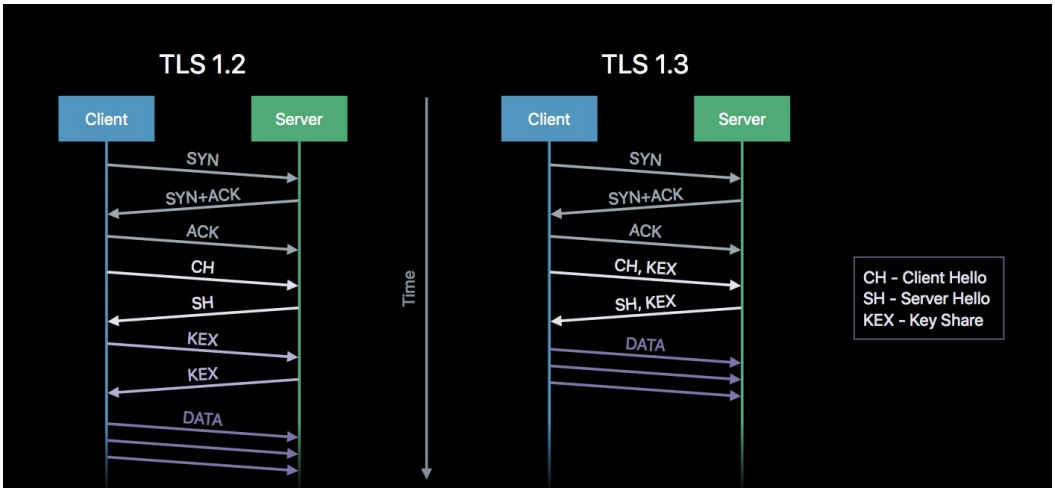
TLS 1.3

在 WWDC 上 Apple 阐述了 TLS 1.3 相比 TLS 1.2 的一些改变：

- 最佳实践

- 取消对老旧加密算法支持，比如 RC4、SHA-1、CBC 和 RSA 加密算法。
- 更简单的配置、更容易测试
- 在 TLS 握手和会话恢复方面性能提升

关于第四点应该是我们最关心的优化点，TLS 握手从原来的四次握手变成了两次握手，也就是说减少了一个 RTT，TLS 1.3 的密钥交换和加密算法的协商都放在了一块。由于这套更高效的握手方法，Apple 宣布大概可以减少三分之一的握手延迟。



虽然正式的版本还没发布，但是现在你可以对 TSL 1.3 beta 版本进行测试了，你可以在 https://developer.apple.com/go/?id=tls13-mobile-profile 下载安装一个配置文件在手机上，同时手机系统升级到 iOS 11 就可以体验最新的 TLS 协议了。同时，在 macOS High Sierra 中，可以通过以下终端命令启用 TLS 1.3。

```
defaults write /Library/Preferences/com.apple.networkd
tcp_connect_enable_tls13 1
```

参考

- 视频地址：Your Apps and Evolving Network Security Standards
- PPT 地址：Presentation Slides (PDF)

- 视频地址：Your Apps and Evolving Network Security Standards
- PPT 地址：Presentation Slides (PDF)

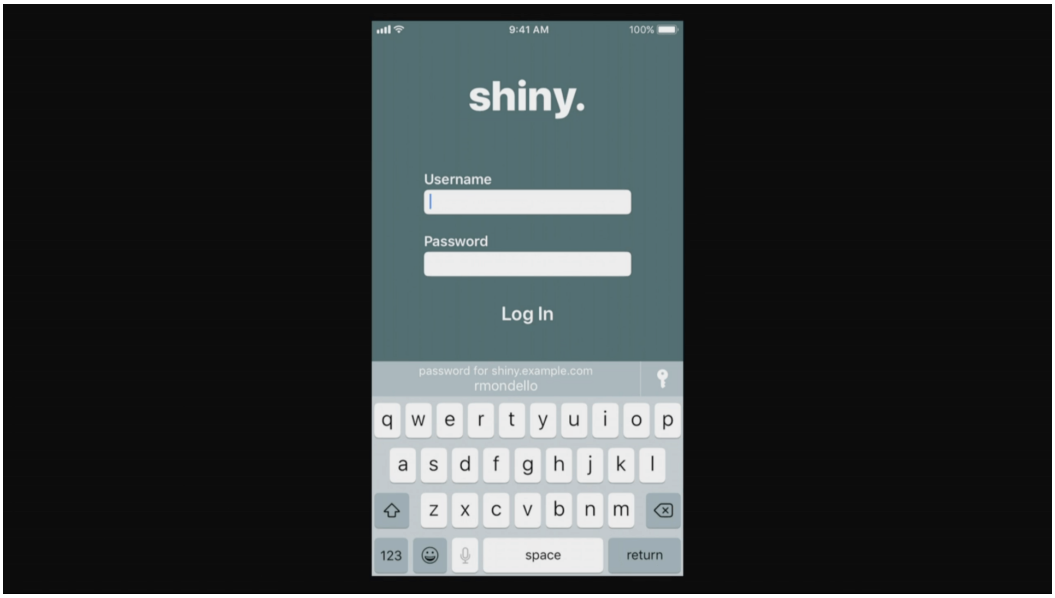
iOS 11 里 App 终于可以密码自动填充了

密码自动填充

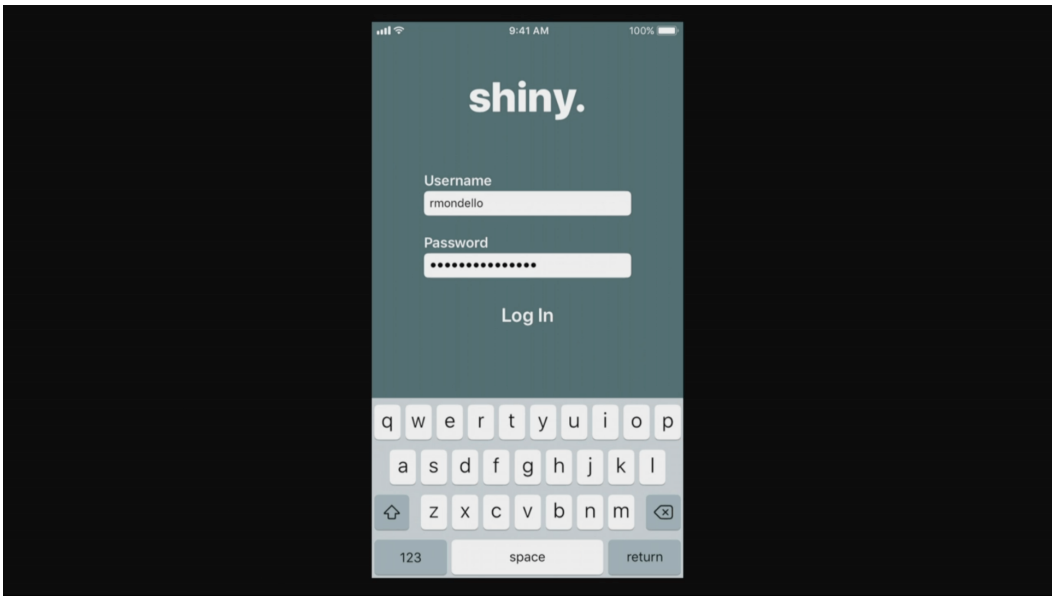
iOS 11 新增了密码自动填充的 API，可以让用户在 App 里使用保存在 Safari 以及 keychain 里的密码。

iOS 以及 macOS 上的 Safari 都内建了密码管理器，每次我们在网页里登录时，Safari 都会询问我们是否要保存密码，然后当我们下一次登录同一个网站的时候，Safari 就会为我们自动填充之前保存的密码。并且这些密码会通过 iCloud keychain 同步到用户所有设备上。

iOS 11 更进了一步，让这些已经保存在 keychain 的密码显示在 App 的虚拟键盘上，接下来让我们来看个 demo：



这里展示的是一个很常见的登录页面，当我们点击输入框的时候，键盘上的 QuickType bar 就会显示出我们之前保存过密码。



点击一下之后，就会把账户和密码自动填充上去。

让 QuickType bar 正确地出现

让 QuickType bar 出现有这么几个条件：

- 密码已经保存在 keychain 里
- 使用原生的 `UITextField` 和 `UITextView`，或者是遵循 `<UITextInput>` 的控件
- iOS 会智能判断哪一个控件要展示 QuickType bar
- 也可以由开发者手动控制让 QuickType bar 出现

我们可以通过修改 `textContentType` 来实现 QuickType bar 的呈现，这是 iOS 10 新引入的一个属性，属于 `UITextInputTraits` 协议的一部分，`UITextField` 和 `UITextView` 都遵守这个协议，通过修改这个属性我们可以让 iOS 提前知道用户会输入什么类型的内容，让**文本补齐**和**文本纠正**更具有针对性。

```
// UITextInputTraits Protocol
// 让 iOS 提前知道用户会输入什么类型的内容
optional var textContentType: UITextContentType! { get set }
```

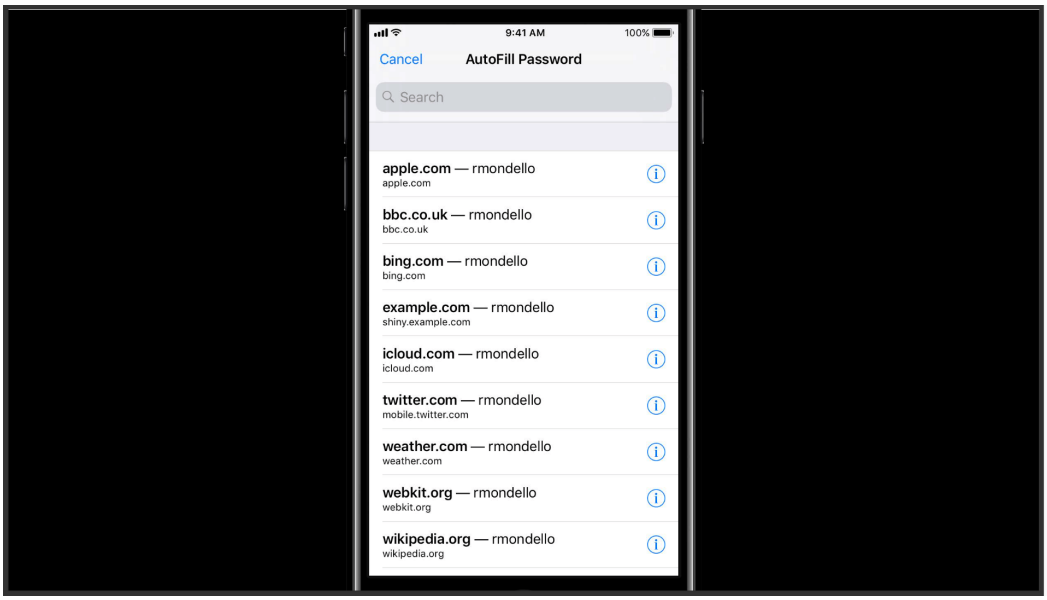
`UITextContentType` 里有很多预设值可以满足我们的需求，例如昵称，地点，组织名，街道名字等等。在 iOS 11 里又引入了两个新的预设值。

```
struct UITextContentType {
    ...
    static let username: UITextContentType
    static let password: UITextContentType
}
```

适配密码自动填充功能很简单，只要把 `UITextField` 或者 `UITextView` 的 textContentType 为 `.username`，`.emailAddress` 或者 `.password` 就可以了。除了通过代码修改，还可以通过 Interface Builder 进行设置。

如果 `UITextField` 的 `textContentType` 设置为 `.password`，即使没有把 `isSecureTextEntry` 设置 `true`，也会显示 QuickType bar。

在我们设置好 `textContentType` 之后，QuickType bar 确实显示出来了，但还没有显示相应的账户密码出来让用户选择，等一下我们再介绍如何显示登录信息出来，但看到这个标志，就意味着 iOS 已经准备好自动填充功能了。点击右边的钥匙 icon，就会弹出 Touch ID 验证。



验证完成后就会显示之前保存的密码，点选了其中一个账户密码之后就会自动填充上去了。

保证把登录信息正确的输入

适配须知

- **iOS 会自动填充 username 控件，以及 password 控件。**
- **即使你的 App 阻止 firstResponder 的修改，但打开密码列表的操作的优先级还是更高。**
- **Touch ID 会让 App 进入 inactive 的状态。**所以不要在 App 进入后台的时候修改登录界面的 UI。

最佳实践

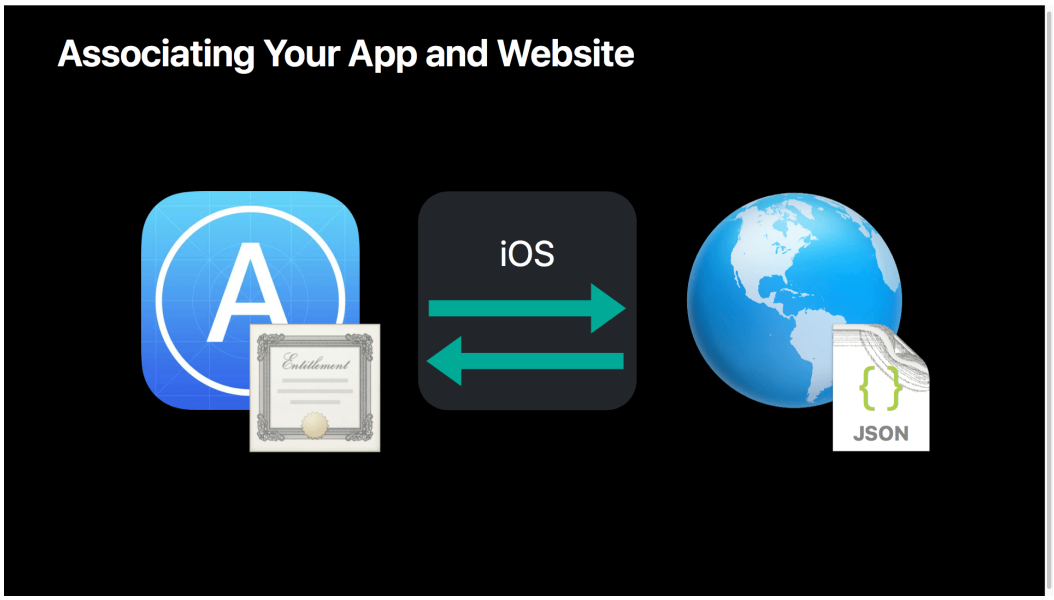
- **不要在 App 进入后台的时候关闭掉登录界面。**
- **通过 textField 的 `text` 属性读取信息。**
- **使用 "didChange" 的代理方法或者通知去验证信息。**可以在这些通知发出，或者是代理方法被调用时去读取登录信息。

```
UITextFieldTextDidChange: Notification.Name
UITextViewTextDidChange: Notification.Name
protocol UITextViewDelegate { optional public func textViewDidChange(...) }
protocol UITextInputDelegate { public func textDidChange(...) }
```

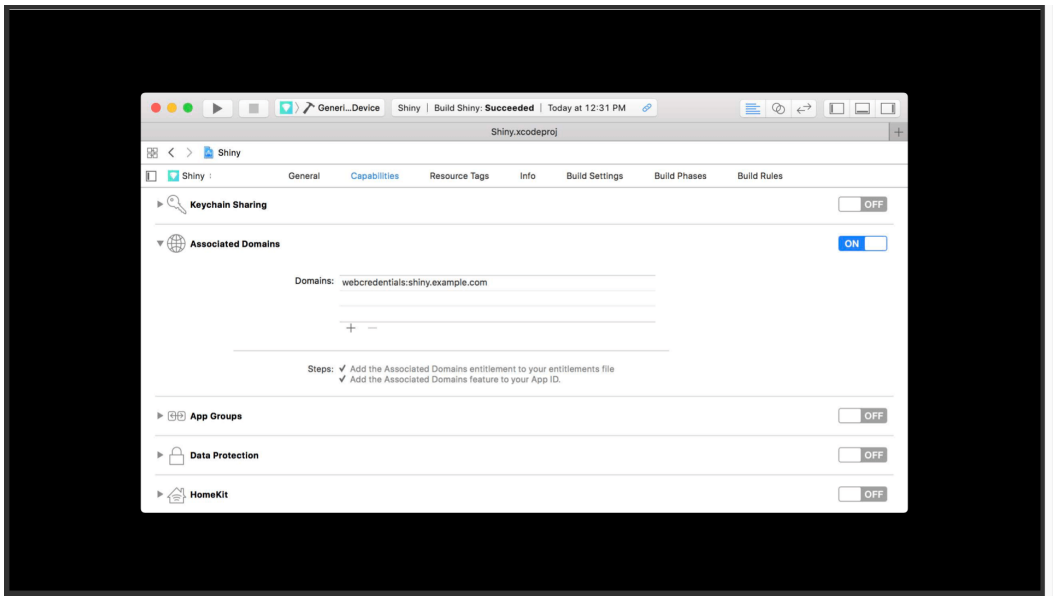
让 QuickType bar 展示正确的登录信息

刚刚我们展示了如何适配自动填充功能，但只有一个🔑出现，并没有把相应的登录信息显示出来。如果你已经适配了 Universal Link，那网站跟 App 就已经关联起来了，iOS 就会把正确的登录信息显示在 QuickType bar 上。

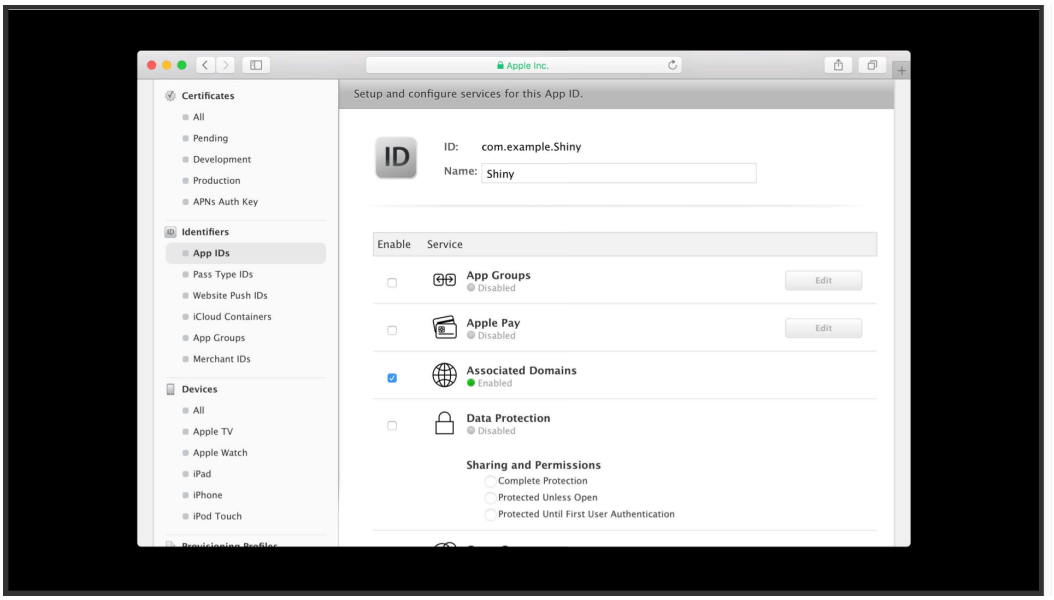
但如果你还没有适配 Universal Link，没关系，接下来我们来介绍另外一种方式。



