Bug修改记录 11.5

1.关于最后剩下的一个get_byte方法

- 关于`get`请求

  原来的`get`请求写的太烂了，当然我没办法把整个方法重构，我只能在接受数据这里做了一个算法，之前如果响应的数据超出缓冲区是没有办法完整到数据的，我也不知道为什么这两个学长没有想到这一点，还是测试提供的样例都是简单的`json`数据返回，加上响应头必然不可能超出边界，但是他的测试样例里面还请求了`www.baidu.com`，这个响应体有四十多万字节，那必然不可能请求完毕，所以基于这一点，我做了修改，当然也是为了`get`请求图片资源而服务

```cpp
// 定义一些数据，在下面需要用到
size_t data_start = 0;                // 响应体的开头下标
size_t data_end = 0;                  // 响应体的结束下标
size_t content_length = 0;            // 响应体的长度
bool is_Location_in_header = false;   // url是否进行了重定向
int count = 0;                        // 计数，为了提高效率

// 我换了一个算法，能读取到全部的数据，就是缓冲区满了需要多次读取的时候
while (1) {
    bzero(readBuffer, BUFSIZ);
    int len = recv(sockfd, readBuffer, BUFSIZ - 1, 0);
    if (-1 == len) {
        perror("recv");
        exit(-1);
    }
    if (len > 0) {
        std_readMessage.insert(std_readMessage.end(), readBuffer, readBuffer + len);

        // 检查收到的数据是否完整包含了响应头，这一步骤只进行一次，目的是为了得到响应的数据
        size_t headers_end = std_readMessage.find("\r\n\r\n");
        if (std::string::npos != headers_end && 0 == count++) {
            // 注意substr第一个参数是子串的开始位置，第二个参数是字串的长度
            std::string headers = std_readMessage.substr(0, headers_end);

            // 查看是否有重定向存在
            if (!is_Location_in_header) {
                size_t location_pos = headers.find("Location:");
                if (std::string::npos != location_pos)
                    is_Location_in_header = true;
            }

            size_t content_length_pos = headers.find("Content-Length:");
            if (std::string::npos != content_length_pos) {
                size_t content_length_end = headers.find("\r\n", content_length_pos);

                content_length = atoi(headers.substr(content_length_pos + strlen("Content-Length: "), content_length_end - content_length_pos - strlen("Cor

                // 拿到所需要的数据
                data_start = headers_end + 4;
                data_end = data_start + content_length;
            }
        }

        // 得到了所有的数据，循环结束
        if (std_readMessage.size() - data_start >= content_length)
            break;
    } else if (0 == len)
        break;  // 服务端关闭了 ...
}
```

最开始想的是判断接受的`len`和缓冲区大小相等就代表数据没接收完毕，但是我天真了，服务器发回来的数据偶尔会丢失一两个字节，这很正常，所以这样导致数据还没接收完循环就结束了，所以我们需要通过响应头的`Content-Length`字段的长度来确定响应体的大小，并且通过相应判断才能或者全部的数据，上面的代码思路就是这样

这样以后我们跑一下`get`请求请求`www.baidu.com`，我录了一段视频可以直观的感受

```cpp
}
std::cout << std_readMessage << std::endl;
```

当我不做处理，直接打印接收到的数据的时候，发现四十多万个字节，接受的时间也就两三秒左右吧，这个时间很正常；说明效率低是后面的代码导致的

这里，我们仔细看各个方法，首先参数需要`LString`，这个是没有办法的，其次在内部多次调用`LString`的拷贝构造，但是这些方法的功能的确没有问题，那么效率低只可能就是`LString`的拷贝构造这里了

```
// 这后面太屎山了，LString的拷贝，太慢了，我请求个baidu.com四十多万个字节，拷贝一次慢的要死，而且设计的
LString readMessage(std_readMessage);

// 设置数据，数据是不包含响应头的
reply->setData(reply, LString(std_readMessage.substr(data_start, data_end - data_start)));
setRecv(this, readMessage);
readStatusCode(readMessage, reply);
if (readMessage.contains("HTTP/1.1")) {
    reply->readHeader(readMessage);
}

close(sockfd);
```

