上海大学

SHANGHAI UNIVERSITY

毕业设计（论文）

UNDERGRADUATE PROJECT（THESIS）

题目：基于 socket 的文件传输的设计与实现

学院 计算机工程与科学

专业 计算机科学与技术

学号 10122060

学生姓名 吕伟彬

指导老师 雷咏梅

起讫日期 2014.02.17—2014.05.30

【目录】

基于 socket 的文件传输设计与实现

摘要

本文从先讨论了 socket 网络编程的方法，然后讨论了文件传输协议的设计与实现，主要包括 HTTP、FTP 和 BitTorrent 协议。涉及到以上几种文件传输协议的协议规范，以及开源实现的实现细节，包括 libcurl（HTTP 和 FTP）以及 libtransmission（BitTorrent）。同时描述了我本人实现的文件传输管理器 wdl 以及一个简单的 HTTP 服务器 wdls 的设计与实现。wdl 支持 FTP、HTTP 和 BitTorrent 协议，多任务多线程下载，支持断点续传；可以从互联网上下载文件。wdls 是一个非常简单的 HTTP 服务器，实现 HTTP200，HTTP404 和 HTTP206，支持断点续传，可以与 wdl 很好地配合。wdl 和 wdls 都是在 Linux 完全采用 C 语言实现，wdls 不依赖任何第三方库。

关键词：Socket,linux,文件传输协议，HTTP 服务器

ABSTRACT

This Document describes the design and implement of file transfer protocol using Socket, including HTTP, FTP and BitTorrent Protocol. The specifications and open source implements of those protocols are involved. Libcurl is introduced as HTTP and FTP implement, while libtransmission as BitTorrent implement. Also, my file transfer manager named wdl and simple HTTP server named wdls are described in the document. Wdl supports FTP, HTTP and BitTorrent Protocol, multi-task, multi-thread and breakpoint-resume. It's OK to download files from the Internet using wdl. And wdls is a very simple HTTP server, which only implements HTTP200, HTTP206 and HTTP404, and breakpoint-resume too. Wdl and wdls work well together. Wdl and wdls are both writen in C language in Linux System. In addition, wdls does not depend on any third-party library.

Keywords:Socket,linux,file transfer protocl,HTTP server

绪论

本课题从科研与实际应用出发，从 Socket 网络编程出发，旨在提高学生对网络数据传输的理解与认识。并能在实际生产中加以运用。

Socket 最早作为 BSD UNIX 的进程通信机制，随着网络的出现，它也开始支持网络通信；事实上，在 UNIX 系统下，网络通信被认为是一种特殊的进程间通信。网络的快速发展，使得 Socket 网络编程越来越流行，以至于现在很多人都忽略了 Socket 作为本机进程间通信的功能。本文中所谈到的 Socket，除非特别说明，都是只指网络套接字。而且 Windows 最早的 IP/TCP 实现就来源于 BSD UNIX 的代码，所以 Windows 下的网络编程接口也是 Socket 兼容的。

Socket 实际上就是操作系统内核（主要是 IP/TCP 协议栈）对外提供的网络编程接口，它属于系统调用层面，而不是特定语言的标准库。因为大部分操作系统都是 C 语言实现的，因此，Socket 接口一般都是以 C 函数的形式提供。而像 C++，Java，Python 这些语言的网络支持，要么是对 C 语言的封装，如 C++；要么就是通过虚拟机（如 Java）或者解释器（如 Python）间接调用 Socket 接口；总之，它们最底层还是要调用操作系统提供的 Socket 接口。因此，了解熟悉 Socket 接口对理解网络编程，无论哪个层面的，都非常有益。

既然 Socket 是作为操作系统网络协议栈对外的接口，那么了解相应的网络协议（主要是 TCP/IP）就很有必要了。本文的前一部分会简要描述 TCP/IP 协议的规范，但不会很深入，对这方面感兴趣的读者可以参考 IP/TCP 详解[1]。接着，我们讨论 Socket 网络编程接口，从系统调用的层面展示网络编程的方法，这要求读者对 UNIX 系统下的 C 语言编程有一定基础，但这些并不在本文的讨论范围内；对这方面感兴趣的读者可以参考 UNIX 环境高级编程[2]。

Socket 套接字其实不仅仅局限于网络编程，它有多种协议类型，最典型的就是 AF_INET 和 AF_INET6，分别对应的是 IPv4 和 IPv6 套接字。AF_UNIX 或者 AF_LOCAL 是 UNIX 域套接字，用于本地进程间通信。还有其他协议特定的类型，比如 AF_IPX 是 IPX 协议套接字，AF_X25 是 ITU-T X.25 / ISO-8208 协议套接字，以及内核的 netlink 套接字 AF_NETLINK。本文只讨论 AF_INET，IPv6 套接字与 IPv4 类似，主要是地址结构不同。

而 AF_INET 协议下又可以指定 SOCK_STREAM 和 SOCK_DGRAM，分别对应 TCP 和 UDP 协议；甚至可以指定 SOCK_RAW 使用原始套接字,原始套接字可以接受到 IP 的数据报文，由程序员手动构造和解析 TCP 或者 UDP 协议首部字段。本文只讨论 SOCK_STREAM。也就是说只讨论 TCP 套接字，同时也只描述 TCP 和 IP 协议规范。

本文还讨论基于 Socket 的网络文件传输协议，FTP、HTTP 和 BitTorrent 协议。FTP 协议是流行的文件传输协议，主要用在托管大量文件的服务器上；HTTP 最初为了互联网浏览网页而设计，虽然协议中有很多与网页服务相关的字段，但本质就是文件传输的协议；BitTorrent 近年来也越来越流行，主要用在大文件的传输上。

最后结合上述的内容来描述我所完成的文件下载器 wdl 和一个简单的 HTTP 服务器 wdls，主要从网络编程的角度描述，而不是从软件工程的角度。

Socket 网络编程是比较底层的一种网络编程模型，国外，主要是美国有很好的研究与实现，主流的网络协议 TCP/IP、FTP、HTTP 等等都来源于国外；而国内对计算机网络底层的研究比较少，也没有什么成熟的产品，要改变现状，首先要从根本上提高计算机行业从业人员的专业素质，不能仅仅满足于开发一些上层应用，而对底层的具体实现知之甚少。

第一章 TCP/IP

引言

TCP/IP 起源于 60 年代末美国政府资助的一个分组交换网络研究项目，到 90 年代已经发展成为计算机之间最常用的联网方式。因为其协议规范以及很多实现都不需要花钱或者只要花很少的钱就可以获得，它被认为是一个开放的系统，是全球互联网的基础。Socket 本质上就是操作系统对外提供的操作 TCP/IP 协议的编程接口，了解 TCP/IP 协议对使用 Socket 进行网络编程有重要意义。本章简要描述了 TCP/IP 协议的规范。

我们有时说 TCP/IP 协议族，起始包括了很多其他协议，比如 ICMP、IGMP、UDP 等，只是 IP 和 TCP 是这些协议中用得最多的。本章也只讨论 IP 和 TCP 协议。

第一节 分层

网络协议通常是分层次的，OSI 定义了七层，这里为了简化讨论，我们采用 TCP/IP 的分层模型。

图 1-1 TCP/IP 协议族的层次

每一层都有不同的功能。最底层的是链路层，也可以称为数据链路层和网络接口层，一般包括了操作系统中的设备驱动程序，主要是网络接口。它们处理与电缆（或者任何其他传输媒介）的物理接口的细节。然后是网络层，这层处理数据分组在网络中的活动，比如路由选路等，这一层主要包括的网络协议有 IP 协议，ICMP 协议和 IGMP 协议。网络层上面是传输层，这层为数据提供端到端的通信，主要是 TCP 和 UDP 协议，也就是说 TCP 协议是基于 IP 协议的。最上面则是应用层，这层不处理任何网络数据传输的细节，而负责应用程序特定的数据细节，这一层的协议有 HTTP、FTP、Telnet 等。

Socket 网络编程主要是完成传输层的连接和数据传输，在此基础上可以实现 FTP、HTTP 等协议，甚至是用户自定义的应用层协议。下面我们具体讨论 IP 协议和 TCP 协议。

第二节 IP：网际协议

IP 是 TCP/IP 协议族中最核心的协议。所有的 TCP、UDP、ICMP 以及 IGMP 数据都是以 IP 数据报格式传输。IP 提供的是不可靠、无连接的数据报传输，所谓的不可靠是指 IP 协议不保证数据真正到达目的地，如果当中的某个地方出错了，比如路由器缓存区用完，那么 IP 数据报会被直接丢弃。无连接则指 IP 数据报之间没有任何关系，它们是互相独立的，IP 协议不维护任何有关数据报状态的信息。可靠性和可连接性都由上层协议，比如 TCP 来完成。

第一条 IP 首部

每个 IP 数据报都有一个 IP 首部，一般其长度是 20 字节，除非包含有选项字段。下图表示了 IP 首部结构，最高位在左边，记作 0bit；最低位在右边，为 31bit。

