第 1 章

◂ TCP/IP协议基础 ▸

本章讲述 Windows 网络编程所需的基础理论概念。这是一个很广的话题，如果要全面论述，一本厚书都不够，根本不可能在一章里讲完。本章将主要讲解 Windows 网络编程中经常涉及的 TCP/IP 概念等。

1.1 什么是 TCP/IP

TCP/IP 是 Transmission Control Protocol/Internet Protocol 的简写，中译名为传输控制协议/因特网互联协议，又名网络通信协议，是 Internet 最基本的协议、Internet 国际互联网络的基础。TCP/IP 协议不是指一个协议，也不是 TCP 和 IP 这两个协议的合称，而是一个协议簇，包括多个网络协议，比如了 IP 协议、IMCP 协议、TCP 协议以及我们更加熟悉的 HTTP 协议、FTP 协议、POP3 协议等。TCP/IP 定义了计算机操作系统如何连入因特网，以及数据如何在它们之间传输的标准。

TCP/IP 协议是为了解决不同系统的计算机之间的传输通信而提出的一个标准，不同系统的计算机采用了同一种协议后就能相互进行通信，从而能够建立网络连接，实现资源共享和网络通信了。就像两个不同语言国家的人，都用英语说话后，就能相互交流了。

1.2 TCP/IP 协议的分层结构

TCP/IP 协议簇按照层次由上到下，可以分成 4 层，分别是应用层、传输层、网际层和网络接口层。其中，应用层（Application Layer）包含所有的高层协议，比如虚拟终端协议（TELecommunications NETwork，TELNET）、文件传输协议（File Transfer Protocol，FTP）、电子邮件传输协议（Simple Mail Transfer Protocol，SMTP）、域名服务（Domain Name Service，DNS）、网上新闻传输协议（Net News Transfer Protocol，NNTP）和超文本传输协议（HyperText Transfer Protocol，HTTP）等。TELNET 允许一台机器上的用户登录到远程机器上，并进行工作；FTP 提供有效地将文件从一台机器上移到另一台机器上的方法；SMTP 用于电子邮件的收发；DNS 用于把主机名映射到网络地址；NNTP 用于新闻的发布、检索和获取；HTTP 用于在

WWW 上获取主页。

应用层的下面一层是传输层（Transport Layer），著名的 TCP 协议和 UDP 协议就在这一层。TCP（Transmission Control Protocol，传输控制协议）是面向连接的协议，提供可靠的报文传输和对上层应用的连接服务。为此，除了基本的数据传输外，它还有可靠性保证、流量控制、多路复用、优先权和安全性控制等功能。UDP（User Datagram Protocol，用户数据报协议）是面向无连接的不可靠传输的协议，主要用于不需要 TCP 的排序和流量控制等功能的应用程序。

传输层下面一层是网际层（Internet Layer，也称 Internet 层或网络层），该层是整个 TCP/IP 体系结构的关键部分，其功能是使主机可以把分组发往任何网络，并使分组独立地传向目标。这些分组可能经由不同的网络，到达的顺序和发送的顺序也可能不同。互联网层使用协议有 IP（Internet Protocol，因特网协议）。

网络层下面是网络接口层（Network Interface Layer），或称数据链路层。该层是整个体系结构的基础部分，负责接收 IP 层的 IP 数据包，通过问络向外发送；或接收处理从网络上来的物理帧，抽出 IP 数据包，向 IP 层发送。链路层是主机与网络的实际连接层，下面就是实体线路了（比如以太网络、光纤网络等）。链路层有以太网、令牌环网等标准，链路层负责网卡设备的驱动、帧同步（就是说从网线上检测到什么信号算作新帧的开始）、冲突检测（如果检测到冲突就自动重发）、数据差错校验等工作。交换机是工作在链路层的网络设备，可以在不同的链路层网络之间转发数据帧（比如十兆以太网和百兆以太网之间、以太网和令牌环网之间），由于不同链路层的帧格式不同，交换机要将进来的数据帧拆掉链路层首部重新封装之后再转发。

不同的协议层对本层数据单元有不同的称谓，在传输层叫作数据段，简称段（segment），在网络层叫作数据包（packet）或 IP 包、分组等，在链路层叫作帧（frame），简称数据帧。数据封装成帧后发到传输介质上，到达目的主机后每层协议再剥掉相应的首部，最后将应用层数据交给应用程序处理。

不同层包含不同的协议，我们可以用图 1-1 来表示各个协议及其所在的层。

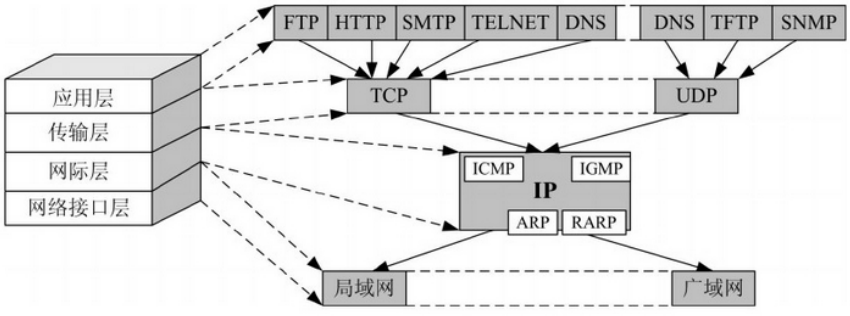


图 1-1

在主机发送端，从传输层开始，会把上一层的数据加上一个报头形成本层的数据，这个过程叫作数据封装；在主机接收端，从最下层开始，每一层数据会去掉首部信息，该过程叫作数据解封，如图 1-2 所示。

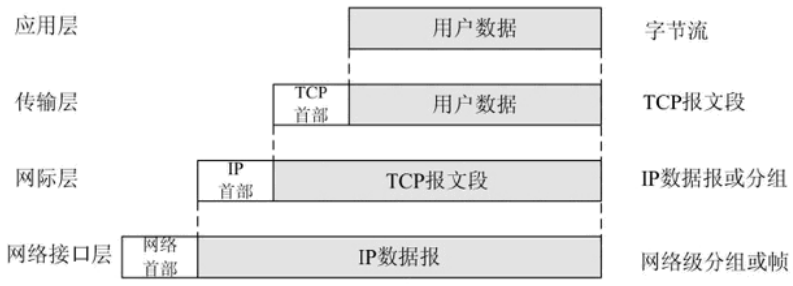


图 1-2

我们来看一个例子。以浏览某个网页为例，看看浏览网页的过程中 TCP/IP 各层做了哪些工作。

发送方：

（1）打开浏览器，输入网址“www.xxx.com”，按回车键，访问网页，其实就是访问 Web 服务器上的网页。在应用层采用的协议是 HTTP，浏览器将网址等信息组成 HTTP 数据，并将数据送给下一层传输层。

（2）传输层在数据前面加上了 TCP 首部，并标记端口为 80（Web 服务器默认端口），将这个数据段传给下一层网络层。

（3）网络层在这个数据段前面加上了自己机器的 IP 和目的 IP，这时这个段被称为 IP 数据包（也可以称为报文），然后将这个 IP 包给了下一层网络接口层。

（4）网络接口层先将 IP 数据包前面加上自己机器的 MAC 地址，以及目的 MAC 地址，这时加上 MAC 地址的数据称为帧，网络接口层通过物理网卡将这个帧以比特流的方式发送到网络上。

互联网上有路由器，它会读取比特流中的 IP 地址进行选路，到达正确的网段，之后这个网段的交换机读取比特流中的 MAC 地址，找到对应要接收的机器。

接收方：

（1）网络接口层用网卡接收到了比特流，读取比特流中的帧，将帧中的 MAC 地址去掉，就成了 IP 数据包，传递给了上一层网络层。

（2）网络层接收了下层传上来的 IP 数据包，将 IP 从包的前面拿掉，取出带有 TCP 的数据（数据段）交给了传输层。

（3）传输层拿到了这个数据段，看到 TCP 标记的端口是 80 端口，说明应用层协议是 HTTP，之后将 TCP 头去掉并将数据交给应用层，告诉应用层对方要求的是 HTTP 的数据。

（4）应用层发送方请求的是 HTTP 数据，调用 Web 服务器程序，把 www.xxx.com 的首页文件发送回去。

如果两台计算机在不同的网段中，那么数据从一台计算机到另一台计算机传输的过程中要经过一个或多个路由器，如图 1-3 所示。

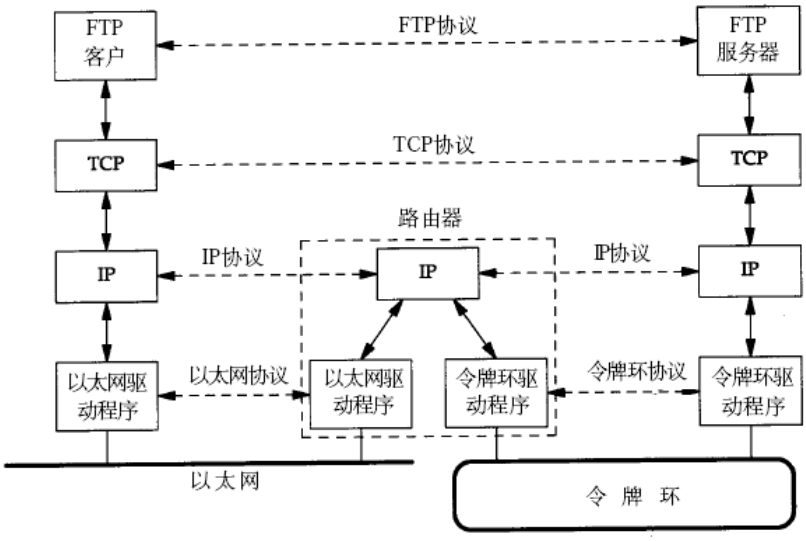


图 1-3

目的主机收到数据包后，如何经过各层协议栈最后到达应用程序呢？整个过程如图 1-4 所示。

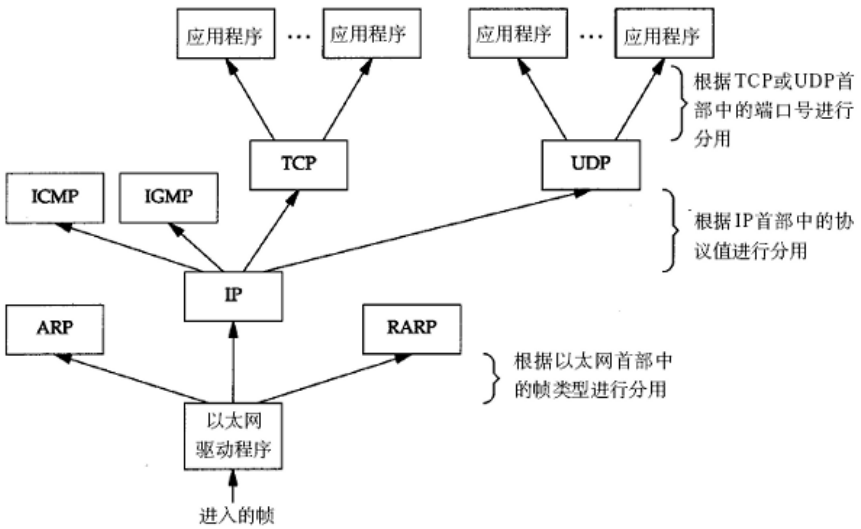


图 1-4

以太网驱动程序首先根据以太网首部中的“上层协议”字段确定该数据帧的有效载荷（payload，指除去协议首部之外实际传输的数据）是 IP、ARP 还是 RARP 协议的数据包，然后交给相应的协议处理。假如是 IP 数据报，IP 协议再根据 IP 首部中的“上层协议”字段确定该数据报的有效载荷是 TCP、UDP、ICMP 还是 IGMP，然后交给相应的协议处理。假如是 TCP 段或 UDP 段，TCP 或 UDP 协议再根据 TCP 首部或 UDP 首部的“端口号”字段确定应该将应用层数据交给哪个用户进程。IP 地址是标识网络中不同主机的地址，而端口号就是同一台主机上标识不同进程的地址，IP 地址和端口号合起来标识网络中唯一的进程。

注意，虽然 IP、ARP 和 RARP 数据报都需要以太网驱动程序来封装成帧，但是从功能上划分，ARP 和 RARP 属于链路层，IP 属于网络层。虽然 ICMP、IGMP、TCP、UDP 的数据都需要 IP 协议来封装成数据报，但是从功能上划分，ICMP、IGMP 与 IP 同属于网络层，TCP 和 UDP 属于传输层。

上面可能说得有点繁杂，这里用一张简图（见图 1-5）来总结一下 TCP/IP 协议模型对数据的封装。

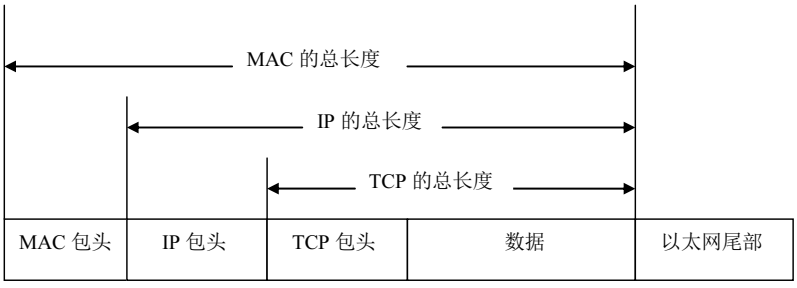


图 1-5

1.3 应用层

应用层位于 TCP/IP 最高层，该层的协议主要有以下几种：

（1）远程登录协议（Telnet）。
（2）文件传送协议（file transfer protocol，FTP）。
（3）简单邮件传送协议（simple mail transfer protocol，SMTP）。
（4）域名系统（domain name system，DNS）。
（5）简单网络管理协议（simple network management protocol，SNMP）。
（6）超文本传送协议（hyperText transfer protocol，HTTP）。
（7）邮局协议（POP3）。

其中，从网络上下载文件时使用的是 FTP 协议，上网游览网页时使用的是 HTTP 协议；在网络上访问一台主机时，通常不直接输入 IP 地址，而是输入域名，用的是 DNS 服务协议，它会将域名解析为 IP 地址；通过 outlook 发送电子邮件时，使用 SMTP 协议，接收电子邮件时会使用 POP3 协议。

1.3.1 DNS

因特网上的主机通过 IP 地址来标识自己，但由于 IP 地址是一串数字，人们记住这个数字去访问主机比较难，因此，因特网管理机构又采用了一串英文来标识一个主机，这串英文是有一定规则的，它的专业术语叫域名（Domain Name）。对用户来讲，用户访问一个网站的时候，

既可以输入该网站的 IP 地址，也可以输入其域名，对访问而言两者是等价的。例如，微软公司的 Web 服务器的域名是 www.microsoft.com，不管用户在浏览器中输入的是 www.microsoft.com 还是 Web 服务器的 IP 地址，都可以访问其 Web 网站。

域名由因特网域名与地址管理机构（Internet Corporation for Assigned Names and Numbers，ICANN）管理，这是为承担域名系统管理、IP 地址分配、协议参数配置，以及主服务器系统管理等职能而设立的非营利机构。ICANN 为不同的国家或地区设置了相应的顶级域名，这些域名通常都由两个英文字母组成。例如，.uk 代表英国、.fr 代表法国、.jp 代表日本。中国的顶级域名是.cn，.cn 下的域名由 CNNIC 进行管理。

域名只是某个主机的别名，并不是真正的主机地址。主机地址只能是 IP 地址，为了通过域名来访问主机，就必须实现域名和 IP 地址之间的转换。这个转换工作就由域名系统（Domain Name System，DNS）来完成。DNS 是因特网的一项核心服务。它作为可以将域名和 IP 地址相互映射的一个分布式数据库，能够使人更方便地访问互联网，而不用去记住能够被机器直接读取的 IP 数字串。一个需要域名解析的用户先将该解析请求发往本地的域名服务器，如果本地的域名服务器能够解析，就直接得到结果；否则本地的域名服务器将向根域名服务器发送请求；依据根域名服务器返回的指针再查询下一层的域名服务器；以此类推，最后得到所要解析域名的 IP 地址。

1.3.2 端口的概念

我们知道，网络上的主机通过 IP 地址来标识自己，方便其他主机上的程序和自己主机上的程序建立通信。主机上需要通信的程序有很多，那么如何才能找到对方主机上的目的程序呢？IP 地址只是用来寻找目的主机的，最终通信还需要找到目的程序。为此，人们提出了端口这个概念，它就是用来标识目的程序的。有了端口，一台拥有 IP 地址的主机可以提供许多服务，比如 Web 服务进程用 80 端口提供 Web 服务、FTP 进程通过 21 端口提供 FTP 服务、SMTP 进程通过 23 端口提供 SMTP 服务，等等。

如果把 IP 地址比作一间旅馆的地址，那么端口就是这家旅馆内某个房间的房号。旅馆的地址只有一个，但房间却有很多个，因此端口也有很多个。端口是通过端口号来标记的，端口号是一个 16 位的无符号整数，范围是从 0 到 65535（2^{16}-1），并且前面 1024 个端口号是留作操作系统使用的，我们自己的应用程序如果要使用端口，通常用 1024 后面的整数作为端口号。

1.4 传输层

传输层为应用层提供会话和数据报通信服务。传输层最重要的两个协议是 TCP 和 UDP。TCP 协议提供一对一、面向连接的可靠通信服务，它能建立连接，对发送的数据包进行排序和确认，并恢复在传输过程中丢失的数据包。与 TCP 不同，UDP 协议提供一对一或一对多、无连接的不可靠通信服务。

1.4.1　TCP 协议

TCP（Transmission Control Protocol，传输控制协议）是面向连接、保证高可靠性（数据无丢失、数据无失序、数据无错误、数据无重复到达）的传输层协议。TCP 协议会给应用层数据加上一个 TCP 头，组成 TCP 报文。TCP 报文首部（TCP 头）的格式如图 1-6 所示。

图 1-6

如果用 C 语言来定义，可以这样写：

```
typedef struct _TCP_HEADER      //TCP 头定义，共 20 个字节
{
 short  sSourPort;              // 源端口号 16bit
 short  sDestPort;              // 目的端口号 16bit
 unsigned int  uiSequNum;       // 序列号 32bit
 unsigned int  uiAcknowledgeNum;// 确认号 32bit
 short  sHeaderLenAndFlag;      // 前 4 位：TCP 头长度；中 6 位：保留；后 6 位：标志位
 short  sWindowSize;            // 窗口大小 16bit
 short  sCheckSum;              // 检验和 16bit
 short  surgentPointer;         // 紧急数据偏移量 16bit
}TCP_HEADER, *PTCP_HEADER;
```

1.4.2　UDP 协议

UDP（User Datagram Protocol，用户数据报协议）是无连接、不保证可靠的传输层协议。它的协议头相对比较简单，如图 1-7 所示。

源端端口	目的端口
用户数据包长度	检验和
数据	

图 1-7

如果用 C 语言来定义，可以这样写：

```
typedef struct  UDP HEADER          // UDP 头定义，共 8 个字节
{
 unsigned short m_usSourPort;       // 源端口号 16bit
 unsigned short m_usDestPort;       // 目的端口号 16bit
 unsigned short m_usLength;         // 数据包长度 16bit
```

```
    unsigned short m_usCheckSum;        // 校验和 16bit
}UDP_HEADER, *PUDP_HEADER;
```

1.5 网络层

网络层向上层提供简单灵活、无连接、尽最大努力交付的数据报服务。该层重要的协议有IP、 ICMP（Internet 控制报文协议）、IGMP（Internet 组管理协议）、ARP（地址转换协议）、RARP（反向地址转换协议）等。

1.5.1 IP 协议

IP（Internet Protocol，网际协议）是 TCP/IP 协议簇中最为核心的协议。它把上层数据包封装成 IP 数据包后进行传输。如果 IP 数据包太大，还要对数据包进行分片后再传输，到了目的地址处再进行组装还原，以适应不同物理网络对一次所能传输数据大小的要求。

1.5.1.1 IP 协议的特点

（1）不可靠

不可靠的意思是它不能保证 IP 数据包能成功地到达目的地。IP 协议仅提供最好的传输服务。发生某种错误时，如某个路由器暂时用完了缓冲区，IP 有一个简单的错误处理算法：丢弃该数据包，然后发送 ICMP 消息包给信源端。任何要求的可靠性必须由上层协议（如 TCP）来提供。

（2）无连接

无连接的意思是 IP 协议并不维护任何关于后续数据包的状态信息。每个数据包的处理是相互独立的。这也说明，IP 数据包可以不按发送顺序接收。如果一信源向相同的信宿发送两个连续的数据包（先是 A，然后是 B），每个数据包都是独立地进行路由选择，可能选择不同的路线，因此 B 可能在 A 到达之前先到达。

（3）无状态

无状态的意思是通信双方不同步传输数据的状态信息，无法处理乱序和重复的 IP 数据包；IP 数据包提供了标识字段，用来唯一标识 IP 数据包，用来处理 IP 分片和重组，不指示接收顺序。

1.5.1.2 IPv4 数据包的包头格式

IPv4 数据包的包头格式如图 1-8 所示。

图 1-8

这里主要说 IPv4 的包头结构，IPv6 结构与之不同。图 1-8 中的“数据”以上部分就是 IP 包头的内容。因为有了选项部分，所以 IP 包头长度是不定长的。如果选项部分没有，那么 IP 包头的长度为（4+4+8+16+16+3+13+8+8+16+32+32）bit=160bit=20 字节，这也就是 IP 包头的最小长度。

- 版本（Version）：占用 4 个比特，标识目前采用的 IP 协议的版本号，一般的值为 0100（IPv4）后 0110（IPv6）。
- 首部长度：IP 包头长度（Header Length）。该字段占用 4 比特，由于在 IP 包头中有变长的可选部分，为了能多表示一些长度，因此采用 4 字节（32 bit）为本字段数值的单位。比如，4 比特最大能表示为 1111，即 15，单位是 4 字节，因此最多能表示的长度为 15 × 4 =60 字节。
- 服务类型（Type of Service，TOS）：占用 8 比特，可用 PPPDTRC0 这 8 个字符来表示。其中，PPP 定义了包的优先级，取值越大，表示数据越重要，含义如表 1-1 所示。

表 1-1　PPP 取值及含义

PPP 取值	含义	PPP 取值	含义
000	普通（Routine）	100	疾速（Flash Override）
001	优先（Priority）	101	关键（Critic）
010	立即（Immediate）	110	网间控制（Internetwork Control）
011	闪速（Flash）	111	网络控制（Network Control）

- D：时延，0 表示普通，1 表示延迟尽量小。
- T：吞吐量，0 表示普通，1 表示流量尽量大。
- R：可靠性，0 表示普通，1 表示可靠性尽量大。
- M：传输成本，0 表示普通，1 表示成本尽量小。
- 0：这是最后一位，被保留，恒定为 0。
- 总长度：占用 16 比特空间，表示以字节为单位的 IP 包的总长度（包括 IP 包头部分和 IP 数据部分）。如果该字段全为 1，就是最大长度了，即 2^{16}-1= 65535 字节≈63.9990234375KB，有些书上写最大是 64KB，其实是达不到的，最大长度只能是 65535 字节，而不是 65536 字节。
- 标识：在协议栈中保持着一个计数器，每产生一个数据报，计数器就加 1，并将此

值赋给标识字段。注意，这个“标识符”并不是序号，IP 是无连接服务，数据报不存在按序接收的问题。当 IP 数据包由于长度超过网络的 MTU（Maximum Transmission Unit,最大传输单元）而必须分片（分片会在后面讲到，意思就是把一个大的网络数据包拆分成一个个小的数据包）时，这个标识字段的值就被复制到所有小分片的标识字段中。相同的标识字段的值使得分片后的各数据包片最后能正确地重装成为原来的大数据包。该字段占用 16 比特。

- 标志（Flags）：占用 3 比特，最高位不使用，第二位称 DF（Don't Fragment）位，DF 位设为 1 时表明路由器不要对该上层数据包分片。如果一个上层数据包无法在不分段的情况下进行转发，那么路由器会丢弃该上层数据包并返回一个错误信息。最低位称 MF（More Fragments）位，为 1 时说明这个 IP 数据包是分片的，并且后续还有数据包，为 0 时说明这个 IP 数据包是分片的，但已经是最后一个分片了。
- 片偏移：该字段的含义是某个分片在原 IP 数据包中的相对位置。第一个分片的偏移量为 0。片偏移以 8 字节为偏移单位。这样，每个分片的长度一定是 8 字节（64 位）的整数倍。该字段占 13 比特。

- 生存时间也称存活时间（Time To Live，TTL）：表示数据包到达目标地址之前的路由跳数。TTL 是由发送端主机设置的一个计数器，每经过一个路由节点就减 1，减到为 0 时，路由就丢弃该数据包，向源端发送 ICMP 差错报文。这个字段的主要作用是防止数据包不断在 IP 互联网络上永不终止地循环转发。该字段占 8 比特。
  - 协议：该字段用来标识数据部分所使用的协议，比如取值 1 表示 ICMP、取值 2 表示 IGMP、取值 6 表示 TCP、取值 17 表示 UDP、取值 88 表示 IGRP、取值 89 表示 OSPF。该字段占 8 比特。
  - 首部校验和（Header Checksum）：用于对 IP 头部的正确性检测，但不包含数据部分。前面提到，每个路由器会改变 TTL 的值，所以路由器会为每个通过的数据包重新计算首部校验和。该字段占 16 比特。
  - 起源和目标地址：用于标识这个 IP 包的起源和目标 IP 地址。值得注意的是，除非使用 NAT（网络地址转换），否则在整个传输的过程中，这两个地址不会改变。这两个地址都占用 32 比特。
  - 选项（可选）：这是一个可变长的字段。该字段属于可选项，主要是给一些特殊的情况使用。最大长度是 40 字节。
  - 填充（Padding）：IP 包头长度（Header Length）这个字段的单位为 32bit，因此必须为 32bit 的整数倍，因此在可选项后面 IP 协议会填充若干个 0，以达到 32bit 的整数倍。

在 Linux 源码中，IP 包头的定义如下：

```
struct iphdr {
#if defined(__LITTLE_ENDIAN_BITFIELD)
 __u8    ihl:4,
     version:4;
```

```
#elif defined (__BIG_ENDIAN_BITFIELD)
 __u8    version:4,
         ihl:4;
#else
#error  "Please fix <asm/byteorder.h>"
#endif
 __u8    tos;
 __be16  tot_len;
 __be16  id;
 __be16  frag_off;
 __u8    ttl;
 __u8    protocol;
 __sum16 check;
 __be32  saddr;
 __be32  daddr;
 /*The options start here. */
};
```

这个定义可以在源码目录的 include/uapi/linux/ip.h 中查到。

1.5.1.3　IP 数据包分片

IP 协议在传输数据包时，将数据包分为若干分片（小数据包）后进行传输，并在目的系统中进行重组。这一过程称为分片（fragmentation）。

要理解 IP 分片，首先要理解下 MTU（最大传输单元，后面数据链路层还会讲到），物理网络一次传送的数据是有最大长度的，因此网络层的下层（数据链路层）的传输单元（数据帧）也有一个最大长度，这个最大长度值就是 MTU，每一种物理网络都会规定链路层数据帧的最大长度，比如以太网的 MTU 为 1500 字节。

IP 协议在传输数据包时，若 IP 数据包加上数据帧头部后长度大于 MTU，则将数据包切分成若干分片（小数据包）后再进行传输，并在目标系统中进行重组。IP 分片既可能在源端主机进行，也可能发生在中间的路由器处，因为不同的网络的 MTU 是不一样的，而传输的整个过程可能会经过不同的物理网络。如果传输路径上某个网络的 MTU 比源端网络的 MTU 要小，路由器就可能对 IP 数据包再次进行分片。分片数据的重组只会发生在目的端的 IP 层。

1.5.1.4　IP 地址的定义

IP 协议中有个概念叫 IP 地址。所谓 IP 地址，就是 Internet 中主机的标识。Internet 中的主机要与别的主机通信，必须具有一个 IP 地址。就像房子要有一个门牌号，这样邮递员才能根据信封上的家庭地址送到目的地。

IP 地址现在有两个版本，分别是 32 位的 IPv4 和 128 位的 IPv6，后者是为了解决前者不够用而产生的。每个 IP 数据包都必须携带目的 IP 地址和源 IP 地址，路由器依靠此信息为数据包选择路由。

这里以 IPv4 为例，IP 地址是由 4 个数字组成的，数字之间用小圆点隔开，每个数字的取值范围在 0~255 之间（包括 0 和 255）。通常有两种表示形式：

（1）十进制表示，比如 192.168.0.1。

（2）二进制表示，比如 11000000.10101000.00000000.00000001。

两种方式可以相互转换，每 8 位二进制数对应一位十进制数，如图 1-9 所示。

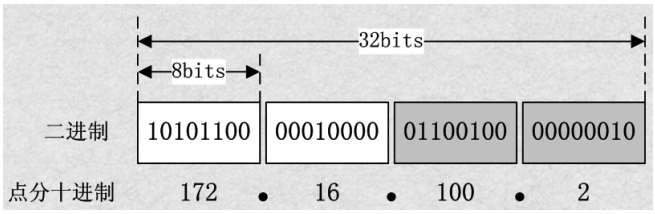


图 1-9

实际应用中多用十进制表示，比如 172.16.100.2。

1.5.1.5 IP 地址的两级分类编址

因特网由很多网络构成，每个网络上都有很多主机，这样便构成了一个有层次的结构。IP 地址在设计的时候就考虑到地址分配的层次特点，把每个 IP 地址分割成网络号（NetID）和主机号（HostID）两部分，网络号表示主机属于互联网中的哪一个网络，而主机号则表示其属于该网络中的哪一台主机，两者之间是主从关系，同一网络中绝对不能有主机号完全相同的两台计算机，否则会报出 IP 地址冲突。IP 地址分为两部分后，IP 数据包从网际上的一个网络到达另一个网络时，选择路径可以基于网络而不是主机。在大型的网际中，这一点优势特别明显，因为路由表中只存储网络信息而不是主机信息，这样可以大大简化路由表，方便路由器的 IP 寻址。

根据网络地址和主机地址在 IP 地址中所占的位数可将 IP 地址分为 A、B、C、D、E 5 类，每一类网络可以从 IP 地址的第一个数字看出，如图 1-10 所示。

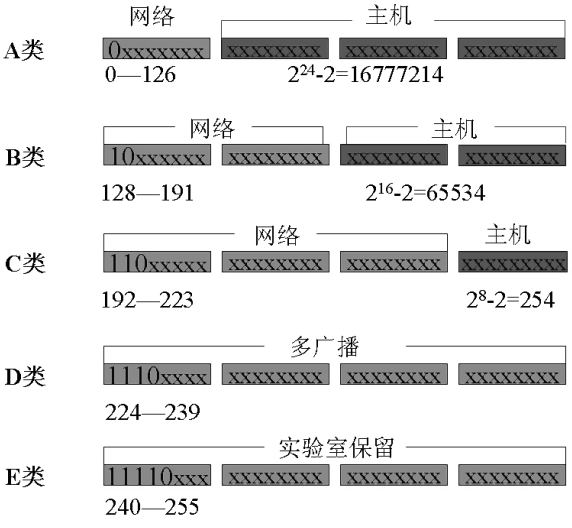


图 1-10

A 类地址的第一位为 0，第二至八位为网络地址，第九至三十二位为主机地址， 这类地址适用于为数不多的主机数大于 2 的 16 次方的大型网络。A 类网络地址的数量最多不超过 126（2 的 7 次方减 2）个，每个 A 类网络最多可以容纳 16777214（2 的 24 次方减 2）台主机。

B 类地址前两位分别为 1 和 0，第三至第十六位为网络地址，第十七至三十二位为主机地址，此类地址用于主机数介于 2 的 8~16 次方之间的中型网络，B 类网络数量最多 16382（2 的 14 次方减 2）个。

C 类地址前三位分别为 1、1、0，四到二十四位为网络地址，其余为主机地址，用于每个网络只能容纳 254（2 的 8 次方减 2）台主机的大量小型网，C 类网络数量上限为（2 的 21 次方减 2）个。

D 类地址前四位为 1、1、1、0， 其余为多目地址。

E 类地址前五位为 1、1、1、1、0，其余位数留待后用。

A 类 IP 的第一个字节范围是 0 到 126，B 类 IP 的第一个字节范围是 128 到 191，C 类 IP 的第一个字节范围是 192 到 223，所以看到 192.X.X.X 肯定是 C 类 IP 地址，大家根据 IP 地址的第一个字节范围就能够推导出该 IP 属于 A 类还是 B 或 C 类。

IP 地址以 A、B、C 两类为主，又以 B、C 两类地址较为常见。除此之外，还有一些特殊用途的 IP 地址：广播地址（主机地址全为 1，用于广播，这里的广播是指同时向网上所有主机发送报文，不是指我们日常听的那种广播）、有限广播地址（所有地址全为 1，用于本网广播）、本网地址（网络地址全 0，后面的主机号表示本网地址）、回送测试地址（127.X.X.X 型，用于网络软件测试及本地机进程间通信）、主机位全 0 地址（这种地址的网络地址就是本网地址）及保留地址（网络号全 1 和 32 位全 0 两种）。由此可见，网络位全 1 或全 0 和主机位全 1 或全 0 都是不能随意分配的。这也就是前面的 A、B、C 类网络的网络数及主机数要减 2 的原因。

