COM多线程模型、DCOM

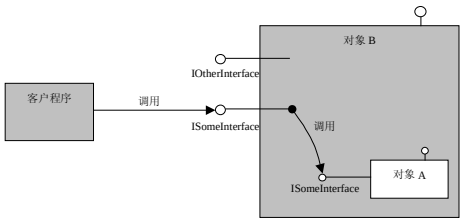
潘爱民

http://www.icst.pku.edu.cn/compcourse

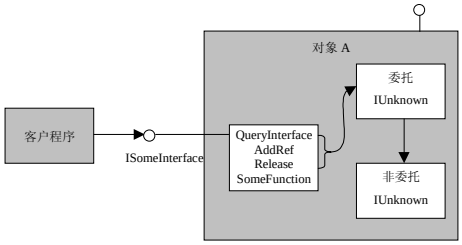
内容

- 复习：COM聚合和COM跨进程模型
- COM线程模型
- 分布式COM(DCOM)
  - DCOM基本结构
  - 对象激活
  - 连接管理
  - 并发管理
  - DCOM安全模型

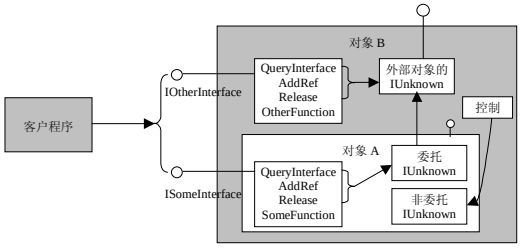
复习：COM包容模型



复习：聚合，支持聚合的对象在非聚合方式下的接口示意图



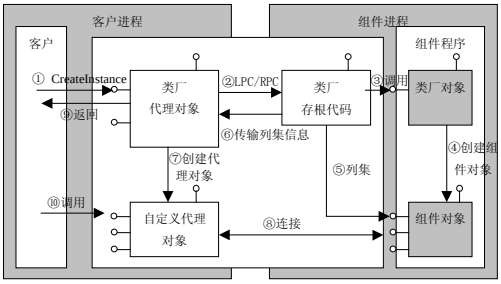
复习：聚合，支持聚合的对象在聚合方式下的接口示意图



聚合模型的要点

- 外部对象
  - 创建内部对象的时候，外部对象必须把自己的IUnknown接口指针传给内部对象
  - 当外部对象接到对于聚合接口的请求时，它必须调用非委托版本的IUnknown的QueryInterface函数，并把结果返回给客户
- 内部对象
  - 内部对象类厂的CreateInstance必须检查pUnkOuter 参数
  - 嵌套聚合：传递最外层的pUnkOuter参数
  - 除了非委托版本的IUnknown之外，其他接口的三个IUnknown调用必须全部委托给外部对象的pUnkOuter

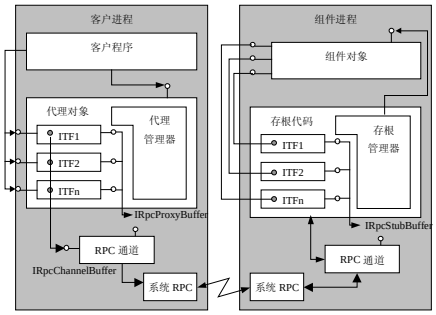
通过类厂建立代理对象和组件对象自定义列集过程



自定义列集的要点

- 对象必须实现IMarshal接口
- 代理对象也必须实现IMarshal接口，并且代理对象与进程外对象之间协作
- 代理对象必须负责所有接口的跨进程操作
- 典型用途：
  - 提高跨进程调用的效率，使用缓存状态等优化技术
  - marshal-by-value

标准列集的proxy和stub结构



进程外组件注意事项

- 自注册方式的变化
  - 命令行参数/RegServer和/UnregServer
- 注册类厂
- 何时被卸载
- 调用CoInitialize和CoUninitialize
- 实现自定义接口的代理/存根组件

多线程特性

- Win32线程和COM线程
- marshaling和同步
- 套间线程
- 自由线程
- 进程内组件的线程模型

进程和线程

- 进程
  - 在Linux平台上，时间和空间概念的结合
  - 在Windows平台上，是个空间概念
- 线程
  - 在Linux平台上，一个执行环境
  - 在Windows平台上，也是一个调度单元，是个时间概念

