Использование потоков, задач и таймеров

admin
January 21 edited March 8

В OverScript потоки, задачи и таймеры могут быть обычными и локальными. Обычные запускают функцию, а локальные выражение.
Простой пример:

```
object t=Task(WriteLine(""), "Hello!"); //функция Task берёт ссылку на функцию в первом аргументе и создаёт задачу с этой ссылкой и аргументом "Hello!"
t.StartTask(); //запуск задачи. Будет выполнена WriteLine с аргументом "Hello!".
//Hello!
ReadKey();
```

Это была обычная задача. А это локальная:

```
object t=LocalTask(WriteLine("Hello!")); //в LocalTask просто указывается выражение
t.StartTask();
//Hello!
ReadKey();
```

Локальные потоки, задачи и таймеры выполняют выражение в контексте функции, в которой они созданы. Они могут менять локальные переменные.

```
string s;
object t=LocalTask(s="Hello!");
t.StartTask();
t.Wait(); //ожидание завершения выполнения задачи
WriteLine("s="+s); //s=Hello!
ReadKey();
```

А теперь - внимание!

```
Test();
ReadKey();
//будет выведено "s=", а не "s=Hello!"

Test(){
    string s="Hello!";
    object t=LocalTask(WriteLine("s="+s));
```

```
    t.StartTask();
}
```

Видим, что хоть в s задано "Hello!", WriteLine в локальной задаче выводит не то, что ожидаем. Это потому, что задача (или конкретно чтение s) была выполнена после завершения функции Test(), когда переменная s уже очищена. Т.е. переменная s в выражении `WriteLine("s="+s)` ссылается на область памяти, использованную функцией Test(), но в ней уже ничего нет (на самом деле, очищаются только переменные ссылочных типов). Кстати, задача может быть выполнена и до завершения функции, и тогда будет выведено "s=Hello!". После старта задачи заранее точно не известно, когда она реально запустится и когда завершится.
Вывод: при использовании локальных потоков, задач и таймеров, если они используют локальные переменные, то нужно дожидаться их завершения там, где они были созданы. С экземплярными переменными проще: их экземпляр удерживается пока существует поток|задача|таймер.
Есть функции RunTask, RunThread, RunTimer, которые принимают выражение и автоматически упаковывают в него все локальные переменные.

```
Test(); //s=Hello!
ReadKey();

Test(){
    string s="Hello!";
    object t=RunTask(WriteLine("s="+s));
}
```

То же самое можно так:

```
Test(); //s=Hello!
ReadKey();

Test(){
    string s="Hello!";
    var e=Expr(WriteLine("s="+s)).BoxAll(true); //BoxAll упаковывает все значения, а true указывает, что только для локальных переменных
    object t=LocalTaskByExpr(e);
    t.StartTask();
}
```

Подробно о всех функциях

Начнём с обычных потоков

```
object t=Thread(Test(string\, int\)); //указывать тип нужно со слешем. string\ - это как default(string) в C#.
t.StartThread("Hello", 123); //аргументы для Test передаются именно тут (с задачами по-другому)
t.Wait();
ReadKey();
//str=Hello; x=123;

Test(string str, int x){
```

```
    WriteLine($"str={str}; x={x};");
}
```

В первой строке создаётся поток с функцией Test. Ссылка на неё берётся автоматически, но можно и так:

```
object fn=FuncRef(Test(string\, int\));
object t=ThreadByFuncRef(fn);
```

В StartThread() вычисляются значения всех передаваемых аргументов, и потом изменение значений в переменных не повлияет на значения в потоке.
И Thread и ThreadByFuncRef принимают опциональные параметры name и isBackground, которыми можно задать имя потока и тип (фоновый или нет).

```
object fn=FuncRef(Test(string\, int\));
object t=ThreadByFuncRef(fn, "TestThread", true);
WriteLine(t.ThreadName()+"; "+t->IsBackground); //TestThread; True
```

Важно запомнить, что функции Thread первым аргументом нужно передавать функцию, но не ссылку на функцию. Так короче писать. Этот аргумент-функция не вычисляется (вызова не происходит), а нужен только для получения ссылки на функцию.

