GeekBrains

ASP.NET MVC и углубление изучения C#

Углубление в мультипоточность

На этом уроке

1. Познакомимся с новыми структурами
2. Поймем, как работает планировщик задач в операционной среде
3. Узнаем паттерн пул объектов
4. Разберем дополнительные виды синхронизации потоков
5. Узнаем о Threadpool

Оглавление

На этом уроке

Важные структуры для использования в мультипоточном коде

Структуры с приставкой Concurrent

Как выполняются потоки

Не lock’ом едины, дополнительные методы синхронизации

Рекомендации

Пул объектов

Threadpool, он же тредпул

Домашнее задание

Дополнительные материалы

Используемые источники

Важные структуры для использования в мультипоточном коде

В библиотеках .Net есть множество структур, которые подходят для разных случаев. Вам уже должны быть известны такие, как List<T>, Dictionary<TKey, TValue>, IReadOnlyCollection<T>, IReadOnlyDictionary<TKey, TValue>, ImmutableList<T>, ImmutableDictionary<TKey,TValue> и т. д.

Все они идеальны для использования в однопоточном приложении. Относительно недавно их также использовали в мультипоточном коде, но грамотно синхронизировали.

Но времена идут и библиотеки пополняются дополнительными удобными структурами, которые заменяют старые подходы к написанию кода. Так же и к этим структурам, которые в своё время были не потокобезопасными, добавили их потокобезопасные вариации.

Структуры с приставкой Concurrent

Для подключения этих структур нужно добавить новое пространство имен “System.Collections.Concurrent”.

Одним из самых часто используемых словарей в мультипоточном приложении является “ConcurrentDictionary<TKey, TValue>”. Он похож на обычный словарь, но название методов отличается:

```
using System;
using System.Collections.Concurrent;
using System.Collections.Generic;
using System.Linq;
using System.Threading;

namespace Interview.Sample
{
      class Program
      {
      private static object lockObjectOne = new object();

      static void Main(string[] args)
      {
            ConcurrentDictionary<int, string> threadSafetyDictionary =
new ConcurrentDictionary<int, string>();

            bool addedFoo = threadSafetyDictionary.TryAdd(1, "foo");

            if (addedFoo)
            {
              Console.WriteLine("Foo has been added!");
            }

            bool addedBar = threadSafetyDictionary.TryAdd(2, "bar");

            if (addedBar)
            {
            Console.WriteLine("Foo has been added!");
            }

            bool removed = threadSafetyDictionary.TryRemove(2, out string
foo);

            if (removed)
            {
            Console.WriteLine($"Has been deleted index 2 with value
{foo}");
            }
      }
      }
}
```

Такие структуры активно используют подход “Try…”, который информирует об успешности выполнения операции. Эти методы скрывают под собой сложный и хорошо спроектированный код, который не всегда понятен новичку.

Они потребляют больше ресурсов, чем обычные коллекции. К примеру, в них точно организовано методы общего назначения подсчета элементов, индикация пустой структуры, получение энумератора и т. д. при помощи блокировок. А значит, слишком активный вызов этих методов будет сказываться на производительности всего приложения.

Поэтому разработчики стараются по максимуму избежать блокировки.

Когда что использовать:

Ситуация	**Структура**
Однопоточное приложение, запись	Классические коллекции
Однопоточное приложение, чтение	Классические коллекции
Многопоточное приложение, запись	Коллекции Concurrent
Многопоточное приложение, чтение	Immutable коллекции (больше потребления памяти) либо Concurrent (больше потребления производительности)

Вам может показаться, что классические коллекции вполне подходят для чтения из разных потоков. И в большинстве случаях это будет работать. Но если элементы у элементов классической коллекции есть свойства с “lazy”, с инициализацией возникнут проблемы.

