libcurl 教程

译者注：这是一篇介绍如何使用 libcurl 的入门教程。文档不是逐字逐句按原文翻译，而是根据笔者对 libcurl 的理解，参考原文写成。文中用到的一 些例子，可能不是出自原文，而是笔者在学习过程中，写的一些示例程序（笔者使用的 libcurl 版本是：7.19.6）。出现在这里主要是为了更好的说明 libcurl 的某些 api 函数的使用。许多例子都参考 libcurl 提供的 example 代码。原文 example 中的提供的示例程序完全使用 C 语言， 而这里笔者提供的例子使用 C++语言。因为能力有限，对于 libcurl 的某些理解和使用可能有误，欢迎批评指正。

目 标

本文档介绍了在应用程序开发过程中，如何正确使用 libcurl 的基本方式和指导原则。文档使用 C 语言来调用 libcurl 的接口，当然也适用于其他与 C 语言接近的语言。

文档主要针对使用 libcurl 来进行开发的人员。文档所撰的应用程序泛指你写的源代码，这些代码使用了 libcurl 进行数据传输。

更多关于 libcurl 的功能和接口信息，可以在相关的主页上查阅。

编译源码

有很多种不同的方式来编译 C 语言代码。这里使用 UNIX 平台下的编译方式。即使你使用的是其他的操作系统，你仍然可以通过阅读本文档来获取许多有用的信 息。

编译

你的编译器必须知道 libcurl 头文件的位置。所以在编译的时候，你要设置头文件的包含路径。可以使用 curl-config 工具来获取这方面的信息：

```
$ curl-config –cflags
```

链接

编译完源码（这时的源代码不是指 libcurl 的源代码，你是你自己写的程序代码）之后，你还必须把目标文件链接成单个可执行文件。你要链接 libcurl 库，以及 libcurl 所依赖的其他库，例如 OpenSLL 库。当然可能还需要一些其他的操作系统库。最后你还要设置一些编译选项，当然可 以使用 curl-config 工具简化操作：

```
$curl-config –libs
```

是否使用 SSL

定制编译 libcurl。与其他库不同的是，libcurl 可以定制编译，根据实际需要是否支持某些特性，如是否支持 SSL 传输，像 HTTPS 和 FTPS。如果决定需要支持 SSL，必须在编译时正确的设置。可以使用’curl-config’来判断 libcurl 库是否支持 SSL：

```
$ curl-config –feature
```

autoconf 宏

当你编写配置脚本来检测 libcurl 及其相应设置时，你可以使用预定义宏。文档 docs/libcurl/libcurl.m4 告诉你如何使用这些宏。

跨平台的可移植的代码

libcurl 的开发人员花费很大的努力，使 libcurl 尽可能在大多数平台上正常运行。

全局初始化

应用程序在使用 libcurl 之前，必须先初始化 libcurl。libcurl 只需初始化一次。可以使用以下语句进行初始化：

```
curl_global_init();
```

curl_global_init()接收一个参数，告诉 libcurl 如何初始化。参数 *CURL_GLOBAL_ALL* 会使 libcurl 初始化所有的子模块和一些默认的选项，通常这是一个比较好的默认参数值。还有两个可选值：

CURL_GLOBAL_WIN32

只能应用于 Windows 平台。它告诉 libcurl 初始化 winsock 库。如果 winsock 库没有正确地初始化，应用程序就不能使用 socket。 在应用程序中，只要初始化一次即可。

CURL_GLOBAL_SSL

如果 libcurl 在编译时被设定支持 SSL，那么该参数用于初始化相应的 SSL 库。同样，在应用程序中，只要初始化一次即可。

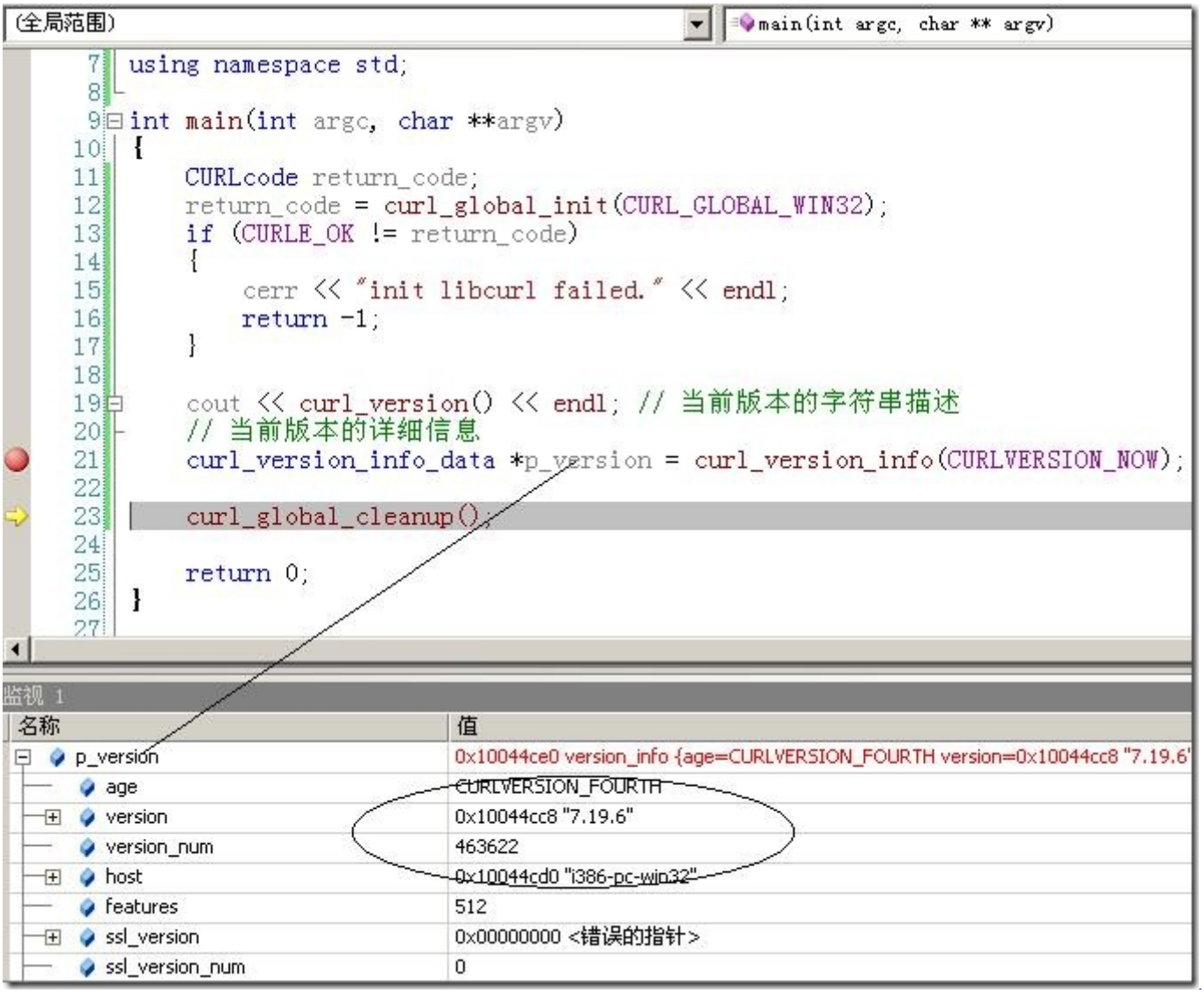
libcurl 有默认的保护机制，如果在调用 curl_easy_perform 时它检测到还没有通过 curl_global_init 进行初始 化，libcurl 会根据当前的运行时环境，自动调用全局初始化函数。但必须清楚的是，让系统自己初始化不是一个好的选择。

