1 编译器

1.1 编译器通用

1.1.1 宏

```
# c++11和ISO c99  支持
system("pause");          /* 阻塞/暂停功能                */
return EXIT_SUCCESS;      /* 返回正常的退出  0            */

f9a2d881-5ee5-11eb-8b02-813fedf98f76
```

1.1.2 const修饰/extern链接inline内联函数

```
const int vaule = 10;    //const全局变量(常量区不能直接或间接修改), 局部变量为符号表键值对不可以修改

'内部链接属性'
extern int a = 10;  // C++默认内部链接
//全局变量定义也是为默认的extern(表示已在其他源文件中定义,不用分配空间,直接使用其他源文件的a变量)
    extern int add(int a, int b);    //函数声明(已在其他源文件中声明定义,直接使用其他源文件的函数)
    static int b =10;         //静态 全局变量为内部链接只能在本文件使用
    # const分配内存请情况
    /* 对 const 修饰的变量取地址会临时分配内存将符号表的值放入内存,指针修改内存不影响符号表
extern
    修饰的可以链接外部的 const 变量自定义数据类型(struct结构体)普通变量用于初始化 const 修饰变量的值 */

#inline 内联函数          //代替宏定义/宏函数,避免宏定义缺陷(运算完整性),声明和实现都要加关键字inline
    // 内联函数没有函数入口和堆栈操作不能使用以下限制,和宏一样直接把源代码替换在调用内联函数的地方
    // 类函数默认是内联函数,操作不当会变成否则会变成普通函数,
    /* 不能存在任何形式的循环语句   不能存在过多的条件判断语句
        函数体不能过于庞大   不能对函数进行取操作址   */

#define cat(a,b) a##b  // 将两个字符变量名合并在一起 --  cat(a,b) -> a##b --> ab  //
```

1.2 VS编译器特有宏

```
    #预定义宏
    __FILE__              /* 宏所在文件的源文件名(路径)    */
    __LINE__              /* 宏所在行的行号                */
    __DATE__              /* 代码编译的日期                */
    __TIME__              /* 代码编译的的时间              */
    __func__              /* 宏所在的当前函数 */
    __cplusplus           /*  */

    __CHAR_UNSIGNED                 /* 如果char类型为无符号,该宏定义为1否则未定义 */
    __COUNTER__                     /* 从0开始,每次使用都会递增1 */
    _DEBUG                          /* 如果设置了/lDd/mDd/mTd改宏定义为1否则未定义*/
    __FUNCTION__                    /* 函数名称 不含修饰名 */
    __FUNCDNAME__                   /* 函数名称 包含修饰名 */
    __FUNCSIG__                     /* 包含了函数签名的函数名 */
    _WIN32                          /* 当编译为32位ARM,64位ARM,X86或64定义为1 否则未定义 */
    _WIN64                          /* 当编译为64位ARM或X64定义为1 否则未定义 */
```

```
    __TIMESTAMP__                   /* 最后一次源代码修改时间和日期 */
```

1.3 Dll

```
'dll导出'
    extern "C" _declspec(dllexport) int function(int x, int y);

'dll导入'
    extern "C"  __declspec(dllimport) __stdcall int function(int x, int y);
    extern "C"  __declspec(dllimport) int function(int x, int y);

' .def'
    EXPORTS
    function @12 NONAME
```

2 C / .c

2.1 标准头文件

```
#include<stdio.h>           /* 系统标准输入输出头文件 */
#include<stdlib.h>          /* 外部系统程序条用头文件 */
#include<time.h>            /* 获取当前系统时间 */
#include<string.h>          /* 字符串处理函数头文件 */
#include<system>            /* 调用系统函数 */
#include<sys/ty pes.h>      /* 获取文件属性头文件 */
#include<sys/stat.h>        /* 获取文件属性头文件 */
#include<stddef.h>          /* 获取偏移量属性头文件 */
#include<stdarg.h>
```

```
void __declspec(naked) func1()
{
    __asm
    {
        ......
    }
}
// VS引用汇编.asm文件

// Win32自定义生成工具
    命令行: ml /c /coff %(fileName).asm
    输出: %(fileName).obj;%(OutPuts)
64
    命令行: ml64 /Fo $(IntDir)%(fileName).obj /c %(fileName).asm
   输出: $(IntDir)%(fileName).obj;%(Outputs)
```

```
'调用约定'
    __cdecl                 从右至左入栈                                调用者清理栈
    __stdcall               从右至左入栈                                自身清理堆栈
    __fastcall              ECX/EDX传送前两个,剩下:从右至左入栈          自身清理堆栈
```

2.2 数据类型

2.2.1 基础数据类型

> sizeof(变量名)　计算数据类型在内存中的大小(Byte)

```
char^1          /* 字符型(1字节) */
short int^2  <=  int^4  <=  long int^4  <=  long long int^8
/* 短整型(2字节) <= 整型(4字节) <= 长整型(4字节)  <= 长长整形(8字节) */
float^4  <=  double^8
/* 单精度浮点型(4字节)7位有效数值  <=  双精度浮点型(8字节)15~16位有效数字 */
```

```
char            1字节     -128 ~ 127 (-2^7 ~ 2^7-1)
unsigend char   1字节     0 ~255 (0 ~2^5-1)
short           2字节     -32768 ~ 32767 (-2^15 ~ 2^15-1)
unsigned short  2字节     0 ~ 65535 (0 ~ 2^16-1)
int             4字节     -2147483648 ~ 2147483647 (-2^31 ~ 2^31-1)
unsigned int    4字节     0 ~ 4294967295 (0 ~ 2^32-1)
long            4字节     -2147483648 ~ 2147483647 (-2^31 ~ 2^31-1)
unsigned long   4字节     0 ~ 4294967295 (0 ~ 2^32-1)
```

```
#原码补码
signed int a = 20       /* 有符号位(系统默认) */
unsigned int a =20      /* 无符号位          */
```

2.2.2 数据类型转换

```
"判断类型"
#include <ctype.h>
    bool a=isalnum("1");                /* 检查所传的字符是否是字母和数字如果是返回非0,
否则返回0 */
"字符串里的数字转换为int类型"
    int a=atoi("8888");             /* 将字符串8转换为数字的8*/
```

2.3.3 位运算

```
'位运算'
    <<          /* 左移运算符(在二进制里，向左位移几倍就是乘以2的N次方)，正数不零，负数补1
*/
    >>          /* 右移运算符(在二进制里，向右位移几倍就是乘以2的N次方)，正数不零，负数补1
*/
    ~           /* 求反运算符(在二进制里，改变符号位，所有的0变1，1变0) */
    &           /* 与运算(在二进制位里比较两个数，两个数同位中同时是1时结果才为1) */
    |           /* 或运算(在二进制位里比较两个数，两个数同位中有一个是1结果就为1) */
    ^           /* 异或运算(在二进制位里比较两个数，两个数同位相同结果就位0) */
```

2.3.4 extern/static关键字

```
#define NAME number;          /*  #define 常量名 值 */
const int vaule = 10;    // const 全局变量(常量区不能直接或间接修改), 局部变量(堆区可以间接修改)
extern int a = 10;            /* 全局变量定义也是为默认的extern , 默认外部链接 */

extern int add(int a, int b);    /* 函数声明 */
extern "C"{ int add();int sub();}    // 声明这个函数时C语言风格函数,c++调用时用C语言风格调用
extern "C"{ #include "ios.h"}    /* 头文件里的函数/变量按照C风格进行处理 */
static int b =10;         /* 静态 全局变量为内部链接只能在本文件使用(地址在函数代码段里面) */
```

2.3 普通语句

```
break;                /* 直接跳出当前循环层 */
continue              /* 结束本轮循环并开始下一轮循环 */
exit(1);              /* 正常退出 */
abort();              /* 非正常退出,弹出框 */
```

2.4 函数

```
    void __declspec(naked)  func(){ ..... }          /* 裸函数 */
```

2.4.1 标准输入输出函数

```
scanf("%d", &a);                  /* 标准输入函数,&取地址符号 */
scanf("%s, a[ ] ");           /* 输入字符串,遇到空格结束 */
printf("%d\n",a);                 /* 标准输出函数,打印 */
ptintf("%p\n", &a);               /* 以十六进制打印变量内存地址 */
' 打印格式: '
    %o          /* 将数据按照八进制输出         */
    %x          /* 将数据按照十六进制小写输出 */
    %X          /* 将数据按照十六进制大写输出 */
    %d          /* 将数据按照十进制类型输出      */
    %p          /* 以十六进制打印内存地址   */
    %s          /* 输出 字符数组         */
    %c          /* 输出 char 型              */
    %5.2f       /* 输出 float 类型 (小数点前面保留5位.2 小数点后面保留2位小数) */
    %lf         /* 输出 double 型 */
    %hd         /* 输出 short 类型 */
    %d          /* 输出 int 类型 */
    %l          /* 输出 long 类型 */
    %ll         /* 输出 long long 类型 */
    %hu         /* 输出 unsigend short 类型 */
    %u          /* 输出 unsigend int 类型 */
    %lu         /* 输出 unsigend long 类型 */
    %llu        /* 输出 unsigeng long long 类型 */
```

```
'无显示输入'
    #include <conio.h>
    int _getch()         /* 有的平台无法使用 */
```

2.4.2数学函数

2.4.2.1 取绝对值

```
#include <math.h>
    abs(a-b);                   /* 取整数的绝对值  */
    fabs(a);                    /* 取浮点数的绝对值 */
    sqrt(a+b);                  /* 开平方 */
    pow(x,y)                    /* x 和 y 是浮点数 */
```

2.4.2.2 随机数

```
#include <time.h>                   /* 引入随机数头文件 */
    srand((unsigned int)time(NULL));    /* 置随机数种子(强转无符号整型) */
    int a = rand();                     /* 生成随机数,并定义a接受 */
```

2.4.3 获取操作系统时间函数

```
    struct tm {
        int tm_sec;     /* seconds after the minute - [0,59] 秒数 */
        int tm_min;     /* minutes after the hour - [0,59] 分 */
        int tm_hour;    /* hours since midnight - [0,23]  时 */
        int tm_mday;    /* day of the month - [1,31]  日 */
        int tm_mon;     /* months since January - [0,11] 月 */
        int tm_year;    /* years since 1900 年 */
        int tm_wday;    /* days since Sunday - [0,6] 周一至周日 */
        int tm_yday;    /* days since January 1 - [0,365] */
        int tm_isdst;   /* daylight savings time flag */
        };

    time(time_t*);              /* 返回从1970-1-1到现在的秒数,传出参数存放返回秒数的地址
*/
    localtime(time_t*);         /* 将秒数转换为时分秒的结构体,返回值是结构体*/
例:
        time_t t;                   // time_t [long long int]
        time(&t);
        struct tm* timeInfo=localtime(&t);
        printf("%d\n",timeInfo->tm_sec);
```

2.4.4 字符串处理函数

```
#include <string.h>
getchar();      // 从键盘接收一个字符并显示出来(按回车结束阻塞,但是只接收第一个字符)
getch();        // 从键盘接收一个字符但不显示出来
getche();       // 有返回的显示(接收一个字符停止阻塞)
gets(string);
            /* 接收字符串并保存在变量里字符串可以带空格遇到换行符或结束标志结束. /getchar强
化 */
puts(string);       /* 获取输出字符串变量,输出完后自动\n换行. /printf强化 */
fgets(string, sizeof, stdin);   /* 从指定的文件内读入字符,直到出现换行结尾或 sizeof -1结
束最后一个加上\0结束<字符串, 指定最大读取长度, 文件指针> */
fputs(string, stdout);      /* 将指定的字符串写入到指定的文件中遇到\0,字符串结束符不写入
文件 */
scanf("%[^\n]", string)     /* 输入字符串,遇到\n结束,适用于字符串中带空格 */
```

```
sscanf(string1, "%s %d", string2, &a);     //将 1 中的数据读取出来,并转换后放入 2 和 a
中 */
    %*s 或者 %*d              /* 跳过数据(字符串,整型) */
    %[width]s                 /* 读取指定宽度(长度)的数据 */
    %[a-z]                    /* 匹配 a 到 z 中的任意字符 */
    %[aBc]                    /* 匹配 a B c 中的一员 */
    %[^a]                     /* 匹配非 a 的任意字符 */
    %[^a-z]                   /* 读取除 a-z 以外的所有字符 */
printf(%s, string);     /* 输出自字符串 */
sprintf(string1, "%s %d", strint2, string3);    /* 将 3 和 2 的数据格式化后放入 1 中
*/
strlen(string);                 /* 字符串的有效长度遇到\0计\0前面的 */
strcpy(string1, string2);       /* 将源字符串2拷贝进目标字符串1中 */
strncpy(string1, string2, number);  /* 拷贝相应数量的源字符串2放入目标字符串1中,无结束标
志 \0 */
    string1[number] = '\0';     /* 手动添加结束标志否则数据溢出出现乱码 */
strcat(string1, string2);       /* 将字符串2追加到字符串1中 */
    string(string1, " ");       /* 字符串追加空格 */
strncat(string1, string2, number);  /* 追加相应数量的源字符串2放入目标字符串1中,有结束标
志 \0 */
strcmp(string1, string2);       /* 比较字符ASCII码大小(返回值>0 <0 0 或者 1 0 -1 大于
小于等于) */
strncmp(string1, string2, number) /* 取相应数字的字符比较字符ASCII码大小(返回值 >0 <0 0
或者 1 0 -1) */
stricmp(string1, string2);      /* 以大小写不敏感方式比较两个字符串 */
strchr(string, 'c');            /* 在字符串 1 中查找 字符2 出现的位置,返回位置地址 */
    strchr(string, "abcde");        /* 匹配多个字符*/
strrchr(string,'c');            /* 在字符串 1 中查找 字符2 最后出现的位置,返回位置地址
*/
strstr(string1, string2);       /* 在字符串 1 中查找 字符串2 出现的位置的地址,返回位置地
址 */
strtok(string, "string");       /* 字符串分割,遇到 "string"  进行切割 */
strspn(string, "string");       /* 在 1 中查找非 2 字符集里的字符出现的第一个位置,返回下
标 */
strcspn(string, "string");      /* 在 1 中找 2 里面所包含的字符出现的第一个位置,返回下标
*/
strpbrk(string1, string2);      /* 在 1 中找 2 里面所包含的字符出现的第一个位置,返回字符
串 */
atof(string);                   /* 把一个以小数形式的字符串转化为浮点型 */
atoi(string);                   /* 跳过前面的所有空格将字符串中的数字和正负号转化为整形
(long类型) */
_wtoi(wchar_t);                 /* 转换宽字符串为整型 */
isalnum(char);                  /* 判断此字符是不是字符数字0-9 a-z A-Z */
```

2.4.5函数不定量参数

```
#include <stdarg.h>
    /* 返回的命令行参数是字符串类型 */
    int print(const char *format, ...)
    {
    va_list args;
    const char *args1;
    va_start(args, format);
    args1 = va_arg(args,const char *);
    va_end(args);
    printf("format=%s args1=%s", format, args1);
    }
```

2.5 指针/内存

> *32位系统中4个字节,64位系统8个字节*

2.5.1 static静态变量/函数

```
static int a = 10;          /* 静态全局变量(只能初始化一次),作用域只是本文件 */
static void fun1( );        /* 静态函数(声明时不能与extern使用),作用域只是本文件 */
```

2.5.2 内存四区

```
#代码区                              /* 程序指令 */
#数据区(静态区,全局区)
    初始化的数据                      /* 初始化的全局变量, 初始化的静态全局变量, 初始化的静态全局变量 */
    未初始化的数据(默认初始值未0)/* 未初始化的静态局部变量, 未初始化的全局变量, 未初始化的静态全局变量 */
    常量                              /* 字符串常量, define定义的常量 */
#堆区                                /* 变量, 数组, 结构体, 指针, 枚举, 函数形参, const常量 */
#栈区                                /* 音频文件, 视频文件, 图像文件, 文本文件, 大数据 */
```

2.5.2.1 申请堆区内存

```
#include <stdlib.h>
int *p =(int *)malloc(sizeof(int) * 10);  /* 开辟堆空间,未初始化 */
    calloc(num, sizeof(size));  /* 在内存区中开辟 num 个长度为 size 字节的连续空间,自动初始化内存为0 */
    realloc(p, size);       /* 重新分配由 malloc calloc 在堆中分配内存空间的大小 */

free(p);                    /* 释放由malloc函数开辟分配的空间 */
```

2.5.2.2 内存空间字符串函数 <string.h>

```
#include <string.h>
memset(p, 0, sizeof);        /* 重置开辟的空间会重置每个字节(int)类型有效重置0<目标, 重置值, 字节大小> */
memcpy(p, src, size);        /* 拷贝相应路径的值放入目标地址中, 源地址和目标地址不能发生重叠,字符大小 */
memmove( );                  /* 和 memcpy() 用法一样,效率偏低, 源地址和目标地址可以发生重叠 */
memcmp(p1, p2, size);        /* 比较 p1 p2 所指向的前 n 个字节 (等于 =  大于>  小于< ) */
```

2.6 复合类型/结构体

2.6.1 结构体 <stddef.h>

```
# 结构体变量
    struct info { }namev;        /* 结构体定义,结构体名称 */
    namev.name = value;          /* 结构体变量(赋值字符串需要用 strcpy() 赋值) */
#结构体指针
    struct info { }stu;          /* 结构体定义 */
    struct info *s = &stu        /* 指针指向stu变量 */
    s->name = value;             /* 给指针成员赋值 */
offsetof(struct info, name);     /* 获取结构体成员相对于结构体首地址的偏移量 */
```

2.6.2 共用体/联合体 (union)

> **用法与结构体一样,大小是最大成员大小,所有成员公用一块内存**

2.6.3 枚举(enum)

> **书写格式与结构体一样,用于流程控制**

2.7 C语言文件操作 <stdio.h >

2.7.1 打开方式/基本函数

```
FILE *fp = fopen("URL", "r");    /* 打开文件,返回值为空打开失败 <需打开文件的路径, 打开文件的设置>,
                                    r/rb(以只读模式打开一个文本文件(不创建文件,文件不存在会报错)),
                                    w/wb(以写的方式打开文件(文件存在会清空文件,不存在则会创建一个文件)),
                                    a/ab(以追加方式打开文件(在文本末尾添加内容,文件不存在会创建文件)),
                                    r+/rb+(以可读, 写的方式打开文件(不创建新文件)),
                                    w+/wb+(以可读, 写的方式打开文件(如果文件存在则清空文件, 文件不存在创建一个文件)), a+/ab+(以添加方式打开文件,打开文件并在末尾跟该文件(文件不存在创建文件)) +b的是操作二进制文件 */
FILE* _fsopen(const char* filename, const char* mode, int shflag);  /* 以独占模式打开*/

fclose(fp);              /* 关闭文件,操作完文件 */
fputc('A',fp);           /* 将一个字符写入文件中,会覆盖之前的内容 */
```

```
char ch = fgetc(fp);     /* 读取文件中的一个字符,顺序读取(使用while循环遍历整个文件断文件是否结尾feof判断文件是否结尾EOF判断字符是否结尾) */

feof(fp);                /* 检测是否读取到文件结尾,返回值非 0 以到结尾, 0 未结尾 */
fputs(str, fp);          /* 将str所指定的字符串写入fp文件夹中,字符串结束符,\0不写入返回值0成功 -1失败 */
fgets(temp, size, fp);   /* 从指定文件夹读入对应大小或者遇到\n,保存到temp中,字符串实际大小大于对应设置大小,再次读取从上次读取结尾处,遇换行后时直接多次调用(多次调用需使用memset重置) */

fprintf(FILE* fp, char* temp);  /* 转换格式化字符串temp(用法与printf()一样),将结果保存搭配fp中,遇\0结束 */
fprintf(fp, "%d %s", 111);      /* 第三个变量是存放数据的变量,或者常量 */
fscanf(FILE* fp, char* temp);       /* 从文件 fp 中读取字符串,并转换格式化数据与 scanf 用法一样 */
long ftell(FILE *fp);       /* 用于得到文件位置指针当前位置相对于文件首的偏移字节数 */
fseek(fp, size, SEEK_SET);  /* 移动文件 指定(size) 光标位置,从指定光标位置开始读写 */
    SEEK_SET                    /* 光标位置从文件开头位置移动 size 个字节 */
    SEEK_CUR                    /* 光标位置从当前位置位置移动 size 个字节 */
    SEEK_END                    /* 光标位置从文件末尾位置位置往前移动 size 个字节 (size为负数) */
int len = ftell(FILE *fp);      /* 返回当前光标位置 */
// fgetpos()和fsepos()配对使用,fgetpos和fsetpos配对使用

rewind(fp);                 /* 把文件光标的读写位置移动到文件开头 */
rename(const char *oldname, const char *newname);   /* 文件重命名[要重命名的文件,新的文件名] */
remove( const char *path );         /* 删除文件[文件路径] */
int Findbyte(FILE* fp, char* bydata);   /*按字节搜索数据*/
```

2.7.2 二进制文件读写操作 <stdio.h>

```
fwrite(ptr, size, num, fp);     /* 按照块级写入文件(覆盖写入)<要写入的数据, 一次写入的大小, 写入次数, 要写入的文件> */
fread(ptr, size, num, fp);      /* 按照块级读取文件 */
```

2.7.3 文件属性/状态

```
# 获取文件状态
stat("URL", s);                 /* 获取文件状态,返回结构体  <文件名, 保存文件信息的结构体> */
    struct stat s = {0};            /* 定义接收结构体 */
    stat("URL", &s);                /* 将 path 文件状态信息保存到 s */
    int size = s.st_size;           /* 得到结构体中的成员变量 */
# 结构体成员变量
    st_dev          /* 文件的设备编号 */
    st_ino          /* 节点 */
    st_mode         /* 文件的类型和存取权限 */
    st_nlink        /* 连接到该文件的硬链接数量,刚建立的文件值为1 */
    st_uid          /* 用户ID */
    st_gid          /* 组ID */
    st_rdev         /* 若此文件为设备文件,则为其设备编号(设备类型) */
    st_size         /* 文件字节数(文件大小) */
    st_blksize      /* 块大小(文件系统的I/O 缓冲区大小) */
    st_blocks       /* 块数 */
    st_atime        /* 最后一次访问时间 */
    st_mtime        /* 最后一次修改时间 */
```

```
    st_ctime        /* 最后一次改变时间(指属性) */
```

2.7.4 文件 重命名/删除 函数

```
remove(fp);             /* 删除文件 成功返回值0, 失败-1 */
rename(aname, bname);   /* 修改文件名 <旧文件名, 新文件名> 成功返回值0, 失败-1 */
fflush(fp);             /* 更新缓冲区,将缓冲区的数据立马写到文件 */
```

2.7.5 结构成员偏移量

```
offsetof(struct a, b)           /* 获取成员b相对于结构体开头a的偏移量 */
```

3 C++

3.1 C++头文件/c++调用C函数/c++导入库

3.1.1 标准头文件

```
#include <iostream>                 /* 系统标准输入输出头文件 */
#include <string>                   /* string类型使用必须引用 */
#include <stdexcept>                /* 标准异常头文件 */
#include <iomanip>                  /* io man ip  */
#include <fstream>                  /* 文件读写头文件 */
#include <locale>                   /* 字节 */
#include <cctypc>                   /* 字符串处理函数 */

using namespace std;                /* 使用 std 命名空间 */
```

3.1.2 C++引用C

```
extern "C"{ int add();int sub();}/* 声明这个函数时C语言风格函数,c++调用时用C语言风格调用(.c不用) */
extern "C"{ #include "ios.h"}       /* 头文件里的函数/变量按照C风格进行处理 */


#ifdef __cplusplus      /* 调用时判断是不是cpp调用,如果是就以C语言方式调用(extren "C") */
extren "C"
{
#endif

    int add();              /*会用C语言方式调用 */

#ifdef __cplusplus
}
#endif
```

3.1.3 c++导入库

```
// 导入 lib 库
1 '配置导入库的目录位置'
  "设置"->"VC++目录"->"库目录(.lib)"/"包含目录(.h)"
2 '导入库'
  1)  #pragma comment(lib, "xxx.lib")
  2) "属性"->"链接器"->"输入"->"附加依赖项"->"xxx.lib;"
```

3.2 C++函数

3.2.1 标准输出流

```
 // #include <iostream>
# cout << "string " << endl;     /* 标准的输出 (从缓冲区输出到屏幕) */
cout.flush() ;                      /* 刷新缓冲区, linux有效 */
cout.put() ;                        /* 向缓冲区写字符,一个字符 */
cout.write(buf, sizeof(buf)) ;      /* 将字符数组输出到显示设备 */
cout.width(sizeof) ;                /* 设置字符段的宽度 */
cout.fill('*') ;                    /* 设置填充字符 * */
cout.setf(ios::left);               /* 设置输出格式状态 */
    ios::left                          /* 输出数据向左对齐 */
    ios::right                         /* 输出数据向右对齐 */
    ios::internal                      /* 数值的符号向左对齐, 数值向右对齐 */
    ios::dec                           /* 设置整数的基数为 10 */
    ios::oct                           /* 设置整数的基数为 8 */
    ios::hex                           /* 设置整数的基数为 16 */
    ios::showbase                      /* 显示整数基数(8为0开头, 16为 0x开头) */
    ios::showpoint                     /* 强制输出浮点数的小点和尾数0 */
    ios::uppercase                     /* 输出16进制输出字母为大写 */
    ios::showpos                       /* 对正数显示 '+' 号 */
    ios::scientific                    /* 浮点数以科学计数法格式输出 */
    ios::fixed                         /* 浮点数以定点(小数)格式输出 */
    ios::unitbuf                       /* 每次输出后刷新所有的流 */
    ios::stdio                         /* 每次输出后清除 stduot, stderr */
cout.unsetf(ios::dec);              /* 卸载 十进制  */
        ios::dec                       /* 设置整数的基数为 10 */
        ios::oct                       /* 设置整数的基数为 8 */
        ios::hex                       /* 设置整数的基数为 16 */
'输出二进程内容'
    #include <bitset>
    std::bitset<要显示的二进制位数>(要显示的变量);
    {
        int test{0xFFFF };
        std::cout<<std::bitset<32>(test)<<endl;
    }
```

3.2.2 标准输入流

```
# cin >> "string";          /* 标准输入( 从输入设备输入到缓冲区) */
cin.get()                       /* 一次只能读一个字符 (C语言的 getchar())    重载 */
cin.get(buf, sizeof(buf));      /* 可以读取字符串, 换行符遗留在缓冲区    重载 */
cin.getline(buf, sizeof(buf));  /* 获取字符串,换行符未遗留在缓冲区中,并且从缓冲区中清除
重载 */
cin.igonre( )                   /*忽略缓冲区字符(无参默认忽略一个字符,可以提供int类型参
数,
根据参数忽略个数)    重载 */
cin.peek()                  /* 查看缓冲区中的第一个字符,并不取走   没有重载 */
cin.putback(buf)            /* 将从缓冲区取出的buf放回缓冲区中,并且放回原位 */
cin.fail() ;                /* 查看缓冲区的标志位, 0为正常 1为异常 */
cin.clear() ;               /* 重置标志位 */
cin.sync() ;                /* 清空缓冲区(vs2013可用,vs2015以上版本无效)  cin.getline
清空缓冲区 */
```

3.2.3 string函数

```
std::string a{"abcdcfghijklmn"};
    .length()                   /* 字符串的长度 */
    .substr()                   /* 截取字符串 */
        std::string a1 = a.substr(5);       /* 从下标位置为5开始截取一直到结尾 */
        std::string a2 = s.substr(5, 3);    /* 从下标为5开始截取长度为3位 */
    .compare()                  /* 比较字符串 */
        a.compare("Hello");         /* 比较字符串大小, 返回int类型的值(相等0,小于负数,
大于正数) */
        a.compare(5,4,"cdef");      /*  比较字符串大小 (起始位置,参与比较的长度,被比较的
字符串) */
        a.compare(5,4, "cdef safasf",0,4)
                /*  (起始位置,参与比较的长度,被比较的字符串,被比较字符串的起始位置,  长度)
*/
    .find()                     /* 搜索字符串,如果没找到返回 std::string::npos */
        a.find("dcfgh");                /* 搜索文本出现的位置 */
        a.find("dcfgh",3);              /* 从位置3开始搜索 dcfgh */
    .rfind()                    /* 从后往前搜索字符串 */
    .insert()                   /* 插入字符串 */
        a.insert(3,"ddddddd");          /* 从位置3的地方插入字符串 */
        a.insert(3,2,'d');              /* 从位置3的地方插入2个字符 */
        a.insert(3,'daasdasdsaa',3,5);  /* 从字符串位置3的地方截取5个字符插入位置a 3的
地方 */

.c__str()
    const char* baseStr = str.c_str();  /* 返回常量指针,查看字符串存放地址 */
.date()
    const char* baseStr = str.c_str();  /* 查看字符串存放地址, 返回常量指针(c++17改成非
常量指针) */
.length()                   /* 查看字符串长度 */
.replace()                  /* 替换字符串 */
    string str(""id=user;);
    str.eplace(3,4,"zhangsan");     /* 被替换内容起始位置,被替换的长度,替换掉内容 */
    str.eplace(3,4,6,'*');          /* 被替换内容起始位置,被替换的长度,替换后的字符长度,
替换掉内容 */
    str.eplace(3,4,"zangsan",7);    /* 被替换内容起始位置,被替换的长度,要替换内容,要替换
内容的长度 */
```

```
    str.eplace(3,4,"zangsan",5,3);  /* 被替换内容起始位置,被替换的长度,要替换内容,截取要替换内容的起                                始位置,要替换内容的长度 */
.erase()                          /* 删除字符串 */
    str..erase();                 /*  删除所有内容 */
    str..erase(3);                /*  从位置3开始后面全部删除 */
    str..erase(3,4);              /*  要删除内容的起始位置,要删除的长度 */
    str.clear();                  /*  删除所有内容 */
```

3.2.4 字符串处理函数

```
' #include <cctyoe> '
    int isupper(char)             /* 判断字符是都为大写字母 */
    int islower(char)             /* 判断字符是否为小写字母 */
    int isalpha(char)             /* 看看字符是否为字母 */
    int isdigit(char)             /* 看看字母是否为数字 */
    int isalnum(char)             /* 看看字符是否为字母或者数字 */
    int isspace(char)             /* 看看字符是不是空白 */
    int isblank(char)             /* 看看字符是不是空格 */
    int ispunct(char)             /* 看看字符是不是标点符号 */
    int isprint(char)             /* 看看字符能不能打印 */
    int iscntrl(char)             /* 看看字符是不是控制字符 */
    int isgraph(char)             /* 看看字符是不是图形字符 */
    int tolower(char)             /* 将字符转换为小写 */
    int toupper(char)             /* 将字符转换为大写 */

std::to_string(123);              /* 可以把数字转换成字符串 */
std::stoi("123456");              /* 可以把字符串转换成数字 */
std::stringstream                 /* #include <sstream> 字符串流处理函数
（std::cout<< 形式处理                                    string函数) */
```

string 字符串转换为数字		
std::to_string(数字) 可以把数字转换为字符串;		
通过以下函数可以把字符串转换为数字		
作用	语法	用法
将字符串转换为int	std::stoi(字符串)	int a=stoi(“123”)
将字符串转换为long	std::stol(字符串)	long a=stol(“123”)
将字符串转换为long long	std::stoll(字符串)	long longa=stoll(“123”)
将字符串转换为unsigned long	std::stoul(字符串)	unsigned long a=stoul(“123”)
将字符串转换为unsigned long long	std::stoull(字符串)	unsigned long long a=stoull(“123”)
将字符串转换为float	std::stof(字符串)	float a=stof(“123”)
将字符串转换为double	std::stod(字符串)	double a=stod(“123”)
将字符串转换为long double	std::stold(字符串)	long double a=stold(“123”)

3.2.5 不定量参数函数

```
'接受不定量参数函数'
    ' #include <cstdarg> '
    int Add(unsingned count, .....)
    {
        char* arg{};                        /* va_list arg; */
```

```
        va_start(arg,count);                    /* 接收传入参数的位置(接收参数位置的指针,参数个数) */
        int x = va_arg(arg, int);               /* 读取参数(参数位置指针,参数类型),每调用一次函数，函数指针指向会自动指向下一个参数 */
        x = va_arg(arg, int);
        std::cout<<x<<(char)10;

        va_end(arg);                            /* 释放指针的内存 */
    }
    void main()
    {
        int a= Add(3,1231,123213,131)           /* 第一个参数是参数个数 */
    }
```

```
'__fastcall'
    int __fastcall a(int, int); /* 被调用函数自己恢复栈平衡,通过寄存器来传参,运行速度较快,x64系统默认 */
'__stdcall'
    int __stdcall a(int, int)   /* 调用函数压栈,并且负责恢复栈平衡 */
'__thiscall'            /*  ---C++中,类的访问  */
'naked call'            /*  ---不常用调用约定,用于实模式驱动开发 */
```

3.3 c++数据类型

3.3.1 c++数据类型转换

```
'查看变量类型'
    typeid(变量名).name()
```

```
# char类型字符串 (const char* 可以隐式转换成 string)
st.c_str();             //string转const char *类型 (st为 string类型的变量 )
int a = static_cast<int>(temp);     //静态类型转换(允许内置的数据类型转换)
dynamic_cast<int>(temp);            //动态类型转换(不允许内置的数据类型转换, 不可父类指针转子类指针)
comst_cast                          //(只能操作指针和引用)
    int *a = const_cast<int *>(p);   //将非const修饰的指针常量和引用加上const,有sonst的卸掉const
    const int *a = const_cast <const int *>(p);
    const int &num = const_cast <const int &>(numF);
    int &num = const_cast<int &>(numF);   //修改const 修饰的 引用
reinterpret_cast                //重新解释转换(不安全)
```

```
'字符数据类型'
    char        1字节           ascii字符
    wchar_t     2字节           宽字节字符
    char16_t    2字节           utf_16字符 u
    char32_t    4字节           utf_32字符 U
```

```
'推断类型
    auto x {0};    // auto 不能用const修饰，auto 根据初始值推断出x的类型
    decltype(a-b) x;   // 依据a-b推断出x的类型，decltype可以保留const和引用类型,如果是引用必须有指向变量
```

```
'std::array'
   ' # include <array>'
    std::array <int, 3> studentId {0};              // array 数组创建声明

        studentId[0];                               // 直接使用下标访问数组元素
        studentId.at(1);                            // 访问数组下标为1的元素
        studentId.size();                           // 求数组有几个元素
        studentId.fill{9};                          // 设置数组每个子元素的值为 99
```

3.3.2 自定义数据类型(typedef/using)

```
'自定义变量名称取别名'
    #define A TypeName                  // 以后代码中的A可以被替换为TypeName
    typedef TypeName A;                 // 以后TypeName类型的名字可以用A替换
    using A=TypeName;                   // 以后TypeName类型的名字可以用A替换
'自定义函数指针类型'
    char* add(int a, int b);                // 函数指针的类型取决于函数的返回值类型
    typedef char(*pfAdd)(int,int);
    pfAdd a = add;
    using pFAdd = char(*)(int,int);
    pFAdd a = (pFAdd)add;
    a(1,2);                                 // 使用指针直接调用函数
```

3.3.3 枚举类型

```
    enum class 类型:基本类型       // 自定义数据类型
    {
        选项..1,
        选项..2,
    };
    /*
    枚举类型基本类型默认为int, 基本类型只能设置为整型， 枚举类型计算是必须强转为int类型，
    成员值默认为上一项的值+1
    */
```

3.3.4 引用/通配符/运算符

```
# 引用
    int &b = a;          //对 a 变量取别名(小名),不能直接赋值,必须是合法空间
    const int &a = 10;  //const 修饰编译器会开辟临时内存,并将值放进去,后续利用指针修改
const修饰值
::                  //双冒号,作用域运算符(全局作用域)
a>b ? a : b;        //三目运算符,c语言三目运算符返回的是变量里面的值, C++返回的值
namespace           //命名空间,必须定义在全局下(可以放 函数 变量 结构体 类... ,命名空间可
以互相嵌套)
```

3.4 指针/内存

3.4.1 指针类型

```
'整型指针'
    int* p;
'字符指针'
    char* p;
'数组指针'
    int (*p)[n];
'指针数组'
    int* p[n];
'函数指针'
    int (*pAdd)(int,int)=Add;                // 函数指针的类型取决于返回值类型
    pAdd(1,2);

    typedef char(*pfAdd)(int,int);           // 声明一个函数指针类型
    using pFAdd = char(*)(int,int);          // 声明一个函数指针类型
    pfAdd a = (pfAdd)add;
'指针函数'
    int* pAdd(int,int);                      //指针函数是指一个返回指针的函数
```

3.4.2 申请内存

```

    // C
    int* p1 = (int*)malloc(sizeof(int)*x);
    int* p2 = (int*)calloc(x,sizof(int));        //申请内存并初始化为0
    int* p3 = (int*)realloc(p2,sizof(int));      //重新分配 p2内存的大小，原数据不丢失
    free(p1); p1=0;

    // C++
    int* p1 = new int;                           // 申请一个int内存大小的堆空间
    int* p2 = new int[x];                        // 申请x个int元素大小的内存空间
    delete p1;  p1=0;
    delete[] p2; p2=0;
```

3.4.3 智能指针

```
'唯一智能指针'
    std::unique_ptr <int> ptr( std::make_unique<int>(5) );          //指针里的值初
始化为5
    std::unique_ptr <int[]> ptr{ std::make_unique<int[]>(5) };      //申请5个元素的
数组指针
        ptr.reset()                          // 释放指针里的内存空间并将指针指向 null
```

```
    int* a = new int[5];
        a = ptr.get;                            // 将ptr保存到内存地址给 a
        a = ptr.release();                      // 将ptr里保存的地址给 a,并且将ptr指向
null

    std::unique_ptr<int> ptrA( std::make_unique<int>(5) );
    std::unique_ptr<int> ptrB{};
    ptrB = std::move(ptrA);                     // 将ptr里保存的地址给 a,并且将ptr指向
null

'共享智能指针'   --> td::make_shared 不支持数组
    std::shared_ptr<int> ptrA{};
    std::shared_ptr<int> ptrA{ std::make_shared<int>(5) };  //指针里的值初始化为5
    std::shared_ptr<int[]> ptrC{new int[5]{ 1,2,3,4,5} };  //申请5个元素的数组指针
        std::shared_ptr<int> ptrD{ ptrA };                  //

        ptrD.use_count();   // 返回当前指针共有多少个对象调用
        ptrD.unique();      // 如果当前智能指针是唯一指向指针，返回true,否则返回false,
C++14
        ptrA.reser();       // 如果当前智能指针是唯一指向指针释放ptrA并将他指向null，否则
不是释放只改指向
```

3.4.4 指针结构体

```
typedef struct Role
{
    int HP;
    int MP;
} PRole;
int main()
{
    Role user;
    PRole puser = &user;
    puser->HP=50;
    puser->MP=50;
    user.HP=50;
    user.MP=50;
}
```

3.4.5 引用

```
    int a1{500};
    int& a2{a1};
```

3.4.6 内存空间操作

```
'memcpy'
    int a[]{1,2,3,4};
    int *p = (int*)malloc(sizeof(int)*4);
    memcoy(p, a, sizeof(int)*4);                    // 复制内存空间数据 {目标，源，内存大小}

'memset'
    int* p = (int*)malloc(sizeof(int)*4);
    memset(p, 0, sizeof(int)*4);                    // 设置内存空间数据 {目标，值[0-FF]，内存大小}
```

3.5 类

3.5.1 class 类

```
#成员属性
#成员函数                          //类内部的成员函数编译器默认前面加 inline 内联函数关键字
#访问权限
    public                        //公共权限
    protected                     //保护权限
    private                       //私有权限
#数据的初始和清理
    构造函数(初始化)               //没有返回值,没有void,与类名相同,重载,可以设置参数,系统默认调用
    拷贝构造函数                   //没有返回值,没有void,与类名相同,参数用 const 修饰
    析构函数(清理)                 //没有返回值,没有void,,类名前面加 ~,,不可以重载,没有参数.系统默认调用
explicit                          //防止构造函数中的隐形类型转换
#静态成员变量和静态成员函数
    static int m_Age;             //静态成员变量会所有类对象共享数据,类内声明,类外实现
    int Person::m_Age = 10;       //全局作用域下的 类外实现
    static void func(){ };        //与所有成员共享函数,不可访问普通成员变量
#创建对象与清除(new运算符和delete运算符)
    Person *p1 = new Person;      //创建新对象,放在堆区空间,不会自动释放,构造函数自行初始化
    delete p1;                    //配合 new 用, 释放对象空间内存,释放对象数组必须 delete [ ] p1;
#this指针                         //对this取 * 返回的永远是对象本体1
mutable 关键字                    //允许修改常函数 值
friend  友元类/函数               //全局函数写在类中声明 (前面加friend关键字) 访问类中的私有成员属性
operator+                         //运算符重载
```

3.6 封装/继承/多态/函数模板

3.6.1 继承

```
class 子类 : 继承方式 父类1, 继承方式 父类2
#菱形继承    //菱形继承,两个父类继承在上一集的父类,子类继承同一种类数据无意义,在两个父类继承时使用虚基类
    virtual 虚基类\虚函数          //里面包含 vbptr指针(虚基类指针),指向 vbtable(虚基类表),表中索引1记    录vbptr地址到继承数据的偏移量 (继承数据 = &vbptr + 偏移量)
        class Animal{ int m_Age; };
        class Sheep : virtual public Animal{ };
        class Tuo : virtual public Animal{ };
        class sheep_Tuo : public Sheep, public Tuo{ };  // 只会继承一份 m_Age 下来
```

3.6.2 多态

> 父类型声明 | 子类型重写虚函数

3.6.3 重载 /重定义/重写

```
#重载                    //同一作用域下,  函数名相同,  参数个数 顺序 类型不同
#重定义(有继承关系)       //子类重新定义父类中同名成员函数, 将父类中同名成员函数隐藏掉
#重写(有继承关系)         //父类中有(纯)虚函数, 子类重写父类中的(纯)虚函数(返回值类型 函数名 形参列表一致)
```

3.6.4 函数模板/类模板

```
template <typename T>T call(T a,T b,T c)             //函数模板化
    /*   template <typename T>T call(T,T,T)      */
    /*   template <typename T>T* call(T* a,T* b,T* c)  */
    /*   template <typename T1,typename T2,typename TR>TR call(T1 a,T2 b)  */

    template <typename T>
    T call(T a,T b,T c)
    {
        return a+b+c;
    }
'函数模板自动转换类型'
    call(1,2,3);
'函数模板指定类型'
    call<int>(1,2,3);           // 指定类型, 通过<typename T>强制所有参数按照指定类型处理
'函数模板的例外处理(指针)'
    template <typename T>T call(T a,T b,T c){};
    template<>int* call(int*,int*);
'函数模板重载'

'函数模板的默认参数'
    template <typename TR=int,typename T1,typename T2>TR call(T1 a,T2 b){}
    template <typename T1,typename T2, typename TR=T1>TR call(T1 a,T2 b){}
'非类型的模板参数'
    template <int Max=200,int Min=100,typename T1>bool call(T1& a,T1 b){ }


decltype(auto) bigger(int& a, int& b)->decltype(a>b?a:b) { return a>b?a:b; }  //decltype返回引用
                            // decltype()用return的值
```

```
auto bigger(int& a, int& b)->decltype(a>b?a:b) { return a>b?a:b; }  // auto 返回值

template<class T>                //类模板

'类模板的模板函数'
template<class T> :: T call
```

```
''
```

3.5 C++ exception 异常

```
#exception  所有标准异常基类
if(a == 0){throw -1;}            //判断是否发生异常返回    抛出异常
try { if(a == 0){throw -1;} }    //可能出现异常的函数(代码段)
catch (int){ };          //异常捕获(int返回异常的类型char double),除了这三种类型其他楼型捕获都用 ...
```