Теперь про локальные потоки
При создании локального потока нужно указывать выражение, которое будет выполнено в потоке.

```
object t=LocalThread(Test("It works!", 777));
t.StartThread(); //аргументы не нужны
//t->Start(); //можно и так запускать. С нелокальными потоками не получится.
t.Wait();
ReadKey();
//str=It works!; x=777;

Test(string str, int x){
    WriteLine($"str={str}; x={x};");
}
```

Тут выражение - вызов функции Test, но оно может быть любым.

```
int a=2, b=3, c;
object t=LocalThread(c=a+b); //локальный поток запишет значение в переменную
t.StartThread();
t.Wait();
WriteLine("c="+c);
ReadKey();
```

Если нужно несколько действий сделать, то используйте безымянную функцию:

```
int a, b, c;
object t=LocalThread(@(a=2, b=3, c=a+b)); //@ - это указание, что нужно вызвать базовую функцию, а имя пустое. Эта функция без имени выполняет все аргументы и возвращает значение
```

Потоки можно прерывать функцией Interrupt (Abort в .NET больше нет). Прерывание происходит только в состоянии WaitSleepJoin.

```
int n;
object t=LocalThread(While(true, Write(++n+" "), Sleep(1000))); //эта функция While то же, что: while(true){Write(++n+" "); Sleep(1000);}, но мы не можем в выражении использовать
t.StartThread();
Sleep(5000);
t.Interrupt(); //после 5 секунд работы потока прерываем его
ReadKey();
//1 2 3 4 5
```

Функция Wait (ожидание завершения потока) используется и для потоков, и для задач. А ещё можно для WaitHandle.
Можно создавать локальные потоки из экспрешена:

```
object e=Expr(WriteLine("OK!"));
object t=LocalThreadByExpr(e);
t.StartThread();
t.Wait();
//OK!
ReadKey();
```

Можно создавать и запускать локальный поток одной функцией RunLocalThread:

```
object t=RunLocalThread(WriteLine("OK!"));
t.Wait();
//OK!
ReadKey();
```

Задачи

Задачи отличаются от потоков тем, что аргументы для вызываемой функции нужно указывать при создании задачи. Значения всех передаваемых аргументов сразу вычисляются, и от своих переменных больше не зависят.

```
object t=Task(UrlEncode("", ""), "Привет", "windows-1251");
t.StartTask(); //t->Start();   - и так можно
```

```
WriteLine(t.TaskResult()); //%cf%f0%e8%e2%e5%f2
ReadKey();
```

Функция TaskResult ждёт завершения потока и возвращает результат. В этом примере возвращается результат функции UrlEncode, которая кодирует строку URL-адреса.
Interrupt нельзя применять к задачам. Задача запускается в потоке, и из кода выполняющегося в потоке можно получить ссылку на текущий поток, а потом прервать его извне:

```
object tp=ValueTuple(null); //ValueTuple создаёт кортеж (System.ValueTuple<>). В данном случае ValueTuple<object>.
object t=Task(Test(object\), tp); //в функцию Test передастся кортеж
t.StartTask();
Sleep(5000);
//в кортеже tp находится ссылка на поток, в котором выполняется Test.
tp->Item1.Interrupt(); //это то же, что и Interrupt(tp->Item1)
//tp->Item1->Interrupt(); //можно и так
WriteLine("end");
ReadKey();
//1 2 3 4 5 end

Test(object tp){
    tp->Item1=Thread(); //Thread() возвращает текущий поток. Теперь в кортеже есть ссылка на него.
    int n;
    while(true){
        Write(++n+" ");
        Sleep(1000);
    }

}
```

Вместо ValueTuple можно использовать, например, массив object[] с одним элементом. Передача же переменной по ссылке (ref) в функцию потока|задачи|таймера не поддерживается.
Переделаем с токеном отмены:

```
object cts=CancellationTokenSource();
object token=cts.CancellationToken();

object t=Task(Test(object\), token);
t.StartTask();
Sleep(5000);
cts.Cancel();
//cts->Cancel(); //можно и так
WriteLine("end");
ReadKey();
//1 2 3 4 5 end
```

```
Test(object token){
    int n;
    while(!token.IsCancellationRequested()){
        Write(++n+" ");
        Sleep(1000);
    }
}
```

