本博客不再更新，已经搬迁至http://kosmisch.net/

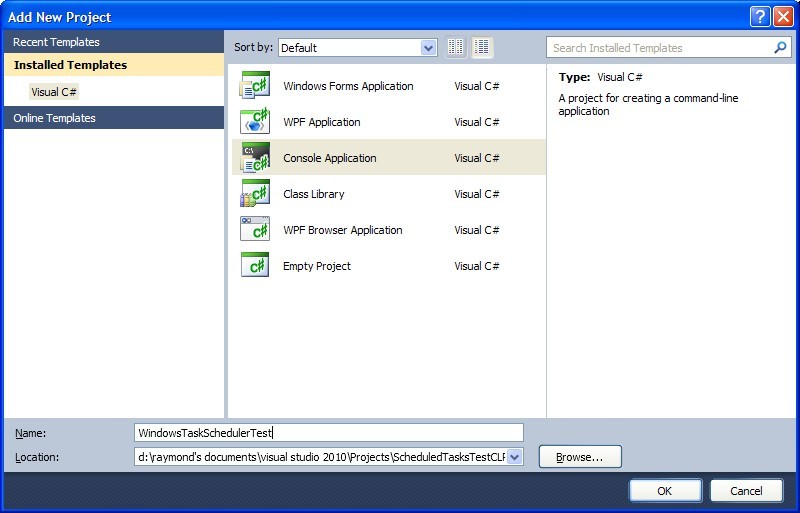
2011-01-26 11:31

**[C#] .NET调用Windows Scheduled Tasks 计划任务**

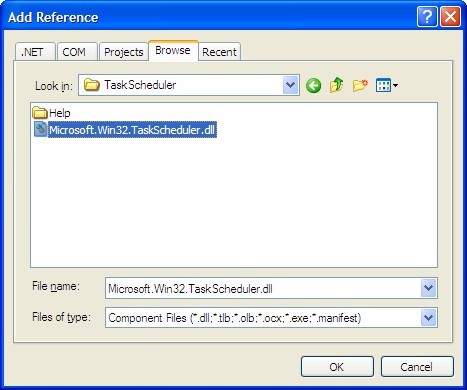
  
Windows Scheduled Tasks 计划任务可以定期执行任务，并且可以循环执行，在一些系统中，使用计划任务比我们创建Windows 服务，在其中使用Timer定时器来执行周期性任务更加的方便。微软针对各个系统平台开放了计划任务MSTASK的API，C/C++接口请参看MSDN 文档：[http://msdn.microsoft.com/zh-cn/library/aa446802](http://msdn.microsoft.com/zh-cn/library/aa446802" \t "_blank)。遗憾的是在.NET中并没有对这些接口进行封装，不过在CodePlex上，有开源的计划任务接口封装，详情参考：[Task Scheduler Managed Wrapper](http://taskscheduler.codeplex.com/)。

本文将介绍如何使用Windows计划任务的托管的封装接口，请首先[下载](http://taskscheduler.codeplex.com/releases/view/40083" \t "_blank)此项目。

**1,创建测试项目**



项目添加对Microsoft.Win32.TaskScheduler 的引用：



在Main方法中添加代码，运行根据用户输入执行相应的操作：添加计划任务、查询计划任务的状态、删除计划任务：

using System;  
using System.Text;  
using Microsoft.Win32.TaskScheduler;  
  
  
namespace WindowsTaskSchedulerTest  
{  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            ShowNavigationMsg();  
            var input = "";  
            var exit = false;  
            while (exit == false && string.IsNullOrEmpty(input = Console.ReadKey().KeyChar.ToString().ToLower()) == false)  
            {  
                switch (input)  
                {  
                    case "a":  
                        CreateTestTask();  
                        break;  
                    case "b":  
                        RetrieveTestTask();  
                        break;  
                    case "c":  
                        DeleteTestTask();  
                        break;  
                    case "d":  
                        exit = true;  
                        break;  
                    default:  
                        ShowNavigationMsg();  
                        break;  
                }  
                  
            }  
        }  
  
        private static void DeleteTestTask()  
        {  
            ShowNavigationMsg();  
        }  
  
        private static void RetrieveTestTask()  
        {  
            ShowNavigationMsg();  
        }  
  
        private static void CreateTestTask()  
        {  
            ShowNavigationMsg();  
        }  
  
        static void ShowNavigationMsg()  
        {  
            Console.WriteLine();  
            Console.WriteLine("Please choose your action by input a charactor:");  
            Console.WriteLine("[a]Add Test Task       [b]Retrieve Test Task Status ");  
            Console.WriteLine("[c]Delete Test Task    [d]Eixt");  
        }  
    }  
}

