**iOS中消息的传递机制**

* [beyondvincent.com](http://getpocket.com/redirect?url=http%3A%2F%2FBeyondVincent.github.io%2Fblog%2F2013%2F12%2F14%2F124-communication-patterns%2F)
* [View Original](http://getpocket.com/redirect?url=http%3A%2F%2FBeyondVincent.github.io%2Fblog%2F2013%2F12%2F14%2F124-communication-patterns%2F)



注1：本文由破船译自[Communication Patterns](http://www.objc.io/issue-7/communication-patterns.html)。

本文目录如下所示：

每个应用程序或多或少，都由一些松耦合的对象构成，这些对象彼此之间要想很好的完成任务，就需要进行消息传递。本文将介绍所有可用的消息传递机制，并通过示例来介绍这些机制在苹果的Framework中如何使用，同时，还介绍了一些最佳实践建议，告诉你什么时机该选择使用什么机制。

虽然这一期的主题是关于Foundation Framework的，不过本文中还介绍了一些超出Foundation Framework(KVO和Notification)范围的一些消息传递机制，另外还介绍了delegation，block和target-action。

大多数情况下，消息传递该使用什么机制，是很明确的了，当然了，在某些情况下该使用什么机制并没有明确的答案，需要你亲自去尝试一下。

本文中，会经常提及接收者[recipient]和发送者[sender]。在消息传递机制中具体是什么意思，我们可以通过一个示例来解释：一个table view是发送者，而它的delegate就是接收者。Core Data managed object context是notification的发送者，而获取这些notification的主体则是接收者。一个滑块(slider)是action消息的发送者，而在代码里面对应着实现这个action的responder就是接收者。对象中的某个属性支持KVO，那么谁修改这个值，谁就是发送者，对应的观察者(observer)则是接收者。

**可用的机制**

首先我们来看看每种机制的具体特点。在下一节中，我会结合一个流程图来介绍如何在具体情况下，选择正确的消息传递机制。最后，将介绍一些来自苹果Framework中的示例，并会解释在某种确定情况下为什么要选择固定的机制。

**KVO**

KVO提供了这样一种机制：当对象中的某个属性值发生了改变，可以对这些值的观察者做出通知。KVO的实现包含在Foundation里面，基于Foundation构建的许多Framework对KVO都有所依赖。要想了解更多关于如何使用KVO，可以阅读本期由Daniel写的的[KVO和KVC](http://www.objc.io/issue-7/key-value-coding-and-observing.html)文章。

如果对某个对象中值的改变情况感兴趣，那么可以使用KVO消息传递机制。这里有两个要求，首先，接收者(会接收到值发生改变的消息)必须知道发送者(值将发生改变的那个对象)。另外，接收者同样还需要知道发送者的生命周期，因为在销毁发送者对象之前，需要取消观察者的注册。如果这两个要求都满足了，消息传递过程中可以是1对多(多个观察者可以注册某个对象中的值)。

如果计划在Core Data对象上使用KVO，需要知道这跟一般的KVO使用方法有点不同。那就是必须结合Core Data的故障机制(faulting mechanism)，一旦core data出现了故障，它将会触发其属性对应的观察者(即使这些属性值没有发生改变)。

**Notification**

在不相关的两部分代码中要想进行消息传递，通知(notifacation)是非常好的一种机制，它可以对消息进行广播。特别是想要传递丰富的信息，并且不一定指望有谁对此消息关心。

通知可以用来发送任意的消息，甚至包含一个userInfo字典，或者是NSNotifacation的一个子类。通知的独特之处就在于发送者和接收者双方并不需要相互知道。这样就可以在非常松耦合的模块间进行消息的传递。记住，这种消息传递机制是单向的，作为接收者是不可以回复消息的。

**delegation**

在苹果的Framework中，delegation模式被广泛的只用着。delegation允许我们定制某个对象的行为，并且可以收到某些确定的事件。为了使用delegation模式，消息的发送者需要知道消息的接收者(delegate)，反过来就不用了。这里的发送者和接收者是比较松耦合的，因为发送者只知道它的delegate是遵循某个特定的协议。

delegate协议可以定义任意的方法，因此你可以准确的定义出你所需要的类型。你可以用函数参数的形式来处理消息内容，delegate还可以通过返回值的形式给发送者做出回应。如果只需要在相对接近的两个模块之间进行消息传递，那么Delegation是一种非常灵活和直接方式。