3.6 C++文件读写 < fstream>

```
ofstream fp("/a.txt", ios::out | ios::trunc);   //创建文件写入对象,并指定路径 打开方式
    fp.open("/a.txt", ios::out | ios::trunc);   //给已创建的文件写入对象设置路径和打开方式
        ios::out                    //以输出(写)方式打开(默认),如果已有同名文件,将其原有内容删除
    if(fp.is_open())                //判断文件对象是否打开成功,返回值为 bool 值
    fp.close();                     //关闭文件对象
ifstream ifs("/a.txt", ios::in)     //创建文件读取对象,指定路径 打开方式
    ifs.open("/a.txt", ios::in)     //给已创建的文件读取对象设置路径和打开方式
        ios::in                     //以输入(读)方式打开
    if(ifs.is_open())               //判断文件对象是否打开成功,返回值为 bool 值,打开成功为 true
```

3.7 STL(容器/迭代器/算法)

3.7.1 容器

```
# 序列式容器, 关联容器
容器 (vector, list, deque, set, map)
    // std::vector
    '#include <vector/容器名>'
    vector<int> v;                      //创建容器并声明迭代器 v
    v.push_back(10);                    //插入数据
    v.end();                            //关闭容器

        std::vector <int> studentId {0,1,2,3}       //申明vector容器，并初始化
        std::vector <int> studentId (5,500);        //声明容器，有5个元素，每个元素都初始化为500
        studentId.assign(5,100);                    //初始化容器，有5个元素，每个元素都初始化为500
        studentId.push_back(100);                   //添加容器元素并赋值
        studentId.clear();                          //清空容器元素
```

```
        studentId.empty();                                  //返回bool类型，判断是否为空元素
```

3.7.2 迭代器

```
#迭代器(iterator)
vector<int>::iterator p1 = v.begin();        //起始迭代器，指向容器中第一个元素的地址
vector<int>::iterator p2 = v.end();          //结束迭代器，指向容器中最后一个元素的下一个位置的地址
    for(vectot<int>::iterator it = v,begin(); it != v.end(); it++) { cout<< *it <<endl; }                                                    //遍历容器
```

3.7.3 算法 < algorithm>

```
#算法(sort, find, copy, for_each)
for_each(vi.begin(), v.end(), function);     //起始位置，结束位置(判断起始位置等于结束位置,等于结束遍历)，递归函数(功能实现)
```

4 数据结构

二叉树

```
'二叉树删除'
    /*
    情况一：叶子节点
        1、删除该节点
        2、将父节点(左或者右)指针置NULL
    */
    /*
    情况2：只有一个子树
        1、删除该节点
        2、将父节点(左或者右)指针指向子树
    */
    /*
    情况3：左右子树都有
        1、用右子树最小的节点取代源节点
        2、再递归删除最小节点
    */
```

5 Windows API

5.1 Windwos数据类型

```
'Windows数据类型'
    BOOL            int
    BYTE            unsigned char
    CHAR            char
    INT             int
    CONST           const
```

```
    DWORD           unsigned long
    DWORD32         unsigned int
    DOWRD64         unsigned long long
    HANDLE          void*
    HINSTANCE       指针
    HKEY            指针
    HWND            /* 句柄指针 */
    SC_HANDLE       // 服务句柄
    HGLOBAL         // 全局内存块句柄
```

5.2 Win32字符串处理函数

```
'WCHAR转CHAR'

    // 宏函数转换,宏函数需使用 USES_CONVERSION;
    char* pa = W2A(L"宽字符");

    // API函数转换
    WideCharToMultByte();

'CHAR转WCHAR'
    /* 宏函数, 宏函数需使用 USES_CONVERSION;*/
   WCHAR* A2W(CHAR*);
    // API函数
    MultiByteToWideChar();

'窗口上的字符串操作' // strsafe.h
    int __cdecl wprintf(LPSTR lpout, LPCTSTR lpFmt, ...);// 格式化字符串和数字,实际中使用StringCchPrintf
    // 将格式化数据写入指定的字符串,结尾会自动填0
    STRSAFEAPI StringCchPrintf(STRSAFE_LPSTR pszDest, size_t cchDest, STRSAFE_LPCSTR pszFormat, ...);

    // 计算字符串的长度,实际中使用StringCchLength或 StringCbLength
    int WINAPI lstrlen(LPCTSTR);

    HRESULT StringCchLength(LPCTSTR, size_t, size_t*);  // 计算字符串长度
    HRESULT StringCchCat(LPSTR, size_t, LPCTSTR);   // 字符串拼接
    HRESULT StringCchCopy(LPTSTR, size_t, LPCTSTR); // 字符串拷贝
```

Windows消息类型

```
' Windwos接收消息类型'
    WM_CREATE           // 窗口被创建前发送的消息
    WM_INITDIALOG       // 对话框窗口创建时的消息
    WM_PAINT            // 窗口重绘消息
    WM_SETICON          // 设置图标的类型

    WM_COMMAND          // 标准控件 子控件被操作时返回给父窗口的消息
    WM_NOTIFY           // 通用控件子控件发送给窗口的通知消息, 涵盖比WM_COMMAND更多操作消息
                /*  lParam 指向了一个 NMHDR的结构 */
```

```
    WM_LBUTTONDOWN        // 鼠标左键被按下
    WM_KEYDOWN            // 键盘被按下
    WM_MOV                // S鼠标移动
    WM_SIZE               // 窗口大小调整
    WM_COMMAND            // 子窗口(菜单)操作消息

    WM_CLOSE              // 当用户点击关闭按钮的时候
    WM_DESTROY            // 窗口销毁后得到的消息,但是进程并没有结束，转入'后台程序'
    WM_NCDESTROY          // 非客户区
    WM_QUIT

    WM_VSCROLL            // 窗口滚动条(垂直)触发消息
    WM_HSCROLL            // 窗口滚动条(水平)触发消息
'Windwos发送消息类型'
    LVM_INSERTCOLUMN          // 新建列表添加目录名
    LVM_SETEXTENDEDLISTVIEWSTYLE

 '回调函数 消息结构体'
   NMHDR     hNmhdr;          // 后面的结构继承(包含) NMHDR
    NMLVCACHEHINT ;
    NMLVDISPINFO;
    NMLVFINDITEM;

'调试打印'
    TRACE（）                              // 与printf()用法一样
```

5.3 Windwos 窗口过程

```
"Windows入口函数"
    int WINPAI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                       LPSTR szCmdLine, int iCmdShow){ return 0; }
'Windows窗口类'
    WNDCLASS windclass      // Windows窗口类
    {
        UINT      style;             // 窗口的类型
       WNDPROC   lpfnWndProc;       // 窗口的消息处理函数
        ......................
    };

'Windows窗口函数'
    RegisterClass(windClass);            // 注册窗口
    HWND CreateWindow(lpClassName, lpWindowName, dwStyle, x, y, .....);  // 创建
窗口
    showWindow(hwnd, nCmdShow);                  // 显示窗口
    UnregisterClass();                   // 注销窗口类
'Windows消息循环'
    MSG msg;
    while(GetMessage(&msg, NULL, 0, 0))          // 接收消息队列消息
    {
        TranslateMessage(&msg);          // 将虚拟键消息转换为字符消息
        DispatchMessage(&msg);           // 将消息派送给窗口过程(消息回调)
    }

'Windwos消息处理函数'
    RegisterWindowMessageA( )        // 注册自定义消息
```

```
    LRESULT CALLBACK WindowProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM
lParam)
    {
        // 如果我们不处理的消息返回给windwos默认的消息处理函数进行处理
        return DefWindowProc(hwnd, uMsg, wParam, lParam);
    }

'对话框'
    DialogBox();                创建对话框
```

5.4 Windows API

```
    DestroyWindow(HWND hWnd);    // 销毁一个指定窗口
    PostQuitMessage(0);          // 向系统表明有个线程有终止请求,通常用来响应WM_DESTROY消
息

    InvalidateRect(HWND, LPRECT, BOOL);      // 更新指定窗口指定区域
```

```
#include<stdlib.h>
#include<Windows.h>
    GetCommandLineA();       /* 检索当前进程的命令行字符串,返回指向当前进程的命令行字符串的
指针 */

    HANDLE GetStdHandle(DWORD nStdHandle);  /* 获得输入、输出/错误的屏幕缓冲区的句柄 */
    HWND GetConsoleWindow(void);                  /*获取当前控制台窗口句柄 */
    BOOL SetConsoleCursorPosition(HANDLE, COORD);   // 命令行移动光标位置
    BOOL SetWindowPos(HWND hWnd, HWND hWndInsertAfter, int x, int y,
        int cx, int cy, UINT uFlags);   /* 设置控制台窗口大小,样式hWndInsertAfter宏设
置类成员变量                            wndNoTopMost，  wndTopMost，wndBottom,wndTop*/

    /* 在指定的坐标开始写入指定次数的字符到指定控制台屏幕缓冲区 */
    BOOL WINAPI FillConsoleOutputCharacter(
      __in HANDLE hConsoleOutput,
      __in TCHAR cCharacter,
      __in DWORD nLength,
      __in COORD dwWriteCoord,
      __out LPDWORD lpNumberOfCharsWritten
      );

    OutputDebugString( );           /* VS调试窗口打印 */
    DWORD GetLastError( );          /* 得到这个函数前面出现的最后一个错误的编号*/

    ShellExecute( );                      /* 打开外部程序或者文件 */
    CreateObject( );                   /* COM */
```

```
'鼠标事件'
   CPoint                           // CPoint 结构定义点的x和y坐标
    DWORD GetMessagePos             // 获取最后一次鼠标事件鼠标的位置
    GetCursorPos(CPoint* cp);       // 获取当前鼠标的位置

'滚动条操作'
    SetScrollRange();               // 设置滚动条的范围
```

```
    GetScrollRange();                // 获取滚动条的范围
    SetScrollPos();                  // 设置滚动条滑块的位置
    GetScrollPos();                  // 获取滚动条滑块的位置

    SetScrollInfo();
    GetScrollInfo();

    ScrollWindow();                  // 滚动指定窗口的客户区内容
'文件操作'
    ImageLoad( );                    // 将文件加载内存获取PE信息
    ImageUnload( );                  // 释放加载到内存中的PE信息

    OPENFILENAMEA OpenFile = {0};            // 文件模态对话框,需要使用ZeroMemory初始
化
    GetOpenFileNameA(OPENFILENAMEA);     //创建一个打开的对话框
        {
      LPSTR pFileName = (LPSTR)malloc(MAX_PATH);
        memset(pFileName, 0, MAX_PATH);
        OPENFILENAMEA OpenFile = { 0 };
        ZeroMemory(&OpenFile, sizeof(OPENFILENAME));
        OpenFile.hwndOwner = hDia;
        OpenFile.lStructSize = sizeof(OpenFile); // 结构大小
        OpenFile.lpstrFile = pFileName; //存储路径的地址
        OpenFile.nMaxFile = MAX_PATH; // 路径大小
        OpenFile.lpstrFilter = ("All\0*.exe;*.dll;*.lib\0\0"); // 文件类型
        OpenFile.Flags = OFN_PATHMUSTEXIST | OFN_FILEMUSTEXIST;
        GetOpenFileNameA(&OpenFile);
        if (*pFileName == 0)
            {
            MessageBox(hDia, TEXT("未选择PE文件!"), TEXT("ERROR"), MB_OK);
            return NULL;
            }

        }
```

```
'定时器'
    /* 郁金香c++ 91课*/
    UINT WINAPI SetTimer(HWND hWnd, UINT nIDEvent, UINT uElapse, TIMERPROC
lpYimerFunc);

    /* 计时器的回调函数*/
    VOID CALLBACK TimerProc(HWND hWnd, UINT uMsg, UINT idEvent, DWORD dvTime);

    /* 销毁计时器*/
    BOOL WINAPI KillTimer(HWND hWnd, UINT_PTR uIDEvent);
```

5.4.1 进程线程操作

```
'进程'
   HANDLE GetCurrentProcess();                        /* 获取当前进程句柄*/
   /* 创建进程*/
    BOOL CreateProcess(LPCWSTR pszImageName, LPCWSTR pszCmdLine,
LPSECURITY_ATTRIBUTES psaProcess,
            LPSECURITY_ATTRIBUTES psaThread, BOOL fInheritHandle, DWORD
fdwCreate, LPVOID pvEnvironment,
```

```
          LPWSTR pszCurDir, LPSTARTUPINFOW psiStartInfo, LPPROCESS_INFORMATION
pProcInfo);

    /* 进程结束*/
    void ExitProcess(UINT uExitCode);                          /* 进程自己正常结束*/
    BOOL TerminateProcess(HANDLE hProcess, UINT uExitCode);     /* 非正常结束(强制
结束)*/

    /* 获取/设置(启用/禁用)进程实时优先级(CPU占用时间片)*/
    BOOL SetProcessPriorityBoost(HANDLE hProcess, BOOL bDisablePriorityBoost);
    GetProcessPriorityBoost();  /* 获取(启用/禁用)当期进程的实时优先级(CPU占用时间片)*/

    BOOL SetPriorityClass(HANDLE hProcess, DWORD dwPriorityClass);  /* 设置进程优先
级*/
    GetPriorityClass();                                        /* 设置进程优先级*/
```

```
'线程'
    HANDLE GetCurrentThread(VOID);                  /* 获取当前线程句柄*/
    DWORD GetCurrentThreadId(VOID);                 /* 获取当前线程ID*/
    BOOL GetExitCodeThread(HANDLE hThread, LPDWORD lpExitCode);     /* 获取线程结束
的返回代码*/
    BOOL CloseHandle(HANDLE hObject);               /* 关闭线程的句柄*/

    DWORD ThreadProc(_In_ LPVOID lpParameter);      /* 线程的消息回调函数*/
    /* 创建线程*/
    HANDLE CreateThread(LPSECURITY_ATTRIBUTES lpThreadAttributes, SIZE_T
dwStackSize,
            LPTHREAD_START_ROUTINE lpStartAddress, __drv_aliasesMen LPVOID
lpParameter, DWORD dwCreateFlags,
            LPDWORD lpThreadId);

    /* 创建远程线程*/
    HANDLE CreateRemoteThread(HANDLE hProcess, LPSECURITY_ATTRIBUTES
lpThreadAttributes, SIZE_T dwStackSize,
        LPTHREAD_START_ROUTINE lpStartAddress, LPVOID lpParamter, DWORD
dwCreationFlags, LPDWORD lpThreadID);

    /* 结束线程*/
    void ExitThread(DWORD dwExitCode);                          /* 线程自己退出*/
    BOOL TerminateThread(HANDLE hThread, DWORD dwExitCode);         /* 强制结束线程
*/


    DWORD ResumeThread(HANDLE hThread);     /* 恢复线程执行*/
    DWORD SuspendThread(HANDLE hThread);    /* 挂起线程*/
    BOOL GetThreadContext(HANDLE hThread, LPCONTEXT lpContext);     /* 检索线程上下
文*/
    BOOL SetThreadContext(HNADLE hTread, const CONTEXT* lpContext);     /* 设置线
程上下文*/
    int GetThreadPriority(HANDLE hThread);      /* 获取线程的优先级(线程的优先级基于进
程的优先级进行调整)*/
    BOOL SetThreadPriority(HANDLE hTread, int nPriority);

'线程管理 (多线程)'
   InitializeCriticalSection( );            // 初始化临界区
    EnterCriticalSection( );                // 进入临界区
    LeaveCriticalSection( );                // 离开临界区
```

```
'进程线程亲缘性'
    /* 进程以二进制决定使用哪个CPU和数量(1[1] 2/2[0x3] 1/2/3[0x7]),线程使用超过进程，以进程的为值 */
    BOOL SetProcessAddinityMask(HANDLE hProcess, DWORD_PTR dwProcessAffinityMask);  /* 设置进程可以使用的CPU*/
    DWORD_PTR SetThreadAddinityMask(HANDLE hThread, DWORD_PTR dwThreadAffinityMask);/* 设置线程可以使用的CPU*/
```

```
'获取模块地址'
    GetModuleHandle（）;          // 获取模块的起始地址
    FARPROC GetProcAddress(HMODULE hMpdule, LPCSTR lpProcName);     // 获取模块函数的地址
    BOOL GetCurrentDirectoryA( );            // 获取当前工作目录

'获取窗口句柄'
    HWND FindWindow( );                  // 通过窗口类获取窗口句柄
    GetProcessId( );                     // 根据当前进程句柄获取PID
    DWORD GetWindwoThreadProcessId( );  //根据窗口句柄获取进程ID和线程ID, 返回值创建窗口线程ID

'Hook'
    HHOOK SetWindowsHookExW(int idHook, HOOKPROC lpfn, HINSTANCE hmod, DWORD dwThreadId); // Hook主线程 ( 1 )
    LRESULT CALLBACK KeyBoardProc(_IN_ int code, _IN_ WPARAM wParam, _IN_ LPARAM lParam);
            /* 键盘钩子的回调函数 与SetWindowsHook一起使用的应用程序/库定义的回调函数*/
    SetWindowlong( );                        // Hook主线程 ( 2 )
    LRESULT CallNextHookEx(HHOOK hhk, int nCode, WPARAM wParam, LPARAM lParam);
    HOOKPROC Hookproc( );
    // 与SetWindowsHookEx函数一起使用的应用程序定义或库定义的回调函数,调用SendMessage函数后，系统将调用此函数
        UnhookWindowsHookEx( );                 // 卸载Hook
    GetWindowThreadProcessId( );            // 获取线程ID


'遍历进程'
    CreateToolhelp32Snapshot( );

        EnumProcesses( );
        EnumProcessModules( );
        GetModuleBaseName( );
        WTSEnumerateProcess( );
        ZwQuerySystemInformation( );        /* nt.dll */

'内存数据读写'
    LPVOID VirtualAllocEx( );                       // 跨进程创建空间
    VirtualFreeEx()                             // 释放远线程申请的空间
    VirtualProtectEx( );                        /* 修改内存保护属性 */
    CreateFileMapping( );
    MapViewOfFile( );

    ReadProcessMemory( );                   /* 跨进程读取数据 */
```

```
    WriteProcessMemory( );                  /* 跨进程写入数据 */

    HANDLE CreateRemoteThread( );                   // 跨进程 启动远程线程
    WaitForSingleObject( );                 // 等待远线程执行完发送信号或者超时多少秒


// 窗口创建
    CreateWindow( );
        CreateWindowEx( );
    DialogBox( );
    CreateDialog( );

    ListView;

// 消息进制
    SendMessage( );                     /* 发送消息 */
    PostMessage( );

        mouse_event                     /* 模拟鼠标事件 */
        keybd_event                     /* 模拟键盘事件 */

// 基础GDI操作(绘图)
    GetDC( );
    RelaseDC( );
    BeginPaint( );
    EndPaint( );
        // 三角函数
        sin
        cos

 DWORD dwErr=   GetLastError( );

// 动态链接库 (DLL)
    LoadLibraryA( );                    /* 加载动态链接库 */
    GetProcAddress( );                  /* 导入动态链接库 */
    FreeLibraryAndExitThread( );        // 创建新线程卸载某个DLL

'ini操作'
    GetPrivateProfileInt( );        // 读取ini文件列表键值
    WritePrivateProfileString( );   // 写入ini文件键值
    GetPrivateProfileString( );     // 读取ini文件键值
```

```
'网络通信'
    'TCP'
        send( );                        // 发送
        recv( );                        // 接收
        WSASend( );                     // 发送
        WSARecv( );                     // 接收
    'UDP'
        sendto( );                      // 发送
        recvfrom( );                    // 接收
        WSASendTo( );                   // 发送
        WSARecvFrom( );                 // 接收

'winsock2.h'
typedef struct _WSANETWORKEVENTS
{
```

```
    long lNetworkEvent;     /* (FD_CLOSE_BIT 5) (FD_CLOSE 0x20) (FD_MAX_EVENTS
10) (FD_ALL_EVENTS 0x19)*/
    int iErrorCode[FD_MAX_EVENTS]
/* 10050-WSAENETDOWN 网络断开  10053-WSAECONNABORTED 软件造成的连接取消 10054-
WSAECONNRESET 连接被对方重设*/
}WSANETWORKEVENTS, FAR* LPWSANETWORKEVENTS;

 /*检测所指定的套接口上网络事件发生的命令*/
WSAEnumNetworkEvents(SOCKET s, WSAEVENT hEventObject, LPWSANETWORKEVENTS
lpNetworkEvents, LPINT lpiCount);
```

5.4.2 GUI

```
    WM_PAINI                    /* 窗口框架需要绘制消息*/

    HDC hdc;
    PAINTSTRUCT ps;
    TEXTMETRIC tm;      // 字体结构体(大小)
```

```
    GetTextMetrics(HDC, TEXTMETRIC*);       // 获取当前系统字体大小'
    GetSystemMetrics(nIndex);           // 返回各种图形项尺寸信息(鼠标,标题,滚动条...)

'获取DC设备'
    // WM_PAINT 消息使用
    //为指定窗口进行绘图工作的准备，并将绘图有关的信息填充到一个PAINTSTRUCT结构中
    HDC BeginPaint(HWND hwnd, LPPAINTSTRUCT lpPaint);       // 初始化DC设备
    BOOL EndPaint(HWND hWnd, const PAINTSTRUCT* lpPaint);// 标记指定窗口的绘画过程结
束

    /* 获取指定窗口客户端区域或整个屏幕 检索设备上下文(DC)的句柄*/
    HDC GetDC(HWDN);                        // 争对于窗口客户区
    HDC GetDCEx();
    HDC GetWindowDC(HWND hWnd);             // 争对于整个窗口(包含标题栏)
    ReleaseDC(HWND, hdc);                   // 释放DC设备句柄

    Invalidate();                           // 给窗口发送WM_PAINT消息刷新窗口

'获取窗口信息'
    BOOL GetClientRect(HWND hWnd, LPRECT lpRect);   // 获取窗口客户区域的坐标
    BOOL GetWindowRect(HWND hWnd, LPRECT lpRect);   // 获取指定窗口的边界矩形的尺寸
    UINT GetTextAlign(HDC hdc);                     // 获取指定设备上下文的文本对齐设
置

'窗口绘制'
    // 在指定的矩形中绘制格式化文本
    int DrawText(HDC hdc, LPCTSTR lpchText, int cchText, LPRECT lprc, UINT
format);

    // 使用当前选定的字体、背景色和文本颜色在指定位置写入字符串
    BOOL TextOut(HDC hdc, int nXStart, int nYStart, LPCTSTR lpString, int c);
    UINT SetTextAlign(HDC hdc, UINT align);     // 指定的设备上下文设置文本对齐标志

    /* 设置一个窗口的区域*/
    int SetWindowRgn(HWND hWnd, HRGN hRgn, BOOL bRedRaw);

    /* 设置绘制样式*/
```

```
    SelectObject(HDC hdc, CPen* cPen);

    /* 绘制矩形*/
    BOOL Rectangle(HDC hdc, int left, int top, int right, int bottom);
'字体'
```

> 郁金香C++ P95

```
'GDI'
    class CBrush :piblic CGdiObject
    class CPen                      /* 画笔, HPEN*/
    class CBrush                    /* 画刷, HBRUSH */
    class CFont                     /* 字体, HFONT*/
    class CBitmap                   /* 位图, HBITMAP*/
    class CPalette                  /* 调色板, HPALETTE*/
    class CRgn                      /* 区域, CsRGN*/

    BOOL MoveToEx(HDC hdc, int x, int y, LPPOINT lppt);     /* 将当前绘图位置一道到某个点*/
    BOOL LineTo(HDC hdc, int x, int y);                 /* 从当前位置到指定点绘制一条直线*/

'画笔CPen'
    /* nPenStyle:画笔样式  [PS_SOLID 实线] [PS_DOT 虚线]*/
    CPen::CPen(int nPenStyle, int m=nWidth, COLORREF crColor);      /* 画笔类构造函数*/

'画刷CBrush'
    /* 构造函数*/
    CBrush();
    CBrush(COLORREF crColor);           /* 类似于 CreateSolidBrush*/
    CBrush(int nIndex, COLORREF crColor); /* 类似于 CreateHatchBrush*/
    CBrush(CBitmap* pBitmap);           /* 类似于 CreatePatternBrush*/

    /* 类成员函数*/
    HBRUSH CBrush::CreateSolidBrush(COLORREF color);    /* 创建并初始化画刷,指定其颜色*/
    CBeush::CreateHatchBrush();         /* 创建并初始化画刷,指定其颜色和填充样式*/
    CBrush::CreatePatternBrush();       /* 初始化画刷为标准位图样式,指定一个位图句柄*/
    CBrush::CreateBrushIndirect();  /* 创建并初始化画刷,用LOGBRUSH这个结构指定样式(画刷风格,颜色,阴影风格)*/
    CBrush::CreateDIBPatternBrush();    /* 初始化画刷位(DIB位图样式)*/
    CBrush::CreateSysClorBrush();   /* 初始化画刷,用系统默颜色样式,可以指定其画刷风格*/
    CBrush::DeleteObject();     /* 删除画刷对象属性*/

'位图CBitmap' /* 郁金香C++102*/
    /* 构造函数*/
    CBitmap();

    /* 类成员函数*/
    CBitmap::LoadBitmap(lpszResourceName);      /* 通过名字加载位图,需要使用LoadImage先加载位图 */
    CBitmap::LoadBitmap(UINT nIDResource);      /* 从资源加载位图*/
'字体类 CFont' /*郁金香C++ P103*/
    CFont::CFont();
```

```
    /* 初始化成员函数*/
    CFont::CreateFontIndirect(const LOGFONT* lpLogFont);
    CFont::CreateFont();
'区域类 CRgn'
    CRgn::CRgn();

    /* 初始化成员函数*/
    CRgn::CreateRectRgn();          /* 创建一个矩形区域*/
    CRgn::CreateRectRgnIndirect();      /* 创建一个矩形区域,CreateRectRgn区别 通过
RECT结构传参*/
    CRgn::CreateEllipticRgn();          /* 创建一个椭圆区域*/
    CRgn::CreateEllipticRgnIndirect(Rect);
    BOOL CRgn::CreatePolygonRgn(LPPOINT lpPoints, int nCount, int nMode);   /* 创
建一个多边形的区域*/
    CRgn::CreatePolyPolygonRgn();   /* 创建多个多边形的区域*/
    CRgn::CreateRoundRectRgn(LPPOINT lpPoints, LPINT lpPolyCounts, int nCound,
int nPolyFillMode);
                    /* 创建一个圆角矩形区域*/
    ......

    /* 其他类成员函数*/
    CRgn::EqualRgn();               /* 判断两个区域是否相等*/
    CRgn::CombineRgn();             /* 将多个区域合成*/
        int WINAPI CombineRgn(HRGN hrgnDest, HRGN hrgnSrc1, HRGN hrgnSrc2, int
fnCombineMode);

    BOOL CRgn::CreateFromData(const XFORM* lpxForm, int nCount, const RGNDATA*
pRgnData);
        HRGN WINAPI ExtCreateRegion(const XFORM* lpx, DWORD nCount, const
RGNDATA* lpData);
```

```
'MFC封装类CDC绘制,CDC集成了GDI的工具操作'
    /* HDC封装类CDC, 类函数绘制*/
    BOOL CDC::Rectangle(int x1, int y1, int x2, int y2);
    BOOL CDC::Rectangle(LPCRECT lpRect);
    /* CDC获取*/
    CDC* CWND::GetDC();
    CDC* CWND::GetWindowDC();
    /* 释放CDC设备上下文*/
    int CWND::ReleaseDC(HDC hDC);

    /* 将源设备上下文的位图到此当前设备上下文*/
    BOOL CDC::BitBlt(int x, int y, int nWidth, int nHeight, CDC* pSrcDC, int
xSrc, int ySrc, DWORD dwRop);

    CDC::CreateCompatibleDC(CDC* pDc);      /* 创建与pDC指定设备兼容的内存设备上下文*/
    COLORREF CDC::GetPixel(int x, int y)const;  /* 指定的点检索像素的RGB颜色*/
    BOOL CDC::FillRgn(CRgn* pRgn, CBrush* pBrush);  /* 用指定画刷填充指定区域*/

    DWORD GetRegionData(HRGN hRgn, DWORD dwCount, LPRGNDATA lpRgnData); /* 用描述
区域的数据填充指定的缓冲区*/
```

D3D

```
    SetRenderState(D3DRS_ZENABLE, TRUE);        // 渲染设置(深度测试, 开启);

    DrawIndexedPrimitive();
```

5.4.3 剪切板操作

```
'剪切板操作'
    BOOL OpenClipboard(HWND hWndNewOwner);      // 打开剪切板并锁住,防止其他应用打开
    BOOL CloseClipboard(VOID);                  // 关闭剪贴板,这使其他窗口或程序能访问
剪贴板
    BOOL EmptyClipboard(VOID);                  // 清空剪贴板内容(并释放剪贴板内数据句
柄)
    HANDLE SetClipboardData(UINT uFormat,HANDLE hMem);
// 把指定数据按照指定格式放入剪切板, hMem可以是字符串
    HANDLE GetClipboardData(UINT uFormat);      // 以指定格式获取当前剪切板句柄
    BOOL IsClipboardFormatAvailable(UINT);      // 判断剪贴板上数据类型
    UINT EnumClipboardFormats(UINT format);     // 获取剪贴板上当前提供的数据格式
```

5.4.4 全局内存块

```
'全局内存块操作'
    HGLOBAL GlobalAlloc(UINTuFlags, DWORDdwBytes);
        // 全局内存管理函数,该函数从堆中分配内存(需要给该地址赋值使用)
    HGLOBAL GlobalFree( HGLOBAL hMem);          // 释放指定的全局内存块(用于剪切板不能
释放)
    LPVOID GlobalLock(HGLOBAL hMem);            // 获取全局块内存地址并锁定
    BOOL GlobalUnlock( HGLOBAL hMem );          // 解除锁定的内存块, 使指向该内存块的指
针无效
    int GlobalSize(HGLOBAL );               // 获取全局内存大小
```

5.5 Windows MFC

5.5.1 MFC数据类型

```
'CString类'
//  数字转转双字节字符串
        CString test;
        test.Format(TEXT("%d"), pe32.th32ProcessID);
        vItem.pszText = test.GetBuffer( );
// 类函数
    CString::GetBuffer();           /* 获取CString指向的字符串缓冲区,  返回
char*\wchar_t* */
    CString::ReleaseBuffer();       /* GetBuffer()获得Buffer缓冲区使用后释放缓冲区的使
用*/
    CString::Right(int num);        /* 取字符右边N个字符*/
```

```
HGLOBAL

'MFC控件消息事件'
    BN_CLICKED              // 鼠标点击消息
    EN_CHANGE               // 文本框控件数据发生改变时发送该消息
    WM_RBUTTONDOWN          // 鼠标右键消息
    LBN_DBLCLK              // 鼠标双击事件
    NM_DBLCLK               // ListControl列表视图双击勒鼠标左键
    WM_HOTKEY               // 按下通过RegisterHotKey函数注册的热键时发送
```

5.5.2 MFC控件

5.5.2.1 MFC对话框

```
'Dialog'
    HWND (HINSTNCE hInstance,  LPSTR lpTemplate, HWND hWndParent,
                  DLGPROC lpDiglogFunc);       // 创建对话框
    BOOL EndDialog(HWND hDlg, INT_PTR nResult);         // 销毁对话框
```

```
'对话框类属性'
    m_hWnd;                 // 窗口句柄

'MFC资源引用'
/*设置 窗口的图标(在窗口的事件处理函数里 当接收到窗口创建处理消息时载入图标) */
    HICON LoadIcon(HINSTANCE hInstance, LPCSTR lpIconName);     // 加载图标资源
    LRESULT SendMessage(HWND hWnd, UNIT Msg, WPARAM wParam, LPARAM lParam);
```

```
'其他程序应用DLLMFC窗口'
    AFX_MANAGE_STATE(AfxGetStaticModuleState());    /* 用于DLL中所调用MFC函数、类、资源时的模块状态切换*/
```

5.5.2.2 MFC菜单资源

```
'右键窗口菜单'
    CMenu popMenu = LoadMenu(IDR_MENU_ID);      // 加载菜单资源
    popMenu.GetSubMenu(0)->TrackPopupMenu(0, point.x, point.y, this);//弹出菜单(参数1是标志)
        popMenu.GetSubMenu(int len);                    // 获取资源 子项指针

'菜单操作API'
    HMENU GetMenu(HWND hWnd);               // 获取窗口的菜单句柄
    UINT GetMenuState(HWENU hMenu, UINT uld, UINT uFlags);  // 获取菜单的状态

    DWORD CheckMenuItem(hmenu hMenu, UINT uldCheckItem, UINT uCheck);   // 设置菜单项选中状态

    BOOL SetWindowPos(HWND hWnd, HWND hWndInsertAfter,
          int X, int Y, int cx, int cy, UINT uFlags);
            // 设置窗口大小,位置,层次等状态

    BOOL CheckMenuRadioItem(HMEN hMenu, UINT idFirst,
          UINT idLast, UINT idCheck, UINT uFlags);  // 指定菜单项并使其成为一个圆按钮项
```

5.5.2.3 控件类

```
    UpdateData(true);            //用于将屏幕上控件中的数据交换到变量中。
    UpdateData(false);           //用于将数据在屏幕中对应控件中显示出来。

    HWND GetDlgItem(HWND hDlg, int IDC_EDIT_USER);  // 获取控件句柄(窗口句柄,控件ID)
'MFC控件类'
-->Tab Control
    CRect re;                              // 选项卡控件类
    obj.InsertItem( );
    obj.GetClientRect( );
    obj.GetCurSel( );                      // 获取选项卡 被点击项的下标
    obj.GetItemCount( );

'获取控件 类的属性'
    GetClassName(HWND hwnd, LPTSTR lpClassNmae, int nMaxCount);  // 获取指定窗口所
属类的名字
    GetClassInfoEx(HINSTANCE hInstance, LPCSTR lpszClass,
                   LPWNDCLASSEXA lpwcx);        // 获取有关窗口类的信息

'控件函数'
    ::OnInitDialog( );                     // 对话框初始化函数初始化
    obj->SetParent( );                     // 设置控件的父窗口
    obj->ShowWindow( );                    // 显示窗口
    obj->MoveWindwo( );                    // 设置控件的位置

    MessageBox(Handle, TEXT("value"), TEXT("title"), MB_OK)             /* 弹窗
*/
    GetParent();                           // 获取父窗口的类指针
```

```
' 列表控件'
    'List Box(单列表)'
    CListBox* cListBox;
    void ResetContent();                   // 清空组合框内容
    int AddString(LPCTSTR lpszString);  // 添加lpszString 至组合框尾部
    int DeleteString(UINT nIndex);      // 删除nIndex行
    int InsertString(int nIndex, LPCTSTR lpszString); // 在nIndex行插入数据
    int SelectString(int mStartAfter, LPCTSTR lpszString);  // 可以选中包含置顶字符
串的行
    int FindString(int nStartAfter, LPCTSTR lpszString) const;
            /* 可以在当前所有行中查找指定的字符串的位置,nStartAfter知名从哪一行开始进行查找
*/
    int GetCount() const;              // 获取行数
    int GetCurSel() const;             // 获取当前选中行的行号,当前未选中返回-1
    int SetCurSel(int nSelect);        // 设置第n行为当前选中行
    int GetText(int nIndex, CString& rString) const; // 根据索引获得项的文本

    GetWindowText();                   // 获取显示内容

   'List Control(列表视图)'
       /* 图标视图*/
        int InsertItem(int nItem, LPCTSTR lpszItem);        // 插入项目
```

```
        int InsertItem(int nItem, LPCTSTR lpszItem, int mImage);    // 插入项目,带
图标序号
        BOOL DeleteItem(int nItem);
        int GetItenCount();         // 获取项目数
        BOOL SetItemText(int nItem, int nSubItem, LPTSTR lpszText);
        int GetItemText(int nItem, int nSubItem, LPTSTR lpszText, int nLen)
const;
        CString GetItemText(int nItem, int nSubItem) const;
        CImageList* SetImageList(CImageList* pImageList, int nImageList);

        POSITION GetFirstSelectedItemPosition();
                // 返回条目的POSITION值返回NULL表示当前列表视图控件没有条目选中
        int GetNextSelectedItem(POSITION);
                // 返回列表视图控件中下一个被选中的条目索引

'CImageList'

    // 创建图标列表
    BOOL Create(int cHigh, int cWide, UINT nFlags, int nInitial, int nGrow);
        CImageList::Create(32,32, ILC_COLOR32|ILC_MASK, 2, 1);

    // 添加资源图标
    ImageListb.add(AxfGetApp()->LoadIcon(IDR_MAINFRAME));

    // 加载图标
    CWinApp::LoadIcon
        HICON LoadIcon(LPCTSTR lpszResourceName) const;
        HICON LoadIcon(UINT nIDResource) const;

    // 设置图标
    CImageList* SetImageList(CImageList* pImageList, int nImageList);

    // 检索文件信息
    DWORD_PTR SHGetFileInfoW(LPCWSTR pszPath, DWORD dwFileAttributes,
SHFILEINFOW *psfi,
                             UINT cbFileInfo, UINTuFlags);

        {
            // 创建资源图标并加载
            CImageList m_imageList;     // 定义为窗口类的成员变量或者全局变量,生命周期长
的
            m_imageList.Create(32, 32, ILC_COLOR32 | ILC_MASK, 2, 1);    创建图标类
            // 设置列表视图图标
            pLisCtr->SetImageList(&this->m_imageList, LVSIL_NORMAL);
            int iLenItem = 0;
            while()
            {
                SHFILEINFO m_shFileInfo = { 0 };
                SHGetFileInfoW(m_fileFind.GetFilePath(), 0,
                               &m_shFileInfo,sizeof(m_shFileInfo), SHGFI_ICON);
                m_imageList.Add(m_shFileInfo.hIcon);
                this->pClaTree-> SetItemImage (iTreeItem,
                    this->m_imageList.Add(shFileInfo.hIcon), 0);
                // pLisCtr-> InsertItem (iLenItem, m_fileFind.GetFileName(),
iLenItem);
            }
        }
        {
```

```
            // 直接设置资源视图
                m_imageList.Add(AfxGetApp()->LoadIconW(IDR_MAINFRAME));      //指
定图标类
        }

        /* 报表视图*/
        InsertColumn(int nLen, LPCTSTR Name,  int nFormat, int nWidth, int
nSubItem);
                            /* 设置菜单项内容, 添加列*/
        InsertItem(int nItem, LPCTSTR lpszItem, int mImage);     // 添加内容
        SetItemText(int nLen, int column,  LPCTSTR lpszValue);   // 添加报表 列的内
容

        SetExtendedStyle(LVS_EX_FULLROWSELECT);      // 设置选中时,整行选中

        GetWindowLongPtr( );              // 获取报表类控件样式
        SetWindowLongPtr( );              // 设置报表类控件样式
        obj.GetExtendedStyle( );              // 获取控件扩展样式
        obj.SetExtendedStyle( );              // 设置控件扩展样式
            obj.InsertItem( );                    // 添加列表子项标题
            obj.SetItemText( );                   // 设置子项内容

   -> '列表'
        LV_COLUMN lv;            // 列表类
        LV_ITEM vItem;           // 列表项类

        ListView_InsertColumn(hListProcess, 3, &lv);     // 设置菜单项内容
        ListView_SetItem(HWND hwnd, LV_ITEM pitem);      // 设置列表子项的内容
        SendMessage(hListProcess, LVM_SETEXTENDEDLISTVIEWSTYLE,
                    LVS_EX_FULLROWSELECT, LVS_EX_FULLROWSELECT); //设置点击全行选中
```

```
'文本对话框类(编辑框)'
    CEdit cEdit;
    int GetWindowText(HWND hWnd, LPSTR lpString, int nMaxCount);// 通过句柄获取控件
的内容
    BOOL SetWindowText(HWND hWnd,  LPCWSTR lpString);   // 设置文本框控件内容(控件句
柄, 内容)
    HWND GetDlgItem(HWND hwnd, int nIDDlgItem);          // 从指定的对话框中获取控件的
句柄
    // CWnd类对窗口句柄线管函数进行封装 可以在主窗口初始化函数里直接调用类函数
    //this->GetDlgItem(IDC_EDIT1)->SetWindowTextW(L"0.");  (控件ID)--（初始化的文
本）\
    edit->GetSel(nStart,nEnd);           // 获取文本框当前选中的起始位置和结束未知
    edit->SetSel(nStart,nEnd);           // 设置文本框选中的起始位置和结束未知
    edit->GetWindwoText(wchar_t* pBuf, size_t size);     // 获取文本框的内容
```

```
'菜单类'
    CMenu cmenu;
    cmenu.LoadMenuW(IDD_ID);
     CMenu* obj.GetSubMenu(int);                 // 获取菜单索引为*的菜单栏 返回指向指
针

    obj->TrackPopupMenu( );                  // 显示菜单
```

```
'组合框'
    // CComboBox: 组合框类函数
    CComboBox* cComBox;
    void ResetContent();                // 清空组合框内容
    int AddString(LPCTSTR lpszString);  // 添加lpszString 至组合框尾部
    int DeleteString(UINT nIndex);      // 删除nIndex行

    // ... 同上,引用单列表框类函数(获取行 内容函数不同)
    int GetLBText(int nIndex, LPCTSTR lpszText) const;  // 获取第n行的内容保存在str
    void GetLBText(int nIndex, CString& rString) const; // 获取第n行的内容保存在str
```

```
'滑块控件(音视频式进度条)'
    CSliderCtrl* cCSliderCtrl;
        /* 常用类函数*/
        GetLineSize();              // 返回滑块常用步长
        SetLineSize();              // 设置滑块控件移动步长,针对 光标键↑↓
        GetPageSize();              // 返回滑块常用步长
        SetPageSize();              // 设置滑块控件移动步长,针对 PageDown和PageUP及鼠
标双击
        GetRange();                 // 返回滑块刻度(可移动范围)的最大值和最小值
        GetRangeMax();              // 返回滑块刻度(可移动范围)的最大值
        GetRangeMin();              // 返回滑块刻度(可移动范围)的最小值
        SetRange(int nMin, int nMax);   // 设置滑块刻度(可移动范围)的范围
        SetRangeMin(int nMin);      // 单独设置滑块刻度(可移动范围)最小值
        GetPos()                    // 获取当前滑块位置
        SetPos(int);                // 设置滑块在刻度线的位置
        SetTicFreq(int);            // 设置滑块控件的刻度线间隔
'进度条控件(文件下载式进度条)'
    CProgressCtrl* cProg;
    OffsetPos(int);             //在原有的基础上增加n
    SetStep(int);               // 设置每次增量的数值,只对StepIt函数增量适用
    StepIt();                   // 在当前位置增加n个距离,使用SetStep设置步长
    // 常用类函数和滑块控件一样
'数字调节(旋转)控件'
    CSpinButtonCtrl* pSpin;
    /* 常用类函数*/
    CSpinButtonCtrl::SetAccel();        // 为spin按钮设置一个加速值
    CSpinButtonCtrl::GetAccel();        // ......
    CSpinButtonCtrl::SetBase();     // 设置旋转按钮控件的基数,这个基数决定伙伴窗口显示的
是十进制还是
                                    // 十六进制,十进制有符号,十六进制无符号
    CSpinButtonCtrl::GetBase();         //....
    CSpinButtonCtrl::GetBuddy();        // 获取spin按钮的伙伴窗口
    CSpinButtonCtrl::SetBuddy();        // 为spin按钮设置伙伴窗口,点击spin时焦点会落到
伙伴窗口上
    CSpinButtonCtrl::SetPos();          // 为spin控件设置一个当前位置
    CSpinButtonCtrl::GetPos();          // 获取.....
    CSpinButtonCtrl::SetRange();        // 设置范围值
    CSpinButtonCtrl::GetRange();        // ......
    CSpinButtonCtrl::SetRange32();      // 设置32位的范围数值
    CSpinButtonCtrl::GetRange32();      // .......
```

```
'选项卡控件 Tab Control'
    CTabCtrl* pTab;
    // 事件消息
    TCN_SELCHANGE                   // 选中项 发生改变时
```

```
    // 常用类函数
    void InserItem(int nItem, LPWCSTR lpwch);       // 在nItem的位置插入选项
    DeleteItem();                                   // 移除某个选项卡
    DeleteAllItems();                               // 移除所有选项卡
    int GetCurSel();                                // 获取当前选中项的下标
    SetCurSel();                                    // 设置....
    // 设置选项窗口
        /* ---设置子窗口的父窗口为选项卡控件*/
        dlg_xxxx->SetParent(pTab);
        /* 设置子窗口相对于父窗口的位置*/
        RECT r1, r2, r3;
        CTabCtrl::GetWindowRect(&r1);           // 获取选项卡控件在屏幕的位置
        CTabCtrl::GetItemRect(0, &r2);          // 获取选项卡子项0 相对于选项卡控件的
位置
        r3.left = 1;                            // 设置左边距离 选项卡控件距离
        r3.top = r2.bottom+1;                   // 设置上边距在 选项卡控件子项按钮下边距+1
的位置
        r3.right = r1.right-r1.left-3;          // 设置右边位置为选项卡右边相对于屏幕距离-
左边屏幕距离
        r3.bottom = r1.bottom- r1.top-3;        // 设置下边位置为选项卡控件相对于屏幕
        m_newDlg.MoveWindow(&r3);
```

```
'文件类'
    class CFile;

    /* 类成员函数*/
    CFile::open();
    CFile::Close();
```

```
'查找文件'
    /* CFileFind类*/
    // 类函数
    int CFileFind::FindFile(LPCTSTR Filepath);      // 搜索目录,成功返回非0,否则返回0
    CFileFind::Close();                         // 关掉搜索请求,释放占用资源
    CFileFind::FindNextFile();                  // 继FindFile后查找下一个文件,最后一
个文件时返回
    CFileFind::GetFileName();                   // 获取文件名字
    CFileFind::GetFilePath();                   // 获取文件路径+文件名字
    CFileFind::GetFileTitle();                  // 获取文件标题
    CFileFind::GetRoot();                       // 获取文件路径

    CFileFind::IsReadOnly();                    // 如果是只读的,返回非0
    CFileFind::IsDots();                        // 判断是否是 ../或者./（当前目录,上
一级目录）
    CFileFind::IsDirectory();                   // 如果是一个目录,返回非0的值
    CFileFind::IsCompressed();                  // 如果是一个压缩文件,返回非0的值
    CFileFind::IsSystem();
    CFileFind::IsHidden();                      // 如果是隐藏的,返回非0
    CFileFind::IsTemporary();                   // 临时文件或者文件夹,返回非0

    WINSHELLAPI DWORD WINAPI SHGetFileInfo(LPCTSTR pszPath, DWORD
dwFileAttributes,
                SHFILEINFO FAR *psfl, UINT cbFileInfo, UINT uFlags);   // 获取文
件信息
```

```
'文件对话框类'
    CFileDialog::CFileDialog();          // 构造函数
        CFileDialog  OpenDlg(TRUE, NULL, NULL,  OFN_HIDEREADONLY |
OFN_OVERWRITEPROMPT,
                             L"所有|*.*||");       // 定义文件对话框类
    // 类成员函数
    ::DoModal();                    // 以模态对话框的方式显示
    ::GetPathName();                // 获取选中文件的路径
    //
    {
            if(OpenDlg.DoModal() == IDOK)
            {
                OpenDlg.GetPathName();
                HBITMAP hb =(HBITMAP)LoadImage();        // 加载突变类型
                HCURSOR hb=(HCURSOR)LoadImage();         // 加载指针类型
            }
    }
```

```
'热键控件'
    ChotKeyCtrl* hotk;
    // 类函数
    void ChotKeyCtrl::SetHotKey(WORD wVirtualKeyCode, WORD wModifiers);

    DWORD ChotKeyCtrl::GetHotKey() const;   /* 获取热键数据*/
    void ChotKeyCtrl::GetHotKey(WORD &wVirtualKeyCode, WORD &wModifiers) const;
            /* 获取热键数据*/
            /* wVirtualKeyCode是后置键(字母数字键)，wModifiers是前置键
(CTRL/SHFIT/ALT...)*/

    // 系统API 不是类函数
    BOOL RegisterHotKey(HWND hWnd, int id, UINT fsModifiers, UINT vk);
        // 注册系统热键(热键ID自定义)
    void SetRules(WORD wInvalidComb, WORD wModifiers);          // 设置失效热键组合
```

```
'图片控件 Picture Control'
   CStatic* stat;
    CStatic::SetBitmap();                                   // 装载图片
    CStatic::GetBitmap();                                   // 获取图片
    CStatic::SetIcon();
    CStatic::GetIcon();
    CStatic::SetCursor();
    CStatic::GetCursor();
    CStatic::SetEnhMetaFile();
    CStatic::GetEnhMetaFile();

    HANDLE LoadImage(HINSTANCE hInst, LPCSTR name, UINT type, int cx, int cy,
                     UINT fuLoad);                          // 加载图片

    IMAGE_CURSOR::SetCursor();
    IMAGE_ICON::SetIcon();

    DeleteObject();     // 释放Bitmap资源
    Cursor();           // 释放DestroyCursor资源
    DestroyIcon();      // 释放Icon资源
```

```
'树状控件 Tree Control'
    CTreeCtrl* ptree;
    // 类函数
    HTREEITEM CTreeCtrl::InsertItem(LPCWSTRitemName);   // 添加项(根项)，返回根项的句
柄

    // 在根项插入子项
    CTreeCtrl::InsertItem(LPCWSTR itemName, HTREEITEM PrentHwnd, TVI_SORT);
        // TVI_ROOT TVI_ROOT TVI_LAST TVI_SORT 插入方式

    CTreeCtrl::DeleteAllItems(void) const;      // 清空所有项
```

```
'Animation Control 动画播放控件'
    CAnimateCtrl* pAnimate;
    // 类成员函数
    CanimateCtrl::Open();                                   // 打开加载AVI视频
    CAnimateCtrl::Play();                                   // 播放没有声音的AVI
    CAnimateCtrl::Seek();                                   // 移动到相应的帧
    CAnimateCtrl::Stop();                                   // 停止播放
    CAnimateCtrl::Close();                                  // 关闭播放
```

```
'IP地址控件 IP Address Control'
    CIPAddressCtrl* pIPAddre;
    // 类成员函数
    bool CIPAddressCtrl::IsBlank();                         // 判断控件是否为空(空返回真)
    void CIPAddressCtrl::ClearAddress();                    // 清空地址
    int CIPAddressCtrl::GetAddress();                       // 获取控件里显示IP地址
    void CIOAddressCtrl::SetAddress();                      // 设置控件IP地址
    void CIPAddressCtrl::SetFieldFocus();                   // 设置控件某一个单元获取焦点
    void CIPAddressCtrl::SetFieldRange();                   // 设置控件某一个单元的数值范
围
```

