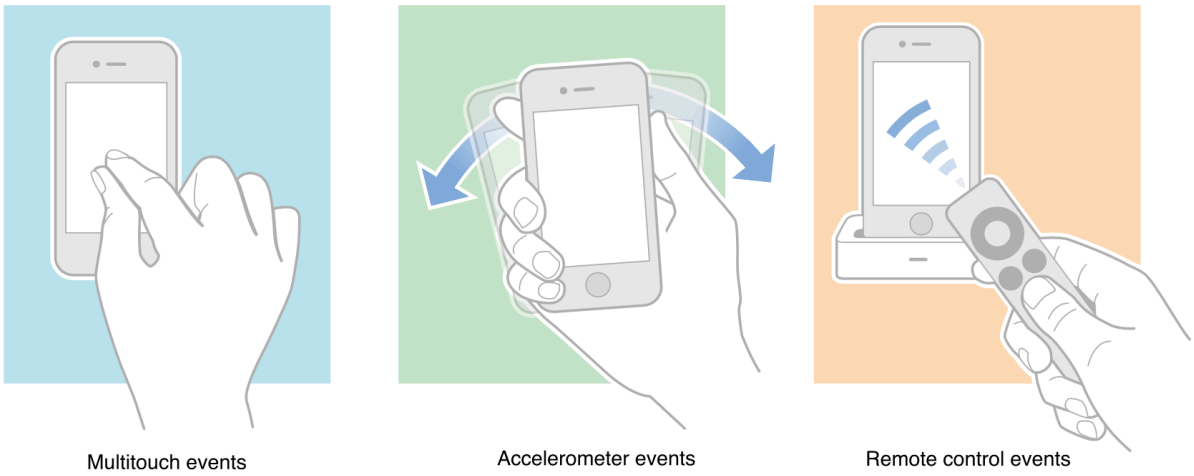
Event Handling Guide for iOS 翻译

关于 iOS 的事件

用户操作iOS设备的方式有很多种，比如触摸屏幕、摇晃设备。当用户操作硬件时，iOS系统就会将这些事件进行捕获，然后翻译成事件信息传递给手机的应用程序进行处理，这样你的程序就可以对用户的人为动作进行识别并响应，当然，你的程序所支持的动作和响应越丰富，那么程序的用户体验就会越好。



1. 随意瞄一瞄

所有的事件都以对象的方式发送给应用程序，以告之其用户的操作信息。在iOS中，事件发生的形式有很多种：Multi-Touch（多指触摸）事件，motion（手势，如tap、slide等）事件，以及控制多媒体的事件。最后的那个事件类型，比较典型的比如远程控制事件，因为其事件源头来自于外部感应器。

1.1 UIKit 框架使得应用程序感知用户手势操作简单易用

iOS应用程序可以识别各种触摸集合事件，并直观而形象地对这些事件进行响应，比如用户通过两根手指对屏幕上的某个区域进行捏合或分离（pinch）操作来缩放，使用滑动手势（flick）对屏幕某一分内容进行翻动。一些非常常见的手势基本都直接在UIKit框架中构建集成（开发者可以直接使用）。举个例子，UIControl的子类，比如UIButton、UISlider，对于用户的特定手势直接响应，Button直接响应用户的点击（tap）事件，Slider直接响应用户的滑动事件。开发者对这些控件进行配置，这些控件就会给特定的目标实例发送用户触摸事件的消息。当然，开发者可以直接使用手势识别类（gesture recognizer）

让iOS视图类（view）具有target-action机制（以对用户的特定的操作进行响应）。开发者将手势识别类（gesture recognizer）实例和某个视图实例进行绑定，就可以像control（控制类）任意指定的用户手势操作进行响应。

手势识别类（gesture recognizer）为开发者提供了复杂事件处理逻辑的高度抽象接口，功能强大、可复用、适配性，自然是开发者实现对用户触摸事件处理的首选。开发者可以直接使用已集成（内建）的手势识别类，并自定义其处理动作，当然也可以自定义手势识别类来处理新的用户手势操作。

1.2 每一个事件都沿着特定路径去查找能够处理它的对象

当iOS捕获到某个事件时，就会将此事件传递给某个看上去最适合处理该事件的对象，比如触摸事件传递给手指刚刚触摸位置的那个视图（view），如果这个对象无法处理该事件，iOS系统就继续将该事件传递给更深层的对象，直到找到能够对该事件作出响应处理的对象为止。这一连串的对象序列被称作为“响应链”（responder chain），iOS系统就是沿着此响应链，由最外层逐步向内存对象传递该事件，亦即将处理该事件的责任进行传递。iOS的这种机制，使得事件处理具有协调性和动态性。

1.3 一个 UIEvent 既是一个触摸（touch）、一个摇晃（shake-motion），或者一个远程控制（remote-control）事件

UIKit的控制器类（control）和手势识别类（gesture recognizer）应该足以用来处理开发者的APP的触摸时间处理，当然这得视情况而定。即便APP自定义的视图可以直接应用手势识别类（gesture recognizer）。其首要原则是，开发者需要对APP的视图类实例的触摸事件进行响应处理，来适配开发自定义的最适合的事件处理方法，比如在某个视图的触摸事件发生时进行绘画。在这些情况下，开发者就需要对底层事件处理负责，实现触摸方法，在这些方法里，开发者需要分析这些原生的触摸事件，并进行合适的响应。

1.4 当用户运动设备时，APP 会接收到运动事件

运动事件提供的信息包括设备的位置、方向和活动信息。通过激活运动事件，开发者可以为应用程序添加微妙而强大的功能特性。加速器和陀螺仪数据使得开发者可以检测设备的

倾斜、翻转和摇晃事件。

运动事件的来源不一样，开发者处理这些事件所需要的框架也随之而变。用户摇晃设备时，UIKit框架就会传递一个UIEvent对象给应用程序。如果开发者想要接受高频率的、连续性的加速器和陀螺仪数据，就需要使用Core Motion框架。

1.5 当用户操作多媒体控制器时，APP 会接收到远程控制事件

iOS控制器和外部感应器会向APP发送远程控制事件信息。这些事件允许用户控制音频和视频，比如调整扬声器的音量、视频播放的进度等。对多媒体远程控制事件的处理，就需要开发者的APP对这些类型的指令进行响应。

2. 预备知识

该文档默认读者具备以下知识属性：

- *iOS APP*开的基础概念
- 创建*APP*用户界面的不同层面
- 视图和视图控制器如何运作，以及如何自定义视图和视图控制器

如果读者对以上内容不熟悉，请自行阅读相关书籍。比如通过阅读《Start Developing iOS Apps Today 》入门，然后阅读《View Programming Guide for iOS》或者 《View Controller Programming Guide for iOS 》，都看了也行。

3. 参考

iOS设备在提供触摸和设备运动事件的同时，大部分的iOS设备也集成有有GPS和指南针硬件，开发者的APP对它们所生成的底层数据，可能也会感兴趣。《Location Awareness Programming Guide 》讨论如何接受并处理这些位置信息数据。

更高级的手势识别技术比如曲线滤波和应用低通滤波器，请参见《WWDC 2012: Building Advanced Gesture Recognizers 》。

许多本书涉及的样例代码都有关于手势识别器和事件处理的代码，其中包含以下项目：

x.《Simple Gesture Recognizers》：理解手势识别的完美开始。此APP展示了如何识别tap、swipe和rotate手势，并根据不同的手势对触摸处的图片进行展示和动画效果处理。

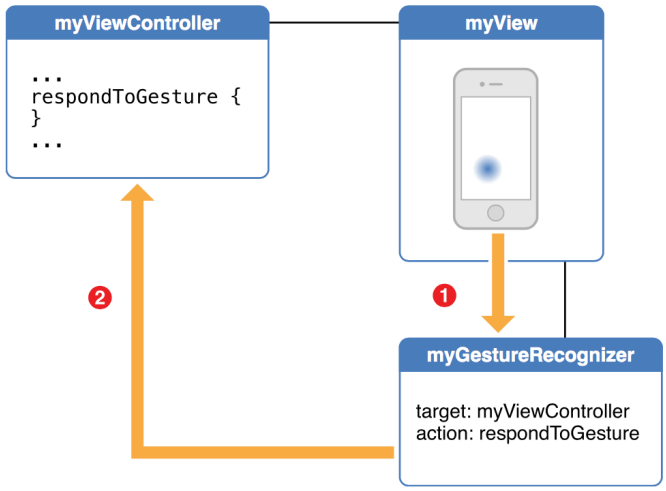
x.《Touches》包含了另个项目，展示了多点触摸和在屏幕内对矩形进行拖拽。一个版本使用了手势识别，另一个版本使用了自定义触摸事件处理方法。后面那个版本对于理解“触摸事件”尤其有意义，因为在触摸发生时直接在屏幕上对其进行展示。

x.《MoveMe》展示了当触摸事件发生时如何对图片进行动画处理。仔细阅读该项目可以帮助读者更深层的理解自定义触摸事件处理。

手势识别器（Gesture Recognizer）

手势识别器将底层的事件处理代码转换为高层的动作处理，开发者可以将其与视图类（view）进行绑定，从而使得视图类可以像控制器一样对这些动作进行响应。手势识别器识别用户的触摸动作，并确定是否和一些特定的手势动作相匹配，比如swipe（划）、pinch（捏）或者rotation（转）。如果识别器识别出来一致，就会向与之绑定的对象发送一条消息。目标对象通常是视图所在的视图控制器实例，从而对手势动作发生响应，如图1-1所示。这种设计模式简单而强大，开发者可动态的决定视图需要对什么样的手势动作进行响应，在不添加任何新的视图类的情况下，为任何视图实体类添加手势识别器。

Figure 1-1 A gesture recognizer attached to a view



1.使用手势识别器简化事件处理

UIKit框架提供内建的手势识别器来检测常见的手势动作。如果可能的话，使用内建的手势识别器是最佳选择，因为它们极大的减少了开发者的代码量。另外，使用标准的手势识别器而不是开发者自定义的可以保证APP完全按照用户期望进行响应。

如果开发者想自己的APP能识别特定的手势，比如对勾状或者旋涡状动作，开发者就可以自定义手势识别器，请参阅“创建自定义手势识别器”章节进行学习。

2.内建手势识别器所能识别的常见手势

在设计APP时就应该考虑APP会用到哪些手势，然后确定是否能够使用内建的某个手势，这样效率更高：

Table 1-1 Gesture recognizer classes of the UIKit framework

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`

APP应该以用户期望的方式对手势进行响应。比如，pinch应该是放大或缩小，而tap应该是选中某物，请参阅《iOS Human Interface Guidelines 》中的“Apps Respond to Gestures, Not Clicks ”章节。

3. 将手势识别器绑定到视图

每个手势识别器都只和一个视图进行绑定，而一个视图，可以绑定许多不同的手势识别器，因为一个视图可能对多个不同的手势进行响应。如果开发者想要某个视图能识别某个手势，开发者就必须将对应的识别器绑定到该视图。当用户触摸该视图时，手势识别器就会在视图之前先接收到该消息，结果就是，手势识别器代替视图来响应用户的触摸动作。

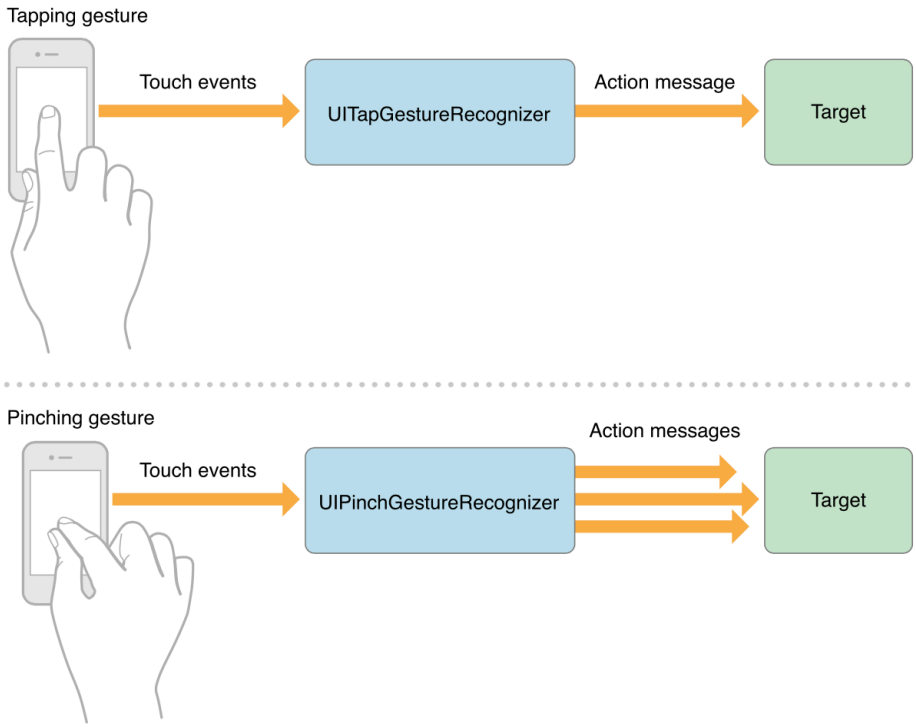
4. 手势触发动作消息

当手势识别器识别到指定的手势，就会给绑定的目标对象发送一条消息。开发者创建手势识别器后，需要为其初始化一个目标对象和消息发送之后的动作处理。

5.间断的和连续的手势

手势动作要么是间断的要么是连续的。间断的手势，比如 tap（点击），每次一下。连续的手势，如 pinch（捏合），就是连续的一个时间段的动作。对于非持续的动作手势，手势识别器会给绑定的目标对象发送一次单个的动作消息。而连续手势动作识别器，则会连续不断的给绑定的目标对象发送动作消息，直到多点触摸动作终止，如图 1-2 所示：

Figure 1-2 Discrete and continuous gestures



6.使用手势识别器对事件进行响应

1. 创建并配置手势识别器实例。

   这一步包括为其绑定消息发送目标，目标动作，以及为其分配制定手势识别标记（比如，tap的次数）。
2. 将手势识别器和视图进行绑定。
3. 实现对应的动作方法，对手势进行处理。

7.使用 Interface builder 为 APP 添加手势识别器

在XCode的Interface builder中，添加手势识别器的方式和为用户界面添加其他类一样，从

对象库当中直接拖动一个手势识别器的类到视图中就可以了，这样就可以直接将其和对应的视图类绑定了。开发者可以自行检查对应的绑定是否正确，有必要的话，就修改一下关联的nib文件。

开发者创建手势识别器之后对象之后，就需要穿件并关联动作方法。一旦对应的手势动作发生了，该方法就会被调用。如果想要将此手势动作关联到此动作方法之外的对象，就需要将此识别器和外部的目标outlet进行关联。对应的代码类似：

Listing 1-1 Adding a gesture recognizer to your app with Interface Builder

```
@interface APLGestureRecognizerViewController ()
@property (nonatomic, strong) IBOutlet UITapGestureRecognizer *tapRecognizer;
@end
@implementation
-(IBAction)displayGestureForTapRecognizer:(UITapGestureRecognizer*)recognizer
     // Will implement method later...
}
@end
```