不过，过渡使用delegation也有一定的风险，如果两个对象的耦合程度比较紧密，相互之间不能独立存在，那么此时就没有必要使用delegate协议了，针对这种情况，对象之间可以知道相互间的类型，进而直接进行消息传递。例如UICollectionViewLayout和NSURLSessionConfiguration。

**block**

Block相对来说，是一种比较新的技术，它首次出现是在OS X 10.6和iOS 4中。一般情况下，block可以满足用delegation实现的消息传递机制。不过这两种机制都有各自的需求和优势。

当不考虑使用block时，一般主要是考虑到block极易引起[retain环](https://developer.apple.com/library/mac/documentation/cocoa/conceptual/memorymgmt/Articles/mmPractical.html#//apple_ref/doc/uid/TP40004447-1000810)。如果发送者需要reatain block，而又不能确保这个引用什么时候被nil，这样就会发生潜在的retain环。

假设我们想要实现一个table view，使用block替代delegate，来当做selection的回调，如下：

上面代码的问题在于self retain了table view，而table view为了之后能够使用block，进而 retain了block。而table view又不能把这个引用nil掉，因为它不知道什么时候不在需要这个block了。如果我们保证不了可以打破这个retain环，而我们又需要retain发送者，此时block不是好的选择。

NSOperation就可以很好的使用block，因为它能再某个时机打破retain环：

乍一看这似乎是一个retain环：self retain了queue，queue retain了operation，而operation retain了completion block，而completion blockretain了self。不过，在这里，将operation添加到queue时，会使operation在某个时机被执行，然后从queue中remove掉（如果没有被执行，就会有大问题了）。一单queue移除了operation之后，retain环就被打破了。

再来一个示例：这里实现了一个视频编码器的类，里面有一个名为encodeWithCompletionHandler:的方法。为了避免出现retain环，我们需要确保编码器这个对象能够在某个时机nil掉其对block的引用。其内部代码如下所示：

在上面的代码中，一旦编码任务完成，就会调用complietion block，进而把引用nil掉。

如果我们发送的消息属于一次性的(具体到某个方法的调用)，由于这样可以打破潜在的retain环，那么使用block是非常不错的选择。另外，如果为了让代码可读性更强，更有连贯性，那最好是使用block了。根据这个思路，block经常可以用于completion handler、error handler等。

