浙江大学

# 本科实验报告

| | |
|---|---|
| 课程名称： | 计算机网络基础 |
| 实验名称： | Lab2 Webget & ByteStream |
| 姓　　名： | 蔡佳伟 |
| 学　　院： | 计算机学院 |
| 系： | 计算机科学与技术学院（系） |
| 专　　业： | 软件工程 |
| 学　　号： | 3220104519 |
| 指导教师： | 高艺 |

2024 年 10 月 19 日

# 浙江大学实验报告

实验名称：Lab2 Webget　　实验类型：设计实验

同组学生：　　实验地点：曹西-304

## 一、 实验目的：

- 学习掌握 Linux 虚拟机的用法
- 学习掌握网页的抓取方法
- 学习掌握 ByteStream 的相关知识
- 学习掌握 C++的新特性

## 二、 实验内容

- 安装配置 Linux 虚拟机并在其上完成本次实验。
- 编写小程序 webget，通过网络获取 web 页面，类似于 wget。
- 实现字节流 ByteStream：
  - 字节流可以从写入端写入，并以相同的顺序，从读取端读取；
  - 字节流是有限的，写者可以终止写入。而读者可以在读取到字节流末尾时，产生 EOF 标志，不再读取；
  - 支持流量控制，以控制内存的使用；
  - 写入的字节流可能会很长，必须考虑到字节流大于缓冲区大小的情况。

## 三、 主要仪器设备

- 联网的 PC 机
- Linux 虚拟机

## 四、 操作方法与实验步骤

- 在你的电脑上安装配置 GNU/Linux:安装 CS144 VirtualBox 虚拟机映像，可以从学校内网下载(http://10.214.131.122:8080/cs_network/cs144_vm.ova)。
- 抓取网页：在正式进行编码工作之前，你需要对本实验第一个任务，即抓取一个网页，有更深刻的理解。
  - 在浏览器中，访问 http://cs144.keithw.org/hello 并观察结果。
  - 在你的虚拟中运行 **telnet cs144.keithw.org http** 命令，它告诉 telnet 程序在你的计算机与另一台计算机（名为 cs144.keithw.org）之间打开一个可靠的字节流，并在这台计算机运行一个特定的服务：“http”服务。
  - 输入 **GET /hello HTTP/1.1** 以及回车键，这告诉服务器 URL 的路径部分。
  - 输入 **Host: cs144.keithw.org** 以及回车键，这告诉服务器 URL 的主机部分。
  - 输入 **Connection: close** 以及回车键，这告诉服务器你已经完成了 HTTP 请求。

- 将返回结果与浏览器的返回结果进行比较。

- 准备工作：从 github 上抓取初始代码文件并完成环境搭建。

- 在你的虚拟机上，运行 **git clone -b lab7-startercode https://github.com/zhanghl-learner/sponge** 命令来获取初始代码文件。

- 运行 **cd sponge** 命令进入 sponge 目录。

- 运行 **mkdir build** 命令构建 build 目录来编译实验代码。

- 运行 **cd build** 命令进入 build 目录

- 运行 **cmake ..**命令建立搭建系统

- 运行 **make** 命令编译源代码，需要注意每次你对项目进行了修改都需要运行 **make** 命令。

- 实验代码的编写将以现代 C++风格完成，使用最新的 2011 特性尽可能安全地编程。请参考 C++指南（http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines）。

- 阅读 sponge 文档：sponge 封装了操作系统函数，请务必在编写实验代码前仔细阅读相关的基础代码文件。

- 阅读入门代码文档（https://cs144.github.io/doc/lab0）。

- 阅读 FileDescriptor, Socket, TCPSocket 以及 Address 这些类的文档。

- 阅读描述了这些类的头文件，请查阅 **libsponge/util** 目录：**file_descriptor.hh**, **socket.hh** 以及 **address.hh**。

- 实现 webget：完成 webget.cc 程序代码的编写以实现抓取网页的功能。

- 从 build 目录下，在文本编辑器或 IDE 下打开**../apps/webget.cc**。

- 在 **get_URL** 函数中完成实现，实现代码使用 **HTTP**（Web）请求的格式。使用 **TCPSocket** 类以及 **Address** 类。

