# Работа с файлами»

## 1. Потоковые классы. Классификация потоков.

Под **файлом** обычно подразумевается именованная информация на внешнем носителе, например на жестком или гибком магнитном диске. Логически файл можно представить как конечное количество последовательных байтов, поэтому такие устройства, как дисплей, клавиатура и принтер, также можно рассматривать как частные случаи файлов. Передача данных с внешнего устройства в оперативную память называется **чтением***,* или **вводом***,* обратный процесс — **записью***,* или **выводом***.*

Обмен данными реализуется с помощью **потоков**. Поток— это абстрактное понятие, относящееся к любому переносу данных от источника к приемнику.

Поток определяется как последовательность байтов и не зависит от конкретного устройства, с которым производится обмен (оперативная память, файл на диске, клавиатура или принтер).

Обмен с потоком для повышения скорости передачи данных производится, как правило, через специальную область оперативной памяти — **буфер***.* Буфер выделяется для каждого открытого файла. При записи в файл вся информация сначала направляется в буфер и там накапливается до тех пор, пока весь буфер не заполнится. Только после этого или после специальной команды сброса происходит передача данных на внешнее устройство. При чтении из файла данные вначале считываются в буфер, причем не столько, сколько запрашивается, а сколько помещается в буфер.

Механизм буферизации позволяет более быстро и эффективно обмениваться информацией с внешними устройствами.

Для поддержки потоков библиотека .NET содержит иерархию классов.

**Основные классы пространства имен System.IO**

|  |  |
| --- | --- |
| Класс | Описание |
| BinaryReader,  BinaryWriter | Чтение и запись значений простых встроенных типов (целочисленных, логических, строковых и т. п.) во внутренней форме представления |
| BufferedStream | Временное хранение потока байтов (например, для последующего переноса в постоянное хранилище) |
| Di rectory,  DirectoryInfo,  File,  FileInfo | Работа с каталогами или физическими файлами: создание, удаление, получение свойств. Возможности классов Fi1е и Directory реализованы в основном в виде статических методов. Аналогичные классы DirectoryInfo и FileInfo используют обычные методы |
| FileStream | Произвольный (прямой) доступ к файлу, представленному как поток байтов |
| MemoryStream | Произвольный доступ к потоку байтов в оперативной памяти |
| StreamWriter, StreamReader | Чтение из файла и запись в файл текстовой информации (произвольный доступ не поддерживается) |
| StringWriter,  StringReader | Работа с текстовой информацией в оперативной памяти |

Таким образом, выполнять обмен с внешними устройствами можно на уровне:

• двоичного представления данных (BinaryReader, BinaryWriter);

• байтов (FileStream);

• текста, то есть символов (StreamWriter, StreamReader).

В .NET используется кодировка Unicode, в которой каждый символ кодируется двумя байтами. Классы, работающие с текстом, являются оболочками классов, использующих байты, и автоматически выполняют перекодирование из байтов в символы и обратно.

Двоичные и байтовые потоки хранят данные в том же виде, в котором они представлены в оперативной памяти, то есть при обмене с файлом происходит побитовое копирование информации. Двоичные файлы применяются не для просмотра их человеком, а для использования в программах.

Доступ к файлам может быть **последовательным**, когда очередной элемент можно прочитать (записать) только после аналогичной операции с предыдущим элементом, и **прямым**, при котором выполняется чтение (запись) произвольного элемента по заданному адресу. Текстовые файлы позволяют выполнять только последовательный доступ, в двоичных и байтовых потоках можно использовать оба метода.

Прямой доступ в сочетании с отсутствием преобразований обеспечивает высокую

скорость получения нужной информации.

Использование классов файловых потоков в программе предполагает следующие операции:

1. Создание потока и связывание его с физическим файлом.
2. Обмен (ввод-вывод).
3. Закрытие файла.

Каждый класс файловых потоков содержит несколько вариантов конструкторов, с помощью которых можно создавать объекты этих классов различными способами и в различных режимах.

