**KeepRunning(Ex).7z**

**目录**

[什么是KeepRunning？](#什么是KeepRunning)

[如何构造KeepRunning？](#如何构造KeepRunning)

[如何破解KeepRunning？](#如何破解KeepRunning)

[如何使用KeepRunningEx?](#如何使用KeepRunningEx)

[重要声明](#重要声明)

**什么是KeepRunning?**

压缩包中封装了应用程序非正常终止的重启功能。假设进程A是一个家长监管程序，开机自动启动，需要密码才能退出。可是你知道的，除了360那些强大的安全软件，没什么进程不能被任务管理器终止，欲关闭监管程序，用任务管理器终止便罢！我们还得给监管程序加几条安全绳！这就是KeepRuning!虽然KeepRunning不能让程序返回执行被终止前正在执行的语句，但对于监管系列程序来说，这已经足够了！

**如何构造KeepRunning?**

如果让你来给程序加上KeepRunning功能，你会怎么做？

作者本人的思路如下：创建一个进程等待程序被终止，然后使用CreateProcess重启程序。

KeepRunning.7z就是这么做的！为了让代码尽可能地重用，我把可能被终止的程序(A部分)、等待程序终止并重启程序（B部分）分为两部分。

A部分链接一个DLL(Running.dll)，后者负责启动B部分进程

B部分等待A终止，并使用CreateProcess重启A

好吧，基本思路建立起来了，但是还有一些细节是需要注意的！比如CreateProcess要求传入A部分的路径，所以我们还要保存这个路径，B部分要等待A部分，那么就必须获得A部分的句柄！KeepRunning.7z把这些工作都在A部分中做了，当然，KeepRunning.7z的做法有两个明显的缺点，正因为这样，才诞生了Ex版本。

我们来看看KeepRunning.7z的做法：

A部分：

1. 声明一个结构：

struct MyProcess

{

HANDLE hProcess;

TCHAR szPath[MAX\_PATH];

};

1. 调用GetModuleHanle(…)保存A部分的路径到szPaht中。
2. 创建B部分进程
3. 调用DuplicateHandle复制A部分进程句柄到B部分，并把复制后的句柄保存到hProcess中。
4. 使用WriteProcessMemory把整个MyProcess结构写入到B部分【更早期的版本使用内存映射文件共享整个结构】
5. 使用内存映射文件共享MyProcess结构的地址（在B部分进程中）

B部分：

1. 使用内存映射文件获得MyProcess结构的地址
2. 等待A部分终止（句柄保存在MyProcess结构hProcess中）
3. 重启A部分（A部分路径保存在MyProcess结构szPath中）
4. B部分终止

KeepRunning的大致做法就是这样，但我们的Running.dll还需要开放一个接口给调用者，这两个接口就是：

VOID KeepRunning();

VOID CloseProcess();

调用KeepRunning会完成上述的A部分的工作。

那么CloseProcess()的工作方式如何？以下是该函数的工作方式：

1. 使用TerminateProcess终止B部分进程
2. 调用ExitProcess终止A部分进程

这就是KeepRunning.7z封装的操作，相信聪明的读者已经发现了很多漏洞！首先，每一个A部分就对应了一个B部分，这浪费资源，而且A部分做了过多的操作，使得A部分的初始化时间长，用户难以接受！最严重的是，CloseProcess调用了ExitProcess!我们的C++析构函数不会被调用！这还不算严重吗？

所有这些问题，都在Ex版本中得到解决！所以：

**应使用Ex版本，保留非Ex版本只是为了保留一种思路！作者本人已不再提供非Ex版本支持。**

以下是Ex版本的做法（依然分为A、B两部分）

A部分：

1. 检查B部分是否已生成，否的话就生成B部分，同时使用事件内核对象等待B部分创建完毕，并把B部分主线程Id保存到共享变量g\_dwKeepRunThreadId中。

#pragma data\_seg("Shared")

DWORD g\_dwKeepRunThreadId=NULL;

#pragma data\_seg()

#pragma comment(linker,"/SECTION:Shared,RWS")

检查g\_dwKeepRunningThreadId是否为NULL，若为NULL，生成B部分

鉴于此，我强烈推荐读者把KeepRunningEx放到Windows目录，这样就能高度共享Running.dll暨KeepRunning.exe

1. 调用PostThreadMessage给B部分主线程发送消息【WM\_APP】(wParam=100)，并在lParam参数中传入GetCurrentProcessId()
2. 欲退出A部分，调用PostThreadMessage给B部分主线程发送消息【WM\_APP】(wParam=101)，并在lParam参数中传入GetCurrentProcessId()
3. 程序做清理工作，并选择自己的方式推出进程【最好是通过主线程main/WinMain函数返回】