<table>
<tr><td>版本号
(4 位)</td><td>首部长度 (4
位)</td><td>服务类型 TOS
(8 位)</td><td colspan="2">字节总长度
(16 位)</td></tr>
<tr><td colspan="3">标识
(16 位)</td><td>标志
(3 位)</td><td>偏移量
(13 位)</td></tr>
<tr><td colspan="2">生存时间 TTL
(8 位)</td><td>协议号
(8 位)</td><td colspan="2">首部校验和
(16 位)</td></tr>
<tr><td colspan="5">源 IP 地址
(32 位)</td></tr>
<tr><td colspan="5">目的 IP 地址
(32 位)</td></tr>
<tr><td colspan="5">选项 (如果有)</td></tr>
<tr><td colspan="5">负载数据
(. . .)</td></tr>
</table>

图 1-2 IP 首部结构

版本号是 4，也就是 IPv4。首部长度是 4 位二进制数，最大 15，表示的是首部占 32bit 字的数目，因此 IP 首部最长为 60 字节。如果没有选项，IP 首部长度为 20 字节。服务类型 (TOS) 字段包括一个 3bit 的优先权子字段 (现在已被忽略)，4bit 的 TOS 字段和 1bit 的保留位。4bit 的 TOS 字段分别表示：最小时延、最大吞吐量、最高可靠性和最小费用。总长度字段表示了整个 IP 数据报的长度，以字节为单位，总长度减去 IP 首部长度就得到了实际负载数据的长度，IP 数据报总长度最大为 65535 字节。标识字段唯一标识每一份 IP 数据报，通常每发送一份 IP 数据报它的值就加 1，最后会回归到 0。TTL 表示 time-to-live，指一个 IP 数据报在网络上的最大生存时间，不过它其实指的不是真实的时间，而是路由的跳数，每经过一个路由，TTL 会减 1，某个路由器发现 IP 数据报的 TTL 为 0，则直接丢弃，然后发送 ICMP 报文通知源主机。这也是 traceroute 的工作方式。

为了加深理解，我们看一下 linux 内核中 IP 首部的结构；

```
struct iphdr {
#if defined(__LITTLE_ENDIAN_BITFIELD)
    __u8    ihl:4,
        version:4;
#elif defined (__BIG_ENDIAN_BITFIELD)
    __u8    version:4,
        ihl:4;
#else
#error  "Please fix <asm/byteorder.h>"
#endif
    __u8    tos;
    __be16  tot_len;
    __be16  id;
```

```
    __be16  frag_off;
    __u8    ttl;
    __u8    protocol;
    __sum16 check;
    __be32  saddr;
    __be32  daddr;
    /*The options start here. */
};
```

第二条 IP 地址

互联网上的每个接口都必须有一个唯一的地址，也就是 IP 地址。就像在前面 IP 首部中看到的一样，IP 地址长 32bit。IP 地址具有一定的结构，分为五类，A、B、C、D 和 E 类地址。都接口主机具有多个 IP 地址，每个接口都有一个对应的 IP 地址。IP 地址还可以分为三类，单播地址（目的地为单个主机，这也是最多的情况），广播地址（目的地是给定网络上的所有主机）以及多播地址（目的地为同一组内所有主机）。

在 TCP/IP 领域中，域名系统（DNS）是一个分布的数据库，它提供了 IP 地址和主机名之间的映射信息。大部分网络程序都允许指定 IP 地址或者域名。

第三节 TCP：传输控制协议

TCP 在 IP 协议之上，提供了一种面向连接的、可靠的字节流服务。面向连接意味着两个使用 TCP 的应用（一般是一个客户端和一个服务器）在交换数据之前必须先建立一条 TCP 连接。而可靠性 TCP 使用以下方式来实现：

1. 数据被分割成 TCP 认为的最合适的大小发送，因为 IP 数据报无法保证接受顺序和发送顺序一致，因此再在接收端将缓存 TCP 数据报直到一个完整 TCP 数据报可以组合。
2. 当 TCP 发送一个数据包后，会启动一个定时器，等待接受方确认收到这个数据包。如果超时则认为发送失败。同样的，如果收到来自对方的数据，将返回一个确认。
3. TCP 将保证它首部和数据的校验和，如果校验和验证失败，则认为该数据包无效，丢弃。
4. IP 数据报会发生重复，TCP 将丢弃重复的数据报
5. TCP 会进行流量控制，保证接受的数据不会超过缓存区大小。

第一条 TCP 首部

TCP 数据报被封装在 IP 数据报中，如下图所示，

图 1-3 TCP 封装

TCP 首部和 IP 首部一样，如果没有任何选项，也是 20 个字节。这意味着，一个 TCP 数据报一般至少是 40 字节的。下图显示了一个 TCP 协议首部的结构。

<table>
<tr><td colspan="8">源端口号
(16 位)</td><td colspan="2">目的端口号
(16 位)</td></tr>
<tr><td colspan="10">序号
(32 位)</td></tr>
<tr><td colspan="10">确认序号
(32 位)</td></tr>
<tr><td>首部长度
(4 位)</td><td colspan="2">保留
(6 位)</td><td>URG</td><td>ACK</td><td>PSH</td><td>RST</td><td>SYN</td><td>FIN</td><td>窗口大小
(16 位)</td></tr>
<tr><td colspan="8">检验和
(16 位)</td><td colspan="2">紧急指针
(16 位)</td></tr>
<tr><td colspan="10">选项
(. . .)</td></tr>
<tr><td colspan="10">负载数据
(. . .)</td></tr>
</table>

图 1-4 TCP 首部

回想上面的 IP 首部，IP 首部中是没有端口号的，但是有 IP 地址，而 TCP 首部中包含了端口号，用于寻找接受和发送数据的应用进程。源 IP 地址和源端口号确定了一个源主机，目的 IP 地址和目的端口号确定了目的主机，从而唯一确认了一个 TCP 连接。其实，最早的 TCP 规范中，将端口号称为插口（socket）。序号用来标识从 TCP 数据报的发送字节流，表示在这个报文段中的第一个数据字节。用于接收端进行报文排序。确认序号用于接收端向发送端确认数据已成功接收，其值是上次接受到数据字节序号加 1，只有在 ACK 被设置时才有效。窗口大小用于接收端向发送端告知接收缓冲区的大小，如果接受端的缓冲区过小，发送端一般会暂停发送，直到接受端有了足够的缓冲区。

TCP 首部中有 6 个标志位，它们分别是

1．URG　紧急指针有效

2．ACK　确认序号有效

3．PSH　接受方应该尽快将这个报文段交给应用层

4．RST　重建连接

5．SYN　同步序号，用于发起连接

6．FIN　发送端已经发送结束

下面同样拿出 linux 内核中 TCP 首部的结构，以加深对 TCP 首部的理解。

```
struct tcphdr {
    __be16 source;
    __be16 dest;
    __be32 seq;
    __be32 ack_seq;
```

```
#if defined(__LITTLE_ENDIAN_BITFIELD)
     __u16  res1:4, doff:4,
             fin:1, syn:1, rst:1, psh:1,
            ack:1, urg:1, ece:1, cwr:1;
#elif defined(__BIG_ENDIAN_BITFIELD)
     __u16  doff:4, res1:4,
            cwr:1, ece:1, urg:1, ack:1,
            psh:1, rst:1, syn:1, fin:1;
#else
#error      "Adjust your <asm/byteorder.h> defines"
#endif
     __be16 window;
     __sum16check;
     __be16 urg_ptr;
};
```

第二条 TCP 连接

TCP 的连接过程是著名的三次握手。首先客户端设置一个初始序号，发送一个 SYN 请求；服务器返回该 SYN 的 ACK 确认的同时也发送一个他自己的 SYN 请求，客户端确认服务器的 SYN 请求后，连接建立。

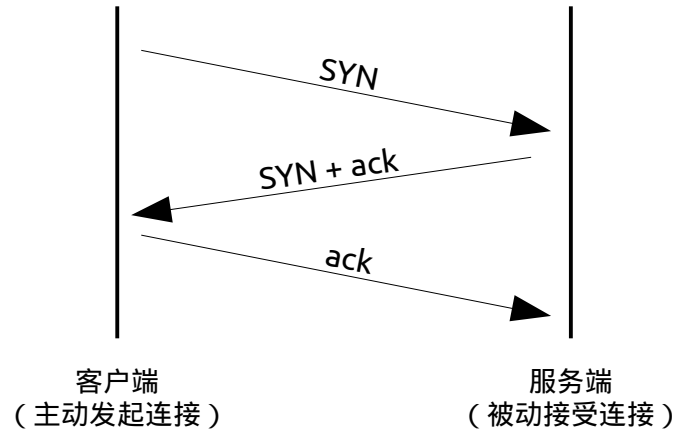

图 1-5 TCP 连接过程

第二章 Socket 网络编程

引言

本章简单地介绍 Socket 套接字 API，主要围绕 IPv4/TCP 描述；同时会用 wdls 的实现作为具体实例。wdls 是完全采用 C 实现简单 HTTP 服务器，它不依赖任何第三方库，直接使用系统调用和标准 C 库。

