C 语言的深度挖掘 (三)

函数指针
回调函数
动态链接库
控制翻转(IOC)与好莱坞模式

西安电子科技大学计算机学院 李龙海

指向函数的指针

- 在C++中，一个函数总是占用一段连续的内存区，而函数名就是该函数所占内存区的首地址。我们可以把函数的这个首地址(或称入口地址)赋予一个指针变量，使该指针变量指向该函数。然后通过指针变量就可以找到并调用这个函数。我们把这种指向函数的指针变量称为“函数指针”。
- 函数指针的定义格式：<返回类型> (*<指针变量>)(<形参表>)。

```
/*fp可以指向返回值类型为double,
有一个int型参数的任何函数。*/
double  (*fp1) (int);
int*  (*fp2)(char[], int);
double  f(int x) {
   ……
}
int* g(char *s, int len) {
  ……
}
```

```
int main() {
    fp1 = f;   // 或为： fp = &f
    double d = fp1(5);
    fp2 = g;
    fp1 = g;   //Error!
    fp2 = f;   //Error!
    ……
}
```

指向函数的指针

- 也可以用typedef为函数指针类型取一个名字，然后再用该函数指针类型来定义指针变量：

  typedef <返回类型> (*<函数指针类型名>)(<形参表>)。

```
typedef double (*FP) (int);
FP fp;
```

- 可以通过一个函数指针来调用它所指向的函数，调用格式为：(*<函数指针变量>) (实参表)；或者 函数指针变量(实参表)；
- 注意不要将函数指针与返回指针的函数搞混了。

```
int *f(int, char*);
int (*f)(int, char*);
```

向函数传递函数

- C++中允许在调用一个函数时把一个函数作为参数传给被调用函数，这时，被调用函数的形参定义为一个函数指针类型，调用时的实参为一个函数的地址。

```
double integrate(double (*f)(double), double a, double b)
{
    …计算函数f在区间[a,b]上的定积分
}

integrate(sin, 0, 1);
Integrate(cos, 1, 2);
```

```
void swap(void *v[], int i, int j) {
    void *temp = v[i];
    v[i] = v[j];
    v[j] = temp;
}
void bubbleSort( void *v[], int len, int (*comp)(void *, void *)) {
    for( int i=0; i<len-1; i++) {
        for(int j=0; j<len-i-1; j++) {
            if( comp(v[j], v[j+1])>0 )
                swap(v, j, j+1);
        }
    }
}
int intcomp(void *a, void *b) {
    return *(int*)a>*(int*)b ? 1 : 0;
}
int floatcomp(void *a, void *b) {
    return *(float*)a>*(float*)b ? 1 : 0;
}
```

```
void main()
{
    void *v[5] = {0};
    printf("Please input 5 numbers:");
    for(int i=0; i<5; i++)
    {
        v[i] = malloc(sizeof(int));
        scanf("%d", (int*)(v[i]));
    }
    bubbleSort(v,5,intcomp);
    for(i=0; i<5; i++)
    {
        printf("%d\n", *(int*)(v[i]));
        free(v[i]);
    }
    return;
}
```

