C#编程 第四季 高级篇

网站：www.sikiedu.com　　老师：SiKi

这里的高级篇

不是高级

更多的内容（不是基础的一些语法）

一些系统类的使用

内容列表：

字符串（4）和正则表达式（1）

委托、Lambda表达式和事件（4）

LINQ（1）

反射和特性（2）

线程、任务和同步（3）

文件操作（3）

网络（3）

Xml操作（3.5）Json操作(3.5)

Excel操作（3）

学习方法：

先掌握语法使用

不深究

之后在使用中熟练掌握

开发工具：

VS2022 community

下载地址：

https://visualstudio.microsoft.com/zh-hans/vs/

补充知识

命名空间

目的：

命名空间的目的是为了给类进行分类，就像我们电脑中的文件夹，所有文件都放在一个文件夹里面肯定是不方便管理的。

定义：

```
namespace namespace_name
{
   // 代码声明
}
```

使用：

第一种方式

```
    namespace_name.item_name;
```

第二种方式

```
    using namespace_name;
```

嵌套命名空间

```
    namespace namespace_name1
        {
                // 代码声明
                namespace namespace_name2
                {
                        // 代码声明
                }
        }
```

命名空间部分代码引用自：https://www.runoob.com/csharp/csharp-namespace.html

C#常用命名空间

https://www.cnblogs.com/makesense/p/4500955.html

字符串和正则表达式

字符串用到比较多

正则表达式用到比较少 - 了解

字符串

字符串类

System.String(string是这个类的别名)

System.Text.StringBuilder

System.String类

1、创建字符串　string s = "www.sikiedu.com";

2、获取字符串长度　　s.Length(属性)

3、比较字符串是否一样 s=="www.sikiedu.com"

4、字符串连接　s="http://"+s;

5、使用类似索引器的语法来取得字符串中的某个字符 stringName[index]　s[0] s[3]

关于string字符串：string创建的字符串实际上是一个不可变的数据类型，一旦对字符串对象进行了初始化，该字符串就不能改变内容了，上面的示例中实际上是创建了一个新的字符串，把旧字符串的内容复制到新字符串中。然后把新字符串的引用赋值为字符串的对象。(重复修改给定的字符串，效率会很低)

关于字符串的更多方法

1、CompareTo()方法，比较字符串的内容

2、Replace()用另一个字符或者字符串替换字符串中给定的字符或者字符串

3、Split()在出现给定字符的地方，把字符串拆分称一个字符串数组

4、SubString()在字符串中检索给定位置的子字符串

5、ToLower()把字符串转换成小写形式

6、ToUpper()把字符串转换成大写形式

7、Trim()删除首尾的空白

8、Concat()方法，合并字符串 - 静态方法

9、CopyTo()方法，把字符串中指定的字符复制到一个数组中

10、Format()方法，格式化字符串 - 静态方法

https://www.cnblogs.com/net-sky/p/10250880.html

https://www.runoob.com/csharp/csharp-string.html

https://docs.microsoft.com/zh-cn/dotnet/api/system.string.format?view=net-5.0

11、IndexOf()方法，取得字符串第一次出现某个给定字符串或者字符的位置

12、IndexOfAny()方法，

13、Insert()把一个字符串实例插入到另一个字符串实例的制定索引处

14、Join()合并字符串数组，创建一个新字符串

StringBuilder类(位于System.Text命名空间下)

1、创建StringBuilder对象

StringBuilder sb = new StringBuilder("www.sikiedu.com");

StringBuilder sb = new StringBuilder(20);

StringBuilder sb = new StringBuilder("www.sikiedu.com",100);

关于StringBuilder对象创建的时候的内存占用

2、Append()方法，给当前字符串追加一个字符

3、Insert()追加特定格式的字符串

4、Remove()从当前字符串中删除字符

5、Replace()在当前字符串中，用某个字符或者字符串全部替换另一个字符或者字符串

6、ToString()把当前stringBuilder中存储的字符串，提取成一个不可变的字符串

正则表达式

什么是正则表达式?

英文Regular Expression,是计算机科学的一个重要概念，她使用一种数学算法来解决计算机程序中的文本检索，匹配等问题，正则表达式语言是一种专门用于字符串处理的语言。在很多语言中都提供了对它的支持，c#也不例外，它可以帮我们解决下面的问题：

1，检索：通过正则表达式，从字符串中获取我们想要的部分

2，匹配：判断给定的字符串是否符合正则表达式的过滤逻辑

你可以认为正则表达式表述了一个字符串的书写规则

判断用户输入的密码是否合法，判断用户输入的邮箱格式是否合法

正则表达式的组成

正则表达式就是由普通字符以及特殊字符（成为元字符）组成的文字模式。该模式描述在查找文字主体时待匹配的一个或多个字符串。

元字符 - 见word文档

常用的操作正则表达式的方法和委托

下面学习一下位于System.Text.RegularExpressions下的Regex类的一些静态方法和委托

1、静态方法IsMatch （返回值是一个布尔类型,用于判断指定的字符串是否与正则表达式字符串匹配，它有三个重载方法）

bool IsMatch(string input, string pattern);

参数:　input:　　要搜索匹配项的字符串。

pattern: 要匹配的正则表达式模式。

返回结果: 如果正则表达式找到匹配项，则为 true；否则，为 false。

bool IsMatch(string input, string pattern, RegexOptions options);

参数: input: 要搜索匹配项的字符串。

pattern: 要匹配的正则表达式模式。

options: 枚举值的一个按位组合，这些枚举值提供匹配选项。

返回结果: 如果正则表达式找到匹配项，则为 true；否则，为 false。

bool IsMatch(string input, string pattern, RegexOptions options, TimeSpan matchTimeout);

参数: input: 要搜索匹配项的字符串。

pattern: 要匹配的正则表达式模式。

options: 枚举值的一个按位组合，这些枚举值提供匹配选项。

matchTimeout: 超时间隔，或 System.Text.RegularExpressions.Regex.InfiniteMatchTimeout 指示该方法不应超时。

返回结果: 如果正则表达式找到匹配项，则为 true；否则，为 false。

关于参数RegexOptions

它是一个枚举类型，有以下枚举值

RegexOptions枚举值	内联标志	简单说明
ExplicitCapture	n	只有定义了命名或编号的组才捕获
IgnoreCase	i	不区分大小写
IgnorePatternWhitespace	x	消除模式中的非转义空白并启用由 # 标记的注释。
MultiLine	m	多行模式，其原理是修改了^和$的含义
SingleLine	s	单行模式，和MultiLine相对应

内敛标志可以更小力度（一组为单位）的定义匹配选项

静态方法Match（System.Text.RegularExpressions）

静态方法Match，使用指定的匹配选项在输入字符串中搜索指定的正则表达式的第一个匹配项。 返回一个包含有关匹配的信息的对象。同样有三个重载方法，参数和IsMatch方法相同。此外，在Regex类中，还有一个同名的非静态方法，适用于多个实例的情况下，效率更高一些。

Match Match(string input, string pattern);

Match Match(string input, string pattern, RegexOptions options);

Match Match(string input, string pattern, RegexOptions options, TimeSpan matchTimeout);

静态方法Matches (System.Text.RegularExpressions)

静态方法Matches,在指定的输入字符串中搜索指定的正则表达式的所有匹配项。跟上面方法不同之处，就是这个方法返回的是所有匹配项，他同样有三个重载方法，并且参数和Match方法完全相同

MatchCollection Matches(string input, string pattern);

MatchCollection Matches(string input, string pattern, RegexOptions options);

MatchCollection Matches(string input, string pattern, RegexOptions options, TimeSpan matchTimeout);

Replaces函数 (System.Text.RegularExpressions)

我们知道正则表达式主要是实现验证,提取,分割,替换字符的功能.Replace函数是实现替换功能的.

1)Replace(string input,string pattern,string replacement)

//input是源字符串,pattern是匹配的条件,replacement是替换的内容,就是把符合匹配条件pattern的内容转换成它

比如string result = Regex.Replace("abc", "ab", "##");

//结果是##c,就是把字符串abc中的ab替换成##

2)Replace(string input,string pattern,string replacement,RegexOptions options)

//RegexOptions是一个枚举类型,用来做一些设定.

//前面用注释时就用到了RegexOptions.IgnorePatternWhitespace.如果在匹配时忽略大小写就可以用RegexOptions.IgnoreCase

比如string result = Regex.Replace("ABc", "ab", "##",RegexOptions.IgnoreCase);

如果是简单的替换用上面两个函数就可以实现了.但如果有些复杂的替换,比如匹配到很多内容,不同的内容要替换成不同的字符.就需要用到下面两个函数

3)Replace(string input,string pattern,MatchEvaluator evaluator);

//evaluator是一个代理,其实简单的说是一个函数指针,把一个函数做为参数参进来

//由于C#里没有指针就用代理来实现类似的功能.你可以用代理绑定的函数来指定你要实现的复杂替换.

4)Replace(string input,string pattern,MatchEvaluator evaluator,RegexOptions options);
//这个函数上上面的功能一样,只不过多了一点枚举类型来指定是否忽略大小写等设置

静态方法Split拆分文本
使用正则表达式匹配的位置，将文本拆分为一个字符串数组,同样有三个重载方法，返回值为字符串数组
string[] Split(string input, string pattern);

string[] Split(string input, string pattern, RegexOptions options);

string[] Split(string input, string pattern, RegexOptions options, TimeSpan matchTimeout);

@符号
我们经常在正则表达式字符串前面加上@字符，这样不让编译器去解析其中的转义字符，而作为正则表达式的语法（元字符）存在。

string s =@"www.baidu.com \n lkjsdflkj";

定位元字符
我们经常在正则表达式字符串前面加上@字符，这样不让编译器去解析其中的转义字符，而作为正则表达式的语法（元字符）存在。

字符	说明
\b	匹配单词的开始或结束
\B	匹配非单词的开始或结束
^	匹配必须出现在字符串的开头或行的开头
$	匹配必须出现在以下位置：字符串结尾、字符串结尾处的 \n 之前或行的结尾。
\A	指定匹配必须出现在字符串的开头（忽略 Multiline 选项）。
\z	指定匹配必须出现在字符串的结尾（忽略 Multiline 选项）。
\z	指定匹配必须出现在字符串的结尾或字符串结尾处的 \n 之前（忽略 Multiline 选项）。
\G	指定匹配必须出现在上一个匹配结束的地方。与 Match.NextMatch() 一起使用时，此断言确保所有匹配都是连续的。

定位元字符示例

示例一：　　区配开始 ^

```
string str = "I am Blue cat";
Console.WriteLine(Regex.Replace(str, "^","准备开始:"));
```

示例二： 区始结束 $

```
string str = "I am Blue cat";
Console.WriteLine(Regex.Replace(str, "$", "  结束了！"));
```

基本语法元字符

字符	说明
.	匹配除换行符以外的任意字符
\w	匹配字母、数字、下划线、汉字　（指大小写字母、0-9的数字、下划线_）
\W	\w的补集　（ 除“大小写字母、0-9的数字、下划线_”之外）
\s	匹配任意空白符　(包括换行符/n、回车符/r、制表符/t、垂直制表符/v、换页符/f)
\S	\s的补集　(除\s定义的字符之外)
\d	匹配数字　(0-9数字)
\D	表示\d的补集　(除0-9数字之外)

在正则表达式中，\是转义字符. * 是元字符 如果要表示一个\ . *字符的话，需要使用\\ \. *

示例

示例一：校验只允许输入数字

```
string strCheckNum1 = "23423423a3", strCheckNum2 = "324234";
Console.WriteLine("匹配字符串"+strCheckNum1+"是否为数
字:"+Regex.IsMatch(strCheckNum1, @"^\d*$"));
Console.WriteLine("匹配字符串" + strCheckNum2 + "是否为数字:" +
Regex.IsMatch(strCheckNum2, @"^\d*$"));
```

示例二：校验只允许输入除大小写字母、0-9的数字、下划线_以外的任何字

```
string strCheckStr1 = "abcds_a", strCheckStr2 = "**&&((((2", strCheckStr3 =
"**&&((((";
string regexStr = @"^\W*$";
```

```
Console.WriteLine("匹配字符串" + strCheckStr1 + "是否为除大小写字母、0-9的数字、下划线_以外的任何字符:" + Regex.IsMatch(strCheckStr1, regexStr));
Console.WriteLine("匹配字符串" + strCheckStr2 + "是否为除大小写字母、0-9的数字、下划线_以外的任何字符:" + Regex.IsMatch(strCheckStr2, regexStr));
Console.WriteLine("匹配字符串" + strCheckStr3 + "是否为除大小写字母、0-9的数字、下划线_以外的任何字符:" + Regex.IsMatch(strCheckStr3, regexStr));
```

反义字符

字符	说明
\W	\w的补集 （ 除“大小写字母、0-9的数字、下划线_”之外）
\S	\s的补集 (除\s定义的字符之外)
\D	表示\d的补集 (除0-9数字之外)
\B	匹配不是单词开头或结束的位置
[ab]	匹配中括号中的字符
[a-c]	a字符到c字符之间是字符
[^x]	匹配除了x以外的任意字符
[^adwz]	匹配除了adwz这几个字符以外的任意字符

```
//示例：查找除ahou这之外的所有字符
string strFind1 = "I am a Cat!", strFind2 = "My Name's Blue cat!";
Console.WriteLine("除ahou这之外的所有字符，原字符为：" + strFind1 + "替换后:" + Regex.Replace(strFind1, @"[^ahou]","*"));
Console.WriteLine("除ahou这之外的所有字符，原字符为：" + strFind2 + "替换后:" + Regex.Replace(strFind2, @"[^ahou]", "*"));
```

重复描述字符

字符	说明
{n}	匹配前面的字符n次
{n,}	匹配前面的字符n次或多于n次
{n,m}	匹配前面的字符n到m次
?	重复零次或一次
+	重复一次或更多次
*	重复零次或更多次

示例：校验输入内容是否为合法QQ号(备注：QQ号为5-12位数字)

```
string isQq1 = "1233", isQq2 = "a1233", isQq3 = "0123456789123", isQq4 =
"556878544";
string regexQq = @"^\d{5,12}$";
Console.WriteLine(isQq1+"是否为合法QQ号(5-12位数字):" + Regex.IsMatch(isQq1,
regexQq));
Console.WriteLine(isQq2 + "是否为合法QQ号(5-12位数字):" + Regex.IsMatch(isQq2,
regexQq));
Console.WriteLine(isQq3 + "是否为合法QQ号(5-12位数字):" + Regex.IsMatch(isQq3,
regexQq));
Console.WriteLine(isQq4 + "是否为合法QQ号(5-12位数字):" + Regex.IsMatch(isQq4,
regexQq));
```

