Платформа параллельных вычислений

Parallel Framework, PFX
Платформа параллельных вычислений

- это набор типов и технологий, являющийся частью платформы .NET.
- PFX предназначена для повышения производительности разработчиков за счёт средств, упрощающих добавление параллелизма в приложения.

- обеспечивает три уровня организации параллелизма:
  - Параллелизм на уровне задач. Библиотека параллельных задач (Task Parallel Library, TPL).
    System.Threading.Tasks и System.Threading
  - Параллелизм при императивной обработке данных
  - Параллелизм при декларативной обработке данных реализуется при помощи параллельного интегрированного языка запросов (PLINQ).

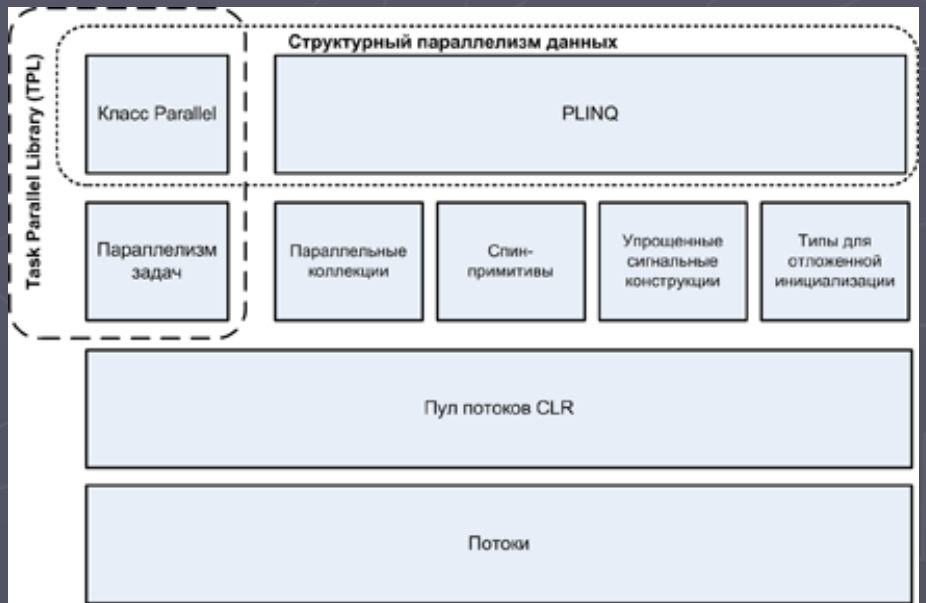
Библиотека параллельных задач TPL (Task Parallel Library)

позволяет распараллелить задачи и выполнять их сразу на нескольких процессорах (для создания многопоточных приложений)

Задача (task) – абстракция более высокого уровня чем поток

using System.Threading.Tasks

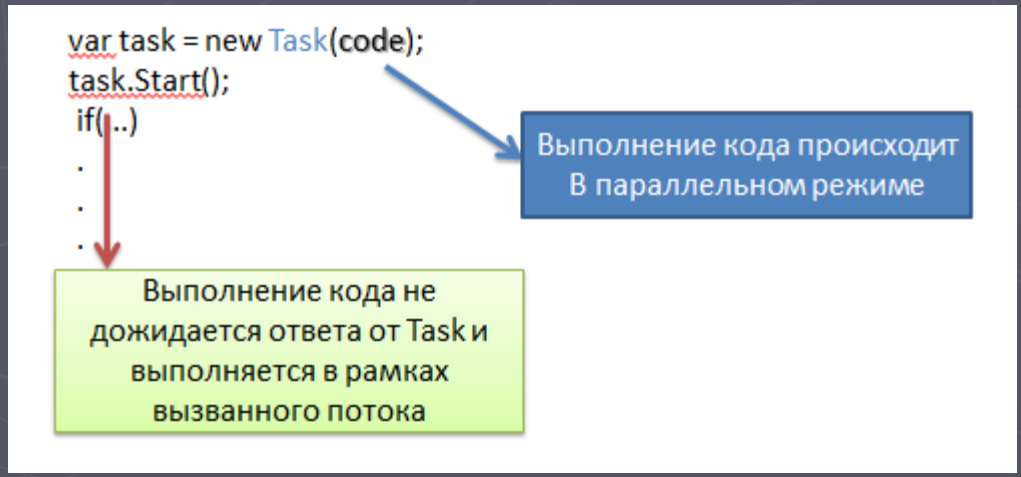
Планировщик библиотеки выполняет диспетчеризацию задач, а также предоставляет единообразный механизм отмены задач и обработки исключительных ситуаций



Класс Task

- описывает отдельную продолжительную операцию, которая запускается асинхронно в одном из потоков из пула потоков (можно запускать синхронно в текущем потоке) – подобна потокам, но абстракция более высокого уровня

- Представлена .Net 4.0
- Среда WinRT



Элементы класса Task<T>

AsyncState	Объект, заданный при создании задачи (аргумент Action<object>)
ConfigureAwait()	Настраивает объект ожидания, используемый для текущей задачи
ContinueWith()	Используются для указания метода, выполняемого после завершения текущей задачи
CreationOptions	Опции, указанные при создании задачи (тип TaskCreationOptions)
CurrentId	Статическое свойство типа int?, которое возвращает целочисленный идентификатор текущей задачи
Delay()	Статический метод; позволяет создать задачу с указанной задержкой старта
Dispose()	Освобождение ресурсов, связанных с задачей
Exception	Возвращает объект типа AggregateException, который соответствует исключению, прервавшему выполнение задачи
Factory	Доступ к фабрике, содержащей методы создания Task и Task<T>
GetAwaiter()	Получает объект ожидания для текущей задачи
Id	Целочисленный идентификатор задачи

