第5章

机制与协议

本章的主要内容是机制与协议及其实现。这些机制包括用户认证、安全、状态标识等机制，是爬虫工程师必须熟练掌握的内容，只有熟练理解这些安全机制，才能更好地分析网站的保护机制。这些协议包括互联网通信的 TCP/UDP 协议、HTTP/HTTPS 协议、网站跟爬虫间的 Robots 协议、常用的收发邮件的 SMTP 和 IMAP 协议，只有熟练掌握了通信协议，才能更好地分析目标网站的数据交互过程。

本章要点如下。

(1) TCP/IP 协议簇组成及参考模型。

(2) socket 通信过程及 Python 实现。

(3) UDP 协议的实现与运用。

(4) TCP 协议的实现与应用。

(5) HTTP 协议的内容。

(6) WebSocket 协议的内容及 WebSocket 爬虫。

(7) SMTP、IMAP 协议内容及 Python 实现。

(8) 常用安全机制 CSRF、Cookie、session、Token。

5.1 TCP/IP 协议簇

5.1.1 互联网协议套件

TCP/IP 协议簇(Internet Protocol Suite，互联网协议套件)简称 TCP/IP，是一个网络通信模型，代表整个网络传输协议家族，是国际网络的基础通信架构。该协议簇以最早通过的两个标准 TCP(传输控制协议)和 IP(网际协议)命名，该协议簇还包含 FTP、SMTP、TCP、UDP、IP 等协议。由于网络通信协议普遍采用分层的结构，当多个层次的协议共同工作时，类似计算机科学中的堆栈，因此该协议簇又被称为 TCP/IP 协议栈(TCP/IP Protocol Stack)。这些协议最早发源于美国国防部(缩写为 DoD)的 ARPA 网络项目，因此也被称作 DoD 模型(DoD Model)，这个协议簇由互联网工程任务组负责维护。

TCP/IP 提供了点对点链接的机制，将数据如何封装、定址、传输、路由以及在目的地如何接收都加以标准化。它将软件通信过程抽象化为四个抽象层，采取协议堆栈的方式，分别实现不同通信间的协议。协议簇下的各种协议，依其功能不同，被分别归属到四个层次结构之中，常被视为简化的七层 OSI 模型。

5.1.2 TCP/IP 协议簇的组成

按照整个通信网络的功能，可以将其划分成不同的功能层级，用于互联网的协议可以比照 TCP/IP 参考模型进行分类。TCP/IP 协议栈起始于第三层协议 IP(网际协议)，所有这些协议都在相应的 RFC 文档中标准化，RFC 文档标记了这些协议的状态，分别是必须(required)、推荐(recommended)、可选(selective)、试验(experimental)、历史(historic)状态。

必须协议指所有的 TCP/IP 应用都必须实现 IP 协议和 ICMP 协议。ICMP 协议主要用于收集有关网络的信息查找错误等工作。对于一个路由器而言，有这两个协议就可以运作，但是实际的路由器一般还需要运行许多推荐使用的协议，以及一些其他的协议。下面介绍常用协议应用场景及定义。

1. 路由器涉及的协议

地址解析协议(Address Resolution Protocol，ARP)是一个通过解析网络层地址来找寻数据链路层地址的网络传输协议。

网际协议(Internet Protocol，IP)也称互联网协议，是用于分组交换数据网络的一种协议。

互联网控制消息协议(Internet Control Message Protocol，ICMP)是互联网协议簇的核心协议之一，它用于在网际协议中发送控制消息，为可能发生在通信环境中的各种问题提供反馈。

用户数据报协议(User Datagram Protocol，UDP)又称用户数据包协议，是一个简单的面向数据报的通信协议，位于 OSI 模型的传输层。

简单网络管理协议(Simple Network Management Protocol，SNMP)构成了互联网工程工作小组(Internet Engineering Task Force，IETF)定义的 Internet 协议簇的一部分，该协议能够支持网络管理系统，监测连接到网络上的设备是否有异常情况。

路由信息协议(Routing Information Protocol，RIP)是一种内部网关协议(IGP)，是最早出现的距离向量路由协议，属于网络层，其主要应用于规模较小的、可靠性要求较低的网络，可以通过不断地交换信息让路由器动态适应网络连接的变化，这些信息包括每个路由器可以到达哪些网络，这些网络有多远等。

2. 万维网用户涉及的协议

万维网用户使用的协议除了上述路由器涉及的 ARP、IP、ICMP、UDP 协议之外，还有下列的一些协议。

传输控制协议(Transmission Control Protocol，TCP)是一种面向连接的、可靠的、基于字节流的传输层通信协议，由 IETF 的 RFC 793 定义。在简化 OSI 模型中，它完成第四层传输层所指定的功能。

域名系统(Domain Name System，DNS)是互联网的一项服务。它是域名和 IP 地址相互映射的一个分布式数据库，DNS 使用 TCP 和 UDP 端口 53。它对于每一级域名长度的限制是 63 个字符，域名总长度不能超过 253 个字符。

超文本传输协议(Hyper Text Transfer Protocol，HTTP)是一种用于分布式、协作式和超媒体信息系统的应用层协议。HTTP 是万维网的数据通信的基础。

文件传输协议(File Transfer Protocol，FTP)是一个在计算机网络上用于在客户端和服

务器之间进行文件传输的应用层协议。

3. 用户计算机涉及的协议

计算机作为终端，除了使用上述必要的协议外还有如下一些协议。

TELNET 是一种应用层协议，在互联网及局域网中使用。它使用虚拟终端的形式，提供双向、以文字字符串为主的命令行接口交互功能，是互联网远程登录服务器的标准协议和主要方式，常用于服务器的远程控制。

简单邮件传输协议(Simple Mail Transfer Protocol，SMTP)是一个在互联网上传输电子邮件的标准。SMTP 使用 TCP 端口 25，要为一个给定的域名启用 SMTP 服务器，需要使用 DNS 的 MX 解析记录。

邮局协议(Post Office Protocol，POP)是 TCP/IP 协议簇中的一员，由 RFC 1939 定义。此协议主要用于支持使用客户端远程管理在服务器上的电子邮件。最新版本为 POP3(Post Office Protocol-Version3)，提供了 SSL 加密的 POP3 协议被称为 POP3S。

动态主机设置协议(Dynamic Host Configuration Protocol，DHCP)是一个用于局域网的网络协议，位于 OSI 模型的应用层，使用 UDP 协议工作，主要用于内部网或网络服务供应商自动分配 IP 地址给用户及内部网管理员对所有计算机的中央控制管理。

安全外壳协议(Secure Shell，SSH)是一种加密的网络传输协议，可在不安全的网络中为网络服务提供安全的传输环境。SSH 通过在网络中创建安全隧道来实现 SSH 客户端与服务器端之间的连接。

网络新闻传输协议(Network News Transport Protocol，NNTP)是一个主要用于阅读和发布新闻文章到 Usenet 上的应用协议，也负责新闻在服务器间的传送。Usenet 是一种分布式的互联网交流系统。

5.1.3　TCP/IP 参考模型

TCP/IP 参考模型是一个抽象的分层模型，在这个模型中，所有的 TCP/IP 系列网络协议都被归类到 4 个抽象的层中，分别是应用层、传输层、网络互联层、网络访问(链接)层，如图 5-1 所示。每个抽象层创建在低一层提供的服务上，并且为高一层提供服务。完成一些特定的任务需要众多的协议协同工作，这些协议分布在参考模型的不同层中的，因此有时称它们为一个协议栈。

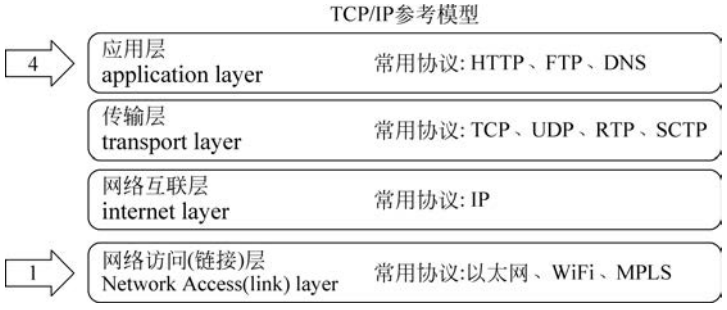

图 5-1　TCP/IP 参考模型

OSI 模型(Open System Interconnection Model，开放式系统互联模型)是一种概念模型，由国际标准化组织提出，是一个试图使各种计算机在世界范围内互联为网络的标准框架。OSI 将计算机网络体系结构划分为以下七层，分别是物理层、数据链路层、网络层、传输

层、会话层、表示层、应用层，如图 5-2 所示。

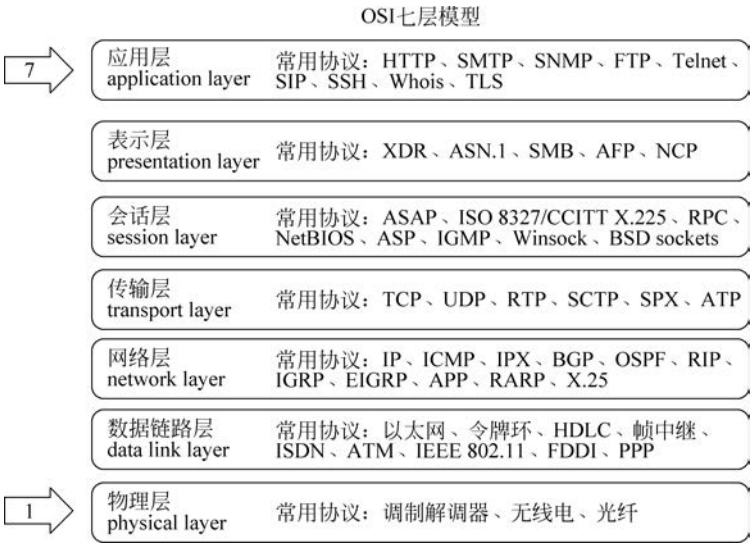

图 5-2　OSI 七层模型示意图及常用协议

OSI 模型和 TCP/IP 参考模型并非是完全对应的，在传输层和网络层之间还需要另外一个层（网络互联层）。特定网络类型专用的一些协议本应运行在网络层上，却运行在基本的硬件帧交换上，比如地址解析协议和生成树协议运行在网络互联功能下。

通常认为 OSI 模型的最上面三层（应用层、表示层和会话层）在 TCP/IP 参考模型中是一个应用层。由于 TCP/IP 有一个相对较弱的会话层，其由 TCP 和 RTP 下的打开和关闭连接组成，并且为在 TCP 和 UDP 下的各种应用提供不同的端口号，这些功能能够被单个的应用程序（或者那些应用程序所使用的库）添加。与此相似的是，IP 是按照将它下面的网络当作一个黑盒子的思想设计的，这样在讨论 TCP/IP 的时候就可以把它当作一个独立的层。

应用层负责处理所有和应用程序协同的工作，利用基础网络交换应用程序专用的数据协议。

传输层的协议能够解决诸如端到端可靠性和保证数据按照正确的顺序到达的问题。在 TCP/IP 协议组中，传输协议也包括所给数据应该被送给哪个应用程序。

网络互联层主要解决主机到主机的通信问题。它所包含的协议设计数据包含在整个网络上的逻辑传输中。

链接层实际上并不是因特网协议组中的一部分，但是它是数据包从一个设备的网络层传输到另外一个设备的网络层的方案。

5.2　TCP 与 UDP 协议

5.2.1　socket 通信

网络套接字（Network Socket）又称网络接口、网络插槽，在计算机科学中是计算机网络中进程间数据流的端点。使用以网际协议（Internet Protocol）为通信基础的网络套接字，被称为网际套接字（Internet Socket）。socket 是一种操作系统提供的进程间通信机制，往往是基于不同主机之间的进程通信，socket 既可以发送消息到指定地址客户端的端口，也可以监听本机的指定端口和接收消息。

1. 用 socket 发送消息

Python 内置 socket 模块，可以通过 import socket 导入。下面将介绍 socket 模块的基本使用方法，以及演示 socket 的通信过程，这是网络的基础，是信息传递的核心过程。

使用 socket 之前需要使用 socket 模块的 socket 方法创建一个 socket 描述符。

```
import socket
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
```

AF_INET 用于指明是在 Internet 进程间通信，还有另一个参数 AF_UNIX 用于同一台机器进程间通信，一般常用 AF_INET。第二个参数 SOCK_STREAM 用于指示套接字类型，SOCK_STREAM 是流式套接字，常用于 TCP 协议。SOCK_DGRAM 是数据包套接字，主要用于 UDP 协议。

创建 socket 描述符是第一步，不管是 socket 客户端或者服务器端，都需要先创建 socket 描述符。第二步是确定目标客户端的 IP 地址和监听端口，将 IP 地址和通信端口按照元组形式传递给 socket 客户端作为连接目标。

一个完整的 socket 客户端发送消息的流程如下。

```
import socket

sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
address = ("192.168.0.103", 5555)
sock.sendto(bytes("Hello World", 'utf-8'), address)    #发送消息
backdata = sock.recvfrom(1024)                         #获取反馈消息：堵塞
print(backdata)                                        #打印反馈消息
```

上面实现的是 UDP 协议的套接字，通过 sendto 方法把经过 bytes 编码的信息发送到指定的服务器端，发送之后 sock.recvfrom 方法将等待回传信息，并且限定接收信息的大小是 1024 字节。下面使用网络调试助手 NetAssist 作客户端，用上面的脚本发送 Hello World，并且使用 NetAssist 回应“你好”。

打开 NetAssist 客户端，在左侧的【协议类型】下拉菜单中选择 UDP 选项，在【本地主机地址】下拉菜单中选择代码中发送的目的地址，在【本地主机端口】文本框中填入代码中的 5555，然后单击【打开】按钮，效果如图 5-3 所示。

图 5-3 对网络调试助手 NetAssist 进行设置

正确设置 NetAssist 之后，运行上面的脚本，观察 NetAssist 数据日志窗口，收到的信息如下。

```
[2020-03-10 17:51:40.174]          #RECV ASCII FROM 192.168.0.103 :60160>
Hello World
```

192.168.0.103:60160 是脚本发送信息使用的 IP 地址和端口，此时运行的脚本 sock.

recvfrom(1024)这一行,这是在等待客户端的响应。接着在 NetAssist 的【数据发送】输出框中输入要回应的信息“你好”,单击【发送】按钮,可观察到本地脚本结束堵塞,打印出如下信息。

```
(b'\xc4\xe3\xba\xc3', ('192.168.0.103', 5555))
```

收到的元组中第一项是数据,其经过了 bytes 编码;第二项是发送端的地址和端口。这就是 UDP 协议通信的过程,UDP 发送消息之前不是先建立连接,而是直接发送消息。而 TCP 客户端在发送消息之前需要先连接,TCP 客户端的源码如下。

```
import socket

client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
address = ("192.168.0.103", 5555)
client.connect(address) #先连接到服务器
client.sendto(bytes("Hello World", 'utf-8'), address)
backdata = client.recvfrom(1024)
print(backdata)
```

同时在 NetAssist 左侧的【协议类型】下拉菜单中选择 TCP Server 选项,然后单击【打开】按钮,接着按照 UDP 流程运行脚本并在 NetAssist 中回复“你好”,观察 NetAssist 数据日志和 Python 脚本的输出信息。

NetAssist 的数据日志打印信息如下。

```
[2020-03-10 18:05:58.697]          #Client 192.168.0.103:51634 gets online.
[2020-03-10 18:05:58.705]          #RECV ASCII FROM 192.168.0.103 :51634>
Hello World
[2020-03-10 18:06:10.080]          #SEND ASCII TO ALL>
你好
[2020-03-10 18:06:10.092]          #Client 192.168.0.103:51634 gets offline.
```

可以看到先建立了一个 Client 的客户端连接,然后收到了信息:Hello World,再发送数据“你好”,最后通知断开 TCP 连接,TCP 客户端脚本收到如下信息。

```
(b'\xc4\xe3\xba\xc3', (0, b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'))
```

从上面案例体验了 UDP、TCP 协议的一些差异,UDP 协议是无状态的,发送之前不需要先建立连接;而 TCP 协议发送数据之前先建立连接,发送之后再关闭连接。

UDP 协议传递数据就像是发送邮件,发送出去后不管对方是否收到。TCP 协议传递数据就像打电话,首先拨号然后先问候一下,说完事情最后再告别。

2. 用 socket 监听消息

下面分别使用 UDP 和 TCP 模式来实现用 socket 监听指定端口的消息。对于 UDP 模式而言,监听本地消息并不复杂,直接绑定本地要监听的地址和端口即可,其源码如下。

```
import socket

client = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
address = ("", 5556)                #监听本地的 IP 和端口,IP 为空则监听所有 IP
client.bind(address)
backdata = client.recvfrom(1024)    #堵塞等待消息
print(backdata)
client.close()
```

当运行此脚本，会堵塞在 client.recvfrom(1024)，当收到消息时进入下一步。打开 NetAssist 在【数据发送】右侧的【远程主机】文本框中填写需要 UDP 监听的 IP 地址和端口号，然后输入一条信息，单击【发送】按钮，如图 5-4 所示。此时处于监听状态的 UDP 脚本会收到该条信息，并打印出来，消息如下，消息含有发送的数据和发送端的地址和 IP 端口号。

```
(b'\xc4\xe3\xba\xc3', ('192.168.0.103', 5556))
```

图 5-4　设置 UDP 监听的 IP 地址和端口

TCP 的服务器端除了绑定监听的端口后信息之外，还需要使用 listen(backlog)参数启动服务，允许 TCP 客户端连接到此服务器，backlog 指正在处理一个请求时，允许连接的新请求数量。设置 listen 后使用 socket.accept 监听绑定的端口，如果有新连接将返回一个建立了 TCP 连接的 socket 对象和 TCP 客户端 IP 及端口的元组，实现代码如下：

```
import socket

server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
address = ("", 5556)
server.bind(address)                          #先连接到服务器
server.listen(10)                             #同时允许存在 10 个 TCP 连接
newclient, clientaddress = server.accept()    #监听连接默认为堵塞式,可设置为非堵塞
print(newclient, clientaddress)               #打印 newclient, clientaddress
data = newclient.recv(1024)
print(f"收到数据:{data}")
newclient.send(bytes("Hello World", 'utf-8'))
newclient.close()                             #关闭与客户端的连接
server.close()                                #关闭 server 服务
```

