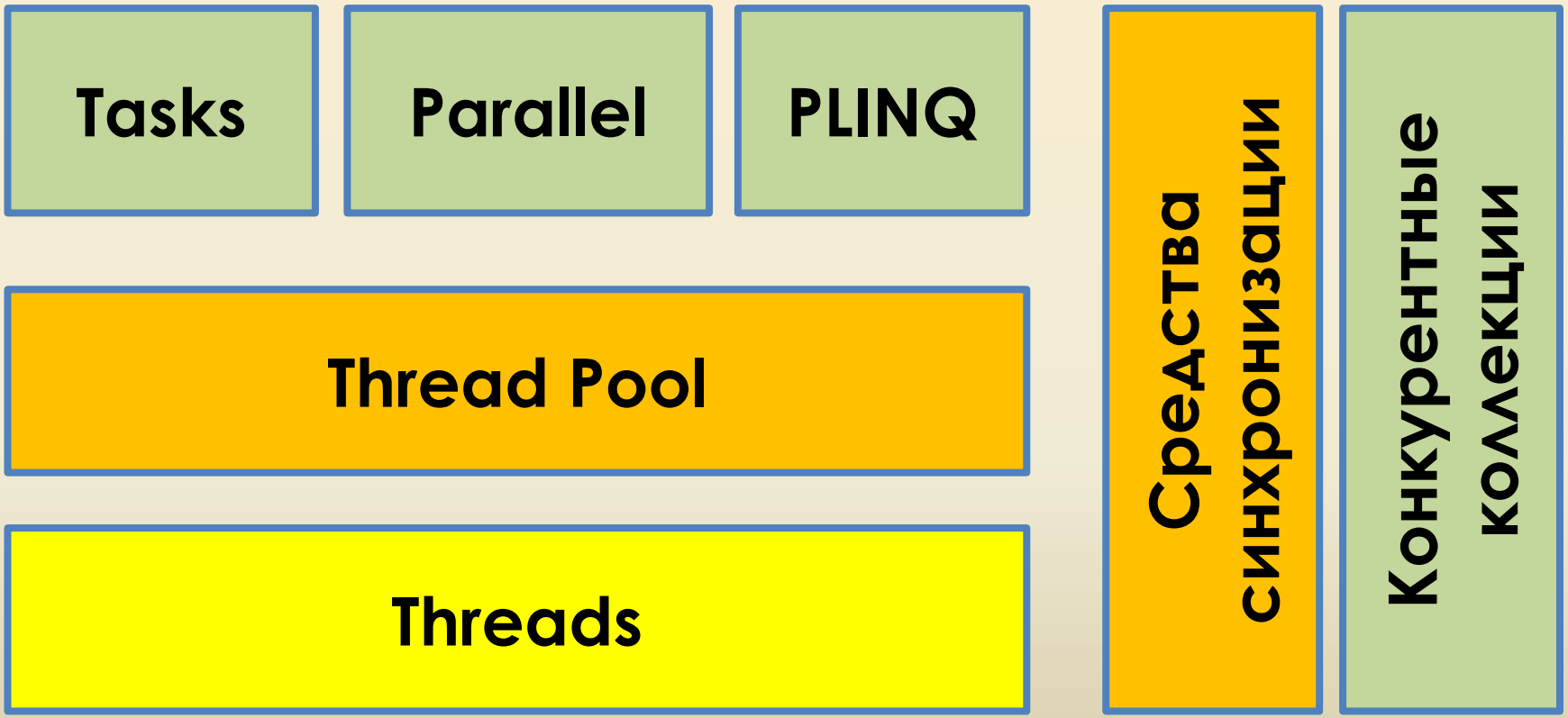
Среда Framework .NET для параллельного программирования



Tasks

- Задачи являются основным строительным блоком библиотеки **Task Parallel Library**
- Задачи представляют собой рабочие элементы, которые могут выполняться параллельно
- В качестве рабочих элементов могут выступать методы, делегаты, лямбда-выражения
- Для выполнения задач используются рабочие потоки пула
- Распределение пользовательских задач по потокам осуществляется планировщиком (**TaskSheduler**)

Основные операции с задачами

```
// Создание задачи
Task t = new Task(SomeWork);

// Запуск задачи
t.Start();

// Ожидание задачи
t.Wait();
```

Создание задачи

Конструкторы Task позволяют инициализировать:

- Функциональный объект типа Action или Action<object>
- Опции TaskCreationOptions
- Токен отмены CancellationToken

Варианты запуска задач

#1

```
var t = new Task(() => Console.WriteLine(“”));
t.Start();
```

#2

```
Task.Run(() => Console.WriteLine(“task ran”));
```

#3

```
Tast tAnother = Task.Factory.StartNew(SomeWork);
```

Ожидание задач

- Ожидание отдельной задачи

  OneTask.Wait();

- Ожидание нескольких задач

  Task.WaitAll(Task1, Task2, Task3);

  Task.WaitAny(Task5, Task6);

