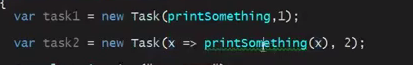


Если нужно использовать таску не единожды то второй способ быстрее так кешируется в methodsgroups

ПЕРЕДАЧА ПАРАМЕТРОВ В TASK



Task

Выполняется на одном из потоков пула потоков

По умолчанию запускается IsBackground false

var action = new **Action**(MyTask); - создаем делегат передавая метод

var task = new Task(action); - ХОЛОДНАЯ ЗАДАЧА Создание экземпляра задачи. Передаем ссылку на экземпляр класса делегат

task.Start(); - Выполнение задачи.

CurrentId – Статический метод

(вернет Task.CurrentId == null если запустить в контексте основоного потока)

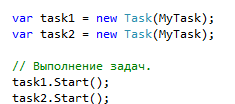
Возвращает уникальный идентификатор выполняющейся в настоящее время задачи Task

Console.WriteLine("MyTask() # " + Task.CurrentId + " завершен.");

Либо на экземпляре Task вызвать метод .Id

Console.WriteLine("Идентификатор задачи task1: " + task1.Id);

При одновременном запуске выполнение задачи происходит в разных потоках task1.Id



Приостановить действие потока до завершения задачи

task1.Wait(); - Метод Main() остается активным до завершения задачи MyTask().

Статические методы .WaitAll и .WaitAny возвращают управление потоков

Task.WaitAll(task1, task2); - дождаться завершения указанных задач

Task.WaitAny(task1, task2); - дождаться завершение какой либо задачи (вернет какая из задач (по номеру передачи аргумента)) была завершена )

Создание экземпляра задачи с использованием свойства Factory, типа TaskFactory.

Task task = Task.Factory.StartNew(MyTask); - ГОРЯЧАЯ ЗАДАЧА, при создании запускается, В случае запуска задачи через TaskFactory, вызов метода task.Start() не требуется.

Использование лямбда-оператора для определения задачи.

Task task = Task.Factory.StartNew(new Action(() =>

{

Console.WriteLine("Задача запущена.");

for (int count = 0; count < 10; count++)

{

Thread.Sleep(500);

Console.WriteLine("В лямбда-операторе, счетчик равен: " + count);

}

Console.WriteLine("Задача завершена.");

}));

Метод обратного вызова (CallBack) который будет запущен по завершению задачи

ПРОДОЛЖЕНИЕ ЗАДАЧИ

.ContinueWith

// Создание экземпляра задачи.

var action = new **Action**(MyTask);

var task = new Task(action);

// Создание продолжения задачи.

var continuationAction = new **Action**<Task>(ContinuationTask);

Task taskContinuation = task.ContinueWith(ContinuationTask);

// Выполнение последовательности задач.

task.Start();

// Ожидание завершения продолжения.

taskContinuation.Wait();

Метод исполняемый как продолжение задачи. Должен соответствовать классу делегата



Пример - static void ContinuationTask(Task task)

.Result

подобно вызову EndInvoke() на классе делегате (останавливает основной поток)

Метод который будет возвращать результат

Создание экземпляра первой задачи.

Сигнатура метода - static bool MyTask()

Task<bool> task1 = Task<bool>.Factory.StartNew(MyTask);

Console.WriteLine("Результат выполнения задачи MyTask: " + task1.Result);

Создание экземпляра второй задачи.

Сигнатура метода - static int SumIt(object v)

Task<int> task2 = Task<int>.Factory.StartNew(SumIt, 3);

Console.WriteLine("Результат выполнения задачи SumIt: " + task2.Result);

CancellationTokenSource

При создании экземпляра можно задать значение во времени выполнения запроса (если не вложиться в указанное время то будет брошено исключение)

Отмена задачи

var cancelTokSrc = new CancellationTokenSource(); - Создать объект источника признаков отмены.

Task task = Task.Factory.StartNew(MyTask, cancelTokSrc.Token, cancelTokSrc.Token); - Запустить задачу, передав признак отмены (структура **CancellationToken**)ей самой (сообщаем TPL что наша задача выполняется с возможностью отмены) и делегату.