当应用程序不再使用 libcurl 的时候，应该调用 curl_global_cleanup 来释放相关的资源。

在程序中，应当避免多次调用 curl_global_init 和 curl_global_cleanup。它们只能被调用一次。

libcurl 提供的功能

在运行时根据 libcurl 支持的特性来进行开发，通常比编译时更好。可以通过调用 curl_version_info 函数返回的结构体来获取运行时的具 体信息，从而确定当前环境下 libcurl 支持的一些特性。下面是笔者在 visual studio2008 中调用相关函数获取 libcurl 版本信息的截图：



使用 **easy interface**

首先介绍 libcurl 中被称为 easy interface 的 api 函数，所有这些函数都是有相同的前缀：curl_easy 。

当前版本的 libcurl 也提供了 multi interface，关于这些接口的详细使用，在下面的章节中会有介绍。在使用 multi interface 之前，你首先应该理解如何使用 easy interface。

要使用 easy interface，首先必须创建一个 easy handle，easy handle 用于执行每次操作。基本上，每个线程都应该有自己的 easy handle 用于数据通信（如果需要的话）。千万不要在多线程之间共享同一个 easy handle。下面的函数用于获取一个 easy handle ：

```
CURL *easy_handle = curl_easy_init();
```

在 easy handle 上可以设置属性和操作(action)。easy handle 就像一个逻辑连接，用于接下来要进行的数据传输。

使用 curl_easy_setopt 函数可以设置 easy handle 的属性和操作，这些属性和操作控制 libcurl 如何与远程主机进行数据通信。一旦在 easy handle 中设置了相应的属性和操作，它们将一直作用该 easy handle。也就是说，重复使用 easy hanle 向远程主机发出请求，先前设置的属性仍然生效。

easy handle 的许多属性使用字符串(以\0 结尾的字节数组)来设置。通过 curl_easy_setopt 函数设置字符串属性时，libcurl 内部会自 动拷贝这些字符串，所以在设置完相关属性之后，字符串可以直接被释放掉（如果需要的话）。

easy handle 最基本、最常用的属性是 URL。你应当通过 CURLOPT_URL 属性提供适当的 URL：

```
curl_easy_setopt(easy_handle, CURLOPT_URL, "http://blog.csdn.net/JGood ");
```

假设你要获取 URL 所表示的远程主机上的资源。你需要写一段程序用来完成数据传输，你可能希望直接保存接收到的数据而不是简单的在输出窗口中打印它们。所 以，你必须首先写一个回调函数用来保存接收到的数据。回调函数的原型如下：

```
size_t write_data(void *buffer, size_t size, size_t nmemb, void *userp);
```

可以使用下面的语句来注册回调函数，回调函数将会在接收到数据的时候被调用：

```
curl_easy_setopt(easy_handle, CURLOPT_WRITEFUNCTION, write_data);
```

可以给回调函数提供一个自定义参数，libcurl 不处理该参数，只是简单的传递：

```
curl_easy_setopt(easy_handle, CURLOPT_WRITEDATA, &internal_struct);
```

如果你没有通过 CURLOPT_WRITEFUNCTION 属性给 easy handle 设置回调函数，libcurl 会提供一个默认的回调函数，它只是简单的将接收到的数据打印到标准输出。你也可以通过 CURLOPT_WRITEDATA 属性给默认回调函数传递一个已经打开的文件指针，用于将数据输出到文件里。

下面是一些平台相关的注意点。在一些平台上，libcurl 不能直接操作由应用程序打开的文件。所以，如果使用默认的回调函数，同时通过 CURLOPT_WRITEDATA 属性给 easy handle 传递一个文件指针，应用程序可能会执行失败。如果你希望自己的程序能跑在任何系统上，你必须避免出现这种情况。

如果以 win32 动态连接库的形式来使用 libcurl，在设置 CURLOPT_WRITEDATA 属性时，你必须**同时** 使用 CURLOPT_WRITEFUNCTION 来注册回调函数。否则程序会执行失败（笔者尝试只传递一个打开的文件指针而不显式设置回调函数，程序并没 有崩溃。可能是我使用的方式不正确。）。

当然，libcurl 还支持许多其他的属性，在接下来的篇幅里，你将会逐步地接触到它们。调用下面的函数，将执行真正的数据通信：

```
success = curl_easy_perform(easy_handle);
```

curl_easy_perfrom 将连接到远程主机，执行必要的命令，并接收数据。当接收到数据时，先前设置的回调函数将被调用。libcurl 可能一 次只接收到 1 字节的数据，也可能接收到好几 K 的数据，libcurl 会尽可能多、及时的将数据传递给回调函数。回调函数返回接收的数据长度。如果回调函数 返回的数据长度与传递给它的长度不一致（即返回长度 != size * nmemb），libcurl 将会终止操作，并返回一个错误代码。

当数据传递结束的时候，curl_easy_perform 将返回一个代码表示操作成功或失败。如果需要获取更多有关通信细节的信息，你可以设置 CURLOPT_ERRORBUFFER 属性，让 libcurl 缓存许多可读的错误信息。

easy handle 在完成一次数据通信之后可以被重用。这里非常建议你重用一个已经存在的 easy handle。如果在完成数据传输之后，你创建另一个 easy handle 来执行其他的数据通信，libcurl 在内部会尝试着重用上一次创建的连接。

对于有些协议，下载文件可能包括许多复杂的子过程：日志记录、设置传输模式、选择当前文件夹，最后下载文件数据。使用 libcurl，你不需要关心这一 切，你只需简单地提供一个 URL，libcurl 会给你做剩余所有的工作。

下面的这个例子演示了如何获取网页源码，将其保存到本地文件，并同时将获取的源码输出到控制台上。

```
/**
 *      @brief libcurl 接收到数据时的回调函数
 *
 *      将接收到的数据保存到本地文件中，同时显示在控制台上。
 *
 *      @param [in] buffer 接收到的数据所在缓冲区
 *      @param [in] size 数据长度
 *      @param [in] nmemb 数据片数量
 *      @param [in/out] 用户自定义指针
 *      @return 获取的数据长度
 */

size_t process_data(void *buffer, size_t size, size_t nmemb, void *user_p)
{
        FILE *fp = (FILE *)user_p;
        size_t return_size = fwrite(buffer, size, nmemb, fp);
        cout << (char *)buffer << endl;
        return return_size;
}



int main(int argc, char **argv)
{
        // 初始化 libcurl
        CURLcode return_code;
        return_code = curl_global_init(CURL_GLOBAL_WIN32);
        if (CURLE_OK != return_code)
        {
                cerr << "init libcurl failed." << endl;
                return -1;
        }


        // 获取 easy handle
        CURL *easy_handle = curl_easy_init();
        if (NULL == easy_handle)
        {
                cerr << "get a easy handle failed." << endl;
                   curl_global_cleanup();
                return -1;
        }
```

```
        FILE *fp = fopen("data.html", "ab+");    //
        // 设置 easy handle 属性
        curl_easy_setopt(easy_handle, CURLOPT_URL, http://blog.csdn.net/JGood);
        curl_easy_setopt(easy_handle, CURLOPT_WRITEFUNCTION, &process_data);
        curl_easy_setopt(easy_handle, CURLOPT_WRITEDATA, fp);


        // 执行数据请求
        curl_easy_perform(easy_handle);

        // 释放资源

        fclose(fp);
        curl_easy_cleanup(easy_handle);
        curl_global_cleanup();


        return 0;
}
```

