Шаблоны параллелизма Parallel

Класс `Parallel` предоставляет два наиболее распространенных шаблона параллельной обработки: параллельные циклы (`Parallel.For, Parallel.ForEach`) и параллельный запуск нескольких независимых задач (`Parallel.Invoke`). Реализация шаблонов построена на задачах (`tasks`), поэтому при использовании методов `Parallel` поддерживаются механизмы отмены с помощью объекта `CancellationToken` и обработка исключений типа `AggregateException`. Важной особенностью шаблонов является «императивность». Оператор, следующий за вызовом метода класса `Parallel`, будет выполняться только после завершения всех задач, неявно созданных в методе.

Parallel.Invoke

Вызов метода позволяет «запустить» (добавить в очередь готовых к выполнению задач) несколько рабочих элементов и дождаться завершения их работы.

Таким образом, вызов метода `Invoke`:

```
Parallel.Invoke(FuncOne, FuncTwo)
```

с помощью задач можно переписать следующим образом:

```
Task taskTwo = Task.Factory.StartNew(() => FuncTwo());
Task taskOne = Task.Factory.StartNew(() => FuncOne());
Task.WaitAll(taskOne, taskTwo);
```

Методы `Parallel` создают задачи не для каждого рабочего элемента, а для набора задач, которые будут выполняться в одном потоке. Поток, в котором осуществляется вызов метода `Invoke`, также используется для обработки элементов.

Использовать метод `Invoke` вместо непосредственного манипулирования с задачами полезно, когда рабочие элементы можно запустить одновременно (нет какой-либо подготовительной работы перед каждой задачей) и вызывающий поток должен дождаться завершения работы всех рабочих элементов. Такая ситуация возникает в алгоритмах типа «разделяй и властвуй» (быстрая сортировка, обработка графов).

В качестве аргументов метода `Parallel.Invoke` можно указывать методы, лямбда-выражения, а также массив делегатов типа `Action`:

```
Action[] actions = new Action[4];
actions[0] = new Action(() => Console.WriteLine("one"));
actions[1] = new Action(() => Console.WriteLine("two"));
actions[2] = new Action(() => Console.WriteLine("three"));
actions[3] = new Action(() => Console.WriteLine("four"));
Parallel.Invoke(actions);
```

В качестве параметров выполнения в методе `Invoke` можно указать токен отмены, максимальную степень параллелизма и используемый планировщик:

```
ParallelOptions  pOptions = new ParallelOptions()
     {
          maxDegreeOfParallelism = 4,
          cancellationToken = cToken,
          TaskScheduler = tScheduler
     };
Parallel.Invoke(pOptions, actions);
```

Параллельные циклы Parallel.For и Parallel.ForEach

Методы `Parallel.For` и `Parallel.ForEach` позволяют распараллелить обработку итераций или обработку элементов какой-либо структуры данных перечислимого типа (массив, список). Методы имеют несколько перегрузок, позволяющие настраивать параллелизм обработки.

В случае независимости обработки элементов перечислимого типа используется самый простой вариант вызова:

```
int[] data = new int[500];
int[] results = new int[500];
// Последовательный цикл
for(int i=0; i<data.Length; i++)
     data[i] = ..
// Параллельный цикл
Parallel.For(0, data.Length, i =>
{
     results[i] = SomeWork(data[i]);
});
Console.WriteLine("Total sum: {0}", results.Sum());
```

Аргументами являются границы индекса `i`, который используется в делегате обработки элементов массива. Как и в последовательном цикле, конечный индекс не участвует в обработке. Делегат `Action<int>` вызывается на каждой итерации и в качестве аргумента принимает значения индекса целочисленного типа.

Метод `Parallel.ForEach` позволяет параллельно обрабатывать элементы перечислимого типа.

```
List<string> words = new List<string> {"first",
          "second", "third", "four", "five" };
Parallel.ForEach(words, s => Console.WriteLine(s));
```

Метод `Parallel.ForEach` позволяет создавать произвольные итераторы в отличие от `Parallel.For`.

```
// Используем произвольный итератор: от 0 до 500 с шагом 10
Parallel.ForEach(SteppedIterator(0, 500, 10), index =>
     Console.WriteLine("Index: {0}", index));

double[] dblData = new double[500];
// Инициализируем данные типа double
Parallel.ForEach(dblData, InitData);
// Вычисляем квадратный корень для каждого элемента массива
Parallel.ForEach(dblData, Math.Sqrt);
```

Параметры цикла

Параллельные циклы поддерживают возможность указания токена отмены и максимальную степень параллелизма:

```
Parallel.For(0, 1000,
               new ParallelOptions()
               { maxDegreeOfParallelism = 4,
                    cancellationToken = ctoken},
               i => SomeFunc(i));
```

Досрочный выход из цикла

В последовательных циклах часто используется досрочный выход из цикла с помощью оператора `break`. Досрочный выход используется в двух типовых случаях:

- поиск единственного решения, которое возникает на заранее неизвестной итерации, при этом результаты выполнения предыдущих итераций не представляют интереса;
- обработка всех итераций до итерации, на которой выполняется определенное условие; интерес представляют и все предшествующие итерации.

