



# 玩儿道编码规范

## 1.3

## 文档修改历史记录

[illegible]

# 目录

<b>1</b>	<b>概览.....</b>	<b>1</b>
1.1	原则和主旨 .....	1
1.2	术语 .....	2
<b>2</b>	<b>通用编程规范.....</b>	<b>3</b>
2.1	明确性和一致性 .....	3
2.2	格式和风格 .....	3
2.3	库的使用 .....	5
2.4	全局变量 .....	5
2.5	变量的声明和初始化 .....	5
2.6	函数的声明和调用 .....	6
2.7	代码语句 .....	7
2.8	枚举 .....	7
• 2.8.1	标志枚举 .....	9
2.9	空格 .....	10
• 2.9.1	空行 .....	10
• 2.9.2	空格 .....	11
2.10	大括号 .....	12
2.11	注释 .....	13
• 2.11.1	内联代码注释 .....	14
• 2.11.2	文件头注释 .....	15
• 2.11.3	类注释 .....	16
• 2.11.4	函数注释 .....	17
• 2.11.5	将代码注释掉 .....	20
• 2.11.6	TODO 待办注释 .....	20
2.12	代码块 .....	21
<b>3</b>	<b>NET 编码规范 .....</b>	<b>21</b>
3.1	类库开发设计规范 .....	21
3.2	文件和结构 .....	21
3.3	程序集属性 .....	21
3.4	命名规范 .....	22
• 3.4.1	综合命名规范 .....	22
• 3.4.2	标识符的大小写命名规范 .....	22
• 3.4.3	匈牙利命名法 .....	24
• 3.4.4	户界面控件命名规范 .....	24
3.5	常量 .....	25

3.6	字符串 .....	26
3.7	数组和集合 .....	27
3.8	结构体 .....	30
•	3.8.1 结构体 vs 类 .....	30
3.9	类 .....	30
•	3.9.1 字段 .....	30
•	3.9.2 属性 .....	31
•	3.9.3 构造函数 .....	31
•	3.9.4 方法 .....	32
•	3.9.5 事件 .....	32
•	3.9.6 成员方法重载 .....	32
•	3.9.7 接口成员 .....	33
•	3.9.8 虚成员方法 .....	33
•	3.9.9 静态类 .....	34
•	3.9.10 抽象类 .....	34
3.10	命名空间 .....	34
3.11	错误和异常 .....	35
•	3.11.1 抛出异常 .....	35
•	3.11.2 异常处理 .....	36
3.12	资源清理 .....	38
•	3.12.1 Try-finally 块 .....	38
•	3.12.2 基础 Dispose 模式 .....	39
•	3.12.3 可终结类型 .....	44
•	3.12.4 重写 Dispose .....	50
3.13	交互操作 .....	51
•	3.13.1 P/Invoke .....	51
•	3.13.2 COM 交互操作 .....	54

# 1 概览

---

本文档为[玩儿道](#)项目组所使用的 .NET 编码规范。该规范源自于产品开发过程中的经验，并在不断完善。

如果您发现一些最佳实践或者话题并没有涵盖在本文档中，请联系我们，以不断充实完善本文档。

任何指导准则都可能会众口难调。本规范的目的在于帮助社区开发者提高开发效率，减少代码中可能出现的 bug，并增强代码的可维护性。万事开头难，采纳一个不熟悉的规范可能在初期会有一些棘手和困扰，但是这些不适应很快便会消失，它所带来的好处和优势很快便会显现，特别是在当您接手他人代码时。

## 1.1 原则和主旨

高质量的代码示例往往具有如下特质：

1. **易懂** – 代码示例必须易读且简单明确。它们必须能展示出重点所在。示例代码的相关部分应当易于重用。示例代码不可包含多余代码。它们必须带有相应文档说明。
2. **正确性** – 示例代码必须正确展示出其欲告知使用者的重点。代码必须经过测试，且可以按照文档描述进行编译和运行。
3. **一致性** – 示例代码应该按照一致的编程风格和设计来保证代码易读。同样的，不同代码示例之间也应当保持一致的风格和设计，使使用者能够很轻松的结合使用它们。一致性将我们一站式示例代码库优良的品质形象传递给使用者，展示出我们对于细节的追求。
4. **流行性** – 代码示例应当展示现行的编程实践，例如使用 **Unicode**，错误处理，防御式编程以及可移植性。示例代码应当使用当前推荐的运行时库和 **API** 函数，以及推荐的项目和生成设置。
5. **可靠性** – 代码示例必须符合法律，隐私和政策标准和规范。不允许展示入侵性或低质的编程实践，不允许永久改变机器状态。所有的安装和执行过程必须可以被撤销。

6. **安全性** - 示例代码应该展示如何使用安全的编程实践：例如最低权限原则，使用运行时库函数的安全版本，以及 SDL 推荐的项目设置。

合理使用编程实践，设计和语言特性决定了示例代码是否可以很好满足上述特性。本编程规范致力于帮助您创建代码示例以使使用者能够作为最佳实践来效仿和学习。

1.2 术语

在整个文档中，会有一些对于标准和实践的推荐和建议。一些实践是非常重要的，必须严格执行，另一些指导准则并不一定处处适用，但是会在特定的场景下带来益处。为了清楚陈述规范和实践的意图，我们会使用如下术语。

术语	意图	理由
☑一定请...	该规范或实践在任何情况下都应该遵守。如果您认为您的应用是例外，则可能不适用。	该规范用于减少 bug。
☒一定不要...	不允许应用该规范或实践。	
☑您应该...	该规范和实践适用于大多数情况。	该规范用于统一编程风格，保持一致和清晰的风格。
☒您不应该..	不应该应用该规范或实践，除非有合理的理由。	
☑您可以...	该标准和规范您可以按需应用。	该规范可用于编程风格，但不总是有益的。

## 2 通用编程规范

这些通用编程规范适用于所有语言-它们对代码风格，格式和结构提供了全局通用的指导。

### 2.1 明确性和一致性

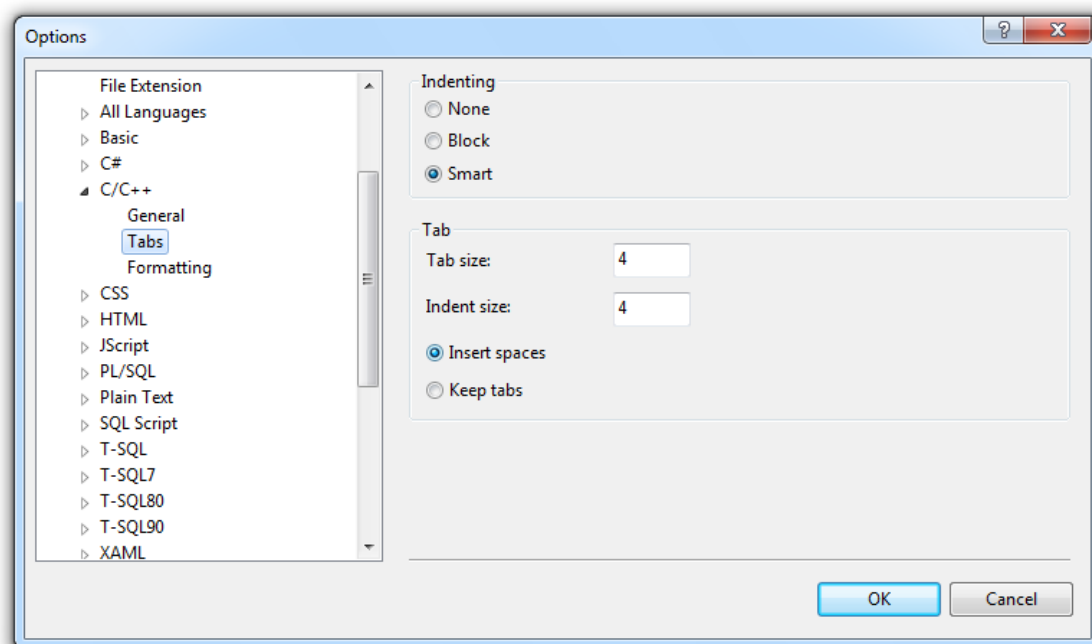
✅ **一定请**确保代码的明确性，易读性和透明性。编程规范致力于确保代码是易懂和易维护的。没有什么胜于清晰、简洁、自描述的代码。

✅ **一定请**确保一旦应用了某编程规范，需在所有代码中应用，以保持一致性。

### 2.2 格式和风格

❌ **一定不要**使用制表符。不同的文字编辑器使用不同的空格来生成制表符，这就带来了格式混乱。所有代码都应该使用 4 个空格来表示缩进。

可以配置 Visual Studio 文字编辑器，以空格代替制表符。



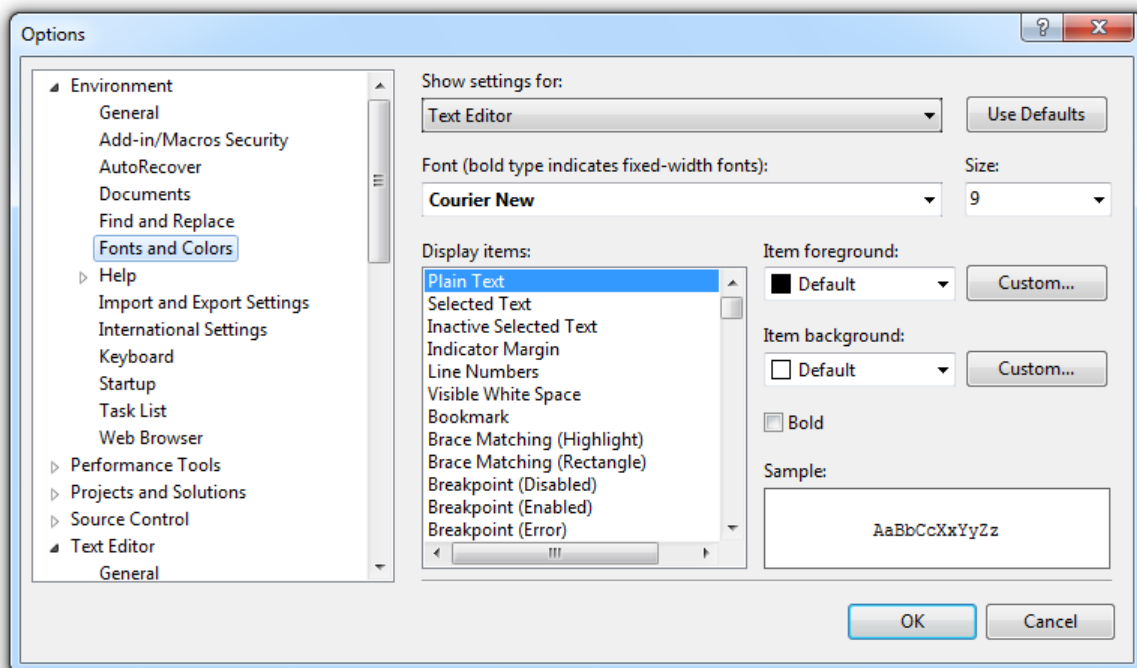
☑您应该限制一行代码的最大长度。过长的代码降低了代码易读性。为了提高易读性，将代码长度设置为 78 列。若 78 列太窄，可以设置为 86 或者 90。