返回的 newclient 是连接了 TCP 客户端的 socket 连接，所以可以直接发送消息，同时不使用的时候需要关闭它。newclient 是每个客户端建立的 TCP 连接，每个 TCP 连接请求都会返回一个 socket 连接对象，也从底层解释了并发流量为什么会影响到业务的处理，因为大量的 TCP 客户端建立连接将是非常耗费资源的一件事情。DDOS 攻击正是基于此原理，异常流量占据服务器连接资源，会导致正常访问无法建立连接，无法得到响应。

当运行上述源码后 server.accept()会处于堵塞状态，当有新连接建立时会结束堵塞状态，返回新的 TCP 连接和客户端的基本信息，针对该客户端的操作都是基于 newclient 对象来处理的。在 NetAssist 中的【协议类型】下拉菜单中选择 TCP Client 选项，在【远程主机端口】文本框中输入 TCP 服务器监听的端口 5556，然后单击【连接】按钮。如图 5-5 所示，建立连接后，服务器端会结束监听返回 newclient，clientaddress，然后堵塞在 newclient.recv(1024)，即等待客户端发送消息。在 NetAssist 的【数据发送】文本框中输入信息并单击【发送】按钮，服务器端会收到消息并打印出来，然后发送一条消息给客户端，最后关闭与 TCP 客户端的连接，停止监听端口并结束服务，至此就完成了一次 TCP 的通信过程。

图 5-5 模拟 TCP 客户端发送请求

TCP 服务器端脚本打印的信息如下。

```
< socket.socket fd = 480, family = AddressFamily.AF_INET, type = socketKind.SOCK_STREAM, proto
 = 0, laddr = ('192.168.0.103', 5556), raddr = ('192.168.0.103', 63603)>

收到数据:b'\xc4\xe3\xba\xc3'
```

对于 Python 而言,直接使用 socket 的场景很少,因为 Python 中有太多的库提供了开箱即用的功能,比如著名的网络请求库 Requests。深入底层和细节的目的在于探究本源,只有在了解底层流程之后才能理解表象背后的逻辑,比如为什么爬虫速度过快会影响网站的运行?那是因为在多线程、高并发情况下目标站点服务器的连接资源被占用,导致正常用户访问困难,影响了目标站点的运营,这已经违反了相关规定,属于危害计算机系统的行为。

5.2.2 UDP 协议

UDP(User Datagram Protocol,用户数据报协议)又称用户数据包协议,是一个简单的面向数据报的通信协议,位于 OSI 模型的传输层。

在 TCP/IP 模型中,UDP 为网络层以上和应用层以下提供了一个简单的接口。UDP 只提供数据的不可靠传递,它一旦把应用程序发给网络层的数据发送出去,就不保留数据备份,UDP 在 IP 数据报的头部仅仅加了复用和数据校验字段的操作,所以 UDP 有时候也被认为是不可靠的数据报协议。

UDP 适用于不需要在程序中执行错误检查和纠正的应用,它避免了协议栈中此类处理的开销。对时间有较高要求的应用程序通常使用 UDP,因为丢弃数据包比等待或重传导致延迟更可取。

1. 可靠性

由于 UDP 缺乏可靠性且属于无连接协议,所以应用程序通常必须允许一些丢失、错误或重复的数据包。某些应用程序(如 TFTP)可能会根据需要在应用程序层中添加基本的可靠性机制。

一些应用程序不太需要可靠性机制,甚至可能因为引入可靠性机制而降低性能,所以它们会使用 UDP 这种缺乏可靠性的协议。流媒体、实时多人游戏和 IP 语音(VoIP)是经常使用 UDP 的应用程序。

在 VoIP 中延迟和抖动是主要问题。如果使用 TCP,那么任何数据包的丢失或错误都将导致抖动,因为 TCP 在请求及重传丢失数据时不会向应用程序提供后续数据。如果使用 UDP,那么应用程序则需要提供必要的握手,例如实时确认已收到的消息。

由于 UDP 缺乏拥塞控制，所以需要基于网络的机制来减少因失控和高速 UDP 流量负荷而导致的拥塞崩溃效应。因为 UDP 发送端无法检测拥塞，所以像使用包队列和丢弃技术的路由器等网络基础设备会被用于降低 UDP 的过大流量。

2. UDP 的应用

许多关键的互联网应用程序都使用 UDP。

(1) 域名系统(DNS)，其中查询阶段必须快速，并且只包含单个请求，后跟单个回复数据包。

(2) 动态主机配置协议(DHCP)，用于动态分配 IP 地址。

(3) 简单网络管理协议(SNMP)。

(4) 路由信息协议(RIP)。

(5) 网络时间协议(NTP)。

音频、视频、在线游戏流量通常使用 UDP 传输。实时视频流和音频流应用程序旨在处理偶尔丢失、错误的数据包，因此只会发生轻微的质量下降，同时避免了重传数据包带来的高延迟。

5.2.3 TCP 协议

TCP(Transmission Control Protocol，传输控制协议)是一种面向连接的、可靠的、基于字节流的传输层通信协议，由 IETF 的 RFC 793 定义。在简化的计算机网络 OSI 模型中，它完成第四层传输层所指定的功能，而用户数据报协议(UDP)是同一层内另一个重要的传输协议。

在因特网协议簇(Internet Protocol Suite)中，TCP 层是位于 IP 层之上，应用层之下的中间层。不同主机的应用层之间经常需要可靠的、像管道一样的连接，但是 IP 层不提供这样的流机制，而是提供不可靠的包交换。

应用层向 TCP 层发送用于网间传输的每一字节通常由 8 位二进制数组成，然后 TCP 把数据流分割成适当长度的报文段(长度通常受该计算机连接的网络的数据链路层的最大传输单元(MTU)的限制)。之后 TCP 把结果包传给 IP 层，由它来通过网络将包传送给接收端实体的 TCP 层。TCP 为了保证不发生丢包，就给每个包一个序号，同时序号也保证了传送到接收端实体的包能按序接收。然后接收端实体对已成功收到的包发回一个相应的确认信息(ACK)；如果发送端实体在合理的往返时延(RTT)内未收到确认，那么对应的数据包就被假设为已丢失并会进行重传。TCP 用一个校验和函数来检验数据是否有错误，在发送和接收时都要计算和校验。

TCP 协议的运行可分为三个阶段：连接创建(Connection Establishment)、数据传送(Data Transfer)和连接终止(Connection Termination)。操作系统将 TCP 连接抽象为套接字表示的本地端点(Localend-Point)，作为编程接口给程序使用。在 TCP 连接的生命期内，本地端点要经历一系列的状态改变。

1. 可靠传输

通常在每个 TCP 报文段中都有一对序号和确认号。TCP 报文发送者称自己的字节流的编号为序号(Sequence Number)，称接收到的对方的字节流编号为确认号。TCP 报文的接收者为了确保可靠性，在接收到一定数量的连续字节流后才发送确认。这是对 TCP 的一种扩展，被称为选择确认(Selective Acknowledgement)。选择确认使得 TCP 接收者可以对

乱序到达的数据块进行确认。每一个字节传输过后，SN 号都会递增 1。

通过使用序号和确认号，TCP 层可以把收到的报文段中的字节按正确的顺序交付给应用层。序号是 32 位的无符号数，当它增大到 $2^{32}-1$ 时，便会回到 0。对于初始化序列号(ISN)的选择是 TCP 中一个关键的操作，它可以确保强壮性和安全性。

TCP 协议使用序号标识每端发出的字节的顺序，从而另一端接收数据时可以重建顺序，无惧传输时的包的乱序交付或丢包。在发送第一个包时(SYN 包)，TCP 协议会选择一个随机数作为序号的初值，以克制 TCP 序号预测攻击。

发送的确认包(Acks)携带了接收到的对方发来的字节流的编号，该编号被称为确认号，以告诉对方已经成功接收的数据流的字节位置。Ack 并不意味着数据已经交付到了上层应用程序。

可靠性通过发送方检测到丢失的传输数据并重传这些数据。包括超时重传(Retransmission Time Out，RTO)与重复累计确认(Duplicate Cumulative Acknowledgements，DupAcks)。

2. 应用场景

TCP 并不是对所有的应用都适合，因为一些新的带有一些内在的脆弱性的运输层协议也会被设计出来。比如，实时应用并不需要甚至无法忍受 TCP 的可靠传输机制。在这种类型的应用中，通常允许出现一些丢包、出错或拥塞的问题，而不是去校正它们。例如通常不使用 TCP 的应用有流媒体、实时多媒体播放器、游戏、IP 电话(VoIP)等。任何不是很需要可靠性或者是想将功能减到最少的应用可以避免使用 TCP。在很多情况下，当只需要多路复用应用服务时，用户数据报协议(UDP)可以代替 TCP 为应用提供服务。

5.2.4 TCP 的三次握手

TCP 的三次握手是 TCP 的工作流程创建连接—数据送达—连接终止中的第一步，即创建连接。

TCP 用三次握手(Three-Way Handshake)创建一个连接。在连接创建过程中，很多参数要被初始化，如序号被初始化以保证按序传输和连接的强壮性。

三次握手的开始，通常由一端打开一个套接字(socket)然后监听来自另一方的连接，这就是通常所指的被动打开。服务器端被动打开以后，客户端就能开始创建主动打开。

客户端通过向服务器端发送一个 SYN 来创建一个主动打开，作为三次握手的一部分，客户端把这段连接的序号设定为随机数 A。

服务器端应当为一个合法的 SYN 回送一个 SYN/ACK，ACK 的确认码应为 A+1，SYN/ACK 包本身又有一个随机产生的序号 B。

最后，客户端再发送一个 ACK。此时包的序号被设定为 A+1，而 ACK 的确认码则为 B+1。当服务器端收到这个 ACK 的时候，就完成了三次握手，并进入了连接创建状态。

如果服务器端接到了客户端发的 SYN 且回了 SYN-ACK 后客户端掉线了，服务器端就没有收到客户端回的 ACK，那么这个连接处于一个中间状态，既没成功，也没失败。于是，在一定时间内没有收到 ACK 的服务器端会重发 SYN-ACK。在 Linux 下默认重试次数为 5 次，重试的间隔时间从 1s 开始，每次都翻倍，重试的 5 次时间间隔为 1s、2s、4s、8s、16s，总共 31s，第 5 次发出后还要等 32s 才知道第 5 次也超时了，所以总共需要 1s+2s+4s+8s+16s+32s=63s，TCP 才会断开这个连接。

图 5-6 即为 TCP 的三次握手示意图。

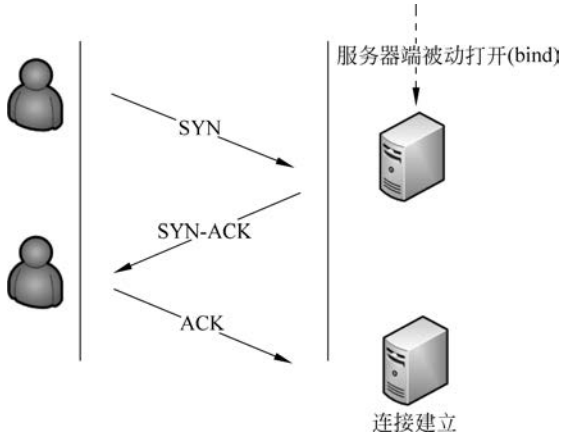

图 5-6 TCP 创建连接前的三次握手

5.2.5 TCP 的四次挥手

TCP 的四次挥手是 TCP 的工作流程创建连接-数据送达-连接终止的最后一步，即终止。连接终止使用了四路握手过程(Four-Way Handshake)，也称四次握手。在这个过程中，连接的每一侧都独立地被终止。当一个端点要停止它这一侧的连接，就向对侧发送 FIN，对侧回复 ACK 表示确认。因此，拆掉一侧的连接过程需要一对 FIN(finish，结束)和 ACK，其分别由两侧端点发出。

首先发出 FIN 的一侧，如果给对侧的 FIN 响应了 ACK，那么就会超时等待 2×MSL 的时间，然后关闭连接。在这段超时等待时间内，本地的端口不能被新连接使用，避免延时的包的到达与随后的新连接相混淆。RFC 793 定义了 MSL 为 2min，在 Linux 下 MSL 设置成了 30s。

连接可以在 TCP 半开状态下工作。即一侧关闭了连接，不再发送数据，但另一侧没有关闭连接，仍可以发送数据，已关闭的一侧仍然应接收数据，直至对侧也关闭连接。

也可以通过三次握手关闭连接：主机 A 发出 FIN，主机 B 回复 FIN&ACK，然后主机 A 回复 ACK。

图 5-7 即为 TCP 的四次挥手示意图。

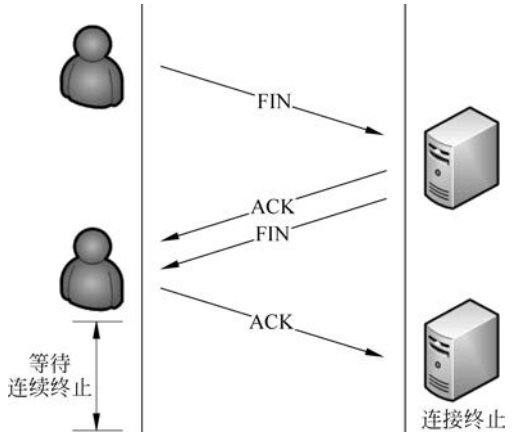

图 5-7 TCP 的四次挥手示意图

5.2.6 TCP 长连接

HTTP 持久连接(HTTP persistent connection,也称 HTTP keep-alive、HTTP connection reuse)是使用同一个 TCP 连接来发送和接收多个 HTTP 请求或应答,而不是为每一个新的请求或应答打开新的连接的方法。

在 HTTP 1.0 中,没有官方的 keep-alive 的操作,通常是在现有协议上添加一个指数,如果浏览器支持 keep-alive,它会在请求的包头中添加如下内容。

```
Connection:keep-Alive
```

当服务器收到请求,作出回应的时候,它也添加一个头在响应中。

```
Connection:keep-alive
```

当客户端发送另一个请求时,它会使用同一个连接,一直持续到客户端或服务器端认为会话已经结束,其中一方中断连接为止。

TCP 长连接拥有如下优势。

(1) 较少的 CPU 和内存的使用(由于同时打开的连接减少了)。

(2) 允许请求和应答的 HTTP 管线化。

(3) 降低拥塞控制(TCP 连接减少了)。

(4) 减少了后续请求的延迟(无须再进行握手)。

(5) 即使报告错误也无须关闭 TCP 连接。

但是 TCP 长连接存在一些劣势:对于单个文件被不断请求的服务(如图片资源),keep-alive 可能会极大地影响性能,因为它在文件被请求之后还保持了不必要的连接很长时间。

5.3 HTTP 与 HTTPS 协议

HTTP 协议与 HTTPS 协议是浏览器常用的协议,也是爬虫工程师最常用的协议。爬虫的核心是模拟用户的浏览器请求数据,这个过程主要是使用 HTTP/HTTPS 协议来和服务器交互。HTTPS 协议是 HTTP 协议的安全版本,它是 HTTP 加 SSL 对 HTTP 的明文传输进行加密。下面将对这两个协议的内容和工作流程做具体的介绍。

5.3.1 HTTP 协议的实现

HTTP(Hyper Text Transfer Protocol,超文本传输协议)是一种用于分布式、协作式和超媒体信息系统的应用层协议,HTTP 是万维网的数据通信的基础。

设计 HTTP 最初的目的是提供一种发布和接收 HTML 页面的方法。通过 HTTP 或者 HTTPS 协议请求的资源由统一资源标识符(Uniform Resource Identifiers,URI)来标识。

HTTP 的发展是蒂姆·伯纳斯-李于 1989 年在欧洲核子研究组织(CERN)发起的。HTTP 的标准制定由万维网协会(World Wide Web Consortium,W3C)和互联网工程任务组(Internet Engineering Task Force,IETF)进行协调,最终发布了一系列的 RFC,其中最著名的是 1999 年 6 月公布的 RFC 2616,定义了 HTTP 协议中现今广泛使用的一个版本——HTTP 1.1。

2014 年 12 月，互联网工程任务组(IETF)的 Hypertext Transfer Protocol Bis(httpbis)工作小组将 HTTP/2 标准提议递交至 IESG 进行讨论，该提议于 2015 年 2 月 17 日被批准。HTTP/2 标准于 2015 年 5 月以 RFC 7540 正式发表，取代 HTTP 1.1 成为 HTTP 的实现标准。

1. 协议概述

HTTP 是一个客户端(用户)和服务器端(网站)之间请求和应答的标准，通常使用 TCP 协议。通过使用网页浏览器、网络爬虫或者其他的工具，客户端发起一个 HTTP 请求到服务器上的指定端口(默认端口为 80)，这个客户端被称为用户代理程序(User Agent)。应答的服务器上存储着一些资源，比如 HTML 文件和图像，这个应答服务器被称为源服务器(Origin Server)。在用户代理程序和源服务器中间可能存在多个“中间层”，比如代理服务器、网关或者隧道(Tunnel)。

尽管 TCP/IP 协议是互联网上最流行的应用，但是在 HTTP 协议中并没有规定它必须使用或它支持的层，事实上 HTTP 可以在任何互联网协议或其他网络上实现。HTTP 假定其下层协议能提供可靠的传输，因此任何能够提供这种保证的协议都可以被其使用，所以其在 TCP/IP 协议簇中使用 TCP 作为它的传输层。

通常由 HTTP 客户端发起一个请求，创建一个到服务器指定端口(默认是 80 端口)的 TCP 连接，HTTP 服务器则在那个端口监听客户端的请求。一旦收到请求，服务器会向客户端返回一个状态，比如“HTTP/1.1 200 OK”，以及返回的内容，如请求的文件、错误消息或者其他信息。

2. HTTP 协议的格式

使用 Chrome 浏览器的开发者工具的 Network 面板抓包，观察数据请求的 Request Headers 的 view parsed 视图，会看见如图 5-8 所示的内容。