8.纯代码的方式添加手势识别器

开发者可以使用纯代码的方式创建并初始化一个UIGestureRecognizer的子类实例，比如UIPinchGestureRecognizer，之后为之指定一个target和需要执行的action就可以了。例如代码清单1-2。通常，target可以指定为视图所在的视图控制器。

创建完手势识别器的时候，需要使用addGestureRecognaizer:方法将其与视图进行绑定,代码清单1-2创建了一个tap手势识别器，并且指定“点击一下”作为手势识别标记，然后将其与视图进行绑定。如代码清单1-2中所示，一般来说视图控制器的创建和初始化等操作是在视图控制器的viewDidLoad方法中。

Listing 1-2 Creating a single tap gesture recognizer programmatically

```
- (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
  }
```

9.对非持续的手势进行响应

创建好手势识别器的时候通常与之关联一个动作处理方法（action），这个处理方法就是用来响应用户的这个手势识别动作的。代码清单1-3就是一个简单的例子。当用户点击了绑定好手势识别器的视图时，该视图会弹出一个带图片的提示框，向用户反馈其“Tap”视图的消息。方法showGestureForTapRecognizer方法，利用识别器的locationsInView属性确定手势动作发生的位置，然后显示位于这个位置那张图片信息。

Listing 1-3 Handling a double tap gesture

```
  - (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;
         }];
}
```

每一种不同的手势识别器都有各自的不同属性，比如代码清单1-4，showGestureForSwipeRecognizer:方法就是用了swipe（划）手势的direction（方向）属性来确定用户是向左还是向右“划”。然后，将图片按照属性值的方向将图片进行渐变消失。

Listing 1-4 Responding to a left or right swipe gestu

```
// 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;
      }];
}
```

10.对连续的手势进行响应

连续的手势识别可以让App对正在进行的连续手势操作（比如拖拽等）进行连续响应。比如，当用户用两根手指对应用某个可见实体在屏幕内进行捏合缩放、拖拽时进行同步运动等。

代码清单1-5 展示了 将图片按照手势的旋转角度进行旋转，当用户停止旋转的时候，动态地将图片旋转至水平位置、同时渐变消失。当用户旋转手指的时候，方法showGestureForRotationRecognizer方法就被持续调用，直到手指全部离开屏幕。

Listing 1-5 Responding to a rotation gesture

```
// 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;
      }];
      }
}
```

每当该方法被调用，图片都会在方法drawImageForGestureRecognizer:中被设置为不透明，手势操作结束的时候，图片就会在方法animatedWithDuration:中被设置为透明。方法showGestureForRotationRecognizer通过手势识别器的状态来确定手势操作是否结束。所有这些状态在后续的章节“有限状态机中手势识别操作”会有详细介绍。

定义手势识别如何交互

通常来讲开发者添加手势识别的时候，也需要指定这些识别器相互之间如何交互、或者如何与其他触摸处理事件代码进行协调，所以此时开发者首先就必须理解这些手势识别器的工作原理。

1.有限状态机中手势识别的操作

手势识别器会以我们预想的方式从一个状态向另一个状态进行跳转。识别器所在的每一个状态，都会依照它们所符合的特定条件向另一个可能的状态进行跳转。具体的状态机回根据手势识别器是非持续或者连续的不一样不一而足，如图1-3所示。所有的状态识别器都会有一个初始的状态（比如UIGestureRecognizerStatePossible），然后分析其所接收到的所有的多点触摸序列，分析的过程中， 对应的手势要么成功识别、要么失败。如果失败，那就意味着将手势识别器的状态转向“失败状态”(UIGestureRecognizerStateFailed).

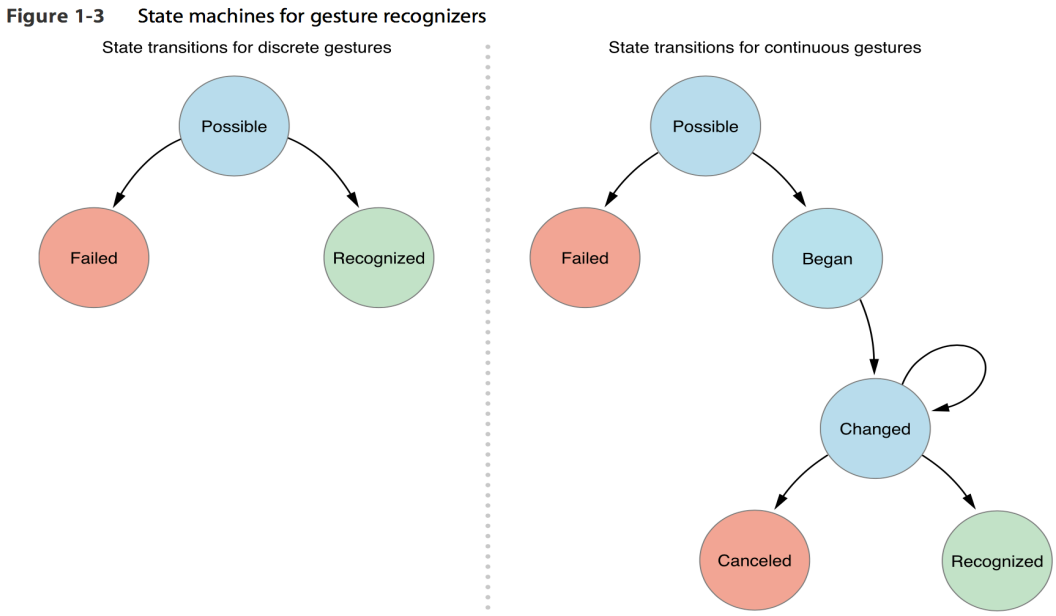


Figure 1-3 State machines for gesture recognizers

对于非持续手势识别，如果是被成功，就会从某个状态转向 “已识别” 状态 UIGestureRecognizerStateRecognized，也就意味着手势分析的整个过程完成。
对于持续手势识别，一旦识别成功，则会首先从某个状态转向 “已开始” 状态（UIGestureRecognizerStateBegan），当手势持续操作并运动时就会由 “已开始” 转向 “已变化” UIGestureRecognizerStateChanged 并持续由该状态继续，当用户的最后一根手指离开视图进行操作，就会转入 “已结束” 状态 UIGestureRecognizerStateEnded，该 “手势” 识别过程也就结束。需要注意的是，结束状态也是手势识别状态的一个。

如果连续手势识别发现用户的手势不符合预期的模式，其状态也可能从“已改变”状态转向“已取消”状态（UIGestureRecognizerStateCancelled）。手势识别器的状态每次发生改变，都会向其目标对象发送一条消息，除非其状态转为“失败”或者“取消”。如此，非持续的手势识别器的状态发生转变，就发送一个消息给其目标对象；连续手势识别器，则会连续发送许多消息。

当手势识别器状态转为“已识别”或者“已结束”时，其状态值就会被重置到初始状态，转向初始状态时不会触发消息发送。

2.与其他手势识别器进行交互

同一个视图对象可以绑定多个手势识别器，使用其 gestureRecognizers 属性可以知悉其被绑定了哪些手势识别器。当然开发者也可以分别使用 addGestureRecognizer 和 removeGestureRecognizer 方法动态增加或者减少绑定的手势识别器，来改变视图对手势操作的识别处理。

当视图被绑定了多个手势识别器时，开发者可能想要对最终处理这些手势的识别器进行修改。默认情况下，哪个手势识别器去接收第一个触摸操作并没有特定的先后顺序，所以每次用户的手势触摸操作都可能由不同的手势识别器接受到并被处理了。所以开发者就想要修改此“默认”设定，已达到以下目的：

- 指定某个识别器优先于另一个识别器，来接收并处理用户的触摸操作。
- 让两个识别器同时进行操作处理。
- 阻止某个手势识别器对某个触摸操作进行分析。

使用被 UIGestureRecognizer 子类覆盖重写的的类方法、代理方法以及成员函数方法来改变这些行为操作。

2.1 申明两个识别器的特定顺序

假设开发者想要识别用户的“滑动”（swipe）手势和“平移”（pan）手势，而这两个手势需要触发两个不同的消息。默认情况下，当用户进“滑动”（swipe）操作时，这个手势会被默认识别为“平移”（pan）。这是因为“滑动”手势操作在被系统识别为“滑动”手势（这是一个持久的连续性手势）之前，系统进行判断时，发现其操作行为完全符合“平移”（这是一个瞬发的非持续手势）操作的所有必要的属性条件，所以系统就将其识别为“平移”。

（由此推断，iOS 系统手势识别，默认情况下，会优先考虑持久性的连续手势操作，一旦其状态符合连续手势操作判定条件，就将其判定为连续手势操作，而不会继续进行分析处理）

如果开发者想要其视图对象同时识别这两个手势，就需要先让“滑动”手势识别器先对手势操作进行识别，然后进行“平移”操作的识别。假如“滑动”手势确定用户的手势触摸操作为“滑动”，另外一个手势识别器就不需要继续分析了。如果“滑动”识别器分析发现用户手势操作不是非持续的“滑动”手势，就会转向失败状态，此时，平移手势就开始对用户操作进行分析了。

开发者指定两个不同的识别器之间的这种先后关系，需要调用手势识别器（就是那个需要放在后面进行手势分析识别器）的 requireGestureRecognizerToFail：方法，如代码 1-6：

Listing 1-6 Pan gesture recognizer requires a swipe gesture recognizer to fail

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

调用该方法的手势识别器 A 会给接收消息的手势识别器 B 发送一个消息，指定一定是 B 对手势操作识别失败之后，A 才开始接收并分析用户的手势操作。在 A 进行等待 B 对用户的手势操作进行分析并直到识别失败的过程中，A 的状态一直会处于某个可能初始状态，除非 B 到达了分析识别“失败”的状态，A 才开始分析并开始转向下一个状态。另一方面，如果 B 识别成功或者识别（成功）开始，A 就会被转向失败状态。关于状态之间的转换信息，请参阅“有限状态机中的手势识别操作”。

备注：如果开发者的 APP 对“单击”和“双击”手势都能识别，而且“单击”手势并不需要等待“双击”手势识别失败，那么开发者所预期的就应该是先处理识别“单击”手势，然后才是“双击”手势，（这也就意味着用户即使真的手指双击，会被识别为两次“单击”操作）。这是特意为了支持多个重复的快速“单击”动作行为，以达到最好的用户体验。
但是如果开发者就想让单击和双击两个动作区别对待，那就让需要让先让“双击手势”识别器进行分析，如果失败，再进行“单击”识别。但是这样一来的话，“单击”操作的识别相对而言就稍微会延迟一点，因为被识别成功之前，还要走一遍“双击”识别过程。

2.2 禁止手势识别器进行触摸分析

通过给识别器添加代理对象，开发者可以修改识别器的行为。协议 UIGestureRecognizerDelegate 提供了几个方法给开发者用来禁止手势识别器的触摸分析功能。gestureRecognizer:shoulReceiveTouch:和 gestureRecognizerShouldBegin:两个方法都可选用。

当用户“触摸”操作开始时，开发者可以使用 gestureRecognizer:shoulReceiveTouch:方法立即决定手势识别器是否需要处理用户的操作。每当“触摸”发生，该方法就会被调用。如果不想对用户的触摸操作进行处理直接将该方法返回“NO”即可，默认情况下返回的是“YES”。该方法并不会修改识别器的状态值。

代码清单 1-7 就是用了代理方法 gestureRecognizer:shoulReceiveTouch: 手势识别器接收来防止某个自定义视图内发生的“触摸”手势。当“触摸”操作发生，该代理方法被调用，然后判断“触摸”是否发生在该视图内，如果是的话，就返回“NO”，从而阻止识别器对手势操作事件的接收。

Listing 1-7 Preventing a gesture recognizer from receiving a touch

```
- (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;
}
```

假如开发者想要等一段时间再决定是不是要接收并处理分析该触摸事件，那就是用代理方法 gestureRecognizerShouldBegin:。通常来讲，对于 UIView 或者 UIControl 子类中的自定义触摸事件处理，开发者可以直接使用该方法，足以媲美使用手势识别器。该方法返回“NO”会直接让手势识别器转为“失败”状态，从而让其他的触摸事件处理继续执行。当手势识别器正在对手势操作进行识别、并即将由某个初始的状态转向下一个状态过程，该方法就会被立即调用，其返回值会决定手势识别过程是否需要继续执行下去，从而确定某个视图或者控制器是不是需要接收处理对应的手势操作事件。

如果某个视图或者视图控制器无法成为手势识别器的代理对象，开发者可以直接使用方法 gestureRecognizer:shoulReceiveTouch:，其方法签名和实现都是一样的。

2.3 允许手势识别同时异步进行

默认情况下，两个不同的手势识别器是不允许同时对手势进行识别处理的，但是假如开发者想要让用户在对某个视图同时进行捏合缩放和旋转操作，开发者就需要改变此默认设定，通过实现方法 gestureRecognizer:shouldRecognizeSimutaneouslyWithGestureRecognizer:可以达到此目的，这是由协议 UIGestureRecognizerDelegate 提供的一个可选方法。该方法会在某个识别器进行手势事件识别处理时被调用，从而决定是否需要锁定事件不让其他识别器进行识别处理。默认情况下，该方法返回“NO”，如果开发者想要两个不同的手势识别器同时对手势进行识别处理，让该方法返回“YES”即可。

备注：开发者只需要对其中一个手势的代理对象实现该方法并返回“YES”即可。这就意味着，两个手势识别器中只要有一个返回“YES”，另外一个返回“NO”就不起作用了。

2.4 指定两个手势识别器的单向关系

如果开发者想让两个识别器进行交互，但是指定为一种单向关系，开发者就可以通过重写方法 canPreventGestureRecognizer:或者 canBePreventedByGestureRecognizer:之一并返回“NO”（默认返回的是“YES”）来达到此目的。举个例子，如果想要在进行旋转操作的时候屏蔽掉捏合缩放操作、而在捏合缩放操作的时候可以进行旋转，开发者就可以这样设置：`[rotationGestureRecognizer canPreventGestureRecognizer:pinchGestureRecognizer];`

然后重写方法“旋转”手势识别器的子类方法并返回“NO”。关于如何创建继承 UIGestureRecognizer 子类的信息请参阅“创建自定义手势识别器”章节。

如果这两个手势识别器需要同时进行不相互干预，请参阅章节 2.3.默认情况下，这两个不同的手势识别器是不能同时进行手势识别操作的，也就是互相屏蔽的，其中任何一个处于活动状态，另外一个都是被屏蔽的。

与其他用户界面操作进行交互

在 iOS6 以及之后的版本中，默认情况下，所有的控制器都不允许手势识别器重复交叠。比如，按钮（button）的默认操作行为是“单击”，然后，有一个“按钮”所在的视图（view）绑定了一个“单击”手势识别器对手势进行处理，当用户“单击”这个按钮的时候，按钮指定的动作处理方法会接收到这个“单击”事件，然而视图的手势识别器却接收不到。这种策略仅应用于以下这些控制器所在视图绑定的手势识别器：

