

C# - P27

SHINEHUA

189 - printed

运算符重载

在C#中，用于实现静态多态性的技术，另一种是函数重载 (overloading)

可以重定义/重载C#中内置的运算符，也可以使用用户自定义类型的运算符
重载运算符是具有特殊名称的函数，~~通过operator关键字与运算符的符号来定义的~~

通过operator关键字与运算符的符号来定义的

与一般函数一样，重载运算符具有返回类型和参数列表

注意运算符只能采用值参数，不能使用ref/out

一元运算符，可以被重载：+，-，!，~，++，--，用整数返回

二元运算符，可以被重载：+=，-=，*=，/=，%

比较运算符，可以被重载：==，!=，<，>，<=，>=

条件运算符，不可以被直接重载：&&，||

赋值运算符，不可以被重载：+=，-=，*=，/=，%+=，%-=

不可被重载的运算符：.，?，:，->，new，is，sizeof，typeof

该关键字operator用于在类或结构中声明运算符，基本形式为

public static <return type> operator <unary op> (<op type> <operand>)

public static <return type> operator <binary op> (<fst op type>, <fst operand>, <snd op type>, <snd operand>)

public static implicit operator <conv type out> (<conv type in> <operand>)

public static explicit operator <conv type out> (<conv type in> <operand>)

return type：运算符的返回值类型

unary op：重载的一元运算符

binary op：重载的二元运算符

op type / fst op type / snd op type：运算数的类型

fst 与 snd 中至少有一个必须是封闭类型

即运算符所属类型，或理解为自定义类型

后两种声明了转换运算符，与前三种不同

[其中] conv.out / conv.in 中有且仅有一个必须是封闭类型

即或者封闭类型转换为其他类型，或者由其他类型转换为封闭类型

C#要求比较运算符必须成对重载，如 == != 仅重载之一，则抛出编译错误

且比较运算符必须返回 Boolean 类型

C#不允许重载赋值运算符，但当 == 元运算符重载时

如重载 +，则编译器自动实现重载版本的 + 元运算符

运算符重载的实质是函数重载，中间过程由编译器完成

接口 (interface) 在 C# 中，接口定义派生类在继承接口时应遵循的语法合同

其中接口定义语义合同“是什么”，派生类定义语义合同“怎么做”

接口定义了属性/方法/事件，接口提供派生类应遵循的标准结构

接口只包含成员的声明，而成员的定义是派生类的责任

使得实现接口的类/结构在形式上保持一致

接口与抽象类 (abstract class) 类似

但抽象类大多用于仅有少数方法由基类声明而由派生类实现

基本的接口定义与使用

```
public interface myInterface {
    void someMethod();
}
```

声明了接口中的方法，不允许定义函数体

public class myClass : myInterface { }

实现接口的方法的具体实现

```
public void someMethod() {
    Console.WriteLine("call someMethod");
}
```

方法的实现不能直接调用到父类

}

接口也可以被其他接口所继承，实现类或结构需要实现所有接口的成员

接口内可以定义属性 T，如：string str = get; set;

但是不可以有字段变量 / 构造函数

派生类实现接口的成员时，必须与接口的格式保持一致

可以通过接口实现多重继承，即解决 C# 中同时继承多个基类

且 C# 中的接口不能指定为 public/protected/internal/private

由于接口的方法均需要由派生类实现方法体，则默认认为 public 的

接口成员不能指定为 new/static/abstract/override/virtual

但是当接口继承另一个接口时，可以使用 new 关键字隐藏父接口的方法

接口只包含成员的签名，且不包含构造函数

只能包含方法/属性/事件/委托的组合

不能直接使用 new 关键字对接口进行实例化

派生类必须实现所继承接口中的所有成员

除派生类本身即是抽象类

C# - P29

接口与抽象类 在C#中，比较接口(interface)与抽象类(abstract class)的区别

接口用于规范，抽象类用于共性

抽象类是类，只能被单继承，但接口可以用于实现多继承

抽象类可能包含部分方法的实现，但接口只能包含方法的声明

接口的实例由实现接口的类给出，抽象类的实例由其派生类给出

向接口中加入新方法时，所有实现该接口的类都必须提供该方法的实现

而向抽象类中添加新方法时，所有派生类即获得了该方法的默认实现

接口中的成员不可使用 access specifier，默认为 public 的

而抽象类中的成员可以指定为 public / protected / internal / private

接口不能包含字段，构造函数，析构函数，静态成员，常量

命名空间 (namespace)，在C#中，用于使一组名称(identifier)与其他名称相隔离开的方式

不同命名空间的相同名字不会产生冲突

C#中通过 namespace 关键字定义命名空间

基本语法结构为 namespace <namespace name> {

 [<definition>]*

[<definition>] = { } 而且 只能写一个类

例如 namespace fst_space {

 public class myClass {

 public void someMethod() {

 Console.WriteLine("call someMethod from fst space");