Для того чтобы проверить в потоке в котором выполняется задача необходимо на экземпляре структуры вызвать метод .ThrowIfCancellationRequested(); (бросить если был запрос на выход)

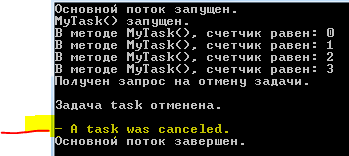
Пример - cancelTok.ThrowIfCancellationRequested(); - Проверить, отменена ли задача, прежде чем ее запускать. (если была отменена то будет брошено исключение  [OperationCanceledException](http://msdn.microsoft.com/en-us/library/system.operationcanceledexception(v=vs.110).aspx))

[OperationCanceledException](http://msdn.microsoft.com/en-us/library/system.operationcanceledexception(v=vs.110).aspx) Будет брошено в том случае если на экземпляре ИСТОЧНИКА ПРИЗНАКА ОТМЕНЫ был вызван метод .Cancel();

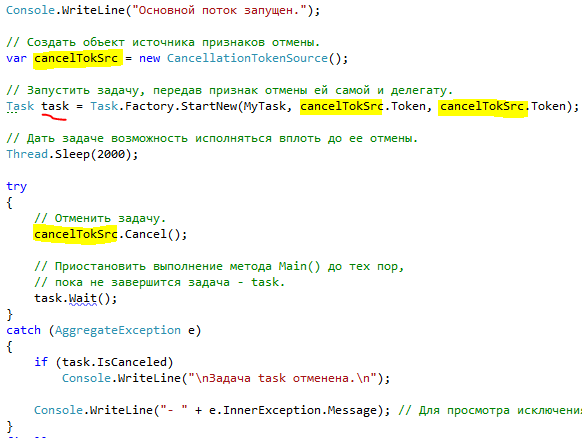
Пример - cancelTokSrc.Cancel(); - Отменить задачу.

.IsCancellationRequested Свойство структуры **CancellationToken** возвращает true если кто либо запросил завершение задачи

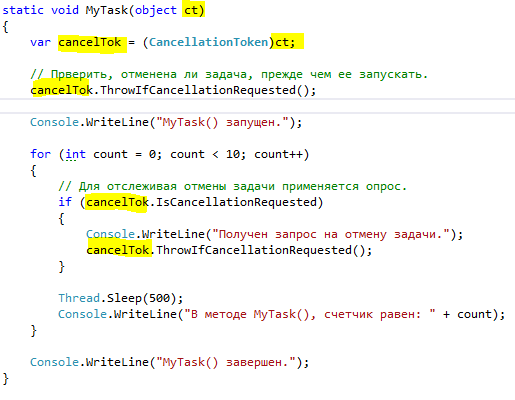
Пример - if (cancelTok.IsCancellationRequested) - Для отслеживая отмены задачи применяется опрос.



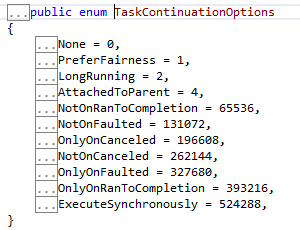
Основной поток



Метод выполняющийся в отдельном потоке

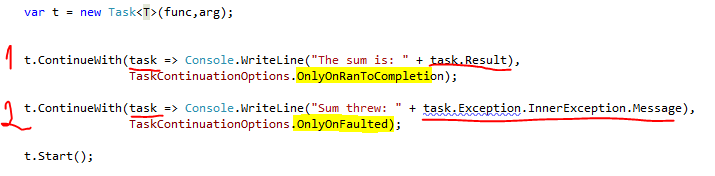


Метод продолжения на экземпляре задачи .ContinueWith с определенными условиями TaskContinuationOptions



1й выполниться, если задача будет окончена успешно то продолжение будет 1

2й выполниться, если будет сгенерирована ошибка



Token.Register и CancellationTokenSource.CreateLinkedTokenSource

// Создание объекта CancellationTokenSource

var cts1 = new CancellationTokenSource();

cts1.Token.Register(() => Console.WriteLine("cts1 canceled"));

// Создание второго объекта CancellationTokenSource

var cts2 = new CancellationTokenSource();

cts2.Token.Register(() => Console.WriteLine("cts2 canceled"));

// Создание нового объекта CancellationTokenSource,

// отменяемого при отмене cts1 или ct2

var linkedCts = CancellationTokenSource.CreateLinkedTokenSource(

cts1.Token, cts2.Token);

linkedCts.Token.Register(() => Console.WriteLine("linkedCts canceled"));

