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

指导教师　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

本文从讨论了 Socket 网络编程的方法，以及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）。同时描述了文件传输管理器以及一个简单的 HTTP 服务器的设计与实现。文件传输管理器支持 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。而简单的 HTTP 服务器只实现 HTTP200，HTTP404 和 HTTP206，支持断点续传，可以与文件传输管理器很好地配合。两者都是在 Linux 完全采用 C 语言实现。

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 and simple HTTP server are described in the document. The file transfer manager supports FTP, HTTP and BitTorrent Protocol, multi-task, multi-thread and breakpoint-resume. It's OK to download files from the Internet using it. And the simple HTTP server, which only implements HTTP200, HTTP206 and HTTP404, and breakpoint-resume too.They two can work well together, and are both writen in C language in Linux System.

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

第一章 绪论

第一节 课题背景与意义

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 是作为操作系统网络协议栈对外的接口，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，详细可以参考 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。

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。

第二节 研究内容

文件传输作为互联网最基本的功能之一，大部分都是基于 Socket 网络编程的。本文 Socket 的网络编程的角度讨论文件传输协议，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。

现有的大部分文件传输软件都只支持单一的文件传输协议，比如 Filezilla 只支持 FTP，而 uTorrent 只支持 BitTorrent 协议，HTTP 一般只有浏览器支持。本文运用现有的开源软件库在同一个软件中实现多种协议的文件传输协议，并用统一的方式管理下载、暂停。为了能更好地体现 Socket 网络编程的方法，还实现了一个简单的 HTTP 服务器程序，该程序没有依赖任何第三方库。

第一章 系统的实现技术

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。Linux 对 Socket 有很好地继承，并且提供了原生 C 语言的接口，为了能更好地体现 Socket 网络编程的概念，文件传输管理器和简单的 HTTP 服务器都在 Linux 下用 C 语言实现。

第一节 开发环境

- 操作系统：Fedora 20 linux3.14 x86_64，Linux 系统对 Socket 接口有良好的支持。
- 编程语言：C，操作系统以 C 函数形式提供 Socket 接口。
- 编译器：GCC-4.7，广泛使用的开源 C 编译器。
- 依赖：libgtk+、libcurl、libtransmission。libgtk 是 linux 系统下流行的图形界面库；libcurl 主要功能就是用不同的协议连接和沟通不同的服务器，支持 FTP 和 HTTP；libtransmission 是 C 语言实现的 BT 协议的客户端库。

第二节 使用 C 语言的原因

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，使用 C 语言可以更好得理解网络编程的概念与方法。

其次，C 语言的效率很高，可以很好地保证程序高效运行、及时的响应。但是另一方面，C 语言本身提供的特性很少，需要程序员对数据结构内存管理有比较深的理解才能很好地使用。

第二节 第三方库

文件传输管理器主要依赖四个第三方库，本章节对其进行简单描述。

2.1 libgtk+-3.0

GTK+ 是一种图形用户界面（GUI）工具包。也就是说，它是一个库（或者，实际上是若干个密切相关的库的集合），它支持创建基于 GUI 的应用程序。可以把 GTK+ 想像成一个工具包，从这个工具包中可以找到用来创建 GUI 的许多已经准备好的构造块。

最初，GTK+ 是作为另一个著名的开放源码项目 —— GNU Image Manipulation Program (GIMP) —— 的副产品而创建的。在开发早期的 GIMP 版本时，Peter Mattis 和 Spencer Kimball 创建了 GTK（它代表 GIMP Toolkit），作为 Motif 工具包的替代，后者在那个时候不是免费的。（当这个工具包获得了面向对象特性和可扩展性之后，才在名称后面加上了一个加号。）

GTK+虽然是用 C 语言写的，但是您可以使用你熟悉的语言来使用 GTK+，因为 GTK+已经被绑定到几乎所有流行的语言上，如：C++,PHP, Guile,Perl, Python, TOM, Ada95, Objective C, Free Pascal, and Eiffel。

2.2 libcurl

LibCurl 是免费的客户端 URL 传输库，支持 FTP,FTPS, HTTP, HTTPS, SCP, SFTP, TFTP, TELNET, DICT, FILE ，LDAP 等协议，其主页是 http://curl.haxx.se/。Libcurl 具备线程安全、IpV6 兼容、易于使用的特点。

2.3 libtransmission

libtransmission 是 linux 下流行的 BT 客户端程序 transmission 所使用的后端，它使用 C 语言实现，被设计与图形界面无关。因此 transmission 程序也就可以轻松地移植到更多平台。而且 libtransmission 是实现了多线程处理，它会自己创建一个线程来完成 BT 协议的内容，而使用者无需关心具体实现。

第二章 文件传输管理器的设计

本章节描述文件管理器的设计，包括功能设计，界面设计、模块划分，和简要的实现描述。文件传输管理器提供给用户文件传输的基本功能。第三章描述具体的实现细节。

第一节 功能分析

首先文件传输管理器要支持多种协议，包括 FTP、HTTP 和 BitTorrent 协议。因为每种协议有着不同的特定，实现可能会很不一样，但是在界面上对用户提供一致的接口。比如同样的工具栏菜单，同样或者略微不同的菜单。

在讨论具体的功能实现之前，我们先对各个协议做简单的描述，分析每个协议的特点，对于理解实现有很大帮助。

HTTP 协议简介

HTTP 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[1]。

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:

1. 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。

2. 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。

3. 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。

4. 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。

5. 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。

HTTP 协议的功能设计

HTTP 协议本身是为了传输超文本而设计的，但事实上超文本其实就是一个文件，因此 HTTP 本身就是适合文件传输的，而且有很大的灵活性。但是 HTTP 为了适应互联网的发展，其中包含了很多与网站相关的特定内容，比如 Host、Language 等首部字段，以及 PUT，TRACE 等请求方法，还有一些与代理，缓存，Cookie 相关的内容，在文件传输管理器中是不需要的。下面列出了在文件传输管理中需要实现的 HTTP 功能。

1. GET 方法：在文件传输管理器中我们只实现 HTTP 的 GET 方法，

这是 HTTP 用于文件传输的典型请求方法，PUT，POST 等都是不需要的。

2. 断点续传：同时通过制定请求中的 Range 首部字段来设置断点传输，实现断点续传和宕机恢复的效果。

3. 响应状态码：文件传输管理器不去区分 HTTP 请求的各种错误，因此可以简单的根据服务器返回的状态码是否大于等于 400 判断请求是否成功。如果状态码大于等于 400 则表示该请求是无效的，自然无法下载文件。否则是有效的请求。

4. URL 重定向：如果服务器返回的是 3xx 的状态码，我们转到相应的 URL 重新请求，并且对用户隐藏此操作，对于用户来说根本就不知道 URL 重定向这件事。

FTP 协议简介

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[2]。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。

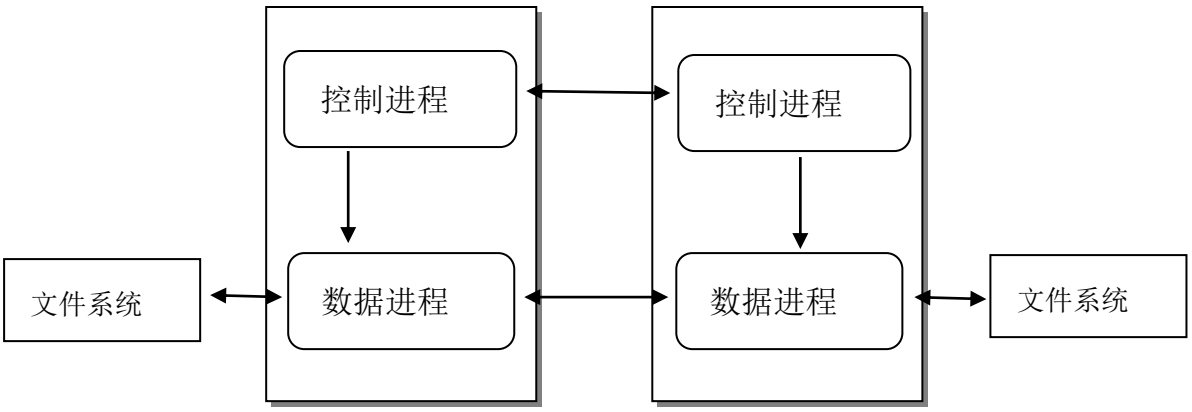

图 2-1 FTP 通信模型

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。

FTP 协议的功能设计

FTP 是专门为了文件传输设计的，它是典型的客户端服务器模型。但是 FTP 协议提供了诸如身份认真、文件上传、目录管理等功能在文件传输管理器中是没有必要的。在文件传输管理器中我们只处理无效认证的请求，如果服务器要求用户进行身份认证，那就简单的报错。同时文件传输管理器不提供上传文件、目录管理等功能，它只实现了简单的文件下载功能。下面列出了文件传输管理器中所实现的 FTP 协议的内容。