第一节 套接字地址结构

大多数套接字函数需要一个套接字地址结构作为参数。不同协议使用不同的地址结构，比如 IPv4 的地址结构为 sockaddr_in，IPv6 的地址结构为 sockaddr_in6。sockaddr_in 结构定义在 netinet/in.h 中：

```
/* Structure describing an Internet (IP) socket address. */
#define __SOCK_SIZE__ 16            /* sizeof(struct sockaddr)   */
struct sockaddr_in {
  __kernel_sa_family_t      sin_family;   /* Address family           */
  __be16              sin_port;     /* Port number              */
  struct in_addr      sin_addr;     /* Internet address         */

  /* Pad to size of `struct sockaddr'. */
  unsigned char             __pad[__SOCK_SIZE__ - sizeof(short int) -
                    sizeof(unsigned short int) - sizeof(struct in_addr)];
};
#define sin_zero      __pad         /* for BSD UNIX comp. -FvK   */
```

sin_family 表示地址协议，可以是 AF_INET 或者 AF_INET6。sin_port 表示端口号，是以网络字节序表示的 16 位无符号数。sin_addr 表示真正的 IP 地址，这里当然是二进制表示的，而不是我们熟知的点分十进制。最后的__pad 是填充字段，为了保证 sockaddr_in 的长度与 sockaddr 结构的长度一致。

in_addr 是一个只有一个字段 s_addr 的结构，一般引用 sin_addr 字段时可以直接 sin_addr.s_addr。

sockaddr 是一个通用的地址结构，为了统一套接字函数的地址参数而出现。

```
struct sockaddr {
      sa_family_t sa_family;  /* address family, AF_xxx   */
      char        sa_data[14];    /* 14 bytes of protocol address */
};
```

下面我们看一下 wdls 是如何处理地址结构的。wdls 作为一个服务器，它需要把套接字和与本机地址绑定，因此要设置自己的 IP 地址结构。下面是 wdls 处理本机地址结构的实现代码。

```
struct sockaddr_in addr;

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

addr.sin_family = AF_INET;

addr.sin_addr.s_addr = htonl(INADDR_ANY);

addr.sin_port = htons(port);
```

先初始化 sockaddr_in 结构，所有字段设置为 0。然后设置地址类型为 AF_INET，表示 IPv4 地址。这里把 sin_addr 设置为 INADDR_ANY 表示让操作系统选择一个本机地址，因为我的测试机器是只有一个网络接口，也就是一个 IP 地址，所以这么处理是没有问题的，但是对于那些有多个 IP 地址的多接口主机，让系统选择一个 IP 地址不是一个好办法。最后设置 sin_port 端口号。这里的 htonl 和 htons 都是将本机字节序转化为网络字节序。其含义是 host to network long 和 host to network short，网络数据采用大端字节序，如果本机字节序也是大端，那么这两个函数不做任何处理，否则转化为大端字节序表示。关于更多字节序的内容，读者可以参考 ON HOLY WARS AND A PLEA FOR PEACE[3]。

同时，系统提供了将点分十进制表示的 IPv4 地址转化为二进制形式的函数。

```
#include <arpa/inet.h>

int inet_aton(const char *strptr,struct in_addr *addrptr)

                                                若转化成功则返回 1，否则返回 0
```

strptr 是点分十进制表示的 IP 地址字符串，而 addrptr 是用于返回二进制表示的地址结构的指针。如果要指定某个 IP 地址，可以这样设置

```
inet_aton("123.123.123.123",&(addr.sin_addr));
```

还有几个类似的地址转换函数 inet_addr 和 inet_ntoa，不再赘述。

第二节 TCP 套接字编程

本节描述编写一个完整的 TCP 客户端/服务器程序所需要的基本套接字函数。同样会引入 wdls 中的具体实例来讲解。

第一条 Socket 函数

为了执行网络通信，进程要做的第一件事情就是调用 socket 函数来创建一个指定的套接字描述符。套接字的类型可以是 IPv4 协议、IPv6 协议等，我们只描述 IPv4 协议套接字的 TCP 协议。要创建一个 TCP 套接字，要像如下调用 socket 函数：

```
int sockfd=socket(AF_INET,SOCK_STREAM,0);  或者

int sockfd=socket(AF_INET,SOCK_STREAM,IPPROTO_TCP);
```

第三个参数如果指定为 0，则系统会自动选择一种类型为 SOCK_STREAM 的套接字，一般就是 TCP 套接字，当然也可以手动指定 IPPROTO_TCP 创建 TCP 套接字。第一种，就是 wdls 中创建套接字的方式。

在 UNIX 系统下，网络套接字被抽象成一个特殊的文件，因此，套接字描述符和文件描述符也是统一管理的，也就是说，文件描述符和套接字描述符是不会重复的。如果已经打开了文件 0、1、2、3、4，再创建一个套接字描述符，则为 5。反之亦然。

第二条 connect 函数

connect 函数用于 TCP 客户端建立与 TCP 服务器的连接。它有三个参数，第一个是 socket 函数返回的有效套接字描述符，第二个和第三个参数分别是 TCP 服务器的地址结构和地址结构的长度。

connect 是客户端调用向服务器发起连接的函数，因此服务端 wdls 并不需要调用。下面代码展示了客户端如何调用 connect 函数向服务器发起 TCP 连接。connect 函数成功返回 0，失败返回-1。

```
sockaddr_in addr;
addr.sin_family = AF_INET;
inet_aton("123.123.123.123",&(addr.sin_addr));    /* 设置地址 */
addr.sin_port = htons(80);                                    /* 设置端口号 */
connect(sockfd,(struct sockaddr*)&addr,sizeof(addr));
```

调用 connect 函数后内核会向目标发起 TCP 连接，这里涉及到 TCP 连接过程中的三次握手，因此 connect 函数是会阻塞的。

第三条 bind 函数

bind 函数将一个本地协议地址赋予一个套接字描述符，对于 IPv4，是一个 32 位的地址和一个 16 位的端口号的组合。对于客户端来说，这个函数是可以不调用的，而且大部分客户端都不调用它；不调用 bind 不代表套接字描述符就没有绑定的地址，而是系统内部自动赋予一个本地的地址和端口号。对于客户端来说，由系统自动设置地址和端口号是没有问题的，但是服务器一般都是要手动调用 bind 的。因为服务器一般有多个网络接口，更重要的是服务器需要把自己的地址和端口号告诉客户端，如果让系统自动设置，那么这个地址和端口号都是随机的。

回想套接字地址结构一节，我们设置了一个本地的套接字地址结构，然后 wdls 就可以直接将其与套接字描述符绑定。

```
bind(sockfd, (struct sockaddr *) &addr, sizeof(addr));
```

第四条 listen 函数

用 socket 创建的套接字默认是主动的，所谓主动套接字就是可以执行 connect 主动发起连接的；但是服务器往往是被动等待客户端连接，这就需要调用 listen 函数了，listen 函数有两个作用，第一个就是将主动模式的套接字转化为被动模式，第二个是设置内核应该为套接字排队的最大连接个数。

在 wdls 中如下调用 listen 函数。

```
listen(sockfd, BACKLOG);
```

其中 BACKLOG 是数字 10 的宏定义。

listen 函数虽然有“监听”的意思，但只是设置套接字的被动模式，并没有任何网络操作，因此也不会阻塞。下面所讲的 accept 函数才真正监听网络连接。

第五条 accept 函数

accept 函数由 TCP 服务器调用，用于从已完成连接队列中返回下一个连接，如果连接队列为空，则进入阻

塞。注意，一个套接字在调用 listen 后就可以接受来自客户端的连接，不过只有服务器调用了 accept 函数后，该连接才真正被服务器处理，在此期间，连接被放在连接完成队列里面。不过一般服务器调用 listen 后就会立马调用 accept，因此，这个时间差基本可以忽略。

accept 函数有三个参数，第一个是套接字描述符，第二个和第三个分别是地址结构的指针和一个长度结构的指针，它们是作为返回值存在。如果有连接已经完成，accept 返回这个连接的套接字描述符，同时在第二个和第三个参数分别返回客户端的地址结构和地址结构的长度。

在 wdls 中用一个循环调用 accpet 函数，每次成功都返回一个套接字描述符，然后创建一个线程来处理这个连接，代码如下；

```
while ((accfd = accept(sockfd, (struct sockaddr *) &arg->addr,
                       &arg->addrlen)) > 0) {
      arg->sockfd = accfd;
      http_thread(arg);
      arg = http_thread_arg_new();
}
```

每次成功返回一个连接套接字后，就设置一个线程参数结构，然后 http_thread 中创建子线程来处理这个连接。关于 http_thread 会在后面详细描述其实现，现在我们只关注于套接字接口相关的函数。

第六条 I/O 操作

当 TCP 连接建立以后，就可以像读写文件一样调用 read 和 write 来读写网络连接。但是所表现的行为有些不同，对套接字的 read 和 write 可能实际输入或者输出的数据少于请求的，这不算是错误，原因在于内核中的输出缓冲区已满，或者还没有接受到来自网络的足够多的数据。

第七条 TCP 客户端实例

本章节展示了一个简单的 TCP 客户端，该程序从命令行读取一个 IP 地址，然后发起 HTTP 请求，读取响应，并打印在终端。为了简单起见，没有做任何错误处理，只为了体现 socket 网络编程的方法。

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#include <sys/socket.h>
#include <arpa/inet.h>

int main(int argc, char *argv[])
{
```

```
        if(argc!=2){
                return -1;
        }
        const char *ip=argv[1];
        struct sockaddr_in addr;
        addr.sin_family=AF_INET;
        inet_aton(ip,&(addr.sin_addr));
        addr.sin_port=htons(80);            /* HTTP 服务端口号 80 */

        int sockfd=socket(AF_INET,SOCK_STREAM,0);
        connect(sockfd,(struct sockaddr*)&addr,sizeof(addr));

        const char request="GET / HTTP/1.1\r\n\r\n";              /* 一个最简单的 HTTP 请求 */

        write(sockfd,request,strlen(request));            /* 将 HTTP 请求发送给服务器 */

        char buff[1024];
        int n=read(sockfd,buff,1024);                     /* 只读取一次数据 */

        buff[n]='\0';
        printf("%s",buff);

        close(sockfd);                      /* 像关闭文件一样关闭套接字 */
        return 0;
}
```

该程序的运行结果如下：（省略了部分内容）