**2,创建计划任务**

创建计划任务需要使用到以下类：

TaskService-计划任务服务；

TaskDefinition-定义一个计划任务；

DailyTrigger-Trigger中的一种，用于触发计划任务，触发器还包括Weekly、Monthly等等：

// Summary:  
    //     Defines the type of triggers that can be used by tasks.  
    public enum TaskTriggerType  
    {  
        // Summary:  
        //     Triggers the task when a specific event occurs. Version 1.2 only.  
        Event = 0,  
        //  
        // Summary:  
        //     Triggers the task at a specific time of day.  
        Time = 1,  
        //  
        // Summary:  
        //     Triggers the task on a daily schedule.  
        Daily = 2,  
        //  
        // Summary:  
        //     Triggers the task on a weekly schedule.  
        Weekly = 3,  
        //  
        // Summary:  
        //     Triggers the task on a monthly schedule.  
        Monthly = 4,  
        //  
        // Summary:  
        //     Triggers the task on a monthly day-of-week schedule.  
        MonthlyDOW = 5,  
        //  
        // Summary:  
        //     Triggers the task when the computer goes into an idle state.  
        Idle = 6,  
        //  
        // Summary:  
        //     Triggers the task when the task is registered. Version 1.2 only.  
        Registration = 7,  
        //  
        // Summary:  
        //     Triggers the task when the computer boots.  
        Boot = 8,  
        //  
        // Summary:  
        //     Triggers the task when a specific user logs on.  
        Logon = 9,  
        //  
        // Summary:  
        //     Triggers the task when a specific user session state changes. Version 1.2  
        //     only.  
        SessionStateChange = 11,  
    }

此枚举对应计划任务中的类型；

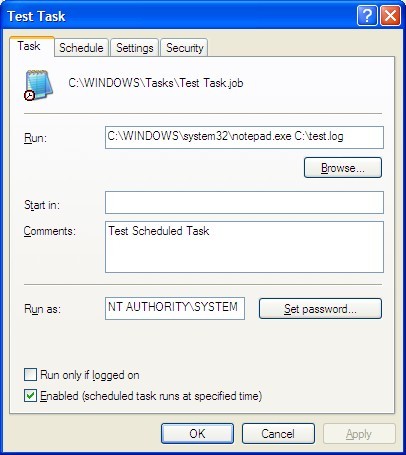
ExecAction-执行操作，定义计划任务需要执行的操作；

以及其它相关类。

修改CreateTestTask方法为：

private static void CreateTestTask()  
        {  
            // 计划任务服务  
            TaskService ts = new TaskService();  
  
            // 创建一个计划任务  
            TaskDefinition td = ts.NewTask();  
            td.RegistrationInfo.Description = "Test Scheduled Task";  
            td.RegistrationInfo.Author = "Raymond";  
              
            //每天启动  
            Trigger dt = new DailyTrigger{DaysInterval = 1};  
            //限制计划任务每次执行时间不能超过1小时  
            dt.ExecutionTimeLimit = TimeSpan.FromHours(1);  
            dt.StartBoundary = DateTime.Parse("12:00 AM");  
            //全天重复执行  
            dt.Repetition.Duration = TimeSpan.FromHours(24);  
            //每隔30分钟重复一次  
            dt.Repetition.Interval = TimeSpan.FromMinutes(30);  
            //将触发器添加到计划任务td的触发器中  
            td.Triggers.Add(dt);  
  
            //创建一个执行操作  
            var exe = new ExecAction("notepad.exe", "C:\\test.log", null);  
            //添加执行操作到计划任务的操作中  
            td.Actions.Add(exe);  
  
            //注册计划任务  
            ts.RootFolder.RegisterTaskDefinition(@"Test Task", td);  
  
            ShowNavigationMsg();  
        }

运行，输入a后，可以看到计划任务中以及增加了我们添加的计划任务，并且默认以系统账户System运行：



**2,查询计划任务状态**

修改RetrieveTestTask方法为：

private static void RetrieveTestTask()  
        {  
            // 计划任务服务  
            TaskService ts = new TaskService();  
            Task t = ts.GetTask(@"Test Task");  
            if (null != t)  
            {  
                Console.WriteLine("Task Name={0}", t.Name);  
                Console.WriteLine("Task Execution Time={0}", t.LastRunTime);  
                Console.WriteLine("Task Last Run Result={0}", t.LastTaskResult);  
                Console.WriteLine("Task Next Execution Time={0}", t.NextRunTime);  
            }  
            ShowNavigationMsg();  
        }

运行输入b，可以得到以下结果：

Please choose your action by input a charactor:  
[a]Add Test Task       [b]Retrieve Test Task Status  
[c]Delete Test Task    [d]Eixt  
bTask Name=Test Task  
Task Execution Time=1/1/0001 12:00:00 AM  
Task Last Run Result=0  
Task Next Execution Time=1/26/2011 11:30:00 AM  
  
Please choose your action by input a charactor:  
[a]Add Test Task       [b]Retrieve Test Task Status  
[c]Delete Test Task    [d]Eixt

除此之外还可以通过Task.Definition 获取添加计划任务时的相关信息以及其它信息。

**3,删除计划任务**

修改DeleteTestTask方法为：

private static void DeleteTestTask()  
        {  
            // 计划任务服务  
            TaskService ts = new TaskService();  
            ts.RootFolder.DeleteTask("Test Task");  
            ShowNavigationMsg();  
        }

运行输入c，在控制面板的计划任务Scheduled Tasks中，Test Task已经被删除了。

**4,其它说明**

这个第三方组件，还提供了很多的对象，以及相关的方法，可以让我们方便的对计划任务进行管理，具体请大家在实践中探索，官方也有简单的文档：[http://taskscheduler.codeplex.com/documentation](http://taskscheduler.codeplex.com/documentation" \t "_blank)