择一匹配

字符　　说明

|　　将两个匹配条件进行逻辑“或”（Or）运算。

示例一：查找数字或字母

```
string findStr1 = "ad(d3)-df";
string regexFindStr = @"[a-z]|\d";
string newStrFind=String.Empty;
MatchCollection newStr =  Regex.Matches(findStr1, regexFindStr);
 newStr.Cast<Match>().Select(m => m.Value).ToList<string>().ForEach(i =>
newStrFind += i);
Console.WriteLine(findStr1 + "中的字母和数字组成的新字符串为:" + newStrFind);
```

示例二：将人名输出（"zhangsan;lisi,wangwu.zhaoliu"）

```
string strSplit = "zhangsan;lisi,wangwu.zhaoliu";
string regexSplitstr = @"[;]|[,]|[.]";
Regex.Split(strSplit, regexSplitstr).ToList().ForEach(i =>
Console.WriteLine(i));
```

下面可以替换为 校验国内手机号 - https://www.jianshu.com/p/37cb110604fb

示例三：校验国内电话号码(支持三种写法校验 A. 010-87654321 B. (010)87654321 C.01087654321 D.010 87654321)

```
string TelNumber1 = "(010)87654321", TelNumber2 = "010-87654321", TelNumber3 =
"01087654321",
```

```
        TelNumber4 = "09127654321", TelNumber5 = "010)87654321",TelNumber6="(010-87654321",
        TelNumber7="91287654321";

Regex RegexTelNumber3 = new Regex(@"\(0\d{2,3}\)[- ]?\d{7,8}|^0\d{2,3}[- ]?\d{7,8}$");
Console.WriteLine("电话号码 " + TelNumber1 + " 是否合法:" + RegexTelNumber3.IsMatch(TelNumber1));
Console.WriteLine("电话号码 " + TelNumber2 + " 是否合法:" + RegexTelNumber3.IsMatch(TelNumber2));
Console.WriteLine("电话号码 " + TelNumber3 + " 是否合法:" + RegexTelNumber3.IsMatch(TelNumber3));
Console.WriteLine("电话号码 " + TelNumber4 + " 是否合法:" + RegexTelNumber3.IsMatch(TelNumber4));
Console.WriteLine("电话号码 " + TelNumber5 + " 是否合法:" + RegexTelNumber3.IsMatch(TelNumber5));
Console.WriteLine("电话号码 " + TelNumber6 + " 是否合法:" + RegexTelNumber3.IsMatch(TelNumber6));
Console.WriteLine("电话号码 " + TelNumber7 + " 是否合法:" + RegexTelNumber3.IsMatch(TelNumber7));
```

对正则表达式分组()

用小括号来指定子表达式(也叫做分组)

示例一：重复单字符 和 重复分组字符

```
Console.WriteLine("请输入一个任意字符串，测试分组：");
string inputStr = Console.ReadLine();
string strGroup1 = @"a{2}";
Console.WriteLine("单字符重复2两次替换为22，结果为："+Regex.Replace(inputStr, strGroup1,"22"));
//重复 多个字符 使用（abcd）{n}进行分组限定
string strGroup2 = @"(ab\w{2}){2}";
Console.WriteLine("分组字符重复2两次替换为5555，结果为：" + Regex.Replace(inputStr, strGroup2, "5555"));
```

示例二：校验IP4地址（如：192.168.1.4，为四段，每段最多三位，每段最大数字为255，并且第一位不能为0）

```
string regexStrIp4 = @"^(((2[0-4]\d|25[0-5]|[01]?\d\d?)\.){3}(2[0-4]\d|25[0-5]|[01]?\d\d?))$";
Console.WriteLine("请输入一个IP4地址：");
string inputStrIp4 = Console.ReadLine();
Console.WriteLine(inputStrIp4 + " 是否为合法的IP4地址：" +
Regex.IsMatch(inputStrIp4, regexStrIp4));
Console.WriteLine("请输入一个IP4地址：");
string inputStrIp4Second = Console.ReadLine();
Console.WriteLine(inputStrIp4 + " 是否为合法的IP4地址：" +
Regex.IsMatch(inputStrIp4Second, regexStrIp4));
```

委托、Lambda表达式和事件

委托

什么是委托?

如果我们要把方法当做参数来传递的话，就要用到委托。简单来说委托是一个类型，这个类型可以赋值一个方法的引用。

声明委托

在C#中使用一个类分两个阶段，首选定义这个类，告诉编译器这个类由什么字段和方法组成的，然后使用这个类实例化对象。在我们使用委托的时候，也需要经过这两个阶段，首先定义委托，告诉编译器我们这个委托可以指向哪些类型的方法，然后，创建该委托的实例。

定义委托的语法如下：

```
delegate void IntMethodInvoker(int x);
```

定义了一个委托叫做IntMethodInvoker,这个委托可以指向什么类型的方法呢?

这个方法要带有一个int类型的参数，并且方法的返回值是void的。

定义一个委托要定义方法的参数和返回值，使用关键字delegate定义。

定义委托的其他案例：

```
delegate double TwoLongOp(long first,long second);
delegate string GetAString();
```

使用委托

```
private delegate string GetAString();
```

```
static void Main(){
        int x = 40;
        GetAString firstStringMethod = x.ToString;
        Console.WriteLine(firstStringMethod());
}
```

在这里我们首先使用GetAString委托声明了一个类型叫做fristStringMethod,接下来~~使用new 对它进行初始化,~~使它引用到x中的ToString方法上，这样firstStringMethod就相当于x.ToString，我们通过firstStringMethod()执行方法就相当于x.ToString()

通过委托示例调用方法有两种方式

```
        fristStringMethod();
        firstStringMethod.Invoke();
```

委托的赋值

GetAString firstStringMethod = new GetAString(x.ToString);只需要把方法名给一个委托的构造方法就可以了

GetAString firstStringMethod = x.ToString;也可以把方法名直接给委托的实例

简单委托示例

定义一个类MathsOperations里面有两个静态方法，使用委托调用该方法

```
class MathOperations{
        public static double MultiplyByTwo(double value){
                return value*2;
        }
        public static double Square(double value){
                return value*value;
        }
}
delegate double DoubleOp(double x);
static void Main(){
        DoubleOp[] operations={
MathOperations.MultiplyByTwo,MathOperations.Square };
        for(int i =0;i<operations.Length;i++){
                Console.WriteLine("Using operations "+i);
                ProcessAndDisplayNumber( operations[i],2.0 );
        }
```

```
}
static void ProcessAndDisplayNumber(DoubleOp action,double value){
        double res = action(value);
        Console.Writeline("Value :"+value+" Result:"+res);
}
```

Action委托和Func委托

除了我们自己定义的委托之外，系统还给我们提供过来一个内置的委托类型，Action和Func

Action委托引用了一个void返回类型的方法，T表示方法参数，先看Action委托有哪些

Action

Action<in T>

Action<in T1,in T2>

Action<in T1,in T2 .... inT16>

Func引用了一个带有一个返回值的方法，它可以传递0或者多到16个参数类型，和一个返回类型

Func<out TResult>

Func<in T,out TResult>

Func<int T1,inT2,,,,,,in T16,out TResult>

案例1

```
delegate double DoubleOp(double x);
```

如何用Func表示

```
Func<double,double>
```

案例2-对int类型排序

对集合进行排序，冒泡排序

```
bool swapped = true;
do{
        swapped = false;
        for(int i =0;i<sortArray.Length -1;i++){
                if(sortArray[i]>sortArray[i+1]){
                        int temp= sortArray[i];
                        sortArray[i]=sortArray[i+1];
                        sortArray[i+1]=temp;
```

```
                        swapped = true;
                }
        }
}while(swapped);
```

这里的冒泡排序只适用于int类型的，如果我们想对他进行扩展，这样它就可以给任何对象排序。

案例2-雇员类

```
class Employee{
        public Employ(string name,decimal salary){
                this.Name = name;
                this.Salary = salary;
        }
        public string Name{get;private set;}
        public decimal Salary{get;private set;}
        public static bool CompareSalary(Employee e1,Employee e2){
                return e1.salary>e2.salary;
        }
}
```

案例2-通用的排序方法

```
public static void Sort<T>( List<T> sortArray,Func<T,T,bool> comparision ){
        bool swapped = true;
        do{
                swapped = false;
                for(int i=0;i<sortArray.Count-1;i++){
                        if(comparision(sortArray[i+1],sortArray[i])){
                                T temp = sortArray[i];
                                sortArray[i]=sortArray[i+1];
                                sortArray[i+1]=temp;
                                swapped = true;
                        }
                }
        }while(swapped);
}
```

案例2-对雇员类排序

```
static void Main(){
        Employee[] employees = {
                new Employee("Bunny",20000),
                new Employee("Bunny",10000),
                new Employee("Bunny",25000),
                new Employee("Bunny",100000),
                new Employee("Bunny",23000),
                new Employee("Bunny",50000),
                };
        Sort(employees,Employee.CompareSalary);
        输出
}
```

多播委托

前面使用的委托都只包含一个方法的调用，但是委托也可以包含多个方法，这种委托叫做多播委托。使用多播委托就可以按照顺序调用多个方法，多播委托只能得到调用的最后一个方法的结果，一般我们把多播委托的返回类型声明为void。

```
        Action action1 = Test1;
        action2+=Test2;
        action2-=Test1;
```

多播委托包含一个逐个调用的委托集合，如果通过委托调用的其中一个方法抛出异常，整个迭代就会停止。

取得多播委托中所有方法的委托

```
Action a1 = Method1;
a1+=Method2;

Delegate[] delegates=a1.GetInvocationList();
foreach(delegate d in delegates){
        //d();
        d.DynamicInvoke(null);
}
```

遍历多播委托中所有的委托，然后单独调用

匿名方法

到目前为止，使用委托，都是先定义一个方法，然后把方法给委托的实例。但还有另外一种使用委托的方式，不用去定义一个方法，应该说是使用匿名方法（方法没有名字）。

```
Func<int,int,int> plus = delegate (int a,int b){
        int temp = a+b;
        return temp;
};
int res = plus(34,34);
Console.WriteLine(res);
```

在这里相当于直接把要引用的方法直接写在了后面，优点是减少了要编写的代码，减少代码的复杂性

Lambda表达式-表示一个方法的定义

从C#3.0开始，可以使用Lambda表达式代替匿名方法。只要有委托参数类型的地方就可以使用Lambda表达式。刚刚的例子可以修改为

```
Func<int,int,int> plus = (a,b)=>{ int temp= a+b;return temp; };
int res = plus(34,34);
Console.WriteLine(res);
```

Lambda运算符“=>”的左边列出了需要的参数，如果是一个参数可以直接写 a=>(参数名自己定义),如果多个参数就使用括号括起来，参数之间以，间隔。

Lambda表达式就是匿名方法的简写形式

多行语句

1，如果Lambda表达式只有一条语句，在方法快内就不需要花括号和return语句，编译器会自动添加return语句

```
        Func<double,double> square = x=>x*x;
        添加花括号，return语句和分号是完全合法的
        Func<double,double> square = x=>{
                return x*x;
                }
```

2，如果Lambda表达式的实现代码中需要多条语句，就必须添加花括号和return语句。

Lambda表达式外部的变量

通过Lambda表达式可以访问Lambda表达式块外部的变量。这是一个非常好的功能，但如果不能正确使用，也会非常危险。示例：

```
int somVal = 5;
Func<int,int> f = x=>x+somVal;
Console.WriteLine(f(3));//8
somVal = 7;
Console.WriteLine(f(3));//10
```

这个方法的结果，不但受到参数的控制，还受到somVal变量的控制，结果不可控，容易出现编程问题，用的时候要谨慎。

事件

事件(event)基于委托，为委托提供了一个发布/订阅机制，我们可以说事件是一种具有特殊签名的委托。

什么是事件？

事件（Event）是类或对象向其他类或对象通知发生的事情的一种特殊签名的委托.

