第四章：网络编程

计算机网络将不同的具有独立功能的多台计算机，通过通信线路连接起来；并在操作系统、网络管理软件、及网络协议栈的支撑和协调下，实现资源共享和信息传递。网络编程指的是面向联网的计算机系统，用软件方式对其进行编程和控制，从而实现特定的功能。本章主要讨论基于套接字的网络编程的基本技术，并给出重要的应用实例。首先，本章从网络编程模型开始，讨论常见的网络协议的数据结构和重要操作。其次，本章讨论了套接字编程的基本概念，并利用其展示了应用层协议的构建过程。接下来，我们讨论了原始套接字的概念，并结合网络安全中包的扫描和伪造等，讨论了原始套接字的典型应用。本章的重点不是讨论计算机网络的设计和架构，而是从程序设计的角度，建立对网络编程的重要概念和能力。

4.1 网络编程模型

网络编程模型基于网络的基本架构，建立相应的编程模型和抽象。

4.1.1 架构

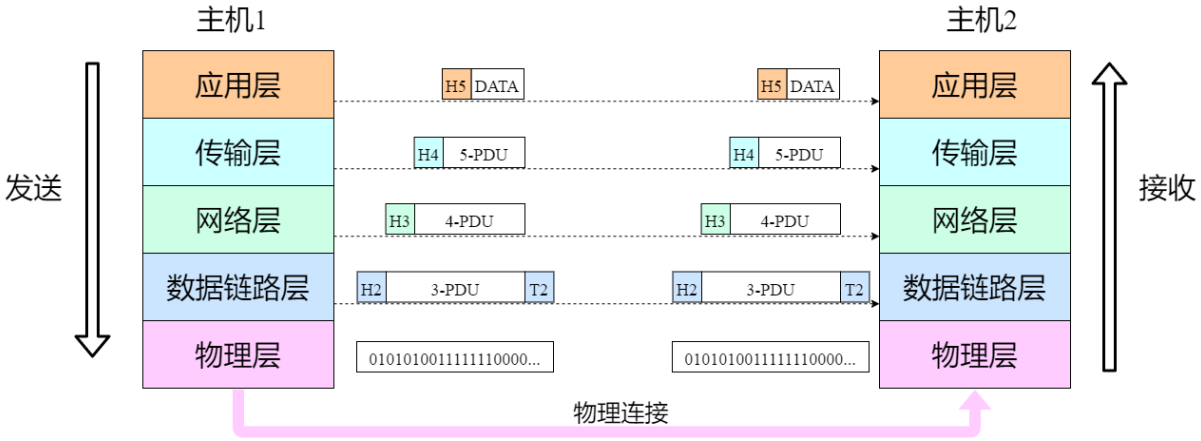
计算机网络广泛采用了分层思想进行架构。从历史发展角度看，最初计算机网络采用的 TCP/IP 协议的四层模型；而后，国际化标准组织基于 TCP/IP 协议进行了细化，提出了 OSI 七层模型，但七层模型过于细节与复杂，并没有得到实际大规模应用，实际的网络协议实现仍然采用了 TCP/IP 的协议架构。

<table>
<tr><th>OSI 七层模型</th><th>TCP/IP 四层模型</th><th>对应协议</th></tr>
<tr><td>应用层</td><td rowspan="3">应用层</td><td rowspan="3">HTTP, FTP, SMTP,
DHCP, DNS</td></tr>
<tr><td>表示层</td></tr>
<tr><td>会话层</td></tr>
<tr><td>传输层</td><td>传输层</td><td>TCP, UDP</td></tr>
<tr><td>网络层</td><td>网络层</td><td>IP, ICMP</td></tr>
<tr><td>数据链路层</td><td rowspan="2">链路层</td><td>ARP， RARP</td></tr>
<tr><td>物理层</td><td></td></tr>
</table>

网络的 TCP/IP 四层模型，采用了清晰的层次结构。首先，最下面一层是链路层，该层主要利用底层的物理连接（如电缆或光纤）传输数据，并利用 MAC 地址识别通信目标；该层的典型协议包括以太网协议、地址解析协议 ARP、以及逆地址解析协议 RARP 等。在链路层上面是网络层，同时也是 TCP/IP 协议栈的核心层，负责处理数据的传输路由和寻址。它使用 IP 协议对数据包进行寻址和路由选择，并通过路由器将数据包转发到目的地。网络层除了 IP 协议，还支持 ICMP 协议、IGMP 协议等。继续向上，传输层主要负责在源端和目的端之间提供端到端的数据传输。它可以保证数据的可靠传输，确保数据的完整性和顺序性，同时还支持流量控制和拥塞控制；常见的传输层协议包括 TCP 和 UDP 等。在最上层是应用层，负责处理特定应用程序的协议和数据交换。该层包括的典型协议包括用于 Web 应用程序的 HTTP 协议，用于电子邮件的 SMTP 协议，以及用于文件传输的 FTP 协议等。

计算机网络采用的这种分层结构，具有许多优点。首先，各个层的功能相对单一，更容易实现和维护；其次，各层互相独立，每一层只需向下和向上通过层间接口提供或使用服务，而不必暴露层内的具体实现；最后，分层结构保证了网络结构的灵活性，例如，增加新的应用层协议，只需从传输层向上扩展即可。

网络的分层结构，也决定了计算机间的网络通信，满足流水线结构，即用户数据从应用层开始，会被层层包装，最终通过物理层建立的链接进行传输。



具体的，上图演示了主机 1 向主机 2 发送报文的过程。从发送方看，用户数据从应用层开始，在向下逐层传递的过程中，每一层都将上一层的报文看成数据单元，并在该

数据单元上附加一个本层协议的首部（在数据链路层还会添加一个尾部）；例如，网络层将传输层的整个报文视为数据 4-PDU，并加上了网络层的报文头部 H3，从而形成本层的报文向下传递。最终，当该数据单元到达物理层时，就转换为电磁信号在物理链路上传播到目标主机。

到达终点后，报文的处理过程和发送过程相反；即报文在由底向上逐层传递的过程中，相应的首部就会被移除，然后由该层进行下一步处理。例如，在网络层，头部 H3 会被移除，并将剩余的数据 4-PDU 整体传递给上传协议进行进一步处理。同时，还需要注意到网络通信往往是全双工的，即接收方在处理完收到的报文后，可以给发送方进行回复，这时通信的过程与此类似，只是发送方和接收方互换了角色。

4.1.2 协议

协议是通信双方必须共同遵从的一组约定，如如何建立连接、如何互相识别、如何确定报文类型等。只有遵守这个约定，计算机之间才能相互通信交流。基于计算机网络的分层结构，协议和分层紧密联系，例如链路层协议、传输层协议、应用层协议等。

协议的三要素是语法、语义和时序。语法是数据的结构或格式，也就是指数据呈现的顺序。例如，简单的协议可以规定数据的前 8 位是发送方的地址，第二个 8 位指接收方的地址，剩下的数据流则是报文自身。语义是指每一段比特流分别表示什么意思，例如，同样是一个目的地址字段，可以是下一跳的地址，也可以指通信终点的地址。时序涉及两个方面：数据应该在何时发送出去以及能够以多快的速度发送。如果发送端的发送速率和接收端处理速率不统一，就有可能出现数据过载或者数据丢失的情况。

协议的定义格式一般比较复杂，但 Linux 系统以库头文件的形式，定义了常用标准协议的数据结构表示及其重要操作。通过合理使用这些标准数据结构，我们不但可以更方便地分析或者构建相应层的数据报文，而且有利于降低编程的复杂度和减少错误。更重要的是，这些数据结构和操作，从软件层面定义了对网络进行操作的行为。

