第一部分、概念的理解

1、什么是Socket？

Socket又称之为“套接字”，是系统提供的用于网络通信的方法。它的实质并不是一种协议，没有规定计算机应当怎么样传递消息，只是给程序员提供了一个发送消息的接口，程序员使用这个接口提供的方法，发送与接收消息。

Socket描述了一个IP、端口对。它简化了程序员的操作，知道对方的IP以及PORT就可以给对方发送消息，再由服务器端来处理发送的这些消息。所以，Socket一定包含了通信的双发，即客户端（Client）与服务端（server）。

2、Socket的通信过程？

每一个应用或者说服务，都有一个端口。比如DNS的53端口，http的80端口。我们能由DNS请求到查询信息，是因为DNS服务器时时刻刻都在监听53端口，当收到我们的查询请求以后，就能够返回我们想要的IP信息。所以，从程序设计上来讲，应该包含以下步骤：

1）服务端利用Socket监听端口；

2）客户端发起连接；

3）服务端返回信息，建立连接，开始通信；

4）客户端，服务端断开连接。

3、Socket双方如何建立起连接？

以下过程用代码表示：

Server端:

```
1        intport = 2000;


2        IPEndPointServerEP = new IPEndPoint(IPAddress.Any,port);


3        Socketserver = new Socket(AddressFamily.InterNetwork, SocketType.Stream,ProtocolType.Tcp);


4        server.Bind(ServerEP);
```

```
5        server.Listen(0);
```

Client端:

```
1        intport = 2000;
```

```
2        IPAddressserverip = IPAddress.Parse("192.168.1.100");
```

```
3        IPEndPointEP = new IPEndPoint(server,port);
```

```
4        Socketserver = new Socket(AddressFamily.InterNetwork, SocketType.Stream,ProtocolType.Tcp);
```

```
5        server.Bind(EP);
```

当服务器端接收到来自客户端的连接以后，需要新建一个socket来处理远端的信息。

下面一段代码应该在服务器端：

```
1        Socketclient = server.Accept();
```

以上很简单的几行代码，将在以后的网络编程中经常用到，后面还会有同步通讯、异步通讯、线程、委托与事件等等

第二部分、各协议的区别

TCP/IP SOCKET HTTP

网络七层由下往上分别为物理层、数据链路层、网络层、传输层、会话层、表示层和应用层。

其中物理层、数据链路层和网络层通常被称作媒体层，是网络工程师所研究的对象；

传输层、会话层、表示层和应用层则被称作主机层，是用户所面向和关心的内容。

http协议　对应于应用层

tcp协议　对应于传输层

ip协议　对应于网络层

三者本质上没有可比性。 何况HTTP协议是基于TCP连接的。

TCP/IP是传输层协议，主要解决数据如何在网络中传输；而HTTP是应用层协议，主要解决如何包装数据。

我们在传输数据时，可以只使用传输层（TCP/IP），但是那样的话，由于没有应用层，便无法识别数据内容，如果想要使传输的数据有意义，则必须使用应用层协议，应用层协议很多，有HTTP、FTP、TELNET等等，也可以自己定义应用层协议。WEB使用HTTP作传输层协议，以封装HTTP文本信息，然后使用TCP/IP做传输层协议将它发送到网络上。

Socket是对TCP/IP协议的封装，Socket本身并不是协议，而是一个调用接口（API），通过Socket，我们才能使用TCP/IP协议。

Http和Socket连接区别

相信不少初学手机联网开发的朋友都想知道Http与Socket连接究竟有什么区别，希望通过自己的浅显理解能对初学者有所帮助。

1、TCP连接

要想明白Socket连接，先要明白TCP连接。手机能够使用联网功能是因为手机底层实现了TCP/IP协议，可以使手机终端通过无线网络建立TCP连接。TCP协议可以对上层网络提供接口，使上层网络数据的传输建立在“无差别”的网络之上。

建立起一个TCP连接需要经过“三次握手”：

第一次握手：客户端发送syn包(syn=j)到服务器，并进入SYN_SEND状态，等待服务器确认；

第二次握手：服务器收到syn包，必须确认客户的SYN（ack=j+1），同时自己也发送一个SYN包（syn=k），即SYN+ACK包，此时服务器进入SYN_RECV状态；