[**C# Task 任务计划**](http://www.cnblogs.com/Googler/archive/2010/06/05/1752213.html)

http://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

using System;  
using System.Threading;  
using System.Collections.Generic;  
  
namespace Rocky  
{  
 #region Task  
 public class Task : IDisposable  
 {  
 #region Fields  
 private string taskName;  
 private Timer timer;  
 private TimerCallback execTask;  
 private ISchedule schedule;  
 private DateTime lastExecuteTime;  
 private DateTime nextExecuteTime;  
 #endregion  
  
 #region Properties  
 /// <summary>   
 /// 任务名称   
 /// </summary>   
 public string Name  
 {  
 set { taskName = value; }  
 get { return taskName; }  
 }  
 /// <summary>   
 /// 执行任务的计划   
 /// </summary>   
 public ISchedule Shedule  
 {  
 get { return schedule; }  
 }  
 /// <summary>   
 /// 该任务最后一次执行的时间   
 /// </summary>   
 public DateTime LastExecuteTime  
 {  
 get { return lastExecuteTime; }  
 }  
 /// <summary>   
 /// 任务下执行时间   
 /// </summary>   
 public DateTime NextExecuteTime  
 {  
 get { return nextExecuteTime; }  
 }  
 #endregion  
  
 #region Methods  
 /// <summary>   
 /// 构造函数   
 /// </summary>   
 /// <param name="schedule">为每个任务制定一个执行计划</param>   
 public Task(TimerCallback callback, ISchedule schedule)  
 {  
 if (callback == null || schedule == null)  
 {  
 throw new ArgumentNullException();  
 }  
 this.execTask = callback;  
 this.schedule = schedule;  
 execTask += new TimerCallback(Execute);  
 TaskScheduler.Register(this);  
 }  
  
 /// <summary>   
 /// 任务内容   
 /// </summary>   
 /// <param name="state">任务函数参数</param>   
 private void Execute(object state)  
 {  
 lastExecuteTime = DateTime.Now;  
 if (schedule.Period == Timeout.Infinite)  
 {  
 nextExecuteTime = DateTime.MaxValue; //下次运行的时间不存在   
 }  
 else  
 {  
 TimeSpan period = new TimeSpan(schedule.Period \* 1000);  
 nextExecuteTime = lastExecuteTime + period;  
 }  
 if (!(schedule is CycExecution))  
 {  
 this.Close();  
 }  
 }  
  
 public void Start()  
 {  
 Start(null);  
 }  
 public void Start(object execTaskState)  
 {  
 if (timer == null)  
 {  
 timer = new Timer(execTask, execTaskState, schedule.DueTime, schedule.Period);  
 }  
 <;span style="color: #0000ff;">else  
 {  
 timer.Change(schedule.DueTime, schedule.Period);  
 }  
 }  
 public void RefreshSchedule()  
 {  
 if (timer != null)  
 {  
 timer.Change(schedule.DueTime, schedule.Period);  
 }  
 }  
 public void Stop()  
 {  
 if (timer != null)  
 {  
 timer.Change(Timeout.Infinite, Timeout.Infinite);  
 }  
 }  
  
 public void Close()  
 {  
 ((IDisposable)this).Dispose();  
 }  
  
 void IDisposable.Dispose()  
 {  
 if (execTask != null)  
 {  
 taskName = null;  
 if (timer != null)  
 {  
 timer.Dispose();  
 timer = null;  
 }  
 execTask = null;  
 TaskScheduler.Deregister(this);  
 }  
 }  
  
 public override string ToString()  
 {  
 return taskName;  
 }  
 #endregion  
 }  
 #endregion  
  
 #region TaskScheduler  
 /// <summary>   
 /// 任务管理中心   
 /// 使用它可以管理一个或则多个同时运行的任务   
 /// </summary>   
 public static class TaskScheduler  
 {  
 private static List<Task> taskScheduler;  
  
 public static int Count  
 {  
 get { return taskScheduler.Count; }  
 }  
  
 static TaskScheduler()  
 {  
 taskScheduler = new List<Task>();  
 }  
  
 /// <summary>  
 /// 查找任务  
 /// </summary>  
 /// <param name="name"></param>  
 /// <returns></returns>  
 public static Task Find(string name)  
 {  
 return taskScheduler.Find(task => task.Name == name);  
 }  
  
 public static IEnumerator<Task> GetEnumerator()  
 {  
 return taskScheduler.GetEnumerator();  
 }  
  
 /// <summary>   
 /// 终止任务   
 /// </summary>   
 public static void TerminateAllTask()  
 {  
 lock (taskScheduler)  
 {  
 taskScheduler.ForEach(task => task.Close());  
 taskScheduler.Clear();  
 taskScheduler.TrimExcess();  
 }  
 }  
  
 internal static void Register(Task task)  
 {  
 lock (taskScheduler)  
 {  
 taskScheduler.Add(task);  
 }  
 }  
 internal static void Deregister(Task task)  
 {  
 lock (taskScheduler)  
 {  
 taskScheduler.Remove(task);  
 }  
 }  
 }  
 #endregion  
  
