所有内容来自：

http://www.apple.com.cn/developer/mac/library/documentation/IntelWeb/Conceptual/CFNetwork/Concepts/chapter_2_section_4.html

介绍

CFNetwork 是核心服务框架中的一个框架，它提供了一个抽象化的网络协议库。这种抽象使得进行各种网络任务都非常容易，比如：

- 使用 BSD sockets
- 利用 SSL 或者 TLS 创建加密的连接
- 解析 DNS 主机
- 涉及到 HTTP 的任务，对 HTTP 和 HTTPS 服务器进行认证
- 涉及到 FTP 服务器的任务
- 发布、解析以及浏览 Bonjour 服务

本书面向那些希望在自己的应用中使用网络协议的开发者。为了更好的理解这本书的内容，读者需要对一些网络编程方面的概念有所了解，比如 BSD sockets，流以及 HTTP 协议。另外，读者还需要熟悉 Mac OS X 编程方面的概念，包括循环运行等等。更多有关 Mac OS X 系统的内容请参考 Mac OS X 技术概述。

本文档的结构

本文档包含如下章节：

- "CFNetwork 概念" 介绍了每个 CFNetwork API 以及相互之间的互动关系。
- "与流相关的操作" 描述了如何利用 CFStream API 发送和接收网络数据。
- "与 HTTP 服务器通讯" 描述了如何发送和接收 HTTP 消息。
- "与认证 HTTP 服务器通讯" 描述了如何与安全 HTTP 服务区进行通讯。
- "操作 FTP 服务器" 描述了如何从一个 FTP 服务器那里上载和下载文件，以及如何下载字典列表。
- "使用网络诊断功能" 描述了如何在自己的应用中增加网络诊断功能。

参考

更多有关 Mac OS X 系统的网络 API，可以阅读如下文档：

- 从网络开始

下面的参考文档都是有关 CFNetwork 的：

- CFFTPStream 参考 是有关 CFFTPStream API 的参考文档。
- CFHTTPMessage 参考 是有关 CFHTTPMessage API 的参考文档。
- CFHTTPAuthentication 参考 是有关 CFHTTPAuthentication API 的参考文档。
- CFHost 参考 是有关 CFHost API 的参考文档。
- CFNetServices 参考 是有关 CFNetServices API 的参考文档。
- CFNetDiagnostics 参考 是有关 CFNetDiagnostics API 的参考文档。

除了 Apple 公司提供的文档之外，下面还有一个有关 socket 层编程的参考文档：

- UNIX 网络编程，第一卷(Stevens, Fenner 和 Rudoff)

CFNetwork 概念

CFNetwork 是一个低层次、高性能的框架，它可以让你能够灵活操纵协议栈。它是 BSD sockets 的扩展，作为一个标准 socket 抽象 API，它可以提供一些对象，使得与 FTP 和 HTTP 服务器进行通讯或者解析 DNS 主机这样的任务变得更加简单。CFNetwork is 不论从物理上还是理论上都是基于 BSD sockets。

就像 CFNetwork 依赖于 BSD sockets，有一些 Cocoa 类也是基于 CFNetwork。NSURL 就是这样的一个类，它被用来和使用标准 Internet 协议的服务器进行通讯。另外，Web Kit 是一些 Cocoa 类的集合，可以用来在窗体中显示网络内容。以上这些类都用于高层次的操作，它们内部已经实现了大部分网络协议的细节。综上所述，整个软件层次结构看起来就如图 1-1 中的图片。

图 1-1 CFNetwork 以及 Mac OS X 中的其他软件层

内容：

何时使用 CFNetwork

CFNetwork 与 BSD sockets 相比有很多优势。它集成了“运行循环”，因此如果你的应用是基于“运行循环”的，那么你可以在使用网络协议的同时不需要实现线程。CFNetwork 还包含一些对象，它们能够帮助你在使用网络协议的时候不用自己完成实现细节。比如，你可以使用 FTP 协议，而不需要实现 CFFTP API 的所有细节。如果你理解了这些网络协议，需要用到它们提供的底层控制同时又不想自己实现所有细节，那么 CFNetwork 应该是你正确的选择。

使用 CFNetwork 而不是 Cocoa 框架 NSURL 有几点好处。CFNetwork 更加专注于网络协议，而 NSURL 更加专注于数据访问，比如通过 HTTP 或者 FTP 传输数据。尽管 NSURL的确也提供了一些可配置功能，可是 CFNetwork 提供的要多得多。另外，NSURL 还需要你使用 Objective-C。如果做不到这点的话，还是应该使用 CFNetwork。有关更多基本网络框架的内容，请阅读 URL 加载系统。

在理解了 CFNetwork 与 Mac OS X 系统中其他的网络 API 的关系之后，你应该对 CFNetwork API 有一定了解，包括构成 CFNetwork 基本结构的两个 API。

CFNetwork 基本结构

在学习 CFNetwork API 之前，你必须首先理解作为 CFNetwork 基础的最主要的 API。CFNetwork 的存在依赖两个 API，这两个 API 是 Core Foundation 框架的一部分，CFSocket 和 CFStream。要使用 CFNetwork 就必须理解这些 API。

本节内容：

CFSocket API

Sockets 是网络通讯的最基本一层。一个 socket 起的作用类似与一个电话线接口，它可以使你连接到另一个 socket 上(不论是本地的还是网络另一端的)，并且向那个 socket 发送数据。

最常见的 socket 抽象概念就是 BSD sockets，而 CFSocket 则是 BSD sockets 的抽象。CFSocket 中包含了少数开销，它几乎可以提供 BSD sockets 所具有的一切功能，并且把 socket 集成进一个"运行循环"当中。CFSocket 并不仅仅限于基于流的 sockets (比如 TCP)，它可以处理任何类型的 socket。

你可以利用 `CFSocketCreate` 功能从头开始创建一个 CFSocket 对象，或者利用 `CFSocketCreateWithNative` 函数从 BSD socket 创建。然后，需要利用函数`CFSocketCreateRunLoopSource` 创建一个"运行循环"源，并利用函数 `CFRunLoopAddSource` 把它加入一个"运行循环"。这样不论 CFSocket 对象是否接收到信息， CFSocket 回调函数都可以运行。

请阅读 CFSocket 参考 中有关 CFSocket API 的更多内容。

CFStream API

对流的读写操作使我们可以以一种设备无关的方式在各种媒体之间交换数据。你可以为内存、文件或者网络（通过sockets）里面的数据创建流。另外在操作流的时候，所有数据可以分次加载。

数据流本质上是在通信通道中串行传输的一个字节序列，它是单向的，所以如果需要双向传输的话必须操作一个输入流（读操作）和一个输出流（写操作）。除了基于文件的流以外，其他流都是不可搜索的，也就是说：在流数据被提供或者接收之后，就不能再从这个流当中获取数据了。

CFStream API 用两个新的 CFType 对象提供了对这些流的一个抽象：CFReadStream 和 CFWriteStream。两个类型的流都遵循常见的核心基础 API 惯例。有关核心基础类型的更多信息，请参考设计概念。

CFStream 的构建基于 CFSocket，同时也是 CFHTTP 和 CFFTP 的基础。在图 1-2 中你可以看到，尽管 CFStream 并不是 CFNetwork的正式成员，它却是几乎所有 CFNetwork 成员的基础。

图 1-2 CFStream API 的结构

你几乎可以用操作 UNIX 文件描述符的方式对流进行读写操作。首先，实例化流对象的时候需要指定流的类型（内存、文件或者socket）并且设置任何一个可选项。然后，打开流并可以进行任意次的读写操作。当流还存在的时候，你可以通过流的属性获取有关它的信息。流属性包括有关流的任何信息，比如它的数据源或者目标，这些都不属于被读写的实际数据范畴之内。当你不再需要一个流的时候，需要关闭并把它丢弃。

CFStream 的函数如果不能进行至少一个字节数据的读写操作的话，它们可能会暂停或者阻塞当前的进程。为了避免在阻塞的时候从一个流读数据或者向一个流写数据，可以使用这些函数的异步操作版本，并且把有关这个流的操作放入一个循环当中。当可以从流中读写数据的时候，你的回调函数就会被调用。

另外，CFStream 还内置了对安全 Sockets 层 (SSL) 协议的支持。你可以建立一个包含流的 SSL 信息的字典，其中的信息包括需要的安全级别或者自签署的认证。然后把这些信息当作 `kCFStreamPropertySSLSettings` 属性传递给流，这样一个流就被转换成了一个 SSL 流。

要创建一个客户定制的 CFStream 是不可能的。比如，如果你想要对客户数据库文件当中的对象进行数据流操作，那么仅仅希望通过创建具有自己风格的 CFStream 对象是办不到这一点的，而只有通过定制 NSStream 的子类(利用 Objective-C)才可以做到。由于 NSStream 对象可以很容易的被转换为 CFStream 对象，所以你创建的 NSStream 子类可以被用在任何需要 CFStream 的地方。任何有关 NSStream 所属类的信息，请参考Cocoa 流编程指南。

CFNetwork API 概念

如果要理解 CFNetwork 框架，你必须首先对组成这个框架的每个部分都要熟悉。CFNetwork 框架被划分为独立的 API，每个 API 都负责一个特定的网络协议。可以把这些 API 结合在一起使用，也可以单独使用，这取决于你的应用需求。在不同的 API 之间，大部分程序开发惯例都是类似的，所以理解每一个 API 是非常重要的。

本节内容：

CFFTP API

与一个 FTP 服务器之间的通讯过程如果利用 CFFTP 就会变得简单一些。通过 CFFTP API，你可以创建 FTP 读操作流(用于下载)以及 FTP 写操作流(用于上载)。利用 FTP 读操作和写操作流你可以完成下列功能：

- 从一个 FTP 服务器下载一个文件
- 上载一个文件到一个 FTP 服务器
- 从 FTP 服务器上下载一个目录
- 在一个 FTP 服务器上创建目录

FTP 操作流与其他任何 CFNetwork 操作流的工作方式都很类似。比如，你可以通过调用函数 `CFReadStreamCreateWithFTPURL` 创建一个 FTP 读操作流，然后可以在任何时候调用函数 `CFReadStreamGetError` 检查这个操作流的状态。

通过设置 FTP 操作流的各种属性，你可以把这个流用于各种应用当中。例如，如果操作流连接到的服务器需要输入一个用户名和密码，你需要设置适当的属性，使操作流可以正常的工作。更多有关 FTP 操作流的各种可见属性，可以参考"对操作流进行设置"。