Элементы класса Task<T>

IsCanceled	Булево свойство, указывающее, была ли задача отменена
IsCompleted	Свойство равно true, если задача успешно завершилась
IsFaulted	Свойство равно true, если задача сгенерировала исключение
Run()	Статический метод; выполняет создание и запуск задачи
RunSynchronously()	Запуск задачи синхронно
Start()	Запуск задачи асинхронно
Status	Возвращает текущий статус задачи (объект типа TaskStatus)
Wait()	Приостанавливает текущий поток до завершения задачи
WaitAll()	Статический метод; приостанавливает текущий поток до завершения всех указанных задач
WaitAny()	Статический метод; приостанавливает текущий поток до завершения любой из указанных задач
WhenAll()	Статический метод; создаёт задачу, которая будет выполнена после выполнения всех указанных задач
WhenAny()	Статический метод; создаёт задачу, которая будет выполнена после выполнения любой из указанных задач

Определение и запуск задачи

- Первый способ создание объекта Task и вызов у него метода Start:

```
Task task = new Task(() => Console.WriteLine("Hello Task!"));
task.Start();
```

- объект Task принимает делегат Action, то есть мы можем передать любое действие, которое соответствует данному делегату

Определение и запуск задачи

- Второй способ заключается в использовании статического метода Task.Factory.StartNew()

```
Task task = Task.Factory.StartNew(() => Console.WriteLine("Hello Task!"));
```

- принимает делегат Action, который указывает, какое действие будет выполняться.
- При этом этот метод сразу же запускает задачу:

Определение и запуск задачи

- Третий способ определения и запуска задач представляет использование статического метода Task.Run():

```
Task task = Task.Run(() => Console.WriteLine("Hello Task!"));
```

- может принимать делегат Action - выполняемое действие и возвращает объект Task.

Создание задачи

аргумент типа Action – метод, выполняемый в задаче

```
int i = 10;
Task task1 = new Task(() =>
                    { i++; Console.WriteLine("Task 1 finished"); });
task1.Start();
```

запускает задачу, вернее, помещает её в очередь запуска планировщика задач - асинхронный запуск

```
Task task2 = Task.Factory.StartNew(() =>
                    { ++i; Console.WriteLine("Task 2 finished"); });


Task task3 = Task.Run(() =>
                    { ++i; Console.WriteLine("Task 3 finished"); });
```

```
Console.WriteLine(i);
Console.WriteLine("Main finished");
```

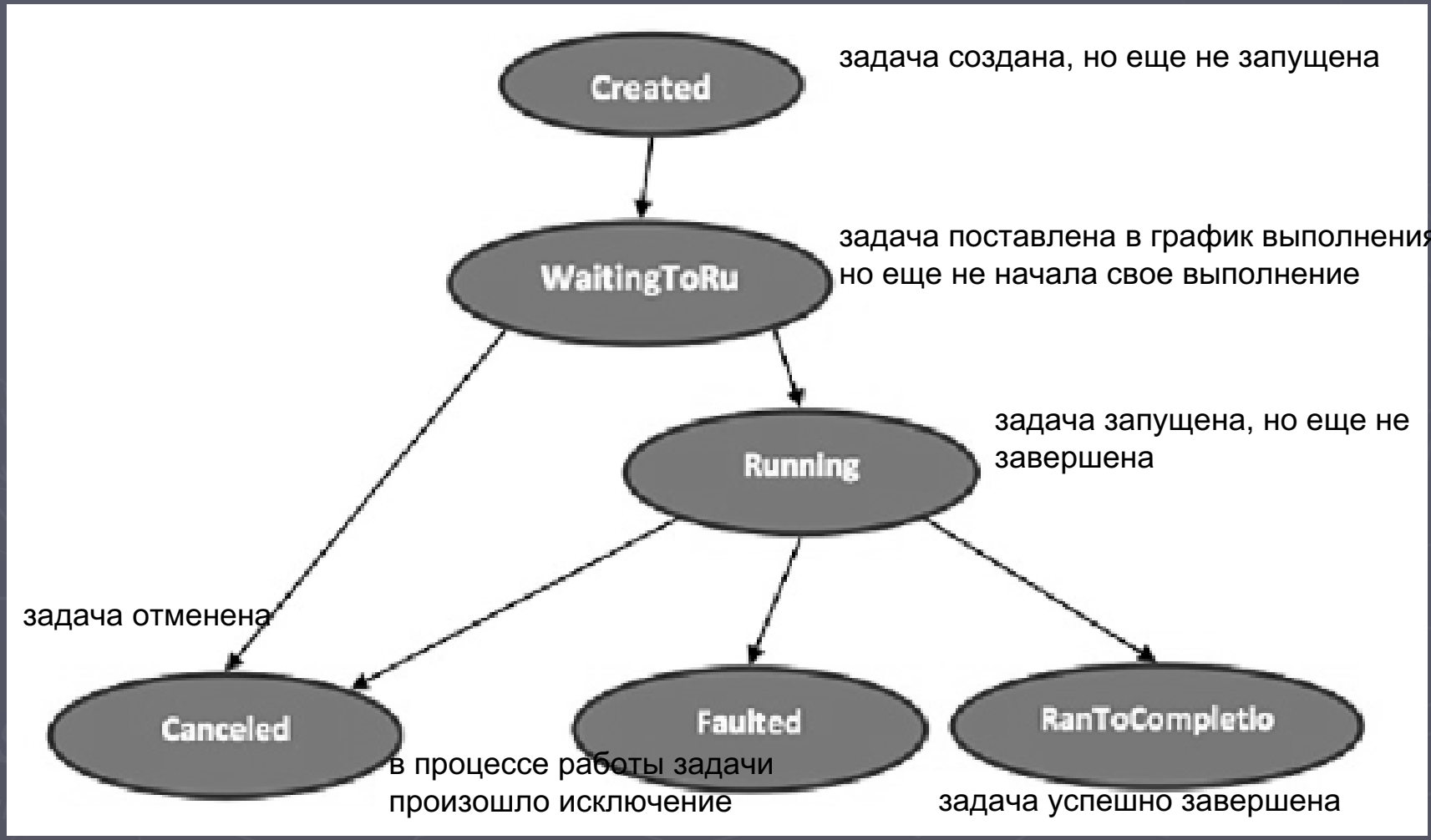
```
10
Main finished
Task 2 finished
Task 3 finished
Task 1 finished
```