Visual C# 示例：

```
// Get and display whether the primary access token of the process belongs
// to user account that is a member of the local Administrators group even
// if it currently is not elevated (IsUserInAdminGroup).
try
{
    bool fInAdminGroup = IsUserInAdminGroup();
    this.lbInAdminGroup.Text = fInAdminGroup.ToString();
}
catch (Exception ex)
{
    this.lbInAdminGroup.Text = "N/A";
    MessageBox.Show(ex.Message, "An error occurred in IsUserInAdminGroup",
        MessageBoxButtons.OK, MessageBoxIcon.Error);
}
```

Column 86

☑一定请在您的代码编辑器中使用定宽字体，例如 Courier New。





## 2.3 库的使用

❌ **一定不要**引用不必要的库，包括不必要的头文件，或引用不必要的程序集。注重细节能够减少项目生成时间，最小化出错几率，并给读者一个良好的印象。

## 2.4 全局变量

✅ **一定请**尽量少用全局变量。为了正确的使用全局变量，一般是将它们作为参数传入函数。永远不要在函数或类内部直接引用全局变量，因为这会引起一个副作用：在调用者不知情的情况下改变了全局变量的状态。这对于静态变量同样适用。如果您需要修改全局变量，您应该将其作为一个输出参数，或返回其一份全局变量的拷贝。

## 2.5 变量的声明和初始化

✅ **一定请**在最小的，包含该局部变量的作用域块内声明它。一般，如果语言允许，就仅在使用前声明它们，否则就在作用域块的顶端声明。

✅ **一定请**在声明变量时初始化它们。

✅ **一定请**在语言允许的情况下，将局部变量的声明和初始化或赋值置于同一行代码内。这减少了代码的垂直空间，确保了变量不会处在未初始化的状态。

```
// C#sample:
string name = myObject.Name;
int val = time.Hours;
```

❌ **一定不要**在同一行中声明多个变量。推荐每行只包含一句声明，这样有利于添加注释，也减少歧义。

例如 Visual C++ 示例，

**Good:**

```
CodeExample *pFirst = NULL; // Pointer of the first element.
CodeExample *pSecond = NULL; // Pointer of the second element.
```

**Bad:**

```
CodeExample *pFirst, *pSecond;
```

后一个代码示例经常被误写为:

```
CodeExample *pFirst, pSecond;
```

这种误写实际上等同于:

```
CodeExample *pFirst;
CodeExample pSecond;
```

## 2.6 函数的声明和调用

函数或方法的名称, 返回值, 参数列表可以有多种形式。原则上应该都将这些置于同一行代码内。如果带有过多参数不能置于一行代码, 可以进行换行: 多个参数一行或者一个参数一行。将返回值置于函数或方法名称的同一行。例如,

单行格式:

```
// C++ function declaration sample:
HRESULT DoSomeFunctionCall (int param1, int param2, int *param3);
// C++ / C# function call sample:
hr = DoSomeFunctionCall (param1, param2, param3);
'VB.NET function call sample:
hr = DoSomeFunctionCall (param1, param2, param3)
```

多行格式:

```
// C++ function declaration sample:
HRESULT DoSomeFunctionCall (int param1, int param2, int *param3,
    int param4, int param5);
// C++ / C# function call sample:
hr = DoSomeFunctionCall (param1, param2, param3,
    param4, param5);
'VB.NET function call sample:
hr = DoSomeFunctionCall (param1, param2, param3, _
    param4, param5)
```

将参数列表置于多行代码时, 每一个参数应该整齐排列于前一个参数的下方。第一个类型/参数对置于新行行首, 并缩进一个制表符宽度。函数或方法调用时的参数列表同样需按照这一格式。

```
// C++ / C# sample:
hr = DoSomeFunctionCall (
    hwnd,
    param1,
    param2,
    param3,
    param4,
    param5);
```

☑ **一定**请将参数排序，并首先将输入参数分组，再将输出参数放置最后。在参数组内，按照能够帮助程序员输入正确值的原则来将参数排序。比如，如果一个函数带有 2 个参数，“left”和“right”，将“left”置于“right”之前，则它们的放置顺序符合其参数名。当设计一系列具有相同参数的函数时，在各函数内使用一致的顺序。比如，如果一个函数带有一个输入类型为句柄的参数作为第一参数，那么所有相关函数都应该将该输入句柄作为第一参数。

## 2.7 代码语句

☒ **一定**不要在同一行内放置一句以上的代码语句。这会使得调试器的单步调试变得更为困难。

```
Good:
// C++ / C#sample:
a = 1;
b = 2;

Bad:
// C++ /C#sample:
a = 1; b = 2;
```

## 2.8 枚举

☑ **一定**请将代表某些值集合的强类型参数，属性和返回值声明为枚举类型。

☑ **一定**请在合适的情况下尽量使用枚举类型，而不是静态常量或“#define”值。枚举类型是一个具有一个静态常量集合的结构体。如果遵守这些规范，定义枚举类型，而不是带有静态常量的结构体，您便会得到额外的编译器和反射支持。

Good:

```
// C# sample:
Public enum Color
{
    Red,
    Green,
    Blue
}
```

Bad:

```
// C# sample:
publicstaticclassColor
{
    publicconst int Red = 0;
    publicconst int Green = 1;
    publicconst int Blue = 2;
}
```

❌ **一定不要**使用公开集合作为枚举（例如操作系统版本，您亲朋的姓名）。

✅ **一定请**为简单枚举提供一个 0 值枚举量，可以考虑将之命名为“None”。如果这个名称对于特定的枚举并不合适，可以自行定义为更准确的名称。

```
// C# sample:
publicenumCompression
{
    None = 0,
    GZip,
    Deflate
}
```

❌ **一定不要**在 .NET 中使用 Enum.IsDefined 来检查枚举范围。Enum.IsDefined 有 2 个问题。首先，它加载反射和大量类型元数据，代价极其昂贵。第二，它存在版本的问题。

Good:

```
// C# sample:
if (c > Color.Black || c < Color.White)
{
    throw new ArgumentOutOfRangeException (...);
}
```

```

Bad:
// C# sample:
if (!Enum.IsDefined (typeof (Color) , c) )
{
    throw new InvalidEnumArgumentException (...);
}

```

### 2.8.1 标志枚举

标志枚举用于对枚举值进行位运算的支持。标志枚举通常用于表示选项。

☑一定要将 `System.FlagsAttribute` 应用于标志枚举。一定不要将此属性用于简单枚举。

☑一定请利用 2 进制强大的能力，因为它可以自由的进行位异或运算。举例，

```

// C# sample:
[Flags]
public enum AttributeTargets
{
    Assembly = 0x0001,
    Class    = 0x0002,
    Struct   = 0x0004,
    ...
}

```

☑您应该提供一些特殊的枚举值，以便进行常见的标志枚举的组合运算。位运算属于高级任务，所以在简单任务中无需使用它们。`FileAccess.ReadWrite` 便是标志枚举特殊值的一个示例。然而，如果一个标志枚举中的某些值组合起来是非法的，您就不应该创建这样的标志枚举。

```

// C# sample:
[Flags]
public enum FileAccess
{
    Read = 0x1,
    Write = 0x2,
    ReadWrite = Read | Write
}

```

❌您不应该在标志枚举中使用 0 值，除非它代表“所有标志被清除了”，并被恰当的命名为类似“None”的名字。如下 C# 示例展示了一常见的检查某一标志是否被设置了的实现（查看如下 if-语句）。该检查运行结果正确，但是有一处例外，便是对于 0 值的检查，它的布尔表达式结果恒为 true。

Bad:

```
[Flags]
public enum SomeFlag
{
    ValueA = 0, // This might be confusing to users
    ValueB = 1,
    ValueC = 2,
    ValueBAndC = ValueB | ValueC,
}

SomeFlag flags = GetValue();
if ((flags & SomeFlag.ValueA) == SomeFlag.ValueA)
{
    ...
}
```

Good:

```
[Flags]
public enum BorderStyle
{
    Fixed3D          = 0x1,
    FixedSingle      = 0x2,
    None             = 0x0
}

if (foo.BorderStyle == BorderStyle.None)
{
    ...
}
```

## 2.9 空格

### 2.9.1 空行

✅您应该使用空行来分隔相关语句块。省略额外的空行会加大代码阅读难度。比如，您可以在变量声明和代码之间有一行空行。

Good:

```
// C++ sample:
```

```

void ProcessItem (const Item& item)
{
    int counter = 0;

    if (...)
    {
    }
}

