iOS开发系列--触摸事件、手势识别、摇晃事件、耳机线控

浏览： 82 发布日期： 2016-09-05 分类： ios

概览

iPhone的成功很大一部分得益于它多点触摸的强大功能，乔布斯让人们认识到手机其实是可以不用按键和手写笔直接操作的，这不愧为一项伟大的设计。今天我们就针对iOS的触摸事件（手势操作）、运动事件、远程控制事件等展开学习：

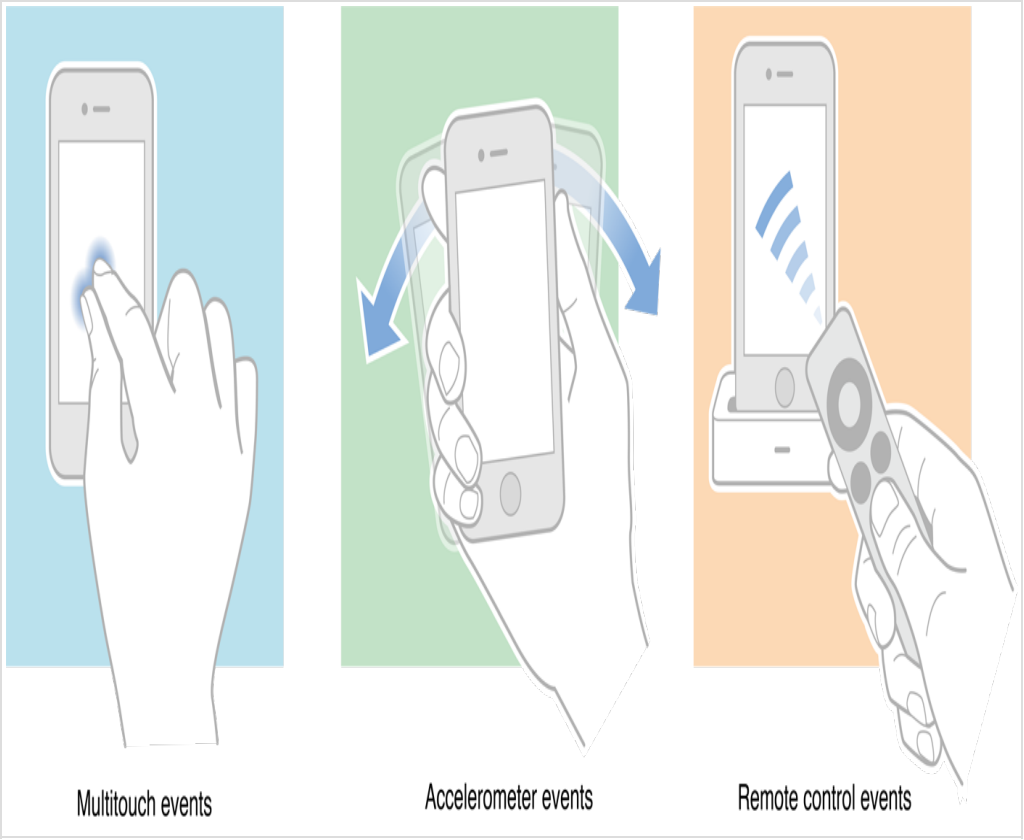
1. iOS事件
2. 触摸事件
3. 手势识别
4. 运动事件
5. 远程控制事件

iOS事件

在iOS中事件分为三类：

1. 触摸事件：通过触摸、手势进行触发（例如手指点击、缩放）
2. 运动事件：通过加速器进行触发（例如手机晃动）
3. 远程控制事件：通过其他远程设备触发（例如耳机控制按钮）

下图是苹果官方对于这三种事件的形象描述：



在iOS中并不是所有的类都能处理接收并事件，只有继承自UIResponder类的对象才能处理事件（如我们常用的UIView、UIViewController、UIApplication都继承自UIResponder，它们都能接收并处理事件）。在UIResponder中定义了上面三类事件相关的处理方法：

事件	说明
触摸事件	
- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event;	一根或多根手指开始触摸屏幕时执行；
- (void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event;	一根或多根手指在屏幕上移动时执行，注意此方法在移动过程中会重复调用；
- (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event;	一根或多根手指触摸结束离开屏幕时执行；
- (void)touchesCancelled:(NSSet *)touches withEvent:(UIEvent *)event;	触摸意外取消时执行（例如正在触摸时打入电话）；
运动事件	
- (void)motionBegan:(UIEventSubtype)motion withEvent:(UIEvent *)event NS_AVAILABLE_IOS(3_0);	运动开始时执行；
- (void)motionEnded:(UIEventSubtype)motion withEvent:(UIEvent *)event NS_AVAILABLE_IOS(3_0);	运动结束后执行；
- (void)motionCancelled:(UIEventSubtype)motion withEvent:(UIEvent *)event	运动被意外取消时执行；

NS_AVAILABLE_IOS(3_0);	
远程控制事件	
- (void)remoteControlReceivedWithEvent:(UIEvent *)event NS_AVAILABLE_IOS(4_0);	接收到远程控制消息时执行；