事件的声明

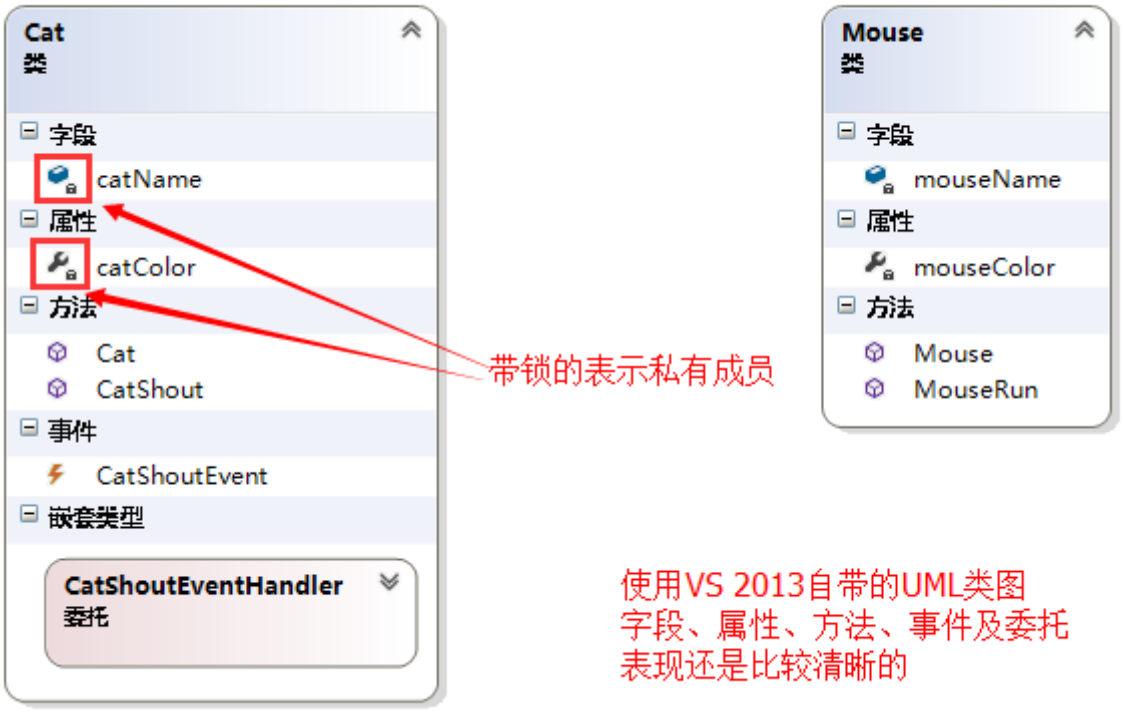
public event 委托类型 事件名;

事件使用event关键词来声明，他的返回类值是一个委托类型。

通常事件的命名，以名字+Event 作为他的名称，在编码中尽量使用规范命名，增加代码可读性。

为了更加容易理解事件，我们还是以前面的动物的示例来说明，有三只动物，猫(名叫Tom)，还有两只老鼠（Jerry和Jack），当猫叫的时候，触发事件(CatShout)，然后两只老鼠开始逃跑(MouseRun)。接下来用代码来实现。(设计模式-观察者模式)

猫捉老鼠的UML图



Cat类和Mouse类

```
class Cat
{
    string catName;
    string catColor { get; set; }
    public Cat(string name, string color)
    {
        this.catName = name;
        catColor = color;
    }
    public void CatShout()
    {
        Console.WriteLine(catColor+" 的猫 "+catName+" 过来了，喵！喵！喵！
\n");
        //猫叫时触发事件
        //猫叫时，如果CatShoutEvent中有登记事件，则执行该事件
        if (CatShoutEvent != null)
            CatShoutEvent();
    }
```

```
        public delegate void CatShoutEventHandler();
        public event CatShoutEventHandler CatShoutEvent;
    }
    class Mouse
    {
        string mouseName;
        string mouseColor { get; set; }
        public Mouse(string name, string color)
        {
            this.mouseName = name;
            this.mouseColor = color;
        }
        public void MouseRun()
        {
            Console.WriteLine(mouseColor + " 的老鼠 " + mouseName + " 说: \"老猫来了, 快跑! \"  \n我跑! ! \n我使劲跑! ! \n我加速使劲跑! ! ! \n");
        }
    }
```

运行代码

```
            Console.WriteLine("[场景说明]: 一个月明星稀的午夜,有两只老鼠在偷油吃\n");
            Mouse Jerry = new Mouse("Jerry", "白色");
            Mouse Jack = new Mouse("Jack", "黄色");

            Console.WriteLine("[场景说明]: 一只黑猫蹑手蹑脚的走了过来\n");
            Cat Tom = new Cat("Tom", "黑色");
            Console.WriteLine("[场景说明]: 为了安全的偷油, 登记了一个猫叫的事件\n");

            Tom.CatShoutEvent += new Cat.CatShoutEventHandler(Jerry.MouseRun);
            Tom.CatShoutEvent += new Cat.CatShoutEventHandler(Jack.MouseRun);

            Console.WriteLine("[场景说明]: 猫叫了三声\n");
```

```
Tom.CatShout();
Console.ReadKey();
```

事件与委托的联系和区别

-事件是一种特殊的委托，或者说是受限制的委托，是委托一种特殊应用，只能施加+=,-=操作符。二者本质上是一个东西。

-event ActionHandler Tick; // 编译成创建一个私有的委托示例，和施加在其上的add, remove方法.

-event只允许用add, remove方法来操作，这导致了它不允许在类的外部被直接触发，只能在类的内部适合的时机触发。委托可以在外部被触发，但是别这么用。

-使用中，委托常用来表达回调，事件表达外发的接口。

-委托和事件支持静态方法和成员方法，delegate(void * pthis, f_ptr)，支持静态返方法时，pthis传null.支持成员方法时，pthis传被通知的对象.

-委托对象里的三个重要字段是，pthis, f_ptr, pnext, 也就是被通知对象引用，函数指针/地址，委托链表的下一个委托节点.

最新课程通知和免费课程下载

用法

在数据（集合）中做查询

基本用不到

预热后期学习数据库（服务器端开发）

了解

可以跳过

反射和特性 - 用的不多

什么是元数据，什么是反射

程序是用来处理数据的，文本和特性都是数据，而我们程序本身（类的定义和BCL中的类）这些也是数据。（BCL - Basic Class Lib基础类库）

有关程序及其类型的数据被称为元数据(metadata)，它们保存在程序的程序集中。

程序在运行时，可以查看其它程序集或其本身的元数据。一个运行的程序查看本身的元数据或者其他程序集的元数据的行为叫做反射。

下面我们我们来学习如何使用Type类来反射数据，以及如何使用特性来给类型添加元数据。

Type位于System.Reflection命名空间下

Type类

预定义类型(int long 和string等),BCL中的类型(Console,IEnumerable等)和程序员自定义类型（MyClass,MyDel等）。 每种类型都有自己的成员和特性。

BCL声明了一个叫做Type的抽象类，它被设计用来包含类型的特性。使用这个类的对象能让我们获取程序使用的类型的信息。

由于 Type是抽象类，因此不能利用它去实例化对象。关于Type的重要事项如下:

对于程序中用到的每一个类型，CLR都会创建一个包含这个类型信息的Type类型的对象。

程序中用到的每一个类型都会关联到独立的Type类的对象。

不管创建的类型有多少个示例，只有一个Type对象会关联到所有这些实例。

System.Type类部分成员

成员	成员类型	描述
Name	属性	返回类型的名字
Namespace	属性	返回包含类型声明的命名空间

Assembly	属性	返回声明类型的程序集。
GetFields	方法	返回类型的字段列表
GetProperties	方法	返回类型的属性列表
GetMethods	方法	返回类型的方法列表

获取Type对象

获取Type对象有两种方式

1，Type t = myInstance.GetType();//通过类的实例来获取Type对象

在object类有一个GetType的方法，返回Type对象，因为所有类都是从object继承的，所以我们可以在任何类型上使用GetType（）来获取它的Type对象

2，Type t = typeof(ClassName);//直接通过typeof运算符和类名获取Type对象

获取里面的属性

```
FieldInfo[] fi = t.GetFields();
foreach(FieldInfo f in fi){
        Console.WriteLine(f.Name+" ");
}
```

Assembly类

Assembly类在System.Reflection命名空间中定义，它允许访问给定程序集的元数据，它也包含了可以加载和执行程序集。

如何加载程序集？

1,Assembly assembly1 = Assembly.Load("SomeAssembly");根据程序集的名字加载程序集，它会在本地目录和全局程序集缓存目录查找符合名字的程序集。

2,Assembly assembly2 = Assembly.LoadFrom(@"c:\xx\xx\xx\SomeAssembly.dll")//这里的参数是程序集的完整路径名，它不会在其他位置搜索。

Assembly对象的使用

1,获取程序集的全名 string name = assembly1.FullName;

2,遍历程序集中定义的类型

```
Type[] types = theAssembly.GetTypes();
foreach(Type definedType in types){
        //
}
```

3,遍历程序集中定义的所有特性(稍后介绍)

```
        Attribute[] definedAttributes =
Attribute.GetCustomAttributes(someAssembly);
```

什么是特性？

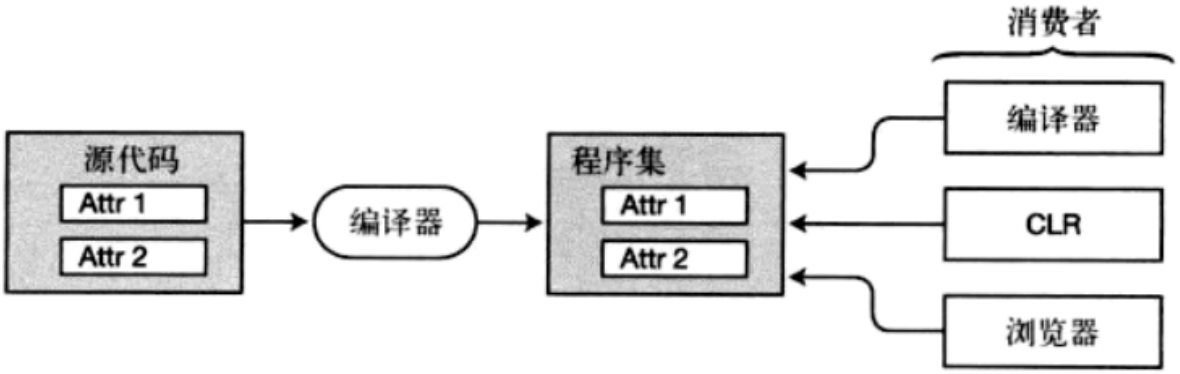
特性(attribute)是一种允许我们向程序的程序集增加元数据的语言结构。它是用于保存程序结构信息的某种特殊类型的类。

将应用了特性的程序结构叫做目标

设计用来获取和使用元数据的程序（对象浏览器）叫做特性的消费者

.NET预定了很多特性，我们也可以声明自定义特性

创建和使用特性



我们在源代码中将特性应用于程序结构；

编译器获取源代码并且从特性产生元数据，然后把元数据放到程序集中；

消费者程序可以获取特性的元数据以及程序中其他组件的元数据。注意，编译器同时生产和消费特性。

关于特性的命名规范，特性名使用Pascal命名法（首字母大写），并且以Attribute后缀结尾，当为目标应用特性时，我们可以不使用后缀。例如对于SerializableAttribute和MyAttributeAttribute这两个特性，我们把他们应用到结构是可以使用Serializable和MyAttribute。

应用特性

先看看如何使用特性。特性的目的是告诉编译器把程序结构的某组元数据嵌入程序集。我们可以通过把特性应用到结构来实现。

在结构前放置特性片段来应用特性；

特性片段被方括号包围，特性片段包括特性名和特性的参数列表；

应用了特性的结构成为特性装饰。

案例1

```
[Serializable]                              //特性
public class MyClass{
        // ...
}
```

案例2

```
[MyAttribute("Simple class","Version 3.57")]    //带有参数的特性
public class MyClass{
        //...
}
```

Obsolete特性->.NET预定义特性

一个程序可能在其生命周期中经历多次发布，而且很可能延续多年。在程序生命周期的后半部分，程序员经常需要编写类似功能的新方法替换老方法。处于多种原因，你可能不再使用哪些调用过时的旧方法的老代码。而只想用新编写的代码调用新方法。旧的方法不能删除，因为有些旧代码也使用的旧方法，那么如何提示程序员使用新代码呢？可以使用Obsolete特性将程序结构标注为过期的，并且在代码编译时，显示有用的警告信息。

```
class Program{
        [Obsolete("Use method SuperPrintOut")]  //将特性应用到方法
        static void PrintOut(string str){
                Console.WriteLine(str);
        }
        [Obsolete("Use method SuperPrintOut",true)]//这个特性的第二个参数表示是是否应该标记为错误，而不仅仅是警告。
        static void PrintOut(string str){
                Console.WriteLine(str);
        }
        static void Main(string[] args){
                PrintOut("Start of Main");
        }
}
```

Conditional特性

Conditional特性允许我们包括或取消特定方法的所有调用。为方法声明应用Conditional特性并把编译符作为参数来使用。

定义方法的CIL代码本身总是会包含在程序集中，只是调用代码会被插入或忽略。

```
#define DoTrace

class Program{
        [Conditional("DoTrace")]
        static void TraceMessage(string str){
                Console.WriteLine(str);
        }
        static void Main(){
                TraceMessage("Start of Main");
                Console.WriteLine("Doing work in Main.")
                TraceMessage("End of Main");
        }
}
```

调用者信息特性

调用者信息特性可以访问文件路径，代码行数，调用成员的名称等源代码信息。

这个三个特性名称为CallerFilePath,CallerLineNumber和CallerMemberName

这些特性只能用于方法中的可选参数

```
public static void PrintOut(string message,[CallerFilePath] string filename="",
[CallerLineNumber]int lineNumber = 0,[CallerMemberName]string callingMember=""){
        Console.WriteLine("Message:"+message);
        Console.WriteLine("Line :"+lineNumber);
        Console.WriteLine("Called from:"+callingMember);
        Console.WriteLine("Message :"+message);
}
```

DebuggerStepThrough特性

我们在单步调试代码的时候，常常希望调试器不要进入某些方法。我们只想执行该方法，然后继续调试下一行。DebuggerStepThrough特性告诉调试器在执行目标代码时不要进入该方法调试。有些方法小并且毫无疑问是正确的，在调试时对其反复单步调试只能徒增烦恼。要小心使用该特性，不要排除了可能出现bug的代码。

该特性位于System.Diagnostics命名空间下

该特性可用于类，结构，构造方法，方法或访问器

```
class Program{
        int _x=1;
```

```
        int X{
                get{return _x;};
                [DebuggerStepThrough]
                set{
                        _x=_x*2;
                        _x+=value;
                }
        }
        public int Y{get;set;}
        [DebuggerStepThrough]
        void IncrementFields(){
                X++;
                Y++;
        }
        static void Main(){
                Program p = new Program();
                p.IncrementFields();
                p.X = 5;
                Console.WriteLine("P.X:"+p.X+" p.Y:"+p.Y);
                Console.ReadKey();
        }
}
```

其他预定义特性

特性	意义
CLSCompliant	声明可公开的成员应该被编译器检查是否符合CLS。兼容的程序集可以被任何.NET兼容的语言使用
Serializable	声明结构可以被序列化
NonSerialized	声明结构不可以被序列化
DLLImport	声明是非托管代码实现的
WebMethod	声明方法应该被作为XML Web服务的一部分暴露
AttributeUsage	声明特性能应用到什么类型的程序结构。将这个特性应用到特性声明上

多个特性

我们可以为单个结构应用多个特性。有下面两种添加方式

独立的特性片段相互叠在一起

```
[Serializable]
[MyAttribute("Simple class","Version 3.57")]
```

单个特性片段，特性之间使用逗号间隔

```
[Serializable,MyAttribute("Simple class","Version 3.57")]
```

