

Параллелизм данных

Класс Parallel

Поддерживает набор методов, которые позволяют выполнять итерации по коллекции данных в параллельном режиме. Основные методы:

- **Parallel.Invoke()** – выполняет параллельно массив делегатов;
- **Parallel.For()** – параллельный эквивалент цикла for;
- **Parallel.ForEach()** – параллельный эквивалент цикла foreach.

Методы блокируют управление до окончания выполнения всех действий. При возникновении необработанного исключения, в каком-то из потоков, оставшиеся рабочие потоки прекращают выполнение, и вызывается исключение **AggregationException**.

Метод `Parallel.Invoke`

Позволяет выполнять один или несколько методов, указываемых в виде его аргументов.

Нет необходимости использовать метод `Wait()`, т.к. `Invoke()` сначала иницииирует выполнение, а затем ожидает завершения всех передаваемых ему методов.

Простая версия метода:

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

Перегруженная версия метода `Parallel.Invoke()`, которая принимает *объект* класса `ParallelOptions`:

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

где `op` – объект класса `ParallelOptions`, с помощью которого можно добавлять маркеры (token) отмены, ограничить максимальное количество рабочих потоков или указать свой планировщик задач (custom task scheduler).

Пример. Работа метода `Invoke()`

```
1 using System;
2 using System.Threading.Tasks;
3 class program
4 { static void Hello()
5   { Console.WriteLine(" Hello");
6   }
7   static void World()
8   { Console.WriteLine(" World!");
9   }
10  static void WorldHello()
11  { Console.WriteLine(" World Hello!");
12  }
13  static void Main(string[] args)
14  { Parallel.Invoke(Hello, World, WorldHello);
15  }
16 }
```



input

```
Hello
World!
World Hello!
```

Метод **Parallel.For**

Аналогичен оператору **for**, но все итерации выполняются в отдельных потоках. Порядок выполнения итераций не определен.

Конструкция метода:

Parallel.For (Int32, Int32, Action)

где первый параметр (Int32) – *начальный индекс* (включительно) *цикла*, второй параметр (Int32) – *конечный индекс* (не включительно) *цикла*, третий параметр (Action) – делегат, который вызывается один раз за итерацию.

Возвращаемым типом у метода **Parallel.For()** является структура **ParallelLoopResult**, в которой содержатся сведения о выполненной части цикла.

Пример. Работа метода `Parallel.For()`

```
1 using System;
2 using System.Threading;
3 using System.Threading.Tasks;
4 class Program
5 { static void Main()
6     { Parallel.For(0, 10, i =>
7         { Console.WriteLine("{0}, Задача: {1}, Поток: {2}", i,
8             Task.CurrentId, Thread.CurrentThread.ManagedThreadId);
9             Thread.Sleep(10);
10        });
11    }
12 }
```



input

```
8, Задача: 9, Поток: 8
4, Задача: 5, Поток: 7
6, Задача: 7, Поток: 10
7, Задача: 8, Поток: 11
9, Задача: 10, Поток: 12
1, Задача: 2, Поток: 4
3, Задача: 4, Поток: 6
2, Задача: 3, Поток: 5
5, Задача: 6, Поток: 9
0, Задача: 1, Поток: 1
```

В одной из перегрузок метода `Parallel.For()` есть третий параметр `Action<int, ParallelLoopState>`. С помощью такой перегрузки метода, можно оказывать влияние на выполнение цикла с помощью методов `Break()` и `Stop()` объекта `ParallelLoopState`:

`Stop()` – применяется в алгоритмах поиска, где после нахождения результата выполнять другие итерации не требуется;

`Break()` – используется для передачи циклу информации, что другие итерации после текущей итерации выполнять не требуется.

