В 4.0 NET Framework были добавлены механизмы распараллеливания задач TPL и PLINQ. Они все такие новые и бла бла бла.

TPL упрощает создание и применение многих потоков. Так же она позволяет автоматически использовать несколько процессоров.

PLINQ дает возможность составлять запросы, для обработки которых автоматически используется несколько процессоров, а также принцип параллелизма, когда это уместно.

Главное причиной появления таких важных новшеств, как TPL и PLINQ , служит возросшее значение параллелизма в современном программировании.

Библиотека TPL определена в пространстве имен System.Threading.Tasks.

Но для работы с ней обычно требуется также включать в программу класс System.

Threading, поскольку он поддерживает синхронизацию и другие средства многопоточной

обработки, в том числе и те, что входят в класс Interlocked.

В этой главе рассматривается и TPL, и PLINQ. Следует, однако, иметь в виду, что

и та и другая тема довольно обширны. Поэтому в этой главе даются самые основы и

рассматриваются некоторые простейшие способы применения TPL и PLINQ.

**Два подхода к параллельному программированию**

Применяя TPL, параллелизм в программу можно ввести двумя основными способами.

Первый из них называется *параллелизмом данных.* При таком подходе одна операция

над совокупностью данных разбивается на два параллельно выполняемых потока

или больше, в каждом из которых обрабатывается часть данных. Так, если изменяется

каждый элемент массива, то, применяя параллелизм данных, можно организовать параллельную

обработку разных областей массива в двух или больше потоках. Нетрудно

догадаться, что такие параллельно выполняющиеся действия могут привести к значительному

ускорению обработки данных по сравнению с последовательным подходом.

Несмотря на то что параллелизм данных был всегда возможен и с помощью класса

Thread, построение масштабируемых решений средствами этого класса требовало немало

усилий и времени. Это положение изменилось с появлением библиотеки TPL, с

помощью которой масштабируемый параллелизм данных без особого труда вводится

в программу.

Второй способ ввода параллелизм называется *параллелизмом задач.* При таком подходе

две операции или больше выполняются параллельно. Следовательно, параллелизм

задач представляет собой разновидность параллелизма, который достигался в

прошлом средствами класса Thread. А к преимуществам, которые сулит применение

TPL, относится простота применения и возможность автоматически масштабировать

исполнение кода на несколько процессоров.

**Класс Task**

В основу TPL положен класс Task. Элементарная единица исполнения инкапсулируется

в TPL средствами класса Task, а не Thread. Класс Task отличается от класса

Thread тем, что он является абстракцией, представляющей асинхронную операцию.

А в классе Thread инкапсулируется поток исполнения. Разумеется, на системном уровне

поток по-прежнему остается элементарной единицей исполнения, которую можно

планировать средствами операционной системы. Но соответствие экземпляра объекта

класса Task и потока исполнения не обязательно оказывается взаимно-однозначным.

Кроме того, исполнением задач управляет планировщик задач, который работает с пудом

потоков. Это, например, означает, что несколько задач могут разделять один и тот

же поток. Класс Task (и вся остальная библиотека TPL) определены в пространстве

имен System.Threading.Tasks.

**Создание задачи**

Создать новую задачу в виде объекта класса Task и начать ее исполнение можно

самыми разными способами. Для начала создадим объект типа Task с помощью конструктора

и запустим его, вызвав метод Start(). Для этой цели в классе Task определено

несколько конструкторов.

public Task(Action действие)

где *действие* обозначает точку входа в код, представляющий задачу, тогда как

Action — делегат, определенный в пространстве имен System. Форма делегата

Action, которой мы собираемся воспользоваться, выглядит следующим образом.

public delegate void Action()

Таким образом, точкой входа должен служить метод, не принимающий никаких

параметров и не возвращающий никаких значений. (Как будет показано далее, делегату

Action можно также передать аргумент.)

Как только задача будет создана, ее можно запустить на исполнение, вызвав метод

Start(). Ниже приведена одна из его форм.

public void Start()

После вызова метода Start() планировщик задач запланирует исполнение задачи.

В приведенной ниже программе все изложенное выше демонстрируется на практике.

В этой программе отдельная задача создается на основе метода MyTask(). После

того как начнет выполняться метод Main(), задача фактически создается и запускается

на исполнение. Оба метода MyTask() и Main() выполняются параллельно.

(***glava24\_1***)

using System.Threading;

using System.Threading.Tasks;

class DemoTask

{

//method as task

static void MyTask()

{

Console.WriteLine("MyTask() started");

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

{

Thread.Sleep(500);

Console.WriteLine("In method MyTask(), count = " + count);

}

Console.WriteLine("My task done.");

}

static void Main()

{

Console.WriteLine("Main thread started.");

//build task objcet

Task tsk = new Task(MyTask);

//start task

tsk.Start();

//method Main() is active until MyTask() to be done

for(int i = 0; i < 60; i++)

{

Console.Write('.');

Thread.Sleep(100);

}

Console.WriteLine("Main thread done.");

}

}

Следует иметь в виду, что по умолчанию задача исполняется в фоновом потоке.

Следовательно, при завершении создающего потока завершается и сама задача. Именно

поэтому в рассматриваемой здесь программе метод Thread.Sleep() использован

для сохранения активным основного потока до тех пор, пока не завершится выполнение

метода MyTask(). Как и следовало ожидать, организовать ожидание завершения

задачи можно и более совершенными способами, что и будет показано далее.

В приведенном выше примере программы задача, предназначавшаяся для параллельного

исполнения, обозначалась в виде статического метода. Но такое требование к

задаче не является обязательным. Например, в приведенной ниже программе, которая

является переработанным вариантом предыдущей, метод MyTask(), выполняющий

роль задачи, инкапсулирован внутри класса.

(***glava24\_2***)

using System.Threading;

using System.Threading.Tasks;

class MyClass

{

//method as task

public void MyTask()

{

Console.WriteLine("MyTask() started.");

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

{

Thread.Sleep(500);

Console.WriteLine("In method MyTask(), count = " + count);

}

Console.WriteLine("MyTask done.");

}

}

class DemoTask

{

static void Main()

{

Console.WriteLine("Main thread started.");

//build object MyClass

MyClass mc = new MyClass();

//buld object of task

Task tsk = new Task(mc.MyTask);

//run task

tsk.Start();

//save method Main() active until MyTask to be done

for (int i = 0; i < 60; i++)

{

Console.Write('.');

Thread.Sleep(100);

}

Console.WriteLine("Main thread done.");

}

}

Результат выполнения этой программы получается таким же, как и прежде. Единственное

отличие состоит в том, что метод MyTask() вызывается теперь для экземпляра

объекта класса MyClass.

В отношении задач необходимо также иметь в виду следующее: после того, как задача

завершена, она не может быть перезапущена. Следовательно, иного способа повторного

запуска задачи на исполнение, кроме создания ее снова, не существует.