触摸事件

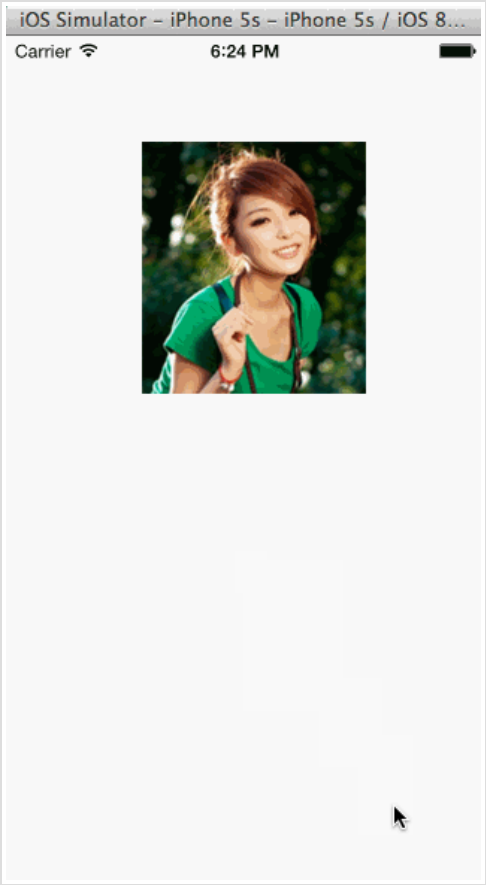
基础知识

三类事件中触摸事件在iOS中是最常用的事件，这里我们首先介绍触摸事件。

在下面的例子中定义一个KCImage，它继承于UIView，在KCImage中指定一个图片作为背景。定义一个视图控制器KCTouchEventViewController，并且在其中声明一个KCImage变量，添加到视图控制器中。既然UIView和UIViewController都继承于UIResponder，那么也就就意味着所有的UIKit控件和视图控制器均能接收触摸事件。首先我们在KCTouchEventViewController中添加触摸事件，并利用触摸移动事件来移动KCImage，具体代码如下：

```
}

-(void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event{
    //取得一个触摸对象（对于多点触摸可能有多个对象）
    UITouch *touch=[touches anyObject];
    //NSLog(@"%@",touch);

    //取得当前位置
    CGPoint current=[touch locationInView:self.view];
    //取得前一个位置
    CGPoint previous=[touch previousLocationInView:self.view];

    //移动前的中点位置
    CGPoint center=_image.center;
    //移动偏移量
    CGPoint offset=CGPointMake(current.x-previous.x, current.y-previous.

    //重新设置新位置
    _image.center=CGPointMake(center.x+offset.x, center.y+offset.y);

    NSLog(@"UIViewController moving...");

}

-(void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event{
    NSLog(@"UIViewController touch end.");
}
@end
```

现在运行程序：



上面示例中我们用到了UITouch类，当执行触摸事件时会将这个对象传入。在这个对象中包含了触摸的所有信息：

window：触摸时所在的窗口

view：触摸时所在视图

tapCount:短时间内点击的次数

timestamp:触摸产生或变化的时间戳

phase:触摸周期内的各个状态

locationInView:方法：取得在指定视图的位置

previousLocationInView:方法：取得移动的前一个位置

从上面运行效果可以看到无论是选择KCImage拖动还是在界面其他任意位置拖动都能达到移动图片的效果。既然KCImage是UIView当然在KCImage中也能触发相应的触摸事件，假设在KCImage中定义三个对应的事件：

```
//

#import "KCImage.h"

@implementation KCImage

- (instancetype)initWithFrame:(CGRect)frame {
    self = [super initWithFrame:frame];
    if (self) {
        UIImage *img=[UIImage imageNamed:@"photo.png"];
        [self setBackgroundColor:[UIColor colorWithPatternImage:img]];
    }
    return self;
}

#pragma mark - UIView的触摸事件
-(void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event{
    NSLog(@"UIView start touch...");
}

-(void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event{
    NSLog(@"UIView moving...");
}

-(void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event{
    NSLog(@"UIView touch end.");
}
@end
```

此时如果运行程序会发现如果拖动KCImage无法达到预期的效果，但是可以发现此时会调用KCImage的触摸事件而不会调用KCTouchEventViewController中的触摸事件。如果直接拖拽其他空白位置则可以正常拖拽，而且从输出信息可以发现此时调用的是视图控制器的触摸事件。这是为什么呢？要解答这个问题我们需要了解iOS中事件的处理机制。