总之，主机号全为 0 或全为 1 时分别作为本网络地址和广播地址使用，这种 IP 地址不能分配给用户使用。D 类网络用于广播，可以将信息同时传送到网上的所有设备，而不是点对点的信息传送，可以用来召开电视电话会议。E 类网络常用于进行试验。网络管理员在配置网络时不应该采用 D 类和 E 类网络。我们把特殊的 IP 地址放在表 1-2 中。

表 1-2　特殊 IP 地址及含义

特殊 IP 地址	含义
0.0.0.0	表示默认的路由，这个值用于简化 IP 路由表
127.0.0.1	表示本主机。使用这个地址，应用程序可以像访问远程主机一样访问本主机
网络号全为 0 的 IP 地址	表示本网络的某主机，如 0.0.0.88 将访问本网络中结点为 88 的主机
主机号全为 0 的 IP 地址	表示网络本身
网络号或主机号位全为 1	表示所有主机
255.255.255.255	表示本网络广播

当前，A 类地址已经全部分配完，B 类也不多了，为了有效并连续地利用剩下的 C 类地址，互联网采用 CIDR（Classless Inter-Domain Routing，无类别域间路由方式）把许多 C 类地

址合起来作为 B 类地址分配，整个世界被分为四个地区，每个地区分配一段连续的 C 类地址：欧洲（194.0.0.0～195.255.255.255）、北美（198.0.0.0～199.255.255.255）、中南美（200.0.0.0～201.255.255.255）、亚太地区（202.0.0.0～203.255.255.255）、保留备用（204.0.0.0～223.255.255.255）。这样每一类都约有 3200 万网址供用。

1.5.1.6 网络掩码

在 IP 地址的两级编址中，IP 地址由网络号和主机号两部分组成。如果我们把主机号部分全部置零，此时得到的地址就是网络地址。网络地址可以用于确定主机所在的网络，为此路由器只需计算出 IP 地址中的网络地址，然后跟路由表中存储的网络地址相比较即可知道这个分组应该从哪个接口发送出去。当分组达到目的网络后再根据主机号抵达目的主机。

要计算出 IP 地址中的网络地址，需要借助于网络掩码，或称默认掩码。它是一个 32 位的数，左边连续 n 位全部为 1，后边 32-n 位连续为 0。A、B、C 三类地址的网络掩码分别为 255.0.0.0、255.255.0.0 和 255.255.255.0。我们通过 IP 地址和网络掩码进行与运算，得到的结果就是该 IP 地址的网络地址。网络地址相同的两台主机处于同一个网络中，它们可以直接通信，而不必借助于路由器。

举个例子，现在有两台主机 A 和 B:A 的 IP 地址为 192.168.0.1，网络掩码为 255.255.255.0；B 的 IP 地址为 192.168.0.254，网络掩码为 255.255.255.0。我们先对 A 做运行，把它的 IP 地址和子网掩码每位相与：

IP ： 11010000.10101000.00000000.00000001
子网掩码：11111111. 11111111. 11111111.00000000
AND 运算
网络号： 11000000.10101000.00000000.00000000
转换为十进制： 192.168.0.0

再把 **B** 的 IP 地址和子网掩码每位相与：

IP ： 11010000.10101000.00000000.11111110
子网掩码：11111111. 11111111. 11111111.00000000
AND 运算
网络号： 11000000.10101000.00000000.00000000
转换为十进制： 192.168.0.0

A 和 B 两台主机的网络号是相同的，因此可以认为它们处于同一网络。

IP 地址越来越不够用，为了不浪费，人们又对每类网络进一步划分出子网，为此 IP 地址的编址又有了三级编址的方法，即子网内的某个主机 IP 地址={<网络号>,<子网号>,<主机号>}，该方法中有了子网掩码的概念。后来又提出了超网、无分类编址和 IPv6。限于篇幅，这里不再叙述。

1.5.2　ARP 协议

网络上的 IP 数据包到达最终目的网络后，必须通过 MAC 地址来找到最终目的主机，而数据包中只有 IP 地址，为此需要把 IP 地址转为 MAC 地址，这个工作就由 ARP 协议来完成。ARP 协议是网际层中的协议，用于将 IP 地址解析为 MAC 地址。通常，ARP 协议只适用于局域网中。ARP 协议的工作过程如下：

（1）本地主机在局域网中广播 ARP 请求，ARP 请求数据帧中包含目的主机的 IP 地址。这一步所表达的意思就是“如果你是这个 IP 地址的拥有者，请回答你的硬件地址”。

（2）目的主机收到这个广播报文后，用 ARP 协议解析这份报文，识别出是询问其硬件地址。于是发送 ARP 应答包，里面包含 IP 地址及其对应的硬件地址。

（3）本地主机收到 ARP 应答后，知道了目的地址的硬件地址，之后的数据包就可以传送了。同时，会把目的主机的 IP 地址和 MAC 地址保存在本机的 ARP 表中，以后通信直接查找此表即可。

我们在 Windows 操作系统的命令行下可以使用“arp –a”命令来查询本机 arp 缓存列表，如图 1-11 所示。

图 1-11

另外，可以使“arp -d”命令清除 ARP 缓存表。

ARP 协议通过发送和接收 ARP 报文来获取物理地址的，ARP 报文的格式如图 1-12 所示。

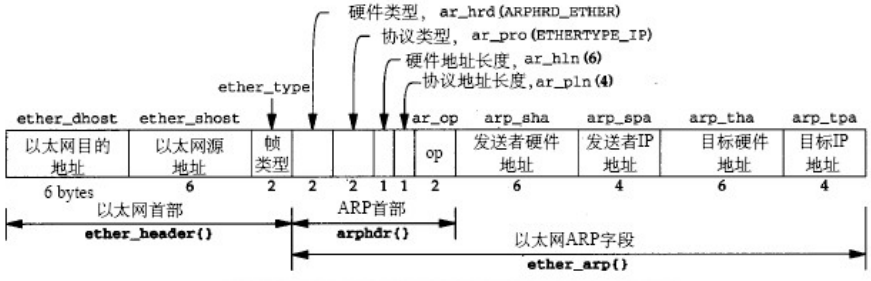


图 1-12

结构 ether_header 定义了以太网帧首部；结构 arphdr 定义了其后的 5 个字段，其信息用于在任何类型的介质上传送 ARP 请求和回答；ether_arp 结构除了包含 arphdr 结构外，还包含源主机和目的主机的地址。如果这个报文格式用 C 语言表述，可以写成这样：

```
//定义常量
#define EPT_IP   0x0800    /* type: IP */
#define EPT_ARP   0x0806    /* type: ARP */
#define EPT_RARP 0x8035    /* type: RARP */
#define ARP_HARDWARE 0x0001    /* Dummy type for 802.3 frames */
#define ARP_REQUEST 0x0001    /* ARP request */
#define ARP_REPLY 0x0002    /* ARP reply */
//定义以太网首部
typedef struct ehhdr
{
unsigned char eh_dst[6];   /* destination ethernet addrress */
unsigned char eh_src[6];   /* source ethernet addresss */
unsigned short eh_type;   /* ethernet pachet type */
}EHHDR, *PEHHDR;
//定义以太网 arp 字段
typedef struct arphdr
{
//arp 首部
unsigned short arp_hrd;    /* format of hardware address */
unsigned short arp_pro;    /* format of protocol address */
unsigned char arp_hln;    /* length of hardware address */
unsigned char arp_pln;    /* length of protocol address */
unsigned short arp_op;     /* ARP/RARP operation */

unsigned char arp_sha[6];    /* sender hardware address */
unsigned long arp_spa;    /* sender protocol address */
unsigned char arp_tha[6];    /* target hardware address */
unsigned long arp_tpa;    /* target protocol address */
}ARPHDR, *PARPHDR;

//定义整个 arp 报文包，总长度 42 字节
typedef struct arpPacket
{
EHHDR ehhdr;
ARPHDR arphdr;
} ARPPACKET, *PARPPACKET;
```

1.5.3 RARP 协议

RARP（Reverse Address Resolution Protocol，逆地址解析协议）允许局域网的物理机器从网关服务器的 ARP 表或者缓存上请求其 IP 地址。比如局域网中有一台主机只知道自己的物理地址而不知道自己的 IP 地址，那么可以通过 RARP 协议发出征求自身 IP 地址的广播请求，然后由 RARP 服务器负责回答。RARP 协议广泛应用于无盘工作站引导时获取 IP 地址。RARP 允许局域网的物理机器从网关服务器 ARP 表或者缓存上请求其 IP 地址。

RARP 协议的工作过程如下：

（1）主机发送一个本地的 RARP 广播，在此广播包中，声明自己的 MAC 地址并且请求任何收到此请求的 RARP 服务器分配一个 IP 地址。

（2）本地网段上的 RARP 服务器收到此请求后，检查其 RARP 列表，查找该 MAC 地址对应的 IP 地址。

（3）如果存在，RARP 服务器就给源主机发送一个响应数据包并将此 IP 地址提供给对方主机使用。

（4）如果不存在，RARP 服务器对此不做任何响应。

（5）源主机收到从 RARP 服务器的响应信息，就利用得到的 IP 地址进行通信。如果一直没有收到 RARP 服务器的响应信息，表示初始化失败。

RARP 的帧格式同 ARP 协议，只是帧类型字段和操作类型不同。

1.5.4　ICMP 协议

ICMP（Internet Control Message Protocol，Internet 控制报文协议）是网络层的一个协议，用于探测网络是否连通、主机是否可达、路由是否可用等。简单地讲，它是用来查询诊断网络的。

虽然和 IP 协议同处网络层，但是 ICMP 报文却是作为 IP 数据包的数据，然后加上 IP 包头后再发送出去的，如图 1-13 所示。

图 1-13

IP 首部的长度为 20 字节。ICMP 报文作为 IP 数据包的数据部分，当 IP 首部的协议字段取值为 1 时其数据部分是 ICMP 报文。ICMP 报文格式如图 1-14 所示。

图 1-14

其中，最上面的（0　8　16　31）指的是比特位，所以前 3 个字段（类型、代码、检验和）一共占了 32 比特（类型占 8 位，代码占 8 位，检验和占 16 位），即 4 字节。所有 ICMP 报文前 4 个字节的格式都是一样的，即任何 ICMP 报文都含有类型、代码和校验和这 3 个字段，

8 位类型和 8 位代码字段一起决定了 ICMP 报文的种类。紧接着后面 4 个字节取决于 ICMP 报文种类。前面 8 个字节就是 ICMP 报文的首部，后面的 ICMP 数据部分的内容和长度取决于 ICMP 报文种类。16 位的检验和字段是对包括选项数据在内的整个 ICMP 数据报文的校验和，其计算方法和 IP 头部校验和的计算方法一样。

ICMP 报文可分为两大类别：差错报告报文和查询报文。每一条（或称每一种）ICMP 报文要么属于差错报告报文，要么属于查询报文，具体如图 1-15 所示。

类型	代码	描　述	查　询	差　错
0	0	回显应答(Ping应答)	•	
3		目的不可达：		
	0	网络不可达		•
	1	主机不可达		•
	2	协议不可达		•
	3	端口不可达		•
	4	需要进行分片但设置了不分片比特		•
	5	源站选路失败		•
	6	目的网络不认识		•
	7	目的主机不认识		•
	8	源主机被隔离（作废不用）		•
	9	目的网络被强制禁止		•
	10	目的主机被强制禁止		•
	11	由于服务类型TOS，网络不可达		•
	12	由于服务类型TOS，主机不可达		•
	13	由于过滤，通信被强制禁止		•
	14	主机越权		•
	15	优先权中止生效		•
4	0	源端被关闭（基本流控制）		•
5		重定向		•
	0	对网络重定向		•
	1	对主机重定向		•
	2	对服务类型和网络重定向		•
	3	对服务类型和主机重定向		•
8	0	请求回显（Ping请求）	•	
9	0	路由器通告	•	
10	0	路由器请求	•	
11		超时：		
	0	传输期间生存时间为0		•
	1	在数据报组装期间生存时间为0		•
12		参数问题：		
	0	坏的IP首部（包括各种差错）		•
	1	缺少必需的选项		•
13	0	时间戳请求	•	
14	0	时间戳应答	•	
15	0	信息请求（作废不用）	•	
16	0	信息应答（作废不用）	•	
17	0	地址掩码请求	•	
18	0	地址掩码应答	•	

图 1-15

从图 1-15 我们可以看出，每一行都是一条（或称每一种）ICMP 报文，要么属于查询，要么属于差错。

1.5.4.1　ICMP 差错报告报文

我们从图 1-15 中可以发现属于差错报告报文的 ICMP 报文蛮多的，为了归纳方便，根据其类型的不同，可以将这些差错报告报文分为 5 种类型：目的不可达（类型=3）、源端被关闭（类型=4）、重定向（类型=5）、超时（类型=11）和参数问题（类型=12）。

从图 1-15 中可以看到，代码字段不同的取值进一步表明了该类型 ICMP 报文的具体情况。比如类型为 3 的 ICMP 报文都是表明目的不可达的，但目的不可达是什么原因呢？此时就用代码字段再进一步说明，比如代码为 0 表示网络不可达、代码为 1 表示主机不可达……

ICMP 协议规定，ICMP 差错报文必须包括产生该差错报文的源数据包的 IP 首部，还必须包括跟在该 IP（源 IP）首部后面的前 8 个字节，这样 ICMP 差错报文的 IP 包长度=本 IP 首部（20 字节）+本 ICMP 首部（8 字节）+ 源 IP 首部（20 字节）+源 IP 包的 IP 首部后的 8 字节=56 字节。我们可以用图 1-16 来表示 ICMP 差错报文。

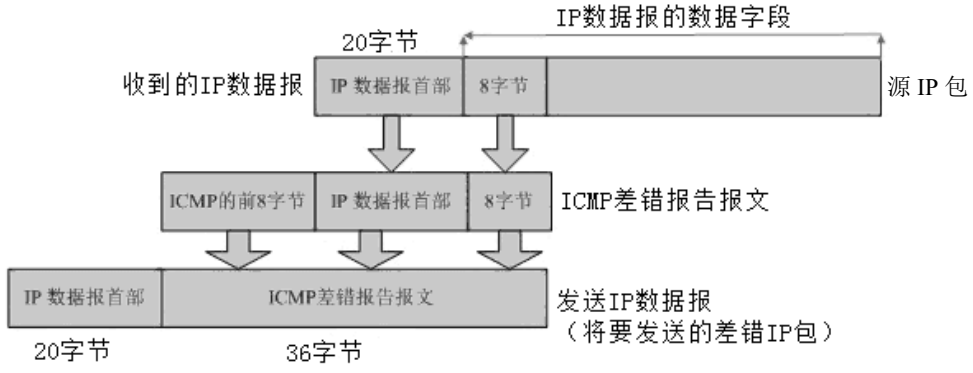


图 1-16

比如我们来看一个具体的 UDP 端口不可达的差错报文，如图 1-17 所示。

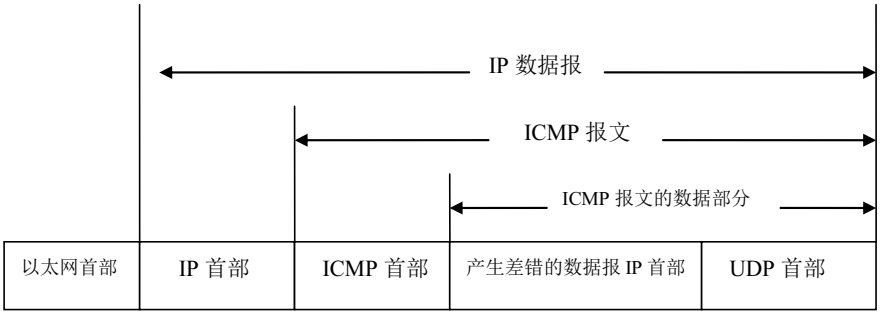


图 1-17

从图 1-17 可以看到 IP 数据包的长度是 56 字节。为了让大家更形象地了解这五大类差错报告报文格式，我们用图形来表示每一类报文：

（1）ICMP 目的不可达报文

目的不可达也称终点不可达，可分为网络不可达、主机不可达、协议不可达、端口不可达、

需要分片但 DF 比特已置为 1 以及源站选路失败等 16 种报文，其代码字段分别置为 0 至 15。当出现以上 16 种情况时就向源站发送目的不可达报文。目的不可达报文的格式如图 1-18 所示。

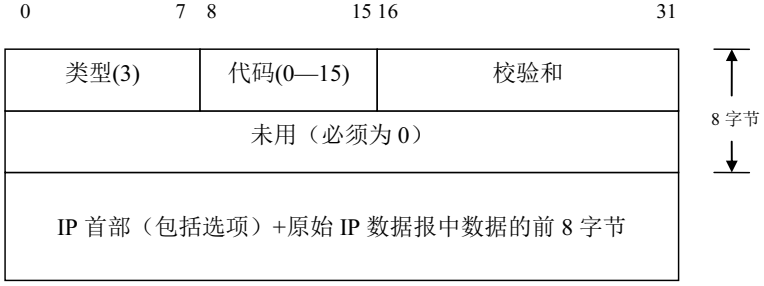


图 1-18

（2）ICMP 源端被关闭报文

也称源站抑制，当路由器或主机由于拥塞而丢弃数据包时，就向源站发送源站抑制报文，使源站知道应当将数据包的发送速率放慢。该类报文格式如图 1-19 所示。

图 1-19

（3）ICMP 重定向报文

当 IP 数据包应该被发送到另一个路由器时，收到该数据包的当前路由器就要发送 ICMP 重定向差错报文给 IP 数据包的发送端。重定向一般用来让具有很少选路信息的主机逐渐建立更完善的路由表。ICMP 重定向报文只能由路由器产生。该类报文格式如图 1-20 所示。

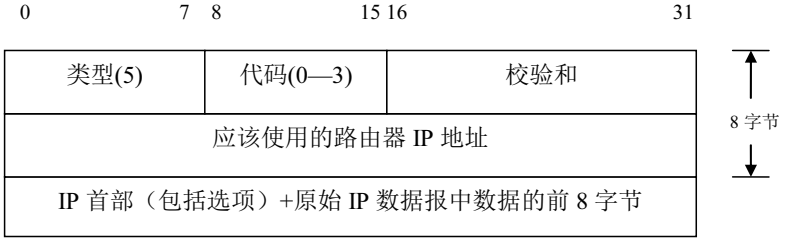


图 1-20

（4）ICMP 超时报文

当路由器收到生存时间为零的数据包时，除丢弃该数据包外，还要向源站发送时间超过报文。当目的站在预先规定的时间内不能收到一个数据包的全部数据包片时，就将已收到的数据

包片都丢弃，并向源站发送时间超时报文。该类报文格式如图 1-21 所示。

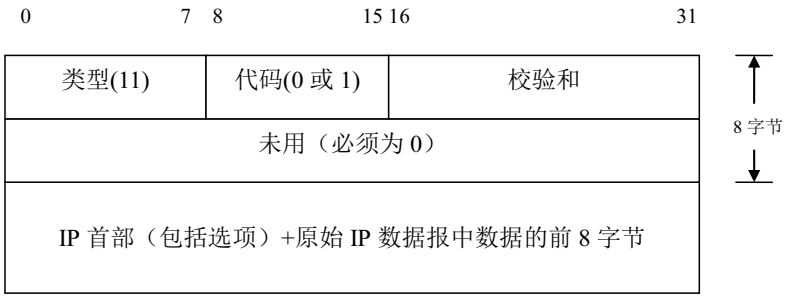
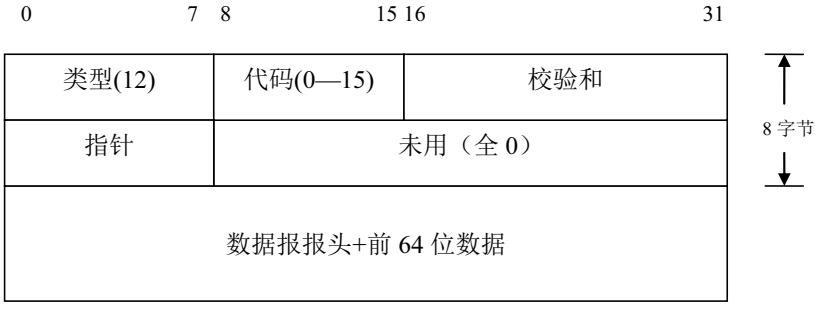


图 1-21

（5）ICMP 参数问题

当路由器或目的主机收到的数据包的首部中的字段值不正确时，就丢弃该数据包，并向源站发送参数问题报文。该类报文格式如图 1-22 所示。



代码为 0 时，数据报某个参数错，指针域指向出错的字节。
代码为 1 时，数据报缺少某个选项，无指针域。

图 1-22

1.5.4.2　ICMP 查询报文

根据功能的不同，ICMP 查询报文可以分为 4 大类：请求回显（Echo）或应答、请求时间戳（Timestamp）或应答、请求地址掩码（Address mask）或应答、请求路由器或通告。请提起精神，后面 ping 编程的时候会用到这方面的理论知识。前面提到，种类由类型和代码字段（见表 1-3）决定。

表 1-3　ICMP 查询报文的类型、代码及含义

类型（TYPE）	代码	含义
8、0	0	回送请求（TYPE=8）、应答（TYPE=0）
13、14	0	时间戳请求（TYPE=13）、应答（TYPE=14）
17、18	0	地址掩码请求（TYPE=17）、应答（TYPE=18）
10、9	0	路由器请求（TYPE=10）、通告（TYPE=9）

这里要提一下回送请求和应答，Echo 的中文翻译为回声，有的文献用回送或回显，本书用“回显”。请求回显的含义就好比请求对方回复一个应答的意思。我们知道 Linux 或 Windows 下有一个 ping 命令，值得注意的是，Linux 下 ping 命令产生的 ICMP 报文大小是 56+8=64 字节，56 是 ICMP 报文数据部分长度，8 是 ICMP 报头部分长度；而 Windows（比如 XP）下 ping 命令产生的 ICMP 报文大小是 32+8=40 字节。该命令就是本机向一个目的主机发送一个请求回显（类型 Type=8）的 ICMP 报文，如果途中没有异常（例如被路由器丢弃、目标不回应 ICMP 或传输失败），则目标返回一个回显应答的 ICMP 报文（类型 Type=0），表明这台主机存在。后面章节还会讲到 ping 命令的抓包和编程。

为了让大家更形象地了解这四类查询报文格式，我们用图形来表示每一类报文。

（1）ICMP 请求回显和应答回显报文格式（见图 1-23）

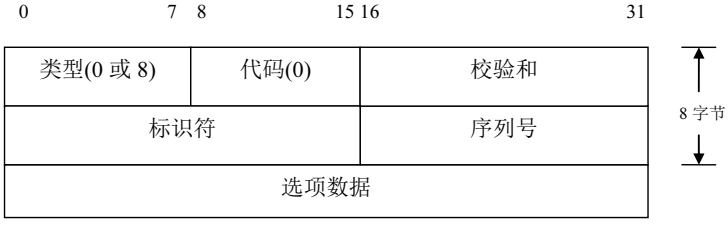


图 1-23

（2）ICMP 时间戳请求和应答报文（见图 1-24）

图 1-24

（3）ICMP 地址掩码请求和应答报文（见图 1-25）

图 1-25

（4）ICMP 路由器请求报文和通告报文（见图 1-26、图 1-27）

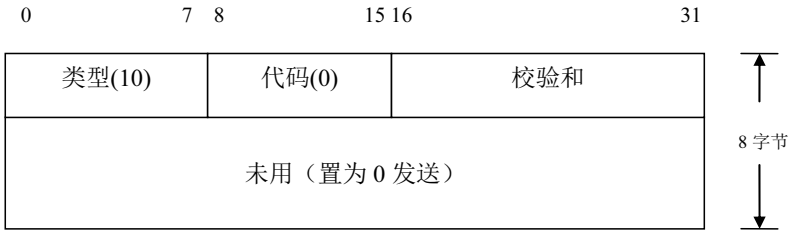


图 1-26

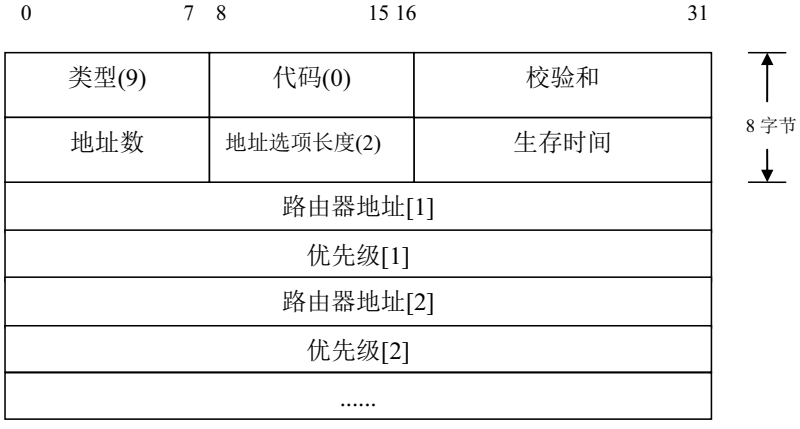


图 1-27

【例 1.1】抓包查看来自 Windows 的 ping 包

（1）启动 Vmware 下的 xp，设置网络连接方式为 NAT，则虚拟机 xp 会连接到虚拟交换机 VMnet8 上。

（2）在 Windows 7 安装并打开抓包软件 Wireshark，选择要捕获网络数据包的网卡是“VMware Virtual Ethernet Adapter for VMnet8”，如图 1-28 所示。

图 1-28

双击图 1-28 中选中的网卡，就开始在该网卡上捕获数据。此时我们在虚拟机 xp（192.168.80.129）下 ping 宿主机（192.168.80.1），可以在 Wireshark 下看到捕获到的 ping 包，图 1-29 是回显请求，我们可以看到 ICMP 报文的数据部分是 32 字节，如果加上 ICMP 报头（8 字节），那就是 40 字节。

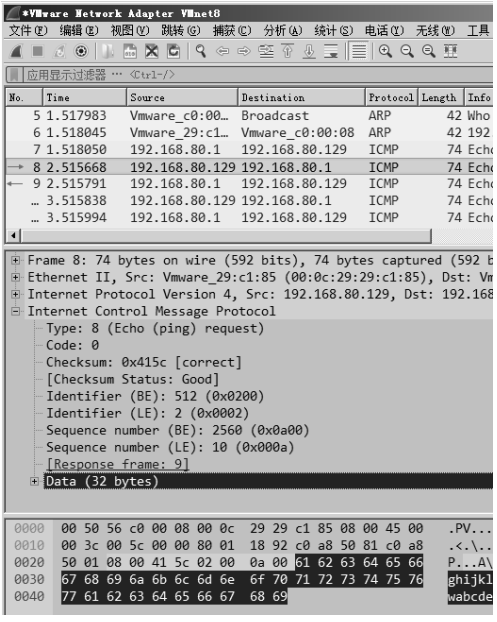


图 1-29

我们可以再一看下回显应答，ICMP 报文的数据部分长度依然是 32 字节，如图 1-30 所示。

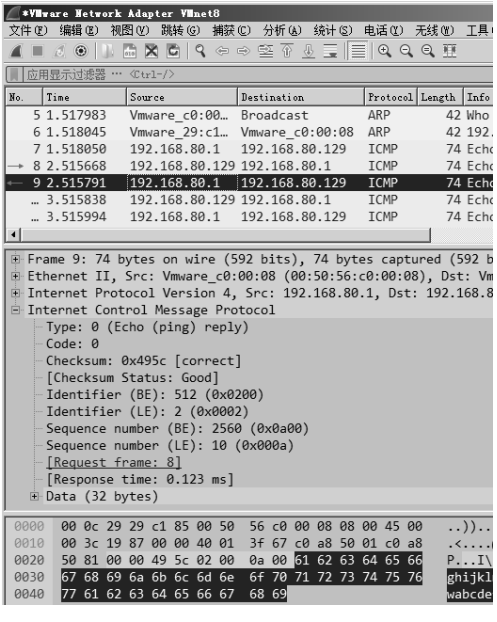


图 1-30

【例 1.2】抓包查看来自 Linux 的 ping 包

（1）启动 Vmware 下的 Linux，设置网络连接方式为 NAT，则虚拟机 Linux 会连接到虚拟交换机 VMnet8 上。

（2）在 Windows 7 中安装并打开抓包软件 Wireshark，选择要捕获网络数据包的网卡是“VMware Virtual Ethernet Adapter for VMnet8”（可参考上例）。

我们在虚拟机 Linux（192.168.80.128）下 ping 宿主机（192.168.80.1），可以在 Wireshark 下看到捕获到的 ping 包。图 1-31 是回显请求，我们可以看到 ICMP 报文的数据部分是 56 字节，如果加上 ICMP 报头（8 字节），那就是 64 字节。

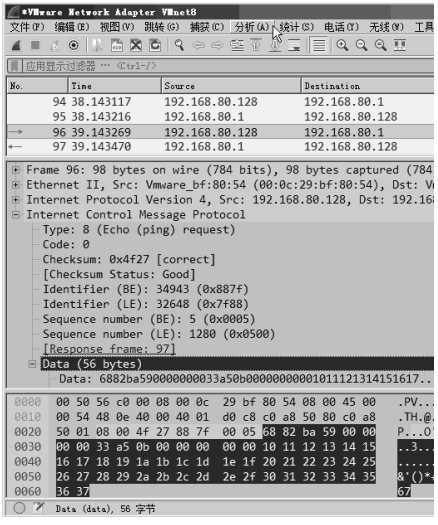


图 1-31

我们可以再看一下回显应答，ICMP 报文的数据部分长度依然是 56 字节，如图 1-32 所示。

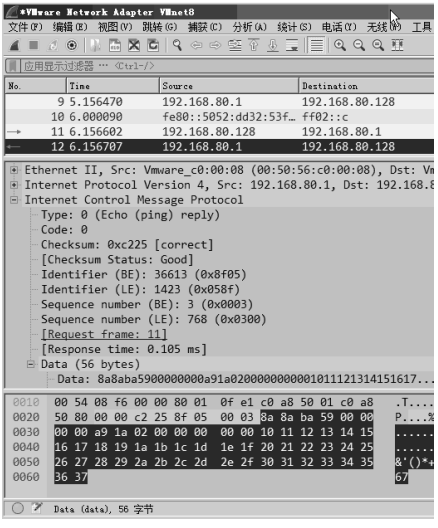


图 1-32

1.6 数据链路层

1.6.1 数据链路层的基本概念

数据链路层最基本的服务是将源计算机网络层来的数据可靠地传输到相邻节点目标计算机的网络层。为达到这一目的，数据链路层主要解决以下 3 个问题：

（1）如何将数据组合成数据块（在数据链路层中将这种数据块称为帧，帧是数据链路层的传送单位）。

（2）如何控制帧在物理信道上的传输，包括如何处理传输差错，如何调节发送速率以使之与接收方相匹配。

（3）在两个网络实体之间提供数据链路通路的建立、维持和释放管理。

1.6.2 数据链路层的主要功能

数据链路层的主要功能如下：

（1）为网络层提供服务

- 无确定的无连接服务，适用于实时通信或者误码率较低的通信信道，如以太网。
- 有确定的无连接服务，适用于误码率较高的通信信道，如无线通信。
- 有确定的面向连接服务，适用于通信要求比较高的场合。

（2）成帧、帧定界、帧同步、透明传输的功能

为了向网络层提供服务，数据链路层必须使用物理层提供的服务。我们知道，物理层是以比特流进行传输的，这种比特流并不保证在数据传输过程中没有错误，接收到的位数量可能少于、等于或者多于发送的位数量，而且它们还可能有不同的值。这时数据链路层为了能实现数据有效的差错控制，就采用了一种“帧”的数据块进行传输。要采帧格式传输，就必须有相应的帧同步技术，这就是数据链路层的“成帧”（也称为“帧同步”）功能。

成帧：两个工作站之间传输信息时，必须将网络层的分组封装成帧，以帧的形式进行传输。将一段数据的前后分别添加首部和尾部就构成了帧。

- 帧定界：首部和尾部中含有很多控制信息，它们的一个重要作用就是确定帧的界限，即帧定界。
- 帧同步：接收方应当能从接收的二进制比特流中区分出帧的起始和终止。
- 透明传输：不管所传数据是什么样的比特组合都能在链路上传输。

（3）差错控制功能

在数据通信过程中可能会因物理链路性能和网络通信环境等因素出现一些传送错误，为了

确保数据通信的准确，就必须使得这些错误发生的概率尽可能低。这一功能也是在数据链路层实现的，就是“差错控制”功能。

（4）流量控制

在双方的数据通信中，如何控制数据通信的流量同样非常重要。它既可以确保数据通信的有序进行，还可以避免通信过程中因为接收方来不及接收而造成的数据丢失。这就是数据链路层的“流量控制”功能。

（5）链路管理

数据链路层的“链路管理”功能包括数据链路的建立、链路的维持和释放三个主要方面。当网络中的两个结点要进行通信时，数据的发送方必须确知接收方是否已处在准备接收的状态。为此，通信双方必须先要交换一些必要的信息，以建立一条基本的数据链路，在传输数据时要维持数据链路，而在通信完毕时释放数据链路。

