Table of Co

通过实现一个TableView来理解IOS UI编程

欢迎关注IOSTips微信订阅号：

先说点题外话。我们在日常做和IOS的UI相关的工作的时候，有一个组件的使用频率非常高--UITabelView。于是就要求我们对UITableView的每一个函数接口，每一个属性都了如指掌，只有这样在使用UITableView的时候，我们才能游刃有余的处理各种需求。不然做出来的东西，很多时候只是功能实现了，但是程序效率和代码可维护性都比较差。举个例子，比如在tableView头部要显示一段文字。我见过的最啰嗦的解决方案是这样的：

1. 子类化一个UIViewController
2. 将根View设置成一个UIScrollView
3. 把头部的Label和TableView加在ScrollView上面
4. 开始各种调整ScrollView和TableView的delegate调用函数里面的参数，让Label能随着TableView滑动

其实如果你熟悉UITableView，那么你几句话就可以搞定

```
UILabel* label = [UILabel alloc] initWithFrame:CGRectMake(0, 0, CGRectGetWidth(self.view.frame), 40);
label.text = @"就是一段文字嘛，干嘛大动干戈";
tableVIew.tableHeaderView = label;
```

所谓工欲善其事必先利器，编程语言和各种库其实本质上就是工具而已。你要想用这些工具来实现产品和Leader提出的各种需求。当然，不止是功能上的实现，还包括程序效率，代码质量。特别想着重强调一下代码质量，如果你不想后面维护自己的代码就像噩梦一样，如果你不想一旦新来一个需求就得对代码大刀阔斧的伤筋动骨，如果你不想给后来者埋坑。那么最好就多注意一下。

这里的代码质量并不是简简单单的指代码写点注释了，利用Xcode提供的一些像pragam或者#warning来解释代码。《编写可阅读代码的艺术》还有其他一些编程的书籍也都说道，真正高质量的代码，是不需要注释的。一个好的代码从逻辑上和结构上都是清晰的。我看到很多很难维护的代码都是因为逻辑结构混乱，和设计模式滥用导致的程序结构紊乱。分析其原因，就会发现很多时候，是因为写代码的人对所使用的工具（主要是objc和UIKit）不是非常熟悉，于是就写了很多凑出来的临时方案，简单的实现了功能。表面看起来挺好的，但是实际上代码已经外强中，骨子里都乱了。后期维护起来会让人痛不欲

生。

了解一下UITableView的一些详细的技术细节甚至是UIKit的一些技术细节对于我们写出比较好的代码，比较好的实现任务是很有必要的。我们通过实现自己的TableView来反观UIKit的UITableView，来加深我们对UITableView和UIKit的理解。在这个过程中，我们会碰到非常多非常细节的问题，而这些正是我们需要注意并且掌握的。

同时，个人一直觉得对于搞IOS开发来说自己实现一遍TableView就像是一种成人礼一样。你能够通过实现一个UITableView来深入的理解UIKit的一些技术细节，对IOS UI编程所使用到的工具，有比较深入的了解。这样，写程序的时候才不会捉襟见肘。

言归正传。开始实现一个TableView。

UIKit提供的基础

又重复了一遍，工欲善其事必先利其器。那么我们就看一下UIKit为我们提供了那些好用的工具让我们来实现一个TableView（当然不是子类化一个UITableView这么简单）。

这里会牵扯到一个另外一个问题，可能有些读者会问可不可以从最最底层的开始做起，来实现一个TableView呢，比如从写一个图形界面库开始。这个从技术上来说，完全可以实现，但是仔细想想在Apple为我们提供了UIKit之后，如果我们不是写游戏的话，貌似完全没有必要重新造这个轮子啊。当然你要写游戏的话，那令当别论，请出门左转有开源的Cocoa2d，作者要是针对从最底层开始构建感兴趣可以看一下Cocoa2d的开源代码，想必肯定大有收获。

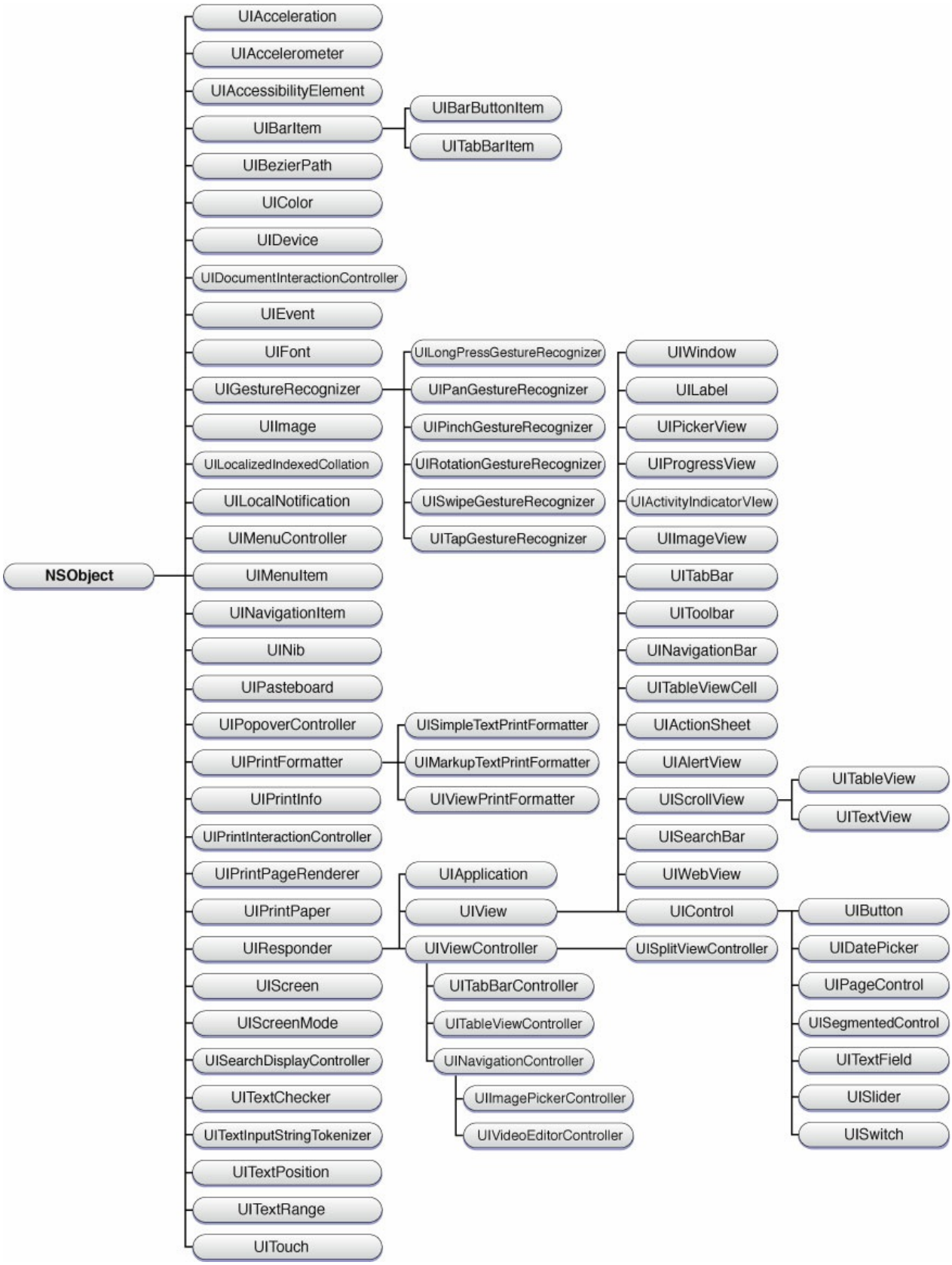
对于实现一个主要在应用中使用的TableView来说，就没有必要重新造这个轮子。从研究UIKit为我们提供的一些对象和功能开始，就可以构建起来TableView，这也是我们文章一开始的目的——通过实现TableView来深入理解UIKit编程。

在研究的过程中，我们也会探讨一些UIKit在设计这个图形界面库的时候一些设计动机。但是，这里必须提醒读者的是，很多探讨只是我和一些朋友探讨的结果，虽然在我们看来是对的。但是可能苹果当初并不是这么想的，我还是写出来，也是做了一些思想斗争的，毕竟谁都想只把大家都认为是正确的东西提供给大家。我这样做是为了给读者提供另一种视角来理解UIKit这个非常牛的库。其中不对的地方，当然非常欢迎各位指正。

UIKit概述

UIKitk框架提供一系列的Class(类)来建立和管理IOS应用程序的用户界面(UI)接口、应用程序对象、事件控制、绘图模型、窗口、视图和用于控制触摸屏等的接口。可以理解成是苹果提供给开发和来操纵程序和界面的一个API库。

其类图如下所示：



UIApplication

UIApplication类提供了对运行在IOS设备上的app集中控制和调度的机制。每一个IOS app必须有一个而且只能有一个UIApplication或者其子类的实例。当程序启动的时候，会调用`UIApplicationMain`函数，在这个函数中会创建一个UIapplication类的单例，这个单例在整个IOS系统中就是你的App的抽象。之后你就能够通过`shareApplication`方法来调用该单例。

UIApplication对象的主要工作是处理用户事件的路由。它也会给UIcontrol对象分发动作消息。另外，UIApplication还维护了当前App打开的窗口的列表。所以，你通过它能够取到你App中任何一个View。

这个app实例还实现了一个delegate，接受各种各样程序运行时的事件，比如：程序启动、低内存警告、程序崩溃等等。

程序还能通过`openURL:`方法来接受和处理一个邮件或者图片文件。比如一个以Email开头的URL将能够唤起Email程序来展示这个邮件。

UIApplication的编程接口让你能够管理一些硬件指定的行为。比如：

- 控制App来响应设备方向变化
- 暂时终止接受触摸事件
- 打开或者关闭接近用户脸部的感应
- 注册远程消息通知
- 打开或者关闭undo-redo UI
- 决定你的程序是否能够支持某一类的URL
- 扩展程序能力，让app能够在后台运行
- 发布或者取消本地通知
- 接受远程控制事件
- 执行程序级别的复位操作

UIApplication必须实现UIApplicationDelegate协议来实现他的一些协议。

Windows&Views（窗口和视图）

窗口和视图是为iPhone应用程序构造用户界面的可视组件。窗口为内容显示提供背景平台，而视图负责绝大部分的内容描画，并负责响应用户的交互。虽然本章讨论的概念和窗口及视图都相关联，但是讨论过程更加关注视图，因为视图对系统更为重要。

视图对iPhone应用程序是如此的重要，以至于在一个章节中讨论视图的所有方面是不可能的。本章将关注窗口和视图的基本属性、各个属性之间的关系、以及在应用程序中如何创建和操作这些属性。本章不讨论视图如何响应触摸事件或如何描画定制内容，有关那些主题的更多信息，请分别参见“事件处理”和“图形和描画”部分。

什么是窗口和视图？

和Mac OS X一样，iPhone OS通过窗口和视图在屏幕上展现图形内容。虽然窗口和视图对象之间在两个平台上有很多相似性，但是具体到每个平台上，它们的作用都有轻微的差别。

1-1-2视图和窗口架构

视图和窗口架构 视图和窗口展示了应用的用户界面，同时负责界面的交互。UIKit和其他系统框架提供了很多视图，你可以就地使用而几乎不需要修改。当你需要展示的内容与标准视图允许的有很大的差别时，你也可以定义自己的视图。

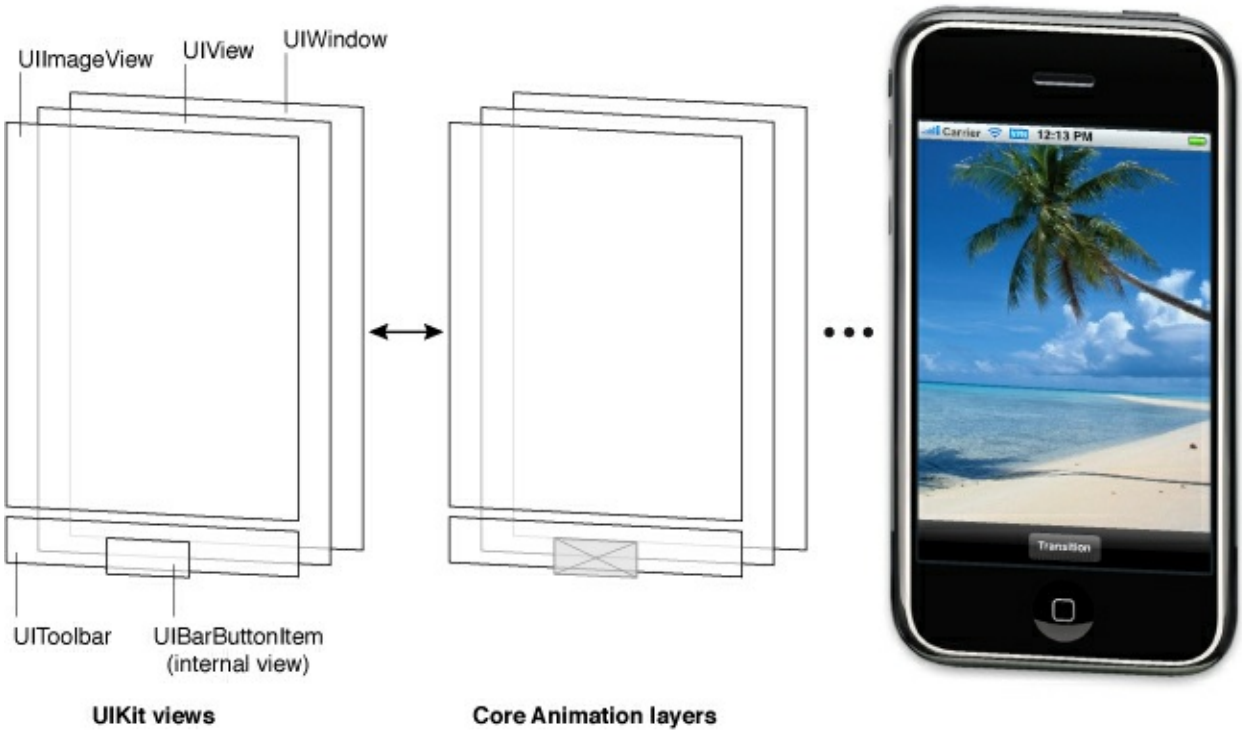
不管你是使用系统的视图还是创建自己的视图，你需要理解UIView和UIWindow类所提供的基本结构。这些类提供了复杂的方法来管理视图的布局和展示。理解这些方法的工作非常重要，使你在应用发生改变时可以确认视图有合适的行为。

视图架构

大部分你想要可视化操作都是由视图对象－即UIView类的实例－来进行的。一个视图对象定义了一个屏幕上的一个矩形区域，同时处理该区域的绘制和触屏事件。一个视图也可以作为其他视图的父视图，同时决定着这些子视图的位置和大小。UIView类做了大量的工作去管理这些内部视图的关系，但是需要的时候你也可以定制默认的行为。

视图与Core Animation层联合起来处理着视图内容的解释和动画过渡。每个UIKit框架里的视图都被一个层对象支持（通常是一个CALayer类的实例），它管理管理着后台的视图存储和处理视图相关的动画。然而，当你需要对视图的解释和动画行为有更多的控制权时，你可以使用层。

为了理解视图和层之间的关系，我们可以借助于一些例子。图1－1显示了ViewTransitions样例程序的视图层次及其对底层Core Animation层的关系。应用中的视图包括了一个window(同时也是一个视图)，一个通用的表现得像一个容器视图的UIView对象，一个图像视图，一个控制显示用的工具条，和一个工具条按钮(它本身不是一个视图但是在内部管理着一个视图)。（注意这个应用包含了一个额外的图像视图，它是用来实现动画的）。为了简化，同时因为这个视图通常是被隐藏的，所以没把它包含在下面的图中。每个视图都有一个相应的层对象，它可以通过视图r属性被访问。（因为工具条按钮不是一个视图，你不能直接访问它的层对象。）在它们的层对象之后是Core Animation的解释对象，最后是用来管理屏幕上的位的硬件缓存。



使用Core Animation的层对象有很重要的性能意义。一个视图对象的绘制代码需要尽量的少被调用，当它被调用时，其绘制结果会被Core Animation缓存起来并在往后可以被尽可能的重用。重用已经解释过的内容消除了通常需要更新视图的开销昂贵的绘制周期。内容的重用在动画中特别重要，我们可以使用已有的内容，这样比创建新的内容开销更小。

视图层次和子视图管理

除了提供自己的内容之外，一个视图也可以表现得像一个容器。当一个视图包含其他视图时，就在两个视图之间创建了一个父子关系。在这个关系中孩子视图被当作子视图，父视图被当作超视图。创建这样一个关系对应用的可视化和行为都有重要的意义。

在视觉上，子视图隐藏了父视图的内容。如果子视图是完全不透明的，那么子视图所占据的区域就完全的隐藏了父视图的相应区域。如果子视图是部分透明的，那么两个视图在显示在屏幕上之前就混合在一起了。每个父视图都用一个有序的数组存储着它的子视图，存储的顺序会影响到每个子视图的显示效果。如果两个兄弟子视图重叠在一起，后来被加入的那个（或者说是排在子视图数组后面的那个）出现在另一个上面。

父子视图关系也影响着一些视图行为。改变父视图的尺寸会连带着改变子视图的尺寸和位置。在这种情况下，你可以通过合适的配置视图来重定义子视图的尺寸。其他会影响到子视图的改变包括隐藏父视图，改变父视图的alpha值，或者转换父视图。

视图层次的安排也会决定着应用如何去响应事件。在一个具体的视图内部发生的触摸事件通常会被直接发送到该视图去处理。然而，如果该视图没有处理，它会将该事件传递给它的父视图，在响应者链中以此类推。具体视图可能也会传递事件给一个干预响应者对象，像视图控制器。如果没有对象处理这个事件，它最终会到达应用对象，此时通常就被丢弃了。

视图绘制周期

UIView类使用一个点播绘制模型来展示内容。当一个视图第一次出现在屏幕前，系统会要求它绘制自己的内容。在该流程中，系统会创建一个快照，这个快照是出现在屏幕中的视图内容的可见部分。如果你从来没有改变视图的内容，这个视图的绘制代码可能永远不会再被调用。这个快照图像在大部分涉及到视图的操作中被重用。

如果你确实改变了视图内容，也不会直接的重新绘制视图内容。相反，使用setNeedsDisplay或者setNeedsDisplayInRect:方法废止该视图，同时让系统在稍候重画内容。系统等待当前运行循环结束，然后开始绘制操作。这个延迟给了你一个机会来废止多个视图，从你的层次中增加或者删除视图，隐藏，重设大小和重定位视图。所有你做的改变会稍候在同一时间反应。

注意：改变一个视图的几何结构不会自动引起系统重画内容。视图的contentMode属性决定了改变几何结构应该如果解释。大部分内容模式在视图的边界内拉伸或者重定位了已有快照，它不会重新创建一个新的快照。获取更多关于内容模式如果影响视图的绘制周期，查看 content modes

当绘制视图内容的时候到了时，真正的绘制流程会根据视图及其配置改变。系统视图通常会实现私有的绘制方法来解释它们的视图，（那些相同的系统视图经常开发接口，好让你可以用来配置视图的真正表现。）对于定制的UIView子类，你通常可以覆盖drawRect:方法并使用该方法来绘制你的视图内容。也有其他方法来提供视图内容，像直接在底部的层设置内容，但是覆盖drawRect:时最通用的技术。

内容模式

视图的内容模式控制着视图如何回收内容来响应视图几何结构的变化，也控制着是否需要回收内容。当一个视图第一次显示时，它通常会解释内容，其结果会被底层的层级树捕获为一张位图。在那之后，改变视图的几何结构不会导致重新创建位图。相反，视图中contentMode属性的值决定着这张位图是否该被拉伸，以适应新的边界或者只是简单的被放到角落或者视图的边界。

视图的内容模式在你进行如下操作时被应用：

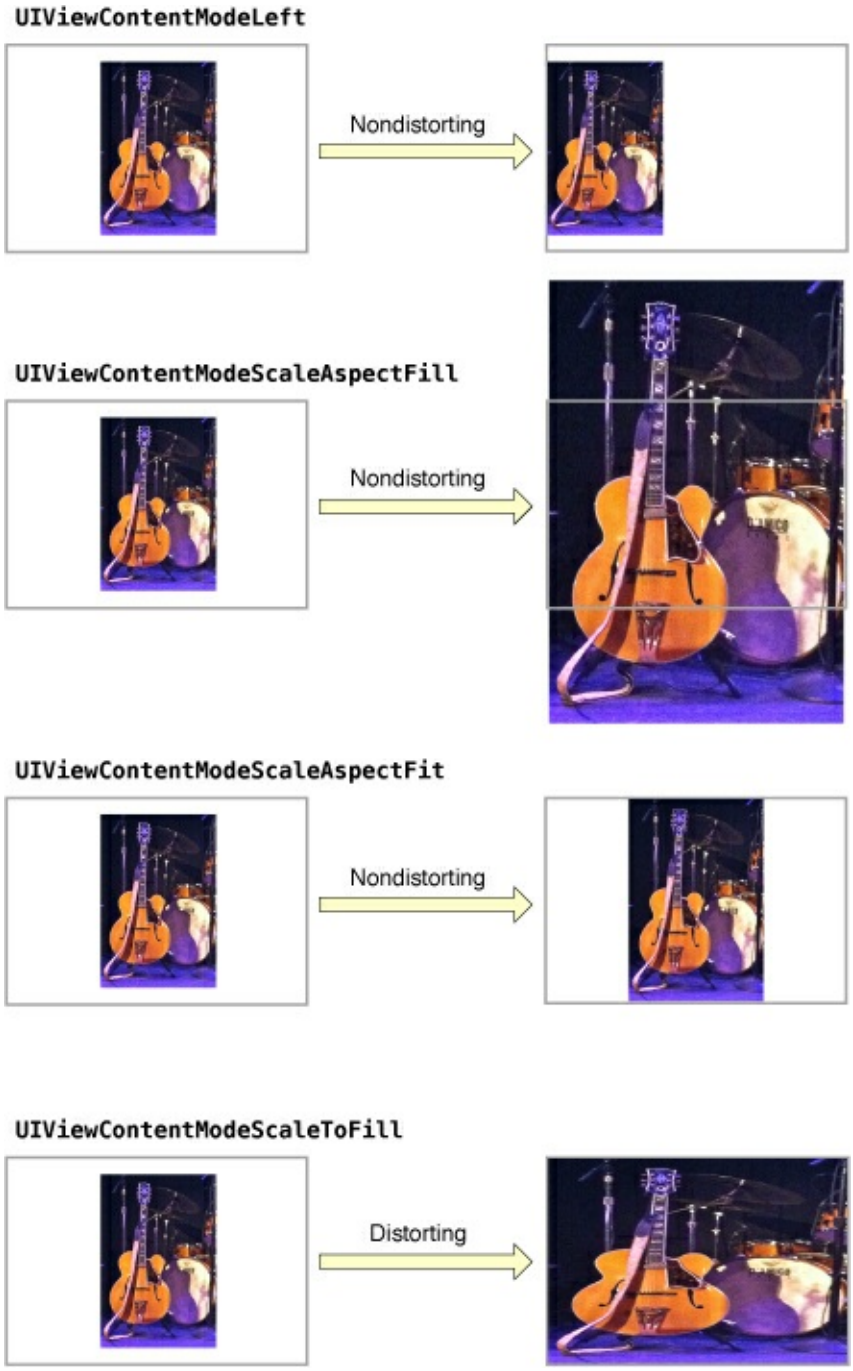
改变视图frame或者bounds矩形的宽度或者高度时。

赋值给视图的transform属性，新的转换包括一个放缩因子。

大部分视图的contentMode值是UIViewContentModeScaleToFiill，它使视图的内容被放缩到适合新框架的值。Figure 1-2展示了使用其他可用的内容模式的结果。正如你在图中所看到的那样，不是所有的内容模式都可以填充视图的范围，可以的模式可能会扭曲内容。

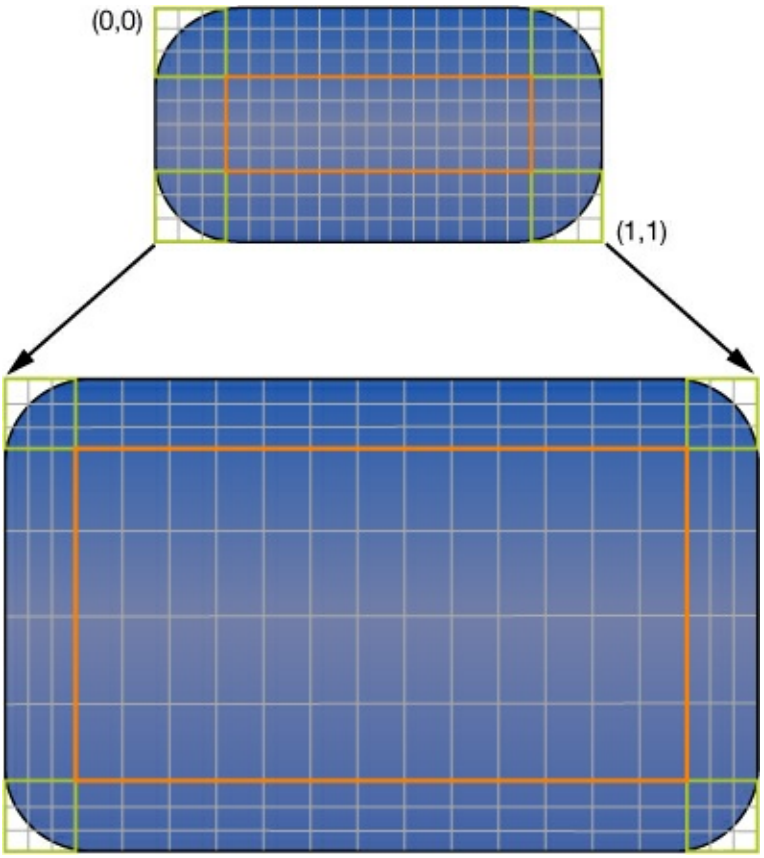
内容模式很好的支持了视图的内容回收，但是当你想视图在放缩和重设尺寸的操作中重绘你也可以用UIViewContentModeRedraw内容模式。设置这个值绘强制系统调用视图的drawRect:方法来响应几何结构的变化。通常来讲，你应该尽可能的避免使用这个模式，同时你不应该在标准的系统视图中使用这个模式。

获取更多骨干与可用的内容模式，查看UIView Class Reference



拉伸视图

你可以指定视图的某部分为可拉伸的，以便当视图的尺寸改变时只有可拉伸的部分被影响到。可拉伸的部分通常给按钮或者其他的部分为重复模式的视图。由你指定的可拉伸区域允许沿着两条或者其中一条轴拉伸。当然，当一个视图沿着两条轴拉伸的时候，视图的边界必须也定义了一个重复的模式来避免任何的扭曲。Figure1-3展示了这种扭曲在视图里是怎么表现自己的。每个视图里的原始像素的颜色都自我复制，以便可以填充更大视图的相应区域。



你可以用contentStretch属性来定义一个视图的可拉伸区域。这个属性的值一个边的值被标准化为0.0到1.0之间的矩形。当拉伸这个视图时，系统将视图的当前边界值和放缩因子乘以标准值，以便决定哪些像素需要被拉伸。使用标准值可以减轻每次改变视图的边界值都更新contentStretch属性的需要。

视图的内容模式也在决定如何视图的可拉伸区域的使用中扮演着重要的角色。只有当内容模式可能绘引起视图内容放缩的时候可拉伸区域才会被使用。这意味这你的可拉伸视图只被UIViewContentModeScaleToFill, UIViewContentModeScaleAspectFit和UIViewContentModeScaleAspectFill内容模式。如果你指定了一个将内容弹到边界或者角落的内容模式（这样就没有真正的放缩内容），这个视图会忽视可拉伸区域。

注意：当需要创建一个可拉伸UIImage对象作为视图的背景时，使用contentStretch属性是推荐的。可拉伸视图完全被Core Animation层处理，这样性能通常更好。

嵌入式动画支持

使用层对象来支持视图的其中一个利益是你可以轻松的用动画处理视图相关的改变。动画是与用户进行信息交流的一个有用的方法，而且应该总是在进行应用设计的过程中考虑使用动画。UIView类的很多属性是动画化的－也就是，可以半自动的从一个值动画的变化到另一个值。为了实现这样一个动画，你需要做的只是：1 告诉UIKit你想要实现一个动画 2 改变这个属性的值 在一个UIView对象中有以下的动画化属性： frame - 你可以使用这个来动画的改变视图的尺寸和位置 bounds - 使用这个可以动画的改变视图的尺寸 center － 使用这个可以动画的改变视图的位置 transform － 使用这个可以翻转或者放缩视图 alpha － 使用这个可以改变视图的透明度 backgroundColor － 使用这个可以改变视图的背景颜色 contentStretch - 使用这个可以改变视图内容如何拉伸

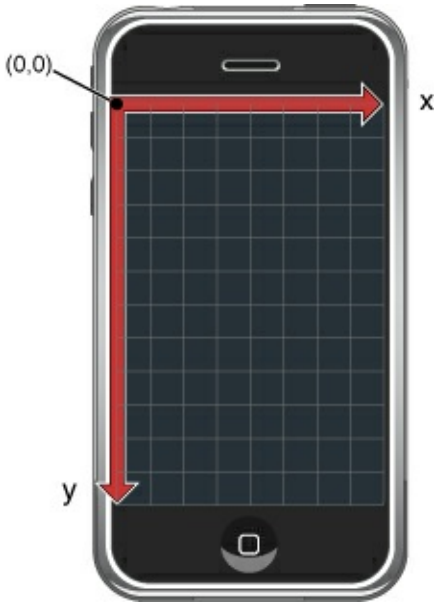
动画的一个很重要的地方是用于从一组视图到另一组视图的过渡。通常来说，会用一个视图控制器来管理关系到用户界面的主要变更的动画。例如，涉及到从高层到底层信息的导航的界面，通常会使用一个导航控制器来管理视图的过渡，这些视图显示了数据的每一个连续层面。然而，你也可以使用动画来创建两组视图的过渡，而不是视图控制器。当你想用一个系统提供的视图控制器无法支持的导航方案时你可能会这样做。

除了用UIKit类可以创建动画外，你也可以用Core Animation层来创建动画。在更低层你有更多的在时间或者动画属性上的控制权。

获取更多关于如何创建一个基于视图的动画，查看 Animations 获取更多关于使用Core Animation创建动画的信息，查看 Core Animation Programming Guide和Core Animation Cookbook.