第三次握手：客户端收到服务器的SYN＋ACK包，向服务器发送确认包ACK(ack=k+1)，此包发送完毕，客户端和服务器进入ESTABLISHED状态，完成三次握手。

握手过程中传送的包里不包含数据，三次握手完毕后，客户端与服务器才正式开始传送数据。理想状态下，TCP连接一旦建立，在通信双方中的任何一方主动关闭连接之前，TCP 连接都将被一直保持下去。断开连接时服务器和客户端均可以主动发起断开TCP连接的请求，断开过程需要经过“四次握手”（过程就不细写了，就是服务器和客户端交互，最终确定断开）

2、HTTP连接

HTTP协议即超文本传送协议(HypertextTransfer Protocol)，是Web联网的基础，也是手机联网常用的协议之一，HTTP协议是建立在TCP协议之上的一种应用。

HTTP连接最显著的特点是客户端发送的每次请求都需要服务器回送响应，在请求结束后，会主动释放连接。从建立连接到关闭连接的过程称为“一次连接”。

1）在HTTP 1.0中，客户端的每次请求都要求建立一次单独的连接，在处理完本次请求后，就自动释放连接。

2）在HTTP 1.1中则可以在一次连接中处理多个请求，并且多个请求可以重叠进行，不需要等待一个请求结束后再发送下一个请求。

由于HTTP在每次请求结束后都会主动释放连接，因此HTTP连接是一种“短连接”，要保持客户端程序的在线状态，需要不断地向服务器发起连接请求。通常的做法是即时不需要获得任何数据，客户端也保持每隔一段固定的时间向服务器发送一次“保持连接”的请求，服务器在收到该请求后对客户端进行回复，表明知道客户端“在线”。若服务器长时间无法收到客户端的请求，则认为客户端“下线”，若客户端长时间无法收到服务器的回复，则认为网络已经断开。

3、SOCKET原理

3.1套接字（socket）概念

套接字（socket）是通信的基石，是支持TCP/IP协议的网络通信的基本操作单元。它是网络通信过程中端点的抽象表示，包含进行网络通信必须的五种信息：连接使用的协议，本地主机的IP地址，本地进程的协议端口，远地主机的IP地址，远地进程的协议端口。

应用层通过传输层进行数据通信时，TCP会遇到同时为多个应用程序进程提供并发服务的问题。多个TCP连接或多个应用程序进程可能需要通过同一个 TCP协议端口传输数据。为了区别不同的应用程序进程和连接，许多计算机操作系统为应用程序与TCP／IP协议交互提供了套接字(Socket)接口。应用层可以和传输层通过Socket接口，区分来自不同应用程序进程或网络连接的通信，实现数据传输的并发服务。

3.2 建立socket连接

建立Socket连接至少需要一对套接字，其中一个运行于客户端，称为ClientSocket，另一个运行于服务器端，称为ServerSocket。

套接字之间的连接过程分为三个步骤：服务器监听，客户端请求，连接确认。

服务器监听：服务器端套接字并不定位具体的客户端套接字，而是处于等待连接的状态，实时监控网络状态，等待客户端的连接请求。

客户端请求：指客户端的套接字提出连接请求，要连接的目标是服务器端的套接字。为此，客户端的套接字必须首先描述它要连接的服务器的套接字，指出服务器端套接字的地址和端口号，然后就向服务器端套接字提出连接请求。

连接确认：当服务器端套接字监听到或者说接收到客户端套接字的连接请求时，就响应客户端套接字的请求，建立一个新的线程，把服务器端套接字的描述发给客户端，一旦客户端确认了此描述，双方就正式建

立连接。而服务器端套接字继续处于监听状态，继续接收其他客户端套接字的连接请求。

4、SOCKET连接与TCP连接

创建Socket连接时，可以指定使用的传输层协议，Socket可以支持不同的传输层协议（TCP或UDP），当使用TCP协议进行连接时，该Socket连接就是一个TCP连接。