Результат задачи. Task<T>

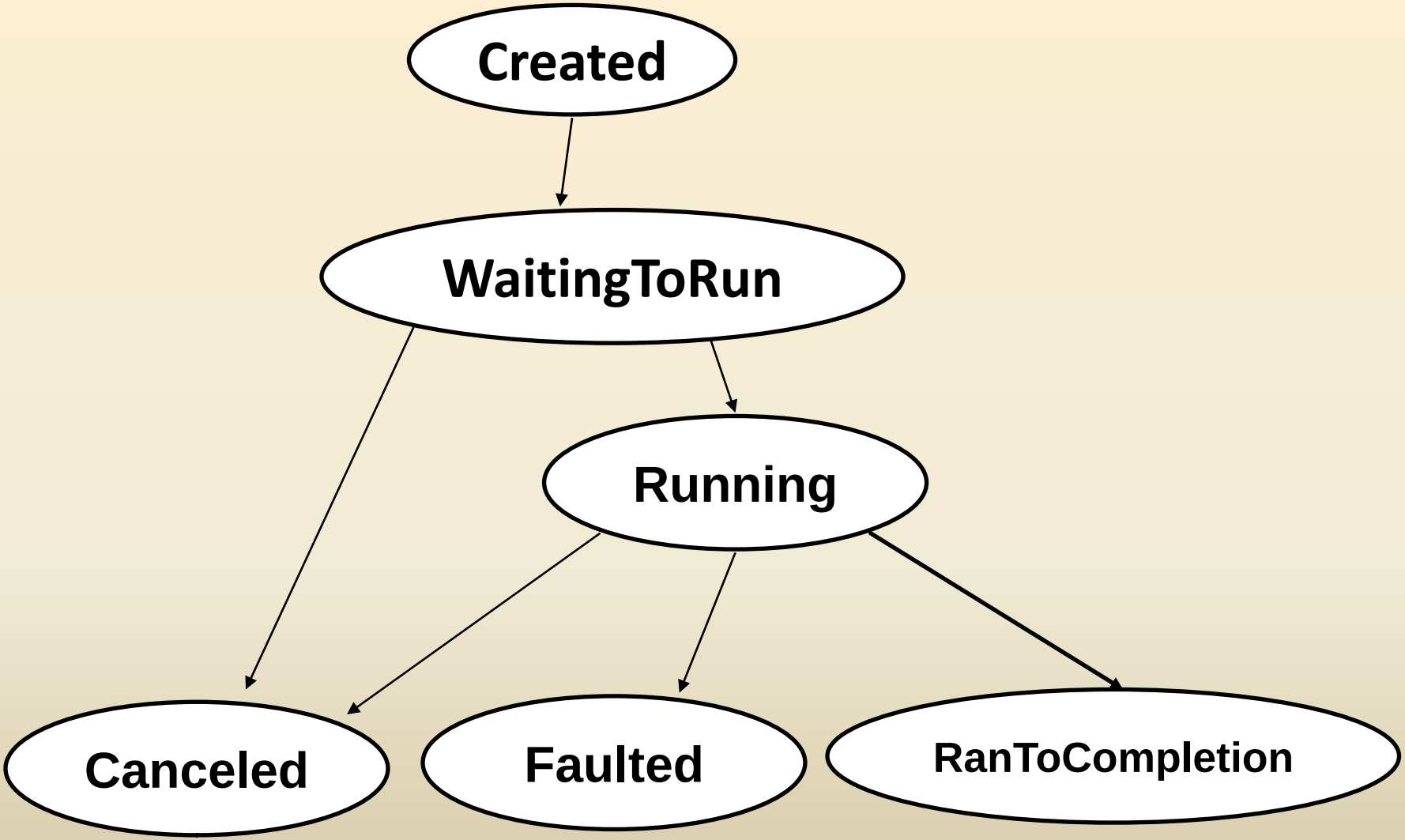
Тип Task<T> позволяет создавать задачу, возвращающую значение типа T.

В качестве рабочего элемента используется объект типа Func<T> (в виде метода или лямбда-выражения)

```
Task<int> t = Task.Run(() => { return 3; });
..
Console.WriteLine(t.Result);
```

Обращение к свойству Result блокирует поток до тех пор, пока задача не завершится.

Статусы задачи



Статус задачи

```
var task = Task.Run(() => {
    DoSomething();
});
Console.WriteLine(task.Status);
if(task.IsFaulted)
    Console.WriteLine(“something bad happened”);
else if(task.IsCanceled)
     Console.WriteLine(“something bad happened”);
```

Вложенные задачи

- В коде задачи можно запускать вложенные задачи, которые могут быть *дочерними* и *недочерними*
- *Недочерние* задачи обладают независимостью от родительской задачи: родитель не дожидается завершения вложенной задачи, статусы задач не взаимосвязаны

```
Task t = Task.Run(() => {
        DoSomething();
        Task.Run(() => InnerTaskWork());
});
```