- *单根手指在 UIButton，UISwitch，UIStepper，UISegementedControl 和 UIPageControl 上的单击事件。*
- *单根手指在 UISlider 的滑块按钮上并且和 UISlider 控件平行方向上的滑动操作。*
- *单根手指在 UISwitch 滑块按钮上并且和 UISwitch 控件同水平方向上的平移操作。*

如果开发者的自定义的空间子类继承自以上控件，又想要改变系统的默认设定，那就应该直接将手势识别器和其子类控件进行绑定，而不是和控件坐在的视图或者其他上层控件进行绑定。这样的话，手识别器就会优先接收到用户的触摸操作事件。一定谨记，请预先阅读《iOS Human Interface Guidlines》（“iOS 用户交互指南”），以保证 APP 拥有自然而良好的用户体验，尤其是在修改某个标准控件的默认行为的时候。

手势识别器处理原生触摸事件

目前为止，我们已经讨论了“手势”以及 APP 应该如何对其进行识别并处理。但是，要创建一个自定义的手势识别器、或者控制手势识别器如何与视图的触摸事件进行交互处理，还需要站在“手势”和“事件”的角度进行更详尽的思考。

1.一个事件包含了所有当前多点触摸序列的“触摸”行为对象

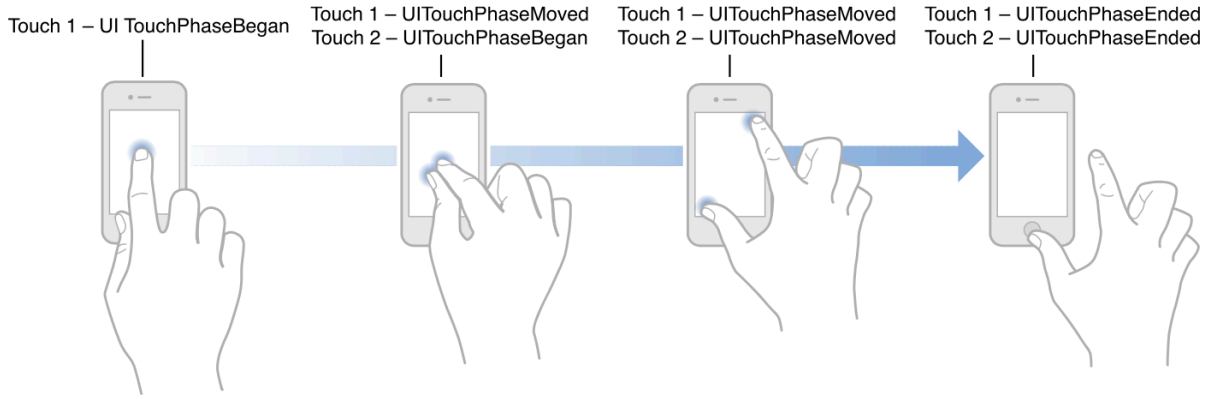
在 iOS 中，每一个“触摸”（touch）行为对象就代表单根手指在屏幕上的一次运动操作。一个“手势”（gesture）可以有一个或者多个“触摸”行为对象，在 iOS 中以 UITouch 类对象进行抽象表示。例如，一个捏合缩小手势就有两个“触摸”行为对象：两根手指在屏幕上以相反运动方向相互靠拢运动。

一个事件（Event）包含了这次多点触摸行为序列的所有“触摸”对象。一个多点触摸行为序列开始于用户第一根手指触摸到屏幕、终止于用户的最后一根手指抬起并离开屏幕。当一根手指进行运动时，iOS 系统就会实例化 UITouch 对象并发送给对应的事件对象。一个多点触摸事件会被抽象成一个类型为 UIEventTypeTouches 的 UIEvent 对象。

每一个“触摸”对象都只追踪一根手指的轨迹，其生命周期也仅限于整个触摸序列的起止期间。在此这段时间内，UIKit 跟踪手指的轨迹并及时更行其对象的属性。这些属性包括触摸的行为的方式、在当前视图对象中的位置、之前的位置以及时间戳。

触摸行为的方式指的是触摸行为的开始（运动或者静止）和结束（也就是手指离开屏幕了）。如图 1-4 所示，App 在任意触摸行为方式的过程中接收到每一个触摸事件。

Figure 1-4 A multitouch sequence and touch phases



备注：一根手指相对于鼠标就没有那精确了。当用户触摸屏幕时，接触的区域实际上是个椭圆形的，且精确度往往比用户预期的稍稍要更低一些。“接触面”大小会因手指的大小和方位、按压的力道、哪根手指以及其他一些因素不一样而变化。设备内置的多点触摸系统会分析这些信息，并计算单个“触摸”点，从而开发者不需要自己编写代码进行处理。

2.App 在 “触摸处理” 方法中接收 “触摸” 对象

再一次多点触摸序列事件中，当有新的触摸行为或者触摸行为发生变化时，app 会发送以下消息：

- *touchesBegan:withEvent:当一个或者多根手指开始触摸屏幕时调用。*
- *touchesMoved:withEvent:当手指开始移动时调用。*
- *touchesEnded:withEvent:当有手指离开屏幕时调用。*
- *touchedCancelled:withEvent:当触摸序列事件被系统事件取消时调用，比如来了电话。*

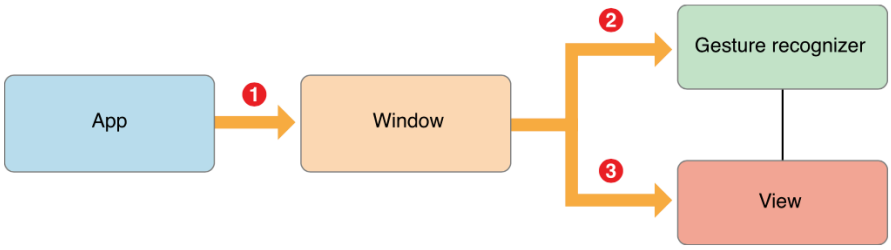
以上每一个方法都和 “手势触摸行为” 对象相关连，比如第一个方法会和 UITouchPhaseBegan 相关连。对应的触摸行为对象存储在 UITouch 的 phase 属性中。

备注：这些方法和手势识别器的状态没有任何关联。手势识别器的状态严格意义上来说仅仅只表示了手势识别器本身的状态信息，而不是其分析识别出来的触摸对象的行为方式。

调节系统向视图对象发送触摸事件信息

有时候开发者可能想要在手势识别器之前，先让视图对象接收触摸事件。但是，在开发者想要修改视图上的触摸事件传输路径之前，首先得理解事件传输默认的行为。举一个简单的实例，当触摸事件发生时，UITouches 触摸对象会先由 UIApplication 对象传递给 UIWindow 对象，然后，在传递给最底层的视图对象之前，UIWindow 对象会逐层向下，将“触摸”对象传递给触摸事件发生位置所在的视图对象所绑定的手势识别器进行识别处理。

Figure 1-5 Default delivery path for touch events



1.手势识别器拥有第一个识别处理触摸事件的机会

window 对象会延迟将“触摸”对象发送给视图对象，从而让手势识别器最先对“触摸”进行分析处理。在延迟期间，如果识别器识别出来触摸手势，window 对象就不会将“触摸对象”传递给视图对象了，并将识别出来的手势序列中其他之前本该发送给视图对象的触摸对象取消掉。

举个例子，假如有一个手识别器用来识别非持续的“双击”操作，这个操作会被转换为两个 UITouch 对象。当触摸事件发生时，这两个对象首先会被传递（触摸事件发生所在的）视图对象对应的 UIApplication 对象，然后就会触发以下序列消息，如图 1-6:

Figure 1-6 Sequence of messages for touches

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

1. 在开始（Began）阶段，window 通过方法 touchedBegan:withEvent:将两个触摸对象发送给手势识别器，手势识别器还没有开始进行识别处理，其状态值为 Possible。window 将同样的 UITouch“触摸”对象发送给视图绑定的手势识别器。
2. 在移动（Moved）阶段，window 通过 touchesMoved:withEvent: 将两个触摸对象发送给手势识别器，手势识别器依然没有检测到手势操作，其状态值为 Possible。window 将这些的 UITouch“触摸”一个对象发送给视图绑定的手势识别器。

3. 在某根手指离开屏幕时，就有一个 UITouch 对象处于结束（Ended）阶段，window 通过 touchesEnded:withEvent: 将该触摸对象发送给手势识别器，手势识别器还没有获取到足够的信息来处理手势操作，其状态值为 Possible。window 自己保留从对应的视图上抓取的“触摸”对象。
4. 在另一根手指也离开屏幕，另外一个 UITouch 也处于结束（Ended）阶段，window 将此 UITouch 对象也发送出去。此时，手势识别器（收集到了足够的信息）成功识别到手势。就在第一个动作消息发送出去之前，视图对象就调用 touchesCancelled:withEvent:方法，就作废掉了 Began 和 Moved 阶段发送过来的触摸对象。而在 Ended 阶段的触摸操作也就被取消了。

现在假定某个手势识别器在最后一步才发现其所分析处理的多点触摸手势无法识别的话，手势识别器就会将自己的状态设置为 UIGestureRecognizerStateFailed.然后 window 对象再会在 touchesEnded:withEvent:消息中，将这两个触摸操作对象在结束阶段发送给绑定的视图对象。

对于连续性手势操作的手势识别器也遵照相同的步骤，尽管看起来貌似在触摸对象到达结束（Ended）阶段之前就能识别出来对应的手势。在识别出对应的手势之前，手势识别器先将自己的状态设置为 UIGestureRecognizerStateBegan（不是 Recognized）。window 会将这次多点触摸事件中后续的所有触摸对象都发送给识别器，而不是被绑定的对象。

2.修改系统向视图对象进行触摸事件的发送行为

开发者可以修改多种 UIGestureRecognizer 的对应属性值，从而修改事件特定的传递路径。如果修改了这些属性的默认值，在事件处理行为上会出现以下差异：

- delaysTouchesBegan(默认值是 NO)。通常情况下，window 会在 Began 和 Moved 阶段将触摸事件发送给 view 和手势识别器对象。如果将此属性值设置成 NO，window 就不会在 Began 阶段将“触摸”（UITouch）对象发送给视图对象，这样可以保证当手势识别器识别到某个手势时，就不有任何相关的 UITouch 对象被发送给绑定的视图。注意对该属性进行设置时，可能会让界面看起来没有什么被“触摸”的视觉效果。

  该属性值的设置类似于 UIScrollView 的 delaysContentTouches 属性；在这种情况下，UIScrollView 就立即随着用户“触摸”动作的进行滚动，而不会将“触摸”（UITouch）对象发送给 SrollView 的子视图对象，所以也就不会有视觉上的反馈效果。

- delaysTouchesEnded(默认值为 YES)。当该属性被设置成 YES 时，可以保证视图对象的动作处理不会结束，这样一来该手势动作还有机会被取消。当手势识别器对触摸事件进行分析时，window 对象不会将 UITouch 对象发送给在 End 阶段发送给绑定

的视图。如果识别器成功识别出来手势操作，UITouch 对象会被取消掉；若是识别失败，window 对象就会将它们通过消息 touchesEnded:wihtEvent:发送给视图对象。如果将该属性值设置成 NO，就会把这些“触摸”（UITouch）对象发送给手势识别器的同时，也发送给视图对象进行分析处理。

设想一下，某个视图对象 view 有一个“点击”（tap）手势识别器可以识别“双击”手势，而用户也刚好双击了。该属性值为 YES 时，视图对象按顺序收到以下几个消息方法都会被调用 touchesBegan:withEvent:, touchesBegan:withEvent:,touchesCancelled:withEvent:, touchesCancelled:withEvent:,如果被设置成了 NO，对应按顺序收到的消息序列就变成：touchesBegan:withEvent:, touchesEnded:withEvent:, touchesBegan:withEvent:, touchesCancelled:withEvent:,也就意味着，在 touchesBegan:withEvent:消息处理事件中，视图对象就可以识别出来双击”手势事件。

如果手势识别器检测到对应的“触摸”（UITouch）对象不属于对应“手势”操作的一部分，该对象就会被直接发送给对应的视图对象。要达到这种效果，识别器会调用 ignoreTouch:forEvent: 方法消息，把对应的“触摸”对象传递出去。

创建自定义手势识别器

要实现自定义的手势识别器，首先在 XCode 中创建一个 UIGestureRecognizer 的子类。然后在子类的头文件中添加代码：

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

然后，从 UIGestureRecognizerSubclass.h 文件中复制下面的这些方法到刚刚创建的子类的头文件中：

```
- (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;
```

这些方法和章节“App 在“触摸处理”方法中接收“触摸”对象”中对应描述的触摸事件处理方法具有完全一样的签名和行为。所有这些开发者重写覆盖的方法时，即使父类对应方法的实现是空的，都注意要先调用一下父类的实现。

另外一点就是，此时 UIGestureRecognizerSubclass.h 的属性 state 是“可读写”的（readwrite）的，而不是“只读”readonly。开发者可以使用 UIGestureRecognizerState 常量值对该属性进行赋值操作。

1.为自定义手势识别器实现触摸事件处理方法

实现自定义手势的中心点就在于实现以下四个方法：touchesBegan:withEvent:, touchesMoved:withEvent:, touchesEnded:withEvent:, and touchesCancelled:withEvent:. 子啊这些方法的实现中，开发者就可以通过改变手势识别的状态值，将底层的触摸手势事件进行识别处理。代码清单1-8，创建了一个非持续手势单指触摸的对勾手势识别器。通过记录手势操作的中心点、亦即向上勾动的转折位置点，所以就可以让外部捕获该坐标点的值。

Listing 1-8 Implementation of a checkmark gesture recognizer

```
#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;
}
```

连续动作手势的识别和非持续动作手势的识，识别器的状态值的转变是不一样的，在章节“有限状态机中手势识别的操作”中有介绍。自定义手势识别器时，通过给其状态state属性赋予相应的值，来指定手势识别对应的连续的还是非持续的手势。如上述1-8的清单中，对勾手势作为一个非持续手势，就不要给其状态赋予Began或者Changed状态值了。

因此，在进行创建手势识别器的子类进行自定义手势识别时，最重要的一点就是准确为其状态属性state赋予准确的值。iOS系统需根据此状态属性，确保手势识别器能够按照预期

对手势进行识别。假如开发者想要使用同步识别、或者想要某个识别器不进行识别处理，iOS系统就需要获知开发者自己创建的手势识别器的状态。

欲知更多如何自定义手势识别器知识，请观看《WWDC 2012: Building Advanced Gesture Recognizers》。

2.重置手势识别器的状态

如果手势识别器的状态值转为 Recognized（识别成功）/ Ended（结束识别），Canceled（取消），UIGestureRecognizer 在其状态回滚到 Possible 初始状态前，会调用 reset 方法消息。

通过实现 reset 方法，将所有的内部状态重置，以便手势识别器可以用于识别用户下一次手势操作，如代码清单 1-9，一旦手势识别器从该方法返回之后，就不再对后续的触摸操作进一步更新处理了。

Listing 1-9 Resetting a gesture recognizer

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

事件传递：响应链

开发者设计 APP 时，通常都想要对一些事件进行动态响应。比如，触摸事件可以发生在屏幕上的许多不同的对象上，开发者就需要确定到底想要哪个对象对哪个事件进行响应，并理解该对象是如何接收该事件的。

