

摘要

我把毕业设计所做的程序命名为 wdl。本文将详细解释 wdl 的工作原理，包括工作流程、多线程控制、任务管理等。

wdl 由我本人单独设计和实现，对 wdl 本身或者本文档有任何疑问或建议，你都可以联系我 [Wiky L](#)。

wdl 的源代码托管在 [Launchpad](#)，使用 GPLv3 授权。

1. 介绍

wdl 是一个下载管理器，支持 FTP、HTTP(S) 和 BitTorrent 协议，多进程多任务下载。完全采用 C 语言实现。wdl 的实现是为了体现 socket 文件传输，wdl 主要依赖以下第三方的库：[GTK+](#)、[libcurl](#) 和 [libtransmission](#)。虽然 wdl 本身没有任何 socket 的调用，但所依赖的库实际上采用的是 socket 通信。在后面本文也将详细解释其实现原理。

GTK+ 是一个 C 语言实现的用于创建用户界面的图形库。最初是为了开发 GNU 图像处理程序（[GIMP](#)）而设计的。现在它已是 linux 系统下最流行的图形库了，[GNOME](#) 就是基于 GTK+ 开发的。

libcurl 可以被解释为 cURL，即 C 语言实现的 URL 传输库。它支持 FTP、FTPS、Gopher、HTTP(S)、SCP、SFTP、TFTP、Telnet 等文件传输协议。在 wdl 中只取其中的 FTP 和 HTTP(S)。libcurl 是线程安全的，兼容 IPv6。

libtransmission 是 linux 系统下著名的 BT 客户端程序 transmission 所采用的 BT 协议库。libtransmission 没有被单独发布，因而也缺少文档支持，我从 transmission 的源代码中将其提取出来。

1.1 面向的读者

本文主要为了毕业设计而写，不过任何对 wdl 工作原理感兴趣的人，或者想借此学习一些相关知识的人也将受益。

1.2 约定

BT：即 BitTorrent 协议。

大小写：为了书写方便，很多专有名词都是全大写或者全小写，比如 GTK+，GLIB 等；但读者应该清楚，除非有特殊说明，Gtk+ 或者 GLib 具有相同的意思。

源代码：当我说源代码路径时，都是相对于库的源代码根目录。比如，在 GLIB 章节说到源代码 glib/gmain.c:3858 表示 GLIB 源代码目录下，glib 目录中 gmain.c 文件中第 3858 行。注意，可能因为版本不同造成不一致。在本文中，相应的源代码版本分别为 [glib2.0-2.38.1](#)、[gtk+3.0-3.8.6](#)、[curl-7.32.0](#)、[transmission-2.82](#)。

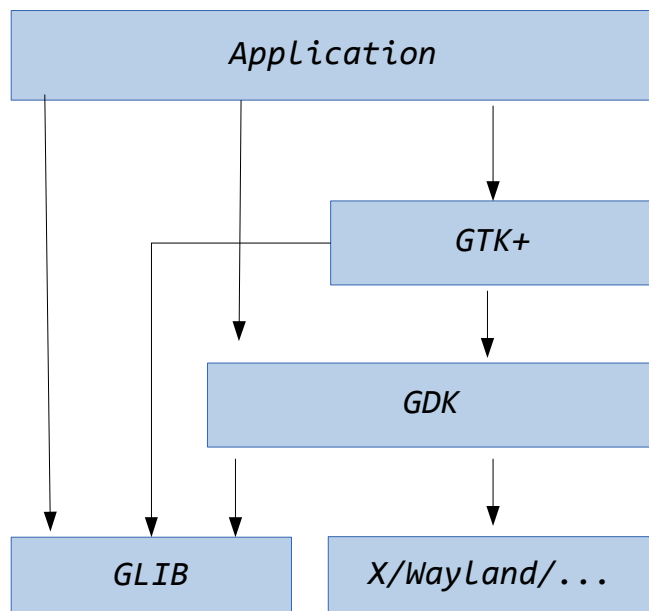
链接：本文中包含很多外部链接，这些链接我无法保证一直有效。

2. 第三方库

wdl 中主要采用了 GTK+、libcurl 和 libtransmission 三个第三方库。本章将详细介绍这三个库在 wdl 中起到的作用。

2.1 GTK+

首先看一下一个典型的 GTK+应用程序的结构。



如图所示，GTK 主要基于 [GLIB](#) 和 [GDK](#) 两个框架。GDK 是实现图形渲染的引擎，负责具体的界面显示，虽然直接影响了 wdl 的界面样式，但与 wdl 的功能实现关系很少；本文将不对 GDK 做具体讨论，有兴趣的读者可以参考 [GDK 的官方文档](#)。