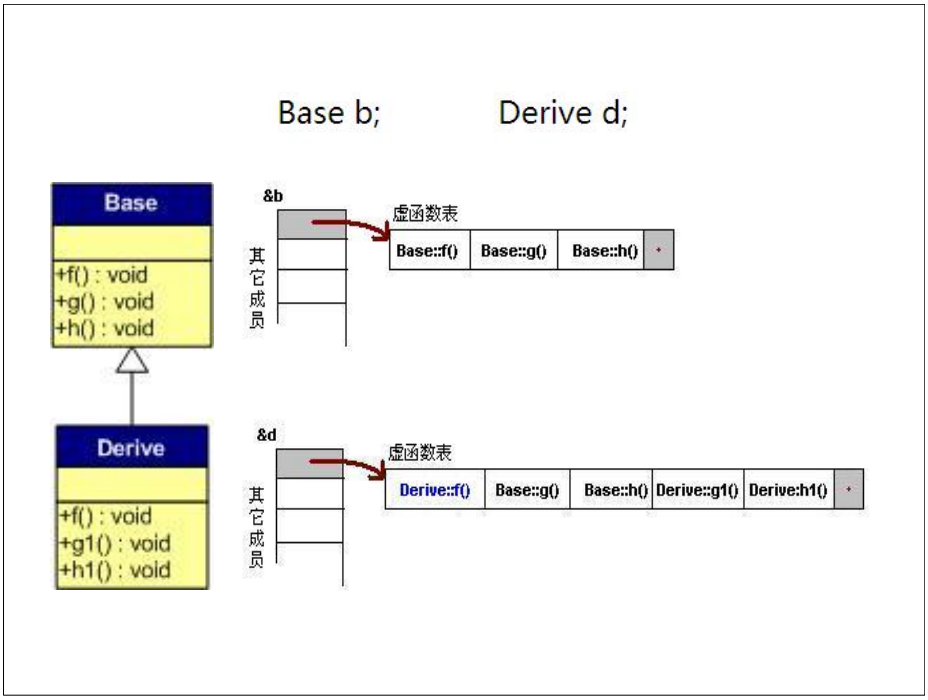
动态绑定

- 在高级语言中，把函数（或过程）调用与响应调用所需要的代码相关联的过程称为绑定。 静态绑定发生在编译期，每次运行调用的函数代码保持不变。动态绑定发生在运行期，实际调用的函数代码根据运行条件会发生改变。
- C/C++中动态绑定本质上都是通过函数指针实现。

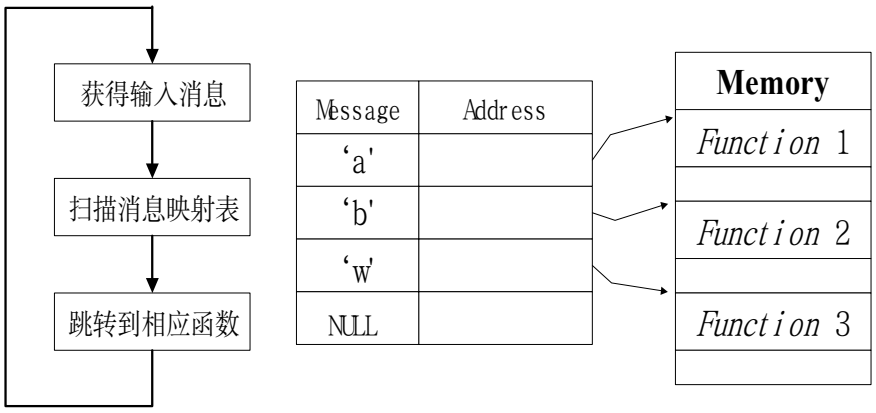
```
void caller(void(*ptr)())
{
    ptr();
}
void func()
{
    printf("Hello World!\n");
}
void main()
{
    void (*p)();
    p = func;
    caller(p);
}
```

C++虚拟函数与虚拟函数表

```
class Base {
public:
   virtual void f() { cout << "Base::f" << endl; }
   virtual void g() { cout << "Base::g" << endl; }
   virtual void h() { cout << "Base::h" << endl; }
};
class Derive : public Base {
public:
   virtual void f() { cout << "Derive::f" << endl; }
};

void fun(Base *p) {
   p->f();
   p->g();
}
```


Base b;
Derive d;
Base
+f() : void
+g() : void
+h() : void
Derive
+f() : void
+g1() : void
+h1() : void
&b
虚函数表
其它成员
Base::f()
Base::g()
Base::h()
&d
虚函数表
其它成员
Derive::f()
Base::g()
Base::h()
Derive::g1()
Derive::h1()

消息映射表的实现


获得输入消息
扫描消息映射表
跳转到相应函数
Message
Address
'a'
'b'
'w'
NULL
Memory
Function 1
Function 2
Function 3

消息映射表的实现

```
typedef void (*MsgHandler)(void);

struct MessageMapEntry {
    char ch;
    MsgHandler handler;
};

void fA() { printf("Press a\n"); }
void fB() { printf("Press b\n"); }
void fW() { printf("Press w\n"); }

MessageMapEntry MessageMap[100] = {
    {'a', fA},
    {'b', fB},
    {'w', fW},
    {0, NULL}
};
```

消息映射表的实现

```
int main()
{
    char ch;
    printf("Press a key:\n");
    while((ch=getchar())!='x')
    {
        int i = 0;
        while(MessageMap[i].ch)
        {
            if(MessageMap[i].ch==ch) {
                MessageMap[i].handler();
                break;
            }
            i++;
        }
    }
    return 0;
}
```

回调函数

- ***回调函数***是由程序员自己定义的但不是由自己显式调用的函数，其调用者往往是框架、容器、服务器、操作系统等，当然也可以是自己的程序。程序员往往将回调函数的地址传递给调用者从而实现调用。

```
void caller(void(*ptr)())
{
    ptr();
}
void func()
{
    printf("Hello World!\n");
}
void main()
{
    void (*p)();
    p = func;
    caller(p);
}
```