事件处理机制

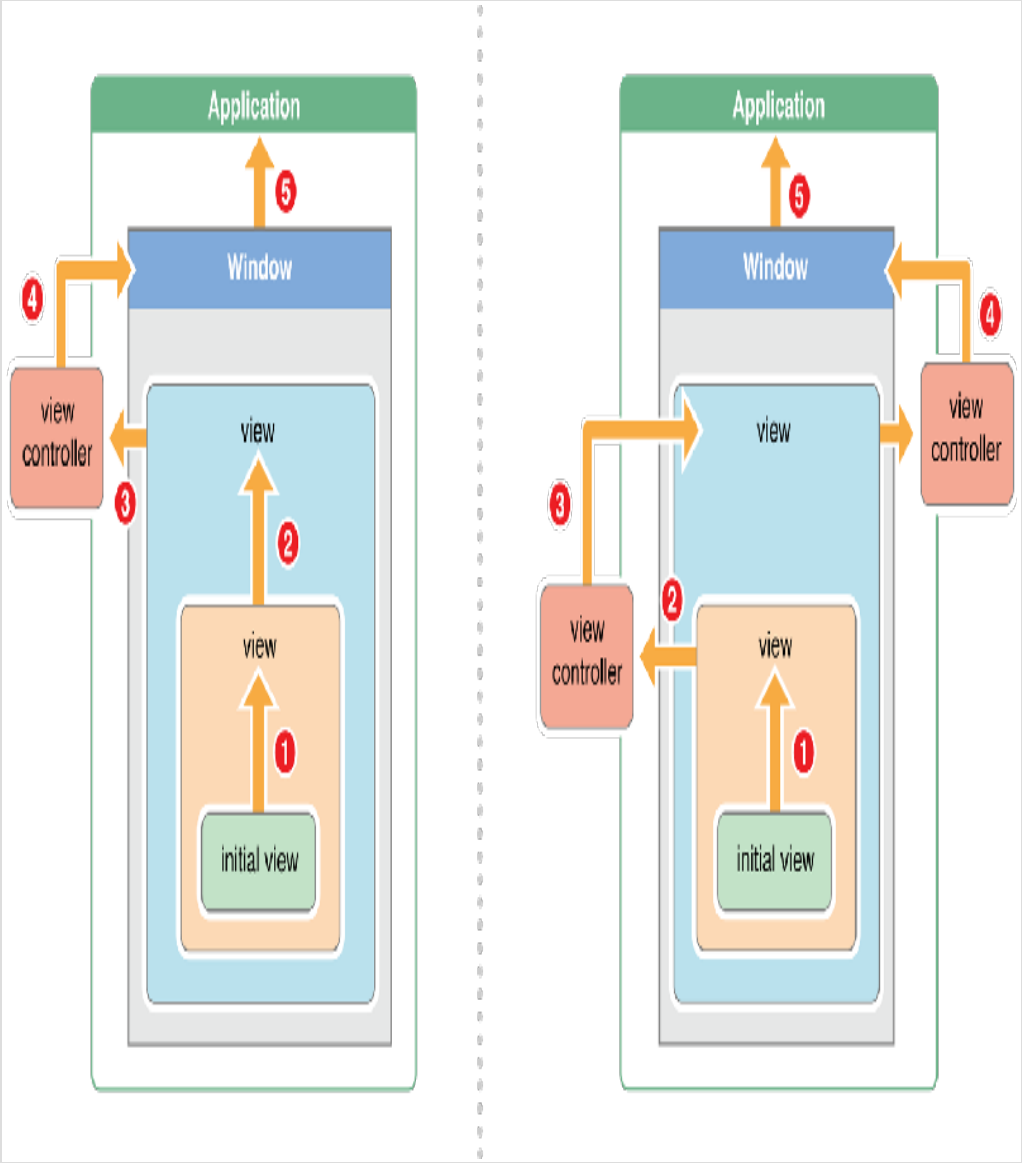
在iOS中发生触摸后，事件会加入到UIApplication事件队列（在这个系列关于iOS开发的第一篇文章中我们分析iOS程序原理的时候就说过程序运行后UIApplication会循环监听用户操作），UIApplication会从事件队列取出最前面的事件并分发处理，通常先分发给应用程序主窗口，主窗口会调用**hitTest:withEvent:**方法（假设称为方法A，注意这是UIView的方法），查找合适的事件触发视图（这里通常称为“hit-test view”）：

1. 在顶级视图（key window的视图）上调用pointInside:withEvent:方法判断触摸点是否在当前视图内；
2. 如果返回NO，那么A返回nil；
3. 如果返回YES，那么它会向当前视图的所有子视图（key window的子视图）发送hitTest:withEvent:消息，遍历所有子视图的顺序是从subviews数组的末尾向前遍历（从界面最上方开始向下遍历）。
4. 如果有subview的hitTest:withEvent:返回非空对象则A返回此对象，处理结束（注意这个过程，子视图也是根据pointInside:withEvent:的返回值来确定是返回空还是当前子视图对象的。并且这个过程中如果子视图的hidden=YES、userInteractionEnabled=NO或者alpha小于0.1都会并忽略）；
5. 如果所有subview遍历结束仍然没有返回非空对象，则A返回顶级视图；

上面的步骤就是点击检测的过程，其实就是查找事件触发者的过程。触摸对象并非就是事件的响应者（例如上面第一个例子中没有重写KCImage触摸事件时，KCImge作为触摸对象，但是事

件响应者却是UIViewController），检测到了触摸的对象之后，事件到底是如何响应呢？这个过程就必须引入一个新的概念“响应者链”。

什么是响应者链呢？我们知道在iOS程序中无论是最后面的UIWindow还是最前面的某个按钮，它们的摆放是有前后关系的，一个控件可以放到另一个控件上面或下面，那么用户点击某个控件时是触发上面的控件还是下面的控件呢，这种先后关系构成一个链条就叫“响应者链”。在iOS中响应者链的关系可以用下图表示：



当一个事件发生后首先看initial view能否处理这个事件，如果不能则会将事件传递给其上级视图（inital view的superView）；如果上级视图仍然无法处理则会继续往上传递；一直传递到视图控制器view controller，首先判断视图控制器的根视图view是否能处理此事件；如果不能则接着判断该视图控制器能否处理此事件，如果还是不能则继续向上传递；（对于第二个图视图控制器本身还在另一个视图控制器中，则继续交给父视图控制器的根视图，如果根视图不能处理则交给父视图控制器处理）；一直到window，如果window还是不能处理此事件则继续交给application（UIApplication单例对象）处理，如果最后application还是不能处理此事件则将其丢弃。

这个过程大家理解起来并不难，关键问题是在这个过程中各个对象如何知道自己能不能处理该事件呢？对于继承UIResponder的对象，其不能处理事件有几个条件：

userInteractionEnabled=NO

hidden=YES

alpha=0~0.01

没有实现开始触摸方法（注意是touchesBegan:withEvent:而不是移动和结束触摸事件）

当然前三点都是针对UIView控件或其子控件而言的，第四点可以针对UIView也可以针对视图控制器等其他UIResponder子类。对于第四种情况这里再次强调是对象中重写了开始触摸方法，则会处理这个事件，如果仅仅写了移动、停止触摸或取消触摸事件（或者这三个事件都重写了）没有写开始触摸事件，则此事件该对象不会进行处理。

