

**МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ**

**РОССИЙСКОЙ ФЕДЕРАЦИИ**

Брянский государственный технический университет

**Утверждаю**

**Ректор университета**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_О.Н.Федонин**

**«\_\_\_\_»\_\_\_\_\_\_\_\_\_\_\_\_2017 г.**

**Программирование в среде Microsoft .NET**

**Создание многопоточных и сетевых приложений**

**Методические указания**

**к выполнению лабораторной работы №10**

**для студентов очной формы обучения по направлениям подготовки 02.03.03 – «Математическое обеспечение и администрирование информационных систем», 09.03.01 – «Информатика и вычислительная техника» и 09.03.04 – «Программная инженерия»**

**Брянск 2017**

УКД 004.43

Программирование в среде Microsoft .Net. Создание многопоточных и сетевых приложений [Текст] + [Электронный ресурс]: Методические указания к выполнению лабораторной работы №10 для студентов очной формы обучения по направлениям подготовки 02.03.03 – «Математическое обеспечение и администрирование информационных систем», 09.03.01 – «Информатика и вычислительная техника» и 09.03.04 – «Программная инженерия». – Брянск: БГТУ, 2017. – 19 с.

Разработал:

Д.А.Коростелёв

канд. техн. наук, доц.

Рекомендовано кафедрой «Информатика и программное обеспечение» БГТУ (протокол №2 от 16.09.2016)

**Методические издания публикуются в авторской редакции**

**Цель работы**

Целью работы является изучение основ создания многопоточных и сетевых приложений с помощью библиотеки классов среды Microsoft .Net.

Продолжительность работы – 2 часа.

**Основы работы с потоками**

Основной функционал для использования потоков в приложении сосредоточен в пространстве имен System.Threading. В нем определен класс, представляющий отдельный поток - класс Thread.

Класс Thread определяет ряд методов и свойств, которые позволяют управлять потоком и получать информацию о нем. 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 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
* 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

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)
* 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

В процессе работы потока его статус многократно может измениться под действием методов. Так, в самом начале еще до применения метода Start его статус имеет значение Unstarted. Запустив поток, мы изменим его статус на Running. Вызвав метод Sleep, статус изменится на WaitSleepJoin. А применяя метод Abort, мы тем самым переведем поток в состояние AbortRequested, а затем Aborted, после чего поток окончательно завершится.

Приоритеты поток располагаются в перечислении ThreadPriority:

* Lowest
* BelowNormal
* Normal
* AboveNormal
* Highest

По умолчанию потоку задается значение Normal. Однако мы можем изменить приоритет в процессе работы программы. Например, повысить важность потока, установив приоритет Highest. Среда CLR будет считывать и анализировать значения приоритета и на их основании выделять данному потоку то или иное количество времени.

**Потокобезопасные коллекции**

В версию 4.0 среды .NET Framework добавлено новое пространство имен System.Collections.Concurrent. Оно содержит коллекции, которые являются потокобезопасными и специально предназначены для параллельного программирования. Это означает, что они могут безопасно использоваться в многопоточной программе, где возможен одновременный доступ к коллекции со стороны двух или больше параллельно исполняемых потоков.

Для безопасного в отношении потоков доступа к коллекциям определен интерфейс IProducerConsumerCollection<T>. Наиболее важными методами этого интерфейса являются TryAdd() и TryTake(). Метод TryAdd() пытается добавить элемент в коллекцию, но это может не получиться, если коллекция заблокирована от добавления элементов. Метод возвращает булевское значение, сообщающее об успехе или неудаче операции.

TryTake() работает аналогичным образом, информируя вызывающий код об успехе или неудаче, и в случае успеха возвращает элемент из коллекции. Ниже перечислены классы из пространства имен System.Collections.Concurrent с кратким описанием их функциональности:

