Потоки в C#

План

- **Класс Thread**
- **Создание потоков**
- **Разделяемые ресурсы**
- **Класс AutoResetEvent**
- **Пул потоков**
- **Библиотека параллельных задач TPL**

Класс Thread

Основной функционал для использования потоков в приложении сосредоточен в пространстве имен System.Threading.

класс, представляющий отдельный поток - класс Thread.

Основные свойства класса:

Статическое свойство CurrentContext позволяет получить контекст, в котором выполняется поток
Статическое свойство CurrentThread возвращает ссылку на выполняемый поток
Свойство IsAlive указывает, работает ли поток в текущий момент
Свойство IsBackground указывает, является ли поток фоновым
Свойство Name содержит имя потока
Свойство Priority хранит приоритет потока - значение перечисления ThreadPriority
Свойство ThreadState возвращает состояние потока - одно из значений перечисления ThreadState

Некоторые методы класса Thread:

Статический метод GetDomain возвращает ссылку на домен приложения
Статический метод GetDomainId возвращает id домена приложения, в котором выполняется текущий поток
Статический метод Sleep останавливает поток на определенное количество миллисекунд
Метод Abort уведомляет среду CLR о том, что надо прекратить поток, однако прекращение работы потока происходит не сразу, а только тогда, когда это становится возможно. Для проверки завершенности потока следует опрашивать его свойство ThreadState

завершится поток, для которого был вызван данный метод
Метод Resume возобновляет работу ранее приостановленного потока
Метод Start запускает поток
Метод Suspend приостанавливает поток

Метод Interrupt прерывает поток на некоторое время
Метод Join блокирует выполнение вызвавшего его потока до тех пор, пока не завершится поток, для которого был вызван данный метод
Метод Resume возобновляет работу ранее приостановленного потока
Метод Start запускает поток
Метод Suspend приостанавливает поток

Статус потока

Статусы потока содержатся в перечислении **ThreadState**:

Aborted: поток остановлен, но пока еще окончательно не завершен
AbortRequested: для потока вызван метод Abort, но остановка потока еще не произошла
Background: поток выполняется в фоновом режиме
Running: поток запущен и работает (не приостановлен)

SuspendRequested: поток получил запрос на приостановку
Unstarted: поток еще не был запущен
WaitSleepJoin: поток заблокирован в результате действия методов Sleep или Join
Stopped: поток завершен
StopRequested: поток получил запрос на остановку
Suspended: поток приостановлен
SuspendRequested: поток получил запрос на приостановку
Unstarted: поток еще не был запущен
WaitSleepJoin: поток заблокирован в результате действия методов Sleep или Join

Приоритеты потоков

Приоритеты потоков располагаются в перечислении **ThreadPriority**:

Lowest
BelowNormal
Normal
AboveNormal
Highest

Создание потоков. Делегат ThreadStart

Для запуска нового потока нам надо определить задачу в приложении, которую будет выполнять данный поток. Для этого мы можем добавить новый метод, производящий какие-либо действия.

Для создания нового потока используется делегат ThreadStart, который получает в качестве параметра метод, который должен выполниться в этом потоке. И чтобы запустить поток, вызывается метод Start.

```
using System.Threading;
class Program
{
    static void Main(string[] args)
    {
        // создаем новый поток
        Thread myThread = new Thread(new
ThreadStart(Count));
        myThread.Start(); // запускаем поток
         for (int i = 1; i < 9; i++)
        {
            Console.WriteLine("Главный поток:");
            Console.WriteLine(i * i);
            Thread.Sleep(300);
        }
         Console.ReadLine();
    }
```

```
        ThreadStart(Count));
            myThread.Start(); // запускаем поток
            for (int i = 1; i < 9; i++)
            {
                Console.WriteLine("Главный
поток:");
                Console.WriteLine(i * i);
                Thread.Sleep(300);
            }
            Console.ReadLine();
        }
```

```
                {
                    Console.WriteLine("Второй поток:");
                    Console.WriteLine(i * i);
                    Thread.Sleep(400);
                }
public static void Count()
    {           }
        for (int i = 1; i < 9; i++)
        {
            Console.WriteLine("Второй поток:");
            Console.WriteLine(i * i);
            Thread.Sleep(400);
        }
    }
}
```

Чтобы запустить этот метод в качестве второго потока, мы сначала создаем объект потока: Thread myThread = new Thread(new ThreadStart(Count));. В конструктор передается делегат ThreadStart, который в качестве параметра принимает метод Count. И следующей строкой myThread.Start() мы запускаем поток. После этого управление передается главному потоку, и выполняются все остальные действия, определенные в методе Main.

