ios局域网联机——苹果官方源码之WiTap剖析

2012-4-12 12:25| 发布者: [benben](http://space.189works.com/space-uid-10034751.html)| 查看: 574| 评论: 0

**摘要**: 在过去的时间里，我一直在考虑的事情是，我该写一篇什么样的文章呢？之前的两篇文章都是先有问题，然后我才有目的的解决问题，现在我的困扰是，我不知道该写什么了呵呵。因为其实，大多数的问题，只要在网上搜索一下 ...

|  |
| --- |
| 在过去的时间里，我一直在考虑的事情是，我该写一篇什么样的文章呢？之前的两篇文章都是先有问题，然后我才有目的的解决问题，现在我的困扰是，我不知道该写什么了呵呵。因为其实，大多数的问题，只要在网上搜索一下（google远比baidu要强得多）,基本上都能找到解决的办法，已经有了许多相关方面的教程或参考资料了，我并不是一个喜欢重复做别人已经做得很好的工作的人，所以我现在需要你的帮助，如果你有好的关于写什么方面的文章的建议，请留言告诉我（**声明：应用我并不在行！**），如果我能实现的话，一定会写出来分享给大家，如果写不出来，大家一起讨论下解决也是很好的！！谢谢！！我甚至翻译了raywenderlich的“怎么在[ios5](http://bbs.189works.com/)上做一个简单的[iphone](http://bbs.189works.com/)应用程序系列”的第一部分，但是当我想要把他发布的时候我放弃了，这个不是我擅长的，子龙山人博客翻译团队做这个更专业，我不应该把这种翻译的文章放在我自己的博客上，所以我想我还是把raywenderlich的这个“怎么在[ios](http://bbs.189works.com/)5上做一个简单的[iphone应用](http://bbs.189works.com/)程序系列”的3部分都翻译完后在送给山人比较好（当然得在人家同意的前提下哈哈）。       最终，我觉得把一些经典的源码分析一下也许会是一个好主意，所以，今天我要写的是苹果官方的源码witap例子的分析。所以，首先你需要下载这个[官方的源码](http://www.189works.com/data/attachment/portal/et2/201204/ET27925201204121203261.zip)。  前提       我们文章的标题已经揭示了这个witap例子的内容是局域网联机的，这个对联机游戏来说真的很有用，联机的话你需要真机设备才能体验到，正常情况下你需要两个真机，不过其实一个真机加一个模拟器也是可以的，我在学习这个例子的时候就是一个真机加模拟器的组合呵呵.(人穷没办法呀?)       首先我们先在模拟器上运行一下这个例子好让我们对这个程序先有个直观的感受。  http://www.189works.com/data/attachment/portal/et2/201204/ET27925201204121203262.png       没有太多的东西，一个状态栏，下面是一个UIView，这个UIView里有一些元素（每个元素，在后面的具体代码里我们会一一指出），总之，现在界面上显示的是3条信息，一条是提示我们等待另一个玩家加入，下面一条是设备的名字（这里我是在模拟器上运行的，所以显示的是我的计算机的名字），最后一条是提示我们或者要加入另一个游戏。我们随便在屏幕上点点看，没有任何反应。我们还是乖乖的听话，让另一个玩家加入我们吧，不然真的我们什么都做不了，首先确保你的两台设备（我是我的电脑和touch）都在同一个网络内，然后打开真机上的这个程序。  http://www.189works.com/data/attachment/portal/et2/201204/ET27925201204121203263.png  look,我们的模拟器发现了我的touch，哈哈，同样的，在真机的tableView里，你也会看到你的模拟器的名字。现在两台机器已经发现彼此了，然我们点一下tableView里的名字试试看吧，有反应了，我们进入了苹果给我们带来的小游戏：  http://www.189works.com/data/attachment/portal/et2/201204/ET27925201204121203264.png       点击弹出通知的continue来继续游戏，在游戏里随便点点，我们点击屏幕上的任意一个色块，我们的另一台设备上的同一色块出现被同步的点击的效果，其实还挺好玩的呵呵。  开始       好了，我们现在对这个例子有了直观的认识了，让我们来开始一步步的分析它吧，对于代码里涉及到的知识点我们会进行不限于代码范围的讲述。       首先，让我们来从main.m文件开始吧。在other Source文件夹里，我们选中main.m来看一下它的代码。  #import int main(int argc, char \*argv[]){ NSAutoreleasePool \*pool = [NSAutoreleasePool new]; UIApplicationMain(argc, argv, nil, @"AppController"); [pool release]; return 0;}  　　第一行是导入UIKit框架，这个不用说了。        下面是main函数，和c程序一样，这个main函数也是我们的程序的入口。其实我们的程序的起点是start函数，在start函数里调用了main函数，然后在main函数里边，构建了一个自动释放池（这个witap的例子最新的版本是1.8，这个版本并没有针对ios5更新，所以这里的代码还是用的自动释放池，在ios5之后由于引入了ARC,所以ios5的main函数和之前的版本的main函数是有变化的，不再使用自动释放池了）。        在这个main函数里边最重要的一句代码就是：UIApplicationMain(argc, argv, nil, @"AppController")，这句是开始我们程序的关键，前两个参数就是main自己的参数，一个代表命令行参数的个数，一个是指向所有命令行参数的指针。第三个参数是代表我们的程序的主类，如果为nil则代表主类是UIApplication类，如果程序中使用自定义的UIApplication类的子类作为主类，你需要自己在这里指定，不过不推荐这样做！！。第四个参数是我们的代理类，如果为nil的话，则程序假设程序的代理来自Main nib文件(ios5之前，ios5改成用委托类的类名生成的字符串来指定了)。        那么UIApplicationMain这个函数又做了什么呢？在这个函数里边，我们根据我们的参数“主类名”，这里是UIApplication类，来实例化一个主类。然后对这个实例会设置他的委托为我们在第四个参数里指定的类，并调用\_run方法，\_run方法又会调用CFRunLoopRunInMode(??)，CFRunLoopRunInMode方法又会根据它的参数来以相应的模式运行RunLoop(RunLoop的概念很重要，我们后边会说明),这样注册到这个模式下的事件在发生时，我们相应的事件处理方法就会收到消息并处理。（这个需要结合下一段来理解）        RunLoop是一个运行回路，每个线程都有一个自己的RunLoop，我们平时并不用管理它是因为我们的主线程中的RunLoop默认情况下就启动了，UIApplication类帮我们做的。RunLoop做的具体的工作是监测输入源，如果有输入源事件的话，RunLoop分发这个事件给事件的处理方法来处理事件，如果没有输入源事件的话，RunLoop就让我们的线程休眠，什么都不做来节省资源消耗。那么什么是输入源呢？输入源包括：用户设备输入、网络连接、周期或延迟事件、异步回调。RunLoop能监测3中类型的对象：sources ([CFRunLoopSource Reference](http://developer.apple.com/library/ios/documentation/CoreFoundation/Reference/CFRunLoopSourceRef/Reference/reference.html#//apple_ref/doc/uid/20001443)), timers ([CFRunLoopTimer Reference](http://developer.apple.com/library/ios/documentation/CoreFoundation/Reference/CFRunLoopTimerRef/Reference/reference.html#//apple_ref/doc/uid/20001444)), and observers ([CFRunLoopObserver Reference](http://developer.apple.com/library/ios/documentation/CoreFoundation/Reference/CFRunLoopObserverRef/Reference/reference.html#//apple_ref/doc/uid/20001442))，要让RunLoop监测这些对象，需要先把他们加入到RunLoop中，针对这三种对象，有3个不同的方法用来加入到RunLoop中：[CFRunLoopAddSource](http://developer.apple.com/library/ios/documentation/CoreFoundation/Reference/CFRunLoopRef/Reference/reference.html#//apple_ref/c/func/CFRunLoopAddSource), [CFRunLoopAddTimer](http://developer.apple.com/library/ios/documentation/CoreFoundation/Reference/CFRunLoopRef/Reference/reference.html#//apple_ref/c/func/CFRunLoopAddTimer), [CFRunLoopAddObserver](http://developer.apple.com/library/ios/documentation/CoreFoundation/Reference/CFRunLoopRef/Reference/reference.html#//apple_ref/c/func/CFRunLoopAddObserver)，这样加入到RunLoop后,RunLoop才会监测这些输入源，如果不想继续监测这个输入源的话，可以用CFRunLoopRemoveSource方法从RunLoop中移出输入源。还有一点要强调的是，在把这些输入源加入到RunLoop时，我们必须要把这些输入源关联到一个或多个RunLoop模式，模式决定了在RunLoop的一次迭代中，什么事件需要处理，因为RunLoop在运行时就指定了以什么模式运行，所以，RunLoop只处理那些关联到它当前运行模式的输入源。通常情况下我们如果添加输入源到RunLoop的话，我们会把它和默认模式 [kCFRunLoopDefaultMode](http://developer.apple.com/library/ios/documentation/CoreFoundation/Reference/CFRunLoopRef/Reference/reference.html#//apple_ref/c/data/kCFRunLoopDefaultMode)相关联，在应用程序或线程闲置的时候就会处理这些事件。事实上你可以定义自己的模式，通过自己的模式你可以对需要处理的事件进行限制（具体请参考[官方文档](http://developer.apple.com/library/ios/#documentation/CoreFoundation/Reference/CFRunLoopRef/Reference/reference.html)）。       最后对于一个程序来说，简单地来总结一下，RunLoop做的工作是什么。       我们的程序有一个UIApplication的实例变量app,这个app实例变量为我们在我们的主线程里（以某种模式，我个人猜测会是默认模式吧）运行了一个RunLoop，并且它把一些输入源加入到了这个RunLoop中(比如触摸事件、网络连接??等等),并与当前运行的这个模式关联，这样有相应事件发生的时候，RunLoop就会监测到这些事件了，当RunLoop监测到这些事件后，就通过委托（前面设定了为AppController）层层分发，直到分发给我们的相应事件的处理方法进行处理，（比如触摸事件的处理方法是touchBegan、touchMove、touchEnd等），这样我们就能正确的处理这些事件了。这也正是我们能处理触摸事件的原因，UIApplication类，已经把相应的输入源加入到我们主线程的RunLoop中了。       下面我们来看看applicationDidFinishLaunching:方法，它是告诉我们UIApplication已经准备好了，可以运行了，是我们的程序可见代码部分，在main函数之后运行的第一个方法，打开AppController.m文件：  - (void) applicationDidFinishLaunching:(UIApplication \*)application{ CGRect rect; //1 UIView\* view; NSUInteger x, y; //Create a full-screen window //2 \_window = [[UIWindow alloc] initWithFrame:[[UIScreen mainScreen] bounds]]; [\_window setBackgroundColor:[UIColor darkGrayColor]]; //Create the tap views and add them to the view controller's view rect = [[UIScreen mainScreen] applicationFrame]; //3  for(y = 0; y < kNumPads; ++y) { //4 for(x = 0; x < kNumPads; ++x) { view = [[TapView alloc] initWithFrame:CGRectMake(rect.origin.x + x \* rect.size.width / (float)kNumPads, rect.origin.y + y \* rect.size.height / (float)kNumPads, rect.size.width / (float)kNumPads, rect.size.height / (float)kNumPads)]; //5 [view setMultipleTouchEnabled:NO]; //6 [view setBackgroundColor:[UIColor colorWithHue:((y \* kNumPads + x) / (float)(kNumPads \* kNumPads)) saturation:0.75 brightness:0.75 alpha:1.0]]; [view setTag:(y \* kNumPads + x + 1)]; //7 [\_window addSubview:view]; //8 [view release]; //9 } } //Show the window [\_window makeKeyAndVisible]; //10 //Create and advertise a new game and discover other availble games [self setup]; //11}        注释1，就是声明一些变量，一个rect，一个view,两个整数。        注释2，实例化一个UIWindow变量，这是我们的主窗口，并把它的大小区域设为全屏大小，[[UIScreen mainScreen] bounds]]得到的就是全屏的区域大小，然后设置它的                   颜色为灰色。        注释3，把我们前面申请的rect变量，设为整个屏幕除了状态栏的大小区域，[[UIScreen mainScreen] applicationFrame]得到的就是除了状态栏的屏幕大小区域。        注释4，这个for循环是添加游戏中的色块儿的（就是上面游戏运行图中的9个色块），在AppController.m文件的最上边，我们看到我们用宏定义了KNumPads为3，所以这里                  是外循环和内循环都是3次，共9次。        注释5，这是实例化我们的色块儿，并分配给我们前面申请的view变量。我们的色块是单独的类TapView的实例，它是继承自UIView的，我们先不管它的实现，就把它当一个                  UIView来对待就好了，后面我们会详细介绍它的内容。在实例化这些色块的时候我们通过简单的计算来给这9个色块划分不同的区域和位置，使这9个色块均匀的分布在                  我们的屏幕上，当然，是去除了状态栏之后的区域。        注释6，设置我们的色块的多点触摸为否，这样我们的色块是不会相应多点触摸了。        注释7，给我们的色块设置不同的tag,这里是编号1到9。        注释8，把我们的色块加入主窗口的子集，这样当我们的主窗口显示的时候，我们的色块作为子视图，也就会显示了。        注释9，以为我们把色块加入window的时候，他帮我们retain了，所以这里我们要release。        注释10，显示我们的window。        注释11，这是调用我们这个类的setup方法，这个方法后面会详述，现在先不管了。   我们接着来看看，这个setup方法是什么：  - (void) setup {  [\_server release]; //1 \_server = nil; [\_inStream removeFromRunLoop:[NSRunLoop currentRunLoop] forMode:NSDefaultRunLoopMode]; //2 [\_inStream release]; \_inStream = nil; \_inReady = NO; [\_outStream removeFromRunLoop:[NSRunLoop currentRunLoop] forMode:NSDefaultRunLoopMode]; //3 [\_outStream release]; \_outStream = nil; \_outReady = NO; \_server = [TCPServer new];//4 [\_server setDelegate:self];  NSError \*error = nil; if(\_server == nil || ![\_server start:&error]) { //5 if (error == nil) { NSLog(@"Failed creating server: Server instance is nil"); } else { NSLog(@"Failed creating server: %@", error); } [self \_showAlert:@"Failed creating server"]; //6 return; } //Start advertising to clients, passing nil for the name to tell Bonjour to pick use default name if(![\_server enableBonjourWithDomain:@"local" applicationProtocol:[TCPServer bonjourTypeFromIdentifier:kGameIdentifier] name:nil]) { //7 [self \_showAlert:@"Failed advertising server"]; return; } [self presentPicker:nil]; //8}  　　注释1、2和3，都是一些清理方法，用来确保当我们操作的这些变量之前用内容的话，先把它们清空，再重新进行操作。（当程序里第二次调用这个方法时，就显出了这几句的效果）                           ，这里先是对一个TCPServer类（后面会讲）实例进行置空操作；然后把一对输入输出流对象，从当前RunLoop中移出，这样我们的RunLoop就不再监测这两个输入                             输出流事件了，从RunLoop中移除后，也对它们进行置空操作；最后又把两个用来标示输入输出流是否准备好的Bool变量设为假，标示我们没有准备好。       标示4，重新初始化这个TCPServer类变量，并把它的委托设为这个AppController。（这里之所以能把TCPServer的委托delegate设为AppController是因为，这个delegate是要求符合TCPServerDelegate 协议的，而我们的AppController在interface部分已经声明了它是符合这个TCPServerDelegate 协议的）       注释5，判断这个TCPServer类实例变量\_server是否为空，并对它调用start:方法，并判断start:方法的返回值。如果有问题根据判断条件输出相应的错误信息。       注释6，当有错的时候弹出警告窗口来说明失败情况。       注释7，对这个\_server变量调用一个方法，这个方法是用来发布我们的服务的。（这个例子用Bonjour实现联机，而boujour实现是通过NSNetService来发布服务，用                 NSNetServiceBrowser来搜索服务来实现的，这也是一个重要的知识点，后面会讲）       注释8，这个是显示我们的这篇文章中第一个图中的界面的一个方法。       这个例子真是千头万绪呀，我希望一部分一部分的拆开来分析，可是它的每一部分总是和其他内容相关联，摘不出来呀，郁闷！好了，我们继续吧，如果要弄明白这个setup方法到底做了什么，我们必需得先分析这个TCPServer类才行，然后还要细分这个注释8中的方法才能真正了解这个setup方法做了哪些工作。  难啃的骨头       我们来见识一下这个TCPServer的真面目吧，打开TCPServer.h文件：  @class TCPServer;NSString \* const TCPServerErrorDomain;typedef enum { kTCPServerCouldNotBindToIPv4Address = 1, kTCPServerCouldNotBindToIPv6Address = 2, kTCPServerNoSocketsAvailable = 3,} TCPServerErrorCode;  　　在文件的最上面部分，我们看到，我们先用@class来修饰我们的TCPServer，这是告诉编译器，这个TCPServer是一个类，这样我们就可以在还没有声明这个类的时候在方法里先用，在以后在实现它的定义。（这样也就是为什么我们后面的协议方法中用到了TCPServer类，但这个类的声明却在协议之后，而我们在编译是不报错的原因）       我们又声明了一个字符串常量，它的定义是在TCPServer.m文件里的，在这里只是声明。       定义了TCPServer的错误代码的枚举值，用来表示不同的错误情况。       接着，我们定义了一个协议（协议是objective-c中，不同的类之间沟通的好方法，个人觉得和symbian里的M类基本上一样）：  @protocol TCPServerDelegate @optional- (void) serverDidEnableBonjour:(TCPServer\*)server withName:(NSString\*)name;- (void) server:(TCPServer\*)server didNotEnableBonjour:(NSDictionary \*)errorDict;- (void) didAcceptConnectionForServer:(TCPServer\*)server inputStream:(NSInputStream \*)istr outputStream:(NSOutputStream \*)ostr;@end  　　协议的名字叫：TCPServerDelegate，这个协议有3个可选的方法，这三个方法第一个是TCPServer用boujour发布服务成功之后我们用来处理一些东西的方法，第二个是失败的时候我们用来处理一些东西的方法，第三个是当TCPServer接受了其他设备的连接请求之后，我们用来处理东西的方法。       下面让我们看看这个TCPServer类的声明：  @interface TCPServer : NSObject {@private id \_delegate; 　　uint16\_t \_port; uint32\_t protocolFamily; CFSocketRef witap\_socket; NSNetService\* \_netService;} - (BOOL)start:(NSError \*\*)error;- (BOOL)stop;- (BOOL) enableBonjourWithDomain:(NSString\*)domain applicationProtocol:(NSString\*)protocol name:(NSString\*)name; - (void) disableBonjour;@property(assign) id delegate;+ (NSString\*) bonjourTypeFromIdentifier:(NSString\*)identifier;  　　这个TCPServer类，被声明继承自NSObject类，并且它遵守NSNetServiceDelegate协议，这个协议是我们的NSNetService类的一些回调方法，就是说如果我们的NSNetService服务发布成功或者失败的话，会调用这个协议里的相应方法来进行处理。事实上这个协议的所有方法都是可选的，如果你不实现他们也不会出错，不过那样的话，我们就不能在服务发布状态改变是做相应的处理了。       在这个interface里，声明了5个私有变量，一个id类的\_delegate,它用来跟踪我们这个TCPServer类的委托，一个uint16\_t类型的\_port变量，存储我们发布服务时绑定的Socket的端口号，一个uint32\_t类型的protocolFamily,来存储我们的socket的协议族，一个CFSocketRef类的 witap\_socket,就是我们的等待其他设备连接的socket，一个NSNetService类的\_netService,就是我们用来发布服务的NSNetService。       start:方法，我们在这个方法里创建并配置我们用来监听网络连接的socket,并创建RunLoop输入源，加入到当前RunLoop中，这样只要有我们的这个socket有连接事件，我们就能得到通知并触发相应的回调。       stop方法，明显不过了，它是停止我们的网络连接服务的，让我们取消对网络连接事件的监听，并释放这个监听的socket。       disableBonjour方法，停止我们的当前的已经发布的服务。       enableBonjourWithDomain:applicationProtocol:name：方法是事实上进行NSNetService服务发布的方法。       接着是一个声明，声明了一个id的属性delegate，这是一个满足TCPServerDelegate协议的属性。       最后是一个bonjourTypeFromIdentifier:方法，这是个辅助方法，它用来返回我们要发布的服务的协议的（这个协议不是委托类的协议，它只是一个代表唯一标识的字符串），并且这个字符串是用要求的，它不能超过14个字符，并且只能包含小写字母、数字和连接符，开关和结尾不能是连接符。        下面，我们该看看这个类的具体实现了,打开TCPServer.m文件：  #import "TCPServer.h"NSString \* const TCPServerErrorDomain = @"TCPServerErrorDomain";@interface TCPServer ()@property(nonatomic,retain) NSNetService\* netService;@property(assign) uint16\_t port;@end  　　首先是包含TCPServer.h文件，然后是我们在.h文件中声明的那个常量字符串的定义。        然后下面……（其实我不太明白苹果这个地方的用法，正常情况下我会觉得这是一个分类，但是如果是分类的话，是不可以添加实例变量的，这个地方是添加了两个属性，这个什么情况？？就算可以在分类里添加属性，那为什么要这么做呢？为什么不在正常的原始类里添加呢？是不是因为这样的话这个属性是在TCPServer.m文件里的，那么在这个分类里声明的属性就对外不可见不可用了呢？那又为什么分类没有自己的implementation呢？它怎么和原始类共用一个呢？如果你知道这个原因，请告诉我，不胜感激！！！）        再接着是这个TCPServer类的实现部分：  @implementation TCPServer@synthesize delegate=\_delegate, netService=\_netService, port=\_port;- (id)init { return self;}- (void)dealloc { [self stop]; [super dealloc];}  　   这里合成了三个属性的set和get方法；然后是初始化方法，只是简单地返回自己；然后是dealloc方法，这会在我们这个类销毁时调用，这个方法里是先调用stop方法停掉网络连接服务，然后调用父类的dealloc方法。       再接着看这个handleNewConnectionFromAddress方法：  - (void)handleNewConnectionFromAddress:(NSData \*)addr inputStream:(NSInputStream \*)istr outputStream:(NSOutputStream \*)ostr { // if the delegate implements the delegate method, call it if (self.delegate && [self.delegate respondsToSelector:@selector(didAcceptConnectionForServer:inputStream:outputStream:)]) { [self.delegate didAcceptConnectionForServer:self inputStream:istr outputStream:ostr]; }}  　　在这个方法里，我们先判断self的委托是否为空（我们在AppController.m的setup方法里的注释4中，把委托设为了AppController），并判断这个委托响不响应didAcceptConnectionForServer:inputStream:outputStream:方法，如果响应，就对self的委托调用这个方法来处理一些事情。        现在轮到一个重量级的方法了，TCPServerAcceptCallBack：  static void TCPServerAcceptCallBack(CFSocketRef socket, CFSocketCallBackType type, CFDataRef address, const void \*data, void \*info) { TCPServer \*server = (TCPServer \*)info; NSLog(@"the server in call back is: %@",server.description); if (kCFSocketAcceptCallBack == type) { // for an AcceptCallBack, the data parameter is a pointer to a CFSocketNativeHandle CFSocketNativeHandle nativeSocketHandle = \*(CFSocketNativeHandle \*)data; uint8\_t name[SOCK\_MAXADDRLEN]; socklen\_t namelen = sizeof(name); NSData \*peer = nil; if (0 == getpeername(nativeSocketHandle, (struct sockaddr \*)name, &namelen)) { peer = [NSData dataWithBytes:name length:namelen]; } CFReadStreamRef readStream = NULL; CFWriteStreamRef writeStream = NULL; CFStreamCreatePairWithSocket(kCFAllocatorDefault, nativeSocketHandle, &readStream, &writeStream); if (readStream && writeStream) { CFReadStreamSetProperty(readStream, kCFStreamPropertyShouldCloseNativeSocket, kCFBooleanTrue); CFWriteStreamSetProperty(writeStream, kCFStreamPropertyShouldCloseNativeSocket, kCFBooleanTrue); [server handleNewConnectionFromAddress:peer inputStream:(NSInputStream \*)readStream outputStream:(NSOutputStream \*)writeStream]; } else { // on any failure, need to destroy the CFSocketNativeHandle // since we are not going to use it any more close(nativeSocketHandle); } if (readStream) CFRelease(readStream); if (writeStream) CFRelease(writeStream); }}  　　这是一个回调方法，在我们的监听网络连接的socket，接收到连接事件后，这个回调方法就被调用。需要说明的是，这个回调方法的格式是固定的。原因是这样的，这个回调方法是要在创建socket的时候传递给socketCreat方法的callout参数的，这个callout是一个函数指针，这个函数指针是CFSocketCallBack 类型的，所以我们的这个回调方法也应该是这个类型的。我们来看一下这个类型：  typedef void (\*CFSocketCallBack) ( CFSocketRef s, CFSocketCallBackType callbackType, CFDataRef address, const void \*data, void \*info);  　　这个类型就是我们的函数指针的定义，它的返回值是void的，也就是没有返回值;再来看看它的参数：        很明显，第一个参数是触发了这个回调的socket本身，第二个是触发这个回调的事件类型，第三个代表请求连接的远端设备的地址，第四个参数有点神奇，它根据回调事件的不同，它代表的东西也不同，如果这个是连接失败回调事件，那它就代表一个错误代码的指针，如果是连接成功的回调事件，它就是一个Socket指针，如果是数据回调事件，这就是包含这些数据的指针，其它情况下它是NULL的，最后一个参数是我们创建socket的时候用的那个CFSocketContext结构的info成员（这个CFSocketContext机构我们会在后面的start:方法里再讲）。       明白这个函数指针的类型再对照着我们的回调函数看是不是结构完全一样，呵呵。 |