Вложенные задачи

- Дочерняя вложенная задача является связанной с родительской:
  - родитель дожидается завершения дочерней задачи;
  - статусы задач при исключениях взаимосвязаны.

```
Task.Run(() => {
    Console.WriteLine(“parent task”);
    Task.Factory.StartNew(() => {
                Console.WriteLine(“child task”);
    }, TaskCreationOptions.AttachedToParent);
});
```

Механизм согласованной отмены задач

- В библиотеке TPL (Parallel, Task, PLINQ) используется унифицированная схема согласованной отмены:
  - Управляющий поток, владеющий объектом **CancellationTokenSource**, может выдать сигнал отмены с помощью метода **Cancel, CancelAfter**
  - При поступлении сигнала отмены планировщик отменяет еще не запланированные действия
  - Параллельные задачи (или итерации в Parallel.For) могут контролировать поступление сигнала отмены и осуществлять преждевременное завершение

Запрос отмены задачи

```
// Управляющий объект для отмены
var cts = new CancellationTokenSource();
var token = cts.Token;
Task t = new Task( () => { .. });
t.Start();
..
cts.Cancel();
```

- Обработка отмены задачи реализуется с помощью методов токена отмены:

  bool IsCancellationRequested()

  Проверка токена отмены: поступил сигнал отмены или нет

  void ThrowIfCancellationRequested()

  Вброс исключения при поступлении сигнала отмены

Обработка отмены

```
..
var cts = new CancellationTokenSource();
var token = cts.Token;
Task.Run(() => {
        while(true)    {
                if(token.IsCancellationRequested)
                        break;
                ..
        }
});
```

Обработка отмены

```
..
var cts = new CancellationTokenSource();
var token = cts.Token;
Task.Run(() => {
        while(true)   {
                token.ThrowIfCancellationRequested();
                ..
        }
});
```

Обработка исключений

- Для обработки исключений, которые могут возникнуть в задачах, try-блок оформляет вызов метода ожидания:

```
try
{
        Task.WaitAll(t1, t2);
}
catch(AggregateException ae)
{
        // Обработка ошибок
}
```

- Для обработки исключений параллельного кода используется объект типа **AggregateException**, который агрегирует все возникнувшие исключения
- Список единичных исключений можно получить с помощью свойства **InnerExceptions**.

```
catch(AggregateException ae)
{
    foreach(Exception e in ae.InnerExceptions)
        Console.WriteLine(e.Message);
}
```

Обработка исключений. Flatten

Для упрощения обработки исключений в случае нескольких уровней вложенности задач применяется метод Flatten, преобразующий иерархическую структуру вложенных исключений в плоский список:

```
..
catch(AggregateException ae)
{
    foreach(Exception e in ae.Flatten().InnerExceptions)
        Console.WriteLine(e.Message);
}
```

Задачи-продолжения

- Задачи-продолжения предназначены для планирования запуска задач после завершения предшествующих задач с тем или иным статусом завершения

- Задачи-продолжения позволяют без дополнительных средств синхронизации реализовать критическую секцию и конструкцию барьера

Задачи-продолжения

- Для планирования задачи-продолжения используются методы

  ContinueWith, ContinueWhenAll, ContinueWhenAny

```
Task first = new Task(DoSomething);
Task second = first.ContinueWith(ShowResults);
```

Задачи-продолжения

- Обработчик задачи-продолжения в качестве аргумента принимает предшествующую задачу

```
var tSecond = tFirst.ContinueWith(t => {
        Console.WriteLine("{0}\n{1}", t.Result, t.Status);
});
```

- Можно указать в каких случаях следует вызывать задачу-продолжения, используя константны перечисления TaskContinuationOptions:

  OnlyOnRanToCompletion, OnlyOnCanceled, OnlyOnFaulted, NotOnFaulted, NotOnCancelled, NotOnRanToCompletion

```
// Основная задача, выполняющая расчёт
Task t1 = Task<int>.Factory.StartNew(() => FindDecision());
// Вывод результатов в отдельной задаче
Task t2 = t1.ContinueWith((prev) =>
        Console.WriteLine("Result: {0}", prev.Result),
        TaskContinuationOptions.OnlyOnRanToCompletion);
// Обработчик ошибок
Task t3 = t1.ContinueWith((prev) =>
        Console.WriteLine("Error: {0}",
        prev.Exception.InnerException.Message),
        TaskContinuationOptions.OnlyOnFaulted);
// Задача была отменена
Task t4 = t1.ContinueWith((prev) => .. ,
                        TaskContinuationOptions.OnlyOnCanceled);
```

```
// Объявляем задачи, которые могут выполняться параллельно
Task[] tasks = new Task[3] ;
..
// Планируем выполнение критической секции
Task tCr = Task.Factory.ContinueWhenAll(tasks,
      (tt) =>  DoCritical() );

// Параллельные задачи
Task t5 = tCr.ContinueWith(Work5);
Task t6 = tCr.ContinueWith(Work6);
// Запускаем задачи
tasks[0].Start(); tasks[1].Start(); tasks[2].Start();
// Ожидаем завершения последней задачи
t6.Wait();
```