 #region ISchedule  
 /// <summary>   
 /// 计划立即执行任务   
 /// </summary>   
 public struct ImmediateExecution : ISchedule  
 {  
 public DateTime ExecutionTime  
 {  
 get { return DateTime.Now; }  
 set { }  
 }  
 public long DueTime  
 {  
 get { return 0; }  
 }  
 public long Period  
 {  
 get { return Timeout.Infinite; }  
 }  
 }  
  
 /// <summary>   
 /// 计划在某一未来的时间执行一个操作一次，如果这个时间比现在的时间小，就变成了立即执行的方式   
 /// </summary>   
 public struct ScheduleExecutionOnce : ISchedule  
 {  
 private DateTime schedule;  
  
 public DateTime ExecutionTime  
 {  
 get { return schedule; }  
 set { schedule = value; }  
 }  
 /// <summary>   
 /// 得到该计划还有多久才能运行   
 /// </summary>   
 public long DueTime  
 {  
 get  
 {  
 long ms = (schedule.Ticks - DateTime.Now.Ticks) / 10000;  
 if (ms < 0)  
 {  
 ms = 0;  
 }  
 return ms;  
 }  
 }  
 public long Period  
 {  
 get { return Timeout.Infinite; }  
 }  
  
 /// <summary>   
 /// 构造函数   
 /// </summary>   
 /// <param name="schedule">计划开始执行的时间</param>   
 public ScheduleExecutionOnce(DateTime time)  
 {  
 schedule = time;  
 }  
 }  
  
 /// <summary>   
 /// 周期性的执行计划   
 /// </summary>   
 public struct CycExecution : ISchedule  
 {  
 private DateTime schedule;  
 private TimeSpan period;  
  
 public DateTime ExecutionTime  
 {  
 get { return schedule; }  
 set { schedule = value; }  
 }  
 public long DueTime  
 {  
 get  
 {  
 long ms = (schedule.Ticks - DateTime.Now.Ticks) / 10000;  
 if (ms < 0)  
 {  
 ms = 0;  
 }  
 return ms;  
 }  
 }  
 public long Period  
 {  
 get { return period.Ticks / 10000; }  
 }  
  
 /// <summary>   
 /// 构造函数,马上开始运行   
 /// </summary>   
 /// <param name="period">周期时间</param>   
 public CycExecution(TimeSpan period)  
 {  
 this.schedule = DateTime.Now;  
 this.period = period;  
 }  
 /// <summary>   
 /// 构造函数，在一个将来时间开始运行   
 /// </summary>   
 /// <param name="shedule">计划执行的时间</param>   
 /// <param name="period">周期时间</param>   
 public CycExecution(DateTime shedule, TimeSpan period)  
 {  
 this.schedule = shedule;  
 this.period = period;  
 }  
 }  
  
 /// <summary>   
 /// 计划的接口   
 /// </summary>   
 public interface ISchedule  
 {  
 /// <summary>   
 /// 返回最初计划执行时间   
 /// </summary>   
 DateTime ExecutionTime { set; get; }  
 /// <summary>   
 /// 初始化执行时间于现在时间的时间刻度差   
 /// </summary>   
 long DueTime { get; }  
 /// <summary>   
 /// 循环的周期   
 /// </summary>   
 long Period { get; }  
 }  
 #endregion  
}

# 网站： [http://haha808.com/?p=2383](http://haha808.com/?p=2383%20)

# C# Task Scheduler Managed Wrapper: 由程式来排程工作

发表于 [2011 年 07 月 28 日](http://haha808.com/?p=2383) 由 [admin](http://haha808.com/?author=1)

# Task Scheduler Managed Wrapper: 由程式来排程工作

|  |  |
| --- | --- |
| Codeplex 软体套件（Package）资讯 | |
| 套件名称 | Task Scheduler Managed Wrapper |
| 作者 | Coordinators: dahall  Developers: gserfaty, SteveHarveyUK |
| 目前版本 | 1.4.1 Stable |
| URL | [http://taskscheduler.codeplex.com/](http://translate.googleusercontent.com/translate_c?rurl=translate.google.com.hk&sl=auto&tl=zh-CN&twu=1&u=http://taskscheduler.codeplex.com/&usg=ALkJrhhdO412ZvYQPmGk0gcNqu1B0pTaQw) |
| 使用难易度 | 中 |
| 使用此套件时可用的辅助工具 | 工作排程器（Task Scheduler）  Task Scheduler Managed Wrapper 的文件 |
| 基础知识 | 基本类别使用。  基础COM Interop。 |

## 排程工作：开发人员必备的知识

大多数的开发人员也许都会有这么一段经历：『老板要我把常用的报表在每周一上班之前寄到他的Email』、『我想要在半夜时重新计算资料库以产生报 表』或是『工作A 如果在上班时段做，会让系统overload，要拿到半夜来做，但我又不想顾到半夜…』等，这些可能会在非工作时间，或是要定时处理的工作（ job），通常作业系统都会提供工具来支援它，以Windows 来说，就是Task Scheduler（工作排程器），也就是常听到的at（命令列化的Task Scheduler）指令。

**NOTE**

这里所指的at 指令，不是指数据机（Modem）在用的那种海斯指令集（Hayes Command Set; AT Command Set），读者可以想像它是@（英文字的at）的意思，表示工作要@ 何时执行。