（6）MAC 寻址

这是数据链路层中 MAC 子层的主要功能。这里所说的“寻址”与“IP 地址寻址”是完全不一样的，因为此处所寻找的地址是计算机网卡的 MAC 地址，也称“物理地址”“硬件地址”“局域网地址（LAN Address）”“以太网地址（Ethernet Address）”，而不是 IP 地址。在以太网中，采用媒体访问控制（Media Access Control，MAC）地址进行寻址，MAC 地址被烧入每个以太网网卡中的。

网络接口层中的数据通常称为 MAC 帧，帧所用的地址为媒体设备地址，即 MAC 地址，也就是通常所说的物理地址。每一块网卡都有一个全世界唯一的物理地址，它的长度固定为 6 字节，比如 00-30-C8-01-08-39。我们在 Linux 操作系统的命令行下用 ifconfig -a 可以看到系统中所有网卡的信息。

MAC 帧的帧头定义如下：

```
typedef struct _MAC_FRAME_HEADER  //数据帧头定义
{
 char  cDstMacAddress[6];    //目的 MAC 地址
 char  cSrcMacAddress[6];    //源 MAC 地址
 short m_cType;              //上一层协议类型，如 0x0800 代表 IP 协议、0x0806 代表 ARP
}MAC_FRAME_HEADER,*PMAC_FRAME_HEADER;
```

1.7 一些容易混淆的术语

1.7.1 MTU

MTU 是最大传输单元（Maximum Transfer Unit）。各种物理网络技术都制定了一个物理

帧的大小，这个大小的限值被称为最大传输单元。

不同物理网络技术的 MTU 不同，寸于一个网络而言,其 MTU 值是由其采用的物理技术决定的,而且通常保持不变。

1.7.2 IP 分组的分片问题

IP 数据包从网络层到了链路层，就会封装成数据帧。如果一个 IP 数据包无法封装在一个数据帧中,就将数据包分成几个长度小于 MTU 的分片。每个分片又叫作数据报，IP 分片也叫作 IP 数据报。然后将分片封装在帧中进行传输。这些分解的分片都传输到目的地后再将这些分片重新组成原来的 IP 数据包。

当一个 IP 数据包从 MTU 大的网络发往 MTU 小的网络时，IP 数据包往往就在路由器上进行分片。

IP 数据包的分片可能在源主机和网络路由器上发生，但重组只在目标主机中进行。

IP 数据包对数据包进行分片时，每一个分片都会独立地成为一个 IP 数据包，分片后的数据包都有自己的 IP 包头和数据区。这一句话很重要，大家要记住，也就是说每个分片（IP 数据报）都有自己的 IP 包头和数据区。

若新网络的 MTU 值不小于原有 MTU，就不必进行分片。

1.7.3 数据段

数据段（segment）是传输层的信息单元。

1.7.4 数据报

数据报（datagram）在不同场合有不同的含义。

第一个场合专指 UDP 数据报，面向无连接的数据传输。采用数据报方式传输时，被传输的分组称为数据报。例如，传输层 TCP 的分组叫作数据段，UDP 的分组叫作数据报。

还有一种场合的数据报是数据包的分组。IP 数据包大于 MTU 值时就需要分片，分成的每片数据是一个 IP 数据报。因此存在分片时，一个完整的 IP 数据包由一个或多个 IP 数据报组成。

1.7.5 数据包

数据包（packet）是网络层传输的数据单元，也称为 IP 包，包中带有足够的寻址信息（IP 地址），可独立地从源主机传输到目的主机。

数据包是 IP 协议中完整的数据单元，由一个或多个数据报组成。也就是说，一个完整的数据包是由若干个数据报组成的。

1.7.6　数据帧

数据帧（frame）是数据链路层的传输单元。为网络层传入的数据添加一个头部和尾部，组成帧。帧根据 MAC 地址寻址。

1.7.7　比特流

比特流（bit）是在物理层的介质上直接实现无结构 bit 流传送的，也就是高低电平信号。

第 2 章

◂ 本机网络信息编程 ▸

俗话说，千里之行，始于足下。网络编程也要从认识自己的电脑网络信息开始。本章将对常见的本机网络信息进行阐述。本章不难，主要是一些函数的使用。所有的本机网络信息都是通过调用 Win32 API 函数获得的。

2.1 获取本地计算机的名称和 IP

网络中的主机通常有一个主机名称和一个或多个 IP 地址。有了这些标识，其他主机就能找到我们，就能和我们通信。

2.1.1 gethostname 函数

gethostname 函数用来检索本地计算机的标准主机名，函数声明如下：

```
int gethostname(char *name,  int  namelen);
```

其中，参数 name 指向接收本地主机名的缓冲区的指针；namelen 表示 name 所指缓冲区的长度，以字节为单位。如果没有出现错误，那么函数返回零；否则，它将返回 SOCKET_ERROR，可以通过调用 WSAGetLastError 来检索特定的错误代码。

例如，我们可以利用下面的代码来获取本地计算机的名称：

```
char szHostName[128];
char szT[20];
if( gethostname(szHostName, 128) == 0 )
  puts("本地计算机名称是:%s", szHostName);
```

2.1.2 gethostbyname 函数

gethostbyname 函数从主机数据库中检索与主机名对应的主机信息，比如 IP 地址等，函数声明如下：

```
hostent * gethostbyname(  const char *name);
```

其中，参数 name 是本地计算机的名称，可以用 gethostname 获得。如果没有出现错误，就将返回指向 hostent 结构的指针；否则，将返回一个空指针，并且可以通过调用 WSAGetLastError 来检索特定的错误号，比如错误号是 WSANOTINITIALISED，表示没有预先成功调用 WSAStartup 函数。

hostent 是一个结构体，定义如下：

```
typedef struct hostent {
  char  *h_name;
  char  **h_aliases;
  short  h_addrtype;
  short  h_length;
  char  **h_addr_list;
} HOSTENT, *PHOSTENT, *LPHOSTENT;
```

其中，h_name 表示主机的正式名称。h_aliases 指向以 NULL 结尾的主机别名数组；h_addrtype 返回地址类型；h_length 表示 ip 地址的长度，ipv4 对应 4 个字节；一般主机可以有多个 ip 地址，比如 www.163.com 就有 121.14.228.43 和 121.11.151.72 两个 ip，h_addr_list 就用来保存多个 ip 地址。

在 Internet 环境下为 AF-INET；h_length 表示地址的字节长度；h_addr_list 指向一个以 NULL 结尾的数组，包含该主机的所有地址。

例如，下面的代码可以获得本机所有的 IP 地址：

```
    struct hostent * pHost;
    int i;
    pHost = gethostbyname(szHostName);
    for( i = 0; pHost!= NULL && pHost->h_addr_list[i]!= NULL; i++ )
    {
         char str[100];
         char addr[20];
         int j;
         LPCSTR psz=inet_ntoa (*(struct in_addr *)pHost->h_addr_list[i]);
         m_IPAddr.AddString(psz);
    }
```

2.1.3　inet_ntoa 函数

inet_ntoa 该函数将一个十进制网络字节序转换为点分十进制 IP 格式的字符串，函数声明如下：

```
char*inet_ntoa(struct  in_addr  in);
```

其中，参数 in 表示 Internet 主机地址的结构，结构体 in_addr 在第 7 章中会介绍。如果函数正确，就返回一个字符指针，指向一块存储着点分格式 IP 地址的静态缓冲区；如果错误，就返回 NULL。

下面我们看一个例子，是一个对话框程序，用来获取本机的名称和 IP 地址。如果大家对

于对话框编程不熟悉，可以参考笔者关于 VC2017 开发的书籍《Visual C++ 2017 从入门到精通》。

【例 2.1】获取本机名称和 IP 地址

（1）新建一个对话框工程，工程名是 test。

（2）切换到资源视图，打开对话框编辑器，在对话框上添加一个编辑框、一个列表框（List Box）和一个按钮。其中，编辑框用来显示本机名称，列表框用来显示本机的 IP 地址。设置按钮的标题是“查询”。为编辑添加控件类型变量 m_HostName，为列表框添加控件变量 m_IPAddr。

（3）设置工程属性。打开工程属性对话框，设置工程为多字节字符工程，如图 2-1 所示。

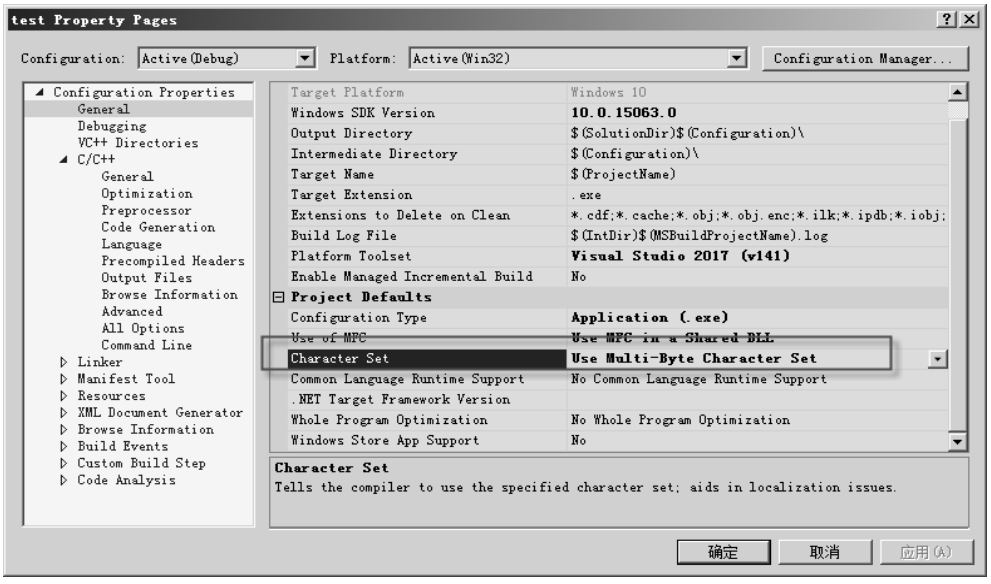


图 2-1

展开“C/C++”→“Preprocessor”，在右边第一行“Preprocessor Definitions”旁的开头添加一个宏：

```
_WINSOCK_DEPRECATED_NO_WARNINGS;
```

注意有个分号。有了这个宏，就可以使用一些传统函数了，而不会出现警告，如图 2-2 所示。

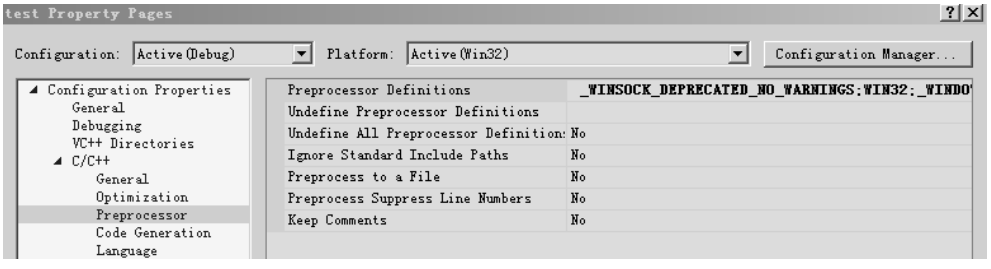


图 2-2

（4）切换类视图，选择 CtestApp，对其成员函数 InitInstance 双击，在 InitInstance 函数中添加 Winsock 库初始化代码：

```
if (!AfxSocketInit())
{
    AfxMessageBox("AfxSocketInit failed");
    return FALSE;
}
```

在 test.h 开头添加头文件包含：

```
#include <afxsock.h>
```

（5）切换到资源视图，在对话框界面中的“查询”按钮上双击，添加事件响应代码：

```
void CtestDlg::OnBnClickedButton1()
{
// TODO: Add your control notification handler code here
char szHostName[128];
char szT[20];

if (gethostname(szHostName, 128) == 0)//获取本机名称
{
    // Get host adresses
    m_HostName.SetWindowText(szHostName);//本机名称显示在编辑框中
    struct hostent * pHost;
    int i;
    pHost = gethostbyname(szHostName);//获取本机网络信息
    for (i = 0; pHost != NULL && pHost->h_addr_list[i] != NULL; i++)
    {
        char str[100];
        char addr[20];
        int j;
        LPCSTR psz = inet_ntoa(*(struct in_addr *)pHost->h_addr_list[i]);
        m_IPAddr.AddString(psz); //把 IP 字符串显示在列表框中
    }
}
}
```

在代码中，首先获取了本机名称，然后显示在编辑框中，接着获取本机网络信息，最后显示在列表框中。

（6）保存工程并运行，结果如图 2-3 所示。

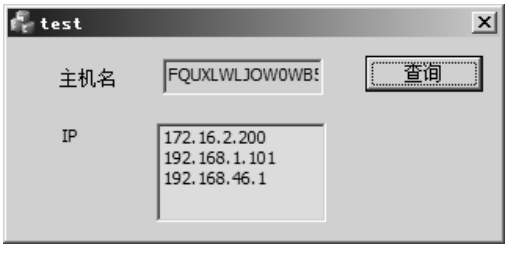


图 2-3

2.2 获取本机子网 IP 地址和子网掩码

子网掩码（subnet mask）又叫网络掩码、地址掩码、子网络遮罩，用来指明一个 IP 地址的哪些位标识的是主机所在的子网以及哪些位标识的是主机的位掩码。子网掩码不能单独存在，必须结合 IP 地址一起使用。子网掩码只有一个作用，就是将某个 IP 地址划分成网络地址和主机地址两部分。

GetAdaptersInfo 函数

GetAdaptersInfo 函数用来检索本地计算机的适配器信息，函数声明如下：

```
ULONG GetAdaptersInfo(PIP_ADAPTER_INFO  pAdapterInfo,  PULONG  pOutBufLen);
```

其中，pAdapterInfo 指向接收 IP 适配器信息结构链表的缓冲区的指针，注意 pAdapterInfo 指向的是一个链表节点的指针；参数 pOutBufLen 指向 ulong 变量的指针，该变量指定 pAdapterInfo 参数指向的缓冲区大小，如果此大小不足以保存适配器信息，pAdapterInfo 将使用所需的大小填充此变量，并返回错误代码 ERROR_BUFFER_OVERFLOW。如果函数成功，返回值为 ERROR_SUCCESS。如果函数失败，返回值是以下错误代码之一：

- ERROR_BUFFER_OVERFLOW：表示接收适配器信息的缓冲区太小。如果 poutbuflen 参数指示的缓冲区大小太小，无法容纳适配器信息，或者 pAdapterInfo 参数为空指针，返回此错误代码时，pOutBufLen 参数指向所需的缓冲区大小。因此，我们可以让 pAdapterInfo 为 NULL 来获得所需缓冲区的大小，然后就可以给 pAdapterInfo 分配空间了。
- ERROR_INVALID_DATA：检索到无效的适配器信息。
- ERROR_INVALID_PARAMETER：存在某个参数无效。如果 pOutBufLen 参数为空指针，或者调用进程对 pOutBufLen 指向的内存没有读/写访问权限，或者调用进程对 pAdapterInfo 参数指向的内存没有写访问权限，就返回此错误。
- ERROR_NO_DATA：本地计算机不存在适配器信息。
- ERROR_NOT_SUPPORTED：本地计算机上运行的操作系统不支持 GetAdaptersInfo 函数。

这个函数在调用的时候，一般分两次调用。第一次调用的时候 pAdapterInfo 设为 NULL，这样 pOutBufLen 将指向获得实际所需缓冲区大小，在第二次调用前就可以为 pAdapterInfo 分配实际所需大小了。下面看例子。

【例 2.2】获取本机 IP 地址和对应掩码

（1）新建一个控制台工程 test。

（2）打开 test.cpp，输入代码：

```
#include "stdafx.h"
#include <atlstr.h>
#include <IPHlpApi.h>
```

```
#include <iostream>
#pragma comment(lib, "Iphlpapi.lib")

using namespace std;

int  tmain(int argc,  TCHAR* argv[])
{
 CString szMark;
 PIP ADAPTER INFO pAdapterInfo=NULL;
 PIP_ADAPTER_INFO pAdapter = NULL;
 DWORD dwRetVal = 0;

 ULONG ulOutBufLen  = sizeof(IP_ADAPTER_INFO);

 // 第一次调用 GetAdapterInfo 获取 ulOutBufLen 大小
 if (GetAdaptersInfo(NULL, &ulOutBufLen) == ERROR_BUFFER_OVERFLOW)
     pAdapterInfo = (IP_ADAPTER_INFO *)malloc(ulOutBufLen);

 if ((dwRetVal = GetAdaptersInfo(pAdapterInfo, &ulOutBufLen)) == NO_ERROR)
 {
     pAdapter = pAdapterInfo;
     while (pAdapter)
     {
         PIP ADDR STRING pIPAddr;
         pIPAddr = &pAdapter->IpAddressList;
         while (pIPAddr)
         {
             cout << "IP:" << pIPAddr->IpAddress.String << endl;
             cout << "Mask:" << pIPAddr->IpMask.String << endl;
             cout << endl;
             pIPAddr = pIPAddr->Next;
         }
         pAdapter = pAdapter->Next;
     }
 }

 if (pAdapterInfo)
     free(pAdapterInfo);

 getchar();
 return 0;
}
```

（3）保存工程并运行，结果如图 2-4 所示。

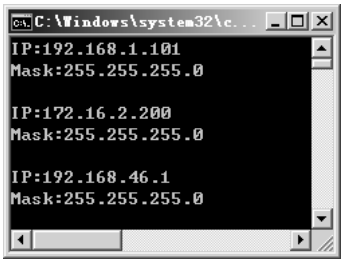


图 2-4

从这个例子和上个例子可以看出，获取本机 IP 地址的方法不止一种。

2.3 获取本机物理网卡地址信息

网卡地址也就是 MAC 地址，是一个用来确认网上设备位置的地址。它的长度是 48 比特（6 字节），由 16 进制的数字组成，分为前 24 位和后 24 位。在 Windows 下，单击【开始】按钮，选择【运行】菜单，输入“cmd”，然后输入“ipconfig /all”（或者输入“ipconfig-all”）就可以看到网卡地址了，如图 2-5 所示。

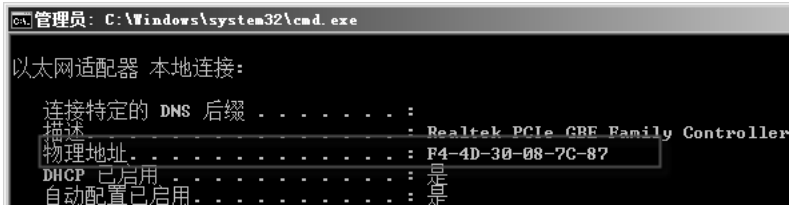


图 2-5

获取网卡 MAC 地址的方法很多，如 Netbios、SNMP、GetAdaptersInfo 等。经过测试发现：Netbios 方法在网线拔出的情况下获取不到 MAC；而 SNMP 方法有时会获取多个重复的网卡的 MAC；还是 GetAdaptersInfo 方法比较好，即使在网线拔出的情况下也可以获取 MAC，而且很准确，不会重复获取网卡。

GetAdaptersInfo 方法也不是十全十美的，也存在一些问题：

- 如何区分物理网卡和虚拟网卡。
- 如何区分无线网卡和有线网卡。
- “禁用”的网卡获取不到。

关于问题 1 和问题 2，笔者的处理办法是：

- 区分物理网卡和虚拟网卡：pAdapter->Description 中包含"PCI"的是物理网卡。（试了 3 台机器可以。）
- 区分无线网卡和有线网卡：pAdapter->Type 为 71 的是无线网卡。（试了 2 个无线网卡可以。）

这些都是笔者的心血啊！希望大家不要再走弯路。关于函数 GetAdaptersInfo，上面已经介绍过了，这里不再赘述。

【例 2.3】获取本机物理网卡的地址信息

（1）新建一个控制台工程 test。

（2）在 test.cpp 中输入如下代码：

```
#include "stdafx.h"
#include <atlbase.h>
#include <atlconv.h>
#include "iphlpapi.h"
#pragma comment ( lib, "Iphlpapi.lib")

int main(int argc, char* argv[])
{
 PIP_ADAPTER_INFO pAdapterInfo;
 PIP_ADAPTER_INFO pAdapter = NULL;
 DWORD dwRetVal = 0;

 pAdapterInfo = (IP_ADAPTER_INFO*)malloc(sizeof(IP_ADAPTER_INFO));
 ULONG ulOutBufLen = sizeof(IP_ADAPTER_INFO);

 if (GetAdaptersInfo(pAdapterInfo, &ulOutBufLen) != ERROR_SUCCESS)
 {
     GlobalFree(pAdapterInfo);
     pAdapterInfo = (IP_ADAPTER_INFO*)malloc(ulOutBufLen);
 }

 if ((dwRetVal = GetAdaptersInfo(pAdapterInfo, &ulOutBufLen)) == NO_ERROR)
 {
     pAdapter = pAdapterInfo;
     while (pAdapter)
     {
// pAdapter->Description 中包含"PCI"的为物理网卡;pAdapter->Type 是 71 的为无线网卡
     if (strstr(pAdapter->Description, "PCI") > 0 || pAdapter->Type == 71)
         {
             printf("------------------------------\n");
             printf("AdapterName: \t%s\n", pAdapter->AdapterName);
             printf("AdapterDesc: \t%s\n", pAdapter->Description);
             printf("AdapterAddr: \t");
             for (UINT i = 0; i <pAdapter->AddressLength; i++)
             {
                 printf("%X%c", pAdapter->Address[i],
                     i == pAdapter->AddressLength - 1 ? '\n' : '-');
             }
             printf("AdapterType: \t%d\n", pAdapter->Type);
             printf("IPAddress: \t%s\n",
pAdapter->IpAddressList.IpAddress.String);
             printf("IPMask: \t%s\n", pAdapter->IpAddressList.IpMask.String);
         }
         pAdapter = pAdapter->Next;
     }
 }
 else
 {
     printf("Callto GetAdaptersInfo failed.\n");
 }
 return 0;
```

```
}
```

（3）保存工程并运行，结果如图 2-6 所示。

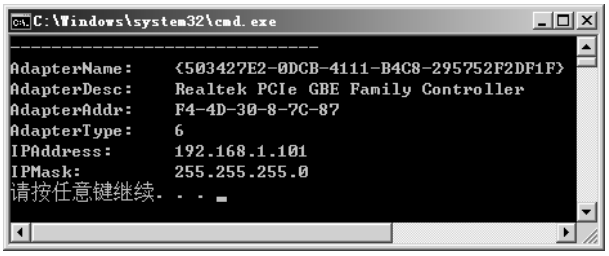


图 2-6

2.4 获取本机所有网卡（包括虚拟网卡）的列表和信息

前一节我们获取了物理网卡的信息。有时候电脑上还存在虚拟网卡，比如安装 VMware 之类的软件，就会自动生成虚拟网卡。本机要获取的是包括虚拟网卡在内的所有网卡的信息，获取的方法依然是使用 GetAdaptersInfo（该函数前面已经介绍过了，这里不再赘述）。

【例 2.4】获取本机所有网卡信息

（1）新建一个控制台工程 test。

（2）打开 test.cpp，输入如下代码：

```
#include "stdafx.h"
#include <Windows.h>
#include <IPHlpApi.h>
#include <iostream>
#pragma comment(lib,"IPHlpApi.lib")
using namespace std;

BOOL GetLocalAdaptersInfo()
{
 //IP_ADAPTER_INFO 结构体
 PIP_ADAPTER_INFO pIpAdapterInfo = NULL;
 pIpAdapterInfo = new IP ADAPTER INFO;

 //结构体大小
 unsigned long ulSize = sizeof(IP ADAPTER INFO);

 //获取适配器信息
 int nRet = GetAdaptersInfo(pIpAdapterInfo, &ulSize);

 if (ERROR_BUFFER_OVERFLOW == nRet)
 {
```

```
        //空间不足，删除之前分配的空间
        delete[]pIpAdapterInfo;

        //重新分配大小
        pIpAdapterInfo = (PIP_ADAPTER_INFO) new BYTE[ulSize];

        //获取适配器信息
        nRet = GetAdaptersInfo(pIpAdapterInfo, &ulSize);

        //获取失败
        if (ERROR_SUCCESS != nRet)
        {
            if (pIpAdapterInfo != NULL)
            {
                delete[]pIpAdapterInfo;
            }
            return FALSE;
        }
    }

    //MAC 地址信息
    char szMacAddr[20];
    //赋值指针
    PIP_ADAPTER_INFO pIterater = pIpAdapterInfo;
    while (pIterater)
    {
        cout << "网卡名称：" << pIterater->AdapterName << endl;

        cout << "网卡描述：" << pIterater->Description << endl;

        sprintf_s(szMacAddr, 20, "%02X-%02X-%02X-%02X-%02X-%02X",
            pIterater->Address[0],
            pIterater->Address[1],
            pIterater->Address[2],
            pIterater->Address[3],
            pIterater->Address[4],
            pIterater->Address[5]);

        cout << "MAC 地址：" << szMacAddr << endl;

        cout << "IP 地址列表：" << endl << endl;

        //指向 IP 地址列表
        PIP_ADDR_STRING pIpAddr = &pIterater->IpAddressList;
        while (pIpAddr)
        {
            cout << "IP 地址：  " << pIpAddr->IpAddress.String << endl;
            cout << "子网掩码：" << pIpAddr->IpMask.String << endl;

            //指向网关列表
            PIP_ADDR_STRING pGateAwayList = &pIterater->GatewayList;
            while (pGateAwayList)
            {
                cout << "网关：    " << pGateAwayList->IpAddress.String << endl;
```

```
                pGateAwayList = pGateAwayList->Next;
            }

            pIpAddr = pIpAddr->Next;
        }
        cout << endl << "---------------------------" << endl;
        pIterater = pIterater->Next;
    }

    //清理
    if (pIpAdapterInfo)
    {
        delete[]pIpAdapterInfo;
    }

    return TRUE;
}

int _tmain(int argc, _TCHAR* argv[])
{
    GetLocalAdaptersInfo();

    cin.get();
    return 0;
}
```

（3）保存工程并运行，结果如图 2-7 所示。

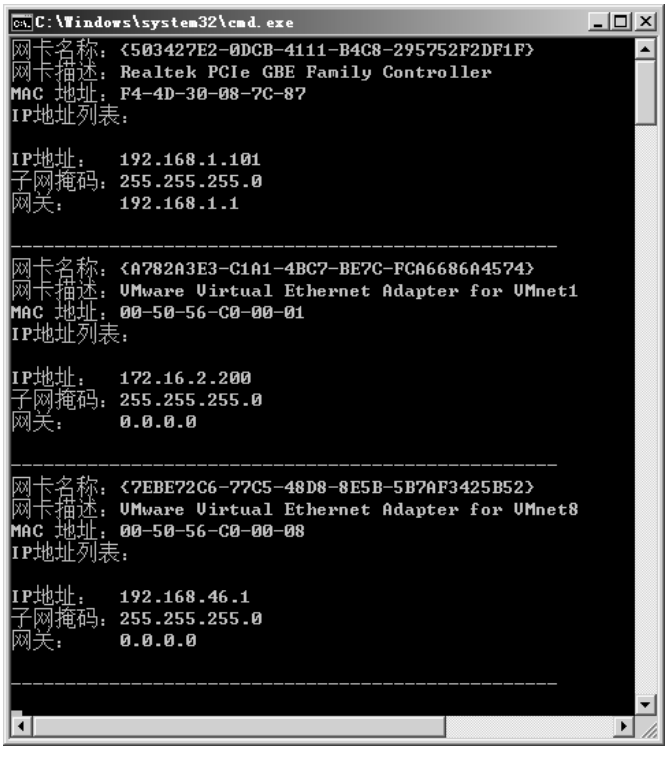


图 2-7

我们可以看到包括 VMware 的虚拟网卡在内的网卡信息也获取到了。

2.5 获取本地计算机的 IP 协议统计数据

通过函数 GetIpStatistics 可以获取当前主机的 IP 协议的统计数据，比如已经收到了多少个数据包。该函数声明如下：

```
ULONG GetIpStatistics(PMIB_IPSTATS pStats);
```

其中，参数 pStats 指向 MIB_IPSTATS 结构的指针，该结构接收本地计算机的 IP 统计信息。如果函数成功，返回值为 NO_ERROR。如果函数失败，返回值是以下错误代码：

- ERROR_INVALID_PARAMETER：pStats 参数为空，或者 GetIpStatistics 无法写入 pStats 参数指向的内存。

结构体 MIB_IPSTATS 的定义如下：

```
typedef struct _MIB_IPSTATS
{
// dwForwarding 指定 IPv4 或 IPv6 的每个协议转发状态，而不是接口的转发状态
    DWORD       dwForwarding;
    DWORD  dwDefaultTTL;              //起始于特定计算机上的数据包的默认初始生存时间
    DWORD       dwInReceives;         //接收到的数据包数
    DWORD       dwInHdrErrors;        //接收到的有头部错误的数据包数
    DWORD       dwInAddrErrors;       //收到的具有地址错误的数据包数
    DWORD       dwForwDatagrams;      //转发的数据包数
    DWORD       dwInUnknownProtos;    //接收到的具有未知协议的数据包数
    DWORD       dwInDiscards;         //丢弃的接收数据包的数目
    DWORD       dwInDelivers;         //已传递的接收数据包的数目
// IP 请求传输的传出数据包数。此数目不包括转发的数据包
    DWORD       dwOutRequests;
    DWORD       dwRoutingDiscards;    //丢弃的传出数据包的数目
    DWORD       dwOutDiscards;        //丢弃的传输数据包数
//此计算机没有到目标 IP 地址的路由的数据包数，这些数据包被丢弃
    DWORD       dwOutNoRoutes;
//允许碎片数据包的所有部分到达的时间量。如果在这段时间内所有数据块都没有到达，数据包将被丢弃
    DWORD       dwReasmTimeout;
    DWORD       dwReasmReqds;         //需要重新组装的数据包数
    DWORD       dwReasmOks;           //成功重新组合的数据包数
    DWORD       dwReasmFails;         //无法重新组合的数据包数
    DWORD       dwFragOks;            //成功分段的数据包数
//由于 IP 头未指定分段而未分段的数据包数，这些数据包被丢弃
    DWORD       dwFragFails;
    DWORD       dwFragCreates;        //创建的片段数
    DWORD       dwNumIf;              //接口的数目
    DWORD       dwNumAddr;            //与此计算机关联的 IP 地址数
    DWORD       dwNumRoutes;          //IP 路由选项卡中的路由数
```

```
} MIB_IPSTATS, *PMIB_IPSTATS;
```

GetIpStatistics 函数返回当前计算机上 IPv4 的统计信息。如果还需要获得 IPv6 的 IP 统计信息，可以用其扩展函数 GetIpStatisticsEx。

【例 2.5】获取本机的 IP 统计数据

（1）新建一个对话框工程，工程名是 Demo。

（2）切换到资源视图，在对话框上放一个列表框和一个按钮。其中，列表框的 ID 是 IDC_LIST。双击按钮，为其添加事件响应代码：

```
void CDemoDlg::OnTest()
{
 CListBox* pListBox = (CListBox*)GetDlgItem(IDC LIST);
 pListBox->ResetContent();

 MIB_IPSTATS IPStats;

 //获得 IP 协议统计信息
 if (GetIpStatistics(&IPStats) != NO_ERROR)
 {
     return;
 }

 CString strText = _T("");
 strText.Format(_T("IP forwarding enabled or disabled:%d"),
     IPStats.dwForwarding);
 pListBox->AddString(strText);
 strText.Format(_T("default time-to-live:%d"),
     IPStats.dwDefaultTTL);
 pListBox->AddString(strText);
 strText.Format(_T("datagrams received:%d"),
     IPStats.dwInReceives);
 pListBox->AddString(strText);
 strText.Format(_T("received header errors:%d"),
     IPStats.dwInHdrErrors);
 pListBox->AddString(strText);
 strText.Format(_T("received address errors:%d"),
     IPStats.dwInAddrErrors);
 pListBox->AddString(strText);
 strText.Format(_T("datagrams forwarded:%d"),
     IPStats.dwForwDatagrams);
 pListBox->AddString(strText);
 strText.Format(_T("datagrams with unknown protocol:%d"),
     IPStats.dwInUnknownProtos);
 pListBox->AddString(strText);
 strText.Format(_T("received datagrams discarded:%d"),
     IPStats.dwInDiscards);
 pListBox->AddString(strText);
 strText.Format(_T("received datagrams delivered:%d"),
```

```
        IPStats.dwInDelivers);
    pListBox->AddString(strText);
    strText.Format(_T("outgoing datagrams requested to send:%d"),
        IPStats.dwOutRequests);
    pListBox->AddString(strText);
    strText.Format(_T("outgoing datagrams discarded:%d"),
        IPStats.dwOutDiscards);
    pListBox->AddString(strText);
    strText.Format(_T("sent datagrams discarded:%d"),
        IPStats.dwOutDiscards);
    pListBox->AddString(strText);
    strText.Format(_T("datagrams for which no route exists:%d"),
        IPStats.dwOutNoRoutes);
    pListBox->AddString(strText);
    strText.Format(_T("datagrams for which all frags did not arrive:%d"),
        IPStats.dwReasmTimeout);
    pListBox->AddString(strText);
    strText.Format(_T("datagrams requiring reassembly:%d"),
        IPStats.dwReasmReqds);
    pListBox->AddString(strText);
    strText.Format(_T("successful reassemblies:%d"),
        IPStats.dwReasmOks);
    pListBox->AddString(strText);
    strText.Format(_T("failed reassemblies:%d"),
        IPStats.dwReasmFails);
    pListBox->AddString(strText);
    strText.Format(_T("successful fragmentations:%d"),
        IPStats.dwFragOks);
    pListBox->AddString(strText);
    strText.Format(_T("failed fragmentations:%d"),
        IPStats.dwFragFails);
    pListBox->AddString(strText);
    strText.Format(_T("datagrams fragmented:%d"),
        IPStats.dwFragCreates);
    pListBox->AddString(strText);
    strText.Format(_T("number of interfaces on computer:%d"),
        IPStats.dwNumIf);
    pListBox->AddString(strText);
    strText.Format(_T("number of IP address on computer:%d"),
        IPStats.dwNumAddr);
    pListBox->AddString(strText);
    strText.Format(_T("number of routes in routing table:%d"),
        IPStats.dwNumRoutes);
    pListBox->AddString(strText);
    }
```