多线程问题

首先一个基本原则就是：绝对不应该在线程之间共享同一个 libcurl handle，不管是 easy handle 还是 multi handle（???在下文中介绍）。一个线程每次只能使用一个 handle。

libcurl 是线程安全的，但有两点例外：信号(signals)和 SSL/TLS handler。 信号用于超时失效名字解析(timing out name resolves)。libcurl 依赖其他的库来支持 SSL/STL，所以用多线程的方式访问 HTTPS 或 FTPS 的 URL 时，应该满足这些库对多线程 操作的一些要求。详细可以参考：

OpenSSL: http://www.openssl.org/docs/crypto/threads.html#DESCRIPTION

GnuTLS: http://www.gnu.org/software/gnutls/manual/html_node/Multi_002dthreaded-applications.html

NSS：宣称是多线程安全的。

什么时候 libcurl 无法正常工作

传输失败总是有原因的。你可能错误的设置了一些 libcurl 的属性或者没有正确的理解某些属性的含义，或者是远程主机返回一些无法被正确解析的内容。

这里有一个黄金法则来处理这些问题：将 CURLOPT_VERBOSE 属性设置为 1，libcurl 会输出通信过程中的一些细节。如果使用的是 http 协 议，请求头/响应头也会被输出。将

CURLOPT_HEADER 设为 1，这些头信息将出现在消息的内容中。

当然不可否认的是，libcurl 还存在 bug。当你在使用 libcurl 的过程中发现 bug 时，希望能够提交给我们，好让我们能够修复这些 bug。你在 提交 bug 时，请同时提供详细的信息：通过 CURLOPT_VERBOSE 属性跟踪到的协议信息、libcurl 版本、libcurl 的客户代码、操作系 统名称、版本、编译器名称、版本等等。

如果你对相关的协议了解越多，在使用 libcurl 时，就越不容易犯错。

上传数据到远程站点

libcurl 提供协议无关的方式进行数据传输。所以上传一个文件到 FTP 服务器，跟向 HTTP 服务器提交一个 PUT 请求的操作方式是类似的：

1. 创建 easy handle 或者重用先前创建的 easy handle。

2. 设置 CURLOPT_URL 属性。

3. 编写回调函数。在执行上传的时候，libcurl 通过回调函数读取要上传的数据。（如果要从远程服务器下载数据，可以通过回调来保存接收到的数据。）回调 函数的原型如下：

```
size_t function(char *bufptr, size_t size, size_t nitems, void *userp);
```

bufptr 指针表示缓冲区，用于保存要上传的数据，size * nitems 是缓冲区数据的长度，userp 是一个用户自定义指针，libcurl 不对该指针作任何操作，它只是简单的传递该指针。可以使用该指针在应用 程序与 libcurl 之间传递信息。

4. 注册回调函数，设置自定义指针。语法如下：

```
// 注册回调函数
curl_easy_setopt(easy_handle, CURLOPT_READFUNCTION, read_function);
// 设置自定义指针
curl_easy_setopt(easy_handle, CURLOPT_READDATA, &filedata);
```

5. 告诉 libcurl，执行的是上传操作。

```
curl_easy_setopt(easy_handle, CURLOPT_UPLOAD, 1L);
```

有些协议在没有预先知道上传文件大小的情况下，可能无法正确判断上传是否结束，所以最好预先使用 CURLOPT_INFILESIZE_LARGE 属性： 告诉它要上传文件的大小：

```
/* in this example, file_size must be an curl_off_t variable */
curl_easy_setopt(easy_handle, CURLOPT_INFILESIZE_LARGE, file_size);
```

6. 调用 curl_easy_perform。

接下来，libcurl 将会完成剩下的所有工作。在上传文件过程中，libcurl 会不断调用先前设置的回调函数，用于将要上传的数据读入到缓冲区，并执 行上传。

下面的例子演示如何将文件上传到 FTP 服务器。笔者使用的是 IIS 自带的 FTP 服务，同时在 FTP 上设置了可写权限。

```
/**
```

```
 *      @brief 读取数据的回调。
 */
size_t read_data(void *buffer, size_t size, size_t nmemb, void *user_p)
{
        return fread(buffer, size, nmemb, (FILE *)user_p);
}


int main(int argc, char **argv)
{
        // 初始化 libcurl
        CURLcode code;
        code = curl_global_init(CURL_GLOBAL_WIN32);
        if (code != CURLE_OK)
        {
                cerr << "init libcurl failed." << endl;
                return -1;
        }


        FILE *fp = fopen("a.html", "rb");
        if (NULL == fp)
        {
                cout << "can't open file." << endl;
                curl_global_cleanup();
                return -1;
        }


        // 获取文件大小
        fseek(fp, 0, 2);
        int file_size = ftell(fp);
        rewind(fp);


        // 获取 easy handle

        CURL *easy_handle = NULL;
        easy_handle = curl_easy_init();
        if (NULL == easy_handle)
        {
                cerr << "get a easy handle failed." << endl;
                fclose(fp);
                curl_global_cleanup();
```

```
                return -1;
        }


        // 设置 eash handle 属性
        curl_easy_setopt(easy_handle, CURLOPT_URL, ftp://127.0.0.1/upload.html);
        curl_easy_setopt(easy_handle, CURLOPT_UPLOAD, 1L);
        curl_easy_setopt(easy_handle, CURLOPT_READFUNCTION, &read_data);
        curl_easy_setopt(easy_handle, CURLOPT_READDATA, fp);
        curl_easy_setopt(easy_handle, CURLOPT_INFILESIZE_LARGE, file_size);


        // 执行上传操作
        code = curl_easy_perform(easy_handle);
        if (code == CURLE_OK)
        {
                cout << "upload successfully." << endl;
        }


        // 释放资源
        fclose(fp);
        curl_easy_cleanup(easy_handle);
        curl_global_cleanup();


        return 0;
}
```

关于密码

客户端向服务器发送请求时，许多协议都要求提供用户名与密码。libcurl 提供了多种方式来设置它们。

一些协议支持在 URL 中直接指定用户名和密码，类似于： protocol://user:password@example.com/path/。libcurl 能正确的识别这种 URL 中的用户名与密码并执行 相应的操作。如果你提供的用户名和密码中有特殊字符，首先应该对其进行 URL 编码。

也可以通过 CURLOPT_USERPWD 属性来设置用户名与密码。参数是格式如 “user:password ”的字符串：

```
curl_easy_setopt(easy_handle, CURLOPT_USERPWD, "user_name:password");
```

（下面这几段文字我理解地模模糊糊）有时候在访问代理服务器的时候，可能时时要求提供用户名和密码进行用户身份验证。这种情况下，libcurl 提供了另 一个属性

CURLOPT_PROXYUSERPWD：

```
curl_easy_setopt(easy_handle, CURLOPT_PROXYUSERPWD, "user_name:password");
```

在 UNIX 平台下，访问 FTP 的用户名和密码可能会被保存在$HOME/.netrc 文件中。libcurl 支持直接从这个文件中获取用户名与密码：

```
curl_easy_setopt(easy_handle, CURLOPT_NETRC, 1L);
```

在使用 SSL 时，可能需要提供一个私钥用于数据安全传输，通过 CURLOPT_KEYPASSWD 来设置私钥：

```
curl_easy_setopt(easy_handle, CURLOPT_KEYPASSWD, "keypassword");
```

HTTP 验证

上一章介绍了如何在 libcurl 中，对需要身份验证的 URL 设置用户名与密码。在使用 HTTP 协议时，客户端有很多种方式向服务器提供验证信息。默认的 HTTP 验证方法是"Basic"，它将用户名与密码以明文的方式、经 Base64 编码后保存在 HTTP 请求头中，发往服务器。当然这不太安全。

当前版本的 libcurl 支持的验证方法有：basic, Digest, NTLM, Negotiate, GSS-Negotiate and SPNEGO。（译者感叹：搞 Web 这么多年，尽然不知道这些 Http 的验证方式，实在惭愧。）可以通过 CURLOPT_HTTPAUTH 属性来设置具体 的验证方式：

```
curl_easy_setopt(easy_handle, CURLOPT_HTTPAUTH, CURLAUTH_DIGEST);
```

