文章目录 站点概览

UIKit: UIControl

发表于 2015-12-13 | 分类于 Cocoa

我们在开发应用的时候，经常会用到各种各样的控件，诸如按钮(`UIButton`)、滑块(`UISlider`)、分页控件(`UIPageControl`)等。这些控件用来与用户进行交互，响应用户的操作。我们查看这些类的继承体系，可以看到它们都是继承于`UIControl`类。`UIControl`是控件类的基类，它是一个抽象基类，我们不能直接使用`UIControl`类来实例化控件，它只是为控件子类定义一些通用的接口，并提供一些基础实现，以在事件发生时，预处理这些消息并将它们发送到指定目标对象上。

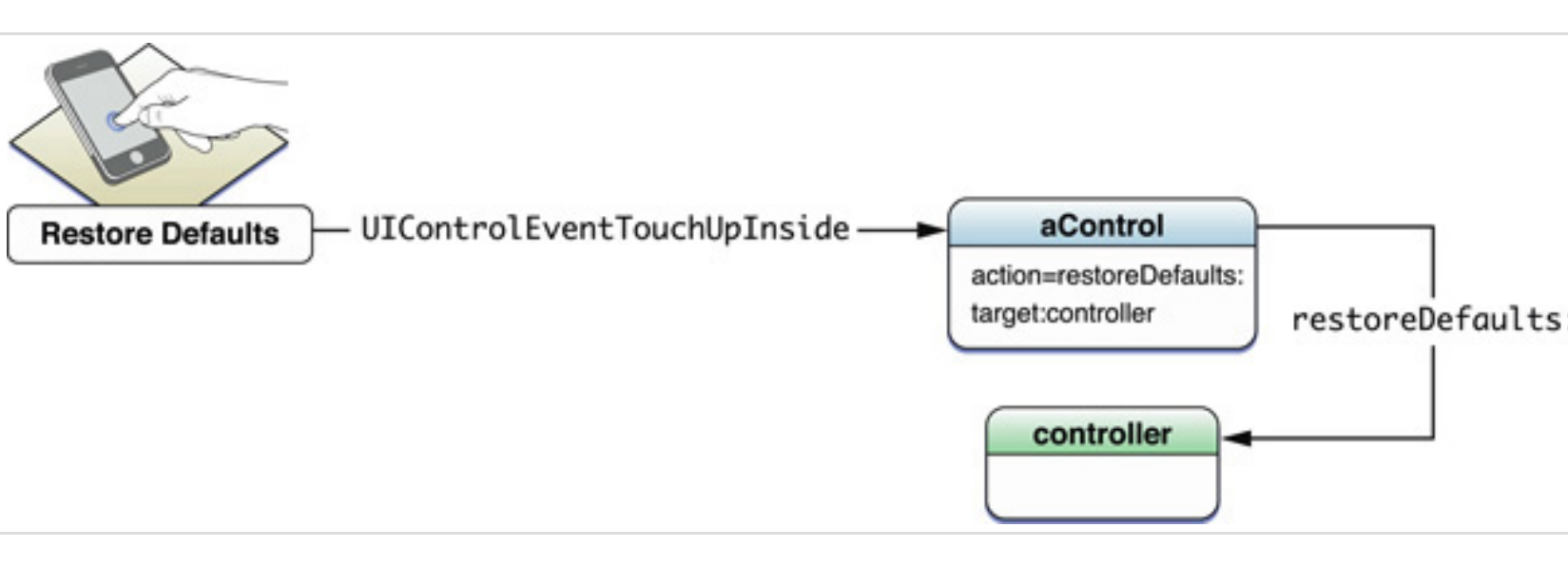
本文将通过一个自定义的`UIControl`子类来看看`UIControl`的基本使用方法。不过在开始之前，让我们先来了解一下`Target-Action`机制。

Target-Action机制

`Target-action`是一种设计模式，直译过来就是”目标-行为”。当我们通过代码为一个按钮添加一个点击事件时，通常是如下处理：

```
[button addTarget:self action:@selector(tapButton:) forControlEvents:UICon
```

