PLINQ представляет собой параллельный вариант языка интегрированных запросов

LINQ и тесно связан с библиотекой TPL. PLINQ применяется, главным образом, для

достижения параллелизма данных внутри запроса. Как станет ясно из дальнейшего,

сделать это совсем не трудно. Как и TPL, тема PLINQ довольно обширна и многогранна,

поэтому в этой главе представлены лишь самые основные понятия данного языка.

**Класс ParallelEnumerable**

Основу PLINQ составляет класс ParallelEnumerable, определенный в пространстве

имен System.Linq. Это статический класс, в котором определены многие

методы расширения, поддерживающие параллельное выполнение операций. По существу,

он представляет собой параллельный вариант стандартного для LINQ класса

Enumerable. Многие его методы являются расширением класса ParallelQuery, а некоторые

из них возвращают объект типа ParallelQuery. В классе ParallelQuery

инкапсулируется последовательность операций, поддерживающая параллельное выполнение.

Имеются как обобщенный, так и необобщенный варианты данного класса.

Мы не будем обращаться к классу ParallelQuery непосредственно, а воспользуемся

несколькими методами класса ParallelEnumerable. Самый главный из них, метод

AsParallel(), описывается в следующем разделе.

**Распараллеливание запроса методом AsParallel()**

Едва ли не самым удобным средством PLINQ является возможность просто создавать

параллельный запрос. Нужно лишь вызвать метод AsParallel() для источника

данных. Метод AsParallel() определен в классе ParallelEnumerable и возвращает

источник данных, инкапсулированный в экземпляре объекта типа ParallelQuery.

Это дает возможность поддерживать методы расширения параллельных запросов. После

вызова данного метода запрос разделяет источник данных на части и оперирует

с каждой из них таким образом, чтобы извлечь максимальную выгоду из распараллеливания.

(Если распараллеливание оказывается невозможным или неприемлемым, то

запрос, как обычно, выполняется последовательно.) Таким образом, добавления в исходный

код единственного вызова метода AsParallel() оказывается достаточно для

того, чтобы превратить последовательный запрос LINQ в параллельный запрос LINQ.

Для простых запросов это единственное необходимое условие.

Существуют как обобщенные, так и необобщенные формы метода AsParallel().

Ниже приведена простейшая обобщенная его форма:

public static ParallelQuery AsParallel(this IEnumerable source)

public static ParallelQuery<TSource>

AsParallel<TSource>(this IEnumerable<TSource> source)

где TSource обозначает тип элементов в последовательном источнике данных

*source.*

Ниже приведен пример, демонстрирующий простой запрос PLINQ.

(***glava24\_12***)

class PLINQDemo

{

static void Main()

{

int[] data = new int[10000000];

//init array data pos nums

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

//now input neg values in data

data[1000] = -1;

data[14000] = -2;

data[15000] = -3;

data[676000] = -4;

data[8024540] = -5;

data[9908000] = -6;

//use Plinq to find negative vals

var negatives = from val in data.AsParallel()

where val < 0

select val;

foreach (var v in negatives)

Console.Write(v + " ");

Console.WriteLine();

}

}

Эта программа начинается с создания крупного массива data, инициализируемого

целыми положительными значениями. Затем в него вводится ряд отрицательных

значений. А далее формируется запрос на возврат последовательности отрицательных

значений.

var negatives = from val in data.AsParallel()

where val < 0

select val;

В этом запросе метод AsParallel() вызывается для источника данных, в качестве

которого служит массив data. Благодаря этому разрешается параллельное выполнение

операций над массивом data, а именно: поиск отрицательных значений параллельно

в нескольких потоках. По мере обнаружения отрицательных значений они добавляются

в последовательность вывода. Эго означает, что порядок формирования последовательности

вывода может и не отражать порядок расположения отрицательных

значений в массиве data.

Следует, однако, иметь в виду, что из-за отличий

в степени загрузки задачами, количества доступных процессоров и прочих факторов

системного характера могут быть получены разные результаты. А самое главное,

что результирующая последовательность совсем не обязательно будет отражать порядок

формирования исходной последовательности.

**Применение метода AsOrdered()**

Как отмечалось в предыдущем разделе, по умолчанию порядок формирования

результирующей последовательности в параллельном запросе совсем не обязательно

должен отражать порядок формирования исходной последовательности. Более того,

результирующую последовательность следует рассматривать как практически неупорядоченную.

Если же результат должен отражать порядок организации источника

данных, то его нужно запросить специально с помощью метода AsOrdered(), определенного

в классе ParallelEnumerable.

public static ParallelQuery AsOrdered(this ParallelQuery source)

public static ParallelQuery<TSource>

AsOrdered<TSource>(this ParallelQuery<TSource> source)

где T S o u r c e обозначает тип элементов в источнике данных *source.* Метод

AsOrdered() можно вызывать только для объекта типа ParallelQuery, поскольку

он является методом расширения класса ParallelQuery.