 }

namespace snd_space {

 public class myClass {

 public void someMethod() {

 Console.WriteLine("call someMethod from snd space");

 }

C#中通过点运算符(.) 使用命名空间中定义的变量/函数/类/结构/...

基本语法结构为：<namespace name>. <item name>

例如 (new fst_space.myClass()).someMethod(); → "call someMethod from fst

 (new! snd_space.myClass()).someMethod(); → "call someMethod from

"snd space"

C# - P30

Shallow

89 - 深度

命名空间 在C#中，通过使用 using 关键字表示程序使用给定命名空间中的名称

如对于 System.Console.WriteLine("Hello World");

等价于 using System;

Console.WriteLine("Hello World"); → "Hello World"

C# 还提供了 using static 声明，基本语法为

using static <fully-qualified-type-name>;

指定一种类型，无需指定类型名即可访问其静态成员和嵌套类型

适用于任何具有静态成员（或嵌套类型）的类型，即使该类型还具有实例成员

但是只能通过类型实例来调用实例成员

如 using static System.Math;

using static System.Console;

WriteLine(PI); → 3.14159265358979

using 关键字还可以用于指定别名（alias），基本语法为

using <alias> = <namespace / typename>

如 using fst = fst_space;

(new fst.myClass()).someMethod(); → "call someMethod from fst space"

using 关键字还可以用于将类型实例与代码绑定，基本语法为

using (<instance define>) {<statements>}

特别注意：using 使用的类型必须实现了 IDisposable 接口

type used in using statement

must be implicitly convertible to 'System.IDisposable'

or implement suitable 'Dispose' method

如 public class myClass : IDisposable {

void IDisposable.Dispose()

Console.WriteLine("call IDisposable.Dispose"); }

using (myClass mc = new myClass()) {

Console.WriteLine("using myClass"); → "using myClass"

}

→ "call IDisposable.Dispose"

Console.WriteLine("end using");

→ "end using"

命名空间

在 C# 中，命名空间可以嵌套使用，基本语法结构为：

```
namespace <namespace name> { ... }
```

```
    <definitions>
```

```
        namespace <subnamespace name> { ... }
```

```
            <definitions>
```

```
                ...
```

使用点运算符访问嵌套的命名空间

```
using main_namespace.sub_namespace;
```

所有预处理器指令均以“#”开始，且在同一行上预处理器指令前只能有空白字符出现

所有预处理器指令不是语句，所以不以“;”结束

C# 的编译器并没有单独的预处理器

但指令被处理时如同存在一个单独的预处理器

与 C/C++ 不同，C# 中的预处理器指令不用于创建宏

一个预处理器指令必须是该行上的唯一指令

预处理器指令可用于禁止编译器编译部分代码，与普通的控制结构类似

也可以用于编写和控制用于提供调试信息的代码

方便在于预处理器指令包含的未执行部分是不需要编译的

#define <symbol> : 用于定义符号常量，传入符号作为 # 计的表达式将返回 true

undef <symbol> : 用于取消符号常量定义

注意 #define/#undef 必须在文件的 first token 之前

编译代码

#if / #elif / #else : 用于创建复合条件指令，用于在调试版本或编译指定配置时

#endif : 用于指定条件指令结束，以 #if 开始的条件指令必须显式地以 #endif 终止

基本语法为：#if / #elif (<symbol> [operator] <symbol>*)

symbol 是用于测试的符号常量，也可以使用 true / false

operator : EQ(==), NE(!=), AND(&&), OR(||), NOT(!)