视图几何结构和坐标系统

UIKit的默认坐标系统把原点设置在左上角，两条轴往下和右扩展。做标志被表示为浮点数，这样允许内容的精确布局和定位而不管底层的屏幕。Figure1-4展示了相对于屏幕的坐标系统。除了屏幕坐标系统窗口和视图也定义了它们自己的本地坐标系统，这样允许你指定相对于视图或者窗口原点的坐标而不是屏幕。



因为每个视图和窗口都定义了它自己的本地坐标系统，你需要留意在任何时间内是哪个坐标系统在起作用。每次绘制或者改变一个视图都是基于一个坐标系统的。在某些绘制中会基于视图本身的坐标系统。在某些几何结构变更中是基于父视图的坐标系统的。UIWindow和UIView类都包含了帮助你从一个坐标系统转换到另一个的方法。

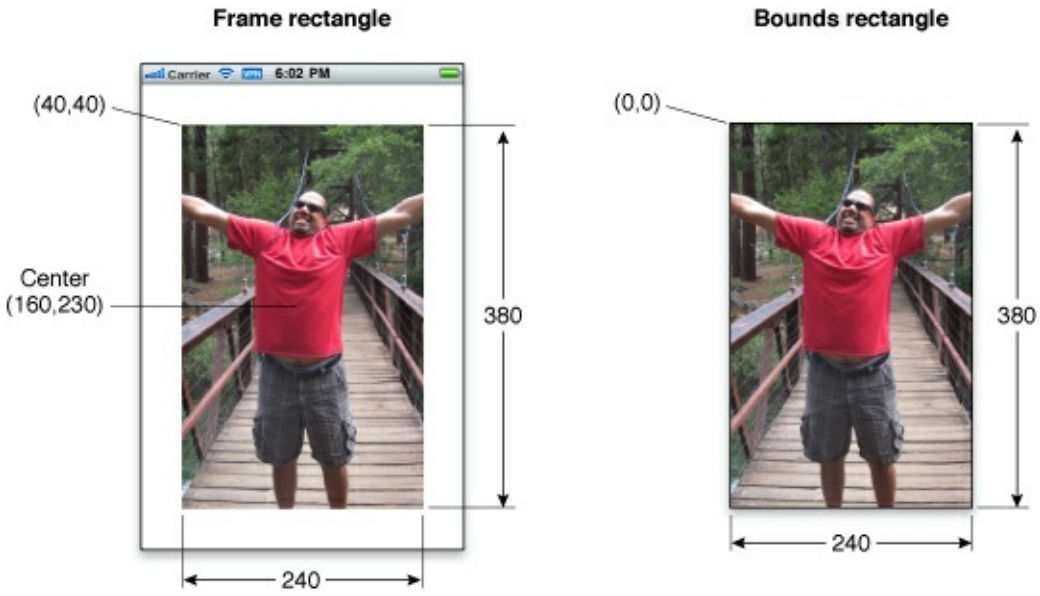
重要：一些iOS技术定义了默认的坐标系统，它们的原点和方向与UIKit的不同。；例如，Core Graphics和OpenGL ES的坐标系统是原点在可视区域的左下角，而y轴往上递增。当绘制或者创建内容时，你的代码应该考虑到一些不同并且适应坐标值。

frame, bounds和center属性之间的关系

视图对象使用frame, bounds和center属性来跟踪它的尺寸和位置： frame属性包含了frame矩形，指定了在父视图坐标系统中该视图的尺寸和位置。 center属性包含了在父视图坐标系统中的已知中心点。 bounds属性包含了边界矩形，指定了在视图本地坐标系统中视图的尺寸。 主要使用center和frame属性来控制当前视图的几何结构。例如，当在运行时构建你的视图层次或者改变视图的尺寸或者位置时你可以使用这些属性。如果你只是要改变视图的位置，那么推荐使用center属性。center属性的值永远是可用的，即使添加了放缩或者转换因子到视图的转换矩阵当中。但是对于frame属性却不是，当视图的转换矩形不等于原始矩阵时它被当作时无效的。

在绘制的过程中主要使用bounds属性。这个边界矩阵在视图的本地坐标系统被解释。这个矩形的默认原点是（0， 0），它的尺寸也适应frame矩形的尺寸。任何绘制在这个矩形当中的东西都是该视图的可视内容的一部分。如果你改变了bounds矩形的原点，任何你绘制在新矩形的东西都会变成该视图可视内容的一部分。

Figure1-5展示了一个图像视图的frame和bounds矩形之间的关系。图中，图像视图的右上角被定位在父视图坐标系统的（40， 40），它的矩形尺寸为240x380。对于bounds矩形，原点是（0， 0），矩形尺寸也是240x380。



即使你可以独立的改变frame，bounds和center属性，其中一个改变还是会影响到另外两个属性： 当你设置了frame属性，bounds属性的尺寸值也改变来适应frame矩形的新尺寸。center属性也会改变为新frame矩形的中心值。 当你设置了center属性，frame的原点也会相应的改变。 当你设置了bounds属性，frame属性会改变以适应bounds矩形的新尺寸。 视图的框架默认不会被它的父视图框架裁剪。这样的化，任何放置在父视图外的子视图都会被完整的解释。你可以改变这种行为，改变父视图的clipsToBounds属性就可以。不管子视图是否在视觉上被裁剪，触屏事件总是发生在目标视图父视图的bounds矩形。换句话说，如果触摸位于父视图外的那部分视图，那么该事件不会被发送到该视图。

坐标系统转换矩阵

坐标系统转换矩阵给改变视图（或者是它的视图）提供了一个轻松和简易的方法。一个仿射转换是一个数学矩阵，它指定了在坐标系统中的点是怎么被映射到另一个坐标系统中的点。你可以对整个视图应用仿射转换，以基于其父视图来改变视图的尺寸，位置或者朝向。你也可以在你的绘制代码中应用仿射转换，以对已解释内容的独立部分实现相同的操控。如何应用仿射转换是基于这样的上下文的： 为了修改整个视图，可以修改视图transform属性的仿射转换值。

为了在视图中的drawRect:方法中修改内容的指定部分，可以修改与当前图形上下文相关的仿射转换。

当你想实现动画时，通常可以修改视图的transform属性值。例如，你可以使用这个属性来制作一个视图围绕中心点翻转的动画。你不应该在其父视图的坐标空间中用这个属性来永久的改变你的视图，像修改它的位置和尺寸。对于这种类型的改变，你可以修改视图的frame矩形。

注意：当修改视图的transform属性值时，所有的转换都是基于视图的中心点来实现的。

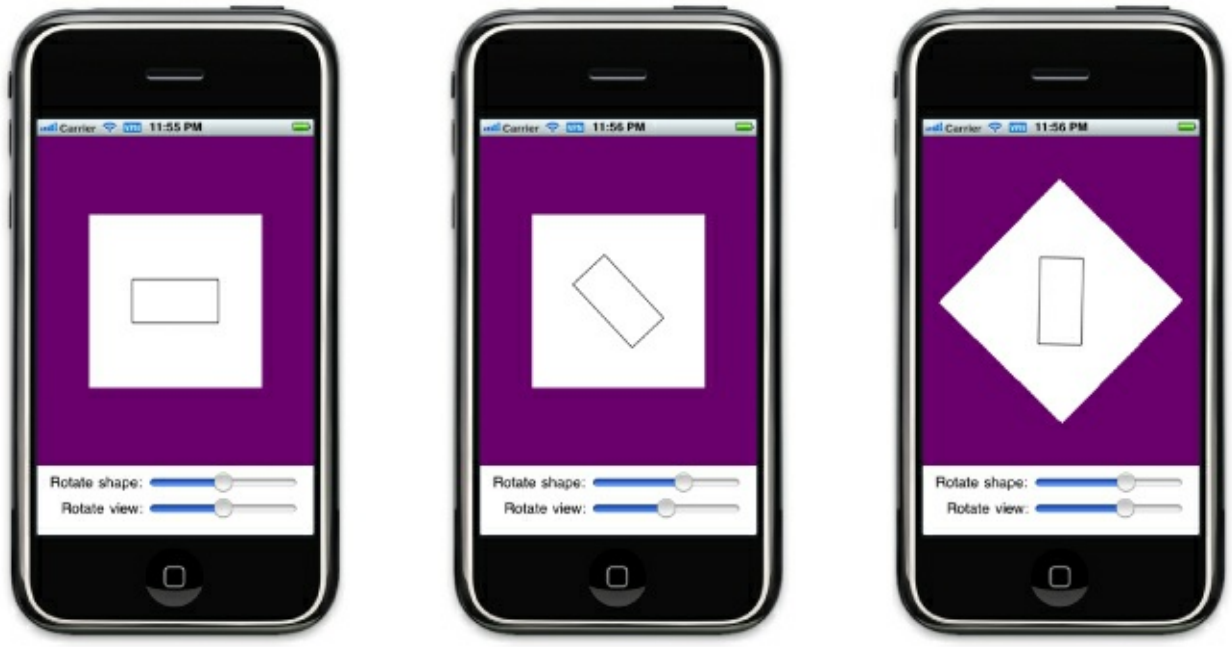
在视图的drawRect:方法中，你可以使用仿射转换来定位或者翻转你想要绘制的项目。相对于在视图某些部位中修正对象的位置，我们更倾向于相对于一个固定点去创建对象，通常是（0， 0），同时在绘制之前使用转换来定位对象。这样的话，如果在视图中对象的位置改变了，你要做的只是修改转换矩阵，这样比为对象重新创建新的位置性能更好开销更低。你可以通过使用CGContextGetCTM方法来获取关于图形上下文的仿射转换，同时可以用Core Graphics的相关方法在绘制中来设置或者修改这个转换矩阵。

当前转换矩阵（CTM）是一个在任何时候都被使用的仿射矩阵。当操控整个视图的几何结构时，CTM就是视图transform属性的值。在drawRect:方法中，CTM是关于图形上下文的仿射矩阵。

每个子视图的坐标系统都是构建在其祖先的坐标系统之上的。所以当你修改一个视图的transform属性，这个改变会影响到视图及其所有的子视图。然而，这些改变只会影响到屏幕上视图的最终解释。因为每个视图都负责绘制自己的内容和对自己的子视图进行布局，所以在绘制和布局的过程中它可以忽略父视图的转换。

Figure1- 6描述了在解释的时候，两个不同的转换因子是如何在视觉上组合起来的。在视图的drawRect:方法中，对一个形状应用一个45度的转换因子会使该形状翻转指定的角度。另外加上一个45度的转换因子会导致整个形状翻转90度。这个形状对于绘制它的视图来讲仍然只是翻转了45度，但是视图自己的转换让它看起来像使翻转了90度。

Figure 1-6 翻转一个视图和它的内容



No rotations | **Shape rotated 45° during drawing** | **Shape and view each rotated 45°**

重要：如果一个视图的transform属性不是其定义时转换矩阵，那么视图的frame属性是未定义的而且必须被忽略。当对视图应用转换时，你必须使用视图的bounds和center属性来获取视图的位置和尺寸。子视图的frame矩形仍然是有效的，因为它们与视图的bounds相关。

获取更多关于在运行时修改视图的transform属性，查看 “Translating, Scaling, and Rotating Views.”获取更多如何在绘制过程中使用转换来定位内容，查看 Drawing and Printing Guide for iOS.

点与像素

在iOS中，所有的坐标值和距离都被指定为使用浮点数，其单元值称为点。点的数量随着设备的不同而不同，而且彼此不相关。要明白关于点的最主要一点是它们提供了一个绘制用的固定框架。

Table 1-1 列出了不同iOS设备的分辨率（点度量）。前为宽后为长。只要你依照这些屏幕的尺寸来设计用户界面，你的视图就回被相应的设备正确显示。

Table 1-1

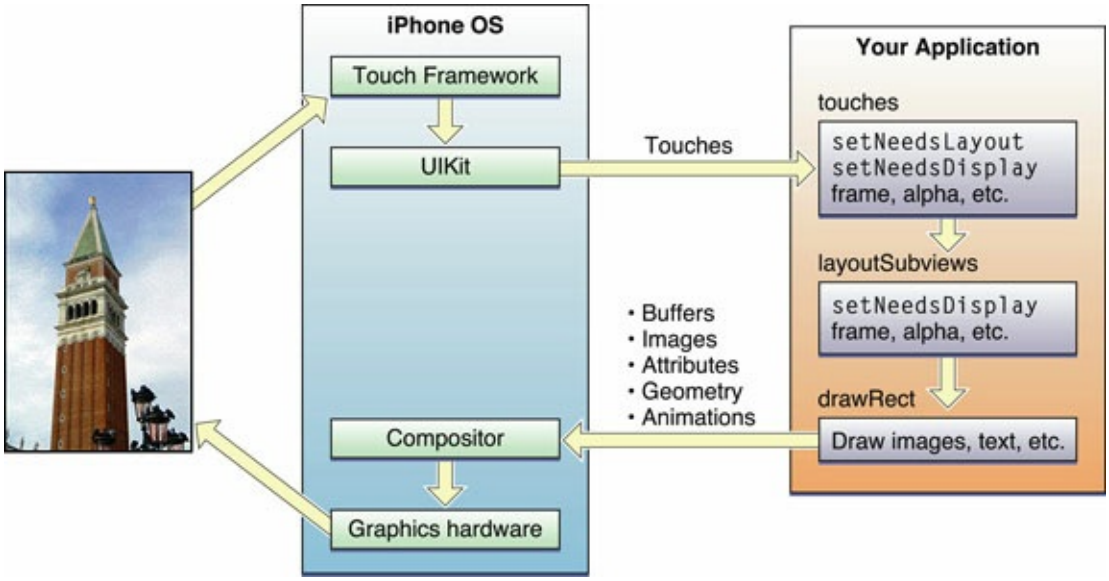
Device	Screen dimensions (in points)
iPhone and iPod touch	320 x 480
iPad	768 x 1024

每一种使用基于点度量系统的设备都定义了一个用户坐标空间。这是几乎在你所有的代码都会用到的标准坐标空间。例如，当你要操控视图的几何结构或者调用Core Graphics方法来绘制内容时会用到点和用户坐标空间。即使有时用户坐标空间里的坐标时直接映射到设备屏幕的像素，你还是永远不应该假设这是永远不变的。相反，你应该记住： 一个点并不一定对应着屏幕上的一个像素 在设备层面，所有由你指定的视图上的坐标在某些点上必须被转化成像素。然而，从用户坐标空间上的点到设备坐标空间上的像素通常由系统来处理。UIKit和Core Graphics都主要使用基于向量的绘制模型，所有的坐标值都被指定为使用点。这样，如果你用Core Graphics画了一条曲线，你会用一些值来指定这条曲线，而不管底层屏幕使用怎样的解决方法。

当你需要处理图像或者其他基于像素的技术，像OpenGL ES时，iOS会帮你管理这些像素。对于存储为应用程序的束中的资源的静态图像文件，iOS定义了一些约定，可以指定不同像素密度的图像，也可以在加载图像时最大限度的适应当前屏幕的解决方案。视图也提供了关于当前放缩因子的信息，以便你可以适当的调整任何基于像素的绘制代码来适应有更高级解决方案的屏幕。在不同屏幕的解决方案中处理基于像素内容的技术可以在"Supporting High-Resolution Screens"和"Drawing and Printing Guide for iOS"找到描述。

视图的运行时交互模型

当用户和界面进行交互时，或者由代码程序性的改变一些东西时，一系列复杂的事件就会发生在UIKit的内部来处理这些交互。在这个系列中的某些点，UIKit唤出你的视图类，同时给它们一个机会去响应程序的行为。理解这些唤出点对于理解视图在哪里融入系统很重要。Figure 1-7 展示了这些事件的基本序列，从用户触屏开始到图形系统更新屏幕内容来响应结束。同样的事件序列也会发生在任何程序性启动的动作。



以下的步骤分解了图1－7中的事件序列，既解释了在每一步发生了什么，也解释了应用如何响应 1 用户触屏 2 硬件报告触摸事件给UIKit框架 3 UIKit框架将触摸事件打包成UIEvent对象，同时分发给适合的视图。（对于UIKit框架如何提交事件给视图的详细解释，查看 Event Handing Guide for iOS） 4 视图中的事件处理代码可能进行以下的动作来响应： 改变视图或者其子视图的属性（frame， bounds， alpha， 等等） 调用setNeedsLayout方法以标记该视图（或者它的子视图）为需要进行布局更新 调用setNeedsDisplay或者setNeedsDisplayInRect:方法以标记该视图（或者它的子视图）需要进行重画 通知一个控制器关于一些数据的更新 当然，哪些事情要做，哪些方法要被调用是由视图来决定的。 5 如果一个视图的几何结构改变了，UIKit会根据以下几条规则来更新它的子视图： a 如果自动重设尺寸的规则在发生作用，UIKit会根据这些规则来调整视图。获取更多关于自动重设尺寸规则如何工作，查看"Handling Layout Changes Automatically Using Autoresizing Rules." b 如果视图实现了layoutSubviews方法，UIKit会调用它。你可以在你的定制视图中覆盖这个方法同时用它来调整任何子视图的位置和大小。例如，一个提供了巨大滚动区域的视图会需要使用几个子视图作为“瓦块”而不是创建一个不太可能放进内存的巨大视图。在这个方法的实现中，视图会隐藏任何屏幕外的子视图，或者重定位它们然后用来绘制新的可视内容。作为这个流程的一部分，视图的布局代码也可以废止任何需要被重画的视图。 6 如果任何视图的任何部分被标记为需要重画，UIKit会要求视图重画自身。 对于显式的定义了drawRect:方法的定制视图，UIKit会调用这个方法。这方法的实现应该尽快重画视图的指定区域，并且不应该再做其他事。不要在这个点上做额外的布局，也不要改变应用的数据模型。提供这个方法仅仅是为了更新视图的可视内容。 标准的系统视图通常不会实现drawRect:方法，但是也会在这个时候管理它们的绘制。

7 任何已经更新的视图会与应用余下的可视内容组合在一起，同时被发送到图形硬件去显示。 8 图形硬件将已解释内容转化到屏幕上。

注意：上面的更新模型主要应用于使用标准系统视图和绘制技术的应用。使用OpenGL ES来绘制的应用通常会配置一个单一的全屏视图和直接绘制相关的OpenGL图像上下文。你的视图还是应该处理触屏事件，但是它是全屏的，毋需给子视图布局或者实现drawRect:方法。获取更多关于使用OpenGL ES的信息，查看 OpenGL ES Programming Guide for iOS.

给定之前的一系列步骤，将自己的定制视图整合进去的方法包括： 事件处理方法： touchesBegan:withEvent: touchesMoved:withEvent: touchesEnded:withEvent: touchesCancelled:withEvent: layoutSubviews方法 drawRect:方法 这些是视图的最常用的覆盖方法，但是你可能不需要覆盖全部。如果你使用手势识别来处理事件，你不需要覆盖事件处理方法。相似的，如果你的视图没有包含子视图或者它的尺寸不会改变，那就没有理由去覆盖layoutSubviews方法。最后，只有当视图内容会在运行时改变，同时你要用UIKit或者Core Graphics等本地技术来绘制时才需要用到drawRect。

要记住这些是主要的整合点，但是不仅仅只有这些。UIView类中有些方法是专门设计来给子类覆盖的。你应该到UIView Class Reference中查看这些方法的描述，以便在定制时清楚哪些方法适合给你覆盖。

有效使用视图的提示

当你需要绘制一些标准系统视图不能提供的内容时，定制视图是很有用的。但是你要负责保证视图的性能要足够的高。UIKit会尽可能的优化视图相关的行为，也会帮助你提高性能。然而，考虑一些提示可以帮助到UIKit。

重要：在调整绘制代码之前，你应该一直收集与你视图当前性能有关的数据。估量当前性能让你可以确定是否真的有问题，同时如果真的有问题，它也提供一个基线，让你在未来的优化中可以比较。

视图不会总是有一个相应的视图控制器

在应用中，视图和视图控制器之间的一对一关系是很少见的。视图控制器的工作是管理一个视图层次，而视图层次经常是包含了多个视图，它们都有自包含特性。对于iPhone应用，每个视图层次通常都填满了整个屏幕，尽管对于iPad应用来说不是。

当你设计用户界面的时候，考虑到视图控制器的所扮演的角色是很重要的。视图控制器提供了很多重要的行为，像协调视图的展示，协调视图的剔除，释放内存以响应低内存警告，还有翻转视图以响应界面的方向变更。逃避这些行为会导致应用发生错误。

获取更多关于视图控制器的信息，查看 View Controller Programming Guide for iOS

最小化定制的绘画

尽管定制的绘画有时是需要的，但是你也应该尽量避免它。真正需要定制绘画的时候是已有的视图类无法提供足够的表现和能力时。任何时候你的内容都应该可以被组装到其他视图，最好结果时组合那些视图对象到定制的视图层次。

利用内容模式

内容模式可以最小化重画视图要花费的时间。默认的，视图使用UIViewContentModeScaleToFill 内容模式，这个模式会放缩视图的已有内容来填充视图的frame矩形。需要时你可以改变这个模式来调整你的内容，但是应该避免使用UIViewContentModeRedraw内容模式。不管哪个内容模式发生作用，你都可以调用setNeedsDisplay或者setNeedsDisplayInRect:方法来强制视图重画它的内容。

可能的话将视图声明为不透明

UIKit使用opaque属性来决定它是否可以优化组合操作。将一个定制视图的这个属性设置为YES会告诉UIKit不需要解释任何在该视图后的内容。这样可以为你的绘制代码提高性能并且是推荐的。当然，如果你将这个属性设置为YES，你的视图一定要用不透明的内容完全填充它的bounds矩形。

滚动时调整视图的绘制行为

滚动会导致数个视图在短时间内更新。如果视图的绘制代码没有被适当的调整，滚动的性能会非常的缓慢。相对于总是保证视图内容的平庸，我们更倾向于考虑滚动操作开始时改变视图行为。例如，你可以暂时减少已解释的内容，或者在滚动的时候改变内容模式。当滚动停止时，你可以将视图返回到前一状态，同时需要时更新内容。

不要嵌入子视图来定制控制

尽管在技术上增加子视图到标准系统控制对象－继承自UIControl的类－是可行的，你还是永远不应该用这种方法来定制它们。控制对象支持定制，它们有显式并且良好归档的接口。例如，UIButton类包含了设置标题和背景图片的方法。使用已定义好的定制点意味着你的代码总是会正确的工作。不用这些方法，而嵌入一个定制的图像视图或者标签到按钮中去会导致应用出现未预期的结果。

UIWindow的作用

和Mac OS X的应用程序有所不同，iPhone应用程序通常只有一个窗口，表示为一个UIWindow类的实例。您的应用程序在启动时创建这个窗口（或者从nib文件进行装载），并往窗口中加入一或多个视图，然后将它显示出来。窗口显示出来之后，您很少需要再次引用它。

在iPhone OS中，窗口对象并没有像关闭框或标题栏这样的视觉装饰，用户不能直接对其进行关闭或其它操作。所有对窗口的操作都需要通过其编程接口来实现。应用程序可以借助窗口对象来进行事件传递。窗口对象会持续跟踪当前的第一响应者对象，并在UIApplication对象提出请求时将事件传递它。

还有一件可能让有经验的Mac OS X开发者觉得奇怪的事是UIWindow类的继承关系。在Mac OS X中，NSWindow的父类是NSResponder；而在iPhone OS中，UIWindow的父类是UIView。因此，窗口在iPhone OS中也是一个视图对象。不管其起源如何，您通常可以将iPhone OS上的窗口和Mac OS X的窗口同样对待。也就是说，您通常不必直接操作UIWindow对象中与视图有关的属性变量。

在创建应用程序窗口时，您应该总是将其初始的边框尺寸设置为整个屏幕的大小。如果您的窗口是从nib文件装载得到，Interface Builder并不允许创建比屏幕尺寸小的窗口；然而，如果您的窗口是通过编程方式创建的，则必须在创建时传入期望的边框矩形。除了屏幕矩形之外，没有理由传入其它边框矩形。屏幕矩形可以通过UIScreen对象来取得，具体代码如下所示：

```
UIWindow* aWindow = [[[UIWindow alloc] initWithFrame:[[UIScreen mainScreen] bounds]] autorelease];
```

虽然iPhone OS支持将一个窗口叠放在其它窗口的上方，但是您的应用程序永远不应创建多个窗口。系统自身使用额外的窗口来显示系统状态条、重要的警告、以及位于应用程序窗口上方的其它消息。如果您希望在自己的内容上方显示警告，可以使用UIKit提供的警告视图，而不应创建额外的窗口。

UIView是作用视图

是UIView类的实例，负责在屏幕上定义一个矩形区域。在iPhone的应用程序中，视图在展示用户界面及响应用户界面交互方面发挥关键作用。每个视图对象都要负责渲染视图矩形区域中的内容，并响应该区域中发生的触碰事件。这一双重行为意味着视图是应用程序与用户交互的重要机制。在一个基于模型-视图-控制器的应用程序中，视图对象明显属于视图部分。

除了显示内容和处理事件之外，视图还可以用于管理一或多个子视图。子视图是指嵌入到另一视图对象边框内部的视图对象，而被嵌入的视图则被称为父视图或超视图。视图的这种布局方式被称为视图层次，一个视图可以包含任意数量的子视图，通过为子视图添加子视图的方式，视图可以实现任意深度的嵌套。视图在视图层次中的组织方式决定了在屏幕上显示的内容，原因是子视图总是被显示在其父视图的上方；这个组织方法还决定了视图如何响应事件和变化。每个父视图都负责管理其直接的子视图，即根据需要调整它们的位置和尺寸，以及响应它们没有处理的事件。

由于视图对象是应用程序和用户交互的主要途径，所以需要在很多方面发挥作用，下面是其中的一小部分：

描画和动画

- 视图负责对其所属的矩形区域进行描画。
- 某些视图属性变量可以以动画的形式过渡到新的值。

布局和子视图管理

- 视图管理着一个子视图列表。
- 视图定义了自身相对于其父视图的尺寸调整行为。
- 必要时，视图可以通过代码调整其子视图的尺寸和位置。
- 视图可以将其坐标系统下的点转换为其它视图或窗口坐标系统下的点。

事件处理

- 视图可以接收触摸事件。
- 视图是响应者链的参与者。

在iPhone应用程序中，视图和视图控制器紧密协作，管理若干方面的视图行为。视图控制器的作用是处理视图的装载与卸载、处理由于设备旋转导致的界面旋转，以及和用于构建复杂用户界面的高级导航对象进行交互。更多这方面的信息请参见“视图控制器的作用”部分。

本章的大部分内容都着眼于解释视图的这些作用，以及说明如何将您自己的定制代码关联到现有的UIView行为中。

UIView类定义了视图的基本行为，但并不定义其视觉表示。相反，UIKit通过其子类来为像文本框、按键、及工具条这样的标准界面元素定义具体的外观和行为。图2-1显示了所有UIKit视图类的层次框图。除了UIView和UIControl类是例外，这个框图中的大多数视图都设计为可直接使用，或者和委托对象结合使用。

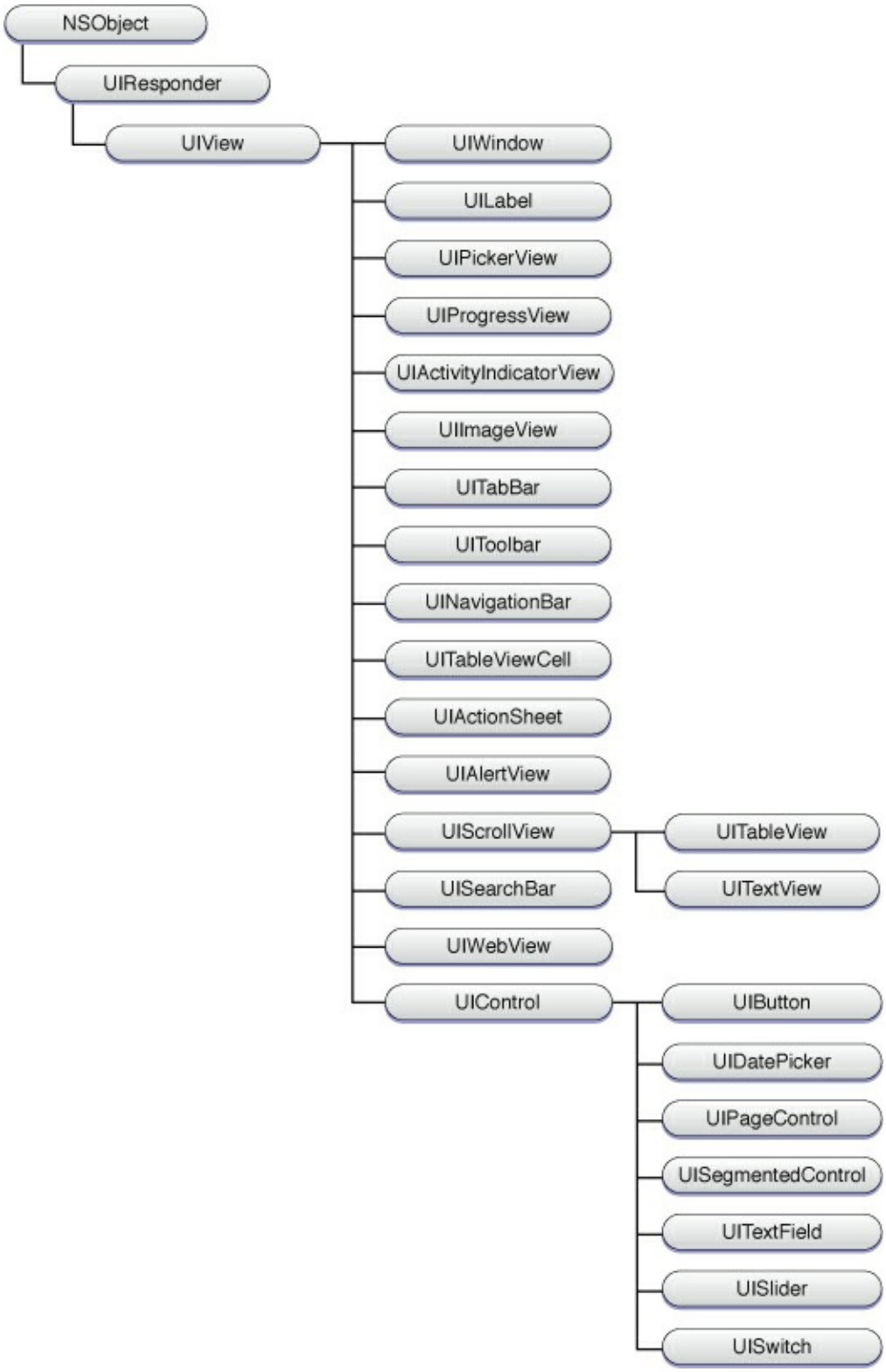


图2-1 视图的类层次

这个视图层次可以分为如下几个大类：

容器

容器视图用于增强其它视图的功能，或者为视图内容提供额外的视觉分隔。比如，UIScrollView类可以用于显示因内容太大而无法显示在一个屏幕上的视图。UIT

UIToolbar对象则是一个特殊类型的容器，用于为一或多个类似于按键的项提供视觉分组。工具条通常出现在屏幕的底部。Safari、Mail、和Photos程序都使用

