TPL Dataflow

Зяблицкая Яна Б10

Конвейер потока данных (Task Parallel Library Dataflow)

- **Что это?**

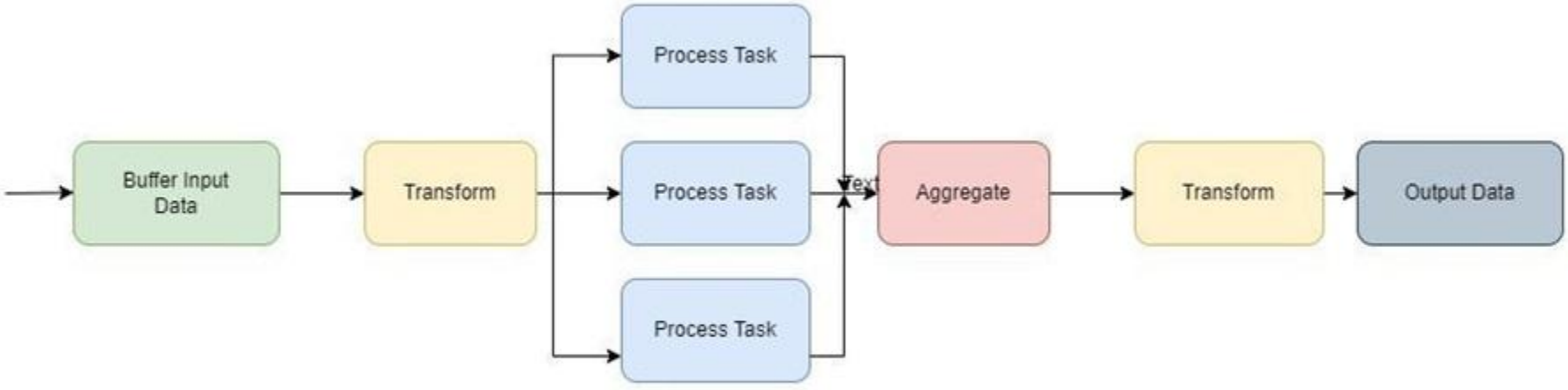
  Библиотека в .NET для построения потоков обработки данных. Разбивает обработку данных на ‘блоки’.

- **Почему это удобно?**

  Библиотека сама заботится об управлении потоками, очередями и балансировкой нагрузки.

- **Для каких задач библиотека используется?**
  - Обработки потоков данных
  - Параллельной обработки данных
  - Построения конвейеров
  - Асинхронного программирования

