

TPL. Параллельное программирование

№ урока: 14 **Курс:** C# Professional

Средства обучения: Компьютер с установленной Visual Studio

Обзор, цель и назначение урока

Библиотека распараллеливания задач (TPL) – вероятно, самое главное новшество .NET Framework 4.0. Эта библиотека усовершенствует многопоточное программирование двумя основными способами: во-первых, она упрощает создание и применение многих потоков, а, во-вторых, она позволяет автоматически использовать несколько процессоров. Иными словами, TPL открывает возможности для автоматического масштабирования приложений с целью эффективного использования ряда доступных процессоров. Еще одним важным средством параллельного программирования, появившемся в .NET Framework 4.0, можно считать PLINQ. Язык PLINQ дает возможность без труда внедрять параллелизм в запросы, что, в свою очередь, позволяет легко (и безопаснее) использовать системные ресурсы.

Изучив материал данного занятия, учащийся сможет:

- Понимать назначение TPL
- Использовать TPL для эффективного управления многопоточностью
- Использовать PLINQ для формирования параллельно обрабатываемых запросов

Содержание урока

1. Введение в параллельное программирование
2. Класс [Task](#)
3. Создание задачи
4. Методы ожидания выполнения задачи
5. Класс [TaskFactory](#)
6. Лямбда-выражения в качестве задачи
7. Создание продолжения задачи
8. Возврат значений из задачи
9. Отмена задачи
10. Класс [Parallel](#)
11. PLINQ

Резюме

- Библиотека TPL определена в пространстве имен `System.Threading.Tasks`. Но для работы с ней обычно требуется также включать в программу пространство имен `System.Threading`, поскольку оно поддерживает синхронизацию и другие средства многопоточной обработки, в том числе и те, что входят в класс [Interlocked](#).
- Параллелизм в программе может существовать в двух основных формах. Первая из них называется параллелизмом данных. При таком подходе одна операция над совокупностью данных разбивается на два или больше параллельно выполняемых потока, в каждом из которых обрабатывается часть данных. Второй способ называется параллелизмом задач. При таком подходе две или больше операций выполняются параллельно.
- Элементарная единица исполнения инкапсулируется в TPL средствами класса [Task](#). В отличие от класса [Thread](#), имеется в виду инкапсуляция асинхронной операции, а не потока исполнения. На системном уровне поток по-прежнему остается элементарной единицей исполнения, но при использовании задач соответствие между объектом класса [Task](#) и потоком исполнения уже не является взаимно-однозначным. Кроме того, исполнением задач управляет планировщик задач, который работает с пулом потоков. Это, например, означает, что несколько задач могут выполняться в одном потоке.
- Всегда нужно помнить, что выполнение задачи по умолчанию происходит в фоновом потоке, что означает прерывание выполнения задачи по завершении работы основного

потока. Если задача завершена, она не может быть перезапущена. Иного способа повторного запуска задачи на исполнение, кроме создания ее снова, не существует.

- В отличие от потоков [Thread](#), задачи не обладают уникальными именами. Вместо этого для идентификации задач используется свойство `Id` типа `int`. Идентификаторы задач уникальны, но не упорядочены, не стоит делать каких-либо предположений касательно очередности создания задач, основываясь на значениях идентификаторов. Идентификатор исполняемой в настоящий момент задачи можно получить с помощью свойства `CurrentId`. Это свойство принимает значение идентификатора задачи, если обращение к нему происходит из контекста задачи либо `null`, если обращающийся код не является исполняемым внутри какой-либо задачи.
- Метод `Wait` позволяет приостановить выполнение вызывающего потока до момента завершения задачи. В момент использования этого метода могут быть сгенерированы два исключения: [ObjectDisposedException](#), если задача освобождена посредством вызова `Dispose`; [AggregateException](#), если задача сама генерирует исключение или же отменяется. Если у задачи существуют порожденные ею подзадачи и исключения происходят внутри них, все эти исключения будут собраны и упакованы в единое исключение [AggregateException](#).
- Дождаться завершения более одной задачи можно используя метода `WaitAll`. Он позволяет приостановить выполнения основного потока до тех пор, пока не завершатся все задачи, переданные ему в качестве аргументов `params`. Если же нужно дождаться завершения не всех задач, а любой, следует применять метод `WaitAny`. Метод `WaitAny` возвращает `Id` задачи, завершившейся первой.
- Класс [Task](#) реализует интерфейс [IDisposable](#), таким образом, для него определен метод `Dispose`. Этот метод нужно вызывать, если в процессе работы программы создается большое количество задач, оставляемых на произвол судьбы, или же необходимо явным образом освобождать ресурсы после завершения работы задачи. Вызов метода `Dispose` допустим только после завершения задачи, поэтому его следует дожидаться, например, применяя метод `Wait`. Если попытаться вызвать `Dispose` для все еще работающей задачи, то будет сгенерировано исключение [InvalidOperationException](#).
- Создать и запустить задачу на выполнение можно при помощи фабричных методов класса [TaskFactory](#). Использовать их нужно в тех случаях, когда создаваемая задача должна начать свою работу сразу после создания.
- Кроме использования обычного метода в качестве задачи, можно использовать лямбда-выражения. Они оказываются особенно полезными в тех случаях, когда единственное назначение метода – решение одноразовой задачи. Лямбда-выражения могут составлять отдельную задачу или же вызывать другие методы.
- Продолжение – это одна задача, которая автоматически начинается после завершения другой задачи. Создать продолжение можно, например, используя метод `ContinueWith`. Очень часто в качестве продолжения задачи используются лямбда-выражения.
- Из задачи можно получить возвращаемое значение. Во-первых, это означает, что с помощью задач можно вычислить некоторый результат. Во-вторых, вызывающий процесс окажется заблокированным до тех пор, пока не будет получен результат. Задачи, возвращающие результат, создаются с использованием делегата [Func](#), а не [Action](#), как обычные задачи.
- Отмена задачи выполняется с использованием признака отмены. Для того, чтобы создать отменяемую задачу, необходимо использовать специальный конструктор, либо фабричный метод класса [TaskFactory](#). Для отмены в задаче должна быть получена копия признака отмены и организован контроль этого признака с целью отслеживать саму отмену. Такое отслеживание можно организовать тремя способами: опросом, методом обратного вызова, с помощью дескриптора ожидания.
- Класс [Parallel](#) позволяет использовать параллельное программирование на основе задач, не прибегая к управлению задачами явным образом. Метод `Invoke` позволяет выполнять один или несколько методов параллельно. Он масштабирует исполнение кода, используя доступные процессоры, если имеется такая возможность. Выполняемые методы должны быть совместимы с делегатом [Action](#).
- `Invoke` сначала инициирует выполнение, а затем ожидает завершения всех переданных ему методов. Такой подход не гарантирует, что методы будут