**Применение идентификатора задачи**

В отличие от класса Thread; в классе Task отсутствует свойство Name для хранения

имени задачи. Но вместо этого в нем имеется свойство Id для хранения идентификатора

задачи, по которому можно распознавать задачи. Свойство Id доступно только для

чтения и относится к типу int.

public int Id { get; }

Каждая задача получает идентификатор, когда она создается. Значения идентификаторов

уникальны, но не упорядочены. Поэтому один идентификатор задачи может

появиться перед другим, хотя он может и не иметь меньшее значение.

Идентификатор исполняемой в настоящий момент задачи можно выявить с помощью

свойства CurrentId. Это свойство доступно только для чтения, относится к типу

static и объявляется следующим образом.

public static Nullable<int> CurrentID { get; }

Оно возвращает исполняемую в настоящий момент задачу или же пустое значение,

если вызывающий код не является задачей.

В приведенном ниже примере программы создаются две задачи и показывается,

какая из них исполняется.

(***glava24\_3***)

class DemoTask

{

//method as task

static void MyTask()

{

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

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

{

Thread.Sleep(500);

Console.WriteLine("In method MyTask() #"

+ Task.CurrentId + ", count = " + count);

}

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

}

static void Main()

{

Console.WriteLine("Main thread started.");

//build 2 task objects

Task tsk = new Task(MyTask);

Task tsk2 = new Task(MyTask);

//run tasks

tsk.Start();

tsk2.Start();

Console.WriteLine("tsk ID: " + tsk.Id);

Console.WriteLine("tsk2 ID: " + tsk2.Id);

//save method Main() active

for(int i = 0; i<60; i++)

{

Console.Write('.');

Thread.Sleep(100);

}

Console.WriteLine("Main done.");

}

}

**Применение методов ожидания**

В приведенных выше примерах основной поток исполнения, а по существу, метод

Main(), завершался потому, что такой результат гарантировали вызовы метода

Thread.Sleep(). Но подобный подход нельзя считать удовлетворительным.

Организовать ожидание завершения задач можно и более совершенным способом,

применяя методы ожидания, специально предоставляемые в классе Task. Самым простым

из них считается метод Wait(), приостанавливающий исполнение вызывающего

потока до тех пор, пока не завершится вызываемая задача. Ниже приведена простейшая

форма объявления этого метода.

public void Wait()

При выполнении этого метода могут быть сгенерированы два исключения. Первым

из них является исключение ObjectDisposedException. Оно генерируется в том

случае, если задача освобождена посредством вызова метода Dispose(). А второе исключение,

AggregateException, генерируется в том случае, если задача сама генерирует

исключение или же отменяется. Как правило, отслеживается и обрабатывается

именно это исключение. В связи с тем что задача может сгенерировать не одно исключение,

если, например, у нее имеются порожденные задачи, все подобные исключения

собираются в единое исключение типа AggregateException. Для того чтобы

выяснить, что же произошло на самом деле, достаточно проанализировать внутренние

исключения, связанные с этим совокупным исключением. А до тех пор в приведенных

далее примерах любые исключения, генерируемые задачами, будут обрабатываться во

время выполнения.

Ниже приведен вариант предыдущей программы, измененный с целью продемонстрировать

применение метода Wait() на практике. Этот метод используется внутри

метода Main(), чтобы приостановить его выполнение до тех пор, пока не завершатся

обе задачи tsk и tsk2.

(***glava24\_3.1***)

static void Main()

{

Console.WriteLine("Main thread started.");

//build 2 task objects

Task tsk = new Task(MyTask);

Task tsk2 = new Task(MyTask);

//run tasks

tsk.Start();

tsk2.Start();

Console.WriteLine("tsk ID: " + tsk.Id);

Console.WriteLine("tsk2 ID: " + tsk2.Id);

tsk.Wait();

tsk2.Wait();

Console.WriteLine("Main done.");

}

}

Как следует из приведенного выше результата, выполнение метода Main() приостанавливается

до тех пор, пока не завершатся обе задачи tsk и tsk2. Следует, однако, иметь

в виду, что в рассматриваемой здесь программе последовательность завершения задач

tsk и tsk2 не имеет особого значения для вызовов метода Wait(). Так, если первой завершается

задача tsk2, то в вызове метода tsk.Wait() будет по-прежнему ожидаться

завершение задачи tsk. В таком случае вызов метода tsk2.Wait() приведет к выполнению

и немедленному возврату из него, поскольку задача tsk2 уже завершена.

В данном случае оказывается достаточно двух вызовов метода Wait(), но того же

результата можно добиться и более простым способом, воспользовавшись методом

WaitAll(). Этот метод организует ожидание завершения группы задач. Возврата из

него не произойдет до тех пор, пока не завершатся все задачи.

public static void WaitAll(params Task[] tasks)

Задачи, завершения которых требуется ожидать, передаются с помощью параметра

в виде массива *tasks.* А поскольку этот параметр относится к типу params,

то данному методу можно отдельно передать массив объектов типа Task или список

задач. При этом могут быть сгенерированы различные исключения, включая и

AggregateException.

Task.WaitAll(tsk, tsk2);

Организуя ожидание завершения нескольких задач, следует быть особенно внимательным,

чтобы избежать взаимоблокировок. Так, если две задачи ожидают завершения

друг друга, то вызов метода WaitAll() вообще не приведет к возврату из него.

Разумеется, условия для взаимоблокировок возникают в результате ошибок программирования,

которых следует избегать. Следовательно, если вызов метода WaitAll()

не приводит к возврату из него, то следует внимательно проанализировать, могут ли

две задачи или больше взаимно блокироваться. (Вызов метода Wait(), который не

приводит к возврату из него, также может стать причиной взаимоблокировок.)

Иногда требуется организовать ожидание до тех пор, пока не завершится любая из

группы задач. Для этой цели служит метод WaitAny().

public static int WaitAny(params Task[] tasks)

Задачи, завершения которых требуется ожидать, передаются с помощью параметра

в виде массива *tasks* объектов типа Task или отдельного списка аргументов типа

Task. Этот метод возвращает индекс задачи, которая завершается первой. При этом

могут быть сгенерированы различные исключения.

Task.WaitAny(tsk, tsk2);

Теперь, выполнение метода Main() возобновится, а программа завершится, как

только завершится одна из двух задач.

Помимо рассматривавшихся здесь форм методов Wait(), WaitAll() и

WaitAny(), имеются и другие их варианты, в которых можно указывать период простоя

или отслеживать признак отмены.

**Вызов метода Dispose()**

В классе Task реализуется интерфейс IDisposable, в котором определяется метод

Dispose(). Ниже приведена форма его объявления.

public void Dispose()

Метод Dispose() реализуется в классе Task, освобождая ресурсы, используемые

этим классом. Как правило, ресурсы, связанные с классом Task, освобождаются автоматически

во время "сборки мусора" (или по завершении программы). Но если эти

ресурсы требуется освободить еще раньше, то для этой цели служит метод Dispose().

