《C#编码规范手册》

要作为一名合格的开发者，最基本的素质就是要做到编码规范，从小我们就接受教导“字如其人”，而写代码亦是如此，良好的代码风格，彰显了个人的工作素养。而良好的代码规范，能够帮助我们进行更好的团队协作，它能方便代码的交流和维护；不会影响编码的效率，不与大众习惯冲突；使代码更美观、阅读更方便；使代码的逻辑更清晰、更易于理解。

那为什么要整理这个规范呢？

最近社区群里有在讨论 C#的编码规范，而网络上也没有一些全面的规范文档，所以我就结合微软官方、Resharper 和 stylecop 的规范，以及曾经网路上的一些规范，整理一套全面的编码规范吧。

任何的标准都可能众口难调，整理的这套编码规范目的在于帮助开发者们提高开发效率，减少代码中的 bug，并增强代码的可维护性和可读性，同时，提升代码的执行效率等。万事开头难，接受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，这也是一劳永逸的事情。

规范目的

一个软件的生命周期中，80%的花费在于维护；

几乎没有任何一个软件，在其整个生命周期中，均由最初的开发人员来维护；

编码规范可以改善软件的可读性，可以让程序员尽快而彻底地理解新的代码。为了执行规范，每个软件开发人员必须一致遵守编码规范；

使用统一编码规范的主要原因，是使应用程序的结构和编码风格标准化，以便于阅读和理解这段代码；

好的编码约定可使源代码严谨、可读性强且意义清楚，与其它语言约定相一致，并且尽可能的直观。

高质量的代码往往具有如下特质 易懂性 – 代码必须易读且简单明确的。它们必须能展示出重点所在，且代码应该做到易于重用，不可包含多余代码，它们必须带有相应文档说明。

正确性 – 代码必须正确展示出其要告知使用者的重点。代码必须经过测试，且可以按照文档描述进行编译和正确运行。

一致性 – 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，不同代码之

间也应当保持一致的风格和设计，让使用者能够很轻松的结合使用它们。一致性能将我们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。

流行性 – 代码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。代码应当使用当下推荐的运行时库和 API 函数，以及推荐的项目和生成设置。

可靠性 – 代码必须符合当地法律，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是可以被撤销的。

安全性 – 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。

1.基本代码格式

1.1 缩进

一定不要使用制表符。不同的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。

配置 Visual Studio 文字编辑器，以空格代替制表符 。

1.2 花括号

花括号一定独占一行。关于花括号的格式问题，也是很多人争论的点，有些人习惯于 Java 的左花括号跟在圆括号后面，但微软官方推荐的是花括号应该独占一行，不与任何语句并列一行。

左花括号 “{” 放于关键字或方法名的下一行并与之对齐。左花括号 “{” 要与相应的右花括号 “}”对齐。

错误的示范：

```
public static void Main(string[] args) {

}
```

正确的示范：

```
public static void Main(string[] args)
```

```
{

}
```
if、while、do 语句后一定要使用{}，即使{}号中为空或只有一条语句。如：

```
if (somevalue == 1)
{
   somevalue = 2;
}
```
右花括号 “}” 后建议加一个注释以便于方便的找到与之相应的 {。如:

```
while(1)
{
   if(valid)
  {
  } // if valid
  else
  {
  } // not valid
} // end forever
```
例外的，类的自动属性花括号与代码合占一行：

```
public string Name{get;set;}
```
1.3 using 排序

引入的命名空间应该按照字母音序排列，这样做的目的在于方便在引入的多个命名空间中直接快速的找到命名空间。

1.4 换行

当表达式超出或即将超出显示器一行显示范围的时候，遵循以下规则进行换行：

1.在逗号后换行。

2.在操作符前换行。

规则 1 优先于规则 2。

当以上规则会导致代码混乱的时候自己采取更灵活的换行规则。

1.5 空行

空行是为了将逻辑上相关联的代码分块，以便提高代码的可阅读性。

在以下情况下使用两个空行：

当接口和类定义在同一文件中时，接口和类的定义之间。

当枚举和类定义在同一文件中时，枚举和类的定义之间。

当多个类定义在同一文件中时，类与类的定义之间。

在以下情况下使用一个空行：

方法与方法、属性与属性之间。

方法中变量声明与语句之间。

方法与方法之间。

方法中不同的逻辑块之间。

方法中的返回语句与其他的语句之间。

属性与方法、属性与字段、方法与字段之间。

语句控制块之后，如 if、for、while、switch。

注释与它注释的语句间不空行，但与其他的语句间空一行。

1.6 空格

在以下情况中要使用到空格：

关键字和左括符 “(” 应该用空格隔开。如：while (true)

注意在方法名和左括符 “(” 之间不要使用空格，这样有助于辨认代码中的方法调用与关键字。

多个参数用逗号隔开，每个逗号后都应加一个空格。

除了 . 之外，所有的二元操作符都应用空格与它们的操作数隔开。一元操作符、++及--与操作数间不需要空格。如：

```
a += c + d;
a = (a + b)/(c*d);
while (d++ == s++)
{
   n++;
}
PrintSize("size is " + size + "\n");
```
语句中的表达式之间用空格隔开。如：for (expr1; expr2; expr3)

1.7 文件定义

通常情况下，一个 cs 文件只能定义一个类、接口、枚举、结构体，特殊情况可将多个类定义在同一 cs 文件，如代码生成器生成的代码或紧密关联的两个 class。

其次类名应该与 cs 文件名保持一致，以便于通过文件名查找类名，比如 UserInfo 类应该在 UserInfo.cs 文件里。

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。

错误示范：

```
a = 1; b = 2;
```
正确示范：

```
a = 1;
b = 2;
```
2.命名规范

2.1 基本命名规范

一定要为各种类型，函数，变量，特性和数据结构选取有意义的命名。其命名应该能直接反映其作用。所谓自注释的代码就是好代码。

名称应该说明“什么”而不是“如何”。通过避免使用公开基础实现（它们会发生改变）的名称，可以保留简化复杂性的抽象层。例如，可以使用 GetNextStudent()，而不是 GetNextArrayElement()。

