Класс Parallel

Введение

Инфраструктура PFX предоставляет базовую форму структурированного параллелизма через три статических метода в классе *Parallel*.

- *Parallel.Invoke*. Запускает массив делегатов параллельно.
- *Parallel.For*. Выполняет параллельный эквивалент цикла for языка C#.
- *Parallel.ForEach*. Выполняет параллельный эквивалент цикла foreach языка C#.

Все три метода блокируются вплоть до завершения всей работы. Как и с PLINQ, в случае необработанного исключения оставшиеся рабочие потоки останавливаются после их текущей итерации, а исключение (либо их набор) передается обратно вызывающему потоку внутри оболочки *AggregateException*.

Parallel.Invoke

Метод *Parallel.Invoke* запускает массив делегатов *Action* параллельно, после чего ожидает их завершения.

Как и в PLINQ, методы *Parallel.** оптимизированы для выполнения работы с интенсивными вычислениями, но не интенсивным вводом-выводом. Тем не менее загрузка двух веб-страниц за раз позволяет легко продемонстрировать использование метода *Parallel.Invoke*:

```
Parallel.Invoke(
  () => new WebClient()
        .DownloadFile("http://www.linqpad.net", "lp.html"),
  () => new WebClient()
        .DownloadFile("http://www.jaoo.dk", "jaoo.html"));
```

На первый взгляд код выглядит удобным сокращением для создания и ожидания двух привязанных к потокам объектов *Task*. Однако существует важное отличие: метод *Parallel.Invoke* будет

работать по-прежнему эффективно, даже если ему передать массив из миллиона делегатов. Причина в том, что он разбивает большое количество элементов на пакеты, которые назначает небольшому числу существующих объектов *Task*, а не создает отдельный объект *Task* для каждого делегата.

Как и со всеми методами класса *Parallel*, объединение результатов возлагается полностью на вас. Это значит, что вы должны помнить о безопасности в отношении потоков. Например, приведенный ниже код не является безопасным к потокам:

```
var data = new List<string>();
Parallel.Invoke(
  () => data.Add(new
WebClient().DownloadString("http://www.foo.com")),
  () => data.Add(new
WebClient().DownloadString("http://www.far.com")));
```

Помещение кода добавления в список внутрь блокировки решило бы проблему, но блокировка создаст узкое место в случае гораздо более крупных массивов быстро выполняющихся делегатов. Лучшее решение предусматривает использование безопасных к потокам коллекций, которые рассматриваются далее – идеальным вариантом в данном случае была бы коллекция *ConcurrentBag*.

Метод *Parallel.Invoke* также имеет перегруженную версию, принимающую объект *ParallelOptions*:

```
public static void Invoke (ParallelOptions options,
                           params Action[] actions);
```

С помощью объекта *ParallelOptions* можно вставить признак отмены, ограничить максимальную степень параллелизма и указать специальный планировщик задач. Признак отмены играет важную роль, когда выполняется (приблизительно) большее количество задач, чем есть процессорных ядер: при отмене все незапущенные делегаты будут отброшены. Однако любые уже

выполняющиеся делегаты продолжат свою работу вплоть до ее завершения.

Parallel.For и Parallel.ForEach

Методы *Parallel.For* и *Parallel.ForEach* реализуют эквиваленты циклов *for* и *foreach* из C#, но с выполнением каждой итерации параллельно, а не последовательно.

Следующий последовательный цикл *for*:

```
for (int i = 0; i < 100; i++)
  Foo(i);
```

распараллеливается примерно так:

```
Parallel.For (0, 100, i => Foo (i));
```

или еще проще:

```
Parallel.For (0, 100, Foo);
```

А представленный далее последовательный цикл *foreach*:

```
foreach (char c in "Hello, world")
  Foo(c);
```

распараллеливается следующим образом:

```
Parallel.ForEach ("Hello, world", Foo);
```

Рассмотрим практический пример. Если мы импортируем пространство имен *System.Security.Cryptography*, то сможем генерировать шесть строк с парами открытого и секретного ключей параллельно:

```
var keyPairs = new string[6];

Parallel.For(0, keyPairs.Length,
       i => keyPairs[i] = RSA.Create().ToXmlString(true));
```

Как и в случае *Parallel.Invoke*, методам *Parallel.For* и *Parallel.ForEach* можно передавать большое количество элементов работы и они будут эффективно распределены по нескольким задачам.