1. RETR：RETR 是客户端向服务器请求一个文件的命令，用于下载文件。

2. REST：REST 也是客户端向服务器请求一个文件的命令，不同于 RETR 的是，它指定了文件的断点。用此命令文件传输管理器可是实现如 HTTP 中 Range 一样的断点续传。

3. 错误处理：同 HTTP 一样，在文件传输管理器中我们不去处理任何错误，如果发生了错误，只是简单地将错误信息告诉用户。在 FTP 中，我们认为响应代码不为 200 都是出错。

BitTorrent 协议简介

BitTorrent 协议[3]（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。

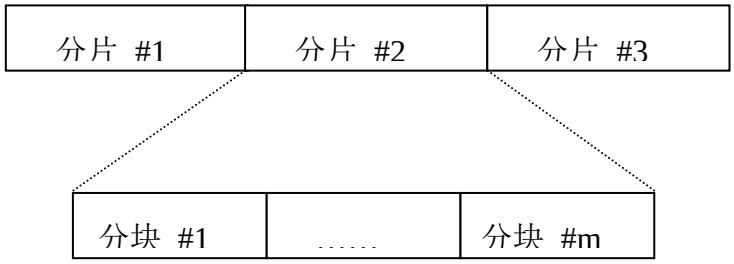

图 3-5 种子分片

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。

BitTorrent 协议的功能设计

BitTorrent 协议本身比较复杂，但都只为一个目的，那就是传输文件。因为文件传输管理器采用了 libtransmission 作为 BT 协议的后端，而 libtransmission 已经把 BT 协议封装得很好了。因此，文件传输管理器几乎不要关心 BT 协议的实现细节。下面列出了文件传输管理器需要完成的 BitTorrent 功能。

1. 文件传输：这涉及到大部分 BitTorrent 协议的内容。包括 THP 和 PWP 协议。

2. 断点续传：对于 BT 协议来说，种子的数据本来就是分片分块传输的，因此断点续传根本就不是问题。事实上 BT 协议本来就是被设计来完成断点续传的。因为大部分通过 BT 协议传输的文件都比较大。

3. 种子信息：对于一个 BT 下载任务来说，它有更多用户感兴趣的信息，比如种子服务器的 URL 地址，比如种子中文件数量以及各个文件名。文件传输管理器将把这些信息显示给用户。

4. 文件选择：很多种子都包含有多个文件，应该提供给用户选择下载文件的能力。用户可以选择性地下载一个种子中的一个或者多个文件。

第二节 界面设计

文件传输管理器的界面采用 GTK+实现，因此在设计界面的时候必须要考虑 GTK+本身的特性。GTK+是 GNOME 基金会开发的，GNOME 基金会还定义了一些界面开发规范，文件传输管理器的界面设计尽量符合该规范的要求。下面是文件传输管理的界面设计。

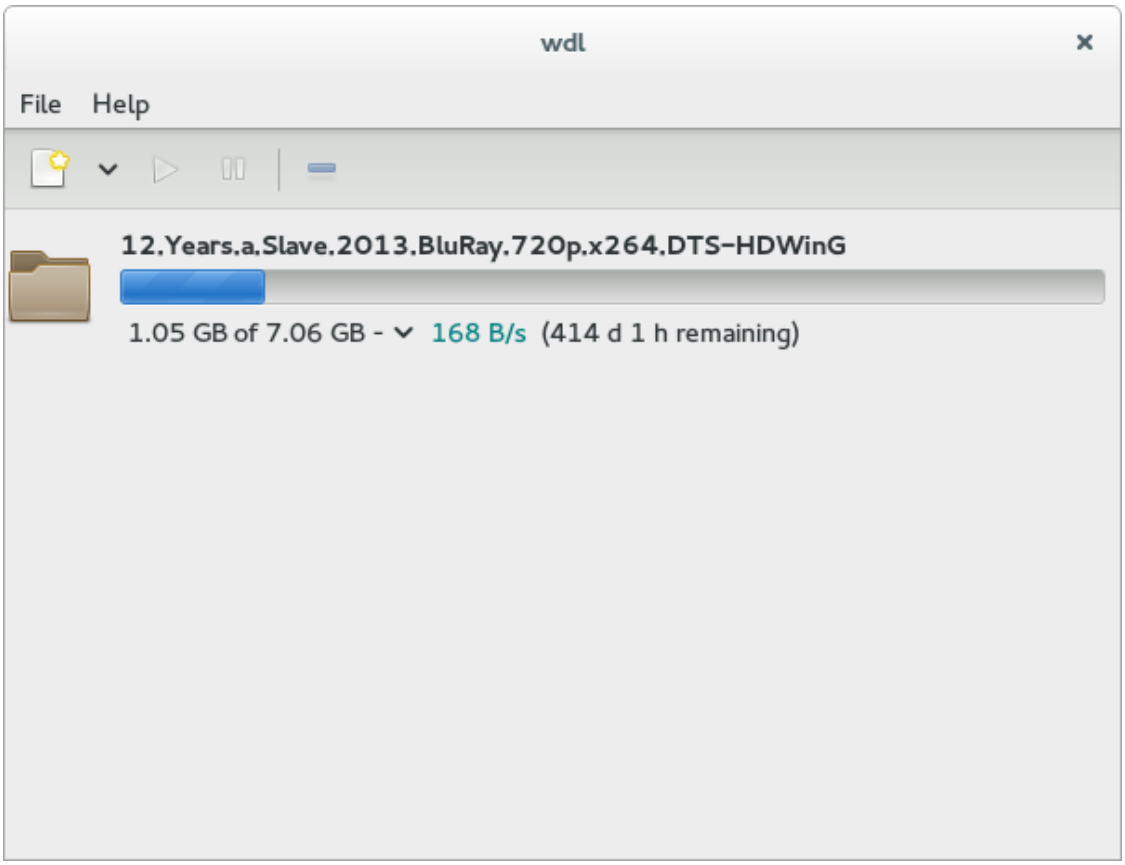

图 2-2 文件传输管理器的主界面

界面主要分为三个部分，菜单栏、工具栏和人物界面。菜单栏只提供了两个菜单项，一个 File 菜单项提供了新建任务的入口。一个 Help 提供了查看软件信息的入口。

工具栏同样有一个新建任务的按钮，作为快键方式存在。然后是控制

下载任务的开始、暂停、删除按钮。这几个按钮能作用于任何下载任务。这也是上一章所说的统一管理方式。

最后是任务界面，任务界面按行排列各个任务，每个任务都有显示下载状态，包括下载速度、文件总大小、已下载大小和剩余时间等。而且这些信息对于不同任务（FTP、HTTP 或者 BT）来说都是一样显示的。

第三节 模块划分

文件传输管理器首先按照协议来划分模块，把协议特定的实现划分为单独模块并且封装成一个类来使用。但是在考虑模块划分时还要考虑具体的实现。比如 FTP 和 HTTP 都是采用 libcurl 的，因此把 FTP 和 HTTP 的实现划分到同一个模块中。而 BT 采用 libtransmission，则划分为单独的模块。下面列出文件传输管理器中各个主要模块。

WlHttper	该模块管理一个 HTTP 或者 FTP 的下载任务，包括界面显示，任务的管理（暂停、下载、重新开始等）。下载任务在一个新线程中执行，WlHttper 完成对线程的管理。
WlBter	与 WlHttper 类似，但是它管理的是一个 BitTorrent 的下载任务，wdl 采用了 libtransmission 实现 BT 下载，而 libtransmission 实现了多线程的控制；因此 WlBter 中只是调用了 libtransmission 提供的管理接口，没有实现具体的多线程控制。
WlDownloader	这个是下载任务的管理器，负责管理一个或者多个 WlHttper 和 WlBter，并对用户提供一致的管理接口。
WlDownloadWindow	wdl 的主窗口，负责窗口菜单，以及 WlDownloader。是 wdl 程序的主界面。
WlUrlDialog	这是一个对话框，当用户点击新建下载任务后弹出该对话框，用户输入下载的 URL 以及保存的文件路径。WlUrlDialog 还会对 URL 的有效性做一次验证。
WlBtFileChooser	当用户选择了一个种子元数据文件后，打开该对话框，显示种子的一些信息。用户可以选择下载种子中的哪些文件。种子文件的保存位置。