不应该在标识符名中使用不常见的或有歧义的缩短或缩略形式的词。比如，使用“GetTemperature” 而不是 “GetTemp”，Temp 到底是 Temperature 的缩写还是 Temporary 的缩写呢。对于公共类型或大家都知道的缩写，则可以使用缩略词，如：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。

不要使用计算机领域中未被普遍接受的缩写。

在适当的时候，使用众所周知的缩写替换冗长的词组名称。例如，用 UI 作为 User Interface 缩 写，用 OLAP 作为 On-line Analytical Processing 的缩写。

在使用缩写时，对于超过两个字符长度的缩写请使用 Pascal 命名法或驼峰命名法。例如使用 HtmlButton 或 HTMLButton；但是，应当大写仅有两个字符的缩写，如：System.IO，而不是 System.Io。

不要在标识符或参数名称中使用缩写。如果必须使用缩写，对于由多于两个字符所组成的缩写请使用驼峰命名法。

2.2 命名原则是：

选择正确名称时的困难可能表明需要进一步分析或定义项的目的。使名称足够长以便有一定的意义，并且足够短以避免冗长。唯一名称在编程上仅用于将各项区分开。表现力强的名称是为了帮助人们阅读；因此，提供人们可以理解的名称是有意义的。不过，请确保选择的名称符合适用语言的规则和标准。

2.3 推荐的命名法

Pascal 命名法：将标识符的首字母和后面连接的每个单词的首字母都大写。可以对三字符或更多字符的标识符使用 Pascal 命名法。例：BackColor

驼峰命名法：标识符的首字母小写，而每个后面连接的单词的首字母都大写。例：backColor

2.4 不推荐的命名法❌

匈牙利命名法：匈牙利命名法是一名匈牙利程序员发明的，这种命名法的基本原则是：变量名＝属性＋类型＋对象描述，如：m_bFlag： m 表示成员变量，b 表示布尔，合起来为：“某个类的成员变量，布尔型，是一个状态标志”。

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）。

2.5 几点是推荐的命名方法：

避免容易被主观解释的难懂的名称，如 AnalyzeThis()，或者属性名 Temp。这样的名称会导致多义性。

在类属性的名称中包含类名是多余的，如 User.UserName。而是应该使用 User.Name，“.”即中文的“的”的意思。

只要合适，在变量名的末尾或开头加计算限定符（Avg、Sum、Min、Max、Index）。

在变量名中使用互补对，如 min/max、begin/end 和 open/close。

布尔变量名通常应该包含 Is，这意味着 True/False 值，如 fileIsFound，若单词的意义本身已经包含是非的情况，可省略 Is，如 Exist。

在命名状态变量时，避免使用诸如 Flag 的术语。状态变量不同于布尔变量的地方是它可以具有两个以上的可能值。比如订单状态不应该是 OrderFlag，而是使用更具描述性的名称，OrderStatus。

即使对于可能仅出现在几个代码行中的生存期很短的变量，仍然需要使用有意义的名称。仅对于短循环索引使用单字母变量名，如 i 或 j。

定义方法名通常使用动词，接口通常使用形容词，类名、属性名、字段名、参数名通常使用名词。

2.6 常用的命名规范

1. 命名严禁使用拼音与英文混合的方式，更不允许直接使用中文。正确的英语拼写和语法可以让阅读者易于理解，避免歧义。注意：即使纯拼音命名的方式也要避免采用。正例：name / order / baidu / alibaba 等国际通用的名称可视为英文。 反例：zhekou(折扣)/Shuliang(数量)/ int 变量=1

2. 类名使用 Pascal 命名法，某些情况例外：DTO/UID 等模块功能缩写或接口定义 IInterface。 正例： UserDTO / XmlService / TFlowInfo /TTouchInfo /IUserService 反例：userDto / XMLService / tflowInfo / ttouchInfo

3. 方法名、参数名、成员变量、局部变量都统一使用驼峰命名法，必须遵从驼峰形式。正例： name / getUserInfo() / userId

4. 常量的命名使用 Pascal 命名法，单词力求语义表达要完整，不要嫌名字长。 正例：MaxStockCount 反例：Max_Count

5. 抽象类命名推荐使用 Base 结尾，异常类命名使用 Exception 结尾，测试类命名以它要测试的类的名称开始，以 Test 结尾。杜绝不规范的缩写，避免望文不知义。

反例：NotFoundException 缩写命名为 NotFoundEx

5. 为了达到代码自注释的目标，任何自定义编程元素在命名时，尽量使用完整的单词组合来表达其意思。 反例：int a 的随意命名方式

6. 如果模块、接口、类、方法使用了设计模式，在命名时需体现出具体模式。将设计模式体现在名字中，有利于阅读者快速理解架构设计理念。 正例：public class OrderFactory / public class LoginProxy

7. 接口和实现类的命名须遵循以下两点规则： （1）对于 Service 和 DAO 类，暴露出来的服务一定是接口。正例：CacheService 实现自 ICacheService 接口。 （2）如果形容能力的接口名称，取对应的形容词为接口名（一般为-able 结尾），正例：IDisposeable 接口。

8. 枚举类的成员名称使用 Pascal 命名法，枚举为特殊的类，成员均为常量，并为其显式指定枚举值，防止将来在中间插入枚举变量导致枚举值混乱，同时最好为没个枚举值打上 Description 标签。 正例：

```
public enum PaymentStatus : sbyte
{
    /// <summary>
    /// 进行中
    /// </summary>
    [Description("进行中")]
    Processing = 1,
    /// <summary>
    /// 成功
    /// </summary>
    [Description("成功")]
    Succeed = 2
}
```

