第3章 软件体系结构风格

内容

- 3.1 概述
- 3.2 数据流风格
- 3.3 过程调用风格
- 3.4 独立构件风格
- 3.5 层次风格
- 3.6 虚拟机风格
- 3.7 客户/服务器风格
- 3.8 表示分离风格
- 3.9 插件风格
- 3.8 微内核风格
- 3.9 SOA风格

插件风格

- 3.9.1 插件技术背景
- 3.9.2 插件机制
- 3.9.3 插件技术基础——动态链接库
- 3.9.4 插件案例分析

3.9.1 插件技术背景

插件定义

- 在计算机中，插件（plug-in，或plugin，extension，或add-on/addon）是一种软件组件，它能够添加特定的功能到现有的软件应用程序中。
- 插件是增强应用程序的常见方式。
- 常见的例子是用于在Web浏览器中添加新功能的插件，如Adobe Flash Player，QuickTime播放器和Java插件（在Web页面上启动一个由用户激活的Java applet）
- Photoshop、SketchUp、3DMax等工具，都支持插件开发

使用插件的原因

- to enable third-party developers to create capabilities to extend an application
- to support features yet unforeseen
- to reduce the size of an application
- to separate source code from an application because of incompatible software licenses

插件 vs. 扩展
（Plug-in and extension）

Plug-ins differ from extensions, which modify or add to existing functionality.

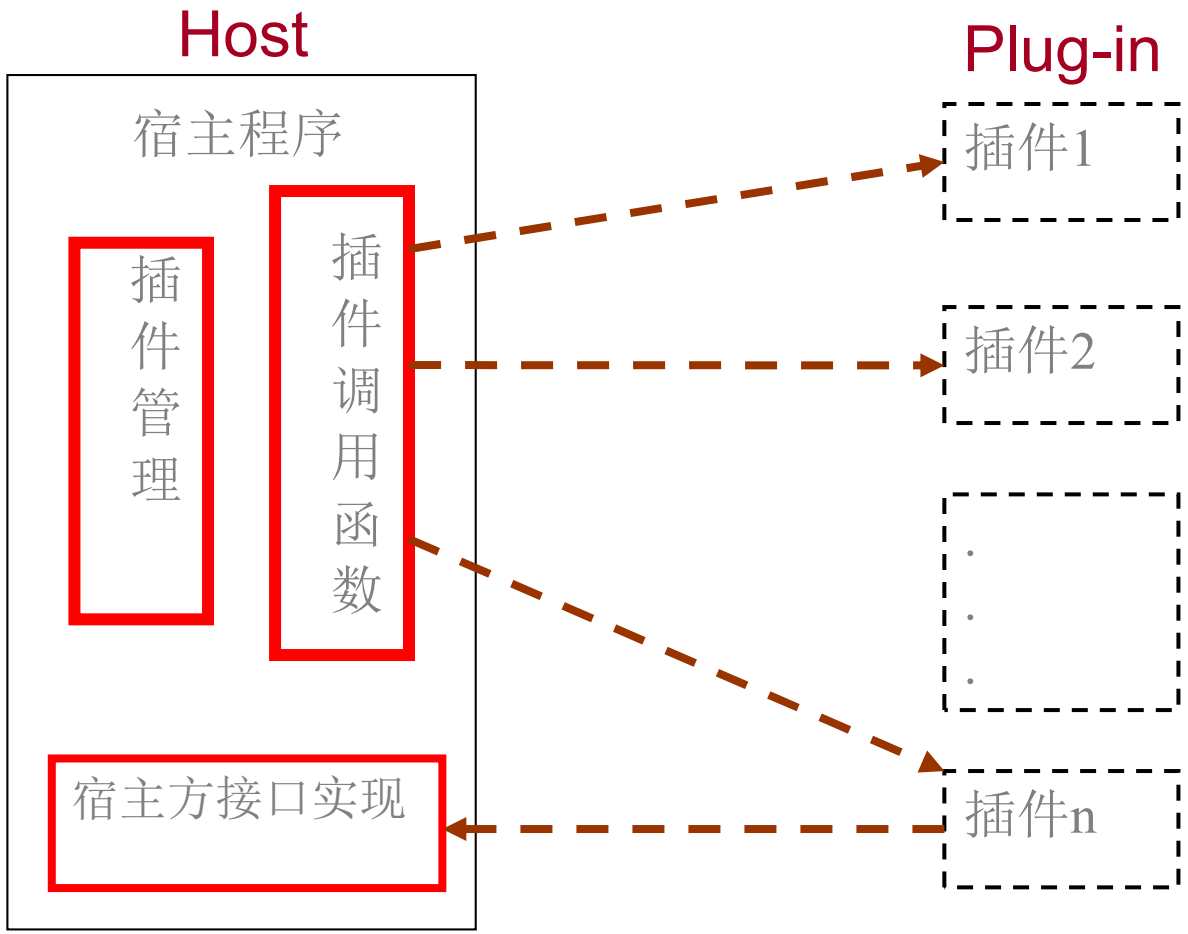
- Plug-ins rely on the host application's user interface and have a well-defined boundary to their possible set of actions.
- Extensions have fewer restrictions on their actions, and may provide their own user-interfaces.