下图表示了各个模块子间的层次关系，箭头表示控制关系。

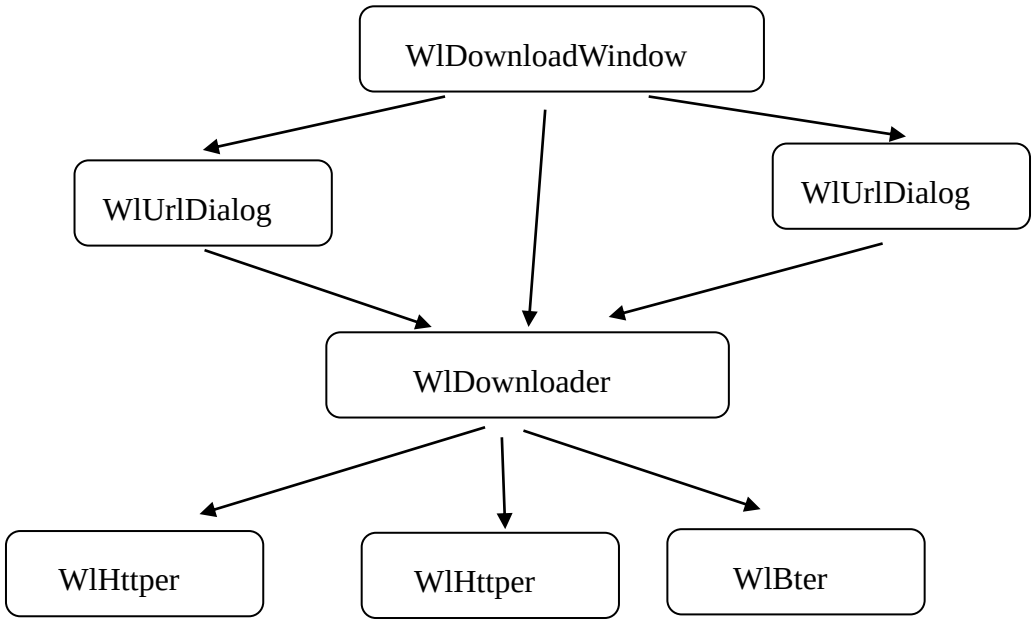

图 2-3 各模块层次

整个 wdl 程序中还包括一些其他模块，比如表示 HTTP 或者 FTP 下载任务右键菜单的模块 WlHttperMenu 和表示 BT 下载任务右键菜单的 WlBterMenu，以及显示一个 BT 任务属性的 WlBterProperties。不过这些模块并不是重点，本文中不详细描述。下一节我们详细描述本节中列出的几个模块。

第三章 文件传输管理系统的实现

本章节描述文件传输管理器的实现，以各个模块的具体实现作为切入点展开。本章重点讨论文件传输管理器的重点模块的实现。

第一节 HTTP、FTP 下载实现

如第二章所描述的那样，HTTP 和 FTP 是放在同一个模块 WlHttper 中实现的，因为他们都采用了 libcurl 作为后端。在描述文件传输管理器如何使用 libcurl 之前，先看一下一个简单的 libcurl 例子，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;
}
```

libcurl 的使用很简单，首先调用 curl_easy_init 创建一个 CURL 对象，然后使用 curl_easy_setopt 设置属性，包括要请求的 URL 地址。设置完后便可以调用 curl_easy_perform 进行网络数据传输。完成数据传输后调用 curl_easy_cleanup 释放资源。注意 libcurl 没有提供线程支持，只是提供了线程安全。也就是说，多线程控制需要使用者自己实现。下面描述文件传输管理器中是如何使用 libcurl 来完成 FTP 和 HTTP 文件传输的。

WlHttper 基于 GtkEventBox，GtkEventBox 是一个 GTK+图形界面的容器类，同时可以接受各种界面信号（鼠标点击等）。我们只讨论 WlHttper 的功能实现，而不讨论具体的界面构造，除非是在涉及到界面显示的功能上。

当使用 wl_httper_new()创建了一个新的 WlHttper 实例时，只是在 GTK+主线程中显示了一个新的下载任务，并没有启动新线程；知道 wl_httper_start()被调用才会创建一个新线程开始下载。也就是说 WlHttper 本身是处在 GTK+的主线程中的，无论创建了多少个 WlHttper 都是运行在 GTK+的主线程中的。只是不同的 WlHttper 分别管理了各自的下载线程。下面是 WlHttper 的实例结构，

```
struct _WlHttper {
GtkEventBox parent;
GtkWidget *iconImage;
GtkWidget *titleLabel;
GtkWidget *progressBar;
GtkWidget *dlLabel;
GtkWidget *totalLabel;
```

```
GtkWidget *speedLabel;
GtkWidget *timeLabel;

GtkIconSize iconSize;
gchar *url;
/* 保存路径和文件流 */
gchar *savePath;
GFileOutputStream *fOutput;
GThread *thread;
CURL *easyCURL;
/* 下载速度 */
gdouble speed;
/* 下载完成百分比 */
gdouble percentDone;
/*
* 这两个字段用来计算下载速度
* dlData 表示在 dlTime 时间内下载的字节数
*/
guint64 dlData;
guint64 dlTime;
/* */
guint timeout;
/* 已下载的数据和总数据 */
guint64 dlNow;
guint64 dlTotal;
/* 当前的状态 */
gint status;
/* 完成后返回结果 */
GAsyncQueue *rqueue;
/* 发送取消命令 */
GAsyncQueue *cqueue;
/* 创建时间，不可设置，初始化自动填写 */
GDateTime *cdt;
/* 右键菜单 */
GtkWidget *popMenu;
```

```
/* 用户自定义数据 */
gpointer userData;
/* 完成下载后的回调函数,第一个参数是下载结果 */
WlHttperCallback finishCallback;
gpointer cbData;
/* 状态改变的回调函数 */
WlHttperStatusCallback statusCallback;
gpointer sData;
};
```

其中 status 表示 WlHttper 当前的下载状态，是一个如下的枚举类型，每个状态分别对应一种界面的显示。

```
enum _WlHttperStatus {
WL_HTTPER_STATUS_START = 11111, /* 下载任务已经开始 */
WL_HTTPER_STATUS_PAUSE = 22222, /* 暂停 */
WL_HTTPER_STATUS_COMPLETE = 33333, /* 下载已经完成 */
WL_HTTPER_STATUS_NOT_START = 44444, /* 下载还未开始 */
WL_HTTPER_STATUS_ABORT = 55555, /* 下载被中止了 */
};
```

url 就是下载的文件的 URL，savePath 表示该下载任务的保存路径。

rqueue 和 cqueue 都是用于多线程数据通信的。cqueue 由主线程向下载线程发送取消消息。rqueue 由下载线程向主线程发送下载结果。

speed、percentDone、dlNow、dlTotal 分别表示下载速度、下载完成百分比、当前下载的数据量、文件总大小。这些数据都由下载线程设置，然后由主线程读取然后在界面上显示。

当新创建一个 WlHttper 实例时，只是将所有字段，包括界面都初始化，status 被初始化为 WL_HTTPER_STATUS_NOT_START。

wl_httper_start()（源代码：src/wlhttper.c:681）开始下载任务。该函数首先检查当前的状态，如果下载状态处于 WL_HTTPER_STATUS_START 或者是 WL_HTTPER_STATUS_PAUSE 则直接退出。

然后打开保存文件的路径，如果无效则退出。

然后创建一个 CURL 对象，并设置必要的参数。如下