#define PI 编译器会编译并执行 #if / #elif / #else 到下一个预处理器指令之间的代码

如 #if (PI)

Console.WriteLine("PI is defined"); → "PI is defined"

#else

Console.WriteLine("PI is not defined");] 未编译，可以无视语法

#endif

C# - P32

预处理器指令 `#warning <message>`: 允许从代码指定位置生成一个警告

`#error <message>`: 允许从代码指定位置生成一个错误

对于 `#warning`, 在显示警告文本后会继续编译

通常用于提醒代码中可以进行的修正由

对于 `#error`, 在显示错误信息后立即退出编译

通常用于指出编译过程遇到的错误

如 `#warning "DEBUG and RELEASE can not be defined simultaneously"`

`#if DEBUG && RELEASE`

`#error "DEBUG and RELEASE defined simultaneously"`

`#endif`

注意 `#warning` 和 `#error` 的显示输出是通过编译器

与程序运行时通过 `Console.WriteLine()` 不同

`#region / #endregion`: 指令用于将一段代码标记为特定名称的一个代码块

基本语法结构为: `#region <region name>`

`<statements>`

`#endregion` = 结束 [该] 代码块

`#region` 标记的代码块不影响编译过程

但是可以被部分编辑器识别, 如 Visual Studio .NET 编辑器

在使用 Visual Studio Code Editor 的大纲特性时

指定一个可展开/折叠的代码块

可以使代码在屏幕上更好地布局

`#line`: 允许修改编译器的行数以及(可选地)输出错误和警告的文件名

可以使用 `#line` 在编写代码时, 将代码发送给编译器前, 使用软件包改变输入代码

意味着编译器报告的行号/文件名与文件中的行号/编辑的文件名不匹配

`#line default` 用于将行号还原为默认行号

如 `#line 164 "Core.cs"`

`// Core.cs, before the intermediate`

`// package mangles it to N = n0 例外且为`

`// later on`

`#line default`

`// now back to 164`

预处理器指令 `#pragma`：用于抑制/还原指定的编译警告
 与命令行选项不同，`#pragma` 可以出现在类/方法的级别
 则对警告的内容和抑制的时间进行更精细的控制

如抑制警告编号 169：字段未使用警告

`#pragma warning disable 169`

```
class TestPragma {
    int neverUsedInt;
}

#pragma warning restore 169
```

正则表达式 (regular expression)，在 C# 中正则表达式是一种匹配已输入文本的模式

.NET Framework 提供了支持正则表达式匹配的引擎

正则表达式的模式 (pattern) 由一个或多个字符/运算符/结构组成

字符转义：正则表达式中的反斜杠字符 (\) 指示其后的字符为特殊字符
 或应按原义解释该字符

| 转义字符 | 描述 | 模式匹配 |
|------------|---|-------------------------|
| \a | 与报警符 (bell) \u0007 匹配 | a : "\u0007" |
| \b | 与退格 \u0008 匹配 | b : "\u0008" |
| \t | 与制表符 \u0009 匹配 | t : "\u0009" |
| \r | 与回车符 \u000D 匹配 | r : "\u000D" |
| \v | 与垂直制表符 \u000B 匹配 | v : "\u000B" |
| \f | 与换页符 \u000C 匹配 | f : "\u000C" |
| \n | 与换行符 \u000A 匹配 | n : "\u000A" |
| \e | 与转义符 \u001B 匹配 | e : "\u001B" |
| \000 | 使用八进制表示形式指定一个字符 000 为三位八进制数字 | lwl040lw : "a b" |
| \x nn | 使用十六进制表示形式指定字符 nn 为二位十六进制数字 | lwl\x20lw : "a b" |
| \c X, \c x | 匹配已指定的 ASCII 控件字符 X, x 为控件字符的字母 | lcC : "\x0003" (Ctrl-C) |
| \u nnnn | 使用十六进制表示形式匹配 Unicode 字符 nnnn 为四位十六进制数字 | lwl\u0020lw : "a b" |
| \ | 在后面带有不识别的转义字符时，与字符匹配 | l : "+" |

正则表达式

字符类与一组字符串中的任何一个字符匹配

字符类描述状态、instant state denoted by 模式匹配

[char-group] 匹配已 char_group 中的任何单个字符

[mn] → "moon"

[^char-group] 匹配已任何不在 char_group 中的单个字符

[^mn] → "moon"

字符串

[first - last] 从 first 到 last 范围中的任何单个字符

[b-d] → "be", "ce", "de"

通配符

. 匹配已除 \n 外的任何单个字符

(.) 用于匹配已任意单个字符

\p{name} 与 name 指定的 Unicode 通用类别或命名块

\p{Lu} → "City"

中的任何单个字符匹配

\P{name} 不在 name 指定的 Unicode 通用类别或命名块

\P{Lu} → "City"

的任何单个字符匹配

\w 与任何单词字符匹配

\w → "Room #!"