在 DemoDlg.cpp 开头包括文件和引用库文件：

```
#include <Iphlpapi.h>
#pragma comment(lib,"IPHlpApi.lib")
```

（3）保存工程并运行，结果如图 2-8 所示。

图 2-8

2.6 获取本机的 DNS 地址

DNS（Domain Name System，域名系统）是互联网的一项服务。它作为将域名和 IP 地址相互映射的一个分布式数据库，能够使人更方便地访问互联网。DNS 使用 TCP 和 UDP 端口 53。当前，对于每一级域名长度的限制是 63 个字符，域名总长度则不能超过 253 个字符。

DNS 是万维网上作为域名和 IP 地址相互映射的一个分布式数据库，能够使用户更方便地访问互联网，而不用去记住能够被机器直接读取的 IP 数串。DNS 查询过程如图 2-9 所示。

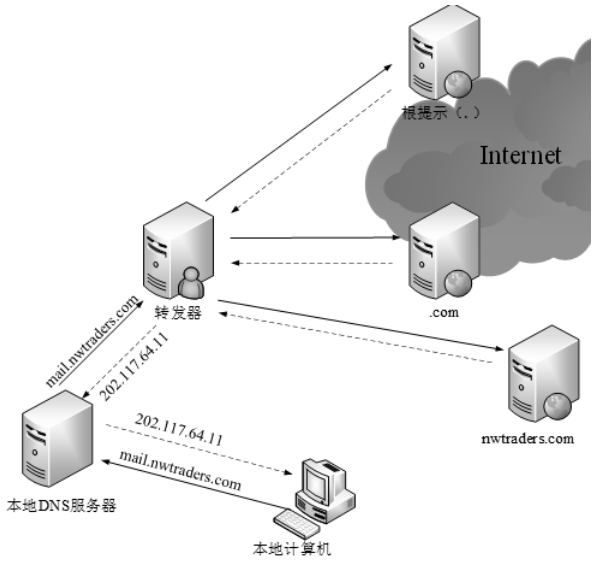


图 2-9

通过函数 GetNetworkParams 可以获得本机上所有配置好的 DNS 地址。该函数声明如下：

```
DWORD GetNetworkParams(PFIXED_INFO pFixedInfo,PULONG     pOutBufLen);
```

其中，参数 pFixedInfo 指向一个缓冲区的指针，该缓冲区包含一个固定的信息结构，该结构接收本地计算机的网络参数（如果函数成功），调用 GetNetworkParams 函数之前，调用方必须分配此缓冲区；pOutBufLen 指向一个 ULONG 变量的指针，该变量指定固定信息结构的大小。如果此大小不足以容纳信息，函数将使用所需大小填充此变量，并返回错误代码 ERROR_BUFFER_OVERFLOW。如果函数成功，返回值为 ERROR_SUCCESS；如果函数失败，将返回错误代码。

【例 2.6】获取本机所有 DNS 地址

（1）新建一个控制台工程 test。

（2）在 test.cpp 中输入代码如下：

```
#include "stdafx.h"
#include <windows.h>
#include <Iphlpapi.h>
#pragma comment(lib,"IPHlpApi.lib")

int main()
{
 DWORD nLength = 0;
 //先获取实际大小，并存入 nLength
 if (GetNetworkParams(NULL, &nLength) != ERROR_BUFFER_OVERFLOW)
 {
     return -1;
 }
 //根据实际所需大小，分配空间
 FIXED_INFO* pFixedInfo = (FIXED_INFO*)new BYTE[nLength];

 //获得本地计算机网络参数
 if (GetNetworkParams(pFixedInfo, &nLength) != ERROR_SUCCESS)
 {
     delete[] pFixedInfo;
     return -1;
 }

 //获得本地计算机 DNS 服务器地址
 char strText[500] =  "本地计算机的 DNS 地址：\n";
 IP_ADDR_STRING* pCurrentDnsServer = &pFixedInfo->DnsServerList;
 while (pCurrentDnsServer != NULL)
 {
     char strTemp[100] = "";
     sprintf(strTemp, "%s\n", pCurrentDnsServer->IpAddress.String);
     strcat(strText, strTemp);
     pCurrentDnsServer = pCurrentDnsServer->Next;
 }
 puts(strText);
```

```
    delete[] pFixedInfo;

    return 0;
}
```

（3）保存工程并运行，运行结果如图 2-10 所示。

图 2-10

此时大家可以通过 ipconfig/all 来对比确认。

2.7 获取本机上的 TCP 统计数据

前面有例子获取了本机的 IP 协议统计数据，现在我们来获取 TCP 协议的统计数据。该功能可以通过函数 GetTcpStatistics 实现，该函数声明如下：

```
ULONG GetTcpStatistics(  PMIB_TCPSTATS pStats);
```

其中，参数 pStats 指向 MIB_TCPSTATS 结构的指针，该结构接收本地计算机的 TCP 统计信息。如果函数成功，返回值为 NO_ERROR；如果函数失败，返回值是以下错误代码：

- ERROR_INVALID_PARAMETER：pStats 参数为空，或者 GetTcpStatistics 无法写入 pStats 参数指向的内存。

结构体 MIB_TCPSTATS 定义如下：

```
typedef struct _MIB_TCPSTATS
{
    DWORD      dwRtoAlgorithm;       //正在使用的重传超时（RTO）算法
    DWORD      dwRtoMin;             // 以毫秒为单位的最小 RTO 值
    DWORD      dwRtoMax;             // 以毫秒为单位的最大 RTO 值
    DWORD   dwMaxConn;// 最大连接数。若此成员为-1，则最大连接数是可变的
    //活动打开的次数。在活动打开状态下，客户端正在启动与服务器的连接
    DWORD      dwActiveOpens;
    //被动打开的次数。在被动打开中，服务器正在侦听来自客户端的连接请求
    DWORD      dwPassiveOpens;
    DWORD      dwAttemptFails;       // 连接尝试失败的次数
    DWORD      dwEstabResets;        // 已重置的已建立连接数
    DWORD      dwCurrEstab;          // 当前建立的连接数
    DWORD      dwInSegs;             // 接收的段数
```

```
    DWORD       dwOutSegs;              // 传输的段数。此数字不包括重新传输的段
    DWORD       dwRetransSegs;          // 重新传输的段数
    DWORD       dwInErrs;               // 收到的错误数
    DWORD       dwOutRsts;              // 使用重置标志集传输的段数
    //系统中当前存在的连接数。此总数包括除侦听连接之外所有状态的连接
    DWORD       dwNumConns;
} MIB_TCPSTATS, *PMIB_TCPSTATS;
```

【例 2.7】获取本机 TCP 协议的统计数据

（1）新建一个对话框工程 Demo。

（2）切换到资源视图，在对话框上放一个列表框和一个按钮。其中，列表框的 ID 是 IDC_LIST。双击按钮，为其添加事件响应代码：

```
void CDemoDlg::OnTest()
{
 CListBox* pListBox = (CListBox*)GetDlgItem(IDC_LIST);
 pListBox->ResetContent();

 MIB_TCPSTATS TCPStats;

 //获得 TCP 协议统计信息
 if (GetTcpStatistics(&TCPStats) != NO_ERROR)
 {
     return;
 }

 CString strText = _T("");
 strText.Format(_T("time-out algorithm:%d"),
     TCPStats.dwRtoAlgorithm);
 pListBox->AddString(strText);
 strText.Format(_T("minimum time-out:%d"),
     TCPStats.dwRtoMin);
 pListBox->AddString(strText);
 strText.Format(_T("maximum time-out:%d"),
     TCPStats.dwRtoMax);
 pListBox->AddString(strText);
 strText.Format(_T("maximum connections:%d"),
     TCPStats.dwMaxConn);
 pListBox->AddString(strText);
 strText.Format(_T("active opens:%d"),
     TCPStats.dwActiveOpens);
 pListBox->AddString(strText);
 strText.Format(_T("passive opens:%d"),
     TCPStats.dwPassiveOpens);
 pListBox->AddString(strText);
 strText.Format(_T("failed attempts:%d"),
     TCPStats.dwAttemptFails);
 pListBox->AddString(strText);
 strText.Format(_T("established connections reset:%d"),
     TCPStats.dwEstabResets);
```

```
    pListBox->AddString(strText);
    strText.Format(_T("established connections:%d"),
        TCPStats.dwCurrEstab);
    pListBox->AddString(strText);
    strText.Format(_T("segments received:%d"),
        TCPStats.dwInSegs);
    pListBox->AddString(strText);
    strText.Format(_T("segment sent:%d"),
        TCPStats.dwOutSegs);
    pListBox->AddString(strText);
    strText.Format(_T("segments retransmitted:%d"),
        TCPStats.dwRetransSegs);
    pListBox->AddString(strText);
    strText.Format(_T("incoming errors:%d"),
        TCPStats.dwInErrs);
    pListBox->AddString(strText);
    strText.Format(_T("outgoing resets:%d"),
        TCPStats.dwOutRsts);
    pListBox->AddString(strText);
    strText.Format(_T("cumulative connections:%d"),
        TCPStats.dwNumConns);
    pListBox->AddString(strText);
    }
```

在 DemoDlg.cpp 开头包含头文件和引用库文件：

```
#include <Iphlpapi.h>                    //包含头文件
#pragma comment(lib,"IPHlpApi.lib")      //引用库文件
```

（3）保存工程并运行，运行结果如图 2-11 所示。

图 2-11

2.8 获取本机上的 UDP 统计数据

前面有例子获取了本机的 TCP 协议统计数据，现在我们来获取 UDP 协议的统计数据。该功能可以通过函数 GetUdpStatistics 实现，函数声明如下：

```
ULONG GetUdpStatistics(  PMIB_UDPSTATS pStats);
```

其中，参数 pStats 指向接收本地计算机的 UDP 统计信息的 MIB_UDPTABLE 结构的指针，PMIB_UDPSTATS 是 MIB_UDPTABLE 结构的指针类型。如果函数成功，返回值为 NO_ERROR；如果函数失败，使用 FormatMessage 获取返回错误的消息字符串。

结构体 MIB_UDPSTATS 定义如下：

```
typedef struct _MIB_UDPSTATS
{
    DWORD       dwInDatagrams;        // 接收的数据包数
    DWORD       dwNoPorts;            //由于指定的端口无效而丢弃的接收的数据包数
    //接收到的错误数据包的数目。此数字不包括 dwNoPorts 成员包含的值
    DWORD       dwInErrors;
    DWORD       dwOutDatagrams;       // 传输的数据包数
    DWORD       dwNumAddrs;           // UDP 侦听器表中的条目数
} MIB_UDPSTATS,*PMIB_UDPSTATS;
```

要获取 IPv6 协议的 UDP 统计信息，可以使用其扩展函数 GetUdpStatisticsEx。

【例 2.8】获取本机的 UDP 协议的统计数据

（1）新建一个对话框工程 Demo。

（2）切换到资源视图，在对话框上放一个列表框和一个按钮。其中，列表框的 ID 是 IDC_LIST。双击按钮，为其添加事件响应代码：

```
void CDemoDlg::OnTest()
{
 CListBox* pListBox = (CListBox*)GetDlgItem(IDC_LIST);
 pListBox->ResetContent();

 MIB_UDPSTATS UDPStats;

 //获得 UDP 协议统计信息
 if (GetUdpStatistics(&UDPStats) != NO_ERROR)
 {
     return;
 }

 CString strText = _T("");
 strText.Format(_T("received datagrams:%d\t\n"),
     UDPStats.dwInDatagrams);
 pListBox->AddString(strText);
 strText.Format(_T("datagrams for which no port exists:%d\t\n"),
```

```
    UDPStats.dwNoPorts);
pListBox->AddString(strText);
strText.Format(_T("errors on received datagrams:%d\t\n"),
    UDPStats.dwInErrors);
pListBox->AddString(strText);
strText.Format(_T("sent datagrams:%d\t\n"),
    UDPStats.dwOutDatagrams);
pListBox->AddString(strText);
strText.Format(_T("number of entries in UDP listener table:%d\t\n"),
    UDPStats.dwNumAddrs);
pListBox->AddString(strText);
}
```

在 DemoDlg.cpp 开头包含头文件和引用库文件：

```
#include <Iphlpapi.h>
#pragma comment(lib,"IPHlpApi.lib")
```

（3）保存工程并运行，运行结果如图 2-12 所示。

图 2-12

2.9 获取本机上支持的网络协议信息

可以通过函数 WSAEnumProtocols 检索有关可用网络传输协议的信息。该函数声明如下：

```
int WSAAPI WSAEnumProtocols( LPINT             lpiProtocols,
  LPWSAPROTOCOL_INFOA lpProtocolBuffer,
  LPDWORD           lpdwBufferLength);
```

其中，参数 lpiProtocols 指向协议值数组；lpProtocolBuffer 指向用 WSAPROTOCOL_INFOA 结构填充的缓冲区的指针；lpdwBufferLength 在输入时，传递给 WSAEnumProtocols 的 lpProtocolBuffer 缓冲区中的字节数。输出时，可以传递给 WSAEnumProtocols 以检索所有请求信息的最小缓冲区大小。如果函数没有出现错误，WSAEnumProtocols 将返回要报告的协议数；否则，将返回 SOCKET_ERROR 的值，并且可以通过调用 WSAGetLastError 来检索特定的错误代码。

值得注意的是，在调用 WSAEnumProtocols 之前要先调用 WSAStartup 函数，否则会得到 WSANOTINITIALISED 的错误码。WSAStartup 启动对 winsock dll 的使用。另外，使用了 WSAStartup 后，结束的时候要用 WSACleanup 进行清理，这两个函数要配套使用。

【例 2.9】获取本机上支持的网络协议信息

（1）新建一个对话框工程 Demo。

（2）切换到资源视图，在对话框上放一个列表框和一个按钮。其中，列表框的 ID 是 IDC_LIST。双击按钮，为其添加事件响应代码：

```
void CDemoDlg::OnTest()
{
 //初始化 WinSock
 WSADATA WSAData;
 if (WSAStartup(MAKEWORD(2,0), &WSAData)!= 0)
 {
     return;
 }

 int nResult = 0;

 //获得需要的缓冲区大小
 DWORD nLength = 0;
 nResult = WSAEnumProtocols(NULL, NULL, &nLength);
 if (nResult != SOCKET_ERROR)
 {
     return;
 }
 if (WSAGetLastError() != WSAENOBUFS)
 {
     return;
 }

 WSAPROTOCOL_INFO* pProtocolInfo = (WSAPROTOCOL_INFO*)new BYTE[nLength];

 //获得本地计算机协议信息
 nResult = WSAEnumProtocols(NULL, pProtocolInfo, &nLength);
 if (nResult == SOCKET_ERROR)
 {
     delete[] pProtocolInfo;
     return;
 }
 for (int n = 0; n < nResult; n++)
 {
     m_ctrlList.AddString(pProtocolInfo[n].szProtocol);
 }

 delete[] pProtocolInfo;
```

```
    //清理 WinSock
    WSACleanup();
    }
```

在 DemoDlg.cpp 开头包含头文件和引用库文件：

```
#include <Winsock2.h>
#pragma comment(lib,"Ws2_32.lib")
```

（3）保存工程并运行，运行结果如图 2-13 所示。

图 2-13

2.10 获取本地计算机的域名

域名（Domain Name）或称网域，是由一串用点分隔的名字组成的 Internet 上某一台计算机或计算机组的名称，用于在数据传输时标识计算机的电子方位（有时也指地理位置）。

可以通过函数 GetNetworkParams 来获取本地计算机的域名。这个函数其实可以检索本地计算机的网络参数，包括域名、主机名等。当然如果本机没有设置域名，那么得到的域名字段内容就是一个空字符串。该函数声明如下：

```
DWORD GetNetworkParams(PFIXED_INFO pFixedInfo,  PULONG     pOutBufLen);
```

其中，参数 pFixedInfo 指向一个缓冲区的指针，该缓冲区包含一个固定的信息结构，该结构接收本地计算机的网络参数（如果函数成功），调用 GetNetworkParams 函数之前，调用方必须分配正确大小的缓冲区才会获得内容信息，如果该参数为 NULL，那么 pOutBufLen 能获得实际所需要的缓冲区大小；参数 pOutBufLen 指向一个 ULONG 变量的指针，该变量指定固定信息结构的大小。如果此大小不足以容纳信息，GetNetworkParams 将使用所需大小填充此变量，并返回错误代码 ERROR_BUFFER_OVERFLOW。如果函数成功，返回值为 ERROR_SUCCESS；如果函数失败，返回值是错误代码。

【例 2.10】获取本机的域名

（1）新建一个对话框工程 Demo。

（2）切换到资源视图，在对话框上放一个按钮，然后添加事件代码：

```
void CDemoDlg::OnTest()
{
 //获得需要的缓冲区大小
 DWORD nLength = 0;
 if (GetNetworkParams(NULL, &nLength) != ERROR_BUFFER_OVERFLOW)
 {
     return;
 }

 FIXED_INFO* pFixedInfo = (FIXED_INFO*)new BYTE[nLength];

 //获得本地计算机网络参数
 if (GetNetworkParams(pFixedInfo, &nLength) != ERROR_SUCCESS)
 {
     delete[] pFixedInfo;
     return;
 }

 //获得本地计算机域名
 CString strText = _T("");
 strText.Format(_T("本地计算机的域名：\n%s"), pFixedInfo->DomainName);
 AfxMessageBox(strText);

 delete[] pFixedInfo;
}
```

（3）保存工程并运行，运行结果如图 2-14 所示。

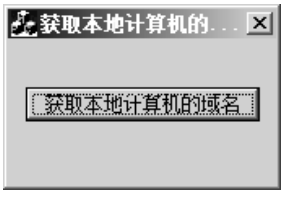


图 2-14

第 3 章

◂ 多线程编程 ▸

3.1 多线程编程的基本概念

3.1.1 为何要用多线程

前面的绝大多数程序都是单线程程序，如果程序中有多个任务，比如读写文件、更新用户界面、网络连接、打印文档等操作，比如按照先后次序，先完成前面的任务才能执行后面的任务。如果某个任务持续的时间较长，比如读写一个大文件，那么用户界面也无法及时更新，这样看起来程序像死掉一样，用户体验很不好。怎么解决这个问题呢？人们提出了多线程编程技术。在采用多线程编程技术的程序中，多个任务由不同的线程去执行，不同线程各自占用一段 CPU 时间，即使线程任务还没有完成，也会让出 CPU 时间给其他线程有机会去执行。这样在用户角度看起来，好像是几个任务同时进行的，至少界面上能得到及时更新了，大大改善了用户对软件的体验，提高了软件的友好度。

3.1.2 操作系统和多线程

要在应用程序中实现多线程，必须要有操作系统的支持。Windows 32 位或 64 位操作系统对应用程序提供了多线程的支持，所以 Windows NT/2000/XP/7/8/10 是一个多线程操作系统。根据进程与线程的支持情况，可以把操作系统大致分为如下几类：

（1）单进程、单线程，MS-DOS 大致是这种操作系统。

（2）多进程、单线程，多数 UNIX（及类 UNIX 的 Linux）是这种操作系统。

（3）多进程、多线程，Win32（Windows NT/2000/XP/7/8/10 等）、Solaris 2.x 和 OS/2 都是这种操作系统。

（4）单进程、多线程，VxWorks 是这种操作系统。

具体到 VC2017++开发环境，它提供了一套 Win32 API 函数来管理线程。用户既可以直接使用这些 Win32 API 函数，也可以通过 MFC 类的方式来使用，只不过 MFC 把这些 API 函数进行了简单的封装。

3.1.3　进程和线程

在了解线程之前，首先要理解进程的概念。简单地说，进程就是正在运行的程序。比如邮件程序正在接收电子邮件就是一个进程，杀毒软件正在杀毒就是一个进程，病毒软件正在传播病毒、破坏系统也是一个进程。程序是指计算机质量的静态集合，是一个静态的概念，而进程是一个动态的概念。Windows 操作系统中能同时运行多个进程，比如正在使用 Word 软件在打字的同时，又用语音聊天工具在聊着天，等等。每个进程都有自己的内存地址空间和 CPU 运行时间等一系列资源。进程有 3 种状态：

（1）运行态：正在 CPU 中运行。
（2）就绪态：运行准备就绪，但其他进程正在运行，所以只能等待。
（3）阻塞态：不能得到所需要的资源而不能运行。

现代操作系统大多支持多线程概念，每个进程中至少有一个线程，所以即使没有使用多线程编程技术，进程也含有一个主线程，所以也可以说，CPU 中执行的是线程，线程是程序的最小执行单位，是操作系统分配 CPU 时间的最小实体。一个进程的执行说到底就是从主线程开始的，如果需要，可以在程序任何地方开辟新的线程，其他线程都是由主线程创建的。一个进程正在运行，也可以说是一个进程中的某个线程正在运行。一个进程的所有线程共享该进程的公共资源，比如虚拟地址空间、全局变量等。每个线程也可以拥有自己私有的资源，如堆栈、在堆栈中定义的静态变量和动态变量、CPU 寄存器的状态等。

线程总是在某个进程环境中创建的，并且会在这个进程内部销毁，正所谓生于进程而挂于进程。线程和进程的关系是：线程是属于进程的，线程运行在进程空间内，同一进程所产生的线程共享同一内存空间，当进程退出时该进程所产生的线程都会被强制退出并清除。线程可与属于同一进程的其他线程共享进程所拥有的全部资源，但是其本身基本上不拥有系统资源，只拥有一点在运行中必不可少的信息（如程序计数器、一组寄存器和线程栈，线程栈用于维护线程在执行代码时需要的所有函数参数和局部变量）。

相对于进程来说，线程所占用资源更少，比如创建进程，系统要为它分配进程很大的私有空间，占用的资源较多，而对多线程程序来说，由于多个线程共享一个进程地址空间，因此占用资源较少。此外，进程间切换时，需要交换整个地址空间，而线程之间切换时只是切换线程的上下文环境，因此效率更高。在操作系统中引入线程带来的主要好处是：

（1）在进程内创建、终止线程比创建、终止进程要快。
（2）同一进程内的线程间切换比进程间的切换要快，尤其是用户级线程间的切换。
（3）每个进程具有独立的地址空间，而该进程内的所有线程共享该地址空间。因此，线程的出现可以解决父子进程模型中子进程必须复制父进程地址空间的问题。
（4）线程对解决客户/服务器模型非常有效。

虽然多线程给应用开发带来了不少好处，但是并不是所有情况下都要去使用多线程，要具体问题具体分析，通常在下列情况下可以考虑使用：

（1）应用程序中的各任务相对独立。
（2）某些任务耗时较多。
（3）各任务有不同的优先级。
（4）一些实时系统应用。

值得注意的是，一个进程中的所有线程共享它们父进程的变量，但同时每个线程可以拥有自己的变量。

3.1.4 线程调度

进程中有了多个线程后，就要管理这些线程如何去占用 CPU，这就是线程调度。线程调度通常由操作系统来安排，不同的操作系统其调度方法不同，比如有的操作系统采用轮询法来调度。Windows NT 以后的操作系统是一个优先级驱动、抢占式操作系统，也就是说线程具有优先级，具有高优先级的可运行的（就绪状态下的）线程总是先运行。如果出现一个更高优先级的线程就绪，正在运行的这个线程就可能在未完成其时间片前被抢占；甚至一个线程可能在未开始其时间片前就被抢占了，而要等待下一次被选择运行。

Windows 调度线程是在内核中进行的。当发生下面这些事件时将触发内核进行线程调度：

（1）线程的状态变成就绪状态，例如一个新创建的线程或者从等待状态释放出来的线程。
（2）线程的时间片结束而离开运行状态。它可能运行结束了，或者进入等待状态。
（3）线程的优先级改变了。
（4）出现了其他更高优先级的线程。

当 Windows 系统进行线程切换的时候，将执行一个上下文转换的操作，即保存正在运行的线程的相关状态，装载另一个线程的状态，开始新线程的执行。

每个线程都被赋予了一个优先级，优先级的取值范围从 0（最低）到 31（最高），并且规定只有 0 页线程（一个系统线程）可以拥有 0 优先级。

线程最初的优先级（值）也称为基础优先级（值），由两个因素决定：进程的优先级类别和线程所处的优先级层次。每个进程都属于某个优先级类别，进程的优先级类别可以分为以下几类（按照从低到高）：

（1）IDLE_PRIORITY_CLASS

该类别被称为空闲优先级类别，该类别的进程中的线程只在系统处于空闲的时候才运行，并且这些线程会被更高优先类别的进程中的线程抢占。屏幕保护程序就是拥有该类别优先级的典型例子。空闲优先级类别能被子进程继承，即拥有空闲优先级类别的进程所创建的子进程也具有空闲优先级类别。该类别定义如下：

```
#define IDLE_PRIORITY_CLASS             0x00000040
```

（2）BELOW_NORMAL_PRIORITY_CLASS

该类别比空闲优先级类别高，但比正常优先级类别低。Windows 2000 以下操作系统不支

持该级别。该类别定义如下：

```
#define BELOW_NORMAL_PRIORITY_CLASS      0x00004000
```

（3）NORMAL_PRIORITY_CLASS

该类别被称为正常优先级类别，是进程默认的优先级类别。该类别定义如下：

```
#define NORMAL_PRIORITY_CLASS           0x00000020
```

（4）ABOVE_NORMAL_PRIORITY_CLASS

该类别比正常优先级类别高，但低于高优先级类别。Windows 2000 以下操作系统不支持该级别。该类别定义如下：

```
#define ABOVE_NORMAL_PRIORITY_CLASS      0x00008000
```

（5）HIGH_PRIORITY_CLASS

该类别被称为高优先级类别。拥有该类别的进程通常要完成实时性的任务，即比如必须要立即执行的任务。该进程中的线程可以抢占正常优先级类别进程和空闲优先级类别进程中的线程。使用该优先级别应该特别慎重，因为一个拥有高优先级类别的进程几乎可以使用所有 CPU 能提供的运行时间，如果该优先级别的进程长时间运行，那么其他线程很可能一直得不到处理器时间。如果在同一时间设置了多个高优先级别的进程，那么它们的线程效率将降低。该类别定义如下：

```
#define      HIGH_PRIORITY_CLASS      0x00000080
```

（6）REALTIME_PRIORITY_CLASS

该类别被称为实时优先级类别，是最高的优先级类别。拥有该类别的进程中的线程能抢占其他所有进程中的线程，包括正在完成重要工作的操作系统进程。比如，该类别的进程在执行过程中可能会能让磁盘缓存不刷新或者鼠标出现停顿没反映。对于该优先级类别，或许应该永远不去使用，因为它会中断操作系统的工作，只有在直接和硬件打交道或完成的任务非常简短时才适合用该优先级类别。该类别定义如下：

```
#define REALTIME_PRIORITY_CLASS          0x00000100
```

上面这些宏都定义在 WinBase.h 中。在用函数 CreateProcess 创建进程的时候，可以指定其优先级类别。此外，还可以通过函数 GetPriorityClass 来获取某个进程的优先级类别，并能通过函数 SetPriorityClass 来改变某个进程的优先级类别。

在进程的每个优先级类别中，不同的线程属于不同优先级层次。从低到高有如下优先级层次：

```
#define THREAD_PRIORITY_IDLE           -15
#define THREAD_PRIORITY_LOWEST         -2
#define THREAD_PRIORITY_BELOW_NORMAL   -1
#define THREAD_PRIORITY_NORMAL         0
#define THREAD_PRIORITY_ABOVE_NORMAL    1
#define THREAD_PRIORITY_HIGHEST        2
#define THREAD_PRIORITY_TIME_CRITICAL   15
```

所有线程在创建（使用函数 CreateThread）的时候都属于 THREAD_PRIORITY_NORMAL 优先级层次，如果要修改优先级层次，可以在调用 CreateThread 时传入 CREATE_SUSPENDED 标志，让线程创建不马上执行。此时，我们再调用函数 SetThreadPriority 修改线程优先级层次，接着调用函数 ResumeThread 让线程变为可调度。通常，对于进程中用于接收用户输入的线程，建议使用 THREAD_PRIORITY_ABOVE_NORMAL 或者 THREAD_PRIORITY_HIGHEST 优先级层次，这样可以保证即时响应用户。对于那些后台工作的线程，尤其是密集使用处理器的线程，可以使用 THREAD_PRIORITY_BELOW_NORMAL 或者 THREAD_PRIORITY_LOWEST 优先级层次，这样可以确保必要的时候能被其他线程抢占，不至于它们老是占用处理器。如果低优先级层次的线程在等待高优先级层次的线程，为了让低优先级层次的线程能得到执行，可以在高优先级层次的线程中使用等待函数 Sleep 或 SleepEx，或者线程切换函数 SwitchToThread。

有了进程的优先级类别和线程的优先级层次，就可以确定一个线程的基础优先级了，具体数值见表 3-1。数值部分就是某个线程的基础优先级值。

表 3-1　线程基础优先级

	线程优先级层次					
进程优先级类别	idle	below normal	normal	above normal	high	real-time
time-critical	15	15	15	15	15	31
highest	6	8	10	12	15	26
above normal	5	7	9	11	14	25
normal	4	6	8	10	13	24
below normal	3	5	7	9	12	23
lowest	2	4	6	8	11	22
idle	1	1	1	1	1	16

表 3-1 中的数值是线程的基础优先级值，是线程开始时拥有的优先级。线程的优先级可以是动态变化的，后来系统可能升高或降低线程的优先级，以确保没有线程处于饥饿状态（好久没有运行）。对于基础优先级处于 16 到 31 之间的线程，系统不会再提高这些线程的优先级，只有基础优先级在 0 到 15 之间的线程才会被系统动态地提高优先级。

系统公平地对待同一优先级的所有线程。比如，对应最高优先级的所有线程，系统将以轮询的方式为这些线程分配时间片，如果这些线程一个都没有准备好运行，那么系统会对下一个最高优先级的所有线程采取轮询的方式分配时间片。如果后来更高优先级的线程运行准备就绪了，那么系统会停止运行低优先级的线程，即使该线程的时间片还没用完也会被停止运行，同时会为高优先级的线程分配完整的时间片。每个线程的优先级取决于两个因素：进程的优先级类别和线程的优先级层次。

线程调度程序不会考虑线程所属的进程，比如进程 A 有 8 个可运行的线程，进程 B 有 3 个可运行的线程，而且这 11 个线程的优先级别相同，那么每一个线程将会使用 1/11 的 CPU 时间，而不是将 CPU 的一半时间分配给进程 A，另一半时间分配给进程 B。

3.1.5　线程函数

线程函数就是线程创建后要执行的函数。执行线程，说到底就是执行线程函数。这个函数是我们自定义的，然后在创建线程的函数时把函数名作为参数传入线程创建函数。

同理，中断线程的执行就是中断线程函数的执行，以后再恢复线程的时候就会在前面线程函数暂停的地方开始继续执行下面的代码。结束线程也就不再运行线程函数了。线程的函数可以是一个全局函数或类的静态函数，通常这样声明：

```
DWORD WINAPI ThreadProc(  LPVOID  lpParameter);
```

其中，参数 lpParameter 指向要传给线程的数据，这个参数是在创建线程的时候作为参数传入线程创建函数中的。函数的返回值应该表示线程函数运行的结果：成功还是失败。注意，函数名 ThreadProc 可以是自定义的函数名，这个函数是用户先定义好再由系统来调用的。

线程函数必须返回一个值，这个返回值会成为该线程的退出代码。

3.1.6　线程对象和句柄

为了方便操作系统对线程进行管理，在创建线程时，系统会开辟一小块内存数据结构来存放线程统计信息，这块数据结构就是线程对象。由于它存在于内核中，因此线程对象是一个内核对象。线程内核对象不是线程本身，而是操作系统用来管理线程的一个小的数据结构。为了引用该对象，系统使用线程句柄来代表线程对象。句柄就是一个 32 位整数值，操作系统会通过这个句柄值来找到所需的内核对象。

内核对象是操作系统创建和管理的，比如创建线程的同时，系统在内核中就创建了一个线程对象。既然线程对象这个数据结构存在于内核中，那么应用程序就不能在内存中直接访问这个数据结构，也不能改变它们的内容，而只能通过 Win32 API 函数来操作，比如关闭线程对象可以用函数 CloseHandle。