```
CURL *easyCURL = curl_easy_init();
/* 设置 URL */
```

```
curl_easy_setopt(easyCURL, CURLOPT_URL, httper->url);
/* 关闭详细信息 */
curl_easy_setopt(easyCURL, CURLOPT_VERBOSE, 0L);
/* 自动重定向 */
curl_easy_setopt(easyCURL, CURLOPT_FOLLOWLOCATION, 1L);
/* 设置接收到数据后的回调函数 */
curl_easy_setopt(easyCURL, CURLOPT_WRITEFUNCTION,
on_curl_write_callback);
/* 设置进度回调函数 */
curl_easy_setopt(easyCURL, CURLOPT_NOPROGRESS, 0L);
curl_easy_setopt(easyCURL, CURLOPT_PROGRESSFUNCTION,
on_curl_progress_callback);
/* 启用SSL */
curl_easy_setopt(easyCURL, CURLOPT_USE_SSL, CURLUSESSL_TRY);
/* 关闭证书认证 */
curl_easy_setopt(easyCURL, CURLOPT_SSL_VERIFYPEER, 0L);
/* 设置用户代理,冒充FIREFOX */
curl_easy_setopt(easyCURL, CURLOPT_USERAGENT,
"Mozilla/5.0 (X11; Ubuntu; Linux i686; rv:24.0)"
" Gecko/20100101 Firefox/24.0");
/* 关闭信号，这可能会引发一个BUG */
curl_easy_setopt(easyCURL, CURLOPT_NOSIGNAL, 1L)
```

其中 on_curl_progress_callback()是下载任务执行过程中的进度函数，如果返回一个非 0 的值，就会中止下载任务。该函数在下载线程里被调用，但是主线程通过消息通信，在必要时候使其返回 0 中止下载。

其次是 on_curl_write_callback()在下载任务有数据传输时被调用，用来指定数据的保存数据，同时根据数据的量和两次调用之间的时间间隔计算下载速度。

完成 CURL 的初始化后，创建两个新的 GAsyncQueue 用于线程通信，也就是 rqueue 和 cqueue。这表明 rqueue 和 cqueue 具有“任务唯一性”。也就是说每次重新开始下载所使用的 GAsycnQueue 都是不同的。

在然后初始化信息，wl_httper_set_default_data()初始化所有下载相关的字段，比如下载速度 speed。

wl_httper_set_default_info()初始化界面，使界面显示开始下载。

最后调用 g_thread_new("http-download",wl_httper_download_thread, httper);创建下载线程。

接着调用 wl_httper_add_timeout(httper)在主线程中创建一个定时器，周期性地更新 GTK+界面，保证界面显示的下载速度、下载完成百分比等有效。这个定时器的事件周期是 300ms。下面将具体讨论下载函数 wl_httper_download_thread()和定时器函数 wl_httper_download_timeout()。

wl_httper_download_thread()接受一个参数，就是 WlHttper 的指针。通过该参数先取得 rqueue 和 cqueue 的指针。并增加引用计数，确保它们在线程执行期间不会被释放。然后调用 curl_easy_perform()开始下载任务。该函数在执行期间会阻塞，但是内部会不断调用前面注册的 on_curl_progress_callback()和 on_curl_write_callback()。

在 on_curl_progress_callback()内部会不断检查是否有来自 cqueue 的消息，如果有（不管是什么），就认为是主线程发送了取消信号，则返回 1，否则返回 0。on_curl_progress_callback()返回 1 后 curl_easy_perfrom 就会返回一个 CURLE_ABORTED_BY_CALLBACK。

在 on_curl_write_callback()除了将下载的数据保存到文件，还要计算下载速度。使用以下代码计算下载速度，结果保存在 httper->speed 中。其中 httper->dlTime、httper->dlData 都是初始化为 0 的。

```
guint64 now = time(NULL);

if (httper->dlTime != 0) {

httper->dlData += len;

if (now - httper->dlTime > 0) { /*  两次调用的时间间隔至少大于 0 */

httper->speed = (gdouble) httper->dlData / (gdouble) (now - httper->dlTime);

httper->dlTime = now;

httper->dlData = 0;

}

} else /*  这是第一次调用该函数  */

httper->dlTime = now;
```

curl_easy_perform()如果没有被中止或者发生什么错误，那么返回

CURLE_OK。返回后，根据返回值给主线程发送结果。

```
if (code == CURLE_OK)

g_async_queue_push(rqueue, (gpointer) WL_HTTPER_STATUS_COMPLETE);

else

g_async_queue_push(rqueue, (gpointer) code);
```

要么成功，要么失败。最后释放对 cqueue 和 rqueue 的引用。退出线程。

wl_httper_download_timeout()由主线程周期性（300ms）调用。首先，它会检查是否有通过 rqueue 传递的数据，如果有表明下载任务已经结束。如果有了下载结果，那么根据下载结果更新界面，完成，被取消，还是出错？如果没有下载结果，也就是说下载还没有结束，那么更新下载速度、下载进度等信息。

```
wl_httper_set_dl_speed(httper, httper->speed);

wl_httper_set_dl_size(httper, httper->dlNow);

wl_httper_set_total_size(httper, httper->dlTotal);
```

因为创建线程时传递的就是 WlHttper 的指针，所以其实 WlHttper 结构是下载线程和主线程的一块共享内存。既然是共享内存，那么必须考虑竞争。rqueue 和 cqueue 本身就是线程安全的，再通过增加引用计数的方式保证了安全性。而 speed 和 percentDone 等字段，除了初始化（在创建线程之前）外，在主线程中就同上述代码一样，只读不写；也就是说，它们只会在下载线程（on_curl_write_callback）中被改变，因此实际上不存在竞争关系。读写之间可能会有一点点的竞争关系，不过这出现的概率非常小，而且就算出现也根本不影响程序的正确性和稳定性。

下图表示了下载线程和主线程之间的关系。图中框大小与程序执行时间长度没有任何关系。回想前文关于 GObject 中主循环的讨论，所谓的定时器其实就是循环当中的一个执行任务。

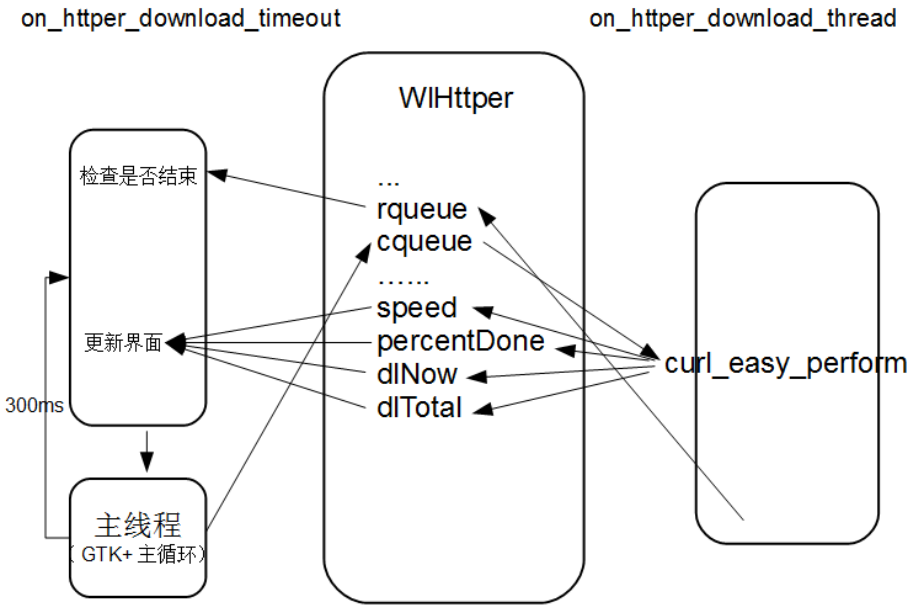

图 4-2 WlHttper 线程通信

上图显示的只是单个任务的情况下主线程与下载线程之间的关系。下图表示了多个任务同时进行时两者的关系。记住，无论多少个 timeout，都是在主线程中调用的，而主线程一直处在一个主循环中。我们无法假设多个 timeout 的调用顺序，但是他们一定是先后调用而不会嵌套。

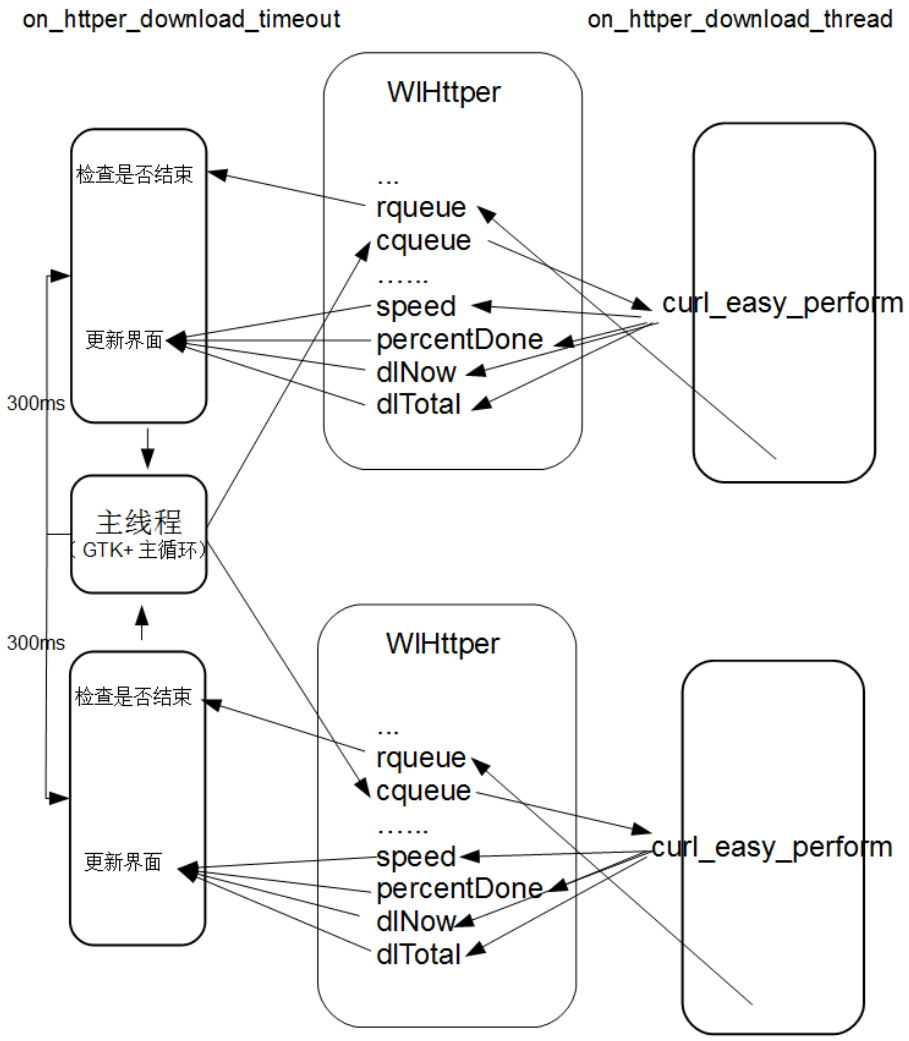

图 4-3 多任务

因为 libcurl 本身是线程安全的，因此在主线程中调用 curl_easy_pause()来暂停下载没有任何问题。wl_httper_pause()主要调用 curl_easy_pause()暂停下载，并且移除 on_httper_download_timeout()定时器。重新开始下载后再添加 on_httper_download_timeout()定时器。

wl_httper_contitnue()再次用不同的参数调用 curl_easy_pause()继续下载。同时重新添加 on_httper_download_timeout()定时器。

wl_httper_abort()中止下载，就像上图所表示的一样，中止下载任务可以在主线程的任何地方调用。这会通过 cqueue 发送取消的消息给下载线程。下载线程将在 on_curl_progress_callback()或者是 on_curl_write_callback()中收到主线程的取消命令。