我们看看`LString`的构造：

```
LString::LString(const char *s)
{
    m_pData = new struct LStringDataStruct;
    m_pData->m_size = unicodeCharCount(s);
    m_pData->m_pLChar = toLCharArray(s);
}
```

调用了两个方法，随便点一个

```
int LString::unicodeCharCount(const char* s)
{
    if (!s)
    {
        return 0;
    }
    int sizeLCharStr = 0;
    const char* pChar = s;
    for (int i = 0; i < strlen(s);)
    {
        int tmp = LChar::getUtf8CharSize(pChar);
        if (tmp == -1)
        {
            std::cout<<"仅支持小于等于三字节的UTF8字符"<<std::endl;
            return 0;
        }
        pChar += tmp;
        i += tmp;
        sizeLCharStr += 1;
    }
    return sizeLCharStr;
}
```

```cpp
LChar* LString::toLCharArray(const char* s)
{
    int sizeLCharStr = m_pData->m_size;
    const char* pChar = s;
    LChar* tempLChar = new LChar[sizeLCharStr + 1];

    for (int i = 0; i < sizeLCharStr; i++)
    {
        unsigned short tempShort = 0;

        LChar::utf8ToUnicodeOne(pChar, &tempShort);
        pChar += LChar::getUtf8CharSize(pChar);

        *(tempLChar + i) = LChar(tempShort);
    }
    *(tempLChar + sizeLCharStr) = LChar('\0');
    return tempLChar;
}
```

好好好，初步看来是一个一个字节拷贝是吧，怪不得这么慢，四十多万个字节...

当然，如果只是发一些小的请求，那效率问题是看不出来的，但是这个效率慢就不是我这边能控制得了的了。

当然，抛开效率不谈，任务是完成了的。

- `get`请求图片资源

  这个问题和上面反应的是一样的，多字节的问题

  这张图里面把抛异常改成了输出，但是这样的话`std::string`显然没有办法正确的转化为`LString`了，因为`return 0`，后续的操作都会错误进行，显然没有办法请求到图片数据

```
int LString::unicodeCharCount(const char* s)
{
    if (!s)
    {
        return 0;
    }
    int sizeLCharStr = 0;
    const char* pChar = s;
    for (int i = 0; i < strlen(s);)
    {
        int tmp = LChar::getUtf8CharSize(pChar);
        if (tmp == -1)
        {
            std::cout<<"仅支持小于等于三字节的UTF8字符"<<std::endl;
            return 0;
        }
        pChar += tmp;
        i += tmp;
        sizeLCharStr += 1;
    }
    return sizeLCharStr;
}
```

