



.NET Framework 开发人员指南

类库开发的设计准则

类库开发的设计准则适用于扩展 .NET Framework 并与其交互的库开发。.NET Framework 设计准则的目标旨在通过提供一种独立于开发所用编程语言的统一编程模型，帮助库设计人员确保其用户获得 API 的一致性及易用性的好处。在开发扩展 .NET Framework 的类和组件时，强烈建议您遵循这些设计准则。不一致的库设计会对开发人员的工作效率造成不良影响并妨碍他们互相吸纳。

这些准则用于帮助类库设计人员理解如何在不同解决方案之间进行权衡。在特殊情况下，要实现好的库设计，可能会需要违反这些设计准则。这类情况应该很少见，所以您必须有充分的理由才能作出这种“违反”决定。

部分版权所有 2005 Microsoft Corporation。保留所有权利。

部分版权所有 Addison-Wesley Corporation。保留所有权利。

有关设计指南的更多信息，请参见 Krzysztof Cwalina 和 Brad Abrams 编著、Addison-Wesley 于 2005 年出版的“Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries”（《框架设计指南：可重用 .NET 库的约定、术语和模式》）。

■ 本节内容

[名称准则](#)

描述命名类库中的类型和成员的准则。

[类型设计准则](#)

描述使用静态和抽象类、接口、枚举和结构的准则。

[成员设计准则](#)

描述设计和使用属性、方法、构造函数、字段、事件和运算符的准则。此外，该节还描述了设计参数的最佳做法。

[扩展性设计](#)

描述设计可扩展库的准则。

[异常设计准则](#)

描述设计、引发和捕获异常的设计准则。

[使用准则](#)

描述使用数组和属性的准则以及实现相等运算符的准则。

■ 相关章节

[.NET Framework 类库参考](#)

描述构成 .NET Framework 的每一个公共类。

[异步编程设计模式](#)

描述用于设计和调用异步方法的 [IAsyncResult](#) 接口及事件驱动模式。



.NET Framework 开发人员指南

名称准则

[请参见](#)

对于组成类库的元素（包括程序集、命名空间、类型、成员和参数），命名准则提供如何为这些元素选择合适的标识符的准则。选择符合这些准则的标识符可以提高您的库的可用性，并使用户相信您的库将不需要学习一组新的规则。

为了提供一致的开发人员体验，公共公开的元素（如公共类和受保护的方法）必须遵守这些准则。然而，为在整个代码中保持一致性以及改进可维护性，应考虑在整个代码中始终使用这些约定。

部分版权所有 2005 Microsoft Corporation。保留所有权利。

部分版权所有 Addison-Wesley Corporation。保留所有权利。

有关设计指南的更多信息，请参见 Krzysztof Cwalina 和 Brad Abrams 编著、Addison-Wesley 于 2005 年出版的“**Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries**”（《框架设计指南：可重用 .NET 库的约定、术语和模式》）。

■ 本节内容

[大小写约定](#)

描述不同的大小写系统和每个系统的使用场合。

[通用命名约定](#)

描述选择明确的可读名称的一般规则。

[程序集和 DLL 的名称](#)

描述命名托管程序集的约定。

[命名空间的名称](#)

描述用于命名空间名称的约定以及如何最小化命名空间之间的冲突。

[类、结构和接口的名称](#)

描述命名类型时应遵循或避免的约定。

[类型成员的名称](#)

描述为方法、属性、字段和事件选择名称的最佳做法。

[参数名](#)

描述选择有含义的参数名称的最佳做法。

[资源的名称](#)

描述为可本地化的资源选择名称的最佳做法。

■ 请参见

概念

[类库开发的设计准则](#)



许多命名约定都与标识符的大小写有关。值得注意的是，公共语言运行库 (CLR) 支持区分大小写和不区分大小写的语言。本主题中描述的大小写约定可帮助开发人员理解和使用库。

大小写样式

下列术语描述了标识符的不同大小写形式。

Pascal 大小写

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 大小写。例如：

```
BackColor
```

大小写混合

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。例如：

```
backColor
```

大写

标识符中的所有字母都大写。例如：

```
IO
```

标识符的大小写规则

如果标识符由多个单词组成，请不要在各单词之间使用分隔符，如下划线（“_”）或连字符（“-”）等。而应使用大小写来指示每个单词的开头。

下列准则是用于标识符的通用规则。

- 对于由多个单词组成的所有公共成员、类型及命名空间名称，要使用 **Pascal** 大小写。
- 注意，这条规则不适用于实例字段。由于[成员设计准则](#)中详细说明的原因，不应使用公共实例字段。
- 对参数名称使用大小写混合。

下表汇总了标识符的大小写规则，并提供了不同类型标识符的示例。

标识符	大小写方式	示例
类	Pascal	AppDomain
枚举类型	Pascal	ErrorLevel
枚举值	Pascal	FatalError
事件	Pascal	ValueChanged
异常类	Pascal	WebException
只读的静态字段	Pascal	RedValue
接口	Pascal	IDisposable
方法	Pascal	ToString

命名空间	Pascal	System.Drawing
参数	Camel	typeName
属性	Pascal	BackColor

首字母缩写词的大小写规则

首字母缩写词是由术语或短语中各单词的首字母构成的单词。例如，HTML 是 Hypertext Markup Language 的首字母缩写。只有在公众广为认知和理解的情况下，才应在标识符中使用首字母缩写词。首字母缩写词不同于缩写词，因为缩写词是一个单词的缩写。例如，ID 是 identifier 的缩写。通常情况下，库名不应使用缩写词。

注意:

可在标识符中使用的两个缩写词是 ID 和 OK。在采用 Pascal 大小写格式的标识符中，这两个缩写词的大小写形式应分别为 Id 和 Ok。如果在采用大小写混合格式的标识符中将这两个缩写词用作首个单词，则它们的大小写形式应分别为 id 和 ok。

首字母缩写词的大小写取决于首字母缩写词的长度。所有首字母缩写词应至少包含两个字符。为了便于这些准则的实施，如果某一首字母缩写词恰好包含两个字符，则将其视为短型首字母缩写词。包含三个或三个以上字符的首字母缩写词为长型首字母缩写词。

下列准则为短型和长型首字母缩写词指定了正确的大小写规则。标识符大小写规则优先于首字母缩写词大小写规则。

两字符首字母缩写词的两个字符都要大写，但当首字母缩写词作为大小写混合格式的标识符的首个单词时例外。

例如，名为 DBRate 的属性是一个采用 Pascal 大小写格式的标识符，它使用短型首字母缩写词 (DB) 作为首个单词。又如，名为 ioChannel 的参数是一个采用大小写混合格式的标识符，它使用短型首字母缩写词 (IO) 作为首个单词。

包含三个或三个以上字符的首字母缩写词只有第一个字符大写，但当首字母缩写词作为大小写混合格式的标识符的首个单词时例外。

例如，名为 XmlWriter 的类是一个采用 Pascal 大小写格式的标识符，它使用长型首字母缩写词作为首个单词。又如，名为 htmlReader 的参数是一个采用大小写混合格式的标识符，它使用长型首字母缩写词作为首个单词。

如果任何首字母缩写词位于采用大小写混合格式的标识符开头，则无论该首字母缩写词的长度如何，都不大写其中的任何字符。

例如，名为 xmlStream 的参数是一个采用大小写混合格式的标识符，它使用长型首字母缩写词 (xml) 作为首个单词。又如，名为 dbServerName 的参数是一个采用大小写混合格式的标识符，它使用短型首字母缩写词 (db) 作为首个单词。

复合词和常用术语的大小写规则

不要将所谓的紧凑格式复合词中的每个单词都大写。这种复合词是指写作一个单词的复合词，如“endpoint”。

例如，hashtable 是一个紧凑格式的复合词，应将其视为一个单词并相应地确定大小写。如果采用 Pascal 大小写格式，则该复合词为 Hashtable；如果采用大小写混合格式，则该复合词为 hashtable。若要确定某个单词是否是紧凑格式的复合词，请查阅最新的词典。

下表列出了不是紧凑格式复合词的一些常用术语。术语先以 Pascal 大小写格式显示，后面的括号中的是其大小写混合格式。

- BitFlag (bitFlag)
- FileName (fileName)
- LogOff (logOff)
- LogOn (logOn)
- SignIn (signIn)
- SignOut (signOut)

```
I  UserName (userName)
I  WhiteSpace (whiteSpace)
```

▣ 区分大小写

大小写准则只是为了使标识符更易于阅读和辨认。不能将大小写规则用作避免库元素之间的命名冲突的手段。

不要假定所有编程语言都区分大小写。事实并非如此。不能仅凭大小写区分名称。

部分版权所有 2005 Microsoft Corporation。保留所有权利。

部分版权所有 Addison-Wesley Corporation。保留所有权利。

有关设计指南的更多信息，请参见 Krzysztof Cwalina 和 Brad Abrams 编著、Addison-Wesley 于 2005 年出版的“Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries”（《框架设计指南：可重用 .NET 库的约定、术语和模式》）。

▣ 请参见

概念

[类库开发的设计准则](#)

[名称准则](#)



.NET Framework 开发人员指南

通用命名约定

请参见

通用命名约定讨论的是如何为库元素选择最适当的名称。这些准则适用于所有标识符。后面各节讨论特定元素（如命名空间或属性）的命名。

选择名称

不要使用匈牙利表示法。

匈牙利表示法是在标识符中使用一个前缀对参数的某些元数据进行编码，如标识符的数据类型。

避免使用与常用编程语言的关键字冲突的标识符。

虽然符合 CLS 的语言必须提供将关键字用作普通字的方法，最佳做法不要求强制开发人员了解如何实现。对于大多数编程语言，语言参考文档都会提供语言所使用的关键字列表。下表提供了某些常用编程语言的参考文档的链接。

语言	链接
C#	C# 参考
C++	C++ Language Reference
Visual Basic	Visual Basic 参考

缩写和首字母缩写词

通常，不应使用缩写或首字母缩写词。这类名称的可读性较差。同样，要确定某个首字母缩写词是否已受到广泛认可也是很困难的。

有关缩写的大写规则，请参见[首字母缩写词的大小写规则](#)。

不要将缩写或缩略形式用作标识符名称的组成部分。

例如，使用 `OnClick` 而不要使用 `OnBtnClick`。

语言特定的名称

在标识符的语义含义仅限于其类型的极少数情况下，应使用一般公共语言运行库（CLR）类型名称，而不要使用语言特定的名称。

例如，将数据转换为 [Int16](#) 的方法应命名为 `ToInt16` 而不是 `ToShort`，因为 `Short` 是 `Int16` 的语言特定的类型名称。

下表显示的是公共编程语言和 CLR 的相应语言特定的类型名称。

C# 类型名称	Visual Basic 类型名称	JScript 类型名称	Visual C++ 类型名称	ILasm.exe 表示形式
sbyte	SByte	sByte	char	int8
byte	Byte	byte	unsigned char	unsigned int8
short	Short	short	short	int16
ushort	UInt16	ushort	unsigned short	unsigned int16
int	Integer	int	int	int32
uint	UInt32	uint	unsigned int	unsigned int32
long	Long	long	__int64	int64

ulong	UInt64	ulong	unsigned __int64	unsigned int64
float	Single	float	float	float32
double	Double	double	double	float64
bool	Boolean	boolean	bool	bool
char	Char	char	wchar_t	char
string	String	string	String	string
object	Object	object	Object	object

在标识符没有语义含义且参数的类型不重要的极少数情况下，应使用通用名称（如值或项），而不要重复类型名称。

部分版权所有 2005 Microsoft Corporation。保留所有权利。

部分版权所有 Addison-Wesley Corporation。保留所有权利。

有关设计指南的更多信息，请参见 Krzysztof Cwalina 和 Brad Abrams 编著、Addison-Wesley 于 2005 年出版的“Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries”（《框架设计指南：可重用 .NET 库的约定、术语和模式》）。

■ 请参见

概念

[类库开发的设计准则](#)

[名称准则](#)



.NET Framework 开发人员指南

程序集和 DLL 的名称

[请参见](#)

大多数情况下，程序集包含全部或部分可重用库，且它包含在单个动态链接库 (DLL) 中。一个程序集可拆分到多个 DLL 中，但这非常少见，在此准则中也没有说明。

程序集和 DLL 是库的物理组织，而命名空间是逻辑组织，其构成应与程序集的组织无关。命名空间可以且经常跨越多个程序集。

考虑按下面的模式命名 DLL：

`<Company>.<Component>.dll`

其中 **<Component>** 包含一个或多个以圆点分隔的子句。

例如，`Contoso.WebControls.dll`。

部分版权所有 2005 Microsoft Corporation。保留所有权利。

部分版权所有 Addison-Wesley Corporation。保留所有权利。

有关设计指南的更多信息，请参见 Krzysztof Cwalina 和 Brad Abrams 编著、Addison-Wesley 于 2005 年出版的“Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries”（《框架设计指南：可重用 .NET 库的约定、术语和模式》）。

请参见

概念

[类库开发的设计准则](#)

[名称准则](#)



.NET Framework 开发人员指南

命名空间的名称

[请参见](#)

为命名空间选择的名称应指示命名空间中的类型所提供的功能。例如，[System.Net.Sockets](#) 命名空间包含的类型允许开发人员使用套接字通过网络进行通信。

命名空间名称的一般格式如下：

```
<Company>.( <Product> | <Technology> ) [ . <Feature> ] [ . <Subnamespace> ]
```

例如，`Microsoft.WindowsMobile.DirectX`。

不要根据组织层次结构确定命名空间层次结构中的名称，因为公司的部门名称经过一段时间后可能会改变。

命名空间名称是长期使用的、不会更改的标识符。组织的不断发展和变化不应使命名空间名称过时。

命名空间和类型的名称冲突

如果选择的命名空间或类型的名称与现有名称冲突，则库的用户将不得不对受影响的项的引用进行限定。在大多数开发情况中，都不应出现这种问题。

本节提供的某些准则适用于下面的命名空间类别：

- 应用程序模型命名空间
- 基础结构命名空间
- 核心命名空间
- 技术命名空间组

应用程序模型中的命名空间提供特定于应用程序中的某个类的功能集。例如，[System.Windows.Forms](#) 命名空间中的类型提供编写 Windows 窗体客户端应用程序所需的功能。[System.Web](#) 命名空间中的类型支持编写基于 Web 的服务器应用程序。通常，在同一应用程序中不会使用不同应用程序模型中的命名空间，因此，这降低了名称冲突影响使用您的库的开发人员的可能性。

基础结构应用程序提供专门的支持，很少在程序代码中进行引用。例如，程序开发工具所使用的 ***.Designer** 命名空间中的类型。***.Permissions** 命名空间是基础结构命名空间的另一个示例。与基础结构命名空间中的类型的名称冲突不可能影响使用您的库的开发人员。

核心命名空间是 **System.*** 命名空间（不包括应用程序命名空间和基础结构命名空间）。[System](#) 和 [System.Text](#) 都是核心命名空间的示例。应尽可能避免与核心命名空间中的类型发生名称冲突。

属于特定技术的命名空间将具有相同的第一和第二级标识符 (`Company.technology.*`)。应避免在技术命名空间中出现名称冲突。

应用程序命名空间准则

不要在单个应用程序模型内为命名空间中的多个类型指定相同的名称。

例如，如果要编写 Windows 窗体应用程序开发人员要使用的特殊控件库，则不应引入名为 `Checkbox` 的类型，因为该应用程序模型已存在同名类型 (`CheckBox`)。

核心命名空间准则

不要指定会与核心命名空间中的任何类型发生冲突的类型名称。

例如，不要使用 `Directory` 作为类型名称，因为这会与 [Directory](#) 类型冲突。

技术命名空间准则

不要引入会导致技术命名空间的类型与应用程序模型命名空间中的类型发生冲突的类型名称（除非该技术不用于该应用程序模型）。

部分版权所有 2005 Microsoft Corporation。保留所有权利。

部分版权所有 Addison-Wesley Corporation。保留所有权利。

有关设计指南的更多信息, 请参见 Krzysztof Cwalina 和 Brad Abrams 编著、Addison-Wesley 于 2005 年出版的 “**Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries**” (《框架设计指南: 可重用 .NET 库的约定、术语和模式》)。

■ 请参见

概念

[类库开发的设计准则](#)

[名称准则](#)



.NET Framework 开发人员指南

类、结构和接口的名称

[请参见](#)

通常，类型名称应该是名词短语，其中名词是由类型表示的实体。例如，[Button](#)、[Stack](#) 和 [File](#) 都具有名称，用于标识由类型表示的实体。从开发人员的角度选择标识实体的名称：名称应反映使用场合。

下面的准则适用于如何选择类型名称。

不要为类名加前缀（如字母 C）。

接口不适用此规则，它应以字母 I 开头。

考虑在派生类的末尾使用基类名称。

例如，从 [Stream](#) 继承的 Framework 类型以 [Stream](#) 结尾，从 [Exception](#) 继承的类型以 [Exception](#) 结尾。

在定义类/接口对（其中类是接口的标准实现）时，一定要确保类和接口的名称除接口名称以字母 I 为前缀外，二者应完全相同。

例如，Framework 提供 [IAsyncResult](#) 接口和 [AsyncResult](#) 类。

泛型类型参数的名称

泛型是 .NET Framework 2.0 版的主要新功能。下面的准则适用于为泛型类型参数选择正确的名称。

用描述性名称为泛型类型参数命名，除非单个字母的名称已完全可以自我说明而无需描述性名称。

[IDictionary\(TKey, TValue\)](#) 是一个符合此准则的接口的示例。

常见类型的名称

下面的准则提供的命名约定有助于开发人员了解某些类的使用场合。准则中提及的从某个其他类型继承的类型，指的是所有的继承者，而不只是直接继承的类型。例如，“向从 [Exception](#) 继承的类型添加 [Exception](#) 后缀”这一准则意味着在继承层次结构中具有 [Exception](#) 的任何类型都应该使用以 [Exception](#) 结尾的名称。

每条这样的准则还用来保留指定的后缀；除非类型满足该准则表述的条件，否则不应使用该后缀。例如，如果类型不是从 [Exception](#) 直接或间接继承的，则类型名称不能以 [Exception](#) 结尾。

向自定义属性类添加 [Attribute](#) 后缀。

[ObsoleteAttribute](#) 和 [AttributeUsageAttribute](#) 是符合此准则的类型名称。

向在事件中使用的类型（如 C# 事件的返回类型）的名称添加 [EventHandler](#) 后缀。

[AssemblyLoadEventHandler](#) 是符合此准则的委托名称。

枚举的名称

不要在枚举值名称中使用前缀。例如，不要对 ADO 枚举使用 [ad](#) 之类的前缀，也不要对多格式文本枚举使用 [rtf](#) 之类的前缀，依此类推。

这还意味着不应在枚举值名称中包含枚举类型名称。下面的代码示例演示了不正确的枚举值命名。

Visual Basic

[复制代码](#)

```
Public Enum Teams
```

```
    TeamsAlpha  
    TeamsBeta  
    TeamsDelta
```

```
End Enum
```

C# [复制代码](#)

```
public enum Teams
{
    TeamsAlpha,
    TeamsBeta,
    TeamsDelta
}
```

对使用位域值的枚举（也称为标志枚举）使用复数名称。

部分版权所有 2005 Microsoft Corporation。保留所有权利。

部分版权所有 Addison-Wesley Corporation。保留所有权利。

有关设计指南的更多信息，请参见 Krzysztof Cwalina 和 Brad Abrams 编著、Addison-Wesley 于 2005 年出版的“Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries”（《框架设计指南：可重用 .NET 库的约定、术语和模式》）。

■ 请参见

概念

[类库开发的设计准则](#)

[名称准则](#)



.NET Framework 开发人员指南

类型成员的名称

[请参见](#)

类型包含以下几种成员：

- l 方法
- l 属性
- l 字段
- l 事件

本节中的准则有助于类库设计者为成员选择与 .NET Framework 一致的名称。

方法的名称

使用动词或动词短语作为方法的名称。

通常，方法对数据进行操作，因此，使用动词描述方法的操作可使开发人员更易于了解方法所执行的操作。定义由方法执行的操作时，应从开发人员的角度仔细选择明确的名称。不要选择描述方法如何执行其操作的动词，也就是说，不要使用实现细节作为方法名称。

属性的名称

使用名词、名词短语或形容词作为属性的名称。

名词短语或形容词适合于属性，因为属性保存数据。

不要使用与 Get 方法同名的属性。

例如，不要将一个属性命名为 `EmployeeRecord`，又将一个方法命名为 `GetEmployeeRecord`。开发人员会不知道使用哪个成员来完成其编程任务。

考虑为属性提供与其类型相同的名称。

如果某个属性已强类型化为某个枚举，则该属性可与该枚举同名。例如，如果有一个名为 `CacheLevel` 的枚举，则返回其中一个枚举值的属性也可以命名为 `CacheLevel`。

事件的名称

在事件处理程序签名中使用命名为“sender”和“e”的两个参数。

`sender` 参数的类型应为 [Object](#)，`e` 参数应是 [EventArgs](#) 的实例或继承自 [EventArgs](#) 的实例。

字段的名称

字段的命名准则适用于静态公共字段和静态受保护字段。不要定义公共实例字段或受保护实例字段。有关更多信息，请参见[字段设计](#)。

不要在字段名称中使用前缀。例如，不要使用 `g_` 或 `s_` 来区分静态字段和非静态字段。

部分版权所有 2005 Microsoft Corporation。保留所有权利。

部分版权所有 Addison-Wesley Corporation。保留所有权利。

有关设计指南的更多信息，请参见 Krzysztof Cwalina 和 Brad Abrams 编著、Addison-Wesley 于 2005 年出版的“Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries”（《框架设计指南：可重用 .NET 库的约定、术语和模式》）。

请参见

概念

[类库开发的设计准则](#)

[名称准则](#)



.NET Framework 开发人员指南

参数名

[请参见](#)

选择适当的参数名称可极大改善库的可用性。适当的参数名称应指示该参数会影响的数据或功能。

使用描述性参数名称。

在大多数情况下，参数名称及其类型应足以确定参数的用法。

考虑使用反映参数含义的名称而不是反映参数类型的名称。

在开发人员工具和文档中，参数的类型通常都是可见的。通过选择一个说明参数的用法或含义的名称，可以向开发人员提供有价值的信息，帮助他们找到任务所需的成员，也有助于向成员传递正确的数据。

部分版权所有 2005 Microsoft Corporation。保留所有权利。

部分版权所有 Addison-Wesley Corporation。保留所有权利。

有关设计指南的更多信息，请参见 Krzysztof Cwalina 和 Brad Abrams 编著、Addison-Wesley 于 2005 年出版的“**Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries**”（《框架设计指南：可重用 .NET 库的约定、术语和模式》）。

■ 请参见

概念

[类库开发的设计准则](#)

[名称准则](#)



.NET Framework 开发人员指南

资源的名称

[请参见](#)

本主题中准则适用于可本地化的资源，如错误信息和菜单文本。

使用点分隔符（“.”）以清晰的层次结构表示标识符。

例如，`Menus.FileMenu.Close.Text` 和 `Menus.FileMenu.Close.Color` 等名称符合此准则。

对异常消息资源使用下面的命名约定。资源标识符应由异常类型名称加上异常的短标识符构成，二者之间以点分隔。

例如，`ArgumentException.BadEnumValue` 符合此准则。

部分版权所有 2005 Microsoft Corporation。保留所有权利。

部分版权所有 Addison-Wesley Corporation。保留所有权利。

有关设计指南的更多信息，请参见 Krzysztof Cwalina 和 Brad Abrams 编著、Addison-Wesley 于 2005 年出版的“**Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries**”（《框架设计指南：可重用 .NET 库的约定、术语和模式》）。

■ 请参见

概念

[类库开发的设计准则](#)

[名称准则](#)



.NET Framework 开发人员指南

类型设计准则

本节提供的准则可帮助库设计人员从各种设计中做出选择，并正确实现类型。

部分版权所有 2005 Microsoft Corporation。保留所有权利。

部分版权所有 Addison-Wesley Corporation。保留所有权利。

有关设计指南的更多信息，请参见 Krzysztof Cwalina 和 Brad Abrams 编著、Addison-Wesley 于 2005 年出版的“Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries”（《框架设计指南：可重用 .NET 库的约定、术语和模式》）。

■ 本节内容

[类型和命名空间](#)

描述为便于识别而对类型和命名空间进行组织的准则。

[在类和结构之间选择](#)

描述在类和结构中实现功能的准则。

[在类和接口之间选择](#)

描述在类和接口中实现功能的准则。

[抽象类设计](#)

描述设计抽象类的准则。

[静态类设计](#)

描述设计静态类的准则。

[接口设计](#)

描述设计接口的准则。

[结构设计](#)

描述设计结构的准则。

[枚举设计](#)

描述设计简单和标志枚举的准则。

[嵌套类型](#)

描述设计嵌套类型的准则。

■ 相关章节

[.NET Framework 类库参考](#)

描述构成 .NET Framework 的每一个公共类。

[类库开发的设计准则](#)

描述类库开发的最佳做法。



.NET Framework 开发人员指南

类型和命名空间

[请参见](#)