3.9.2 插件机制

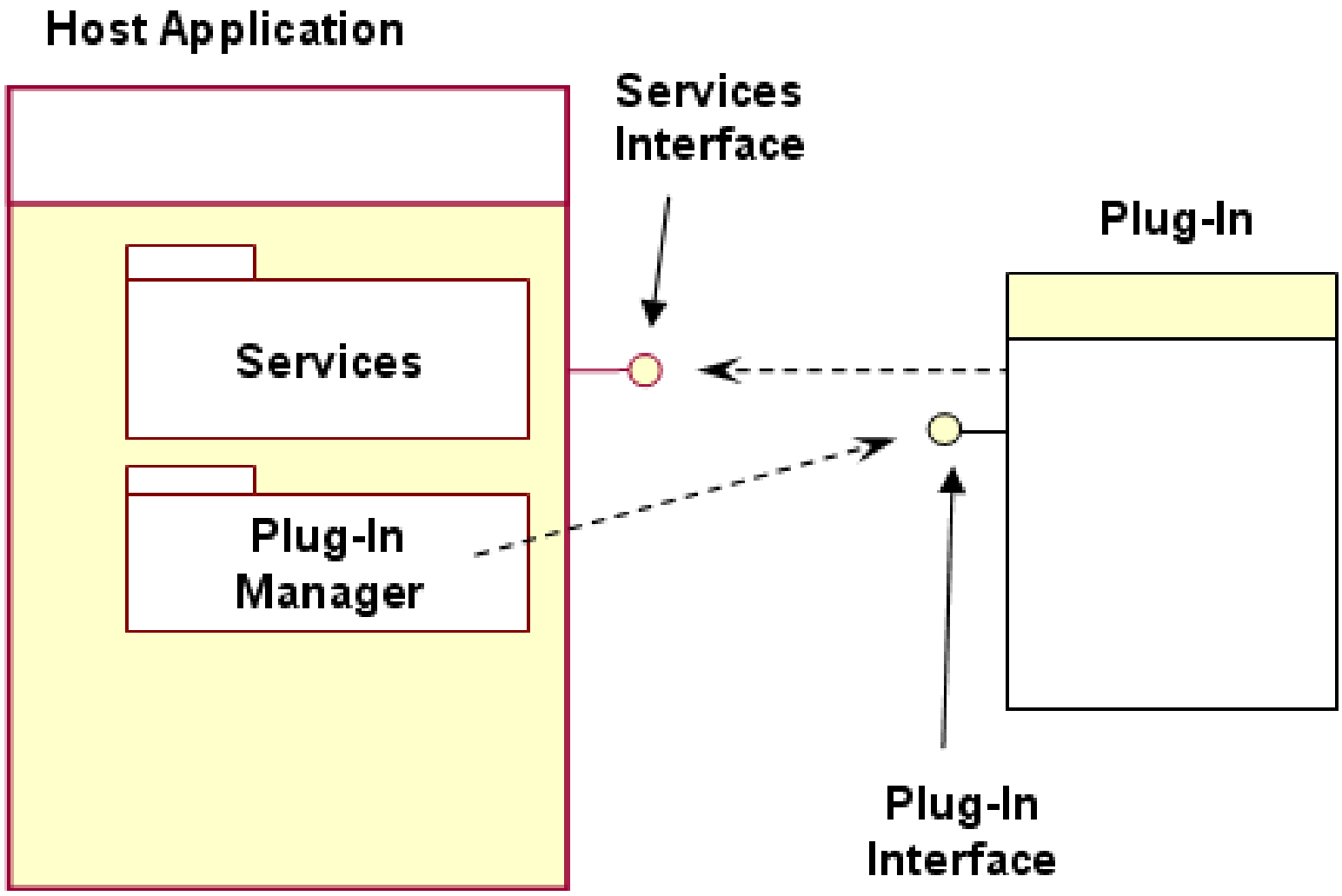
插件技术

- 所谓插件技术，就是在程序的设计开发过程中，把整个应用程序分成宿主程序(host)和插件(plug-in)两个部分。宿主程序与插件能够互相通信，而且在宿主程序不变的情况下，可以通过增减或修改插件来调整应用程序的功能。

插件系统体系结构



插件系统构件


Host Application
Services Interface
Plug-In
Services
Plug-In Manager
Plug-In Interface

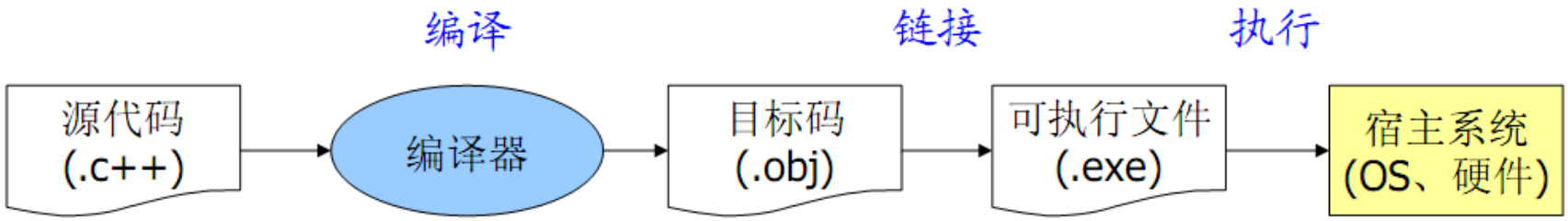
- The **host application** provides services which the plug-in can use, including a way for plug-ins to register themselves with the host application and a protocol for the exchange of data with plug-ins.
- **Plug-ins** depend on the services provided by the host application and do not usually work by themselves.

