WDL 设计与实现

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。

目录

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 gib/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。

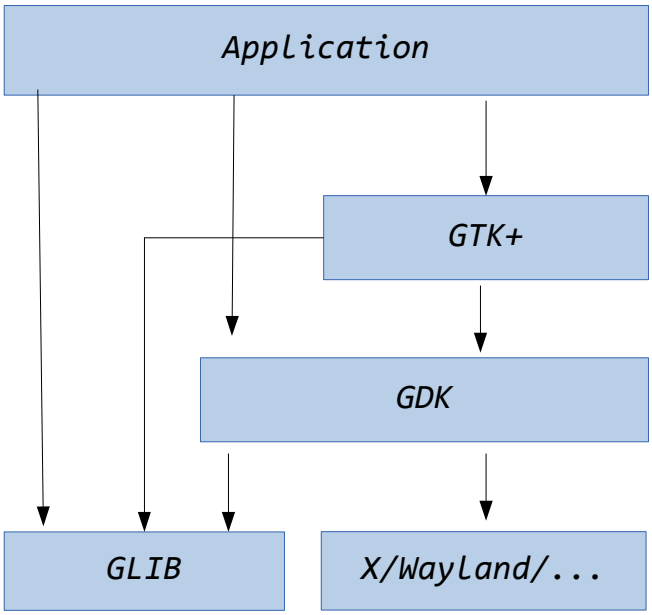
链接：本文中包含很多外部链接，这些链接我无法保证一直有效。

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 本身也依赖与 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 GtkWindow,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 gnulib。

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 面向对象

因为采用了 GObject 的框架，因此在描述具体实现，之前有必要解释一下 GObject 面向对象框架的使用方

式。我们知道，在 C++里，类和对象是不同的概念，同样的，在 GObject 里也是不同的概念。而且，在 GObject 里，类和对象分别用两个不同的结构表示。以 WlDownloader 为例，

```
struct _WlDownloader {
	GtkScrolledWindow parent;
	/* Private */
	GtkWidget *vBox;
	GList *list;
	gpointer *selected;

	/* Callback */
	WlHttperSelectedCallback httperSelected;
	gpointer httperSelectedData;
	WlHttperStatusCallback httperStatus;
	gpointer httperStatusData;
};

struct _WlDownloaderClass {
	GtkScrolledWindowClass parentClass;
};
```

_WlDownloader 和_WlDownloaderClass 分别是 WlDownloader 的实例（对象）结构和类结构。实例结构中保存了一个具体实例独一无二的信息，而类结构则保存了类中独一无二的信息。无论我们创建多少 WlDownloader 的实例对象，其类结构在内存中只会有一份；类结果中的字段类似与 C++类中的静态成员。不过在 wdl 各个类中都没有使用到静态成员，也就是类结构中总是只有一个父类的字段，因此后面将不对类结果做更多描述。

实例结构的第一个字段必须是父类的实例结构，如果你熟悉 C 语言，那么很容易理解这点。比如创建一个 WlDownloader 的实例，

```
WlDownloader *dl=wl_downloader_new();
```

那么便可以直接使用*(GtkScrolledWindow*)dl* 的方式强制转化为父类的对象，结构之间也就有了继承与派生的关系。为了更加直观方便地进行类型转化，一般会定义几个宏来完成。比如

```
#define WL_DOWNLOADER(obj)  \
	G_TYPE_CHECK_INSTANCE_CAST((obj),WL_TYPE_DOWNLOADER,WlDownloader)
```

其中 G_TYPE_CHECK_INSTANCE_CAST 是 GObject 内部定义的，完成通用形式的类型转化。

当你使用 WL_DOWNLOADER(ptr)时就表示把 ptr 转化为 WlDownloader。

但毕竟 C 语言本身不提供面向对象的功能，因此，不能像 C++中那样通过->的方式调用成员函数。一般成员函数的第一个参数就是类的实例对象，比如

```
void wl_downloader_remove_httper(WlDownloader * dl, WlHttper * httper);
```

在 GObject 中还提供了信号和属性的功能。可以给类添加信号和属性。属性其实只是对类的另一种访问方式，大部分类的属性访问都类似于 set 和 get 的成员函数。在 wdl 中并没有使用任何属性，因此不再继续讨论，有兴趣的读者可以参考官方文档。下一节讨论 GObject 的信号机制。

2.1.4 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。如果可以注册多个回调函数，那么回调函数一般按注册时间依次调用，除非有特殊设置，比如 g_signal_connect_after。

wdl 本身并不使用 GObject 的信号机制，但是 GTK+通过 GObject 的信号机制来完成对界面事件（鼠标点击等）的响应。

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。

2.1.5 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_PRIOPRITY_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。