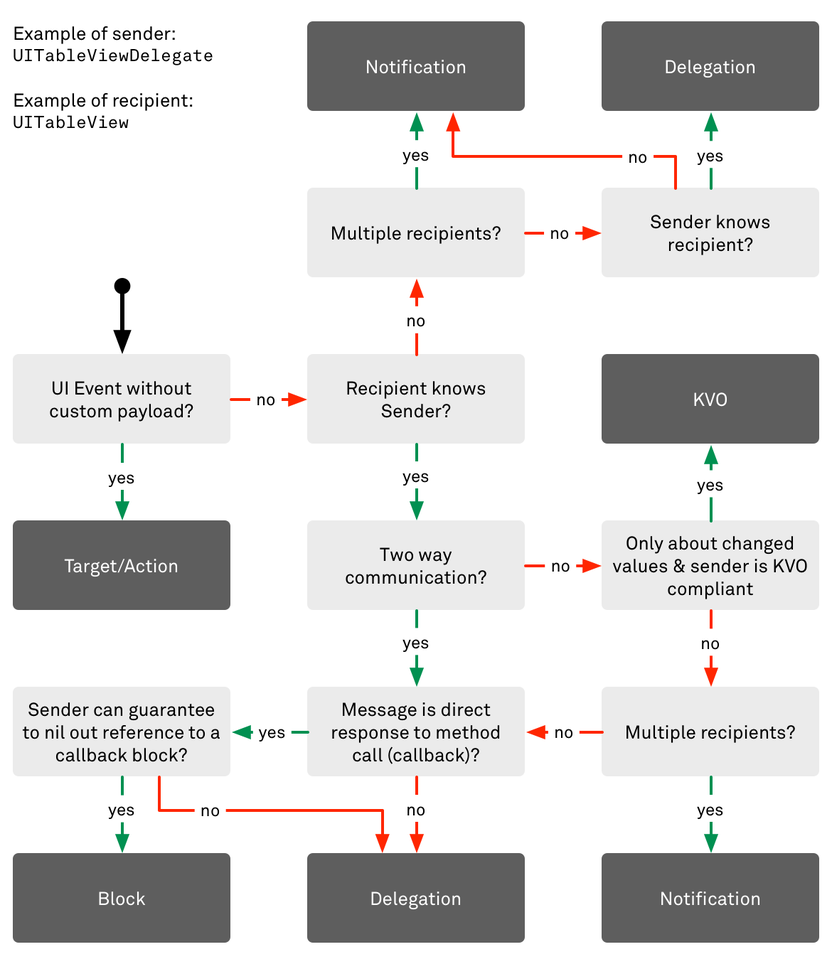
**Target-Action**

Target-Action主要被用于响应用户界面事件时所需要传递的消息中。iOS中的UIControl和Mac中的NSControl/NSCell都支持这种机制。Target-Action在消息的发送者和接收者之间建立了一个非常松散耦合。消息的接收者不知道发送者，甚至消息的发送者不需要预先知道消息的接收者。如果target是nil，action会在[响应链(responder chain)](https://developer.apple.com/library/ios/documentation/general/conceptual/Devpedia-CocoaApp/Responder.html)中被传递，知道找到某个能够响应该aciton的对象。在iOS中，每个控件都能关联多个target-action。

基于target-action消息传递的机制有一个局限就是发送的消息不能携带自定义的payload。在Mac的action方法中，接收者总是被放在第一个参数中。而在iOS中，可以选择性的将发送者和和触发action的事件作为参数。除此之外，没有别的办法可以对发送action消息内容做控制。

**做出正确的选择**

根据上面讨论的结果，这里我画了一个流程图，来帮助我们何时使用什么消息传递机制做出更好的决定。忠告：流程图中的建议并非最终的答案；可能还有别的选项依然能实现目的。只不过大多数情况下此图可以引导你做出正确的决定。



上图中，还有一些细节需要做更近一步的解释：

上图中的有个盒子这样说到：sender is KVO compliant(发送者支持compliant)。这不仅以意味着当值发生改变时，发送者会发送KVO通知，并且观察者还需要知道发送者的生命周期。如果发送者被存储在一个weak属性中，那么发送者有可能被nil掉，进而引起观察者发生leak。

另外底部的一个盒子说到：message is direct response to method call(消息直接在方法的调用代码中响应)。也就是说处理消息的代码跟方法的调用代码处于相同的地方。

最后，在左下角，处于一个决策问题的判断状态：sender can guarantee to nil out reference to block?(发送者能够确保nil掉到block的引用吗？)，这实际上涉及到之前我们讨论到基于block 的APIs已经潜在的retain环。使用block时，如果发送者不能保证在某个实际能够把对block的引用nil掉，那么将会遇到retain环的问题。