5.5.2.4 MFC动态控件

```
'动态创建文本框'
    CEdit mEdit;
        mEdit.CWnd::Create();
        virtual BOOL Create(LPCTSTR lpszClassName, LPCTSTR lpszWindowName,
                    DWORD dwStyle, Const RECT& rect, CWnd* pParentWnd, UINT nID,
                    CCreateContext* pContext = NULL);

        mEdit.CWnd::Create();
        BOOL CreateEx(DWORD dwExStyle, LPCTSTR lpszClassName, LPCTSTR
lpszWindowName,
                    DWORD dwStyle, Const RECT& rect, CWnd* pParentWnd, UINT nID,
                    CCreateContext* pContext = NULL);

'动态控件绑定窗口事件' /*郁金香c++ 88课*/
    WNDPROC oldProc;        // 窗口过程结构体(消息回调函数)
    /* 获取指定窗口信息，  (获取窗口过程)*/
    long GetWindowLong(HWND hWnd, int nIndex);
    /* 设置窗口信息, (设置窗口过程)*/
    LONG SetWindowLong(HWND hWnd, int nIndex, LONG dwNewLong);/*dwNewLong消息回调
函数地址*/
    /* 在新消息回调函数return的地方调用旧的窗口过程(GetWindowLong获取的窗口过程)*/
```

```
    LPESULT CallWindowProc(WNDPROC lpPrevWndFunc, HWND hWnd, UINT Msg, WPARAM
wParam,
                           LPARAM LParam);
```

```
'计时器' /* 郁金香c++ 90课*/
    UINT WINAPI SetTimer(HWND hWnd, UINT nIDEvent, UINT uElapse, TIMERPROC
lpYimerFunc);
    UINT CWnd::SetTimer(UINT nIDEvent, UINT nElapse, void(CALLBACK EXPORT*
lpfnTimer(HWND,UINT,UINT,DOWRD)) );

    /* 计时器的回调函数*/
    VOID CALLBACK TimerProc(HWND hWnd, UINT uMsg, UINT idEvent, DWORD dvTime);

    /* 销毁计时器*/
    BOOL WINAPI KillTimer(HWND hWnd, UINT_PTR uIDEvent);
    BOOL CWnd::KillTimer(UINT_PTR nIDEvent);
```

5.6 Windows套接字

> 郁金香C++ P113

```
'库文件'
   #include <winsock2.h>
    ws2_32.lib                                          /* 库文件*/

'套接字结构'

    SOCKET                                          /* 套接字句柄*/

    struct sockaddre{ };                                /* 常规套接字结构*/
    struct sockaddr_in{ };                              /* TCP/IP/IPV4 */
```

```
'WINAPI'
    int WSAStartup(Word vVersionRequested, LPWSADATA lpWSAData);        /*
socket字库绑定(绑定)*/
    int WSACleanup();                                                   /* 卸载套接字库
*/

    SOCKET WSAAPI socket(int af, int type, int protocol);           /* 创建绑定特定
传输服务的套接字*/
    int closesocket(SOCKET s);                                      /* 关闭套接字*/

    int bind(SOCKET s, const sockaddr* addr, int namelen);  /* 将套接字与绑定本地绑
定(sockaddr套接字结构)*/
    int WSAAPI listen(SOCKET s, int backlog);           /* 绑定要监听的端口*/

    u_short htons(u_short hostshort);               /* 将本地字节序转换为TCP/IP网络字
节序*/
    u_long htonl(u_long hostlong);

    inet_addr(IN const char* cp);               /* inet_addr("127.0.0.1"),字符串转
换IP地址*/
```

```
    int WSAAPI Send(SOCKET s, const char* buf, int len, int flags);      /* 发送消息*/
    int WSAAPI recv(SOCKET s, char* buf, int len, int flags);
/* 接收数据*/

    /* 服务端*/
    SOCKECT WSAAPI accept(SOCKET s, _OUT_ sockaddr* addr, int* addrlen);
/* 等待客户端进行连接*/

    /* 客户端*/
    int WSAAPI connect(SOCKET s, const sockaddr* name, int namelen);
/* 连接客户端*/
```

```
'MFC套接字'
    class CSocket: public CAsyncSocket;                        /* 套接字类*/
    /* 类函数*/
    BOOL CWinApp::AfxSocketInit(WSADATA* lpwsaData=NULL);/* 加载套接字字库,环境初始化,类结束时自动卸载字库*/
    BOOL CSocket::Create(UINT nSocketPort=0, int nSocketType=SOCK_STREAM,
LPCTSTR lpszSocketAddress=NULL);                                    /* 创建套接字对象*/
    CSocket::Listen();
    CAsyncSocket::OnAccept(int nErrorCode);          /* 基于消息驱动,单独的线程等待客户端消息*/
```

6 Windows内核

6.1 WinDbg调试

```
'WinDbg调试命令'
    dd 8003f000          // 以4个字节查看某个地址的数据
    dq 8003f000          // 以8个字节查看某个地址的数据
    eq 8003f000 00~00    // 写入8字节

    r gdtr               // 查看GTDR寄存器中存放的 GDT表地址
    r gdtl               // 查看GTDR寄存器中存放的 GDT表的大小

    dt struct            // 查看内核中的结构体

    rdmsr 174            // 查看CS

    F5 g                     // go 继续运行
    u Driver!DriverEntry     // 汇编查看函数
    F8 F11               // 单步步入
    F10                  // 单步步过
    Shift+F11                // 返回到上一层函数
    bp Driver!function       // 下 int3 断点
    bl                   // 显示断点列表
    bc  num              // 清除指定断点
    bd  num              // 禁用指定断点
    be  num              // 启用指定断点

    a                // 进入修改指令界面
```

```
    !process 0 0
    .process id         // 使用某个进程的上下文
```

```
/* 数据类型*/
    BYTE    (unsigned char)字节        字节  0-0xff
    WORD    (unsigned short)字         2字节 0-0xffff
    DWORD   (unsigned long)双字        四字节 0-0xffffffff
    FWORD                             6字节
    QWORD                             8字节

    byte字节 [8 Bit] 1字节 <  WORD字 [16Bit] 2字节 < DWROD双字 [32Bit] 4字节
```

6.2 保护模式

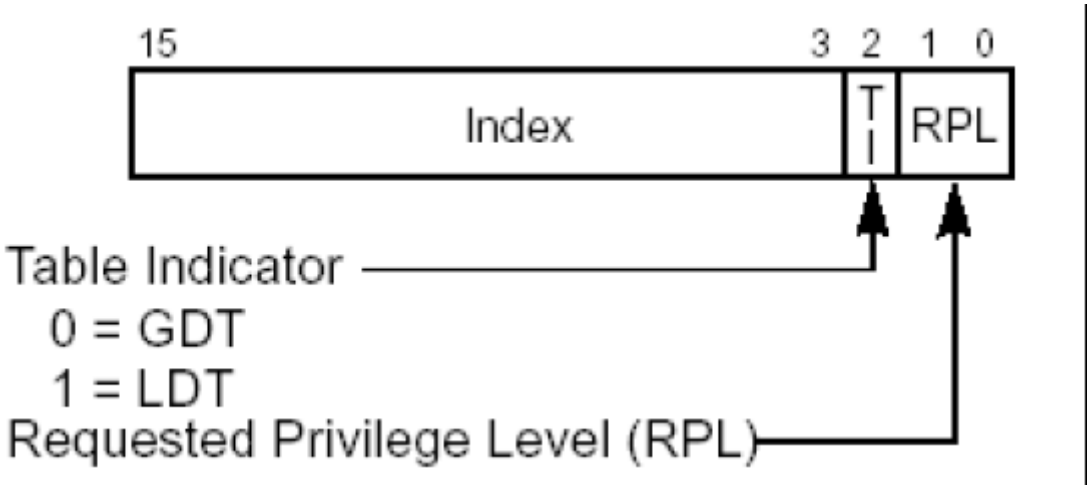
6.2.1 段寄存器

> 段寄存器:`ES`,`CS代码段`,`SS`,`DS`,`FS`,`GS`,`LDTR`,`TR`.共8个
>
> 全局描述符表`GDT`,局部描述表`LDT`
>
> `GTDR寄存器48位大小`, GTD表存放地址[32位] 和GTD表大小[16位] ()

```
'段寄存器结构'
    struct SegMent  /* 96位大小*/
    {
        WORD Selector;      // 16位,段选择子(可见)
        WORD Attributes;    // 16位属性
        DWORD Base;         // 32位Base段基址
        DWORD Limit;        // 32位Linit段限长
    }

'段寄存器成员'
    /* Selector,FS:Base可能不同*/
    ES  Selector(002B)  Attributes(可读可写)    Base(0)         Limit(0xffffffff)
LES
    CS  Selector(0023)  Attributes(可读可执行)   Base(0)         Limit(0xffffffff)
    SS  Selector(002B)  Attributes(可读可写)    Base(0)         Limit(0xffffffff)
LSS
    DS  Selector(002B)  Attributes(可读可写)    Base(0)         Limit(0xffffffff)
LDS
    FS  Selector(0053)  Attributes(可读可写)    Base(0x7FFDE000)    Limit(0xFFF)
LFS
    GS  Selector(002B)  Attributes(---)     Base(---)           Limit(---)
LGS

    /* CS一般存放的代码段, CS的改变意味着EIP的改变*/
        LTR /* 装载TSS修改TR寄存器*/
        STR /* 读TR寄存器,只读了TR的16位,也就是选择子*/


'会同时修改EIP和CS段寄存器的指令'
    jmp far   call far   retf    int    ireted
    JMP FAR 0040:0041840D
```

```
_asm sgdt a;                # 读取GDT寄存器,3环也可以用
```

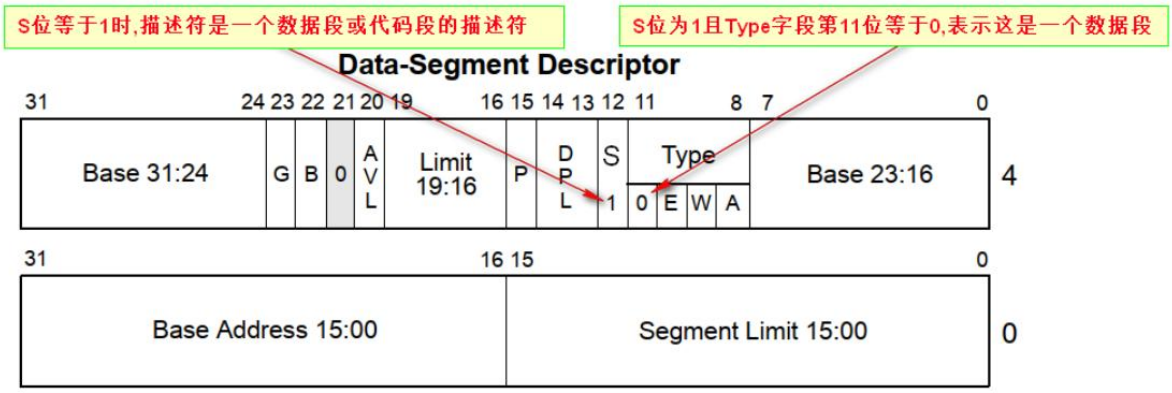
6.2.2 段选择子



6.2.3 GDT

> GDT-全局描述表
>
> 数据段
>
> 当前程序在CPU下的特特权级别(CPL)： CS 段选择子,当前程序运行权限
>
> 描述符特权等级(DPL)位于段描述符高4字节的13-14位,访问描述符所具备的权限(一致代码段例外)
>
> 请求特权级别(RPL)访问段描述符时的请求等级
>
> CS和SS的段选择子最后两位总是相同的

```
 '段描述符'
    /* 加载段描述符至段寄存器(代替mov指令加载段描述符) RPL <=DPL */
'GDT全局描述符'
Attributes: //从第8位到23位,16个字节
    G位       // 位于高4个字节第23的位置G=0,Limit单位为字节最大0xFFFFF,G=1,单位是4KB,后面默认有0xFFF
    P位       // 位于高4个字节第15的位置,P=1段描述符有效,P=0段描述符无效(大于0x8有效)
    DPL      // 访问描述符所具备的权限
    S位       // 位于高4个字节第12位,S=1时,描述符是代码段或者数据段描述符. 当S=0是系统段描述符(0xf/0x9)
    Type域    // S=1时,第十一位是1是代码段否则是数据段(大于0x8是代码段)
    D/B位     // 位于高4字节第二十二位,
Base: //高4字节0位-7位和24-31位,低四字节16位-31位

Limit: //高四字节第16位-19位,低四字节第0位-15位
```



6.2.3.1 数据段

> 数据段的权限检查: `CPL<=DPL 并且 RPL<=DPL`
>
> 1) 代码间的跳转要修改CS时: 先得到段描述符, 判断查`GDT`还是`LDT`,
>
> 2) 判断是否是`代码段`,`调用门`,`TSS任务段`,`任务门`
>
> 3) 如果是代码段,之后代码段权限检查`非一致代码段CPL==DPL`并且`RPL<=DPL`, `一致代码段CPL>=DPL`
>
> /* 一致代码段修饰 允许低权限程序(应用层)直接访问高权限代码段(内核) */
>
> 4) 通过后CPU会将描述符加载至CS段寄存器, 将CS.Base+Offset写入EIP,执行EIP

表 4-3 代码段和数据段描述符类型

类型（TYPE）字段					描述符类型	说明
十进制	位 11	位 10	位 9	位 8		
		E	W	A		
0	0	0	0	0	数据	只读
1	0	0	0	1	数据	只读，已访问
2	0	0	1	0	数据	可读/写
3	0	0	1	1	数据	可读/写，已访问
4	0	1	0	0	数据	向下扩展，只读
5	0	1	0	1	数据	向下扩展，只读，已访问
6	0	1	1	0	数据	向下扩展，可读/写
7	0	1	1	1	数据	向下扩展，可读/写，已访问
		C	R	A		
8	1	0	0	0	代码	仅执行
9	1	0	0	1	代码	仅执行，已访问
10	1	0	1	0	代码	执行/可读
11	1	0	1	1	代码	执行/可读，已访问
12	1	1	0	0	代码	一致性段，仅执行
13	1	1	0	1	代码	一致性段，仅执行，已访问
14	1	1	1	0	代码	一致性段，执行/可读
15	1	1	1	1	代码	一致性段，执行/可读，已

6.2.3.2 系统段

Table 3-2. System-Segment and Gate-Descriptor Types

Type Field					Description
Decimal	11	10	9	8	
0	0	0	0	0	Reserved
1	0	0	0	1	16-Bit TSS (Available)
2	0	0	1	0	LDT
3	0	0	1	1	16-Bit TSS (Busy)
4	0	1	0	0	16-Bit Call Gate
5	0	1	0	1	Task Gate
6	0	1	1	0	16-Bit Interrupt Gate
7	0	1	1	1	16-Bit Trap Gate
8	1	0	0	0	Reserved
9	1	0	0	1	32-Bit TSS (Available)
10	1	0	1	0	Reserved
11	1	0	1	1	32-Bit TSS (Busy)
12	1	1	0	0	32-Bit Call Gate
13	1	1	0	1	Reserved
14	1	1	1	0	32-Bit Interrupt Gate
15	1	1	1	1	32-Bit Trap Gate

6.2.3.3 调用门

> 调用门执行流程 `CALL CS:EIP(EIP是废弃的)`
>
> 1) 根据CS的值查GDT表,对应的段描述符表, `这个描述符是一个调用门`
>
> 2) 在调用门描述符中存储另一个代码段 段的选择子
>
> 3) 选择子指向的段 段.base+偏移地址就是真正要执行的地址
>
> 门描述符 `S位是0  TYPE域是1100`

在调用门描述符中，定义了**目标过程（例程）所在代码段的选择子**，以及**段内偏移量**

高32位

31 ~ 16	15	14 ~ 13	12	11 ~ 8	7	6	5	4 ~ 0
段内偏移量 31~16	P	DPL	0	TYPE: 1 1 0 0	0	0	0	参数个数

低32位

31 ~ 16	15 ~ 0
例程所在代码段选择子	段内偏移量 15~0

调用门描述符的格式

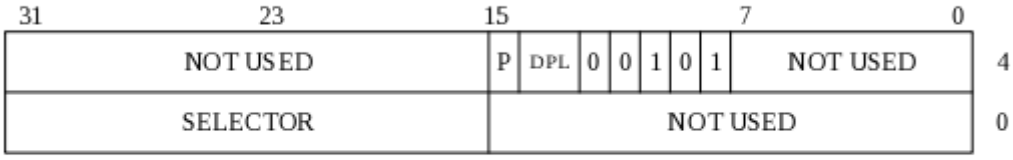

```
#长调用
    # 跨段不提权    /* 发生改变的寄存器,ESP, EIP, CS */
    CALL FAR CS:EIP(EIP是废弃的) --> ESP-8   --> [ESP+4]=返回地址, [ESP+8]=调用者CS
    RETF
```

```
#跨段提权(R3层执行R0层)  /* 发生改变的寄存器, ESP, EIP, CS, SS */
CALL FAR CS:EIP(EIP是废弃的) --> ESP=? --> [ESP+4]=返回地址, [esp+8]=调用者CS,
                                    [esp+c]=调用者ESP, [ESP+0x10] =调用者ss
RETF

/*
跨段调用时,一旦有权限切换,就会切换堆栈
CS的权限一旦改变,SS的权限也要随着改变,CS和SS的等级必须一样
JMP FAR只能跳转到同级非一致代码段,但CALL FAR可以通过调用门提权,提升CPL的权限
*/
```

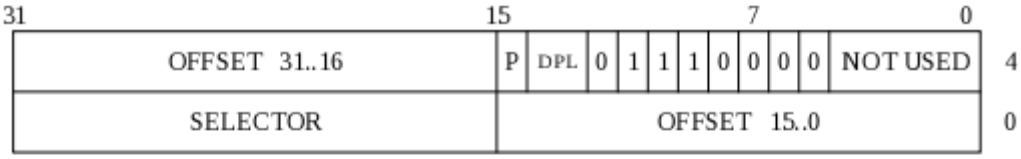
6.2.4 IDT

> IDT-中断描述符表
>
> IDT中断描述符表包含三种门描述符: ---中断门、陷阱门 与调用门类似
>
> 任务门描述符(TASK GATE), 中断门描述符(INTERRUPT GATE), 陷阱门描述符(TRAP GATE)

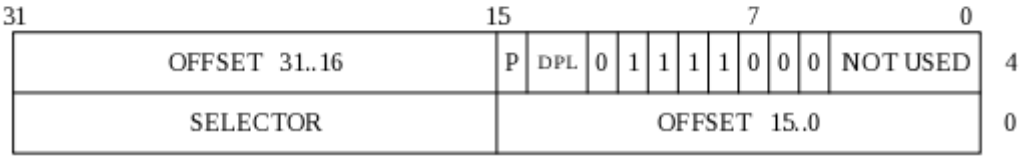
80386 TASK GATE



80386 INTERRUPT GATE



80386 TRAP GATE



6.2.4.1 中断门

6.2.4.2 陷阱门

6.2.4.3 任务门

> 任务门执行流程:
>
> 1): INT n中断号
>
> 2): 查找IDT表,找到`中断门描述符`
>
> 3): 通过中断门描述符,查GDT表,找到`任务段描述符`
>
> 4): 使用TSS段中的值修改各个寄存器
>
> 5): IRETD 返回
>
> `TSS在内存中,104个字节,里面存放了寄存器的值`

6.2.4.4 TSS段描述符和TSS任务段

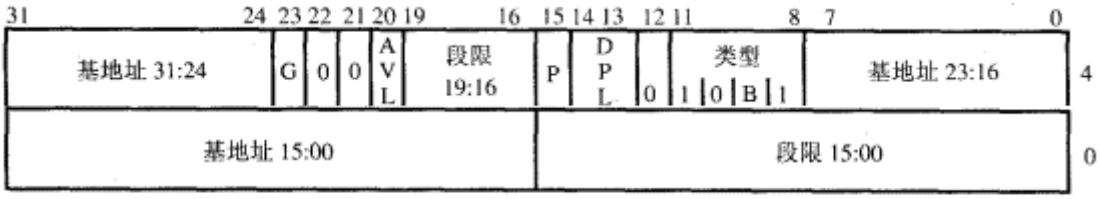


图 5．2　TSS 描述符的格式

31 15 0		
I/O Map Base Address	T	100
	LDT Segment Selector	96
	GS	92
	FS	88
	DS	84
	SS	80
	CS	76
	ES	72
EDI		68
ESI		64
EBP		60
ESP		56
EBX		52
EDX		48
ECX		44
EAX		40
EFLAGS		36
EIP		32
CR3 (PDBR)		28
	SS2	24
ESP2		20
	SS1	16
ESP1		12
	SS0	8
ESP0		4
	Previous Task Link	0

6.2.4.5 TR寄存器

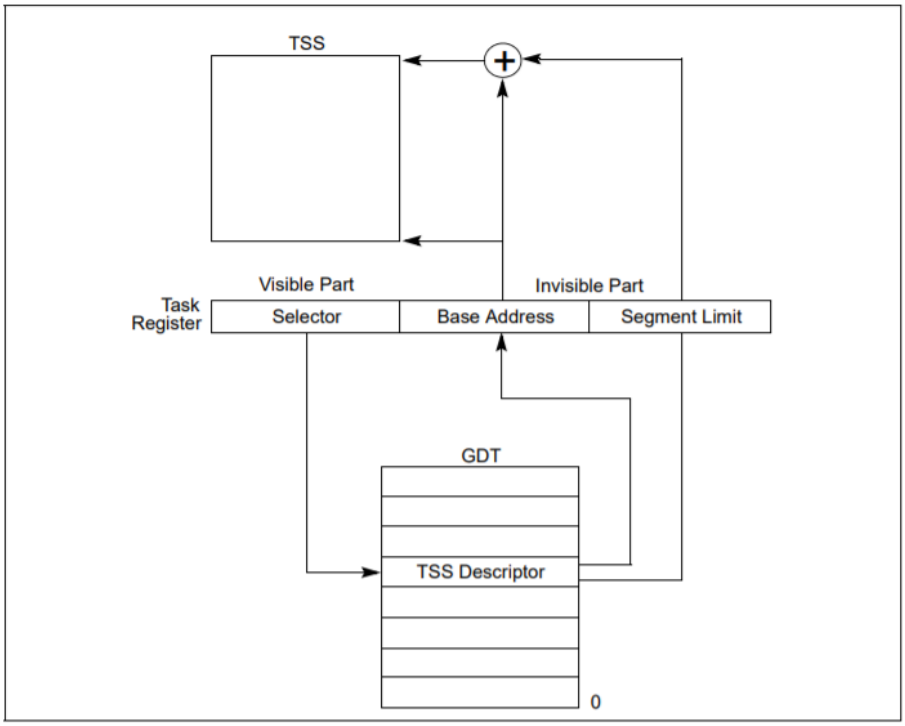


Figure 7-5. Task Register

6.2.5 SSDT

> `KeServiceDescriptorTable (SSTD)`：系统描述符表
>
> `KeServiceDescriptorTableShadow (SSDT Shadow)`

```
typedef struct _KSYSTEM_SERVICE_TABLE    /* SSDT表项*/
{
    PULONG ServiceTableBase;             // 服务函数地址表基址
    PULONG ServiceCounterTableBase;
    ULONG NumberOfService;               // 服务函数个数
    PULONG ParamTableBase;               // 服务函数参数表基址
}KSYSTEM_SERVICE_TABLE, *PKSYSTEM_SERVICE_TABLE;

typedef struct _SERVICE_DESCRIPTOR_TABLE
{
    SYSTEM_SERVICE_TABLE ntoskrnel; // ntoskrnl.exe的服务函数
    SYSTEM_SERVICE_TABLE win32k;    // win32k.sys的服务函数(gdi.dll/user.dll的内核支
持)
    SYSTEM_SERVICE_TABLE NotUsed1;
    SYSTEM_SERVICE_tABLE NotUsed2;
}SYSTEM_DESCRIPTOR_TABLE, *PSYSTEM_DESCRIPTOR_TABLE;

typedef struct ServiceDescriptorTable
{
    unsigned int *ServiceTableBase;             // SSDT表的基地址
    unsigned int *ServiceCounterTable(0);//包含着 SSDT 中每个服务被调用次数的计数器,由
KiSystemService更新
    unsigned int NumberOfServices;      // SSTD表服务的数目
    unsigned char *ParamTableBase;              // 包含每个系统服务参数字节数表的基地址
}ServiceDescriptorTableEntery_t, *PServiceDescriptorTableEntry_t;
```

6.2.5.1 中断和异常

错误类型	IDT表中断号
页错误	0xE
段错误	0xD
处零错误	0x0
双重错误	0x8

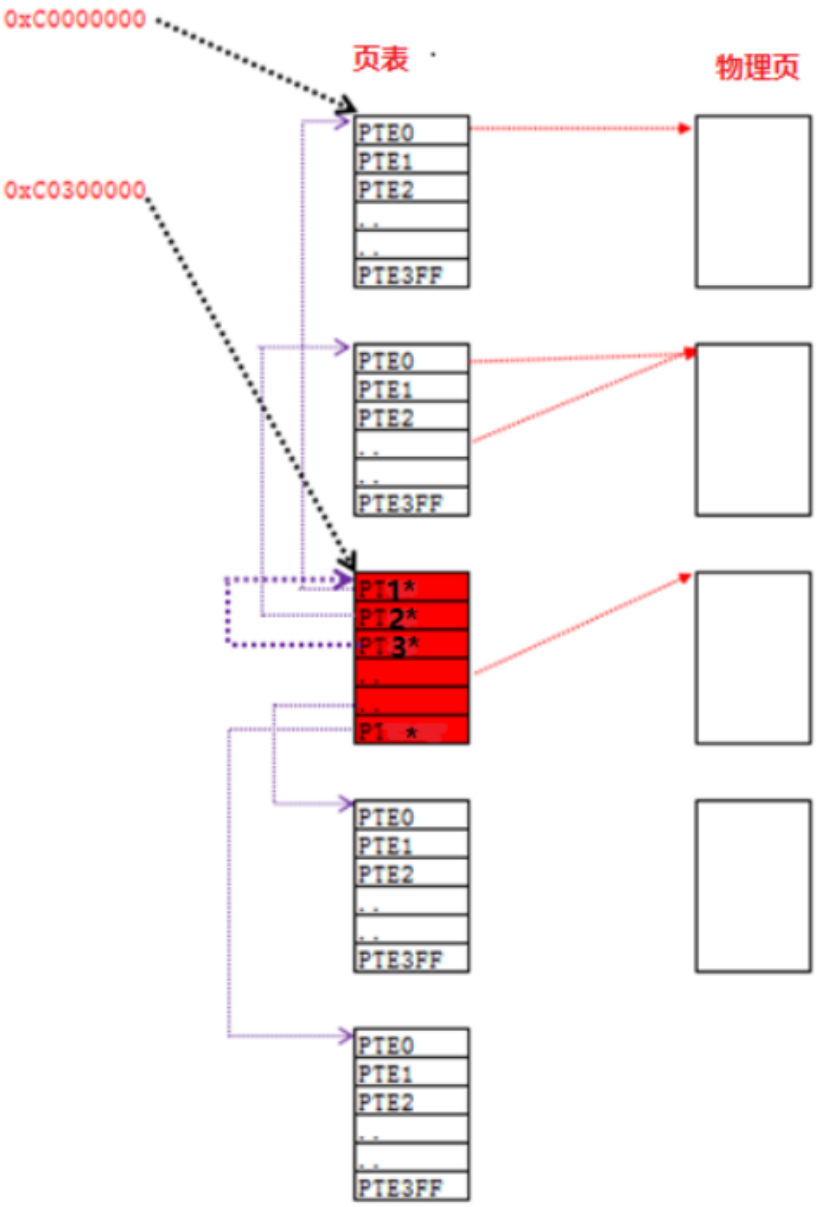
6.2.5.2 控制寄存器(CR)

6.2.6 内存分页

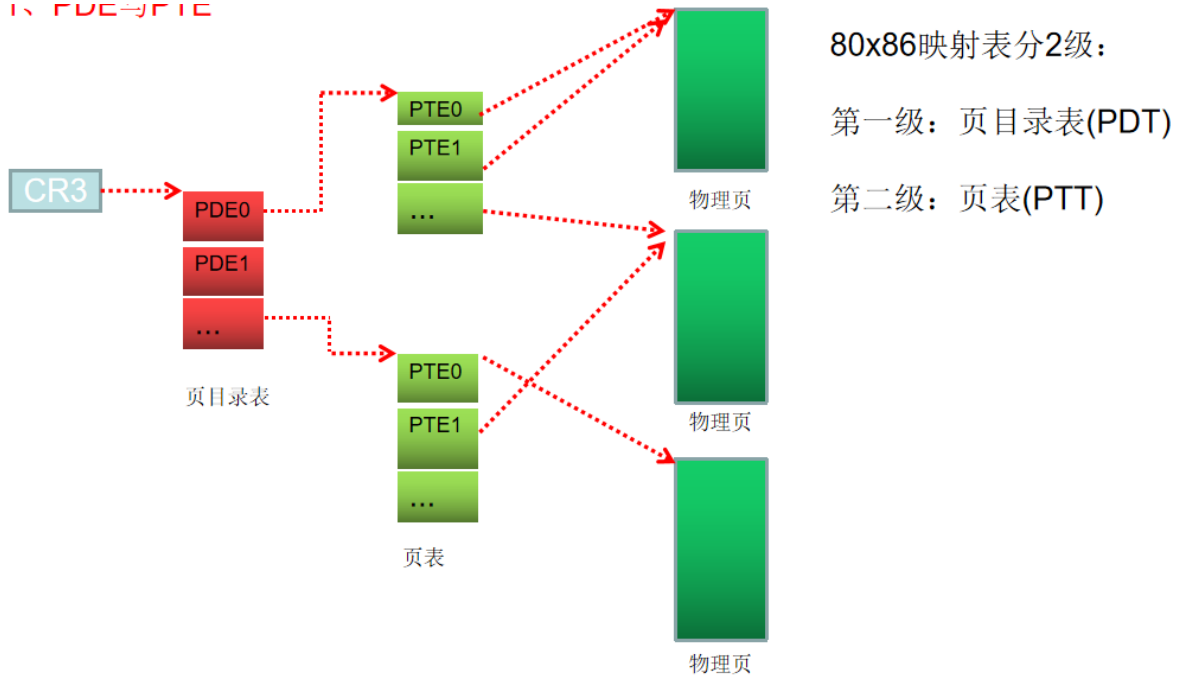
> MOV eax, dword ptr ds:[0x12345678]
>
> 有效地址: 0x12345678
>
> 线性地址: ds.base+0x12345678
>
> 物理地址: 映射的物理页地址

6.2.6.1 [10-10-12]分页

> `ntkrnlpa.exe`
>
> 10-10-12: `PDT-PTI-物理页偏移`
>
> 页目录表(PDT)基址:`C0300000`
>
> 访问页目录的公式:`0xC0300000+ PDI*4`
>
> 访问页表公式:`0xC0000000+ PDI*4096+ PTI*4`



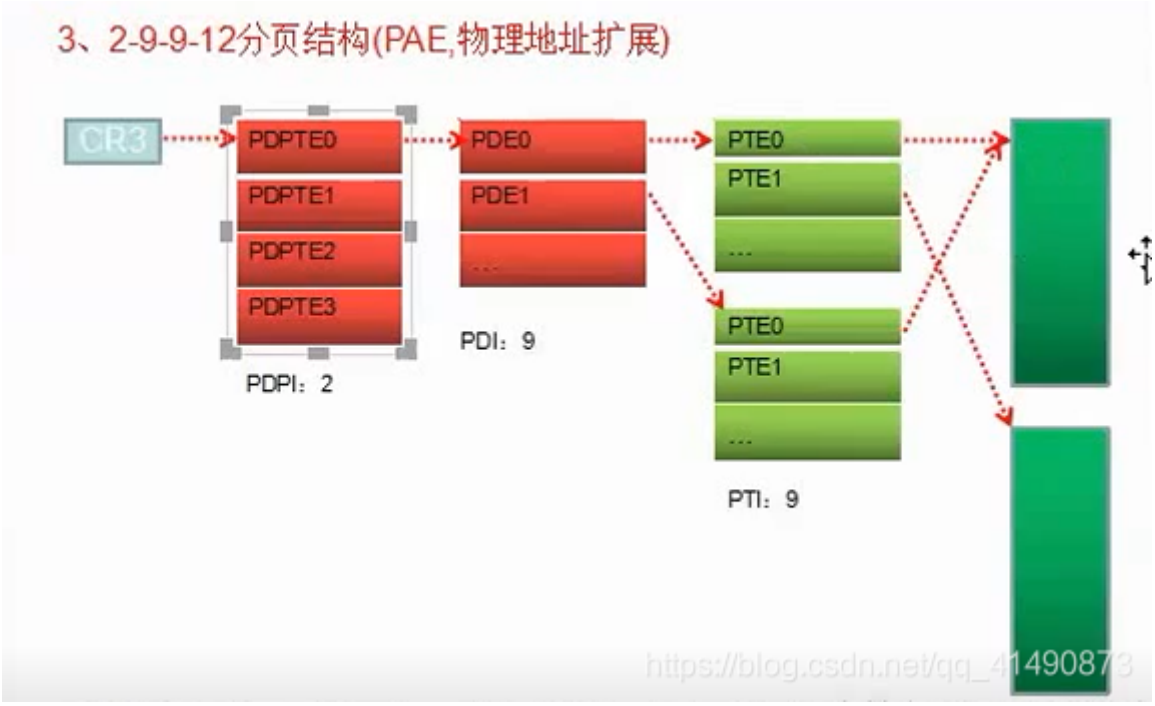
PDE和PTE属性




```
# P：有效位。0 表示当前表项无效。
# R/W： 0 表示只读。1表示可读可写。
# U/S:：0 表示3特权级程序可访问，1表示只能0、1、2特权级可访问。
# PWT、PCD、请看后面的填坑篇
# A:：0 表示该页未被访问，1表示已被访问。
# D:：脏位。0表示该页未写过，1表示该页被写过。
# PS:：只存在于页目录。0表示这是4KB页，指向一个页表。1表示这是4MB大页，直接指向物理页。低22位是页内偏移
# PAT：这个不管
# G：如果G位为1刷新TLB时将不会刷新PDE/PTE的G位为1的页，G=1切换进程该PTE扔然有效(这里学完TLB才能明白)
```

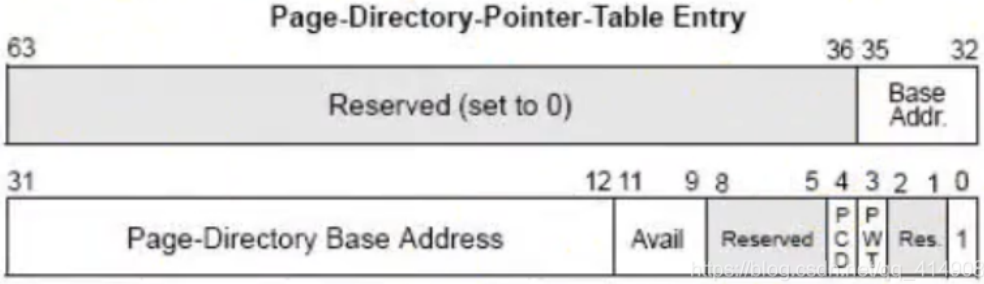
6.2.6.2 [2-9-9-12]分页

> `ntoskrnl.exe`
>
> 2-9-9-12分页又称为PAE(物理地址扩展) 分页
>
> PDPT: 页目录指针表



PDPTE/PDE/PTE

1、Page-Directory-Point-Table Entry



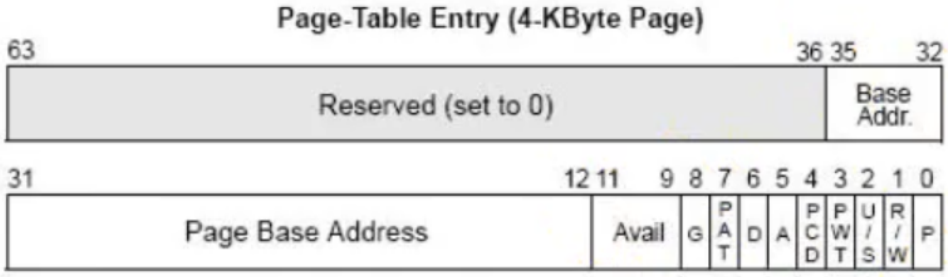
2、PDE结构

特别说明：

1、当PS=1时是大页，35-21位是大页的物理地址，这样36位的物理地址的低21位为0，这就意味着页的大小为2MB，且都是2MB对齐。

2、当PS=0时，35-12位是页表基址,低12位补0，共36位。

3、PTE结构



特别说明：

PTE中35-12是物理页基址，24位，低12位补0

物理页基址+12位的页内偏移指向具体数据

6.2.6.3 TLB表

6.2.7 系统调用

```
'_KUSER_SHARED_DATA'
    X86     内核起始地址:0xFFDF0000      内核结束地址:0xFFDF0FFF
           用户起始地址:0x7FFE0000       用户结束地址:0x7FFE0FFF

    X64       内核起始地址:0xFFFFF780`00000000      内核结束地址:0xFFFFF780`00000FFF
          用户起始地址:0x7FFE0000        用户结束地址:0x7FFE0FFF

'_Ktrap_frame' // 应用程序从3环进入0环,3环的寄存器信息由操作系统保存在结构里


'_ETHREAD'// 进程线程相关的结构体
    +000 Tcb :_KTHREAD  // dt _KTHREAD


'_KPCR'  // 描述当前CPU的状态
  +0x000     NtTib         :_NT_TIB     // dt _NT_TIB
  +128                     :_EPRCB      // dt _EPRCB

      dd KeNumberProcessrs  // 查看CPU数量(几核)
        dd KiProcessorBlock L2  // 查看两个核的KBCR
```

```
'syscall(貌似de指令别名sysenter)' // CPU通过MSR寄存器获取数据,与操作系统没有关系
     // 进入0环后执行的 NT!KiFastCallEntery例程
     // 直接调用是读取MSR寄存器获取EIP CS ESP,ss的值是 IA32_SYSENTER_CS里的值+8
    IA32_SYSENTER_CS 174h
    IA32_SYSENTER_ESP 175h
    IA32_SYSENTER_EIP 176h
    /* RDMSR/WRMST 可以读写MSR寄存器*/
    /* kd> rdmsr 174 // 查看CS*/

'中断门进入0环'    // 通过中断号(int 2Eh)查表用中断门进入0环
    // 进入0环后执行的 NT!KiSystemService例程
```