1. ConcurrentQueue<T>. Этот класс коллекции реализован со свободным от блокировок алгоритмом и использует 32 массива, которые внутренне скомбинированы в связный список. Для доступа к элементам очереди применяются методы Enqueue(), TryDequeue() и TryPeek(). Имена этих методов очень похожи на уже известные методы Queue<T>, но с добавлением префикса Try к тем из них, которые могут дать сбой. Поскольку этот класс реализует интерфейс IProducerConsumerCollection<T>, методы TryAdd() и TryTake() просто вызывают Enqueue() и TryDequeue().
2. ConcurrentStack<T>. Очень похож на ConcurrentQueue<T>, но с другими методами доступа к элементам. Класс ConcurrentStack<T> определяет методы Push(), PushRange(), TryPeek(), TryPop() и TryPopRange(). Внутри этот класс использует связный список для хранения элементов.
3. ConcurrentBag<T>. Этот класс не определяет никакого порядка для добавления или извлечения элементов. Он реализует концепцию отображения потоков на используемые внутренне массивы, и старается избежать блокировок. Для доступа к элементам применяются методы Add(), TryPeek() и TryTake().
4. ConcurrentDictionary<TKey, TValue>. Безопасная в отношении потоков коллекция ключей и значений. Для доступа к членам в неблокирующем режиме служат методы TryAdd(), TryGetValue(), TryRemove() и TryUpdate(). Поскольку элементы основаны на ключах и значениях, ConcurrentDictionary<TKey, TValue> не реализует интерфейс IProducerConsumerCollection<T>.
5. ConcurrentXXX. Эти коллекции безопасны к потокам в том смысле, что возвращают false, если какое-то действие над ними невозможно при текущем состоянии потоков. Прежде чем предпринимать какие-то дальнейшие действия, всегда следует проверять успешность добавления или извлечения элементов. Полностью доверять коллекции решение задачи нельзя.
6. BlockingCollection<T>. Коллекция, которая осуществляет блокировку и ожидает, пока не появится возможность выполнить действие по добавлению или извлечению элемента. BlockingCollection<T> предлагает интерфейс для добавления и извлечения элементов методами Add() и Take(). Эти методы блокируют поток и затем ожидают, пока не появится возможность выполнить задачу. Метод Add() имеет перегрузку, которой можно также передать CancellationToken. Эта лексема всегда отменяет блокирующий вызов. Если не нужно, чтобы поток ожидал бесконечное время, и не хотите отменять вызов извне, доступны также методы TryAdd() и TryTake(). В них можно указать значение таймаута — максимального периода времени, в течение которого вы готовы блокировать поток и ждать, пока вызов не даст сбой.

**Сетевые приложения**

**Класс IPAddress**

Класс IPAddress представляет IP-адрес. Сам адрес доступен в виде свойства GetAddressBytes и может быть преобразован в десятичный формат с разделителями-точками с помощью метода ToString(). В этом классе также реализован статический метод Parse(), который эффективно выполняет преобразование, обратное ToString() — из десятичного формата с разделителями-точками в IPAddress:

IPAddress ip = IPAddress.Parse("8.8.8.8");

byte[] adress = ip.GetAddressBytes();

string ipString = ip.ToString();

В классе IPAddress есть несколько открытых, доступных только для чтения полей, которые возвращают предопределенные IP-адреса:

1. IPAddress.None. Возвращает адрес, который означает, что ни один сетевой интерфейс не должен использоваться. Это поле используется классом Socket, чтобы указать серверу не ожидать активности клиента.
2. IPAddress.Loopback. Возвращает предопределенный адрес обратной связи 127.0.0.1. Этот адрес используется не для соединения с сетью, а для локальных операций на одной машине.
3. IPAddress.Broadcast. Возвращает широковещательный IP-адрес. В широковещательных сообщениях можно посылать данные всем компьютерам в локальной сети.
4. IPAddress.Any. У компьютера может быть несколько сетевых плат с несколькими IP-адресами. Сокет использует IPAddress.Any, чтобы ожидать действия на любом из этих сетевых интерфейсов.

**Класс IPHostEntry.**

Класс IPHostEntry инкапсулирует информацию об определенном компьютере-хосте. Этот класс делает имя хоста доступным через свойство HostName (которое возвращает строку), а свойство AddressList возвращает массив объектов IPAddress.

**Класс Dns.**

Класс Dns способен взаимодействовать с DNS-сервером по умолчанию для извлечения IP-адреса. Он имеет два важных статических метода — GetHostEntry(), который использует DNS-сервер для получения деталей хоста по заданному его имени, и GetHostByAddress(), который также возвращает детали хоста, но на этот раз используя IP-адрес. Оба метода возвращают объект IPHostEntry.

Класс Dns отличается от классов IPAddress и IPHostEntry тем, что обладает способностью действительно взаимодействовать с серверами для получения информации. В отличие от него IPAddress и IPHostEntry — это скорее просто структуры данных с удобными свойствами, обеспечивающими доступ к лежащим в основе данным.

