**Первый простой пример**

Доступ к Интернету организуется на основе классов WebRequest и WebResponse.

Поэтому, прежде чем рассматривать этот процесс более подробно, было бы полезно

обратиться к простому примеру, демонстрирующему порядок доступа к Интернету

по принципу запроса и ответа. Глядя на то, как эти классы применяются на практике,

легче понять, почему они организованы именно так, а не как-то иначе.

В приведенном ниже примере программы демонстрируется простая, но весьма

типичная для Интернета операция получения гипертекстового содержимого из конкретного

веб-сайта. В данном случае содержимое получается из веб-сайта издательства

McGraw-Hill по адресу www.McGraw-Hill.com, но вместо него можно подставить

адрес любого другого веб-сайта. В этой программе гипертекстовое содержимое выводится

на экран монитора отдельными порциями по 400 символов, чтобы полученную

информацию можно было просматривать, не прибегая к прокрутке экрана.

(***glava26\_1***)

using System;

using System.Net;

using System.IO;

class NetDemo

{

static void Main()

{

int ch;

//first create request object WebRequest use certain URI

HttpWebRequest req = (HttpWebRequest)

WebRequest.Create("http://www.McGraw-Hill.com");

//second send request and get response from it

HttpWebResponse resp = (HttpWebResponse)

req.GetResponse();

//get stream from resp

Stream istrm = resp.GetResponseStream();

//and npw read and show hypertext content,

//got from URI. this content ouput on the screen

//by parts by 400 symbols. After every part

//nedd to press ENTER, to get next 400 symbols

for (int i = 1; ; i++)

{

ch = istrm.ReadByte();

if (ch == -1) break;

Console.Write((char)ch);

if ((i % 400) == 0)

{

Console.Write("\npress <Enter>.");

Console.ReadLine();

}

}

//close response. istrm will close as well

resp.Close();

}

}

Итак, выше приведена часть гипертекстового содержимого, полученного из вебсайта

издательства McGraw-Hill по адресу www.McGraw-Hill.com. В рассматриваемом

здесь примере программы это содержимое просто выводится в исходном виде на

экран посимвольно и не форматируется в удобочитаемом виде, как это обычно делается

в окне браузера.

Проанализируем данную программу построчно. Прежде всего обратите внимание

на использование в ней пространства имен System.Net. Как пояснялось ранее, в этом

пространстве имен находятся классы сетевого подключения к Интернету. Обратите

также внимание на то, что в данную программу включено пространство имен System.

IO, которое требуется для того, чтобы прочитать полученную на веб-сайте информацию,

используя объект типа Stream.

В начале программы создается объект типа WebRequest, содержащий требуемый

URI. Как видите, для этой цели используется метод Create(), а не конструктор. Это

статический член класса WebRequest. Несмотря на то что класс WebRequest является

абстрактным, это обстоятельство не мешает вызывать статический метод данного класса.

Метод Create() возвращает объект типа HttpWebRequest. Разумеется, его значение

требуется привести к типу HttpWebRequest, прежде чем присвоить его переменной

req ссылки на объект типа HttpWebRequest. На этом формирование запроса

завершается, но его еще нужно отправить по указанному URL

Для того чтобы отправить запрос, в рассматриваемой здесь программе вызывается

метод GetResponse() для объекта типа WebRequest. Отправив запрос, метод

GetResponse() переходит в состояние ожидания ответа. Как только ответ будет получен,

метод GetResponse() возвратит объект типа WebResponse, в котором инкапсулирован

ответ. Этот объект присваивается переменной resp. Но в данном случае ответ

принимается по протоколу HTTP, и поэтому полученный результат приводится к

типу HttpWebResponse. Среди прочего в ответе содержится поток, предназначаемый

для чтения данных из источника по указанному URL

Далее поток ввода получается в результате вызова метода GetResponseStream()

для объекта resp. Это стандартный объект класса Stream со всеми атрибутами и средствами,

необходимыми для организации потока ввода. Ссылка на этот поток присваивается

переменной istrm, с помощью которой данные могут быть прочитаны из источника

по указанному URI, как из обычного файла.