\W 与任何非单词字符匹配

\W → "Room #!"

\s 与任何空白字符匹配

\s → "a_e"

\S 与任何非空白字符匹配

\S → "a_e"

\d 与任何十进制数字匹配

\d → "Room #!"

\D 与任何非十进制数字匹配

\D → "Room #!"

定位点：定位点/原子零宽度断言会使匹配成功或失败，取决于字符串中的当前位置

但不会使正则表达式引擎在字符串中前进/使用字符串

断言 | 描述 | 模式匹配

^ 匹配必须从字符串/行的开头开始

^eva) → "eva eva eva"

注意与 [^char-group] 中的意义不同

\$ 匹配必须出现在字符串末尾

\$eva) → "eva eva eva"

或行/字符串末尾的 \n 之前

\A 匹配必须出现在字符串的开头

\A → "eva eva eva"

注意与 [^char-group] 中的意义不同

\Z 匹配必须出现在字符串末尾或末尾的 \n 之前

\Z → "eva eva eva"

注意与 [^char-group] 中的意义不同

\G 匹配必须出现在上一个匹配结束的地方

\G\id : "135a9"

注意与 [^char-group] 中的意义不同

\b 匹配单词边界，即单词与空格间的位置

\verb\b : "never"

注意与 [^char-group] 中的意义不同

\B 匹配非单词边界

\verb\B : "verb"

注意与 [^char-group] 中的意义不同

C# - P35

| 正则表达式 | 分组构造 | 描述正则表达式的子表达式，通常用于捕获输出字符串的子字符串 | 模式匹配 |
|--|---------------------------|---|----------------------------------|
| 分组构造 (subexpression) | 描述被捕获匹配子表达式 | 模式匹配 <code>(\w)\w : "deep"</code> | |
| | 并分配到一个0-based的符号中 | | |
| <code>(?<name> subexpression)</code> | 将匹配子表达式捕获到命名组中 | <code>(?<double>\w)\w<double>:</code> | |
| | | "deep" | |
| <code>(?<name1-name2> subexpression)</code> | 定义平衡了组定义 | | Write (?<Line>)? : |
| <code>(?: subexpression)</code> | 定义非捕获组 | | "Console.WriteLine()" |
| | | | |
| <code>(?imnsx-imnsx: subexpression)</code> | 应用/禁用 subexpression 中指定选项 | <code>A\d{2}(?i:\w)\b :</code> | |
| | | "A12\tl A12\tL a12\tl" | |
| <code>(\{\!\!\{proto\}\!\!\} : tgr, \!\!b\!\!t\!\!s\!\!e\!\!d\!\!u\!\!.sq\!\!p\!\!t\!\!.com\!\!s\!\!o\!\!l\!\!e\!\!: sq\!\!p\!\!t\!\!)</code> | 零宽度正预测先行断言 | <code>\w+(?=.) :</code> | |
| <code>(?= subexpression)</code> | 零宽度正预测先行断言 | "The dog ran. The sun is out." | |
| | | | |
| <code>(\!\!\!r\!\!o\!\!n\!\!t\!\!s\!\!f\!\! : tgr, \!\!b\!\!t\!\!s\!\!e\!\!d\!\!u\!\!.sq\!\!p\!\!t\!\!.com\!\!s\!\!o\!\!l\!\!e\!\!: sq\!\!p\!\!t\!\!)</code> | 零宽度负预测先行断言 | <code>\b(?!un)\w+\b</code> | |
| <code>(?! subexpression)</code> | 零宽度负预测先行断言 | | "unsure sure used" |
| | | | |
| <code>(?<= subexpression) wA</code> | 零宽度正回顾后发断言 | <code>(?<=19)\d{2}\b :</code> | |
| | | "1851 1949 1905 2017" | |
| <code>(?<! subexpression) wuab</code> | 零宽度负回顾后发断言 | <code>(?<!wo)man\b :</code> | |
| <code>(?> subexpression)</code> | 非回溯("贪婪")子表达式 | | "Hi woman Hi <u>man</u> " |
| | | | |
| | | | <code>[13579](?>A+B+):</code> |
| | | | "IABB 3ABBC 5AC" |
| | | | |