Как и потоки, создавать задачи можно из ссылки на функцию:

```
object fn=FuncRef(Test(string\, int\));
object t=TaskByFuncRef(fn);
```

Кстати, если у функции есть ref-параметры, то получить ссылку можно так:

```
object fn=FuncRef(Test(ref string\)); //добавил ref
//object fn=FuncRef(Test(ref "")); //не получится, т.к. "" не переменная, а литерал (пустая строка)
Test(ref string s){}
```

Вообще-то, ref можно ставить только перед переменными, но в преобразовании типа тоже можно, а `ref string\` и есть преобразование ничего в string (т.е. значение по умолчанию). В OverScript ключевое слово ref указывает, что нужно найти именно функцию с ref-параметром. В отличие от C#, его писать не обязательно (если параметр ref, а аргумент не переменная, то при загрузке будет ошибка).

Локальные задачи, как я уже показывал в первых примерах, создаются функциями LocalTask и LocalTaskByExpr. Покажу ещё:

```
string name="Jack";
object t=LocalTask(WriteLine(Format("Hello {0}! Today is {1}.", name, Now()->DayOfWeek)));
t.StartTask();
t.Wait();
//Hello Jack! Today is Saturday.
ReadKey();
```

Из экспрешена (результата функции Expr):

```
int a=5, b=2;
object e=Expr(a+b);
object t=LocalTaskByExpr(e);
t.StartTask();
WriteLine(t.TaskResult()); //7
ReadKey();
```

Как и поток, локальную задачу можно создавать и запускать одной функцией:

```
object t=RunLocalTask(WriteLine("OK!"));
t.Wait();
//OK!
ReadKey();
```

С таймерами то же самое

Таймер - это объект типа System.Timers.Timer.

```
object t=Timer(OnTimedEvent(null, null), 1000);
t.StartTimer();
//SignalTime: 23.01.2022 15:24:46
//SignalTime: 23.01.2022 15:24:47
//SignalTime: 23.01.2022 15:24:48
Sleep(3000);
t.StopTimer();
ReadKey("end");

OnTimedEvent(object source, object e){
    WriteLine("SignalTime: " + e->SignalTime);
}
```

Можно задать параметры enabled и autoReset.

```
object t=Timer(OnTimedEvent(null, null), 1000, true, false); //true - это enabled, a false - autoReset
ReadKey();
//SignalTime: 23.01.2022 15:56:42

OnTimedEvent(object source, object e){
    WriteLine("SignalTime: " + e->SignalTime);
}
```

Таймер сработал один раз потому, что autoReset=false. В OnTimedEvent передаются сам объект таймера и объект типа System.Timers.ElapsedEventArgs.
Таймер срабатывает не дожидаясь завершения выполнения предыдущего вызова. Т.е. если функция выполняемая таймером не успеет завершиться, то появится несколько одновременно выполняемых функций. Чтобы такого не происходило, нужно в Timer() пятым аргументом передать true.

```
object t=Timer(OnTimedEvent(null, null), 1000, true, true, true);
```

Так таймер будет выполнять функцию OnTimedEvent только, если предыдущий вызов завершился. Таймер не приостанавливается, а просто пропускаются вызовы функции. В большинстве случаев нужно именно такое поведение, и это также лучше тем, что интерпретатору не нужно создавать каждый раз новый массив с

аргументами для функции (просто записывает в старый массив новые значения source и e), а значит будет меньше работы сборщику мусора.
Функции таймера можно передавать дополнительные аргументы.

```
int X=555;
string Str="test";
object t=Timer(OnTimedEvent(object\, object\, int\, string\), 1000, true, true, true, X, Str); //дополнительные аргументы x и str указываются в конце
ReadKey();
//SignalTime: 23.01.2022 16:24:59; x=555; str=test
//SignalTime: 23.01.2022 16:25:00; x=555; str=test
//SignalTime: 23.01.2022 16:25:01; x=555; str=test
//...

OnTimedEvent(object source, object e, int x, string s){
    WriteLine("SignalTime: " + e->SignalTime+"; x="+x+"; str="+s);
}
```