向代理服务器发送验证信息时，可以通过 CURLOPT_PROXYAUTH 设置验证方式：

```
curl_easy_setopt(easy_handle, CURLOPT_PROXYAUTH, CURLAUTH_NTLM);
```

也可以同时设置多种验证方式（通过按位与）， 使用`CURLAUTH_ANY`将允许 libcurl 可以选择任何它所支持的验证方式。通过 CURLOPT_HTTPAUTH 或 CURLOPT_PROXYAUTH 属性设置的多种验证方式，libcurl 会在运行时选择一种它认为是最好的方式与服务器通信：

```
curl_easy_setopt(easy_handle, CURLOPT_HTTPAUTH,  CURLAUTH_DIGEST|CURLAUTH_BASIC);
// curl_easy_setopt(easy_handle, CURLOPT_HTTPAUTH,  CURLAUTH_ANY);
```

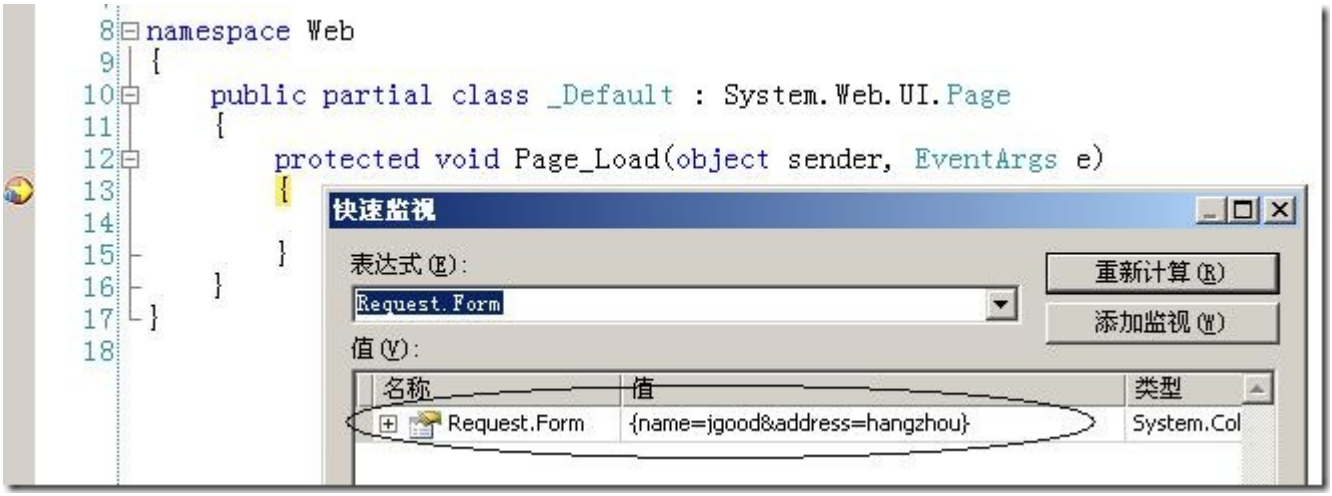
HTTP Post

这一章介绍如何使用 libcurl 以 Post 方式向 HTTP 服务器提交数据。

方法一，也是最简单的方式，就像 html 中使用<form>标签提交数据一样，只需向 libcurl 提供一个包含数据的字符串即可。下面是笔 者学习过程中的一个 demo 程序：

```
int main(int argc, char **argv)
{
	code = curl_global_init(CURL_GLOBAL_WIN32);
	CURL *easy_handle = curl_easy_init();


	curl_easy_setopt(easy_handle, CURLOPT_URL,
http://localhost:2210/Default.aspx);
	// 单个域 post
	curl_easy_setopt(easy_handle, CURLOPT_POSTFIELDS,
"name=jgood&address=hangzhou");
	code = curl_easy_perform(easy_handle);

	curl_easy_cleanup(easy_handle);
	curl_global_cleanup();


	return 0;
}
```

