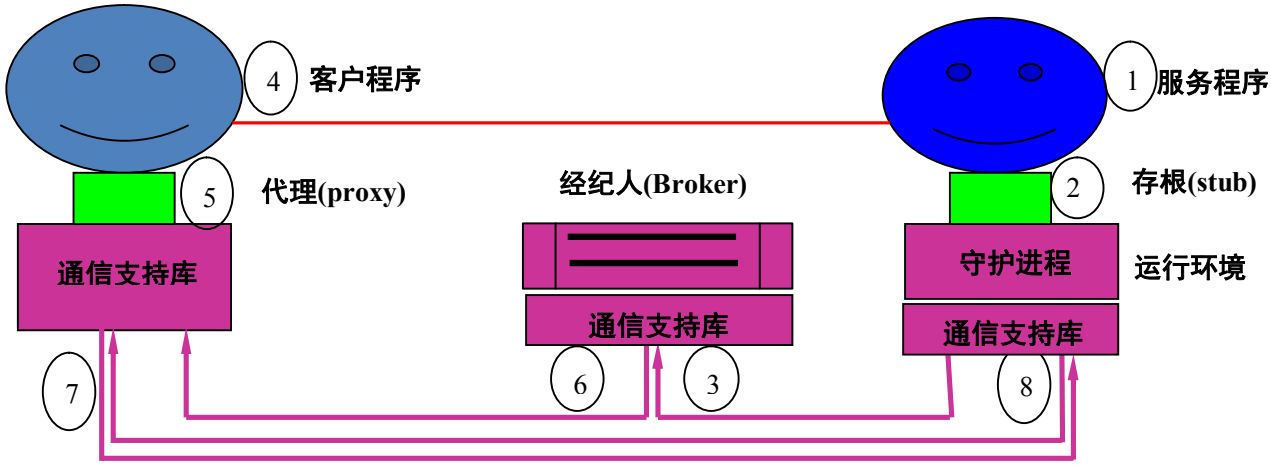
中间件技术课程复习

一、软件互操作的架构



某个应用程序使用了一个第三方的服务组件 printf.dll，使用光盘发布，在光盘的根目录下有四个目录：stub， proxy, original，client 分别存放服务程序的存根 prinf_stub.dll，服务程序的代理 prinf.dll 和配置参数文件 config.txt（包括 port=80，IP=10.10.10.10 两项），服务程序 printf.dll，以及客户应用程序 Client.exe。守护进程由微软提供，其运行程序为 server.exe，放在光盘根目录下的 Daemon 目录下，server.exe 监听的端口号为 24。在光盘根目录下的 communication 目录下有通信支持库组件 pipeline.dll。现有一用户 A 要求将应用程序安装在其机器的 c:/app 目录下，运行。另一用户 B 要求将应用程序安装在其机器的 d:/app 目录下，而服务组件 printf.dll 安装在另一服务器（IP 地址为 192.17.105.10）的 c:/service 目录上运行。请你分别为用户 A 和用户 B 给出配置安装方案。

为用户 A: 同一进程中同进程情况下的互操作：将客户程序./client/client.exe 文件，以及服务程序./original/printf.dll 文件，拷贝到用户机器上的应用程序运行目录 C:\app\下，即完成了部署，用户即可打开和运行 client.exe 文件；

为用户 B: 不同机器间的互操作: 将客户程序./client/client.exe 文件，以及服务程序的代理./proxy/printf.dll 文件，配置文件./proxy/config.txt，通信支持库组件./communication/pipeline.dll 四个文件拷贝到用户机器上的应用程序运行目录 d:\app\下，将 config.txt 中的 port=80 改成 port=24，IP=10.10.10.10 改成 IP=192.17.105.10，即完成了一端的部署。再将./ Daemon/server.exe, ./stub/print_stub.dll, ./original/printf.dll, 通信支持库组件./ communication/pipeline.dll 四个文件拷贝到服务器机器侠的 c:/service 目录下，

启动运行 server.exe，然后用户即可打开和运行 client.exe 文件。

二、对于由独立模块以动态链接方式组合而成的应用程序，每个模块都独立，存在不断升级改版的特征，解决其互操作问题的办法：1）对于模块内部：绝对寻址→相对寻址；2）对于模块之间：直接寻址→间接寻址；3）对于多程序并发执行环境，实现模块共享：虚拟内存，和代码可重入（栈）。独立模块之间互操作的协同：函数调用规范。

