Параллельное программирование и TPL

Task Parallel Library. Класс Parallel. Класс Task.

[Что такое TPL](#_3znysh7)

[Класс Parallel](#_2et92p0)

[Класс Task](#_3dy6vkm)

[Применение TPL в приложении "Рассыльщик".](#_1t3h5sf)

[Домашнее задание](#_4d34og8)

[Дополнительные материалы](#_2s8eyo1)

[Используемая литература](#_17dp8vu)

# 

# 

# Что такое TPL

Безусловно, что использование класса Thread, для работы с потоками, представляет действенный подход, но не единственный. В .NET Framework существует ещё один мощный способ разработки приложений с использованием многопоточности - это Task Parallel Library (TPL), библиотека параллельных задач, и начиная с четвёртой версии фреймворка, она является основным и предпочтительным способом написания параллельного и многопоточного кода. Одна из причин, которая побудила разработчиков создать данную библиотеку, является желание упростить саму процедуру добавления параллелизма в приложении. Кроме этого, преследовалась цель повышения производительности, а также управление состоянием потока, его отмена и некоторые иные операции.

Основой TPL является концепция задач, которая в свою очередь есть не что иное, как асинхронные операции. Если говорить более абстрактно, то задача похожа на поток. Данный подход предоставляет разработчику определённые преимущества:

* В сравнении с Thread, появляется большая возможность контроля, такая как ожидание, отмена, состояния и иные.
* Ресурсы системы используются более эффективно, автоматическое использование ThreadPool и балансировка нагрузки для повышения производительности.

Библиотека использует пространство имён System.Threading и System.Threading.Tasks, поэтому при использовании необходимо добавить их в проект.

# Класс Parallel

Этот класс обеспечивает параллельное исполнение и значительно упрощает трудоёмкость разработки.

В нём присутствуют некоторые методы, которые будут нам интересны.

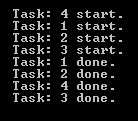
Статический метод Parallel.Invoke, принимающий массив объектов Action, в качестве параметра.

|  |
| --- |
| static void Main()  {  Parallel.Invoke(  ParallelMethod,  ParallelMethod,  ParallelMethod,  () =>  {  Console.WriteLine("Task: {0} start.", Task.CurrentId);  Thread.Sleep(5000);  Console.WriteLine("Task: {0} done.", Task.CurrentId);  });  Console.ReadKey();  }  static void ParallelMethod()  {  Console.WriteLine("Task: {0} start.", Task.CurrentId);  Thread.Sleep(5000);  Console.WriteLine("Task: {0} done.", Task.CurrentId);  } |

Как уже говорилось, данный метод принимает массив объектов Action, в данном случае это четыре объекта, три из них - метод ParallelMethod() и лямбда-выражение, выполняющие одну и туже работу:

|  |
| --- |
| Console.WriteLine("Task: {0} start.", Task.CurrentId);  Thread.Sleep(5000);  Console.WriteLine("Task: {0} done.", Task.CurrentId); |

Запустив приложение, на исполнение запускаются параллельно 4 задачи, далее они приостанавливаются на 5 секунд и происходит вывод сообщения, что они выполнены.



Также класс содержит методы, работающие с коллекциями данных. Данный подход представлен двумя статическими методами - Parallel.For и Parallel.ForEach.

Первый предоставляет возможность выполнять цикл параллельно. Он определяется как:

|  |
| --- |
| ParallelLoopResult For(int, int, Action<int>) |

Здесь принимается начальное значение цикла первым параметром, конечное - вторым, а указатель на метод, выполняющийся каждую итерацию - третьим.

Метод возвращает структуру ParallelLoopResult, представляющую состояния выполнения цикла.

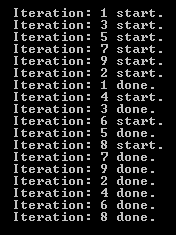
Имеет свойства: IsCompleted типа bool, которая указывает, что все итерации были выполнены, и цикл завершен успешно, а также LowestBreakIteration типа long?, содержащую null, IsCompleted = true или, в противном случае номер последней выполненной итерации.