备用构造：用于修改正则表达式以启动 either/or 匹配

| 备用构造 | 描述 | 模式匹配 |
|---|--|---|
| | 匹配以 分隔的任何一个元素 | <code>this is at : "this is <u>the day</u>"</code> |
| <code>(?(<expression>) yes no)</code> | 如果正则表达式模式由 expression 匹配决定，则匹配 yes，否则匹配可选的 no expression 解释为零宽度断言 | <code>(?(A)\d{2})\b1\b1\d{3}\b1)</code> "A10 C103 910" |
| <code>(?<name> yes no)</code> | 如果 name/已命名/已编号的捕获组具有匹配，则匹配 yes 部分。 | <code>(?<quoted>"")? (?(quoted).+?" S+S)</code> "Dogs.jpg "Playing.jpg"" |
| | | |
| | | |

正则表达式中的限定符：指定在输入字符串中必须存在上一个元素的实例数量才完成匹配

元素可以是字符/组/字符串

① 量限定符 | 描述 | 模式匹配

| | | |
|---------------------|---------------------|---|
| <code>\d*</code> | 匹配上一个元素零次或多次 | <code>\d*\.\d</code> : ".01", "11.01" |
| <code>+</code> | 匹配上一个元素一次或多次 | <code>bet</code> : "beat", "been" |
| <code>{n}</code> | 匹配上一个元素恰好n次 | <code>rai?n</code> : "rain", "bran" |
| <code>{n,}</code> | 匹配上一个元素至少n次 | <code>\d{3}</code> : "1024", "10,001" |
| <code>{n,m}</code> | 匹配上一个元素至少n次,至多m次 | <code>\d{2,3}</code> : "29", "121" |
| <code>*?</code> | 匹配零次或多次,但匹配次数尽可能少 | <code>\d{3,5}</code> : "1234", "327678" |
| <code>+?</code> | 匹配一次或多次,但次数尽可能少 | |
| <code>??</code> | 匹配零次或一次,但次数尽可能少 | |
| <code>{n}?</code> | 匹配恰好n次 | |
| <code>{n,}?</code> | 匹配至少n次,但次数尽可能少 | |
| <code>{n,m}?</code> | 匹配至少n次,至多m次,但次数尽可能少 | |

反向引用构造：允许在同一正则表达式中随时标识以前匹配的子表达式

反向引用构造 | 描述 | 模式匹配

| | |
|---|--|
| <code>\\$</code> <code>(start, min)</code> 匹配编号子表达式的值 | <code>(\w){1}</code> <code>\\$seek</code> |
| <code>\k<name></code> 匹配命名表达式的值 | <code>(?<char>\w)\k<char></code> <code>"seek"</code> |

替换：在替换模式中使用的正则表达式

替换 | 描述

模式匹配

替换模式

| | | | | |
|------------------------|-------------------|--|---------------------------|---------------------------------------|
| <code>\\$number</code> | 替换按组number匹配的子字符串 | <code>\b(\w+)(\s)(\w+)\b</code> | <code>\\$3\\$2\\$1</code> | <code>"one two" → "two one"</code> |
| <code>\\$name</code> | 替换按命名组name匹配的子字符串 | <code>\b(?<word1>\w+)(\s)</code> | <code>\\$\{word1\}</code> | <code>"one two"</code> |
| <code>\\$</code> | 替换子字符串 | <code>(?<word2>\w+)\b</code> | <code>\\$\{word2\}</code> | <code>→ "two one"</code> |
| <code>\\$`</code> | 替换匹配前输入字符串的所有文本 | <code>\b(\d+)\\$?</code> | <code>\\$\\$\\$1</code> | <code>"103 USD" → "\\$103"</code> |
| <code>\\$`</code> | 替换匹配后输入字符串的所有文本 | <code>B+\\$</code> | <code>\\$\`</code> | <code>"AABBCC" → "AAAACC"</code> |
| <code>\\$+</code> | 替换最后捕获的组 | <code>B+(C+)2</code> | <code>\\$\+</code> | <code>"AABBCC DD" → "AAACC DD"</code> |
| <code>\\$-</code> | 替换整个输入字符串 | <code>B+</code> | <code>\\$_</code> | <code>"AABBCC" → "AAAABBCCCC"</code> |