5、Socket连接与HTTP连接

由于通常情况下Socket连接就是TCP连接，因此Socket连接一旦建立，通信双方即可开始相互发送数据内容，直到双方连接断开。但在实际网络应用中，客户端到服务器之间的通信往往需要穿越多个中间节点，例如路由器、网关、防火墙等，大部分防火墙默认会关闭长时间处于非活跃状态的连接而导致 Socket 连接断连，因此需要通过轮询告诉网络，该连接处于活跃状态。

而HTTP连接使用的是“请求—响应”的方式，不仅在请求时需要先建立连接，而且需要客户端向服务器发出请求后，服务器端才能回复数据。

很多情况下，需要服务器端主动向客户端推送数据，保持客户端与服务器数据的实时与同步。此时若双方建立的是Socket连接，服务器就可以直接将数据传送给客户端；若双方建立的是HTTP连接，则服务器需要等到客户端发送一次请求后才能将数据传回给客户端，因此，客户端定时向服务器端发送连接请求，不仅可以保持在线，同时也是在“询问”服务器是否有新的数据，如果有就将数据传给客户端。

HTTP连接是什么意思

HTTP是一个属于应用层的面向对象的协议，由于其简捷、快速的方式，适用于分布式超媒体信息系统。它于1990年提出，经过几年的使用与发展，得到不断地完善和扩展。目前在WWW中使用的是HTTP/1.0的第六版，HTTP/1.1的规范化工作正在进行之中，而且HTTP-NG(Next Generation of HTTP)的建议已经提出.(协议，算是全球定位！）

WWW的核心——HTTP协议

众所周知，Internet的基本协议是TCP/IP协议，目前广泛采用的FTP、Archie Gopher等是建立在TCP/IP协议之上的应用层协议，不同的协议对应着不同的应用。WWW服务器使用的主要协议是HTTP协议，即超文体传输协议。由于HTTP协议支持的服务不限于WWW，还可以是其它服务，因而HTTP协议允许用户在统一的界面下，采用不同的协议访问不同的服务，如FTP、Archie、SMTP、NNTP等。另外，HTTP协议还可用于名字服务器和分布式对象管理。

2.1 HTTP协议简介

HTTP是一个属于应用层的面向对象的协议，由于其简捷、快速的方式，适用于分布式超媒体信息系统。它于1990年提出，经过几年的使用与发展，得到不断地完善和扩展。目前在WWW中使用的是HTTP/1.0的第六版，HTTP/1.1的规范化工作正在进行之中，而且HTTP-NG(Next Generation of HTTP)的建议已经提出。

HTTP协议的主要特点可概括如下：

1.支持客户/服务器模式。

2.简单快速：客户向服务器请求服务时，只需传送请求方法和路径。请求方法常用的有GET、HEAD, POST。每种方法规定了客户与服务器联系的类型不同。 由于HTTP协议简单，使得HTTP服务器的程序规模小，因而通信速度很快。

3.灵活：HTTP允许传输任意类型的数据对象。正在传输的类型由Content-Type加以标记。

4.无连接：无连接的含义是限制每次连接只处理一个请求。服务器处理完客户的请求，并收到客户的应答后，即断开连接。采用这种方式可以节省传输时间。

5.无状态：HTTP协议是无状态协议。无状态是指协议对于事务处理没有记忆能力。缺少状态意味着如果后续处理需要前面的信息，则它必须重传，这样可能导致每次连接传送的数据量增大。另一方面，在服务器不需要先前信息时它的应答就较快。