**打起精神继续战斗**

     好吧，让我们接着第一部分继续剖析这个TCPServer.m文件吧，上一部分中我们是讲到了TCPServerAcceptCallBack：这个回调方法，讲到了它的格式，它是一个符合CFSocketCallBack这个函数指针类型的函数。

     现在让我们继续研究它的实现吧，为了方便阅读，我们再一次把这个函数的实现展示出来：

[复制代码](javascript:void(0);)

static void TCPServerAcceptCallBack(CFSocketRef socket, CFSocketCallBackType type, CFDataRef address, const void \*data, void \*info)  
 {

TCPServer \*server = (TCPServer \*)info; //1

if (kCFSocketAcceptCallBack == type) { //2

// for an AcceptCallBack, the data parameter is a pointer to a CFSocketNativeHandle

CFSocketNativeHandle nativeSocketHandle = \*(CFSocketNativeHandle \*)data; //3

uint8\_t name[SOCK\_MAXADDRLEN]; //4

socklen\_t namelen = sizeof(name);

NSData \*peer = nil; //5

if (0 == getpeername(nativeSocketHandle, (struct sockaddr \*)name, &namelen)) {

peer = [NSData dataWithBytes:name length:namelen];

}

CFReadStreamRef readStream = NULL; //6

CFWriteStreamRef writeStream = NULL;

CFStreamCreatePairWithSocket(kCFAllocatorDefault, nativeSocketHandle, &readStream, &writeStream);

if (readStream && writeStream) {

CFReadStreamSetProperty(readStream, kCFStreamPropertyShouldCloseNativeSocket, kCFBooleanTrue); //7

CFWriteStreamSetProperty(writeStream, kCFStreamPropertyShouldCloseNativeSocket, kCFBooleanTrue);

[server handleNewConnectionFromAddress:peer inputStream:(NSInputStream \*)readStream outputStream:(NSOutputStream \*)writeStream];

} else {

// on any failure, need to destroy the CFSocketNativeHandle

// since we are not going to use it any more

close(nativeSocketHandle);

}

if (readStream) CFRelease(readStream); //8

if (writeStream) CFRelease(writeStream);

}

}

[复制代码](javascript:void(0);)

　　注释1，我们看到这里它把这个函数的参数info转成了一个TCPServer类型，并申请了一个TCPServer类的变量来跟踪它。（我们在第一部分里已经说了，这里的info参数就是触发这个回调的socket再被创建时，传入到创建函数里的CFSocketContext结构的info成员，在后面的start方法里我们会看到，这个结构的info成员是我们的这个TCPServer类本身）

     注释2，这里是判断一下我们这次回调的事件类型，如果事件是成功连接我们就进行一系列操作，否则我们什么也不做。

     注释3，这里我们是把这个函数的参数data转成了一个CFSocketNativeHandle类型。（同样的也是我们在第一部分里说过的，当这个回调事件的类型是连接成功的时候，这个data就是一个CFSocketNativeHandle类型指针，这个CFSocketNativeHandle类型其实就是我们的特定平台的socket,你就当成正常的socket理解就行了，值得注意的是这里的socket是什么，是呢儿来的？我们知道，在正常的socket流程中，作为服务器的一方会有一个socket一直处于监听连接的状态，一旦有新的连接请求到来，系统会自己创建一个新的socket与这个请求的客户端进行连接，此后客户端和服务器端就通过这个新的连接进行通讯，而服务器负责监听网络连接的socket则继续监听连接。现在这个函数里的这个data应该就是在响应连接请求的时候系统自己创建的新的socket吧。<**声明:这些概念我是了解自互联网，如果有什么不对的地方请及时指出，我会及时纠正，以免误导他人**>）

     注释4，申请了一个255大小的数组用来接收这个新的data转成的socket的地址，还申请了一个socklen\_t变量来接收这个地址结构的大小。

     注释5，这里又申请了一个NSData类的变量peer,在这个注释的if语句的{}里，我们看到它是用来存储我们的新的socket的地址的。我们先来看这个if语句的判断表达式吧，其实这里是一个getpeername()函数的调用，这个函数有3个参数，第一个参数是一个已经连接的socket,这里就是nativeSocketHandle;第二个参数是用来接收地址结构的，就是说这个函数从第一个参数的socket中获取与它捆绑的端口号和地址等信息，并把它存放在这第二个参数中；第三个参数差不多了，它是取出这个socket的地址结构的数据长度放到这个参数里面。如果没有错误的话这个函数会返回0，如果有错误的话会返回一个错误代码。这里判断了getpeername的返回值，没有错误的情况下，把得到的地址结构存储到我们申请的peer里。

     注释6，申请了一对输入输出流，用CFStreamCreatePairWithSocket()方法把我们申请的这一对输入输出流和我们的已建立连接的socket（即现在的nativeSocketHandle）进行绑定，这样我们的这个连接就可以通过这一对流进行输入输出的操作了，这个函数操作完成之后，这两个输入输出流会被重新指向，使其指向有效的地址区域。此函数的的一个参数是一个内存分配器（苹果管理优化内存的一种措施，更多信息可网上查询），第二个参数就是想用我们第三和第四个参数代表的输入输出流的socket，第三和第四个参数就是要绑定到第二个参数表示的socket的输入输出流的地址。

     注释7，如果我们的CFStreamCreatePairWithSocket()方法操作成功的话，那么我们现在的readStream和writeStream应该指向有效的地址，而不是我们在刚申请时赋给的NULL了。此时，判断这两个流是不是NULL就等于说我们判断函数CFStreamCreatePairWithSocket()有没有操作成功。

      如果成功的话，我们就设置这两个流的属性，这里是把这两个流的属性kCFStreamPropertyShouldCloseNativeSocket设置为真，默认情况下这个属性是假的，这个设为真就是说，如果我们的流释放的话，我们这个流绑定的socket也要释放。（这里是对两个流都进行了相同的属性设置，事实上苹果的文档里在对CFStreamCreaterPairWithSocket()这个方法说明的时候提到，大多数流的属性是共享的，你只要对这一对中的一个设置了属性，那么也会自动为另一个流设置这个属性）。然后我们对注释1中得到的TCPServer类的server变量调用handleNewConnectionFromAddress：方法，这个方法的三个参数，一个是已连接socket的地址，另两个就是输入和输出流了。这个方法的内容很简单，在本系列的第一部分我们已经介绍了，这里就不重复了。

      如果失败的话，我们就销毁着了已经连接的socket。

      注释8，这里是先对流的内容进行清空操作，防止在使用它们的时候，里面有我们不需要的垃圾数据。

      真是出了一头汗呀，这个函数太费劲了讲起来⋯⋯好在是说完了，不过，呵呵，下一个函数依然是艰巨呀。

**艰苦继续，你敢坚持吗？**

     哈哈，又是一个巨大的函数呀，恐怖的start方法，不要害怕，我们会把它分析的支离破碎的哈哈哈，好吧，先看看它的实现：

[复制代码](javascript:void(0);)

- (BOOL)start:(NSError \*\*)error {

CFSocketContext socketCtxt = {0, self, NULL, NULL, NULL}; //1

// Start by trying to do everything with IPv6. This will work for both IPv4 and IPv6 clients

// via the miracle of mapped IPv4 addresses.