**NOTE**

Unix/Linux 上也有这样子的工具，名称是cron。

排程工作在网管和系统管理的领域用的非常多，所有与系统与网路有关的周期性工作，都会交由排程工作来做，当然，资料库管理以及每段固定时间（周／月 ／季／半年／年等）需要执行的统计工作，也多由排程工具来做（以SQL Server 为例，就是SQL Server Agent），它可以把许多的工作自动化起来，让人员可以在同一时间去做别的事情。 因此，除了网管和系统工程师要学会如何操作它，开发人员也应该要学着 如何使用，尤其是商用系统的开发人员。

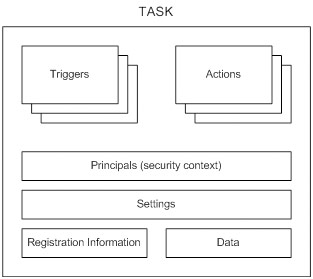
在Windows 95 的Plus! Pack 时期，微软加入了一个排程执行工具，称为mstask.exe，此即Task Scheduler 1.0，并且在Windows 98 开始内建在作业系统中，Windows NT 与之后的更新版本（ Windows 2000, XP, 2003 等）将这个工具以Windows 服务的方式来执行，并且在控制台放置一个功能区，作为GUI 编辑的介面，不过它只有开放C/C++ 的COM 介面给应用程式存取，对于Visual Basic 以及支援COM Automation 的程式语言或平台未提供支援，这点对于非C/C++ 的开发人员而言似乎不是一件好事，连.NET Framework 也未内建此类支援，仍然要由C/C++ 来存取。 所幸，Windows Vista 开始，Task Scheduler 升级为2.0 版，这个版本开放了COM 自动化介面的API 供开发人员使用，同时也允许1.0 时期的C/C++ 介面存取2.0 版本的函式库功能以保持相容性。

不过在Windows Vista 还没有出现之前，就已经有人试着将C/C++ 的COM 介面控制方式，在.NET Framework 上实现，所幸在COM 互通功能（Interoperability）的强力支援下，这个功能成功的被实现出来，也就是本文所要介绍的Task Scheduler Managed Wrapper Library。

## Task Scheduler 基本概念

Task Scheduler 是由一个或数个Task（工作）组成，一个Task 代表一个任务，它会有几个参数：

|  |  |
| --- | --- |
| 参数 | 使用者介面 |
| 触发器（trigger），指示此工作会在何时触发。 在Task Scheduler 2.0 时，一个工作可以有多个触发器。 1.0 时则每个工作只能有一个触发器。 | http://i.msdn.microsoft.com/ee922578.task_scheduler001%28zh-tw,MSDN.10%29.jpg |
| 动作（action），指示此工作会触发哪些程式或指令。 在Task Scheduler 2.0 时，一个工作可以有多个动作（最多32 个动作）。 1.0 时则每个工作只能有一个动作。 | http://i.msdn.microsoft.com/ee922578.task_scheduler002%28zh-tw,MSDN.10%29.jpg |
| 执行权限（Principals, Security Context），指示此工作要用哪个帐户或安全权限执行，所有动作都会使用相同的权限来呼叫。 | http://i.msdn.microsoft.com/ee922578.task_scheduler003%28zh-tw,MSDN.10%29.jpg |
| 条件（Settings），可以决定工作在何种情况下才会启动。 | http://i.msdn.microsoft.com/ee922578.task_scheduler004%28zh-tw,MSDN.10%29.jpg |
| 登录资料（Registration Information），记录系统管理的相关资讯，像是建立者（author）或工作说明（description）。 | http://i.msdn.microsoft.com/ee922578.task_scheduler005%28zh-tw,MSDN.10%29.jpg |
| 资料（Data），此工作所需要的额外资讯。 |  |

  
图：Task 内部结构图（来源：MSDN Task Scheduler SDK）

Task Scheduler 的动作支援四种类型，分别为启动程式、自动发信、显示讯息以及COM 自订处理器（handler），其中只有COM 自订处理器一项必须要由程式来做以外，其他三种均可以透过使用者介面设定。

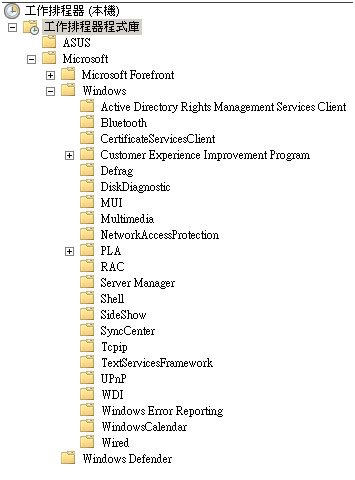
触发器则2.0 和1.0 的支援不太相同，列表如下：

|  |  |  |  |
| --- | --- | --- | --- |
| 触发器 | 功能 | 2.0 | 1.0 |
| Event trigger | 以指定事件为触发条件。 | √ |  |
| Time trigger | 以指定日期时间为触发条件（在1.0 版称为Once Trigger）。 | √ | √ |
| Daily trigger | 每天触发（以日历为主）。 | √ | √ |
| Weekly trigger | 每周触发（以日历为主）。 | √ | √ |
| Monthly trigger | 每月触发（以日历为主）。 | √ | √ |
| Monthly day-of-week (DOW) trigger | 以day-of-week方式触发，像是每周的第几天；每月的第几周或是每年的第几个月。 | √ | √ |
| Idle trigger | 在系统闲置时触发（在1.0 版称为OnIdle Trigger）。 | √ | √ |
| Registration trigger | 在工作​​登录或更新时触发。 | √ |  |
| Boot trigger | 在开机时触发（在1.0 版称为System Start Trigger）。 | √ | √ |
| Logon trigger | 在登入时触发。 | √ | √ |
| Session state change trigger | 在Terminal Service 的会谈状态变更时触发。 | √ |  |

