Паралельне програмування з використанням бібліотеки TPL та PLINQ

ConcurrentBag, ConcurrentQueue, ConcurrentDictionary, ConcurrentStack, BlockedCollection

Лекція 5. Паралельні колекції

1. Паралельні колекції

Динамічні структури даних простору System.Collections і System.Collections.Generic не є потокобезпечними. Звернення до колекції декількох потоків може призводити до проблем гонки даних: втрата елементів, вихід індексу за межі і аварійне завершення роботи програми.

Для реалізації паралельних обчислень над колекціями слід використовувати паралельні колекції, розміщені у просторі імен System.Collections.Concurrent. Ці колекції можуть безпечно використовуватися в паралельній програмі, де можливий одночасний доступ до колекції з боку двох або більше паралельно виконуваних потоків (задач).

Застосування паралельних колекцій не потребує використання додаткових засобів синхронізації.

Колекція	**Опис**	**Основні методи**
ConcurrentQueue<T>	Черга.	TryDequeue() і TryPeek().
ConcurrentStack<T>	Стек	Push(), PushRange(), TryPeek(), TryPop() і TryPopRange().
ConcurrentBag<T>	Не впорядкована колекція (аналог List чи масиву)	Add(), TryPeek() і TryTake().
ConcurrentDictionary <TKey, TValue>	Словник	TryAdd(), TryGetValue(), TryRemove() і TryUpdate().
BlockingCollection<T>	Обмежена колекція	Add() і Take(). TryAdd() і TryTake()

Спочатку розглянемо роботу з простим універсальним словником.

Приклад 1. Не паралельний словник

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;
using System.Threading.Tasks;

namespace Lab_5
{
        class Program
        {
                static void Main(string[] args)
                {
```

```
                // Створюємо звичайний словник
                 var dic = new Dictionary<string, int>();
                // Паралельно оновлюємо значення елементу з ключем "one"
                 Parallel.For(0, 100000, i =>
                 {
                  if(dic.ContainsKey("one"))
                     dic["one"]++;
                  else
                  dic.Add("one", 1);
                 });
            Console.WriteLine("Element one: {0}", dic["one"]);
            Console.ReadKey();
         }
        }
}
```

Результат роботи може бути правильним, а може призвести до неправильних результатів і навіть до переривання.

Для безпечного відносно потоків доступу до колекцій визначений інтерфейс **IProducerConsumerCollection<T>.** Найбільш важливими методами цього інтерфейсу є TryAdd() і TryTake(). Метод TryAdd() намагається додати елемент в колекцію, але це може не вийти, якщо колекція заблокована від додавання елементів. Метод повертає булеве значення, що повідомляє про успіх або невдачу операції.

TryTake() працює аналогічним чином, інформуючи код про успіх або невдачу, і у разі успіху повертає елемент з колекції.

Приклад 2. Паралельний словник

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;
using System.Threading.Tasks;
using System.Collections.Concurrent;
namespace Lab_5
{
    class Program
    {
        static void Main(string[] args)
        {
            // Створюємо паралельний словник
            var dic = new ConcurrentDictionary<string, int>();
            //Паралельно оновлюємо значення елементу з ключем "one"
            Parallel.For(0, 100000, i =>
            {
              if(dic.ContainsKey("one"))
                dic["one"]++;
              else
                dic.TryAdd("one", 1);
            });
           Console.WriteLine("Element one: {0}", dic["one"]);
           Console.ReadKey();
        }
    }
}
```

2. Колекція ConcurrentBag<T>

Колекція ConcurrentBag<T> призначена для зберігання невпорядкованої колекції об'єктів (повтори дозволені). Відсутність певного порядку вибору елементів підвищує продуктивність операції читання і додавання для ConcurrentBag. Об'єкт ConcurrentBag однаково ефективний при однопоточному додаванні і при багатопотокочному.

Усередині об'єкту ConcurrentBag містяться зв'язані списки для кожного потоку. Елементи додаються в той список, який асоційований з поточним потоком. При виборі елементів спочатку очищується локальна черга цього потоку. Якщо в локальній черзі містяться елементи, то вибір є максимально ефективним. Якщо локальна черга порожня, то потік запозичує елементи з локальних черг інших потоків. Таким чином, вибір елементів з ConcurrentBag здійснюється за принципом LIFO з урахуванням локальності.