witap\_socket = CFSocketCreate(kCFAllocatorDefault, PF\_INET6, SOCK\_STREAM, IPPROTO\_TCP, kCFSocketAcceptCallBack, (CFSocketCallBack)&TCPServerAcceptCallBack, &socketCtxt); //2

if (witap\_socket != NULL) // the socket was created successfully //3

{

protocolFamily = PF\_INET6; //4

} else // there was an error creating the IPv6 socket - could be running under iOS 3.x //5

{

witap\_socket = CFSocketCreate(kCFAllocatorDefault, PF\_INET, SOCK\_STREAM, IPPROTO\_TCP, kCFSocketAcceptCallBack, (CFSocketCallBack)&TCPServerAcceptCallBack, &socketCtxt);

if (witap\_socket != NULL)

{

protocolFamily = PF\_INET;

}

}

if (NULL == witap\_socket) { //6

if (error) \*error = [[NSError alloc] initWithDomain:TCPServerErrorDomain code:kTCPServerNoSocketsAvailable userInfo:nil];

if (witap\_socket) CFRelease(witap\_socket);

witap\_socket = NULL;

return NO;

}

int yes = 1; //7

setsockopt(CFSocketGetNative(witap\_socket), SOL\_SOCKET, SO\_REUSEADDR, (void \*)&yes, sizeof(yes));

// set up the IP endpoint; use port 0, so the kernel will choose an arbitrary port for us, which will be advertised using Bonjour

if (protocolFamily == PF\_INET6) //8

{

struct sockaddr\_in6 addr6;

memset(&addr6, 0, sizeof(addr6));

addr6.sin6\_len = sizeof(addr6);

addr6.sin6\_family = AF\_INET6;

addr6.sin6\_port = 0;

addr6.sin6\_flowinfo = 0;

addr6.sin6\_addr = in6addr\_any;

NSData \*address6 = [NSData dataWithBytes:&addr6 length:sizeof(addr6)];

if (kCFSocketSuccess != CFSocketSetAddress(witap\_socket, (CFDataRef)address6)) {

if (error) \*error = [[NSError alloc] initWithDomain:TCPServerErrorDomain code:kTCPServerCouldNotBindToIPv6Address userInfo:nil];

if (witap\_socket) CFRelease(witap\_socket);

witap\_socket = NULL;

return NO;

}

// now that the binding was successful, we get the port number

// -- we will need it for the NSNetService

NSData \*addr = [(NSData \*)CFSocketCopyAddress(witap\_socket) autorelease];

memcpy(&addr6, [addr bytes], [addr length]);

self.port = ntohs(addr6.sin6\_port);

} else { //9

struct sockaddr\_in addr4;

memset(&addr4, 0, sizeof(addr4));

addr4.sin\_len = sizeof(addr4);

addr4.sin\_family = AF\_INET;

addr4.sin\_port = 0;

addr4.sin\_addr.s\_addr = htonl(INADDR\_ANY);

NSData \*address4 = [NSData dataWithBytes:&addr4 length:sizeof(addr4)];

if (kCFSocketSuccess != CFSocketSetAddress(witap\_socket, (CFDataRef)address4)) {

if (error) \*error = [[NSError alloc] initWithDomain:TCPServerErrorDomain code:kTCPServerCouldNotBindToIPv4Address userInfo:nil];

if (witap\_socket) CFRelease(witap\_socket);

witap\_socket = NULL;

return NO;

}

// now that the binding was successful, we get the port number

// -- we will need it for the NSNetService

NSData \*addr = [(NSData \*)CFSocketCopyAddress(witap\_socket) autorelease];

memcpy(&addr4, [addr bytes], [addr length]);

self.port = ntohs(addr4.sin\_port);

}

// set up the run loop sources for the sockets

CFRunLoopRef cfrl = CFRunLoopGetCurrent(); //10

CFRunLoopSourceRef source = CFSocketCreateRunLoopSource(kCFAllocatorDefault, witap\_socket, 0);

CFRunLoopAddSource(cfrl, source, kCFRunLoopCommonModes);

CFRelease(source);

return YES;

}

[复制代码](javascript:void(0);)

　　 先看一下这个函数的定义，他返回BOOL类型的值以表示期望的操作是否成功；他还有一个NSError\*\*类型的参数，我们看定义会发现，其实NSError是一个继承者NSObject的类，NSError\*\*是一个指针的指针，就是说error这个参数是一个指针的指针，那么\*error是一个指针，\*error指向一个NSError对象。

      注释1，我们定义了一个CFSocketContext结构类型变量socketCtxt，并对这个结构进行初始化。我们来解释一下这个结构的定义，这个结构有5个成员。第一个成员是这个结构的版本号，这个必需是0；第二个成员可以是一个你程序内定义的任何数据的指针，这里我们传入的是self，就是这们这个类本身了，所以我们的TCPServerAcceptCallBack这个回调方法可以把它的info参数转成TCPServer，并且这个参数会被传入在这个结构内定义的所有回调函数；第三、四、五这三个成员其实就是3个回调函数的指针，一般我们都设为NULL,就是不用它们。（如果你要具体了解这个结构，请参考[官方文档](http://developer.apple.com/library/ios/#documentation/CoreFoundation/Reference/CFSocketRef/Reference/reference.html)）

      注释2，这是创建一个socket并把它赋给我们在TCPServer.h文件里定义的witap\_socket变量，这个CFSocketCreate（）方法有7个参数之多，我们来一个一个解释吧,你如果想了解的更清楚我建议你还是查阅[官方文档](http://developer.apple.com/library/ios/#documentation/CoreFoundation/Reference/CFSocketRef/Reference/reference.html)。第一个是一个内存分配器（我们前边已经提到过了）;第二个是我们要创建的socket的协议族（更多这方面的知识你可能需要参考TCP/IP协议），这里我们是传入PF\_INET6(表明我们希望用IPv6协议)；第三个参数是socket的类型，我们这里是传入SOCK\_STREAM（表明我们是要数据流服务的，还有一种选择是数据报服务，这两种是基于不同的协议的，数据流是基于TCP/IP协议的，数据报是基于UDP协议的）;第四个参数是我们要创建的socket所用的具体的协议，这里我们传入IPPROTO\_TCP 表明我们是遵守TCP/IP协议的;第五个是回调事件的类型，就是说当这个类型的事件发生时我们的回调函数会被调用，我们这里传入kCFSocketAcceptCallBack表明当连接成功里我们的回调会被触发。（这里可以设置不只一个回调事件类型，多个不同的事件类型用"|"(位或运算符)连起来就可以了） ;第六个就是我们的回调函数的地址，当我们指定的回调事件出现时就调用这个回调函数，我们传入我们的TCPServerAcceptCallBack（）回调函数的地址;第七个是一个结构指针，这个结构就是CFSocketContext类型，它保存socket的上下文信息，我们这里传入我们在注释1中定义的socketCtxt的地址。（这个CFSocketCreate（）函数会拷贝一份这个结构的数据，所以在出了这个create函数之后，这个结构可以被置为NULL。）

      注释3，我们通过判断这个witap\_socket是否为空来判断我们刚才执行的socket创建工作有没有成功。

      注释4，如果我们刚才的创建工作成功了，我们就把我们在TCPServer.h文件中定义的protocolFamily设为PF\_INET6。

      注释5，如果刚才的创建失败了，我们再进行一次socket的创建工作，这次和刚才不同的是，这次我们把CFSocketCreate函数的协议族参数设为PF\_INET,这表示这次是使用IPv4协议。同样的，在创建操作完成之后判断witap\_socket的状态，如果创建成功了就把protocolFamily设为PF\_INET，如果又失败了就什么也不做。

      注释6，如果两次创建操作都失败了，如果我们的参数error这个指针的指针是用效的，我们就初始化一个NSError,这个NSError的内容就是我们在TCPServer.h里定义好的错误代码，和在TCPServer.m里定义的常量字符串TCPServerErrorDomain，然后我们把这个NSError对象赋给\*error，\*error才是一个指向NSError类型的指针；下面一句是如果这个witap\_socket不为空，就把这个witap\_socket释放了。（我个人认为这句是多余的，这句代码本身就包含在witap\_socket是空的if语句里了，还有必要再判断一次吗？），然后按苹果的逻辑，把这个witap\_socket释放，之后把它设为NULL，防止野指针。然后返回一个NO，这是告诉我们的这个函数的调用都，我们创建socket失败了。

      注释7，首先是定义了一个int变量yes,并初始化它的值是1。然后调用setsockopt（）方法来设置socket的选项，这个方法的第一个参数要求是一个socket的描述符，这里是通过CFSocketGetNative()方法来得到我们的socket对象针对于这个ios平台的描述符；第二个是需要设置的选项定义的层次，这里是SOL\_SOCKET（这方面的东西我并不了解，如果你想更多了解，可以参考[这篇文章](http://blog.sina.com.cn/s/blog_50cc0ffd0100awfg.html)）;第三个参数是我们要设置的选项的名字，这里是SO\_REUSEADDR。（表示允许重用本地地址和端口，就是说充许绑定已被使用的地址（或端口号），缺省条件下，一个套接口不能与一个已在使用中的本地地址捆绑。但有时会需要“重用”地址。因为每一个连接都由本地地址和远端地址的组合唯一确定，所以只要远端地址不同，两个套接口与一个地址捆绑并无大碍。）;第四个参数是一个指针，指向要设置的选项的选项值的缓冲区，这里是传入上面申请的int变量yes的地址，就是说我们把这个选项设为1;第五个参数是这个选项值数据缓冲区的大小，这里用sizeof得友yes的数据长度并传了进去。

      注释8，如果我们的protocolFamily是PF\_INET6的话，我们对这个socket进行相应的配置。先是申请一个sockaddr\_in6的结构变量addr6，这个结构是一个IPv6协议的地址结构，我们来看一下它的成员定义:

[复制代码](javascript:void(0);)

struct sockaddr\_in6 {

u\_char sin6\_len;

u\_char sin6\_family;

u\_int16m\_t sin6\_port;

u\_int32m\_t sin6\_flowinfo;

struct in6\_addr sin6\_addr;

}

[复制代码](javascript:void(0);)

　　我们来看，很明显这个sin6\_len是这个结构的大小;sin6\_family成员是指的协议;sin6\_port指的是端口;sin6\_flowinfo这个在微软MSDN上的说明只有一句话，IPv6的流信息;sin6\_addr这是一个IN6\_ADDR的结构，这个结构是真正存储我们的地址的。

      (对于这个sin6\_flowinfo，我并不懂这些协议上的东西，所以我查阅资料上的解释是，sin6\_flowinfo是与IPv6新增流标和流量类字段类相对应的一个选项，我们在编程时通常设为0。**引自：**[***此文***](http://wenku.baidu.com/view/6beef84f852458fb770b563e.html))

      现在让我们来看看这里就做了什么吧，我们先用memset方法把这个刚申请的结构清零;然后把结构的大小赋给了结构成员sin6\_len;把结构的协议族设为AF\_INET6;这里把这个结构里的端口号设为0，那么在socket进行绑定操作的时候，系统会为我们分配一个任意可用的端口;这个sin6\_flowinfo置为0就不说了;把这个地址结构的sin6\_addr成员设置为in6addr\_any，这里更清晰的解释还是请你查阅资料，这里可以简单理解为填上这个值系统会自动为我们填上一个可用的本地地址（这个一个可用的本地地址是说，有的机器可能会用多个网卡，会有多个地址）;然后申请一个NSData变量address6把我们的地址结构addr6的信息进行拷贝存储。

     然后下面，调用CFSocketSetAddress方法把我们的witap\_socket和上面刚设置好的地址address6进行绑定，这其实就是BSDSocket里的bind一样的。然后这里判断了绑定操作的执行结果，如果绑定失败的话，就进行相应的清理工作并返回失败（这些具体代码我们之前已经说过了），如果成功的话，就从这个绑定好的socket里拷贝出实际的地址并存储在addr6里，并把在TCPServer.h里定义的属性port设为系统为这个socket分配的实际的端口号，在后面发布NSNetService的时候需要用这个端口号。

     注释9，这个基本上和注释8是一样的，略微不同的是这个是基于IPv4的操作，小小的不同相信只要一看就明白了，我就不再重复说明了吧呵呵。

     注释10，这里申请了一个RunLoop的变量cfrl用来跟踪当前的RunLoop，通过CFRunLoopGetCurrent()方法得到当前线程正在运行的RunLoop，然后把它赋给cfrl;然后创建了一个RunLoop的输入源变量source,这里通过CFSocketCreateRunLoopSource()方法，这个方法的第一个参数是内存分配器，第二个就是我们想要做为输入源来监听的socket对象，第三个参数是代表在RunLoop中处理这些输入源事件时的优先级，数小的话优先级高。然后把这个创建的RunLoop输入源赋给变量source，接着把这个输入源source加入到当前RunLoop来进行监测，把输入源加入到RunLoop是通过CFRunLoopAddSource（）这个方法实现的，这个方法的第一个参数就是我们希望加入的RunLoop，第二个参数是要加入到第一个参数里的输入源，这里是source，最后一个参数就是这个我们要加入的输入源要关联的模式，这里是传入的kCFRunLoopCommonModes。（这里的kCFRunLoopCommonModes需要说明，这个kCFRunLoopCommonModes它并不是一个模式，苹果称它为伪模式，它其实是几个模式的合集，[kCFRunLoopDefaultMode](http://developer.apple.com/library/ios/documentation/CoreFoundation/Reference/CFRunLoopRef/Reference/reference.html#//apple_ref/c/data/kCFRunLoopDefaultMode)必定是这个KCFRunLoopCommonModes的一个子集。你可以自己加入一些其它的模式到这个KCFRunLoopCommonModes里，这个通俗点解释怎么说呢，比如说这个KCFRunLoopCommonModes里有两个子集，即有两个模式，我们假设是模式1和模式2，那么当我们把输入源关联到模式的时候传入KCFRunLoopCommonModes的话，这个输入源就会和这两个模式，模式1和模式2，都进行关联，这样不管我们的RunLoop是以模式1运行的还是以模式2运行的，它都会监测我们的这个输入源）;加入了输入源之后RunLoop就自动保持了这个输入源，我们现在就可以释放这个输入源了。最后返回操作成功。

**最黑暗的已经过去，准备迎接光明**

       终于这个巨无霸方法start告一段落了，剩下的都没有这么长的了，可以稍微放松一下了，来看看简单的stop方法：

[复制代码](javascript:void(0);)

- (BOOL)stop {

　　[self disableBonjour]; //1

if (witap\_socket) { //2

CFSocketInvalidate(witap\_socket);

CFRelease(witap\_socket);

witap\_socket = NULL;

}

return YES;

}

[复制代码](javascript:void(0);)

　　看过了start的庞大，再看看这个stop是不是有点小清新的感觉哈哈哈哈。首先看看它是一个返回BOOL型变量的方法，不过它在任何情况下都返回YES，其实这个返回值就没什么意义了。

     注释1，这是调用它的disableBonjour方法来停止它的NSNetService服务的。（后面我们会讲这个方法）

     注释2，判断这个用来监听网络连接的socket是否有效，如果为真，就先把这个socket设为无效，再释放这个sockt资源，并把它置为NULL。

     让我们看一个stop方法里调用的disableBonjour方法又做了什么：

[复制代码](javascript:void(0);)

- (void) disableBonjour

{

if (self.netService) {

[self.netService stop];

[self.netService removeFromRunLoop:[NSRunLoop currentRunLoop] forMode:NSRunLoopCommonModes];

self.netService = nil;

}

}

[复制代码](javascript:void(0);)

　　它也是很简单的，先判断这个发布的netService服务是不是有效，如果是就先停止这个服务，然后把它从RunLoop里移除使其不再被监听（后面会看到NSNetService也是需要加入RunLoop来进行监测的），然后把这个netService置为NULL。

      下面我们再来看一个比较关键的方法：enableBonjourWithDomain方法：

[复制代码](javascript:void(0);)

- (BOOL) enableBonjourWithDomain:(NSString\*)domain applicationProtocol:(NSString\*)protocol name:(NSString\*)name

{

if(![domain length]) //1

domain = @""; //Will use default Bonjour registration doamins, typically just ".local"

if(![name length])

name = @""; //Will use default Bonjour name, e.g. the name assigned to the device in iTunes

if(!protocol || ![protocol length] || witap\_socket == NULL) //2

return NO;

self.netService = [[NSNetService alloc] initWithDomain:domain type:protocol name:name port:self.port]; //3

if(self.netService == nil)

return NO;

[self.netService scheduleInRunLoop:[NSRunLoop currentRunLoop] forMode:NSRunLoopCommonModes]; //4

[self.netService publish];