当用户触摸事件发生时，UIKit 框架就会为此创建一个事件对象，该对象就包含了能够处理该事件对象所必要的信息。然后 UIKit 框架将此对象列入 active app 的事件队列。对于触摸事件，事件对象就是组装的一系列 UIEvent 对象。对于动作事件，所对应的事件对象因开发者所使用的框架、开发者感兴趣的动作事件类型不同而异。

每一个事件都会沿着特定的路径传递下去，直到某个对象能处理为止。首先，单例对象 UIApplication 会从事件队列的顶端取出来一个事件并进行派遣，以便开始处理。通常，该事件会被派送给 app 的主窗体（window）对象，然后由此 window 对象将事件传递给最初事件发生所在的“初始对象”进行处理，这个初始对象是什么，就依赖于实事件的类型：

- Touch Event（触摸事件）。对于触摸事件，主窗体对象首先尝试将事件传递给事件发生所在的视图 view 对象。该视图对象 view 就是所谓的 hit-test 视图对象。这个寻找 hit-test 视图对象的过程被称作 hit-testing，在章节“Hit-Testing 返回触摸事件发生所在的视图对象”有介绍。

- Motion and remote control events（运动和远程控制事件）。对于这些事件，主窗体对象会将摇晃事件或者远程控制事件发送给第一响应器（the first responder）进行处理。第一处理器在章节“响应器链由多个响应器对象组成”。

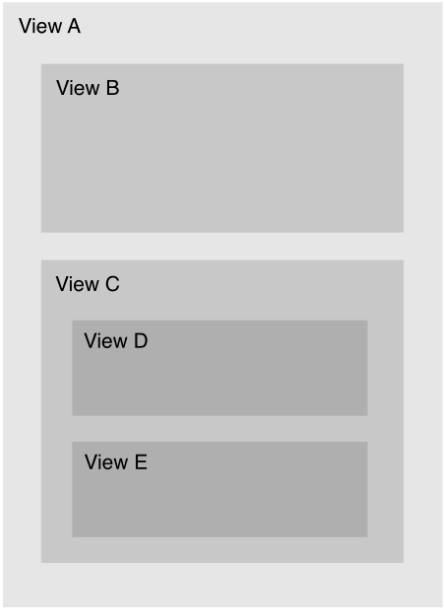
事件对象路径的最终目标，就是找到能够处理事件并进行响应的对象。因此，UIKit 先会将事件发送给最适合处理该事件的对象。对于触摸事件，该对象就是 hit-test 视图对象，对于其他的事件，该对象就是第一响应器。以下部分会更详细地对 hit-test 视图对象和所确定的第一响应器进行解释。

1.Hit-Testing 返回触摸事件发生所在的视图对象

iOS 系统使用 Hit-Testing 去查找到底触摸事件发生在哪个视图对象上。Hit-Testing 先检查触摸对象所在的位置是否在对应任意屏幕上的视图对象的区域范围内。如果在的话，就开始对此视图对象的子视图对象进行同样的检查。视图树中最底层那个包含此触摸点位置的视图对象，就是要查找的 hit-test 视图对象。iOS 一旦确定 hit-test 视图对象，就会把触摸事件传递给它进行处理。

举个例子，假设用户触摸了视图 E，如图 2-1 所示。iOS 就会按照以下顺序对子视图进行检查来查找 hit-test 视图：

Figure 2-1 Hit-testing returns the subview that was touched



1. 触摸点在视图 A 的区域范围内，然后开始检查子视图 B 和 C
2. 触摸点不在 B 的范围而在 C 的范围，于是就开始检查 D 和 E 视图
3. 触摸点不在 D 的范围而在 E 的范围，而 E 视图是视图树最底层的并包含触摸点的视图对象，所以 E 就成为了 hit-test 视图。

hitTest:withEvent 方法会返回给定的 CGPoint 和 UIEvent 所在的 hit-test 视图对象。hitTest:withEvent:方法会先调用 pointInside:withEvent:方法。如果传入 hitTest:withEvent:的 CGPoint 点对象位于视图对象的区域范围内，pointInside:withEvent:返回值就是 YES，然后，该 hitTest:withEvent:就会依次在返回 YES 的子视图对象上调用 hitTest:withEvent:。

如果传入 hitTest:withEvent:的点不在视图对象的范围内，第一次调用 pointInside:withEvent:就会返回 NO，这个点就被忽略掉了，hitTest:withEvent:就返回 nil。如果子视图返回 NO，那么整个视图树的分支都会被忽略掉，因为如果子视图不包含这个点，那子视图的子视图就更不会包含这些点了。这也就意味着，如果父视图都不包含某个触摸事件的点，子视图即使包含了这个点，也不会接收到此触摸事件，因为只有父视图的区域范围包含了触摸事件发生的位置点，事件才会被继续向子视图传递。如果子视图的 clipsToBounds 属性被设置为 NO（亦即子视图超出父视图的部分不会被切割掉，也就是说子视图会有一部分不处在父视图的范围内），这种情况是会发生的。

备注：触摸对象 UITouch 在其生命周期内会和 hit-test 视图对象一直关联在一起，即使 UITouch 在后续的时间里移动并离开该视图对象的范围。

hit-test 视图对象拥有最先对触摸事件进行处理的机会，如果 hit-test 视图对象无法处理该事件，事件对象就会沿着响应器的视图链（参见“响应器链由多个响应器对象组成”）向上传递，直到找到最适合处理该事件的对象为止。

2. 响应器链由多个响应器对象组成

许多类型的事件都依赖于响应器链（responder chain）进行事件传递。响应器链就是一系列的相关联的响应器对象。如果第一个响应器无法处理事件，响应器就会将事件对象传递给响应器链的下一个响应器对象。

一个响应器对象就是一个能够对事件进行处理和响应的实体对象。UIResponder 类是所有响应器对象的基类，不仅定义了事件处理的编程接口，同时还定义了通用的响应器行为。UIApplication、UIViewController、UIView 类的实体对象都是响应器，也就意味着，所有的视图对象和关键控制器对象都是响应器对象。注意 Core Animation layers 不是响应器。

第一响应器被指定第一个接收事件。通常来讲，第一响应器是一个视图 view 对象。通过做两件事，一个对象就变成了第一响应器：

1.重写 canBecomeFirstResponder 使其返回 YES；
2.接收 becomeFirstResponder 消息。如果有必要，对象本身可以自己发送此消息。

备注：再将某个对象赋值为第一响应器之前，一定要确保 APP 已经建立好了对象图谱。比如，通常应该在重写的 viewDidAppear:方法中调用 becomeFirstResponder 方法，但是如果写在了 viewWillAppear 里面，此时因为对象图谱还没有建立起来，becomeFirstResponder 的返回值就 NO 了。

也不仅仅只是事件对象依赖于响应器链，响应器链可以被用于处理以下所有对象：

- **Touch Events**（触摸事件）。如果 hit-test 视图对象无法处理触摸事件，事件就会从 hit-test 视图沿着响应链网上传递，直到找到合适的处理该事件的对象。

- **Motion Events**（运动事件）。要使用 UIKit 处理“摇动”（shake-motion）事件，第一响应器就必须实现方法 motionBegan:withEvent:或者 motionEnded:withEvent:之一，具体请参见“使用 UIEvent 检测摇动事件”章节。

- **Remote Control Events**（远程控制事件）。要对远程控制事件进行处理，第一响应器必须实现基类 UIResponder 的 remoteControlReceivedWithEvent:方法。

- **Action messages**（动作消息）。当用户操作了某个控件，如按钮 button、switch，对应的动作方法的目标是 nil，该消息会从以控件视图对象为开始的响应器链被发送出去。

- **Editing-menu messages**（编辑菜单消息）。当用户点击了编辑菜单的指令，iOS 系统就会使用响应器链去查找到对应实现了必要处理方法（如 cut:,copy:以及 paste:）的对象。要获得更多信息，请参阅章节“显示和管理编辑菜单”和样例代码项目 CopyPasteTile。

- **Text Editing**（文本编辑）。当用户点击某个文本区域（UITextField）或者文本视图（UITextView）时，对应的视图就会成为第一响应器。默认情况下，虚拟键盘会弹出来，而且对应的 UITextField 或者 UITextView 就会被选中并变成正在编辑状态。如果开发者觉得合适的话，可以使用自定义的视图来代替默认的软键盘作为用户的输入区域视图。要获取更多信息，请参见章节“自定义数据输入视图”。

当用户点击某个 UITextField 或者 UITextView 的时候，UIKit 会自动把将对应的对象设置为第一响应器。对于其他的第一响应器，App 必须使用 becomeFirstResponder 方法显示地进行设置。

3.响应器链的特定传递路径

如果（事件发生所在的）初始对象（要么是 hit-test 视图，要么是第一响应器）无法对事件进行处理，UIKit 就会把事件传递给响应器链的下一个响应器对象。每个响应器对象都可以决定是自己进行事件处理，还是将事件通过方法 nextResponder 的调用，传递给下一个事件响应器。此过程一直进行下去，直到找到了处理该事件的对象，或者到达了响应器链的最后一个响应器了。

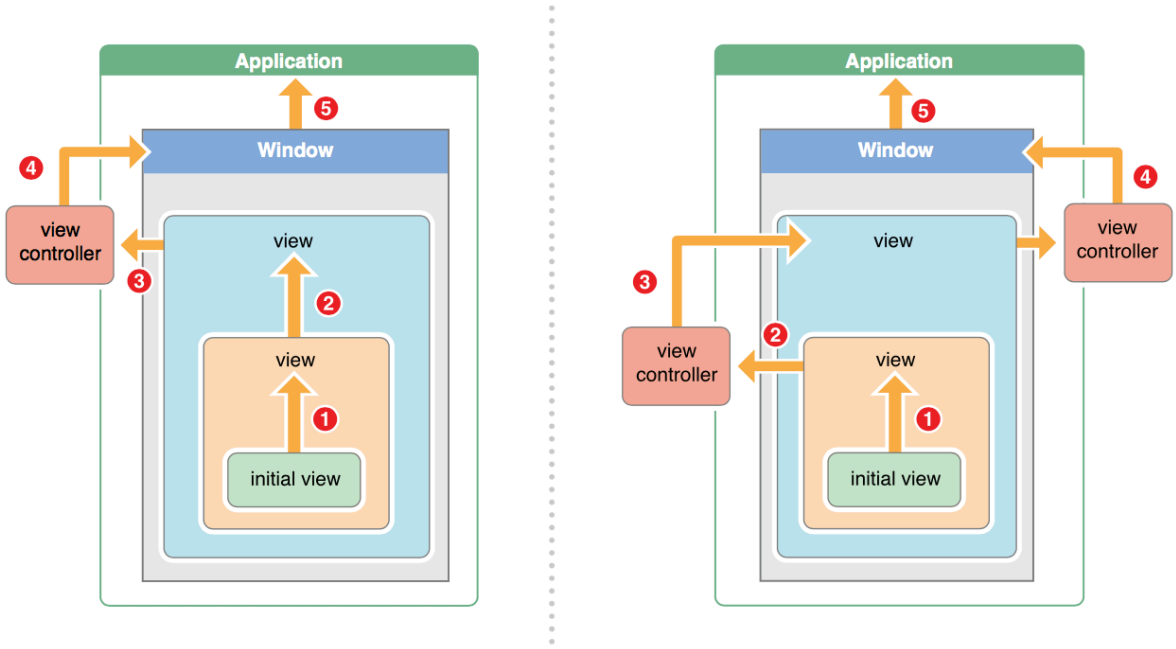
响应器链开始于 iOS 检测到事件并将其传递到（事件发生所在的）初始对象，通常来讲这个对象是一个视图对象 view。初始视图对象会最先有机会对事件进行处理。如图 2-2 所示，就是两个不同的 app 中事件的不同的两条事件传递路径。App 的事件传递路径由其特定的结构所决定，但所有的事件传递路径都遵循同样的逻辑方法。

左边 APP 的事件传递路径如下：

1. 初始视图对象尝试对事件进行处理，如果无法处理，就会将事件传递给其父视图对象，因为视图树中，初始视图对象也并不是最顶端的对象。
2. 父视图也进行同样的尝试，因为同样的原因也只能将事件继续向上传递。
3. 视图控制器中最顶层的视图也进行同样的尝试，结果发现也处理不了，于是就传递了视图控制器。
4. 视图控制器也一样无法处理，于是继续向上传递给了主窗体对象（window）。

5. 主窗体也无法处理，于是就继续传递给 app 的单例实体对象。
6. 如果最后单例实体对象还无法处理，此事件就被丢弃了。

Figure 2-2 The responder chain on iOS



虽然右边的 APP 传递路径略微不一样，但是事件传递遵循的逻辑方法还是一样的：

1. 视图将事件沿着其视图控制器的视图树向上传递，直到最顶端的视图。
2. 顶端是图无法处理，就直接传递给视图控制器。
3. 视图控制器无法处理，就会将事件传递给其顶端视图所在的父视图。重复 1-3，直到到达最顶端的根视图控制器（root view controller）。
4. 跟视图控制器将事件传递给主窗体对象。
5. 主窗体对象传递给 app 的单例实体对象。

重点注意：如果开发者自己实现某个视图对象，来处理远程控制（remote control）事件、动作消息（action message）、使用 UIKit 的摇动（shake-motion）事件，或者编辑菜单消息，千万不要直接将事件或者消息发送给 nextResponder，来达到将事件沿着响应器链向上传递的目的，而是实现父类的当前事件处理方法，让 UIKit 框架来完成响应器链的事件消息传递。

多点触摸事件

通常，开发者可以使用 UIKit 提供的标准控件和手势识别器来处理大部分的触摸事件。手势识别器允许开发者将触摸事件的识别和对应的动作处理区分开来。在某些情况中，有可能开发者想要根据触摸事件做一些其他事情，比如触摸位置进行绘画，要实现这样的效果，重新实现一次这样的触摸识别处理，当然没有什么好处。如果视图的内容和触摸事件本身紧密相关，开发者可以直接对事件进行处理。当用户触摸视图对象时，对触摸事件进行接收，基于事件的属性对时间进行识别处理，并作出合适的响应。

1.创建 UIReponder 子类

要实现自定义触摸事件处理，首先就要创建一个 UIResponder 子类。该子类对象可以继承自以下的任一对象类：

子类对象	选择该子类作为第一响应器的原因
UIView	用于实现自定义绘制视图
UIViewController	想要处理其它类型的事件，比如“摇动”（shake-motion）事件
UIControl	想要实现自定义触摸行为的控件
UIApplication 或者 UIWindow	继承实现这两个类，这就很少见了，因为一般人是不会这么干的。

然后，针对继承后的子类接收多点触摸事件：

1. 子类必须实现 UIResponder 的触摸事件处理的方法，请参阅章节“在子类中实现触摸事件处理方法”。
2. 视图对象要接受触摸事件，必须将其 userInteractionEnabled 属性值设置为 YES。如果子类继承自 UIViewController，其管理的视图对象就必须支持用户交互。
3. 接收事件的视图对象必须是可见的，不能是纯透明或隐藏的。