另外需要注意的是，这里所说的对象的含义和 C++中面向对象的对象概念不同，这里的对象可以理解为操作系统在内核中的一块数据结构，存放一些管理和统计所需的信息。除了线程对象外，内核对象还包括进程对象、文件对象、事件对象、临界区对象、互斥对象和信号量对象等，也有句柄标识。

知道了线程对象的概念，我们就应该知道线程对象句柄的关闭函数 CloseHandle 并不能用来结束线程。

3.1.7　线程对象的安全属性

线程对象是一个内核对象，内核中的东西非常重要，系统通常会为内核对象指定一个安全属性。安全属性是在创建时指定的，主要描述这个对象的访问权限，比如谁可以访问该对象，谁不能访问该对象。系统会在线程对象创建的时候用一个结构体 SECURITY_ATTRIBUTES 来描述其安全性，通常会把这个结构体作为参数传入创建线程对象的函数（也就是创建线程的函数）中。该结构体定义如下：

```
typedef struct _SECURITY_ATTRIBUTES {  DWORD nLength;
 LPVOID  lpSecurityDescriptor;  BOOL  bInheritHandle;
} SECURITY_ATTRIBUTES, *PSECURITY_ATTRIBUTES;
```

其中，字段 nLength 表示该结构体的大小，单位是字节；lpSecurityDescriptor 指向线程对象的安全描述符，用来控制该线程对象是否能共享访问，如果该字段为 NULL，则内核对象被赋予一个默认的安全描述符；bInheritHandle 表示内核对象创建函数返回的句柄能否被新创建的进程所继承，如果该字段为 TRUE，则新的进程可以继承线程句柄。

3.1.8 线程标识

既然句柄是用来标识线程对象的，那么线程本身用什么来标识呢？在创建线程的时候，系统会给线程分配一个唯一的 ID 作为线程的标识，这个 ID 号从线程创建开始存在，一直伴随着线程的结束才消失。线程结束后该 ID 就会自动消失，我们不需要显式清除它。

通常线程创建成功后会返回一个线程 ID。

3.1.9 多线程编程的 3 种库

在 VC2017 开发环境中，通常有 3 种方式来开发多线程程序，分别是利用 Win32 API 函数来开发多线程程序、利用 CRT 库（C Runtime Library）函数来开发多线程程序和利用 MFC 库来开发多线程程序。这 3 种方式各有利弊，但有一点要注意，在 Win32 API 创建的线程（函数）中最好不要使用 CRT 库函数，因为这会引起少许的内存泄漏，原因是当 Win32 API 创建的线程在终止时不能正确地清理由 CRT 函数为静态数据和静态缓冲区分配的内存，对长时间运行的线程会引起不可预测的结果。CRT 库函数要用在 CRT 库函数创建的线程中。或许有人要说，要在 Win32 API 创建的线程中写控制台或开辟内存怎么办呢？答案是都用相应的 Win32 API 函数来代替，无论是读写控制台或者是内存管理，Win32 API 完全可以替代 CRT。这里讲的不要混用，是指不要在线程函数中混用，主线程中还是可以使用 CRT 函数的。

大家要知道，CRT 问世的时候，当时还没有多线程的概念，CRT 库函数都是针对单线程版本的。后来多线程出来了，微软和其他开发工具公司都针对 CRT 进行了多线程版本的改造。单线程版本的 CRT 在现在的 VC2017 中已经不用了。

这 3 种开发方式只是利用的库不同而已，但它们都可以用在不同类型的程序中，比如 MFC 程序或非 MFC 程序。

3.2 利用 Win32 API 函数进行多线程开发

在用 Win32 API 线程函数进行开发之前，我们首先要熟悉这些 API 函数。常见的与线程有关的 API 函数见表 3-2。

表 3-2　与线程有关的 API 函数

API 函数	含义
CreateThread	创建线程
CreateRemoteThread	在其他进程中创建线程
GetCurrentThreadId	得到当前线程的 ID
GetCurrentThread	得到当前线程的伪句柄
GetThreadId	得到某个指定线程的 ID
GetThreadPriority/ SetThreadPriority	得到/设置线程的优先级水平
GetThreadTimes	得到与线程相关的时间信息
OpenThread	得到某个存在的线程对象句柄
GetExitCodeThread	得到线程的退出码
SuspendThread/ ResumeThread	暂停/继续一个线程
Sleep/ SleepEx	暂停线程
TerminateThread	结束一个线程
ExitThread	正常结束一个线程

3.2.1　线程的创建

在 Win32 API 中，创建线程的函数是 CreateThread，该函数声明如下：

```
HANDLE CreateThread( LPSECURITY_ATTRIBUTES lpThreadAttributes,
  SIZE_T dwStackSize, LPTHREAD_START_ROUTINE lpStartAddress,
  LPVOID lpParameter, DWORD dwCreationFlags, LPDWORD lpThreadId);
```

其中，参数 lpThreadAttributes 是指向线程对象安全属性结构 SECURITY_ATTRIBUTES 的指针，该参数决定返回的句柄是否可以被子进程所继承，如果为 NULL 表示不能被继承；dwStackSize 表示线程堆栈的初始大小，如果为零采用默认的堆栈大小；lpStartAddress 指向线程函数的地址，线程函数就是线程创建后要执行的函数；lpParameter 指向传给线程函数的参数；dwCreationFlags 表示线程创建的方式，如果该参数为零，则线程创建后立即执行（就是立即执行线程函数），如果该参数为 CREATE_SUSPENDED，则线程创建后不会执行，一直要等到调用函数 ResumeThread 后才会执行；lpThreadId 指向一个 DWORD 变量，用来得到线程标识符（线程的 ID）。如果函数成功，返回线程的句柄（严格地讲，应该是线程对象的句柄），若函数失败则返回 NULL，可以用函数 GetLastError 来查看错误码。

CreateThread 创建完子线程后，主线程会继续执行 CreateThread 后面的代码，这就可能会出现创建的子线程还没执行完主线程就结束了，比如控制台程序，主线程结束就意味着进程结束了。在这种情况下，我们就需要让主线程等待，等待子线程全部运行结束后再继续执行主线程。还有一种情况，主线程为了统计各个子线程的工作的结果而需要等待子线程结束完毕后再继续执行，此时主线程就要等待了。VC2017 提供了等待函数来阻止某个线程的运行，直到某个指定的条件被满足，等待函数才会返回。如果条件没有满足，调用等待条件的函数将处于等待状态，并且不会占用 CPU 时间。

等待线程结束可以用等待函数 WaitForSingleObject 或 WaitForMultipleObjects。前者用于

等待一个线程对象的结束，后者用于等待多个线程对象的结束，但最多只能等待 64 个线程对象。这两个函数在线程同步中会详细解释。

线程创建之后，系统会为线程创建一个相关的内核对象——线程对象，该对象用线程句柄来引用，CreateThread 如果创建成功后会返回线程对象句柄（简称线程句柄），并且该线程对象的引用计数加 1。系统和用户可以利用线程句柄来对相应的线程进行必要的操纵，比如暂停、继续、等待完成等。如果我们不需要这些线程控制操作，则可以调用函数 CloseHandle 来关闭句柄，该函数声明如下：

```
BOOL  CloseHandle(HANDLE  hObject);
```

其中，参数 hObject 是传入的线程句柄。如果函数成功就返回非零，否则返回零。

CloseHandle 函数会使得线程对象的引用计数减 1，当变为 0 时，系统删除该内核对象。关闭线程句柄和线程退出并没有联系，所以可以在线程退出之前关闭，甚至刚刚创建成功的时候关闭句柄，比如可以这样写：

```
CloseHandle(CreateThread(…));
```

当然前提是不需要对线程进行控制的。如果不使用 CloseHandle 函数来关闭线程句柄，当整个应用程序结束时，系统也会对其进行回收，但这是一个不好的习惯。况且在很多情况下，我们在程序运行期间需要频繁地开启线程，如果不去关闭句柄就会导致系统资源越来越少，导致程序的不稳定。因此，每个线程句柄都应该要去关闭。

下面看一个例子，该例中会创建 500 个线程，每个线程函数中会向屏幕打印传入的线程参数。我们可以看到每个线程执行的时间不是固定的。

【例 3.1】在控制台程序中创建线程

（1）新建一个控制台工程。

（2）在 Test.cpp 输入如下代码：

```
#include "stdafx.h"
#include <windows.h>
#include <strsafe.h>

#define MAX_THREADS 500 //要创建的线程个数
#define BUF_SIZE 255

typedef struct _MyData {  //定义传给线程的参数的类型
 int val1;
 int val2;
} MYDATA, *PMYDATA;

DWORD WINAPI ThreadProc(LPVOID lpParam) //线程函数
{
 HANDLE hStdout;
 PMYDATA pData;

 TCHAR msgBuf[BUF_SIZE];
```

```
    size_t cchStringSize;
    DWORD dwChars;

    hStdout = GetStdHandle(STD_OUTPUT_HANDLE); //得到标准输出设备的句柄，为了打印
    if (hStdout == INVALID_HANDLE_VALUE)
        return 1;
    pData = (PMYDATA)lpParam; //把线程参数转为实际的数据类型
    //用线程安全函数来打印线程参数值
    StringCchPrintf(msgBuf, BUF_SIZE, _T("Parameters = %d, %d\n"), //构造字符串
        pData->val1, pData->val2);
    //得到字符串长度，存于 cchStringSize
    StringCchLength(msgBuf, BUF_SIZE, &cchStringSize);
    //在终端窗口输出字符串
    WriteConsole(hStdout, msgBuf, cchStringSize, &dwChars, NULL);
    HeapFree(GetProcessHeap(), 0, pData); //释放分配的空间

    return 0;
    }

    int _tmain(int argc, _TCHAR* argv[])
    {
    PMYDATA pData;
    DWORD dwThreadId[MAX_THREADS]; //线程 ID 数组
    HANDLE hThread[MAX_THREADS]; //线程句柄数组
    int i;
    printf("-----------begin----------------------\n");
    //创建 MAX_THREADS 个线程
    for (i = 0; i < MAX_THREADS; i++)
    {
        //为线程参数数据分配空间
        pData = (PMYDATA)HeapAlloc(GetProcessHeap(),HEAP_ZERO_MEMORY,sizeof(MYDATA));
        if (pData == NULL) //如果分配失败，则结束进程
            ExitProcess(2);

        //为每个线程产生唯一的数据
        pData->val1 = i;
        pData->val2 = i;
        //创建线程
        hThread[i] = CreateThread(NULL,0, ThreadProc,pData,0,&dwThreadId[i]);
        if (hThread[i] == NULL) //如果创建失败则结束进程
            ExitProcess(i);
    }//for

    for (i = 0; i < MAX_THREADS; i++)
    {
        WaitForSingleObject(hThread[j], INFINITE); //等待第 j 个线程结束
    CloseHandle(hThread[i]); //线程创建后关闭对应的线程对象句柄，以释放资源
    }
    printf("-----------end----------------------\n");
    return 0;
    }
```

在上述代码中，我们首先用 Win32 API 函数 HeapAlloc 为线程参数的数据开辟空间，该函数在指定的堆上开辟一块内存空间。函数 HeapAlloc 分配的内存要用函数 HeapFree 来释放。CRT 中的内存管理函数完全可以用 Win32 API 中的内存管理函数所代替。

在 for 循环里创建所有线程后，主线程会继续执行，由于我们在 for 后面调用了函数 WaitForSingleObject 来循环等待每一个线程的结束，因此主线程就一直在这里等待所有子线程运行结束，并且每当一个线程结束，就关闭其线程对象的句柄以释放资源。函数 WaitForSingleObject 用了参数 INFINITE，表示无限等待的意思，只要子线程不结束，调用（该函数的）线程将一直等待下去。

在线程函数 ThreadProc 中，只是把传入的线程参数的结构体字段打印到控制台上。函数 StringCchPrintf 是 sprintf 的替代者，StringCchLength 是 strlen 的替代者，CRT 中的函数完全可以用 Win32 API 中的字符串处理函数所代替。

Win32 API 函数 GetStdHandle 用来获取标准输出设备的句柄，最后由 WriteConsole 代替了 CRT 库中的 printf 函数，来打印输出到控制台窗口。这两个函数都是 Win32 中关于控制台编程的 API 函数。

再次强调，CreateThread 创建的线程函数中最好不要使用 CRT 库函数，我们完全可以用对应的 Win32 API 函数来替代 CRT 库函数，上面的代码证实了这一点。

函数 ExitProcess 用来结束一个进程及其所有线程，声明如下：

```
VOID ExitProcess(  UINT uExitCode);
```

其中，参数 uExitCode 是进程退出码，可以用 API 函数 GetExitCodeProcess 来获取它。

（3）保存工程并运行，运行结果如图 3-1 所示。

图 3-1

由于我们打印了 502 行数据，而控制台窗口默认显示的行没有这么多，因此导致开始很多行数据没有显示，可以在图 3-1 的控制台窗口标题栏上右击，然后选择“属性”命令，通过属性对话框（见图 3-2）来设置。在属性对话框中选择“布局”选项卡，然后在“屏幕缓冲区大小”下面的“高度”文本框中输入 600，那么控制台窗口最多就可以显示 600 行了，最后别忘了单击“确定”按钮，然后重新运行我们的程序。

图 3-2

3.2.2　线程的结束

线程的结束通常由以下原因所致：

（1）在线程函数中调用 ExitThread 函数。
（2）线程所属的进程结束了，比如进程调用了 TerminateProcess 或 ExitProcess。
（3）线程函数执行结束后（return）返回了。
（4）在线程外部用函数 TerminateThread 来结束线程。

第 1 种方式最好不用，因为线程函数如果有 C++对象，则 C++对象不会被销毁；第 2 和 4 种方式尽量避免使用，因为它们不让线程有机会做清理工作、不会通知与线程有关的 DLL、不会释放线程初始栈。第 3 种方式推荐使用，线程函数执行到 return 后结束，是最安全的方式，尽量应该将线程设计成这样的形式，即想让线程终止运行时，它们就能够 return（返回）。当用该方式结束线程的时候，会导致下面事件的发生：

（1）在线程函数中创建的所有 C++对象均将通过它们的撤销函数正确地撤销。
（2）操作系统将正确地释放线程堆栈使用的内存。
（3）与线程有关的 DLL 会得到通知，即 DLL 的入口函数（DllMain）会被调用。
（4）线程的结束状态从 STILL_ACTIVE 变为线程函数的返回值。
（5）由线程初始化的 I/O 等待都会取消。
（6）线程拥有的任何资源（比如窗口和钩子）都会得到释放。
（7）线程对象会被设为有信号状态，所以可以用函数 WaitForSingleObject 来等待线程的结束，比如：

```
WaitForSingleObject(hThread, INFINITE);
```

（8）如果当前线程是进程中的唯一主线程，则线程结束的同时所属的进程也结束。

另外，线程结束时并不意味着线程对象会自动释放，必须调用 CloseHandle 来释放线程对象。

结束线程的函数有两个：一个是在线程内部使用的函数 ExitThread，另外一个是在线程外部使用的函数 TerminateThread。函数 ExitThread 声明如下：

```
VOID ExitThread(DWORD dwExitCode);
```

其中，参数 dwExitCode 是传给线程的退出码，以后可以通过函数 GetExitCodeThread 来获取一个线程的退出码。该函数被调用的时候，线程堆栈会被释放。通常该函数在线程函数中调用。调用 ExitThread 函数来结束线程，通常会导致下列事件发生：

（1）如果线程函数中有 C++对象，则 C++对象得不到释放，因此有 C++对象的线程函数不调用 ExitThread。

（2）操作系统将正确地释放线程堆栈使用的内存。

（3）与线程有关的 DLL 会得到通知，即 DLL 的入口函数（DllMain）会被调用。

（4）线程的结束状态码为从 STILL_ACTIVE 变为 dwExitCode 参数确定的值。

（5）由线程初始化的 I/O 等待都会取消。

（6）线程拥有的任何资源（比如窗口和钩子）都会得到释放。

（7）线程对象会被设为有信号状态，所以可以用函数 WaitForSingleObject 来等待线程的结束，比如：

```
WaitForSingleObject(hThread, INFINITE);
```

（8）如果当前线程是进程中的唯一主线程，则线程结束的同时所属的进程也会结束。

由此可见，只有第一条和线程函数返回结束时的情况不同。

函数 GetExitCodeThread 用来获取线程的结束状态值。该函数声明如下：

```
BOOL GetExitCodeThread(HANDLE  hThread,  LPDWORD  lpExitCode);
```

其中，参数 hThread 是线程句柄；lpExitCode 是一个指针，指向用于存放获取到的线程结束状态的变量。如果函数成功就返回非零，否则返回零。如果线程还没有结束，则获取到的结束状态值为 STILL_ACTIVE，如果线程已经结束，则结束状态值可能是由函数 ExitThread 或 TerminateThread 的参数确定的值，或者是线程函数的返回值。

函数 TerminateThread 用来强制结束一个线程，这个函数尽量少用，因为它会导致一些线程资源没有机会释放。该函数声明如下：

```
BOOL  TerminateThread( HANDLE  hThread,  DWORD  dwExitCode);
```

其中，参数 hThread 是要关闭的线程的句柄；dwExitCode 为传给线程的退出码。如果函数成功就返回非零，否则返回零。该函数是一个危险的函数，非一些极端场合不要使用，比如线程中有网络阻塞函数 recv，此时结束线程通常没有更好的办法，只能使用 TerminateThread 了。

下面看几个线程结束有关的例子。

【例 3.2】得到线程的退出码

（1）新建一个控制台工程。

（2）在 Test.cpp 中输入如下代码：

```
#include "stdafx.h"
#include "windows.h"
#include <strsafe.h>
#define BUF_SIZE 255 //字符串缓冲区长度

DWORD WINAPI ThreadProc(LPVOID lpParameter)
{
 HANDLE hStdout;
 TCHAR msgBuf[BUF_SIZE]; //字符串缓冲区
 size_t cchStringSize; //存储字符串长度
 DWORD dwChars;

 hStdout = GetStdHandle(STD_OUTPUT_HANDLE);//得到标准输出设备的句柄，为了在终端打印
 if (hStdout == INVALID_HANDLE_VALUE)
     return 1;
 StringCchPrintf(msgBuf, BUF_SIZE, _T("线程 ID = %d\n"),
GetCurrentThreadId());//构造字符串
//得到字符串长度，存于 cchStringSize
 StringCchLength(msgBuf, BUF_SIZE, &cchStringSize);
//在终端窗口输出字符串
 WriteConsole(hStdout, msgBuf, cchStringSize, &dwChars, NULL);
//在终端窗口输出字符串
 WriteConsole(hStdout, _T("线程即将结束\n"), 7, &dwChars, NULL);
 ExitThread(5); //结束本线程
 WriteConsole(hStdout, _T("这句话不会有机会打印了\n"), 12, &dwChars, NULL);
 return 0;
}
int _tmain(int argc, _TCHAR* argv[])
{
 HANDLE h;
 DWORD dwCode,dwID;

 h = CreateThread(NULL, 0, ThreadProc, NULL, 0, &dwID); //创建子线程
 Sleep(1500); //主线程等待 1.5 秒
 GetExitCodeThread(h, &dwCode); //得到线程退出码
 printf("ID 为%d 的线程退出码：%d\n", dwID,dwCode); //输出结果
 CloseHandle(h); //关闭线程句柄
}
```

函数 GetCurrentThreadId 可以在线程函数中得到本线程的 ID，该值在 CreateThread 创建线程时确定，如果 CreateThread 函数最后一个参数为 NULL，子线程也会有 ID。

函数 ExitThread 设置了线程退出码为 5，因此 GetExitCodeThread 函数得到的子线程的退出码为 5。

（3）保存工程并运行，运行结果如图 3-3 所示。

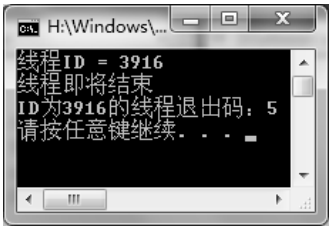


图 3-3

函数 TerminateThread 用来强制结束一个线程，声明如下：

```
BOOL TerminateThread( HANDLE hThread, DWORD dwExitCode);
```

其中，参数 hThread 是要结束的线程的句柄；dwExitCode 是传给线程的退出码，可以以后用函数 GetExitCodeThread 来获取该退出码。如果函数成功就返回非零，否则返回零。函数 TerminateThread 是具有危险性的函数，只应该在某些极端情况下使用。当 TerminateThread 结束线程时，线程将没有任何机会去执行用户模式下的代码以及释放初始栈。并且，依附在该线程上的 DLL 将不会被通知到该线程结束了。此外，如果要结束的目标线程拥有一个临界区，则临界区将不会被释放；如果要结束的目标线程从堆上分配了空间，则分配的堆空间将不会被释放。因此，这个函数尽量不去使用，比如下面的例子将产生死锁。

【例 3.3】TerminateThread 结束线程导致死锁

（1）新建一个控制台工程。

（2）在 Test.cpp 中输入如下代码：

```
#include "stdafx.h"
#include "windows.h"
DWORD WINAPI ThreadProc(LPVOID lpParameter)
{
 char* p;
 while (1) //循环的分配和释放空间
 {
     p = new char[5];
     delete []p;
 }
}

int _tmain(int argc, _TCHAR* argv[])
{
 HANDLE h;
 char* q;

 h = CreateThread(NULL, 0, ThreadProc, NULL, 0, NULL); //创建子线程
 Sleep(1500); //主线程等待 1.5 秒
 TerminateThread(h, 0); //结束子线程
 q = new char[2]; //主线程中分配空间，但程序停在此行，不再执行下去，因为死锁了
 printf("分配成功\n");
 delete[]q;
```

```
    CloseHandle(h); //关闭线程句柄

    return 0;
}
```

在上面的代码中，主线程执行到“q = new char[2];”时停滞不前了，因为发生了死锁。为什么会产生死锁呢？这是因为子线程中用了 new/delete 操作符向系统申请和释放堆空间，进程在其分配和回收内存空间时都会用到同一把锁。如果该线程在占用该锁时被杀死，即线程临死前还在进行 new 或 delete 操作，其他线程就无法再使用 new 或 delete 了，所以主线程中再用 new 时就无法成功执行了。

（3）保存工程并运行，运行结果如图 3-4 所示。

图 3-4

这个例子说明一旦函数 TerminateThread 结束线程，线程函数就将立即结束，非常暴力。那么上面的例子应该如何让线程优雅地退出呢？简单的方法是用一个全局变量和 WaitForSingleObject 函数。

【例 3.4】控制台下结束线程

（1）新建一个控制台工程。

（2）在 Test.cpp 中输入如下代码：

```
#include "stdafx.h"
#include "windows.h"

BOOL gbExit=TRUE; //控制子线程中的循环是否结束

DWORD WINAPI ThreadProc(LPVOID lpParameter)
{
 char* p;
 while (gbExit)
 {
     p = new char[5];
     delete[]p;
 }
 return 0;
}
```

```
int _tmain(int argc, _TCHAR* argv[])
{
 HANDLE h;
 char* q;

 h = CreateThread(NULL, 0, ThreadProc, NULL, 0, NULL); //创建线程
 Sleep(1500); //主线程休眠一段时间，让出 CPU 给子线程运行一段时间

 gbExit = FALSE; //设置标记，让子线程中的循环结束，以结束子线程
 WaitForSingleObject(h, INFINITE); //等待子线程的退出

 h = NULL;
 q = new char[2]; //主线程中分配空间
 printf("分配成功\n");
 delete[]q; //释放空间
 CloseHandle(h); //关闭子线程句柄

 return 0;
}
```

由于子线程结束的时候系统会向线程句柄发送信号，因此可以使用等待函数 WaitForSingleObject 来等待线程句柄的信号，一旦有信号了，就说明子线程结束，主线程就可以继续执行下去了。由于子线程是线程函数正常返回后退出的，因此 new/delete 的锁不再被占用，主线程就可以正常使用 new/delete 了。

（3）保存工程并运行，运行结果如图 3-5 所示。

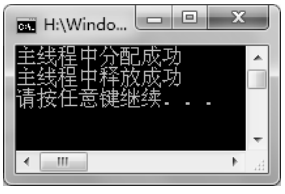


图 3-5

【例 3.5】图形界面下结束线程

（1）新建一个对话框工程。

（2）切换到资源视图，打开对话框设计器，然后删除上面所有的控件，并添加两个按钮，标题分别设为“开启线程”和“结束线程”。接着为“开启线程”按钮添加事件处理函数，代码如下：

```
void CTestDlg::OnBnClickedButton1()
{
 // TODO:  在此添加控件通知处理程序代码
 CClientDC dc(this);
 dc.TextOut(0, 0, _T("线程已启动")); //在对话框上显示线程已启动
GetDlgItem(IDC_BUTTON1)->EnableWindow(0); //设置“开启线程”按钮不可用
```

```
gbExit = TRUE;
 ghThread = CreateThread(NULL, 0, ThreadProc, m_hWnd, 0, NULL); //创建线程
}
```

ghThread 是一个全局变量，保存线程句柄，定义如下：

```
HANDLE ghThread;
```

然后添加线程函数：

```
DWORD WINAPI ThreadProc(LPVOID lpParameter)
{
 while (gbExit)
     ;
 return 0;
}
```

代码很简单，gbExit 是控制循环结束的全局变量，定义如下：

```
BOOL gbExit = TRUE;
```

为“结束线程”添加事件处理函数，代码如下：

```
void CTestDlg::OnBnClickedButton2()
{
 // TODO:  在此添加控件通知处理程序代码
 if (!gbExit)
     return; //如果已经结束则直接返回
 gbExit = FALSE; //设置循环结束变量
 WaitForSingleObject(ghThread, INFINITE); //等待子线程退出
CloseHandle (ghThread);//关闭线程句柄
GetDlgItem(IDC_BUTTON1)->EnableWindow();//设置“开启线程”按钮可用
 CClientDC dc(this);
 dc.TextOut(0, 0, _T("线程已结束")); //在对话框上显示已经结束
}
```

这种方式结束线程是优雅的，尽量不要使用 TerminateThread 函数来结束线程。

（3）保存工程并运行，运行结果如图 3-6 所示。

图 3-6

3.2.3　线程和 MFC 控件交互

在 MFC 程序中，经常有这样的需求，需要把在线程计算的结果显示在某个 MFC 控件上，或者后台线程的工作时间较长，需要在界面上反映出它的进度。

那线程如何和界面打交道呢？或许有人会想到启动线程时把 MFC 控件对象的指针传参给

线程函数，然后直接在线程函数中使用 MFC 控件对象并调用其方法来显示，但这种方式是不规范的，有可能会出现问题，因为 MFC 控件对象不是线程安全的，不能跨线程使用，或许此种方式在小程序中不会出问题，但是不出问题不等于没有问题，放在大型程序中早晚要出问题。再次强调：不要在子线程中操作主线程中创建的 MFC 控件对象，否则会带来意想不到的问题。在主线程中界面控件应该由主线程来控制，如果在子线程中也操作了界面控件，就会导致两个线程同时操作一个控件，若两个线程没有进行同步，就可能会发生错误。

那么在 MFC 程序中用 Win32 API 进行多线程开发时，应该如何和界面打交道呢？不同的情况有不同的处理方式。

如果仅仅是把线程计算的结果显示一下，第一种方法是把控件句柄传给线程，然后在线程函数中调用 Win32 API 函数或发送控件消息来操作控件；第二种方法是把界面主窗口的句柄传给线程，然后在线程函数中向主窗口发送自定义消息，接着可以在主窗口的自定义消息处理函数中调用控件对象的方法来操作控件。总之，如果涉及界面操作，应该传主窗口或控件窗口的句柄给子线程，而不要传主窗口或控件的对象指针，比如主窗口的 this。另外，窗口句柄传给线程后，不要试图去通过句柄来获得窗口对象指针，比如想通过 FromHandle 函数把 HWND 转为（对话框）窗口对象指针：

```
   CMyDialog *pDlg = static_cast<CMyDialog*>(CWnd::FromHandle(reinterpret_cast<
HWND> ( pData ) ) );
```

这种情况系统会分配一个临时的窗口对象给你，而不是真正的主窗口对象。原因是强调 HWND 和 CWnd 的映射关系只能在一个线程模块（THREAD_MODULE_STATE）中使用，即不能跨线程同时也不能跨模块转换两者。

如果后台工作比较耗时，用户希望它尽快完成工作并且想知道其处理进度，则不能在线程函数中发送消息来更新界面，因为这样会拖慢子线程的工作速度。此时，应该设置一个进度变量（比如一个全局变量），放在子线程中不断累加，而在主线程中采用每隔一段时间去获取该变量值，并转换成百分比，然后把百分比以字符串或进度条的形式显示在界面上。这种方式相当于主线程主动轮询的方式，但界面操作依然是在主线程中完成。如果要增强同步程度，可以把间隔时间设置短一点，但代价也是降低工作效率。或许有人想完全和计算进度同步，想在线程函数中每计算一步就发送一个界面更新消息去反映一次进度，但这样会拖慢线程工作的计算效率，如果你的线程计算需要追求速度。这是因为 SendMessage 是一个阻塞函数，必须要等界面更新完毕后才能返回，在这个过程中线程就阻塞在那里了。有人或许又想到了非阻塞发送消息函数 PostMessage，这个将直接导致界面死掉。比如：

```
   DWORD WINAPI ThreadProc(LPVOID lpParameter)
   {
    HWND hPos = (HWND)lpParameter; //获取进度条控件的句柄
    i = 0;
    for (i = 0; i < 88;i++) //循环做 88 次计算工作
    {
        myComputeWork();//计算工作
        ::PostMessage(hPos,PBM_SETPOS, i, 0); //发送设置进度的消息
        //Sleep(1);
```

```
    }

    return 0;
}
```

在上面的线程函数中，不停地循环做计算工作 myComputeWork，并且每计算完一次，就向进度条控件发送一次进度前进的消息。由于 PostMessage 是非阻塞函数，它向消息队列扔一条消息后就会立即返回，而界面操作通常比较慢，因此线程函数的循环向消息队列扔 PBM_SETPOS 消息会非常快，导致线程结束前消息队列中其他界面消息（比如鼠标点击、菜单操作等）无法进入消息队列，就不能接收用户操作了，看起来就像卡死了。如果我们让线程函数慢点扔消息呢？比如在 PostMessage 后面加一个 Sleep 函数，这样界面虽然不会卡死了，但是，通常这种循环计算工作用户对速度都是有要求的，人为的减慢将是不可接受的。虽然每隔一段时间去轮询进度的方法不能完全同步线程计算工作，但通常用户不会对此有严格的要求，只要有一个大概的反映进度就可以了。

下面我们看几个例子来加深一下理解。第一个例子通过在线程中把计算结果向控件发送控件消息来显示。第二个例子向主窗口发送自定义消息，然后在自定义消息处理函数中调用控件对象的方法来显示结果。两个例子传给线程函数的都是窗口句柄。相比较而言，第二种方法更简单些，因为控件消息大家使用起来不习惯，尤其对于 SDK 编程不熟悉的人来讲，更喜欢用 MFC 的方式来操作控件。

【例 3.6】发送控件消息在状态栏中显示线程计算的结果

（1）新建一个单文档工程。

（2）切换到资源视图，打开菜单设计器，然后在“视图”菜单下添加菜单项“开始计算”，ID 为 ID_WORK。当用户点击该菜单项的时候，将开启一个线程，线程中将进行一个计算工作，然后把计算结果发送控件消息显示到状态栏上去。

为“开始计算”菜单项添加 CMainFrame 类下的事件处理函数：

```
void CMainFrame::OnWork()
{
 // TODO:  在此添加命令处理程序代码
 CreateThread(NULL, 0, ThreadProc, m_wndStatusBar.m_hWnd, 0, NULL); //创建线程
}
```

代码很简单，使用 API 函数 CreateThread 来创建一个线程：线程函数是 ThreadProc，线程参数是状态栏的句柄 m_wndStatusBar.m_hWnd。由于 LPVOID 类型占用 4 个字节，而 m_hWnd 也占用 4 个字节，所以可以把句柄直接给 LPVOID。

（3）在 MainFrame.cpp 中添加一个全局的线程函数，代码如下：

```
DWORD WINAPI ThreadProc(LPVOID lpParameter)
{
 HWND hwnd = (HWND)lpParameter; //把参数转为句柄
 int nCount = 4; //定义状态栏的 4 个部分
 //定义状态栏每个部分的大小，每个元素是每部分右边的纵坐标
```

```
    int array[] = { 100, 200, 300, -1 };
    //向状态栏发送分割部分消息，把状态栏分为 4 个部分，array 存放每部分右边的纵坐标
    ::SendMessage(hwnd, SB_SETPARTS, (WPARAM)nCount, (LPARAM)array);
    //把计算结果发送给控件
    ::SendMessage(hwnd, SB_SETTEXT, (LPARAM)0, (WPARAM)TEXT("1+1=2"));
    ::SendMessage(hwnd, SB_SETTEXT, (LPARAM)1, (WPARAM)TEXT("2+2=4"));
    ::SendMessage(hwnd, SB_SETTEXT, (LPARAM)2, (WPARAM)TEXT("3+3=4"));
    ::SendMessage(hwnd, SB_SETTEXT, (LPARAM)3, (WPARAM)TEXT("4+4=8"));
    return 0;
    }
```