相信到了这里大家对于上面点击图片为什么不能拖拽已经很明确了。事实上通过前面的解释大家应该可以猜到即使KCImage实现了开始拖拽方法，如果在KCTouchEventViewController中设置KCImage对象的userInteractionEnabled为NO也是可以拖拽的。

> 注意：上面提到**hitTest:withEvent:**可以指定触发事件的视图，这里就不再举例说明，这个方法重写情况比较少，一般用于自定义手势，有兴趣的童鞋可以访问：Event Delivery: The Responder Chain。

手势识别

简介

通过前面的内容我们可以看到触摸事件使用起来比较容易，但是对于多个手指触摸并进行不同的变化操作就要复杂的多了。例如说如果两个手指捏合，我们虽然在触摸开始、移动等事件中可以通过UITouchs得到两个触摸对象，但是我们如何能判断用户是用两个手指捏合还是横扫或者拖动呢？在iOS3.2之后苹果引入了手势识别,对于用户常用的手势操作进行了识别并封装成具体的类供开发者使用，这样在开发过程中我们就不必再自己编写算法识别用户的触摸操作了。在iOS中有

手势	说明
UITapGestureRecognizer	点按手势
UIPinchGestureRecognizer	捏合手势
UIPanGestureRecognizer	拖动手势
UISwipeGestureRecognizer	轻扫手势，支持四个方向的轻扫，但是不同的方向要分别定义轻扫手势
UIRotationGestureRecognizer	旋转手势
UILongPressGestureRecognizer	长按手势

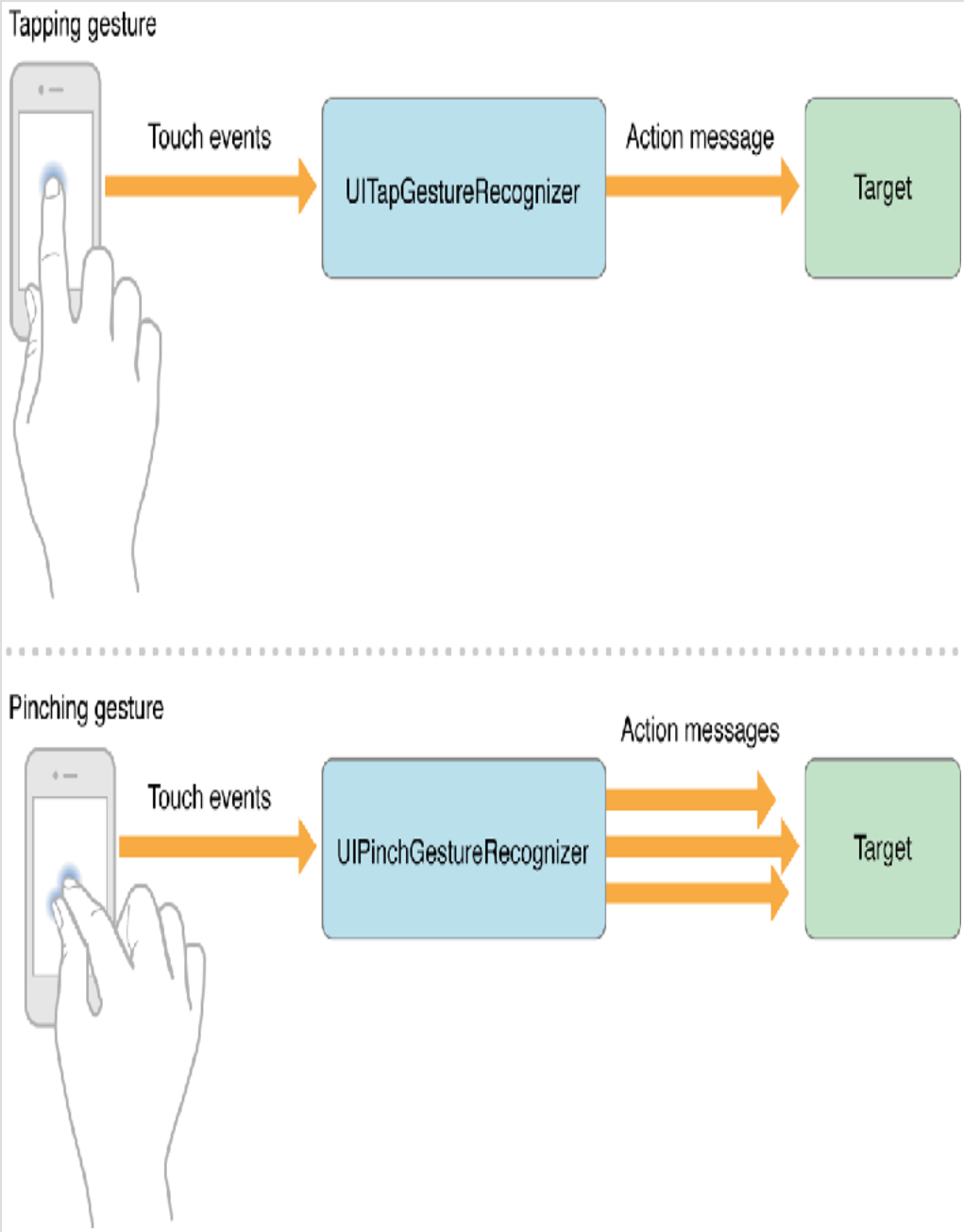
所有的手势操作都继承于UIGestureRecognizer，这个类本身不能直接使用。这个类中定义了这几种手势共有的一些属性和方法(下表仅列出常用属性和方法)：