Приклад 3.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;
using System.Threading.Tasks;
using System.Collections.Concurrent;
namespace Lab_5
{
    class Program
    {
        static void Main(string[] args)
        {
            var bag = new ConcurrentBag<int>();
            for (int i = 0; i < 10; i++)
                bag.Add(i);

            foreach (int k in bag)
            {
                bag.Add(k);
                Console.Write(bag.Count + " ");
            }

           Console.ReadKey();
        }
    }
}
```

У цьому фрагменті в циклі foreach додаються нові елементи і виводиться розмір колекції. Отримуємо 10 ітерацій, але розмір колекції при цьому збільшується до 20:

```
11 12 13 14 15 16 17 18 19 20
```

Заміна колекції ConcurrentBag на список List призвела б до виникнення необробленого виключення.

Ця колекція має високу продуктивність за рахунок, того що не піклується про збереження порядку доданих в неї елементів, її зручно використовувати, коли необхідно зібрати дані з різних потоків для відправки результуючого набору як параметр в наступний метод.

Приклад 4.

У цьому прикладі показано, як додавати і видаляти елементи з ConcurrentBag<T>

https://msdn.microsoft.com/ru-ru/library/dd381779(v=vs.110).aspx

```
using System;
using System.Collections.Concurrent;

class ConcurrentBagDemo
{
    // Demonstrates:
    //      ConcurrentBag<T>.Add()
    //      ConcurrentBag<T>.IsEmpty
    //      ConcurrentBag<T>.TryTake()
    //      ConcurrentBag<T>.TryPeek()
    static void Main()
    {
        // Construct and populate the ConcurrentBag
        ConcurrentBag<int> cb = new ConcurrentBag<int>();
        cb.Add(1);
        cb.Add(2);
        cb.Add(3);

        // Consume the items in the bag
        int item;
        while (!cb.IsEmpty)
        {
            if (cb.TryTake(out item))
                Console.WriteLine(item);
            else
                Console.WriteLine("TryTake failed for non-empty bag");
        }

        // Bag should be empty at this point
        if (cb.TryPeek(out item))
            Console.WriteLine("TryPeek succeeded for empty bag!");
        Console.ReadKey();
    }

}
```

Розглянемо приклад використання списку та реалізацію паралельної версії з цією самою функціональністю, але з використанням **ConcurrentBag<T>.**

Приклад 5. Не паралельне використання списку

Використання класу List<> для зберігання колекції об'єктів класу Person, а також для зберігання цілих чисел.

```
using System;
using System.Collections.Generic;
using System.Collections;
using System.Linq;
using System.Text;

namespace Lab_5
```

```
{
    public class Person
    {
        public string Name;        //имя
        public int Age;            // возраст
        public string Role;        // роль
        public string GetName() { return Name; }
        public int GetAge() { return Age; }
        public Person(string N, int A)
        {
            this.Name = N;
            this.Age = A;
        }
        public void Passport()
        {
            Console.WriteLine("Name = {0} Age = {1}", Name, Age);
        }
    }

    class Program
    {

        static void Main(string[] args)
        {
            List<Person> pers = new List<Person>();
            pers.Add(new Person("Іванов",18));
            pers.Add(new Person("Петров",20));
            pers.Add(new Person("Морозов", 3));

            foreach (Person x in pers) x.Passport();

            List<int> lint = new List<int>();
            lint.Add(5); lint.Add(1); lint.Add(3);
            lint.Sort();
            int a = lint[2];
            Console.WriteLine(a);

            foreach (int x in lint) Console.Write(x + " ");

            Console.ReadLine();

        }
    }
}
```

Приклад 6. Паралельна робота з колекцією ConcurrentBag<T>

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;
using System.Threading.Tasks;
using System.Collections.Concurrent;
namespace Lab_5
{
    public class Person
    {
        public string name;        //имя
        public int age;            // возраст
        public string role;        // роль
        public string GetName() { return name; }
        public int GetAge() { return age; }
        public Person(string n, int a)
        {
```

```
            this.name = n;
            this.age = a;
        }
        public void Passport()
        {
            Console.WriteLine("Name = {0} Age = {1}", name, age);
        }
    }

    class Program
    {

        static void Main(string[] args)
        {
            //List<Person> pers = new List<Person>();
            ConcurrentBag<Person> pers = new ConcurrentBag<Person>();
            pers.Add(new Person("Іванов", 18));
            pers.Add(new Person("Петров", 20));
            pers.Add(new Person("Морозов", 3));
            pers.Add(new Person("Сидоров", 21));
            pers.Add(new Person("Яхно", 21));
            pers.Add(new Person("Яценюк", 41));
            foreach (Person x in pers) x.Passport();
  //          List<int> lint = new List<int>();
            ConcurrentBag<int> lint = new ConcurrentBag<int>();
            lint.Add(5); lint.Add(1); lint.Add(3);
  //          lint.Sort();
  //          int a = lint[2];
            int a;
            if (lint.TryTake(out a))
                Console.WriteLine(a);
            foreach (int x in lint) Console.Write(x + " ");
            Console.ReadLine();
        }
    }

}
```