也就是说，当按钮的点击事件发生时，会将消息发送到`target`(此处即为self对象)，并由`target`对象的`tapButton:`方法来处理相应的事件。其基本过程可以用下图来描述：



> 注：图片来源于官方文档Cocoa Application Competencies for iOS - Target Action

即当事件发生时，事件会被发送到控件对象中，然后再由这个控件对象去触发`target`对象上的`action`行为，来最终处理事件。因此，`Target-Action`机制由两部分组成：即目标对象和行为`Selector`。目标对象指定最终处理事件的对象，而行为`Selector`则是处理事件的方法。

有关`Target-Action`机制的具体描述，大家可以参考Cocoa Application Competencies for iOS - Target Action。我们将会在下面讨论一些`Target-action`更深入的东西。

实例：一个带Label的图片控件

回到我们的正题来，我们将实现一个带Label的图片控件。通常情况下，我们会基于以下两个原因来实现一个自定义的控件：

- 对于特定的事件，我们需要观察或修改分发到`target`对象的行为消息。
- 提供自定义的跟踪行为。

本例将会简单地结合这两者。先来看看效果：

这个控件很简单，以图片为背景，然后在下方显示一个Label。

先创建`UIControl`的一个子类，我们需要传入一个字符串和一个UIImage对象：

```
@interface ImageControl : UIControl

- (instancetype)initWithFrame:(CGRect)frame title:(NSString *)title image:

@end
```

基础的布局我们在此不讨论。我们先来看看`UIControl`为我们提供了哪些自定义跟踪行为的方法。

跟踪触摸事件

如果是想提供自定义的跟踪行为，则可以重写以下几个方法：

```
- (BOOL)beginTrackingWithTouch:(UITouch *)touch withEvent:(UIEvent *)event
- (BOOL)continueTrackingWithTouch:(UITouch *)touch withEvent:(UIEvent *)ev
- (void)endTrackingWithTouch:(UITouch *)touch withEvent:(UIEvent *)event
- (void)cancelTrackingWithEvent:(UIEvent *)event
```

这四个方法分别对应的时跟踪开始、移动、结束、取消四种状态。看起来是不是很熟悉？这跟`UIResponse`提供的四个事件跟踪方法是不是挺像的？我们来看看`UIResponse`的四个方法：

```
- (void)touchesBegan:(NSSet<UITouch *> *)touches withEvent:(UIEvent *)ever
- (void)touchesMoved:(NSSet<UITouch *> *)touches withEvent:(UIEvent *)ever
- (void)touchesEnded:(NSSet<UITouch *> *)touches withEvent:(UIEvent *)ever
- (void)touchesCancelled:(NSSet<UITouch *> *)touches withEvent:(UIEvent *)
```

我们可以看到，上面两组方法的参数基本相同，只不过`UIControl`的是针对单点触摸，而`UIResponse`可能是多点触摸。另外，返回值也是大同小异。由于`UIControl`本身是视图，所以它实际上也继承了`UIResponse`的这四个方法。如果测试一下，我们会发现在针对控件的触摸事件发生时，这两组方法都会被调用，而且互不干涉。

为了判断当前对象是否正在追踪触摸操作，`UIControl`定义了一个`tracking`属性。该值如果为YES，则表明正在追踪。这对于我们是更加方便了，不需要自己再去额外定义一个变量来做处理。

在测试中，我们可以发现当我们的触摸点沿着屏幕移出控件区域名，还是会继续追踪触摸操作，`cancelTrackingWithEvent:`消息并未被发送。为了判断当前触摸点是否在控件区域类，可以使用`touchInside`属性，这是个只读属性。不过实测的结果是，在控件区域周边一定范围内，该值还是会被标记为YES，即用于判定`touchInside`为YES的区域会比控件区域要大。

观察或修改分发到target对象的行为消息

对于一个给定的事件，`UIControl`会调用`sendAction:to:forEvent:`来将行为消息转发到`UIApplication`对象，再由`UIApplication`对象调用其`sendAction:to:fromSender:forEvent:`方法来将消息分发到指定的`target`上，而如果我们没有指定`target`，则会将事件分发到响应链上第一个想处理消息的对象上。而如果子类想监控或修改这种行为的话，则可以重写这个方法。

