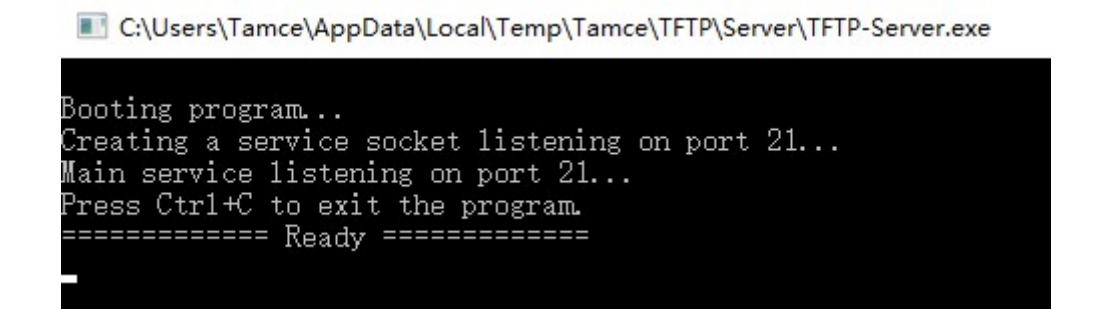


使用方法：

服务端程序：

直接打开运行即可，服务端程序会暴露其所在目录下的文件；服务端自动监听 21 端口，目前暂时就不写根据参数监听端口了。

服务端支持多线程，最大连接数理论可达到 1024（连接服务端时服务端随机返回的端口范围是 1024，但因为没有做碰撞检测，因此连接数大时可能导致冲突而发生异常）

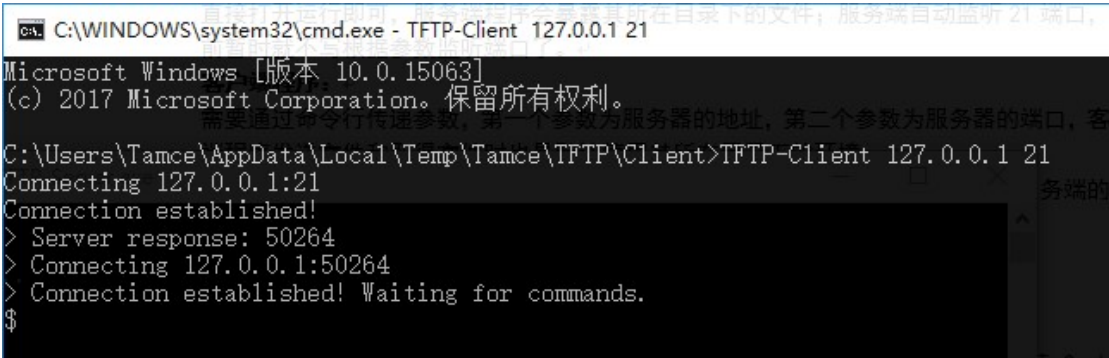


监听中的服务端程序

客户端程序：

需要通过命令行传递参数，第一个参数为服务器的地址，第二个参数为服务器的端口，客户端程序发送文件和取得文件时也是默认使用其所在目录下的环境。

客户端程序在运行后会进入<命令模式>，此时只需要用户输入命令就可以完成与服务端的交互。

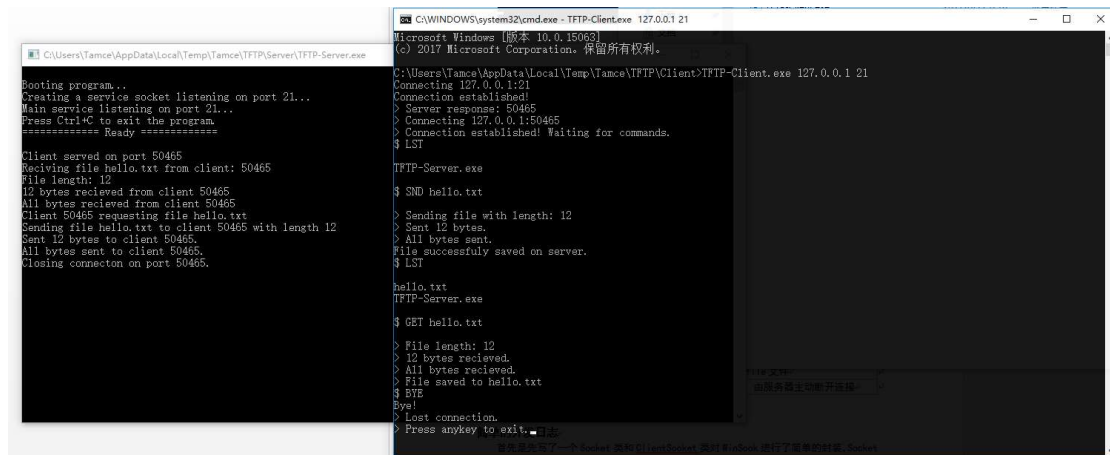


客户端的命令模式

可用的命令格式和作用如下：

命令格式	作用
HEY	向服务端发送问好信息
SND file	向服务端发送 file 文件
LST	向服务端请求文件列表
GET file	从服务端下载 file 文件
BYE	向服务器告别，由服务器主动断开连接

下图展示了一次较完整的交互过程



```
C:\Users\Tamece\AppData\Local\Temp\Tamece\TFTP\Server>TFTP-Server.exe
Booting program...
Creating a service socket listening on port 21...
Main service listening on port 21...
Press Ctrl+C to exit the program.
***** Ready *****

Client served on port 50465
Receiving file hello.txt from client: 50465
File length: 12
12 bytes recieved from client 50465
All bytes recieved from client 50465
Client 50465 requesting file hello.txt
Sending file hello.txt to client 50465 with length 12
Sent 12 bytes to client 50465.
All bytes sent to client 50465.