控件

控件用于创建大多数应用程序的用户界面。控件是一种特殊类型的视图，继承自UIControl超类，通常用于显示一个具体的值，并处理修改这个值所需要的所有用

显示视图

控件和很多其它类型的视图都提供了交互行为，而另外一些视图则只是用于简单地显示信息。具有这种行为的UIKit类包括UIImageView、 UILabel、UIProgre

文本和web视图

文本和web视图为应用程序提供更为高级的显示多行文本的方法。UITextView类支持在滚动区域内显示和编辑多行文本；而UIWebView类则提供了显示HTML内

警告视图和动作表单

警告视图和动作表单用于即刻取得用户的注意。它们向用户显示一条消息，同时还有一或多个可选的按键，用户通过这些按键来响应消息。警告视图和动作表单的

导航视图

页签条和导航条和视图控制器结合使用，为用户提供从一个屏幕到另一个屏幕的导航工具。在使用时，您通常不必直接创建UITabBar和UINavigationBar的项，

窗口

窗口提供一个描画内容的表面，是所有其它视图的根容器。每个应用程序通常都只有一个窗口。更多信息请参见“UIWindow的作用”部分。

除了视图之外，UIKit还提供了视图控制器，用于管理这些对象。更多信息请参见“视图控制器的作用”部分。

1125视图控制器

运行在iPhone OS上的应用程序在如何组织内容和如何将内容呈现给用户方面有很多选择。含有很多内容的应用程序可以将内容分为多个屏幕。在运行时，每个屏幕的背后都是一组视图对象，负责显示该屏幕的数据。一个屏幕的视图后面是一个视图控制器其作用是管理那些视图上显示的数据，并协调它们和应用程序其它部分的关系。

UIViewController类负责创建其管理的视图及在低内存时将它们从内容中移出。视图控制器还为某些标准的系统行为提供自动响应。比如，在响应设备方向变化时，如果应用程序支持该方向，视图控制器可以对其管理的视图进行尺寸调整，使其适应新的方向。您也可以通过视图控制器来将新的视图以模式框的方式显示在当前视图的上方。

除了基础的UIViewController类之外，UIKit还包含很多高级子类，用于处理平台共有的某些高级接口。特别需要提到的是，导航控制器用于显示多屏具有一定层次结构的内容；而页签条控制器则支持用户在一组不同的屏幕之间切换，每个屏幕都代表应用程序的一种不同的操作模式。

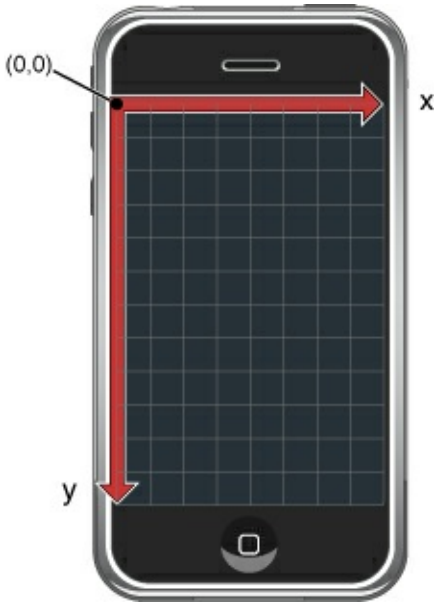
113动画

12布局

《核心动画编程》的某个翻译版本把UIKit的布局模型翻译成了几何布局模型，这个词非常贴切，原始的英文是“struts and springs”。字面翻译就是结构和弹簧。其实说白了就是一种绝对布局模型，这种布局模型的核心数据就是一个对象的几何属性。所以翻译成几何布局模型还是比较贴切的。

121坐标系

UIKit中的坐标是基于这样的坐标系统：以左上角为坐标的原点，原点向下和向右为坐标轴正向。坐标值由浮点数来表示，内容的布局和定位因此具有更高的精度，还可以支持与分辨率无关的特性。图2-3显示了这个相对于屏幕的坐标系统，这个坐标系统同时也用于UIWindow和UIView类。视图坐标系统的方向和Quartz及Mac OS X使用的缺省方向不同，选择这个特殊的方向是为了使布局用户界面上的控件及内容更加容易。



您在编写界面代码时，需要知道当前起作用的坐标系统。每个窗口和视图对象都维护一个自己本地的坐标系统。视图中发生的所有描画都是相对于视图本地的坐标系统。但是，每个视图的边框矩形都是通过其父视图的坐标系统来指定，而事件对象携带的坐标信息则是相对于应用程序窗口的坐标系统。为了方便，UIWindow和UIView类都提供了一些方法，用于在不同对象之间进行坐标系统的转换。

虽然Quartz使用的坐标系统不以左上角为原点，但是对于很多Quartz调用来说，这并不是问题。在调用视图的drawRect:方法之前，UIKit会自动对描画环境进行配置，使左上角成为坐标系统的原点，在这个环境中发生的Quartz调用都可以正确地在视图中描画。您唯一需要考虑不同坐标系统之间差别的场合是当您自行通过Quartz建立描画环境的时候。

122平面内布局

在UIKit的几何布局模型中核心的一个数据结构是：`CGRect`，它确定了一个View（或者Layer，我们这里先只考虑View的情况，想不详细展开来说其他的）在父View中坐标系的绝对位置。

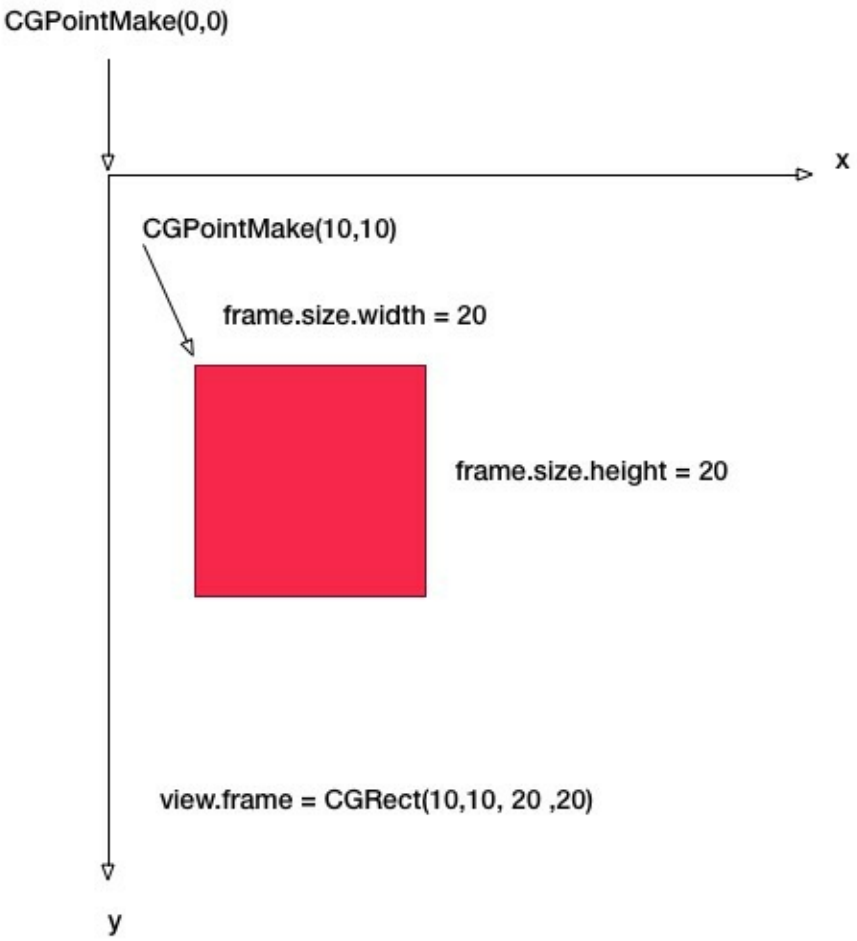
那让我们来看一下CGRect的定义：

```
/* Points. */

struct CGPoint {
  CGFloat x;
  CGFloat y;
};
typedef struct CGPoint CGPoint;

/* Sizes. */

struct CGSize {
  CGFloat width;
  CGFloat height;
};
typedef struct CGSize CGSize;

/* Rectangles. */

struct CGRect {
  CGPoint origin;
  CGSize size;
};
typedef struct CGRect CGRect;
```

我们发现其实一个CGRect中包含了一个原点（point）和一组宽高的信息（size）。其实一个CGRect就是描述了一个长方形的块，就像下图的红色方块一样的东西，我们的每一个View在坐标系中都会被表示为一个长方形的块状物。

比如我们有一个位置是{{10,10},{20,20}}的View：

```
UIView* aView = [UIView new];
aView.frame = CGRectMake(0, 0 , 100 ,100)
```



在它的父类的坐标系中展示如下图：

我们能够发现红色的View的frame信息所描述的几何位置，其实是其在父View坐标系中的绝对位置。死死的写在那里的。所以像UIKit这样的布局模型又叫绝对布局模型，如果你用过jave的Swing或者c++的QT，你可能会觉得这种绝对布局模型好麻烦，好啰嗦。没有布局管理器的概念，什么都是绝对的。但是只能说各有各的好处把。QT之类的有布局管理器的开发复杂界面的确方便，但是像在iphone这样的手机设备上，机器屏幕有限、设备性能有限，用绝对布局模型还是比较合适。苹果在IOS5之后也引入了一些相对布局的东西（autolayout）正好这里有篇文章是说其性能的Auto Layout Performance on iOS。读过之后你能发现自动布局在复杂界面情况下的性能的确比较差的。所以像UIKit这种比较原始的绝对布局在性能上还是有优势的。

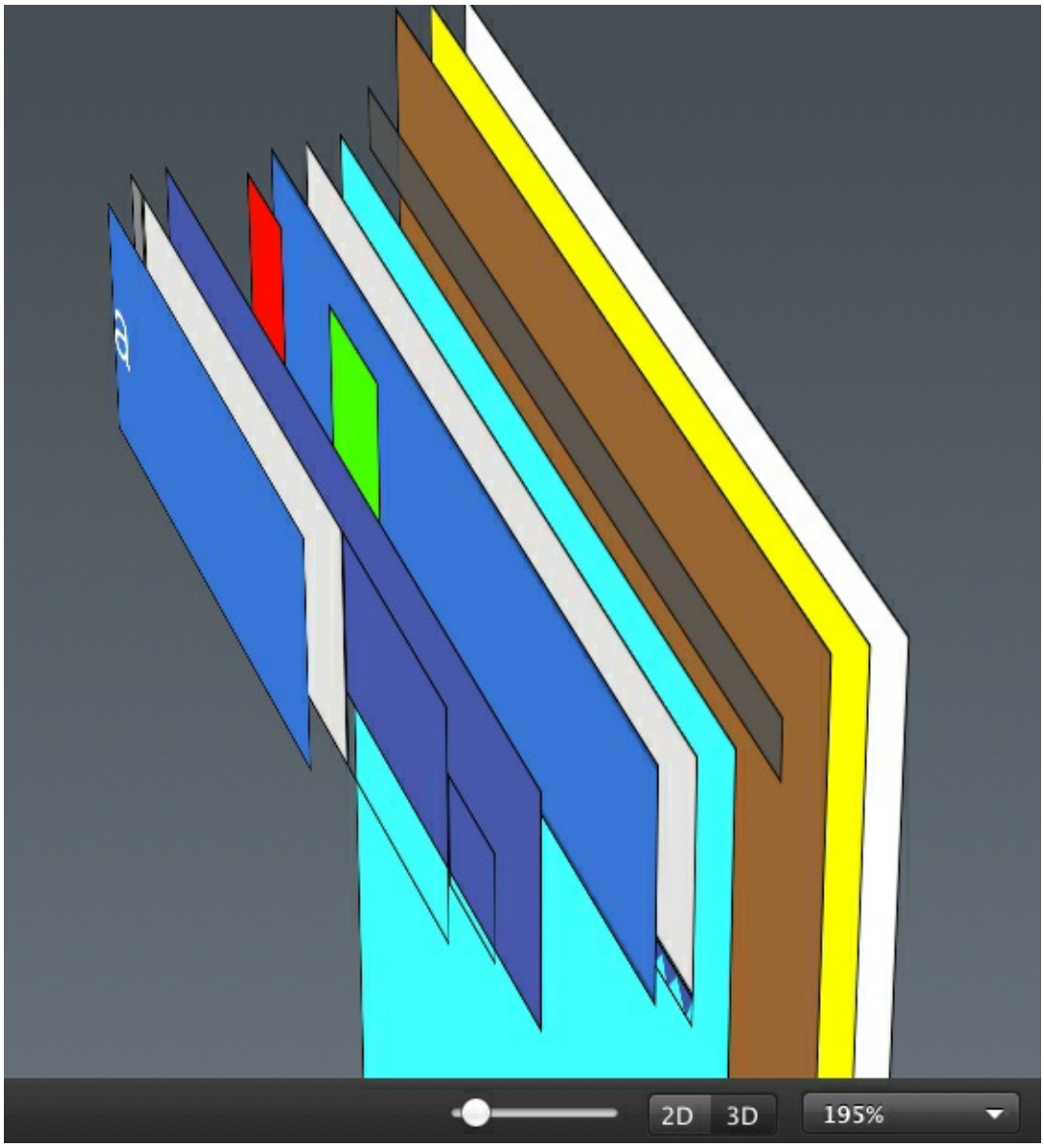
扯回来，通过上图我们能够发现，UIKit的坐标系是一个二维平面坐标系，以左上角为原点，x轴横向扩展，y轴纵向向下扩展。y轴的防线可能和我们以前上学的时候，学的坐标系有点不太一样。这个估计是考虑在ios屏幕上布局的时候我们一般都是从上往下布局，y轴向下方便我们布局吧。既然知道了UIKit的坐标系统是一个二维平面坐标系统，那么我们以前学的很多几何知识就能够在这个坐标系统中尽情使用了。这里知识点太多不一而足，也是埋个伏笔，知道我们在些TableView的时候会用到很多几何上的知识。

同时，你可以把整个UIKit的View布局系统看成一个递归的系统，一个view在父view中布局，父view又在其父view中布局，最后直到在UIWindow上布局。这样递归的布局开来，就能构建起我们看到的app的界面。

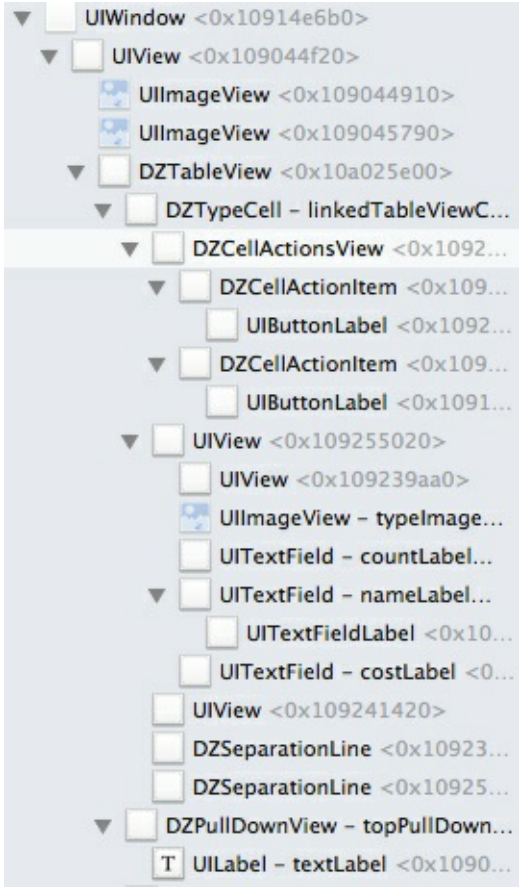
Z-Order布局

不知道读者有没有试图想过当我们的一个程序真正运行起来的时候，那些被我们实例化的UIView和他们的子类们是以怎样的形态出现在我们的屏幕上的。前面我们介绍了IOS的几何布局框架，而通过这些框架组合起来的UIView和其子类们组成了什么东西？

是一个树形结构，或者准确说是一个倒金字塔结构。我们先看一张3D的DZTableView在只有一个Cell时候的结构图：



通过这张图我们能够非常明显的看到这是一个一层叠一层结构。最先面的是UIWindow的实例，我们的DZTableVIew还有其他一些自定义的控件们，一层叠一层的堆在了UIWindow的实例上面。我们再换个角度看一下：



这张图是个典型的树结构的图，一层层展开就是之后就是我们的DZTableView。通过这两张图比较形象的展示，出DZTableview最后在空间上成了一个什么东西。是的，是“空间”。用这两张图的主要目的就是，希望读者能够在自己的脑海中构建起对于UIView层次数Z-order（z轴）的概念。因为除了除了我布局UIView的各种子类的实例对象的时候，除了布局其二维属性 `frame` 等，其实我们还需要布局他们的3维属性，即Z-order顺序。

那我们来看看UIKit给我们提供了哪些函数来做这个事情：

```
- addSubview:
- bringSubviewToFront:
- sendSubviewToBack:
- removeFromSuperview
- insertSubview:atIndex:
- insertSubview:aboveSubview:
- insertSubview:belowSubview:
- exchangeSubviewAtIndex:withSubviewAtIndex:
- isDescendantOfView:

//属性部分
 superview property
 subviews property
 window property
```

这些函数通过他们的名字很容易理解他们的意思。 `addSubView:` 直接将一个View附加在当前View上面。而这个附加过程，可以暂时先简单的理解成一个堆栈的PUSH操作。先附加的View被压在下面，后附加的View在最上面。在暂时把其他函数抛在一遍的情况下，和对堆栈的操作一样，我们是通过UIView的添加顺序（LILO）来控制视图的Z-Order的。但是和堆栈不同的是，UIView没有提供POP操作。因为实际上，解释一个视图中其子视图的比较好的模型是可随机访问的数组，类似于NSArray。不同的是，在父视图中没有直接删除子视图操作。删除操作下放给了我子视图。如果一个要移除一个视图，我们需要调用子视图的 `removeFromSuperview` 方法，来将其从父视图的层次数种移除。然后其他视图就像你在一堆积木中，抽了比较下面的一根一样，在其下放的保持原先的顺序不动，在其上方的自动下落，添补被抽离的积木造成的空缺，z-order的顺序自动减一。

我曾经思考过，为什么UIKit在设计的时候，不直接在父视图中完成所有的子视图层次数的操作，而是把删除的操作放到了子视图中。刚开始从设计的角度去考虑，像增删改查这些基本的操作不应该放在一起吗？放在一起的话，能够降低子视图和父视图之间的耦合。全部都在父视图中做了，子视图只是一个被统治而已。这样的模型比较简单，也比较好理解。后来在世界

的编程实现中，慢慢体会到了apple那群牛逼的工程师的用心良苦。个人认为这绝对是他们对实际编程经验的一个提炼。我们在附加视图的时候，一半都是有了一个父视图的实例有了一个子视图的实例，然后将子视图附加在父视图上，而这个过程一半都是在父视图的类的某些成员函数中进行的。在父视图的成员函数中进行，我们能够非常方便的获取当前视图（父视图）和其子视图的实例，然后调用`addSubView`函数。而在删除的时候，我们一般都是在子视图中进行的。直接在子视图的类的某个函数中，当检测到子视图满足一定的状态的时候，将其删除。而在子视图中相对来说获取父视图的实例编程是比较啰嗦的：

```
//假设UIView存在一个函数removeSubView:那么这个过程是
[self.superView removeSubView:self]
//而removeSubView大概要做这么几个事情

....
-（void） removeSubView:(UIView*)a
{
   //确定是否有这个子视图
   if([self isContentSubView:a])
   {
      //找到子视图的Z-order
      int index = [self indexOfSubView:a];
      //删除掉
      [self removeSubViewAtIndex:index];
   }
}
```

何必要获取一遍父视图的实例进行删除操作呢，直接把这个过程封装起来，用一句：`[self removeFromSuperView]`多好啊。 而且即使在父视图进行层次数操作的时候，也是有了子视图的实例之后进行操作：`[aView removeFromeSuperView]`。这种封装，应该属于实用性的一种封装。细细品味，很有味道。

而其他的调整子视图的Z-Order的函数，通过函数名字我们也很容易理解。

1. `insertSubview:atIndex:`直接将一个子视图加到特定的位置。
2. `insertSubview:aboveSubview:`将一个子视图加到一个相对位置，在特定视图的上方。
3. `insertSubview:belowSubview:`将一个子视图加到一个相对位置，在特定视图的下方。
4. `exchangeSubviewAtIndex:withSubviewAtIndex:`交换两个特定位置的视图
5. `bringSubviewToFront`将一个视图挪到最上方。
6. `sendSubviewToBack`将一个视图挪到最下方。

而我们其实在调整视图的Z-order的时候，是否想过调整Z-order用什么用呢？想当然的一个答案是，视图不就是层次结构的嘛，有个Z-order也很正常。但是，如果我们能够完全在二维平面中，完成对各种视觉结构的展示，就用不到三维结构。而偏偏我们这个世界是三维的，xyz三个轴才能确定一个物体。看一下那些优秀的游戏引擎把基本上都是直接用3D的模型来描述一个视觉对象。这种强3D的模型，能够接近真实的描述视觉结构，渲染出更加丰富多彩的虚拟世界。而UIKit中核心动画部分也是使用了类似于3D信息比较强的3D模型。所以，使用3D的模型来描述视觉结构师再正常不过的事情了，只是，在UIView这个层次上强3D的模型对于渲染界面来说，意义不是很大，只需要简单的保留一个z-order就OK了。这样利用xyz三个维度构建出来的UIKit的视图模型才能比较真实的拟合我们看到真实世界的例子。

举个例子，比如透明度这个事情。比如我们为了实现给一个UILabel加一个图片背景，我们会怎么做？

```
UIImageView* imgView = [[UIImageView alloc] initWithImage:[UIImage imageNamed:@"a"]];
[self.view addSubview:imgView];

UILabel* label = [UILabel new];
//将Label的背景设置成透明的，这样就能够看到label下面的图片了
label.backgroundColor = [UIColor clearColor];
[self.view addSubview:label];
```

我们会把Label的背景设置成透明的，这样label下放的图片自然就会显示出来。这和我们在真实3D世界中，通过玻璃看窗外一个道理。玻璃叠在了窗外的景色上面，但是当光纤映射到我们眼中时，我们看到的是一副图像。同样Label的例子在我们严重也是一个整体的图像。虽然在使用者严重他们不会在意你有Label了，label有透明度了。label后面还有图像了。他们只关心看在严重的整个图像看起来是个什么样子，但是对于我们开发者来说。我们需要一个恰当的模型来描述这种结构，而对于UIKit来说增加一个Z-order足矣，够用了。至于那些复杂的3D结构信息，交给更加底层的核心动画来处理吧。

如何布局

布局的时候我们都需要做些什么事情

布局顾名思义，就是确定一个View的位置。也就是说我们要在布局中做的事情用一句话说就是：确定UIView的frame属性的值。给每一个UIView和其子类的实例确定frame的属性值。

1、 初始化函数 `- (id)initWithFrame:(CGRect)aRect`

objc构建一个对象使用的是两段式，首先分配内存 `alloc` 然后 `init`，这样的好处就是将内存操作和初始化操作解耦合，让我们能够在初始化的时候对对象做一些必要的操作。这是个很好的思路，我们在做很多事情的时候都可以使用这种两段式的思路。比如布局一个UIView，我们可以分成两部，初始化必要的子view和变量，然后在合适的时机进行布局。

而这个两段式的第一步就是：

```
- (id)initWithFrame:(CGRect)aRect
```

这个函数是无论你用什么初始化函数都会被调用的一个，比如你用 `[UIView new]` 或者 `[[UIView alloc] init]` 都会调用initWithFrame这个函数（有些UIView的子类有特殊情况，比如UITableViewCell，怀疑apple对其做过特殊处理），所以你要是对一个view的变量有初始化的操作尽量往 `initWithFrame` 里面放还是非常合适的。这样能够保证，以后在使用的时候所有的变量都被正确的初始化过。而我们一般会在 `initWithFrame` 中做些什么呢？

1. 添加子View
2. 初始化属性变量
3. 其他一些共用操作

所以我们一般会看到这样的代码

```
- (instancetype) initWithFrame:(CGRect)arect
{
   self = [super WithFrame:arect];
   if (!self) {
      return nil;
   }
   [self commitInit];
   <#init data#>
   return self;
}

- (void) commomInit
{
   <#common init data#>
}
```

在初花的时候将一些共用的初始化操作独立成一个函数 `commomInit` 然后再其中做上面说的事情，这样做的好处就是将初始化的代码集中到一起，如果你在实现的一个其他的什么initWithXXX的时候，直接调用commonInit就可以了。

不得不说的是，千万不要被这个函数的名称withFrame给忽悠了，以为这个函数使用布局用的。在代码逻辑比较清晰的工程中，几乎很少看到在这个函数中进行界面布局的工作。因为UIKit给你提供了一个专门的函数layoutSubViews来干这个事情。而且，在这个函数中做的界面布局的工作，是一次性编码，你的界面布局没有任何复用性，如果父View的大小变了之后，这个View还是傻傻的保持原来的模样。同时也会造成，初始化函数臃肿，导致维护上的困难。

2、 `layoutSubviews` 和 `setNeedsLayout`

上面说了一些 `initWithFrame` 的事情，告诫了千万不要在里面做界面布局的事情，那应该在什么地方做呢？

```
layoutSubviews
```

就是这个地方，这是苹果提供给你专门做界面布局的函数。

我们来看一下文档:

```
The default implementation of this method does nothing on iOS 5.1 and earlier. Otherwise, the default implementation uses any con

Subclasses can override this method as needed to perform more precise layout of their subviews. You should override this method o

You should not call this method directly. If you want to force a layout update, call the setNeedsLayout method instead to do so prior
```

苹果都说了这个是子类化View的时候布局用的。那我们最好是老老实实的在里面做布局的工作。

如何布局

这是个比较有意思的话题，因为可能很多人认为很简单，绝对布局嘛就是写一些死数字嘛，直接写CGRectMake(10,10,20,20)这样的坐标不就行了。如果你真这样认为，那么下面的话可能对你有帮助。

首先，尽量不要在布局的时候直接写死数字，比较稳妥的变法是使用常亮或者宏定义，甚至你定义一个临时变量也都ok，这样代码的可维护性就会变得比较好。

其次，谁说绝对布局的框架不能写成相对布局的方式。Apple提供了一个 `CGGeometry.h` 的文件，里面定义了大量的方便几何布局的函数。比如CGRectGetMaxX用来获取一个View的最大x坐标。你可能会问这有什么用？我们来看段代码：

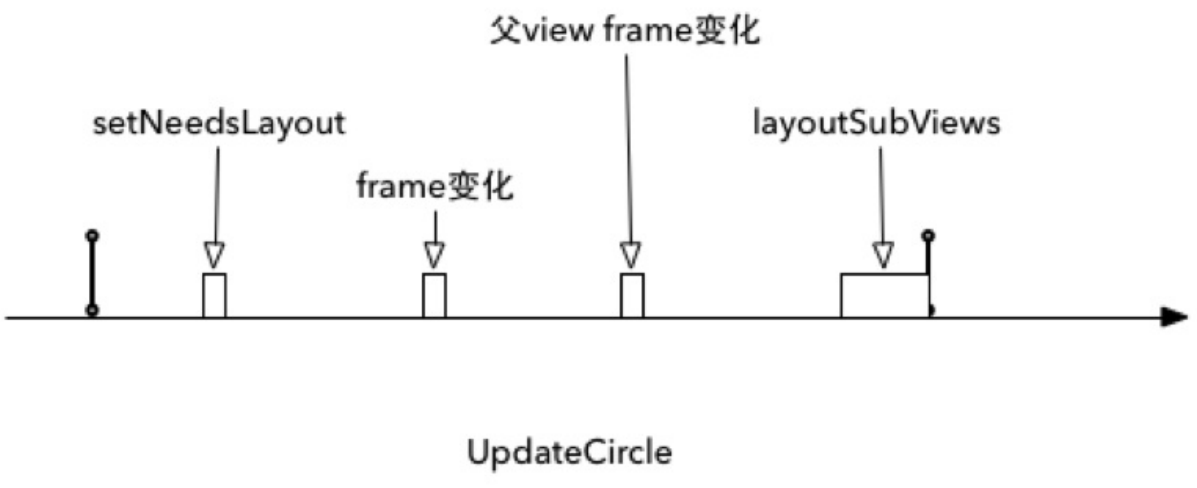
```
    _imageView.frame = CGRectMake(0, 0, width, height);
    _textLabel.frame = CGRectMake(CGRectGetMaxX(_imageView.frame), 0, CGRectGetWidth(self.frame) - CGRectGetMaxX(_imageView
```

下面那个_textLabel的布局就是在_imageView的大小而确定的。这不就是一些布局管理器做的事情吗，这不就是相对布局的概念嘛。所以我们完全可以使用UIKit的几何坐标系统完成一些相对布局的事情，而且也推荐这样做。

什么时候布局

这个就看功能需要了，不过有一点是肯定的就是不要直接调用layoutSubviews函数。UIKit和runtime是捆绑很密切的，apple为了防止界面重新布局过于频繁，所以只在runloop合适的实际来做布局的工作。里面具体的细节，可以google。

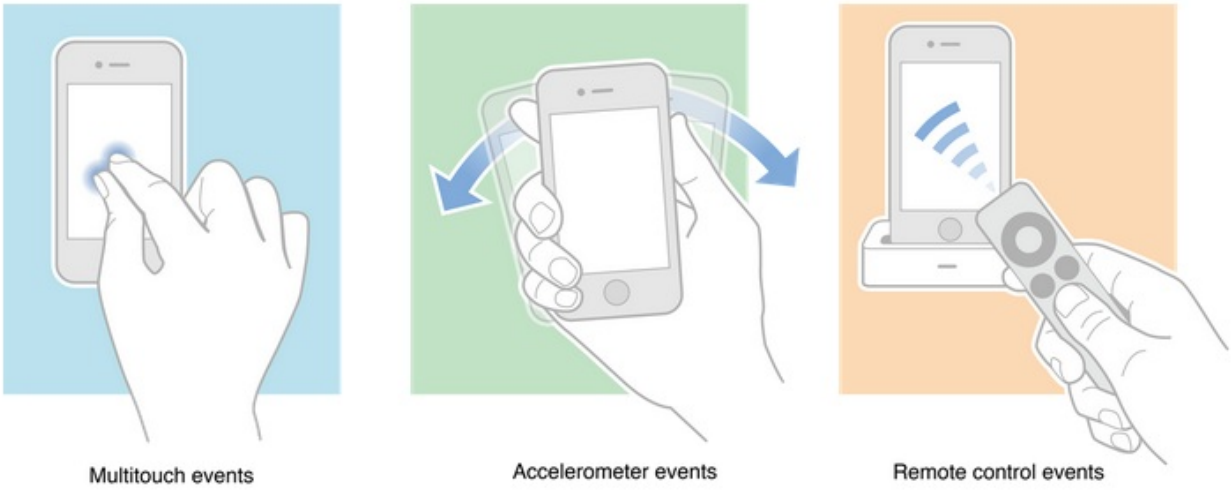
一般你需要重新布局的时候调用 `setNeedsLayout` 标记一下，“我需要重新布局了”。就行了，系统会在下次runloop合适的时机给你布局。



13交互

本篇内容将围绕iOS中事件及其传递机制进行学习和分析。在iOS中，事件分为三类：

- 触控事件（单点、多点触控以及各种手势操作）
- 传感器事件（重力、加速度传感器等）
- 远程控制事件（远程遥控iOS设备多媒体播放等）



这三类事件共同构成了iOS设备丰富的操作方式和使用体验，本次就首先来针对第一类事件：触控事件，进行学习和分析。

我们通常不是简单的把View布局到屏幕上就完事了。我们还需要提供能力让用户能与这些View进行交互。而在UIKit提供的框架中主要的就是触摸事件，当然还有摇晃了等其他一些事情，我们这里先不考虑，主要关于来自屏幕的事件和对其处理方式。

触摸事件是指我们对用户的手指触击屏幕以及在屏幕上移动时的一个抽象。从程序层面上讲就是，在用户发生上述行为时，系统不断发送给我们App的那些事件对象，然后按照特定的路径传递给我们App中的一些对象来处理。处理这些事件的对象主要有两类一个是UIViewController的子类和UIView的子类。在IOS中，我们用UITouch对象来表示一个触摸，而用UIEvent对象来描述一个事件。UIEvent事件对象中包含与一些列与用户操作相关的所有UITouch触摸对象，同时还可以提供与特定窗口相关联的触摸对象。其实，在实际的使用过程中，在UITouch和UIEvent两个对象中，我们使用比较多的一个对象是UITouch。所以我们这里先了解一下UITouch的都包含了那些用户触摸的数据。然后我们再来看看系统式如何传递这些事件并处理他们的。

触摸事件响应相关函数

UITouch对象解析

通过阅读UITouch Class Reference，我们看到在一个UITouch对象中主要是存储了与触摸用户触摸相关的信息：位置信息和时间信息。

```
//获取位置信息
- locationInView:
- previousLocationInView:
//获取时间信息，只读属性
timestamp  property
....
//其他
```

如果我们来对触摸进行抽象的话，也会主要在触摸对象中存储着两类信息。因为对于一个事件来说，大家公认的要素是：时间地点人物故事情节。在用户触摸屏幕这个事情上，人物是用户，故事情节就是开发者在app安排给用户的一些逻辑，这些都是更高层的抽象要关心的事情。而只有时间和地点（位置）是一个触摸对象必须关心的。有了这两方面的信息我们就能够确定，用户在什么时间触摸或触击了屏幕的哪个位置。位置新的的表示使用的CGPoint，对头，就是我们在集合布局框架中介绍的存储点信息的对象。但是UITouch对象存储的知识整个触摸或者触击时间的片段信息或者说状态信息。如果我们要完整的描述一个事件，我们需要一些列的UITouch对象。

假设一个事件有三个过程：从用户手指点到屏幕，在屏幕上移动，再到从屏幕上移开。那么这三个过程最少对对应三个UITouch对象。这就要说到我们是怎样处理触摸事件的了。

处理触摸事件

UIKit使用UIResponder作为响应对象，来响应系统传递过来的时间并进行处理。UIApplication、UIViewController、UIView、和所有从UIView派生出来的UIKit类（包括UIWindow）都直接或间接地继承自UIResponder类。

```
- touchesBegan:withEvent:
- touchesMoved:withEvent:
- touchesEnded:withEvent:
- touchesCancelled:withEvent:
```

事件传递：响应链

当你设计应用程序时，你很可能想要动态地响应事件。 比如，一个触摸(touch)事件可以发生在屏幕上的不同对象中，并且对于一个给定事件你必须决定你想要响应哪个对象并理解哪个对象如何接收事件。

当一个用户生成的事件发生时，UIKit创建一个事件对象，它包含了需要处理该事件的各种信息。 然后把事件对象放在活动应用程序的事件队列中。 对于多个触摸事件，对象是一组打包进一个UIEvent对象的触摸事件。 对于运动事件(motion events)，事件对象根据你使用的不同框架以及你感兴趣的不同移动事件类型发生改变。

事件沿着一个指定的路径传递知道它遇见可以处理它的对象。 首先一个UIApplication 对象从队列顶部获取一个事件并分发(dispatches)它以便处理。 通常，它把事件传递给应用程序的关键窗口对象，该对象把事件传递给一个初始对象来处理。 初始对象取决于事件的类型。

- 触摸事件。 对于触摸事件，窗口对象首先尝试把事件传递给触摸发生的视图。那个视图被称为hit-test(点击测试)视图。寻找hit-test视图的过程被称为hit-testing, 参见 “Hit-Testing Returns the View Where a Touch Occurred.”