不仅 GTK+ 本身依赖于 GLIB，wdl 本身也依赖于 GLIB。本节将详细解释 GLIB 如何在 GTK+ 中起作用，关于 wdl 如何使用 GLIB 的内容在 [wdl 的实现](#) 中具体讨论。

2.1.1 GTK+命名规范

在使用 GTK+ 时，了解其命名规范是很有必要的。GTK+ 的命名规范其实是 GNOME 定义的。详细可以参考官方文档。本节做简单介绍。

普通类型名：全小写，以 'g' 开头，比如 gint, gchar。

类名：驼峰写法，首字母大写，比如 GtkWidget,GdkEvent。

函数名：小写夹下划线写法，以如 gtk_main,gtk_window_new

常数：大写夹下划线写法，比如 GTK_WINDOW_TOPLEVEL,GTK_ORIENTATION_HORIZONTAL。

2.1.2 GLIB

GLIB 不是 [GLIBC](#)，两者有时会被混淆。GLIBC 指的是 GNU C library，有时也被称为 glibc。

GLIB 最初是 GTK+的一部分，自 GTK+2.0 以后，开发者认为可以将 GTK+中与 GUI 无关的部分独立出来，于是便有了 GLIB。GLIB 主要有 5 部分组成，分别是 GObject、Glib、GModule、GThread 和 GIO。

GObject 是指 GLib Object System，提供了 C 语言以及跨语言的面向对象框架。GTK+采用 GObject 面向对象的方式组织，一个典型的 GTK+类继承树如下。

```
GObject
  GInitiallyUnowned
    GtkWidget
      GtkContainer
        GtkBin
          GtkWindow
            GtkDialog
              GtkAboutDialog
              GtkAppChooserDialog
              GtkColorChooserDialog
              GtkColorSelectionDialog
              GtkFileChooserDialog
              GtkFontChooserDialog
              GtkFontSelectionDialog
              GtkMessageDialog
              GtkPageSetupUnixDialog
              GtkPrintUnixDialog
              GtkRecentChooserDialog
            GtkApplicationWindow
            GtkAssistant
            GtkOffscreenWindow
            GtkPlug
          GtkAlignment
          GtkComboBox
            GtkAppChooserButton
            GtkComboBoxText
          GtkFrame
```

对于不了解 GObject 的读者，可以简单地将其理解为 C++中类提供的特性，虽然两者还是有很大差别。毕竟面向对象仅仅是一种软件组织方式。wdl 本身也采用 GObject 的面向对象框架。

Glib（这里需要注意大小写）包含了一些通用的处理函数，字符串处理，内存管理等。其中比较重要的是主循环的概念，后面将详细讨论。

GModule 提供动态加载模块的功能，wdl 未引入动态模块，不予讨论，有兴趣的读者可以参考[官方文档](#)。

GThread 顾名思义，提供了多线程的支持。

GIO 提供了较高层的文件系统 API。

2.1.3 GObject 信号

每个 GObject 对象可以注册多个信号，每个信号可以注册多个回调函数。在内部实现中，每个信号都有一个信号 ID 表示，是一个 guint 类型的值。每个信号的回调函数用 HandlerList 结构表示，源代码 gobject/gsignal.c: 251

```
struct _HandlerList
{
    guint    signal_id;

    Handler  *handlers;

    Handler  *tail_before; /* normal signal handlers are appended here */

    Handler  *tail_after;  /* CONNECT_AFTER handlers are appended here */
};
```

Handler 结构表示单个回调函数，本身实现为一个双向链表，源代码 gobject/gsignal.c: 259。

```
struct _Handler
{
    gulong    sequential_number;

    Handler    *next;

    Handler    *prev;

    GQuark    detail;

    guint      ref_count;

    guint      block_count : 16;

#define HANDLER_MAX_BLOCK_COUNT (1 << 16)

    guint      after : 1;

    guint      has_invalid_closure_notify : 1;

    GClosure    *closure;
};
```