CFFTP 操作流可以用于同步模式或者异步模式。为了打开在 FTP 读操作流创建的时刻就指定的与 FTP 服务器的连接，需要调用函数 `CFReadStreamOpen`。如果要从该流读取数据，使用 `CFReadStreamRead` 函数，这个要用到读操作流的引用 `CFReadStreamRef`，该引用是 FTP 读操作流创建时候的返回值。`CFReadStreamRead` 函数可以把 FTP 服务器的输出填充到一个缓冲区中。

有关使用 CFFTP API 的更多内容，请参考"操作 FTP 服务器"。

CFHTTP API

如果需要发送和接收 HTTP 消息，可以使用 CFHTTP API。就像 CFFTP 是 FTP 协议的抽象一样，CFHTTP 也是 HTTP 协议的一个抽象。

超文本传输协议 (HTTP) 是在客户端和服务器之间的一种请求/相应协议。客户端创建一个请求消息后，这个会被串行化，也就是说这个消息会被转换成原始的字节流（消息在串行化以前是不能被传输的）接着请求消息被发送到服务器端。请求消息通常会要求访问一个文件，比如一个网页。服务器应答、发送回来一个字符串接下来就是一个消息。必要的时候，这个过程会重复多次。

创建一个 HTTP 请求消息，需要指定如下参数：

- 请求方法。它可以是超文本传输协议定义的多个请求方法之一，比如 `OPTIONS`、`GET`、`HEAD`、`POST`、`PUT`、`DELETE`、`TRACE` 和 `CONNECT`
- URL。比如 `http://www.apple.com`
- HTTP 的版本。比如版本 1.0 或者 1.1
- 消息头。指定消息头名称，比如 `User-Agent`，以及它的值比如 `MyUserAgent`
- 消息体

消息被创建之后，你就必须将它串行化。一个请求消息在串行化之后应该看起来是这个样子：

```
GET / HTTP/1.0\r\nUser-Agent: UserAgent\r\nContent-Length: 0\r\n\r\n
```

反串行化是串行化的相反操作。在反串行化之后，从客户端或者服务器端接收到的原始字节流就会被恢复成它的本地表现形式。CFNetwork 提供了从接收到的串行化消息中获取如下信息所需要的所有函数：消息类型（请求还是应答）、HTTP 版本、URL、消息头和消息体。

CFHTTP 的更多使用范例在"与 HTTP 服务器通讯"中可以找到。

CFHTTPAuthentication API

如果你在向一个认证服务器发送的 HTTP 请求中没有包含证书(或者带有无效证书)，服务器就会返回一个挑战认证要求(就是大家通常知道的 401 或者 407 应答)。CFHTTPAuthentication API 会把认证证书传递给 HTTP 挑战认证消息。CFHTTPAuthentication 支持下面几种认证方案：

- 基本
- 摘要
- NT LAN 管理器 (NTLM)
- 简单保护 GSS-API 协商机制 (SPNEGO)

Mac OS X v10.4 版本中新增加的一个功能就是在不同请求之间的持久性。在 Mac OS X v10.3 版本中，每当一个请求被挑战的时候，你就必须重新开始一个认证对话。而现在你可以为每一个服务器维护一组

CFHTTPAuthentication 对象。当你接到一个 401 或者 407 应答时，就找到适用于当前服务器的的对象和证书并加以使用即可。CFNetwork 会利用对象中存储的信息尽快的对请求进行处理。

通过不同请求之间的持久性，新版本的 CFHTTPAuthentication 的性能得到大大提高。更多有关如何使用 CFHTTPAuthentication 的内容可以参考"与认证 HTTP 服务器通讯"。

CFHost API

利用 CFHost API 可以获取主机信息，包括主机名、地址以及可达性等信息。这种获取有关主机的信息的过程叫做解析。

CFHost 的用法基本类似于 CFStream：

- 创建一个 CFHost 对象。
- 开始解析 CFHost 对象。
- 获取地址、主机名称或者可达性等信息。
- 在完成所有任务之后销毁 CFHost 对象。

就像所有 CFNetwork 类一样，CFHost 同时兼容于 IPv4 和 IPv6。利用 CFHost，你写的代码可以完全透明的处理 IPv4 和 IPv6 。

CFHost 与 CFNetwork 的其他部分结合的非常紧密。比如，有一个 CFStream 的函数叫做 `CFStreamCreatePairWithSocketToCFHost`，它可以直接利用 CFHost 对象创建 CFStream 对象。更多有关 CFHost 对象函数的内容，请参考 CFHost 参考。

CFNetServices API

如果你希望在自己的应用中利用 Bonjour 来注册一个服务或者发现一些服务，那么可以使用 CFNetServices API。Bonjour 是 Apple 的一个"零配置网络(ZEROCONF)"的实现，它可以帮助你发布、发现以及解析那些网络服务。

为了实现 Bonjour，CFNetServices API 中定义了三个对象类型：CFNetService、CFNetServiceBrowser 和 CFNetServiceMonitor。一个 CFNetService 对象表示一个单独的网络服务，比如一个打印机或者一个文件服务器。它包含了另一个计算机识别这个服务器所需要的所有信息，比如名称、类型、域和端口号。CFNetServiceBrowser 对象用来在域中搜索域以及网络服务。CFNetServiceMonitor 对象则被用来监控 CFNetService 对象的变化，比如在 iChat 中的状态消息。

有关 Bonjour 的完整描述，请参考 Bonjour 概述。有关使用 CFNetServices 和实现 Bonjour 的更多内容，可以参考 NSNetServices 和 CFNetServices 编程指南。

CFNetDiagnostics API

连接到网络的应用程序都需要一个稳定的连接。如果网络中断，那么应用程序就会出现问题。利用 CFNetDiagnostics API，用户可以自己诊断网络的下面这些问题：

- 物理连接错误(比如网线被拔掉)
- 网络错误(比如 DNS 或者 DHCP 服务器不再相应)
- 配置错误(比如代理设置错误)

一旦诊断到网络错误，CFNetDiagnostics 就会指导用户修复问题所在。你可能在 Safari 不能连接到一个网站的时候真正见到过 CFNetDiagnostics。CFNetDiagnostics 辅助界面如图 1-3。

图 1-3 网络诊断辅助

通过向 CFNetDiagnostics 提供有关网络错误的各种信息，你可以调用函数 `CFNetDiagnosticDiagnoseProblemInteractively` 引导用户通过提示信息找到一个解决方案。另外，你还可以利用 CFNetDiagnostics 查询网络的连接状态，并且向用户提供统一的错误信息。

如果想知道如何将 CFNetDiagnotics 集成到自己的应用程序当中，可以阅读"使用网络诊断功能"。CFNetDiagnostics 是 Mac OS X v10.4 版本中新出现的 A

与流相关的操作

本章讨论如何创建、打开流，以及如何检查读写流过程中发生的错误。另外还描述了如何从一个读操作流当中读取数据、如何向一个写操作流当中写入数据、如何在读写操作流的时候防止死锁以及如何通过一个代理服务器获取流。

本章内容：

处理读操作流

我们从创建一个读操作流开始。列表 2-1 为一个文件创建了读操作流。

列表 2-1 为文件创建读操作流

```
CFReadStreamRef myReadStream = CFReadStreamCreateWithFile(kCFAllocatorDefault, fileURL);
```

在这行代码中，`kCFAllocatorDefault` 参数指定了当前缺省的系统定位符，它用于定位操作流的内存地址，`fileURL` 指定了创建的读操作流所对应的文件名称，比如`file:///Users/joeuser/Downloads/MyApp.sit`。

一旦操作流被创建，它就可以被打开。打开一个操作流会导致这个流占用它所需要的任何系统资源，比如用于打开文件的文件描述符。列表 2-2 中的示例代码说明了如何打开读操作流。

列表 2-2 打开一个读操作流

```
if (!CFReadStreamOpen(myReadStream)) {

    CFStreamError myErr = CFReadStreamGetError(myReadStream);

    // An error has occurred.

        if (myErr.domain == kCFStreamErrorDomainPOSIX) {

        // Interpret myErr.error as a UNIX errno.

        } else if (myErr.domain == kCFStreamErrorDomainMacOSStatus) {

        // Interpret myErr.error as a MacOS error code.

            OSStatus macError = (OSStatus)myErr.error;

        // Check other error domains.

    }

}
```

如果操作流打开成功，那么 `CFReadStreamOpen` 函数返回 `TRUE`，如果因为某种原因打开失败就会返回 `FALSE`。如果 `CFReadStreamOpen` 返回 `FALSE`，示例程序调用了`CFReadStreamGetError` 函数，它返回了一个 `CFStreamError` 类型的结构，其中包含两个值：一个域代码和一个错误代码。域代码决定了错误代码将被怎样解释。比如，如果域代码是 `kCFStreamErrorDomainPOSIX`，错误代码是一个 UNIX

errno 值。其他的错误域比如是 kCFStreamErrorDomainMacOSStatus，这说明错误代码是MacErrors.h 中定义的一个 OSStatus 值，如果是 kCFStreamErrorDomainHTTP，这说明错误代码是枚举对象 CFStreamErrorHTTP 中定义的一个值。

打开一个操作流可能会耗费比较长的时间，因此 CFReadStreamOpen 和 CFWriteStreamOpen 两个函数都不会被阻塞，它们返回 TRUE 表示操作流的打开过程已经开始。如果想要检查打开过程的状态，可以调用函数 CFReadStreamGetStatus 和 CFWriteStreamGetStatus。如果返回 kCFStreamStatusOpening 说明打开过程仍然在进行中，如果返回 kCFStreamStatusOpen 说明打开过程已经完成，而返回 kCFStreamStatusErrorOccurred 说明打开过程已经完成，但是失败了。大部分情况下，打开过程是否完成并不重要，因为 CFStream 中负责读写操作的函数在操作流打开以前会被阻塞。

想要从读操作流中读取数据的话，需要调用函数 CFReadStreamRead。它类似于 UNIX 的 read() 系统调用。二者的相同之处包括：都需要缓冲区和缓冲区大小作为参数，都会返回读取的字节数，如果到了文件末尾会返回 0，如果遇到错误就会返回 -1。另外，二者都会在至少一个字节可以被读取之前被阻塞，并且如果没有遇到阻塞的情况下都会继续读取。列表 2-3 是从读操作流中获取数据的示例程序。

列表 2-3 从读操作流中获取数据 (阻塞)

```
CFIndex numBytesRead;

do {

    UInt8 buf[kReadBufSize];

    numBytesRead = CFReadStreamRead(myReadStream, buf, sizeof(buf));

    if( numBytesRead > 0 ) {

        handleBytes(buf, numBytesRead);

    } else if( numBytesRead < 0 ) {

        CFStreamError error = CFReadStreamGetError(myReadStream);

        reportError(error);

    }

} while( numBytesRead > 0 );
```

