O'REILLY®

Współbieżność w języku C#

Receptury

Helion

Stephen Cleary

Tytuł oryginału: Concurrency in C# Cookbook

Tłumaczenie: Lech Lachowski

ISBN: 978-83-283-2990-4

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *http://helion.pl* (księgarnia internetowa, katalog książek)

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
ftp://ftp.helion.pl/przyklady/wspcre.zip

Spis treści

ROZDZIAŁ 3.

Podstawy przetwarzania równoległego

W tym rozdziale omówimy wzorce dla programowania równoległego. Programowanie równoległe jest stosowane do podzielenia ograniczonych możliwościami obliczeniowymi fragmentów pracy i rozdzielenia ich na wiele wątków. Przedstawione poniżej receptury przetwarzania równoległego uwzględniają jedynie zadania uzależnione od wydajności procesora. Jeśli masz naturalnie asynchroniczne operacje (takie jak zadania ograniczone możliwościami we-wy), które chcesz wykonywać równolegle, zapoznaj się z rozdziałem 2., a w szczególności z recepturą 2.4.

Abstrakcje przetwarzania równoległego omówione w tym rozdziale są częścią biblioteki zadań równoległych (ang. *Task Parallel Library* — **TPL**). Jest ona wbudowana we framework .NET, ale nie jest dostępna na wszystkich platformach (zobacz tabela 3.1):

Tabela 3.1. Wsparcie platform dla biblioteki TPL

Platforma	Wsparcie dla przetwarzania równoległego
.NET 4.5	✓
.NET 4.0	✓
Mono iOS/Droid	✓
Windows Store	✓
Windows Phone Apps 8.1	✓
Windows Phone SL 8.0	×
Windows Phone SL 7.1	×
Silverlight 5	×

3.1. Równoległe przetwarzanie danych

Problem

Mamy kolekcję danych i musimy wykonać tę samą operację na każdym elemencie tych danych. Ta operacja jest ograniczona możliwościami obliczeniowymi procesora i może zająć trochę czasu.

Rozwiązanie

Typ `Parallel` zawiera metodę `ForEach` specjalnie przeznaczoną do tego celu. Kod przedstawiony w tym przykładzie przyjmuje kolekcję macierzy i odwraca je wszystkie:

```
void RotateMatrices(IEnumerable<Matrix> matrices, float degrees)
{
    Parallel.ForEach(matrices, matrix => matrix.Rotate(degrees));
}
```

W niektórych sytuacjach będziemy chcieli zatrzymać pętlę wcześniej, na przykład przy napotkaniu nieprawidłowej wartości. Poniższy kod odwraca każdą macierz, ale jeśli zostanie napotkana nieprawidłowa macierz, pętla zostanie przerwana:

```
void InvertMatrices(IEnumerable<Matrix> matrices)
{
    Parallel.ForEach(matrices, (matrix, state) =>
    {
        if (!matrix.IsInvertible)
            state.Stop();
        else
            matrix.Invert();
    });
}
```

Bardziej powszechną sytuacją jest potrzeba posiadania możliwości anulowania równoległej pętli. Różni się to od zatrzymania pętli. Pętla jest *zatrzymywana* z wewnątrz pętli, a *anulowanie* odbywa się z zewnątrz pętli. Przykładowo `Cancella` `↪tionTokenSource` może być anulowany za pomocą przycisku anulowania, co spowoduje anulowanie równoległej pętli, takiej jak ta:

```
void RotateMatrices(IEnumerable<Matrix> matrices, float degrees,
    CancellationToken token)
{
    Parallel.ForEach(matrices,
```

```
            new ParallelOptions { CancellationToken = token },
            matrix => matrix.Rotate(degrees));
    }
```