GClosure 结构表示程序中的回调函数，上述结构表示了单个信号与回调函数（信号处理函数）之间的关系。简单地说，就是单个信号对应多个由链表组成的回调函数。同时回调函数可分为三类，普通的：

HandlerList 结构中的 handlers 字段，优先调用的：HandlerList 结构中 tail_before 字段，以及滞后调用的：HandlerList 结构中的 tail_after 字段。

信号与具体对象是的联系是通过一个全局静态的 hash 表指定的。源代码 gobject/gsignal.c: 303

```
static GHashTable    *g_handler_list_bsa_ht = NULL;
```

该 hash 表是 gpointer（其实是 void *，C 语言里的通用指针）和 GBSearchArray 的对应。

GBSearchArray 是二叉查找树结构，与具体对象相关的所有 HandlerList 结构。

因此，指定具体对象，先通过 `g_hash_table_lookup()` 查找到 `GBsearchArray` 结构，再调用 `g_bsearch_array_lookup()` 就可以找到该对象某个特定信号的处理函数。

2.1.4 GMainLoop

[`GMainLoop`](#) (the main event loop) 管理 GLIB 和 GTK+ 应用程序中所有事件的源 (sources)。事件源可以理解为直接导致事件发生的对象。事件可以任意数量，任意源；比如文件描述符 (普通文件，管道或者套接字) 或者定时器。一般用 [`g_source_attach\(\)`](#) 将一个新事件添加到 `GMainLoop` 中。

为了让多个独立的源可以在不同线程中被处理，每个源与一个 [`GMainContext`](#) 关联。`GMainContext` 表示主循环中事件源的集合；一个 `GMainLoop` 有且仅有一个 `GMainContext`。一个 `GMainContext` 只能在一个线程中执行，但事件源可以在其他线程中添加或者删除。比如，在 A 线程中可以为 B 线程中的 `GMainContext` 添加或删除事件源。

每个事件源都有一个相关优先级。默认优先级为 `G_PRIORITY_DEFAULT`，其值为 0。小于 0 的值表示更高的优先级，反之，大于 0 表示低优先级。拥有高优先级源的事件总是比低优先级源相关的事件优先执行。

也可以添加空闲 (idle) 函数以及相关的优先级。没有更高优先级的事件要执行时就会被执行。

`GMainLoop` 表示一个主事件循环。使用 [`g_main_loop_new\(\)`](#) 创建。添加了初始化的事件源后，调用 [`g_main_loop_run\(\)`](#)。这会不断检查事件源中的事件，然后执行。最后，其中一个事件中调用了 [`g_main_loop_quit\(\)`](#) 就会退出主循环，也就是 `g_main_loop_run()` 将返回。

可以通过递归的方式创建主循环。这也是 GTK 应用程序用来显示模态对话框所采用的方式。注意，事件源有个相关的 `GMainContext`，而该 `GMainContext` 中所有相关的主循环都会检查和事件源的事件。

GTK+ 中有一些函数是对 `GMainLoop` 的封装，比如 `gtk_main()`, `gtk_main_quit()` 和 `gtk_events_pending()`。

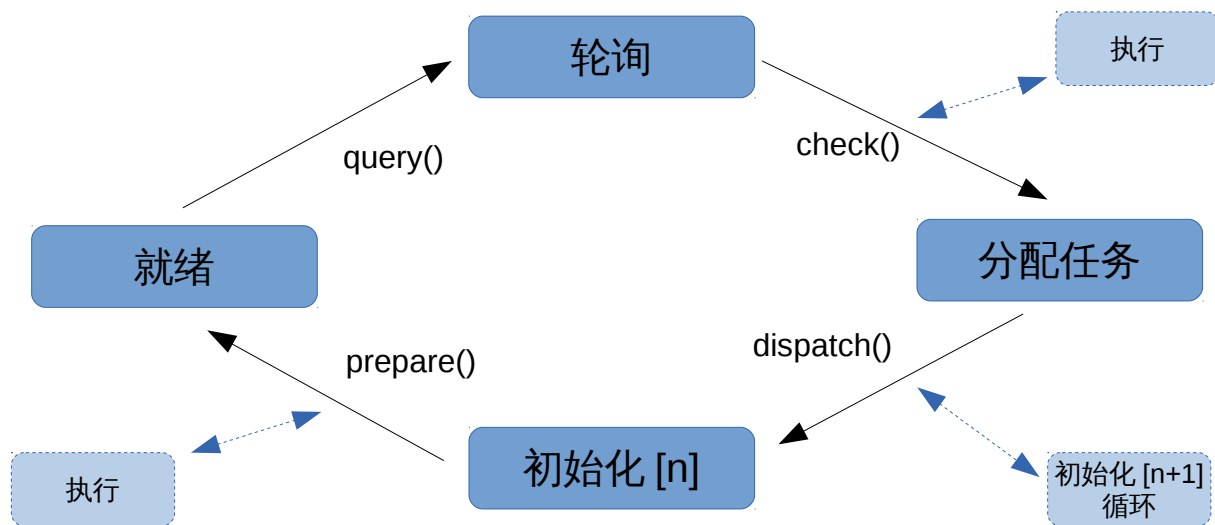
下面代码是 `GMainLoop` 的 C 结构体，一个 `GMainLoop` 只有一个 `GMainContext`，一个 `GMainContext` 则可以关联多个事件源。可以看到，`GMainLoop` 只是 `GMainContext` 的简单封装，因此，很多时候可以将 `GMainLoop`，也就是主循环直接看作 `GMainContext`。

```
struct _GMainLoop {
    GMainContext *context;

    gboolean is_running;

    gint ref_count;
};
```