```
g_async_queue_push(httper->cqueue, (gpointer) 1); /* 取消下载任务 */
```

```
    g_async_queue_unref(httper->cqueue);
    g_async_queue_unref(httper->rqueue); /* 直到下载线程退出 rqueue 和 cqueue 才会真正释放
*/
    httper->cqueue = NULL;
    httper->rqueue = NULL;
```

wl_httper_redownload()执行重新下载任务。该操作只是先后调用 wl_httper_abort()和 wl_httper_start()。

最后下载任务完成后下载线程退出，wl_httper_download_timeout()定时器也被移除。

WlHttper 还提供了注册改变状态回调函数和下载结束回调函数的功能。分别对应 WlHttper 结构中的 statusCallback 和 finishCallback。如果设置了这些回调函数。那么在特定的地方就会被调用。以下载结束回调函数为例。在 on_httper_download_timeout()中如果检查到下载结束，就会执行下面的代码，

```
if (httper->finishCallback)
httper->finishCallback((gulong) popdata, httper->cbData);
```

状态改变回调函数的调用类似。分别使用 wl_httper_set_callback()和 wl_httper_set_status_callback()设置下载结束回调函数和状态改变回调函数。下载状态改变回调函数主要用在让主窗口可以检查到当前任务的状态，以改变工具栏按钮的形式。

第二节 BT 下载实现

WlBter 处理一个 BT 下载任务，包括下载控制以及图形界面的显示；基本结构与 WlHttper 类似，但是 WlBter 并不管理一个线程，BT 下载统一由 libtransmission 管理。WlBter 只是调用了 libtransmission 提供的接口。更新界面的方式与 WlHttper 类似，都是通过周期性的回调函数来完成。

WlBter 的结构体如下，

```
struct _WlBter {
GtkEventBox parent;
/* GUI */
GtkWidget *icon;
GtkWidget *titleLabel;
GtkWidget *dlLabel;
GtkWidget *totalLabel;
GtkWidget *speedLabel;
```

```
GtkWidget *timeLabel;
GtkWidget *progressBar;
/* Menu */
GtkWidget *popMenu;

guint64 totalLast;

/* libtransmission 的会话，由外部传递进来 */
tr_session *session;
/* torrent 的指针，可以是外部传递，也可以是自身创建 */
tr_torrent *torrent;
/* 当前的下载状态 */
WlBterStatus status;
/* 定时器 */
guint timeout;

/* 状态改变回调函数 */
WlBterStatusCallback statusCB;
gpointer statusCBData;
/* */
gpointer userData;
};
```

除去界面元素，主要的是 tr_session 对象 session，该对象是外部传递的指针，真正由 WlDownloader 管理。所有的 WlBter 都共用一个 tr_session 对象。同样的 tr_torrent 也是由外部传递的，由 WlBtFileChooser 真正创建然后传递给 WlDownloader，WlDownloader 再根据这个 torrent 和自身的 session 创建一个 WlBter 任务，并管理该任务。

定时器 timeout 使用方式和 WlHttper 中一样。

因为下载完全由 libtransmission 控制，因此，WlBter 只是在 libtransmission 的基础上封装，同时加上了图形界面。下面是周期性更新界面的函数代码；

```
static gboolean wl_bter_timeout(gpointer data)
{
WlBter *bter = WL_BTER(data);
tr_torrent *torrent = bter->torrent;
```

```
const tr_stat *stat = tr_torrentStatCached(torrent);


if(stat->error!=0) {
g_message("bt error");
wl_bter_set_status (bter,WL_BTER_STATUS_ABORT);
return FALSE;
}


/*  下载百分比  */
gtk_progress_bar_set_fraction(GTK_PROGRESS_BAR(bter->progressBar),
stat->percentDone);
/*  下载速度  */
wl_bter_set_dl_speed(bter, stat->pieceDownloadSpeed_KBps);
/*  已下载数据和总数据  */
wl_bter_set_total_size(bter, stat->sizeWhenDone);
wl_bter_set_dl_size(bter, stat->haveValid+stat->haveUnchecked);
/*  剩余时间  */
wl_bter_set_rtime(bter, stat->pieceDownloadSpeed_KBps,
stat->leftUntilDone);
if (stat->percentDone == 1.0) {
g_message("complete");
wl_bter_set_status(bter, WL_BTER_STATUS_COMPLETE);
return FALSE;
}
return TRUE;
}
```

主要就是通过 tr_torrentStatCached()函数获取当前种子的状态信息，下载速度、下载进度之类的；进行适当的处理后在界面上显示。如果任务已经完成了，那么就通过返回 FALSE 结束周期性地调用。

下图表示了 WlBter 与 tr_session 线程的关系。

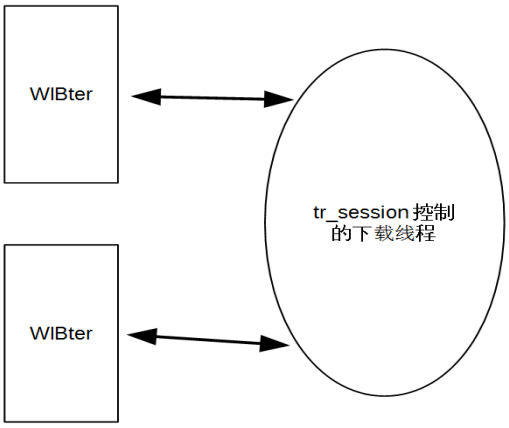

图 4-4 WlBter 的线程模型

再结合前面的 WlHttper，一个完整的程序线程结构大概是这样的。

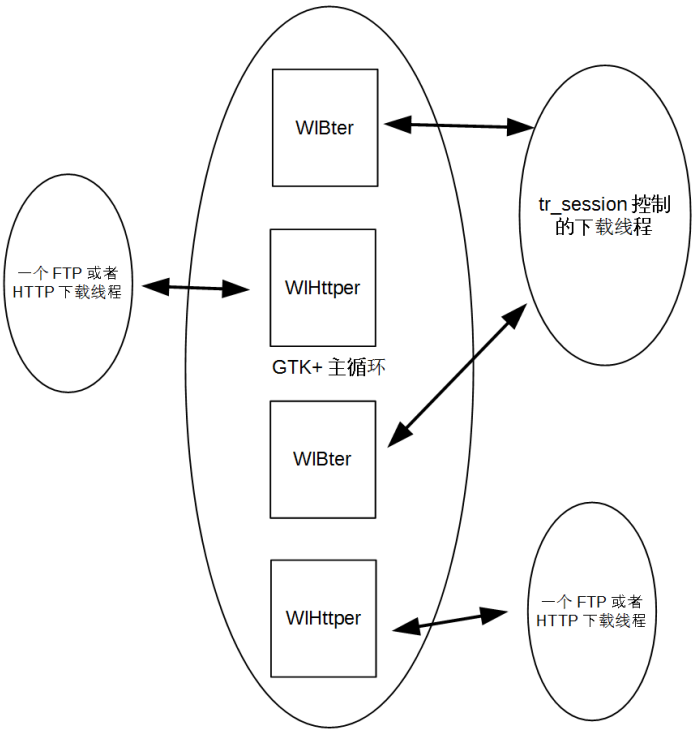

图 4-5 wdl 线程模型

第三节 下载管理实现

WlDownloader 继承自 GtkScrolledWindow。内部使用 GtkBox 管理多个任务。包括 WlHttper 和 WlBter。并对外提供统一的管理方式。它维护一

个当前选中的任务，选中的任务会高亮显示。

WlDownloader 维护一个 tr_session 对象，管理所有 BT 任务。

很多函数都是对当前选中任务的操作，比如