回调函数例一：Linux中的信号机制

- Linux信号机制是在应用软件层次上对中断机制的一种模拟，是一种异步通信方式
- 每个进程都有一个自己私有的信号处理函数映射表，当该进程成收到一个信号时，对应的信号处理函数被触发执行。
- 一个进程可以向另外一个进程发送信号，也可以向自己发送信号；操作系统内核也可以向一个进程发送信号，以通知某些硬件事件。
- 信号处理函数映射表中共有64个表项。前32个信号，编号为1~31，有预定义的含义和处理函数；后32个作为扩充。
- 前32个是不可靠信号(非实时的)，后32个为可靠信号(实时信号)。不可靠信号和可靠信号的区别在于前者不支持排队，可能会造成信号丢失，而后者不会。

回调函数例一：Linux中的信号机制

```
$ 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
```

回调函数例一：Linux中的信号机制

- ✔ 如何向一个进程发送信号？

所需头文件	#include <signal.h> #include <sys/types.h>	
函数原型	int kill(pid_t pid, int sig)	
函数传入值	pid：	正数：要发送信号的进程号
		0：信号被发送到所有和当前进程在同一个进程组的进程
		-1：信号发给所有的进程表中的进程（除了进程号最大的进程外）
		<-1：信号发送给进程组号为-pid 的每一个进程
	sig：信号	
函数返回值	成功：0	
	出错：-1	

回调函数例一：Linux中的信号机制

- 如何设置信号关联动作?

所需头文件	#include <signal.h>	
函数原型	typedef void (*sighandler_t)(int); sighandler_t signal(int signum, sighandler_t handler);	
函数传入值	signum：指定信号代码	
	handler：	SIG_IGN：忽略该信号
		SIG_DFL：采用系统默认方式处理信号
		自定义的信号处理函数指针
函数返回值	成功：以前的信号处理配置	
	出错：-1	

回调例二：DOS中设置中断处理函数

```
//  dos.h
//  返回中断号为intr_num的中断处理程序
void (* getvect(int intr_num))();
//  设置中断号为intr_num的中断处理程序为isr
void  setvect(int intr_num,void (* isr)());

void (*old_interrupt)();
void new_interrupt()
{
    // do something
    old_interrupt();
    // do something
}
int main()
{
    old_interrupt = getvect(10);
    setvect(10, new_interrupt);
    /*  将自己驻留在内存中  */
}
```

回调例三：在Linux中创建线程

1. **一个进程包含多个线程**
2. **多个线程共享所属进程的资源：**
   - 内存空间
   - 打开的文件
   - 信号
   - 安全权限
   - …
3. **为每个线程在共享的内存空间中都开辟了一个独立的栈空间（局部变量、函数形参等），拥有自己独立的寄存器环境，拥有独立的线程局部存储空间(TLS)**
4. **GunLibC中的pthread库实现了多线程API。**
5. **因为pthread并非Linux系统的默认库，编译时必须加上-lpthread参数**

回调例三：在Linux中创建线程

1. 线程标识：一个pthread_t 类型的变量
2. 线程属性：一个pthread_attr_t 类型的结构体
3. 创建线程：

   int pthread_create(pthread_t* tid, pthread_attr_t *attr, void*(*start_routine)(void*), void *arg);

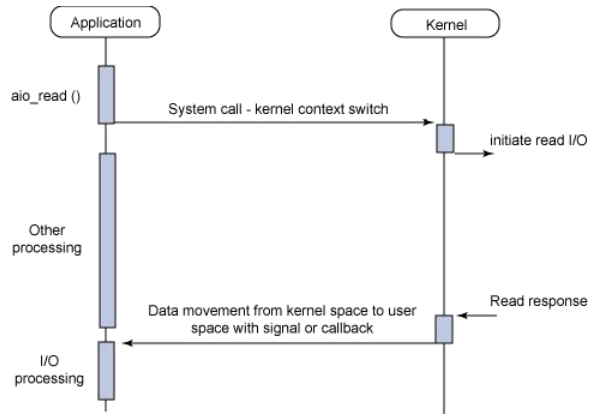
   - tid：输出参数，用于返回所创建现成的标识；
   - attr：用于设定线程属性；
   - start_routine：用于指定线程主函数
   - arg：为线程主函数传递的参数