Основные принципы работы TPL Dataflow



TPL Dataflow Workflow

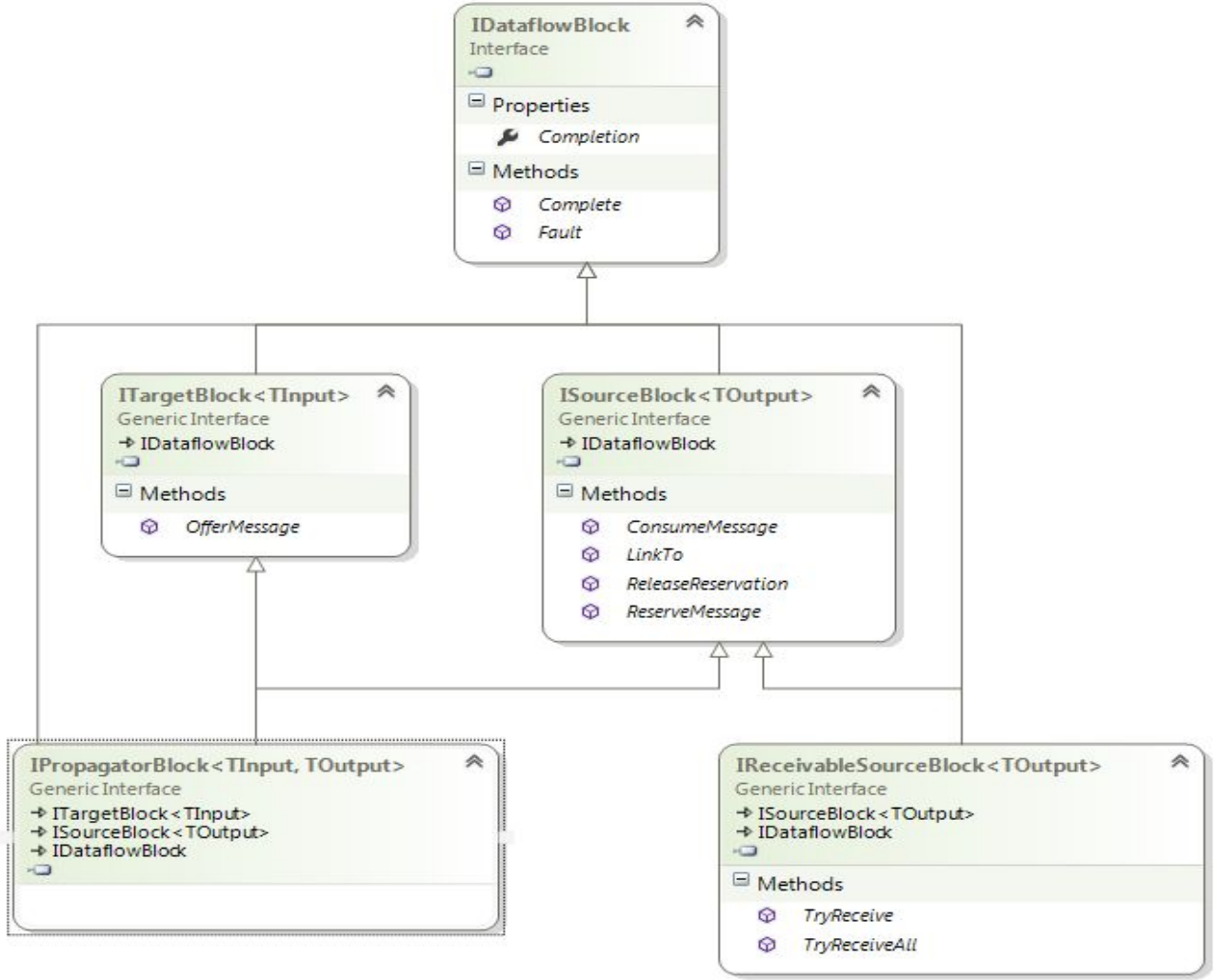
источник

Главные концепции

- `NuGet package System.Threading.Tasks.Dataflow`

- Ключевые интерфейсы от которых наследуются все классы блоков
  - `IDataflowBlock`
  - `ITargetBlock<TInput>`
  - `ISourceBlock<TOutput>`
  - `IPropagatorBlock<TInput, TOutput>`
  - `IReceivableSourceBlock<TOutput>`

- Передача сообщений
  - `public static bool Post<TInput> (ITargetBlock<TInput> target, TInput item);`
  - `public static Task<bool> SendAsync<TInput> (ITargetBlock<TInput> target, TInput item, CancellationToken cancellationToken);`
  - `public static bool TryReceive<TOutput> (IReceivableSourceBlock<TOutput> source, out TOutput item);`
  - `public static Task<TOutput> ReceiveAsync<TOutput> (ISourceBlock<TOutput> source);`


IDataflowBlock
Interface
Properties
Completion
Methods
Complete
Fault
ITargetBlock<TInput>
Generic Interface
IDataflowBlock
Methods
OfferMessage
ISourceBlock<TOutput>
Generic Interface
IDataflowBlock
Methods
ConsumeMessage
LinkTo
ReleaseReservation
ReserveMessage
IPropagatorBlock<TInput, TOutput>
Generic Interface
ITargetBlock<TInput>
ISourceBlock<TOutput>
IDataflowBlock
IReceivableSourceBlock<TOutput>
Generic Interface
ISourceBlock<TOutput>
IDataflowBlock
Methods
TryReceive
TryReceiveAll

Интерфейс *IDataflowBlock*

```
public interface IDataflowBlock
{
    // Завершает блок, сигнализируя, что больше данных не будет.
    void Complete();
    // Переводит блок в состояние ошибки.
    void Fault(Exception exception);
    // Возвращает задачу, которая завершится, когда блок завершит свою работу.
    Task Completion { get; }
}
```

Интерфейс *ITargetBlock<TInput>*

```
public interface ITargetBlock<TInput> :
IDataflowBlock

{

    // Низкоуровневая передача сообщения в блок.

    DataflowMessageStatus OfferMessage(

        DataflowMessageHeader messageHeader,

        TInput messageValue,

        ISourceBlock<TInput> source,

        bool consumeToAccept);

}
```

- `OfferMessage(...)`: используется для передачи сообщений от источника в блок.