```
/*
* @description  开始选中的下载任务
*/
void wl_downloader_start_selected(WlDownloader * dl);
/*
* @description  暂停选中的下载任务
*/
void wl_downloader_pause_selected(WlDownloader * dl);
/*
* @description  继续选中的下载任务
*/
void wl_downloader_continue_selected(WlDownloader * dl);
/*
* @description  获取选中任务的状态
* @return  如果没有选中，返回 0
*/
WlHttperStatus wl_downloader_get_selected_status(WlDownloader * dl);
```

wl_downloader_set_selected_callback()提供了当选中任务改变时的回调函数，实现与 WlHttper 的回调函数类似。

```
if (dl->httperSelected)

dl->httperSelected(dl, dl->httperSelectedData);
```

WlDownloader 管理多个下载任务，主要是 GTK+编程。

第四节 新建下载实现

提供一个让用户输入下载 URL 的对话框。主要是 GTK+编程。但是每次按下确认按钮后，会先使用 libcurl 来确认输入的 URL 是否有效，如果无效则提示，有效则新建一个下载任务。本节主要描述如何确认一个 URL 是否有效。

要确认一个URL是否有效必须对该URL发起一个HTTP的GET请求，

然后根据 HTTP 的响应来确认是否有效。回想 HTTP 响应码，我们认为 HTTP 响应码大于 400 则表面请求失败，否则是成功的。因此我们只需要对输入的 URL 发起一个 HTTP 请求，然后获取 HTTP 响应码，这里不需要获取任何实际的 HTTP 正文。

因为 curl_easy_perform()是阻塞的，为了保证界面其他事件的及时响应，采用多线程的方式。

简单地说就是创建一个线程对指定的 URL 发起一个 HTTP 请求，但是获取 HTTP 响应码后就立刻结束，然后根据 HTTP 响应码判断 URL 是否有效。

check_url_thread()就是线程的执行函数，

```
CURL *curl = curl_easy_init();
curl_easy_setopt(curl, CURLOPT_URL, url);
curl_easy_setopt(curl, CURLOPT_WRITEFUNCTION, curl_write_function);

CURLcode ret = curl_easy_perform(curl);
glong code;
curl_easy_getinfo(curl, CURLINFO_RESPONSE_CODE, &code);
curl_easy_cleanup(curl);
```

回调函数 curl_write_function 只是单纯地返回 0 表示中止数据传输，然后 curl_easy_perform()，libcurl 保证调用 curl_write_function()时一定已经接受了 HTTP 响应，curl_write_function()只会被调用一次。然后使用 curl_easy_getinfo()获取 HTTP 的响应码。

第五节 界面主窗口实现

WlDownloadWindow 是 wdl 程序的主窗口类，主要就是显示窗口界面、菜单项以及工具栏显示。

WlDownloadWindow 其实只是作为一个界面存在，它没有实现任何实际的功能，但是为用户提供了控制程序的能力，比如工具栏和菜单栏。用户可以通过工具栏控制下载任务开始、暂停、删除，或者是新建一个任务。同时也将当前任务的状态用工具栏显示，比如，如果选中的任务是正在下载，那么，工具栏中开始按钮设置为不可用，而暂停按钮为可用；反之则开始按钮可用，暂停按钮不可用。

下面是更新工具栏状态的实现代码，

```
static inline void
wl_dl_window_enable_button_by_httper_status(WlDownloadWindow * window, WlHttperStatus status)
```

```
{

switch (status) {
case WL_HTTPER_STATUS_NOT_START:
case WL_HTTPER_STATUS_ABORT:
case WL_HTTPER_STATUS_PAUSE:
wl_dl_window_set_start_enabled(window, TRUE);
wl_dl_window_set_pause_enabled(window, FALSE);
wl_dl_window_set_remove_enabled(window, TRUE);
break;
case WL_HTTPER_STATUS_START:
wl_dl_window_set_start_enabled(window, FALSE);
wl_dl_window_set_pause_enabled(window, TRUE);
wl_dl_window_set_remove_enabled(window, TRUE);
break;
case WL_HTTPER_STATUS_COMPLETE:
wl_dl_window_set_start_enabled(window, FALSE);
wl_dl_window_set_pause_enabled(window, FALSE);
wl_dl_window_set_remove_enabled(window, TRUE);
break;
default:
g_warning("unknown status!");
}
}
```

第四章 简单 HTTP 服务器程序的设计

本章节描述简单的 HTTP 服务器的设计，包括功能设计和程序的逻辑设计。该 HTTP 服务器程序只实现了几个简单的 HTTP 功能，包括 HTTP200、HTTP206 和 HTTP404。同时使用了目录限定技术，防止客户端请求的资源超过预设的内容。

第一节 程序流程

HTTP 服务器是典型的“被动“服务器，它不会主动向客户端发送什么，完全依靠客户端的请求，然后执行响应。下图描述了 HTTP 服务器的对客户端请求的处理流程。

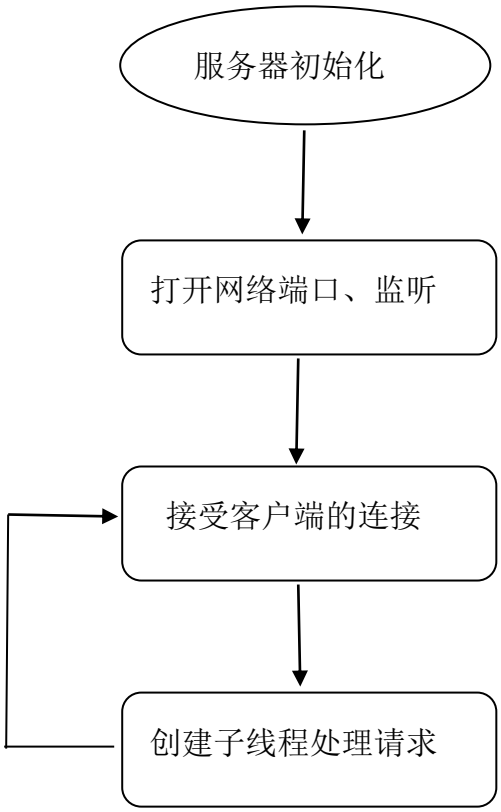

图 4-1 HTTP 服务器程序的主线程流程

可以看到，在主线程中只是接受来自客户端的连接，而不处理任何 HTTP 相关的细节。这是为了保证服务器能够最快的速度响应客户端的连接请求，在实际生产环境中，为了提高速度还会引入线程池的技术。下图表示了子线程处理 HTTP 请求并做出响应的流程。

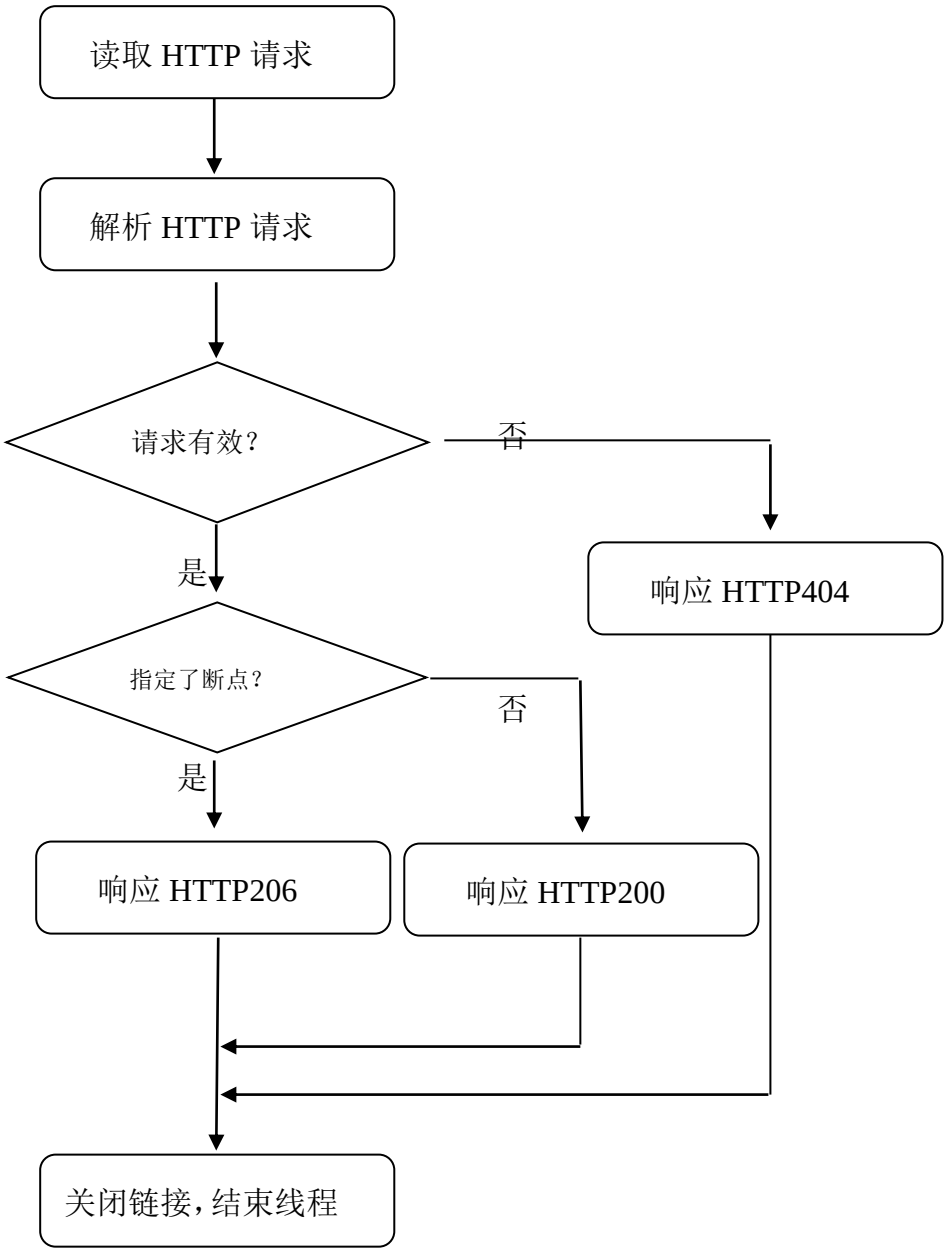

图 4-2 HTTP 请求的响应流程

处理 HTTP 请求的过程就如上图所示，整个流程很简单。

第二节 数据结构

程序中最主要的数据结构就是表示 HTTP 请求的 HttpRequest 结构。该结构的定义如下：