Поддержку сокетов в .NET обеспечивают классы в пространстве имен System.Net.Sockets — начнем с их краткого описания.

Классы для работы с сокетами представлены в табл. 1.

**Таблица 1**

***Классы для работы с сокетами***

|  |  |
| --- | --- |
| **Класс** | **Описание** |
| MulticastOption | Класс MulticastOption устанавливает значение IP-адреса для присоединения к IP-группе или для выхода из нее. |
| NetworkStream | Класс NetworkStream реализует базовый класс потока, из которого данные отправляются и в котором они получаются. Это абстракция высокого уровня, представляющая соединение с каналом связи TCP/IP. |
| TcpClient | Класс TcpClient строится на классе Socket, чтобы обеспечить TCP-обслуживание на более высоком уровне. TcpClient предоставляет несколько методов для отправки и получения данных через сеть. |
| TcpListener | Этот класс также построен на низкоуровневом классе Socket. Его основное назначение — серверные приложения. Он ожидает входящие запросы на соединения от клиентов и уведомляет приложение о любых соединениях. |
| UdpClient | UDP — это протокол, не организующий соединение, следовательно, для реализации UDP-обслуживания в .NET требуется другая функциональность. |
| SocketException | Это исключение порождается, когда в сокете возникает ошибка. |
| Socket | Последний класс в пространстве имен System.Net.Sockets — это сам класс Socket. Он обеспечивает базовую функциональность приложения сокета. |

**Класс Socket**

Класс Socket играет важную роль в сетевом программировании, обеспечивая функционирование как клиента, так и сервера. Главным образом, вызовы методов этого класса выполняют необходимые проверки, связанные с безопасностью, в том числе проверяют разрешения системы безопасности, после чего они переправляются к аналогам этих методов в Windows Sockets API.

Рассмотрим некоторые важные свойства и методы класса Socket (табл. 2).

**Таблица 2**

***Свойства и методы класса Socket***

|  |  |
| --- | --- |
| **Свойство или метод** | **Описание** |
| AddressFamily | Дает семейство адресов сокета — значение из перечисления Socket.AddressFamily. |
| Available | Возвращает объем доступных для чтения данных. |
| Blocking | Дает или устанавливает значение, показывающее, находится ли сокет в блокирующем режиме. |
| Connected | Возвращает значение, информирующее, соединен ли сокет с удаленным хостом. |
| LocalEndPoint | Дает локальную конечную точку. |
| ProtocolType | Дает тип протокола сокета. |
| RemoteEndPoint | Дает удаленную конечную точку сокета. |
| SocketType | Дает тип сокета. |
| Accept() | Создает новый сокет для обработки входящего запроса на соединение. |
| Bind() | Связывает сокет с локальной конечной точкой для ожидания входящих запросов на соединение. |
| Close() | Заставляет сокет закрыться. |
| Connect() | Устанавливает соединение с удаленным хостом. |
| GetSocketOption() | Возвращает значение SocketOption. |
| IOControl() | Устанавливает для сокета низкоуровневые режимы работы. Этот метод обеспечивает низкоуровневый доступ к лежащему в основе классу Socket. |
| Listen() | Помещает сокет в режим прослушивания (ожидания). Этот метод предназначен только для серверных приложений. |
| Receive() | Получает данные от соединенного сокета. |
| Poll() | Определяет статус сокета. |
| Select() | Проверяет статус одного или нескольких сокетов. |
| Send() | Отправляет данные соединенному сокету. |
| SetSocketOption() | Устанавливает опцию сокета. |
| Shutdown() | Запрещает операции отправки и получения данных на сокете. |

Пример сервера, построенного на основе сокетов, представлен ниже:

using System;

using System.Collections.Concurrent;

using System.Net;

using System.Net.Sockets;

using System.Text;

using System.Threading.Tasks;

using ClassLibraryCities;

using Newtonsoft.Json;

namespace Server

{

class Program

{

private static ConcurrentDictionary<string, City> \_cities = new ConcurrentDictionary<string, City>();

static void Main(string[] args)

{

// Устанавливаем для сокета локальную конечную точку

IPHostEntry ipHost = Dns.GetHostEntry("localhost");

IPAddress ipAddr = ipHost.AddressList[0];

IPEndPoint ipEndPoint = new IPEndPoint(ipAddr, 11000);

// Создаем сокет Tcp/Ip

Socket sListener = new Socket(ipAddr.AddressFamily, SocketType.Stream, ProtocolType.Tcp);