2.2 HTTP协议的几个重要概念

1.连接(Connection)：一个传输层的实际环流，它是建立在两个相互通讯的应用程序之间。

2.消息(Message)：HTTP通讯的基本单位，包括一个结构化的八元组序列并通过连接传输。

3.请求(Request)：一个从客户端到服务器的请求信息包括应用于资源的方法、资源的标识符和协议的版本号

4.响应(Response)：一个从服务器返回的信息包括HTTP协议的版本号、请求的状态(例如“成功”或“没找到”)和文档的MIME类型。

5.资源(Resource)：由URI标识的网络数据对象或服务。

6.实体(Entity)：数据资源或来自服务资源的回映的一种特殊表示方法，它可能被包围在一个请求或响应信息中。一个实体包括实体头信息和实体的本身内容。

7.客户机(Client)：一个为发送请求目的而建立连接的应用程序。

8.用户代理(User agent)：初始化一个请求的客户机。它们是浏览器、编辑器或其它用户工具。

9.服务器(Server)：一个接受连接并对请求返回信息的应用程序。

10.源服务器(Origin server)：是一个给定资源可以在其上驻留或被创建的服务器。

11.代理(Proxy)：一个中间程序，它可以充当一个服务器，也可以充当一个客户机，为其它客户机建立请求。请求是通过可能的翻译在内部或经过传递到其它的服务器中。一个代理在发送请求信息之前，必须解释并且如果可能重写它。

代理经常作为通过防火墙的客户机端的门户，代理还可以作为一个帮助应用来通过协议处理没有被用户代理完成的请求。

12.网关(Gateway)：一个作为其它服务器中间媒介的服务器。与代理不同的是，网关接受请求就好象对被请求的资源来说它就是源服务器；发出请求的客户机并没有意识到它在同网关打交道。

网关经常作为通过防火墙的服务器端的门户，网关还可以作为一个协议翻译器以便存取那些存储在非HTTP系统中的资源。

13.通道(Tunnel)：是作为两个连接中继的中介程序。一旦激活，通道便被认为不属于HTTP通讯，尽管通道可能是被一个HTTP请求初始化的。当被中继的连接两端关闭时，通道便消失。当一个门户(Portal)必须存在或中介(Intermediary)不能解释中继的通讯时通道被经常使用。

14.缓存(Cache)：反应信息的局域存储。

2.3 HTTP协议的运作方式

HTTP协议是基于请求／响应范式的。一个客户机与服务器建立连接后，发送一个请求给服务器，请求方式的格式为，统一资源标识符、协议版本号，后边是MIME信息包括请求修饰符、客户机信息和可能的内容。服务器接到请求后，给予相应的响应信息，其格式为一个状态行包括信息的协议版本号、一个成功或错误的代码，后边是MIME信息包括服务器信息、实体信息和可能的内容。

许多HTTP通讯是由一个用户代理初始化的并且包括一个申请在源服务器上资源的请求。最简单的情况可能是在用户代理(UA)和源服务器(O)之间通过一个单独的连接来完成(见图2-1)。

当一个或多个中介出现在请求／响应链中时，情况就变得复杂一些。中介由三种：代理(Proxy)、网关(Gateway)和通道(Tunnel)。

一个代理根据URI的绝对格式来接受请求，重写全部或部分消息，通过URI的标识把已格式化过的请求发送到服务器。

网关是一个接收代理，作为一些其它服务器的上层，并且如果必须的话，可以把请求翻译给下层的服务器协议。

一个通道作为不改变消息的两个连接之间的中继点。当通讯需要通过一个中介(例如：防火墙等)或者是中介不能识别消息的内容时，通道经常被使用。图2-2

上面的图2-2表明了在用户代理(UA)和源服务器(O)之间有三个中介(A,B和C)。一个通过整个链的请求或响应消息必须经过四个连接段。这个区别是重要的，因为一些HTTP通讯选择可能应用于最近的连接、没有通道的邻居，应用于链的终点或应用于沿链的所有连接。尽管图2-2是线性的，每个参与者都可能从事多重的、并发的通讯。例如，B可能从许多客户机接收请求而不通过A，并且／或者不通过C把请求送到A，在同时它还可能处理A的请求。

任何针对不作为通道的汇聚可能为处理请求启用一个内部缓存。缓存的效果是请求／响应链被缩短，条件是沿链的参与者之一具有一个缓存的响应作用于那个请求。下图说明结果链，其条件是针对一个未被UA或A加缓存的请求，B有一个经过C来自O的一个前期响应的缓存拷贝。

图2-3

在Internet上，HTTP通讯通常发生在TCP/IP连接之上。缺省端口是TCP 80，但其它的端口也是可用的。但这并不预示着HTTP协议在Internet或其它网络的其它协议之上才能完成。HTTP只预示着一个可靠的传输。