在 asp.net Web 服务器上跟踪调试，得到客户程序提交上来的数据，下面是截图：

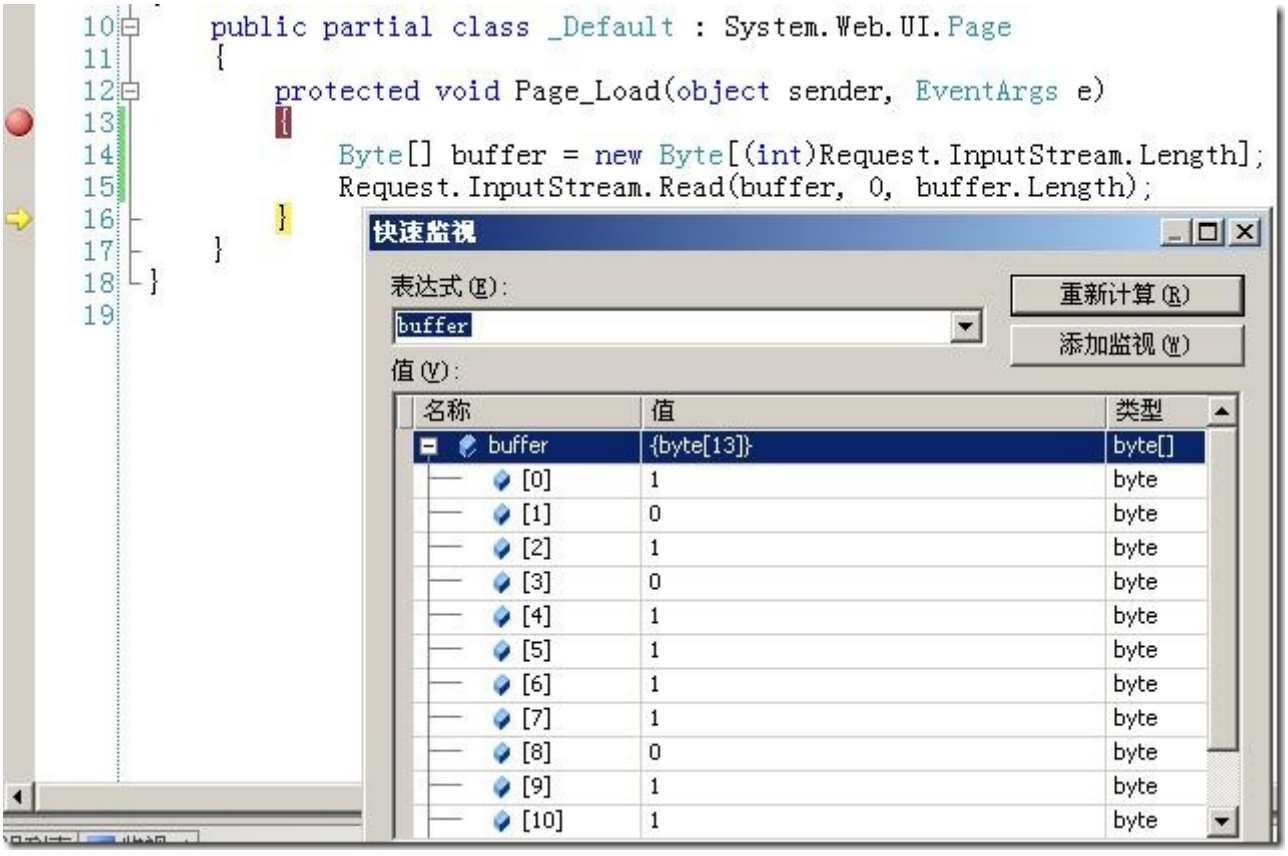


上面的代码够简单吧~_~ 有时候，我们需要提交一些二进制数据到 HTTP 服务器，使用方法一就不行了，因为方法一中实际提交的是一个字符串，字符串遇到\0 就表示结束了。所以在上 传二进制数据的时候，必须明确的告诉 libcurl 要提交的数据的长度。在上传二进制数据的时候，还应该设置提交的 Content-Type 头信息。下面 的示例代码：

```
int main(int argc, char **argv)
{
	curl_global_init(CURL_GLOBAL_WIN32);
	CURL *easy_handle = curl_easy_init();


	// 上传二进制数据
	char data[] = { 1, 0, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 0 };
```

```
        curl_slist *http_headers = NULL;
        http_headers = curl_slist_append(http_headers, "Content-Type: text/xml");

        curl_easy_setopt(easy_handle, CURLOPT_HTTPHEADER, http_headers);
        curl_easy_setopt(easy_handle, CURLOPT_URL,
http://localhost:2210/Default.aspx);
        curl_easy_setopt(easy_handle, CURLOPT_POSTFIELDS, data);
        curl_easy_setopt(easy_handle, CURLOPT_POSTFIELDSIZE, sizeof(data));


        curl_easy_perform(easy_handle);


        curl_slist_free_all(http_headers);
        curl_easy_cleanup(easy_handle);
        curl_global_cleanup();


        return 0;
}
```

在 asp.net Web 服务器上跟踪调试，得到客户程序提交上来的二进制数据，下面是截图：



上面介绍的两种方式，可以完成大部分的 HTTP POST 操作。但上面的两种方式都不支持 multi-part formposts。Multi-part formposts 被认为是提交二进制数据(或大量数据)的更好方法，可以在 RFC1867, RFC2388 中找到他们的定义。何为 Multi-part？其实，就我理解，就是在 Post 提交的时候，有不同的数据单元，每个单元有自己的名称与内 容，内容可以是文本的，也可以是二进制的。同时，每个数据单元都可以有自己的消息头，MIME 类型，这些数据单元组成一个链表，提交到 HTTP 服务器。 libcurl 提供了方便的 api 用于支持 multi-part formposts。使用 curl_formadd 函数，可以添加不同的数据数据单元，然后提交到服务器。下面是一个 multi-part formposts 的例子（更详细的使用，请参考：http://curl.haxx.se/libcurl/c/curl_formadd.html）：

```
int main()
{

	curl_global_init(CURL_GLOBAL_WIN32);
	CURL *easy_handle = curl_easy_init();

		// 使用 multi-parts form post
		curl_easy_setopt(easy_handle, CURLOPT_URL,
http://localhost:2210/Default.aspx);
		curl_httppost *post = NULL;
		curl_httppost *last = NULL;

		// 文本数据
		curl_formadd(&post, &last, CURLFORM_COPYNAME, "name", CURLFORM_COPYCONTENTS,
"JGood", CURLFORM_END);
		curl_formadd(&post, &last, CURLFORM_COPYNAME, "address", CURLFORM_COPYCONTENTS,
"HangZhou", CURLFORM_END);

		// 文本文件中的数据
		curl_formadd(&post, &last, CURLFORM_COPYNAME, "file", CURLFORM_FILECONTENT,
"ReadMe.txt", CURLFORM_END);
		curl_easy_setopt(easy_handle, CURLOPT_HTTPPOST, post);
		curl_easy_perform(easy_handle);

		curl_formfree(post);
		curl_easy_cleanup(easy_handle);
		curl_global_cleanup();


		return 0;
}
```

最后要说明的是，所有在 easy handle 上设置的属性都是"sticky"的，什么意思？就是说在 easy handle 上设置的属性都将被保存，即使执行完 curl_easy_perform 之后，这些属性值仍然存在。通过将 CURLOPT_HTTPGET 设 为 1 可以使 easy handle 回到最原始的状态：

```
curl_easy_setopt(easy_handle, CURLOPT_HTTPGET, 1L);
```

显示进度

libcurl 支持通信过程中的进度控制。通过将 CURLOPT_NOPROCESS 设置为 0 开启进度支持。该选项默认值为 1。对大多数应用程序，我们需 要提供一个进度显示回调。libcurl 会不定期的将当前传输的进度通过回调函数告诉你的程序。回调函数的原型如下：

```
int progress_callback(void *clientp, double dltotal, double dlnow, double ultotal, double ulnow);
```

通过 CURLOPT_PROGRESSFUNCTION 注册该回调函数。参数 clientp 是一个用户自定义指针，应用程序通过 CURLOPT_PROCESSDATA 属性将该自定义指定传递给 libcurl。libcurl 对该参数不作任何处理，只是简单将其传递给回调函数。

在 C++中使用 libcurl

在 C++中使用 libcurl 跟在 C 语言中没有任何区别，只有一个地方要注意：回调函数不能是类的非静态成员函数。例如：

```
class AClass {
     static size_t write_data(void *ptr, size_t size, size_t nmemb, void *ourpointer)
     {
       /* do what you want with the data */
     }
}
```

代理

什么是代理？Merrian-Webster 的解释是：一个通过验证的用户扮演另一个用户。今天，代理已经被广泛的使用。许多公司提供网络代理服务器，允 许员工的网络客户端访问、下载文件。代理服务器处理这些用户的请求。

libcurl 支持 SOCKS 和 HTTP 代理。使用代理，libcurl 会把用户输入的 URL 提交给代理服务器，而不是直接根据 URL 去访问远程资源。

当前版本的 libcurl 并不支持 SOCKS 代理的所有功能。

对于 HTTP 代理来说，即使请求的 URL 不是一个合法的 HTTP URL（比方你提供了一个 ftp 的 url），它仍然会先被提交到 HTTP 代理。

代理选项

CURLOPT_PROXY 属性用于设置 libcurl 使用的代理服务器地址：

```
curl_easy_setopt(easy_handle, CURLOPT_PROXY, "proxy-host.com:8080");
```

可以把主机名与端口号分开设置：

```
curl_easy_setopt(easy_handle, CURLOPT_PROXY, "proxy-host.com");
curl_easy_setopt(easy_handle, CURLOPT_PROXYPORT, "8080");  // 端口号是用字符串还是整数？？
```

有些代理服务器要求用户通过验证之后才允许接受其请求，此时应该先提供验证信息：

```
curl_easy_setopt(easy_handle, CURLOPT_PROXYUSERPWD, "user:password");
```

还要告诉 libcurl 使用的代理类型（如果没有提供，libcurl 会认为是 HTTP 代理）：

```
curl_easy_setopt(easy_handle, CURLOPT_PROXYTYPE, CURLPROXY_SOCKS4);
```

环境变量

对于有些协议，libcurl 会自动检测并使用一些环境变量，并根据这些环境变量来确定要使用的代理服务器。这些环境变量的名称格式一般是" [protocol]_proxy"（注意小写）。例如输入一个 HTTP 的 URL，那么名称为"http_proxy"的环境变量就会被检测是否存在，如 果存在，libcurl 会使用该环境变量指定的代理。相同的规则也适用于 FTP。

这些环境变量的值的格式必须是这样的："[protocol://][user:password@]machine[:port]"。libcurl 会 忽略掉[protocol://]，如果没有提供端口号，libcurl 使用该协议的默认端口。

有两个比较特殊的环境变量：'all_proxy'与'no_proxy'。如果一个 URL 所对应的协议，它的环境变量没有设置，那么 'all_proxy'指定的代理将被使用。'no_proxy'则指定了一个不应被使用的代理主机的列表。例如：no_proxy 的值是 '192.168.1.10'，即使存在 http_proxy，它的值也是'192.168.1.10'，'192.168.1.10'也不会被作为代 理。no_proxy=”*”表示不允许使用任何代理。