После этого в программе выполняется чтение данных из веб-сайта издательства

McGraw-Hill по адресу www.McGraw-Hill.com и последующий их вывод на экран.

А поскольку этих данных много, то они выводятся на экран отдельными порциями

по 400 символов, после чего в программе ожидается нажатие клавиши <Enter>, чтобы

продолжить вывод. Благодаря этому выводимые данные можно просматривать без

прокрутки экрана. Обратите внимание на то, что данные читаются посимвольно с помощью

метода ReadByte(). Напомним, что этот метод возвращает очередной байт из

потока ввода в виде значения типа int, которое требуется привести к типу char. По

достижении конца потока этот метод возвращает значение -1.

И наконец, ответный поток закрывается при вызове метода Close() для объекта

resp. Вместе с ответным потоком автоматически закрывается и поток ввода. Ответный

поток следует закрывать в промежутках между последовательными запросами. В противном

случае сетевые ресурсы могут быть исчерпаны, препятствуя очередному подключению

к Интернету.

И в заключение анализа рассматриваемого здесь примера следует обратить особое

внимание на следующее: для отображения гипертекстового содержимого, получаемого

от сервера, совсем не обязательно использовать объект типа HttpWebRequest или

HttpWebResponse. Ведь для решения этой задачи в данной программе оказалось достаточно

стандартных методов, определенных в классах WebRequest и WebResponse,

и не потребовалось прибегать к специальным средствам протокола HTTP. Следовательно,

вызовы методов Create() и GetResponse() можно было бы написать следующим

образом.

// Сначала создать объект запроса типа WebRequest по указанному URI.

WebRequest req = WebRequest.Create("http://www.McGraw-Hill.com");

// Затем отправить сформированный запрос и получить на него ответ.

WebResponse resp = req.GetResponse();

В тех случаях, когда не требуется приведение к конкретному типу реализации протокола,

лучше пользоваться классами WebRequest и WebResponse, так как это дает

возможность менять протокол, не оказывая никакого влияния на код программы. Но

поскольку во всех примерах, приведенных в этой главе, используется протокол HTTP,

то в ряде примеров демонстрируются специальные средства этого протокола из классов

HttpWebRequest и HttpWebResponse.

**Обработка сетевых ошибок**

Программа из предыдущего примера составлена верно, но она совсем не защищена

от простейших сетевых ошибок, которые способны преждевременно прервать ее выполнение.

Конечно, для программы, служащей в качестве примера, это не так важно,

как для реальных приложений. Для полноценной обработки сетевых исключений,

которые могут быть сгенерированы программой, необходимо организовать контроль

вызовов методов Create(), GetResponse() и GetResponseStream(). Следует особо

подчеркнуть, что генерирование конкретных исключений зависит от используемого

протокола. И ниже речь пойдет об ошибках, которые могут возникнуть при использовании

протокола HTTP, поскольку средства сетевого подключения к Интернету, доступные

в С#, рассматриваются в настоящей главе на примере именно этого протокола.

**Исключения, генерируемые методом Create()**

Метод Create(), определенный в классе WebRequest, может генерировать четыре

исключения. Так, если протокол, указываемый в префиксе URI, не поддерживается,

то генерируется исключение NotSupportedException. Если формат URI оказывается

недействительным, то генерируется исключение UriFormatException. А если у

пользователя нет соответствующих полномочий для доступа к запрашиваемому сетевому

ресурсу, то генерируется исключение System.Security.SecurityException.

Кроме того, метод Create() генерирует исключение ArgumentNullException, если

он вызывается с пустой ссылкой, хотя этот вид ошибки не имеет непосредственного

отношения к сетевому подключению.