- `Accepted`
- `Postponed`
- `Declined`
- `DecliningPermanently`

Предопределенные типы блоков потоков данных

```
bufferBlock<T>
BroadcastBlock<T>
WriteOnceBlock<T>
```

Три категории:

- блоки буферизации

```
TransformManyBlock<TInput, TOutput>
TransformBlock<TInput, TOutput>
ActionBlock<TInput>
```

- блоки выполнения

```
BatchBlock<T>
JoinBlock<T1, T2>
BatchedJoinBlock<T1,T2>
```

- блоки группировки

Типы конфигурации блоков

`DataflowBlockOptions`

- `BoundedCapacity`: максимальное количество элементов в блоке (по умолчанию не ограничено).
- `TaskScheduler`: планировщик задач для управления потоками.
- `CancellationToken`: токен для отмены обработки.
- `EnsureOrdered`: сохранять ли порядок обработки (по умолчанию true)

`ExecutionDataflowBlockOptions`

- `MaxDegreeOfParallelism`: количество задач, которые могут выполняться одновременно (по умолчанию 1).
- `BoundedCapacity`: максимальное количество элементов в очереди.
- `TaskScheduler, CancellationToken, EnsureOrdered`.

`GroupingDataflowBlockOptionsJoinBlock<T1, T2>`

- `Greedy`: определяет, будет ли блок "жадно" собирать элементы (по умолчанию true).
- `BoundedCapacity, TaskScheduler, CancellationToken, EnsureOrdered`.

Три категории:

- блоки буферизации
- блоки выполнения
- блоки группировки

Пример построения конвейера

```
static async Task Main(string[] args) {

// 1. Источник данных:

BufferBlock var bufferBlock = new BufferBlock<int>();

// 2. Преобразование Удвоение числа

var transformBlock = new TransformBlock<int, int>(x => x * 2);

// 3. Группировка: BatchBlock собирает числа в массивы по 3

var batchBlock = new BatchBlock<int>(3);

// 4. Вывод: ActionBlock выводит группы чисел

var actionBlock = new ActionBlock<int[]>(batch => {

     Console.WriteLine($"Batch: {string.Join(", ", batch)}");

     }

);
```

Пример построения конвейера

```
// Устанавливаем связи между блоками
bufferBlock.LinkTo(transformBlock, new DataflowLinkOptions { PropagateCompletion = true });
transformBlock.LinkTo(batchBlock, new DataflowLinkOptions { PropagateCompletion = true });
batchBlock.LinkTo(actionBlock, new DataflowLinkOptions { PropagateCompletion = true });
// Заполняем BufferBlock числами от 1 до 10
for (int i = 1; i <= 10; i++) {
       bufferBlock.Post(i);
}
// Завершаем BufferBlock (остальные блоки завершаются автоматически из-за PropagateCompletion)
bufferBlock.Complete();
// Ожидаем завершения конечного ActionBlock
await actionBlock.Completion;
Console.WriteLine("Processing complete!"); }
}
```

Вывод:

Batch: 2, 4, 6

Batch: 8, 10, 12

Batch: 14, 16, 18

Batch: 20

Processing complete!

Пример задачи

Задача

Онлайн-магазин принимает заказы, которые обрабатываются следующим образом:

1. Заказы поступают из базы данных.
2. Для каждого заказа нужно проверить наличие товаров на складе.
3. Если товар есть, нужно подсчитать общую стоимость заказа.
4. После обработки заказ нужно записать обратно в базу данных.