下列准则可帮助您组织类型和命名空间，以便可以方便地查找和使用它们。

避免使用过多的命名空间。

应在同一方案中使用的类型尽可能放在同一命名空间中。用户在开发常见方案时，不应需要导入很多的命名空间。

避免将设计用于高级方案的类型与设计用于常见编程任务的类型放入同一命名空间中。

一般情况下，应将高级类型放入一般命名空间内的某个命名空间中，并将 `Advanced` 用作该命名空间的名称的最后一个标识符。例如，与 XML 序列化相关的常用类型位于 [System.Xml.Serialization](#) 命名空间中，而高级类型则位于 [System.Xml.Serialization.Advanced](#) 命名空间中。

定义类型时要指定类型的命名空间。

未指定命名空间的类型放在全局命名空间中。由于全局命名空间中的类型未在特定于功能的命名空间中，因此使用开发工具很难找到这些类型。此外，全局命名空间中的名称冲突问题也无法解决。有关更多信息，请参见[命名空间的名称](#)。

部分版权所有 2005 Microsoft Corporation。保留所有权利。

部分版权所有 Addison-Wesley Corporation。保留所有权利。

有关设计指南的更多信息，请参见 Krzysztof Cwalina 和 Brad Abrams 编著、Addison-Wesley 于 2005 年出版的“[Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries](#)”（《框架设计指南：可重用 .NET 库的约定、术语和模式》）。

请参见

概念

[类型设计准则](#)

[类库开发的设计准则](#)



.NET Framework 开发人员指南

标准子命名空间

[请参见](#)

子命名空间是命名空间中的命名空间。使用子命名空间可以限制开发人员在查找适用于常见编程任务的正确类型时所必须检查的类型数量。下列准则可帮助将某些特殊类型组织到主要功能命名空间下的已知命名空间中。

使用带有 `.Design` 后缀的命名空间来包含为基命名空间提供设计时功能的类型。

例如，与 Windows 窗体组件的设计时配置和行为相关的类型位于 [System.Windows.Forms.Design](#) 命名空间中。

使用带有 `.Interop` 后缀的命名空间来包含为基命名空间提供互操作功能的类型。

`.Interop` 子命名空间中的类型允许软件与 Microsoft 组件对象模型 (COM) 对象等旧版代码进行交互。

对主互操作程序集 (PIA) 中的所有代码使用带有 `.Interop` 后缀的命名空间。

有关主互操作程序集的更多信息，请参见[主互操作程序集](#)。

部分版权所有 2005 Microsoft Corporation。保留所有权利。

部分版权所有 Addison-Wesley Corporation。保留所有权利。

有关设计指南的更多信息，请参见 Krzysztof Cwalina 和 Brad Abrams 编著、Addison-Wesley 于 2005 年出版的“Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries”（《框架设计指南：可重用 .NET 库的约定、术语和模式》）。

请参见

概念

[类型设计准则](#)

[类库开发的设计准则](#)

[类型和命名空间](#)



.NET Framework 开发人员指南

在类和结构之间选择

[请参见](#)

类是引用类型，而结构是值类型。引用类型在堆中分配，内存管理由垃圾回收器处理。值类型在堆栈上或以内联方式分配，且在超出范围时释放。通常，值类型的分配和释放开销更小。然而，如果在要求大量的装箱和取消装箱操作的情况下使用，则值类型的表现就不如引用类型。有关更多信息，请参见[装箱和取消装箱 \(C# 编程指南\)](#)。

有关值类型和引用类型的更多信息，请参见[通用类型系统概述](#)。

不要定义结构，除非该类型具备以下所有特征：

- l 它在逻辑上表示单个值，与基元类型（整型、双精度型等）类似。
- l 它的实例大小小于 16 字节。
- l 它是不可变的。
- l 它将不必频繁被装箱。

如果这些条件中的一个或多个没有满足，则创建引用类型而不是结构。不遵守此准则会对性能产生负面影响。

部分版权所有 2005 Microsoft Corporation。保留所有权利。

部分版权所有 Addison-Wesley Corporation。保留所有权利。

有关设计指南的更多信息，请参见 Krzysztof Cwalina 和 Brad Abrams 编著、Addison-Wesley 于 2005 年出版的“Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries”（《框架设计指南：可重用 .NET 库的约定、术语和模式》）。

请参见

概念

[类型设计准则](#)

[类库开发的设计准则](#)



.NET Framework 开发人员指南

在类和接口之间选择

[请参见](#)

接口定义实施者必须提供的一组成员的签名。接口不能提供成员的实现细节。例如，[ICollection](#) 接口定义与使用集合相关的成员。实现该接口的每个类都必须提供这些成员的实现细节。类可以实现多个接口。

类定义每个成员的成员签名和实现细节。**Abstract**（在 **Visual Basic** 中为 **MustInherit**）类的行为在某方面与接口或普通类相同，即可以定义成员，可以提供实现细节，但并不要求一定这样做。如果抽象类不提供实现细节，从该抽象类继承的具体类就需要提供实现。

虽然抽象类和接口都支持将协定与实现分离开来，但接口不能指定以后版本中的新成员，而抽象类可以根据需要添加成员以支持更多功能。

优先考虑定义类，而不是接口。

在库的以后版本中，可以安全地向类添加新成员；而对于接口，则只有修改现有代码才能添加成员。

如果需要提供多态层次结构的值类型，则应定义接口。

值类型必须从 [ValueType](#) 继承，并且只能从 [ValueType](#) 继承，因此值类型不能使用类来分离协定和实现。这种情况下，如果值类型要求多态行为，则必须使用接口。

请考虑定义接口来达到类似于多重继承的效果。

如果一个类型必须实现多个协定，或者协定适用于多种类型，请使用接口。例如，[IDisposable](#) 是由许多不同情况下使用的类型实现的。如果要求从基类继承的类可处置，会使类层次结构很不灵活。[MemoryStream](#) 等应从其父类继承基于流的协定的类，不可能还是可处置的。

部分版权所有 2005 Microsoft Corporation。保留所有权利。

部分版权所有 Addison-Wesley Corporation。保留所有权利。

有关设计指南的更多信息，请参见 Krzysztof Cwalina 和 Brad Abrams 编著、Addison-Wesley 于 2005 年出版的“[Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries](#)”（《框架设计指南：可重用 .NET 库的约定、术语和模式》）。

☐ 请参见

概念

[类型设计准则](#)

[类库开发的设计准则](#)

[在类和结构之间选择](#)



.NET Framework 开发人员指南

抽象类设计

[请参见](#)

任何情况下，抽象类都不应进行实例化，因此，正确定义其构造函数就非常重要。确保抽象类功能的正确性和扩展性也很重要。下列准则有助于确保抽象类能够正确地设计并在实现后可以按预期方式工作。

不要在抽象类型中定义公共的或受保护的内部（在 Visual Basic 中为 Protected Friend）构造函数。

具有 **public** 或 **protected internal** 可见性的构造函数用于能进行实例化的类型。任何情况下抽象类型都不能实例化。

应在抽象类中定义一个受保护构造函数或内部构造函数。

如果在抽象类中定义一个受保护构造函数，则在创建派生类的实例时，基类可执行初始化任务。内部构造函数可防止抽象类被用作其他程序集中的类型的基类。

对于您提供的每个抽象类，至少应提供一个具体的继承类型。

这样有助于库设计者在设计抽象类时找到问题所在。同时意味着开发人员在进行高级别开发时，即使不了解抽象类和继承，也可以使用具体类而不必学习这些概念。例如，.NET Framework 提供抽象类 [WebRequest](#) 向统一资源标识符发送请求，使用 [WebResponse](#) 接收统一资源标识符的回复。Framework 提供了 [HttpWebRequest](#) 和 [HttpWebResponse](#) 类，分别作为这两个抽象类的几个具体实现之一，它们是相应抽象类的 HTTP 特定的实现。

部分版权所有 2005 Microsoft Corporation。保留所有权利。

部分版权所有 Addison-Wesley Corporation。保留所有权利。

有关设计指南的更多信息，请参见 Krzysztof Cwalina 和 Brad Abrams 编著、Addison-Wesley 于 2005 年出版的“Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries”（《框架设计指南：可重用 .NET 库的约定、术语和模式》）。

☐ 请参见

概念

[类型设计准则](#)

[类库开发的设计准则](#)

[在类和接口之间选择](#)



.NET Framework 开发人员指南

静态类设计

[请参见](#)

静态类只包含从 [Object](#) 继承的实例成员，也没有可调用的构造函数。下面的准则有助于确保正确设计静态类。

不要认为静态类可无所不包。

[Environment](#) 类使用静态类的方式值得学习。此类提供对当前用户环境的信息的访问。

不要声明或重写静态类中的实例成员。

如果某个类设计了实例成员，则该类不应标记为静态的。

如果编程语言没有对静态类的内置支持，则应将静态类声明为密封的和抽象的，并添加一个私有实例构造函数。

部分版权所有 2005 Microsoft Corporation。保留所有权利。

部分版权所有 Addison-Wesley Corporation。保留所有权利。

有关设计指南的更多信息，请参见 Krzysztof Cwalina 和 Brad Abrams 编著、Addison-Wesley 于 2005 年出版的“[Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries](#)”（《框架设计指南：可重用 .NET 库的约定、术语和模式》）。

■ 请参见

概念

[类型设计准则](#)

[类库开发的设计准则](#)



.NET Framework 开发人员指南

接口设计

[请参见](#)

接口定义实施者必须提供的一组成员的签名。接口不能提供成员的实现细节。例如，[ICollection](#) 接口定义与使用集合相关的成员。实现该接口的每个具体类都必须提供这些成员的实现细节。虽然类只能从单个类继承，但可以实现多个接口。下面的准则有助于确保正确设计接口。

如果一组包含某些值类型的类型需要支持某些常用功能，则必须定义接口。

值类型必须从 [ValueType](#) 继承。因此，抽象类不能用于指定值类型的协定；而必须改用接口。

避免使用标记接口（没有成员的接口）。

自定义属性提供了一种标记类型的方式。有关自定义属性的更多信息，请参见[编写自定义属性](#)。如果可以将属性检查推迟到执行代码时才进行，则首选自定义属性。如果需要进行编译时检查，则不能使用此准则。

请提供至少一种接口实现的类型。

这样有助于确保正确设计和顺利实现接口。[Int32](#) 类提供 [IComparable](#) 接口的一个实现。

对于定义的每个接口，请提供至少一个使用该接口的成员（例如，采用该接口作为参数的方法，或类型化为接口的属性）。

这是另一种有助于确保正确设计和顺利使用接口的机制。

不要向以前提供的接口添加成员。

添加新成员需要修改实现以前版本的接口的代码。这就是为什么在可能的情况下，通常首选使用类而不是接口的主要原因之一。有关更多信息，请参见[在类和接口之间选择](#)。

如果接口的交付定义要求更多成员，则可以实现新的接口和使用该接口的适当成员。

部分版权所有 2005 Microsoft Corporation。保留所有权利。

部分版权所有 Addison-Wesley Corporation。保留所有权利。

有关设计指南的更多信息，请参见 Krzysztof Cwalina 和 Brad Abrams 编著、Addison-Wesley 于 2005 年出版的“[Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries](#)”（《框架设计指南：可重用 .NET 库的约定、术语和模式》）。

■ 请参见

概念

[类型设计准则](#)

[类库开发的设计准则](#)

[在类和接口之间选择](#)



.NET Framework 开发人员指南

结构设计

[请参见](#)

结构是值类型。结构是在堆栈上或以内联方式分配的，当结构超出范围时将被释放。通常情况下，值类型的内存空间分配和释放的开销较小；但在需要大量装箱和取消装箱操作的方案中，值类型的执行性能较引用类型要差。有关更多信息，请参见[装箱和取消装箱 \(C# 编程指南\)](#)。

有关值类型和引用类型的更多信息，请参见[通用类型系统概述](#)。

不要为结构提供默认的构造函数。

如果某一结构定义了默认构造函数，则在创建该结构的数组时，公共语言运行库会自动对每个数组元素执行该默认构造函数。

有些编译器（如 C# 编译器）不允许结构拥有默认构造函数。

对值类型实现 `System.IEquatable`1`。

在确定两个值类型是否相等时，[IEquatable\(T\)](#) 要优于 [Equals](#)。通过使用接口，调用方可避免装箱和托管反射的不良性能影响。

确保所有实例数据均设置为零、false 或 null（根据需要）的状态是无效的。

如果遵循这一准则，新构造的值类型实例不会处于不可用的状态。例如，下面的结构的设计是错误的。参数化构造函数有意确保存在有效的状态，但在创建结构数组时不执行该构造函数。这意味着实例字段 `label` 初始化为 `null`（在 Visual Basic 中为 `Nothing`），这对于此结构的 `ToString` 实现是无效的。

Visual Basic

```
Public Structure BadStructure

    Private label As String

    Private width As Integer

    Private length As Integer

    Public Sub New(ByVal labelValue As String, ByVal widthValue As Integer, ByVal lengthValue As Integer)
        If ((labelValue = Nothing) _
            OrElse (labelValue.Length = 0)) Then
            Throw New ArgumentNullException("label")
        End If
        label = labelValue
        width = widthValue
        length = lengthValue
    End Sub

    Public Overrides Function ToString() As String
        ' Accessing label.Length throws a NullReferenceException
        ' when label is null.
        Return String.Format("Label length: {0} Label: {1} Width: {2} Length: {3}", label.Length, label, width, length)
    End Function
End Structure
```

C#

[复制代码](#)

```
public struct BadStructure
{
    string label;
    int width;
    int length;

    public BadStructure(string labelValue, int widthValue, int lengthValue)
    {
        if (labelValue == null || labelValue.Length == 0)
        {
            throw new ArgumentNullException("label");
        }
    }
}
```

```

    }
    label = labelValue;
    width = widthValue;
    length = lengthValue;
}

public override string ToString()
{
    // Accessing label.Length throws a NullReferenceException
    // when label is null.
    return String.Format("Label length: {0} Label: {1} Width: {2} Length: {3}",
        label.Length, label, width, length);
}
}

```

在下面的代码示例中，GoodStructure 的设计对 label 字段的状态未作任何假定。ToString 方法设计为处理 null 标签。

Visual Basic

```

Public Structure GoodStructure

    Private label As String

    Private width As Integer

    Private length As Integer

    Public Sub New(ByVal labelValue As String, ByVal widthValue As Integer, ByVal lengthValue As Integer)
        label = labelValue
        width = widthValue
        length = lengthValue
    End Sub

    Public Overrides Function ToString() As String
        ' Handle the case where label might be
        ' initialized to null;
        Dim formattedLabel As String = label
        Dim formattedLabelLength As Integer
        If (formattedLabel = Nothing) Then
            formattedLabel = "<no label value specified>"
            formattedLabelLength = 0
        Else
            formattedLabelLength = label.Length
        End If
        Return String.Format("Label Length: {0} Label: {1} Width: {2} Length: {3}", format
    End Function
End Structure

```

C#

 [复制代码](#)

```

public struct GoodStructure
{
    string label;
    int width;
    int length;

    public GoodStructure (string labelValue, int widthValue, int lengthValue)
    {
        label = labelValue;
        width = widthValue;
        length = lengthValue;
    }

    public override string ToString()
    {
        // Handle the case where label might be
        // initialized to null;
        string formattedLabel = label;
        int formattedLabelLength;
    }
}

```

```
        if (formattedLabel == null)
        {
            formattedLabel = "<no label value specified>";
            formattedLabelLength = 0;
        } else
        {
            formattedLabelLength = label.Length;
        }
        return String.Format("Label Length: {0} Label: {1} Width: {2} Length: {3}",
            formattedLabelLength, formattedLabel, width, length);
    }
}
```

不要显式扩展 `System.ValueType`。

有些编译器不允许扩展 [ValueType](#)。

部分版权所有 2005 Microsoft Corporation。保留所有权利。

部分版权所有 Addison-Wesley Corporation。保留所有权利。

有关设计指南的更多信息, 请参见 Krzysztof Cwalina 和 Brad Abrams 编著、Addison-Wesley 于 2005 年出版的“[Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries](#)”(《框架设计指南: 可重用 .NET 库的约定、术语和模式》)。

■ 请参见

概念

[类型设计准则](#)

[类库开发的设计准则](#)

[在类和结构之间选择](#)



.NET Framework 开发人员指南

枚举设计

[请参见](#)

枚举提供成组的常数值，它们有助于使成员成为强类型以及提高代码的可读性。枚举分为简单枚举和标志枚举两种。简单枚举包含的值不用于组合，也不用于按位比较。标志枚举应使用按位 **OR** 操作进行组合。标志枚举值的组合使用按位 **AND** 操作检查。

下列指南介绍了枚举设计的最佳做法。

一定要优选使用枚举而不是静态常量。

下面的代码示例演示了不正确的设计。

Visual Basic

[复制代码](#)

```
Public Class BadFurnishings

    Public Shared Table As Integer = 1
    Public Shared Chair As Integer = 2
    Public Shared Lamp As Integer = 3

End Class
```

C#

[复制代码](#)

```
public static class BadFurnishings
{
    public static int Table = 1;
    public static int Chair = 2;
    public static int Lamp = 3;
}
```

下面的代码示例演示应使用来代替静态常量的枚举。

Visual Basic

[复制代码](#)

```
Public Enum GoodFurnishings

    Table
    Chair
    Lamp

End Enum
```

C#

[复制代码](#)

```
public enum GoodFurnishings
{
    Table,
    Chair,
    Lamp
}
```

不要对开放集（如操作系统版本）使用枚举。

向已提供的枚举添加值会中断现有代码。有时可以接受这种做法，但不应在可能出现这种情况的场合设计枚举。

不要定义供将来使用的保留枚举值。

某些情况下，您可能认为为了向提供的枚举添加值，值得冒可能中断现有代码的风险。还可以定义使用其值的新的枚举和成员。

一定不要将 sentinel 值包括在枚举中。

Sentinel 值用于标识枚举中的值的边界。通常，sentinel 值用于范围检查，它不是一个有效的数据值。下面

的代码示例定义一个带有 `sentinel` 值的枚举。

Visual Basic

 [复制代码](#)

```
Public Enum Furniture
```

```
    Desk
    Chair
    Lamp
    Rug
    LastValue
```

```
End Enum
```

C#

 [复制代码](#)

```
public enum Furniture
{
    Desk,
    Chair,
    Lamp,
    Rug,
    LastValue    // The sentinel value.
}
```

一定要在简单枚举中提供一个零值。

如果可能，将此值命名为 `None`。如果 `None` 不适合，请将零值赋给最常用的值（默认值）。

考虑将 `System.Int32`（大多数编程语言的默认数据类型）用作枚举的基础数据类型，除非出现以下任何一种情况：

- 1 枚举是标志枚举，且您有 32 个以上的标志或者期望在将来有更多的标志。
- 1 基础类型需要与 [Int32](#) 不同，以便易于与期望不同大小的枚举的非托管代码进行互操作。
- 1 较小的基础类型可以节省大量空间。如果期望枚举主要用作控制流的参数，其大小就不太重要。如果出现下面的情况，大小节省可能会很重要：
 - 1 期望枚举被用作非常频繁地实例化的结构或类中的字段。
 - 1 期望用户创建枚举实例的大型数组或集合。
 - 1 预计要序列化大量枚举实例。

不要直接扩展 `System.Enum`。

一些编译器不允许扩展 [Enum](#)，除非间接地使用生成枚举的语言特定的关键字来进行扩展。

部分版权所有 2005 Microsoft Corporation。保留所有权利。

部分版权所有 Addison-Wesley Corporation。保留所有权利。

有关设计指南的更多信息，请参见 Krzysztof Cwalina 和 Brad Abrams 编著、Addison-Wesley 于 2005 年出版的“[Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries](#)”（《框架设计指南：可重用 .NET 库的约定、术语和模式》）。

☐ 请参见

概念

[设计标志枚举](#)

[向枚举添加值](#)

[类型设计准则](#)

[类库开发的设计准则](#)



.NET Framework 开发人员指南
设计标志枚举
[请参见](#)

标志枚举用于屏蔽位域及执行按位比较。在可以同时指定多个枚举值的情况下，它们是可使用的正确设计。例如，可以任意组合 [GenericUriParserOptions](#) 枚举值以配置一般的统一资源标识符 (URI) 分析器。

对标志枚举值使用 2 的幂，以便这些值可以使用按位“或”运算自由组合。

重要说明：

必须使用 2 的幂或 2 的幂组合，按位运算才能正常执行。

考虑为常用的标志组合提供特殊的枚举值。

组合标志枚举值是一种中级技能，对于实现常见方案的开发人员来说不是必需的。例如，[FileShare](#) 枚举包含 [ReadWrite](#) 值以指定可以打开共享文件进行读写。这向开发人员表明他们可以打开共享文件进行读写，并且他们无需了解如何将枚举值组合指定为单个值。

在某些组合值无效时，避免创建标志枚举。

此问题通常表明该枚举的含义不够精确。请考虑将该枚举划分为两个或多个枚举，使每个枚举都有一组更精确的值。例如，让我们来看看以下的未正确定义的枚举。

Visual Basic

[复制代码](#)

```
<Flags()> _
Public Enum PurchaseTypes

    SalePrice
    RegularPrice
    Book
    CompactDisk

End Enum
```

C#

[复制代码](#)

```
[Flags]
public enum PurchaseTypes
{
    SalePrice,
    RegularPrice,
    Book,
    CompactDisk
}
```

设计器意欲将该枚举与以下方法一起使用。

Visual Basic

```
Public Overloads Function FindPriceForItem(ByVal title As String, ByVal purchase As PurchaseTypes)
    Return 0
End Function
```

C#

[复制代码](#)

```
public float FindPriceForItem(string title, PurchaseTypes purchase)
```

在调用该方法时，`purchase` 参数只指定 `SalePrice` 或 `RegularPrice` 值中的一个值，并且只指定 `Book` 或 `CompactDisk` 值中的一个值。除非开发人员查阅了相关文档或有使用该方法的经验，否则将无法确定这种要求。最好是将这两类信息分开，将每类信息分别放在各自的枚举中，如下面的代码示例所示。

Visual Basic

[复制代码](#)

```
Public Enum ItemType
```

```
        Book
        CompactDisk

End Enum

Public Enum PriceType

    SalePrice
    RegularPrice

End Enum
```

C# [复制代码](#)

```
public enum ItemType
{
    Book,
    CompactDisk
}

public enum PriceType
{
    SalePrice,
    RegularPrice,
}
```

该方法的签名现在将相应地更改以反映这一重新设计，如下面的代码示例所示。

Visual Basic

```
Public Overloads Function FindPriceForItem(ByVal title As String, ByVal name As ItemType,
    Return 0
End Function
```

C# [复制代码](#)

```
public float FindPriceForItem(string title, ItemType name, PriceType price)
```

避免将标志枚举值设置为零，除非该值用于指示所有标志都被清除。应按照下一项准则中的说明正确地命名此类值。

注意，此项准则只适用于标志枚举。简单枚举可以也应当使用零值。

将标志的零值命名为 **None**。对于标志枚举，该值必须始终表示所有标志都被清除。

重要说明：

不要使用标志枚举中的零值指示任何其他状态。无法检查是否显式设置了零值标志（与未设置标志相对）。

部分版权所有 2005 Microsoft Corporation。保留所有权利。

部分版权所有 Addison-Wesley Corporation。保留所有权利。

有关设计指南的更多信息，请参见 Krzysztof Cwalina 和 Brad Abrams 编著、Addison-Wesley 于 2005 年出版的“Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries”（《框架设计指南：可重用 .NET 库的约定、术语和模式》）。

■ 请参见

概念

[枚举设计](#)

[类型设计准则](#)

[类库开发的设计准则](#)

[向枚举添加值](#)



.NET Framework 开发人员指南

向枚举添加值

[请参见](#)

下面的准则讨论如何为库用户引入可能影响重大的更改。如果向以前提供的枚举添加值，现有应用程序代码可能没有足够的能力妥善处理这些新值。

尽管有一些小的兼容风险，还是请考虑向枚举添加值。

此准则适于在提供多个库版本时使用。若要使用最少的代码向现有枚举添加值，可实现返回值全集的新成员，并使用 [ObsoleteAttribute](#) 属性标记现有成员（返回原始值集的成员）。如果不希望进行重大更改，则可以定义一个新枚举，以及新枚举的关联成员，并将现有的成员和枚举标记为已过时。

部分版权所有 2005 Microsoft Corporation。保留所有权利。

部分版权所有 Addison-Wesley Corporation。保留所有权利。

有关设计指南的更多信息，请参见 Krzysztof Cwalina 和 Brad Abrams 编著、Addison-Wesley 于 2005 年出版的“[Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries](#)”（《框架设计指南：可重用 .NET 库的约定、术语和模式》）。

■ 请参见

概念

[类型设计准则](#)

[类库开发的设计准则](#)

[枚举设计](#)

[设计标志枚举](#)



.NET Framework 开发人员指南

嵌套类型

[请参见](#)