名称	说明

属性	
@property(nonatomic,readonly) UIGestureRecognizerState state;	手势状态
@property(nonatomic, getter=isEnabled) BOOL enabled;	手势是否可用
@property(nonatomic,readonly) UIView *view;	触发手势的视图（一般在触摸执行操作中我们可以通过此属性获得触摸视图进行操作）
@property(nonatomic) BOOL delaysTouchesBegan;	手势识别失败前不执行触摸开始事件，默认为NO；如果为YES，那么成功识别则不执行触摸开始事件，失败则执行触摸开始事件；如果为NO，则不管成功与否都执行触摸开始事件；
方法	
- (void)addTarget:(id)target action:(SEL)action;	添加触摸执行事件
- (void)removeTarget:(id)target action:(SEL)action;	移除触摸执行事件
- (NSUInteger)numberOfTouches;	触摸点的个数（同时触摸的手指数）
- (CGPoint)locationInView:(UIView*)view;	在指定视图中的相对位置
- (CGPoint)locationOfTouch:(NSUInteger)touchIndex inView:(UIView*)view;	触摸点相对于指定视图的位置
- (void)requireGestureRecognizerToFail:(UIGestureRecognizer *)otherGestureRecognizer;	指定一个手势需要另一个手势执行失败才会执行
代理方法	
- (BOOL)gestureRecognizer:(UIGestureRecognizer *)gestureRecognizer shouldRecognizeSimultaneouslyWithGestureRecognizer:(UIGestureRecognizer *)otherGestureRecognizer;	一个控件的手势识别后是否阻断手势识别继续向下传播，默认返回NO；如果为YES，响应者链上层对象触发手势识别后，如果下层对象也添加了手势并成功识别也会继续执行，否则上层对象识别后则不再继续传播；

手势状态

这里着重解释一下上表中手势状态这个对象。在六种手势识别中，只有一种手势是离散手势，它就是UITapGestureRecgnier。离散手势的特点就是一旦识别就无法取消，而且只会调用一次手势操作事件（初始化手势时指定的触发方法）。换句话说其他五种手势是连续手势，连续手势的特点就是会多次调用手势操作事件，而且在连续手势识别后可以取消手势。从下图可以看出两者调用操作事件的次数是不同的：



在iOS中将手势状态分为如下几种：

```
typedef NS_ENUM(NSInteger, UIGestureRecognizerState) {
    UIGestureRecognizerStatePossible,   // 尚未识别是何种手势操作（但可能已经触发
    UIGestureRecognizerStateBegan,      // 手势已经开始，此时已经被识别，但是这个
    UIGestureRecognizerStateChanged,    // 手势状态发生转变
    UIGestureRecognizerStateEnded,      // 手势识别操作完成（此时已经松开手指）
    UIGestureRecognizerStateCancelled,  // 手势被取消，恢复到默认状态
    UIGestureRecognizerStateFailed,     // 手势识别失败，恢复到默认状态
    UIGestureRecognizerStateRecognized = UIGestureRecognizerStateEnded //
};
```

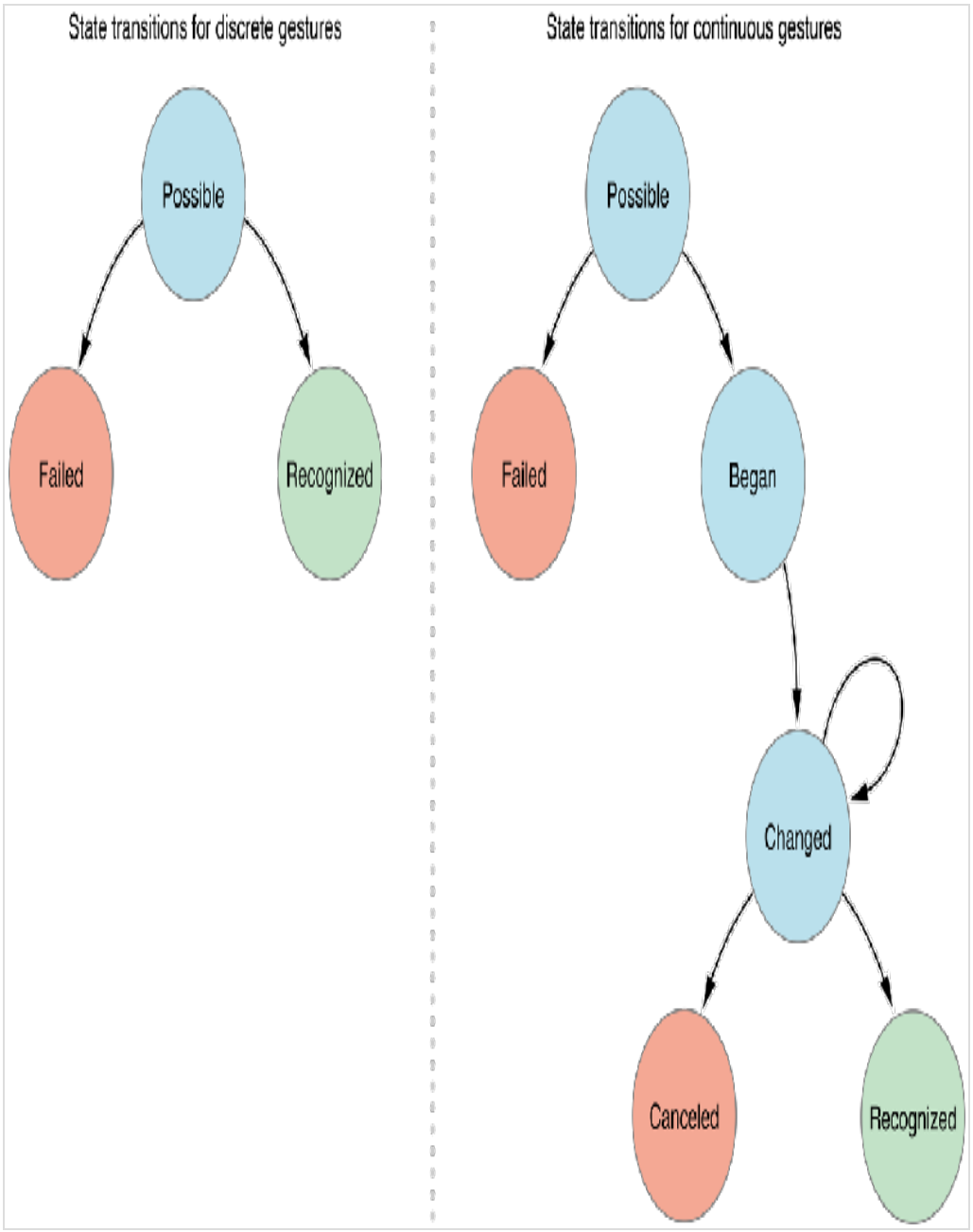
对于离散型手势UITapGestureRecgnizer要么被识别，要么失败，点按（假设点按次数设