Пример:

|  |
| --- |
| static void Main()  {  Parallel.For(1, 10, ParallelMethod);  Console.ReadKey();  }  static void ParallelMethod(int iteration)  {  Console.WriteLine("Iteration: {0} start.", iteration);  Thread.Sleep(1000);  Console.WriteLine("Iteration: {0} done.", iteration);  } |

Слегка изменённый предыдущий пример. В параллельном цикле запускается 10 раз ParallelMethod().

На выводе можно наблюдать всю картину работы планировщика, т.е. запуск задач на выполнение не строго по номеру итерации, а согласно работе ThreadPool в бекграунде библиотеки. От запуска к запуску и при увеличении количества итераций картина может существенно меняться.



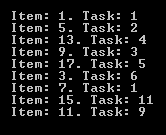
Последний, рассматриваемый нами, метод - Parallel.ForEach. По его названию уже можно предположить, что он является параллельной версией цикла foreach, да так, собственно, и есть.

Имеет определение:

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

Также возвращается структура ParallelLoopResult, а параметрами принимает коллекцию, над которой происходит операция и делегат, вызываемый для каждого элемента этой коллекции.

|  |
| --- |
| static void Main()  {  List<int> collection = new List<int>() {1, 3, 5, 7, 9, 11, 13, 15, 17};  Parallel.ForEach(collection, ParallelMethod);  Console.ReadKey();  }  static void ParallelMethod(int item)  {  Console.WriteLine("Item: {0}. Task: {1}", item, Task.CurrentId);  Thread.Sleep(1000);  } |



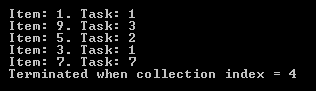
В примере в параллельной манере происходит вывод элементов коллекции и номера задачи, которая обрабатывает элемент, в консоль. Опять же на выводе можно наблюдать работу планировщика.

Как нам известно, можно выйти из цикла, применив оператор break. В случае параллельной итерации эта возможность сохраняется:

|  |
| --- |
| static void Main()  {  List<int> collection = new List<int>() {1, 3, 5, 7, 9, 11, 13, 15, 17};  ParallelLoopResult state = Parallel.ForEach(collection, ParallelMethod);  if (!state.IsCompleted)  Console.WriteLine("Terminated when collection index = {0}", state.LowestBreakIteration);  Console.ReadKey();  }  static void ParallelMethod(int item, ParallelLoopState state)  {  Console.WriteLine("Item: {0}. Task: {1}", item, Task.CurrentId);  if (item == 9)  state.Break();  Thread.Sleep(1000);  } |

Для использования данной возможности изменим ParallelMethod(), добавив в качестве второго принимаемого параметра, тип ParallelLoopResult и далее в методе при item == 9, будем вызывать метод Break() данного типа.

В основном методе возвращаемое значение Parallel.ForEach сохраняем в переменной state и выводим сообщение в случае преждевременного выхода из цикла.



Здесь также можно увидеть, что цикл прерывается нечетко после вызова **Break()** и какое-то время обработка коллекции продолжается.

# 

# 

# Класс Task

Этот класс обеспечивает исполнение самостоятельной задачи в отдельном потоке. В самом простом случае, его использование подобно использованию Thread, несколько упрощенному:

|  |
| --- |
| static void Main()  {  Task task = new Task(TaskMethod);  task.Start();  Console.WriteLine("Ждем окончания работы задачи.");  task.Wait();  Console.ReadKey();  }  static void TaskMethod()  {  Console.WriteLine("Задача {0} выполняется.", Task.CurrentId);  Thread.Sleep(2000);  Console.WriteLine("Задача {0} завершена.", Task.CurrentId);  } |



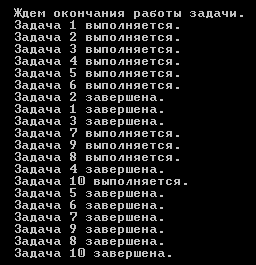
Все достаточно легко - в конструктор Task, передается делегат типа Action и запускаем задачу, с помощью task.Start(). Вызывая task.Wait() мы гарантированно дождемся окончания работы фонового потока.