Win32线程

- Win32系统线程本身只有一种，根据应用模型可分为两种
- CreateThread，创建线程
- UI线程(user-interface thread)
  - 包含消息队列，当线程首次调用Win32 User或GDI函数时产生
  - 常常包含消息循环，组合GetMessage/ TranslateMessage/DispatchMessage
- 辅助线程(worker thread)
  - 一条执行线索，没有UI，没有消息概念

COM线程

- 按照COM对象的执行环境，分为套间线程和自由线程
- 套间线程(apartment thread)
  - 位于一个STA中(Single-Threaded Apartment)
  - 一个套间对应一个线程
- 自由线程(free thread)
  - 位于MTA中(Multi-Threaded Apartment)
  - 一个进程有一个MTA，它可以包含任意数量的自由线程

marshaling

- 调用者与被调用者如果位于不同的线程中，则调用过程要有两次线程切换，线程切换也需要用到marshaling机制
- COM对象的线程相依性
  - 有的COM对象只能在一个线程中运行
  - 内含UI的COM对象只能在创建线程上运行
- 线程之间的marshaling机制与进程间的marshaling过程一致

同步

- 只能被一个线程访问的对象不需要同步
  - 例如Windows的窗口过程
  - 但是对于全部变量的访问，仍需要同步保护
- 有可能被多个线程访问的对象需要有同步机制
  - Event、Semaphore、CriticalSection、Mutex
  - 这样的代码被称为thread-safe

Apartment(套间)

- 是一个逻辑概念，也有实体对应
- 是COM对象的执行环境
- 分为三种套间
  - STA
  - MTA
  - COM+引入TNA(thread-neutral apartment) *

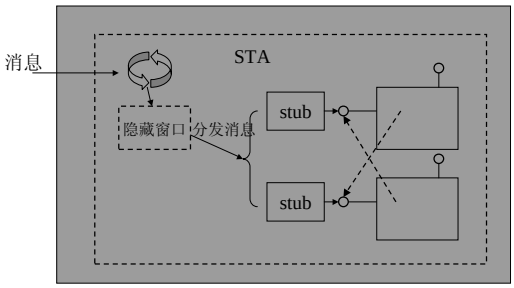
STA套间

- 每个STA套间包含一个线程
  - STA套间和线程有一一对应关系
- 当线程被创建后，用COM库初始化就建立起一个STA套间
  - CoInitialize(NULL);
  - 或者CoInitializeEx(NULL,COINIT_APARTMENTTHREADED);
- 在线程结束之前，调用CoUninitialize结束套间
- 套间线程包含消息循环
- COM在套间线程中创建了一个隐藏的窗口
  - 用途：同步、分发消息

STA套间(续)

- 在STA中创建的COM对象都属于这个STA
- STA对象不必处理同步，因为对象的方法代码只能被这个STA套间的线程调用
- 但DLL程序的引出函数如DllGetClassObject和DllCanUnloadNow等仍需同步处理。
- 类厂是否需要线程安全，取决于类厂的策略
- 如何把接口指针交给调用者
  - 自动marshaling
  - 手工marshaling

STA接收调用示意图



STA套间传递接口指针

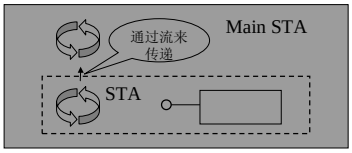
- 通过COM传递的接口指针，COM都会自动marshaling
- 手工marshaling
  - CoMarshalInterface和CoUnmarshalInterface
  - CoMarshalInterThreadInterfaceInStream和CoGetInterfaceAndReleaseStream

套间线程要点

- STA客户调用STA对象的过程
  - RPC通道，通过消息传递调用，RPC通道发出消息后，调用MsgWaitForMultipleObjects阻塞调用方，但仍然分发消息，所以UI仍是活的
- STA客户跨进程调用的过程
  - 代理对象先得到一个RPC线程，并把列集后的数据包交给它，RPC线程又生成辅助线程，由它负责跨进程调用，RPC线程也负责分发消息，所以UI也是活的
- 从另一个进程进入套间线程的过程。
  - 调用首先到达一个RPC线程，RPC线程向套间线程隐藏窗口发送消息，并把列集数据传过去