Сравнение внешних и внутренних циклов

Методы *Parallel.For* и *Parallel.ForEach* обычно лучше всего работают на внешних, а не на внутренних циклах. Причина в том, что посредством внешних циклов вы предлагаете распараллеливать более крупные порции работы, снижая накладные расходы по управлению. Распараллеливание сразу внутренних и внешних циклов обычно излишне. В следующем примере для получения ощутимой выгоды от распараллеливания внутреннего цикла обычно требуется более 100 ядер:

```
Parallel.For(0, 100, i =>
{
  Parallel.For(0, 50, j => Foo(i, j)); // Внутренний цикл лучше
});                                    // выполнять
последовательно.
```

Индексированная версия *Parallel.ForEach*

Временами полезно знать индекс итерации цикла. В случае последовательного цикла *foreach* это легко:

```
int i = 0;
foreach (char c in "Hello, world")
  Console.WriteLine(c.ToString() + i++);
// выполнять последовательно.
```

Однако инкрементирование разделяемой переменной не является безопасным к потокам в параллельном контексте. Взамен должна использоваться следующая версия *ForEach*:

```
public static ParallelLoopResult ForEach<TSource>(
  IEnumerable<TSource> source,
  Action<TSource, ParallelLoopState, long> body)
```

Мы проигнорируем класс *ParallelLoopState* (он будет рассматриваться далее). Пока что нас интересует третий параметр типа *long*, который отражает индекс цикла:

```
Parallel.ForEach("Hello, world", (c, state, i) =>
{
  Console.WriteLine(c.ToString() + i);
});
```

Чтобы применить такой прием в практическом примере, вернемся к программе проверки орфографии, которую мы писали с помощью PLINQ. Следующий код загружает словарь и массив с миллионом слов для целей тестирования:

```
if (!File.Exists("WordLookup.txt")) // Содержит около 150 000
слов
  new WebClient().DownloadFile(
    "http://www.albahari.com/ispell/allwords.txt",
    "WordLookup.txt");

var wordLookup = new HashSet<string>(
  File.ReadAllLines("WordLookup.txt"),
  StringComparer.InvariantCultureIgnoreCase);

var random = new Random();
string[] wordList = wordLookup.ToArray();

string[] wordsToTest = Enumerable.Range(0, 1000000)
  .Select(i => wordList[random.Next(0, wordList.Length)])
  .ToArray();

wordsToTest[12345] = "woozsh";      // Внесение пары
wordsToTest[23456] = "wubsie";       // орфографических ошибок.
```

Мы можем выполнить проверку орфографии в массиве *wordsToTest* с использованием индексированной версии *Parallel.ForEach*:

```
var misspellings = new ConcurrentBag<Tuple<int, string>>();

Parallel.ForEach(wordsToTest, (word, state, i) =>
{
  if (!wordLookup.Contains(word))
    misspellings.Add(Tuple.Create((int)i, word));
});
```

Обратите внимание, что мы должны объединять результаты в безопасную к потокам коллекцию: по сравнению с применением PLINQ необходимость в таком действии считается недостатком. Преимущество перед PLINQ связано с тем, что мы избегаем использования индексированной операции запроса *Select*, которая менее эффективна, чем индексированная версия метода *ForEach*.

ParallelLoopState: раннее прекращение циклов

Поскольку тело цикла в параллельном методе *For* или *ForEach* представляет собой делегат, выйти из цикла до его полного завершения с помощью оператора *break* не получится. Вместо этого придется вызвать метод *Break* или *Stop* на объекте *ParallelLoopState*.

Получить объект *ParallelLoopState* довольно просто: все версии методов *For* и *ForEach* перегружены для приема тела цикла типа *Action<TSource, ParallelLoop State>*. Таким образом, для распараллеливания следующего цикла:

```
foreach (char c in "Hello, world")
  if (c == ',')
    break;
  else
    Console.Write(c);
```

нужно поступить так:

```
Parallel.ForEach("Hello, world", (c, loopState) =>
{
  if (c == ',')
    loopState.Break();
  else
    Console.Write(c);
});
// ВЫВОД: Hlloe
```

В выводе несложно заметить, что тела циклов могут завершаться в произвольном порядке. Помимо такого отличия вызов *Break* выдает, по меньшей мере, те же самые элементы, что и при последовательном выполнении цикла: приведенный пример будет всегда выводить минимум буквы H, e, l, l и o в каком-нибудь порядке. Напротив, вызов *Stop* вместо *Break* приводит к принудительному завершению всех потоков сразу после их текущей итерации. В данном примере вызов *Stop* может дать подмножество букв H, e, l, l и o, если другой поток отстал. Вызов *Stop* полезен, когда обнаружено то, что требовалось найти, или выяснилось, что что-то пошло не так, и потому результаты просматриваться не будут.