// Отмена одного из объектов CancellationTokenSource (я выбрал cts2)

cts2.Cancel();

// Показываем, какой из объектов CancellationTokenSource был отменен

Console.WriteLine("cts1 canceled={0}, cts2 canceled={1}, linkedCts={2}",

cts1.IsCancellationRequested, cts2.IsCancellationRequested,

linkedCts.IsCancellationRequested);



TaskCreationOptions.AttachedToParent если не применять то элементы массива будут равны 0

Task<**Int32**[]> parent = new Task<**Int32**[]>(() =>

{

var results = new **Int32**[3]; // Создание массива для результатов

// Создание и запуск 3 дочерних заданий

new Task(() => results[0] = Sum(10000),

TaskCreationOptions.AttachedToParent).Start();

new Task(() => results[1] = Sum(20000),

TaskCreationOptions.AttachedToParent).Start();

new Task(() => results[2] = Sum(30000),

TaskCreationOptions.AttachedToParent).Start();

// Возвращается ссылка на массив

// (элементы могут быть не инициализированы)

return results;

});

// Вывод результатов после завершения родительского и дочерних заданий

var cwt = parent.ContinueWith(

parentTask => Array.ForEach(parentTask.Result, Console.WriteLine));

// Запуск родительского задания, которое запускает дочерние

parent.Start();

Родительское задание создает и запускает три объекта Task. По умолчанию за-

дания-потомки попадают на самый верхний уровень и не имеют отношения к сво-

ему предку. Однако при установке флага TaskCreationOptions.AttachedToParent

родительское задание завершается только после завершения всех его потом-

ков.

Создание фабрики TaskFactory

p.Exception.Flatten().InnerExceptions все необработанные исключения

Task parent = new Task(() =>

{

var cts = new CancellationTokenSource();

var tf = new TaskFactory<**Int32**>(cts.Token,

TaskCreationOptions.AttachedToParent,

TaskContinuationOptions.ExecuteSynchronously,

TaskScheduler.Default);

// Задание создает и запускает 3 дочерних задания

var childTasks = new[] {

tf.StartNew(() => Sum(cts.Token, 10000)),

tf.StartNew(() => Sum(cts.Token, 2000)),

tf.StartNew(() => Sum(cts.Token, **Int32**.MaxValue)) // Исключение

// OverflowException

};

// Если дочернее задание становится источником исключения,

// отменяем все дочерние задания

for (**Int32** task = 0; task < childTasks.Length; task++)

childTasks[task].ContinueWith(

t => cts.Cancel(), TaskContinuationOptions.OnlyOnFaulted);

// После завершения дочерних заданий получаем максимальное

// возвращенное значение и передаем его другому заданию

// для вывода

//первый параметр массив задач, и делегат для каждого элемента, и

//среди них выбираем только ОДИН ЭЛЕМЕНТ КОТОРЫЙ ИМЕЕТ МАКСИМАЛЬНОЕ ЗНАЧЕНИЕ

tf.ContinueWhenAll(

childTasks, completedTasks => completedTasks.Where(

t => !t.IsFaulted && !t.IsCanceled).Max(t => t.Result),

**CancellationToken**.None)

.ContinueWith(t => Console.WriteLine("The maximum is: " + t.Result),

TaskContinuationOptions.ExecuteSynchronously);

});

// После завершения дочерних заданий выводим,

// в том числе, и необработанные исключения

parent.ContinueWith(p =>

{

// Текст помещен в StringBuilder и однократно вызван

// метод Console.WriteLine просто потому, что это задание

// может выполняться параллельно с предыдущим,

// и я не хочу путаницы в выводимом результате

StringBuilder sb = new StringBuilder(

"The following exception(s) occurred:" + Environment.NewLine);

foreach (var e in p.Exception.Flatten().InnerExceptions)

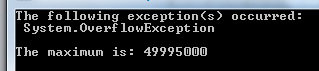
sb.AppendLine(" " + e.GetType().ToString());

Console.WriteLine(sb.ToString());

}, TaskContinuationOptions.OnlyOnFaulted);

// Запуск родительского задания, которое может запускать дочерние

parent.Start();



Вызов статических методов ContinueWhenAll и ContinueWhenAny классов

TaskFactory или TaskFactory<TResult> делает недействительными следующие фла-

ги TaskContinuationOption: NotOnRanToCompletion, NotOnFaulted и NotOnCanceled. Игнорируются и такие вспомогательные флаги, как OnlyOnCanceled, OnlyOnFaulted

и OnlyOnRanToCompletion. То есть методы ContinueWhenAll и ContinueWhenAny за-

пускают следующее задание вне зависимости от того, каким оказывается результат

выполнения предыдущего.