B部分：

1. 触发事件，A部分继续运行
2. 把主线程Id保存到全局变量dwMainThreadId中
3. while(GetMessage()等待A部分消息
4. 若接受到A部分的消息【wParam=100】，创建一个线程等待A部分终止，具体做法：
5. 调用OpenProcess并传入SYNCHRONIZE | PROCESS\_QUERY\_LIMITED\_INFORMATION权限，进程Id来自A部分
6. 调用QueryFullProcessImageName获得A部分进程路径
7. 等待A部分终止：

UINT WINAPI WaitProcessCallback(LPVOID p)

{

MSG\* pMsg=(MSG\*)p;

HANDLE hProcess =

OpenProcess(SYNCHRONIZE | PROCESS\_QUERY\_LIMITED\_INFORMATION /\*等待终止 | QueryFullProcessImageName\*/,FALSE,(DWORD)pMsg->lParam);

TCHAR szPath[MAX\_PATH];

DWORD cbSize=\_countof(szPath);

QueryFullProcessImageName(hProcess,0,szPath,&cbSize);

DWORD dwWait = WaitForSingleObjectEx(hProcess,INFINITE,TRUE);

switch (dwWait)

{

case WAIT\_IO\_COMPLETION:

//nothing to do here.

break;

case WAIT\_OBJECT\_0:

STARTUPINFO si={sizeof(si)};

PROCESS\_INFORMATION pi;

CreateProcess(0,szPath,0,0,FALSE,0,0,0,&si,&pi);

break;

}

**if(InterlockedDecrement(&g\_nCount)==0) { //递减使用次数，当递减为0，退出进程**

**CloseHandle(hProcess);**

**PostThreadMessage(dwMainThreadId,WM\_QUIT,0,0);//退出程序**

**}**

return GetLastError();

}

等待中把线程置为可提醒状态，这是为了完美退出而设的。注意加粗部分，若没有需要等待的进程，那么B部分就必须退出【调用PostThreadMessage这是一种较高明的做法，因为这样可以从主函数返回，C/C++运行库可以做一些清理操作，B部分也易于扩展】。

若B部分受到消息【wParam=101】表明A部分需要取消KeepRun功能，方便退出，B部分就会在监视进程列表中删除A部分的进程Id：

VOID WINAPI APCFunc(ULONG\_PTR) {/\*nothing to do here;\*/}

int APIENTRY \_tWinMain(\_In\_ HINSTANCE hInstance,

\_In\_opt\_ HINSTANCE hPrevInstance,

\_In\_ LPTSTR lpCmdLine,

\_In\_ int nCmdShow)

{

UNREFERENCED\_PARAMETER(hPrevInstance);

UNREFERENCED\_PARAMETER(lpCmdLine);

// TODO: 在此放置代码。

**CEvent g\_CreateEvent(0,FALSE,FALSE,\_T("KeepRunningCreateEvent"));**

g\_CreateEvent.SetEvent();

dwMainThreadId = GetCurrentThreadId();//保存主线程Id

MSG msg;

while (GetMessage(&msg,NULL,0,0)/\*等待消息\*/)

{

if(msg.message==WM\_APP) {

if(msg.wParam==KEEPRUNNING) {

InterlockedIncrement(&g\_nCount);//递增使用次数

Map\* pMap=new Map;

pMap->ProcessId=msg.lParam;

pMap->WaitThread = chBEGINTHREADEX(0,0,WaitProcessCallback,&msg,0,0);

MapList.AddToTail(pMap);

}

else if(msg.wParam==CANCELKEEP){

for(Map\* pMap=MapList.GetFirst();pMap!=NULL;pMap=MapList.GetNext()) {

if(pMap->ProcessId==(DWORD)msg.lParam) {

**QueueUserAPC(APCFunc,pMap->WaitThread,0);**

MapList.EmptyNode(pMap);

}

}

}

}

}

return GetLastError();

}

其中Map是我定义的一个结构：

struct Map {

DWORD ProcessId;

HANDLE WaitThread;//正在等待进程终止的线程

};

代码中CList,CEvent,都是我写的类【不是MFC的类】，可以帮助我封装很多操作，所有我写的类与其它比较有用（不是我写的）的类都在我云盘分享的CommonFiles目录，这些文件可以简化我的编程。

代码中还用到两个宏：

#define KEEPRUNNING 100

#define CANCELKEEP 101

这在A部分与B部分都有定义。

注意WinMain函数加粗部分，这就是CancelKeepRunning操作，等待函数会返回WAIT\_IO\_COMPTION.没错，就是你刚刚看到的那样！正在等待的线程其实什么也不做，只是退出也已（当然也会在退出前做清理操作）