Статус задачи. Status

перечисление System.Threading.Tasks.TaskStatus, которое имеет следующие значения:

- Canceled: задача отменена
- Created: задача создана, но еще не запущена
- Faulted: в процессе работы задачи произошло исключение
- RanToCompletion: задача успешно завершена
- Running: задача запущена, но еще не завершена
- WaitingForActivation: задача ожидает активации и постановки в график выполнения
- WaitingForChildrenToComplete: задача завершена и теперь ожидает завершения прикрепленных к ней дочерних задач
- WaitingToRun: задача поставлена в график выполнения, но еще не начала свое выполнение

Состояния Task



```
task1 Id: 1
Task1 Starts
task1 is Completed: False
task1 Status: Running
Task1 Ends
```

```
Task task1 = new Task(() =>
{
    Console.WriteLine($"Task{Task.CurrentId} Starts");
    Thread.Sleep(1000);
    Console.WriteLine($"Task{Task.CurrentId} Ends");
 });
task1.Start(); //запускаем задачу

// получаем информацию о задаче
Console.WriteLine($"task1 Id: {task1.Id}");
Console.WriteLine($"task1 is Completed: {task1.IsCompleted}");
Console.WriteLine($"task1 Status: {task1.Status}");

task1.Wait(); // ожидаем завершения задачи
```

Ожидание завершения задачи. Wait()

```
Task task1 = new Task(() => Console.WriteLine("Task1 is executed"));
task1.Start();

Task task2 = Task.Factory.StartNew(() => Console.WriteLine("Task2 is executed"));

Task task3 = Task.Run(() => Console.WriteLine("Task3 is executed"));

task1.Wait();    // ожидаем завершения задачи task1
task2.Wait();    // ожидаем завершения задачи task2
task3.Wait();    // ожидаем завершения задачи task3
```

- задачи не выполняются последовательно.
- Первая запущенная задача может завершить свое выполнение после последней задачи

```
int i = 10;
Task task1 = new Task(() =>
                    { i++; Console.WriteLine("Task 1 finished"); });
task1.Start();


Task task2 = Task.Factory.StartNew(() =>
                    { ++i; Console.WriteLine("Task 2 finished"); });


Task task3 = Task.Run(() =>
                    { ++i; Console.WriteLine("Task 3 finished"); });
```

```
Task.WaitAll(task1, task2,task3);
```

```
Console.WriteLine(i);
Console.WriteLine("Main finished");
```

```
Task 2 finished
Task 1 finished
Task 3 finished
13
Main finished
```

Wait(), WaitAll() и WaitAny() останавливают основной поток до завершения задачи (или задач)
task1.Wait(1000);

Wait()

блокирует вызывающий поток, в котором запущена задача, пока эта задача не завершит свое выполнение

```
Console.WriteLine("Main Starts");
// создаем задачу
Task task1 = new Task(() =>
{
    Console.WriteLine("Task Starts");
    Thread.Sleep(1000);      // задержка на 1 секунду - имитация долгой работы
    Console.WriteLine("Task Ends");
 });
task1.Start();  // запускаем задачу
task1.Wait();   // ожидаем выполнения задачи
Console.WriteLine("Main Ends");
```

основной поток остановит свое выполнение и будет ждать завершения задачи.

```
Main Starts
Task Starts
Task Ends
Main Ends
```

Синхронный запуск задачи

- По умолчанию задачи запускаются асинхронно
- RunSynchronously() - можно запускать синхронно

```
Console.WriteLine("Main Starts");
// создаем задачу
Task task1 = new Task(() =>
{
    Console.WriteLine("Task Starts");
    Thread.Sleep(1000);
    Console.WriteLine("Task Ends");
 });
task1.RunSynchronously(); // запускаем задачу синхронно
Console.WriteLine("Main Ends"); // этот вызов ждет завершения задачи task1
```

Вложенные задачи

- Одна задача может запускать другую - вложенную задачу.
- При этом эти задачи выполняются независимо друг от друга.

```
var outer = Task.Factory.StartNew(() =>       // внешняя задача
{
    Console.WriteLine("Outer task starting...");
    var inner = Task.Factory.StartNew(() =>  // вложенная задача
    {
        Console.WriteLine("Inner task starting...");
        Thread.Sleep(2000);
        Console.WriteLine("Inner task finished.");
    });
});
outer.Wait(); // ожидаем выполнения внешней задачи
Console.WriteLine("End of Main");
```

```
Outer task starting...
End of Main
```

Ожидание выполнения внешней задачи никак не влияет на вложенную.
Она может даже не успеть начать выполняться.

TaskCreationOptions.AttachedToParent

```
var outer = Task.Factory.StartNew(() =>       // внешняя задача
{
    Console.WriteLine("Outer task starting...");
    var inner = Task.Factory.StartNew(() =>  // вложенная задача
    {
        Console.WriteLine("Inner task starting...");
        Thread.Sleep(2000);
        Console.WriteLine("Inner task finished.");
    }, TaskCreationOptions.AttachedToParent);
});
outer.Wait(); // ожидаем выполнения внешней задачи
Console.WriteLine("End of Main");
```

внешняя задача завершится только когда завершатся все прикрепленные к ней вложенные задачи.

```
Outer task starting...
Inner task starting...
Inner task finished.
End of Main
```

Массив задач

Можно определить все задачи в массиве непосредственно через объект Task:

```
Task[] tasks1 = new Task[3]
{
    new Task(() => Console.WriteLine("First Task")),
    new Task(() => Console.WriteLine("Second Task")),
    new Task(() => Console.WriteLine("Third Task"))
};
// запуск задач в массиве
foreach (var t in tasks1)
    t.Start();
```

можно использовать методы Task.Factory.StartNew или Task.Run и сразу запускать все задачи:

```
Task[] tasks2 = new Task[3];
int j = 1;
for (int i = 0; i < tasks2.Length; i++)
    tasks2[i] = Task.Factory.StartNew(() => Console.WriteLine($"Task {j++}"));
```