嵌套类型是作为某其他类型的成员的类型。嵌套类型应与其声明类型紧密关联，并且不得用作通用类型。有些开发人员会将嵌套类型弄混淆，因此嵌套类型不应是公开可见的，除非不得不这样做。在设计完善的库中，开发人员几乎不需要使用嵌套类型实例化对象或声明变量。

在声明类型使用和创建嵌套类型实例时，嵌套类型很有用，但不在公共成员中公开嵌套类型的使用。

如果嵌套类型和其外部类型之间的关系需要成员可访问性语义，则要使用嵌套类型。

由于嵌套类型被视为是声明类型的成员，因此嵌套类型可以访问声明类型中的所有其他成员。

如果可能在声明类型的外部引用类型，则不要使用嵌套类型。

在常见方案中，不应要求对嵌套类型进行变量声明和对象实例化。例如，处理在某一类上定义的事件的事件处理程序委托不应嵌套在该类中。

如果需要由客户端代码实例化类型，则不要使用嵌套类型。如果某种类型具有公共构造函数，就可能不应进行嵌套。

理想情况下，嵌套类型仅由它的声明类型进行实例化和使用。如果嵌套类型具有公共构造函数，则表示该类型不单由其声明类型使用。通常情况下，嵌套类型不应针对其声明类型以外的类型执行任务。如某种类型具有更广泛的用途，就很可能不应进行嵌套。

不要将嵌套类型定义为接口的成员。许多语言不支持这样的构造。

部分版权所有 2005 Microsoft Corporation。保留所有权利。

部分版权所有 Addison-Wesley Corporation。保留所有权利。

有关设计指南的更多信息，请参见 Krzysztof Cwalina 和 Brad Abrams 编著、Addison-Wesley 于 2005 年出版的“Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries”（《框架设计指南：可重用 .NET 库的约定、术语和模式》）。

请参见

概念

[类型设计准则](#)

[类库开发的设计准则](#)

[类型和命名空间](#)



.NET Framework 开发人员指南

成员设计准则

类型和接口可以包含以下任何成员：

- | 方法
- | 属性
- | 构造函数
- | 事件
- | 字段

部分版权所有 2005 Microsoft Corporation。保留所有权利。

部分版权所有 Addison-Wesley Corporation。保留所有权利。

有关设计指南的更多信息，请参见 Krzysztof Cwalina 和 Brad Abrams 编著、Addison-Wesley 于 2005 年出版的“[Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries](#)”（《框架设计指南：可重用 .NET 库的约定、术语和模式》）。

■ 本节内容

[成员重载](#)

描述重载成员的准则。

[显式实现接口成员](#)

描述显式接口实现的准则。

[在属性和方法之间选择](#)

描述确定在哪些情况下将功能作为属性（与方法相对）实现的准则。

[属性设计](#)

描述实现属性的准则。

[构造函数设计](#)

描述实现构造函数的准则。

[事件设计](#)

描述实现事件的准则。

[字段设计](#)

描述定义字段的准则。

[运算符重载](#)

描述重载运算符的准则。

[转换运算符](#)

描述实现转换运算符的准则。

[参数设计](#)

描述定义参数的准则。

■ 相关章节

[.NET Framework 类库参考](#)

描述构成 .NET Framework 的每一个公共类。

[类库开发的设计准则](#)

描述类库开发的最佳做法。



.NET Framework 开发人员指南

成员重载

[请参见](#)

成员的签名包含成员的名称和参数列表。每个成员签名在类型中必须是唯一的。只要成员的参数列表不同，成员的名称可以相同。如果类型的两个或多个成员是同类成员（方法、属性、构造函数等），它们具有相同的名称和不同的参数列表，则称该同类成员进行了重载。例如，[Array](#) 类包含两个 [CopyTo](#) 方法。第一个方法采用一个数组和一个 [Int32](#) 值，第二个方法采用一个数组和一个 [Int64](#) 值。

注意:

如公共语言运行库规范所述，更改某个方法的返回类型并不能使该方法变得唯一。仅更改返回类型不能定义重载。

重载成员在同一功能上应有所不同。例如，某个类型具有两个 [CopyTo](#) 成员，其中第一个成员向数组复制数据，第二个成员向文件复制数据，这样是不正确的。对成员进行重载通常是为了提供带少量参数或不带参数且易于使用的重载。这些成员调用功能更强大、要求经验丰富才能正确使用重载。易于使用的重载通过向复杂重载传递默认值，支持常见的方案。例如，[File](#) 类提供 [Open](#) 方法的重载。简单重载 [Open](#) 采用文件路径和文件模式作为参数。它调用具有路径、文件模式、文件访问和文件共享参数的 [Open](#) 重载，并为文件访问和文件共享参数提供常用的默认值。如果开发人员不需要复杂重载所具有的灵活性，则不必了解文件访问和共享模型就可以打开文件。

为了便于维护和版本控制，简单重载应使用复杂重载来执行操作；基础功能不应在多个位置实现。

重载准则

下面的准则有助于确保正确设计重载成员。

尽量使用描述性参数名称指示简单重载所使用的默认值。

此准则尤其适用于 [Boolean](#) 参数。复杂重载的参数名称应通过描述相反的状态或操作来指示简单重载所提供的默认值。例如，[String](#) 类提供下面的重载：

Visual Basic

 [复制代码](#)

```
Overloads Public Shared Function Compare( _
    ByVal strA As String, _
    ByVal strB As String _
) As Integer

Overloads Public Shared Function Compare( _
    ByVal strA As String, _
    ByVal strB As String, _
    ByVal ignoreCase As Boolean _
) As Integer
public static int Compare(
    string strA,
    string strB
);

public static int Compare(
    string strA,
    string strB,
    bool ignoreCase
);
```

第二个重载提供一个名为 `ignoreCase` 的 [Boolean](#) 参数。即简单重载区分大小写，仅当要忽略大小写时，才需要使用复杂重载。默认值通常应为 `false`。

避免随意更改重载中的参数名称。如果某重载中的一个参数与另一重载的一个参数表示相同的输入，则这两个参数应具有同一名称。

例如，不要执行下面的操作：

Visual Basic

 [复制代码](#)

```
Public Sub Write(message as String, stream as FileStream)
End Sub
```

```
Public Sub Write(line as String, file as FileStream, closeStream as Boolean)
End Sub
```

C# [复制代码](#)

```
public void Write(string message, FileStream stream){}
public void Write(string line, FileStream file, bool closeStream){}
```

这些重载的正确定义如下所示:

Visual Basic [复制代码](#)

```
Public Sub Write(message as String, stream as FileStream)
End Sub
Public Sub Write(message as String, stream as FileStream, _
    closeStream as Boolean)
End Sub
```

C# [复制代码](#)

```
public void Write(string message, FileStream stream){}
public void Write(string message, FileStream stream, bool closeStream){}
```

保持重载成员中参数顺序的一致性。在所有重载中, 同名参数的位置应该相同。

例如, 不要执行下面的操作:

Visual Basic [复制代码](#)

```
Public Sub Write( message as String, stream as FileStream)
End Sub
Public Sub Write(stream as FileStream, message as String, _
    closeStream as Boolean)
End Sub
```

C# [复制代码](#)

```
public void Write(string message, FileStream stream){}
public void Write(FileStream stream, string message, bool closeStream){}
```

这些重载的正确定义如下所示:

Visual Basic [复制代码](#)

```
Public Sub Write(message as String, stream as FileStream)
End Sub
Public Sub Write(message as String, stream as FileStream, _
    closeStream as Boolean)
End Sub
```

C# [复制代码](#)

```
public void Write(string message, FileStream stream){}
public void Write(string message, FileStream stream, bool closeStream){}
```

此准则有两项约束:

- 1 如果重载采用变量参数列表, 则该列表必须是最后一个参数。
- 1 如果重载采用 **out** 参数, 按照约定, 这类参数应作为最后的参数。

如果需要具有扩展性, 则仅将最长的重载设为 **virtual** (在 **Visual Basic** 中为 **Overridable**)。较短的重载只是调用较长的重载。

下面的代码示例对此进行了演示。

Visual Basic [复制代码](#)

```
Public Sub Write(message as String, stream as FileStream)
```

```

        Me.Write(message, stream, false)
    End Sub

    Public Overridable Sub Write( _
        message as String, stream as FileStream, closeStream as Boolean)
        ' Do work here.
    End Sub

```

C#

 [复制代码](#)

```

public void Write(string message, FileStream stream)
{
    this.Write(message, stream, false);
}
public virtual void Write(string message, FileStream stream, bool closeStream)
{
    // Do work here.
}

```

不要对重载成员使用 **ref** 或 **out** 修饰符。

例如，不要执行下面的操作。

Visual Basic

 [复制代码](#)

```

Public Sub Write(message as String, count as Integer)

...

Public Sub Write(message as String, ByRef count as Integer)

```

C#

 [复制代码](#)

```

public void Write(string message, int count)

...

public void Write(string message, out int count)

```

通常，如果设计中出现这种情况，则很可能存在更深层的设计问题。考虑是否应该重命名某个成员，以便提供关于方法执行的确切操作的更多信息。

允许为可选参数传递 **null**（在 **Visual Basic** 中为 **Nothing**）。如果方法带有引用类型的可选参数，则允许传递 **null** 以指示应使用默认值。这样可不必在调用成员前检查 **null**。

例如，在下面的示例中，开发人员就不必检查 **null**。

Visual Basic

 [复制代码](#)

```

Public Sub CopyFile (source as FileInfo, _
    destination as DirectoryInfo, _
    newName as string)

    If newName Is Nothing
        InternalCopyFile(source, destination)
    Else
        InternalCopyFile(source, destination, newName)
    End If
End Sub

```

C#

 [复制代码](#)

```

public void CopyFile (FileInfo source, DirectoryInfo destination, string newName)
{
    if (newName == null)

```

```
{
    InternalCopyFile(source, destination);
}
else
{
    InternalCopyFile(source, destination, newName);
}
}
```

使用成员重载而不要用默认参数定义成员。默认参数不符合 CLS，不能在某些语言中使用。

下面的代码示例演示的方法设计是不正确的。

Visual Basic

[复制代码](#)

```
Public Sub Rotate (data as Matrix, Optional degrees as Integer = 180)
    ' Do rotation here
End Sub
```

此代码应重新设计为两个重载，由简单重载提供默认值。下面的代码示例演示的设计是正确的。

Visual Basic

[复制代码](#)

```
Overloads Public Sub Rotate (data as Matrix)
    Rotate(data, 180)
End Sub

Overloads Public Sub Rotate (data as Matrix, degrees as Integer)
    ' Do rotation here
End Sub
```

部分版权所有 2005 Microsoft Corporation。保留所有权利。

部分版权所有 Addison-Wesley Corporation。保留所有权利。

有关设计指南的更多信息，请参见 Krzysztof Cwalina 和 Brad Abrams 编著、Addison-Wesley 于 2005 年出版的“Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries”（《框架设计指南：可重用 .NET 库的约定、术语和模式》）。

■ 请参见

概念

[成员设计准则](#)

[类库开发的设计准则](#)



.NET Framework 开发人员指南

显式实现接口成员

[请参见](#)

接口是支持一些功能的协定。实现接口的类必须为接口中指定的成员提供实现细节。例如，[IEnumerator](#) 接口定义成员签名，必须实现成员签名才能支持对一组对象（如集合）进行枚举。若要实现 [IEnumerator](#)，类必须实现 [Current](#)、[MoveNext](#) 和 [Reset](#) 成员。

当接口成员由类显式实现时，只能通过使用对接口的引用来访问该成员。这将导致隐藏接口成员。显式实现接口成员的常见原因不仅是为了符合接口的协定，而且也是为了以某种方式改进它（例如，提供应用来代替接口的弱类型方法的强类型方法）。显式实现接口成员的另一个常见原因是存在不应由开发人员调用显式接口成员的时候。例如，[GetObjectData](#) 成员是最常显式实现的，因为它由序列化基础结构调用而不用从代码调用。

下列设计准则有助于确保您的库设计仅在需要时使用显式接口实现。

如果没有充分理由，应避免显式实现接口成员。

要理解显式实现需要具备很高深的专业知识。例如，很多开发人员不知道显式实现的成员是可以公共调用的，即使其签名是私有的也一样。由于这个原因，显式实现的成员不显示在公共可见的成员列表中。显式实现成员还会导致对值类型的不必要装箱。

如果成员只应通过接口调用，则考虑显式实现接口成员。

这主要包括支持 .NET Framework 基础结构（如数据绑定或序列化）的成员。例如，[IsReadOnly](#) 属性只应由数据绑定基础结构通过使用对 [ICollection\(T\)](#) 接口的引用来访问。由于满足此准则，[List\(T\)](#) 类显式实现该属性。

考虑显式实现接口成员以模拟变体（即，更改重写成员中的参数或返回类型）。

为了提供接口成员的强类型版本，通常会这么做。

考虑显式实现接口成员以隐藏一个成员并添加一个具有更好名称的等效成员。

这样可以有效地重命名成员。例如，[Stream](#) 显式实现 [Dispose](#) 并在相应的位置提供 [Close](#) 方法。

不要将显式成员用作安全边界。

显式实现成员不提供任何安全性。通过使用对接口的引用，这些成员都是可以公共调用的。

如果显式实现的成员的功能意在由派生类特殊化，则一定要提供具有相同功能的受保护虚拟成员。

不能重写显式实现的成员。如果在派生类中重新定义成员，则派生类不能调用基类实现。应通过使用与显式接口成员相同的名称或将 [Core](#) 附加到接口成员名称来命名受保护成员。

部分版权所有 2005 Microsoft Corporation。保留所有权利。

部分版权所有 Addison-Wesley Corporation。保留所有权利。

有关设计指南的更多信息，请参见 Krzysztof Cwalina 和 Brad Abrams 编著、Addison-Wesley 于 2005 年出版的“[Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries](#)”（《框架设计指南：可重用 .NET 库的约定、术语和模式》）。

请参见

概念

[成员设计准则](#)

[类库开发的设计准则](#)

[接口设计](#)



.NET Framework 开发人员指南

在属性和方法之间选择

[请参见](#)

通常，方法代表操作而属性代表数据。属性应像字段一样使用，这意味着属性不应进行复杂的计算，也不应产生副作用。在不违反下列准则的情况下，应考虑使用属性而不是方法，因为属性对于经验较少的开发人员更易于使用。

如果成员表示类型的逻辑属性 (Attribute)，请考虑使用属性 (Property)。

例如，[BorderStyle](#) 是一个属性 (Property)，因为边框样式是 [ListView](#) 的属性 (Attribute)。

如果属性值存储在进程内存中并且该属性只是用于提供对值的访问，则要使用属性而不是方法。

下面的代码示例阐释了这一准则。`EmployeeRecord` 类定义对私有字段提供访问的两个属性。在本主题的末尾显示了完整示例。

Visual Basic

 [复制代码](#)

```
Public Class EmployeeRecord

    Private employeeIdValue as Integer
    Private departmentValue as Integer

    Public Sub New()
    End Sub

    Public Sub New (id as Integer, departmentId as Integer)
        EmployeeId = id
        Department = departmentId
    End Sub

    Public Property Department as Integer
        Get
            Return departmentValue
        End Get
        Set
            departmentValue = value
        End Set
    End Property

    Public Property EmployeeId as Integer
        Get
            Return employeeIdValue
        End Get
        Set
            employeeIdValue = value
        End Set
    End Property

    Public Function Clone() as EmployeeRecord
        Return new EmployeeRecord(employeeIdValue, departmentValue)
    End Function
End Class
```

C#

 [复制代码](#)

```
public class EmployeeRecord
{
    private int employeeId;
    private int department;
    public EmployeeRecord()
    {
    }
    public EmployeeRecord (int id, int departmentId)
    {
        EmployeeId = id;
        Department = departmentId;
    }
    public int Department
```

```

    {
        get {return department;}
        set {department = value;}
    }
    public int EmployeeId
    {
        get {return employeeId;}
        set {employeeId = value;}
    }
    public EmployeeRecord Clone()
    {
        return new EmployeeRecord(employeeId, department);
    }
}

```

在下列情况下要使用方法而不是属性。

- | 操作比字段集慢数个数量级。即使考虑提供异步版本的操作来避免阻止线程，该操作也很可能因开销太大而不能使用属性。特别是，访问网络或文件系统（一次性初始化除外）的操作最可能是方法，而不是属性。
- | 操作是转换，如 `Object.ToString` method。
- | 操作在每次调用时都返回不同的结果，即使参数不发生更改也是如此。例如，[NewGuid](#) 方法在每次调用时都返回不同的值。
- | 操作具有很大的显而易见的副作用。注意，一般不将填充内部缓存视为是显而易见的副作用。
- | 操作返回内部状态的副本（这不包括在堆栈上返回的值类型对象的副本）。
- | 操作返回一个数组。

如果操作返回一个数组，应使用方法，原因是：要保留内部数组，必须返回数组的深层副本而不是对属性所使用的数组的引用。这一事实加之开发人员将属性视同字段一样使用的事实，可能会导致代码效率十分低下。在下面的代码示例中阐释了这一点，该示例使用属性返回一个数组。在本主题的末尾显示了完整示例。

Visual Basic

 [复制代码](#)

```

Public Class EmployeeData

    Dim data as EmployeeRecord()
    Public Sub New(data as EmployeeRecord())
        Me.data = data
    End Sub
    Public ReadOnly Property Employees as EmployeeRecord()
        Get
            Dim newData as EmployeeRecord() = CopyEmployeeRecords()
            Return newData
        End Get
    End Property

    Private Function CopyEmployeeRecords() as EmployeeRecord()
        Dim newData(UBound(data)) as EmployeeRecord
        For i as Integer = 0 To UBound(data)
            newData(i) = data(i).Clone()
        Next i
        Console.WriteLine ("EmployeeData: cloned employee data.")
        Return newData
    End Function
End Class

```

C#

 [复制代码](#)

```

public class EmployeeData
{
    EmployeeRecord[] data;
    public EmployeeData(EmployeeRecord[] data)
    {
        this.data = data;
    }
    public EmployeeRecord[] Employees
    {

```

```

        get
        {
            EmployeeRecord[] newData = CopyEmployeeRecords();
            return newData;
        }
    }
    EmployeeRecord[] CopyEmployeeRecords()
    {
        EmployeeRecord[] newData = new EmployeeRecord[data.Length];
        for(int i = 0; i < data.Length; i++)
        {
            newData[i] = data[i].Clone();
        }
        Console.WriteLine ("EmployeeData: cloned employee data.");
        return newData;
    }
}

```

使用此类的开发人员假定属性不会比字段访问的开销大，因此根据这一假定编写了应用程序代码，如下面的代码示例所示。

Visual Basic

[复制代码](#)

```

Public Class RecordChecker
    Public Shared Function FindEmployees( _
        dataSource as EmployeeData, _
        department as Integer) as Collection(Of Integer)

        Dim storage as Collection(Of Integer) = new Collection(Of Integer)()
        Console.WriteLine("Record checker: beginning search.")
        For i as Integer = 0 To UBound(dataSource.Employees)
            If dataSource.Employees(i).Department = department
                Console.WriteLine("Record checker: found match at {0}.", i)
                storage.Add(dataSource.Employees(i).EmployeeId)
                Console.WriteLine("Record checker: stored match at {0}.", i)
            Else
                Console.WriteLine("Record checker: no match at {0}.", i)
            End If
        Next i
        Return storage
    End Function
End Class

```

C#

[复制代码](#)

```

public class RecordChecker
{
    public static Collection<int> FindEmployees(EmployeeData dataSource,
        int department)
    {
        Collection<int> storage = new Collection<int>();
        Console.WriteLine("Record checker: beginning search.");
        for (int i = 0; i < dataSource.Employees.Length; i++)
        {
            if (dataSource.Employees[i].Department == department)
            {
                Console.WriteLine("Record checker: found match at {0}.", i);
                storage.Add(dataSource.Employees[i].EmployeeId);
                Console.WriteLine("Record checker: stored match at {0}.", i);
            }
            else
            {
                Console.WriteLine("Record checker: no match at {0}.", i);
            }
        }
        return storage;
    }
}

```

注意，在每次循环迭代中都会访问 `Employees` 属性，在部门匹配时也会访问该属性。每次访问该属性时，

就会创建一个 **employees** 数组的副本，该副本在短暂的使用过后就需要进行垃圾回收。通过将 **Employees** 实现为一个方法，可以向开发人员表明：与访问字段相比，该操作在计算上的开销较大。开发人员更愿意调用一次方法，并将方法调用的结果进行缓存以便进行处理。

示例

下面的代码示例演示一个假定属性访问在计算上的开销不大的完整应用程序。**EmployeeData** 类不适当地定义了一个返回数组副本的属性。

Visual Basic

[复制代码](#)

```
Imports System
Imports System.Collections.ObjectModel

Namespace Examples.DesignGuidelines.Properties
    Public Class EmployeeRecord

        Private employeeIdValue as Integer
        Private departmentValue as Integer

        Public Sub New()
        End Sub

        Public Sub New (id as Integer, departmentId as Integer)
            EmployeeId = id
            Department = departmentId
        End Sub

        Public Property Department as Integer
            Get
                Return departmentValue
            End Get
            Set
                departmentValue = value
            End Set
        End Property

        Public Property EmployeeId as Integer
            Get
                Return employeeIdValue
            End Get
            Set
                employeeIdValue = value
            End Set
        End Property

        Public Function Clone() as EmployeeRecord
            Return new EmployeeRecord(employeeIdValue, departmentValue)
        End Function
    End Class

    Public Class EmployeeData

        Dim data as EmployeeRecord()
        Public Sub New(data as EmployeeRecord())
            Me.data = data
        End Sub
        Public ReadOnly Property Employees as EmployeeRecord()
            Get
                Dim newData as EmployeeRecord() = CopyEmployeeRecords()
                Return newData
            End Get
        End Property

        Private Function CopyEmployeeRecords() as EmployeeRecord()
            Dim newData(UBound(data)) as EmployeeRecord
            For i as Integer = 0 To UBound(data)
                newData(i) = data(i).Clone()
            Next i
            Console.WriteLine ("EmployeeData: cloned employee data.")
            Return newData
        End Function
    End Class
```

```

Public Class RecordChecker
    Public Shared Function FindEmployees( _
        dataSource as EmployeeData, _
        department as Integer) as Collection(Of Integer)

        Dim storage as Collection(Of Integer) = new Collection(Of Integer)()
        Console.WriteLine("Record checker: beginning search.")
        For i as Integer = 0 To UBound(dataSource.Employees)
            If dataSource.Employees(i).Department = department
                Console.WriteLine("Record checker: found match at {0}.", i)
                storage.Add(dataSource.Employees(i).EmployeeId)
                Console.WriteLine("Record checker: stored match at {0}.", i)
            Else
                Console.WriteLine("Record checker: no match at {0}.", i)
            End If
        Next i
        Return storage
    End Function
End Class

Public Class Tester
    Public Shared Sub Main()
        Dim records(2) as EmployeeRecord
        Dim r0 as EmployeeRecord = new EmployeeRecord()
        r0.EmployeeId = 1
        r0.Department = 100
        records(0) = r0
        Dim r1 as EmployeeRecord = new EmployeeRecord()
        r1.EmployeeId = 2
        r1.Department = 100
        records(1) = r1
        Dim r2 as EmployeeRecord = new EmployeeRecord()
        r2.EmployeeId = 3
        r2.Department = 101
        records(2) = r2
        Dim empData as EmployeeData = new EmployeeData(records)
        Dim hits as Collection(Of Integer) = _
            RecordChecker.FindEmployees(empData, 100)
        For Each i as Integer In hits
            Console.WriteLine("found employee {0}", i)
        Next i
    End Sub
End Class
End Namespace

```

C#

 [复制代码](#)

```

using System;
using System.Collections.ObjectModel;
namespace Examples.DesignGuidelines.Properties
{
    public class EmployeeRecord
    {
        private int employeeId;
        private int department;
        public EmployeeRecord()
        {
        }
        public EmployeeRecord(int id, int departmentId)
        {
            EmployeeId = id;
            Department = departmentId;
        }
        public int Department
        {
            get {return department;}
            set {department = value;}
        }
        public int EmployeeId
        {
            get {return employeeId;}
            set {employeeId = value;}
        }
    }
}

```

```

    }
    public EmployeeRecord Clone()
    {
        return new EmployeeRecord(employeeId, department);
    }
}

public class EmployeeData
{
    EmployeeRecord[] data;
    public EmployeeData(EmployeeRecord[] data)
    {
        this.data = data;
    }
    public EmployeeRecord[] Employees
    {
        get
        {
            EmployeeRecord[] newData = CopyEmployeeRecords();
            return newData;
        }
    }
    EmployeeRecord[] CopyEmployeeRecords()
    {
        EmployeeRecord[] newData = new EmployeeRecord[data.Length];
        for(int i = 0; i < data.Length; i++)
        {
            newData[i] = data[i].Clone();
        }
        Console.WriteLine ("EmployeeData: cloned employee data.");
        return newData;
    }
}

public class RecordChecker
{
    public static Collection<int> FindEmployees(EmployeeData dataSource,
        int department)
    {
        Collection<int> storage = new Collection<int>();
        Console.WriteLine("Record checker: beginning search.");
        for (int i = 0; i < dataSource.Employees.Length; i++)
        {
            if (dataSource.Employees[i].Department == department)
            {
                Console.WriteLine("Record checker: found match at {0}.", i);
                storage.Add(dataSource.Employees[i].EmployeeId);
                Console.WriteLine("Record checker: stored match at {0}.", i);
            }
            else
            {
                Console.WriteLine("Record checker: no match at {0}.", i);
            }
        }
        return storage;
    }
}

public class Tester
{
    public static void Main()
    {
        EmployeeRecord[] records = new EmployeeRecord[3];
        EmployeeRecord r0 = new EmployeeRecord();
        r0.EmployeeId = 1;
        r0.Department = 100;
        records[0] = r0;
        EmployeeRecord r1 = new EmployeeRecord();
        r1.EmployeeId = 2;
        r1.Department = 100;
        records[1] = r1;
        EmployeeRecord r2 = new EmployeeRecord();
        r2.EmployeeId = 3;
    }
}