套间线程：典型情形

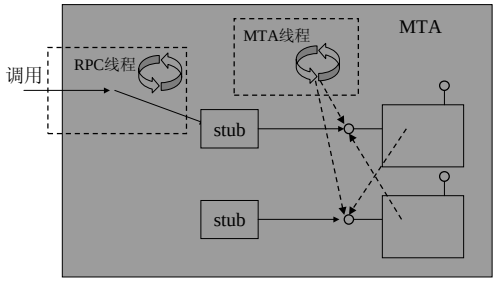
- 客户创建套间线程，然后套间线程的主函数创建COM对象，套间线程再调用CoMarshalInterThreadInterfaceInStream函数，把接口指针列集到流对象中，然后通知客户线程对象已经创建成功，客户线程接到通知后，利用流对象指针把对象的接口指针散集出来，以后客户线程就可以与对象通讯了。



MTA套间

- 每个进程至多只有一个MTA套间
- MTA套间中可以包含一个或者多个线程
  - 线程初始化：CoInitializeEx(NULL, COINIT_MULTITHREADED);
- Windows的版本：
  - NT 4.0或者Windows 95 + DCOM扩展
- MTA套间中的对象必须thread-safe
  - 被MTA中线程创建的对象为MTA对象
  - MTA中的线程都可以直接访问MTA对象
  - 使用Win32多线程编程技术保证thread-safe

MTA接收调用示意图



MTA中线程的特点

- 使用CoInitializeEx(NULL, COINIT_MULTITHREADED)初始化
- 若客户与对象都在MTA中，则调用直接在客户线程中执行
- 若客户程序在另一个进程中，则调用通过proxy/stub，直接在RPC线程上执行
- 若客户在STA中，则调用通过proxy/stub，直接在RPC线程上执行
- 没有隐藏窗口，MTA线程创建的对象并没有生存在任何特定的线程中，而是生存在MTA中
- MTA对象必须是线程安全的，允许重入

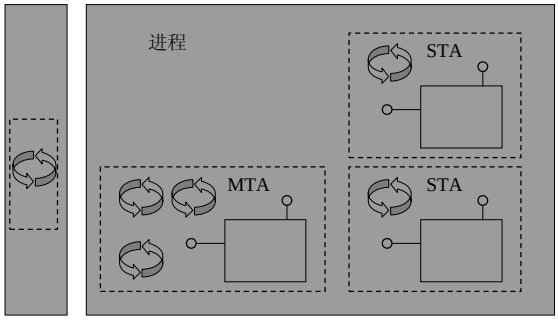
MTA线程要点

- MTA线程中的客户调用STA对象或其他进程中的对象
  - 客户调用代理对象，代理对象又调用RPC通道，RPC通道阻塞客户线程，如果客户线程中有窗口的话，则会被锁住
- 调用进入MTA线程中的对象
  - RPC通道直接使用RPC线程，并调用存根代码，存根再调用组件对象。这中间没有线程切换。

STA和MTA之间关系

- 不同进程之间，不管什么线程类型，都需要proxy/stub
- 在同一个进程内不同STA之间，也需要proxy/stub
- 在STA内部，一个对象调用另一个对象的方法不需要proxy/stub
- 在MTA内部，对象和调用者之间调用不需要proxy/stub
- 从STA调用MTA，需要proxy/stub
- 从MTA调用STA，需要proxy/stub

进程内的STA和MTA



进程内对象的线程模型

- CLSID\{clsid}

  InprocServer32

  ThreadingModel = “Apartment”或“Free”、“Both”
- DllGetClassObject和DllCanUnloadNow同步
- 当COM库创建对象时
  - 如果客户线程模型与对象的要求一致 则在客户线程中创建对象，返回直接指针
  - 如果客户线程模型与对象的要求不一致：COM会产生一个相应的线程供对象使用

“Both”类型

- 这种对象必须是thread-safe
- 它总是位于创建者的套间中
- 本质上具有“Free”对象的特性，但是在运行时刻可能会表现出“Apartment”对象的特性
- 例子：
  - 接口proxy/stub都是“Both”类型

Main STA

- 第一个STA，往往是进程的主线程
- 如果一个COM class的ThreadingModel值为空，则默认为Main STA
- 在引入MTA之前的对象都使用Main STA

客户线程与对象模型组合表

客户线程 / 对象模型	MTA线程	Main STA	非Main-STA
“Both”	直接	直接	直接
“Apartment”	Proxy/Stub	直接	直接
“Free”	直接	Proxy/stub	Proxy/stub
“”(无)	Proxy/stub	直接	Proxy/stub