Как и в случае с потоками, возможно запустить массив задач. Для этой цели воспользуемся статическим методом Task.Factory.StartNew():

|  |
| --- |
| static void Main()  {  Task[] tasksArr = new Task[10];  for (int i = 0; i < tasksArr.Length; i++)  {  tasksArr[i] = Task.Factory.StartNew(TaskMethod);  }  Console.WriteLine("Ждем окончания работы задачи.");  Task.WaitAll(tasksArr);  Console.ReadKey();  }  static void TaskMethod()  {  Console.WriteLine($"Задача {Task.CurrentId} выполняется.");  Thread.Sleep(2000);  Console.WriteLine($"Задача {Task.CurrentId} завершена.");  } |

Массив tasks, состоящий из 10 элементов, задачи в нем запускаются посредством Factory.StartNew().

Важнейшим моментом является статический метод - WaitAll(), класса Task, именно он позволяет ожидать завершения всего массива задач.



Разберёмся как запускать задачи, которые принимают делегат, указывающий на методы с параметрами:

|  |
| --- |
| static void Main()  {  int timeSpan = 2000;  Task task1 = new Task(() => TaskMethodWithParameters(timeSpan));  task1.Start();  Task.Factory.StartNew(() => TaskMethodWithParameters(timeSpan));  Console.WriteLine("Ждем окончания работы задачи.");  Console.ReadKey();  }  static void TaskMethodWithParameters(int timeSpan)  {  Console.WriteLine($"Задача {Task.CurrentId} выполняется.");  Console.WriteLine($"Значение timeSpan = {timeSpan} ms.");  Thread.Sleep(timeSpan);  Console.WriteLine($"Задача {Task.CurrentId} завершена.");  } |

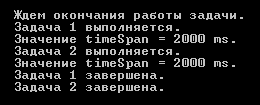
Переменная timeSpan содержит величину задержки в миллисекундах, а наш метод изменен, чтоб была возможность принять задержку в качестве параметра. Далее этот метод запускается двумя различными способами:

|  |
| --- |
| Task task1 = new Task(() => TaskMethodWithParameters(timeSpan));  task1.Start(); |

и

|  |
| --- |
| Task.Factory.StartNew(() => TaskMethodWithParameters(timeSpan)); |

Второй способ заметно короче при одинаковой работе, что использовать - как дело вкуса, так и зависит от конкретной задачи.



Помимо передачи параметров, в большинстве задач было бы неплохо и получать результаты их работы. Для этого при объявлении переменной типа Task её необходимо типизировать. Задача task1 возвращает int, задача task2 - string, соответственно необходимо выполнять метод, возвращающий данный тип.

Свойство Result типа Task и будет представлять результат работы задачи.

|  |
| --- |
| static void Main()  {  int x = 3;  int y = 2;  string message = "тестовое сообщение";  Console.WriteLine("Запуск задач.");  Task<int> task1 = new Task<int>(() => TaskMethodAdd(x, y));  task1.Start();  Task<string> task2 = Task.Factory.StartNew(() => TaskMethodStringToUpper(message));  int resultAdd = task1.Result;  string resultStringToUpper = task2.Result;  Console.WriteLine($"Результат TaskMethodAdd: {resultAdd}");  Console.WriteLine($"Результат TaskMethodString: {resultStringToUpper}");  Console.ReadKey();  }  static int TaskMethodAdd(int x, int y)  {  Thread.Sleep(2000);  return x + y;  }  static string TaskMethodStringToUpper(string message)  {  Thread.Sleep(2000);  return message.ToUpper();  } |



Также ещё одной очень удобной функцией является возможность задавать продолжение задачи, т.е. описывать методы, исполняемые после завершения текущей. Данный подход возможен с помощью метода ContinueWith().

Сигнатура метода:

|  |
| --- |
| Task ContinueWith(Action<Task> continuation) |

Где continuation - задача, запускаемая, после текущей.