```
typedef struct {
    HttpStartLine *startLine;
    Dlist *headers;
} HttpRequest;
```

HttpStartLine 是表示 HTTP 请求首行的结构，Dlist 是一个双向链表，headers 是结构为 HttpHeader 的双向链表。

HttpStartLine 结构定义如下：

```
typedef struct {
    HttpMethod method;
    char url[HTTP_URL_MAX];
    HttpVersion version;
} HttpStartLine;
```

HttpMethod 是表示 HTTP 请求方法的枚举类型，可取的值有 HTTP_GET,HTTP_POST 和 HTTP_HEAD。

url 是表示请求的 URL，长度限制为 HTTP_URL_MAX，其值为 1024。

HttpVersion 是表示 HTTP 版本的枚举类型，可取的值有 HTTP_0_9，HTTP_1_0 和 HTTP_1_1。

HttpHeader 表示 HTTP 的一条首部字段，是典型的键值模式，其结构定义如下：

```
typedef struct {
    char *name;
    char *value;
} HttpHeader;
```

name 表示首部字段名，value 就是其字段的值。

第五章 简单 HTTP 服务器程序的实现

本节我们描述如何实现一个简单的 HTTP 服务器的。在描述 HTTP 服务器的具体实现之前，本章先对简单 HTTP 服务器的功能进行分析。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，

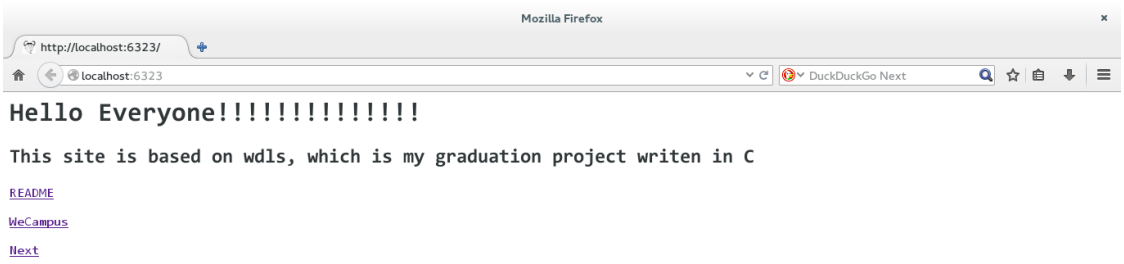

图 5-1 HTTP 服务器运行截图

图 5-2 HTTP 服务器运行截图

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。

第一节 主线程监听连接

在主线程中，先解析命令行参数指定的端口号，然后打开该端口号监听网络连接。要完成该操作一共有四步；

一、 调用 socket 函数创建套接字。

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。

二、 调用 bind 函数绑定本地地址和端口号

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));
```

三、 调用 listen 函数设置套接字为被动模式

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。

四、 调用 accept 接受来自客户端的网络连接。

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。

接收到来自客户端的连接后 accept 函数返回一个有效的套接字描述符。这时主线程调用 http_thread 函数创建子线程来处理这个连接。http_thread 其实是对 POSIX 线程函数的封装。关于 POSIX 线程的内容，本文不做过多描述，感兴趣的读者可以参考 POSIX 线程[4]。

主线程一直处在一个 while 循环中接受来自客户端的连接，除非出错，否则这个循环不会退出，或者用户在给进程发送了 SIGINT 信号，这一般是在控制终端中按下了 CTRL+C。