Conversely, the host application operates independently of the plug-ins, making it possible for end-users to add and update plug-ins dynamically without needing to make changes to the host application.

3.9.3 插件技术基础——动态链接库 (Dynamic Linkable Library)

回顾：Compiler (编译器)

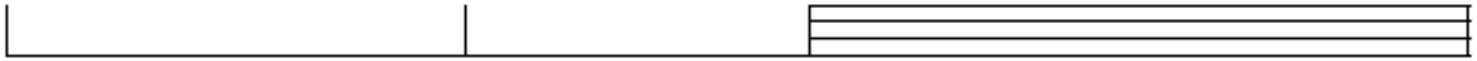
- This is in contrast to a compiler which does not execute its input program (the source code) but translates it into another language, usually executable machine code (also called object code) which is output to a file for later execution. (编译器不会执行输入的源程序代码，而是将其翻译为另一种语言，通常是可执行的机器码或目标码，并输出到文件中以便随后链接为可执行文件并加以执行)



回顾：Interpreter versus compiler (解释器和编译器)

编译器

编译　　链接　　执行

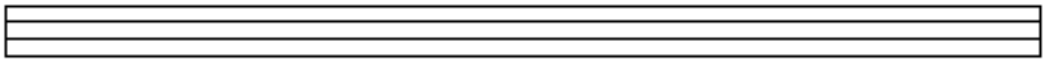


分析程序结构
源代码→目标码
确定内存分配与访问方案

分配内存
执行目标码(机器码)