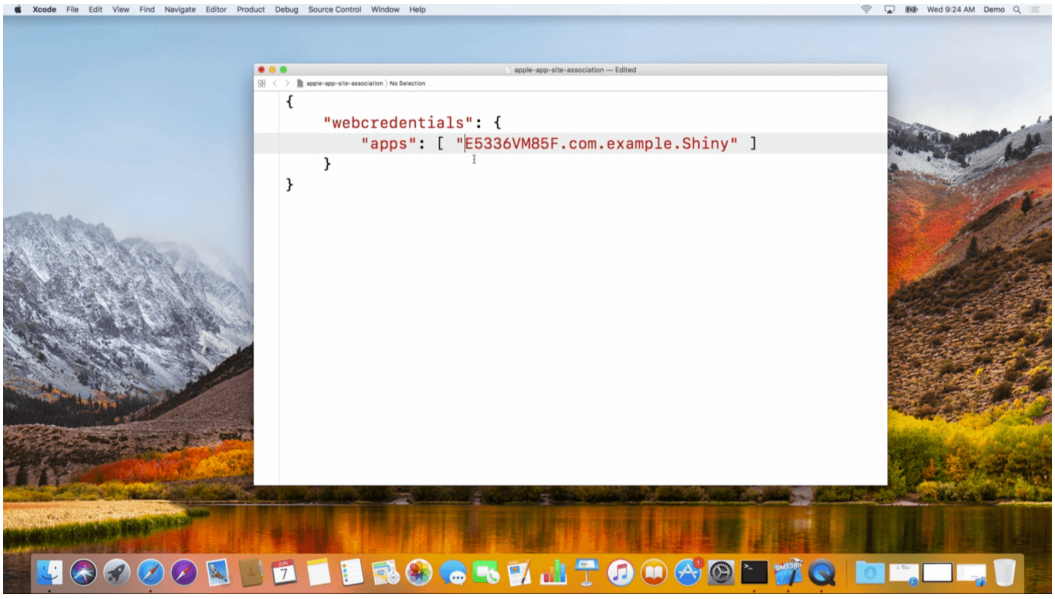
左边代表的是你的 App，右边是你的服务器，App 内部声明与之相关联的网站，iOS 就会向服务器发起一个请求(`/.well-known/apple-app-site-association`)，如果服务器返回了正确格式的 JSON 文件，里面会声明一系列相关联的 App，两者相对应的话，那系统就会将 App 和网站关联起来。



App 需要做的很简单，在项目设置里，把 Associated Domains 的功能打开，然后添加一个 domain，内容填 `webcredentials:` 加上你的服务器域名（在这里是`webcredentials:shiny.example.com`）。

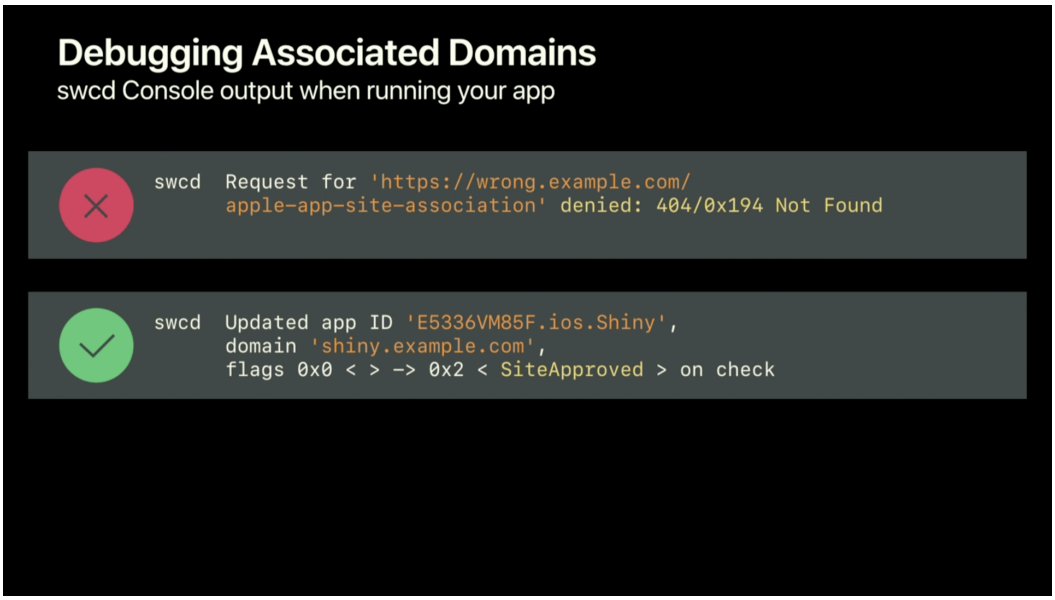


如果之前没有设置过 Associated Domains，那记得去苹果的开发者网站里，把 App 的 Associated Domains 功能给打开。



接下来我们还需要让服务端认可这个关联操作，建立一个名为 `apple-app-site-association` 的 JSON 文件，格式如上图，`apps` 字段里填上网站相关联的 App 的数组，每个 App 都填上团队 ID 加上 bundle ID 就可以了（基本上跟 Universal Link 一样）。最后把这个 JSON 文件放在服务器根目录下的 `.well-known` 文件夹里就可以了。

测试的时候记得只有在真机上才能正确地测试，完成了上面的设置之后，我们的 App 就可以呈现出正确的登录信息了。



测试的时候，我们可以连接上手机，打开 Console，通过 filter 找到 swcd 命令，每次编译运行的时候它都会告诉我们有没有与服务器成功地建立起链接。

第三方登录服务

前面我们介绍了如何关联自己的服务器和域名，但还有一种情况，登录第三方服务，例如微博，Twitter 这些社交账户，这里我们可以使用 `SafariViewController` 去完成登录，一方面是因为 Safari 本身就有带有登录信息填充功能，另一方面是 Safari 对于 App 来说是一个黑盒，可以让用户信息得到很好的保护。

现在 `SafariViewController` 添加了很多自定义项，可以让它看起来与 App 更加一致。在这里左上角的按钮被改为了 Cancel，原有的 Done 按钮在这个情境下会显得很奇怪，整体 UI 也调整成了蓝底白字。

Safari 更多的自定义项，请查看 Session 225 - What's New in Safari View Controller。

关于使用 SaraiViewController 完成 OAuth 的流程，请查看 Session 504 - Introducing Safari View Controller。

总结

- **登录流程阻碍过多是让用户放弃 App 的一个主要因素。**通过登录信息自动填充可以让登录步骤减少，借此优化用户体验。
- **系统会智能地判断哪些地方需要密码自动填充。**但我们也需要给系统足够的信息去进行判断。
- **如何保证密码自动填充正常工作：**
  - 使用 `UITextContentType.username` 和 `UITextContentType.password`
  - 正确设置 Associated Domains 服务的 webcrendtials 项

iOS 11 定位技术中的一些新特性

定位技术在 App 开发过程中占据非常重要的位置，尤其是在一些户外型 App，如：Keep、共享单车等。正是因为定位技术，他们才能给用户最佳体验。在 iOS 11 中，定位技术又做出了一些改进，特别是在授权访问，并且用户体验也得到了较大的提升。本文末尾也会简单提及 iOS 11 中如何针对这块进行适配。

概览

- API 改进
- 授权与使用指示
- 最佳实践

API 改进

CLGeocoder API

在 iOS 11 中，CLGeocoder 开始支持 Contacts 框架了。

首先，CLGeocoder 中此方法已过期：

```
- (void)geocodeAddressDictionary:(NSDictionary *)addressDictionary
completionHandler:(CLGeocodeCompletionHandler)completionHandler
```

现在 CLGeocoder 中新增了几个 API：

- 其中新增的下面这个方法来代替了上述方法：

  ```
   - (void)geocodePostalAddress:(CNPostalAddress *)postalAddress
  completionHandler:(CLGeocodeCompletionHandler)completionHandler;
  ```

- 在 iOS 11 进行地理编码时，现在支持使用区域设置了，API 如下：

  ```
  - (void)reverseGeocodeLocation:(CLLocation *)location preferredLocale:
  (nullable NSLocale *)locale completionHandler:
  (CLGeocodeCompletionHandler)completionHandler;

  - (void)geocodeAddressString:(NSString *)addressString inRegion:(nullable
  CLRegion *)region preferredLocale:(nullable NSLocale *)locale
  completionHandler:(CLGeocodeCompletionHandler)completionHandler;

  - (void)geocodePostalAddress:(CNPostalAddress *)postalAddress
  preferredLocale:(nullable NSLocale *)locale completionHandler:
  (CLGeocodeCompletionHandler)completionHandler;
  ```

我们可通过 **geocodePostalAddress** 相关 API 进行地理编码，可以把邮政地址（`CNPostalAddress`）转成地标(`CLPlacemark`)，并且在 `CLPlacemark` 中还可以拿到 **postalAddress** (`CNPostalAddress`)邮政地址，需要注意的是 **addressDictionary** 方法已经过期。（Some Exciting Changed for you）.

```
@property (nonatomic, readonly, copy, nullable) NSDictionary
*addressDictionary;
```

举个例子简单展示如何通过地理编码把邮政地址转成地标：

```
CNMutablePostalAddress *postalAddress = [[CNMutablePostalAddress alloc]
init];
postalAddress.street = @"Keyan Road No.9 Nanshan";
postalAddress.city = @"Shenzhen";
postalAddress.state = @"Guangdong";
postalAddress.postalCode = @"518000";
postalAddress.country = @"China";
postalAddress.ISOCountryCode = @"CN";

if (@available(iOS 11.0, *)) {
   [self.geocoder geocodePostalAddress:postalAddress
completionHandler:^(NSArray<CLPlacemark *> * _Nullable placemarks, NSError *
_Nullable error) {
       if (!error) {
           NSLog(@"**********geocodePostalAddress************");
           CLPlacemark *placemark = placemarks.lastObject;
           NSLog(@"%@", placemark.name);
           NSLog(@"**********geocodePostalAddress************");
       }
   }];
} else {
   // Fallback on earlier versions
}

// print
**********geocodePostalAddress************
No. Nine Keyan Road
**********geocodePostalAddress************
```

我们再针对区域设置举个例子，注意，用户的手机语言主体是英语，但我们在针对邮政地址进行地理编码时任然可以通过设置 `preferredLocale` 参数来得到我们想要的不同语言的地标信息。

```
NSLocale *locale = [[NSLocale alloc] initWithLocaleIdentifier:@"zh_Hans_CN"];
[self.geocoder geocodePostalAddress:postalAddress preferredLocale:locale
completionHandler:^(NSArray<CLPlacemark *> * _Nullable placemarks, NSError *
_Nullable error) {
  if (!error) {
      NSLog(@"**********geocodePostalAddress-preferredLocale************");
      CLPlacemark *placemark = placemarks.lastObject;
      NSLog(@"%@--%@", placemark.name, placemark.postalAddress);
      NSLog(@"**********geocodePostalAddress-preferredLocale************");
  }
}];

// print
**********geocodePostalAddress-preferredLocale************
科研路9号
**********geocodePostalAddress-preferredLocale************
```

通过打印信息可看出，设置了 preferredLocale 后，得到的 placemark 是中文信息。

指向与移动方向（Heading & Course）

Core Location 支持两种方式来获取方向相关的信息：**Heading** & **Course**。

用过地图的导航同学都知道，地图上有一个带方向的定位小蓝点，并且能够随着手机的转动而转动，这使得用户能够辨别当前的方向，从而提高导航用户体验，这是通过 `CLHeading` 来实现的。但苹果认为，有时候单靠 `Heading` 是不靠谱的，`Heading` 仅代表用户面朝方向，而 `Course` 苹果则解释为用户实际移动的方向。因此 `Heading` 与 `Course` 相结合能够带来更好的用户体验。

- **Heading**（指向）：即设备的指向（设备中的磁力计计算结果）。
- **Course**（移动方向）：即设备的移动方向（通过 `GPS` 计算），不考虑设备的指向。

至于使用 `Heading` 还是 `Course` 最终取决于 App 的使用，有时候我们可能是其中一种或者结合两种方式来使用。比如，我们在步行的时候，使用 `Heading` 可能更加合适；而在驾驶过程中，则使用 `Course` 更加合适，因为它能代表汽车的行驶方向。

在 iOS 11 中，**Heading** & **Course** 得到了增强，`Core Location` 会自动探测当前情况，使得 `Heading` 和 `Course` 达成一定的'共识'，并且 `Course` 提供的信息能使 `Heading` 的信息更加精准。两者的特点如下：

- 更精准的设备指向估算
- 全自动化

在 iOS 11 中，也为 CMDeviceMotion 新增的一个属性 heading，用于获取 `CLHeading` 原始值。

```
/*
 *  heading
 *
 *  Discussion:
 *    Returns heading angle in the range [0,360) degrees with respect to the
CMAttitude reference frame. A negative value is returned
 *    for CMAttitudeReferenceFrameXArbitraryZVertical and
CMAttitudeReferenceFrameXArbitraryCorrectedZVertical.
 *
 */
@property(readonly, nonatomic) double heading NS_AVAILABLE(NA, 11_0);
```

授权（Authorization）

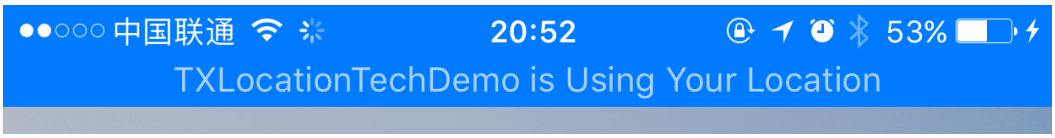
此模块是今年最大的一个更新点。

授权模式（Authorization Modes）

`Core Location` 支持两种模式：使用应用时能访问用户位置 & 未使用应用时能访问用户位置。（`When In Use` & `Always`）

回顾（Review）

- `Always`（**始终**）授权模式能使你随时随地访问用户位置，除此外还能使用后台监控相关 APIs。如果 App 在前后台都需要访问用户位置即使 App 没有运行，那么应采用此种模式。
- `When In Use`（**使用期间**）授权模式只能在用户使用该应用时才能访问其位置信息，但也有特殊情况，比如：你使用了后台持续定位相关 API (**Continuous Background Location**)，在手机屏幕上方会出现蓝条视图，这就是提醒用户，该 App 正在运行并且在访问位置服务，那么此时，我们可将此种情况纳入 “用户正在使用该应用” 这种情况。



关于更多 `Always` 与 `When In Use` 两种模式的一些访问权限区别，如下：

更详细的如下：（截自官方文档）

Table 1-1 APIs and technologies for which an iOS app can specify location updates

API	Requires location updates in background mode	Can use location updates in background mode
`startUpdatingLocation`	Yes	Yes
`CLVisit`	No	Yes
`CLRegion`	No	Yes
`startMonitoringSignificantLocationChanges`	No	Yes
`CLBeaconRegion`	No	Yes, on a case-by-case basis

Table 1 Authorizations supported by different location services

Service	In-Use	Always
Standard location service	Yes	Yes
Significant-change location service	No	Yes
Visits service	No	Yes
Region monitoring	No	Yes
iBeacon ranging	Yes	Yes
Heading service	Yes	Yes
Geocoding services	Yes	Yes

> With **in-use** authorization, your app can use most location services but **cannot use services that automatically relaunch the app**, such as region monitoring or the significant location change service. Your app must always start location services in the foreground. If you enable the background location capability for your app, any location services that you start in the foreground will continue to run while your app is in the background.

`When In Use` 授权模式，即应用程序可以使用大部分的定位服务，但是无法使用那些可以自动从后台唤醒应用程序的服务，如：区域监听、有效位置发生改变的服务。因此当处于该授权模式时，应用程序必须始终在前台启动位置服务。如果应用程序启用了后台定位的能力，那么你在前台启动的所有定位服务将在应用程序处于后台时继续保持运行。

> With **always** authorization, your app can use any location service and can start those services from either the foreground or the background. If a location-related event occurs when your app is not running, **the system is allowed to launch your app so that it can process the event**.

`Always` 授权模式，即应用程序可以使用任何定位服务，并且在前后台都可以启用在**使用期间**授权模式下不能够使用的那些服务，如：区域监听、有效位置发生改变的服务。如果在应用程序没有运行的情况下发生了一个定位相关事件，那么系统会自动唤醒你的应用程序以便处理该事件。

至此，相信大部分同学已经懂了 `Always` 与 `When In Use` 两种模式区别。

为了让用户能更好控制 App 的访问权限，又能让开发者给予用户更好的使用体验，在 iOS 11 中，苹果引进了一种新的授权模式，即 `Always` 与 `When In Use` 两种模式的一个平衡点，后文将会介绍。

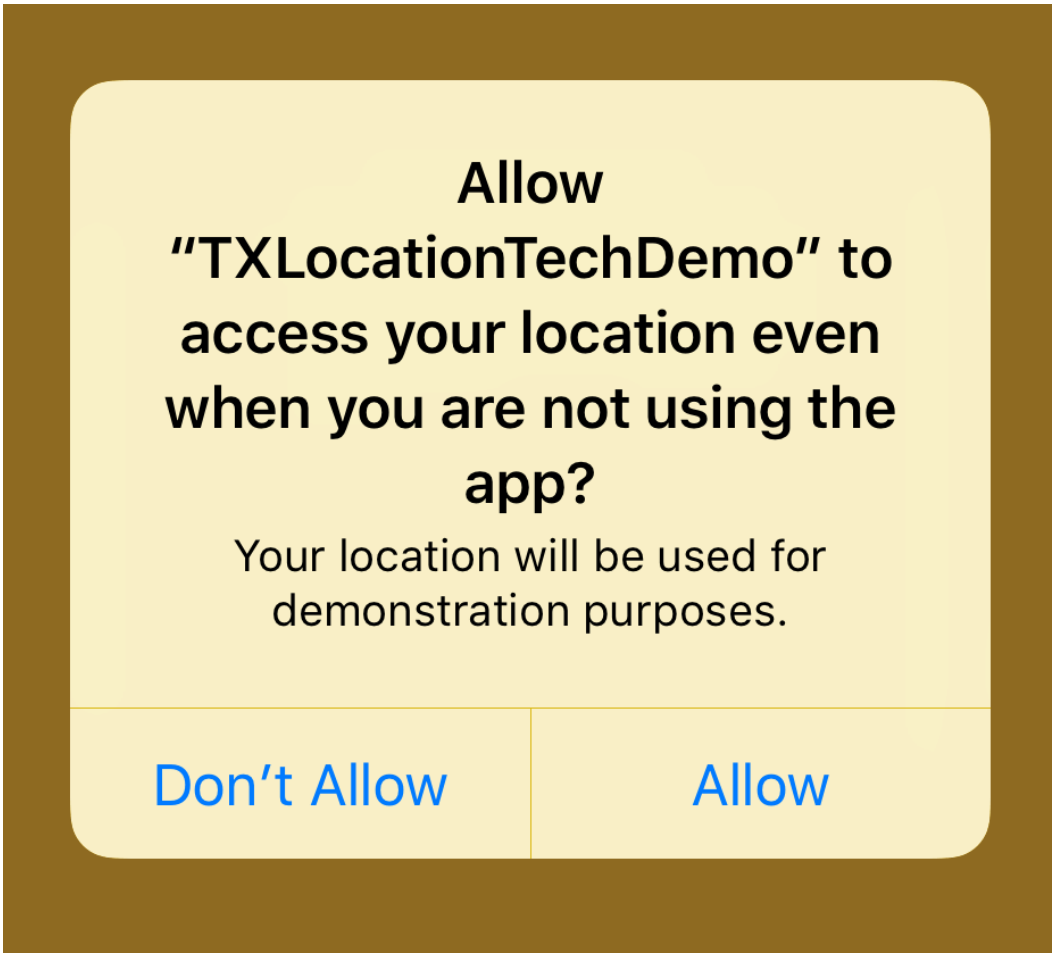
使用 `Always` 授权的动机（Motivations for Requiring Always）