当所有数据都被读取之后，你应该调用 CFReadStreamClose 函数关闭操作流，这样可以释放与它相关的系统资源。接着通过调用函数 CFRelease 释放操作流的引用对象。你也可以通过把引用对象设置为 NULL 使它无效。请参考列表 2-4 中的示例说明。

列表 2-4 释放一个读操作流

```
CFReadStreamClose(myReadStream);

CFRelease(myReadStream);

myReadStream = NULL;
```

处理写操作流

处理写操作流的方式与处理读操作流的方式非常类似。其中一个主要的区别就是，函数 CFWriteStreamWrite 不会保证接受你所传递给它的所有数据。相反，CFWriteStreamWrite 会返回它所接受的字节数。你会留意到在列表 2-5 中，如果写入的字节数与需要写入的总字节数不相同的话，缓冲区会根据这个情况进行调整。

列表 2-5 创建、打开、写入和释放一个写操作流

```
CFWriteStreamRef myWriteStream =

        CFWriteStreamCreateWithFile(kCFAllocatorDefault, fileURL);

if (!CFWriteStreamOpen(myWriteStream)) {

    CFStreamError myErr = CFWriteStreamGetError(myWriteStream);

    // An error has occurred.

    if (myErr.domain == kCFStreamErrorDomainPOSIX) {

    // Interpret myErr.error as a UNIX errno.

    } else if (myErr.domain == kCFStreamErrorDomainMacOSStatus) {

        // Interpret myErr.error as a MacOS error code.

        OSStatus macError = (OSStatus)myErr.error;

        // Check other error domains.

    }

}

UInt8 buf[] = “Hello, world”;

UInt32 bufLen = strlen(buf);



while (!done) {

    CFTypeRef bytesWritten = CFWriteStreamWrite(myWriteStream, buf, strlen(buf));

    if (bytesWritten < 0) {

        CFStreamError error = CFWriteStreamGetError(myWriteStream);
```

```
        reportError(error);

    } else if (bytesWritten == 0) {

        if (CFWriteStreamGetStatus(myWriteStream) == kCFStreamStatusAtEnd) {

            done = TRUE;

        }

    } else if (bytesWritten != strlen(buf)) {

        // Determine how much has been written and adjust the buffer

        bufLen = bufLen - bytesWritten;

        memmove(buf, buf + bytesWritten, bufLen);



        // Figure out what went wrong with the write stream

        CFStreamError error = CFWriteStreamGetError(myWriteStream);

        reportError(error);



    }

}

CFWriteStreamClose(myWriteStream);

CFRelease(myWriteStream);

myWriteStream = NULL;
```

处理操作流的时候防止阻塞

当利用流进行通讯的时候，常常会发生数据传输耗费大量时间，尤其是进行基于 socket 的流操作时。如果流操作是同步方式，那么整个应用都不得不停下来等待数据传输完成。因此，强烈建议您的代码中利用其它方法防止阻塞。

在读写 CFStream 对象的时候，有两种方法防止阻塞：

- 轮询 — 在进行读操作的时候，在从流中读取数据以前检查是否有字节可读。在进行写操作的时候，在写入流之前检查数据是否可以无阻塞的写入。
- 利用一个循环 — 注册接收一个与流相关的事件，并把流放入一个循环当中。当与流相关的事件发生时，你的回调函数(由注册函数指定) 就会被调用。

下面几节将一一介绍这些方法。

本节内容：

利用轮询防止阻塞

轮询的时候，需要确定读写操作流的状态是否就绪。在写入一个写操作流的时候，这是通过调用函数 `CFWriteStreamCanAcceptBytes` 来实现的。如果返回 `TRUE`，那么你就可以利用 `CFWriteStreamWrite` 函数，因为它可以马上进行写入操作而不会被阻塞。类似的，对于一个读操作流，在调用 `CFReadStreamRead` 函数之前，可以调用函数`CFReadStreamHasBytesAvailable`。通过这种流轮询方式，你可以避免为了等待操作流就绪而阻塞整个线程。

列表 2-6 是针对读操作流的一个轮询范例。

列表 2-6 轮询一个读操作流

```
while (!done) {

    if (CFReadStreamHasBytesAvailable(myReadStream)) {

        UInt8 buf[BUFSIZE];

        CFIndex bytesRead = CFReadStreamRead(myReadStream, buf, BUFSIZE);

        if (bytesRead < 0) {

            CFStreamError error = CFReadStreamGetError(myReadStream);

            reportError(error);

        } else if (bytesRead == 0) {

            if (CFReadStreamGetStatus(myReadStream) == kCFStreamStatusAtEnd) {

                done = TRUE;

            }

        } else {

            handleBytes(buf, bytesRead);

        }
```

```
    } else {

        // ...do something else while you wait...

    }

}
```

列表 2-7 是一个写操作流的轮询范例。

列表 2-7 轮询一个写操作流

```
UInt8 buf[] = “Hello, world”;

UInt32 bufLen = strlen(buf);



while (!done) {

    if (CFWriteStreamCanAcceptBytes(myWriteStream)) {

        int bytesWritten = CFWriteStreamWrite(myWriteStream, buf, strlen(buf));

        if (bytesWritten < 0) {

            CFStreamError error = CFWriteStreamGetError(myWriteStream);

            reportError(error);

        } else if (bytesWritten == 0) {

            if (CFWriteStreamGetStatus(myWriteStream) == kCFStreamStatusAtEnd)

            {

                done = TRUE;

            }

        } else if (bytesWritten != strlen(buf)) {

            // Determine how much has been written and adjust the buffer

            bufLen = bufLen - bytesWritten;

            memmove(buf, buf + bytesWritten, bufLen);



            // Figure out what went wrong with the write stream

            CFStreamError error = CFWriteStreamGetError(myWriteStream);

            reportError(error);

        }

    } else {

        // ...do something else while you wait...

    }

}
```

利用循环防止阻塞

线程的循环会监控某些事件的发生。当这些事件发生的时候，循环会调用某个特定的函数。循环会一直监测它的输入源的事件。在进行网络传输的时候，当你注册的事件发生时，你的回调函数会被循环所执行。这使你不必轮询你的 socket 操作流，这种轮询会降低线程执行效率。如果对循环不是很熟悉，可以阅读循环中的内容。

本例以创建一个 socket 读操作流开始：

```
CFStreamCreatePairWithSocketToCFHost(kCFAllocatorDefault, host, port,

                                     &myReadStream, NULL);
```

其中 CFHost 引用对象 `host` 确定了读操作流指向的远程主机，`port` 参数确定了这个主机所使用的端口号码。`CFStreamCreatePairWithSocketToCFHost` 函数返回了新的读操作流引用 `myReadStream`。最后一个参数 `NULL` 确定调用者不想创建一个写操作流。如果你想要创建一个写操作流，那么最后一个参数应该是，比如 `&myWriteStream` 这样的形式。

在打开 socket 读操作流之前，需要创建一个上下文对象，这个对象在你注册接收流相关的事件时会用到：

```
CFStreamClientContext myContext = {0, myPtr, myRetain, myRelease, myCopyDesc};
```

第一个参数为 `0` 表示版本号码。`info` 参数，也就是 `myPtr`，它指向你想要传递给回调函数的数据的指针。通常情况下，`myPtr` 是一个指向你所定义的数据结构的指针，这个结构中包含了与这个流相关的指针。`retain` 参数指向一个能够保存 `info` 参数的函数，因此如果你象上面的代码那样将这个参数设置为 `myRetain`， CFStream 会调用`myRetain(myPtr)` 函数来保留 `info` 指针。类似的，`release` 参数 `myRelease` 指向一个释放 info 参数的函数。当操作流与上下文分离开时，CFStream 可以调用`myRelease(myPtr)`。最后，`copyDescription` 参数指向一个函数，这个函数能够提供对操作流的描述。比如，如果你需要利用上面的操作流客户端上下文调用`CFCopyDesc(myReadStream)` 函数，CFStream 就会调用 `myCopyDesc(myPtr)`。

客户端上下文还允许你将 `retain`、`release` 和 `copyDescription` 等参数设置为 `NULL`。如果你将 `retain` 和 `release` 参数设置为 `NULL`，那么系统就会认为你会一直保存`info` 指针直到操作流本身被销毁。如果你设置 `copyDescription` 参数为 `NULL`，那么如果需要的话，系统将会提供一个 `info` 指针指向的内存的基本描述信息。

在客户端上下文设置好之后，可以调用函数 `CFReadStreamSetClient` 登记接收与操作流相关的事件。`CFReadStreamSetClient` 要求你指定一个回调函数，以及想要接收的事件。下面列表 2-8 的范例中，回调函数需要接收 `kCFStreamEventHasBytesAvailable`，`kCFStreamEventErrorOccurred` 以及 `kCFStreamEventEndEncountered` 事件。然后利用函数 `CFReadStreamScheduleWithRunLoop` 在循环中调度这个操作流。列表 2-8 中的范例说明了如何进行这个操作。

列表 2-8 在循环中调度一个操作流

```
CFOptionsFlags registeredEvents = kCFStreamEventHasBytesAvailable |

        kCFStreamEventErrorOccurred | kCFStreamEventEndEncountered;

if (CFReadStreamSetClient(myReadStream, registeredEvents, myCallBack, &myContext)

{

    CFReadStreamScheduleWithRunLoop(myReadStream, CFRunLoopGetCurrent(),

                                    kCFRunLoopCommonModes);

}
```

当已经在循环中调度了操作流之后，你就可以象列表 2-9 中那样打开操作流了。

列表 2-9 打开一个非阻塞的读操作流

```
if (!CFReadStreamOpen(myReadStream)) {

    CFStreamError myErr = CFReadStreamGetError(myReadStream);

    if (myErr.error != 0) {

    // An error has occurred.

        if (myErr.domain == kCFStreamErrorDomainPOSIX) {

        // Interpret myErr.error as a UNIX errno.

            strerror(myErr.error);

        } else if (myErr.domain == kCFStreamErrorDomainMacOSStatus) {

            OSStatus macError = (OSStatus)myErr.error;

            }

        // Check other domains.

    } else

        // start the run loop

        CFRunLoopRun();

}
```

现在，只需要等待回调函数被执行到了。在你的回调函数中，要检查事件代码并采取相应的措施。参见列表 2-10。

列表 2-10 网络事件回调函数

```
void myCallBack (CFReadStreamRef stream, CFStreamEventType event, void *myPtr) {

    switch(event) {

        case kCFStreamEventHasBytesAvailable:

            // It is safe to call CFReadStreamRead; it won't block because bytes

            // are available.

            UInt8 buf[BUFSIZE];

            CFIndex bytesRead = CFReadStreamRead(stream, buf, BUFSIZE);

            if (bytesRead > 0) {

                handleBytes(buf, bytesRead);

            }

            // It is safe to ignore a value of bytesRead that is less than or

            // equal to zero because these cases will generate other events.

            break;

        case kCFStreamEventErrorOccurred:

            CFStreamError error = CFReadStreamGetError(stream);

            reportError(error);

            CFReadStreamClose(stream);

            CFRelease(stream);

            break;
```

```
        case kCFStreamEventEndEncountered:

            reportCompletion();

            CFReadStreamClose(stream);

            CFRelease(stream);

            break;

    }

}
```

当回调函数接收到 kCFStreamEventHasBytesAvailable 事件代码时，它会调用 CFReadStreamRead 读取数据。

当回调函数接收到 kCFStreamEventErrorOccurred 事件代码时，它会调用 CFReadStreamGetError 来获取错误信息以及它自己的错误处理函数 (reportError) 来处理这个错误.