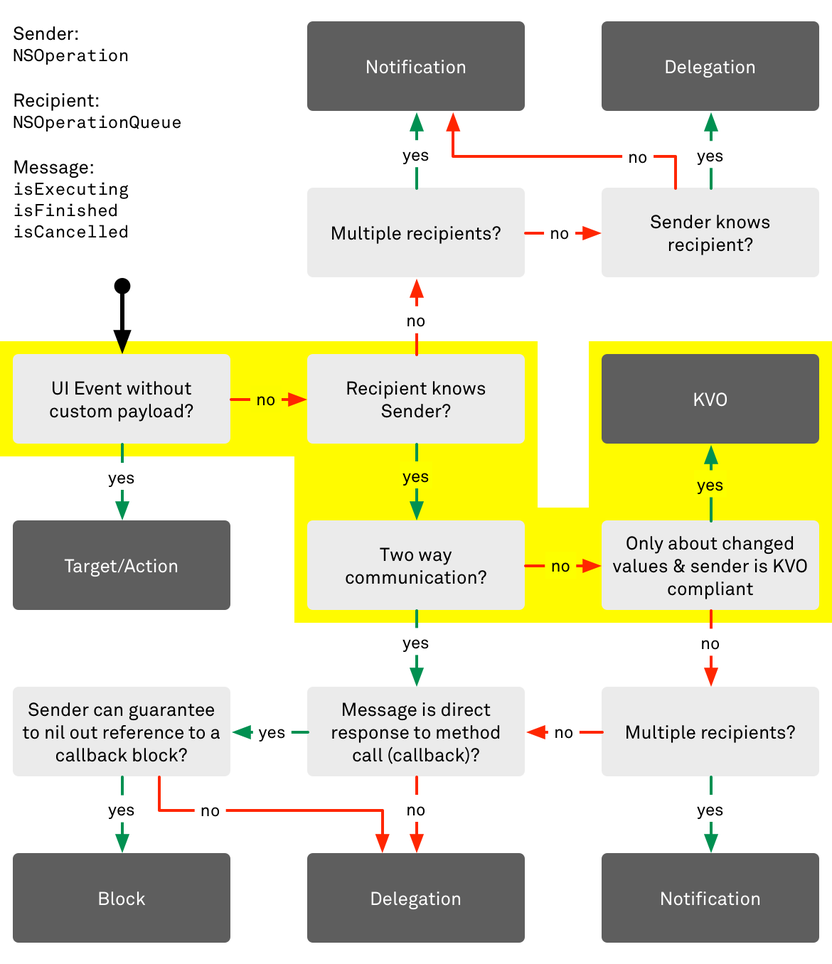
**Framework示例**

本节我们通过一些来自苹果Framework的示例，来看看在实际使用某种机制之前，苹果是处于何种原因做出选择的。

**KVO**

NSOperationQueue就是lion给了KVO来观察队列中operation状态属性的改变情况(isFinished, isExecuting, isCancelled)。当状态发生了改变，队列会受到一个KVO通知。为什么operationqueue要是用KVO呢？

消息的接收者(operation queue)明确的知道发送者(opertation)，以及通过retain来控制operation的生命周期。另外，在这种情况下，只需要单向的消息传递机制。当然，如果这样考虑：如果operation queue只关心operation值的改变情况，可能还不足以说服大家使用KVO。但是我们至少可以这样理解：什么机制可以对值的改变进行消息传递呢。