2.7 各层命名规约：

A) Service/DAO 层方法命名规约

1）获取单个对象的方法用 Get 做前缀。

2）获取多个对象的方法用 List 做后缀，如：GetOrdersList。

3）获取统计值的方法用 Count 做后缀。

4）添加或更新的方法用 Save 或 Add。

5）删除的方法用 Remove/Delete。

6）修改的方法用 Update。

B) 领域模型命名规约

1） 实体对象：如 UserInfo，UserInfo 即为数据库表名。

2） 数据传输对象：xxxDTO，xxx 为业务领域相关的名称。

3） 展示对象：xxxViewModel，xxx 一般为网页名称。

2.8 命名空间和程序集命名

命名空间名称采用 Pascal 命名法，且首字符大写。命名空间名称尽量反映其内容所提供的整体功能，一般以“域名.项目名.模块名”的命名方式。如：Masuit.MyBlogs.Models。

其次命名空间应该与文件夹层级结构保持一致，比如在项目 Masuit.MyBlogs.Core 中，Masuit.MyBlogs.Core.Infrastructure.Services 则表示该命名空间位于项目的 Masuit.MyBlogs.Core/Infrastructure/Services 文件夹下。

2.9 常见的标识符的命名规范

标识符	规范	命名结构	示例
类，结构体	Pascal命名法	名词	`public class ComplexNumber {...}` `public struct ComplextStruct {...}`
命名空间	Pascal命名法	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。	`namespace Microsoft.Sample.Windows7`

枚举	Pascal命名法	名词
		☑ 一定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。
			`[Flags]`
			`public enum ConsoleModifiers`
			`{ Alt, Control }`

方法	Pascal命名法	动词或动词短语	`public void Print() {...}` `public void ProcessItem() {...}`

Public属性	Pascal命名法	名词或形容词
		☑ 一定要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” 。
		☑ 一定要以表肯定的短语来命名布尔属性，（CanSeek，而不是CantSeek）。当以 “Is” “Can” 或 “Has” 作布尔属性的前缀有意义时，也可以这样做。
			`public string CustomerName`
			`public ItemCollection Items`
			`public bool CanRead`

非Public属性	驼峰命名法	名词或形容词 ☑ 一定要使用'_' 前缀，保持代码一致性。	`private string _name;`

事件	Pascal命名法	动词或动词短语
		☑ 一定要用现在式或过去式来表明事件之前或是之后的概念。
		☒ 一定不要使用 “Before” 或者 “After” 前缀或后缀来指明事件的先后。
			`// 关闭窗口后引发的关闭事件。`
			`public event WindowClosed`
			
			`// 在关闭窗口之前引发的关闭事件。`
			`public event WindowClosing`

委托	Pascal命名法	☑ 一定要为用于事件的委托增加 ‘EventHandler’ 后缀。 ☑ 一定要为除了用于事件处理程序之外的委托增加 ‘Callback’ 后缀。 ☒ 一定不要为委托增加 “Delegate” 后缀。	`public delegate WindowClosedEventHandler`
接口	Pascal命名法，并带有 ‘I’ 前缀	名词	`public interface IDictionary`

常量	Pascal命名法用于Public常量;
	驼峰命名法用于Internal常量;
	只有1或2个字符的缩写需全部字符大写。
		名词	`public const string MessageText = "A";`
			`private const string messageText = "B";`
			`public const double PI = 3.14159...;`

参数，变量	驼峰命名法	名词	`int customerID;`
泛型参数	Pascal命名法，带有 ‘T’ 前缀	名词 ☑ 一定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。 ☑ 一定以T作为描述性类型参数的前缀。 ☑ 应该使用 T 作为单字符类型参数的名称。	`T,TItem, TPolicy`
资源	Pascal命名法	名词 ☑ 一定要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。 ☑ 一定要为命名资源使用字母数字字符和下划线。	`ArgumentExceptionInvalidName`

2.10 全局变量

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。

变量的声明和初始化

一定是在最小的，包含该局部变量的作用域块内声明它。一般情况，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。

```
void MyMethod()
{
   int int1 = 0;
   if (condition)
  {
      int int2 = 0;
       ...
  }
}
```

避免不同层次间的变量重名，如：

```
int count;
...
void MyMethod()
{
   if (condition)
  {
     int count = 0; // 避免
     ...
  }
  ...
}
```

一定要在声明变量时初始化它们的默认值，降低被抛空引用异常的概率。

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。

错误示范：

```
string str;
str="123";
```

正确示范：

```
string str="123";
```

一行只建议作一个声明，并按字母顺序排列。如：

```
int level; // 推荐
int size; // 推荐
int x, y; // 不推荐
```

2.11 常量定义

不允许任何未经预先定义的常量直接出现在代码中。 反例：string name =”tflow” + userId;

在 long 或者 float 赋值时，数值后使用大写 L 或者 F，特别是 L 不能写成小写的 l，小写的 l 容易跟数字 1 混淆，造成误解。

不要使用一个常量类维护所有常量，要按常量功能进行归类，分开维护。因为大而全的常量类，杂乱无章，使用查找才能定位到常量，不利于理解和维护。 正例：缓存相关的常量放在类 CacheConstants 下；系统配置常量放 ConfigConstants 下。

如果变量值仅在一个固定范围内变化用 enum 类型来定义。

一定要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。

```
public class Int32
{
   public const int MaxValue = 0x7fffffff;
   public const int MinValue = unchecked（（int）0x80000000）;
}
```

一定要为预定义的对象实例使用 public static 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，

```
public class ShellFolder
{
   public static readonly ShellFolder ProgramData = new ShellFolder（"ProgramData"）;
   public static readonly ShellFolder ProgramFiles = new ShellFolder（"ProgramData"）;
   ...
}
```

3.注释规范

3.1 所有的类、接口、方法、属性，都必须使用 VS 支持的文档注释，该类注释采用.Net 已定义好的 Xml 标签来标记，在声明接口、类、方法、属性、字段都应该使用该类注释，以便代码完成后直接生成代码文档，让别人更好的了解代码的实现和接口。在方法名或类名的上方，输入///即可生成文档注释块：

/// <summary>
/// 注释
/// </summary>
/// <returns></returns>
3.2 所有的抽象方法（包括接口中的方法）必须要用 vs 注释、除了返回值、参数、异常说明外，还必须指出该方法做什么事情，实现什么功能。