Пример. Использование объекта

ParallelLoopState

```
1 using System;
2 using System.Threading;
3 using System.Threading.Tasks;
4 class program
5 { static void Main()
6   { Parallel.For(0, 10, (i,state) =>
7     { if (i == 5) state.Break();
8       Console.WriteLine("{0}, Задача: {1}, Поток: {2}",i,Task.CurrentId,
9        Thread.CurrentThread.ManagedThreadId);
10      Thread.Sleep(50);
11    });
12  }
13 }
```



input

```
0, Задача: 1, Поток: 1
3, Задача: 4, Поток: 6
1, Задача: 2, Поток: 4
5, Задача: 6, Поток: 8
2, Задача: 3, Поток: 5
4, Задача: 5, Поток: 7
```

Метод Parallel.ForEach

Позволяет создавать распараллеленный вариант цикла **foreach**.

Одна из простых форм объявления метода:

```
Parallel.ForEach<TSource>(IEnumerable<TSource> source, Action<TSource> body);
```

где **source** – обозначает коллекцию данных, обрабатываемых в цикле, **body** – метод, который будет выполняться на каждом шаге цикла.

Пример. Работа метода `Parallel.ForEach()`

```
1 using System;
2 using System.Threading;
3 using System.Threading.Tasks;
4 class program
5 { static void Main()
6   { int[] data = { 0,1,2,3,4,5,6,7,8,9,10};
7     Parallel.ForEach<int>(data, 1 =>
8       { Console.WriteLine("{0,4}",1);
9       });
10  }
11 }
```



```
0
6
7
8
9
10
1
2
4
5
3
```

Параллельное выполнение цикла методом `Parallel.ForEach()` также как и метод `Parallel.For()` можно остановить, вызвав метод `Break()` для экземпляра объекта `ParallelLoopState`, передаваемого через параметр `body`.

Форма метода:

```
Parallel.ForEach<TSource>(IEnumerable<TSource> source, Action<TSource, ParallelLoopState> body);
```

Пример. Использование объекта

ParallelLoopState

```
1 using System;
2 using System.Threading;
3 using System.Threading.Tasks;
4 class program
5 { static void Main()
6   { char[] data = { 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h' };
7     Parallel.ForEach(data, (l, state) =>
8       { if (l == 'e') state.Break();
9         Console.WriteLine(" "+l);
10      });
11   }
12 }
```



input

b
c
d
e
a

Для того что бы получить *индекс* параллельной версии цикла `foreach` используется следующая форма метода `Parallel.Foreach()`:

```
Parallel.ForEach<TSource>  
(IEnumerable<TSource> source,  
Action<TSource,ParallelLoopState,long> body
```

где `long` – индекс цикла.

Пример. Использование цикла с использованием индексов цикла

```
1 using System;
2 using System.Threading;
3 using System.Threading.Tasks;
4 class program
5 { static void Main()
6   { char[] data = { 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h' };
7     Parallel.ForEach(data, (l, state,i) =>
8       { Console.WriteLine(l+" "+i);
9     });
10  }
11 }
```



input

```
b 1
d 3
e 4
f 5
g 6
h 7
a 0
c 2
```

Синхронизация в параллельных циклах

Одно из распространенных применений циклов – агрегация. Простая операция агрегирования – суммирование ряда значений. При использовании параллельных циклов синхронизация агрегированного значения между потоками становится проблемой. Если два или более потоков одновременно обращаются к общему изменяемому значению, существует вероятность того, что они будут использовать несогласованные значения, приводящие к неверному результату.

Пример. Просуммировать все целые числа от одного до миллиона.

1) цикл foreach;

поскольку цикл выполняется в одном потоке, нет никакого риска возникновения проблем с синхронизацией, поэтому результат всегда будет правильным:

```
using System;
using System.Threading.Tasks;
using System.Linq;
class program {
    static void Main() {
        long total = 0;
        foreach (int value in Enumerable.Range(1, 1000000))
        {    total += value; }
        Console.WriteLine(total);    // total = 500000500000
    }
}
```

2) цикл Parallel.ForEach;

получим ошибки синхронизации.

```
using System;
using System.Threading.Tasks;
using System.Linq;
class program {
    static void Main() {
        long total = 0;
        Parallel.ForEach(Enumerable.Range(1, 1000000), value =>
        { total += value; });
        Console.WriteLine(total);    // total = 49984049439
    }
}
```

Несколько потоков выполнения считывали и обновляли общую переменную, она несколько раз оставалась в несогласованных состояниях.

