Подключение и использование OverScript в своем .NET-проекте

admin
February 17 edited March 18

Всё просто:
1. Добавляем в проект ссылку на OverScript.dll;
2. Пишем `using OverScript;` ;
3. Создаём экземпляры Script и Executor;
4. Запускам выполнение скрипта методом Execute.

```
Script script = new Script(@"G:\scripts\test.txt");
Executor executor = new Executor(script);
//можно сразу: Executor executor = new Script(@"G:\scripts\test.txt");
executor.Execute(); //запускает и ждёт завершения исполнения
executor.Dispose(); //если executor больше не будем использовать, то нужно освободить ресурсы
```

При создании Script можно передать делегат изменения статуса загрузки:

```
static void Main(string[] args)
{
    Executor executor = new Script(@"G:\scripts\test.txt", OnLoadingProgress);
    //Loading step 0 / 5
    //Loading step 1 / 5
    //Loading step 2 / 5
    //Loading step 3 / 5
    //Loading step 4 / 5
    //Loading step 5 / 5
    //Loading completed.
```

```
    executor.Execute();
    executor.Dispose();
}
static void OnLoadingProgress(object sender, int step)
{
    if (step < 0) Console.WriteLine("Loading completed.");
    else Console.WriteLine($"Loading step {step} / {Script.LoadingSteps}");
}
```

Перехват исключения:

```
Executor executor = new Script(@"G:\scripts\test.txt");
try {
    executor.Execute();
}
catch(Exception ex)
{
    Exception e = ex.Data.Contains(ExceptionVarName.StackTrace) ? (Exception)Activator.CreateInstance(ex.GetType(), ex.Message + Environment.

    throw e;
}
```

Запускать исполнение одного executor-а в нескольких потоках нельзя, т.к. они будут использовать одно хранилище переменных. Запускать несколько раз тоже нельзя:

```
Executor executor = new Script(@"G:\scripts\test.txt");
executor.Execute();
executor.Execute(); //Re-execution is not supported.
```

При каждом вызове Execute создаётся новый экземпляр главного класса скрипта (это класс, в который автоматически оборачивается весь код). Запрет на повторное исполнение связан с тем, что в памяти могут оставаться объекты (включая главный экземпляр), и при

их дальнейшей финализации могут повредиться текущие переменные, т.к. при новом исполнении будет использоваться то же хранилище переменных.

В Execute можно передать аргументы (string[]):

```
executor.Execute(new string[] { "hello", "world", "test" });
```

Получить эти аргументы в скрипте можно функцией StartArgs(): `WriteLine(Join('|', StartArgs()));`.
Есть возможность задания значений глобальных свойств. OverScript код в G:\scripts\test.txt:

```
int x=int\GetGlobal("x");
SetGlobal("x", x*2);
```

C#:

```
Executor executor = new Script(@"G:\scripts\test.txt");
executor["x"] = 5; //или так: executor.SetGlobal("x", 5);
executor.Execute();
Console.WriteLine("x="+ executor["x"]); //x=10
executor.Dispose();
```

Можно **вызывать методы** в скрипте. Скрипт:

```
string ConcatStr(string s, string s2){
    return s+s2;
}
```

C#:

```
Executor executor = new Script(@"G:\scripts\test.txt");
executor.Execute(); //сначала нужно, чтобы был создан экземпляр главного класса
```

```
Console.WriteLine(executor.Call("ConcatStr","hello ", "world")); //hello world
executor.Dispose();
```

Если нужно часто вызывать, то лучше получить из скрипта делегат и вызывать через него (будет быстрее работать). Скрипт:

```
object GetDelegate(){
    return Delegate(ConcatStr("", ""), typeof("System.Func`3[System.String,System.String,System.String]"));
}

string ConcatStr(string s, string s2){
    return s+s2;
}
```

C#:

```
Executor executor = new Script(@"G:\scripts\test.txt");
executor.Execute();
var concatStr = (Func<string, string, string>)executor.Call("GetDelegate");
Console.WriteLine(concatStr("hello ","world")); //hello world
executor.Dispose();
```

Максимальное кол-во одновременно существующих экзекуторов ограничено по умолчанию 64. Его можно задать так: `ExecPool.Capacity = 128;` . Сделать это можно только один раз и до создания экзекуторов. ExecPool - это пул экзекуторов. При создании экзекутора, он добавляется в этот пул, а при вызове Dispose удаляется из него. Технически это нужно для хранилища переменных. У каждого экзекутора есть ID (индекс в пуле), который является индексом в массиве массивов хранилища.