планировщика заданий

TaskScheduler.FromCurrentSynchronizationContext

Передаем объект который синхронизирует поток с основным потоком (по типу Dispatcher,CurrentContext)

private readonly TaskScheduler \_taskScheduler =TaskScheduler.FromCurrentSynchronizationContext();

private void TestCommandExecute(object obj)

{

try

{

TextBox t = obj as TextBox;

var task=Task.Run(() => "Text");

task.ContinueWith((r)=>GetValue(r,t), \_taskScheduler);

}

catch (Exception e)

{

throw e;

}

}

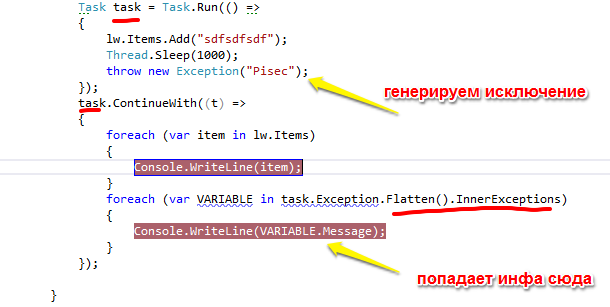
private void GetValue(Task<string> r,TextBox t)

{

t.Text = r.Result;

}

Обработка исключений Task



Parallel

ParallelOptions

Environment.ProcessorCount - Количество логических ядер центрального процессора

var options = new ParallelOptions

{

MaxDegreeOfParallelism = Environment.ProcessorCount>2

? Environment.ProcessorCount-1:1

}; - MaxDegreeOfParallelism настраиваем количество потоков которые будут использоваться

Метод класса Parallel .Invoke

Внимание!

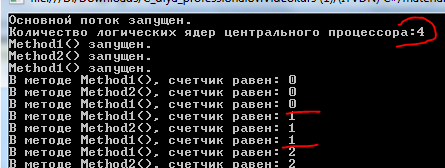
Выполнение метода Main() приостанавливается,

пока не произойдет возврат из метода Invoke().

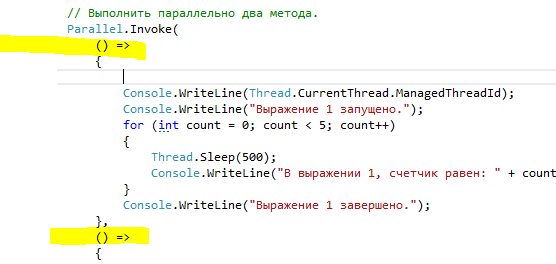
Parallel.Invoke( Method1, Method2,Method1,Method2); - запустятся все 4 метода

Или

Parallel.Invoke(options, Method1, Method2,Method1,Method2); - зависит от ядер процессора



Можно передать несколько лямда операторов



Parallel.For()

Применение метода Parallel.For() для организации параллельно выполняемого цикла обработки данных.

Внимание!

Выполнение метода Main() приостанавливается,

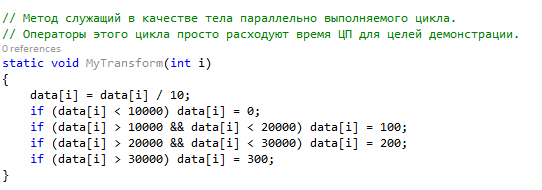
пока не произойдет возврат из метода For().

// Распараллелить цикл методом For().

Parallel.For(0, data.Length, new Action<int>(MyTransform)); - 1й аргумент начальное значение итератора

2й аргумент условие (меньше какого либо числа)

Метод выполняющийся параллельно



Parallel.For работает в данном случае на 2 секунды быстрей обычного метода

// Параллельный вариант инициализации массива в цикле.