опять же можем столкнуться с тем, что все задачи из массива могут завершиться после того, как отработает метод Main, в котором запускаются эти задачи

```
Task[] tasks = new Task[3];
for(var i = 0; i < tasks.Length; i++)
{
    tasks[i] = new Task(() =>
    {
        Thread.Sleep(1000); // эмуляция долгой работы
        Console.WriteLine($"Task{i} finished");
    });
    tasks[i].Start();    // запускаем задачу
}
Console.WriteLine("Завершение метода Main");
```

```
Завершение метода Main
```

Task.WaitAll(tasks)

Если необходимо завершить выполнение программы или вообще выполнять некоторый код лишь после того, как все задачи из массива завершатся

```
Task[] tasks = new Task[3];
for(var i = 0; i < tasks.Length; i++)
{
    tasks[i] = new Task(() =>
    {
        Thread.Sleep(1000); // эмуляция долгой работы
        Console.WriteLine($"Task{i} finished");
    });
    tasks[i].Start();   // запускаем задачу
}
Console.WriteLine("Завершение метода Main");

Task.WaitAll(tasks); // ожидаем завершения всех задач
```

порядок выполнения самих задач в массиве также недетерминирован

сначала завершатся все задачи, и лишь только потом будет выполняться последующий код из метода Main

Task.WaitAny(tasks) - ждет, пока завершится хотя бы одна из массива задач.

- Задачи могут быть вложенные
- Запуск задачи

```
 int i = 10;
Task task3 = Task.Run(() =>
                    { ++i; Console.WriteLine("Task 3 finished"); });

// task3.Start(); // System.InvalidOperationException:
//Start нельзя вызывать для уже запущенной задачи.
```

- Синхронный запуск

```
 Action<object> method = x =>
                     { Thread.Sleep(1000);
                      Console.WriteLine(x.ToString()); };

var task4 = new Task(method, TaskCreationOptions.LongRunning);
task4.RunSynchronously();
```

задаёт вид задачи (например, LongRunning – долгая задача)

выполняет задачу синхронно

TaskCreationOptions:

AttachedToParent - Указывает, что задача присоединена к родительской задаче в иерархии задач.

DenyChildAttach – не дает дочерним задачам присоединиться к родительской.

LongRunning – позволяет планировщику задач превысить допустимое для решения задачи количество потоков для ускорения выполнения.

PreferFairness - задачи, запланированные ранее, будут выполняться ранее, а более поздние — позднее, но не обязательно.

Возврат результата

- Task<TResult> - описывает задачу, возвращающую значение типа Tresult
- принимают аргументы типа
  - Func<TResult>
  - Func<object, TResult> (опционально – аргументы типа CancellationToken и TaskCreationOptions)

```
int n1 =4, n2 = 5;
Task<int> sumTask = new Task<int>(() => Sum(n1, n2));
sumTask.Start();

int result = sumTask.Result;
Console.WriteLine($"{n1} + {n2} = {result}"); // 4 + 5 = 9


int Sum(int a, int b) => a + b;
```

не надо его приводить к типу int, оно уже само по себе будет представлять число

```
Func<int> func = () =>
                        { Thread.Sleep(1000);
                            return ++i; };
            Task<int> task = new Task<int>(func);
            Console.WriteLine(task.Status);          // Created
            task.Start();
            Console.WriteLine(task.Status);          // WaitingToRun
            task.Wait();
            Console.WriteLine(task.Result);          // 14
            Console.WriteLine("Main finished");
```

```
Created
WaitingToRun
14
Main finished
Для продолжения нажмите любую клавишу
```

```
Task<Person> defaultPersonTask = new Task<Person>(() => new Person("Tom", 37));
defaultPersonTask.Start();

Person defaultPerson = defaultPersonTask.Result;
Console.WriteLine($"{defaultPerson.Name} - {defaultPerson.Age}"); // Tom - 37

record class Person(string Name, int Age);
```

Обработка исключений

- System.AggregateException

```
Task task5 = Task.Run(() => {
                                throw new Exception(); });
            try
            {
                task5.Wait();
            }
            catch (AggregateException ex)
            {
                var message = ex.InnerException.Message;
                Console.WriteLine(message);
            }
```

Отмена выполнения задач

- Структура CancellationToken - токен отмены

```
CancellationTokenSource tokenSource =
                    new CancellationTokenSource();
// используем  токен в двух задачах
new Task(method, tokenSource.Token).Start();
new Task(method, tokenSource.Token).Start();

//  отменяем  задачи
tokenSource.Cancel();
```

Алгоритм отмены задачи

1. Создание объекта CancellationTokenSource, который управляет и посылает уведомление об отмене токену.

1. С помощью свойства CancellationTokenSource.Token получаем собственно токен - объект структуры CancellationToken и передаем его в задачу, которая может быть отменена.

```
CancellationTokenSource cancelTokenSource = new CancellationTokenSource();
CancellationToken token = cancelTokenSource.Token;
```

Для передачи токена в задачу можно применять один из конструкторов класса Task

```
CancellationTokenSource cancelTokenSource = new CancellationTokenSource();
CancellationToken token = cancelTokenSource.Token;
Task task = new Task(() => { выполняемые_действия}, token); //
```

3. Определяем в задаче действия на случай ее отмены.

4. Вызываем метод CancellationTokenSource.Cancel(), который устанавливает для свойства CancellationToken.IsCancellationRequested значение true.

- метод CancellationTokenSource.Cancel() не отменяет задачу, он лишь посылает уведомление об отмене

Каким образом будет происходить выход из задачи, это решает сам разработчик.

5. Класс **CancellationTokenSource** реализует интерфейс IDisposable. И когда работа с объектом CancellationTokenSource завершена, у него следует вызвать метод Dispose для освобождения всех связанных с ним используемых ресурсов.