当回调函数接收到 kCFStreamEventEndEncountered 事件代码时，它会调用它自己的函数(reportCompletion) 处理数据结尾，然后调用函数 CFReadStreamClose 关闭操作流和 CFRelease 函数释放操作流引用。调用 CFReadStreamClose 会导致操作流被取消调度，因此回调函数不需要从循环中删除操作流。

利用防火墙

有两种方式设置操作流的防火墙。对大多数流来说，可以通过 SCDynamicStoreCopyProxies 函数获取代理设置，然后通过设置 kCFStreamHTTPProxy (或者kCFStreamFTPProxy) 参数将结果设置到操作流上。SCDynamicStoreCopyProxies 函数是系统配置框架的一部分，因此需要在项目中包含<SystemConfiguration/SystemConfiguration.h>，这样才能使用该函数。接着在使用之后把代理词典引用释放调。这个过程看起来类似列表 2-11 中的例子。

列表 2-11 通过代理服务器浏览一个操作流

```
CFDictionaryRef proxyDict = SCDynamicStoreCopyProxies(NULL);

CFReadStreamSetProperty(readStream, kCFStreamPropertyHTTPProxy, proxyDict);
```

不过，如果你需要在多个操作流中使用代理设置的话，就会稍微复杂一点。这种情况下，获取用户机器的防火墙设置需要5个步骤:

1. 创建一个指向动态存储会话的单独持久性句柄，比如 SCDynamicStoreRef。
2. 把指向动态存储会话的句柄放入循环当中，使它能够得到代理变化的通知。
3. 利用 SCDynamicStoreCopyProxies 获取最新的代理设置。
4. 在得到变动通知后更新代理的拷贝。
5. 在使用完毕后清除 SCDynamicStoreRef 。

可以使用函数 SCDynamicStoreCreate 来创建一个指向动态存储会话的句柄，并向它传递一个定位器、一个描述过程的名称、一个回调函数和一个动态存储上下文SCDynamicStoreContext。在你的应用进行初始化的时候就会执行这个步骤。具体代码类似列表 2-12。

列表 2-12 创建一个指向动态存储会话的句柄

```
SCDynamicStoreContext context = {0, self, NULL, NULL, NULL};

systemDynamicStore = SCDynamicStoreCreate(NULL,

                                          CFSTR("SampleApp"),

                                          proxyHasChanged,

                                          &context);
```

在创建了指向动态存储的引用之后，你需要把它加入循环当中。首先，对这个动态存储引用进行设置，使它能够监控代理的任何变化。这个可以通过函数SCDynamicStoreKeyCreateProxies 和 SCDynamicStoreSetNotificationKeys 来实现。接着，你可以利用函数 SCDynamicStoreCreateRunLoopSource 和CFRunLoopAddSource 把动态存储引用加入到循环当中。你的代码应该类似列表 2-13。

列表 2-13 把一个动态存储引用加入循环

```
// Set up the store to monitor any changes to the proxies

CFStringRef proxiesKey = SCDynamicStoreKeyCreateProxies(NULL);

CFArrayRef keyArray = CFArrayCreate(NULL,

                                    (const void **)(&proxiesKey),

                                    1,

                                    &kCFTypeArrayCallBacks);

SCDynamicStoreSetNotificationKeys(systemDynamicStore, keyArray, NULL);

CFRelease(keyArray);

CFRelease(proxiesKey);



// Add the dynamic store to the run loop

CFRunLoopSourceRef storeRLSource =

    SCDynamicStoreCreateRunLoopSource(NULL, systemDynamicStore, 0);

CFRunLoopAddSource(CFRunLoopGetCurrent(), storeRLSource, kCFRunLoopCommonModes);

CFRelease(storeRLSource);
```

一旦动态存储引用被加入了循环，可以利用它调用函数 SCDynamicStoreCopyProxies 加载代理词典的当前代理设置，如何做请参考列表 2-14 。

列表 2-14 加载代理词典

```
gProxyDict = SCDynamicStoreCopyProxies(systemDynamicStore);
```

一旦把动态存储引用加入了循环当中，每当代理发生变化，你的回调函数就会被调用。当前的代理词典会被释放掉，并利用新的代理设置重新载入这个代理词典。列表 2-15 中是一个回调函数的范例。

列表 2-15 代理回调函数

```
void proxyChanged() {

    CFRelease(gProxyDict);

    gProxyDict = SCDynamicStoreCopyProxies(systemDynamicStore);

}
```

由于所有的代理信息都是最新的，这样就可以使用这些代理。在创建了您的读写操作流之后， 调用函数 `CFReadStreamSetProperty` 或者 `CFWriteStreamSetProperty` 设置 `kCFStreamPropertyHTTPProxy` 代理。如果你的操作流是一个读操作流，名称为 `readStream`，那么你的函数调用将如列表 2-16 所示。

列表 2-16 把代理信息加入一个操作流

```
CFReadStreamSetProperty(readStream, kCFStreamPropertyHTTPProxy, gProxyDict);
```

在使用完毕代理设置信息之后，请确保释放了词典和动态存储引用，并将动态存储引用从循环中清除。请参考列表 2-17。

列表 2-17 清理代理信息

```
if (gProxyDict) {

    CFRelease(gProxyDict);

}



// Invalidate the dynamic store's run loop source

// to get the store out of the run loop

CFRunLoopSourceRef rls = SCDynamicStoreCreateRunLoopSource(NULL, systemDynamicStore, 0);

CFRunLoopSourceInvalidate(rls);

CFRelease(rls);

CFRelease(systemDynmaicStore);
```

与 HTTP 服务器通讯

本章介绍了如何创建、发送以及接收 HTTP 请求和响应。

内容：

创建一个 CFHTTP 请求

一个 HTTP 请求就是一个消息，其中包含了一个能够被远程服务器调用的方法、能够操作的对象(URL)、消息头以及消息体。被调用的方法通常是下面这些：`GET`, `HEAD`, `PUT`,`POST`, `DELETE`, `TRACE`, `CONNECT` 或者`OPTIONS`。 用 CFHTTP 创建一个 HTTP 请求需要经过如下四个步骤：

1. 利用 `CFHTTPMessageCreateRequest` 函数创建一个 CFHTTP 消息对象。
2. 利用函数 `CFHTTPMessageSetBody` 设置消息体。
3. 利用函数 `CFHTTPMessageSetHeaderFieldValue` 设置消息头。
4. 通过调用函数 `CFHTTPMessageCopySerializedMessage` 将消息串行化。

范例代码看起来将类似于表 3-1 中的代码。

表 3-1 创建一个 HTTP 请求

```
CFStringRef url = CFSTR("http://www.apple.com");

CFURLRef myURL = CFURLCreateWithString(kCFAllocatorDefault, url, NULL);

requestMethod = CFSTR("GET");

CFHTTPMessageRef myRequest =

    CFHTTPMessageCreateRequest(kCFAllocatorDefault, requestMethod, myURL,

                               kCFHTTPVersion1_1);

CFHTTPMessageSetBody(myRequest, bodyData);

CFHTTPMessageSetHeaderFieldValue(myRequest, headerField, value);

CFDataRef mySerializedRequest = CFHTTPMessageCopySerializedMessage(myRequest);
```

在这段范例代码中，`url` 是第一个通过调用函数 `CFURLCreateWithString` 被转换为 CFURL 对象的。接下来调用函数 `CFHTTPMessageCreateRequest`， 它有四个参数：`kCFAllocatorDefault` 这个参数

说明，创建消息引用的时候用到了缺省的系统内存定位器，requestMethod 指定了方法，比如 POST 方法，myURL 指定了 URL，比如http://www.apple.com，而 kCFHTTPVersion1_1 确定了消息的 HTTP 版本是 1.1。CFHTTPMessageCreateRequest 所返回的消息对象引用 (myRequest) 接下来会和消息体 (bodyData)一同被传递给 CFHTTPMessageSetBody 。然后需要利用同一个消息对象引用、消息头的名称 (headerField)以及需要被设置的值 (value) 一起来调用CFHTTPMessageSetHeaderFieldValue 。表示消息头的参数是一个 CFString 对象，比如 Content-Length，表示值的参数是一个 CFString 对象，比如 1260。最后，通过调用 CFHTTPMessageCopySerializedMessage 函数将消息串行化处理，然后通过一个写流发送给预定的接收者，这个接收者在本例中就是 http://www.apple.com。

一旦不再需要这个消息，请释放消息对象以及串行化后的消息。这个步骤请参考表 3-2 中的范例代码。

表 3-2 释放一个 HTTP 请求

```
CFRelease(myRequest);

CFRelease(myURL);

CFRelease(url);

CFRelease(mySerializedRequest);

myRequest = NULL;

mySerializedRequest = NULL;
```

创建一个 CFHTTP 响应

创建一个 HTTP 响应的步骤几乎与创建一个 HTTP 请求的步骤完全相同。唯一不同的地方是，调用的函数不是 CFHTTPMessageCreateRequest，而是CFHTTPMessageCreateResponse，不过所用参数都是相同的。