```

**Bad:**

```

// C++ sample:
void ProcessItem (const Item& item)
{
    int counter = 0;

    // Implementation starts here
    //
    if (...)
    {
    }

}

```

在本例中，过多的空行造成了空行滥用，并不能使代码更易于阅读。

☑您应该使用 2 行空行来分隔方法实现或类型声明。

## 2.9.2 空格

空格通过降低代码密度以增加可读性。以下是使用空格符的一些指导规范:

☑您应该像如下般在一行代码中使用空格。

```

Good:
// C++ / C# sample:
CreateFoo () ;// No space between function name and parenthesis
Method (myChar, 0, 1) ;// Single space after a comma
x = array[index];// No spaces inside brackets
while (x == y) // Single space before flow control statements
if (x == y) // Single space separates operators

```

**Bad:**

```
// C++ / C# sample:
CreateFoo ( ) ;// Space between function name and parenthesis
Method (myChar,0,1) ;// No spaces after commas
CreateFoo ( myChar, 0, 1 ) ;// Space before first arg, after last arg
x = array[ index ] ;// Spaces inside brackets
while (x == y) // No space before flow control statements
if (x==y) // No space separates operators
```

## 2.10 大括号

☑一定请在[一站式示例代码库](#)的代码示例中使用Allman风格的大括号用法。

Allman 风格是以 Eric Allman 命名的，有时也被称为"ANSI 风格"。该风格将大括号与相关代码置于下一行内，与控制语句的缩进相同。大括号内的语句缩进一个等级。

**Good:**

```
// C++ / C# sample:
if (x > 5)
{
    y = 0;
}
```

**Bad (in All-In-One Code Framework samples) :**

```
// C++ / C# sample:
if (x > 5) {
    y = 0;
}
```

☑您应该在即使是单行条件式的情况下也使用大括号。这样做使得将来增加条件式更简便，并减少制表符引起的歧义。

**Good:**

```
// C++ / C# sample:
if (x > 5)
{
    y = 0;
}
```

**Bad:**

```
// C++ / C# sample:
```



```
if (x > 5) y = 0;
```

## 2.11 注释

☑您应该使用注释来解释一段代码的设计意图。一定不要让注释仅仅是重复代码。

### Good:

```
// Determine whether system is running Windows Vista or later operating
// systems (major version >= 6) because they support linked tokens, but
// previous versions (major version < 6) do not.
```

### Bad:

```
// The following code sets the variable i to the starting value of the
// array. Then it loops through each item in the array.
```

☑您应该使用 `//` 注释方式，而不是 `/* */` 来为 C++ 和 C# 代码作注释。即使注释跨越多行代码，单行注释语法 (`// ...`) 仍然是首选。

```
// Determine whether system is running Windows Vista or later operating
// systems (major version >= 6) because they support linked tokens, but
// previous versions (major version < 6) do not.
if (Environment.OSVersion.Version.Major >= 6)
{
    ' Get and display the process elevation information (IsProcessElevated)
    ' and integrity level (GetProcessIntegrityLevel). The information is not
    ' available on operating systems prior to Windows Vista.
    If (Environment.OSVersion.Version.Major >= 6) Then
    EndIf
}
```

☑您应该将注释缩进为被其描述的代码的同一级。

☑您应该在注释中使用首字母大写的完整的句子，以及适当的标点符号和拼写。

### Good:

```
// Initialize the components on the Windows Form.
InitializeComponent ();

' Initialize the components on the Windows Form.
InitializeComponent ()
```

**Bad:**

```
//initialize the components on the Windows Form.
InitializeComponent ();

'initialize the components on the Windows Form
InitializeComponent ()
```

### 2.11.1 内联代码注释

内联注释应该置于独立行，并与所描述的代码具有相同的缩进。其之前需放置一个空行。其之后不需要空行。描述代码块的注释应该置于独立行，与所描述的代码具有相同的缩进。其之前和之后都有一个空行。举例：

```
if (MAXVAL >= exampleLength)
{
    // Reprort the error.
    ReportError (GetLastError () ) ;

    // The value is out of range, we cannot continue.
    return E_INVALIDARG;
}
```

当内联注释只为结构体，类成员变量，参数和短语句做描述时，则内联注释允许出现在和实际代码的同一行。在本例中，最好将所有变量的注释对齐。

```
class Example
{
public:
    ...

    void TestFunction
    {
        ...
        do
        {
            ...
        }
        while (!Finished) ; // Continue if not finished.
    }

private:
    int m_length;          // The length of the example
```

```
float m_accuracy; // The accuracy of the example
};
```

❌您不应该在代码中留有过多注释。如果每一行代码都有注释，便会影响到可读性和可理解性。单行注释应该用于代码的行为并不是那么明显易懂的情况。

如下代码包含多余注释：

```
Bad:
// Loop through each item in the wrinkles array
for (int i = 0; i <= nLastWrinkle; i++)
{
    Wrinkle *pWrinkle = apWrinkles[i];    // Get the next wrinkle
    if (pWrinkle->IsNew () && // Process if it's a new wrinkle
        nMaxImpact < pWrinkle->GetImpact ()) // And it has the biggest impact
    {
        nMaxImpact = pWrinkle->GetImpact (); // Save its impact for comparison
        pBestWrinkle = pWrinkle;           // Remember this wrinkle as well
    }
}
```

更好的实现如下：

```
Good:
// Loop through each item in the wrinkles array, find the Wrinkle with
// the largest impact that is new, and store it in 'pBestWrinkle'.
for (int i = 0; i <= nLastWrinkle; i++)
{
    Wrinkle *pWrinkle = apWrinkles[i];
    if (pWrinkle->IsNew () && nMaxImpact < pWrinkle->GetImpact ())
    {
        nMaxImpact = pWrinkle->GetImpact ();
        pBestWrinkle = pWrinkle;
    }
}
```

✅您应该为那些仅通过阅读很难理解其意图的代码添加注释。

### 2.11.2 文件头注释

✅一定请为每一份手写代码文件加入文件头注释。如下为文件头注释：

VC++ 和 VC# 文件头注释模板:

```

/***** Module Header *****/
Module Name:  <File Name>
Project:      <Sample Name>
Copyright (c) Microsoft Corporation.

<Description of the file>

This source is subject to the Microsoft Public License.
See http://www.microsoft.com/opensource/licenses.mspx#Ms-PL.
All other rights reserved.

THIS CODE AND INFORMATION IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND,
EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE IMPLIED
WARRANTIES OF MERCHANTABILITY AND/OR FITNESS FOR A PARTICULAR PURPOSE.
/*****/

```

举例,

```

/***** Module Header *****/
Module Name:  CppUACSelfElevation.cpp
Project:      CppUACSelfElevation
Copyright (c) Microsoft Corporation.

User Account Control (UAC) is a new security component in Windows Vista and
newer operating systems. With UAC fully enabled, interactive administrators
normally run with least user privileges. This example demonstrates how to
check the privilege level of the current process, and how to self-elevate
the process by giving explicit consent with the Consent UI.

This source is subject to the Microsoft Public License.
See http://www.microsoft.com/opensource/licenses.mspx#Ms-PL.
All other rights reserved.

THIS CODE AND INFORMATION IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND,
EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE IMPLIED
WARRANTIES OF MERCHANTABILITY AND/OR FITNESS FOR A PARTICULAR PURPOSE.
/*****/

```

### 2.11.3 类注释

☑您应该为每一个重要的类或结构体作注释。注释的详细程度应该视代码的用户群而定。

C++ 类注释模板:

```
//
//  NAME:  class <Class name>
//  DESCRIPTION:  <Class description>
//
```

C#和 VB.NET 使用.NET 描述性 XML 文档化注释。当您编译.NET 项目，并带有/doc 命令时，编译器会在源代码中搜索 XML 标签，并生成 XML 文档。

C# 类注释模板:

```
///

```

举例,

```
//
//  NAME:  class CodeExample
//  DESCRIPTION:  The CodeExample class represents an example of code, and
//  tracks the length and complexity of the example.
//
class CodeExample
{
    ...
};

///

```

#### 2.11.4 函数注释

☑您应该为所有重要的 Public 或非 Public 函数作注释。注释的详细程度应该视代码的用户群而定。

### C++ 函数注释模板:

```

/*-----
FUNCTION: <Function prototype>

PURPOSE:
<Function description>

PARAMETERS:
<Parameter name> -<Parameter description>

RETURN VALUE:
<Description of function return value>

EXCEPTION:
<Exception that may be thrown by the function>

EXAMPLE CALL:
<Example call of the function>

REMARKS:
<Additional remarks of the function>
-----*/

```

C# 和 VB.NET 使用 .NET 描述性 XML 文档化注释。在一般情况下，至少需要一个<summary>元素，一个<parameters>元素和<returns>元素。会抛出异常的方法应使用<exception>元素来告知使用者哪些异常会被抛出。

### C# 函数注释模板:

```

///<summary>
///<Function description>
///</summary>
///<param name="Parameter name">
/// <Parameter description>
/// </param>
///<returns>
///<Description of function return value>
///</returns>
///<exception cref="<Exception type>">
///<Exception that may be thrown by the function>

```

```
///</exception>
```

举例,

```
/*-----
FUNCTION: IsUserInAdminGroup (HANDLE hToken)

PURPOSE:
The function checks whether the primary access token of the process
belongs to user account that is a member of the local Administrators
group, even if it currently is not elevated.

PARAMETERS:
hToken -the handle to an access token.

RETURN VALUE:
Returns TRUE if the primary access token of the process belongs to user
account that is a member of the local Administrators group. Returns FALSE
if the token does not.

EXCEPTION:
If this function fails, it throws a C++ DWORD exception which contains
the Win32 error code of the failure.

EXAMPLE CALL:
    try
    {
if (IsUserInAdminGroup (hToken) )
        wprintf (L"User is a member of the Administrators group\n") ;
    else
        wprintf (L"User is not a member of the Administrators group\n") ;
    }
    catch (DWORD dwError)
    {
        wprintf (L"IsUserInAdminGroup failed w/err %lu\n", dwError) ;
    }
-----*/

///<summary>
/// The function checks whether the primary access token of the process
/// belongs to user account that is a member of the local Administrators
/// group, even if it currently is not elevated.
///</summary>
///<param name="token">The handle to an access token</param>
///<returns>
/// Returns true if the primary access token of the process belongs to
```

```

/// user account that is a member of the local Administrators group.
/// Returns false if the token does not.
///</returns>
///<exception cref="System.ComponentModel.Win32Exception">
/// When any native Windows API call fails, the function throws a
/// Win32Exception with the last error code.
///</exception>

