基于 libcurl 的通用网络传输库的实现

通用网络传输库

libcurl 是一个易用的，支持多种网络协议，跨平台的 URL 传输库，在现实世界中有着广泛的应用，本文通过对 libcurl 进行一定的封装，来实现一个通用的网络库，这个库可以用来完成诸如自定义应用协议的传输，文件的上传下载等功能。

邱 俊涛, 软件工程师, Jinfonet
关闭 [x]
邱俊涛，毕业于昆明理工大学计算机科学与技术专业，对机械控制、电子、人工智能等方面有浓厚的兴趣，对计算机科学的底层比较熟悉。喜欢 C/Java/Python 等语言。

2011 年 1 月 11 日

- 内容

基础知识

libcurl 简介

libcurl 是一个易用的，支持多协议的 URL 传输库，支持众多的协议，如 FTP, HTTP, HTTPS, IMAP, POP3, SMTP, TELNET 等，同时，libcurl 支持 SSL 验证，基于 HTTP 协议的文件上传，代理，客户端缓冲等。使用 libcurl，可以很轻松的实现一个简单的 WEB 浏览器，或者一个 FTP 的客户端应用，邮件客户端等。

libcurl 具有很好的移植性，她可以很好的工作在主流的操作系统如 Solaris, FreeBSD, GNU Linux, Windows 等平台。很多 Linux 的发行版中都会自带 libcurl 库，并且提供一个 curl 的命令行工具，使用 curl，可以很方便的对基于 WEB 的应用进行测试。

通用网络库功能描述

本文中提到的通用网络库，主要功能为：通过对 libcurl 的封装，实现自定义应用层协议的传输 (发送 / 接收)，即对原生的字节流的传输；文件的上传下载，在传输较大的文件时，我们需要事实的向应用程序报告传输的进度，因此需要实现传输中的进度回调；在遇到错误或者异常时，需要提供错误处理机制。

回页首

网络库的设计与实现

通用网络库包括两个模块：内存管理模块及数据传输模块，内存管理模块主要负责大文件传输时的内存分配，释放，修改 (扩大 / 缩小) 原有的缓冲区等。文件传输模块主要负责自定义应用协议的传输，定长文件的传输，不定长文件的传输，读文件至内存，发送内存中的文件等。

内存管理模块

内存管理模块通过使用标准库中的 malloc 或者 realloc 来分配内存，并使用标准库中的 free 来释放申请到的内存。在网络库中，对内存的使用尽量使用内存管理接口中定义的 API，这样有助于错误的排查，也利于各模块各司其职。

内存管理接口

内存管理模块处理的最小单元为一个结构体，mem_block：

```
/**
* 内存块数据结构
*/
typedefstruct{
    unsignedchar*memory; /* 内存块内容 */
    size_t size;  /* 内存块长度 */
}mem_block;
```

基于这个数据结构，定义了 5 个 API，如下：

```
/**
* 说明：初始化内存块 (mem_block 结构指针 )
*
* 参数：
* block : 内存块指针
*
* 返回值：
* 无
*
*/
voidmem_block_init(mem_block *block);
```

```
/**
* 说明：释放内存块 (mem_block 结构指针 )
*
* 参数：
* block : 内存块指针
*
* 返回值：
* 无
*
*/
voidmem_block_free(mem_block *block);

/**
* 说明：重新分配内存块大小
*
* 参数：
* ptr ：原始指针
* size : 新的块大小
*
* 返回值：
* void * : 分配成功
* NULL ：分配失败 , 详细错误可查看 errno
*/
void*mem_block_realloc(void*ptr, size_t size);

/**
* 说明：根据 block 克隆一个新的 mem_block
*
* 参数：
* block ：原始内存块
*
* 返回值：
* mem_block * : 克隆成功
* NULL       : 克隆失败 , 详细错误可查看 errno
*/
mem_block *mem_block_dup(constmem_block *block);

/**
* 说明：根据 memory 和 length 来初始化 block
*
* 参数：
* block ：原始内存块
* memory : 数据块
* length : 数据块长度
*
* 返回值：
* 无
*/
voidmem_block_new(mem_block *block, char*memory, unsignedintlength);
```