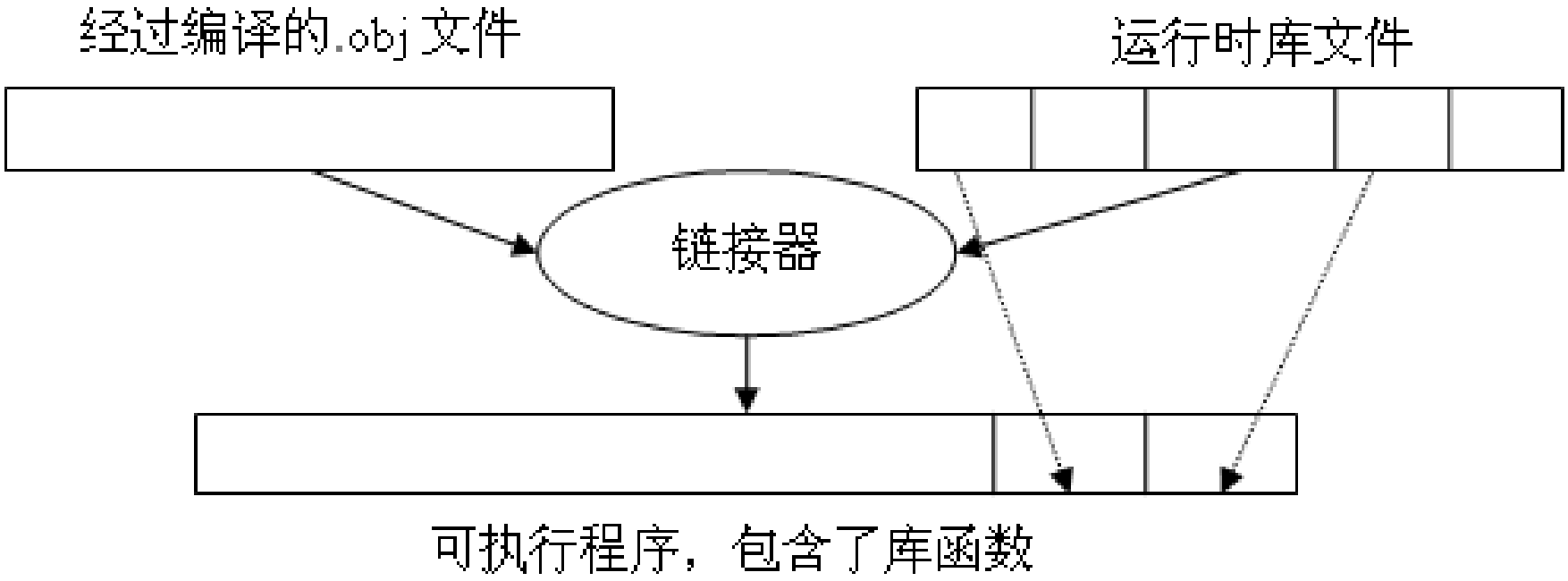
解释器

解释执行



分析程序结构
确定内存分配与访问方案
分配内存
解释并执行程序

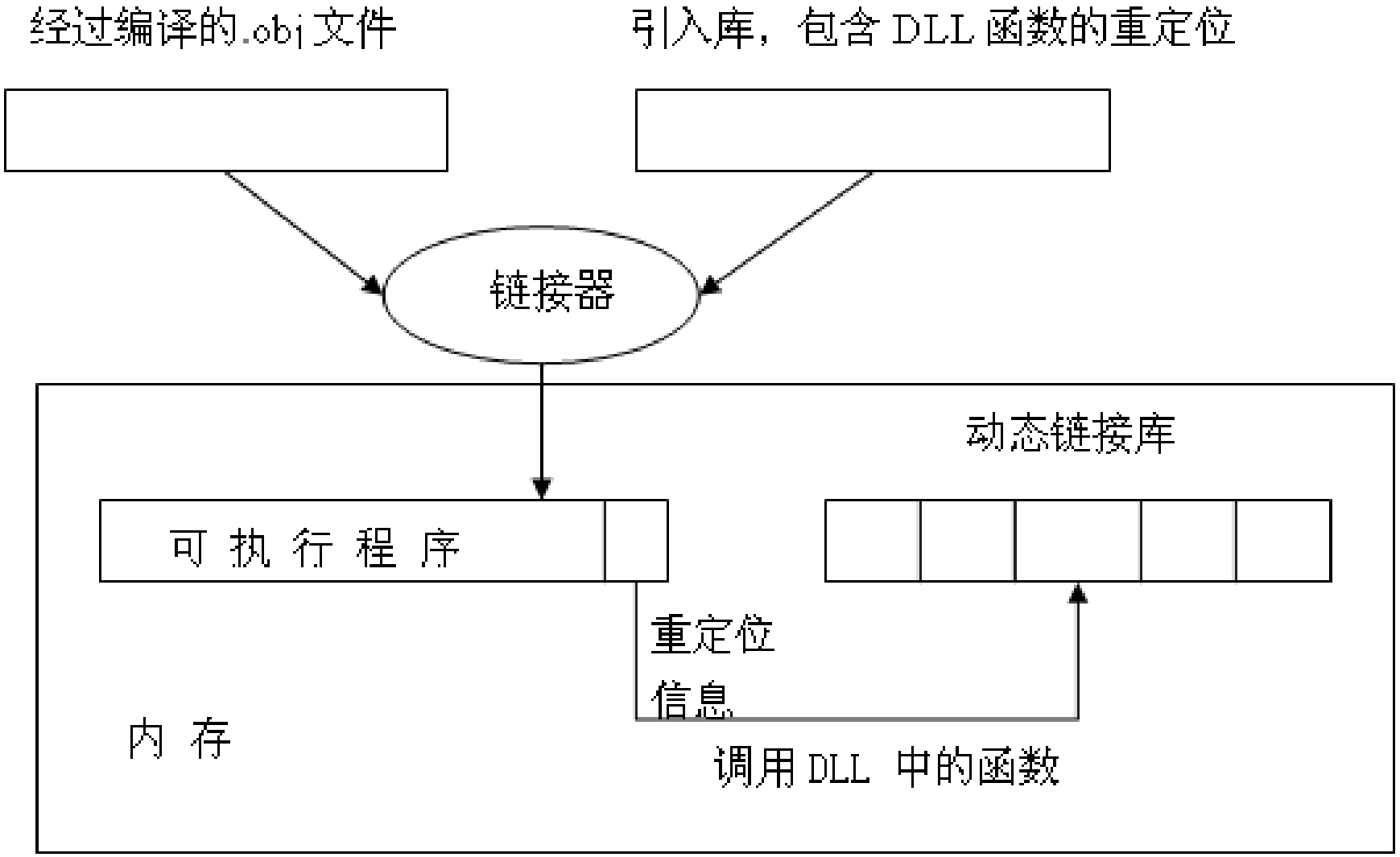
静态链接



动态链接

动态链接指的是在链接时并没有将库函数中的函数链接到应用程序的可执行文件中，链接是在程序中运行时动态地执行的。采用动态连接方式的库文件即为DLL(Dynamic Linkable Library)。尽管链接器并不把动态链接的函数复制到可执行文件中，但是它仍要清楚这些函数在什么地方以及怎样调用它们，为此需要引入库（**import library，.Lib文件**）来帮助连接器使用DLL，引入库中包含了DLL中函数的重定位信息。