Это особенно важно в тех программах, где создается большое число задач, оставляемых

на произвол судьбы.

Следует, однако, иметь в виду, что метод Dispose() можно вызывать для отдельной

задачи только после ее завершения. Следовательно, для выяснения факта завершения

отдельной задачи, прежде чем вызывать метод Dispose(), потребуется некоторый

механизм, например, вызов метода Wait(). Именно поэтому так важно было

рассмотреть метод Wait(), перед тем как обсуждать метод Dispose(). Ели же попытаться

вызвать Dispose() для все еще активной задачи, то будет сгенерировано исключение

InvalidOperationException.

Во всех примерах, приведенных в этой главе, создаются довольно короткие задачи,

которые фазу же завершаются, и поэтому применение метода Dispose() в этих примерах

не дает никаких преимуществ. (Именно по этой причине вызывать метод Dispose()

в приведенных выше программах не было никакой необходимости. Ведь все они завершались,

как только завершалась задача, что в конечном итоге приводило к освобождению от

остальных задач.) Но в целях демонстрации возможностей данного метода и во избежание

каких-либо недоразумений метод Dispose() будет вызываться явным образом при

непосредственном обращении с экземплярами объектов типа Task во всех последующих

примерах программ. Если вы обнаружите отсутствие вызовов метода Dispose() в исходном

коде, полученном из других источников, то не удивляйтесь этому. Опять же, если

программа завершается, как только завершится задача, то вызывать метод Dispose() нет

никакого смысла — разве что в целях демонстрации его применения.

**Применение класса TaskFactory для запуска задачи**

Приведенные выше примеры программы были составлены не так эффективно, как

следовало бы, поскольку задачу можно создать и сразу же начать ее исполнение, вызвав

метод StartNew(), определенный в классе TaskFactory. В классе TaskFactory

предоставляются различные методы, упрощающие создание задач и управление ими.

По умолчанию объект класса TaskFactory может быть получен из свойства Factory,

доступного только для чтения в классе Task. Используя это свойство, можно вызвать

любые методы класса TaskFactory. Метод StartNew() существует во множестве

форм.

public Task StartNew(Action action)

где *action* — точка входа в исполняемую задачу. Сначала в методе StartNew() автоматически

создается экземпляр объекта типа Task для действия, определяемого параметром

*action,* а затем планируется запуск задачи на исполнение. Следовательно,

необходимость в вызове метода Start() теперь отпадает.

Например, следующий вызов метода StartNew() в рассматривавшихся ранее программах

приведет к созданию и запуску задачи tsk одним действием.

Task tsk = Task.Factory.StartNew(MyTask);

После этого оператора сразу же начнет выполняться метод MyTask().

Метод StartNew() оказывается более эффективным в тех случаях, когда задача создается

и сразу же запускается на исполнение. Поэтому именно такой подход и применяется

в последующих примерах программ.

**Применение лямбда-выражения в качестве задачи**

Кроме использования обычного метода в качестве задачи, существует и другой, более

рациональный подход: указать лямбда-выражение как отдельно решаемую задачу.

Напомним, что лямбда-выражения являются особой формой анонимных функций. Поэтому

они могут исполняться как отдельные задачи. Лямбда-выражения оказываются

особенно полезными в тех случаях, когда единственным назначением метода является

решение одноразовой задачи. Лямбда-выражения могут составлять отдельную задачу

иди же вызывать другие методы. Так или иначе, применение лямбда-выражения в качестве

задачи может стать привлекательной альтернативой именованному методу.

В приведенном ниже примере программы демонстрируется применение лямбда-

выражения в качестве задачи. В этой программе код метода MyTask() из предыдущих

примеров программ преобразуется в лямбда-выражение.

(***glava24\_4***)

class DemoLambdaTask

{

static void Main()

{

Console.WriteLine("Main thread started.");

//next lambda func is used for task

Task tsk = Task.Factory.StartNew(() =>

{

Console.WriteLine("Task started.");

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

{

Thread.Sleep(500);

Console.WriteLine("Count = " + count);

}

Console.WriteLine("Task done.");

});

//wait for tsk

tsk.Wait();

//cleare tsk

tsk.Dispose();

Console.WriteLine("Main thread ended.");

}

}

Помимо применения лямбда-выражения для описания задачи, обратите также

внимание в данной программе на то, что вызов метода tsk.Dispose() не делается до

тех пор, пока не произойдет возврат из метода tsk.Wait(). Как пояснялось в предыдущем

разделе, метод Dispose() можно вызывать только по завершении задачи.

Для того чтобы убедиться в этом, попробуйте поставить вызов метода tsk.Dispose()

в рассматриваемой здесь программе перед вызовом метода tsk.Wait(). Вы сразу же

заметите, что это приведет к исключительной ситуации.

**Создание продолжения задачи**

Одной из новаторских и очень удобных особенностей библиотеки TPL является возможность

создавать продолжение задачи. *Продолжение* — это одна задача, которая автоматически

начинается после завершения другой задачи. Создать продолжение можно,

в частности, с помощью метода ContinueWith(), определенного в классе Task.

Ниже приведена простейшая форма его объявления:

public Task ContinueWith(Action<Task> действие\_продолжения)

где *действие\_продолжения* обозначает задачу, которая будет запущена на исполнение

по завершении вызывающей задачи. У делегата Action имеется единственный параметр

типа Task. Следовательно, вариант делегата Action, применяемого в данном

методе, выглядит следующим образом.

public delegate void Action<in T>(T obj)

В данном случае обобщенный параметр T обозначает класс Task.

Продолжение задачи демонстрируется на примере следующей программы.

(***glava24\_5***)

class ContinuationDemo

{

//task method

static void MyTask()

{

Console.WriteLine("MyTask() started.");

for (int count = 0; count < 5; count++)

{

Thread.Sleep(500);

Console.WriteLine("In method MyTask(), count = " + count);

}

Console.WriteLine("MyTask done.");

}

//method as continuation

static void ContTask(Task t)

{

Console.WriteLine("Continuation started.");

for (int count = 0; count < 5; count++)

{

Thread.Sleep(500);

Console.WriteLine("In continuation count = " + count);

}

Console.WriteLine("Continuation done.");

}

static void Main()

{

Console.WriteLine("Main thread started.");

//build object first task

Task tsk = new Task(MyTask);

//now create continuation task

Task taskCont = tsk.ContinueWith(ContTask);

//start task

tsk.Start();

//wait for continuation to be done

taskCont.Wait();

tsk.Dispose();

taskCont.Dispose();

Console.WriteLine("Main thread ended.");

}

Как следует из приведенного выше результата, вторая задача не начинается до тех

пор, пока не завершится первая. Обратите также внимание на то, что в методе Main()

пришлось ожидать окончания только продолжения задачи. Дело в том, что метод

MyTask() как задача завершается еще до начала метода ContTask как продолжения

задачи. Следовательно, ожидать завершения метода MyTask() нет никакой надобности,

хотя если и организовать такое ожидание, то в этом будет ничего плохого.