При последовательном выполнении оба случая можно реализовать с помощью одного оператора `break`. При параллельном выполнении используются два разных метода `Stop` и `Break` объекта `ParallelLoopState`, который является аргументом метода обработчика.

```
var bag1 = new ConcurrentBag<int>();
Parallel.For(0, 1000, (int i, ParallelLoopState pState) =>
     {
          if(i == 50)
               pState.Break();
          else
          {
               Thread.Sleep(10);
               bag1.Add(i);
          }
});
```

```
var bag2 = new ConcurrentBag<int>();
Parallel.For(0, 10000, (i, state) =>
     {
          if (i == 50)
               state.Stop();
          else
          {
               Thread.Sleep(10);
               bag2.Add(i);
          }

     });
Console.WriteLine("Break 50, Smaller: {0}, bigger: {1}",
          bag1.Where(i => i < 50).Count(),
          bag1.Where(i => i > 50).Count());

Console.WriteLine("Stop  50, Smaller: {0}, bigger: {1}",
          bag2.Where(i => i < 50).Count(),
          bag2.Where(i => i > 50).Count());
```

Вывод:

```
Break 50, Smaller: 50, Bigger: 20
Stop  50, Smaller: 38, Bigger: 24
```

Метод `Break` вызван на 50-итерации. Все итерации с меньшими номерами гарантировано будут выполнены, даже если они еще не начались выполняться. Итерации с большими номерами отменяются, если они еще не начались выполняться. Метод `Stop` отменяет все итерации, которые еще не начались выполняться.

Разделение данных

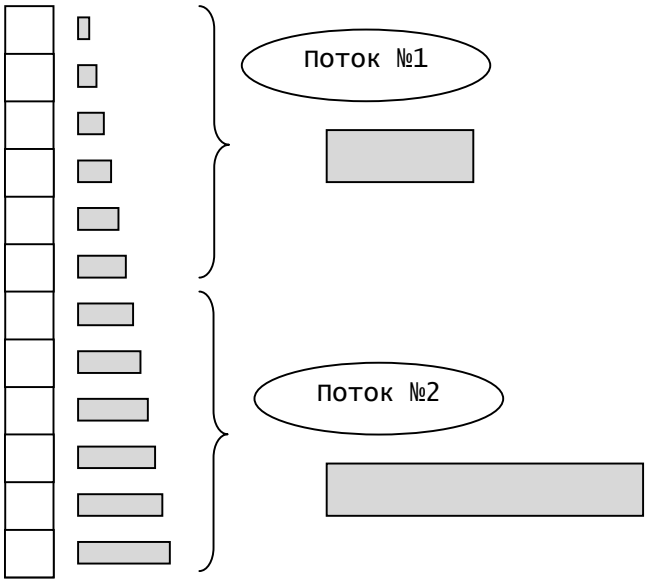
Распределение итераций по рабочим потокам осуществляется либо по равным диапазонам индекса (`range-partitioning`), либо по блокам (`chunk-partitioning`). По умолчанию осуществляется разделение по диапазону (статическая декомпозиция). Планировщик до начала выполнения цикла разделяет итерации по свободным рабочим потокам. В процессе обработки цикла нет необходимости в синхронизации доступа.

```
     Parallel.For(0, 100, i =>
          Console.WriteLine("Iteration: {0}, task: {1},
               thread: {2}", i, Task.TaskId,
                         Thread.ManagedThreadId));
```

В результате получаем, что первые 50 итераций выполнялись одним рабочим потоком, следующие 50 итераций выполнялись вторым рабочим потоком. Статическая

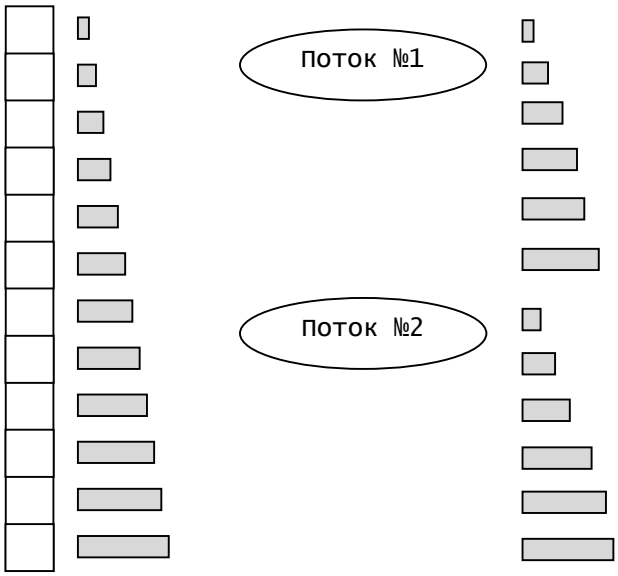
декомпозиция является эффективной при относительно одинаковой вычислительной нагрузке на каждой итерации.

Часто обработка i-элемента зависит от номера индекса. Разная вычислительная «нагрузка» при обработке элементов приводит к несбалансированности параллельной обработки – некоторые потоки быстро выполнят свою работу, а какие-то потоки будут продолжать работать. При статической декомпозиции освободившиеся потоки не помогают загруженным, но могут использоваться планировщиком для обработки других задач.