- 运行 **make** 命令编译程序。

- 运行**./apps/webget cs144.keithw.org /hello** 命令进行程序的测试。

- 通过上面的测试后运行 **make check_webget** 命令进行自动测试。

- 完成编码工作。

注意点：

- 在 HTTP 中，每行必须以“\r\n”结尾。

- Connection:close 这句代码必须包含在客户端的请求中。

- 确保从服务器读取和打印所有的输出，直到套接字到达“EOF”，即文件的末尾。

- 可靠字节流：实现该实验的第二个任务，完成 ByteStream 的代码编写工作，ByteStream 的功能在上面已经做了一定的介绍。

- 打开 **libsponge/byte_stream.hh** 以及 **libsponge/byte_stream.cc** 文件，并完成接口内的方法的实现。

- 完成 writer 的方法，如下：

```
// Write a string of bytes into the stream. Write as many
// as will fit, and return the number of bytes written.
size_t write(const std::string &data);

// Returns the number of additional bytes that the stream has space for
size_t remaining_capacity() const;

// Signal that the byte stream has reached its ending
void end_input();

// Indicate that the stream suffered an error
void set_error();
```

- 完成 reader 的方法，如下：

```
// Peek at next "len" bytes of the stream
std::string peek_output(const size_t len) const;

// Remove ``len'' bytes from the buffer
void pop_output(const size_t len);

// Read (i.e., copy and then pop) the next "len" bytes of the stream
std::string read(const size_t len);

bool input_ended() const;   // `true` if the stream input has ended
bool eof() const;           // `true` if the output has reached the ending
bool error() const;         // `true` if the stream has suffered an error
size_t buffer_size() const; // the maximum amount that can currently be peeked/read
bool buffer_empty() const;  // `true` if the buffer is empty

size_t bytes_written() const;  // Total number of bytes written
size_t bytes_read() const;     // Total number of bytes popped
```

- 完成 byte_stream 中代码的编写后，运行 **make check_lab0** 命令进行自动测试。

- 温馨提示：当你在开发代码的时候，可能会遇到无法解决的问题，下面给出解决的办法。

- 运行 **cmake .. -DCMAKE_BUILD_TYPE=RelASan** 命令配置 build 目录，使编译器能够检测内存错误和未定义的行为并给你很好的诊断。

- 你还可以使用 valgrind 工具。

- 你也可以运行 **cmake .. -DCMAKE_BUILD_TYPE=Debug** 命令配置并使用 GNU 调试器（**gdb**）。

- 你可以运行 **make clean** 和 **cmake .. -DCMAKE_BUILD_TYPE=Release** 命令重置构建系统。

- 如果你不知道如何修复遇到的问你题，你可以运行 **rm -rf build** 命令删除 build 目录，创建一个新的 build 目录并重新运行 **cmake..**命令。

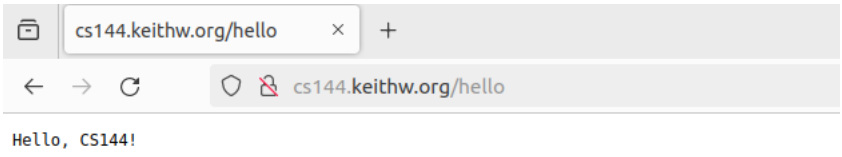
## 五、 实验数据记录和处理

- 第二步中抓取网页的运行结果

```
cjwjiwang@cjwjiwang-virtual-machine:~/Desktop$ telnet cs144.keithw.org http
Trying 104.196.238.229...
Connected to cs144.keithw.org.
Escape character is '^]'.
^]
telnet>
GET /hello HTTP/1.1
Host: cs144.keithw.org
Connection: close

HTTP/1.1 200 OK
Date: Fri, 18 Oct 2024 07:18:48 GMT
Server: Apache
Last-Modified: Thu, 13 Dec 2018 15:45:29 GMT
ETag: "e-57ce93446cb64"
Accept-Ranges: bytes
Content-Length: 14
Connection: close
Content-Type: text/plain

Hello, CS144!
Connection closed by foreign host.
```