```

```
        r2.Department = 101;
        records[2] = r2;
        EmployeeData empData = new EmployeeData(records);
        Collection<int> hits = RecordChecker.FindEmployees(empData, 100);
        foreach (int i in hits)
        {
            Console.WriteLine("found employee {0}", i);
        }
    }
}
```

部分版权所有 2005 Microsoft Corporation。保留所有权利。

部分版权所有 Addison-Wesley Corporation。保留所有权利。

有关设计指南的更多信息, 请参见 Krzysztof Cwalina 和 Brad Abrams 编著、Addison-Wesley 于 2005 年出版的“Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries”(《框架设计指南: 可重用 .NET 库的约定、术语和模式》)。

☐ 请参见

概念

[属性设计](#)

[成员设计准则](#)

[类库开发的设计准则](#)



.NET Framework 开发人员指南

属性设计

[请参见](#)

通常，方法代表操作而属性代表数据。属性像字段一样使用，这意味着属性不应进行复杂的计算，也不应产生副作用。有关属性设计的更多信息，请参见[索引属性设计](#)和[属性更改通知事件](#)。

下列准则可帮助确保正确地设计属性。

如果调用方不应当更改属性值，则要创建只读属性。

注意，属性类型的可变性会影响最终用户可以更改的内容。例如，如果定义一个返回读/写集合的只读属性，则最终用户不能向该属性分配其他集合，但可以修改该集合中的元素。

不要提供仅支持 Set 操作的属性。

如果无法提供属性 `getter`，可以改用一個方法来实现该功能。方法名称应以 `Set` 开头，并按原样后跟属性名。例如，[AppDomain](#) 使用一个名为 [SetCachePath](#) 的方法，而不是名为 `CachePath` 的仅支持 `Set` 操作的属性。

避免从属性 `getter` 中引发异常。

属性 `getter` 应是没有任何前提条件的简单操作。如果 `getter` 可能会引发异常，请考虑将该属性重新设计为方法。此项建议不适用于索引器。索引可以因参数无效而引发异常。

在属性 `setter` 中引发异常是有效并可以接受的。

部分版权所有 2005 Microsoft Corporation。保留所有权利。

部分版权所有 Addison-Wesley Corporation。保留所有权利。

有关设计指南的更多信息，请参见 Krzysztof Cwalina 和 Brad Abrams 编著、Addison-Wesley 于 2005 年出版的“[Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries](#)”（《框架设计指南：可重用 .NET 库的约定、术语和模式》）。

■ 请参见

概念

[成员设计准则](#)

[类库开发的设计准则](#)



.NET Framework 开发人员指南

索引属性设计

[请参见](#)

索引属性允许像访问数组一样对一组项（例如字符串中的字符，或 [BitArray](#) 中的位）进行访问。索引属性（称为索引器或默认属性）与常规属性不同，因为索引属性接受参数，参数指示要访问组中的哪个元素。索引属性的实现应尽可能简单，因为索引器经常在循环中使用。下面的准则帮助确保您的类型在适当情况下包含设计良好的索引。

避免使用具有多个参数的索引属性。

如果一个索引器需要多个参数，请重新评估该属性是否确实表示对逻辑集合的访问。如果不是，则改用方法，并考虑选择以 [Get](#) 或 [Set](#) 开头的方法名。

避免为索引器设置除 **System.Int32**、**System.Int64**、**System.String**、**System.Object**、枚举或泛型类型参数之外的其他参数类型。

如果设计需要其他类型的参数，应该仔细重新评估该成员是否确实表示对逻辑集合的访问。如果不是，则改用方法，并考虑选择以 [Get](#) 或 [Set](#) 开头的方法名。

将“Item”名称用于索引属性，除非明显有更好的名称（有关示例，请参见 **System.String.Chars** (**System.Int32**) 属性）。

使用 [IndexerNameAttribute](#) 属性可自定义索引器的名称。

不要同时提供在语义上等价的索引器和方法。

在下面的代码示例中，索引器应改为方法。

Visual Basic

[复制代码](#)

```
<System.Runtime.CompilerServices.IndexerNameAttribute("PositionsHeld")> _  
    Public Property Item (skillId as Integer) as JobInfoCollection  
  
...  
  
Public Function GetPositions(skillId as Integer, _  
    minJobLevel as Integer) _  
    as JobInfoCollection
```

C#

[复制代码](#)

```
[System.Runtime.CompilerServices.IndexerNameAttribute("PositionsHeld")]  
    public JobInfoCollection this [int skillId]  
  
...  
  
public JobInfoCollection GetPositions(int skillId, int minJobLevel)
```

不要在一个类型中提供一组以上的重载索引器。

一些编译器（如 C# 编译器）会强制实施此准则。

有些语言不支持多组索引器。如果使用了多组索引器，有些开发人员将无法访问这些成员。

不要使用非默认索引属性。

一些编译器（如 C# 编译器）会强制实施此准则。并不是所有编程语言都支持非默认索引属性。如果使用了非默认索引属性，有些开发人员将无法访问这些成员。

部分版权所有 2005 Microsoft Corporation。保留所有权利。

部分版权所有 Addison-Wesley Corporation。保留所有权利。

有关设计指南的更多信息, 请参见 Krzysztof Cwalina 和 Brad Abrams 编著、Addison-Wesley 于 2005 年出版的“Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries”（《框架设计指南: 可重用 .NET 库的约定、术语和模式》）。

☐ 请参见

概念

[成员设计准则](#)

[类库开发的设计准则](#)

[属性设计](#)



.NET Framework 开发人员指南

属性更改通知事件

[请参见](#)

属性更改通知事件用于在属性值因内部或外部活动发生更改时向代码发出通知。这使得代码可以根据需要更新相关状态（例如，改变用户界面中控件的外观）。

当修改高级 API（通常是设计器组件）中的属性值时，应考虑引发更改通知事件。

这项准则适用于那些可通过更改通知将重要值添加到库的高级成员。例如，提供用户界面或与之交互的对象使用更改通知使相关用户界面对象相应地得到更新。当不向库添加值或通知会频繁地执行以致对性能造成严重影响的情况下，不应使用更改通知事件。例如，在每次向常规集合添加元素或将元素从中删除时就引发更改通知事件是不正确的。若要避免给常用类型增加不必要的复杂性，在需要这一功能时应使用一个专用集合。.NET Framework 2.0 版的库提供了 [Collection\(T\)](#)，该集合用作常规集合。Framework 还为需要通知的集合提供了 [BindingList\(T\)](#)。

当属性值由于外部因素发生更改时，应考虑引发更改通知事件。

如果属性值由于某种外部因素（如用户输入）发生更改，则应在更改永久生效之前使用更改通知事件指示该值即将更改。在更改永久生效后，可使用另一事件指示该值已进行了更改。例如，[Control](#) 类提供了 [Validating](#) 和 [Validated](#) 事件来向代码发出验证控件的通知。

部分版权所有 2005 Microsoft Corporation。保留所有权利。

部分版权所有 Addison-Wesley Corporation。保留所有权利。

有关设计指南的更多信息，请参见 Krzysztof Cwalina 和 Brad Abrams 编著、Addison-Wesley 于 2005 年出版的“Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries”（《框架设计指南：可重用 .NET 库的约定、术语和模式》）。

■ 请参见

概念

[成员设计准则](#)

[类库开发的设计准则](#)

[索引属性设计](#)

[属性设计](#)



.NET Framework 开发人员指南

构造函数设计

[请参见](#)

构造函数是一类特殊的方法，用于初始化类型和创建类型的实例。类型构造函数用于初始化类型中的静态数据。类型构造函数由公共语言运行库 (CLR) 在创建类型的任何实例之前调用。类型构造函数是 **static**（在 **Visual Basic** 中为 **Shared**）方法，不能带任何参数。实例构造函数用于创建类型的实例。实例构造函数可以带参数，也可以不带参数。不带任何参数的实例构造函数称为默认构造函数。

下列准则描述了创建构造函数的最佳做法。

将构造函数参数用作设置主要属性的快捷方式。

通过使用构造函数设置属性应与直接设置属性相同。下面的代码示例演示一个 `EmployeeRecord` 类，该类可以通过调用构造函数初始化，也可以通过直接设置属性初始化。`EmployeeManagerConstructor` 类演示如何使用构造函数初始化 `EmployeeRecord` 对象。`EmployeeManagerProperties` 类演示如何使用属性初始化 `EmployeeRecord` 对象。`Tester` 类演示了两种方式初始化的对象的状态是相同的。

Visual Basic [复制代码](#)

```
Imports System
Imports System.Collections.ObjectModel
namespace Examples.DesignGuidelines.Constructors

    ' This Class can get its data either by setting
    ' properties or by passing the data to its constructor.
    Public Class EmployeeRecord

        private employeeIdValue as Integer
        private departmentValue as Integer

        Public Sub New()
            End Sub

        Public Sub New(id as Integer, department as Integer)
            Me.employeeIdValue = id
            Me.departmentValue = department
        End Sub

        Public Property Department as Integer
            Get
                Return departmentValue
            End Get
            Set
                departmentValue = value
            End Set
        End Property

        Public Property EmployeeId as Integer
            Get
                Return employeeIdValue
            End Get
            Set
                employeeIdValue = value
            End Set
        End Property

        Public Sub DisplayData()
            Console.WriteLine("{0} {1}", EmployeeId, Department)
        End Sub
    End Class

    ' This Class creates Employee records by passing
    ' arguments to the constructor.
    Public Class EmployeeManagerConstructor
        Dim employees as Collection(Of EmployeeRecord) = _
            new Collection(Of EmployeeRecord)()
```

```

    Public Sub AddEmployee(employeeId as Integer, department as Integer)
        Dim record as EmployeeRecord = new EmployeeRecord(employeeId, department)
        employees.Add(record)
        record.DisplayData()
    End Sub
End Class

' This Class creates Employee records by setting properties.
Public Class EmployeeManagerProperties
    Dim employees as Collection(Of EmployeeRecord)= _
        new Collection(Of EmployeeRecord)()
    Public Sub AddEmployee(employeeId as Integer, department as Integer)
        Dim record as EmployeeRecord = new EmployeeRecord()
        record.EmployeeId = employeeId
        record.Department = department
        employees.Add(record)
        record.DisplayData()
    End Sub
End Class

Public Class Tester
    ' The following method creates objects with the same state
    ' using the two different approaches.
    Public Shared Sub Main()
        Dim byConstructor as EmployeeManagerConstructor = _
            new EmployeeManagerConstructor()
        byConstructor.AddEmployee(102, 102)

        Dim byProperties as EmployeeManagerProperties = _
            new EmployeeManagerProperties()
        byProperties.AddEmployee(102, 102)
    End Sub
End Class
End Namespace

```

C#

 [复制代码](#)

```

using System;
using System.Collections.ObjectModel;
namespace Examples.DesignGuidelines.Constructors
{
    // This class can get its data either by setting
    // properties or by passing the data to its constructor.
    public class EmployeeRecord
    {
        private int employeeId;
        private int department;

        public EmployeeRecord()
        {
        }
        public EmployeeRecord(int id, int department)
        {
            this.employeeId = id;
            this.department = department;
        }
        public int Department
        {
            get {return department;}
            set {department = value;}
        }
        public int EmployeeId
        {
            get {return employeeId;}
            set {employeeId = value;}
        }
        public void DisplayData()
        {
            Console.WriteLine("{0} {1}", EmployeeId, Department);
        }
    }
    // This class creates Employee records by passing

```

```

// arguments to the constructor.
public class EmployeeManagerConstructor
{
    Collection<EmployeeRecord> employees = new Collection<EmployeeRecord>();

    public void AddEmployee(int employeeId, int department)
    {
        EmployeeRecord record = new EmployeeRecord(employeeId, department);
        employees.Add(record);
        record.DisplayData();
    }
}
// This class creates Employee records by setting properties.
public class EmployeeManagerProperties
{
    Collection<EmployeeRecord> employees = new Collection<EmployeeRecord>();
    public void AddEmployee(int employeeId, int department)
    {
        EmployeeRecord record = new EmployeeRecord();
        record.EmployeeId = employeeId;
        record.Department = department;
        employees.Add(record);
        record.DisplayData();
    }
}
public class Tester
{
    // The following method creates objects with the same state
    // using the two different approaches.
    public static void Main()
    {
        EmployeeManagerConstructor byConstructor =
            new EmployeeManagerConstructor();
        byConstructor.AddEmployee(102, 102);

        EmployeeManagerProperties byProperties =
            new EmployeeManagerProperties();
        byProperties.AddEmployee(102, 102);
    }
}

```

注意，在这些示例中，以及在设计良好的库中，这两种方式都可以创建具有相同状态的对象。开发人员首选哪种方式并不重要。

如果构造函数参数只用于设置一个属性，请务必为构造函数参数和该属性使用相同的名称。这类参数和属性之间的唯一差异应是大小写不同。

前面的示例已对此准则进行了演示。

根据需要，可在实例构造函数中引发异常。

构造函数与其他方法一样，应引发并处理异常。具体地说，构造函数不应捕捉和隐藏它无法处理的任何异常。有关异常的更多信息，请参见[异常设计准则](#)。

如果需要公共默认构造函数，请在类中进行显式声明。

如果类支持默认构造函数，则显式定义默认构造函数是最佳做法。尽管某些编译器会自动向类中添加默认构造函数，但显式添加默认构造函数会使代码更易于维护。即使由于您添加了带参数的构造函数，导致编译器停止发出默认构造函数，这样也可确保定义默认构造函数。

避免在结构中使用默认构造函数。

许多编译器（包括 C# 编译器）不支持在结构中使用无参数构造函数。

不要在对象的构造函数中调用对象的虚成员。

无论是否调用了定义派生程度最高的重写的类型的构造函数，调用虚成员都会导致调用派生程度最高的重写。下面的代码示例对此进行了演示。基类构造函数执行时，即使尚未调用派生类的构造函数，也会调用

派生类的成员。此示例输出 `BadBaseClass` 以显示 `DerivedFromBad` 构造函数尚未更新状态字段。

Visual Basic

[复制代码](#)

```
Imports System

Namespace Examples.DesignGuidelines.MemberDesign
    Public Class BadBaseClass

        Protected state as String
        Public Sub New()

            state = "BadBaseClass"
            SetState()
        End Sub
        Public Overridable Sub SetState()
        End Sub
    End Class

    Public Class DerivedFromBad
        Inherits BadBaseClass
        Public Sub New()

            state = "DerivedFromBad "
        End Sub

        Public Overrides Sub SetState()
            Console.WriteLine(state)
        End Sub
    End Class

    Public Class tester

        Public Shared Sub Main()

            Dim b as DerivedFromBad = new DerivedFromBad()
        End Sub
    End Class
End Namespace
```

C#

[复制代码](#)

```
using System;

namespace Examples.DesignGuidelines.MemberDesign
{
    public class BadBaseClass
    {
        protected string state;
        public BadBaseClass()
        {
            state = "BadBaseClass";
            SetState();
        }
        public virtual void SetState()
        {
        }
    }

    public class DerivedFromBad : BadBaseClass
    {
        public DerivedFromBad()
        {
            state = "DerivedFromBad ";
        }
        public override void SetState()
        {
            Console.WriteLine(state);
        }
    }

    public class tester
```

```
{  
    public static void Main()  
    {  
        DerivedFromBad b = new DerivedFromBad();  
    }  
}
```

部分版权所有 2005 Microsoft Corporation。保留所有权利。

部分版权所有 Addison-Wesley Corporation。保留所有权利。

有关设计指南的更多信息, 请参见 Krzysztof Cwalina 和 Brad Abrams 编著、Addison-Wesley 于 2005 年出版的“**Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries**” (《框架设计指南: 可重用 .NET 库的约定、术语和模式》)。

☐ 请参见

概念

[成员设计准则](#)

[类库开发的设计准则](#)

[类型构造函数设计](#)



.NET Framework 开发人员指南

类型构造函数设计

[请参见](#)

类型构造函数用于初始化类型中的静态数据。它由公共语言运行库 (CLR) 在创建类型的任何实例之前调用。类型构造函数是 **static** (在 **Visual Basic** 中为 **Shared**) 方法, 不能带任何参数。

下列准则有助于确保您使用静态构造函数的方法符合最佳做法。

一定要将类型构造函数设为私有。

类型构造函数 (也称为类构造函数或静态构造函数) 用于初始化类型。**CLR** 在创建类型的第一个实例或调用类型上的任何静态成员之前调用类型构造函数。如果类型构造函数不是私有的, 则它可由 **CLR** 以外的代码调用。根据在构造函数中执行的操作的不同, 这可能导致意外行为。

不要从类型构造函数引发异常。

如果类型构造函数引发异常, 则该类型在引发异常的应用程序域中不可用。

考虑以内联方式初始化静态字段而不是显式使用静态构造函数, 因为 **CLR** 可以优化没有显式定义的静态构造函数的类型的性能。

下面的代码示例演示了一个不能优化的设计。

Visual Basic

[复制代码](#)

```
Public Class BadStaticExample
    Shared runId as Guid
    Shared Sub New()
        runId = Guid.NewGuid()
    End Sub
    ' Other members...
End Class
```

C#

[复制代码](#)

```
public class BadStaticExample
{
    static Guid runId;
    static BadStaticExample()
    {
        runId = Guid.NewGuid();
    }
    // Other members...
}
```

下面的代码示例可以优化。

Visual Basic

[复制代码](#)

```
Public Class GoodStaticExample
    Shared runId as Guid = Guid.NewGuid()
    ' Other members...
End Class
```

C#

[复制代码](#)

```
public class GoodStaticExample
{
    static Guid runId = Guid.NewGuid();
    // Other members...
}
```

部分版权所有 2005 Microsoft Corporation。保留所有权利。

部分版权所有 Addison-Wesley Corporation。保留所有权利。

有关设计指南的更多信息, 请参见 Krzysztof Cwalina 和 Brad Abrams 编著、Addison-Wesley 于 2005 年出版的“**Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries**” (《框架设计指南: 可重用 .NET 库的约定、术语和模式》)。

☐ 请参见

概念

[成员设计准则](#)

[类库开发的设计准则](#)

[构造函数设计](#)



.NET Framework 开发人员指南

事件设计

[请参见](#)

事件是操作发生时允许执行特定于应用程序的代码的机制。事件要么在相关联的操作发生前发生（事前事件），要么在操作发生后发生（事后事件）。例如，当用户单击窗口中的按钮时，将引发一个事后事件，以允许执行特定于应用程序的方法。事件处理程序委托会绑定到系统引发事件时要执行的方法。事件处理程序会添加到事件中，以便当事件引发时，事件处理程序能够调用它的方法。事件可以具有特定于事件的数据，例如，按下鼠标事件可以包含有关屏幕光标位置的数据。

事件处理方法的签名与事件处理程序委托的签名是相同的。事件处理程序签名遵循下面的约定：

- 1 返回类型为 [Void](#)。
- 1 第一个参数命名为 `sender`，是 [Object](#) 类型。它是引发事件的对象。
- 1 第二个参数命名为 `e`，是 [EventArgs](#) 类型或 [EventArgs](#) 的派生类。它是特定于事件的数据。
- 1 该方法有且仅有两个参数。

有关事件的更多信息，请参见 [处理和引发事件](#)。

使用 `System.EventHandler<T>`，而不要手动创建用作事件处理程序的新委托。

此准则主要适用于新的功能区域。如果您是在已经使用非泛型事件处理程序的区域中扩展功能，则可以继续使用非泛型事件处理程序，以保持设计一致。

如果您的库针对的是不支持泛型的 .NET Framework 版本，则无法遵循此准则。

考虑使用 `System.EventArgs` 的派生类作为事件参数，除非您完全确定事件决不会需要向事件处理方法传递任何数据（这种情况下可以直接使用 `System.EventArgs` 类型）。

如果您定义的事件采用 [EventArgs](#) 实例而不是您定义的派生类，则不能够在以后的版本中向该事件添加数据。出于上述原因，建议创建一个空的 [EventArgs](#) 派生类。这使您能够在以后的版本中在不引入重大更改的情况下向事件添加数据。

使用受保护的虚方法来引发每个事件。这只适用于未密封类的非静态事件，而不适用于结构、密封类或静态事件。

遵循此准则可使派生类能够通过重写受保护的方法来处理基类事件。受保护的 `virtual`（在 Visual Basic 中是 `Overridable`）方法的名称应该是为事件名加上 `On` 前缀而得到的名称。例如，名为“`TimeChanged`”的事件的受保护的虚方法被命名为“`OnTimeChanged`”。

重要说明：

重写受保护的虚方法的派生类无需调用基类实现。即使没有调用基类的实现，基类也必须继续正常工作。

使用一个参数，该参数已类型化为引发事件的受保护方法的事件参数类。该参数应命名为 `e`。

[FontDialog](#) 类提供下面的方法，该方法引发 [Apply](#) 事件：

Visual Basic

 [复制代码](#)

```
Protected Overridable Sub OnApply( ByVal e As EventArgs )  
protected virtual void OnApply(EventArgs e);
```

当引发非静态事件时，不要将 `null`（在 Visual Basic 中为 `Nothing`）作为 `sender` 参数进行传递。

对于静态事件，`sender` 参数应该为 `null`（在 Visual Basic 中是 `Nothing`）。

当引发事件时，不要将 `null`（在 Visual Basic 中为 `Nothing`）作为事件数据参数进行传递。

如果没有事件数据，则传递 [Empty](#)，而不要传递 `null`。

事件处理方法中会发生任意代码执行，对此一定要做好准备。

考虑将引发事件的代码放置在 `try-catch` 块中，以防止由事件处理程序引发的未处理异常所导致的程序终

止。

考虑引发最终用户可以取消的事件。这仅适用于事前事件。

如果您正在设计可取消的事件, 请使用 [CancelEventArgs](#) (而非 [EventArgs](#)) 作为事件数据对象 `e` 的基类。

部分版权所有 2005 Microsoft Corporation。保留所有权利。

部分版权所有 Addison-Wesley Corporation。保留所有权利。

有关设计指南的更多信息, 请参见 Krzysztof Cwalina 和 Brad Abrams 编著、Addison-Wesley 于 2005 年出版的 “[Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries](#)” (《框架设计指南: 可重用 .NET 库的约定、术语和模式》)。

■ 请参见

概念

[成员设计准则](#)

[类库开发的设计准则](#)

[自定义事件处理程序设计](#)



.NET Framework 开发人员指南

自定义事件处理程序设计

[请参见](#)

如果不使用泛型 [EventHandler\(TEventArgs\)](#) 委托，下面的准则有助于正确设计事件处理程序。

对事件处理程序使用 `System.Void` 返回类型。

事件处理程序可以调用多个方法，但不能从一个事件处理程序接收多个返回值。通过使方法返回 [Void](#)，就可以防止丢失返回值数据。

事件处理程序的参数不能多于两个。

部分版权所有 2005 Microsoft Corporation。保留所有权利。

部分版权所有 Addison-Wesley Corporation。保留所有权利。

有关设计指南的更多信息，请参见 Krzysztof Cwalina 和 Brad Abrams 编著、Addison-Wesley 于 2005 年出版的“[Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries](#)”（《框架设计指南：可重用 .NET 库的约定、术语和模式》）。

■ 请参见

概念

[成员设计准则](#)

[类库开发的设计准则](#)

[事件设计](#)



.NET Framework 开发人员指南

字段设计

[请参见](#)

字段保存对象的关联数据。在大多数情况下，库中的所有非静态字段对开发人员都应是不可见的。下面的准则有助于在库设计中正确使用字段。

不要提供公共的或受保护的实例字段。

公共字段和受保护字段未经版本控制，不受代码访问安全性要求的保护。使用私有字段，并通过属性公开这些私有字段，而不要使用公共可见字段。

对不会更改的常数使用常数字段。

例如，[Math](#) 类将 [E](#) 和 [PI](#) 定义为静态常数。

编译器将 **const** 字段的值直接插入调用代码中，这意味着任何情况下，**const** 值都不能更改，可避免引起兼容性问题。

对预定义对象实例使用公共静态只读字段。

例如，[DateTime](#) 类提供静态只读字段，使用这些字段可以获取设置为最大或最小时间值的 [DateTime](#) 对象。请参见 [MaxValue](#) 和 [MinValue](#)。