接下来，我们以几个常用的网络协议为例，对网络层次中最重要和最常用的几个协议及其数据结构定义，进行分析。基于此，我们不但可以建立对这几个协议更深入

的理解，还可以对其他协议进行类似的分析和使用。

Ethernet II

Ethernet II 是数据链路层使用的一种帧格式，其规定的以太网帧结构是：

6 Bytes	6 Bytes	2 Bytes	46 ~ 1500 Bytes	4 Bytes
目的地址	源地址	协议	数据	FCS

其中目的地址（6 字节）和源地址（6 字节）字段分别代表了接收方和发送方的 MAC 地址；协议字段（2 字节）反映该数据报使用的上层协议；数据字段（从 46 到 1500 字节不等）是上层数据报的长度（如 IP 数据报）；尾部 FCS 字段（4 字节）用于帧校验，一般采用的是 CRC 校验和，从编程的角度，如果我们不关心校验，可以将其忽略。

Linux 系统在 net/ethernet.h 头文件中，定义了描述以太网帧首部的数据结构 ether_header，其具体定义如下：

```
#include <net/ethernet.h>
typedef struct  ether_header {
    u_char  ether_dhost[ETH_ALEN];
    u_char  ether_shost[ETH_ALEN];
    u_short ether_type;
} ether_header_t;
struct ether_addr{
  uint8_t ether_addr_octet[ETH_ALEN];
};
#define   ETHERTYPE_IP     0x0800    // IP
#define   ETHERTYPE_ARP    0x0806    // ARP
```

ether_dhost、ether_shost 和 ether_type 分别对应 Ethernet II 帧首部的前三个字段，即目的 MAC 地址、源 MAC 地址、以及上层协议类型。该头文件除了定义 MAC 地址长度的常量 ETH_ALEN 外（实际上该常量的定义在头文件 linux/if_ether.h 中），还定义了其它许多常量（如 ETHERTYPE_IP 和 ETHERTYPE_ARP 等）。

除了定义以太网报文的数据结构外，系统还提供了若干函数，以方便对以太网报文进行操作。例如，头文件 netinet/ether.h 定义了地址转换的相关函数：

```
#include <netinet/ether.h>
extern char *ether_ntoa(const struct ether_addr *__addr);
extern struct ether_addr *ether_aton(const char *__asc);
```

第一个函数将数据报中的以太网地址转为字符串，而第二个函数完成反向转换。

通过运用系统提供的数据结构以及相关函数，我们可以更容易地操作或构建以太网数据报。例如，如下程序根据输入的数据 data 和协议，构建一个以太网数据报：

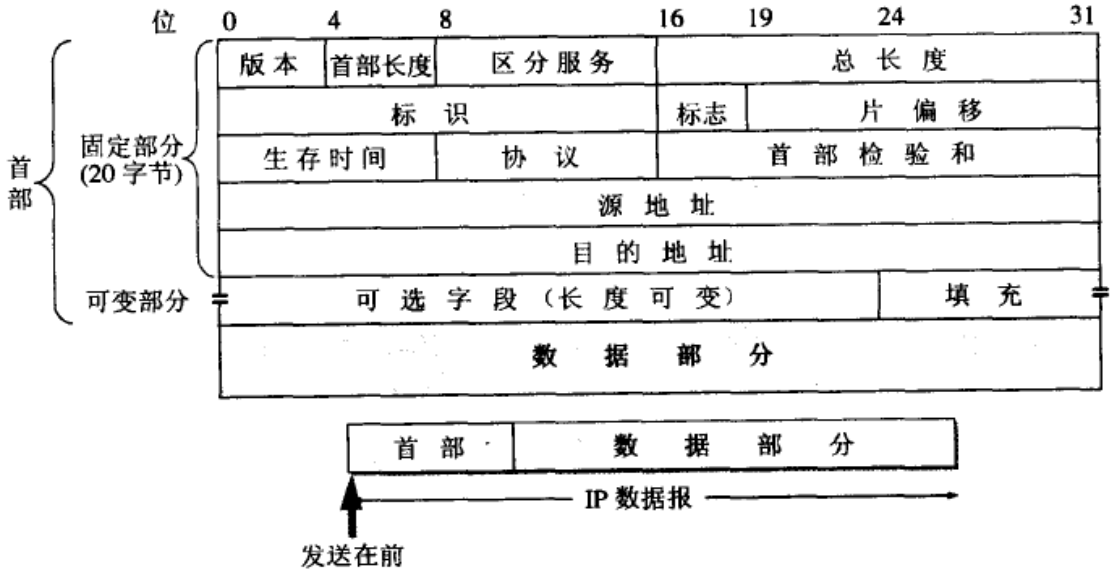
```
#include <netinet/in.h>
#include <net/ethernet.h>
#include <netinet/ether.h>
char *dst_addr = "00:01:02:03:04:05";
char *src_addr = "06:07:08:09:0a:0b";
#define BUF_SIZE 1024
#define DATA "hello, world"
#define HEADER_SIZE (sizeof(struct ether_header))

struct ether_header *
create_ether_packet(char *src_mac, char *dst_mac, unsigned short
proto, char *data){
    static unsigned char buff[BUF_SIZE];
    struct ether_header *eth = (struct ether_header *)(buff);
    memcpy(eth->ether_dhost, ether_aton(dest_addr), ETH_ALEN);
    memcpy(eth->ether_shost, ether_aton(source_addr), ETH_ALEN);
    eth->ether_type = htons(proto);
    strcpy(buff + HEADER_SIZE, DATA);
    return buf;
}
create_ether_packet(dst_addr, src_addr, IPPROTO_IP, DATA);
```

在这个例子中，我们制作了一个自定义的 MAC 帧。该数据帧的接收方 MAC 地址为 00-01-02-03-04-05，发送方 MAC 地址为 06-07-08-09-0a-0b。协议字段为内容为传入的参数 proto。数据部分填充为"hello, world"的 ASCII 码。

值得注意的是，这里并没有添加 MAC 帧中的 FCS 字段，这是因为通常情况下 FCS 的添加和移除会由内核或者硬件来完成。另外，在 10M 以太网的 MAC 帧中，数据部分的最小长度为 46 字节，当数据部分长度不足时，网卡会自动将其填充至 46 字节。

IP 协议

IP 协议是网络层使用的最为核心的协议，它接受传输层的报文作为数据部分，添加首部组成网络层的 IP 数据报，其标准格式为：



IP 数据报的格式

Linux 系统在 netinet/ip.h 头文件中，定义了描述 IP 数据报首部的数据结构，其具体定义如下：

```
#include <netinet/ip.h>
struct iphdr{
    unsigned int version:4;
    unsigned int ihl:4;
    uint8_t tos;
    uint16_t tot_len;
    uint16_t id;
    uint16_t frag_off;
    uint8_t ttl;
    uint8_t protocol;
    uint16_t check;
    uint32_t saddr;
    uint32_t daddr;
    /*The options start here. */
};
```

结构体 iphdr 中的成员与 IP 数据报首部的字段一一对应（此例以大端存储机器为例，小端机器与此类似）。从程序设计的角度，其中比较重要字段包括：版本 version，占用 4 个比特，其值一般取常数 4 或 6，分别代表 IPv4 或 IPv6 协议；首部长度 ihl，占用 4 个比特，代表首部长度（以 4 字节为单位），其值在 5 和 15 之间，因此最小长度为 4*5=20 字节；区分服务 tos 一般不使用；全长字段 tot_len 占用 16 个比特，代表 IP 数据报的整体长度，即包括头和数据，最小为 20 字节，最长为 65535 字节；标识符字段 id 占用 16 个比特，代表报文的分片的标识；分片偏移 ip_off 占用 16 个比特，代表当前 IP 数据报的分片，在原始报文中的偏移；存活时间 ttl 占用 8 个比特，代表该报文在网络中的存活时间（以秒为单位）；协议 protocol 占用 8 个比特，代表该 IP 报文数据段所使用的协议；校验和 check 占用 16 个比特，对 IP 数据报进行纠错；源和目的 IP 地址 saddr 和 daddr 分别代表发送方和接受方的 IP 地址。