Например, файлы можно открывать только для чтения, только для записи или для чтения и записи. Эти режимы доступа к файлу содержатся в перечислении FileAccess, определенном в пространстве имен System.IO.

|  |  |
| --- | --- |
| Значение | Описание |
| Read | Открыть файл только для чтения |
| ReadWrite | Открыть файл для чтения и записи |
| Write | Открыть файл только для записи |

Возможные режимы открытия файла определены в перечислении FileMode

|  |  |
| --- | --- |
| Значение | Описание |
| Append | Открыть файл, если он существует, и установить текущий указатель в конец файла. Если файл не существует, создать новый файл. |
| Create | Создать новый файл. Если в каталоге уже существует файл с таким же именем, он будет стерт. |
| CreateNew | Создать новый файл. Если в каталоге уже существует файл с таким же именем, возникает исключение IOException. |
| Open | Открыть существующий файл. |
| OpenOrCreate | Открыть файл, если он существует. Если нет, создать файл с таким именем |
| Truncate | Открыть существующий файл. После открытия он должен быть обрезан до нулевой длины. |

Режим FileMode.Append можно использовать только совместно с доступом типа FileAccess.Write, то есть для файлов, открываемых для записи.

Режимы совместного использованияфайла различными пользователями определяет перечисление FileShare.

|  |  |
| --- | --- |
| Значение | Описание |
| None | Совместное использование открытого файла запрещено. Запрос  на открытие данного файла завершается сообщением об ошибке |
| Read | Позволяет открывать файл для чтения одновременно нескольким  пользователям. Если этот флаг не установлен, запросы на открытие  файла для |
| ReadWrite | Позволяет открывать файл для чтения и записи одновременно  нескольким пользователям |
| Write | Позволяет открывать файл для записи одновременно нескольким  пользователям |

## 2. Потоки байтов (FileStream)

Ввод-вывод в файл на уровне байтов выполняется с помощью класса FileStream,который является наследником абстрактного класса Stream, определяющего набор стандартных операций с потоками.

Элементы класса Stream.

|  |  |
| --- | --- |
| Значение | Описание |
| BeginRead,  BeginWrite | Начать асинхронный ввод или вывод |
| CanRead,  CanSeek,  CanWrite | Свойства, определяющие, какие операции поддерживает поток: чтение, прямой доступ и/или запись |
| Сlose | Закрыть текущий поток и освободить связанные с ним ресурсы (сокеты, указатели на файлы и т. п.) |
| EndRead,  EndWrite | Ожидать завершения асинхронного ввода; закончить асинхронный вывод |
| Flush | Записать данные из буфера в связанный с потоком источник данных и очистить буфер. Если для данного потока буфер не используется, то этот метод ничего не делает |
| Length | Возвратить длину потока в байтах |
| Position | Возвратить текущую позицию в потоке |
| Read,  ReadByte | Считать последовательность байтов (или один байт) из текущего потока и переместить указатель в потоке на количество считанных байтов |
| Seek | Установить текущий указатель потока на заданную позицию |
| SetLength | Установить длину текущего потока |
| Write,  WriteByte | Записать последовательность байтов (или один байт) в текущий потоки переместить указатель в потоке на количество записанных байтов |

Текущая позиция в потоке первоначально устанавливается на начало файла (для любого режима открытия, кроме Append) и сдвигается на одну позицию при записи каждого байта.

Для установки желаемой позиции чтения используется метод Seek, имеющий два параметра: первый задает смещение в байтах относительно точки отсчета, задаваемой вторым. Точки отсчета задаются константами перечисления SeekOrigin:

начало файла — Begin,

текущая позиция — Current,

конец файла — End.

Пример.