下图表示一个 `GMainContext` 的循环过程。



初始化完成后进入准备就绪阶段，然后轮询事件源，检测到事件发生就为其分配相应的任务然后执行。执行完成后，进入下一次循环。

2.1.5 创建新的事件源类型

GMainLoop 有个不寻常的功能，就是除了可以使用内置的事件源类型外，还可以自定义事件源类型。一个新事件类型可以用来处理 GDK 事件（鼠标点击等）。自定义事件类型一般源于 GSource 结构。自定义事件源类型的结构体内部以 GSource 结构作为第一个元素，其他元素则作为该事件源特有的结构（这种方式和 GObject 的类继承十分相似，不过这里没有采用 GObject）。新建一个事件源对象，调用 [g_source_new\(\)](#)，需要指定对事件源具体操作的函数和事件源结构体的大小。

2.1.6 主循环过程

所谓主循环过程，就是指当我们调用 [g_main_loop_run\(\)](#) 时，究竟执行了什么。源代码在 `glib/gmain.c:3858`。 `g_main_loop_run()` 的前一部分主要做一些检查，多线程同步等。然后开始执行下面代码：

```

g_atomic_int_inc (&loop->ref_count);      /* 增加引用计数 */
loop->is_running = TRUE;
while (loop->is_running)
    g_main_context_iterate (loop->context, TRUE, TRUE, self);

```

可以看到，程序进入了由 while 控制的循环。单步执行

[g_main_context_iterate\(\)](#)。 `g_main_context_iterate()` 在 `glib/gmain.c:3649` 实现，执行循环中的单步任务。 `g_main_context_iterate()` 首先检查是否有事件源已经就绪，如果没有事件源就绪，在这里将阻塞直到有一个事件源就绪。然后开始执行最高优先级的事件。

因此 `g_main_loop_run()` 会在循环中不断执行，直到在一个事件中调用了 `g_mian_loop_quit()` 退出。

2.1.7 GMainLoop 在 GTK+ 中

下面是一个最小的 GTK+ 应用程序，编译运行后将显示一个 360x250 大小的窗口。

```
#include <gtk/gtk.h>

int main(int argc, char *argv[])
{
    gtk_init(&argc,&argv);

    GtkWidget *window=gtk_window_new(GTK_WINDOW_TOPLEVEL);
    gtk_window_set_default_size(GTK_WINDOW(window),360,250);
    gtk_window_set_position(GTK_WINDOW(window),GTK_WIN_POS_CENTER);
    g_signal_connect(G_OBJECT(window),"destroy",
                     G_CALLBACK(gtk_main_quit),NULL);

    gtk_widget_show(window);

    gtk_main();

    return 0;
}
```

本节将根据上述代码来解释 GTK+ 的工作原理。

在调用任何 GTK+ 函数之前，必须先调用 [gtk_init\(\)](#)，该函数完成 GTK+ 所需要的所有初始化工作同时解析命令行参数。

[gtk_window_new\(\)](#) 创建一个窗口，[gtk_window_set_default_size\(\)](#) 和 [gtk_window_set_position\(\)](#) 分别设置该窗口的默认大小和所在位置。