Здесь новый поток будет производить действия, определенные в методе Count. В данном случае это возведение в квадрат числа и вывод его на экран. И после каждого умножения с помощью метода Thread.Sleep мы усыпляем поток на 400 миллисекунд. Чтобы запустить этот метод в качестве второго потока, мы сначала создаем объект потока: Thread myThread = new Thread(new ThreadStart(Count));. В конструктор передается делегат ThreadStart, который в качестве параметра принимает метод Count. И следующей строкой myThread.Start() мы запускаем поток. После этого управление передается главному потоку, и выполняются все остальные действия, определенные в методе Main.

Хотя в данном случае явным образом мы не используем делегат ThreadStart, но неявно он создается. Компилятор C# выводит делегат из сигнатуры метода Count и вызывает соответствующий конструктор.

еще одна форма создания потока: Thread myThread = new Thread(Count);

Хотя в данном случае явным образом мы не используем делегат ThreadStart, но неявно он создается. Компилятор C# выводит делегат из сигнатуры метода Count и вызывает соответствующий конструктор.

Если нам надо передать какие-нибудь параметры в поток

Для этой цели используется делегат **ParameterizedThreadStart**. Его действие похоже на функциональность делегата ThreadStart.

```
ParameterizedThreadStart(Count));
                myThread.Start(number);
                 for (int i = 1; i < 9; i++)       {
                    Console.WriteLine("Главный
поток:");
class Program{          Console.WriteLine(i * i);
    static void Main(string[] args)    {
                    Thread.Sleep(300);
        int number = 4;
                }
        // создаем новый поток
                Console.ReadLine();
        Thread myThread = new Thread(new
              {
ParameterizedThreadStart(Count));
        myThread.Start(number);
         for (int i = 1; i < 9; i++)       {
            Console.WriteLine("Главный поток:");
            Console.WriteLine(i * i);
            Thread.Sleep(300);
        }
        Console.ReadLine();
    }
```

```
                for (int i = 1; i < 9; i++)       {
                    int n = (int)x;
                     Console.WriteLine("Второй поток:");
                    Console.WriteLine(i*n);
public static void Thread.Sleep(400); {     }    }}
```

```
Count(object x)
        for (int i = 1; i < 9; i++)       {
            int n = (int)x;
             Console.WriteLine("Второй поток:");
            Console.WriteLine(i*n);
            Thread.Sleep(400);       }    }}
```

ограничение: мы можем запускать во втором потоке только такой метод, который в качестве единственного параметра принимает объект типа object. Поэтому в данном случае нам надо дополнительно **привести** переданное значение к типу int, чтобы его использовать в вычислениях.

Передача через лямбда

```
static void Main(string[] args) {
Thread thread1 = new Thread(() => Do(6, "Hello"));
thread1.Start();
}
public static void Do(int i, string s) {
Console.WriteLine("'{0}, {1}' done!", i, s);
}
```

Несколько параметров различного типа

Классовый подход:

```
class Program
{
    static void Main(string[] args)
    {
        Counter counter = new Counter();
        counter.x = 4;
        counter.y = 5;

        Thread myThread = new Thread(new
ParameterizedThreadStart(Count));
        myThread.Start(counter);

        //.....
    }
```

```
                Console.WriteLine("Второй поток:");
                Console.WriteLine(i*c.x *c.y);
            }   }}

        public class Counter
public static void Count(object obj)    {
        {
        for (int i = 1; i < 9; i++)       {
            public int x;
            Counter c = (Counter)obj;
            public int y;
        }
            Console.WriteLine("Второй поток:");
            Console.WriteLine(i*c.x *c.y);
        }   }}

public class Counter
{
    public int x;
    public int y;
}
```

нужному нам типу.

Для решения данной проблемы рекомендуется объявлять все используемые методы и переменные в специальном классе, а в основной программе запускать поток через ThreadStart.