2.在子类中实现触摸事件处理方法

iOS 将触摸对象识别为多点触摸序列的一部分。在一次多点触摸序列中，app 会将这一系列的事件消息发送给目标响应器。要接收并处理这些消息，响应器对象类就必须实现以下几个 UIResponder 的方法：

```
- (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;
```

备注：这些方法和自定义创建一个手势识别器时所需要实现的方法，拥有同样的签名，在章节“创建自定义手势识别器”中有介绍。

每一个触摸方法都相应的对应触摸事件的一个状态：开始（Began），移动（Moved），结束（Ended）和取消（Canceled）。每个方法都有两个参数：触摸事件集合序列和对应的事件。

触摸事件集合序列就是使用 NSSet 容器承载的 UITouch 对象，用于代表当前阶段对应的新的或发生变化的触摸对象。举个例子，当某个“触摸”对象从 Began 状态转向 Moved 状态，app 就会调用 touchesMoved:withEvent:方法，被传入该方法的触摸对象序列集合，就会包含此触摸对象和其他同样处于移动状态的触摸对象。另一个参数是包含了事件发生对应的 *所有* 触摸对象的 UIEvent 对象。传入的触摸对象序列集合会有所不一样，因为上一个事件消息之后，可能有一些触摸对象的状态并没有发生变化（也就是没有移动过）。

所有处理触摸事件的视图对象都期望能接收到完整的触摸事件流，所以开发者创建自己的子类时要注意以下几条规则：

- 如果自定义的响应器继承自 UIView 或者 UIViewController，就应该实现所有的事件处理方法。
- 如果子类继承自其他响应器类，其中某些不需要处理的事件方法的实现可以是空的。
- 在所有的方法中，记得调用父类对应的方法实现。

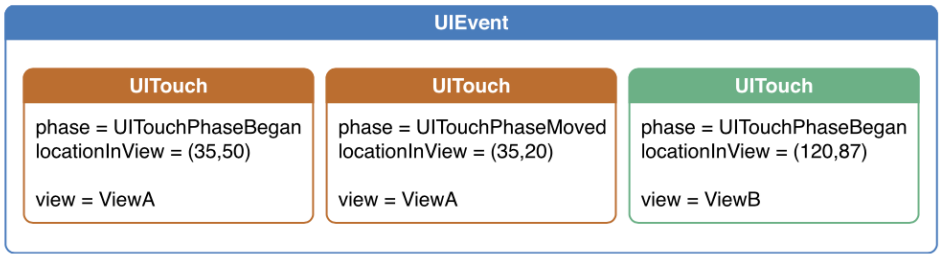
如果开发者尝试不让某个响应器对象，在指定的某个阶段接收触摸事件对象，其导致的结果可能会对应产生未知且非预期的行为。

3.追踪触摸事件的状态和位置

iOS 会对多点触摸序列事件进行追踪。iOS 会记录每一个触摸对象的属性，包括其状态、位置坐标，移动之前的位置坐标，以及时间戳。使用这些属性来确定如何响应触摸事件。

触摸对象使用 phase 属性存储其状态值，每一个不同的状态对应一个触摸事件方法。触摸对象存储位置信息的方法有三个：触摸事件所在的窗体对象中的位置，对应所在窗体中视图对象的位置，或者触摸事件所在视图对象的位置。如图 3-1 的例子，展示了正在进行的触摸事件对应的触摸对象的位置信息。

Figure 3-1 Relationship of a UIEvent object and its UITouch objects



当手指触摸屏幕时，对应的触摸对象会在其事件的生命周期内，始终和所处的窗体对象以及视图对象关联，即使之后“触摸”位置不在视图区域范围内。通过触摸对象所在的位置信息，决定需要对触摸事件如何响应。比如，如果两下触摸快而连续，如果这两下发生在同一个视图内，就可以将其当作是一次“双击”事件。一个触摸对象可以同时存储当前位置和前一个位置信息（如果有前一个位置的话）。

4.检索查询触摸对象

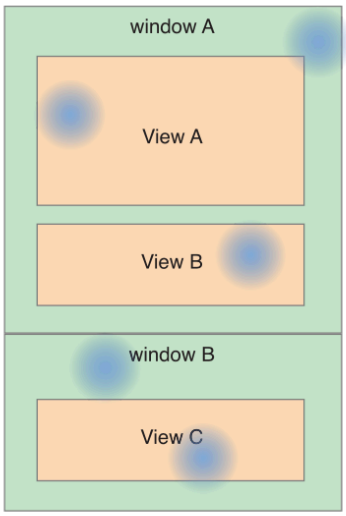
在某个事件处理方法中，开发者可以从一下两个对象中获取到和事件相关的触摸对象的信息：

- UITouch 对象序列集合。被传入的 NSSet 包含了所有该方法所代表的状态下新生成的、或者发生变化的触摸对象，比如 UITouchPhaseBegan 对应的 touchesBegan:withEvent:方法。
- 事件对象（UIEvent）。被传入的 UIEvent 对象包含了给定的多点触摸序列事件相关的所有触摸对象。

默认情况下 multipleTouchEnabled 属性值为 NO，这也就意味着一个视图对象仅仅只接收多点触摸序列事件的第一个触摸对象。当该属性为“禁用”（disabled）状态时，开发者通过 anyObject 方法就只能索引到一个触摸对象，因为整个触摸事件集合对象里就只包含了一个触摸对象。

开发者通过视图对象的 locationInView:方法，可以获得触摸事件对象 UITouch 的位置信息 。传入 self，开发者所获得的也就是 UITouch 所在视图对象坐标系中的位置坐标信息。类似的，属性 previousLocationInView:方法就可以获取到触摸点的前一个位置点坐标。开发者还能指定一个触摸事件 Tap（点击）几次（tapCount），触摸对象什么时候生成、最后是什么时候发生变化的（时间戳 timestamp），以及触摸对象所在的状态（touch phase）。

Figure 3-2 All touches for a given touch event

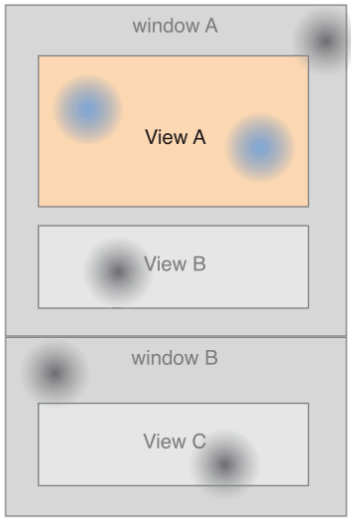


如果开发者只对特定窗体相关联的触摸对象感兴趣，那就调用 UIEvent 对象的方法 touchesForWindow: 。视图 3-3 显示了窗体 A 中的所有触摸对象。

Figure 3-3 All touches belonging to a specific window

如果开发者感兴趣的是和视图对象关联的触摸对象，那就调用 UIEvent 对象的方法 touchesForView:。如图 3-4 显示视图对象 A 的所有触摸对象。

Figure 3-4 All touches belonging to a specific view



处理 Tap 手势

除了让 APP 能够识别处理 tap 手势之外，开发者可能还想将单击、双击和三击给区分开来，此时使用触摸事件的属性 tapCount 来确定触摸事件对应用户在视图对象中的点击次数。

获取该值的最好的地方就是 touchesEnded:withEvent:方法，因为刚好这个方法就对应了用户手指结束点击、离开屏幕的时刻。在多点触摸事件结束时，通过查找触摸事件阶段的点击次数，开发者可以确定手指是不是真的在点击，比如，手指“按下”并“拖动”。代码清单 1-3 就是一个例子，展示了判断某视图对象中所发生的点击事件是不是“双击”。

Listing 3-1 Detecting a double tap gesture

```
- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event {
}
- (void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event {
}
- (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event {
      for (UITouch *aTouch in touches) {
            if (aTouch.tapCount >= 2) {
             // The view responds to the tap
             [self respondToDoubleTapGesture:aTouch];
            }
      }
}
- (void)touchesCancelled:(NSSet *)touches withEvent:(UIEvent *)event {
}
```

5.处理 Swipe 和 Drag 手势

水平和垂直方向的滑动手势是开发者可以追踪的简单手势之一。要检测滑动手势，就要追踪对应的轴向移动信息。然后，通过检查以下几个问题，来确定触摸移动是不是滑动手势操作：

- 用户的手指移动距离够远吗？
- 手指的移动相对比较直的直线运动吗？
- 移动速度够快、快到可以称之为“滑动”吗？

要回答这些问题，就需要存储触摸对象的初始位置信息，当其移动时再与之进行比较。

代码清单 3-2，展示了开发者可以用来检测视图对象中水平方向滑动（swipe）手势的基本方法。在这个例子中，视图对象 view 有一个 startTouchPosition 属性用来存储触摸操作的初始位置坐标信息。在 toucesEnded:方法中，比较“触摸”结束位置的坐标和初始的位置坐标，来确定这是不是一个“滑动”手势。如果移动得太远或者移动地得不够远，就不认为这是一个滑动手势。代码清单中并没有显示对应方法 myProcessRightSwipe:和方法 myProcessLeftSwipe:的实现，但是自定义视图对象可以在那里对滑动手势进行处理。

Listing 3-2 Tracking a swipe gesture in a view

```
#define HORIZ_SWIPE_DRAG_MIN 12
#define VERT_SWIPE_DRAG_MAX   4
- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event {
      UITouch *aTouch = [touches anyObject];
      // startTouchPosition is a property
      self.startTouchPosition = [aTouch locationInView:self];
}
- (void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event {
}
- (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event {
      UITouch *aTouch = [touches anyObject];
      CGPoint currentTouchPosition = [aTouch locationInView:self];
      //  Check if direction of touch is horizontal and long enough
      if (fabsf(self.startTouchPosition.x - currentTouchPosition.x) >=
  HORIZ_SWIPE_DRAG_MIN &&
          fabsf(self.startTouchPosition.y - currentTouchPosition.y) <=
  VERT_SWIPE_DRAG_MAX)
      {
          // If touch appears to be a swipe
          if (self.startTouchPosition.x < currentTouchPosition.x) {
            [self myProcessRightSwipe:touches withEvent:event];
          } else {
              [self myProcessLeftSwipe:touches withEvent:event];
```

```
        }
        self.startTouchPosition = CGPointZero;
}
- (void)touchesCancelled:(NSSet *)touches withEvent:(UIEvent *)event {
        self.startTouchPosition = CGPointZero;
}
```

注意这段代码并没有检查手势操作事件移动过程的中间位置坐标，这也就意味着该手势可能是在整个屏幕中转了一圈，如果开始点和结束点在一条线上，还是会被认做为滑动手势操作。一个更为精密的滑动手势操作识别器，就需要在 touchesMoved:withEvent:方法中检查移动过程中位置信息。要检查垂直方向的滑动手势，开发者可以使用类似的代码，只需要把 x 和 y 坐标进行交换一下就可以了。

代码清单 3-3，显示了更简单的追踪单指触摸事件的实现，这次是用户拖动视图对象在屏幕中移动。此处，自定义视图对象类仅实现了 touchesMoved:withEvent:方法。该方法计算触摸对象在视图中当前位置和前一个位置的差值，然后使用这个差值，来设置试图对象的位置。

Listing 3-3 Dragging a view using a single touch

```
- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event {
}
- (void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event {
        UITouch *aTouch = [touches anyObject];
        CGPoint loc = [aTouch locationInView:self];
        CGPoint prevloc = [aTouch previousLocationInView:self];
        CGRect myFrame = self.frame;
        float deltaX = loc.x - prevloc.x;
        float deltaY = loc.y - prevloc.y;
        myFrame.origin.x += deltaX;
        myFrame.origin.y += deltaY;
        [self setFrame:myFrame];
}
- (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event {
}
- (void)touchesCancelled:(NSSet *)touches withEvent:(UIEvent *)event {
}
```

6.处理更为复杂的多点触摸序列事件

点击 Taps、拖动 drags、滑动 swipes 只使用了一根手指而且易于追踪处理。处理包含有双指、多指的触摸事件就更具有挑战性。开发者可能需要追踪所有状态下的每个触摸事件对

象，记录触摸对象发生变化的属性值，并对其内部状态进行适当的修改。要追踪和处理怼点触摸，就需要：

- 将视图对象的 multipleTouchEnabled 属性设置为 YES。
- 使用 Core Foundation 字典对象（CFDictionaryRef）来追踪触摸对象在手势操作事件过程中不同状态下的变化。

在处理多点触摸事件时，开发者通常需要存储某个触摸对象的状态信息，后续用来对触摸对象进行比较。比如，开发者可能会想要比较每个触摸对象的初始位置和终止位置。在方法 touchesBegan:withEvent:中，开发者可以通过 locationInTheView:方法获取到每个触摸对象的初始位置信息，然后将触摸事件对象的地址作为 key 值，把这些位置信息存储到某个 CFDictionaryRef 对象中。然后，在 touchesEnded:withEvent:方法中，就可以用来索引到被传入的触摸对象的初始位置，并和其当前位置进行比较。

备注：之所以使用 CFDictionaryRef 数据类型而不是 NSDictionary 来追踪触摸点，是因为 NSDictionary 需要复制其存储的 key 信息，（以 UITouch 对象作为 key 的条件下）UITouch 类却没有声明遵循对象复制必须的 NSCopying 协议。

代码清单 3-4 就展示了如何在某个 Core Foundation 的字典中存储 UITouch 对象初始位置信息。方法 cacheBeginPointForTouches 方法存储了每个触摸点相对于父视图坐标系的位置信息，这样一来就可以以同样的坐标系为参照，来比较所有触摸点的位置坐标了。

Listing 3-4 Storing the beginning locations of multiple touches

```
- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event {
     [self cacheBeginPointForTouches:touches];
}
- (void)cacheBeginPointForTouches:(NSSet *)touches {
    if ([touches count] > 0) {
        for (UITouch *touch in touches) {
            CGPoint *point = (CGPoint
*)CFDictionaryGetValue(touchBeginPoints,
touch);
            if (point == NULL) {
                point = (CGPoint *)malloc(sizeof(CGPoint));
                CFDictionarySetValue(touchBeginPoints, touch, point);
            }
            *point = [touch locationInView:view.superview];
        }
      }
}
```

代码清单 3-5 是基于上一个例子写的。这段代码展示了如何从字典对象中索引出初始的位置信息，然后，再获取同样的触摸点对应的当前位置信息，使用这些信息，就可以计算出来几何变形矩阵（这一部分没有显示）。

Listing 3-5 Retrieving the initial locations of touch objects

```
- (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event {
      CGAffineTransform newTransform = [self
      incrementalTransformWithTouches:touches];
}
- (CGAffineTransform)incrementalTransformWithTouches:(NSSet *)touches {
      NSArray *sortedTouches = [[touches allObjects]
      sortedArrayUsingSelector:@selector(compareAddress:)];
      // Other code here
      CGAffineTransform transform = CGAffineTransformIdentity;
      UITouch *touch1 = [sortedTouches objectAtIndex:0];
      UITouch *touch2 = [sortedTouches objectAtIndex:1];
      CGPoint beginPoint1 =
      *(CGPoint*)CFDictionaryGetValue(touchBeginPoints,touch1);
      CGPoint currentPoint1 = [touch1 locationInView:view.superview];
      CGPoint beginPoint2 = *(CGPoint
      *)CFDictionaryGetValue(touchBeginPoints,touch2);
      CGPoint currentPoint2 = [touch2 locationInView:view.superview];
      // Compute the affine transform
      return transform;
}
```