[g_signal_connect\(\)](#) 为窗口注册回调函数，当 'destroy' 信号发出时（用户点击窗口关闭按钮）调用 [gtk_mian_quit\(\)](#)，将退出程序。

创建的窗口默认是不可见的，因此调用 [gtk_widget_show\(\)](#) 显示窗口。

最后调用 [gtk_main\(\)](#)，该函数启动 GTK+ 内部的主循环，源代码 `gtk/gtkmain.c: 1144`。

```
Loop = g_main_loop_new (NULL, TRUE);
main_loops = g_slist_prepend (main_loops, Loop);

if (g_main_loop_is_running (main_loops->data))
{
    gdk_threads_leave ();    /* 为了兼容低版本 GTK+, 现在可以不用考虑 */
}
```

```
g_main_loop_run (loop);

gdk_threads_enter ();    /* 为了兼容低版本GTK+, 现在可以不用考虑 */

gdk_flush ();

}
```

调用 `gtk_main()` 后程序进入 `g_main_loop_run()` 中的循环，直到调用 `gtk_main_quit()` 退出。当 GTK+ 程序运行时，主线程一直处于 `gtk_main()` 控制的循环当中，不断接受事件。事件通常是由用户界面发出的，比如鼠标点击，窗口拖动等，还有定时器（`timeout`）等；同时还有由窗口管理器或其他应用程序发出的事件。GTK+ 接收事件后，做出响应，通常是已注册的回调函数，在上述例子中，调用 `gtk_main_quit()` 就是 GTK+ 在接收 'destroy' 事件后做出的响应。

正如前文 GLIB 章节所说的那样，`GMainContext` 只能在单线程中执行，而 GTK+ 使用的是 `GMainContext` 的简单封装 `GMainLoop`，因此 GTK+ 函数也只能在单线程中执行，也就是说一个 GTK+ 应用程序，GUI 相关的代码必须是在同一个线程中。

同时，因为回调函数是一个循环中的一部分，属于 `GMainContext` 循环过程中的任务执行那部分。为了保证其他事件得到及时响应。回调函数应该尽可能快速执行，否则可能导致 GTK+ 出现响应延时。

GTK+ 采用了 `GObject` 框架，包括信号注册与回调函数的功能。

2.2 libcurl

wdl 使用 `libcurl` 来完成 HTTP 和 FTP 的数据传输。本节将讨论 HTTP 和 FTP 协议，以及 `libcurl` 的使用。`libcurl` 本身支持非常多的协议，但是提供了相对一致的接口。

2.2.1 URL

URL 即统一资源定位符 (Uniform Resource Locator)，是对互联网上得到的资源的位置的访问方法的一种简洁表示，是互联网上标准资源的地址。互联网上每个文件有唯一的 URL 表示，包含了 HTTP 客户端该如何获取它的一些信息。URL 的结构如下

```
scheme://domain:port/path?query#fragment
```

`scheme` 表示协议，忽略大小写；一般有 `http`, `https`, `ftp` 等，如果没有指定，默认 `http`。

`domain` 表示点分的 IP 地址或者是 DNS 域名；比如 `[db8:0cec::99:123a]` 或者 `example.org`。

`path` 表示在指定主机上该资源的位置，区分大小写；比如 `/index.html`，如果没有指定，默认为 `/`。

`query` 包含了发送给服务器的一些参数，是键\值的对应组，多个由 `&` 隔开；比如 `first=A&second=B`。

`fragment` 表示页面中的位置，一般是 HTML 文件中某个章节。

其中 `scheme`，`path`，`query`，`fragment` 都可以不指定。如 `www.example.com` 是合法的 URL。

但是如果指定了响应的值，必须是相应合法的值。

不可打印字符比如中文，需要通过十六进制编码的方式传输，比如 `%AD`。

在一些协议中，使用了认证机制，比如 FTP；可以在 FTP 的 URL 中加入用户名和密码。如下

```
ftp://username:password@domain/path?query#fragment
```