// Назначаем сокет локальной конечной точке и слушаем входящие сокеты

try

{

sListener.Bind(ipEndPoint);

sListener.Listen(10);

// Начинаем слушать соединения

while (true)

{

Console.WriteLine("Ожидаем соединение через порт {0}", ipEndPoint);

// Программа приостанавливается, ожидая входящее соединение

Socket socket = sListener.Accept();

Task taskSocket = new Task(Action, socket);

taskSocket.Start();

}

}

catch (Exception ex)

{

Console.WriteLine(ex.ToString());

}

finally

{

Console.ReadLine();

}

}

private static void Action(object o)

{

Socket socket = o as Socket;

if (socket != null)

{

while (true)

{

try

{

// Мы дождались клиента, пытающегося с нами соединиться

byte[] bytes = new byte[10240];

int bytesRec = socket.Receive(bytes);

string json = Encoding.UTF8.GetString(bytes, 0, bytesRec);

CityResponse response = new CityResponse { IsSuccess = false };

try

{

var request = JsonConvert.DeserializeObject<CityRequest>(json);

if (request != null)

{

response.Key = request.Key;

City city;

switch (request.Type)

{

case CityRequestType.Get:

if (\_cities.TryGetValue(request.Key, out city))

{

response.City = city;

response.IsSuccess = true;

}

else

{

response.ErrorMessage = "Ключ не найден";

}

break;

case CityRequestType.Add:

if (\_cities.ContainsKey(request.Key))

{

response.ErrorMessage = "Город с таким ключем уже существует";

}

else

{

\_cities.AddOrUpdate(request.Key, request.City, (s, city1) => request.City);

response.IsSuccess = true;

}

break;

case CityRequestType.Update:

if (\_cities.ContainsKey(request.Key))

{

\_cities.AddOrUpdate(request.Key, request.City, (s, city1) => request.City);

response.IsSuccess = true;

}

else

{

response.ErrorMessage = "Город с таким ключем не существует";

}

break;

case CityRequestType.Remove:

if (\_cities.ContainsKey(request.Key))

{

if (\_cities.TryRemove(request.Key, out city))

{

response.IsSuccess = true;

}

else

{

response.ErrorMessage = "Не удалось удалить город";

}

}

else

{

response.ErrorMessage = "Город с таким ключем не существует";

}

break;

default:

throw new ArgumentOutOfRangeException();

}

}

}

catch (Exception exception)

{

response.ErrorMessage = exception.Message;

}

// Показываем данные на консоли

Console.WriteLine("Полученный json: " + json);

// Отправляем ответ клиенту\

var jsonResponse = JsonConvert.SerializeObject(response);

byte[] msg = Encoding.UTF8.GetBytes(jsonResponse);

Console.Write("Отправленный json: " + jsonResponse);

socket.Send(msg);

}

catch (Exception)

{

break;

}

}

socket.Shutdown(SocketShutdown.Both);

socket.Close();

}

}

}

}

Пример клиента, взаимодействующего с сервером по сокетам, приведем ниже.

using System;

using System.Net;

using System.Net.Sockets;

using System.Text;

using ClassLibraryCities;

using Newtonsoft.Json;

namespace TestClient

{

class Program

{

static void Main(string[] args)

{

// Буфер для входящих данных

byte[] bytes = new byte[10240];

// Соединяемся с удаленным устройством

// Устанавливаем удаленную точку для сокета

IPHostEntry ipHost = Dns.GetHostEntry("localhost");

IPAddress ipAddr = ipHost.AddressList[0];

IPEndPoint ipEndPoint = new IPEndPoint(ipAddr, 11000);

Socket sender = new Socket(ipAddr.AddressFamily, SocketType.Stream, ProtocolType.Tcp);

// Соединяем сокет с удаленной точкой

sender.Connect(ipEndPoint);

while (true)

{

try

{

CityRequest request = null;

Console.Write("Действие (add, get): ");

string action = Console.ReadLine();

if (action == "add")

{

string title = Console.ReadLine();

request = new CityRequest

{

City = new City {Title = title},

Key = title,

Type = CityRequestType.Add

};

}

else if (action == "get")

{

string key = Console.ReadLine();

request = new CityRequest

{

Key = key,

Type = CityRequestType.Get

};

}

else

{

continue;