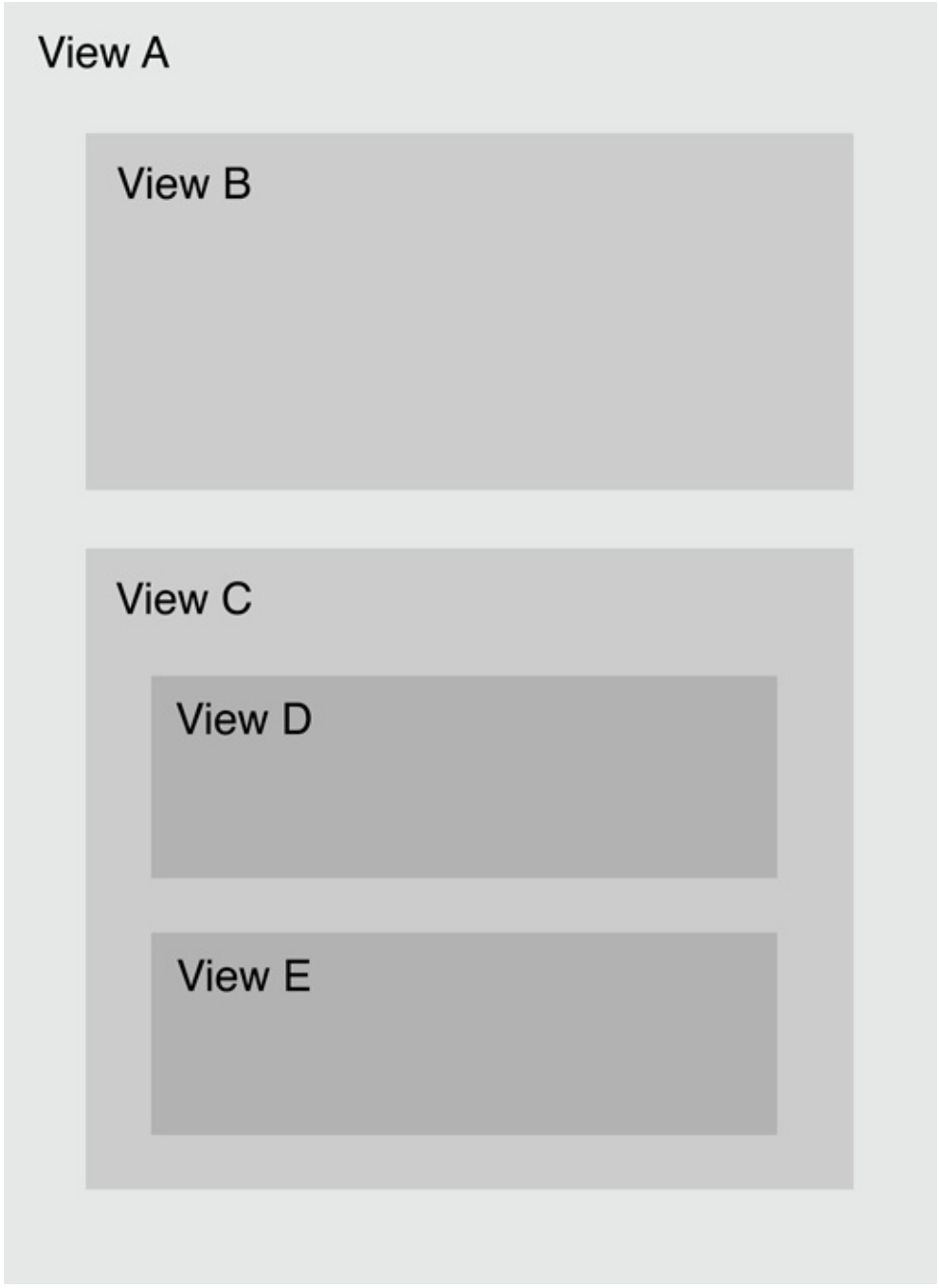
- 运动和远程控制事件。 对于这些事件，窗口对象把shaking-motion(摇晃运动)或远程控制事件传递给第一响应者来处理。第一响应者请参见 “The Responder Chain Is Made Up of Responder Objects.”

这些事件路径的最终目标是找到一个可以处理并相应事件的对象。 因此，UIKit首先把事件传递给最适合处理该事件的对象。对于触摸事件，最适合处理该事件的对象是hit-test视图，而对于其它事件，那个对象是第一响应者。 以下章节讲述了更多关于如何决定hit-test视图和第一响应者的详情。

一、Hit-Testing 返回触摸事件发生的视图

iOS 使用hit-testing来找到事件发生的视图。 Hit-testing包括检查触摸事件是否发生在任何相关视图对象的范围内， 如果是，则递归地检查所有视图的子视图。在视图层次中的最底层视图，如果它包含了触摸点，那么它就是hit-test视图。等 iOS决定了hit-test视图之后，它把触摸事件传递给该视图以便处理。

假设用户触摸了视图E。iOS通过以下吮吸检查子视图来查找hit-test视图：



因为触摸发生在视图A范围内，所以它检查子视图B和C。触摸不在视图B范围内，但是它在视图C范围内，所以它检查子视图D和E。触摸没有在视图D范围内，但是它在视图E范围内。视图E是视图层次结构的最底层并且它包含了触摸，因此它是hit-test视图。

hitTest:withEvent: 方法为给定的CGPoint 和 UIEvent返回hit test 视图。hitTest:withEvent:方法通过在自身调用 pointInside:withEvent: 方法开始。 如果传递到方法hitTest:withEvent:内的点在视图的范围内，pointInside:withEvent:返回YES。然后，方法递归地给每个子视图调用hitTest:withEvent:方法并返回YES。

如果传递到hitTest:withEvent:方法的电不再视图的范围内，pointInside:withEvent:方法返回NO,点被忽视，并且 hitTest:withEvent:返回nil. 如果一个子视图返回NO，则视图层次的整个分支都被忽视，因为如果触摸事件没有发生在那个子视图中，那么事件也不会在任何一个该子视图的子视图中发生。 这意味着在一个子视图中的任何点，如果它在其父视图的外面，那么它不能接收触摸事件，因为触摸点必须在父视图和子视图的范围内。但是如果子视图的clipsToBounds 特性被设置为NO时这可以发生。(主意：一个触摸对象在生命周期内都跟它的hit-test视图相关，即使触摸事件在稍候会移出该视图。)

hit-test是第一处理触摸事件的视图。如果hit-test视图不能处理该事件，事件沿着视图的响应链传递直到系统找到可以处理该事件的视图，响应链在 “The Responder Chain Is Made Up of Responder Objects”中有描述。

二、响应链由响应者对象组成

很多类型的事件都在事件传递中依赖响应链。 响应链是一系列相连的响应者对象。 它由第一个响应者开始，以应用对象结束。 如果第一响应者不能处理该事件，它把事件传递给响应链中的下一个响应者。

响应者对象是可以响应并处理各种事件的对象。 UIResponder 类是所有响应者对象的基类，它为事件处理和通用响应者行为都定义了可编程接口。 UIApplication, UIViewController, 和UIView的实例都是响应者，就是说所有的视图和大多数主要对象都是响应者。 请注意Core 动画层不是响应者。

第一响应者被设计为首先接收事件。通常，第一响应者是一个视图对象。 一个对象通过实现以下事情可以成为第一响应者：重写 canBecomeFirstResponder 方法并返回YES。

接收一个becomeFirstResponder 消息。如果必要，一个对象可以给自己发送该消息。

注意：确保你的应用程序在把一个对象分配为第一响应者之前已经建立了它的对象图。比如，通常你重写viewDidAppear: 方法时会调用becomeFirstResponder方法。 如果你尝试在viewWillAppear:方法里分配第一响应者，那么你的对象图还没有建立，因此becomeFirstResponder方法返回NO。

事件并不是依赖响应链的唯一对象。响应链可以用于以下所有对象：

触摸事件。如果hit-test视图无法处理一个触摸事件。该事件在以hit-test视图开始的响应链中往上传递。

运动事件。 要想用UIKit处理摇晃运动事件，第一响应者必须实现UIResponder类的motionBegan:withEvent: 方法或 motionEnded:withEvent: 方法，参见 “Detecting Shake-Motion Events with UIEvent.”

远程控制事件。 要想处理远程控制事件，第一响应者必须实现UIResponder类的 remoteControlReceivedWithEvent:方法。

动作消息。当用户操纵一个控件，比如一个按钮或开关，并且操作方法的目标(target)为nil时，消息通过以控件视图开始的响应者链里被发送。

编辑菜单消息。 当用户点击了编辑菜单的命令时，iOS使用一个响应者链来找出实现必要方法的对象(比如cut:, copy:, 以及 paste:) 。 更多信息，请看“Displaying and Managing the Edit Menu” 以及示例代码项目，CopyPasteTile.

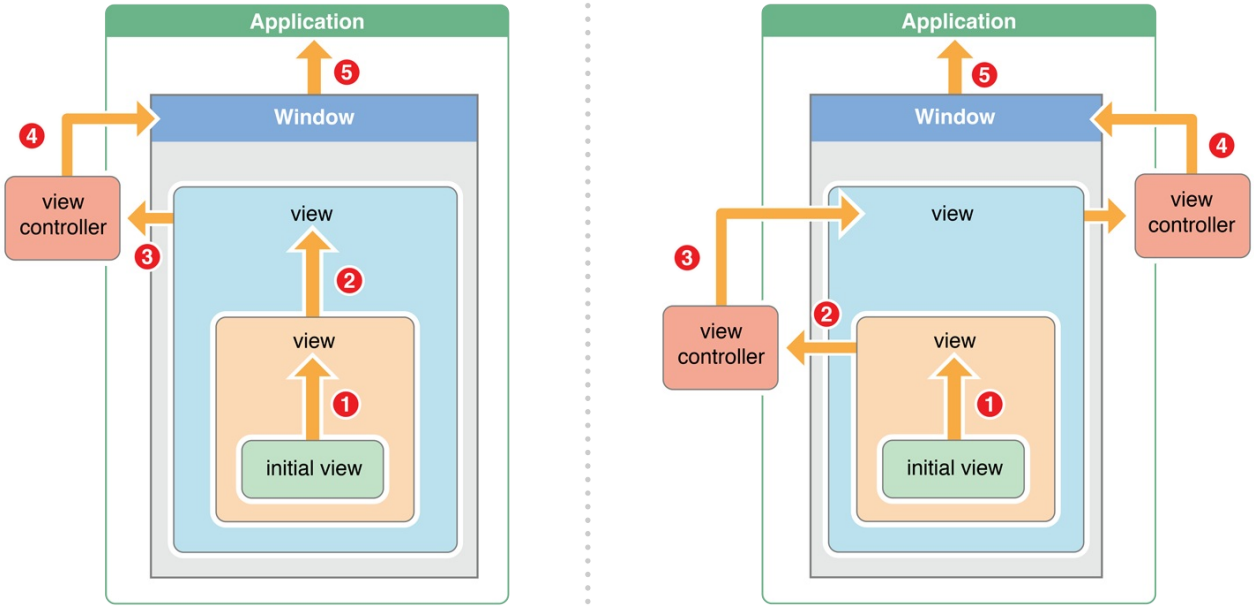
文本编辑。 当用户点击一个文本区或一个文本视图时，那视图自动成为第一响应者。 默认情况下，虚拟键盘出现后文本区或文本视图成为可编辑状态。 你可以显示一个自定义输入视图来取代键盘，如果它更适合你的应用程序。 你还可以添加一个自定义输入视图到任何响应者对象。 更多信息，请看 “Custom Views for Data Input”.

UIKit 自动把用户点击的文本区或文本视图设置为第一响应者；应用程序必须明确地通过becomeFirstResponder方法设置所有其它第一响应者对象。

三、响应者链遵循一个特定的传递路径

如果初始对象---hit-test视图或者第一响应者链--不能处理一个事件， UIKit 把事件传递给响应者链中的下一个响应者。 每个响应者决定是否想要处理该事件或者调用nextResponder 方法把该事件传递给它的下个响应者。 该过程一直持续直到找到一个响应者来处理该事件或者没有任何其它响应者。

当iOS侦测到一个事件时，响应者链序列开始并把事件传递给一个初始对象，初始对象通常为一个视图。 初始视图首先可以处理一个事件。 图2-2显示了为两个应用程序配置的两个不同事件传递路径。 应用程序的事件传递路径依赖于它的特定结构， 但是所有的事件传递路径遵循同样的试探法(heuristics)。



对于左边的应用程序，事件沿着以下路径：

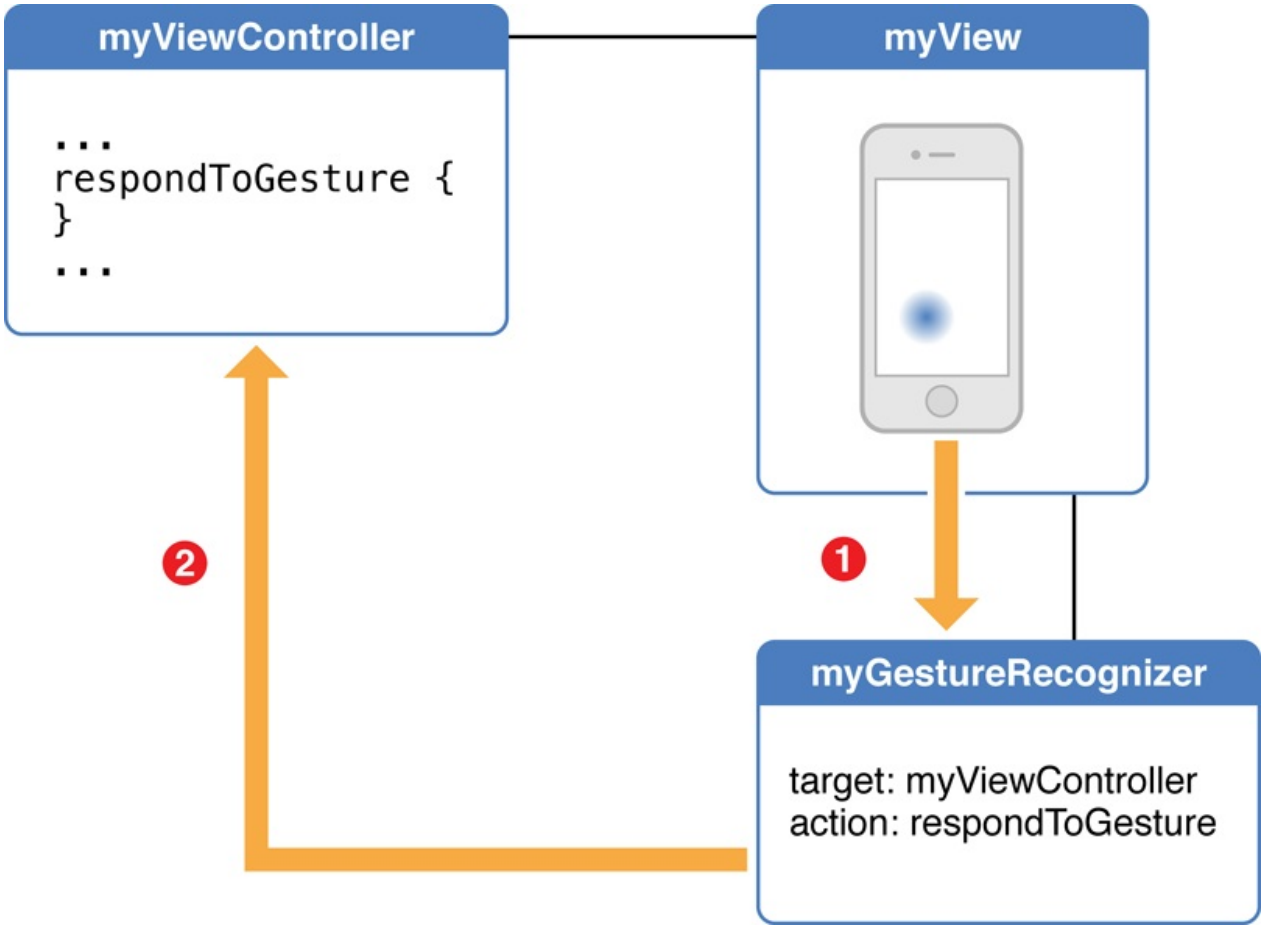
1. 初始视图尝试处理事件或消息。 如果它不能处理该事件，它把事件传递给它的父视图，因为初始视图在它的视图控制器的视图层次里不是最顶层视图。
2. 父视图尝试处理该事件。 如果父视图不能处理该事件，它把事件传递给父视图的父视图，因为该父视图还不是视图层次结构里的最顶层视图。
3. 视图控制器的视图层次结构中的最顶层视图尝试处理该事件。如果它还是不能处理该事件，它把事件传递给它的视图控制器。
4. 视图控制器尝试处理该事件，如果它不能处理，则把事件传递给窗口。
5. 如果窗口还是不能处理该事件，它把事件传递给单个应用程序对象。
6. 如果应用程序对象还是不能处理该事件，它把事件丢弃。

右边的应用程序沿着稍稍不同的路径，但是所有的事件传递路径遵循这些探索法：

1. 一个视图在它的视图控制器的视图层次中把事件向上传递，直到它达到最顶层视图。
2. 最顶层视图把事件传递给它的视图控制器。
3. 视图控制器把事件传递给它的最顶层视图的父视图。 重复步骤1-3 直到事件到达根视图控制器。
4. 根视图控制器把事件传递给窗口对象。
5. 窗口把事件传递给应用程序对象。

> 重要提示： 如果你实现了一个自定义视图用UIKit来处理远程控制事件，操作消息，摇晃运动事件，或者编辑菜单消息，不要直接把事件或消息分配给nextResponder并在响应者链里向上发送。 相反，调用父类的当前事件处理方法的实现方法并且让UIKit为你处理响应者链的遍历。

手势识别把低层次的事件处理代码转换为高层次的操作。 它们是你连接到视图的各种对象，它们允许视图像控件一样响应各种操作。 手势识别翻译(interpret)各种触摸来决定它们是否响应一个特定的手势，比如一个点击(swipe), 捏合(pinch)，或旋转。 如果它们识别到它们指定的手势，就给目标对象发送一个操作信息。 目标对象通常是视图的视图控制器，它响应如图1-1中所示的手势。该设计模式既有力又简单；你可以动态地决定视图响应什么操作，你也可以给视图添加各种手势识别而不必要子类化该视图。



一、使用手势识别器来简化事件处理

UIKit 框架提供了一些已经预先定义好的手势识别来侦测各种常用手势。如果可能的话，最好是使用预定义的手势识别，因为它们缩减了你必须写的代码总量。另外，使用一个标准的手势识别而不是自己编写以确保你的应用程序如用户期望的那样工作。

如果你想你的应用程序识别一个独特的手势，比如一个勾或一个漩涡状运动，你可以创建你自己的自定义手势识别。 学习如何设计和实现你自己的手势识别，请看 “Creating a Custom Gesture Recognizer.”

1、内建手势识别来识别各种常用手势

当你设计应用程序时，考虑你想要开启什么手势。 然后，对于每个手势，决定列表1-1中的预定义手势识别是否就足够。

Gesture	UIKit class
Tapping (any number of taps)	UITapGestureRecognizer
Pinching in and out (for zooming a view)	UIPinchGestureRecognizer
Panning or dragging	UIPanGestureRecognizer
Swiping (in any direction)	UISwipeGestureRecognizer
Rotating (fingers moving in opposite directions)	UIRotationGestureRecognizer
Long press (also known as “touch and hold”)	UILongPressGestureRecognizer

你的应用程序应该只以用户期待的方式来响应。 比如，一个捏合动作应该缩放，而轻击动作应该选择一些东西。 关于正确使

用手势的指南，请看iOS Human Interface Guidelines 中的 “Apps Respond to Gestures, Not Clicks”。

2、手势识别连接到视图

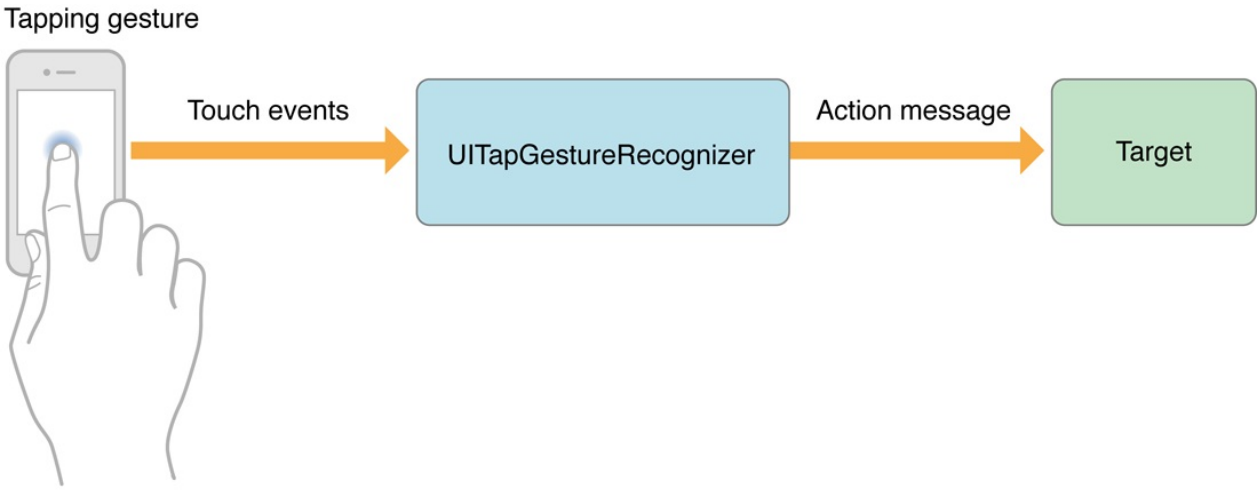
每个手势识别都跟一个视图相关联。 通常，视图可以有多个手势识别，因为单个视图可能响应多个不同的手势。 要想一个手势识别能够识别在一个特殊视图上发生的各种触摸，你必须把手势识别连接到那个视图。 当用户触摸了那个视图，手势识别先于视图对象接收一个触摸发生的信息。 结果，手势识别可以通过视图的行为来响应各种触摸。

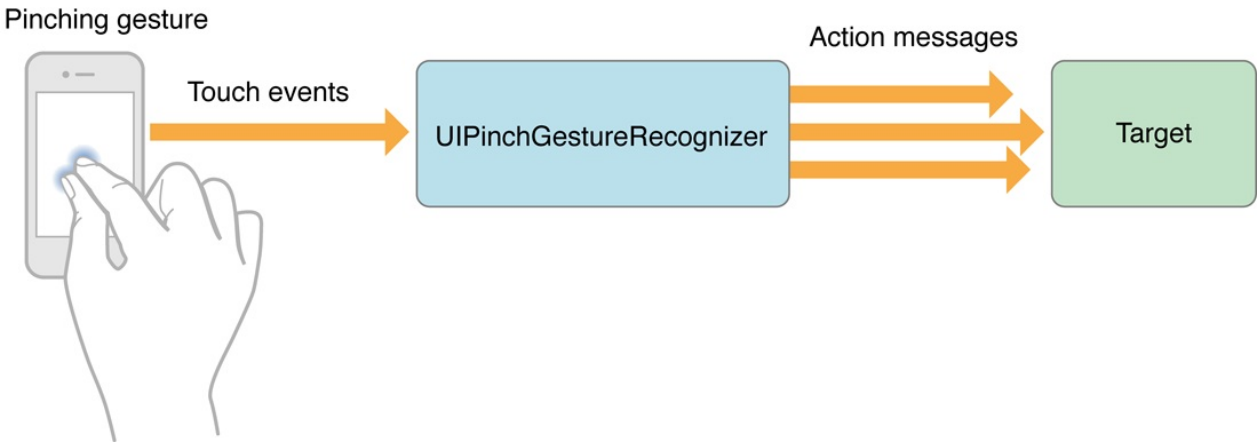
3、 手势触发操作消息

当一个手势识别识别出其制定的手势后，它给它的目标发送一个操作消息。 要想创建一个手势识别，你需要初始化一个目标和一个操作。

1). 离散和连续的手势

手势有离散手势，也有连续手势。 一个离散手势，比如一个轻击(tap)，只发生一次。一个连续手势，比如捏合(pinching)，发生时持续一段时间。 对于离散手势，手势识别就给目标发送一个操作消息。对于连续手势，手势识别则保持发送操作消息给目标直到多点触摸序列结束，如图1-2.