+0x000 DbgEbp	调试等其他作用
+0x004 DbgEip	
+0x008 DbgArgMark	
+0x00c DbgArgPointer	
+0x010 TempSegCs	
+0x014 TempEsp	
+0x018 Dr0	
+0x01c Dr1	
+0x020 Dr2	
+0x024 Dr3	
+0x028 Dr6	
+0x02c Dr7	
+0x030 SegGs	
+0x034 SegEs	
+0x038 SegDs	
+0x03c Edx	
+0x040 Ecx	
+0x044 Eax	
+0x048 PreviousPreviousMode	windows 中非易失性寄存器需要在中断例程中先保存
+0x04c ExceptionList	
+0x050 SegFs	
+0x054 Edi	
+0x058 Esi	
+0x05c Ebx	
+0x060 Ebp	
+0x064 ErrCode	
+0x068 Eip	中断发生时，保存被中断的代码段和地址，iret 返回到此地址
+0x06c SegCs	
+0x070 EFlags	
+0x074 HardwareEsp	中断发生时，若发生权限变换，则要保存旧堆栈
+0x078 HardwareSegSs	
+0x07c V86Es	虚拟 8086 方式下，变换需要保存段寄存器
+0x080 V86Ds	
+0x084 V86Fs	
+0x088 V86Gs	

6.2.8 进程线程KPCR

6.2.8.1 进程

```
' x86'
struct _KPROCESS
{
    +0x000 DOSPATCHER_HEADER: Header;   //"可等待"对象, 互斥体,Event事件[ 0x10个字节
]
    +0x018 DirectoryTableBase;      // 页目录表基址 [ Uint4B[2] ]
```

```
    +0x38 KernelTime;           // 在内核运行的时间 [ Uint4B ]
    +0x3c UserTime;             // 在3环运行的时间 [ Uint4B ]
    +0x50       ;               // 指向进程里线程链表中的第一个线程
    +0x54       ;               // 指向进程里线程链表中最后一个线程
    +0x5c Addinity;             // 线程能在哪个CPU上运行,拆成二进制,00000000，能运行的
填1[ Uint4B ]
    +0x62 BasePriority;         // 线程基本优先级 [ Char ]
}

PsLookupProcessByProcessId(HANDLE pid, _EPROCESS* eProcess);
struct _EPROCESS        // 内核进程结构体
{
    +0x00 _KPROCESS: Pcb;               // 进程相关信息[0x6c个字节]
    +0x70 _LARGE_INTEGER: CreateTime;   // 进程创建时间 [ Uint8B ]
    +0x78 _LERGE_INTEGER: ExitTime;     // 进程退出时间 [ Uint8B ]
    +0x84 UniqueProcessId;              // 进程编号,任务管理器中的PID [ Ptr32 Void ]
    +0x88 _LIST_ENTRY: ActiveProcessLinks;
                // 双向链表,所有活动进程都连接在一起( PsActiveHead全局链表头指向当前操作系
统的第一个进程)
    +0x90 QuotaUsage;                   // 物理页相关统计信息[ Uint4B[3] ]
    +0x9c QuotaPeak;                    // 物理页相关统计信息[ Uint4B[3] ]
    +0xa8 CommitCharge;                     // 虚拟内存相关统计信息 [ Uint4B ]
    +0xac PeakVirtualSize;                  // 虚拟内存相关统计信息 [ Uint4B ]
    +0xb0 VirtualSize;                      // 虚拟内存相关统计信息 [ Uint4B ]
    +0xbc DebugPort;                    // 调试相关 [ Ptr32 Void ]
    +0xc0 ExceptionPort;                // 调试相关 [ Ptr32 Void ]
    +0xc4 ObjectTable;                  // 句柄表 [ Prt32 _HANDLE_TABLE ]

    +0x11c VadRoot;                     // 标识0-2G哪些地址没占用 [ Ptr32 Void]
    +0x174 ImageFileName;               // 进程镜像文件名,最后十六个字节 [ UChar[16]
]
    +0x1a0 ActiveThreads;               // 活动线程数量 [ Uint4B ]
    +0x1b0 Peb;                         // 3环进程环境块 [ Ptr32 _PEB ]
}
```

6.2.8.2 线程

```
' x86'
struct _KTHREAD
{
    +0x00 _DISPATCHER_HEADER: Header;   // 可等待对象(互斥体,Event事件可以使用
WaitForSingleObject) [0x10]
    +0x18 InitialStack;             // 0环的栈,线程切换有关 [ Ptr32 Void ]
    +0x1c StackLimit;               // 0环的栈,线程切换有关 [ Ptr32 Void ]
    +0x20 Teb;                      // 线程的环境块,线程描述位于用户地址控件地址fs[0]-
>TEB [Ptr32 Void]
    +0x28 KernelStack;              // 0环的栈,线程切换有关 [ Ptr32 Void ]
    +0x2c DebugActive;              // 如果值为-1不能使用调试寄存器:DR0-DR7 [ UChar ]
    +0x2d State;                    // 线程状态: 就绪/等待/运行 [UChar]
    +0x34 _KAPC_STATE: ApcState;        // APC相关
    +0x6c BasePriority;                 // 线程优先级, 初始值是所属进程的KPROCESS-
>BasePriority,
                                        // 可以使用 KeSetBasePriorityThread()
函数重新设定
    +0x70 _KWAIT_BLOCK[4]: WaitBlock;       // 等待哪个对象
    +0xe0 ServiceTable;                 // SSDT系统服务表 [ Ptr32 Void ]
    +0xe8 ApcQueueLock;                 // APC相关 [ Uint4B ]
```

```
    +0x134 TrapFrame;                    // 进0环时保存3环环境
    +0x138 ApcStatePointer;              // APC相关 [ Ptr32[2] _KAPC_STATE ]
    +140 PreviousMode;                   // 0环调用还是3环调用 [ UChar ]
    +0x14c _KAPC_STATE: SavedApcState;   // APC相关
    +0x1b0 _LIST_ENTRY: ThreadListEntry;     // 链表串联进程中的所有线程
}

struct _ETHREAD          // 内核线程结构体
{
    +00 _KTHREAD: Tcb            // 线程相关详细信息 [0x1c0]
    +0x1ec _CLIENT_iD: Cid;      // 进程ID、线程ID
    +0x220 ThreadsProcess;       // 指向自己所属进程 [ Ptr32 _EPROCESS ]
    +0x22c _LIST_ENTRY: ThreadListEntry;     // 双向链表,串联进程中所有线程
}

    _LIST_ENTRY: KiWaitListHead;     // 等待链表,线程调用了Sleep或者
WaitForSingleObject登函数就挂载这个链表
    _LIST_ENTRY: KiDispatcherReadyListHead;  // 调度链表(就绪链表)
```

6.2.8.2.1 KPCR

> KPCR: CPU控制区
>
> 1): 当线程进入0环时, FS:[0]指向KPCR, 3环时FS:[0]->TEB
>
> 2): 每个CPU都有一个KPCR(一个核一个)
>
> 3): KPCR中存储了CPU本身要用的一些重要数据: GDT IDT以及线程相关的一些信息

```
'X86'
struct _NT_TIB      /* 和TEB结构类似 */
{
    +0x0 ExceptionList;      // 0环异常处理函数,[Ptr32
_EXCEPTION_REGISTRATION_RECORD]
    +0x04 StackBase;         //  当前线程栈的栈底 [ Ptr32 Void ]
    +0x08 StackLimit;        // 当前线程栈的大小 [ Ptr32 Void ]
    +0x18 Self;              // 指向这个结构自己 [ Ptr32 _NT_TIB ]
}

struct _KPRCB
{
    +0x04 CurrentThread;         // 当前执行线程的结构[Ptr32 _KTHREAD]
    +0x08 NextThread;            // 下一个执行线程的结构 [Ptr32 _KTHREAD]
    +0x0c IdleThread;            // 如果没有需要切换的线程,要执行的空闲线程 [Ptr32
_KTHREAD]
}

struct _KPCR
{
    +00 _NT_TIB: NtTib;          // [ 0x1c ]
    +0x1c SelfPcr;               // 指向_KPCR自己
    +0x20 Prcb;                  // 指向 _KPRCB: PrcbData [ Ptr32 _KPRCB ]
    +0x038 IDT;                  // IDT表基址 [Ptr32 _KIDTENTRY ]
    +0x3c GDT;                   // GDT表基址 [Ptr32 _KGDTENTRY ]
    +0x40 TSS;                   // 指向TSS, 每个CPU都有有一个TSS [ Ptr32 _KTSS ]
    +0x51 Number;                // CPU编号 [ UChar ]
    +0x120 _KPRCB: PrcbData;      // 拓展结构体
}
```

6.2.8.2.2 线程切换

```
KiFindReadyThread();        // 搜索调度链表
KiSwapContext();            // 线程切换
```

6.2.9 APC

> 线程APC队列:

```
    struct _KAPC
    {


    }
```

```
'APC挂入流程'
    QueueUserAPC();                         // Kernel32.dll(3环)
    NtQueueApcThread();                     // Ntosker.exe(3环)

    KeInitializeApc();                      // 分配空间,初始化KAPC结构体(0环)
    KeInsertQueueApc();
    KiInsertQueueApc();                     // 将KAPC插入指定APC队列
```

6.2.10 内存管理

```
'Windbg'
    !vad 0x123456               //  0x123456-->_EPROCESS+11C VadRoot
```

```
    struct _MMVAD_FLAGS // 内存属性
    {
        +000  Protection;   // 1 READONLY  2 EXECUTE  3 EXECUTE_READ  4
READWITER
                            // 5 WRITECOPY  6 EXECUTE_READWRITE  7
EXECUTE_WRITECOPY

        Private Memory          //  VirtualAlloc/VirtualAllocEx申请的内存
        Mapped Memory           // CreateFileMapping 映射
    }
```

```
    HADNLE CreateFileMapping()                  // 创建物理页内核对象
    void* MAPViewOfFile();                      // 将物理页与线性地址进行映射
```

6.2.11 异常

```
    typedef struct _EXCEPTION_RECORD
    {
        DWORD ExceptionCode;            // 异常代码
        DWORD ExceptionFlags;           // 异常状态
        struct _EXCEPTION_RECORD* ExceptionRecord;      // 下一个异常
        PVOID ExceptionAddress;             // 异常发生地址
        DWORD NumberParameters;             // 附加参数个数
        ULONG_PTR ExceptionInformation;
        [EXCEPTION_MAXIMUM_PARAMERERS];     // 附加参数指针
    }
```

6.3 Windows驱动

```
'压力测试'
    P98
'Win32子系统三大核心部件'
    kernel32.dll                    /* --> 文件,网络,注册表 */
    user32.dll                      /* --> 用户窗口,对话框,按钮 */
    gdi32.dll                       /* --> 绘图, 窗口绘制*/

'Native API'
    ntdll.dll                       /* --> Windows原生API */

'操作系统内核'
    ntoskrnl.exe                    /* 集成PE文件,dll文件*/
```

```
'Ctrl + R'
     perfmon.msc                            /* 打开性能监视器 */
'双机调试'
    bcdedit/debug on    // cmd设置系统为调试模式

    // 设置网络调试进行连接,获取Key
    bcdedit/dbgsettings net hostip:192.168.0.2 port:50010
        Key=1n7azearrrjme.2yvmrnvi4o02t.2ld90ebhcb0so.11xom58pcnv2e

    bcdedit /dbgsettings serial baudrate:115200 debugport:2


'build驱动编译
    # makfile 格式固定,(代码前导不能有空格)
     !INCLUDE $(NTMAKEENV)\makefile.def
    # Sources
    TARGETNAME=DDK_HellWord     // 指定生成的驱动的名字
    TARGETTYPE=DRIVER           // 指定生成文件类型 Driver是驱动文件
    TARGETPATH=Check            // 指定生成驱动所有路径
    INCLUDES=$(BASEDIR)\inc;\       // 指定相关头文件所在模具路径
            $(BASEDIR)\inc\wxp;\

    // 空一行
    SOURCES=mini_ddk.c\         // 指定驱动源代码 *.c *.cpp


'cmd设置驱动服务'
```

```
/*
注册服务([sc create]:创建一个服务 [binPath]:驱动文件路径 [type]:驱动类型 [start]:该服务的启动类型)  */
sc create FirstDriver binPath= "C:\FirstDirver.sys" type= kernel start= demand

/* 创建驱动服务后,使用sc命令启动服务 [sc start]:等同于 StartService函数*/
sc start FirstDriver

/* 停止服务 */
sc stop FirstDriver

/* 删除服务 */
sc delete FirstDriver
```

6.3.1 内核数据结构

```
#include <ntifs.h>
#include <ntddk.h>                // 当ntddk.h和ntifs.h发生冲突时,先包含ntifs.h可解决
#include <wdm.h>
#include <wdf.h>                  // WDF框架
#include <ntdddisk.h>             // 磁盘

#define NTSTRSAFE_LIB             // 使用ntstrsafe静态库时更好兼容Windows2000
#include <ntstrsafe.h>
```

```
'#预定义宏'
    __FILE__              /* 宏所在文件的源文件名(路径)    */
    __LINE__              /* 宏所在行的行号                */
    __DATE__              /* 代码编译的日期                */
    __TIME__              /* 代码编译的的时间              */
    __func__              /* 宏所在的当前函数 */
    __cplusplus           /*  */
    UNICODE_STRING        /* 内核字符串结构 */
    LIST_ENTRY            // 内核链表结构
    LARGE_INTEGER         // LONGLONG (int64)

    __CHAR_UNSIGNED                 /* 如果char类型为无符号,该宏定义为1否则未定义 */
    __COUNTER__                     /* 从0开始,每次使用都会递增1 */
    _DEBUG                          /* 如果设置了/lDd/mDd/mTd改宏定义为1否则未定义*/
    __FUNCTION__                    /* 函数名称 不含修饰名 */
    __FUNCTIONW__                   // 当前执行函数的函数名(unicode),以'\0'表示结尾
    __FUNCDNAME__                   /* 函数名称 包含修饰名 */
    __FUNCSIG__                     /* 包含了函数签名的函数名 */
    _WIN32                          /* 当编译为32位ARM,64位ARM,X86或64定义为1 否则未定义 */
    _WIN64                          /* 当编译为64位ARM或X64定义为1 否则未定义 */
    __TIMESTAMP__                   /* 最后一次源代码修改时间和日期 */

    DELAY_ONE_MILLISECOND           // 延迟时间一毫秒
```

```
'内核数据结构'
    typedef struct P2C_IDTR_
    {
        P2C_U16 limit;
```

```
        P2C_U32 base;
    }
    _asm sidt m_idt     // 读出一个P2C_IDTR结构,并返回IDT地址

    UNICODE_STRING                          /* 字符串结构*/
    KSPIN_LOCK                              /* 自旋锁结构*/
    LIST_ENTRY                              /* 链表结构*/
    POOL_TYPE                               /* 内存属性联合体*/
    ONJECT_ATTRIBUTES                       // 该结构描述对象句柄的属性到打开句柄的例程
    KEY_VALUE_PARTIAL_INFORMATION           /* 注册表项的值条目的值信息的结构 */
    IO_STATUS_BLOCK                         // 函数操作结果结构 P61
    LARGE_INTEGER                           // 文件读取位置结构体(读取起始位置)
```

```
    typedef struct DRIVER_OBJECT, *PDRIVER_OBJECT;          /* 驱动对象*/
    typedef struct _DEVICE_OBJECT *PDEVICE_OBJECT;;         /* 设备对象*/
```

6.1.1.0 LDT

> 系统所有程序 驱动对象里都有这个结构 是驱动对象的成员
>
> qudongduixiang1->DriverSection(PVOID DriverSection) 这个结构体在什么时候有的 io管理器在加载我们驱动的时候 调用DriverEntry这个历程的时候 把我们驱动对象也加入到系统的一个全局链表中
>
> 就是为了方便io管理器进行维护或者方便对象管理器进行维护 为了查找方便 这个全局链表呢 把系统所有驱动程序串起来就是连接起来

```
    // DriverObject->DriverSection指向 _LDR_DATA_TABLE_ENTRY
typedef struct _LDR_DATA_TABLE_ENTRY {
    LIST_ENTRY InLoadOrderLinks;
    LIST_ENTRY InMemoryOrderLinks;
    LIST_ENTRY InInitializationOrderLinks;//已经调用DriverEntry这个函数的所有驱动程序
    PVOID DllBase;
    PVOID EntryPoint;//驱动的进入点 DriverEntry
    ULONG SizeOfImage;
    UNICODE_STRING FullDllName;//驱动的满路径
    UNICODE_STRING BaseDllName;//不带路径的驱动名字
    ULONG Flags;
    USHORT LoadCount;
    USHORT TlsIndex;
    union {
        LIST_ENTRY HashLinks;
        struct {
            PVOID SectionPointer;
            ULONG CheckSum;
        };
    };
    union {
        struct {
            ULONG TimeDateStamp;
        };
        struct {
            PVOID LoadedImports;
        };
    };
} LDR_DATA_TABLE_ENTRY, *PLDR_DATA_TABLE_ENTRY;
```

6.3.2 Windows内核API

6.3.2.1 基础API

```
    UNREFERENCED_PARAMETER(a);       /* 告诉编译器，已经使用了该变量，不必检测警告 */
    NT_SUCCESS(status)       /* 宏函数,判断HSTATUS返回值的函数是否STATUS_SUCCESS*/

    PIRP IoGetTopLevelIrp();     /* 返回当前线程的TopLevelIrp字段的值。*/

    // 将输入的有符号整数转换为有符号大整数
    LARGE_INTEGER RtlConvertLongToLargeInteger(LONG SignedInteger);
    NTSTATUS RtlGetVersion(PRTL_OSVERSIONINFOW lpVersionInformation);
```

```
   #define INITCODE code_seg("init")         // 将入口函数放入INIT标志内存中,驱动加载成功后该函数卸载
' 驱动程序入口函数(驱动对象的指针, 结构指向当前驱动所对应的注册表位置) '
   #pragma INITCODE
    NTSTATUS DriverEntry(
        PDRIVER_OBJECT DriverObject,
        PUNICODE_STRING RegistryPath);

'驱动的卸载函数'
    void DriverUnload(PDRIVER_OBJECT DriverObject);

'驱动断点函数'
   KdBreakPoint();                  // Debug模式下断点
    DbgBreakPoint();                // Release版本断点

   HANDLE PsGetCurrentProcessId();           // 获取当前执行代码的进程Id(内核)
    NTHALAPI KIRQL KeGetCurrentIrql();       // 获取当前IRQL

    PAGED_CODE();
'时间睡眠'
    LARGE_INTEGER Interval;
    Interval.QuadPart = (5*1000* ((-10)*1000)); // 设置时间5秒
    KeDelayExecutionThread(KernelMode, FALSE, &Interval)

'运行错误停止系统(蓝屏报错)'
    void KeBugCheckEx(ULONG BugCheckCode,
                      ULONG_PTR BugCheckParameter1,
                      ULONG_PTR BugCheckParameter2,
                      ULONG_PTR BugCheckParameter3,
                      ULONG_PTR BugCheckParameter4);


'获取函数地址'
    /* 通过函数名获取函数地址*/
    PVOID MmGetSystemRountineAddress(PUNICODE_STRING SystemRountineName);

'系统关机回调函数 •P101'
// 当系统进入关机或者重启逻辑时会发送IRP_MJ_SHUTDOWN信号, 先执行这个`系统关机回调`
    NSTSTATUS IoRegisterShutdownNotification(
            PDEVICE_OBJCET DeviceObject);   /* 设备对象指针*/

// 驱动已注册`系统关机回调`, 驱动卸载必须使用这个函数来移除`系统关机回调`
    void IoUnregisterShutdownNotification(
```

```
        PDEVICE_OBJECT DeviceObject);    /* 指向驱动程序的设备对象指针*/

// 设置系统关机消息时的需要执行函数(分发函数)
    DriverObject->MajorFuntion[IRP_MJ_SHUTDOWN] = SysShutdown;
```

6.3.2.2 字符串操作

```
    ANSI_STING          /* 内核ANSI码类型字符串结构 */
    UNICODE_STRING      /* 内核UNICODE字符串结构 P36*/
    UCHAR               /*(unsigned char)是无符号的*/

    UNICODE_STRING str=RTL_CONSTANT_STRING(L"my first string"); /*定义常量
UNICODE_STRING*/
    DECLARE_CONST_UNICODE_STRING(ntDeviceName, L"name");    /*定义常量
UNICODE_STRING, 常量名ntDeviceName*/

'初始化内核字符串结构,将Buffer指向参数SourceString字地址,使用期间参数SourceString必须有效'
    VOID RtlInitAnsiString();
    VOID RtlInitUnicodeString(
            PUNICODE_STRING DestinationString,      // 指向 UNICODE_STRING结构的指
针
            PCWSTR SourceString);       /* L""形字符串*/

'内核宏函数,初始化内核字符串结构 "Ntstrsafe.h",将DestinationString指向Buffer' •P35
    VOID RtlInitEmptyUnicodeString(PUNICODE_STRING DestinationString,
                                   PCWTR Buffer,
                                   USHORT BufferSize);

'字符串拷贝'
   RtlCopyString();     /* ANSI_STRING码字符串拷贝*/
   NTSYSAPI VOID RtlCopyUnicodeString(
       PUNICODE_STING DestinationString,
       PUNICODE_STRING SourceString);

    NTSTATUS RtlUnicodeStringCopyString();
     //将以"\0"结尾的字符串pszSrc拷贝到DestinationString所指向的内存中,成功返回
STATUS_SUCCCESS
    // 只能在IRQL:PASSIVE_LEVEL下运行

'字符串比较'
    RtlCompareString();         /* ANSI_STRING字符串比较*/
    RtlCompareUnicodeString(    /* UNICODE_STRING字符串比较*/
        PUNICODE_STRING String1,
        PUNICODE_STRING String2,
        BOOLEAN CaseInsensitive);

'字符串转换'
    RtlAnsiStringToUnicodeString();     /* ANSI_STRING转Unicode_STRING*/
    RtlUnicodeStringToAnsiString(       /* UNICODE_STRING转ANSI_STRING*/
        PANSI_STRING DestinationString,
        PUNICODE_STRING SourceString,
        BOOLEAN AllocateDestinationString);

'格式化字符串' /* sprintf*/
    NTSTRSAFEDDI RtlStringCchPrintfA(
            NTSTRSAFE_PSTR pszDest,     // 需要格式化到哪个字符数组里面
(NTSTRSAFE_PSTR==char*)
```

```
            size_t cchDest,              // 字符串大小
            NTSTRSAFE_PCSTR pszFormat,   // 需要初始化的字符串(char*)("sad%d %c")
            ...);                        // 需要格式化的参数, N
'讲UNICODE_STRING字符串拷贝到PCWSTR'
    NTSTATUS RtlAppendUnicodeToString(PUNICODE_STRING, PCWSTR);

'字符串打印'
    // 消息打印
    KdPrint(("111%d", a));               // 宏函数,Dbg模式会调用DbgPrint,Rele模式不操
作
    DbgPrint("%ws\n", __FUNCTIONW__);        // 表示打印一个以'\0'结束的Unicode字符串
        "%wZ"                                // 打印UNICODE_STRING类型(不以'\0'为结尾
的)
```

6.3.2.3 自旋锁

```
    KSPIN_LOCK my_spin_lock;         // 锁结构,
'初始化自旋锁/队列自旋锁'
    VOID KeInitializeSpinLock(PKSPIN_LOCK SpinLock);

'自旋锁自选锁上锁/解锁 , 当使用自旋锁时,需要将它全局/静态或池中 KSPIN_LOCK'
    // ===自旋锁上锁
    void KeAcquireSpinLock(SqinLock, OldIrql);
    // ===自旋锁释放 P41
    VOID KeReleaseSpinLock(PKSPIN_LOCK SpinLock, KIRQL NewIrql);

'队列自旋锁获取和释放'
    // ===队列自旋锁上锁
    VOID KeAcquireInStackQueuedSpinLock(PKSPIN_LOCK SpinLock,
                                        PKLOCK_QUEUE_HANDLE LockHandle);
    // ===队列自旋锁释放'
    void KeReleaseInStackQueuedSpinLock(PKLOC_QUEUE_HANDLE LockHandle);

/*将事件对象初始化为同步（单个服务程序）或通知类型的事件，并将其设置为有信号或无信号状态*/
void KeInitializeEvent(PKEVENT Event, EVENT_TYPE Type, BOOLEAN Status);
```

6.3.2.4 内核链表操作

```
    LIST_ENTRY               // 链表结构体
'初始化头节点'
    VOID InitializeListHead(PLIST_ENTRY ListHead);

'判断当前链表是否是空链表(只有头节点)'
    BOOLENA IsListEmpty(const LIST_ENTRY * ListHead);   // 返回ture是空链表,否则非空

'插入节点'
    '===插入节点到第一位置'
    VOID InsertHeadList(PLIST_ENTRY ListHead, PLIST_ENTRY Entry);

    '===插入节点到末尾'
    VOID InsertTailList(PLIST_ENTRY ListHead, PLIST_ENTRY Entry);

    '===以自旋锁方式插入链表节点'
```

```
    PLIST_ENTRY ExInterlockedInsertHeadList(PLIST_ENTRY ListHead,
                                            PLIST_ENTRY ListEntry,
                                            PKSPIN_LOCK lock);
'移除节点,成功移除返回从链表移除的节点指针,无节点移除返回NULL'
    '===移除第一个节点'
    PLIST_ENTRY RemoveHeadList(PLIST_ENTRY ListHead);

    '===移除末尾节点'
    PLIST_ENTRY RemoveTailList(PLIST_ENTRY ListHead);
    BOOLEAN RemoveEntryList(PLIST_ENTRY Entry);

    '===以自旋锁方式删除一个节点'
    PLIST_ENTRY ExInterlockedRemoveHeadList(PLIST_ENTRY ListHead,
                                            PKSPIN_LOCK Lock);
'遍历节点'
    '宏函数,宏返回一个结构实例的基地址,该结构的类型和结构所包含的一个域(成员)地址已知' •P39
    PCHAR CONTAINING_RECORD(PCHAR Address, TYPE type, PCHAR Field);

'修改内存页表属性(可读可写)'
    MmCreateMdl()
```

6.3.2.5 内存操作

> 非分页内存指这块内存的内容不会被置换到磁盘上

```
    typedef enum _POOL_TYPE {PagedPool, NonPagedPool, NonPagedPoolExecute,
                             NonPagedPoolNx};       // 重要属性
'在内存池中申请内存'
    '===在内存池中分配内存'/* malloc*/
    PVOID ExAllocatePool(POOL_TYPE PoolType, SIZE_T NumberOfBytes);

    '===在内存池中分配内存并设置使用者标志'
    PVOID ExAllocatePoolWithTag(
                    POOL_TYPE PoolType,
                    SIZE_T NumberOfByTes,
                    ULONG Tag);
'释放内存池中的内存'
    ExFreePool(a);  /* free*/
    void ExFreePoolWithTag(PVOID P, ULONG Tag);

'内存初始化' /* memset*/
    void RtlFillMemory(PVOID Destination, SIZE_T Length, ULONG Pattern);

'内存拷贝'  /* memcpy*/
    void RtlCopyMemory(VOID* Destination, VOID* Source, SIZE_T Length);
```

非分页旁视列表

```
'初始化旁视列表对象'
    void ExInitializeNPagedLookasideList(PNPAGED_LOOCKASIDE_LIST Lookaside,
                    PALLOCATE_FUNCTION Allocate,
                    PFREE_FUNCTION Free,
                    ULONG Flags,
                    SIZE_T Size,
                    ULONG Tag,
```

```
                    USHORT Depth);
'可以初始化分页/非分页旁视列表对象'
    NTSTATUS ExInitializeLookasideListEx(PLOOKASIDE_LIST_EX Lookaside,
                    PALLOCATE_FUNCTION_EX Allocate,
                    PFREE_FUNCTION_EX Free, POOL_TYPE PoolType,
                    ULONG Flage,
                    SIZE_T Size,
                    ULONG Tag,
                    USHORT Depth);
'申请旁视列表对象内存'
    PVOID ExAllocateFromNPagedLookasideList(PNPAGED_LOOKASIDE_LIST Lookaside);

'释放旁视列表对象内存'
    void ExFreeToNPagedLookasideList(PNPANGED_LOOKASIDE_LIST Lookaside);

'删除旁视对象'
    void ExDeleteNPagedLookasideList(PAPAGED_LOOKASIDE_LIST Lookaside);
```

```
'内存操作'
    // 验证一个地址是否可用,检查给定虚拟地址的读或写操作是否会发生页错误 #include <ntifs.h>
    BOOLEAN MmIsAddressValid(PVOID VirtualAddress);
```

6.3.2.6 句柄和对象

> 一般 `Zw` 开头的内核函数只能在ORQL是PASSIVE_LEVEL下运行

```
    OBJECT_ATTRIBUTES           // 该结构描述对象句柄的属性到打开句柄的例程
    DRIVER_OBJECT               // 驱动对象
    DEVICE_OBJECT               // 设备对象

'初始化句柄结构属性信息'
    VOID InitializeObjectAttributes(
                POBJECT_ATTRIBUTES p, /* 需要初始化的结构*/
                PUNICODE_STRING n,  /* 谁打开句柄的(对象的名字)*/
                ULONG a,            /* 标志/权限*/
                HANDLE r,           /* 可以为NULL*/
                PSECURITY_DESCRIPTOR s);    /* 可以为NULL*/
'对象'
    /* 对象创建*/
    ObCreateObject()
        ObCreateObject(KeGetPreviousMode(), PsProcessType, ObjectAttributes,
                  KeGetPreviousMode(), 0, 0x258, 0, 0, &ProcessObject);
    /* 释放对象指针*/
    void ObDereferenceObject(a);

'句柄对象'
    /* 创建一个句柄对象'        <ntifs.h> P52*/
    NTSYSAPI NTSTATUS ZwCreateEvent(
                            PHANDLE EventHandle, /* 用于保存EVENT句柄*/
                            ACCESS_MASK DesiredAccess,  /* EVENT打开权限*/
                            POBJECT_ATTRIBUTES ObjectAttributes, /* EVENT属性结构
*/
                            EVENT_TYPE EventType,   /* EVENT类型*/
                            BOOLEAN InitialState);  /* EVENT的初始状态*/
```

```
    /* 打开一个句柄对象*/
    NTSYSCALLAPI NTSTATUS ZwOpenEvent(
                                PHANDLE EventHandle,
                                ACCESS_MASK DesiredAccess,
                                POBJECT_ATTRIBUTES ObjectAttributes)

    /* 获取句柄对象(EventHandle)对应的Event对象指针 ---对象化'  •P53 */
        /* 通过对象的句柄获取对象指针*/
    NTSTATUS ObReferenceObjectByHandle(
                    HANDLE Handle,          /* ZwCreateEvent打开的Event对象的句柄*/
                    ACCESS_MASK DesiredAccess, /* 需要获取此对象的权限*/
                    POBJECT_TYPE ObjectType,    /* 对象的类型*/
                    KPROCESSOR_MODE AccessMode,     /*访问模式(用户态和内核态)*/
                    PVOID* Object,  /* 返回参数,函数执行成功保存对象的指针*/
                    POBJECT_HANDLE_INFORAMTION HandleInformation);  /* 保留,设置为
NULL*/


    /* 通过一个名字来获得一个对象得指针(闭源函数,文档没有公开,声明直接使用了)*/
    /* 调用ObReferenceObjectByName会使驱动对象的引用计数增加,ObDereferenceObject释放驱
动对象引用*/
    extern POBJECT_TYPE* IoDriverObjectType;
    NTSTATUS ObReferenceObjectByName(
        PUNICODE_STRING ObjectName,         // 指向驱动对象名字的UNICODE_STRING
        ULONG Attributes,               // OBJ_CASE_INSENSITIVE(不区分大小写 驱动名
可以写大写或小写)
        PACCESS_STATE AccessState,      // NULL(结构比较复杂设计对象特性的东西access
state)
        ACCESS_MASK DesiredAccess,      // 0(访问权限可以写0为完全访问 不受控制
FILE_ALL_ACCESS)
        POBJECT_TYPE ObjectType,        // *IoDriverObjectType(对象的类型)
        KPROCESSOR_MODE AccessMode,     //  KernelMode(访问模式(用户态和内核态))
        PVOID ParseContext,             // NULL(设备上下文)
        PVOID* Object);                 // 返回参数,函数执行成功保存对象的指针

    // 关闭句柄对象
    NTSYSQPI NTSTATUS ZwClose(HHANDLE Handle);

    // 原子地 交换函数地址
    PVOID InterlockedExchangePointer(PVOID volatile *Target, PVOID value);
```

6.3.2.7 注册表操作

> ---P54
>
> 注册表路径 `C:\Windows\System32\config`

```
    KEY_VALUE_PARTIAL_INFORMATION       /* 注册表项的值条目的值信息的结构 */
'打开&创建 一个注册表键'  //IRQL==PASSIVE_LEVEL (InitializeObjectAttributes初始化结构)
    NTSTATUS ZwCreateKey(
                    PHANDLE KeyHandle,  /* 返回参数,成功打开或创建键值后保存键值的句柄*/
                    ACCESS_MASK DesiredAccess,  /* 打开权限*/
                    PBOJECT_ATTRIBUTES ObjectAttributes, /* 键值对象的属性*/
                    ULONG TitleIndex,       /* 0*/
                    PUNICODE_STRING Class,  /* NULL*/
                    ULONG CreateOptions,    /* 打开&创建注册表键的选项 */
                    PULONG Disposition);    /* 返回参数，注册表打开或是创建得到的*/
```

```
    NTSYSAWPI NTSTATUS ZwOpenKey(
                            PHANDLE KeyHandle,
                            ACCESS_MASK DesiredAccess,
                            POBJECT_ATTRIBUTES ObjectAttributes);
'关闭注册表句柄'
    NTSTATUS ZwClose(HANDLE Handle);

'修改注册表'
    NTSTATUS ZwSetValueKey(
                        HANDLE KeyHandle,    /* 要修改的注册表键的句柄*/
                        PUNICODE_STRING ValueName, /* 要修改的注册表键值的名字*/
                        ULONG TitleIndex,        /* 保留参数,设置为0*/
                        ULONG Type,          /* 键值的类型*/
                        PVOID Data,          /* 要往键值中写入数据的缓冲区*/
                        ULONG DataSize);     /* Data缓冲区的大小*/

'读取注册表' •P58
    NTSTATUS ZwQueryValueKey(
                HANDLE KeyHandle,    /* 需要被查询的注册表键句柄*/
                PUNICODE_STRING VauleName,  /* 需要被查询的键值名字*/
                KEY_VALUE_INFORMATION_CLASS KeyValueInformationClass, /* 枚举值,查
询类型*/
                PVOID KeyValueInformation,  /* 接受键值信息的缓冲区*/
                ULONG Length,    /* 缓冲区大小,单位字节*/
                PULONG ResultLength);    /* 返回参数,缓冲区中实际键值信息需要的大小*/
```

6.3.2.8 文件操作

```
    IO_STATUS_BLOCK             // 函数操作结果结构 P61
    LARGE_INTEGER               // 文件读取位置结构体(读取起始位置)
'打开&创建文件,设备'     •P62
    NTSTATUS ZwCreateFile(
              OUT PHANDLE FileHandle,     /* 文件打开后,句柄存放的地址*/
              IN ACCESS_MASK DesiredAccess,  /* 申请的权限*/
              IN POBJECT_ATTRIBUTES Object_Attribute,     /* 对象属性*/
              OUT PIO_STATUS_BLOCK IoStatusBlock,     /* 把函数操作结果放入该结构*/
              IN PLARGE_INTEGER AllocationSize OPTIONAL, /* 文件初始分配的大小,不常用
置0*/
              IN ULONG FileAttributes, /* 控制新建立的文件属性,一般为0或
FILE_ATTRIBUTE_NORMAL*/
              IN ULONG ShareAccess,  /* 共享访问权限,允许别的代码打开这个文件所持有的权限
*/
              IN ULONG CreateDisposition,     /* 文件打开意图(新建/打开/覆盖...),选择项
不能组合*/
              IN ULONG CreateOptions,         /* 同步操作(防止异步操作产生未决)*/
              IN PVOID EaBuffer OPTIONAL,     /* 设备驱动中必须设为NULL*/
              IN ULONG EaLength);             /*  设为0*/
'关闭文件句柄'
    NTSTATUS ZwClose(HANDLE Handle);

'读取文件'  •P64
        /* 函数成功返回值STATUS_SUCCESS,读取文件长度之外的返回STATUS_END_OF_FILE(读取完
毕)*/
    NTSTATUS ZwReadFile(
             IN HANDLE FileHandle,        /* 成功打开或者创建成功的文件的句柄*/
```

```
            IN HANDLE Event OPTIONAL, /* 事件Event对象指针,用于异步完成读取,同步读取忽略用NULL*/
            IN PIO_APC_ROUTINE ApcRoutine OPTUINAL,
                    /* 回调例程,用于异步读取,同步读取忽略用NULL*/
            IN PVOID ApcContext OPTIONAL,        /* 未知,一般为NULL*/
            OUT PIO_STATUS_BLOCK IoStatusBlock, /* 存储执行的返回结果*/
            OUT PVOID Buffer,                    /* 读取文件成功,存储缓冲区*/
            IN ULONG Length,                     /* 描述缓冲区的长度*/
            IN PLARGE_INTEGER ByteOffset OPTIONAL,
                /* 表示要读取的文件的偏移量(读取文件的起始位置),文件句柄不一定支持直接读取当前偏移量*/
            IN PULONG Key OPTIONAL);     /* 读取文件时使用的一种附加信息,一般使用为NULL*/

'写入文件(和读取文件的参数类似)' •P64
    NTSTATUS ZwWriteFile(
             HANDLE FileHandle,
             HANDLE Event,
             PIO_APC_ROUTINE ApcRoutine,
             PVOID ApcContext,
             PIO_STATUS_BLOCK IoStatusBlock,
             PVOID Buffer,
             ULONG Length,
             PLAREG_INTEGER ByteOffset,
             PULONG Key);
```

6.3.2.9 进程、线程与事件

```
'注册 线程/进程和桌面句柄 操作的回调函数列表'
NTSTATUS ObRegisterCallbacks(
        POB_CALLBACK_REGISTRATION CallbackRegistration,
            /* 指向回调函数和其他注册信息列表的一个结构的指针*/
        PVOID* RegistrationHandle); /* 输出参数,指向接收标识已注册回调函数集的值的变量的指针*/
```

进程

```
'打开进程'
    NTSYSAPI NTSTATUS ZwOpenProcess(
        PHANDLE ProcessHandle,  /* 输出参数,用于存放获取到的句柄值*/
        ACCESS_MASK DesiredAccess,  /* 句柄权限*/
        POBJECT_ATTRIBUTES ObjectAttributes,    /* 指向一个结构,该结构指定要应用于进程对象的句柄*/
        PCLIENT_ID ClientId);   /* 指向客户端ID的指针,用于标识要打开进程的线程*/
```

线程

```
'生成系统线程'  •P66
    NTSTATUS PsCreateSystemThread(
        OUT PHANDLE ThreadHandle,       /* 句柄指针,返回的线程句柄放入该地址*/
        IN ULONG DesiredAccess,         /* 未知,总是填0*/
        IN POBJECT_ATTRIBUTES ObjectAttributes OPTIONAL,    /* 未知,总是填NULL*/
        IN HANDLE ProcessHandle OPTIONAL,   /* 未知,总是填NULL*/
        OUT PCLIENT_ID ClientId OPTIONAL,   /* 未知,总是填NULL*/
        IN PKSTART_ROUTINE StartRoutine,    /* 要执行的线程启动函数*/
        IN PVOID StartContext);     /* 线程函数的参数*/
```

```
'线程启动函数(回调函数)原型,调用者使用启动函数时当参数传递进去'
        VOID CustomThreadProc(IN PVOID context);

'线程结束函数(线程的结束应该是线程自己调用结束函数)'
        NTSTATUS PsTerminateSystemThread(
            NTSTATUS ExitStatus);   /* 指定终止系统线程的状态，STATUS_SUCCESS */
'关闭线程句柄'
    ZwClose(HANDLE);

    PTHREAD KeGetCurrentThread();                                   /* 获取当前线程*/
    KPRIORITY KeSetPriorityThread(PKTHREAD Thread, KPRIORITY Priority); /* 设置驱动程序创建线程的运行优先级*/
    NTSTATUS KeDelayExecutionThread(KPROCESSOR_MODE WaitMode, BOOLEAN Alertable, PLARGE_INTEGER Interval);
                /* 将当前线程置于可报警或不可报警的等待状态一段指定的时间间隔*/
```

事件

```
    KEVENT                                  /* 事件结构*/
   '初始化事件结构(该事件对象不需要销毁)  •P67
    VOID KwInitializeEvent(
         IN PRKEVENT Event,          /* 需要初始化的事件*/
         IN EVENT_TYPE Type,         /* 事件类型Synchr(同步事件),NotIfic(通告事件)*/
         IN BOOLEAN State)           /* 初始化状态,一般为FALSE(设置无信号状态)*/

'设置事件使用函数' •P68
    LONG KeSetEvent(
            IN PRKEVENT Event,           /* 要设置的事件*/
            IN KPRIORITY Increment,      /* 用于提升优先权*/
            IN BOOLEAN Wait);        /* 是否后面紧接着一个KeWaitSingleObject来等待这个事件*/

'设置当前线程为阻塞状态,直到给定调度器对象设置为信号状态,继续执行'
        /* 当初始化事件结构时,类型使用同步事件(有信号),只有此函数可以等待函数内部自动重设事件(无信号)*/
        /* 当使用通告事件被设置(有状态)被拦截时，需要使用KeResetEvent手动重设事件(无信号)*/
    NTSTATUS KeWaitForSingleObject(
              PVOID Object,                  /* 等待的事件对象指针*/
              KWAIT_REASON WaitReason,
                 /* 指定等待的原因,驱动程序应设置Executive,除非它不是上下文执行*/
              KPROCESSOR_MODE WaitMode,/* 指定等待按照 内核/用户 模式等待*/
              BOOLEAL Alertable,          /*未知,设为0*/
              PLARGE_INTEGER Timeout);    /*设置等待时间(整数时绝对时间,负数时现在往后的相对时间)*/

    NTSTATUS KeWaitForMultipleObjects(
            ULONG Count,
            PVOID Object[],
            Waittype,
            KWAIT_REASON WaitReason,
            KPROCESSOR_MODE WaitMode,
            BOOLEAN Alertable,
            PLARGE_INTEGER Timeout,
            PKWAIT_BLOCK WaitBlockArray);

    KeWaitSingleObject();
```

```
'重设事件对象状态' •P68
    LONG KeResetEvent(IN PRKEVENT Event);            /* 需要重设的事件对象指针*/
```

