Lab1 实验报告

姓名	学号	邮箱	完成时间
陈嘉昀	211220137	jiayunchen@smail.nju.edu.cn	2023年10月1日

1 数据结构设计

1.1 ByteStream

用一个string来存储就足够了，一旦有数据往外pop，就把后面的数据往前面移动。

```
40  //! \param[in] len bytes will be removed from the output side of the buffer
41  void ByteStream::pop_output(const size_t len) {
42      assert(len <= _curr_size && "pop too much bytes");
43      for(size_t i = len; i < _curr_size; i += 1) {
44          buffer[i - len] = buffer[i];
45      }
```

这里没有选择使用头尾指针维护的队列，因为考虑到很多情况下pop出去的不止是单个字符，而是一个字符串，使用队列会出现字符串在容器中存储不连续的情况，而使用string则可以更方便地利用substr的api解决这一部分：

```
34  //! \param[in] len bytes will be copied from the output side of the buffer
35  string ByteStream::peek_output(const size_t len) const {
36      assert(len <= _curr_size && "read too much bytes");
37      return this->buffer.substr(0, len);
38  }
```

1.2 StreamReassembler

由于实验指南中要求“不要存储冗余的bytes”，所以最终选择了一个最朴素的设计：只保留一个buffer。

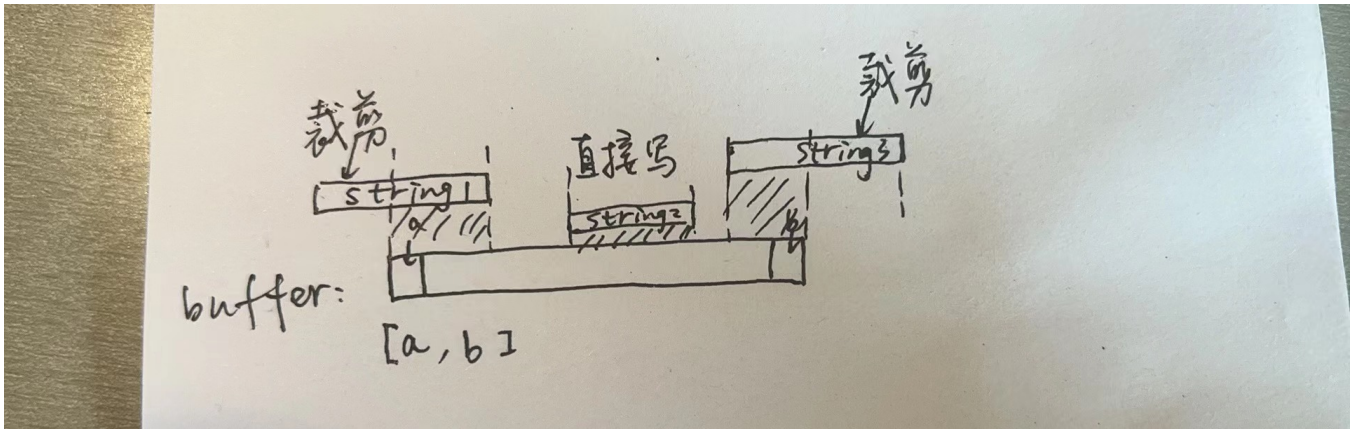
对于这个buffer，它实际上对应了原始数据流中的一个“下标区间”，随着buffer中的数据写进自己的ByteStream中写数据，buffer的“下标区间”在原始数据流中“滑动”。

不妨设buffer的“物理下标”是p_index，某个字节在原始数据流中的“逻辑下标”是l_inedx, 并使用_offset变量来维护两者之间的联系，如下：

```
26
27      size_t _offset; // physical_index = logic_index - offset
28
```

每次来临一个新的substring的时候，需要先对其进行“裁剪”，使其正确落入buffer内：

假设buffer对应的逻辑下标的区间是在$[a,b]$, 那么substring中不在此区间内的部分都需要被裁剪：



类似如下代码：

```
        else if(index < _offset && end_index < _offset + _capacity) {
            // cut data head.
            written_data = data.substr(_offset - index, data.length() - _offset + index);
            written_begin = _offset;
        }
        else if(index >= _offset && end_index >= _offset + _capacity) {
            // cut data tail
            written_data = data.substr(0, _offset + _capacity - index);
            written_begin = index;
        }
        else if(index < _offset && end_index >= _offset + _capacity) {
            // cut data both head and tail
            written_data = data.substr(_offset - index, _capacity);
            written_begin = _offset;
        }
```

这样，就保证了在StreamReassembler中存储的bytes是无冗余的。

此外，再使用一个bool数组来维护buffer中的byte是否使用，并且根据其中true的数量计算unassembled_bytes。

2 运行结果

```
14/16 Test #50: t_address_dt .....................   Passed    0.02 sec
      Start 51: t_parser_dt
15/16 Test #51: t_parser_dt ......................   Passed    0.02 sec
      Start 52: t_socket_dt
16/16 Test #52: t_socket_dt ......................   Passed    0.02 sec

100% tests passed, 0 tests failed out of 16

Total Test time (real) =  15.31 sec
[100%] Built target check_lab1_2
```

3 杂谈

补充一点点调试的小经验：因为ByteStream和StreamAssembler中都有一些私有的、用于维护的变量，如果编写自己的测试样例的时候，难免需要访问其中的私有变量，于是就搭建了一个小小的测试框架：

C++

```
// test.cpp
class Test {
public:
    static void print_state(const ByteStream& t) {
        ...
    }

    static void print_state(const StreamReassembler& t) {
        ...
    }
};

int main() {
    ...
    StreamReassembler t(10);
    t.push_substring("abcdefgh", 0, false);
    Test::print_output(t, 8);
    ...
    return 0;
}
```

然后，只要把Test类设置为需要测试的类的friend即可，调试起来方便许多。