不要将可变类型的实例指定给只读字段。

使用可变类型创建的对象可以在创建后进行修改。例如，数组和大多数集合是可变类型，而 [Int32](#)、[Uri](#) 和 [String](#) 是不可变类型。对于保存可变引用类型的字段，只读修饰符可防止字段值被改写，但不能防止可变类型被修改。

下面的代码示例演示使用只读字段会出现的问题。[BadDesign](#) 类创建一个只读字段，并使用只读属性公开该字段。这不能防止 [ShowBadDesign](#) 类修改该只读字段的内容。

Visual Basic

[复制代码](#)

```
Imports System

Namespace Examples.DesignGuidelines.Fields

    Public Class BadDesign

        Public Readonly dataValues as Integer() = {1,2,3}

        Public Readonly Property Data as Integer ()

            Get
                Return dataValues
            End Get
        End Property

        Public Sub WriteData()

            For Each i as Integer In dataValues

                Console.Write ("{0} ", i)
            Next i
            Console.WriteLine()
        End Sub
    End Class

    Public Class ShowBadDesign

        Public Shared Sub Main()

            Dim bad as BadDesign = new BadDesign()
            ' The following line will write: 1 2 3
            bad.WriteData()
        End Sub
    End Class
End Namespace
```

```

    Dim badData as Integer() = bad.Data
    For i as Integer = 0 To badData.Length - 1

        badData(i) = 0
    Next i

    ' The following line will write: 0 0 0
    ' because bad's data has been modified.
    bad.WriteData()
End Sub
End Class
End Namespace

```

C#

 [复制代码](#)

```

using System;

namespace Examples.DesignGuidelines.Fields
{
    public class BadDesign
    {
        public readonly int[] data = {1,2,3};

        public int [] Data
        {
            get {return data;}
        }
        public void WriteData()
        {
            foreach (int i in data)
            {
                Console.Write ("{0} ", i);
            }
            Console.WriteLine();
        }
    }
    public class ShowBadDesign
    {
        public static void Main()
        {
            BadDesign bad = new BadDesign();
            // The following line will write: 1 2 3
            bad.WriteData();

            int[] badData = bad.Data;
            for (int i = 0; i< badData.Length; i++)
            {
                badData[i] = 0;
            }
            // The following line will write: 0 0 0
            // because bad's data has been modified.
            bad.WriteData();
        }
    }
}

```

部分版权所有 2005 Microsoft Corporation。保留所有权利。

部分版权所有 Addison-Wesley Corporation。保留所有权利。

有关设计指南的更多信息, 请参见 Krzysztof Cwalina 和 Brad Abrams 编著、Addison-Wesley 于 2005 年出版的 “Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries” (《框架设计指南: 可重用 .NET 库的约定、术语和模式》)。

☐ 请参见

概念

[成员设计准则](#)

[类库开发的设计准则](#)



.NET Framework 开发人员指南
运算符重载
[请参见](#)

运算符重载允许使用 “+”、“-”、“=” 和 “!=” 等运算符合并和比较类型。通过在类型中添加运算符重载，开发人员可以像使用内置基元类型一样使用该类型。只有在运算对类型具有很直观的意义（例如，支持表示数值的类型的两个实例相加）的情况下，才应进行运算符重载。不应使用运算符重载为非直观运算提供语法快捷方式。

下面的示例演示 [DateTime](#) 类的加法运算的签名。

[Visual Basic]

[复制代码](#)

```
Public Shared Function op_Addition(ByVal d As DateTime, _
    ByVal t As TimeSpan _
) As DateTime
```

[C#]

[复制代码](#)

```
public static DateTime op_Addition(
    DateTime d,
    TimeSpan t
);
```

考虑在其用法应类似于基元类型的类型中定义运算符重载。

例如，[String](#) 定义运算符 == 和 !=。

除非至少有一个操作数属于定义重载的类型，否则不要提供运算符重载。

C# 编译器强制执行这一准则。

以对称方式重载运算符。

例如，如果重载相等运算符，也应重载不等运算符。同样，如果重载小于运算符，也应重载大于运算符。

考虑为每个重载运算符所对应的方法提供友好的名称。

必须遵守此项准则才能符合 CLS。下表列出了运算符符号、其相应的替换方法以及运算符名称。

C# 运算符符号	替换方法名称	运算符名称
未定义	ToXxx 或 FromXxx	op_Implicit
未定义	ToXxx 或 FromXxx	op_Explicit
+（二进制）	Add	op_Addition
-（二进制）	Subtract	op_Subtraction
*（二进制）	Multiply	op_Multiply
/	Divide	op_Division
%	Mod	op_Modulus
^	Xor	op_ExclusiveOr
&（二进制）	BitwiseAnd	op_BitwiseAnd
	BitwiseOr	op_BitwiseOr

&&	And	op_LogicalAnd
 	Or	op_LogicalOr
=	Assign	op_Assign
<<	LeftShift	op_LeftShift
>>	RightShift	op_RightShift
未定义	LeftShift	op_SignedRightShift
未定义	RightShift	op_UnsignedRightShift
==	Equals	op_Equality
>	CompareTo	op_GreaterThan
<	CompareTo	op_LessThan
!=	Equals	op_Inequality
>=	CompareTo	op_GreaterThanOrEqual
<=	CompareTo	op_LessThanOrEqual
*=	Multiply	op_MultiplicationAssignment
-=	Subtract	op_SubtractionAssignment
^=	Xor	op_ExclusiveOrAssignment
<<=	LeftShift	op_LeftShiftAssignment
%=	Mod	op_ModulusAssignment
+=	Add	op_AdditionAssignment
&=	BitwiseAnd	op_BitwiseAndAssignment
 =	BitwiseOr	op_BitwiseOrAssignment
,	Comma	op_Comma
/=	Divide	op_DivisionAssignment
--	Decrement	op_Decrement
++	Increment	op_Increment
- (一元)	Negate	op_UnaryNegation
+ (一元)	Plus	op_UnaryPlus
~	OnesComplement	op_OnesComplement

部分版权所有 2005 Microsoft Corporation。保留所有权利。

部分版权所有 Addison-Wesley Corporation。保留所有权利。

有关设计指南的更多信息，请参见 Krzysztof Cwalina 和 Brad Abrams 编著、Addison-Wesley 于 2005 年出版的“**Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries**”（《框架设计指南：可重用 .NET 库的约定、术语和模式》）。

☐ 请参见

概念

[成员设计准则](#)

类库开发的设计准则



.NET Framework 开发人员指南

转换运算符

[请参见](#)

转换运算符可将对象从一种类型转换为另一种类型。转换运算符可以是隐式的也可以是显式的。隐式转换运算符不需要在源代码中指定类型转换即可执行转换。显式转换运算符则要求在源代码中指定类型转换才能执行转换。

下面的签名演示 [Point](#) 类的显式转换运算符，该转换运算符用于在 [Point](#) 和 [Size](#) 之间进行转换。

[Visual Basic]

[复制代码](#)

```
Public Shared Function op_Explicit( _  
    ByVal p As Point _  
    ) As Size
```

[C#]

[复制代码](#)

```
public static Size op_Explicit(  
    Point p  
);
```

如果最终用户未明确要求此类转换，则不要提供相应的转换运算符。

理想情况下，应存在客户研究数据，以支持定义转换运算符。此外，如果存在一些示例，其中一个或多个类似类型需要此类转换，也可以支持定义转换运算符。

不要在类型域之外定义转换运算符。

例如，[Int32](#)、[Double](#) 和 [Decimal](#) 都是数字类型，而 [DateTime](#) 不是数字类型。将 [Double](#) 类型转换为 [DateTime](#) 类型不应以转换运算符的形式实现。如果要将一种类型转换为不同域中的另一种类型，请使用构造函数。

如果转换可能丢失信息，则不要提供隐式转换运算符。

例如，从 [Double](#) 到 [Single](#) 的转换不应是隐式转换，原因是 [Double](#) 的精度高于 [Single](#)。对于有损转换，可以提供显式转换运算符。

不要在隐式强制转换中引发异常。

隐式强制转换是由系统调用的；用户可能不会觉察发生了转换，这会给调试代码带来困难。

如果对强制转换运算符的调用导致有损转换，而该运算符的协定不允许有损转换，则会引发 **System.InvalidCastException**。

部分版权所有 2005 Microsoft Corporation。保留所有权利。

部分版权所有 Addison-Wesley Corporation。保留所有权利。

有关设计指南的更多信息，请参见 Krzysztof Cwalina 和 Brad Abrams 编著、Addison-Wesley 于 2005 年出版的“Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries”（《框架设计指南：可重用 .NET 库的约定、术语和模式》）。

[请参见](#)

[概念](#)

[成员设计准则](#)

[类库开发的设计准则](#)



.NET Framework 开发人员指南

参数设计

[请参见](#)

本主题中的准则帮助您为成员参数选择正确的类型和名称。此外，下列主题还提供了参数设计准则。

- | [在枚举和布尔参数之间选择](#)
- | [参数数目可变的成员](#)
- | [指针参数](#)
- | [传递参数](#)
- | [验证参数](#)

使用派生程度最小的参数类型提供成员所需的功能。

下面的代码示例阐释了这一准则。BookInfo 类从 Publication 类继承。Manager 类实现两个方法：BadGetAuthorBiography 和 GoodGetAuthorBiography。BadGetAuthorBiography 即使只使用 Publication 中声明的成员，也使用对 BookInfo 对象的引用。GoodGetAuthorBiography 方法演示了正确的设计。

Visual Basic

[复制代码](#)

```
' A Class with some basic information.
Public Class Publication
    Dim Protected authorValue as String
    Dim Protected publicationDateValue as DateTime

    Public Sub new(author as String, publishDate as DateTime)
        Me.authorValue = author
        Me.PublicationDateValue = publishDate
    End Sub

    Public Readonly Property PublicationDate as DateTime
        Get
            Return publicationDateValue
        End Get
    End Property

    Public Readonly Property Author as String
        Get
            Return authorValue
        End Get
    End Property
End Class

' A Class that derives from Publication
Public Class BookInfo
    Inherits Publication

    Dim isbnValue as String

    Public Sub new(author as string, _
        publishDate as DateTime, _
        isbn as String)
        MyBase.New(author, publishDate)
        Me.isbnValue = isbn
    End Sub

    Public Readonly Property Isbn as String
        Get
            Return isbnValue
        End Get
    End Property
End Class

Public Class Manager
    ' This method does not use the Isbn member
    ' so it doesn't need a specialized reference to Books
    Shared Function BadGetAuthorBiography(book as BookInfo) as String
```

```

    Dim biography as String = ""
    Dim author as String = book.Author
    ' Do work here.
    Return biography
End Function

' This method shows the correct design.
Shared Function GoodGetAuthorBiography(item as Publication) as String
    Dim biography as String = ""
    Dim author as String = item.Author
    ' Do work here.
    Return biography
End Function

```

C#

 [复制代码](#)

```

// A class with some basic information.
public class Publication
{
    string author;
    DateTime publicationDate;

    public Publication(string author, DateTime publishDate)
    {
        this.author = author;
        this.publicationDate = publishDate;
    }
    public DateTime PublicationDate
    {
        get {return publicationDate;}
    }
    public string Author
    {
        get {return author;}
    }
}

// A class that derives from Publication
public class BookInfo :Publication
{
    string isbn;
    public BookInfo(string author, DateTime publishDate, string isbn) :
        base(author, publishDate)
    {
        this.isbn = isbn;
    }
    public string Isbn
    {
        get {return isbn;}
    }
}

public class Manager
{
    // This method does not use the Isbn member
    // so it doesn't need a specialized reference to Books
    static string BadGetAuthorBiography(BookInfo book)
    {
        string biography = "";
        string author = book.Author;
        // Do work here.
        return biography;
    }
    // This method shows the correct design.
    static string GoodGetAuthorBiography(Publication item)
    {
        string biography = "";
        string author = item.Author;
        // Do work here.
        return biography;
    }
}

```

不要使用保留的参数。

库的未来版本可以添加采用其他参数的新重载。

下面的代码示例首先演示了违反此项准则的不正确方法，然后演示了采用正确设计的方法。

Visual Basic

 [复制代码](#)

```
Public Sub BadStoreTimeDifference (localDate as DateTime, _
    toWhere as TimeZone, _
    reserved as Object)
    ' Do work here.
End Sub

Public Sub GoodCStoreTimeDifference (localDate as DateTime, _
    toWhere as TimeZone)
    ' Do work here.
End Sub

Public Sub GoodCStoreTimeDifference (localDate as DateTime, _
    toWhere as TimeZone, _
    useDayLightSavingsTime as Boolean)
    ' Do work here.
End Sub
```

C#

 [复制代码](#)

```
public void BadStoreTimeDifference (DateTime localDate,
    TimeZone toWhere,
    Object reserved)
{
    // Do work here.
}

public void GoodCStoreTimeDifference (DateTime localDate,
    TimeZone toWhere)
{
    // Do work here.
}

public void GoodCStoreTimeDifference (DateTime localDate,
    TimeZone toWhere,
    bool useDayLightSavingsTime)
{
    // Do work here.
}
```

不要使用公开显露的采用指针、指针数组或多维数组作为参数的方法。

使用大多数库时都无需了解这些高级功能。

将所有输出参数放在所有按值传递参数和引用传递参数（不包括参数数组）之后，即使这会导致参数在重载间排序不一致也要如此。

这种约定使得方法签名更易于理解。

在重写成员或实现接口成员时，要保持参数命名的一致性。

重写应使用相同的参数名。重载应使用与声明成员相同的参数名。接口实现应使用接口成员签名中定义的同名称。

部分版权所有 2005 Microsoft Corporation。保留所有权利。

部分版权所有 Addison-Wesley Corporation。保留所有权利。

有关设计指南的更多信息，请参见 Krzysztof Cwalina 和 Brad Abrams 编著、Addison-Wesley 于 2005 年出版的“Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries”（《框架设计指南：可重用 .NET 库的约定、术语和模式》）。

■ 请参见

概念

[成员设计准则](#)

[类库开发的设计准则](#)



.NET Framework 开发人员指南

在枚举和布尔参数之间选择

[请参见](#)

下列准则可帮助您确定参数类型应是枚举还是 [Boolean](#) 值。

如果成员本来有两个或多个布尔参数，则使用枚举。

枚举可以显著提高成员签名的可读性。请考虑下面的方法调用：

Visual Basic

[复制代码](#)

```
Type.GetType("Contoso.Controls.Array", True, False)
```

C#

[复制代码](#)

```
Type.GetType("Contoso.Controls.Array", true, false);
```

如果不查阅文档或添加代码注释，很难理解类似于这样的调用。如果调用使用枚举值而非多个布尔值，则阅读起来要容易得多，如下面的代码示例所示。

Visual Basic

[复制代码](#)

```
BetterType.GetType("Contoso.Controls.Array", _  
    ErrorOptions.ThrowOnError, _  
    CasingOptions.CaseInsensitive)
```

C#

[复制代码](#)

```
BetterType.GetType("Contoso.Controls.Array",  
    ErrorOptions.ThrowOnError,  
    CasingOptions.CaseInsensitive);
```

除非完全确定永不需要多于两个的值，否则不要使用布尔值。

枚举允许在以后的版本中添加值，但向枚举添加值可能会引入兼容性问题。有关更多信息，请参见[向枚举添加值](#)。

部分版权所有 2005 Microsoft Corporation。保留所有权利。

部分版权所有 Addison-Wesley Corporation。保留所有权利。

有关设计指南的更多信息，请参见 Krzysztof Cwalina 和 Brad Abrams 编著、Addison-Wesley 于 2005 年出版的“Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries”（《框架设计指南：可重用 .NET 库的约定、术语和模式》）。

请参见

概念

[成员设计准则](#)

[类库开发的设计准则](#)

[参数设计](#)



.NET Framework 开发人员指南

验证参数

[请参见](#)

下列准则有助于确保正确验证参数。

一定要验证传递给公共的、受保护的或显式实现的成员的参数。如果验证失败，则引发 `System.ArgumentException` 或其派生类之一。

此准则不要求验证代码在公共可见成员中。将参数传递给处理验证的内部方法是可以接受的。

一定要验证枚举参数。

不能假设枚举参数为在枚举中定义的值，因为公共语言运行库 (CLR) 支持将任何整数值强制转换为枚举值，而不论该值是否在枚举中定义。

不要将 `System.Enum.IsDefined(System.Type, System.Object)` 用于枚举范围检查，因为它是基于枚举的运行库类型，而该类型会随版本的不同而有所更改。

较高版本的库可向提供的枚举添加值。使用 [IsDefined](#) 验证数据是危险的，因为现有代码（它不处理新值）将视该新值为有效的输入，原因是 [IsDefined](#) 为新值返回 `true`。检查输入是否在程序可支持的值范围中，如果不在，则引发一个异常。

应注意传递的可变参数在经过验证后可能已发生了更改。

如果成员是安全敏感的，则制作可变对象的私有副本并使用该副本进行验证和处理。这仅适用于可变数据。不需要复制不可变数据（如 [Uri](#) 对象）。

部分版权所有 2005 Microsoft Corporation。保留所有权利。

部分版权所有 Addison-Wesley Corporation。保留所有权利。

有关设计指南的更多信息，请参见 Krzysztof Cwalina 和 Brad Abrams 编著、Addison-Wesley 于 2005 年出版的“[Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries](#)”（《框架设计指南：可重用 .NET 库的约定、术语和模式》）。

☐ 请参见

概念

[成员设计准则](#)

[类库开发的设计准则](#)

[参数设计](#)



.NET Framework 开发人员指南

传递参数

[请参见](#)

方法参数可以通过值，通过引用传递，也可以作为输出参数传递。通过值传递参数时，方法获取调用方数据的副本，但不能改变调用方的数据副本。通过引用传递参数时，方法获取指向调用方数据的指针。此数据与调用方共享。如果方法对引用参数做出更改，则这些更改是对调用方的数据进行的。使用引用参数时，方法可以使用数据的初始状态。输出参数与引用参数类似，不同之处在于，输出参数以独占方式用于向调用方返回数据，而引用参数可用于将数据传入方法，也可用于从方法中接收数据。

避免使用输出参数或引用参数。

使用定义输出参数或引用参数的成员需要开发人员理解指针、值类型和引用类型之间的细微差别以及输出参数和引用参数之间的初始化差异。

不要通过引用传递引用类型。

通过引用传递一个对象使方法能够用不同的实例替换该对象。在大多数情况下，方法应使用提供的对象，而不应将其替换。对于此规则，有一些少量的例外（例如，可用于交换引用的方法）。

部分版权所有 2005 Microsoft Corporation。保留所有权利。

部分版权所有 Addison-Wesley Corporation。保留所有权利。

有关设计指南的更多信息，请参见 Krzysztof Cwalina 和 Brad Abrams 编著、Addison-Wesley 于 2005 年出版的“[Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries](#)”（《框架设计指南：可重用 .NET 库的约定、术语和模式》）。

■ 请参见

概念

[成员设计准则](#)

[类库开发的设计准则](#)

[参数设计](#)



.NET Framework 开发人员指南

参数数目可变的成员

[请参见](#)

数组用于将数目可变的参数传递给成员。某些语言（如 **C#**）提供了一个关键字，用于修饰传递可变参数的数组。对于不提供关键字的语言，[ParamArrayAttribute](#) 属性可提供此功能。该关键字和属性影响成员签名的最后一个参数。该参数必须是一维数组。

下面的代码示例演示定义和调用参数数目可变的方法。注意，在 `DemonstrateVariableParameters` 方法中，调用 `UseVariableParameters` 前参数不会放入数组。

Visual Basic

[复制代码](#)

```
Public Shared Sub UseVariableParameters(ParamArray list() as Integer)
    For i as Integer = 0 to list.Length - 1
        Console.WriteLine(list(i))
    Next i
    Console.WriteLine()
End Sub

Public Shared Sub DemonstrateVariableParameters()

    Manager.UseVariableParameters(1,2,3,4,5)
End Sub
```

C#

[复制代码](#)

```
public static void UseVariableParameters(params int[] list)
{
    for ( int i = 0 ; i < list.Length ; i++ )
    {
        Console.WriteLine(list[i]);
    }
    Console.WriteLine();
}

public static void DemonstrateVariableParameters()
{
    Manager.UseVariableParameters(1,2,3,4,5);
}
```

下列准则有助于了解何时使用可变数组作为参数是适合的和有益的。

如果需要最终用户传递少量元素，则考虑向数组参数添加 **params** 关键字。

通常，如果开发人员要传递很多元素，则 **params** 关键字可能用处不大，因为开发人员不太可能内联传递大量对象。

如果调用方几乎总要将输入放入数组，则不要使用 **params** 数组。

例如，字节数据通常在字节数组中存储和处理。通常，将 **params** 关键字添加到字节数组参数不能解决问题，因为开发人员通常不使用还未存储到字节数组中的单个字节。

如果数组由采用 **params** 数组参数的成员进行了修改，则不要使用 **params** 数组。

公共语言运行库 (CLR) 可能已创建了一个临时数组对象。如果方法修改临时数组，则这些修改对调用方是不可用的。

考虑在简单重载中使用 **params** 关键字，即使更复杂的重载不能使用它。

在某个重载（即使不是所有重载）中使用 **params** 数组，对开发人员都是有好处的。

尽量对参数进行排序，以使它能够使用 **params** 关键字。


这意味着, 只要可能, 数组参数就应该是指定的最后一个参数。下面的代码示例演示的参数排序是不正确的。

Visual Basic [复制代码](#)

```
Overloads Public Function Add (i as Integer, j as Integer, numberBase as Int16) _  
    as Integer
```

C# [复制代码](#)

```
public int Add (int i, int j, short numberBase)
```

Visual Basic [复制代码](#)

```
Overloads Public Function Add (i as Integer, j as Integer, k as Integer, _  
    numberBase as Int16) as Integer
```

C# [复制代码](#)

```
public int Add (int i, int j, int k, short numberBase)
```

Visual Basic [复制代码](#)

```
' Can't use params array.  
Overloads Public Function Add (numbers() as Integer, numberBase as Int16) _  
    as Integer
```

C# [复制代码](#)

```
// Can't use params array.  
public int Add (int [] numbers, short numberBase)
```

这些参数应重新排序, 如下所示:

Visual Basic [复制代码](#)

```
Overloads Public Function Add (numberBase as Int16, i as Integer, j as Integer) _  
    as Integer
```

C# [复制代码](#)

```
public int Add (short numberBase, int i, int j)
```

Visual Basic [复制代码](#)

```
Overloads Public Function Add (numberBase as Int16, i as Integer, _  
    j as Integer, k as Integer) as Integer
```

C# [复制代码](#)

```
public int Add (short numberBase, int i, int j, int k)
```

Visual Basic [复制代码](#)

```
' Can use params array.  
Overloads Public Function Add (numberBase as Int16, _  
    ParamArray numbers() as Integer) as Integer
```

C# [复制代码](#)

```
// Can use params array.  
public int Add (short numberBase, params int [] numbers)
```

考虑在性能极为敏感的 **API** 中为使用少量参数的调用提供特殊重载和代码路径。

按照此准则, 在调用带有少量参数的成员时可以避免创建数组。参数名称应为数组参数的单数形式, 后跟一个数字后缀。下面的代码示例演示一个符合此准则的成员签名。

Visual Basic [复制代码](#)

```
Public Shared Sub WriteLine( _  
    format as String, _  
    arg0 as Object, _  
    arg1 as Object, _  
    arg2 as Object _  
)
```

C# [复制代码](#)

```
public static void WriteLine(  
    string format,  
    object arg0,  
    object arg1,  
    object arg2  
)
```

注意 **null**（在 **Visual Basic** 中为 **Nothing**）可以作为 **params** 数组参数进行传递。

在处理 **null** 数组之前，成员应对其进行检查。

不要使用 **varargs** 方法，这种方法以省略号的形式表示。

因为 **varargs** 调用约定不符合 **CLS**，所以不应在公共成员中使用。它可以在内部使用。

部分版权所有 2005 Microsoft Corporation。保留所有权利。

部分版权所有 Addison-Wesley Corporation。保留所有权利。

有关设计指南的更多信息，请参见 Krzysztof Cwalina 和 Brad Abrams 编著、Addison-Wesley 于 2005 年出版的“**Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries**”（《框架设计指南：可重用 .NET 库的约定、术语和模式》）。

请参见**概念**[成员设计准则](#)[类库开发的设计准则](#)[参数设计](#)



.NET Framework 开发人员指南

指针参数

[请参见](#)

指针是一项高级编程功能，只应该在十分注重性能的情况下使用。指针允许对内存地址进行访问。下面的准则有助于确保您的库设计可有效使用指针。

避免对指针参数进行高开销的参数检查。

您通常应该检查参数；但是，对于对性能敏感的成员，参数检查的开销通常是不值得的。

在设计带指针的成员时，要遵守与指针相关的惯常约定。

例如，成员不需要将起始索引作为参数，因为可以使用简单的指针算法来提供指针地址，该指针地址作为添加到相应起始索引位置的指针基地址。

部分版权所有 2005 Microsoft Corporation。保留所有权利。

部分版权所有 Addison-Wesley Corporation。保留所有权利。

有关设计指南的更多信息，请参见 Krzysztof Cwalina 和 Brad Abrams 编著、Addison-Wesley 于 2005 年出版的“**Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries**”（《框架设计指南：可重用 .NET 库的约定、术语和模式》）。