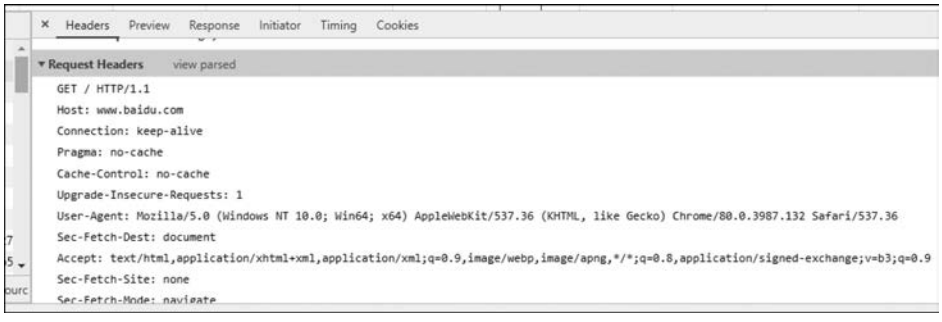

图 5-8　在 Chrome 浏览器中观察 HTTP 协议的格式

这就是 HTTP 协议的格式，首行内容是 GET / HTTP/1.1，表明用的是 GET 请求，请求路径是根路径，使用的是 HTTP 协议的 1.1 版本。第二行开始的键值对形式的信息是请求的请求头，它是关于请求的附加参数。Host、Connection、Pragma、Cache-Control 等附加参数有不同的含义，用来指示客户端的相关状态。请求头信息再往下，会有一行空行，然后是请求体，请求头里面是要发送给服务器的数据。对于服务器的响应请求而言，里面是响应数据。

一个真实的 HTTP 请求和响应报文有着严格的格式要求。其中第一行是请求行，指明请求方法、请求路径以及请求协议和版本。第二行开始到空行的内容是请求头，表示客户端的相关信息。空行之后是请求体，携带要发送给服务器的相关信息。一个请求的

响应也分为四部分，分别是响应行、响应头、空行、响应体。响应的响应行格式区别于请求行，响应行是类似 HTTP/1.1 200 OK 的格式，首先指明协议版本，然后是状态码和附加信息。

```
#HTTP 的请求内容
GET /s?wd = HelloWorld HTTP/1.1
Host: www.baidu.com
Connection: keep - alive
Pragma: no - cache
Cache - Control: no - cache
Upgrade - Insecure - requests: 1
User - Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/80.0.3987.132 Safari/537.36
Sec - Fetch - Dest: document
Accept: text/html,application/xhtml + xml,application/xml;q = 0.9,image/webp,image/apng, * /
* ;q = 0.8,application/signed - exchange;v = b3;q = 0.9
Accept - Encoding: gzip, deflate, br
Accept - Language: zh - CN,zh;q = 0.9,en;q = 0.8

wd = HelloWorld

#HTTP 的响应内容
HTTP/1.1 200 OK
Bdpagetype: 3
Connection: keep - alive
Content - Encoding: gzip
Content - Type: text/html;charset = utf - 8
Date: Wed, 11 Mar 2020 01:36:04 GMT
Server: BWS/1.1
Set - Cookie: delPer = 0; path = /; domain = .baidu.com
domain = .baidu.com
Strict - Transport - Security: max - age = 172800
Traceid: 158389056405468905061188755089349401513 9
Vary: Accept - Encoding

…
```

一个完整的 HTTP 请求报文格式如图 5-9 所示。

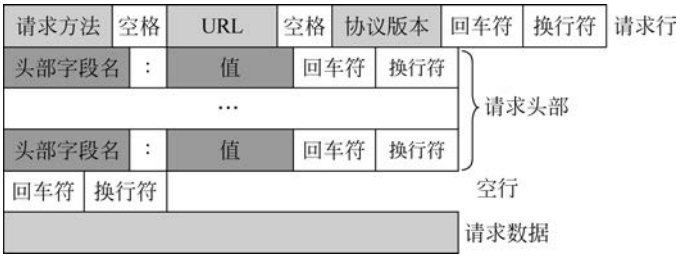

图 5-9　HTTP 请求报文格式

5.3.2　使用 socket 实现 HTTP 协议服务器

HTTP 协议是基于 TCP 协议传输的，也就是通过 TCP 服务器返回网页。如果知道了 HTTP 响应格式，是不是可以使用 socket 编写 HTTP 服务呢？答案是肯定的，下面根据图 5-10 中的 HTTP 响应报文格式来打造一个 HTTP 服务。

图 5-10　HTTP 响应报文格式

这个服务也很简单，只需要返回字符 Hello World。将前面 socket 通信中的 TCP Server 代码简单处理一下，返回 HTTP 协议格式的内容的源码如下。

```
import socket

server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
address = ("", 5556)
server.bind(address)                                    #先连接到服务器
server.listen(10)                                       #同时允许存在 10 个 TCP 连接
newclient, clientaddress = server.accept()              #监听连接默认为堵塞式,可设置为非堵塞
print(newclient, clientaddress)                         #打印 newclient, clientaddress
data = newclient.recv(1024)
print(f"收到数据:{data}")
lines = "HTTP/1.1 200 OK\r\n"
headers = """Content-Type: text/html;charset=utf-8
Date: Wed, 11 Mar 2020 01:36:04 GMT
Server: BWS/1.1
Set-Cookie: delPer=0; path=/;
"""
body = "Hello World"
newclient.send(bytes(lines+headers+'\r\n'+body, 'utf-8'))
newclient.close()                                       #关闭与客户端的连接
server.close()                                          #关闭 server 服务
```

运行此 HTTP 服务器端脚本，然后先在浏览器中打开【开发者工具】的 Network 面板对请求进行记录，输入 localhost:5556 后按 Enter 键访问，页面显示文字 Hello World，如图 5-11 所示。

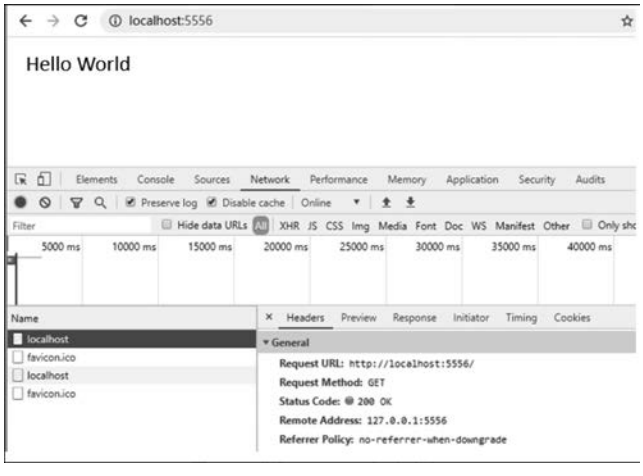

图 5-11　获得 HTTP 服务器端响应

Chrome 浏览器记录的 localhost 请求的 HTTP 报文如下。

```
#请求报文
GET / HTTP/1.1
Host: localhost:5556
Connection: keep-alive
Pragma: no-cache
Cache-Control: no-cache
Upgrade-Insecure-requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/80.0.3987.132 Safari/537.36
Sec-Fetch-Dest: document
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/
*;q=0.8,application/signed-exchange;v=b3;q=0.9
Sec-Fetch-Site: none
Sec-Fetch-Mode: navigate
Sec-Fetch-User: ?1
Accept-Encoding: gzip, deflate, br
Accept-Language: zh-CN,zh;q=0.9,en;q=0.8

#响应报文
HTTP/1.1 200 OK
Content-Type: text/html;charset=utf-8
Date: Wed, 11 Mar 2020 01:36:04 GMT
Server: BWS/1.1
Set-Cookie: delPer=0; path=/;

Hello World
```

HTTP 响应报文中的响应行、响应头、响应体都是在 HTTP 服务源码中设置的信息，这就是 HTTP 服务器的本质。

下面再看服务器端显示的信息。

```
<socket.socket fd=500, family=AddressFamily.AF_INET, type=socketKind.SOCK_STREAM, proto
=0, laddr=('127.0.0.1', 5556), raddr=('127.0.0.1', 55598)> ('127.0.0.1', 55598)

#收到数据
b'GET / HTTP/1.1\r\n
Host: localhost:5556\r\n
Connection: keep-alive\r\n
Pragma: no-cache\r\n
Cache-Control: no-cache\r\n
Upgrade-Insecure-requests: 1\r\n
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/80.0.3987.132 Safari/537.36\r\n
Sec-Fetch-Dest: document\r\n
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/
*;q=0.8,application/signed-exchange;v=b3;q=0.9\r\nSec-Fetch-Site: none\r\nSec-
Fetch-Mode: navigate\r\nSec-Fetch-User: ?1\r\n
Accept-Encoding: gzip, deflate, br\r\n
Accept-Language: zh-CN,zh;q=0.9,en;q=0.8\r\n
\r\n\r\n'
```

服务器端先是建立了一个 socket 连接，然后返回新连接对象。接着在连接对象上收到了 HTTP 协议格式的数据，这个数据就是浏览器发送的 HTTP 请求的报文内容。也就是说 HTTP 服务器在收到 HTTP 客户端的数据后会对数据进行解析，而这个数据本质上是

一个字符串，只是通过回车符、换行符而有了特定的意义。

这也是现在使用的大多数请求库的底层核心，不管是 request 还是 urllib，它们的底层都是 socket 封装的 HTTP 协议格式，同时提供了对 HTTP 响应格式的解析，所以使用者可以方便地获得相关的属性和内容。不管是响应的响应头还是响应体或者状态码，都预先解析成为响应对象的属性参数。

5.3.3　HTTPS 协议的实现

HTTPS(Hyper Text Transfer Protocol Secure，超文本传输安全协议)是一种通过计算机网络进行安全通信的传输协议。HTTPS 经由 HTTP 进行通信，但利用 SSL/TLS 来加密数据包。HTTPS 开发的主要目的，是提供对网站服务器的身份认证，保护交换数据的隐私与完整性。该协议由网景公司(Netscape)在 1994 年首次提出，随后扩展到互联网上。

HTTPS 的主要作用是在不安全的网络上创建一个安全信道，并在适当的加密包和服务器证书可被验证且可被信任时，对窃听和中间人攻击进行合理的防护。

HTTPS 的信任基于预先安装在操作系统中的证书颁发机构(CA)，因此到一个网站的 HTTPS 连接仅在如下情况下可被信任。

(1) 浏览器正确地实现了 HTTPS 且操作系统中安装了正确且受信任的证书。

(2) 证书颁发机构仅信任合法的网站。

(3) 被访问的网站提供了一个有效的证书，也就是说证书是一个由操作系统信任的证书颁发机构签发的(大部分浏览器会对无效的证书发出警告)。

该证书正确地验证了被访问的网站(例如访问 https://www.baidu.com 时，收到了签发给 baidu.com 而不是其他域名的证书)。

此协议的加密层(SSL/TLS)能够有效地提供认证和高强度的加密。

1. HTTPS 与 HTTP 的差异

HTTP 的 URL 由 http://开头，默认使用的监听端口是 80，而 HTTPS 的 URL 则是由 https://开头，默认使用的监听端口是 443。HTTP 不是安全的，而且攻击者可以通过监听和中间人攻击等手段，获取网站账户等敏感信息。HTTPS 的设计可以防止这些攻击，在其正确配置时网站是安全的。

2. SSL 协议的工作过程

当使用 HTTPS 时，如果客户端发送一个 ClientHello 消息，那么内容将包括支持的协议版本(如 TLS 1.0 版)、客户端生成的随机数(稍后用于生成会话密钥)、支持的加密算法(如 RSA 公钥加密)和支持的压缩算法。客户端会收到一个 ServerHello 的响应消息，内容包括确认使用的加密通信协议版本(如 TLS 1.0 版本，如果浏览器与服务器支持的版本不一致，服务器将关闭加密通信)、一个服务器生成的随机数(稍后用于生成对话密钥)、确认使用的加密方法(如 RSA 公钥加密)、服务器证书。

当双方知道了连接参数，客户端与服务器端会交换证书(依靠被选择的公钥系统)，这些证书通常基于 X.509，不过已有草案支持以 OpenPGP 为基础的证书。

服务器端请求客户端公钥，客户端有证书即双向身份认证，没证书时会随机生成公钥。客户端与服务器端通过公钥保密协商共同的主私钥(双方随机协商)，这些会通过精心谨慎设计的伪随机数功能实现。结果可能使用 Diffie-Hellman 交换，或使用简化的公钥加密，双方各自用私钥解密，所有其他关键数据的加密均使用这个“主密钥”。数据传输中的记录层

(Record layer)用于封装更高层的 HTTP 等协议。记录层数据可以被随意压缩、加密,与消息验证码压缩在一起,每个记录层包都有一个 Content-Type 段记录更上层用的协议。

继续运行之前的 HTTP 服务器,现在在 Chrome 浏览器中使用 https://localhost:5556 来访问 socket 服务器,此时会发现页面提示“建立连接失败”。这是因为客户端使用了 HTTPS 协议的工作流程,而 socket 服务器无法对该过程进行有效回应,客户端就终止了连接。socket 服务器显示收到如下的信息,这就不再是之前的明文信息。这条数据是客户端发送的 ClientHello 信息,但是服务器没有正确回应 ServerHello,客户端认为这是一个不安全连接,从而终止了 TCP 连接。

```
#收到数据
b'\x16\x03\x01\x02\x00\x01\x00\x01\xfc\x03\x03b\x84k\xcb\xaf\x07[\xac\x0c\xa4\xee\xb4\
xb3\xc9\x11\x8d\xa1\x88\xe2/j\x08\xbeJAi\xe4\xbfo\x9bj\xb4 \x06\xf8\x86\x95\xc0\xcb\xde\
x8d7\x90\x05\xad0!\xeb\t\xcb\xeewZ\xf9 <\x11bnwt\xf1\x84\xd6 * \x90\x00"\x8a\x8a\x13\x01\
x13\x02\x13\x03\xc0 + \xc0/\xc0,\xc00\xcc\xa9\xcc\xa8\xc0\x13\xc0\x14\x00\x9c\x00\x9d\
x00/\x005\x00\n\x01\x00\x01\x91jj\x00\x00\x00\x00\x00\x0e\x00\x0c\x00\x00\tlocalhost\x00
\x17\x00\x00\xff\x01\x00\x01\x00\x00\n\x00\n\x00\x08ZZ\x00\x1d\x00\x17\x00\x18\x00\x0b\
x00\x02\x01\x00\
...
0\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'
```

这里不再继续深入探讨 socket 如何对 HTTPS 的连接做响应,因为不会用到这样的技术细节,Web 的 HTTPS 部署是很简单的事情,通过 Nginx 可以快速部署 SSL 证书。

5.3.4 关于 TLS 与 SSL 协议

在上文提到了 HTTPS 是 HTTP 通过 SSL/TLS 来加密数据包的,这里对 SSL/TLS 做简要介绍。传输层安全性协议(Transport Layer Security,TLS)及其前身安全套接层(Secure Sockets Layer,SSL)是一种安全协议,目的是为互联网通信提供安全和数据完整性保障。网景公司(Netscape)在 1994 年推出首版网页浏览器——网景导航者时,同时推出了 HTTPS 协议,其以 SSL 进行加密,这是 SSL 的起源。

SSL 包含记录层(Record Layer)和传输层,记录层协议确定传输层数据的封装格式。传输层安全协议使用 X.509 认证,之后利用非对称加密演算来对通信方做身份认证,之后交换对称密钥作为会谈密钥(Session Key)。这个会谈密钥用来将通信两方交换的数据做加密,保证两个应用间通信的保密性和可靠性,使客户端与服务器端应用之间的通信不被攻击者窃听。

TLS 利用密钥算法在互联网上提供端点身份认证与通信保密,其基础是公钥基础设施。不过在实现的典型例子中,只有网络服务者能被可靠身份验证,其客户端则不一定。这是因为公钥基础设施普遍为商业化运营,电子签名证书通常需要付费购买。协议的设计在某种程度上能够使主从架构应用程序通信本身能预防窃听、干扰和消息伪造。

TLS 包含三个基本阶段。

(1) 对等协商支持密钥算法。

(2) 基于非对称密钥的信息传输加密和身份认证、基于 PKI 证书的身份认证。

(3) 基于对称密钥的数据传输保密。

在第一阶段,客户端与服务器端协商使用密码算法,当前广泛实现的算法选择如下。

(1) 公钥私钥非对称密钥保密系统:RSA、Diffie-Hellman、DSA。

（2）对称密钥保密系统：RC2、RC4、IDEA、DES、Triple DES、AES以及Camellia。

（3）单向散列函数：MD5、SHA1以及SHA256。

2014年10月，Google发布在SSL 3.0中发现设计缺陷的消息，建议禁用此协议。攻击者可以向TLS发送虚假错误提示，然后将安全连接强行降级到过时且不安全的SSL 3.0，然后就可以利用其中的设计漏洞窃取敏感信息。Google在自己公司的相关产品中陆续禁止回溯兼容，强制使用TLS协议。Mozilla也在11月25日发布的Firefox 34中彻底禁用了SSL 3.0，微软同样发出了安全通告。

SSL一直存在不安全的漏洞，1.0版本因为其严重的安全漏洞未被公开，2.0版本发布后也因为数个严重的安全漏洞被3.0版本替代。直到2014年，Google公布3.0版本存在设计缺陷，众多厂商宣布在产品中停止对SSL的支持。SSL和TSL的发展如表5-1所示。

表5-1　SSL和TSL的发布时间和状态

协　　议	发布时间	状　　态
SSL 1.0	未公布	未公布
SSL 2.0	1995年	已于2011年弃用
SSL 3.0	1996年	已于2015年弃用
TLS 1.0	1999年	计划于2020年弃用
TLS 1.1	2006年	计划于2020年弃用
TLS 1.2	2008年	-
TLS 1.3	2018年	-

5.3.5　一次爬虫请求的过程

当使用requests库或者urllib库去请求一条URL地址时，这个请求经历的过程如下。

1. 解析请求对象

首先是requests或其他请求库解析构造的请求对象，包括解析出域名，提取出表单信息，提取出设置的Cookie信息、Headers信息、代理信息等。

2. 解析域名

如果请求URL地址使用的是域名而不是IP地址，将会先把域名解析成IP地址。首先创建一个UDP的socket，把查询报文发给本地域名服务器，本地的域名服务器查到域名后，将对应的IP地址放在应答报文中返回。如果本地的DNS服务器不能解析，将向更高一级的DNS服务器发送请求。这样以此类推，一直向下解析，直到查询到所请求的域名为止。

3. 建立与目标站点的连接

当获得目标站点的IP地址之后，创建一个TCP的socket，socket通过TCP的三次握手建立TCP连接。建立连接后，将Headers、Cookie、请求数据按照HTTP协议的格式，组合成字符串并用bytes编码后发送给服务器的socket。