反串行化一个发送过来的 HTTP 请求

要反串行化一个发送过来的 HTTP 请求，需要首先利用函数 CFHTTPMessageCreateEmpty 创建一个空消息，将 isRequest 参数设置为 TRUE 表示即将创建一个空的请求消息。然后利用函数 CFHTTPMessageAppendBytes 将发送过来的消息扩展到新创建的空消息上。函数 CFHTTPMessageAppendBytes 可以对消息进行反串行化并且清除其中可能包含的所有控制信息。重复进行这个操作，直到函数 CFHTTPMessageIsHeaderComplete 返回 TRUE。如果你不检查函数 CFHTTPMessageIsHeaderComplete 是否返回TRUE 的话，那么得到的消息可能是不完整、不可靠的。利用这两个函数的范例代码在表 3-3 中。

表 3-3 反串行化一个消息

```
CFHTTPMessageRef myMessage = CFHTTPMessageCreateEmpty(kCFAllocatorDefault, TRUE);

if (!CFHTTPMessageAppendBytes(myMessage, &data, numBytes) {

    //Handle parsing error

}
```

在这个例子中，data 就是即将被扩展到空消息上面的数据，而 numBytes 是 data 的长度。这时你也可以调用函数 CFHTTPMessageIsHeaderComplete 来验证扩展之后得到的消息头是完整的。

```
if (CFHTTPMessageIsHeaderComplete(myMessage) {

    // Perform processing.

}
```

在消息被反串行化之后，你可以调用下面任何一个函数来从消息当中获取信息：

- CFHTTPMessageCopyBody 可以获取消息体的一个拷贝
- CFHTTPMessageCopyHeaderFieldValue 可以获取消息头中指定域的值的拷贝
- CFHTTPMessageCopyAllHeaderFields 可以获取消息头中所有的域的拷贝
- CFHTTPMessageCopyRequestURL 可以获取消息的 URL 的拷贝
- CFHTTPMessageCopyRequestMethod 可以获取消息的请求方法的拷贝

当你不在需要这个消息时，请恰当的释放并销毁对象。

反串行化一个发送过来的 HTTP 响应

就像创建一个 HTTP 请求非常类似于创建一个 HTTP 响应一样，反串行化一个发送过来的 HTTP 请求也非常类似于反串行化一个发送过来的 HTTP 响应。唯一的重要区别就是，在调用 CFHTTPMessageCreateEmpty 函数的时候，你必须将 isRequest 设置为 FALSE，表示即将创建的消息是一个响应消息。

利用一个读操作流把 HTTP 请求串行化并发送出去

可以利用一个 CFReadStream 对象将 CFHTTP 请求串行化并且发送出去。在利用一个 CFReadStream 对象发送 CFHTTP 请求的时候，对流进行打开操作就会使消息被串行化并且马上发送出去。利用 CFReadStream 对象发送 CFHTTP 请求的话，获取请求对应的应答就会比较容易，因为可以直接从流的属性中获取这个应答。

本节内容：

将一个 HTTP 请求串行化并发送出去

利用一个 CFReadStream 对象把一个 HTTP 请求串行化并发送出去，需要首先创建一个 CFHTTP 请求，然后按照"创建一个 CFHTTP 请求"中描述的那样设置消息体和消息头。然后调用函数 CFReadStreamCreateForHTTPRequest 创建一个 CFReadStream 对象，并把刚刚创建的请求传递过去。最后用 CFReadStreamOpen 打开读操作流。

在函数 CFReadStreamCreateForHTTPRequest 被调用之后，它会生成刚刚传递过来的 CFReadStream 对象的一个拷贝。因此，如果必要的话，你可以在调用CFReadStreamCreateForHTTPRequest 之后

马上释放 CFReadStream 对象。

由于读操作流会在 CFHTTP 请求被创建的时候同服务器建立一个 socket 连接（服务器的地址由 `myUrl` 参数指定），因此必须过一段时间我们才可以认为流已经真正被打开了。而在打开读操作流的同时，请求就会被串行化并发送出去。

有关如何串行化和发送一个 HTTP 请求的代码范例在表 3-4 中。.

表 3-4 用一个读操作流串行化一个 HTTP 请求

```
CFHTTPMessageRef myRequest = CFHTTPMessageCreateRequest(kCFAllocatorDefault,

        requestMethod, myUrl, kCFHTTPVersion1_1);

CFHTTPMessageSetBody(myRequest, bodyData);

CFHTTPMessageSetHeaderFieldValue(myRequest, headerField, value);



CFReadStreamRef myReadStream = CFReadStreamCreateForHTTPRequest(kCFAllocatorDefault, myRequest);



CFReadStreamOpen(myReadStream);
```

检查应答

调用函数 `CFReadStreamCopyProperty` 从读操作流当中获取消息应答：

```
CFHTTPMessageRef myResponse = CFReadStreamCopyProperty(myReadStream, kCFStreamPropertyHTTPResponseHeader);
```

你可以通过调用函数 `CFHTTPMessageCopyResponseStatusLine` 从应答消息当中获取完整的状态行：

```
CFStringRef myStatusLine = CFHTTPMessageCopyResponseStatusLine(myResponse);
```

或者通过调用函数 `CFHTTPMessageGetResponseStatusCode` 从应答消息当中仅仅获取状态代码：

```
UInt32 myErrCode = CFHTTPMessageGetResponseStatusCode(myResponse);
```

处理认证错误

如果函数 `CFHTTPMessageGetResponseStatusCode` 所返回的状态代码是 401 (表示远程服务器要求认证信息) 或者 407 (表示一个代理服务器要求认证)，那么你需要将认证信息附加在请求之后，然后重新发送。有关如何处理认证相关的信息，请参考"与需要认证的 HTTP 服务器进行通讯"。

处理重定向错误

在函数 `CFReadStreamCreateForHTTPRequest` 创建了一个读操作流之后，缺省情况下是禁止对这个流进行自动重定向的。如果请求发送目标地址的统一资源定位符，也就是 URL，被重定向到另一个 URL 的话，那么发送这个请求会得到一个错误，状态代码在 300 到 307 之间。如果你接收到一个重定向错误的话，你需要关闭并重新创建这个流，使能它的自动重定向设置，然后打开这个流。参见表 3-5。

表 3-5 重定向一个 HTTP 流

```
CFReadStreamClose(myReadStream);

CFReadStreamRef myReadStream =

    CFReadStreamCreateForHTTPRequest(kCFAllocatorDefault, myRequest);

if (CFReadStreamSetProperty(myReadStream, kCFStreamPropertyHTTPShouldAutoredirect, kCFBooleanTrue) == false) {

    // something went wrong, exit

}

CFReadStreamOpen(myReadStream);
```

也许每次当你创建一个读操作流的时候，都想要使能它的自动重定向设置。

与认证 HTTP 服务器通讯

本章介绍了如何利用 CFHTTPAuthentication API 与认证 HTTP 服务器进行通讯。并解释了如何找到匹配的认证对象与证书、把它们用于一个 HTTP 请求，以及存储起来以后使用。

通常来说，如果在你的 HTTP 请求之后 HTTP 服务器返回一个 401 或者 407 响应，这就说明这个服务器市认证服务器，需要证书。在 CFHTTPAuthentication API 中，每套证书都存储在一个 CFHTTPAuthentication 对象中。不过，每个不同的认证服务器和连接到这个服务器上的每个不同的用户都需要一个单独的 CFHTTPAuthentication 对象。如果要和这些服务器通讯，你需要在进行 HTTP 请求的时候用到 CFHTTPAuthentication 对象，下面我们会详细介绍这个过程。

本章内容：

处理认证

在你的应用中增加对认证的支持，可以使它与认证 HTTP 服务器交互(如果服务器返回 401 或者 407 应答的话)。尽管 HTTP 认证并不是一个很难的概念，它的处理过程还是比较复杂的。具体过程如下：

1. 客户端向服务器发送一个 HTTP 请求。
2. 服务器向客户端返回一个挑战。
3. 客户端将最开始发送的请求与证书捆绑并重新发送给服务器。
4. 在客户端和服务器端有一个谈判过程。
5. 当服务器完成对客户端的认证之后，它把应答返回给客户端。

直行这个过程需要几个步骤，在图 4-1 和图 4-2 中可以看到完整的工作流程。

图 4-1 处理认证

图 4-2 找到一个认证对象

如果一个 HTTP 请求得到的应答是 401 或者 407，那么客户端首先要做的就是找到一个合法的 CFHTTPAuthentication 对象。一个认证对象中包含了证书以及在一个 HTTP 请求消息中替服务器验证您的身份的其他信息。一旦您已经在这个服务器上获得了授权，您就会获得一个合法的认证对象。不过，多数情况下，您需要通过`CFHTTPAuthenticationCreateFromResponse` 函数根据响应创建该对象。具体内容参考列表 4-1。

注意： 有关认证的所有代码实例都来自 ImageClient 应用。

列表 4-1 创建一个认证对象

```
if (!authentication) {

    CFHTTPMessageRef responseHeader =

        (CFHTTPMessageRef) CFReadStreamCopyProperty(

            readStream,

            kCFStreamPropertyHTTPResponseHeader

        );



    // Get the authentication information from the response.

    authentication = CFHTTPAuthenticationCreateFromResponse(NULL, responseHeader);

    CFRelease(responseHeader);

}
```

如果新的认证对象是合法的，那么就可以进入 图 4-1 的下一个步骤。而如果认证对象是不合法的，那么就丢弃认证对象和证书，并检查证书是否无效。更多有关证书的内容，可以参考"安全证书"。

无效的证书意味着服务器不能接受当前的登录请求信息，并且会继续等待新的证书。而如果证书是有效的，同时服务器还是拒绝您的请求，那说明服务区拒绝于您交互，这个时候就应该放弃。我们假定证书是无效的，那么需要重新从创建认证对象开始进行整个过程，直到获得了可以使用的证书和一个合法的认证对象。这个过程可以用类似列表 4-2 的代码表示。

列表 4-2 找到一个合法的认证对象

```
CFStreamError err;

if (!authentication) {

    // the newly created authentication object is bad, must return

    return;



} else if (!CFHTTPAuthenticationIsValid(authentication, &err)) {



    // destroy authentication and credentials

    if (credentials) {

        CFRelease(credentials);

        credentials = NULL;

    }

    CFRelease(authentication);

    authentication = NULL;



    // check for bad credentials (to be treated separately)

    if (err.domain == kCFStreamErrorDomainHTTP &&

        (err.error == kCFStreamErrorHTTPAuthenticationBadUserName

        || err.error == kCFStreamErrorHTTPAuthenticationBadPassword))

    {

        retryAuthorizationFailure(&authentication);

        return;

    } else {

        errorOccurredLoadingImage(err);

    }
```

```
}
```