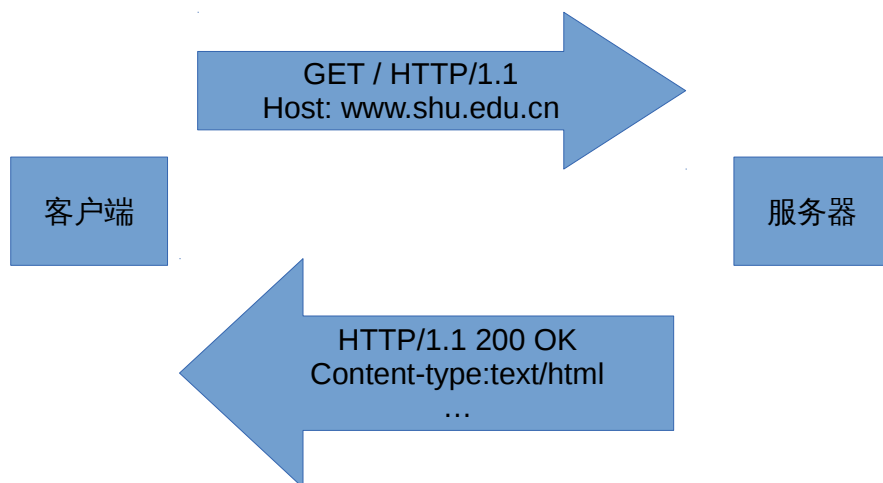
2.2.2 HTTP

HTTP 即超文本传输协议（Hyper Text Transfer Protocol）。本小节讨论 HTTP 的协议内容，将不会讨论 HTTP 引出的一些扩展概念，比如 Web、链接管理、缓存、代理等，关于这方面的详细信息可以参考《HTTP 权威指南》，同时我也假设读者在这方面有了一定的知识背景。

Web 内容都是存储在 Web 服务器上的。Web 服务器所使用的是 HTTP 协议，因此经常会被称为 HTTP 服务器。这些 HTTP 服务器存储了互联网中的数据，如果 HTTP 客户端请求的话，它们就提供数据。

浏览一个页面时（比如 <http://www.shu.edu.cn>），浏览器会向服务器 www.shu.edu.cn 发送一条 HTTP 请求。服务器会去寻找所期望的对象（在此例子中一般使用默认的 `/index.html`），如果成功，就将对象、对象模型、对象长度以及其他一些信息放在 HTTP 响应中发给客户端。

一个 HTTP 事务是由一条（从客户端发往服务器）请求命令和一条（从服务器发给客户端）响应结果组成的。这种通信是通过名为 HTTP 报文（HTTP message）的格式化数据块进行的。如下图所示。



HTTP 支持几种不同的请求命令，这些命令称为 HTTP 方法（HTTP method）。每条 HTTP 请求都包含一个方法。这些方法告诉服务器要执行什么动作（获取页面，运行一个网关程序或者删除一个文件等）。五种常见的方法分别为

- GET 从服务器向客户端发送指定的资源
- PUT 将来自客户端的数据存储到一个指定的服务器的资源中去
- DELETE 从服务器中删除命名资源

- POST 将客户端数据发送到一个服务器网管应用程序
- HEAD 仅发送响应中的 HTTP 首部

每条 HTTP 响应报文返回时都会携带一个状态码。状态码是一个三位数的值，表示客户端请求的结果。

- 100 ~ 199 信息性状态码
- 200 ~ 299 成功状态码，最常见的是 200。
- 300 ~ 399 重定向状态码，表示所请求的资源已经移动了位置，请客户端重新使用新位置请求。
- 400 ~ 499 客户端错误状态码，一般是客户端发送了错误的请求；最常见的是 404，服务器无法找到指定的资源。
- 500 ~ 599 服务器错误状态码，客户端发送了有效的请求，但服务器自身却出现了问题。

一般认为大于等于 400 的状态码表示请求失败，反之表示请求有效。伴随着每个状态码，HTTP 响应中还会有一条解释性的内容。在上述例子中是“OK”，如果是 404 则一般是“Not Found”。

HTTP 报文是由一行一行的简单字符串组成的，都是纯文本。下图显示了一个简单事务使用的 HTTP 报文。

(a) 请求报文		(b) 响应报文	
GET /index.html HTTP/1.1	起始行	GET /index.html HTTP/1.1	
Accept: text/* Accept-Language: en,fr	首部	Accept: text/* Accept-Language: en,fr	
	主体	Hi! I'm a message!	