| 正则表达式 | 杂项构造 | 描述 | 模式匹配 |
|----------------|-------------------|--|------|
| (?imnsx-imnsx) | 在模式中对选项进行设置/禁用 | $\backslash b A (?i) b \backslash w + \backslash b \rightarrow \underline{ABA} \underline{Able} \underline{Act}$ | |
| (?#注释) | 内联注释，在第一个反斜杠处终止 | $\backslash b A (?# \text{单词以 } A \text{ 开头}) \backslash w + \backslash b$ | |
| # [行尾] | 注释以非转义的#开头，并持续到行尾 | $(?x) \backslash b A \backslash w + \backslash b \# \text{单词以 } A \text{ 开头}$ | |

(#在 System.Text.RegularExpressions 中提供了 Regex 类用于支持正则表达式

bool Regex.IsMatch(string input, string pattern)

bool Regex.IsMatch(string input, string pattern,

(RegexOptions options, TimeSpan matchTimeout)

indicate whether specified regular expression find match in specified input string
using specified matching options and time-out interval

(should return Match object if match found)

std::string Match Regex.Match(string input, string pattern)

Match Regex.Match(string input, string pattern,

(RegexOptions options, TimeSpan matchTimeout)

Search input string for first occurrence of specified regular expression

using specified matching options and time-out interval

2. FINAL API

REGEX API

MatchCollection Regex.Match(string input, string pattern)

MatchCollection Regex.Match(string input, string pattern,

(RegexOptions options, TimeSpan matchTimeout)

Search input string for all occurrences of specified regular expression

(should return MatchCollection object)

String Regex.Replace(string input, string pattern, string replacement)

String Regex.Replace(string input, string pattern, MatchEvaluator evaluator)

replace all strings in specified input string match specified regular expression

evaluator with specified replacement string / string returned by MatchEvaluator delegate

(should return string)

String Regex.Escape(string str) by replacing with escape code

escape minimal set of {\, *, +, ?, {, }, [,], (,), ^, \$, ., #, white space}

instruct regex expression engine to interpret characters literally rather than metacharacters

(should return string)

String[] Regex.Split(string input, string pattern)

split input string into array of substrings at position defined by regular expression pattern

特性(attribute)，在C#中用于在运行时传递程序中元素(类，方法，结构，枚举，组件)行为信息的声明性标签。

声明性标签通过放置在对应元素前的方括号[]描述。

可以向特性向程序添加声明性信息。

也可以用于添加元数据，如编译器指令和注释，描述，方法，类。

定义特性的基本语法结构为：

[<attribute> (<positional_parameters> [, <name_parameter> = <value>]*)]

解释：[] 表示特性，() 表示参数，, 表示逗号，= 表示赋值，* 表示可选。

名字和值都在方括号内定义，并放置在对应的element之前。

positional_parameters 规定必须提供的信息。

name_parameter 规定可选的信息。

.NET Framework 提供了三种预定义特性。

AttributeUsage 描述如何使用一个自定义特性类，规定特性可应用到的项目类型。

定义的基本语法为 [AttributeUsage(<Validation>, <Targets>, <AllowMultiple>)]

<Validation> 规定特性可被放置的语言元素。

<Targets> 规定特性可被放置的语义元素。

<AllowMultiple> 枚举器 AttributeTargets 的组合(按位或)，默认值为 AttributeTargets.All。

<Validation> 可选地为特性的 AllowMultiple 属性传入布尔值。

如果为 true 则特性是多用的，默认值是 false。

<Inherited> 表示特性是否可被派生类继承，默大认值为 false。

Conditional 标记条件方法，其执行依赖于指定的预处理标识符。

定义的基本语法为 [Conditional(<conditional_symbol>)]

当运行到条件方法调用时，根据出现该调用时是否定义标识符确定是否执行该调用。

如果已定义则执行调用，否则省略调用以及对调用参数的计算。

可以替代封闭#if 和#endif 内部方法，更整洁从而减少出错。

条件方法必须是类声明或委托声明中的方法。

如果是接口方法的声明或实现，则发生编译时错误。

条件方法必须具有返回类型。

不能用 override 修饰符标记条件方法，但可以用 virtual 修饰符。

其重写方法隐含为有条件的方法，且不能用 Conditional 显式标记。

不能用于委托创建表达式，否则发生编译时错误。

遗漏处有遗漏项，如类名、方法名等。

特性

`Obsolete` 标记不应被使用的程序实体，通知编译器丢弃特定的目标元素

通常用于当新方法定义后，仍想保留原有的旧方法

同时提供一个信息提示应使用新方法而非旧方法

`[Obsolete("old method", true)]` 定义的基本语法结构为 `new class { [Obsolete] old method; }`