现在您已经有了一个合法的认证对象，下一步就是继续按照 图表 4-1 进行处理。首先，确定你在什么地方需要证书。如果不需要，那么就将这个认证对象应用于 HTTP 请求当中。在列表 4-4 (`resumeWithCredentials`)里，认证对象就被应用于 HTTP 请求中。

在不保存证书的条件下(这点在"在内存中保存证书" 和 "在永久存储中保存证书"中有所解释)，获取合法证书的唯一方式就是提示用户输入。大部分时间里，证书都需要用户的名称和密码。通过向函数 `CFHTTPAuthenticationRequiresUserNameAndPassword` 传递认证对象，你可以看出是否需要用户名和用户密码。如果证书的确需要用户名和密码，那么就需要提示用户输入这些信息，并保存在证书词典当中。对于 NTLM 服务器来说，证书还需要域信息。在有了新的证书之后，你可以利用列表 4-4中的`resumeWithCredentials` 函数将认证对象应用于 HTTP 请求。整个过程在列表 4-3 中说明。

注意： 在代码中，如果注释行前后都有椭圆，这说明该行为超出了本文档范围，但是需要实现。这与普通注释说明当前正在进行地行为是不同的。

列表 4-3 寻找证书(如果需要的话)并应用

```
// ...continued from
Listing 4-2

else {

    cancelLoad();

    if (credentials) {

        resumeWithCredentials();

    }

    // are a user name & password needed?

    else if (CFHTTPAuthenticationRequiresUserNameAndPassword(authentication))

        {

        CFStringRef realm = NULL;

        CFURLRef url = CFHTTPMessageCopyRequestURL(request);



         // check if you need an account domain so you can display it if necessary

        if (!CFHTTPAuthenticationRequiresAccountDomain(authentication)) {

            realm = CFHTTPAuthenticationCopyRealm(authentication);

        }

        // ...prompt user for user name (user), password (pass)

        // and if necessary domain (domain) to give to the server...



        // Guarantee values

        if (!user) user = (CFStringRef)@"";

        if (!pass) pass = (CFStringRef)@"";



        CFDictionarySetValue(credentials, kCFHTTPAuthenticationUsername, user);

        CFDictionarySetValue(credentials, kCFHTTPAuthenticationPassword, pass);



        // Is an account domain needed? (used currently for NTLM only)

        if (CFHTTPAuthenticationRequiresAccountDomain(authentication)) {

            if (!domain) domain = (CFStringRef)@"";

            CFDictionarySetValue(credentials,

                                 kCFHTTPAuthenticationAccountDomain, domain);

        }

        if (realm) CFRelease(realm);

        CFRelease(url);

    }

    else {

        resumeWithCredentials();

    }

}
```

列表 4-4 将认证对象应用于一个请求

```
void resumeWithCredentials() {

    // Apply whatever credentials we've built up to the old request

    if (!CFHTTPMessageApplyCredentialDictionary(request, authentication,

                                                credentials, NULL)) {

        errorOccurredLoadingImage();

    } else {

        // Now that we've updated our request, retry the load

        loadRequest();

    }

}
```

在内存中保存证书

如果你希望与一个认证服务器保持频繁的通讯，那么也许应该对证书进行重复使用以避免多次提示用户输入服务器的用户名和密码。本节介绍了如果对支持单次认证代码(比如"处理认证"中所述)进行修改，使它可以在内存中保存证书供以后重复使用。

为了重复使用证书，需要对你的代码中的三个数据结构进行修改。

1. 创建一个可变数组来保存所有认证对象。

```
2.  CFMutableArrayRef authArray;
```

3. 而不是这样：

```
4.  CFHTTPAuthenticationRef authentication;
```

5.
6. 利用一个数据词典，创建认证对象到证书之间的映射关系。

```
7.  CFMutableDictionaryRef credentialsDict;
```

8. 而不是这样：

```
9.  CFMutableDictionaryRef credentials;
```

10.
11. 在任何利用该数据结构修改当前认证对象和当前证书的地方，维护这个数据结构。

```
12. CFDictionaryRemoveValue(credentialsDict, authentication);
```

13. 而不是这样：

```
14. CFRelease(credentials);
```

15.

现在，在创建了一个 HTTP 请求之后，在每次载人以前都要寻找到一个相匹配的认证对象。列表 4-5中是一个简单没有经过优化的方法，说明了如何找到合适的对象。

列表 4-5 寻找一个匹配的认证对象

```
CFHTTPAuthenticationRef findAuthenticationForRequest {

    int i, c = CFArrayGetCount(authArray);

    for (i = 0; i < c; i ++) {

        CFHTTPAuthenticationRef auth = (CFHTTPAuthenticationRef)

                CFArrayGetValueAtIndex(authArray, i);

        if (CFHTTPAuthenticationAppliesToRequest(auth, request)) {

            return auth;

        }

    }

    return NULL;

}
```

如果认证数组中包含一个匹配的认证对象，接下来就看看证书仓库中是否保存有正确的证书。这样做可以避免你提示用户再次输入用户名和密码。可以参考列表 4-6 中的方式利用函数 `CFDictionaryGetValue` 查找证书。

列表 4-6 查找证书仓库

```
credentials = CFDictionaryGetValue(credentialsDict, authentication);
```

接下来把匹配的认证对象和证书应用到最开始的 HTTP 请求中，并且重新发送这个请求。

警告： 在接收到服务器要求以前，不要把证书应用于 HTTP 请求。在你上次得到认证之后，服务器可能已经发生了变化，这样做可能会产生安全隐患。

通过如上所属的修改，你的应用就可以保存认证对象和证书供以后重复使用。

在永久存储中保存证书

在内存中保存证书可以让用户在应用一次运行期间不用再次输入服务器的用户名和密码。不过，当应用退出后，所有这些证书都会被释放。为了防止丢失这些证书，可以把它们保存在永久存储中，这样每个服务器的证书只需要生成一次就可以了。keychain 是我们推荐用来保存证书的地方。尽管你可以有多个 keychains，本文档还是中的 keychain 特指用户的缺省 keychain。利用 keychain 意味着你存储的认证信息可以被访问同一个服务器的其他应用所使用，反之亦然。

通过 keychain 存储和获取证书需要用到两个函数：一个用于查找证书词典，另一个用于存储最近被访问到的证书。本文档中，这两个函数声明如下：

```
CFMutableDictionaryRef findCredentialsForAuthentication(

        CFHTTPAuthenticationRef auth);



void saveCredentialsForRequest(void);
```

函数 `findCredentialsForAuthentication` 首先检查内存中的证书词典，确定证书是否保存在本地缓冲区中。参考列表 4-6 中相关的实现。

如果证书没有在内存缓冲区，那么需要搜索 keychain 。搜索 keychain 需要用到函数 `SecKeychainFindInternetPassword`。该函数需要大量参数，这些参数，及其如何用于 HTTP 认证证书的相关描述，如下所示：

`keychainOrArray`
`NULL` 表示用户的缺省 keychain 列表。

`serverNameLength`
`serverName` 的长度，通常等于 `strlen(serverName)`。

`serverName`
从 HTTP 请求中获取到的服务器名。

`securityDomainLength`
安全域的长度，如果没有该域就等于0。在范例代码中，把 `realm ? strlen(realm) : 0` 传递进去同时考虑到两种情况。

`securityDomain`
认证对象的域，通过函数 `CFHTTPAuthenticationCopyRealm` 来获取。

`accountNameLength`
`accountName` 的长度。由于 `accountName` 等于 `NULL`，所以这个值为0。

`accountName`
在获取 keychain 数据项的时候是没有帐号名称的，所以这个值是 `NULL`。

`pathLength`
`path` 的长度，如果没有路径的话，该值为0。在范例代码中，传递给路径变量的值是 `? strlen(path) : 0`，这样可以同时考虑到两种情况。

`path`
认证对象的路径，通过函数 `CFURLCopyPath` 来获取。

`port`
端口号，通过函数 `CFURLGetPortNumber` 来获取。

`protocol`
表示协议类型的字符串，比如 HTTP 或者 HTTPS。通过调用函数 `CFURLCopyScheme` function 可以获得协议类型。

`authenticationType`
认证类型，通过函数 `CFHTTPAuthenticationCopyMethod` 获取。

`passwordLength`
0，因为在获取 keychain 数据项的时候不需要密码。

`passwordData`
`NULL`，因为在获取 keychain 数据项的时候不需要密码。

`itemRef`
keychain 数据项引用对象，`SecKeychainItemRef`，在找到正确的 keychain 数据项之后返回该值。

正确情况下，代码应该类似列表 4-7。

列表 4-7 搜索 keychain

```
didFind =

    SecKeychainFindInternetPassword(NULL,

                                    strlen(host), host,

                                    realm ? strlen(realm) : 0, realm,

                                    0, NULL,

                                    path ? strlen(path) : 0, path,

                                    port,

                                    protocolType,

                                    authenticationType,

                                    0, NULL,

                                    &itemRef);
```

我们假定 `SecKeychainFindInternetPassword` 可以成功返回，创建了一个 keychain 属性列表(`SecKeychainAttributeList`)，其中包含一个单独的 keychain 属性(`SecKeychainAttribute`)。这个 keychain 属性列表将包含用户名和密码。如果需要载入 keychain 属性列表，需要调用函数 `SecKeychainItemCopyContent` 并且将函数`SecKeychainFindInternetPassword` 返回的 keychain 数据项引用对象(`itemRef`)传递给它。这个函数会用帐号的用户名和 `void **` 作为密码填充 keychain 属性。

用户名和密码还可以被用来创建一套新的证书。列表 4-8 说明了这个过程。

列表 4-8 为 keychain 载入服务器证书

```
if (didFind == noErr) {



    SecKeychainAttribute     attr;

    SecKeychainAttributeList attrList;

    UInt32                   length;
```

```
    void                    *outData;

    // To set the account name attribute
    attr.tag = kSecAccountItemAttr;
    attr.length = 0;
    attr.data = NULL;

    attrList.count = 1;
    attrList.attr = &attr;

    if (SecKeychainItemCopyContent(itemRef, NULL, &attrList, &length, &outData)
        == noErr) {

        // attr.data is the account (username) and outdata is the password
        CFStringRef username =
            CFStringCreateWithBytes(kCFAllocatorDefault, attr.data,
                                    attr.length, kCFStringEncodingUTF8, false);
        CFStringRef password =
            CFStringCreateWithBytes(kCFAllocatorDefault, outData, length,
                                    kCFStringEncodingUTF8, false);
        SecKeychainItemFreeContent(&attrList, outData);

        // create credentials dictionary and fill it with the user name & password
        credentials =
            CFDictionaryCreateMutable(NULL, 0,
                                      &kCFTypeDictionaryKeyCallBacks,
                                      &kCFTypeDictionaryValueCallBacks);
        CFDictionarySetValue(credentials, kCFHTTPAuthenticationUsername,
                             username);
        CFDictionarySetValue(credentials, kCFHTTPAuthenticationPassword,
                             password);

        CFRelease(username);
        CFRelease(password);
    }
    CFRelease(itemRef);
}
```