请参见

概念

[成员设计准则](#)

[类库开发的设计准则](#)

[参数设计](#)



.NET Framework 开发人员指南

扩展性设计

扩展性是添加或修改对象行为的能力。可以使用很多种不同的机制使库成为可扩展库。每种不同的机制都有其自己的优缺点。

部分版权所有 2005 Microsoft Corporation。保留所有权利。

部分版权所有 Addison-Wesley Corporation。保留所有权利。

有关设计指南的更多信息, 请参见 Krzysztof Cwalina 和 Brad Abrams 编著、Addison-Wesley 于 2005 年出版的 “Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries” (《框架设计指南: 可重用 .NET 库的约定、术语和模式》)。

■ 本节内容

[未密封的类](#)

描述未密封类的设计准则。

[受保护的成员](#)

描述实现受保护成员的准则。

[事件和回调](#)

描述使用事件和回调方法的准则。

[虚成员](#)

描述使成员成为虚成员的准则。

[抽象类型和接口](#)

描述创建抽象类型和接口的准则。

[用于实现抽象的基类](#)

描述使用基类实现抽象的准则。

[通过密封类限制扩展性](#)

描述密封类和成员的准则。

■ 相关章节

[.NET Framework 类库参考](#)

描述构成 .NET Framework 的每一个公共类。

[类库开发的设计准则](#)

描述类库开发的最佳做法。



.NET Framework 开发人员指南

未密封的类

[请参见](#)

其他类可以从未密封类继承。许多其他扩展性机制（如受保护成员和虚成员）要求类是未密封的。这些附加机制为添加或自定义类型的功能提供了功能强大的方式，但需要大量开销。即使没有附加扩展性机制，未密封类在许多开发方案中也非常有用。

考虑一种很好的方式，即采用不具有虚成员或受保护成员的未密封类，向框架提供资源开销少的适用扩展性。

默认情况下，大多数类都不应是密封的。这样，开发人员才能根据他们的特定情况来自定义类。例如，如果某个类是未密封的，则开发人员可以向从该类派生的类型添加一个构造函数，然后使用该构造函数将基类属性初始化为所需的值。开发人员还可以添加一些方法重载，用于传递方案特定的默认值。

部分版权所有 2005 Microsoft Corporation。保留所有权利。

部分版权所有 Addison-Wesley Corporation。保留所有权利。

有关设计指南的更多信息，请参见 Krzysztof Cwalina 和 Brad Abrams 编著、Addison-Wesley 于 2005 年出版的“**Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries**”（《框架设计指南：可重用 .NET 库的约定、术语和模式》）。

■ 请参见

概念

[类库开发的设计准则](#)

[扩展性设计](#)

[通过密封类限制扩展性](#)



.NET Framework 开发人员指南

受保护的成员

[请参见](#)

未密封类的受保护成员为开发人员提供了一种自定义类行为的方式。例如，引发事件的方法通常定义为具有受保护的可见性，以允许派生类在引发该事件之前或之后提供其他处理。

重要说明:

术语“受保护”并不意味着进行任何安全检查或调用方验证。只需通过定义声明类型的派生类，即可访问受保护的成员。

出于安全、文档和兼容性分析方面的考虑，将未密封类的受保护成员视为公共成员。任意代码都可以通过创建子类来访问受保护成员。

部分版权所有 2005 Microsoft Corporation。保留所有权利。

部分版权所有 Addison-Wesley Corporation。保留所有权利。

有关设计指南的更多信息，请参见 Krzysztof Cwalina 和 Brad Abrams 编著、Addison-Wesley 于 2005 年出版的“Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries”（《框架设计指南：可重用 .NET 库的约定、术语和模式》）。

请参见

概念

[类库开发的设计准则](#)

[扩展性设计](#)



.NET Framework 开发人员指南

事件和回调

[请参见](#)

回调方法是一种在操作或活动完成时由委托自动调用的方法。例如，某一异步设计模式使用 [AsyncCallback](#) 委托指定在异步操作完成时执行的代码。该设计模式用在 [BeginWrite](#) 方法中，该方法使用回调处理异步写操作的结果。

事件是与回调类似的机制。事件允许在特定的情况下执行用户指定的代码，这些情况通常涉及状态更改或活动的开始或结束。事件比回调易于使用，因为语言语法和工具为识别和处理事件提供了统一的编码做法。此外，事件由称为事件处理程序的委托处理，这些委托具有定义完善的签名模式。有关事件的更多信息，请参见[事件设计](#)。

下列准则可帮助确保您的库设计根据最佳做法使用事件和回调。

避免在对性能敏感的 API 中使用回调。

尽管回调和事件对于许多开发人员来说更易于理解和使用，但从性能和内存消耗的角度看，它们不如[虚成员](#)可取。

要了解调用委托将会执行任意代码，这可能会造成安全性、正确性和兼容性方面的问题。

事件和回调允许在公共语言运行库 (CLR) 的上下文中执行任意代码。在检查代码和安全性时，要仔细检查这些扩展点以确定是否存在安全漏洞。

部分版权所有 2005 Microsoft Corporation。保留所有权利。

部分版权所有 Addison-Wesley Corporation。保留所有权利。

有关设计指南的更多信息，请参见 Krzysztof Cwalina 和 Brad Abrams 编著、Addison-Wesley 于 2005 年出版的“[Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries](#)”（《框架设计指南：可重用 .NET 库的约定、术语和模式》）。

■ 请参见

[概念](#)

[扩展性设计](#)

[类库开发的设计准则](#)



.NET Framework 开发人员指南

虚成员

[请参见](#)

virtual（在 **Visual Basic** 中为 **Overridable**）成员允许通过提供该成员的不同实现来更改成员行为。如果类型的派生类要处理给定的特定情况，通常会使用虚成员。例如，[WebRequest](#) 类定义向任何统一资源定位符 (URI) 发送请求的功能。[FtpWebRequest](#) 类是 [WebRequest](#) 的派生类，前者重写后者的虚方法，以处理向使用文件传输协议 (FTP) 方案的 URI 发送请求。

虚成员的性能高于回调和事件，但是不比非虚方法高。

必要时才可使用虚成员，还应对设计、测试和维护虚成员所需的开销有所了解。

在不同版本之间更改虚成员的实现可能会导致不易察觉的版本不兼容。因此，正确设计和彻底测试虚成员的开销是很大的。

关于可访问性，首选使用受保护成员，而不是公共成员。公共成员应通过调用受保护虚成员提供扩展性（如果需要）。

不涉及继承的所有情况都需要使用的成员应该是公共的。

部分版权所有 2005 Microsoft Corporation。保留所有权利。

部分版权所有 Addison-Wesley Corporation。保留所有权利。

有关设计指南的更多信息，请参见 Krzysztof Cwalina 和 Brad Abrams 编著、Addison-Wesley 于 2005 年出版的“[Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries](#)”（《框架设计指南：可重用 .NET 库的约定、术语和模式》）。

请参见

概念

[类库开发的设计准则](#)

[扩展性设计](#)



.NET Framework 开发人员指南

抽象类型和接口

[请参见](#)

抽象类型和接口是用于指定编程抽象的两种机制。抽象指定继承者或实施者必须遵循的一个协定。抽象类型可以选择提供实现的详细信息；而接口不能提供实现的任何详细信息。

除非通过开发若干使用抽象的具体实现和 API 对抽象进行了测试，否则不要提供抽象。

如果在实际方案中未对抽象进行测试就提供了抽象，将很有可能遗漏一些设计问题；而且，对这些问题的修复很可能甚至必然导致在以后的版本中引入兼容性问题。

在设计抽象时，则要在抽象类与接口之间谨慎选择。

有关选择抽象类型还是接口的详细讨论，请参见[在类和接口之间选择](#)。

考虑为抽象的具体实现提供引用测试。通过这样的测试，用户应可以测试出抽象的实现是否正确地履行了协定。

通过引用测试，可以验证是否正确实现了某一接口。例如，对 [ICollection\(T\)](#) 进行的测试可以验证：在对实现该接口的实例两次调用 **Add** 方法后，**Count** 属性是否增加 2。

部分版权所有 2005 Microsoft Corporation。保留所有权利。

部分版权所有 Addison-Wesley Corporation。保留所有权利。

有关设计指南的更多信息，请参见 Krzysztof Cwalina 和 Brad Abrams 编著、Addison-Wesley 于 2005 年出版的“**Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries**”（《框架设计指南：可重用 .NET 库的约定、术语和模式》）。

■ 请参见

概念

[类库开发的设计准则](#)

[扩展性设计](#)



.NET Framework 开发人员指南

用于实现抽象的基类

请参见

用于实现抽象的基类是设计用于帮助开发人员实现抽象类和接口（抽象）的类。这些基类为抽象提供了一些实现细节，在某些情况下不用继承就可以使用它们。例如，可以使用 [Collection\(T\)](#) 创建一个集合，也可以从其进行继承以定义强类型集合类。

下面的代码示例演示如何使用 [Collection\(T\)](#) 类创建一个强类型集合对象。

Visual Basic

[复制代码](#)

```
Public Class PointManager
    Implements IEnumerable

    Private pointCollection As Collection(Of Point) = New Collection(Of Point)

    Public Sub AddPoint(ByVal p As Point)
        pointCollection.Add(p)
    End Sub

    Public Function RemovePoint(ByVal p As Point) As Boolean
        Return pointCollection.Remove(p)
    End Function

    Public Function GetEnumerator() As IEnumerator _
        Implements IEnumerable.GetEnumerator

        Return pointCollection.GetEnumerator
    End Function
End Class
```

C#

[复制代码](#)

```
public class PointManager : IEnumerable
{
    Collection<Point> pointCollection = new Collection<Point>();

    public void AddPoint(Point p)
    {
        pointCollection.Add(p);
    }
    public bool RemovePoint(Point p)
    {
        return pointCollection.Remove(p);
    }
    public IEnumerator GetEnumerator()
    {
        return pointCollection.GetEnumerator();
    }
}
```

下面的代码示例演示如何使用 [Collection\(T\)](#) 类定义一个强类型集合。

Visual Basic

[复制代码](#)

```
Public Class PointCollection
    Inherits Collection(Of Point)
End Class
```

C#

[复制代码](#)

```
public class PointCollection : Collection<Point> {}
```

[CollectionBase](#) 类是 .NET Framework 基类的另一个示例。该类可帮助开发人员实现非泛型集合。与 [Collection\(T\)](#) 不同的是，[CollectionBase](#) 不能直接使用。

只有用于实现抽象的基类能为使用库的开发人员带来很大价值时，才应将这些基类作为库的一部分予以提供。如果某一基类只用于帮助实现库，则该基类不应是公共可见的。若要在内部使用基类来简化库开发工作，公共成员应将工作委托给该基类，而不是从该基类继承。

如果基类设计用于公共 API 中，则在命名时避免为基类使用 Base 后缀。

如果库将基类作为返回类型或参数类型公开，则该基类不应带有 Base 后缀。

部分版权所有 2005 Microsoft Corporation。保留所有权利。

部分版权所有 Addison-Wesley Corporation。保留所有权利。

有关设计指南的更多信息，请参见 Krzysztof Cwalina 和 Brad Abrams 编著、Addison-Wesley 于 2005 年出版的“**Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries**”（《框架设计指南：可重用 .NET 库的约定、术语和模式》）。

■ 请参见

概念

[类库开发的设计准则](#)

[扩展性设计](#)



.NET Framework 开发人员指南

通过密封类限制扩展性

[请参见](#)

可以使用密封来限制开发人员可以扩展您的框架的方式。如果密封了某个类，则其他类不能从该类继承。如果密封了某个成员，则派生类不能重写该成员的实现。默认情况下，不应密封类型和成员。密封可防止对库的类型和成员进行自定义，也会影响某些开发人员对可用性的认识。此外，使用面向对象的框架还有一个重要优点，即扩展性。如果所作决定不能体现出这一优点，请仔细权衡。

除非有充分的理由，否则不要密封类。

不要因为预见不到需要扩展某个类的情形，就认为将其密封起来是合适的。如果类满足如下条件，则应将其密封：

- 类是静态类。
- 类包含带有安全敏感信息的继承的受保护成员。
- 类继承多个虚成员，并且密封每个成员的开发和测试开销明显大于密封整个类。
- 类是一个要求使用反射进行快速搜索的属性。密封属性可提高反射在检索属性时的性能。

不要在密封类型中声明受保护成员或虚成员。

如果类型是密封的，则它不能有派生类。受保护成员只能从派生类进行访问，虚成员也只能在派生类中重写。

考虑密封重写的成员。

可以使用这种方式来确保派生类不会修改或跳过当前类和所有派生类所需的行为。

部分版权所有 2005 Microsoft Corporation。保留所有权利。

部分版权所有 Addison-Wesley Corporation。保留所有权利。

有关设计指南的更多信息，请参见 Krzysztof Cwalina 和 Brad Abrams 编著、Addison-Wesley 于 2005 年出版的“**Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries**”（《框架设计指南：可重用 .NET 库的约定、术语和模式》）。

■ 请参见

概念

[类库开发的设计准则](#)

[扩展性设计](#)

[未密封的类](#)



.NET Framework 开发人员指南

异常设计准则

异常是报告错误的标准机制。应用程序和库不应使用返回代码来传递错误信息。异常的采用增进了框架设计的一致性，允许无返回类型的成员（如构造函数）报告错误。异常还允许程序处理错误或根据需要终止运行。默认行为是在应用程序不处理引发的异常时，终止应用程序。有关 .NET Framework 中的异常的详细介绍，请参见[处理和引发异常](#)。

部分版权所有 2005 Microsoft Corporation。保留所有权利。

部分版权所有 Addison-Wesley Corporation。保留所有权利。

有关设计指南的更多信息，请参见 Krzysztof Cwalina 和 Brad Abrams 编著、Addison-Wesley 于 2005 年出版的“Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries”（《框架设计指南：可重用 .NET 库的约定、术语和模式》）。

■ 本节内容

[异常引发](#)

描述引发异常的准则。

[异常处理](#)

描述捕捉异常的准则。

[捕捉和引发标准异常类型](#)

描述 .NET Framework 所提供的常见异常的处理准则。

[设计自定义异常](#)

描述新异常类型的定义准则。

[异常和性能](#)

描述使用设计模式避免与异常相关的性能问题的准则。

■ 相关章节

[.NET Framework 类库参考](#)

描述构成 .NET Framework 的每一个公共类。

[类库开发的设计准则](#)

描述类库开发的最佳做法。



.NET Framework 开发人员指南

异常引发

[请参见](#)

当某一成员无法成功执行它应执行的操作时，将引发异常。这称为执行故障。例如，如果 [Connect](#) 方法无法连接到指定的远程终结点，则这就是一个执行故障，将有一个异常被引发。

下列准则可帮助确保在适当时引发异常。

不要返回错误代码。异常是报告框架中的错误的主要手段。

[异常设计准则](#)讨论了使用异常的许多好处。

尽可能不对正常控制流使用异常。除了系统故障及可能导致争用状态的操作之外，框架设计人员还应设计一些 **API** 以便用户可以编写不引发异常的代码。例如，可以提供一种在调用成员之前检查前提条件的方法，以便用户可以编写不引发异常的代码。

下面的代码示例演示如何进行测试以防止在消息字符串为 **null**（在 Visual Basic 中为 **Nothing**）时引发异常。

Visual Basic

 [复制代码](#)

```
Public Class Doer

    ' Method that can potential throw exceptions often.
    Public Shared Sub ProcessMessage(ByVal message As String)
        If (message = Nothing) Then
            Throw New ArgumentNullException("message")
        End If
    End Sub

    ' Other methods...
End Class

Public Class Tester

    Public Shared Sub TesterDoer(ByVal messages As ICollection(Of String))
        For Each message As String In messages
            ' Test to ensure that the call
            ' won't cause the exception.
            If (Not (message) Is Nothing) Then
                Doer.ProcessMessage(message)
            End If
        Next
    End Sub
End Class
```

C#

 [复制代码](#)

```
public class Doer
{
    // Method that can potential throw exceptions often.
    public static void ProcessMessage(string message)
    {
        if (message == null)
        {
            throw new ArgumentNullException("message");
        }
    }
    // Other methods...
}

public class Tester
{
    public static void TesterDoer(ICollection<string> messages)
    {
        foreach (string message in messages)
        {
            // Test to ensure that the call
```

```
// won't cause the exception.
if (message != null)
{
    Doer.ProcessMessage(message);
}
}
```

有关可以减少异常引发数量的设计方案的其他信息, 请参见[异常和性能](#)。

不要包含可以根据某一选项引发或不引发异常的公共成员。

例如, 不要定义如下所示的成员:

Visual Basic

[复制代码](#)

```
Private Function ParseUri(ByVal uriValue As String, ByVal throwOnError As Boolean) As Uri
```

C#

[复制代码](#)

```
Uri ParseUri(string uriValue, bool throwOnError)
```

不要包含将异常作为返回值或输出参数返回的公共成员。

此项准则适用于公共可见的成员。使用私有帮助器方法构造和初始化异常是可以接受的。

考虑使用异常生成器方法。从不同的位置引发同一异常会经常发生。为了避免代码膨胀, 请使用帮助器方法创建异常并初始化其属性。

帮助器方法不得引发异常, 否则堆栈跟踪将无法正确反映出引发异常的调用堆栈。

不要从异常筛选器块中引发异常。当异常筛选器引发异常时, 公共语言运行库 (CLR) 将捕获该异常, 然后该筛选器返回 **false**。此行为与筛选器显式执行和返回 **false** 的行为无法区分, 因此很难调试。

有些语言 (如 C#) 不支持异常筛选器。

避免从 **finally** 块中显式引发异常。可以接受因调用引发异常的方法而隐式引发的异常。

部分版权所有 2005 Microsoft Corporation。保留所有权利。

部分版权所有 Addison-Wesley Corporation。保留所有权利。

有关设计指南的更多信息, 请参见 Krzysztof Cwalina 和 Brad Abrams 编著、Addison-Wesley 于 2005 年出版的 “Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries” (《框架设计指南: 可重用 .NET 库的约定、术语和模式》)。

■ 请参见

概念

[类库开发的设计准则](#)

[选择要引发的正确异常类型](#)

[异常设计准则](#)



.NET Framework 开发人员指南

选择要引发的正确异常类型

[请参见](#)

下列设计准则可帮助您确保正确地使用现有异常，并在适当的时候创建对您的库有价值的新异常。

考虑引发 `System` 命名空间中的现有异常，而不是创建自定义异常类型。

有关 .NET Framework 所提供的最常用异常类型的详细准则，请参见[捕捉和引发标准异常类型](#)。

如果错误状态可以通过不同于现有任何其他异常的方法以编程方式进行处理，则要创建并引发自定义异常。否则，引发一个现有异常。

有关创建自定义异常的详细准则，请参见[设计自定义异常](#)。

引发适当的最具体（派生程度最大）的异常。例如，如果某方法收到一个 `null`（在 Visual Basic 中为 `Nothing`）参数，则该方法应引发 `System.ArgumentNullException`，而不是引发该异常的基类型 `System.ArgumentException`。

部分版权所有 2005 Microsoft Corporation。保留所有权利。

部分版权所有 Addison-Wesley Corporation。保留所有权利。

有关设计指南的更多信息，请参见 Krzysztof Cwalina 和 Brad Abrams 编著、Addison-Wesley 于 2005 年出版的“[Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries](#)”（《框架设计指南：可重用 .NET 库的约定、术语和模式》）。

请参见

概念

[类库开发的设计准则](#)

[异常设计准则](#)

[异常引发](#)



.NET Framework 开发人员指南

包装异常

[请参见](#)

若要包装某个异常，请将该异常指定为某个新异常的内部异常，然后引发该新异常。仅在原始异常对于接收该异常的用户没有意义，或异常的调用堆栈易混淆或没有用处时，才应进行这种处理。例如，考虑一个为基于 XML 的配置文件提供管理功能的库。配置文件管理器在内部使用 XML 读取器来读取文件。如果配置文件的格式错误，则 XML 读取器可能会引发异常，其中包括一条消息和用于 XML 读取器及其支持类型的调用堆栈详细信息，而这些对于应用程序用户都没有意义。在这种情况下，就适合由配置文件管理器来包装 XML 读取器异常并再次引发一个指示问题真实性质的新异常。

下面的准则有助于确保仅在适当时正确包装异常。

如果低层异常在高层操作的上下文中没有意义，则考虑在更适当的异常中包装在低层引发的特定异常。

一般不应这样做，因为会使调试更加困难。如果确定低层不是错误的真实来源，才适合这样做。

避免捕捉和包装非特定异常。

这种处理会隐藏错误，因此需要避免。此规则的例外情况包括：相对于原始异常的实际类型，包装异常所传递的严重情况对调用方更有意义。例如，[TypeInitializationException](#) 异常包装从静态构造函数引发的所有异常。

包装异常时务必要指定内部异常。

这样，工具可显示问题的基础详细信息，也有助于调试代码。下面的代码示例演示如何包装异常。

Visual Basic

[复制代码](#)

```
Public Sub SendMessages()  
    Try  
        EstablishConnection()  
    Catch e As System.Net.Sockets.SocketException  
        Throw New CommunicationFailureException("Cannot access remote computer.", e)  
    End Try  
End Sub
```

C#

[复制代码](#)

```
public void SendMessages()  
{  
    try  
    {  
        EstablishConnection();  
    }  
    catch (System.Net.Sockets.SocketException e)  
    {  
        throw new CommunicationFailureException(  
            "Cannot access remote computer.",  
            e);  
    }  
}
```

部分版权所有 2005 Microsoft Corporation。保留所有权利。

部分版权所有 Addison-Wesley Corporation。保留所有权利。

有关设计指南的更多信息，请参见 Krzysztof Cwalina 和 Brad Abrams 编著、Addison-Wesley 于 2005 年出版的“Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries”（《框架设计指南：可重用 .NET 库的约定、术语和模式》）。

请参见

概念

[类库开发的设计准则](#)

[异常设计准则](#)

[异常引发](#)



.NET Framework 开发人员指南

错误信息设计

[请参见](#)

下面的准则有助于确保异常消息有意义且格式正确。

在引发异常时为开发人员提供丰富且有意义的消息文本。消息应说明导致异常的原因并清楚描述避免该异常需采取的操作。

生成库时，消息应针对开发人员设计。

不要在不要相应权限的异常消息中透露安全敏感信息。

有关安全库设计的更多信息，请参见[编写安全类库](#)。

如果希望使用不同语言的开发人员使用您的组件，则应考虑对您的组件所引发的异常消息进行本地化。

有关开发可本地化的代码的更多信息，请参见[本地化](#)。

部分版权所有 2005 Microsoft Corporation。保留所有权利。

部分版权所有 Addison-Wesley Corporation。保留所有权利。

有关设计指南的更多信息，请参见 Krzysztof Cwalina 和 Brad Abrams 编著、Addison-Wesley 于 2005 年出版的“Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries”（《框架设计指南：可重用 .NET 库的约定、术语和模式》）。

■ 请参见

概念

[类库开发的设计准则](#)

[选择要引发的正确异常类型](#)

[异常设计准则](#)

[异常引发](#)



.NET Framework 开发人员指南

异常处理

[请参见](#)

下面的准则有助于确保库正确处理异常。

不要通过在框架代码中捕捉非特定异常（如 `System.Exception`、`System.SystemException` 等）来处理错误。

如果捕捉异常是为了再次引发或传输给其他线程，则可以捕捉这些异常。下面的代码示例演示的异常处理是不正确的。

Visual Basic

[复制代码](#)

```
Public Class BadExceptionHandlingExample1

    Public Sub DoWork()
        ' Do some work that might throw exceptions.
    End Sub

    Public Sub MethodWithBadHandler()
        Try
            DoWork()
        Catch e As Exception
            ' Handle the exception and
            ' continue executing.
        End Try
    End Sub
End Class
```

C#

[复制代码](#)

```
public class BadExceptionHandlingExample1
{
    public void DoWork()
    {
        // Do some work that might throw exceptions.
    }
    public void MethodWithBadHandler()
    {
        try
        {
            DoWork();
        }
        catch (Exception e)
        {
            // Handle the exception and
            // continue executing.
        }
    }
}
```

避免通过在应用程序代码中捕捉非特定异常（如 `System.Exception`、`System.SystemException` 等）来处理错误。某些情况下，可以在应用程序中处理错误，但这种情况极少。

应用程序不应该处理异常，否则可能导致意外状态或可利用状态。如果不能预知所有可能的异常原因，也不能确保恶意代码不能利用产生的应用程序状态，则应该允许应用程序终止，而不是处理异常。

如果捕捉异常是为了传输异常，则不要排除任何特殊异常。

只捕捉能够合法处理的异常，而不要在 `catch` 子句中创建特殊异常的列表。在非特定异常处理程序中，不能处理的异常不应视为特殊处理的特殊情况。下面的代码示例演示对以再次引发为目的的特殊异常进行的不正确测试。