Отмена задачи с помощью генерации исключения

- генерация исключения OperationCanceledException
- применяется метод ThrowIfCancellationRequested() объекта CancellationToken

исключение возникает только тогда, когда мы останавливаем текущий поток и ожидаем завершения задачи с помощью методов Wait или WaitAll. Если эти методы не используются для ожидания задачи, то для нее просто устанавливается состояние Canceled

Продолжения (**continuation task**)

- сообщает задаче, что после её завершения она должна продолжить делать что-то другое

```
Task task6 = Task.Run(() =>
                    Console.Write("Doing.."));

Task task7 = task6.ContinueWith(t =>
                  Console.Write("continuation"));
```

После того как задача завершается, отказывает или отменяется, задача task7 (продолжение) запускается

```
Main finished
Doing..continuationД
```

```
Task task1 = new Task(() =>
{
    Console.WriteLine($"Id задачи: {Task.CurrentId}");
});

// задача продолжения - task2 выполняется после task1
Task task2 = task1.ContinueWith(PrintTask);

task1.Start();

// ждем окончания второй задачи
task2.Wait();
Console.WriteLine("Конец метода Main");

void PrintTask(Task t)
{
    Console.WriteLine($"Id задачи: {Task.CurrentId}");
    Console.WriteLine($"Id предыдущей задачи: {t.Id}");
    Thread.Sleep(3000);
}
```

принимает делегат Action<Task>. То есть метод PrintTask, который передается в вызов ContinueWith, должен принимать параметр типа Task

после завершения задачи task1 сразу будет вызываться задача task2

Благодаря передачи в метод параметра Task, мы можем получить различные свойства предыдущей задачи

```
Id задачи: 1
Id задачи: 2
Id предыдущей задачи: 1
Конец метода Main
```

можем передавать конкретный результат работы предыдущей задачи

```
Task<int> sumTask = new Task<int>(() => Sum(4, 5));

// задача продолжения
Task printTask = sumTask.ContinueWith(task => PrintResult(task.Result));

sumTask.Start();

// ждем окончания второй задачи
printTask.Wait();
Console.WriteLine("Конец метода Main");

int Sum(int a, int b) => a + b;
void PrintResult(int sum) => Console.WriteLine($"Sum: {sum}");
```

фактически представляет задачу sumTask, из которой извлекается результат

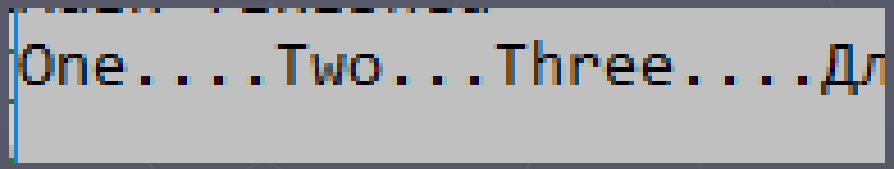
printTask является задачей продолжения, выполняется сразу после sumTask и получает ее результат.

можно построить целую цепочку последовательно выполняющихся задач

```
Task task1 = new Task(() => Console.WriteLine($"Current Task: {Task.CurrentId}"));

// задача продолжения
Task task2 = task1.ContinueWith(t =>
    Console.WriteLine($"Current Task: {Task.CurrentId}  Previous Task: {t.Id}"));

Task task3 = task2.ContinueWith(t =>
    Console.WriteLine($"Current Task: {Task.CurrentId}  Previous Task: {t.Id}"));


Task task4 = task3.ContinueWith(t =>
    Console.WriteLine($"Current Task: {Task.CurrentId}  Previous Task: {t.Id}"));

task1.Start();

task4.Wait();    //  ждем завершения последней задачи
Console.WriteLine("Конец метода Main");
```

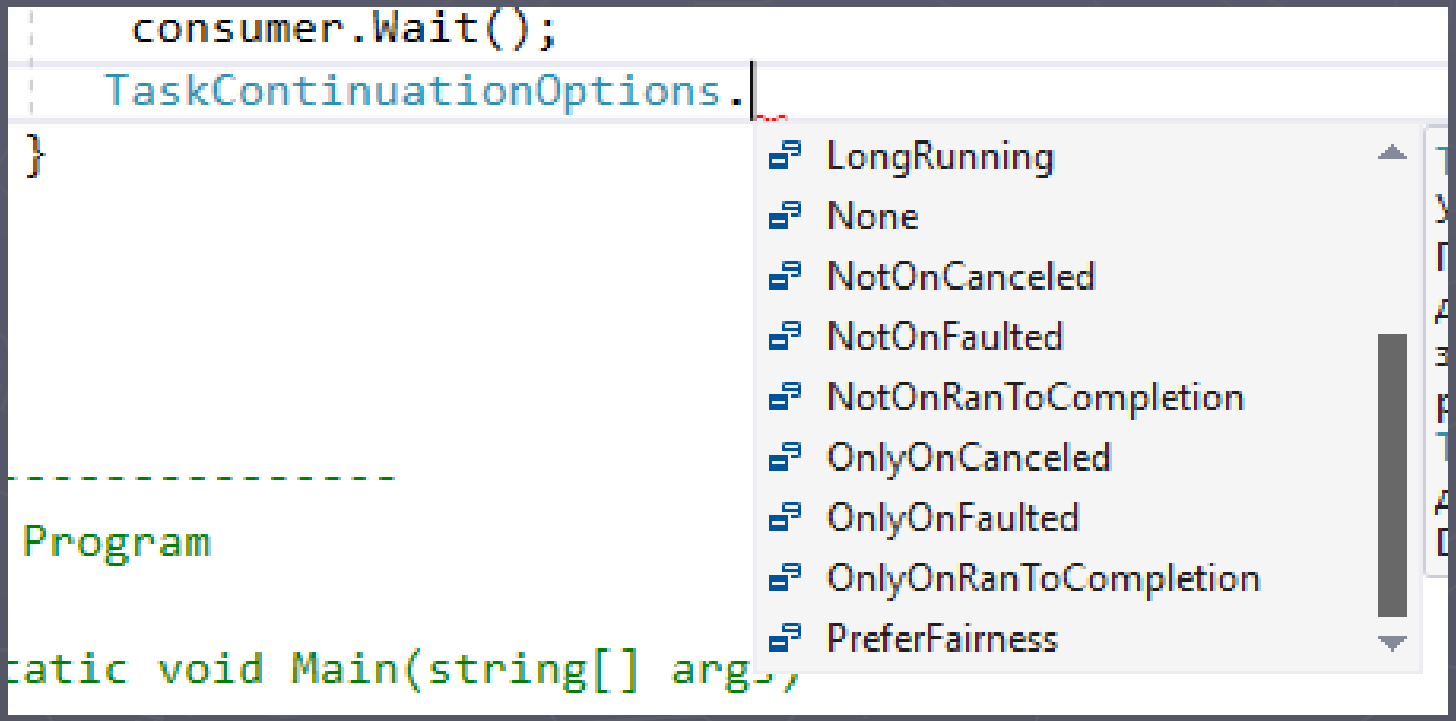
```
Current Task: 1
Current Task: 2  Previous Task: 1
Current Task: 3  Previous Task: 2
Current Task: 4  Previous Task: 3
Конец метода Main
```

```
Task task8 = Task.Run(() => Console.Write("One...."));

Task task9 = Task.Run(() => Console.Write("Two..."));

Task continuation = Task.WhenAll(task8, task9).
                        ContinueWith(t => Console.Write("Three...."));
```



