Технология PLINQ (.NET 4.0)

Общее описание

Технология PLINQ (Parallel LINQ) обеспечивает автоматическое распараллеливание локальных запросов LINQ. Для использования PLINQ достаточно вызвать метод AsParallel() для входной последовательности, после чего использовать обычные запросы LINQ.

Пример. Нахождение простых чисел в диапазоне от 3 до 100000:

```
var parallelQuery =
from n in Enumerable.Range (3, 100000-3)
  .AsParallel()
  where Enumerable.Range(0, (int)Math.Sqrt(n)/2 + 1)
    .Select(e => e==0 ? 2 : e*2+1).All(i => n % i>0)
  select n;
int[] primes = parallelQuery.ToArray();
```

Метод AsParallel преобразует последовательность типа IEnumerable<T> в последовательность типа ParallelQuery<T>. Этот метод является методом расширения класса System.Linq.ParallelEnumerable, в котором также реализованы «параллельные» варианты всех стандартных запросов LINQ to Objects.

Метод AsSequential() отключает параллельный режим (этот метод преобразует последовательность типа ParallelQuery<T> в последовательность IEnumerable<T>). Он может применяться, например, в случае, когда результирующая параллельная последовательность имеет гарантированно небольшое число элементов, и для ее последующей обработки параллельный режим может привести к неоправданным накладным расходам. Метод AsEnumerable работает так же, как и AsSequential.

Для создания «на пустом месте» параллельных последовательностей можно использовать методы класса ParallelEnumerable: Range(int start, int count) (возвращает последовательность типа ParallelQuery<int>) и Repeat<T>(T element, int count) (возвращает последовательность типа ParallelQuery<T>).

Для запросов, в которых используются две входные последовательности (Join, GroupJoin, Concat, Union, Intersect, Except, Zip) необходимо вызвать метод AsParallel для *каждой* входной последовательности (в противном случае будет возбуждено исключение).

Для последующих запросов в цепочке вызывать AsParallel не требуется, так как любая «параллельная» реализация запроса возвращает «параллельную» последовательность. Более того, лишние вызовы AsParallel делают запрос менее эффективным, поскольку приводят к дополнительным действиям по распараллеливанию (слиянию параллельных фрагментов последовательности и их новому разбиению).

Некоторые запросы не могут быть эффективно распараллелены. В этом случае PLINQ использует обычную, непараллельную версию соответствующих запросов.

Технология PLINQ реализована только для локальных запросов LINQ to Objects. Запросы PLINQ, как и запросы LINQ to Objects, являются ленивыми.

Сохранение порядка следования элементов

Побочный эффект распараллеливания состоит в том, что при последующем слиянии данных их порядок может отличаться от порядка, в котором они располагались бы при выполнении непараллельного запроса. Если порядок элементов в результирующем запросе является существенным, то необходимо явно указать это с помощью метода AsOrdered, вызываемого сразу после метода AsParallel. Однако это снижает эффективность распараллеливания, поскольку заставляет PLINQ запоминать исходные позиции элементов и использовать эту информацию при слиянии результирующих элементов.

Если в дальнейшем сохранение исходного порядка не требуется, то следует вызвать метод AsUnordered, который отключает контроль за сохранением исходного порядка и тем самым повышает эффективность распараллеливания.

Ограничения на распараллеливание запросов

Перечисленные ниже запросы LINQ не выполняются в параллельном режиме, если элементы обрабатываемой последовательности не располагаются на своих исходных позициях:

- Take, TakeWhile, Skip, SkipWhile,
- индексные варианты методов Select и SelectMany.

Поэтому данные запросы следует вызывать в начале обработки последовательности, до того как в ней произойдет изменение расположения элементов (например, в результате выполнения запроса Where).