3.3 方法内部单行注释，在被注释语句上方另起一行，尽量使用//注释。避免使用/* */注释，注意与代码对齐。

3.4 所有的枚举类型字段必须要有注释，并说明每个枚举值的用途。

3.5 若团队没有要求使用某种特定的语言进行注释，使用本地化的语言进行代码注释即可。

3.6 代码修改的同时，注释也要进行相应的修改，尤其是参数、返回值、异常、核心逻辑等的修改。

3.7 谨慎直接删掉代码。在上方详细说明，而不是简单地注释掉。如果无用，则删除。代码被注释掉有两种可能性： 1）后续会恢复此段代码逻辑。 2）永久不用。前者如果没有备注信息，难以知晓注释动机。后者建议直接删掉（因为代码仓库保存了历史代码可供回滚）。

3.8 修改代码时，记得也要更新相应的注释。

3.9 避免杂乱的注释，如一整行星号做分割线。而是应该使用空白将注释同代码分开。

3.10 避免在块注释的周围加上印刷框。这样看起来可能很漂亮，但是难于维护。

3.11 在部署发布之前，移除所有临时或无关的注释，以避免在日后的维护工作中产生混乱。

3.12 如果需要用注释来解释复杂的代码节，需检查此代码以确定是否应该重写它。尽一切可能不注释难以理解的代码，而应该重写它。尽管一般不应该为了使代码更简单以便于人们使用而牺牲性能，但必须保持性能和可维护性之间的平衡。

3.13 在编写代码时就注释，因为以后很可能没有时间这样做。另外，如果有机会复查已编写的代码，在今天看来很明显的东西六周以后或许就不明显了。

3.14 避免多余的或不适当的注释，不应包含个人情绪内容，如幽默的不必要的备注（虽然博主经常在项目中这样干）。

3.15 在编写注释时使用完整的句子。注释应该阐明代码，而不应该增加多义性。

3.16 难以理解的代码一定要写注释!!

3.17 在整个应用程序中，使用具有一致的标点和结构的统一样式来构造注释。

3.18 用空白将注释同注释分隔符分开。在没有 IDE 的情况下通过其他文本编辑器查看注释时，这样做会使注释很明显且容易被找到。

3.19 代码修改变更记录不应使用注释标明修改日期和修改人，注释应只针对代码不记录版本，代码版本应该使用代码版本系统进行管理。

3.20 对于注释的要求：

第一、能够准确反应设计思想和代码逻辑；

第二、能够描述业务含义，使别的开发者能够迅速了解到代码背后的信息。完全没有注释的大段代码对于阅读者形同天书，注释是给自己看的，即使隔很长时间，也能清晰理解当时的思路；注释也是给继任者看的，使其能够快速接替自己的工作。

好的命名、代码结构是自注释的，注释力求精简准确、表达到位。避免出现注释的一个极端：过多过滥的注释，代码的逻辑一旦修改，修改注释是相当大的负担。注释是用来解释一段代码的设计意图的。一定不要让注释仅仅是一些无用的代码。

3.21 TODO 待办注释

todo 注释是 VS 支持的一种特殊注释，当注释打上 todo 前缀后，VS 会自动感知到该注释处有未完成的任务而自动加入到 VS 的任务列表里，供开发者快速定位。

4 类和接口的声明

4.1 类的声明

使用 Pascal 命名法。

用名词或名词短语命名类。

使用全称避免缩写，除非缩写已是一种公认的约定，如 URL、HTML 6 房东 不要使用类型前缀，如在类名称上对类使用 C 前缀。例如，使用类名称 FileStream，而不是 CFileStream。

不要使用下划线字符 “_”。

有时候需要提供以字母 I 开始的类名称，虽然该类不是接口。只要 I 是作为类名称组成部分的整个单词的第一个字母，这是适当的。例如，类名称 IdentityStore 是适当的。在适当的地方，使用复合单词命名派生的类。派生类名称的第二个部分应当是基类的名称。例如，ApplicationException 对于从名为 Exception 的类派生的类是适当的

名称，原因 ApplicationException 是一种 Exception。请在应用该规则时进行合理的判断。例如，Button 对于从 Control 派生的类是适当的名称。尽管按钮是一种控件，但是将 Control 作为类名称的一部分将使名称不必要地加长。

public class FileStreampublic class Button
public class String

4.2 构造函数

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。

在恰当的时候，从实例构造函数内抛出异常。

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。

4.3 字段的声明

不要使用是 public 或 protected 的实例字段。如果避免将字段直接公开给开发人员，可以更轻松地对类进行版本控制，原因是在维护二进制兼容性时字段不能被更改为属性。考虑为字段提供 get 和 set 属性访问器，而不是使它们成为公共的。 get 和 set 属性访问器中可执行代码的存在使得可以进行后续改进，如在使用属性或者得到属性更改通知时根据需要创建对象。下面的代码示例阐释带有 get 和 set 属性访问器的私有实例字段的正确使用。例：

```
public class Control: Component{
   private int _handle;
   public int Handle
  {
      get
     {
         return _handle;
     }
  }
}
```

private、protected 使用 驼峰命名法。

public 使用 Pascal 命名法。

拼写出字段名称中使用的所有单词。仅在开发人员一般都能理解时使用缩写。

```
class SampleClass
{
   private string _url;
   private string _destinationUrl;
}
```

不要对字段名使用匈牙利语表示法。好的名称描述语义，而非类型。

对预定义对象实例使用公共静态只读字段。如果存在对象的预定义实例，则将它们声明为 对象本身的公共静态只读字段。使用 Pascal 命名法，原因是字段是公共的。

```
public struct Color
{
   public static readonly Color Red = new Color(0x0000FF);

   public Color(int rgb)
   {
      // Insert code here.
   }

   public Color(byte r, byte g, byte b)
   {
      // Insert code here.
   }

   public byte RedValue
   {
      get
      {
         return Color;
      }
   }
}
```