Примітка

Метод Sort() і доступ до елементу за індексом не припустимі для ConcurrentBag<T>.

3. Колекція BlockingCollection<T>

Колекція, яка здійснює блокування і чекає, поки не з'явиться можливість виконати дію з додавання або вибору елементу.

Завдання цієї колекції надати обмежене в розмірі сховище, причому, якщо потік спробує записати в заповнену колекцію, він буде блокований, поки в колекції не звільниться місце, а якщо потік спробує прочитати елемент з порожньої колекції, то він буде блокований, поки там не з'явиться хоч би один елемент.

BlockingCollection<T> пропонує інтерфейс для додавання і вибору елементів методами Add() і Take(). Ці методи блокують потік і потім чекають, поки не з'явиться можливість виконати завдання.

Метод Add() має перевантаження, якому можна також передати CancellationToken. Ця лексема завжди відміняє блокуючий виклик.

Якщо не треба, щоб потік чекав нескінченний час, і не хочете відміняти виклик ззовні, доступні також методи TryAdd() і TryTake(). В них можна вказати значення таймауту - максимального періоду часу, протягом якого ви готові блокувати потік і чекати, поки виклик не дасть збій.

Об'єкт BlockingCollection дозволяє сформувати модифіковану паралельну колекцію на базі ConcurrentStack, ConcurrentQueue або ConcurrentBag. Модифікації паралельних колекцій застосовуються для реалізації шаблону "виробник-споживач". Метод Take для BlockingCollection викликається потоком-споживачем і призводить до блокування потоку у разі відсутності елементів. Додавання елементів потоком-виробником призводить до розблокування (звільненню) споживача.

При створенні колекції є можливість встановлення максимальної місткості. Якщо колекція повністю заповнена, то операція додавання елементу призводить до блокування поточного потоку до тих пір, поки потік-споживач не витягне, хоч би один елемент.

Метод **CompleteAdding** дозволяє завершити додавання елементів в колекцію. Звернення до операції **Add** призведе до виключень. Властивість IsAddingCompleted дозволяє перевірити статус завершення. Вибір елементів з "завершеної" колекції дозволений, поки колекція не порожня. Операція **Take** для порожньої завершеної колекції призводить до генерації виключення. Властивість IsCompleted дозволяє встановити, чи є колекція порожньою і завершеною одночасно.

Тип колекції, яка використовується для формування BlockingCollection, визначає порядок вибору елементів. Якщо об'єкт BlockingCollection створюється без вказівки паралельної колекції, то як базова колекція використовується черга типу ConcurrentQueue.

Приклад 6

У наступному прикладі показано, як додавати і видаляти елементи з BlockingCollection одночасно:

https://msdn.microsoft.com/ru-ru/library/dd267312(v=vs.110).aspx

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

class BlockingCollectionDemo
{
    static void Main()
    {
        AddTakeDemo.BC_AddTakeCompleteAdding();
        TryTakeDemo.BC_TryTake();
        FromToAnyDemo.BC_FromToAny();
        ConsumingEnumerableDemo.BC_GetConsumingEnumerable();
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
```

```
}
class AddTakeDemo
{
    // Demonstrates:
    //      BlockingCollection<T>.Add()
    //      BlockingCollection<T>.Take()
    //      BlockingCollection<T>.CompleteAdding()
    public static void BC_AddTakeCompleteAdding()
    {
        using (BlockingCollection<int> bc = new BlockingCollection<int>())
        {

            // Spin up a Task to populate the BlockingCollection
            using (Task t1 = Task.Factory.StartNew(() =>
            {
                bc.Add(1);
                bc.Add(2);
                bc.Add(3);
                bc.CompleteAdding();
            }))
            {