特性目标

我们可以将特性应用到字段和属性等程序结构上。我们还可以显示的标注特性，从而将它应用到特殊的目标结构。要使用显示目标，在特性片段的开始处放置目标类型，后面跟冒号。

例如，如下的代码用特性装饰方法，并且还把特性应用到返回值上。

```
[return:MyAttribute("This value ...","Version2.3")]
[method:MyAttribute("Print....","Version 3.6")]
public long ReturnSettings(){
...
```

c#定义了10个标准的特性目标。

event field method param property return type typevar assembly module

其中type覆盖了类，结构，委托，枚举和接口。

typevar指定使用泛型结构的类型参数。

全局特性

我们可以通过使用assembly和module目标名称来使用显式目标说明符把特性设置在程序集或模块级别。

程序集级别的特性必须放置在任何命名空间之外，并且通常放置在AssemblyInfo.cs文件中

AssemblyInfo.cs文件通常包含有关公司，产品以及版权信息的元数据。

```
[assembly: AssemblyTitle("ClassLibrary1")]
[assembly: AssemblyDescription("")]
[assembly: AssemblyConfiguration("")]
[assembly: AssemblyCompany("")]
[assembly: AssemblyProduct("ClassLibrary1")]
[assembly: AssemblyCopyright("Copyright ©  2015")]
[assembly: AssemblyTrademark("")]
[assembly: AssemblyCulture("")]
```

自定义特性

应用特性的语法和之前见过的其他语法很不相同。你可能会觉得特性跟结构是完全不同的类型，其实不是，特性只是某个特殊结构的类。所有的特性类都派生自System.Attribute。

声明自定义特性

声明一个特性类和声明其他类一样。有下面的注意事项

声明一个派生自System.Attribute的类

给它起一个以后缀Attribute结尾的名字

(安全起见，一般我们声明一个sealed的特性类)

特性类声明如下：

```
        public sealed class MyAttributeAttribute : System.Attribute{
...
```

特性类的公共成员可以是

字段

属性

构造函数

构造函数

特性类的构造函数的声明跟普通类一样，如果不写系统会提供一个默认的，可以进行重载

构造函数的调用

```
[MyAttribute("a value")]          调用特性类的带有一个字符串的构造函数

[MyAttribute("Version 2.3","Mackel")]   //调用特性类带有两个字符串的构造函数
```

构造函数的实参，必须是在编译期间能确定值的常量表达式

如果调用的是无参的构造函数，那么后面的（）可以不写

构造函数中的位置参数和命名参数

```
public sealed class MyAttributeAttribute:System.Attribute{
        public string Description;
        public string Ver;
        public string Reviewer;
        public MyAttributeAttribute(string desc){
                Description = desc;
        }
}
        //位置参数（按照构造函数中参数的位置）
                        //命名参数，按照属性的名字进行赋值
[MyAttribute("An excellent class",Reviewer = "Amy McArthur",Ver="3.12.3")]
```

限定特性的使用

有一个很重要的预定义特性可以用来应用到自定义特性上，那就是AttributeUsage特性，我们可以使用它来限制特性使用在某个目标类型上。

例如，如果我们希望自定义特性MyAttribute只能应用到方法上，那么可以用如下形式使用AttributeUsate:

```
[AttributeUsage(AttributeTarget.Method)]
public sealed class MyAttributeAttribute : System.Attribute{
...
```

AttributeUsage有是三个重要的公共属性如下:

ValidOn 保存特性能应用到的目标类型的列表，构造函数的第一个参数就是AttributeTarget 类型的枚举值

Inherited 一个布尔值，它指示特性是否会被特性类所继承 默认为true

AllowMutiple 是否可以多个特性的实例应用到一个目标上 默认为false

AttributeTarget枚举的成员

All Assembly Class Constructor Delegate Enum Event Field
GenericParameter Interface Method Module Parameter Property ReturnValue Struct

多个参数之间使用按位或|运算符来组合

AttributeTarget.Method|AttributeTarget.Constructor

自定义特性一般遵守的规范

特性类应该表示目标结构的一些状态

如果特性需要某些字段，可以通过包含具有位置参数的构造函数来收集数据，可选字段可以采用命名参数按需初始化

除了属性之外，不要实现公共方法和其他函数成员

为了更安全，把特性类声明为sealed

在特性声明中使用AttributeUsage来指定特性目标组

案例

```
[AttributeUsage(AttributeTarget.Class)]
public sealed class ReviewCommentAttribute:System.Attribute{
        public string Description{get;set;}
```

```
        public string VersionNumber{get;set;}
        public string ReviewerID{get;set;}
        public ReviewerCommentAttribute(string desc,string ver){
                Description = desc;
                VersionNumber = ver;
        }
}
```

访问特性(消费特性)

1,我们可以使用IsDefined方法来，通过Type对象可以调用这个方法，来判断这个类上是否应用了某个特性

```
bool isDefined = t.IsDefined( typeof(AttributeClass),false )
```

第一个参数是传递的需要检查的特性的Type对象

第二个参数是一个bool类型的，表示是否搜索AttributeClass的继承树

2, `object[] attArray = t.GetCustomAttributes(false);`

它返回的是一个object的数组，我们必须将它强制转换成相应类型的特性类型

bool的参数指定了它是否搜索继承树来查找特性

调用这个方法后，每一个与目标相关联的特性的实例就会被创建

线程，任务和同步

线程

对于所有需要等待的操作，例如移动文件，数据库和网络访问都需要一定的时间，此时就可以启动一个新的线程，同时完成其他任务。一个进程的多个线程可以同时运行在单核CPU上或多核CPU的不同内核上。

线程是程序中独立的指令流。在VS编辑器中输入代码的时候，系统会分析代码，用下划线标注遗漏的分号和其他语法错误，这就是用一个后台线程完成。Word文档需要一个线程等待用户输入，另一个线程进行后台搜索，第三个线程将写入的数据存储在临时文件中。运行在服务器上的应用程序中等待客户请求的线程成为侦听器线程。

进程包含资源，如Window句柄，文件系统句柄或其他内核对象。每个进程都分配的虚拟内存。一个进程至少包含一个线程。

一个应用程序启动，会启动一个进程（应用程序运行的载体），然后进程启动多个线程。

程序 - > 进程 - > 线程

程序（我们写的代码 - 静态概念）

进程 -> 正在进行中的程序（正在运行的）

进程是操作系统启动起来

线程是在进程中启动起来

异步委托

创建线程的一种简单方式是定义一个委托,并异步调用它。 委托是方法的类型安全的引用。Delegate类 还支持异步地调用方法。在后台,Delegate类会创建一个执行任务的线程。

接下来定义一个方法，使用委托异步调用（开启一个线程去执行这个方法）

```
        static int TakesAWhile(int data,int ms){
                Console.WriteLine("TakesAWhile started!");
                Thread.Sleep(ms);//程序运行到这里的时候会暂停ms毫秒,然后继续运行下一语句
                Console.WriteLine("TakesAWhile completed");
                return ++data;
        }
        public delegate int TakesAWhileDelegate(int data,int ms);// 声明委托
        static void Main(){
                TakesAWhileDelegate d1 = TakesAWhile;
                IAsyncResult ar = d1.BeginInvoke(1,3000,null,null);
                while(ar.IsCompleted ==false ){
                        Console.Write(".");
                        Thread.Sleep(50);
                }
                int result = d1.EndInvoke(ar);
                Console.WriteLine("Res:"+result);
        }
```

等待句柄IAsyncResult.AsyncWaitHandle

当我们通过BeginInvoke开启一个异步委托的时候，返回的结果是IAsyncResult，我们可以通过它的AsyncWaitHandle属性访问等待句柄。这个属性返回一个WaitHandler类型的对象，它中的WaitOne（）方法可以等待委托线程完成其任务，WaitOne方法可以设置一个超时时间作为参数（要等待的最长时间），如果发生超时就返回false。

```
static void Main(){
        TakesAWhileDelegate d1 = TakesAWhile;
```

```
        IAsyncResult ar = d1.BeginInvoke(1,3000,null,null);
        while(true){
                Console.Write(".");
                if(ar.AsyncWaitHanle.WaitOne(50,false)){
                        Console.WriteLine("Can get result now");
                        break;
                }

        }
        int result = d1.EndInvoke(ar);
        Console.WriteLine("Res:"+result);
}
```

Handle 翻译 句柄 实际=》处理器、处理方法

异步回调-回调方法

等待委托的结果的第3种方式是使用异步回调。在BeginInvoke的第三个参数中，可以传递一个满足AsyncCallback委托的方法，AsyncCallback委托定义了一个IAsyncResult类型的参数其返回类型是void。对于最后一个参数，可以传递任意对象，以便从回调方法中访问它。（我们可以设置为委托实例，这样就可以在回调方法中获取委托方法的结果）

```
static void Main(){
        TakesAWhileDelegate d1 = TakesAWhile;
        d1.BeginInvoke(1,3000,TakesAWhileCompleted,d1);
        while(true){
                Console.Write(".");
                Thread.Sleep(50);
        }
}
static void TakesAWhileCompleted(IAsyncResult ar){//回调方法是从委托线程中调用的，并不是从主线程调用的，可以认为是委托线程最后要执行的程序
        if(ar==null) throw new ArgumentNullException("ar");
        TakesAWhileDelegate d1 = ar.AsyncState as TakesAWhileDelegate;
        int result = d1.EndInvoke(ar);
        Console.Write("Res:"+result);
}
```

异步回调-Lambda表达式

等待委托的结果的第3种方式是使用异步回调。在BeginInvoke的第三个参数中，可以传递一个满足AsyncCallback委托的方法，AsyncCallback委托定义了一个IAsyncResult类型的参数其返回类型是void。对于最后一个参数，可以传递任意对象，以便从回调方法中访问它。（我们可以设置为委托实例，这样就可以在回调方法中获取委托方法的结果）

```
static void Main(){
        TakesAWhileDelegate d1 = TakesAWhile;
        d1.BeginInvoke(1,3000,ar=>{
                        int result = d1.EndInvoke(ar);
                        Console.WriteLine("Res:"+result);
                        },null);
        while(true){
                Console.Write(".");
                Thread.Sleep(50);
        }
}
```

Thread类

使用Thread类可以创建和控制线程。Thread构造函数的参数是一个无参无返回值的委托类型。

```
static void Main(){
        var t1 = new Thread(ThreadMain);
        t1.Start();
        Console.WriteLine("This is the main thread.");
}
static void ThreadMain(){
        Console.WriteLine("Running in a thread.");
}
```

在这里哪个先输出是无法保证了线程的执行有操作系统决定，只能知道Main线程和分支线程是同步执行的。在这里给Thread传递一个方法，调用Thread的Start方法，就会开启一个线程去执行，传递的方法。

Thread类-Lambda表达式

上面直接给Thread传递了一个方法，其实也可以传递一个Lambda表达式。(委托参数的地方都可以使用Lambda表达式)

```
static void Main(){
        var t1 = new Thread( ()=>Console.WriteLine("Running in a thread, id : "+Thread.CurrentThread.ManagedThreadId) );
        t1.Start();
        Console.WriteLine("This is the main Thread . ID : "+Thread.CurrentThread.ManagedThreadId)
}
```

给线程传递数据-通过Start

给线程传递一些数据可以采用两种方式，一种方式是使用Start，一种方式是创建一个自定义的类，把线程的方法定义为实例方法，这样就可以初始化实例的数据，之后启动线程。

```
        public struct Data{//声明一个结构体用来传递数据
                public string Message;
        }
        static void ThreadMainWithParameters(Object o ){
                Data d=(Data)o;
                Console.WriteLine("Running in a thread , received :"+d.Message);
        }
        static void Main(){
                var d = new Data{Message = "Info"};
                var t2 = new Thread(ThreadMainWithParameters);
                t2.Start(d);
        }
```

给线程传递数据-自定义类

```
public class MyThread{
        private string data;
        public MyThread(string data){
                this.data = data;
        }
        public void ThreadMain(){
                Console.WriteLine("Running in a thread , data : "+data);
        }
```

```
}
var obj = new MyThread("info");
var t3 = new Thread(obj.ThreadMain);
t3.Start();
```

后台线程和前台线程

只有一个前台线程在运行，应用程序的进程就在运行，如果多个前台线程在运行，但是Main方法结束了，应用程序的进程仍然是运行的，直到所有的前台线程完成其任务为止。

在默认情况下，用Thread类创建的线程是前台线程。线程池中的线程总是后台线程。

在用Thread类创建线程的时候，可以设置IsBackground属性，表示它是一个前台线程还是一个后台线程。

看下面例子中前台线程和后台线程的区别：

```
class Program{
        static void Main(){
                var t1 = new Thread(ThreadMain){IsBackground=false};
                t1.Start();
                Console.WriteLine("Main thread ending now.");
        }
        static void ThreadMain(){
                Console.WriteLine("Thread +"+Thread.CurrentThread.Name+"
started");
                Thread.Sleep(3000);
                Console.WriteLine("Thread +"+Thread.CurrentThread.Name+"
started");
        }
}
```

后台线程用的地方：如果关闭Word应用程序，拼写检查器继续运行就没有意义了，在关闭应用程序的时候，拼写检查线程就可以关闭。

当所有的前台线程运行完毕，如果还有后台线程运行的话，所有的后台线程会被终止掉。

线程的优先级

线程有操作系统调度，一个CPU同一时间只能做一件事情（运行一个线程中的计算任务），当有很多线程需要CPU去执行的时候，线程调度器会根据线程的优先级去判断先去执行哪一个线程，如果优先级相同的话，就使用一个循环调度规则，逐个执行每个线程。

在Thead类中，可以设置Priority属性，以影响线程的基本优先级 ,Priority属性是一个ThreadPriority枚举定义的一个值。定义的级别有Highest ,AboveNormal,Normal,BelowNormal 和 Lowest。

控制线程

1,获取线程的状态（Running还是Unstarted,,,），当我们通过调用Thread对象的Start方法，可以创建线程，但是调用了Start方法之后，新线程不是马上进入Running状态，而是出于Unstarted状态，只有当操作系统的线程调度器选择了要运行的线程，这个线程的状态才会修改为Running状态。我们使用Thread.Sleep()方法可以让当前线程休眠进入WaitSleepJoin状态。

2,使用Thread对象的Abort()方法可以停止线程。调用这个方法，会在终止要终止的线程中抛出一个ThreadAbortException类型的异常，我们可以try catch这个异常，然后在线程结束前做一些清理的工作。

3,如果需要等待线程的结束，可以调用Thread对象的Join方法，表示把Thread加入进来，停止当前线程，并把它设置为WaitSleepJoin状态，直到加入的线程完成为止。