Closing connection on port 50465.

C:\WINDOWS\system32\cmd.exe - TFTP-Client.exe 127.0.0.1 21
Microsoft Windows [版本 10.0.15063]
(c) 2017 Microsoft Corporation. 保留所有权利。

C:\Users\Tamece\AppData\Local\Temp\Tamece\TFTP\Client>TFTP-Client.exe 127.0.0.1 21
Connecting 127.0.0.1:21
Connection established!
> Server response: 50465
> Connecting 127.0.0.1:50465
> Connection established! Waiting for commands.
$ LST
TFTP-Server.exe
> SND hello.txt
> Sending file with length: 12
> Sent 12 bytes.
> All bytes sent.
File successfully saved on server.
$ LST
hello.txt
TFTP-Server.exe
$ GET hello.txt
> File length: 12
> 12 bytes recieved.
> All bytes recieved.
> File saved to hello.txt
$ BYE
Eye!
Lost connection.
Press anykey to exit.
```

一次较为完整的交互过程

## 简单的开发日志

首先是先写了一个 `Socket` 类和 `ClientSocket` 类对 `WinSock` 进行了简单的封装, `Socket` 类实例负责监听端口或者连接一个地址, 当连接成功的时候返回一个连接状态的 `ClientSocket` 实例, 或当监听的端口有客户端连接时, 通过 `accept` 接受一个请求时也返回一个连接状态的 `ClientSocket` 实例。

第一版本的程序只是对封装的 `Socket` 类进行测试, 让程序监听端口, 然后用 `telnet` 测试, 服务端能收发信息之后再用封装的 `Socket` 写客户端, 使用之前写好的服务端进行测试; 通过这个保证 `Socket` 类的封装和实现没有问题, 能够实现监听端口、连接、收发的基本功能。