                // Spin up a Task to consume the BlockingCollection
                using (Task t2 = Task.Factory.StartNew(() =>
                {
                    try
                    {
                        // Consume consume the BlockingCollection
                        while (true) Console.WriteLine(bc.Take());
                    }
                    catch (InvalidOperationException)
                    {
                        // An InvalidOperationException means that Take() was
called on a completed collection
                        Console.WriteLine("That's All!");
                    }
                }))

                    Task.WaitAll(t1, t2);
            }
        }
    }
}

class TryTakeDemo
{
    // Demonstrates:
    //      BlockingCollection<T>.Add()
    //      BlockingCollection<T>.CompleteAdding()
    //      BlockingCollection<T>.TryTake()
    //      BlockingCollection<T>.IsCompleted
    public static void BC_TryTake()
    {
        // Construct and fill our BlockingCollection
        using (BlockingCollection<int> bc = new BlockingCollection<int>())
        {
            int NUMITEMS = 10000;
            for (int i = 0; i < NUMITEMS; i++) bc.Add(i);
            bc.CompleteAdding();
            int outerSum = 0;

            // Delegate for consuming the BlockingCollection and adding up
all items
```

```
            Action action = () =>
            {
                int localItem;
                int localSum = 0;

                while (bc.TryTake(out localItem)) localSum += localItem;
                Interlocked.Add(ref outerSum, localSum);
            };

            // Launch three parallel actions to consume the
BlockingCollection
            Parallel.Invoke(action, action, action);

            Console.WriteLine("Sum[0..{0}) = {1}, should be {2}", NUMITEMS,
outerSum, ((NUMITEMS*(NUMITEMS - 1))/2));
            Console.WriteLine("bc.IsCompleted = {0} (should be true)",
bc.IsCompleted);
        }
    }
}

class FromToAnyDemo
{

    // Demonstrates:
    //      Bounded BlockingCollection<T>
    //      BlockingCollection<T>.TryAddToAny()
    //      BlockingCollection<T>.TryTakeFromAny()
    public static void BC_FromToAny()
    {
        BlockingCollection<int>[] bcs = new BlockingCollection<int>[2];
        bcs[0] = new BlockingCollection<int>(5); // collection bounded to 5
items
        bcs[1] = new BlockingCollection<int>(5); // collection bounded to 5
items

        // Should be able to add 10 items w/o blocking
        int numFailures = 0;
        for (int i = 0; i < 10; i++)
        {
            if (BlockingCollection<int>.TryAddToAny(bcs, i) == -1)
numFailures++;
        }
        Console.WriteLine("TryAddToAny: {0} failures (should be 0)",
numFailures);

        // Should be able to retrieve 10 items
        int numItems = 0;
        int item;
        while (BlockingCollection<int>.TryTakeFromAny(bcs, out item) != -1)
numItems++;
        Console.WriteLine("TryTakeFromAny: retrieved {0} items (should be
10)", numItems);
    }
}

class ConsumingEnumerableDemo
{
    // Demonstrates:
    //      BlockingCollection<T>.Add()
    //      BlockingCollection<T>.CompleteAdding()
    //      BlockingCollection<T>.GetConsumingEnumerable()
    public static void BC_GetConsumingEnumerable()
```

```
    {
        using (BlockingCollection<int> bc = new BlockingCollection<int>())
        {

            // Kick off a producer task
            Task.Factory.StartNew(() =>
            {
                for (int i = 0; i < 10; i++)
                {
                    bc.Add(i);
                    Thread.Sleep(100); // sleep 100 ms between adds
                }

                // Need to do this to keep foreach below from hanging
                bc.CompleteAdding();
            });

            // Now consume the blocking collection with foreach.
            // Use bc.GetConsumingEnumerable() instead of just bc because the
            // former will block waiting for completion and the latter will
            // simply take a snapshot of the current state of the underlying
collection.
            foreach (var item in bc.GetConsumingEnumerable())
            {
                Console.WriteLine(item);
            }
        }
    }
}
```

4. Колекція ConcurrentDictionary

Паралельний словник окрім потокобезпечності додавання і видалення елементів надає розширені функціональні можливості: метод умовного додавання GetOrAdd () і методи оновлення значеньAddOrUpdate().

Приклад 8.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;
using System.Threading.Tasks;
using System.Collections.Concurrent;
namespace Lab1_5{
    class Program
    {
        static void Main(string[] args)
        {
          // Створюємо паралельний словник
           var cd = new ConcurrentDictionary<string, int>();
          // Хочемо отримати елемент з ключем "b", якщо такого немає - створюємо
          int value = cd.GetOrAdd("b", (key) => 555);
          // Перевіряємо: value = 555;
          value = cd.GetOrAdd("b", -333);
          // Паралельно намагаємося відновити елемент з ключем "a"
          Parallel.For(0, 100000, i =>
           {
             // Якщо ключа немає - додаємо
             // Якщо є - оновлюємо значення
              cd.AddOrUpdate("a", 1, (key, oldValue) => oldValue + 1);
            });
           Console.WriteLine("Element a: {0}", cd["a"]);
```

```
            Console.ReadKey();
        }
    }
}
```