1)планировка на основе завершения множества предшествующих задач

Установка статуса продолжения



2)использовании объекта ожидания

- Объект ожидания – это любой объект, имеющий методы OnCompleted() и GetResult() и свойство IsCompleted.

```
Task<int> what = Task.Run(() => Enumerable.Range(1, 100000)
                              .Count(n=>(n%2==0)));
// получаем объект продолжения
 var awaiter = what.GetAwaiter();

// что делать после окончания предшественника
awaiter.OnCompleted(() =>          {
// получаем результат вычислений предшественника

            int res = awaiter.GetResult();
            Console.WriteLine(res);
        });
```

...50000

делегат, содержащий код продолжения

По умолчанию – в разных потоках

Параллелизм при императивной обработке данных
Класс Parallel

- System.Threading.Tasks.Parallel позволяет распараллеливать циклы и последовательность блоков кода
- For(), ForEach(), Invoke() – шаблоны (на задачах, поддерж искл. и токен отмены)

являются параллельными аналогами циклов for и foreach

могут принимать аргумент типа ParallelOptions для настройки поведения метода

Parallel.For

Parallel.For(int, int, Action<int>)

указывает на метод, который будет выполняться один раз за итерацию:

указание начального и конечного значения счётчика (типа int или long) и тела цикла в виде объекта делегата

```
Parallel.For(1, 10, z=>
                { int r = 1;
                        for (int y = 1; y <= 10; y++)
                    {
                     r *= z;
                     Console.WriteLine(r);
                     }
             });
```

Дополнительные возможности:
Досрочный выход
Пакетная обработка диапазонов
Реализация агрегированных операций

```
4
16
64
256
1024
4096
16384
65536
262144
1048576
2
4
8
16
32
64
128
256
512
1024
6
36
216
1296
7776
46656
279936
1679616
10077696
60466176
```

```
Parallel.For(1, 10, (int z, ParallelLoopState pd) =>
                {
                    Console.WriteLine(z);
                    int r = 1;
                    for (int y = 1; y <= 10; y++)
                    {
                        r *= z;
                    }

                });
            Console.WriteLine("Stop");
```

```
3
4
6
8
9
1
2
7
5
Stop
```

Поддерживается императивность – оператор следующий за вызовом метода будет вызван после завершения всех задач

```
Parallel.For(1, 5, Square);

// вычисляем квадрат числа
void Square(int n)
{
    Console.WriteLine($"Выполняется задача {Task.CurrentId}");
    Console.WriteLine($"Квадрат числа {n} равен {n * n}");
    Thread.Sleep(2000);
}
```

```
Выполняется задача 1
Выполняется задача 2
Квадрат числа 4 равен 16
Выполняется задача 4
Квадрат числа 1 равен 1
Выполняется задача 3
Квадрат числа 3 равен 9
Квадрат числа 2 равен 4
```

Parallel.ForEach

- ParallelLoopResult ForEach<TSource>
- (IEnumerable<TSource> source,Action<TSource> body)

коллекция, делегат, выполняющийся один раз за итерацию для каждого перебираемого элемента коллекции

На выходе метод возвращает структуру ParallelLoopResult, которая содержит информацию о выполнении цикла

```
ParallelLoopResult listFact = Parallel.ForEach<int>
                              (new List<int>() { 1, 3, 5, 8 },
                              Factorial);
```

```
Parallel.ForEach(Directory.GetFiles(path, "*.jpg"),
                                 image => Process(image));
```

- **IsCompleted**: определяет, завершилось ли полное выполнение параллельного цикла
- **LowestBreakIteration**: возвращает индекс, на котором произошло прерывание работы цикла

```
ParallelLoopResult result = Parallel.For(1, 10, Factorial);

if (!result.IsCompleted)
        Console.WriteLine("Выполнение цикла завершено на итерации {0}",
                                        result.LowestBreakIteration);
```

Досрочный выход из цикла

- Stop() - прекращение всех, еще не начавшихся итераций
- Break() – прекращение всех, еще не начавшихся итераций, **за исключением итераций с меньшим значением**

```
Parallel.For(1, 10, (int z, ParallelLoopState pd) =>
                {
                    int r = 1;
                    for (int y = 1; y <= 10; y++)
                    {
                        if (r == 5) pd.Stop();
                        r *= z;
                    }
                });
```

Досрочный выход из цикла

К примеру, применив Stop() и Break() на 5 итерации цикла:

Stop() завершит цикл;

Break() подождет выполнения 1-4 итераций, после чего завершит цикл.

При этом в обоих случаях, начавшиеся, но не завершенные итерации цикла не остановятся.