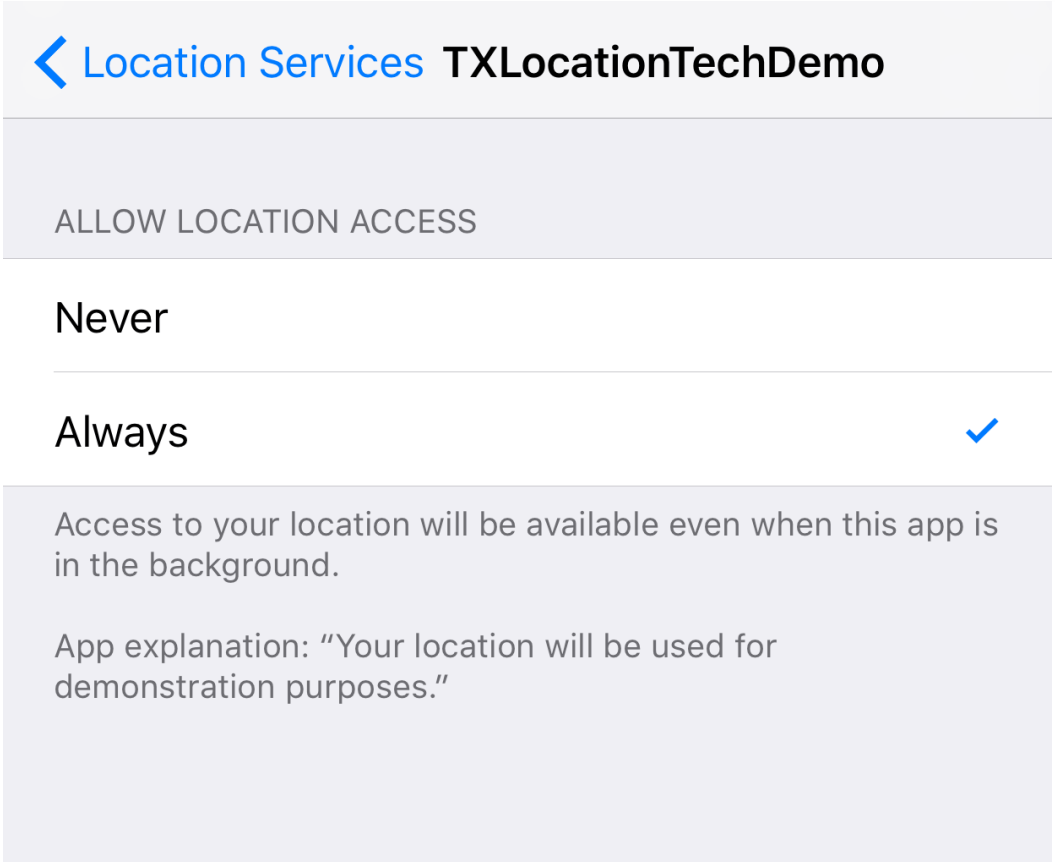
- 困惑（Developer confusion）

用很多开发者喜欢向用户请求 `Always` 授权模式，但是又从未使用任何后台监控相关 APIs，如：**startMonitoringSignificantLocationChanges**等，开发者可能对 `Always` 与 `When In Use` 两种授权模式感到困惑，因此选择了 `Always` 授权模式，而非 `When In Use`。

- 追求神奇的体验（pursuit of magical experiences）

有一些基于地理定位的 App 比较极端，只提供请求 `Always` 授权模式，要么 Always 要么 Never，就像这样：





显然这种用户体验非常糟糕，用户有时候并不希望 App 一直能访问他的位置信息。

iOS 11 中 `Always` 授权（Always Authorization in iOS 11）

- 必须支持 `When In Use` 授权模式（**NSLocationWhenInUseUsageDescription**）

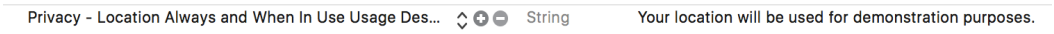
在 iOS 11 中，为了避免开发者只提供请求 `Always` 授权模式这种情况，加入此限制，如果不提供 `When In Use` 授权模式，那么 `Always` 相关授权模式也无法正常使用（**包括现有的 App。因此小伙伴们在 iOS 11 GM版本发布时，赶紧做适配哈！**）。在 info.plist 中进行配置：

经测试，在 iOS 11 中如果不提供该授权模式，将无法访问用户位置信息，在 Xcode 9 beta3 中 Debug 调试窗会出现如下信息：

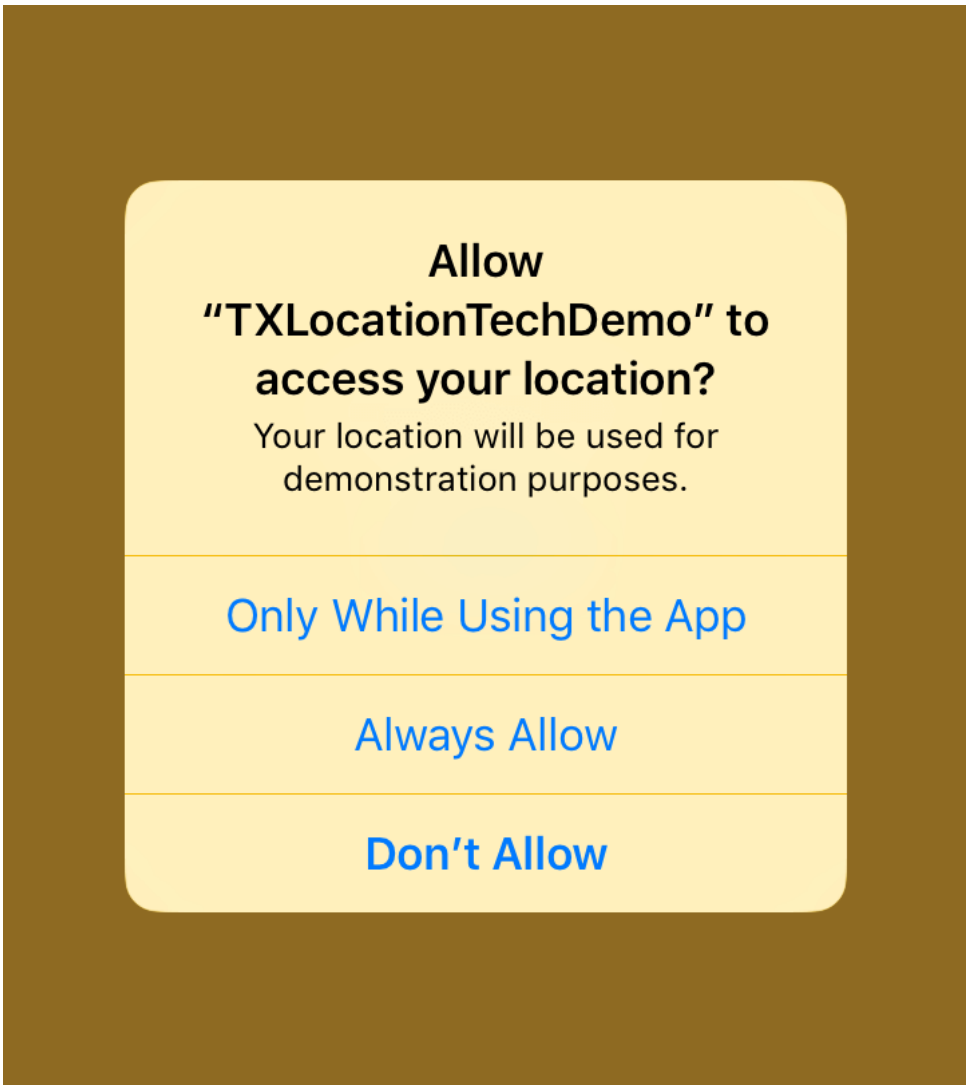
> This app has attempted to access privacy-sensitive data without a usage description. The app's Info.plist must contain an NSLocationWhenInUseUsageDescription key with a string value explaining to the user how the app uses this data

- `Always` 授权模式弹窗包括 `When In Use` 选项（Prompts for Always include WhenInUse options）

iOS 11 为此引入了一个新的 Key：**NSLocationAlwaysAndWhenInUseUsageDescription**。如果要请求 `Always` 授权模式，在 info.plist 中必须配置该 Key：



效果如图所示：



否则将无法向用户请求 `Always` 授权模式，iOS 11 在 **Xcode 9 beta3** 中 Debug 调试窗会出现如下信息：

> This app has attempted to access privacy-sensitive data without a usage description. The app's Info.plist must contain both NSLocationAlwaysAndWhenInUseUsageDescription and NSLocationWhenInUseUsageDescription keys with string values explaining to the user how the app uses this data

注意：在 iOS 11 及后期版本 Core Location 将不再使用 NSLocationAlwaysUsageDescription

支持 `Always` 授权（Supporting Always Authorization）

兼容 iOS 11 以下系统版本

如果要支持老版本，即 iOS 11 以下系统版本，那么建议在 info.plist 中配置所有的 Key（即使 NSLocationAlwaysUsageDescription 在 iOS 11及以上版本不再使用）：

- NSLocationWhenInUseUsageDescription
- NSLocationAlwaysAndWhenInUseUsageDescription
- NSLocationAlwaysUsageDescription

帮助用户选择

在授权请求的描述中，我们应该解释请求该种授权模式的原因以及将会带来那些新的功能特性。

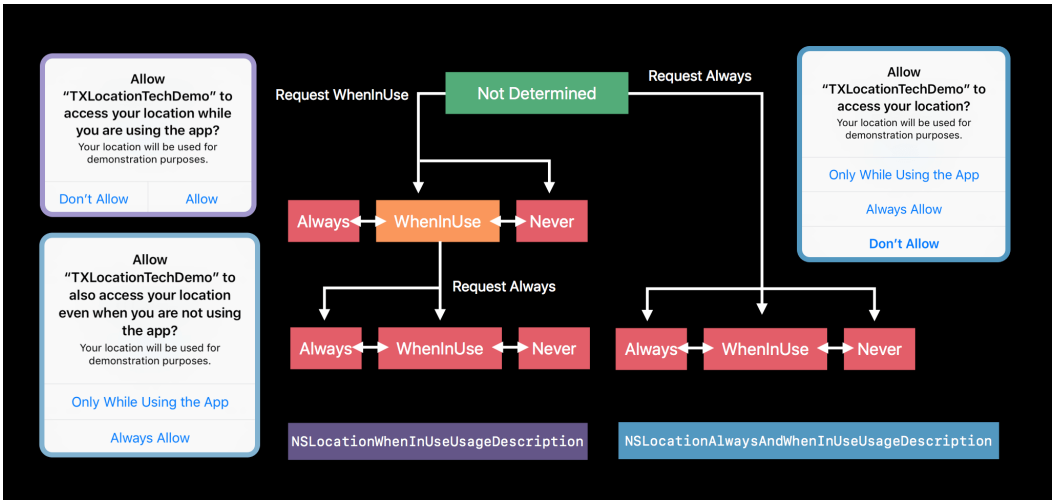
请求授权（Requesting Authorization）

- 对于所有 App 而言，最好是在需要使用用户位置信息时，才发送授权请求。

- 建议只有在真的需要用到 `Always` 授权模式才使用，而非随意选择。
- 如果要请求 `Always` 授权模式，建议先请求 `When In Use` 授权模式，这样会提高用户的授权率。

从 `When In Use` 到 `Always` 授权（From WhenInUse to Always）

直接上图：



一目了然！

使用指示（Location Usage Indication）

iOS 10 中

- 使用 Core Location 中大部分的定位服务时，都会显示一个实心箭头；使用区域监听相关 APIs 时，会显示一个空心箭头
- 如果使用 `When In Use` 授权模式，并且使用了持续定位相关 API (**Continuous Background Location**)，在手机屏幕上方会出现蓝条状态条。而 `Always` 授权模式则不会。

iOS 11 中

- 新的箭头样式显示策略：当通过 Core Location 请求获取位置信息时，状态栏上会显示空心箭头；当接收位置信息时，状态栏会显示实心箭头（For a few seconds）。
- 使用 `Always` 授权模式，并且使用了持续定位相关 API (**Continuous Background Location**)，在手机屏幕上方也会出现蓝条状态条。

总结

本次最大的更新主要是在**授权**，提升了用户体验。另外，需要注意的是，在 **iOS 11 GM 版本**发布时，开发者需要**及时适配**，避免部分功能在 iOS 11 上用户无法正常使用。

实践

此示例程序提供 `Swift` & `Objective-C` 双版本（**FYI**），主要是展示：

- iOS 11 上定位**授权流程**及**适配问题**
- iOS 11 上定位**使用指示**
- CNPostalAddress **地理编码**

GitHub 地址：TXLocationTechDemo

参考

1. https://developer.apple.com/videos/play/wwdc2017/713/
2. https://developer.apple.com/library/content/documentation/UserExperience/Conceptual/LocationAwarenessPG/CH5-SW1
3. https://developer.apple.com/library/content/documentation/UserExperience/Conceptual/LocationAwarenessPG
4. https://developer.apple.com/documentation/corelocation/requesting_permission_to_use_location_services#ove
5. http://nshipster.com/core-location-in-ios-8/

StoreKit 的新改变

概述

本文主要介绍 iOS 10.3 以来，以及 iOS 11 版本中 `StoreKit` 相关的，全文分为三个主要部分，分别介绍：

- 应用内购买(IAP)的实现流程
- 应用内购买(IAP)在 App Store 中推广
- 用户打分、评论、开发者回复等机制的改变

背景

对于在 App Store 中上架的应用来说，通常应用内购买(In-app purchase，简称 IAP) 在总利润中占据了相当大的比例。所以能够使用 IAP ，把内容卖给用户并在此过程中提供良好的用户体验对开发者或者公司而言就显得非常重要。整个 IAP 的过程，在客户端的实现依赖于 StoreKit 这个框架，因此这一节主要讲述的是苹果在 iOS 11 系统中对 StoreKit 框架 作出的改进。

不管读者有没有使用 StoreKit 的经验，在介绍 StoreKit 的新特性之前，我们先来复习一下 IAP 的整个流程。首先，IAP 主要用来销售电子的内容或者服务。至于实物交易，则应该通过 Apple Pay(或者国内的各种支付工具)来完成。

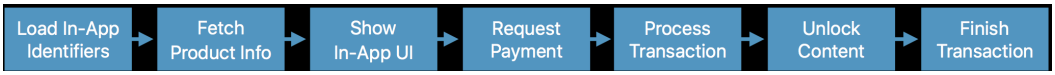
其次，IAP 可以分为四种类型:

1. 可消耗的产品: 比如金币
2. 不可消耗的产品: 比如解锁应用的高级功能
3. 不可自动续期的订阅
4. 可自动续期的订阅

订阅是一种新的 App 购买方式，主要是为了解决开发者缺少持续收入，只能一次性卖软件的问题。如果用户通过订阅的方式购买软件，那么在订阅期结束以后，需要继续订阅才能继续使用 App。因此第三种和第四种 IAP 的区别主要就在于订阅期结束以后，用户是否会自动续订。目前默认的选择是自动续订。

IAP 的流程

这一节 Session 主要讨论的还是前两种 IAP 的方式。如果开发者想要提供 IAP 的功能，他们首先要要在应用内加载 In-app Identifier，下面是完整的流程:



1. 加载 In-app Identifier
2. 客户端从 App Store 中获取本地化的商品信息。
3. 把 IAP 的购买界面展示给用户，用户可以同意购买并点击购买按钮。
4. 用户授权购买，客户端向服务器发送购买请求。
5. 服务器处理购买请求，并把结果返回给 StoreKit。
6. 如果购买请求验证通过，客户端此时解锁内容或者提供金币。
7. 至此，整个交易流程结束。

加载 In-app Identifier

In-app Identifier 为每个可销售的商品提供一个唯一标示。在客户端上，我们可以直接写死代码:

```
let identifiers = ["com.myCompany.myApp.product1",
"com.myCompany.myApp.product2"]
```

也可以从自己的服务器动态的获取:

```
let identifiers = remoteIdentifiers()
```

获取商品信息

加载本地化的商品信息非常简单，只要把 identifier 传递到 `SKProductsRequest` 这个类的初始化方法中即可:

```
let request = SKProductsRequest(productIdentifiers: identifierSet)
request.delegate = self
request.start()
```

设置好代理后就可以发送请求了，如果请求成功，会在 `didReceive` 这个回调函数中收到一个商品数组，这个数组中的每一个元素和请求中的每一个 identifier 对应:

```
func productRequest(_ request: SKProductsRequest, didReceive response:
SKProductsResponse) {
    for product in response.products {
        // 本地化的商品名称和描述信息
        product.localizedTitle
        product.localizedDescription
        // 价格和国家
        product.price
        product.priceLocale
        // 内容的大小和版本
        product.downloadContentLengths
        product.downloadContentVersion
    }
}
```

需要强调的是，开发者不要缓存这些网络请求的结果，而是每次都获取实时的、最新的数据，因为用户可能会切换 App Store 的地区而且货币的汇率可能会实时变化。

展示购买界面

购买页面由各个应用自己负责绘制。但这个页面应该经过精心设计，因为不同的页面美观程度不一样，这会对用户是否购买产生较大的影响。感兴趣的读者可以参考苹果开发者网站的这个链接，里面介绍了一些能够帮助提高销量的技巧。

在展示 UI 时，我们可能涉及到货币单位的处理，比如是用人民币计价还是用美元，我们可以用这样的代码来完成:

```
let formatter = NumberFormatter()
formatter.numberStyle = .currency
formatter.locale = product.priceLocale
let formattedString = formatter.string(from: product.price)
```

需要强调的是，这里我们只要使用从 App Store 拿到的价格和国家就可以了，剩下的工作都应该交给 StoreKit 完成。尤其是这里的国家，如果不设置，默认会使用设备地区。但如果用户选择的地点是美国，而 App Store 则登录了中国商店，这里默认就会显示美元为单位。但显然应该以 App Store 商店所在的国家为准。