6.3.2.10 设备对象

```
    /* 生成控制设备 •P71 */
        /* ===IoCreateDevice生成的控制设备具有默认的安全属性,只有管理员权限的用户才能打开它
*/
    NTSTAUS IoCreateDevice(
        IN PDRIVER_OBJECT DriverObject,      /* 驱动对象,直接从入口函数获取
(DriverEntry)*/
        IN ULONG DeviceExtensionSize,        /* 设备扩展的大小*/
        IN PUNICODE_STRING DeviceName OPTIONAL,      /* 设备名字, \\Device\\MyDev
*/
        IN DEVICE_TYPE DeviceType,           /* 设备类型*/
        IN ULONG Characteristics,
            /* 设备属性, (安全打开文件设备)FILE_DEVICE_SECURE_OPEN*/
        IN BOOLEAN Exclusive,    /* 是否独占设备(设置独占设备,该设备只能同时打开一个句
柄)*/
        OUT PDEVICE_OBJECT *DeviceObject);
            /* 生成的设备对象指针(传入一个PDEVICE_OBJECT空指针的地址)*/

        /* ===生成任何用户都可以打开的设备对象 •P72*/
    NTSTATUS IoCreateDeviceSecure(
        IN PDRIVER_OBJECT DriverObject,      /* 驱动对象,直接从入口函数获取
(DriverEntry)*/
        IN ULONG DeviceExtensionSize,    /* 设备扩展的大小*/
        IN PUNICODE_STRING DeviceName OPTIONAL, /* 设备名字*/
        IN DEVICE_TYPE DeviceType,       /* 设备类型*/
        IN ULONG DeviceCharacteristics,          /* 设备属性*/
        IN BOOLEAN Exclusive,    /* 是否独占设备(设置独占设备,该设备只能同时打开一个句
柄)*/
        IN PUNICODE_STRNG DefaultSDDLString,
            /* 特殊格式的字符串表示这个设备对象的安全设置 "D:P(A;;GA;;WD)"允许任何用户访问
该设备*/
        IN LPCGUID DeviceClassGuid,      /* 设备的GUID,全球唯一标识*/
        OUT PDEVICE_OBJECT *DeviceObject);
            /* 生成的设备对象指针(传入一个PDEVICE_OBJECT空指针的地址)*/

    /* 删除设备对象*/
    VOID IoDeleteDevice(PDEVICE_OBJECT DeviceObject);
```

```
'获取设备'
    /*通过名字获取设备对象*/
    NTSTATUS IoGetDeviceObjectPointer(
        IN PUNICODE_STRING ObjectName,  /* 设备的名字*/
        IN ACCESS_MASK DesiredAccess,    /* 期望的访问权限,直接FILE_ALL_ACCESS获取所有
权限*/
        OUT PFILE_OBJECT* FileObject,     /* 返回参数,获取设备对象时会同时获取到一个文件
对象,
                如果不需要ObDereferenceObject释放(解除引用)*/
        OUT PDEVICE_OBJECT* DeviceObject);  /* 返回参数,得到返回设备对象指针*/

    /* 获取驱动程序的设备对象列表*/
    NTSTATUS IoEnumerateDeviceObjectList(
```

```
        PDRIVER_OBJECT DriverObject,
        PDEVICE_OBJECT* DeviceObjectList,
        ULONG DeviceObjectListSize,
        PULONG ActualNumberDeviceObject);

    /*获取设备所在设备栈最顶端得那个设备*/
    PDEVICE_OBJECT IoGetAttachedDevice(PDEVICE_OBJECT DeviceObject);
```

```
'设备绑定'
    /* 根据设备名对设备绑定*/
     NTSTATUS IoAttachDevice(
            IN PDEVICE_OBJECT SourceDevice,     /* 生成的过滤设备*/
            IN PUNICODE_STRING TargetDevice,    /* 串口的设备名*/
            OUT PDEVICE_OBJECT* AttachedDevice);    /* 返回被绑定的设备指针*/
    /* 根据设备对象指针进行绑定*/
        // 适合高版本(Windowss 2000SP4/Windows Xp)以上系统,更安全
    NTSTATUS IoAttachDeviceToDeviceStackSafe(
        IN PDEVICE_OBJECT SourceDevice,     /* 过滤设备*/
        IN PDEVICE_OBJECT TargetDevice,     /* 要被绑定的设备栈中的设备*/
        IN OUT PDEVICE_OBJECT* AttachedToDeviceObject); /* 返回最终被绑定的设备*/

        // 兼容低版本,最后一个参数使用返回值返回(返回NULL,表示失败), 其他都一样
    PDEVICE_OBJECT IoAttachDeviceToDeviceStack(
        IN PDEVICE_OBJECT SourceDevice,     /* 过滤设备*/
        IN PDEVICE_OBJECT TargetDevice);    /* 要被绑定的设备栈中的设备*/

    /* 解除设备绑定,参数传入被绑定设备的指针*/
    void IoDetachDevice(PDEVICE_OBJECT TargetDevice);
```

6.3.3 驱动与应用层的通信

6.3.3.1 驱动层

6.2.3.1.1 IRP

> `IRP(IO_STACK_LOCATION 结构)`输入输出请求包

```
'IRP请求功能代码'
    IRP_MJ_CREATE                   // 请求一个句柄(打开符号链接请求),
//CreateFile();
    IRP_MJ_CLOSE                    // 关闭句柄(关闭符号链接请求), //CloseHandle();
    IRP_MJ_WRITE                    // 传送数据到设备, // WriteFile();
    IRP_MJ_READ                     // 从设备读取数据, ReadFile();
    IRP_MJ_DEVICE_CONTROL           // 控制操作 //DeviceIoControl();
                                // 设备控制码存放 irpsp-
>Parameters.DeviceIoControl.IoControlCode

    IRP_MJ_POWER                    // 电源操作
    IRP_MJ_SHUTDOWN                 // 关闭系统前会产生此消息

    IRP_MJ_PNP                      // 绑定的设备即插即用(设备插拔)
        IRP_MN_REMOVE_DEVICE    // 当设备有序或者意外地拔出时,PNP管理器发送此
IRP(MinorFunction子功能代码)
    /* -->当PNP请求过来时,是没有必要担心还有未决的IRP,这是因为系统要求卸载设备,Windows已处理
掉所有未决IRP*/
```

```
'驱动对象分发函数 •P74'
    DRIVER_DISPATCH DriverDispatch;     // IRP回调(分发) 函数

    /*分发函数中处理请求的第一步是获得请求的当前堆栈空间*/
    IRP_MJ_MAXIMUM_FUNCTION          // 宏,驱动的分发函数数组大小
    IO_STACK_LOCATION                // I/O堆栈结构

    /* 获取指向的IRP(IO_STACK_LOCATION 结构) 堆栈空间*/
        /* 获取当前进程堆栈空间(返回指向指定IRP() 中调用方的 I/O 堆栈位置的指针)*/
        /* 返回获取IRP操作类型*/
    PIO_STACK_LOCATION IoGetCurrentIrpStackLocation(PIRP Irp);
        PIO_STACK_LOCATION irpsp = IoGetCurrentIrpStackLocation(Irp);

    /* 分发函数原型*/
    NTSTATUS cwkDispatch(
        IN PDEVICE_OBJECT dev,  /* 要发送请求的目标对象*/
        IN PIRP irp);       /* 请求内容的数据结构指针*/

    // 原子地 交换函数地址 X64 _InterlockedExchangePointer_HLERelease()
    PVOID InterlockedExchangePointer(PVOID volatile *Target, PVOID value);
```

```
'驱动设备设置数据交互的方式'
    DO_BUFFERED_IO          // 缓冲区方式读写,操作系统将应用层提供的缓冲区数据复制到内核地址里
    DO_DIRECT_IO            // 直接方式读写, 操作系统将用户模式下的缓冲区锁住,内核再映射一份指向地址
    不设置                  // 其他方式读写,直接读取应用层的地址,发生进程切换时会发生错误
    pDevice->Flags |= DO_BUFFERED_IO;       /* 设置驱动设备与3环的交互方式 DO_DIRECT_IO(直接读写)*/

'IRP数据结构'
    Irp->IoStatus.Information;                          /*  返回数据的大小*/
    Irp->IoStatus.Status = STATUS_UNSUCCESSFUL;     /* 返回值,例程执行状态*/
    Irp->PendingReturned;                           /* 例程执行完成,设置IRP状态标识为挂起状态*/
    Irp->CurrentLocation;                           /* 为真时,当前在IRP栈最底端,否则不是*/

    /*写数据缓冲区描述*/
    irp->MDLAddress     // 应用层的地址空间映射给内核,会在页表中增加一个映射
    irp->AssociatedIrp.SystemBuffer // 把应用层(R3层)内存空间中的缓冲区数据拷贝到内核空间(高2G)
    irp->UserBuffer     // 应用层缓冲区地址直接放在UserBuffer给内核空间访问,存在进程不同访问错误数据


'IRP栈空间操作'
    /*获取当前IRP栈空间(返回指向指定 IRP() 中调用方的 I/O 堆栈位置的指针)*/
    PIO_STACK_LOCATION IoGetCurrentIrpStackLocation(PIRP Irp);

    /* 发送一个IRP给指定设备对象关联的驱动程序,这里会把IO栈指针减1*/
    /* IoSkipCurrentIrpStackLocation后,经过这样调用后使得下一层的驱动里的IO栈就是当前所用的IO栈*/
    NTSTATUS IoCallDriver(PDEVICE_OBJECT DeviceObject,
        __drv_aliasesMem PIRP Irp);
```

```
    /* 跳过当前设备IRP栈空间， 下层驱动程序将收到与本层驱动程序收到的一模一样的
IO_STACK_LOCATION*/
    /* 跳过当前栈，其实就是使IO栈指针加1*/
    void IoSkipCurrentIrpStackLocation(PIRP Irp);

    /* 复制当前IRP栈空间,传递给下层驱动,与IoSkipCurrentIrpStackLocation冲突,会*/
    /* 覆盖下层驱动的IoCompletion(完成例程)*/
    void IoCopyCurrentIrpStackLocationToNext(PIRP Irp);


    /* 注册一个IoCompletion例程,当下层完成指定IRP请求时会调用该例程*/
    void IoSetCompletionRoutine(PIRP Irp, PIO_COMPLETION_ROUTINE
CompletionRoutine, PVOID ConText,
        BOOLEAN InvokeOnSuccess, BOOLEAN InvokeOnError, BOOLEAN InvokeOnCancel);

    /*IoCompletion回调函数*/
    NTSTATUS IoCompletionRoutine(PDEVICE_oBJECT DeviceObject, PIRP Irp, PVOID
Context);


    /* IoCompleteRequest函数表示调用者已完成给定的I/O请求的所有处理,并将给定的IRP返回给I/O
管理器*/
        // IoCompleteRequest宏函数包装成IofCompleteRequest
    void IoCompleteRequest(PIRP Irp, CCHAR PriorityBoost);
    // --->
        {Irp->IoStatus.Information = 0;
        Irp->IoStatus.Status = STATUS_UNSUCCESSFUL;
        IoCompleteRequest(Irp, IO_NO_INCREMENT); }

void IoMarkIrpPengding
```

```
'创建IRP'
    PIRP IoAllocateIrp(CCHAR StackSize, BOOLEAN ChargeQuote);

'获取R3层数据交换发送虚拟地址' •P114
    /* 获取数据在应用层缓冲区的虚拟地址,映射到内核*/
    // Windows 2000以下版本使用
    PVOID MmGetSystemAddressForMdl(MDL);
    // Windows高版本使用
    MmGetSystemAddressForMdlSafe();
        PBYTE* pBuf = MmGetSystemAddressForMdlsafe(irp->MalAddress,
NorMalPagePriority);
        ULONG length = irpsp->Parameters.Write.Length;      /* 获取写缓冲区的长度*/
```

```
'电源例程'
    //向电源管理器发送信号,表示驱动程序已准备好下一个电源IRP
    void PoStartNextPowerIrp(PIRP Irp);

    // 将电源IRP传递给设备栈中的下一个驱动程序,特殊例程替代IoCallDriver
    NTSTATUS PoCallDriver(PDEVICE_OBJECT DeviceObject, _drv_aliasesMen PIRP
Irp);
```

6.2.3.1.2 符号链接

```
'建立控制设备的符号链接'
    /* 设置一个设备对象名称和该设备的用户可是名称之间的字符链接*/
        /* 符号链接在Windows中是全局存在,如果符号链接的名字已在系统中存在了,会执行失败*/
    NTSTATUS IoCreteSymbolicLink(
        IN PUNICODE_STRING SymbolicLinkName, /* 符号链接名 \\??\\MyDev */
        IN PUNICODE_STRING DeviceName);         /* 设备名*/

'删除设备对象符号链接'
    NTSTATUS IoDeleteSymbolicLink(PUNICODE_STRING SymbolicLinkName);    /* 符号链
接名*/
```

```
'通过端口/管道文件与应用层通信'
NTSTATUS FLTAPI FltCreateCommunicationPort(
            PFLT_FILTER Filter,
          PFLT_PORT* ServerPort,
            POBJECT_ATTRIBUTES ObjectAttributes,
            PVOID ServerPortCookie,
            PFLT_CONNECT_NOTIFY ConnectNotifyCallback,
            PFLT_DISCONNECT_NOTIFY DisconnectNotifyCallback,
            PFLT_MESSAGE_NOTIFY MessageNotifyCallback,
            LONG MaxConnections);
```

6.2.3.1.3 WorkItem

```
'例程分配工作项'
    PIO_WORKITEM IoAllocateWorkItem(
        PDEVICE_OBJECT DeviceObject);   /* 调用方的驱动程序对象或调用方的设备对象*/
'将WorkItem例程与工作项关联,并将工作项插入队列,供系统辅助线程以后处理'
    void IoQueueWorkItem(
        __drv_aliasesMem PIO_WORKITEM IoWorkItem,
            /* 指向由IoAllocateWorkItem 分配或由 IoInitializeWorkItem初始化的
IO_WORKITEM结构*/
        PIO_WORKITEM_ROUTINE WorkerRoutine, /* 指向工作项例程的指针*/
        WORK_QUEUE_TYPE QueueType,
            /* 要处理工作项的系统辅助线程的类型。驱动程序必须指定 DelayedWorkQueue*/
        __drv_aliasesMem PVOID Context);
            /* 指定工作项的特定于驱动程序的信息。系统将此值作为上下文参数传递到 WorkItem*/
```

6.2.3.2 应用层

6.2.3.2.1 服务操作

```
'打开服务管理器返回服务管理器句柄,使用完关闭句柄'
    SC_HANDLE WINAPI OpenSCManager(LPCTSTR lpMachineName,
                                   LPCTSTR lpDatabaseName,
                                   DWORD dwDesiredAccess);
'关闭服务管理器/服务的句柄'
    BOOL CloseServiceHandle(SC_HANDLE hSCObject);
```

```
'注册(创建)服务后自动打开服务并返回句柄,需要使用CloseServiceHandle关闭'  •P16
    SC_HANDLE WINAPI CreateService(SC_HANDLE hSCManager,
                                   LPCTSTR lpServiceName,
                                   LPCTSTR lpDisplayNmae,
                                   DWORD dwDesiredAccess,
                                   DWORD dwServiceType,
                                   DWORD dwStartType,
                                   DWORD dwErrorControl,
                                   LPCTSTR lpBinaryPathName,     // 驱动文件的路径
                                   LPCTSTR lpLoadOrderGroup,
                                   LPDWORD lpdwTagld,
                                   LPCTSTR lpDependencies,
                                   LPCTSTR lpserviceStartName,
                                   LPCTSTR lpPassword);

'打开已经存在的服务的句柄(使用完毕需使用CloseServiceHandle关闭句柄)' •P18
    SC_HANDLE WINAPI OpenService(SC_HANDLE hSCManager,
                                 LPCTSTR lpServiceName,
                                 DWORD dwDesiredAccess );

'启动已存在的服务(P18)，作为内核驱动服务dwNumServiceArgs和lpServiceArgVectors可以为NULL'
    BOOL WINAPI StartService(SC_HANDLE hService,
                             DWORD dwNumServiceArgs,
                             LPCTSTR *lpServiceArgVectors);

'停止/暂停/恢复服务->(服务句柄， 服务操作类型， 传出参数:服务的最新状态)' •P19
    BOOL WINAPI ControlService(
    SC_HANDLE hService,
    DWORD dwControl,
    LPSERVICE_STATUS lpServiceStatus);  //停止必须要一个SERVICE_STATUS接收

'删除服务，先OpenService打开有删除权限的服务获取句柄 '
    BOOL WINAPI DeleteService(SC_HANDLE hService);
```

6.2.3.2.2 设备操作

```
'创建/打开文件 和最常用的I/O设备驱动'
    HANDLE CreateFile(
        LPCTSTR lpFileName,          /* 普通文件名/设备文件名 (符号链接名) \\.\MyDev */
        DWORD dwDesiredAccess,       /* 访问模式(写/读)*/
        DWORDdwShareMode,            /* 共享模式*/
        LPSECURITY_ATTRIBUTES lpSecurityAttributes,          /* 指向安全属性的指针*/
        DWORD dwCreationDisposition,     /* 如何创建 */
        DWORD dwFlagsAndAttributes,      /* 文件属性 */
        HANDLE hTemplateFile);           /* 用于复制文件的句柄 */
    DWORD CloseHandle(HANDLE);       // 关闭打开的文件驱动

'生成符号'
    IoRegisterDeviceInterface();

'获取设备'
    IoGetDeviceObjectPointer();

'将控制代码直接发送到指定的设备的驱动程序,使相应的设备执行相应的操作(32位应用通信64位驱动不会发生反应)'
```

```
    BOOL DeviceIoControl(
        HANDLE hDevice,              /* 设备句柄 */
        DWORD dwIoControlCode,       /* 操作的控制代码 */
        LPVOID lpInBuffer,           /* 应用层传递给驱动的数据缓冲区 */
        DWORD nInBufferSize,         /* 应用层传递给驱动缓冲区的大小 */
        LPVOID lpOutBuffer,          /* 驱动给应用层传递数据缓冲区 */
        DWORD nOutBufferSize,        /* 驱动给应用层传递的缓冲区大小 */
        LPDWORD lpBytesReturned,     /* 驱动实际传递给应用层的数据大小 */
        LPOVERLAPPED lpOverlapped    /* 指向 OVERLAPPED 结构的指针 */
);
    # 生成一个自己的设备控制请求功能号(宏函数)
    '生成一个拥有写入权限的控制代码'
    void CTL_CODE(  FILE_DEVICE_UNKNOWN,     /* 设备类型*/
                0X911,                   /* 生成这个功能号的核心数字(0x800-0xfff)*/
                METHOD_BUFFERED,         /* 缓冲的的方式*/
                FILE_WRITE_DATA);        /* 操作需要的权限(写入权限),
FILE_ANY_ACCESS(所有权限)*/
    '生成一个拥有读取权限的控制代码'
    void CTL_CODE(  FILE_DEVICE_UNKNOWN,     /* 设备类型*/
                0X912,
                METHOD_BUFFERED,
                FILE_READ_DATA)          /* 设备读取权限*/

'已完成给指定的 I/O请求的所有处理,并返回给定的IRP的 I/O 管理器'
    void IoCompleteRequest(
    PIRP Irp,                        /* -->指向要完成的IRP指针 */
    CCHAR PriorityBoost              /* -->指定一个系统定义的CCHAR常数
IO_NO_INCREMENT */
);
```

6.2.3.2.3 Win64上32位App重定向文件/注册表

```
•P83
'关闭文件重定向,使32位应用程序在64位系统下运行时访问真实Syetem32目录'
    BOOL WINAPI Wow64DisableWow64FsRedirection(
                _Out_ PVOID* OldValue);      /* 输出参数,保存原来文件重定向器的状态,方便
恢复*/

'恢复文件重定向'
    BOOL WINAPI Wow64RevertWow64Redirection(
                _In_ PVOID OldValue);    /* 原来文件重定向器的状态*/

'注册表被重定向'
    /* 执行32位APP时,创建在注册表项在32位子表里面,执行64位APP时,创建在64位注册表*/
    RegCreateKeyEx(HKEY_LOCAL_MACHINE, TEXT("SoftWare\\Hello"), 0 NULL, 0,
                   KEY_READ, NULL, &hKey, NULL);
    CloseHandle(hKey);
    /* 32位App指定注册项在64位注册表
    (KEY_WOW64_64KEY通知WOW64子系统打开64位注册表. KEY_WOW64_32KEY.....) */
    RegCreateKeyEx(HKEY_LOACL_MACHINE, TEXT("Software\\Hello"), 0, NULL, 0,
                  KEY_READ|KEY_WOW64_64KEY, NULL, &hKey, NULL);
    CloseHandle(hKey);
```

6.3.4 驱动设备

> `类驱动`是指`统一管理一类设备`的驱动程序.
>
> `端口驱动`是指类驱动之下和实际硬件交互的的驱动

6.3.4.1 键盘驱动设备(KbdClass)

> PS/2键盘的设备栈,没有自己另外安装其他键盘过滤程序
>
> 最顶层的设备对象是驱动`KbdClass`生成的设备对象,驱动名:`"\\DRIVER\\Kbdclass" 键盘驱动(类驱动)`
>
> 中间层的设备对象是驱动`i8042ptr/Kbdhid`生成的设备对象(`端口驱动`)
>
> PS/2键盘i8042ptr: `"\\DRIVER\\i8042ptr` USB键盘Kbdhid: `"\\Driver\\Kbdhid"`
>
> 最底层的设备对象是驱动`ACPI`生成的设备对象(`底层驱动`)

```
    class KbdClass              //键盘驱动类(不管是USB键盘还是PS/2都经过他)
    extern POBJECT_TYPE* IoDriverObjectType;        // 键盘驱动的对象类型

    /* 键盘设备读取出的键扫描码*/
    typedef struct _KEYBOARD_INPUT_DATA
    {
        USHORT UnitId;  // 对于\\DEVICE\\KeyboardPort0这个值是
0,\\DEVICE\\KeyboardPort1这个值是1,依此类推
        USHORT MakeCode;        // 扫描码
        USHORT Flags;       // 标志,标志一个键按下还是弹起
        USHORT Reserved;    // 保留
        ULONG ExtraInformation;     // 扩展信息
    }KEYBOARD_INPUT_DATA, *PKEYBOARD_INPUT_DATA;

    /* Flags可能的取值*/
    #define KEY_MAKE                  0         // 键盘按下
    #define KEY_BREAK                 1         // 键盘弹起
    #define KEY_E0                   2
    #define Key_E1                   4
    #define KEY_TERMSRV_SET_LED       8
    #define KEY_TERMSRV_SHADOW        0x10
    #define KEY_TERMSRV_VKPACKET      0x20
```

KbdClass类驱动

```
'键盘类驱动输入数据队列'
            struct {
            PKEYBOARD_INPUT_DATA InputData,     // 指向输入数据队列的开头
            PKEYBOARD_INPUT_DATA DataIn,        // 指向要进入队列的新数据,被放在队列中
的位置
            PKEYBOARD_INPUT_DATA DataOut,       // 指向要出队列的数据,在队列中的开始位
置
            PKEYBOARD_INPUT_DATA DataEnd        // 指向输入数据队列的结尾
            ULONG InputCount };                 // 输入数据队列中数据的个数

    /* Kbdclass提供的类服务回调函数,类服务回调将输入数据从设备的输入数据缓冲区传输到类数据队列
*/
    VOID _stdcall KEYBOARDCLASSERVICECALLBACK(
            IN PPDEVICE_OBJECT DeviceObject,
```

```
            IN PKEYBOARD_INPUT_DATA InputDataStart,
            IN PKEYBOARD_INPUT_DATA InnputDataEnd,
            IN OUT PULONG InputDataConsumed);
```

i8042ptr驱动

```
'i8042ptr驱动自定义设备扩展 端口键盘输入数据队列`
            struct{
            PKEYBOARD_INPUT_DATA InputData,      // 指向输入数据队列的开头
            PKEYBOARD_INPUT_DATA DataIn,         // 指向要进入队列的新数据,被放在队列中
的位置
            PKEYBOARD_INPUT_DATA DataOut,        // 指向要出队列的数据,在队列中的开始位
置
            PKEYBOARD_INPUT_DATA DataEnd         // 指向输入数据队列的结尾
            ULONG InputCount };                  // 输入数据队列中数据的个数

    I8042KeyboardInterruptService();        /* 会调用上层类驱动的处理输入数据队列的回调
函数,文档未公开*/

    /* 将DPC排队等待执行 (进行更多处理的延迟过程调用)*/
    BOOLEAN KeInsertQueueDpc(PRKDPC Dpc, PVOID SystemArgument1, __drv_aliasesMem
PVIOD SystemArgument2);
```

6.3.4.2 磁盘驱动设备(WDF框架_[FMDF])

```
    struct _WDF_DRIVER_CONFIG config;           /* 驱动程序的可配置项*/

    /* WDF驱动回调函数原型*/
    NTSTATUS EvtDriverDeviceAdd(
        IN WDFDRIVER Driver,
        IN PWDFDEVICE_INIT DeviceInit);

    // 初始化WDF驱动结构
    void WDF_DRIVER_CONFIG_INIT(
        PWDF_DRIVER_CONFIG Config,                  /* WDF驱动配置项*/
        PFN_WDF_DRIVER_DEVICE_ADD EvtDriverDeviceAdd);  /* WDF驱动回调函数*/

    // 为驱动程序创建 WDF框架驱动程序对象
    NTSTATUS WdfDriverCreate(
        PDRIVER_OBJECT DriverObject,                /* 驱动对象*/
        PUNICODE_STRING RegistrPath,                /* 驱动注册表路径*/
        PWDF_OBJECT_ATTRIBUTES DriverAttributes,    /* WDF驱动的驱动对象属性,
WDF_NO_OBJECT_ATTRIBUTES*/
        PWDF_DRIVER_CONFIG DriverConfig,            /* 驱动可配置项*/
        WDFDRIVER *Driver);                        /* 创建成功的WDF驱动对象,
WDF_NO_HANDLE*/

    // 通过WDF设备获取设备对应的驱动对象句柄
    WDFDRIVER WdfDeviceGetDriver(WDFDEVICE Device);

    // 获取驱动在注册表中的路径
    PWSTR WdfDriverGetRegistryPath(WDFDRIVER Driver);
```

```
    WDFDEVICE Device;                          /* HANDLE类型, WDF驱动设备*/
    struct _WDF_OBJECT_ATTRIBUTE DeviceAttr;         /* 设备对象属性*/

    // 初始化WDF_OBJECT_ATTRIBUTES结构,并将对象的驱动程序定义的上下文信息插入到结构中
    void WDF_OBJECT_ATTRIBUTES_INIT_CONTEXT_TYPE(
        WDF_OBJECT_ATTRIBUTES* _attributes,
         _contexttype);

'WDF设备操作'
   // 创建一个WDF框架设备
    NTSTATUS WdfDeviceCreate(
    PWDFDEVICE_INIT *DeviceInit,
    PWDF_OBJECT_ATTRIBUTES DeviceAttributes,
    WDFDEVICE *Device);                        // 返回新创建的设备指针

    // 获取设备的扩展 (DEVICE_EXTENSION 自定义设备扩展结构)
    DEVICE_EXTENSION* DeviceGetExtension(
        WDFDEVICE device);

    // 将名字分配给设备的设备对象
    NTSTATUS WdfDeviceInitAssignName(
        PWDFDEVICE_INIT DeviceInit,
        PCUNICODE_STRING DeviceName);

    // 设置指定设备的设备类型 [Applies to KMDF only]
    void WdfDeviceInitSetDeviceType(
        PWDFDEVICE_INIT DeviceInit,
        DEVICE_TYPE DeviceType);         // FILE_DEVICE_DISK磁盘设备类型

    // 设置驱动程序访问 设备IRP读写请求中数据缓冲区的访问方法 [Applies to KMDF and UMDF]
    void WdfDeviceInitSetIoType(
        PWDFDEVICE_INIT DeviceInit,
        WDF_DEVICE_IO_TYPE IoType);      // WdfDeviceIoDirect 直接访问

    // 指定设备是否为独占设备  [Applies to KMDF only]
    void WdfDeviceInitSetExclusive(
        PWDFDEVICE_INIT DeviceInit,
        BOOLEAN IsExclusive);
```

```
    WDFQUEUE wdfQueue;                             /* HANDLE类型, WDF驱动队列 */
    struct _WDF_IO_QUEUE_CONFIG wdfQueueConf;        /* WDF框架 队列配置信息*/

    // 初始化驱动程序的WDF_IO_QUEUE_CONFIG结构(队列)    [Applies to KMDF and UMDF]
    void WDF_IO_QUEUE_CONFIG_INIT_DEFAULT_QUEUE(
        PWDF_IO_QUEUE_CONFIG Config,
        WDF_IO_QUEUE_DISPATCH_TYPE DispatchType);   // 队列请求的调度类型
WdfIoQueueDispatchSequential

    // 为指定的WDF设备创建I/O队列
    NTSTATUS WdfIfQueueCreate(
        WDFDEVICE Device,
        PWDF_IO_QUEUE_CONFIG Config,
        PWDF_OBJECT_ATTRIBUTES QueueAttributes,
        WDFQUEUE *Queue);
```

QT嵌入式开发

Qt基础快捷键

```
ctrl + R                    //编译运行
ctrl + B                    //编译
ctrl + /                    //注释
ctrl + F                    //查找 ESC 退出
ctrl + i                    //自动对齐
ctrl + shift +方向键       //整行移动
F4                          //文件切换 ( cpp 和 h)
```

头文件

```
#include <QApplication>          /* Qt的所有基础头文件 */
#include <QWidget>               /* 窗口控件的父类头文件[Widget] */
#include <QPushButton>           //常用按钮控件
#include <QDebug>                //命令控制
#include <QMenuBar>              /* 菜单栏控件 */
#include <QStatusBar>            /* 工具栏 */
#include <QLabel>                /* 文本信息 */
#include <QDockWidget>           /* 铆接部件 */
#include <QTextEdit>             /* 文本编辑控件 */
```

QT数据类型转换

```
temp.toUtf8().data();           //Qstring转char*调用toUtf8()转换为 QBtyeArray,再调用data()转换为
    /* Qstring 默认会加双引号 */
    char* b = a.toUtf8().data();
QString::number(a);             //int a 转换为QString类型
```

lanbda表达式

```
"lambda"  /* lambda表达式创建匿名函数c++11新特性  Qt5以下需在.pro文件声明 [CONFIG += c++1] */
    int num = []()->int{ return 100; }();
            [ ]             /* 不引入任何任何函数对象参数 */
            [=]             /* 引入表达式外部变量, 地址传递 */
            [&]             /* 引用传递外部变量,在匿名函数内部使用 */
            [a]             /* 按a值的方式传递, 变量只读状态 */
            [&a]            /* 将a按引用传递 */
            [a, &b]         /* a按值传递, b按引用方式传递 */
            [=, a, &b]      /* 除了a使用值传递,b使用引用传递,其它都按引入变量(地址)传递 */
"mutable关键字"
    /*使用mutable关键字可以修改lambda按值传参[a]使拷贝的数值*/
    connect(btn, &QPushButton::clicked, [a]()mutable { int a= 0; })
```

信号和槽

```
"信号和槽"  /* 绑定控件 */
/* connect(信号的发送者[指针], 发送的信号[函数入口地址], 信号的接收者[指针], 信号处理[槽函数入口])*/
    connect(btn, &QPushButton::clicked, this, &QWidget::close);        //链接信号
            &QPushButton::clicked               //点击信号
            &QPushButton::pressed               //按下信号(鼠标按下)
            &QPushButton::released              //释放(鼠标抬起)
            &QPushButton::triggered             //触发
            &QPushButton::toggled               //开关状态
            &QPushButton::function
    connect(btn, SIGNAL(clicked()), this, SLOT(close()));
                                /* QT4版本写法,Qt5兼容写法, [致命缺点参数类型不做检查] */
    connect(btn, SIGNAL(clicked(QString)), this, SLOT(function(QString))); /* 有参*/
    coonetc(btn, &QPushButton::clicked, [=](){ int a= 0; })          /* 匿名函数 */
                /* 当connect链接时,内部会将按钮进入锁状态[只读],用&传参会报错,使用= */

'槽函数'    /* 当绑定多个槽函数时,槽函数的执行顺序时随机的,不能控制 */

    &QWidget::function
        close()                     // 关闭窗口的槽信号
```

```
# 断开信号和槽的链接
    disconnect(btn, &QPushButton::clicked, this, &QWidget::close);     //断开链接
```

> 当信号或者槽发生重载时, 需要使用函数指针明确指向
>
> 信号和槽的参数个数并不一定一致,信号的参数可以多余槽,但槽函数的参数个数不能多余信号
>
> 信号和槽的参数必须一一对应

```
# 自定义信号和槽  (emit  触发自定义信号)
    自定义信号        //写在 signals下  返回值是 void 只需要声明不需要实现 可以带参数和不带参数发生重载
    自定义槽函数      //写在 public slots下,QT5版本后可以写在 public下, 可以发生重载, 需要有声明有实现
# 触发自定义信号和槽
    emit btn->pressed();                /* 触发信号 */
```

QMainWindow基础控件

主窗口

```
# 主窗口 Widget
    resize(600, 400);                   /* 重置窗口大小[没固定,可拖拽大小] */
    this->setWindowTitle("title");      /* 设置窗口标题 */
    this->setFixedSize(x, y);           /* 固定窗口大小 */
    this->setFixedHeight(600);          /* 设置固定的高 */
    this->setFixedWidth(400);           /* 设置固定的宽*/
```

按钮

```
# 设置按钮
    QPushButton *btn1 = new QPushButton("Text", this);
    /* 创建一个按钮[设置按钮显示名字,按钮的父窗口] */
    {
        QPushButton *btn = new QPushButton;
        btn->setParent (this);          /* 设置按钮的父窗口,如果是结构函数内调用可以用
this代替 */
        btn->setText("text");           /* 设置按钮的显示文本 */
        btn->move(x, y);                /* 移动按钮位置,设置坐标 */
    }
```

菜单栏

```
#include <QmenuBar>
    QmenuBar *bar = menuBar();              //菜单栏, 只能有一个  #include
<QmenuBar>
        this->setMenuBar(bar)                   //把菜单栏设置窗口
        QMenu *fileMenu = bar -> addMenu("name");   //添加菜单栏
            QAction *action1 = fileMenu -> addAction("name");   //添加菜单项
            qm1->addSeparator();                //在菜单项之间添加分割线
```

工具栏

```
#include <QToolBar>
    QToolBar *toolBar = new QToolBar(this); //工具栏, 可以创建多个工具,用new创建需指定
父窗口
        addToolBar(Qt::LeftToolBarArea, toolBar);      //给某个工具栏设置窗口,并设置
停靠方向
            toolBar->setMoveable(false);                //设置是否允许拖拽移动
            toolBar->setAllowedAreas(Qt::LeftToolBarArea |
Qt::ReightToolBarArea);
                                    /* 设置只允许工具栏停靠的位置  (左 | 右 | 上 | 下) */
            toolBar->setFloatable(false);           //设置工具栏是否允许浮动
            toolBar->addAction("name");             //工具栏新建菜单项
            toolBar->addSeparator();                //工具栏菜单项之间添加分割线
```

状态栏

```
#include <QStatusBar>
/* 状态栏 只能有一个 */
    QStatusBar *stBar = statusNar();
    setStatusBar(stBar);                    //状态栏设置窗口,
    QLable *label1 = new QLabel("text");    //创建新状态栏信息 #include <QLabel (文
本控件)>
        stBar->addWidget(label1);           //将文本信息绑定到状态栏中 (从左往右添加信
息)
        stBar->addPermanentWidget(label1);  //将文本信息绑定到状态栏中 (从右往左添加信
息)
```

铆接部件/浮动窗口

```
#include <QDockWidget>
/* 浮动窗口(铆接部件)可以有多个,能从主窗口中拖拽出来[围绕 核心部件/中心部件使用] */
QDockWidget *dkWidget = new QDockWidget("text", this);
    addDockWidget(Qt::BottomDockWidgetArea, dkWidget); //设置浮动窗口  并设置默认停靠位
    dkWidget->setAllowedAreas(Qt::LeftToolBarArea | Qt::ReightToolBarArea)
                                                                //设置浮动窗口允许停靠的位置
```

核心部件

```
#include <QTextEdit>
/* 中心部件一般只设置一个 */
    QTextEdit *textEdit = new QtextEdit(this);      //文本编辑  #include <QTextEdit>
    setCentralWidget(textEdit);                  //设置为中间内容
```

UI界面

资源文件添加

```
ui->object->setIcon(QIcon(": + 前缀 + src/a.png"));       //添加小图标
    ui>object->setIcon(QIcon(:/images/a.png));
```

UI界面绑定信号和槽

```
# UI控件绑定信号和槽
connect(ui->object, &QAction::triggered, [=](){ });
    connect(ui->pushbotton_1, &QPushButton::clicked, this,[=](){  });

# UI控件
        push Button                 //按钮
        Tool Button                 //工具按钮, 一般只用来显示图片
        Radio Button                //单选按钮
            ui->object->setChecked(true);   //单选框默认选中
        Group Box                   //分组框 (同一类型的单选按钮放在一个分组框里面)
            connct(ui->object, &QCheckBox::stateChanged, [=](int state){ });
                        //将选中的状态放入参数 state中,未选中值 选中值为2 半选中为1
```

对话框

模态对话框和非模态对话框

```
/* 模态对话框和非模态对话框 */
#include <QDialog>
    /* 模态对话框(不可以对其他窗口进行编辑)  非模态对话框(可以对其他对话框进行编辑) */
    /* 创建模态对话框 */
        QDialog dl(this);
            dl.resize(500, 350);                //指定对话框大小
            dl.exec();                          //窗口阻塞
    /* 创建非模态对话框 */
        QDialog* dl = new QDiallog(this);
            dl->resize(120, 40);                //指定对话框大小
            dl->show();                         //显示对话框
            dl->setAttribute(Qt::WA_DeleteOnClose);  //设置对话框属性 (当窗口关闭时new出来的对象同步释放掉)
```

标准对话框

```
/* 消息对话框 [默认模态对话框]*/
#include <QMessageBox>

    QMessageBox::critical(this, "title", "value");       //错误对话框
        /* 参数: 【父窗口,标题,中间文本内容】*/
    QMessageBox::information(this, "title", "value");    //信息提示对话框
        /* 参数: 【父窗口,标题,中间文本内容】*/
    QMessageBox::warning(this, "title", "value");        //警告对话框
        /* 参数: 【父窗口,标题,中间文本内容】*/
    QMessageBox::question(this, "title", "value",
                          QMessageBox::save | QMessageBox::Cancel,
QMessageBox::Cancel);
        //询问对话框 返回值按键类型( (父窗口, 标题, 中间文本, 按键类型(默认Yes/No), 默认关联回车的按键)
            /* 参数: 【父窗口,标题,中间文本内容,new几个按钮[按钮类型],关联回车的按键】*/
                    QMessageBox::save           // 确定按钮
                    QMessageBox::Cancel         // 取消按钮

                if(QMessageBox::Cancel == (QMessageBox::question(this, "title",
"value",
                                                 QMessageBox::save |
QMessageBox::Cancel)))                                                  //判断用户是否点击了第一个按钮

/* 颜色选择对话框 */
#include <QColorDialog>              //颜色获取对话框,返回用户选择的颜色
       Qcolor a = QColorDialog::getColor(); // 会弹出颜色选择对话框，接收用户选择的颜色属性
        qDeBug<<a.red()<<a.green()<<a.blue();

/* 文件对话框 */
#include <QFileDialog>
    QString fileName = QFileDialog::getOpenFileName(this, "title","./src/", "
(*.exe)");
        /*打开文件选择框【父窗口,框标题,默认打开的目录路径,筛选文件后缀名[类型]】, 返回值是选择的文件路径 */

/* 字体选择对话框 */
#include <QFontDialog>
    bool ok;
```

```
    QFont ft = QFontDialog::getFont(&ok, QFont("微软雅黑", 36))
                                //获取用户选取的的字体信息(参数2 为提前设好的默认)
        ft.family();         // 字体 返回值QString类型
        ft.pointSize();      // 字号
        ft.bold();           // 是否加粗
        ft.italic();         // 是否倾斜
```

界面布局控件

```
" Layouts "
    Vertical Layout                          // 水平布局
    Horizontal Layout                        // 垂直布局
    Grid Layout                              // 网状布局/栅栏布局(两行两列)
    Form Layout                              // 网状布局
```

```
/* 单选按钮默认选中*/
    ui->object->setChecked(true);                // 按钮默认被选中
        connect(ui->reboot, &QRadioButton::clicked, [](){ });   // 捕获按钮选中信号

/* 列表 */
#include <QListWidgetItem>
    listWidget
        QListWidgetItem* item = new QListWidgetItem("text");  //创建的列表行
        item->setTextAlignment(Qt::AlignHCenter);            //设置文本水平居中对齐
        ui->object->addItem(item);                    //将创建的行加入到列表单元中
            ui->object->addItems(QStringList()<<"第一行"<<"第二行"<<"第三行"<<"第四
行");
                                    //一次性将不同行的文本加入列表,但不能设置对齐方式

    treeWidget                    //树列表
        ui->object->setHeaderLabels(QStringList()<<"姓名"<<"性别");  //设置头的标签
        QTreeWidgetItem* liItem = new QTreeWidgetItem(QStringList()<<"第一个根列表"
<<"第二个根列表");            // 创建根列表
        ui->object->addTopLevelItem(liItem);     //将创建的根列表加载到顶层
            QStringList here1;               //添加子节点并设置文本
            here1<<"张三"<<"男";
            QTreeWidgetItem* li1 = new QTreeWidgetItem(here1);  //创建子节点变量
            liItem->addChild(li1);               //根列表挂载子节点

    tableWidget                   //表格
        ui->object->setColumnCount(3);               //设置表格有多少列
        ui->object->setHorizontalHeaderLabels(QStringList()<<"姓名"<<"性别"<<"年
龄");
                                                     //设置每列的水平表头
        ui->object->setRowCount(5);                 // 设置行数
        ui->object->setItem(0, 0, new QTableWidgetItem("亚瑟"));
                                                 //设置正文(1,2参数为表格的列和行)
            QStringList nameList;
            nameList<<"张三"<<"李四"<<"王二麻子"<<"翠花"<<"二狗子";
            QList<QString> sexList;
            sexList<<"男"<<"男"<<"男"<<"女"<<"男";
            int age[] = {10, 20, 30, 40, 50}
            for(int i=0; i<3; i++)
```

```
            {
                int col =0;
                ui->object->setItem(i, col++, new
QTableWidgetItem(nameList[i]));
                ui->object->setItem(i, col++, new
QTableWidgetItem(sexList.at[i]));
                ui->object->setItem(i, col++, new
QTableWidgetItem(QString::mumber(age)
            );
    toolBox                 //分组栏(QQ好友分组)
    tabWidget               //分组标签(浏览器上面不同页面之间的)
    stackedWidget           //栈控件(类似于Qt左侧栏的菜单页)需绑定按钮切换页
        connect(ui->btn, &QPushButton::clicked, [=](){
               ui->stackedWidget->setCurrentIndex(0);   //绑定按钮,点击跳转至第0号位
页面
        });
    frame                           //做边框用
    dockWidget                      //浮动窗口
    comboBox                        //下拉框
        ui->object->addItem("text");//添加下拉框选项(文本)
    fomtComboBox                    //字体下拉框
    lineEdit                        //单行文本框
    textEdit                        //文本框(可修改里面的字体倾斜颜色)
    plainTextEdit                   //单纯的文本框(不能设置字体倾斜颜色)
    spinBox                         //整数码表(时间戳)
    dounleSpinBox                   //小数码表
    timeEdit                        //时间
    dateEdit                        //日期
    date/timeEdit                   //时间和日期
    horizontalSlider                //滚动条(视频播放)
    label                           //标签(可加载图片)
        ui->object->setPixmap(QPixmap(": + 前缀 + src/a.png"));  //显示图片
        QMovie* img = new QMovie(": + 前缀 + src/a.gif")           //创建动态图变量
        ui->object->setMovie(ing);                                //动态图变量绑定标签
        img->start();                                             //动态图播放
```