Как выполняются потоки

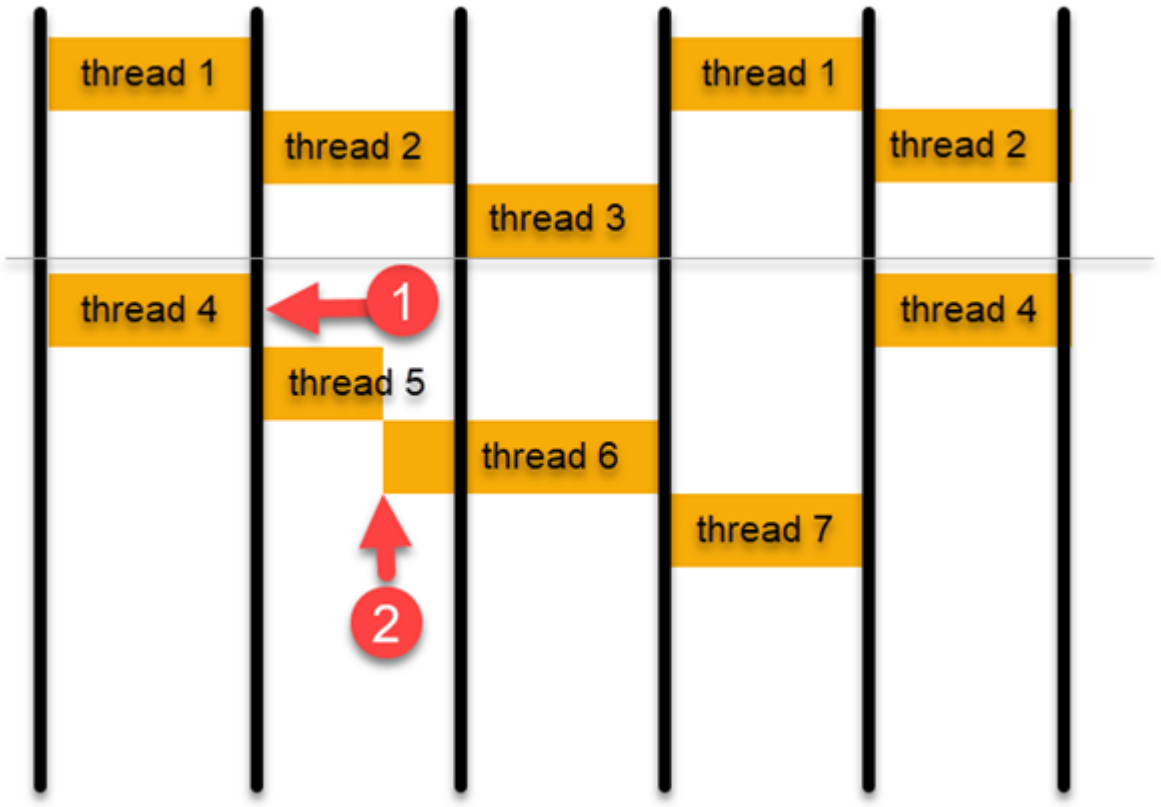
Потоки выполнения могут в разы ускорить отклик приложения и обработку данных. Но потоки выполнения напрямую зависят от количества процессоров и ядер. Соответственно, если создать в своем приложении большое количество потоков, им сложно будет конкурировать за процессорное время.

Чаще всего на одно ядро приходится по два потока, но все зависит от конкретного случая. Например, если процессор компьютера старый, то .Net может посчитать технологию MMX у Intel процессоров за дополнительное ядро.

Планировщик задач чередует выполнение потоков и создает видимости работы каждого приложения.

Квант времени процесса и потока – время, в течение которого этот поток выполняется на процессоре.

Разберем схему выполнения потоков на процессоре с двумя ядрами:



Черные толстые линии — это квант времени процессора. Тонкая серая линия разделяет логические ядра процессора. На первом ядре чередуются потоки с первого по третий, а на втором — с четвертого по седьмой.

На первом ядре не происходит никаких блокировок и все потоки получают свой полный квант времени. На втором поток под номером четыре в конце работы ставит блокировку на каком-либо объекте. Планировщик передает управление пятому потоку, а тот пытается во время работы взять блокировку на тот же объект, что и четвертый поток. Но объект заблокирован. Соответственно, оставшийся квант времени отдается другому потоку. Это и есть потеря времени выполнения потока в классическом виде.

Такой подход не всегда удобен. Например, для высоконагруженных программ простой потоков может быть критичен. Но в .Net есть дополнительные методы синхронизации, которые не отдают квант времени работы другому потоку, а ждут освобождения заблокированного объекта.

Не lock’ом единым, дополнительные методы синхронизации

На предыдущем уроке мы разобрали один из самых популярных методов синхронизации — “lock”. Под собой он скрывает блокировку методом “Monitor.Enter и Monitor.Exit” и дополнительным кодом, который этот метод оптимизирует в зависимости от среды окружения. Так происходит, потому что ваше приложение может запускаться на различных конфигурациях.

К примеру:

1. Один процессор и одно ядро;
2. Один процессор и много ядер;
3. Несколько процессоров.

В зависимости от конфигурации lock будет вести себя по-разному. В целом, синхронизация потоков выполнения по этому оператору является гибридной.

В .Net помимо гибридной синхронизации есть еще несколько видов.

Уровень ядра:

1. Mutex
2. Semaphore
3. Events

Уровень пользователя:

1. Volatile
2. Interlocked
3. MemoryBarrier

Гибридные:

1. Lock as Monitor
2. ReaderWriterLock / ReaderWriterLockSlim

Считается, что синхронизация уровня ядра одна из самых дорогостоящих, потому что мы задействуем системные методы операционной системы. Уровень пользователя можно посчитать дешевле, а гибридные синхронизации более адаптивны к конфигурации и пытаются достичь максимального “freelock” эффекта.

Разберем несколько примеров из уровня ядра и пользователя:

```
using System;
using System.Collections.Concurrent;
using System.Collections.Generic;
using System.Linq;
using System.Threading;

namespace Interview.Sample
{
	class Program
	{
	private static Mutex _mutex = new Mutex();

	static void Main(string[] args)
	{
		for (int i = 0; i < 10; i++)
		{
		Thread thread = new Thread(new ThreadStart(RunThread));
		thread.Start();
		}
	}

	static void RunThread()
	{
		_mutex.WaitOne();

		Thread.Sleep(1000);

		_mutex.ReleaseMutex();
	}
	}
}
```

Создается объект “Mutex”. Затем при входе в метод потока он захватывается потоком исполнения, а другие потоки ждут, пока он не освободится.

