编写安全的SQLServer扩展存储过程 PDF转换可能丢失图片或格式，建议阅读原文
https://www.100test.com/kao_ti2020/454/2021_2022__E7_BC_96_E5_86_99_E5_AE_89_E5_c98_454845.htm SQL Server 的扩展存储过程，其实就是一个普通的 Windows DLL,只不过按照某种规则实现了某些函数而已。 近日在写一个扩展存储过程时，发现再写这类动态库时，还是有一些需要特别注意的地方。之所以会特别注意，是因为DLL运行于SQL Server的地址空间，而SQL Server到底是怎么进行线程调度的，却不是我们能了解的，即便了解也无法控制。 我们写动态库一般是自己用，即便给别人用，也很少像SQL Server这样，一个动态库很有可能加载多次，并且都是加载到一个进程的地址空间中。我们知道，当一个动态库加载到进程的地址空间时，DLL所有全局与局部变量初始化且仅初始化一次，以后再次调用LoadLibrary函数时，仅仅增加其引用计数而已，那么很显然，假如有一全局 int ,初始化为0，调用一个函数另其自加，此时其值为1,然后再调用LoadLibray,并利用返回的句柄调用输出函数输出该值，虽然调用者觉得自己加载后立即输出，然后该值确实1而不是0。windows是进程独立的，而在线程方面，假如不注意，上面的情况很可能会程序员带来麻烦。 介绍一下我的扩展存储过程，该动态库导出了三个函数:Init,work,Final,Init读文件，存储信息于内存，work简单的只是向该内存检索信息,Final回收内存。如上所说，假如不考虑同一进程空间多次加载问题，两次调用Init将造成无谓的浪费，因为我第一次已经读进了内存，要是通过堆分配内存，还会造成内存泄露。 我使用的引用计数解决的该问题，代码很短

，直接贴上来：#include "stdafx.h"#include using namespace std.extern "C" {RETCODE __declspec(dllexport) xp_part_init(SRV_PROC *srvproc).RETCODE __declspec(dllexport) xp_part_process(SRV_PROC *srvproc).RETCODE __declspec(dllexport) xp_part_finalize(SRV_PROC *srvproc).}#define XP_NOERROR 0#define XP_ERROR 1HINSTANCE hInst = NULL.int nRef = 0.void printError (SRV_PROC *pSrvProc, CHAR* szErrorMsg).ULONG __GetXpVersion(){ return ODS_VERSION.}SRVRETCODE xp_part_init(SRV_PROC* pSrvProc){typedef bool (*Func)().if(nRef == 0){hInst = ::LoadLibrary("part.dll").if(hInst == NULL){printError(pSrvProc,"不能加载part.dll").return XP_ERROR.}Func theFunc = (Func)::GetProcAddress(hInst,"Init").if(!theFunc()){::FreeLibrary(hInst).printError(pSrvProc,"不能获得分类号与专辑的对应表").return XP_ERROR.}} nRef.return (XP_NOERROR).}SRVRETCODE xp_part_process(SRV_PROC* pSrvProc){typedef bool (*Func)(char*).if(nRef == 0){printError(pSrvProc,"函数尚未初始化,请首先调用xp_part_init").return XP_ERROR.}Func theFunc = (Func)::GetProcAddress(hInst,"Get").BYTE bType.ULONG cbMaxLen,cbActualLen.BOOL fNull.char szInput[256] = {0}.if (srv_paraminfo(pSrvProc, 1, amp.cbMaxLen, (ULONG*)amp.fNull) == FAIL){printError(pSrvProc,"srv_paraminfo 返回 FAIL").return XP_ERROR.}szInput[cbActualLen] = 0.string strInput =

szInput.string strOutput = ".".int cur,old = 0.while(string::npos != (cur = strInput.find(’ . ’ ,old))){strncpy(szInput,strInput.c_str() old,cur - old).szInput[cur - old] = 0.old = cur 1.theFunc(szInput).if(string::npos ==strOutput.find((string)"." szInput))strOutput = szInput.}strcpy(szInput,strOutput.c_str()).if (FAIL == srv_paramsetoutput(pSrvProc, 1, (BYTE*)(szInput 1), strlen(szInput) - 1,FALSE)){printError (pSrvProc, "srv_paramsetoutput 调用失败").return XP_ERROR.}srv_senddone(pSrvProc, (SRV_DONE_COUNT | SRV_DONE_MORE), 0, 0).return XP_NOERROR.}SRVRETCODE xp_part_finalize(SRV_PROC* pSrvProc){typedef void (*Func)().if(nRef == 0)return XP_NOERROR.Func theFunc = (Func)::GetProcAddress(hInst,"Fin").if((--nRef) == 0){theFunc().::FreeLibrary(hInst).hInst = NULL.}return (XP_NOERROR).} 我想虽然看上去不是很高明，然而问题应该是解决了的。 还有一点说明，为什么不使用Tls，老实说，我考虑过使用的，因为其实代码是有一点问题的，假如一个用户调用xp_part_init,然后另一个用户也调用xp_part_init，注意我们的存储过程可是服务器端的，然后第一个用户调用xp_part_finalize，那么会怎样，他仍然可以正常使用xp_part_process，这倒无所谓，然而第一个用户调用两次xp_part_finalize，就能够影响第二个用户了，他的xp_part_process将返回错误。 使用Tls 似乎可以解决这问题，例如再添加一个tls_index变量，调用 TlsSetValue保存用户私人数据，TlsGetValue检索私人数据，当xp_part_init时，假如

该私人数据为0，执行正常的初始化过程，（即上面的xp_part_init）执行成功后存储私人数据为1，假如是1，直接返回，xp_part_finalize时，假如私人数据为1，则执行正常的xp_part_finalize，然后设私人数据为0，假如是0，直接返回。好像想法还是不错的，这样隔离了多个用户，安全性似乎提高了不少，然而事实是不可行的。因为Tls保存的并不是私人数据，而是线程本地变量，我们不能保证一个用户的多次操作都是用同一个线程执行的，这个由SQL Server自己控制，事实上我在查询分析器里多次执行的结果显示，SQL Server内部似乎使用了一个线程池。既然如此，那这种想法也只能作罢。