# 课程介绍

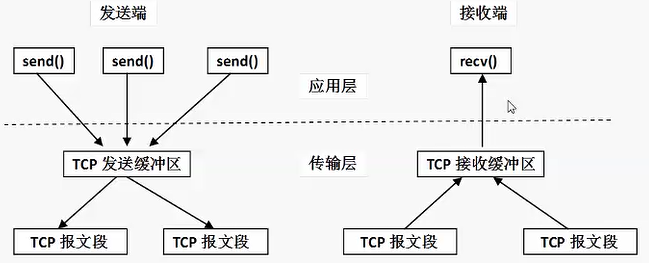
* 1. TCP数据IO处理;（了解）
* 2. TCP的自定义协议文件传输系统demo;（编写）
* 3. pickle序列化;（掌握）

1. TCP数据IO处理
   1. 什么叫IO处理

IO处理实际就是我们所说的文件对象的读写操作。

对于网络编程来说，就是socket的recv(recvfrom)和send(sendto)的操作。

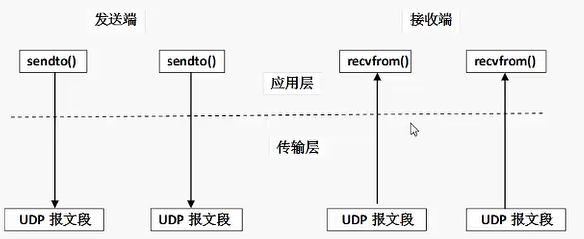
* 1. TCP数据流的数据通信总结
* 所谓的数据流，实际是依赖于系统内核层提供了缓存区的操作。
* TCP的发送和接收，都是从缓存区里获取数据，是一种典型的缓存IO操作。
* recv默认阻塞，等待写端写入数据，当收到空数据后，则代表写端已经关闭，4次挥手已经完成前2次。
* 在数据IO代码设计时，要求是接收端不断接收，直到发送端发送完成。接收端不应提前结束，除非发现发送端异常数据攻击。



* 1. TCP数据传输和UDP的比较

UDP发送端执行一次写操作（sendto)，UDP模块把它封装成一个UDP数据报并发送。

接收端针对每一个数据报执行读操作（recvfrom），否则就会发生丢包，并且如果用户没有指定足够的应用程序缓冲区来读取数据报，则UDP数据报就会被截断（接收不完整）。



1. TCP文件传输demo
   1. 任务需求

仿照SCP传输文件的方式，完成一个自定义协议的命令交互方式的文件传输服务器和客户端程序，要求：

1、客户端可以查看当前服务器共享的目录下的文件信息，比如文件名；

2、客户端可以下载服务器共享目录下的文件；

3、客户端可以上传文件到服务器的共享目录下；（选做）

* 1. demo实现
     1. 模型设计

在实际开发中，一般的开发流程是：

先整体框架的设计，分解功能模块，再进行核心或难点的demo实验，最后功能的整合。

对于这个项目来看，有2种方法设计：

第一种：

1. 客户端向服务器发起连接，服务器等待用户输入的请求。

2. 客户端用户输入请求字符串，发送给服务器，服务器收到后开始解析。

3. 服务器根据解析的结果，进行相关的数据处理（打开文件，下载或上传或得到文件列表）。

4. 客户端接收到数据后，处理（显示文件列表，保存文件）。

5. 服务器等待下一次的连接。

第二种：

1. 客户端等待用户输入；

2. 解析用户输入后，判断是下载、上传、获取列表的哪一个功能，进入处理函数代码执行；

3. 处理函数中，连接服务器，发送请求，处理响应；

4. 关闭服务器连接，重复执行第1步；

第一种适合做并发服务器，整体程序稍微复杂，不适合小业务的开发。

我们采用第二种方案

* + 1. 客户端模型

def file\_client():

print("==== 文件传输客户端 ====")

server\_info = ('172.16.80.130', 16688)

while True:

command = input('<client>:')

if command[0] == 'G':

process\_get(server\_info, command)

elif command[0] == 'P':

process\_put(server\_info, command)

elif command[0] == 'L':

process\_list(server\_info, command)

else:

break

* + 1. 服务端模型

def file\_server():

listen\_sock = init\_tcp\_server(16688)

# 服务器主循环

while True:

new\_sock, \_ = listen\_sock.accept()

print("获取新连接", new\_sock)

rheader = new\_sock.recv(1024)

# 解析请求头

if rheader[0:1] == b'G':

down\_process(new\_sock, rheader)

elif rheader[0:1] == b'P':

put\_process(new\_sock, rheader)

elif rheader[0:1] == b'L':

list\_process(new\_sock, rheader)

new\_sock.close()

1. pickle序列化
   1. 为什么需要序列化

在实际开发中，程序员处理的是对象化的数据，比如列表，字典这些，这些python提供的数据类型，大大加快了开发的效率，但当程序退出后，这些数据结构随之消失，为了下次再进行处理，如果又重新生成这些对象，势必造成开发效率的低下。