Теперь про **отмену исполнения**. Допустим, вы запускаете исполнение в отдельном потоке. Если вызвать Dispose экзекутора, то происходит принудительная отмена. При этом ожидания фактического окончания выполнения не делается. Если произойдёт обращение к хранилищу переменных, то произойдёт исключение ScriptExecutionException. Но скорее всего будет выбрасываться ExecutingForciblyCanceledException (наследует от ExecutingCanceledException).
Отменить с ожиданием завершения выполнения можно методом **Cancel**: `executor.Cancel(true);` . Если без true, то будет делаться не принудительная, а мягкая отмена. При принудительной не выполняются блоки catch/finally, методы Dispose и финализаторы (методы

Finalize), а при мягкой выполняются (но если в Dispose() вызываются другие пользовательские функции, то выдаст исключение). При мягкой выбрасывается ExecutingCanceledException. У мягкой отмены есть ещё одно важное отличие от принудительной: она срабатывает только при переходе к следующей логической строке кода. Принудительная же может обрывать выполнение выражений. Пример:

```
int n;
While(true, Write((n++)+" "), Sleep(1000));
//0 1 2 3 4 5 6 7...
```

Функция While выполняет аргументы пока первый true. Её выполнение может прервать только принудительная отмена. Мягкая может прервать только такой вариант:

```
int n;
while(true){Write((n++)+" "); Sleep(1000);}
```

Я написал в одну строку, но это несколько логических строк:

```
int n;
while(true){ //инструкция while
    Write((n++)+" "); //вызов функции
    Sleep(1000);  //вызов функции
} //конец while
```

Мягкая отмена (выброс ExecutingCanceledException) произойдёт на одной из этих четырёх строк.

Важно понимать, что любая отмена не может прервать выполнение низкоуровневых операций, которые выполняются не на уровне OverScript-кода. Если поток находится в состоянии WaitSleepJoin (ThreadState.WaitSleepJoin - поток заблокирован), то отмена сработает только при разблокировании потока.

После отмены нельзя вызывать методы через Call. Если метод уже запущен либо вызван через делегат, то он завершится с исключением. Cancel ждёт завершения выполнения метода Execute, но не Call и делегатов! Способа ожидания завершения любых выполнений пока нет.

Я уже писал выше, что мягкая остановка позволяет выполняться финализаторам (пока не вызван Dispose экзекутора). На самом деле позволяется выполняться не только финализаторам, а всем потокам с наивысшим приоритетом (финализаторы выполняются именно в таком потоке). Т.е. если вы запустите свой поток с приоритетом ThreadPriority.Highest, то мягкая отмена на него не повлияет. При принудительной отмене финализаторы пропускаются без выброса исключения.
Исключения в финализаторе по умолчанию не игнорируются (выбрасываются через AppDomain.CurrentDomain.UnhandledException). Чтобы их подавлять, нужно задать: `executor.IgnoreFinalizerExceptions = true;` . Если исключения включены, то будет FinalizationFailedException с вложенным конкретным исключением.

Если вы используете финализаторы, которые должны делать что-то важное, то они должны быть выполнены до вызова Dispose или принудительной отмены. Для этого нужно вызвать сборку мусора и дождаться её окончания:

```
Executor executor = new Script(@"G:\scripts\test.txt");
executor.Execute();
GC.Collect();
GC.WaitForPendingFinalizers();
executor.Dispose();
```

Покажу ещё пример с исключением в финализаторе. Скрипт:

```
WriteLine("Start...");
new Foo("Test"); //создаём экземпляр, который потом будет уничтожаться сборщиком мусора
Sleep(5000);
WriteLine("Text that will not be shown");

class Foo{
    string Str;
    New(string str){
        Str=str;
    }
    Finalize(){
        WriteLine("Finalize! "+Str);
        5/0; //эта строка вызовет исключение
    }
}
```

C#:

```
using System;
using System.Threading;
using System.Threading.Tasks;
using OverScript;

namespace OSTestForConsole
{
    class Program
    {
        static void Main(string[] args)
        {
            AppDomain.CurrentDomain.UnhandledException += new UnhandledExceptionEventHandler(OnUnhEx);

            Executor executor = new Script(@"G:\scripts\test.txt");
            Task t = Task.Run(() => executor.Execute());
            Thread.Sleep(2000);
            Console.WriteLine("Cancel...");
            executor.Cancel();
            Console.WriteLine("Garbage collection...");
            GC.Collect();
            GC.WaitForPendingFinalizers();
            Console.WriteLine("End");
            Console.ReadKey();

        }

        static void OnUnhEx(object sender, UnhandledExceptionEventArgs args)
        {
            Exception e = (Exception)args.ExceptionObject;
            Console.WriteLine($"Unhandled exception occured ({e.GetType()}): "+ e.Message+" INNER: "+e.InnerException?.Message);
            Console.ReadKey();
        }
    }
}
```

Результат будет такой:

> Start...
> Cancel...
> Garbage collection...
> Finalize! Test
> Unhandled exception occured (OverScript.FinalizationFailedException): An error occurred during finalization. INNER: Attempted to divide by zero.