内存管理实现

内存管理模块中，较为重要的 API 为 mem_block_new，mem_block_new 会在内部调用 mem_block_realloc 进行内存的分配：

```
voidmem_block_new(mem_block *block, char*memory, unsignedintlength){
        if(block == NULL){
                return;
        }

        if(memory == NULL || length <= 0){
```

```
            return;
        }
        block->memory =
mem_block_realloc(block->memory, \
block->size+strlen(memory)+1);
        strncpy(block->memory, memory, length);
        block->size = length;
}
```

mem_block_realloc 会根据传入的参数判断是重新修改已有的内存还是新开辟一块新的内存以供使用。如果传入的第一个参数为空，则新开辟一块新的内存返回，否则，调用 realloc 来修改已有的内存块 (扩大 / 缩小)：

```
void *mem_block_realloc(void *ptr, size_t size){
        if(ptr != NULL){
                return realloc(ptr, size);
        }else{
                return malloc(size);
        }
}
```

如果成功，返回指向新空间的指针，否则返回 NULL。

数据传输模块

数据传输模块负责实际数据的传输，包括原生 (raw) 的字节流的发送 / 接收；文件的发送 / 接收，很多时候，接收文件时并不知道文件的大小，因此文件的接收 API 需要分配内存以容纳变长的文件；数据块的发送 / 接收；传输模块提供将文件写出 / 读如内存管理模块中定义的 mem_block 结构的接口。

数据传输接口

为了方便传输中的进度显示，我们需要定义一个回调函数，C 语言中，通常使用回调函数来完成异步事件，如事件 - 监听器的实现。在传输模块中，有一个函数指针的定义：

```
/**
* 为 UI 提供的处理进度回调接口
*/
typedefvoid(*processing)(doublecurrent, doubletotal);
```

这样，可以很容易通过这个回调函数来更新 UI 控件上的进度条或者如剩余时间等提示信息。在数据传输模块中，还定义了一下几个接口：

```
/**
* 说明：
* 向 url 发送长度为 req_len 的 req_buf，并接受数据至缓冲区 res_buf, 长度
* 为 res_len, res_buf 即 res_len 可以被服务端改写。
*
```

```
* 参数：
* url    ：需要读取的 URL( 统一资源描述 )
* req_buf : 请求数据缓冲区
* req_len : 请求数据缓冲区长度
* res_buf : 响应数据缓冲区指针
* res_len : 响应数据缓冲区长度指针
*
* 返回值：
* -1    : 操作失败
* 0     : 操作成功
*/
intraw_send_recv(constchar*url, void*req_buf, unsignedintreq_len, \
                                void**res_buf, unsignedint*res_len);

/**
* 说明：
* 打开长度为 file_size 的文件 file_name, 发送至 url 指定的位置 , 通过调用
* processing 更新进度
*
*/
intraw_send_file(constchar*url, constchar*file_name, \
                                constunsignedintfile_size, processing proc);

/**
* 说明：
* 从 url 指定的位置读文件，并存入文件 file_name, 通过调用 processing 更新进度
*
*/
intraw_recv_file(constchar*url, constchar*file_name, \
                                constunsignedintfile_size, processing proc);

/**
* 说明：
* 从 url 上读取定常 (n) 个字符，将内容存入 buffer, length 标识实际读到的长度 .
*
* 参数：
* url    : 需要读取的 URL( 统一资源描述 )
* buffer : 读入缓冲区
* length : 读入缓冲区长度
*
* 返回值：
* -1     : 读取失败
* length : 读取成功
*/
intraw_read_n( constchar*url, void*buffer, unsignedintlength);

/**
* 说明：
* 从 url 上读取不定长的内容到 buffer, 如果 buffer 较小，则 url_write_u 负责分配新的
* 内存并同时修改 length 值长度，内存由调用者负责释放
*
* 参数：
* url     : 需要读取的 URL( 统一资源描述 )
* p_buffer : 读入缓冲区的指针
* p_length : 读入缓冲区长度的指针
* proc   : 为更新 UI 而定义的回调函数
*
* 返回值：
*  -1    : 读取失败
* !-1    : 读取成功
*/
intraw_read_u(constchar*url, void**p_buffer, unsignedint*p_length, \
                        processing proc);

/**
* 说明：
```

```
* 向 url 上写入定长 (length) 个字符，写入内容在 buffer 内 , length 标识 buffer 长度 .
*
* 参数：
* url    : 需要写入的 URL( 统一资源描述 )
* buffer : 写入缓冲区
* length : 写入缓冲区长度
*
* 返回值：
* -1     : 写入失败
* length : 写入成功
*/
intraw_write_n(constchar*url, void*buffer, unsignedintlength);

/**
* 说明：
* 向 URL 写入定长字符串，写入的内容存储在 buffer 中，length 为 buffer 的长度
* raw_write_u 与 raw_write_n 的区别为：raw_write_u 的内部可能将 buffer 分
* 多次，多块传输。
*
* 参数：
* url    : 需要写入的 URL( 统一资源描述 )
* buffer : 写入缓冲区
* length : 写入缓冲区长度
* proc   : 为更新 UI 而定义的回调函数
*
* 返回值：
* -1     : 写入失败
* length : 写入成功
*/
intraw_write_u(constchar*url, void*buffer, unsignedintlength, \
                                processing proc);

/**
* 说明：
* 将 url 指定的文件读入内存块 block
*
* 参数：
* url    : 需要读取的 URL( 统一资源描述 )
* block : 内存块指针
*
* 返回值：
* -1     : 读取失败
* length : 读取成功，长度为 block 的 size
*
*/
inturl_read_mem(constchar*url, mem_block *block);

/**
* 说明：
* 将 block 指定内存块写入 url
*
* 参数：
* url    : 需要写入的 URL( 统一资源描述 )
* block : 内存块指针
*
* 返回值：
* -1     : 写入失败
* length : 写入成功，长度为 block 的 size
*/
inturl_write_mem(constchar*url, constmem_block *block);
```