Любопытно, что в качестве продолжения задачи нередко применяется лямбда-

выражение. Для примера ниже приведен еще один способ организации продолжения

задачи из предыдущего примера программы.

Любопытно, что в качестве продолжения задачи нередко применяется лямбда-

выражение. Для примера ниже приведен еще один способ организации продолжения

задачи из предыдущего примера программы.

Task taskCont = tsk.ContinueWith((first) =>

{

Console.WriteLine("Продолжение запущено");

for (int count = 0; count < 5; count++)

{

Thread.Sleep(500);

Console.WriteLine("В продолжении подсчет равен " + count);

}

Console.WriteLine("Продолжение завершено");

)

};

В этом фрагменте кода параметр first принимает предыдущую задачу (в данном

случае — tsk).

Помимо метода ContinueWith(), в классе Task предоставляются и другие методы,

поддерживающие продолжение задачи, обеспечиваемое классом TaskFactory. К их числу

относятся различные формы методов ContinueWhenAny() и ContinueWhenAll(),

которые продолжают задачу, если завершится любая или все указанные задачи соответственно.

**Возврат значения из задачи**

Задача может возвращать значение. Это очень удобно по двум причинам.

Во-первых, это означает, что с помощью задачи можно вычислить некоторый результат.

Подобным образом поддерживаются параллельные вычисления. И во-вторых, вызывающий

процесс окажется блокированным до тех пор, пока не будет получен результат.

Это означает, что для организации ожидания результата не требуется никакой

особой синхронизации.

Для того чтобы возвратить результат из задачи, достаточно создать эту задачу, используя

обобщенную форму Task<TResult> класса Task. Ниже приведены два конструктора

этой формы класса Task:

public Task(Func<TResult> функция)

public Task(Func<Object, TResult> функция, Object состояние)

где *функция* обозначает выполняемый делегат. Обратите внимание на то, что он должен

быть типа Func, а не Action. Тип Func используется именно в тех случаях, когда

задача возвращает результат. В первом конструкторе создается задача без аргументов,

а во втором конструкторе — задача, принимающая аргумент типа Object, передаваемый

как *состояние.* Имеются также другие конструкторы данного класса.

Как и следовало ожидать, имеются также другие варианты метода StartNew(),

доступные в обобщенной форме класса TaskFactory<TResult> и поддерживающие

возврат результата из задачи. Ниже приведены те варианты данного метода, которые

применяются параллельно с только что рассмотренными конструкторами класса

Task.

public Task<TResult> StartNew(Func<TResult> функция)

public Task<TResult> StartNew(Func<Object, TResult> функция, Object состояние)

В любом случае значение, возвращаемое задачей, подучается из свойства Result в

классе Task, которое определяется следующим образом.

public TResult Result { get; internal set; }

Аксессор set является внутренним для данного свойства, и поэтому оно оказывается

доступным во внешнем коде, по существу, только для чтения. Следовательно, задача

получения результата блокирует вызывающий код до тех пор, пока результат не будет

вычислен.

В приведенном ниже примере программы демонстрируется возврат задачей значений.

В этой программе создаются два метода. Первый из них, MyTask(), не принимает

параметров, а просто возвращает логическое значение true типа bool. Второй метод,

SumIt(), принимает единственный параметр, который приводится к типу int, и возвращает

сумму из значения, передаваемого в качестве этого параметра.

(возвращаемый тип передается в тип аргументов, и результат сохраняется в Result, который встроен в сам класс Task)

(***glava24\_6***)

class DemoTask

{

//simple method, returns result have no arguments

static bool MyTask()

{

return true;

}

//method returns summ positive numbers

//passed as parameter

static int SumIt(object v)

{

int x = (int)v;

int sum = 0;

for (; x > 0; x--)

sum += x;

return sum;

}

static void Main()

{

Console.WriteLine("Main thread started.");

//build object first task

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

Console.WriteLine("Result after MyTask: " + tsk.Result);

//build object second task

Task<int> tsk2 = Task<int>.Factory.StartNew(SumIt, 3);

Console.WriteLine("Result after SumIt: " + tsk2.Result);

tsk.Dispose();

tsk2.Dispose();

Console.WriteLine("Main thread ended.");

}

}

Помимо упомянутых выше форм класса Task<TResult> и метода

StartNew<TResult>, имеются также другие формы. Они позволяют указывать другие

дополнительные параметры.

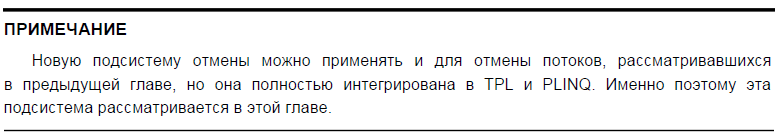
**Отмена задачи и обработка исключения AggregateException**

В версии 4.0 среды .NET Framework внедрена новая подсистема, обеспечивающая

структурированный, хотя и очень удобный способ отмены задачи. Эта новая подсистема

основывается на понятии *признака отмены.* Признаки отмены поддерживаются в

классе Task, среди прочего, с помощью фабричного метода StartNew().



Отмена задачи, как правило, выполняется следующим образом. Сначала получается

признак отмены из источника признаков отмены. Затем этот признак передается

задаче, после чего она должна контролировать его на предмет получения запроса

на отмену. (Этот запрос может поступить только из источника признаков

отмены.) Если получен запрос на отмену, задача должна завершиться. В одних случаях

этого оказывается достаточно для простого прекращения задачи без каких-

либо дополнительных действий, а в других — из задачи должен быть вызван метод

ThrowIfCancellationRequested() для признака отмены. Благодаря этому в отменяющем

коде становится известно, что задача отменена.

Признак отмены является экземпляром объекта типа CancellationToken,

т.е. структуры, определенной в пространстве имен System.Threading. В структуре

CancellationToken определено несколько свойств и методов, но мы воспользуемся

двумя из них. Во-первых, это доступное только для чтения свойство

IsCancellationRequested, которое объявляется следующим образом.

public bool IsCancellationRequested { get; }

Оно возвращает логическое значение true, если отмена задачи была запрошена для

вызывающего признака, а иначе — логическое значение false. И во-вторых, это метод

ThrowIfCancellationRequested(), который объявляется следующим образом.

public void ThrowIfCancellationRequested()

Если признак отмены, для которого вызывается этот метод, получил запрос на отмену,

то в данном методе генерируется исключение OperationCanceledException.

