一个在线小说阅读器

Developed by 18307130071_赵予珩

整体架构

服务器端

服务器端按章节保存小说文本（.txt格式），监听以下三个端口：

1. 通用的消息传输端口 `65432` 用于检测用户连接，为用户提供基本的信息包括当前服务器中有哪些小说，每本小说有多少章节，支持并发访问。
2. 下载端口 `65500` 用于提供下载服务，接收用户请求并提供某小说某一章的下载服务，支持并发访问。
3. 在线阅读端口 `65501` 用于在线阅读传输缓存文件，执行逻辑与下载端口类似，支持并发访问。

客户端

客户端采用python提供的tkinter库绘制了简单的图形界面，提供以下功能：

1. 小说选择：从服务器的 `65432` 端口接收 当前小说列表并为用户展示选择。
2. 对选定小说的二级操作

    1）下载：连接服务器的 `65500` 端口，按章节并发下载。

    2）在线阅读

    在线阅读通过缓存的形式进行，用户选定某一页的时候检测缓存，如果无对应缓存则请求服务器的 `65501` 端口进行缓存下载。在系统退出的时候会清空缓存。具体提供以下功能：

    a. 章节选择

    b. 翻页

    c. 保存书签：将当前浏览的页面存为书签

    d. 跳页：切换到指定页或切换到书签位置

设计思路

核心思想

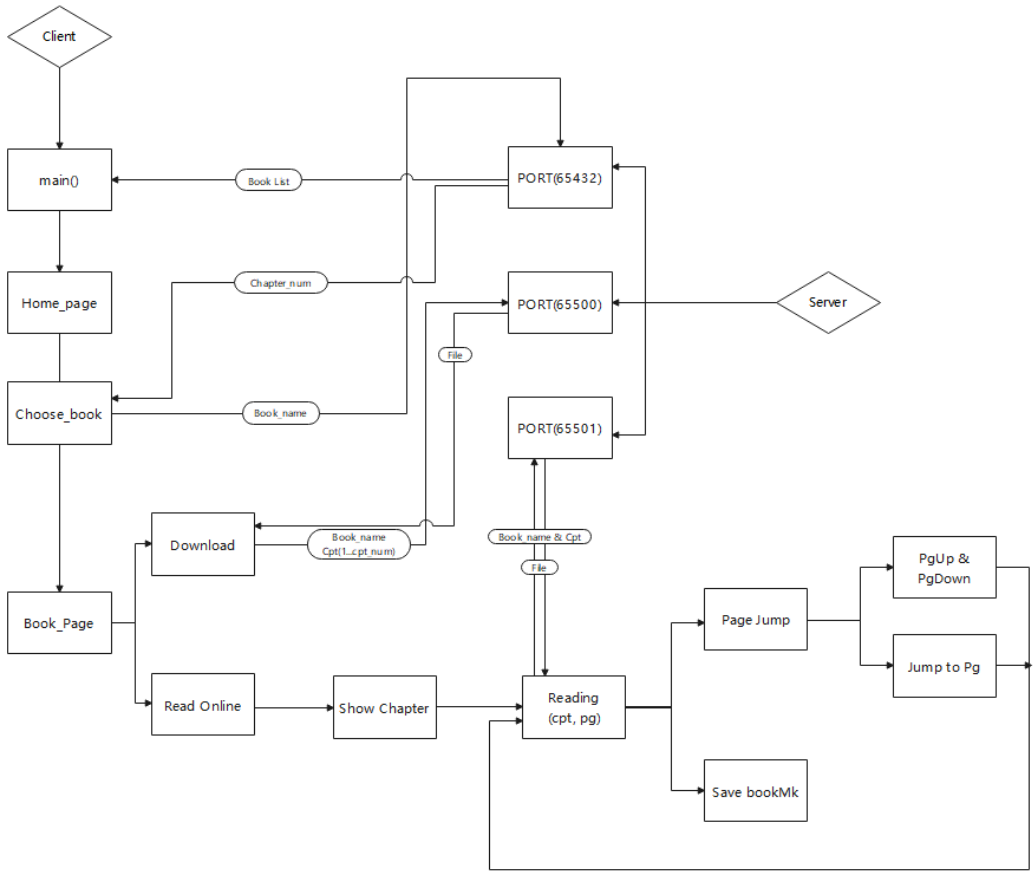
从用户需求出发，实现了一个胖客户端-瘦服务器模型。客户端提供UI以及操作选择，页面切换等功能并且向服务器的不同端口发送请求数据。服务器只需要监听来自不同端口的请求并调用文件传输功能提供响应。