回调例四：在Windows中创建线程

```
typedef struct _MyData {
    int val1;
    int val2;
} MYDATA, *PMYDATA;
//子线程函数
DWORD WINAPI ThreadProc( LPVOID lpParam ) {
    PMYDATA pData = (PMYDATA)lpParam;
    /* do something;  */
    return 0;
}
void main()  // 主线程
{
    MYDATA mydata;
    mydata.val1 = 3, mydata.val2 = 5;
    CreateThread(NULL, 0, ThreadProc, &mydata, 0, NULL);
    /* do something;  */
}
```

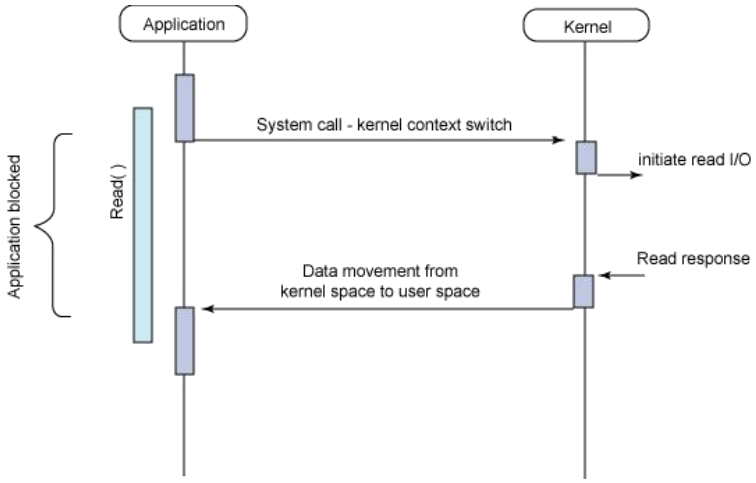
例五：Linux中异步文件IO

1. 进程发起文件IO请求之后不用等待IO的动作的完成，可以继续做其他事情。
2. 内核在完成该进程的IO请求之后，用信号机制或回调函数机制通知该进程。



例五：Linux中异步文件IO

1. 下面是同步IO的时序图。



例五：Linux中异步文件IO

1. int aio_read(struct aiocb *aiocbp);
   aio_read函数在请求进行排队之后会立即返回，执行成功返回0，出错返回-1，并设置errno的值
2. 其中，struct aiocb主要包含以下字段：

```
int        aio_fildes;     /* 要被读写的fd */
void *     aio_buf;        /* 读写操作对应的内存buffer */
__off64_t  aio_offset;     /* 读写操作对应的文件偏移 */
size_t     aio_nbytes;     /* 需要读写的字节长度 */
int        aio_reqprio;    /* 请求的优先级 */
struct sigevent aio_sigevent;   /* 异步事件，定义异步操作完成时的通知信号或回调函数 */
```

例五：Linux中异步文件IO

```
 struct sigevent
{
        int  sigev_notify; //notification type
        int  sigev_signo; //signal number
        union  sigval   sigev_value; //signal value
        void  (*sigev_notify_function)(union sigval);
          /*  找到你了！传说中的回调函数！  */
        pthread_attr_t *sigev_notify_attributes;
}
union sigval
{
        int sival_int; //integer value
        void *sival_ptr; //pointer value
}
```

例六：在Windows中设置消息钩子

```
typedef LRESULT (CALLBACK *HOOKPROC)(int,WPARAM,LPARAM);
void main()
{
    HMODULE hDll=LoadLibrary("MyHook.dll");
    HOOKPROC lpfn = (HOOKPROC)GetProcAddress(hDll, "HookFun");
    HHOOK hook=SetWindowsHookEx(WH_SYSMSGFILTER,lpfn,hDll,0);
    /*  驻留内存，监视 */
    UnhookWindowsHookEx(hook);
}
/*==========传说中的分隔线==========*/
/* MyHook.dll */
LRESULT CALLBACK HookFun(int nCode,WPARAM wParam,LPARAM lParam)
{
    /* do something  */
    return CallNextHookEx(nCode,wParam,lParam);
}
```