4.4 静态字段

使用名词、名词短语或者名词的缩写命名静态字段。

使用 Pascal 命名法。

建议尽可能使用静态属性而不是公共静态字段。

4.5 参数的声明

使用描述性参数名称。参数名称应当具有足够的描述性，以便参数的名称及其类型可用于在大多数情况下确定它的含义。

对参数名称使用驼峰命名法。

使用描述参数的含义的名称，而不要使用描述参数的类型的名称。开发工具将提供有关参数的类型的有意义的信息。因此，通过描述意义，可以更好地使用参数的名称。

不要给参数名称加匈牙利语类型表示法的前缀，仅在适合使用它们的地方使用它们。

不要使用保留的参数。保留的参数是专用参数，如果需要，可以在未来的版本中公开它们。相反，如果在类库的未来版本中需要更多的数据，请为方法添加新的重载。

Type GetType(string typeName)
string Format(string format, object args)

4.6 方法的声明

使用动词或动词短语命名方法，使用 Pascal 命名法。

RemoveAll()
GetCharArray()
Invoke()

下列情形，需要进行参数校验： 1） 调用频次低的方法。

2） 执行时间开销很大的方法。此情形中，参数校验时间几乎可以忽略不计，但如果因为参数错误导致中间执行回退，或者错误，那得不偿失。

3） 需要极高稳定性和可用性的方法。

4） 对外提供的开放接口，不管是 RPC/API/HTTP 接口。

5） 敏感权限入口。

下列情形，不需要进行参数校验： 1） 极有可能被循环调用的方法。但在方法说明里必须注明外部参数检查要求。

2） 底层调用频度比较高的方法。毕竟是像纯净水过滤的最后一道，参数错误不太可能到底层才会暴露问题。一般 DAO 层与 Service 层都在同一个应用中，部署在同一台服务器中，所以 DAO 的参数校验，可以省略。

3） 被声明成 private 只会被自己代码所调用的方法，如果能够确定调用方法的代码传入参数已经做过检查或者肯定不会有问题，此时可以不校验参数。

4.7 异步方法

异步方法的命名一般以 Async 结尾，方法签名带有 async 标识，方法体内部有出现 await 关键字，且异步方法的返回值有三种：

1.没有任何返回值的 void

2.返回一个 Task 任务的 Task，可以获得该异步方法的执行状态

3.返回 Task 可以获得异步方法执行的结果和执行状态

如果你认为你的异步任务不需要知道它的执行状态（是否出现异常等）可以使用没有返回值的 void 签名（强烈建议不要在正式项目中使用 void 的异步方法）

```
public static async void FireAndForgetAsync()
{
   var myTask = Task.Run(() =>
   {
      for (int i = 0; i < 10; i++)
      {
         Console.WriteLine("DoWork : {0}", i);
         Thread.Sleep(500);
      }
   });
   await myTask;
}
```

如果你需要知道任务的执行状态则使用 Task 的签名

```
   static Task SayHello(string name)
{
   return Task.Run(() =>
   {
      Console.WriteLine("你好：{0}",name);
   });
}
static async Task SayHelloAsync(string name)
{
   await SayHello(name);
}
```

如果你需要获得异步方法的结果可以使用 Task 的签名

```
    static Task<int> SumArray(int[] arr)
{
    return Task.Run(() => arr.Sum());
}
static async Task<int> GetSumAsync(int[] arr)
{
    int result = await SumArray(arr);
    return result;
}
static async void GetTaskOfTResult()
{
    int[] arr = Enumerable.Range(1, 100).ToArray();
    Console.WriteLine("result={0}", GetSumAsync(arr).Result);
    int result = await GetSumAsync(arr);
    Console.WriteLine("result={0}", result);
}
```

除了上面的三个异步方法的例子，还可以使用 lambda 表达式来创建异步方法 只要在 lambda 表达式参数前加 async 在表达式内部使用 await 即可

```
    public static void TestAsyncLambda()
{
    Action act = async () =>
    {
        for (int i = 0; i < 10; i++)
        {
            Console.WriteLine("Do Work {0}", i);
            await Task.Delay(500);
        }
    };
    act();
}
```

4.8 属性 (property)的声明

使用名词或名词短语命名属性。

使用 Pascal 命名法。

考虑用与属性的基础类型相同的名称创建属性。例如，如果声明名为 Color 的属性，则属 性的类型同样应该是 Color。

```
public class SampleClass
{
   public Color BackColor{get;set;}
}
```
以下代码示例阐释提供其名称与类型相同的属性。

```
public enum Color
{
   Red,
   Green
}
public class Control
{
   public Color Color{get;set;}
}
```

4.9 事件的声明

对事件处理程序名称使用 EventHandler 后缀。

指定两个名为 sender 和 e 的参数。sender 参数表示引发事件的对象。sender 参数始终是 object 类型的，即使在可以使用更为特定的类型时也如此。与事件相关联的状态封装 在名为 e 的事件类的实例中。对 e 参数类型使用适当而特定的事件类。

用 EventArgs 后缀命名事件参数类。

考虑用动词命名事件。

使用动名词（动词的“ing”形式）创建表示事件前的概念的事件名称，用过去式表示事件后。例如，可以取消的 Close 事件应当具有 Closing 事件和 Closed 事件。不要使用 BeforeXxx/AfterXxx 命名模式。

不要在类型的事件声明上使用前缀或者后缀。例如，使用 Close，而不要使用 OnClose。

通常情况下，对于可以在派生类中重写的事件，应在类型上提供一个受保护的方法（称为 OnXxx）。此方法只应具有事件参数 e，因为发送方总是类型的实例。

```
public delegate void MouseEventHandler(object sender, MouseEventArgs e);
public class MouseEventArgs : EventArgs
{
```

```
  int x;
  int y;
  public MouseEventArgs(int x, int y)
  {
    this.x = x;
    this.y = y;
  }
  public int X
  {
    get
    {
      return x;
    }
  }
  public int Y
  {
    get
    {
      return y;
    }
  }
}
```