Синхронизацию уровня пользователя часто используют там, где нужно изменить значение переменной:

Метод	Назначение
Increment()	Безопасно инкрементирует значение на 1
Decrement()	Безопасно декрементирует значение на 1
Exchange()	Безопасно меняет два значения местами

CompareExchnage()	Безопасно проверяет на эквивалентность два значения, и, если они эквивалентны, меняет на третье значение

Простой пример:

```
using System;
using System.Collections.Concurrent;
using System.Collections.Generic;
using System.Linq;
using System.Threading;

namespace Interview.Sample
{
	class Program
	{
	  private static int _counter;

	static void Main(string[] args)
	{
		for (int i = 0; i < 10; i++)
		{
		    Thread thread = new Thread(new ThreadStart(RunThread));
		    thread.Start();
		}
	}

	static void RunThread()
	{
		int val = Interlocked.Increment(ref _counter);
	}
	}
}
```

Рекомендации

На первых этапах мы рекомендуем использовать оператор “lock” при синхронизации потоков. Он прост и оптимизирован под разные конфигурации.

Создание потоков — это ресурсоемкая операция, которая напрямую вызывает системные методы операционной системы и впоследствии должна выделить ресурсы на новый поток. Для экономии ресурсов лучше не создавать потоки напрямую, а работать через паттерн “threadpool”.

Пул объектов

Смысл данного паттерна в том, чтобы ненужный объект вернуть в хранилище, и, если он снова потребуется, вернуть его оттуда. Если же свободных объектов нет, то нужно их создать.

Давайте рассмотрим пример использования этого паттерна:

```
using System;
using System.Collections.Concurrent;
using System.Collections.Generic;
using System.Linq;
using System.Threading;

namespace Interview.Sample
{
	class Program
	{
	private static int _counter;

	static void Main(string[] args)
	{
		ObjectPool<ClassicPullItem> _pool = new
ObjectPool<ClassicPullItem>();

		ClassicPullItem item = _pool.Get();

		item.Id = 100;
		item.Name = "Just4Fun";

		_pool.Release(item);

		ClassicPullItem itemSecond = _pool.Get();
	}
	}

	public sealed class ObjectPool<TPullItem>
	where TPullItem : PullItem, new()
	{
	private readonly Queue<TPullItem> _queue = new Queue<TPullItem>();

	public TPullItem Get()
	{
		if (_queue.Count > 0)
		{
		return _queue.Dequeue();
		}

		return new TPullItem();
	}

	public void Release(TPullItem item)
	{
		if (item is null)
		{
		return;
		}

		  item.Reset();

		_queue.Enqueue(item);
	}

	}

	public abstract class PullItem
	{
```

```
      public abstract void Reset();
      }

      public sealed class ClassicPullItem : PullItem
      {
      public int Id { get; set; }

      public string Name { get; set; }

      public override void Reset()
      {
            Id = 0;

            Name = string.Empty;
      }
      }
}
```

Существует класс ObjectPool с открытым дженерик типом, но ограниченный по типу класса. Внутри него есть очередь, которая содержит ненужные объекты, а также два метода — один выдает освободившийся объект или создает новый, а второй очищает объект и регистрирует его как освободившийся.

Такой подход часто применяют для контроля ресурса.

Threadpool, он же тредпул

Создание потоков выполнения — ресурсоемкая задача, поэтому мы рекомендуем использовать что-то по типу пула потоков. В библиотеке .Net уже есть функционал, который кэширует потоки, минимизируя риски выделения лишних ресурсов.

Разберем пример:

```
using System;
using System.Collections.Concurrent;
using System.Collections.Generic;
using System.Linq;
using System.Threading;

namespace Interview.Sample
{
      class Program
      {
      static void Main(string[] args)
      {
            for (int i = 0; i < 100; i++)
            {
                ThreadPool.QueueUserWorkItem(new
WaitCallback(WaitCallback));
            Thread.Sleep(1000);
            }
      }

      private static void WaitCallback(object? state)
      {
            Console.WriteLine($"{Thread.CurrentThread.ManagedThreadId}");
      }
      }
}
```

При исполнении этого примера вывод будет таким:

```
 4
 5
 4
 4
 5
 5
 4
 5
 4
 4
 5
 6
 5
 4
```

Важно понимать, что пул потоков идеален для фоновых задач. Этот класс активно используется при работе с асинхронным кодом, так как “Task, Task<T>” напрямую связаны с использованием пула потоков.

Домашнее задание

1. Используя знания о пуле потоков, напишите свой микро пул с использованием новых структур данных.
2. Добавьте в ваш пул настройку для максимального количества регистрируемых потоков или кидайте ошибку.

Дополнительные материалы

1. https://docs.microsoft.com/en-us/dotnet/standard/threading/using-threads-and-threading

Используемые источники

1. https://docs.microsoft.com/en-us/dotnet/api/system.threading.thread?view=net-5.0
2. https://docs.microsoft.com/en-us/dotnet/standard/threading/using-threads-and-threading