这些 API 的参数，参数顺序，每个参数的意义，以及返回值等信息都在其头部的注释中描述。

数据传输实现

虽然 curl 提供了更高级的协议封装，比如 HTTP, FTP，SMTP 等协议，但是，大部分情况下，应用层的协议需要应用的实现者来自定义，比如在传输中，需要客户端和服务端定义好，第一个数据包的意义，第二个数据包的意义，后续的数据包的个数，以及分别的意义等，所以我们使用 curl 提供的 easy 类型的接口。

使用 easy 系列的 API 的步骤如下：

```
//CURL 结构，定义在 curl.h 中
CURL *curl;

// 初始化 CURL 结构
curl = curl_easy_init();

// 设置连接 URL，这里一般可能会有多个设置选项
curl_easy_setopt(curl, CURLOPT_URL, "your.host.url.here");

// 启动
curl_easy_perform(curl);

// 发送请求
curl_easy_send(curl, request, strlen(request), &iolen);

// 接受响应
curl_easy_recv(curl, buf, 1024, &iolen);

// 清理 curl 结构，释放资源等
curl_easy_cleanup(curl);
```

我们这里来看一下网络库的 raw_send_recv 接口，这个接口的作用为，发送定常的缓冲区到 url 上，并尝试读取响应信息，如果为读 / 写的连接超时，则错误返回，此时可以通过 get_last_error 来获取详细错误信息，如果正常，则响应信息存入响应缓冲区。调用者需要释放该过程中申请的内存。

```
/**
* 发送 req_buf, 然后将回馈信息回填如 res_buf, 具体接口描述参看 ut_ios.h
*/
intraw_send_recv(constchar*url, void*req_buf, unsignedintreq_len, \
                        void**res_buf, unsignedint*res_len){
    CURL *curl;
    CURLcode res;

    intsocket;
    intoffset;

    size_t ret_len;
    size_t new_size;

    char*temp = (char*)malloc(sizeof(char)*1024);
    char*term = (char*)malloc(sizeof(char)*1);

    curl = curl_easy_init();

    if(curl == NULL){
```

```
        on_error("raw_send_recv : 初始化 CURL 失败 \n");
        return-1;
    }

    curl_easy_setopt(curl, CURLOPT_URL, url);
    curl_easy_setopt(curl, CURLOPT_CONNECT_ONLY, 1L);

    res = curl_easy_perform(curl);

    if(res != CURLE_OK){
        on_error("raw_send_recv : 启动 CURL 失败，错误信息：%s\n", \
            curl_easy_strerror(res));
        return-1;
    }

    res = curl_easy_getinfo(curl, CURLINFO_LASTSOCKET, &socket);

    if(res != CURLE_OK){
        on_error("raw_send_recv : 获取套接字失败，错误信息：%s\n", \
            curl_easy_strerror(res));
        return-1;
    }

    /* wait for the socket to become ready for sending */
    if(!wait_on_socket(socket, FD_SEND, ONE_MINUTE)){
        on_error("raw_send_recv : 套接字连接超时 ( 发送时 )\n");
        return-1;
    }

    res = curl_easy_send(curl, req_buf, req_len, &ret_len);

    if(res != CURLE_OK){
        on_error("raw_send_recv : 传输数据失败，错误信息：%s\n", \
            curl_easy_strerror(res));
        return-1;
    }