4. 数据转发

不能直接建立与目标站点的通信连接，数据要经过一系列的路由器和交换机转发，最后送到服务器，中间有很多传信人。

5. 服务器端解析

当服务器收到了客户端的socket数据后，将按照HTTP报文格式对其进行解析，将解

析出请求方法、请求的资源、请求体的数据。重要的是会解析出 Headers 中的 Cookie 及 User-Agent 头信息、Host 信息以及客户端的 IP 地址。

服务器将根据 Cookie 判断是否具有相应请求的权限，将根据 User-Agent、Host 信息判断是否是正常来源的请求，可能还会记录一下访问 IP，该 IP 访问太频繁了就不返回内容，这也是早期反爬虫的基本措施。内部经过一系列的处理后，服务器将处理结果封装成 HTTP 协议的响应报文发送给服务器端所连接的 TCP 客户端。

6. 数据转发

路由器和交换机将根据发送 HTTP 请求报文时留下的路由表记录和交换表记录转发 HTTP 响应报文。

7. 解析响应报文

本地 socket 收到响应报文后，会进行四次挥手关闭连接。如果有重定向，将用重定向地址重复上述流程。如果没有重定向，那么相关的库就解析出 HTTP 响应报文的内容。

8. 构造响应对象

最后根据解析的响应报文信息构造一个响应对象返回给用户。

上述流程大致描述了爬虫请求的过程，以及基础反爬虫的一些措施，真实的流程远远比这更加复杂。在 Python 中这些细节完全不必考虑，现成的请求库已经做好了封装，只需要专注于业务逻辑，发送请求然后再解析响应内容。

5.3.6 HTTP 响应状态码

所有 HTTP 响应报文的第一行都是状态行，其内容依次是当前 HTTP 版本号，3 位数字组成的状态代码，以及描述状态的短语，彼此由空格分隔。

状态代码的第一个数字代表当前响应的类型。

(1) 1xx 消息：请求已被服务器接收，继续处理。

(2) 2xx 成功：请求已成功被服务器接收、理解并接受。

(3) 3xx 重定向：需要后续操作才能完成这一请求。

(4) 4xx 请求错误：请求含有词法错误或者无法被执行。

(5) 5xx 服务器错误：服务器在处理某个正确请求时发生错误。

常用响应状态码及说明见表 5-2。

表 5-2 常见 HTTP 响应状态码及说明

状态码	状态码英文名称	说　明
100	Continue	继续处理
101	Switching Protocols	切换协议
200	OK	请求成功，一般用于 GET 与 POST 请求
201	Created	成功请求并创建了新的资源
202	Accepted	已经接收请求，但未处理完成
203	Non-Authoritative Information	请求成功，但返回的 meta 信息不在原始的服务器，而是一个副本
204	No Content	服务器成功处理，但未返回内容
205	Reset Content	服务器处理成功，用户终端应重置文档视图
206	Partial Content	服务器成功处理了部分 GET 请求

续表

状态码	状态码英文名称	说　明
300	Multiple Choices	多种选择。请求的资源可包括多个位置，相应可返回一个资源特征与地址的列表用于用户终端(例如浏览器)选择
301	Moved Permanently	永久重定向。请求的资源已被永久地移动到新的 URI，返回的信息会包括新的 URI，浏览器会自动定向到新的 URI
302	Found	临时重定向。资源只是临时被移动，客户端应继续使用原有的 URI
303	See Other	查看其他地址
304	Not Modified	所请求的资源未修改。服务器返回此状态码时，不会返回任何资源
305	Use Proxy	所请求的资源必须通过代理访问
307	Temporary Redirect	临时重定向。与 302 类似，使用 GET 请求重定向
400	Bad Request	客户端请求的语法错误，服务器无法理解
401	Unauthorized	请求要求用户的身份认证
403	Forbidden	服务器理解请求客户端的请求，但拒绝响应
404	Not Found	服务器无法根据客户端的请求找到资源
405	Method Not Allowed	客户端请求中的方法被禁止
406	Not Acceptable	服务器无法根据客户端请求的数据完成请求
407	Proxy Authentication Required	请求要求代理的身份认证，与 401 类似，但请求者应当使用代理进行授权
408	Request Time-out	服务器等待客户端发送的请求时间过长，已超时
409	Conflict	服务器完成客户端的 PUT 请求时可能返回此代码，表明服务器处理请求时发生了冲突
410	Gone	资源已被永久删除
411	Length Required	服务器无法处理客户端发送的不带 Content-Length 的请求信息
412	Precondition Failed	客户端请求信息的先决条件错误
413	Request Entity Too Large	由于请求的实体过大，服务器无法处理
414	Request-URI Too Large	请求的 URI 过长，服务器无法处理
415	Unsupported Media Type	服务器无法处理请求附带的媒体格式
416	Requested range not satisfiable	客户端请求的范围无效
417	Expectation Failed	服务器无法满足 Expect 的请求头信息
500	Internal Server Error	服务器内部错误，无法完成请求
501	Not Implemented	服务器不支持请求的功能，无法完成请求
502	Bad Gateway	作为网关或者代理工作的服务器尝试执行请求时，从远程服务器接收到了一个无效的响应
503	Service Unavailable	由于超载或系统维护，服务器暂停处理请求
504	Gateway Time-out	充当网关或代理的服务器超时
505	HTTP Version not supported	服务器不支持请求的 HTTP 协议的版本

5.3.7 HTTP 请求头与响应头

HTTP 请求头提供了关于请求、响应或者其他的发送实体的信息。HTTP 的头信息包括通用头、请求头、响应头和实体头四部分。每个头域由一个域名、冒号和域值三部分组成。

（1）通用头标：可用于请求，也可用于响应，作为一个整体而不是特定资源与事务相关联。

（2）请求头标：允许客户端传递关于自身的信息和希望的响应形式。

（3）响应头标：服务器用于传递自身信息的响应。

（4）实体头标：定义被传送资源的信息，如标识图片、HTML 文档、媒体资源。既可用于请求，也可用于响应。

常见的请求头如表 5-3 所示，常见响应头如表 5-4 所示。

表 5-3　常见请求头一览表

常见请求头	解　　释	示　　例
Accept	指定客户端能够接收的内容类型	Accept：text/plain，text/html
Accept-Charset	浏览器可以接收的字符编码集	Accept-Charset：iso-8859-5
Accept-Encoding	指定浏览器可以支持的 Web 服务器返回内容的压缩编码类型	Accept-Encoding：compress，gzip
Accept-Language	浏览器可接收的语言	Accept-Language：en，zh
Accept-Ranges	可以请求网页实体的一个或者多个子范围字段	Accept-Ranges：bytes
Authorization	HTTP 授权的授权证书	Authorization：Basic QWxhZGRpbjpvcGVuIHNlc2FtZQ==
Cache-Control	指定请求和响应遵循的缓存机制	Cache-Control：no-cache
Connection	表示是否需要持久连接。（HTTP 1.1 版本默认进行持久连接）	Connection：close
Cookie	HTTP 请求发送时，会把保存在该请求域名下的所有 Cookie 值一起发送给 Web 服务器	Cookie：$Version=1；Skin=new；
Content-Length	请求内容长度	Content-Length：348
Content-Type	请求的与实体对应的 MIME 信息	Content-Type：application/x-www-form-urlencoded
Date	请求发送的日期和时间	Date：Tue，15 Nov 2010 08：12：31 GMT
Expect	请求特定的服务器行为	Expect：100-continue
From	发出请求的用户的 Email	From：user@email.com
Host	指定请求的服务器的域名和端口号	Host：www.zcmhi.com
If-Match	只有请求内容与实体相匹配才有效	If-Match："737060cd8c284d8af7ad3082f209582d"
If-Modified-Since	如果请求的部分在指定时间之后被修改，则请求成功，未被修改则返回 304 代码	If-Modified-Since：Sat，29 Oct 2010 19：43：31 GMT
If-None-Match	如果内容未改变则返回 304 代码，参数为服务器先前发送的 Etag，将与服务器回应的 Etag 比较判断是否改变	If-None-Match："737060cd8c284d8af7ad3082f209582d"
If-Range	如果实体未改变，服务器发送客户端丢失的部分，否则发送整个实体。其参数也为 Etag	If-Range："737060cd8c284d8af7ad3082f209582d"
If-Unmodified-Since	只在实体在指定时间之后未被修改才请求成功	If-Unmodified-Since：Sat，29 Oct 2010 19：43：31 GMT

续表

常见请求头	解 释	示 例
Max-Forwards	限制信息通过代理和网关传送的时间	Max-Forwards：10
Pragma	用来包含实现特定的指令	Pragma：no-cache
Proxy-Authorization	连接到代理的授权证书	Proxy-Authorization：Basic QWxhZGRpbjpvcGVuIHNlc2FtZQ==
Range	只请求实体的一部分，指定请求范围	Range：bytes=500-999
Referer	先前网页的地址，当前请求网页紧随其后，即来路	Referer：http://www.zcmhi.com/archives/71.html
TE	客户端愿意接收的传输编码	TE：trailers，deflate；q=0.5
Upgrade	向服务器指定某种传输协议以便服务器进行转换（如果支持）	Upgrade：HTTP/2.0，SHTTP/1.3，IRC/6.9，RTA/x11
User-Agent	User-Agent 的内容包含发出请求的用户信息	User-Agent：Mozilla/5.0（Linux；X11）
Via	通知中间网关、代理服务器地址或通信协议	Via：1.0 fred，1.1 nowhere.com（Apache/1.1）
Warning	关于消息实体的警告信息	Warn：199 Miscellaneous warning

表 5-4 常见响应头一览表

常见响应头	解 释	示 例
Accept-Ranges	表明服务器是否支持指定范围请求及支持哪种类型的分段请求	Accept-Ranges：bytes
Age	从原始服务器到代理缓存形成的估算时间（以秒计，非负）	Age：12
Allow	对某网络资源的有效的请求行为，不允许则返回 405	Allow：GET，HEAD
Cache-Control	告诉所有的缓存机制是否可以缓存及缓存哪种类型	Cache-Control：no-cache
Content-Encoding	Web 服务器支持的返回内容压缩编码类型	Content-Encoding：gzip
Content-Language	响应体的语言	Content-Language：en，zh
Content-Length	响应体的长度	Content-Length：348
Content-Location	请求资源可替代的另一备用地址	Content-Location：/index.htm
Content-MD5	返回资源的 MD5 校验值	Content-MD5：Q2hlY2sgSW50ZWdyaXR5IQ==
Content-Range	在整个返回体中本部分的字节位置	Content-Range：bytes 21010-47021/47022
Content-Type	返回内容的 MIME 类型	Content-Type：text/html；charset=utf-8
Date	原始服务器消息发出的时间	Date：Tue，15 Nov 2010 08：12：31 GMT
ETag	请求变量的实体标签的当前值	ETag："737060cd8c284d8af7ad3082f209582d"
Expires	响应过期的日期和时间	Expires：Thu，01 Dec 2010 16：00：00 GMT
Last-Modified	请求资源的最后修改时间	Last-Modified：Tue，15 Nov 2010 12：45：26 GMT
Location	用重定向接收方到非请求 URL 的位置来完成请求或标识新的资源	Location：http://www.zcmhi.com/archives/94.html

续表

常见响应头	解　　释	示　　例
Pragma	实现特定的指令，可应用到响应链上的任何接收方	Pragma：no-cache
Proxy-Authenticate	指出认证方案和可应用到代理的该URL上的参数	Proxy-Authenticate：Basic
refresh	应用于重定向或一个新的资源被创造时，在5s之后重定向（由网景公司提出，被大部分浏览器支持）	Refresh：5；url＝http://www.zcmhi.com/archives/94.html
Retry-After	如果实体暂时不可取，通知客户端在指定时间之后再次尝试	Retry-After：120
Server	Web服务器的软件名称	Server：Apache/1.3.27（UNIX）（Red-Hat/Linux）
Set-Cookie	设置Http Cookie	Set-Cookie：UserID＝JohnDoe；Max-Age＝3600；Version＝1
Trailer	指出头域在分块传输编码的尾部存在	Trailer：Max-Forwards
Transfer-Encoding	文件传输编码	Transfer-Encoding:chunked
Vary	告诉下游代理是使用缓存响应还是从原始服务器请求	Vary：*
Via	告知代理客户端响应是通过哪里发送的	Via：1.0 fred,1.1 nowhere.com（Apache/1.1）
Warning	警告实体可能存在的问题	Warning：199 Miscellaneous warning
WWW-Authenticate	表明客户端请求实体应该使用的授权方案	WWW-Authenticate：Basic

5.4　WebSocket协议

5.4.1　协议内容

WebSocket是一种网络传输协议，可在单个TCP连接上进行全双工通信，位于OSI模型的应用层。WebSocket使得客户端和服务器之间的数据交换变得更加简单，允许服务器端主动向客户端推送数据。

在WebSocket API中，浏览器和服务器只需要完成一次握手，两者之间就可以创建持久性的连接，并进行双向数据传输。HTML5定义了WebSocket协议，能更好地节省服务器资源和带宽，并且能够更实时地进行通信。WebSocket使用ws或wss的统一资源标志符，类似于HTTPS。其中wss表示使用了TLS的WebSocket，其格式如下所示。

```
ws://likeinlove.com/wsapi
wss://likeinlove.com/wsapi
```

WebSocket与HTTP和HTTPS使用相同的TCP端口，可以绕过大多数防火墙的限制。默认情况下，WebSocket协议使用80端口。当运行在TLS之上时，则默认使用443端口。

WebSocket是独立的、创建在TCP上的协议。WebSocket通过HTTP/1.1协议的101状态码进行握手。为了创建WebSocket连接，需要通过浏览器发出请求，之后服务器进行回应，这个过程通常称为握手。

一个典型的 WebSocket 握手请求如下，这是客户端请求头。

```
GET / HTTP/1.1
Upgrade: websocket
Connection: Upgrade
Host: likeinlove.com
Origin: http://example.com
Sec-Websocket-Key: sN9cRrP/n9NdMgdcy2VJFQ==
Sec-Websocket-Version: 13
```

服务器回应的消息内容如下。

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-Websocket-Accept: fFBooB7FAkLlXgRSz0BT3v4hq5s=
Sec-Websocket-Location: ws://likeinlove.com/
```

上面涉及的字段说明如下。

(1) Connection：必须设置 Upgrade，表示客户端希望连接升级。

(2) Upgrade：字段必须设置 Websocket，表示希望升级到 Websocket 协议。

(3) Sec-Websocket-Key：是随机的字符串，服务器端会用这些数据构造出一个 SHA-1 的信息摘要。把 Sec-Websocket-Key 加上一个特殊字符串 258EAFA5-E914-47DA-95CA-C5AB0DC85B11，然后计算 SHA-1 摘要，之后进行 BASE-64 编码，将结果作为 Sec-Websocket-Accept 头的值再返回给客户端。如此操作，可以尽量避免普通 HTTP 请求被误认为是 WebSocket 协议。

(4) Sec-Websocket-Version：表示支持的 WebSocket 版本。RFC 6455 要求使用的版本是 13，之前草案的版本均应当弃用。

(5) Origin：是可选的，通常用来表示在浏览器中发起此 WebSocket 连接所在的页面，类似于 Referer。但与 Referer 不同的是，Origin 只包含了协议和主机名称。

其他一些定义在 HTTP 协议中的字段，如 Cookie 等，也可以在 WebSocket 中使用。

WebSocket 协议可以用于股票或基金的实时报价、基于位置的应用、在线教育、体育实况更新、协同编辑、视频会议、社交聊天、弹幕、多玩家游戏、智能家居等需要高实时通信的场景。

5.4.2　Python 连接 WebSocket


视频讲解

在 HTML5 中的 WebSocket 中有四个事件、两个方法、两个属性，这些就是 WebSocket 的核心接口。

四个事件，分别对应建立连接时、收到消息时、通信错误时、关闭连接时四个状态下的调用方法，四个事件名及作用如下。

(1) open：连接建立时触发。

(2) message：客户端接收服务器端数据时触发。

(3) error：通信发生错误时触发。

(4) close：连接关闭时触发。

两个方法主要用于发送数据、主动关闭连接，两个方法名及作用如下。

(1) socket.send()：使用连接发送数据。

(2) socket.close()：关闭连接。

两个属性分别是表示连接状态的属性、等待传输数据的大小，其状态名及意义如下。

(1) socket.readyState：表示连接状态。0 表示连接尚未建立；1 表示连接已建立，可以进行通信；2 表示连接正在关闭；3 表示连接已经关闭或者连接不能打开。

(2) socket.bufferedAmount：正在发送队列中等待传输的字节数。

不管使用哪种开发语言，都会实现上面的四个事件、两个方法、两个属性。对于 Python 而言也不例外，Python 中支持 WebSocket 的库，也会提供基于标准接口的属性和方法。Python 3 目前支持 WebSocket 客户端连接的库有 websockets、websocket-client、aiowebsocket 等。这里以 websocket_client 的客户端为例，介绍几个常用的方法和接口，使用 pip install websocket_client 命令安装 WebSocket。

下面是 websocket_client 文档提供的一个案例，案例中使用的 WebSocket 连接地址是测试用的网址，案例地址参见附录。

```
import websocket
try:
    import thread
except ImportError:
    import _thread as thread
import time

def on_message(ws, message):
    """
    有消息时调用
    :param ws:
    :param message:
    :return:
    """
    print(message)

def on_error(ws, error):
    """
    发生错误时调用
    :param ws:
    :param error:
    :return:
    """
    print(error)

def on_close(ws):
    """
    服务器端关闭时调用
    :param ws:
    :return:
    """
    print("###closed###")

def on_open(ws):
    """
    打开 ws 连接时调用
    :param ws:
    :return:
    """
```

```
        def run(*args):
            for i in range(3):
                time.sleep(1)
                ws.send("hello")
            time.sleep(1)
            ws.close()
            print("thread terminating...")

        thread.start_new_thread(run, ())

    if __name__ == "__main__":
        websocket.enableTrace(True)      #开启状态监控
        ws = websocket.WebsocketApp("ws://echo.websocket.org/",
                                    on_message = on_message,
                                    on_error = on_error,
                                    on_close = on_close)
        ws.on_open = on_open             #单独绑定打开时连接
        ws.run_forever()
```

on_open()、on_close()、on_error()、on_ message()分别是 WebSocket 中的四个重要事件，四个事件在 WebSocket 相应的状态下会自动调用，并传入 WebSocket 连接对象和其他附加信息。上述源码的作用是连接到 ws://echo.websocket.org/，这是一个 WebSocket 的测试网址，向这个测试网址发送消息，它会把发送的消息转发回来，运行中的输出信息如下所示。