Вводные данные

```
// Исходные данные:

var orders = new List<Order> {

new Order { Id = 1, ProductId = 101, Quantity = 2 },

new Order { Id = 2, ProductId = 102, Quantity = 1 },

new Order { Id = 3, ProductId = 103, Quantity = 5 },

new Order { Id = 4, ProductId = 104, Quantity = 3 },

new Order { Id = 5, ProductId = 101, Quantity = 1 }

};
```

```
class Order {

public int Id { get; set; }

public int ProductId { get; set; }

public int Quantity { get; set; }

public bool IsInStock { get; set; }

public double TotalPrice { get; set; }

}
```

BufferBlock и TransformBlock

```
// 1. Источник данных: BufferBlock для хранения заказов
var ordersBuffer = new BufferBlock<Order>();
// 2. Проверка наличия на складе: TransformBlock
var checkStockBlock = new TransformBlock<Order, Order>(async order => {
        Console.WriteLine($"Checking stock for Order {order.Id}...");
        await Task.Delay(500);  // Эмуляция запроса в базу данных
        order.IsInStock = order.ProductId % 2 == 0;  // Эмуляция: товар есть на складе, если ID продукта чётный
        return order;
        }, new ExecutionDataflowBlockOptions {
                MaxDegreeOfParallelism = 3 // Проверяем до 3 заказов одновременно
        }
);
```

Обработка

```
// 3. Подсчёт стоимости заказа: TransformBlock
var calculatePriceBlock = new TransformBlock<Order, Order>(async order => {
      if (!order.IsInStock) {
            Console.WriteLine($"Order {order.Id} is out of stock!");
            return order;
      }
      Console.WriteLine($"Calculating price for Order {order.Id}...");
      await Task.Delay(300); // Эмуляция сложных расчётов
      order.TotalPrice = order.Quantity * 100; // Эмуляция цены: 100 за единицу товара
      return order;
      }, new ExecutionDataflowBlockOptions {
            MaxDegreeOfParallelism = 5 // Рассчитываем до 5 заказов одновременно
      }
);
```

Action Block

```
// 4. Сохранение результатов: ActionBlock
 var saveOrderBlock = new ActionBlock<Order>(async order => {
       if (!order.IsInStock) {
            Console.WriteLine($"Skipping saving Order {order.Id} (out of stock).");
            return;
      }
      Console.WriteLine($"Saving Order {order.Id} with Total Price: {order.TotalPrice:C}");
      await Task.Delay(200); // Эмуляция записи в базу данных
      }, new ExecutionDataflowBlockOptions {
            MaxDegreeOfParallelism = 2 // Сохраняем до 2 заказов одновременно
      }
);
```

Связывание блоков

```
ordersBuffer.LinkTo(checkStockBlock, new DataflowLinkOptions { PropagateCompletion = true });
checkStockBlock.LinkTo(calculatePriceBlock, new DataflowLinkOptions { PropagateCompletion = true });
calculatePriceBlock.LinkTo(saveOrderBlock, new DataflowLinkOptions { PropagateCompletion = true });
// 5. Добавляем заказы в источник
foreach (var order in orders) {
     await ordersBuffer.SendAsync(order);
}
// Завершаем работу источника
ordersBuffer.Complete();
// Ожидаем завершения всех операций
await saveOrderBlock.Completion;
Console.WriteLine("All orders processed!");
```