从 keychain 中获取证书的功能只有在你首先把证书存储在 keychain 中才会有用。它的操作步骤与载入证书非常类似。首先，检查 keychain 中是否已经保存了证书，调用函数 `SecKeychainFindInternetPassword`，传递用户名作为 `accountName` 以及 `accountName` 的长度作为 `accountNameLength`。

如果数据项的确存在的话，修改其中的密码。设置 keychain 属性中的 `data` 域使它包含用户名，这样属性就算正确修改了。接着，调用函数`SecKeychainItemModifyContent`，并给它传递 keychain 数据项引用对象 (`itemRef`)、 keychain 属性列表以及新的密码。通过修改 keychain 数据项而不是直接覆盖它， keychain 数据项就可以被正确的更新并且所有与它相关的元数据也可以保留下来。数据项看起来就像列表 4-9中的样子。

列表 4-9 修改 keychain 数据项

```
// Set the attribute to the account name
attr.tag = kSecAccountItemAttr;
attr.length = strlen(username);
attr.data = (void*)username;
```

```
// Modify the keychain entry

SecKeychainItemModifyContent(itemRef, &attrList, strlen(password),

                             (void *)password);
```

如果数据项不存在的话，你只好从头创建一个了。函数 `SecKeychainAddInternetPassword` 可以完成这个任务。它的参数与函数 `SecKeychainFindInternetPassword` 相同，不过与调用函数 `SecKeychainFindInternetPassword` 不同的是，你需要向 `SecKeychainAddInternetPassword` 传递用户名和密码。在成功调用函数`SecKeychainAddInternetPassword` 之后需要释放 keychain 数据项引用对象，除非你需要在其它地方用到它。请参考列表 4-10 中的函数调用。

列表 4-10 存储一个新的 keychain 数据项

```
SecKeychainAddInternetPassword(NULL,

                               strlen(host), host,

                               realm ? strlen(realm) : 0, realm,

                               strlen(username), username,

                               path ? strlen(path) : 0, path,

                               port,

                               protocolType,

                               authenticationType,

                               strlen(password), password,

                               &itemRef);
```

认证防火墙

认证防火墙与认证服务器非常相似，唯一不同的地方就是，每个失败的 HTTP 请求都需要通过代理认证和服务器认证两个阶段检查。这意味着你需要为代理服务器和原始服务器准备独立的存储空间（包括本地的和永久的）。这样一来，一个失败的 HTTP 应答的过程应该是这样的：

- 确定应答的状态码是否是 407 (表示一个代理挑战)。如果是的话，就要通过检查本地的代理存储区和永久代理存储区，找到一个匹配的认证对象和证书。如果两个存储区都，没有匹配的认证对象和证书，那么就需要向用户请求证书。然后把认证对象应用于 HTTP 请求之后再次尝试。
- 确定应答的状态码是否是 401 (表示一个服务器挑战)。如果是这样，参照前面 407 应答的相同步骤，唯一不同的是，使用原始服务器存储区。

使用代理服务器的时候还有几个比较小的区别。首先，keychain 调用的参数来自于代理服务器和端口，而不是原始服务器的 URL 。其次，在请求用户输入用户名和密码的时候，请确保提示符明确说明了密码的作用范围。

通过这些要点，你的应用就可以正常工作于认证防火墙环境了。

操作 FTP 服务器

本章介绍了如何利用 CFFTP API 的一些基本特性。对 FTP 服务的管理是异步进行的，而对文件传输的管理是同步方式实现的。

本章内容：

下载一个文件

使用 CFFTP 的方式与使用 CFHTTP 非常类似，因为它们都是基于 CFStream。与其他异步使用 CFStream 的 API 类似，利用 CFFTP API下载文件需要你首先创建一个指向该文件的读操作流，以及一个回调函数。当读操作流接收到数据之后，回调函数就会运行，你需要以适当的方式下载收到的字节。这个过程通常需要两个函数：一个用来建立数据流，另一个作为回调函数。

本节内容：

建立 FTP 数据流

首先需要利用 `CFReadStreamCreateWithFTPURL` 函数创建一个读操作流，需要给这个函数传递一个指向远程服务器中需要下载的文件的URL 字符串。这个 URL 字符串看起来是这个样子： `ftp://ftp.example.com/file.txt`。请注意，这个字符串中包含了服务器名、路径以及文件。接下来需要创建一个指向本地存储位置的写操作流，这是通过调用函数 `CFWriteStreamCreateWithFile` 来完成的，需要给这个函数传递下载后的文件存储的位置。

由于写操作流和读操作流需要保持同步，所以最好能够创建一个结构，其中包含所有的公用信息，比如代理字典、文件大小、写入的字节数、剩下的字节数以及一个缓冲区。这个结构可能会是列表 5-1 中的样子。

列表 5-1 一个流结构

```
typedef struct MyStreamInfo {



    CFWriteStreamRef  writeStream;

    CFReadStreamRef   readStream;

    CFDictionaryRef   proxyDict;
```

```
    SInt64            fileSize;

    UInt32            totalBytesWritten;

    UInt32            leftOverByteCount;

    UInt8             buffer[kMyBufferSize];



} MyStreamInfo;
```

利用你刚刚创建的读操作流和写操作流对结构进行初始化。然后可以让操作流客户上下文 (CFStreamClientContext) 的info 域指向这个结构，这在将来会有用。

用 CFWriteStreamOpen 函数打开写操作流，这样你就可以开始写入本地文件了。通过调用函数 CFWriteStreamGetStatus 并检查它的返回值是否为kCFStreamStatusOpen 或者 kCFStreamStatusOpening，可以检查操作流是否正常打开。

读操作流打开之后，需要将读操作流与一个回调函数相关联。调用函数 CFReadStreamSetClient 并给它传递读操作流、回调函数可以接收的网络事件、回调函数的名称以及CFStreamClientContext 对象。再加上早先已经设置了操作流客户上下文的 info 域，这样一旦回调函数开始运行，你创建的结构就可以被发送给回调函数。

有些 FTP 服务器可能需要提供用户名，还有一些要求密码。如果你所访问的服务器要求提供一个用户名来进行认证，那么可以调用函数 CFReadStreamSetProperty，并给这个函数传递读操作流、kCFStreamPropertyFTPUserName 属性，以及一个指向 CFString 对象的引用，其中包含了用户名。另外，如果还需要密码的话，可以设置kCFStreamPropertyFTPPassword 属性。

有些电脑还会用到 FTP 代理。调用函数 SCDynamicStoreCopyProxies 并传递参数 NULL，可以在词典中查询代理设置。该函数会返回一个动态存储的引用。接下来我们设置读操作流的 kCFStreamPropertyFTPProxy 属性，并把代理词典作为值传递给它。在这里我们设置了代理服务器、指定了端口，并返回了一个布尔值，这个值表示对 FTP 数据流强制采用被动模式。

除了我们提到的属性，FTP 数据流还有几个其他属性。下面是这些属性的一个完整列表。

- kCFStreamPropertyFTPUserName — 用于登录的用户名(可设置的、可查询的，进行匿名 FTP 连接的时候不用设置)
- kCFStreamPropertyFTPPassword — 用于登录的密码(可设置的、可查询的，进行匿名 FTP 连接的时候不用设置)
- kCFStreamPropertyFTPUsePassiveMode — 是否使用被动模式(可设置的、可查询的)
- kCFStreamPropertyFTPResourceSize — 正在被下载的对象的期望大小，如果可以获得的话(可查询的，只用于 FTP 读操作流)
- kCFStreamPropertyFTPFetchResourceInfo — 是否获取资源信息，比如大小。在开始下载之前查询(可设置的、可查询的)，设置该属性可能会对性能产生影响
- kCFStreamPropertyFTPFileTransferOffset — 传输开始所在的文件偏移位置(可设置的、可查询的)
- kCFStreamPropertyFTPAttemptPersistentConnection — 是否尝试重新连接(可设置的、可查询的)
- kCFStreamPropertyFTPProxy — 包含了代理词典的键值对的 CFDictionary 类型(可设置的、可查询的)
- kCFStreamPropertyFTPProxyHost — FTP 代理主机的名称(可设置的、可查询的)
- kCFStreamPropertyFTPProxyPort — FTP 代理主机的端口号码(可设置的、可查询的)

在对读操作流设置的正确的属性值之后，可以利用函数 CFReadStreamOpen 打开这个流。如果这个过程没有返回错误结果，那么可以认为所有的操作流都被正确的建立了。

实现回调函数

你的回调函数将接收三个参数：读操作流、事件类型以及你的 MyStreamInfo 结构。事件类型决定了需要采取什么样的措施。

最常见的事件就是 kCFStreamEventHasBytesAvailable，在读操作流接收到来自服务器的字节时候会接收到这个事件。首先，通过函数 FReadStreamRead 来检查已经读取了多少字节。确定返回值大于0(小于0表示错误)，或者等于0(下载已经完成)。如果返回值是正的，那么可以通过写操作流将读操作流当中的数据写入磁盘。

调用函数 CFWriteStreamWrite 可以把数据写入写操作流。有时候，函数 CFWriteStreamWrite 可能在完成写入读操作流中的所有数据之前就返回了。因此，需要建立一个循环，只要还有数据需要写入就继续写操作。这个循环的实例代码在列表 5-2 中，其中 info 就是 "建立操作流" 中的 MyStreamInfo 结构。这种写入写操作流的方法使用了“阻塞”操作流。你也可以使写操作流成为事件驱动，以获得更好的性能，不过代码会更加复杂一点。

列表 5-2 将读操作流当中的数据写入写操作流

```
bytesRead = CFReadStreamRead(info->readStream, info->buffer, kMyBufferSize);



//...make sure bytesRead > 0 ...



bytesWritten = 0;

while (bytesWritten < bytesRead) {

    CFIndex result;



    result = CFWriteStreamWrite(info->writeStream, info->buffer + bytesWritten, bytesRead - bytesWritten);

    if (result <= 0) {

        fprintf(stderr, "CFWriteStreamWrite returned %ld\n", result);

        goto exit;

    }

    bytesWritten += result;

}

info->totalBytesWritten += bytesWritten;
```