Należy pamiętać o tym, że każde zadanie równoległe może działać w innym wątku, dlatego każdy współdzielony stan musi być chroniony. Poniższy kod odwraca każdą macierz i zlicza macierze, które nie mogły zostać odwrócone:

```
// Uwaga: to nie jest najbardziej efektywna implementacja.
// To tylko przykład użycia blokady do ochrony współdzielonego stanu.
int InvertMatrices(IEnumerable<Matrix> matrices)
{
    object mutex = new object();
    int nonInvertibleCount = 0;
    Parallel.ForEach(matrices, matrix =>
    {
        if (matrix.IsInvertible)
        {
            matrix.Invert();
        }
        else
        {
            lock (mutex)
            {
                ++nonInvertibleCount;
            }
        }
    });
    return nonInvertibleCount;
}
```

Dyskusja

Metoda `Parallel.ForEach` umożliwia przetwarzanie równoległe na sekwencji wartości. Podobnym rozwiązaniem jest technologia **PLINQ** (ang. *Parallel LINQ*). PLINQ zapewnia prawie te same możliwości za pomocą składni podobnej do LINQ. Jedną z różnic pomiędzy `Parallel` i PLINQ jest to, że PLINQ zakłada możliwość wykorzystania wszystkich rdzeni na komputerze, podczas gdy `Parallel` będzie dynamicznie reagować na zmieniające się warunki procesora.

`Parallel.ForEach` to równoległa pętla `foreach`. Jeśli trzeba wykonać równoległą pętlę `for`, klasa `Parallel` obsługuje również metodę `Parallel.For`. Ta metoda jest szczególnie użyteczna, jeśli masz wiele tablic danych, z których wszystkie wykorzystują ten sam indeks.

Zobacz również

Receptura 3.2 omawia równoległą agregację szeregu wartości, w tym sum i średnich.

Receptura 3.5 opisuje podstawy PLINQ.

Rozdział 9. omawia anulowanie.

3.2. Równoległa agregacja

Problem

Na zakończenie równoległej operacji musimy zagregować wyniki. Przykładami agregacji są sumy, średnie itp.

Rozwiązanie

Klasa `Parallel` obsługuje agregację poprzez koncepcję **wartości lokalnych**, które są zmiennymi występującymi lokalnie w ramach równoległej pętli. Oznacza to, że ciało pętli może po prostu uzyskać dostęp do tej wartości bezpośrednio, bez konieczności przejmowania się synchronizacją. Gdy pętla jest gotowa do agregacji wszystkich swoich lokalnych wyników, robi to za pomocą delegata `localFinally`. Należy zwrócić uwagę, że delegat `localFinally` *potrzebuje* zsynchronizować dostęp do zmiennej, która przechowuje wynik końcowy. Oto przykład sumy równoległej:

```
// Uwaga: to nie jest najbardziej efektywna implementacja.
// To tylko przykład użycia blokady do ochrony współdzielonego stanu.
static int ParallelSum(IEnumerable<int> values)
{
    object mutex = new object();
    int result = 0;
    Parallel.ForEach(source: values,
        localInit: () => 0,
        body: (item, state, localValue) => localValue + item,
        localFinally: localValue =>
        {
            lock (mutex)
                result += localValue;
        });
        return result;
}
```

Parallel LINQ ma bardziej naturalną możliwość obsługi agregacji niż klasa `Parallel`:

```
static int ParallelSum(IEnumerable<int> values)
{
    return values.AsParallel().Sum();
}
```

No dobrze, to był cios poniżej pasa, ponieważ PLINQ ma wbudowaną obsługę wielu typowych operatorów (na przykład `Sum`). PLINQ oferuje również ogólną obsługę agregacji poprzez operator `Aggregate`:

```
static int ParallelSum(IEnumerable<int> values)
{
    return values.AsParallel().Aggregate(
        seed: 0,
        func: (sum, item) => sum + item
    );
}
```

Dyskusja

Jeśli używasz już klasy `Parallel`, możesz chcieć skorzystać z jej wsparcia dla agregacji. W przeciwnym razie wsparcie PLINQ jest w większości scenariuszy bardziej ekspresyjne i ma krótszy kod.