Перечисленные ниже запросы реализуют специальные методы распараллеливания (разбиения исходных последовательностей на части, обрабатываемые в отдельном потоке), которые могут приводить к более медленной работе по сравнению с последовательным вариантом:

- Join, GroupBy, GroupJoin, Distinct, Union, Intersect, Except.

Варианты запроса Aggregate с явным указанием параметра seed не обеспечивают распараллеливание, однако имеются специальные параллельные реализации данного запроса (см. далее).

Явное включение параллельного режима и задание числа потоков

Даже если запрос допускает распараллеливание, оно не гарантируется (PLINQ может выполнить запрос в последовательном режиме, если сочтет, что это обеспечит более эффективное его выполнение). Можно явным образом «заставить» PLINQ распараллеливать запросы, если после запроса AsParallel вызвать метод WithExecutionMode(ParallelExceutionMode.ForceParallelism).

Другое значение типа ParallelExecutionMode — Default — означает, что решение об использовании параллельного режима будет принимать сама система PLINQ.

Имеется метод WithDegreeOfParallelism(n), ограничивающий «степень параллелизма» (т. е. количество используемых потоков) значением n целого типа. Задается верхняя граница числа потоков; на практике это число может оказаться меньше (или запрос может быть выполнен в последовательном режиме). Метод WithDegreeOfParallelism должен вызываться сразу после вызова метода AsParallel; если в дальнейшем требуется изменить значение n, то необходимо еще раз вызвать пару методов AsParallel и WithDegreeOfParallelism.

Побочные эффекты в PLINQ

При использовании запросов PLINQ не следует использовать внешние переменные, например:

```
int i = 0;
var query = from n in Enumerable.Range(0, 999)
  .AsParallel() select n * i++;
```

Даже если с помощью процедур синхронизации обеспечить безопасный (бесконфликтный) доступ к внешней переменной i, это не решит проблемы, поскольку не обеспечит соответствия текущего значения i индексу соответствующего элемента исходной последовательности. Добавление метода AsOrdered также не решит проблемы, так как этот метод гарантирует лишь, что элементы будут *добавлены в результирующую последовательность* в том же порядке, в котором они находились в исходной последовательности (их *обработка* в том же порядке не гарантируется).

Правильный вариант реализации предыдущего запроса:

```
var query = Enumerable.Range(0,999).AsParallel()
  .Select ((n, i) => n * i);
```

Распределение элементов по потокам

PLINQ поддерживает три варианта распределения входных элементов по потокам:

- *блочное (chunk) распределение* (динамическое распределение, средняя производительность),
- *диапазонное (range) распределение* (статическое распределение, производительность от низкой до высокой),
- *hash-распределение* (статическое распределение, низкая производительность).

Для запросов, требующих сравнения элементов (GroupBy, Join, GroupJoin, Intersect, Except, Union, Distinct) всегда используется hash-распределение.

Для других видов запросов можно явно выбирать один из вариантов: блочный или диапазонный. По умолчанию, если входная последовательность является индексируемой (массив или последовательность, реализующая интерфейс IList<T>), то используется диапазонный вариант, иначе используется блочный вариант.

Диапазонное распределение обеспечивает более быструю обработку длинных последовательностей, для которых обработка каждого элемента требует примерно одинакового времени. В других случаях более быстрым будет блочное распределение.

Для явного задания диапазонного распределения надо выполнить одно из следующих действий:

- если запрос начинается с вызова Enumerable.Range, то достаточно заменить его на ParallelEnumerable.Range;
- в других ситуациях достаточно вызвать для исходной (непараллельной) последовательности метод ToList или ToArray (или убедиться, что исходная последовательность является массивом или реализует интерфейс List<T>).

Для явной реализации блочного распределения достаточно «обернуть» исходную последовательность в объект Partitioner (из пространства имен System.Collections):

Partitioner.Create(input_sequence, true).AsParallel() …

Блочное разбиение работает следующим образом: каждый рабочий поток периодически получает очередную порцию (chunk) элементов из исходной последовательности для обработки. PLINQ начинает с небольших порций (1-2 элемента), а затем увеличивает размер порций по мере дальнейшей обработки последовательности. Подобная схема обеспечивает равномерную загрузку потоков, однако требует синхронизации при доступе к исходной последовательности.