显式地将 CURLOPT_PROXY 属性设置为空，可以禁止 libcurl 检查并使用环境变量来使用代理。

SSL 和代理

SSL 为点到点通信提供安全保障。它包含一些强壮的加密措施和其他安全检测，这使得上面讲到的代理方式不适用于 SSL。除非代理服务器提供专用通道，对进 出该代理服务器的数据不作任何检测或禁止。通过 HTTP 代理服务器打开 SSL 连接，意味着代理服务器要直接连接到目标主机的指定端口。因为代理服务器对在 专用通道上传输的数据的类型毫无所知，所以它往往会使某些机制失效，如缓存机制。许多组织只允许在 443 端口上创建这种类型的数据通道。

代理通道(Tunneling Through Proxy)

正如上面讲到的，要使 SSL 工作必须在代理服务器创建专用数据通道，通常专用通道只被限制应用于 HTTPS。通过 HTTP 代理在应用程序与目标之间创建一 个专用数据通道，应该预防在该专有通道上执行非 HTTP 的操作，如进行 FTP 上传或执行 FTP 命令。代理服务器管理员应该禁止非法的操作。

通过 CURLOPT_HTTPPROXYTUNNEL 属性来告诉 libcurl 使用代理通道：

```
curl_easy_setopt(easy_handle, CURLOPT_HTTPPROXYTUNNEL, 1L);
```

有时候你想通过代理通道执行平常的 HTTP 操作，而实际上却可能使你不经过代理服务器而直接与远程主机进行交互。libcurl 不会代替这种新引入的行 为。

自动配置代理

许多浏览器支持自动配置代理，例如 NetScape。libcurl 并不支持这些。

持久化的好处(Persistence Is The Way to Happiness)

当需要发送多次请求时，应该重复使用 easy handle。

每次执行完 curl_easy_perform，licurl 会继续保持与服务器的连接。接下来的请求可以使用这个连接而不必创建新的连接（如果目标主机 是同一个的话）。这样可以减少网络开销。

即使连接被释放了，libcurl 也会缓存这些连接的会话信息，这样下次再连接到目标主机上时，就可以使用这些信息，从而减少重新连接所需的时间。

FTP 连接可能会被保存较长的时间。因为客户端要与 FTP 服务器进行频繁的命令交互。对于有访问人数上限的 FTP 服务器，保持一个长连接，可以使你不需要 排除等待，就直接可以与 FTP 服务器通信。

libcurl 会缓存 DNS 的解析结果。

在今后的 libcurl 版本中，还会添加一些特性来提高数据通信的效率。

每个 easy handle 都会保存最近使用的几个连接，以备重用。默认是 5 个。可以通过 CURLOPT_MAXCONNECTS 属性来设置保存连接的数量。

如果你不想重用连接，将 CURLOPT_FRESH_CONNECT 属性设置为 1。这样每次提交请求时，libcurl 都会先关闭以前创建的连接，然后重 新创建一个新的连接。也可以将 CURLOPT_FORBID_REUSE 设置为 1，这样每次执行完请求，连接就会马上关闭。

libcurl 使用的 HTTP 消息头

当使用 libcurl 发送 http 请求时，它会自动添加一些 http 头。我们可以通过 CURLOPT_HTTPHEADER 属性手动替换、添加或删除相应 的 HTTP 消息头。

Host

http1.1（大部分 http1.0)版本都要求客户端请求提供这个信息头。

Pragma

"no-cache"。表示不要缓冲数据。

Accept

"*/*"。表示允许接收任何类型的数据。

Expect

以 POST 的方式向 HTTP 服务器提交请求时，libcurl 会设置该消息头为"100-continue"，它要求服务器在正式处理该请求之前，返回一 个"OK"消息。如果 POST 的数据很小，libcurl 可能不会设置该消息头。

自定义选项

当前越来越多的协议都构建在 HTTP 协议之上（如：soap），这主要归功于 HTTP 的可靠性，以及被广泛使用的代理支持（可以穿透大部分防火墙）。 这些协议的使用方式与传统 HTTP 可能有很大的不同。对此，libcurl 作了很好的支持。

自定义请求方式(CustomRequest)

HTTP 支持 GET, HEAD 或者 POST 提交请求。可以设置 CURLOPT_CUSTOMREQUEST 来设置自定义的请求方式，libcurl 默认以 GET 方式提交请求：

```
curl_easy_setopt(easy_handle, CURLOPT_CUSTOMREQUEST, "MYOWNREQUEST");
```

修改消息头

HTTP 协议提供了消息头，请求消息头用于告诉服务器如何处理请求；响应消息头则告诉浏览器如何处理接收到的数据。在 libcurl 中，你可以自由的添加 这些消息头：

```
struct curl_slist *headers=NULL; /* init to NULL is important */
headers = curl_slist_append(headers, "Hey-server-hey: how are you?");
headers = curl_slist_append(headers, "X-silly-content: yes");
/* pass our list of custom made headers */
curl_easy_setopt(easyhandle, CURLOPT_HTTPHEADER, headers);
curl_easy_perform(easyhandle); /* transfer http */
curl_slist_free_all(headers); /* free the header list */
```

对于已经存在的消息头，可以重新设置它的值：

```
headers = curl_slist_append(headers, "Accept: Agent-007");
headers = curl_slist_append(headers, "Host: munged.host.line");
```

删除消息头

对于一个已经存在的消息头，设置它的内容为空，libcurl 在发送请求时就不会同时提交该消息头：

```
headers = curl_slist_append(headers, "Accept:");
```

强制分块传输(Enforcing chunked transfer-encoding)

（这段文字理解可能有误码）以非 GET 的方式提交 HTTP 请求时，如果设置了自定义的消息头"Transfer- Encoding:chunked"，libcurl 会分块提交数据，即使要上传的数据量已经知道。在上传数据大小未知的情况下，libcurl 自动采用 分块上传数据。（译者注：非 GET 方式提交请求，提交的数据量往往比较大。）

HTTP 版本

每一次 http 请求，都包含一个表示当前使用 http 版本的消息头。libcurl 默认使用 HTTP 1.1。可以通过 CURLOPT_HTTP_VERSION 属性来设置具体的版本号：

```
curl_easy_setopt(easy_handle, CURLOPT_HTTP_VERSION, CURL_HTTP_VERSION_1_0);
```

FTP 自定义命令

并不是所以的协议都像 HTTP 那样，通过消息头来告诉服务器如何处理请求。对于 FTP，你就要使用另外的方式来处理。

发送自定义的命令到 ftp 服务器，意味着你发送的命令必须是能被 ftp 服务器理解的命令（FTP 协议中定义的命令，参考 rfc959）。

下面是一个简单的例子，在文件传输操作操作之前删除指定文件：

```
headers = curl_slist_append(headers, "DELE file-to-remove");
/* pass the list of custom commands to the handle */

curl_easy_setopt(easyhandle, CURLOPT_QUOTE, headers);
// curl_easy_setopt(easyhandle, CURLOPT_POSTQUOTE, headers); // 在数据传输之后操行删除操作
curl_easy_perform(easyhandle); /* transfer ftp data! */
curl_slist_free_all(headers); /* free the header list */
```

FTP 服务器执行命令的顺序，同这些命令被添加到列表中顺序是一致的。发往服务器的命令列表中，只要有一个命令执行失败，ftp 服务器就会返回一个错误代 码，此时 libcurl 将直接返回 CURLE_QUOTE_ERROR，不再执行剩余的 FTP 命令。