Parallel.For(0, data.Length, i => data[i] = rand.Next());

// Последовательный вариант инициализации массива в цикле.

for (int i = 0; i < data.Length; i++) data[i] = rand.Next();

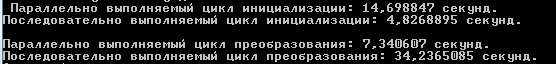
// Параллельный вариант преобразования данных в цикле.

Parallel.For(0, data.Length, MyTransform);

// Последовательный вариант преобразования данных в цикле.

for (int i = 0; i < data.Length; i++) MyTransform(i);

Результаты



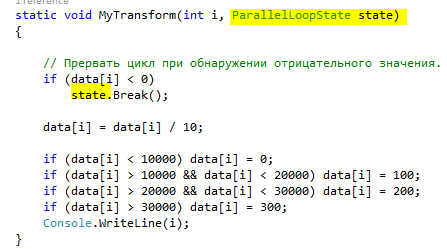
Отмена параллельного цикла

ParallelLoopState - Позволяет итерациям циклов Parallel взаимодействовать с другими итерациями.

Экземпляр этого класса предоставляется каждому циклу параллельным классом;

невозможно создавать экземпляры в пользовательском коде.

Метод который будет выполнен паралельно



(ParallelLoopState)

state.Break(); - отменить дальнейшее выполнение цикла

**ParallelLoopResult**

ParallelLoopResult - Предоставляет состояние выполнения цикла Parallel.

**ParallelLoopResult** loopResult = Parallel.For(0, data.Length, MyTransform);

// Проверить, завершился ли цикл.

if (!loopResult.IsCompleted) – завершился ли цикл

Console.WriteLine("\nЦикл завершился преждевременно." +

" На шаге {0} обнаружено отрицательное значение.\n", loopResult.LowestBreakIteration); - на каком шаге был остановлен цикл

Parallel.ForEach

Разница заключается в том что на каждой итерации цикла в метод заходит не значение итератора, а элемент массива



Результат работы цикла можно определить при помощи свойств. Если свойство

IsCompleted возвращает значение true, значит, цикл пройден полностью, и все

элементы обработаны. Если свойство IsCompleted возвращает значение false,

а свойство LowestBreakIteration — значение null, значит, каким-то из потоков был

вызван метод Stop. Если же в последнем случае значение, возвращаемое свойством

LowestBreakIteration, отлично от null, значит, каким-то из потоков был вызван

метод Break. При этом возвращенное свойством LowestBreakIteration значение

типа Int64 указывает индекс последнего гарантированно обработанного элемента.

PLINQ

.AsParallel

Возвращает не IEnumerable, а ParallelQuery<int>

Запрос PLINQ для поиска отрицательных значений.

var negatives = from val in data.AsParallel()

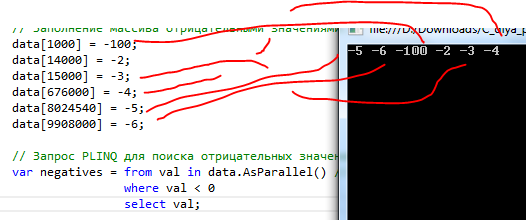
where val < 0

select val;

или

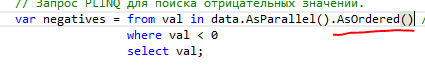
ParallelEnumerable.AsParallel<int>(data)

НО ВОЗВРАЩАЕТ ЭЛЕМЕНТЫ НЕ В ПОРЯДКЕ В КОТОРОМ ОНИ НАХОДИЛИСЬ В МАССИВЕ



**ЕСЛИ ПОРЯДОК ВАЖЕН ТО ПРИМЕНЯЕТСЯ**

.AsOrdered()



**Отмена параллелизма**

// Запрос PLINQ для поиска отрицательных значений.

ParallelQuery<int> negatives = from val in data

.AsParallel() – распараллелить

.WithCancellation(token.Token) – передаем структуру ранее созданного объекта (var token = new CancellationTokenSource();)

.WithExecutionMode(ParallelExecutionMode.ForceParallelism) - ParallelExecutionMode. В любом случае распараллелить запрос (так как параллелить или нет компилятор решает самостоятельно)

where val < 0

select val;

token.CancelAfter(50); - Создание задачи для отмены запроса по истечении 100 миллисекунд.

Если запрос не будет заверен в течении указанного времени то будет брошено исключение

Покажет только результат успевших перечислений