Windows API中的其它例子

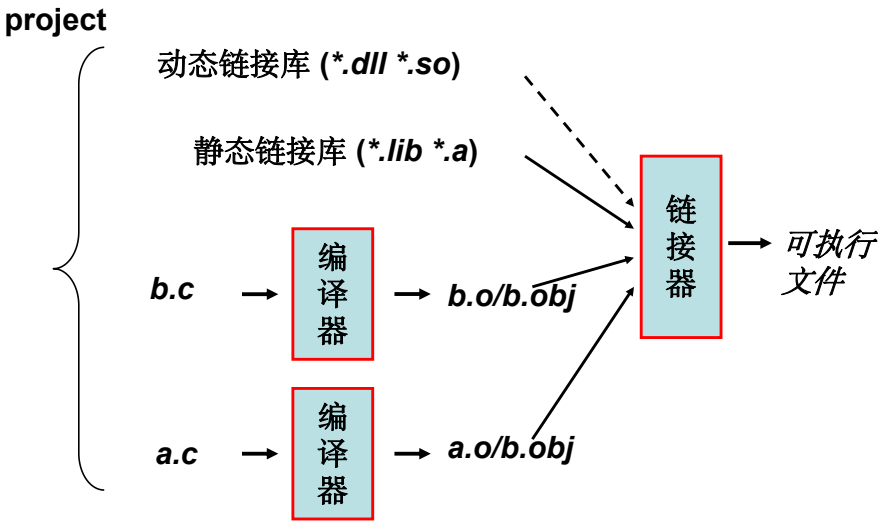
```
BOOL ReadFileEx(
    HANDLE hFile,                // handle to file
    LPVOID lpBuffer,             // data buffer
    DWORD nNumberOfBytesToRead,  // number of bytes to read
    LPOVERLAPPED lpOverlapped,
    LPOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine
);


BOOL EnumWindows(
    WNDENUMPROC lpEnumFunc, // callback function
    LPARAM lParam           // application-defined value
);
BOOL CALLBACK EnumWindowsProc(
    HWND hwnd,      // handle to parent window
    LPARAM lParam   // application-defined value
);
```

分别编译与链接（Linking）

大多数高级语言都支持分别编译，程序员可以显式地把程序划分为独立的模块或文件，然后每个独立部分分别编译。在编译之后，由链接器把这些独立的片段（称为**编译单元**）“粘接到一起”。（想想这样做有什么好处？）

在C/C++中，这些独立的编译单元包括obj文件（一般的源程序编译而成，.o文件）、静态链接库文件（.lib或.a文件）、动态链接库文件（.dll或.so文件）等。

分别编译与链接



链接器的主要工作

1. 将分散的数据和机器代码**收集并合成**一个单一的可加载并可执行的文件；
2. **符号解析**：由多个程序模块(源程序)构建一个可执行程序时，模块之间的相互引用通过符号进行。程序也可以通过符号来引用代码库(lib库)中的功能。符号解析就是将符号引用和符号定义关联起来。
3. **地址重定位**：编译器产生的各个目标文件(obj文件)中数据和代码的地址一般都是从0开始。因此如果一个程序包含多个目标文件时就会产生地址重叠。重定位就是为每个目标文件重新定义加载地址，并修改相应的代码和数据以反映这种变化。

静态链接与动态链接

- ✔ **静态链接方式**：在程序执行之前完成所有的组装工作，生成一个可执行的目标文件（如Windows下的EXE文件）。
- ✔ **动态链接方式**：在程序已经为了执行被装入内存之后完成链接工作，并且在内存中一般只保留该编译单元的一份拷贝。
- ✔ 将函数名称和函数执行体动态链接（或动态绑定）的过程既可以发生在程序装载时，也可以发生在程序运行时（在需要时才会引入函数的方式）。

预编译函数库：静态库、动态库

- 预编译的函数库：模块化、可重用性强、编译速度快、保护知识产权
- 静态库（.a后缀）是一系列的目标文件的归档，静态链接时，静态库中的目标函数体会被复制到程序的可执行文件中去
- 动态库（libname.so[.major.minor.release]）不具有“复制”操作，程序运行时根据需要载入内存，且动态地将函数调用与函数体关联到一起。动态库在内存中只有一份拷贝，因此也称为共享库。易于升级。

静态链接库与动态链接库

- 可以将静态链接库或动态链接库看成是一种仓库，它提供给你一些已经编译成机器代码的可以直接拿来用的数据、函数或类，它们是实现代码共享的一种方式。
- 静态链接库中的机器代码和数据都被直接包含在最终生成的EXE文件中
- 动态链接库的内容不必被包含在最终的EXE文件中，EXE文件执行时可以“动态”地引用和卸载这个与EXE独立的DLL文件。