В противном случае никаких действий не выполняется. В отменяющем коде можно

организовать отслеживание упомянутого исключения с целью убедиться в том, что

отмена задачи действительно произошла. Как правило, с этой целью сначала перехватывается

исключение AggregateException, а затем его внутреннее исключение

анализируется с помощью свойства InnerException или InnerExceptions.

Признак отмены получается из источника признаков отмены, который представляет

собой объект класса CancellationTokenSource, определенного в пространстве

имен System. Threading. Для того чтобы получить данный признак, нужно

создать сначала экземпляр объекта типа CancellationTokenSource.

Признак отмены, связанный с данным источником, оказывается

доступным через используемое только для чтения свойство Token, которое

объявляется следующим образом.

public CancellationToken Token { get; }

Это и есть тот признак, который должен быть передан отменяемой задаче.

Для отмены в задаче должна быть получена копия признака отмены и организован

контроль этого признака с целью отслеживать саму отмену. Такое отслеживание

можно организовать тремя способами: опросом, методом обратного вызова и с помощью

дескриптора ожидания. Проще всего организовать опрос, и поэтому здесь будет

рассмотрен именно этот способ. С целью опроса в задаче проверяется упомянутое

выше свойство IsCancellationRequested признака отмены. Если это свойство содержит

логическое значение true, значит, отмена была запрошена, и задача должна быть завершена. Опрос может оказаться весьма эффективным, если организовать

его правильно. Так, если задача содержит вложенные циклы, то проверка свойства

IsCancellationRequested во внешнем цикле зачастую дает лучший результат, чем

его проверка на каждом шаге внутреннего цикла.

Для создания задачи, из которой вызывается метод ThrowIfCancellationRequested(),

когда она отменяется, обычно требуется передать признак отмены как самой задаче,

так и конструктору класса Task, будь то непосредственно или же косвенно через метод

StartNew(). Передача признака отмены самой задаче позволяет изменить состояние отменяемой

задачи в запросе на отмену из внешнего кода.

public Task StartNew(Action<Object> action, Object состояние,

CancellationToken признак\_отмены)

В этой форме признак отмены передается через параметры, обозначаемые как

*состояние* и *признак\_отмены.* Это означает, что признак отмены будет передан как

делегату, реализующему задачу, так и самому экземпляру объекта типа Task.

public delegate void Action<in T>(T obj)

В данном случае обобщенный параметр Т обозначает тип Object. В силу этого

объект *obj* должен быть приведен внутри задачи к типу CancellationToken.

И еще одно замечание: по завершении работы с источником признаков отмены

следует освободить его ресурсы, вызвав метод Dispose().

Факт отмены задачи может быть проверен самыми разными способами. Здесь применяется

следующий подход: проверка значения свойства IsCanceled для экземпляра

объекта типа Task. Если это логическое значение true, то задача была отменена.

В приведенной ниже программе демонстрируется отмена задачи. В ней применяется

опрос для контроля состояния признака отмены. Обратите внимание на

то, что метод ThrowIfCancellationRequested() вызывается после входа в метод

MyTask(). Это дает возможность завершить задачу, если она была отмена еще до ее

запуска. Внутри цикла проверяется свойство IsCancellationRequested. Если это

свойство содержит логическое значение true, а оно устанавливается после вызова метода

Cancel() для экземпляра источника признаков отмены, то на экран выводится

сообщение об отмене и далее вызывается метод ThrowIfCancellationRequested()

для отмены задачи.

(***glava24\_7***)

class DemoCancelTask

{

//Task method

static void MyTask(Object ct)

{

CancellationToken cancelTok = (CancellationToken)ct;

//check if task canceled before run it

cancelTok.ThrowIfCancellationRequested();

Console.WriteLine("MyTask started.");

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

{

//this example for detection of cancelation of task

if (cancelTok.IsCancellationRequested)

{

Console.WriteLine("Got request of cancelation task.");

cancelTok.ThrowIfCancellationRequested();

}

Thread.Sleep(500);

Console.WriteLine("In method MyTask() count = " + count);

}

Console.WriteLine("MyTask done.");

}

static void Main()

{

Console.WriteLine("Main thread started.");

//create object reason cancelation

CancellationTokenSource cancelTokSrc = new CancellationTokenSource();

//start task, pass reason of cancelation to itself, and delegate

Task tsk = Task.Factory.StartNew(MyTask, cancelTokSrc.Token,

cancelTokSrc.Token);

//give to task time to run until cancel

Thread.Sleep(2000);

try

{

//cancel task

cancelTokSrc.Cancel();

//pause Main() until tsk to be done

tsk.Wait();

}

catch (AggregateException exc)

{

if (tsk.IsCanceled)

Console.WriteLine("\nTask tsk canceled\n");

//to look exception, remove comments from next string of code

//Console.WriteLine(exc);

}

finally

{

tsk.Dispose();

cancelTokSrc.Dispose();

}

Console.WriteLine("Main done.");

}

}

Как следует из приведенного выше результата, выполнение метода MyTask()

отменяется в методе Main() лишь две секунды спустя. Следовательно, в методе

MyTask() выполняются четыре шага цикла. Когда же перехватывается исключение

AggregateException, проверяется состояние задачи. Если задача tsk отменена, что

и должно произойти в данном примере, то об этом выводится соответствующее сообщение.

Следует, однако, иметь в виду, что когда сообщение AggregateException

генерируется в ответ на отмену задачи, то это еще не свидетельствует об ошибке, а просто

означает, что задача была отменена.

Выше были изложены лишь самые основные принципы, положенные в основу отмены

задачи и генерирования исключения AggregateException. Тем не менее эта

тема намного обширнее и требует от вас самостоятельного и углубленного изучения,

если вы действительно хотите создавать высокопроизводительные, масштабируемые

приложения.

**Другие средства организации задач**

В предыдущих разделах был описан ряд понятий и основных способов организации

и исполнения задач. Но имеются и другие полезные средства. В частности, задачи

можно делать вложенными, когда одни задачи способны создавать другие, или же порожденными,

когда вложенные задачи оказываются тесно связанными с создающей

их задачей.

В предыдущем разделе было дано краткое описание исключения AggregateException,

но у него имеются также другие особенности, которые могут оказаться

весьма полезными. К их числу относится метод Flatten(), применяемый для преобразования

любых внутренних исключений типа AggregateException в единственное

исключение AggregateException. Другой метод, Handle(), служит для обработки

исключения, составляющего совокупное исключение AggregateException.

При создании задачи имеется возможность указать различные дополнительные

параметры, оказывающие влияние на особенности ее исполнения. Для этой цели указывается

экземпляр объекта типа TaskCreationOptions в конструкторе класса Task

или же в фабричном методе StartNew(). Кроме того, в классе TaskFactory доступно

целое семейство методов FromAsync(), поддерживающих модель асинхронного программирования

(АРМ — Asynchronous Programming Model).