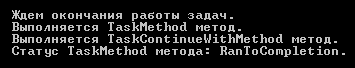
|  |
| --- |
| static void Main()  {  Task task = new Task(TaskMethod);  task.ContinueWith(TaskContinueWithMethod);  task.Start();  Console.WriteLine("Ждем окончания работы задач.");  Console.ReadKey();  }  static void TaskMethod()  {  Thread.Sleep(2000);  Console.WriteLine("Выполняется TaskMethod метод.");  }  static void TaskContinueWithMethod(Task task)  {  Thread.Sleep(2000);  Console.WriteLine("Выполняется TaskContinueWithMethod метод.");  } |



Здесь за выполнением task следует вызов задачи с TaskContinueWithMethod. Данных подход позволяет строить цепочки вызовов, очень сходными с асинхронными вызовами.

Из примера видно, что в метод, используемый в ContinueWith, должен принимать параметр типа Task, благодаря этому возможно получить доступ к свойствам предыдущей задачи:

|  |
| --- |
| static void Main()  {  int x = 3;  int y = 2;  Task task = new Task(() => TaskMethod(x, y));  task.ContinueWith(TaskContinueWithMethod);  task.Start();  Console.WriteLine(" Ждем окончания работы задач.");  Console.ReadKey();  }  static int TaskMethod(int x, int y)  {  Console.WriteLine("Выполняется TaskMethod метод.");  Thread.Sleep(2000);  return x + y;  }  static void TaskContinueWithMethod(Task task)  {  var taskStatus = task.Status;  Console.WriteLine("Выполняется TaskContinueWithMethod метод.");  Console.WriteLine($"Статус TaskMethod метода: {taskStatus}.");  } |



В примере, мы получаем на выводе статус предыдущей задачи.

# 

# 

# Применение TPL в приложении "Рассыльщик".

Как и в случае с потоками, задачей распараллеливания является отправка писем.

Подойдем к решению этой задачи с помощью двух подходов - при помощи Task:

|  |
| --- |
| class EmailSendServiceClass  {  private string strLogin; // email, c которого будет рассылаться почта  private string strPassword; // пароль к email, с которого будет рассылаться почта  private string strSmtp = "smtp.yandex.ru"; // smtp - server  private int iSmtpPort = 25; // порт для smtp-server  private string strBody; // текст письма для отправки  private string strSubject; // тема письма для отправки  public EmailSendServiceClass(string sLogin, string sPassword)  {  strLogin = sLogin;  strPassword = sPassword;  }  private void SendMail(Emails2 email)  {  string mail = email.Email;  string name = email.Name;  using (MailMessage mm = new MailMessage(strLogin, mail))  {  mm.Subject = strSubject;  mm.Body = "Hello world!";  mm.IsBodyHtml = false;  SmtpClient sc = new SmtpClient(strSmtp, iSmtpPort);  sc.EnableSsl = true;  sc.DeliveryMethod = SmtpDeliveryMethod.Network;  sc.UseDefaultCredentials = false;  sc.Credentials = new NetworkCredential(strLogin, strPassword);  try  {  sc.Send(mm);  }  catch (Exception ex)  {  MessageBox.Show("Невозможно отправить письмо " + ex);  }  }  }  public void SendMails(IQueryable<Emails2> emails)  {  foreach (Emails2 email in emails)  {  Task.Factory.StartNew(() => SendMail(email));  }  }  } |

А также при помощи Parallel:

|  |
| --- |
| ...  public void SendMails(IQueryable<Emails2> emails)  {  Parallel.ForEach(emails, SendMail);  }  ... |

# Домашнее задание

1. Даны 2 двумерных матрицы. Размерность 100х100 каждая. Напишите приложение, производящее параллельное умножение матриц. Матрицы заполняются случайными целыми числами от 0 до10.
2. \*В некой директории лежат файлы. По структуре они содержат 3 числа, разделенные пробелами. Первое число - целое, обозначает действие, 1- умножение и 2- деление, остальные два - числа с плавающей точкой. Написать многопоточное приложение, выполняющее выполняющее вышеуказанные действия над числами и сохраняющими результат в файл result.dat. Количество файлов в директории заведомо велико.

# Дополнительные материалы

1. https://msdn.microsoft.com/ru-ru/default.aspx
2. https://metanit.com

# Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. Рихтер Дж. “CLR via C#”
2. Эндрю Троелсен - “ЯЗЫК ПРОГРАММИРОВАНИЯ C#5.0 И ПЛАТФОРМА .NET 4.5”