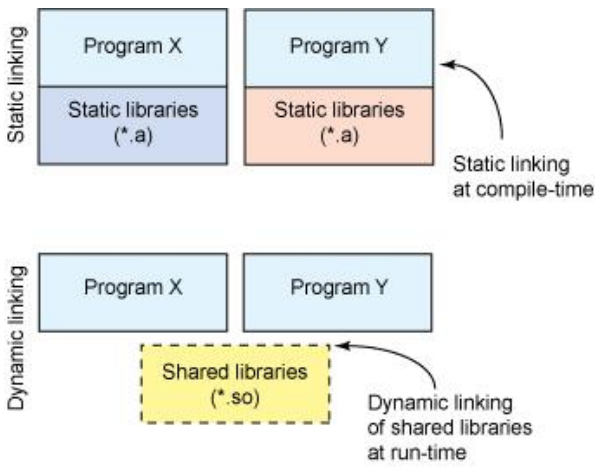
动态链接库

1. 动态链接库（Dynamic Link Library）是一个可以被其它程序共享的程序模块，其中封装了一些可以被共享的数据、函数和资源。
2. 如果一个可执行程序使用了一个DLL，当可执行程序运行时，操作系统会把DLL加载到内存，并解析可执行程序对该DLL的符号引用，使得可执行程序能够调用DLL中的函数功能。
3. 扩展名一般是dll（so），也有可能是fon、ocx、drv、sys（ko）等。DLL中虽然包含了可执行代码却不能单独执行，而应由其他应用程序直接或间接调用。

使用动态链接库的优点

1. DLL文件与可执行文件独立，只要输出接口不变，更换DLL文件不会对EXE文件造成任何影响，因而极大地提高了可维护性和可扩展性。
2. 被多个应用程序共享时，在内存中只有一份拷贝，因而更加节省内存
3. 可以在多种编程语言之间共享代码

静态库与动态库



Linux中如何使用静态库

```
// hello.h
#ifndef HELLO_H
#define HELLO_H
void hello(char *name);
#endif
// hello.c
#include <stdio.h>
void hello(char *name) {
        printf("Hello %s!\n", name);
}
```

Linux中如何使用静态库

如何生成静态库：

- gcc -c hello.c
- ar -crv libmyhello.a hello.o

如何链接静态库

- gcc -o test main.c –lmyhello
- gcc -o test main.c -L. –lmyhello
- gcc main.c libmyhello.a -o test

VC6.0中使用静态lib库的三种方法

1. 利用编译器指令#pragma comment(lib , …)
2. 将lib库文件所在目录设置在VC环境中
3. 将lib库文件设置在工程中。

Linux中如何使用动态库

如何生成动态库

- gcc -fPIC -c hello.c (生成hello.o文件)
- gcc -shared -o libmyhello.so hello.o (生成hello.so)

如何链接动态库（加载时链接）

- gcc -o test main.c –lmyhello

如何找到动态库

- /usr/lib和/lib目录
- 环境变量： LD_LIBRARY_PATH
- 配置文件/etc/ld.so.conf

VC6中动态链接库的创建

1. **第一步**：通过编译开关(编译参数)将编译器和链接器设置为输出DLL状态；或者在VC6中用向导创建一个“Win32 Dynamic Link Library”工程
2. **第二步**：将程序中的一些函数设置为**导出函数**，设置导出函数的方法有两种：
   - 利用**.DEF**文件
   - 利用VC扩展关键字**__declspec(dllexport)**
3. VC在生成DLL的同时还会产生一个**导入库(import lib)**，该静态库中列出了DLL输出的所有函数和变量的名称，但不包含任何实现代码。
4. 在生成的DLL文件头部，还包含一个**输出符号表**，其中记录了该DLL输出的所有函数和变量的名称及相对偏移。

利用__declspec(dllexport)导出函数

1. 在函数定义或函数声明的最前面加上关键字__declspec(dllexport)即可将该函数设置为导出函数
2. 为了键入方便，一般定义一个宏：

```
# define EXPORT  extern "C" __declspec(dllexport)
EXPORT int add( int x, int y)
{
    return x + y;
}
```

利用.DEF文件导出函数

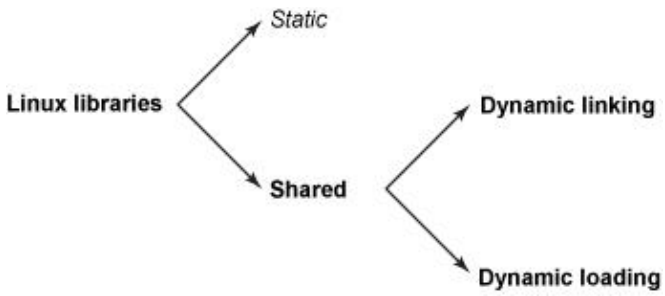
1. 在“Win32 Dynamic Link Library”工程中添加一个后缀名为.def的文本文件，该文件一般采用如下格式：

```
LIBRARY   <DLL文件名>
DESCRIPTION “用途描述”
EXPORTS
 funcionA   @1
 funcionB   @2
 funcionC   @3
```