Qt事件

```
# 鼠标事件
    void enterEvent(QEvent *);                  //鼠标进入事件
    void leaveEvent(QEvent *);                  //鼠标离开事件
    void mousePressEvent(QMouseEvent *ev);      //鼠标按下
        QString str = QString("鼠标按下了 x=%1, y=%2").arg(ev->x()).arg(ev->y());
        if(ev->button() == Qt::LeftButton)          //判断是否是鼠标左键被按下获取点击
位置的信息
    void mouseReleaseEvent(QMouseEvent *ev);    //鼠标释放
    void mouseMoveEvent(QMouseEvent *ev);       //鼠标移动
        if(ev->buttons() & Qt::LeftButton)  //判断是否为左键按下时移动(连续动作用
buttons和位与运算判断不能用 ==)
    this->serMouseTracking(true);           //设置鼠标追踪(代码写在控件的构造函数里,当
鼠标进入控件范围时进行追踪)
```

```
#定时器事件timerEvent  <QTimerEvent>
    void timerEvent(QTimerEvent *t);    //重构定时器函数
```

```
        static int num =1;                  //构建临时静态变量(临时静态变量只会被初始化一次)
        int time1 = startTimer(1000);       //启动定时器 参数单位为毫秒级 返回定时器唯一的ID
号
        if(e->timerId() == time1){ };       //当多个不同时间段定时器事件时,判断事件Id执行操
作
    QTimer *timer_A = new QTimer(this);
                                    //在Widget构造函数中New一个QTimer类 #include
<QTimer>
        timer_A->start(500);                //启动定时器
        connect(timer_A, &QTimer::timeout, [=](){ qDebug()<<"";});
                                      //绑定事件"匿名"函数   timeout每隔指定毫秒抛出信号
    QTime Stop | Continue
        connect(ui->pushButton, &QPushBUtton::clicked, [=](){
               timer_A->stop();                 //暂停时间事件
        });
```

```
#Event事件分发器
    bool Event(Qevent *e);                  //鼠标分发器,可重写里面的事件捕获函数,在头文件声
明源文件实现
        bool MyWidget::event(QEvent *e)
        {
            if(e->type() == QEvent::MouseButtonPress) { return true;}
                                                //需要捕获的事件,函数体写返回ture表示事件拦
截
            return Widget::event(e);            //其他不需要捕获的事件返回给父类处理
        }
```

Linux

> 万物皆数字,一切皆文件

Linux基础操作

Linux快捷键

```
ctrl+alt+t                  # 打开终端
ctrl+l                      # 清屏
ctrl+c                      # 在终端在退出锁定(终止程序)
Ctrl + Alt + F3             # 切换到字符界面
Ctrl + Alt + F1             # 切换到图形化界面
shell 命令快捷键
       history              # 历史命令/查看在shell敲过的命令
       tab                  # 自动补全
       ctrl + a             # 将光标移到最前面
       ctrl + e             # 将光标移到最后端
       ctrl + u             # 清空命令行
```

通配符

```
*                    # 匹配0-N个任意字符
?                    # 匹配1个任意字符
```

Shell命令

```
$ pwd                          # 查看当前路径 (pwd 打印工作目录 缩写)
$ date                       # 时间(运行/bin目录下的date可执行程序)
$ tree                       # 使用树状图显示当前目录结构(Windos)
$ reboot                       # 重启电脑
$ exit                       # 退出/注销
$ sudo cat a.txt               # 使用超级用户读文件
$ mv                           # 移动
$ ps                           # 查看进程信息PID:aux -a(显示终端上所有进程,包括其他用户进
程),
                                   # -u(显示进程的纤细状态)      -x(显示没有控制终端的进
程))
$ ps ajx                           # 查看所有进程组
$ ps -Lf                           # 查看线程号(LWP)

$ ls > out                   # 属性胡重定向(将输出内容输出到文本out里 没有文件会自动创建,  >>追
加)
    |                          # 管道:一个命令的输出可以通过管道作为另一个命令的输入
   $ ps aux | grep a           # 从系统运行进程中过滤出 a 进程相关的进程:两条及以上(有一条是
过滤器本身的进程)

$ top                          # 查看任务列表(Windows任务管理器)
$ kill                       # 终止进程: -SIGKILL(杀死进程)
$ pwd                          # 显示当前工作路径
$ which                      # 查看源命令存储得到位置
$ su                           # 切换用户:不加-不会切换工作目录, su- root(会切换工作目录
到/home下的家目录)
$ sudo                       # 临时获取一次root权限,执行完该命令后权限失效
$ adduser                      # 新建用户
$ deluser                      # 删除用户
$ chown                      # 修改文件所有者: chown 用户名 文件或目录名
$ stat filename              # 查看文件属性
$ cat /etc/passwd              # 查看用户
$ addgroup                   # 新建用户组
$ delgroup                   # 删除用户组
$ chgrp                      # 修改文件所属组: chgrp 组名 文件或目录名
$ cat /etc/group               # 查看用户组
$ sudo chown user:group c.txt    # 同时修改用户组 (sudo chown 用户名:组名 c.txt)
$ exit                       # 注销用户
$ ssh  userName@IP           # 使用Linux自带的ssh连接远程linux系统
   $ scp -r userName@IP:/home/src  /src/           # 拷贝下载文件
   $ scp -r ./src userName@IP:/home/src/           # 上传文件
```

编辑模式

```
i/a/o                        #向光标前面插入文本/向光标后面插入文本/向光标下一行插入文本
O/I/A                        #向光标上一行插入文本/向光标所在行的行首插入文本/向光标所在行的行
尾插文本
s/S                          #删除光标所在位置字符前提插入文本/删除光标所在行整行前提插入文本
```

命令模式

```
h, j, k, l                      # 左/下/上/右    光标移动
gg, G                           # 跳转至文件行首/跳转至文件末行
gg=G, nG                        # 排版代码/跳转至n行
0, $                            # 跳转至光标所在行的行首/跳转至光标所在行的行尾
Shift + z + z                   # 保存退出
u, ctal+r                       # 撤销上一次文本修改/反撤销
yy, n+yy, v+y                   # 复制单行/复制n行/区域型复制   :移动到多行首行的任意位置复制
dd, n+dd, v+x, dw               # 剪切/多行剪切/区域形剪切/剪切光标位置到单词结尾
p, P                            # 往光标所在行的下一行粘贴/ 往光标所在行的下一行粘贴
D, d$, d0, vd                   # 删除/从光标处删到行尾/从光标处删到行首/区域形删除
/text                           # 在文件中搜索文本内容
* , #                           # 将光标移到单词上,在命令行模式下使用 *()正序 #(倒序) 跳转
```

末行模式

```
:set nu(nonu)                 # 显示/不显示行号
:! +shell                     # 末行模式加可执行shell命令
:n                            # 跳转至n行
:w                            # 保存但不退出
:wq                           # 保存退出
:x                            # 保存退出
:q!                           # 不保存强制退出
:s/旧内容/新内容               # 替换单行内容中的内容中一个
    :s/旧内容/新内容/g        # 替换一行内容中的多个(所有)原内容
    :%s/旧内容/新内容/g       # 整个文档的内容替换
    :n,ns/旧内容/新内容/g     # n行-n行替换掉所有旧内容中内容
    : +ctrl +k(n)             # 向上(下) 翻使用过的命令

:sp, :vsp                     # 上下/左右分屏编辑
    ctrl +w +w                # 切换分屏
    :wqall                    # 保存所有分屏内容并退出
:e                            # 更新当前打开的文件内容
```

gcc

```
$ gcc -o test.c F             编译: -E(预处理源文件,不进行编译), -S(编译源文件,但是不进行汇编), -c(编译汇编源文件,但不进行链接), -o(将文件编译成可执行文件),|-g(在编译时生成调式信息,可以被调试器调试), -D(程序编译时指定宏) -w(不生成任何警告信息) -Wall(生成所有警告信息), -On(n四个级别0[没优化] 1[缺省值] 3[优化级别最高]), -l<L> (编译时指定库的名字,去掉lib和.a 剩下的就是名字), -L(指定编译搜索库的路径),      -std(指定编译器规范), -I <i>./src/ (指定include 头文件搜索目录)
```

文件管理

shell命令

```
$ ls                          # 查看目录 -l(查看目录下文件详细信息), -a(查看目录下所有文件), -i(查看文件inode), -d(查看目录本身的属性), -h(文件大小单位信息)
$ cd ./                     # 打开文件(如果只有cd则返回 /home下的用户目录, cd - 返回到上一次工作过的目录,)
$ file                      # 查看基本文件属性
$ mkdir                     # 创建空目录 mkdir 路径 -p(创建一个完整路径的文件)
```

```
$ rmdir                    # 删除空目录: -C 路径(指定解压缩在那个目录)
$ rm                       # 删除文件: -r(递归删除, 删除目录下所有文件目录再删除此目录), -
i(交互式删除询问是否删除)
$ touch                    # 在当前目录下创建一个空文件
$ cat                          # 打印文本内容
$ more                     # 文件内容分屏显示, q退出,无法使用光标,man手册功能键
$ less                     # 文件内容分屏显示, q退出,可以使用光标逐行下翻,man手册功能键
$ head 5                   # 从头开始显示几行
$ tac                          # 逆序显示文本内容
$ cp                           # 拷贝复制: -r(拷贝目录) -a (会拷贝属性 时间)
$ ln -s 源文件 链接文件          # 创建软链接(快捷方式)
$ chmod u/g/o/a/ +/-/= rwx       # 修改文件读写权限 :user/group/other/all rwx读写运行
权限
    $ chomd u-w,g+r,o+r c.txt        # 修改文件权限
    $ chomd u=rw,g=x,o=rwx c.txt     # 权限赋值,
    $ chomd a+x c.txt                # 所有人的权限修改
    $ chmod 721 c.txt                # 数字设定法r(4), w(2), x(1),  -rwx-w---x
r/w/x对于文件和目录的区别(读写运行权限)
    文件       # r(可以查看文件内容,cat head tail more less...), w(可以增加/修改/删除文
件内容,vim > >>), x(可以被运行, ./a.out)
    目录       # r(目录文件的内容可以被查看,ls tree), w(目录文件的内容可以被增删改, mkdir
touch rm mv)
#x(可以被进入, cd)
$ find /etc/ -                   # 按文件属性搜索文件/目录: -size +3k(按大小搜索), -
maxdepth 1(指定搜索层级,在其它参数之前指定该参数), -name "name"(按名称搜索) -type
f/d/e(按类型搜索)
    $ find ./ -maxdepth 1 -type f -exec ls -l {} \; # -exec(shell脚本 对搜索结果执行
某些命令)
    $ find ./ -maxdepth 1 -type f -exec rm -i {} \;
    $ find ./ -maxdepth 1 -type f | xargs ls -l     # 对搜索结果执行某些命令(不能搜索
名字带空格的)
    $ find ./ -maxdepth 1 -type f -print0 | xargs -0 ls -l
                                        # 对搜索结果的分隔符采用AIISC码的     null进行
分割,解决空格问题
$ grep -r "love" /etc -n                    # 按文件内容搜索文件
$ find -name "a" | xargs grep -n "love" # 搜索指定目录/文件下包含指定内容的文件
$ umask                                     # 查看文件掩码
$ stat                                  # 查看文件详细信息
```

Linux根目录结构

```
/                       # 根目录
/bin                    # 可执行二进制文件目录
/boot                   # Linxu系统启动文件
/dev                    # Linux设备文件,访问该目录下某文件相当于访问某设备
/etc                    # 系统配置文件存放目录(不建议放可执行文件)
/home                   # 家目录
/lib                    # 库目录
/lost+fount             # 系统异常遗失的片段(错误日志, 碎片)
/mnt                    # 光盘默认挂载点
/opt                    # 操作系统额外软件安装目录(音乐 QQ.....)
/proc                   # 存放系统核心 内存中的数据 ---临时存储
/root                   # 系统管理员 root 的家目录
/sbin                   # 管理员用户才能 用的可执行程序
/tmp                    # 一般用户或正在执行的程序临时存放文件的目录
/srv                    # 服务启动之后需要访问的数据目录
/usr                    # 用户应用程序存放目录了
```

```
/var                        # 放置系统执行过程中经常变化的文件，如临时日志
```

Linux下文件属性

> . 开头的文件或目录为隐藏文件/目录

```
- 普通文件                  # 占用磁盘存储
d 目录文件                  # 占用磁盘存储
l 链接文件                  # 占用磁盘存储 (类似于windows下快捷方式)
   链接文件
      ln 源文件 链接文件              # 硬链接文件占磁盘空间 但是删除源文件不会影响硬链接文件(复制拷贝)
      ln -s 源文件 链接文件           # 软链接文件不占磁盘空间 但是删除源文件会影响软链接文件(快捷方式)
      硬链接和拷贝（复制）区别:
                    # 无论修改了哪一个链接之后的文件 两个文件都会改变 保持一致  但是拷贝不会
                    # 改变软链接文件就是相当于间接的改变了源文件
                    # 查看文件时默认链接数为1 如果有链接一次递增
                    # 如果创建的软链接文件和源文件在不同的目录下，需要使用绝对路径
b 块设备文件          # 伪文件，不占用磁盘空间
c 字符设备文件        # 伪文件，不占用磁盘空间
p 管道文件            # 伪文件，不占用磁盘空间
s 套接字              # 伪文件，不占用磁盘空间
未知文件
文件读写权限
      rwx                  # 读写执行权限,没有权限用 - 占位
--help
   date --help             # 指定命令 查询帮助信息
    man手册
     1卷:                  # 命令的帮助信息
     2卷:                  # 系统调用的帮助信息
     3卷:                  # 库函数
    man手册功能键
     Enter                    # 一次滚动手册页的一行
     空格页                   # 显示手册页的下一屏
     b                        # 回滚一屏
     f                        # 前滚一屏
     q                        # 退出 man 命令
     h                        # 列出所有功能键
     /word                    # 搜索 word 字符
     n                        # 下一个
```

```
Linux压缩文件
      $ tar -zcvf 压缩包名.tar.gz 压缩源文件/目录-zxvf v      # 压缩
      $ tar -zxvf 压缩包名.tar.gz                            # 解压缩
      $ tar -jcvf 压缩包名.tar.bz2 压缩源文件/目录            # 压缩
      $ tar -jxvf 压缩包名.tar.bz2                           # 解压缩
```

库

库的工作原理

```
静态库工作原理          # gcc进行链接后,静态库的代码和测试文件的代码被打包到了可执行文件中
动态库工作原理          # gcc链接后,动态库不会和可执行程序打包到一起,程序启动之后动态库会被动态加载到内存中
```

静态库

> ***linux(`libxxx.a`), windows(`libxxx.lib`)***

```
生成静态库
    $ gcc -c 1.c 2.c 3.c 4.c              #将多个源文件生成若干个 .o的文件(1.o, 2.o, 3.o, 4.o)
    $ ar rce libxxx.a 1.o 2.o 3.o 4.o     #将多个 .o 文件进行打包成静态库: rce(-r: 替换, -c: 创建,  -s: 索引)
引入静态库
    $ gcc main.c -o app -I<i> ./include/ -L ./  -l xxx      # 编译时指定库文件 头文件路径
```

动态库/共享库

> ***linux(`libxxx.so`) windows(`libxxx.dll`)***

```
生成动态库
    $ gcc 1.c 2.c 3.c 4.c -c -fpic            #生成 .o文件:-fpic/-fPIC(使用相对地址记录代码位置)
    $ gcc -shared 1.o 2.o 3.o 4.o -o libxxx.so  # 生成动态库
 引入动态库
    $ gcc main.c -o cpp -I<i> ./include / -L ./ -xxx  # 将多个 .o 文件进行打包成动态库
    $ ldd xxx                                 # 查看可执行程序动态读取是否成功(找不到动态库,路径)
```

> $ 找不到动态库 libcalc.so => mot found
> 程序执行时定位库文件顺序:
> 搜索eft文件的 `DT_RPATH段` => 环境变量 `LD_LIBRARY_PATH` => `/etc/ld.so.cache`文件列表 => `/lib/`, `/usr/lib`目录

MakeFile

> 全自动化编译工具
>
> Makefile中的其他规则一般都是为第一条规则服务的

Makefile规则/语法

```
$ make          #执行Makefile里面的shell命令(也可查看当前目录下是否有执行MakeFile文件)

文件命名格式
    makefile
    Makefile

Makefile中的规则
```

```
    # Makefile中的其他规则一般都是为第一条规则服务的
    # 语法
        目标 ...: 依赖 ...    # 目标:最终要生成的文件(伪目标不算)  依赖:生成目标的源材料
            命令  ...         # 命令(shell命令):通过目标操作依赖生成目标,命令前面带"Tab缩进"
            ....
    # 一个Makefile文件中可以有一个或多个规则
    # 一个Makefile中有多个规则,对应有多个目标,执行make生成一个最终目标 == 第一条规则中的目标
        clean:
            rm $(object) $(target) -f  #清理工作,在shell命令中执行make时多带个clean参数可执行 -如果同一目录下有个clean文件,由于这个规则没有依赖检测更新规则比对时间时会一直以为这个Makefile生成的文件是最新的,就不会执行规则中的命令
        .PHONY:clean            #clean规则并不会生成新文件,所以可以把它声明成 伪目标,就不会对目标和依赖进行时间比对
        clean:
            rm $(object) $(target) -f
```

Makefile工作原理

```
1) 命令在执行前,先检测规则中的依赖是否存在
    if(存在)? 执行命令 : 向下检测其他规则(检测有没有其中一个规则是用来生成这个依赖的 下举例2);
2) 检测更新
    * 在执行规则中的命令时,比较  目标和所有依赖文件时间(如果依赖时间比目标时间晚,说明依赖被修改更新过)
        if(依赖时间比目标时间晚) ? 需重新生成目标 : 目标不需要更新,对应规则中的命令不需要执行
```

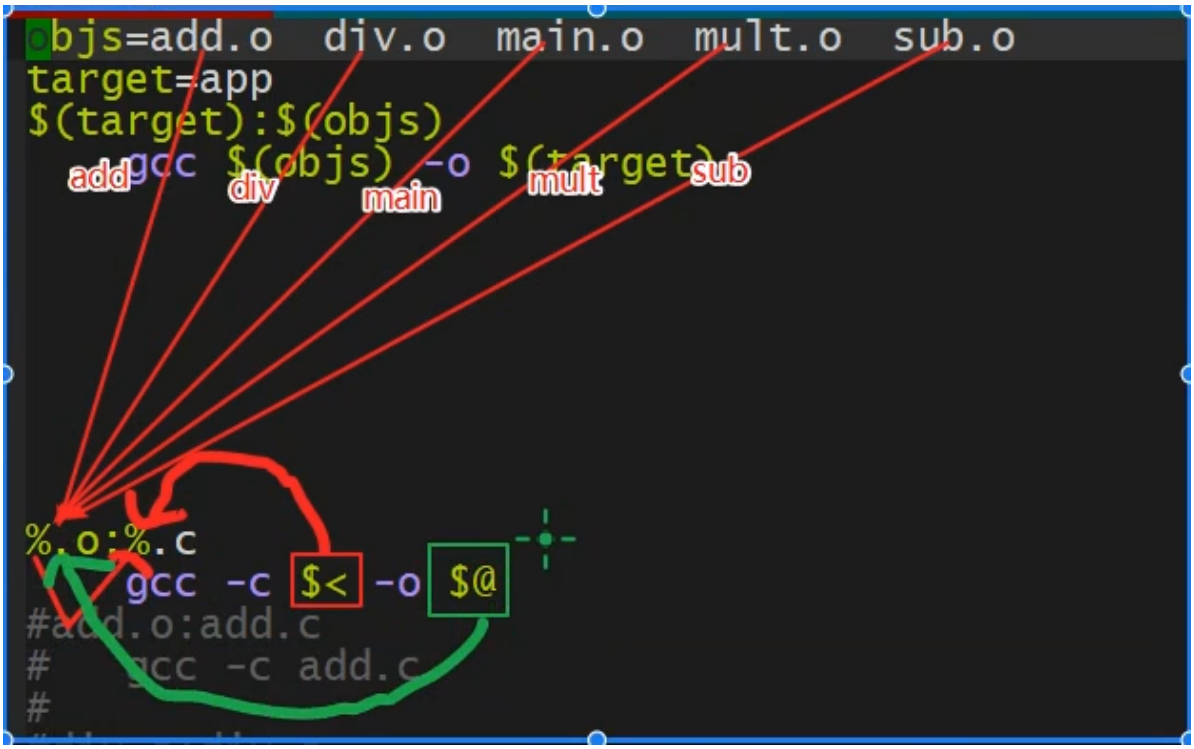
Makefile变量/函数

```
自定义变量
    变量名=变量值     #变量定义(没有数据类型,自定义变量一般小写)
    temp = hello,world
    取变量的值
        value = $(temp) # $取值(有返回值)
默认的自带变量
    自带变量大写
        CC 默认等于 cc       #在Linux下,cc等价于gcc
自动变量  #只能在规则的命令中使用
    app:main.c hello.c a.c b.c
        gcc -c main hello.c a.c b.c
    $(CC) -c $^      #可替代上一语句
    $@:规则中的目标                  -app
    $<:规则中的第一个依赖            -main.c
    $^:规则中的所有依赖               -main.c hello.c a.c b.c
    $(CC):gcc
```

```
函数
    # $(wildcard PATTERN...)  获取指定目录下指定类型的文件列表,多个目录用空格隔开
        src = $(wildcard *.c ./sub/*.c)      #  返回格式 a.c b.c c.c d.c
    # $(patsubst <pattern>, <replacement>, <text>)  模式字符串替换函数
        $(patsubst %.c, %.o, $(src))
```

Makefile模式匹配

```
Makefile模式匹配
    %  #通配符,匹配一个字符串
    %.c : %.c  #两个%匹配的时同一个字符串
```



S

Makefile编写

```
# 举例: 目录下有 add.c div.c head.c main.c mult.c sub.c 生成 app(可执行程序)
 1) #缺点:任意一个源文件被修改,所有的源文件都要重新编译
        $ vi Makefile                    # 创建makefile文件
    # 编辑Makefile
        1:app:add.c div.c head.c main.c mult.c sub.c  #建立依赖
        2:  gcc add.c div.c head.c main.c mult.c sub.c -o app  #shell命令编译生成
app文件,前面加Tab缩进
```

```
 2) #缺点:书写臃肿,不适用于文件较多
    # 检测第一条规则中的依赖不存在时,向下检测有没有其中一个规则是用来生成这个依赖的
        $ vi Makefile                    # 创建makefile文件
    # 编辑Makefile
        1:app:add.o div.o head.o main.o mult.o sub.o
        2:  gcc app:add.o div.o head.o main.o mult.o sub.o
        3:
        4:add.o:add.c
        5:  gcc -c add.c
        6:div.o:div.c
        7:  gcc -c div.c
        8:head.o:head.c
        9:  gcc -c head.c
        10:main.o:mian.c
        11: gcc -c main.c
        12:mult.o:mult.c
```

```
        13: gcc -c mult.c
        14:sub.o:sub.c
        15: gcc -c sub.c
        16:clean:
        17: rm
```

```
3)  #匹配模式    缺点:获取.o依赖文件需要手写,书写麻烦
        $ vi Makefile                    # 创建makefile文件
    # 编辑Makefile
        1:object = add.o div.o head.o main.o mult.o sub.o
        2:target = app
        3:$(target):$(object)
        4:  gcc $(object) -o $(target)
        5:
        6:%.o:%.c
        7:  gcc -c $< -o $@                 #编译本规则(第6行规则)中的第一个依赖,生成本
规则中的目标文件
```

```
4)
        $ vi Makefile                           # 创建makefile文件
    # 编辑Makefile
        1:src = $(wildcard ./*.c)               #获取当前目录下所有.c文件
        2:object = $(patsubst %.c, %.o, $(src))     #取变量src里面的值将.c替换成.o
        3:target = app
        4:$(target):$(object)           #设置第一天Makefile规则
        5:  gcc $(object) -o $(target)
        6:%.o:%.c
        7:  gcc -c $< -o $@             #编译本规则(第6行规则)中的第一个依赖,生成本规则
中的目标文件
```

```
5) # 比4多一个清理程序,生成app文件后清理掉所有的.o文件
        $ vi Makefile                           # 创建makefile文件
    # 编辑Makefile
        1:src = $(wildcard ./*.c)               #获取当前目录下所有.c文件
        2:object = $(patsubst %.c, %.o, $(src))     #取变量src里面的值将.c替换成.o
        3:target = app
        4:$(target):$(object)           #设置第一天Makefile规则
        5:  gcc $(object) -o $(target)
        6:%.o:%.c
        7:  gcc -c $< -o $@             #编译本规则(第6行规则)中的第一个依赖,生成本规则
中的目标文件
        8:.PHONY:clean              #将clean定义成伪目标
        9:clean:
        10: rm $(object) $(target) -f   #清理.c生成的.o文件的命令,在shell命令中执行
make时多带个clean参数可执行
```

GDB调试

> `gcc test.c -o test -g`在编译时需要加参数 `-g`,会在可执行程序中加入调试信息,这个程序就可调试

GDB调试命令

```
'gdb启动和退出'
# gdb启动    shell命令
    $ gdb app            #app 可执行程序必须是有可调式代码的程序
# 可调试程序设置传入参数
    (gdb):set args 10 20                #将10 20作为参数传入调试程序
    (gdb):show args                  #可查看传入的参数
        Argument list to give program being debugged when it is started is "10
20"  #返回查看的参数
# 退出gdb调试
    (gdb):quit                       #退出gdb调试,返回shell终端界面
```

```
'查看代码'
# 默认当前文件是main入口函数所在文件
`1) 当前文件
    #从默认行显示代码
        (gdb):l                      #查看当前文件value(代码段), 默认显示10行
        (gdb):list                   #查看当前文件value(代码段),默认显示10行,敲回车继续
显示代码
    #从指定行号开始显示代码段
        (gdb):l 10                   #第10行在中间位置,显示第十行前五行,后五行
    #从指定的函数开始显示
        (gdb):l main                 #从指定的函数名函数段开始查看,上下显示五行
`2) 非当前文件
    #指定文件指定行号显示
        (gdb):l test.cpp:10          #从指定行号(上下各五行)开始查看; l 文件名:行号
    #指定文件指定函数查看
        (gdb):l test.cpp:main        #从指定文件的指定函数段开始查看(上下各五行)
`3) 设置显示的行号
    #查看默认一次性能显示的行数
        (gdb):show list
        (gdb):show listsize          #等价于 show list
    #更改可显示函数
        (gdb):set list 20            #设置一次性可显示20行代码段
        (gdb):set listsize 20        #等价于(set list 20)
```

断点操作

```
"设置断点"
    #命令: break == b
    `1)在当前文件设置断点
        # b 行号
            (gdb):b 10               #在第十三行设置断点
            (system):Breakpoint 1 at 0x400c23: file test.cpp, line 10.    #系统回
复,断点已设置
        # b 函数名
            (gdb):b main             #在main函数入口设置断点
            (system):Breakpoint 2 at 0x400c38: file test.cpp, line 15.
    `2)在非当前(指定)文件设置断点
        # b 文件名:行号
            (gdb):b test.cpp:15      #指定文件下的指点行号设置断点
            (system):Breakpoint 3 at 0x400c16: file test.cpp, line 15.
        # b 文件名:函数名
            (gdb):b test.cpp:main    #指定文件下指定函数入口处设置断点
```

```
(system):Breakpoint 4 at 0x400c246: file test.cpp, line 15.
```

```
"查看断点"
    #命令: i == info
        (gdb): i b                  #可查看设置的所有断点
```

```
"删除断点"
    #命令: d == del == delete
    # d 断点的编号, 删除断点
        (gdb):d 10                  #(gdb): i b,可查看断点信息,Num
```

```
"设置断点无效"
    #命令: dis == disable
    # dis 断点的编号, 使断点静默
        (gdb):dis 1                 #使编号1的断点失效, (gdb): i b,可查看断点信息
```

```
"无效断点生效"
    #命令: ena == enable
    # ena 断点的编号
        (gdb):ena 1                 #使失效的某个断点重新生效, (gdb): i b,可查看断点信
息
```

```
"设置条件断点"
    #一般用在循环里面,
    #b 行号 if 变量==值
        (gdb):b 18 if i==1   #(b-设置断点, 18-行号, if-判断, i==0-判断条件)
```

调试命令

```
'gdb操作变量'
    # 打印变量的值
    # 命令: p==print
        (gdb):p a
        (gdb):p/x a          # 以16进制打印变量a的值
    # 打印变量的数据类型
        (gdb):ptype a
    # 变量值的自动显示
        # 一直跟踪打印变量a的值
            (gdb):display a
        # i==info, 查看自动打印值的所有变量和信息
            (gdb):i dislay       # 返回自动跟踪变量的编号.是否生效
        # 取消自动跟踪打印变量值
            (gdb):undisplay num      # 使用i display查看编号,输入编号取消
    # 设置变量值(一般用于循环)
        (gdb):set var a=5            # 设置变量a=5 (set-设置, var-类型(类型变量))
```

```
"运行gdb程序"
    # 通过start, 只运行一行就停了
        (gdb):start
    # 通过run, 一直运行直到遇到断点
        (gdb):run
    # 在断点处停下,从断点的位置继续运行,停在下一个断点
    #命令: c==continue
```

```
"单步调试"
    # s==setp,向下执行一行,遇到函数体会进入函数体
        (gdb):s
    #跳出函数体,要跳出的函数体里面不能有断点(有的话设置为无效)
        (gdb):finish
    # n==next,向下执行一行,遇到函数体不进入函数体
        (gdb):n
    # 跳出循环,要跳出的循环体里面不能有断点,有的话设置为无效否则跳不出去
    #执行完循环体的最后一行(循环开始的开始的第一行),才能跳出
        (gbd):unitl
```

Linux文件I/O

> 标准C函数有缓冲区.Linux系统I/O不带缓冲区,磁盘读写的时候需要缓冲区提升效率,用于网络通信不需要缓冲区

Linux File结构体

```
//Linxu c FILE结构体定义: /usr/include/libio.h
struct _IO_FILE {
  int _flags;           /* High-order word is _IO_MAGIC; rest is flags. */
#define _IO_file_flags _flags
  /* The following pointers correspond to the C++ streambuf protocol. */
  /* Note:  Tk uses the _IO_read_ptr and _IO_read_end fields directly. */
  char* _IO_read_ptr;   /* Current read pointer */
  char* _IO_read_end;   /* End of get area. */
  char* _IO_read_base;  /* Start of putback+get area. */
  char* _IO_write_base; /* Start of put area. */
  char* _IO_write_ptr;  /* Current put pointer. */
  char* _IO_write_end;  /* End of put area. */
  char* _IO_buf_base;   /* Start of reserve area. */    //缓冲区的起始位置
  char* _IO_buf_end;    /* End of reserve area. */      //缓冲区的结束位置
  /* The following fields are used to support backing up and undo. */
  char *_IO_save_base; /* Pointer to start of non-current get area. */
  char *_IO_backup_base;  /* Pointer to first valid character of backup area */
  char *_IO_save_end; /* Pointer to end of non-current get area. */
  struct _IO_marker *_markers;
  struct _IO_FILE *_chain;
  int _fileno;
#if 0
  int _blksize;
#else
  int _flags2;
#endif
  _IO_off_t _old_offset; /* This used to be _offset but it's too small.  */
#define __HAVE_COLUMN /* temporary */
  /* 1+column number of pbase(); 0 is unknown. */
  unsigned short _cur_column;
  signed char _vtable_offset;
  char _shortbuf[1];
  /*  char* _save_gptr;  char* _save_egptr; */
  _IO_lock_t *_lock;
#ifdef _IO_USE_OLD_IO_FILE
    }   //结构体中
```

虚拟地址空间文件描述符

> 虚拟地址空间是不存在的,的虚拟地址空间地址会被映射到物理内存中
>
> 文件描述符表默认大小: `1024`, 每次进程启动之后,都有一个文件描述符表
>
> `文件描述表是一个 int 类型数组, 0~1023`
>
> 所以`每个进程默认能打开的文件个数: 1024`
>
> 文件描述表的前三个文件描述符默认被使用了的, ----`标准输入[0]`, `标准输出[1]`, `标准错误[2]`
>
> 文件描述符如果被占用了,需要使用其他未被占用(选数组排序最小的)

LinuxAPI

```
'errno --> 属于Linux系统函数库里面的一个全局变量,记录的是一个错误号'
    #include <stdio.h>
    /* perror打印的是errno对应的错误描述 */
    void perror(const char* s);     /* -s:用户描述 */
        perror("hello");
        // 例: -s为"hello",实际输出--> hello:xxxxxxx(实际的错误描述)
```

Linux 文件 I/O

```
/* open 打开文件*/
    #include <sys/types.h>
    #include <sys/stat.h>
    #include <fcntl.h>

    "打开已经存在的文件"
    int open(const char* pathname, int flags);
        int fp = open("./c.txt", O_RDWR);
        /* 参数:
            - pathname: 要打开的文件路径,
            - flags: 对文件的操作权限设置(互斥,只能选一个)
                0->O_RDONLY->读
                1->O_WRONLY->写
                2->O_RDWR->读写
        return: 成功 ? 文件描述符 : -1
        */

    "使用 open 函数创建一个新文件"
    int open(const char* pathname, int flags, mode_t mode);
        int fp = open('./c.txt', O_RDWR|O_CREAT, 0777);
        /* 参数
            - pathname: 要打开的文件路径,
            - flags: 对文件的操作权限设置
                必选项:
                    0->O_RDONLY ->读
                    1->O_WRONLY ->写
                    2->O_RDWR   ->读写
                    3->O_APPEND ->追加写入
                可选项: O_CREAT->文件不存在,创建新文件
```

```
            - mode: 八进制的数,表示用户对新创建的文件的操作权限->0775(0表示为八进制)
<mode & ~umask>
                按位与计算: 0和任何数都为0
                按位或: 1和任何数都为1
                ~umask: 文件掩码(shell终端命令:umask查看,可设置)
          返回值:
            - 成功：文件描述符
            - 失败： -1
        */
```

```
/* close 关闭文件*/
    #include <unistd.h>
    int close(int fp);
        close(fp);

/* read 读取文件*/
    #include <unistd.h>
    ssize_t read(int fp, void* buf, size_t count);
        /* 参数:
            - fp: open得到的文件描述符,通过fp操作文件
            - buf: 缓冲区,存储读到的数据,传入数组的地址
            - count: buf的大小
        返回值:
            - 成功: 返回实际读到的字节数
            - 失败：==  0:代表这个文件已经读完了 返回-1,errno会被设置
        */

/* write 写入文件*/
    #include <unistd.h>
    ssize_t write(int fp, const void* buf, size_t count);
    /* 参数:
        - fp: open得到的文件描述符,通过fp操作文件
        - buf: 要往磁盘写入到数据 // STDOUT_FILENO
        - count: 要写的数据的实际大小
    返回值:
        成功：返回实际写入的字节数
        失败：返回-1,errno会被设置
    */
```

```
/* fseek */
     // yodon
    #include <sys/types.h>
    #include <unistd.h>

        // int fseek(FILE* stream, long offset, int whence)
        // C语言(获取当前文件指针的位置, 移动文件指针到文件头,获取文件长度)
    off_t lseek(int fp, off_t offset, int whence);
        lseek(fp, 0, SEEK_SET);     // 移动文件指针到文件头
        lseek(fp, 0, SEEK_CUR);     // 获取当前文件指针的位置
        lseek(fp, 0, SEEK_END);     // 获取文件长度
        lseek(fp, 100, SEEK_END);   // 扩展文件长度,当前10B,扩展为110B,增加的字节填充为
0
            write(fp, " ", 1);          // 扩展完要进行一次写操作
        /* 参数:
            - fp: open得到的文件描述符,通过fp操作文件
            - offset: 偏移量
```

```
            - whence:
                SEEK_SET: 设置文件指针的偏移量
                SEEK_CUR: 设置偏移量(当前位置 + 第二个参数偏移量)
                SEEK_END: 设置偏移量(文件大小 + )第二个参数偏移量
        */
```

```
'stat/lstat函数'
    #include <sys/types.h>
    #include <sys/stat.h>
    #include <unistd.h>

    struct stat{
    dev_t                   st_dev;         // 文件的设备编号
    ino_t                   st_ino;         // 节点
    mode_t                  st_mode;        // 文件的类型和存储权限
    nlink_t                 st_nlink;       // 转到该文件的硬连接熟路,刚建立的文件值为1
    uid_t                   st_uid;         // 用户id
    gid_t                   st_gid;         // 组id
    dev_t                   st_rdev;        // (设备类型)若此文件为设备文件,则为其设备
编号
    off_t                   st_size;        // 文件字节数(文件大小)
    blksize_t               st_blksize;     // 块大小(文件系统的I/O 缓冲区大小)
    blkcnt_t                st_blocks;      // 块数
    time_t                  st_atime;       // 最后一次访问时间
    time_t                  st_mtime;       // 最后一次修改时间
    time_t                  st_ctime;       // 最后一次改变时间(指属性)
};
    // 如果文件是链接文件,则会获取软链接指向的文件的属性
    int stat(const char* pathname, struct stat* buf);   // 查看文件属性
    int lstat(const char* pathname, struct stat* buf);// 查看软链接文件(快捷方式)的属
性
```

```
'文件属性操作'
#include <unistd.h>
    int access(const char* pathname, int mode); /* 判断文件权限,或者判断文件是否存在
*/
    int chmod(const char* filename, int mode);  /* 修改文件权限 */
    int chown(const char* path, uid_t owner, gid_t group);  /* 修改文件所有者 */
    int truncate(const char* path, off_t length);/* 修改文件大小 */
'目录属性操作'
    int rename(const char* oldpath, const char* newpath);   /* 文件的重命名 */
    int chdir(const char* path);        /* 修改进程的工作目录 */
    int mkdir(const char* pathname, mode_t mode);       /* 创建一个目录 */
    int rmdir(const char* pathname);                /* 删除一个目录,只能删除空目录*/
    char* getcwd(char* buf, size_t size);  /* 查看当前进程的工作目录 [ $ pwd ]*/
        /*参数:
            - buf: 得到的路径,指向一个有内存大小的数组
            - size: 修饰buf缓冲区大小
        返回值: 指向buf内存的指针【和buf指向同一块地址】 */
```

```
"目录遍历函数"
        struct dirent{
```

```
        ino_t d_ino;                // 此目录进入点的inode
        ff_t d_off;                 // 目录文件开头至此目录进入点的偏移量
        signed short int d_reclen;  // d_name 的长度.不包括NULL字符
        unsigned char d_type;       // d_name 所指的文件类型
        har d_name[256];            // 文件名
    };
    d_type
        DT_BLK      // - 块设备
        DT_CHR      // - 字符设备
        DT_DIR      // - 目录
        DT_LNK      // - 软链接
        DT_FIFO     // - 管道
        DT_REG      // - 普通文件
        DT_SOCK     // - 套接字
        DT_UNKNOWN  // - 未知

    DIR* opendir(const char* name); /*打开一个文件目录*/
    struct dirent* readdir(DIR* dirp);  /* 读取一个目录文件的属性*/
    int closedir(DIR* dirp);        /* 关闭目录文件*/
    /* 实例代码
        DIR* dirfp = opendir("./day/");
        readdir(dirfp);
        closedir(dirfp)
    */
    #include <dirent.h>
    int scandir(const char* dir, struct dirent*** namelist, int (*filter)(const
void* b),
            int (*compare)(const struct dirent**, const struct dirent**) );
    // 遍历一个目录
    /* 参数:
        - dir: 要操作的目录
        - namelist: 传出参数,指向dirent指针数组,传个&struct dirent**的地址去接收它,
                        会给这个二级指针分配内存,不使用了需要释放
        - filter: 过滤回调函数,根据函数中指定的条件筛选文件名
        - compare: 对查找的文件名排序,可以直接使用默认的排序函数alphassort()和
versionsort()
    返回值: 读到的当前目录下的文件个数*/
    int alphassort(const void* a, const void* b);  // 一般默认用这个
    int versionsort(const void* a, const void* b);
```

```
'dup/dup2'
    int dup(init oldfp); // 复制文件描述符
    int dup2(int fd1, int fd2);  // 文件描述符的重定向(fd2重定向指向fd1)

"fcntl函数"
    #Include <stdio.h>
    #include <fcntl.h>
    /* 使用fcntl复制文件描述符 == dup */
    int funtl(int fd, int cmd, .../* arg */);
        int ret = fcntl(fd, F_DUPFD);
        /* 参数:
            - F_DUPFD: 复制文件描述符(复制第一个参数fd)
            - 不能修改O_RDONLY, O_WRONLY, or O_RDWR. 可修改可选项
                - O_APPEND: 数据追加
                - O_NONBLOCK: 设置非阻塞
        */
```

```
'获取系统时间'
#include <time.h>
    time_t time(time_t* tloc);          /* 返回以长整型,从1970-01-01到现在的秒数 */
        time_t tm = time(NULL);
    struct tm{
        int tm_sec;
        int tm_min;
        int tm_hour;
        int tm_mday;
        int tm_mon;
        int tm_year;
        int tm_wday;
        int tm_yday;
        int tm_isdst;
    };

    struct tm* localtime(const time_t* timep);          /*  时间转换成一个结构体 */
        struct tm* loc = localtime(&tm);

    char* asctime(const struct tm* tm);             /* 格式化时间字符串 */
        char* curtim = asctime(loc)
```

Linux进程

> 进程的5中状态:`创建`,`就绪`,`运行`,`阻塞`,`退出`
>
> 并行:多个人在某一时间点干同一件事
>
> 并发:描述的是在一个非常短的时间段内处理若干个工作

查看进程

```
"命令查看进程"
    $ ps aux/ajx
        # 参数:
        # - a: 显示当前终端下所以的程序(所有用户)
        # - u: 显示用户信息
        # - x: 打印有关的终端信息
        # - j: 显示更多详细的用户信息
```

```
'查看进程函数'
    #include <sys/types.h>
    #include <unistd.h>

    pid_t getpid(void);     // 当前进程的id
    pid_t getppid(void);    // 当前进程的父进程的id
```

杀死进程

```
# 查看信号
$ kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL       5) SIGTRAP
 6) SIGABRT      7) SIGBUS       8) SIGFPE       9) SIGKILL      10) SIGUSR1
11) SIGSEGV      12) SIGUSR2     13) SIGPIPE     14) SIGALRM     15) SIGTERM
16) SIGSTKFLT    17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN      22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM    27) SIGPROF     28) SIGWINCH    29) SIGIO       30) SIGPWR
31) SIGSYS       34) SIGRTMIN    35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3
38) SIGRTMIN+4   39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9   44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14  49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11  54) SIGRTMAX-10 55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7
58) SIGRTMAX-6   59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1   64) SIGRTMAX

# 杀死进程
    $ kill -9 PID      # $: kill -SIGKILL PID
```