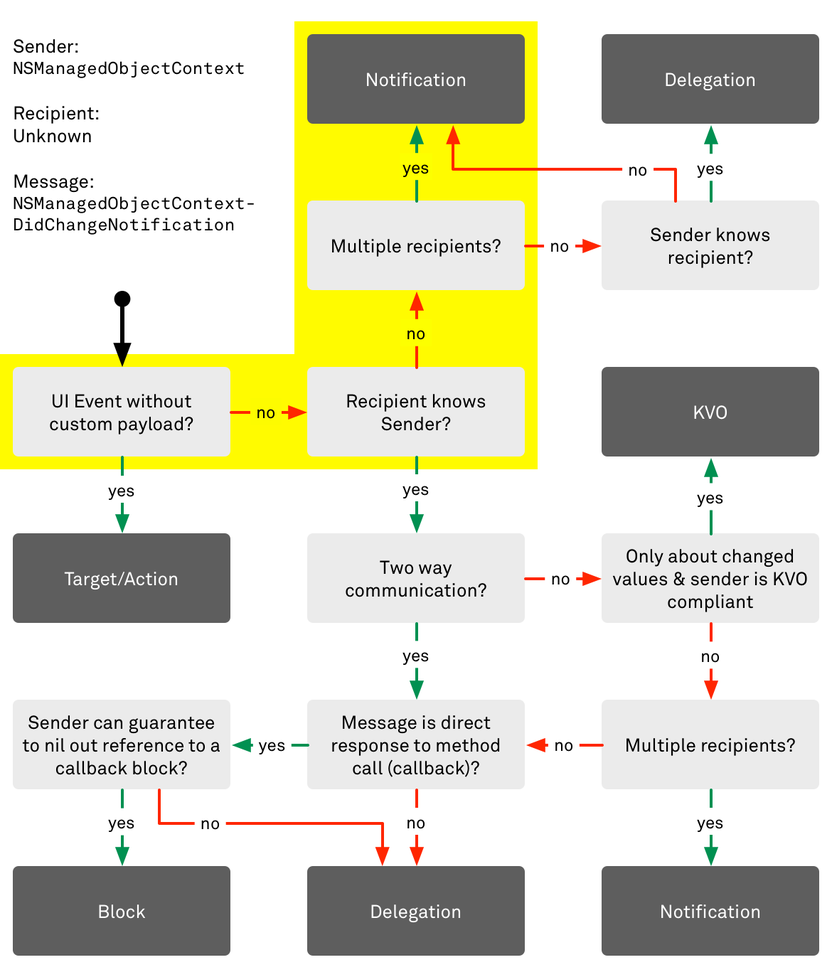


当然KVO也不是唯一的选择。我们可以这样设计：operation queue作为operation的delegate，operation会调用类似operationDidFinish: 或 operationDidBeginExecuting: 这样的方法，来将它的state传递给queue。这样一来，就不太方便了，因为operation需要将其state属性保存下来，一遍调用这些delegate方法。另外，由于queue不能主动获取state信息，所以queue也必须保存着所有operation的state。

**Notifications**

Core Data使用notification来传递事件(例如一个managed object context内部的改变——NSManagedObjectContextDidChangeNotification)。

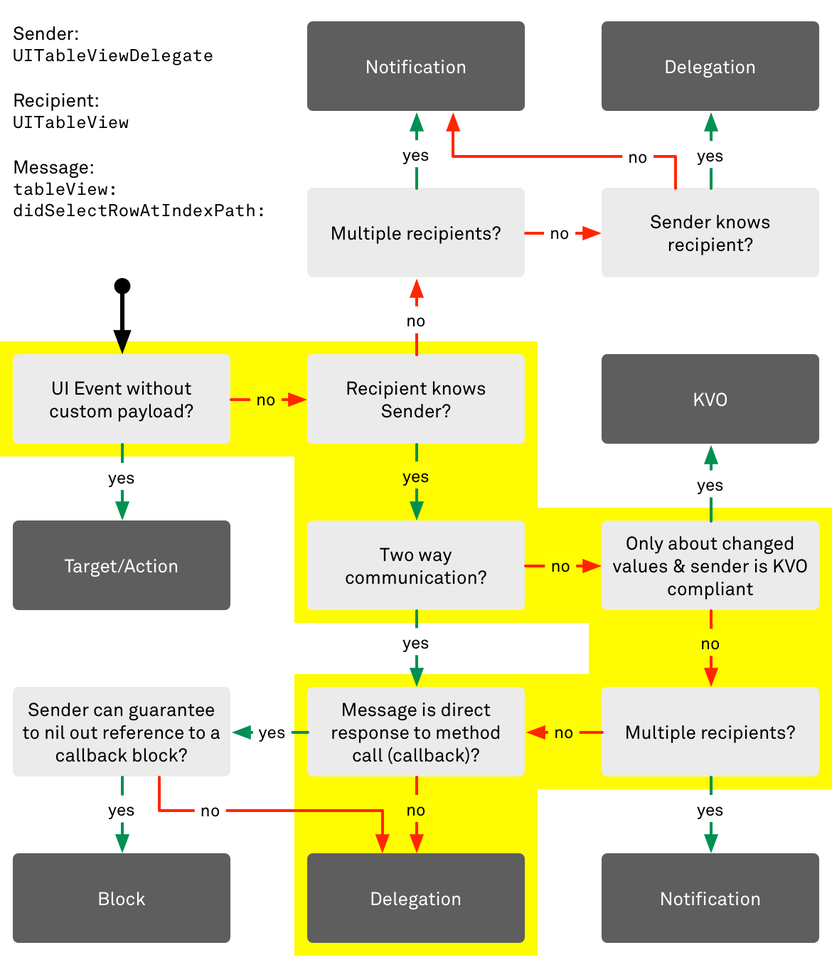
change notification是由managed object context发出的，所以我们不能确定消息的接收者一定知道发送者。如果消息并不是一个UI事件，而有可能多个接收者对该消息感兴趣，并且消息的传递属于单向(one-way communication channel)，那么notification是最佳选择。



**Delegation**

Table view的delegate有多种功能，从accessory view的管理，到屏幕中cell显示的跟踪，都与delegate的功劳。例如，我们来看看 tableView:didSelectRowAtIndexPath: 方法。为什么要以delegate调用的方式来实现？而又为啥不用target-action方式？