Этот пример демонстрирует, что исключение в финализаторе (Finalize) выбрасывается как необработанное исключение. Исключение не будет выбрасываться, если перед запуском исполнения добавить: `executor.IgnoreFinalizerExceptions = true;` . Если делать принудительную отмену (`executor.Cancel(true);`), то финализаторы вообще не будут выполняться (сразу завершаться без какого-либо исключения).
Кстати, если вместо `new Foo("Test");` написать `Foo f=new Foo("Test");` , то на экземпляр останется ссылка в хранилище переменных, а значит сборка мусора его не затронет. Перед сборкой мусора можно обнулить все переменные главного экземпляра (корневые переменные) так: `executor.ClearRootVars();` . Тогда ссылок на созданные объекты не останется, и они будут уничтожены.

Литералы всех скриптов хранятся в едином хранилище. Если у вас в нескольких скриптах есть, например, строка "Hello world!", то храниться она будет в единственном экземпляре. Сбросить (удалить) все литералы можно так: `Literals.Reset();` .

В процессе исполнения могут создаваться объекты (экземпляры пользовательских классов и вообще любые) требующие утилизации (вызова Dispose()). При отмене исполнения эти объекты будут существовать, пока существуют ссылки на них в хранилище переменных. Хранилище уничтожается при вызове Dispose() экзекутора, но объекты будут существовать, пока их не уничтожит сборщик мусора. Не факт, что у всех объектов есть финализатор, который правильно освободит ресурсы, к тому же часто нужно, чтобы это делалось сразу, а не при сборке мусора. У тех объектов, у которых есть метод Dispose() (реализуют интерфейс IDisposable), отвечающий за очистку, можно вызвать его так: `executor.DisposeStored();` . Это сработает для всех IDisposable в хранилище переменных. Но если вы до этого принудительно отменили исполнение, то Dispose() экземпляров пользовательских классов не будет выполняться. Т.е. если в вашем скрипте есть класс Foo с прописанным Dispose(), то этот Dispose() экземпляра Foo после отмены будет выдавать исключение. Если отмена была мягкая, то Dispose() будет срабатывать, если он не вызывает других пользовательских функций (пользовательские - ваши функции в скрипте). Чтобы Dispose() вызывался независимо от способа произведённой ранее

отмены, нужно сделать так: `executor.DisposeStored(true);`. Так, на время очистки, будет снят режим отмены, и все Dispose() смогут выполниться как обычно, без каких-либо ограничений.

Ещё раз проговорю для закрепления:
1. После принудительной отмены выполнение любого кода будет выдавать исключение.
2. Если во время мягкой отмены выполняется код в блоке try, то catch и finally будут выполнены. Если выполняется код метода Dispose(), то он тоже будет выполнен, но вызов им других пользовательских функций выбросит исключение.
3. Финализаторы будут выполняться только при мягкой отмене. Они могут вызывать любые другие функции, т.к. поток будет иметь высший приоритет, а мягкая отмена не влияет на такие потоки. При принудительной выполнение кода финализаторов пропускается без выброса исключения.
4. Если вы сделали мягкую отмену, а до этого передали экземпляр пользовательского класса в стороннюю программу, то при попытке вызова функций (через Call или интерфейсы) этого экземпляра возникнет исключение. Будет разрешён только вызов Dispose() без вызова других функций. Когда ссылки на экземпляр не станет, то сборщик мусора вызовет финализатор (не сразу, а когда посчитает нужным).
5. Если вы передали в стороннюю программу делегат на функцию в вашем скрипте, то после отмены делегаты при вызове тоже будут выдавать исключение.
6. DisposeStored(true) на время снимает режим отмены, что позволяет выполнить Dispose() всех экземпляров пользовательских классов. При этом открывается нежелательная возможность внешних вызовов (пункты 4, 5), которые могут повредить состояние экземпляра или вернуть некорректный результат. Поэтому при работе с OverScript-объектами (CustomObject, CustomType) в сторонних программах стоит проверять статус экзекутора: `objFromScript.ExecutorIsValid`. Или можно заранее передавать ссылку на экзекутор и проверять: `executor.IsValid`. IsValid возвращает true, если экзекутор действующий и его статус Running или Completed.