```
wiky@thunder:test$ ./a.out 202.120.127.189
HTTP/1.1 400 Bad Request
Content-Type: text/html; charset=us-ascii
Server: Microsoft-HTTPAPI/2.0
Date: Tue, 20 May 2014 05:54:25 GMT
......
```

第三章 应用层协议

引言

前两章简单描述了 IP/TCP 协议规范以及相应的 socket 套接字编程。本章讨论基于 IP/TCP 协议的应用层协议，包括 HTTP、FTP 和 BitTorrent，以及同样基于 socket 的开源实现。其中 HTTP 和 FTP 使用流行广泛的开源网络库 libcurl 来描述，而 BitTorrent 使用 libtransmission 来描述。

本章会引用部分源代码来描述，分别是 curl-7.3.2 和 transmission-2.8.2。在引用源代码时会标出所在文件以及所在行号。比如在讨论 FTP 时，lib/ftp.c:2256 表示 curl-7.3.2 源代码目录下 lib/ftp.c 文件中第 2256 行。

第一节 FTP：文件传输协议

FTP 是一个古老的文件传输协议，具体协议规范读者可以参考 RFC959[4]。这里只做简单介绍，结合 libcurl 的实现。

尽管 HTTP 协议已经代替了 FTP 大多数功能，FTP 仍然是互联网上传输文件的一种有效途径。FTP 客户端可以向服务器发送命令来下载文件、上传文件、创建甚至改变服务器上的目录，只要有相应的权限。FTP 相对于 HTTP 的优势就在于他有权限管理，不过大部分 FTP 服务器都允许匿名访问。回想第一章第一节中描述的网络分层模型，FTP 处于应用层，在 TCP 协议之上，或者说是基于 TCP 协议的。

FTP 服务一般运行在 20 和 21 两个端口。端口 20 用于在客户端和服务器之间传输数据，称之为数据流；而端口 21 则用于传输控制命令，称之为控制流。在 FTP 中有几个术语需要明确，

- 服务端控制进程：服务器上运行的 FTP 控制流管理进程，用于和客户端交换控制信息。
- 服务端数据进程：服务器上运行的 FTP 数据流管理进程，由控制进程管理，与客户端交换数据。
- 客户端控制进程：客户端上运行的 FTP 控制流管理进程，用于和服务器交换控制信息。
- 客户端数据进程：客户端上运行的 FTP 数据流管理进程，由控制进程管理，与服务器交换数据。

下图描述了 FTP 的通信模型。

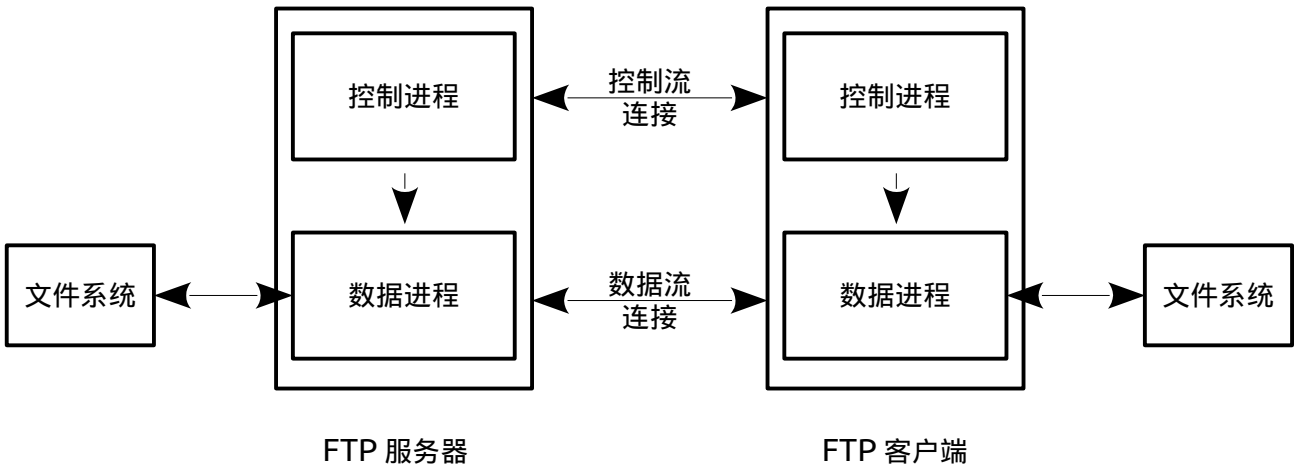

图 3-1 FTP 通信模型

按照 FTP 的设计，控制进程和数据进程可以不在同一个主机上，主要保证控制进程可以控制数据进程进行及时的数据传输。当然，大部分情况下，控制进程和数据进程都是在同一个系统中，甚至可能是同一个进程。

FTP 的控制命令以 NVT ASCII 串的格式传输。每个命令以三个或者四个大写的 NVT ASCII 字符开始，后面带有选项参数和一个 CR/LF 对来表示命令结束。应答由三个 NVT ASCII 数字以及一个选项消息组成。一个长应答也许会有多个消息组成，第一个消息的三个数字后带有一个破折号，最后的消息不带有破折号。中间的消息无须携带三个数字，但是如果带有三个数字，则也需要破折号。FTP 支持很多控制命令，这里只列出其中几个，

- RETR：从远程系统取回一个文件。
- PWD：显示服务器端的当前工作目录。
- LIST：显示当前工作目录下的文件列表。

FTP 提供了认证机制，但是大部分站点都支持匿名访问。FTP 的控制流使用 TELNET 协议交换信息，包含 TELNET 命令和选项。然后大多数 FTP 控制桢都是简单的 ASCII 文本，可以分为 FTP 命令和 FTP 消息。FTP 消息是对 FTP 命令的响应，它由带有解释文本的应答代码构成。

FTP 有两种连接方式，主动方式和被动方式。主动方式的连接过程是：客户端向服务器的 FTP 端口（默认是 21）发送连接请求，服务器接受链接，并建立一条命令链路。当需要传输数据时，客户端在命令链路上用 PORT 命令告诉服务器：“我打开了 X 端口”。于是服务器从 20 端口向客户端的 X 端口发起链接，建立一条数据链路来传输数据（这是大部分 FTP 客户端的传输方式）。被动方式的链接过程是：客户端向服务器的 FTP 端口（默认是 21）发起连接请求，服务器接受连接，建立一条命令链路。当需要传送数据时，服务器在命令链路上用 PASV 命令告诉客户端：“我打开了 X 端口”。于是客户端向服务器的 X 端口发送链接请求，建立数据链路来传输数据。

以客户端发送 FTP 的 PORT 命令为例。FTP 命令大部分都是纯文本，PORT 当然也是。典型的 PORT 命令形式为 PORT 10,2,0,2,4,31。该命令发送的消息其实是表示 10.2.0.2:1039 的 IP 地址和端口。PORT 是命令，后面的数字表示 IP 地址和端口号。前四个表示 IPv4 的地址 10.2.0.2。后面的 4,31 表示十六进制的 0x04 和 0x0F，合并后 0x040F 就是十进制端口值 1039。命令用\r\n 结尾。

下面代码演示了 FTP 的 PORT 命令。

```
/* 将 IP 地址和端口表示为 a,b,c,d,e,f 形式 */

p_address_to_port_repr (&addr, port, bytes, sizeof (bytes));

/* 构造 PORT 请求.  */

snprintf(request,”PORT %s\r\n”,bytes);

/* 发送请求 */

  nwritten = fd_write (csock, request, strlen (request), -1);
```

被动模式的 FTP 通常用在处于防火墙之后的 FTP 客户访问外界 FTP 服务器的情况，因为在这种情况下，防火墙通常配置为不允许外界访问防火墙之后主机，而只允许由防火墙之后的主机发起的连接请求通过。因此，在这种情况下不能使用主动模式的 FTP 传输，而被动模式的 FTP 可以良好的工作。

FTP 的响应信息由两部分组成，响应码和描述信息。常见的响应码有 125,150,220。例如 200 的描述信息为”Command okay”。下面列出了是部分响应码。

响应码	响应格式
120	120 Service ready in nnn minutes
125	125 Data connection already open;transfer starting
150	150 File status; about to open data connection
200	200 Command okay
211	211 System status, or system help reply
530	530 Not logged in

响应码大于 400 表示命令执行失败。

FTP 断点续传的原理与 HTTP 断点续传类似。都是向服务器发送请求获取特定位置的数据。在 HTTP 中通过首部字段 RANGE 指定，而在 FTP 中，客户端向服务器发送 REST 指令指定文件偏移位置。比如