将 CURLOPT_HEADER 设置为 1，libcurl 获取目标文件的信息，并以 HTTP 消息头的样式来输出消息头。

FTP 自定义 CUSTOMREQUEST

使用 CURLOPT_CUSTOMREQUEST 属性，可以向 FTP 服务器发送命令。"NLST"是 ftp 默认的列出文件列表的命令。 下面的代码用于列出 FTP 服务器上的文件列表：

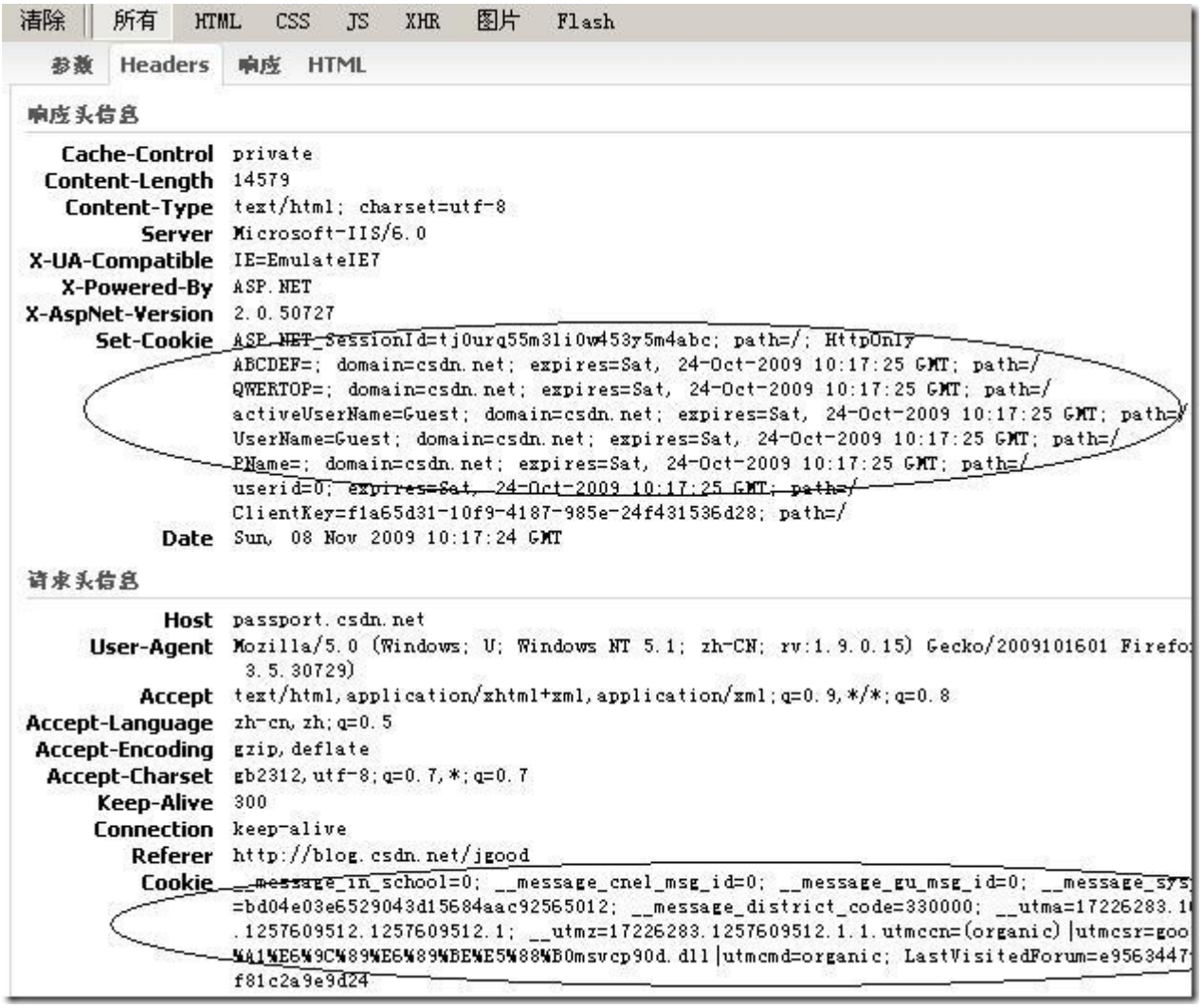
```
int main(int argc, char **argv)
{
        curl_global_init(CURL_GLOBAL_WIN32);
```

```
        CURL *easy_handle = curl_easy_init();
        curl_easy_setopt(easy_handle, CURLOPT_URL, "ftp://127.0.0.1/");
        curl_easy_setopt(easy_handle, CURLOPT_CUSTOMREQUEST, "NLST");
        curl_easy_perform(easy_handle);

        curl_easy_cleanup(easy_handle);
        curl_global_cleanup();


        return 0;
}
```

Cookies Without Chocolate Chips

cookie 是一个键值对的集合，HTTP 服务器发给客户端的 cookie，客户端提交请求的时候，也会将 cookie 发送到服务器。服务器可以根据 cookie 来跟踪用户的会话信息。cookie 有过期时间，超时后 cookie 就会失效。cookie 有域名和路径限制，cookie 只能发给指定域名 和路径的 HTTP 服务器。

cookie 以消息头"Set-Cookie"的形式从 HTTP 服务器发送到客户端；客户端发以消息头"Cookie"的形式将 Cookie 提交到 HTTP 服务器。为了对这些东西有个直观的概念，下图是 FireFox 中，使用 Firebug 跟踪到的 cookie 消息头：



在 libcurl 中，可以通过 CURLOPT_COOKIE 属性来设置发往服务器的 cookie：

```
curl_easy_setopt(easy_handle, CURLOPT_COOKIE, "name1=var1; name2=var2;");
```

下面的例子演示了如何使用 libcurl 发送 cookie 信息给 HTTP 服务器，代码非常的简单：

```
int main(int argc, char **argv)
{
        curl_global_init(CURL_GLOBAL_WIN32);
        CURL *easy_handle = curl_easy_init();


        curl_easy_setopt(easy_handle, CURLOPT_URL,
http://localhost:2210/Default.aspx);
        curl_easy_setopt(easy_handle, CURLOPT_COOKIE, "name=JGood; address=HangZhou");


        curl_easy_perform(easy_handle);
```

```
        curl_easy_cleanup(easy_handle);
        curl_global_cleanup();


        return 0;
}
```

下图是在 ASP.NET Web 服务器上调试时跟踪到的 Cookie 数据：

在实在的应用场景中，你可能需要保存服务器发送给你的 cookie，并在接下来的请求中，把这些 cookie 一并发往服务器。所以，可以把上次从服务器收 到的所有响应头信息保存到文本文件中，当下次需要向服务器发送请求时，通过 CURLOPT_COOKIEFILE 属性告诉 libcurl 从该文件中读取 cookie 信息。

设置 CURLOPT_COOKIEFILE 属性意味着激活 libcurl 的 cookie parser。在 cookie parser 被激活之前，libcurl 忽略所以之前接收到的 cookie 信息。cookie parser 被激活之后，cookie 信息将被保存内存中，在接下来的请求中，libcurl 会自动将这些 cookie 信息添加到消息头里，你的应用程序 不需要做任何事件。大多数情况下，这已经足够了。需要注意的是，通过 CURLOPT_COOKIEFILE 属性来激活 cookie parser，给 CURLOPT_COOKIEFILE 属性设置的一个保存 cookie 信息的文本文件路径，可能并不需要在磁盘上物理存在。

