

Многопоточность

Большинство приложений сегодня строятся на многопоточной структуре то есть запуск отдельных потоков процессов выполняющих большой объем операций, это еще более стало актуальным с появлением мультитядерных и мультипроцессорных систем.

В C# работа с потоками происходит через класс **Thread**.

Сейчас создадим класс который будет выполнять некоторую большую работу запустим два экземпляра данной работы и посмотрим их параллельное выполнение

```
namespace ConsoleApplication3
{
    class Program
    {
        static void Main(string[] args)
        {
            car mers = new car("Мерс");
            car bmw = new car("Бэха");
            System.Threading.Thread t1 = new System.Threading.Thread(new Sys
tem.Threading.ThreadStart(mers.Run));
            System.Threading.Thread t2 = new System.Threading.Thread(new Sys
tem.Threading.ThreadStart(bmw.Run));
            t1.Start();
            t2.Start();
        }
    }
    class car
    {
        string _name;
        public car(string name) { this._name = name; }
        public void Run()
        {
            for (int i=0;i<10000;i++)
                Console.WriteLine(this._name+" "+i);
        }
    }
}
```

Итак, создается поток, в конструкторе указывается метод который будет выполняться в данном потоке. Запуск потока происходит при выполнении метода **Start** экземпляра класса **thread** .

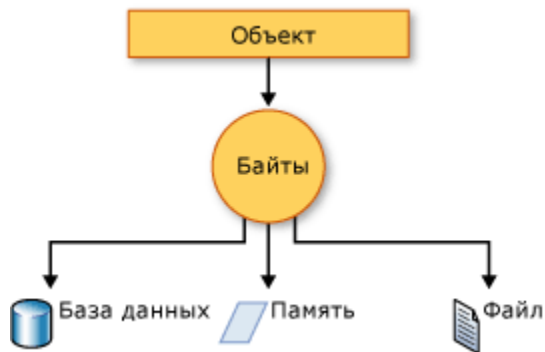
На скриншоте можете посмотреть как потоки борются за процессорное время



```
C:\WINDOWS\system32\cmd.exe
Мерс 9903
Мерс 9904
Мерс 9905
Бэха 9816
Бэха 9817
Бэха 9818
Бэха 9819
Бэха 9820
Бэха 9821
Бэха 9822
Бэха 9823
Бэха 9824
Бэха 9825
Бэха 9826
Бэха 9827
Бэха 9828
Бэха 9829
Бэха 9830
Бэха 9831
Мерс 9906
Мерс 9907
Мерс 9908
Мерс 9909
Мерс 9910
Мерс 9911
```

Сериализация


Сериализация представляет собой процесс преобразования объекта в поток байтов для хранения объекта или передачи его в память, базу данных или файл. Ее основное назначение — сохранить состояние объекта для того, чтобы иметь возможность воссоздать его при необходимости. Обратный процесс называется **десериализацией**.



Объект сериализуется в поток, который переносит не только данные, но сведения о типе объекта, такие как его версию, язык и региональные параметры, а также имя сборки. Из этого потока объект можно сохранить в базе данных, файле или памяти.

Пространство имен **System.Runtime.Serialization** содержит классы, необходимые для сериализации и десериализации объектов. Кроме того, для того, чтобы сериализация стала возможной необходимо также объявление пространств имен:

```
using System.Runtime.Serialization.Formatters.Binary;
using System.Runtime.Serialization;
using System.IO;
```



Различают двоичную и XML-сериализацию.

При XML-сериализации информация сериализуется в XML-поток. XML-сериализация может также использоваться для сериализации объектов в потоки XML, которые соответствуют спецификации SOAP (Simple Object Access Protocol – простой протокол доступа к объектам). SOAP - это протокол, основанный на XML и созданный специально для передачи вызовов процедур с использованием XML.

Подробно мы будем говорить о XML-сериализации, позже. А пока познакомимся с двоичной сериализацией в файлы.

Для того, чтобы класс стал сериализуемым, достаточно объявить его с атрибутом `Serializable`. Например, вот так

```
namespace Serial
{
    [Serializable()]
    class Worker
    {
        public int age;
        public int yoe;
    }
}
```

После этого экземпляр класса можно, например, целиком сохранять в файл и читать из файла (именно экземпляр класса целиком, а не поля класса по отдельности). Вот пример такого использования сериализуемого класса