```
REST 42314
```

表示从文件的 42314 字节处开始传输数据。在重启 wdl 后，要继续未完成的下载，就要先获取已下载数据的大小，然后向服务器发送 REST 指令，继续下载。

第二节 libcurl 的 FTP 实现

libcurl 没有实现 FTP 的所有命令，事实上，因为 FTP 的一些命令用得很少，很多 FTP 实现都选择性地忽略了它们。但至少实现了 RETR 命令。RETR 后面跟一个文件名，向服务器请求相应的文件。我们就用这个命令作为例子讲解 libcurl 是如何实现 RETR 命令的。

在开始 FTP 数据传输之前，需要先进行 FTP 连接，而 FTP 连接的基础是 TCP 的连接。底层的 TCP 连接时通过系统调用的 socket 套接字完成的，不过 libcurl 对 socket 进行了一定层度的封装。

FTP 连接在函数 ftp_connect 中完成。该函数首先将 ftp_statemach_act 函数指针赋值给 connectdata 结构中的 statemach_act 字段，然后通过 ftp_multi_statemach 函数间接调用 ftp_statemach_act。

ftp_statemeach_act 根据 FTP 连接的当前状态不同调用不同的处理函数，默认情况下采用的是 FTP 被动模式。

```
lib/ftp.c:3101

    case FTP_PASV:

      result = ftp_state_pasv_resp(conn, ftpcode);

      break;
```

ftp_state_pasv_resp 函数先解析地址，将域名或者字符串形式的 IP 地址转化为二进制表示，然后调用 Curl_connecthost 发起服务器的连接。

Curl_connecthost 内部又调用了 singleipconnect 完成对服务器的连接。

singleipconnect 先调用 Curl_socket 创建一个套接字描述符，然后调用 connect 函数完成连接。

```
lib/connect.c:955

res = Curl_socket(conn, ai, &addr, &sockfd);
```

```
lib/connect.c:1022
rc = connect(sockfd, &addr.sa_addr, addr.addrlen);
if(-1 == rc)
        error = SOCKERRNO;
```

Curl_socket 只是 socket 函数的简单包裹，代码在 lib/connect.c:1272。

完成 TCP 连接后发送 FTP 的 USER 命令进行登录，该操作在函数 ftp_state_user 中完成。

```
lib/ftp.c:835
static CURLcode ftp_state_user(struct connectdata *conn) {
        CURLcode result;
        struct FTP *ftp = conn->data->state.proto.ftp;
        /* send USER */
        PPSENDF(&conn->proto.ftpc.pp, "USER %s", ftp->user?ftp->user:"");
```

USER 默认是 anonymous，发送完 USER 后，有些服务器会要求密码，有些则直接返回 2xx 表示登录成功了。如果要求密码，则发送密码，否则登录成功。这些操作在函数 ftp_state_user_resp 中完成。

```
lib/ftp.c:2626
if((ftpcode == 331) && (ftpc->state == FTP_USER)) {
        /* 331 Password required for ... (the server requires to send the user's password too) */
        PPSENDF(&ftpc->pp, "PASS %s", ftp->passwd?ftp->passwd:"");
        state(conn, FTP_PASS);
        } else if(ftpcode/100 == 2) {
        /* 230 User ... logged in. (the user logged in with or without password) */
        result = ftp_state_loggedin(conn);
        }
        ......
```

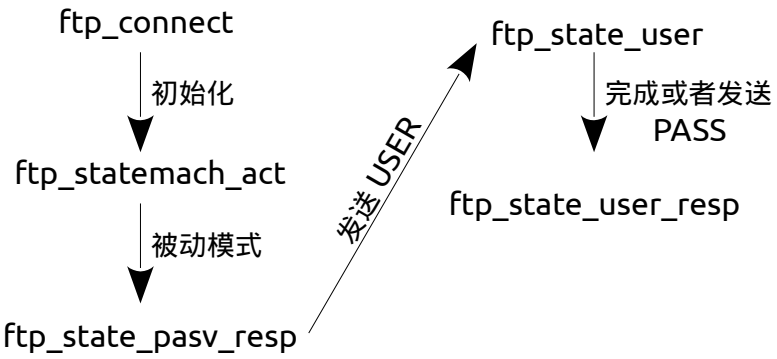

图 3-2 FTP 的连接过程

完成登录后，就可以使用 RETR 命令下载文件了，在函数 ftp_state_retr 中完成。它先判断是否指定了下载位置（断点续传），如果指定了，那么使用 REST 命令从断点开始下载，否则使用 RETR 命令。

```
lib/ftp.c:2331

PPSENDF(&ftpc->pp, "RETR %s", ftpc->file);
```

PPSENDF 是一个宏定义，展开以后其实内部调用了 Curl_ftpsendf()。

Curl_ftpsendf()（源代码 lib/ftp.c：4135）发送 ftp 请求的，它的参数可以变，类似于 printf。如果要发送 PORT 命令，可以这样调用 Curl_ftpsendf(conn,”PORT %s,%s,%s,%s,%s,%s”,a,b,c,d,e,f)。

Curl_ftpsendf()先根据指定的形式构造字符串，

```
write_len = vsnprintf(s, SBUF_SIZE-3, fmt, ap);
```

在添加\r\n 结束符。

```
strcpy(&s[write_len], "\r\n");
```

最后发送数据。

```
 res = Curl_write(conn, conn->sock[FIRSTSOCKET], sptr, write_len, &bytes_written);
```

Curl_GetFTPResponse()（源代码 lib/ftp.c：675）获取返回的 FTP 响应。FTP 的请求命令与响应都是纯文本形式，很容易解析。

第三节 HTTP：超文本传输协议

HTTP 和 FTP 一样是一种应用层协议，它是一种通用的、无状态的协议，除了原本的超文本传输之外还可以用在其他很多地方。很多应用程序使用 HTTP 和服务器交换数据。当然使用 HTTP 最多的当然是互联网上的网页服务。本章节只对 HTTP 协议规范做简单介绍，关于 HTTP 协议的详细内容，读者可以参考 RFC2616[5]。

HTTP 协议由请求和响应构成，是一个标准的客户端服务器模型。HTTP 允许传输任何类型的数据对象，正在传输的类型由首部字段中的 Content-Type 标记。HTTP 协议是无状态的协议，这意味着协议本身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，这样可能会导致每次连接传送的数据量增大。

HTTP 的请求信息包括几个方面，首先是请求行，然后是请求的 HTTP 首部，最后是实际可选的消息内容。请求行和 HTTP 首部字段必须用回车换行（\r\n）结尾，HTTP 首部结束用一个空行（只有\r\n）结束。消息内容可有可无。下面是一个典型的 HTTP 请求：

```
GET /index.html HTTP/1.1\r\n

Host: www.shu.edu.cn\r\n

Accept: */*\r\n

\r\n
```

HTTP 请求行又包括三个部分；请求方法，上例中是 GET，请求的资源，上例中是/index.html（一般是网站的主页），还有表示 HTTP 协议的 HTTP/1.1。请求的资源是任意的，看服务器是否有相应的解释。而请求方法在 HTTP/1.1 中共定义了八种，下面列出最常用的几种。

- GET：请求特定的资源。
- POST：向特定资源提交数据进行处理请求，比如提交表单和上传文件。
- PUT：向指定资源位置上传其最新内容。
- DELETE：请求服务器删除指定的资源。

HTTP 首部的字段其实是可以自定义的，只要通信双方都认同。不过 HTTP/1.1 协议定义了总多 HTTP 首部，比如 Accept 表示客户端能接收的资源类型，Accept-Language 则表示客户端能接收的语言类型，Host 则表示客户端请求的服务器地址，很多 HTTP 服务器都要求客户端在请求中加入 Host 字段，如果 Host 的值与其本身的域名不匹配，则不提供服务。

HTTP 的响应消息也分为三部分，首先是响应行，然后是响应的 HTTP 首部字段，最后是响应消息正文。同样用回车换行（\r\n）来分割各个部分。下面是一个典型的 HTTP 响应，省略了消息正文。

```
HTTP/1.1 200 OK\r\n
Content-Type: text/html; charset=us-ascii \r\n
Server: Microsoft-HTTPAPI/2.0 \r\n
Date: Wed, 21 May 2014 04:53:20 GMT \r\n
Connection: close \r\n
Content-Length: 334 \r\n
\r\n
[消息正文]
```

响应行也分为三个部分，首先是 HTTP 版本号，上例中是 HTTP/1.1，一般和客户端请求的版本号会一致。还有就是响应代码，HTTP 协议定义了大量状态码，状态码由一个三位的整数表示，在这里是 200。最后是一个状态的解释字符串，上例中是 OK，这个字符串可以是服务器自定义的，不过一般客户端直接解释状态码。

HTTP 的状态码分为五类，分别是:

- 消息：以 1 开头的状态码，如 100、101；表示请求已经接受，需要继续处理。
- 成功：以 2 开头的状态码，如 200、201；表示请求已经被服务器接受、理解而且被处理。
- 重定向：以 3 开头的状态码，如 300、301；表示请求的资源已经移动了位置，请使用新位置重新请求。
- 请求错误：以 4 开头的状态码，如 400、404；表示客户端的请求是一个错误的请求，服务器无法处理，最典型的情况是客户端的请求服务器无法找到，返回 404。
- 服务器错误：以 5 开头的状态码，如 500，501；表示服务器在处理客户端的请求时出现了一个错误。很可能是执行一个有错误的脚本引起的。

HTTP 响应的 HTTP 首部和请求的首部格式是一样的，但是字段不太一样，主要是针对响应的字段。

HTTP 断点续传是通过指定下载位置实现的，HTTP 请求中可以在首部字段中添加 RANGE，比如