```

任何调用失败会带来副作用的方法或函数，都需要清楚地注释交代那些副作用的后果。一般而言，代码在发生错误或失败时不应该有副作用。当出现副作用时，在编写代码时应该有清楚的理由。当函数没有输出参数，或者只有一些单纯作为输出参数的情况下，无需交代理由。

### 2.11.5 将代码注释掉

当您用多种方法实现某些任务时，将代码注释掉便是必须的。除了第一个方法，其余实现方法都会被注释掉。使用 [-or-] 来分隔多个方法。举例，

```

// C++ / C# sample:
// Demo the first solution.
DemoSolution1 ();

// [-or-]

// Demo the second solution.
//DemoSolution2 ();


' VB.NET sample:
' Demo the first solution.
DemoSolution1 ();

' [-or-]

' Demo the second solution.
'DemoSolution2 ();

```

### 2.11.6 TODO 待办注释

 **一定不要在已发布的代码示例中使用 TODO 待办注释。**每一个代码示例都必须完整，在代码中不能有未完成的任务。



2.12 代码块

☑一定请在大量代码会因为更具结构化而获益时，使用代码块声明。通过作用域或者功能性分类，将大量代码分组，会改善代码易读性和结构。

C# 代码块:

```
#region Helper Functions for XX
...
#endregion
```

3 NET 编码规范

以下编程规范适用于 C# 和 VB.NET。

3.1 类库开发设计规范

MSDN上的[类库开发设计规范](#)对如何编写优秀的托管代码进行了细致的讨论。本章节的内容着重于一些重要的规范，以及一站式代码示例库对于该规范的一些例外情况。因此，建议你同时参照这两份文档。

3.2 文件和结构

☒一定不要在一个源文件内拥有一个以上的 Public 类型，除非它们只有泛型参数个数的差别，或者具有嵌套关系。一个文件内有多个内部类型是允许的。

☑一定请以源文件所含的 Public 类名命名该文件。比如， MainForm 类应该在 MainForm.cs 文件内，而 List<T>类应该在 List.cs 文件内。

3.3 程序集属性

程序集应当包含适当的属性值来描述其命名，版权，等等。

Standard	Example
将 copyright 设置为 Copyright © Microsoft	[assembly: AssemblyCopyright ("Copyright © Microsoft Corporation 2010")]

Corporation 2010	
将 AssemblyCompany 设置为 Microsoft Corporation	<code>[assembly: AssemblyCompany ("Microsoft Corporation")]</code>
将 AssemblyTitle 和 AssemblyProduct 设置为当前示例名	<code>[assembly: AssemblyTitle ("CSNamedPipeClient")]</code> <code>[assembly: AssemblyProduct ("CSNamedPipeClient")]</code>

### 3.4 命名规范

#### 3.4.1 综合命名规范

☑一定请为各种类型，函数，变量，特性和数据结构选取有意义的命名。其命名应能反映其作用。

☒您不应该在标识符名中使用缩短或缩略形式的词。比如，使用“GetWindow”而不是“GetWin”。对于公共类型，线程过程，窗口过程，和对话框过程函数，为“ThreadProc”，“DialogProc”，“WndProc”等使用公共后缀。

☒一定不要使用下划线，连字号，或其他任何非字母数字的字符。

#### 3.4.2 标识符的大小写命名规范

如下表格描述了对不同类型标识符的大小写命名规范。

标识符	规范	命名结构	示例
类，结构体	Pascal 规范	名词	<code>public class ComplexNumber { ... }</code> <code>public struct ComplexStruct { ... }</code>
命名空间	Pascal 规范	名词 ☒一定不要以相同的名称来命名命名空间和其内部的类型。	<code>namespace Microsoft.Sample.Windows7</code>
枚举	Pascal 规范	名词 ☑一定请以复数名词或名词短语来命名标志枚举，以单数名词或名词短语来命名简单枚举。	<code>[Flags]</code> <code>public enum ConsoleModifiers { Alt, Control }</code>
方法	Pascal 规范	动词或动词短语	<code>public void Print () { ... }</code> <code>public void ProcessItem () { ... }</code>

Public 属性	Pascal 规范	名词或形容词 <input checked="" type="checkbox"/> 一定请以集合中项目的复数形式命名该集合，或者单数名词后面跟“List”或者“Collection”。 <input checked="" type="checkbox"/> 一定请以肯定短语来命名布尔属性，（CanSeek，而不是CantSeek）。当以“Is,” “Can,” or “Has”作布尔属性的前缀有意义时，您也可以这样做。	<pre>publicstring CustomerName public ItemCollection Items public boolCanRead</pre>
非 Public 属性	Camel 规范或 _camel 规范	名词或形容词 <input checked="" type="checkbox"/> 一定请在您使用 '_' 前缀时，保持代码一致性。	<pre>privatestring name; privatestring _name;</pre>
事件	Pascal 规范	动词或动词短语 <input checked="" type="checkbox"/> 一定请用现在式或过去式来表明事件之前或是之后的概念。 <input checked="" type="checkbox"/> 一定不要使用“Before”或者“After”前缀或后缀来指明事件的先后。	<pre>// A close event that is // raised after the window is // closed. publicevent WindowClosed  // A close event that is // raised before a window is // closed. publicevent WindowClosing</pre>
委托	Pascal 规范	<input checked="" type="checkbox"/> 一定请为用于事件的委托增加‘EventHandler’后缀。 <input checked="" type="checkbox"/> 一定请为除了用于事件处理程序之外的委托增加‘Callback’后缀。 <input checked="" type="checkbox"/> 一定不要为委托增加“Delegate”后缀。	<pre>publicdelegate WindowClosedEventHandler</pre>
接口	Pascal 规范，带有‘I’前缀	名词	<pre>publicinterfaceIDictionary</pre>
常量	Pascal 规范用于 Public 常量； Camel 规范用于 Internal 常量； 只有 1 或 2 个字符的缩写需全部字符大写。	名词	<pre>publicconststring MessageText = "A"; privateconststring messageText = "B"; publicconst double PI= 3.14159...;</pre>
参数，变量	Camel 规范	名词	<pre>int customerID;</pre>
泛型参数	Pascal 规范，带有‘T’前缀	名词 <input checked="" type="checkbox"/> 一定请以描述性名称命名泛型参数，除非单字符名称已有足够描述性。	<pre>T, TItem, TPolicy</pre>

		<div><div>☑一定请以 T 作为描述性类型参数的前缀。</div><div>☑您应该使用 T 作为单字符类型参数的名称。</div></div>	
资源	Pascal 规范	<div>名词</div> <div><div>☑一定请提供描述性强的标识符。同时，尽可能保持简洁，但是不应该因空间而牺牲可读性。</div><div>☑一定请仅为命名资源使用字母数字字符和下划线。</div></div>	ArgumentExceptionInvalidName

3.4.3 匈牙利命名法

☒一定不要在.NET 中使用匈牙利命名法（例如，不要在变量名称内带有其类型指示符）。

3.4.4 户界面控件命名规范

用户控件应该使用如下前缀，其重要目的是使代码更易读。

控件类型	前缀
Button	btn
CheckBox	chk
CheckedListBox	lst
ComboBox	cmb
ContextMenu	mnu
DataGrid	dg
DateTimePicker	dtp
Form	suffix: XXXForm
GroupBox	grp
ImageList	iml
Label	lb
ListBox	lst
ListView	lvw
Menu	mnu
MenuItem	mnu
NotificationIcon	nfy
Panel	pnl

PictureBox	pct
ProgressBar	prg
RadioButton	rad
Splitter	spl
StatusBar	sts
TabControl	tab
TabPage	tab
TextBox	tb
Timer	tmr
TreeView	tvw

比如，对于“File | Save”菜单选线，“Save”菜单项应该命名为“mnuFileSave”。

### 3.5 常量

☑一定请将那些永远不会改变值定义为常量字段。编译器直接将常量字段嵌入调用代码处。所以常量值永远不会被改变，且并不会打破兼容性。

```
publicclassInt32
{
    publicconstint MaxValue = 0x7fffffff;
    publicconstint MinValue = unchecked ( (int) 0x80000000 );
}

PublicClass Int32
PublicConst MaxValue AsInteger = &H7FFFFFFF
PublicConst MinValue AsInteger = &H80000000
EndClass
```

☑一定请为预定义的对象实例使用 `public static (shared) readonly` 字段。如果有预定义的类型实例，也将该类型定义为 `public static readonly`。举例，

```
publicclassShellFolder
{
    publicstaticreadonlyShellFolder ProgramData = newShellFolder ("ProgramData");
    publicstaticreadonlyShellFolder ProgramFiles = newShellFolder ("ProgramData");
    ...
}
```

```

PublicClass ShellFolder
PublicSharedReadOnly ProgramData AsNew ShellFolder ("ProgramData")
PublicSharedReadOnly ProgramFiles AsNew ShellFolder ("ProgramFiles")
...
EndClass