线程池

创建线程需要时间。 如果有不同的小任务要完成,就可以事先创建许多线程 ，在应完成这些任务时发出请求。 这个线程数最好在需要更多的线程时增加,在需要释放资源时减少。不需要 自己创建线程池，系统已经有一个ThreadPool类管理线程。 这个类会在需要时增减池中线程的线程数,直到达到最大的线程数。 池中的最大线程数是可配置的。 在双核 CPU 中 ,默认设置为1023个工作线程和 1000个 I/o线程。也可以指定在创建线程池时应立即启动的最小线程数,以及线程池中可用的最大线程数。 如果有更多的作业要处理,线程池中线程的个数也到了极限,最新的作业就要排队,且必须等待线程完成其任务。

线程池示例

```
static void Main(){
        int nWorkerThreads;
        int nCompletionPortThreads;
        ThreadPool.GetMaxThreads(out nWorkerThreads,out nCompletionPortThreads);
        Console.WriteLine("Max worker threads : " +nWorkerThreads+" I/O completion threads
:"+nCompletionPortThreads );
        for(int i=0;i<5;i++){
                ThreadPool.QueueUserWorkItem(JobForAThread);
        }
        Thread.Sleep(3000);
```

```
}
static void JobForAThread(object state){
        for(int i=0;i<3;i++){
                Console.WriteLine("Loop "+i+" ,running in pooled thread
"+Thread.CurrentThread.ManagedThreadId);
                Thread.Sleep(50);
        }
}
```

示例应用程序首先要读取工作线程和 I/o线程的最大线程数,把这些信息写入控制台中。接着在 for循 环中,调 用ThreadPool.QueueUserWorkItem方法,传递一个WaitCallBack类型的委托 ,把 JobForAThread方法赋予线程池中的线程。 线程池收到这个请求后,就会从池中选择一个线程 来调用该方法。 如果线程池还没有运行,就会创建一个线程池,并启动第一个线程。 如果线程池已经在运行,且有一个空闲线程来完成该任务,就把该作业传递给这个线程。

使用线程池需要注意的事项:

线程池中的所有线程都是后台线程 。 如果进程的所有前台线程都结束了,所有的后台线程就会停止。 不能把入池的线程改为前台线程 。

不能给入池的线程设置优先级或名称。

入池的线程只能用于时间较短的任务。 如果线程要一直运行(如 Word的拼写检查器线程),就应使用Thread类创建一个线程。

任务

在.NET4 新的命名空间System.Threading.Tasks包含了类抽象出了线程功能，在后台使用的ThreadPool进行管理的。任务表示应完成某个单元的工作。这个工作可以在单独的线程中运行，也可以以同步方式启动一个任务。

任务也是异步编程中的一种实现方式。

启动任务

启动任务的三种方式:

```
TaskFactory tf = new TaskFactory();
Task t1 = tf.StartNew(TaskMethod);

Task t2 = TaskFactory.StartNew(TaskMethod); - 不再支持

Task t3 = new Task(TaskMethod);
t3.Start();
```

我们创建任务的时候有一个枚举类型的选项TaskCreationOptions

连续任务

如果一个任务t1的执行是依赖于另一个任务t2的，那么就需要在这个任务t2执行完毕后才开始执行t1。这个时候我们可以使用连续任务。

```
static void DoFirst(){
        Console.WriteLine("do  in task : "+Task.CurrentId);
        Thread.Sleep(3000);
}
static void DoSecond(Task t){
        Console.WriteLine("task "+t.Id+" finished.");
        Console.WriteLine("this task id is "+Task.CurrentId);
        Thread.Sleep(3000);
}
Task t1 = new Task(DoFirst);
Task t2 = t1.ContinueWith(DoSecond);
Task t3 = t1.ContinueWith(DoSecond);
Task t4 = t2.ContinueWith(DoSecond);

Task t5 = t1.ContinueWith(DoError,TaskContinuationOptions.OnlyOnFaulted);
```

任务层次结构

我们在一个任务中启动一个新的任务，相当于新的任务是当前任务的子任务，两个任务异步执行，如果父任务执行完了但是子任务没有执行完，它的状态会设置为WaitingForChildrenToComplete，只有子任务也执行完了，父任务的状态就变成RunToCompletion

```
static void Main(){
        var parent = new Task(ParentTask);
        parent.Start();
        Thread.Sleep(2000);
        Console.WriteLine(parent.Status);
        Thread.Sleep(4000);
        Console.WriteLine(parent.Status);
        Console.ReadKey();
}
static void ParentTask(){
```

```
        Console.WriteLine("task id "+Task.CurrentId);
        var child = new Task(ChildTask);
        child.Start();
        Thread.Sleep(1000);
        Console.WriteLine("parent started child , parent end");
}
static void ChildTask(){
        Console.WriteLine("child");
        Thread.Sleep(5000);
        Console.WriteLine("child finished ");
}
```

线程问题-争用条件（资源访问冲突）

```
public class StateObject{
        private int state = 5;
        public void ChangeState(int loop){
                if(state==5){
                        state++;//6
                        Console.WriteLine("State==5:"+state==5+"  Loop:"+loop);//false
                }
                state = 5;
        }
}

static void RaceCondition(object o ){
        StateObject state = o as StateObject;
        int i = 0;
        while(true){
                state.ChangeState(i++);
        }
}
static void Main(){
        var state = new StateObject();
        for(int i=0;i<20;i++){
                new Task(RaceCondition,state).Start();
        }
        Thread.Sleep(10000);
}
```

使用lock（锁）解决争用条件的问题

```
static void RaceCondition(object o ){
        StateObject state = o as StateObject;
        int i = 0;
        while(true){
                lock(state){
                        state.ChangeState(i++);
                }

        }
}
```

另外一种方式是锁定StateObject中的state字段，但是我们的lock语句只能锁定个引用类型。因此可以定义一个object类型的变量sync，将它用于lock语句，每次修改state的值的时候，都使用这个一个sync的同步对象。就不会出现争用条件的问题了。下面是改进后的ChangeState方法

```
private object sync = new object();
public void ChangeState(int loop){
        lock(sync){
                if(state==5){
                        state++;
                        Console.WriteLine("State==5:"+state==5+"  Loop:"+loop);
                }
                state = 5;
        }
}
```

线程问题-死锁

```
public class SampleThread{
        private StateObject s1;
        private StateObject s2;
        public SampleThread(StateObject s1,StateObject s2){
                this.s1= s1;
                this.s2 = s2;
        }
        public void Deadlock1(){
                int i =0;
                while(true){
                        lock(s1){
                                lock(s2){
```

```
                                        s1.ChangeState(i);
                                        s2.ChangeState(i);
                                        i++;
                                        Console.WriteLine("Running i : "+i);
                                }
                        }
                }
        }
        public void Deadlock2(){
                int i =0;
                while(true){
                        lock(s2){
                                lock(s1){
                                        s1.ChangeState(i);
                                        s2.ChangeState(i);
                                        i++;
Console.WriteLine("Running i : "+i);
                                }
                        }
                }
        }
}
var state1 = new StateObject();
var state2 = new StateObject();
new Task(new SampleTask(s1,s2).DeadLock1).Start();
new Task(new SampleTask(s1,s2).DeadLock2).Start();
```

如何解决死锁

在编程的开始设计阶段，设计锁定顺序

文件操作

学习的知识点

1，通过FileInfo和DirectoryInfo类来读取文件和文件夹属性

查看文件属性，创建文件，移动文件，重命名文件

判断路径是否存在，创建目录

2，通过File读写文件

读写文件

3，使用流来读写文件

FileStream

StreamReader（读取流-读取数据） StreamWriter（写入流-向别人传输）

文件系统

下面的类用于浏览文件系统和执行操作，比如移动，复制和删除文件。

System.MarshalByRefObject　　这个是.NET类中用于远程操作的基对象类，它允许在应用程序域之间编组数据。

FileSystemInfo　　这是表示任何文件系统对象的基类

FileInfo和File　　这些类表示文件系统上的文件

DirectoryInfo和Directory　　表示文件系统上的文件夹

Path　　包含用于处理路径名的一些静态方法

DriveInfo　　它的属性和方法提供了指定驱动器的信息

表示文件和文件夹的.NET类

我们有两个用于表示文件夹的类和两个用于表示文件的类

Directory（文件夹）和File（文件）类只包含静态方法，不能被实例化。如果只对文件夹或文件执行一个操作，使用这些类就很有效，省去了去实例化.NET类的系统开销。

DirectoryInfo类和FileInfo类实现与Directory和File相同的公共方法，他们拥有一些公共属性和构造函数，这些类的成员都不是静态的。需要实例化这些类，之后把每个实例与特定的文件夹或者文件关联起来。如果使用同一个对象执行多个操作，使用这些类比较合适，这是因为在构造的时候他们将读取合适文件系统对象的身份验证和其他信息，无论每个对象调用了多少方法，都不需要再次读取这些信息。

FileInfo和DirectoryInfo类

对于FileInfo和DirectoryInfo类中的很多方法也可以使用File和Directory中的很多方法实现。

1,完成一个文件的拷贝

```
FileInfo myFile = new FileInfo(@"c:\pxx\xx\xxx\xxx.txt");
myFile.CopyTo(@"d:\xx\xx.txt");//拷贝文件
```

对应的File处理方式

```
File.Copy(@"c:\xxx\xx\xx\xx.txt",@"d:\xx\xx\xx.txt");
```

2,判断一个文件夹是否存在

```
DirectoryInfo myFolder = new DirectoryInfo(@"c:\program files");
```

myFolder.Exists

对于FileInfo，或者DirectoryInfo进行构造的时候，如果传递了一个不存在的文件或者文件夹路径，这个时候不会出现异常，只有当你使用这个文件或者文件夹的时候才会出现问题。

FileInfo和DirectoryInfo的对象都可以通过Exists属性判断这个文件或者文件夹是否存在。

FileInfo和DirectoryInfo的属性列表

CreationTime　创建文件或文件夹的时间

DirectoryName(用于FileInfo)　包含文件夹的完整路径

Parent(用于DirectoryInfo)　指定子目录的父目录

Exists　文件或文件夹是否存在

Extension　文件的扩展名，对于文件夹，它返回空白

FullName　文件或文件夹的完整路径名

LastAccessTime　最后一次访问文件或文件夹的时间

LastWriteTime　最后一个修改文件或文件夹的时间

Name　文件或文件夹的名称

Root（仅用于DirectoryInfo）　路径的根部分

Length（仅用于FileInfo）　返回文件的大小（以字节为单位）

FileInfo和DirectoryInfo的方法列表

Create()　创建给定名称的文件夹或者空文件，对于FileInfo,该方法会返回一个流对象，以便于写入文件。

Delete()　删除文件或文件夹。对于文件夹有一个可以递归的Delete选项

MoveTo()　移动或重命名文件或文件夹

CopyTo()　(只用于FileInfo)复制文件，文件夹没有复制方法，如果想要复制完整的目录树，需要单独复制每个文件和文件夹

GetDirectories()　(只适用于DirectoryInfo)返回DirectoryInfo对象数组，该数组表示文件夹中包含的所有文件夹

GetFiles()　(只适用于DirectoryInfo)返回FileInfo对象数组，该数组表示文件夹中所有的文件

GetFileSystemInfos()　(只适用于DirectoryInfo)返回FileInfo和DirectoryInfo对象，它把文件夹中包含的所有对象表示为一个FileSystemInfo引用数组

小案例

创建时间，最后一次访问时间和最后一次写入时间都是可写入的。

```
FileInfo test = new FileInfo(@"c:\myfile.txt");
Console.WriteLine(test.Exists);
Console.WriteLine(test.CreationTime);
test.CreationTime = new DataTime(2010,1,1,7,20,0);
Console.WriteLine(test.CreationTime);
```

Path类

我们不能去实例化Path类，Path类提供了一些静态方法，可以更容易的对路径名执行操作。

```
Console.WriteLine(Path.Combine(@"c:\my documents","Readme.txt"));
```

在不同的操作系统上，路径的表示是不一样的 windows上是 \ ，在Unix就是/ ,我们可以使用Path.Combine连接两个路径，不用关心在哪个系统上。

Path类的一些静态字段

AltDirectorySeparatorChar　　提供分割目录的字符，在windows上使用 \ 在Unix上用 /

DirectorySpeparatorChar 提供分割目录的字符，在windows上使用 / 在Unix上用 \

PathSeparator　　提供一种与平台无关的方式，来指定划分环境变量的路径字符串，默认为分号

VolumeSepartorChar　　提供一种与平台无关的方式，来指定容量分割符，默认为冒号

读写文件

在.NET 2.0扩展了File类，通过File可以读写文件。

读取文件

在.NET 2.0扩展了File类，通过File可以读写文件。

1,File.ReadAllText(FilePath);根据文件路径读取文件中所有的文本

2,File.ReadAllText(FilePath,Encoding);//Encoding可以指定一个编码格式

Encoding.ASCII;

3,File.ReadAllBytes()方法可以打开二进制文件把内容读入一个字节数组

4,File.ReadAllLines() 以行的形式读取文件，一行一个字符串，返回一个字符串的数组

写入文件

我们读取文件有ReadAllText() ReadAllLines()和ReadAllBytes()这几个方法，对应的写入文件的方法有WriteAllText() WriteAllLines()和WriteAllBytes()