只要读操作流中还有剩余的字节，就重复进行整个这个过程。

其他两个需要特别注意的事件就是 kCFStreamEventErrorOccurred 以及 kCFStreamEventEndEncountered。如果出现错误，可以通过函数 CFReadStreamGetError 查找错误，然后退出。如果遇到文件末尾，那么你的下载就结束了，可以退出。

在完成一切操作之后，确保删除所有的操作流以及没有其他进程使用这个操作流。首先，关闭写操作流，并将 client 设置为 NULL。接着从循环语句当中取消操作流，并释放它们。在完成之后删除操作流。

上传一个文件

上传文件与下载文件是比较类似的。 和下载文件相同，你需要一个读操作流和一个写操作流。不同的是，在上载文件的时候，读操作流是用于访问本地文件而写操作流用于远程文件。我们可以参考"建立操作

流"中的说明，不过其中涉及到读操作流的部分，需要修改为写操作流，反之同理。

在回调函数中，不是寻找 kCFStreamEventHasBytesAvailable 事件，而是寻找 kCFStreamEventCanAcceptBytes 事件。首先，用读操作流从文件中读取字节，把读取出来的数据存储进 MyStreamInfo 所代表的缓冲区。接着，运行 CFWriteStreamWrite 函数将缓冲区中的数据推送到写操作流中。CFWriteStreamWrite 的返回值是写入操作流的字节数。如果写入操作流的字节数小于从文件中读取的字节数，那么请计算二者相差的字节，并将它们存储到缓冲区中。在下一个写循环中，如果还有未写入的字节，那么在从读操作流中读取数据以前，请将这些未写入的数据写入写操作流。只要写操作流可以接收字节数据(CFWriteStreamCanAcceptBytes)，就可以重复这个过程。在列表 5-3 中查看这个循环的代码。

列表 5-3 将数据写入写操作流

```
do {

    // Check for leftover data

    if (info->leftOverByteCount > 0) {

        bytesRead = info->leftOverByteCount;

    } else {

        // Make sure there is no error reading from the file

        bytesRead = CFReadStreamRead(info->readStream, info->buffer,

                                     kMyBufferSize);

        if (bytesRead < 0) {

            fprintf(stderr, "CFReadStreamRead returned %ld\n", bytesRead);

            goto exit;

        }

        totalBytesRead += bytesRead;

    }



    // Write the data to the write stream

     bytesWritten = CFWriteStreamWrite(info->writeStream, info->buffer, bytesRead);

    if (bytesWritten > 0) {



        info->totalBytesWritten += bytesWritten;



        // Store leftover data until kCFStreamEventCanAcceptBytes event occurs again

        if (bytesWritten < bytesRead) {

            info->leftOverByteCount = bytesRead - bytesWritten;

            memmove(info->buffer, info->buffer + bytesWritten,

                    info->leftOverByteCount);

        } else {

            info->leftOverByteCount = 0;

        }

    } else {

        if (bytesWritten < 0)

            fprintf(stderr, "CFWriteStreamWrite returned %ld\n", bytesWritten);

        break;

    }



} while (CFWriteStreamCanAcceptBytes(info->writeStream));
```

另外，在下载文件的时候，还需要检查 kCFStreamEventErrorOccurred 和 kCFStreamEventEndEncountered 事件。

创建一个远端目录

如果想要在一个远程服务器上创建一个目录，需要象上载文件一样创建一个写操作符。不同的是，传递给函数 CFWriteStreamCreateWithFTPURL 的 CFURL 对象是一个目录路径，而不是一个文件，文件名以一个斜杠结尾。比如，正确的路径应该是 ftp://ftp.example.com/newDirectory/，而不是ftp://ftp.example.com/newDirectory/newFile.txt。当循环调用到回调函数的时候，它会发送 kCFStreamEventOpenCompleted 事件，表示目录已经被创建。

只有一层的目录可以用 CFWriteStreamCreateWithFTPURL 函数创建。另外，只有你拥有对服务器的适当权限，才可以创建目录。

下载一个目录列表

从 FTP 下载一个目录列表的方式与下载或者上载文件略有不同。这是因为需要对得到的数据进行扫描。首先，建立一个读操作流获取目录列表。这个过程和下载文件类似：创建操作流、注册一个回调函数、在循环中安排操作流(如果必要，设置用户名、密码和代理信息)并最终打开这个操作流。在下面的实例中，在查询目录列表的时候你不需要读操作流或者写操作流，因为得到的数据将会显示在屏幕上而不是文件中。

在回调函数中，需要留意 `kCFStreamEventHasBytesAvailable` 事件。在从读操作流载入数据之前，需要确保操作流中没有上次回调函数运行时候的遗留数据。从`MyStreamInfo` 结构的 `leftOverByteCount` 域中获取偏移量。然后从操作流中读取数据，这时候需要将刚刚计算到的偏移量考虑在内。另外还需要计算缓冲区大小和读取的字节数。这个过程在列表 5-4 中完成。

列表 5-4 从目录列表中获取数据

```
// If previous call had unloaded data

int offset = info->leftOverByteCount;



// Load data from the read stream, accounting for the offset

bytesRead = CFReadStreamRead(info->readStream, info->buffer + offset,

                             kMyBufferSize - offset);

if (bytesRead < 0) {

    fprintf(stderr, "CFReadStreamRead returned %ld\n", bytesRead);

    break;

} else if (bytesRead == 0) {

    break;

}

bufSize = bytesRead + offset;

totalBytesRead += bufSize;
```

在数据被读入缓冲区中之后，建立一个循环来扫描这些数据。被扫描的数据不一定是整个目录列表，它可以是(有可能是)列表的一部分。创建扫描数据的循环可以用函数 `CFFTPCreateParsedResourceListing`，该函数需要的参数包括数据的缓冲区、缓冲区的大小以及一个字典引用。它返回的是扫描的字节数。只要这个返回值大于0，循环就不会中止。函数 `CFFTPCreateParsedResourceListing` 创建的字典包含了所有的字典列表信息，更多有关字典中的"键"的信息请参考 "建立操作流"。接着，按照列表 5－5 中所示，计算读取的字节数以及缓冲区大小。

列表 5-5 载入字典列表并扫描

```
do

{

    bufRemaining = info->buffer + totalBytesConsumed;



    bytesConsumed = CFFTPCreateParsedResourceListing(NULL, bufRemaining,

                                                     bufSize, &parsedDict);

    if (bytesConsumed > 0) {



        // ...Print out data from parsedDict...

        CFRelease(parsedDict);



        totalBytesConsumed += bytesConsumed;

        bufSize -= bytesConsumed;

        info->leftOverByteCount = bufSize;



    } else if (bytesConsumed == 0) {



        // This is just in case. It should never happen due to the large buffer size

        info->leftOverByteCount = bufSize;

        totalBytesRead -= info->leftOverByteCount;

        memmove(info->buffer, bufRemaining, info->leftOverByteCount);



    } else if (bytesConsumed == -1) {
```

```
        fprintf(stderr, "CFFTPCreateParsedResourceListing parse failure\n");

        // ...Break loop and cleanup...

    }



} while (bytesConsumed > 0);
```

当操作流已经更多字节时，清理所有的操作流并将它们从循环当中清除掉。

使用网络诊断功能

在很多基于网络功能的应用程序当中，会发生一些与应用无关的网络错误。而很多用户这个时候并不知道为什么应用程序会出错。CFNetDiagnostics API 可以让你迅速而简单的帮助用户修复网络问题，而你只需要做很少的工作。

如果你的应用程序使用了 CFStream 对象，那么可以通过调用 `CFNetDiagnosticCreateWithStreams` 来创建一个网络诊断对象的引用(`CFNetDiagnosticRef`)。`CFNetDiagnosticCreateWithStreams` 函数的参数包括一个定位器对象、一个读操作流以及一个写操作流。如果你的应用程序仅仅会用到一个读操作流或者写操作流，那么没有用到的参数会被设置为 `NULL`。

如果没有流对象，你还可以通过一个 URL 直接创建一个网络诊断对象引用。操作方式是这样的，调用 `CFNetDiagnosticCreateWithURL` 函数并把一个定位器对象和一个作为`CFURLRef` 参数的 URL 对象传递给它。它会返回一个网络诊断对象的引用供你使用。

如果要通过网络诊断助手来诊断问题的话，需要调用 `CFNetDiagnosticDiagnoseProblemInteractively` 函数，并把网络诊断对象应用传递给它。列表 6-1 中说明了如何在一个循环中利用 CFNetDiagnostics 和流对象。

列表 6-1 在一个流对象发生错误的时候使用 CFNetDiagnostics API

```
    case kCFStreamEventErrorOccurred:

        CFNetDiagnosticRef diagRef =

            CFNetDiagnosticCreateWithStreams(NULL, stream, NULL);

        (void)CFNetDiagnosticDiagnoseProblemInteractively(diagRef);

        CFStreamError error = CFReadStreamGetError(stream);

        reportError(error);

        CFReadStreamClose(stream);

        CFRelease(stream);

        break;
```

CFNetworkDiagnostics 还可以让你在不使用网络诊断助手的时候有能力查询问题的状态。这是通过调用 `CFNetDiagnosticCopyNetworkStatusPassively` 来实现的，它会返回一个常数值，比如 `kCFNetDiagnosticConnectionUp` 或者 `kCFNetDiagnosticConnectionIndeterminate`。

文档修订历史

本表列出了 CFNetwork 编程指南的修订历史。

日期	备注
2006-05-23	更新了与 CFReadStreamCreateForHTTPRequest 有关的内容。
2006-04-04	更新了与 HTTP 服务器通讯方面的代码。
2006-03-08	对通篇进行了少量改动。
2006-02-07	更新了大量内容，把参考信息转移到新的文档 "CFNetwork 引用"。修改原来的标题“CFNetwork 服务编程指南 ”。
2005-08-11	更正了对 CFNetServiceCreate 中端口参数的描述。
2005-04-29	更新了有关 Mac OS X v10.4 版本的内容。将 "Rendezvous" 修改为 "Bonjour "。修改原来的标题“CFNetwork 服务”。
2004-02-01	增加了对 CFFTP 和 CFHost 的描述，并阐明了 CFNetwork 服务当前支持的协议。更正了小节“处理写操作流”和 “使用循环防止阻塞”中的示例代码。更正了对 `CFReadStreamClientCallback` 和 `CFWriteStreamClientCallback` 回调函数中 `clientContext` 参数的描述。