4.10 集合

集合是一组组合在一起的类似的类型化对象，如哈希表、查询、堆栈、字典和列表，集合的命名建议用复数。

4.11 成员方法重载

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。

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。

4.12 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。

我们更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）。

5 接口的声明

用名词或名词短语，或者描述行为的形容词命名接口。例：

接口名称 IComponent 使用描述性名词

接口名称 ICustomAttributeProvider 使用名词短语

接口名称 IPersistable 使用形容词。

使用 Pascal 命名法。

少用缩写。

给接口名称加上字母 I 前缀，以指示该类型为接口。在定义类/接口对（其中类是接口的标准实现）时使用相似的名称。两个名称的区别应该只是接口名称上有字母 I 前缀。

不要使用下划线字符“_”。

```
public interface IServiceProvider
public interface IFormatable
```

以下代码示例阐释如何定义 IComponent 接口及其标准实现 Component 类。

```
public interface IComponent
{
   // Implementation code goes here.
}
public class Component: IComponent
{
   // Implementation code goes here.
```

}

6. 关于枚举

枚举的每个值均代表了具体的含义，它是一个强类型化的常量参数;

一定要使用 enum 来定义枚举，而非使用常量类。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，会得到额外的编译器和反射支持。

错误示范：

```
public static class Color
{
   public const int Red = 0;
   public const int Green = 1;
   public const int Blue = 2;
}
```

正确示范：

```
public enum Color
{
   Red,
   Green,
   Blue
}
```

显式指定枚举值，防止将来在中间插入枚举变量导致枚举值混乱。

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。

正确示范

```
if （c > Color.Black || c < Color.White）
{
   throw new ArgumentOutOfRangeException（...）;
}
```

错误示范：

```
if （!Enum.IsDefined（typeof（Color）, c））
{
   throw new InvalidEnumArgumentException（...）;
}
```

7.字符串

一定不要使用‘+’操作符来拼接大量字符串。我们应该使用 StringBuilder 来实现拼接工作。然而，拼接少量的字符串时，推荐使用 C#6 的$插值字符串 。

正确示范：

```
StringBuilder sb = new StringBuilder();
for (int i = 0; i < 10; i++)
{
  sb.Append(i.ToString());
}
```

一定要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。

一定要在对文化未知的字符串做比较时，使用 StringComparison.Ordinal 和 StringComparison.OrdinalIgnoreCase 作为安全默认值，以提高性能表现。

一定要在向用户输出结果时，使用基于 StringComparison.CurrentCulture 的字符串操作。

一定要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。

一定要使用 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）） 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。

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

8. 数组和集合

我们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。不过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，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：

错误示范：

```
public static readonly char[] InvalidPathChars = { '\"', '<', '>', '|'};
```

调用者可以修改数组内的值：

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

我们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（）;
}
```

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


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

一定要将代表了读/写集合的属性或返回值声明为 Collection 或其子类，将代表了只读集合的属性或返回值声明为 ReadOnlyCollection 或其子类。

我们应该重新考虑对于 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。

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

我们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 和 IEnumerable 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）
{
```

```
    ...
}
```

关于 HashCode 和 Equals 的处理，遵循如下规则：

（1）只要重写 Equals，就必须重写 GetHashCode。

（2） 因为 HashSet 存储的是不重复的对象，依据 HashCode 和 Equals 进行判断，所以 HashSet 存储的对象必须重写这两个方法。

不要在 foreach 循环里进行元素的 Remove/Add 操作。Remove 元素需使用 Linq 方式，如果并发操作，需要对 List 对象加锁。

```
List<string> list = new List<string>();
list.add("1");
list.add("2");
List.RemoveAll(o=>o.id==0);
```

9.结构体

一定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。

一定要为值类型实现 IEquatable 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.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。

结构体 vs 类:

没有特殊情况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。

10.静态类

一定要合理使用静态类。静态类应该被用于框架内基于对象的核心支持辅助类。

11. 抽象类

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

一定要为抽象类定义一个 Protected，或 Internal 的构造函数。

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。

```
public abstract class Claim
{
  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。

```
public abstract class Claim
{
  internal Claim（）
  {
    ...
  }
}
```

12 第三方库的使用

一定不要引用不必要的库，或引用不必要的程序集。移除不必要的引用能够减少项目生成时间，最小化出错几率，减小项目生成后的体积， 并给读者留下一个良好的印象。

13 控制语句块

13.1 复合语句

复合语句是指包含"父语句{子语句;子语句;}"的语句，使用复合语句应遵循以下几点

子语句要缩进。

左花括号“{” 在复合语句父语句的下一行并与之对齐，单独成行。

即使只有一条子语句要不要省略花括号“ {}”。 如：

```
while(d += s++)
{
   n++;
}
```

13.2 return 语句

return 语句中不使用括号，除非它能使返回值更加清晰。如：

```
return;
return myDisk.size();
return (size>0 ? size : defaultSize);
```

13.3 条件语句

在 if/else/for/while/do 语句中必须使用大括号。即使只有一行代码，避免采用单行的编码方式。

```
if (condition)
{
   statements;
}

if (condition)
{
   statements;
}
else
{
   statements;
}

if (condition)
{
   statements;
}
else if (condition)
{
   statements;
}
```

```
else
{
   statements;
}
```
如果非得使用 if()...else if()...else...方式表达逻辑，避免后续代码维护困难，请勿超过 3 层。

除常用方法（如 getXxx/isXxx）等外，不要在条件判断中执行其它复杂的语句，将复杂逻辑判断的结果赋值给一个有意义的布尔变量名，以提高可读性。很多 if 语句内的逻辑相当复杂，阅读者需要分析条件表达式的最终结果，才能明确什么样的条件执行什么样的语句，那么，如果阅读者分析逻辑表达式错误呢？

正确示范：

```
bool existed = (file.open(fileName, "w") != null) && (...) || (...);
if (existed)
{
 ...
}
```
错误示范：

```
if ((file.open(fileName, "w") != null) && (...) || (...))
{
 ...
}
```
避免采用取反逻辑运算符。取反逻辑不利于快速理解，并且取反逻辑写法必然存在对应的正向逻辑写法。 正例：使用 if (x < 628) 来表达 x 小于 628。 反例：使用 if (!(x >= 628)) 来表达 x 小于 628。

13.4 for、foreach 语句

```
for (initialization; condition; update)
{
   statements;
}
```
空的 for 语句（所有的操作都在 initialization、condition 或 update 中实现）使用格式