动态链接


经过编译的.obj文件
引入库，包含 DLL 函数的重定位
链接器
动态链接库
可 执 行 程 序
重定位
信息
内 存
调用 DLL 中的函数

DLL到进程地址空间的映射

要调用DLL中的函数，首先必须把DLL的文件映像映射到调用进程的地址空间中。有两种方法可以实现这一映射：

a.装入时动态链接（**load-time dynamic linking**）

b.运行时动态链接（**run-time dynamic linking**）

装入时动态链接

当应用程序运行时，操作系统在装载应用程序时要查看EXE文件映像的内容，并将所有被引用的DLL文件映像映射到进程的地址空间中。

系统在寻找DLL文件时，按以下目录次序进行搜索：

1) 包含可执行应用程序的目录。

2) 当前目录。

3) Windows的系统目录，使用GetSystemDirectory函数可以返回该目录的路径。

4) Windows目录，使用GetWindowsDirectory函数可以返回该目录的路径。

如果按上述次序找不到DLL，应用程序即被终止。

运行时动态链接

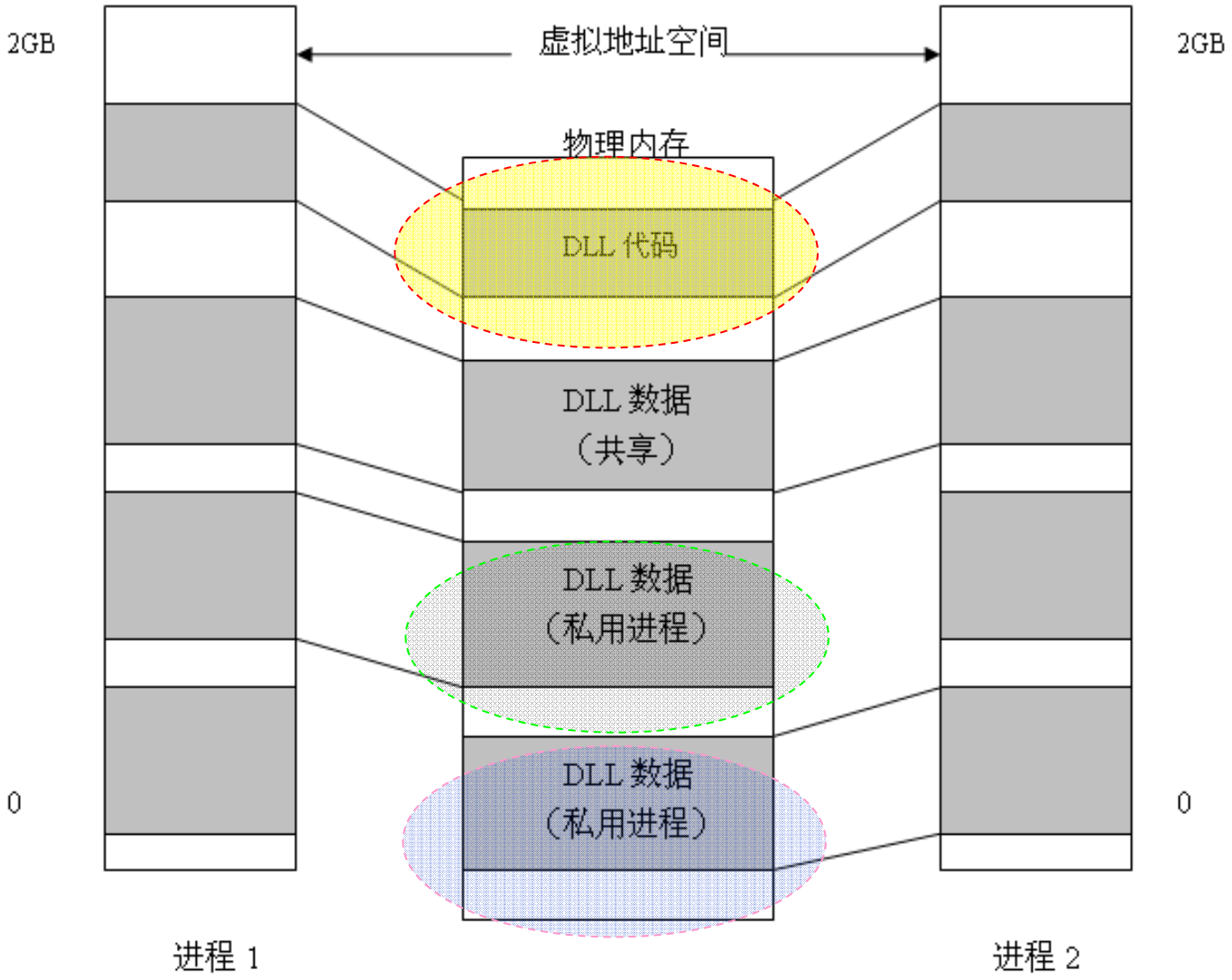
将链接推迟到运行期间，那么正确的DLL就可以判定，然后被动态链接，这便是运行时动态链接的基本思路。