2）张三的机器使用的是 windows 操作系统，原装有 word，dephi 两个应用程序，两个程序都很正常，今天他在机器上新安装了一个 realplayer 应用程序后，发现 word'启动出错，报“内存访问错误”，dephi 启动变得很慢，请分析原因。

word'启动出错，报“内存访问错误”可能原因是安装 realplayer 应用程序时，把机器中已有的某个共享库的新版本拷贝到了 windows 的 system32 目录下，把原来的老版本覆盖掉了，因为 word 是使用老版本，因此它创建的对象，没有新版本所期望的对象数据成员，于是在执行该共享库中的代码时，就会访问在内存中没有分配空间的数据成员，于是就发生“内存访问错误”的异常。
dephi 启动变得很慢的原因是，安装 realplayer 应用程序时，把机器中已有的某个共享库的新版本拷贝到了 windows 的 system32 目录下，把原来的老版本覆盖掉了。新版本的共享库文件要比老版本的大，加载时要占更多的内存空间，挤占了 dephi 中其它模块的内存空间，导致 dephi 中位于其后的二进制可执行模块不能被加载到它所期望的内存位置，再加上这些模块又是在编译时是以直接寻址方式编译的，因此在启动加载时，须要对这些模块进行扫描，解析，并对地址部分做修改，扫描、解析和修改费时间，因此启动就变慢。

3）就软件互操作而言，在面向对象的编程中，接口有哪些基本特性？

二进制特性；
接口不变性；
继承性(纵向)，和扩展性（横向）；
全球范围内的标识性；
多态性——运行过程中的多态性；
对等性；
完全联通性；

4）中间件技术是有关客户方软件与服务方软件之间互操作的技术，它的基本特性有哪些？

1）二进制级的互操作；
2）服务程序有多个客户；
3）客户和服务都是邦联形式的组合，它们彼此具有独立性，都会独自升级进化，

但又共同维持邦联性；

4）客户和服务之间的邦联契约既要具有永恒性，又具有可延伸性（横向、纵向）；

5）对应用程序员，保持编程模式的不变性；

三、面向对象编程模式下的独立模块之间的互操作：服务提供方（类的定义以及类的成员函数的实现）负责服务类的实例对象的生命周期管理，也就是实例对象的创建（new）和销毁（delete）。服务类的实例对象的 new 和 delete 不能由交由客户方来执行。

有了这个限制，客户方访问服务，就只有通过接口这条唯一的途径来实现了。因此，接口成了软件互操作中最重要和最核心的概念，对于搞软件设计的人，必须对接口这个概念有清晰的认识。

服务方必须要有一套管理机制，来实现实例对象的生命周期管理，和模块的生命周期管理。模块在类的上一级，一个模块可能包含多个类的定义和实现。对于对象的生命周期管理，当没有用户要继续访问某一对象时，该对象就失去了继续呆在内存中的意义，可以 delete，释放其占用的内存。对于模块的生命周期管理，对它包含的类，如果相应的对象都被 delete 了，那么这个模块也就没有必要继续呆在内存了，可以从内存中将其清除，释放其占用的内存。

因此在服务方，对每个对象，都要跟踪记录当前有多少客户拿走了其开放的接口。对于客户，当用完了某个接口时，就要通知该服务方，告知其该接口已经使用完毕，以后不再使用。当一个对象的用户计数器为 0 时，表明已经没有客户要继续对其访问了，因此可 delete。

有如下接口：

```
Class   IUnKnown {
    public:
    virtual void Delete( ) = 0;
    virtual   void ** QueryInterface(char *InterfaceName) = 0;
}
class IPersistence: public IUnKnown {
    public:
    virtual bool Load(const char *pszFileName) = 0;
    virtual bool Save(const char *pszFileName) = 0;
};
class IFunctuion : public IUnKnown {
    public:
    virtual int Length( ) = 0;
    virtual int Find(const char *psz) = 0;
}
```

有服务程序：

```
class FastString : public IFuction, public IPersistence {
      virtual void ** QueryInterface(char* InterfaceName);
      virtual void Delete( ); // deletes this instance
```

```
        virtual int Length( ); // returns # of characters
        virtual int Find(const char *psz) const; // returns offset
        virtual bool Load(const char *pszFileName);
        virtual bool Save(const char *pszFileName);
};
void class FastString::Delete( ) {
   delete this;
}
```