- 编写的 webget 关键代码，即 get_URL 中的代码

```cpp
void get_URL(const string &host, const string &path) {
    // Your code here.

    // You will need to connect to the "http" service on
    // the computer whose name is in the "host" string,
    // then request the URL path given in the "path" string.

    // Then you'll need to print out everything the server sends back,
    // (not just one call to read() -- everything) until you reach
    // the "eof" (end of file).
    cout<<"call GET_URL CJW!\n";
    TCPSocket tcp_socket;
    tcp_socket.connect(Address(host,"http"));
    string request="GET "+path+" HTTP/1.1\r\nHost: "+host+"\r\nConnection: close\r\n\r\n";
    tcp_socket.write(request);

    while(!tcp_socket.eof()){
        cout<<tcp_socket.read();
    }
    tcp_socket.shutdown(SHUT_RDWR);

    // cerr << "Function called: get_URL(" << host << ", " << path << ").\n";
    // cerr << "Warning: get_URL() has not been implemented yet.\n";
}
```

- 使用 webget 抓取网页运行结果

```
cjwjiwang@cjwjiwang-virtual-machine:~/lab2/sponge/build$ ./apps/webget cs144.kei
thw.org /hello
call GET_URL CJW!
HTTP/1.1 200 OK
Date: Sat, 19 Oct 2024 14:53:55 GMT
Server: Apache
Last-Modified: Thu, 13 Dec 2018 15:45:29 GMT
ETag: "e-57ce93446cb64"
Accept-Ranges: bytes
Content-Length: 14
Connection: close
Content-Type: text/plain

Hello, CS144!
```

- 运行 make check_webget 的测试结果展示

```
Hello, CS144!
cjwjiwang@cjwjiwang-virtual-machine:~/lab2/sponge/build$ make check_webget
[100%] Testing webget...
Test project /home/cjwjiwang/lab2/sponge/build
    Start 31: t_webget
1/1 Test #31: t_webget ........................   Passed    1.09 sec

100% tests passed, 0 tests failed out of 1

Total Test time (real) =   1.09 sec
[100%] Built target check_webget
```

- 实现 ByteStream 关键代码截图（描述总体，省略细节部分）

```
class ByteStream {
  private:
    // Your code here -- add private members as necessary.

    // Hint: This doesn't need to be a sophisticated data structure at
    // all, but if any of your tests are taking longer than a second,
    // that's a sign that you probably want to keep exploring
    // different approaches.
    std::deque<char> _buffer; // keep bytes buffer
    size_t _capacity; // max bytes stream
    size_t _bytes_written = 0;
    size_t _bytes_read = 0;
    bool _input_ended = false;
    bool _error{};  //!< Flag indicating that the stream suffered an error.
```

```cpp
ByteStream::ByteStream(const size_t capacity)
    : _buffer(), _capacity(capacity), _bytes_written(0), _bytes_read(0), _input_ended(false), _error(false){}


size_t ByteStream::write(const string &data) {
    size_t space_left = remaining_capacity();
    size_t bytes_to_write = std::min(data.size(), space_left);

    for(size_t i = 0; i < bytes_to_write; i++){
        _buffer.push_back(data[i]);
    }

    _bytes_written += bytes_to_write;
    return bytes_to_write;
}

//! \param[in] len bytes will be copied from the output side of the buffer
string ByteStream::peek_output(const size_t len) const {
    size_t bytes_to_peek = std::min(len,_buffer.size());
    return std::string(_buffer.begin(),_buffer.begin() + bytes_to_peek);
}

//! \param[in] len bytes will be removed from the output side of the buffer
void ByteStream::pop_output(const size_t len) {
    size_t bytes_to_pop = std::min(len,_buffer.size());
    for(size_t i = 0; i< bytes_to_pop; i++){
        _buffer.pop_front();
    }
    _bytes_read += bytes_to_pop;
    }
```

```cpp
  //! Read (i.e., copy and then pop) the next "len" bytes of the stream
  //! \param[in] len bytes will be popped and returned
  //! \returns a string
  std::string ByteStream::read(const size_t len) {
      std::string result = peek_output(len);
      pop_output(len);
      return result;
  }

  void ByteStream::end_input() {_input_ended=true;}

  bool ByteStream::input_ended() const { return _input_ended; }

  size_t ByteStream::buffer_size() const { return _buffer.size(); }

  bool ByteStream::buffer_empty() const { return _buffer.empty(); }

  bool ByteStream::eof() const { return _input_ended && _buffer.empty(); }

  size_t ByteStream::bytes_written() const { return _bytes_written; }

  size_t ByteStream::bytes_read() const { return _bytes_read; }

  size_t ByteStream::remaining_capacity() const { return _capacity- _buffer.size(); }
```