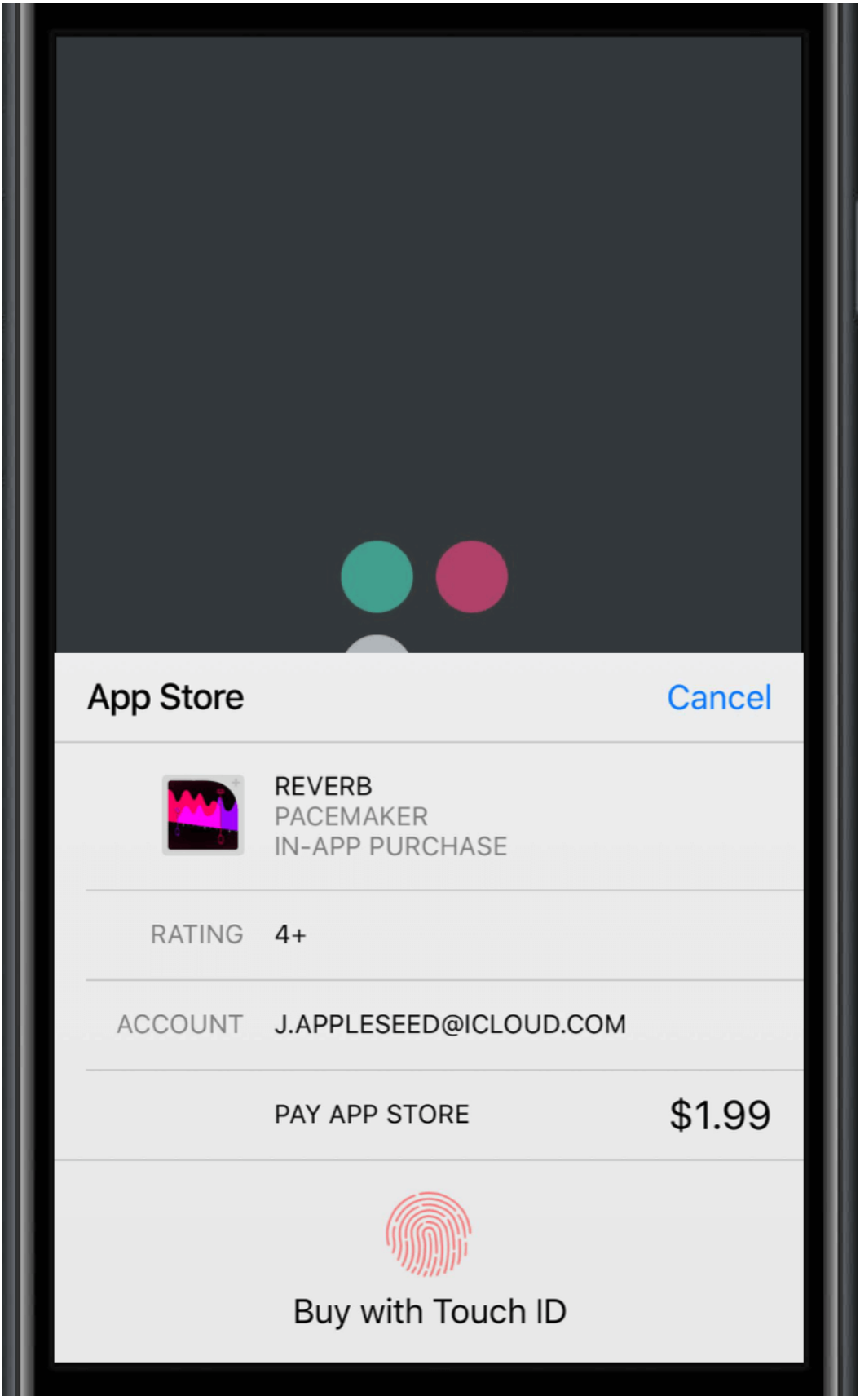
切记，一切交给 StoreKit 处理，开发者只要拿着被格式化过的字符串就可以了，比如汇率计算之类的也都由 StoreKit 处理。

发送购买请求

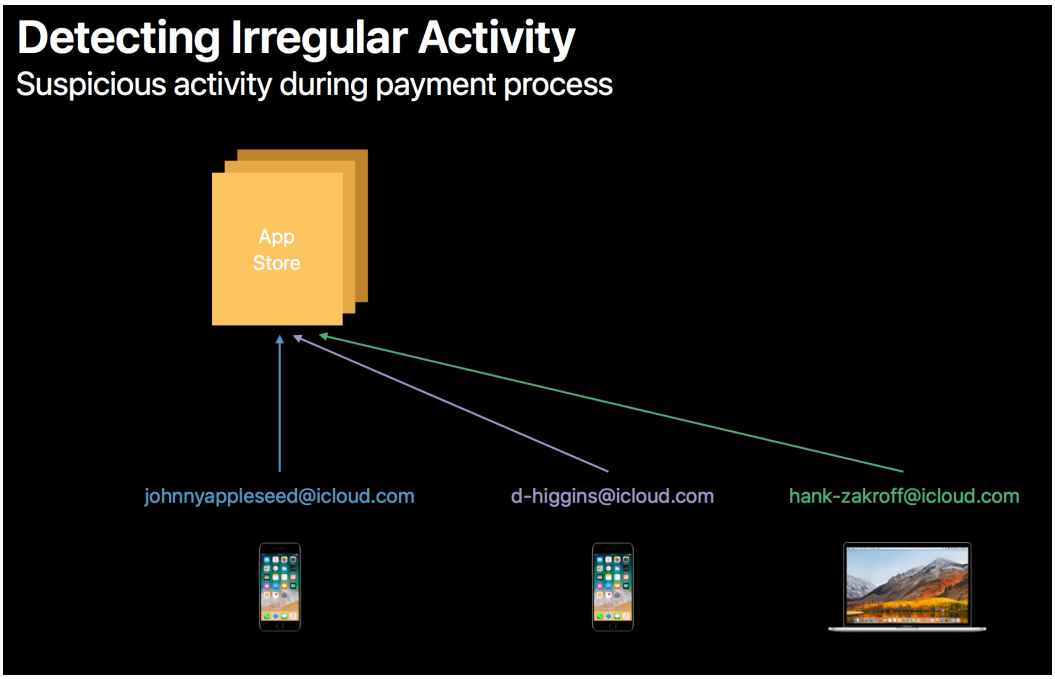
这一步也很简单， 两行代码就可以搞定。只需要把之前拿到的商品对象传到 `SKPayment` 的初始化方法中，构造一个 `SKPayment` 实例，再把这个实例加入到购买队列中即可:

```
let payment = SKPayment(product: product)
SKPaymentQueue.default().add(payment)
```

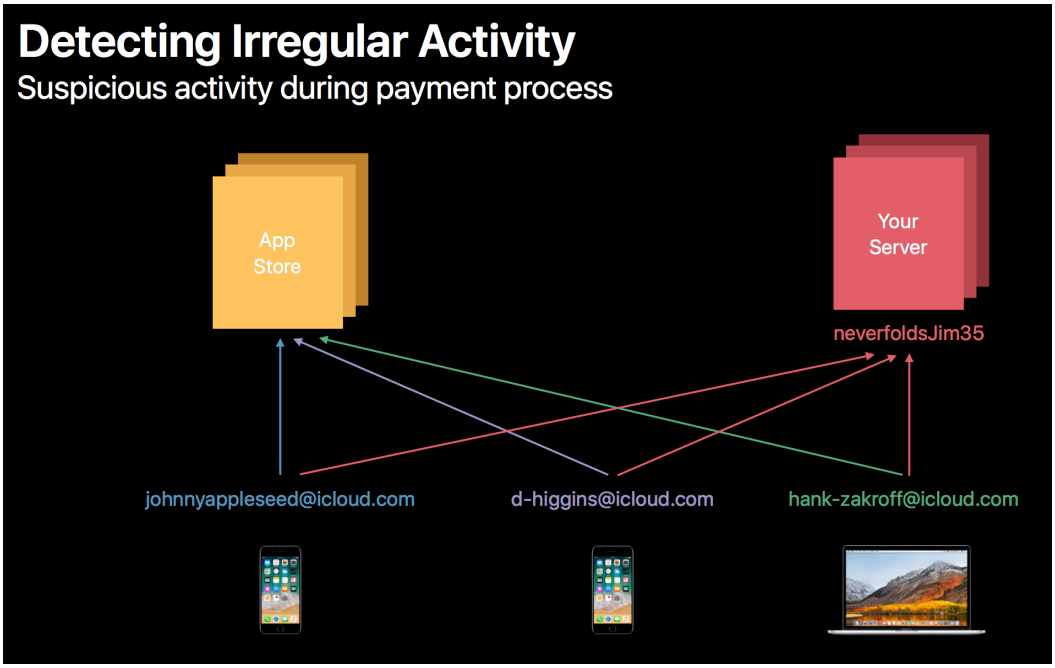
此时应用内会弹出苹果设计的购买窗口，用户只要使用指纹即可同意付款。在 iOS 上，苹果重新设计了购买页面，看起来更加美观。



苹果设计了一套强大的反作弊机制来检测可疑的交易，苹果希望在交易发生之前就把它们拦截下来，而不是在交易发生后再退款。举个例子，假设有三个不同的用户(Apple ID) 购买同样的一份商品，这个场景非常普通:



但如果这个商品是游戏中的金币，而且这三个用户购买的金币对应在游戏里都是同一个账户。换句话说，三个不同的人向同一个游戏账户充值，事情就显得很可疑了。



因此，苹果希望开发者在请求支付的同时，还携带一个用户在他们账户体系中的 ID。这种 ID 可以是透明 ID，也就是说并不一定是真实的用户账户名，只要是用户名或者其他 ID 的哈希值即可。也就是说苹果并不关心开发者上传什么具体的 ID，只要这个 ID 能唯一对应一个真实用户即可。

要实现这一步非常简单，只要在原来的请求基础上新增一行代码，设置 ID 即可:

```
let payment = SKPayment(product: product)
payment.applicationUsername = hash(yourCustomerAccountName)
SKPaymentQueue.default().add(payment)
```

有了这样的账户唯一标识，苹果可以更方便的发现异常交易并提前拦截它们。

处理购买请求

当用户的购买请求经过 StoreKit 和苹果服务器的验证后，开发者可以在回调函数中接收到。这个回调函数应该尽早的注册，因为在应用程序生命周期内的任何时刻，客户端都有可能接收到回调。所以苹果推荐在 `AppDelegate` 的 `didFinishLaunchingWithOptions` 函数内注册:

```
import UIKit
import StoreKit

@UIApplicationMain
class AppDelegate: UIResponder, UIApplicationDelegate,
SKPaymentTransactionObserver {
    func application(application: UIApplication,
didFinishLaunchingWithOptions launchOptions: [NSObject: AnyObject]?) -> Bool
{
    SKPaymentQueue.default().add(self)
    return true
}
```

Demo 中的代码比较简单，直接让 `self` 去接受回调函数。在实际开发中，可以使用一个单独的类来处理回调:

```
// 处理 SKPaymentQueueObserver 事件
// MARK: - SKPaymentTransactionObserver
func paymentQueue(_ queue: SKPaymentQueue, updatedTransactions transactions:
[SKPaymentTransaction]) {
    for transaction in transactions {
        switch transaction.transactionState {
        case .purchased:
            // Validate the purchase
        }
    }
}
```

通过检查 transaction 的状态，我们可以指定每种状态下的处理逻辑。如果状态显示已购买，我们还是应该和自己的服务器进行一次校验，确保交易真实有效而不是通过越狱后的某些插件完成的。被检验的，是一种被苹果称之为收据(receipt)的凭证，就像我们在超市购物或者饭店就餐后拿到的收据一样，每一个购买的商品都有自己的收据。这个收据由苹果签发，保存在客户端本地。

如果验证通过，就可以为用户提供购买的内容了。

另一种需要关注的状态被称为延迟状态(Deferred Stete)。如果用户把账户设置为经过允许才能购买，比如小孩子购买电子内容需要经过父母的同意。当小孩子确认购买后，还需要经过他们父母的确认，交易才能真正进行。这个过程是异步的，开发者会接收到一个延迟状态，开发者不应该让应用阻塞住，比如展示一个模态视图。也许过了几分钟，几小时甚至几周，当交易被批准后，最终还是会触发回调并且进入 `.purchased` 状态。

如果开发者想要测试这种延迟状态，他们可以在沙盒模式下把购买请求的 `simulatesAskToBuyInSandbox` 标记设置为 true:

```
let payment = SKMutablePayment(product: product)
payment.simulatesAskToBuyInSandbox = true
SKPaymentQueue.default().add(payment)
```

如果交易发生了错误，对于 `error` 对象应该具体问题具体分析，而不是简单的弹出一个警告窗口。比如用户点击了取消按钮，这也会导致一个 `error`，但我们就没必有必要再提醒用户一次，他们取消交易了。

解锁内容

如果收据验证通过，开发者需要为用户提供他们购买的内容。如果是一些需要下载的数据，可以使用此前介绍过的“按需加载资源”(On demand resources)，也可以直接在 iTunes Connect 中上传这些资源，然后直接通过 `SKProduct` 对象来下载它们。

当然，如果想要把这些资源存放在自己的服务器上也没有问题，只要注意自己管理好下载逻辑就行了，比如后台下载，断点续传等等。

结束交易

在最后一步中，如果是正常的交易，那么简单的结束它们就行了。但对于其他异常的交易，比如出现了一些 error，也要妥善的结束它们。如果是自动续期的订阅，也会经过这一步。如果交易没有被正确的结束，它们会一直停留在上文提到的队列中，每次回调函数被调用时，队列中都会有这些没有被结束的交易。

结束交易的代码非常简单，只有一行:

```
SKPaymentQueue.default().finishTransaction(transaction)
```

恢复购买

如果应用提供了 IAP 功能，在上架审核前，一定要确保自己的应用提供了 “恢复购买”(Restore) 按钮。这个按钮要和购买按钮区别开，也不能被简单的当做一种备份来处理。因为在实际场景中，确实存在很多用户拥有多个设备，需要在另一个设备上恢复此前的购买。

恢复购买仅对不可消耗的资源和自动订阅生效，如果用户购买的是可消耗资源(比如金币)，或者不可自动续期的订阅，就无法享受恢复购买功能了，这时候需要 App 自己的账号体系来同步数据。

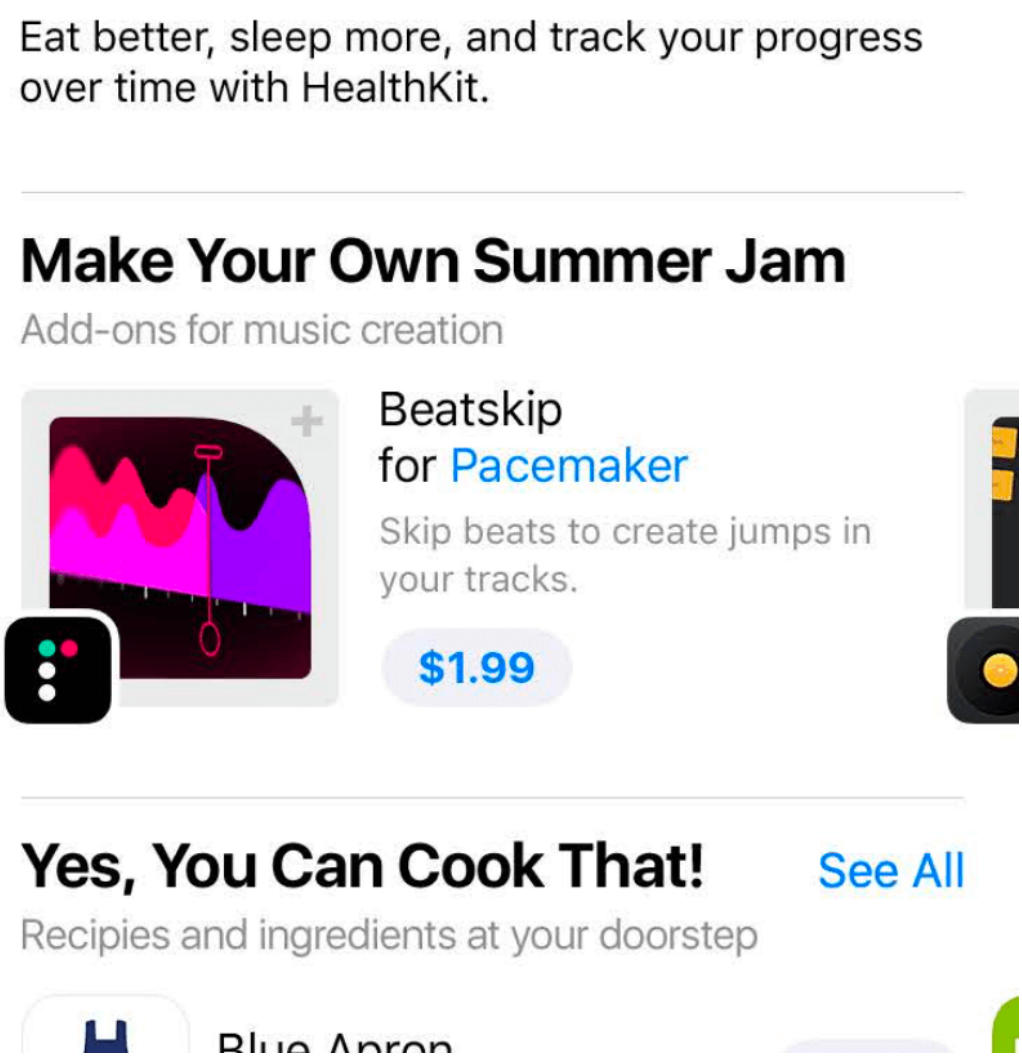
恢复购买的本质其实是调用了下面这行命令:

```
SKPaymentQueue.default().restoreCompletedTransactions()
```

它会让所有已经购买过的商品重新出现在 `SKPaymentQueue` 队列中，重复一次之前的流程。

推广应用内购买

在 iOS 11 之前，IAP 的入口相对比较难找，只有用户使用到相关功能时才会出现。但是在 iOS 11 中，IAP 会在 App Store 中出现，开发者可以在自己的应用页面内和搜索结果中展示可以购买的商品，App Store 首页甚至会推荐一些优秀的 IAP 商品。这样，用户在 App Store 中就可以购买商品，而不必打开应用。