2. 利用DEF文件导出函数的一个优点是将来可以按序号查找该函数，比一般的按名字查找更高效。

动态链接库的两种链接方法

1. 装载时动态链接(Load-time Dynamic Linking)
2. 运行时动态装载并链接(Run-time Dynamic Loading)



Linux中动态加载动态库

- void *dlopen(const char *file, int mode);
- 功能：打开指定的动态链接库文件，并将它读入内存，返回一个句柄给调用进程。
- 参数：
  - file: 库文件名称
  - mode: 符号解析时间、作用范围
    - RTLD_LAZY: 暂缓决定，等有需要时再解析出符号
    - RTLD_NOW: 立即决定，返回前解出所有符号
    - RTLD_GLOBAL：动态库中定义的符号可被其后打开的其他库重定位
    - RTLD_LOCAL：与RTLD_GLOBAL作用相反，动态库中定义的符号不能被其后打开的其它库重定位，缺省RTLD_LOCAL

Linux中动态加载动态库

- void *dlopen(const char *file, int mode);
- 功能：打开指定的动态链接库文件，并将它读入内存，返回一个句柄给调用进程。
- 参数：
  - file: 库文件名称
  - mode: 符号解析时间、作用范围
    - RTLD_LAZY: 暂缓决定，等有需要时再解析出符号
    - RTLD_NOW: 立即决定，返回前解出所有符号
    - RTLD_GLOBAL：动态库中定义的符号可被其后打开的其他库重定位
    - RTLD_LOCAL：与RTLD_GLOBAL作用相反，动态库中定义的符号不能被其后打开的其它库重定位，缺省RTLD_LOCAL

Linux中动态加载动态库

- void *dlsym(void *handle, const char *name)
- 功能：根据动态链接库操作句柄与符号，返回符号对应的地址，可以是函数也可以是变量。
- 参数：
  - handle : dlopen返回的动态链接库句柄
  - name :符号名称，可为函数名或变量名称

- int dlclose(void *handle)
- 功能：关闭指定句柄的动态链接库，只有当此动态链接库的使用计数为0时，才会真正被系统卸载

Linux中动态加载动态库的例子

```
#include <stdio.h>
int a = 12;
int hello(int x, int y)
{
    int tmp = x + y;
    printf("helloworld %d\n", tmp);
    return tmp;
}
```

- // gcc -c -fPIC test.c
- // gcc -shared -fPIC -o test.so test.o

Linux中动态加载动态库的例子

```
#include <dlfcn.h>
#include <stdio.h>
typedef int (*hello_t)(int, int);
int main(){
        void * lib = dlopen("test.so", RTLD_LAZY);
        hello_t hello = (hello_t)dlsym(lib, "hello");
        int * a1 = (int *)dlsym(lib, "a");
        printf("%d\n", *a1);
        hello(3, 5);
        dlclose(lib);
}
```

- // gcc –o test dlopen.c -ldl

Windows中动态加载和链接***DLL***文件

1. 利用Windows API函数LoadLibrary, GetProcAddress和FreeLibrary实现运行动态加载、链接和释放DLL

```
// MyDll.cpp

extern"C" __declspec(dllexport) int add(int a, int b)
{
    return a + b;
}

//  CallMyDll.cpp

HINSTANCE hInstance=LoadLibrary("MyDll.dll");
typedef int (*AddProc)(int,int);
AddProc add=(AddProc)GetProcAddress(hInstance,"add");
cout<<"3+5="<<add(3,5)<<endl;
FreeLibrary(hInstance);
```

*LoadLibrary*查找*DLL*的路径

1. 应用程序被加载的目录
2. 当前子目录(默认子目录)(GetCurrentDirectory)
3. Winodws\System32子目录
4. Windows子目录
5. 环境变量PATH中标识的子目录

Windows中静态加载链接*DLL*

1. 静态方式的特点是由编译器利用导入库(.lib)将DLL的加载、链接和卸载代码直接添加到.exe文件中。
2. 采用静态加载方式的优点：
   - 调用程序更简单，易读
   - 运行效率高
3. 采用静态加载方式的缺点：
   - 不够灵活，无法选择加载时机，无法更换DLL文件