**NOTE**

在Task Scheduler 2.0 使用者介面中看到的触发器类型，可能会不止表中所列的那几项，像是『锁定工作站时触发』以及『解除工作站锁定时触发』均不在上表中，但事实上它们都是Event Trigger 的一种，只是已经有固定的参数了。

Task Scheduler 2.0 中新增资料夹阶层的功能，有助于将各个工作分类存放，但Task Scheduler 1.0 只支援单​​一根资料夹（root folder），因此在资料夹使用上要特别注意相容性的问题。

  
图：Task Scheduler 2.0 资料夹阶层功能

## Task Scheduler Managed Wrapper Library

Task Scheduler Managed Wrapper Library 最早是在Code Project 网站中的一个专案，由David Hall 于2002 年所开发，当时还只是.NET Framework 1.0 的时代，在Code Project 的专案内容中，Da​​vid 说明了如何由Platform SDK所附的mstask.idl，使用Tlbimp.exe 产生引用的C# 程式码再移植到他的专案中的方法。 里面使用了许多的COM Interoperability 的技巧，值得学习COM Interop 的开发人员作为参考。

**NOTE**

通常要探知COM 介面的原始结构，由SDK 所附的标头档（.h 档）以及介面定义清单档（Interface Definition List, IDL，即\*.idl）就可以得到许多COM 的介面宣告，包括定义、方法、参数型别、属性清单、常数宣告以及列举项（值）等都可以由这里获得，因此若要开发COM/API 桥接到.NET Framework 的程式时，API 宣告的标头档以及COM 的介面清单档都是重要的参考资料。

Task Scheduler Managed Wrapper Library 可以使用在.NET 的任何一个版本，并且可同时支援Task Scheduler 1.0 以及2.0 版本，开发人员也可以选用微软所提供的Task Scheduler API 来存取Task Scheduler 2.0 版。

使用方式很简单，只要在它的网站中下载TaskScheduler.zip 并解压缩，即可得到Microsoft.Win32.TaskScheduler.dll（核心元件）、TimeSpan2.dll（特制的 TimeSpan 物件函式库）以及Microsoft.Win32 .TaskSchedulerEditor.dll（使用者介面元件）三个档案，引用时需要引用 Microsoft.Win32.TaskScheduler.dll 以及TimeSpan2.dll 两个档案，若要使用到它的使用者介面函式库，则还要再引用Microsoft.Win32.TaskSchedulerEditor.dll。

下列范例程式为简单操作Task Scheduler API 的范例：

[C#]

1. using System; using Microsoft.Win32.TaskScheduler; class Program { static void Main(string[] args) { // Get the service on the local machine TaskService ts = new TaskService(); // Create a new task definition and assign properties TaskDefinition td = ts.NewTask(); td.RegistrationInfo.Description = "Does something"; // Create a trigger that will fire the task at this time every other day td.Triggers.Add(new DailyTrigger { DaysInterval = 2 }); // Create an action that will launch Notepad whenever the trigger fires td.Actions.Add(new ExecAction("notepad.exe", "c:\\test.log", null)); // Register the task in the root folder ts.RootFolder.RegisterTaskDefinition(@"Test", td); // Remove the task we just created ts.RootFolder.DeleteTask("Test"); } }

Task Scheduler Wrapper 的起点是TaskService 类别，它提供了管理与列举资料夹的功能，以及取得工作以及连线状态的功能，并且也可以在建构时设定连结到指定的伺服器。

TaskFolder 物件是处理资料夹的物件，它具有下列功能：

* 建立与删除子资料夹（CreateFolder 与DeleteFolder 方法）。
* 登录工作（RegisterTaskDefinition 方法）以及删除工作（DeleteTask 方法）。
* 取得子工作清单（Tasks 属性）。
* 取得子资料夹清单（SubFolders 属性）。
* 设定与取得Security Descriptior 的SDDL Form 资讯（GetSecurityDescriptorSddlForm 与SetSecurityDescriptotSddlForm 方法）。

它的使用方法也很简单，例如下列程式可以建立与删除资料夹：

[C#]

1. // create folder. TaskService service = new TaskService(); TaskFolder folder = service.GetFolder(@"\"); folder.CreateFolder(this.txtFolderName.Text, folder.GetSecurityDescriptorSddlForm(AccessControlSections.Access)); folder = null; service = null;

[C#]

1. // create folder. TaskService service = new TaskService(); TaskFolder folder = service.GetFolder(@"\"); folder.CreateFolder(this.txtFolderName.Text, folder.GetSecurityDescriptorSddlForm(AccessControlSections.Access)); folder = null; service = null;

[C#]