为了方便进行程序设计，系统还提供了常用的常量定义和操作函数。例如，头文件 netinet/in.h 定义了许多有用的宏和函数：

```
#include <netinet/in.h>
IPPROTO_ICMP = 1
IPPROTO_TCP = 6
IPPROTO_UDP = 17
extern uint32_t ntohl (uint32_t __netlong);
extern uint16_t ntohs (uint16_t __netshort);
extern uint32_t htonl (uint32_t __hostlong);
extern uint16_t htons (uint16_t __hostshort);
```

这些函数完成网络数据和本机数据的转换，从而有效屏蔽了本地和网络数据格式表示间的差异性。例如，函数 ntohl 将网络层的四字节的整型数据__netlong，转换为本机表示的四字节数据。

通过有效利用系统提供的这些数据结构表示和操作函数，我们可以高效的实现 IP 数据报的分析和操作。例如，假定我们要从源机器（假定 IP 地址是 192.168.31.2），向目的机器（假定 IP 地址是 192.168.31.3），发送字符串数据"hello, world"。如下实例程序展示了“手工”制作一个 IP 数据报的流程：

```
#include<netinet/in.h>
#include<netinet/ip.h>
#define SRC_IP "192.168.31.2"    // source ip addr
#define DST_IP "192.168.31.3"    // destination ip addr
#define BUF_SIZE 1024
#define DATA "hello, world"
#define HEADER_SIZE (sizeof(struct iphdr))
unsigned short checksum(unsigned char* buf, int size);
int main(){
    unsigned char buff[BUF_SIZE] = {'\0'};
    struct iphdr *iph = (struct iphdr*)(buff);
    iph->ihl  = 5;  // length of IP header (in #words)
    iph->version = 4;  // IPv4
    iph->ttl  = 64;    // to live 64 seconds
    iph->saddr = inet_addr(SRC_IP);
    iph->daddr = inet_addr(DST_IP);
    strcpy(buff + HEADER_SIZE, DATA); // data
    iph->tot_len = htons(HEADER_SIZE + strlen(DATA));
    iph->check = htons(checksum(buff, HEADER_SIZE));
    return 0;
}
```

整个 IP 数据报放在 buff 数组中，首先，我们分别填充了报文头部的各个字段，再填充数据。尽管这个程序并不复杂，但有三个关键点需要特别注意：第一，由于部分字段的值就取默认的零值（第 10 行），因此，无必要再进行显式的赋值；例如，协议字段 protocol 取默认的 IPPROTO_IP。第二，需要特别注意值的本地表示和网络表示的转换；例如，填充报文长度字段 tot_len 时，需要利用函数 htons，将本地表示转换为网络表示。第三，利用 inet_addr 函数（定义在头文件 arpa/inet.h 中），我们可以很方便的将十进制表示的 IP 地址转化为二进制形式。

与 MAC 帧不同的是，IP 首部中的校验和字段 check 需要我们计算并填充，这个工作由 checksum 函数完成：

```
unsigned short checksum(unsigned char* buf, int size){
    unsigned int check_sum = 0;
    for(int i = 0; i < size; i += 2){
        unsigned short first = (unsigned short)buf[i] << 8;
        unsigned short second = (unsigned short)buf[i+1] & 0x00ff;
        check_sum += first + second;
    }
    while (1){
        unsigned short c = (check_sum >> 16);
        if(c > 0){
            check_sum = (check_sum << 16) >> 16;
            check_sum += c;
        } else
            break;
    }
    return ~check_sum;
}
```

需要特别注意的是，在计算校验和之前，IP 头中的 check 字段必须首先清零。

UDP 协议

UDP 协议是传输层的一个常用通信协议。UDP 是面向无连接的，其报文结构也比较简单，具体格式为：

0								1								2								3							
0	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
源端口																目标端口															
长度																校验															

Linux 系统在头文件 netinet/udp.h 中，给出了 UDP 数据报头部的数据结构 udphdr，其定义如下：

```
#include <netinet/udp.h>
struct udphdr{
    uint16_t source;
    uint16_t dest;
    uint16_t len;
    uint16_t check;
};
```

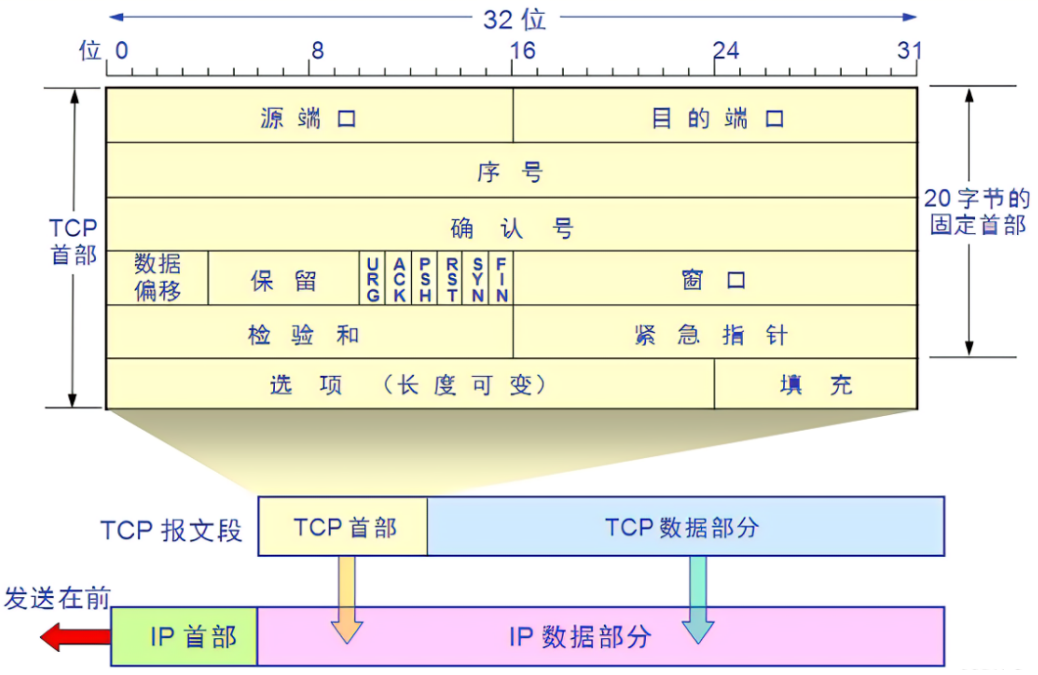
其中源（source）和目的端口（dest）字段都为 16 比特，分别代表发送方和接收方的端口号；报文长度字段 len 占用 16 比特，表示整个 UDP 报文的长度，即包括头和数据部分；字段 check 是校验和，但在 UDP 中不是强制的（仅限 IPv4），因此可以简单置为零。

借助这些数据结构定义以及函数，我们同样很方便的分析或者制作 UDP 报文。例如，如果我们要从端口 12345 向端口 23456 发送包含字符串数据"hello, world"的 UDP 报文，我们可以按如下方式“手工”构造 UDP 报文：

```
#include<netinet/udp.h>
#define SRC_PORT 12345    // source tcp port
#define DST_PORT 23456    // destination tcp port
#define BUF_SIZE 1024
#define DATA "hello, world"

int main(){
    unsigned char buff[BUF_SIZE] = {'\0'};
    struct udphdr *p = (struct udphdr *)(buff);
    p->source = htons(SRC_PORT);
    p->dest = htons(DST_PORT);
    p->len = sizeof(*p) + strlen(DATA);
    strcpy(buff + sizeof(*p), DATA);
    return 0;
}
```