    if(req_len != ret_len){
        on_error("raw_send_recv : 数据未发送完成，剩余：%d\n", \
            (req_len - ret_len));
        return-1;
    }

    new_size = 0;

    for(;;){
        if(!wait_on_socket(socket, FD_RECV, ONE_MINUTE)){
            on_error("raw_send_recv : 套接字连接超时 ( 接受时 )\n");
            return-1;
        }

        memset(temp, '\0', 1024);
        res = curl_easy_recv(curl, temp, 1024, &ret_len);

        if(res != CURLE_OK){
            break;
        }

        new_size += ret_len;

        if(new_size > *res_len){
            *res_buf = realloc(*res_buf, new_size);
        }

        if(*res_buf == NULL){
            on_error("raw_send_recv : 为写扩展缓存出错 : %s\n", \
                new_size, strerror(errno));
            break;
        }
```

```
            offset = new_size - ret_len;
            memcpy((char*)*res_buf+offset, temp, ret_len);

            *res_len = new_size;
        }
        //terminal character
        memset(term, '\0', 1);
        memcpy((char*)*res_buf+*res_len, term, 1);
        curl_easy_cleanup(curl);

        ut_status = UT_OK;
        return0;
}
```

在这个接口的实现中，先初始化 curl，然后尝试连接，并通过 curl_easy_getinfo 函数获取此时服务端的 socket 句柄，然后就可以读写此句柄了，wait_on_socket 通过调用多路侦听调用 select, 当 socket 上有数据到达则立即返回，否则等待 timeout 时长后返回。此处的超时设置为 1 分钟。发送请求之后，进入一个无限 for 循环，与向 socket 写数据不同的是，在读的时候，还无法得知需要读如多长的数据，这样就需要动态的扩展内存 (通过 realloc 系统调用)。

限于篇幅，其他的 API 与 raw_send_recv 的调用过程比较类似，就不一一列举了，raw_send_recv 比较有代表性，而且较之基于具体协议的实现更加灵活，其他的 API 的实现可以参考附件中的代码。

其他工具函数

一个完整的函数库必须提供完善的错误处理，并尝试从不严重的错误中恢复。至少，一个函数库需要提供详细的错误信息，方便库的使用者进行调试。

错误处理

通用网络库提供一个错误处理函数，on_error，在网络库内部使用这个处理函数来完成错误的报告，如果在调试模式下，on_error 会向标准输出打印一行错误信息，指出详细的错误信息，如果在非调试模式下 (以库的方式提供给别的程序员使用)，则将错误消息写入一个静态的缓冲区，别的程序员通过调用 get_last_error 来获取详细的错误描述：

```
/*
* 打印错误信息到错误缓冲区 (ut_error_message)，私有函数
*/
staticinton_error(constchar*format, ...){
        intbytes_written;
        va_list arg_ptr;
```

```
        va_start(arg_ptr, format);
        ut_status = UT_ERROR;

#ifdefDEBUG_MODE
        bytes_written = vfprintf(stderr, format, arg_ptr);
#else
bytes_written = \
vsnprintf(ut_error_message, UT_ERROR_LEN, format, arg_ptr);
#endif

        va_end(arg_ptr);

        returnbytes_written;
}
```