Метод **GetOrAdd** реалізує можливість отримання значення по ключу або у разі відсутності елементу - додавання.

```
if(dic.ContainsKey(sKey))
  val = dic[sKey];
else
  dic.Add(sKey, sNewValue);
```

Але метод GetOrAdd виконується атомарно на відміну від наведеного фрагмента, тобто декілька потоків не можуть одночасно перевірити наявність елементу і здійснити додавання.

Приклад 9.

У наступному прикладі два екземпляри класу Task використовуються для одночасного додавання елементів в клас ConcurrentDictionary<TKey, TValue>, потім виконується виведення усього вмісту, для того, щоб показати, що додавання елементів виконане успішно.У прикладі також показано, як використовувати методи AddOrUpdate, TryGetValue, GetOrAdd для додавання, оновлення і вибору елементів з колекції.

Для порівняння даних у словнику використовується інтерфейс IEqualityComparer<T>.

https://msdn.microsoft.com/ru-ru/library/dd997369(v=vs.110).aspx

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


namespace Lab5_ConcDict
{
   // The type of the Value to store in the dictionary:
   class CityInfo : IEqualityComparer<CityInfo>
   {
      public string Name { get; set; }               //назва міста
      public DateTime lastQueryDate { get; set; }    //остання дата запиту
      public decimal Longitude { get; set; }         //довгота
      public decimal Latitude { get; set; }          //широта
      public int[] RecentHighTemperatures { get; set; }  //останні високі температури

      public CityInfo(string name, decimal longitude, decimal latitude, int[] temps)
      {
         Name = name;
         lastQueryDate = DateTime.Now;
         Longitude = longitude;  //довгота
         Latitude = latitude;    //широта
         RecentHighTemperatures = temps;
```

```
        }

        public CityInfo()
        {
        }

        public CityInfo(string key)
        {
            Name = key;
            Longitude = Decimal.MaxValue;
            Latitude = Decimal.MaxValue;
            lastQueryDate = DateTime.Now;
            RecentHighTemperatures = new int[] { 0 };

        }
        public bool Equals(CityInfo x, CityInfo y)
        {
            return x.Name == y.Name && x.Longitude == y.Longitude && x.Latitude == y.Latitude;
        }

        public int GetHashCode(CityInfo obj)
        {
            CityInfo ci = (CityInfo)obj;
            return ci.Name.GetHashCode();
        }
    }

    class Program
    {
        // Create a new concurrent dictionary.
        static ConcurrentDictionary<string, CityInfo> cities = new ConcurrentDictionary<string,
CityInfo>();

        static void Main(string[] args)
        {
            CityInfo[] data =
            {
                new CityInfo(){ Name = "Boston", Latitude = 42.358769M, Longitude = -71.057806M,
RecentHighTemperatures = new int[] {56, 51, 52, 58, 65, 56,53}},
                new CityInfo(){ Name = "Miami", Latitude = 25.780833M, Longitude = -80.195556M,
RecentHighTemperatures = new int[] {86,87,88,87,85,85,86}},
                new CityInfo(){ Name = "Los Angeles", Latitude = 34.05M, Longitude = -118.25M,
RecentHighTemperatures =  new int[] {67,68,69,73,79,78,78}},
                new CityInfo(){ Name = "Seattle", Latitude = 47.609722M, Longitude =  -
122.333056M, RecentHighTemperatures =  new int[] {49,50,53,47,52,52,51}},
                new CityInfo(){ Name = "Toronto", Latitude = 43.716589M, Longitude = -
79.340686M, RecentHighTemperatures =  new int[] {53,57, 51,52,56,55,50}},
                new CityInfo(){ Name = "Mexico City", Latitude = 19.432736M, Longitude = -
99.133253M, RecentHighTemperatures =  new int[] {72,68,73,77,76,74,73}},
                new CityInfo(){ Name = "Rio de Janiero", Latitude = -22.908333M, Longitude = -
43.196389M, RecentHighTemperatures =  new int[] {72,68,73,82,84,78,84}},
                new CityInfo(){ Name = "Quito", Latitude = -0.25M, Longitude = -78.583333M,
RecentHighTemperatures =  new int[] {71,69,70,66,65,64,61}}
            };