**Исключения, генерируемые методом GetResponse()**

При вызове метода GetResponse() для получения ответа по протоколу HTTP

может произойти целый ряд ошибок. Эти ошибки представлены следующими исключениями:

InvalidOperationException, ProtocolViolationException,

NotSupportedException и WebException. Наибольший интерес среди них вызывает

исключение WebException.

У исключения WebException имеются два свойства, связанных с сетевыми ошибками:

Response и Status. С помощью свойства Response можно получить ссылку на

объект типа WebResponse в обработчике исключений. Для соединения по протоколу

HTTP этот объект описывает характер возникшей ошибки. Свойство Response объявляется

следующим образом.

public WebResponse Response { get; }

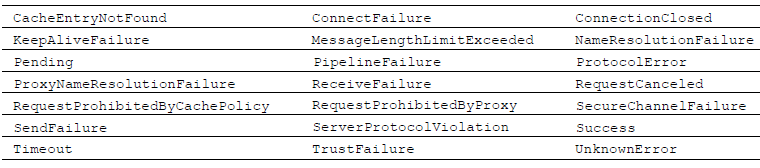
Когда возникает ошибка, то с помощью свойства Status типа WebException можно

выяснить, что именно произошло. Это свойство объявляется следующим образом:

public WebExceptionStatus Status { get; }

где WebExceptionStatus — это перечисление, которое содержит приведенные ниже

значения.



Как только будет выяснена причина ошибки, в программе могут быть предприняты

соответствующие действия.

**Исключения, генерируемые методом GetResponseStream()**

Для соединения по протоколу HTTP метод GetResponseStream() из класса

WebResponse может сгенерировать исключение ProtocolViolationException, которое

в целом означает, что в работе по указанному протоколу произошла ошибка.

Что же касается метода GetResponseStream(), то это означает, что ни один из действительных

ответных потоков недоступен. Исключение ObjectDisposedException

генерируется в том случае, если ответ уже утилизирован. А исключение IOException,

конечно, генерируется при ошибке чтения из потока, в зависимости от того, как организован

ввод данных.

**Обработка исключений**

В приведенном ниже примере программы демонстрируется обработка всевозможных

сетевых исключений, которые могут возникнуть в связи с выполнением программы

из предыдущего примера, в которую теперь добавлены соответствующие обработчики

исключений.

(***glava26\_2***)

class NetDemo

{

static void Main()

{

int ch;

try

{

//first create request object WebRequest use certain URI

HttpWebRequest req = (HttpWebRequest)

WebRequest.Create("http://www.McGraw-Hill.com/sdfsdf");

//second send request and get response from it

HttpWebResponse resp = (HttpWebResponse)

req.GetResponse();

//get stream from resp

Stream istrm = resp.GetResponseStream();

//and npw read and show hypertext content,

//got from URI. this content ouput on the screen

//by parts by 400 symbols. After every part

//nedd to press ENTER, to get next 400 symbols

for (int i = 1; ; i++)

{

ch = istrm.ReadByte();

if (ch == -1) break;

Console.Write((char)ch);

if ((i % 400) == 0)

{

Console.Write("\npress <Enter>.");

Console.ReadLine();

}

}

//close response. istrm will close as well

resp.Close();

}

catch (WebException exc)

{

Console.WriteLine("Web error: " + exc.Message

+ "\nState code: " + exc.Status);

}

catch (ProtocolViolationException exc)

{

Console.WriteLine("Protocol error: " + exc.Message);

}

catch (UriFormatException exc)

{

Console.WriteLine("Format error URI: " + exc.Message);

}

catch (NotSupportedException exc)

{

Console.WriteLine("Uknown Protocol: " + exc.Message);

}

catch (IOException exc)

{

Console.WriteLine("IO error: " + exc.Message);

}

catch (System.Security.SecurityException exc)

{

Console.WriteLine("Security error: " + exc.Message);

}

catch (InvalidOperationException exc)

{

Console.WriteLine("Wrong operation: " + exc.Message);