```
--- request header ---
GET / HTTP/1.1
Upgrade: websocket
Host: echo.websocket.org
Origin: http://echo.websocket.org
Sec-Websocket-Key: 73Q6fh2msLK3H/fHHoYc2w==
Sec-Websocket-Version: 13
Connection: upgrade


-----------------------
--- response header ---
HTTP/1.1 101 Web socket Protocol Handshake
Connection: Upgrade
Date: Wed, 11 Mar 2020 13:44:37 GMT
Sec-Websocket-Accept: 8W2JMXTIJB2/WAPV2MX156aEe7E=
Server: Kaazing Gateway
Upgrade: websocket
-----------------------
send: b'\x81\x85\xcf\xdc\xab\xea\xa7\xb9\xc7\x86\xa0'
hello
send: b'\x81\x85\xfd@h\x83\x95%\x04\xef\x92'
hello
send: b'\x81\x85[\x7f\x99"3\x1a\xf5N4'
send: b'\x88\x82\x16a\x14S\x15\x89'
###closed###
```

5.4.3　案例：虚拟货币实时价格爬虫

本案例的目标在于抓取市面上主流的几千个数字货币的实时价格，并将其保存到 MongoDB 中，然后通过另一个脚本来实时更新价格。

1. 网站分析

目标网站是英为财情，URL 地址是 https://cn.investing.com/crypto/currencies，这上面有 2790 多个数字货币，图 5-12 即为目标页面。打开网站后首先进行环境检测，环境检测通过后跳转到数据页面，页面含有认证过的 Cookie 信息，同时该 Cookie 信息和 User Agent 值相对应，使用 Cookie 时需要注意携带对应的 User Agent 值。

查找加密货币（名称/代码）　　美元　　USD

#	名称	代码	价格 (USD)	市值	成交量(24H)	交易份额	涨跌(24H)	涨跌(7日)
1	比特币	BTC	7,839.9	$143.71B	$38.61B	0%	-2.88%	-10.22%
2	以太坊	ETH	196.34	$21.78B	$16.02B	0%	-3.28%	-11.75%
3	瑞波币	XRP	0.2069	$9.12B	$2.14B	0%	-3.08%	-11.04%
4	比特币现金	BCH	263.59	$4.85B	$3.33B	0%	-3.38%	-17.71%
5	泰达币	USDT	1.0052	$4.65B	$46.24B	0%	+0.05%	+0.06%

图 5-12　目标页面

1）分析 WebSocket 的连接过程

价格(USD)列是动态更新的，光标在闪烁代表有波动，但是其他字段是相对固定的，应该指定更新周期，而不是实时更新。打开浏览器的开发者工具，选择 Network 面板中的 WS 选项，这里记录了 WebSocket 的连接信息，单击 Messages 选项可以看见 WebSocket 连接交互的数据情况，如图 5-13 所示。

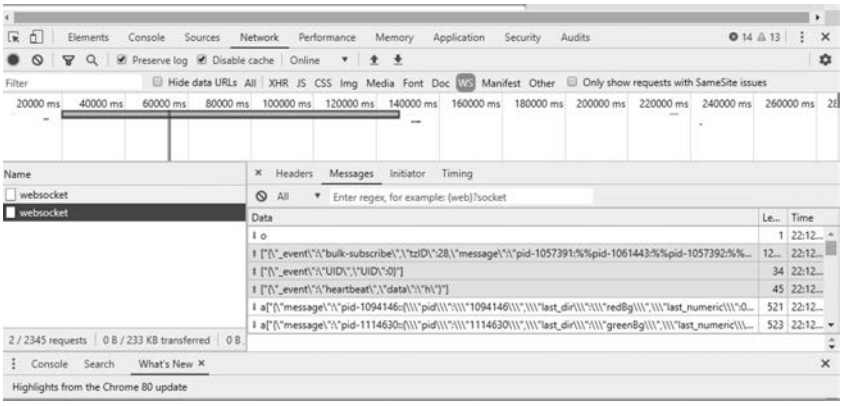

图 5-13　WebSocket 连接交互的数据情况

单击 Headers 选项可以看见请求的相关头信息和详细 URL，请求地址是 wss://stream148.forexpros.com/echo/396/0zj_g4mu/websocket。

再看 WebSocket 交互的数据的特点，刚建立连接的时候客户端向服务器端发送了三条数据，然后服务器端再源源不断地返回信息。这三条数据应该和服务器端返不返回数据有关，客户端向服务器端发送的这三条数据如下。

```
["{\"_event\":\"bulk-subscribe\",\"tzID\":28,\"message\":\"pid-1057391:%%pid-1061443:%%pid-1057392:%%pid-1061410:%%pid-1061453:%%pid-1099037:%%pid-1061445:%%pid-1061444:%%pid-1061448:%%pid-1061796:%%pid-1061794:%%pid-1062537:%%pid-…1094146:%%pid-1066618:%%pid-1066620:%%pid-1061455:%%pid-1066643:%%pid-1061791:%%pid-1061824:%%pid-1061781:%%pid-1061782:%%pid-1141873:%%pid-1138414:%%pid-1061517:%%pid-1094126:%%pid-1094111:%%pid-1066622:%%pid-1123468:%%pid-1066628:%%cmt-6-5-945629:%%domain-6:\"}"]
```

```
["{\"_event\":\"UID\",\"UID\":0}"]

["{\"_event\":\"heartbeat\",\"data\":\"h\"}"]
```

第一条数据中出现了很多 pid 字段，在服务器端返回的信息中也含有这些 pid，那么第一条数据应该是告诉服务器端需要实时更新那些数字货币的价格。

第三条数据在服务器端源源不断地发送数据时保持一定的频率发送，再结合数据中的关键字 heartbeat，其中文是心跳的意思，那么这应该是心跳数据，是为了告诉服务器端目前状态正常，可以继续发送实时数据。

至于第二条数据暂时不清楚其作用，为弄明白它的作用，可以找一个 WebSocket 连接测试网站，经过测试发现当 WebSocket 连接建立发送第一条数据后，服务器端就源源不断地传回实时价格。如果什么都不做，再等几分钟 WebSocket 服务器端就主动断开连接了。如果在服务器端发送数据的时候，保持一定频率发送心跳数据，那么 WebSocket 的连接很久都不会断，因此可以暂时忽略第二个数据。通过在线工具，测试情况如图 5-14 所示，在搜索引擎中搜索 WebSocket 在线测试可获得该工具。

图 5-14　测试 WebSocket 连接

然后接着分析 pid 是哪里来的。在 HTML 源码中随便找一个 pid，找到的 pid 来源在每个价格数据所在的 a 标签节点下的 class 属性节点中，如下源码所示。

```
<a class = "pid - 1061443 - last" href = "/crypto/currency - pairs?c1 = 195&amp;c2 = 12" target
 = "_blank" boundblank = ""> 196.38 </a >
```

196.38 是价格，a 标签的 class 属性中的 pid-1061443-last 的中间部分就是 pid。每个价格都有这样的 pid 参数，但并不是所有的数字货币都有 pid，主要看数字货币对应的价格是否被 a 标签包裹，如果被 a 标签包裹则是有 pid 的，即可以发送给 WebSocket 服务器端更新价格。

现在就弄清楚了数据交互的流程。首先是创建 WebSocket 连接，然后发送需要更新实时价格的 pid 列表，WebSocket 连接则会不断更新这些 pid 对应的价格，当然在这个过程中还需要按照一定频率发送心跳数据，不然服务器端会认为客户端不再需要更新。

2）分析 HTML 页面的加载

首先访问全部加密货币的地址 https://cn.investing.com/crypto/currencies，通过环境检测后，初始页面只有 100 条数据，这 100 条是热门数字货币。然后浏览器继续加载，等一段时间后才加载完，剩下的数据是通过后台加载的，可以猜想是通过 JavaScript 加载、XHR 加载或者 CSS 加载的，具体是哪个目前还不确定。

继续看开发者工具里面的数据包，其中有一条 POST 请求十分可疑。观察它的响应数据如下，这个 POST 响应文件的大小在 4MB 左右，它返回一个字典，有三个字段，分别是 html、css_crypto_type、ids，html 字段的内容是 HTML 页面中剩下的几千个数字货币列表，

ids是剩下部分的数字货币的pid,有pid就代表可以更新价格。

```
[
html: "< tr > < td class = "rank icon"> 101 </td >< td class = "flag" >< i class = "cryptoIcon c_
terra - luna middle"></i ></td >< td class = "left bold elp name cryptoName first js - currency -
name"
title = "Terra">< span > Terra </span > </td > < td class = "left noWrap elp symb js - currency -
symbol"
… 省略大约 4MB 数据
title = "LUNA"> LUNA </td >< td class = "price js - currency - price">< span > 0.203882 </span > </td>
< td class = "js - market - cap" data - value = "58670360.359213"> &#x24;58.67M </td >
< td class = "js - 24h - volume"data - value = "3513811.0250506"> &#x24;3.51M </td > < tdclass
= "js - total - vol"> 0 % </td >< td class = "js - currency - change - 24h redFont"> - 0.95 % </td >"
css_crypto_type: "v_221588_cryptoIconAll"
ids: [1066743, 1066619, 1138411, 1062033, 1094102, 1061531, 1153119, 1153135, 1152984,
1061482, 1135966, …
]
```

再看这个POST请求的相关信息,请求地址、请求头及表单数据如下。请求头中的Cookie头需要特别关注一下,Cooke中含有初始打开页面环境检测成功后的验证信息。将下面的请求地址、请求头及表单在Postman中测试一下,故意去掉Cookie结果,会返回503错误代码,那么由此可知,Cookie的作用非常大。

```
Request URL: https://cn.investing.com/crypto/Service/LoadCryptoCurrencies
Request Method: POST
Status Code: 200 OK
Remote Address: 106.2.12.11:443
Referrer Policy: no - referrer - when - downgrade

Accept: application/json, text/javascript, * / * ; q = 0.01
Accept - Encoding: gzip, deflate, br
Accept - Language: zh - CN,zh;q = 0.9,en;q = 0.8
Cache - Control: no - cache
Connection: keep - alive
Content - Length: 13
Content - Type: application/x - www - form - urlencoded
Cookie: PHPSESSID = v03nqkrfte89362ikojbcliklp; geoC = CN; prebid _ page = 0; prebid _ session = 0;
adBlockerNewUserDomains = 1583927473; StickySession = id.89631268069.800cn.investing.com; __gads = ID
= a06bedf2692e5492:T = 1583927475:S = ALNI _ MZMOIDbci8CK4Rrg9h2pMSLeQ4cQw; _ga = GA1.2.1545695630.
1583927474; _ gid = GA1. 2. 2060687312. 1583927476; Hm _ lvt _ a1e3d50107c2a0e021d734fe76f85914 =
1583927476; _gat = 1; _gat_allSitesTracker = 1; nyxDorf = ZWFkNmU0YSM0Y21mMGA3KzVjP2QzKjUzZm8 % 3D; Hm_
lpvt_a1e3d50107c2a0e021d734fe76f85914 = 1583940726
Host: cn.investing.com
Origin: https://cn.investing.com
Pragma: no - cache
Referer: https://cn.investing.com/crypto/currencies
Sec - Fetch - Dest: empty
Sec - Fetch - Mode: cors
Sec - Fetch - Site: same - origin
User - Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/80.0.3987.132 Safari/537.36
X - Requested - With: XMLHttpRequest

lastRowId: 100
```

对于获取虚拟货币列表部分,由于试试一次性获取有限数量的列表,这里采用Selenium来抓取,避免通过直接请求接口受到限制。

2. 确定需求

下面来明确需求，总体需求是先请求所有数字货币的列表，然后将所有数字货币的初始状态信息存储在 MongoDB 数据库中，然后创建 WebSocket 信息，不断地更新 MongoDB 中的价格。

存储在 MongoDB 中的字段设计如下。

(1) pid：数字货币对应的 pid。

(2) index：在网页列表的排序。

(3) name：数字货币的名字。

(4) code：数字货币的代码。

(5) price：数字货币对应的价格。

(6) value：作为数字货币的总市值。

(7) deal：作为数字货币的 24 小时成交量。

(8) share：作为交易数字货币的份额。

(9) gain：作为数字货币的 14 小时跌涨值。

(10) gains：作为数字货币的 7 日跌涨值。

(11) insert_time：插入数据库时间戳。

(12) update_time：最近更新时间戳。

3. 分步实现

首先解析首页所有数字货币信息的代码。虚实货币的 pid 是一次性获取后保存在数据库中的，所以抓取首页数字货币种类的 ID 信息和 WebSocket 更新价格可以分开来处理。创建一个 DcSpider 的项目文件夹，在项目文件夹下面创建两个项目文件，其中一个是 dcspider.py，另一个是 dcupdate.py。dcspider.py 用于获取全部的数字货币列表信息，并将其写入 MongoDB，dcupdate.py 用来更新 MongoDB 中已经存在的 pid 的数字货币的价格。

其中 dcspider.py 文件内的爬虫获取虚拟货币的 ID 信息，该网站采用的是 Cloudflare (5 秒盾)安全工具，用来防御爬虫程序。Cloudflare 是目前主流的网站安全工具之一，主要通过环境检测来识别是否为异常流量访问，并且识别策略在不断更新，所以这里信息抓取的方案就是使用自动化浏览器。使用 Selenium 抹除自动化浏览器的被识别的特征，注意需要使用 Chrome 浏览器 90 及以下版本，然后通过 Cloudflare 的检验，接着跳转到虚拟货币列表，不断翻页，抓取完所有的列表，并存储到 MongDB 数据库中，其实现源码如下。

```
from lxml import etree
import logging
import time
import pymongo
from selenium import webdriver
from selenium.webdriver.chrome.options import Options
from selenium.webdriver.support import expected_conditions as EC
from selenium.webdriver.support.wait import WebDriverWait
from selenium.webdriver.common.by import By

logger = logging.getLogger('scspider')    # 创建一个记录器用于输出控制台
logger.setLevel(logging.INFO)
ch = logging.StreamHandler()
ch.setLevel(logging.INFO)
```

```
fmt = logging.Formatter(
    "%(asctime)s - %(filename)s[line:%(lineno)d] - %(levelname)s: %(message)s")
ch.setFormatter(fmt)
logger.addHandler(ch)


def run():
    ua = 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/89.0.4389.114 Safari/537.36'
    options = Options()
    options.add_argument("--disable-blink-features")
    options.add_argument("--disable-blink-features=AutomationControlled")
    options.add_argument(
        f'--user-agent={ua}')
    driver = webdriver.Chrome(options=options)
    client = pymongo.MongoClient("mongodb://localhost:27017/")
    db = client["fictitious"]
    col = db["data"]
    js = "window.open('https://cn.investing.com/crypto/currencies')"
    driver.execute_script(js)
    time.sleep(10)
    driver.switch_to.window(driver.window_handles[-1])
    iframe = WebDriverWait(driver, 10).until(EC.presence_of_element_located((By.TAG_NAME,
"iframe")))
    # 切换到 iframe 上下文
    driver.switch_to.frame(iframe)
    element = driver.find_element(By.XPATH, '//*[@id="challenge-stage"]/div/label/
input')
    element.click()
    while True:
        try:
            WebDriverWait(driver, 15).until(EC.title_contains(u"数字货币行情_所有加密虚"))
        except BaseException:
            driver.execute_script("window.stop();")
        rp = etree.HTML(driver.page_source)
        keys = ["index", "name", "code", "price", "value", "deal", "share", "gain",
"gains"]
        tr_html = rp.xpath('//*[@id="DataTables_Table_0"]/tbody//tr')
        for el in tr_html:
            values = [i for i in el.xpath('td//text()') if len(i.strip()) > 0]
            item = dict(zip(keys, values))
            pid_el = el.xpath('td/a[starts-with(@class,"pid-")]/@class')
            if pid_el:
                pid = pid_el[0].split('-')[1]
                item["pid"] = pid
            item['update_time'] = item['insert_time'] = time.time()
            col_data = col.find_one({"code": item['code']})              # 查询是否存在
            if col_data:
                col.update_one({"code": item['code']}, {"$set": item})  # 存在即更新
                logger.info(f"更新数据成功:{item}")
            else:
                col.insert_one(item) # 不存在即插入
                logger.info(f"插入数据成功:{item}")
        element = driver.find_element(By.XPATH, '//*[@id="DataTables_Table_0_next"]')
        tabindex = element.get_attribute('tabindex')
        if str(tabindex) == "-1":
            break
```

```
            else:
                driver.execute_script("arguments[0].click();", element)
                time.sleep(3)


if __name__ == '__main__':
    run()
```

源码的思路是这样的，首先设置日志输出控制台、设置 MongoDB 连接、设置对应的 URL 地址和 Headers 字典，然后分别请求首页的 100 条数据和后台加载的剩下的几千条数据。分别解析出列表的元素节点，遍历元素节点获取 pid 和其他几个参数信息列表，构建信息字典 item。最后对每项 item 查询 MongoDB，如果存在则更新，如果不存在则插入数据。这部分脚本不必实时运行，只需要按照一定频率刷新 MongoDB 中的数字货币列表。

运行脚本控制台，输出日志如下。

```
2020-03-12 09:59:57,194 - html.py[line:53] - INFO: 下载数据成功
2020-03-12 09:59:57,546 - html.py[line:76] - INFO: 更新数据成功:{'index': '1', 'name': '比
特币', 'code': 'BTC', 'price': '7,711.3', 'value': '$143.18B', 'deal': '$38.60B', 'share': '0%',
'gain': '-2.83%', 'gains': '-11.22%', 'pid': '1057391', 'update_time': 1583978397.5363529,
'insert_time': 1583978397.5363529}
…
2020-03-12 10:00:21,476 - html.py[line:81] - INFO: 数据处理完成,总数:2794
```

再登录 MongoDB 服务器，查询 MongoDB 中的数据情况，数据显示如下。