采用多线程的方法实现功能。服务器端对于所监听的3个端口都实现了多线程：当检测到新的连接的时候，启动新的线程，根据不同的端口调用不同的处理方法进行相应。客户端在下载的时候也采用了多线程的思路：下载某本小说时会打开多个线程，每个线程并发地对该小说的某一章进行下载，以提高下载效率。

考虑到本次实验中采用面向连接的方式（TCP）进行数据传输，需要处理可能产生的粘包问题：对于可能出现粘包的连接，先传输一个定长（2字节）的消息头部来表示消息长度。接收端收到这个头部以后可以直接接收相应长度的消息进行处理。这样就解决了面向连接传输产生的粘包问题

执行逻辑

程序的执行逻辑如下图所示：



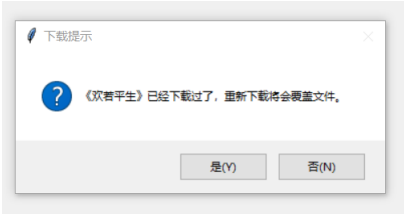
由上图可以看出用户端提供了较多的方法，服务器端仅提监听三个不同的端口并分别对请求提供相应：在 `65432` 端口，当检测到新的连接建立的时候，向用户端传输服务器中所存储的小说的信息；当用户选定某本小说的时候，接收用户选择的书名，向用户端传输 `chapter_num` ，告诉用户这本小说有多少章。在 `65500` 端口，当用户选择下载某本小说的时候，可以检测到来自用户的并发连接请求，多线程地处理多个请求，服务器的每个子线程都能接收一个用户子线程发送的书名和章节号，并确定传输哪本书的哪一章节给这个用户子线程连接。在 `65501` 端口，当用户选择在线阅读某一章且无本地缓存（不考虑下载目录）的时候，会连接这个端口发送缓存请求，服务器会检测到连接，接收用户端发送的书名和章节号，并传输相应文件给用户。

服务器对于上述三个端口，都通过多进程处理，支持并发访问。

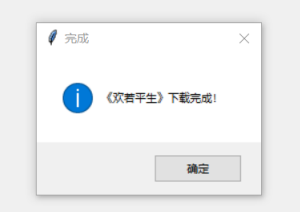
用户端的根据用户在界面上的不同选择来提供不同的功能。首先，用户端启动后先连接服务器，获取书目信息并缓存到内存中。接着进入主界面，提供一系列书目给用户选择：

当用户选定了某本小说后，客户端会向服务器发送书名，并接收服务器传来的章节数目。随后切换界面至书籍界面，供用户选择执行不同的操作包括在线阅读和下载：

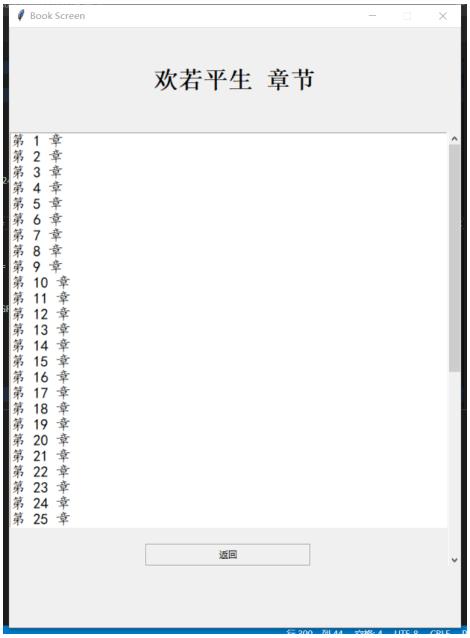
用户选择 `下载` 后，客户端代码会先检查 `.\Downloads\book_name` 目录下是否有小说文件，如果有的话表示已经下载，会弹出提示确认是否重新下载：