TCP 协议

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，其报文格式如下：



Linux 系统在 netinet/tcp.h 头文件中，定义了描述 TCP 报文首部的数据结构 tcphdr，其具体定义如下所示：

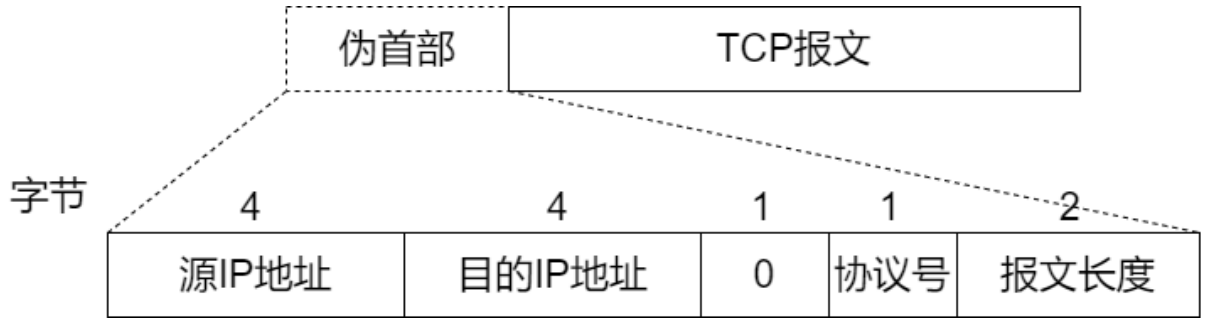
```
#include <netinet/tcp.h>
struct tcphdr{
    uint16_t source;
    uint16_t dest;
    uint32_t seq;
    uint32_t ack_seq;
    uint16_t res1:4; uint16_t doff:4;
    uint16_t fin:1; uint16_t syn:1; uint16_t rst:1; uint16_t
psh:1;
    uint16_t ack:1; uint16_t urg:1; uint16_t res2:2;
    uint16_t window;
    uint16_t check;
    uint16_t urg_ptr;
};
```

从程序设计的角度，重要的头部字段如下：源端口 source 和目的端口（dest）都是 16 个比特，分别表示发送方和接收方的端口号；而序列号 seq 和确认号 ack_seq 都为 32 比特，分别表示发送方的序号和接收方期待接受的下一个序号；数据偏移字段 doff 占用 4 个比特，表示 TCP 携带数据从 TCP 头开始的偏移量（以 4 字节计），例如，当 doff=5 时，数据位于 5*4=20 字节偏移处开始；接收窗口 window 占用 16 个比特，表示接收方可以接受的最大数据字节数，用于流量控制；校验和字段 check 占用 16 个比特，用于计算 TCP 报文的校验和（即包括 TCP 头以及数据）。

借助这些数据结构定义以及函数，我们同样很方便的分析或者制作 TCP 报文。例如，如果我们要从端口 12345 向端口 23456 发送包含字符串数据"hello, world"的 TCP 报文，我们可以按如下方式“手工”构造 TCP 报文：

```
#include<netinet/in.h>
#include<netinet/tcp.h>
#define SRC_PORT 12345    // source tcp port
#define DST_PORT 23456    // destination tcp port
#define BUF_SIZE 1024
#define DATA "hello, world"

unsigned short tcp_checksum(struct tcphdr *tcp);

int main(){
    unsigned char buff[BUF_SIZE] = {'\0'};
    struct tcphdr *th = (struct tcphdr *)(buff);
    th->source = htons(SRC_PORT);
    th->dest = htons(DST_PORT);
    th->syn = 1;    // connection request
    th->window = htons(14600);
    th->doff = 5;
    strcpy(buff + sizeof(struct tcphdr), DATA);
    th->check = htons(tcp_checksum(th));
    return 0;
}
```

该构造过程和 IP 报文的构造过程类似，不再赘述。需要特别注意的是，TCP 在计算校验和时，需要在报文面前加上伪首部，即校验和计算包括伪首部、TCP 首部以及数据，其中伪首部的结构为：



伪首部包括源和目的的 IP 地址（各占四个字节）、协议号、和报文长度等；但伪首部

只用于 TCP 校验和的计算，实际通信时并不会添加。尽管 TCP 校验和的计算原理并不复杂，但计算过程比较繁琐，此处不再赘述，感兴趣的读者可参考相关文献。

TCP 协议具有很多优势，如可靠连接、流量控制等等，因此成为了更高层协议广泛采用的底层传输协议。但从程序设计的角度看，程序员不必关心这些特性的实现细节，而是可以直接基于这些特性，构建更高层的抽象，这也再次提现了计算机网络分层架构具有的优势。

4.2 套接字

套接字（Socket）是计算机网络编程中提供的通信接口，用于在同一计算机或不同计算机的进程之间传输数据；从程序设计的角度看，套接字是程序对网络协议栈进行访问和控制的重要机制。

套接字常用的函数原型如下：

```
#include <sys/so>
int socket(int domain, int type, int protocol);
int bind(int sockfd, struct sockaddr *addr, socklen_t addrlen);
int listen(int sockfd, int backlog);
int accept(int sockfd, struct sockaddr *addr, socklen_t
*addrlen);
```

其中，socket()函数用于创建并返回一个新的套接字，返回的套接字以文件描述符的数据结构给出（即整型 int），这种设计方便以文件访问的形式，来一致的操作套接字。socket 函数接受三个参数：网络域 domain、网络类型 type 和协议 protocol；通过给 socket 函数提供不同的参数组合，可以定义不同套接字类型，下表给出了常用的参数组合。

参数	常用取值	描述
domain	AF_INET	IPv4 协议族
	AF_INET6	IPv6 协议族
	AF_UNIX	UNIX 域套接字协议族

type	SOCK_STREAM	流套接字，用于可靠的、面向连接的传输（TCP）
	SOCK_DGRAM	数据报套接字，用于不可靠的、无连接的传输（UDP）
	SOCK_RAW	原始套接字，提供对网络协议的直接访问
protocol	0	自动选择适当的协议
	6	TCP 协议
	17	UDP 协议

在利用套接字编程时，经常用到以下组合，来创建不同类型的套接字：

```
/* Create an IPv4 socket for TCP */
socketfd = socket (AF_INET, SOCK_STREAM, 0);
/* Create an IPv4 socket for UDP */
socketfd = socket (AF_INET, SOCK_DGRAM, 0);
/* Create a UNIX domain socket*/
socketfd = socket (AF_UNIX, SOCK_STREAM, 0);
/* Create a raw socket for Ethernet frames */
socketfd = socket (AF_PACKET, SOCK_RAW, htons (ETH_P_ALL));
```

函数 bind()将一个本地地址与套接字关联，使其可以监听该地址并接受连接请求。

```
bind(server_sock_fd, (struct sockaddr *)&server_addr,
sizeof(server_addr)
```

在上述代码中，我们为 server_sock_fd 设置了监听地址和端口号，其中，server_addr 是一个指向结构体的指针，该结构体包含了希望绑定到的地址和端口号。

函数 listen()将一个套接字设置为监听模式，以便接受来自客户端的连接请求。

```
listen(server_sock_fd, 5)
```

在上面的代码中，我们尝试将 server_sock_fd 设置为监听模式，并且指定了等待连接的队列最大长度为 5。

函数 accept()接受来自客户端的连接请求，创建并返回一个新的套接字，用于与客

户端通信。

```
accept(server_sock_fd, (struct sockaddr *)&client_addr,
&client_addr_len)
```

利用套接字提供的函数，我们可以进行服务器端或客户端编程。我们首先给出一个使用套接字创建服务器的实例，为简单起见，代码中省略了相关的错误检查逻辑。

```
#define HOST_PORT 12345
#define BUF_SIZE 1024
void main (int argc, char argv){
    int sock_fd = socket(AF_INET, SOCK_STREAM, 0);
    int yes = 0;
    setsockopt(sock_fd, SOL_SOCKET, SO_REUSEADDR, &yes,
sizeof(int));
    struct sockaddr_in host_addr;
    memset(&host_addr, 0, sizeof(host_addr));
    host_addr.sin_family = AF_INET;
    host_addr.sin_port = htons(HOST_PORT);
    host_addr.sin_addr.s_addr = 0;
    bind(sock_fd, (struct sockaddr *)&host_addr, sizeof(struct
sockaddr));        // #1: bind
    listen(sock_fd, 10);    // #2: listen
    while(1){
        struct sockaddr_in client_addr;
        memset(&client_addr, 0, sizeof(client_addr));
        unsigned int client_addr_len = sizeof(client_addr);
        int client_sock_fd = accept(sock_fd, (struct sockaddr
*)&client_addr, &client_addr_len);    // #3: accept
        char *msg = "hello, world\n";
        printf("sent a message to the client:\n");
        write(client_sock_fd, msg, strlen(msg));
        close(client_sock_fd);
    }
}
```

