Computer Networks Project Report

章凌豪

13307130225@fudan.edu.cn

Overview

本次 Project 用 Python 2.7 开发。

网络部分使用了 socket + select，服务器部分使用了 threading，GUI 使用了 PyQt4。

Protocol Design

Overview

Claptrap 是应用层的字符串协议，每个包由 12 字节的 Header 和不定长度的 Payload 组成。

所有字符以 UTF-8 编码。

Header

Header 由 8 个字节的 CLAPTRAP （协议标识）和 4 个字节的 `Packet-Length` 域组成。

Payload

Payload 基本格式如下：

```
Method
Parameter0: Value 0
Parameter1: Value 1
...
/n
Content
```

也就是第一行指定方法，接下来若干行给出一系列参数，再间隔一个空行后是内容域。

所有带有内容域的包均有 `Content-Length` 参数来验证长度。

下面列出所有请求和响应的格式，每个方法或是参数的含义根据其名字应该是很明确的。

Client

Request Method	Description	Parameters	Content	Possible Response Methods
LOGIN	用户登陆	{Username, Password}	None	RESP_LOGIN
LOGOUT	用户注销	None	None	RESP_LOGOUT
LIST_ROOMS	获取房间列表	{Room-Name}	None	RESP_LIST_ROOMS
LIST_MEMBERS	获取当前房间用户列表	None	None	RESP_LIST_MEMBERS
JOIN_ROOM	进入房间	{Room-Name}	None	RESP_JOIN_ROOM

EXIT_ROOM	退出房间	None	None	RESP_EXIT_ROOM
SEND_MSG	发送消息	{Content-Length}	Message	RESP_SEND_MSG
RECV_MSG	接收消息	None	None	NO_NEW_MSG / NEW_MSG

Server

Response Method	Possible Status	Parameters	Content
RESP_LOGIN	OK / AUTH_FAILED	{Status}	None
RESP_LOGOUT	OK / NOT_LOGIN	{Status}	None
RESP_LIST_ROOMS	OK / NOT_LOGIN	{Status, Content-Length}	Rooms
RESP_LIST_MEMBERS	OK / NOT_IN_ROOM / NOT_LOGIN	{Status, Content-Length}	Members
RESP_JOIN_ROOM	OK / NO_SUCH_ROOM / NOT_LOGIN	{Status}	None
RESP_EXIT_ROOM	OK / NOT_IN_ROOM / NOT_LOGIN	{Status}	None
RESP_SEND_MSG	OK / NOT_IN_ROOM / NOT_LOGIN	{Status}	None
NO_NEW_MSG	NOT_IN_ROOM / NO_NEW_MSG / NOT_LOGIN	{Status}	None
NEW_MSG	OK	{Status, Content-Length}	Messages

Content Formats

下面列出所有内容域的格式：

SEND_MSG > Message

```
Message
```

RESP_LIST_ROOMS > Rooms

由 `\n` 分隔的若干房间名。

```
Room0\n
Room1\n
...
```

RESP_LIST_MEMBERS > Members

由 `\n` 分隔的若干用户名。

```
Member0\n
Member1\n
...
```

NEW_MSG > Messages

由 `\n` 分隔的若干行消息，每行消息含发送方、发送时间、消息内容，由`\t`分隔。

```
From\tTime\tContent\n
From\tTime\tContent\n
...
```

Implementation Details

这一部分主要讨论代码实现上的细节，但略去所有与业务逻辑相关的部分不谈。

Network

Why select?

本次 Project 使用了 socket + select 进行网络通讯。其中 select 是一个 Python 自带的用于监视多个文件描述符的模块，常用于管理 socket 的 IO 事件。使用 select 的理由有三：

1. 使用 Python 进行 socket 开发时 select 基本上是标配。
2. Python 的 socket 特性使得如果不使用 select 来管理 socket 事件，就只能在 blocking socket 和 non-blocking socket 之间二选一。如果选择 blocking socket，那么 server 只能同时连接一个 client，新的 client 的连接请求会被一直阻塞；如果选择 non-blocking socket，那么在一个请求不能马上完成的情况下就会触发异常，也就无法保持长连接。一个折中方案是采用 non-blocking socket + timeout，给每个 socket 允许很长的超时，但这样不仅不自然而且很丑陋。
3. select 在这里的作用只是监视所有的 socket IO，使得多用户同时在线成为可能，实际的网络通讯还是由 socket 进行。