动态链接库的应用举例

1. 所有的Windows API函数都是以动态链接库导出函数形式提供的。大部分API函数都存放在kernel32.dll、user32.dll和gdi32.dll三个动态库中，相应的导入库为kernel32.lib、user32.lib和gdi32.lib
2. 应用软件的插件技术
3. 每个Windows驱动程序本质上都是动态链接库

Windows API

1. **Windows API** (Application Programming Interface)是Windows操作系统为应用程序设计提供的一组函数(过程)调用接口。应用程序通过调用这些函数就可以获得操作系统的底层服务，访问系统管理的各种软、硬件资源。
2. 所有的Windows API函数都是以动态链接库导出函数形式提供的。大部分API函数都存放在kernel32.dll、user32.dll和gdi32.dll三个动态库中，相应的导入库为kernel32.lib、user32.lib和gdi32.lib
3. Windows API函数的使用说明在MSDN中可以找到(http://www.microsoft.com/msdn)

基于DLL的消息映射

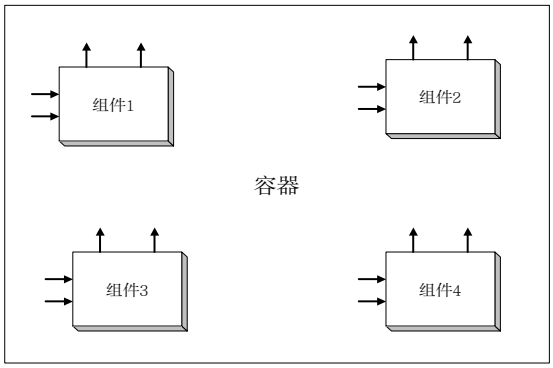
思考：

如何将前面讲述的消息映射表的例子改造成“热插拔”版本?

将消息映射表存储在配置文件中，将每个消息处理函数都放在一个独立的动态库中。当增加新的消息处理时，只需修改配置文件，增加动态库，根本不需要停止主程序的运行。

上述问题作为小作业1（Linux环境）。一个星期后（11月5日）提交至xdlilh@gmail.com，注明姓名学号!

容器（框架）与组件



在DLL消息映射表的例子中，主程序相当于容器，动态库相当于组件。

容器（框架）与组件

1. 容器实现了所有组件所需的通用服务，并且负责管理组件的生命周期。
2. 组件对容器的功能进行扩展，它是一种没有生命力的被动软件模块。
3. 组件是一组函数(子程序)的集合体，这些函数由容器在适当的时机调用，我们称这些函数为回调函数
4. 容器提供的通用服务也以函数调用的形式给出
5. 组件提供给容器的调用接口，以及容器提供给组件的调用接口必须事先约定好

好莱坞模式与IoC

1. 好莱坞模式（Hellywood Principle）：

   Don't call us, we'll call you!

2. 控制反转（Inversion of Control）：

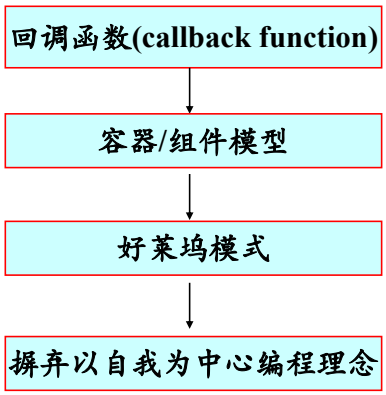
   第三方的基础软件库经常以两种方式提供给我们：

   （1）框架的形式：框架提供了通用流程，我们可以通过框架定义的规范(调用接口/交互协议)就可以把自己的代码(组件)植入到通用流程中，完成与具体应用相关的需求定制。

   （2）工具库(Library)形式：对于工具库，使用者必须撰写调用它们的逻辑，工具库完成相应的工作后会把控制权返回给调用者。

   在框架形式中，“主仆”关系发生了反转，控制发生了反转!

好莱坞模式与IoC



容器与组件的例子

1. Photoshop的外挂滤镜(扩展名为8bf，实质为动态库)存放在\Program Files\Photoshop 7.0\Plug-Ins目录下。 Photoshop是容器，滤镜是组件。
2. Acrobat PDF的插件(扩展名为api，实质为动态库)， 都存放在Acrobat安装目录下的Plug_ins文件夹中。
3. J2EE应用服务器是容器，EJB是组件。
4. MVC框架Struts中，使用者负责实现Action、Form、View等组件，而不用关心三者之间的具体协作关系。
5. 浏览器是容器，Applet组件。
6. Linux操作系统是容器，可加载内核模块（LKM）是组件。而驱动程序是一种特殊的LKM。