有客户程序：

```
IFunction *pfs = 0;
IPersistence *ppo = 0;
pfs = CreateFastString("Fed animal list:tiger,fog,dog, monkey");
if (pfs)   {
pfs->Find(“fog”);
      ppo = (IPersistence *)pfs-> QueryInterface ("IPersistence");
      if (ppo)        {
                  ppo->Save("C:\\animal_list.dat");
                  ppo->Delete();
      }
       pfs->Delete();
}
```

1）分析上述客户程序存在什么问题？

会出现“内存访问错误”，程序异常崩溃现象。原因是执行 ppo->Delete()，服务端 FastString 类的对象已经从内存中释放，因此在执行 pfs->Delete()语句时，要再访问该对象，但是在内存中该对象已经不存在了。于是就出现 “内存访问错误”，程序异常崩溃。

为了解决该问题，将 Class IUnKnown 改为：

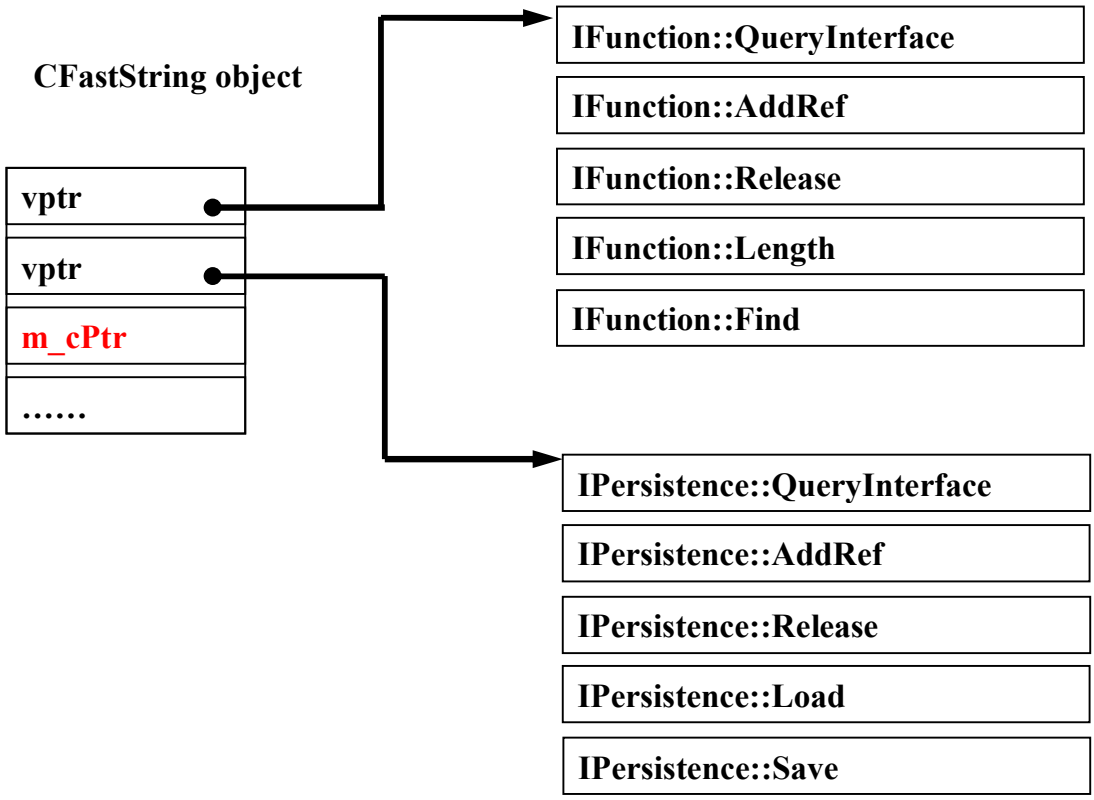
```
Class   IUnKnown {
     public:
     virtual   void ** QueryInterface(char *InterfaceName) = 0;
     virtual   void     AddRef( ) = 0 ;
     virtual   void     Release( ) = 0;
}
```