[self.netService setDelegate:self];

return YES;

}

[复制代码](javascript:void(0);)

　　这个方法也用一个BOOL的返回值表示操作的成功或失败。并且这个方法有三个参数，它们都是我们用来发布NSNetService服务时要用的参数。第一个参数是发布服务用的域，第二个参数是我们要发布的网络服务的类型信息，第三个参数是用来表示我们这个服务的名字。

     注释1，如果我们的参数domain和name如果字符长度是0的话，就把它们设为@“”。（等下会解释这个@“”在这里的意义）

     注释2，如果参数protocol，即服务的协议，如果它不存在，或者它的字符长度为0，又或者witap\_socket为NULL(也就是说我们这个用来监听的socket无效的话),直接返回失败。

     注释3，这里是初始化一个NSNetService服务，并把它赋给在TCPServer.h里定义的netService属性。这个NSNetService的初始化方法用到了4个参数，前3个就是我们现在正在解释的这个方法的3个参数，最后一个是我们之前得到的端口号。第一个参数domain，它代表我们发布服务用的域，本地域是用@"local."来表示的，但是当我们想用本地域的时候，是不用直接传这个@"local."字符串进去的，我们只要传@""进去就行了，系统会自己把它按成本地域来使用的；第二个参数这个网络服务的类型，这个类型必需包含服务类型和传输层信息（传输层概念请参考TCP/IP协议），这个服务类型的名字和传输层的名字都要有“\_”字符作为前缀。比如这个例子的服务类型的完整的名字其实是@"**\_witap.\_tcp.**"，看到了吧，它们都有前缀"\_"，这里还有一点是要强调的，在这字符串结尾的"."符号是必需的，它表示这个域名字是绝对的；第三个参数是这个服务的名字，这个名字必需是唯一的，如果这个名字是@""的话，系统会自动把设备的名字作为这个服务的名字；第四个参数就是端口号了，这是我们发布这个服务用的。这个端口号必须是在应用程序里为这个服务获得的，这里就是witap\_socket在绑定时我们获得的那个端口号，获得之后赋给了port属性，所以这里传入的是self.port。初始化之后，把这个初始化过的NSNetService赋给.h文件里定义的netService属性，接着通过判断这个netService属性是否有效来判断这个NSNetService的初始化是否成功。如果初始化失败的话，直接返回操作失败。

      注释4，对这个netService属性调用scheduleInRunLoop:forMode:方法，从名字也能看得出来，这是把这个netService加入到当前RunLoop中，并关联到相应的模式。这部分内容在前面的start方法的注释10中已经有了详细的讲述，这里就不啰嗦了。然后对这个netService调用publish方法，这个方法是真正的发布我们的服务的。接着又设置这个netService的委托为这个TCPService类本身，netService是一个NSNetService类，这个类的委托是一个符合NSNetServiceDelegate的通用类型，在TCPService类的interface声明部分，我们看到这个类是声明自己符合NSNetServiceDelegate这个协议的，所以这里可以把netService的委托设为这个TCPService类本身。最后，返回操作成功。

**柳暗花明**

      到了这一步，这个类剩下的内容就很明了了，让我们看看NSNetServiceDelegate协议的两个方法吧：

[复制代码](javascript:void(0);)

- (void)netServiceDidPublish:(NSNetService \*)sender

{

if (self.delegate && [self.delegate respondsToSelector:@selector(serverDidEnableBonjour:withName:)])

[self.delegate serverDidEnableBonjour:self withName:sender.name];

}

- (void)netService:(NSNetService \*)sender didNotPublish:(NSDictionary \*)errorDict

{

if (self.delegate && [self.delegate respondsToSelector:@selector(server:didNotEnableBonjour:)])

[self.delegate server:self didNotEnableBonjour:errorDict];

}

[复制代码](javascript:void(0);)

　　这是NSNetServiceDelegate协议的其中两个方法，一个会在NSNetService发布成功时被调用，一个会在发布失败时会调用。它们都有一个参数，这个参数是触发这个回调的NSNetService本身。两个方法的内容都很简单。

      发布成功方法中先判断self的委托是否有效，如果有效的话它接着判断这个委托是不是响应serverDidEnableBonjour:withName方法，如果响应的话，就对它的委托调用这个方法。在[这个系列的第一部分](http://www.cnblogs.com/dingwenjie/archive/2012/04/10/2438687.html)的setup方法的注释4中，我们已经解释过了这里的self的委托被设置为AppController了，所以这里是判断AppController是不是响应这个serverDidEnableBonjour:withName方法，如果响应就对它调用这个方法。（关于这个被调用的方法我们在以后讲回到AppController时再讲）

     发布失败方法和这个成功方法基本上一样，不同的只是对应的方法，也不再重复说明了。

     最后是两个辅助性的方法：

[复制代码](javascript:void(0);)

- (NSString\*) description

{

return [NSString stringWithFormat:@"<%@ = 0x%08X | port %d | netService = %@>", [self class], (long)self, self.port, self.netService];

}

+ (NSString\*) bonjourTypeFromIdentifier:(NSString\*)identifier

{

if (![identifier length])

return nil;

return [NSString stringWithFormat:@"\_%@.\_tcp.", identifier];

}

[复制代码](javascript:void(0);)

 　　description方法是帮助我们调试程序时方便用的，它把这个类的信息返回给我们，方便我们输出到控制台进行查看。它把这个TCPService类对象的类名，地址，port属性，netService属性都返回了，非常方便。

      bonjourTypeFromIdentifier：这个方法真是一个纯粹的辅助方法，它在这个例子中的用途就是通过传入在AppController.m中定义的宏kGameIdentifier，然后返回一个完整的事实上的NSNetService的初始化方法中用的网络服务的类型。

 在[此系列的第1部分](http://www.cnblogs.com/dingwenjie/archive/2012/04/10/2438687.html)中，我们讲到要完全弄明白AppController类的setup方法的话，必须先弄清楚这个TCPService类是怎么回事，在[此系列第2部分中](http://www.cnblogs.com/dingwenjie/archive/2012/04/12/2439991.html)我们已经完全地分析过这个TCPService类是什么样的了，现在再让我们回过头来看看这个AppController类的setup方法，是不是除了最后一句调用的presentPicker以外，前面所有的部分我们都能很好地明白它到底干了什么了吧。

     在我们讨论这个presentPicker方法之前，我们先把第二部分中的遗留问题解决一下吧，还记得吗？我们在第2部分里讲到，我们实现的NSNetServiceDelegate协议的两个方法里，都是调用了TCPService的委托的协议方法，这个委托就是AppController，好的，那么我们现在来看看AppController实现的TCPServerDelegate协议的方法。

     第一个方法serverDidEnableBonjour：withName:方法，它是在NSNetService发布成功的回调里调用的：

- (void) serverDidEnableBonjour:(TCPServer \*)server withName:(NSString \*)string{ [self presentPicker:string];}

     看到了吧，这个方法的实现只有一个方法调用，它也是调用presentPicker方法。先不说这个presentPicker方法，这个等下再说。我们先找找NSNetService发布失败的回调里调用的方法server:didNotEnableBonjour:，不用怀疑，你找不到它的，苹果并没有实现它（因为它们都是可选的，所以可以不实现），那也就是说在TCPService类里的NSNetService发布失败的回调方法里，这个方法是不响应的，所以在NSNetService发布失败的回调方法里的判断条件为假，这个回调没有执行后面对委托的方法的调用。

## 黎明前的最后黑暗

     就像我们说要理解AppController类的setup方法需要先了解TCPService类一样，这里要理解presentPicker方法也有一个必需先了解的类，这就是BrowserViewController类，这个类简单点来说是搜索在当前网络内搜索特定服务（我们在TCPService类里enableBonjourWithDomain方法发布的服务）的。

    同样地我们先来看它的头文件内容，打开BrowserViewController.h文件：

#import //1#import @class BrowserViewController; //2@protocol BrowserViewControllerDelegate //3@required// This method will be invoked when the user selects one of the service instances from the list.// The ref parameter will be the selected (already resolved) instance or nil if the user taps the 'Cancel' button (if shown).- (void) browserViewController:(BrowserViewController \*)bvc didResolveInstance:(NSNetService \*)ref;@end

     注释1，导入需要的框架。

     注释2，声明这个BrowserViewController类。

     注释3，这里是定义了一个BrowserViewControllerDelegate协议，这个协议里有一个方法browserViewController:didResolveInstance:，而这个方法是被@required修饰的，表明这个方法是必需的，也就是说如果一个类声明要遵守这个BrowserViewControllerDelegate协议，它就必需要实现这个方法。我们之前看到的协议方法都是@optional修饰的，被@optional修饰的协议方法是可选的，可以不实现。这个方法是干什么的呢？这个方法是在当用户从我们搜索到的服务列表里选择的服务完成解析的时候调用的。（后面会详述）

     下面看这个类的具体定义：

1 @interface BrowserViewController : UITableViewController { 2 3 @private 4 id \_delegate; 5 NSString \*\_searchingForServicesString; 6 NSString \*\_ownName; 7 NSNetService \*\_ownEntry; 8 BOOL \_showDisclosureIndicators; 9 NSMutableArray \*\_services;10 NSNetServiceBrowser \*\_netServiceBrowser;11 NSNetService \*\_currentResolve;12 NSTimer \*\_timer;13 BOOL \_needsActivityIndicator;14 BOOL \_initialWaitOver;15 }16 17 @property (nonatomic, assign) id delegate;18 @property (nonatomic, copy) NSString \*searchingForServicesString;19 @property (nonatomic, copy) NSString \*ownName;20 21 - (id)initWithTitle:(NSString \*)title showDisclosureIndicators:(BOOL)showDisclosureIndicators showCancelButton:(BOOL)showCancelButton;22 - (BOOL)searchForServicesOfType:(NSString \*)type inDomain:(NSString \*)domain;23 24 @end

    在第1行，我们看到这个类是继承自UITableViewController的，那么它会有一个表视图（在这个类里这个表视图就是用来列出搜索到的服务的列表的）。并且这个类声明自己符合NSNetServiceDelegate和NSNetServiceBrowserDelegate两个协议，NSNetServiceDelegate协议我们在TCPService类里已经说了，这个NSNetServiceBrowserDelegate协议就是NSNetServiceBrowser类的一些回调方法，是当NSNetServiceBrowser类搜索到服务时，或其它特定情况发生时，分别相对应的一些回调方法。

    第3-14行，声明了一堆私有变量\_delegate是一个符合BrowserViewControllerDelegate的委托，\_searchingForServicesString是我们要搜索的服务的完整类型名（这个例子里事实上就是TCPService类里的bonjourTypeFromIdentifier：返回的字符串），\_ownName是这个程序自己发布的服务的名字，\_ownEntry是这个我们程序自己发布的服务，这\_showDisclosureIndicators是决定是否显示这个tableView的cell的accessoryView的，\_services是程序搜索到的所有符合自己搜索条件的服务数组，\_netServiceBrowser是我们用来搜索发现服务的类，\_currentResolve这个是当前正在解析的服务，\_timer一个定时器在例子里作用是延迟调用方法，\_needsActivityIndicator当前活动指示器，\_initialWaitOver标识initialWaitOver:方法是否执行完毕。

    第17-19行，添加三个属性。

    第21-22行，定义两个方法，一个是初始化这个BroswerViewController类的，一个是在特定域中搜索特定服务的。

    现在轮到看下这个BroswerViewController类的实现了，打开BroswerViewController.m文件：

1 #import "BrowserViewController.h" 2 3 #define kProgressIndicatorSize 20.0 4 5 // A category on NSNetService that's used to sort NSNetService objects by their name. 6 @interface NSNetService (BrowserViewControllerAdditions) 7 - (NSComparisonResult) localizedCaseInsensitiveCompareByName:(NSNetService \*)aService; 8 @end 9 10 @implementation NSNetService (BrowserViewControllerAdditions)11 - (NSComparisonResult) localizedCaseInsensitiveCompareByName:(NSNetService \*)aService {12 return [[self name] localizedCaseInsensitiveCompare:[aService name]];13 }14 @end

     先包含头文件，然后定义一个宏，接着为NSNetService创建一个分类，为这个分类添加一个方法，这个方法是按照NSNetService服务的名字来给它们排序的。

     接着下面是：

1 @interface BrowserViewController() 2 @property (nonatomic, retain, readwrite) NSNetService \*ownEntry; 3 @property (nonatomic, assign, readwrite) BOOL showDisclosureIndicators; 4 @property (nonatomic, retain, readwrite) NSMutableArray \*services; 5 @property (nonatomic, retain, readwrite) NSNetServiceBrowser \*netServiceBrowser; 6 @property (nonatomic, retain, readwrite) NSNetService \*currentResolve; 7 @property (nonatomic, retain, readwrite) NSTimer \*timer; 8 @property (nonatomic, assign, readwrite) BOOL needsActivityIndicator; 9 @property (nonatomic, assign, readwrite) BOOL initialWaitOver;10 11 - (void)stopCurrentResolve;12 - (void)initialWaitOver:(NSTimer \*)timer;13 @end

     为在BrowserViewController.h文件中没有添加属性的剩余私有变量添加属性声明，并添加了两个方法定义，stopCurrentResolve方法是用来停止当前正在解析的NSNetService服务的发布或者解析动作的，initialWaitOver:方法就是用来延迟处理一些操作的方法。

    BrowserViewController的具体实现：

1 @implementation BrowserViewController 2 3 @synthesize delegate = \_delegate; 4 @synthesize ownEntry = \_ownEntry; 5 @synthesize showDisclosureIndicators = \_showDisclosureIndicators; 6 @synthesize currentResolve = \_currentResolve; 7 @synthesize netServiceBrowser = \_netServiceBrowser; 8 @synthesize services = \_services; 9 @synthesize needsActivityIndicator = \_needsActivityIndicator;10 @dynamic timer;11 @synthesize initialWaitOver = \_initialWaitOver;

     为这些相应的属性合成set和get方法，这里值得说一个的是，@synthesize这个是说如果你自己不实现set和get方法，系统会根据属性给你生成全适的set和get方法，如果你自己实现了set和get的话，系统就不帮你生成了;@dynamic是说这个set和get方法必须由程序员自己实现。

    看第一个方法，初始化这个BrowserViewController类的方法：

1 - (id)initWithTitle:(NSString \*)title showDisclosureIndicators:(BOOL)show showCancelButton:(BOOL)showCancelButton { 2 3 if ((self = [super initWithStyle:UITableViewStylePlain])) { 4 self.title = title; 5 \_services = [[NSMutableArray alloc] init]; 6 self.showDisclosureIndicators = show; 7 8 if (showCancelButton) { 9 // add Cancel button as the nav bar's custom right view10 UIBarButtonItem \*addButton = [[UIBarButtonItem alloc]11 initWithBarButtonSystemItem:UIBarButtonSystemItemCancel target:self action:@selector(cancelAction)];12 self.navigationItem.rightBarButtonItem = addButton;13 [addButton release];14 }15 16 // Make sure we have a chance to discover devices before showing the user that nothing was found (yet)17 [NSTimer scheduledTimerWithTimeInterval:1.0 target:self selector:@selector(initialWaitOver:) userInfo:nil repeats:NO];18 }19 20 return self;21 }

     前面大量的代码我们一般都是比较详细地一句一句地讲它的作用，由于前面的讲述已经把基本的正常该讲述的都讲过了，现在开始我们只讲逻辑以提高效率，当然如果有该着重讲解的知识点的话，我还是会细致地讲的。

    第3句是调用它的父类（UITableViewController）的初始化方法，选择一种风格为这个类进行初始化。

    然后设置它的标题（继承自父类的），为这个服务数组进行初始化工作，接着给决定是否显示cell的accessoryView的属性赋值。

    如果需要显示取消按钮的话，创建一个取消按钮，设置这个按钮的响应函数为cancelAction，并把按钮设为导航栏的右边按钮。

    在一秒钟后调用initialWaitOver方法,主要是用来保证在显示给用户之前，有充足的时间让它搜索到服务的。

    最后返回自己。

    现在看一下cancelAction的实现：

1 - (void)cancelAction {2 [self.delegate browserViewController:self didResolveInstance:nil];3 }

   很简单地让它的委托去处理了，这个委托方法后面遇到时再说吧。

   再看initialWaitOver方法的实现：

1 - (void)initialWaitOver:(NSTimer \*)timer {2 self.initialWaitOver= YES;3 if (![self.services count])4 [self.tableView reloadData];5 }

    设置这个initialWaitOver属性为真，如果搜索到的服务数为0的话，重新刷新这个表视图。之所以要在1秒钟后调用这个方法，是要给我们留出足够的时间来搜索发现服务。

