计网 Lab1 实验报告

221220134 佟一飞

一、 程序结构与设计

用 vector 维护若干待放入字节流的区间，保证这些区间不重叠且按照左端点从小到大的顺序。

当插入一个新区间的时候，设当前字节流在等待 expected 开始的字符串，首先裁剪掉区间小于 expected 的部分，然后裁剪掉超出 reassembler 容量的部分：

```
if ( is_last_substring )
  eof_index = first_index + data.size();
// first slice
if ( first_index < expected ) {
  if ( expected - first_index > data.size() )
    return;
  uint64_t n = data.size();
  assert( expected - first_index <= data.size() );
  data = data.substr( expected - first_index, n - ( expected - first_index ) );
  first_index = expected;
}
if ( first_index - expected > output_.writer().available_capacity() )
  return;
if ( first_index + data.size() - expected > output_.writer().available_capacity() ) {
  data = data.substr( 0, output_.writer().available_capacity() - ( first_index - expected ) );
}
```

之后遍历 vector，裁剪掉待插入的区间中已经存在于 vector 的部分：

设当前待插入区间为[l,r]，遍历到 vector 中区间[li,ri]，如果[l,r]完全包含于[li,ri]，则销毁待插入区间。如果完全无交集且 r<li，则结束循环，如果完全无交集且 l>ri，则 continue。否则用 string 的 substr 方法切分待插入区间。

```
// second slice(make sure buffer is sorted)
vector<pair<uint64_t, string>> temp = {};
pair<uint64_t, string> now;
now.first = first_index, now.second = data;
int flag = 0;
for ( size_t i = 0; i < buffer.size(); i++ ) {
  uint64_t l = buffer[i].first, r = buffer[i].first + buffer[i].second.size() - 1;
  if ( now.first > r )
    continue;
  else if ( now.first >= l ) {
    if ( now.first + now.second.size() - 1 <= r ) {
      flag = 1;
      break;
    } else {
      uint64_t n = now.second.size();
      assert( r - now.first + 1 <= now.second.size() );
      now.second = now.second.substr( r - now.first + 1, n - ( r - now.first + 1 ) );
      now.first = r + 1;
    }
  } else {
    pair<uint64_t, string> res;
    res.first = now.first;
    res.second = now.second.substr( 0, min( l - now.first, now.second.size() ) );
    temp.push_back( res );
    if ( now.first + now.second.size() - 1 > r ) {
      uint64_t n = now.second.size();
      assert( r - now.first + 1 <= now.second.size() );
      now.second = now.second.substr( r - now.first + 1, n - ( r - now.first + 1 ) );
      now.first = r + 1;
    } else {
      flag = 1;
      break;
    }
  }
}
if ( !flag )
  temp.push_back( now );
```

把裁剪后得到的若干小区间插入 vector 即可，注意保证 vector 的有序性：

```
// inserting temp
size_t pos = 0;
for ( size_t i = 0; i < temp.size(); i++ ) {
  int flg = 0;
  while ( pos < buffer.size() ) {
    if ( temp[i].first < buffer[pos].first ) {
      buffer.insert( buffer.begin() + pos, temp[i] );
      stored += temp[i].second.size();
      flg = 1;
      break;
    } else
      pos++;
  }
  if ( !flg ) {
    buffer.push_back( temp[i] );
    stored += temp[i].second.size();
  }
}
```

之后遍历 vector，把左端点等于 expected 的所有区间放入字节流直到左端点大于 expected，然后把放入字节流的区间从 vector 中删除：

```
// pushing
for ( size_t i = 0; i < buffer.size() - 1; i++ ) {
  if ( ( buffer[i].first > buffer[i + 1].first ) && buffer.size() == 3 )
    assert( 0 );
}
for ( size_t i = 0; i < buffer.size(); i++ ) {
  if ( expected == buffer[i].first ) {
    output_.writer().push( buffer[i].second );
    expected += buffer[i].second.size();
    stored -= buffer[i].second.size();
  } else
    break;
}
while ( buffer.size() > 0 && buffer[0].first <= expected )
  buffer.erase( buffer.begin() );
```

当 expected 等于 eof 的右端点的时候，关闭字节流：

```
if ( eof_index != 1145141919810 && eof_index == expected )
  output_.writer().close();
```

Byres_pending 直接用一个变量记录 vector 中总长度就行了：

```
uint64_t Reassembler::bytes_pending() const
{
  // Your code here.
  return stored;
}
```

二、 实验结果

```
tongyf@tongyf-virtual-machine:~/Desktop/minnow$ cmake --build build --target check1
Test project /home/tongyf/Desktop/minnow/build
      Start  1: compile with bug-checkers
 1/17 Test  #1: compile with bug-checkers ........   Passed    7.91 sec
      Start  3: byte_stream_basics
 2/17 Test  #3: byte_stream_basics ...............   Passed    0.02 sec
      Start  4: byte_stream_capacity
 3/17 Test  #4: byte_stream_capacity .............   Passed    0.02 sec
      Start  5: byte_stream_one_write
 4/17 Test  #5: byte_stream_one_write ............   Passed    0.02 sec
      Start  6: byte_stream_two_writes
 5/17 Test  #6: byte_stream_two_writes ...........   Passed    0.03 sec
      Start  7: byte_stream_many_writes
 6/17 Test  #7: byte_stream_many_writes ..........   Passed    0.15 sec
      Start  8: byte_stream_stress_test
 7/17 Test  #8: byte_stream_stress_test ..........   Passed    0.04 sec
      Start  9: reassembler_single
 8/17 Test  #9: reassembler_single ...............   Passed    0.02 sec
      Start 10: reassembler_cap
 9/17 Test #10: reassembler_cap ..................   Passed    0.02 sec
      Start 11: reassembler_seq
10/17 Test #11: reassembler_seq ..................   Passed    0.07 sec
      Start 12: reassembler_dup
11/17 Test #12: reassembler_dup ..................   Passed    0.06 sec
      Start 13: reassembler_holes
12/17 Test #13: reassembler_holes ................   Passed    0.02 sec
      Start 14: reassembler_overlapping
13/17 Test #14: reassembler_overlapping ..........   Passed    0.02 sec
      Start 15: reassembler_win
14/17 Test #15: reassembler_win ..................   Passed    0.55 sec
      Start 37: compile with optimization
15/17 Test #37: compile with optimization ........   Passed    2.99 sec
      Start 38: byte_stream_speed_test
            ByteStream throughput: 0.62 Gbit/s
16/17 Test #38: byte_stream_speed_test ...........   Passed    0.24 sec
      Start 39: reassembler_speed_test
            Reassembler throughput: 4.05 Gbit/s
17/17 Test #39: reassembler_speed_test ...........   Passed    0.24 sec

100% tests passed, 0 tests failed out of 17

Total Test time (real) =  12.44 sec
Built target check1
```