действительно выполняться параллельно. Кроме того, отсутствует возможность указать порядок выполнения методов от первого и до последнего, этот порядок не обязательно будет совпадать со списком аргументов.

- В классе `Parallel` также существует метод `For`, который позволяет выполнять цикл `for` параллельно. Это оказывается очень удобным, если внутри цикла необходимо выполнить сложную, нетривиальную задачу. В противном случае, распараллеливание может привести к потере производительности. Применение параллельного `For` не гарантирует параллельного выполнения. Оно будет реализовано только в том случае, если для работы доступно несколько процессоров.
- Прервать выполнение цикла `Parallel.For` можно при помощи метода `Break`. Вызов данного метода не гарантирует мгновенной остановки цикла, но гарантирует выполнение всех итераций, предшествующих вызову метода. Использовать `Break` удобно, когда в цикле происходит поиск каких-то значений. Для безусловной остановки цикла, которая не будет обращать внимание на любые шаги, которые еще могут быть выполнены, лучше применять метод `Stop`, а не `Break`.
- Кроме цикла `For`, параллельно также можно выполнять и цикл `Foreach`. Для него действуют все те же ограничения, что и для цикла `For`.
- PLINQ представляет собой параллельный вариант языка интегрированных запросов LINQ и тесно связан с библиотекой TPL. PLINQ применяется, главным образом, для достижения параллелизма данных внутри запроса. Основу PLINQ составляет класс `ParallelEnumerable`, определенный в пространстве имен `System.Linq`.

Закрепление материала

1. Что такое TPL?
2. Какие преимущества имеет TPL по сравнению с обычной многопоточностью?
3. Как создать задачу?
4. Что такое продолжение задачи?
5. Как получить значение из выполняемой задачи?
6. Каким образом можно отменить задачу?
7. В чем состоят особенности применения класса `Parallel`?
8. Что такое PLINQ?
9. Как организовать параллельно обрабатываемый запрос?
10. Как отменить параллельный запрос?

Дополнительное задание

Создайте массив чисел размерностью в 1 000 000 или более. Используя генератор случайных чисел, проинициализируйте этот массив значениями. Создайте PLINQ запрос, который позволит получить все нечетные числа из исходного массива.

Самостоятельная деятельность учащегося

Задание 1

Выучите основные конструкции и понятия, рассмотренные на уроке.

Задание 2

Создайте два метода, которые будут выполняться в рамках параллельных задач. Организуйте вызов этих методов при помощи `Invoke` таким образом, чтобы основной поток программы (метод `Main`) не остановил свое выполнение.

Задание 3

Зайдите на сайт MSDN.

Используя поисковые механизмы MSDN, найдите самостоятельно описание темы по каждому примеру, который был рассмотрен на уроке, так, как это представлено ниже, в разделе «Рекомендуемые ресурсы», описания данного урока. Сохраните ссылки и дайте им короткое описание.

Рекомендуемые ресурсы

MSDN: TPL

<http://msdn.microsoft.com/ru-ru/library/dd460717.aspx>

MSDN: Параллелизм данных

<http://msdn.microsoft.com/ru-ru/library/dd537608.aspx>

MSDN: Параллелизм задач

<http://msdn.microsoft.com/ru-ru/library/dd537609.aspx>

MSDN: Потенциальные ошибки, связанные с параллелизмом данных и задач

<http://msdn.microsoft.com/ru-ru/library/dd997392.aspx>

MSDN: PLINQ

<http://msdn.microsoft.com/ru-ru/library/dd460688.aspx>