    下面是这个类自己实现的属性searchingForServicesString的set和get方法：

1 - (NSString \*)searchingForServicesString { 2 return \_searchingForServicesString; 3 } 4 5 // Holds the string that's displayed in the table view during service discovery. 6 - (void)setSearchingForServicesString:(NSString \*)searchingForServicesString { 7 if (\_searchingForServicesString != searchingForServicesString) { 8 [\_searchingForServicesString release]; 9 \_searchingForServicesString = [searchingForServicesString copy];10 11 // If there are no services, reload the table to ensure that searchingForServicesString appears.12 if ([self.services count] == 0) {13 [self.tableView reloadData];14 }15 }16 }

     这个属性的get方法就是正常的get方法没什么好说的，在set方法里，多了一点东西，也是当搜索到的服务数为0的时候，重新刷新表视图，目地是让这个搜索的服务的类型名字出现（后面在表视图的数据源方法里会看到）。

    再看ownName属性的set和get方法：

1 - (NSString \*)ownName { 2 return \_ownName; 3 } 4 5 // Holds the string that's displayed in the table view during service discovery. 6 - (void)setOwnName:(NSString \*)name { 7 8 if (\_ownName != name) { 9 \_ownName = [name copy];10 11 if (self.ownEntry)12 [self.services addObject:self.ownEntry];13 14 NSNetService\* service;15 16 for (service in self.services) {17 if ([service.name isEqual:name]) {18 self.ownEntry = service;19 [\_services removeObject:service];20 break;21 }22 }23 24 [self.tableView reloadData];25 }26 }

     同样的只有set方法是有额外操作的，如果这个ownEntry这个属性就是自己发布的服务是有效的话，就把这个服务加入到services这个服务数组里，然后对这个数组里的所有服务的名字进行比较，如果有服务的名字和我们想要让ownName这个属性设置成的名字一样的话，就把这个服务设为ownEntry，同时从services服务数组里把这个服务移除。这样做的用意是，当我们的ownEntry属性当前是有效的前提下，想换一个服务为作为自己的服务的话，这样就可以完成两个服务的交换。然后再次重新刷新表视图。

     下一个方法是真正开始搜索相应的服务的方法：

1 - (BOOL)searchForServicesOfType:(NSString \*)type inDomain:(NSString \*)domain { 2 3 [self stopCurrentResolve]; 4 [self.netServiceBrowser stop]; 5 [self.services removeAllObjects]; 6 7 NSNetServiceBrowser \*aNetServiceBrowser = [[NSNetServiceBrowser alloc] init]; 8 if(!aNetServiceBrowser) { 9 // The NSNetServiceBrowser couldn't be allocated and initialized.10 return NO;11 }12 13 aNetServiceBrowser.delegate = self;14 self.netServiceBrowser = aNetServiceBrowser;15 [aNetServiceBrowser release];16 [self.netServiceBrowser searchForServicesOfType:type inDomain:domain];17 18 [self.tableView reloadData];19 return YES;20 }

     第3-6行是一些清理性的工作，当我们在进行服务搜索的时候，调用stopCurrentResolve方法是假设有服务正在服务的话就先停止这个解析，然后对netServiceBrowser属性停止搜索服务的动作，它也是假设当这次搜索服务进行之前的状态是正在进行搜索服务的状态，把之前搜索到的所有服务从服务数组中移除。

     初始化一个NSNetServiceBrowser对象，如果初始化失败，返回搜索失败。

     把这个NSNetServiceBrowser对象的委托设为这个类本身（因为这个类已经声明自己符合它的协议了），然后把这个对象赋给netServiceBrowser属性。

     用这个searchForServicesOfType:inDomain：来搜索服务，第一个参数是要搜索的服务的类型名，第二个参数是要搜索的域的名字。

     重新刷新表视图，返回搜索成功。

     来看下这个stopCurrentResolve的具体内容：

1 - (void)stopCurrentResolve {2 self.needsActivityIndicator = NO;3 self.timer = nil;4 5 [self.currentResolve stop];6 self.currentResolve = nil;7 }

    给当前活动指示器赋值为假，调用timer属性的set方法，把这个timer置为nil，在这个timer的set方法里，把这个timer设为新的值之前先释放了之前的timer;停止当前正在尝试解析或发布的服务，把currentResolve属性置为nil。

    来看一下timer的set和get方法：

1 - (NSTimer \*)timer { 2 3 return \_timer; 4 } 5 6 // When this is called, invalidate the existing timer before releasing it. 7 - (void)setTimer:(NSTimer \*)newTimer { 8 9 [\_timer invalidate];10 [newTimer retain];11 [\_timer release];12 \_timer = newTimer;13 }

    值得说的是在这个timer的set方法里，在把timer设为新的newTimer前，先停止并释放原先的timer。

     下面该看看我们的表视图的数据源方法了，tableView就是通过这些数据源方法获取信息来往它自己的行里填充数据的。这个类里实现了3个数据源方法，第一个：

1 - (NSInteger)numberOfSectionsInTableView:(UITableView \*)tableView {2 return 1;3 }

     返回这个tableView的分组数，这里就是一个分组，所以返回1。

1 - (NSInteger)tableView:(UITableView \*)tableView numberOfRowsInSection:(NSInteger)section {2 // If there are no services and searchingForServicesString is set, show one row to tell the user.3 NSUInteger count = [self.services count];4 if (count == 0 && self.searchingForServicesString && self.initialWaitOver)5 return 1;6 7 return count;8 }

     这个用来返回分组中的行数。先获取存储服务的数组的元素个数，如果这个元素个数是0，并且我们搜索的服务类型的名字有效，并且initialWaitOver属性为真（这个代表着我们已经等待了一秒钟了，initialWaitOver方法执行完了）的话，我们就返回1，否则则直接返回服务数组的元素的个数。

1 - (UITableViewCell \*)tableView:(UITableView \*)tableView cellForRowAtIndexPath:(NSIndexPath \*)indexPath { 2 3 static NSString \*tableCellIdentifier = @"UITableViewCell"; 4 UITableViewCell \*cell = (UITableViewCell \*)[tableView dequeueReusableCellWithIdentifier:tableCellIdentifier]; 5 if (cell == nil) { 6 cell = [[[UITableViewCell alloc] initWithStyle:UITableViewCellStyleDefault reuseIdentifier:tableCellIdentifier] autorelease]; 7 } 8 9 NSUInteger count = [self.services count];10 if (count == 0 && self.searchingForServicesString) {11 // If there are no services and searchingForServicesString is set, show one row explaining that to the user.12 cell.textLabel.text = self.searchingForServicesString;13 cell.textLabel.textColor = [UIColor colorWithWhite:0.5 alpha:0.5];14 cell.accessoryType = UITableViewCellAccessoryNone;15 // Make sure to get rid of the activity indicator that may be showing if we were resolving cell zero but16 // then got didRemoveService callbacks for all services (e.g. the network connection went down).17 if (cell.accessoryView)18 cell.accessoryView = nil;19 return cell;20 }21 22 // Set up the text for the cell23 NSNetService \*service = [self.services objectAtIndex:indexPath.row];24 cell.textLabel.text = [service name];25 cell.textLabel.textColor = [UIColor blackColor];26 cell.accessoryType = self.showDisclosureIndicators ? UITableViewCellAccessoryDisclosureIndicator : UITableViewCellAccessoryNone;27 28 // Note that the underlying array could have changed, and we want to show the activity indicator on the correct cell29 if (self.needsActivityIndicator && self.currentResolve == service) {30 if (!cell.accessoryView) {31 CGRect frame = CGRectMake(0.0, 0.0, kProgressIndicatorSize, kProgressIndicatorSize);32 UIActivityIndicatorView\* spinner = [[UIActivityIndicatorView alloc] initWithFrame:frame];33 [spinner startAnimating];34 spinner.activityIndicatorViewStyle = UIActivityIndicatorViewStyleGray;35 [spinner sizeToFit];36 spinner.autoresizingMask = (UIViewAutoresizingFlexibleLeftMargin |37 UIViewAutoresizingFlexibleRightMargin |38 UIViewAutoresizingFlexibleTopMargin |39 UIViewAutoresizingFlexibleBottomMargin);40 cell.accessoryView = spinner;41 [spinner release];42 }43 } else if (cell.accessoryView) {44 cell.accessoryView = nil;45 }46 47 return cell;48 }

       这个方法向数据源请求一个cell走入到tableView相应的位置。具体就是配备cell的具体内容。

       第3-7行，这是苹果针对tableView的一种优化方式。

       这里定义了一个字符串来代表被用的cell对象，默认情况下这个字符串应该是你用的cell类的类名。不过其实你可以改变它为随意的值。

       这里调用了一个帮助方法"dequeueReusableCellWithIdentifier"来返回一个重用的cell，这到底是干什么的呢？这是一个很重要的性能优化，设想一下表视图可能包含一下非常大的数量的行数据，但是同一时间只有特定数量的行能显示在屏幕上。所以不用当一个行滚动到屏幕内的时候，不用每次都创建一个新的cell，系统可以通过重用这些已经创建过的但是不在屏幕显示循环内的cell来提升性能。这就是这个dequeueReusableCellWithIdentifier调用干了什么。

      如果没有可以重用的cell，我们会创建一个新的cell。

      第9行，得到服务数组的元素个数。

      如果服务数为0，并且searchingForServicesString属性有效，就显示一行，这一行的文本信息是这个searchingForServicesStirng属性，就是要搜索的服务类型的名字。然后，如果cell的accessoryView有效，把accessoryView置为nil（为了防止当我们的连接断开的时候，之前显示的服务已经移除的情况下这里还在显示信息）。

      第22-26行，正常地根据相应的索引信息设置相应的cell的textLable的内容为数组里相应的服务的名字，并且对显示的字的颜色进行设置，还根据showDisclosureIndicators

属性对cell的accessoryView的风格进行了设置。

      第29-41行，如果needsActivityIndicator属性为真，就是说我们需要显示活动指示器（就是我们常看到的齿轮状的转动的进度指示器），并且当前正在解析的服务就是当前这个行的服务的话，初始化并设置一个活动指示器，把它设为当前cell的accessoryView。否则的话，就判断这个当前cell的accessoryView是否有效，有效的话就把它置为nil。

      最后返回这个配置过的cell。

      看过了tableView的数据源方法了，现在该看一看它的相应委托方法了，一个tableView通过它的数据源方法来获取数据，通过它的委托方法来响应对它的操作。第一个：

1 - (NSIndexPath \*)tableView:(UITableView \*)tableView willSelectRowAtIndexPath:(NSIndexPath \*)indexPath {2 // Ignore the selection if there are no services as the searchingForServicesString cell3 // may be visible and tapping it would do nothing4 if ([self.services count] == 0)5 return nil;6 7 return indexPath;8 }

     这个方法的作用是当我们点击tableView的一行并松开手指的时候，告诉它的委托一个特定行将要被选中，如果服务数组的元素个数为0的话返回nil，表明不希望这个被点的行被选中（在这个例子里就是要忽略没有搜索到服务只有一个cell显示的是要搜索的服务的名字的情况），正常返回indexPath表示，希望被点的这行被选中。

    另一个委托方法：

1 - (void)tableView:(UITableView \*)tableView didSelectRowAtIndexPath:(NSIndexPath \*)indexPath { 2 // If another resolve was running, stop it & remove the activity indicator from that cell 3 4 if (self.currentResolve) { 5 // Get the indexPath for the active resolve cell 6 NSIndexPath\* indexPath = [NSIndexPath indexPathForRow:[self.services indexOfObject:self.currentResolve] inSection:0]; 7 8 // Stop the current resolve, which will also set self.needsActivityIndicator 9 [self stopCurrentResolve];10 11 // If we found the indexPath for the row, reload that cell to remove the activity indicator12 if (indexPath.row != NSNotFound)13 [self.tableView reloadRowsAtIndexPaths:[NSArray arrayWithObject:indexPath] withRowAnimation:UITableViewRowAnimationNone];14 }15 16 // Then set the current resolve to the service corresponding to the tapped cell17 self.currentResolve = [self.services objectAtIndex:indexPath.row];18 [self.currentResolve setDelegate:self];19 20 // Attempt to resolve the service. A value of 0.0 sets an unlimited time to resolve it. The user can21 // choose to cancel the resolve by selecting another service in the table view.22 [self.currentResolve resolveWithTimeout:0.0];23 24 // Make sure we give the user some feedback that the resolve is happening.25 // We will be called back asynchronously, so we don't want the user to think we're just stuck.26 // We delay showing this activity indicator in case the service is resolved quickly.27 self.timer = [NSTimer scheduledTimerWithTimeInterval:1.0 target:self selector:@selector(showWaiting:) userInfo:self.currentResolve repeats:NO];28 }

     这个方法是告诉它的委托，一个行现在被选中了。

     第4-14行，如果在一个行被选中的时候，有另一个服务解析正在运行，停止它，并从它对应的cell里移除活动指示器。

     第17-18行，设置这个当前在解析的服务为我们选中的行对应的服务。

     第22行，开始解析这个选中的服务，Timeout为0是说，对于这个解析过程来说没有超时时间限制。

     第27行，初始化一个NSTimer，用于在1秒钟后调用showWaiting方法，同时这个NSTimer的userInfo被设置为这个当前解析的服务。然后把这个NSTimer赋给timer属性。（还记得timer属性的set方法吗？在把timer赋为新的值之前是先取消并释放之前的timer。）

     还是先来看看showWaiting方法做什么吧，这样才能更好地理解这句话的作用，showWaiting的实现：

1 - (void)showWaiting:(NSTimer \*)timer { 2 if (timer == self.timer) { 3 NSNetService\* service = (NSNetService\*)[self.timer userInfo]; 4 if (self.currentResolve == service) { 5 self.needsActivityIndicator = YES; 6 7 NSIndexPath\* indexPath = [NSIndexPath indexPathForRow:[self.services indexOfObject:self.currentResolve] inSection:0]; 8 if (indexPath.row != NSNotFound) { 9 [self.tableView reloadRowsAtIndexPaths:[NSArray arrayWithObject:indexPath] withRowAnimation:UITableViewRowAnimationNone];10 // Deselect the row since the activity indicator shows the user something is happening.11 [self.tableView deselectRowAtIndexPath:indexPath animated:YES];12 }13 }14 }15 }

     这个showWaiting方法是从这个timer里得到在tableView:didSelectRowAtIndexPath方法里选中的服务，然后把needsActivityIndicator属性设为真，就是说现在需要显示活动指示器了，然后根据这个服务在服务数据里的索引号得到这个服务在tableView里的行索引，再根据这个得到的行索引对这个行进行刷新以显示这个活动指示器，并且也取消掉这个行的选中状态（就是取消这个行的高亮）。

     好的，知道了这个showWaiting方法干什么是不是就明白了tableView:didSelectRowAtIndexPath这个方法的第27行做什么了呀？它主要就是在解析选中的服务开始1秒钟后，显示活动指示器的。（事实上解析的过程一般情况下是很快的，会在1秒钟内完成，在解析完成之后会有一个回调方法被调用，在这个回调方法里其实是取消了这个timer的，也就是说如果1秒钟内完成了解析，这个showWaiting方法是不会调用的）

- (void)sortAndUpdateUI { // Sort the services by name. [self.services sortUsingSelector:@selector(localizedCaseInsensitiveCompareByName:)]; [self.tableView reloadData];}

     这是一个排序方法，用我们在NSNetService的分类的里定义的排序方法对这个服务数组里的服务进行排序，然后再根据这个顺序刷新显示tableView。

     该轮到来看一下这个NSNetServiceBrowserDelegate的协议方法了：

1 - (void)netServiceBrowser:(NSNetServiceBrowser \*)netServiceBrowser didRemoveService:(NSNetService \*)service moreComing:(BOOL)moreComing { 2 3 // If a service went away, stop resolving it if it's currently being resolved, 4 // remove it from the list and update the table view if no more events are queued. 5 6 if (self.currentResolve && [service isEqual:self.currentResolve]) { 7 [self stopCurrentResolve]; 8 } 9 [self.services removeObject:service];10 if (self.ownEntry == service)11 self.ownEntry = nil;12 13 // If moreComing is NO, it means that there are no more messages in the queue from the Bonjour daemon, so we should update the UI.14 // When moreComing is set, we don't update the UI so that it doesn't 'flash'.15 if (!moreComing) {16 [self sortAndUpdateUI];17 }18 }