Как упоминалось ранее в этой главе, задачи планируются на исполнение экземпляром

объекта класса TaskScheduler. Как правило, для этой цели предоставляется

планировщик, используемый по умолчанию в среде .NET Framework. Но этот планировщик

может быть настроен под конкретные потребности разработчика. Кроме того,

допускается применение специализированных планировщиков задач.

**Класс Parallel**

В примерах, приведенных до сих пор в этой главе, демонстрировались ситуации,

в которых библиотека TPL использовалась таким же образом, как и класс Thread. Но

это было лишь самое элементарное ее применение, поскольку в TPL имеются и другие

средства. К их числу относится класс Parallel, который упрощает параллельное исполнение

кода и предоставляет методы, рационализирующие оба вида параллелизма:

данных и задач.

Класс Parallel является статическим, и в нем определены методы For(),

ForEach() и Invoke(). У каждого из этих методов имеются различные формы. В частности,

метод For() выполняет распараллеливаемый цикл for, а метод ForEach() —

распараллеливаемый цикл foreach, и оба метода поддерживают параллелизм данных.

А метод Invoke() поддерживает параллельное выполнение двух методов иди

больше. Как станет ясно дальше, эти методы дают преимущество реализации на

практике распространенных методик параллельного программирования, не прибегая

к управлению задачами иди потоками явным образом.

**Распараллеливание задач методом Invoke()**

Метод Invoke(), определенный в классе Parallel, позволяет выполнять один

иди несколько методов, указываемых в виде его аргументов. Он также масштабирует

исполнение кода, используя доступные процессоры, если имеется такая возможность.

public static void Invoke(params Action[] actions)

Выполняемые методы должны быть совместимы с описанным ранее делегатом

Action. Напомним, что делегат Action объявляется следующим образом.

public delegate void Action()

Следовательно, каждый метод, передаваемый методу Invoke() в качестве аргумента,

не должен ни принимать параметров, ни возвращать значение. Благодаря тому что

параметр *actions* данного метода относится к типу params, выполняемые методы

могут быть указаны в виде переменного списка аргументов. Для этой цели можно также

воспользоваться массивом объектов типа Action, но зачастую оказывается проще

указать список аргументов.

Метод Invoke() сначала инициирует выполнение, а затем ожидает завершения

всех передаваемых ему методов. Это, в частности, избавляет от необходимости (да и

не позволяет) вызывать метод Wait(). Все функции параллельного выполнения метод

Wait() берет на себя. И хотя это не гарантирует, что методы будут действительно выполняться

параллельно, тем не менее, именно такое их выполнение предполагается,

если система поддерживает несколько процессоров. Кроме того, отсутствует возможность

указать порядок выполнения методов от первого и до последнего, и этот порядок

не может быть таким же, как и в списке аргументов.

В приведенном ниже примере программы демонстрируется применение метода

Invoke() на практике. В этой программе два метода MyMeth() и MyMeth2() выполняются

параллельно посредством вызова метода Invoke(). Обратите внимание на

простоту организации данного процесса.

(***glava24\_8***)

class DemoParallel

{

//method task

static void MyMeth()

{

Console.WriteLine("MyMeth started.");

for (int count = 0; count < 5; count++)

{

Thread.Sleep(500);

Console.WriteLine("In method MyMeth count = " + count);

}

Console.WriteLine("MyMeth ended.");

}

//another task method

static void MyMeth2()

{

Console.WriteLine("MyMeth2 started.");

for (int count = 0; count < 5; count++)

{

Thread.Sleep(500);

Console.WriteLine("In method MyMeth2 count = " + count);

}

Console.WriteLine("MyMeth2 ended.");

}

static void Main()

{

Console.WriteLine("Main thread started.");

//do parallel methods

Parallel.Invoke(MyMeth, MyMeth2);

Console.WriteLine("Main thread ended.");

}

}

В данном примере особое внимание обращает на себя следующее обстоятельство:

выполнение метода Main() приостанавливается до тех пор, пока не произойдет

возврат из метода Invoke(). Следовательно, метод Main(), в отличие от методов

MyMeth() и MyMeth2(), не выполняется параллельно. Поэтому применять метод

Invoke() показанным здесь способом нельзя в том случае, если требуется, чтобы исполнение

вызывающего потока продолжалось.

В приведенном выше примере использовались именованные методы, но для вызова

метода Invoke() это условие не является обязательным. Ниже приведен переделанный

вариант той же самой программы, где в качестве аргументов в вызове метода

Invoke() применяются лямбда-выражения.

(***glava24\_8.1***)

class DemoParallel

{

static void Main()

{

Console.WriteLine("Main thread started.");

//do parallel methods

Parallel.Invoke(() =>

{

Console.WriteLine("MyMeth started.");

for (int count = 0; count < 5; count++)

{

Thread.Sleep(500);

Console.WriteLine("In method MyMeth count = " + count);

}

Console.WriteLine("MyMeth ended.");

},

() =>

{

Console.WriteLine("MyMeth2 started.");

for (int count = 0; count < 5; count++)

{

Thread.Sleep(500);

Console.WriteLine("In method MyMeth2 count = " + count);

}

Console.WriteLine("MyMeth2 ended.");

}

);

Console.WriteLine("Main thread ended.");

}

}

Эта программа дает результат, похожий на результат выполнения предыдущей

программы.

**Применение метода For()**

В TPL параллелизм данных поддерживается, в частности, с помощью метода For(),

определенного в классе Parallel. Этот метод существует в нескольких формах. Его

рассмотрение мы начнем с самой простой формы, приведенной ниже:

public static ParallelLoopResult

For(int fromInclusive, int toExclusive, Action<int> body)

где *fromInclusive* обозначает начальное значение того, что соответствует переменной

управления циклом; оно называется также итерационным, или индексным, значением;

a *toExclusive —* значение, на единицу больше конечного. На каждом шаге

цикла переменная управления циклом увеличивается на единицу. Следовательно,

цикл постепенно продвигается от начального значения *fromInclusive* к конечному

значению *toExclusive* минус единица. Циклически выполняемый код указывается

методом, передаваемым через параметр *body.* Этот метод должен быть совместим с

делегатом Action<int>, объявляемым следующим образом.

public delegate void Action<in T>(T obj)

Для метода For() обобщенный параметр T должен быть, конечно, типа int.

Значение, передаваемое через параметр *obj,* будет следующим значением переменной

управления циклом. А метод, передаваемый через параметр *body,* может быть

именованным или анонимным. Метод For() возвращает экземпляр объекта типа

ParallelLoopResult, описывающий состояние завершения цикла. Для простых циклов

этим значением можно пренебречь.