}

}

}

**Практический пример создания программы MiniCrawler**

Для того чтобы показать, насколько просто программировать для Интернета средствами

классов WebRequest и WebReponse, обратимся к разработке скелетного варианта

*поискового робота* под названием MiniCrawler. Поисковый робот представляет

собой программу последовательного перехода от одной ссылки на сетевой ресурс к

другой. Поисковые роботы применяются в поисковых механизмах для каталогизации

содержимого. Разумеется, поисковый робот MiniCrawler не обладает такими развитыми

возможностями, как те, что применяются в поисковых механизмах. Эта программа

начинается с ввода пользователем конкретного адреса URI, по которому затем читается

содержимое и осуществляется поиск в нем ссылки. Если ссылка найдена, то программа

запрашивает пользователя, желает ли он перейти по этой ссылке к обнаруженному сетевому

ресурсу, найти другую ссылку на имеющейся странице или выйти из программы.

Несмотря на всю простоту такого алгоритма поиска сетевых ресурсов, он служит

интересным и наглядным примером доступа к Интернету средствами С#.

Программе MiniCrawler присущ ряд ограничений. Во-первых, в ней обнаруживаются

только абсолютные ссылки, указываемые по гипертекстовой команде href="http.

Относительные ссылки при этом не обнаруживаются. Во-вторых, возврат к предыдущей

ссылке в программе не предусматривается. И в-третьих, в ней отображаются только

ссылки, но не окружающее их содержимое. Несмотря на все указанные ограничения

данного скелетного варианта поискового робота, он вполне работоспособен и может

быть без особых хлопот усовершенствован для решения других задач. На самом деле

добавление новых возможностей в программу MiniCrawler — это удобный случай освоить

на практике сетевые классы и узнать больше о сетевом подключении к Интернету.

Ниже приведен полностью исходный код программы MiniCrawler.

(***MiniCrawler***)

class MiniCrawler

{

//find link in string

static string FindLink(string htmlstr, ref int startloc)

{

int i;

int start, end;

string uri = null;

i = htmlstr.IndexOf("href=\"http", startloc,

StringComparison.OrdinalIgnoreCase);

if (i != -1)

{

start = htmlstr.IndexOf('"', i) + 1;

end = htmlstr.IndexOf('"', start);

uri = htmlstr.Substring(start, end - start);

startloc = end;

}

return uri;

}

static void Main(string[] args)

{

string link = null;

string str;

string answer;

int curloc; //current position answer

if (args.Length != 1)

{

Console.WriteLine("Do: MiniCrawler <uri>");

return;

}

string uristr = args[0]; //current uri

HttpWebResponse resp = null;

try

{

do

{

Console.WriteLine("Go to url " + uristr);

//create request object WebRequest

HttpWebRequest req = (HttpWebRequest)

WebRequest.Create(uristr);

uristr = null; //forbid use this uri again

//send request and get response

resp = (HttpWebResponse)req.GetResponse();

//get stream IO from response

Stream istrm = resp.GetResponseStream();

//envelope stream in StreamReader

StreamReader rdr = new StreamReader(istrm);

//read whole page

str = rdr.ReadToEnd();

curloc = 0;

do

{

//find dnext uri

link = FindLink(str, ref curloc);

if (link != null)

{

Console.WriteLine("Found a link: " + link);

Console.Write("Go to link, Keep looking, Exit?");

answer = Console.ReadLine();

if (string.Equals(answer, "G", StringComparison.OrdinalIgnoreCase))

{

uristr = string.Copy(link);

}

else if (string.Equals(answer, "E", StringComparison.OrdinalIgnoreCase))

{

break;

}

else if (string.Equals(answer, "K", StringComparison.OrdinalIgnoreCase))

{

Console.WriteLine("Find next link.");

}

else

{

Console.WriteLine("No more links found.");

break;

}

}

} while (link.Length > 0);

//close respond source

if (resp != null) resp.Close();

} while (uristr != null);

}

catch (WebException exc)