我们把状态栏分为 4 个部分，每个部分显示一个计算结果，当然这里也没有什么计算过程，直接把计算结果发送出去了。数组 array 存放每部分右边的纵坐标，这个坐标是客户区坐标，都是相对于客户区左边的，最后一个元素“-1”表示状态栏剩下部分的纵坐标一直持续到状态栏右边结束。消息 SB_SETPARTS 是状态栏进行分割的消息，SB_SETTEXT 是为状态栏某个部分设置文本的消息。

（4）定位到函数 CMainFrame::OnCreate，把该函数中的一个语句注释掉：

```
    //m_wndStatusBar.SetIndicators(indicators, sizeof(indicators)/sizeof(UINT));
```

这条语句是框架用来切分状态栏部分的。为了防止和我们分割状态栏发生冲突，所以注释掉，否则每次最大化窗口的时候我们的分割就会失效。

（5）保存工程并运行，运行结果如图 3-7 所示。

图 3-7

【例 3.7】发送自定义消息在状态栏中显示线程计算的结果

（1）新建一个单文档工程。

（2）切换到资源视图，打开菜单设计器，然后在“视图”菜单下添加菜单项“开始计算”，ID 为 ID_WORK。当用户点击该菜单项的时候，将开启一个线程，并把主框架窗口的句柄传给线程函数，线程函数中将把计算结果向主框架窗口发送自定义消息，然后在自定义消息处理函数中调用状态栏对象的方法来显示结果。

为“开始计算”菜单项添加 CMainFrame 类下的事件处理函数：

```
    void CMainFrame::OnWork()
    {
    // TODO:  在此添加命令处理程序代码
    CreateThread(NULL, 0, ThreadProc, m_hWnd, 0, NULL); //创建线程
    }
```

代码很简单，就用 API 函数 CreateThread 来创建一个线程，线程函数是 ThreadProc，线程参数是主框架窗口的句柄 m_hWnd。由于 LPVOID 类型占用 4 个字节，而 m_hWnd 也占用 4 个字节，因此可以把句柄直接给 LPVOID。接着添加线程函数：

```
DWORD WINAPI ThreadProc(LPVOID lpParameter)
{
 HWND hwnd = (HWND)lpParameter; //把参数转为句柄
 CString strRes = _T("结果是 100b/s");
 ::SendMessage(hwnd, MYMSG_SHOWRES, WPARAM(&strRes), NULL);
 return 0;
}
```

我们把 CString 对象的地址作为消息参数传给消息处理函数。MYMSG_SHOWRES 是在 MainFrame.cpp 开头定义的自定义消息，定义如下：

```
#define MYMSG_SHOWRES  WM_USER +10
```

然后在消息映射表中添加消息映射：

```
ON_MESSAGE(MYMSG_SHOWRES, OnShowRes)
```

OnShowRes 是自定义消息 MYMSG_SHOWRES 的处理函数，定义如下：

```
LRESULT CMainFrame::OnShowRes(WPARAM wParam, LPARAM lParam)
{
 CString* pstr = (CString*)wParam;

 m_wndStatusBar.SetPaneInfo(1, 10001, SBPS_NORMAL, 300);
 m_wndStatusBar.SetPaneText(1, *pstr);

 return 0;
}
```

该函数把接收到的字符串显示在状态栏第一个窗格上：SetPaneInfo 用来设置状态栏第一个窗格的宽度为 300，函数 SetPaneText 用来把收到的字符串显示在第一个窗格上。注意：窗格次序从 0 开始，最左边的是第 0 个窗格。

最后，在 MainFrame.h 中对该函数进行声明：

```
afx_msg LRESULT OnShowRes(WPARAM wParam, LPARAM lParam);
```

该声明写在 DECLARE_MESSAGE_MAP()前面，并且因为是消息处理函数，所以开头要加上 afx_msg。

（3）保存工程并运行，运行结果如图 3-8 所示。

图 3-8

【例 3.8】主动轮询并显示线程工作的进度

（1）新建一个对话框工程。

（2）切换到资源视图，打开对话框编辑器，删除上面的所有控件，然后添加一个按钮和进度条控件，按钮标题是“开启线程”，并为两个控件添加控件变量，分别为 m_btn 和 m_pos。为按钮添加事件处理函数：

```
void CTestDlg::OnBnClickedButton1()
{
 // TODO:  在此添加控件通知处理程序代码
 gjd = 0; //初始化线程工作进度变量
 m_btn.EnableWindow(0); //开启线程时按钮变灰
 m_pos.SetRange(0, 100); //设置进度条范围
 m_pos.SetPos(0); //设置进度条起点位置
 SetTimer(1, 50, NULL); //开启计时器，每隔 50 毫秒轮询一次进度
 //开启线程并关闭句柄
 CloseHandle(CreateThread(NULL, 0, ThreadProc, m_hWnd, NULL, NULL));
}
```

其中，gjd 是整型全局变量，用来记录线程的计算工作进度。由于我们不需要对线程进行控制，因此线程句柄可以开始就关闭了。

添加线程函数：

```
DWORD WINAPI  ThreadProc(LPVOID lpParameter)
{
 int  i=0;
 float res=0.01;
 CString strRes;
 HWND hwnd = (HWND)lpParameter; //把参数转为句柄
 for (i = 0; i < 88;i++)
 {
     res += myComputeWork();//计算工作
     gjd++;
 }

 //发送计算结果自定义消息更新界面
 strRes.Format(_T("计算结果%.2lf"), res);
 ::SendMessage(hwnd, MYMSG_SHOWRES, WPARAM(&strRes), NULL);

 return 0;
}
```

代码很简单，循环 88 次做我们的计算工作，然后把计算结果组织成字符串并通过自定义消息发送出去，以此显示在界面上。myComputeWork 是一个自定义的全局函数，定义如下：

```
float myComputeWork()
{
 int i=0,j;
 double d = 1.0;
 while (i < 2000)
```

```
{
    i++;
    for (j = -600; j < 600; j++)
            d += sin(0.01);
}
return d;
}
```

因为使用了正弦函数 sin，所以文件开头不要忘了包含 math.h。

接着，添加自定义消息 MYMSG_SHOWRES 的定义、消息映射以及消息处理函数：

```
LRESULT CTestDlg::OnShowRes(WPARAM wParam, LPARAM lParam)
{
CString* pstr = (CString*)wParam;
KillTimer(1); //停止计时器
m_pos.SetPos(100); //设置进度条最右边
m_btn.EnableWindow(1); //让“开启按钮”使能
CClientDC dc(this);
dc.TextOut(0, 0, *pstr); //在对话框左上角显示结果字符串

return 0;
}
```

（3）为对话框添加计时器消息处理函数：

```
void CTestDlg::OnTimer(UINT_PTR nIDEvent)
{
// TODO:  在此添加消息处理程序代码和/或调用默认值
float f = gjd / 87.0;
int per = f * 100;
m_pos.SetPos(per);

CDialogEx::OnTimer(nIDEvent);
}
```

这是主动轮询的核心所在，我们用一个计时器每隔一段时间来获取进度变量的值，并换算成百分百，然后显示在进度条上。这样看起来进度条就和线程计算工作在几乎同时前进了。

（4）保存工程并运行，运行结果如图 3-9 所示。

图 3-9

3.2.4 线程的暂停和恢复

在上面的程序中，线程句柄似乎没啥用，这小节讲述线程的暂停和恢复继续运行，线程句柄就很重要了。暂停线程执行的 API 函数是 SuspendThread，声明如下：

```
DWORD SuspendThread( HANDLE hThread);
```

其中，参数 hThread 是要暂停的线程句柄，该句柄必须要有 THREAD_SUSPEND_RESUME 访问权限。如果函数成功就返回以前暂停的次数，否则返回-1，此时可以用 GetLastError 来获得错误码。当函数成功的时候，线程将暂停执行，并且线程的暂停次数递增一次。每个线程都有一个暂停计数器，最大值为 MAXIMUM_SUSPEND_COUNT，如果暂停计数器大于零，线程则暂停执行。另外，这个函数一般不用于线程同步，如果对一个拥有同步对象（比如信号量或临界区）的线程调用 SuspendThread 函数，则有可能会引起死锁，尤其当被暂停的线程想要获取同步对象的时候。

恢复线程执行的函数是 ResumeThread，但不是说调用该函数线程就会恢复执行，该函数主要是减少暂停计数器的次数。线程的暂停计数器如果恢复到零，线程才会恢复执行。该函数声明如下：

```
DWORD ResumeThread( HANDLE hThread);
```

其中，参数 hThread 是要减少暂停次数的线程句柄，该句柄必须要有 THREAD_SUSPEND_RESUME 访问权限。如果函数成功就返回以前的暂停次数，若返回值大于 1，则表示线程依旧处于暂停状态，如果函数失败就返回-1，此时可以用 GetLastError 来获得错误码。函数 ResumeThread 会检查线程的暂停计数器，如果 ResumeThread 返回值为零，就说明线程当前没有暂停；如果 ResumeThread 返回值大于 1，则暂停计数器减 1，且线程依旧处于暂停状态中；如果 ResumeThread 返回值为 1，则暂停计数器减 1，并且原来暂停的线程将恢复执行。

下面我们来看一个图形界面的例子，演示这几个函数的使用。

【例 3.9】线程的暂停、恢复和中途终止

（1）新建一个对话框工程。

（2）切换到资源视图，打开对话框编辑器，删除上面的所有控件，然后添加 4 个按钮和进度条控件，4 个按钮标题是“开启线程”“暂停线程”“恢复线程”和“结束线程”，并为“开启线程”按钮和进度条控件添加控件变量（分别为 m_btn 和 m_pos）。为“开启线程”按钮添加事件处理函数：

```
void CTestDlg::OnBnClickedButton1()
{
 // TODO:  在此添加控件通知处理程序代码
 gjd = 0; //初始化线程工作进度变量
gbExit = FALSE; //结束线程函数中循环的全局变量
 m_btn.EnableWindow(0); //开启线程时按钮变灰
 m_pos.SetRange(0, 100); //设置进度条范围
 m_pos.SetPos(0); //设置进度条起点位置
```

```
SetTimer(1, 50, NULL); //开启计时器，每隔 50 毫秒轮询一次进度
//开启线程并关闭句柄
ghThread = CreateThread(NULL, 0, ThreadProc, m_hWnd, NULL, NULL);
}
```

其中，gjd 是整型全局变量，用来记录线程的计算工作进度。由于我们不需要对线程进行控制，因此线程句柄可以开始就关闭了。gbExit 是一个 BOOL 型的全局变量，用来控制线程函数中循环的结束。ghThread 是一个全局变量，用来存放线程句柄。

添加线程函数：

```
DWORD WINAPI  ThreadProc(LPVOID lpParameter)
{
 int  i=0;
 float res=0.01;
 CString strRes;
 HWND hwnd = (HWND)lpParameter; //把参数转为句柄
 for (i = 0; i < 88;i++)
 {
     if (gbExit) //控制循环退出
          break;
     res += myComputeWork();//计算工作
     gjd++; // myComputeWork 每执行一次，该进度变量就累加一次
 }

 if (gbExit) strRes.Format(_T("线程被中途结束掉了"), res);
 else strRes.Format(_T("计算结果%.2lf"), res);
 ::SendMessage(hwnd, MYMSG_SHOWRES, WPARAM(&strRes), NULL); //发送自定义消息

 return 0;
}
```

代码很简单，循环 88 次做我们的计算工作，然后把计算结果组织成字符串并通过自定义消息发送出去，以此显示在界面上。gbExit 是用来控制循环退出条件的，当用户点击“结束进程”的时候会置该变量为 TRUE。myComputeWork 是一个自定义的全局函数，模拟一个计算工作，定义如下：

```
float myComputeWork()
{
 int i=0,j;
 double d = 1.0;
 while (i < 2000)
 {
     i++;
     for (j = -600; j < 600; j++)
             d += sin(0.01);
 }
 return d;
}
```

因为使用了正弦函数 sin，所以文件开头不要忘了包含 math.h。

接着，添加自定义消息 MYMSG_SHOWRES 的定义、消息映射以及消息处理函数：

```
LRESULT CTestDlg::OnShowRes(WPARAM wParam, LPARAM lParam)
{
 CString* pstr = (CString*)wParam;
 KillTimer(1); //停止计时器
CloseHandle(ghThread); //关闭进程句柄
 ghThread = NULL;
 m_pos.SetPos(100); //设置进度条最右边
 m_btn.EnableWindow(1); //让“开启按钮”使能
 CClientDC dc(this);
 dc.TextOut(0, 0, *pstr); //在对话框左上角显示结果字符串

 return 0;
}
```

（3）为对话框添加计时器消息处理函数：

```
void CTestDlg::OnTimer(UINT_PTR nIDEvent)
{
 // TODO:  在此添加消息处理程序代码和/或调用默认值
 float f = gjd / 87.0;
 int per = f * 100;
 m_pos.SetPos(per);

 CDialogEx::OnTimer(nIDEvent);
}
```

这是主动轮询的核心所在，我们用一个计时器每隔一段时间来获取进度变量的值，并换算成百分百，然后显示在进度条上。这样看起来进度条就和线程计算工作在几乎同时前进了。

（4）添加“暂停线程”按钮的事件处理函数，代码如下：

```
void CTestDlg::OnBnClickedButton2()
{
 // TODO:  在此添加控件通知处理程序代码
 if (ghThread)
     SuspendThread(ghThread);
}
```

再添加“恢复线程”按钮的事件处理函数，代码如下：

```
void CTestDlg::OnBnClickedButton3()
{
 // TODO:  在此添加控件通知处理程序代码
 if (ghThread)
     ResumeThread(ghThread);
}
```

再添加“结束线程”按钮的事件处理函数，代码如下：

```
void CTestDlg::OnBnClickedButton4()
{
```

```
    // TODO:  在此添加控件通知处理程序代码
    gbExit = TRUE; //通过全局变量来停止线程函数中的循环以此来结束线程
    }
```

（5）保存工程并运行，运行结果如图 3-10 所示。

图 3-10

3.2.5　消息线程和窗口线程

前面所创建的线程没有消息循环，也没有在线程中创建窗口，通常把这种线程称为工作线程。其实函数 CreateThread 创建线程还可以拥有消息队列，甚至创建窗口。拥有消息队列的线程称为消息线程。消息线程有两种类型：创建了窗口的消息线程和没有创建窗口的消息线程，前者通常称为窗口线程（或 UI 线程）。窗口线程中既然创建了窗口，那也必须要有窗口过程函数，由窗口过程函数对窗口消息进行处理，并且窗口和消息循环要在一个线程中，因此大家不要跨线程处理 MFC 控件对象，每个控件都是一个窗口，都有各自的消息循环，只是支持 MFC 把它封装掉罢了。

要让一个线程成为消息线程，方法是在线程函数中创建消息循环，并在循环中调用 API 函数的 GetMessage 或 PeekMessage。一旦在线程中调用了这两个函数，系统就会为线程创建一个消息队列，这样这两个函数就可以获取消息了。大家一定要明确：消息队列是系统创建的，消息循环是线程创建的。

函数 GetMessage 声明如下：

```
   BOOL GetMessage(LPMSG  lpMsg, HWND  hWnd,UINT  wMsgFilterMin, UINT
wMsgFilterMax);
```

其中，参数 lpMsg 指向 MSG 结构，该结构存放从线程消息队列中获取到的消息；hWnd 为收到的窗口消息所对应窗口的句柄，这个窗口必须属于当前线程，如果该参数为 NULL，则函数将收到所属当前线程的任一窗口的窗口消息以及当前线程消息队列中窗口句柄为 NULL 的消息，因此如果该参数为 NULL，则不管线程消息是不是窗口消息都将被收到，如果该参数为-1，则只会收到窗口句柄为 NULL 的消息；wMsgFilterMin 指定所收到的消息值的最小值；wMsgFilterMax 指定所收到的消息值的最大值，如果 wMsgFilterMin 和 wMsgFilterMax 为零，那么 GetMessage 将收到所有可得到的消息。如果函数收到的消息不是 WM_QUIT,，就返回非零，否则返回零。要注意的是，如果 GetMessage 从消息队列中取不到消息，就不会返回而阻塞在那里，一直等到取到消息才返回。因此，当线程消息队列中没有消息时 GetMessage 使得

线程进入 IDLE 状态，被挂起；当有消息到达线程时 GetMessage 被唤醒，获取消息并返回。另外，该函数获取消息之后将删除消息队列中除 WM_PAINT 消息之外的其他消息，而 WM_PAINT 则只有在其处理之后才被删除。GetMessage 函数只有在接收到 WM_QUIT 消息时才返回 0，此时消息循环退出。

函数 PeekMessage 的主要功能是查看消息队列中是否有消息，当然也可以取出消息。即使消息队列中没有消息，该函数也会立即返回。相对而言，实际开发中 GetMessage 用的多一点。

在没有窗口的消息线程中，消息循环通常这样写：

```
while (GetMessage(&msg, NULL, NULL, NULL))
{
    switch (msg.message)
    {
    case MYMSG1: //自定义的消息
        break;
    case MYMSG2: //自定义的消息
        break;
    }
}
```

对于有窗口的消息线程，消息循环通常这样写：

```
while (GetMessage(&msg, NULL, NULL, NULL))
{
    TranslateMessage(&msg);//如果要字符消息，这句也要
    DispatchMessage(&msg); //把消息派送到窗口过程中去
}
```

函数 TranslateMessage 将虚拟键消息转换为字符消息。函数 DispatchMessage 必须要有，它把收到的窗口消息回传给操作系统，由操作系统调用窗口过程函数对消息进行处理。

向线程发送消息可以使用函数 SendMessage、PostMessage 或 PostThreadMessage。SendMessage 和 PostMessage 根据窗口句柄来发送消息，所以如果要向某个线程中的窗口发送消息，可以使用 SendMessage 或 PostMessage。需要注意的是，SendMessage 要一直等到消息处理完才返回，所以如果它发送的消息不是本线程创建的窗口的窗口消息，则本线程会被阻塞；PostMessage 则不会，它会立即返回。另外，如果 PostMessage 的句柄参数为 NULL，则相当于向本线程发送一个非窗口的消息。

函数 PostThreadMessage 根据线程 ID 来向某个线程发送消息，声明如下：

```
BOOL PostThreadMessage(DWORD idThread, UINT  Msg, WPARAM  wParam,  LPARAM
lParam);
```

其中，参数 idThread 为线程 ID，函数就是向该 ID 的线程投递消息；参数 Msg 表示要投递消息的消息号；wParam 和 lParam 为消息参数，可以附带一些信息。如果函数成功就返回非零，否则返回零。需要注意的是，目标线程必须要有一个消息循环，否则 PostThreadMessage 将失败。此外，PostThreadMessage 发送的消息不需要关联一个窗口，这样目标线程就不需要为了接收消息而创建一个窗口了。

通常，PostThreadMessage 用于消息线程。SendMessage 或 PostMessage 用于窗口线程。

下面我们看几个例子来加深一下对这几个函数使用的理解。

【例 3.10】PostThreadMessage 发送消息给无窗口的消息线程

（1）新建一个对话框工程。

（2）切换到资源视图，打开对话框编辑器，删除上面所有的控件，然后添加 3 个按钮，标题分别是“创建线程”“发送线程消息 1”和“发送线程消息 2”。为“创建线程”按钮添加事件处理函数，代码如下：

```
void CTestDlg::OnBnClickedButton2()
{
 // TODO:  在此添加控件通知处理程序代码
 CloseHandle(CreateThread(NULL, 0, ThreadProc, NULL, NULL, &m dwThID));
}
```

线程函数是 ThreadProc。因为我们不需要控制线程，所以创建线程后，马上调用函数 CloseHandle 关闭其句柄。线程 ID 保存在 m_dwThID 中，该变量是类 CTestDlg 的成员变量：

```
DWORD m_dwThID;
```

在 TestDlg.cpp 开头定义两个自定义消息：

```
#define MYMSG1 WM_USER+1
#define MYMSG2 WM_USER+2
```

为“发送线程消息 1”按钮添加事件处理函数，代码如下：

```
void CTestDlg::OnBnClickedButton1()
{
 // TODO:  在此添加控件通知处理程序代码
 CString str = _T("祖国");
 //向 ID 为 m_dwThID 的线程发送消息
 PostThreadMessage(m_dwThID, MYMSG1, WPARAM(&str),0);
 Sleep(100); //等待 100 毫秒
}
```

把字符串 str 作为消息参数发送给线程函数，然后主线程等待 100 毫秒，这样可以让子线程有机会把字符串显示一下，如果不等待，因为 PostThreadMessage 函数会立即返回，所以函数 OnBnClickedButton1 会很快结束，则局部变量 str 会很快销毁，子线程将收不到字符串。

同样，为“发送线程消息 2”按钮添加事件处理函数，代码如下：

```
void CTestDlg::OnBnClickedButton3()
{
 // TODO:  在此添加控件通知处理程序代码
 CString str = _T("强大");
 //向 ID 为 m_dwThID 的线程发送消息
 PostThreadMessage(m_dwThID, MYMSG1, WPARAM(&str), 0);
 Sleep(100); //等待 100 毫秒
}
```

把字符串 str 作为消息参数发送给线程函数，然后主线程等待 100 毫秒。

（3）保存工程并运行，运行结果如图 3-11 所示。

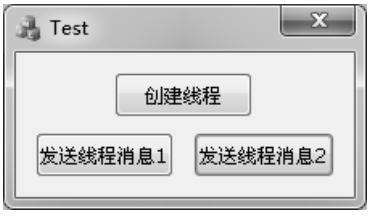


图 3-11

3.2.6 线程同步

线程同步是多线程编程中重要的概念。它的基本意思就是同步各个线程对资源（比如全局变量、文件）的访问。如果不对资源访问进行线程同步，就会产生资源访问冲突的问题。比如，一个线程正在读取一个全局变量，而读取全局变量的这个语句在 C++语言中只是一条语句，但在 CPU 指令处理这个过程的时候需要用多条指令来处理这个读取变量的过程。如果这一系列指令被另外一个线程打断了，也就是说 CPU 还没有执行完全部读取变量的所有指令就去执行另外一个线程了，而另外一个线程却要对这个全局变量进行修改，修改完后又返回原先的线程，继续执行读取变量的指令，此时变量的值已经改变了，这样第一个线程的执行结果就不是预料的结果了。

因此，多个线程对资源进行访问，一定要进行同步。VC2017 提供了临界区对象、互斥对象和事件对象和信号量对象等 4 个同步对象来实现线程同步。

下面我们来看一个线程不同步的例子。模拟这样一个场景，甲乙两个窗口在售票，一共 10 张票，每张票的号码不同，每卖出一张票，就打印出卖出票的票号。我们可以把开辟的两个线程当作两个窗口在卖票，如果线程没有同步，就可能会出现两个“窗口”卖出的“票”是相同的，就发生了错误。

【例 3.11】不用线程同步的卖票程序

（1）新建一个控制台工程。

（2）在 Test.cpp 中输入 main 函数代码：

```
int _tmain(int argc, _TCHAR* argv[])
{
 int i;
 HANDLE h[2];

 for (i = 0; i < 2;i++)
     h[i] = CreateThread(NULL, 0, threadfunc, (LPVOID)i, 0, 0);
 for (i = 0; i < 2; i++)
 {
     WaitForSingleObject(h[i], INFINITE);
```

```
        CloseHandle(h[i]);
    }
   printf("卖票结束\n");
    return 0;
   }
```

首先开启两个线程，线程函数是 threadfunc，并把 i 作为参数传入（为了区分不同的窗口）。最后无限等待两个线程结束，一旦结束就关闭其线程句柄。

在 main 函数上面输入线程函数和全局变量，代码如下：

```
   #define  BUF_SIZE 100
   int gticketId = 10; //当前卖出票的票号
   DWORD WINAPI threadfunc(LPVOID param)
   {
    HANDLE hStdout;
    DWORD  i,dwChars;
    size_t szlen;
    TCHAR chWin, msgBuf[BUF_SIZE];

    if (param == 0) chWin = _T('甲'); //甲窗口
    else chWin = _T('乙'); //乙窗口


    while (1)
    {
        if (gticketId <= 0) //如果票号小于等于零，就跳出循环
            break;
        hStdout = GetStdHandle(STD_OUTPUT_HANDLE); //为了打印，得到标准输出设备的句柄
        if (hStdout == INVALID_HANDLE_VALUE)
        {
            return 1;
        }
        //构造字符串
        StringCchPrintf(msgBuf, BUF_SIZE, _T("%c 窗口卖出的车票号 = %d\n"), chWin,
gticketId);
       StringCchLength(msgBuf, BUF_SIZE, &szlen);  //得到字符串长度
        WriteConsole(hStdout, msgBuf, szlen, &dwChars, NULL);
        gticketId--;//每卖出一张车票，车票就减少一张
    }
   }
```

线程不停地卖票，每次卖出一张票就打印出车票号，同时减少一张。

最后添加所需头文件：

```
   #include "windows.h"
   #include <strsafe.h> //字符串处理函数需要
```

（3）保存工程并运行，从运行结果（见图 3-12）可以看出不同的窗口居然卖出了同号的车票，这就说明没有线程同步的话程序出现问题了。

图 3-12

3.2.6.1 临界区对象

临界区对象通过一个所有线程共享的对象来实现线程同步。线程要访问被临界区对象保护的资源，必须先要拥有该临界区对象。如果另一个线程要访问资源，则必须等待上一个访问资源的线程释放临界区对象。临界区对象只能用于一个进程内的不同线程之间的同步。

临界区的意思是一段关键代码，执行代码相当于进入临界区。要执行临界区代码，必须先独占临界区对象。比如可以把对某个共享资源进行访问这个操作看作一个临界区，要执行这段代码（访问共享资源），必须先拥有临界区对象。临界区对象好比一把钥匙，只有拥有了这把钥匙才能对共享资源进行访问。如果这把钥匙在其他线程手里，则当前线程只能等待，一直等到其他线程交出钥匙。VC2017 提供了几个操作临界区对象的函数。

（1）InitializeCriticalSection 函数

该函数用来初始化一个临界区对象。函数声明如下：

```
void  InitializeCriticalSection(LPCRITICAL_SECTION  lpCriticalSection);
```

其中，参数 lpCriticalSection 为指向一个临界区对象的指针。CRITICAL_SECTION 是一个结构体，定义了和线程访问相关的控制信息，具体内容我们不需要去管，它定义在 WinBase.h 中。

通常使用该函数之前会先定义一个 CRITICAL_SECTION 类型的全局变量，然后把地址传入该函数。

（2）EnterCriticalSection 函数

该函数用于等待临界区对象的所有权，如果能获得临界区对象，那么该函数返回，否则函数进入阻塞，线程进入睡眠状态，一直到拥有临界区对象的线程释放临界区对象。该函数声明如下：

```
void  EnterCriticalSection( LPCRITICAL_SECTION  lpCriticalSection);
```

其中，参数 lpCriticalSection 为指向一个临界区对象的指针。

（3）TryEnterCriticalSection 函数

该函数也是用于等待临界区对象的所有权，和 EnterCriticalSection 不同的是，函数

TryEnterCriticalSection 不管有没有获取到临界区对象所有权，都将立即返回，相当于一个异步函数。函数声明如下：

```
BOOL  TryEnterCriticalSection( LPCRITICAL_SECTION  lpCriticalSection);
```

其中，参数 lpCriticalSection 为指向一个临界区对象的指针。如果成功获取临界区对象所有权，函数就返回非零，否则返回零。

（4）LeaveCriticalSection 函数

该函数用于释放临界区对象的所有权。声明如下：

```
void  LeaveCriticalSection(  LPCRITICAL_SECTION  lpCriticalSection);
```

其中，参数 lpCriticalSection 为指向一个临界区对象的指针。需要注意的是，线程获得临界区对象所有权，在使用完临界区后必须调用该函数释放临界区对象的所有权，让其他等待临界区的线程有机会进入临界区。该函数通常和 EnterCriticalSection 函数配对使用，它们中间的代码就是临界区代码。

（5）DeleteCriticalSection 函数

该函数用来删除临界区对象，释放相关资源，使得临界区对象不再可用。函数声明如下：

```
void  DeleteCriticalSection(  LPCRITICAL_SECTION  lpCriticalSection);
```

其中，参数 lpCriticalSection 为指向一个临界区对象的指针。

下面我们对前面线程不同步的卖票例子进行改造，加入临界区对象，使得线程同步。

【例 3.12】使用临界区对象同步线程

（1）新建一个控制台工程。

（2）在 Test.cpp 中输入如下代码：

```
#include "stdafx.h"
#include "windows.h"
#include <strsafe.h>

#define  BUF_SIZE 100 //输出缓冲区大小
int gticketId = 10; //记录卖出的车票号
CRITICAL_SECTION gcs; //定义临界区对象

DWORD WINAPI threadfunc(LPVOID param)
{
 HANDLE hStdout;
 DWORD  i, dwChars;
 size_t szlen;
 TCHAR chWin,msgBuf[BUF_SIZE];

 if (param == 0) chWin = _T('甲'); //甲窗口
 else chWin = _T('乙'); //乙窗口
 while (1)
 {
```

```
        EnterCriticalSection(&gcs);
        if (gticketId <= 0)
        {
            LeaveCriticalSection(&gcs); //注意要释放临界区对象所有权
            break;
        }

        hStdout = GetStdHandle(STD_OUTPUT_HANDLE); //得到标准输出设备的句柄，为了打印
        if (hStdout == INVALID_HANDLE_VALUE)
        {
            LeaveCriticalSection(&gcs); //注意要释放临界区对象所有权
            return 1;
        }
        //构造字符串
        StringCchPrintf(msgBuf, BUF_SIZE, _T("%c 窗口卖出的车票号 = %d\n"), chWin,
gticketId);         StringCchLength(msgBuf, BUF_SIZE, &szlen); //得到字符串长度
        WriteConsole(hStdout, msgBuf, szlen, &dwChars, NULL); //在终端打印车票号
        gticketId--;//车票减少一张
        LeaveCriticalSection(&gcs); 释放临界区对象所有权
        Sleep(1); //让出 CPU，让另外的线程有机会执行
    }
   }
   int _tmain(int argc, _TCHAR* argv[])
   {
    int i;
    HANDLE h[2];

    InitializeCriticalSection(&gcs); //初始化临界区对象

    for (i = 0; i < 2; i++)
        h[i] = CreateThread(NULL, 0, threadfunc, (LPVOID)i, 0, 0); //开辟两个线程
    for (i = 0; i < 2; i++)
    {
        WaitForSingleObject(h[i], INFINITE); //等待线程结束
        CloseHandle(h[i]);
    }
    DeleteCriticalSection(&gcs); //删除临界区对象
    printf("卖票结束\n");
    return 0;
   }
```

程序中使用了临界区对象来同步线程。gcs 是临界区对象，通常定义成一个全局变量。在线程函数中，我们把用到全局变量 gticketId 的地方都包围进临界区内，这样一个线程在使用共享的全局变量 gticketId 时，其他线程就只能等待了。

（3）保存工程并运行，可以看到每次卖出的车票的号码都是不同的，运行结果如图 3-13 所示。

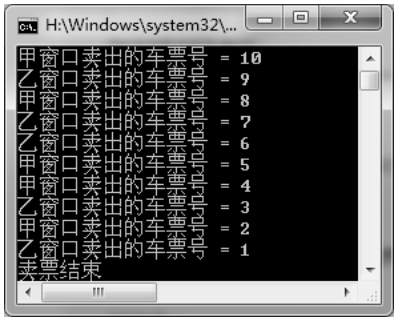


图 3-13

3.2.6.2　互斥对象

互斥对象也称互斥量（Mutex），它的使用和临界区对象有点类似。互斥对象不仅能保护一个进程内的共享资源，还能保护系统中进程间的资源共享。互斥对象属于系统内核对象。

只有拥有互斥对象的线程才具有访问资源的权限，由于互斥对象只有一个，因此就决定了任何情况下共享资源都不会同时被多个线程所访问。互斥对象的使用通常需要结合等待函数，当没有线程拥有互斥对象时，系统会为互斥对象设置有信号状态（相当于向外发送信号），此时若有线程在等待该互斥对象（利用等待函数在等待），则该线程可以获得互斥对象，此时系统会将互斥对象设为无信号状态（不向外发送信号），如果又有线程在等待，则只能一直等待下去，直到拥有互斥对象的线程释放互斥对象，然后系统重新设置互斥对象为有信号状态。

下面先介绍一下等待函数。我们前面已经接触过 WaitForSingleObject 函数，这个就是等待函数，类似的还有 WaitForMultipleObjects。所谓等待函数，就是用来等待某个对象产生信号的函数，比如一个线程对象在线程生命期内是处于无信号状态的，当线程终止时系统会设置线程对象为有信号状态，因此我们可以用等待函数来等待线程的结束。类似的，互斥对象没有被任何线程拥有的时候，系统会将它设置为有信号状态，一旦被某个线程拥有，就会设为无信号状态。线程可以调用等待函数来阻塞自己，直到信号产生后等待函数才会返回，线程才会继续执行。

函数 WaitForSingleObject 用来等待某个对象的信号，它知道对象有信号或等待超时才返回，函数声明如下：

```
DWORD  WaitForSingleObject(HANDLE hHandle, DWORD dwMilliseconds);
```