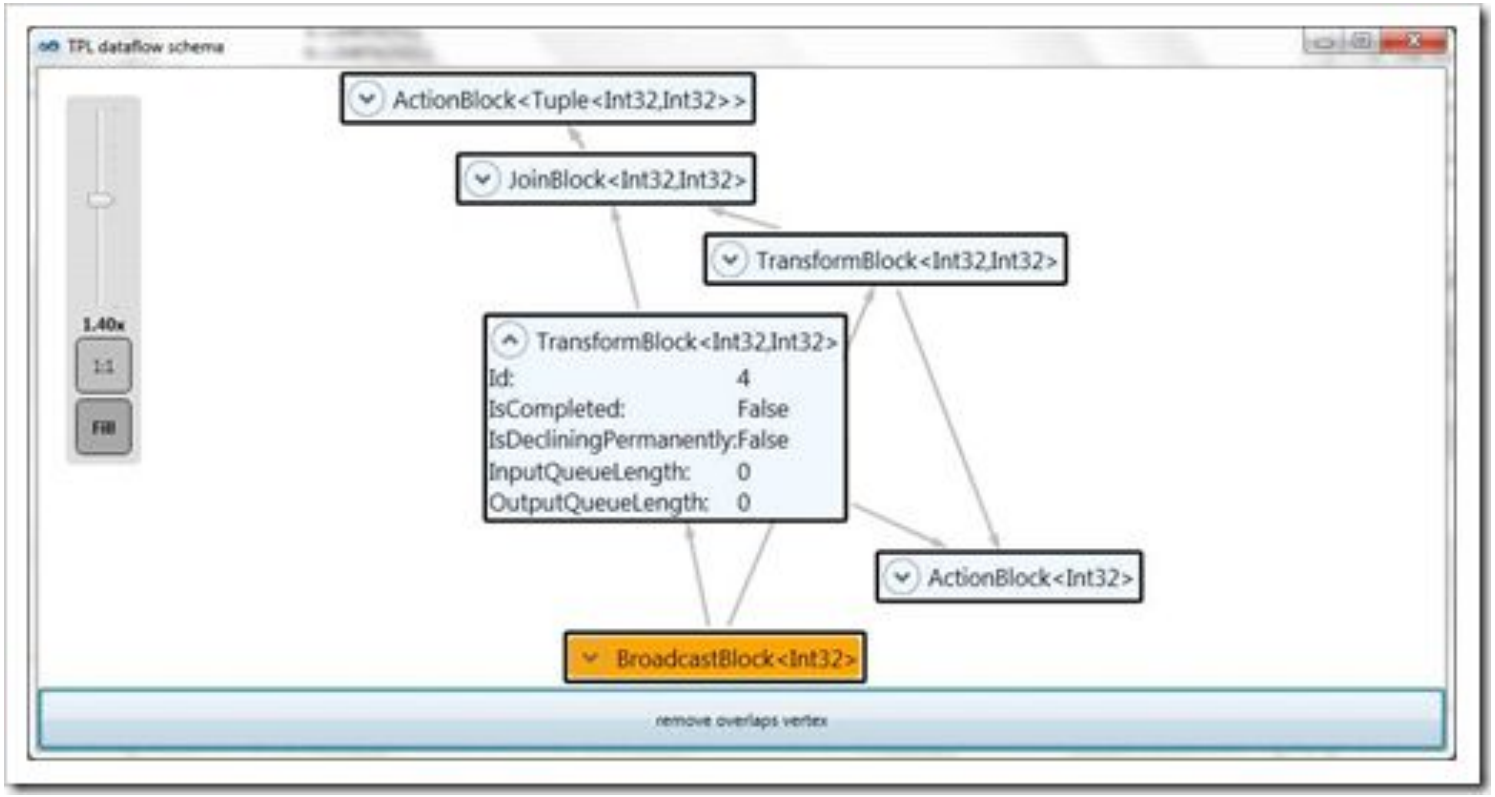
Вывод задачи

```
Checking stock for Order 3...
Checking stock for Order 1...
Checking stock for Order 2...
Checking stock for Order 4...
Checking stock for Order 5...
Calculating price for Order 2...
Order 1 is out of stock!
Order 3 is out of stock!
Skipping saving Order 1 (out of stock).
Skipping saving Order 3 (out of stock).
Saving Order 2 with Total Price: $100.00
Calculating price for Order 4...
Order 5 is out of stock!
Skipping saving Order 5 (out of stock).
Saving Order 4 with Total Price: $300.00
All orders processed!
```

Когда не стоит использовать TPL Dataflow?

- Если задача требует минимальной обработки данных без сложного управления потоками
- Если задача требует минимальной задержки (например, в системах реального времени), так как TPL Dataflow не оптимизирован для задач, где важна минимальная задержка обработки одного элемента
- Если нужно управлять потоками вручную, задавая их количество, приоритет или другие параметры

Визуализация TPL Dataflow с помощью TPL DataFlow Debugger Visualizer



Источники

- https://medium.com/@thanh.pham/introduction-to-tpl-dataflow-in-c-f99dc510763e
- https://learn.microsoft.com/en-us/dotnet/standard/parallel-programming/dataflow-task-parallel-library
- https://www.linkedin.com/pulse/introduction-tpl-dataflow-farhad-shiri
- https://hamidmosalla.com/2018/08/04/what-is-tpl-dataflow-in-net-and-when-should-we-use-it/