    这个方法会在NSNetServiceBrowser探索到的服务中，有服务变为不可用了或者消失了的时候被调用。

    第6-7行，如果当前正在解析的服务就是这个消失的服务，停止对它的解析。

    第9-11行，把这个不可用的服务从服务数组里移除，如果这个不可用的的服务是我们自己的服务，把这个我们自己的服务设置为nil。

    第15-16行，如果没有更多的服务不可用消息到达的时候，更新服务列表及tableView。

1 - (void)netServiceBrowser:(NSNetServiceBrowser \*)netServiceBrowser didFindService:(NSNetService \*)service moreComing:(BOOL)moreComing { 2 // If a service came online, add it to the list and update the table view if no more events are queued. 3 if ([service.name isEqual:self.ownName]) 4 self.ownEntry = service; 5 else 6 [self.services addObject:service]; 7 8 // If moreComing is NO, it means that there are no more messages in the queue from the Bonjour daemon, so we should update the UI. 9 // When moreComing is set, we don't update the UI so that it doesn't 'flash'.10 if (!moreComing) {11 [self sortAndUpdateUI];12 }13 }

       这个会在搜索到可用服务时调用，如果新发现在的这个服务和我们自己程序发布的服务的名字一样的话，我们就把这个用来跟踪自己发布的服务的这个ownEntry属性设为这个服务。如果名字不一样的话，就把它加入到服务数组里。同样地，如果没有更多的可用服务被发现的话，就更新服务列表和tableView。

      下面再看两个NSNetServiceDelegate的协议方法吧，它们是在NSNetService的地址被解析失败或成功时调用的：

1 - (void)netService:(NSNetService \*)sender didNotResolve:(NSDictionary \*)errorDict {2 [self stopCurrentResolve];3 [self.tableView reloadData];4 }

     很显然这个是在NSNetService解析失败时调用的，在这个例子里这个方法是永远不会被调用的，因为前面设置解析时间限制时我们用的是0，它是说是没有限制的，永远不会超时的。这个方法里，停止当前在解析的服务，重新刷新tableView。

1 - (void)netServiceDidResolveAddress:(NSNetService \*)service {2 assert(service == self.currentResolve);3 4 [service retain];5 [self stopCurrentResolve];6 7 [self.delegate browserViewController:self didResolveInstance:service];8 [service release];9 }

     这个方法是在NSNetService解析成功时调用的。首先是用断言来保证触发这个回调方法的服务是我们正在解析的服务currentResolve。先对这个服务进行一次retain操作，然后停止对这个服务的解析操作，对它的委托调用browserViewController：didResolveInstance:方法（事实上它的委托是AppController类，这个方法后面再讲），先前执行了一次retain，现在对这个服务执行release操作。

     最后这个BrowserViewController类就剩下一个dealloc方法了：

1 - (void)dealloc { 2 3 // Cleanup any running resolve and free memory 4 [self stopCurrentResolve]; 5 self.services = nil; 6 [self.netServiceBrowser stop]; 7 self.netServiceBrowser = nil; 8 [\_searchingForServicesString release]; 9 [\_ownName release];10 [\_ownEntry release];11 12 [super dealloc];13 }

     就是一些必要的清理操作，这样这个BrowserViewController类就算讲完了。

## 再无战事

     终于所有的大块头都讲完了，我承认，这篇文章有点草草的意思，不过确实是代码太多，而且大部分都是正常的逻辑，没有太多特殊的知识点需要详述，所以就有很多地方都只是大概地讲了作用及流程。再往后的话，应该就只剩下正常的逻辑执行顺序需要理清楚就可以了。下一篇会是这个系列的完结篇，在完结篇里我会给出一个打印出来的这个程序的整个执行流程，根据这个流程，再来看我们讲到的所有方法会更好。

## 直捣黄龙

      在上一篇文章中我们提到要想理解AppController类的presentPicker方法必须先了解BrowserViewController类，并且上一篇文章中我们已经对这个BrowserViewController类进行了全面的了解，现在是时候回过头来看这个presentPicker方法了。

     打开AppController文件，我们来看这个方法的实现：

[复制代码](javascript:void(0);)

1 - (void) presentPicker:(NSString \*)name {

2 if (!\_picker) {

3 \_picker = [[Picker alloc] initWithFrame:[[UIScreen mainScreen] applicationFrame] type:[TCPServer bonjourTypeFromIdentifier:kGameIdentifier]];

4 \_picker.delegate = self;

5 }

6

7 \_picker.gameName = name;

8

9 if (!\_picker.superview) {

10 [\_window addSubview:\_picker];

11 }

12 }

[复制代码](javascript:void(0);)

      首先我们看到这个方法是有一个参数name的，它的作用是传给AppController的变量\_picker，让\_picker设置自己的一个Label的显示内容的。

      第2 行，判断这个\_picker是否为空。

      第3 行，如果这个\_picker是空的话，初始化一个Picker类的实例，并把这个实例赋给\_picker

      第4行，调用Picker类的属性delegate的set方法。

      第5行，调用Picker类的属性gameName的set方法。

      第9-10行，如果这个\_picker还没有加入到\_window（我们的程序的主窗口）的话，就把它作为子视图加入到\_window里。

      好，我们来看看Picker类的实现，先看Picker.h文件：

[复制代码](javascript:void(0);)

1 #import <UIKit/UIKit.h>

2 #import "BrowserViewController.h"

3

4 @interface Picker : UIView {

5

6 @private

7 UILabel \*\_gameNameLabel;

8 BrowserViewController \*\_bvc;

9 }

10

11 @property (nonatomic, assign) id<BrowserViewControllerDelegate> delegate;

12 @property (nonatomic, copy) NSString \*gameName;

13

14 - (id)initWithFrame:(CGRect)frame type:(NSString \*)type;

15

16 @end

[复制代码](javascript:void(0);)

    第1-2行，导入需要的UIKit框架和BrowserViewController的头文件。

    第4 行，可以看到，这个类是继承自UIView的。

    第7-8行，声明两个私有变量，一个是一个Label，用来显示游戏的名字，一个是BrowserViewController类的一个实例，用来搜索相应的服务。

    第10-11行，声明两个属性，一个是一个符合BrowserViewControllerDelegate协议的委托；一个是游戏名字的字符串。

    第14行，这个类的初始化方法，带有两个参数，一个是表示这个view的大小的，一个要搜索的服务的类型。

     接着来看看具体的实现，打开Picker.m文件：

[复制代码](javascript:void(0);)

1 #import "Picker.h"

2

3 #define kOffset 5.0

4

5 @interface Picker ()

6 @property (nonatomic, retain, readwrite) BrowserViewController \*bvc;

7 @property (nonatomic, retain, readwrite) UILabel \*gameNameLabel;

8 @end

9

10 @implementation Picker

11

12 @synthesize bvc = \_bvc;

13 @synthesize gameNameLabel = \_gameNameLabel;

[复制代码](javascript:void(0);)

     几乎合之前所有的.m文件的实现都一样，这里是先包含.h文件，然后定义了一个宏，接着为刚才那两个私有变量添加属性，然后为这两个新添加的属性合成set和get方法。

     先看这个类的清理方法：

[复制代码](javascript:void(0);)

1 - (void)dealloc {

2 // Cleanup any running resolve and free memory

3 [self.bvc release];

4 [self.gameNameLabel release];

5

6 [super dealloc];

7 }

[复制代码](javascript:void(0);)

    这是一些必要的清理操作，不做过多陈述。

    再来看一下这里为在Picker.h文件里添加的那两个属性实现的set和get方法：

[复制代码](javascript:void(0);)

- (id<BrowserViewControllerDelegate>)delegate {

return self.bvc.delegate;

}

- (void)setDelegate:(id<BrowserViewControllerDelegate>)delegate {

[self.bvc setDelegate:delegate];

}

- (NSString \*)gameName {

return self.gameNameLabel.text;

}

- (void)setGameName:(NSString \*)string {

[self.gameNameLabel setText:string];

[self.bvc setOwnName:string];

}

[复制代码](javascript:void(0);)

      我们可以看到其实这些set和get方法还是有些特殊性的。

      delegate的get方法返回的是这个Picker类的bvc属性的delegate，set方法也是设置的这个Picker类的bvc属性的delegate。

      这个gameName的get方法是返回这个Picker类的gameLabel属性的text，set方法不光设置了Picker类的gameLabel的text，还对这个Picker类的bvc调用了ownName调用了set方法（上一篇文章已经讲过了BrowserViewController类的所有方法了）。

      然后来看这个Picker类的初始化方法：

[复制代码](javascript:void(0);)

1 - (id)initWithFrame:(CGRect)frame type:(NSString\*)type {

2

3 if ((self = [super initWithFrame:frame])) {

4 // add autorelease to the NSNetServiceBrowser to release the browser once the connection has been

5 // established. An active browser can cause a delay in sending data.

6 // <rdar://problem/7000938>

7 self.bvc = [[[BrowserViewController alloc] initWithTitle:nil showDisclosureIndicators:NO showCancelButton:NO]autorelease];

8 [self.bvc searchForServicesOfType:type inDomain:@"local"];

9

10 self.opaque = YES;

11 self.backgroundColor = [UIColor blackColor];

12

13 UIImageView\* img = [[UIImageView alloc] initWithImage:[UIImage imageNamed:@"bg.png"]];

14 [self addSubview:img];

15 [img release];

16

17 CGFloat runningY = kOffset;

18 CGFloat width = self.bounds.size.width - 2 \* kOffset;

19

20 UILabel\* label = [[UILabel alloc] initWithFrame:CGRectZero];

21 [label setTextAlignment:UITextAlignmentCenter];

22 [label setFont:[UIFont boldSystemFontOfSize:15.0]];

23 [label setTextColor:[UIColor whiteColor]];

24 [label setShadowColor:[UIColor colorWithWhite:0.0 alpha:0.75]];

25 [label setShadowOffset:CGSizeMake(1,1)];

26 [label setBackgroundColor:[UIColor clearColor]];

27 label.text = @"Waiting for another player to join game:";

28 label.numberOfLines = 1;

29 [label sizeToFit];

30 label.frame = CGRectMake(kOffset, runningY, width, label.frame.size.height);

31 [self addSubview:label];

32

33 runningY += label.bounds.size.height;

34 [label release];

35

36 self.gameNameLabel = [[UILabel alloc] initWithFrame:CGRectZero];

37 [self.gameNameLabel setTextAlignment:UITextAlignmentCenter];

38 [self.gameNameLabel setFont:[UIFont boldSystemFontOfSize:24.0]];

39 [self.gameNameLabel setLineBreakMode:UILineBreakModeTailTruncation];

40 [self.gameNameLabel setTextColor:[UIColor whiteColor]];

41 [self.gameNameLabel setShadowColor:[UIColor colorWithWhite:0.0 alpha:0.75]];

42 [self.gameNameLabel setShadowOffset:CGSizeMake(1,1)];

43 [self.gameNameLabel setBackgroundColor:[UIColor clearColor]];

44 [self.gameNameLabel setText:@"Default Name"];

45 [self.gameNameLabel sizeToFit];

46 [self.gameNameLabel setFrame:CGRectMake(kOffset, runningY, width, self.gameNameLabel.frame.size.height)];

47 [self.gameNameLabel setText:@""];

48 [self addSubview:self.gameNameLabel];

49

50 runningY += self.gameNameLabel.bounds.size.height + kOffset \* 2;

51

52 label = [[UILabel alloc] initWithFrame:CGRectZero];

53 [label setTextAlignment:UITextAlignmentCenter];

54 [label setFont:[UIFont boldSystemFontOfSize:15.0]];

55 [label setTextColor:[UIColor whiteColor]];

56 [label setShadowColor:[UIColor colorWithWhite:0.0 alpha:0.75]];

57 [label setShadowOffset:CGSizeMake(1,1)];

58 [label setBackgroundColor:[UIColor clearColor]];

59 label.text = @"Or, join a different game:";

60 label.numberOfLines = 1;

61 [label sizeToFit];

62 label.frame = CGRectMake(kOffset, runningY, width, label.frame.size.height);

63 [self addSubview:label];

64

65 runningY += label.bounds.size.height + 2;

66

67 [self.bvc.view setFrame:CGRectMake(0, runningY, self.bounds.size.width, self.bounds.size.height - runningY)];

68 [self addSubview:self.bvc.view];

69

70 }

71

72 return self;

73 }

[复制代码](javascript:void(0);)

      第3行，根据参数frame的大小初始化这个picker类（它是继承自UIView的）。

      第7-8行，对bvc属性（BrowserViewController类）进行初始化，并在特定域中搜索特定服务。

      第10-11行，设置自己为不透明，并设置自己的背景颜色。

      第13-15行，给自己加一个背景图片。

      第17-63行，给自己加3个label，其中包括一个就是gameNameLabel属性，gameNameLabel属性用来显示游戏的名字（目前只是设置为@""了，但是在上一篇文章中我们已经了解了，当它搜索到自己发布的服务的时候，会把它设置自己发布的服务的名字），其它两个label是显示的一些提示性信息。

      第65-68行，设置Picker类的bvc属性的view（即BrowserViewController这个tableViewController的tableView）的显示范围，并把这个bvc的view作为子视图加入到picker里。

      第72行，返回自己。

      现在这个AppController类的presentPicker方法是不是完全明白了？呵呵

      其实到这儿为止，这个例子里所涉及到的关键操作基本上都讲到了，剩下没讲的方法的就是一些具体的操作上的东西了，我在后面会大致的讲一下它们的作用，不再逐句解释。

      下面来迅速地说明一下这些剩余的方法的作用。首先是TapView.h文件：

[复制代码](javascript:void(0);)

1 #import <UIKit/UIKit.h>

2

3 @interface TapView : UIView

4 {

5 BOOL localTouch;

6 BOOL remoteTouch;

7 }

8

9 - (void) touchDown:(BOOL)remote;

10 - (void) touchUp:(BOOL)remote;

11

12 @end

[复制代码](javascript:void(0);)

     我们看到这个TapView是继承自UIView的，它有两个布尔变量用来标识是远程的touch还是本地的touch。两个方法表示按下和抬起手指时针对的处理方法。

     来看TapView.m文件：

[复制代码](javascript:void(0);)

1 - (void) touchDown:(BOOL)remote

2 {

3 // set "tap down" visual state if necessary

4 if(!localTouch && !remoteTouch)

5 self.frame=CGRectInset(self.frame, kActivationInset, kActivationInset);

6

7 if (remote)

8 remoteTouch = YES;

9 else

10 localTouch = YES;

11 }

[复制代码](javascript:void(0);)

     这个方法是在手指按下游戏中的色块时把色块变小来表示色块被点击了，并同时设置相应的远程或本地触摸的标记为真。

[复制代码](javascript:void(0);)

1 - (void) touchUp:(BOOL)remote

2 {

3 BOOL wasDown = localTouch || remoteTouch;

4

5 if (remote)

6 remoteTouch = NO;

7 else

8 localTouch = NO;

9

10 BOOL isDown = localTouch || remoteTouch;

11

12 // run "tap up" visual animation if necessary

13 if(wasDown != isDown) {

14 [UIView beginAnimations:nil context:NULL];

15 [UIView setAnimationDuration:0.1];

16 self.frame = CGRectInset(self.frame, -kActivationInset, -kActivationInset);

17 [UIView commitAnimations];

18 }

19 }

[复制代码](javascript:void(0);)

    这个方法是在手指点击游戏中的色块后手指离开屏幕时用来恢复色块大小为原始大小，并还原相应的远程或本地触摸标识为假。

1 - (void) localTouchUp

2 {

3 [self touchUp:NO];

4 [(AppController\*)[[UIApplication sharedApplication] delegate] deactivateView:self];

5 }

   这个方法是在当发生的是本机的触摸事件时调用的方法，它调用touchUp方法，并给这个方法的表示远程触摸的参数传入假，然后调用AppController的deactivateView方法。

   然后是touch事件的3个协议方法：

[复制代码](javascript:void(0);)

1 - (void) touchesBegan:(NSSet\*)touches withEvent:(UIEvent\*)event

2 {

3 [self touchDown:NO];

4 [(AppController\*)[[UIApplication sharedApplication] delegate] activateView:self];

5 }

6

7 - (void) touchesEnded:(NSSet\*)touches withEvent:(UIEvent\*)event

8 {

9 [self localTouchUp];

10 }

11

12 - (void)touchesCancelled:(NSSet \*)touches withEvent:(UIEvent \*)event

13 {

14 [self localTouchUp];

15 }

[复制代码](javascript:void(0);)

    在began方法里是调用touchDown方法，并传入NO给touchDown方法中表示远程的参数。end方法和cancel方法都只是调用localTouchUp方法而已。