用户确认重新下载或者下载目录内没有该小说则客户端运行多个子进程分别向服务器的 `65500` 端口请求该小说的第1章到第 `chapter_num` 章的文件传输，完成后输出提示信息：（若下载出现异常也会报错）



用户选择 `在线阅读` 后，切换至章节选择页，根据之前缓存的章节数目提供不同的章节给用户选择，鼠标滚轮进行上下滑动：



选定某一章后，切换到阅读界面，同时客户端检查有无缓存，若 `.\Temp\book_name\book_namecpt.txt` 不存在则向服务器的 `65501` 端口请求缓存下载，并分页显示在阅读界面上：

阅读界面提供翻页，跳页和保存书签功能，同时可以返回上级章节选择页面。翻页和跳页的逻辑差不多，根据要跳转的目标页面和章节为参数重新打开一个新的阅读界面即可，当目标页面不存在的时候会跳出错误信息。跳页时从弹窗中获取用户的输入信息，根据输入信息进行跳转，也支持跳转到书签页面，前提是用户已经存储了书签：

`存为书签` 选项会记录当前的章节以及页号作为一个tuple，映射到一个全局的 `BookMks` 字典中，唯一对应一个书名。所以每本书一次只能保存一个书签，系统初始会把 `BookMks` 中的所有tuple置为 `(-1, -1)` 。所以在退出系统的时候不会保存书签信息。如果 已有书签则会弹出提示询问是否替换。若无书签则直接更新书签信息：

技术实现

多线程

本次实验中分两个方面实现了多线程

1. 服务器端多线程处理用户程序对端口的连接：

```
## 这里是服务器端的main函数逻辑
def main():
    # 服务器端设置三个连接
    tcp_server_connection_socket = socket.socket(socket.AF_INET,
socket.SOCK_STREAM)
    tcp_server_download_socket = socket.socket(socket.AF_INET,
socket.SOCK_STREAM)
    tcp_server_reading_socket = socket.socket(socket.AF_INET,
socket.SOCK_STREAM)
    # 分别捆绑到三个不同的端口
    tcp_server_connection_socket.bind((HOST,PORT))
    tcp_server_download_socket.bind((HOST,DOWNLOAD_PORT))
    tcp_server_reading_socket.bind((HOST,READING_PORT))
    # 设置允许同时连接的最大数量
    tcp_server_connection_socket.listen(128)
    tcp_server_download_socket.listen(128)
    tcp_server_reading_socket.listen(128)
    # 分别打开三个进程对三个端口进行监听
    _thread.start_new_thread(listen_connection,("Listen connection",
tcp_server_connection_socket))
    _thread.start_new_thread(listen_download, ("Listen download",
tcp_server_download_socket))
    _thread.start_new_thread(listen_reading, ("Listen reading",
tcp_server_reading_socket))

    while True:
```

```
        # 轮询主连接状态，若服务器端主连接关闭，则表示服务器超时（主连接超时时间3600s)或连接次数达到上限，服务器关闭
        if getattr(tcp_server_connection_socket, '_closed') == True:
            print("Main server stop")
            break
```

```
## 这里是三个监听函数，监听了三个不同的端口，由三个不同的进程调用执行
def listen_download(name, sock):
    # 监听下载端口65500
    index = 0
    while True:
        new_socket_client, client_addr = sock.accept()
        index += 1
        # 检测到该端口上的连接到来，打开一个新的线程调用下载处理函数download_connection来进行处理
        _thread.start_new_thread(download_connection,(index, new_socket_client, client_addr))
        # 下载请求大于1024后退出
        if index > 1024:
            break
    sock.close()
    _thread.exit()
## 下面两个函数分别监听了65432端口和65501端口，与上面的下载端口监听逻辑相同
def listen_connection(name, sock):
    ...
def listen_reading(name, sock):
    ...
```