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.IO;

namespace prim1

{

class Program

{

static void Main(string[] args)

{

FileStream f = new FileStream("test.txt",FileMode.Create, FileAccess.ReadWrite);

f.WriteByte(100);

byte[] x = new byte[10];

Random rand=new Random();

for (byte i = 0; i < 10; ++i )

{

x[i] = (byte)rand.Next(10);

f.WriteByte(i); // записывается 10 чисел от 0 до 9

}

f.Write(x, 0, 5 );//записывается 5 элементов массива

byte[] у = new byte[20];//еще один массив

f.Seek(0, SeekOrigin.Begin);

f.Read( у,0, 20 );

foreach (byte elem in у) Console.Write(" " + elem);

Console.WriteLine();

f.Seek(5, SeekOrigin.Begin);

int а = f.ReadByte();

Console.WriteLine(а);

а = f.ReadByte(); // чтение 6-го элемента

Console.WriteLine(а );

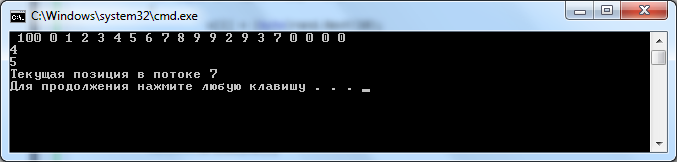
Console.WriteLine( "Текущая позиция в потоке " + f.Position );

f.Close();

}

}

}



Операции по открытию файлов могут завершиться неудачно, например, при ошибке в имени существующего файла или при отсутствии свободного места на диске, поэтому *рекомендуется всегда контролировать результаты* этих операций.

В случае непредвиденных ситуаций среда выполнения генерирует различные исключения, обработку которых следует предусмотреть в программе, например:

* FileNotFoundException, если файла с указанным именем в указанном каталоге не существует;
* DirectoryNotFoundException, если не существует указанный каталог;
* Argument Except ion, если неверно задан режим открытия файла;
* IOException, если файл не открывается из-за ошибок ввода-вывода.

При закрытии файла освобождаются все связанные с ним ресурсы, например, для файла, открытого для записи, в файл выгружается содержимое буфера. Поэтому рекомендуется всегда закрывать файлы после окончания работы, в особенности файлы, открытые для записи. Если буфер требуется выгрузить, не закрывая файл, используется метод Flush.

## 3. Потоки символов (StreamWriter, StreamReader)

Символьные потоки **StreamWriter** и **StreamReader** работают с Unicode-символами, следовательно, ими удобнее всего пользоваться для работы с файлами, предназначенными для восприятия человеком. Эти потоки являются наследниками классов **TextWrite**r и **TextReader** соответственно, которые обеспечивают их большей частью функциональности.

КлассTextWriter

|  |  |
| --- | --- |
| Значение | Описание |
| Close | Закрыть файл и освободить связанные с ним ресурсы. Если в процессе записи используется буфер, он будет автоматически очищен |
| Fl ush | Очистить все буферы для текущего файла и записать накопленные в них данные в место их постоянного хранения. Сам файл при этом не закрывается |
| NewLine | Используется для задания последовательности символов, означающих начало новой строки. |
| Write | Записать фрагмент текста в поток |
| WriteLinе | Записать строку в поток и перейти на другую строку |

Класс TextReader

|  |  |
| --- | --- |
| Значение | Описание |
| Peek | Возвратить следующий символ, не изменяя позицию указателя в файле |
| Read | Считать данные из входного потока |
| ReadBlock | Считать из входного потока указанное пользователем количество символов и записать их в буфер, начиная с заданной позиции |
| ReadLine | Считать строку из текущего потока и возвратить ее как значение типа string. Пустая строка (null) означает конец файла (EOF) |
| ReadToEnd | Считать все символы до конца потока, начиная с текущей позиции, и возвратить считанные данные как одну строку типа string |

Пример 1.

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.IO;

namespace ConsoleApplication1

{

class Program

{

static void Main(string[] args)

{

try

{

StreamWriter f = new StreamWriter("text.txt");

f.WriteLine("Вывод в текстовый файл:");

double a = 12234;

int b = 29;

f.WriteLine("a = {0,6:C}\n b = {1}", a, b);

f.Close();

}

catch (Exception e)

{

Console.WriteLine("Error: " + e.Message);

return;

}

//чтение

try

{

StreamReader f = new StreamReader("text.txt" );

string s = f.ReadToEnd();

Console.WriteLine(s);

f.Close();

}

catch (FileNotFoundException e )

{

Console.WriteLine( e.Message );

Console.WriteLine("Проверьте правильность имени файла!");

return;