在我们的实例中，做了个小小的处理，将外部添加的`Target-Action`放在控件内部来处理事件，因此，我们的代码实现如下：

```
// ImageControl.m
- (void)sendAction:(SEL)action to:(id)target forEvent:(UIEvent *)event {
        // 将事件传递到对象本身来处理
    [super sendAction:@selector(handleAction:) to:self forEvent:event];
}

- (void)handleAction:(id)sender {

    NSLog(@"handle Action");
}

// ViewController.m

- (void)viewDidLoad {
    [super viewDidLoad];

    self.view.backgroundColor = [UIColor whiteColor];

    ImageControl *control = [[ImageControl alloc] initWithFrame:(CGRect)
    // ...

    [control addTarget:self action:@selector(tapImageControl:) forContro
}
- (void)tapImageControl:(id)sender {

    NSLog(@"sender = %@", sender);
}
```

由于我们重写了`sendAction:to:forEvent:`方法，所以最后处理事件的`Selector`是`ImageControl`的`handleAction:`方法，而不是ViewController的`tapImageControl:`方法。

另外，`sendAction:to:forEvent:`实际上也被`UIControl`的另一个方法所调用，即`sendActionsForControlEvents:`。这个方法的作用是发送与指定类型相关的所有行为消息。我们可以在任意位置(包括控件内部和外部)调用控件的这个方法来发送参数`controlEvents`指定的消息。在我们的示例中，在ViewController.m中作了如下测试：

```
- (void)viewDidLoad {
    // ...
    [control addTarget:self action:@selector(tapImageControl:) forControlE

    [control sendActionsForControlEvents:UIControlEventTouchUpInside];
}
```

可以看到在未点击控件的情况下，触发了`UIControlEventTouchUpInside`事件，并打印了`handle Action`日志。

Target-Action的管理

为一个控件对象添加、删除`Target-Action`的操作我们都已经很熟悉了，主要使用的是以下两个方法：

```
// 添加
- (void)addTarget:(id)target action:(SEL)action forControlEvents:(UIContr

- (void)removeTarget:(id)target action:(SEL)action forControlEvents:(UICon
```

如果想获取控件对象所有相关的target对象，则可以调用`allTargets`方法，该方法返回一个集合。集合中可能包含`NSNull`对象，表示至少有一个nil目标对象。

