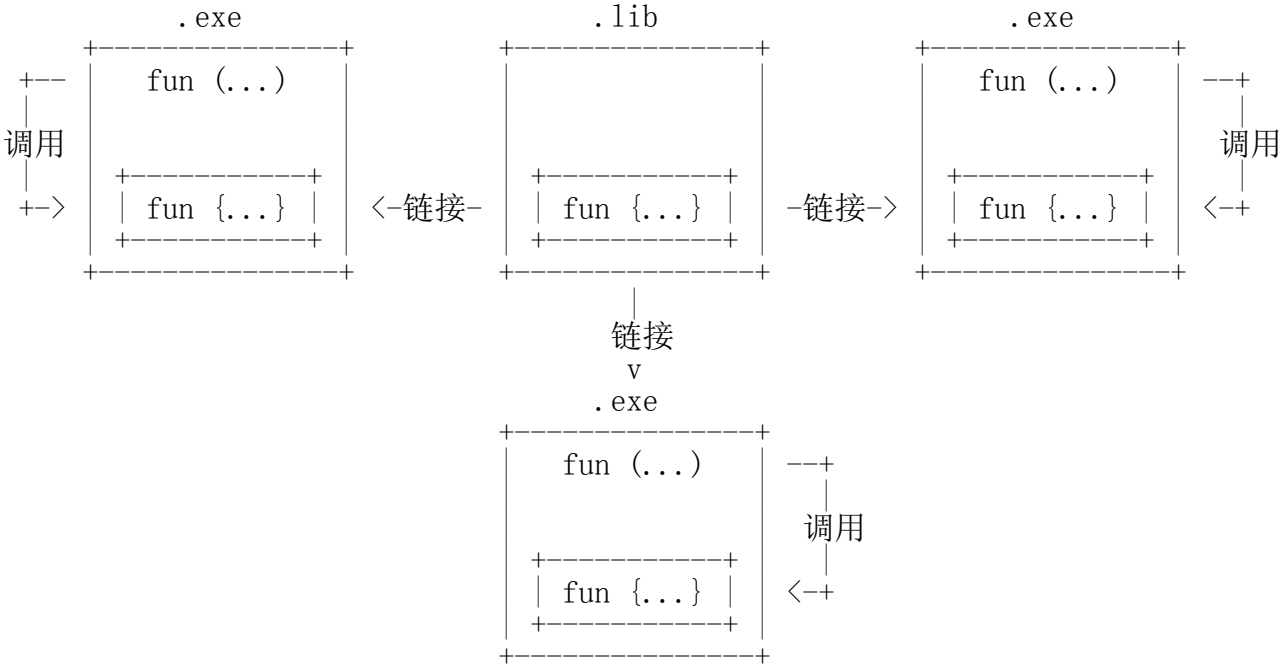
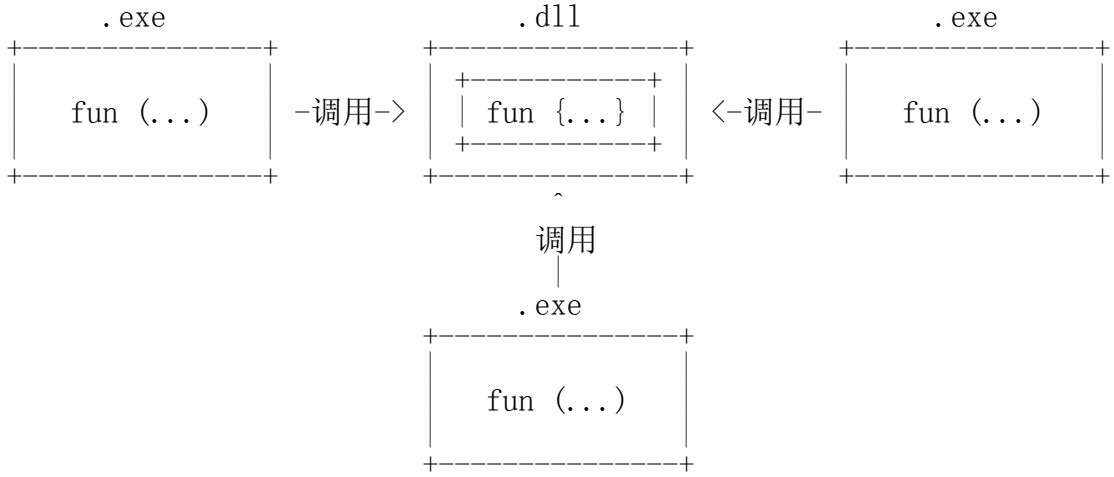
==========
第八课　库
==========

一、库的分类

静态库：库中代码被链接到可执行程序或动态库中，运行时不需要。
　　扩展名为.lib，目标文件(.obj)的归档



动态库：库中代码不被链接到可执行程序或其它动态库中，运行时需要
　　且为多个使用者共享。扩展名为.dll。



二、静态库
----

1. 使用静态库

在C/C++程序中可以直接使用C/C++标准库中的函数、类型、对象等，
不需要显式指明链接哪个静态库，但使用其它静态库，需要显式指明。

2. 创建静态库

1) 创建新项目
2) 添加源文件
3) 编写库代码

3. 链接静态库

1) 代码中加入#pragma comment (lib, "../Lib/CLib.lib")
2) 通过Project/Settings...设置，添加../Lib/CLib.lib