Visual Basic

[复制代码](#)

```
Public Class BadExceptionHandlingExample2
```



```
Public Sub DoWork()  
    ' Do some work that might throw exceptions.  
End Sub  
  
Public Sub MethodWithBadHandler()  
    Try  
        DoWork()  
    Catch e As Exception  
        If TypeOf e Is StackOverflowException Or _  
            TypeOf e Is OutOfMemoryException Then  
            Throw  
        End If  
    End Try  
End Sub  
End Class
```

C#

 [复制代码](#)

```
public class BadExceptionHandlingExample2  
{  
    public void DoWork()  
    {  
        // Do some work that might throw exceptions.  
    }  
    public void MethodWithBadHandler()  
    {  
        try  
        {  
            DoWork();  
        }  
        catch (Exception e)  
        {  
            if (e is StackOverflowException ||  
                e is OutOfMemoryException)  
                throw;  
            // Handle the exception and  
            // continue executing.  
        }  
    }  
}
```

如果了解特定异常在给定上下文中引发的条件，请考虑捕捉这些异常。

应该只捕捉可以从中恢复的异常。例如，试图打开不存在的文件而导致的 [FileNotFoundException](#) 可以由应用程序处理，因为应用程序可以将问题传达给用户，并允许用户指定其他文件名或创建该文件。如果打开文件的请求会生成 [ExecutionEngineException](#)，则不应该处理该请求，因为没有任何把握可以了解该异常的基础原因，应用程序也无法确保继续执行是安全的。

不要过多使用 `catch`。通常应允许异常在调用堆栈中往上传播。

捕捉无法合法处理的异常会隐藏关键的调试信息。

使用 `try-finally` 并避免将 `try-catch` 用于清理代码。在书写规范的异常代码中，`try-finally` 远比 `try-catch` 更为常用。

使用 `catch` 子句是为了允许处理异常（例如，通过纪录非致命错误）。无论是否引发了异常，使用 `finally` 子句即可执行清理代码。如果分配了昂贵或有限的资源（如数据库连接或流），则应将释放这些资源的代码放置在 `finally` 块中。

捕捉并再次引发异常时，首选使用空引发。这是保留异常调用堆栈的最佳方式。

下面的代码示例演示一个可引发异常的方法。此方法在后面示例中引用。

Visual Basic

 [复制代码](#)

```
Public Sub DoWork(ByVal anObject As Object)  
    ' Do some work that might throw exceptions.  
    If (anObject = Nothing) Then
```

```

        Throw New ArgumentNullException("anObject", "Specify a non-null argument.")
    End If
    ' Do work with o.
End Sub

```

C#

 [复制代码](#)

```

public void DoWork(Object anObject)
{
    // Do some work that might throw exceptions.
    if (anObject == null)
    {
        throw new ArgumentNullException("anObject",
            "Specify a non-null argument.");
    }
    // Do work with o.
}

```

下面的代码示例演示捕捉一个异常，并在再次引发该异常时对它进行错误的指定。这会使堆栈跟踪指向再次引发作为错误位置，而不是指向 `DoWork` 方法。

Visual Basic

 [复制代码](#)

```

Public Sub MethodWithBadCatch(ByVal anObject As Object)
    Try
        DoWork(anObject)

    Catch e As ArgumentNullException
        System.Diagnostics.Debug.Write(e.Message)
        ' This is wrong.
        Throw e
        ' Should be this:
        ' throw
    End Try
End Sub

```

C#

 [复制代码](#)

```

public void MethodWithBadCatch(Object anObject)
{
    try
    {
        DoWork(anObject);
    }
    catch (ArgumentNullException e)
    {
        System.Diagnostics.Debug.Write(e.Message);
        // This is wrong.
        throw e;
        // Should be this:
        // throw;
    }
}

```

不要使用无参数 `catch` 块来处理不符合 CLS 的异常（不是从 `System.Exception` 派生的异常）。支持不是从 `Exception` 派生的异常的语言可以处理这些不符合 CLS 的异常。

.NET Framework 2.0 版在 [Exception](#) 的派生类中包装了不符合 CLS 的异常。

部分版权所有 2005 Microsoft Corporation。保留所有权利。

部分版权所有 Addison-Wesley Corporation。保留所有权利。

有关设计指南的更多信息，请参见 Krzysztof Cwalina 和 Brad Abrams 编著、Addison-Wesley 于 2005 年出版的“Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries”（《框架设计指南：可重用 .NET 库的约定、术语和模式》）。

☐ 请参见

概念

[类库开发的设计准则](#)

[异常设计准则](#)



.NET Framework 开发人员指南

捕捉和引发标准异常类型

[请参见](#)

下面的准则介绍 .NET Framework 所提供的某些最常用异常的最佳做法。有关框架所提供的异常类的完整列表, 请参见 [.NET Framework 类库参考](#) 文档。

Exception 和 SystemException

不要引发 `System.Exception` 或 `System.SystemException`。

不要在框架代码中捕捉 `System.Exception` 或 `System.SystemException`, 除非打算再次引发。

避免捕捉 `System.Exception` 或 `System.SystemException`, 在顶级异常处理程序中除外。

ApplicationException

一定要从 `T:System.Exception` 类而不是 `T:System.ApplicationException` 类派生自定义异常。

最初考虑自定义异常应该从 [ApplicationException](#) 类派生; 但是并未发现这样有很大意义。有关更多信息, 请参见[处理异常的最佳做法](#)。

InvalidOperationException

如果处于不适当的状态, 则引发 `System.InvalidOperationException` 异常。如果没有向属性集或方法调用提供适当的对象当前状态, 则应引发 `System.InvalidOperationException`。例如, 向已打开用于读取的 `System.IO.FileStream` 写入时, 应引发 `System.InvalidOperationException` 异常。

一组相关对象的组合状态对于操作无效时, 也应引发此异常。

ArgumentException、ArgumentNullException 和 ArgumentOutOfRangeException

如果向成员传递了错误的参数, 则引发 `System.ArgumentException` 或其子类型之一。如果适用, 首选派生程度最高的异常类型。

下面的代码示例演示当参数为 `null` (在 Visual Basic 中为 `Nothing`) 时引发异常。

Visual Basic

[复制代码](#)

```
If (anObject = Nothing) Then
    Throw New ArgumentNullException("anObject", "Specify a non-null argument.")
End If
```

C#

[复制代码](#)

```
if (anObject == null)
{
    throw new ArgumentNullException("anObject",
        "Specify a non-null argument.");
}
```

在引发 `System.ArgumentException` 或其派生类型之一时, 设置 `System.ArgumentException.ParamName` 属性。此属性存储导致引发异常的参数的名称。请注意, 可以使用构造函数重载之一设置该属性。

使用属性 setter 的隐式值参数的名称的值。

下面的代码示例演示一个属性, 该属性在调用方传递 `null` 参数时引发异常。

Visual Basic

[复制代码](#)

```

Public Property Address() As IPAddress
    Get
        Return IPAddr
    End Get
    Set(ByVal value As IPAddress)
        If IsNothing(value) Then
            Throw New ArgumentNullException("value")
        End If
        IPAddr = value
    End Set
End Property

```

C#

 [复制代码](#)

```

public IPAddress Address
{
    get
    {
        return address;
    }
    set
    {
        if(value == null)
        {
            throw new ArgumentNullException("value");
        }
        address = value;
    }
}

```

不要允许公开调用的 API 显式或隐式引发 `System.NullReferenceException`、`System.AccessViolationException`、`System.InvalidCastException` 或 `System.IndexOutOfRangeException`。进行参数检查以避免引发这些异常。引发这些异常会公开方法的实现细节，这些细节可能会随时间发生更改。

❑ `StackOverflowException`

不要显式引发 `System.StackOverflowException`。此异常只应由公共语言运行库 (CLR) 显式引发。

不要捕捉 `System.StackOverflowException`。

以编程方式处理堆栈溢出极为困难。应允许此异常终止进程并使用调试确定问题的根源。

❑ `OutOfMemoryException`

不要显式引发 `System.OutOfMemoryException`。此异常只应由 CLR 基础结构引发。

❑ `ComException` 和 `SEHException`

不要显式引发 `System.Runtime.InteropServices.COMException` 或 `System.Runtime.InteropServices.SEHException`。这些异常只应由 CLR 基础结构引发。

不要显式捕捉 `System.Runtime.InteropServices.SEHException`。

❑ `ExecutionEngineException`

不要显式引发 `System.ExecutionEngineException`。

部分版权所有 2005 Microsoft Corporation。保留所有权利。

部分版权所有 Addison-Wesley Corporation。保留所有权利。

有关设计指南的更多信息，请参见 Krzysztof Cwalina 和 Brad Abrams 编著、Addison-Wesley 于 2005 年出版的 “Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET

Libraries”（《框架设计指南：可重用 .NET 库的约定、术语和模式》）。

■ 请参见

概念

[类库开发的设计准则](#)

[选择要引发的正确异常类型](#)

[异常设计准则](#)



.NET Framework 开发人员指南

设计自定义异常

[请参见](#)

下列指南有助于确保正确设计您的自定义异常。

避免使用深的异常层次结构。

有关更多信息, 请参见[类型和命名空间](#)。

一定要从 `System.Exception` 或其他常见基本异常之一派生异常。

请注意, [捕捉和引发标准异常类型](#) 具有一个指南, 指出不应从 [ApplicationException](#) 派生自定义异常。

异常类名称一定要以后缀 `Exception` 结尾。

一致的命名约定有助于降低新库的学习曲线。

应使异常可序列化。异常必须可序列化才能跨越应用程序域和远程处理边界正确工作。

有关使类型可序列化的更多信息, 请参见[序列化](#)。

一定要在所有异常上都提供（至少是这样）下列常见构造函数。确保参数的名称和类型与在下面的代码示例中使用的那些相同。

Visual Basic

[复制代码](#)

```
Public Class NewException
    Inherits BaseException
    Implements ISerializable

    Public Sub New()
        MyBase.New()
        ' Add implementation.
    End Sub

    Public Sub New(ByVal message As String)
        MyBase.New()
        ' Add implementation.
    End Sub

    Public Sub New(ByVal message As String, ByVal inner As Exception)
        MyBase.New()
        ' Add implementation.
    End Sub

    ' This constructor is needed for serialization.
    Protected Sub New(ByVal info As SerializationInfo, ByVal context As StreamingContext)
        MyBase.New()
        ' Add implementation.
    End Sub
End Class
```

C#

[复制代码](#)

```
public class NewException : BaseException, ISerializable
{
    public NewException()
    {
        // Add implementation.
    }
    public NewException(string message)
    {
        // Add implementation.
    }
    public NewException(string message, Exception inner)
    {
        // Add implementation.
    }
}
```

```
    }

    // This constructor is needed for serialization.
    protected NewException(SerializationInfo info, StreamingContext context)
    {
        // Add implementation.
    }
}
```

考虑提供异常属性，以便可以以编程方式访问除消息字符串之外与异常相关的额外信息。

部分版权所有 2005 Microsoft Corporation。保留所有权利。

部分版权所有 Addison-Wesley Corporation。保留所有权利。

有关设计指南的更多信息，请参见 Krzysztof Cwalina 和 Brad Abrams 编著、Addison-Wesley 于 2005 年出版的“Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries”（《框架设计指南：可重用 .NET 库的约定、术语和模式》）。

■ 请参见

概念

[类库开发的设计准则](#)

[选择要引发的正确异常类型](#)

[异常设计准则](#)



.NET Framework 开发人员指南

异常和性能

[请参见](#)

引发异常可能会对性能造成不良的影响。对于经常性执行失败的代码，可以使用设计模式最大限度地减少性能问题。本主题描述两种设计模式，在异常可能会严重影响性能时使用这两种模式很有帮助。

不要由于担心异常可能会对性能造成不良影响而使用错误代码。

利用设计来减少性能问题。在本主题中描述了两种设计模式。

对于可能在常见方案中引发异常的成员，可以考虑使用 **Tester-Doer** 模式来避免与异常相关的性能问题。

Tester-Doer 模式将可能引发异常的调用划分为两部分：**Tester** 和 **Doer**。**Tester** 对可能导致 **Doer** 引发异常的状态执行测试。测试恰好插入在引发异常的代码之前，从而防范异常发生。

下面的代码示例演示此模式的 **Doer** 部分。该示例包含一个方法，在向该方法传递 **null**（在 **Visual Basic** 中为 **Nothing**）值时该方法将引发异常。如果频繁地调用该方法，就可能会对性能造成不良影响。

Visual Basic

[复制代码](#)

```
Public Class Doer

    ' Method that can potential throw exceptions often.
    Public Shared Sub ProcessMessage(ByVal message As String)
        If (message = Nothing) Then
            Throw New ArgumentNullException("message")
        End If
    End Sub

    ' Other methods...
End Class
```

C#

[复制代码](#)

```
public class Doer
{
    // Method that can potential throw exceptions often.
    public static void ProcessMessage(string message)
    {
        if (message == null)
        {
            throw new ArgumentNullException("message");
        }
    }
    // Other methods...
}
```

下面的代码示例演示此模式的 **Tester** 部分。该方法利用一个测试来避免在 **Doer** 将引发异常时调用 **Doer** (**ProcessMessage**)。

Visual Basic

[复制代码](#)

```
Public Class Tester

    Public Shared Sub TesterDoer(ByVal messages As ICollection(Of String))
        For Each message As String In messages
            ' Test to ensure that the call
            ' won't cause the exception.
            If (Not (message) Is Nothing) Then
                Doer.ProcessMessage(message)
            End If
        Next
    End Sub
End Class
```

C#

[复制代码](#)

```
public class Tester
{
    public static void TesterDoer(ICollection<string> messages)
    {
        foreach (string message in messages)
        {
            // Test to ensure that the call
            // won't cause the exception.
            if (message != null)
            {
                Doer.ProcessMessage(message);
            }
        }
    }
}
```

注意，当在测试涉及可变对象的多线程应用程序中使用该模式时，必须解决可能出现的争用状态问题。线程可以在测试之后且 **Doer** 执行之前更改可变对象的状态。使用线程同步技术可以解决这些问题。

对于可能在常见方案中引发异常的成员，可以考虑使用 **TryParse** 模式来避免与异常相关的性能问题。

若要实现 **TryParse** 模式，需要为执行可在常见方案中引发异常的操作提供两种不同的方法。第一种方法 **X**，执行该操作并在适当时引发异常。第二种方法 **TryX**，不引发异常，而是返回一个 [Boolean](#) 值以指示成功还是失败。由对 **TryX** 的成功调用所返回的任何数据都通过使用 **out**（在 **Visual Basic** 中为 **ByRef**）参数予以返回。[Parse](#) 和 [TryParse](#) 方法就是此模式的示例。

为每个使用 **TryParse** 模式的成员提供一个引发异常的成员。

只提供 **TryX** 方法几乎在任何时候都不是正确的设计，因为使用该方法需要了解 **out** 参数。此外，对于大多数常见方案来说，异常对性能的影响不会构成问题；因此应在大多数常见方案中提供易于使用的方法。

部分版权所有 2005 Microsoft Corporation。保留所有权利。

部分版权所有 Addison-Wesley Corporation。保留所有权利。

有关设计指南的更多信息，请参见 Krzysztof Cwalina 和 Brad Abrams 编著、Addison-Wesley 于 2005 年出版的“**Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries**”（《框架设计指南：可重用 .NET 库的约定、术语和模式》）。

■ 请参见

概念

[类库开发的设计准则](#)

[异常设计准则](#)



.NET Framework 开发人员指南

使用准则

[请参见](#)

本节中的准则讨论使用数组和正确实现属性的方法。还提供有关实现相等运算符和 [Equals](#) 方法的准则。

- | [数组使用指南](#)
- | [属性使用指南](#)
- | [实现 `Equals` 方法](#)
- | [Equals 和相等运算符 \(`==`\) 的实现准则](#)
- | [设计模式](#)

部分版权所有 2005 Microsoft Corporation。保留所有权利。

部分版权所有 Addison-Wesley Corporation。保留所有权利。

有关设计指南的更多信息, 请参见 Krzysztof Cwalina 和 Brad Abrams 编著、Addison-Wesley 于 2005 年出版的“[Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries](#)” (《框架设计指南: 可重用 .NET 库的约定、术语和模式》)。

请参见

概念

[类库开发的设计准则](#)



.NET Framework 开发人员指南

数组使用指南

[请参见](#)

有关数组和数组用法的一般说明, 请参见[数组](#)和 [System.Array](#) 类。

■ 数组与集合的比较

类库的设计人员可能需要就何时使用数组和何时返回集合做出困难的决定。尽管这些类型有相似的使用模型, 但它们有不同的性能特性。一般来说, 在支持使用 **Add**、**Remove** 或其他方法来处理集合时, 应使用集合。

有关使用集合的更多信息, 请参见[集合和数据结构](#)。

■ 数组的用法

不要返回数组的内部实例。这会允许调用代码更改数组。下面的示例演示数组 `badChars` 是如何被访问 `Path` 属性 (即使该属性不实现 `set` 访问器) 的任意代码所更改的。

Visual Basic

[复制代码](#)

```
Imports System
Imports System.Collections
Imports Microsoft.VisualBasic

Public Class ExampleClass
    NotInheritable Public Class Path
        Private Sub New()
            End Sub

        Private Shared badChars() As Char = {Chr(34), "<"c, ">"c}

        Public Shared Function GetInvalidPathChars() As Char()
            Return badChars
        End Function

    End Class

    Public Shared Sub Main()
        ' The following code displays the elements of the
        ' array as expected.
        Dim c As Char
        For Each c In Path.GetInvalidPathChars()
            Console.Write(c)
        Next c
        Console.WriteLine()

        ' The following code sets all the values to A.
        Path.GetInvalidPathChars()(0) = "A"c
        Path.GetInvalidPathChars()(1) = "A"c
        Path.GetInvalidPathChars()(2) = "A"c

        ' The following code displays the elements of the array to the
        ' console. Note that the values have changed.
        For Each c In Path.GetInvalidPathChars()
            Console.Write(c)
        Next c
    End Sub
End Class
```

C#

[复制代码](#)

```
using System;
using System.Collections;

public class ExampleClass
{
    public sealed class Path
```

```

{
    private Path(){}
    private static char[] badChars = {'\"', '<', '>'};
    public static char[] GetInvalidPathChars()
    {
        return badChars;
    }
}
public static void Main()
{
    // The following code displays the elements of the
    // array as expected.
    foreach(char c in Path.GetInvalidPathChars())
    {
        Console.Write(c);
    }
    Console.WriteLine();

    // The following code sets all the values to A.
    Path.GetInvalidPathChars()[0] = 'A';
    Path.GetInvalidPathChars()[1] = 'A';
    Path.GetInvalidPathChars()[2] = 'A';

    // The following code displays the elements of the array to the
    // console. Note that the values have changed.
    foreach(char c in Path.GetInvalidPathChars())
    {
        Console.Write(c);
    }
}
}

```

不能通过将 `badChars` 数组设置为 **readonly**（在 **Visual Basic** 中为 **ReadOnly**）纠正上一个示例中出现的问题。您可以克隆 `badChars` 数组并返回副本，但这将对性能产生明显的影响。有关详细信息，请参见下面的“返回数组的属性”小节。下面的代码示例演示如何修改 `GetInvalidPathChars` 方法以返回 `badChars` 数组的副本。

Visual Basic

[复制代码](#)

```

Public Shared Function GetInvalidPathChars() As Char()
    Return CType(badChars.Clone(), Char())
End Function

```

C#

[复制代码](#)

```

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

```

返回数组的属性

若要避免由返回数组的属性所导致的代码低效，则应当使用集合。在下面的代码示例中，每个 `myObj` 属性调用都创建数组的一个副本。因此，在下面的循环中将创建该数组的 $2n+1$ 个副本。

Visual Basic

[复制代码](#)

```

Dim i As Integer
For i = 0 To obj.myObj.Count - 1
    DoSomething(obj.myObj(i))
Next i

```

C#

[复制代码](#)

```

for (int i = 0; i < obj.myObj.Count; i++)
    DoSomething(obj.myObj[i]);

```

有关更多信息，请参见[在属性和方法之间选择](#)。

返回数组的字段

不要使用数组的 **readonly**（在 **Visual Basic** 中为 **ReadOnly**）字段。如果使用了该字段，则数组为只读

数组，无法进行更改，但是数组中的元素可以更改。下面的代码示例演示如何更改只读数组 `InvalidPathChars` 中的元素。

C# [复制代码](#)

```
public sealed class Path
{
    private Path(){}
    public static readonly char[] InvalidPathChars = {'\\', '<', '>', '|'}
}
//The following code can be used to change the values in the array.
Path.InvalidPathChars[0] = 'A';
```

在集合中使用索引属性

索引属性应该只用作集合类或接口的默认成员。不要在非集合类型中创建函数家族。方法模式（如 **Add**、**Item** 和 **Count**）发出类型应该是集合的信号。

返回空数组

[String](#) 和 [Array](#) 属性在任何情况下都不应该返回 **null** 引用。**Null** 在此上下文中会难于理解。例如，用户可能假定下面的代码是有效的。

Visual Basic [复制代码](#)

```
Public Sub DoSomething()
    Dim s As String = SomeOtherFunc()
    If s.Length > 0 Then
        ' Do something else.
    End If
End Sub
```

C# [复制代码](#)

```
public void DoSomething()
{
    string s = SomeOtherFunc();
    if (s.Length > 0)
    {
        // Do something else.
    }
}
```

一般的规则是应该以相同的方式处理 **null**、空字符串 ("") 和空 (0 项) 数组。应返回空数组而不是 **null** 引用。

部分版权所有 2005 Microsoft Corporation。保留所有权利。

部分版权所有 Addison-Wesley Corporation。保留所有权利。

有关设计指南的更多信息，请参见 Krzysztof Cwalina 和 Brad Abrams 编著、Addison-Wesley 于 2005 年出版的“Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries”（《框架设计指南：可重用 .NET 库的约定、术语和模式》）。

请参见

概念

[类库开发的设计准则](#)

[使用准则](#)

参考

[Array](#)



.NET Framework 开发人员指南
属性使用指南
[请参见](#)

.NET Framework 使开发人员能够发明新的声明性信息类型、为各种程序实体指定声明性信息，以及在运行环境中检索属性信息。例如，框架可以定义一个可放置在程序元素（如类和方法）上的 `HelpAttribute` 属性，以提供从程序元素到其文档的映射。新的声明性信息类型通过属性类的声明来定义，这些类可能有定位参数和命名参数。有关属性的更多信息，请参见[编写自定义属性](#)。

下面的规则概括了属性类的使用指南：

- 1 将 `Attribute` 后缀添加到自定义属性类，如下面的示例所示。

Visual Basic [复制代码](#)

```
Public Class ObsoleteAttribute{}
```

C# [复制代码](#)

```
public class ObsoleteAttribute{}
```

- 1 在属性上指定 `AttributeUsage` 以精确定义它们的用法，如下面的示例所示。

Visual Basic [复制代码](#)

```
<AttributeUsage(AttributeTargets.All, Inherited := False, AllowMultiple := True)> _  
  
Public Class ObsoleteAttribute  
    Inherits Attribute  
    ' Insert code here.  
End Class
```

C# [复制代码](#)

```
[AttributeUsage(AttributeTargets.All, Inherited = false, AllowMultiple = true)]  
public class ObsoleteAttribute: Attribute {}
```

- 1 尽可能密封属性类，以便不能从它们派生类。
- 1 为必需的参数使用定位参数（构造函数参数）。提供与每个定位参数同名的只读属性，但更改大小写以将它们区分开。这样就可以在运行时访问参数。
- 1 对可选参数使用命名变量，并为每个命名变量提供读/写属性。
- 1 不要同时用命名参数和定位参数来定义参数。下面的代码示例阐释了这种模式。

Visual Basic [复制代码](#)

```
Public Class NameAttribute  
    Inherits Attribute  
    Private userNameValue as String  
    Private ageValue as Integer  
  
    ' This is a positional argument.  
    Public Sub New(userName As String)  
        userNameValue = userName  
    End Sub  
  
    Public ReadOnly Property UserName() As String  
        Get  
            Return userNameValue  
        End Get  
    End Property  
  
    ' This is a named argument.  
    Public Property Age() As Integer  
        Get  
            Return ageValue  
        End Get  
        Set  
            ageValue = value  
        End Set  
    End Property  
End Class
```

C#

 [复制代码](#)

```
public class NameAttribute: Attribute
{
    string userName;
    int age;

    // This is a positional argument.
    public NameAttribute (string userName)
    {
        this.userName = userName;
    }
    public string UserName
    {
        get
        {
            return userName;
        }
    }
    // This is a named argument.
    public int Age
    {
        get
        {
            return age;
        }
        set
        {
            age = value;
        }
    }
}
```

部分版权所有 2005 Microsoft Corporation。保留所有权利。

部分版权所有 Addison-Wesley Corporation。保留所有权利。

有关设计指南的更多信息，请参见 Krzysztof Cwalina 和 Brad Abrams 编著、Addison-Wesley 于 2005 年出版的“**Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries**”（《框架设计指南：可重用 .NET 库的约定、术语和模式》）。