## 回头望月

     是时候回到AppController类认真地看一下了，来看看它的头文件AppController.h吧：

[复制代码](javascript:void(0);)

#import "TapView.h"

#import "BrowserViewController.h"

#import "Picker.h"

#import "TCPServer.h"

@interface AppController : NSObject <UIApplicationDelegate,   
 　UIActionSheetDelegate,   
 　　　　　　　 BrowserViewControllerDelegate,

TCPServerDelegate,   
 NSStreamDelegate>

{

UIWindow \*\_window;

Picker \*\_picker;

TCPServer \*\_server;

NSInputStream \*\_inStream;

NSOutputStream \*\_outStream;

BOOL \_inReady;

BOOL \_outReady;

}

- (void) activateView:(TapView\*)view;

- (void) deactivateView:(TapView\*)view;

@end

[复制代码](javascript:void(0);)

    在这里可以看到它包含了我们前面讲过的所有的主要的类，这一看就是要做大事的类呀呵呵。再看它声明自己遵守的协议，呵呵，有五个之多。

    实例变量\_window这是我们的主窗口，\_picker一个Picker类的实例，用来搜索和显示发布的特定服务，\_server这个TCPServer类的实例用来发布服务，\_inStream和\_outStream这是一对输入输出流，然后一对布尔变量\_inReady和\_outReady表示输入输出流的准备状态。

    一对方法用来激活和去激活我们在游戏中的色块。

    来看看这个AppController的具体实现，打开AppController.m文件：

[复制代码](javascript:void(0);)

1 #import "AppController.h"

2 #import "Picker.h"

3

4 #define kNumPads 3

5 #define kGameIdentifier @"witap"

6

7

8 @interface AppController ()

9 - (void) setup;

10 - (void) presentPicker:(NSString \*)name;

11 @end

[复制代码](javascript:void(0);)

    没有过多要解释的东西，这里的东西该解释的前面都已经解释过了呵呵。

[复制代码](javascript:void(0);)

1 - (void) \_showAlert:(NSString \*)title

2 {

3 UIAlertView \*alertView = [[UIAlertView alloc] initWithTitle:title message:@"Check your networking configuration." delegate:self cancelButtonTitle:@"OK" otherButtonTitles:nil];

4 [alertView show];

5 [alertView release];

6 }

[复制代码](javascript:void(0);)

   这个是用来显示警告信息的。

   applicationDidFinishLaunching方法，dealloc方法，setup方法，presentPicker方法我们都已经讲过了，接着是destroyPicker方法：

1 - (void) destroyPicker {

2 [\_picker removeFromSuperview];

3 [\_picker release];

4 \_picker = nil;

5 }

     这个是在我们进入游戏后，销毁用来显示搜索到的服务列表的视图(即\_picker)的。

1 - (void) alertView:(UIAlertView \*)alertView clickedButtonAtIndex:(NSInteger)buttonIndex

2 {

3 [self setup];

4 }

      这个方法是在当我们在远端设备断开连接的时候显示一个错误或警告的时候，点击了警告上的按钮的话调用的方法，这是重新调用setup方法，重新开始新的服务发布和搜索过程。

1 - (void) send:(const uint8\_t)message

2 {

3 if (\_outStream && [\_outStream hasSpaceAvailable])

4 if([\_outStream write:(const uint8\_t \*)&message maxLength:sizeof(const uint8\_t)] == -1)

5 [self \_showAlert:@"Failed sending data to peer"];

6 }

    这个是用来发送消息对无端设备的，其实就是向输出流写入数据。

[复制代码](javascript:void(0);)

1 - (void) activateView:(TapView \*)view

2 {

3 [self send:[view tag] | 0x80];

4 }

5

6 - (void) deactivateView:(TapView \*)view

7 {

8 [self send:[view tag] & 0x7f];

9 }

[复制代码](javascript:void(0);)

    游戏中色块的激活和去激活方法，它们的实现内容都是调用send方法，不同的是向send方法发送的数据不一样。

[复制代码](javascript:void(0);)

1 - (void) openStreams

2 {

3 \_inStream.delegate = self;

4 [\_inStream scheduleInRunLoop:[NSRunLoop currentRunLoop] forMode:NSDefaultRunLoopMode];

5 [\_inStream open];

6 \_outStream.delegate = self;

7 [\_outStream scheduleInRunLoop:[NSRunLoop currentRunLoop] forMode:NSDefaultRunLoopMode];

8 [\_outStream open];

9 }

[复制代码](javascript:void(0);)

    打开输入输出流，并把它们加入到当前runLoop中，并把它们的委托设为AppController自己。

[复制代码](javascript:void(0);)

1 - (void) browserViewController:(BrowserViewController \*)bvc didResolveInstance:(NSNetService \*)netService

2 {

3 if (!netService) {

4 [self setup];

5 return;

6 }

7

8 // note the following method returns \_inStream and \_outStream with a retain count that the caller must eventually release

9 if (![netService getInputStream:&\_inStream outputStream:&\_outStream]) {

10 [self \_showAlert:@"Failed connecting to server"];

11 return;

12 }

14 [self openStreams];

15 }

[复制代码](javascript:void(0);)

     这个是在解析我们要连接的服务完成的时候，在BrowserViewController的NSNetService的协议方法里调用的。这里做的处理是，如果这个解析的服务无效，重新调用setup方法并返回。接着对这个服务调用getInputStrem:outPutStream方法来判断这个连接是否真正连接好。然后调用openStreams方法，打开输入输出流。（这个方法是在发起连接服务的设备上调用的）

[复制代码](javascript:void(0);)

1 - (void) stream:(NSStream \*)stream handleEvent:(NSStreamEvent)eventCode

2 {

3

4 UIAlertView \*alertView;

5 switch(eventCode) {

6 case NSStreamEventOpenCompleted:

7 {

8 NSLog(@"case open completed\n");

9 [self destroyPicker];

10

11 [\_server release];

12 \_server = nil;

13

14 if (stream == \_inStream)

15 \_inReady = YES;

16 else

17 \_outReady = YES;

18

19 if (\_inReady && \_outReady) {

20 NSLog(@"show the message game started\n");

21 alertView = [[UIAlertView alloc] initWithTitle:@"Game started!" message:nil delegate:nil cancelButtonTitle:nil otherButtonTitles:@"Continue", nil];

22 [alertView show];

23 [alertView release];

24 }

25 break;

26 }

27 case NSStreamEventHasBytesAvailable:

28 {

29 NSLog(@"case has bytes available\n");

30 if (stream == \_inStream)

31 {

32 uint8\_t b;

33 int len = 0;

34 len = [\_inStream read:&b maxLength:sizeof(uint8\_t)];

35 if(len <= 0)

36 {

37 if ([stream streamStatus] != NSStreamStatusAtEnd)

38 [self \_showAlert:@"Failed reading data from peer"];

39 }

40 else

41 {

42 //We received a remote tap update, forward it to the appropriate view

43 if(b & 0x80)

44 [(TapView \*)[\_window viewWithTag:b & 0x7f] touchDown:YES];

45 else

46 [(TapView \*)[\_window viewWithTag:b] touchUp:YES];

47 }

48 }

49 break;

50 }

51 case NSStreamEventErrorOccurred:

52 {

53 NSLog(@"case error occurred\n");

54 [self \_showAlert:@"Error encountered on stream!"];

55 break;

56 }

57

58 case NSStreamEventEndEncountered:

59 {

60 NSLog(@"case end encountered\n");

61 NSArray \*array = [\_window subviews];

62 TapView \*view;

63 UIAlertView \*alertView;

64

65

66 //Notify all tap views

67 for(view in array)

68 [view touchUp:YES];

69

70 alertView = [[UIAlertView alloc] initWithTitle:@"Peer Disconnected!" message:nil delegate:self cancelButtonTitle:nil otherButtonTitles:@"Continue", nil];

71 [alertView show];

72 [alertView release];

73

74 break;

75 }

76 default:

77 {

78 NSLog(@"case there is no corresponding case match this event\n");

79 }

80 }

81 }

[复制代码](javascript:void(0);)

      这个方法比较重要，这个是流的协议方法。根据不同的流事件做不同的处理：

      当流事件是流打开完成的时候，销毁\_picker，释放\_service，根据触发事件的流设置对就的准备好状态为真，当输入输出两个流都准备好的时候，显示一个警告窗口告诉我们游戏开始。

      当流事件是有可用数据的时候，判断这个触发事件的流是不是输入流，如果是就处理，不是就不处理。当是输入流触发事件的时候，从流中读出信息，如果读取出错的话显示一个警告，如果成功的话，根据得到的数据，相应的激活或去激活相应的色块。（具体这个地方的位运算，只要把send方法发送的数据转成2进制就明白这个地方位运算的意义了）

     当流事件是出现错误的时候，显示一个警告。（前面已经说过了如果这个警告上的按钮被点击的话会调用setup方法重新开始整个过程）

     当流事件是流已经结束的时候（比如远端设备连接断开的时候），把所有色块恢复原始状态，显示一个警告，说明对方断开连接。

     其它事件情况下输出一条log语句，没有相应的匹配的事件。

      最后是两个TCPServerDelegate协议的方法：

1 - (void) serverDidEnableBonjour:(TCPServer \*)server withName:(NSString \*)string

2 {

3 [self presentPicker:string];

4 }

     这个是在服务发布成功之后，让\_picker显示服务的名字的。

[复制代码](javascript:void(0);)

1 - (void)didAcceptConnectionForServer:(TCPServer \*)server inputStream:(NSInputStream \*)istr outputStream:(NSOutputStream \*)ostr

2 {

3

4 if (\_inStream || \_outStream || server != \_server)

5 return;

6

7 [\_server release];

8 \_server = nil;

9

10 \_inStream = istr;

11 [\_inStream retain];

12 \_outStream = ostr;

13 [\_outStream retain];

14

15 [self openStreams];

16 }

[复制代码](javascript:void(0);)

      这个是在接受了服务的连接请求之后调用的（这个方法只会在被动连接的一方设备上调用）。如果输入或输出流其中有一个有效或者这个方法中的server参数不是这个被连接的设备上的服务本身的话，直接返回。

     如果不是上述情况的话，就释放并停止这个服务，把它赋为nil，把输入输出流赋值为参数istr和ostr，打开输入输出流。

## 重新理顺所有逻辑

    到此，这个例子中所有的方法我们基本上都讲完了，现在你应该能知道这个例子中的每一个方法干了什么了，但是这个只是明白了每个方法干了什么还是不够的，目前看来我们的逻辑还是不够清晰，不够顺畅。下面我给出这个例子的执行流程（并不是所有方法都列了出来，是绝大多数都列出来了），对照着这个流程来看每个方法会很容易理清楚所有的逻辑的。

## [执行流程下载地址](http://dl.dbank.com/c0hydfltcs)

   （一）真机和模拟器一起调试，模拟器作为连接发起方时的执行过程：

**下面的过程是模拟器运行后自动执行的程序流程**

**[AppController.m]--[applicationDidFinishLaunching:]**

**[AppController.m]--[setup]**

**[TCPServer.m]--[init]**

**[TCPServer.m]--[start:]**

**[TCPServer.m]--[bonjourTypeFromIdentifier:]**

**[TCPServer.m]--[enableBonjourWithDomain:applicationProtocol:name:]**

**[AppController.m]--[presentPicker:]**

**[TCPServer.m]--[bonjourTypeFromIdentifier:]**

**[Picker.m]--[initWithFrame:type:]**

**[BrowserViewController.m]--[initWithTitle:showDisclosureIndicators:showCancelButton:]**

**[BrowserViewController.m]--[searchForServicesOfType:inDomain:]**

**[BrowserViewController.m]--[stopCurrentResolve]**

**[BrowserViewController.m]--[setTimer:]**

**[BrowserViewController.m]--[numberOfSectionsInTableView:]**

**[BrowserViewController.m]--[tableView:numberOfRowsInSection:]**

**[BrowserViewController.m]--[searchingForServicesString]**

**[Picker.m]--[setDelegate:]**

**[Picker.m]--[setGameName:]**

**[BrowserViewController.m]--[setOwnName:]**

**[TCPServer.m]--[netServiceDidPublish:]**

**[AppController.m]--[serverDidEnableBonjour:withName:]**

**[AppController.m]--[presentPicker:]**

**[Picker.m]--[setGameName:]**

**[BrowserViewController.m]--[setOwnName:]**

**[BrowserViewController.m]--[numberOfSectionsInTableView:]**

**[BrowserViewController.m]--[tableView:numberOfRowsInSection:]**

**[BrowserViewController.m]--[searchingForServicesString]**

**[BrowserViewController.m]--[netServiceBrowser:didFindService:moreComing:]**

**the service is: <NSNetService 0x4b3fb00> local. \_witap.\_tcp. dingwenjie**

**[BrowserViewController.m]--[sortAndUpdateUI]**

**[BrowserViewController.m]--[numberOfSectionsInTableView:]**

**[BrowserViewController.m]--[tableView:numberOfRowsInSection:]**

**[BrowserViewController.m]--[searchingForServicesString]**

**[BrowserViewController.m]--[initialWaitOver:]**

**[BrowserViewController.m]--[numberOfSectionsInTableView:]**

**[BrowserViewController.m]--[tableView:numberOfRowsInSection:]**

**[BrowserViewController.m]--[searchingForServicesString]**

**下面的过程是真机运行程序后，模拟器自动执行的程序流程**

**[BrowserViewController.m]--[netServiceBrowser:didFindService:moreComing:]**

**the service is: <NSNetService 0x4e1b840> local. \_witap.\_tcp. “DWJ”的 iPad**

**[BrowserViewController.m]--[sortAndUpdateUI]**

**[BrowserViewController.m]--[numberOfSectionsInTableView:]**

**[BrowserViewController.m]--[tableView:numberOfRowsInSection:]**

**[BrowserViewController.m]--[tableView:cellForRowAtIndexPath:]**

**下面的过程点击了模拟器的服务列表之后，建立了联机状态**

**[BrowserViewController.m]--[tableView:willSelectRowAtIndexPath:]**

**[BrowserViewController.m]--[tableView:didSelectRowAtIndexPath:]**

**[BrowserViewController.m]--[setTimer:]**

**[BrowserViewController.m]--[netServiceDidResolveAddress:]**

**[BrowserViewController.m]--[stopCurrentResolve]**

**[BrowserViewController.m]--[setTimer:]**

**[AppController.m]--[browserViewController:didResolveInstance:]**

**the netService didResolveInstance is: <NSNetService 0x4e0f2a0> local. \_witap.\_tcp. “DWJ”的 iPad**

**[AppController.m]--[openStreams]**

**[AppController.m]--[stream:handleEvent:]**

**case open completed**

**[AppController.m]--[destroyPicker]**

**[Picker.m]--[dealloc]**

**[BrowserViewController.m]--[dealloc]**

**[BrowserViewController.m]--[stopCurrentResolve]**

**[BrowserViewController.m]--[setTimer:]**

**[TCPServer.m]--[dealloc]**

**[TCPServer.m]--[stop]**

**[TCPServer.m]--[disableBonjour]**

**about to call NetService:stop**

**[AppController.m]--[stream:handleEvent:]**

**case open completed**

**[AppController.m]--[destroyPicker]**

**show the message game started**

**[AppController.m]--[stream:handleEvent:]**

**case there is no corresponding case match this event**

**下面是点击一次本机的TapView执行如下**

**2011-12-02 15:26:45.055 WiTap[6260:40b] [/Users/administrator/Downloads/WiTap/TapView.m]--[touchesBegan:withEvent:]**

**2011-12-02 15:26:45.056 WiTap[6260:40b] [/Users/administrator/Downloads/WiTap/TapView.m]--[touchDown:]**

**2011-12-02 15:26:45.057 WiTap[6260:40b] [/Users/administrator/Downloads/WiTap/AppController.m]--[activateView:]**

**2011-12-02 15:26:45.058 WiTap[6260:40b] [/Users/administrator/Downloads/WiTap/AppController.m]--[send:]**

**2011-12-02 15:26:45.059 WiTap[6260:40b] [/Users/administrator/Downloads/WiTap/AppController.m]--[stream:handleEvent:]**