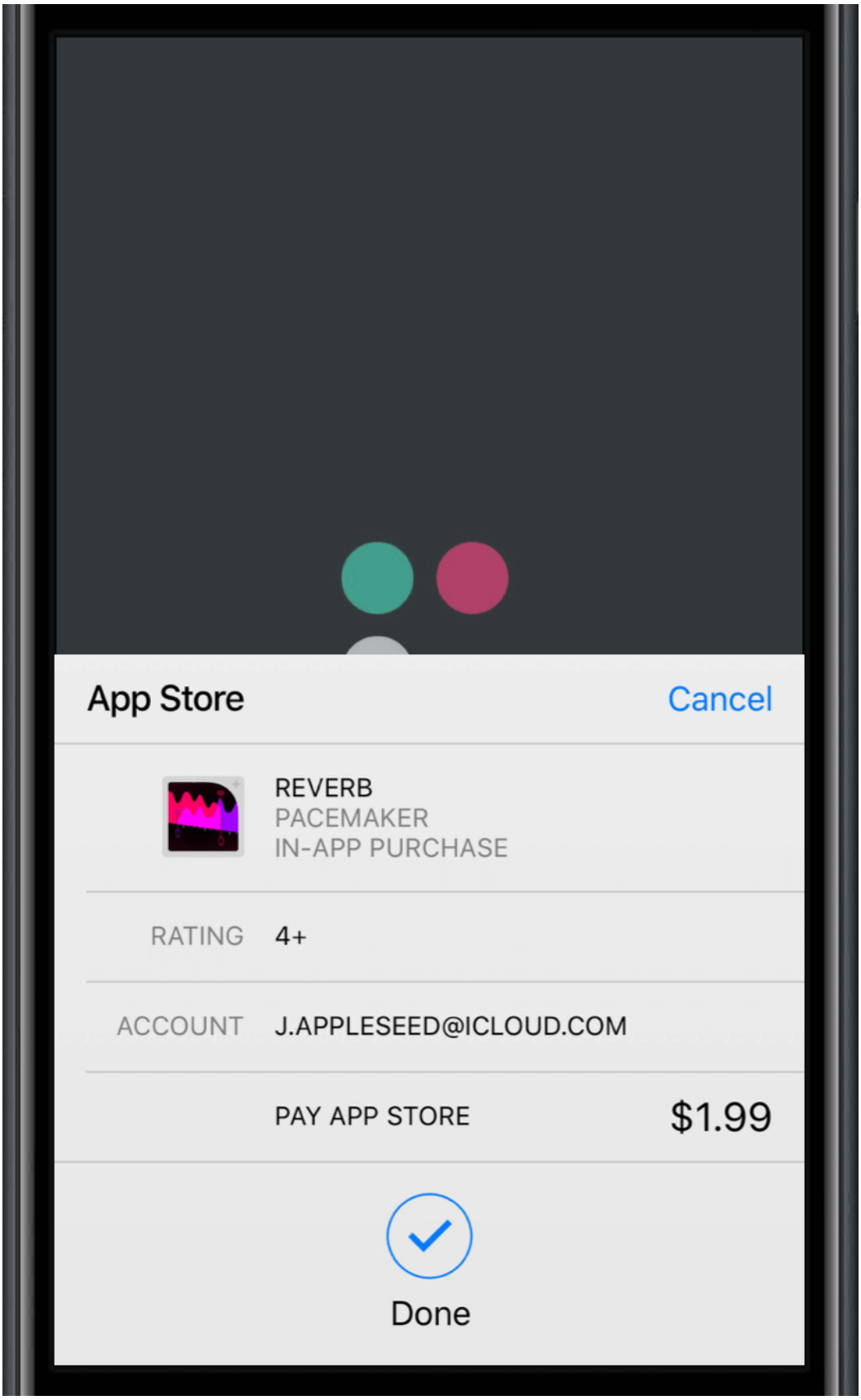
推广商品很容易，开发者可以在 iTunes Connect 中选择最多 20 个需要推广的商品，并且配上相应的图片，然后在代码中实现代理方法，处理由 App Store 发来的购买请求。在绝大多数情况下，StoreKit 承担了主要工作，开发者只需要一行代码就可以完成。

当用户在 App Store 中点击并购买商品时，App Store 会自动打开应用，通过 `SKPaymentTransactionObserver` 协议的一个新的代理方法把本次购买的相关信息传送过来。如果应用没有打开，App Store 会自动下载免费应用或者提示用户购买收费应用。这种情况下，应用无法立刻自动打开，所以下载完成后用户会收到一条通知，当用户点击通知时，应用会被自动打开并且继续之前的逻辑。

开发者需要新实现的代码如下:

```
// Continuing a Transaction from the App Store
// MARK: - SKPaymentTransactionObserver
func paymentQueue(_ queue: SKPaymentQueue, shouldAddStorePayment payment:
SKPayment, forProduct product: SKProduct) -> Bool {
    return true
}
```

方法参数中包含了本次购买的所有相关信息，如果返回值是 `true`，用户就会看到 iOS 11 中经过重新设计的、优美的购买页面:

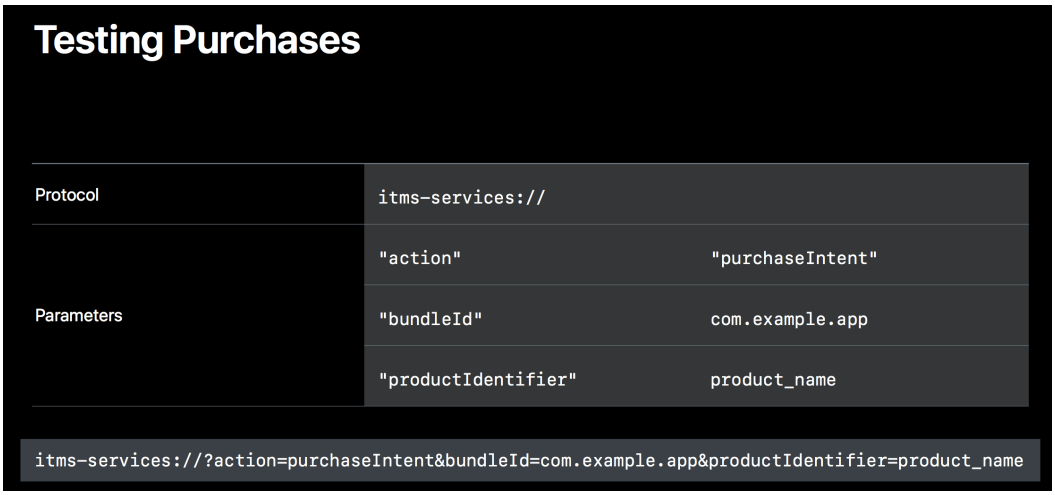


如果用户还在激活应用、注册账号，或者已经购买过这个商品，开发者可以选择暂停购买，返回 `false` 并在将来合适的时候把 `payment` 加入到购买队列中。

```swift
func paymentQueue(_ queue: SKPaymentQueue, shouldAddStorePayment payment:
SKPayment, forProduct product: SKProduct) -> Bool {
    // 先暂停购买，保存下 payment 对象
    return false
}

SKPaymentQueue.default().add(savedPayment) // 再次购买
```

如果不想继续交易，只要简单的返回 `false` 即可，但一定要给用户相关的提示。否则用户会发现点击购买按钮以后打开了你的应用，但什么事情都没有发生(其实是交易被取消了)。

如果开发者想要测试这个新的功能，可以使用 System URL 来进行。整个 URL 由 action、bundleid 和 productIdentifier 三个参数构成。第一个参数是固定的 `purchaseIntent`，第二个参数填写自己的 bundleId，第三个参数则是被购买的商品 id。



更改商品可见性

默认情况下，在 iTunes Connect 中设置的顺序就是商品在 App Store 中的展现顺序。但如果客户端有自己的逻辑，比如直到用户达成了某个先决条件，才能购买某个商品。这时候可以在一开始将商品的位置调整到比较靠后，等希望用户购买时再调整到靠前的位置。或者用户已经购买了某个高级功能后，相关的商品就不应该再出现， 因为它们无法再次被购买了。

如果想要更改可见性和顺序，开发者首先要获取默认的顺序，然后再更新:

```swift
// Reading Visibility Override of a Promoted In-App Purchase
// Fetch Product Info for Hidden Beaches pack
let storePromotionController = SKProductStorePromotionController.default()

storePromotionController.update(storePromotionVisibility: .hide, forProduct:
proSubscription, completionHandler: { (error: Error?) in
    // Complete
})
```

`SKProductStorePromotionController` 是 iOS 11 中为了推广 IAP 而新增的类，首先拿到默认的 Controller，然后把需要更新的商品作为参数传到 `update` 方法中并且设置它的可见性。这里的代码中可见性为 `.hide`，表示将这个商品隐藏。

如果想要检查更新是否生效，首先要拿到默认的 `SKProductStorePromotionController` 对象，并且调用 `fetchStorePromotionVisibility` 方法:

```
let storePromotionController = SKProductStorePromotionController.default()
storePromotionController.fetchStorePromotionVisibility(forProduct:
hiddenBeaches, completionHandler: { (visibility:
SKProductStorePromotionVisibility, error: Error?) in
    // 如果没有修改过可见性，这里的 visibility 就是默认值
})
```

更改商品展示顺序

更改顺序的方法也是类似，可以在客户端指定一个数组，并更新商品顺序。数组中的商品按照它们在数组中的位置排序，也就是说想要某个商品最先展示，就把它放到数组的第一个位置上。数组中的所有商品都将比不在数组中的商品具备更高的优先级，而那些不在数组中的商品，将按照它们在 iTunes Connect 中的顺序展示。

```
// Updating Order Override of Promoted In-App Purchases
// Fetch Product Info for Pro Subscription, Fishing Hot Spots, and Hidden
Beaches
let storePromotionController = SKProductStorePromotionController.default()
let newProductsOrder = [hiddenBeaches, proSubscription, fishingHotSpots]
storePromotionController.updateStorePromotionOrder(newProductsOrder,
completionHandler: { (error: Error?) in
    // 完成更新
})
```

用户打分、评价和开发者回复

过去，当应用发布一个新的版本时，旧版本的评分就会被重置。在 iOS 11 中，开发者可以选择重置评分的时间。

在 iOS 10.3 版本中，开发者终于可以对用户的评论作出回复了。通过 `SKStoreReviewController`，开发者可以邀请用户对应用作出打分和评价，当开发者回复后， 被回复的用户会收到通知和邮件，提醒他们查看开发者的回复，在邮件的末尾会有一个链接，允许用户修改评分和评论，苹果的统计显示，用户每次更新打分，平均提高了 1.5 颗星，可见这项功能的重要性。

`SKStoreReviewController` 提供了一个非常漂亮的应用内打分界面，用户只需要简单的选择分数并提交即可，不再需要跳转到 App Store 中。同时，开发者应该尽可能使用这个类，因为将来所有的使用模态视图来提示用户评分的行为将只能通过这个类来完成。

当然，这个类的使用也有一定的限制。比如苹果会限制弹窗的次数，用户也可以在设置里面关闭它，从而不接收任何弹窗。苹果给出的建议是，`SKStoreReviewController` 其实是一种对用户的打扰，因此尽量在某个任务结束，或者成就达成后弹出它，而不是打断用户某个正在进行的任务。它的使用非常简单:

```
//通过 SKStoreReviewController 请求用户打分和评论
if shouldPromptUser() {
    SKStoreReviewController.requestReview()
}
shouldPromptUser() -> Bool {
    return true
}
```

另一种评分机制就是之前说的，在 iOS 10.3 中提出的 Deep Link，允许用户跳转到 App Store 中编写评论。这种方式更适合一些固定的链接，比如让用户点击某个按钮后触发。链接的构造方式很简单，只要在自己应用的 url 后面加上一个 `action=write-review` 的 query 即可:

```
https://itunes.apple.com/us/app/itunes-u/id490217893?action=write-review
```

参考资料

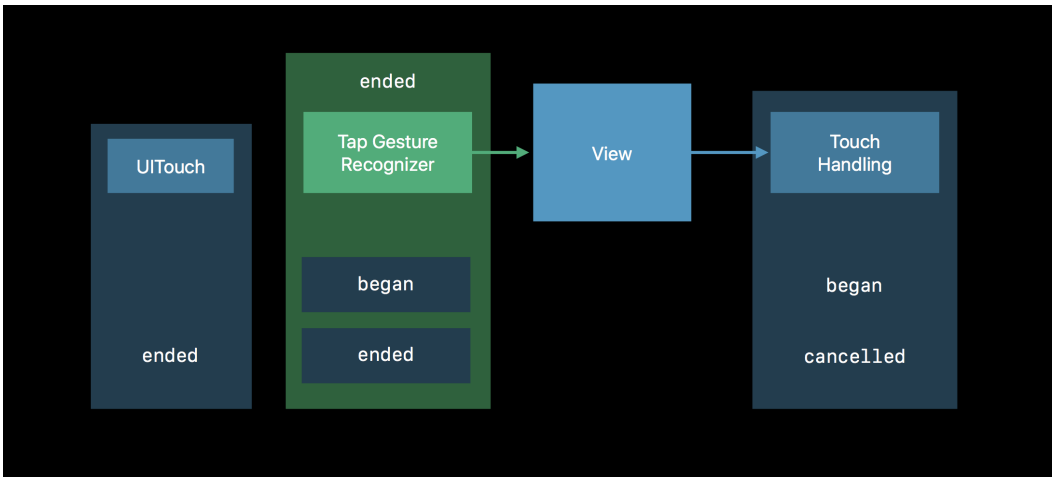
1. https://linkmaker.itunes.apple.com/
2. https://developer.apple.com/app-store/ratings-and-reviews/
3. https://developer.apple.com/in-app-purchase
4. Advanced StoreKit

精通 UIKit UIGestureRecognizer System

这片文章深入的讲解了UIGestureRecognizer系统，在iOS11中新的手势API，以及Drag and Drap对现有的手势识别器有什么影响。

1. UITouch 和 UIGestureRecognizer

UITouch是手指触摸屏幕这一事件的代理，从触摸开始(began)到取消(cancelled)再到结束(ended)，它代表了和屏幕的一个完整的交互行为。而UIGestureRecognizer是一个抽象类，可以通过设定目标(target)和动作(action)使其作用于视图(UIView)上。iOS定义了所有常见的手势，例如点击(tag)，滑动(pan)或者捏合(pinch), 并且提供了相应的方法来协调处理不同手势。 为了更好的理解，我们先看来一个例子。 一个UIView，以及一个Touch Handling作为响应器(Responser)作用于这个View上。我们先来一个点击动作，当手指触摸在这个View上时，一个UITouch开始于状态*began*。系统会把这个状态发送给所有相关的手势识别器(UIGestureRecognizer)，在本例中也就是这个点击手势识别器(UITapGestureRecognizer). 点击识别器会从初始的默认状态*possible*变为*began*, 由此触发了Touch Handling中的方法`began`。 当手指离开View，UITouch状态变成*ended*，点击手势识别器状态随后也变为*ended*, 并且触发Touch Handling中的`cancelled`方法。如图1.



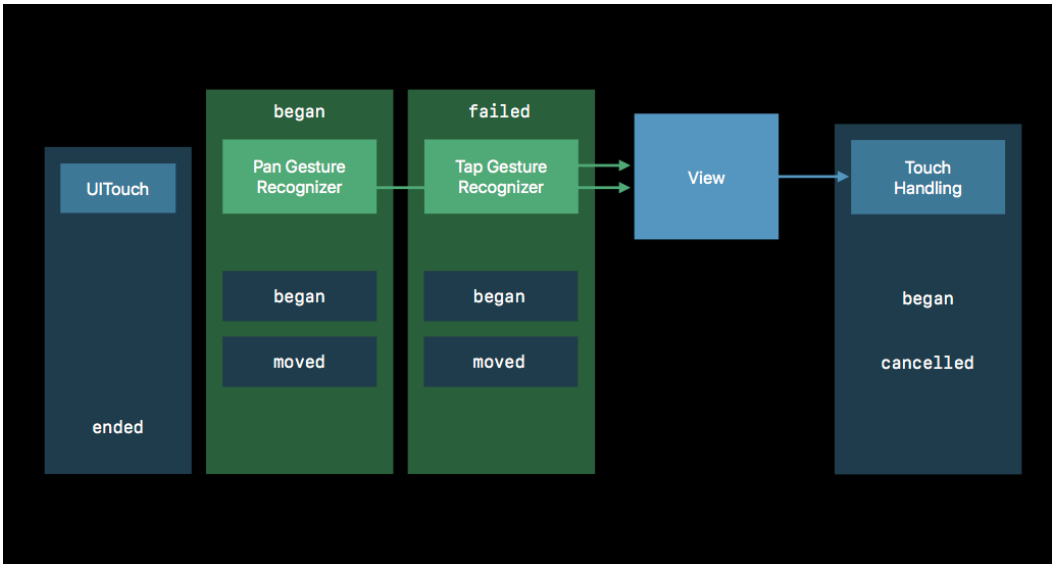
UIGestureRecognizer三个经常用到属性以及它们的默认值，如图2

```
// Influencing responder based touch handling

class UIGestureRecognizer : NSObject {

    open var delaysTouchesEnded: Bool // default is true.
    open var cancelsTouchesInView: Bool // default is true.
    open var delaysTouchesBegan: Bool // default is false.
}
```

需要注意的是，如果`var delaysTouchesBegan`设置为`true`，那触摸响应将会被忽略。

同步识别

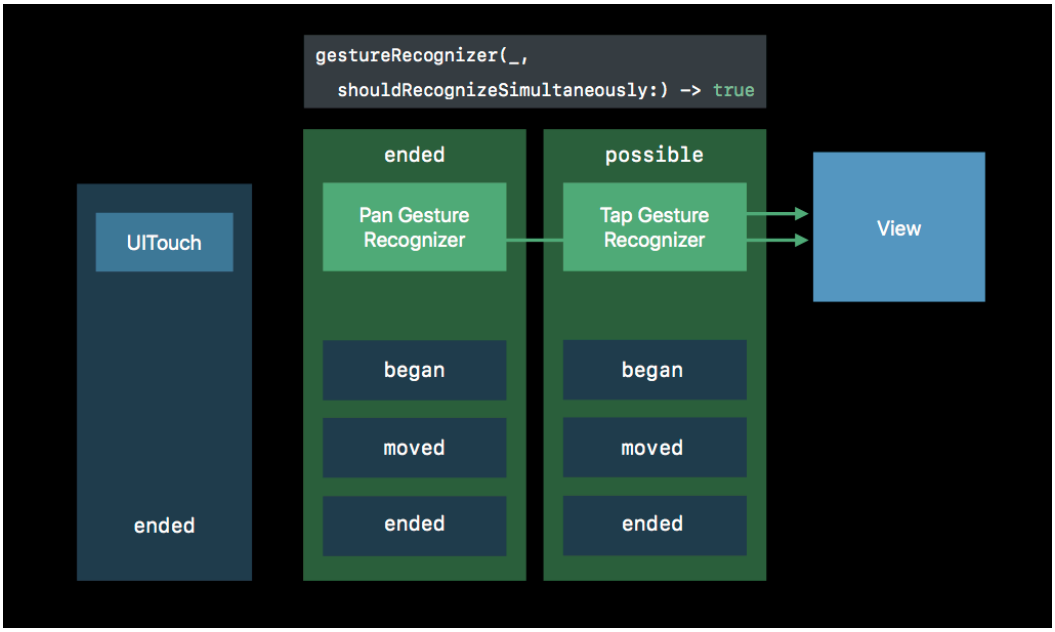
让我们再看另外一个例子。当同时存在两个手势识别器时，例如一个是PanGestureRecognizer另一个是TapGestureRecognizer, 如果手指在View上拖动，即使移动距离没有超过Tap手势识别器允许的最大范围，也就是说动作仍然在Tap手势所允许的范围内，然而Tap手势识别器依然会在Touch动作尚未结束前变成*failed*状态，从而导致Pan手势识别器被终止，如图3。



这是因为系统默认只允许一个手势识别器被执行。针对这种情况就需要用到`UIGestureRecognizerDelegate`中的方法：

```
func gestureRecognizer(_ gestureRecognizer: UIGestureRecognizer,
shouldRecognizeSimultaneouslyWith otherGestureRecognizer: UIGestureRecognizer)
-> Bool
```