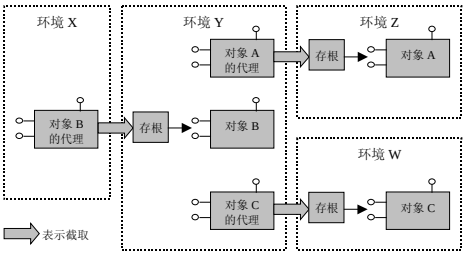
进程外对象的线程模型

- 与客户线程模型没有关系
- 由组件服务程序自己来控制
  - 类厂对象通过CoRegisterClassObject来传递
  - 类厂的CreateInstance函数

COM+增强

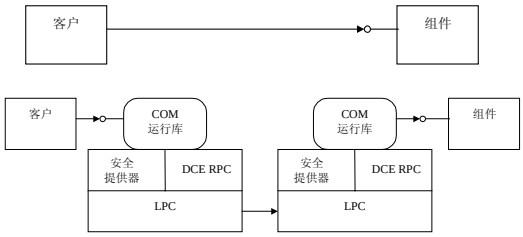
- 环境
  - 对象的执行环境，包含环境信息，如事务、安全性等
  - 在原来的模型中，我们认为MTA和STA就是一个执行环境
  - 跨环境调用也需要proxy/stub
- 轻量代理
  - 不需要线程切换的跨环境调用
- TNA(Thread neutral apartment)

COM+环境

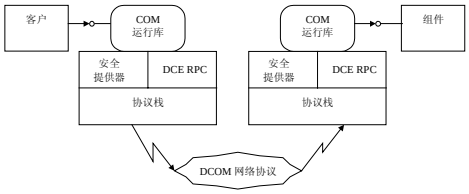


休息

从进程内走向进程外
进程透明性



从本机走向远程主机
DCOM



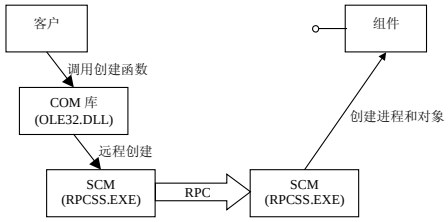
预备：RPC(Remote Procedure Call)

- RPC Client and RPC Server
- 可以在TCP、UDP上实现
- RPC是分布式应用的基础，过程如下：
  - 客户准备：三元组(远程机器、server、函数)
  - 在RPC Server上需要运行一个RPC端口管理服务，每个RPC Server向它登记注册
  - RPC client与远程机的RPC端口管理服务联系，请求RPC Server的端口号
  - 然后RPC client与RPC server直接联系
- 其他远程调用途径，如HTTP、SOAP

DCOM要点

- 创建远程对象
- 把进程内对象放到远程机器上
- DCOM的连接管理
- DCOM并发性管理
- DCOM安全性

DCOM组件对象的创建过程



对象激活(activation)

- 对象激活：
  - 创建新的组件对象——类厂对象
  - 建立已有组件对象与客户之间的连接——名字对象
- 远程对象的创建：
  - 标识一个远程对象：CLSID+RemoteServerName
- 如何获取RemoteServerName 信息
  - DCOM配置工具指定远程服务器名
  - 客户程序在代码中显式指定远程服务器名

创建DCOM组件(一)

- 位置透明性：客户程序不必知道组件运行在本地或远程机器上
- RemoteServerName信息：

HKEY_CLASSES_ROOT\APPID\{appid-guid}
"RemoteServerName"="<DNS name>"

HKEY_CLASSES_ROOT\CLSID\{clasid-guid}
"AppID"="<appid-guid >"

- RemoteServerName信息不能被传递
- 客户创建组件对象的代码不必修改

使用DCOM配置工具配置组件的RemoteServerName信息

创建DCOM组件(二)

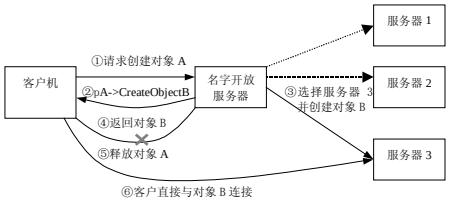
- 在CoGetClassObject和CoCreateInstanceEx函数中指定服务器信息

```
typedef struct  _COSERVERINFO {
   DWORD dwReserved1;
   LPWSTR pwszName;
   COAUTHINFO  *pAuthInfo;
   DWORD dwReserved2;
}  COSERVERINFO;

typedef struct _MULTI_QI {
   const IID*   pIID;
   IUnknown *   pItf;
   HRESULT      hr;
} MULTI_QI;
```