Дополнительно в объекте часть информации может требовать или не требовать сохранения и дальнейшего восстановления, в этом случае применяется метки-атрибуты `[SerializableAttribute]` и `[NonSerializedAttribute]`. Сериализуемая информация содержит не только данные, но и сведения о типе объекта (версию, язык и региональные параметры, а также имя сборки). Основную информацию по выполнению сериализации проводит специальный объект - `Formatter`.

```
using System;
using System.IO;
using System.Runtime.Serialization;
using System.Runtime.Serialization.Formatters.Binary;
namespace Serial
{
    class Program
    {
        static void Main(string[ ] args)
        {
            Worker w = new Worker();    // Задаем экземпляр класса.
            w.age = 22;
            w.yoe = 2;

            // Сериализуем класс.
            FileStream fs = new FileStream("1.txt", FileMode.Create, FileAccess.Write);
            IFormatter bf = new BinaryFormatter();
            bf.Serialize(fs, w);
            fs.Close();

            // Десериализуем класс.
            fs = new FileStream("1.txt", FileMode.Open, FileAccess.Read);
            Worker w1 = (Worker)bf.Deserialize(fs);
            Console.WriteLine("age: " + w1.age + ", yoe: " + w1.yoe);
            fs.Close();
        }
    }
}
```



Приведенный пример выведет на экран

"age: 22, yoe: 2"

Атрибуты

Атрибут - это некоторая дополнительная информация, которая может быть приписана к типам, полям, методам, свойствам и некоторым другим конструкциям языка.

Атрибуты помещаются в исполняемый файл и могут оттуда при необходимости извлекаться.

Все атрибуты являются *классами* (потомками класса **System.Attribute**). Набор атрибутов **.NET** открыт для дополнения, т.е. вы можете определять собственные атрибуты и применять их к вышеуказанным элементам вашего кода.

Атрибуты делятся на предопределенные (встроенные) и пользовательские (которые пишет программист).

Встроенные атрибуты могут использоваться, например, при сериализации (сохранении в поток) данных класса. Скажем, вам надо, чтобы у класса сохранялись не все данные - в этом случае вы можете пометить те данные, которые не надо сохранять, специальным атрибутом.

Компоненты, которые вы располагаете на форме (кнопки, метки и т. п.) имеют некоторый набор свойств (шрифт, местоположение, видимость и т. п.).

В **IDE Visual Studio** вы можете выбрать в окне **Properties** один из двух способов расположения этих свойств - по алфавиту или по категориям. В какую категорию попадет то или иное свойство, определяется специальным встроенным атрибутом.

Атрибуты в C# заключаются в квадратные скобки.

Ниже пример определения и использования пользовательского атрибута:

```
using System;
namespace test
{
    //Объявление атрибута.
    AttributeUsage(AttributeTargets.All, Inherited = false, AllowMultiple = true)]

    class TestAttribute: System.Attribute
    {
        private string name;           // Поле атрибута
        public TestAttribute(string name) // Конструктор атрибута
        {
            this.name = name;
        }
        public virtual string Name      // Свойство только для чтения
        {
            get { return name; }
        }
    }
    //Конец объявления атрибута
}
```

```
//Применение атрибута к классу

[TestAttribute("NAME")]
class Test
{
    static void Main()
    {
        GetAttribute(typeof(Test));
    }

    public static void GetAttribute(Type t)
    {
        TestAttribute att=
            (TestAttribute) Attribute.GetCustomAttribute(t, typeof(TestAttribute));

        Console.WriteLine("{0}", att.Name);
    }
}
}
```

Как видно, атрибут `TestAttribute` является потомком класса `System.Attribute`. Перед определением нового атрибута мы видим строку

Как видно, атрибут **TestAttribute** является потомком класса **System.Attribute**.

Перед определением нового атрибута мы видим строку

```
[AttributeUsage(AttributeTargets.All, Inherited = false, AllowMultiple = true)]
```

Этой строкой определяется область применения нашего атрибута - первый параметр (**AttributeTargets.All**) говорит о том, что мы сможем применять наш атрибут **TestAttribute** к чему угодно, второй (**Inherited=false**) - что атрибут не будет наследоваться (т.е. если мы применим атрибут **TestAttribute** к некоторому классу, то у потомков этого класса атрибута **TestAttribute** не будет), третий (**AllowMultiple = true**) - что атрибут к каждому элементу может применяться только один раз (заметим в скобках, что для определения области применения нашего пользовательского атрибута мы используем другой атрибут - **AttributeUsage**).

```
class TestAttribute: System.Attribute
{
    private string name;           // Поле атрибута
    public TestAttribute(string name) // Конструктор атрибута
    {
        this.name = name;
    }
    public virtual string Name     // Свойство только для чтения
    {
        get { return name; }
    }
}
```

Далее мы определяем для нашего атрибута внутреннюю переменную **name** типа **string** и конструктор с параметром типа **string**. В конструкторе мы записываем значение в переменную **name**.