- 运行 make check_lab0 测试结果。

```
cjwjiwang@cjwjiwang-virtual-machine:~/lab2/sponge/build$ make check_lab0
[100%] Testing Lab 0...
Test project /home/cjwjiwang/lab2/sponge/build
    Start 26: t_byte_stream_construction
1/9 Test #26: t_byte_stream_construction .......   Passed    0.00 sec
    Start 27: t_byte_stream_one_write
2/9 Test #27: t_byte_stream_one_write ..........   Passed    0.00 sec
    Start 28: t_byte_stream_two_writes
3/9 Test #28: t_byte_stream_two_writes .........   Passed    0.00 sec
    Start 29: t_byte_stream_capacity
4/9 Test #29: t_byte_stream_capacity ...........   Passed    0.31 sec
    Start 30: t_byte_stream_many_writes
5/9 Test #30: t_byte_stream_many_writes ........   Passed    0.01 sec
    Start 31: t_webget
6/9 Test #31: t_webget .........................   Passed    1.04 sec
    Start 53: t_address_dt
7/9 Test #53: t_address_dt .....................   Passed    0.01 sec
    Start 54: t_parser_dt
8/9 Test #54: t_parser_dt ......................   Passed    0.00 sec
    Start 55: t_socket_dt
9/9 Test #55: t_socket_dt ......................   Passed    0.02 sec
```

## 六、 实验数据记录和处理

- 完成 webget 程序编写后的测试结果和 Fetch a Web page 步骤的运行结果一致吗？如果不一致的话你认为问题出在哪里？请描述一下所写的 webget 程序抓取网页的流程。

  一致的，Webget 程序抓取网页的流程大致如下：

  1. 准备环境：在抓取网页之前，首先要准备好程序的运行环境。一般需要安装支持 HTTP 请求的库，如 Python 中的`requests`库或者`urllib`库。此阶段还需要设置必要的请求头信息（如 User-Agent），以模拟真实用户访问网页。

  2. 发起 HTTP 请求：

  - Webget 程序首先通过 HTTP 协议向目标网站发送请求，通常是`GET`请求。这个请求可以是带有特定参数的 URL，以获取特定的网页内容。请求可以包含头部信息（headers），如浏览器类型、语言、以及是否允许重定向等信息。

  - 一些网站对频繁抓取有防范措施，可能会阻止机器访问或要求通过验证码。因此，有时需要进行反爬虫的应对，例如设置合理的抓取间隔、使用代理 IP 等。

3. 接收服务器响应：

- 服务器收到请求后，会返回相应的响应内容。这通常包括 HTTP 状态码（如 200 表示成功，404 表示页面不存在）以及网页的 HTML 内容或其他格式的数据（如 JSON、XML 等）。

- 如果状态码显示请求失败（如 403 禁止访问、500 服务器错误等），Webget 程序需要处理错误并作出相应的调整，例如重试请求或记录日志。

4. 解析网页内容：

- Webget 程序抓取到网页的原始数据后，通常需要对其进行解析。对于 HTML 网页，可以使用解析库（如`BeautifulSoup`或`lxml`）来提取特定的元素或内容。