Parallel.Invoke()

- позволяет распараллелить исполнение блоков операторов – набор задач, которые выполняются в одном потоке

Parallel.Invoke (FuncOne, Func Two…)

Методы, лямбда-выражения, массив Action

```
 Parallel.Invoke(
                  () =>     new
WebClient().DownloadFile("http://www.belstu.by", "start.html"),
                  () => new
WebClient().DownloadFile("http://www.go.by", "go.html"));
```

Их можно запустить одновременно

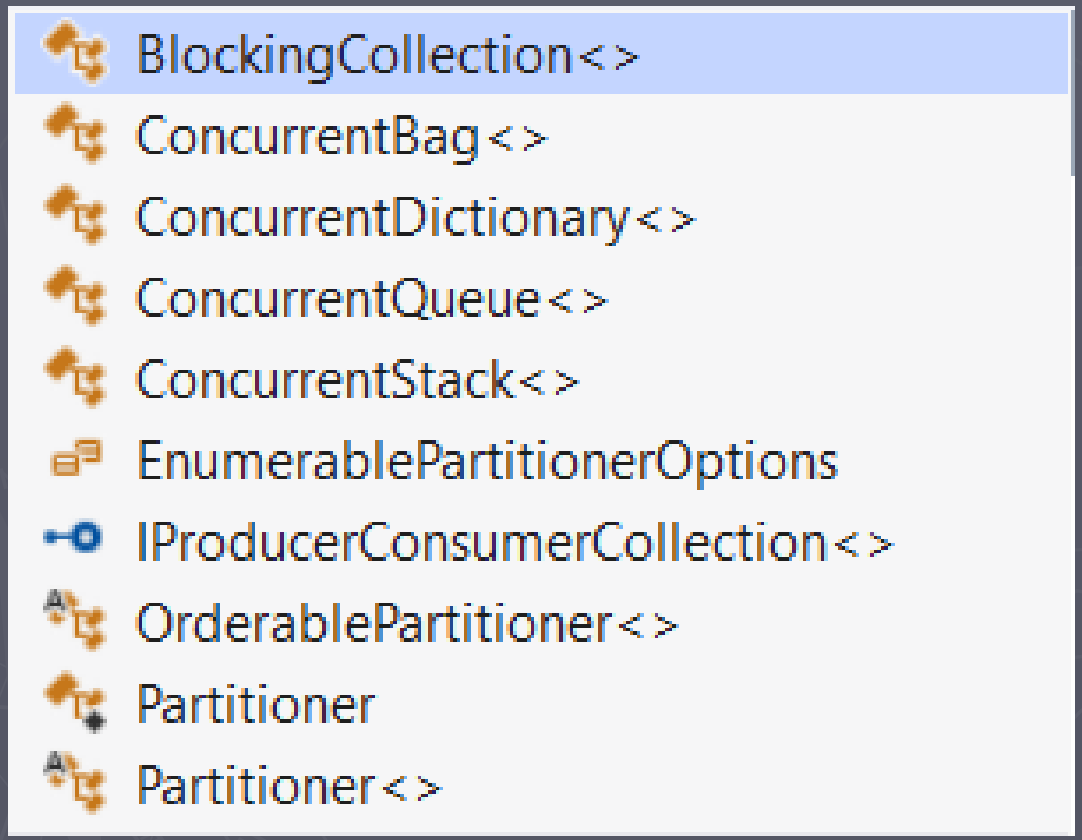
Вызывающий поток должен дождаться завершения всех рабочих элементов

```
// метод Parallel.Invoke выполняет три метода
Parallel.Invoke(
    Print,
    () =>
    {
        Console.WriteLine($"Выполняется задача {Task.CurrentId}");
        Thread.Sleep(3000);
    },
    () => Square(5)
);

void Print()
{
    Console.WriteLine($"Выполняется задача {Task.CurrentId}");
    Thread.Sleep(3000);
}
// вычисляем квадрат числа
void Square(int n)
{
    Console.WriteLine($"Выполняется задача {Task.CurrentId}");
    Thread.Sleep(3000);
    Console.WriteLine($"Результат {n * n}");
}
```

```
Выполняется задача 1
Выполняется задача 3
Выполняется задача 2
Результат 25
```

Коллекции, поддерживающие параллелизм

System.Collections.Concurrent



Коллекции, поддерживающие параллелизм

- System.Collections.Concurrent Класс BlockingCollection<T>
- реализация шаблона «поставщик - потребитель»
- поддерживает параллельное добавление и извлечение элементов из нескольких потоков;
- допускает указание максимальной емкости;
- поддерживает операции вставки и удаления, блокирующиеся при опустошении или заполнении коллекции;
- поддерживает условные операции вставки и удалении, не блокирующиеся или блокирующиеся лишь на определенное время;
- инкапсулирует все типы коллекций, реализующие интерфейс IProducerConsumerCollection<T>;
- поддерживает отмену с помощью токенов отмены;
- поддерживает два вида перечисления с помощью оператора foreach:
- перечисление "только для чтения";
- перечисление, при котором элементы по мере перечисления удаляются.

`Add()`	Добавляет элемент в коллекцию
`AddToAny()`	Статический метод, который добавляет элемент в любую из указанных `BlockingCollection<T>`
`CompleteAdding()`	После вызова этого метода добавление элементов невозможно
`GetConsumingEnumerable()`	Возвращает перечислитель, который перебирает элементы с их одновременным удалением из коллекции
`Take()`	Получает элемент и удаляет его из коллекции. Если коллекция пуста и у коллекции был вызван метод `CompleteAdding()`, генерируется исключение
`TakeFromAny()`	Статический метод, который получает элемент из любой указанной `BlockingCollection<T>`
`TryAdd()`	Пытается добавить элемент в коллекцию, в случае успеха возвращает `true`. Дополнительно может быть задан временной интервал и токен отмены
`TryAddToAny()`	Статический метод, который пытается добавить элемент в любую из указанных коллекций

`TryTake()`	Пытается получить элемент (с удалением из коллекции), в случае успеха возвращает `true`
`TryTakeFromAny()`	Статический метод, который пытается получить элемент из любой указанной `BlockingCollection<T>`
`BoundedCapacity`	Свойство возвращает максимальное число элементов, которое можно добавить в коллекцию без блокировки поставщика (данный параметр может быть задан при вызове конструктора `BlockingCollection<T>`)
`IsAddingCompleted`	Возвращает `true`, если вызывался `CompleteAdding()`
`IsCompleted`	Возвращает `true`, если вызывался `CompleteAdding()` и коллекция пуста