ограничение:

метод Thread.Start не является типобезопасным, то есть мы можем передать в него любой тип, и потом нам придется приводить переданный объект к нужному нам типу.

Например:

Для решения данной проблемы рекомендуется объявлять все используемые методы и переменные в специальном классе, а в основной программе запускать поток через ThreadStart.

Например:

```
                Counter counter = new Counter(5, 4);

                Thread myThread = new Thread(new
        ThreadStart(counter.Count));
class Program   myThread.Start();
{               //.....................
    static void Main(string[] args)
    {       }
        Counter counter = new Counter(5, 4);

        Thread myThread = new Thread(new
ThreadStart(counter.Count));
        myThread.Start();
        //.....................
    }
}
```

```
public class Counter
{
    private int x;
    private int y;

    public Counter(int _x, int _y)
    {
        this.x = _x;
        this.y = _y;
    }
```

```
    public Counter(int _x, int _y)
    {
        this.x = _x;
        this.y = _y;
    }

    public void Count()
    {
        for (int i = 1; i < 9;
i++)
        {
            Console.WriteLine
("Второй поток:");
            Console.WriteLine
(i * x * y);
            Thread.Sleep(400)
;
        }
    }}
```

Разделяемые ресурсы

Ресурсы, общие для всей программы. Это могут быть общие переменные, файлы, другие ресурсы.

```
class Program {
    static int x=0;
    static void Main(string[] args)     {
        for (int i = 0; i < 5; i++)         {
            Thread myThread = new Thread(Count);
            myThread.Name = "Поток " + i.ToString();
            myThread.Start();
        }

        Console.ReadLine();
    }
```

```
            Thread.CurrentThread.Name, x);
                x++;
                Thread.Sleep(100);
            }
        }
public static void Count()
    {
        x = 1;
        for (int i = 1; i < 9; i++)
        {
            Console.WriteLine("{0}: {1}",
Thread.CurrentThread.Name, x);
            x++;
            Thread.Sleep(100);
        }
    }}
```

в процессе работы будет происходить переключение между потоками, и значение переменной x становится непредсказуемым.

Для этого используется ключевое слово lock .

Оператор lock определяет блок кода, внутри которого весь код блокируется и становится недоступным для других потоков до завершения работы текущего потока

Решение проблемы состоит в том, чтобы синхронизировать потоки и ограничить доступ к разделяемым ресурсам на время их использования каким-нибудь потоком. Для этого используется ключевое слово lock .

Оператор lock определяет блок кода, внутри которого весь код блокируется и становится недоступным для других потоков до завершения работы текущего потока

```
                {
                    Thread myThread = new
    Thread(Count);
                    myThread.Name = "Поток " +
    i.ToString();
class Program {
                    myThread.Start();
    static int x=0;
                }
    static object locker = new object();
    static void Main(string[] args)
                Console.ReadLine();
    {
            }
        for (int i = 0; i < 5; i++)
        {
            Thread myThread = new Thread(Count);
            myThread.Name = "Поток " + i.ToString();
            myThread.Start();
        }

        Console.ReadLine();
    }
```

```
                for (int i = 1; i < 9; i++)
                {
                    Console.WriteLine("{0}: {1}",
Thread.CurrentThread.Name, x);
                    x++;
                    Thread.Sleep(100);
                }
            }
        }
    }
public static void Count()
        lock (locker)
        {
            x = 1;
            for (int i = 1; i < 9; i++)
            {
                Console.WriteLine("{0}: {1}",
Thread.CurrentThread.Name, x);
                x++;
                Thread.Sleep(100);
            }
        }
    }
```

lock, объект locker блокируется, и на время его блокировки монопольный доступ к блоку кода имеет только один поток.

После окончания работы блока кода, объект locker освобождается и становится доступным для других потоков.

Для блокировки с ключевым словом lock используется объект-заглушка, в данном случае это переменная locker.

Когда выполнение доходит до оператора lock, объект locker блокируется, и на время его блокировки монопольный доступ к блоку кода имеет только один поток.

После окончания работы блока кода, объект locker освобождается и становится доступным для других потоков.

использовать мониторы, представленные классом **System.Threading.Monitor**.