流

什么是流

流是一种数据的处理方式

为什么使用流处理数据？

数据小，可以直接一次性搬运，数据大，可以把数据当做水，接一个水管，一点一点搬运。

流媒体

流是一个用于传输数据的对象，数据可以向两个方向传输：

如果数据从外部源传输到程序中，这就是读取流

如果数据从程序传输到外部源中，这就是写入流

外部源可能是

一个文件

网络上的数据

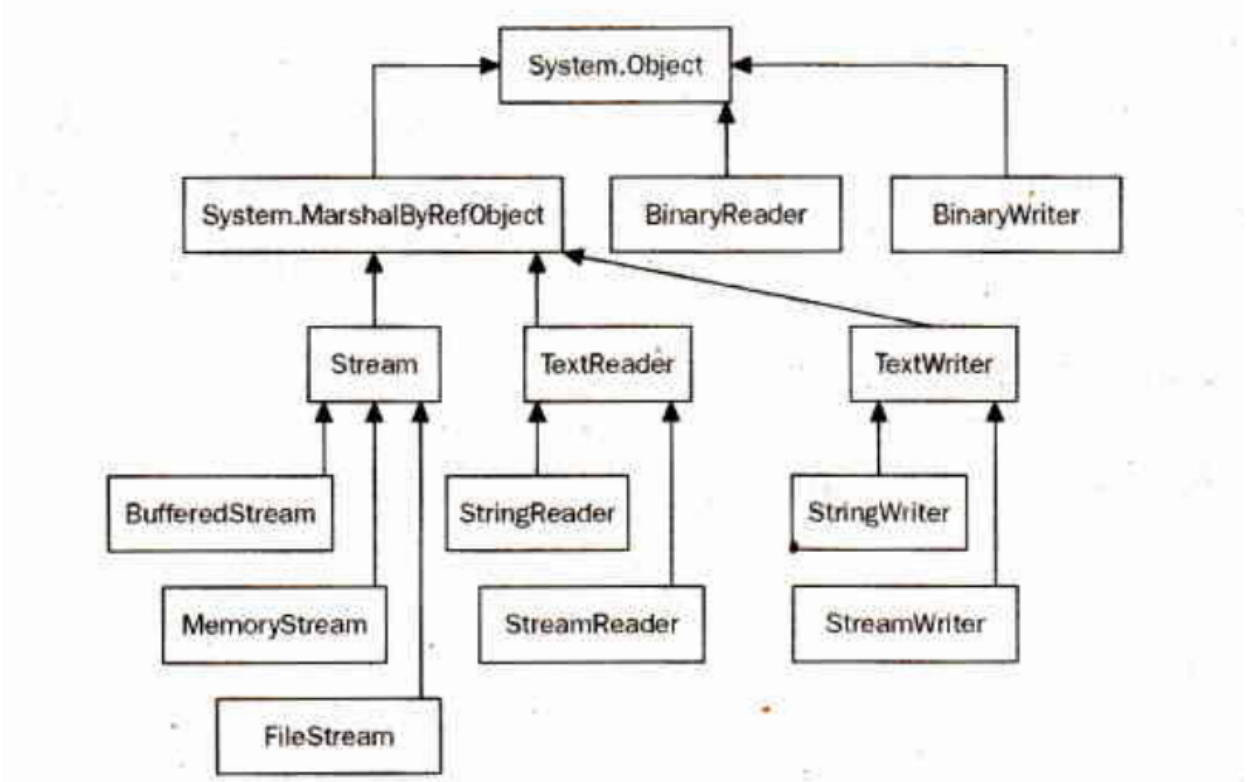
内存区域上

读写到命名管道上

读写内存使用System.IO.MemorySystem

处理网络数据使用System.Net.Sockets.NetworkStream

与流相关的类



FileStream(文件流) 这个类用于任意文件（包括二进制文件）中读写。

StreamReader(流读取器)和StreamWriter(流写入器)专门用于读写文本文件

使用FileStream类读写二进制文件

FileStream实例用于读写文件中的数据，要构造FileStream实例，需要提供下面的4中信息:

要访问的文件 - 一般提供一个文件的完整路径名

表示如何打开文件的模式 - 新建文件或打开一个现有文件，如果打开一个现有的文件，写入操作是覆盖文件原来的内容，还是追加到文件的末尾?

表示访问文件的方式 - 只读 只写 还是读写

共享访问 - 表示是否独占访问文件，如果允许其他流同时访问文件，则这些流是只读 只写 还是读写文件

构造函数的参数

构造函数的参数的取值

FileMode(打开模式)　　Append,Create,CreateNew,Open,OpenOrCreate和Truncate

FileAccess(读取还是写入)　　Read,ReadWrite和Write

FileShare(文件共享设置) Delete,Inheritable,None,Read,ReadWrite和Write

注意

如果文件不存在 Append Open和Truncate会抛出异常

如果文件存在 CreateNew会抛出异常

Create 和 OpenOrCreate Create会删除现有的文件，新建一个空的文件,OpenOrCreate会判断当前是否有文件，没有的话才会创建

FileMode和FileAccess 解释：

https://blog.csdn.net/qq_42351033/article/details/88072011

创建FileStream文件流

```
FileStream fs = new FileStream(@"c:\xx\xx.doc",FileMode.Create);
FileStream fs2 = new
FileStream(@"c:\xx\xx.doc",FileMode.Create,FileAccess.Write);
FileStream fs3 = new
FileStream(@"c:\xx\xx.doc",FileMode.Create,FileAccess.Write,FileShare.None);
```

通过FileInfo打开文件流

```
FileInfo myfile1 = new FileInfo(@"c:\xx\xx.doc");
FileStream fs4= myfile1.OpenRead();
FileInfo myfile2 = new FileInfo(@"c:\xx\xx.doc");
FileStream fs5= myfile2.OpenWrite();
FileInfo myfile3 = new FileInfo(@"c:\xx\xx.doc");
FileStream fs6= myfile3.Open(FileMode.Append,FileAccess.Write,FileShare.None);
FileInfo myfile4 = new FileInfo(@"c:\xx\xx.doc");
FileStream fs7= myfile4.Create();
```

FileStream文件流的关闭

当我们使用完了一个流之后，一定要调用fs.Close();方法去关闭流，关闭流会释放与它相关联的资源，允许其他应用程序为同一个文件设置流。这个操作也会刷新缓冲区。

从文件流中读取内容和写入内容

从文件流读取内容

1,int nextByte = fs.ReadByte();读取一个字节(0-255)的数据,返回结果，如果达到流的末尾，就返回-1

2,int nBytesRead = fs.Read(ByteArray,0,nBytes);//读取多个字节，第一个是存放的数组，第二个参数是开始存放的索引，第三个参数是要读取多少个字节。返回的结果是读取的自己的实际个数，如果达到流的末尾 返回-1

向文件流写入内容

1,byte NextByte = 100;fs.WriteByte(NextByte);把一个字节写入文件流中

2,fs.Write(ByteArray,0,nBytes); 把一个自己数组写入文件流中 参数同上

读取流 - 写入流

输入流　输出流

读写文本文件

我们对文本文件的读写一般使用StreamReader和StreamWriter,因为不同的文本有不同的编码格式，这个StreamReader会帮我们自动处理，所以我们不需要关心文本文件的编码是什么

1，创建文本的读取流(会检查字节码标记确定编码格式)

```
StreamReader sr = new StreamReader(@"c:\xx\ReadMe.txt");
```

2，指定编码格式

```
StreamReader str = new StreamReader(@"c:\xx\xx.txt",Encoding.UTF8);
```

(可取的编码格式 ASCII Unicode UTF7 UTF8 UTF32)

3，在文件流的基础上创建文本读取流

```
FileStream fs = new
FileStream(@"c:\xx\xx.txt",FileMode.Open,FileAccess.Read,FileShare.None);
StreamReader sr = new StreamReader(fs);
```

4，通过文件信息创建文本读取流-第二种方式

```
FileInfo myFile = new FileInfo(@"c:\xx\xx.txt");
StreamReader sr = myFile.OpenText();
```

流的关闭 sr.Close();

读取文本文件

1,string nextLine = sr.ReadLine();//读取一行字符串

2,string restOfStream = sr.ReadToEnd();//读取流中所有剩余的文本内容

3,int nextChar = sr.Read();//只读取一个字符

4,

```
int nChars = 100;
char[] charArray = new char[nChars];
```

int nCharsRead = sr.Read(charArray,0,nChars); 读取多个字符，第一个参数是要存放的字符数组，第二个参数是从数组的哪一个索引开始放，第三个参数是读取多少个字符 返回值是实际读取的字符的个数

文本写入流StreamWriter-创建

StreamWriter的创建

1,StreamWriter sw = new StreamWriter(@"c:\xx\xx.txt");(默认使用UTF-8编码)

2,StreamWriter sw = new StreamWriter(@"c:\xx\xx.txt",true,Encoding.ASCII)

第二个参数表示是否以追加的方式打开, 第三个参数是编码方式

3,通过FileStream创建StreamWriter

```
FileStream fs = new
FileStream(@"c:\xx\xx.txt",FileMode.CreateNew,FileAccess.Write,FileShare.Read);
StreamWriter sw = new StreamWriter(fs);
```

4,通过FileInfo创建StreamWriter

```
FileInfo myFile = new FileInfo(@"c:\xx\xx.txt");
StreamWriter sw = myFile.CreateText();
```

所有流用完之后关闭 sw.Close();

文本写入流StreamWriter-写入

1,写入一行字符

```
string nextLine = "x xx x x x x ";sw.Write(nextLine);
```

2,写入一个字符

```
char nextChar = 'a';
sw.Write(nextChar);
```

3,写入字符数组

```
char[] charArray = ..;
sw.Write(charArray);
```

4,写入字符数组的一部分

```
sw.Write(charArray,StartIndex,Length);
```

1: 要写入的数组 2: 开始索引 3写入长度

网络

网络是什么?

从远程服务器上获取数据

把本地数据上传到服务器上

（游戏）服务器开发的基础

WebClient概述

从MSDN中我们可以得知，WebClient的作用就是“Provides common methods for sending data to and receiving data from a resource identified by a URI.”也就是说我们可以通过这个类去访问与获取网络上的资源文件。

WebClient类不能被继承,我们可以通过WebRequest和WebResponse这两个类来处理向URI标示的资源和获取数据了。这两个类功能挺强大的，但不足之处的是利用WebRequest和WebResponse时设置过于复杂，使用起来颇为费劲。而WebClient可以理解为对WebRequest和WebResponse等协作的封装。它使人们使用起来更加简单方便，然后它也有先天不足的地方。那就是缺少对cookies/session的支持,用户无法对是否自动url转向的控制，还有就是缺少对代理服务器的支持等等，不过我们可以通过重写WebClient的一些方法来实现这些功能。

WebClient的函数与基本用法

WebClient提供四种将数据上载到资源的方法:

OpenWrite 返回一个用于将数据发送到资源的 Stream。

UploadData 将字节数组发送到资源并返回包含任何响应的字节数组。

UploadFile 将本地文件发送到资源并返回包含任何响应的字节数组。

UploadValues 将 NameValueCollection 发送到资源并返回包含任何响应的字节数组。

另外WebClient还提供三种从资源下载数据的方法:

DownloadData 从资源下载数据并返回字节数组。

DownloadFile 从资源将数据下载到本地文件。

OpenRead 从资源以 Stream 的形式返回数据。

WebClient与其他网络相关类的区别

WebClient和HttpWebRequst是用来获取数据的2种方式，一般而言，WebClient更倾向于“按需下载”，事实上掌握它也是相对容易的，而HttpWebRequst则允许你设置请求头或者对内容需要更多的控制，后者有点类似于form中的submit。虽然两者都是异步请求事件，但是WebClient是基于事件的异步，而HttpWebRequst是基于代理的异步编程。

WebClient使用范例

```
private void button1_Click(object sender, RoutedEventArgs e)
```

```
        {
            //通过WebClient方式去获取资源文件
            Uri uri = new Uri("http://localhost:2052/Images/cnblogs.png", UriKind.Absolute);
            WebClient webClient = new WebClient();
            webClient.OpenReadAsync(uri);
            webClient.OpenReadCompleted += new
OpenReadCompletedEventHandler(client_OpenReadCompleted);
        }

        void client_OpenReadCompleted(object sender, OpenReadCompletedEventArgs e)
        {
            Stream stream = e.Result;
            BitmapImage bitmap = new BitmapImage();
            bitmap.SetSource(stream);
            this.image1.Source = bitmap;
        }
```

WebRequest和WebResponse入门案例

```
public Form1()
{
WebRequest wrq = WebRequest.Create("");
WebResponse wrs = wrq.GetResponse();
Stream strm = wrs.GetResponseStream();
StreamReader sr = new StreamReader(strm);
while ( (line = sr.ReadLine()) != null)
{
         Console.WriteLine(line);
}
strm.Close();
}
```

案例分析

WebRequest类是支持不同网络协议的类层次结构的一部分，为了给请求类型接收一个对正确对象的引用，需要一个工厂(factory)机制。WebRequest.Create()方法会为给定的协议创建合适的对象。

WebRequest类代表要给某个URI发送信息的请求，URI作为参数传送给Create()方法。

WebResponse类代表从服务器获取的数据。调用WebRequest.GetResponse()方法，实际上是

把请求发送给Web服务器，创建一个Response对象，检查返回的数据。

HTTP标题信息

HTTP协议的一个重要方面就是能够利用请求和响应数据流发送扩展的标题信息。标题信息可以包括cookies、以及发送请求的特定浏览器(用户代理)的一些详细信息。

WebRequest类和WebResponse类提供了读取标题信息的一些支持。而两个派生的类HttpWebRequest类和HttpWebResponse类提供了其他HTTP特定的信息。用HTTP URI创建WebRequest会生成一个HttpWebRequest对象实例。因为HttpWebRequest派生自WebRequest，可以在需要WebRequest的任何地方使用新实例。

还可以把实例的类型强制转换为HttpWebRequest引用，访问HTTP协议特定的属性。同样，在使用HTTP时，GetResponse()方法调用会返回WebResponse引用，也可以进行一个简单的强制转换，以访问HTTP特定的特性。

```
WebRequest wrq = WebRequest.Create("");
HttpWebRequest hwrq = (HttpWebRequest)wrq;
Console.WriteLine("Request Timeout (ms) = " + wrq.Timeout);
Console.WriteLine("Request Keep Alive = " + hwrq.KeepAlive);
Console.WriteLine("Request AllowAutoRedirect = " + hwrq.AllowAutoRedirect);
```

imeout属性的单位是毫秒，其默认值是100 000。可以设置这个属性，以控制WebRequest对象在产生WebException之前要在响应中等待多长时间。可以检查属性WebException.Status，看看产生异常的原因。这个枚举类型包括超时的状态码、连接失败、协议错误等。

KeepAlive属性是对HTTP协议的特定扩展，所以可以通过HttpWebRequest引用访问这个属性。该属性允许多个请求使用同一个连接，在后续的请求中节省关闭和重新打开连接的时间。其默认值为true。

AllowAutoRedirect属性也是专用于HttpWebRequest类的，使用这个属性可以控制Web请求是否应自动跟随Web服务器上的重定向响应。其默认值也是true。如果只允许有限的重定向，可以把HttpWebRequest的MaximumAutomaticRedirections属性设置为想要的数值。