这样设计的服务器可以支持多用户的并发访问，下载和阅读。

2. 用户端多线程地请求服务器 `65500` 端口进行小说下载：

```
def download(file_name, fold):
    # 判断是否已经下载，若有，弹出提示
    path = ".\\" + fold + "\\" + file_name + "\\" + file_name + str(1) + ".txt"
    if os.path.exists(path) and fold == "Downloads":
        hint = "《" + file_name + "》已经下载过了，重新下载将会覆盖文件。"
        Continue = tkinter.messagebox.askquestion('下载提示', hint)
        print(Continue)
        if Continue == "yes":
            for i in range(chapter_num):
                path = ".\\" + fold + "\\" + file_name + "\\" + file_name + str(i+1) + ".txt"
                os.remove(path)
        else:
            return
    try:
        # 检查是否有相应子文件夹，若无则创建
        dir_name = ".\\" + fold + "\\" + file_name
        if not os.path.isdir(dir_name):
            os.mkdir(dir_name)
        cpt = 1
        while cpt <= chapter_num:
            # 循环对每一章进行下载
            print("ask for downloading chapter %d..." % cpt)
            # 开始一个子线程下载这一章，调用download_cpt函数执行
```

```
            _thread.start_new_thread(download_cpt, (file_name, cpt, fold))
            cpt += 1
    except Exception:
        err = "《" + file_name + "》下载失败。"
        tkinter.messagebox.showerror("下载错误", err)
    i = 1
    while i <= chapter_num:
        while True:
            try:
                f = open(".\\" + fold + "\\" + file_name + "\\" + file_name
+ str(i) + ".txt", "rb")
                f.close()
            except Exception:
                continue
            break
        i += 1
    hint = "《" + file_name + "》下载完成！"
    tkinter.messagebox.showinfo("完成", hint)
```

实现多线程下载可以加快下载速度，尤其是当章节较多的情况下性能较好。

考虑到执行多线程下载时，服务器端只监听下载端口并作出响应而不区分是何用户连接，在每次下载时，用户程序需要先传输书目信息和章节信息给服务器端，服务器端再根据相应的信息进行相应，这样就避免了传错文件或者无法识别用户身份所带来的问题：

```
## 子线程调用，向服务器请求某本书某章的文件传输
def download_cpt(file_name, cpt, fold):
    path = ".\\" + fold + "\\" + file_name + "\\" + file_name + str(cpt) +
".txt"
    tcp_download_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    tcp_download_socket.connect((HOST, DOWNLOAD_PORT))
    # 先传输书名和章节号，进行了粘包处理操作，在下一部分进行了介绍
    msg = bytes(file_name, encoding = "utf-8")
    # 先将书名长度存为length
    length = len(msg)
    # 先传输2字节的length作为头部给服务器端
    tcp_download_socket.send(length.to_bytes(length=2, byteorder='big',
signed=True))
    # 再传输书名
    tcp_download_socket.send(msg)
    # 传输2字节长度的章节号给服务器端
    tcp_download_socket.send(cpt.to_bytes(length=2, byteorder='big',
signed=True))
    # 接收数据
    recv_data = tcp_download_socket.recv(1024)
    while recv_data:
        with open(path, "ab") as f:
            f.write(recv_data)
        recv_data = tcp_download_socket.recv(1024)
```

TCP粘包处理

设置socket时采用的是 `socket.socket(socket.AF_INET, socket.SOCK_STREAM)` 面向连接的文件传输也就是TCP，是基于流的，会产生粘包。也就是说TCP所传输的数据是一连串数据，没有界限。发送方发送的若干个数据包到达接收方时，全部都放在接收缓冲区，后一包数据的头紧接着前一包数据的尾，应用程序无法区分数据包的头和尾。所以说要处理粘包。本次实验中采用的粘包处理方法是在传输每条信息之前先发送一个定场（2字节）的头部信息表示下一个要传输的信息的长度，接收方只需要按照这个长度来接收数据就可以区分不同的数据。上面 `download_cpt` 函数已经展示了发送时的处理方法。我们继续以文件下载为例展示接收方（这里是服务器）对于头部的处理：

```
## 服务器端的下载处理函数，检测到65500端口的连接时，启动子进程执行这个函数
def download_connection(index, connection, client_addr):
    try:
        # 超时36000秒也就是10小时（主连接和阅读连接的超时为1小时）
        connection.settimeout(36000)
        # 先从客户端接收2字节的头部作为接收长度
        length = connection.recv(2)
        length = int().from_bytes(length, byteorder='big', signed=True)
        # 按这个长度接收接下来的书名
        file_name = connection.recv(length).decode("utf-8")
        # 最后接收2字节的章节号
        chapter_index = connection.recv(2)
        chapter_index = int().from_bytes(chapter_index, byteorder='big',
signed=True)
        print("Begin downloading %d at client address: %s" % (index,
client_addr))
        print("file name:", file_name)
        print("chapter index:", chapter_index)
        file_content = None
        try:
            f = open(".\Books\\" + file_name + "\\" + file_name +
str(chapter_index) + ".txt", "rb")
            file_content = f.read()
            f.close()
        except Exception as res:
            path = ".\Books\\" + file_name + "\\" + file_name +
str(chapter_index) + ".txt"
            print("Invalid file name:", path)
        if file_content:
            # 传输相应的文件给客户端
            connection.send(file_content)
        print("Download connection %d closed" % index)
        connection.close()
    except socket.timeout:
        print("Downloading time out!")
        connection.close()
    _thread.exit()
```