二、用手势识别响应事件

你需要做三件事来添加一个内建手势识别到应用程序：

1. 创建并配置一个手势识别实例. 该步骤包括分配一个目标，操作，以及有时候分配手势指定的各种属性(比如numberOfTapsRequired).
2. 把手势识别连接到一个视图。
3. 实现处理手势的操作方法。

1、使用界面生成器(Interface Builder)来添加一个手势识别到应用程序

在Xcode里的界面生成器中，添加手势识别到应用程序跟你添加任何对象到用户界面(add any object to your user interface)

是一样的---就是从对象库中拖拉手势识别到一个视图。完成该操作后，手势识别自动连接到那个视图。 你可以检查手势识别被连接到了哪个视图，并且如果必要，在nib 文件中更改该连接(change the connection in the nib file)。

当你创建完手势识别对象后，你需要创建并连接一个操作方法(create and connect an action)。任何时候手势识别(gesture recognizer)识别手势时都将调用该方法。如果你需要在该操作方法外面引用手势识别，你还应该为手势识别创建并连接一个输出口 (create and connect an outlet)。你的代码应该跟列表1-1相似.

```
@interface APLGestureRecognizerViewController ()
@property (nonatomic, strong) IBOutlet UITapGestureRecognizer *tapRecognizer;
@end

@implementation
- (IBAction)displayGestureForTapRecognizer:(UITapGestureRecognizer *)recognizer
    // Will implement method later...
}
@end
```

2、通过程序添加一个手势识别

你可以通过分配和初始化一个具体(concrete)的UIGestureRecognizer 子类的实例来创建一个手势识别，比如UIPinchGestureRecognizer 。 当你初始化手势识别时，指定一个目标对象和一个操作选择器(selector)，正如列表1-2中所示，通常目标对象就是视图的视图控制器。

如果你通过程序创建一个手势识别，你需要调用addGestureRecognizer: 方法来把它连接到视图。 列表1-2 创建了一个单个轻击手势识别，指定识别该手势需要一次轻击(tap)，然后把手势识别对象连接到一个视图。 通常，你在视图控制器的viewDidLoad 方法里创建手势识别，如列表1-2所示。

列表1-2 通过程序创建一个单一的轻击手势识别

```
- (void)viewDidLoad {
    [super viewDidLoad];

    // Create and initialize a tap gesture
    UITapGestureRecognizer *tapRecognizer = [[UITapGestureRecognizer alloc]
        initWithTarget:self action:@selector(respondToTapGesture:)];

    // Specify that the gesture must be a single tap
    tapRecognizer.numberOfTapsRequired = 1;

    // Add the tap gesture recognizer to the view
    [self.view addGestureRecognizer:tapRecognizer];

    // Do any additional setup after loading the view, typically from a nib
}
```

3、响应离散手势

当你创建一个手势识别时，你把该识别(recognizer)连接到一个操作方法。使用该操作方法来响应手势识别的手势。列表1-3提供了一个响应一个离散手势的例子。 当用户轻击了带有手势识别的视图时，视图控制器显示一个图像视图(image view)表示"Tap"发生。 showGestureForTapRecognizer: 方法决定了视图中手势的位置，该位置从recognizer的locationInView: 特性中获取，然后把图片显示到该位置。

注意： 接下来的3段代码例子取自 Simple Gesture Recognizers 样本代码工程，你可以查看它以获得更多上下文。

列表1-3 处理一个双击手势

```
- (IBAction)showGestureForTapRecognizer:(UITapGestureRecognizer *)recognizer {
      // Get the location of the gesture
     CGPoint location = [recognizer locationInView:self.view];

      // Display an image view at that location
     [self drawImageForGestureRecognizer:recognizer atPoint:location];

      // Animate the image view so that it fades out
     [UIView animateWithDuration:0.5 animations:^{
         self.imageView.alpha = 0.0;
     }];
}
```

每个手势识别都有它自己的特性集。比如，如列表1-4中，showGestureForSwipeRecognizer: 方法使用点击(swipe)手势识别的direction 特性来决定用户是向左清扫还是向右。 然后，它使用该值来让图像从点击方向消失。

列表1-4 响应一个向左或向右清扫手势

```
// Respond to a swipe gesture
- (IBAction)showGestureForSwipeRecognizer:(UISwipeGestureRecognizer *)recognizer {
      // Get the location of the gesture
      CGPoint location = [recognizer locationInView:self.view];

      // Display an image view at that location
      [self drawImageForGestureRecognizer:recognizer atPoint:location];

      // If gesture is a left swipe, specify an end location
      // to the left of the current location
      if (recognizer.direction == UISwipeGestureRecognizerDirectionLeft) {
          location.x -= 220.0;
      } else {
          location.x += 220.0;
      }

      // Animate the image view in the direction of the swipe as it fades out
      [UIView animateWithDuration:0.5 animations:^{
          self.imageView.alpha = 0.0;
          self.imageView.center = location;
      }];
}
```

4、响应连续手势

连续手势允许应用程序在手势发生时响应。 比如，用户可以一边做捏合手势，一边就可以看见缩放，或者允许用户沿着屏幕拖拉一个对象。

列表1-5 以跟手势同样的角度显示一个"Rotate"图片，并且当用户停止旋转时， 动画该图片让其在旋转回水平方向的同时消失。 当用户旋转他的手指时，持续调用 showGestureForRotationRecognizer: 方法直到两个手指都拿起。

列表1-5 响应一个旋转手势

```
// Respond to a rotation gesture
- (IBAction)showGestureForRotationRecognizer:(UIRotationGestureRecognizer *)recognizer {
      // Get the location of the gesture
      CGPoint location = [recognizer locationInView:self.view];

      // Set the rotation angle of the image view to
      // match the rotation of the gesture
      CGAffineTransform transform = CGAffineTransformMakeRotation([recognizer rotation]);
      self.imageView.transform = transform;

      // Display an image view at that location
      [self drawImageForGestureRecognizer:recognizer atPoint:location];

     // If the gesture has ended or is canceled, begin the animation
     // back to horizontal and fade out
     if (([recognizer state] == UIGestureRecognizerStateEnded) || ([recognizer state] == UIGestureRecognizerStateCancelled)) {
         [UIView animateWithDuration:0.5 animations:^{
             self.imageView.alpha = 0.0;
             self.imageView.transform = CGAffineTransformIdentity;
         }];
     }

}
```

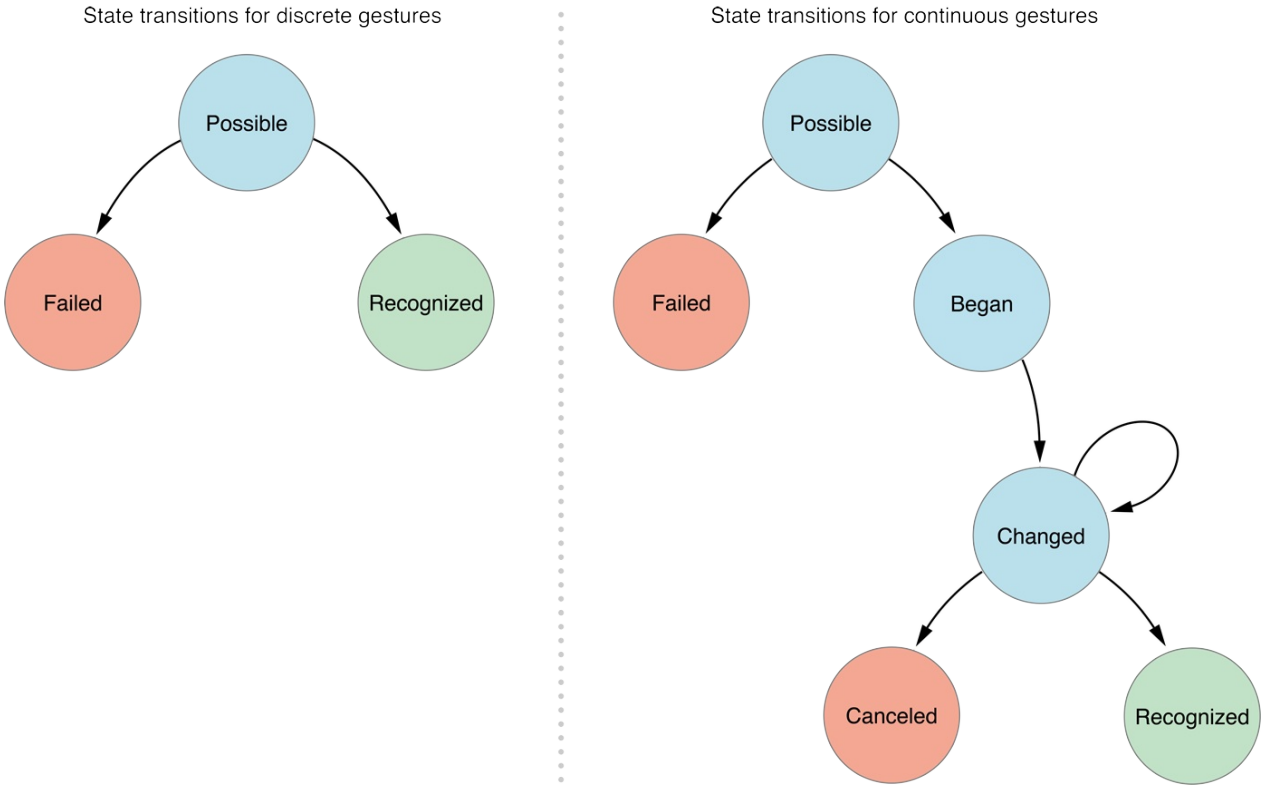
每次调用该方法时，图片都在drawImageForGestureRecognizer: 方法中被设置为不透明。 当手势完成时，图片在animateWithDuration: 方法中被设置为透明。showGestureForRotationRecognizer:方法通过检查手势识别的状态来判断手势是否完成。 关于这些状态的更多信息，请看“Gesture Recognizers Operate in a Finite State Machine.”

三、定义手势识别如何交互

通常情况下，当你把手势识别添加到应用程序时，你需要了解你想要你的识别(recognizers)之间如何发生交互，或者跟应用程序的其它任何触摸事件处理代码如何发生交互。要想完成这些，你首先需要理解更多点关于手势识别如何工作的知识。

1、手势识别在一个有限状态机里操作

手势识别以一种预定义的方式从一个状态过渡到另一个状态。 每种状态都可以根据它们遇到的特定条件(conditions)过渡到几种可能的未来状态中的一种。确切的状态机根据手势识别是离散或是连续会发生变化，正如图1-3中所示。所有的手势识别在一个可能的状态(UIGestureRecognizerStatePossible)中开始， 它们分析接收到的任何多点触摸序列，并且在分析过程中成功识别手势或者识别识别一个手势。失败识别手势意味着手势识别过渡到失败状态 (UIGestureRecognizerStateFailed).



当一个离散手势识别识别出它的手势，手势识别从可能状态过渡到被识别状态 (UIGestureRecognizerStateRecognized)，然后识别完成。

对于连续手势，当手势第一次被识别时，手势识别从可能状态开始(UIGestureRecognizerStateBegan)。然后，它从开始状态过渡到改变状态(UIGestureRecognizerStateChanged), 然后当手势发生时又从改变状态变为改变状态。 当用户的最后一个手指从视图上举起时，手势识别过渡到结束状态(UIGestureRecognizerStateEnded), 识别完成。 注意结束状态是识别完成状态的一个别名。

如果一个连续手势不再适应预期的模式时，它的识别还可以从改变状态过渡到取消状态(UIGestureRecognizerStateCancelled)。

每次手势识别改变状态时，手势识别都给它的目标发送一个操作消息，除非它过渡到失败或取消状态。 然而，一个离散手势识别只在它从可能状态过渡到被识别状态时才发送一个单个操作消息。一个连续手势识别在状态发生改变时发送多个操作消息。

当一个手势识别到达被识别(或结束)状态时，它把状态重置为可能(Possible)状态。把状态重置为可能状态不会触发一个操作消息。

2、跟其它手势识别发生交互

一个视图可以带有多个手势。使用视图的gestureRecognizers 特性来确定视图都带有哪些手势识别。 你还可以分别(respectively)使用addGestureRecognizer: 方法和removeGestureRecognizer: 方法添加和删除视图上的手势识别来动态改变视图如何处理手势。

当视图带有多个手势识别时，你可能想要改变竞争(competing)手势识别是如何接收和分析触摸事件。默认情况下，没有设置哪个手势识别首先接收到第一个触摸，因此每次触摸都可以以不同的顺序传送给手势识别。 你可以重写该默认行为来：

指定一个手势识别应该比另一个手势识别优先分析某个触摸。

允许两个手势识别来同时操作。

阻止某个手势识别分析某个触摸

使用UIGestureRecognizer 类方法，委托方法，以及其子类重写的方法来影响这些行为。

1）为两个手势识别声明一个特定顺序

想象一下，你想要识别一个快速滑动(swipe)和一个慢速拖动(pan)手势，你想要用这两个手势触发不同的操作。默认情况下，当用户尝试swipe时，该手势会被理解为一个pan。 这是因为swipe(一个离散手势)手势在满足各种必要条件被理解为一个swipe手势之前，首先满足pan(一个连续手势)手势的各种必要条件。

要想视图同时识别swipes 和pans，你想要swipe手势识别在pan手势之前来分析触摸事件。 如果swipe手势识别已经确定某个触摸是一个swipe， 那么pan手势识别就绝没有必要再去分析该触摸。 如果swipe手势识别确定该触摸不是一个swipe， 它过渡到Failed状态，然后pan手势识别应该开始分析该触摸事件。

你可以通过调用手势识别的requireGestureRecognizerToFail: 方法来说明两个手势识别之间这种类型的关系，如列表1-6所示。在该列表中，两个手势识别都被连接到了相同的视图上。

列表1-6 pan手势识别要求swipe手势识别到达fail状态

```
- (void)viewDidLoad {
    [super viewDidLoad];
    // Do any additional setup after loading the view, typically from a nib
    [self.panRecognizer requireGestureRecognizerToFail:self.swipeRecognizer];
}
```

requireGestureRecognizerToFail: 方法给接收者发送了一个消息，并且指定了一些otherGestureRecognizer，它们必须在接收识别(receiving recognizer)开始工作之前进入Failed 状态。当它等待其它手势识别过渡到Failed状态期间，接收识别(receiving recognizer)始终处于Possible状态。如果其它手势识别失败了，receiving recognizer 开始分析触摸事件并移动到

下个状态。换句话说，如果其它手势识别过渡到Recognized 或者 Began状态，receiving recognizer就移动到Failed 状态。关于状态过渡的更多信息， 请看“Gesture Recognizers Operate in a Finite State Machine.”

注意：如果你的应用程序识别同时支持单击和双击，并且你的单击手势识别不要求双击识别失败，那么即使是用户双击时，你也应该期待在双击操作之前接收单击操作。该行为是有意设计的，因为最好的用户体验通常都开启多种类型的操作。

如果你想要这两种操作不兼容，你的单击识别必须要求双击识别进入失败状态。 然而，这样你的单击操作会滞后用户的输入，因为单击识别会等到双击识别失败后才开始识别。

2）阻止手势识别分析触摸

你可以通过添加一个委托对象到手势识别来改变手势识别的行为。UIGestureRecognizerDelegate 协议提供了一组方法来阻止手势识别分析触摸。 你可以选择使用协议中的 gestureRecognizer:shouldReceiveTouch: 方法 和 gestureRecognizerShouldBegin: 方法中的一个来使用。

当一个触摸开始时，如果你可以立即确定手势识别是否应该考虑该触摸，使用 gestureRecognizer:shouldReceiveTouch: 方法来实现。每次有新触摸时都调用该方法。 阻止手势识别注意到一个触摸的发生，请返回NO。默认值是YES。该方法不改变手势识别的状态。

列表1-7 使用 gestureRecognizer:shouldReceiveTouch: 委托方法来阻止手势识别接收到来自一个自定义子视图中发生的触摸。

```
- (void)viewDidLoad {
    [super viewDidLoad];
    // Add the delegate to the tap gesture recognizer
    self.tapGestureRecognizer.delegate = self;
}

// Implement the UIGestureRecognizerDelegate method
-(BOOL)gestureRecognizer:(UIGestureRecognizer *)gestureRecognizer shouldReceiveTouch:(UITouch *)touch {
    // Determine if the touch is inside the custom subview
    if ([touch view] == self.customSubview){
        // If it is, prevent all of the delegate's gesture recognizers
        // from receiving the touch
        return NO;
    }
    return YES;
}
```

如果你需要在确定手势识别是否应该分析一个触摸之前一直等待。使用 gestureRecognizerShouldBegin: 委托方法。 通常，如果你有一个UIView 或 UIControl子类并带有跟手势识别想冲突的自定义触摸事件处理，你可以使用该方法。返回NO，让手势识别立即进入失败状态，允许其他触摸处理来处理。 当手势识别想要过渡到Possible状态以外的状态时，如果手势识别将阻止一个视图或控件接收一个触摸，该方法被调用。

如果你的视图或视图控制器不能成为手势识别委托，你可以使用UIView的fgestureRecognizerShouldBegin:方法 。该方法的签名和实现是一样的。

3）开启同时手势识别

默认情况下，两个手势识别不能同时识别它们的不同手势。但是，假设你想让用户可以同时捏合并旋转一个视图，你需要改变默认行为，你可以通过调用gestureRecognizer:shouldRecognizeSimultaneouslyWithGestureRecognizer: 方法来实现。该方法是 UIGestureRecognizerDelegate 协议的一个可选方法。 当一个手势识别的手势分析可能阻碍另一个手势识别识别它的手势时可以调用该方法，反之亦然。 该方法默认范围NO。当你想让两个手势同时分析它们的手势时，返回YES。

注意：你只在一个手势识别需要开启同时识别功能时才需要实现一个委托并返回YES。 然而，它还意味着返回NO并不一定阻止同时识别功能，因为其他手势识别的委托可能返回了YES。

4）给两个手势指定一个单向关系

如果你想要控制两个识别(recognizers)是如何交互，但是你需要指定一个单向关系，你可以重写 canPreventGestureRecognizer: 或 canBePreventedByGestureRecognizer: 子类方法并返回NO(默认为YES)。 比如，如果你想用一个旋转手势阻止一个捏合手势，但是你又不想捏合手势阻止一个旋转手势，你可以用如下语句指定。

```
[rotationGestureRecognizer canPreventGestureRecognizer:pinchGestureRecognizer];
```

同时，重写旋转手势识别的子类方法来返回NO. 更多管理如何子类化 UIGestureRecognizer的信息，请看 “Creating a Custom Gesture Recognizer.”

如果没有手势需要阻止另外手势，使用gestureRecognizer:shouldRecognizeSimultaneouslyWithGestureRecognizer:，它在“Permitting Simultaneous Gesture Recognition.” 里描述。 默认情况下，捏合手势阻止旋转手势，或者旋转手势阻止捏合手势，因为两个手势不能同时被识别。

5) 跟别的用户界面控件交互

在iOS 6.0 或以后版本中，默认控件操作方法防止(prevent)重复手势识别的行为。比如，一个按钮的默认操作是一个单击。如果你有一个单击手势识别绑定到一个按钮的父视图上，然后用户点击该按钮，最后按钮的操作方法接收触摸事件而不是手势识别。 它只用于手势识别跟一个控件的默认操作重复时，包括：

单个手指在UIButton, UISwitch, UIStepper, UISegmentedControl, and UIPageControl 上的单击。

单个手势在 UISlider 上的快速滑动(swipe)，轻扫方向跟slider平行

单个手指的在UISwitch 控件上的慢速拖动(pan)手势，方向跟switch平行。

如果你有一个这些控件的自定义子类，你想要改变其默认操作，把手势识别直接连接到控件而不是连接到其父视图。然后，手势识别首先接收到触摸事件。一如往常，请确保你已经阅读了 iOS Human Interface Guidelines 文档以确保你的应用程序提供了一个直观的用户体验，特别是当你重写一个标准控件的默认行为时。

四、手势识别解读(interpret)原始触摸事件

目前位置，你已经学习了关于手势以及应用程序如何识别并响应它们。 然而，要想创建一个自定义手势识别或想控制时手势别如何跟视图的触摸事件处理相交互，你需要更具体地(specifically)思考触摸和事件的方方面面。

1、一个事件包含了单前多点触摸序列的所有触摸

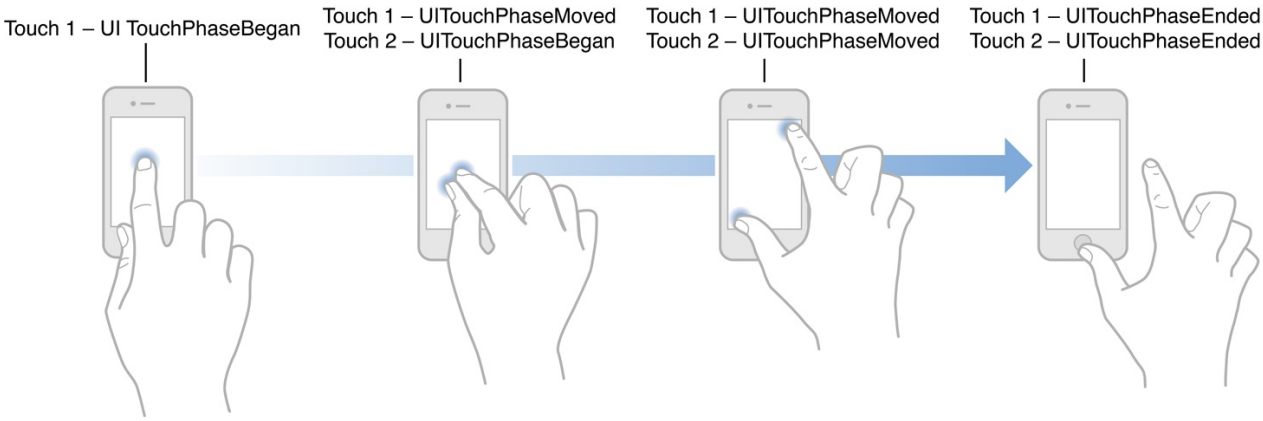
在iOS中，一个触摸是一个手指在屏幕上的存在或运动。 一个手势有一个或多个触摸，它由UITouch 对象表示。比如，一个pinch-close手势有两个触摸--两个手指在屏幕上从相反方向朝着彼此移动。

一个事件包含(encompasses)一个多点触摸序列的所有触摸。 一个多点触摸序列以一个手指触摸屏幕开始，以最后一个手指离开屏幕结束。 当一个手指移动时，iOS给事件触摸对象。一个多点触摸事件由 UIEventTypeTouches 类型的UIEvent 对象表示。

每个触摸对象只跟踪一个手指，并且只在触摸序列期间跟踪。 在序列期间，UIKit跟踪手指并更新触摸对象的各种属性。 这些属性包括触摸阶段(phase)，它在视图中的位置，它的前一个位置，以及它的时间戳。

触摸阶段表明一个触摸何时开始，它是移动的还是静止的，以及它何时结束---当手指不再触摸屏幕的时间。 正如图1-4所示，应用程序在任何触摸的每个阶段之间接收事件对象。

图1-4 一个多点触摸序列和触摸阶段



注意：手指没有鼠标点击精确。 当用户触摸屏幕时，接触的区域实际上是椭圆的，并且会比用户期待的位置稍微篇低。 该接触面会根据手指的尺寸和方向，手指使用时的压力，以及其它因素的不同而发生改变。底层多点触摸系统会替你分析该信息并计算一个单击点，因此你不需要自己写代码来实现它。

2、 应用程序在触摸处理方法中接收触摸

在一个多点触摸序列期间， 应用程序在新触摸发生或者给出的触摸阶段发生改变时发送这些信息；它调用以下方法：

当一个或多个手指触摸屏幕时调用

当一个或多个手势移动时调用

当一个或多个手指离开屏幕时调用

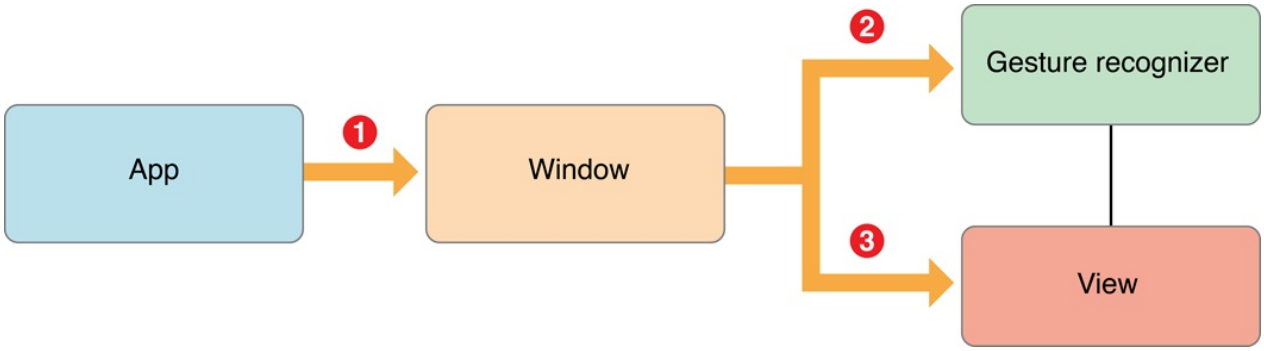
当触摸序列被系统事件取消时调用，比如有一个来电。

每个方法都跟一个触摸阶段相关联；比如，touchesBegan: 方法跟 UITouchPhaseBegan 方法相关联。 触摸对象的阶段(phase)被存储在其phase 特性里。

注意： 这些方法跟手势识别状态没有关联，比如UIGestureRecognizerStateBegan 和 UIGestureRecognizerStateEnded 等。 手势识别器状态严格表示手势识别器自身的阶段，不表示正在被识别的触摸对象阶段。

五、 调节触摸到视图的传递

可能有时候你想要在手势识别器之前接收到一个触摸。但是在你可以改变触摸到视图的传递路径之前，你需要理解其默认行为。在简单情况下，当一个触摸发生时，触摸对象从UIApplication对象传递到UIWindow对象。 然后，窗口首先把触摸发送给触摸发生的视图上关联的任何手势识别器，而不是先发送给视图对象自身。



1、手势识别器首先识别一个触摸

窗口延迟把触摸对象传递给视图，这样手势识别器就可以首先分析触摸。延迟期间，如果手势识别器识别出一个触摸手势，然后窗口就绝不会再把触摸对象传递给视图，并且还取消任何先前传递给视图的任何触摸对象，这些触摸对象都是被识别序列的一部分。比如，如果你有一个手势识别器用来识别一个离散手势，该手势要求一个双手指的触摸，该触摸就会被解释成两个单独的触摸对象。 当触摸发生时，触摸对象从英语程序对象传递到触摸发生视图的窗口对象，然后发生以下序列，请看图1-6。

Figure 1-6 Sequence of messages for touches

图1-6 触摸消息序列

	❶	❷	❸	❹
Gesture Recognizer	touchesBegan:	touchesMoved:	touchesEnded:	touchesEnded:
tapGestureRecognizer.state	Possible	Possible	Possible	Recognized
View	touchesBegan:	touchesMoved:		touchesCancelled:

1. 窗口在Began 阶段发送两个触摸对象---通过 touchesBegan:withEvent: 方法---给手势识别器。 手势识别器还不能识别该手势，因此它的状态是Possible. 窗口发送这些同样的触摸给手势识别器相关联的视图。

1. 窗口在Moved阶段发送两个触摸对象---通过touchesMoved:withEvent: 方法---- 给手势识别器。 识别器任然不能侦测该手势，状态还是Possible。 窗口然后发送这些触摸到相关联的视图。

1. 窗口在Ended阶段发送一个触摸对象--- 通过touchesEnded:withEvent: 方法---给手势识别器。 虽然该触摸对象对于手势来说信息还不够，但是窗口还是把该对象扣住(withhold)不发送给视图。

1. 窗口在Ended阶段发送另一个触摸。 手势识别器这是可以识别出该手势，因此把状态设置为Recognized. 就在第一个操作信息被发送之前，视图调用touchesCancelled:withEvent: 方法来使先前在Began 和Moved阶段发送的触摸对象无效(invalidate)。触摸在Ended阶段被取消。

现在假设手势识别器在最后一步确定它正在分析的多点触摸序列不是它的手势。它把状态设置为UIGestureRecognizerStateFailed.。 然后窗口在Ended阶段发送这两个触摸对象给相关联的视图---通过touchesEnded:withEvent: 消息。

一个连续手势的手势识别器遵循一个相似的序列，除了它更有可能在触摸对象到达Ended 阶段之前就识别出它的手势。一旦识别出它的手势，它把状态设置为 UIGestureRecognizerStateBegan (而不是Recognized). 窗口把多点触摸序列中的所有子序列触摸对象发送给手势识别器，而不是发送到附属的(attached)视图。

2、影响到视图的各个触摸的传递

你可以改变 UIGestureRecognizer 特性的几个值来改变默认传递路径，让它们以特定的方式传递。如果你改变这些特性的默认值，以下行为将发生变化：

delaysTouchesBegan(默认为NO)---正常情况下，窗口在Began 和 Moved 阶段把触摸对象发送给视图和手势识别器。 把delaysTouchesBegan设置为YES,使得窗口不会在Began阶段把触摸对象发送给视图。 这样做确保一个手势识别器识别它的手势时，没有把部分触摸事件传递给相连的视图。 设置该特性时请谨慎，因为它会使你的界面反应迟钝。

delaysTouchesEnded(默认为YES)---当该特性被设置为YES时，它确保视图不会完成一个动作，而该动作是手势可能想在稍候取消的。当一个手势识别器正在分析一个触摸事件时，窗口不会不会在Ended阶段传递触摸对象到相连的视图。如果一个手势识别器识别出它的手势，则触摸对象被取消。 如果手势识别器没有识别出它的手势，窗口通过一个touchesEnded:withEvent:消息把这些对象传递给视图。设置该特性为NO，允许视图和手势识别器可以同时在Ended阶段分析触摸对象。

例如，设想一个视图有一个点击手势识别器，它要求双击，然后用户双击了该视图。 把特性设置为YES后，视图获得touchesBegan:withEvent:, touchesBegan:withEvent:, touchesCancelled:withEvent:, 以及 touchesCancelled:withEvent: 信息序列。如果该特性被设置为NO，视图获得以下信息序列：touchesBegan:withEvent:, touchesEnded:withEvent:, touchesBegan:withEvent:, andtouchesCancelled:withEvent:，它表示在touchesBegan:withEvent: 消息中，视图可以识别一个双击。 If a gesture recognizer detects a touch that it determines is not part of its gesture, it can pass the touch directly to its view. To do this, the gesture recognizer callsignoreTouch:forEvent: on itself, passing in the touch object.