Дополнительные аргументы вычисляются при создании таймера и потом на свои переменные не ссылаются.
Как и потоки/задачи таймеры можно создавать из ссылки на функцию.

```
object fn=FuncRef(OnTimedEvent(object\, object\, int\, string\));
object t=TimerByFuncRef(fn, 1000, true, true, true, X, Str);
```

В локальный таймер заряжается выражение:

```
object t=LocalTimer(WriteLine("Hello! "+TickCount()), 1000, true);
ReadKey();
//Hello! 928742687
//Hello! 928743687
//Hello! 928744671
//...
```

Ещё:

```
int X;
object t=LocalTimer(X++, 100, true);
Sleep(500);
WriteLine(X); //5 //за 500 миллисекунд таймер сработал 5 раз
Sleep(200);
WriteLine(X); //7 //за 200 мс таймер сработал 2 раза
t.StopTimer(); //кстати, можно так: t.Dispose();
ReadKey();
```

Локальный таймер можно создавать из экспрешена:

```
object e=Expr(WriteLine(TickCount()));
object t=LocalTimerByExpr(e, 1000, true);
```

Для ленивых: RunTask, RunThread, RunTimer
Если вам не нужно менять значения локальных переменных, можно просто создавать и запускать задачи|потоки|таймеры этими функциями, которые заменяют локальные переменные в принимаемом выражении конкретными значениями.
Пример с задачей:

```
object task=GetTask("Hello!");
WriteLine(task.TaskResult()); //HELLO!
ReadKey();

object GetTask(string s){
    return RunTask(s.ToUpper());
}
```

В этом примере `s.ToUpper()` - это фактически `"Hello!".ToUpper()` (после упаковки локальной переменной s).
Пример с потоком:

```
object thread=GetThread("Hello!");
thread.Wait();
//HELLO!
WriteLine(thread.ThreadName()); //TestThread
WriteLine(thread.ThreadState()); //Stopped
WriteLine(thread.IsAlive()); //False
WriteLine(thread.ManagedThreadId()); //4
ReadKey();

object GetThread(string s){
    return RunThread(WriteLine(s.ToUpper()), "TestThread", true); //"TestThread" - это имя потока, а true - задаёт, то что он фоновый. Это опциональные параметры.
}
```

Теперь создание и запуск таймера:

```
object timer=GetTimer("Hello!");
Sleep(3000);
//HELLO! HELLO! HELLO!
timer.StopTimer();
ReadKey();
```

```
object GetTimer(string s){
    return RunTimer(Write(s.ToUpper()+" "), 1000);
}
```

В RunTimer третьим параметром можно задать autoReset, а четвёртым, пропускать ли вызовы, если предыдущий не завершен.

```
return RunTimer(Write(s.ToUpper()+" "), 1000, true, true);
```

Если вы пытаетесь создать таймер, поток или задачу с упаковкой локальных переменных, а в выражении локальной переменной присваивается значение, то вылетит ошибка:

```
Test();

Test(){
    int x;
    object t=RunTimer(x=123, 1000, false, true);
    //Ошибка: Left operand variable 'x' in an assignment expression cannot be boxed.
}
```

Функций RunTaskByExpr, RunThreadByExpr, RunTimerByExpr нет. Используйте Expr+BoxAll. Пример с таймером:

```
Test();
//TEST
ReadKey();

Test(){
    string s="Test";
    object e=Expr(WriteLine(s.ToUpper())).BoxAll(true);
    object timer=LocalTimerByExpr(e, 1000, true, false);
    //object t=RunTimer(e.Eval(), 1000, false); //или так можно
}
```

Не путайте пакующие Run* функции с RunLocalTask и RunLocalThread, которые ничего не пакуют, а просто создают и сразу запускают задачу/поток.