- Для использования BlockingCollection<T>:

- Создайте экземпляр класса, указав при необходимости коллекцию, реализующую интерфейс IProducerConsumerCollection<T> и максимальный размер коллекции.
- Вызовите методы Add или TryAdd для добавления элементов в нижележащую коллекцию.
- Вызовите методы Take или TryTake для удаления (получения) элементов нижележащей коллекции.

```
static int x = 0;
```

поставщик - потребитель

```
BlockingCollection<int> blockcoll = new BlockingCollection<int>();
            for (int producer = 0; producer < 5; producer++)
            {
                Task.Factory.StartNew(() =>
                {    x++;
                    for (int ii = 0; ii < 3; ii++)
                    {
                        x++;
                        Thread.Sleep(100);
                        int id =  x;
                        blockcoll.Add(id);
                        Console.WriteLine("Produser  add " + id);
                    }
                });
            }

            Task consumer = Task.Factory.StartNew(
                    () =>
                    {
                        foreach (var item in blockcoll.GetConsumingEnumerable())
                        {
                            Console.WriteLine(" Reading " +item);
                        }
                    });
            consumer.Wait();
```

автоматически создал экземпляр параллельной очереди

добавление элементов в коллекцию.

возвращает (потенциально) бесконечную последовательность, которая возвращает элементы, когда они становятся доступными.

```
Produser  add 8
Produser  add 8
Produser  add 8
Produser  add 8
Produser  add 12
Produser  add 12
Produser  add 12
Produser  add 15
Produser  add 16
Produser  add 16
Produser  add 18
Produser  add 16
 Reading 8
 Reading 8
 Reading 8
 Reading 8
 Reading 12
 Reading 12
 Reading 12
 Reading 15
 Reading 16
 Reading 16
 Reading 16
 Reading 18
Produser  add 18
 Reading 18
Produser  add 19
 Reading 19
Produser  add 20
 Reading 20
```

Асинхронное программирование

При асинхронном вызове поток выполнения разделяется на две части:

- в одной выполняется метод,
- а в другой – процесс программы.

- Асинхронный вызов методов реализуется средой исполнения при помощи пула потоков

Асинхронные методы, async и await

обладает следующими признаками:

- В заголовке метода используется модификатор async;
- Метод содержит одно или несколько выражений await;
- В качестве возвращаемого типа используется один из следующих
- void
- Task
- Task<T>
- ValueTask<T>

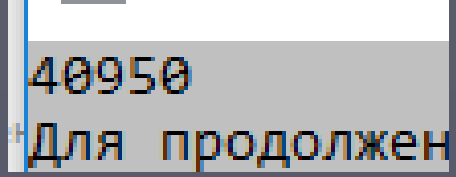
обычно является задачей

```
var результат = await выражение;
оператор(ы);
```

```
await выражение;
 оператор(ы);
```

```
var awaiter = выражение.GetAwaiter();
 awaiter.OnCompleted(()=>
    {
            var результат = awaiter.GetResult();
             оператор(ы);
     }
```

может применяться только внутри метода (или лямбда-выражения) со специальным модификатором async



- методы с модификатором async - называются асинхронные функции

```
static  private async void ReadFromWeb()
        {
            var web = new WebClient();
            var text =
                  await
web.DownloadStringTaskAsync("https://msdn.microsoft.com/ru-ru/");
            Console.WriteLine(text.Length);
        }

    static void Main(string[] args)
        {

            ReadFromWeb();
            Thread.Sleep(3000);
        }
```

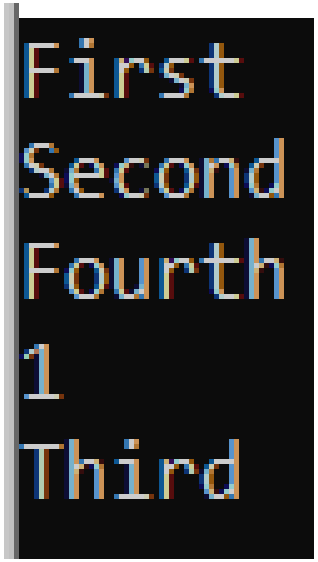
создаёт задачу Task<string> для чтени
сайта и возвращает управление

приостановить выполнение метода до тех пор, пока эта задача не завершится

асинхронная функция, которая не блокирует при вызове основной поток

- В стандартных классах платформы .NET многие методы, выполняющие долгие операции, получили поддержку в виде асинхронных аналогов. Async в названии

```
qwe();
Console.WriteLine("Fourth");
Thread.Sleep(2000);

static async Task<int> www()
{
    Console.WriteLine("Second");
    await Task.Delay(1000);
    return 1;
}
static async void qwe()
{
    Console.WriteLine("First");
    int a = await www();
    Console.WriteLine(a);
    Console.WriteLine("Third");
}
```



await приостанавливает выполнение текущего метода, при этом последовательность выполнения инструкций во внешнем контексте продолжается.

```
qwe();
Console.WriteLine("Fourth");
Thread.Sleep(2000);

static async void www()
{
    Console.WriteLine("Second");
    await Task.Delay(1000);
}
static async void qwe()
{
    Console.WriteLine("First");
    www();
    Console.WriteLine("Third");
}
```

```
First
Second
Third
Fourth
```

Функции, помеченные async, но не имеющие await работают как синхронные

- 1) В сигнатуру метода добавляется async
- 2) Имя метода async, по соглашению, заканчивается суффиксом «Async»
- 3) Тип возврата
  - Task <TResult>, если return
  - Task, если нет оператора return
  - void , если обработчик событий async.
- 4) Обычно метод включает в себя хотя бы одно выражение await, которое отмечает точку, в которой метод не может продолжаться до тех пор, пока ожидаемая асинхронная операция не будет завершена.