如果一个手势识别器侦测到一个不属于该手势的触摸，它可以把该触摸直接传递给它的视图。 要想实现它，手势识别器对自己调用ignoreTouch:forEvent:方法，它在触摸对象中传递。

六、创建一个自定义手势识别器

要想实现一个自定义手势识别器，首先在Xcode里创建一个 UIGestureRecognizer 的子类。然后，在子类的头文件中加入以下import指令。

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

下一步，从UIGestureRecognizerSubclass.h中拷贝以下方法声明到你的头文件；这些是你在子类中需要重写的方法。

```
- (void)reset;
- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event;
- (void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event;
- (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event;
- (void)touchesCancelled:(NSSet *)touches withEvent:(UIEvent *)event;
```

这些方法跟早先在“An App Receives Touches in the Touch-Handling Methods.” 中所描述的相关触摸事件处理有完全相同的签名和行为。 在所有这些需要重写的方法中，你必须调用父类的实现，即使该方法有一个null实现。

注意：UIGestureRecognizerSubclass.h中的 state 特性目前是readwrite状态，而不是readonly,就跟它在UIGestureRecognizer.h中一样。 你的子类可以通过给特性分配 UIGestureRecognizerState 常量(constants)来改变其状态。

1、为自定义手势识别器实现触摸事件处理方法

一个自定义手势识别器实现的核心是四个方法： touchesBegan:withEvent:, touchesMoved:withEvent:,touchesEnded:withEvent:, and touchesCancelled:withEvent:. 在这些方法中，你通过设置一个手势识别器的状态，把低层触摸事件解析为手势识别。列表1-8创建了一个离散单击勾选(checkmark)手势的手势识别器。 它记录了手势的中心点---即勾的上升开始点---这样客户就可以获取这个值。

该例子只有一个单一视图，但是大多是应用程序都有多个视图。一般来说，你应该把触摸位置转换为屏幕的坐标系，这样你就可以正确的识别出跨越(span)多个视图的手势。

列表1-8 一个勾(checkmark)手势识别器的实现

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

// Implemented in your custom subclass
- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event {
   [super touchesBegan:touches withEvent:event];
   if ([touches count] != 1) {
      self.state = UIGestureRecognizerStateFailed;
      return;
   }
}

- (void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event {
   [super touchesMoved:touches withEvent:event];
   if (self.state == UIGestureRecognizerStateFailed) return;
   UIWindow *win = [self.view window];
   CGPoint nowPoint = [touches.anyObject locationInView:win];
   CGPoint nowPoint = [touches.anyObject locationInView:self.view];
   CGPoint prevPoint = [touches.anyObject previousLocationInView:self.view];

   // strokeUp is a property
   if (!self.strokeUp) {
      // On downstroke, both x and y increase in positive direction
      if (nowPoint.x >= prevPoint.x && nowPoint.y >= prevPoint.y) {
         self.midPoint = nowPoint;
         // Upstroke has increasing x value but decreasing y value
      } else if (nowPoint.x >= prevPoint.x && nowPoint.y <= prevPoint.y) {
         self.strokeUp = YES;
      } else {
         self.state = UIGestureRecognizerStateFailed;
      }
   }
}

- (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event {
   [super touchesEnded:touches withEvent:event];
   if ((self.state == UIGestureRecognizerStatePossible) && self.strokeUp) {
      self.state = UIGestureRecognizerStateRecognized;
   }
}

- (void)touchesCancelled:(NSSet *)touches withEvent:(UIEvent *)event {
   [super touchesCancelled:touches withEvent:event];
   self.midPoint = CGPointZero;
   self.strokeUp = NO;
   self.state = UIGestureRecognizerStateFailed;
}
```

离散手势和连续手势的状态过渡是不一样的，正如在 “Gesture Recognizers Operate in a Finite State Machine.” 中所描述。当你创建一个自定义手势识别器时，你通过给它分配响应的状态来表明(indicate)它是离散手势或是连续手势。比如，列表1-8 中的复选标记(checkmark)手势识别器，它不会把状态设置为Began 或者 Changed，因为它是离散手势。

当你子类化一个手势识别器时，最重要的事情是正确地设置手势识别器的state。 为了手势识别器的的交互如预期，iOS需要了解一个手势识别器的状态。比如，如果你想要实现同时识别或者要求一个手势识别器失败，iOS需要了解识别器的当前状态。

关于创建自定义手势识别器的更多信息，请看 WWDC 2012: Building Advanced Gesture Recognizers.

1. 重置手势识别器的状态

如果你的手势识别器过渡到Recognized/Ended, Canceled, 或者Failed状态，UIGestureRecognizer 类刚好在手势识别器过渡回Possible状态前调用reset 方法。

实现reset 方法来重置任何内部状态，这样你的识别器能准备进行识别手势的一个新的尝试，正如列表1-9所示。当手势识别器从该方法返回后，它将不再接收任何在进行的触摸更新。

列表1-9 重置一个手势识别器

```
- (void)reset {
    [super reset];
    self.midPoint = CGPointZero;
    self.strokeUp = NO;
}
```

UIScrollView详解

UITableView的父类是UIScrollView。当然我们要实现一个TableView也需要继承自UIScrollView，那么我们就需要看一下UIScrollView的一些属性和方法。

可能你很难相信，UIScrollView和一个标准的UIView差异并不大，scroll view确实会多一些方法，但这些方法只是UIView一些属性的表面而已。因此，要想弄懂UIScrollView是怎么工作之前，你需要了解 UIView，特别是视图渲染过程的两步。

光栅化和组合

渲染过程的第一部分是众所周知的光栅化，光栅化简单的说就是产生一组绘图指令并且生成一张图片。比如绘制一个圆角矩形、带图片、标题居中的UIButtons。这些图片并没有被绘制到屏幕上去；取而代之的是，他们被自己的视图保持着留到下一个步骤用。

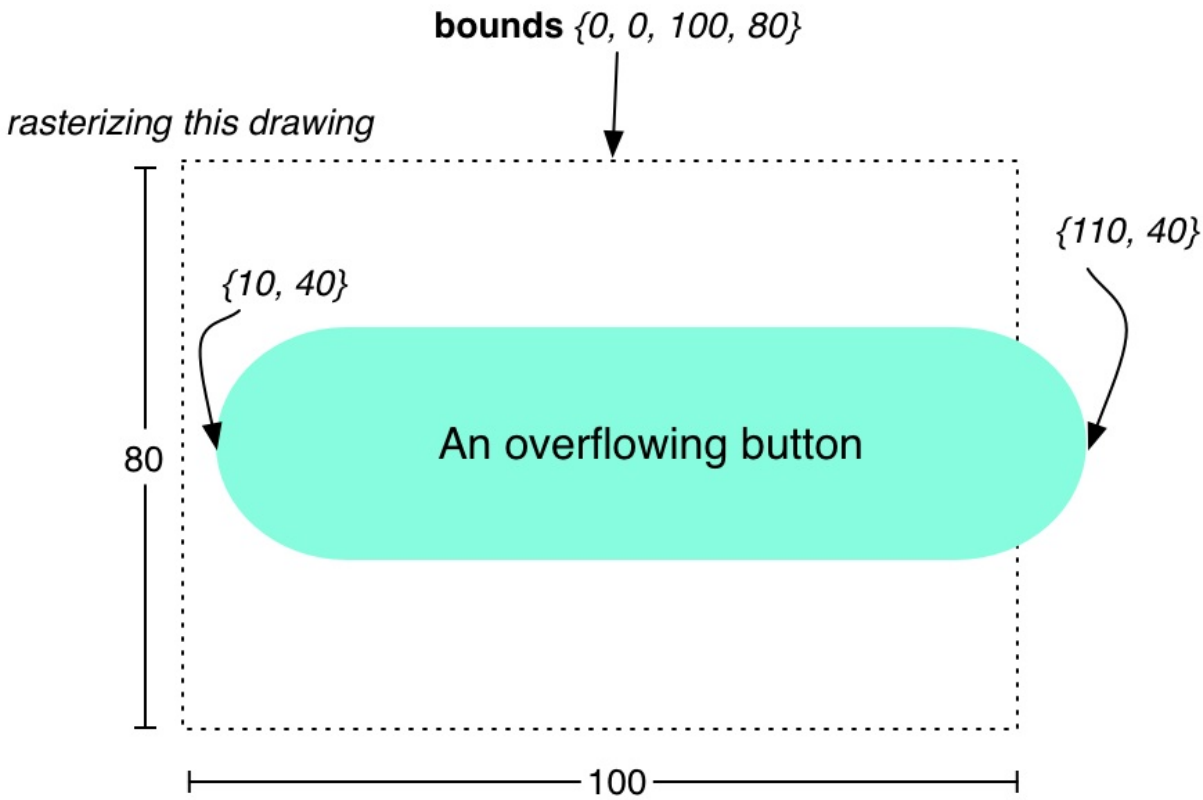
一旦每个视图都产生了自己的光栅化图片，这些图片便被一个接一个的绘制，并产生一个屏幕大小的图片，这便是上文所说的组合。视图层级(view hierarchy)对于组合如何进行扮演了很重要的角色：一个视图的图片被组合在它父视图图片的上面。然后，组合好的图片被组合到父视图的父视图图片上 面，就这样。。。最终视图层级最顶端是窗口(window)，它组合好的图片便是我们看到的东西了。

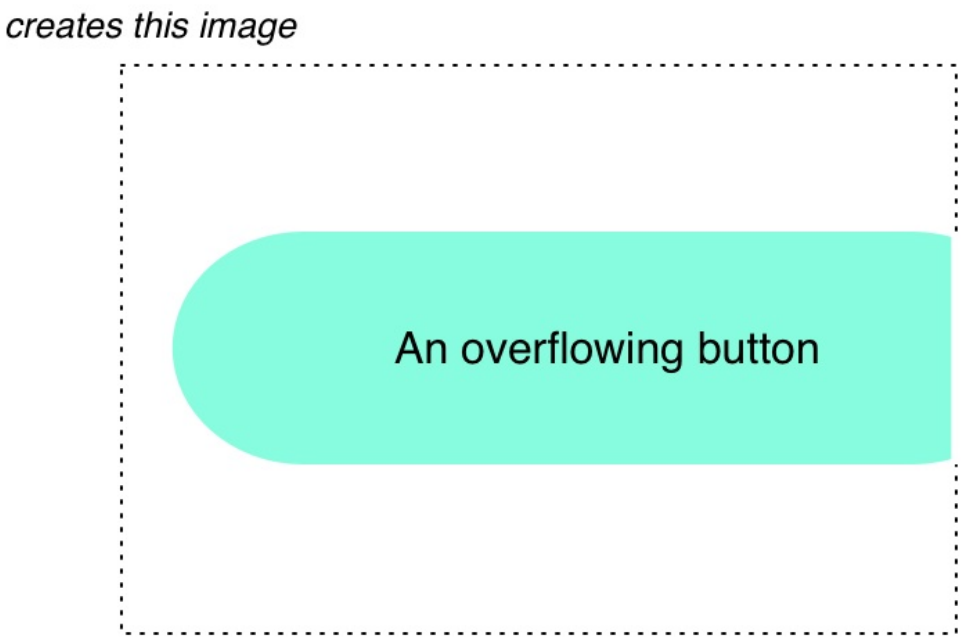
概念上，依次在每个视图上放置独立分层的图片并最终产生一个图片，单调的图像将会变得更容易理解，特别是如果你以前使用过像Photoshop这样的工具。我们还有另外一篇文章详细解释了像素是如何绘制到屏幕上去的。

现在，回想一下，每个视图都有一个bounds和frame。当布局一个界面时，我们需要处理视图的frame。这允许我们放置并设置视图的大小。 视图的frame和bounds的大小总是一样的，但是他们的origin有可能不同。弄懂这两个工作原理是理解UIScrollView的关键。

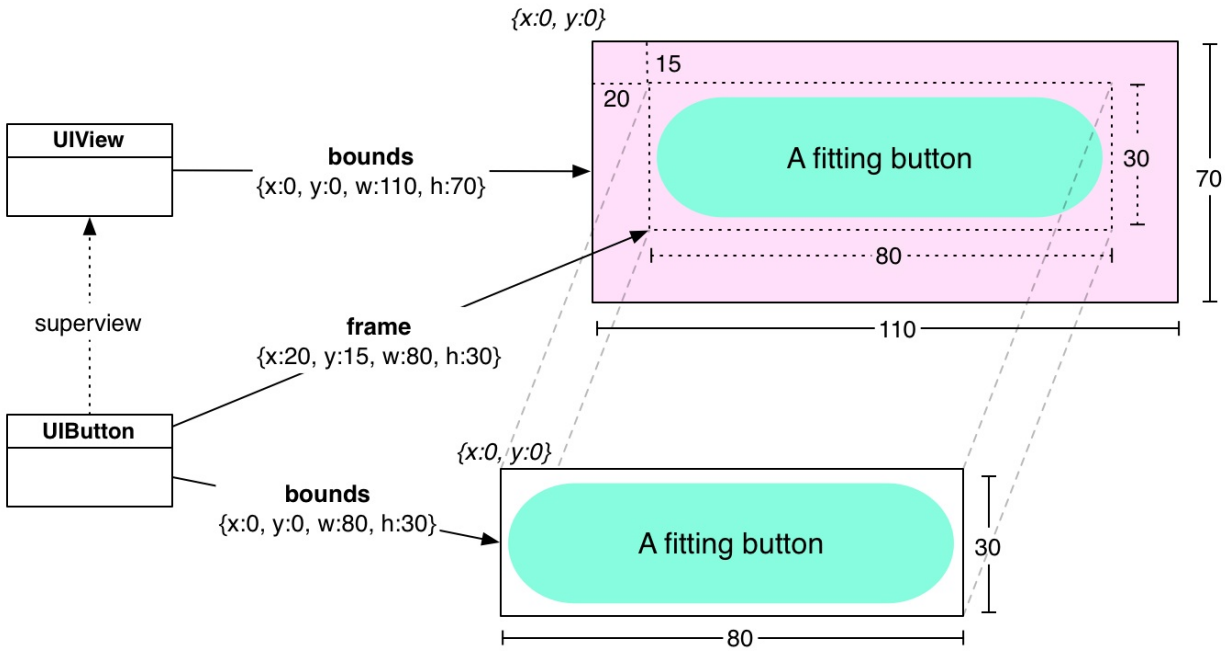
在光栅化步骤中，视图并不关心即将发生的组合步骤。也就是说，它并不关心自己的frame(这是用来放置视图的图像)或自己在视图层级中的位置(这 是决定组合的顺序)。这时视图只关心一件事就是绘制它自己的content。这个绘制发生在每个视图的drawRect:方法中。

在drawRect:方法被调用前，会为视图创建一个空白的图片来绘制content。这个图片的坐标系统是视图的bounds。几乎每个视图 bounds的origin都是{0，0}。因此，当在删格化图片左上角绘制一些东西的时候，你都会在bounds的origin({x:0,y:0}) 处绘制。在一个图片右下角的地方绘制东西的时候，你都会绘制在{x:width, y:height}处。如果你的绘制超出了视图的bounds，那么超出的部分就不属于删格化图片的部分了，并且会被丢弃。





在组合的步骤中，每个视图将自己光栅化图片组合到自己父视图的光栅化图片上面。视图的frame决定了自己在父视图中绘制的位置，frame的 origin表明了视图光栅化图片左上角相对父视图光栅化图片左上角的偏移量。所以，一个origin为{x:20,y:15}的frame所绘制的图片 左边距其父视图20点，上边距父视图15点。因为视图的frame和bounds矩形的大小总是一样的，所以光栅化图片组合的时候是像素对齐的。这确保了 光栅化图片不会被拉伸或缩小。



记住，我们才仅仅讨论了一个视图和它父视图之间的组合操作。一旦这两个视图被组合到一起，组合的结果图片将会和父视图的父视图进行组合。。。这是一个雪球效应。

考虑一下组合图片背后的公式。视图图片的左上角会根据它frame的origin进行偏移，并绘制到父视图的图片上：

```
CompositedPosition.x = View.frame.origin.x - Superview.bounds.origin.x;

CompositedPosition.y = View.frame.origin.y - Superview.bounds.origin.y;
```

我们可以通过几个不同的frames看一下：

这样做是有道理的。我们改变button的frame.origin后，它会改变自己相对紫色父视图的位置。注意，如果我们移动button直到它 的一部分已经在紫色父视图bounds的外面，当光栅化图片被截去时这部分也将会通过同样的绘制方式被截去。然而，技术上讲，因为iOS处理组合方法的原 因，你可以将一个子视图渲染在其父视图的bounds之外，但是光栅化期间的绘制不可能超出一个视图的bounds。

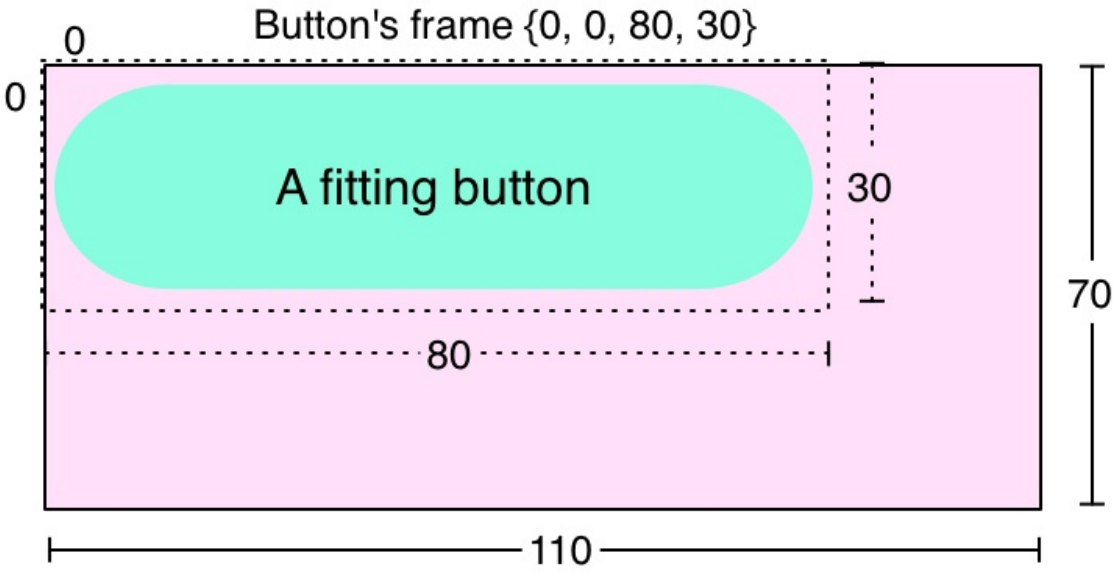
Scroll View的Content Offset

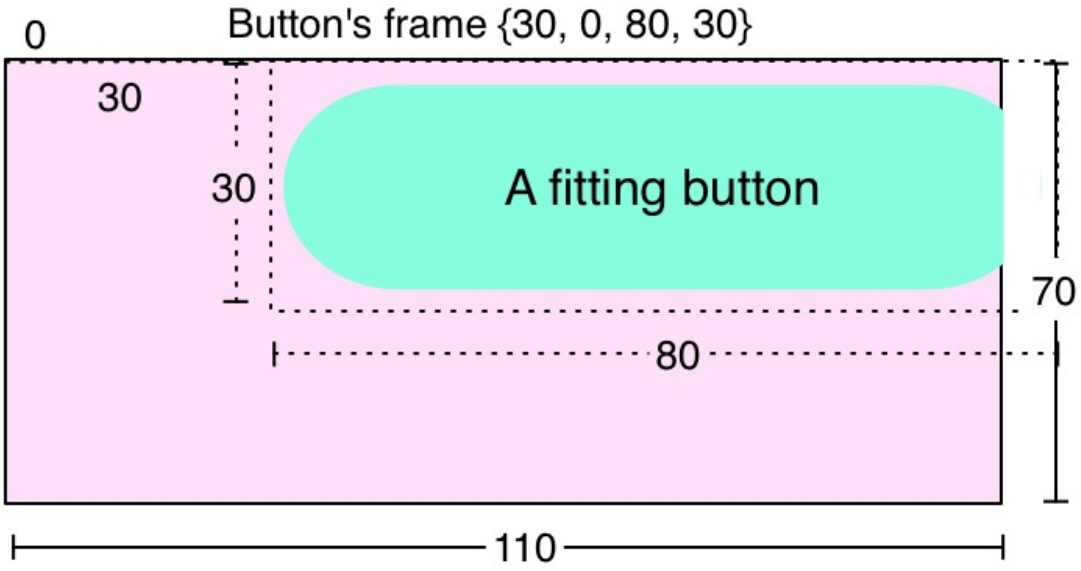
现在，我们所讲的跟UIScrollView有什么关系呢？一切都和它有关！考虑一种我们可以实现的滚动：我们有一个拖动时frame不断改变的视 图。这达到了相同的效果，对吗？如果我拖动我的手指到右边，那么拖动的同时我增大视图的origin.x，瞧，这货就是scroll view。

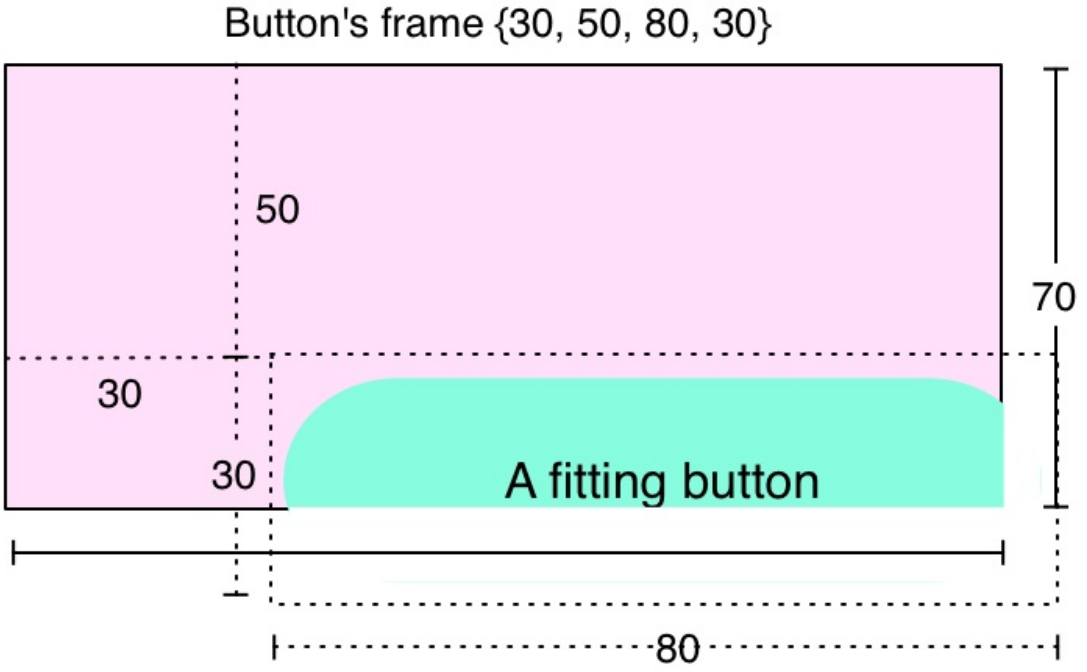
当然，在scroll view中有很多具有代表性的视图。为了实现这个平移功能，当用户移动手指时，你需要时刻改变每个视图的frames。当我们提出组合一个view的光栅化图片到它父视图什么地方时，记住这个公式：

```
CompositedPosition.x = View.frame.origin.x - Superview.bounds.origin.x;

CompositedPosition.y = View.frame.origin.y - Superview.bounds.origin.y;
```

我们减少Superview.bounds.origin的值(因为他们总是0)。但是如果他们不为0呢？我们用和前一个图例相同的frames，但是我们改变了紫色视图bounds的origin为{-30,-30}。得到下图：


Button's frame {0, 0, 80, 30}
0
0
A fitting button
30
70
80
110


Button's frame {30, 0, 80, 30}
0
30
30
A fitting button
70
80
110


Button's frame {30, 50, 80, 30}
50
70
30
30
A fitting button
80

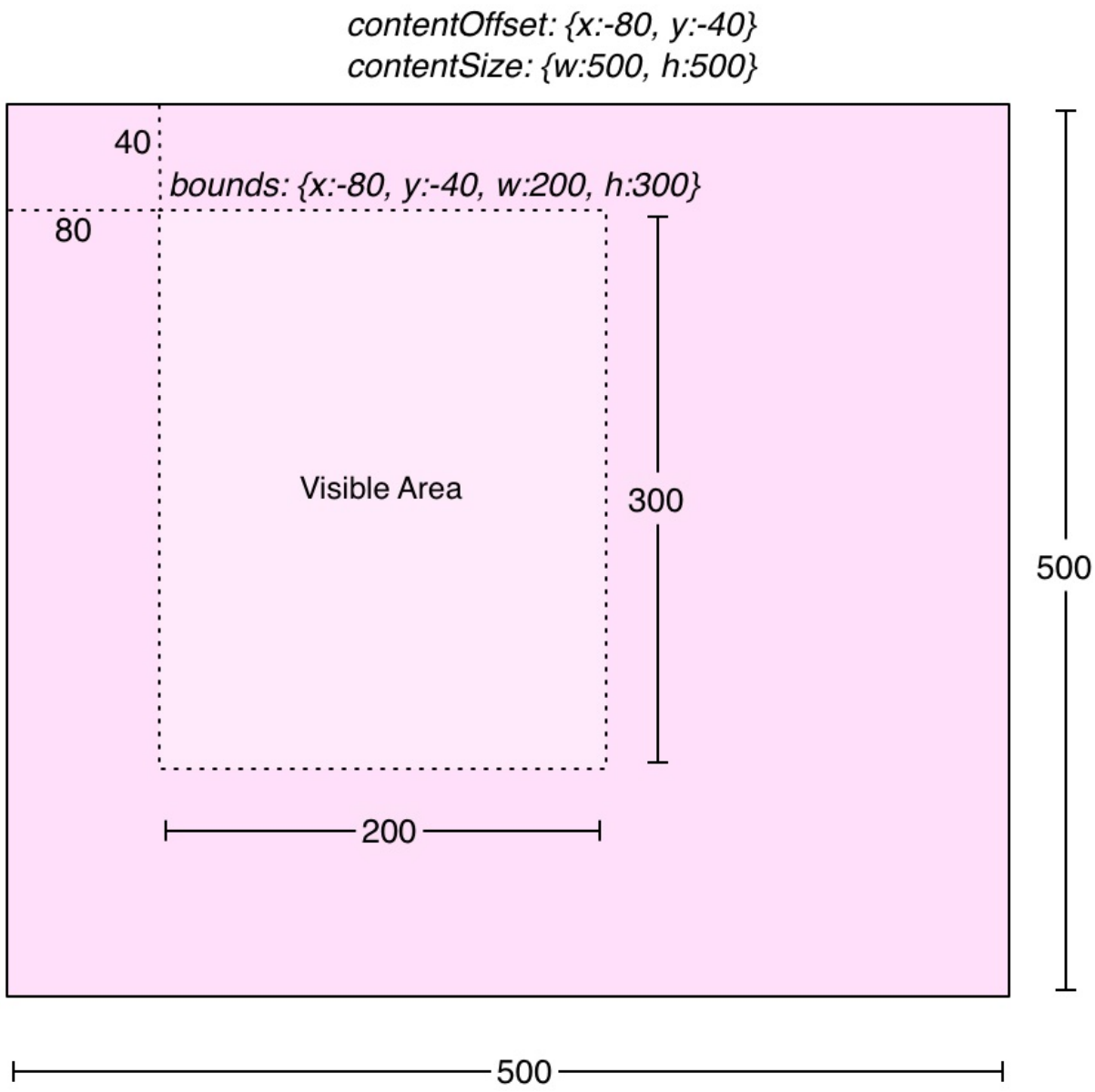
现在，巧妙的是通过改变这个紫色视图的bounds，它每一个单独的子视图都被移动了。事实上，这正是一个scroll view工作的原理。当你设置它的contentOffset属性时：它改变scroll view.bounds的origin。事实上，contentOffset甚至不是实际存在的。代码看起来像这样：

- (void)setContentOffset:(CGPoint)offset { CGRect bounds = [self bounds]; bounds.origin = offset; [self setBounds:bounds]; } 注意：前一个图例，只要足够的改变bounds的origin，button将会超出紫色视图和button组合成的图片的范围。这也是当你足够的移动scroll view时，一个视图会消失！

世界之窗：Content Size

现在，最难的部分已经过去了，我们再看看UIScrollView另一个属性：contentSize。scroll view的content size并不会改变其bounds的任何东西，所以这并不会影响scroll view如何组合自己的子视图。反而，content size定义了可滚动区域。scroll view的默认content size为{w:0,h:0}。既然没有可滚动区域，用户是不可以滚动的，但是scroll view任然会显示其bounds范围内所有的子视图。

当content size设置为比bounds大的时候，用户就可以滚动视图了。你可以认为scroll view的bounds为可滚动区域上的一个窗口：



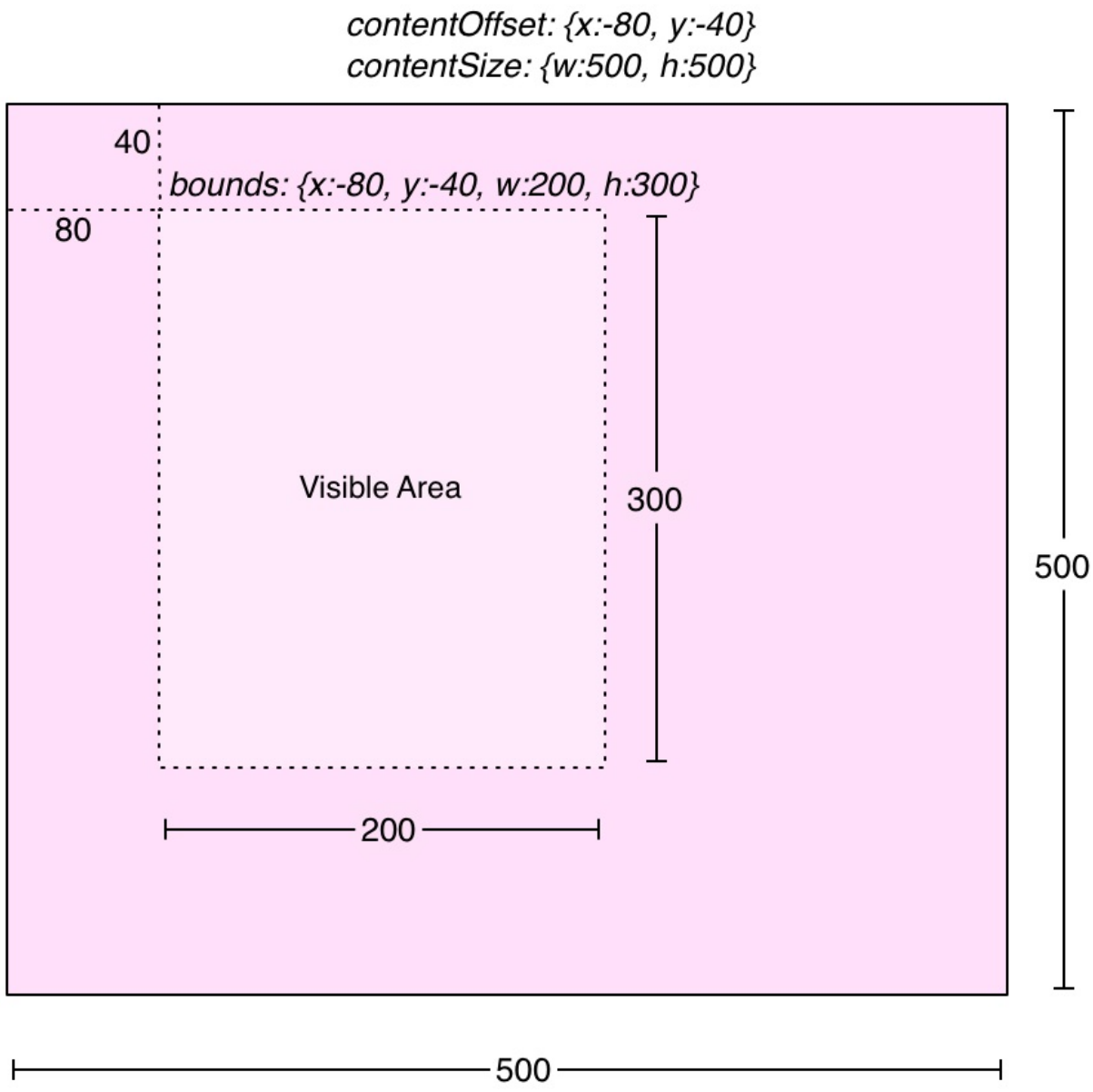
当content offset为{x:0,y:0}时，可见窗口的左上角在可滚动区域的左上角处。这也是content offset的最小值；用户不能再往可滚动区域的左边或上边移动了。那儿没啥，别滚了！

content offset的最大值是content size和scroll view size的差。这也在情理之中：从左上角一直滚动到右下角，用户停止时，滚动区域右下角边缘和滚动视图bounds的右下角边缘是齐平的。你可以像这样记 下content offset的最大值：