报文的第一行就是起始行，在请求报文中用来说明请求什么，在响应报文中说明发生了什么情况，用\r\n分割。

起始行后面有零个或者多个首部字段。每个首部字段都包含一个名字和一个值，用：分开。每个首部字段之间用\r\n分割，首部结束用一个空行分割。关于首部各个字段的具体内涵，请读者自行参考相关文献。

HTTP 首部之后跟着消息主体，请求和响应依据具体情况都可以没有消息主体，但响应一般都会有。

HTTP 起始行中还有一个 HTTP 版本号，现在一般都用 HTTP/1.1 或者 HTTP/1.0，差别主要在支持的首部上，这已经超出了本书的讨论范围。在本文中，所有的 HTTP 报文都采用 HTTP/1.1。

下面是我写的一个简单 socket 程序，该程序对命令行参数指定的 URL 执行 HTTP 的 GET 方法。将返回的结果输出到标准输出。为了简化 URL 的解析，使用了第三方的 libsoup。该程序的编译运行环境为 Linux 3.11 x86_64、GCC 4.8.1、libsoup2.4。

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <libsoup/soup.h>
#include <errno.h>

int main(int argc, char *argv[])
{
    if(argc!=2){
        fprintf(stderr, "Usage: hget url\n");
        return 0;
    }

    SoupURI *url=soup_uri_new(argv[1]);
    if(url==NULL){
        fprintf(stderr, "Invalid URL!\n");
        return -1;
    }else if(strcmp("http",soup_uri_get_scheme(url))){
        fprintf(stderr, "Only HTTP is supported!\n");
        return -2;
    }

    /* 解析主机地址 */
    struct addrinfo hints;
    struct addrinfo *res=NULL;
    memset(&hints,0,sizeof(struct addrinfo));
    hints.ai_family=AF_UNSPEC; /* Allow IPv4 or IPv6 */
    hints.ai_socktype=SOCK_STREAM; /* Stream socket */

```

```

hints.ai_flags=0;
hints.ai_protocol=0;          /* Any protocol */
if(getaddrinfo(soup_uri_get_host(url), "http", &hints,&res)){
    perror("Invalid host");
    return -3;
}
int sockfd=socket(res->ai_family,res->ai_socktype,res->ai_protocol);
if(sockfd<0){
    perror("Fail to create socket");
    return -4;
}
/* 发起 socket 链接 */
if(connect(sockfd,res->ai_addr,res->ai_addrlen)){
    perror("Fail to connect");
    return -5;
}
/* 构造 HTTP 请求报文 */
char request[4096];
snprintf(request,4096,"GET %s HTTP/1.1\r\n"
           "Host: %s\r\nAccept: */*\r\nConnection: close\r\n\r\n",
           soup_uri_get_path(url),soup_uri_get_host(url));
/* 发送 HTTP 请求报文 */
write(sockfd,request,strlen(request));
char response[4096];
int n;
/* 读取 HTTP 响应并输出到标准输出 */
while((n=read(sockfd,response,4096))>0){
    write(STDOUT_FILENO,response,n);
}

freeaddrinfo(res);
/* 关闭套接字 */
close(sockfd);
return 0;

```

```
}
```

运行结果如下：

```
$. /a.out http://www.baidu.com

HTTP/1.1 200 OK
Date: Sun, 23 Mar 2014 10:38:15 GMT
Content-Type: text/html
Transfer-Encoding: chunked
Connection: Close
Vary: Accept-Encoding
Set-Cookie: BAIDUID=C6099A3FCCEC016FD7C78B679511F0F8:FG=1; expires=Thu, 31-Dec-37 23:55:55 GMT;
max-age=2147483647; path=/; domain=.baidu.com
Set-Cookie: BDSVRTM=0; path=/
Set-Cookie: H_PS_PSSID=5229_1435_5224_5722_4261_5565_4759_5659; path=/; domain=.baidu.com
P3P: CP=" OTI DSP COR IVA OUR IND COM "
Expires: Sun, 23 Mar 2014 10:38:00 GMT
Cache-Control: private
Server: BWS/1.1
BDPAGETYPE: 1
BDQID: 0xcc97b76d0023d327
BDUSERID: 0

405e

<!DOCTYPE html><!--STATUS OK--><html><head><meta http-equiv="content-type"
content="text/html; charset=utf-8"><link rel="dns-prefetch.....<省略>
```

2.2.3 HTTP 在 libcurl 中

在上一节我们简单讨论了 HTTP 协议，并且实现了一个发送 HTTP 请求的简单程序。本节讨论 libcurl 内部是如何实现 HTTP 传输的，主要关注 libcurl 的数据传输而不是内存管理、模块划分、错误处理等内容。

在讨论 libcurl 的实现之前，先看一下如何使用 libcurl 发送 HTTP 请求。下面这个程序可以看作是上一节 HTTP 小程序的 libcurl 版本。