}

catch( Exception e )

{

Console.WriteLine("Error");

return;

}

//построчное чтение

try

{

StreamReader f = new StreamReader( "text.txt" );

string s;

long i = 0;

while ( ( s = f.ReadLine() ) != null )

Console.WriteLine( ++i+":"+ s );

f.Close();

}

catch( FileNotFoundException e )

{

Console.WriteLine( e.Message );

Console.WriteLine( "Проверьте правильность имени файла!" );

return;

}

catch ( Exception e )

{

Console.WriteLine( "Error: " + e.Message );

return;

}

}

}

}

Пример 2

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.IO;

namespace ConsoleApplication2

{

class Program

{

static void Main(string[] args)

{

try

{

Random r=new Random();

StreamWriter f = new StreamWriter("numbers.txt");

for (int i = 0; i < 3; i++)

{

for(int j=0;j<3;j++)

{

int a = r.Next(10);

if (j != 2)

f.Write(a + " ");

else

f.Write(a);

}

if(i!=2) f.WriteLine();

}

f.Close();

}

catch (Exception e)

{

Console.WriteLine("Error: " + e.Message);

return;

}

try

{

StreamReader f = new StreamReader("numbers.txt" );

string s;

const int n = 20;

int[] a = new int[n];

string[] buf;

while ((s = f.ReadLine() ) != null )

{

buf = s.Split( ' ' ) ;

long sum = 0;

for ( int i = 0; i < buf.Length; ++i )

{

a [ i ] = Convert.ToInt32( buf[i] );

sum += a [ i ] ;

}

Console.WriteLine( "{0} сумма: {1}", s, sum );

}

f.Close();

}

catch( FileNotFoundException e )

{

Console.WriteLine( e.Message );

Console.WriteLine( "Проверьте правильность имени файла!");

return;

}

catch( Exception e )

{

Console.WriteLine("Error; " + e.Message);

return;

}

}

}

}

## Двоичные файлы (BinaryWriter, BinaryReader)

**Двоичные файлы** (BinaryWriter, BinaryReader) хранят данные в том же виде, в котором они представлены в оперативной памяти, то есть во внутренней форме представления. Двоичные файлы применяются не для просмотра их человеком, а для использования в программах.

Выходной поток BinaryWriter поддерживает произвольный доступ, то есть имеется возможность выполнять запись в произвольную позицию двоичного файла. Двоичный файл открывается на основе базового потока, в качестве которого чаще всего используется поток FileStream. Входной двоичный поток содержит перегруженные методы чтения для всех простых встроенных типов данных.

**Основные методы двоичных потоков.**

Класс BinaryWriter

|  |  |
| --- | --- |
| Значение | Описание |
| BaseStream | Базовый поток, с которым работает объект BinaryWriter |
| Close | Закрыть поток |
| Fl ush | Очистить все буферы для текущего файла и записать накопленные в них данные в место их постоянного хранения. Сам файл при этом не закрывается |
| Seek | Установить позицию в текущем потоке |
| Write | Записать фрагмент текста в поток |

Класс BinaryReader

|  |  |
| --- | --- |
| Значение | Описание |
| BaseStream | Базовый поток, с которым работает объект BinaryWriter |
| Close | Закрыть поток |
| Peek Char | Возвратить следующий символ, не изменяя позицию указателя в файле |
| Read | Считать данные из входного потока и сохранить в массиве, передаваемом как входной параметр |
| ReadХХХХ | Считать из входного потока данные определенного типа |

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.IO;

namespace ConsoleApplication3

{

class Program

{

static void Main(string[] args)

{

try

{

BinaryWriter fout = new BinaryWriter(new FileStream( "binary", FileMode.Create) );

double d = 0;

while ( d < 4 )

{

fout.Write( d );

d += 0.33;

}

fout.Seek( 16, SeekOrigin.Begin ); // второй элемент файла

fout.Write( 8888d );

fout.Close();

}

catch( Exception e )

{

Console.WriteLine("Error: " + e.Message );

return;

}

try

{

FileStream f =new FileStream("binary", FileMode.Open );