`[Obsolete(<message>, <iserror>)]`

`message`: 用于编译器警告/错误的提示信息

`iserror`: Boolean value indicating whether obsolete element usage considered as error

(`Element.Obsolete("old", true)`) true if obsolete element usage generate compiler error

(`Element.Obsolete("old", false)`) false if obsolete element usage generate compiler warning

[`iserror = false`] 默认值为 `false`, 即发出编译器警告

特别注意：被标记为不应被使用的程序实体仍然存在于定义中

即不可使用与 `Obsolete` 的方法相同参数列表的方法定义

`[Obsolete("Don't use the older version", true)]`

`static void someMethod() { ... }`

`static void someMethod() { ... }`

会产生冲突，already define member "someMethod" with same parameter types

.NET Framework 允许创建自定义特性，用于存储声明性信息，且可在运行时被检索

该信息根据设计标准和应用程序需要，可与任何目标元素相关

创建并使用自定义特性包含四个步骤

声明自定义特性：自定义特性应派生自 `System.Attribute` 类

构建自定义特性：`readonly`, `public`, `private`

在目标程序元素上应用自定义特性

通过反射访问特性：包含编写简单的程序读取元数据以便查找各种符号

元数据用于描述其他数据的数据和信息，程序使用反射在运行时访问特性

`[AttributeUsage(AttributeTargets.Class)]` public class <attribute name> : Attribute {

`AttributeTargets.Constructor | AttributeTargets.Field | AttributeTargets.Method | AttributeTargets.Property | AttributeTargets.ReturnValue`

`private string name;`

`AttributeTargets.Field | AttributeTargets.Method | AttributeTargets.Property | AttributeTargets.ReturnValue`

`public string Name {`

`get { return name; }`

`AllowMultiple = true]` `(EField) (EProperty) (EMethod) (EConstructor) (EReturnValue) (EClass)`

`(EField) (EProperty) (EMethod) (EConstructor) (EReturnValue) (EClass)`

`[<attribute name>("Irony")]`

`class test { }`

`public <attribute name>(string name){`

`this.name = name; }`

}

反射

(reflection), 在 C# 中指程序可以访问、检测、修改其本身状态或行为的能力。

程序集包含模块，模块包含类型，类型包含成员

反射提供了封装程序集、模块、类型的对象

可以动态地创建类型的实例，将类型绑定到现有对象，或从现有对象中获取类型
则可以调用类型的方法或访问字段和属性

优点：提高程序的灵活性和扩展性

降低耦合性，提高自适应能力

允许程序创建和控制任何类的对象，无需提前硬编码目标类

缺点：反射基本上是一种解释操作，用于字段和方法接入时远慢于直接编码

主要应用于对灵活性和扩展性要求高的系统框架，不适用于普通程序

反射绕过了原代码的技术，模糊了程序内部逻辑

比相应的直接编码更复杂，会带来维护问题

用途：允许在运行时查看特性 (attribute) 信息

允许审查集合中的各种类型，以及实现化类型

允许延迟绑定的方法和属性 (property)

允许运行时创建新类型，再使用新类型执行任务

属性

(property)。在 C# 中是类 (class)、结构 (structure)、接口 (interface) 的命名成员 (named member)

类/结构中的成员变量/方法称为域 (field)，属性是域的扩展，可以使用相同语法访问

使用访问器 (accessor) 使私有域的值可被访问/操作

属性不会确定存储位置，相反使用可读写的访问器

访问器包含有助于获取 (读取/计算) 或设置 (写入) 属性的可执行语句

声明包含 get 访问器 / set 访问器，或同时包含两者

```
public class someClass {
```

```
    public string name {
```

```
        get {
```

```
            return name;
```

```
        }
```

```
        set {
```

```
            name = value;
```

抽象类可拥有抽象属性 (abstract property)，抽象属性应在派生类中实现

```
public abstract string name { get; set; }
```