Результаты каждый раз разные.

Синхронизация с помощью блокировки

Самый простой способ добавить синхронизацию в цикл – создать блокировку,

```
using System;
using System.Threading.Tasks;
using System.Linq;
class program {
    static void Main() {
        object sync = new object();
        long total = 0;
        Parallel.ForEach(Enumerable.Range(1, 1000000), value =>
        { lock (sync)
          { total += value; }
        });
        Console.WriteLine(total);    // total = 500000500000
    }
}
```

Или разложить данные на разделы, которые могут быть индивидуально суммированы, а затем объединить эти результаты в конце процесса. Тогда во время итераций цикла блокировка не требуется.

```
using System;
using System.Threading.Tasks;
using System.Linq;
class program {
    static void Main() {
        object sync = new object();
        long total = 0;
        Parallel.ForEach(Enumerable.Range(1, 1000000), () => 0L,
            (value, pls, localTotal) =>
            { return localTotal += value; },
            localTotal =>
            { lock (sync)
                { total += localTotal; }
            });
        Console.WriteLine(total); // total = 500000500000
    }
}
```

3) цикл Parallel.For;

```
using System;
using System.Threading.Tasks;
using System.Linq;
class program {
    static void Main() {
        object sync = new object();
        long total = 0;
        Parallel.For(1, 1000001, () => 0L,
            (value, pls, localTotal) =>
            { return localTotal += value; },
            localTotal =>
            { lock (sync)
                { total += localTotal; }
            });
        Console.WriteLine(total); // total = 500000500000
    }
}
```

Исключения и параллельные циклы

При работе с параллельными циклами For или ForEach обработка исключений несколько усложняется. Когда исключение создается в одном потоке выполнения, вполне вероятно, что существуют другие итерации цикла, выполняющиеся параллельно. Они не могут быть просто прекращены, поскольку это может привести к несоответствиям. Чтобы предотвратить такие ошибки данных, итерации цикла, которые уже были запланированы в других потоках, будут продолжены. Это похоже на вызов `ParallelLoopState.Метод Stop` для завершения параллельного цикла.

Класс **AggregateException** является подклассом исключения, поэтому предоставляет все стандартные функции исключения. Кроме того, он имеет свойство, которое содержит коллекцию внутренних исключений. При создании параллельного цикла все встречающиеся исключения включаются в свойство, гарантируя, что никакие детали исключения не будут потеряны.

Исключение **AggregateException** будет вызвано даже в случае, если во время обработки цикла возникнет только одно исключение. Другие исключения могут быть созданы только в том случае, если существует проблема с самой командой цикла, например делегат действия, определяющий тело цикла, является нулевым.

Пример.

```
using System;
using System.Threading.Tasks;
using System.Linq;
class program {
    static void Main() {
        try
        { Parallel.For(-10, 10, i =>
            { Console.WriteLine("100/{0}={1}", i, 100 / i);
            });
        }
        catch (AggregateException ex)
        { Console.WriteLine(ex.Message);
        }
    }
}
```


100/-10=-10

100/-8=-12

100/-6=-16

100/-4=-25

100/-2=-50

One or more errors occurred. (Attempted to divide by zero.)

100/-8=-12

100/-2=-50

100/-4=-25

100/-6=-16

100/-10=-10

One or more errors occurred. (Attempted to divide by zero.)

100/-8=-12

100/-7=-14

100/-6=-16

100/-5=-20

100/-4=-25

100/-3=-33

100/-2=-50

100/-1=-100

100/-10=-10

One or more errors occurred. (Attempted to divide by zero.)

Чтобы показать сообщение непосредственно перед исключением, добавить в цикле команду

```
if (i == 0) Console.WriteLine("Деление на ноль.");
```

```
100/-8=-12
100/-7=-14
100/-5=-20
100/-3=-33
100/-2=-50
100/-1=-100
Деление на ноль.
100/-6=-16
100/-4=-25
100/-10=-10
One or more errors occurred. (Attempted to divide by zero.)
```