Главная особенность метода For() состоит в том, что он позволяет, когда такая возможность

имеется, распараллелить исполнение кода в цикле. А это, в свою очередь, может

привести к повышению производительности. Например, процесс преобразования

массива в цикле может быть разделен на части таким образом, чтобы разные части массива

преобразовывались одновременно. Следует, однако, иметь в виду, что повышение

производительности не гарантируется из-за отличий в количестве доступных процессоров

в разных средах выполнения, а также из-за того, что распараллеливание мелких

циклов может составить издержки, которые превышают сэкономленное время.

В приведенном ниже примере программы демонстрируется применение метода

For() на практике. В начале этой программы создается массив data, состоящий из

1000000000 целых значений. Затем вызывается метод For(), которому в качестве "тела"

цикла передается метод MyTransform(). Этот метод состоит из ряда операторов, выполняющих

произвольные преобразования в массиве data. Его назначение — сымитировать

конкретную операцию. Как будет подробнее пояснено несколько ниже, выполняемая

операция должна быть нетривиальной, чтобы параллелизм данных принес

какой-то положительный эффект. В противном случае последовательное выполнение

цикла может завершиться быстрее.

Эта программа состоит из двух циклов. В первом, стандартном, цикле for инициализируется

массив data. А во втором цикле, выполняемом параллельно методом

For(), над каждым элементом массива data производится преобразование. Как упоминалось

выше, это преобразование носит произвольный характер и выбрано лишь

для целей демонстрации. Метод For() автоматически разбивает вызовы метода

MyTransform() на части для параллельной обработки отдельных порций данных,

хранящихся в массиве. Следовательно, если запустить данную программу на компьютере

с двумя доступными процессорами или больше, то цикл преобразования данных

в массиве может быть выполнен методом For() параллельно.

Следует, однако, иметь в виду, что далеко не все циклы могут выполняться эффективно,

когда они распараллеливаются. Как правило, мелкие циклы, а также циклы,

состоящие из очень простых операций, выполняются быстрее последовательным способом,

чем параллельным. Именно поэтому цикл for инициализации массива данных

не распараллеливается методом For() в рассматриваемой здесь программе. Распараллеливание

мелких и очень простых циклов может оказаться неэффективным потому,

что время, требующееся для организации параллельных задач, а также время, расходуемое

на переключение контекста, превышает время, экономящееся благодаря параллелизму.

В подтверждение этого факта в приведенном ниже примере программы.

(Короче в методе Parallel.For() i встроена уже, поэтому не надо указывать или инициализировать ее.)

(***glava24\_9***)

class DemoParallelFor

{

static int[] data;

//this method is body for parallel cycle

//operator of this cycle just use time CP for demonstration

static void MyTransform(int i)

{

data[i] = data[i] / 10;

if (data[i] < 10000) data[i] = 0;

if (data[i] > 10000 & data[i] < 20000) data[i] = 100;

if (data[i] > 20000 & data[i] < 30000) data[i] = 200;

if (data[i] > 30000) data[i] = 300;

}

static void Main()

{

Console.WriteLine("Main thread started.");

//create oject of Stopwatch

//to contain time of cycle

Stopwatch sw = new Stopwatch();

data = new int[100000000];

//initialazie data

sw.Start();

//parallel cycl with For()

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

sw.Stop();

Console.WriteLine("Parallel cycle init {0} seconds", sw.Elapsed.TotalSeconds);

sw.Reset();

sw.Start();

//initialize data int simple for

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

sw.Stop();

Console.WriteLine("Simple cycle {0} seconds", sw.Elapsed.TotalSeconds);

Console.WriteLine();

sw.Reset();

sw.Start();

//parallel cycl with For()

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

sw.Stop();

Console.WriteLine("Parallel cycle {0} seconds", sw.Elapsed.TotalSeconds);

sw.Reset();

sw.Start();

//ordered variant

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

sw.Stop();

Console.WriteLine("Ordered cycle {0} seconds", sw.Elapsed.TotalSeconds);

Console.WriteLine("Main thread ended.");

}

}

Прежде всего, обратите внимание на то, что параллельный вариант цикла инициализации

массива данных выполняется приблизительно в три раза медленнее, чем

последовательный. Дело в том, что в данном случае на операцию присваивания расходуется

так мало времени, что издержки на дополнительно организуемое распараллеливание

превышают экономию, которую оно дает. Обратите далее внимание на то,

что параллельный вариант цикла преобразования данных выполняется быстрее, чем

последовательный. В данном случае экономия от распараллеливания с лихвой возмещает

издержки на его дополнительную организацию.

Oбратите внимание на применение класса Stopwatch для вычисления

времени выполнения цикла. Этот класс находится в пространстве имен System.

Diagnostics. Для того чтобы воспользоваться им, достаточно создать экземпляр его

объекта, а затем вызвать метод Start(), начинающий отчет времени, и далее — метод

Stop(), завершающий отсчет времени. А с помощью метода Reset() отсчет времени

сбрасывается в исходное состояние. Продолжительность выполнения можно получить

различными способами. В рассматриваемой здесь программе для этой цели использовано

свойство Elapsed, возвращающее объект типа TimeSpan. С помощью этого

объекта и свойства TotalSeconds время отображается в секундах, включая и доли секунды.

Как показывает пример рассматриваемой здесь программы, класс Stopwatch

оказывается весьма полезным при разработке параллельно исполняемого кода.

Как упоминалось выше, метод For() возвращает экземпляр объекта типа

ParallelLoopResult. Эго структура, в которой определяются два следующих свойства.

public bool IsCompleted { get; }

public Nullable<long> LowestBreakIteration { get; }

Свойство IsCompleted будет иметь логическое значение true, если выполнены

все шаги цикла. Иными словами, при нормальном завершении цикла это свойство

будет содержать логическое значение true. Если же выполнение цикла прервется

раньше времени, то данное свойство будет содержать логическое значение false.

Свойство LowestBreakIteration будет содержать наименьшее значение переменной

управления циклом, если цикл прервется раньше времени вызовом метода

ParallelLoopState.Break().

Для доступа к объекту типа ParallelLoopState следует использовать форму метода

For(), делегат которого принимает в качестве второго параметра текущее состояние

цикла. Ниже эта форма метода For() приведена в простейшем виде.

public static ParallelLoopResult For(int fromInclusive, int toExclusive,

ActionCint, ParallelLoopState> body)

В данной форме делегат Action, описывающий тело цикла, определяется следующим

образом.

public delegate void Action<in T1, in T2>(T argl, T2 arg2)

Для метода For() обобщенный параметр T1 должен быть типа int, а обобщенный

параметр Т2 — типа ParallelLoopState. Всякий раз, когда делегат Action вызывается,

текущее состояние цикла передается в качестве аргумента *аrg2.*

Для преждевременного завершения цикла следует воспользоваться методом Break(),

вызываемым для экземпляра объекта типа ParallelLoopState внутри тела цикла,

определяемого параметром *body.* Метод Break() объявляется следующим образом.

public void Break()

Вызов метода Break() формирует запрос на как можно более раннее прекращение

параллельно выполняемого цикла, что может произойти через несколько шагов цикла

после вызова метода Break(). Но все шаги цикла до вызова метода Break() все же выполняются.