BinaryReader fin = new BinaryReader( f );

long n = f.Length / 8; // количество чисел в файле

double[] x = new double[n];

long i = 0;

try

{

while( true ) x[i++] = fin.ReadDouble(); // чтение

}

catch ( EndOfStreamException e ) {}

foreach( double d in x ) Console.Write( " " + d ); // вывод

fin.Close();

f.Close();

}

catch ( FileNotFoundException e )

{

Console.WriteLine( e.Message );

Console.WriteLine( "Проверьте правильность имени файла!" );

return;

}

catch ( Exception e )

{

Console.WriteLine( "Error: " + e.Message );

return;

}

}

}

}

## Сериализация.

3. Сохранение объектов (сериализация).

В С# есть возможность сохранять на внешних носителях не только данные примитивных типов, но и объекты. Сохранение объектов называется сериализацией, а восстановление сохраненных объектов — десериализацией. При сериализации объект преобразуется в линейную последовательность байтов. Это сложный процесс, поскольку объект может включать множество унаследованных полей и ссылки на вложенные объекты, которые, в свою очередь, тоже могут состоять из объектов сложной структуры.

К счастью, сериализация выполняется автоматически, достаточно просто пометить класс как сериализуемый с помощью атрибута [Serializable]. Атрибуты — это дополнительные

сведения о классе, которые сохраняются в его метаданных. Те поля, которые сохранять не требуется, помечаются атрибутом [NonSerialized].

Объекты можно сохранять в одном из двух форматов: двоичном или SOAP (в виде XML-файла). В первом случае следует подключить к программе пространство имен System.Runtime.Serialization.Formatters.Binary, во втором — пространство System.Runtime.Serialization.Formatters.Soap.

Рассмотрим *сохранение объектов в двоичном формате.* Для этого используется класс BinaryFormatter, в котором определены два метода:

* Serialize( поток, объект ) - сохраняет заданный объект в заданном потоке;

Deserialize (поток ) - восстанавливает объект из заданного потока.

Для *сохранения объекта в двоичном формате необходимо:*

1. Подключить к программе пространство имен System.Runtime.Serialization.

Formatters.Binary.

2. Пометить сохраняемый класс и связанные с ним классы атрибутом [Serializable].

3. Создать поток и связать его с файлом на диске или с областью оперативной памяти.

4. Создать объект класса BinaryFormatter.

5. Сохранить объекты в потоке.

6. Закрыть файл.

**Пример.**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.IO;

using System.Runtime.Serialization.Formatters.Binary;

namespace ConsoleApplication5

{

[Serializable]

struct Person

{

public string name;

public int age;

[NonSerialized]

DateTime date;//дата создания объекта

public Person(string N, int A)

{

name = N; age = A; date = DateTime.Now;

}

public void Show()

{

Console.WriteLine(Name + ", " + Age + ", объект создан " + date);

}

public void Input()

{

Console.Write("Имя:"); name = Console.ReadLine();

Console.Write("Возраст"); age = Convert.ToInt32(Console.ReadLine());

}

}

class Program

{

static void Main(string[] args)

{

Person p1 = new Person();

p1.Show();

Person p2 = new Person("Иванов", 20);

p2.Show();

p1.Input();

p1.Show();

FileStream f1 = new FileStream("Demo.bin", FileMode.Create);

BinaryFormatter bf = new BinaryFormatter();

bf.Serialize(f1, p1); // сохранение объекта в потоке f

bf.Serialize(f1, p2); // сохранение объекта в потоке f

Person[] mas1 = { p1, p2 };

bf.Serialize(f1, mas1);

f1.Close();

Console.WriteLine("Объекты сохранены в поток");

Console.WriteLine("Выгружаем объекты из потока");

FileStream f2 = new FileStream("Demo.bin", FileMode.Open);

Person d = (Person)bf.Deserialize(f2); // восстановление объекта

d.Show();

Person x = (Person)bf.Deserialize(f2);x.Show();

Person[]mas2 = (Person[])bf.Deserialize(f2);

foreach (Person p in mas2) p.Show();

f2.Close();

}

}

}