1）HMODULE LoadLibrary(LPCTSTR lpszLibFile)；

2）BOOL FreeLibrary(HMODULE hLibModule)；

3）FARPROC GetProcAddress（HMODULE hModule, LPCTSTR lpszProc）；

DLL被映射到多个调用进程地址空间的一般情况



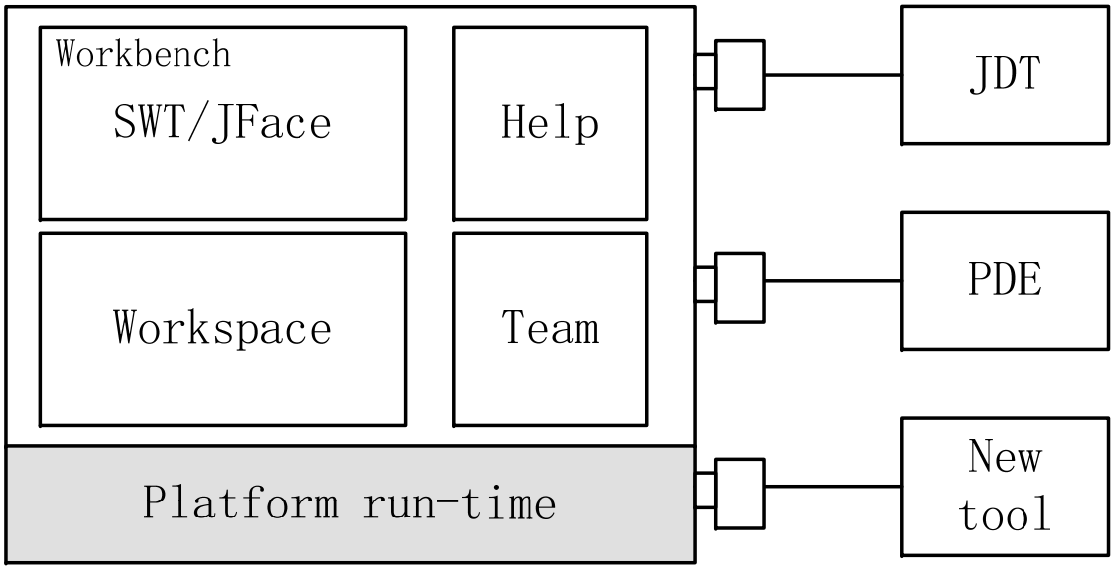
VC++中动态库创建和使用方法

- VC++动态链接库(DLL)编程深入浅出
- 演练：创建和使用动态链接库
- 导出函数动态库模板
- 导出类动态库模板

3.9.4 插件案例分析

案例1：Eclipse插件平台

- Eclipse是一个框架和一组服务，用于通过插件组件构建开发环境。Eclipse附带了一个标准的插件集，包括Java开发工具（Java Development Tools，JDT）和插件开发环境（Plug-in Development Environment，PDE）等。开发人员可以通过PDE对Eclipse进行扩展，构建与Eclipse环境无缝集成的工具。



案例2：模拟播放器程序

- ① 这不是一个真的播放器，当然也不能真的播放音频文件；
- ② 每个插件“支持”一种格式的音频文件，如“wav”、“mp3”等，通过增加插件可以使系统“支持”更多的音频格式；
- ③ 插件接口简单，其功能实现只是弹出一个对话框，表明哪个插件的哪个功能被调用了。制作这个播放器的真正目的是演示插件技术的原理和实现方法，只要掌握了其原理和方法就完全可以开发出有用的插件系统。

VC++插件实现方法

在插件系统中，插件实现方法主要有三种：

- 普通输出函数的DLL方式（导出函数）；
- 使用C++的多态性（导出类）；
- 使用COM类别（category）机制。

VC++插件实现实例

通过一个模拟的音频播放器来介绍插件的这三种实现技术。一般音频播放器都有这样一些基本功能:装载音频文件（LoadFile），播放（Play），暂停（Pause），停止（Stop）。

本例将提供这四个功能。但宿主程序本身并不会直接实现这些功能，而是调用插件的实现。每个插件支持一种音频格式，所以每个插件的功能实现都是不同。

普通输出函数方式
---插件的实现

创建一个动态链接库Plug.dll，为了支持四个基本功能，它输出相应的四个函数：

- Void LoadFile（const char *szFileName）；
- Void Play（）；
- Void Pause（）；
- Void Stop（）；

为了使宿主程序在运行时能知道这个插件可以支持什么格式的音频文件，插件程序还应输出一个函数供宿主程序查询用：

- Void GetSupportedFormat(char *szFormat)