Следует, также иметь в виду, что отдельные части цикла могут и не выполняться

параллельно. Так, если выполнено 10 шагов цикла, то это еще не означает, что

все эти 10 шагов представляют 10 первых значений переменной управления циклом.

Прерывание цикла, параллельно выполняемого методом For(), нередко оказывается

полезным при поиске данных. Так, если искомое значение найдено, то продолжать

выполнение цикла нет никакой надобности. Прерывание цикла может оказаться

полезным и в том случае, если во время очередной операции встретились недостоверные

данные.

В приведенном ниже примере программы демонстрируется применение метода

Break() для прерывания цикла, параллельно выполняемого методом For().

Это вариант предыдущего примера, переработанный таким образом, чтобы метод

MyTransform() принимал теперь объект типа ParallelLoopState в качестве своего

параметра, а метод Break() вызывался при обнаружении отрицательного значения

в массиве данных. Отрицательное значение, по которому прерывается выполнение

цикла, вводится в массив data внутри метода Main(). Далее проверяется состояние

завершения цикла преобразования данных. Свойство IsCompleted будет содержать

логическое значение false, поскольку в массиве data обнаруживается отрицательное

значение. При этом на экран выводится номер шага, на котором цикл был прерван.

(В этой программе исключены все избыточные циклы, применявшиеся в ее предыдущей

версии, а оставлены только самые эффективные из них: последовательно выполняемый

цикл инициализации и параллельно выполняемый цикл преобразования.)

(***glava24\_10***)

class DemoParallelForWithLoopResult

{

static int[] data;

//body of this method is for parallel cycle

//operators this cycle just use CP for demonstration

static void MyTransform(int i, ParallelLoopState pls)

{

//break cycle if negative value

if (data[i] < 0) pls.Break();

data[i] = data[i] / 10;

if (data[i] < 10000) data[i] = 0;

if (data[i] > 10000 & data[i] < 20000) data[i] = 100;

if (data[i] > 20000 & data[i] < 30000) data[i] = 200;

if (data[i] > 30000) data[i] = 300;

}

static void Main()

{

Console.WriteLine("Main thread started.");

data = new int[100000000];

//init data

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

//place negativ value in data

data[1000] = -10;

//parallel variant init array in cycle

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

//check if cycle is done

if (!loopResult.IsCompleted)

Console.WriteLine("\nCycle stoped, bcs of negative number" +

" step #" + loopResult.LowestBreakIteration + "\n");

Console.WriteLine("Main thread done.");

}

}

Как следует из приведенного выше результата, цикл преобразования данных преждевременно

завершается после 1000 шагов. Дело в том, что метод Break() вызывается

внутри метода MyTransform() при обнаружении в массиве данных отрицательного

значения.

Помимо двух описанных выше форм метода For() существует и ряд других его

форм. В одних из этих форм допускается указывать различные дополнительные параметры,

а в других — использовать параметры типа long вместо int для пошагового

выполнения цикла. Имеются также формы метода For(), предоставляющие такие дополнительные

преимущества, как, например, возможность указывать метод, вызываемый

по завершении потока каждого цикла.

И еще одно, последнее замечание: если требуется остановить цикл, параллельно

выполняемый методом For(), не обращая особого внимания на любые шаги цикла,

которые еще могут быть в нем выполнены, то для этой цели лучше воспользоваться

методом Stop(), чем методом Break().

**Применение метода ForEach()**

Используя метод ForEach(), можно создать распараллеливаемый вариант цикла

foreach. Существует несколько форм метода ForEach(). Ниже приведена простейшая

форма его объявления:

public static ParallelLoopResult

ForEach<TSource>(IEnumerable<TSource> source,

Action<TSource> body)

где *source* обозначает коллекцию данных, обрабатываемых в цикле, a *body —* метод,

который будет выполняться на каждом шаге цикла. Как пояснялось ранее в этой книге,

во всех массивах, коллекциях (описываемых в главе 25) и других источниках данных

поддерживается интерфейс IEnumerable<T>. Метод, передаваемый через параметр

*body,* принимает в качестве своего аргумента значение или ссылку на каждый обрабатываемый

в цикле элемент массива, но не его индекс. А в итоге возвращаются сведения

о состоянии цикла.

Аналогично методу For(), параллельное выполнение цикла методом

ForEach() можно остановить, вызвав метод Break() для экземпляра объекта типа

ParallelLoopState, передаваемого через параметр body, при условии, что используется

приведенная ниже форма метода ForEach().

public static ParallelLoopResult

ForEach<TSource>(IEnumerable<TSource> source,

Actior<TSource, ParallelLoopState> body);

В приведенном ниже примере программы демонстрируется применение метода

ForEach() на практике. Как и прежде, в данном примере создается крупный массив

целых значений. А отличается данный пример от предыдущих тем, что метод, выполняющийся

на каждом шаге цикла, просто выводит на консоль значения из массива.

Как правило, метод WriteLine() в распараллеливаемом цикле не применяется, потому

что ввод-вывод на консоль осуществляется настолько медленно, что цикл оказывается

полностью привязанным к вводу-выводу. Но в данном примере метод WriteLine()

применяется исключительно в целях демонстрации возможностей метода ForEach().

При обнаружении отрицательного значения выполнение цикла прерывается вызовом

метода Break(). Несмотря на то что метод Break() вызывается в одной задаче, другая

задача может по-прежнему выполняться в течение нескольких шагов цикла, прежде

чем он будет прерван, хотя это зависит от конкретных условий работы среды выполнения.

(***glava24\_11***)

class DemoParallelForWithLoopResult

{

static int[] data;

//method for parallel cycle

//variable v passed value of arra elements

//not index of this element

static void DisplayData(int v, ParallelLoopState pls)

{

//break if negative number

if (v < 0) pls.Break();

Console.WriteLine("Value: " + v);

}

static void Main()

{

Console.WriteLine("Main thread");

data = new int[100000000];

//init data

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

//place negative value in data

data[100000] = -10;

//use parallel cycle method ForEach()

//to output data on the screen

ParallelLoopResult loopResult = Parallel.ForEach(data, DisplayData);

//check if cycle is done

if (!loopResult.IsCompleted)

Console.WriteLine("\nCycle break because of negativ value." +

" Step #" + loopResult.LowestBreakIteration + ".\n");

Console.WriteLine("Main thread is done.");

}

}

В приведенной выше программе именованный метод применяется в качестве делегата,

представляющего "тело" цикла. Но иногда удобнее применять анонимный метод, лямбда выражение.