创建子进程

> 新创建的子进程会复制父进程的数据,如果父子进程都不修改数据则共用内存,如果谁修改数据谁就会重新寻找内存存储数据
>
> `子进程会继承父进程中设置的 阻塞信号集、信号处理动作[回调函数]`

```
#include <unistd.h>

//在当前进程中创建一个子进程
    pid_t fork(void);
        pid_t ret_pid = fork();
    /* 返回值:
        - >0: 代表父进程的返回值
        - ==0: 子进程的返回值       */
```

```
'多进程的gdb调试'
    gcc fork.c -g
    # 默认gdb跟踪父进程
        # 设置跟踪父进程
            set follow-fork-mode parent
        # 设置跟踪子进程
            set follow-fork-mode child
```

exec函数族

```
'exec函数族'
    '会替换进程中的数据,exec函数族一般在子进程中运行，内核数据可能不会被修改'
    #include <unistd.h>
    extern char** environ;

    // 不是Linux系统函数

    /* 常用,适用于自己写的程序,没有环境变量配置 */
```

```
    int execl(const char* path, const char* arg, ...);
        execl("/home/test/a.out", "xxx", "hell". "123", NULL);
        execl("bin/ps", "ps", "aux", NULL);      //终端执行程序时启动 ps aux操作
        /*参数
            - path: 可执行程序的路径,建议绝对路径
            - arg: 第二个参数arg,无用参数(随便写,一般为了看起来舒服,写成和参数1 相同的值)
            - 第三个参数开始: 可执行程序执行过程中需要的真正参数
            - 最后一个参数: NULL(结束转义符类似于\0)    */
    /* 常用,适用于系统自带,有环境变量*/
    int execlp(const char* file, const char* arg, ...);
        execp("ps", "xxx", "aux", NULL);
                /* 参数
            - file: 可执行程序的名字,在执行程序前会自动搜索系统环境变量PATH
            - arg: 第二个参数arg,无用参数(随便写,一般为了看起来舒服,写成和参数1 相同的值)
            - 第三个参数开始: 可执行程序执行过程中需要的真正参数
            - 最后一个参数: NULL(结束转义符类似于\0)      */

    int execle(const char* path, const char* arg, ..., char* const envp[]);
                /* 参数
            - path: 可执行程序的名字
            - arg: 第二个参数arg,无用参数(随便写,一般为了看起来舒服,写成和参数1 相同的值)
            - 第三个参数开始: 可执行程序执行过程中需要的真正参数
            - envp[]: 从这个参数指定的路径所属第一个参数对应的文件名    */
            char* encp[] = {"/home/test", "/a/n", "/bin/", NULL};
    int execv(const char* path, char* const argv[]);
        char* argv[] = {"ps", "aux", NULL};
        exev(ps, argv);
        /* 参数
            - path: 可执行程序的名字
            - argv[]: 参数列表     */
    int execvp(const char* file, char* const argv[]);

    // Linux系统函数
    int execve(const char* path, char* const argv[], char* const envp[]);
```

> 每个子进程结束后,都会自己释放自己地址空间中的用户区的数据.内核区的PCB块需要父进程手动释放

结束进程

```
'结束进程'
    //  --->封装的库函数
        #include <stdio.h>
        void exit(int status);
            exit(-1);   // 类似于return, -1返回的时子进程的状态
    // Linux系统函数
        #include <unistd.h>
        void _exit(int status);
```

子进程回收

```
'进程回收'
    #include <sys/types.h>
    #include <sys/wait.h>
```

```
    // 父进程中有子进程存活的情况下阻塞,当子进程死掉时回收进程(一次只能回收一个),子进程死完了阻塞取消
    pid_t wait(int* status);    // 当前父进程是否还有子进程 ? 如果有,调用该函数会默认阻塞 : 没有,不阻塞
        /* 参数:
            - status: 记录了子进程退出时候的状态(类似return函数)
            - return:
                >0: 被回收的子进程ID
                ==-1: 调用失败          */

            pid_t ret = wait(NULL);
            int s;
            int ret = wait(&S);
            if(WIFEXITED(s)) {printf("子进程退出的状态码:%d\n", WEXITSTATUS(s)); }
                                    // 获取子进程退出的状态码,返回return值
            if(WIFSIGNALED(s)) {printf("干掉子进程的信号:%d\n", WTERMSIG(s)); }


    // 在wait基础上扩展,回收指定的子进程,可以设置函数的阻塞和非阻塞状态
    pid_t waitpid(pid_t pid, int* status, int options);
        /* 参数:
            - pid:
                >0: 某个子进程的pid
                =0: 回收当前进程组的子进程
                -1: 回收所有子进程 == wait(NULL)
                <0: 某个进程组的组ID(回收指定组里面的属于子进程)
            - status: NULL
            - options: 设置函数阻塞或非阻塞状态
                - 0: 阻塞
                - WNOHANG: 非阻塞
            return :
                - >0: 回收的子进程的ID
                - ==0: options == WNOHANG,还有子进程存在[死了的子进程回收成功]
                - -1: 所有的子进程已经回收了,所以回收错误          */
```

进程间的通信

> 进程间通信: `管道(匿名管道, 有名管道)`, `内存映射`, `本地套接字`, `网络套接字`, `消息队列`, `共享内存`
>
> 父子进程始终共享什么东西？ `文件描述符`, `内存映射区`

管道

> 管道的本质: `内核缓冲区`, 拥有文件的特质(I/O)可以按照对文件的处理出路管道
>
> 匿名管道(没有文件的实体) 有名管道(有文件实体,不储存数据)
>
> 管道默认为`阻塞`状态, `fcntl`可以设置管道为非阻塞

匿名管道

> `匿名管道`:没有名字,在磁盘上没有实体,是内存中的一块缓冲区,只能实现有血缘关系的进程间的通信. 环形队列数据只能被读取一次. 内存缓冲区有读端和写端两部分. 父进程被销毁,管道自动被释放.

```
'创建匿名管道'
```

```
#include <unistd.h>
int pipe(int pipefd[2]);
    /* 参数
        - pipefd: 传出参数
            - pipefd[0]: 管道的读端(描述符)
            - pipefd[1]: 管道的写端(描述符)
        return : 成功? 0 : -1
    - 重定向: STDOUT_FILENO stdou(参数默认绑定指向终端) //管道输出文件描述符,默认指向终端
*/
        int fd[2];
        int ret_pipe = pipe(fd);
        if(ret_pipe == -1){ perror("匿名管道创建失败");  exit(0); }

'设置匿名管道非阻塞状态'
    // 使用 fcntl函数
        // 将读端设置为非阻塞状态
        int flag = fcntl(fd[0], F_GETFL);  // 获取读端是否阻塞状态, fd[0] 读端
        flag |= NONBLOCK;
        fcntl(fd[0], F_SETFL, flag);
```

有名管道

> 有名管道:磁盘上有一个伪文件(size: 0),通过伪文件映射内核缓冲区[环形队列,数据只能被读一次]
> 默认为阻塞状态,实现没有血缘关系的进程间通信
>
> 当读写端在两个进程中分离(一个进程是读端,一个进程写端),
>
> 其中一端被关闭时,另一端会自动解除阻塞(读写同时打开时不会)

```
'命令行创建管道文件'
    $ mkfifo test       # 在shell命令行创建管道文件
```

```
'代码函数创建管道'
    #include <sys/types.h>
    #include <sys/stat.h>
    int mkfifo(const char* pathname, mode_t mode);
        /* 参数:
            - pathname: 要创建的管道文件的路径和名字
            - mode: 用户对管道文件的操作权限,八进制的数->0775(o表示为八进制)  <mode &
~umask>
        */
```

文件/内存映射

> 将磁盘文件的数据映射到内存,内存和磁盘文件实时同步, 用户通过修改内存就能修改磁盘文件.

```
"函数原型"
    #include <sys/mman.h>

    //映射得到的内存 在进程用户区共享库加载的区域(虚拟地址空间)
    void* mmap(void* addr, size_t length, int prot, int flags, int fd, off_t
oddset);
        /* 参数
            - addr: 内存地址(传 NULL, 默认由内核申请指定地址)
            - length: 要映射的数据长度(不能写0)
            - prot: 用户对内存映射区的操作权限(读操作是必须拥有的权限)
```

```
                PROT_WRITE: 写    PROT_READ: 读     PROT_WRITE|PROT_READ: 读写
PROT_EXEC: 执行
            - flags:
                MAP_SHARED: 内存映射区会自动与磁盘文件同步(进程间通信必须设置选项)
                MAP_PRIVATE: 不自动同步(不能同步就不能实现进程通信)
            - fd: 文件描述符 (通过open打开一个磁盘文件得到的,文件大小不性能为0)
                open文件时需指定open的flags权限要与mmap指定的 prot权限一致
            - offset: 偏移量 (设置开始读取文件内容位置,只能偏移4k整数倍),0是不偏移
            return:  成功 ?  指向映射区起始位置的指针 : MAP_FAILED ((void *) -1)
*/
        int fd = open("./src", O_RDWD);
        int size_File = lseek(fd, 0, SEEK_END);
        void* ptr = mmap(NULL, size_File, PROT_WRITE|PROT_READ, MAP_SHARED, fd,
0);
        if(ptr == MAP_FAILED){ perror("内存映射失败"); exit(-1); }


'释放内存映射区'
    int munmap(void* addr, size_t length);
        /* 参数
            - addr: 指向内存映射区的指针 (mmap的返回值)
            - length: 申请内存时的长度一致 (mmap的第二个参数)
            return:  成功 ?  0 : -1       */
```

```
'通过内存映射的进程间通信'
    /* 有血缘关系
        在还没有子进程的时候,通过唯一的父进程先创建内存映射区,内存映射区有了之后再创建子进程.
        父子进程共享创建出来的内存映射区          */

    /* 没有血缘关系的进程间通信
        准备一个大小非0的磁盘文件,进程1,2通过这个磁盘分别创建内存映射区,两个进程得到操作对应内
存映射区的指针,
        通过这个磁盘文件实现两进程间内存区域数据的同步 (和管道通信的区别是没有阻塞)
*/
```

信号

信号的定义

```
# 查看信号
    $ kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL       5) SIGTRAP
 6) SIGABRT      7) SIGBUS       8) SIGFPE       9) SIGKILL      10) SIGUSR1
11) SIGSEGV     12) SIGUSR2     13) SIGPIPE     14) SIGALRM     15) SIGTERM
16) SIGSTKFLT   17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO       30) SIGPWR
31) SIGSYS      34) SIGRTMIN    35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3
38) SIGRTMIN+4  39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7
58) SIGRTMAX-6  59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX

# 查看信号详细信息
```

```
    $ man 7 signal

    2) SIGINT   # 终止进程[Ctrl + c],可阻塞
    2) SIGQUIT  # 终止信号[Ctrl+\],可阻塞
    9) SIGKILL   # 杀死信号,不可阻塞
```

```
# Linux中信号的行为  -->进程收到这个信号之后的处理动作
    `Term`      # 终止进程
    `Ign`       # 当前进程忽略掉这个信号
    `Core`      # 终止进程,并生成一个core文件(,一般进程发生某种错误时,core文件时gdb调式时使用)
        # 默认情况下,操作系统不允许生成core文件
        $ ulimit -1 可查看
            core file size (blocks , -c) 0    # -->文件大小为0,就是不允许生成文件
        # 放开系统限制
            ulimit -c size(int)/unlimited       # size指定文件大小, unlimited无限制
    `Stop`          # 暂停当前进程
    `Cont`          # 解除暂停,继续执行当前进程
```

```
# 信号的集中状态
    1. 产生       2. 未决(没有被处理)        3. 递达(被处理了)
        # 信号阻塞时是未决的一种特殊状态
```

```
# 进程收到信号后对信号的处理
    1. 执行信号的默认处理动作       2. 忽略(信号的默认动作不是忽略)       3. 执行用户自定义操作
        # 2.3需要对信号进行捕捉
    - 9)SIGKILL, 19)SIGSTOP  不允许被捕捉.忽略和阻塞
```

信号相关函数

```
'kill'
    #include <sys/types.h>
    #include <signal.h>
    //通过这个函数给某个进程发信号
    int kill(pid_t pid, int sig);
        /* 参数
            - pid:
                >0: 将信号发送给指定的进程 -->最常用
                =0: 将信号发送给当前进程的进程组
                =1: 将信号发送给所有有权限接受这个信号的进程
                <1: 这个pid=某个进程组的ID(123)取反(-123),信号发送给了ID为123的进程组中的每一个进程
            - sig: 要发送的信号(宏值/整数)
        */

'raise'
    #include <signal.h>
    // 当前进程自己给自己发送信号
    int raise(int sig);
        /* 参数:
            -sig : 要发送的信号
        返回值:
            - 0:  成功
            != 0: 失败
```

```
        */
        kill(getpid(),1); /*可以实现 raise功能 */

'abort'
    #include <stdio.h>
    //发送 SIGABRT 给当前进程自己，终止自己并产生一个core文件(类似于 exit()退出当前进程)
    void abort(void);
        abort();
```

```
'alarm'
    #include <unistd.h>
    //计时器，函数调用开始倒计时，当计时为0时会给函数会给当前进程发送一个SIGALRM信号
    // SIGALRM-> 默认终止进程
    unsigned int alarm(unsigned int seconds);
        /* 参数:
            - seconds: 倒计时时长，单位秒。如果参数为0，计时器无效
        返回值: 倒计时剩余的时间
        */
        alarm(100);   /* -->该函数不阻塞*/
        alarm(0);       /* 取消倒计时*/

        /* 实际时间 = 内核时间 + 用户时间 + 消耗的的时间 */
```

```
'setitimer'
    #include <sys/time.h>
    //可以实现周期性定时
    struct itimerval{
        struct timeval it_value;        //第一次触发定时器的时长，倒计时的时长
        struct timeval it_interval;     //定时器第二次以后每隔多长时间会被触发一次(频率)
    };
    struct timeval{     /* 总时间 = tv_sec + tv_usec */
        time_t  tv_sec;     //秒
        suseconds_t tv_usec;    //微秒,如果不使用，初始化为0
    };

    int setitimer(int which, const struct itimerval* new_val, struct itimerval*
old_value);
    /* 实现周期性定时 */
    /*参数:
        - whioh: 定时器以什么时间计时
            ITIMER_REAL: 真实的时间，时间到达发送SIGALRM信号(结束进程) -> 常用
            ITIMER_VIRTUAL: 用户时间，时间到达发送SIGVTALRM信号
            ITIMER_PROF: 内核时间    ，时间到达发送SIGPROF信号
        - new_val: 设置定时器属性
        - old_value: 记录了上一次定时器设置的属性，一般不使用，设置为NULL   */

{

        /* 当计时器触发时会发送14号信号,会杀死当前进程,所以要实现重复要捕捉14号信号 */
        struct sigaction act;
        act.sa_flags=0;
        act.sa_handler=function;
        sigemptyset(&act.sa_mask);
        sigaction(SIGALRM, &act, NULL);
        struct itimerval time_newItme;
        //设置闹钟(第一次定时器触发属性)
```

```
        time_newItme.it_value.tv_sec = 10;
        time_newItme.it_value.tv_usec = 0;
        //设置定时器重复的频率
        time_newItme.it_interval.tv_sec = 20;
        time_newItme.it_interval.tv_usec = 0;
        setitimer(ITIMER_REAL, &time_newItme, NULL);  //该函数并不阻塞

}
```

SIGCHLD信号

> SIGCHLD产生的条件: `1、子进程死了[自杀,他杀]` `2、子进程暂停了` `3、子进程由暂停状态重新恢复运行`
>
> 产生的信号发送给父进程，父进程会默认忽略此信号

进程间信号通信

```
"sigqueue"
    union sigval{
        /* 参数二选一[传出一个整形数 或者 内存的首地址]*/
        int sival_int;              /* 两个没有血缘关系的进程间通信 */
        void* sival_ptr;            /* 有血缘关系间的进程间通信*/
    };
    int sigqueue(pid_t pid, int sig, const union sigval vaule);
    /* 将信号发送给某个参数,并传递一些额外的数据 */
```

信号捕捉

```
'signal'
/* 不遵循POSIX规范,在某些特殊Linux环境下表现出来的行为不同 */

    #include <signal.h>
    typedef void(*sighandler_t)(int);   /*回调函数,int型参数就是捕捉到的信号 */
    sighandler_t signal(int signum, sighandler_t handler);
        /* 参数:
            -signum: 要捕捉的信号
            - handler: 回调函数，当信号被捕捉后，调用回调函数
        */
        signal(2, func);

'sigaction'
/* sigaction()遵循POSIX可移植性规范,所有Linux遵循*/

    struct sigaction{
        void (*sa_handler)(int);    /* 函数指针，指向信号被捕捉后的处理函数[默认调用函数] */
        void (*sa_sigaction)(int, siginfo_t*, void*);
                        /* 信号被捕捉后的处理函数[不常用, sa_handler互斥],用于进程间通信 */
        sigset_t sa_mask;
        /* 临时阻塞信号集[在信号捕捉函数执行过程中,临时阻塞某些信号],sa_mask=0不阻塞任何信号 */
        int sa_flags;           /* 设置信号捕捉后,调用的处理函数的函数指针[默认调用sa_handler] */
                /* sa_flags=0[默认] 调用sa_handler, sa_flags=SA_SIGINFO 调用sa_sigaction  */
```

```
        void (*sa_restorer)(void);  /* 被废弃的函数指针,为兼容旧版本,不使用 */
    };

    int sigaction(int signum, const struct sigaction* act, struct sigaction*
oldact);
        /* 信号捕捉,会阻塞信号 */
        /* 参数
            - signum: 要捕捉的信号
            - act: 捕捉到信号后的处理动作[结构体]
            - oldact: 上一次对信号捕捉的设置 / 信号被捕捉前的设置[结构体],可以设置NULL
        返回值: 成功[0] / 失败[-1] */
```

信号集函数

> 阻塞信号集/未决信号集
>
> `信号的"未决"是状态，指的是从信号的产生到信号被处理前的这一段时间`
>
> `信号的'阻塞'是一个开关动作,指的是阻止信号被处理,但不是阻止信号产生`
>
> `信号的阻塞就是让系统暂时保留信号信息留待以后发送`

```
// 信号集共有64个位
#include <signal.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <stdio.h>

int sigemptyset(sigest_t* set);
    /* 清空自定义信号集[信号集所有的标志位设置为0，不阻塞任何一个信号自己],可用于初始化 */
    /* 参数:
        - set: 自定义的信号集[8字节的int类型(64个标志位)] */

int sigfillset(sigset_t* set);
    /* 阻塞所有信号[自定义信号集所有的标志位值设置为1] */

int sigaddset(sigset_t* set, int signum);
    /* 阻塞信号 */
    /* 将指定信号设置到自定义信号集中[指定信号的标志位改为1] */
    /* 参数:
        - set: 自定义信号集
        - signum: 将指定信号设置到信号集中,阻塞该信号  */

int sigdelset(sigset_t* set, int signum);
    /* 将指定的信号从设置好的自定义信号集中删除出去 */

int sigismember(const sigest_t* set, int signum);
    /* 判断某个信号是否设置到了自定义信号集[标志位为1阻塞状态]*/
    /* 参数:
        - set: 自定义的信号集
        - signum: 指定信号的编号/信号对应的宏
        返回值: 成功[0] / 失败[-1]  [标志位为1阻塞状态]  */

int sigprocmask(int how, const sigset_t* set, sigset_t* oldset);
    /* 将自定义信号集中的数据映射到内核中去,修改内核的阻塞信号集 */
    /* 参数:
```

```
        - how:     --- 如何对内核阻塞信号集进行处理
            SIG_BLOCK: 将用户设置的阻塞信号信息添加到内核中,内核中原本的数据不变
                假设内核中默认的信号集mask, mask |= set
            SIG_UNBLOCK: 清除用户在内核中设置的数据,    mask &= ~set (内核中的标志位取反)
            SIG_SETMASK: 覆盖原来的值
        - set: 已经初始化/设置好的自定义阻塞
        - oldset: 拷贝设置前内核中阻塞信号集中的状态(旧状态)[可以设置为空] */

int sigpending(sigset_t* set);
    /* 读取内核中的未决信号集 */
    /* 参数:
        - set: 传出参数,保存内存中未决状信号集的信息
    */
```

守护进程

> 守护进程[精灵进程] 是`后台进程`,`守护进程脱离终端，输入输出会到自己的文件里`,`周期性的执行某些动`,
>
> `需要将进程提升为会话,它才能变成守护进程`

进程组

> PGID
>
> `多个进程的集合[最少一个]`,`Linux里所有的进程都属于某一个进程组`,`默认父子进程再同一个进程组里`
>
> `当前进程组中第一个进程就是进程组默认组长`,`进程组ID和进程组组长的进程ID相同`

```
    pid_t getpgrp(void);                                /* 获取当前进程所在进程组的组ID */
        /* 返回值: 当前进程组的组ID  POSIX标准 */

    pid_t getpgid(pid_t pid);                           /* 获取指定进程所在进程组的组ID */
        /* 参数:
            - pid: 要查询的进程的PID
        返回值: 指定进程的组ID */

    int setpgid(pid_t pid, pid_t pgid);                 /* 重新指定某个进程的进程组/重新创建新的进程组 */
        /*参数:
            - pid: 要操作进程的PID
            - pgid: 新进程组ID[设置自己的ID创建新的进程组]
        返回值:  */
```

会话

> SID
>
> `多个进程组的集合`, `只有普通进程才能创建新的会话[进程组组长和会话的会长不能重新创建会话]`,
>
> `普通进程创建新的会话会脱离原来的操作终端`

```
    #include <unistd.h>
    pid_t getsid(pid_t pid);            /* 查看指定进程的会话ID */
        /* 参数:
            - pid: 要查看的进程
        返回值: 成功[会话id] 失败[-1]
        */
    pid_t setsid(void);                 /* 调用该函数的进程创建会话[只有普通进程能创建会话] */
        pid_t sid = setsid();
        /* 返回:当前会话id 失败[0] */
```

创建守护进程

> 守护进程的创建步骤:`父进程创建子进程【必须的步骤】`-->`杀死父进程[释放父进程资源]【必须的步骤】`
>
> -->`将子进程提升为会话[setsid()]`->`修改进程的工作目录,防止有些不安全目录被卸载[chdir()]`
>
> -->`修改文件umask掩码[文件权限]` -->`关闭/重定向 文件描述符[脱离绑定在终端的输出]`
>
> -->`守护进程核心的操作流程【必须的步骤】`

```
#include <unistd.h>
    int chdir(const char* path);          /*切换进程目录*/
        /*参数:
            - path: 进程要切换的目录
        */

#include <sys/types.h>
#include <sys/stat.h>
    mode_t umask(mode_t mask);          /*修改文件umask掩码*/
        /* 参数:
            - mask: 8进制数
        */

/* 关闭绑定在终端的标准输入/输出/错误  或者重定向 */
     - 标准输入 --> close(0)
     - 标准输出 --> close(1)
     - 标准错误 --> clsoe(2)

     - 重定向文件描述符: /dev/null  设备文件（垃圾回收站）  --> 推荐
        int fd = open("/dev/null", O_RDWR)
        dup2(fd, STDIN_FILENO);
        dup2(fd, STDOUT_FILENO);
        dup2(fd, STDERR_FILENO);
```

线程

> 线程是轻量级进程(LWP), Linux下本质仍然是进程,线程是从进程中分出去的
>
> Linux以进程为单位分配进程,线程是系统资源调度的最小单位,
>
> 线程分出去系统会当作进程处理[线程参与抢占CPU资源]
>
> 线程存在进程当中(进程可以认为是线程的容器),多个子线程与父线程共用同一个虚拟地址空间

> 安装线程 man page: `"sudo apt-get install manpages-posix-dev"`
>
> `ps -Lf` 查看线程的LWP号

> 线程共享资源: `文件描述符表`, `信号的处理方式[信号捕捉后的回调函数]`, `当前的工作目录`, `用户ID和组ID`, `内存地址空间[.text[代码区] .data[全局变量区] .bss heap[堆] 共享库]`
>
> 线程不共享资源: `线程ID[unsigned long int]`, `处理器现场和栈指针[内核栈]`, `独立的栈空间[栈]`, `errno变量`, `阻塞信号集`, `调度优先级[线程的优先级]`

线程基础操作

```
/* 编译有调用子线程的库时必须添加线程库的名字[pthread] */
" gcc pthread_create.c -lpthread"

'创建子线程'
#include <pthread.h>
/* 创建的每一个子线程只会执行自己回调函数里面的代码 */
    int pthread_create(pthread_t* thread, const pthread_attr_t* attr,
                       void *(*start_routine) (void *), void* arg);
        /* 参数:
            - thread: 传出参数,线程创建成功后保存着子线程的id
                            [pthread_t线程id的类型(unsigned long int)]
            - attr: 设置线程属性[可以不使用, 传NULL(使用默认的系统属性)]
            - start_routine: 回调函数,子线程的处理逻辑
            - arg: 给第三个参数[回调函数]传参
        返回值:  成功【0】 / 失败【错误号, char* 类型】    strerror(线程名)可获取线程名 */
        pthread_t tid;
        int number =100;
        int ret = pthread_create(&tid, NULL, callBack, (void*)&number);

'获取当前线程的线程ID'
#include <pthread.h>
    pthread_t pthread_self(void);                   /* 返回线程ID */
```

```
"线程退出"
/* 线程执行完毕退出不会对其他子线程产生影响(包括主线程退出) */
#include <pthread.h>
    void pthread_exit(void* retval);
        /* 参数
            - retval: 线程退出的时候的返回值   可以指定为NULL*/
        pthread_exit(NULL);
```

```
'主线程回收子线程资源'
#include <pthread.h>
    int pthread_join(pthread_t thread, void** retval);
        /* 阻塞函数,调用一次回收一次子进程 */
        /*参数:
            - thread: 需要回收的子线程ID
            - retval: 二级指针, 使用这个变量接收子进程退出时返回的值 [可以指定为NULL] 也可以接收返回值
        */
```

```
'线程分离'
    /*设置了线程分离后,子线程执行完毕后,不需要主线程回收子线程资源 */
#include <pthread.h>
    int pthread_detach(pthread_t thread);
        /*子线程创建成功后,设置分离,主线程执行完毕后设置线程退出,就不会影响子进程继续运行 */
        /*参数:
            - thread: 要和主线程分离的子线程ID
            */
```

```
"线程取消"
/* 主线程调用线程取消函数,可以终止子进程的运行 */
/* 函数调用后并不能马上终止子进程,当子线程处理函数运行到一个取消点的位置,子线程终止 4*/
/* 如果子线程没有取消点,子进程就不会终止 */
/* 取消点: 在程序中有从用户区到内核区的切换[调用系统函数],这个位置就是取消点 */
#include <pthread.h>
    int pthread_cancel(pthread_t thread);
        /* 主线程调用该函数 */
        /*参数:
            -thread: 要取消的线程ID
        */
```

```
"比较线程ID是否相同"
/* 在Linux下线程ID[无符号长整形], 有的操作系统封装不同pthread_t是结构体,无法用 == 比较两个
线程ID */
#include <pthread.h>
    int pthread_equal(pthread_t t1, pthread_t t2);
        /*参数:
            要比较的两个线程的ID
        返回值: 两个ID相同【非0】  不同【0】*/
```

```
"线程属性"
/* 设置线程属性需要先申请一个变量保存属性,使用完后释放 */
/* 线程的属性类:  pthread_attr_t */
#include <pthread.h>
    int pthread_attr_init(pthread_attr_t* attr);                /* 线程属性结构体初
始化 */
    int pthread_attr_destroy(pthread_attr_t* attr);             /*释放线程属性*/、
        pthread_attr_t t;
        pthread_attr_init(&t);
        pthread_attr_destroy(&t);

"通过线程属性实现线程分离"
#include <pthread.h>
    int pthread_attr_setdetachstate(pthread_attr_t* attr, int detachstate);
        /*参数:
            - attr: 线程属性结构体
            - detachstate:
                PTHREAD_CREATE_DETACHED: 设置线程分离
                PTHREAD_CREATE_JOINABLE: 设置主线程子线程不分离
        */
    int pthread_attr_getdetachstate(const pthread_attr_t* attr, int*
detachstate);
        /* 获取线程的状态 */
```

线程同步

> 当一个线程对内存进行操作时,其他线程都步可以对这个内存地址进行操作,知道该线程完成操作

互斥锁/互斥量

> 互斥锁的类型: `pthread_mutex_t` 可以理解为结构体
>
> 互斥锁可以让多个线程串行处理临界区的资源(`代码块`)
>
> `restrict修饰符 (被修饰过的变量不能被其他指针引用),用其他指针操作修饰过变量的地址是不允许的`

```
    ' restrict 是一个修饰符 '
    ' 线程初始化之前就需要创建锁,并初始化 '
     pthread_mutex_t mutex;              /* 创建互斥锁 */
#include <pthread.h>
    int pthread_mutex_init(pthread_mutex_t* restrict mutex,
                           const pthread_mutexattr_t* restrict attr);       /* 初始化互斥锁*/
        /*参数:
            - mutex: 互斥锁的地址
            - attr: 互斥锁的属性,一般使用默认属性【NULL】        */
    int pthread_mutex_destroy(pthread_mutex_t* mutex);          /* 释放互斥锁的资源 */

    int pthread_mutex_lock(pthread_mutex_t* mutex);             /* 将参数指定的互斥锁上锁 */
        /* 如果已被锁上,其他线程调用加锁就会被阻塞在这里 */

    int pthread_mutex_trylock(pthread_mutex_t* mutex);          /* 将指定参数的互斥锁尝试上锁 */
        /* 尝试加锁,加锁失败函数直接返回,不会阻塞在这里 */

    int pthread_mutex_unlock(pthread_mutex_t* mutex);           /* 解锁函数 */
```

读写锁

> 读写锁是一把锁(锁定读操作,锁定写操作)
>
> 读写锁的类型:`pthread_rwlock_t`
>
> `读取的时候可以并发进行, 写的时候独占, 写的优先级高于读的优先级`
>
> `加读锁可以阻塞写锁,读锁全部关闭后写锁锁上,不允许读锁`

```
#include <pthread.h>
    int pthread_rwlock_init(pthread_rwlock_t* restrict rwlock,      /* 初始化读写锁 */
                            const pthread_rwlockattr_t* restrict attr);
        /* 参数:
            - rwlock: 读写锁的地址
            - attr: 读写锁的属性 [使用默认属性 设置NULL]
            */
    int pthread_rwlock_destroy(pthread_rwlock_t*  rwlock);      /* 释放读写锁资源 */
```

```
    int pthread_rwlock_rdlock(pthread_rwlock_t*  rwlock);        /* 加读锁[失败会阻塞在这里等待] */
        /* 被加了写锁时, 加读锁被阻塞, 读锁可以同时进行 */

    int pthread_rwlock_tryrdlock(pthread_rwlock_t*  rwlock);     /* 尝试加读锁[失败会直接返回] */

    int pthread_rwlock_wrlock(pthread_rwlock_t*  rwlock);        /* 加写锁[失败阻塞等待] */

    int pthread_rwlock_trywrlock(pthread_rwlock_t*  rwlock);     /* 尝试加写锁[失败直接返回] */

    int pthread_rwlock_unlock(pthread_rwlock_t*  rwlock);        /* 读写锁解锁 */
```

条件变量

> 条件变量能够引起某个线程的阻塞
>
> `---某个条件满足之后,阻塞线程`
>
> `---某个条件满足之后,解除线程阻塞`
>
> 使用条件变量进行线程同步,多个线程操作共享数据,不能解决数据混乱问题,`需要配合互斥锁使用`
>
> 条件变量的类型: `pthread_cond_t`

```
    steuct timespec{        /* 时间结构体,从1970-01-01算 */
        time_t tv_sec;
        long  tv_nsec;
    };

    struct timeval{
        time_t tv_sec;          /* 秒 */
        suseconds_t tv_usec;    /* 微秒 */
    };
    struct timezone{
        int tz_minuteswest;
        int tz_dsttime;
    };
#inlcude <sys/time.h>
    int gettimeofday(struct timeval* tv, struct timezone* tz);
```

```
#include <pthread.h>
    int pthread_cond_init(pthread_cond_t* restrict cond,            /* 初始化条件变量 */
                          const pthread_condattr_t* restrict attr);
        /* 参数:
            - cond: 指向条件变量的存放地址
            - attr: 条件变量的属性[使用默认属性,这个值设置为 NULL]     */

    int pthread_cond_destroy(pthread_cond_t* cond);     /* 释放条件变量资源 */

    int pthread_cond_wait(pthread_cond_t* restrict cond,  /* 线程调用该函数后一直阻塞 */
```

```
                        pthread_mutex_t* restrict mutex);
        /* 参数:
            - cond: 条件变量
            - mutex: 互斥锁 */

    int pthread_cond_timedwait(pthread_cond_t* restrict cond, pthread_mutex_t*
restrict mutex,
                               const struct timespec* restrict abstime); /* 在指定
时间后解除阻塞*/
        /* 参数:
            - cond: 条件变量
            - mutex: 互斥锁
            - abstime: 阻塞的时间 上有结构体和函数
                - 当前的时间 + 要阻塞的时长
                    struct timeval val;
                    gettimeofday(&val, NULL);  */

    int pthread_cond_signal(pthread_cond_t* cond);      /* 唤醒一个或多个阻塞在环境变
量函数上的线程 */

    int pthread_cond_broadcast(pthread_cond_t* cond);   /* 唤醒所有在阻塞环境变量的函
数 */
```

```
.....
```

Linux 网络通信

网络套接字

> 这个结构体存储IP地址和端口

```
typedef unsigned short   uint16_t;
    typedef unsigned int   uint32_t;
    typedef uint16_t   in_port_t;
    typedef uint32_t   in_addr_t;
    typedef unsigned short int   sa_family_t;
    #define __SOCKADDR_COMMON_SIZE (sizeof(unsigned short int))

    struct in_addr{
        in_addr_t  s_addr;      /* ip地址 unsigned int */
    };

    struct sockaddr{
        sa_family_t sa_family;          /* 地址组协议[IP协议] unsigned short int */
        char sa_data[14];               /* IP和端口 */
    }

    struct sockaddr_in{             /* IPv4 【可以强转 struct sockaddr类型】*/
        sa_family_t  sin_family;    /* 地址族/通讯协议 【AF_INET, AF_INET6】unsigned
short int */
        in_port_t  sin_port;        /* 端口 unsigned short */
        struct in_addr  sin_addr;   /* IP地址结构体  unsigned int*/
            // sin_addr.s_addr设置宏值 INADDR_ANY 操作系统会自动获取本机IP地址赋值给变量
```

```
        unsigned char sin_zero[sizeof(struct sockaddr) - __SOCKADDR_COMMON_SIZE
-
                               sizeof(in_port_t) - sizeof(struct in_addr)];
    };
```

字节序转换函数

> 小端(主机字节序):低地址存放低位节字　　大端(网络字节序):低地址存放高位字节

```
$ netstat 查看网络信息命令
```

```
#include <arpa/inet.h>

" short int  --> 2字节 【 *ro*s() --> 进行端口转换 】"
    uint16_t htons(uint16_t hostshort);     /* 主机字节序转换成网络字节序 短整型 */
        /* 参数:
            - hostshort: 主机字节数的short型数值(要转换的主机字节序)
        返回值: 转换后得到数(网络字节序) */

    uint16_t ntohs(uint16_t netshort);      /* 网络字节序转主机字节序 */
        /* 参数:
            - netshort: 要转换的网络字节序
        返回值: 主机字节序  */

" long -->4字节 32位下long是4字节 【 *to*l() --> 进行IP转换 】"
    uint32_t htonl(uint32_t hostlong);      /* 主机字节序转网络字节序  长整型*/
    uint32_t ntohl(uint32_t netlong);       /* 网络字节序转主机字节序  */
```

IP地址转换函数

```
#include <arpa/inet.h>
    /* 字符串: 192.168.1.100 (点分十进制字符串) */
    /* inet_pton【p-->点分十进制字符串 IP】【n-->network】 */

    int inet_pton(int af, const char* src, void* dst);
        /* 将主机字节序的字符串IP[192.168.1.100] 转换 网络字节序的整型数 */
        /* 参数:
            - af: 地址族协议 ipv4[], ipv6]
                AF_INET         // ipv4
                AF_INET6        // ipv6
            - src: 点分十进制字符串 IP[192.168.1.100]
            - dst: 传出参数,需要一块内存的地址,将转换得到的网络字节序(整型数)传到这块内存中
        返回值: 成功[1]  失败[-1]  输入的不是一个有效的表达式[0] */

    const char* inet_ntop(int af, const void* src, char* dst, socklen_t size);
        /* 网络字节序的整型IP 转换 点分十进制字符串IP*/
        /* 参数:
            - af: 地址族协议[AF_INET, AF_INET6]
            - src: 指向要转换的 网络字节序整型IP的地址
            - dst: 指向转换成功后存储点分十进制字符串的地址
            - size: 修饰第三个参数dst对应的内存大小[socklen_t 是一个int ]
        返回值: 成功[非空指针指向dst] 失败[NULL]  */

    // 将网络地址转换成“.”点隔的字符串格式返回的地址在下一次调用将被覆盖
```

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

TCP/UDP

```
#include <sys/types.h>
#include <sys/socket.h>
/* arpa/inet.h 包含这个头文件,上面有两个头文件就可以不写了 */
#include <arpa/inet.h>

    int socket(int domain, int type, int protocol);
        /* 创建一个套接字 [用完释放文件描述符close()] */
        /* 参数:
            - domain: 地址族协议
                AF_INET                 // IPv4
                AF_INET6                // IPv6
                AF_UNIX==AF_LOCAL       // 进行本地套接字通信(进程间通信) [这两个宏值是等价的]
            - type: 通信过程中使用的协议类型
                SOCK_STREAM             // 流式协议,【默认使用TCP】
                SOCK_DGRAM              // 报式协议,【默认使用UDP】
            - protocol:             // 协议, 一般写0[ 协议类型里默认使用的TCP]
        返回值:  成功【文件描述符(大于0), 指向内核缓冲区】  失败【-1】    */

    int bind(int sockfd, const struct sockaddr* addr, socklen_t addrlen);
        /* 绑定函数, 将fd和本地的IP+Port进程绑定 */
        /* 参数:
            - sockfd: 通过调用sockt函数得到的文件描述符
            - addr: 将IP和端口初始化到 sockaddr* addr变量中 sockaddr_in可以强转sockaddr
            -addrlen: 第二个参数struct sockaddr* addr结构体占的内存大小
        返回值: 失败【-1】 */

    int listen(int sockfd, int backlog);            /* 设置监听 */
        /* 参数:
            - sockfd: 通过调用sockt函数得到的文件描述符
            - backlog: 已经连接成功,但是还没有被处理的最大连接的数量
                指定的数指不能大于 /proc/sys/net/core/somaxconn里存储的数据,大于就会默认是它
        返回值: 失败【-1】*/

    int accept(int sockfd, struct sockaddr* addr, socklen_t* addrlen);
        /* 默认阻塞函数,阻塞等待客户端请求. 请求到达,接收客户端连接 */
        /* [用完释放文件描述符close()] */
        /*参数:
            - sockfd: 用于监听的文件描述符(套接字)
            - addr: 传出参数, 记录了连接成功客户端的IP和端口信息
            - addrlen: 保存第二个参数struct sockaddr* addr结构体的大小
        返回值: 成功【用于通信的文件描述符(大于0)】  失败【-1】 */

    int connect(int sockfd, const struct sockaddr* addr, socklen_t addrlen);
    /* 客户端使用该函数连接服务器, TCP需要,UDP不需要连接直接发送 */
    /* 参数:
        - sockfd: 用于通信的文件描述符
        - addr: 客户端要连接服务器的地址信息
        - addrlen: 第二个参数struct sockaddr* addr结构体的大小
```

```
    返回值: 连接成功【0】  连接失败【-1】*/

    int shutdown(int sockfd, int how);
    /* 套接字 半关闭函数 */
    /* 参数:
        - sokcfd: 要操作的套接字文件描述符
        - how:
            SHUT_RD: 关闭读端
            SHUT_WR: 关闭写端
            SHUT_RDWR: 关闭读写端 */

'TCP'  // TCP可以通过描述符用read/write直接操作读写
    ssize_t send(int sockfd, const void* buf, size_t len, int flags);
    ssize_t recv(int sockfd, void* buf, size_t len, int flags);

'UDP'
    ssize_t sendto(int sockfd, const void* buf, size_t len, int flags,
                   const struct sockaddr* dest_addr, socklen_t addrlen);
    /* 参数:
        - sockfd: 通信的描述符
        - buf: 要发送的数据
        - len: 数据长度
        - flags: 一般填0
        - dest_addr: 接收端信息
        - addrlen: dest_addr大小*/

    ssize_t recvfrom(int sockfd, void* buf, size_t len, int flag,
                     struct sockaddr* src_addr, socklen_t* addrlen);
    /* 阻塞函数*/
    /* 参数:
        - sockfd: 通信的描述符
        - buf: 接收数据的缓冲区
        - len: 缓冲区大小
        - flags: 一般填0
        - src_addr: 接收发送端信息的缓冲区,不需要可以指定为NULL
        - addrlen: src_addr大小(传入传出参数)*/
```

I/O多路转接

Select

```
#include <sys/select.h>
    struct fd_set   // fd_set文件描述符集,有128个字节,有1024个位,分别对应文件描述符0-1023
    {

    }       // sizeof(fd_set) == 128

    int select(int nfds, fd_set* readfds, fd_set* writefds,
               fd_set* exceptfds, struct timeval* timeout);
        /* 参数:
            - nfds: 要检测文件描述符的最大数量+1
            - readfds: 读集合(检测文件描述符指向的内核读缓冲区是否有数据)-->传入传出参数
            - writefds: 写集合(检测文件描述符指向的内核写缓冲区是否能写数据)-->传入传出参数
            - exceptfds: 异常集合-->传入传出参数,为NULL不检测
```

```
            - timeout: 阻塞or非阻塞
                NULL: 永久阻塞,直到检测到需要检测的文件描述符有变化
                timeout->tv_sec=0, timeout->tv_usec=0: 不阻塞
                timeout->tv_sec>0||timeout->tv_usec>0: 阻塞对应时长
                */

    void FD_ISSET(int fd, fd_set* set); /* 判断某个文件描述符在集合中的标志位是0还是1*/
    void FD_ZERO(fd_set* set);          /* 将某个集合全部标志位初始化为0,共有1024个标志
位*/
    void FD_CLR(int fd, fd_set* set);   /* 设置某个文件描述符在集合中的标志位为0*/
    void FD_SET(int fd, fd_set* set);   /* 设置某个文件描述符在集合中的标志位为1*/
```

poll

```
#include <poll.h>

struct pollfd{
    int fd;         /* 设置内核检测的描述符*/
    short events;   /* 内核检测文件描述符的事件类型(读POLLIN、写POLLOUT、错误POOLERR...
等)*/
    short revents;  /* 文件描述符实际发生的事件*/
}

    int poll(struct pollfd* fds, nfds_t nfds, int timeout);
        /* 参数:
            - fds: 要检测的文件描述符和事件结构体数组-->struct pollfd fds[];
            - nfds: 数组中最后一个有效元素的下标+1
            - timeout:
                0:不阻塞
                -1:阻塞(直至描述符发生变化)
                >0: 阻塞时长
        返回值: -1失败  >0检测到发生变化的描述符数量*/
```

epoll

> 内核和用户区以共享内存的方式
>
> `LT模式`:缺省模式
>
> `ET模式`:高速工作模式

```
#include <sys/epoll.h>

    typedef union epoll_data{
        void* ptr;
        int fd;
        uint32_t u32;
        uint64_t u64;
    }epoll_data_t;
    struct epoll_event{
        uint32_t events;        // Epoll events, Epoll检测的事件
            /*EPOLLIN, EPOLLOUT, EPOLLERR, EPOLLET(设置边沿触发)*/
        epoll_data_t data;      // User data variable
    }

    int epoll_create(int size);     // 创建一颗epoll红黑树
    /* 参数:
```

```
        - size: 现在版本没有意义,随便填大于0的数
    返回值: >0:epoll树的根节点(也是文件描述符)  -1:失败*/

    int epoll_ctl(int epfd, int op, int fd, struct epoll_event* event);
    /* epoll树管理: 添加节点,删除节点,修改已有节点属性*/
    /* 参数:
        - epfd: epoll_create创建出来的 树根节点
        - op:
            EPOLL_CTL_ADD: 添加新节点
            EPOLL_CTL_MOD: 修改某个节点的属性
            EPOLL_CTL_DEL: 删除某个节点
        - fd: 需要检测的文件描述符
        - event: epoll事件结构体变量,删除的时候传NULL
    */
    int epoll_wait(int epfd, struct epoll_event* events, int maxevents, int
timeout);
    /* 检测函数*/
    /* 参数:
        - epfd: epoll_create创建出来的 树根节点
        - events: 传出参数,发生变化的描述符信息
        - maxevents: events数组的大小( sizeof(events)/sizeof(struct epoll_event) )
        - timeout: 0:不阻塞  -1:一直阻塞,直到检测的某个描述符发生变化  >0:阻塞时长(毫秒)
    返回值: >0:发生变化的描述符数量  -1:失败*/
```