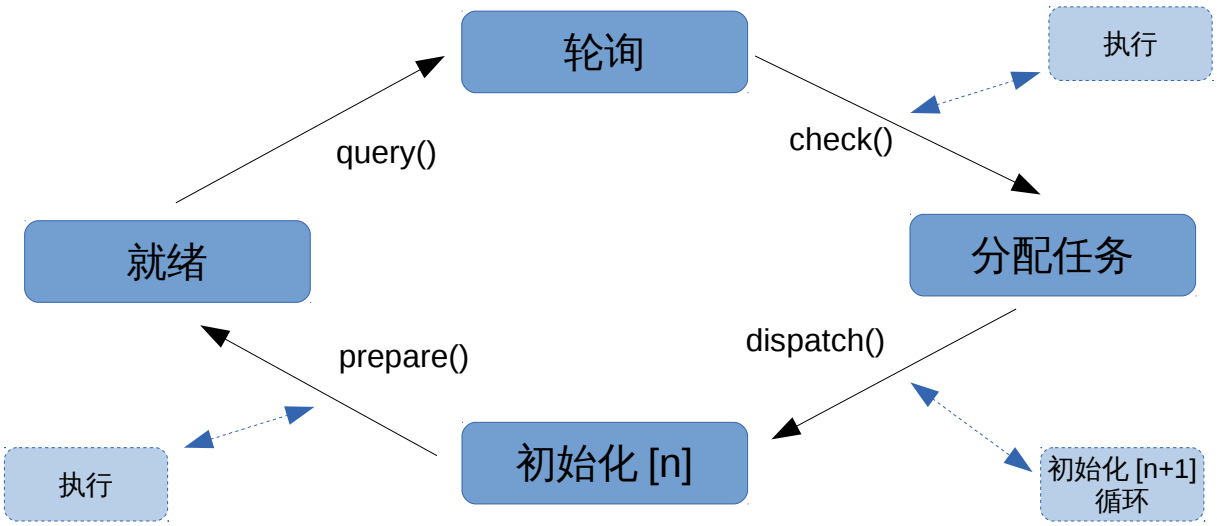
GTK+中有一些函数是对 GMainLoop 的封装，比如 gtk_main(),gtk_main_quit()和 gtk_events_pending()。