2）请你重新写上述 FastString 类的定义，并写出它的 QueryInterface，AddRe，Release 三个函数的实现代码，满足客户对任一接口的获取，还有实例对象的生命周期管理。

```
class FastString : public IFuction, public IPersistence {
 intm_cPtrs = 0;
   virtualvoid ** QueryInterface(char* InterfaceName);
   virtualvoidAddRef( );
         virtualvoid Release( );
   virtual int Length( );
   virtual int Find(const char *psz) const;
   virtual bool Load(const char *pszFileName);
   virtual bool Save(const char *pszFileName);
};

void ** FastString::QueryInterface(char* InterfaceName)   {
     if (strcmp(pszType, "IFunction") == 0) {
     AddRef( );
          return static_cast< IFunction *>(this);
     }
     else if (strcmp(pszType, "IPersistence") == 0)   {
          AddRef( );
          return static_cast<IPersistence *>(this);
     }
     else if (strcmp(pszType, "IUnKnownt") == 0) {
          AddRef( );
          return static_cast< IFunction *>(this);
     }
     else
          return 0;
}
void FastString::AddRef()   {
     ++m_cPtrs;
     return;
}
void FastString::Release()   {
    if (--m_cPtrs == 0) {
         delete this;
   }
   return;
}
```

3）画出 Class FastString 实例对象的内存布局结构图；



4）分析上述客户代码，其中调用了服务模块开放给客户的一个函数 CreateFastString，来初始获得 IFunction 接口指针，请给出该函数的实现代码。服务方给客户方的头文件 Faststring.h 中要给出该函数的定义，请你写出该函数的定义，要求指明该函数使用 C 语言函数调用规范。

```
IFunction * __declspec(dllexport) __cdecl    CreateFastString(const char *psz)
    return new FastString(psz);
}

extern "C" IFunction * CreateFastString(const char *psz);
```

5）重写上述客户代码，包含 Faststring.h，并使之不存在第一问中的问题；

```
IFunction *pfs = 0;
IPersistence *ppo = 0;
If(    pfs == 0)
    pfs = CreateFastString("Fed animal list:tiger,fog,dog, monkey");
If (pfs)
    pps->Find("fog");
     ppo = (IPersistence *)pfs-> QueryInterface ("IPersistence");
```

```
    if (ppo)        {
                ppo->Save("C:\\animal_list.dat");
                ppo->Release();
              ppo = 0;
    }
     pfs->Release();
    pfs = 0;
}
```

6）对于服务方创建的对象，例如该例子中的 FastString 类的实例对象，客户方是通过其实现的接口来对其访问的，如果客户方不再须要访问了，那么服务方创建的对象就应该销毁，释放其占用的内存空间，否则就会一直保留在内存中，导致内存泄漏的问题。请结合上述客户代码，指出在什么情况下会出现内存泄漏？对此问题，请你从对客户程序的编写要求，以及编译器辅助方面，分析其解决办法？

当客户拿取了一个接口，在使用后，如果忘记调用 Release（），那么服务对象的计数器就不会减到 0，也就是，服务方总以为客户还要继续该对象，于是也就不释放该对象，于是该对象就一直占着内存，随着时间的推移，越来越多的对象被创建，慢慢地，内存就会被耗光，出现异常。

因此，预防发生内存泄漏的措施是：

1）对于客户程序员来说，对于一个接口变量，一定要记得赋初值为 0；

2）当客户程序员要使用某个接口时，先要检查接口变量是否为 0，如果为 0，说明该接口还未从服务方获取，因此就去调用获取函数来获取接口，否则就表明该接口已经获得，不要重复获取它了。

3）获取某个接口不一定能够成功，因此在使用接口前，一定要核查接口变量不为 0，方可使用接口。

4）在用完一个接口以后，一定要记得调用 Release( ), 然后给接口变量复位为 0，表明该接口以后不能直接使用。

对于第一点，第二点和第三点，编译器可以帮忙客户程序员把关。对于第四点，如果接口变量为局部变量，编译器可以在函数返回时，添加一段代码，检查接口变量是否为 0，如果不为 0，替客户程序员添加一行 Release()代码；如果接口变量为成员变量，编译器则可在析构函数中添加代码，来检查接口变量是否为 0，如果不为 0，替客户程序员添加一行 Release()代码；如果为函数调用传递来的变量，说明该接口不是由自己获得，因此就不要调用 Release。接口变量如果为全局变量或者是静态变量，编译器则无法帮忙，因为这些变量的寿命周期为进程的生命周期，编译器无法判断，后面是否还要使用该接口。正是因为这一点，全局变量在新一代面向对象语言（Java,.Net)中被取消了。静态变量没有取消，因此千万不要把接口变量定义成静态变量。