如果你需要使用 NetScape 或者 FireFox 浏览器的 cookie 文件，你只要用这些浏览器的 cookie 文件的路径来初始化 CURLOPT_COOKIEFILE 属性，libcurl 会自动分析 cookie 文件，并在接下来的请求过程中使用这些 cookie 信息。

libcurl 甚至能够把接收到的 cookie 信息保存成能被 Netscape/Mozilla 的浏览器所识别的 cookie 文件。通过把这些称为 cookie-jar。通过设置 CURLOPT_COOKIEJAR 选项，在调用 curl_easy_cleanup 释放 easy handle 的时候，所有的这些 cookie 信息都会保存到 cookie-jar 文件中。这就使得 cookie 信息能在不同的 easy handle 甚至在浏览器之间实现共享。

FTP Peculiarities We Need

在使用 FTP 协议进行数据传输的时候，需要创建两个连接。第一个连接用于传输控制命令，另一个连接用于传输数据。（关于 FTP 的通信过程，请参考这篇文 章：http://www.wangjia.net/bo-blog/post/698/）。FTP 通信需要创建两个连接这个事实往往被很多人忽略。根据第二个连接的发起方是谁，可以分为主动模式与被动模式。libcurl 对此都提供了支持。 libcurl 默认使用被动模式，因为被动模式可以方便的穿透防火墙，NAT 等问题。在被动模式下，libcurl 要求 ftp 服务器打开一个新的端口监 听，然后 libcurl 连接该端口用于数据传输。如果使用主动模式，程序必须告诉 FTP 服务器你监听的 IP 与端口，通过设置 CURLOPT_FTPPORT 属性来完成。

Headers Equal Fun

（这段文字我理解的很模糊，请读者参考原文）有些协议提供独立于正常数据的 消息头、meta-data。正常的数据流里通常不包括 信息头和元数据。可以将 CURLOPT_HEADER 设置为 1，使信息头、元数据也能出现在数据流中。libcurl 的强大之处在于，它能够从数据流中解 析出消息头，....

Post Transfer Information

[curl_easy_getinfo]

安全考虑

请参考原文，此处略。

使用 multi interface 同时进行多项传输

上面介绍的 easy interface 以同步的方式进行数据传输，curl_easy_perform 会一直阻塞到数据传输完毕后返回，且一次操作只能发送一次请求，如果要 同时发送多个请求，必须使用多线程。

而 multi interface 以一种简单的、非阻塞的方式进行传输，它允许在一个线程中，同时提交多个相同类型的请求。 在使用 multi interface 之前，你应该掌握 easy interface 的基本使用。因为 multi interface 是建立在 easy interface 基础之上的，它只是简单的将多个 easy handler 添加到一个 multi stack，而后同时传输而已。

使用 multi interface 很简单，首先使用 curl_multi_init()函数创建一个 multi handler，然后使用 curl_easy_init()创建一个或多个 easy handler，并按照上面几章介绍的接口正常的设置相关的属性，然后通过 curl_multi_add_handler 将这些 easy handler 添加到 multi handler，最后调用 curl_multi_perform 进行数据传输。

curl_multi_perform 是异步的、非阻塞的函数。如果它返回 CURLM_CALL_MULTI_PERFORM，表示数据通信正在进行。

通过 select()来操作 multi interface 将会使工作变得简单（译者注：其实每个 easy handler 在低层就是一个 socket，通过 select()来？？？理这些 socket，在有数据可读/可写/异常的时候，通知应用程序）。在调用 select()函数之前，应该使用 curl_multi_fdset 来初始化 fd_set 变量。

select()函数返回时，说明受管理的低层 socket 可以操作相应的操作（接收数据或发送数据，或者连接已经断开），此时应该马上调用 curl_multi_perform，libcurl 将会执行相应操作。使用 select()时，应该设置一个较短的超时时间。在调用 select() 之前，造成不要忘记通过 curl_multi_fdset 来初始化 fd_set，因为每次操作，fd_set 中的文件描述符可能都不一样。

如果你想中止 multi stack 中某一个 easy handle 的数据通信，可以调用 curl_multi_remove_handle 函数将其从 multi stack 中取出。千万另忘记释放掉 easy handle（通过 curl_easy_cleanup()函数）。

当 multi stack 中的一个 eash handle 完成数据传输的时候，同时运行的传输任务数量就会减少一个。当数量降到 0 的时候，说明所有的数据传输已经完成。

curl_multi_info_read 用于获取当前已经完成的传输任务信息，它返回每一个 easy handle 的 CURLcode 状态码。可以根据这个状态码来判断每个 easy handle 传输是否成功。

下面的例子，演示了如何使用 multi interface 进行网页抓取：

```
int main(int argc, char **argv)
{
        // 初始化
        curl_global_init(CURL_GLOBAL_WIN32);
```

```
        CURLM *multi_handle = NULL;
        CURL *easy_handle1 = NULL;
        CURL *easy_handle2 = NULL;

        extern size_t save_sina_page(void *buffer, size_t size, size_t count, void
*user_p);
        extern size_t save_sohu_page(void *buffer, size_t size, size_t count, void
*user_p);
        FILE *fp_sina = fopen("sina.html", "ab+");
        FILE *fp_sohu = fopen("sohu.html", "ab+");

        multi_handle = curl_multi_init();

        // 设置 easy handle
        easy_handle1 = curl_easy_init();
        curl_easy_setopt(easy_handle1, CURLOPT_URL, "http://www.sina.com.cn");
        curl_easy_setopt(easy_handle1, CURLOPT_WRITEFUNCTION, &save_sina_page);
        curl_easy_setopt(easy_handle1, CURLOPT_WRITEDATA, fp_sina);

        easy_handle2 = curl_easy_init();
        curl_easy_setopt(easy_handle2, CURLOPT_URL, "http://www.sohu.com");
        curl_easy_setopt(easy_handle2, CURLOPT_WRITEFUNCTION, &save_sohu_page);
        curl_easy_setopt(easy_handle2, CURLOPT_WRITEDATA, fp_sohu);

        // 添加到 multi stack
        curl_multi_add_handle(multi_handle, easy_handle1);
        curl_multi_add_handle(multi_handle, easy_handle2);

        //
        int running_handle_count;
        while (CURLM_CALL_MULTI_PERFORM == curl_multi_perform(multi_handle,
&running_handle_count))
        {
                cout << running_handle_count << endl;
        }

        while (running_handle_count)
        {
                timeval tv;
                tv.tv_sec = 1;
                tv.tv_usec = 0;

                int max_fd;
                fd_set fd_read;
```

```
                fd_set fd_write;
                fd_set fd_except;

                FD_ZERO(&fd_read);
                FD_ZERO(&fd_write);
                FD_ZERO(&fd_except);

                curl_multi_fdset(multi_handle, &fd_read, &fd_write, &fd_except,
&max_fd);
                int return_code = select(max_fd + 1, &fd_read, &fd_write, &fd_except,
&tv);
                if (SOCKET_ERROR == return_code)
                {
                        cerr << "select error." << endl;
                        break;
                }
                else
                {
                        while (CURLM_CALL_MULTI_PERFORM ==
curl_multi_perform(multi_handle, &running_handle_count))
                        {
                                cout << running_handle_count << endl;
                        }
                }
        }

        // 释放资源
        fclose(fp_sina);
        fclose(fp_sohu);
        curl_easy_cleanup(easy_handle1);
        curl_easy_cleanup(easy_handle2);
        curl_multi_cleanup(multi_handle);
        curl_global_cleanup();

        return 0;
}

size_t save_sina_page(void *buffer, size_t size, size_t count, void *user_p)
{
        return fwrite(buffer, size, count, (FILE *)user_p);
}

size_t save_sohu_page(void *buffer, size_t size, size_t count, void *user_p)
{
```

```
        return fwrite(buffer, size, count, (FILE *)user_p);
}
```

SSL, 证书，其他技巧

[seeding, passwords, keys, certificates, ENGINE, ca certs]

在 easy handler 之间共享数据