}

string jsonRequest = JsonConvert.SerializeObject(request);

byte[] msg = Encoding.UTF8.GetBytes(jsonRequest);

// Отправляем данные через сокет

sender.Send(msg);

// Получаем ответ от сервера

int bytesRec = sender.Receive(bytes);

Console.WriteLine("Запрос к серверу: {0}\n\n", jsonRequest);

Console.WriteLine("Ответ от сервера: {0}\n\n", Encoding.UTF8.GetString(bytes, 0, bytesRec));

}

catch (Exception)

{

break;

}

}

// Освобождаем сокет

sender.Shutdown(SocketShutdown.Both);

sender.Close();

}

}

}

**Задание на лабораторную работу**

1. Разработать многопоточный сетевой сервер, осуществляющий централизованное хранение справочника согласно варианту (табл. 3). Сервер должен поддерживать обработку следующих видов запросов: получение элемента справочника по ключу, добавление нового элемента в справочник (с ключом), удаление элемента справочника по ключу, обновление элемента справочника по ключу.
2. Дополнительно реализовать автосохранение справочника в JSON-файл (в отдельном потоке через интервал времени 10 сек).
3. При запуске сервера необходимо автоматически загружать элементы справочника из JSON-файла.
4. Разработать сетевой клиент в виде Windows Forms приложения, осуществляющее отправку запросов к серверу: добавление элемента в справочник, поиск элемента в справочнике, удаление элемента из справочника, обновление элемента справочника.

**Таблица 3**

***Варианты заданий***

|  |  |
| --- | --- |
| **Номер** | **Справочник** |
| 1. | Реки |
| 2. | Моря |
| 3. | Горные вершины |
| 4. | Озера |
| 5. | Острова |
| 6. | Улицы |
| 7. | Книги |
| 8. | Авторы |
| 9. | Фильмы |
| 10. | Кинотеатры |
| 11. | Спортивные клубы |
| 12. | Спортсмены |
| 13. | Автомобили |
| 14. | Самолеты |
| 15. | Банки |
| 16. | Продукты |
| 17. | Растения |
| 18. | Животные |
| 19. | Звезды |
| 20. | Валюты |

**Контрольные вопросы**

1. Какие классы и как используются для организации многопоточности?
2. Каким образом осуществляется передача данных между потоками и синхронизация?
3. Какие классы и как используются для организации сетевого взаимодействия?

# Список рекомендуемой литературы

1. Нейгел, К. C# 5.0 и платформа .NET 4.5 для профессионалов / К.Нейгел, Б.Ивьен, Дж.Глинн, М.Скиннер, К.Уотсон. – М.: Диалектика, 2013. – 1440 с.
2. Неш, Т. C# 2010: ускоренный курс для профессионалов / Т.Неш – М.: Вильямс, 2010. – 592 с.
3. Снелл, М. Microsoft Visual Studio 2008 / М.Снелл, Л.Пауэрс. – СПб: БХВ-Петербург, 2009. – 1200 с.
4. Троелсен, Э. Язык программирования С# 5.0 и платформа .NET 4.5 / Э.Троелсен. – 6-е изд. – М.: Вильямс, 2015. – 1312 с.
5. Рихтер, Дж. CLR via C#. Программирование на платформе Microsoft .NET Framework 4.5 на языке C# / Дж.Рихтер. – СПб.: Питер, 2016. – 896 с.

Программирование в среде Microsoft .Net. Создание многопоточных и сетевых приложений [Текст] + [Электронный ресурс]: Методические указания к выполнению лабораторной работы №10 для студентов очной формы обучения по направлениям подготовки 02.03.03 – «Математическое обеспечение и администрирование информационных систем», 09.03.01 – «Информатика и вычислительная техника» и 09.03.04 – «Программная инженерия». – Брянск: БГТУ, 2017. – 19 с.

КОРОСТЕЛЁВ ДМИТРИЙ АЛЕКСАНДРОВИЧ

Научный редактор Д.И. Булатицкий

Компьютерный набор Д.А. Коростелёв

Иллюстрации Д.А. Коростелёв

Подписано в печать 07.07.2017 г. Формат 60х84 1/16 Бумага офсетная. Офсетная печать. Усл.печ.л. 0,63 Уч.-изд.л. 0,63 Тираж 1 экз.

Брянский государственный технический университет

Кафедра «Информатика и программное обеспечение», тел. 56-09-84

241035, Брянск, бульвар 50 лет Октября, 7 БГТУ