- Методы *Parallel.For* и *Parallel.ForEach* возвращают объект *ParallelLoopResult*, который открывает доступ к свойствам с именами *IsCompleted* и *LowestBreakIteration*. Они сообщают, полностью ли завершился цикл, и если это не так, то на какой итерации он был прекращен. Если свойство *LowestBreakIteration* возвращает null, тогда в цикле вызывался метод *Stop* (а не *Break*).

В случае длинного тела цикла может потребоваться прервать другие потоки где-то на полпути тела метода при раннем вызове *Break* или *Stop*, что реализуется за счет опроса свойства *ShouldExitCurrentIteration* в различных местах кода. Указанное свойство принимает значение true немедленно после вызова *Stop* или очень скоро после вызова *Break*.

- Свойство *ShouldExitCurrentIteration* также становится равным true после запроса отмены либо в случае генерации исключения в цикле.

Свойство *IsExceptional* позволяет узнать, произошло ли исключение в другом потоке. Любое необработанное исключение приведет к останову цикла после текущей итерации каждого потока: чтобы избежать его, вы должны явно обрабатывать исключения в своем коде.

Оптимизация посредством локальных значений

Методы *Parallel.For* и *Parallel.ForEach* предлагают набор перегруженных версий, которые работают с аргументом обобщенного типа по имени *TLocal*. Такие перегруженные версии призваны помочь оптимизировать объединение данных из циклов с интенсивными итерациями.

На практике данные методы редко востребованы, т.к. их целевые сценарии в основном покрываются PLINQ (что, в принципе, хорошо, поскольку иногда их перегруженные версии выглядят слегка устрашающими).

По существу вот в чем заключается проблема: предположим, что необходимо просуммировать квадратные корни чисел от 1 до 10 000 000. Вычисление 10 миллионов квадратных корней легко распараллеливается, но суммирование их значений – дело хлопотное, т.к. обновление итоговой суммы должно быть помещено внутрь блокировки:

```
object locker = new object();
double total = 0;
Parallel.For(1, 10000000,
       i => { lock (locker) total += Math.Sqrt(i); });
```

Выигрыш от распараллеливания более чем нивелируется ценой получения 10 миллионов блокировок, а также блокированием результата.

Однако в реальности нам не нужны 10 миллионов блокировок. Представьте себе команду волонтеров по сборке большого объема мусора. Если все работники совместно пользуются единственным мусорным ведром, то хождения к нему и состязания сделают процесс крайне неэффективным. Очевидное решение предусматривает снабжение каждого работника собственным или “локальным” мусорным ведром, которое время от времени опустошается в главный контейнер.

Именно таким способом работают версии *TLocal* методов *For* и *ForEach*. Волонтеры – это внутренние рабочие потоки, а локальное значение представляет локальное мусорное ведро.

Чтобы класс *Parallel* справился с этой работой, необходимо предоставить два дополнительных делегата.

1. Делегат, который указывает, каким образом инициализировать новое локальное значение.
2. Делегат, который указывает, каким образом объединять локальную агрегацию с главным значением.

Кроме того, вместо возвращения *void* делегат тела цикла должен возвращать новую агрегацию для локального значения. Ниже приведен переделанный пример:

```
object locker = new object();
double grandTotal = 0;
Parallel.For(1, 10000000,
  () => 0.0, // Инициализировать локальное значение.
  (i, state, localTotal) => // Делегат тела цикла. Обратите внимание,
    localTotal + Math.Sqrt(i), // что он возвращает новый локальный итог.
  localTotal => // Добавить локальное значение
    { lock (locker) grandTotal += localTotal; } // к главному значению.
);
```

Здесь по-прежнему требуется блокировка, но только вокруг агрегирования локального значения с общей суммой, что делает процесс гораздо эффективнее.

- Как утверждалось ранее, PLINQ часто хорошо подходит для таких сценариев. Распараллелить наш пример с помощью PLINQ можно было бы просто так:

```
ParallelEnumerable.Range (1, 10000000).Sum (i => Math.Sqrt (i))
```

(Обратите внимание, что мы применяем *ParallelEnumerable* для обеспечения разбиения на основе диапазонов: в данном случае оно улучшает производительность, потому что все числа требуют равного времени на обработку.)

В более сложных сценариях вместо *Sum* может использоваться LINQ-операция *Aggregate*. Если вы предоставите фабрику локальных начальных значений, то ситуация будет в чем-то аналогична предоставлению функции локальных значений для *Parallel.For*.