При блочной (динамической) декомпозиции распределение элементов (итераций цикла) по рабочим потокам осуществляется динамически на протяжении всей обработки цикла. Блочная декомпозиция приводит к более сбалансированному разбиению, но требует затрат на синхронизацию доступа к элементам структуры. Для выполнения блочной декомпозиции необходимо использовать объект `Partitioner` пространства `System.Collection.Concurrent`:

```
// Список элементов
List<string> list = ..
Parallel.ForEach(Partitioner.Create(list, true), s => {
     // Обработка элементов списка
     });
```



Вычисление агрегированных значений

При обработке итераций цикла можно обращаться к разделяемым переменным, например, при вычислении агрегированных величин: суммы, среднего значения и т.п., но необходимо использовать средства синхронизации (блокировки, атомарные операторы).

```
Parallel.For(0, N, i =>
     {
          double d = Compute(ar[i]);
          Interlocked.Add(ref totalSum, d);
     });
```

Если несколько потоков, участвующих в обработке цикла, будут осуществлять обновление общей переменной, то возникает проблема гонки данных. Для предотвращения искажений, используется атомарный оператор добавления `Interlocked.Add`. Тем не менее, синхронизация всех итераций является избыточной. Те итерации, которые обрабатываются одним рабочим потоком, будут выполняться строго последовательно, и поэтому между ними нет конфликта доступа к общей переменной.

Класс `Parallel` предоставляет перегрузку `Parallel.For`, которая позволяет использовать частные переменные – общие для всех итераций, выполняющихся в одном рабочем потоке.

```
// Вычисление арифметического среднего
int [] ar = new int[] {3, 2, 7, 1};
Parallel.For(
     0, N, // индексы массива
     () => 0.0,     // инициализация частных переменных
     (i, state, partial) =>
          {// нет необходимости в блокировке оператора
```

```
                partial += ar[i];
            },
        // конечная редукция частных переменных
        partial =>Interlocked.Add(ref sum, partial))
);
avgValue = sum / N;
```

В этом примере перегрузка `Parallel.For` содержит следующие параметры. Первые два аргумента задают диапазон итераций – от 0 до N. Третий параметр определяет делегат инициализации. Этот делегат вызывается один раз в каждом потоке, участвующем в обработке итераций. Четвертый аргумент задает делегат обработки, который вызывается на каждой итерации. Делегат обработки принимает три аргумента: индекс элемента, объект `ParallelLoopState` и локальная переменная, общая для всех итераций, выполняющихся в одном потоке. Последний параметр задает делегат финальной обработки. Таким образом, вычисление суммы распараллеливается с помощью частных сумм в каждом потоке и агрегированием частных сумм в конце обработки. Синхронизация необходима только в финальном делегате, так как здесь возможна параллельная работа с общей переменной `sum`.

Пакетная обработка итераций

Метод `Parallel.ForEach` обладает еще одной полезной перегрузкой, позволяющей вызывать обработчик не на каждой итерации, а только по одному разу для каждого рабочего потока. В случае большего числа итераций, во много раз превосходящее число рабочих потоков, «пакетная» обработка цикла может увеличить быстродействие. Ниже приведен вызов метода Parallel.ForEach с пакетной обработкой.

```
Parallel.ForEach(
     //
     Partitioner.Create(0, N),
     // Начальная инициализация
     () => 0.0,
     // Обработчик цикла
     (range, state, partial) =>
     {
          for(int i=range.Item1; i< range.Item2; i++)
               partial += ar[i];
          return partial;
     },
     // финальный этап
     partial =>Interlocked.Add(ref sum, partial)
);
```

В качестве первого параметра выступает разделитель, который создается для обрабатываемой структуры данных. В примере используется перегрузка разделителя, создающая ряд целых чисел от 0 до N-1. Второй параметр определяет делегат, который вызывается при инициализации обработки на каждом потоке. В примере делегат обнуляет

значение локальной переменной потока. Третий параметр задает делегат, который вызывается в каждом потоке один раз для обработки всех элементов данного потока. Делегат обработки содержит три аргумента: первый аргумент `range` – предоставляет доступ к граничным элементам, обрабатываемым в данном потоке; второй аргумент `state` типа `ParallelLoopState` предоставляет возможности по досрочному выходу из цикла; третий аргумент `partial` представляет собой локальную переменную потока. Делегат обработки возвращает значение типа локальной переменной. Возвращаемые значения делегатов обработки используются как входные аргументы финального делегата. В рассматриваемом примере каждый поток вычисляет свои локальные суммы. Финальный делегат вычисляет итоговую сумму с помощью атомарного оператора `Interlocked.Add`. Применение средств синхронизации в финальном обработчике связано с возможной параллельностью вызовов. Таким образом, в каждом потоке, участвующем в обработке цикла, последовательно вызывается инициализирующий делегат, делегат обработки и финальный делегат. Каждый делегат обработки работает со своей локальной переменной, поэтому синхронизация при обработке элементов не требуется.