而如果想获取某个target对象及事件相关的所有action，则可以调用`actionsForTarget:forControlEvent:`方法。

不过，这些都是`UIControl`开放出来的接口。我们还是想要探究一下，`UIControl`是如何去管理`Target-Action`的呢？

实际上，我们在程序某个合适的位置打个断点来观察`UIControl`的内部结构，可以看到这样的结果：

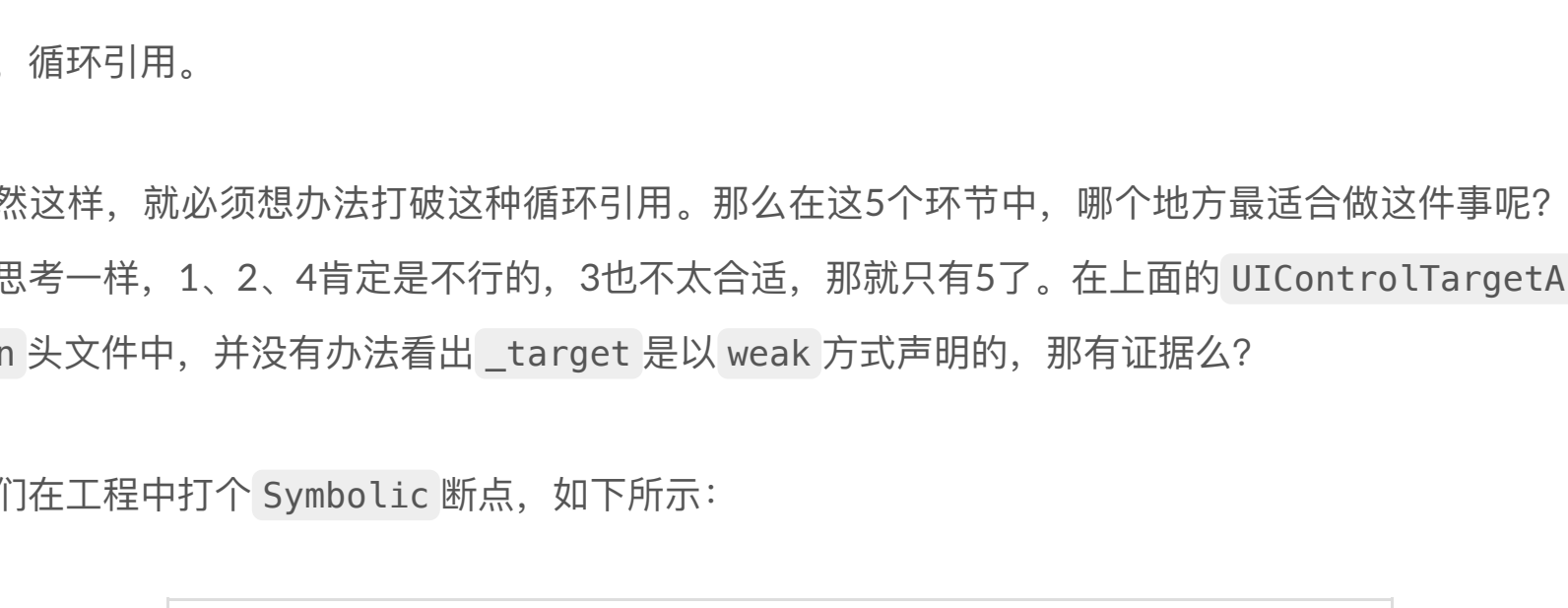
```
▼ L control = (ImageControl *) 0x7f94b867fc10
  ▼ UIControl
    ▶ UIView
    ▼ _targetActions = (__NSArrayM *) @"2 objects"
      ▼ [0] = (UIControlTargetAction *) 0x7f94b8437630
        ▶ NSObject
        ▶ _target = (ViewController *) 0x7f94b867abc0
        ▶ _action = (SEL) "touchDown:"
          _eventMask = (unsigned long long) 1
          _cancelled = (bool) false
      ▼ [1] = (UIControlTargetAction *) 0x7f94b8437500
        ▶ NSObject
        ▶ _target = (ViewController *) 0x7f94b867abc0
        ▶ _action = (SEL) "tapImageControl:"
          _eventMask = (unsigned long long) 64
          _cancelled = (bool) false
    _downTime = (double) 0
```

因此，`UIControl`内部实际上是有一个可变数组(`_targetActions`)来保存`Target-Action`，数组中的每个元素是一个`UIControlTargetAction`对象。`UIControlTargetAction`类是一个私有类，我们可以在iOS-Runtime-Header中找到它的头文件：

```
@interface UIControlTargetAction : NSObject {
    SEL _action;
    BOOL _cancelled;
    unsigned int _eventMask;
    id _target;
}

@property (nonatomic) BOOL cancelled;

- (void).cxx_destruct;
- (BOOL)cancelled;
- (void)setCancelled:(BOOL)arg1;

@end
```

可以看到`UIControlTargetAction`对象维护了一个`Target-Action`所必须的三要素，即`target`，`action`及对应的事件`eventMask`。