这里下载时只传送了一个书名这种不定长的数据，所以只使用了一个头部，在连接刚建立的时候需要从服务器端传输所有书的书名给客户端，只需要在传输每一个书名之前都配上一个头部，即可实现粘包问题的解决。值得注意的是，在传输多条不定长数据时，头部的另一个作用是可以表示数据流的结束，即当传输的头部header解码后等于end时，表示这之后不会再传输相关的数据。在客户端可以执行下面的代码来接收全部的小说列表：

```
## 从服务器接收小说列表，采用头部避免粘包
# 设置结束值为-1，避免与某个长度值冲突
```

```
end = -1
header = tcp_client_connection_socket.recv(2)
header = int().from_bytes(header, byteorder='big', signed=True)
while header != end:
    # 循环接收直到收到的header为结束标记
    book_name = tcp_client_connection_socket.recv(header)
    book_name = book_name.decode("utf-8")
    # 更新书籍列表到内存中
    Books.append(book_name)
    # 更新书签信息到内存中
    BookMks[book_name] = (-1, -1)
    # 接收下一个header
    header = tcp_client_connection_socket.recv(2)
    header = int().from_bytes(header, byteorder='big', signed=True)
```

其它技术细节

1. 在UI中采用了Listbox + Scrollbar的形式进行了章节展示，并且设置了点击某个表项即可跳转到阅读的机制：

```
# Scrollbar用于鼠标滚轮上下滑动
bar = Scrollbar(interface)
bar.pack(side = RIGHT, fill = Y)
# Kistbox展示所有章节供选择
lb = Listbox(interface, yscrollcommand = bar.set, selectmode='single',
width=60, height=25, font="kaiti")

def CurSelet(evt):
    value=lb.get(lb.curselection())
    pos, cpt, i = len(value) - 3, 0, 0
    # 获取表项值
    while True:
        cpt += int(value[pos]) * (10 ** i)
        pos -= 1
        i += 1
        if value[pos] == " ":
            break
    interface.destroy()
    # 跳转至相应章节的第一页阅读
    reading_page(file_name, cpt, 1, sock)

# 设置选择某条表项时执行跳转
lb.bind('<<ListboxSelect>>',CurSelet)
for i in range(chapter_num):
    lb.insert(END,"第 " + str(i+1) + " 章")
    lb.pack(fill = "y", pady = (0, 20))
    bar.config(command=lb.yview)
```

2. 为了使切换窗口更平滑，采用在Frame上绘制用户界面的方式，在切换时销毁上一个Frame而不是销毁整个window，不会产生空窗期。
3. 由于是在线阅读，所有阅读过的章节会保存在 `.\Temp\book_name\` 文件夹内，在正常退出系统（从主界面的退出按钮退出）时，会清空 `Temp` 文件夹，实现了缓存处理：

```
def quit_system():
    temp_dir = ".\Temp"
    # 清除Temp文件夹
    shutil.rmtree(temp_dir)
    global win
    # 关闭窗口
    win.destroy()
    # 重新创建空的Temp文件夹
    os.mkdir(temp_dir)
```

4. 调用了socket包实现socket，_thread包实现多线程，os包和shutil包实现客户端的文件管理。

以上就是设计文档的全部内容了，具体细节请结合前述的流程图参照源代码。