下一个例子，如代码清单3-6，并没有使用字典对象追踪触摸点的变化，但是，依然可以触摸事件的多点触摸。这个例子展示的是，当手指按压着标记有Welcome字样的视图板，在屏幕内移动时，将视图版也跟着移动。当用户双击视图版时，还会改变其对应的语言。这段代码源自于《MoveMe》的样例工程，有需要的可以阅读一下，从而更好的理解事件处理的整个上下文。

Listing 3-6 Handling a complex multitouch sequence

```
- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event {
      // App supports only single touches, so anyObject retrieves just
      // that touch from touches
             UITouch *touch = [touches anyObject];
      // Move the placard view only if the touch was in the placard view
      if ([touch view] != placardView) {
      // In case of a double tap outside the placard view, update
      // the placard's display string
            if ([touch tapCount] == 2) {
```

```
                [placardView setupNextDisplayString];
            }
            return;
      }
      // Animate the first touch
      CGPoint touchPoint = [touch locationInView:self];
      [self animateFirstTouchAtPoint:touchPoint];
      }
}
- (void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event {
      UITouch *touch = [touches anyObject];
// If the touch was in the placardView, move the placardView to its location
      if ([touch view] == placardView) {
            CGPoint location = [touch locationInView:self];
            placardView.center = location;
      }
}
- (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event {
      UITouch *touch = [touches anyObject];
      // If the touch was in the placardView, bounce it back to the center
      if ([touch view] == placardView) {
            // Disable user interaction so subsequent touches
            // don't interfere with animation
            self.userInteractionEnabled = NO;
            [self animatePlacardViewToCenter];
      }
}
- (void)touchesCancelled:(NSSet *)touches withEvent:(UIEvent *)event {
       // To impose as little impact on the device as possible, simply set
       // the placard view's center and transformation to the original values
       placardView.center = self.center;
       placardView.transform = CGAffineTransformIdentity;
}
```

要确定多点触摸序列事件中最后一根手指什么时候离开视图对象，就要看传入的组合对象中有多少个UITouch触摸对象，传入的UIEvent事件对象中有多少个UITouch触摸对象。如果这两个参数中的一致，也就意味着多点触摸序列事件结束了，如代码清单3-7：

Listing 3-7 Determining when the last touch in a multitouch sequence has ended

```
- (void)touchesEnded:(NSSet*)touches withEvent:(UIEvent*)event {
    if ([touches count] == [[event touchesForView:self] count]) {
        // Last finger has lifted
    }
}
```

需要记住一点，传入的set里的对象，对应的是和视图相关联、并在某个给定阶段中发生变化或者新加的触摸对象，然而方法touchesForView返回的对象，则是和视图相关联的所有的UITouch对象。

7.指定自定义触摸事件的行为

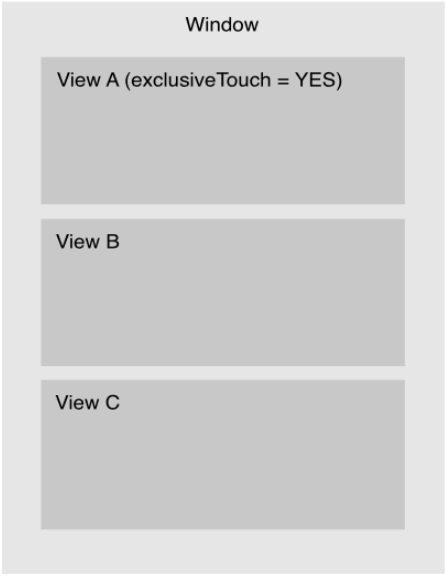
通过修改某个特定的手势识别器、某个特定的视图对象、或者APP中所有的触摸事件，来自定义APP处理触摸事件的方式。开发者可以通过一下几种方式，来修改对应的触摸事件流：

- **启用多点触摸事件的传输**。默认情况下，视图对象除了多点触摸的第一个触摸对象之外，其他的都会给忽略掉。如果开发者想要视图对象能响应处理多点触摸，就必须修改对应的属性来启用此功能，开发者可以在Interface Builder或者手工纯代码进行设置，将视图对象的multipleTouchEnabled属性设置为YES即可。

- **限制某个视图的事件传输**。默认情况下，视图对象的exclusiveTouch属性的值为NO，对应的意思是，此视图对象不会阻止窗体中的其他视图对象接收触摸事件对象。如果将指定的某个视图对应的这个值设置为YES，那么这个视图对象将会是仅有的唯一接收并追踪触摸事件的视图对象。

  如果你的视图并非独有，用户可以一根手指在某个视图中点击、另一根手指在另一个视图操作，每个视图都有单独追踪各自的手势操作。假如有视图设置如图3-5所示，视图A是一个独享触摸事件操作的视图对象。如果用户只在A中进行触摸操作，触摸操作就会被识别。但是假如用户先把手指按压在视图B中，然后又用另外一根手指在A中触摸，此时A是无法接收到触摸事件的，因为此时A视图不是唯一追踪触摸事件的视图对象。类似的，假如用户先用手指按压在A视图里，然后用另外一根手指去B中触摸操作，B就没有办法接收到触摸事件了，因为A是是唯一指定接收触摸事件的对象。其他任何情况下，用户对B和C同时操作，这两个视图都可以接收到各自的触摸事件对象。

Figure 3-5 Restricting event delivery with an exclusive-touch view



- **限制子视图的事件传递**。通过重写自定义视图类的hitTest:withEvent:可以使得多点触摸事件不被传递给某个指定的子视图。请参阅“重写Hit-Test拦截触摸事件”章节的描述。

当然，开发者也可以直接完全关闭触摸事件的传递，或者关闭一段时间也可以：

- **完全关闭触摸事件传递**。将试图对象的userInteractionEnabled属性为NO就可以关闭了。需要注意的是，如果某个视图被隐藏了或者纯透明状态下，也是无法接收触摸事件的。
- **在一段时间内关闭触摸事件传递**。有时候开发者可能只是想暂时地关闭掉触摸事件的传递，比如正在执行动画效果的过程时。此时，可以通过调用方法beginIgnoringInteractionEvent停止接收事件传递，然后在我们需要继续接收触摸事件时，调用方法endIgnoringInteractionEvent就可以了。

8.重写 Hit-Test 拦截触摸事件

如果开发者自定的视图对象包含有一些子视图对象的时候，就需要确定好触摸事件的处理，是在父视图层还是在子视图层。如果选择在父视图层处理，也就意味着所有的子视图对象不实现touchesBegan:withEvent, touchesEnded:withEvent, touchesMoved:withEvent等系列方法，与此同时，重写父试图类的hitTest:withEvent:方法并将其自己返回即可。

重写hit-testing，将其自身设置为hit-test的视图对象返回，可以确保父视图对象接收到所有的触摸事件，因为父视图所截取并接收到的触摸对象，会优先被传递给其子视图对象进行处理。如果父视图对象没有重写hit-test的方法，触摸事件对象就会被绑定到事件所触发

的子视图对象上，父视图对象也就接收不到了。

回想一下两个hit-test方法，一个是UIView视图对象的hitTest:withEvent:和CALayer对象的hitTest:，如章节“Hit-Testing返回触摸事件发生所在的视图对象”所描述。开发者几乎不需要自己调用这两个方法，更多的是重写这两个方法来拦截传递给子视图对象的触摸事件。然而，有时候响应器在事件转发之前进行hit-testing。

PS：我重新回去查看了一下英文原文，根本没有描述过 CALayer 的 hitTest:方法。所以跑去查了一下官方文档，简单说明一下，CALayer 的 hitTest:方法原型为– (CALayer *)hitTest:(CGPoint)p;其作用是 返回消息接收对象的 layer 树最远的子节点对象。传入参数是一个 CGPoint 结构体对象，即父 layer 坐标系中的位置坐标点，返回值是一个 CALayer 对象，如果消息接收者的 layer 对象包含这个点，就会将该 layer 返回，如果这个点落在接收对象的区域之外，就返回 nil。

CALayer 本身只能进行内容展示，不能对用户的操作或者任何外部事件进行处理，所以我的理解是，苹果提供此方法，大家可以发挥自己的想象力，到底要不要、什么时候要使用 CALayer 此方法，就我个人的拙见，当某一个视图对象包含有多个 CALayer 对象来进行内容展示，在接收到触摸事件时，可以用来进行更为微妙的内部展示效果的自定义处理。

9.转发触摸事件

要将触摸事件转发给另外一个响应器对象，只需要向其合适的触摸事件处理消息即可。使用该技术时需要特别小心，因为 UIKit 框架的类对象的设计本身，是无法接收没有与其绑定的触摸事件对象的。如果某个响应器要处理触摸事件，触摸事件对应的视图对象就必须拥有此响应器的引用。如果开发者想要按照某些特定的条件或者规则，将事件转发给 APP 的其他响应器对象，所有的响应器对象都必须是开发者自己的定义的 UIView 子类的实例对象。

举个例子，某个 APP 有三个自定义视图对象：A，B，C。当用户触摸 A 时，APP 的窗体就会通过 hit-test 检查到 A，并将事件传递给 A。在特定的条件下，A 要么将此事件传递给 B，要么就传递给 C。这种情况下，A、B、C 都必须清楚此转发动作，而且 B、C 必须能够处理这些没有和他们绑定的触摸事件。

事件转发通常需要对触摸对象进行分析，以确定它们是否应该被转发。有以下几种方法进行分析：

- 对于某个“覆盖图层”视图对象，比如常见的父视图，使用 hit-testing 来拦截触摸事件，从而进行分析是否需要将其转发给其子视图对象。

- 在自定的 UIWindow 子类中重写 sendEvent:方法，分析触摸事件对象，然后将其转发给合适的响应器。

重写 sendEvent:方法可以让开发者监控 APP 所接收到的所有事件。UIApplication 和 UIWindow 都实在 sendEvent 方法中将接收到的事件进行派发，所以此方法就类似于所有事件进入 APP 的漏斗入口点一样。极少数的 APP 需要重写 sendEvent 方法，但是如果真的要重写的话，一定确保调用父类的实现一[super sendEvent:theEvent]。绝对不要篡改事件的分发处理。

代码清单 3-8 展示了如何在 UIWindow 的一个子类中重写 sendEvent。该示例代码中，事件对象被发送给自定义的帮助响应器， 在一个与之绑定的视图上执行线性转换动画。

Listing 3-8 Forwarding touch events to helper responder objects

```
- (void)sendEvent:(UIEvent *)event {
    for (TransformGesture *gesture in transformGestures) {
       // Collect all the touches you care about from the event
      NSSet *touches = [gesture observedTouchesForEvent:event];
      NSMutableSet *began = nil;
      NSMutableSet *moved = nil;
      NSMutableSet *ended = nil;
      NSMutableSet *canceled = nil;
      //Sort touches by phase to handle--similar to normal event dispatch
      for (UITouch *touch in touches) {
          switch ([touch phase]) {
             case UITouchPhaseBegan:
               if (!began) began = [NSMutableSet set];
               [began addObject:touch];
               break;
             case UITouchPhaseMoved:
               if (!moved) moved = [NSMutableSet set];
               [moved addObject:touch];
               break;
             case UITouchPhaseEnded:
               if (!ended) ended = [NSMutableSet set];
               [ended addObject:touch];
               break;
             case UITouchPhaseCancelled:
               if (!canceled) canceled = [NSMutableSet set];
               [canceled addObject:touch];
               break;
             default:
               break;
      }
   }
```

```
            // Call methods to handle the touches
        if (began) [gesture touchesBegan:began withEvent:event];
        if (moved) [gesture touchesMoved:moved withEvent:event];
        if (ended) [gesture touchesEnded:ended withEvent:event];
        if (canceled) [gesture touchesCancelled:canceled withEvent:event];
    }
    [super sendEvent:event];
}
```

注意，子类重写的 sendEvent 方法中调用了父类的实现，这对于整个触摸事件流的完整性非常重要。

10.处理多点触摸事件的最佳实践

在处理触摸和运动事件的时候，开发者应该遵循以下几个推荐的技术和模式：

1. **记得实现事件取消方法**。

   在取消方法的实现中，将试图对象的状态恢复成事件发生之前的初始状态。如果不这么做的话，视图对象将会处于一个变形的状态下，在某些情况下，可能会导致其他视图对象接收到此取消事件的消息。

2. **如果开发者在 UIView，UIViewController 或者 UIRespondor 的子类中处理事件**：

   - 实现所有的事件处理方法，即使这些方法什么都不做。
   - 不要在这些方法中调用父类的实现。

3. **如果是在其他 UIKit 框架响应器类的子类中处理事件**：

   在这里可不需要实现所有的事件处理方法。
   在这些自定义实现的方法中，确保调用父类的实现，举个例子：
   [super touchesBegan:touches withEvent:event];

4. **不要将事件转发给任何不属于 UIKit 框架的响应器对象**。

   事件转发，应该转发给开发者自定义的视图对象的其他的响应器实例对象。另外，确保这些响应器对象清楚地知道，事件转发正在进行，同时它们可以接收到这些没有与之绑定的触摸事件。

5. **不要显示地调用 nextResponder 方法将事件沿着响应器链向上传递，而是调用父类的实现，让 UIKit 框架来处理响应器链的事件传递。**

6. 不要在事件处理过程中使用“约数转换整数”的代码，这会导致精度丢失。

为了维持代码良好的兼容性，iOS 所传达的触摸事件在一个 320x480 的坐标系空间内。然而，在高分辨率的设备上，对应的分辨率刚好为此分辨率的两倍：640x960.这也就意味着，触摸事件在高分辨率的设备上以半个点为基点，而在一些老的设备上则是整个点为基点。

运动事件

当用户移动、摇晃或者倾斜设备时，就会产生运动事件。这些运动事件是设备上的硬件检测到的，也就是我们所说的**加速器**(accelerometer) 和**陀螺仪**(accelerometer) 。

加速器 实际上是由三个加速器组成的，x 轴、y 轴和 z 轴。每一个加速器都会在直线方向上实时的计算各自轴向上的向量变化。将这三个加速器组合在一起，就可以检测到设备任何方向上的运动，并获得设备的当前方位。尽管这是三个不同的加速器，但是本文将这三个组合定义为一个完整的加速器实体。陀螺仪则用来检测在 x，y，z 三个方向上的旋转角度。

所有的运动事件都源自于相同的硬件。有以下几个方法可以获取硬件信息数据，使用哪一个则取决于 APP 的需求：

- 如果只需要检测常用的设备方位信息，而不需要方位信息的向量，可以使用 UIDevice 类。更多信息请参见章节 “使用 UIDevice 获取当前设备方位” 。
- 如果想要 APP 响应用户摇晃设备的事件，可以使用 UIKit 框架的运动事件处理方法，从传入的 UIEvent 对象中获取相应的信息。更多相关信息请参见章节 “使用 UIEvent 检测摇晃运动事件” 。
- 如果 UIDevice 和 UIEvent 还觉得不够，那就使用 Core Motion 框架直接访问加速器、陀螺仪和设备运动类。更多信息请参见章节 “使用 Core Motion 捕获设备动作” 。