请求和响应类把大多数重要的标题显示为属性，也可以使用Headers属性本身显示标题的总集合。在GetResponse()方法调用的后面添加如下代码

```
WebRequest wrq = WebRequest.Create("");
```

```
WebResponse wrs = wrq.GetResponse();
WebHeaderCollection whc = wrs.Headers;
for(int i = 0; i < whc.Count; i++)
{
Console.WriteLine("Header " + whc.GetKey(i) + " : " + whc[i]);
}
```

异步页面请求

WebRequest类的另一个特性就是可以异步请求页面。由于在给主机发送请求到接收响应之间有很长的延迟，因此，异步请求页面就显得比较重要。像WebClient.DownloadData()和WebRequest.GetResponse()等方法，在响应没有从服务器回来之前，是不会返回的。

如果不希望在那段时间中应用程序处于等待状态，可以使用BeginGetResponse() 方法和EndGetResponse()方法，BeginGetResponse()方法可以异步工作，并立即返回。在底层，运行库会异步管理一个后台线程，从服务器上接收响应。

BeginGetResponse() 方法不返回WebResponse对象，而是返回一个执行IAsyncResult接口的对象。使用这个接口可以选择或等待可用的响应，然后调用EndGetResponse()搜集结果。

也可以把一个回调委托发送给BeginGetResponse()方法。该回调委托的目的地是一个返回类型为void并把IAsyncResult引用作为参数的方法，当工作线程完成了搜集响应的任务后，运行库就调用该回调委托，通知用户工作已完成。如下面的代码所示，在回调方法中调用EndGetResponse()可以接收WebResponse对象：

```
public Form1()
{
InitializeComponent();
WebRequest wrq = WebRequest.Create("");
wrq.BeginGetResponse(new AsyncCallback(OnResponse), wrq);
}
protected void OnResponse(IAsyncResult ar)
{
WebRequest wrq = (WebRequest)ar.AsyncState;
WebResponse wrs = wrq.EndGetResponse(ar);
// read the response ...
}
```

实用工具类-URI

Uri和UriBuilder是System命名空间下的两个类。UriBuilder可以通过给定的字符串，从而构建一个URI，而Uri类可以分析，组合和比较URI。

1,创建Uri类

```
Uri uri = new Uri("http://www.microsoft.com/somefolder/somefile.htm?order=true");
```

2，分析Uri取得一些属性

```
string query = uri.Query;// ?order=true 得到参数部分
string absPath = uri.AbsolutePath; //   /somefoler/somefile.htm 得到路径部分
string scheme = uri.Scheme; // http 得到协议
int port = uri.Port ; // 80 访问端口
string host = uri.Host; // www.microsoft.com
bool isDefaultPort = uri.IsDefaultPort;
```

番外：URI和URL的区别

https://www.pianshen.com/article/50261935688/

https://www.jianshu.com/p/ba15d066f777

实用工具类-UriBuilder

```
UriBuilder builder = new
UriBuilder("http","www.microsoft.com",80,"somefolder/somefile.htm");
```

也可以单独给每个属性部分赋值

```
builder.Scheme = "http";
builder.Host="www.microsoft.com";
builder.Port = 80;
builder.Path="somefolder/somefile.htm";

//通过builder构建uri
Uri completeUri = builder.Uri;
```

实用工具类-IPAddress

创建IP地址

```
IPAddress ipAddress = IPAddress.Parse("234.34.5.3");
byte[] address = ipAddress.GetAddressBytes();//得到四个位置上的具体值
```

```
string ipString = ipAddress.ToString();//得到ip的字符串
```

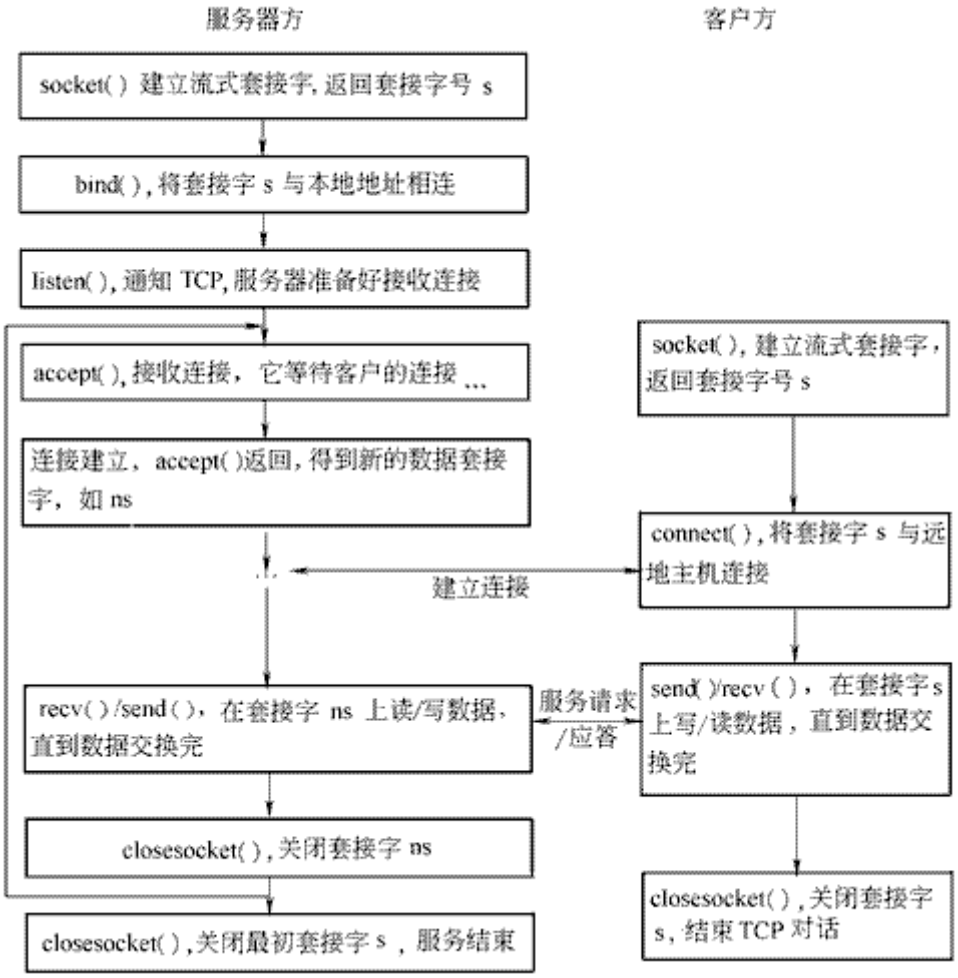
实用工具类-Dns和IPHostEntry

```
IPHostEntry ipentry = Dns.Resolve("www.baidu.com");
IPHostEntry ipentry = Dns.GetHostByAddress("23.34.3.43");
```

通过域名或者ip地址获取主机信息

Socket - 套接字 - 插口、插座

Socket(套接字)编程（Tcp）



1.基于Tcp协议的Socket通讯类似于B/S架构，面向连接，但不同的是服务器端可以向客户端主动推送消息。

使用Tcp协议通讯需要具备以下几个条件：

(1).建立一个套接字(Socket)

(2).绑定服务器端IP地址及端口号--服务器端

(3).利用Listen()方法开启监听--服务器端

(4).利用Accept()方法尝试与客户端建立一个连接--服务器端

(5).利用Connect()方法与服务器建立连接--客户端

(5).利用Send()方法向建立连接的主机发送消息

(6).利用Recive()方法接受来自建立连接的主机的消息(可靠连接)

TcpServer

```
 public static void TcpServer(IPEndPoint serverIP)
        {
            Console.WriteLine("客户端Tcp连接模式");
            Socket tcpServer = new Socket(AddressFamily.InterNetwork, SocketType.Stream,
ProtocolType.Tcp);
            tcpServer.Bind(serverIP);
            tcpServer.Listen(100);
            Console.WriteLine("开启监听...");
            new Thread(() =>
            {
                while (true)
                {
                    try
                    {
                        TcpRecive(tcpServer.Accept());
                    }
                    catch (Exception ex)
                    {
                        Console.WriteLine(string.Format("出现异常: {0}", ex.Message));
                        break;
                    }
                }
            }).Start();
            Console.WriteLine("\n\n输入\"Q\"键退出。");
            ConsoleKey key;
            do
            {
                key = Console.ReadKey(true).Key;
            } while (key != ConsoleKey.Q);
            tcpServer.Close();
        }
```

TcpRecive

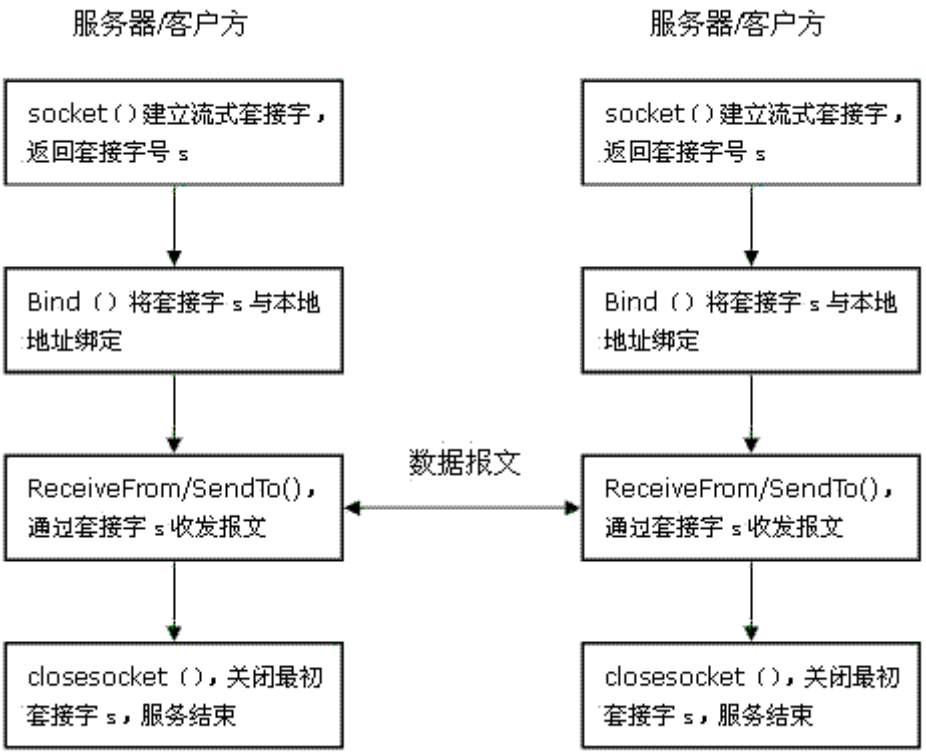
```
 public static void TcpRecive(Socket tcpClient)
        {
            new Thread(() =>
            {
                while (true)
                {
                    byte[] data = new byte[1024];
                    try
                    {
                        int length = tcpClient.Receive(data);
                    }
                    catch (Exception ex)
                    {
                        Console.WriteLine(string.Format("出现异常: {0}", ex.Message));
                        break;
                    }
                    Console.WriteLine(string.Format("收到消息: {0}",
Encoding.UTF8.GetString(data)));
                    string sendMsg = "收到消息! ";
                    tcpClient.Send(Encoding.UTF8.GetBytes(sendMsg));
                }
            }).Start();
        }
```

TcpClient

```
  public static void TcpServer(IPEndPoint serverIP)
        {
            Console.WriteLine("客户端Tcp连接模式");
            Socket tcpClient = new Socket(AddressFamily.InterNetwork, SocketType.Stream,
ProtocolType.Tcp);
            try
            {
                tcpClient.Connect(serverIP);
            }
            catch (SocketException e)
            {
                Console.WriteLine(string.Format("连接出错: {0}", e.Message));
                Console.WriteLine("点击任何键退出! ");
                Console.ReadKey();
                return;
```

```
            }
            Console.WriteLine("客户端: client-->server");
            string message = "我上线了...";
            tcpClient.Send(Encoding.UTF8.GetBytes(message));
            Console.WriteLine(string.Format("发送消息: {0}", message));
            new Thread(() =>
            {
                while (true)
                {
                    byte[] data = new byte[1024];
                    try
                    {
                        int length = tcpClient.Receive(data);
                    }
                    catch (Exception ex)
                    {
                        Console.WriteLine(string.Format("出现异常: {0}", ex.Message));
                        break;
                    }
                    Console.WriteLine(string.Format("{0} 收到消息: {1}",
DateTime.Now.ToString("yyyy-MM-dd HH:mm:ss"), Encoding.UTF8.GetString(data)));
                }
            }).Start();
            Console.WriteLine("\n\n输入\"Q\"键退出。");
            ConsoleKey key;
            do
            {
                key = Console.ReadKey(true).Key;
            } while (key != ConsoleKey.Q);
            tcpClient.Close();
        }
```

Socket(套接字)编程 (Udp)



基于Udp协议是无连接模式通讯，占用资源少，响应速度快，延时低。至于可靠性，可通过应用层的控制来满足。(不可靠连接)

(1).建立一个套接字(Socket)

(2).绑定服务器端IP地址及端口号--服务器端

(3).通过SendTo()方法向指定主机发送消息

(需提供主机IP地址及端口)

(4).通过ReciveFrom()方法接收指定主机发送的消息

(需提供主机IP地址及端口)

UdpServer

```
  public static void UdpServer(IPEndPoint serverIP)
        {
            Console.WriteLine("客户端Udp模式");
            Socket udpServer = new Socket(AddressFamily.InterNetwork, SocketType.Dgram,
ProtocolType.Udp);
            udpServer.Bind(serverIP);
            IPEndPoint ipep = new IPEndPoint(IPAddress.Any, 0);
            EndPoint Remote = (EndPoint)ipep;
            new Thread(() =>
            {
                while (true)
                {
```

```
                    byte[] data = new byte[1024];
                    try
                    {
                        int length = udpServer.ReceiveFrom(data, ref Remote);//接受来自服务器
的数据
                    }
                    catch (Exception ex)
                    {
                        Console.WriteLine(string.Format("出现异常: {0}", ex.Message));
                        break;
                    }
                    Console.WriteLine(string.Format("{0} 收到消息: {1}",
DateTime.Now.ToString("yyyy-MM-dd HH:mm:ss"), Encoding.UTF8.GetString(data)));
                    string sendMsg = "收到消息! ";
                    udpServer.SendTo(Encoding.UTF8.GetBytes(sendMsg), SocketFlags.None,
Remote);
                }
            }).Start();
            Console.WriteLine("\n\n输入\"Q\"键退出。");
            ConsoleKey key;
            do
            {
                key = Console.ReadKey(true).Key;
            } while (key != ConsoleKey.Q);
            udpServer.Close();
        }
```