正如我们在流程图中看到的一样，使用target-action时，不能传递自定义的数据。而在选中table view的某个cell时，collection view不仅仅需要告诉我们有一个cell被选中了，还需要告诉我们是哪个cell被选中了(index path)。按照这样的一种思路，那么从流程图中可以看到应该使用delegation机制。

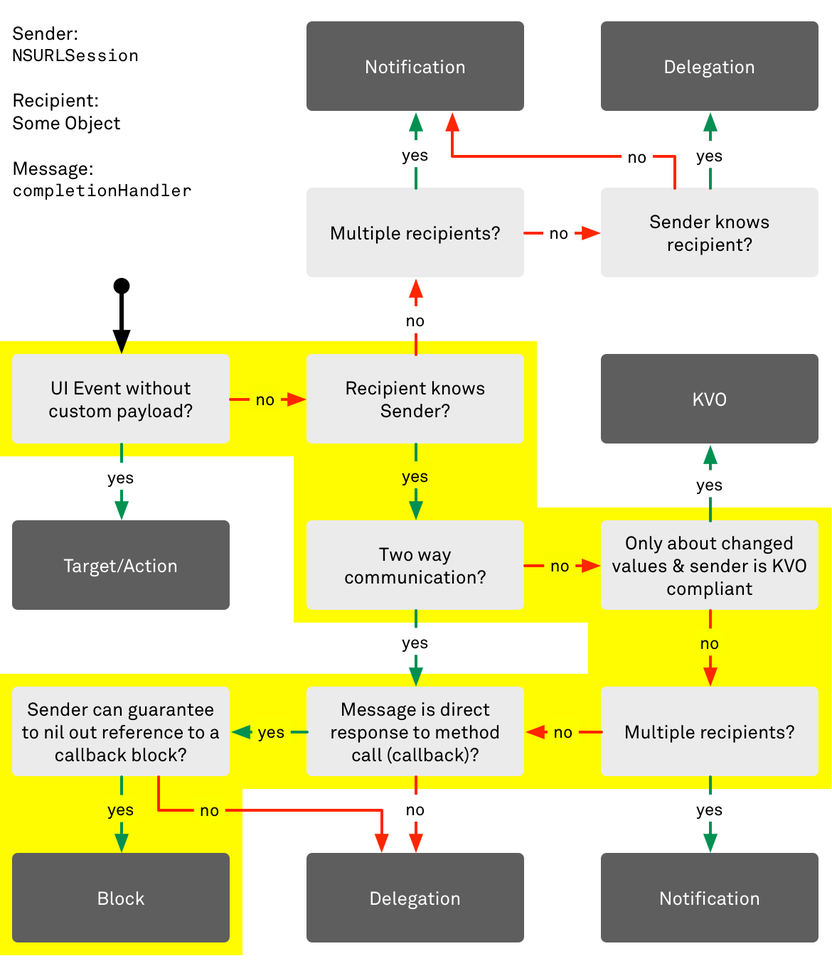


如果消息传递中，不包含选中cell的index path，而是每当选中项改变时，我们主动去table view中获取到选中cell的相关信息，会怎样呢？其实这会非常的麻烦，因为这样一来，我们就必须记住当前选中项相关数据，以便获知被选中的cell。

同理，虽然我们也可以通过观察table view中选中项的index paths属性值，当该值发生改变时，获得一个选中项改变的通知。不过，我们会遇到与上面同样的问题：不做任何记录的话，我们如何获知被选中项的相关信息。

**Blocks**

关于block的介绍，我们来看看[NSURLSession dataTaskWithURL:completionHandler:]吧。从URL loading system返回到调用者，这个过程具体是如何传递消息的呢？首先，作为这个API的调用者，我们知道消息的发送者，但是我们并没有retain这个发送者。另外，这属于单向消息传递——直接调用dataTaskWithURL:方法。如果按照这样的思路对照着流程图，我们会发现应该使用基于block消息传递的机制。