{

Console.WriteLine("Web error: " + exc.Message);

}

catch (ProtocolViolationException exc)

{

Console.WriteLine("Protocol error: " + exc.Message);

}

catch (UriFormatException exc)

{

Console.WriteLine("Uri format error: " + exc.Message);

}

catch (NotSupportedException exc)

{

Console.WriteLine("Uknown protocol error: " + exc.Message);

}

catch (IOException exc)

{

Console.WriteLine("IO error: " + exc.Message);

}

finally

{

if (resp != null) resp.Close();

}

Console.WriteLine("Exit MiniCrawler.");

}

}

Рассмотрим подробнее работу программы MiniCrawler. Она начинается с ввода

пользователем конкретного URI в командной строке. В методе Main() этот URI сохраняется

в строковой переменной uristr. Затем по указанному URI формируется запрос,

и переменной uristr присваивается пустое значение, указывающее на то, что данный

URI уже использован. Далее отправляется запрос и получается ответ. После этого содержимое

читается из потока ввода, возвращаемого методом GetResponseStream()

и заключаемого в оболочку класса StreamReader. Для этой цели вызывается метод

ReadToEnd(), возвращающий все содержимое в виде строки из потока ввода.

Далее программа осуществляет поиск ссылки в полученном содержимом. Для этого

вызывается статический метод FindLink(), определяемый в программе MiniCrawler.

Этот метод вызывается со строкой содержимого и исходным положением, с которого

начинается поиск в полученном содержимом. Эти значения передаются методу

FindLink() в виде параметров htmlstr и startloc соответственно. Обратите внимание

на то, что параметр startloc относится к типу ref. Сначала в методе FindLink()

создается копия строки содержимого в нижнем регистре, а затем осуществляется поиск

подстроки href="http, обозначающей ссылку. Если эта подстрока найдена, то

URI копируется в строковую переменную uri, а значение параметра startloc обновляется

и становится равным концу ссылки. Но поскольку параметр startloc относится

к типу ref, то это приводит к обновлению соответствующего аргумента метода

Main(), активизируя поиск с того места, где он был прерван. В конечном итоге

возвращается значение переменной uri. Эта переменная инициализирована пустым

значением, и поэтому если ссылка не найдена, то возвращается пустая ссылка, обозначающая

неудачный исход поиска.

Если ссылка, возвращаемая методом FindLink(), не является пустой, то она отображается

в методе Main(), и далее программа запрашивает у пользователя очередные

действия. Пользователю предоставляются одна из трех следующих возможностей:

перейти по найденной ссылке, нажав клавишу <П>, искать следующую ссылку в имеющемся

содержимом, нажав клавишу <И>, иди же выйти из программы, нажав клавишу

<В>. Если пользователь нажмет клавишу <П>, то программа осуществит переход

по найденной ссылке и получит новое содержимое по этой ссылке. После этого поиск

очередной ссылки будет начат уже в новом содержимом. Этот процесс продолжается

до тех пор, пока не будут исчерпаны все возможные ссылки.

В качестве упражнения вы сами можете усовершенствовать программу MiniCrawler,

дополнив ее, например, возможностью перехода по относительным ссылкам. Сделать

это не так уж и трудно. Кроме того, вы можете полностью автоматизировать поисковый

робот, чтобы он сам переходил по найденной ссылке без вмешательства со стороны

пользователя, начиная со ссылки, обнаруженной на самой первой странице полученного

содержимого, и продолжая переход по ссылкам на новых страницах. Как только

будет достигнут тупик, поисковый робот должен вернуться на один уровень назад, найти

следующую ссылку и продолжить переход по ссылке. Для организации именно такого

алгоритма работы программы вам потребуется стек, в котором должны храниться

идентификаторы URI и текущее состояние поиска в строке URI. С этой целью можно,

в частности, воспользоваться коллекцией класса Stack. В качестве более сложной, но

интересной задачи попробуйте организовать вывод ссылок в виде дерева.