```
contentOffset.x = contentSize.width - bounds.size.width;
contentOffset.y = contentSize.height - bounds.size.height;
```

用Content Insets对窗口稍作调整

contentInset属性可以改变content offset的最大和最小值，这样便可以滚动出可滚动区域。它的类型为UIEdgeInsets，包含四个值：{top，left，bottom，right}。当你引进一个inset时，你改变了content offset的范围。比如，设置content inset顶部值为10，则允许content offset的y值达到10。这介绍了可滚动区域周围的填充。



这咋一看好像没什么用。实际上，为什么不仅仅增加content size呢？除非没办法，否则你需要避免改变scroll view的content size。想要知道为什么？想想一个table view（UItableView是UIScrollView的子类，所以它有所有相同的属性），table view为了适应每一个cell，它的可滚动区域是通过精心计算的。当你滚动经过table view的第一个或最后一个cell的边界时，table view将content offset弹回并复位，所以cells又一次恰到好处的紧贴scroll view的bounds。

当你想要使用UIRefreshControl实现拉动刷新时发生了什么？你不能在table view的可滚动区域内放置UIRefreshControl，否则，table view将会允许用户通过refresh control中途停止滚动，并且将refresh control的顶部弹回到视图的顶部。因此，你必须将refresh control放在可滚动区域上方。这将允许首先将content offset弹回第一行，而不是refresh control。

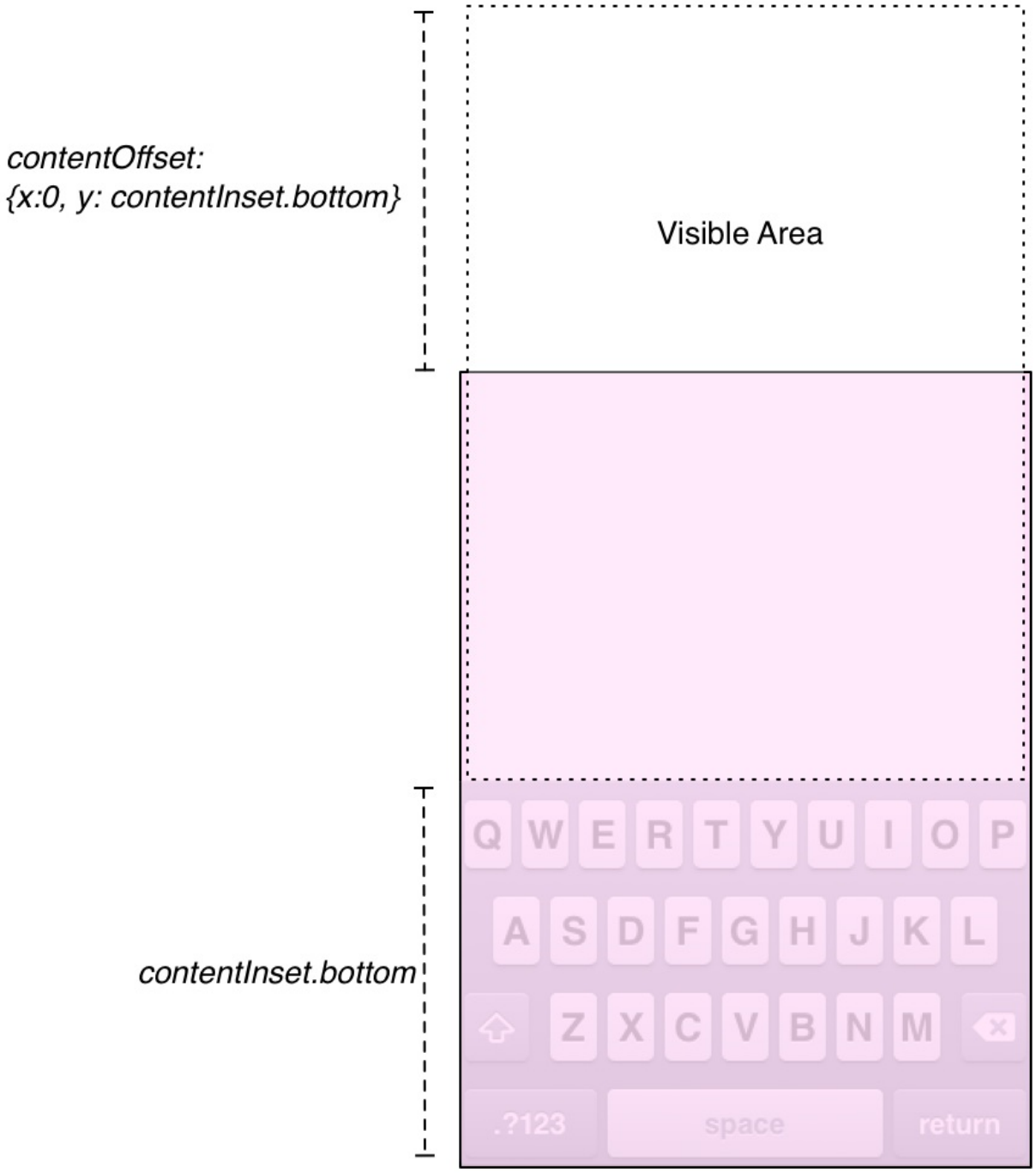
但是等等，如果你通过滚动足够多的距离初始化pull-to-refresh机制，因为table view设置了content inset，这将允许content

offset将refresh control弹回到可滚动区域。当刷新动作被初始化时，content inset已经被校正过，所以content offset的最小值包含了完整的refresh control。当刷新完成后，content inset恢复正常，content offset也跟着适应大小，这里并不需要为content size做数学计算。(这里可能比较难理解，建议看看EGOTableViewPullRefresh这样的类库就应该明白了)

如何在自己的代码中使用content inset？当键盘在屏幕上时，有一个很好的用途：你想要设置一个紧贴屏幕的用户界面。当键盘出现在屏幕上时，你损失了几百个像素的空间，键盘下面的东西全都被挡住了。

现在，scroll view的bounds并没有改变，content size也并没有改变(也不需要改变)。但是用户不能滚动scroll view。考虑一下之前一个公式：content offset的最大值并不同于content size和bounds的大小。如果他们相等，现在content offset的最大值是{x:0,y:0}.

现在开始出绝招，将界面放入一个scroll view。scroll view的content size仍然和scroll view的bounds一样大。当键盘出现在屏幕上时，你设置content inset的底部等于键盘的高度。



这允许在content offset的最大值下显示滚动区域外的区域。可视区域的顶部在scroll view bounds的外面，因此被截取了(虽然它在屏幕之外了，但这并没有什么)。

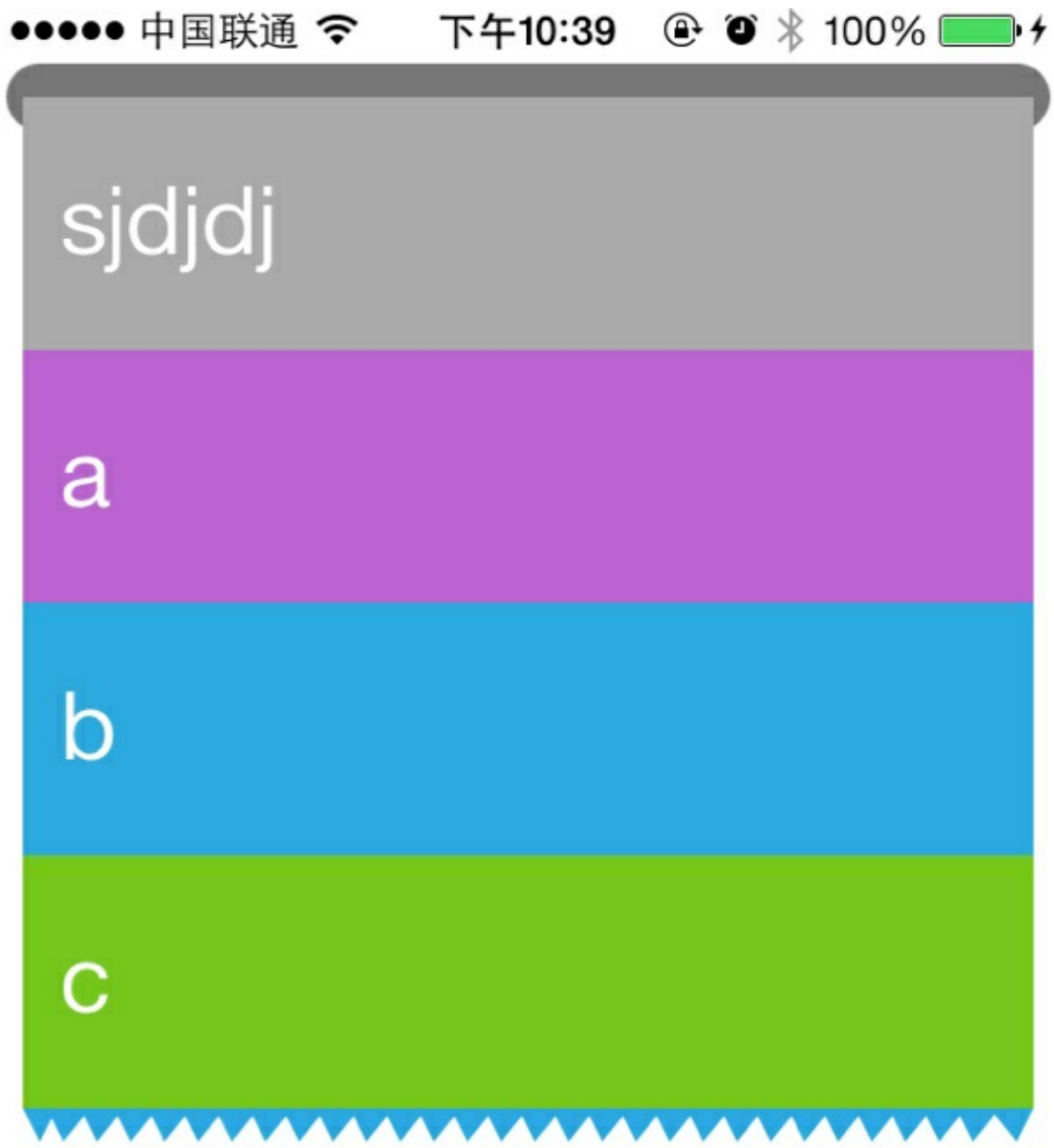
但愿这能让你理解一些滚动视图内部工作的原理，你对缩放感兴趣？好吧，我们今天不会谈论它，但是这儿有一个有趣的小窍门：检查 viewForZoomingInScrollView:方法返回视图的transform属性。你将再次发现scroll view只是聪明的利用了UIView已经存在的属性。

impletation(实现TableView)

好了在上面的工作准备的差不多了之后，我们大概了解了UIKit给我们提供了一些什么基础的工具。貌似我们就可以大刀阔斧的开始搞了。不对，等等，貌似我们缺了掉什么。好像是设计模式相关的东西，比如享元模式、责任链模式等等。这些东西就在我们用的时候，说一下吧。读者也可以照一本设计模式的书放在身边，以备不时之需。

项目相关的代码可以从：DZTableView获取。

先看个效果图：



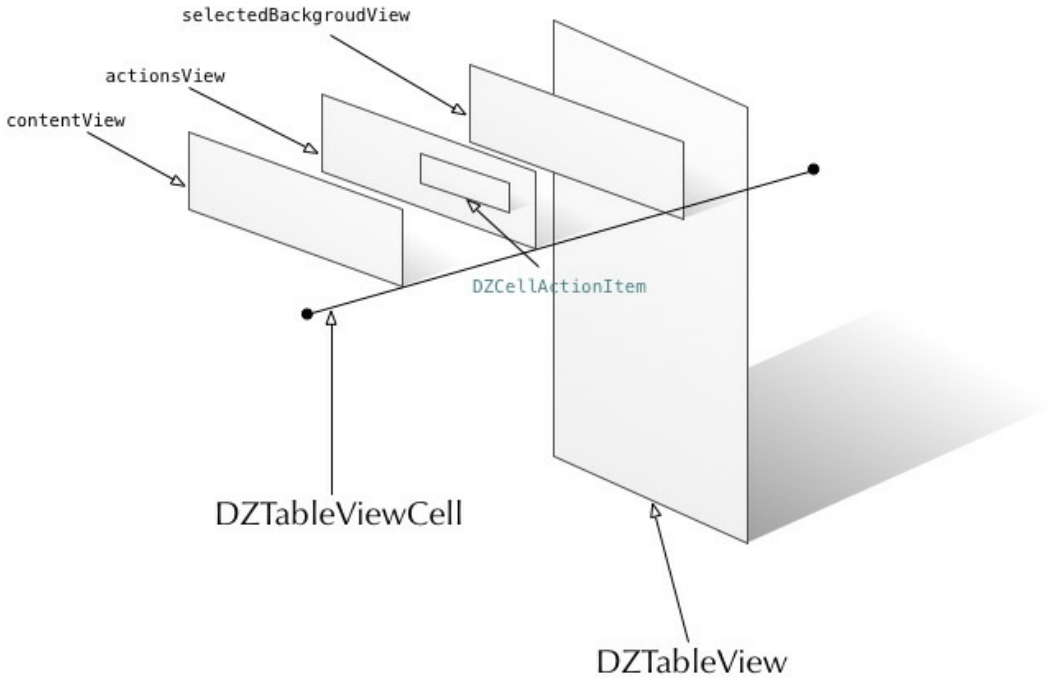
先说一下我们都实现了些什么东西：

1. 基本的TableView对Cell的布局
2. Cell的增加和删除
3. 右滑出现删除和编辑菜单
4. 下拉输入并新建一个cell

废话不多说开始干活！！！

解释一下整个UI的层次架构

下面这张图大概说明了整个DZTableView的View的结构树。



整个的TableView分成两个主要的组成部分:DZTableView和DZTableViewCell。这个结构和UITableView的结构是类似的。

DZTableView是tableView的主体部分，主要负责整个tableview的布局和渲染。而DZTableViewCell则是被布局和渲染的对象。DZTableView只是实现了y轴上纵向布局的tableView，没有分组。而我们通常看到的很多很炫的右滑删除等效果则是在DZTableViewCell上扩展得来的。

DZTableViewCell最基础的类主要有三个层次：

1. 负责渲染转中状态的selectedBackgroudView
2. 负责渲染和控制滑动效果的actionsView，actionsView上面各种功能的对象是DZCellActionItem
3. 负责渲染Cell主体内容的contentVIew。

而完成一个TableView主要的工作就是在UISCrollView上对cell进行合理的布局。

子类化UIScrollView实现对Cell的布局

解释一下为什么要从UIScrollView继承来完成TableView。这个和TableView的功能是密切相关的。TableView是一种内容数量大小不确定的布局方式，于是其需要在有限的屏幕（640*960）内展示无限的内容，而有这个功能的类就是UIScrollView。所以DZTableView从UIScrollView继承而来。

```
@interface DZTableView : UIScrollView
```

然后我们来看一下怎样去布局。分析一下，一个纵向的TableView布局的话，基本上是一个Cell接一个cell在纵向上确定他们的frame就能够布局出来了。那么我们的主要任务就是确定cell的位置。

为了确定cell的位置我们定义了一些变量：

```
typedef map<int, float> DZCellYoffsetMap;
typedef vector<float>   DZCellHeightVector;
.....
DZCellHeightVector _cellHeights;
DZCellYoffsetMap _cellYOffsets;
```

_cellHeights存储了所有cell的高度，而_cellYOffsets存储了每一个cell在y轴方向上的坐标。每一个cell在横向上是以填满为准的。即从View的最左侧开始布局（x=0）一直到最右侧右侧(width=view的宽度)。所以一般一个cell的绝对位置就是:

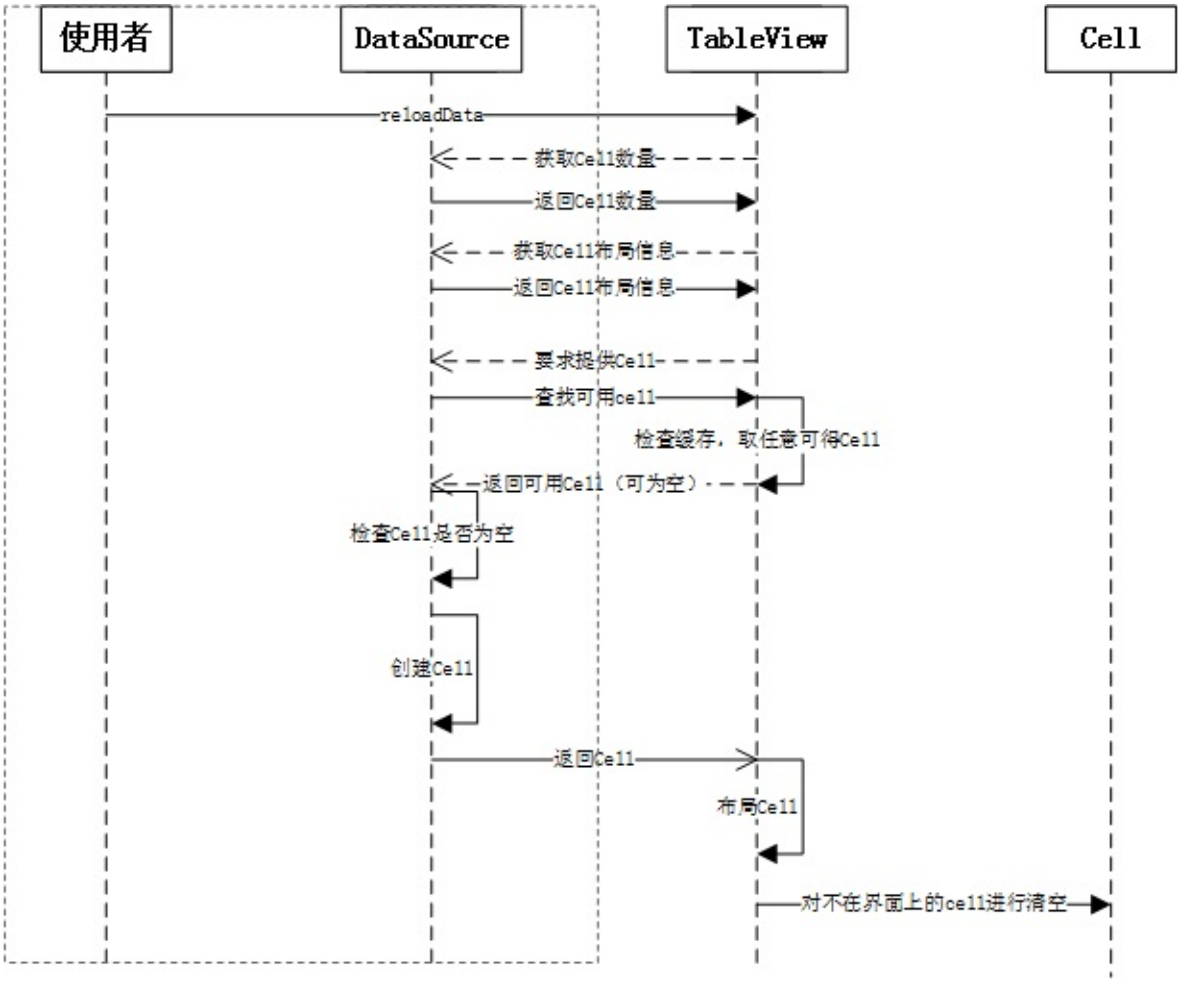
```
- (CGRect) _rectForCellAtRow:(int)rowIndex
{
    if (rowIndex < 0 || rowIndex >= _numberOfCells) {
        return CGRectZero;
    }
    float cellYoffSet = _cellYOffsets.at(rowIndex);
    float cellHeight  = _cellHeights.at(rowIndex);
    return CGRectMake(0, cellYoffSet - cellHeight, CGRectGetWidth(self.frame), cellHeight);
}
```

开始提到的几个关键的临时变量实在reduceContentSize函数中初始化的

```
- (void) reduceContentSize
{
    _numberOfCells = [_dataSource numberOfRowsInDZTableView:self];
    _cellYOffsets = DZCellYoffsetMap();
    _cellHeights = DZCellHeightVector();
    float height = 0;
    for (int i = 0  ; i < _numberOfCells; i ++) {
        float cellHeight = (_dataSourceReponse.funcHeightRow? [_dataSource dzTableView:self cellHeightAtRow:i] : kDZTableViewDefau
        _cellHeights.push_back(cellHeight);
        height += cellHeight;
        _cellYOffsets.insert(pair<int, float>(i, height));
    }
    if (height < CGRectGetHeight(self.frame)) {
        height = CGRectGetHeight(self.frame) + 2;
    }
    height += 10;
    CGSize size = CGSizeMake(CGRectGetWidth(self.frame), height);

    [self setContentSize:size];
    [self reloadPiceGradientColor];
}
```

这样一来我们就能够确认每一个cell的在TableView中的绝对位置，以后无论是正常情况下的布局，或者在增加或者删除cell时的布局，就比较简单了。直接调用 `_rectForCellAtRow` 函数获取cell的frame，然后布局就ok了。

Cell的重用



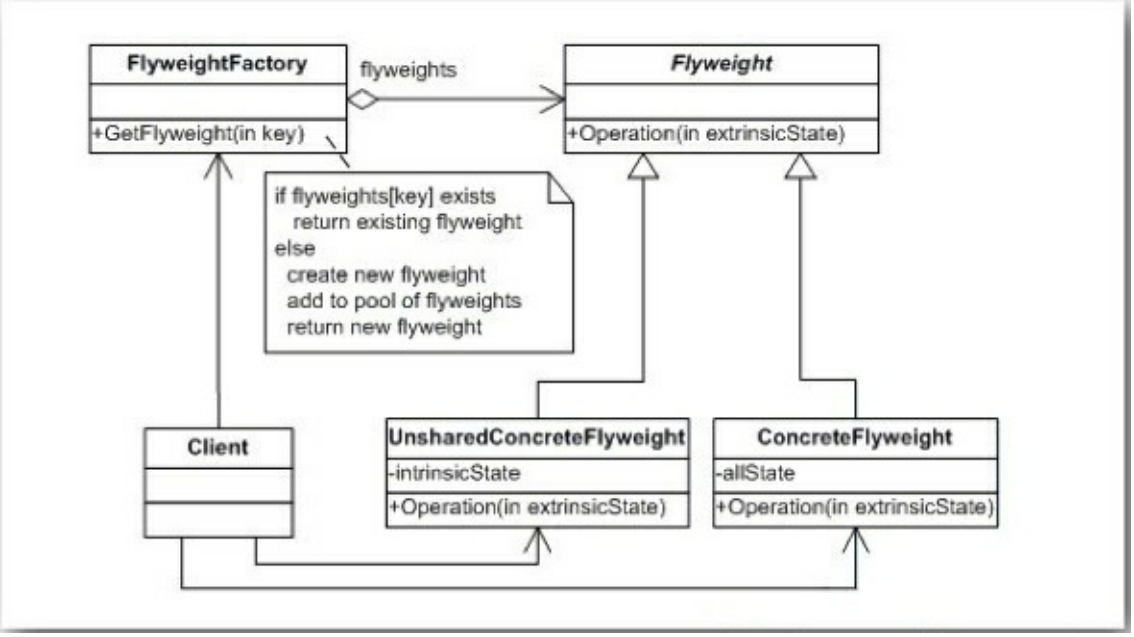
在使用UITableView的时候我们应该熟悉这样的接口：

```
- (id)dequeueReusableCellWithIdentifier:(NSString *)identifier

//ios6
- (id)dequeueReusableCellWithIdentifier:(NSString *)identifier forIndexPath:(NSIndexPath *)indexPath
```

在要使用一个Cell的时候我们先去看看tableView中有没有可以重用的cell，如果有就用这个可以重用的cell，只有在没有的时候才去创建一个Cell。这就是享元模式。

享元模式可以理解成，当细粒度的对象数量特别多的时候运行的代价会相当大，此时运用共享的技术来大大降低运行成本。比较突出的表现就是内容有效的抑制内存抖动的情况发生，还有控制内存增长。它的英文名字是flyweight，让重量飞起来。哈哈。名副其实，在一个TableView中Cell是一个可重复使用的元素，而且往往需要布局的cell数量很大。如果每次使用都创建一个Cell对象，系统的内容抖动会非常明显，而且系统的内存消耗也是比较大的。突然一想，享元模式只是给对象实例共享提供了一个比较霸道的名字吧。

一个典型的享元模式的UML图示例如下：



而在DZTableView中的实现中，享元模式中Cell的实例的存储和共享主要是在tableView中完成的。

```
NSMutableSet* _cacheCells;
NSMutableDictionary* _visibleCellsMap;
```

我们定义了两个用来存储两种不同类型的cell的容器：

1. _cacheCells 存储不再使用过程中，可以被复用的cell
2. _visibleCellsMap 按照键值对的方式存储了在使用中的cell。key是cell的顺序信息，即是自上而下的第几个cell。

而我们获取一个cell的函数如下：

```
- (DZTableViewCell*) _cellForRow:(NSInteger)rowIndex
{
    DZTableViewCell* cell = [_visibleCellsMap objectForKey:@(rowIndex)];
    if (!cell) {
        cell = [_dataSource dzTableView:self cellAtRow:rowIndex];
        DZCellActionItem* deleteItem = [DZCellActionItem buttonWithType:UIButtonTypeCustom];
        deleteItem.backgroundColor = [UIColor redColor];
        [deleteItem addTarget:self action:@selector(deleteCellOfItem:) forControlEvents:UIControlEventTouchUpInside];
        [deleteItem setTitle:@"删除" forState:UIControlStateNormal];
        deleteItem.edgeInset = UIEdgeInsetsMake(0, 10, 0, 260);
        DZCellActionItem* editItem = [DZCellActionItem buttonWithType:UIButtonTypeCustom];
        editItem.edgeInset = UIEdgeInsetsMake(0, 80, 0, 180);
        editItem.backgroundColor = [UIColor greenColor];
        [editItem setTitle:@"编辑" forState:UIControlStateNormal];
        [editItem addTarget:self action:@selector(editCellOfItem:) forControlEvents:UIControlEventTouchUpInside];
        cell.actionsView.items = @[deleteItem,editItem ];
    }
    return cell;
}
```

我们分几种情况来说明一下在布局cell的时候cell的重用问题。

已经在界面上的**cell**

对于已经在界面的cell我们很明显没有必要去重新构建，甚至没有必要去数据源去要。直接获取到相应的cell就好了。

```
DZTableViewCell* cell = [_visibleCellsMap objectForKey:@(rowIndex)];
```

没有在界面上的**cell**

对于没有在界面的cell，我们就需要去数据那里去要：

```
cell = [_dataSource dzTableView:self cellAtRow:rowIndex];
```

数据源在处理这个请求的时候就是按照上面我们说的享元模式的规则来了：

```
- (DZTableViewCell*) dzTableView:(DZTableView *)tableView cellAtRow:(NSInteger)row
{
    static NSString* const cellIdentifiy = @"detifail";
    DZTypeCell* cell = (DZTypeCell*)[tableView dequeueDZTalbeViewCellForIdentifiy:cellIdentifiy];
    if (!cell) {
        cell = [[DZTypeCell alloc] initWithIdentifiy:cellIdentifiy];
    }
    NSString* text = _timeTypes[row];
    return cell;
}
```

先去看看tableView中有没有可以重用的cell，有就用，没有就新建。但是tableView是怎么知道有可以重用的cell的呢。

DZTableView 可重用cell的cache

首先我们看一下获取重用cell的函数：

```
- (DZTableViewCell*) dequeueDZTalbeViewCellForIdentifiy:(NSString*)identifiy
{
    DZTableViewCell* cell = Nil;
    for (DZTableViewCell* each  in _cacheCells) {
        if ([each.identifiy isEqualToString:identifiy]) {
            cell = each;
            break;
        }
    }
    if (cell) {
        [_cacheCells removeObject:cell];
    }
    return cell;
}
```

很明显我们去_cacheCells中检查有没有特定identifiy的cell存在，如果有就说明有可重用的cell。这是一个直接获取的过程，那么久必然会存在往里面放cell的过程。

```
- (void) layoutNeedDisplayCells
{
    ...
    [self cleanUnusedCellsWithDispalyRange:displayRange];
    ...
}
- (void) cleanUnusedCellsWithDispalyRange:(NSRange)range
{
    NSDictionary* dic = [_visibleCellsMap copy];
    NSArray* keys = dic.allKeys;
    for (NSNumber* rowIndex  in keys) {
        int row = [rowIndex intValue];
        if (!NSLocationInRange(row, range)) {
            DZTableViewCell* cell = [_visibleCellsMap objectForKey:rowIndex];
            [_visibleCellsMap removeObjectForKey:rowIndex];
            [self enqueueTableViewCell:cell];
        }
    }
}
```

我们在布局完cell的时候，回去清理界面上无用的cell。同时把这些cell放入可重用cell的容器中。等待下次使用的时候，复用。

DZTableViewCell相关

当然，如果只是DZTableView单方面的想去重用cell是不肯能的。我们需要对DZTableViewCell做一些处理，才能够让这套享元模式运转起来。上面的代码中我们已经看到了，我们为DZTableViewCell添加了一些属性:

```
//DZTableViewCell_private.h
@interface DZTableViewCell ()
@property (nonatomic, strong) NSString* identifiy;
@property (nonatomic, assign) NSInteger index;
@end
```

identifiy标识了这个cell的种类。方便我们复用同一种类的cell。因为DZTableViewCell上可能会存在多种不同种类的cell，如果没有标识的重用起来就不知道获取到的cell是否能够适应特定的种类了。

还有一个index信息，这个是cell的顺序信息，主要是为了方便定位cell的位置用的。

值得注意的是这个定义是以Catogory的方式，定义在DZTableViewCell_private.h文件中的，而该文件只在DZTableView.mm中被引用，这样就避免了上面这些属性暴露给使用者，方式使用者使用方式不当导致的问题。或句话说，这些都是私有变量。必须被保护起来。

同时我们还定义和实现了一个函数：

```
- (void) prepareForReused;
....
- (void) prepareForReused
{
   _index = NSNotFound;
   [self setIsSelected:NO];
}
```

既然我们要复用一个Cell，那么就得在复用之前把Cell清理干净把，不然带着老数据去使用，用着用着就乱了，你就不知道cell的数据是对的还是错的了。

响应和处理事件

前面说过一个tableView应该是可交互的，而主要的交互就是能够确认用户点击了哪一个cell。