```

### 3.6 字符串

❌ **一定不要**使用 '+' 操作符（VB.NET 中的 '&'）来拼接大量字符串。相反，您应该使用 `StringBuilder` 来实现拼接工作。然而，拼接少量的字符串时，一定请使用 '+' 操作符（VB.NET 中的 '&'）。

```

Good:
StringBuilder sb = newStringBuilder();
for (int i = 0; i < 10; i++)
{
    sb.Append(i.ToString());
}

Bad:
string str = string.Empty;
for (int i = 0; i < 10; i++)
{
    str += i.ToString();
}

```

✅ **一定请**使用显式地指定了字符串比较规则的重载函数。一般来说，需要调用带有 `StringComparison` 类型参数的重载函数。

✅ **一定请**在对文化未知的字符串做比较时，使用 `StringComparison.Ordinal`

和 `StringComparison.OrdinalIgnoreCase` 作为您的安全默认值，以提高性能表现。

✅ **一定请**在向用户输出结果时，使用基于 `StringComparison.CurrentCulture` 的字符串操作。

✅ **一定请**在比较语言无关字符串（符号，等）时，使用非语言学的 `StringComparison.Ordinal` 或者 `StringComparison.OrdinalIgnoreCase` 值，而不是基于 `CultureInfo.InvariantCulture` 的字符串操作。一般而言，不要使用基于 `StringComparison.InvariantCulture` 的字符串操作。一个例外是当你坚持其语言上有意义，而与具体文化无关的情况。

✅一定请使用String.Equals的重载版本来测试 2 个字符串是否相等。比如，忽略大小写后，判断 2 个字符串是否相等，

```
if (str1.Equals (str2, StringComparison.OrdinalIgnoreCase) )

If (str1.Equals (str2, StringComparison.OrdinalIgnoreCase) ) Then
```

❌一定不要使用 String.Compare或CompareTo的重载版本来检验返回值是否为 0，来判断字符串是否相等。这 2 个函数是用于字符串排序，而非检查相等性。

✅一定请在字符串比较时，以String.ToUpperInvariant函数使字符串规范化，而不用String.ToLowerInvariant。

### 3.7 数组和集合

✅您应该在低层次函数中使用数组，来减少内存消耗，增强性能表现。对于公开接口，则偏向选择集合。

集合提供了对于其内容更多的控制权，可以随着时间改善，提高可用性。另外，不推荐在只读场景下使用数组，因为数组克隆的代价太高。

然而，如果您把熟练开发者作为目标，对于只读场景使用数组也是个不错的主意。数组的内存占用比较低，这减少了工作区，并因为运行时的优化能更快的访问数组元素。

❌一定不要使用只读的数组字段。字段本身只读，不能被修改，但是其内部元素可以被修改。以下示例展示了使用只读数组字段的陷阱：

```
Bad:
publicstaticreadonlychar[] InvalidPathChars = { '\\', '<', '>', '|' };
```

这允许调用者修改数组内的值：

```
InvalidPathChars[0] = 'A';
```

您可以使用一个只读集合（只要其元素也是不可变的），或者在返回之前进行数组克隆。然而，数组克隆的代价可能过高：

```
Public static ReadOnlyCollection<char> GetInvalidPathChars ()
{
    return Array.AsReadOnly (badChars) ;
}

Public static char[] GetInvalidPathChars ()
{
    return (char[]) badChars.Clone () ;
}
```

☑您应该使用不规则数组来代替使用多维数组。一个不规则数组是指其元素本身也是一个数组。构成元素的数组可能有不同大小，这样相较于多维数组能减少一些数据集的空间浪费（例如，稀疏矩阵）。另外，CLR 能够对不规则数组的索引操作进行优化，所以在某些情景下，具有更好的性能表现。

```
// 不规则数组
int[][] jaggedArray =
{
    newint[] {1,2,3,4},
    newint[] {5,6,7},
    newint[] {8},
    newint[] {9}
};

Dim jaggedArray AsInteger () () = NewInteger () () _
{ _
    NewInteger () {1, 2, 3, 4}, _
    NewInteger () {5, 6, 7}, _
    NewInteger () {8}, _
    NewInteger () {9} _
}

// 多维数组
int [,] multiDimArray =
{
    {1,2,3,4},
    {5,6,7,0},
    {8,0,0,0},
    {9,0,0,0}
};
```



```
Dim multiDimArray (,) As Integer = _
{ _
    {1, 2, 3, 4}, _
    {5, 6, 7, 0}, _
    {8, 0, 0, 0}, _
    {9, 0, 0, 0} _
}
```

☑ **一定**请将代表了读/写集合的属性或返回值声明为 `Collection<T>` 或其子类，将代表了只读集合的属性或返回值声明为 `ReadOnlyCollection<T>` 或其子类。

☑ **您应该**重新考虑对于 `ArrayList` 的使用，因为所有添加至其中的对象都被当做 `System.Object`，当从 `ArrayList` 取回值时，这些对象都会拆箱，并返回其真实的值类型。所以我们推荐您使用定制类型的集合，而不是 `ArrayList`。比如，.NET 在 `System.Collection.Specialized` 命名空间内为 `String` 提供了强类型集合 `StringCollection`。

☑ **您应该**重新考虑对于 `Hashtable` 的使用。相反，您应该尝试其他字典类，例如 `StringDictionary`，`NameValueCollection`，`HybridCollection`。除非 `Hashtable` 只存储少量值，最好不要使用 `Hashtable`。

☑ **您应该**在实现集合类型时，为其实现 `IEnumerable` 接口，这样该集合便能用于 LINQ to Objects。

☒ **一定**不要在同一个类型上同时实现 `IEnumerator<T>` 和 `IEnumerable<T>` 接口。同样，也不要同时实现非泛型接口 `IEnumerator` 和 `IEnumerable`。所以，一个类型只能成为一个集合或者一个枚举器，而不可二者皆得。

☒ **一定**不要返回数组或集合的 `null` 引用。空值数组或集合的含义在代码环境中很难被理解。比如，一个用户可能假定如下代码能够正常运行，所以应该返回一个空数组或集合，而不是 `null` 引用。

```
int[] arr = SomeOtherFunc ();
foreach (int v in arr)
{
    ...
}
```

```
}
```

### 3.8 结构体

✅ **一定请**确保将所有实例数据设置为 0 值，false 或者是 null。当创建结构体数组时，这样能防止意外创建了无效实例。

✅ **一定请**为值类型实现 IEquatable<T>接口。值类型的 Object.Equals 方法会引起装箱操作。且因为使用了反射特性，所以其默认实现效率不高。IEquatable<T>.Equals 较其有相当大的性能提升，且其实现可以不引发装箱操作。

#### 3.8.1 结构体vs类

❌ **一定不要**定义结构体，除非其具有如下特性：

- 它在逻辑上代表了一个单值，类似于原始类型（例如，int、double，等等）。
- 其示例大小小于 16 字节。
- 它是不可变的。
- 它不会引发频繁的装箱拆箱操作。

其余情况下，您应该定义类，而不是结构体。

### 3.9 类

✅ **一定请**使用继承来表示 “is a” 关系，例如 “猫是一种动物”。

✅ **一定请**使用接口，例如 IDisposable，来表示 “can do” 关系，例如 “对象能被释放”。

#### 3.9.1 字段

❌ **一定不要**提供 Public 或 Protected 的实例字段。Public 或 Protected 字段并没有受到代码访问安全需求的保护。我们应该使用 Private 字段，并通过属性来提供访问接口。

✅一定请将预定义对象实例定义为 `public static readonly` 字段。

✅一定请将永远不会改变的字段定义为常量字段。

❌一定不要将可变类型定义为只读字段。

### 3.9.2 属性

✅一定请创建只读属性，如果用户不应该具有修改这些属性值的能力。

❌一定不要提供只写属性。如果没有提供属性设置器，请使用一个方法来实现相同的功能。方法名应该以 `Set` 开头，后面跟着属性名。

✅一定请为所有属性提供合理的默认值，并确保默认值不会引发安全漏洞或一个极端低效的设计。

❌您不应该从属性访问器内抛出异常。属性访问器应该只包含简单的操作，且不带任何前置条件。如果一个属性访问器可能抛出异常，那么考虑将其重新设计为一个方法。该推荐方法并不适用于索引器。索引器可以因为无效实参而抛出异常。从属性设置器内抛出异常是有效且可接受的。

### 3.9.3 构造函数

✅一定请尽量减少构造函数的工作量。除了得到构造函数参数，设置主要数据成员，构造函数不应该有太多的工作量。其余工作量应该被推迟，直到必须。

✅一定请在恰当的时候，从实例构造函数内抛出异常。

✅一定请在需要默认构造函数的情况下，显式的声明它。即使有时编译器为自动的为您的类增加一个默认构造函数，但是显式的声明使得代码更易维护。这样即使您增加了一个带有参数的构造函数，也能确保默认构造函数仍然会被定义。

☒ **一定不要在对象构造函数内部调用虚方法。**调用虚方法时，实际调用了继承体系最底层的覆盖（override）方法，而不考虑定义了该方法的类的构造函数是否已被调用。

### 3.9.4 方法

☑ **一定请将所有输出参数放置于所有传值和传引用参数（除去参数数组）的之后，**即使它引起了重载方法之前不一致的参数顺序。

☑ **一定请验证传递给 Public, Protected 或显式实现的成员方法的实参。**如果验证失败，则抛出 `System.ArgumentException`，或者其子集：如果一个 `null` 实参传递给成员，而成员方法不支持 `null` 实参，则抛出 `ArgumentNullException`。如果实参值超出由调用方法定义的可接受范围，则抛出 `ArgumentOutOfRangeException` 异常。

### 3.9.5 事件

☑ **一定请注意事件处理方法中可能会执行任意代码。**考虑将引发事件的代码放入一个 `try-catch` 块中，以避免由事件处理方法中抛出的未处理异常引起的程序终止。