            // Add some key/value pairs from multiple threads.
            Task[] tasks = new Task[2];    //дві задачі
```

```
//заповнення словника
        tasks[0] = Task.Run(() =>
        {
           for (int i = 0; i < 2; i++)
           {
              if (cities.TryAdd(data[i].Name, data[i]))
                 Console.WriteLine("Added {0} on thread {1}", data[i],
                    Thread.CurrentThread.ManagedThreadId);
              else
                 Console.WriteLine("Could not add {0}", data[i]);
           }
        });

        tasks[1] = Task.Run(() =>
        {
           for (int i = 2; i < data.Length; i++)
           {
              if (cities.TryAdd(data[i].Name, data[i]))
                 Console.WriteLine("Added {0} on thread {1}", data[i],
                    Thread.CurrentThread.ManagedThreadId);
              else
                 Console.WriteLine("Could not add {0}", data[i]);
           }
        });

        Task.WaitAll(tasks);      //очікування завершення задач

        // Enumerate collection from the app main thread.
        // Note that ConcurrentDictionary is the one concurrent collection
        // that does not support thread-safe enumeration.

      //виводимо назви міст з масиву міст
        foreach (var city in cities)
        {
           Console.WriteLine("{0} has been added.", city.Key);
        }
       //виклик методів
        AddOrUpdateWithoutRetrieving();    //
        RetrieveValueOrAdd();              //
        RetrieveAndUpdateOrAdd();          //

        Console.WriteLine("Press any key.");
        Console.ReadKey();
     }

     // This method shows how to add key-value pairs to the dictionary
     // in scenarios where the key might already exist.
     //Додавання міста в словник, якщо в словнику вже є елемент з таким ключем
     private static void AddOrUpdateWithoutRetrieving()
     {
        // Sometime later. We receive new data from some source.
     //створення об'єкту
        CityInfo ci = new CityInfo()
        {
           Name = "Toronto",
           Latitude = 43.716589M,
           Longitude = -79.340686M,
```

```
            RecentHighTemperatures = new int[] { 54, 59, 67, 82, 87, 55, -14 }
        };

        // Try to add data. If it doesn't exist, the object ci is added. If it does
        // already exist, update existingVal according to the custom logic in the
        // delegate.
        //пробуємо додати об'єкт. Якщо такого об'єкту немає, додаємо його в словник.
        //якощ він вже є, оновлюємо його значення
        cities.AddOrUpdate(ci.Name, ci,
            (key, existingVal) =>
            {
                // If this delegate is invoked, then the key already exists.
                // Here we make sure the city really is the same city we already have.
                // (Support for multiple cities of the same name is left as an exercise for the reader.)
                try
                {
                    if (ci != existingVal)
                        throw new ArgumentException("Duplicate city names are not allowed: {0}.",
ci.Name);
                }
                catch (ArgumentException e)
                {
                    Console.WriteLine(e.Message);
                }
                // The only updatable fields are the temerature array and lastQueryDate.
                existingVal.lastQueryDate = DateTime.Now;
                existingVal.RecentHighTemperatures = ci.RecentHighTemperatures;
                return existingVal;
            });

        // Verify that the dictionary contains the new or updated data.
        //виводимо температури
        Console.Write("Most recent high temperatures for {0} are: ", cities[ci.Name].Name);
        int[] temps = cities[ci.Name].RecentHighTemperatures;
        foreach (var temp in temps) Console.Write("{0}, ", temp);
        Console.WriteLine();
    }

    // This method shows how to use data and ensure that it has been
    // added to the dictionary.
    private static void RetrieveValueOrAdd()
    {
        string searchKey = "Caracas";
        CityInfo retrievedValue = null;

        try
        {
            retrievedValue = cities.GetOrAdd(searchKey, GetDataForCity(searchKey));
        }
        catch (ArgumentException e)
        {
            Console.WriteLine(e.Message);
        }