这个方法可以让两个手势识别器保持同步识别从而避免上述的问题出现。需要注意的是，尽管返回`true`可以保证两个识别器同步识别，但返回`false`却不一定能阻止同步识别，因为在另外的手势识别器代理中该方法可能也被设为`true`，如图4。



Failure Requirements

然而，因为同步识别使得两个手势识别器都被触发，但以什么顺序发生却无法被控制，由此可能会产生一些并不是我们期待的结果。

如果我们想要一个识别器等到另一个识别器失效后再发送动作那么就需要用到失效请求(Failure Requirements). 这涉及到处理两个手势之间的关系，例如如果不希望在双击(Double Tap)的同时触发两次单击(Single Tap)， 或者希望等到Tap失效后再触发Pan的动作，我们可以用到`UIGestureRecognizer`中一个比较静态的方法：

```
func require(toFail otherGestureRecognizer: UIGestureRecognizer)
```

或者使用`UIGestureRecognizerDelegate`中动态一点的两个方法：

```
func gestureRecognizer(_ gestureRecognizer: UIGestureRecognizer,
         shouldRequireFailureOf otherGestureRecognizer: UIGestureRecognizer) -
> Bool

func gestureRecognizer(_ gestureRecognizer: UIGestureRecognizer,
       shouldBeRequiredToFailBy otherGestureRecognizer: UIGestureRecognizer) -
> Bool
```

从这个例子可以看出，Failure requirement的作用就是阻止另外一个识别器发送多余动作。

Hit Testing

为了确定由哪个View来响应手势，需要用到“命中测试”(Hit testing). 当触发动作发生时，命中测试使用*反向深度优先搜索*算法，从根视图开始不停的询问hitTest方法触摸是否在某个View中，直到找到触摸作用最深层的那个View，系统保存它并把这个触摸事件分配给这个View。这里的最深层其实是指的最后被渲染的View，我们可以把根视图window看作做最浅层，那么最深层就是离我们最近的那层。

如果你想放大或者缩小命中测试下面的两个方法通常需要被重载：

```
class UIView : NSObject {

        func hitTest(_ point: CGPoint, with event: UIEvent?) -> UIView?

        func point(inside point: CGPoint, with event: UIEvent?) -> Bool
}
```

另外， 一些UIView的属性也会影响到命中测试的运行, 例如下面的这些：

```
class UIView : NSObject {

        var isUserInteractionEnabled: Bool

        var alpha: CGFloat
        var isHidden: Bool

        var isMultipleTouchEnabled: Bool
}
```

新添属性

在iOS11中UIGestureRecognizer有了一个新的属性`name`, 如图5。

```
Debugging only
@available(iOS 11.0, *)
open var name: String? // name for debugging to appear in logging
```

这个属性主要是方便在调试的时候区分不同的手势而不需要像以前一样去查看每个动作方法。

自定义手势识别器 (Custom UIGestureRecognizer)

对于自定义手势，一个原则是"晚点开始，尽快结束"（begin late and fail fast）. 如果开始的过早那么别的手势可能就无法被执行，类似的，自定义手势应该尽早的失效以免影响到别的手势的失效请求(Failure Requirements).

对于不属于这个手势的触摸应该使用方法`ignore(_ touch:, for: event:)`去忽略掉。 最后需要注意的是不要忘记使用`touchCancelled(_ with:)`方法去取消一个触摸。由于drag and drop功能的加入，这个方法会被使用的更加频繁.

2. 控制系统手势交互(System Gesture Interaction)

有时，一些特殊的系统手势会阻碍到你App中的应用手势，例如当用户从屏幕底部向上滑动时，他有可能是想调出控制中心来进行一些快捷操作，也有可能只是想向上拖动App中的视图。在此之前系统只能靠一些推测来判定用户意图。例如，如果顶部的状态栏被隐藏那么从底部上划时系统手势，也就是调出控制中心这一手势也会能推迟。当然这种猜测并不总是准确的。

在iOS11中对于这种情况不再需要猜测，你可以在你的App中决定系统手势什么时候被延迟，在哪里被延迟。对此你需要用到一个新的API: `preferredScreenEdgesDeferringSystemGestures() -> UIRectEdge`, 如图6。

```
class MyViewController: UIViewController {

    // override to return which screen edges to defer system gestures
    override func preferredScreenEdgesDeferringSystemGestures() -> UIRectEdge {
        return deferControlCenter ? .bottom : UIRectEdge()
    }

    // call whenever your method would return a different screen edge
    var deferControlCenter : Bool {
        didSet { setNeedsUpdateOfScreenEdgesDeferringSystemGestures() }
    }

}
```

在UIViewController中重载这个方法能让你设定在哪些地方系统手势会被延迟，从而确保你App内的应用手势可以被优先识别出来。如果你想动态的改变这个设定，只需要调用`setNeedsUpdateOfScreenEdgesDeferringSystemGestures()`这个方法来告知系统。

然而，如非沉浸式的游戏或画画的应用，尽量不要推迟系统手势。因为这样一方面可能会有违用户的使用习惯，另外一方面有很多应用手势并不会和系统手势冲突，例如轻点(Tap), 捏合(Pinch)，旋转(Rotate)，长按(Long Press)，对于这种手势完全没必要去控制系统手势。

3. UIDragInteraction对现有手势识别的影响

Drag and Drop 是iOS11最新引入的一项技术，这使得用户可以用一种更有趣的方式和App进行交互。在此我们主要介绍一下添加了拖动(Drag)功能之后会对已存在的手势有什么影响。

首先我们先了解一下如何添加拖动(Drag)。在iOS11中有一个新的类`UIDragInteraction`, 这个类使用起来非常简单，你只需要指定一个代理(delegate)并把这个`UIDragInteraction`添加到View上就行，如图7。

然而，如果我们在同一个View上再添加一个长按手势(UILongPressGestureRecognizer)，那么当我们长按并拖动这个View时，长按手势会被推迟，长按手势触发的动作自然也无法被调用。如果我们希望长按手势被识别，那么我们需要在这个View长按并保持住，这样长按手势的状态就会变为`began`。

在例如iPad上分屏这样的紧凑视图界面中，如果添加了拖动功能，那么长按手势将会被推迟到触摸结束，因此我们需要长按然后松开手指才能使得长按手势变成`began`的状态。

尽管适配`UIDragInteraction`是非常简单的，但我们还有几点建议：

- 首先是你要检验现有的手势和动作是否会受到拖动手势的影响。
- 其次你需要去处理手势识别器的`cancelled`状态，因为当拖动开始时触摸事件会被标记为`cancelled`，很多时候我们都必须对这个状态作出相应的处理。
- 最后一点要注意的是，当拖动进行时你的App是处于完全交互的状态，所以可能会产生一些意想不到的效果，开发人员必须要对于提高警惕。

总结

本文重点介绍了iOS中多个手势之间的相互影响和作用，以及在将要到来iOS11中如何更好的控制系统手势给用户更好的交互体验。最后讲解了新添加的Drag功能对已有的手势有带来哪些影响，以及开发人员应该如何更好的适配这个新功能。

参考

视频地址：Modern User Interaction on iOS

PPT地址：Modern User Interaction on iOS

Photos APIs 新特性

本节要介绍的是 Photos APIs 的一些新特性。简单的概括有下面这几点内容：

- UIImagePickerController 的大幅优化
- 授权模式的改进
- 动图的支持
- iCloud 照片图库的优化
- 照片项目的扩展

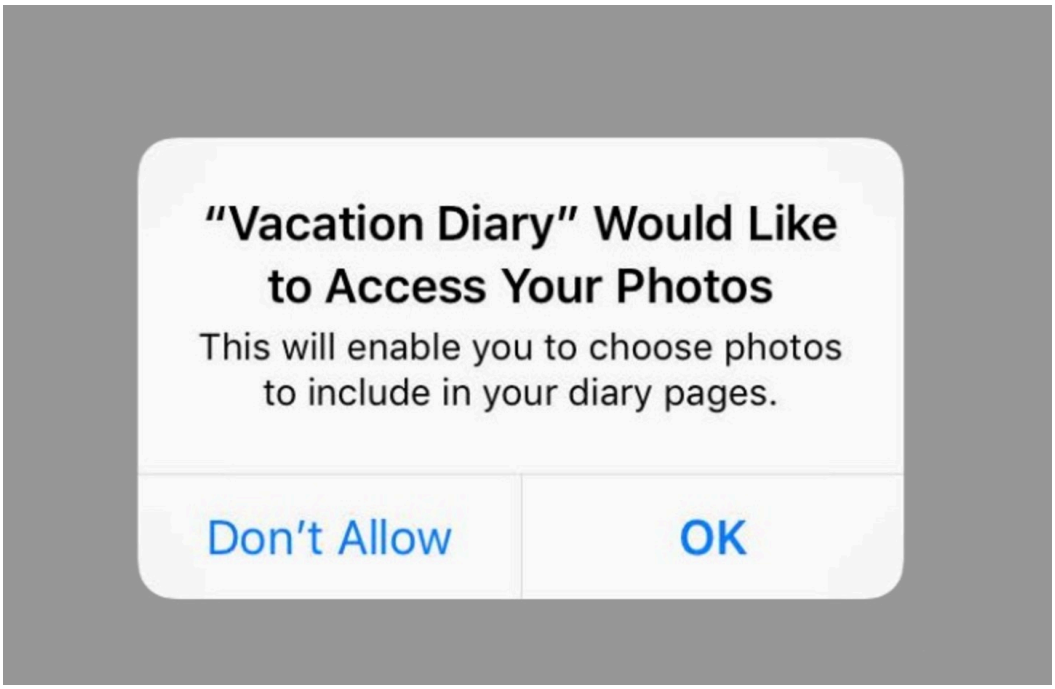
后续内容，会对这几个点依次展开。

UIImagePickerController 的大幅优化

`UIImagePickerController`是系统提供的和相册及相机交互的一个类，通过这个类，你可以在应用中选择照片和视频。在 iOS 11 里，图片选择器有了许多的改进和新功能的引入。

隐私授权的改进

一直以来，Apple 十分关注用户的隐私安全。所以，之前在任何情况下，如果获取 Photos 资源，都需要获取用户的授权才可以进行。正如下面弹窗这样，请求用户的授权。



正因为授权过程的存在，使得应用程序与用户之间产生了矛盾。对于用户而言，需要他们打开一级隐私，这不是用户想要的；而对于应用程序来说，应用在未获取权限的情况下，无法执行相应的程序和操作，即便它自身有很多优秀的功能，都会因为未授权而无法使用。

在 iOS 11 中，如果通过`UIImagePickerController`访问相册资源，这个`警告弹窗`不会再出现，会直接进行程序运行。看到这里，你或许会问：那用户的隐私保护怎么办?

首先需要介绍一下`UIImagePickerController`新的授权模式。自 iOS 11 开始，`UIImagePickerController`成为了一个自动授权 API。也就是说，当应用程序要显示 API 的内容，将会是从一个`自动处理的沙盒`和`安全环境`中获取，应用不再访问用户的`Photo Library`。

并且，只有用户本人可以和`UIImagePickerController UI`进行互动。当用户做出一个选择，系统会取出选中的照片或视频，发送到应用中。这样就消除了前面提出的在应用中因为授权而产生的矛盾，同时这也让用户有了更高级别的隐私。因为不存在授权，也就不会再有请求授权。使用起来更为方便了。

元数据的获取更为方便

Photos 拥有丰富的`元数据(metadata)`，内容包括创建日期、照片的格式，及一些其他不同类型的元数据。在 iOS 11 中，获取这些信息变得容易了很多。系统提供了一个全新的键值`UIImagePickerControllerImageURL`，会包含所有`UIImagePickerController`的结果。我们可以使用这里的 URL，将对应数据读入应用并按照需要进行处理。该 URL 是`文件 URL`，指向一个应用中的`临时文件`，如果之后想对文件继续操作，建议把文件移动到更永久的文件路径中。

```
public func imagePickerController(_ picker: UIImagePickerController,
didFinishPickingMediaWithInfo info: [String : Any]) {
  if let imageURL = info[UIImagePickerControllerImageURL] as? URL {
        print(imageURL)
    }
}
```

HEIF 图片格式的引入

iOS 11 中，Photos 引入了一种新的图片格式`HEIF`。同时，Apple 意识到生态系统完全接受`HEIF`需要一段时间，考虑到新类型图片格式的兼容性。Apple 为`UIImagePickerController`提供了一个新属性`imageExportPresent`，让兼容过程变得更为容易。

```
var imageExportPreset: UIImagePickerControllerImportExportPreset { get set }

let imagePicker = UIImagePickerController()
imagePicker.imageExportPreset = .compatible
self.present(imagePicker, animated: true, completion: nil)

let imagePicker = UIImagePickerController()
imagePicker.imageExportPreset = .current
self.present(imagePicker, animated: true, completion: nil)
```

`imageExportPresent`拥有两种类型：

- .compatible (兼容模式)
- .current (当前模式)

在`compatible (兼容模式)`下，如果用户选中的源图片是`HEIF 格式`，系统会通过转换，提供一个 JPEG 格式的图片。当然，JPEG 是该属性的默认值，如果不需要有什么改变，就不用再做更多的事情。

如果，需要获取的照片格式与拍摄时的格式相同，只需把属性值设为`current (当前模式)`，这样就会得到与 Photo Library 里相同格式的图片，包括`HEIF 格式`。

视频文件的获取更为方便

iOS 11 中，对视频选择的功能，也有了很好的改进。暂时把这部分内容放在一边，先来简单了解下`AVFoundation`。`AVFoundation`是 Apple 提供的框架，用于`丰富编辑`及`照片播放`。通过`AVFoundation`导出的素材可以拥有丰富的格式。

值得称赞的是，`UIImagePickerController`现在也有了类似的功能，引入了一个新属性`videoExportPreset`。

```
var videoExportPreset: String { get set }
```

你可以通过这个方法来告诉系统，你所选中的视频需要以哪种格式返回。这样，你就可以轻松得到预设格式的资源内容了。

我们来看一个例子：

```
import AVFoundation
let imagePicker = UIImagePickerController()
imagePicker.videoExportPreset = AVAssetExportPresetHighestQuality
self.present(imagePicker, animated: true, completion: nil)
```

如上代码中，首先，导入`AVFoundation`；接着，创建一个`UIImagePickerController`实例，并描述我们想要资源文件以哪种格式返回（这里我们请求的是最高品质）；之后显示选择器。

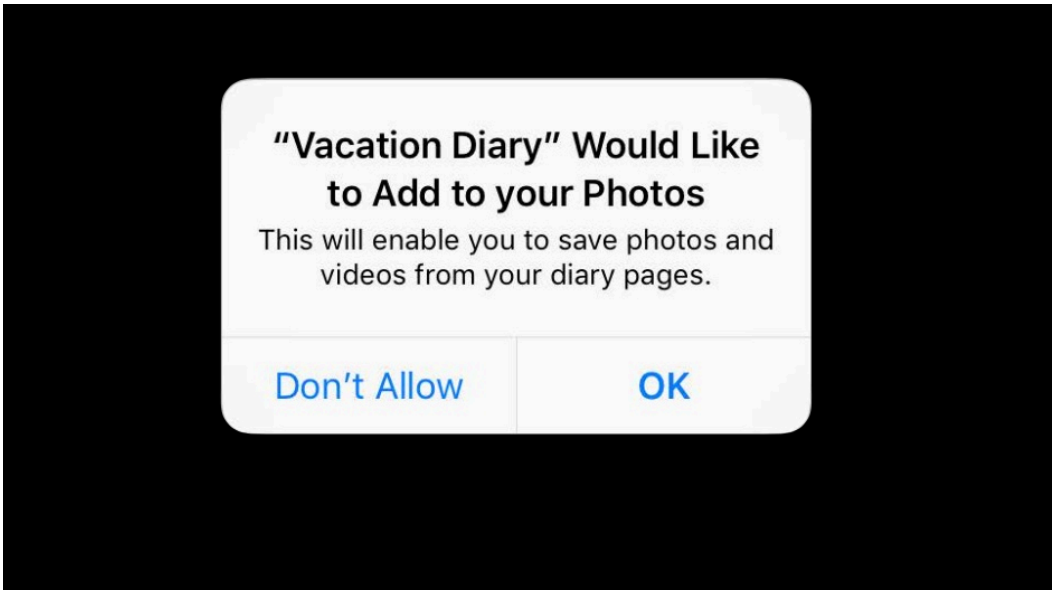
当用户做出选择时，无论是什么格式，系统都对其进行交叉编译，得到匹配格式，之后返回给用户。关于可用预设的完整清单，可以通过接口`AVAssetExportSession`查看。

照片和视频的保存有了新的隐私模式

前面，通过一些巧妙的设计，在保护用户隐私的情况下，已经实现了无缝选取。实际上，iOS 11 也对图片和视频的保存做了很多的优化。

在 iOS 11 中，保存一张照片或一段视频到用户的图片库中，系统提供了一个全新的安全模型及权限级别。`UIImagePickerController`对于保存`图片资源`及`视频资源`分别提供了权限级别。一个是`UIImageWriteToSavedPhotosAlbum`，另一个是`UISaveVideoAtPathToSavedPhotosAlbum`。

```
public func UIImageWriteToSavedPhotosAlbum(_ image: UIImage, _
completionTarget: Any?, _ completionSelector: Selector?, _ contextInfo:
UnsafeMutableRawPointer?)


public func UISaveVideoAtPathToSavedPhotosAlbum(_ videoPath: String, _
completionTarget: Any?, _ completionSelector: Selector?, _ contextInfo:
UnsafeMutableRawPointer?)
```



这两种方式都只会请求`添加授权`，对于用户来说`添加授权`是很小的要求。因为这个权限只允许添加内容到用户的 Photo Library，而不涉及到读取权限。所以，很大程度上，用户会愿意给出这个权限。

PHAsset 获取的改进

我们来看一个例子：

```
    public func imagePickerController(_ picker: UIImagePickerController,
didFinishPickingMediaWithInfo info: [String : Any]) {
        if let asset = info[UIImagePickerControllerPHAsset] as? PHAsset {
            print(asset)
        }
    }
```

上述代码中，我们实现了一个代理方法。在获得结果词典时，有了一个全新的键，键名为`UIImagePickerControllerPHAsset`。取得该键的值，将会得到对应的资产对象，可以对其进行自由使用。

通过这些改变，增强了用户的隐私保护，也让`UIImagePickerController`成为更强大而功能齐全的 API，满足了市面上大部分应用的需求。然而，有时会出现需要和照片框架进行更深入集成的需求，在这些场景下，Apple 推荐使用`PhotoKit`。

PhotoKit

和照片相关的应用，一直以来是 App Store 里最受欢迎的一类。这一次，`PhotoKit`做了一些改进，可以让你写出拥有更棒用户体验的新功能。

Live Photo 介绍

首先一起了解下`Live Photo`的效果。Live Photo 效果包含：

- **循环效果**
- **弹跳效果**
- **长曝光效果等**

其中`循环效果`，是通过仔细分析视频帧，并无缝地和这些视频帧无止境循环缝合在一起；`弹跳效果`，它的工作原理和`循环效果`也是相似的；最后是`长曝光效果`，它将分析 Live Photo 的视频帧，创造令人惊艳的静物。