Socket tricks

为了更好地符合需求，需要对 Python 的 socket 进行一些封装。

首先，由于 `socket.recv` 所能收取的最大缓冲区长度是由系统控制的，我们就要利用包中的 `Packet-Length` 域来确保每次从缓冲区中读取一个完整的包。

网络部分的核心是以下两个与接收包有关的函数：

```
# 从缓冲区收取长度为 length 的数据
def recv_all(sock, length):
    length = int(length)
    data = ''
    received = 0
    while received < length:
        part = sock.recv(length - received)
        if len(part) == 0:
            # 长度不匹配
            raise socket.error('Socket recv EOF')
        data += part
        received += len(part)
    return data
```

```
# 从缓冲区收取一整个包
def recv_packet(sock):
    try:
        # 先收取 8 字节协议标识
        buff = recv_all(sock, 8)
        try:
            assert buff == 'CLAPTRAP'
        except:
            raise BadProtocolException
        # 再收取 4 字节 Packet-Length
        buff = recv_all(sock, 4)
        # struct 用于将整型数据编码为字符串
        packet_length = struct.unpack('!I', buff)[0] - 12
        # 收取剩下的数据
        packet_data = recv_all(sock, packet_length)
    except socket.error as e:
        raise
    return packet_data
```

发送包相对来讲就轻松许多，主要通过如下函数封装数据：

```
def packetify(data):
    length = len(data) + 12
    protocol = 'CLAPTRAP'
    length = struct.pack('!I', length)
    packet_data = protocol + length + data
    return packet_data
```

再使用 `socket.sendall(packetify(data))' 发送即可。

Server Class

Server 类从 `threading.Thread` 继承而来，通过以下代码实现对每一个 client 创建一个新的线程维护与其的连接：

```
server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
server_socket.bind((HOST, PORT))
server_socket.listen(5)

CONNECTION_LIST = []
CONNECTION_LIST.append(server_socket)

while True:
    read_sockets, write_sockets, error_sockets = select.select(CONNECTION_LIST,[],[])
    for sock in read_sockets:
        # 这是一个新 client
        if sock == server_socket:
            conn, addr = server_socket.accept()
            CONNECTION_LIST.append(conn)
            # 创建一个新 Server 实例（线程）
            server = ChatServer(conn, addr)
            server.start()
```

在 Server 内部通过以下代码来处理与 client 之间的通信：

```
try:
    read_sockets, write_sockets, error_sockets = select.select([self.conn],[],[])
    if len(read_sockets) > 0:
        # 收取一个包
        packet_data = libchat.recv_packet(self.conn)
        # 解包并触发对应的 request handler
        self.dispatch(libchat.parse(packet_data, libchat.REQUEST))
except:
    # client 断开连接，进程退出
    print 'client disconnected.'
    self.logout(None)
    return
```

Client Class

Client 类没有太多可讲的，有一点就是要注意处理字符编码。

协议中所有数据以 UTF-8 编码，而在服务器和客户端内部数据以 unicode 形式储存（为了方便字符串操作和 GUI 呈现）。那么在每个 request handler 的开头和结尾部分要做好编码转换工作。

User Interface

GUI 使用 PyQt4 实现，也没有太多可讲的。

默认每 1 秒钟拉取一次新消息，每 5 秒钟刷新房间列表和用户列表。

Demo Notes

提交文件中的 `chat_server.exe` 和 `chat_client_gui.exe` 是用 32 位 pyinstaller 导出的可执行文件，正常情况下应该能在各个平台上运行。

也可以从源码运行，依赖为 Python 2.7 和 PyQt4。

服务器默认端口为 6666，最大支持 5 个客户端同时在线。可以通过 `chat_server.exe PORT MAX_ONLINE` 来重新指定端口和最大同时在线数目。

服务器在启动时会从 `globals.txt` 中读取一些预设的账号和房间列表，为了测试程序可以通过修改该文件添加新的账号和房间，要求格式与原来相同且文件编码为 UTF-8，否则会报错。

启动客户端时可以使用预设的用户名和密码登录，在界面右上角可以看到房间列表，双击可以进入房间，右下角可以看到当前房间的在线用户。进入房间后可以发送消息（CTRL+ENTER）。由于业务逻辑不是这次 Project 的重点，也为了实现的便利，只接收当前房间的消息，不推送历史消息。