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。

2.1.6 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。

2.1.7 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。

2.1.8 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
```

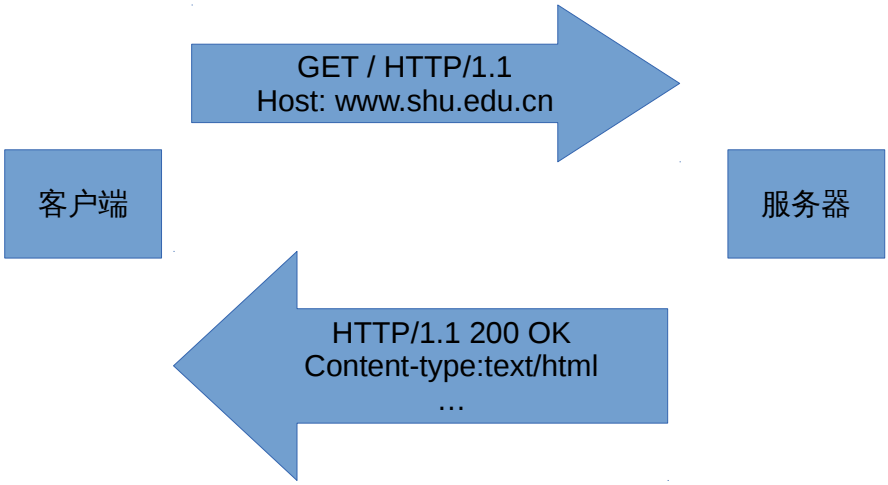
2.2.2 HTTP

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

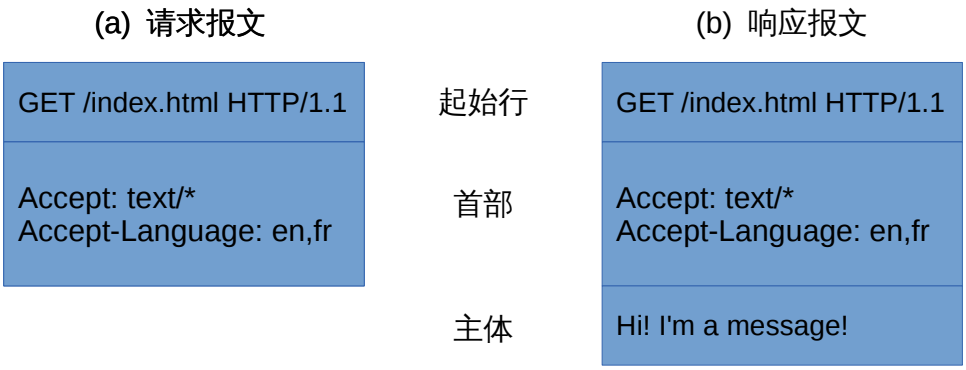
- GET　从服务器向客户端发送指定的资源
- PUT　将来自客户端的数据存储到一个指定的服务器的资源中去
- DELETE　从服务器中删除命名资源
- POST 将客户端数据发送到一个服务器网管应用程序
- HEAD 仅发送响应中的 HTTP 首部

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　服务器错误状态吗，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 的状态吗表示请求失败，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。



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(socket<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。

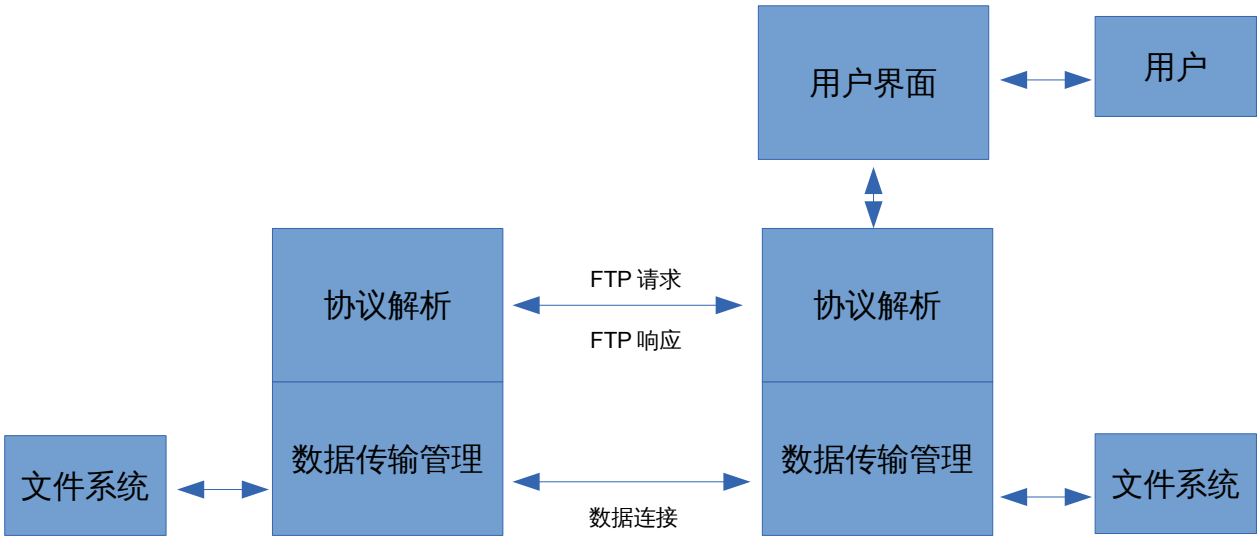
2.2.4 FTP

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);
```

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。

2.2.5 FTP 在 libcurl 中

本节主要关注 libcurl 是如何发送和接受 FTP 命令消息的。在 libcurl 中执行不同协议的操作都是调用 curl_easy_perform()。然后再根据协议调用相关函数。

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()先根据用户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。

2.3 libtransmission

libtransmission...

3. wdl 的实现

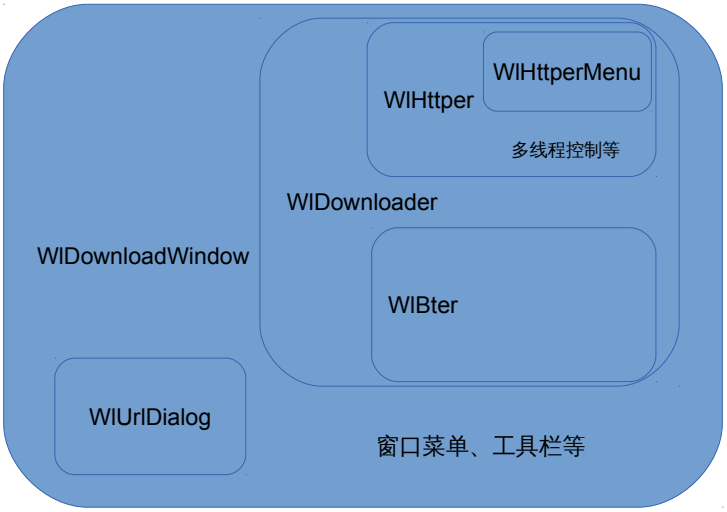
wdl 的整体结构采用 GObject 提供的面向对象机制，读者只要将其理解为 C++中的类机制就行。同 C++的