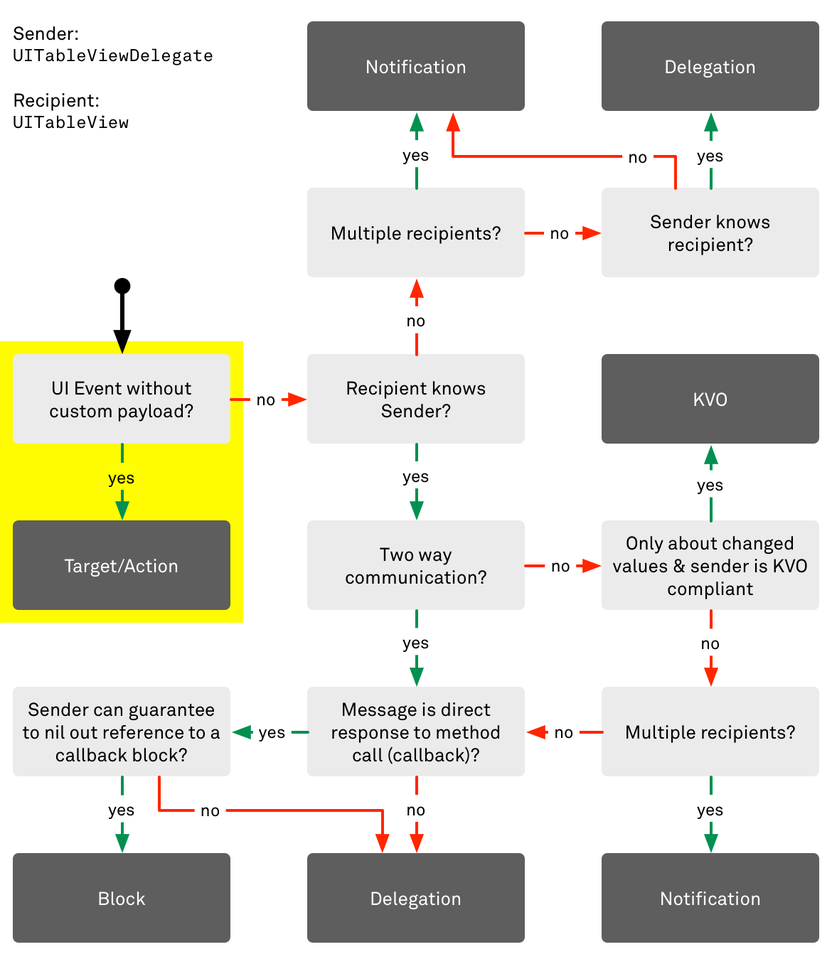


还有其它可选的机制吗？当然有了，苹果自己的NSURLConnection就是最好的例子。NSURLConnection在block问世之前就已经存在了，所以它并没有利用block进行消息传递，而是使用delegation机制。当block出现之后，苹果在NSURLConnection中添加了sendAsynchronousRequest:queue:completionHandler:方法(OSX 10.7 iOS 5)，因此如果是简单的task，就不必在使用delegate了。

在OS X 10.9 和 iOS 7中，苹果引入了一个非常modern的API：NSURLSession，其中使用block当做消息传递机制(NSURLSession仍然有一个delegate，不过是用于别的目的)。

**Target-Action**

Target-Action用的最明显的一个地方就是button(按钮)。button除了需要发送一个click事件以外，并不需要再发送别的信息了。所以Target-Action在用户界面事件传递过程中，是最佳的选择。



如果taget已经明确指定了，那么action消息回直接发送给指定的对象。如果taget是nil，action消息会以冒泡的方式在响应链中查找一个能够处理该消息的对象。此时，我们拥有一种完全解耦的消息传递机制——发送者不需要知道接收者，以及其它一些信息。

Target-Action非常适用于用户界面中的事件。目前也没有其它合适的消息传递机制能够提供同样的功能。虽然notification最接近这种在发送者和接收者解耦关系，但是target-action可以用于响应链(responder chain)——只有一个对象获得action并作出响应，并且action可以在响应链中传递，直到遇到能够响应该action的对象。

**小结**

首次接触这些机制，感觉它们都能用于两个对象间的消息传递。但是仔细琢磨一番，会发现它们各自有其需求和功能。

文中给出的决策流程图可以为我们选择使用何种机制提供参考，不过图中给出的方案并不是最终答案，好多地方还需要亲自去实践。