置为1，并且没有添加长按手势）下去一次不松开则此时什么也不会发生，松开手指立即识别并调用操作事件，并且状态为3（已完成）。

但是连续手势要复杂一些，就拿旋转手势来说，如果两个手指点下去不做任何操作，此时并不能识别手势（因为我们还没旋转）但是其实已经触发了触摸开始事件，此时处于状态0；如果此时旋转会被识别，也就会调用对应的操作事件，同时状态变成1（手势开始），但是状态1只有一瞬间；紧接着状态变为2（因为我们的旋转需要持续一会），并且重复调用操作事件（如果在事件中打印状态会重复打印2）；松开手指，此时状态变为3，并调用1次操作事件。

为了大家更好的理解这个状态的变化，不妨在操作事件中打印事件状态，会发现在操作事件中的状态永远不可能为0（默认状态），因为只要调用此事件说明已经被识别了。前面也说过，手势识别从根本还是调用触摸事件而完成的，连续手势之所以会发生状态转换完全是由于触摸事件中的移动事件造成的，没有移动事件也就不存在这个过程中状态变化。

大家通过苹果官方的分析图再理解一下上面说的内容：



使用手势

在iOS中添加手势比较简单，可以归纳为以下几个步骤：

1. 创建对应的手势对象；
2. 设置手势识别属性【可选】；
3. 附加手势到指定的对象；
4. 编写手势操作方法；

为了帮助大家理解，下面以一个图片查看程序演示一下上面几种手势，在这个程序中我们完成以下功能：

如果点按图片会在导航栏显示图片名称；

如果长按图片会显示删除按钮，提示用户是否删除；

如果捏合会放大、缩小图片；

如果轻扫会切换到下一张或上一张图片；

如果旋转会旋转图片；

如果拖动会移动图片；

具体布局草图如下：

为了显示导航条，我们首先将主视图控制器KCPhotoViewController放入一个导航控制器，然后在主视图控制器中放一个UIImage用于展示图片。下面是主要代码：

```
//
//  KCGestureViewController.m
//  TouchEventAndGesture
//
//  Created by Kenshin Cui on 14-3-16.
//  Copyright (c) 2014年 Kenshin Cui. All rights reserved.
//

#import "KCPhotoViewController.h"
#define kImageCount 3

@interface KCPhotoViewController (){
    UIImageView *_imageView;//图片展示控件
    int _currentIndex;//当前图片索引
}

@end

@implementation KCPhotoViewController

- (void)viewDidLoad {
    [super viewDidLoad];

    [self initLayout];

    [self addGesture];
}
```

运行效果：

在上面示例中需要强调几点：

UIImageView默认是不支持交互的，也就是userInteractionEnabled=NO ，因此要接收触摸事件（手势识别），必须设置userInteractionEnabled=YES(在iOS中UILabel、UIImageView的userInteractionEnabled默认都是NO，UIButton、UITextField、UIScrollView、UITableView等默认都是YES)。

轻扫手势虽然是连续手势但是它的操作事件只会在识别结束时调用一次，其他连续手势都会调用多次，一般需要进行状态判断；此外轻扫手势支持四个方向，但是如果要支持多个方向需要添加多个轻扫手势。

手势冲突

细心的童鞋会发现在上面的演示效果图中当切换到下一张或者上一张图片时并没有轻扫图片而是在空白地方轻扫完成，原因是如果我轻扫图片会引起拖动手势而不是轻扫手势。换句话说，两种手势发生了冲突。

冲突的原因很简单，拖动手势的操作事件是在手势的开始状态（状态1）识别执行的，而轻扫手势的操作事件只有在手势结束状态（状态3）才能执行，因此轻扫手势就作为了牺牲品没有被正确识别。我们理想的情况当然是如果在图片上拖动就移动图片，如果在图片上轻扫就翻动图片。如何解决这个冲突呢？