```
#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;
}

```

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，这里不详细描述。

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。

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。

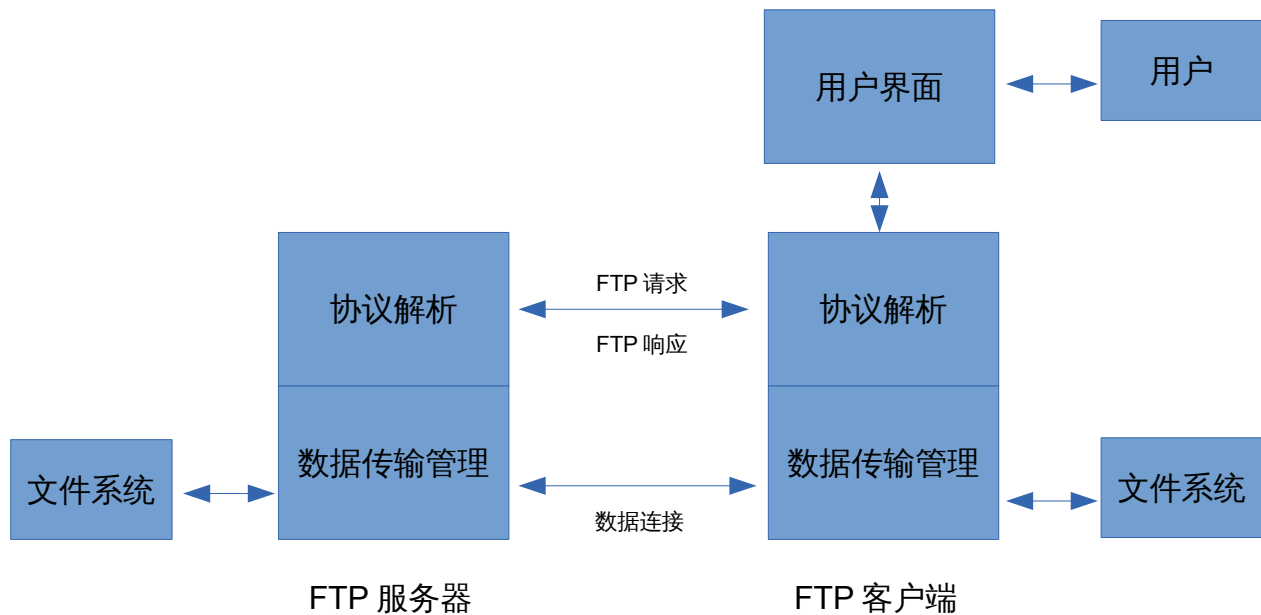
2.2.4 FTP

FTP 即文件传输协议（File Transfer Protocol）。是基于 TCP/IP 的网络上进行两台计算机文件传输的协议。尽管 HTTP 替代了大多数 FTP 的功能，但 FTP 依然是客户机与服务器进行文件传输的有效方式。FTP 允许客户端从服务器上下载文件，或者上传、创建甚至删除服务器上的文件或目录。

本章节主要讨论 FTP 的数据传输方式，相关的 FTP 认证、安全等内容不在讨论范围内。有兴趣的读者可以参考 [RFC959](#)。

FTP 和 HTTP 一样，是应用层协议，基于传输层。FTP 是一个 8 位的客户端-服务器协议，能操作任何类型的文件而不需要进一步处理，就像 MIME 或者 Unicode 一样。但是，FTP 有着极高的延时，这意味着，从开始请求到第一次接收数据之间时间可能比较长，并且不时地需要执行一些冗长的登录过程。

FTP 服务一般运行在 20 和 21 两个端口。端口 20 用于客户端与服务器之间的数据传输，而端口 21 用户客户端与服务器之间的控制信息传输。典型的 FTP 客户端-服务器模型如下



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 控制帧都是简单的 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，服务器接受连接，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。

2.3 libtransmission

libtransmission...

3. wdl 的实现

3.1 模块划分

3.2 界面设计

3.3 程序启动

3.4 下载任务

3.5 多线程控制