1.使用 UIDevice 获取当前设备方位

当开发者仅需要知道设备的常用方位信息、而不需要切确的向量方位信息时，就可以使用UIDevice的方法。使用UIDevice很简单，而且不需要开发者自己去计算方位信息的向量。

在获取当前方位信息之前，开发者需要调用beginGeneratingDeviceOrientationNotification方法，告诉UIDevice类开始生成设备方位信息通知。这样就启用了加速器硬件，因为有可能为节省电量被关掉了。代码清单4-1是在viewDidLoaded方法中实现的一个例子。

启用方位通知后，直接使用UIDevice对象的orientation属性即可获取到当前的方位信息。如果开发者想在设备方位发生变化时得到通知，注册通知UIDeviceOrientationDidChangeNotification即可。设备的方位信息会使用UIDeviceOrientation常量传递，指定设备是处于横屏模式、竖屏模式、屏边朝上或者朝下等等。这些常量代指

设备的物理方位，而不是对应APP用户层面的方位信息。

当不再需要知道设备的方位信息时，记得调用UIDevice的方法来禁用掉方位通知，该方法位endGenerateDeviceNotifications。当加速器硬件再别地方也不使用的时候，系统就会关闭掉硬件检测，从而减少电量的损耗。

Listing 4-1 Responding to changes in device orientation

```
-(void) viewDidLoad {
//Request to turn on accelerometer and begin receiving accelerometer events
     [[UIDevice currentDevice]
     beginGeneratingDeviceOrientationNotifications];
     [[NSNotificationCenter defaultCenter] addObserver:self
     selector:@selector(orientationChanged:)
     name:UIDeviceOrientationDidChangeNotification
      object:nil];
}
- (void)orientationChanged:(NSNotification *)notification {
     // Respond to changes in device orientation
}
-(void) viewDidDisappear {
//Request to stop receiving accelerometer events and turn off accelerometer
     [[NSNotificationCenter defaultCenter] removeObserver:self];
     [[UIDevice currentDevice] endGeneratingDeviceOrientationNotifications];
}
```

还有一个关于响应UIDevice方位发生变化的例子，请参见AlternateViews示例代码。

2.使用 UIEvent 检测摇晃运动事件

当用户摇晃设备时，iOS就会对加速器的数据信息进行评估。如果这些数据符合特定的标准，iOS就会识别出摇晃操作，并生成对应的UIEvent事件对象来表示此事件。然后将此事件对象发送给当前的活动状态的APP进行处理。需要注意是，APP可以同时对摇晃动作事件和设备方位变化进行响应。

运动事件相对于触摸事件要更简单。系统会告诉APP某个运功开始了、结束了，而不是某个运动发生了。而且，运动事件仅有一个事件类型（UIEventTypeMotion），一个事件子类型UIEventSubtypeMotionShake，和一个时间戳。

2.1 指定运动事件的第一响应器

要接收运动事件，就需要指定一个响应器对象为第一响应器，也就是开发者想要用来处理运动事件的对象。代码清单4-2就是一个将自己指定为第一响应器的例子：

Listing 4-2 Becoming first responder

```
- (BOOL)canBecomeFirstResponder {
    return YES;
}
- (void)viewDidAppear:(BOOL)animated {
    [self becomeFirstResponder];
}
```

运动事件会通过响应器链去查找可以处理事件的对象。当用户摇晃设备时，iOS会向第一响应器发送运动事件。如果第一响应器不处理，就会沿着响应器链向上传递。更多信息请参阅“响应器链的特定传递路径”章节。如果事件沿着响应链一直传递到了window对象还没有被处理，与此同时，UIApplication的applicationSupportsShakeToEdit属性为YES（也是默认值），iOS就会展示一个Undo和Redo指令的表单。

2.2 实现运动处理方法

有三个运动处理方法，motionBegan:withEvent:，motionEnded:withEvent:，motionCancelled:withEvent:。要处理运动事件，就必须至少实现motionBegan:withEvent:和motionEnded:withEvent:之一。如果是响应器，还应该实现motionCancelled:withEvent:方法，以响应iOS取消了运动事件。如果摇晃事件被打断了，或者摇晃事件最后无效了（比如摇晃得太久了），该事件就会被取消掉。

代码清单4-3取自样例代码工程GLPaint的片段。该APP中，用户可以在屏幕上绘画，通过摇晃设备擦除掉绘画。这段代码再motionEnded:withEvent:方法中判断是否发生了摇晃事件，如果是的话，就会发送一个通知，来执行shake-to-erase方法。

Listing 4-3 Handling a motion event

```
- (void)motionEnded:(UIEventSubtype)motion withEvent:(UIEvent *)event {
    if (motion == UIEventSubtypeMotionShake)
    {// User was shaking the device. Post a notification named "shake."
        [[NSNotificationCenter defaultCenter] postNotificationName:@"shake"
      object:self];
    }

}
```

备注：出了简单之外，另一个考虑使用shake-motion events而不是Core Motion的原因是，开发者可以直接在iOS模拟器中模拟摇晃事件进行测试和调试。要获取更多iOS模拟器的信息，请查阅

“iOS模拟器用户指南”。

2.3 设置并检查运动事件所需要的硬件能力

在APP访问设备相关的某些特性（比如加速器数据）时，开发者必须先确定好APP所需要的硬件能力。明确地讲，就是需要在APP的Info.plist文件中添加对应的键值对。在运行时，仅当对应的设备具备对应的硬件能力，iOS才可以启动对应的APP。比如，如果开发者的APP依赖于陀螺仪数据，那陀螺仪就应该列入对应设备的硬件要求，但凡没有陀螺仪的设备，就不能启动对应的APP。APP Store（苹果应用商店）也会根据此信息，在用户下载APP时提示用户，免得用户下载无法运行的APP。

通过向APP的属性列表文件添加对应的“键值”，来申明APP所需要的设备硬件能力。有两种UIRequiredDeviceCapabilities基于硬件源的“键值”，来处理运动事件：

- 加速器（accelerometer）
- 陀螺仪（gyroscope）

备注：如果开发者的APP仅仅只需要检测设备方位的变化，是不需要添加加速器键值的。

开发者既可以使用数组、也可以使用字典数据类型指定对应的键值。如果使用的是数组，直接将对应的key存储到数组中；如果使用的是字典，直接存储key对应的启用的属性值为布尔型。这两种方式下，所有没有被列入进去的属性，都是不需要的。更多信息，请参阅“信息属性列表键值参考”中的“UIRequiredDeviceCapabilities”。

假如开发者的APP有需要使用陀螺仪数据的特性，但是该特性对于用户体验而言也不是必须的，所以开发者希望没有陀螺仪硬件的设备也能下载使用。在这种陀螺仪并非必要依赖的情况下，可能还是想尝试获取陀螺仪数据，开发者就需要在运行时去检查陀螺仪是否可用。开发者可以使用CMMotionManager的gyroAvailable属性进行检查。

3.使用 Core Motion 捕获设备动作

Core Motion框架就是用来访问原始的加速器、陀螺仪数据的，然后将数据传递给某个APP进行处理。Core Motion使用特有的算法来处理收集到的原始数据，从而能呈现出更精确的信息。这项处理是在框架自己的线程中进行的（PS：也就是说肯定不是APP在主线程中）。

Core Motion和UIKit是完全分开的，它和UIEvent（UI事件对象）模型没有关联，也不使用响应链，而是将移动事件直接传递给需要对事件进行处理的应用程序，仅此而已。

有三种数据对象用来指代Core Motion事件，每一个对象都可能封装了一个或多个计量

值：

- CMAccelerometerData对像捕获的是加速器三维物理坐标轴各轴向的加速度信息。
- CMGyroData对象捕获的是三维物理坐标轴各轴向的旋转角度值。
- CMDeviceMotion对象封装了各种不同的计量值，包括坐标位置信息、更有用的旋转角度和加速器数据。

CMMotionManager类是Core Motion的中心切入点。开发者实例化一个CMMotionManager对象，指定刷新的时间间隔，标明开始刷新的起点，然后处理其发送过来的运动事件。一个APP应该仅创建一个CMMotionManager的实例对象，多个这样的对象会影响APP从加速器和陀螺仪接收数据的频率。

Core Motion框架所有的数据封装类都是CMLogItem的子类，这个类定义了一个时间戳，所以运动数据会使用时间做标记，并记录到文件中。APP可以比较运动事件和较早的事件的时间戳，从而确定事件之间的实际刷新频率。

之前描述过的每一种运动数据类型，CMMotionManager都提供两种数据获取方法：

- **Pull**. 某个app指定事件刷新获取数据的起点，然后间歇性地去获取最近收集到的运动数据。
- **Push**.某个app指定一个刷新时间间隔，然后实现block来处理数据。然后指定事件刷新获取数据的起点，然后向Core Motion框架传递一个操作队列和执行的block。Core Motion会将每次事件更新传递给block，也就是在操作队列执行一个任务。

对于大部分的APP，推荐使用Pull方式，尤其是游戏。这种方法通常来讲更高效、代码量也更少。Push的方式对于一些数据收集类的APP会更合适，这类APP需要捕获所有的运动计量值。两种方法具有良好的线程安全设计，push和pull都是在操作队列中执行开发者设定的block任务，Core Motion不会中断已有的线程任务。

重点：使用Core Motion框架，开发者最好在真机上进行测试和调试，因为iOS模拟器上是无法支持加速器和陀螺仪数据的。

一旦处理完必要的数据，一定要记住立即终止掉运动数据的更新捕获。这样的话Core Motion框架就会关闭掉运动传感器，从而节省电量。

3.1 选择运动事件更新的时间间隔

当开发者想要使用Core Motion获取运动数据时，就需要指定运动数据更新获取的时间间隔。此时就应该根据APP的需要，选取一个最大的时间间隔，因为时间间隔越大，向APP发送事件的数量或频率就越低，这样也就节省更多电量。表4-1就列举了一些常见的更新频率，并解释了对应频率下所产生的数据可以用来做什么。极少数的APP才需要以100次

每秒的频率来获取加速器的事件信息。

Table 4-1 加速器事件常见的几种频率

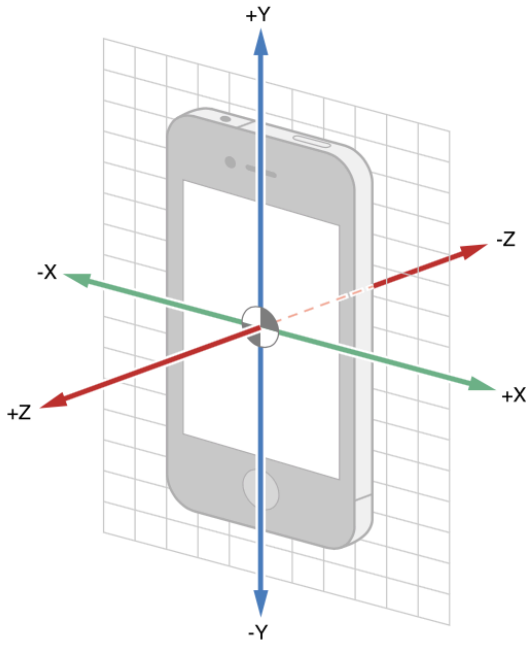
事件频率(Hz)	用法
10–20	适用于确定设备当前的方位指向向量。
30–60	适用于游戏或者其他需要使用用户实时生成加速器事件的APP。
70–100	适用于需要检测高频运动的APP。例如，开发者想要使用此频率来检测用户快速敲击或者或摇晃设备的事件。

开发者可以设定时间间隔值为10ms，对应的也就是100Hz的事件更新频率，但是大部分的APP其实使用更大的时间间隔操作起来也会很高效。

3.2 使用 Core Motion 处理加速器事件

加速器计量的事三个轴向上的运动向量，如图 4-1 所示。使用 Core Motion 框架，每次移动都会被封装了 CMAcceleration 结构体的 CMAcececeleromoterData 对象所捕获。

Figure 4-1 The accelerometer measures velocity along the x, y, and z axes



创建 CMMotionManager 实例对象并调用以下任一方法，即可开始接收并处理加速器数据：

- startAccelerometerUpdates – Pull 方式

调用此方法后，Core Motion 就会按照最大的加速器活动测量值，持续不断地更新 CMMotionManager 的 accelerometer 属性值。然后，开发者可以间歇性的获取此属性数据样本，比如在游戏 APP 中常见的渲染循环中进行此操作。

- startAccelerometerUpdatesToQueue:withHandler: -- Push 的方式

在调用此方法之前，先给 accelerometerUpdateInterval 属性一个时间间隔值，创建一个 NSOperationQueue 的实例对象，并实现一个 CMAccelerometerHandler 类型的 block 来处理加速器更新。然后，调用方法 startAccelerometerUpdatesToQueue:withHandler:（使用创建好的 motion-manager 对象调用），传入刚刚创建好的 queue 操作队列对象和事件处理的 block。在指定的时间间隔下，Core Motion 就会将最新的加速器活动事件样本传入 block，在队列中执行处理任务。

代码清单 4-1 截取自样例代码工程 MotionGraphs。此 APP 中，用户可以通过滑块来指定更新的时间间隔。然后，创建一个 CMMotionManager 实例对象，检查设备是否带有加速器硬件，并将此时间间隔赋值给 motion-manager 对象。此 APP 使用 push 的方式来获取加速器数据，然后将其绘制在图上。注意，APP 在 stopUpdates 方法中停止了加速器的更新。

Listing 4-4 Accessing accelerometer data in MotionGraphs

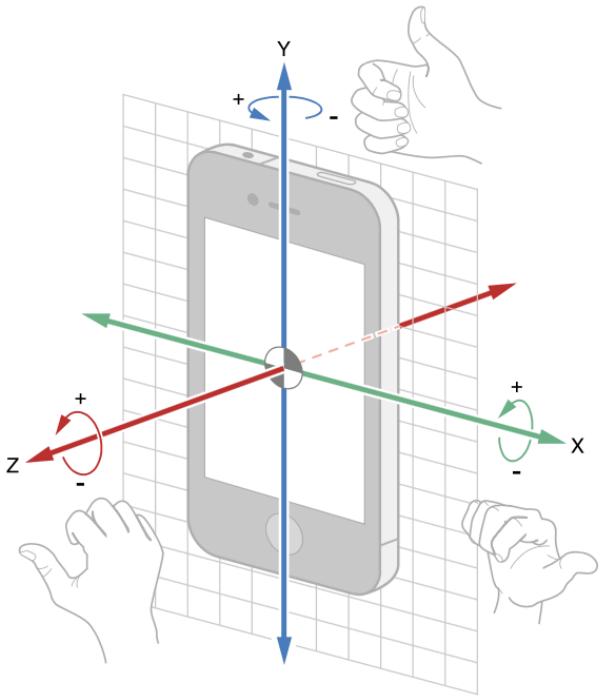
```
static const NSTimeInterval accelerometerMin = 0.01;
– (void)startUpdatesWithSliderValue:(int)sliderValue {
   // Determine the update interval
   NSTimeInterval delta = 0.005;
   NSTimeInterval updateInterval = accelerometerMin + delta * sliderValue;
   // Create a CMMotionManager
   CMMotionManager *mManager = [(APLAppDelegate *)[[UIApplication
  sharedApplication] delegate] sharedManager];
   APLAccelerometerGraphViewController * __weak weakSelf = self;
   // Check whether the accelerometer is available
   if ([mManager isAccelerometerAvailable] == YES) {
   // Assign the update interval to the motion manager
   [mManager setAccelerometerUpdateInterval:updateInterval];
   [mManager startAccelerometerUpdatesToQueue:[NSOperationQueue mainQueue]
   withHandler:^(CMAccelerometerData *accelerometerData, NSError *error) {
  [weakSelf.graphView addX:accelerometerData.acceleration.x
  y:accelerometerData.acceleration.y z:accelerometerData.acceleration.z];
  [weakSelf setLabelValueX:accelerometerData.acceleration.x
```

```
  y:accelerometerData.acceleration.y z:accelerometerData.acceleration.z];
}];
}
      self.updateIntervalLabel.text = [NSString stringWithFormat:@"%f",
  updateInterval];
}
- (void)stopUpdates {
  CMMotionManager *mManager = [(APLAppDelegate *)[[UIApplication
  sharedApplication] delegate] sharedManager];
  if ([mManager isAccelerometerActive] == YES) {
      [mManager stopAccelerometerUpdates];
  }
}
```