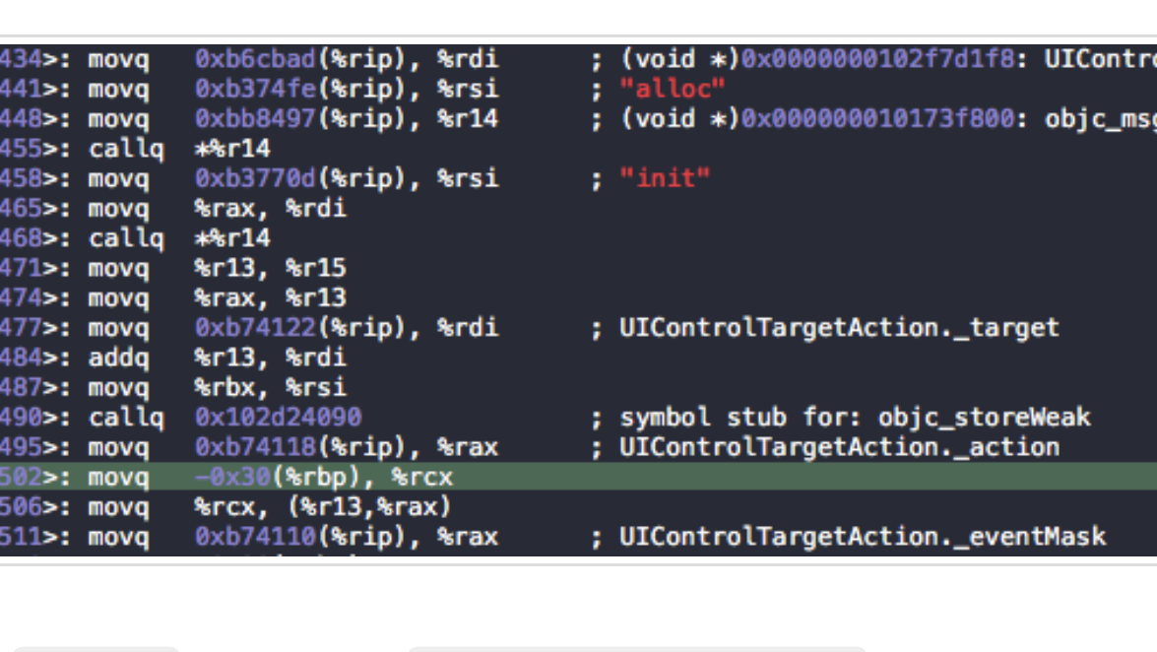
如果仔细想想，会发现一个有意思的问题。我们来看看实例中ViewController(target)与ImageControl实例(control)的引用关系，如下图所示：



嗯，循环引用。

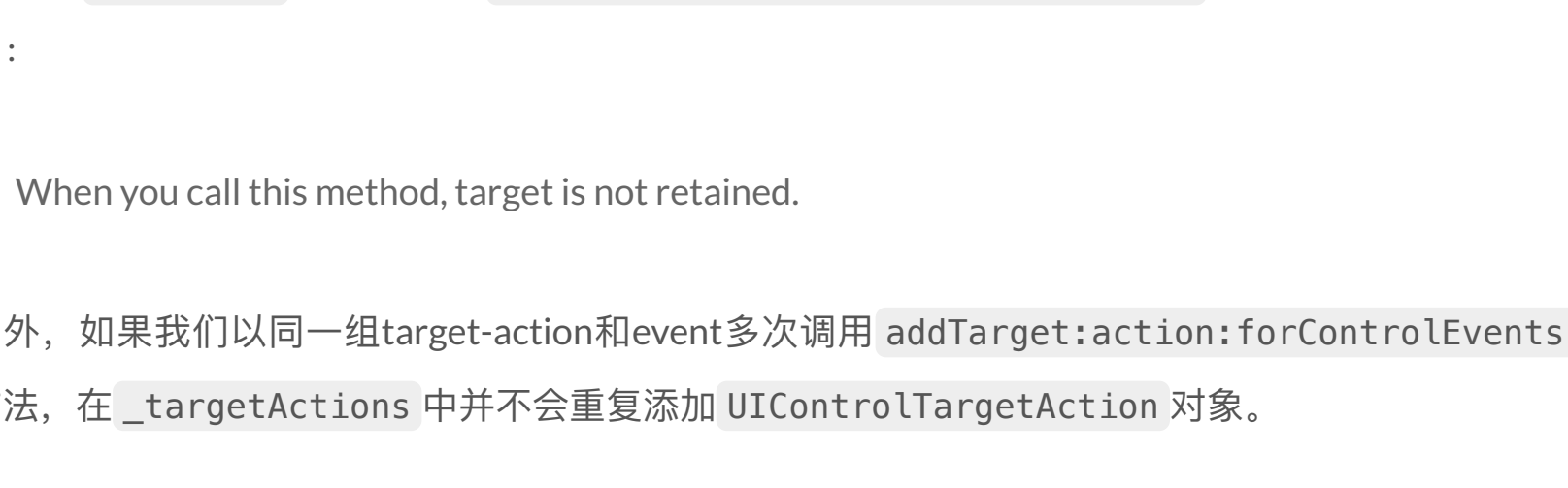
既然这样，就必须想办法打破这种循环引用。那么在这5个环节中，哪个地方最适合做这件事呢？仔细思考一样，1、2、4肯定是不行的，3也不太合适，那就只有5了。在上面的`UIControlTargetAction`头文件中，并没有办法看出`_target`是以`weak`方式声明的，那有证据么？