现存的`PhotoKit`媒体类型有这些：

```
enum PHAssetMediaType : Int {
        case unknown
        case image
        case video
        case audio
    }

    struct PHAssetMediaSubtype : OptionSet {
        static var photoPanorama
        static var photoHDR
        static var photoScreenshot
        static var photoLive
        static var photoDepthEffect
        static var videoStreamed
        static var videoHighFrameRate
        static var videoTimelapse
    }
```

如果用户拍摄了一段视频，会想在应用中进行观看，并将拍摄的内容以`视频方式`使用。如果用户拍摄了一张`Live Photo`，同样会想要在应用中看到内容以`Live Photo`的方式呈现。为此，iOS 11 提供了三种媒体类型来实现对应目标：

- **image**
- **video**

- **photoLive**

`Live Photo`效果比较复杂。为此，iOS 11 中引入了全新的`PHAsset`属性`playbackStyle`，让你可以简单实现`Live Photo`的播放。

```
    class PHAsset : PHObject {
        var playbackStyle: PHAssetPlaybackStyle { get }
    }
    enum PHAssetPlaybackStyle : Int {
        case unsupported
        case image
        case imageAnimated
        case livePhoto
        case video
        case videoLooping
    }
```

`playbackStyle`属性，是`唯一`可以用来查看和决定要使用什么样的图片管理器 API、用什么样的视图来表现、以及为该视图设置什么样的 UI 限制。同时，Apple 更新了 PhotoKit 示例应用，包含所有这些新的播放风格。这里介绍下其中的三种，它们和前面提到的`Live Photo 效果`相关。从`imageAnimated`开始。

Animated Image

```
    imageManager.requestImageData(for: asset, options: options) {
        (data, dataUTI,  orientation, info) in    // 使用示例项目中的
animatedImageView
        let animatedImage = AnimatedImage(data: data)
        animatedImageView.animatedImage = animatedImage
    }
```

iOS 11 有了一个期待已久的新功能。现在，在内置应用“照片”中支持了`动画 GIF`的播放。如果要在你的应用中播放 GIF，只需要从图像管理器请求图像数据，然后使用图像 IO 和 Core Graphics 进行播放。接下来是`Live Photo`。

Live Photo

```
    imageManager.requestLivePhoto(for: asset, targetSize: pixelSize,
contentMode: .aspectFill, options: options) {
        (livePhoto, info) in    // 使用示例项目中的 PHLivePhotoView
        livePhotoView.livePhoto = livePhoto
    }
```

`Live Photos`一直很受用户的关注，如何在应用中更好地呈现它们，非常重要，也非常简单。在如上的这个例子里，首先从图像管理器`请求`一张`Live Photo`，之后设置`PHLivePhotoView`。在你的应用里，用户可以通过轻触播放一张`Live Photo`，正如用户在内置`“照片”应用`里的操作一样。

Looping Video

```
    playerItem, info in
    DispatchQueue.main.async {
        let player = AVQueuePlayer()
        playerLooper = AVPlayerLooper(player: player, templateItem:
playerItem)
        playerLayer.player = player
        player.play()
    }}
```

在今年所推出的视频循环中，既包括`弹跳效果`，也包括`Live Photo`的循环效果。现在，在你的应用里播放这些和播放普通视频非常相似。可以请求播放器项目，并使用`AVFoundation`播放，还可以使用`AVPlayerLooper`取得循环效果。 可见，表现用户的媒体变得更为轻便，以他们真正想表现的方式，你也可以在自己的应用中，对这些全新的媒体类型，更为创新地表现。

iCloud 照片图库的改进

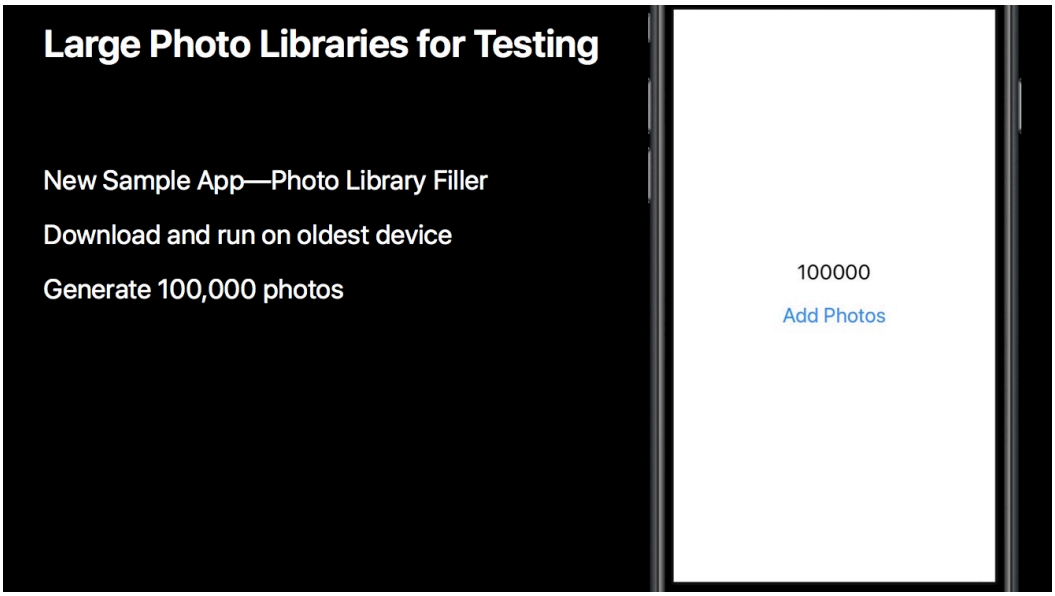
“iCloud 照片图库”可以与“照片”应用完美搭配使用。当用户开启“iCloud 照片图库”时，用户的照片和视频会被安全上传到 iCloud 中，同时这些更改会同步到用户的其他设备中。自动上传 iCloud 操作的触发条件是，设备连接到 Wi-Fi 且电量充足。根据用户的网络情况，在所有设备和 iCloud.com 上看到同步照片和视频所需的时间可能会不同。

使用 iPhone 拍照的用户，也常会使用“照片”相关的第三方应用。这些用户，大致可以分为 3 类：轻度用户、中度爱好者和重度专业用户。对于重度用户而言，由于自身图库中有很多内容，在第一次使用应用时，需要加载大批量的照片，这个过程中会十分耗时。而且这种耗时的加载状态，会对应用的用户体验大打折扣。

在 iOS 11 中，针对如何更快速高效地操作“大型照片图库”这一点做了优化，后面的内容会依次展开描述。

创建一个用于测试的“大型照片图库”

前面提到，如果图库中有大量内容，在应用加载数据时，会处于长时间的加载状态。而你如果想创建一个拥有大批量图片的图库，还是有些难度的。友好的是，Apple 为开发者提供了一个用户创建图库的示例应用：Photo Library Filler 。下载该应用并安装到测试设备，点击`“Add Photos” 按钮`，它便会迅速生成一个拥有大批量图片的图库供测试使用。



到这里，你就拥有了一个可用于测试的“大型照片图库”。

现在，如何从“照片图库”提取图片

看下面这段代码：

```
let assets = PHAsset.fetchAssets(with: options)
```

这个方法用于从用户的“照片图库”提取图片，等号右侧用于提取资产，左侧为提取结果。

```
let options = PHFetchOptions()options.predicate = NSPredicate(format:
"isFavorite = %d", true)
options.sortDescriptors = [NSSortDescriptor(key: "creationDate", ascending:
true)]
let assets = PHAsset.fetchAssets(with: options)
```

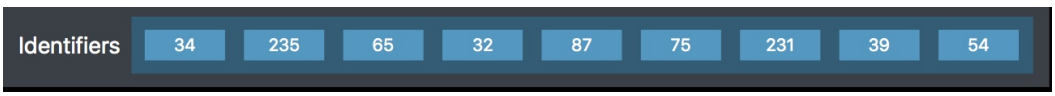
在上述代码中，首先提取库里所有的 Asset，并进行筛选，筛选条件为isFavorite = true，之后按照对应的创建日期进行排序。这时，如果在这些自定义提取里发现耗时，那么简化这里的筛选条件会十分必要。不同的筛选方式，可能会意味着查询耗时的巨大差异。造成这种差异的原因是你的操作可能在数据库优化路径之外进行，同时又试图回到优化路径中，这样就产生了不同的耗时差距。

比这种自定义提取更好的是，尽可能避免这种操作。例如下面这个例子中，我们实际上提取的是用户最喜欢的智能相册。然后在智能相册里提取 Asset。这样既可以使用已有的关键字和排序描述符，还可以保证操作是在数据库优化路径中进行的。

```
let smartAlbums = PHAssetCollection.fetchAssetCollections(with: .smartAlbum,
subtype: .smartAlbumFavorites, options: nil)
let assets = PHAsset.fetchAssets(in: smartAlbums.firstObject, options: nil)
```

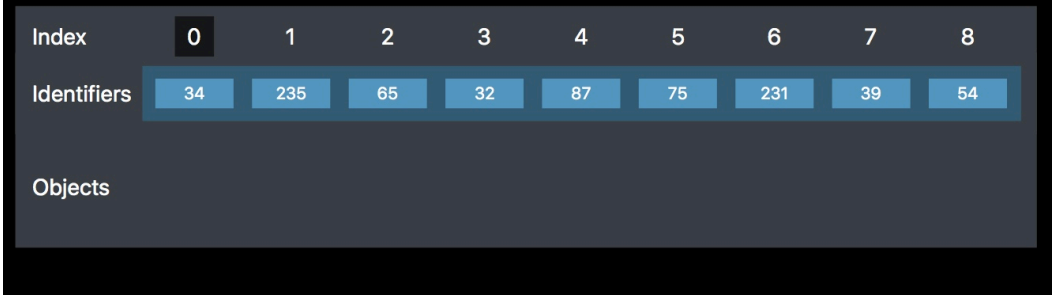
接着，看下返回结果。返回对象是一个PHFetchResult类型的对象。PHFetchResult 类型非常像一个数列，并且可以像数组一样来使用。但从内部实现来看，PHFetchResult和数列的工作机制还是完全不同的。并且这也是PhotoKit在大型图库操作方面，能够如此快速高效的原因之一。

我们来看看它的内部工作机制。

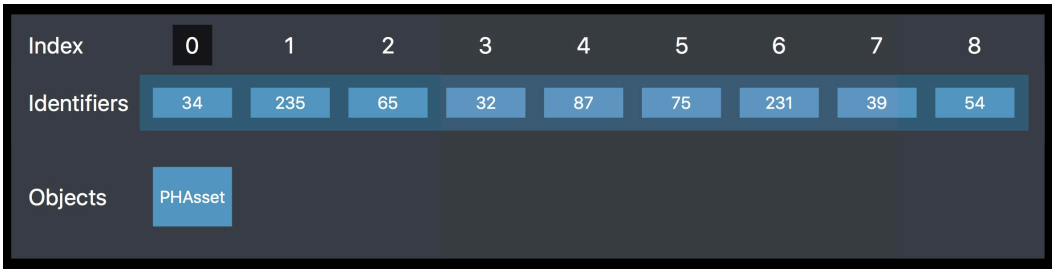


最开始，它只包含一个标识符列表。这意味着可以迅速返回对应的 Asset。但开始使用后，有更多工作必须执行。我们以一个枚举作为例子。

```swift
let assets = PHAsset.fetchAssets(with: options)
assets.enumerateObjects { (asset, index, stop) in
    // do something with the asset
}
```



在这里，我们从`索引 0 开始`枚举。目前只有一个标识符，你还需要从数据库里提取元数据。为此创建一个`PHAsset 对象`，以便将 Asset 的元数据返回给你。



实际上，同时也创建了`一个批处理`。

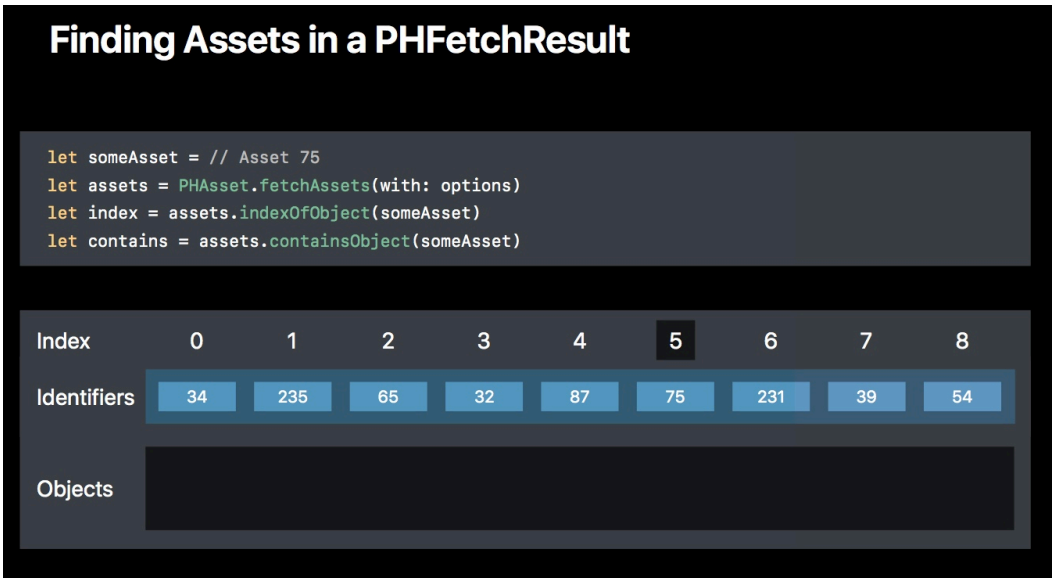
当我们继续枚举时，`索引 1 和 2`实际已经在内存中了。枚举继续，它将访问硬盘，获取后续 Asset 的元数据。

在提取结果量级较小的情况下，这样的提取，并不会有太大的影响。但如果提取结果包含`10万个 Asset`。其中，每一批都需要`占用几 kb 内存`。如果是`10w 批`，那将会产生`几百兆`的内存用量。更糟糕的是，每一批都需要`几毫秒`的提取时间，如果有`10w 批`，就需要`消耗 10s`来枚举这样的一个大型提取结果。所以，应该尽量`避免枚举操作`。

在 PHFetchResult 中查询 Asset 更优的方式

实际开发过程中，枚举操作总是可能会出现的。这里列举一个例子。现在，你需要从一个提取结果中查询一个`Asset 的索引`。

第一种方式，可以通过枚举该提取结果，通过`“等于”`比较返回的对象，来获取对应`Asset 的索引`。但是，枚举会非常耗时，所以更好的方法是通过另一种方式，使用`高端 API`。



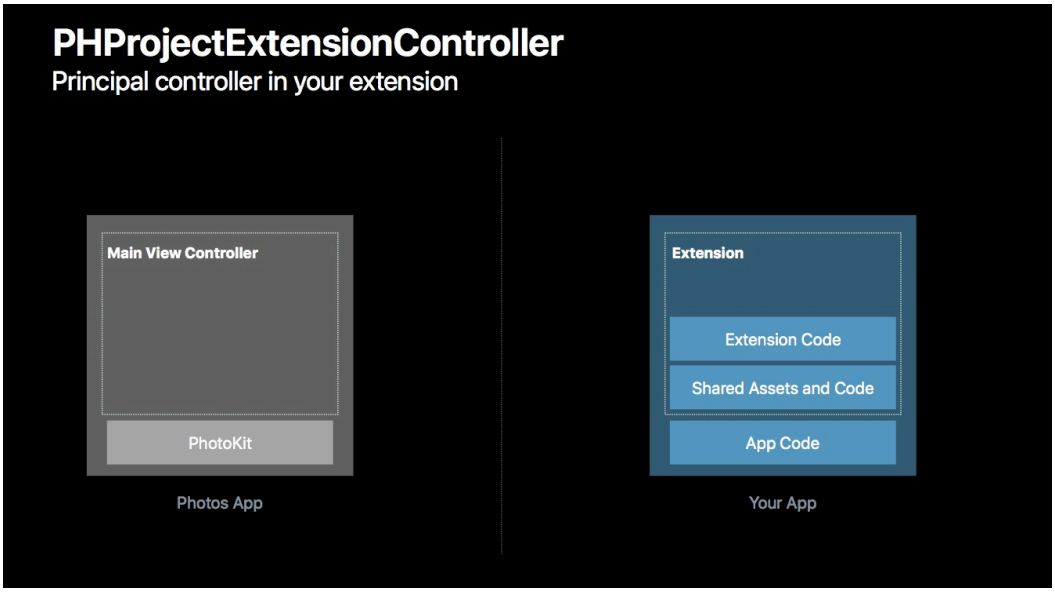
如上，通过使用`indexOfObject`来进行 Asset 索引的查询。而`indexOfObject`方法内部是通过比较“对象标识符”，以找到符合条件的 Asset，这样就不会有附加的“硬盘访问”和“数据库提取”。进而避免了第一种方式中，因为枚举出现的耗时操作。同样的，对`containtsObject`也是如此。

照片项目的拓展

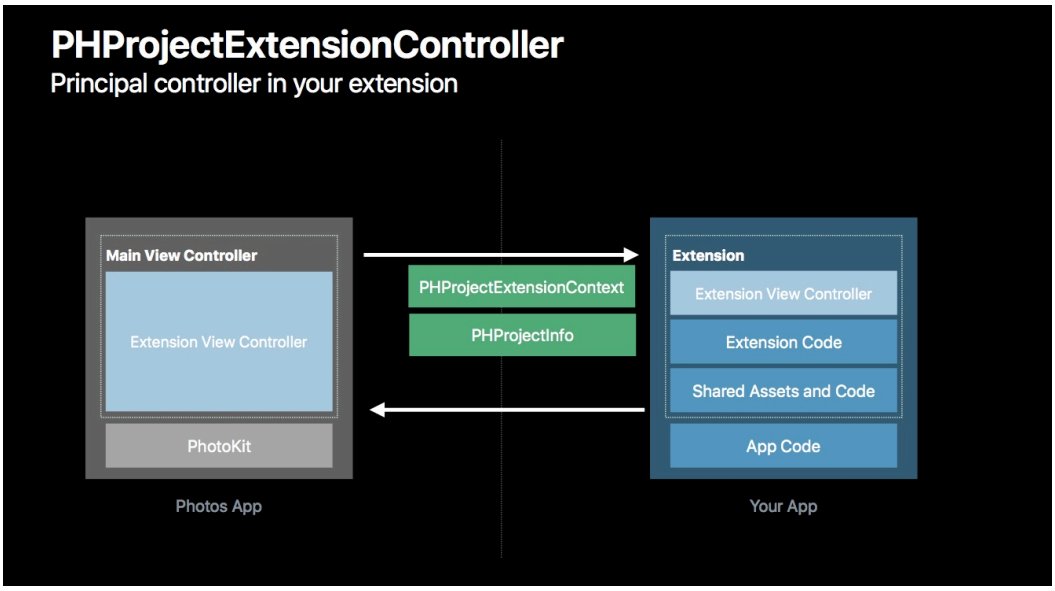
一直以来，Apple 允许用户围绕照片创建丰富的有创意的项目。

PHProjectExtensionController 的引入

现在，`“照片”`中添加了一个`新的扩展`。对应的，`Xcode`中也添加了一个`新模板`，开发者在自己的应用里可以`轻松创建`这些扩展。此外，`“照片”应用`会自动发现你的拓展，大大提高了扩展应用被用户知道的概率。不仅如此，Apple 为了让这些扩展更容易为用户所发现，给这些扩展应用提供了`App Store`的直接链接。该链接会打开`App Store`窗口，并自动显示支持该扩展的应用。