其中，参数 hHandle 是对象句柄；dwMilliseconds 表示等待超时的时间，单位是毫秒，如果该参数是 0，那么函数测试对象信号状态后立即返回，如果参数是宏 INFINITE，表示函数不设超时时间一直等待对象有信号为止。如果函数成功，那么函数返回值如下：

- WAIT_ABANDONED，表示指定的对象是互斥对象，该互斥对象在拥有它的线程结束时没有被释放，互斥对象的所有权将被赋予调用本函数的线程，同时互斥对象被设为无信号状态。
- WAIT_OBJECT_0，表示指定的对象处于有信号状态了。
- WAIT_TIMEOUT，表示等待超时，同时对象仍处于无信号状态。

如果函数失败，则返回 WAIT_FAILED ((DWORD)0xFFFFFFFF)，相当于-1。

WaitForMultipleObjects 可以用来等待多个对象，但数目不能超过 64。该函数相当于在循环中调用 WaitForSingleObject，一般用 WaitForSingleObject 即可。

下面介绍与互斥对象有关的 API 函数。

（1）CreateMutex 函数

该函数创建或打开一个互斥对象。声明如下：

```
HANDLE CreateMutex( LPSECURITY_ATTRIBUTES lpMutexAttributes,
BOOL bInitialOwner, LPCTSTR lpName );
```

其中，参数 lpMutexAttributes 为指向 PSECURITY_ATTRIBUTES 结构的指针，该结构表示互斥的安全属性，主要决定函数返回的互斥对象句柄能否被子进程继承，如果该参数为 NULL，则函数返回的句柄不能被子进程继承；bInitialOwner 决定调用该函数创建互斥对象的线程是否拥有该互斥对象的所有权，如果该参数为 TRUE，表示创建该互斥对象的线程拥有该互斥对象的所有权；lpName 是一个字符串指针，用来确定互斥对象的名称，该名称区分大小写，长度不能超过 MAX_PATH，如果该参数为 NULL，则不给互斥对象起名（为互斥对象起名字的目的是在不同进程之间进行线程同步）。如果函数成功就返回互斥对象句柄，否则函数返回 NULL。

（2）ReleaseMutex 函数

该函数用来释放互斥对象的所有权，这样其他等待互斥对象的线程就可以获得所有权。函数声明如下：

```
BOOL  ReleaseMutex( HANDLE hMutex);
```

其中，参数 hMutex 是互斥对象的句柄。如果函数成功就返回非零，否则返回零。需要注意的是，函数 ReleaseMutex 是用来释放互斥对象所有权的，并不是销毁互斥对象。当进程结束的时候，系统会自动关闭互斥对象句柄，也可以使用 CloseHandle 函数来关闭互斥对象句柄，当最后一个句柄被关闭的时候系统销毁互斥对象。

下面我们通过互斥对象实现线程同步来改写例 3.11。

【例 3.13】使用互斥对象同步线程

（1）新建一个控制台工程。

（2）在 Test.cpp 中输入如下代码：

```
#include "stdafx.h"
#include "windows.h"
#include <strsafe.h>

#define  BUF_SIZE 100 //输出缓冲区大小
int gticketId = 10;  //记录卖出的车票号
HANDLE ghMutex; //互斥对象句柄

DWORD WINAPI threadfunc(LPVOID param)
```

```
    {
    HANDLE hStdout;
    DWORD  i, dwChars;
    size_t szlen;
    TCHAR chWin, msgBuf[BUF_SIZE];

    if (param == 0) chWin = _T('甲'); //甲窗口
    else chWin = _T('乙'); //乙窗口
    while (1)
    {
        WaitForSingleObject(ghMutex, INFINITE); //等待互斥对象有信号
        if (gticketId <= 0) //如果车票全部卖出了，则退出循环
        {
            ReleaseMutex(ghMutex); //释放互斥对象所有权
            break;
        }

        hStdout = GetStdHandle(STD_OUTPUT_HANDLE); //得到标准输出设备的句柄，为了打印
        if (hStdout == INVALID_HANDLE_VALUE)
        {
            ReleaseMutex(ghMutex);
            return 1;
        }
        //构造字符串
        StringCchPrintf(msgBuf, BUF_SIZE, _T("%c 窗口卖出的车票号 = %d\n"), chWin,
gticketId);
        StringCchLength(msgBuf, BUF_SIZE, &szlen);
        WriteConsole(hStdout, msgBuf, szlen, &dwChars, NULL); //控制台输出
        gticketId--;//车票减少一张
        ReleaseMutex(ghMutex); //释放互斥对象所有权
        //Sleep(1); //这句可以不用了
    }
    }
    int _tmain(int argc, _TCHAR* argv[])
    {
    int i;
    HANDLE h[2];

    printf("使用互斥对象同步线程\n");
    ghMutex = CreateMutex(NULL, FALSE, _T("myMutex")); //创建互斥对象

    for (i = 0; i < 2; i++)
        h[i] = CreateThread(NULL, 0, threadfunc, (LPVOID)i, 0, 0); //创建线程
    for (i = 0; i < 2; i++)
    {
        WaitForSingleObject(h[i], INFINITE); //等待线程结束
        CloseHandle(h[i]); //关闭线程对象句柄
    }
    CloseHandle(ghMutex); //关闭互斥对象句柄
    printf("卖票结束\n");
    return 0;
```

```
}
```

程序通过互斥对象来实现线程同步。主线程中首先创建互斥对象，并把句柄存在全局变量 ghMutex 中，创建的时候第二个参数是 FALSE，意味着主线程不拥有该互斥对象所有权。在线程函数中，在用到共享的全局变量 gticketId 之前调用等待函数 WaitForSingleObject 来等待互斥对象有信号，一旦等到，就可以进行关于 gticketId 的操作了。等操作完毕后再用函数 ReleaseMutex 来释放互斥对象所有权，使得互斥对象重新有信号，这样其他等待该互斥对象的线程可以得以执行。

与例 3.12 使用临界区对象来实现线程同步相比，该例的线程函数中不需要用 Sleep(1)来使得当前线程让出 CPU，因为其他线程已经在等待信号对象的信号了，一旦拥有互斥对象的线程释放所有权，其他线程马上可以等待结束，得以执行。

（3）保存工程并运行，由运行结果（见图 3-14）可以看出每次卖出的车票的号码都是不同的。

图 3-14

3.2.6.3 事件对象

事件对象也属于系统内核对象。它的使用方式和互斥对象有点类似，但功能更多一些。当等待的事件对象有信号状态时，等待事件对象的线程得以恢复，继续执行；如果等待的事件对象处于无信号状态，则等待该对象的线程将挂起。

事件可以分为两种：手动事件和自动事件。手动事件的意思是当事件对象处于有信号状态时，它会一直处于这个状态，一直到调用函数将其设置为无信号状态为止。自动事件是指当事件对象处于有信号状态时，如果有一个线程等待到该事件对象的信号后，事件对象就变为无信号状态了。

事件对象也要使用等待函数，比如 WaitForSingleObject。关于等待函数上一节已经介绍过了，这里不再赘述。

下面介绍有关事件对象的几个 API 函数。

（1）CreateEvent 函数

该函数用于创建或打开一个事件对象，声明如下：

```
HANDLE  CreateEvent(LPSECURITY_ATTRIBUTES lpEventAttributes, BOOL
```

```
bManualReset,BOOL bInitialState, LPCTSTR lpName);
```

其中，参数 lpEventAttributes 是指向 SECURITY_ATTRIBUTES 结构的指针，该结构表示一个安全属性，如果该参数为 NULL，表示函数返回的句柄不能被子进程继承；bManualReset 用于确定是创建一个手动事件还是一个自动事件；bInitialState 用于指定事件对象的初始状态，如果为 TRUE 就表示事件对象创建后处于有信号状态，否则为无信号状态；lpName 指向一个字符串，该字符串表示事件对象的名称，该名称字符串是区分大小写的，长度不能超过 MAX_PATH，如果该参数为 NULL，则表示创建一个无名字的事件对象，事件对象的名称不能和其他同步对象的名称（比如互斥对象的名称）相同。如果函数成功就返回新创建的事件对象句柄，否则返回 NULL。

（2）SetEvent 函数

该函数将事件对象设为有信号状态。

```
BOOL   SetEvent(  HANDLE  hEvent);
```

其中，参数 hEvent 表示事件对象句柄。如果函数成功就返回非零，否则返回零。

（3）ResetEvent 函数

该函数将事件对象重置为无信号状态。声明如下：

```
BOOL   ResetEvent(HANDLE  hEvent);
```

其中，参数 hEvent 是事件对象句柄，如果函数成功就返回非零，否则返回零。

当进程结束的时候，系统会自动关闭事件对象句柄，也可以调用 CloseHandle 来关闭事件对象句柄，当与之关联的最后一个句柄被关掉后，事件对象被销毁。

下面我们通过事件对象实现线程同步来改写例 3.11。

【例 3.14】使用事件对象同步线程

（1）新建一个控制台工程。

（2）在 Test.cpp 中输入如下代码：

```
#include "stdafx.h"
#include "windows.h"
#include <strsafe.h>

#define  BUF_SIZE 100 //输出缓冲区大小
int gticketId = 10;  //记录卖出的车票号
HANDLE ghEvent; //事件对象句柄

DWORD WINAPI threadfunc(LPVOID param)
{
 HANDLE hStdout;
 DWORD  i, dwChars;
 size_t szlen;
 TCHAR chWin, msgBuf[BUF_SIZE];
```

```
    if (param == 0) chWin = _T('甲'); //甲窗口
    else chWin = _T('乙'); //乙窗口
    while (1)
    {
        WaitForSingleObject(ghEvent, INFINITE); //等待事件对象有信号
        if (gticketId <= 0) //如果车票全部卖出了，就退出循环
        {
            SetEvent(ghEvent); //设置事件对象有信号
            break;
        }

        hStdout = GetStdHandle(STD_OUTPUT_HANDLE); //得到标准输出设备的句柄，为了打印
        if (hStdout == INVALID_HANDLE_VALUE)
        {
            SetEvent(ghEvent); //释放事件对象所有权
            return 1;
        }
        //构造字符串
        StringCchPrintf(msgBuf, BUF_SIZE, _T("%c 窗口卖出的车票号 = %d\n"), chWin,
gticketId);
        StringCchLength(msgBuf, BUF_SIZE, &szlen);
        WriteConsole(hStdout, msgBuf, szlen, &dwChars, NULL); //控制台输出
        gticketId--;//车票减少一张
        SetEvent(ghEvent); //设置事件对象有信号
        //Sleep(1); //这句可以不用了
    }
   }
   int _tmain(int argc, _TCHAR* argv[])
   {
    int i;
    HANDLE h[2];
    printf("使用事件对象同步线程\n");
    ghEvent = CreateEvent(NULL, FALSE, TRUE,_T("myEvent")); //创建事件对象

    for (i = 0; i < 2; i++)
        h[i] = CreateThread(NULL, 0, threadfunc, (LPVOID)i, 0, 0); //创建线程
    for (i = 0; i < 2; i++)
    {
        WaitForSingleObject(h[i], INFINITE); //等待线程结束
        CloseHandle(h[i]); //关闭线程对象句柄
    }
    CloseHandle(ghEvent); //关闭事件对象句柄
    printf("卖票结束\n");
    return 0;
   }
```

程序利用事件对象来同步两个线程。首先创建一个事件对象，并在开始时设置有信号状态。然后在使用共享的全局变量 gticketId 之前需要等待，等到事件对象的信号后线程开始操作与 gticketId 有关的代码，同时事件对象处于无信号状态，一旦与 gticketId 有关操作完成就利用 SetEvent 函数设置事件对象为有信号状态，以便其他在等待事件对象的线程能得以执行。

（3）保存工程并运行，由运行结果（见图 3-15）可以看出每次卖出的车票的号码都是不同的。

图 3-15

3.2.6.4　信号量对象

信号量对象也是一个内核对象。它的工作原理是：信号量内部有计数器，当计数器大于零时，信号量对象处于有信号状态，此时等待信号量对象的线程得以继续进行，同时信号量对象的计数器减一；当计数器为零时，信号量对象处于无信号状态，此时等待信号量对象的线程将被阻塞。下面介绍和信号量操作有关的 API 函数。

（1）CreateSemaphore 函数

该函数创建或打开一个信号量对象，声明如下：

```
HANDLE  CreateSemaphore (LPSECURITY_ATTRIBUTES lpSemaphoreAttributes,
   LONG lInitialCount, LONG lMaximumCount, LPCTSTR lpName);
```

其中，参数 lpSemaphoreAttributes 指向 SECURITY_ATTRIBUTES 结构的指针，该结构表示安全属性，如果为 NULL，就表示函数返回的句柄不能被子进程继承；lInitialCount 表示信号量的初始计数，该参数必须大于等于零，并且小于等于 lMaximumCount；lMaximumCount 指定信号量对象计数器的最大值，该参数必须大于零；lpName 指向一个字符串，该字符串指定信号量对象的名称，区分大小写，并且长度不能超过 MAX_PATH，如果为 NULL，则创建一个无名信号量对象。如果函数成功就返回信号量对象句柄，如果指定名字的信号量对象已经存在，就返回那个已经存在的信号量对象的句柄，如果函数失败就返回 NULL。

（2）ReleaseSemaphore 函数

该函数用来为信号量对象的计数器增加一定数量，声明如下：

```
BOOL  ReleaseSemaphore(HANDLE hSemaphore, LONG lReleaseCount, LPLONG
lpPreviousCount);
```

其中，参数 hSemaphore 为信号量对象句柄；lReleaseCount 指定要将信号量对象的当前计数器增加的数目，该参数必须大于零，如果该参数使得计数器的值大于其最大值（在创建信号量对象的时候设定），计数器值将保持不变，并且函数返回 FALSE；lpPreviousCount 指向一

个变量，该变量存储信号量对象计数器的前一个值。如果函数成功就返回非零，否则返回零。

下面我们通过信号量对象实现线程同步来改写例 3.11。

【例 3.15】使用信号量对象同步线程

（1）新建一个控制台工程。

（2）在 Test.cpp 中输入如下代码：

```
#include "stdafx.h"
#include "windows.h"
#include <strsafe.h>

#define  BUF_SIZE 100 //输出缓冲区大小
int gticketId = 10;  //记录卖出的车票号
HANDLE ghSemaphore; //信号量对象句柄

DWORD WINAPI threadfunc(LPVOID param)
{
 HANDLE hStdout;
 DWORD  i, dwChars;
 size_t szlen;
 LONG cn;
 TCHAR chWin, msgBuf[BUF_SIZE];

 if (param == 0) chWin = _T('甲'); //甲窗口
 else chWin = _T('乙'); //乙窗口
 while (1)
 {
     WaitForSingleObject(ghSemaphore, INFINITE); //等待信号量对象有信号
     if (gticketId <= 0) //如果车票全部卖出了，就退出循环
     {
         ReleaseSemaphore(ghSemaphore,1,&cn); //释放信号量对象所有权
         break;
     }

     hStdout = GetStdHandle(STD_OUTPUT_HANDLE); //得到标准输出设备的句柄，为了打印
     if (hStdout == INVALID_HANDLE_VALUE)
     {
         ReleaseSemaphore(ghSemaphore,1, &cn); //释放信号量对象所有权
         return 1;
     }
     //构造字符串
     StringCchPrintf(msgBuf, BUF_SIZE, _T("%c 窗口卖出的车票号 = %d\n"), chWin,
gticketId);
     StringCchLength(msgBuf, BUF_SIZE, &szlen);
     WriteConsole(hStdout, msgBuf, szlen, &dwChars, NULL); //控制台输出
     gticketId--;//车票减少一张
     ReleaseSemaphore(ghSemaphore,1, &cn); //释放信号量对象所有权
     //Sleep(1); //这句可以不用了
 }
```

```
}
int _tmain(int argc, _TCHAR* argv[])
{
 int i;
 HANDLE h[2];
 printf("使用信号量对象同步线程\n");
 ghSemaphore = CreateSemaphore(NULL, 1, 50, _T("mySemaphore"));//创建信号量对象

 for (i = 0; i < 2; i++)
     h[i] = CreateThread(NULL, 0, threadfunc, (LPVOID)i, 0, 0); //创建线程
 for (i = 0; i < 2; i++)
 {
     WaitForSingleObject(h[i], INFINITE); //等待线程结束
     CloseHandle(h[i]); //关闭线程对象句柄
 }
 CloseHandle(ghSemaphore); //关闭信号量对象句柄
 printf("卖票结束\n");
 return 0;
}
```

上面的代码通过信号量对象来同步两个线程。首先创建一个计数器为 1 的信号量对象，因为信号量计数器大于 0，所以信号量对象处于有信号状态，然后在子线程中的等待函数就可以等到该信号，并且信号量对象计数器减一变为零，则其他等待函数就只能阻塞了，等到共享的全局变量 gticketId 操作完成后，让信号量对象计数器加 1，计数器大于零了则信号量对象重新变为有信号状态，其他线程得以等待返回继续执行。

（3）保存工程并运行，运行结果如图 3-16 所示。

图 3-16

3.3 CRT 库中的多线程函数

CRT 库的全称是 C Run-time Libraries，即 C 运行时库，包含了 C 常用的函数（如 printf、

malloc、strcpy 等），为运行 main 做了初始化环境变量、堆、IO 等资源，并在结束后清理。在 Windows 环境下，VC2017 提供的 C Run-time Libraries 分为动态运行时库、静态运行时库、调试版本（Debug）、发行版本（Release）等，它们都是支持多线程的，以前老的 VC 版本还有单线程版本 CRT，现在单线程版本 CRT 已经淘汰了。我们可以在 IDE 工程属性中进行设置，选择不同版本的 CRT，比如打开工程属性对话框，然后在左边选择“C/C++”→“代码生成”，在右边的“运行库”旁边可以选择不同的 CRT 库，如图 3-17 所示。

图 3-17

其中，/MT 表示多线程静态链接的 Release 版本的 CRT 库，在 LIBCMT.LIB 中实现。/MTd 表示多线程静态链接的 Debug 版本的 CRT 库，在 LIBCMTD.LIB 中实现。/MD 表示多线程 DLL 的 Release 版本的 CRT 库，在 MSVCRT.LIB 中实现。/MDd 表示多线程 DLL 的 Debug 版本的 CRT 库，在 MSCVRTD.LIB 中实现。通常这里保持默认即可。

CRT 库中提供了创建线程和结束线程的函数，比如创建线程函数_beginthread 和_beginthreadex、结束线程函数_endthread 和_endthreadex。_beginthread 和_endthread 对应使用，_beginthreadex 和_endthreadex 对应使用。前面 Win32 API 函数 CreateThread 创建的线程中不应使用 CRT 库中的函数，现在_beginthread 和_beginthreadex 创建的线程则可以使用 CRT 库函数。其实，在_beginthread 和_beginthreadex 内部都调用了 API 函数 CreateThread，但在调用该 API 函数前做了很多初始化工作，在调用后又做了不少检查工作，这使得线程能更好地支持 CRT 库函数。函数_endthread 和_endthreadex 的内部其实调用了 API 函数 ExitThread，但它们还做了许多善后工作。

如果要在控制台程序下使用 CRT 中的线程函数，就要包括头文件 process.h。

函数_beginthread 声明如下：

```
uintptr_t  _beginthread( void( *start_address )( void * ), unsigned  stack_size,
```

```
void *arglist );
```

其中，参数 start_address 是线程函数的起始地址，该线程函数的调用约定必须是__cdecl 或__clrcall（用于托管）；stack_size 是线程的堆栈大小，如果为零，就使用系统默认值；arglist 指向传给线程函数参数的指针。函数如果成功就返回线程句柄（根据平台不同，uintptr_t 可能为 unsigned integer 或 unsigned __int64），如果失败就返回-1。需要注意的是，如果创建的线程很快退出了，则_beginthread 可能返回一个无效句柄。

_beginthread 创建的线程可以用函数_endthread 来结束，该函数声明如下：

```
void _endthread();
```

如果在线程函数中使用_endthread，该函数后面的代码将得不到执行。此外，当线程函数返回的时候系统也会自动调用_endthread，并且_endthread 会自动关闭线程句柄。正因为这个原因，我们不需要再去显式调用 CloseHandle 函数来关闭线程句柄，而且也不应该在主线程中使用等待函数（比如 WaitForSingleObject）来等待子线程句柄的方式去判断子线程是否结束，比如如下代码可能会出现句柄无效的异常报错：

```
WaitForSingleObject((HANDLE)ghThread1, INFINITE); //等待子线程退出
CloseHandle((HANDLE)ghThread1);//关闭线程句柄
```

单步调式时很容易报错，如图 3-18 所示。

图 3-18

正确的方式是如果要等待_beginthread 创建的线程结束，就可以使用同步对象，比如事件等，后面的例子我们会演示。

函数_beginthreadex 比_beginthread 功能强大一些，并且更安全些，声明如下：

```
uintptr_t _beginthreadex(void *security, unsigned stack_size, unsigned
( *start_address )( void * ),void *arglist, unsigned initflag, unsigned *thrdaddr );
```

其中，参数 security 表示线程的安全描述符；stack_size 是线程的堆栈大小，如果为零，就使用系统默认值；start_address 是线程函数的起始地址，该线程函数的调用约定必须是

__stdcall 或__clrcall（用于托管）；arglist 指向传给线程函数参数的指针；initflag 用于指示线程创建后是否立即执行，0 表示立即执行，CREATE_SUSPENDED 表示创建后挂起；thrdaddr 指向一个 32 位的变量，该变量用来存放线程 ID。函数如果成功就返回线程句柄（根据平台不同，uintptr_t 可能为 unsigned integer 或 unsigned __int64），如果失败就返回 0。

_beginthread 相当于_beginthreadex 的功能子集，但是使用_beginthread 既无法创建带有安全属性的新线程，也无法创建初始能暂停的线程，还无法获得线程 ID。

_beginthreadex 的功能类似于 API 函数 CreateThread，虽然功能类似，但是推荐使用_beginthreadex，这是因为不少人对 CRT 函数更熟悉些，所以在线程函数中的某些需求经常会想用 CRT 函数去解决。前面提到过，在 CreateThread 创建的线程中使用 CRT 函数会产生一些内存泄漏。

_beginthreadex 创建的线程可以使用函数_endthreadex 来结束，如果在线程函数中调用_endthreadex，那么该函数后面的代码将都不会执行。同样，_beginthreadex 创建的线程函数返回时，系统会自动调用_endthreadex，但_endthreadex 并不会去关闭线程句柄，所以要开发者显式地调用 CloseHanlde 来关闭线程句柄。因为_endthreadex 并不会去关闭线程句柄，所以可以在主线程中使用等待函数（比如 WaitForSingleObject）来等待子线程句柄，以此判断子线程是否结束。_beginthreadex 函数的使用流程和 CreateThread 几乎一样。

下面看几个小例子，第一个例子利用_beginthread 函数不断创建线程，看最多能创建多少个线程。第二个例子和前面章节类似的卖票程序，用互斥对象来同步_beginthread 函数创建的两个线程，这是一个控制台程序，在这个程序中我们要向控制台打印信息，可以直接使用 CRT 库中的 printf 函数，因为线程也是 CRT 库函数_beginthread 创建的。

【例 3.16】利用_beginthread 不断创建线程

（1）新建一个对话框工程。

（2）切换到资源视图，打开对话框编辑器，删除上面所有的控件，然后添加 4 个按钮和 2 个静态控件，按钮的标题分别设为“启动”“暂停”“继续”和“结束线程”，一个静态控件的标题设为“已经创建的线程数：”，并把该静态控件放在左上角，然后把另外一个静态控件放在它的右边，并设 ID 为 IDC_THREAD_COUNT。双击“启动”按钮，添加事件处理函数，代码如下：

```
void CTestDlg::OnBnClickedButton1()
{
 // TODO:  在此添加控件通知处理程序代码
 if (_beginthread(threadFunc1, 0, m_hWnd) != -1) //创建线程
     GetDlgItem(IDC_BUTTON1)->EnableWindow(FALSE); //按钮变为不可用
if (!ghEvent)
     ghEvent = CreateEvent(NULL, FALSE, FALSE, NULL);
}
```

一旦成功创建线程，按钮就变为不可用。其中，ghEvent 是一个事件句柄，是全局变量，定义如下：

```
HANDLE ghEvent = NULL;
```

通过这个事件句柄我们将用于等待子线程的退出。threadFunc1 是线程函数，并把对话框句柄 m_hWnd 作为参数传给线程函数。threadFunc1 函数的代码如下：

```
void threadFunc1(void *pArg)
{
 HWND hWnd = (HWND)pArg;
 g_nCount = 0;
 g_bRun = true;
 while (g_bRun) //不断地创建新的线程
 {
     if (_beginthread(threadFunc2, 0, hWnd) == -1)
     {
         g_bRun = false; //如果创建失败了，就置 false，准备退出循环
         break;
     }
 }
 ::PostMessage(hWnd, WM_SHOW_THREADCOUNT, 1, 0); //发送消息通知，线程结束
SetEvent(ghEvent); //设置事件状态
}
```

代码很简单，就是不停地在循环中创建线程，一直到失败。需要注意的是，程序结尾用 PostMessage ，不要用 SendMessage，因为我们后面主线程会等待子线程的结束，等待的时候主线程会挂起，所以如果用 SendMessage，SendMessage 就会无法返回（因为主线程挂起了），这样子线程和主线程互相等待了。其中，g_nCount 和 g_bRun 都是全局变量，定义如下：

```
bool g_bRun = false; // 控制循环结束
long g_nCount = 0; //统计所创建的线程个数
```

WM_SHOW_THREADCOUNT 是自定义消息，定义如下：

```
#define WM_SHOW_THREADCOUNT WM_USER+5
```

threadFunc2 也是线程函数，定义如下：

```
void threadFunc2(void *pArg)
{
 HWND hWnd = (HWND)pArg;

 g_nCount++; //线程个数累加
 ::SendMessage(hWnd, WM_SHOW_THREADCOUNT, 0, g_nCount);//发送消息显示线程个数
 while (g_bRun) //如果程序还在创建线程，则每个子线程一直运行
     Sleep(1000);
}
```

threadFunc2 线程函数只是把当前已经创建的线程个数通过发送消息去显示。接着添加 WM_SHOW_THREADCOUNT 的消息处理函数：

```
LRESULT CTestDlg::OnMyMsg(WPARAM wParam, LPARAM lParam)
{
 CString str;
```

```
if (wParam == 1)
    GetDlgItem(IDC_BUTTON1)->EnableWindow(TRUE);  //线程准备结束了，则让按钮使能
else
{
    str.Format(_T("%d"), g_nCount);
    GetDlgItem(IDC_THREAD_COUNT)->SetWindowText(str); //显示线程个数
    UpdateData(FALSE);
}
return 0;
}
```

别忘了添加消息映射：

```
ON_MESSAGE(WM_SHOW_THREADCOUNT, OnMyMsg)
```

（3）切换到资源视图，打开对话框编辑器，双击“暂停”按钮，为其添加事件处理函数，代码如下：

```
void CTestDlg::OnBnClickedButton2()
{
// TODO:  在此添加控件通知处理程序代码
if (ghThread1)
    SuspendThread((HANDLE)ghThread1); //用 API 函数暂停线程的执行
}
```

再为“恢复”按钮添加事件处理函数，代码如下：

```
void CTestDlg::OnBnClickedButton3()
{
// TODO:  在此添加控件通知处理程序代码
if (ghThread1)
    ResumeThread((HANDLE)ghThread1); //用 API 函数恢复线程的执行
}
```

再为“结束线程”按钮添加事件处理函数，代码如下：

```
void CTestDlg::OnBnClickedButton4()
{
// TODO:  在此添加控件通知处理程序代码
if (!ghThread1)
    return;

if (!g_bRun)
    return; //如果已经结束就直接返回
g_bRun = false; //设置循环结束变量

WaitForSingleObject(ghEvent, INFINITE); //无限等待事件有信号
CloseHandle(ghEvent); //关闭事件句柄
ghEvent = NULL;
GetDlgItem(IDC_BUTTON1)->EnableWindow();//设置“开启线程”按钮可用
}
```

（3）保存工程并运行，运行结果如图 3-19 所示。

图 3-19

【例 3.17】利用互斥对象同步_beginthread 创建的线程

（1）新建一个控制台工程。

（2）打开 Test.cpp，在其中输入如下代码：

```
#include "stdafx.h"
#include "windows.h"
#include "process.h"
#include <clocale>

int gticketId = 10;  //记录卖出的车票号
CCriticalSection gcs; // 定义 CCriticalSection 对象

void threadfunc(LPVOID param)
{
 TCHAR chWin;

 if (param == 0) chWin = _T('甲'); //甲窗口
 else chWin = _T('乙'); //乙窗口
 while (1)
 {
     gcs.
     if (gticketId <= 0) //如果车票全部卖出了，则退出循环
     {
         ReleaseMutex(ghMutex); //释放互斥对象所有权
         break;
     }
     setlocale(LC_ALL, "chs"); //为控制台设置中文环境
     _tprintf(_T("%c 窗口卖出的车票号 = %d\n"), chWin, gticketId); //打印信息
     gticketId--;//车票减少一张
     ReleaseMutex(ghMutex); //释放互斥对象所有权
 }
}
int _tmain(int argc, _TCHAR* argv[])
{
 int i;
 uintptr_t h[2];

 printf("使用互斥对象同步线程\n");
 ghMutex = CreateMutex(NULL, FALSE, _T("myMutex")); //创建互斥对象
```

```
    for (i = 0; i < 2; i++)
        h[i] = _beginthread(threadfunc, 0,(LPVOID)i); //创建线程
    for (i = 0; i < 2; i++)
    {
        WaitForSingleObject((HANDLE)h[i], INFINITE); //等待线程结束
        CloseHandle((HANDLE)h[i]); //关闭线程对象句柄
    }
    CloseHandle(ghMutex); //关闭互斥对象句柄
    printf("卖票结束\n");
    return 0;
}
```

（3）保存工程并运行，运行结果如图 3-20 所示。

图 3-20

【例 3.18】_beginthreadex 函数的简单示例

（1）新建一个控制台工程。

（2）在 Test.cpp 中输入如下代码：

```
#include "pch.h"
#include <tchar.h>
#include <windows.h>
#include <stdio.h>
#include <process.h>

unsigned gCounter;
unsigned __stdcall ThreadFunc(void* pArguments)
{
    while (gCounter < 500000) //不断循环累加
        gCounter++;
    printf("子线程运行结果:%d\n", gCounter);
    return 0;
}

int _tmain(int argc, _TCHAR* argv[])
{
```

```
    HANDLE hThread;
    unsigned threadID;

    //创建一个子线程
    hThread = (HANDLE)_beginthreadex(NULL, 0, &ThreadFunc, NULL, 0, &threadID);
    WaitForSingleObject(hThread, INFINITE); //等待子线程结束
    printf("子线程运行结果应该是 500000;实际结果是%d\n", gCounter); //打印结果
    CloseHandle(hThread); //关闭线程句柄，销毁线程对象
    return 0;
}
```

_beginthreadex 创建的线程可以使用 WaitForSingleObject 函数来等待子线程句柄 hThread 的方式判断子线程释放结束，并且要显式地关闭子线程句柄。

（3）保存工程并运行，运行结果如图 3-21 所示。

图 3-21

3.4 MFC 多线程开发

前面纯粹使用 Win32 API 函数进行多线程开发，现在我们利用 MFC 库来进行多线程开发。MFC 对多线程的支持是通过对多线程开发相关的 Win32 API 进行简单的封装后实现的。

在 MFC 中，用类 CWinThread 的对象来表示一个线程，比如每个 MFC 程序的主线程都有一个继承自 CWinApp 的应用程序类，而 CWinApp 继承自 CWinThread。类 CWinThread 支持两种线程类型：工作者线程和用户界面线程。工作者线程没有收发消息的功能，通常用于后台计算工作，比如耗时的计算过程、打印机的后台打印等；用户界面线程具有消息队列和消息循环，可以收发消息，一般用于处理独立于其他线程执行之外的用户输入，响应用户及系统所产生的事件和消息等。

类 CWinThread 的成员中不但包含了控制线程的相关成员函数（比如暂停和恢复），而且包括线程的 ID 和句柄，主要成员可以见表 3-3。

表 3-3　类 CWinThread 的成员

类 CWinThread 的成员	含义
m_bAutoDelete	指定线程结束时是否要销毁 CWinThread 对象
m_hThread	当前线程的句柄
m_nThreadID	当前线程的 ID

（续表）

类 CWinThread 的成员	含义
m_pMainWnd	保存指向应用程序的主窗口的指针
m_pActiveWnd	指向容器应用程序的主窗口，当一个 OLE 服务器被现场激活时
CWinThread	构造一个 CWinThread 对象
CreateThread	创建线程
GetMainWnd	查询指向线程主窗口的指针
GetThreadPriority	获取当前线程的优先级
PostThreadMessage	向其他 CWinThread 对象传递一条消息
ResumeThread	减少一个线程的挂起计数
SetThreadPriority	设置当前线程的优先级
SuspendThread	增加一个线程的挂起计数

3.4.1 线程的创建

在 MFC 中有两种方式可以创建线程：一种是调用 MFC 库中的全局函数 AfxBeginThread；另一种是先定义 CWinThread 对象，然后调用成员函数 CWinThread::CreateThread 来创建线程。