首先，服务端设置 socket 使用基于 TCP 的通信方式 SOCK_STREAM（第 4 行），并调用 bind 函数绑定到本机（第 12 行），以监听指定端口 12345（第 10 行），在 setsocket()函数中设置了 SO_REUSEADDR，表示服务器启用了目的重用，在 listen 函数中设置服务器等待连接的队列最大长度为 10（第 13 行）。

接着，服务器调用 accept 函数（第 18 行），开始尝试接受来自客户端的连接，特

别需要注意的是，该函数的调用是阻塞的，亦即在连接到来前，该函数调用不会返回。而当连接到来后，服务器把客户端的信息写入 client_addr 后；便向套接字写入给定的信息（第 21 行），该信息通过套接字发送给客户端。

最后，服务器关闭套接字 client_sock_fd（第 22 行），并进入下一轮循环，继续等待并处理后续到达的连接。

我们可以实现一个客户端，来实现与服务端的通信。为此，我们给出如下的客户端程序，其用来向服务器建立连接，接收并答应服务器端返回的消息：

```
#define SERVER_IP "127.0.0.1"
#define SERVER_PORT 12345
#define BUF_SIZE 1024
int main(){
    struct sockaddr_in server_addr;
    char buffer[BUF_SIZE];
    int sock_fd = socket(AF_INET, SOCK_STREAM, 0);
    memset(&server_addr, 0, sizeof(server_addr));
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(SERVER_PORT);
    inet_pton(AF_INET, SERVER_IP, &server_addr.sin_addr);
    connect(sock_fd, (struct sockaddr *)&server_addr,
sizeof(server_addr);
    ssize_t recv_bytes = read(sock_fd, buffer, BUF_SIZE);
    printf("Received: %.*s\n", (int)recv_bytes, buffer);
    close(sock_fd);
    return 0;
}
```

客户端程序相比服务器端更加简洁，它先向服务器端发起连接（第 12 行），然后读取套接字中的数据（第 13 行）。

该程序执行效果如下所示：

```
$ ./hello-server.out            |$ ./hello-client.out
sent a message to the client:   |hello, world
...                             |
```

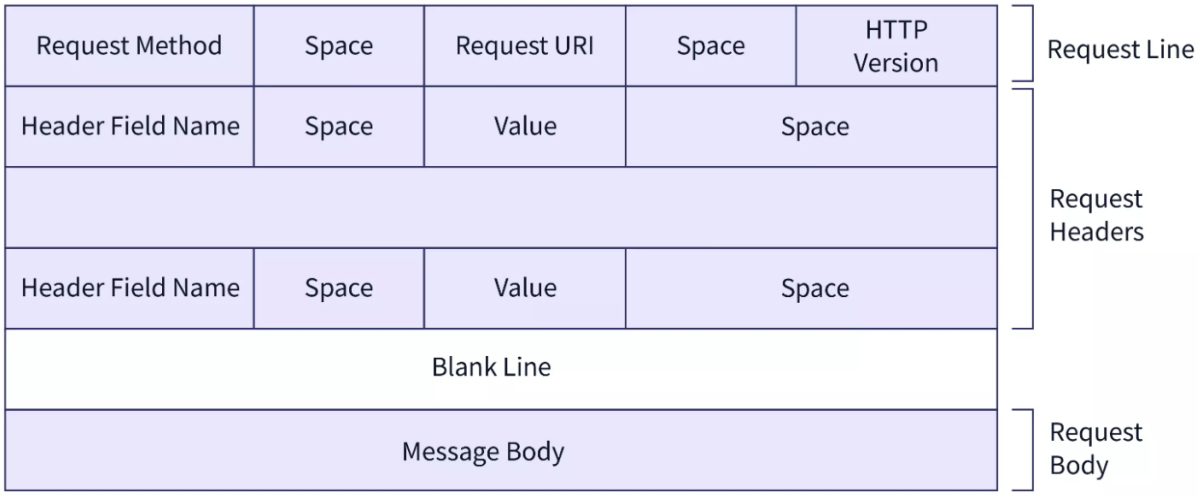
4.3 应用实例

利用套接字提供的网络编程抽象，我们可以很方便的构建基于底层网络协议的高层协议。在本节中，我们将以构建一个 HTTP 服务器为实例，讨论套接字对超文本传输协议的支持，以展示套接字编程在实际场景中的应用。

4.3.1 HTTP 协议

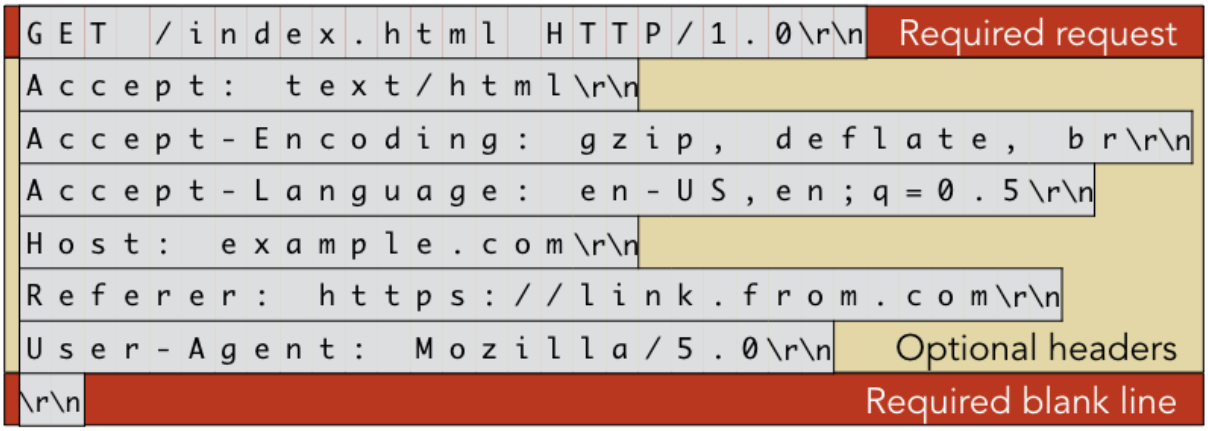
超文本传输协议（Hypertext Transfer Protocol，HTTP）是一种用于在互联网上传输超文本文档的协议，并且它是一个明文协议。HTTP 协议使用两种报文形式，分别是 HTTP 请求报文和 HTTP 响应报文。

HTTP 请求是指客户端发送请求到服务器以获取资源。HTTP 请求使用的报文以一系列的头部行开始，每一行都以一个 CRLF 两字符序列结尾，即回车-换行。请求的第一行必须是指定的请求行，并且必须遵循给定的结构。在第一行之后，其他所有的头部都是可选的，但它们为客户端和服务器提供了其他信息。在请求头部的结尾，有一个单独的空白行（即仅由回车-换行 CRLF 组成），而后是请求体的内容：



GET 请求是最常见的 HTTP 请求方式，表示客户端正在请求一个文件的副本。例如，下图给出了来自客户端的 GET 请求。第 1 行表明这是一个 HTTP 的 GET 请求，需要得到文件/index.html；使用的 HTTP/1.0 的协议。第 2 行到第 7 行是请求头部，分别