以上简要介绍了HTTP协议的宏观运作方式，下面介绍一下HTTP协议的内部操作过程。

首先，简单介绍基于HTTP协议的客户/服务器模式的信息交换过程，如图2-4所示，它分四个过程，建立连接、发送请求信息、发送响应信息、关闭连接。

图2-4

在WWW中，“客户”与“服务器”是一个相对的概念，只存在于一个特定的连接期间，即在某个连接中的客户在另一个连接中可能作为服务器。WWW服务器运行时，一直在TCP80端口(WWW的缺省端口)监听，等待连接的出现。

下面，讨论HTTP协议下客户/服务器模式中信息交换的实现。

1.建立连接连接的建立是通过申请套接字(Socket)实现的。客户打开一个套接字并把它约束在一个端口上，如果成功，就相当于建立了一个虚拟文件。以后就可以在该虚拟文件上写数据并通过网络向外传送。

2.发送请求

打开一个连接后，客户机把请求消息送到服务器的停留端口上，完成提出请求动作。

HTTP/1.0 请求消息的格式为：

请求消息=请求行(通用信息|请求头|实体头) CRLF[实体内容]

请求 行=方法请求URL HTTP版本号 CRLF

方法=GET|HEAD|POST|扩展方法

U R L=协议名称+宿主名+目录与文件名

请求行中的方法描述指定资源中应该执行的动作，常用的方法有GET、HEAD和POST。

不同的请求对象对应GET的结果是不同的，对应关系如下：

对象 GET的结果

文件文件的内容

程序该程序的执行结果

数据库查询 查询结果

HEAD——要求服务器查找某对象的元信息，而不是对象本身。

POST——从客户机向服务器传送数据，在要求服务器和CGI做进一步处理时会用到POST方法。POST主要用于发送HTML文本中FORM的内容，让CGI程序处理。

一个请求的例子为：

GEThttp://networking.zju.edu.cn/zju/index.htm HTTP/1.0

头信息又称为元信息，即信息的信息，利用元信息可以实现有条件的请求或应答 。

请求头——告诉服务器怎样解释本次请求，主要包括用户可以接受的数据类型、压缩方法和语言等。

实体头——实体信息类型、长度、压缩方法、最后一次修改时间、数据有效期等。

实体——请求或应答对象本身。

3.发送响应

服务器在处理完客户的请求之后，要向客户机发送响应消息。

HTTP/1.0的响应消息格式如下：

响应消息=状态行(通用信息头|响应头|实体头) CRLF 〔实体内容〕

状态 行=HTTP版本号 状态码原因叙述

状态码表示响应类型

1×× 保留

2×× 表示请求成功地接收

3×× 为完成请求客户需进一步细化请求

4×× 客户错误

5×× 服务器错误

响应头的信息包括：服务程序名，通知客户请求的URL需要认证，请求的资源何时能使用。

4.关闭连接

客户和服务器双方都可以通过关闭套接字来结束TCP/IP对话

第三部分、在**IOS**里面的使用

在CFSocket中，TCP连接的创建为

```
csocket = CFSocketCreate(
                         kCFAllocatorDefault,
                         PF_INET,
                        SOCK_STREAM,
                        IPPROTO_TCP,
                         kCFSocketReadCallBack,
                        TCPServerConnectCallBack,
                        &ctx);
```

。。。。

```
sError = CFSocketConnectToAddress(csocket,address, -1);
```

这里在连接成功时回调用TCPServerConnectCallBack方法，

那么如果需要UDP传输数据的话，

```
csocket = CFSocketCreate(
                                    kCFAllocatorDefault,
                                    PF_INET,
                                   SOCK_DGRAM,
                                    IPPROTO_UDP,
                                   kCFSocketConnectCallBack,
                                   TCPServerConnectCallBack,
                                   &ctx);
```

1.TCP是有连接的，可靠的、可控制的、无边界的socket通信。

2.UDP是无连接的、不可靠的 数据报通信。但是效率高。

所以在TCP中要用回调来确定是否连接成功，而原生的socket连接成功是通过serveice发回信息来确定。UDP无连接，所以不需要回调，而不管是否发送成功。

第三部分、完整的使用方法

客户端：

导入头文件：

```
#import <sys/socket.h>

#import <netinet/in.h>

#import <arpa/inet.h>

#import <unistd.h>
```