```
Range： bytes=20000-
```

该字段表示客户端希望服务器返回从 20000 字节开始数据。比如客户端第一次下载了某个资源的前 20000 字节的数据，想继续下载就可以指定该字段。服务器返回的响应码是 206，表示服务器处理了部分请求，即只返回部分请求资源的数据。同时在响应的 HTTP 首部中会加入 Content-Range 字段，如下

```
Content-Range=bytes 20000-29999/30000
```

表示返回的数据是指定资源从 20000 字节到 29999 字节的数据，总长度为 30000。注意，数据的字节位置是从 0 开始的，因此总长度为 30000 的数据，最后一个字节位置是 29999。

第四节 libcurl 的 HTTP 实现

libcurl 实现了 HTTP 大部分功能，我们同样只用 HTTP 最简单的实例来说明 libcurl 是如何实现 HTTP 的。因为 GET 是 HTTP 请求中最普遍的，也是 libcurl 创建 HTTP 请求时的默认方法，因此我们以 GET 请求方法为例，讲解 libcurl 是如何实现 HTTP 数据传输的。在我们讨论具体实现之前，先会想 TCP 套接字连接一节所展示的简单 HTTP 客户端实例，我们可以用 libcurl 来实现，如下。

```
#include <stdio.h>
#include <curl/curl.h>
int main(int argc, char *argv[])
{
	if(argc!=2){
		printf("Usage: a.out URL\n");
		return 0;
	}
	CURLcode code;
	CURL *easy=curl_easy_init();

	code=curl_easy_setopt(easy,CURLOPT_URL,argv[1]);
	if(code!=CURLE_OK){
		fprintf(stderr,"Invalid URL!\n");
		return -1;
	}
	curl_easy_setopt(easy,CURLOPT_WRITEDATA,stdout);

	code=curl_easy_perform(easy);

	curl_easy_cleanup(easy);
	return 0;
```

```
}
```

使用 curl_easy_init()创建好一个 CURL 对象后，调用 curl_easy_setopt()设置 url，然后就可以调用 curl_easy_perform 来执行网络数据传输了。

可以看到，libcurl 隐藏了 URL 的解析过程以及具体的 socket 通信。curl_easy_init()创建一个 CURL 对象，然后调用 curl_easy_setopt()分别设置 URL 和数据输出位置。在本例子中，输出到 stdout。最后调用 curl_easy_perform()发送 HTTP 请求并接受响应。

curl_easy_setopt()可以设置很多参数，这里不做过多详细描述。唯独讲解一下关于 CURLOPT_WRITEFUNCTION 和 CURLOPT_PROGRESSFUNCTION。

其中 CURLOPT_WRITEFUNCTION 是一个原型为 size_t function(char *ptr, size_t size, size_t nmemb, void userdata)的函数指针，一旦 libcurl 接收到了数据，就会调用该函数。ptr 表示接受到的数据指针，实际数据大小是 size*nmemb，由用户自行觉得如何处理数据。如果该函数返回的值不是 szie*nmemb 那么就认为数据传输被用户中止。curl_easy_perform 将返回 CURLE_ABORTED_BY_CALLBACK。

同样的 CURLOPT_PROGRESSFUNCTION 表示一个原型为 int function(void *clientp, double dltotal, double dlnow, double ultotal, double ulnow)的函数指针。它会被 libcurl 周期性地调用表示数据传输的进度。dltotal 和 dlnow 分别表示下载的数据总量和当前的下载量。ultotal 和 ulnow 则分别对应上传的数据总量和当前上传量。如果该函数返回非 0 值同样会中止数据传输，curl_easy_perform 返回 CURLE_ABORTED_BY_CALLBACK。

CURL 其实是 void 的别名（源代码 include/curl/curl.h：93）。真正的结构是 SessionHandle（源代码 lib/urldata.h：1614），SessionHandle 包含了完成数据传输用到的各种字段和设置，主要由 curl_easy_setopt()设置。

curl_easy_setopt()（源代码 lib/easy.c：439）其实是 Curl_setopt()（源代码 lib/url.c：647）的简单封装。Curl_setopt()则是一个 switch 结构，主要就是完成选项的配置。

curl_easy_perform()（源代码 lib/easy.c：470）发起连接，完成必要的数据传输。在本例子中，它将接受到的数据写到 stdout，因为之前的 curl_easy_setopt()设置。

首先需要构造和发送 HTTP 首部，该任务由 Curl_http()（源代码：lib/http.c：1633）完成。该函数首先做一些通用检查，是否是 HTTP 协议，HTTP 方法，HTTP 协议版本，以及一些 HTTP 首部。以 HTTP 首部 Transfer-Encoding 为例，下面是 Curl_http()对 Transfer-Encoding 做的检查和操作。

```
ptr = Curl_checkheaders(data, "Transfer-Encoding:");

if(ptr) {

       /* Some kind of TE is requested, check if 'chunked' is chosen */

       data->req.upload_chunky = Curl_compareheader(ptr, "Transfer-Encoding:", "chunked");

} else {

       if((conn->handler->protocol&CURLPROTO_HTTP) &&

                              data->set.upload &&

                              (data->set.infilesize == -1)) {

              if(conn->bits.authneg)
```

```
                /* don't enable chunked during auth neg */
                ;
            else if(use_http_1_1(data, conn)) {
                    /* HTTP, upload, unknown file size and not HTTP 1.0 */
                    data->req.upload_chunky = TRUE;
            } else {
                    failf(data, "Chunky upload is not supported by HTTP 1.0");
                    return CURLE_UPLOAD_FAILED;
            }
        } else {
            /* else, no chunky upload */
            data->req.upload_chunky = FALSE;
        }
        if(data->req.upload_chunky)
           te = "Transfer-Encoding: chunked\r\n";
}
```

程序先检查 Transfer-Encoding 首部是否被设置。如果被设置了，设置 chunked 标志。如果没有 Transfer-Encoding 首部，首先检查是否是 HTTP 协议，是否有数据要上传，然后根据上传的数据来设置 Transfer-Encoding 首部的值，也就是“自动设置”。其他首部字段如 Host，Referer 等都是通过类似的流程处理。

所谓的断点续传，只需要在首部中加入 Range 字段并设置相应的断点位置就可以。

构造完成 HTTP 首部后，程序进入一个 switch 结构，判断 HTTP 方法

```
switch(httpreq) {
        case HTTPREQ_POST_FORM:
        …
        case HTTPREQ_PUT:
        ...
```

根据不同的 HTTP 方法，添加特定的 HTTP 首部，比如 POST 方法中需要添加 Content-Type 和 Content-Length 两个首部字段。如果没有指定任何 HTTP 首部，会默认添加 Host 和 Accept 两个首部。如下

```
GET / HTTP/1.1
Host: www.baidu.com
Accept: */*
```

完成方法特定首部的构造后，发送 HTTP 请求。Curl_write()（源代码 lib/sendf.c：231）完成发送 HTTP 请求的工作，它的第二个参数就是一个 socket 文件描述符。

完成 HTTP 请求的发送后，就是接受 HTTP 响应。Curl_done()（源代码 lib/url.c:5623）完成接收 HTTP 响应的工作。

HTTP 响应的起始行用\r\n 分割，每个首部字段之间也用\r\n 分割，首部名与对应的值用：分割，首部与消息主体用一个空行\r\n 分割，都是纯文本内容；很容易解析。

第五节 wdls 的 HTTP 实现

本节我们描述 wdls 是如何实现一个简单的 HTTP 服务器的。在描述 wdls 的具体实现之前，我们先看一下 wdls 到底能做什么。在终端运行 wdls 后，会显示

```
Server address: http://localhost:6323

Server root path: ./root

Server running... press ctrl+c to stop.
```

此时，wdls 已经开始正常运行，它打开了 6323 端口作为监听端口，此时可以在浏览器中输入 http://localhost:6362 访问 wdls 运行的小网站。下图是浏览器中运行的截图，

图 3-3 wdls 主页

图 3-4 wdls 的子页面

同时我们可以在 wdls 的终端中看到这样的输出，

```
Server address: http://localhost:6323
Server root path: ./root
Server running... press ctrl+c to stop.
    127.0.0.1:36372 requests ./root/index.html
    127.0.0.1:36373 requests ./root/favicon.ico
    127.0.0.1:36374 requests ./root/favicon.ico
    127.0.0.1:36387 requests ./root/README
    127.0.0.1:36388 requests ./root/favicon.ico
```

显示的是客户端的 IP 地址和端口号，以及它请求的资源。wdls 不支持保活连接，因此我们可以看到每次请求资源都用了新的 TCP 连接（端口号不一样）。下面开始描述 wdls 的具体实现。

wdls 是一个典型的多线程并发服务器，在主线程监听来自客户端的 TCP 连接，一旦接收到连接，则创建新线程，并传递连接参数，由子线程处理该连接。子线程创建后，先读取来自客户端的请求，解析该请求，然后做出响应。程序的整体流程很简单，下面具体描述各个模块。

```
int sockfd = Socket(AF_INET, SOCK_STREAM, 0);
/* 设置地址重利用，避免重启程序后端口未被释放 */
int opt=1;
setsockopt(sockfd,SOL_SOCKET,SO_REUSEADDR,&opt,sizeof(opt));

struct sockaddr_in addr;
```

```
memset(&addr, 0, sizeof(addr));
addr_set((struct sockaddr *) &addr, AF_INET, arg_get_port());
Bind(sockfd, (struct sockaddr *) &addr, sizeof(addr));
Listen(sockfd, BACKLOG);
```

首先在主线程创建套接字并绑定本地地址，然后监听端口，端口号可以由命令行参数设置，默认是6323。这里要注意的是以大写字母开头的包裹函数，比如 Socket 和 Bind。所谓的包裹函数只是执行了错误除了的系统调用或者库函数的简单封装。比如 Socket 的实现如下