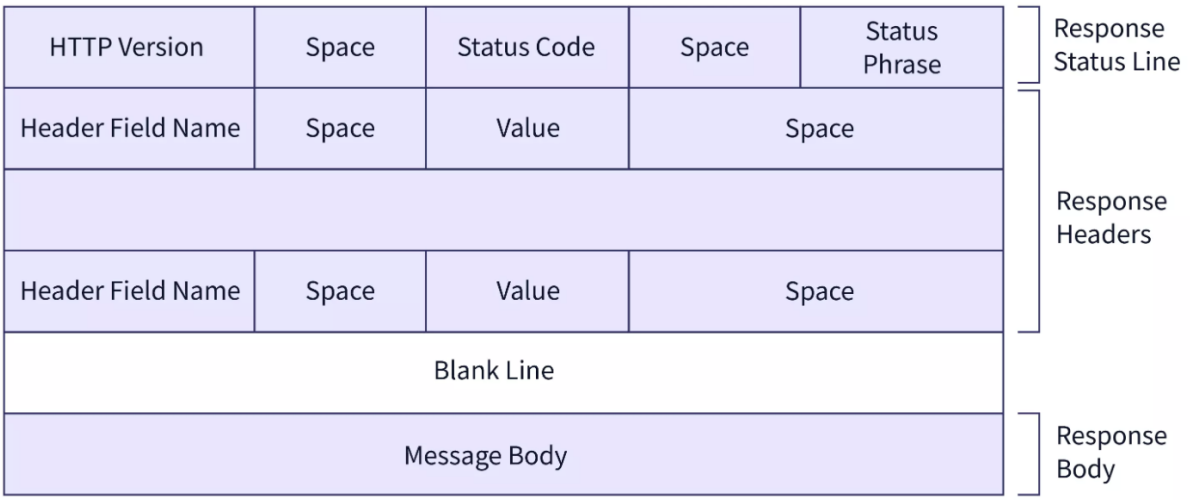
代表了客户端能够接受的数据类型、编码、语言、主机、来源、以及所使用的客户端（一般是浏览器）等信息。服务器需要根据请求行和请求头等信息，综合进行处理，并给客户端进行响应。



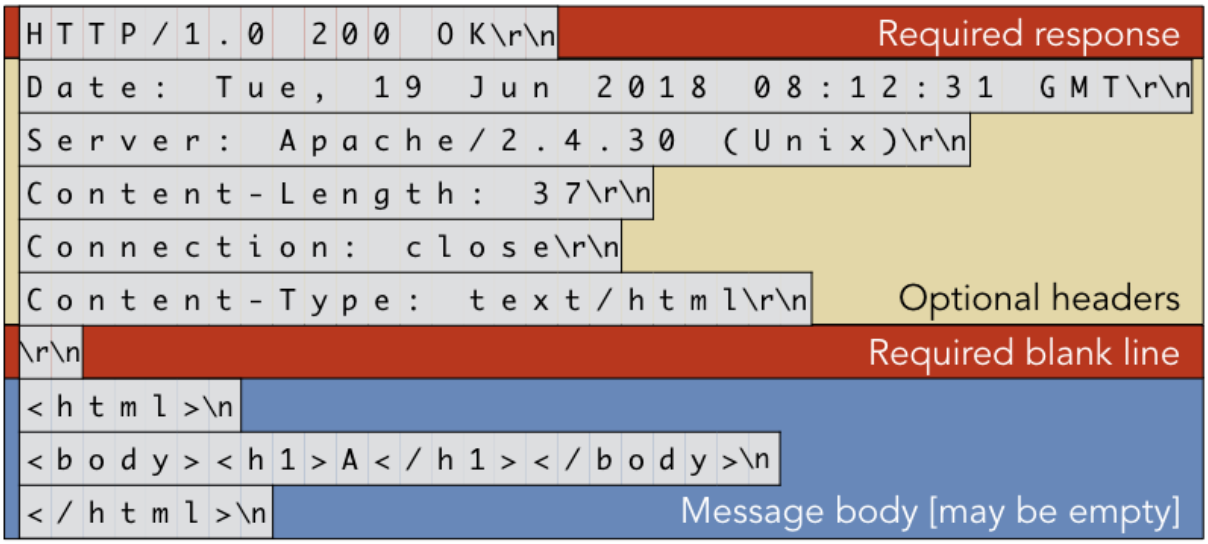
HTTP 请求除了包括上面讨论的 GET 请求外，还包括 POST 请求，DELETE 请求等等，常见的请求方式总结在如下的表中，限于篇幅，此处不再赘述，感兴趣的读者可参考 HTTP 请求的相关资料。

方法	描述
GET	请求特定的页面信息，并返回实体主体。
POST	向指定资源提交数据进行处理请求（例如，提交表单或上传文件）。数据被包含在请求正文中。POST 请求可能会导致新的资源的建立和/或已有资源的修改。
PUT	从客户端向服务器传送的数据取代指定的文档的内容。
HEAD	请求服务器响应与 GET 请求相同的响应，但没有响应主体。
DELETE	请求服务器删除指定的页面。

HTTP 服务器接收到请求并接受处理后，会发送 HTTP 响应给客户端；响应报文也满足特定的格式，如下图所示。具体地，响应报文以一个必需的响应行开始（第一行），其中存放了响应的状态；接下来，与请求报文格式相似，响应报文也包含一系列的头部行，每一行以一个两字符序列结尾；在头部行的结尾，有一个单独的空白行（仅由 CRLF 两个字符组成）；最后，是响应体的内容：



下图给出了对刚才讨论过的 GET 请求，服务器对应的响应报文。第一行的响应状态指明了 HTTP 协议的版本号、服务器成功响应的状态码 200、以及一个提示信息 OK。响应头（第 2 到 6 行）依次包括时间、服务器类型信息、响应体的长度、连接状态、以及内容的类型等。在空白行后是响应体，响应体是存储在 Web 服务器指定的根目录中的 index.html 文件的内容。



响应码用来标识响应的状态。除了上述标识成功的响应码 200 外，还有许多其它响应码，来标识其它的响应状态。下表中列出了典型的响应码。限于篇幅，感兴趣的读者可参考 HTTP 规范的相关资料。

状态码	描述
200	OK - 请求已成功
204	No Content - 请求成功，但响应中无内容
400	Bad Request - 请求无效，服务器无法理解

404	Not Found - 未找到请求的资源
500	Internal Server Error - 服务器内部错误

HTTP 协议的工作过程大致如下：客户端首先向服务器发送 HTTP 请求，请求通过传输层到达服务器（一般是 TCP 传输）；服务器进行处理后，将 HTTP 响应返回给服务器。具体的，客户机先在浏览器中输入需要访问网页的 URL 或者点击某个网页中链接，浏览器会根据 URL 中的域名，通过 DNS 解析出目标网页的 IP 地址。接着，客户端会通过 TCP/IP 协议，来和服务端建立链接。然后，客户机基于建立的连接，发送 HTTP 请求报文给服务器。接下来，服务器接到请求后，给予相应的响应信息。一般情况下，一旦 Web 服务器向浏览器发送了请求数据，Web 服务器会关闭 TCP 连接，如果浏览器或者服务器需要保持该连接，可以在其头信息加入 keep-alive 头部，则 TCP 连接将仍然保持打开状态，于是，浏览器可以继续通过相同的连接发送请求。

根据实际场景的不同，HTTP 协议会变的十分复杂，想要了解更多内容的读者可以阅读相关材料。

4.3.2 Web 服务器

Web 服务器接受 HTTP 协议的请求，基于 HTTP 协议的规定进行处理并返回 HTTP 的响应。在本节，我们用套接字，实现一个简单的 HTTP 服务器，进一步理解利用套接字编程提供的能力，在底层 TCP/IP 协议栈上，构建上层应用协议的一般技术。 我们给出的这个 Web 服务器接受 GET 请求，并给出响应。

```
#define SERVER_IP "127.0.0.1"
#define SERVER_PORT 8080
#define BUF_SIZE 1024
void do_request(int client_sock){
    char request[BUF_SIZE] = {'\0'};
    ssize_t num_bytes = read(client_sock, request, BUF_SIZE-1);
    // Parse the HTTP request
    char method[16] = {'\0'}, path[128] = {'\0'};
    sscanf(request, "%s %s", method, path);
    // Handle only GET requests for simplicity.
    if(strcasecmp(method, "GET") != 0){
        perror("Unsupported HTTP method");
        return;
    }
    // Open and read the requested file.
    char response[BUF_SIZE] = {'\0'};
    int file_fd = open(path, O_RDONLY);
    if(file_fd < 0){
        #define HTML_404 "<h1>404 Not Found</h1>"
        sprintf(response,
         "HTTP/1.1 404 Not Found\r\n"
         "Content-Type: text/html\r\n"
         "Content-Length: %d\r\n\r\n"
         HTML_404, strlen(HTML_404));
         write(client_sock, response, strlen(response));
         return;
    }
    ssize_t bytes_read;
    struct stat st;
    fstat(file_fd, &st);
    sprintf(response, "HTTP/1.1 200 OK\r\n");
        "Content-Length:%d\r\n\r\n", st.st_size);
    write(client_sock, response, strlen(response));
    while((bytes_read = read(file_fd, response,
sizeof(response))) > 0){
        write(client_sock, response, bytes_read);
    close(file_fd);
```