第二节 解析 HTTP 请求

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。下面是程序中处理 HTTP 请求的实现代码。

```
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。

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[5]。

第三节 响应 HTTP 请求

根据 HTTP 请求的类型，分为三种响应状态。如果 HTTP 请求无效，返回 HTTP 的 404 响应，如果是有效的请求则判断是否为断点续传，如果是返回 HTTP 的 206 响应，否则是 200 响应。

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，子线程就结束了，这样一个 HTTP 请求就处理完毕。

第六章 系统运行与测试

本章描述文件传输管理器和 HTTP 服务器的运行与测试。主要从用户的角度看待系统的功能。我们先对文件传输管理器进行测试，然后在本机运行 HTTP 服务器，两者配合测试。

第一节 文件传输管理器的运行与测试

文件传输管理器的基本功能就是下载文件，这里我们使用ftp.gnu.org站点的文件资源作为测试用例。首先运行程序，在菜单栏点击新建HTTP/FTP 下载的按钮，弹出下载对话框。如下

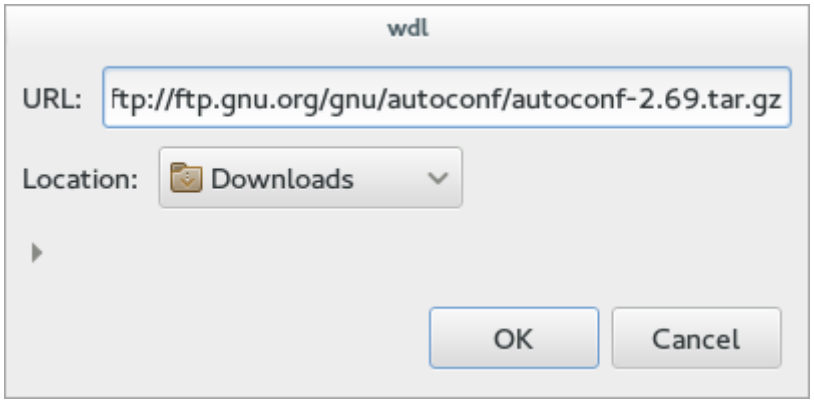

图 6-1 下载对话框

可以设置下载 URL 和文件的保存路径，这里使用默认的文件保存路径。当用户按下确认按钮后，会先进行一次网络请求，确定指定的 URL 是有效的。如果有 URL 无效则提示用户，否则开始下载。

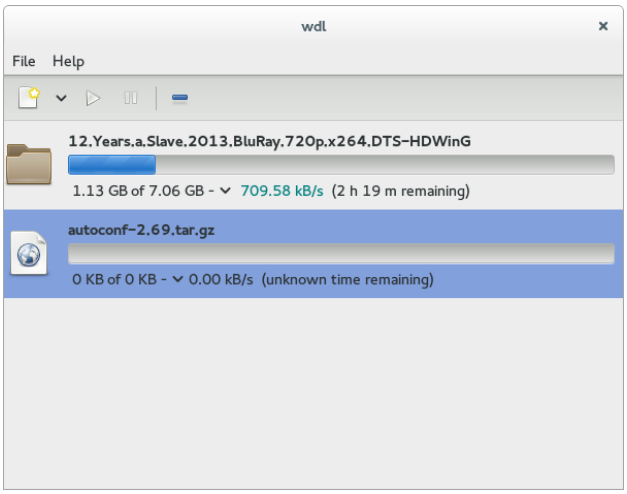

图 6-2 下载管理界面

新建的下载任务默认是不自动开始的，用户点击开始按钮后任务才会开始。

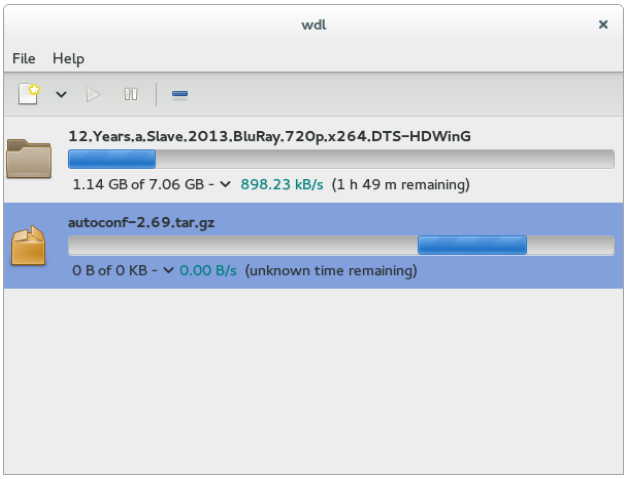

图 6-3 开始任务

如果进度条在滚动表示下载正在初始化，一般是协议的连接过程，尤其是 FTP 协议的连接过程比较长。当下载真正开始后，显示下载速度、下载文件大小、文件总大小等信息。如下，

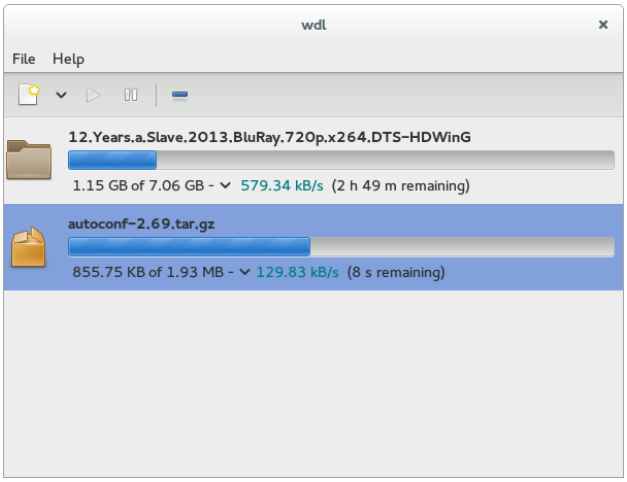

图 6-4 任务下载中

程序可以暂停，也可以直接关闭，重新开启后可以断点续传。重新开启后，任务是以暂停状态，用户要重新点击开始任务。

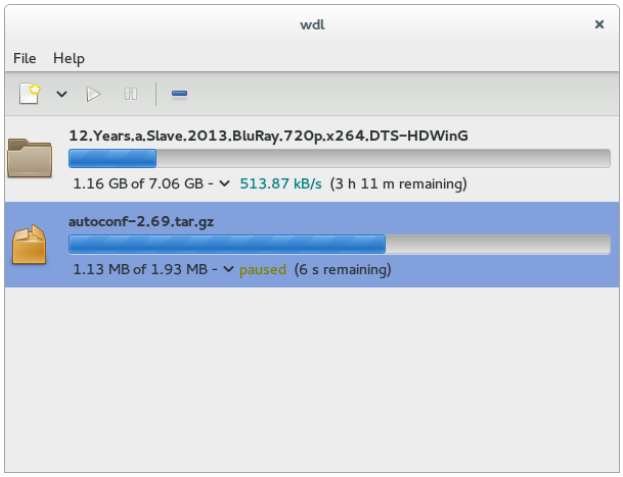

图 6-5 重启程序

任务下载完成后进度条满，而且显示状态为“complete”。最后验证文件的完整性。因为这次下载的文件是一个.tar.gz 的压缩文件，可以简单地用解压的方式判断文件是否正确。

解压没有任何错误，说明文件传输管理器经过重启，断点续传后下载的文件依然是正确的。

第二节 配合测试

上一节使用了互联网上的资源来测试文件传输管理器的功能，本节我们使用本机运行简单 HTTP 服务器来配合测试文件传输管理器的功能。

现在本机运行简单 HTTP 服务器，然后在文件传输管理上输入 http://localhost:6323/Fedora-20-x86_64-DVD.iso。这会让文件传输管理器从本机的 HTTP 服务器上请求资源。结果如下，

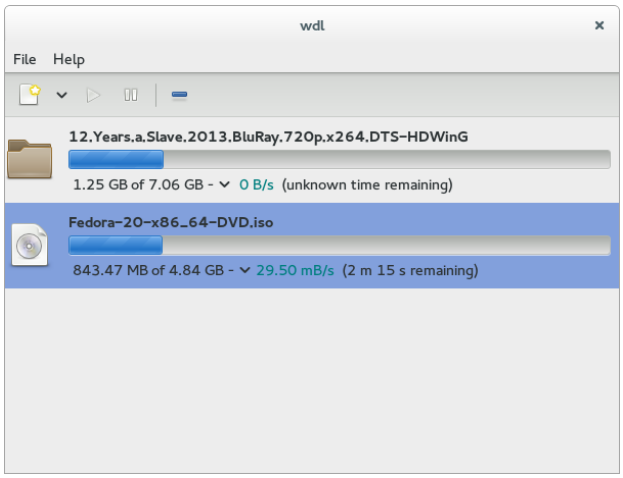

图 6-6 配合测试

因为本机测试的数据是不经过互联网的，因此下载速度非常快，达到29MB/S。同样测试可以正常暂停，重启断点续传，最后完成。

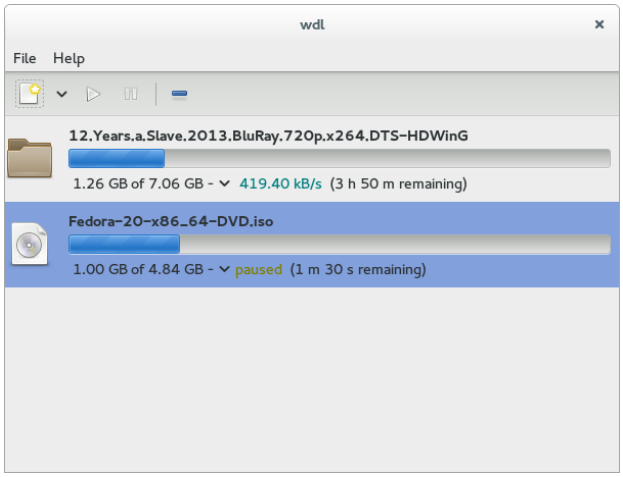

图 6-7 本机测试暂停

可见，无论是文件传输管理器还是简单的 HTTP 服务器，其设计和实现都是可靠的。

结语

互联网是现代计算机行业不可缺少的一部分，通过本次毕业设计加深了我对网络中文件传输的理解与认识。本文描述了互联网上文件传输协议的设计和实现，通过实现一个文件传输管理器和 HTTP 服务器充分展示了网络中文件传输的技术细节。

大部分网络服务都是以客户端、服务器的形式提供的，本文描述的文件传输管理器就充当了客户端了角色，而 HTTP 服务器则是服务器。它们按照预定的协议规范进行数据交换以保证数据传输的有效性。网络传输有两个重要的指标，分别是传输效率和数据完整性文件传输管理器和 HTTP 服务器的实现中都注重文件的完整性，对传输效率考虑较少。

对于文件传输管理器可以从以下方面改善网络传输效率；多任务时智能分配网络带宽，判断资源是否有效和直接下载可以合并，部分代码可以优化提高效率。

对于 HTTP 服务器可以从以下方面改善网络传输效率；使用线程池来提供链接的响应速度，一些常用文件比如网站主页预读到内存，也就是缓存机制。

致　　谢

历时两个多月的毕业设计最后还是圆满完成了，期间遇到过不少问题，不过还是一一克服。

这里我要感谢雷咏梅老师的指导与帮助，规范了我在毕业设计上的操作。我还要感谢 libcurl 和 transmission 的开发者们，是他们无私地开放了源代码，让我有机会深入学习。在毕业设计课题和论文地完成过程中，我的同学杨同也给我提供了很大帮助，在此感谢他。

由于我的学术水平有限，所写论文难免有不足之处，恳请各位老师和学友批评和指正！

参考文献

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

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

[3] J. Fonseca, B. Rezaetal etal. http://jonas.nitro.dk/bittorrent/bittorrent-rfc.html. May 2014.

[4] POSIX 线程 http://zh.wikipedia.org/wiki/POSIX_Threads. May 2014.

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

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

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

[8] GTK+ official website. http://www.gtk.org. May 2014.

[9] C-URL official website. http://curl.haxx.se/. May 2014.

[10] Transmission officail website. http://www.transmissionbt.com/. May 2014.

附录 1：

附录 2：