我们在工程中打个`Symbolic`断点，如下所示：



☑ Symbolic Breakpoint
Symbol: -[UIControl addTarget:action:forControlEvents:]
Module: Example: "libSystem.B.dylib"
Condition:
Ignore: 0 times before stopping
Action: Add Action
Options: ☐ Automatically continue after evaluating actions

运行程序，程序会进入`[UIControl addTarget:action:forControlEvents:]`方法的汇编代码页，在这里，我们可以找到一些蛛丝马迹。如下图所示：



```
0x1023f4d04 <+434>: movq   0xb6cbad(%rip), %rdi      ; (void *)0x0000000102f7d1f8: UIControlTargetAction
0x1023f4d0b <+441>: movq   0xb374fe(%rip), %rsi      ; "alloc"
0x1023f4d12 <+448>: movq   0xbb8497(%rip), %r14      ; (void *)0x000000010173f800: objc_msgSend
0x1023f4d19 <+455>: callq  *%r14
0x1023f4d1c <+458>: movq   0xb3770d(%rip), %rsi      ; "init"
0x1023f4d23 <+465>: movq   %rax, %rdi
0x1023f4d26 <+468>: callq  *%r14
0x1023f4d29 <+471>: movq   %r13, %r15
0x1023f4d2c <+474>: movq   %rax, %r13
0x1023f4d2f <+477>: movq   0xb74122(%rip), %rdi      ; UIControlTargetAction._target
0x1023f4d36 <+484>: addq   %r13, %rdi
0x1023f4d39 <+487>: movq   %rbx, %rsi
0x1023f4d3c <+490>: callq  0x102d24090               ; symbol stub for: objc_storeWeak
0x1023f4d41 <+495>: movq   0xb74118(%rip), %rax      ; UIControlTargetAction._action
0x1023f4d48 <+502>: movq   -0x30(%rbp), %rcx
0x1023f4d4c <+506>: movq   %rcx, (%r13,%rax)
0x1023f4d51 <+511>: movq   0xb74110(%rip), %rax      ; UIControlTargetAction._eventMask
```

可以看到，对于`_target`成员变量，在`UIControlTargetAction`的初始化方法中调用了`objc_storeWeak`，即这个成员变量对外部传进来的`target`对象是以`weak`的方式引用的。

其实在`UIControl`的文档中，`addTarget:action:forControlEvents:`方法的说明还有这么一句：

> When you call this method, target is not retained.

另外，如果我们以同一组target-action和event多次调用`addTarget:action:forControlEvents:`方法，在`_targetActions`中并不会重复添加`UIControlTargetAction`对象。

小结

控件是我们在开发中常用的视图工具，能很好的表达用户的意图。我们可以使用UIKit提供的控件，也可以自定义控件。当然，`UIControl`除了上述的一些方法，还有一些属性和方法，以及一些常量，大家可以参考文档。

示例工程的代码已上传到github，可以在这里下载。另外，推荐一下SVSegmentedControl这个控件，大家可以研究下它的实现。

参考

1. UIControl Class Reference
2. UIKit User Interface Catalog - About Controls
3. Cocoa Application Competencies for iOS - Target Action
4. iOS-Runtime-Header: UIControlTargetAction
5. SVSegmentedControl

UIKit: UIImage
Perfect smooth scrolling in UITableViews