扩展只存在于开发者的应用内。对于开发者来说，如果你在 App Store 已经有了一个关于 Photos 相关的应用。此时，就可以将扩展代码移动到该扩展空间内，并加以利用。之后，添加一个视图控制器，并实现`PHProjectExtensionController`协议。一切就位，“照片应用”便可以发现你的扩展了。



在用户选择“扩展应用”，并用它创建一个项目时，`"照片应用"`会发送一些字节数据（`PHProjectExtensionContext`、`PHProjectInfo`）到对应的`“扩展应用”`。之后`“照片应用”`得到相应的返回结果，知道可以安装你的视图控制器。

过程中遵循的协议本身，对支持的`项目类型`有一个可选属性，可用于快速描述想让用户选择的选项。

```
optional public var supportedProjectTypes: [PHProjectTypeDescription] { get }
```

在实际使用过程中，用户既可以选择退出，也可以选择直接进入扩展。这些，在视图控制器里也有特定的函数方法。

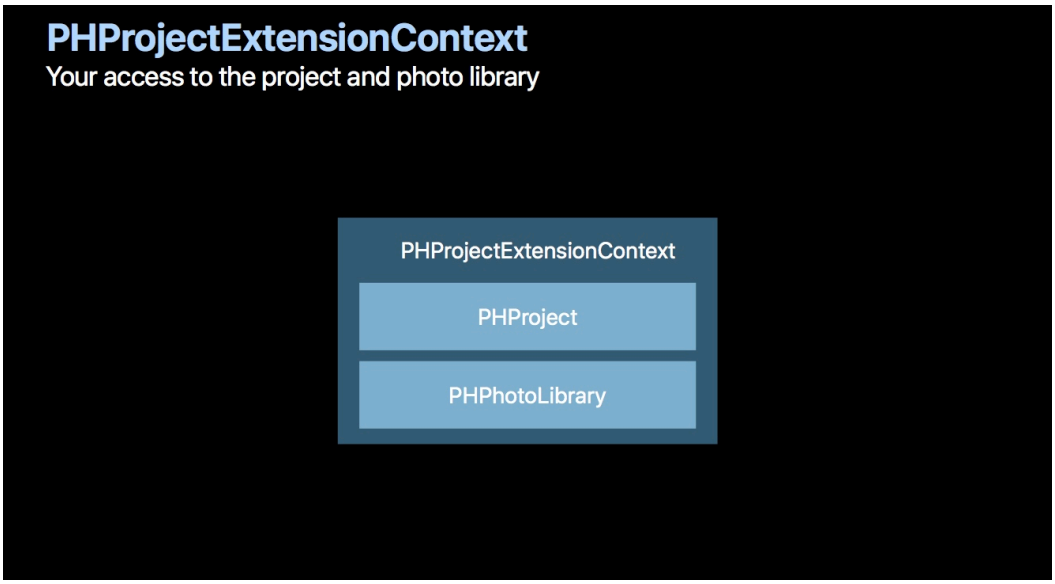
```
public protocol PHProjectExtensionController : NSObjectProtocol {
    //第一次使用该扩展创建项目时调用
  public func beginProject(with extensionContext: PHProjectExtensionContext,
        projectInfo: PHProjectInfo, completion: @escaping (Error?) -> Void)

    //每次用户回到以前创建的项目时调用
    public func resumeProject(with extensionContext:
PHProjectExtensionContext,
        completion: @escaping (Error?) -> Void)

    //用户离开项目时调用
public func finishProject(completionHandler completion: @escaping () -> Void)
}
```

通过第一个函数方法，我们可以得到上下文及项目详细。在第一次使用该扩展创建项目时，会调用该方法。

第二个函数方法，我们同样可以获得上下文。在每次用户回到扩展项目时，会调用该方法。

最后一个函数方法。如果用户在扩展项目内，当他们决定切换离开，会调用该函数。通过回调，你可以清理任何正在处理的数据，或是关闭任何让处理器忙碌的任务，或是正在执行的动画。

PHProjectExtensionContext 是什么



在`PHProjectExtensionContext`这个容器里，包含两个非常重要的对象。一个是`PHProject`，另一个是`PHPhotoLibrary`。

PHProject 介绍

```
// PHProject.h
// Photos
class PHProject : PHAssetCollection {    var projectExtensionData: Data { get
}
}
```

`PHProject`本身只是`PHAsset`的一个`子集`。在子集中，创建`PHProject`，只添加了一个非常重要的属性`projectExtensionData`。可以用于保存任何你需要的数据，它是你正在使用的`资产标识符`的列表。也许是一些基本的`布局信息`或`配置信息`。同时属性`projectExtensionData`并不是为了照片缓存、缩略图之类的存在。因为这些功能，你可以快速地创建或把它们缓存到其他位置。为了它小而有用，抛开了这些功能，并且`projectExtensionData`被限制定为`1 兆`。因为这里面的信息，是一系列的字符串，所以`1 兆`大小已经足够了。即使用户不断创建项目，也不会增大用户的库。

PHProjectChangeRequest

```
do {
    let changeRequest = PHProjectChangeRequest(project: self.project)
    try self.library.performChangesAndWait {
        changeRequest.projectExtensionData =
NSKeyedArchiver.archivedData(withRootObject: cloudIdentifiers)
    }} catch {
  print("Failed to save project data: \(error.localizedDescription)")
}
```

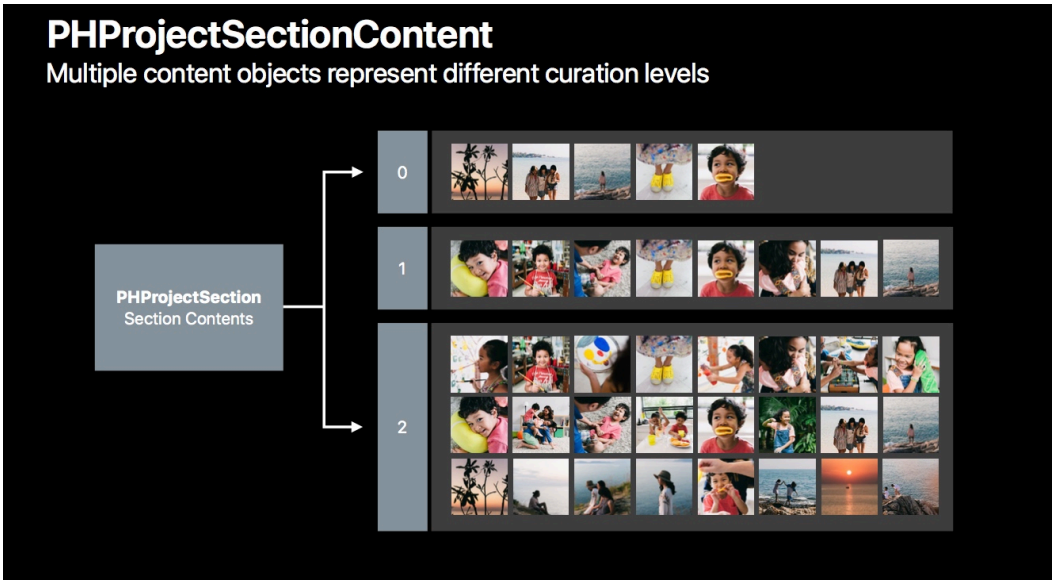
设置数据非常简单。只需按上述实例化，实例化后，可以在`Photo Library`调用`performChangesAndWait`函数，在里面将数据设置成任何你想要的样式。

PHProjectInfo 介绍

最高层的`ProductInfo`分为下面这几个区：

当看到这个构造时，可能会问，为什么数组里面还有数组。为什么是这种嵌套结构。但是如果想想“照片”应用里的“回忆”功能，会发现这些是有道理的。

“回忆”本身建立于大量资产之上，允许用户回忆时，可以切换`“显示照片摘要”`或`“显示所有照片”`。通过下面的一张图解，可以更清晰地描述为什么使用数组。



`Section Contents`数组是已排序数组。`索引为 0`的对象是最优内容，是资产集合最精炼的摘要；而`数组末端`的对象内容是最多的，包含了所有的照片数据。开发过程中，开发者需要根据具体的需求，选择性地使用。

PHCloudIdentifier 介绍

`PHCloudIdentifier`是一个全新的概念。当你想把数据存到用户的`Photo Library`时，数据可能会被同步到用户其他的设备中。为了确保在保存的数据，合理地同步到其他设备中，iOS 11 推出了一个新对象`PHCloudIdentifier`。

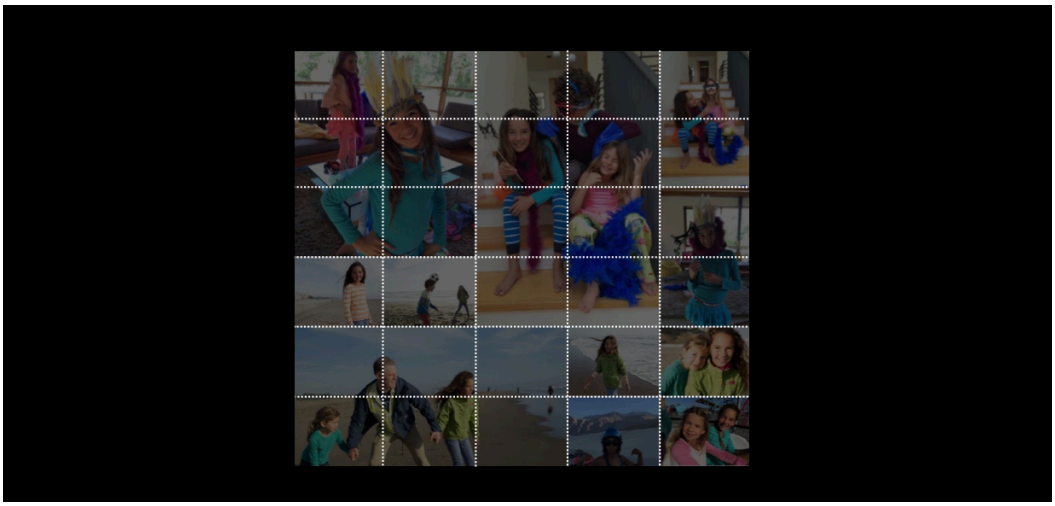
```
//获取当前 Cloud Identifiers
cloudIdentifiers += dataDict.value(forKey: "contentIdentifiers") as!
[PHCloudIdentifier]

//转换为 Local Identifiers
let localIdentifiers = self.library.localIdentifiers(for: cloudIdentifiers)
```

可以将它看做资产的`全局标识符`，但也并不像全局字符串那么简单，因为还需要处理同步和同步状态等情况。而这些复杂操作，系统已经为我们做了。你必须要做的唯一操作是，在提取之前，确保你的转换总是从`全局标识符`到`本地标识符`。可以通过`PHPhotoLibrary`里的方法来进行双向转换。

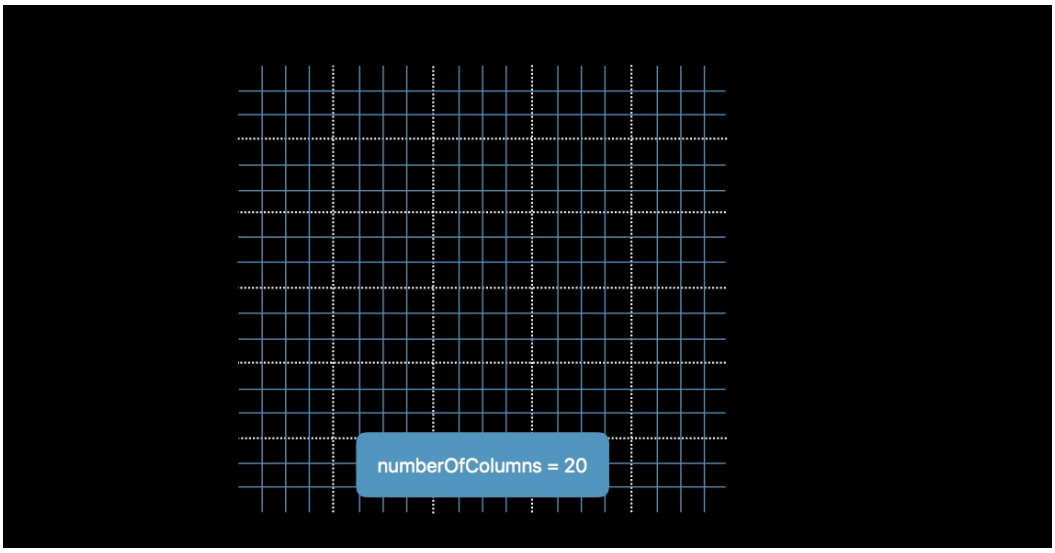
关于“视图布局”的改进

例如上述这种网格布局，对用户来说是很愉快的。如果开发者可以直接访问这个布局，不是会很棒吗？在 iOS 11 中，你确实可以进行访问了。

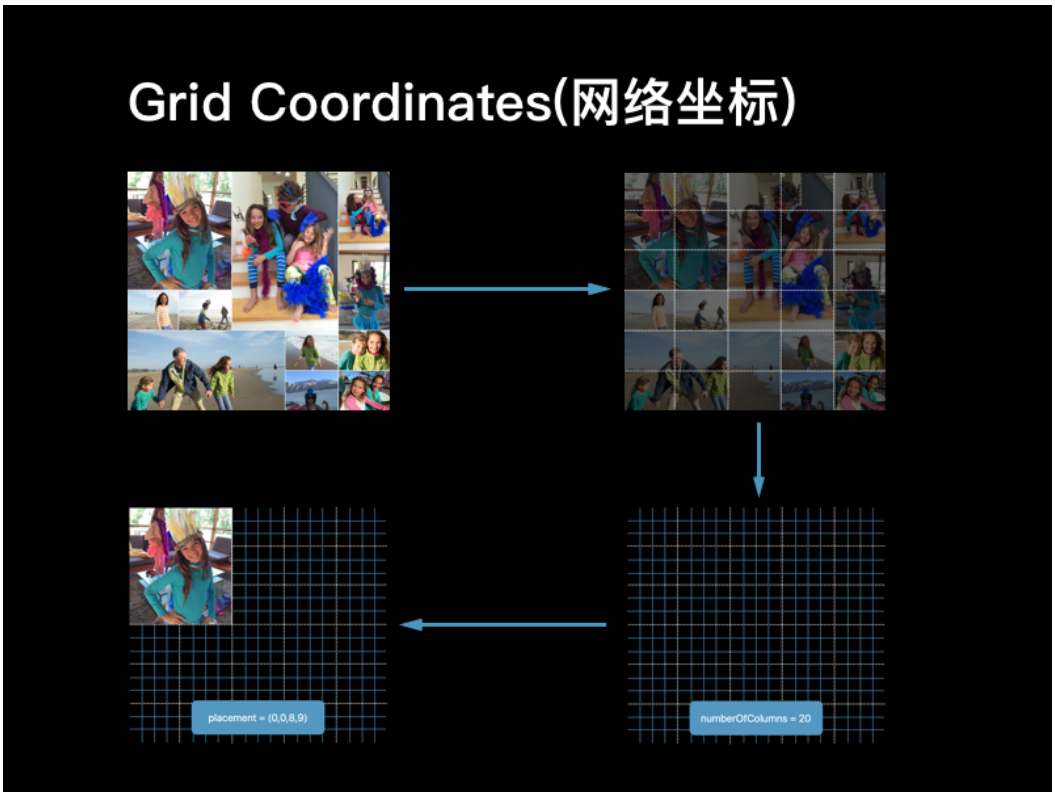
为了支持访问，系统首先确定了一个坐标系。



如果你查看“回忆”功能，会发现所有内容都被排列在一个由 4x3 单元格构成的网络中。但它是一个非正方形尺寸，不利于拓展。所以又有了下面这样的结构。



例如上述实例图片的的布局，可以被转化为一个由 20 个统一列组成的网络空间。确定了这个坐标系，就可以根据需求任意缩放。



并且，通过这个坐标系，也可以和系统互相传递坐标信息。例如上图的坐标为`(0, 0, 8 , 9)`。

PHProjectElement 介绍

```
// PHProjectElementclass
PHProjectElement : NSObject, NSSecureCoding {
        //权重的范围是 0.0 - 1.0, 默认为 0.5
var weight: Double { get }
        //元素在网络布局中的坐标
var placement: CGRect { get }
}
```

在`Section Content`里，系统提供了一组元素。所有元素都是`PHProjectElement`的子集。这里有两个非常重要的属性：

- `placement`（位置）

- `weight`（权重）

`“位置”属性`，前面已经有所介绍了，这里介绍下`“权重”属性`。再次回到“回忆”功能，在大量资产中如果想确定最相关的照片，系统需要有自己的评分系统。这里，评分系统通过给每个元素一个权重值，来代表每个元素的重要性。权重值从`0.0`到`1.0`，默认值是`0.5`，也就是说，权重值为`0.5`的资产代表普通。

RegionsOfInterest 是什么

```
var regionsOfInterest: [PHProjectRegionOfInterest] { get }
```

这里，有这样的一个概念，称为`“兴趣区”`。也就是这里的`regionsOfInterest`属性，这是`PHProjectAssetElement`所特有的。

在`macOS API`里，已经有很多方法，可以用来进行`面部识别`，从而寻找图片中的脸。但从这些方法无法知道这些脸的相关性。来看下面一副示例图：

你会注意到，这些脸部有对应的标识符。在不同的照片里相同的脸，会看到被标记为相同的标识符。这样的表示十分有趣。如果你正在处理动画、幻灯片等效果，这将非常有用。因为你现在可以真正把大集合中的图片位置彼此联系起来了。对于体验上的改进来说，这会是一个很棒的属性。

总结

本节主要介绍了 Photos APIs 的新特性，主要包含了以下几点内容：

- 改进的授权模式
- 大幅优化的 UIImagePickerController
- 全新的图片格式 HEIF
- 大型图片库的创建
- 及为 Photos 创建项目扩展

回头来看开篇提出的三点疑问：

- 如何以一种不违反用户信任的方式获取及保存内容到相册？
- 是否可以为 Photo Library 创建扩展内容？
- 如何在应用中简单、高效地实现这些操作？

这些，在新的 Photos APIs 里都有了相应的解决方案。综上可以看到 Photos 也在越来越完善，可扩展性越来越强，功能也在越来越强大。

相关资料

- WWDC 2017 Session 505 - What's New in Photos APIs
- WWDC 2017 Session 505 - Creating Large Photo Libraries for Testing

- 开发者文档 - Photos
- 开发者文档 - PhotosUI
- PhotoKit 示例应用