注意：默认情况下C++编译器会做函数换名，
可通过extern "C"抑制C++编译器做函数换名。

C   -> C
CPP -> CPP
CPP -> C
C   -> CPP

范例：CLib、UseCLib、CPPLib、UseCPPLib

三、动态库
---

1. 特点

1) 运行时独立存在。
2) 不会链接到可执行程序。
3) 使用时加载。

2. 与静态库比较

1) 由于静态库是将代码嵌入到使用程序中，多个程序使用时，
   会有多份代码，所以代码体积会增大。
   动态库代码只需要一份，
   其它程序通过函数地址以共享方式使用动态库中的代码，
   所以体积小。

2) 静态库发生变化后，新的代码需要重新链接(嵌入)到使用程序中。
   动态库发生变化后，只要函数的接口未发生变化，
   使用该动态库的程序无需重新链接。

3. 创建

1) 建立新项目
2) 添加源文件
3) 编写库代码

4. 导出

1) 声明导出：在函数声明前加_declspec (dllexport)前缀，
   若无声明则加在定义之前。

2) 模块定义文件导出：.def

```
LIBRARY CDll
EXPORTS
c_add @1
c_sub @2
```

动态库的导入库(Import Library)扩展名也是.lib，但其内容与静态库完全不同。
导入库中只存放函数名与函数序号(Ordinal)的映射表。
链接器在链接阶段，根据此映射表，为该动态库的使用者，
生成由函数序号定位函数入口(Entry Point)的代码。

3) 声明导出的函数序号由编译器确定，可能因为动态库的修改而发生变化。
   模块定义文件导出的函数序号，由程序员指定，不随动态库的改变而改变。

.def	.lib	.dll
c_add @100	c_add -> 100	100 -> 0x00026009
c_sub @200	c_sub -> 200	200 -> 0x0002511D

5. 静态加载

1) 链接导入库(.lib)，声明为导入_declspec (dllimport)。

2) 动态库需要放在如下位置：

A. 可执行程序所在目录   -- 推荐使用
B. 当前工程目录   -- 依赖于开发环境，调试时使用
C. WINDOWS目录   \
D. WINDOWS/system32目录   + 注意版本问题
E. WINDOWS/system目录   /
F. PATH环境变量指定的目录 -- 依赖于系统环境，不推荐使用

3) 可执行程序启动时加载动态库，退出时卸载动态库。

6. 动态加载

1) 定义函数指针类型：typedef ...

2) 加载动态库

```
HMODULE LoadLibrary (
    LPCTSTR lpFileName // 动态库文件名(按路径规则搜索)
                       // 或绝对/相对路径(按指定路径加载)
);
```

成功返回动态库实例句柄(HINSTANCE)，失败返回NULL。

3) 获取函数地址

```
FARPROC GetProcAddress (
    HMODULE hModule,    // 动态库实例句柄
    LPCSTR  lpProcName, // 函数名(注意C++换名问题)
);
```

成功返回函数地址，失败返回NULL。

4) 卸载动态库

```
BOOL FreeLibrary (
    HMODULE hModule // 动态库实例句柄
);
```

成功返回TRUE，失败返回FALSE。

5) 可执行程序调用LoadLibrary时加载动态库，
   调用FreeLibrary时卸载动态库。

范例：CDll、UseCDll、CPPDll、UseCPPDll、LoadDll

7. 加载方式与导出方式的对比

1) 模块定义文件(.def)可以为动态库中的函数指定序号，
   即便日后修改了动态库中的代码，导致其中函数的名称及入口地址发生了改变，
   但只要其序号保持不变，仍可以在不重新链接的情况下，被正确地调用。

范例：DefDll、UseDefDll

2) 动态加载不需要导入库。
   因此只要函数接口(函数名+形参表+返回类型)不变，
   动态库做任何修改都不需要重新链接。

3) 从动态库的使用灵活性方面看，
   动态加载优于静态加载(LoadLibrary > .lib)，
   模块定义文件导出优于声明导出(.def > dllexport)。

4) 若被静态加载的动态库不存在，则程序无法启动，
   而动态加载只有在LoadLibrary该库时才会报错。

8. 导出类

编库时声明为导出，用库时声明为导入。

```
#ifdef CPPDLL_EXPORTS
#define CPPDLL_API _declspec (dllexport)
#else
#define CPPDLL_API _declspec (dllimport)
#endif /* CPPDLL_EXPORTS */
```

```
class CPPDLL_API CMath
{
    ...
};
```

范例：CPPDll、UseCPPDll

9. 入口函数

入口函数不是动态库所必须的，常用于动态库内部的初始化和善后处理。

```
BOOL WINAPI DllMain (
    HANDLE hinstDLL,    // 动态库实例句柄
    DWORD  dwReason,    // 被调用的原因
    LPVOID lpvReserved // 保留
);
```

返回TRUE表示动态库加载成功，FALSE表示失败。

动态库加载/卸载时会被调用。如：LoadLibrary/FreeLibrary。

dwReason取值：

```
    DLL_PROCESS_ATTACH - 进程加载，在主线程中调用LoadLibrary
    DLL_PROCESS_DETACH - 进程卸载，在主线程中调用FreeLibrary
    DLL_THREAD_ATTACH  - 线程加载，在子线程中调用LoadLibrary
    DLL_THREAD_DETACH  - 线程卸载，在子线程中调用FreeLibrary
```

范例：WinDll、WinUseDll