```
rs:PRIMARY> show dbs
admin       0.000GB
config      0.000GB
fictitious  0.001GB
local       0.001GB
test        0.000GB
rs:PRIMARY> use fictitious
switched to db fictitious
rs:PRIMARY> db.data.find()
{ "_id" : "1057391", "index" : "1", "name" : "比特币", "code" : "BTC", "price" : "7,711.3",
"value" : "$143.18B", "deal" : "$38.60B", "share" : "0%", "gain" : "-2.83%", "gains" :
"-11.22%", "update_time" : 1583978397.5363529, "insert_time" : 1583978397.5363529, "pid" :
"1057391" }
…
Type "it" for more
```

下面再实现 dcupdate.py 内的脚本逻辑，这部分的脚本逻辑相对简单，只需要从 MongoDB 中查询出所有的 pid 号，然后在创建与服务器的 WebSocket 连接的时候分批发给 WebSocket 服务器，告诉它要更新这些数据。当数据有变化的时候，WebSocket 服务器会把更新价格发给客户端，客户端解析后更新 MongoDB 中的数据。再看 WebSocket 服务器发送过来的数据，处理过后可阅读性的数据如下，数据是一个字典的样子，需要将中间信息处理一下转换成更新信息。

```
{'message': 'pid-1057391::
{"pid":"1057391","last_dir":"greenBg","last_numeric":7833.1,"last":"7,833.1","bid":
"0.0","ask":"0.0","high":"7,964.9","low":"7,761.9","last_close":"8,031.4","pc":"-198.3",
"pcp":"-2.53%","pc_col":"redFont","turnover":"1.13M","turnover_numeric":1131451,
"time":"11:51:44","timestamp":1583927504}'}
```

在 WebSocket 返回的数据中获取 last_numeric 字段，这个字段主要是实时价格，对应数据库中的 price 字段。需要注意的是，WebSocket 服务器发回来的所有信息都是价格，同时还推送了实时新闻。这部分实现源码如下。

源码中主要实现了 on_message()事件，该事件会在 WebSocket 服务器端发送消息过来时自动调用。WebSocket 握手之后，WebSocket 服务器端会发送一条消息过来标记开始，发送的消息是字母 o，然后 WebSocket 服务器端查询出所有的 pid，将其分批组成查询请求发送给 WebSocket 服务器端，这一步只在每个连接开始时进行一次。

后面的回调都是 WebSocket 服务器传回来的消息，消息不是标准的 JSON 数据，因此使用正则表达式解析其中的 pid、价格、更新时间，返回的数据样本及正则如下，最后获得相应的结果并将其更新到数据库。

```
#返回的更新数据
'a["{\\"message\\":\\"pid-1114630::{\\\\\\"pid\\\\\\":\\\\\\"1114630\\\\\\",\\\\\\"
last_dir\\\\\\":\\\\\\"redBg\\\\\\",\\\\\\"last_numeric\\\\\\":1.00089,\\\\\\"last\\\\\
\\":\\\\\\"1.000890\\\\\\",\\\\\\"bid\\\\\\":\\\\\\"0.000000\\\\\\",\\\\\\"ask\\\\\\":\\
\\\\"0.000000\\\\\\",\\\\\\"high\\\\\\":\\\\\\"1.001470\\\\\\",\\\\\\"low\\\\\\":\\\\\
\"0.999130\\\\\\",\\\\\\"last_close\\\\\\":\\\\\\"0.999753\\\\\\",\\\\\\"pc\\\\\\":\\\\
\\" + 0.001137\\\\\\",\\\\\\"pcp\\\\\\":\\\\\\" + 0.11 % \\\\\\",\\\\\\"pc_col\\\\\\":\\\\
\\"greenFont\\\\\\",\\\\\\"turnover\\\\\\":\\\\\\"31.95M\\\\\\",\\\\\\"turnover_numeric\\
\\\\\":31952611,\\\\\\"time\\\\\\":\\\\\\"5:19:58\\\\\\",\\\\\\"timestamp\\\\\\":
1583990397}\\"}"]'
#正则匹配
re.findall(r'"pid. * ?(\d + ). * last_numeric. * ?:(\d * \.?\d * ). + timestamp. * ?(\d + )',
message)

#匹配结果如下
[('1114630', '1.00089', '1583990397')]
```

其中发送的数据按照浏览器中的 WebSocket 数据包的格式组装，组装格式如下。组装后需要将其格式化成 JSON 字符串。

```
["{\"_event\":\"bulk-subscribe\",\"tzID\":28,\"message\":\"
pid-1061448:% %                          #需要更新的 pid
pid-1061796                              #需要更新的 pid
:% % cmt-6-5-945629:% % domain-6:\"}"]
```

组装数据时新增了一个 subsection 函数，该函数用来对一个列表进行分割，将一个大列表分割成包含指定长度小列表的列表。比如按照小列表长度 2 分割[1,2,3,4,5,6]，分割结果就是[[1,2],[3,4],[5,6]]，这样做的目的是将 pid 按批次发送给 WebSocket 服务器。

入口函数 run()负责处理整体的异常，出现异常将重新建立 WebSocket 连接，然后继续刷新。可能会出现 WebSocket 服务器主动断开连接的情况。

完整的实现源码如下。

```
import pymongo
import logging
import websocket
from json import dumps
import re

try:
```

```
    import thread
except ImportError:
    import _thread as thread
import time

logger = logging.getLogger('scupdate') #创建一个记录器用于输出控制台
logger.setLevel(logging.INFO)
ch = logging.StreamHandler()
ch.setLevel(logging.INFO)
fmt = logging.Formatter(
    "%(asctime)s - %(filename)s[line:%(lineno)d] - %(levelname)s: %(message)s")
ch.setFormatter(fmt)
logger.addHandler(ch)

url = 'wss://stream55.forexpros.com/echo/698/eaaov7r8/websocket'

heartbeat = ["{\"_event\":\"heartbeat\",\"data\":\"h\"}"]
client = pymongo.MongoClient("mongodb://localhost:27017/")
db = client["fictitious"]
col = db["data"]

def subsection(l: list, n: int):
    """
    列表分割成块
    :param l:
    :param n:
    :return:
    """
    for i in range(0, len(l), n):
        yield l[i:i + n]

def on_message(ws, message):
    """
    有消息时调用
    :param ws:
    :param message:
    :return:
    """
    #logger.info(f"收到消息:{message}")
    if message == 'o':
        query = col.aggregate([{'$group': {'_id': "$pid"}}]) #{"$limit": 200}
        pids = ["pid-" + item['_id'] for item in query if item['_id'] != None]
        for pid in subsection(pids, 100):
            pid_str = ':%%'.join(pid)
            data = "{\"_event\":\"bulk-subscribe\",\"tzID\":28,\"message\":\"" + pid_
str + ":%%cmt-6-5-945629:%%domain-6:\"}"
            ws.send(dumps([data]))
    else:
        search = re.findall(r'"pid.*?(\d+).*last_numeric.*?:(\d*\.?\d*).+
timestamp.*?(\d+)', message)
        if search:
            refresh = search[0]
            col.update_one({'pid': refresh[0]}, {"$set": {'price': refresh[1], "update":
refresh[-1]}})
            logger.info(f"更新:{refresh}")
```

```
def on_error(ws, error):
    """
    发生错误时调用
    :param ws:
    :param error:
    :return:
    """
    logger.exception(error)


def on_close(ws):
    """
    服务器端关闭时调用
    :param ws:
    :return:
    """
    logger.info("关闭 Websocket 连接")


def on_open(ws):
    """
    打开 ws 连接时调用
    :param ws:
    :return:
    """
    logger.info("打开连接,启动心跳线程!")

    def run(*args):
        while True:
            time.sleep(10)
            ws.send(dumps(heartbeat))

    thread.start_new_thread(run, ())


def run():
    while True:
        try:
            websocket.enableTrace(True)      #开启状态监控
            ws = websocket.WebsocketApp(url,
                                        on_message=on_message,
                                        on_error=on_error,
                                        on_close=on_close)
            ws.on_open = on_open             #单独绑定打开时连接
            ws.run_forever()
        except BaseException as e:
            logger.exception(e)

if __name__ == "__main__":
    run()
```

控制台输出的效果如下。

```
2020-03-12 14:29:07,786 - dcupdate.py[line:75] - INFO: 更新:('1071620', '0.002945', '
1583994509')
…
```

5.5 SMTP 协议与 IMAP 协议

SMTP 协议和 IMAP 协议是处理邮件的重要协议，一个负责邮件发送，一个负责邮件的读取。两者在自动化办公方面有着重要应用。在网络爬虫领域，SMTP 可以用于日志的反馈、运行结果的反馈，IMAP 可用于读取邮箱中的验证码，完成自动验证。

5.5.1 SMTP 协议

SMTP 是一个在互联网上传输电子邮件的标准，是一个相对简单的基于文本的协议。在其之上指定了一条消息的一个或多个接收者（在大多数情况下被确认是存在的），然后消息文本会被传输。可以很简单地通过 telnet 程序来测试一个 SMTP 服务器。SMTP 使用 TCP 端口 25，要为一个给定的域名决定一个 SMTP 服务器，需要使用 DNS 的 MX 记录。

在 20 世纪 80 年代早期 SMTP 开始被广泛使用。当时它只是作为 UUCP 的补充，UUCP 更适合用于在间歇连接的机器间传送邮件。相反，发送和接收的机器在持续连线的网络情况下时，SMTP 工作得最好。

由于这个协议开始是基于纯 ASCII 文本的，它在二进制文件上处理得并不好。例如 MIME 的标准被开发来编码二进制文件以使其通过 SMTP 来传输。今天大多数 SMTP 服务器都支持 8 位 MIME 扩展，它使二进制文件的传输变得几乎和纯文本一样简单。

SMTP 是一个“推”的协议，它不允许根据需要从远程服务器上“拉”来消息。要做到这点，邮件客户端必须使用 POP3 或 IMAP。另一个 SMTP 服务器可以使用 ETRN 在 SMTP 上触发一个发送。

5.5.2 IMAP 协议

IMAP 以前称作交互邮件访问协议，是一个应用层协议，用来从本地邮件客户端（如 Microsoft Outlook、Outlook Express、Foxmail、Mozilla Thunderbird）访问远程服务器上的邮件。

IMAP 和 POP3 是邮件访问最为普遍的 Internet 标准协议。事实上所有现代的邮件客户端和服务器都对两者给予支持。IMAP 现在的版本是 IMAP 第 7 版第 3 次修订版。

IMAP 为邮件访问提供了相对于广泛使用的 POP3 邮件协议的另外一种选择。基本上，两者都允许一个邮件客户端访问邮件服务器上存储的信息。

IMAP 特有的功能是支持连接和断开两种操作模式。当使用 POP3 时，客户端只会在一段时间内连接到服务器，直到它下载完所有新信息，客户端才断开连接。在 IMAP 中，只要用户界面是活动的并且在连续下载信息，客户端就会一直连接服务器。对于有很多或者很大邮件的用户来说，使用 IMAP4 模式可以获得更快的响应时间。

IMAP 支持多个客户同时连接到一个邮箱，POP3 协议支持单一用户连接。

IMAP 支持访问消息中的 MIME 部分和部分获取。几乎所有的 Internet 邮件都是以 MIME 格式传输的。MIME 允许消息包含一个树形结构，这个树形结构的叶节点都是单一内容类型，而非叶子节点都是多块类型的组合。IMAP4 协议允许客户端获取任何独立的 MIME 部分和获取信息的一部分或者全部，这些机制使得用户无须下载附件就可以浏览消息内容或者在获取内容的同时浏览消息。

IMAP 支持在服务器保留消息状态信息。通过使用在 IMAP4 协议中定义的标志客户

端可以跟踪消息状态，例如邮件是否被读取、回复或者删除。这些标识存储在服务器，所以多个客户在不同时间访问一个邮箱可以知道其他用户所做的操作。

IMAP 支持在服务器上访问多个邮箱。IMAP4 客户端可以在服务器上创建、重命名或删除邮箱(通常以文件夹形式显现给用户)，还允许服务器提供对于共享和公共文件夹的访问。

IMAP 支持服务器端搜索。IMAP4 提供了一种机制给客户，使客户可以要求服务器搜索符合多个标准的信息。在这种机制下客户端就无须下载邮箱中的所有信息来完成这些搜索。

IMAP 支持一个定义良好的扩展机制。吸取早期 Internet 协议的经验，IMAP 的扩展定义了一个明确的机制，很多对于原始协议的扩展已被提议并广泛使用。无论是使用 POP3 还是 IMAP4 获取消息，客户端均使用 SMTP 协议来发送消息。邮件客户端可能是 POP 客户端或者 IMAP 客户端，但都会使用 SMTP。

IMAP4 也支持明文传输密码。因为加密机制的使用需要客户端和服务器双方的类型一致，明文密码的使用是在一些客户端和服务器类型不同的情况下(例如 Microsoft Windows 客户端和非 Windows 服务器)。使用 SSL 也可以对 IMAP4 的通信进行加密，例如在 SSL 上的 IMAP4 通信，通过 993 端口传输或者在 IMAP4 线程创建的时候声明 STARTTLS。

5.5.3 Python 使用 SMTP 关键接口

对于 SMTP 协议而言最主要的作用是发送邮件，那么它的关键流程就是登录 SMTP 服务器—构建消息—发送消息—结束。下面将以 Python 实现这条流程为例来介绍 SMTP 的核心 API 操作，Python 中实现 SMTP 协议的库是 smtplib，使用命令 pip install smtplib 安装 smtplib 库。

第一步，使用前先初始化一个 SMTP 对象。

```
smtpObj = smtplib.SMTP()
```

第二步，连接 SMTP 服务器，传入服务器地址 host 和端口，端口要看具体 SMTP 邮件服务器提供的端口，对大多数邮件应用来说，端口在【个人中心】→【更多设置】→【SMTP/IMAP 服务】中可见。

```
smtpObj.connect(host, 25)
```

第三步，登录认证，分别使用 Email 用户名和密码。

```
smtpObj.login(email_user, email_pass)
```

第四步，构建邮件消息对象。邮件消息分为纯文本消息、带 HTML 格式的消息、带文件附件的消息、邮件正文中显示图片的 HTML 格式的消息。其中纯文本消息和纯 HTML 消息都是 MIMEText 类实例化的，唯一的区别是纯文本消息指定的邮件类型是 plain，HTML 格式的消息指定的类型是 html；带附件的消息需要先构建附件消息对象 MIMEMultipart，然后分别使用 MIMEText 类构建正文消息和附件消息，通过 MIMEMultipart.attach(MIMEText)与带附件的邮件消息关联起来；对于邮件正文显示图片的 HTML 邮件消息，先构建带附件的消息对象 MIMEMultipart，然后将图片数据构建成

MIMEImage 对象，预先设置 MIMEImage 对象的 ID，再在 HTML 源码中引用这个 ID 就可以将图片插入 HTML 中。

第五步，发送邮件。参数分别是发件人地址、收件人地址、邮件消息对象、其他可选项，前三个是必要的。邮件消息对象 msg 需要使用其他类构建，常用的对象包括文本格式邮件、HTML 格式邮件、附件的邮件，下面将逐一示例。

```
smtpObj.sendmail(from_addr, to_addrs, msg, mail_options=(),
rcpt_options=())
```

以简单的文本消息邮件来演示整个流程，源码如下。

```
import smtplib
from email.mime.text import MIMEText
from email.header import Header
import logging

logger = logging.getLogger('smtp')      #创建一个记录器用于输出控制台
logger.setLevel(logging.INFO)
ch = logging.StreamHandler()
ch.setLevel(logging.INFO)
fmt = logging.Formatter("%(asctime)s - %(filename)s[line:%(lineno)d] - %(levelname)
s: %(message)s")
ch.setFormatter(fmt)
logger.addHandler(ch)

receivers = ['24xxxxxx539@qq.com']      #收件人列表,可设置多个收件人

#构建消息对象,三个参数分别是消息正文、消息类型 plain、正文编码
message = MIMEText('这是正文内容,请查收', 'plain', 'utf-8')

#创建 Email 的头信息,在邮箱上显示发件人和收件人的名字
message['From'] = Header("发件人 name", 'utf-8')
message['To'] = Header("收件人 name", 'utf-8')
message['Subject'] = Header('这是 Email 主题文字', 'utf-8')

try:
    user = '25xxxxx277@qq.com'
    smtpObj = smtplib.SMTP()
    smtpObj.connect('smtp.qq.com', 587)    #QQ 邮箱的 SMTP 的 host 和 SSL 连接端
    smtpObj.login(user, "ouq****b")
    smtpObj.sendmail(user, receivers, message.as_string()) #通过 as_string()将 message 对
                                                           #象构建成字符串格式
except smtplib.SMTPException as e:
    logger.exception(e)
```

发送普通文本类型的邮件在构建消息对象的时候指明邮件类型为 plain，即文本类型。在发送邮件的时候，使用消息对象的 as_string 方法将消息对象转换为文本格式的字符串，服务器收到后会解析其中的信息。注意 QQ 邮箱的 SMTP 服务的 host 地址是 smtp.qq.com，只允许使用 SSL 连接的 587 端口。

图 5-15　纯文本邮件的效果

纯文本邮件的效果如图 5-15 所示。

5.5.4 Python 使用 IMAP 关键接口

尽管 Python 内置的 imaplib 模块提供了 IMAP4 协议的接口，但是其灵活性不如第三方的 imapclient 库，这里以 imapclient 库为例，介绍如何使用 IMAP 协议获取邮件数据，然后使用 pyzmail36 快速对下载的邮件进行解析。

首先使用安装命令 pip install imapclient 安装需要的包，然后使用导入模块命令 from imapclient import IMAPClient。

第一步，传入 IMAP 服务器的 host 地址和端口，同时指定 SSL 连接和 timeout，初始化 IMAP 对象。

```
imapobj = IMAPClient('imap.qq.com', port = 993, ssl = True, timeout = 30)
```

第二步，登录认证，使用 IMAP 对象的 login 方法登录服务器。

```
imapobj.login(user, password)
```

第三步，选择要处理的邮箱文件夹。

```
imapobj.select_folder(name) #默认选择主邮箱文件夹 INBOX,即收件箱
```

第四步，搜索要处理的邮件，搜索使用 search 方法，返回邮件的 UID。criterion 是搜索条件，如果为 ALL，则返回全部邮件的 UID。

```
UID = imapobj.search('ALL')
```

第五步，下载邮件，使用 fetch()下载给定的 UID 或 UID 列表的正文内容。第一个参数是 UID 或 UID 列表，第二个参数是下载的文件的内容，一般为['BODY[]']，即正文。

```
messages = imapobj.fetch(UID, ['BODY.PEEK[]'])
```

获取到的内容是原始邮件字符列表或者单个原始邮件，单个原始邮件的内容如下源码所示。其中 BODY[]对应的值，就是在 SMTP 邮件中构建的 MIMEMultipart 或 MIMEText 消息对象转换成字符串格式的结果。这里要将这些字符串再转换为消息对象。