3.3 处理旋转角度数据

陀螺仪计量的是设备在三个轴向上各自的旋转角度，如图 4-2 所示。

Figure 4-2 The gyroscope measures rotation around the x, y, and z axes



每次需要陀螺仪更新，Core Motion 都会携带出一个有误差的旋转角度估值，并将此信息使用一个 CMGyroData 的对象返回出来。CMGyroData 有一个 CMRotationRate 结构体类型的 rotationRate 的属性，用来捕获三个轴向上各自的每秒的旋转角度。注意，CMGyroData 计量得到的旋转角度值是有误差的。开发者可以使用 CMDeviceMotion 类来获取跟家精确、几乎没有误差的计量值。更多信息请参见章节 “处理加工过的设备运动数据” 。

在分析旋转角度数据时，尤其是分析 CMRotationMatrix 结构体的字段域时，注意遵守“右手规则”从而确定旋转的方向，如图 4-2 所示。举个例子，沿着 x 轴握住右手，并将大拇指指向 x 轴正值方向，旋转角度的正值方向就是剩下的四根手指的朝向，对应的负值方向就是其相反方向。

创建 CMMotionManager 实例对象，并调用一下任意方法即可开始接收并处理旋转角度数据了：

- startGyroUpdates – pull 方式

调用此方法后，Core Motion 就会根据最新的陀螺仪活动测量值，不断的更新 CMMotionManager 中的 gyroData 属性的值，然后，开发者可以周期性的获取此属性值。如果开发者确定使用此方法，就将 update-interval（更新时间间隔）设置为所需要的最大时间间隔，Core Motion 就会以此时间间隔进行更新。

- startGyroUpdatesToQueue:withHandler: -- push 方式

在调用此方法前，先将赋值给 gyroUpdateInterval 属性一个时间间隔，创建一个 NSOperationQueue 的实例对象，并实现一个类型为 CMGyroHandler 的 block 来处理陀螺仪的更新。然后，调用 motion-manager 的 startGyroUpdatesToQueue:withHandler 方法，传入刚刚创建的操作队列和 block。在指定的时间间隔下，Core motion 会不断的将最新的陀螺仪活动取样数据传入 block，在队列中执行对应的任务。

代码清代 4-5 也取自样例代码项目 MotionGraphs，和代码清代 4-4 很相似。此 APP 使用 push 的方式来获取对应的陀螺仪数据并绘制在屏幕上。

Listing 4-5 Accessing gyroscope data in MotionGraphs

```
static const NSTimeInterval gyroMin = 0.01;
 - (void)startUpdatesWithSliderValue:(int)sliderValue {
// Determine the update interval
       NSTimeInterval delta = 0.005;
       NSTimeInterval updateInterval = gyroMin + delta * sliderValue;
       // Create a CMMotionManager
       CMMotionManager *mManager = [(APLAppDelegate *)[[UIApplication
  sharedApplication] delegate] sharedManager];
       APLGyroGraphViewController * __weak weakSelf = self;
       // Check whether the gyroscope is available
       if ([mManager isGyroAvailable] == YES) {
            // Assign the update interval to the motion manager
            [mManager setGyroUpdateInterval:updateInterval];
            [mManager startGyroUpdatesToQueue:[NSOperationQueue mainQueue]
  withHandler:^(CMGyroData *gyroData, NSError *error) {
                 [weakSelf.graphView addX:gyroData.rotationRate.x
```

```
  y:gyroData.rotationRate.y z:gyroData.rotationRate.z];
                [weakSelf setLabelValueX:gyroData.rotationRate.x
  y:gyroData.rotationRate.y z:gyroData.rotationRate.z];
}];
}
       self.updateIntervalLabel.text = [NSString stringWithFormat:@"%f",
  updateInterval];
}
- (void)stopUpdates{
       CMMotionManager *mManager = [(APLAppDelegate *)[[UIApplication
  sharedApplication] delegate] sharedManager];
       if ([mManager isGyroActive] == YES) {
            [mManager stopGyroUpdates];
      }
}
```

3.4 处理加工过的设备运动数据

如果设备同时拥有加速器和陀螺仪硬件，Core Motion 提供了能处理来自这两种感应器的原始数据的服务。设备运动使用传感器融合算法提取出原始数据，并生成对应设备信息，包括有方位、几乎没有误差的旋转角度、设备的重力方向以及用户生成的加速信息。一个 CMDeviceMotion 类的实例对象，就可以封装所有的这些信息。另外，开发者不需要自己过滤出加速器数据，因为 device-motion 已经将重力和用户的加速信息分开了。

开发者可以通过 CMDeviceMotion 对象的 attitude 属性获取方位数据，对应封装的是一个 CMAttitude 对象。每一个 CMAttitude 对象都封装了表示方位信息的三个的数学表达式：

- 一个四元组
- 一个旋转矩阵
- 三个欧拉角（章动角θ、旋进角（即进动角）ψ和自转角 j）

创建 CMMotionManager 类对象，调用以下两个方法之一，即可开始接收并处理设备运动更新数据：

- startDeviceMotionUpdate – pull 方式

调用此方法后，Core Motion 会根据加速器和陀螺仪活动得到的最新的测量值，持续不断地更新 CMMotionManager 的 deviceMotion 属性，该属性对应的是一个 CMDeviceMotion 对象的封装。然后，开发者可以周期性的获取此属性。如果开发者确定使用此方法，就将 update-interval（更新时间间隔）属性（对应字段为 deviceMotionUpdateInterval）设置为所需要的最大时间间隔，Core Motion 就会以此时间间隔进行更新。
代码片段 4-6 展示了如何使用这种方法

- startDeviceMotionUpdatesToQueue:withHandler: -- push 方式

在调用此方法前，先给deviceMotionUpdateInterval 属性赋值一个更新间隔，创建一个类对象NSOperateQueue的实例，并实现一个CMDeviceMotionHandler类型的block来处理加速器的更新。然后调用motion-manager的startDeviceMotionUpdatesToQueue:withHandler:方法，传入刚刚创建的queue和block。Core Motion会以设定的更新间隔，将加速器和陀螺仪的活动取样信息，以CMDeviceMotion为载体传入block，在queue中执行对应的任务。

代码清单4-6截取自pArk样例代码，展示了如何启用和停止设备运动更新。startDeviceMotion方法使用的是pull方式，通过一个参考系来启动设备更新。更多设备运动参考系 信息请参见章节 “设备方位以及参考系” 。

Listing 4-6 Starting and stopping device motion updates

```
- (void)startDeviceMotion {
     // Create a CMMotionManager
     motionManager = [[CMMotionManager alloc] init];
     // Tell CoreMotion to show the compass calibration HUD when required
     // to provide true north-referenced attitude
     motionManager.showsDeviceMovementDisplay = YES;
     motionManager.deviceMotionUpdateInterval = 1.0 / 60.0;
     // Attitude that is referenced to true north
[motionManager
startDeviceMotionUpdatesUsingReferenceFrame:CMAttitudeReferenceFrameXTrueNort
hZVertical];
}
- (void)stopDeviceMotion {
     [motionManager stopDeviceMotionUpdates];
}
```

3.5 设备方位以及参考系

每一个 CMDeviceMotion 对象都包含了设备的方位、空间上的方向信息。设备的方位总是以一个对应的 frame 来计量。Core Motion 会在 APP 开始进行设备运动更新时，建立一个对应的 frame，然后，CMAttitude 就可以根据对应的初始化的参考系和设备当前的参考系计算得出一个旋转角度值。

在 Core Motion 参考系中，z 轴方向总是垂直的，x 轴和 y 轴总是和重心引力方向相垂直的，这样的话对应的重力向量值可以表示为[0,0,-1]。这也被称作重力参考系。如果把从 CMAttitude 对象获取出来的旋转矩阵和重力参考相乘，就会的得到设备在重力上的一个参考值，数学表达式为：

$$\text{deviceMotion.gravity} = R \begin{pmatrix} 0 \\ 0 \\ -1 \end{pmatrix}$$

开发者可以修改 CMAttitude 所使用的参考系，通过缓存包含参考系信息的方位对象，作为传入值参数传入 multiplyByInverseOfAttitude 即可。对应的方位参数接收到变化信息，这样一来，就可以计算出来方位信息和传入的参考系之间的变化。

大部分的 APP 都会关注设备方位信息的变化。要知道这些信息有什么用处，可以想象一下棒球游戏中，用户通过旋转设备进行“挥棒”操作。通常，在开始投球时，棒球拍会有一个初始的静止角度，然后，根据设备方位信息从“挥棒”开始的变化，来渲染棒球拍。代码清单 4-7 就是一个例子。

Listing 4-7 Getting the change in attitude prior to rendering

```
-(void) startPitch {
    // referenceAttitude is a property
    self.referenceAttitude = self.motionManager.deviceMotion.attitude;
}
- (void)drawView {
    CMAttitude *currentAttitude = self.motionManager.deviceMotion.attitude;
    [currentAttitude multiplyByInverseOfAttitude: self.referenceAttitude];
    // Render bat using currentAttitude
    [self updateModelsWithAttitude:currentAttitude];
    [renderer render];
}
```

在这个例子中，方法 multiplyByInverseOfAttitude 返回之后，currentAttitude 对象就表示了方位信息从 referenceAttitude 对象和从 CMAttitude 实例对象取出的最近的数据取样信息的变化。

远程控制事件

远程控制事件让用户可以设备的多媒体。如果 APP 可以播放音频或者视频内容，开发者可能就想让 APP 能够对用户的远程控制操作进行响应，这种控制行为可以来自于传输控制器或者外部传感器。（外部传感器的话就必须遵从苹果公司提供的规格说明。）iOS 将指令转换为 UIEvent 对象，并将事件对象传递给 APP。APP 将这些对象发送给第一响应器，如果第一响应器无法处理，事件对象就会沿着响应链向上传递。更多信息，请参见章节“响应器链的特定传输路径”。

本章描述了如何接受并处理远程控制事件。对应的代码样例取自 Audio Mixer（MixerHost）样例代码项目。

1.预设 APP 接收远程控制事件

要接收远程控制事件，开发者的 APP 必须做三件事：

- **成为第一响应器**。展示多媒体内容的视图或者视图控制器必须是第一响应器。
- **开启远程控制事件的传递**。APP 必须显示地要求开始接收远程控制事件。
- **开始播放音频**。开发者的 APP 必须是“当前正在播放”的 APP。重申一下，即使开发者的 APP 是第一响应器，而且已经开启了远程事件接收，除非 APP 开始播放音频，否则 APP 是无法接收到远程控制事件的。

要将对应的视图或者视图控制器成为第一响应器，对应的就必须重写 UIRespondor 类的 canBecomeFirstResponder 方法并返回 YES，并且还要在合适的时间向自己发送 becomeFirstRespondor 消息。比如，某个视图控制器可以在重写的 viewDidAppear 方法中调用 becomeFirstRespondor，如代码清单 5-1.此代码样例同时展示了视图控制器通过调用 UIApplication 的 beginReceivingRemoteControlEvent 方法启用远程控制时间的传递。

Listing 5-1 Preparing to receive remote control events

```
-(void)viewDidAppear:(BOOL)animated {
      [super viewDidAppear:animated];
      // Turn on remote control event delivery
      [[UIApplication sharedApplication] beginReceivingRemoteControlEvents];
      // Set itself as the first responder
      [self becomeFirstResponder];
}
```

一旦视图或者视图控制器不再处理音频或者视频，就应该关闭掉远程控制事件的传递。如代码清单 5-2 所示，同时也应该在 viewWillDisappear 方法中注销掉第一响应器的状态。

Listing 5-2 Ending the receipt of remote control event

```
- (void)viewWillDisappear:(BOOL)animated {
    // Turn off remote control event delivery
    [[UIApplication sharedApplication] endReceivingRemoteControlEvents];
    // Resign as first responder
    [self resignFirstResponder];
    [super viewWillDisappear:animated];
}
```

2.处理远程控制事件

要处理远程控制事件，第一响应器必须实现 UIRespondor 所申明的方法 remoteControlReceivedWithEvent。对应的实现方法评估传入的每一个 UIEvent 对象的子类型，然后根据对应的子类型，向处理音频或者视频内容的对象发送合适的消息。代码清单 5-3 展示了向音频处理对象发送“播放”、“暂停”、“停止”消息。其他的远程控制 UIEvent 子类型也是可能的，详细请参见 UIEvent 类参考。

Listing 5-3 Handling remote control events

```
- (void)remoteControlReceivedWithEvent:(UIEvent *)receivedEvent {
    if (receivedEvent.type == UIEventTypeRemoteControl) {
        switch (receivedEvent.subtype) {
            case UIEventSubtypeRemoteControlTogglePlayPause:
                [self playOrStop: nil];
                break;
            case UIEventSubtypeRemoteControlPreviousTrack:
                [self previousTrack: nil];
                break;
            case UIEventSubtypeRemoteControlNextTrack:
                [self nextTrack: nil];
                break;
            default:
                break;
      }
   }
}
```

3.设备上测试远程控制事件

使用“当前正在播放控制项”来测试 APP 是否可以正常接收并处理远程控制事件。这些控制项在运行 iOS4.0 及以上版本的 iOS 设备上都是可用的。要访问这些控制项，按两次 Home 键，然后在屏幕底部向右滑动直到找到音频后台播放控制项。这些控制项会向当前或最近播放音频的 APP 发送远程控制事件。音频后台播放控制项右边图标，就表示当前接收远程控制事件的 APP。

以测试为目的话，开发者可以用代码的方式让 APP 开始后台播放音频，然后通过点击当前播放控制项来进行测试。注意，正式发布的 APP 是不应该直接用代码控制开始后台播放，而是让用户进行控制。