比如，在机器学习中，我们常常需要把训练好的模型存储起来，这样在进行决策时直接将模型读出，而不需要重新训练模型，这样就大大节约了时间。

这个从程序员可操作的对象转化到计算机可以保存的二进制内容的过程，就称之为序列化。

* 1. pickle序列化的特点

python提供的序列化方法比较多，这里我们介绍一个叫做pickle的模块来完成序列化的过程。

pickle使用的数据格式是特定于Python的。这使得它不受诸如JSON或XDR的外部标准限值，但是这也意味着非Python程序可能无法重建pickled Python对象。默认情况下，pickle数据格式使用相对紧凑的二进制表示。如果需要最佳大小特征，可以有效的压缩pickled数据。pickletools模块包含可以用于对pickle生成的数据流进行分析的工具。目前有5种不同的协议可以用于pickle。使用的协议越高，就需要更新的Python版本去读取pickle产生的数据：

协议v0是原始的“人类可读”协议，并且向后兼容早期的Python版本；

协议v1是一个旧的二进制格式，也与早期版本的Python兼容；

协议v2在Python 2.3中引入，它提供更高效的pickling；

协议v3是在Python 3.0添加的协议，它明确支持bytes对象，且不能被Python 2.x 进行unpickle操作；这是默认协议，也是当需要兼容其他Python 3版本时被推荐使用的协议；

协议4是在Python 3.4添加的协议，它添加了对极大对象的支持，pickling更多种类的对象，以及一些数据格式的优化。

* 1. pickle模块提供的方法

# 将指定的Python对象通过pickle序列化作为bytes对象返回，而不是将其写入文件

dumps(obj, protocol=None, \*, fix\_imports=True)

# 将通过pickle序列化后得到的字节对象进行反序列化，转换为Python对象并返回

loads(bytes\_object, \*, fix\_imports=True, encoding="ASCII", errors="strict")

# 将指定的Python对象通过pickle序列化后写入打开的文件对象中，等价于`Pickler(file, protocol).dump(obj)`

dump(obj, file, protocol=None, \*, fix\_imports=True)

# 从打开的文件对象中读取pickled对象表现形式并返回通过pickle反序列化后得到的Python对象

load(file, \*, fix\_imports=True, encoding="ASCII", errors="strict")

* 1. 举例

>>> import pickle

>>>

>>> var\_a = {'a':'str', 'c': True, 'e': 10, 'b': 11.1, 'd': None, 'f': [1, 2, 3], 'g':(4, 5, 6)}

# 序列化

>>> var\_b = pickle.dumps(var\_a)

>>> var\_b

b'\x80\x03}q\x00(X\x01\x00\x00\x00eq\x01K\nX\x01\x00\x00\x00aq\x02X\x03\x00\x00\x00strq\x03X\x01\x00\x00\x00fq\x04]q\x05(K\x01K\x02K\x03eX\x01\x00\x00\x00gq\x06K\x04K\x05K\x06\x87q\x07X\x01\x00\x00\x00bq\x08G@&333333X\x01\x00\x00\x00cq\t\x88X\x01\x00\x00\x00dq\nNu.'

# 反序列化

>>> var\_c = pickle.loads(var\_b)

>>> var\_c

{'e': 10, 'a': 'str', 'f': [1, 2, 3], 'g': (4, 5, 6), 'b': 11.1, 'c': True, 'd': None}

* 1. 自定义结构的序列化

首先，自定义一个数据类型：

class Student(object):

def \_\_init\_\_(self, name, age, sno):

self.name = name

self.age = age

self.sno = sno

def \_\_repr\_\_(self):

return 'Student [name: %s, age: %d, sno: %d]' % (self.name, self.age, self.sno)

pickle模块可以直接对自定数据类型进行序列化/反序列化操作，无需编写额外的处理函数或类。

>>> stu = Student('Tom', 19, 1)

>>> print(stu)

Student [name: Tom, age: 19, sno: 1]

# 序列化

>>> var\_b = pickle.dumps(stu)

>>> var\_b

b'\x80\x03c\_\_main\_\_\nStudent\nq\x00)\x81q\x01}q\x02(X\x04\x00\x00\x00nameq\x03X\x03\x00\x00\x00Tomq\x04X\x03\x00\x00\x00ageq\x05K\x13X\x03\x00\x00\x00snoq\x06K\x01ub.'

# 反序列化

>>> var\_c = pickle.loads(var\_b)

>>> var\_c

Student [name: Tom, age: 19, sno: 1]

# 持久化到文件

>>> with open('pickle.txt', 'wb') as f:

... pickle.dump(stu, f)

...

# 从文件总读取数据

>>> with open('pickle.txt', 'rb') as f:

... pickle.load(f)

...

Student [name: Tom, age: 19, sno: 1]

1. 课程总结
   1. 重点
      * 1. 基于数据流的数据发送和请求的特点。
        2. 文件的读写操作，网络数据流的读写操作。
        3. 如何序列化一个列表。
   2. 难点
      * 1. 文件上传，数据粘包的问题及解决方案。
   3. 如何掌握？
      * 1. 画流程图
        2. 抄写代码
   4. 排错技巧（技巧）
      * 1. 熟练使用print跟踪打印
        2. 打印数据大小等信息来判断
2. 课后练习
   * + 1. 重新编写简单文件上传下载功能
3. 面试题
   * + 1. TCP和UDP的区别
       2. TCP/IP协议的分层结构
4. 扩展知识或课外阅读推荐（可选）