☒ **一定不要在有性能要求的 API 中使用事件。**虽然事件易于理解和使用，但是就性能和内存消耗而言，它们不如虚函数。

### 3.9.6 成员方法重载

☑ **一定请使用成员方法重载，而不是定义带有默认参数的成员方法。**默认参数并不是 CLS 兼容的，所以不能被某些语言重用。同时，带有默认参数的成员方法存在一个版本问题。我们考虑成员方法的版本 1 将可选参数默认设置为 123。当编译代码调用该方法，且没有指定可选参数时，编译器在调用处直接将 123 嵌入代码中。现在，版本 2 将默认参数修改为 863，如调用代码没有重新编译，那么它会调用版本 2 的方法，并传递 123 作为其参数。（123 是版本 1 的默认参数，而不是版本 2 的。）。

Good:

```

Public OverloadsSub Rotate (ByVal data As Matrix)
    Rotate (data, 180)
EndSub

PublicOverloadsSub Rotate (ByVal data As Matrix, ByVal degrees AsInteger)
    ' Do rotation here
EndSub

Bad:
PublicSub Rotate (ByVal data As Matrix, OptionalByVal degrees AsInteger = 180)
    ' Do rotation here
EndSub

```

❌ **一定不要**任意改动重载方法中的参数名。如果一个重载函数中的参数代表着另一个重载函数中相同的参数，该参数则应该有相同的命名。具有相同命名的参数应该在重载函数中出现在同一位置。

✅ **一定请**仅将最长重载函数设为虚函数（为了拓展性考虑）。短重载函数应该一直调用到长重载函数。

### 3.9.7 接口成员

❌ **您不应该**在没有合理理由的情况下显式的实现接口成员。显式实现的成员可能使开发者感到困惑，因为他们不会出现在 **Public** 成员列表内，且会造成对值类型不必要的装箱拆箱操作。

✅ **您应该**在成员只通过接口来调用的情况下，显式的实现成员接口。

### 3.9.8 虚成员方法

相较于回调和事件，虚成员方法性能上有更好的表现。但是比非虚方法在性能上低一点。

❌ **一定不要**在没有合理理由的情况下，将成员方法设置为虚方法，您必须意识到相关设计，测试，维护虚方法带来的成本。

✅ **您应该**倾向于为虚成员方法设置为 **Protected** 的访问性，而不是 **Public** 访问性。**Public** 成员应该通过调用 **Protected** 的虚方法来提供拓展性（如果需要的话）。

### 3.9.9 静态类

✅一定请合理使用静态类。静态类应该被用于框架内基于对象的核心支持辅助类。

### 3.9.10 抽象类

❌一定不要在抽象类中定义 Public 或 Protected-Internal 的构造函数。

✅一定请为抽象类定义一个 Protected，或 Internal 构造函数。

Protected 构造函数更常见，因为其允许当子类创建时，基类可以完成自己的初始化工作。

```
publicabstractclassClaim
{
    protected Claim ()
    {
        ...
    }
}
```

internal 构造函数用于限制将抽象类的实现具化到定义该类的程序集。

```
publicabstractclassClaim
{
    internal Claim ()
    {
        ...
    }
}
```

## 3.10 命名空间

✅一定请在一站式代码示例库中使用 Visual Studio 创建的项目的默认的命名空间。无需重命名命名空间为 Microsoft.Sample.TechnologyName。

## 3.11 错误和异常

### 3.11.1 抛出异常

✅ **一定请**通过抛出异常来告知执行失败。异常在框架内是告知错误的主要手段。如果一个成员方法不能成功的如预期般执行，便应该认为是执行失败，并抛出一个异常。一定不要返回一个错误代码。

✅ **一定请**抛出最明确，最有意义的异常（继承体系最底层的）。比如，如果传递了一个 `null` 实参，则抛出 `ArgumentNullException`，而不是其基类 `ArgumentException`。抛出并捕获 `System.Exception` 异常通常都是没有意义的。

❌ **一定不要**将异常用于常规的控制流。除了系统故障或者带有潜在竞争条件的操作，您编写的代码都不应该抛出异常。比如，在调用可能失败或抛出异常的方法前，您可以检查其前置条件，举例，

```
// C# 示例:
if (collection != null &&!collection.IsReadOnly)
{
    collection.Add (additionalNumber) ;
}

' VB.NET 示例:
If ( (Not collection IsNothing) And (Not collection.IsReadOnly) ) Then
    collection.Add (additionalNumber)
EndIf
```

❌ **一定不要**从异常过滤器块内抛出异常。当一个异常过滤器引发了一个异常时，该异常会被 CLR 捕获，该过滤器返回 `false`。该行为很难与过滤器显式的执行并返回错误区分开，所以会增加调试难度。

```
' VB.NET sample
' This is bad design. The exception filter (When clause)
' may throw an exception when the InnerException property
' returns null
Try
...
Catch e As ArgumentException _
When e.InnerException.Message.StartsWith ("File")
...

```

EndTry

☒ **一定不要**显式的从 finally 块内抛出异常。从调用方法内隐式的抛出异常是可以接受的。

### 3.11.2 异常处理

☒ **您不应该**通过捕获笼统的异常，例如 `System.Exception`，`System.SystemException`，或者.NET 代码中其他异常，来隐藏错误。一定要捕获代码能够处理的、明确的异常。您应该捕获更加明确的异常，或者在 Catch 块的最后一条语句处重新抛出该普通异常。以下情况隐藏错误是可以接受的，但是其发生几率很低：

```

Good:
// C# sample:
try
{
    ...
}
catch (System.NullReferenceException exc)
{
    ...
}
catch (System.ArgumentOutOfRangeException exc)
{
    ...
}
catch (System.InvalidCastException exc)
{
    ...
}

' VB.NET sample:
Try
    ...
Catch exc As System.NullReferenceException
    ...
Catch exc As System.ArgumentOutOfRangeException
    ...
Catch exc As System.InvalidCastException
    ...
EndTry

```

**Bad:**



```

// C# sample:
try
{
    ...
}
catch (Exception ex)
{
    ...
}

' VB.NET sample:
Try
    ...
Catch ex As Exception
    ...
EndTry

```

☑一定请在捕获并重新抛出异常时，倾向使用 **throw**。这是保持异常调用栈的最佳途径：

```

Good:
// C# sample:
try
{
    ... // Do some reading with the file
}
catch
{
    file.Position = position; // Unwind on failure
    throw; // Rethrow
}

' VB.NET sample:
Try
    ... ' Do some reading with the file
Catch ex As Exception
    file.Position = position ' Unwind on failure
    Throw ' Rethrow
EndTry

Bad:
// C# sample:
try
{
    ... // Do some reading with the file
}
catch (Exception ex)

```

```

{
    file.Position = position; // Unwind on failure
throw ex; // Rethrow
}

' VB.NET sample:
Try
    ... ' Do some reading with the file
Catch ex As Exception
    file.Position = position ' Unwind on failure
Throw ex ' Rethrow
EndTry

```

## 3.12 资源清理

❌ 一定不要使用 `GC.Collect` 来进行强制垃圾回收。

### 3.12.1 Try-finally 块

✅ 一定请使用 `try-finally` 块来清理资源，`try-catch` 块来处理错误恢复。一定不要使用 `catch` 块来清理资源。

一般来说，清理的逻辑是回滚资源（特别是原生资源）分配。举例：

```

// C# sample:
FileStream stream = null;
try
{
    stream = new FileStream (...);
    ...
}
finally
{
    if (stream != null)
        stream.Close ();
}

' VB.NET sample:
Dim stream As FileStream = Nothing
Try
    stream = New FileStream (...)
    ...
Catch ex As Exception
    If (stream IsNot Nothing) Then
        stream.Close ()
    EndIf

```

```
EndTry
```

为了清理实现了 `IDisposable` 接口的对象，C# 和 VB.NET 提供了 `using` 语句来替代 `try-finally` 块。

```
// C# sample:
using (FileStream stream = new FileStream (...))
{
    ...
}

' VB.NET sample:
Using stream AsNew FileStream (...)
    ...
EndUsing
```

许多语言特性都会自动的为您写入 `try-finally` 块。例如 C#/VB 的 `using` 语句，C# 的 `lock` 语句，VB 的 `SyncLock` 语句，C# 的 `foreach` 语句以及 VB 的 `For Each` 语句。

### 3.12.2 基础 `Dispose` 模式

该模式的基础实现包括实现 `System.IDisposable` 接口，声明实现了所有资源清理逻辑的 `Dispose (bool)` 方法，该方法被 `Dispose` 方法和可选的终结器所共享。请注意，本章节并不讨论如何编写一个终结器。可终结类型是该简单模式的拓展，我们会在下个章节中讨论。如下展示了基础模式的简单实现：

```
// C# sample:
public class DisposableResourceHolder : IDisposable
{
    private bool disposed = false;
    private SafeHandle resource; // Handle to a resource

    public DisposableResourceHolder ()
    {
        this.resource = ... // Allocates the native resource
    }

    public void DoSomething ()
    {
        if (disposed)
            throw new ObjectDisposedException (...);

        // Now call some native methods using the resource
    }
}
```

```

        ...
    }

    public void Dispose ()
    {
        Dispose (true) ;
        GC.SuppressFinalize (this) ;
    }

    protected virtual void Dispose (bool disposing)
    {
        // Protect from being called multiple times.
        if (disposed) return;

        if (disposing)
        {
            // Clean up all managed resources.
            if (resource != null)
                resource.Dispose () ;
        }

        disposed = true;
    }
}

' VB.NET sample:
Public Class DisposableResourceHolder
    Implements IDisposable

    Private disposed As Boolean = False
    Private resource As SafeHandle ' Handle to a resource

    Public Sub New ()
        resource = ... ' Allocates the native resource
    EndSub

    Public Sub DoSomething ()
        If (disposed) Then
            Throw New ObjectDisposedException (...)
        EndIf

        ' Now call some native methods using the resource
        ...
    EndSub

    Public Sub Dispose () Implements IDisposable.Dispose
        Dispose (True)
    EndSub