1. // delete folder. this.\_taskService = new TaskService(); TaskFolder folder = this.\_taskService.GetFolder(@"\MyFolder"); if (folder.Tasks.Count > 0) { throw new InvaildOperationException("TASKS\_EXIST"); } if (folder.SubFolders.Count > 0) { throw new InvaildOperationException("SUB\_FOLDERS\_EXIST"); } TaskFolder folderParent = this.\_taskService.GetFolder(folder.Path.Replace(@"\" + folder.Name, "")); folderParent.DeleteFolder(folder.Name); folderParent = null; folder = null;

**NOTE**

Task Scheduler 的根资料夹为“\”，且Task Scheduler 1.0 只支援根资料夹，不支援建立资料夹的任何作业。

**NOTE**

TaskFolder.CreateFolder() 需要的第二个参数，可以直接由父资料夹的GetSecurityDescriptorSddlForm() 方法来取得它的Security Descriptor SDDL Form 作为安全设定的基础。

**NOTE**

在Task Scheduler 2.0 上执行删除资料夹的作业时，即使会成功，也会掷出一个NotV1SupportedException，这是程式码上的小瑕疪，可以利用try/catch 的方式来避开，或是直接修改原始程式码，加上版本的处理来解决它。

Task 物件则是代表工作，它的实际设定为TaskDefintion 物件（由Task.Defintion 属性取得），而它的执行状态以及设定值，可直接由相关的属性得知：

* Enabled 设定工作是否启用。
* LastRunTime 以及LastRunResult 可以取得最后一次执行的时间与结果。
* NextRunTime 为下一次执行时间。
* State 可取得目前工作的状态。
* NumberOfMissedRuns 可得知被遗漏排程的次数。
* Run() 与RunEx() 可以执行工作。
* Stop() 可停止工作的执行。

TaskDefinition 物件装载了Task 的基本设定，以及其包含的Action、Trigger、RegistrationInfo 与Settings 等物件的设定（可参考前面的说明），Actions 和Triggers 都是集合物件，可用Add() 来加入特定的Action 或Trigger，或用AddNew() 来建立新的Action 或Trigger，再设定它们的属性值。

可用的Action 均继承自Action 物件，支援的Action 清单如下：

|  |  |  |
| --- | --- | --- |
| 动件 | 功能 | 物件名称 |
| Email Action | 发送Email（1.0 不支援）。 | EmailAction |
| Show Message Action | 显示讯息（1.0 不支援）。 | ShowMessageAction |
| Execution Action | 执行应用程式、批次档或指令。 | ExecAction |
| COM Handler Action | 执行自订的COM 处理器（1.0 不支援）。 | ComHandlerAction |

例如建立一个Email 的Action 的程式如下：

[C#]

1. EmailAction eAction = new EmailAction("Task fired", "sender@email.com", "recipient@email.com", "You just got a message", "smtp.company.com"); eAction.Cc = "alternate@email.com";

可用的Trigger 均继承自Trigger 物件，支援的Triggers 清单如下：

|  |  |  |
| --- | --- | --- |
| 触发器 | 功能 | 物件名称 |
| Event trigger | 以指定事件为触发条件（1.0 不支援）。 | EventTrigger |
| Time trigger | 以指定日期时间为触发条件（在1.0 版称为Once Trigger）。 | TimeTrigger |
| Daily trigger | 每天触发（以日历为主）。 | DailyTrigger |
| Weekly trigger | 每周触发（以日历为主）。 | WeeklyTrigger |
| Monthly trigger | 每月触发（以日历为主）。 | MonthlyTrigger |
| Monthly day-of-week (DOW) trigger | 以day-of-week 方式触发，像是每周的第几天；每月的第几周或是每年的第几个月。 | MonthlyDOWTrigger |
| Idle trigger | 在系统闲置时触发（在1.0 版称为OnIdle Trigger）。 | IdleTrigger |
| Registration trigger | 在工作​​登录或更新时触发（1.0 不支援）。 | RegistrationTrigger |
| Boot trigger | 在开机时触发（在1.0 版称为System Start Trigger）。 | BootTrigger |
| Logon trigger | 在登入时触发。 | LogonTrigger |
| Session state change trigger | 在Terminal Service 的会谈状态变更时触发（1.0 不支援）。 | SessionStateChangeTrigger |

例如设定每月执行的触发器的程式如下：

[C#]

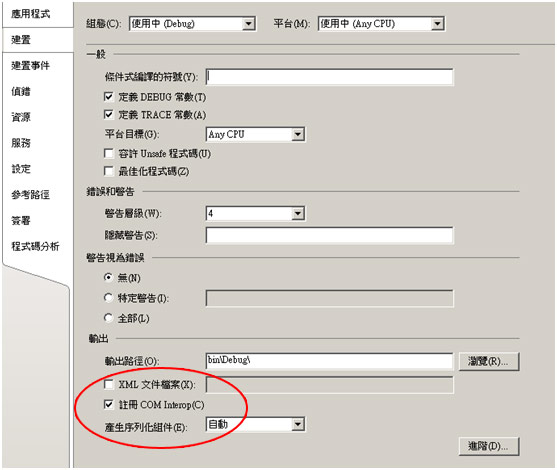
1. MonthlyTrigger mTrigger = new MonthlyTrigger(); mTrigger.StartBoundary = DateTime.Today + TimeSpan.FromHours(10); mTrigger.DaysOfMonth = new int[] { 10, 20 }; mTrigger.MonthsOfYear = MonthsOfTheYear.July | MonthsOfTheYear.November;