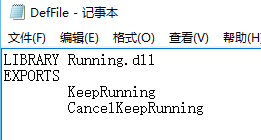
讲到清理操作，不要以为我没有关闭g\_CreateEvent,其实类的析构函数会自动关闭它（我是个健忘的人，所以会让类的析构函数来帮我调用CloseHandle）

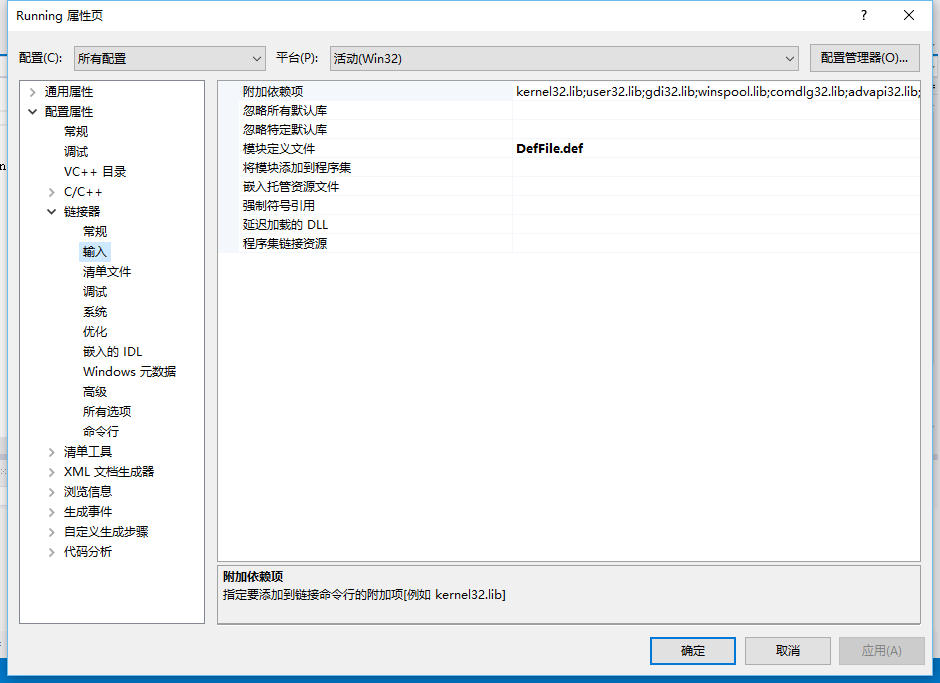
KeepRunningEx.7z的思路与细节已经暴露在你眼前了！这个压缩包的Running.dll开放了两个函数接口:

VOID KeepRunning();

VOID Cancel KeepRunning();

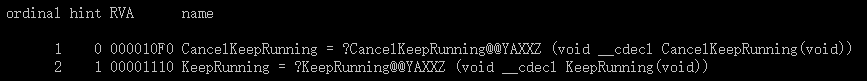
你觉得我们的讨论完整吗？让我再来帮你问一个问题，如果开发者使用动态链接，那么GetProcAddress链接的符号名是什么？编译器会改编符号名！所以我们必须使用Def文件来导出这两个函数，并把Def文件附加到项目的模块定义文件!





至于隐式链接，就根本不用担心这个问题啦（当然，如果你不是使用Microsoft的编译器，这个问题还是需要考虑的）

我们用dumpbin工具来看看Running.dll的导出段（为了篇幅，我裁剪了无关片段）：



如果不使用Def文件的话，GetProcAddress的第二个参数要填入“=”后面的符号，你现在知道Def文件的重要性了吧！

**思路我们讨论完了，接下来是比较轻松的：如何使用？**

如果刚刚的思路讨论你已经清楚了，那么这里不言自明了！只要链接到Running.dll调用dll导出的那两个函数就可以了【非Ex版本调用KeepRunning与CloseProcess ；Ex版本以上已说明】

当然，让我们把注意力放在提高性能上面，Ex版本让我们可以一个KeepRunning.exe(B部分)监视多个程序(A部分)，这大大节省了资源，所以我们应该让Ex版本高度共享，把它放在Windows目录或System32目录是比较好的做法，这样，无论多少个程序需要KeepRun功能，只需要启动一个KeepRunning.exe，非第一个启动的程序不必创建KeepRunning.exe进程，初始化时间大大缩短！

**如何强制停止KeepRun?**

请注意“强制”字眼！也就是破解KeepRun功能，这一节是为破解党准备的！KeepRun思路的讨论已经让我们知道怎么做了！我们用任务管理器关闭KeepRunning.exe进程，那么结束A部分就简单了！本例是这样，其他程序也是类似的做法，逃不出我们的思路讨论！当然，安全软件的思路更为复杂，我们的思路讨论不针对他们。