函数 AfxBeginThread 是 MFC 库中的全局函数，不是 Win32 API 函数，只能在 MFC 程序中使用。该函数创建并启动一个线程，有两种重载形式，分别用于创建工作者线程（辅助线程）和用户界面线程（UI 线程）。创建工作者线程的函数形式如下：

```
   CWinThread* AfxBeginThread(AFX_THREADPROC pfnThreadProc, LPVOID pParam,
     int nPriority = THREAD_PRIORITY_NORMAL,UINT nStackSize = 0, DWORD
dwCreateFlags = 0,LPSECURITY_ATTRIBUTES lpSecurityAttrs = NULL );
```

其中，pfnThreadProc 为工作线程的线程函数地址。工作线程的线程函数形式如下：

```
   UINT __cdecl MyFunction( LPVOID pParam );
```

需要注意的是，该线程函数的返回值类型是 UINT，并且函数调用约定为__cdecl，而不是 WINAPI，前面 CreateThread 创建的线程函数的返回值类型为 DWORD，调用约定为 WINAPI，即__stdcall。其中，pParam 为传给线程函数的参数；nPriority 为线程的优先级，如果为 0，即宏 THREAD_PRIORITY_NORMAL，则线程与其父线程具有相同的优先级；nStackSize 表示线程为自己分配的堆栈的大小，其单位为字节，如果该参数为 0，则线程的堆栈被设置成与父线程堆栈相同大小；dwCreateFlags 用来确定线程在创建后释放立即开始执行，如果为 0 则线程在创建后立即执行，如果为 CREATE_SUSPEND，则线程在创建后立刻被挂起；lpSecurityAttrs 表示线程的安全属性指针，一般为 NULL。当函数成功时返回 CWinThread 对象的指针，如果失败就返回 NULL。

用户界面线程也可以用 AfxBeginThread 创建，注意不同的是第一个参数。创建用户界面线程的 AfxBeginThread 函数形式如下：

```
   CWinThread* AfxBeginThread(CRuntimeClass* pThreadClass,
   int nPriority = THREAD_PRIORITY_NORMAL,UINT nStackSize = 0,DWORD dwCreateFlags
```

```
= 0,LPSECURITY_ATTRIBUTES lpSecurityAttrs = NULL );
```

其中，参数 pThreadClass 指向从 CWinThread 派生的子类对象的 RUNTIME_CLASS，RUNTIME_CLASS 可以从一个 C++类名获得运行时的类结构；其他参数和函数返回值与前面介绍的相同，不再赘述。

用户界面线程通常用于处理用户输入和响应用户事件，这些行为独立于该应用程序的其他线程。用户界面线程必须包含有消息循环，以便可以处理用户消息。创建用户界面线程时，必须首先从 CWinThread 派生类，而且必须要重写类的 InitInstance 函数。

实际上，AfxBeginThread 内部会先新建一个 CWinThread 对象，然后调用 CWinThread::CreateThread 来创建线程，最后 AfxBeginThread 会返回这个 CWinThread 对象，如果我们没有把 CWinThread::m_bAutoDelete 设为 FALSE，则当线程函数返回的时候会自动删除这个 CWinThread 对象。因此，注意不要等线程结束的时候去关闭线程句柄，因为此时可能 CWinThread 对象已经销毁了，根本无法引用其成员变量 m_hThread（线程句柄）了。比如：

```
CWinThread *pwinthread1
pwinthread1 = AfxBeginThread(threadfunc, (LPVOID)0);
WaitForSingleObject(pwinthread1->m_hThread, INFINITE); //等待线程结束
CloseHandle(pwinthread1->m_hThread); //可能已经是无效指针
```

该段代码在单步调试的时候会报异常错误，因为最后一句中的 pwinthread1 很可能是无效的。既然删除了 CWinThread 对象，那么我们就不必去关闭线程句柄了。

此外，如果我们把 CWinThread::m_bAutoDelete 设为 TRUE，那么最后要自己去删除 CWinThread 对象（比如 delete pwinthread1;），否则会造成内存泄漏。

CWinThread::CreateThread 内部是通过 _beginthreadex 函数来创建线程的。只不过 AfxBeginThread 和 CWinThread::CreateThread 做了更多的初始化和检查工作。在 AfxBeginThread 创建的线程中使用 CRT 库函数是安全的。

下面我们创建一个用户界面线程，在用户界面线程中会创建一个窗口，并且点击窗口的时候会出现一个信息框。

【例 3.19】AfxBeginThread 创建用户界面线程

（1）新建一个单文档工程。

（2）切换到类视图，添加一个 MFC 类 CMyThread（继承于 CWinThread），作以为用户界面类；然添加一个 MFC 类 CMyWnd（继承于 CFrameWnd），用于在界面线程中创建窗口。

（3）打开 MyWnd.h，把 CMyWnd 构造函数的访问属性改为 public，同时添加一个进度条变量：

```
public:
 CProgressCtrl   m_pos; //进度条控件变量
 CMyWnd(); //构造函数
```

为 CMyWnd 添加 WM_CREATE 的消息处理函数 OnCreate。在该函数中我们创建一个进度条控件并设置计时器，代码如下：

```
int CMyWnd::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
 int i;
 if (CFrameWnd::OnCreate(lpCreateStruct) == -1)
     return -1;

 // TODO:  在此添加您专用的创建代码
 //创建进度条
 m_pos.Create(WS_CHILD | WS_VISIBLE, CRect(10, 10, 300, 50), this, 10001);
 m_pos.SetRange(0, 100); //设置范围
 m_pos.SetStep(1); //设置步长
 SetTimer(1, 50, NULL); //开启计时器，时间间隔为 50ms
 return 0;
}
```

为 CMyWnd 添加计时器消息 WM_TIMER 的消息处理函数 OnTimer，在其中我们让计时器向前走一步，代码如下：

```
void CMyWnd::OnTimer(UINT_PTR nIDEvent)
{
 // TODO:  在此添加消息处理程序代码和/或调用默认值
 m_pos.StepIt(); //进度条向前走一步
 CFrameWnd::OnTimer(nIDEvent);
}
```

最后为 CMyWnd 添加窗口销毁消息 WM_DESTROY 的消息处理函数 OnDestroy，在其中我们销毁计时器，代码如下：

```
void CMyWnd::OnDestroy()
{
 CFrameWnd::OnDestroy();
 // TODO:  在此处添加消息处理程序代码
 KillTimer(1); //销毁计时器
}
```

好了，我们在线程中创建的窗口完成了，该窗口运行的时候会不停让进度条往前滚动。

（4）打开 MyThread.cpp，找到函数 CMyThread::InitInstance，我们在其中添加创建上述窗口的代码：

```
BOOL CMyThread::InitInstance()
{
 // TODO:    在此执行任意逐线程初始化
    CMyWnd *pFrameWnd = new  CMyWnd(); //分配空间
    pFrameWnd->Create(NULL, _T("线程中创建的窗口" )); //创建窗口
    pFrameWnd->ShowWindow(SW_SHOW);  //显示窗口
    pFrameWnd->UpdateWindow();

    return  TRUE;
}
```

虽然我们用 new 分配了一个窗口的堆空间，但是不要用 delete 去删除它，因为在窗口销毁的时候，系统会自动删除这个 C++对象。最后在该文件开头包含头文件 MyWnd.h。

（5）切换到资源视图，打开菜单设计器，然后在“视图”菜单下添加一个菜单项“创建用户界面线程”，并为其添加视图类 CTestView 的事件处理函数，在其中我们将开启一个界面线程，代码如下：

```
void CTestView::On32771()
{
 // TODO:  在此添加命令处理程序代码
 AfxBeginThread(RUNTIME_CLASS(CMyThread)); //创建界面线程
}
```

类 CMyThread 就是我们上面创建的界面线程类，最后在文件开头包含头文件 MyThread.h。

（6）保存工程并运行，运行结果如图 3-22 所示。因为这两个窗口是在不同线程中创建的，所以在任务栏里会出现这两个窗口，它们是相互独立的。需要注意的是，如果直接关闭主线程中的窗口，就会导致子线程直接关闭，子线程窗口的销毁动作得不到执行（大家可以 CMyWnd::OnDestroy 中显示一个信息框来验证），从而造成内存泄漏，所以应该先关闭子线程窗口再关闭主线程窗口。

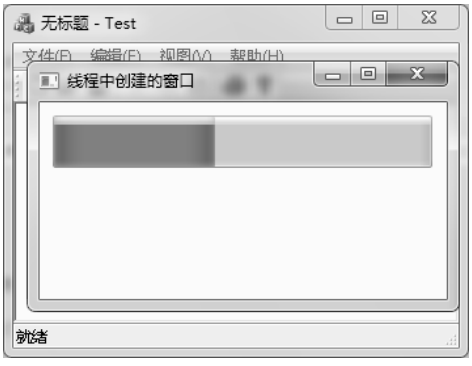


图 3-22

3.4.2　线程同步

我们知道，线程同步可以通过同步对象来实现，前面章节介绍了直接用 Win32 API 进行线程同步。在 MFC 中，对同步对象进行了 C++封装，各个同步函数称为了 C++类的成员函数。在 MFC 中，用于线程同步的类有 CCriticalSection（临界区类）、互斥类（CMutex）、事件类（CEvent）和信号量类（CSemaphore），这些类都从同步对象类 CSyncObject 派生。我们来看一下类 CSyncObject 在 afxmt.h 中的定义：

```
class CSyncObject : public CObject
{
 DECLARE DYNAMIC(CSyncObject)

// Constructor
public:
 explicit CSyncObject(LPCTSTR pstrName);

// Attributes
```

```
public:
 operator HANDLE() const;
 HANDLE  m hObject;

// Operations
 virtual BOOL Lock(DWORD dwTimeout = INFINITE);
 virtual BOOL Unlock() = 0;
 virtual BOOL Unlock(LONG /* lCount */, LPLONG /* lpPrevCount=NULL */)
     { return TRUE; }

// Implementation
public:
 virtual ~CSyncObject();
#ifdef _DEBUG
 CString m strName;
 virtual void AssertValid() const;
 virtual void Dump(CDumpContext& dc) const;
#endif
 friend class CSingleLock;
 friend class CMultiLock;
};
```

其中，m_hObject 存放同步对象的句柄。函数 Lock 用于锁定某个同步对象，它在内部只是简单地调用等待函数 WaitForSingleObject。Unlock 是一个纯虚函数，因此类 CSyncObject 是一个纯虚类，所以该类不应该直接用在程序中，而应该使用它的子类。另外，在末尾有两个友元类 CSingleLock 和 CMultiLock，这两个类没有父类也没有子类，主要用于对共享资源的访问控制。要使用 4 大同步类（CCriticalSection、CMutex、CEvent、 CSemaphore）来同步线程，必须要使用 CSingleLock 或 CMultiLock 来等待或释放同步对象。当一次只需要等待一个同步对象时，使用类 CSingleLock；当一次要等待多个同步对象时，使用类 CMultiLock。

类 CSingleLock 的常见成员见表 3-4。

表 3-4　类 CSingleLock 的常见成员

类 CSingleLock 的常见成员	含义
CSingleLock	构造一个 CSingleLock 对象
IsLocked	判断同步对象释放处于锁定状态
Lock	对同步对象上锁，即等待某个同步对象
UnLock	释放某个同步对象，即解锁

（1）构造函数 CSingleLock 的声明如下：

```
CSingleLock( CSyncObject* pObject, BOOL bInitialLock = FALSE );
```

其中，参数 pObject 为指向同步对象的指针，不可以为 NULL；bInitialLock 表明该同步对象在初始的时候是否锁定同步对象。

（2）函数 Lock 的声明如下：

```
BOOL Lock(DWORD dwTimeOut = INFINITE );
```

其中，参数 dwTimeOut 为等待同步对象变为可用（有信号状态）所用的时间，单位是毫

秒，如果为 INFINITE，则函数一直等到同步对象有信号为止。如果函数成功就返回非零，否则返回零。

通常当同步对象变为有信号时，Lock 函数将成功返回，同时线程将拥有该同步对象。如果同步对象处于无信号状态（不可用），那么 Lock 等待 dwTimeOut 毫秒或一直等下去，直到同步对象有信号。等待 dwTimeOut 毫秒时，若等待超时，则 Lock 返回零。

（3）函数 Unlock 用于释放某个同步对象，声明如下：

```
BOOL Unlock();
```

如何函数成功就返回非零，否则返回零。

（4）函数 IsLocked 判断同步对象释放处于锁定状态，声明如下：

```
BOOL IsLocked( );
```

如果同步对象被锁定，函数返回非零，否则返回零。

在使用 CSingleLock 进行线程同步的时候，不要在多个线程中共享一个 CSingleLock 对象，通常在一个线程中定义一个对象。比如：

```
UINT threadfunc() //线程函数
{
// gCritSection 是类 CCriticalSection 的全局对象
CSingleLock singleLock(&gCritSection);
singleLock.Lock();  // 试图对共享资源进行上锁
if (singleLock.IsLocked())  // 判断资源释放上锁
{
 //
     使用共享资源
     //
    singleLock.Unlock(); //使用完毕后解锁
}
}
```

3.4.2.1　临界区类

类 CCriticalSection 对临界区对象的操作进行了 C++封装。关于临界区的概念前面已经介绍过，这里不再赘述。类 CCriticalSection 的常见成员函数见表 3-5。

表 3-5　CCriticalSection 的常见成员函数

类 CCriticalSection 的常见成员	含义
m_sect	结构体 CRITICAL_SECTION 类型的变量
Lock	用于获得临界区对象的访问权
UnLock	释放临界区对象

类 CCriticalSection 的用法有两种：一种是单独使用，另一种是和 CSingleLock 或 CMultiLock 联合使用。

- 单独使用 CCriticalSection 时，首先创建一个 CCriticalSection 对象，然后在需要访问

临界区时先调用 CCriticalSection::Lock 函数进行锁定，即获得临界区对象的访问权，然后开始执行临界区代码，在执行完临界区后再调用 CCriticalSection:: UnLock 函数释放临界区对象。

- 第二种方法先定义一个 CSingleLock 对象，并把 CCriticalSection 对象的指针作为参数传入其构造函数。然后在需要访问临界区的地方调用函数 CSingleLock::Lock，用完临界区后再调用函数 CSingleLock:: Unlock，比如：

```
UINT threadfunc()
{
// m_CritSection 是类 CCriticalSection 的对象
CSingleLock singleLock(&m_CritSection);
singleLock.Lock();  // 试图对共享资源进行上锁
if (singleLock.IsLocked())  // 判断资源释放上锁
{
 //
    使用共享资源
 //
    singleLock.Unlock(); //使用完毕后解锁
}
}
```

下面我们来演示一下这两种用法。同前面 Win32 API 线程同步一样，我们也来对例 3.11 进行改造。

【例 3.20】单独使用 CCriticalSection 对象来同步线程

（1）新建一个控制台工程，并在向导的“应用程序设置”界面中勾选“MFC”复选框，这是因为 CCriticalSection 属于 MFC 类，如图 3-23 所示。

图 3-23

（2）在 Test.cpp 中输入如下代码：

```
// Test.cpp : 定义控制台应用程序的入口点
#include "stdafx.h"
#include "Test.h"
#include "afxmt.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#endif

// 唯一的应用程序对象
CWinApp theApp;
using namespace std;
int gticketId = 10;  //记录卖出的车票号
CCriticalSection gcs; // 定义 CCriticalSection 对象

UINT  threadfunc(LPVOID param)
{
 TCHAR chWin;

 if (param == 0) chWin = _T('甲'); //甲窗口
 else chWin = _T('乙'); //乙窗口
 while (1)
 {
     gcs.Lock();
     if (gticketId <= 0) //如果车票全部卖出了，则退出循环
     {
         gcs.Unlock();
         break;
     }
     setlocale(LC_ALL, "chs"); //为控制台设置中文环境
     _tprintf(_T("%c 窗口卖出的车票号 = %d\n"), chWin, gticketId); //打印信息
     gticketId--;//车票减少一张
     gcs.Unlock(); //释放临界区对象所有权
     Sleep(1); //让出 CPU 让其他线程有机会执行
 }
 return 0;
}

int _tmain(int argc, TCHAR* argv[], TCHAR* envp[])
{
 int nRetCode = 0;
 CWinThread *pwinthread1, *pwinthread2;
 HMODULE hModule = ::GetModuleHandle(NULL);

 if (hModule != NULL)
 {
     // 初始化 MFC 并在失败时显示错误
     if (!AfxWinInit(hModule, NULL, ::GetCommandLine(), 0))
     {
```

```
            // TODO:  更改错误代码以符合您的需要
            _tprintf(_T("错误:  MFC 初始化失败\n"));
            nRetCode = 1;
        }
        else
        {
            // TODO:  在此处为应用程序的行为编写代码
            puts("利用 CCriticalSection 同步线程");
            //创建第一个卖票线程
            pwinthread1 = AfxBeginThread(threadfunc, (LPVOID)0);
            //创建第二个卖票线程
            pwinthread2 = AfxBeginThread(threadfunc, (LPVOID)1);
            WaitForSingleObject(pwinthread1->m_hThread, INFINITE); //等待线程结束
            //等待线程结束
            WaitForSingleObject((HANDLE)pwinthread2->m_hThread, INFINITE);
            puts("卖票结束");
        }
    }
    else
    {
        // TODO:  更改错误代码以符合您的需要
        _tprintf(_T("错误:  GetModuleHandle 失败\n"));
        nRetCode = 1;
    }

    return nRetCode;
}
```

程序很简单，首先创建两个工作线程，然后主线程就等待它们执行完毕。在线程函数中，每当要卖票了，就先 Lock，卖完票后再 Unlock。

（3）保存工程并运行，运行结果如图 3-24 所示。

图 3-24

【例 3.21】联合使用类 CCriticalSection 和类 CSingleLock 来同步线程

（1）新建一个控制台工程，并在向导的“应用程序设置”界面中勾选“MFC”复选框。

（2）打开 Test.cpp，在其中输入如下代码：

```
#include "stdafx.h"
#include "Test.h"
#include "afxmt.h" //线程同步类所需的头文件
#ifdef _DEBUG
#define new DEBUG_NEW
#endif

// 唯一的应用程序对象
CWinApp theApp;
using namespace std;
int gticketId = 10;  //记录卖出的车票号
CCriticalSection gcs; // 定义 CCriticalSection 对象

UINT  threadfunc(LPVOID param)
{
 TCHAR chWin;

 if (param == 0) chWin = _T('甲'); //甲窗口
 else chWin = _T('乙'); //乙窗口

 CSingleLock singleLock(&gcs); //定义一个单锁对象，参数为 CCriticalSection 对象地址
 while (1)
 {
     singleLock.Lock(); //上锁
     if (gticketId <= 0) //如果车票全部卖出了，则退出循环
     {
         singleLock.Unlock();
         break;
     }
     setlocale(LC_ALL, "chs"); //为控制台设置中文环境
     _tprintf(_T("%c 窗口卖出的车票号 = %d\n"), chWin, gticketId); //打印信息
     gticketId--;//车票减少一张
     singleLock.Unlock(); //解锁
     Sleep(1); //让出 CPU，让其他线程有机会执行
 }
 return 0;
}
int _tmain(int argc, TCHAR* argv[], TCHAR* envp[])
{
 int nRetCode = 0;
 CWinThread *pwinthread1, *pwinthread2;
 HMODULE hModule = ::GetModuleHandle(NULL);

 if (hModule != NULL)
 {
     // 初始化 MFC 并在失败时显示错误
     if (!AfxWinInit(hModule, NULL, ::GetCommandLine(), 0))
     {
         // TODO:  更改错误代码以符合您的需要
         _tprintf(_T("错误:  MFC 初始化失败\n"));
         nRetCode = 1;
```

```
        }
        else
        {
            // TODO:  在此处为应用程序的行为编写代码
            puts("联合使用类 CCriticalSection 和类 CSingleLock 来同步线程");
            pwinthread1 = AfxBeginThread(threadfunc, (LPVOID)0);
            pwinthread2 = AfxBeginThread(threadfunc, (LPVOID)1);
            WaitForSingleObject(pwinthread1->m_hThread, INFINITE); //等待线程结束
            //等待线程结束
            WaitForSingleObject((HANDLE)pwinthread2->m_hThread, INFINITE);
            puts("卖票结束");
        }
    }
    else
    {
        // TODO:  更改错误代码以符合您的需要
        _tprintf(_T("错误:  GetModuleHandle 失败\n"));
        nRetCode = 1;
    }

    return nRetCode;
}
```

上述代码通过定义 CSingleLock 局部对象来同步两个线程，也可以定义两个全局的 CSingleLock 对象，然后根据不同的线程分别使用不同的全局对象，比如线程函数也可以这样写：

```
CCriticalSection gcs; // 定义 CCriticalSection 对象
CSingleLock singleLock(&gcs);
CSingleLock singleLock2(&gcs);
UINT  threadfunc(LPVOID param)
{
 TCHAR chWin;

 if (param == 0) chWin = _T('甲'); //甲窗口
 else chWin = _T('乙'); //乙窗口
 while (1)
 {
     if (param==0) singleLock.Lock();
     else singleLock2.Lock();
     if (gticketId <= 0) //如果车票全部卖出了，则退出循环
     {
         if (param == 0)  singleLock.Unlock();
         else singleLock2.Unlock();
         break;
     }
     setlocale(LC_ALL, "chs"); //为控制台设置中文环境
     _tprintf(_T("%c 窗口卖出的车票号 = %d\n"), chWin, gticketId); //打印信息
     gticketId--;//车票减少一张
     if (param == 0) singleLock.Unlock();
```

```
        else singleLock2.Unlock();
        Sleep(1);
    }

    return 0;
}
```

两种方式的运行效果相同，但明显第二种方式啰唆了，通常用第一种方式即可。

（3）保存工程并运行，运行结果如图 3-25 所示。

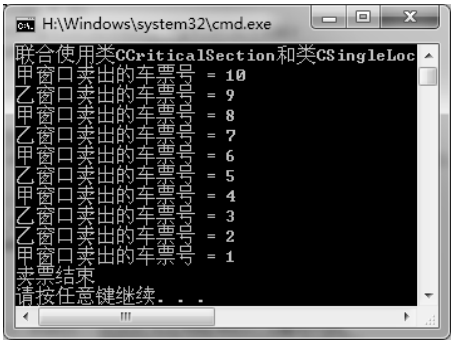


图 3-25

3.4.2.2　互斥类

MFC 中的互斥类 CMutex 封装了利用互斥对象来进行线程同步的操作。互斥类不但能同步一个进程中的线程，还能同步不同进程之间的线程。该类是 CSyncObject 的子类，继承了 Lock 函数并重载了 Unlock 函数，利用这两个函数实现线程同步。

要用互斥类来同步线程也有两种使用方式：一种是 CMutex 类单独使用，另一种是联合 CSingleLock 或 CMultiLock 类一起使用。当在单独使用的时候，先定义一个 CMutex 对象，然后调用该类的 Lock 函数来等待互斥对象的所有权，如果等到就开始访问共享资源，访问完毕后再调用该类的 Unlock 函数来释放互斥对象的所有权。

实际上，CMutex 类只是简单地对 Win32 API 的互斥操作函数进行了封装。比如，CMutex 的构造函数中会调用 API 函数 CreateMutex 来创建互斥对象并判断释放创建成功。如果要等待互斥对象的所有权，就调用其父类 CSyncObject 的 Lock 函数，而 CSyncObject::Lock 中调用了 WaitForSingleObject。该类的 Unlock 函数重载了父类的 Unlock，它的实现如下：

```
BOOL CMutex::Unlock()
{
    return ::ReleaseMutex(m_hObject);
}
```

实际只是简单地调用了 API 函数 ReleaseMutex 来释放互斥对象的所有权。

下面我们单独使用 CMutex 类来改写例 3.11，增加线程的同步功能。

【例 3.22】单独使用 CMutex 类实现线程同步

（1）新建一个控制台工程，并在向导的“应用程序设置”界面中勾选“MFC”复选框。

（2）打开 Test.cpp，在其中输入如下代码：

```
#include "stdafx.h"
#include "Test.h"
#include "afxmt.h"//线程同步类所需的头文件
#ifdef _DEBUG
#define new DEBUG_NEW
#endif

int gticketId = 10;  //记录卖出的车票号
CMutex gmux; // 定义 CMutex 对象

UINT  threadfunc(LPVOID param)
{
 TCHAR chWin;

 if (param == 0) chWin = _T('甲'); //甲窗口
 else chWin = _T('乙'); //乙窗口

 while (1)
 {
     gmux.Lock();
     if (gticketId <= 0) //如果车票全部卖出了，则退出循环
     {
         gmux.Unlock();
         break;
     }
     setlocale(LC_ALL, "chs"); //为控制台设置中文环境
     _tprintf(_T("%c 窗口卖出的车票号 = %d\n"), chWin, gticketId); //打印信息
     gticketId--;//车票减少一张
     gmux.Unlock(); //解锁
     Sleep(1); //让出 CPU，让其他线程有机会执行
 }
 return 0;
}
int _tmain(int argc, TCHAR* argv[], TCHAR* envp[])
{
 int nRetCode = 0;
 CWinThread *pwinthread1, *pwinthread2;
 HMODULE hModule = ::GetModuleHandle(NULL);

 if (hModule != NULL)
 {
     // 初始化 MFC 并在失败时显示错误
     if (!AfxWinInit(hModule, NULL, ::GetCommandLine(), 0))
     {
         // TODO:  更改错误代码以符合您的需要
```

```
            _tprintf(_T("错误: MFC 初始化失败\n"));
            nRetCode = 1;
        }
        else
        {
            // TODO: 在此处为应用程序的行为编写代码
            puts("单独使用类 CMutex 来同步线程");
            pwinthread1 = AfxBeginThread(threadfunc, (LPVOID)0);
            pwinthread2 = AfxBeginThread(threadfunc, (LPVOID)1);
            WaitForSingleObject(pwinthread1->m_hThread, INFINITE); //等待线程结束
            //等待线程结束
            WaitForSingleObject((HANDLE)pwinthread2->m_hThread, INFINITE);
            puts("卖票结束");
        }
    }
    else
    {
        // TODO: 更改错误代码以符合您的需要
        _tprintf(_T("错误: GetModuleHandle 失败\n"));
        nRetCode = 1;
    }

    return nRetCode;
}
```

（3）保存工程并运行，运行结果如图 3-26 所示。

图 3-26

3.4.2.3　事件类

MFC 中的事件类 CEvent 封装了利用事件对象来进行线程同步的操作。关于事件对象概念前面我们已经介绍过了，这里不再赘述。其实该类也是对 Win32 API 事件对象操作进行简单封装。它的常用成员函数如表 3-6。

表 3-6　CEvent 的常用成员函数

CEvent 的常用成员函数	含义
CEvent	构造一个 CEvent 对象
SetEvent	设置事件有信号（可用）
ResetEvent	设置事件无信号（不可用）

如果要等待事件对象变为可用，可以直接使用 API 函数 WaitForSingleObject 或者调用其父类的 Lock 函数（内部也是调用 WaitForSingleObject）。

事件类同步线程也有两种方式：一种是单独使用，另一种是联合 CSingleLock 或 CMultiLock 来使用。

下面我们用事件类为例 3.11 增加线程同步功能。

【例 3.23】单独使用类 CEvent 实现线程同步

（1）新建一个控制台工程，并在向导的“应用程序设置”界面中勾选“MFC”复选框。

（2）打开 Test.cpp，在其中输入如下代码：

```
#include "stdafx.h"
#include "Test.h"
#include "afxmt.h"//线程同步类所需的头文件
#ifdef  DEBUG
#define new DEBUG_NEW
#endif

int gticketId = 10;  //记录卖出的车票号
CEvent gEvent; // 定义 CEvent 对象

UINT  threadfunc(LPVOID param)
{
 TCHAR chWin;

 if (param == 0) chWin = _T('甲'); //甲窗口
 else chWin = _T('乙'); //乙窗口

 while (1)
 {
     gEvent.Lock();
     if (gticketId <= 0) //如果车票全部卖出了，则退出循环
     {
         gEvent.SetEvent();
         break;
     }
     setlocale(LC_ALL, "chs"); //为控制台设置中文环境
     _tprintf(_T("%c 窗口卖出的车票号 = %d\n"), chWin, gticketId); //打印信息
     gticketId--;//车票减少一张
     gEvent.SetEvent();
     //Sleep(1); //让出 CPU，让其他线程有机会执行
 }
 return 0;
```

```
}
int _tmain(int argc, TCHAR* argv[], TCHAR* envp[])
{
 int nRetCode = 0;
 CWinThread *pwinthread1, *pwinthread2;
 HMODULE hModule = ::GetModuleHandle(NULL);

 if (hModule != NULL)
 {
     // 初始化 MFC 并在失败时显示错误
     if (!AfxWinInit(hModule, NULL, ::GetCommandLine(), 0))
     {
         // TODO:  更改错误代码以符合您的需要
         _tprintf(_T("错误:  MFC 初始化失败\n"));
         nRetCode = 1;
     }
     else
     {
         // TODO:  在此处为应用程序的行为编写代码
         puts("单独使用类 CMutex 来同步线程");
         gEvent.SetEvent(); //设置事件处于有信号状态
         pwinthread1 = AfxBeginThread(threadfunc, (LPVOID)0);
         pwinthread2 = AfxBeginThread(threadfunc, (LPVOID)1);
         WaitForSingleObject(pwinthread1->m hThread, INFINITE); //等待线程结束
         WaitForSingleObject(pwinthread2->m_hThread, INFINITE); //等待线程结束
         puts("卖票结束");
     }
 }
 else
 {
     // TODO:  更改错误代码以符合您的需要
     _tprintf(_T("错误:  GetModuleHandle 失败\n"));
     nRetCode = 1;
 }

 return nRetCode;
}
```

（3）保存工程并运行，运行结果如图 3-27 所示。

图 3-27

3.4.2.4 信号量类

MFC 中的信号量类 CSemaphore 封装了利用信号量对象来进行线程同步的操作。关于信号量概念前面我们已经介绍过了，这里不再赘述。类 CSemaphore 在构造函数中调用了 CreateSemaphore 函数来创建信号量对象，并重载了父类的 UnLock 函数，里面调用了 ReleaseSemaphore 函数。说到底，也是对 Win32 API 的信号量对象操作进行了简单封装。等待信号量对象有信号可以用 API 函数 WaitForSingleObject 或者调用其父类的 Lock 函数（内部也是调用 WaitForSingleObject）。

信号量类同步线程也有两种方式：一种是单独使用，另一种是联合 CSingleLock 或 CMultiLock 来使用。

下面我们用信号量类为例 3.11 增加线程同步功能。

【例 3.24】单独使用类 CSemaphore 实现线程同步

（1）新建一个控制台工程，并在向导的“应用程序设置”界面中勾选“MFC”复选框。

（2）打开 Test.cpp，在其中输入如下代码：

```
#include "stdafx.h"
#include "Test.h"
#include "afxmt.h"//线程同步类所需的头文件
#ifdef _DEBUG
#define new DEBUG_NEW
#endif

int gticketId = 10;  //记录卖出的车票号
CSemaphore gSp( 1, 50, _T("mySemaphore")); // 定义 CSemaphore 对象

UINT  threadfunc(LPVOID param)
{
 TCHAR chWin;

 if (param == 0) chWin = _T('甲'); //甲窗口
 else chWin = _T('乙'); //乙窗口

 while (1)
 {
     gSp.Lock();
     if (gticketId <= 0) //如果车票全部卖出了，则退出循环
     {
         gSp.Unlock();
         break;
     }
     setlocale(LC_ALL, "chs"); //为控制台设置中文环境
     _tprintf(_T("%c 窗口卖出的车票号 = %d\n"), chWin, gticketId); //打印信息
     gticketId--;//车票减少一张
     gSp.Unlock();
 }
 return 0;
```

```
}
int _tmain(int argc, TCHAR* argv[], TCHAR* envp[])
{
 int nRetCode = 0;
 CWinThread *pwinthread1, *pwinthread2;
 HMODULE hModule = ::GetModuleHandle(NULL);

 if (hModule != NULL)
 {
     // 初始化 MFC 并在失败时显示错误
     if (!AfxWinInit(hModule, NULL, ::GetCommandLine(), 0))
     {
         // TODO:  更改错误代码以符合您的需要
         _tprintf(_T("错误:  MFC 初始化失败\n"));
         nRetCode = 1;
     }
     else
     {
         // TODO:  在此处为应用程序的行为编写代码
         puts("单独使用类 CSemaphore 来同步线程");
         pwinthread1 = AfxBeginThread(threadfunc, (LPVOID)0);
         pwinthread2 = AfxBeginThread(threadfunc, (LPVOID)1);
         WaitForSingleObject(pwinthread1->m_hThread, INFINITE); //等待线程结束
         WaitForSingleObject(pwinthread2->m_hThread, INFINITE); //等待线程结束
         puts("卖票结束");
     }
 }
 else
 {
     // TODO:  更改错误代码以符合您的需要
     _tprintf(_T("错误:  GetModuleHandle 失败\n"));
     nRetCode = 1;
 }

 return nRetCode;
}
```

（3）保存工程并运行，运行结果如图 3-28 所示。

图 3-28