```
int Socket(int family, int type, int protocol)
{
	int sockfd = socket(family, type, protocol);
	if (G_UNLIKELY(sockfd < 0)) {
		sys_exit("Fail to create socket descriptor");
	}
	return sockfd;
}
```

如果调用 socket 出错，则简单地退出程序。G_UNLKEYLY 是一个宏定义，其实是 GCC 定义地分支判断宏地包裹，如下

```
#define G_UNLIKELY(x) __builtin_expect(!!(x), 0)
```

其他包裹函数的实现都是类似的，后面不再赘述，所有包裹函数都在文件 Socket.h 和 Socket.c 中定义和实现。

setsockopt 函数用来设置套接字选项，上述代码中调用 setsockopt 是设置地址可以重利用。默认情况下，如果一个地址被绑定了，那么在解除绑定（一般是程序退出）后的一段时间内是不允许重新绑定的，设置地址重利用后可以突破这个限制。保证每次地址都可以绑定成功。

主线程在完成套接字创建并绑定本地地址后在一个无限循环中接受来自客户端的请求。

```
int accfd;
HttpThreadArg *arg = http_thread_arg_new();
while ((accfd = Accept(sockfd, (struct sockaddr *) &arg->addr,
                               &arg->addrlen)) > 0) {
	arg->sockfd = accfd;
	http_thread(arg);
	arg = http_thread_arg_new();
}
```

一旦接受到来自客户端的连接，首先构造好线程参数 arg，然后调用 http_thread 函数创建子线程来处理

HTTP 连接。在没有出错的情况下，主线程在这个 while 循环中一直执行，直到用户按下 CTRL+C 手动终止进程。下面我们来看子线程的实现。http_thread 函数在 http.c 中实现，其实非常简单，它只是对 POSIX 线程的简单封装。如下，关于 POSIX 线程，读者可以参考 Wikipedia 的解释[6]。

```
void http_thread(HttpThreadArg * arg) {
	pthread_t tid;
	Pthread_create(&tid, NULL, httpPthread, arg);
	Pthread_detach(tid);
}
```

因此实际执行 HTTP 处理的是 httpPthread 函数。在具体解释 httpPthread 函数之前，我先阐述一下处理 HTTP 请求和返回响应的基本流程。首先我们要接受 HTTP 请求，并解析它，解析 HTTP 请求的第一步就是将请求“分行”。回想 HTTP 的请求首部是根据回车换行分割的\r\n。因此按\r\n 将 HTTP 请求分为若干行，第一行是请求行，按照 Method Url HTTP/1.1 的方式解析，主要是 Method 和 Url。后面若干行，直到遇到一个空行都是 HTTP 首部的字段，按照 name:value 的形式解析。最后一个空行表示 HTTP 首部结束。wdls 不支持请求中附带数据，因此不处理任何空行后面的数据。

下面是 httpPthread 函数中解析 HTTP 请求的代码。

```
HttpRequest *request = http_request_new();
while ((line = http_readline(sockfd))) {
	if (line == NULL) {		/* 出错 */
		goto CLOSE;
	} else if (line[0] == '\0') {	/* 空行，意味着首部结束 */
		Free(line);
		break;
	} else if (first) {		/* 首行 */
		first = 0;
		request->startLine = http_start_line_parse(line); /* 解析首行 */
		if (request->startLine == NULL) {
			goto CLOSE;
		}
	} else {				/* HTTP 首部 */
		http_request_add_header_from_line(request, line);
	}
	Free(line);
}
```

http_readline 函数每次从套接字中读取一行，这个函数在 httpbuf.c 中实现，内部使用了缓冲机制，我们后面讨论。如果读到的数据是 NULL 则表示读取出错，调到出错处理例程。如果读到了空行，则表示 HTTP 请求已经结束，如果读到的是第一行，则表示是 HTTP 请求行，http_start_line_parse 对其进行解析；最后如果是 HTTP 首部中的一行，http_request_add_header_from_line 对其进行解析并加入 HTTP 请求的结构中。如果没有出错，那么就得到了一个有效的 HttpRequest 结构。然后我们调用 echoToClient 对其请求进行响应。

```
echoToClient(sockfd, request);
```

echoToClient 函数向客户端返回请求的资源，如果资源不存在返回一个预定义的 404 错误。该函数先打开客户端请求的资源，如果打开失败，则认为客户端的请求无效，返回 404 错误。

```
int fd = open(path, O_RDONLY);
Free(path);


if (fd <= 0) {              /* 指定的资源没有找到，返回 404 错误 */
       const char *res = response404();
       Write(sockfd, res, strlen(res));
       return;
}
```

response404 只是返回一个定义好的 404 消息的字符串。

如果打开成功，说明客户端请求的资源是有效的，这时判断 HTTP 请求中是否包含一个 Range 字段，表示断点续传。这里注意，Range 其实除了指定断点之外，可以指定一个中间范围的区域，比如

```
Range: bytes=2000-3000/4000
```

表示只取中间 2000 到 3000 字节的数据，但是 wdls 不处理这种情况，wdls 只处理指定开始位置到文件结束的情况。

```
if (range == NULL) {
       /* 没有指定数据位置，不是断点 */
       res = response200(length);
} else {
       /* 断点续传，返回 206 */
       off_t position = getRangePosition(range);
       res = response206(length, position);
       setFilePosition(fd, position);
}
```

如果指定了断点，那么要向客户端返回 206 消息，同时设置文件的传输位置，否则就是 200 消息。最后我们先把 HTTP 的响应消息返回给客户端，然后再将文件传输过去。

```
if (Write(sockfd, res, strlen(res)) < 0) {
        goto OUT;
}

ssize_t readn;
char buf[1024];
while ((readn = Read(fd, buf, 1024)) > 0) {
        if (Write(sockfd, buf, readn) < 0) {
                goto OUT;
        }
}
```

echoToClient 函数返回后，httpPthread 就结束了，这样一个 HTTP 请求就处理完毕。最后我们再看一下，http_readline 函数是如何实现每次返回一行数据的。

我们知道从套接字中读取数据是没法指定读取一行的，行读取的方式必须应用层自己实现。这里我使用了缓冲的方式实现。每次读取指定大小的数据，但是每次返回一行，如果不足一行就再读取数据，直到有一行，或者读到结尾则返回剩余的所有数据。但是这里又有个问题，如果使用静态缓冲区，那么各个线程是共享的，必然会互相干扰。为此我使用了线程专有数据。

所谓线程专有数据（Thread-Specific Data），是 POSIX 线程定义的一种实现线程内全局数据的方式。其实是线程内独享的一块缓冲区。关于线程专有数据的内容不在本文的讨论范围内，对此有兴趣的读者可以参考 GNU C 标准库里的解释[7]。

第六节 BitTorrent 协议

BitTorrent 协议（当前版本为 1.0，下面简称 BTP/1.0）是一种通过互联网的分享文件的协议。始于 2002 年。虽然它包含了高度中央化的成分，但还是可以看作一种点对点（P2P）协议。虽然一份正式的，详尽的，完整的协议描述一直欠缺，但是 BTP 本身却已经想当成熟，在不同的平台上都有相应的实现。BTP/1.0 由 Bram Cohen 设计和实现，FTP 实现在某些情况下导致服务器和宽带资源不堪重负，在这种情况下 BTP 作为一种 P2P 协议用以取代 FTP。通常情况下，客户端在下载一个文件的时候不会占用上行带宽。BTP 利用该情况让客户端互相传输一些数据。对比 FTP，这是 BTP 由了巨大的可伸缩性和成本管理的优势。

元数据文件（metadata file）为客户端提供了种子服务器和种子的信息。一般以.torrent 结尾，关联的媒体类型为”application/x-bittorrent”。客户端如何读取元数据文件不在本文的讨论范围内。一般都是在网页上下载元数据文件，然后由用户客户端解析。一个元数据文件中至少包含以下内容，种子服务器的 URL，备用种子服务器的 URL，种子作者的评注，创建该元数据文件的客户端名称与版本，创建日期，下载文件的信息。根据文件数量（单文件种子和多文件种子）下载文件的信息由所不同，这里不具体描述。元数据文件的信息全部用 BT 编码。BT 编码是一种增强的 BNF 语法[8]。

一个种子所有数据都是通过分片和分块传输的。种子被分为一个或多个分片。每片表示了一个范围内的数据，使用分片特征码（SHA1）来验证完整性。当通过 PWP 传输数据时，分片又被分为一个或多个块。如下图所示。

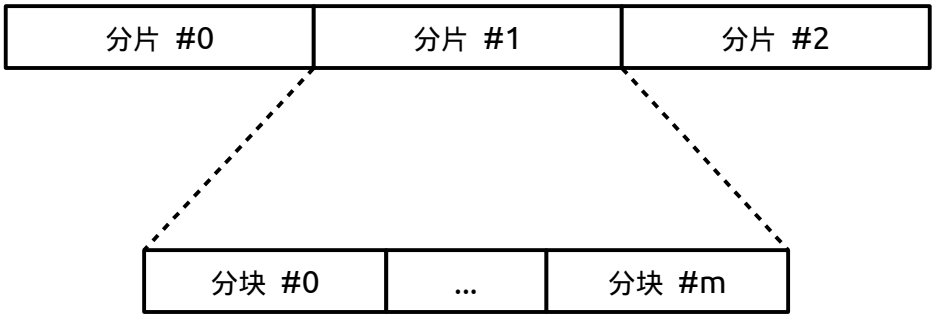

图 3-5 种子分片

元数据文件中指明了种子的分片数量。每片大小时固定的，可以用下面的公式计算：