在iOS中，如果一个手势A的识别部分是另一个手势B的子部分时，默认情况下A就会先识别，B就无法识别了。要解决这个冲突可以利用**- (void)requireGestureRecognizerToFail:(UIGestureRecognizer *)otherGestureRecognizer;**方法来完成。正是前面表格中UIGestureRecognizer的最后一个方法，这个方法可以指定某个手势执行的前提是另一个手势失败才会识别执行。也就是说如果我们指定拖动手势的执行前提为轻扫手势失败就可以了，这样一来当我们手指轻轻滑动时系统会优先考虑轻扫手势，如果最后发现该操作不是轻扫，那么就会执行拖动。只要将下面的代码添加到添加手势之后就能解决这个问题了（注意为了更加清晰的区分拖动和轻扫[模拟器中拖动稍微快一点就识别成了轻扫]，这里将长按手势的前提设置为拖动失败，避免演示拖动时长按手势会被识别）：

```
//解决在图片上滑动时拖动手势和轻扫手势的冲突
[panGesture requireGestureRecognizerToFail:swipeGestureToRight];
[panGesture requireGestureRecognizerToFail:swipeGestureToLeft];
//解决拖动和长按手势之间的冲突
[longPressGesture requireGestureRecognizerToFail:panGesture];
```

运行效果：

两个不同控件的手势同时执行

我们知道在iOS的触摸事件中，事件触发是根据响应者链进行的，上层触摸事件执行后就不再向下传播。默认情况下手势也是类似的，先识别的手势会阻断手势识别操作继续传播。那么如何让两个有层次关系并且都添加了手势的控件都能正确识别手势呢？答案就是利用代理的**-(BOOL)gestureRecognizer:(UIGestureRecognizer *)gestureRecognizer shouldRecognizeSimultaneouslyWithGestureRecognizer:(UIGestureRecognizer *)otherGestureRecognizer**方法。这个代理方法默认返回NO，会阻断继续向下识别手势，如果返回YES则可以继续向下传播识别。

下面的代码控制演示了当在图片上长按时同时可以识别控制器视图的长按手势（注意其中我们还控制了只有在UIImageView中操作的手势才能向下传递，如果不控制则所有控件都可以向下传递）

```
//
//  KCGestureViewController.m
//  TouchEventAndGesture
//
//  Created by Kenshin Cui on 14-3-16.
//  Copyright (c) 2014年 Kenshin Cui. All rights reserved.
//

#import "KCPhotoViewController.h"
#define kImageCount 3

@interface KCPhotoViewController ()<UIGestureRecognizerDelegate>{
    UIImageView *_imageView;//图片展示控件
    int _currentIndex;//当前图片索引
}

@end

@implementation KCPhotoViewController

- (void)viewDidLoad {
    [super viewDidLoad];

    [self initLayout];

    [self addGesture];
}
```

运动事件

前面我们主要介绍了触摸事件以及由触摸事件引出的手势识别，下面我们简单介绍一下运动事件。在iOS中和运动相关的有三个事件:开始运动、结束运动、取消运动。

监听运动事件对于UI控件有个前提就是监听对象必须是第一响应者（对于UIViewController视图控制器和UIAPPlication没有此要求）。这也就意味着如果监听的是一个UI控件那么**-(BOOL)canBecomeFirstResponder;**方法必须返回YES。同时控件显示时（在**-(void)viewWillAppear:(BOOL)animated;**事件中）调用视图控制器的**becomeFirstResponder**方法。当视图不再显示时（在**-(void)viewDidDisappear:(BOOL)animated;**事件中）注销第一响应者身份。

由于视图控制器默认就可以调用运动开始、运动结束事件在此不再举例。现在不妨假设我们现在在开发一个摇一摇找人的功能，这里我们就自定义一个图片展示控件，在这个图片控件中我们可以通过摇晃随机切换界面图片。代码比较简单：

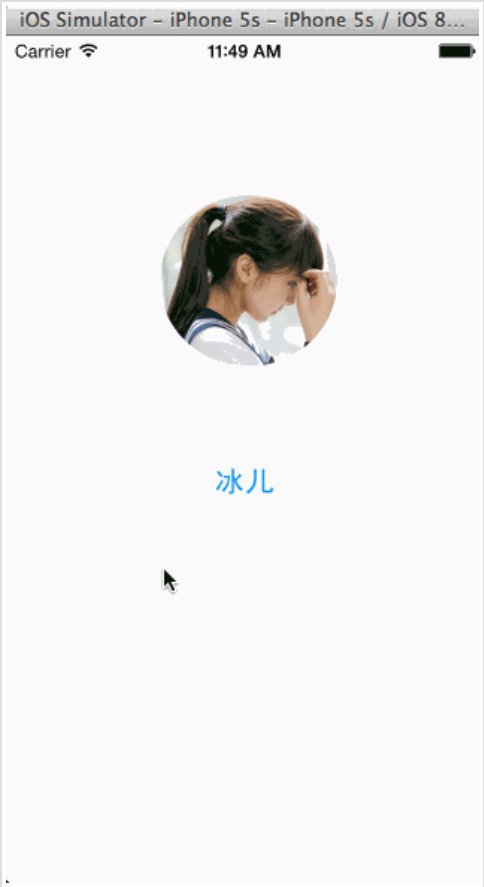
KCImageView.m

```
//
//  KCImageView.m
//  TouchEventAndGesture
//
//  Created by Kenshin Cui on 14-3-16.
//  Copyright (c) 2014年 Kenshin Cui. All rights reserved.
//

#import "KCImageView.h"
#define kImageCount 3

@implementation KCImageView

- (instancetype)initWithFrame:(CGRect)frame {
    self = [super initWithFrame:frame];
    if (self) {
        self.image=[self getImage];
    }
    return self;
}

#pragma mark 设置控件可以成为第一响应者
-(BOOL)canBecomeFirstResponder{
    return YES;
}

#pragma mark 运动开始
```