- 解析时可以根据 HTML 的标签、类名、ID 等选择器，提取所需的数据，例如标题、链接、文章内容等。

- 如果返回的数据是 JSON 格式，则可以直接解析为字典或对象，方便处理。

5. 数据处理和存储：

- 提取到需要的内容后，Webget 程序通常会对数据进行处理，比如清理无用的字符、格式化数据等。

- 处理好的数据可以保存到本地文件、数据库，或者以其他形式存储供后续使用。

6. 重复执行和防爬策略应对：

- 如果需要抓取多个网页或者进行定时抓取，Webget 程序通常会设置循环或者调度任务。每次请求之间应当设置合理的时间间隔，避免触发网站的反爬虫机制。

- 对于有些反爬虫机制较强的网站，可以通过模拟用户行为（如加入随机延时、动态加载页面的模拟、处理 Cookies 等）来避免被封禁。

7. 日志与异常处理：

- 抓取过程中可能会出现各种异常情况，如网络超时、服务器错误等。因此 Webget 程序需要对这些异常进行日志记录，并提供重试机制或错误处理逻辑，以提高程序的健壮性。

- 请描述 ByteStream 是如何实现流控制的？

ByteStream 实现流控制的方式主要依赖于其对缓冲区容量的管理、输入结束标志的控制以及读取和写入操作的同步。这种流控制的设计确保了字节流在受限的资源（如内存）条件下，能够有效地进行数据传输。以下是 ByteStream 实现流控制的关键点：

1. 缓冲区容量控制 ByteStream 在初始化时会指定一个固定的容量（_capacity），这决定了缓冲区最多能够存储的字节数量。流的写入操作会受这个容量限制，因此在容量耗 尽前，写入操作会被允许，而一旦缓冲区达到容量，写入就会受限。 剩余容量计算：使用 remaining_capacity() 方法来计算当前缓冲区剩余的可用空间，这确保了 每次写入时不会超过缓冲区的容量。 当剩余容量为 0 时，写入操作会被阻止或仅部分写入数据，保证缓冲区不会溢出。 例如，write 函数根据缓冲区剩余空间来决定可以写入多少字节： size_t space_left = remaining_capacity(); size_t bytes_to_write = std::min(data.size(), space_left);

2. 输入结束的管理 ByteStream 实现了一个 _input_ended 标志，用来指示流的输入是否已经 结束。当调用 end_input() 方法时，流会被标记为结束状态，此时不再允许写入更多数据。这是一种流的终止信号，表示后续的传输只允许读取，确保不会有更多的输入。 input_ended() 方法会返回输入是否结束，确保程序能根据输入状态来做进一步操作。这种输入终止机制有效地控制了数据流的生命周期，避免在写入结束后继续 发生写入。

3. 读取和写入的分离 通过 ByteStream 的 read 和 write 方法，流的读取和写入操作被分离开来。每次写入操作将数据添加到缓冲区中，而

读取操作从缓冲区取出数据。在这个过程中，ByteStream 实现了以下几个控制机制： buffer_size()：返回当前缓冲区中存储的字节数，读取操作根据这个数值 决定可以读取的最大字节数，确保不会读取超过缓冲区的内容。 buffer_empty()：判断缓冲区是否为空，防止读取操作在缓冲区为空的情况下发生。 eof()：通过 eof 标志判断流的读取是否已经到达末尾，确保流的读取行为 符合预期。

4. 读写操作的同步写入限制：写入时根据 remaining_capacity() 确保不会超过缓冲区容量。读取限制：读取时根据 buffer_size() 确保不会读取超过缓冲区当前存储的数据。 通过这种方式，ByteStream 的流控制确保了流的顺畅运行，防止写入过多导致溢出，也防止读取时获取到无效数据。

5. 错误管理 ByteStream 还实现了 _error 标志，当出现错误时可以调用 set_error() 设置错误状态，并在后续操作中通过 error() 方法检查是否有错误发生。这种错误管理机制为流的控制提供了进一步的可靠性保障。