Фактически конструкция оператора lock из прошлой темы инкапсулирует в себе синтаксис использования мониторов.

Наряду с оператором lock для синхронизации потоков мы можем использовать мониторы, представленные классом **System.Threading.Monitor**.

Фактически конструкция оператора lock из прошлой темы инкапсулирует в себе синтаксис использования мониторов.

```
                {
                    Thread myThread = new
            Thread(Count);
                    myThread.Name = "Поток " +
            i.ToString();
                    myThread.Start();
                }
                Console.ReadLine();
            }
class Program
{
    static int x=0;
    static object locker = new object();
    static void Main(string[] args)
    {
        for (int i = 0; i < 5; i++)
        {
            Thread myThread = new Thread(Count);
            myThread.Name = "Поток " + i.ToString();
            myThread.Start();
        }

        Console.ReadLine();
    }
```

```
                Console.WriteLine("{0}: {1}",
            Thread.CurrentThread.Name, x);
                    x++;
                    Thread.Sleep(100);
                }       }
public static void Count(){   {   finally
        try         {   Monitor.Exit(locker);
            Monitor.Enter(locker);   }
            x = 1;
            for (int i = 1; i < 9; i++)          {
                Console.WriteLine("{0}: {1}",
Thread.CurrentThread.Name, x);
                x++;
                Thread.Sleep(100);
            }       }
        finally         {
            Monitor.Exit(locker);
        }
    }
}
```

lock.

А в блоке try...finally с помощью метода Monitor.Exit происходит освобождение объекта locker, и он становится доступным для других потоков

Метод Monitor.Enter блокирует объект locker так же, как это делает оператор lock.

А в блоке try...finally с помощью метода Monitor.Exit происходит освобождение объекта locker, и он становится доступным для других потоков

- еще ряд методов, которые позволяют управлять синхронизацией потоков. Так, метод Monitor.Wait освобождает блокировку объекта и переводит поток в очередь ожидания объекта. Следующий поток в очереди готовности объекта блокирует данный объект. А все потоки, которые вызвали метод Wait, остаются в очереди ожидания, пока не получат сигнала от метода Monitor.Pulse или Monitor.PulseAll, посланного владельцем блокировки. Если метод Monitor.Pulse отправлен, поток, находящийся во главе очереди ожидания, получает сигнал и блокирует освободившийся объект. Если же метод Monitor.PulseAll отправлен, то все потоки, находящиеся в очереди ожидания, получают сигнал и переходят в очередь готовности, где им снова разрешается получать блокировку объекта.

очереди ожидания, пока не получат сигнала от метода Monitor.Pulse или Monitor.PulseAll, посланного владельцем блокировки. Если метод Monitor.Pulse отправлен, поток, находящийся во главе очереди ожидания, получает сигнал и блокирует освободившийся объект. Если же метод Monitor.PulseAll отправлен, то все потоки, находящиеся в очереди ожидания, получают сигнал и переходят в очередь готовности, где им снова разрешается получать блокировку объекта.

Класс AutoResetEvent

Класс является оберткой над объектом ОС Windows "событие" и позволяет переключить данный объект-событие из сигнального в несигнальное состояние

```
                for (int i = 0; i < 5; i++)       {
                    Thread myThread = new
            Thread(Count);
                    myThread.Name = "Поток " +
            i.ToString();
                    myThread.Start();
                }
class Program{
    static AutoResetEvent waitHandler = new
AutoResetEvent(true);
    static int x=0;
                    Console.ReadLine();
    static void Main(string[] args)    {
        for (int i = 0; i < 5; i++)       {
            Thread myThread = new Thread(Count);
            myThread.Name = "Поток " + i.ToString();
            myThread.Start();
        }

        Console.ReadLine();
    }
```

```
                {
                    Console.WriteLine("{0}: {1}",
    Thread.CurrentThread.Name, x);
                    x++;
                    Thread.Sleep(100);
                }
public static void Count()
    {
        waitHandler.WaitOne();
        x = 1;
        for (int i = 1; i < 9; i++)
        {
            Console.WriteLine("{0}: {1}",
Thread.CurrentThread.Name, x);
            x++;
            Thread.Sleep(100);
        }
        waitHandler.Set();
    }
}
```

Когда начинает работать поток, то первым делом срабатывает определенный в методе Count вызов waitHandler.WaitOne().