首先，Web 服务器使用基于 TCP 的通信方式建立链接（第 44 行），并调用 bind 函数监听指定端口 8080（第 49 行）。接着，服务器调用 accept 函数（第 52 行），开始尝试接受来自客户端的连接，当有客户端的连接请求到来时，服务器将调用 do_request 函数处理请求（第 53 行）。

在 do_request 函数中（第 4 行），服务器通过 read 函数将 HTTP 请求报文读入到缓冲区中 request（第 6 行）；然后，服务器根据 HTTP 请求报文的格式，对请求方法和文件路径进行读取（第 9 行）。如果不是 GET 请求，将上报错误信息（第 12 行）；对于 GET 请求，服务器将根据文件路径尝试打开文件（第 17 行），如果文件不存在，服务器将根据 HTTP 响应报文的格式，返回 404 错误（第 25 行），否则，服务器先通过 client_sock 写入 HTTP 响应报文的响应头（第 33 行），再写入文件内容（第 35 行），待文件内容读取完成后，关闭该文件（第 36 行）。

最后，服务器处理完该请求后，关闭套接字 client_sock_fd（第 52 行），并进入下一轮循环，继续等待并处理后续到达的请求。

客户端可以用各种方式来访问服务器。例如，当客户端利用浏览器访问服务器时：http://127.0.0.1:8080/index.html，将得到并显示此文件 index.html 的页面：

若访问的文件不存在，则收到不存在提示：

4.4 原始套接字及其应用

原始套接字（Raw socket）是一种面向低层网络协议的套接字，它绕过了通常的传输层协议（如 TCP 和 UDP）甚至网络层协议（如 IP），允许应用程序直接访问底层数据报。借助原始套接字，用户可以自定义协议头部，并进行底层的网络操作。在本小节，我们将讨论原始套接字，并展示其典型的应用，包括：网络流量嗅探、数据包的伪造，等等。

4.4.1 原始套接字

原始套接字是一种特殊的网络编程接口，直接对接 IP 层及更下层的网络协议，允许程序员创建、控制和处理这些层的数据包。由于它可以绕过操作系统的协议栈，原始套接字为开发者提供了对网络协议的细粒度控制，因此它在自定义协议开发、网络监测及安全领域中都很有价值。

在 POSIX 标准中，原始套接字的创建和操作仍然基于标准的套接字 API 函数：

```
#include <sys/socket.h>
int socket(int domain, int type, int protocol);
```

但具有不同的参数组合如下：

参数	常用取值	描述
domain	AF_PACKET	低级的数据链路访问，用于捕获网络层以下的数据包。
type	SOCK_RAW	原始套接字，提供对网络协议的直接访问
protocol	ETH_P_ALL	用于在链路层（数据链路层）捕获所有的数据帧
	ETH_P_ARP	指定要捕获 ARP 协议类型的数据帧
	ETH_P_RARP	指定要捕获 RARP 协议类型的数据帧

其中常用的组合是采用 AF_PACKET 域，并采用 SOCK_RAW 的通信类型，并采用 ETH_P_ALL 的协议，这能够捕获和处理链路层及其上的所有数据报。

我们可以利用以下程序中的原始套接字，来捕获以太网帧：

```
#define BUF_SIZE 2048
int main() {
    char buffer[BUF_SIZE] = '\0';
    char *if_name = "eth0";
    int sock_fd = socket(AF_PACKET, SOCK_RAW, htons(ETH_P_ALL));
    struct sockaddr_ll sa;
    memset(&sa, 0, sizeof(struct sockaddr_ll));
    sa.sll_family = AF_PACKET;
    sa.sll_protocol = htons(ETH_P_ALL);
    sa.sll_ifindex = if_nametoindex(if_name);
    bind(sock_fd, (struct sockaddr*)&sa, sizeof(struct
sockaddr_ll);
    while(1){
        int n = recvfrom(sock_fd, buffer, BUF_SIZE, 0, 0, 0);
        printf("Received a frame of length: %d bytes\n", n);
    }
    close(sock_fd);
    return 0;
}
```

首先，该程序调用函数 bind，将链路层的原始套接字（第 5 行）绑定在特定接口 eth0 上（第 11 行）；接着，程序调用 recvfrom，捕获所有类型的以太网帧（第 13 行），数据包的内容存放在缓冲区 buffer 中。

程序运行时，将不断打印类似如下的信息，显式不断收取了数据包：

```
$ sudo ./sniff
Received a frame of length: 73 bytes
Received a frame of length: 73 bytes
...
```

数据包的获取一般是网络监控和分析中的第一个步骤。在获取到数据报后，我们往往希望进一步对网络报进行解析，以深入了解其结构和含义。接下来，我们按网络协议分层的方式，以一个 TCP 报文为例，讨论数据报解析的过程。

从程序设计的角度看，对数据报的分析就是根据读入的缓冲区的包内容，按照本章第 1 小节讨论的包格式，进行解析的过程，并且解析是逐层进行的，即从底层的链路层开始，到网络层、传输层、以及应用层。下面的函数 packet_analyze 接受数据包缓冲区的起始地址 buf 和长度 n，对包进行分析。

首先，该程序调用函数 analyze_eth，解析以太网包的结构，并得到其上层的协议 inner_ether_proto（第 3 行）；然后，程序按照 inner_ether_proto 的可能情况分类讨论（第 4 行），如果该包是 IP 包的话，则取得 IP 包的起始地址（第 6 行），回想一下，IP 包的数据地址其实就是以太网包的数据载荷部分；接下来，程序继续调用 analyze_ip 函数，得到 IP 包中的协议类型 inner_ip_proto（第 7 行），并对其可能的值进行分类讨论；如果该类型是 TCP 的话（第 9 行），则取得 TCP 的包头部（第 10 行），并调用 analyze_tcp 函数，进行 TCP 的包分析。以上过程可以继续进行下去，进一步分析上层协议。

```
void packet_analyze(char *buf, ssize_t n){
    // first analyze the ether packer, get its inner protocol
    int inner_ether_proto = analyze_eth(buf);
    switch(inner_ether_proto) {
        case ETHERTYPE_IP: {
            char *ip_addr = buf + sizeof(struct ether_header);
            int inner_ip_proto = analyze_ip(ip_addr);
            switch (inner_ip_proto) {
                case IPPROTO_TCP: {
                    char *tcp_addr = ip_addr + sizeof(struct
iphdr);
                    analyze_tcp(tcp_addr);
}
```

下面的函数 analyze_eth 解析以太网帧。程序解析源和目标 MAC 地址，并识别内部协议类型，可能的值有 ARP、IPv4 和 IPv6 等，需要注意，此处省略了以太网层的很多其它协议。对于每种协议类型，程序会进行特定的处理逻辑。最后，程序返回上层的协议 proto。

```
int analyze_eth(char *buf){
    struct ether_header *ethhdr = (struct ether_header *)buf;
    unsigned char *src = ethhdr->ether_shost;
    unsigned char *dst = ethhdr->ether_dhost;
    int proto = ntohs(ethhdr->ether_type);
    switch (proto){
        case ETHERTYPE_ARP:  // ...
        case ETHERTYPE_IP:   // ...
        case ETHERTYPE_IPV6: // ...
    }
    return proto;
}
```