```
defaultdict(dict,
            {1064: {b'SEQ': 5,
                    b'BODY[]': b'Received: from \r\n\r\n\r\n\r\n\r\n—… b1 _
f4dd72807519f61528f30b0bf26c61ed--\r\n\r\n\r\n'}})
```

第六步，字符串格式转为消息对象。使用三方库 pyzmail36 的 pyzmail.PyzMessage.factory 方法，可以快速生成解析邮件对象 PyzMessage，快速获取邮件的对象属性。

```
# pip install pyzmail36     #适用于 python3.6/3.7/3.8
# pip install pyzmail39     #适用于 python3.9/3.10
import pyzmail
message = pyzmail.PyzMessage.factory(BODY)
```

第七步，解析 PyzMessage 对象，PyzMessage 提供了便捷的 message.get_address()或 message.get_addresss()方法来获得收件人、发件人、抄送人的信息，还提供了 text_part 属性和 html_part 属性来区分文本邮件和 HTML 格式邮件。对于有附件的邮件，附件是 PyzMessage 对象的可解析 MailPart 对象，附件的 MailPart 对象提供了直接获取附件名的

get_filename 方法和获取附件内容的 message.get_payload 方法。

```
message.get_address('to')      #获取收件人
message.get_address('cc')      #获取抄送人
message.get_address('from')    #获取发件人
```

第八步，移动或删除邮件，先将邮件进行删除标记，然后使用删除方法。

```
imapobj.delete_messages(IDS)      #标记删除
imapobj.copy(IDS, filename)       #把邮件复制到 filename 文件夹
imapobj.expunge()                 #永久删除
```

第九步，断开连接。

```
imapobj.logout()
```

上述完整流程的源码如下。

```
from imapclient import IMAPClient
import pyzmail

imapobj = IMAPClient('imap.qq.com', port = 993, ssl = True, timeout = 30)
imapobj.login('2xxxxxx7@qq.com', 'ouq*** bab')      #连接至 IMAP 服务器
imapobj.select_folder("INBOX")                      #进入默认收件箱
ids = imapobj.search()                              #获得全部邮件的 ID
id = ids[ - 1]                                      #获得最新一封邮件
emaildata = imapobj.fetch(id, ['BODY.PEEK[]'])      #下载 ID 的邮件内容
message = pyzmail.PyzMessage.factory(emaildata[id][b"BODY[]"].decode())
#使用第三方解析格式化库
print(message.get_address('to'))                    #收件人
print(message.get_address('cc'))                    #抄送人
print(message.get_address('from'))                  #发件人

message.text_part.get_payload()                     #获取文本邮件的正文
message.html_part.get_payload()                     #获取 HTML 格式邮件的正文

messages = message.mailparts #获取邮件解析对象列表，包括附件、正文.返回一个 MailPart 对象
                             #组成的列表

#MailPart 是 message 的解析对象，单独将正文、附件作为一个解析邮件处理
MailPart.get_filename()                            #如果 MailPart 是附件，将获得附件文件名
MailPart.get_payload()                             #如果 MailPart 是附件，将获得附件数据

imapobj.copy(id, filename)                         #将邮件复制到 filename 文件夹
imapobj.delete_messages(id)
imapobj.expunge()                                  #永久删除邮件
```

下面会用具体的案例来演示 IMAP 协议在各个场景中的使用方法。

5.5.5　案例一：发送 HTML 格式的邮件

HTML 格式的消息和文本类型的消息使用流程基本一致，只是在构建消息对象时正文传输应该是 HTML 源码字符串，正文类型由 plain 改为 html。发送 HTML 格式邮件的测试源码如下。

```
import smtplib
from email.mime.text import MIMEText
from email.header import Header
import logging

logger = logging.getLogger('smtp')      #创建一个记录器用于输出控制台
logger.setLevel(logging.INFO)
ch = logging.StreamHandler()
ch.setLevel(logging.INFO)
fmt = logging.Formatter(
    "%(asctime)s - %(filename)s[line:%(lineno)d] - %(levelname)s: %(message)s")
ch.setFormatter(fmt)
logger.addHandler(ch)

receivers = ['24xxxxxx39@qq.com']              #收件人列表,可设置多个收件人
#构建消息对象,三个参数分别是消息正文、消息类型 plain、正文编码
msg = """
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>一封 Email</title>
</head>
<body>
<div style="background-color: brown">
    <a href="http://www.likeinlove.com">
        这是一封 Email 邮件
    </a>
</div>
</body>
</html>
"""
message = MIMEText(msg, 'html', 'utf-8')
#创建 Email 的头信息,在邮箱上显示发件人和收件人的名字
message['From'] = Header("发件人 name", 'utf-8')
message['To'] = Header("收件人 name", 'utf-8')
message['Subject'] = Header('这是 Email 主题文字', 'utf-8')

try:
    user = '255xxxxx77@qq.com'
    smtpObj = smtplib.SMTP()
    smtpObj.connect('smtp.qq.com', 587)          #QQ 邮箱的 SMTP 的 host 和 SSL 连接端
    smtpObj.login(user, "o****ab")
    smtpObj.sendmail(user, receivers, message.as_string())    #通过 as_string()将 message
                                                              #对象构建成字符串格式
except smtplib.SMTPException as e:
    logger.exception(e)
```

当然邮件正文可以只是 HTML 文档的一些片段。HTML 格式邮件的效果如图 5-16 所示。

图 5-16 HTML 格式邮件的效果图

5.5.6　案例二：发送带附件的邮件

要发送带附件的邮件，除了邮件消息的构建之外，其他操作同一般文本邮件一致。构建带附件的邮件对象使用 MIMEMultipart 类，这个对象一样具有发件人、收件人、主题，但是在处理邮件正文和邮件附件的时候有差异。邮件正文使用正常的 MIMEText 类构建，不过要通过 MIMEMultipart 对象的 attach 方法与之关联。邮件附件也使用 MIMEText 类构建，但是在类型选择上有差异，还需设置 MIMEText 对象的两个额外字段 Content-Type 和 Content-Disposition，然后通过 MIMEMultipart 对象的 attach 方法与之关联。带附件消息对象的实现源码如下。

```
import smtplib
from email.mime.text import MIMEText
from email.header import Header
from email.mime.multipart import MIMEMultipart
import logging

logger = logging.getLogger('smtp')      #创建一个记录器用于输出控制台
logger.setLevel(logging.INFO)
ch = logging.StreamHandler()
ch.setLevel(logging.INFO)
fmt = logging.Formatter(
    "%(asctime)s - %(filename)s[line:%(lineno)d] - %(levelname)s: %(message)s")
ch.setFormatter(fmt)
logger.addHandler(ch)

receivers = ['243xxxxx39@qq.com']      #收件人列表,可设置多个收件人
#构建消息对象,三个参数分别是消息正文、消息类型 plain、正文编码
msg = """
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>一封 Email</title>
</head>
<body>
<div style="background-color: brown">
    <a href="http://www.likeinlove.com">
        这是一封 Email 邮件
    </a>
</div>
</body>
</html>
"""
#创建一个带附件的实例
message = MIMEMultipart()
message['From'] = Header("发件人 name", 'utf-8')
message['To'] = Header("收件人 name", 'utf-8')
message['Subject'] = Header('这是 Email 主题文字', 'utf-8')
#构建邮件正文内容
message.attach(MIMEText(msg, 'html', 'utf-8'))

#构造图片附件
png = MIMEText(open('1.png', 'rb').read(), 'base64', 'utf-8')
png["Content-Type"] = 'application/x-png' #内容类型为 png
```

```
    #filename 是显示的附件名
    png["Content-Disposition"] = 'attachment; filename="1.png"'
    message.attach(png)
    #构造文本附件
    txt = MIMEText(open('1.txt', 'rb').read(), 'base64', 'utf-8')
    txt["Content-Type"] = 'application/octet-stream'
    txt["Content-Disposition"] = 'attachment; filename="1.txt"'
    message.attach(txt)
    try:
        user = '255xxxxx77@qq.com'
        smtpObj = smtplib.SMTP()
        smtpObj.connect('smtp.qq.com', 587) #QQ 邮箱的 SMTP 的 host 和 SSL 连接端
        smtpObj.login(user, "o*****ab")
        smtpObj.sendmail(user, receivers, message.as_string())      #通过 as_string()将 message
                                                                    #对象构建成字符串格式
    except smtplib.SMTPException as e:
        logger.exception(e)
```

附件的两个参数 Content-Type 和 Content-Disposition 都是 HTTP 协议的请求头字段，Content-Type 用于指示内容是什么类型，Content-Disposition 用于指示内容以什么方式展示。附件的编码选择 base64，这是网络传输常用的编码格式。

常用的 Content-Type 值如下。

```
image/jpeg:jpeg 图片
application/x-jpg:jpg 图片
application/x-png:png 图片
application/pdf:pdf 文件
text/plain:txt 文本
application/msword:doc 文档
application/x-xls:xls 文档
application/x-ppt:PPT 文档
```

常用的 Content-Disposition 格式如下。

```
inline:默认值，表示回复中的消息体会以页面的一部分或者整个页面的形式展示
attachment:消息体应该被下载到本地
attachment; filename="filename.jpg":消息体下载到本地，filename 是下载文件名
```

发送带附件的邮件，收件箱收到的带附件邮件的效果如图 5-17 所示。

图 5-17 收件箱中带附件邮件的效果

5.5.7　案例三：发送显示图片的 HTML 格式的邮件

构建正文带图片的邮件的消息对象和构建带附件的邮件消息对象的步骤基本一致，但有细微差别。首先都是构建带附件的消息对象 MIMEMultipart，然后创建 HTML 类型的 MIMEText 对象，使用 MIMEMultipart 对象的 attach 方法将之与 MIMEText 对象关联，就相当于添加了正文。正文中需要的图片要预先读出数据构建 MIMEImage 对象，然后再设置 MIMEImage 对象的 ID，使用 MIMEMultipart 对象的 attach 与之关联起来，接着在邮件正文的 HTML 文档中引用设置的 ID 即可实现插入图片。构建 MIMEMultipart 对象的源码如下。

```
import smtplib
from email.mime.image import MIMEImage
from email.mime.text import MIMEText
from email.header import Header
from email.mime.multipart import MIMEMultipart
import logging
logger = logging.getLogger('smtp')      #创建一个记录器用于输出控制台
logger.setLevel(logging.INFO)
ch = logging.StreamHandler()
ch.setLevel(logging.INFO)
fmt = logging.Formatter(
    "%(asctime)s - %(filename)s[line:%(lineno)d] - %(levelname)s: %(message)s")
ch.setFormatter(fmt)
logger.addHandler(ch)

receivers = ['243xxxx539@qq.com']      #收件人列表,可设置多个收件人
#构建消息对象,三个参数分别是消息正文、消息类型 plain、正文编码
msg = """
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>一封 Email</title>
</head>
<body>
<div style="background-color: brown">
    <a href="http://www.likeinlove.com">
        这是一封 Email 邮件
    </a>
    <img src="cid:likeinlove.com" alt="">
</div>
</body>
</html>
"""
#创建一个带附件的实例
message = MIMEMultipart()
message['From'] = Header("发件人 name", 'utf-8')
message['To'] = Header("收件人 name", 'utf-8')
message['Subject'] = Header('这是 Email 主题文字', 'utf-8')
#构建邮件的正文内容
message.attach(MIMEText(msg, 'html', 'utf-8'))
#构造 HTML 文档中的图片
with open('1.png', 'rb') as f:
    img = MIMEImage(f.read())
```

```
img.add_header('Content-ID', '<likeinlove.com>')
message.attach(img)
try:
    user = '255xxxxx77@qq.com'
    smtpObj = smtplib.SMTP()
    smtpObj.connect('smtp.qq.com', 587)      #QQ邮箱的SMTP的host和SSL连接端
    smtpObj.login(user, "ouqpccttonnhebab")
    smtpObj.sendmail(user, receivers, message.as_string())      #通过as_string()将message
                                                                #对象构建成字符串格式
except smtplib.SMTPException as e:
    logger.exception(e)
```

在HTML中引用设置的ID时使用cid:ID格式，如上面源码中MIMEImage对象设置的ID是<likeinlove.com>，用尖括号标识，引用时使用<img src="cid:likeinlove.com"alt=""">，和使用URL地址类似。

收件箱的效果如图5-18所示。

图5-18 收件箱中HTML格式的邮件显示图片

5.5.8 案例四：自动读取邮箱验证码

这里以读取GitHub登录的邮箱验证码为例，将使用关键接口实现从指定的邮箱文件夹查找目标邮件、从目标邮件正文中获取验证码、获取验证码后删除邮件。这里只实现获取验证码部分，整体的登录在后面的章节中会完全实现。

GitHub登录地址是https://github.com/login，在登录界面输入账号和密码后会进行登录环境检测，如果是不安全登录环境或不常用登录地址都需进行邮箱验证。如果没有出现邮箱验证，可以登录后在Account settings→Account security选项中撤销常用设备的session，然后重新登录即可见验证码文本框。

这里将使用Python通过IMAP协议获取验证码，源码如下。

```
from imapclient import IMAPClient
import pyzmail
import re
from datetime import date
import logging

logger = logging.getLogger('imap')    #创建一个记录器用于输出控制台
logger.setLevel(logging.INFO)
ch = logging.StreamHandler()
```

```
ch.setLevel(logging.INFO)
fmt = logging.Formatter("%(asctime)s - %(filename)s[line:%(lineno)d] - %(levelname)
s: %(message)s")
ch.setFormatter(fmt)
logger.addHandler(ch)


def run():
    imapobj = IMAPClient('imap.qq.com', port=993, ssl=True, timeout=30)
    imapobj.login('25xxxxx7@qq.com', 'oxxxxxxxxxb')
    logger.info(imapobj.list_folders())                    #打印邮箱文件夹列表
    imapobj.select_folder("其他文件夹/邮件归档")           #进入 Github 邮件所在的文件夹
    ids = imapobj.search([u'SINCE', date.today()])         #获得今天收到的未读邮件
    logger.info(f"今天收到邮件的 ID 列表:{ids}")
    id = ids[-1]                                           #选择最新收到的邮件
    data = imapobj.fetch(id, ['BODY.PEEK[]'])              #下载邮件内容
    message = pyzmail.PyzMessage.factory(data[id][b"BODY[]"].decode()) #使用第三方解析
                                                                       #格式化库
    logger.info(f"收件人{message.get_address('to')}")      #收件人
    logger.info(f"抄送{message.get_address('cc')}")        #抄送人
    logger.info(f"发件人{message.get_address('from')}")    #发件人
    text = message.get_payload()                           #获取 HTML 格式的邮件正文
    item = re.search('code: (\d{6})', text).group(1)
    logger.info(f"验证码是{item}")
    imapobj.delete_messages(id)                            #delete
    imapobj.expunge()                                      #save delete
    logger.info("邮件删除成功@")
    imapobj.logout()
    return item


if __name__ == '__main__':
    try:
        run()
    except BaseException as e:
        logger.exception(e)
```

为了方便阅读代码和理解流程，上面的源码是按照逻辑从上到下书写的。实际使用时往往分步实现，比如登录、获取、验证、删除可能面对不一样的异常情况。

源码中通过 imapobj.list_folders 方法打印邮箱内的文件夹名，文件夹名与应用中看到的名字可能会有差别，例如“收件箱”的名字是 INBOX，是默认的邮件存放文件夹；“邮件归档”显示为“其他文件夹/邮件归档”。需要通过正确的邮件文件夹名才能进入指定的文件夹，从而获取该文件夹下的邮件。

通过 message.get_payload 方法，获取文本类型的邮件正文。如果是 HTML 格式的邮件正文，需要处理一下编码，下面将获取 HTML 格式邮件的正文，并使用它标记的字符集来解码。

```
message.html_part.get_payload().decode(message.html_part.charset)
```

获取验证码使用的是正则匹配，如果邮件是 HTML 格式的，还可以使用 lxml 来解析 HTML 的文档节点。删除邮件分为两步，第一步是将指定邮件标记为删除，第二部是永久删除邮件。delete_messages 方法是将指定邮件标记为删除，expunge 方法将永久删除这些有删除标记的邮件。

5.6 Robots 协议

1. Robots 协议简介

Robots 协议不是一个标准，而是约定俗成的，是一个君子协议，只有在大家都遵守的情况下才有用。Robots 协议约定的内容写在 robots.txt 文本中，存放于网站根目录下，是 ASCII 编码的文本文件。它通常告诉网络搜索引擎的漫游器（又称网络蜘蛛），此网站中的哪些内容是不应被搜索引擎的漫游器获取的，哪些是可以被漫游器获取的。因为一些系统中的 URL 会区分大小写，所以 robots.txt 的文件名应统一为小写字母。robots.txt 应放置于网站的根目录下。

2. Robots 规则

Robots 协议的主要内容写在 robots.txt 文本文件中，主要有四个关键词，分别是 User-agent、Disallow、Allow、Sitemap，分别用于告诉爬虫哪些搜索引擎爬虫可访问、爬虫禁止访问的目录、爬虫允许访问的目录、网站地图地址。配置目录时支持通配符和指定格式，具体案例如下。