Метод WaitOne указывает, что текущий поток переводится в состояние ожидания, пока объект waitHandler не будет переведен в сигнальное состояние. И так все потоки у нас переводятся в состояние ожидания.

Во-первых, создаем переменную типа AutoResetEvent. Передаем в конструктор значение true, мы тем самым указываем, что создаваемый объект изначально будет в сигнальном состоянии.

Когда начинает работать поток, то первым делом срабатывает определенный в методе Count вызов waitHandler.WaitOne().

Метод WaitOne указывает, что текущий поток переводится в состояние ожидания, пока объект waitHandler не будет переведен в сигнальное состояние. И так все потоки у нас переводятся в состояние ожидания.

уведомляет все ожидающие потоки, что объект waitHandler снова находится в сигнальном состоянии, и один из потоков "захватывает" данный объект, переводит в несигнальное состояние и выполняет свой код. А остальные потоки снова ожидают.

После завершения работы метод Set уведомляет все ожидающие потоки, что объект waitHandler снова находится в сигнальном состоянии, и один из потоков "захватывает" данный объект, переводит в несигнальное состояние и выполняет свой код. А остальные потоки снова ожидают.

код.

Но если бы мы написали AutoResetEvent waitHandler = new AutoResetEvent(false), тогда объект изначально был бы в несигнальном состоянии, а поскольку все потоки блокируются методом waitHandler.WaitOne() до ожидания сигнала, то у нас попросту случилась бы блокировка программы, и программа не выполняла бы никаких действий.

Так как в конструкторе AutoResetEvent мы указываем, что объект изначально не находится в сигнальном состоянии, то первый из очереди потоков захватывает данный объект и начинает выполнять свой код.

Но если бы мы написали AutoResetEvent waitHandler = new AutoResetEvent(false), тогда объект изначально был бы в несигнальном состоянии, а поскольку все потоки блокируются методом waitHandler.WaitOne() до ожидания сигнала, то у нас попросту случилась бы блокировка программы, и программа не выполняла бы никаких действий.

несколько объектов AutoResetEvent, то мы можем использовать для отслеживания состояния этих объектов методы и , которые в качестве параметра принимают массив объектов класса WaitHandle - базового класса для AutoResetEvent.

Если у нас в программе используется несколько объектов AutoResetEvent, то мы можем использовать для отслеживания состояния этих объектов методы и , которые в качестве параметра принимают массив объектов класса WaitHandle - базового класса для AutoResetEvent.

Пул потоков

класс **ThreadPool**, который по мере необходимости уменьшает и увеличивает количество потоков в пуле до максимально допустимого.

Значение максимально допустимого количества потоков в пуле может изменяться. В случае двуядерного ЦП оно по умолчанию составляет 1023 рабочих потоков и 1000 потоков ввода-вывода.

nWorkerThreads, out nCompletionThreads);

Console.WriteLine("Максимальное количество потоков: " + nWorkerThreads + "\nПотоков ввода-вывода доступно: " + nCompletionThreads);

for (int i = 0; i < 5; i++)
ThreadPool.QueueUserWorkItem(JobForAThread); Thread.Sleep(3000);
Console.ReadLine(); }

```
static void Main() {
int nWorkerThreads;
int nCompletionThreads;
ThreadPool.GetMaxThreads(out nWorkerThreads, out
nCompletionThreads);

Console.WriteLine("Максимальное количество потоков: "
+ nWorkerThreads + "\nПотоков ввода-вывода доступно: "
+ nCompletionThreads);

for (int i = 0; i < 5; i++)
ThreadPool.QueueUserWorkItem(JobForAThread);
Thread.Sleep(3000); Console.ReadLine(); }
```

```
        Console.WriteLine("цикл {0}, выполнение
        внутри потока из пула {1}", i,
        Thread.CurrentThread.ManagedThreadId);
        Thread.Sleep(50);
    } }
static void JobForAThread(object state) {
for (int i = 0; i < 3; i++) {

Console.WriteLine("цикл {0}, выполнение внутри потока
из пула {1}", i, Thread.CurrentThread.ManagedThreadId);
Thread.Sleep(50);

} }
```

Библиотека параллельных задач TPL

библиотека параллельных задач TPL (Task Parallel Library), основной функционал которой располагается в пространстве имен System.Threading.Tasks.