Для того чтобы посмотреть, к какому результату может привести применение метода

AsOrdered(), подставьте его вызов в приведенный ниже запрос из предыдущего

примера программы.

var negatives = from val in data.AsParallel().AsOrdered()

where val < 0

select val;

После выполнения программы порядок следования элементов в результирующей

последовательности будет отражать порядок их расположения в исходной последовательности.

**Отмена параллельного запроса**

Параллельный запрос отменяется таким же образом, как и задача. И в том и в другом

случае отмена опирается на структуру CancellationToken, получаемую из класса

CancellationTokenSource. Получаемый в итоге признак отмены передается запросу с

помощью метода WithCancellation(). Отмена параллельного запроса производится

методом Cancel(), который вызывается для источника признаков отмены. Главное отличие

отмены параллельного запроса от отмены задачи состоит в следующем: когда параллельный

запрос отменяется, он генерирует исключение OperationCanceledException,

а не AggregateException. Но в тех случаях, когда запрос способен сгенерировать несколько

исключений, исключение OperationCanceledException может быть объединено

в совокупное исключение AggregateException. Поэтому отслеживать лучше оба

вида исключений.

public static ParallelQuery<TSource>

WithCancellation<TSource>(

this ParallelQuery<TSource> source,

CancellationToken CancellationToken)

где *source* обозначает вызывающий запрос, a *CancellationToken —* признак отмены.

Этот метод возвращает запрос, поддерживающий указанный признак отмены.

В приведенном ниже примере программы демонстрируется порядок отмены

параллельного запроса, сформированного в программе из предыдущего примера.

В данной программе организуется отдельная задача, которая ожидает в течение 100

миллисекунд, а затем отменяет запрос. Отдельная задача требуется потому, что цикл

foreach, в котором выполняется запрос, блокирует выполнение метода Main() до

завершения цикла.

(***glava24\_13***)

using System;

using System.Linq;

using System.Threading;

using System.Threading.Tasks;

class PLINQCancelDemo

{

static void Main()

{

CancellationTokenSource cancelTokSrc = new CancellationTokenSource();

int[] data = new int[10000000];

//init array data pos nums

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

//now input neg values in data

data[1000] = -1;

data[14000] = -2;

data[15000] = -3;

data[676000] = -4;

data[8024540] = -5;

data[9908000] = -6;

//use Plinq to find negative vals

var negatives = from val in data.AsParallel().

WithCancellation(cancelTokSrc.Token)

where val < 0

select val;

//create task to cancel after 100mseconds

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

{

Thread.Sleep(10);

cancelTokSrc.Cancel();

});

try

{

foreach (var v in negatives)

Console.Write(v + " ");

}

catch(OperationCanceledException exc)

{

Console.WriteLine(exc.Message);

}

catch(AggregateException exc)

{

Console.WriteLine(exc);

}

finally

{

cancelTask.Wait();

cancelTokSrc.Dispose();

cancelTask.Dispose();

}

Console.WriteLine();

}

}

Если запрос отменяется до его завершения, то на экран выводится только сообщение об исключительной ситуации.

**Другие средства PLINQ**

Как упоминалось ранее, PLINQ представляет собой довольно крупную подсистему.

Это объясняется отчасти той гибкостью, которой обладает PLINQ. В PLINQ доступны

и многие другие средства, помогающие подстраивать параллельные запросы под

конкретную ситуацию. Так, при вызове метода WithDegreeOfParallelism() можно

указать максимальное количество процессоров, выделяемых для обработки запроса,

а при вызове метода AsSequential() — запросить последовательное выполнение

части параллельного запроса. Если вызывающий поток, ожидающий результатов от

цикла foreach, не требуется блокировать, то для этой цели можно воспользоваться

методом ForAll(). Все эти методы определены в классе ParallelEnumerable.

А в тех случаях, когда PLINQ должен по умолчанию поддерживать последовательное

выполнение, можно воспользоваться методом WithExecutionMode(), передав ему в

качестве параметра признак ParallelExecutionMode.ForceParallelism.

**Вопросы эффективности PLINQ**

Далеко не все запросы выполняются быстрее только потому, что они распараллелены.

Как пояснялось ранее в отношении TPL, издержки, связанные с созданием

параллельных потоков и управлением их исполнением, могут "перекрыть" все преимущества,

которые дает распараллеливание. Вообще говоря, если источник данных

оказывается довольно мелким, а требующаяся обработка данных — очень короткой,

то внедрение параллелизма может и не привести к ускорению обработки запроса.

Поэтому за рекомендациями по данному вопросу следует обращаться к информации

корпорации Microsoft.