```

```

        GC.SuppressFinalize (Me)
    EndSub

    ProtectedOverridableSub Dispose (ByVal disposing As Boolean)
    ' Protect from being called multiple times.
    If disposed ThenReturn

    If disposing Then
    ' Clean up all managed resources.
    If (resource IsNotNothing) Then
        resource.Dispose ()
    EndIf
    EndIf

    disposed = True
EndSub

EndClass

```

☑一定请为包含了可释放类型的类型实现基础 `Dispose` 模式。

☑一定请拓展基础 `Dispose` 模式来提供一个终结器。比如，为存储非托管内存的缓冲实现该模式。

☑您应该为即使类本身不包含非托管代码或可清理对象，但是其子类可能带有的类实现该基础 `Dispose` 模式。一个绝佳的例子便是 `System.IO.Stream` 类。虽然它是一个不带任何资源的抽象类，大多数子类却带有资源，所以应该为其实现该模式。

☑一定请声明一个 `protected virtual void Dispose (bool disposing)` 方法来集中所有释放非托管资源的逻辑。所有资源清理都应该在该方法中完成。用终结器和 `IDisposable.Dispose` 方法来调用该方法。如果从终结器内调用，则其参数为 `false`。它应该用于确保任何在终结中运行的代码不应该被其他可终结对象访问到。下一章节我们会详细介绍终结器的细节。

```

// C# sample:
protectedvirtualvoid Dispose (bool disposing)
{
    if (disposing)
    {
        // Clean up all managed resources.
    }
}

```

```

if (resource != null)
    resource.Dispose ();
}
}

' VB.NET sample:
ProtectedOverridableSub Dispose (ByVal disposing As Boolean)
' Protect from being called multiple times.
If disposed ThenReturn

If disposing Then
' Clean up all managed resources.
If (resource IsNotNothing) Then
    resource.Dispose ()
EndIf
EndIf

disposed = True
EndSub

```

☑一定请通过简单的调用 `Dispose (true)`，以及 `GC.SuppressFinalize (this)` 来实现 `IDisposable` 接口。仅当 `Dispose (true)` 成功执行完后才能调用 `SuppressFinalize`。

```

// C# sample:
publicvoid Dispose ()
{
    Dispose (true) ;
    GC.SuppressFinalize (this) ;
}

' VB.NET sample:
PublicSub Dispose () Implements IDisposable.Dispose
    Dispose (True)
    GC.SuppressFinalize (Me)
EndSub

```

☒一定不要将无参 `Dispose` 方法定义为虚函数。`Dispose (bool)` 方法应该被子类重写。

☒您不应该从 `Dispose (bool)` 内抛出异常，除非包含它的进程被破坏（内存泄露，不一致的共享状态，等等）等极端条件。用户不希望调用 `Dispose` 会引发异常。比如，考虑在 C#代码内手动写入 `try-finally` 块：

```

    TextReader tr = new StreamReader (File.OpenRead ("foo.txt")) ;
    try
    {
        // Do some stuff
    }
    finally
    {
        tr.Dispose () ;
        // More stuff
    }

```

如果 Dispose 可能引发异常，那么 finally 块的清理逻辑不会被执行。为了解决这一点，用户需要将所有对于 Dispose（在它们的 finally 块内!）的调用放入 try 块内，这将导致一个非常复杂的清理处理程序。如果执行 Dispose（bool disposing）方法，即使终结失败也不会抛出异常。如果在一个终结器环境内这样做会终止当前流程。

☑一定请在对象被终结之后，为任何不能再被使用的成员抛出一个 ObjectDisposedException 异常。

```

// C# sample:
public class DisposableResourceHolder : IDisposable
{
    private bool disposed = false;
    private SafeHandle resource; // Handle to a resource

    public void DoSomething ()
    {
        if (disposed)
            throw new ObjectDisposedException (...);

        // Now call some native methods using the resource
        ...
    }

    protected virtual void Dispose (bool disposing)
    {
        if (disposed) return;
        // Cleanup
        ...
        disposed = true;
    }
}

```

```

' VB.NET sample:
PublicClass DisposableResourceHolder
Implements IDisposable

Private disposed AsBoolean = False
Private resource As SafeHandle ' Handle to a resource

PublicSub DoSomething ()
If (disposed) Then
ThrowNew ObjectDisposedException (...)
EndIf

' Now call some native methods using the resource
...
EndSub

ProtectedOverridableSub Dispose (ByVal disposing AsBoolean)
' Protect from being called multiple times.
If disposed ThenReturn

' Cleanup
...

disposed = True
EndSub

EndClass

```

### 3.12.3 可终结类型

可终结类型是通过重写终结器并在 `Dispose (bool)` 中提供终结代码路径来拓展基础 `Dispose` 模式的类型。

如下代码是一个可终结类型的示例：

```

// C# sample:
publicclassComplexResourceHolder : IDisposable
{
    bool disposed = false;
    privateIntPtr buffer; // Unmanaged memory buffer
    privateSafeHandle resource; // Disposable handle to a resource

    public ComplexResourceHolder ()
    {
        this.buffer = ... // Allocates memory
        this.resource = ... // Allocates the resource
    }
}

```



```

public void DoSomething ()
{
    if (disposed)
        throw new ObjectDisposedException (...);

    // Now call some native methods using the resource
    ...
}

~ComplexResourceHolder ()
{
    Dispose (false);
}

public void Dispose ()
{
    Dispose (true);
    GC.SuppressFinalize (this);
}

protected virtual void Dispose (bool disposing)
{
    // Protect from being called multiple times.
    if (disposed) return;

    if (disposing)
    {
        // Clean up all managed resources.
        if (resource != null)
            resource.Dispose ();
    }

    // Clean up all native resources.
    ReleaseBuffer (buffer);

    disposed = true;
}

}

' VB.NET sample:
Public Class DisposableResourceHolder
    Implements IDisposable

    Private disposed As Boolean = False
    Private buffer As IntPtr ' Unmanaged memory buffer
    Private resource As SafeHandle ' Handle to a resource

```

```

PublicSubNew ()
    buffer = ... ' Allocates memory
    resource = ... ' Allocates the native resource
EndSub

PublicSub DoSomething ()
If (disposed) Then
ThrowNew ObjectDisposedException (...)
EndIf

' Now call some native methods using the resource
...
EndSub

ProtectedOverridesSub Finalize ()
    Dispose (False)
MyBase.Finalize ()
EndSub

PublicSub Dispose () Implements IDisposable.Dispose
    Dispose (True)
    GC.SuppressFinalize (Me)
EndSub

ProtectedOverridableSub Dispose (ByVal disposing AsBoolean)
' Protect from being called multiple times.
If disposed ThenReturn

If disposing Then
' Clean up all managed resources.
If (resource IsNotNothing) Then
    resource.Dispose ()
EndIf
EndIf

' Clean up all native resources.
ReleaseBuffer (Buffer)

disposed = True
EndSub

EndClass

```

☑**一定请**在类型应该为释放非托管资源负责，且自身没有终结器的情况下，将该类型定义为可终结的。

当实现终结器时，简单的调用 `Dispose (false)`，并将所有资源清理逻辑放入 `Dispose (bool disposing)` 方法。

```
// C# sample:
public class ComplexResourceHolder : IDisposable
{
    ...
    ~ComplexResourceHolder ()
    {
        Dispose (false);
    }

    protected virtual void Dispose (bool disposing)
    {
        ...
    }
}

' VB.NET sample:
Public Class DisposableResourceHolder
Implements IDisposable

    ...
    Protected Overrides Sub Finalize ()
        Dispose (False)
    MyBase.Finalize ()
    EndSub

    Protected Overridable Sub Dispose (ByVal disposing As Boolean)
        ...
    EndSub

EndClass
```

☑**一定请**谨慎的定义可终结类型。仔细考虑任何一个您需要终结器的情况。带有终结器的实例从性能和复杂性角度来说，都需付出不小的代价。

☑**一定请**为每一个可终结类型实现基础 `Dispose` 模式。该模式的细节请参考先前章节。这给予该类型的使用者以一种显式的方式去清理其拥有的资源。

☑您应该在终结器即使面临强制的应用程序域卸载或线程中止的情况也必须被执行时，创建并使用临界可终结对象（一个带有包含了 `CriticalFinalizerObject` 的类型层次的类型）。

☑一定请尽量使用基于 `SafeHandle` 或 `SafeHandleZeroOrMinusOneIsInvalid`（对于 Win32 资源句柄，其值如果为 0 或者 -1，则代表其为无效句柄）的资源封装器，而不是自己来编写终结器。这样，我们便无需终结器，封装器会为其资源清理负责。安全句柄实现了 `IDisposable` 接口，并继承自 `CriticalFinalizerObject`，所以即使面临强制的应用程序域卸载或线程中止，终结器的逻辑也会被执行。

```

///<summary>
/// Represents a wrapper class for a pipe handle.
///</summary>
[SecurityCritical (SecurityCriticalScope.Everything) ,
HostProtection (SecurityAction.LinkDemand, MayLeakOnAbort = true) ,
SecurityPermission (SecurityAction.LinkDemand, UnmanagedCode = true) ]
internalsealedclass SafePipeHandle : SafeHandleZeroOrMinusOneIsInvalid
{
    private SafePipeHandle ()
        : base (true)
    {
    }

    public SafePipeHandle (IntPtr preexistingHandle, bool ownsHandle)
        : base (ownsHandle)
    {
        base.SetHandle (preexistingHandle) ;
    }

    [ReliabilityContract (Consistency.WillNotCorruptState, Cer.Success) ,
DllImport ("kernel32.dll", CharSet = CharSet.Auto, SetLastError = true) ]
    [return: MarshalAs (UnmanagedType.Bool) ]
    private static extern bool CloseHandle (IntPtr handle) ;

    protected override bool ReleaseHandle ()
    {
        return CloseHandle (base.handle) ;
    }
}

///<summary>
/// Represents a wrapper class for a local memory pointer.