KCShakeViewController.m

```
//
//  KCShakeViewController.m
//  TouchEventAndGesture
//
//  Created by Kenshin Cui on 14-3-16.
//  Copyright (c) 2014年 Kenshin Cui. All rights reserved.
//

#import "KCShakeViewController.h"
#import "KCImageView.h"

@interface KCShakeViewController (){
    KCImageView *_imageView;
}

@end

@implementation KCShakeViewController

- (void)viewDidLoad {
    [super viewDidLoad];


}

#pragma mark 视图显示时让控件变成第一响应者
-(void)viewDidAppear:(BOOL)animated{
```

运行效果（下图演示时使用了模拟器摇晃操作的快捷键,没有使用鼠标操作）：



远程控制事件

在今天的文章中还剩下最后一类事件：远程控制，远程控制事件这里主要说的就是耳机线控操作。在前面的事件列表中，大家可以看到在iOS中和远程控制事件有关的只有一个**-(void)remoteControlReceivedWithEvent:(UIEvent *)event NS_AVAILABLE_IOS(4_0);**事件。要监听到这个事件有三个前提（视图控制器UIViewController或应用程序UIApplication只有两个）

> 启用远程事件接收（**使用[[UIApplication sharedApplication] beginReceivingRemoteControlEvents];**方法）。
>
> 对于UI控件同样要求必须是第一响应者（对于视图控制器UIViewController或者应用程序UIApplication对象监听无此要求）。
>
> 应用程序必须是当前音频的控制者，也就是在iOS 7中通知栏中当前音频播放程序必须是我们自己开发程序。

基于第三点我们必须明确，如果我们的程序不想要控制音频，只是想利用远程控制事件做其他的事情，例如模仿iOS7中的按音量+键拍照是做不到的，目前iOS7给我们的远程控制权限还仅限于音频控制（当然假设我们确实想要做一个和播放音频无关的应用但是又想进行远程控制，也可以隐藏一个音频播放操作，拿到远程控制操作权后进行远程控制）。

运动事件中我们也提到一个枚举类型UIEventSubtype，而且我们利用它来判断是否运动事件，在枚举中还包含了我们运程控制的子事件类型，我们先来熟悉一下这个枚举（从远程控制子事件类型也不难发现它和音频播放有密切关系）：

```
typedef NS_ENUM(NSInteger, UIEventSubtype) {
    // 不包含任何子事件类型
    UIEventSubtypeNone                              = 0,

    // 摇晃事件（从iOS3.0开始支持此事件）
    UIEventSubtypeMotionShake                       = 1,

    //远程控制子事件类型（从iOS4.0开始支持远程控制事件）
    //播放事件【操作：停止状态下，按耳机线控中间按钮一下】
    UIEventSubtypeRemoteControlPlay                 = 100,
    //暂停事件
    UIEventSubtypeRemoteControlPause                = 101,
    //停止事件
    UIEventSubtypeRemoteControlStop                 = 102,
    //播放或暂停切换【操作：播放或暂停状态下，按耳机线控中间按钮一下】
    UIEventSubtypeRemoteControlTogglePlayPause      = 103,
    //下一曲【操作：按耳机线控中间按钮两下】
    UIEventSubtypeRemoteControlNextTrack            = 104,
    //上一曲【操作：按耳机线控中间按钮三下】
    UIEventSubtypeRemoteControlPreviousTrack        = 105,
    //快退开始【操作：按耳机线控中间按钮三下不要松开】
    UIEventSubtypeRemoteControlBeginSeekingBackward = 106,
    //快退停止【操作：按耳机线控中间按钮三下到了快退的位置松开】
    UIEventSubtypeRemoteControlEndSeekingBackward   = 107,
    //快进开始【操作：按耳机线控中间按钮两下不要松开】
    UIEventSubtypeRemoteControlBeginSeekingForward  = 108,
    //快进停止【操作：按耳机线控中间按钮两下到了快进的位置松开】
    UIEventSubtypeRemoteControlEndSeekingForward    = 109,
```

这里我们将远程控制事件放到视图控制器（事实上很少直接添加到UI控件，一般就是添加到UIApplication或者UIViewController），模拟一个音乐播放器。

1.首先在应用程序启动后设置接收远程控制事件，并且设置音频会话保证后台运行可以播放（注意要在应用配置中设置允许多任务）

```
//
//  AppDelegate.m
//  TouchEventAndGesture
//
//  Created by Kenshin Cui on 14-3-16.
//  Copyright (c) 2014年 Kenshin Cui. All rights reserved.
//

#import "AppDelegate.h"
#import "ViewController.h"
#import <AVFoundation/AVFoundation.h>
#import "KCApplication.h"

@interface AppDelegate ()

@end

@implementation AppDelegate


- (BOOL)application:(UIApplication *)application didFinishLaunchingWithO
    _window=[[UIWindow alloc]initWithFrame:[UIScreen mainScreen].bounds]

    _window.backgroundColor =[UIColor colorWithRed:249/255.0 green:249/2

    //设置全局导航条风格和颜色
    [[UINavigationBar appearance] setBarTintColor:[UIColor colorWithRed:
```

2.在视图控制器中添加远程控制事件并音频播放进行控制

```
//
//  ViewController.m
//  RemoteEvent
//
//  Created by Kenshin Cui on 14-3-16.
//  Copyright (c) 2014年 Kenshin Cui. All rights reserved.
//

#import "ViewController.h"

@interface ViewController (){
    UIButton *_playButton;
    BOOL _isPlaying;
}

@end

@implementation ViewController

- (void)viewDidLoad {
    [super viewDidLoad];

    [self initLayout];
}

-(BOOL)canBecomeFirstResponder{
    return NO;
}
```

运行效果(真机截图)：

注意：

> 为了模拟一个真实的播放器，程序中我们启用了后台运行模式，配置方法：在info.plist中添加UIBackgroundModes并且添加一个元素值为