UdpClient

```
public static void UdpClient(IPEndPoint serverIP)
        {
            Console.WriteLine("客户端Udp模式");
            Socket udpClient = new Socket(AddressFamily.InterNetwork, SocketType.Dgram,
ProtocolType.Udp);
            string message = "我上线了...";
            udpClient.SendTo(Encoding.UTF8.GetBytes(message), SocketFlags.None, serverIP);
            Console.WriteLine(string.Format("发送消息: {0}", message));
            IPEndPoint sender = new IPEndPoint(IPAddress.Any, 0);
            EndPoint Remote = (EndPoint)sender;
            new Thread(() =>
            {
```

```
                while (true)
                {
                    byte[] data = new byte[1024];
                    try
                    {
                        int length = udpClient.ReceiveFrom(data, ref Remote);//接受来自服务器
的数据
                    }
                    catch (Exception ex)
                    {
                        Console.WriteLine(string.Format("出现异常: {0}", ex.Message));
                        break;
                    }
                    Console.WriteLine(string.Format("{0} 收到消息: {1}",
DateTime.Now.ToString("yyyy-MM-dd HH:mm:ss"), Encoding.UTF8.GetString(data)));
                }
            }).Start();
            Console.WriteLine("\n\n输入\"Q\"键退出。");
            ConsoleKey key;
            do
            {
                key = Console.ReadKey(true).Key;
            } while (key != ConsoleKey.Q);
            udpClient.Close();
        }
```

TCP和UDP的区别

TCP协议和UDP协议连接过程的区别

1.基于连接与无连接;

2.对系统资源的要求（TCP较多，UDP少）;

3.UDP程序结构较简单;

4.流模式与数据报模式 ;

5.TCP保证数据正确性，UDP可能丢包，TCP保证数据顺序，UDP不保证。

socket - TcpClient,TcpListener,UdpClient

应用程序可以通过 TCPClient、TCPListener 和 UDPClient 类使用传输控制协议 (TCP) 和用户数据文报协议 (UDP) 服务。这些协议类建立在 System.Net.Sockets.Socket 类的基础之上，负责数据传送的细节。(也就是说TCPClient、TCPListener 和 UDPClient 类是用来简化Socket)

TcpClient 和 TcpListener 使用 NetworkStream 类表示网络。使用 GetStream 方法返回网络流，然后调用该流的 Read 和 Write 方法。NetworkStream 不拥有协议类的基础套接字，因此关闭它并不影响套接字。

UdpClient 类使用字节数组保存 UDP 数据文报。使用 Send 方法向网络发送数据，使用 Receive 方法接收传入的数据文报。

TcpListener

TcpListener 类提供一些简单方法，用于在阻止同步模式下侦听和接受传入连接请求。可使用 TcpClient 或 Socket 来连接 TcpListener。可使用 IPEndPoint、本地 IP 地址及端口号或者仅使用端口号，来创建 TcpListener。可以将本地 IP 地址指定为Any，将本地端口号指定为 0（如果希望基础服务提供程序为您分配这些值）。如果您选择这样做，可在连接套接字后使用LocalEndpoint 属性来标识已指定的信息。

Start 方法用来开始侦听传入的连接请求。Start 将对传入连接进行排队，直至您调用 Stop 方法或它已经完成 MaxConnections 排队为止。可使用 AcceptSocket 或 AcceptTcpClient 从传入连接请求队列提取连接。这两种方法将阻止。如果要避免阻止，可首先使用 Pending 方法来确定队列中是否有可用的连接请求。

调用 Stop 方法来关闭 TcpListener。

TcpClient

TcpClient 类提供了一些简单的方法，用于在同步阻止模式下通过网络来连接、发送和接收流数据。为使 TcpClient 连接并交换数据，使用 TCP ProtocolType 创建的 TcpListener 或 Socket 必须侦听是否有传入的连接请求。可以使用下面两种方法之一连接到该侦听器：

(1) 创建一个 TcpClient，并调用三个可用的 Connect 方法之一。

（2）使用远程主机的主机名和端口号创建 TcpClient。此构造函数将自动尝试一个连接。

给继承者的说明要发送和接收数据，请使用 GetStream 方法来获取一个 NetworkStream。调用 NetworkStream 的 Write 和Read 方法与远程主机之间发送和接收数据。使用 Close 方法释放与 TcpClient 关联的所有资源。

UdpClient

UdpClient 类提供了一些简单的方法，用于在阻止同步模式下发送和接收无连接 UDP 数据报。因为 UDP 是无连接传输协议，所以不需要在发送和接收数据前建立远程主机连接。但您可以选择使用下面两种方法之一来建立默认远程主机：

使用远程主机名和端口号作为参数创建 UdpClient 类的实例。

创建 UdpClient 类的实例，然后调用 Connect 方法。

可以使用在 UdpClient 中提供的任何一种发送方法将数据发送到远程设备。使用 Receive 方法可以从远程主机接收数据。

UdpClient 方法还允许发送和接收多路广播数据报。使用 JoinMulticastGroup 方法可以将 UdpClient 预订给多路广播组。使用 DropMulticastGroup 方法可以从多路广播组中取消对 UdpClient 的预订。

接受最新教程和免费课程下载

XML操作

数据存储和传输？

一所大学3万个学生，记录学生信息，年龄，姓名，学号，年级，各科成绩。

十几种敌人的信息：类型、攻击力大小、移动速度、颜色

1、直接把数据写到记事本里面（数据多了之后，不方便管理）

2、Excel - 方便进行数据管理 - 给人用的

文件比较大，程序读取比较慢

3、XML格式和Json格式的文本

数据方便管理

程序读取快

XML

XML 指可扩展标记语言

XML 被设计用来传输和存储数据。XML 被设计用来结构化、存储以及传输信息。

xml文档展示

----------xml文档

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<note>
<to>George</to>
<from>John</from>
<heading>Reminder</heading>
<body>Don't forget the meeting!</body>
</note>
```

这个 XML 文档仍然没有做任何事情。它仅仅是包装在 XML 标签中的纯粹的信息。我们需要编写软件或者程序，才能传送、接收和显示出这个文档。

xml标签

第一行是 XML 声明。它定义 XML 的版本 (1.0) 和所使用的编码 (ISO-8859-1 = Latin-1/西欧字符集)。

下一行描述文档的根元素（像在说：“本文档是一个便签”）：<note>

接下来 4 行描述根的 4 个子元素（to, from, heading 以及 body）：

```
<to>George</to>
<from>John</from>
<heading>Reminder</heading>
<body>Don't forget the meeting!</body>
```

最后一行定义根元素的结尾：

```
</note>
```

XML 文档形成一种树结构

XML 文档必须包含根元素。该元素是所有其他元素的父元素。

XML 文档中的元素形成了一棵文档树。这棵树从根部开始，并扩展到树的最底端。

所有元素均可拥有子元素：

```
<root>
  <child>
    <subchild>.....</subchild>
  </child>
  <child>
    <subchild>.....</subchild>
  </child>
</root>
```

XML 元素

XML 元素指的是从（且包括）开始标签直到（且包括）结束标签的部分。

```
<bookstore>
<book category="CHILDREN">
  <title>Harry Potter</title>
  <author>J K. Rowling</author>
  <year>2005</year>
  <price>29.99</price>
</book>
<book category="WEB">
  <title>Learning XML</title>
  <author>Erik T. Ray</author>
  <year>2003</year>
  <price>39.95</price>
```

</book>
</bookstore>
<bookstore> 和 <book> 都拥有元素内容，因为它们包含了其他元素。<author> 只有文本内容，因为它仅包含文本。
只有 <book> 元素拥有属性 (category="CHILDREN")。

xml语法规则

所有 XML 元素都须有关闭标签

```
<p>This is a paragraph</p>
```

XML 标签对大小写敏感,标签 <Letter> 与标签 <letter>

```
<Message>这是错误的。</message>
<message>这是正确的。</message>
```

XML 必须正确地嵌套

```
<b><i>This text is bold and italic</b></i>
<b><i>This text is bold and italic</i></b>
```

XML 文档必须有根元素

```
<root>
  <child>
    <subchild>.....</subchild>
  </child>
</root>
```

XML 的属性值须加引号

```
<note date=08/08/2008>
<to>George</to>
<from>John</from>
</note>
<note date="08/08/2008">
<to>George</to>
<from>John</from>
</note>
```

XML 中的注释

```
<!-- This is a comment -->
```

XML 命名规则

XML 元素必须遵循以下命名规则:

名称可以含字母、数字以及其他的字符

名称不能以数字或者标点符号开始

名称不能以字符 “xml”（或者 XML、Xml）开始

名称不能包含空格

可使用任何名称，没有保留的字词。

实例

```
<skills>
        <skill>
                <id>2</id>
                <name lang="cn">天下无双</name>
                <damage>123</damage>
        </skill>
        <skill>
                <id>3</id>
                <name lang="cn">永恒零度</name>
                <damage>93</damage>
        </skill>
        <skill>
                <id>4</id>
                <name lang="cn">咫尺天涯</name>
                <damage>400</damage>
        </skill>
</skills>
```

C#操作XML

在C#中使用控制台程序，用 XMLDocument进行xml操作，包括查询，增加，修改，删除和保存。

```
<skills>
        <skill>
                <id>2</id>
                <name lang="cn">天下无双</name>
                <damage>123</damage>
        </skill>
```

```
        <skill>
                <id>3</id>
                <name lang="cn">永恒零度</name>
                <damage>93</damage>
        </skill>
        <skill>
                <id>4</id>
                <name lang="cn">咫尺天涯</name>
                <damage>400</damage>
        </skill>
</skills>
```

Json操作

JSON

JSON 是存储和交换文本信息的语法。类似 XML。

JSON 比 XML 更小、更快，更易解析。JSON跟XML一样是一种是数据格式。

JSON(JavaScript Object Notation) 是一种轻量级的数据交换格式。它基于ECMAScript的一个子集。 JSON采用完全独立于语言的文本格式，但是也使用了类似于C语言家族的习惯（包括C、C++、C#、Java、JavaScript、Perl、Python等）。这些特性使JSON成为理想的数据交换语言。 易于人阅读和编写，同时也易于机器解析和生成(网络传输速率)。

```
{
    "employees": [
        { "firstName":"Bill" , "lastName":"Gates" },
        { "firstName":"George" , "lastName":"Bush" },
        { "firstName":"Thomas" , "lastName":"Carter" }
    ]
}
```

什么是JSON

JSON 是轻量级的文本数据交换格式

JSON 独立于语言 *

JSON 具有自我描述性，更易理解

* JSON 使用 JavaScript 语法来描述数据对象，但是 JSON 仍然独立于语言和平台。JSON 解析器和 JSON 库支持许多不同的编程语言。

JSON 语法规则

数据在键值对中

数据由逗号分隔

花括号保存对象

方括号保存数组

JSON 名称/值对

JSON 数据的书写格式是：名称/值对。

名称/值对组合中的名称写在前面（在双引号中），值对写在后面(同样在双引号中)，中间用冒号隔开："firstName":"John"

JSON 值可以是:

数字（整数或浮点数）

字符串（在双引号中）

逻辑值（true 或 false）

数组（在方括号中）

对象（在花括号中）

null

JSON数据结构

json简单说就是javascript中的对象和数组，所以这两种结构就是对象和数组两种结构，通过这两种结构可以表示各种复杂的结构

1、对象：对象在js中表示为“{}”括起来的内容，数据结构为 {key：value,key：value,...}的键值对的结构，在面向对象的语言中，key为对象的属性，value为对应的属性值，所以很容易理解，取值方法为 对象.key （c# 对象[key]）获取属性值，这个属性值的类型可以是 数字、字符串、数组、对象几种。

2、数组：数组在js中是中括号“[]”括起来的内容，数据结构为["java","javascript","vb",...]，取值方式和所有语言中一样，使用索引获取，字段值的类型可以是 数字、字符串、数组、对象几种。

经过对象、数组2种结构就可以组合成复杂的数据结构了。

Json官网

Json.org 去官网看下学习教程

找到litjson的官网

bejson.com

使用什么工具解析?

1、Newtonsoft.Json

2、fastjson

LitJson停止更新

右键项目-》管理NuGet程序包

Excel操作

1,使用OLEDB操作Excel

关于OLEDB介绍参考

http://www.cnblogs.com/moss_tan_jun/archive/2012/07/28/2612889.html

2,连接字符串

```
"Provider=Microsoft.Jet.OLEDB.4.0;" + "Data Source=" + fileName + ";" + ";Extended Properties=\"Excel 8.0;HDR=YES;IMEX=1\"";
"Provider=Microsoft.ACE.OLEDB.12.0;" + "Data Source=" + fileName + ";" + ";Extended Properties=\"Excel 12.0;HDR=YES;IMEX=1\"";
```

Excel操作

```
//加载Excel
public static DataSet LoadDataFromExcel(string filePath)
{
    try
    {
        string strConn;
        strConn = "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=" + filePath + ";Extended Properties='Excel 8.0;HDR=False;IMEX=1'";
        OleDbConnection OleConn = new OleDbConnection(strConn);
        OleConn.Open();
```

```
                String sql = "SELECT * FROM  [Sheet1$]";//可是更改Sheet名称, 比
如sheet2, 等等

                OleDbDataAdapter OleDaExcel = new OleDbDataAdapter(sql,
OleConn);
                DataSet OleDsExcle = new DataSet();
                OleDaExcel.Fill(OleDsExcle, "Sheet1");
                OleConn.Close();
                return OleDsExcle;
            }
            catch (Exception err)
            {
                MessageBox.Show("数据绑定Excel失败!失败原因: " + err.Message,
"提示信息",
                    MessageBoxButtons.OK, MessageBoxIcon.Information);
                return null;
            }
        }
```

接受最新课程通知和免费课程下载链接

SiKi学院出品 - www.sikiedu.com SiKi老师