1. 创建连接

```
CFSocketContext sockContext = {0, // 结构体的版本，必须为0

self,


// 一个任意指针的数据，可以用在创建时CFSocket对象相关联。这个指针被传递给所有的上下文中定义的回调。

NULL, // 一个定义在上面指针中的retain的回调， 可以为NULL

NULL, NULL};

CFSocketRef _socket = (kCFAllocatorDefault,// 为新对象分配内存，可以为nil

PF_INET, // 协议族，如果为0或者负数，则默认为PF_INET

SOCK_STREAM, // 套接字类型，如果协议族为PF_INET,则它会默认为SOCK_STREAM

IPPROTO_TCP, // 套接字协议，如果协议族是PF_INET且协议是0或者负数，它会默认为IPPROTO_TCP

kCFSocketConnectCallBack, // 触发回调函数的socket消息类型，具体见CallbackTypes

TCPServerConnectCallBack, // 上面情况下触发的回调函数

&sockContext // 一个持有CFSocket结构信息的对象，可以为nil

);



if (_socket != nil) {
```

```
    struct sockaddr_in addr4;   //IPV4
    memset(&addr4, 0, sizeof(addr4));
    addr4.sin_len = sizeof(addr4);
    addr4.sin_family = AF_INET;
    addr4.sin_port = htons(8888);
    addr4.sin_addr.s_addr = inet_addr([strAddress UTF8String]);  // 把字符串的地址转换为机器可识别的网络地址


    // 把sockaddr_in结构体中的地址转换为Data
    CFDataRef address = CFDataCreate(kCFAllocatorDefault, (UInt8*)&addr4, sizeof(addr4));
    CFSocketConnectToAddress(_socket, // 连接的socket
address, // CFDataRef类型的包含上面socket的远程地址的对象
-1 // 连接超时时间，如果为负，则不尝试连接，而是把连接放在后台进行，如果_socket消息类型为kCFSocketConnectCallBack，将会在连接成功或失败的时候在后台触发回调函数
);
    CFRunLoopRef cRunRef = CFRunLoopGetCurrent();    // 获取当前线程的循环
    // 创建一个循环，但并没有真正加如到循环中，需要调用CFRunLoopAddSource
    CFRunLoopSourceRef sourceRef = CFSocketCreateRunLoopSource(kCFAllocatorDefault,_socket, 0);
    CFRunLoopAddSource(cRunRef, // 运行循环
    sourceRef,  // 增加的运行循环源, 它会被retain一次
    kCFRunLoopCommonModes  // 增加的运行循环源的模式
    );
    CFRelease(courceRef);
}
```

2. 设置回调函数

```
// socket回调函数的格式：

static void TCPServerConnectCallBack(CFSocketRefsocket, CFSocketCallBackType type, CFDataRef address, const void *data, void*info) {

    if (data != NULL) {

        // 当socket为kCFSocketConnectCallBack时，失败时回调失败会返回一个错误代码指针，其他情况返回NULL

        NSLog(@"连接失败");

        return;

    }

    TCPClient *client = (TCPClient *)info;

    // 读取接收的数据

    [info performSlectorInBackground:@selector(readStream) withObject:nil];

}
```

3. 接收发送数据

```
// 读取接收的数据

- (void)readStream {

    char buffer[1024];

    NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];

    while (recv(CFSocketGetNative(_socket), //与本机关联的Socket 如果已经失效返回－1:INVALID_SOCKET

            buffer, sizeof(buffer), 0)) {

        NSLog(@"%@", [NSString stringWithUTF8String:buffer]);
```

```
    }
}
// 发送数据
- (void)sendMessage {
     NSString*stringTosend = @"你好";
    char *data = [stringTosend UTF8String];
    send(SFSocketGetNative(_socket), data, strlen(data) + 1, 0);
}
```