Еще чуть ниже мы пишем в классе атрибута свойство только для чтения

После создания класса атрибута мы применяем его к другому классу **Test**. Для этого мы должны создать экземпляр атрибута **TestAttribute** непосредственно перед классом **Test**:

```
[TestAttribute("NAME")]  
class Test  
{
```

Делаем мы это в квадратных скобках. При этом у нас вызывается конструктор с параметром атрибута.

```
[TestAttribute("NAME")]
class Test
{
    static void Main()
    {
        GetAttribute(typeof(Test));
    }
    public static void GetAttribute(Type t)
    {
        TestAttribute att=
            (TestAttribute) Attribute.GetCustomAttribute(t, typeof(TestAttribute));
        Console.WriteLine("{0}", att.Name);
    }
}
```

В классе **Test** мы в методе **GetAttribute** просто выводим на консоль значение свойства **name** атрибута. Для этого мы воспользовались статическим методом **GetCustomAttribute** класса **Attribute**. Этот метод принимает в качестве параметров тип, к которому атрибут применяется (т.е. класс **Test** в нашем случае) и собственно атрибут, который мы применяем (у нас это **TestAttribute**). Возвращает же он экземпляр атрибута, который мы и используем для получения значения свойства **name**.

Метод **GetAttribute** мы вызываем в конструкторе класса **Test**.
Результатом выполнения нашей программы будет вывод на консоль слова **"NAME"**

Рефлексия

Иногда нам требуется динамически создать некий код. Естественно, что этот код у нас будет храниться в некоторой сборке. При этом сама созданная динамически сборка может существовать только в памяти или же может быть сохраненной на диск в виде файла.

Сейчас мы посмотрим, как это можно сделать.

Для начала небольшое замечание по порядку создания соответствующих объектов. Сначала мы должны сгенерировать сборку, затем на основании этой сборки - модуль, потом на основании этого модуля - тип (например, класс), потом на основании этого типа (класса) - его члены (конструкторы, методы и т.п.). И, уже в самом конце, мы создаем непосредственно сгенерированный на предыдущих шагах тип.

Вот пример такого кода:



// Создание имени сборки.

```
AssemblyName an = new AssemblyName("MyAssembly");  
an.Version = new Version("1.0.0.0");
```

// Создание сборки.

```
AssemblyBuilder ab;  
ab = AppDomain.CurrentDomain.DefineDynamicAssembly(an, AssemblyBuilderAccess.Save);
```

// Создание модуля в сборке.

```
ModuleBuilder mb = ab.DefineDynamicModule("MyModule", "My.dll");
```

// Создание типа в сборке.

```
TypeBuilder tb = mb.DefineType("MyClass", TypeAttributes.Public);
```



```
// Создание конструктора без параметров.
```

```
ConstructorBuilder cb0 = tb.DefineConstructor(MethodAttributes.Public, CallingConventions.Standard, null);
```

```
// Добавление кода для конструктора.
```

```
ILGenerator il0 = cb0.GetILGenerator();
```

```
il0.Emit(OpCodes.Ret);
```

```
// Создание конструктора с параметром типа string.
```

```
ConstructorBuilder cb = tb.DefineConstructor(MethodAttributes.Public,  
CallingConventions.Standard, new Type[] { typeof(string)});
```

```
// Добавление кода для конструктора.
```

```
ILGenerator il = cb.GetILGenerator();
```

```
il.EmitWriteLine("Constructor");
```

```
il.Emit(OpCodes.Ret);
```

```
// Непосредственное создание типа.
```

```
tb.CreateType();
```

```
// Сохранение типа в файл.
```

```
ab.Save("qqq.dll");
```

Приведенный код при запуске создаст на жестком диске файл `qqq.dll`, в котором будет класс с 2-я конструкторами, причем второй конструктор будет при вызове выводить строку "Constructor".

Несколько пояснений по коду.

Первое. Очень часто при создании типов и членов этих типов надо указать их атрибуты (модификаторы доступа типа `public` и т. п.). Это мы делаем через перечисления `TypeAttributes` и `MethodAttributes`, которое содержит соответствующие значения (`Public`, например). Несколько необходимых значений из этих перечислений можно соединить через побитовое "или".

Второе. Метод `Emit` класса `ILGenerator` в качестве параметра принимает перечисление `OpCodes`, которое фактически содержит инструкции на языке IL - языке, который является аналогом для .NET обычного ассемблера. Это означает, что его инструкции не столь очевидны для реального программирования - именно поэтому в качестве примера таких IL-инструкций и была приведена самая простая из них - а именно выход из функции (`OpCodes.Ret`).