Диапазонное разбиение выполняется путем предварительного распределения всех элементов (примерно поровну) по потокам, поэтому в данном случае синхронизация не требуется. Однако при этом равномерная загрузка потоков не гарантируется (если время обработки различных элементов существенно различается).

При диапазонном разбиении PLINQ сам определяет способ распределения элементов по потокам. В частности, вместо разбиения исходной последовательности на большие смежные части может использовать разбиение по полосам (что окажется более эффективным, например, при параллельной реализации запроса TakeWhile).

Параллельное выполнение запроса Aggregate

Если в запросе явно не указывается параметр seed, то распараллеливание будет выполняться корректно только в случае, если агрегирующая операция является коммутативной и ассоциативной (поскольку для каждого потока используется свой аккумулятор, которые в результате объединяются).

Если параметр seed указан явно, то варианты запроса Aggregate с теми же параметрами, что и в LINQ to Objects, всегда выполняются последовательно, так как каждый элемент должен обрабатываться совместно с одним и тем же аккумулятором.

Однако в PLINQ предусмотрены новые варианты метода Aggregate с параметром seed, обеспечивающие его распараллеливание. В них используется так называемая *фабрика аккумуляторов*, обеспечивающая генерацию локального аккумулятора для каждого потока. Кроме того, необходимо предусмотреть специальную функцию, обеспечивающую объединение локальных аккумуляторов для получения итогового результата (глобального аккумулятора). В данном варианте Aggregate используются следующие параметры (все они являются делегатами):

- seedFactory —вызывается один раз в каждом потоке, возвращает локальный аккумулятор для каждого потока,
- updateAccumulatorFunc — вызывается для каждого элемента в потоке, добавляя его к локальному аккумулятору,
- combineAccumulatorFunc — вызывается один раз в каждом потоке, объединяет локальные аккумуляторы,
- resultSelector — параметр-делегат, обеспечивает завершающую обработку полученного глобального аккумулятора.

Имеется вариант метода, в котором вместо фабрики-делегата можно указать обычную переменную, однако надо учитывать, что ее значение будет передаваться во все потоки, и если это значение является ссылкой, то все потоки будут обращаться к одному и тому же объекту.

Пример. Рассмотрим следующий последовательный алгоритм подсчета частоты появления латинских символов в некоторой длинной строке:

```
int[] result =
  text.Aggregate(new int[26],
    (letterFrequencies, c) =>
    {
      int index = char.ToUpper (c) - 'A';
      if (index >= 0 && index <= 26)
        letterFrequencies[index]++;
      return letterFrequencies;
    });
```

Для его распараллеливания необходимо использовать новый вариант метода Aggregate, включенный в PLINQ:

```
int[] result =
  text.AsParallel().Aggregate (() => new int[26],
    (localFrequencies, c) =>
    {
      int index = char.ToUpper (c) - 'A';
      if (index >= 0 && index <= 26)
        localFrequencies [index]++;
      return localFrequencies;
    },
    (mainFreq, localFreq) =>
       mainFreq.Zip(localFreq,
         (f1, f2) => f1 + f2)
       .ToArray(),
    finalResult => finalResult);
```

Аккумулирование данных в каждом потоке изменяет локальный аккумулятор, но это не приводит к конфликтам, так как каждый локальный аккумулятор создается в своем потоке.

В данном примере использован метод Zip, включенный в интерфейс LINQ to Objects в версии .NET 4.0. Этот метод «объединяет» элементы двух последовательностей, имеющие одинаковые индексы (способ объединения задается вторым параметром-делегатом; «лишние» элементы более длинной последовательности не обрабатываются).