函数 on_error 被 static 修饰，则在文件之外无法访问此函数，同样，缓冲区及库的当前状态也定义为 static:

```
staticcharut_error_message[UT_ERROR_LEN];
staticintut_status = UT_OK;
```

如果网络库的当前状态 ut_status 的值为 UT_ERROR, 则 get_last_error 返回错误缓冲区中的内容，否则返回 NULL。

```
/**
* 获取最近一次发生的错误
*/
constchar*get_last_error(void){
        if(ut_status == UT_OK){
                returnNULL;
        }
        returnut_error_message;
}
```

应该注意的是，从 get_last_error 的实现可以看出，ut_error_message 缓冲区会被重写，如果错误发生了，不调用 get_last_error 而做了一些可能出错的其他操作，再一次调用 get_last_error 时，得到的错误描述会为最后一次的错误信息。因此，在调用了一个可能出错的 API 之后，需要紧接着调用 get_last_error 以获得详细的错误信息。

回页首

网络库的使用

测试用例

通用网络库的开发部分已经完成了，下面我们对其进行一些测试，熟悉一下 API 的用法。第一个例子是上传文件至服务器的例子，程序向服务器指定端口发送字节流，并实时更新传输进度。服务端在建立与客户端的连接之后，将从 socket 中读入的字节写入文件，并保存。第二个例子测试自定义应用协议的网络程序，客户端发送请求并等待服务端处理，服务端处理后返回响应信息。

测试用例 1

上传文件至服务器，首先定义一个用于更新上传进度的函数 update:

```
static void update(double current, double total){
	fprintf(stderr, "%% %.2f processed\n", (current/total)*100);
}
```

函数 raw_send_file 每成功的写入一个数据块，就调用一次 update, 报告上传的进度：

```
int upload_file_test(){
	int res = 0;
	res = raw_send_file("192.168.21.2:9527", \
			"alice.rmvb", \
			 get_file_len("alice.rmvb"), \
			 update);

	if(res < 0){
		fprintf(stderr, "发送文件错误：%s\n", get_last_error());
	}else{
		fprintf(stderr, "发送文件成功 \n");
	}
}
```

如果发生错误，使用 get_last_error 来查看详细错误，否则可以看到上传的进度，这里的 update 只是一个简单的打印，如果在实际的应用中，可以与一个进度条绑定，更好的体现上传的的进度。

图 1. 发送失败

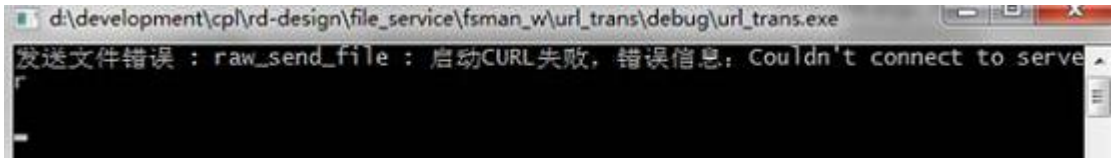
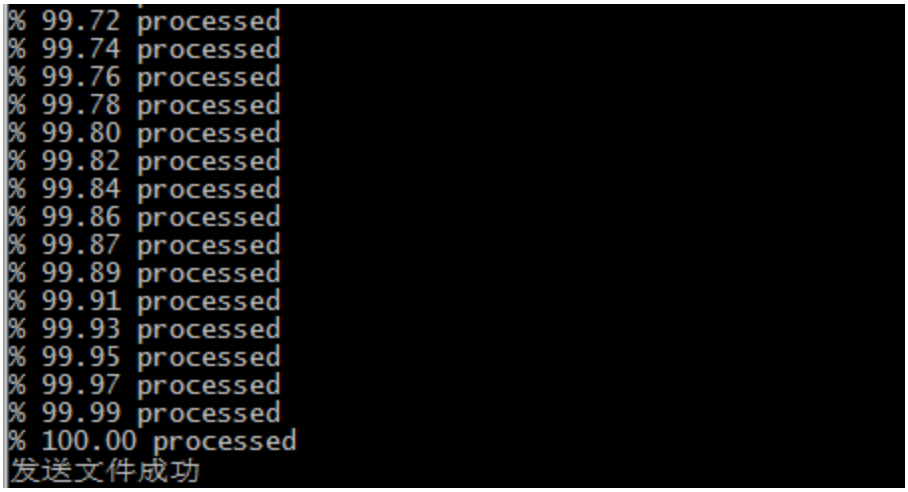