## 进阶功能：COM 处理器

这是Task Scheduler 2.0 才有的新功能，允许开发人员自行建立一个支援Task Scheduler COM 工作的元件，让Task Scheduler 帮你执行，虽然它名为COM 处理器，但是.NET 程式可以利用COM Interop 方式将元件公开给COM 介面，如此就可​​以当做一个COM 元件供外部存取。 另外，COM 处理器的工作只可以在程式码中实作，无法透过GUI 介面处理，因此这个能力算是保留给开发人员使用的，也就是需要额外撰写登录COM 工作的程式码并在安装程式使用（或是撰写成独立的小程式亦可）。

要建立COM 处理器，请执行下列步骤：

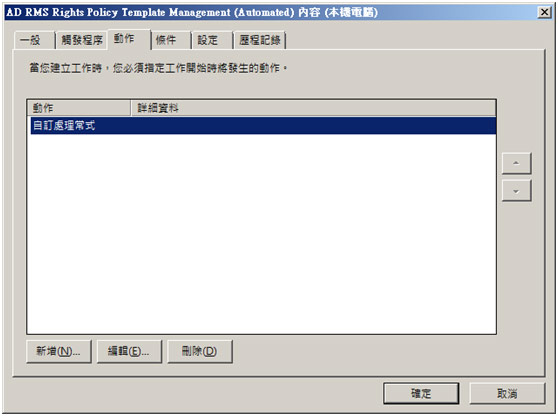
1. 建立一个类别库专案。
2. 加入Microsoft.Win32.TaskScheduler.dll 以及TimeSpan2.dll 的参考。
3. 建立一个新类别，并设定与实作Microsoft.Win32.TaskScheduler 命名空间的ITaskHandler 介面。
4. 设定专案要产生Type Library，即注册COM Interop：



1. 建置专案，并撰写注册COM Handler工作的程式码，例如下列的程式码：

[C#]

1. ComHandlerAction comAction = new ComHandlerAction(new Guid("{CE7D4428-8A77-4c5d-8A13-5CAB5D1EC734}")); comAction.Data = "Something specific the COM object needs to execute. This can be left unassigned as well.";
2. 登录完成的工作，可以在动作中看到『自订处理常式』，代表是COM Handler，而且这个动作是无法被编辑的：



ITaskHandler 介面具有四个方法，分别是Start()、Pause()、Resume() 以及Stop() 四个方法，代表启动、暂停、继续以及​​停止四个动作，由Task Scheduler 视情况呼叫控制。 其中Start() 方法会传入Data 参数，这个参数是注册COM Handler 时所给定的Data 属性值，作为执行时的参数，而Stop() 方法会需要传回一个RetCode，代表成功或失败，若执行成功应传回0，若出现非零值时会被判断是失败。

**NOTE**

RetCode 参数的COM 宣告是HRESULT 值，因此参数设为0时，会被转换成ERROR\_SUCCESS，代表执行成功，若是非零值时，会被转译成错误的资讯，并写入执行记录中。

下列程式码是COM Handler 的范例实作，功能为每五秒于事件记录中写入一笔资料，会写入12 次：

1. [ComVisible(true), Guid("CE7D4428-8A77-4c5d-8A13-5CAB5D1EC734"), ClassInterface(ClassInterfaceType.None)] public sealed class MyCOMTask : ITaskHandler { private ITaskHandlerStatus handlerService; private Timer timer; private DateTime lastWriteTime = DateTime.MinValue; private byte writeCount = 0; private const string file = @"C:\TaskLog.txt"; void ITaskHandler.Start(object pHandlerServices, string Data) { handlerService = pHandlerServices as ITaskHandlerStatus; lastWriteTime = DateTime.Now; timer\_Elapsed(null, null); timer.Enabled = true; } void ITaskHandler.Stop(out int pRetCode) { timer.Enabled = false; pRetCode = 0; } void ITaskHandler.Pause() { timer.Enabled = false; } void ITaskHandler.Resume() { timer.Enabled = true; } public MyCOMTask() { timer = new Timer(5000) { AutoReset = true }; timer.Elapsed += new ElapsedEventHandler(timer\_Elapsed); } void timer\_Elapsed(object sender, ElapsedEventArgs e) { if (writeCount < 12) { try { using (StreamWriter wri = File.AppendText(file)) wri.WriteLine("Log entry {0}", DateTime.Now); handlerService.UpdateStatus((short)(++writeCount / 12), string.Format("Log file started at {0}", lastWriteTime)); } catch { } } if (writeCount >= 12) { timer.Enabled = false; writeCount = 0; handlerService.TaskCompleted(0); } } }

## 结语

Task Scheduler Wrapper Library 是相容性高（可用在1.0 和2.0）的一组工具函式库，并直接控制Task Scheduler 的功能，以实现能利用程式码去存取与控制工作排程的能力，不必再额外去撰写命令列指令或是使用GUI 工具去设定排程了，读者可以善用它来开发自己的工具，像是Web 化的排程管理员或集中的管理工具等。