程序 analyze_ip 会继续解析 IP 头部。程序首先识别 IP 数据包的协议类型（第 4 行），根据可能的值（如 TCP 和 UDP 等），对于每种协议类型，进行相应处理。

```
int analyze_ip(char *buf) {
    struct iphdr *ip_addr = (struct iphdr *)buf;
    int proto = ip_addr->protocol;
    switch(proto){
        case IPPROTO_TCP: // ...
        case IPPROTO_UDP: // ...
    }
    return proto;
}
```

函数 analyze_tcp()解析 TCP 报文。该函数读取 TCP 包的源和目标端口信息；并且还可以对包的内容进行进一步处理。例如，这个示例代码中，打印其有效载荷的前 10 个字符，可用以除错调试。

```
int analyze_tcp(char *addr){
    struct tcphdr *tcp_addr = (struct tcphdr *)addr;
    unsigned short src_port = ntohs(tcp_addr->th_sport);
    unsigned short dst_port = ntohs(tcp_addr->th_dport);
    // print the payload
    unsigned char data_offset = (tcp_addr->th_off);
    char *payload = addr + sizeof(int)*data_offset;
    // randomly print the payload
    for(int i=0; i<10; i++)
        printf("%c", payload[i]);
    return 0;
}
```

总结下来，协议的分层组织方式以及包的层级结构，决定了我们能够通过这种逐层的方式，递归解析和处理网络数据包的各个层次。但是，需要特别注意的是，这些分层也不是绝对的，有时也需要层之间的关联信息。例如，如果我们需要分析 TCP 的数据报的载荷，但是 TCP 报文头部并不包含报文的整体总长度；因此，我们需要从 IP 数据包的包头信息中，计算得到该长度并传递过来。

4.4.2 网络嗅探

网络嗅探是一种监视和捕获网络上流动的数据包的技术，允许用户审查数据包的内容和元数据。它可以用于网络故障诊断、性能监测和安全审计。为有效进行嗅探，设备常被设置为“混杂模式”，捕获所有流经的数据包。

混杂模式（Promiscuous Mode）是网络设备的一种特殊工作模式。当网络设备（如网卡）处于此模式时，它会捕获通过它的所有数据包，而不仅仅是发送给其 MAC 地址的数据包。这与其正常工作模式相对，正常情况下，网卡只会接收那些目标 MAC 地址与其自己相匹配的数据包。

下面我们通过开启混杂模式，来讨论网络数据包嗅探器的工作原理。

```
#define PACKET_LEN 65536
int main() {
    unsigned char buffer[PACKET_LEN];
    struct sockaddr saddr;
    int raw_socket = socket(AF_PACKET, SOCK_RAW,
htons(ETH_P_ALL));
    const char *device = "eth0";  // interface name
    struct ifreq ifr;
    memset(&ifr, 0, sizeof(struct ifreq));
    strcpy(ifr.ifr_name, device);
    ioctl(raw_socket, SIOCGIFFLAGS, &ifr);
    ifr.ifr_flags |= IFF_PROMISC;
    ioctl(raw_socket, SIOCSIFFLAGS, &ifr);
    while (1) {
        int saddr_len = sizeof(saddr);
        int data_size = recvfrom(raw_socket, buffer, PACKET_LEN,
0, &saddr, (socklen_t*)&saddr_len);
        printf("Received a packet of size %d bytes\n",
data_size);
    }
    close(raw_socket);
    return 0;
}
```

首先，该程序使用 socket()函数创建了一个原始套接字，通过 AF_PACKET 和 SOCK_RAW 指明了套接字级别为数据链路层，允许捕获所有的以太网帧（第 5 行）。接着，程序为套接字开启了混杂模式（第 11 行），这样可以捕获所有到达网络接口的数据包，而不仅仅是目标地址为该接口的数据包。混杂模式是通过 setsockopt()函数并使用 PACKET_MR_PROMISC 标志设置的。一旦套接字被正确配置，程序将不断地使用 recvfrom()函数将报文收取到缓冲区 buffer 中（第 15 行），并允许对包进行进一步的分析和处理。

4.4.3 包的伪造

数据包伪造指的是修改或创建欺骗性的网络数据包，并将其注入到网络中。通常，伪造的目的是欺骗目标系统，使其认为数据包来自受信任的源。这可能导致网络安全问题，导致访问控制绕过、会话劫持或拒绝服务攻击等。

包的伪造可以分成两个步骤完成：首先，是完成数据包的构建，这利用本章前面第 1 小节讨论的技术不难做到；其次，是利用原始套接字，将构建好的数据报通过特定的网络协议层级发出。

例如，如下程序片段示例，展示了伪造一个以太网数据报并发出的全过程：

```
int forge_ether_packet(char *src_mac, char *dst_mac, char *data){
    struct etherhdr *p = create_ether_packet(src_mac, dst_mac,
IPPROTO_IP, data);
    // create a raw socket
    int raw_sock = socket(...);
    // send data
    sendmsg(raw_sock, p);
}
```

对于更上层的数据报文，其流程非常类似，只是由于协议格式的复杂性，构建数据报的过程相对更繁琐。接下来，我们给出伪造和发送 ICMP 数据报的示例。这个示例仍然是包括上面的两个核心步骤，省略的部分无关代码。

```
int forge_icmp_packet(char *dst_ip, char *data){
    struct icmphdr *p = create_icmp_packet(dst_ip, data);
    // create a raw socket, on ICMP level
    int raw_sock = socket(PF_INET, SOCK_RAW, IPPROTO_ICMP);
    // send data
    send_icmp(raw_sock, dst_ip, p, sizeof(*p)+strlen(data));
}
```

构建 ICMP 数据报的工作由下面的函数 create_icmp_packet 完成。

```
struct icmp *create_icmp_packet(char *dst_ip) {
    static char buf[BUF_SIZE] = {'\0'};
    struct icmp *icmph = buf;
    icmph.icmp_type = ICMP_ECHO;
    icmph.icmp_code = 0;
    icmph.icmp_id = htons(0);
    icmph.icmp_seq = htons(0);
    icmph.icmp_cksum = icmp_checksum((unsigned short *)&icmph,
sizeof(icmph));
    return buf;
}
```

其校验和由如下函数 icmp_checksum 完成。

```
unsigned short icmp_checksum(unsigned short *ptr, int nbytes) {
    long sum;
    unsigned short oddbyte;
    unsigned short answer;
    sum = 0;
    while (nbytes > 1) {
        sum += *ptr++;
        nbytes -= 2;
    }
    if (nbytes == 1) {
        oddbyte = 0;
        *((uint8_t *) &oddbyte) = *(uint8_t *)ptr;
        sum += oddbyte;
    }
    sum = (sum >> 16) + (sum & 0xffff);
    sum += (sum >> 16);
    answer = ~sum;
    return answer;
}
```

最后，函数 send_icmp_packet 将构建好的数据报，通过原始套接字发出。

```
int send_icmp_packet(int sock, char *dst_ip, char *data, int len)
{
    struct sockaddr_in dest_addr;
    dest_addr.sin_family = AF_INET;
    inet_pton(AF_INET, dst_ip, &dest_addr.sin_addr);
    sendto(s, data, len, 0, (struct sockaddr *)&dest_addr,
sizeof(dest_addr));
    return 0;
}
```

需要特别注意的是：由于我们创建的原始套接字工作在 ICMP 层，我们可以直接发送 ICMP 协议的数据报，而不用再构建底层的数据报。

4.5 本章小结

本章主要结合套接字接口，对网络编程进行了全面的讨论。首先，我们介绍了网络模型的架构和协议。然后，我们讨论了套接字，并以 HTTP 协议为例，讨论了基于套接字技术的上层网络协议的实现。接着，我们讨论了原始套接字，并通过讨论其在网络嗅探和包的伪造两方面的应用实例，展示了套接字在协议栈下层的使用方式。

4.6 深入阅读

想要更深入了解网络编程的读者，可以继续阅读《UNIX 网络编程卷 1：套接字 API》。