```
分片大小 = 种子大小 / 分片数量
```

只有最后一个分片可以比固定的大小小。分片的大小由种子的分布者决定。一般使元数据文件的大小不超过 70KB 的分片大小为合适。为了计算一个分片在一个文件或者多个文件中的位置，种子被看作时一个单一的，连续的字节流。如果种子包含多个文件，那种子就被看作这些文件以出现在元数据文件中的顺序的串联。从概念上讲，种子只在最后所有分片都下载完成，而且检验没有问题后，才被转化为一个或者多个文件。但在实际应用中，BT 实现可能会根据操作系统和文件系统的特性，采用一个更好的方式。

块大小是由具体的 BT 实现决定的，与分片大小无关。但一旦大小决定了，分片中块的数量可以用以下公式计算：

```
分块数量 = 分片大小 / 分片块大小 + ! ! ( 分片大小 % 分块大小 )
```

% 是模运算符，！是否定运算符。双重否定运算符是为了保证最后一个因子要么为 0 要么是 1。

BTP/1.0 协议本身分为两个部分，分别为 THP（The Tracker HTTP Protocol）和 PWP（The Peer Wire protocol）。

THP（The Tracker HTTP Protocol）是让节点互相了解的一种机制。一个种子服务器是一个 HTTP 服务器，与节点连接并让节点加入种子群。种子服务器就是 BTP/1.0 中唯一的中心化元素。种子服务器本生并提供任何下载数据。种子服务器依靠节点发送请求。如果节点没有请求，种子服务器会认为节点以'死'。客户端要与种子服务器取得链接，必须向元数据文件中'announce'指定的 URL 发送一个标准的 HTTP GET 请求。GET 请求必须按照 HTTP 协议添加参数。

PWP（The Peer Wire Protocol）的目的是实现种子节点之间的通信，包括数据传输。在节点读取元数据文件并与种子服务器通信后，获取了其他节点的信息，然后使用 PWP 与其他节点通信。PWP 是建立在 TCP 上的，并用异步消息处理所有通信。本地节点会打开一个端口来接听其他节点的连接。这个端口通过 THP 告诉种子服务器。因为 BTP/1.0 没有指定任何端口，BTP 实现负责选择一个端口。想要与本地节点通信的其他节点，必须打开一个连接到此端口的 TCP 连接，并且完成握手操作。握手必须在发送其他任何数据之前完成。如果违反了握手规则（握手失败），本地节点必须关闭与其他节点的链接。

完成 PWP 握手的 TCP 链接两端都有可能向对方发送消息。PWP 消息有两种作用，一就是更新相邻的节点状态，而是传送数据块。PWP 消息可以分为两类，面向状态的消息和面向数据的消息。

面向状态的消息，此类消息用了告知相邻节点状态的改变。在节点状态改变时必须发送这种消息，不管对

方是什么状态，面向状态的消息可以分为几类：Interested，Unintereted，Choked，Unchoked，Have 和 Bitfield。面向数据的消息，此类消息处理消息的请求和发送。可分为三类：Request，Cancel，Piece。

节点状态，对于链接的两端，节点必须维护以下两个状态：

- Choked：设置时，意味着堵塞的节点不允许请求数据。
- Interested：当设置时，意味着节点希望从其他节点那获取数据。这表明节点将开始请求数据，如果不是堵塞的。

一个堵塞的节点不能发送任何面向数据的消息。但是可以发送任何其他消息给其他节点。一个未堵塞的节点可以发送面向数据的消息给其他节点。要如何堵塞，堵塞多少节点和开发多少节点全由具体实现决定。这是故意允许节点采用不同的启发式方法进行节点选择。一个感兴趣的节点，其实就是告诉其他节点它希望尽早地得到面向数据地消息，一旦该节点被开发。必须注意地时，节点不能一厢情原假设其他节点需要它地数据，并对其 interested。也许有一些原因让节点对除了有数据的节点感兴趣。

节点消息，所有在 PWP 消息中的整数成员都是 4 字节大端序的。还有就是所有编号和偏移量都是 0 开始。一个 PWP 消息有如下的机构：

消息长度	消息 ID	负载数据

消息长度：一个表示消息长度的整数，不包括该字段本身。如果有一条消息没有实际负载内容，那么该大小为 1。大小为 0 的消息时可能的，作为保活消息周期性地发送。除了这个限制，消息上都要加上 4 字节，BTP 没有指定一个最大长度。因此 BTP 实现可以选择一个不同地限制，比如可以选择与使用太大消息长度地节点断开链接。

消息 ID：这是一个单字节值，表示了消息地类型。BTP/1.0 定义了 9 种消息类型。

负载数据：这是一个可变长度地字节流。

如果收到地一条消息不遵循该结构，那么该链接应该（SHOULD）被直接丢弃。比如，接收方必须保证消息 ID 是一个合法地值，而负载是期望中的。

为了兼容以后可能的协议扩展，客户端应该忽略未知的消息。当然也有一些情况下客户端在收到未知消息后会选择关闭链接，为了安全考虑或者认为保存大型的未知消息是一种资源浪费。BTP/1.0 定义了以下几种消息：

Choke：该消息的 ID 是 0 并且没有负载。节点发送该消息告诉对方，对方已被堵塞。

Unchoke：该消息的 ID 是 1 并且没有负载。该节点发送该消息告诉对方，对方已被解除堵塞。

Interested：该消息 ID 是 2 并且没有负载数据。节点发送该消息告诉对方，希望获得对方的数据。

Uninterested：该消息 ID 是 3 并且没有负载。节点发送该消息告诉对方，已经不对对方的任何分片感兴趣。

Have：该消息 ID 是 4 并且由一个长度为 4 的负载。负载表示的是节点已经拥有的数据分片的编号。收到该消息的节点必须验证编号，如果编号不在指定范围内则关闭链接。同样的，收到该消息的节点必须发送一个 Interested 消息给发送者如果它确实需要该分片。或者发送一个对该分片的请求。

Bitfield：该消息 ID 是 5 并且由一个变长的负载。负载表示了发送者成功下载的数据分片，第一个字节的高位表示了编号为 0 的分片。如果一个位是 0 说明该发送者没有该分片。节点必须在完成握手后立刻发送该

消息，不能如果一个分片都没有可以选择不发送。在节点之间通信过程的其他时间该消息不能发送。

Request：该消息 ID 是 6 有一个长度是 12 的负载。负载是 3 个整数值，表示了发送者希望获取的分片中的块。接收者只能发送 piece 消息给发送者作为回应。不过要遵循上述的 choke 和 interested 机制。负载有如下结构。负载有如下的结构：

分片编号	块偏移量	块长度

Piece：该消息 ID 是 7 并且由一个变长的负载。负载前两个两个整数指名哪个分片中哪个块，第三个字段表示该块的数据。注意，数据长度可以通过消息总长度减去 9 来获得。负载有如下结构。

分片编号	块偏移量	块数据

Cancel：该消息 ID 是 8 并且有一个长度为 12 的负载。负载是三个整数值，表示了发送者曾请求过但现在不需要了的块编号和所在分片编号。接收者收到该消息后必须去除请求标志。负载有如下结构：

分片编号	块偏移量	块长度

BTP/1.0 没有规定要用什么特定的顺序下载分片。然而，经验表明，按照“最少优先”下载的等待时间是最少的。为了找到最少的分片，客户端必须从所有相邻节点中计算分片编号二进制位为 1 时的分片。和最小的分片就是最少分片，应该优先请求。

参考文献

[1] W.Richard Stevens. TCP/IP Illustrated Volume 1: The Protocol. Addison Wesley, 1994.

[2] W.Richard Stevens, Stephen A Rago. 尤晋元，张亚英，戚正伟 译。 Advanced Programming in the UNIX Environment. UNIX 环境高级编程。人民邮电出版社，2006 年 5 月 1 日。

[3] Danny Cohen. ON HOLY WARS AND A PLEA FOR PEACE. http://www.ietf.org/rfc/ien/ien137.txt. 1 April 1980.

[4] J. Postel, J. Reynolds. FILE TRANSFER PROTOCOL (FTP). http://www.ietf.org/rfc/rfc959.txt. October 1985.

[5] R. Fielding, UC Irvine et al. Hypertext Transfer Protocol – HTTP/1.1. http://www.ietf.org/rfc/rfc2616.txt. June 1999

[6] Wikipedia. POSIX Threads. https://en.wikipedia.org/wiki/POSIX_Threads.

May 2014.

[7] GNU libc manual. Thread-specific Data. https://www.gnu.org/software/libc/manual/html_node/Thread_002dspecific-Data.html. May 2014.

[8] Wikypedia. Backus-Naur Form. https://en.wikipedia.org/wiki/Backus-Naur_form. May 2014.