Zobacz również

Receptura 3.5 omawia podstawy PLINQ.

3.3. Równoległe wywołanie

Problem

Mamy wiele metod wywoływanych równolegle; metody te są (w większości) niezależne od siebie.

Rozwiązanie

Klasa `Parallel` zawiera prostą składową `Invoke`, która jest przeznaczona dla tego scenariusza. Oto przykład kodu, który dzieli tablicę na pół i przetwarza każdą połówkę niezależnie:

```
static void ProcessArray(double[] array)
{
    Parallel.Invoke(
        () => ProcessPartialArray(array, 0, array.Length / 2),
        () => ProcessPartialArray(array, array.Length / 2, array.Length)
    );
}

static void ProcessPartialArray(double[] array, int begin, int end)
{
    // Przetwarzanie wymagające obliczeniowo...
}
```

Do metody `Parallel.Invoke` można również przekazać tablicę delegatów, jeśli liczba wywołań nie jest znana aż do czasu wykonywania:

```
static void DoAction20Times(Action action)
{
    Action[] actions = Enumerable.Repeat(action, 20).ToArray();
    Parallel.Invoke(actions);
}
```

Metoda `Parallel.Invoke` obsługuje anulowanie podobnie jak inne składowe klasy `Parallel`:

```
static void DoAction20Times(Action action, CancellationToken token)
{
    Action[] actions = Enumerable.Repeat(action, 20).ToArray();
    Parallel.Invoke(new ParallelOptions { CancellationToken = token },
actions);
}
```

Dyskusja

Metoda `Parallel.Invoke` jest doskonałym rozwiązaniem dla prostych wywołań równoległych. Nie sprawdzi się jednak zbyt dobrze, jeśli chcesz wywołać akcję dla każdego elementu danych wejściowych (zamiast niej należy użyć metody `Parallel.ForEach`) lub jeśli każda akcja generuje jakieś dane wyjściowe (wtedy należy skorzystać z Parallel LINQ).

Zobacz również

Receptura 3.1 opisuje metodę `Parallel.ForEach`, która wywołuje akcję dla każdego elementu danych.

Receptura 3.5 omawia Parallel LINQ.

3.4. Równoległość dynamiczna

Problem

Mamy bardziej złożoną sytuację równoległą, w której struktura i liczba równoległych zadań zależą od informacji znanych dopiero w czasie wykonywania programu.

Rozwiązanie

Biblioteka zadań równoległych jest skoncentrowana wokół typu `Task`. Klasa `Parallel` i technologia Parallel LINQ zapewniają tylko wygodne funkcje opakowujące dla tego typu. Gdy potrzebna jest równoległość dynamiczna, najłatwiej jest użyć typu `Task` bezpośrednio.

Oto jeden z przykładów, w którym dla każdego węzła binarnego drzewa trzeba wykonać kosztowne przetwarzanie. Struktura drzewa nie będzie znana aż do czasu wykonywania, dlatego jest to dobry scenariusz dla równoległości dynamicznej. Metoda `Traverse` przetwarza bieżący węzeł, a następnie tworzy dwa zadania podrzędne, po jednym dla każdej gałęzi pod węzłem (w tym przykładzie zakładamy, że węzły nadrzędne muszą być przetwarzane przed podrzędnymi). Metoda `ProcessTree` rozpoczyna przetwarzanie poprzez utworzenie zadania nadrzędnego najwyższego poziomu i czekanie na jego zakończenie:

```
void Traverse(Node current)
{
    DoExpensiveActionOnNode(current);
    if (current.Left != null)
    {
        Task.Factory.StartNew(() => Traverse(current.Left),
            CancellationToken.None,
            TaskCreationOptions.AttachedToParent,
            TaskScheduler.Default);
    }
    if (current.Right != null)
    {
        Task.Factory.StartNew(() => Traverse(current.Right),
            CancellationToken.None,
            TaskCreationOptions.AttachedToParent,
            TaskScheduler.Default);
    }
}
```

```
public void ProcessTree(Node root)
{
    var task = Task.Factory.StartNew(() => Traverse(root),
        CancellationToken.None,
        TaskCreationOptions.None,
        TaskScheduler.Default);
    task.Wait();
}
```