当然，图片资源想要获得也很简单，由于直接解析的话响应体的数据段是乱码，但是我们把他塞到一个文件当中，并以图片的格式结尾命名，图片就请求到了，我写了一个测试程序如下，当然只能请求`http`协议的图片，`https`协议会返回`302`永久迁移，当然`https`不在最开始的计划内...

```
#include <arpa/inet.h>
#include <netdb.h>
#include <unistd.h>

#include <cstring>
#include <fstream>
#include <iostream>
#include <string>
#include <vector>

int main() {
    std::string path = "/img/PCtm_d9c8750bed0b3c7d089fa7d55720d6cf.png";
    std::string host = "www.baidu.com";

    std::string send_message = "GET " + path + " HTTP/1.1\r\n";
    send_message += "Host: " + host + "\r\n";
    send_message += "Connection: keep-alive\r\n";
    send_message += "User-Agent: Mozilla/5.0\r\n";
    send_message += "Accept-Language: zh-CN,zh;q=0.9,en;q=0.8,en-GB;q=0.7,en-
US;q=0.6\r\n";
    send_message += "\r\n";

    int connect_fd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
```

```
    if (-1 == connect_fd) {
        perror("socket");
        return -1;
    }

    struct hostent *p_hostent = gethostbyname(host.c_str());

    struct sockaddr_in server_addr;
    server_addr.sin_family = AF_INET;
    memcpy(&(server_addr.sin_addr), p_hostent->h_addr_list[0],
sizeof(server_addr.sin_addr));
    server_addr.sin_port = htons(80);

    int ret = connect(connect_fd, (struct sockaddr *)&server_addr,
sizeof(server_addr));
    if (-1 == ret) {
        perror("connect");
        return -1;
    }

    send(connect_fd, send_message.c_str(), send_message.size(), 0);

    char readBuffer[BUFSIZ] = {0};
    std::string std_readMessage;

    // 定义一些数据，在下面需要用到
    size_t data_start = 0;                // 响应体的开头下标
    size_t data_end = 0;                  // 响应体的结束下标
    size_t content_length = 0;            // 响应体的长度
    bool is_Location_in_header = false;   // url是否进行了重定向
    int count = 0;                        // 计数，为了提高效率

    // 我换了一个算法，能读取到全部的数据，就是缓冲区满了需要多次读取的时候
    while (1) {
        bzero(readBuffer, BUFSIZ);
        int len = recv(connect_fd, readBuffer, BUFSIZ - 1, 0);
        if (-1 == len) {
            perror("recv");
            exit(-1);
        }
        if (len > 0) {
            std_readMessage.insert(std_readMessage.end(), readBuffer, readBuffer
+ len);

            // 检查收到的数据是否完整包含了响应头，这一步骤只进行一次，目的是为了得到响应的数
据
            size_t headers_end = std_readMessage.find("\r\n\r\n");
            if (std::string::npos != headers_end && 0 == count++) {
                // 注意substr第一个参数是子串的开始位置，第二个参数是字串的长度
                std::string headers = std_readMessage.substr(0, headers_end);

                // 查看是否有重定向存在
```

```
                if (!is_Location_in_header) {
                    size_t location_pos = headers.find("Location:");
                    if (std::string::npos != location_pos)
                        is_Location_in_header = true;
                }

                size_t content_length_pos = headers.find("Content-Length:");
                if (std::string::npos != content_length_pos) {
                    size_t content_length_end = headers.find("\r\n",
content_length_pos);

                    content_length = atoi(headers.substr(content_length_pos +
strlen("Content-Length: "), content_length_end - content_length_pos -
strlen("Content-Length: ")).c_str());

                    // 拿到所需要的数据
                    data_start = headers_end + 4;
                    data_end = data_start + content_length;
                }
            }

            // 得到了所有的数据，循环结束
            if (std_readMessage.size() - data_start >= content_length)
                break;
        } else if (0 == len)
            break;  // 服务端关闭了...
    }

    // std::cout << std_readMessage << std::endl;
    // std::cout << "Received data size: " << std_readMessage.size() << " bytes"
<< std::endl;

    // 将数据部分从std_readMessage中提取出来
    std::string data_str = std_readMessage.substr(data_start, data_end -
data_start);

    // 将数据部分写入文件
    FILE *file = fopen("image.jpg", "w+");
    if (nullptr == file) {
        perror("fopen");
        return -1;
    }

    fwrite(data_str.c_str(), 1, data_str.size(), file);

    fclose(file);

    std::cout << "Image saved to image.jpg" << std::endl;

    close(connect_fd);

    return 0;
```

```
}
```

可以自己试一下，能正确请求到图片资源

- 关于其他的请求

  其他的请求肯定也存在`get`请求读取数据这里的问题，但是其他请求不同于`get`，他们的响应报文大多数情况都不会太长，所以现在的勉强够用，测试那边也没有出问题，这个等后续再说吧...