        // Use the data.
        if (retrievedValue != null)
        {
```

```
        Console.Write("Most recent high temperatures for {0} are: ", retrievedValue.Name);
        int[] temps = cities[retrievedValue.Name].RecentHighTemperatures;
        foreach (var temp in temps) Console.Write("{0}, ", temp);
    }
    Console.WriteLine();
}
// This method shows how to retrieve a value from the dictionary,
// when you expect that the key/value pair already exists,
// and then possibly update the dictionary with a new value for the key.
private static void RetrieveAndUpdateOrAdd()
{
    CityInfo retrievedValue;
    string searchKey = "Buenos Aires";

    if (cities.TryGetValue(searchKey, out retrievedValue))
    {
        // use the data
        Console.Write("Most recent high temperatures for {0} are: ", retrievedValue.Name);
        int[] temps = retrievedValue.RecentHighTemperatures;
        foreach (var temp in temps) Console.Write("{0}, ", temp);

        // Make a copy of the data. Our object will update its lastQueryDate automatically.
        CityInfo newValue = new CityInfo(retrievedValue.Name,
                                         retrievedValue.Longitude,
                                         retrievedValue.Latitude,
                                         retrievedValue.RecentHighTemperatures);

        // Replace the old value with the new value.
        if (!cities.TryUpdate(searchKey, retrievedValue, newValue))
        {
            //The data was not updated. Log error, throw exception, etc.
            Console.WriteLine("Could not update {0}", retrievedValue.Name);
        }
    }
    else
    {
        // Add the new key and value. Here we call a method to retrieve
        // the data. Another option is to add a default value here and
        // update with real data later on some other thread.
        CityInfo newValue = GetDataForCity(searchKey);
        if (cities.TryAdd(searchKey, newValue))
        {
            // use the data
            Console.Write("Most recent high temperatures for {0} are: ", newValue.Name);
            int[] temps = newValue.RecentHighTemperatures;
            foreach (var temp in temps) Console.Write("{0}, ", temp);
        }
        else
            Console.WriteLine("Unable to add data for {0}", searchKey);
    }
}

//Assume this method knows how to find long/lat/temp info for any specified city.
static CityInfo GetDataForCity(string name)
{
    // Real implementation left as exercise for the reader.
    if (String.CompareOrdinal(name, "Caracas") == 0)
```

```
            return new CityInfo()
            {
               Name = "Caracas",
               Longitude = 10.5M,
               Latitude = -66.916667M,
               RecentHighTemperatures = new int[] { 91, 89, 91, 91, 87, 90, 91 }
            };
         else if (String.CompareOrdinal(name, "Buenos Aires") == 0)
            return new CityInfo()
            {
               Name = "Buenos Aires",
               Longitude = -34.61M,
               Latitude = -58.369997M,
               RecentHighTemperatures = new int[] { 80, 86, 89, 91, 84, 86, 88 }
            };
         else
            throw new ArgumentException("Cannot find any data for {0}", name);
      }
   }

}
```

Висновки

Звичайні колекції не дозволяють змінювати об'єкт, який використовується в циклі foreach. Паралельні колекції, забезпечуючи багатопотоковий доступ, дозволяють додавати елементи усередині циклу foreach при переборі елементів. При цьому зміни, що вносяться усередині перелічування, не відбиваються на поточному нумераторі:

Паралельні колекції спроектовані для застосування в багатопотокових сценаріях і не є повною мірою еквівалентом звичайних колекцій з блокуваннями. У разі однопроцесорної системи або при використанні паралельних колекцій в одному потоці їх ефективність може бути нижче, ніж використання звичайних колекцій. Тому рекомендується використовувати паралельні колекції тільки у разі багатопотокової роботи.

Контрольні запитання і завдання для самостійного виконання

1. Які паралельні колекції є в C#?
2. Яка різниця між звичайними і паралельними колекціями?
3. Яку паралельну колекцію можна використовувати для реалізації списку?
4. Яку паралельну колекцію можна використувати для стека, черги?
5. Яке призначення ConcurrentBag<T>?
6. Яке призначення BlockingCollection<T> і у чому особливості цієї колекції?
7. Який метод призначений для додавання елементу у ConcurrentDictionary<T>? Для видалення елементу?
8. У яких випадках використання паралельних колекцій не є ефективним?
9. Яке призначення методів GetOrAdd () і AddOrUpdate()?