用CoCreateInstanceEx创建远程组件举例

```
HRESULT hr = S_OK;
MULTIQI mqi[] = {{IID_IBackupAdmin, NULL , hr1}};
COSERVERINFO srvinfo = {0, NULL, NULL, 0};
Srvinfo.pwszName = pFileServerName;

HRESULT hr = CoCreateInstanceEx( CLSID_MyBackupService,  NULL,
      CLSCTX_SERVER, &srvinfo,
      sizeof(mqi)/sizeof(mqi[0]), &mqi);
if (SUCCEEDED(hr)) {
       if (SUCCEEDED(mqi[0].hr))
       {
          IBackupAdmin* pBackupAdmin=mqi[0].pItf;
          hr=pBackupAdmin->StartBackup();
          pBackupAdmin->Release();
       }
}
```

用分派服务组件对象实现动态负载平衡功能



远程创建进程内组件：代理进程(Surrogate)

- 代理进程优点：
  - 进程内组件程序中的严重错误只影响到代理进程，不会使客户进程崩溃；
  - 一个代理进程可以同时为多个客户提供服务；
  - 在代理进程中运行进程内服务可使DLL享有代理进程的安全性。

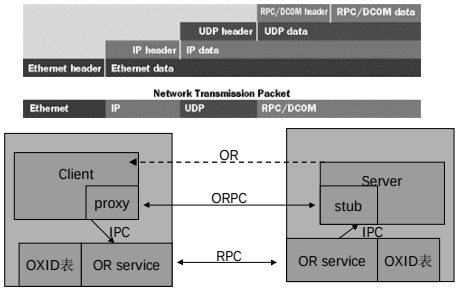
进程内组件被装入代理进程的条件

- 系统注册表中，在COM对象的CLSID关键字下必须要指定AppID值，以及对应的AppID关键字；
- 客户程序在创建对象实例时，必须设置CLSCTX_LOCAL_SERVER标志；
- 组件对象的CLSID关键字下不指定LocalServer32、LocalServer、LocalService值；
- 组件对象的CLSID关键字包含InProcServer32 子键；
- 在InProcServer32子键中指定的DLL文件必须存在；
- 组件对象对应的AppID键下指定DllSurrogate值。

控制远程对象的生存期

- DCOM优化了远程对象的AddRef和Release调用，客户程序不必考虑优化
- OR (OXID Resolver)服务
  - OXID(object exporter identifier) 对象
- OXID对象实现了IRemUnknown接口
  - RemQueryInterface
  - RemAddRef和RemRelease
- 参考资料：MSJ，1998年第3期
  - Understanding the DCOM Wire Protocol by Analyzing Network Data Packets

DCOM使用ORPC协议实现远程通信



Pinging机制(一)

- 对于非正常情况，若组件进程非正常终止，客户可以根据返回值判断出来
- 若客户非正常终止，组件进程该怎么办？
- 为了检测客户程序是否非正常终止，DCOM提供了“pinging”机制
- 每个被远程使用的对象都有“pingPeriod”和“numPingsToTimeOut”计数

Pinging机制(二)

- ping周期：pingPeriod*numPingsToTimeOut
- 当前DCOM版本中，pingPeriod=2(分)且numPingsToTimeOut=3，这些值不能被改变
- pinging机制的优化：OXID解析器产生ping集，减轻网络负担

连接传递

- 连接具有可传递性，因为接口的列集数据(OR或者OBJREF)包含机器相关的信息
- 连接传递与创建传递含义不同，DCOM不支持创建传递
- 可用连接传递间接支持创建传递
- 利用连接传递性可实现动态负载平衡

并发管理

- 分布式环境下的基本问题
- 同步方式转向异步方式
- 多线程模型
- 消息过滤器

消息过滤器(message filter)机制

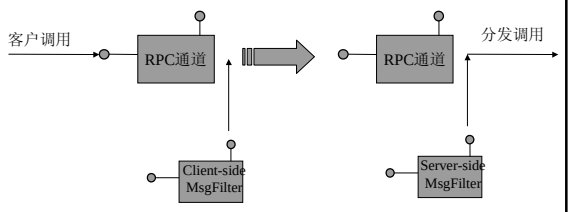
- 是STA套间特有的一种控制机制
  - 处理跨线程调用过程中的阻塞现象
- 既可用于客户程序，也可用于组件程序
- COM本身提供了缺省的实现，又允许我们使用自定义的message filter