```

```

///</summary>
[SuppressUnmanagedCodeSecurity,
HostProtection (SecurityAction.LinkDemand, MayLeakOnAbort = true)]
internalsealedclassSafeLocalMemHandle : SafeHandleZeroOrMinusOneIsInvalid
{
    public SafeLocalMemHandle ()
        : base (true)
    {
    }

    public SafeLocalMemHandle (IntPtr preexistingHandle, bool ownsHandle)
        : base (ownsHandle)
    {
        base.SetHandle (preexistingHandle);
    }

    [ReliabilityContract (Consistency.WillNotCorruptState, Cer.Success),
DllImport ("kernel32.dll", CharSet = CharSet.Auto, SetLastError = true)]
    privatestaticexternIntPtr LocalFree (IntPtr hMem);

    protectedoverridebool ReleaseHandle ()
    {
        return (LocalFree (base.handle) == IntPtr.Zero);
    }
}

```

❌ **一定不要在终结器代码路径内访问任何可终结对象**，因为这样会有一个极大的风险：它们已经被终结了。例如，一个可终结对象 A，其拥有一个指向另一个可终结对象 B 的对象，在 A 的终结器内并不能很安心的使用 B，反之亦然。终结器是被随机调用的。

但是操作拆箱的值类型字段是可以接受的。

同时也请注意，在应用程序域卸载或进程退出的某些时间点上，存储于静态变量的对象会被回收。如果 Environment.HasShutdownStarted 返回 True，则访问引用了一个可终结对象的静态变量（或调用使用了存储于静态变量的值的静态函数）可能会不安全。

❌ **一定不要从终结器的逻辑内抛出异常**，除非是系统严重故障。如果从终结器内抛出异常，CLR 可能会停止整个进程来阻止其他终结器被执行，并阻止资源以一个受控制的方式释放。

### 3.12.4 重写 Dispose

如果您继承了实现 `IDisposable` 接口的基类，您必须也实现 `IDisposable` 接口。记得要调用您基类的 `Dispose (bool)`。

```
public class DisposableBase : IDisposable
{
    ~DisposableBase ()
    {
        Dispose (false) ;
    }

    public void Dispose ()
    {
        Dispose (true) ;
        GC.SuppressFinalize (this) ;
    }

    protected virtual void Dispose (bool disposing)
    {
        // ...
    }
}

public class DisposableSubclass : DisposableBase
{
    protected override void Dispose (bool disposing)
    {
        try
        {
            if (disposing)
            {
                // Clean up managed resources.
            }

            // Clean up native resources.
        }
        finally
        {
            base.Dispose (disposing) ;
        }
    }
}
```

## 3.13 交互操作

### 3.13.1 P/Invoke

☑一定请在编写P/Invoke签名时，查阅[P/Invoke交互操作助理](#)以及<http://pinvoke.net>。

☑您可以使用 `IntPtr` 来进行手动封送处理。虽然会牺牲易用性，类型安全和维护性，但是通过将参数和字段声明为 `IntPtr`，您可以提升性能表现。有时，通过 `Marshal` 类的方法来执行手动封送处理比依赖默认交互操作封送处理的性能更高。举例来说，如果有一个字符串的大数组需要被传递跨越交互操作边界，但是托管代码只需其中几个元素，那么您就可以将数组声明为 `IntPtr`，手动的访问所需的几个元素。

☒一定不要锁定短暂存在的对象。锁定短暂存在的对象会延长在P/Invoke调用时内存缓冲区的生存期。锁定会阻止垃圾回收器重新分配托管堆内对象内存，或者是托管委托的地址。然而，锁定长期存在的对象是可以接受的，因为其是在应用程序初始化期间创建的，相较于短暂存在对象，它们并不会被移动。长期锁定短暂存在的对象代价非常高昂，因为内存压缩经常发生在第0代的垃圾回收时，垃圾回收器不能重新分配被锁定的对象。这样会造成低效的内存管理，极大的降低性能表现。更多复制和锁定请参考<http://msdn.microsoft.com/en-us/library/23acw07k.aspx>。

☑一定请在 P/Invoke 签名中将 `CharSet` 设为 `CharSet.Auto`，`SetLastError` 设为 `true`。举例，

```
// C# sample:
[DllImport("kernel32.dll", CharSet = CharSet.Auto, SetLastError = true)]
public static extern SafeFileMappingHandle OpenFileMapping (
    FileMapAccess dwDesiredAccess, bool bInheritHandle, string lpName);

' VB.NET sample:
<DllImport("kernel32.dll", CharSet:=CharSet.Auto, SetLastError:=True) > _
Public Shared Function OpenFileMapping ( _
    ByVal dwDesiredAccess As FileMapAccess, _
    ByVal bInheritHandle As Boolean, _
    ByVal lpName As String) _
    As SafeFileMappingHandle
EndFunction
```

☑您应该将非托管资源封装进 `SafeHandle` 类。`SafeHandle` 类已经在[可终结类型](#)章节中进行了讨论。比如，文件映射句柄被封装成如下代码：

```

///<summary>
/// Represents a wrapper class for a file mapping handle.
///</summary>
[SuppressUnmanagedCodeSecurity,
HostProtection (SecurityAction.LinkDemand, MayLeakOnAbort = true) ]
internalsealedclass SafeFileMappingHandle : SafeHandleZeroOrMinusOneIsInvalid
{
    [SecurityPermission (SecurityAction.LinkDemand, UnmanagedCode = true) ]
    private SafeFileMappingHandle ()
        : base (true)
    {
    }

    [SecurityPermission (SecurityAction.LinkDemand, UnmanagedCode = true) ]
    public SafeFileMappingHandle (IntPtr handle, bool ownsHandle)
        : base (ownsHandle)
    {
        base.SetHandle (handle) ;
    }

    [ReliabilityContract (Consistency.WillNotCorruptState, Cer.Success) ,
    DllImport ("kernel32.dll", CharSet = CharSet.Auto, SetLastError = true) ]
    [return: MarshalAs (UnmanagedType.Bool) ]
    private static extern bool CloseHandle (IntPtr handle) ;

    protected override bool ReleaseHandle ()
    {
        return CloseHandle (base.handle) ;
    }
}

''' <summary>
''' Represents a wrapper class for a file mapping handle.
''' </summary>
''' <remarks></remarks>
<SuppressUnmanagedCodeSecurity () , _
HostProtection (SecurityAction.LinkDemand, MayLeakOnAbort:=True) > _
FriendNotInheritableClass SafeFileMappingHandle
Inherits SafeHandleZeroOrMinusOneIsInvalid

<SecurityPermission (SecurityAction.LinkDemand, UnmanagedCode:=True) > _
PrivateSubNew ()

```



```

MyBase.New (True)
EndSub

<SecurityPermission (SecurityAction.LinkDemand, UnmanagedCode:=True) > _
PublicSubNew (ByVal handle As IntPtr, ByVal ownsHandle AsBoolean)
MyBase.New (ownsHandle)
MyBase.SetHandle (handle)
EndSub

<ReliabilityContract (Consistency.WillNotCorruptState, Cer.Success) , _
    DllImport ("kernel32.dll", CharSet:=CharSet.Auto, SetLastError:=True) > _
PrivateSharedFunction CloseHandle (ByVal handle As IntPtr) _
As<MarshalAs (UnmanagedType.Bool) >Boolean
EndFunction

ProtectedOverridesFunction ReleaseHandle () AsBoolean
Return SafeFileMappingHandle.CloseHandle (MyBase.handle)
EndFunction

EndClass

```

☑您应该在调用会设置 Win32 最后错误代码的 P/Invoked 函数失败时，抛出 Win32Exception 异常。如果函数使用了非托管资源，请在 finally 块内释放资源。

```

// C# sample:
SafeFileMappingHandle hMapFile = null;
try
{
    // Try to open the named file mapping.
    hMapFile = NativeMethod.OpenFileMapping (
        FileMapAccess.FILE_MAP_READ,    // Read access
        false,                          // Do not inherit the name
        FULL_MAP_NAME                    // File mapping name
    );
    if (hMapFile.IsInvalid)
    {
        throw new Win32Exception ();
    }

    ...
}
finally
{
    if (hMapFile != null)
    {

```

```


// Close the file mapping object.
    hMapFile.Close ();
    hMapFile = null;
}
}

' VB.NET sample:
Dim hMapFile As SafeFileMappingHandle = Nothing
Try
    ' Try to open the named file mapping.
    hMapFile = NativeMethod.OpenFileMapping ( _
        FileMapAccess.FILE_MAP_READ, _
        False, _
        FULL_MAP_NAME)
    If (hMapFile.IsInvalid) Then
        ThrowNew Win32Exception
    EndIf


    ...
Finally
    If (Not hMapFile IsNothing) Then
        ' Close the file mapping object.
        hMapFile.Close ()
        hMapFile = Nothing
    EndIf
EndTry


```

### 3.13.2 COM交互操作

 **一定不要**以 GC.Collect 来进行强制垃圾回收以释放有性能要求的 API 内的 COM 对象。释放 COM 对象一个常见的方法是将 RCW 引用设置为 null，并调用 System.GC.Collect 以及

System.GC.WaitForPendingFinalizers。如果对性能有较高要求，我们并不推荐这样做，因为这样会引起垃圾回收运行太频繁。如果在服务器应用程序代码中使用该方法则极大的降低了性能和可拓展性。您应该让垃圾回收器自己决定何时执行垃圾回收。

 **您应该**使用 Marshal.FinalReleaseComObject 或者 Marshal.ReleaseComObject 来手动管理 RCW 的生存周期。相较于 GC.Collect 来进行强制垃圾回收，这样做具有较高的效率。

 **一定不要**进行跨套间调用。当您从托管应用程序调用一个 **COM** 对象时，请确保托管代码的套间类型和 **COM** 对象套间类型是相匹配的。通过使用匹配的套间，您可以避免跨套间调用时的线程切换。