代码组织方式一样，也是在头文件（.h）里定义类，在源文件（.c）里实现类。比如文件 wlhttper.h 和 wlhttper.c 分别是类 WlHttper 的定义于实现。

wdl 的命名规范沿用了 GTK+的命名规范。Wl 为前缀，功能类似 C++中的命名空间。下面列出了 wdl 中实现的类以及具体功能。

类名	父类	功能
WlHttper	GtkEventBox	管理一个 HTTP 或者 FTP 下载任务，包括任务的界面，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 会完成对线程的管理。在使用 WlHttper 类时不用考虑多线程（这也是面向对象意义所在）。但后面我们会讨论 WlHttper 是如何管理多线程的。
WlBter	GtkEventBox	与 WlHttper 类似，但是它管理的是一个 BT 下载任务，**TODO**
WlHttperMenu	GtkMenu	管理一个 WlHttper 的右键菜单，默认一些简单的功能，也可以额外添加。该类被嵌入 WlHttper 中。
WlHttperProperties	GtkDialog	这是一个对话框，用来显示一个 HTTP 或者 FTP 下载任务的信息。URL 地址和保存路径。
WlUrlDialog	GtkDialog	这是一个对话框，用来供用户输出 URL 地址。由 WlDownloadWindow 调用，开始一个 HTTP 或者 FTP 下载。
WlDownloader	GtkScrolledWindow	下载任务的总管理，管理 WlHttper 和 WlBter，对外提供一致的访问方式。
WlDownloadWindow	GtkWindow	wdl 的主窗口类，负责窗口菜单，以及 WlDownloader。
TODO	**TODO**	**TODO**

下面用层次图的方式让读者更清楚的了解各个类之间的关系和层次。



3.1 模块实现

wdl 采用了面向对象的组织方式，因此可以将每个类的实现看作单个模块。本章节将讨论各个模块的具体实现，包括流程数据结构等。以及各个类（模块）之间是如何协作的。

3.1.1 WlHttper

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);
```

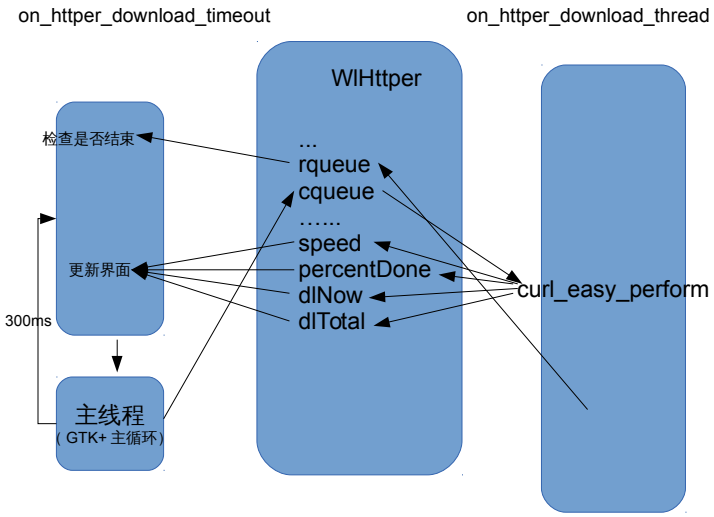
要么成功，要么失败。最后释放对 cqueue 和 rqueue 的引用。退出线程。

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。



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。

3.1.2 WlHttperMenu

WlHttperMenu 继承自 GtkMenu。是 WlHttper 的右键菜单，主要是 GTK+的编程。

3.1.3 WlDownloader

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。

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。

3.1.4 WlHttperProperties

用来显示 WlHttper 信息的对话框，目前仅仅显示下载 URL 和保存文件的路径。主要是 GTK+编程。

3.1.5 WlUrlDialog

提供一个让用户输入下载 URL 的对话框。主要是 GTK+编程。

3.1.6 WlDownloadWindow

wdl 的主窗口，主要是 GTK+的编程。

需要根据 wlDownloader 所选中的任务状态来改变工具栏按钮是否可用，比如如果当前选中任务正在下载，那么开始按钮就设置为灰色。这需要注册 WlDownloader 改变选中任务的回调函数，以及 WlHttper 状态改变回调函数才能完成。下面代码显示了如何处理 WlHttper 状态改变。

```
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!");
	}
}
```