第二个版本的时候尝试使用了多线程, 让服务端独立主线程之外启动一个服务线程来接受请求; 客户端也是用了多线程, 分为主线程、agent 线程和工作线程, 主线程中启动另外两个线程, agent 线程加入主线程中, 负责接收用户输入的指令, 然后 push 到一个 Task 队列中, 而工作线程则是不断的从 Task 队列中获取要完成的 Task 并且完成处理这个 Task (比如连接、发送、断开等)。在这期间发生了一些意想不到的问题, `ClientSocket` 会莫名其妙断开, 仔细思考无果后请教知乎才恍然大悟, 原来是临时对象的析构导致了 `ClientSocket` 的析构函数被调用从而其内部的 `SOCKET` 被断开, 之后使用 c++11 的移动语义来避免这个问题。

后来仔细想了想客户端的多线程是完全毫无意义的...于是在第三个版本中移除客户端的多线程 (可惜了那么多代码全删了, 不过使用了 git 进行版本控制, 随时要找回来看也是没问题的), 服务端参考 FTP 的实现, 主要服务提供在 21 端口, 在客户端连接后服务端发送一个随机的端口号给客户端然后断开连接, 之后服务端开启一个新的线程, 并在发送回客户端的那个端口号监听并处理客户的请求。

暂时不准备改进的已知缺点

- 1. 客户端连接 21 端口会让服务端在一个新端口开启一个线程监听请求，如果客户端迟迟向这个新端口发起连接请求的话，服务端的这条线程会一直执行，浪费资源。  
**解决方法：**新的这个工作线程在 accept 的时候启动一个定时 (使用 select)，如果一段时间内没有连接请求那么直接返回，结束线程的运行。
- 2. 每次有客户端连接，都会创建一个 workingThread 被 push 到 vector 中，而线程执行完成后 vector 中的记录依旧不会被删除。**解决方法：**定时执行一个清理方法，检查 vector 中的各个线程是否在工作，如果已经退出，则删除那个元素。
- 3. 服务端不管文件权限，在客户端要求取得文件的时候也不检查文件是否存在，客户端发送文件的时候也不管文件是否已经存在，这些也均未测试，预期可能发生死锁。  
**解决方法：**操作前进行细致的检查。

协议：

约定：

服务端发送的普通信息以 \0 代表结束。

流程：

由上至下为时间顺序

客户端	服务端
	监听 21 端口
连接 21 端口	接受连接请求
收到端口号 p	取随机端口号 p (50000-51024) 以字符串形式发送给客户端
	监听端口 p
被动断开 21 端口的连接	发送完毕主动断开 21 端口的连接
连接端口 p	接受连接请求
	服务端发送问好信息
	服务端持续等待客户端消息
客户端发送请求	服务端接受请求处理并发送结果
客户端发送请求	服务端接受请求处理并发送结果
客户端发送断开连接请求	服务端发送告别信息并主动断开连接
客户端被动断开连接	
(客户端主动断开连接)	(服务端被动断开连接)

客户端请求格式：

每个指令都占用 3 个字节，以字符串形式描述如下：

指令	作用
HEY	让服务端返回问好信息
SND	发送文件
LST	获取文件列表
GET	下载文件
BYE	断开连接

其他指令只需发送三个字节即可，只有 SND 和 GET 较为特殊，以下给出说明：

**SND:**

如果指令为 SND，则接下来第一个字节为空格，而后跟着 32 个字节的文件名，不足 32 字节在末尾补 0 字节填充；接下来连续的 4 个字节为一个整数，表示接下来的文件内容长度（字节）；而后是这么多的文件数据，如下图：

字节数	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36
说明	's'	'N'	'P'	' '	文件名																															
字节数	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71	72
说明	(int)n				n个字节的文件数据																															

**GET:**

客户端应该发送的 GET 指令如下：前三个字节为 GET，第四个字节为空格，接下来的 32 个字节代表要下载的文件名。

字节数	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36
说明	'G'	'E'	'T'	' '	文件名																															

服务端返回的数据格式为：

前四个字节代表一个整数 n，接下来的 n 个字节代表文件数据。