```
for (initialization; condition; update);
```

foreach 语句使用格式

```
foreach (object obj in array)
{
   statements;
}
```

注意：

在循环过程中不要修改循环计数器。

对每个空循环体给出确认性注释。

循环体中的语句要考量性能，以下操作尽量移至循环体外处理：如定义对象、变量、获取数据库连接，进行不必要的 try-catch 操作（这个 try-catch 是否可以移至循环体外）。

13.5 while 语句

```
while (condition)
{
   statements;
}
```

空的 while 语句使用格式

```
while (condition);
```

13.6 do - while 语句

```
do
{
   statements;
} while (condition);
```

13.7 switch - case 语句

在一个 switch 块内，每个 case 要么通过 break/return 等来终止，要么注释说明程序将继续执行到哪一个 case 为止；在一个 switch 块内，都必须包含一个 default 语句并且放在最后，即使空代码。

```
switch (condition)
{
   case 1:
      statements;
      break;
   case 2:
      statements;
      break;
   default:
      statements;
      break;
}
```

注意：

语句 switch 中的每个 case 各占一行。

为所有 switch 语句提供 default 分支。

所有的非空 case 语句必须用 break; 语句结束。

13.8 try - catch 语句

```
try
{
   statements;
}
catch (ExceptionClass e)
{
   statements;
}
finally
{
   statements;
}
```

13.9 using 块语句

```
using (object)
{
   statements;
}
```

或使用 C#8 的 using 语法

using var ms=new MemoryStream();

14 DRY

避免出现重复的代码（Don’t Repeat Yourself），即 DRY 原则。

随意复制和粘贴代码，必然会导致代码的重复，在以后需要修改时，需要修改所有的副本，容易遗漏。必要时抽取共性方法，或者抽象公共类，甚至是组件化。

正确示范：一个类中有多个 public 方法，都需要进行数行相同的参数校验操作，这个时候请抽取：private bool CheckParam(DTO dto) {...}

15 异常处理

异常不要用来做流程控制，条件控制。异常设计的初衷是解决程序运行中的各种意外情况，且异常的处理效率比条件判断方式要低很多。

catch 时请分清稳定代码和非稳定代码，稳定代码指的是无论如何不会出错的代码。对于非稳定代码的 catch 尽可能进行区分异常类型，再做对应的异常处理。对大段代码进行 try-catch，使程序无法根据不同的异常做出正确的应激反应，也不利于定位问题，这是一种不负责任的表现。 正确示范：用户注册的场景中，如果用户输入非法字符，或用户名称已存在，或用户输入密码过于简单，在程序上作出分门别类的判断，并提示给用户。

捕获异常是为了处理它，不要捕获了却什么都不处理而抛弃之，如果不想处理它，请将该异常抛给它的调用者。最外层的业务使用者，必须处理异常，将其转化为用户可以理解的内容。

try 块放到了事务代码中，catch 异常后，如果需要回滚事务，一定要注意手动回滚事务。

finally 块必须对资源对象、流对象进行关闭，有异常也要做 try-catch。

方法的返回值可以为 null，不强制返回空集合，或者空对象等，必须添加注释充分说明什么情况下会返回 null 值。

对于公开给外部调用的 http/api 开放接口必须使用“错误码”；而应用内部推荐异常抛出；

一定要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。

一定要抛出最明确，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，

```
if （collection != null && !collection.IsReadOnly）
{
  collection.Add（additionalNumber）;
}
```

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。

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。

所有捕获到的异常都应该通过日志系统记录下来，方便后期排查系统故障。

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：

正确示范：

```
try
{
  …
}
catch（System.NullReferenceException exc）
{
  …
}
catch（System.ArgumentOutOfRangeException exc）
{
  …
}
catch（System.InvalidCastException exc）
{
  …
```

```
}
```
错误示范：

```
try
{
   …
}
catch （Exception ex）
{
   …
}
```
在捕获并重新抛出异常时，倾向使用 throw 。保持异常调用栈的最佳途径：

正确示范：

```
try
{
   ... // Do some reading with the file
}
catch
{
   file.Position = position; // Unwind on failure
   throw; // Rethrow
}
```
错误示范

```
try
{
   ... // Do some reading with the file
}
catch （Exception ex）
{
   file.Position = position; // Unwind on failure
   throw ex; // Rethrow
}
```

16.安全与性能

16.1 隐私保护和风险控制

1. 隶属于用户个人的页面或者功能必须进行权限控制校验。防止没有做水平权限校验就可随意访问、修改、删除别人的数据，比如查看他人的私信内容、修改他人的订单。
2. 用户敏感数据禁止直接展示，必须对展示数据进行脱敏。中国大陆个人手机号码显示为:156****1234，隐藏中间 4 位，防止隐私泄露。
3. 用户输入的 SQL 参数严格使用参数绑定或者 METADATA 字段值限定，防止 SQL 注入，禁止字符串拼接 SQL 访问数据库。
4. 用户请求传入的任何参数必须做有效性验证。忽略参数校验可能导致：

   1）page size 过大导致内存溢出

   2）恶意 order by 导致数据库慢查询

   3）任意重定向

   4）SQL 注入

   5）反序列化注入

   6）正则输入源串拒绝服务 ReDoS

   说明：代码用正则来验证客户端的输入，有些正则写法验证普通用户输入没有问题，但是如果攻击人员使用的是特殊构造的字符串来验证，有可能导致死循环的结果。

4. 禁止向 HTML 页面输出未经安全过滤或未正确转义的用户数据。
5. 表单、AJAX 提交必须执行 CSRF 安全验证。 说明：CSRF(Cross-site request forgery)跨站请求伪造是一类常见编程漏洞。对于存在 CSRF 漏洞的应用/网站，攻击者可以事先构造好 URL，只要受害者用户一访问，后台便在用户不知情的情况下对数据库中用户参数进行相应修改。
6. 在使用平台资源，譬如短信、邮件、电话、下单、支付，必须实现正确的防重放的机制，如数量限制、疲劳度控制、验证码校验，避免被滥刷而导致资损。如注册时发送

验证码到手机，如果没有限制次数和频率，那么可以利用此功能骚扰到其它用户，造成短信资源浪费。

7. 发贴、评论、发送即时消息等用户生成内容的场景必须实现防刷、文本内容违禁词过滤等风控策略。

16.2 依赖注入

对于实例化调用链较长的类，推荐使用依赖注入容器进行对象的实例化操作。

16.3 NullReferenceException

对于任何的 Get 操作，都应该做判空处理：