**2011-12-02 15:26:45.059 WiTap[6260:40b] case there is no corresponding case match this event**

**2011-12-02 15:26:45.135 WiTap[6260:40b] [/Users/administrator/Downloads/WiTap/TapView.m]--[touchesEnded:withEvent:]**

**2011-12-02 15:26:45.135 WiTap[6260:40b] [/Users/administrator/Downloads/WiTap/TapView.m]--[localTouchUp]**

**2011-12-02 15:26:45.136 WiTap[6260:40b] [/Users/administrator/Downloads/WiTap/TapView.m]--[touchUp:]**

**2011-12-02 15:26:45.137 WiTap[6260:40b] [/Users/administrator/Downloads/WiTap/AppController.m]--[deactivateView:]**

**2011-12-02 15:26:45.137 WiTap[6260:40b] [/Users/administrator/Downloads/WiTap/AppController.m]--[send:]**

**2011-12-02 15:26:45.139 WiTap[6260:40b] [/Users/administrator/Downloads/WiTap/AppController.m]--[stream:handleEvent:]**

**2011-12-02 15:26:45.140 WiTap[6260:40b] case there is no corresponding case match this event**

**下面是点击一次远程的TapView执行如下**

**2011-12-02 15:29:00.261 WiTap[6260:40b] [/Users/administrator/Downloads/WiTap/AppController.m]--[stream:handleEvent:]**

**2011-12-02 15:29:00.262 WiTap[6260:40b] case has bytes available**

**2011-12-02 15:29:00.263 WiTap[6260:40b] [/Users/administrator/Downloads/WiTap/TapView.m]--[touchDown:]**

**2011-12-02 15:29:00.344 WiTap[6260:40b] [/Users/administrator/Downloads/WiTap/AppController.m]--[stream:handleEvent:]**

**2011-12-02 15:29:00.344 WiTap[6260:40b] case has bytes available**

**2011-12-02 15:29:00.345 WiTap[6260:40b] [/Users/administrator/Downloads/WiTap/TapView.m]--[touchUp:]**

**真机退出程序后，模拟器执行**

**2011-12-05 10:55:01.545 WiTap[350:40b] [/Users/administrator/Downloads/WiTap/AppController.m]--[stream:handleEvent:]**

**2011-12-05 10:55:01.546 WiTap[350:40b] case has bytes available**

**2011-12-05 10:55:01.547 WiTap[350:40b] [/Users/administrator/Downloads/WiTap/AppController.m]--[stream:handleEvent:]**

**2011-12-05 10:55:01.547 WiTap[350:40b] case end encountered**

**2011-12-05 10:55:01.548 WiTap[350:40b] [/Users/administrator/Downloads/WiTap/TapView.m]--[touchUp:]**

**2011-12-05 10:55:01.548 WiTap[350:40b] [/Users/administrator/Downloads/WiTap/TapView.m]--[touchUp:]**

**2011-12-05 10:55:01.549 WiTap[350:40b] [/Users/administrator/Downloads/WiTap/TapView.m]--[touchUp:]**

**2011-12-05 10:55:01.550 WiTap[350:40b] [/Users/administrator/Downloads/WiTap/TapView.m]--[touchUp:]**

**2011-12-05 10:55:01.550 WiTap[350:40b] [/Users/administrator/Downloads/WiTap/TapView.m]--[touchUp:]**

**2011-12-05 10:55:01.551 WiTap[350:40b] [/Users/administrator/Downloads/WiTap/TapView.m]--[touchUp:]**

**2011-12-05 10:55:01.551 WiTap[350:40b] [/Users/administrator/Downloads/WiTap/TapView.m]--[touchUp:]**

**2011-12-05 10:55:01.552 WiTap[350:40b] [/Users/administrator/Downloads/WiTap/TapView.m]--[touchUp:]**

**2011-12-05 10:55:01.553 WiTap[350:40b] [/Users/administrator/Downloads/WiTap/TapView.m]--[touchUp:]**

**模拟器点击alterView的continue按钮后执行**

**2011-12-05 10:55:41.627 WiTap[350:40b] [/Users/administrator/Downloads/WiTap/AppController.m]--[alertView:clickedButtonAtIndex:]**

**2011-12-05 10:55:41.628 WiTap[350:40b] [/Users/administrator/Downloads/WiTap/AppController.m]--[setup]**

**余下过程和模拟器刚启动是执行的一样**

（二）真机和模拟器一起调试，模拟器作为连接响应方时的执行过程

**以下过程是真机先启动，模拟器后启动的时候，模拟器执行的流程**

**2011-12-05 11:02:26.410 WiTap[422:40b] [/Users/administrator/Downloads/WiTap/AppController.m]--[applicationDidFinishLaunching:]**

**2011-12-05 11:02:26.416 WiTap[422:40b] [/Users/administrator/Downloads/WiTap/AppController.m]--[setup]**

**2011-12-05 11:02:26.417 WiTap[422:40b] [/Users/administrator/Downloads/WiTap/Networking/TCPServer.m]--[init]**

**2011-12-05 11:02:26.418 WiTap[422:40b] [/Users/administrator/Downloads/WiTap/Networking/TCPServer.m]--[start:]**

**2011-12-05 11:02:26.419 WiTap[422:40b] [/Users/administrator/Downloads/WiTap/Networking/TCPServer.m]--[bonjourTypeFromIdentifier:]**

**2011-12-05 11:02:26.420 WiTap[422:40b] [/Users/administrator/Downloads/WiTap/Networking/TCPServer.m]--[enableBonjourWithDomain:applicationProtocol:name:]**

**2011-12-05 11:02:26.422 WiTap[422:40b] [/Users/administrator/Downloads/WiTap/AppController.m]--[presentPicker:]**

**2011-12-05 11:02:26.422 WiTap[422:40b] [/Users/administrator/Downloads/WiTap/Networking/TCPServer.m]--[bonjourTypeFromIdentifier:]**

**2011-12-05 11:02:26.423 WiTap[422:40b] [/Users/administrator/Downloads/WiTap/Picker.m]--[initWithFrame:type:]**

**2011-12-05 11:02:26.424 WiTap[422:40b] [/Users/administrator/Downloads/WiTap/Networking/BrowserViewController.m]--[initWithTitle:showDisclosureIndicators:showCancelButton:]**

**2011-12-05 11:02:26.425 WiTap[422:40b] [/Users/administrator/Downloads/WiTap/Networking/BrowserViewController.m]--[searchForServicesOfType:inDomain:]**

**2011-12-05 11:02:26.425 WiTap[422:40b] [/Users/administrator/Downloads/WiTap/Networking/BrowserViewController.m]--[stopCurrentResolve]**

**2011-12-05 11:02:26.426 WiTap[422:40b] [/Users/administrator/Downloads/WiTap/Networking/BrowserViewController.m]--[setTimer:]**

**2011-12-05 11:02:26.429 WiTap[422:40b] [/Users/administrator/Downloads/WiTap/Networking/BrowserViewController.m]--[numberOfSectionsInTableView:]**

**2011-12-05 11:02:26.430 WiTap[422:40b] [/Users/administrator/Downloads/WiTap/Networking/BrowserViewController.m]--[tableView:numberOfRowsInSection:]**

**2011-12-05 11:02:26.430 WiTap[422:40b] [/Users/administrator/Downloads/WiTap/Networking/BrowserViewController.m]--[searchingForServicesString]**

**2011-12-05 11:02:26.432 WiTap[422:40b] [/Users/administrator/Downloads/WiTap/Picker.m]--[setDelegate:]**

**2011-12-05 11:02:26.433 WiTap[422:40b] [/Users/administrator/Downloads/WiTap/Picker.m]--[setGameName:]**

**2011-12-05 11:02:26.433 WiTap[422:40b] [/Users/administrator/Downloads/WiTap/Networking/BrowserViewController.m]--[setOwnName:]**

**2011-12-05 11:02:26.441 WiTap[422:40b] [/Users/administrator/Downloads/WiTap/Networking/BrowserViewController.m]--[netServiceBrowser:didFindService:moreComing:]**

**2011-12-05 11:02:26.444 WiTap[422:40b] the service is: <NSNetService 0x4e3a3f0> local. \_witap.\_tcp. “DWJ”的 iPad**

**2011-12-05 11:02:26.445 WiTap[422:40b] [/Users/administrator/Downloads/WiTap/Networking/BrowserViewController.m]--[sortAndUpdateUI]**

**2011-12-05 11:02:26.446 WiTap[422:40b] [/Users/administrator/Downloads/WiTap/Networking/BrowserViewController.m]--[numberOfSectionsInTableView:]**

**2011-12-05 11:02:26.447 WiTap[422:40b] [/Users/administrator/Downloads/WiTap/Networking/BrowserViewController.m]--[tableView:numberOfRowsInSection:]**

**2011-12-05 11:02:26.448 WiTap[422:40b] [/Users/administrator/Downloads/WiTap/Networking/BrowserViewController.m]--[tableView:cellForRowAtIndexPath:]**

**2011-12-05 11:02:27.159 WiTap[422:40b] [/Users/administrator/Downloads/WiTap/Networking/TCPServer.m]--[netServiceDidPublish:]**

**2011-12-05 11:02:27.160 WiTap[422:40b] [/Users/administrator/Downloads/WiTap/AppController.m]--[serverDidEnableBonjour:withName:]**

**2011-12-05 11:02:27.161 WiTap[422:40b] [/Users/administrator/Downloads/WiTap/AppController.m]--[presentPicker:]**

**2011-12-05 11:02:27.162 WiTap[422:40b] [/Users/administrator/Downloads/WiTap/Picker.m]--[setGameName:]**

**2011-12-05 11:02:27.162 WiTap[422:40b] [/Users/administrator/Downloads/WiTap/Networking/BrowserViewController.m]--[setOwnName:]**

**2011-12-05 11:02:27.163 WiTap[422:40b] [/Users/administrator/Downloads/WiTap/Networking/BrowserViewController.m]--[numberOfSectionsInTableView:]**

**2011-12-05 11:02:27.163 WiTap[422:40b] [/Users/administrator/Downloads/WiTap/Networking/BrowserViewController.m]--[tableView:numberOfRowsInSection:]**

**2011-12-05 11:02:27.164 WiTap[422:40b] [/Users/administrator/Downloads/WiTap/Networking/BrowserViewController.m]--[tableView:cellForRowAtIndexPath:]**

**2011-12-05 11:02:27.409 WiTap[422:40b] [/Users/administrator/Downloads/WiTap/Networking/BrowserViewController.m]--[netServiceBrowser:didFindService:moreComing:]**

**2011-12-05 11:02:27.410 WiTap[422:40b] the service is: <NSNetService 0x4e3fee0> local. \_witap.\_tcp. dingwenjie**

**2011-12-05 11:02:27.411 WiTap[422:40b] [/Users/administrator/Downloads/WiTap/Networking/BrowserViewController.m]--[sortAndUpdateUI]**

**2011-12-05 11:02:27.411 WiTap[422:40b] [/Users/administrator/Downloads/WiTap/Networking/BrowserViewController.m]--[numberOfSectionsInTableView:]**

**2011-12-05 11:02:27.412 WiTap[422:40b] [/Users/administrator/Downloads/WiTap/Networking/BrowserViewController.m]--[tableView:numberOfRowsInSection:]**

**2011-12-05 11:02:27.413 WiTap[422:40b] [/Users/administrator/Downloads/WiTap/Networking/BrowserViewController.m]--[tableView:cellForRowAtIndexPath:]**

**2011-12-05 11:02:27.425 WiTap[422:40b] [/Users/administrator/Downloads/WiTap/Networking/BrowserViewController.m]--[initialWaitOver:]**

**以下过程是真机点击服务列表后，模拟器执行的流程**

**2011-12-05 11:03:49.417 WiTap[422:40b]**

**\*\*\*\*\*\*\*\*\*\*\*\*\*TCPServerAcceptCallBack is runing\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**2011-12-05 11:03:49.418 WiTap[422:40b] the server in call back is: <TCPServer = 0x04E35CE0 | port 49885 | netService = <NSNetService 0x4b212e0> local. \_witap.\_tcp. dingwenjie>**

**2011-12-05 11:03:49.418 WiTap[422:40b] [/Users/administrator/Downloads/WiTap/Networking/TCPServer.m]--[handleNewConnectionFromAddress:inputStream:outputStream:]**

**2011-12-05 11:03:49.419 WiTap[422:40b] [/Users/administrator/Downloads/WiTap/AppController.m]--[didAcceptConnectionForServer:inputStream:outputStream:]**

**2011-12-05 11:03:49.420 WiTap[422:40b] [/Users/administrator/Downloads/WiTap/Networking/TCPServer.m]--[dealloc]**

**2011-12-05 11:03:49.421 WiTap[422:40b] [/Users/administrator/Downloads/WiTap/Networking/TCPServer.m]--[stop]**

**2011-12-05 11:03:49.421 WiTap[422:40b] [/Users/administrator/Downloads/WiTap/Networking/TCPServer.m]--[disableBonjour]**

**2011-12-05 11:03:49.422 WiTap[422:40b] about to call NetService:stop**

**2011-12-05 11:03:49.423 WiTap[422:40b] [/Users/administrator/Downloads/WiTap/AppController.m]--[openStreams]**

**2011-12-05 11:03:49.424 WiTap[422:40b] [/Users/administrator/Downloads/WiTap/AppController.m]--[stream:handleEvent:]**

**2011-12-05 11:03:49.424 WiTap[422:40b] case open completed**

**2011-12-05 11:03:49.425 WiTap[422:40b] [/Users/administrator/Downloads/WiTap/AppController.m]--[destroyPicker]**

**2011-12-05 11:03:49.426 WiTap[422:40b] [/Users/administrator/Downloads/WiTap/Picker.m]--[dealloc]**

**2011-12-05 11:03:49.426 WiTap[422:40b] [/Users/administrator/Downloads/WiTap/Networking/BrowserViewController.m]--[dealloc]**

**2011-12-05 11:03:49.427 WiTap[422:40b] [/Users/administrator/Downloads/WiTap/Networking/BrowserViewController.m]--[stopCurrentResolve]**

**2011-12-05 11:03:49.427 WiTap[422:40b] [/Users/administrator/Downloads/WiTap/Networking/BrowserViewController.m]--[setTimer:]**

**2011-12-05 11:03:49.430 WiTap[422:40b] [/Users/administrator/Downloads/WiTap/AppController.m]--[stream:handleEvent:]**

**2011-12-05 11:03:49.431 WiTap[422:40b] case open completed**

**2011-12-05 11:03:49.431 WiTap[422:40b] [/Users/administrator/Downloads/WiTap/AppController.m]--[destroyPicker]**

**2011-12-05 11:03:49.432 WiTap[422:40b] show the message game started**

**2011-12-05 11:03:49.499 WiTap[422:40b] [/Users/administrator/Downloads/WiTap/AppController.m]--[stream:handleEvent:]**

**2011-12-05 11:03:49.508 WiTap[422:40b] case there is no corresponding case match this event**

**以下过程是真机点击一个TapView的时候，模拟器执行的流程**

**2011-12-05 11:04:39.845 WiTap[422:40b] [/Users/administrator/Downloads/WiTap/AppController.m]--[stream:handleEvent:]**

**2011-12-05 11:04:39.846 WiTap[422:40b] case has bytes available**

**2011-12-05 11:04:39.847 WiTap[422:40b] [/Users/administrator/Downloads/WiTap/TapView.m]--[touchDown:]**

**2011-12-05 11:04:39.866 WiTap[422:40b] [/Users/administrator/Downloads/WiTap/AppController.m]--[stream:handleEvent:]**

**2011-12-05 11:04:39.867 WiTap[422:40b] case has bytes available**

**2011-12-05 11:04:39.868 WiTap[422:40b] [/Users/administrator/Downloads/WiTap/TapView.m]--[touchUp:]**

    呵呵，勉强地完成了这个系列，感觉对于长系列的东西我还是缺乏控制力呀，感觉讲的越到后来越不清楚了……

    (能力有限，文中可能会有不对的地方，希望大家指教。谢谢！！)

备注：最后的调试过程的显示真的是很重要，因为这个案例中所有的类既要充当服务器（接收者），又要充当客户端（发送者），所有每个方法在兼顾二者的情况下会变得挺麻烦，当只是把设备当做其中一端的话，用的代码根本不会去走，因为这些代码可能就是针对另一端而言了，但是不管怎么样，这个例子还是很成熟了，对于点对点的局域网通信，如果有哪些地方或者逻辑，或者代码的执行顺序不清楚了，可以反复的研读调试流程的代码叙述，这样就会很容易懂了。

总之而言，服务端有服务端的代码，与客户端没关系，反之，同样成立。

还有对本案例的释放，有时候我感觉还是很难把握。还是要多做。

我的想法是，在写这种p2p的例子的时候，先考虑一种情况，然后只写一种情况下的代码，之后在写另一种，最后把二者整合。