服务器端：

```
CFSockteRef _socket;
CFWriteStreamRef outputStream = NULL;
int setupSocket() {
    _socket = CFSocketCreate(kCFAllocatorDefault, PF_INET, SOCK_STREAM,IPPROTO_TCP,
kCFSocketAcceptCallBack, TCPServerAcceptCallBack, NULL);
    if (NULL == _socket) {
        NSLog(@"Cannot create socket!");
        return 0;
    }

    int optval = 1;
    setsockopt(CFSocketGetNative(_socket), SOL_SOCKET, SO_REUSEADDR, // 允许重用本地地址和端
口
(void *)&optval, sizeof(optval));

    struct sockaddr_in addr4;
```

```
    memset(&addr4, 0, sizeof(addr4));
    addr4.sin_len = sizeof(addr4);
    addr4.sin_family = AF_INET;
    addr4.sin_port = htons(port);
    addr4.sin_addr.s_addr = htonl(INADDR_ANY);
    CFDataRef address = CFDataCreate(kCFAllocatorDefault, (UInt8*)&addr4, sizeof(addr4));

    if (kCFSocketSuccess != CFSocketSetAddress(_socket, address)) {
        NSLog(@"Bind to address failed!");
        if (_socket)
            CFRelease(_socket);
        _socket = NULL;
        return 0;
    }

    CFRunLoopRef cfRunLoop = CFRunLoopGetCurrent();
    CFRunLoopSourceRef source = CFSocketCreateRunLoopSource(kCFAllocatorDefault,_socket, 0);
    CFRunLoopAddSource(cfRunLoop, source, kCFRunLoopCommonModes);
    CFRelease(source);

    return 1;
}
// socket回调函数，同客户端
void TCPServerAcceptCallBack(CFSocketRefsocket, CFSocketCallBackType type, CFDataRef address,
const void *data, void*info) {
```

```
    if (kCFSocketAcceptCallBack == type) {
        // 本地套接字句柄
        CFSocketNativeHandle nativeSocketHandle = *(CFSocketNativeHandle *)data;
        uint8_t name[SOCK_MAXADDRLEN];
        socklen_t nameLen = sizeof(name);
        if (0 != getpeername(nativeSocketHandle, (struct sockaddr *)name,&nameLen)) {
            NSLog(@"error");
            exit(1);
        }
        NSLog(@"%@ connected.", inet_ntoa( ((struct sockaddr_in*)name)->sin_addr )):
        CFReadStreamRef iStream;
        CFWriteStreamRef oStream;
        // 创建一个可读写的socket连接
         CFStreamCreatePairWithSocket(kCFAllocatorDefault,nativeSocketHandle, &iStream, &oStream);
        if (iStream && oStream) {
            CFStreamClientContext streamContext = {0, NULL, NULL, NULL};
            if (!CFReadStreamSetClient(iStream, kCFStreamEventHasBytesAvaiable,
                                       readStream, // 回调函数，当有可读的数据时调用
                                       &streamContext)){
                exit(1);
            }
            if (!CFReadStreamSetClient(iStream, kCFStreamEventCanAcceptBytes,writeStream,
&streamContext)){
                exit(1);
            }
```

```
            CFReadStreamScheduleWithRunLoop(iStream,
CFRunLoopGetCurrent(),kCFRunLoopCommomModes);

            CFWriteStreamScheduleWithRunLoop(wStream,
CFRunLoopGetCurrent(),kCFRunLoopCommomModes);

            CFReadStreamOpen(iStream);

            CFWriteStreamOpen(wStream);

        } else {

             close(nativeSocketHandle);

        }

    }

}

// 读取数据

void readStream(CFReadStreamRef stream,CFStreamEventType eventType, void *clientCallBackInfo) {

    UInt8 buff[255];

    CFReadStreamRead(stream, buff, 255);

    printf("received: %s", buff);

}

void writeStream (CFWriteStreamRef stream, CFStreamEventTypeeventType, void *clientCallBackInfo) {

    outputStream = stream;

}

main {

    char *str = "nihao";



    if (outputStream != NULL) {

        CFWriteStreamWrite(outputStream, str, strlen(line) + 1);
```

```
    } else {
        NSLog(@"Cannot send data!");
    }
}
// 开辟一个线程线程函数中
void runLoopInThread() {
    int res = setupSocket();
    if (!res) {
        exit(1);
    }
    CFRunLoopRun();    // 运行当前线程的CFRunLoop对象
}
```