Задачи и класс Task

В библиотеке классов .NET задача представлена специальным классом - классом Task, который находится в пространстве имен System.Threading.Tasks. Данный класс описывает отдельную задачу, которая запускается асинхронно в одном из потоков из пула потоков. Хотя ее также можно запускать синхронно в текущем потоке.

task.Start();

В качестве параметра объект Task принимает делегат Action, то есть мы можем передать любое действие, которое соответствует данному делегату, например, лямбда-выражение, как в данном случае, или ссылку на какой-либо метод.

Первый способ создание объекта Task и вызов у него метода Start:

```
Task task = new Task(() => Console.WriteLine("Hello Task!"));

task.Start();
```

В качестве параметра объект Task принимает делегат Action, то есть мы можем передать любое действие, которое соответствует данному делегату, например, лямбда-выражение, как в данном случае, или ссылку на какой-либо метод.

в качестве параметра принимает делегат Action, который указывает, какое действие будет выполняться. При этом этот метод сразу же запускает задачу:

```
Task task = Task.Factory.StartNew(() =>
Console.WriteLine("Hello Task!"));
```

Второй способ заключается в использовании статического метода Task.Factory.StartNew(). Этот метод также в качестве параметра принимает делегат Action, который указывает, какое действие будет выполняться. При этом этот метод сразу же запускает задачу:

```
Task task = Task.Factory.StartNew(() =>
Console.WriteLine("Hello Task!"));
```

```
Task task = Task.Run(() =>
Console.WriteLine("Hello Task!"));
```

Метод Task.Run() также в качестве параметра может принимать делегат Action - выполняемое действие и возвращает объект Task.

Третий способ определения и запуска задач представляет использование статического метода Task.Run():

```
Task task = Task.Run(() => Console.WriteLine("Hello Task!"));
```

Метод Task.Run() также в качестве параметра может принимать делегат Action - выполняемое действие и возвращает объект Task.

```
static void Main(string[] args) {
Task task = new Task(Display);
task.Start();
task.Wait();
Console.WriteLine("Завершение метода Main"); Console.ReadLine(); }
```

Чтобы указать, что метод Main должен подождать до конца выполнения задачи, нам надо использовать метод Wait:

```
static void Main(string[] args) {
Task task = new Task(Display);
task.Start();
task.Wait();
Console.WriteLine("Завершение метода Main");
Console.ReadLine(); }
```

Свойства класса Task

Класс Task имеет ряд свойств, с помощью которых мы можем получить информацию об объекте. Некоторые из них:

AsyncState: возвращает объект состояния задачи

CurrentId: возвращает идентификатор текущей задачи

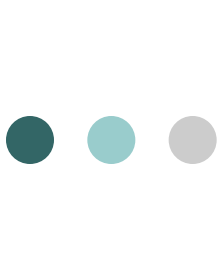
Exception: возвращает объект исключения, возникшего при выполнении задачи

Status: возвращает статус задачи

Вложенные задачи

Одна задача может запускать другую - вложенную задачу. При этом эти задачи выполняются независимо друг от друга. Например:

```
{ Console.WriteLine("Inner task
starting... ");
static void Main(string[] args) {
Thread.Sleep(2000);
var outer = Task.Factory.StartNew(() => // внешняя задача
Console.WriteLine("Inner task
{ Console.WriteLine(" Outer task starting... ");
finished.");
var inner = Task.Factory.StartNew(() => // вложенная
});
задача
});
{ Console.WriteLine("Inner task starting...");
outer.Wait(); // ожидаем выполнения
Thread.Sleep(2000);
внешней задачи Console.WriteLine("End of
Console.WriteLine("Inner task finished.");
Main"); Console.ReadLine(); }
});
});
outer.Wait(); // ожидаем выполнения внешней задачи
Console.WriteLine("End of Main"); Console.ReadLine(); }
```



Литература

https://metanit.com/sharp/tutorial/

https://professorweb.ru/my/csharp/thread_and_files/1/1_16.php