```
var user=users.FirstOrDefault(...);
if(user!=null)
{
   string name=user.Name；
}
```

或使用 C#6 的 null 值表达式

```
var user=users.FirstOrDefault(...);
string name=user?.Name；
```

16.4 资源释放

一定要使用 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：

```
FileStream stream = null;
try
{
   stream = new FileStream（...）;
   ...
}
finally
{
   if （stream != null)
   {
```

```
    stream.Close（）;
  }
}
```

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

```
using （FileStream stream = new FileStream（...））
{
  ...
}
```

或

```
using FileStream stream = new FileStream（...）;
```

许多语言特性都会自动的为您写入 try-finally 块。例如 C#的 using 语句，lock 语句。

16.5 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：

```
public class DisposableResourceHolder : IDisposable
{
  private bool disposed = false;
  private SafeHandle resource; // 处理一个资源

  public DisposableResourceHolder（）
  {
    this.resource = ... // 分配非托管资源
  }

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

    // 使用资源调用一些本机方法
    ...
```

```
  }

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

  protected virtual void Dispose（bool disposing）
  {
    // 防止被多次调用。
    if （disposed）
    {
      return;
    }

    if （disposing）
    {
      // 清理所有托管资源。
      if （resource != null）
      {
        resource.Dispose（）;
      }
    }

    disposed = true;
  }
}
```

一定要为包含了可释放类型的类型实现基础 Dispose 模式。

一定要给拓展基础 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。

我们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。一个绝佳的例子就是 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。

一定要声明一个 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。

protected virtual void Dispose（bool disposing）

```
{
  if （disposing）
  {
    // 清理所有托管资源。
    if （resource != null）
       {
        resource.Dispose（）;
    }
  }
}
```

一定要通过简单的方式调用 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 。

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

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。

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。

一定要在对象被终结之后，为任何不能再被使用的成员抛出一个

ObjectDisposedException 异常。

```
public class DisposableResourceHolder : IDisposable
{
  private bool disposed = false;
  private SafeHandle resource; // 处理一个资源

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

    // 使用资源调用一些本机方法
    ...
  }

  protected virtual void Dispose（bool disposing）
  {
    // 防止被多次调用。
    if （disposed）
    {
      return;
    }

    if （disposing）
    {
      // 清理所有托管资源。
      if （resource != null）
      {
        resource.Dispose（）;
      }
    }

    disposed = true;
  }
}
```

16.6 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：

```
public class ComplexResourceHolder : IDisposable
{
   bool disposed = false;
   private IntPtr buffer; // 非托管内存缓冲区
   private SafeHandle resource; // 处理非托管资源

   public ComplexResourceHolder（）
   {
      this.buffer = ... // 分配内存
      this.resource = ... // 分配资源
   }

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

      // 使用资源调用一些本机方法
      ...
   }

   ~ComplexResourceHolder（）
   {
      Dispose（false）;
   }

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

   protected virtual void Dispose（bool disposing）
   {
      // 防止被多次调用。
      if （disposed）
      {
        return;
      }

      if （disposing）
```

```
    {
      // 清理所有托管资源
      if （resource != null）
      {
        resource.Dispose（）;
      }
    }

    // 清理所有本机资源
    ReleaseBuffer（buffer）;
    disposed = true;
  }
}
```

一定要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。

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

  protected virtual void Dispose（bool disposing）
  {
    …
  }
}
```

一定要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。

一定请要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。

我们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） 。

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>
/// 表示管道句柄的包装器类。
/// </summary>
[SecurityCritical（SecurityCriticalScope.Everything）,
HostProtection（SecurityAction.LinkDemand, MayLeakOnAbort = true）,
SecurityPermission（SecurityAction.LinkDemand, UnmanagedCode = true）]
internal sealed class 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>
/// 表示本地内存指针的包装器类。
/// </summary>
[SuppressUnmanagedCodeSecurity,
HostProtection（SecurityAction.LinkDemand, MayLeakOnAbort = true）]
internal sealed class SafeLocalMemHandle : 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）]
  private static extern IntPtr LocalFree（IntPtr hMem）;

  protected override bool 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。

16.7 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）
      {
        // 清理托管资源
      }

      // 清理本机资源
    }
    finally
    {
      base.Dispose（disposing）;
    }
  }
}
```

7 其它约束

1. 一个方法只完成一个任务。单一职责原则，不要把多个任务组合到一个方法中，即使那些任务非常小。
2. 使用 C#的特有类型，而不是 System 命名空间中定义的别名类型。如字符串推荐使用 string 而非 String 类型。
3. 别在程序中使用固定数值，用常量代替。
4. 避免使用很多成员变量。声明局部变量，并传递给方法。不要在方法间共享成员变量。如果在几个方法间共享一个成员变量，那就很难知道是哪个方法在什么时候修改

了它的值。

5. 不在代码中使用具体的路径和驱动器名。 使用相对路径，并使路径可编程。
6. 应用程序启动时作些“自检”并确保所需文件和附件在指定的位置。必要时检查数据库连接。出现任何问题给用户一个友好的提示。
7. 如果需要的配置文件找不到，应用程序需能自己创建使用默认值的一份。
8. 如果在配置文件中发现错误值，应用程序要抛出错误，给出提示消息告诉用户正确值。
9. 在一个类中，字段定义全部统一放在 class 的头部、所有方法或属性的前面。
10. 在一个类中，所有的属性全部定义在一个属性块中。

结语 以上，只是规范，不是规定，所以不是强制要求一定要这样做，大家自取所需就好了。

如有遗漏，欢迎大家随时补充。

如有不合理之处，也接受大家的批评指正。