{if(szFormat!=0) strcpy(szFormat,”mp3”);}

至此，这个插件就制作完了。可以依样画葫芦再做一个Plug3.dll,它“支持”.wma文件。

普通输出函数方式
---宿主程序的实现

宿主程序是一个基于对话框的标准Windows程序。它启动时会搜索约定目录（可以约定所有插件都存放在宿主程序所在目录的Plugins子目录下）并使用Win32函数Loadlibrary加载所有插件。每加载一个插件DLL，就调用另一个Win32函数GetSupportedFormat的地址，并调用此函数返回插件所支持的格式名（即是音频文件的的扩展名），然后把（格式名，DLL句柄）二元组保存下来。

当用户通过菜单打开文件时，宿主程序会根据扩展名决定调用哪个插件的LoadFile函数，并指明此插件DLL的句柄为当前使用的插件的DLL句柄（比如保存到变量m_hInst中）。此后当用户通过按钮调用Play等其他功能时，就调用句柄为m_hInst的插件的相应功能。

普通输出函数方式
---宿主程序的实现

```
typedef void（*PLAY）();
if(m_hInst)
   {
          PLAY Play =
          (PLAY)::GetProcAddress(m_hInst,”Play”);
          Play();
   }
```

另外，当程序退出时，应该调用FreeLibrary函数卸载插件。

基于类多态性方式
——插件的实现

```
Class ICppPlugin
{
Public:
        ICppPlugin(){};
        Virtual ~ICppPlugin（）= 0 {}；
        Virtual  void Release（）；
        Virtual void GetSupportedFormat(char *szFormat) = 0;
        Virtual void LoadFile(const char *szFileName) = 0 ;
        Virtual void Play();
        Virtual void Stop();
        Virtual void Pause();
};
bool CreateObject(void **pObj) {*pObj = new CppPlugin1();return *pObj!=0;}
```

基于类多态性方式
——宿主程序的实现

插件的加载过程与第一种方法相似。所不同的是，加载DLL后，首先调用的是插件程序的CreateObject输出函数来创建对象：

```
typedef bool(*_CreateObject)(void **pObj); //定义一个函数指针类型_CreateObject
createObj=(_CreateObj)::GetProcAddress(hInst,"CreateObject");
//获取CreateObject 的地址，hInst为DLL句柄
ICppPlugin *pObj = 0; //定义一个ICppPlugin的指针
createObj((void**)&pObj); //创建对象接下来查询插件所支持的格式名。本方式中，GetSupportedFormat已成为ICppPlugin的成员函数：
CString str;
pObj->GetSurpportedFormat(str.GetBuffer(8));
str.ReleaseBuffer();
```

基于类多态性方式
——宿主程序的实现

另外，需要保存的除（格式名，DLL句柄）二元组映射之外，还需要保存（格式名，创建对象函数指针）二元组映射以备后用：

```
m_formatMap[str] = hInst;
m_factoryMap[str] = createObj; // str存放的是格式名字符串的小写形式
```

同样，在打开文件时选择使用哪个插件：

```
ICppPlugin *m_pObj;// *m_pObj存放当前使用的对象指针，在程序初始化时要置为0
if（m_pObj）{ m_pObj->Release();m_pObj = 0;}
m_factoryMap [strEx]((void**)&m_pObj);//调用CreateObject
m_pObj->LoadFile((LPCSTR)strFileName);//strFileName是音频文件全路径名
```

以后就可以使用m_pObj来调用其它操作了，例如：if(m_pObj) m_pObj->Play();

在宿主程序退出时需要卸载DLL。

作 业

- 基于Java的多态性实现一个模拟的播放器插件程序。

（参考文献：李延春，软件插件技术的原理与实现，《计算机系统应用》，2003 ）

提示：

（1）宿主程序实现：可基于Java AWT框架搭建软件界面，由AAPlayer、AAPFrame、AboutBox三个Java类构成，AAPlayer是整个程序的主类，负责构建AAPFrame类的实例，进行基本的配置等。AAPFrame 类负责初始化音频播放器界面，选择加载音频播放插件，为相关按钮添加事件，控制音频的播放等操作。AboutBox用来介绍软件信息；

（2）插件实现：模拟音频播放器的播放插件都是基于接口IPlayerPlugin实现的，包括MP3PlayerPlugin、WAVPlayerPlugin、OGGPlayerPlugin等，用户还可以根据自己的需求，去扩展不同文件格式的播放插件。