图 2. 发送成功



测试用例 2

首先，将文件 request.xml 加载到缓冲区 req_buf 中，如果 req_buf 的长度不足，加载函数会自己重新分配内存，调用者负责释放该部分内存。服务器在 8384 端口侦听，在这个例子中，服务端会解析传入的字节流，并按照 xml 方式解析，完成后组织 xml 响应信息返回给客户端，最终的结果存入 res_buf，长度为 res_len。

```
int raw_rw_test(){
	int res = 0, fd = 0;
	char *req_buf;
	char *res_buf;

	int req_len = 1024;
	int res_len = 1024;

	req_buf = (char *)malloc(sizeof(char) * req_len);
	res_buf = (char *)malloc(sizeof(char) * res_len);

	if(req_buf == NULL){
		fprintf(stderr, "为读缓冲区分配内存失败 \n");
		return -1;
	}

	fd = load_file_to_mem("request.xml", &req_buf, &req_len);

	if(fd < 0){
		fprintf(stderr, "读取文件至 buffer 失败 \n");
		free(req_buf);
		return -1;
	}

	res = raw_send_recv("10.111.43.145:8384", \
			req_buf, \
			req_len, \
			&res_buf, \
			&res_len);
	if(res < 0){
		fprintf(stderr, "读写错误：%s\n", get_last_error());
	}else{
		fprintf(stderr, "读取长度 [%d]\n 读取内容：%s\n", res_len, res_buf);
	}

	return 0;
}
```

如果 raw_send_recv 发生错误，如网络不通，或者读写超时等，通过 get_last_error 来打印详细错误信息，如果一切正常，则打印响应信息的长度及内容。当然，这里只是测试数据的正确性，如果是实际的应用中，可以将响应信息内容进行抽取，排列，最终展现在客户端。

回页首

结束语

libcurl 库有更多的有意思的主题可供学习，研究，比如 https 部分，基于 HTTP 表单的提交，将 libcurl 作为一个 HTTP 的客户端来测试基于 WEB 的服务等等。本文中的应用知识 libcurl 的一小部分。

同样，命令行实用程序 curl 也值得花时间来学习，curl 事实上是一个小巧而方便的网络客户端程序，如果不想使用 libcurl 从零开始的话，花时间阅读下 curl 的手册会有很大的帮助，很可能你的需求 curl 已经完全可以解决。

回页首

下载

描述	名字	大小
样例代码	url_trans_art.zip	85KB

http://www.ibm.com/developerworks/apps/download/index.jsp?contentid=607395&filename=url_trans_art.zip&method=http&locale=zh_CN

参考资料

学习

- http://curl.haxx.se/cURL 及 libcurl 官方网站。
- http://curl.haxx.se/libcurl/c/example.htmllibcurl 发布的包中的实例代码的作用解释。
- 访问 developerWorks Open source 专区获得丰富的 how-to 信息、工具和项目更新以及最受欢迎的文章和教程，帮助您用开放源码技术进行开发，并将它们与 IBM 产品结合使用。
- 随时关注 developerWorks 技术活动和网络广播。