- 当遇到超出 capacity 范围的数据流的时候，该如何进行处理？如果不限制流的长度的时候该如何处理？

  1.超出 capacity 范围的数据流处理
  ByteStream 的 capacity(容量)定义了缓冲区能够存储的最大字节数。为了确保流的稳定性和内存安全性，当数据流超出容量时，必须采取一些措施来处理这种情况。
  常见处理方式：
  （1）部分写入： 如果一次写入的数据超出剩余容量，可以选择只写入一部分数据，并返回实际写入的字节数。程序可以在下一次有空间时继续写入剩余数据。 这是 ByteStream 的一种常见处理方式，确保数据流不会超出缓冲区的限制。 size_t space_left = remaining_capacity(); size_t bytes_to_write = std::min(data.size(), space_left); 这种方式不会丢失数据，只是需要外部调用者进行多次写入操作来处理超出容量的数据流。
  （2）阻塞或等待： 在某些应用场景中，特别是需要实时数据传输时，当数据超出容量时，可以选择阻塞写入操作，直到缓冲区有足够空间。这种方式适用于生产者/消费者模型，生产者（写入方）会在缓冲区有空间时继续写入数据。 使用信号量或条件变量控制，当缓冲区有空间时才允许写入，防止数据溢出。
  （3）丢弃部分数据： 如果数据的完整性不是首要考虑，可以选择丢弃

超出容量部分的数据。这种方法在实时性要求高、但可以容忍数据丢失的场景（如实时视频流）可能适用。 if (data.size() > remaining_capacity()) { // 丢弃数据或警告 }
（4）触发错误或异常： 如果需要强制保证每次写入都成功，而无法写入的情况被视为错误，那么可以触发错误或抛出异常，提示缓冲区溢出。这样可以让程序调用方采取相应的处理措施。 if (data.size() > remaining_capacity()) { throw std::overflow_error("Buffer overflow: unable to write all data"); }

2. 不限制流长度的处理
在某些场景下，可能不希望限制流的长度（即不设置容量限制），允许流在 任意时间写入任意数量的数据。这种情况下，需要考虑如何处理无限制的流。
处理方式：
（1）动态扩展缓冲区：如果不限制流的长度，可以选择动态调整缓冲区的大小， 使其能够容纳不断增长的数据。可以通过容器如 std::vector 或 std::deque 来实现缓冲区的动态扩展。每次写入新数据时，如果缓冲区满了，则增加缓冲区的容量。// 使用 std::vector 或 std::deque 自动扩展 _buffer.push_back(data[i]); // 无需担心容量限制。这种方法会根据实际需要分配更多内存，确保流的长度不受限制。然而，它的代价是占用更多的内存，可能会影响系统的整体性能，特别是在大数据流的场景下。
（2）分页处理：对于非常大的数据流，可以考虑使用分页技术，将数据拆分为多个小块进行存储和处理。这种方式可以避免一次性占用过多的内存，而是根据需要逐步处理数据。分页处理通常需要额外的管理机制来协调不同页面的数据顺序和流动。
（3）流式处理：如果流的数据量非常大，不适合全部存储在内存中，可以考虑将数据以流式的方式进行处理。流式处理意味着数据在被写入后直接进行处理或传输，缓冲区只存储非常短时间内的数据。例如，可以在数据流中每次读取一小部分进行处理，然后立即清除已经处理的数据。这种方式通常用于视频流、音频流或者大规模数据传输场景。 磁盘或外部存储：当数据量非常大时，可以将数据临时写入磁盘或外部存储设备，而不是全部存储在内存中。通过这种方式，可以避免内存溢出，同时允许处理大数据流。例如，日志系统可以将超出的数据写入磁盘文件中，或者数据库系统可以将数据缓存到外部存储中以减少内存消耗。

## 七、 实验数据记录和处理

1.一开始的操作，发现 GET 等三个命令要很快地输入，于是复制到一起，然后一起输入，即完成

2.make 的时候发现缺少一个库，于是 sudo apt 安装，即可，引入数据结构时发现没有 deque，于是#include 即可