```
User-agent: * 这里的*代表所有的搜索引擎种类，*是一个通配符
Disallow: /admin/ 禁止爬寻 admin 目录下面的目录
Disallow: /cgi-bin/*.htm 禁止访问/cgi-bin/目录下的所有以".htm"为后缀的 URL(包含子目录)
Disallow: /*?* 禁止访问网站中所有包含问号的网址
Disallow: /.jpg$ 禁止抓取网页中所有 jpg 格式的图片
Disallow:/ab/adc.html 禁止爬取 ab 文件夹下面的 adc.html 文件
Allow: /cgi-bin  允许爬寻 cgi-bin 目录下面的目录
Allow: /tmp 允许爬寻 tmp 的整个目录
Allow: .htm$ 仅允许访问以".htm"为后缀的 URL
Allow: .gif$ 允许抓取网页和 gif 格式的图片
Sitemap: 网站地图 告诉爬虫这个页面是网站地图
```

5.7 安全与会话机制

本章主要介绍客户端与服务器端交互中的常用安全机制。CSRF（Cross-site request forgery，跨站请求伪造）常用于表单的验证，防止跨域攻击；Cookie 用于在客户端存储用户状态；session 用于服务器端保持与客户端的会话状态；Token 用于临时的有效期限内的用户身份标识。这些机制广泛应用于前后端通信的场景中。对于爬虫而言，这些机制是爬虫请求数据能否成功的关键，只有按照前后端的安全机制才能模拟出真实的客户端，从而获取有效的信息。

5.7.1 CSRF 攻击与保护

1. XSRF 攻击

跨站请求伪造（Cross-site request forgery，缩写为 CSRF 或者 XSRF）也被称为 one-click attack 或者 session riding，是一种挟制用户在当前已登录的 Web 应用程序上执行非本意的操作的攻击方法。跟跨网站脚本（XSS）相比，XSS 利用的是用户对指定网站的信任，CSRF 利用的是网站对用户网页浏览器的信任。

攻击者通过一些技术手段欺骗用户的浏览器去访问一个曾经认证过的网站并运行一些

操作，简单的身份验证只能保证请求发自某个用户的浏览器，却不能保证请求本身是用户自愿发出的。防止 CSRF 攻击的常用方式有 HTTP 头的 Referer 字段检测，这个字段用来标明请求来源于哪个地址，在每个表单页请求中加入一个随机值的校验 Token(令牌)，用户提交表单数据对 Token 进行校验。通过 Referer 请求头检测，无法有效判断消息来源是否可靠，因为这个值可以被修改。通过 Token 校验只能保证提交的数据是来自表单页的，爬虫可以先请求表单页，提取出其中的 Token 字段再和其他表单数据一起发送给服务器，但也无法判断消息来源是否是真实用户。

2. XSRF 保护

浏览器有一个很重要的概念是同源策略(Same-Origin Policy)。同源是指域名、协议、端口相同，不同源的客户端脚本(JavaScript、ActionScript)在没明确授权的情况下，不能读写对方的资源。

由于第三方站点没有访问 Cookie 数据的权限(同源策略)，所以可以在每个请求中包括一个特定的参数值作为令牌来匹配存储在 Cookie 中的对应值，如果两者匹配，后台处理程序认定请求有效。伪造的请求可以使用 Cookie，但是无法获取对应的令牌，因此能防御 XSRF 攻击。

常规意义的防 CSRF 攻击给爬虫带来的困难有限，只需要爬虫在提交数据前请求表单页，并获取页面或者 Cookie 中的 Token 值。但是专门针对爬虫的 Token 值却不是这么简单的，往往这个值是通过 JavaScript 动态生成，这样即使爬虫先请求了表单页，爬虫也无法从页面中提取出 Token。更复杂的表单页的 Token 不是后台随机生成的，而是用户提交表单时 JavaScript 代码根据一系列复杂的算法临时生成的，将其传到网站后台再根据一系列算法解密校验。这就给爬虫带来了极大的伤害，必须解密 Token 的生成算法才能伪造请求，不然就只能通过控制浏览器的方式发送请求，这有效地降低了爬虫效率。

5.7.2 CSRF 验证过程

这里以 Python 的 Tornado 框架为例来演示前后端是如何防止 CSRF 攻击的。在第 3 章中，首次使用 Tornado 开发了一个简单的日志服务平台，在创建 Tornado 应用时指定了一个参数 xsrf_cookies=False，意思是不开启 CSRF 验证，下面来看看为什么不能开启这个参数，开启了这个参数会发生什么情况。

Tornado 是 MVC(Model View Controller)框架，是模型(model)—视图(view)—控制器(controller)的缩写，视图是经过控制器渲染后返回的，因此还需要创建一个 HTML 文档模板文件 xsrf.html 文件，然后输入下面的源码。

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>XFRS 防御流程</title>
</head>
<body>
<form method="post" action="/test">
    {% module xsrf_form_html() %}
    <input type="text" name="msg"/>
    <input type="submit" value="提交"/>
```

```
</form>
</body>
</html>
```

xsrf.html 模板的目的是交给 Tornado 渲染后返回给浏览器，然后用户才能输入。控制器会自动将 HTML 源码中的“{% module xsrf_form_html() %}”替换成名字是_xsrf 的表单，value 值是类似 2|116fb6e0|d12fe87ec40268202bfbbb1267ebea9a|1584087183 这样的字符串，其在用户提交表单的时候自动提交。新建一个 xsrf.py 文件写入下面的 Python 代码。

```
import tornado.web
import tornado.ioloop
import tornado
import json
import logging

class Xsrf(tornado.web.RequestHandler):

    def get(self):
        self.render('xsrf.html')          #将模板渲染后返回

    def post(self):
        post_data = self.request.body_arguments
        post_data = {x: post_data.get(x)[0].decode("utf-8") for x in post_data.keys()}
        if not post_data:
            post_data = self.request.body.decode('utf-8')
            post_data = json.loads(post_data)
        logging.info(post_data)
        return self.write(json.dumps(post_data))


if __name__ == "__main__":
    app = tornado.web.Application([
        (r"/test", Xsrf),
    ],
        xsrf_cookies=True,
        debug=True,
        reuse_port=True
    )
    app.listen(5006)
    tornado.ioloop.IOLoop.current().start()
```

该 Web 服务只有一条路由记录对应一个 URL 地址，分别实现了该 URL 上的 GET 请求和 POST 请求，GET 请求会返回经过渲染的表单，POST 请求会打印提交的表单内容。

运行该 Web 服务，在浏览器中输入 http://localhost:5006/test，服务器返回一个表单数据，表单数据中有一个隐藏的 input 框，同时 Cookie 中有一个 csrftoken 字段。

HTML 渲染后的源码如下，增加了一个名为_xsrf 的隐藏文本框，这是模板中的{% module xsrf_form_html()%}语法渲染的。

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>XFRS 防御流程</title>
```

```
</head>
<body>
<form method="post" action="/test">
    <input type="hidden" name="_xsrf" value="2|752ebd92|b56ee30ca04363524fbab06003aae1e8|
1584087183"/>
    <input type="text" name="msg"/>
    <input type="submit" value="提交"/>
</form>
</body>
</html>
```

再看本地 Cookie 情况，Cookie 增加了两条和 xsrf 字段相关的值，结果如下所示。_xsrf 表单中的值和文本框中的值一样，为了给 Ajax 请求使用 Cookie 验证 XSRF，csrftoken 值用于在服务器检验_xsrf 的值。

```
csrftoken:FKHPjn1lusiYfZsVfQLZ2sjOK6ZTLQgwTWc7hFmiGzsOQihNxabwiOHh24VpSWGy
_xsrf:2|0f8170c9|cfc12e57daecae0935157d3b79052cb3|1584087183
```

在文本框中输入 XSRF 然后提交数据，页面返回的表单内容如下。

```
{"_xsrf": "2|752ebd92|b56ee30ca04363524fbab06003aae1e8|1584087183", "msg": "XSRF"}
```

这个过程看似复杂，其实对于开发者而言是不必关注的，这一切都会由网络框架自动完成，开发者只需要启用这个功能即可。

这就是 CSRF 的验证流程：通过在表单、Cookie 中设置相关的校验参数，阻止跨域访问。对于爬虫而言，就必须要先去请求表单页，获取其中的_xsrf 和 csrftoken 参数，将其和表单数据一起提交，请求才能通过校验。

5.7.3　Cookie 机制

1. Cookie 简介

Cookie 的复数形态是 Cookies，又被称为“小甜饼”，其类型为“小型文本文件”，指某些网站为了辨别用户身份而存储在用户本地终端(Client Side)上的数据，通常经过加密。

HTTP 协议是无状态的，即服务器不知道用户上一次做了什么，这严重阻碍了交互式 Web 应用程序的实现。在典型的网上购物场景中，用户浏览了几个页面，买了一盒饼干和两瓶饮料，最后结账时，由于 HTTP 的无状态性，不通过额外的手段，服务器并不知道用户到底买了什么，所以 Cookie 就是用来绕开 HTTP 的无状态性的特殊手段之一。服务器可以设置或读取 Cookie 中包含的信息，借此维护用户跟服务器会话中的状态。

在刚才的购物场景中，当用户选购了第一件商品后，服务器在向用户发送网页的同时还发送了一段 Cookie，Cookie 记录着那件商品的信息。当用户访问另一个页面时，浏览器会把 Cookie 发送给服务器，于是服务器知道用户之前选购了什么。用户继续选购饮料，服务器就在原来那段 Cookie 里追加新的商品信息，结账时服务器读取发送来的 Cookie 就行了。

Cookie 另一个典型的应用是当登录一个网站时，网站往往会要求用户输入用户名和密码，并且用户可以勾选【下次自动登录】选项。如果勾选了，那么下次访问同一网站时，用户会发现没输入用户名和密码就已经登录了。这是因为在前一次登录时，服务器发送了包含登录凭据(用户名加密码的某种加密形式)的 Cookie 到用户的硬盘上。用户第二次登录时，如果该 Cookie 尚未到期，浏览器会发送该 Cookie，服务器验证凭据，于是不必输入用户名和

密码就让用户登录了。

2. Cookie 的组成

打开浏览器，按 F12 键打开开发者调试工具，单击 Application 面板，在 Storage 栏下有一项 Cookies 选项，Cookies 选项存放着当前网站的 Cookie 信息，如图 5-19 所示。

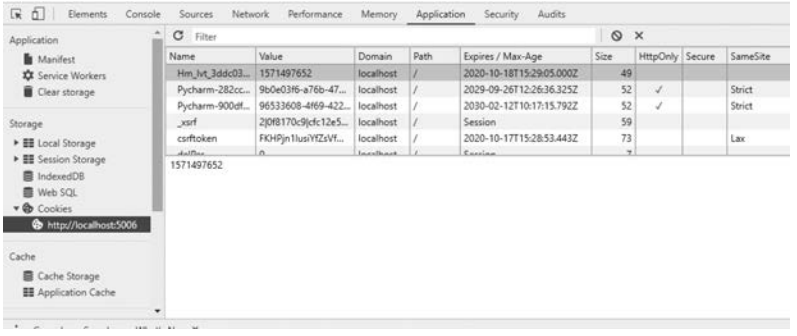

图 5-19　当前网站的 Cookie 信息

对于常用的 Chrome 浏览器而言，在 Chrome 浏览器中产生的 Cookie 存放在浏览器本地的 SQLite 数据库中，Chrome 浏览器统一加载和管理 Cookie 信息。

单条 Cookie 值具有表 5-5 中所示的属性。Web 服务器设置客户端的 Cookie，通过在服务器端返回浏览器的 HTTP 响应中设置 Set-Cookie 请求头，浏览器将解析出 Set-Cookie 对应的值并写入本地的 Cookie 数据库中。如果要删除 Cookie，只需要将 Cookie 的过期时间设置在当前时间之前，浏览器将删除该条 Cookie。在客户端发送 HTTP 请求时，浏览器将自动检测本地的 Cookie 数据，将满足条件的 Cookie 按照 name＝value 并以分号连接成字符串，附加在 HTTP 的 Cookie 请求头中。

表 5-5　Cookie 中的字段及含义

参　数　名	说　　明
Name	Cookie 名称
Value	Cookie 值
Domain	指定了可以访问该 Cookie 的 Web 站点或域名，允许一个子域可以设置或获取其父域的 Cookie
Path	指定了 Web 站点上可以访问该 Cookie 的目录
Expires/Max-Age	Cookie 的有效期，可以是时间戳整数、时间元组或者 datetime 类型，格式为 UTC 时间
Size	Cookie 的大小，一般不超过 4KB
HttpOnly	防止客户端脚本通过 document.Cookie 属性访问 Cookie，有助于保护 Cookie 不被跨站脚本攻击窃取或篡改
Secure	指定是否使用 HTTPS 安全协议发送 Cookie。使用 HTTPS 安全协议，可以保护 Cookie 在浏览器和 Web 服务器间的传输过程中不被窃取和篡改

Cookie 并不是安全的，因为 Cookie 并没有和客户端绑定，使用相同 Cookie 的客户端具有相同的权限，并且 Cookie 是可以被篡改的。假设一用户登录网站后，后台发送给浏览器一个标识登录状态的 Cookie，并将 Cookie 的有效期设为一周，如果用户修改了该 Cookie 的有效期，是不是就能永远保持登录状态了？如果没有其他措施，理论上可以实现永久登录状态，为了处理这一弊端又引入了 Session 和 Token，这是后两节将讲述的内容。

5.7.4 会话

Session(会话)对象存储特定用户会话所需的属性及配置信息,当用户在应用程序的Web页之间跳转时,存储在Session对象中的变量不会丢失,而是在整个用户会话中一直存在下去。当用户请求来自应用程序的Web页时,如果该用户还没有会话,则Web服务器将自动创建一个Session对象。当会话过期或被放弃后,服务器将终止该会话,Session对象常用于在服务器端保持会话状态。

在5.7.3节中说过,Cookie是不安全的、可被篡改的。那么服务器如何才能保证对应Cookie信息的有效性?如果在服务器端有一条该Cookie相关的记录信息,那么是不是就能保证该Cookie的真实性?这就是Session主要解决的问题。

Session解决了Cookie安全性的问题,但是又带来了新的问题:如果每个连接的用户都在后台存储一个对应的Session信息,随着用户的增长,服务器成本会越来越高,应用程序的性能会越来越差。为了解决新的问题,又引入了新的方案和优化措施,比如用Redis缓存Session、优化Session的大小和生命周期,最重要的方案是使用Token,这也是5.7.5节将讲解的。

Session对爬虫的阻碍作用并不大,因为Session的会话跟踪主要是通过HTTP请求中的Cookie信息来存取会话状态,爬虫一般携带了正常的Cookie信息。Session是客户端不可控的,只能通过客户端的相应操作来改变Session的相关状态。

5.7.5 Token与JWT

Token是服务器端分配给客户端的身份令牌,是由用户唯一的身份标识、当前的时间戳、签名经过一系列加密后返回给客户端用于身份校验的字符串,Token存储在客户端中,服务器端只负责校验Token的有效性。

Token可以解决Cookie容易被篡改、Session消耗资源多的问题。将一串含有身份信息的加密字符串放到客户端中,这样不必消耗大量的空间来存放Session,只负责Token的计算校验。一般Token用于标识用户的登录状态,具有无状态、可扩展、安全的特点。目前与Token相关的标准是JWT(JSON Web Token),这是一种基于JSON格式用以产生访问令牌的开源标准。

JWT由三部分组成:头部(Header)、载荷(Payload)、签名(Signature)。这三部分分别由不同的格式组成,头部和载荷是JSON串,签名是字符串。头部和载荷分别使用Base64编码然后用来计算签名,最后同签名使用“.”连接成原始Token字符串即Header.Payload.Signature,可以在原始Token字符串的基础上进行加密。

1. 创建Header

typ表明数据是JWT类型的,alg表示使用了HMAC-SHA256算法来创建JWT签名。

```
{
    "typ": "JWT",
    "alg": "HS256"
}
```

2. 创建Payload

Payload用于保存JWT数据的部分,也称为JWT的声明。

```
{
"UID":"xxxx-xxxx-xxxx-xxxx-xxxx"
}
```

3. 计算 Signature

首先将创建的 Header 和 Payload 分别使用 Base64 编码，然后使用“.”连接起来，再将拼接起来的字符串使用 Header 中声明的算法加上密钥进行加密。

```
Signature = Hash(base64urlEncode(Header) + "." + base64urlEncode(Payload), Key)
```

4. 得到 Token

最后将 Base64 编码的 Header 和 Base64 编码的 Payload 及 Signature 用“.”连接起来，这就是 JWT 标准产生的 Token 字符串。

5. 校验

当用户使用附带 JWT 的请求调用接口时，后台应用将 Base64 编码的 Header 和 Payload 拿来计算一遍，比较其与 JWT 中附带的签名是否一致，如果签名一致，则表示 JWT 是有效的，否则该 JWT 无效。

5.7.6 案例：获取本地 Chrome 浏览器中的任意 Cookie 信息

下面的案例将从正常的 Chrome 浏览器中获取到存储的 Cookie 信息，这不是通过 Selenium 的方式，而是通过访问 Chrome 浏览器存储在本地的 SQLite 数据库获得 Cookie 数据，通过该方式可以获得存储在本地浏览器中的所有域名的 Cookie。

这里使用的是 browser_cookie3 库，支持从 Chrome、Firefox、Opera、Edge 和 Chromium 浏览器中获取指定域名的 Cookie 信息，返回的数据是 CookieJar 对象。CookieJar 对象是常用的 Cookie 信息对象，可直接用于使用 requests 库产生的请求中，其项目地址参见附录。

使用 browser_cookie3 库之前先运行下列安装命令，tldextract 用于从 URL 字符串中提取域名字符串，这两个库均支持 Python 3.7 版本。

```
pip install browser-cookie3
pip install tldextract
```

下面使用 browser_cookie3 从本地 Chrome 浏览器中获取百度网址的 Cookie，然后请求个人百度主页，并从中提取出主页设置的名称。运行下面的代码前，需要手动打开 Chrome 浏览器然后登录个人的百度账号，这样浏览中才有验证过的 Cookie 信息。

```
import browser_cookie3
import requests
import tldextract
import logging

logger = logging.getLogger('cookie')    #创建一个记录器用于输出控制台
logger.setLevel(logging.INFO)
ch = logging.StreamHandler()
ch.setLevel(logging.INFO)
fmt = logging.Formatter("%(filename)s[line:%(lineno)d] - %(levelname)s: %(message)s")
ch.setFormatter(fmt)
```

```
logger.addHandler(ch)

def chromecookie(url):
    domain = tldextract.extract(url).registered_domain
    cookie = browser_cookie3.chrome(domain_name = domain)
    logger.info(f"获取目标 cookie {cookie}")
    return cookie

def get_baidu_name():
    url = "https://www.baidu.com/my/index"
    cookie = chromecookie(url)
    response = requests.get(url, cookies = cookie)
    name = response.text.split('"username":')[ - 1].split(',')[0]
    logger.info(name)

if __name__ == '__main__':
    get_baidu_name()
```

运行上述代码，控制台的输出日志如下。

```
getchrome.py[line:18] - INFO: 获取目标 cookie < CookieJar[< Cookie Hm_lvt_ad52b3 …
getchrome.py[line:27] - INFO: "我 ***** 2"
```

该库的作用是实现完全无痕的浏览器交互模式，通过其他方式完成严格的登录流程比如使用图色识别交互、其他自动化工具交互，然后使用 browsercookie 获取登录后的 Cookie 信息，这是一个通用的解决方案，用来应对最为严格的登录防护措施。