Message filter

- 是一个简单的COM对象，它实现了IMessageFilter接口：

```
class IMessageFilter : public IUnknown
{
public :
   DWORD HandleInComingCall ( DWORD dwCallType,
                              HTASK threadIDCaller,
                              DWORD dwTickCount,
                              LPINTERFACEINFO pInterfaceInfo) = 0;
   DWORD RetryRejectedCall(HTASK threadIDCallee, DWORD dwTickCount,
                              DWORD dwRejectType) = 0;
   DWORD MessagePending (HTASK threadIDCallee, DWORD dwTickCount,
                              DWORD dwPendingType) = 0;
};
```

Message Filter作用示意图



IMessageFilter 接口

- 返回值类型都是DWORD
- 类型HTASK用来标识一个调用，通常为逻辑线程ID，在一台机器上唯一存在
- HandleInComingCall用于组件对象一方的消息过滤器
- RetryRejectedCall和MessagePending用于客户一方的消息过滤器

实现消息过滤器机制

- 客户程序或者组件程序实现过滤器对象，然后用CoRegisterMessageFilter函数指定使用自定义的消息过滤器；否则COM使用缺省的过滤器对象。
- 消息过滤器只用于当前STA，它不支持marshaling，不能跨套间使用

组件方的消息过滤器

- 当COM检测到有一个外部调用进入当前进程时，它首先会调用消息过滤器的HandleInComingCall成员函数。
- HandleInComingCall函数的参数dwCallType指示了调用的类型，参数pInterfaceInfo标识了对象、接口和成员函数信息
- HandleInComingCall函数的返回值：
  - SERVERCALL_REJECTED
  - SERVERCALL_RETRYLATER
  - SERVERCALL_ISHANDLED

客户程序方的消息过滤器

- 当客户的调用被组件拒绝之后，COM在作了有关的处理后调用过滤器的RetryRejectedCall成员函数，由客户决定是否继续
- RetryRejectedCall返回-1表示放弃调用；否则返回一个以毫秒计的时间数，经过这段时间后，COM将再次调用组件对象
- 缺省对话框：

客户程序的消息处理

- RPC通道也为客户程序分发消息，所以在远程调用完成之前，客户程序的消息队列也会接收到消息，MessagePending成员函数让客户有机会控制消息处理方案。
- 返回值：
  - PENDINGMSG_CANCELCALL - 取消当前调用
  - PENDINGMSG_WAITNOPROCESS - 客户不处理消息
  - PENDINGMSG_WAITDEFPROCESS - 处理部分消息，丢掉其他消息

DCOM安全模型

- Windows NT安全机制
- 激活安全性
- 调用安全性
- 运行时刻动态设置安全性
- 安全性配置

Windows NT安全模型概念

- domain、user和user group
- authentication，认证
- security identifier，SID
- security description，SD
- Access Control List，ACL

Windows NT安全模型

- 用户和资源管理
  - 认证用户的身份
  - SID描述用户
  - SD描述资源的安全性
- SSP(Security Support Providers)
  - SSPI

激发安全性(launching security)

- 由SCM来控制。决定“哪些用户能够(或者不能够)在激活时刻启动服务器进程”
- 因为此安全性决定了客户是否有启动进程的许可，所以这项安全性不能在程序中控制
- 通过注册表实施launching security
  - 先找到APPID的LaunchPermission设置
  - 机器范围内的缺省设置

访问安全性

- 决定“哪些用户可以与服务器进程的对象进行实际的通信”
- 控制过程
  - 服务进程调用CoInitializeSecurity
  - AppID键中的AccessPermission设置
  - 机器范围内的DefaultAccessPermission设置
  - 隐式调用CoInitializeSecurity，含服务器进程的principal和SYSTEM账号