Jeśli nie masz sytuacji typu zadanie nadrzędne – podrzędne, możesz zaplanować, aby każde zadanie było uruchamiane po innym, stosując **kontynuację** zadań. Kontynuacja jest osobnym zadaniem, które jest wykonywane, gdy pierwotne zadanie zostanie zakończone:

```
Task task = Task.Factory.StartNew(
    () => Thread.Sleep(TimeSpan.FromSeconds(2)),
    CancellationToken.None,
    TaskCreationOptions.None,
    TaskScheduler.Default);
Task continuation = task.ContinueWith(
    t => Trace.WriteLine("Zadanie zostało wykonane" ),
    CancellationToken.None,
    TaskContinuationOptions.None,
    TaskScheduler.Default);
// Argument "t" dla kontynuacji jest taki sam jak "task".
```

Dyskusja

Powyższy przykładowy kod wykorzystuje właściwości `CancellationToken.None` oraz `TaskScheduler.Default`. Tokeny anulowania zostały opisane w recepturze 9.2, a dyspozytory zadań w recepturze 12.3. Zawsze dobrym pomysłem jest bezpośrednio określić `TaskScheduler` używany przez metody `StartNew` i `ContinueWith`.

Ten układ zadań nadrzędnych i podrzędnych jest typowy dla równoległości dynamicznej. Nie jest jednak wymagany. Można również przechowywać każde nowe zadanie w wątkowo bezpiecznej kolekcji, a następnie za pomocą `Task.WaitAll` poczekać na zakończenie ich wszystkich.

Używanie typu `Task` dla przetwarzania równoległego całkowicie różni się od wykorzystania `Task` dla przetwarzania asynchronicznego. Zobacz poniżej.

W programowaniu równoległym typ `Task` służy dwóm celom: może być zadaniem równoległym lub zadaniem asynchronicznym. Zadania równoległe mogą wykorzystywać składowe blokujące, takie jak `Task.Wait`, `Task.Result`, `Task.WaitAll` i `Task.WaitAny`. Często używają również `AttachedToParent` do tworzenia między zadaniami relacji nadrzędne – podrzędne. Zadania równoległe powinny być tworzone za pomocą metod `Task.Run` lub `Task.Factory.StartNew`.

W przeciwieństwie do tego, w przypadku zadań asynchronicznych powinno się unikać składowych blokujących i preferować użycie `await`, `Task.WhenAll` i `Task.WhenAny`. Zadania asynchroniczne nie używają `AttachedToParent`, ale mogą formować dorozumiany rodzaj relacji nadrzędne – podrzędne poprzez oczekiwanie na kolejne zadanie.

Zobacz również

Receptura 3.3 omawia równoległe wywoływanie sekwencji metod, gdy wszystkie te metody są znane przy rozpoczynaniu pracy równoległej.

3.5. Parallel LINQ

Problem

Mamy do wykonania przetwarzanie równoległe na sekwencji danych, tworzące kolejną sekwencję danych lub podsumowanie tych danych.

Rozwiązanie

Większość programistów jest zaznajomiona z technologią LINQ, którą można wykorzystać do napisania opartych na modelu *pull* obliczeń na sekwencjach. Parallel LINQ (PLINQ) rozszerza to wsparcie LINQ o przetwarzanie równoległe.