广播/组播/端口复用

> 广播: 只能在局域网中使用,接收端需要绑定本机与服务器发送端口

IP地址	说明
224.0.0.0- 224.0.0.255	局部链接多播地址:是由路由协议和其他用途保留的地址,路由器并不转发属于此范围的IP包
224.0.1.0- 224.0.1.255	预留多播地址:公用组播地址,可用于internet; 使用前需申请
224.0.2.0- 238.255.255.255	预留多播地址:用户可用组播地址(临时组地址), 全网范围内有效
239.0.0.0- 239.255.255.255	本地管理组播地址,供组织内部使用,类似于私有IP地址,不能用于internet,可限制多播范围

```
#include <net/if.h>

    unsigned int if_nametoindex(const char* ifname);     // 同过网卡名字获取网卡编号
    char* if_indextoname(unsigned int ifindex, char* ifname); // 通过网卡编号获取网
卡名字
```

```
    struct in_addr{
      in_addr_t s_addr;
    };

    struct ip_mreqn{
        struct in_addr imr_multiaddr;    // 组播组IP
        struct in_addr imr_address;      // 本地某一网络设备接口的IPD地址
        int imr_ifindex;                 // 网卡编号,if_nametoindex()获取
```

```
};

int setsockopt(int sockfd, int level, int optname,
               const void* optval, socklen_T optlen);
/* 设置端口复用在描述符bind之前设置(服务器程序重启之后,端口还未被释放)
    设置UDP广播属性 */
/* 参数:
    - sockfd: 要设置的套接字文件描述符
    - level: 权限级别
        SOL_SOCKET: 端口复用/UDP广播
        IPPROTO_IP: 组播/
    - optname: 操作
        SO_REUSEADDR: IP地址复用
        SO_REUSEPORT: 端口复用
        SO_BROADCAST: 允许发送广播数据段
    - optval: -->地址里面存放的值(int)
        1: 可以 复用/广播
        0: 不能 复用/广播
    - optlen: optval对应的内存大小(sizeof(类型))
返回值: 成功【0】, 失败【-1】*/

// 发送广播
setsockopt(sockfd, SOL_SOCKET, SO_BROADCAST, int*, sizeof(int));

// 服务端指定外出接口(组播)
setsockopt(sockfd, IPPROTO_IP, IP_MULTICAST_IF, struct in_addr*,
           sizeof(struct in_addr) );
// 客户端加入组播
setsockopt(sockfd, IPPROTO_IP, IP_ADD_MEMBERSHIP, struct ip_mreqn*,
           sizeof(struct ip_mreqn) );
```

level（级别）	optname（选项名）	get	set	说　明	标志	数据类型
SOL_SOCKET	SO_BROADCAST	•	•	允许发送广播数据报	•	int
	SO_DEBUG	•	•	开启调试跟踪	•	int
	SO_DONTROUTE	•	•	绕过外出路由表查询	•	int
	SO_ERROR	•		获取待处理错误并清除		int
	SO_KEEPALIVE	•	•	周期性测试连接是否仍存活	•	int
	SO_LINGER	•	•	若有数据待发送则延迟关闭		linger()
	SO_OOBINLINE	•	•	让接收到的带外数据继续在线留存	•	int
	SO_RCVBUF	•	•	接收缓冲区大小		int
	SO_SNDBUF	•	•	发送缓冲区大小		int
	SO_RCVLOWAT	•	•	接收缓冲区低水位标记		int
	SO_SNDLOWAT	•	•	发送缓冲区低水位标记		int
	SO_RCVTIMEO	•	•	接收超时		timeval()
	SO_SNDTIMEO	•	•	发送超时		timeval()
	SO_REUSEADDR	•	•	允许重用本地地址	•	int
	SO_REUSEPORT	•	•	允许重用本地端口	•	int
	SO_TYPE	•		取得套接字类型		int
	SO_USELOOPBACK	•	•	路由套接字取得所发送数据的副本	•	int
IPPROTO_IP	IP_HDRINCL	•	•	随数据包含的IP首部	•	int
	IP_OPTIONS	•	•	IP首部选项		(见正文)
	IP_RECVDSTADDR	•	•	返回目的IP地址	•	int
	IP_RECVIF	•	•	返回接收接口索引	•	int
	IP_TOS	•	•	服务类型和优先权		int
	IP_TTL	•	•	存活时间		int
	IP_MULTICAST_IF	•	•	指定外出接口		in_addr()
	IP_MULTICAST_TTL	•	•	指定外出TTL		u_char
	IP_MULTICAST_LOOP	•	•	指定是否环回		u_char
	IP_ADD_MEMBERSHIP		•	加入多播组		ip_mreq()
	IP_DROP_MEMBERSHIP		•	离开多播组		ip_mreq()
	IP_BLOCK_SOURCE		•	阻塞多播源		ip_mreq_source()
	IP_UNBLOCK_SOURCE		•	开通多播源		ip_mreq_source()
	IP_ADD_SOURCE_MEMBERSHIP		•	加入源特定多播组		ip_mreq_source()
	IP_DROP_SOURCE_MEMBERSHIP		•	离开源特定多播组		ip_mreq_source()

本地套接字

```
#include <sys/un.h>
    #define UNIX_PATH_MAX 108
    struct sockaddr_un{
        sa_family_t sun_family;     // 地址族协议[AF_LOCAL||AF_UNIX]
AF_LOCAL==AF_UNIX
        char sun_path[UNIX_PATH_MAX];   // 套接字文件(伪文件) 路径,不需要手动创建
    }
    /* 本地套接字通信流程和TCP是一样, socket(AF_LOCAL,sock_stream, 0 ) */
```

```
    // 如果使用本地套接字,bind失败,显示被占用,可以删除那个套接字文件
    #include <unistd.h>
    int unlink(const char* path);       // 删除文件
```

LibEvent框架

> `www.libevent.org`
>
> `LibEvent`: 基于回调的事件驱动

```
'LibEvent 安装'
    # 解压LibEvent安装包
    $: tar zxvf libevent-2.1.8-stable.tar.gz
    # 进入解压后的文件夹,执行configure执行(检测安装环境,并生成makefile)
    $: sudo ./configure
    # 如果提示没有 openssl,执行后重新 ./configure
        $: sudo apt-get install libssl-dev
    # 安装(把 ./.libs/libevent.so 拷贝到 /user/local/lib/libevent.so)
    $: sudo make install
```

```
$: gcc server.c -o a.out -levent        // 编译的时候要加载第三方库 levent
```

```
#include <event2/event.h>
    struct event_base{
        const struct eventop* evsel;
        void* evbase;
        int event_count;
        int event_count_active;
        ....
    };

    struct event_base* event_base_new(void);    // 创建event_base框架, 返回指针
    void event_base_free(struct event_base* base); // 释放event_base框架, 返回指针
    int event_reinit(struct event_base* base);  // 重新初始化event_base框架,子进程被
创建后

    const char** event_get_supported_methods(void);     // 检测支持哪些I/O转接函数
    const char* event_base_get_method(
            const struct event_base* base);         // 查看当前使用的I/O转接函数
```

```
#include <event2/event.h>
```

```
#define EV_TIMEOUT              0x01
#define EV_READ                 0x02
#define EV_WRITE                0x04
#define EV_SIGNAL               0x08
#define EV_PERSIST              0x10        // 修饰某个事件是持续触发的
#define EV_ET                   0x20        // 边沿模式
typedef void (*event_callback_fn)(evutil_socket_t fd, short what, void* arg);
    // 事件处理回调函数
    /* 参数:
        - fd: 触发事件的描述符
        - what: 实际触发的事件
        - arg: 指向event本身 */

'事件操作'
    struct event* event_new(struct event_base* base, evutil_socket_t fd,short
what,
        void (*event_callback_fn)(evutil_socket_t, short, void*),void* arg);
    // 创建事件实例,事件被nwe出来不能直接被检测
    /* 参数:
        - base: base事件框架
        - fd: 要检测的描述符
        - what: 需要检测的事件  ---->上面的宏
        - event_callback_fn: 事件处理回调函数--> fn(fd, what, arg); 与这些参数相同
        - arg: 指向event本身*/
    int event_add(struct event* ev, const struct timeval* tv);
    // 添加事件实例到检测框架树
    /* 参数:
        ev: 事件实例
        tv: 超时检测,如果tv事件内事件没有被触发,也会强制调用事件处理回调函数,为NULL不会强制
*/
    void event_free(struct event* event);   // 释放事件实例资源,先event_del再释放
    int event_del(struct event* ev);        // 将事件实例从框架检测树上摘下来
```

事件循环

> 事件循环: 检测对应的事件是否被触发了

```
#include <event2/event.h>

#define EVLOOP_ONCE             0x01
#define EVLOOP_NONBLOCK         0x02
#define EVLOOP_NO_EXIT_ON_EMPTY 0x04

struct timeval{
    long tv_sec;
    long tv_usec;
}
'启动事件循环'
    int event_base_loop(struct event_base* base, int flags);
        /* 参数:
            - base: 通过event_base_new 创建得event_base框架的指针
            - flags:
                EVLOOP_ONCE: 循环阻塞检测某个事件,事件被触发后停止事件循环
                EVLOOP_NONBLOCK: 不阻塞,不停的检测,当事件触发后停止事件循环
                EVLOOP_NO_EXIT_ON_EMPTY: 一直进行事件检测,如果没有了要检测的事件也不退出
*/
    // 不停的循环检测,所有要检测的事件都被触发了并且处理完毕结束检测
```

```
    int event_base_dispatch(struct event_base* base);   // 一般使用这个

'终止事件循环'
    // 事件处理函数正在执行,执行完毕在tv时长后退出循环事件,如果指定为NULL,直接退出
    int event_base_loopexit(struct event_base* base, const struct timeval* tv);
    // 不管是否有正在事件处理函数正在执行, 马上终止
    int event_base_loopbreak(struct event_base* base);

'事件优先级'

    // EVENT_MAX_PRIORITIES == 256
    int event_base_priority_init(struct event_base* base, int n_priorities);
    // 初始化事件框架的优先级等级
    /* 参数:
        - base: event_base框架的指针
        - n_priorities: 等级个数, 3级->[0,1,2], 等级最高EVENT_MAX_PRIORITIES
    返回值: */

    int event_priority_set(struct event* event, int priority);  // 设置事件的等级
    /* 参数:
        - event: 事件
        - priority: 事件对应的等级*/

    int event_base_get_npriorities(struct event_base* base);    // 获取当前事件的等
级个数
```

bufferEvent

> 带缓冲区的事件,主要应用于套接字通信

```
#include <event2/bufferevent.h>

typedef void(*buffevent_data_callback)(struct bufferevent*bev, void* ctx);
typedef void(*buffevent_event_cb)(struct bufferevent* bev, short events, void*
ctx);

    struct bufferevent* bufferevent_socket_new(struct event_base* base,
            evutil_socket_t fd, enum bufferevent_options options);
    /* 创建创建带缓冲区的事件,读缓冲区默认禁用的bufferevent_enable()设置,主要用于通信 */
    /* 参数:
        - base: 框架指针
        - fd: 描述符,客户端可以指定为-1,在bufferevent_socket_connect()的时候自动创建
        - options:
            BEV_OPT_CLOSE_ON_FREE: 当创建的bufferevent*被释放时,底层的资源也被释放了
    返回值: 得到带缓冲区的bufferevent事件的指针 */
    void bufferevent_free(struct bufferevent* bev);     // 释放带缓冲区的事件

    void bufferevent_setcb(struct bufferevent* bufev,
            void (*read_callback)(struct bufferevent*bev, void* ctx),
            void (*write_callback)(struct bufferevent*bev, void* ctx),
            void (*event_callback)(struct bufferevent* bev, short events, void*
ctx),
            void* arg);
    /* 设置读写事件回调函数*/
    /* 参数:
        - bufev: 带缓冲区的event事件
        - read_callback: 读事件回调函数, 事件函数参数(bufev, arg);
```

```
        - write_callback: 写事件回调函数, 事件函数参数(bufev, arg);
        - event_callback: 特殊事件处理回调函数, 事件函数参数(bufev, events[看下面宏],
arg);
            BEV_EVENT_READING: 缓冲区有读操作时发生某事件,去i读取某些数据-->对方关闭了连接
            BEV_EVENT_WRITING: 写入操作时发生某事件,当数据被bufferevent_write写入之后才
会被触发
            BEV_EVENT_ERROR: 操作时发生错误, EVUTIL_SOCKET_ERROR()查看错误信息
            BEV_EVENT_TIMEOUT: 发生超时
            BEV_EVENT_EOF: 遇到文件结束指示,对方已经关闭了连接
            BEV_EVENT_CONNECTED: 请求的连接过程已经完成,客户端建立连接成功会第一次触发
        - arg: 回调函数参数指针*/

'事件检测操作'
#define EV_TIMEOUT              0x01
#define EV_READ                 0x02
#define EV_WRITE                0x04
#define EV_SIGNAL               0x08
#define EV_PERSIST              0x10        // 修饰某个事件是持续触发的
#define EV_ET                   0x20        // 边沿模式

    void bufferevent_enable(struct bufferevent* bufev, short events);   //设置某个
事件有效
    void bufferevent_disable(struct bufferevent* bufev, short events);  //设置某个
事件无效
    short bufferevent_get_enabled(struct bufferevent* bufev);   // 获取带缓冲区事件
的有效事件

'缓冲区操作'
    // 向缓冲区写入数据
    int bufferevent_write(struct bufferevent* bufev, const void* data, size_t
size);

    // 向缓冲区读出数据,读出后缓冲区数据清空
    size_t bufferevent_read(struct bufferevent* bufev, void* data, size_t size);
```

```
'客户端连接服务器'
#include <event2/bufferevent.h>
    int bufferevent_socket_connect(struct bufferevent* bev,
                    struct sockaddr* address, int addrlen);
    // 连接服务器函数
    /* 参数:
        - bev: 带缓冲区的event事件,里面封装了fd
        - address: 地址族协议/端口/IP
        - addrlen: address的内存大小*/
'服务器监听'
#include <event2/listener.h>
    typedef void (*evconnlistener_callback)(struct evconnlistener* listener,
            evutil_socket_t sock, struct sockaddr* addr, int len, void* ptr);
    /* 参数:
        - listener: evconnlistener_new()/...new_bind() 时创建的监听event的指针
        - sock: 通信的描述符
        - addr: 客户端的ip和端口
        - len: addr的大小
        - ptr: evconnlistener_new()/...new_bind() 时传入的回调函数参数*/

    struct evconnlistener* evconnlistener_new(struct event_base* base,
            evconnlistener_callback ev_cb, void* ptr, unsigned flags,
```

```
            int backlog, evutil_socket_t fd);
    // 以绑定端口的描述设置监听
    /* 参数:
        - base: libevent框架指针
        - ev_cb: 描述符的回调函数
        - ptr: 回电函数的最后一个参数
        - flags: 描述符属性
            LEV_OPT_CLOSE_ON_FREE: 监听event 释放的时候, 资源也会被释放
            LEV_OPT_REUSEABLE: 设置端口复用,监听描述符被关闭后,其他描述符能马上绑定端口
        - backlog: 已经连接成功,还没有被处理的最大连接的数量,设置-1,会自动选择一个
        - fd: 监听 已经创建出来并且绑定端口的描述符*/

    struct evconnlistener* evconnlistener_new_bind(struct event_base* base,
            evconnlistener_callback ev_cb, void* ptr, unsigned flags,
            int backlog, const struct sockaddr* sa, int socklen);
    // 创建用于监听的描述符,并且绑定和设置监听
    /* 参数:
        - base: libevent框架指针
        - ev_cb: 描述符的回调函数,接收新连接后
        - ptr: 回电函数的最后一个参数
        - flags: 描述符属性
            LEV_OPT_CLOSE_ON_FREE: 监听event 释放的时候, 资源也会被释放
            LEV_OPT_REUSEABLE: 设置端口复用,监听描述符被关闭后,其他描述符能马上绑定端口
        - backlog: 已经连接成功,还没有被处理的最大连接的数量,设置-1,会自动选择一个
        - sa: 地址族协议/端口/IP, 本地的
        - socklen: sa的大小*/

    void evconnlistener_set_cb(struct evconnlistener* lev,
            evconnlistener_callback cb, void* arg);          // 重置监听事件的回调函
数
    void evconnlistener_free(struct evconnlistener* lev);    // 释放 监听event


    int evconnlistener(struct evconnlistener* lev);      // 设置监听无效
    int evconnlistener_enable(struct evconnlistener* lev);  // 设置监听有效
```

HTTP

```
http状态码:
    1xx：指示信息--表示请求已接收，继续处理
    2xx：成功--表示请求已被成功接收、理解、接受
    3xx：重定向--要完成请求必须进行更进一步的操作
    4xx：客户端错误--请求有语法错误或请求无法实现
    5xx：服务器端错误--服务器未能实现合法的请求
常见状态码:
    200 OK                          客户端请求成功
    400 Bad Request         客户端请求有语法错误，不能被服务器所理解
    401 Unauthorized      请求未经授权，这个状态代码必须和WWW-Authenticate报头域一起使用
    403 Forbidden               服务器收到请求，但是拒绝提供服务
    404 Not Found             请求资源不存在，eg：输入了错误的URL
    500 Internal Server Error      服务器发生不可预期的错误
    503 Server Unavailable         服务器当前不能处理客户端的请求，一段时间后可能恢复正常
```

请求报文

```
// get请求: 第一行的第二部分: 包含了用户提交的数据, 从请求行的第二部分的?开始
GET /?username=subwen%40qq.com&phone=11223344&email=11%40aa.com&date=2019-01-
01&sex=male&class=3&rule=on HTTP/1.1
Host: 192.168.36.58:6789
Connection: keep-alive
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML,
like Gecko) Chrome/73.0.3683.75 Safari/537.36
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;
q=0.8,application/signed-exchange;v=b3
Accept-Encoding: gzip, deflate
Accept-Language: zh,zh-CN;q=0.9,en;q=0.8
```

```
// post请求
POST / HTTP/1.1
Host: 192.168.36.58:6789
Connection: keep-alive
Content-Length: 98
Cache-Control: max-age=0
Upgrade-Insecure-Requests: 1
Origin: null
Content-Type: application/x-www-form-urlencoded
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML,
like Gecko) Chrome/73.0.3683.75 Safari/537.36
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;
q=0.8,application/signed-exchange;v=b3
Accept-Encoding: gzip, deflate
Accept-Language: zh,zh-CN;q=0.9,en;q=0.8

username=subwen%40qq.com&phone=11223344&email=11%40aa.com&date=2019-01-
01&sex=male&class=3&rule=on
```

响应报文

```
// 响应报头
HTTP/1.1 200 Ok
Server: micro_httpd
Date: Fri, 18 Jul 2014 14:34:26 GMT
/* 告诉浏览器发送的数据是什么类型 */
Content-Type: text/plain; charset=iso-8859-1 (必选项) 西欧编码->不支持中文
/* 发送的数据的长度 */
Content-Length: 32
Location:url
Content-Language: zh-CN
Last-Modified: Fri, 18 Jul 2014 08:36:36 GMT
Connection: close

#include <stdio.h>
int main(void)
```

```
{
    printf("hello world!\n");
    return 0;
}
```

字符转换

> utf-8转换asiic

```
    encodeStr(char* to, int tosize, const char* form)
```

数据库

MySQL

搭建MySQL数据库

```
# 安装MySQL服务
$: sudo apt-get install mysql-server
$: sudo apt install mysql-client
$: sudo apt install libmysqlclient-dev
•
# 检查MySQL是否安装成功
$: sudo netstat -tap | grep mysql
•
# 数据库测试
$: mysql -u root -p
•
# 输入密码
    mysql>                          '如果界面变成这样就说明进入了MySQL'
    mysql>create database web;      '创建网站数据库，一会儿要用 web单词可以替换自定义'
    mysql>quit                      '输入 quit 退出'
•
# 开放端口 (将bind-address = 127.0.0.1注释)
    `MariaDB配置文件位置: /etc/mysql/mariadb.conf.d/50-server.conf`
    `MySQL配置文件位置: /etc/mysql/mysql.conf.d/mysqld.cnf`
# 输入以下命令查看MySQL账户密码(可以自行修改)
$: sudo cat /etc/mysql/debian.cnf
•
    user = debian-sys-maint
    password = v2jTk4fe7U6HRP63

`mysql服务设置`
$: sudo service mysql start     '开启MySQL服务'
$: sudo service mysql stop      '停止MySQL服务'
$: sudo service mysql restart   '重启MySQL服务'
```

MySQL用户配置

```
`新增用户`
    #允许本地IP访问localhost，127.0.0.1的新建用户
    sql> create user 'test'@'localhost' identified by '123456';
    #允许外网IP访问，%通配所有远程主机
    sql> create user 'root'@'%' identified by '123456';

`删除用户`
    sql> delete from mysql.user where user = 'test';

`对新用户分配数据库权限`
    sql> grant all on *.* to 'root'@'%' identified by '123456';      #对所有库拥有权
限
    sql> grant all on db1.* to 'test'@'%' identified by 'password'; #对指定库db1拥
有权限
    sql> grant select on db1.t1 to 'gggg3'@'%' identified by '123';
                                                                    # 对指定库下的指定表拥有
权限db1.t1
    sql>grant select  (id,name) on db1.ti to 'gggg4'@'%' identified by '123';
                                                                    #赋予查看id和name的权限
    sql> grant select (id,name),update (name) on db1.t1 to 'gggg5'@'%'
identified by '123';
                #赋予查看id和name的权限，并赋予修改name的权限

`删除用户的库权限`
    sql> revoke select on db1.* to 'gggg5'@'%';                    #删除权限
    sql> revoke select,update,insert,delete on mohui.* from '1234567'@'%';
`刷新`
    sql> flush privileges;
```

Oracle

> 安装Oracle数据库会默认自动创建scott和hr两个用户以及其方案

Oracle命令行

```
'命令行'
    $: sqlplus / as sysdba          # 以oracle超级管理员登陆oracle
    $: sqlplus 用户名/密码           # 本地登录Oracle
    $: sqlplus 用户名/密码@//IP/实例名   # 远程登陆Oracle,实例名默认orcl

    $: lsnrctl start                # 启动监听服务,对internet开放
    $: lsnrctl stop                 # 关闭监听服务
    $: lsnrctl status               # 查看监听状态

    SQL> startup;                           # 启动数据库实例
    SQL> shutdown;                          # 关闭数据库实例
    SQL> exit | quit;                               # 退出SQL命令行

    SQL> show user;                         # 查看当前登录账户
    SQL> alter user 用户名 account unlock;          #给用户解锁,oracle的用户默认锁定
scott
    SQL> alter user 用户名 identified by "新密码";         # 修改用户密码
```

```
    SQL> select userenv("language") from dual;       # 查看当前语言环境
    SQL> select * form v$nls_parameters;         #查看当前sqlplus会话环境变量

    # 设置oracle默认加载项
./Oracle/WINDOWS.X64_193000_db_home/sqlplus/admin/glogin.sql
    SQL> source ./path/to/sql-file.sql;          # 执行本地 SQL 文件中的的 SQL 语句
    SQL> set linesize 140;                       # 设置显示的行宽大小为140
    SQL> set pagesize 50;                        # 设置显示一页数据量(多少条)
    SQL> alter session set NLS_DATE_FORMAT = 'yyyy-mm-dd';  # 修改日期格式
```

```
    SQL> select * from tab;              # 查看当前用户拥有哪些表
    SQL> desc 表单名;                      # 查看指定表单结构
```

```
# SQL比较运算符
    = 等于          > 大于              >= 大于等于
    < 小于          <= 小于等于          <>或者!=   不等于
# 逻辑表达式(&& || !)
    and 与        or 或          not 非
# 通配符
    % 若干个字符      _ 任意一个字符
    escape '\'       ' SQL语句最后加上escape +'符号',表示该符号以转移字符使用
# 表达式计算
    # 表达式计算,任何数加NULL都为NULL
    SQL> select 3+列名 from dual;          # oracle提供的专门计算表达式的虚表(不存在)
        SQL> select 3+nvl(列名, 值) from dual;// 判断如果某个列的值为NULL,就返回对应第二个值
```

```
select      # --- 查询(可以select 列或者具体的值)
    '--- 查询语句得到的是结果集'
    '--- 增删改语句得到的是执行这个语句受影响的行数'
    select *;                            # 查询所有内容
    SQL> select * from 表单名;              # 查询某个表单里所有列
    SQL> select 列名 from 表单名;                 # 查询*表单的*列的数据
    SQL> select 列名1 as '别名1', 列名2 as '别名2', 列名1*10+列名2  as '别名3' from 表单名;
            #查询某单的某两列和 列1*10+列2 的数据,并对显示的列以取的别名显示,不会修改真实数值
        SQL> select 列名1 别名1, 列名2 别名2, 列名1*10+列名2 别名3 from 表单名;
            #查询某单某两列和的数据, as和双引号可以省, 双引号在别名有空格的时候不能省

    SQL> select distinct 列名 from 表单名;                    #去重显示某表单某列的数据

    SQL> select * from 表单名 where 列名=值;                   #单条件查询
        SQL> select * from 表单名 where 列名 is null;             # 判断所有列名值为null的行
        SQL> select * from 表单名 where 列名 is not null;         # 判断所有列名值不为null的行
        SQL> select * from 表单名 where 列名1=值 and 列名2='字符串';  #多条件查询,两个条件相等
        SQL> select * from 表单名 where 列名 in (值1, 值2);       #查询所有列名值为值1或者值2的行
        SQL> select * from 表单名 where 列名 not in (值1, 值2);  #查询所有列名值不为值1或者值2的行
        SQL>select * from 表单名 where 列名 not between 值1 and 值2;#查询列名值在值1和值2之间的(包括两值)
```

```
            SQL>select * from 表单名 where between '1981-02-01' and '1982-01-
31';#字符串之间

    SQL> select * from 表单名 where 列名 like 's%';  # %通配符模糊查找,某列所有s*的字符
串
        SQL> select * from 表单名 where 列名 like '____';    #查找所有 四个字符的随机
字符串

    SQL> select * from 表单名 order by 列名1 asc; #排序所有,默认升序可以asc
        SQL> select * from 表单名 order by 列名1 asc,列名2 desc
        SQL> select * from 表单名 order by 列 desc nulls last;  # 倒序排列,null排在
最后面
        SQL> select * from 表单名 where 列名=值  order by 列名1 desc,列名2 desc
                                    #排序查找,desc降序
```

```
# 单行函数
`字符函数`
    lower       # 讲字符串转小写
        select lower('hEllO', 'woRld') from dual;

    upper       # 将字符转大写
        select upper(列名1) from 表单名;

    initcap     # 将首字母变成大写
        select upper(列名1,列名2) from 表单名;
        select 'hello' || 'word' || 123 from dual;  # 可以拼接字符串

    concat      # 字符串拼接,只能支持两个参数
        select upper(列名1,列名2) from 表单名;

    substr      # 提取字符串子串
        select substr(字符串, 截取位置) from dual;          #从截取位置开始截取
        select substr(字符串, 截取位置, 截取个数) from dual;       #从截取位置开始截取截
取N个

    instr       # 查找字符串中是否包含此字字符串
        select substr(字符串, 子字符串) from dual;     # 查找字符串中包含字串,返回索引

    lpad,rpad   # 左右填充
    trim        # 裁剪
        select trim('  hello world  ') from dual;   # 默认裁剪两边空白字符
        select trim('H' form ' HH hello worHld H ') from dual;  # 指定裁剪里面的H字
符

    replace     # 替换字符串子串
        select replace("hello word", 'el', "ccc") from dual; #将字符串中的el全部替换
成ccc

`数值函数` #number
    round
    trunc
    ceil, floor
    mod

`转换函数`
    to_char     数字/日期 转字符串
        select 值, to_char(值, 'L9999') from 表单;
```

```
                #将number列的数字转换成本地货币4个9就是最大4位数
    to_number   字符串转数字
    to_data     字符串转日期

`日期函数` # data
    sysdate     # 获取系统时间,显示某日期
        select sysdata 今天, sysdata-1 昨天, sysdata+1 明天 from dual;    #日期显示
    months_between      #计算两个时间相差的月份,自动计算大小月(30, 31天)
    add_months      #添加多少月后的日期
        select add_months(sysadta, 12) from dual; #日期加12个月
    last_day        #获取当前月份最后一天
        select last_day(sysdate) from dual;
    next_day        #查看下一个星期几是几月几号
        select next_day(sysdate, '星期四') from dual;
                #填写日期的字符串写中文还是英文取决于环境变量设置

`通用函数`
    nvl(条件, 返回值);                  # 判断条件值为NULL,就返回 返回值
    nvl2(值, 返回值1, 返回值2);     # 判断exp值不为NULL,就返回val1,否则返回val2
    nullif(值1, 值2);               #当值1==值2返回null, 不相等返回值1
    coalesce(值1, 值2, ..., 值N);       # 从左往右找第一个不为null的值

`条件语句`
    case when then else end     #和c语言switch语句类似
    :    case 条件
            when 值 then 表达式
            when 值 then 表达式
            else 表达式
        end

    decode(条件, 值1, 返回值1, 值2, 返回值2, ..., 不匹配返回值);
            # Oracle额外扩充的和case语句的功能
```

```
# 多行函数
`统计函数`
    sum
        select sum(列) from 表单;       # 统计所有行的某一列值总和

    count   # 计数函数,某一行有值,计数就+1
        select count(*) from 表单;     # 一共有多少行
        select count(列) from 表单;     # 某一列有值的有多少行
        select count(distinct 列) from 表单;    # 去重复统计

    max/ min    # 最大值/最小值
        select max(列), min(列) from 表单;

    avg         # 平均值
        select avg(列) from 表单;  # 如果某一行的值NULL,则不计入统计(除的时候不会加上它)
`分组统计`
    group by
        # 根据列或者多列进行分组,该列/多列 值是相同的分为同一组
        # select 后面跟的列只能是group by后面出现的列,或者统计函数的表达式
        # 能使用where就尽量不使用having
        select 列 from 表单 group by 列1, 列2, 列3, ... having 条件;
        select 列 from 表单 group by where 条件 列1, 列2, 列3, ...;
            select 列1, 列2 from 表单 group by 列1;
```

```
            select 列1, 列2 from 表单 where 条件 group by 列1;  # 列>值 或者==之类的判断条件
            select 列1, 列2 from 表单 group by 列1 having 条件; # 列>值 或者==之类的判断条件
```

SQL语句

> SQL语句对大小写不敏感
>
> SQL语句可以写在多行以;号结束
>
> 关键词不能被缩写也不能被分行

```
    SQL> create database 数据库名;           # 创建指定名称数据库
    SQL> show databases;                  # 打开/显示全部数据库
    SQL> use 数据库名;                       # 打开/使用数据库
    SQL> drop datase 数据库名;               # 删除指定名称的数据库

    SQL> create table 表单名(int id, char(5) username, int age);
                                          # 创建指定名称的数据表单(]括号里面为表单每一列的定义类型)
    SQL> show tables                      # 显示数据库中的表单
    SQL> drop table 表单名;                  # 删除指定名称数据库表单
```

```
`SQL语句(关键词用``反引号括起来)`

    #select `列1`, `列2`, 值, `zhi` from `表单`;
                                  # 查询表单里面的指定字符段.可同时查多个字符段(列)和所有具体的值

insert      # ---新增
    insert into 表单 vaules (列1,列2,列3,列4, 列5); # 新增表单项,必须跟列一一对上
    insert into 表单 (name, id, ) value ('张三', 1);         # 新增指定列名添加指定值的列

delete      # ---删除
    delete from `表单`;                                # 删除表单所有数据
    delete from 表单 where 列1 = ufo  and id > 1;    #删除表单中列1值等于ufo的数据和id大于1的数据

update      # ---修改
    update 表单 set title = 'ceo',user='1' where id = 1;
                                          # 将id列=1的数据的title列的值设置为ceo,user值设置为1

where   # 筛选(判断)                                  # and 和     > 大于     < 小于
    delete from 表单 where id in(1,2,3,4);          #当id在1,2,3,4中出现的删处

limit   # 限制取几条数据
    select * from 表单 limit 2;             # 从表单中只取2条数据
    select * from 表单 limit 10 10;        # 越过十条数据取十条数据
    skip = (page - 1) * size
```

```
`SQL 查询函数`
    count()          # --- 分页功能,查询总条数
    max()/min()      # ---最大值,最小值
    avg()            # ---平均值
```

汇编/逆向

```
BOOL DebugActiveProcess(DWORD dwProcessId);      //使调试器连接到一个活动过程并调试它
```

x86

寄存器

cpu寄存器

```
eax:                    通常:函数执行的结果会通过eax传递
ebx:                    通常:DS段的数据指针
ecx:                    通常:计数寄存器
edx:                    通常:I/O指针
esp:                    通常:栈顶
ebp:                    通常:栈底
esi:
edi:

eip:                    cpu执行位置
```

标志位寄存器

```
EFL         /* 标志位寄存器的汇总 */

0       CF          进位标志寄存器   /* 当运算结果[数据大小]的 最高位 产生进位或借位时，为1
无符号看c位 */
2       PF          奇偶标志位          /* 运算结果转换为2进制后,里面1的个数是偶数[1] 奇
数[0] */
4       AF          辅助进位标志          /* 在操作byte时,最后一位向高位进位或者借位时
                                          在操作字形时,第3位向高位进位或借位时标志位会变
[1] */
6       ZF          零标志位            /* 当运算结果为0时其为1,否则为0 */
7       SF          符号标志位           /* 保存运算结果转换为2进制后最高位的数指 */
11      OF          溢出标志位           /* 有符号看o位 正数+正数=负数 溢出了 */
10      DF          方向位             /* 增长方向减的时候 1*/
```

汇编指令

```
    mov             /* 移动/存放数据 */
            mov dword ptr ds:[0xf10300], 100;
        movs                /* 操作数据的源地址和目标地址可以同时为内存, edi和esi会++ */
            moves dword ptr ds:[edi], dowrd ptr ds:[esi];
        movsx           /* 符号位扩展(带符号位扩展),后面的操作数必须小于前面的操作数 */
            mov al, 0xff
            movsx cx,al /* 会先将al扩展为16位(0xff)原本为负数用f填充 扩展为0xffff然后将
cx填满 cx=0xffff */
            mov al, 0xf
            movsx cx,al     /*源操作数为0x0f为正数用0填充 cx = 0x000f */
        movzx       /* 补0扩展(无符号扩展),后面的操作数必须小于前面的操作数 */
            mov al, 0xff
            mov cx, al      /* cx = 0x00ff */
```

```
    add             /* 加法 */
    adc             /* 带进位的加法 */
    sub             /* 减法 */
    sbb             /* 带借位的减法 */
```

```
 /* 位运算指令 */
    and             /* 与运算 & [两个为1才为1] */
    or              /* 或运算 | [两个有一个为1就是1] */
    xor             /* 异或   ^  两个数不相同时取1 */
    not             /* 非     ~  两个数相同时取反,有一个为1时取1 */
    lea             /* 取地址存放 */
```

```
/* 条件跳转指令 */
    cmp                 /* 比较 */
    jmp                 /* 无条件跳转 [修改eip的值]*/
                /* 地址计算: 要跳转的地址-补丁地址-5 */

    // 助记符(等于或者不等于)
    je jz               /* 条件标志位(ZF=1), ==跳转, !=不跳转向下执行  */
    jne jnz             /* 条件标志位(ZF=0), !=跳转, ==不跳转向下执行 */

    // 有符号条件跳转
    jg jnle             /* 条件标志位(OF=0||SF==ZF), >跳转, <=不跳转向下执行 */
    jge jnl             /* 条件标志位(SF=OF), >=跳转, <不跳转向下执行 */
    jl jnge             /* 条件标志位(SF!=OF), <跳转, >=不跳转向下执行 */
    jle jng             /* 条件标志位(ZF=1&&SF!=OF),<=跳转, >不跳转 */

    // 无符号条件跳转(Above高于 Below低于)
    // (相对条件 低于/高于并且等于时跳转,或者高于/低于时不跳转 两种指令)
    ja jnbe jnle        /* 条件标志位(CF=0||ZF==0), >跳转, <=不跳转向下执行 */
    jnb jae jnc         /* 条件标志位(CF=0), >=跳转, <不跳转向下执行 */
    jb jnae jc jnc      /* 条件标志位(CF=1), <跳转, >=不跳转向下执行 */
    jbe jna             /* 条件标志位(CF=1&&ZF=1), <=跳转, >不跳转向下执行*/
```

```
/* 位移指令 */
    SAL/SAR  Reg/Mem, CL/Imm    /* 算术移位指令 */
        SAL                             /* 算术左移,最高位数放入CF位,整体左移最后一位补
0 */
```

```
            SAL Reg/Mem, CL/Imm
                SAL eax,1   /* 4字节 */
                SAL ax,1    /* 2字节 */
                SAL al,1    /* 1字节 */
        SAR                             /* 算术右移,最低位放入CF位,最高位补符号位(之前
的最高位) */

    SHL/SHR Reg/Mem, CL/Imm     /* 逻辑移位指令 */
        SHL                             /* 逻辑左移,与算术左移用法一样 */
        SHR                             /* 逻辑右移, 最低为放入CF位,最高位补0 */

    [ROL r/m, i8] [ROR r/m, CL] /* 循环移位指令 */
        ROL                             /* 循环左移[整体左移],最高位放入最低位并保存到
CF位 */
        ROR                             /* 循环右移,    */
    [RCL r/m, i8][RCR r/m, CL]  /* 带进位的循环移位指令 */
        RCL                          /*带进位的循环左移[整体左移]最高位放入CF位,将原CF位
放入最低位*/
        ROR                          /* 带进位的循环右移 */
```

```
    push
        push eax;       /* esp+4,并把eax里的值放入 */
    pushad      /*本指令将EAX,ECX,EDX,EBX,ESP,EBP,ESI,EDI 这8个32位通用寄存器依次压入
堆栈,其中SP的值是在此条件指令未执行之前的值.压入堆栈之后,ESP-32->ESP.*/
    pop             /* 将栈顶的值保存进寄存器,栈顶+4 */
    popad
    ret             /* pop eip */


    xchg            /* 交换数据 */
        chng eax,edx;
    stos        /* 将eax的值存储到edi指向的内存地址里面,edi的增长方向由d标志位决定,执行完
edi会++ */
        stos byte ptr es:[edi];
        stos word ptr es:[edi];
        stop dword ptr es:[edi];
    rep             /* 按计数寄存器(ECX)中指定的次数重复执行 字符串指令每执行一次ECX减1 */
        rep movs dword ptr es:[edi],dword ptr ds:[esi];
```

##

```
and     // 逻辑_与运算
```

X64

```
'64位寄存器'
    RAX         数据易丢失          返回值寄存器
    RCX         数据易丢失          第1个整形参数
    RDX         数据易丢失          第2个整形参数
    R8          数据易丢失          第3个整形参数
    R9          数据易丢失          第4个整形参数
    R10-R11     数据易丢失          必须根据需要由调用方保留；在 syscall/sysret 指令中
使用
```

```
    R12-R15      非易丢失          必须由被调用方保留
    RDI          非易丢失          必须由被调用方保留
    RSI          非易丢失          必须由被调用方保留
    RBX          非易丢失          必须由被调用方保留
    RBP          非易丢失          可用作帧指针；必须由被调用方保留
    RSP          非易丢失          堆栈指针
    RIP                            当前执行地址

    XMM0         数据易丢失        第1个FP参数
    XMM1         数据易丢失        第2个FP参数
    XMM2         数据易丢失        第3个FP参数
    XMM3         数据易丢失        第4个FP参数
    XMM4-XMM5    数据易丢失        必须根据需要由调用方保留
    XMM6-XMM15   非易丢失          必须根据需要由被调用方保留
```

CE

CE数据类型

```
'CE搜索中的数据类型'
 //扫描结果绿色代表着该地址为 全局变量/静态变量/程序二级制代码
    字节                          char/unsigned char
    2字节                         short int/unsigned short/ wchar_t
    4字节                         int/unsigned int
    8字节                         long long/unsigned long long
    单浮点                         float
    双浮点                         double
    字符串                         char*
    字符串(UTF-16)                 wchar_t*
    字节数组                       char[]
```

OD

OD快捷键

```
'OD快捷键'
    F2                      # 下断点
    F3                      # 加载一个可执行程序
    F4                      # 程序执行到光标处
    F5                      # 缩小/还原当前窗口
    F7                      # 单步步入
    F8                      # 单步步过
    F9                      # 直接运行程序,遇到断点程序暂停
    Ctrl + F2               # 直接运行程序到起始处,重新开始调试程序
    Ctrl + F9               # 执行到返回处
    Alt + F9                # 执行到用户代码处,快速跳出系统函数
    Alt + B                 # 断点编辑栏
    Ctrl +G                 # 输入16进制地址快速定位到该地址处/API函数,快速定位到该API
    Shift + F2              # 下条件断点
```

```
    Ctrl+s                  # 搜索汇编代码
    Ctrl+L                  # 搜索汇编代码时,下一条搜索项

'左下角命令'
    db 0x123444                 /* 以1字节的形式查看地址 0x****里的数据 */
    dd 0x213213                 /* 以4字节的形式查看地址0x***里的数据 */
    dp MessageBoxA              /* 打断点 */
```

IDA

```
"IDA快捷键"
    x                           /* 交叉引用 */
```

```
-dCULYURE=all                       /* IDA搜索中文字符串 */
```

网络攻防

踩点

通过搜索引擎

```
// 通过搜索引擎(爬虫)爬取网站,搜集网站信息,  Google爬虫NB

    baidu.com inurl:login

    site:baidu.com "login"
```

kail Linux

```
# DNS解析器抓取域名信息
    $ dnsenum www.baidu.com

    $ nslookup baidu,com

    $ whois www.baidu.com
```

```
# 探测网络包获取瓦米海洛拓扑结构
    $ traceroute 192.168.0.1
```

查点

Kail Linux

> 漏洞查询平台:`cnvd`

```
# telnet 抓取ssh旗帜(查看端口是否开启)
    $ telnet 127.0.0.1 22

    $ telnet www.baidu.com 80
         GET / HTTP/1.1          #获取80端口网页信息

# ping
    $ ping -c 3 127.0.0.1       # 设置 ping 次数
    $ ping -I etch0 127.0.0.1   # 指定网卡法宝

# arping 需主机和靶机需在同一子网内
    $ arping 127.0.0.1

# fping 同时ping多个ip
    fping -g 192.168.36.0/24            # 查看同一局域网下0-24ip池的主机
```