动态安全性控制

- 若进程调用了CoInitiallizeSecurity函数，则进程不再使用注册表的静态设置

```
HRESULT CoInitializeSecurity(
    PSECURITY_DESCRIPTOR pVoid,        //Points to security descriptor
    DWORD cAuthSvc,                    //Count of entries in asAuthSvc
    SOLE_AUTHENTICATION_SERVICE * , //Array of names to register
    void * pReserved1,                 //Reserved for future use
    DWORD dwAuthnLevel,          //The default authentication level for proxies
    DWORD dwImpLevel,            //The default impersonation level for proxies
    RPC_AUTH_IDENTITY_HANDLE pAuthInfo, // Reserved
    DWORD dwCapabilities,        //Additonal client and/or serverside capabilities
    void * pvReserved2);         //Reserved for future use
```

- IAccessControl接口

动态安全性控制(续)

- CoInitiallizeSecurity支持三种类型设置方式，由pVoid和dwCapabilities参数决定：
  - AppID，转向注册表设置
  - SD，与Win32安全编程模型结合
  - IAccessControl接口，简化安全编程模型
- 认证级别，authentication level
  - 设定本进程的引入或引出对象至少使用该级别
- 模仿级别，impersonation level
  - 客户允许对象能够执行哪些操作，客户可以隐藏自己的身份、禁止对象模仿自己等

IAccessControl接口

- 通过IAccessControl接口可以简化安全编程模型
- 定义了许多数据结构用来描述访问控制
- COM提供了IAccessControl的一个实现，应用程序可以创建其实例，然后设置内部的安全控制信息
- 应用程序也可以自己实现IAccessControl以便更加灵活地控制访问许可

调用安全性，calling security

- 客户通过代理对象的IClientSecurity接口，控制调用安全性，
  - 允许对某个接口代理设置安全性
  - 允许对某个套间的某个接口代理设置安全性
- 组件程序调用CoGetCallContext获得IServerSecurity接口，进一步获得安全信息

激活安全性

- 几个创建函数中包含COSERVERINFO 结构，其中的pAuthInfo 成员指向一个

```
typedef struct _COAUTHINFO {
    DWORD dwAuthnSvc;
    DWORD dwAuthzSvc;
    LPWSTR pwszServerPrincName;
    DWORD dwAuthnLevel;
    DWORD dwImpersonationLevel;
     COAUTHIDENTITY * pAuthIdentityData;
    DWORD dwCapabilities;
} COAUTHINFO;
```

- 激活安全性只影响激活过程，也就是创建过程，当创建完成之后，客户得到的接口指针不受激活安全性的影响

服务器进程的身份

- 决定“服务器进程将运行在哪个用户的身份下”，AppID下的RunAs键
- 三种设置方案：
  - 交互用户，interactive user
  - 启动用户，launching user
  - 指定用户
- 三种方案各有利弊，适用于不同的场合

安全性配置(一)

- 几个注册表名称

  LaunchPermission

  AccessPermission

  RunAs：

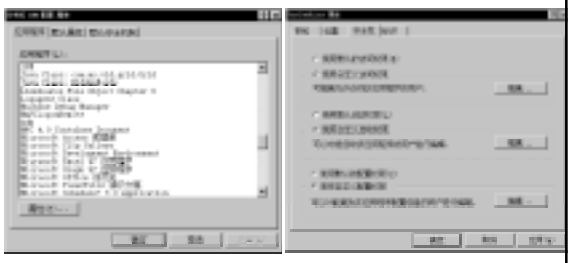
  Run as Activator

  Run as Interactive User

  Run as a fixed user account
- DCOM统一使用的缺省设置

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Ole
   "EnableDCOM"="Y"
   "DefaultAccessPermission" = <self-relative security descriptor>
   "DefaultLaunchPermission" = <self-relative security descriptor>
   "LegacyImpersonationLevel" = <dword_impLevel>
   "LegacyAuthenticationLevel" = <dword_authLevel>
```

安全性配置(二)



安全性配置(三)

小结：DCOM特性

- 可伸缩性
- 可配置性
- 安全性
- 协议无关性
- 平台独立性

开发DCOM组件

- 首先保证本地server成功
  - 注册类厂
  - 为自定义接口编写proxy/stub
- 配置安全性
  - 从本地走向远程的根本变化
- DLL组件，考虑MTS和COM+

DCOM常见问题

- 安全性设置有问题？
- 创建成功，但是QueryInterface其他接口失败，可能接口proxy/stub注册有问题
- 调试
  - /embedding
  - HRESULT
  - RunAs=Interactive User