PLINQ dobrze sprawdza się w scenariuszach strumieniowych, gdy mamy sekwencję danych wejściowych i generujemy sekwencję danych wyjściowych. Oto prosty przykład, w którym po prostu mnożymy przez dwa każdy element w sekwencji (rzeczywiste scenariusze będą znacznie bardziej wymagające obliczeniowo niż proste mnożenie):

```
static IEnumerable<int> MultiplyBy2(IEnumerable<int> values)
{
```

```
    return values.AsParallel().Select(item => item * 2);
}
```

Ten kod może generować swoje dane wyjściowe w dowolnej kolejności. Jest to ustawienie domyślne dla Parallel LINQ. Można także określić kolejność, która ma być zachowana. Poniższy kod jest nadal przetwarzany równolegle, ale zachowuje oryginalną kolejność:

```
static IEnumerable<int> MultiplyBy2(IEnumerable<int> values)
{
    return values.AsParallel().AsOrdered().Select(item => item * 2);
}
```

Innym naturalnym wykorzystaniem PLINQ jest równoległe agregowanie lub podsumowywanie danych. Poniższy kod wykonuje równoległe sumowanie:

```
static int ParallelSum(IEnumerable<int> values)
{
    return values.AsParallel().Sum();
}
```

Dyskusja

Klasa `Parallel` jest dobra dla wielu scenariuszy, ale kod PLINQ jest prostszy, gdy wykonujemy agregację lub przekształcamy jedną sekwencję w drugą. Należy pamiętać, że klasa `Parallel` jest bardziej przyjazna dla innych procesów w systemie niż PLINQ. Jest to szczególnie istotne, jeżeli przetwarzanie równoległe odbywa się na serwerze.

PLINQ zapewnia równoległe wersje wielu różnych operatorów, w tym filtrowania (`Where`), rzutowania (`Select`) oraz wielu agregacji, takich jak `Sum`, `Average` i bardziej ogólna `Aggregate`. Zasadniczo wszystko, co można zrobić za pomocą zwykłego LINQ, można zrobić równolegle za pomocą PLINQ. To sprawia, że PLINQ jest doskonałym wyborem, jeśli mamy istniejący kod LINQ, który skorzystałby na wykonywaniu równoległym.

Zobacz również

Receptura 3.1 opisuje sposób użycia klasy Parallel do wykonywania kodu dla każdego elementu w sekwencji.

Receptura 9.5 omawia sposób anulowania zapytań PLINQ.

Skorowidz

N

O

P

R

S

T

U

W

Z

Ż

O'REILLY®

Kod współbieżny — responsywność, skalowalność, nowoczesność!

Współbieżność jest jednym z wymogów nowoczesnych aplikacji, niezależnie od ich rodzaju i platformy. Tworzenie współbieżnych aplikacji jest co prawda dość złożone, jednak cecha ta czyni interfejs użytkownika bardziej responsywnym, a systemy — skalowalnymi. Zrozumienie istoty współbieżności i umiejętność jej zaimplementowania jest już koniecznością dla profesjonalnego dewelopera, nawet jeśli zagadnienia te wydają się trudne i skomplikowane.

Ta książka przedstawia nowoczesne podejście do współbieżności. Jest przeznaczona dla programistów, którzy chcą poszerzyć wiedzę i umiejętności. Omówiono tu zagadnienia wielowątkowości i przedstawiono kilka różnych rodzajów współbieżności, w tym programowanie równoległe, asynchroniczne i reaktywne. Opisano biblioteki, dzięki którym programowanie współbieżnych aplikacji staje się znacznie łatwiejsze poprzez podniesienie poziomu abstrakcji. Zawarte w książce receptury uzupełniono o działający kod i przedyskutowano sposób ich działania.

Stephen Cleary — jest doświadczonym programistą. Od samego początku tworzy oprogramowanie *open source*, między innymi bibliotek Boost C++. Opublikował również kilka własnych bibliotek i narzędzi.

W książce między innymi:

- omówienie różnych rodzajów współbieżności
- reaktywność i przepływ zdarzeń
- biblioteka zadań równoległych (TPL)
- biblioteki rozszerzeń reaktywnych w LINQ
- asynchroniczne programowanie obiektowe
- synchronizacja wątków

ISBN 978-83-283-2990-4

9 788328 329904