```
- (void) addTapTarget:(id)target selector:(SEL)selecotr
{
    self.userInteractionEnabled = YES;
    UITapGestureRecognizer* tapGesture = [[UITapGestureRecognizer alloc] initWithTarget:target action:selecotr];
    tapGesture.numberOfTapsRequired = 1;
    tapGesture.numberOfTouchesRequired = 1;
    [self addGestureRecognizer:tapGesture];
}
...
[self addTapTarget:self selector:@selector(handleTapGestrue:)];
...
- (void) handleTapGestrue:(UITapGestureRecognizer*)tapGestrue
{
    CGPoint point = [tapGestrue locationInView:self];
    NSArray* cells = _visibleCellsMap.allValues;
    for (DZTableViewCell* each in cells) {
        CGRect rect = each.frame;
        if (CGRectContainsPoint(rect, point)) {
            if ([_actionDelegate respondsToSelector:@selector(dzTableView:didTapAtRow:)]) {
                [_actionDelegate dzTableView:self didTapAtRow:each.index];
            }
            each.isSelected = YES;
            _selectedIndex = each.index;
        }
        else
        {
            each.isSelected = NO;
        }
    }
}
```

我们在tableview上面加了一个单机的事件UITapGestureRecognizer。然后再相应处处理了一下。主要是获取了用户点击位置，然后找到点击位置上的cell。这样就确认了用户点了哪个cell，在把这个信息传出去就好了。

25在DZTableViewCell上扩展功能

251Cell结构设计

选中态

这个应该是所有View的一个基础功能，在很多基于UIView的空间上我们都能看到 setHeightlight 或者 setSelected 之类的函数，用来在用户选中该空间的时候，给用户一个反馈。DZTableViewCell的是 setSelected 。关于选中态主要有两部分的事情，一是选中时机，二是如何表现选中态。

选中态的判断

选中太的判断主要是依靠触摸事件来判断，当用户触摸到cell的时候表示选中，用户手指离开的时候为不选中。于是我们通过重载UIView的一些列触摸事件的响应函数就能够做到对选中态的判断。

```
- (void) touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event
{
   [super touchesBegan:touches withEvent:event];
   [self setIsSelected:YES];
}

- (void) touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event
{
   [super touchesEnded:touches withEvent:event];
   [self setIsSelected:NO];
}

- (void) touchesCancelled:(NSSet *)touches withEvent:(UIEvent *)event
{
   [super touchesCancelled:touches withEvent:event];
   [self setIsSelected:NO];
}
```

选中态的展示

回归一下刚开始的时候说到的，我们整个DZTableView的UIView数层次。一个Cell的最底层是一个 _selectedBackgroudView。这个就是用来展示选中态的。当Cell的选中态改变的时候，我们只要重新布局一下 _selectedBackgroudView就可以了。

```
- (void) setIsSelected:(BOOL)isSelected
{
   if (_isSelected != isSelected) {
      _isSelected = isSelected;
      [self setNeedsLayout];
   }
}
- (void) layoutSubviews
{
   ....
   if (_isSelected) {
      _selectedBackgroudView.frame = _contentView.bounds;
      _selectedBackgroudView.hidden =  NO;
      [_contentView insertSubview:_selectedBackgroudView atIndex:0];
   }
   else
   {
      _selectedBackgroudView.hidden = YES;
   }
   ....
```

253手势与功能

254子类化扩展

接口和数据获取

通过上面的阐述我们已经把DZTableView的框架搭起来了，实现了一个TableView的布局方式，还有cell的重用。但是还有一个非常关键的问题，tableView布局信息的数据怎么来，还有我们应该向外给提供者调用什么样的接口。

这个问题，貌似苹果已经做得很好了。而DZTableView要做的就是尽可能的让接口和苹果的保持一致，这样对于使用者而言，没有太大的学习成本。

数据获取

```
@class DZTableView;
@class DZTableViewCell;
@class DZPullDownView;
@protocol DZTableViewSourceDelegate <NSObject>
- (NSInteger) numberOfRowsInDZTableView:(DZTableView*)tableView;
- (DZTableViewCell*) dzTableView:(DZTableView*)tableView cellAtRow:(NSInteger)row;
- (CGFloat) dzTableView:(DZTableView*)tableView cellHeightAtRow:(NSInteger)row;
@end
```

点击等事件响应

```
@class DZTableView;
@class DZTableViewCell;
@protocol DZTableViewActionDelegate <NSObject>

- (void) dzTableView:(DZTableView*)tableView didTapAtRow:(NSInteger)row;
- (void) dzTableView:(DZTableView *)tableView deleteCellAtRow:(NSInteger)row;
- (void) dzTableView:(DZTableView *)tableView editCellDataAtRow:(NSInteger)row;

@end
```

DZTableView的成员方法

```
- (DZTableViewCell*) dequeueDZTalbeViewCellForIdentifiy:(NSString*)identifiy;
- (void) reloadData;
- (void) insertRowAt:(NSSet *)rowsSet withAnimation:(BOOL)animation;
- (void) removeRowAt:(NSInteger)row withAnimation:(BOOL)animation;

- (void) manuSelectedRowAt:(NSInteger)row;
```

视图控制器DZTableViewController

在第一章中，我们简单说了一下ViewController在整个UIKit中的作用。简单归纳一下就是：

- 创建和管理视图。
- 管理视图上显示的数据。
- 设备方向变化，调整视图大小以适应屏幕。
- 负责视图和模型之间的数据及请示的传递。

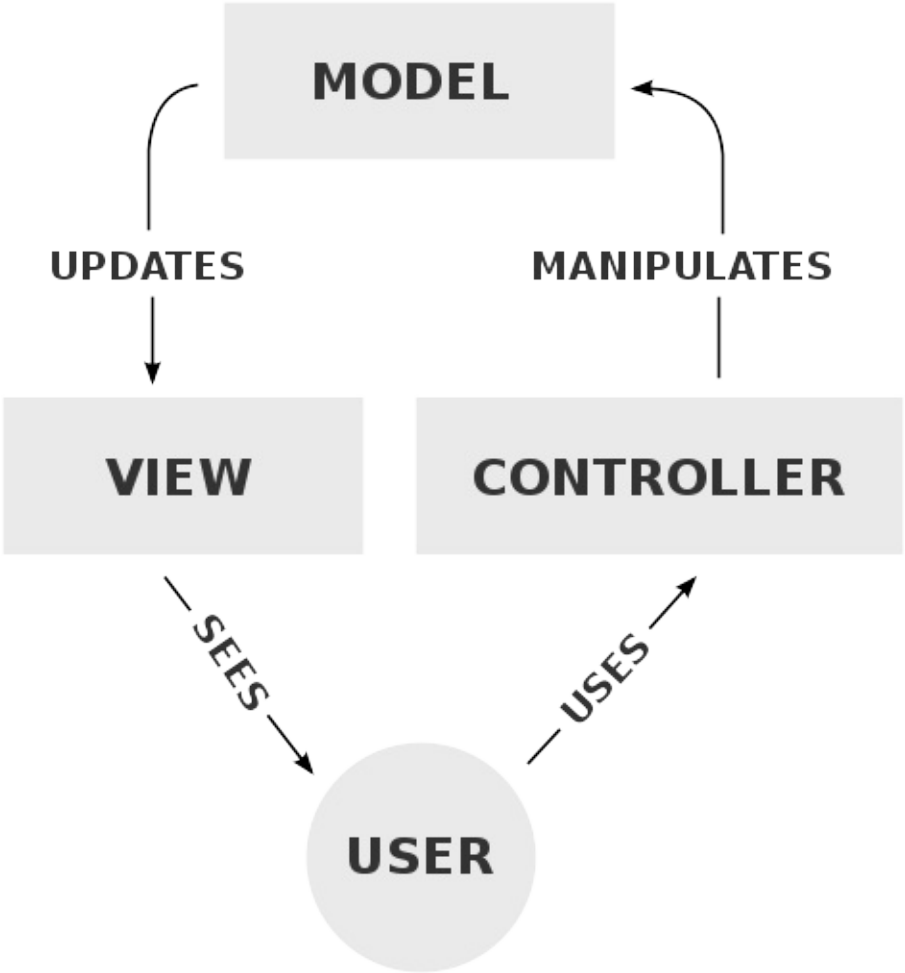
大家都熟悉的MVC架构中，ViewController正式处于C层。起到了对View层和Model层的组织作用，同时控制应用程序的流程。并且会处理用户事件，对之做出相应。 从另外一个角度——三层架构来看，ViewController承载着我们整个程序绝大部分的业务逻辑。于是，在我们实际的编程实践中，会发现很多逻辑控制的代码其实都写在了ViewController生命周期的各个函数中。

MVC模式最早由Trygve Reenskaug在1978年提出[1]，是施乐帕罗奥多研究中心（Xerox PARC）在20世纪80年代为程序语言Smalltalk发明的一种软件设计模式。MVC模式的目的是实现一种动态的程序设计，使后续对程序的修改和扩展简化，并且使程序某一部分的重复利用成为可能。除此之外，此模式通过对复杂度的简化，使程序结构更加直观。软件系统通过对自身基本部分分离的同时也赋予了各个基本部分应有的功能。专业人员可以通过自身的专长分组：

- （控制器Controller）- 负责转发请求，对请求进行处理。
- （视图View） - 界面设计人员进行图形界面设计。
- （模型Model） - 程序员编写程序应有的功能（实现算法等等）、数据库专家进行数据管理和数据库设计(可以实现具体的功能)。

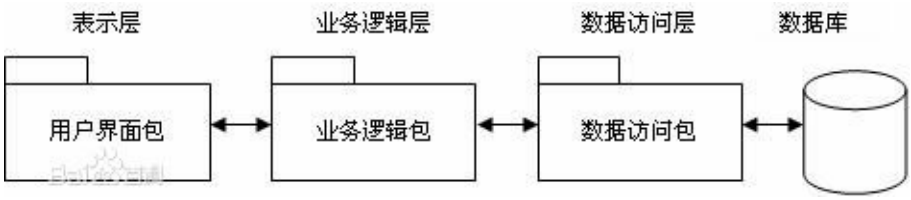
除了将应用程序划分为三种组件，模型 - 视图 - 控制器（MVC）设计定义它们之间的相互作用。[2]

- 模型（Model） 用于封装与应用程序的业务逻辑相关的数据以及对数据的处理方法。“模型”有对数据直接访问的权力，例如对数据库的访问。“模型”不依赖“视图”和“控制器”，也就是说，模型不关心它会被如何显示或是如何被操作。但是模型中数据的变化一般会通过一种刷新机制被公布。为了实现这种机制，那些用于监视此模型的视图必须事先在此模型上注册，从而，视图可以了解在数据模型上发生的改变。（比较：观察者模式（软件设计模式））
- 视图（View）能够实现数据有目的的显示（理论上，这不是必需的）。在视图中一般没有程序上的逻辑。为了实现视图上的刷新功能，视图需要访问它监视的数据模型（Model），因此应该事先在被它监视的数据那里注册。
- 控制器（Controller）起到不同层面间的组织作用，用于控制应用程序的流程。它处理事件并作出响应。“事件”包括用户的行为和数据模型上的改变。



三层架构(3-tier architecture) 通常意义上的三层架构就是将整个业务应用划分为：表现层（UI）、业务逻辑层（BLL）、数据访问层（DAL）。区分层次的目的即为了“高内聚，低耦合”的思想。

1. 表现层（UI）：通俗讲就是展现给用户的界面，即用户在使用一个系统的时候他的所见所得。
2. 业务逻辑层（BLL）：针对具体问题的操作，也可以说是对数据层的操作，对数据业务逻辑处理。
3. 数据访问层（DAL）：该层所做事务直接操作数据库，针对数据的增添、删除、修改、查找等。



表示层

位于最外层（最上层），最接近用户。用于显示数据和接收用户输入的数据，为用户提供一种交互式操作的界面。 业务逻辑层

业务逻辑层（Business Logic Layer）

无疑是系统架构中体现核心价值的部分。它的关注点主要集中在业务规则的制定、业务流程的实现等与业务需求有关的系统设计，也即是说它是与系统所应对的领域（Domain）逻辑有关，很多时候，也将业务逻辑层称为领域层。例如Martin Fowler在《Patterns of Enterprise Application Architecture》一书中，将整个架构分为三个主要的层：表示层、领域层和数据源层。作为领域驱动设计的先驱Eric Evans，对业务逻辑层作了更细致地划分，细分为应用层与领域层，通过分层进一步将领域逻辑与领域逻辑的解决方案分离。 业务逻辑层在体系架构中的位置很关键，它处于数据访问层与表示层中间，起到了数据交换中承上启下的作用。由于层是一种弱耦合结构，层与层之间的依赖是向下的，底层对于上层而言是“无知”的，改变上层的设计对于其调用的底层而言没有任何影响。如果在分层设计时，遵循了面向接口设计的思想，那么这种向下的依赖也应该是一种弱依赖关系。因而在不改变接口定义的前提下，理想的分层式架构，应该是一个支持可抽取、可替换的“抽屉”式架构。正因为如此，业务逻辑层的设计对于一个支持可扩展的架构尤为关键，因为它扮演了两个不同的角色。对于数据访问层而言，它是调用者；对于表示层而言，它却是被调用者。依赖与被依赖的关系都纠结在业务逻辑层上，如何实现依赖关系的解耦，则是除了实现业务逻辑之外留给设计师的任务。 数据层

数据访问层：

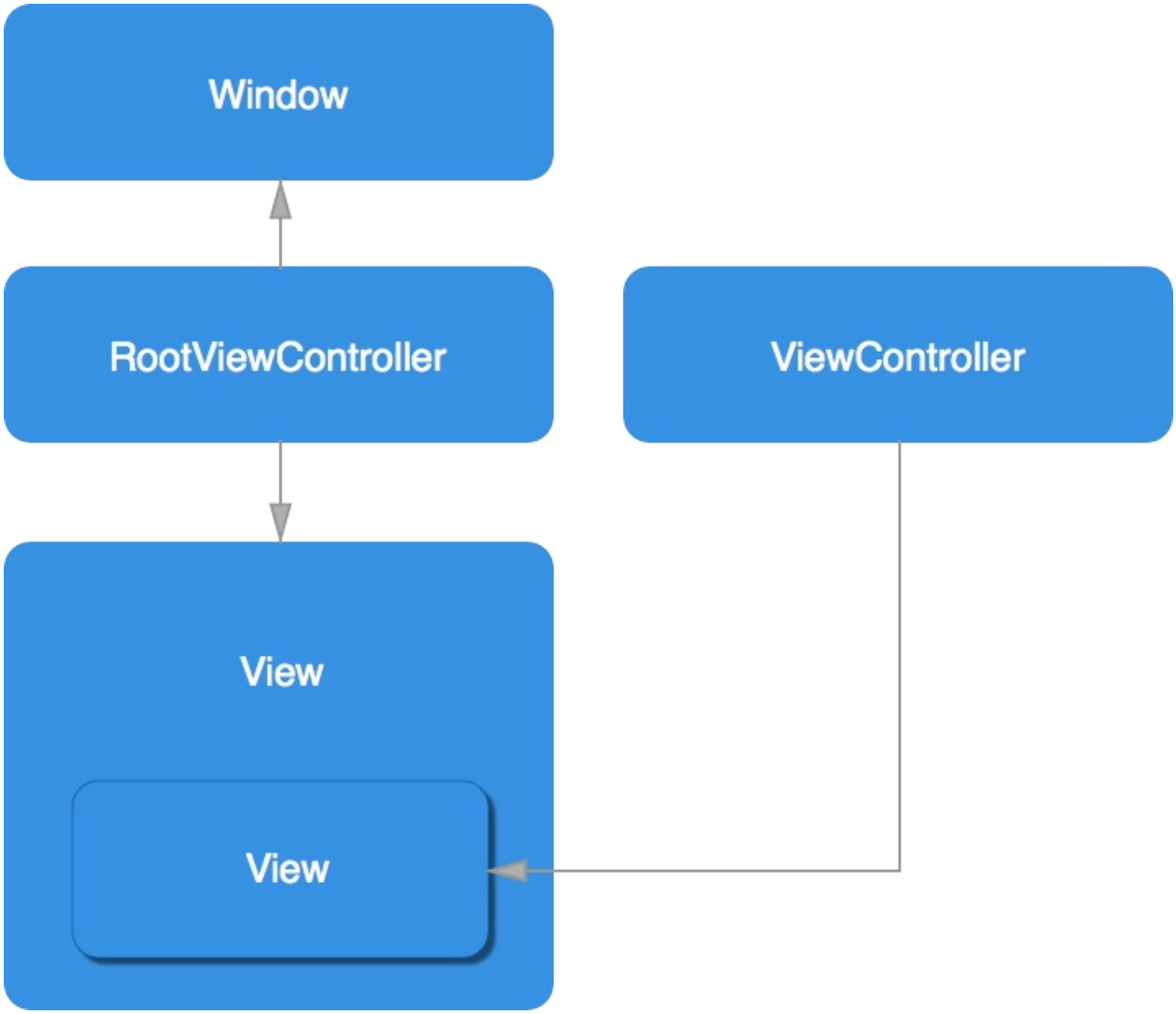
有时候也称为是持久层，其功能主要是负责数据库的访问，可以访问数据库系统、二进制文件、文本文档或是XML文档。 简单的说法就是实现对数据表的Select，Insert，Update，Delete的操作。如果要加入ORM的元素，那么就会包括对象和数据表之间的mapping，以及对象实体的持久化。

UIViewController的生命周期

MainView主视图

在 iOS 5.0 以前，视图控制器容器只属于苹果系统所有，苹果不建议你自定义视图控制器容器。实际上，在视图控制器编程指南这一章中明确告知我们不要使用它。以前苹果公司对于视图控制器容器的总体描述是“一个管理整个屏幕内容的视图控制器”，而现在的描述是“一个包含本身视图内容的单元集合”。为什么苹果不希望我们自定义像tab bar controllers 和navigation controllers这样的视图控制器容器呢？更准确的说，下面这条语句会带来什么问题：

```
[viewControllerA.view addsubView:viewControllerB.view]
```

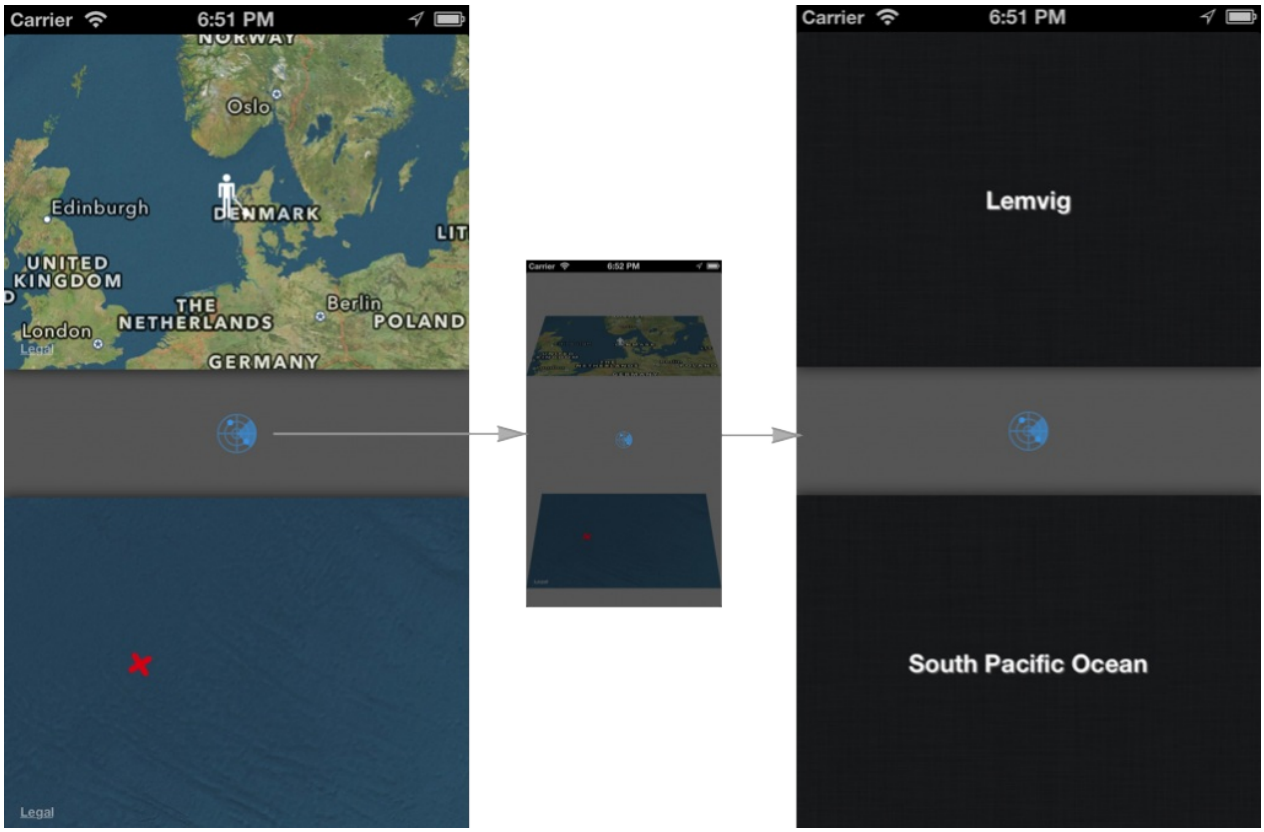


UIWindow作为应用程序的根视图，负责监听和分发屏幕旋转和视图布局等消息。在上图中，ViewController的view插入到根视图控制器当中的一个视图中作为其子视图，那么这个view就不会再接收到UIWindow传来的消息，像viewWillAppear：这种方法就不会被调用。

在ios5.0以前我们自定义的视图控制器容器，将会持有子类视图控制器的一个引用，并且需要我们手动的传递在父类视图控制器中调用的事件消息给子类视图控制器，准确地完成这项工作太难了！

实例详解:

小时候你在沙滩玩耍时，你的父母是否曾告诉过你：如果你用你的小铁锹一直挖，一直挖，最终你能挖到地球对面的中国去(作者在美国)。我父母曾这样给我说过，我做了一个叫Tunnel的应用来检验一下这个观点。你可以附加上GitHub Repo并运行这个程序，这样可以更好的帮助你理解这个实例。



如果想获得当前位置相对面的地点(地球的对面)，移动拿着铁锹的那个小人，地图会告知你准确的出口位置。触摸一下雷达按钮，地图就会翻转并显示确切的地名。

在当前的屏幕上，有两个地图视图控制器，每个都需要处理拖拽，标注位置，还有更新地图这些事件。翻转页面后会呈现对面地点的两个地图视图控制器。所有的这些视图控制器都存在于一个父类视图控制器容器中，这些视图控制器持有各自的视图，以确保视图的布局和翻转可以正确的进行。

根视图控制器拥有两个视图容器，添加两个的目的就是为了更方便各自容器的子视图进行布局，执行动画等操作，下面我们来解释一下。

```
- (void)viewDidLoad
{
    [super viewDidLoad];

    //Setup controllers
    _startMapViewController = [RGMapViewController new];
    [_startMapViewController setAnnotationImagePath:@"man"];
    [self addChildViewController:_startMapViewController];         // 1
    [topContainer addSubview:_startMapViewController.view];        // 2
    [_startMapViewController didMoveToParentViewController:self];  // 3
    [_startMapViewController addObserver:self
                              forKeyPath:@"currentLocation"
                                 options:NSKeyValueObservingOptionNew
                                 context:NULL];

    _startGeoViewController = [RGGeoInfoViewController new];       // 4
}
```

_startMapViewController用来显示开始的位置，先实例化并用标注图片初始化。

1. _startMapViewController作为根视图控制器的一个子视图，添加到其中，子类中的willMoveToParentViewController:方法将会自动执行。

2. 上面的子类视图作为第一个视图容器的视图的子视图添加到其中。

3. 子视图收到一个通知，它现在有了一个父视图。

4. 进行地理位置编码的子视图控制器被初始化，但是现在还没有被插入到任何视图或控制器结构中。

布局

根视图控制器定义的两个视图容器分别决定了它们的子视图控制器的大小。子类视图控制器不知道自己将会添加到哪个视图容器中，所以它们的size大小必须可变的。

```
- (void) loadView
{
   mapView = [MKMapView new];
   mapView.autoresizingMask = UIViewAutoresizingFlexibleWidth |UIViewAutoresizingFlexibleHeight;
   [mapView setDelegate:self];
   [mapView setMapType:MKMapTypeHybrid];

   self.view = mapView;
}
```

现在它们会以父类视图的bounds来布局，这样增加了子类视图的复用性，如果我们把它推入导航控制器栈中，它们依然会正确的布局。

转换效果

Apple提供的视图控制器容器的API是如此的详细，以至于我们几乎可以以我们能想到的任意方式来初始化容器或者执行各种动画。Apple还提供了方便使用的block方法来交换当前屏幕上的两个视图。

transitionFromViewController:toViewController: 上面这个方法为我们揭示了许多细节

```
- (void) flipFromViewController:(UIViewController*) fromController
           toViewController:(UIViewController*) toController
              withDirection:(UIViewAnimationOptions) direction
{
   toController.view.frame = fromController.view.bounds;                    // 1
   [self addChildViewController:toController];                              //
   [fromController willMoveToParentViewController:nil];                     //

   [self transitionFromViewController:fromController
                 toViewController:toController
                       duration:0.2
                        options:direction | UIViewAnimationOptionCurveEaseIn
                     animations:nil
                     completion:^(BOOL finished) {

                        [toController didMoveToParentViewController:self];  // 2
                        [fromController removeFromParentViewController];    // 3
                     }];
}
```

1.我们添加的toViewController执行动画之前，fromViewController会收到一个将会移除的通知，如果fromViewController是容器视图等级的视图，那么此时控制器的viewWillDisappear:方法将会被调用。

2.toViewController接收到来自父类视图的消息，其对应的方法会被调用。

3.fromViewController被移除。

上面的这个方法，可以自动转换出旧的视图并替换成新的视图。如果你想实现你自己的转换方法，并且想让每一个时刻都只能显示一个视图，你必须对旧的视图调用removeFromSuperview方法，新的视图调用addSubview:方法。调用方法的次序如果出错将会导致UIViewControllerHierarchyInconsistency警告。比如在你添加视图之前调用视图的didMoveToParentViewController:将会出错。

为了能够使用 UIViewAnimationOptionTransitionFlipFromTop动化序列，我们必须把子类视图添加到视图容器控制器上，而不是添加到根视图控制器上，否则的化，执行动画将会导致整个根视图翻转。

通信

视图控制器应该是能够复用，包含当前视图的一个实体，子类视图也应该遵循这一经验法则。为了遵循这一法则，父类视图

控制器应该只负责两个任务：布局子类视图控制器的根视图，通过自身暴露的API接口与子类视图控制器通信，不应该直接修改子类视图的树形结构和状态。

在Tunnel应用实例中，父类视图控制器会适时监听map view controllers当中的currentLocation属性。

```
[_startMapViewController addObserver:self
                          forKeyPath:@"currentLocation"
                             options:NSKeyValueObservingOptionNew
                             context:NULL];
```

在地图旁边拿着铁锹的小男孩移动时，相应的属性就会变化，这时父类视图控制器将新的位置相对的(地球对面)的地点传递给另一个地图。

类似地，当你点击雷达按钮，父类视图控制器在新的子类视图控制器中设置与以前位置对应的地理位置。

```
[_startGeoViewController setLocation:_startMapViewController.currentLocation];
[_targetGeoViewController setLocation:_targetMapViewController.currentLocation];
```

尽量选择独立的方法来完成子类视图与父类视图的通信(KVO, 通知中心, 代理模式)，这与上面提到的通信法则一致，即子类视图控制器应该独立，并且具有复用性。在我们的例程中，我们可以把一个视图控制器压入到导航控制器栈中，但是通信接口仍然选择相同的API。

Extend DZTableView的可扩展性探讨

既然我们要实现一个类似于UITableView一样非常通用的组件，也就要求DZTableView的可扩展性就要好一点。这包括：

1. 属性的可配置型
2. 功能上的扩展性，方便子类化

为了展示这个，特意做了右滑删除，还有下拉新建cell的功能。因为本文的主要目的是通过自己构建一个TableView来解释IOS UI编程。所以就不详细展开讨论。看一下代码大概就能明白了。

什么是可扩展性

我们要理解可扩展性这个东西，最好的一个方法就是给他下个定义。因为我一直坚信一句老话：you can say it， you know it。只有当你能够准确定义一个东西并且描述它的时候，你才能够真正说你理解了它。于是我们查了一些资料来找关于可扩展性的定义。

1. Wiki上说：可扩放性(Scalability)是指问题规模和处理器数目之间的函数关系（Wiki）。这个怎么看也觉得和我们想要讨论的问题，有点不太搭边。
2. 百度百科上说：设计良好的代码允许更多的功能在必要时可以被插入到适当的位置中。这样做的目的的是为了应对未来可能需要进行的修改，而造成代码被过度工程化地开发。 可扩展性可以通过软件框架来实现：动态加载的插件、顶端有抽象接口的认真设计的类层次结构、有用的回调函数构造以及功能很有逻辑并且可塑性很强的代码结构。 可扩展性是软件设计的原则之一，它以添加新功能或修改完善现有功能来考虑软件的未来成长。可扩展性是软件拓展系统的能力。 简单地说，可扩展性就是关于如何处理更大规模的业务。比如，Web应用程序就是允许更多的人使用你的服务。如果你不能弄清楚如何提高性能的同时向外扩展，没关系。只要你能处理更大规模的用户，即使是存在多个单点故障也没有问题。组合的可扩展性要求要满足用户不断发展的要求，还要满足因技术发展需要而实现的扩展和升级的需求。 百度百科。 这个解释貌似开始靠点谱了，但是有种大杂烩的感觉。像是还有些什么东西不能够说出来。

在苦苦寻找后，最后发现没能够发现比较合适的资料来解释可扩展性这个东西。那么就只能够说说我自己的看法了。

可扩展性是一种设计概念，代表了我们对未来的一种预想，我们希望在现有的架构或者设计基础上，当未来某些方面发生噶边的时候，我们能够以最小的改动来适应这种变化。我们的改动越小，并且对这种变化的适应性越好，我们就会说这个设计的可扩展性是非常好的。简单来说，就是让当前设计去适应未来不确定的变化。

很多时候，让设计有可扩展性是和设计者本身有着非常直接的联系。举个例子来说，我们在UI编程的时候，经常会遇到很多hardcode的代码，把很多控件的坐标用死数字写死。我们对这样的代码嗤之以鼻，因为他们适应不了任何变化。一旦屏幕大小发生改变，甚至是父视图的布局发生改变，就会让整个界面混乱。因为，那些用死数字写死的坐标的控件，真的是死的。无论周围条件发生怎样的改变，他们都无动于衷，呆若木鸡。当我们去问做出这样设计的工程师的时候，他们会非常简单的反问你：需求不就说要展示成这个样子吗？也就是说，他们没有任何对未来的预期。在他们眼中，需求就是最终的模样，之后不会发生任何改变。

而与之形成对比的是，在UI编程的时候，有很多人即使在像IOS这种使用绝对布局模型的框架下编程的时候，他们也尽可能把相对布局的观念运用在编程实践。比如一个界面要布局三个控件，他们不是将三个控件的坐标写死，而是尽可能的找到这三个控件布局之间的关系。尽可能的通过描述这种关系来进行布局。比如他们发现着三个控件是居中对齐的，那么他们就会用代码去描述这种居中对齐的关系。他们这样做带来的好处是，当父视图布局或者其他条件发生改变的时候。整个界面会自动调整布局来适应这种变化。当你去问做出这样的设计决策的工程师，为什么要这样做的时候。他们会说：他妈的产品和设计的需求天天变，你现在写代码不考虑这种变化，将来你就得花更多的时间去改代码。