■ 请参见

概念

[类库开发的设计准则](#)

[使用准则](#)



.NET Framework 开发人员指南

实现 Equals 方法

[请参见](#)

有关实现相等运算符 (==) 的信息, 请参见 [Equals 和相等运算符 \(==\) 的实现准则](#)。

- | 重写 **GetHashCode** 方法, 使类型可以在哈希表中正确地工作。
- | 不要在 **Equals** 方法实现中引发异常。相反, 对于空参数返回 **false**。
- | 遵循在 [Object.Equals 方法](#) 上定义的协定, 如下所示:
 - | **x.Equals(x)** 返回 **true**。
 - | **x.Equals(y)** 与 **y.Equals(x)** 返回相同的值。
 - | 当且仅当 **x.Equals(z)** 返回 **true** 时, (**x.Equals(y) && y.Equals(z)**) 返回 **true**。
 - | 只要不修改 **x** 和 **y** 所引用的对象, **x.Equals(y)** 的连续调用就返回相同的值。
 - | **x.Equals(null)** 返回 **false**。
- | 对于某些类型的对象, 让 **Equals** 测试值相等而不是引用相等会更好。如果两个对象有相同的值, 这些 **Equals** 实现返回 **true**, 即使它们不是相同的实例。构成对象值的定义由类型的实施者决定, 但一般是对象的实例变量中存储的部分或全部数据。例如, 字符串的值基于字符串的字符; 对于字符串的任何两个按相同的顺序包含完全相同的字符的实例, **String** 类的 **Equals** 方法返回 **true**。
- | 当基类的 **Equals** 方法提供值相等性时, 派生类中的 **Equals** 重写应调用继承的 **Equals** 实现。
- | 如果用支持运算符重载的语言进行编程, 并选择重载指定类型的相等运算符 (==), 则该类型应重写 **Equals** 方法。**Equals** 方法的这些实现应与相等运算符返回相同的结果。遵循这些准则有助于确保使用 **Equals** 的类库代码 (如 [ArrayList](#) 和 [Hashtable](#)) 的工作方式与应用程序代码使用相等运算符的方式一致。
- | 如果实现值类型, 应考虑重写 **Equals** 方法, 以获得优于 [ValueType](#) 的默认 **Equals** 方法实现的性能。如果重写 **Equals** 并且语言支持运算符重载, 应重载值类型的相等运算符。
- | 如果实现引用类型, 并且类型看起来像一个基类 (如 [Point](#)、[String](#)、[BigInteger](#) 等), 应考虑重写引用类型的 **Equals** 方法。对于大多数引用类型而言, 即使重写 **Equals**, 也不应重载相等运算符。但是, 如果实现的引用类型将具有值语义 (如复数类型), 应重写相等运算符。
- | 如果在给定类型上实现 [IComparable](#) 接口, 应重写该类型的 **Equals**。

示例

下面的代码示例演示如何实现、重写调用及重载 **Equals** 方法。

实现 Equals 方法

下面的代码示例包含两个对 **Equals** 方法的默认实现的调用。

Visual Basic

[复制代码](#)

```
Imports System
Class SampleClass
    Public Shared Sub Main()
        Dim obj1 As New System.Object()
        Dim obj2 As New System.Object()
        Console.WriteLine(obj1.Equals(obj2))
        obj1 = obj2
        Console.WriteLine(obj1.Equals(obj2))
    End Sub
End Class
```

C#

[复制代码](#)

```
using System;
class SampleClass
{
    public static void Main()
    {
        Object obj1 = new Object();
        Object obj2 = new Object();
        Console.WriteLine(obj1.Equals(obj2));
        obj1 = obj2;
    }
}
```

```

        Console.WriteLine(obj1.Equals(obj2));
    }
}

```

上述代码的输出结果如下:

 [复制代码](#)

```

False
True

```

重写 Equals 方法

下面的代码示例展示一个 `Point` 类, 该类重写 `Equals` 方法以提供值相等性; 还展示了一个从 `Point` 派生的 `Point3D` 类。由于 `Point` 类对 `Equals` 的重写是继承链中引入值相等性的第一个重写方法, 因此不调用基类的 `Equals` 方法 (它从 `Object` 继承并检查引用相等性)。但是, 由于 `Point` 以提供值相等性的方式实现 `Equals`, 因此 `Point3D.Equals` 调用 `Point.Equals`。

Visual Basic

 [复制代码](#)

```

Namespace Examples.DesignGuidelines.EqualsImplementation

Public Class Point
    Protected x As Integer
    Protected y As Integer

    Public Sub New (xValue As Integer, yValue As Integer)
        Me.x = xValue
        Me.y = yValue
    End Sub

    Public Overrides Overloads Function Equals(obj As Object) As Boolean

        If obj Is Nothing OrElse Not Me.GetType() Is obj.GetType() Then
            Return False
        End If

        Dim p As Point = CType(obj, Point)
        Return Me.x = p.x And Me.y = p.y
    End Function

    Public Overrides Function GetHashCode() As Integer
        Return x Xor y
    End Function
End Class

Public Class Point3D
    Inherits Point
    Private z As Integer

    Public Sub New (xValue As Integer, yValue As Integer, zValue As Integer)
        MyBase.New(xValue, yValue)
        Me.z = zValue
    End Sub

    Public Overrides Overloads Function Equals(obj As Object) As Boolean
        Return MyBase.Equals(obj) And z = CType(obj, Point3D).z
    End Function

    Public Overrides Function GetHashCode() As Integer
        Return MyBase.GetHashCode() Xor z
    End Function
End Class

End Namespace

```

C#

 [复制代码](#)

```

using System;

namespace Examples.DesignGuidelines.EqualsImplementation

```

```

{
class Point: object
{
    protected int x, y;

    public Point(int xValue, int yValue)
    {
        x = xValue;
        y = yValue;
    }
    public override bool Equals(Object obj)
    {
        // Check for null values and compare run-time types.
        if (obj == null || GetType() != obj.GetType())
            return false;

        Point p = (Point)obj;
        return (x == p.x) && (y == p.y);
    }
    public override int GetHashCode()
    {
        return x ^ y;
    }
}

class Point3D: Point
{
    int z;

    public Point3D(int xValue, int yValue, int zValue) : base(xValue, yValue)
    {
        z = zValue;
    }
    public override bool Equals(Object obj)
    {
        return base.Equals(obj) && z == ((Point3D)obj).z;
    }
    public override int GetHashCode()
    {
        return base.GetHashCode() ^ z;
    }
}
}

```

`Point.Equals` 方法检查 `obj` 参数是否非空以及它是否与此对象引用了相同类型的实例。如果任何一个检查失败，该方法都返回 `false`。`Equals` 方法使用 [GetType](#) 方法确定两个对象的运行时类型是否相同。注意，这里不使用 `typeof`（在 Visual Basic 中为 `TypeOf`），因为它返回静态类型。如果该方法改为使用 `obj is Point` 形式的检查，则当 `obj` 是从 `Point` 派生的类的实例时，检查结果将返回 `true`，即使 `obj` 和当前实例不属于相同的运行时类型亦如此。验证了两个对象的类型相同后，该方法将 `obj` 强制转换为类型 `Point`，并返回两个对象实例变量的比较结果。

在 `Point3D.Equals` 中，在完成任何其他操作之前调用继承的 `Equals` 方法。继承的 `Equals` 方法执行下列检查：`obj` 是否不为 `null`，`obj` 是否是与此对象相同的类的实例，以及继承的实例变量是否匹配。仅当继承的 `Equals` 返回 `true` 时，方法才比较派生类中引入的实例变量。具体说来，除非 `obj` 已被确定属于 `Point3D` 类型或者是从 `Point3D` 派生的类，否则不执行到 `Point3D` 的强制转换。

使用 Equals 方法比较实例变量

在前面的示例中，相等运算符 (`==`) 用于比较各实例变量。在某些情况下，在 `Equals` 实现中用 `Equals` 方法比较实例变量是适当的，如下面的代码示例所示。

Visual Basic



```

Imports System

Class Rectangle
    Private a, b As Point

    Public Overrides Overloads Function Equals(obj As [Object]) As Boolean
        If obj Is Nothing Or Not Me.GetType() Is obj.GetType() Then

```

```

        Return False
    End If
    Dim r As Rectangle = CType(obj, Rectangle)
    ' Use Equals to compare instance variables.
    Return Me.a.Equals(r.a) And Me.b.Equals(r.b)
End Function

Public Overrides Function GetHashCode() As Integer
    Return a.GetHashCode() ^ b.GetHashCode()
End Function
End Class

```

C# [复制代码](#)

```

using System;
class Rectangle
{
    Point a, b;
    public override bool Equals(Object obj)
    {
        if (obj == null || GetType() != obj.GetType()) return false;
        Rectangle r = (Rectangle)obj;
        // Use Equals to compare instance variables.
        return a.Equals(r.a) && b.Equals(r.b);
    }
    public override int GetHashCode()
    {
        return a.GetHashCode() ^ b.GetHashCode();
    }
}

```

重写相等运算符 (==) 和 Equals 方法

有些编程语言（如 **C#**）支持运算符重载。当类型重载相等运算符 (==) 时，它还应该重写 **Equals** 方法以提供相同的功能。一般是通过根据重载的相等运算符 (==) 编写 **Equals** 方法来达到这一目的，如下面的代码示例所示。

C# [复制代码](#)

```

public struct Complex
{
    double re, im;
    public override bool Equals(Object obj)
    {
        return obj is Complex && this == (Complex)obj;
    }
    public override int GetHashCode()
    {
        return re.GetHashCode() ^ im.GetHashCode();
    }
    public static bool operator ==(Complex x, Complex y)
    {
        return x.re == y.re && x.im == y.im;
    }
    public static bool operator !=(Complex x, Complex y)
    {
        return !(x == y);
    }
}

```

由于 **Complex** 是 **C# struct**（一种值类型），我们知道没有类从 **Complex** 派生。因此，**Equals** 方法不需要比较每个对象的 **GetType** 结果。相反，它使用 **is** 运算符检查 **obj** 参数的类型。

部分版权所有 2005 Microsoft Corporation。保留所有权利。

部分版权所有 Addison-Wesley Corporation。保留所有权利。

有关设计指南的更多信息，请参见 Krzysztof Cwalina 和 Brad Abrams 编著、Addison-Wesley 于 2005 年出版的“Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries”（《框架设计指南：可重用 .NET 库的约定、术语和模式》）。

■ 请参见

概念

类库开发的设计准则



.NET Framework 开发人员指南

Equals 和相等运算符 (==) 的实现准则

[请参见](#)

下面的规则概括了 **Equals** 方法和等号运算符 (==) 的实现准则:

- 1 每次实现 **Equals** 方法时都实现 **GetHashCode** 方法。这可以使 **Equals** 和 **GetHashCode** 保持同步。
- 1 每次实现相等运算符 (==) 时, 都重写 **Equals** 方法, 使它们执行同样的操作。这样, 使用 **Equals** 方法的基础结构代码 (如 [Hashtable](#) 和 [ArrayList](#)) 的行为就与用相等运算符编写的用户代码相同。
- 1 每次实现 [IComparable](#) 时都要重写 **Equals** 方法。
- 1 实现 [IComparable](#) 时, 应考虑实现相等 (==)、不相等 (!=)、小于 (<) 和大于 (>) 运算符的运算符重载。
- 1 不要在 **Equals**、**GetHashCode** 方法或相等运算符 (==) 中引发异常。

有关 **Equals** 方法的相关信息, 请参见[实现 Equals 方法](#)。

■ 在值类型中实现相等运算符 (==)

大多数编程语言中都没有用于值类型的默认相等运算符 (==) 实现。因此, 只要相等有意义就应该重载相等运算符 (==)。

应考虑在值类型中实现 **Equals** 方法, 这是因为 [System.ValueType](#) 的默认实现和自定义实现都不会执行。

每次重写 **Equals** 方法时都实现相等运算符 (==)。

■ 在引用类型中实现相等运算符 (==)

大多数语言确实为引用类型提供默认的相等运算符 (==) 实现。因此, 在引用类型中实现相等运算符 (==) 时应小心。大多数引用类型 (即使是实现 **Equals** 方法的引用类型) 都不应重写相等运算符 (==)。

如果类型是 [Point](#)、[String](#)、[BigNumber](#) 等基类型, 则应重写相等运算符 (==)。每当考虑重载加法 (+) 和减法 (-) 运算符时, 也应该考虑重载相等运算符 (==)。

部分版权所有 2005 Microsoft Corporation。保留所有权利。

部分版权所有 Addison-Wesley Corporation。保留所有权利。

有关设计指南的更多信息, 请参见 Krzysztof Cwalina 和 Brad Abrams 编著、Addison-Wesley 于 2005 年出版的 “[Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries](#)” (《框架设计指南: 可重用 .NET 库的约定、术语和模式》)。

■ 请参见

概念

[类库开发的设计准则](#)

[实现 Equals 方法](#)

[使用准则](#)

参考

[Object.Equals](#)



.NET Framework 开发人员指南

设计模式

[请参见](#)

本主题提供在类库中实现通用设计模式的准则。

部分版权所有 2005 Microsoft Corporation。保留所有权利。

部分版权所有 Addison-Wesley Corporation。保留所有权利。

有关设计指南的更多信息, 请参见 Krzysztof Cwalina 和 Brad Abrams 编著、Addison-Wesley 于 2005 年出版的“Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries”(《框架设计指南: 可重用 .NET 库的约定、术语和模式》)。

■ 本节内容

[实现 Finalize 和 Dispose 以清理非托管资源](#)

描述推荐设计模式, 该模式要在类库中实现以使用 [Finalize](#) 和 [Dispose](#) 方法清理非托管资源。

[超时的用法](#)

描述在基类库中使用超时以指定调用方最多愿意花多长时间等待方法调用完成的准则。

■ 相关章节

[异步编程设计模式](#)

描述异步编程设计准则。

■ 请参见

概念

[类库开发的设计准则](#)



.NET Framework 开发人员指南

实现 Finalize 和 Dispose 以清理非托管资源

[请参见](#)

注意:

有关使用 C++ 的终止和释放的信息, 请参见 [Destructors and Finalizers in Visual C++](#)。

类实例经常封装对不受运行库管理的资源（如窗口句柄 (HWND)、数据库连接等）的控制。因此, 应该既提供显式方法也提供隐式方法来释放这些资源。通过在对象上实现受保护的 [Finalize](#)（在 C# 和 C++ 中为析构函数语法）可提供隐式控制。当不再有任何有效的对象引用后, 垃圾回收器在某个时间调用此方法。

在有些情况下, 您可能想为使用该对象的程序员提供显式释放这些外部资源的能力, 以便在垃圾回收器释放该对象之前释放这些资源。当外部资源稀少或者昂贵时, 如果程序员在资源不再使用时显式释放它们, 则可以获得更好的性能。若要提供显式控制, 需实现 [IDisposable](#) 提供的 [Dispose](#)。在完成使用该对象的操作时, 该对象的使用者应调用此方法。即使对对象的其他引用是活动的, 也可以调用 [Dispose](#)。

注意, 即使在通过 [Dispose](#) 提供显式控制时, 也应该使用 [Finalize](#) 方法提供隐式清理。[Finalize](#) 提供了候补手段, 可防止在程序员未能调用 [Dispose](#) 时造成资源永久泄漏。

有关如何实现 [Finalize](#) 和 [Dispose](#) 以清理非托管资源的更多信息, 请参见[垃圾回收](#)。下面的代码示例演示实现 [Dispose](#) 的基本设计方案。此示例需要 [System](#) 命名空间。

Visual Basic

[复制代码](#)

```
' Design pattern for a base class.

Public Class Base
    Implements IDisposable
    ' Field to handle multiple calls to Dispose gracefully.
    Dim disposed as Boolean = false

    ' Implement IDisposable.
    Public Overloads Sub Dispose() Implements IDisposable.Dispose
        Dispose(True)
        GC.SuppressFinalize(Me)
    End Sub

    Protected Overloads Overridable Sub Dispose(disposing As Boolean)
        If disposed = False Then
            If disposing Then
                ' Free other state (managed objects).
                disposed = True
            End If
        End If
        ' Free your own state (unmanaged objects).
        ' Set large fields to null.
    End Sub

    Protected Overrides Sub Finalize()
        ' Simply call Dispose(False).
        Dispose(False)
    End Sub
End Class

' Design pattern for a derived class.
Public Class Derived
    Inherits Base

    ' Field to handle multiple calls to Dispose gracefully.
    Dim disposed as Boolean = false

    Protected Overloads Overrides Sub Dispose(disposing As Boolean)
        If disposed = False Then
            If disposing Then
                ' Release managed resources.
                disposed = True
            End If
        End If
    End If
End Class
```



```

' Release unmanaged resources.
' Set large fields to null.
' Call Dispose on your base class.
Mybase.Dispose(disposing)
End Sub
' The derived class does not have a Finalize method
' or a Dispose method without parameters because it inherits
' them from the base class.
End Class

```

C# [复制代码](#)

```

// Design pattern for a base class.
public class Base: IDisposable
{
    //Implement IDisposable.
    public void Dispose()
    {
        Dispose(true);
        GC.SuppressFinalize(this);
    }

    protected virtual void Dispose(bool disposing)
    {
        if (disposing)
        {
            // Free other state (managed objects).
        }
        // Free your own state (unmanaged objects).
        // Set large fields to null.
    }

    // Use C# destructor syntax for finalization code.
    ~Base()
    {
        // Simply call Dispose(false).
        Dispose (false);
    }
}
// Design pattern for a derived class.
public class Derived: Base
{
    protected override void Dispose(bool disposing)
    {
        if (disposing)
        {
            // Release managed resources.
        }
        // Release unmanaged resources.
        // Set large fields to null.
        // Call Dispose on your base class.
        base.Dispose(disposing);
    }
    // The derived class does not have a Finalize method
    // or a Dispose method without parameters because it inherits
    // them from the base class.
}

```

有关阐释 **Finalize** 和 **Dispose** 的实现设计方案的更详细的代码示例, 请参见[实现 Dispose 方法](#)。

自定义释放方法名称

有时特定于域的名称比 **Dispose** 更合适。例如, 文件封装可能需要使用 **Close** 方法名称。在此情况下, 可以通过专用方式实现 **Dispose** 并创建调用 **Dispose** 的公共 **Close** 方法。下面的代码示例阐释了这种模式。可以用适合您的域的方法名称替换 **Close**。此示例需要 [System](#) 命名空间。

Visual Basic [复制代码](#)

```

' Do not make this method overridable.
' A derived class should not be allowed

```

```
' to override this method.
Public Sub Close()
    ' Call the Dispose method with no parameters.
    Dispose()
End Sub
```

C#

 复制代码

```
// Do not make this method virtual.
// A derived class should not be allowed
// to override this method.
public void Close()
{
    // Call the Dispose method with no parameters.
    Dispose();
}
```

Finalize

下面的规则概括了 **Finalize** 方法的使用准则：

- 1 仅在要求终结的对象上实现 **Finalize**。存在与 **Finalize** 方法相关的性能开销。
- 1 如果需要 **Finalize** 方法，应考虑实现 **IDisposable**，以使类的用户可以避免因调用 **Finalize** 方法而带来的开销。
- 1 不要提高 **Finalize** 方法的可见性。该方法的可见性应该是 **protected**，而不是 **public**。
- 1 对象的 **Finalize** 方法应该释放该对象拥有的所有外部资源。此外，**Finalize** 方法应该仅释放由该对象控制的资源。**Finalize** 方法不应该引用任何其他对象。
- 1 不要对不是对象的基类的对象直接调用 **Finalize** 方法。在 C# 编程语言中，这不是有效的操作。
- 1 应在对象的 **Finalize** 方法中调用基类的 **Finalize** 方法。

注意：

基类的 **Finalize** 方法通过 C# 和 C++ 析构函数语法自动进行调用。

释放

下面的规则概括了 **Dispose** 方法的使用准则：

- 1 在封装明确需要释放的资源类型上实现释放设计方案。用户可以通过调用公共 **Dispose** 方法释放外部资源。
- 1 在通常包含控制资源的派生类型的基类型上实现释放设计方案，即使基类型并不需要也如此。如果基类型有 **Close** 方法，这通常指示需要实现 **Dispose**。在这类情况下，不要在基类型上实现 **Finalize** 方法。应该在任何引入需要清理的资源的派生类型中实现 **Finalize**。
- 1 使用类型的 **Dispose** 方法释放该类型所拥有的所有可释放资源。
- 1 对实例调用了 **Dispose** 后，应通过调用 [GC.SuppressFinalize](#) 禁止 **Finalize** 方法运行。此规则的一个例外是当必须用 **Finalize** 完成 **Dispose** 没有完成的工作的情况，但这种情况很少见。
- 1 如果基类实现了 **IDisposable**，则应调用基类的 **Dispose** 方法。
- 1 不要假定 **Dispose** 将被调用。如果 **Dispose** 未被调用，也应该使用 **Finalize** 方法释放类型所拥有的非托管资源。
- 1 当资源已经释放时，在该类型上从实例方法（非 **Dispose**）引发一个 **ObjectDisposedException**。该规则不适用于 **Dispose** 方法，该方法应该可以在不引发异常的情况下被多次调用。
- 1 通过基类型的层次结构传播对 **Dispose** 的调用。**Dispose** 方法应释放由此对象以及此对象所拥有的任何对象所控制的所有资源。例如，可以创建一个类似 **TextReader** 的对象来控制 **Stream** 和 **Encoding**，两者均在用户不知道的情况下由 **TextReader** 创建。另外，**Stream** 和 **Encoding** 都可以获取外部资源。当对 **TextReader** 调用 **Dispose** 方法时，**TextReader** 应继而对 **Stream** 和 **Encoding** 调用 **Dispose**，使它们释放其外部资源。
- 1 考虑在调用了某对象的 **Dispose** 方法后禁止对该对象的使用。重新创建已释放的对象是难以实现的方案。
- 1 允许 **Dispose** 方法被调用多次而不引发异常。此方法在首次调用后应该什么也不做。

部分版权所有 2005 Microsoft Corporation。保留所有权利。

部分版权所有 Addison-Wesley Corporation。保留所有权利。

有关设计指南的更多信息, 请参见 Krzysztof Cwalina 和 Brad Abrams 编著、Addison-Wesley 于 2005 年出版的“Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries”(《框架设计指南: 可重用 .NET 库的约定、术语和模式》)。

■ 请参见

概念

[类库开发的设计准则](#)

[垃圾回收](#)

[设计模式](#)

参考

[IDisposable.Dispose](#)

[Object.Finalize](#)



.NET Framework 开发人员指南

超时的使用

[请参见](#)

使用超时指定调用方最多愿意花多长时间等待方法调用完成。

超时可以是方法调用参数的形式，如下所示。

Visual Basic

[复制代码](#)

```
server.PerformOperation(timeout)
```

C#

[复制代码](#)

```
server.PerformOperation(timeout);
```

超时也可以用作服务器类的属性，如下所示。

Visual Basic

[复制代码](#)

```
server.Timeout = timeout  
server.PerformOperation()
```

C#

[复制代码](#)

```
server.Timeout = timeout;  
server.PerformOperation();
```

应该首选第一种方法，因为操作与超时之间的关联更清晰。如果服务器类被设计成与可视化设计器一起使用的组件，则基于属性的方法可能更好。

超时一直以来都是用整数表示的。整数超时可能难以使用，这是因为超时的单位是什么不明显，并且很难将时间单位翻译为常用的毫秒。

更好的方法是将 [TimeSpan](#) 结构用作超时类型。[TimeSpan](#) 解决了上述整数超时的问题。下面的代码示例演示如何使用 [TimeSpan](#) 类型的超时。

Visual Basic

[复制代码](#)

```
Public Class Server  
    Public Sub PerformOperation(timeout As TimeSpan)  
        ' Insert code for the method here.  
        Console.WriteLine("performing operation with timeout {0}", _  
            timeout.ToString())  
    End Sub  
End Class
```

C#

[复制代码](#)

```
public class Server  
{  
    public void PerformOperation(TimeSpan timeout)  
    {  
        // Insert code for the method here.  
        Console.WriteLine("performing operation with timeout {0}",  
            timeout.ToString());  
    }  
}
```

如果超时被设置为 `TimeSpan(0)`，当没有立即完成操作时，方法应引发异常。如果超时为 `TimeSpan.MaxValue`，则操作应永远等待而不超时，就像没有设置超时一样。支持这两个值不需要服务器类，但是如果指定了不受支持的超时值，服务器类应引发 [ArgumentException](#)。

如果超时过期并且引发了异常，服务器类应取消基础操作。

如果使用了默认超时，服务器类应包含一个静态属性，用以指定当用户没有指定超时时使用的超时。下面的代码示例演示如何实现指定默认超时的属性。

Visual Basic

[复制代码](#)

```

Class ServerWithDefault
    Private Shared defaultTimeout As New TimeSpan(1000)

    Public Overloads Sub PerformOperation()
        Me.PerformOperation(DefaultOperationTimeout)
    End Sub

    Public Overloads Sub PerformOperation(timeout As TimeSpan)
        ' Insert code here.
        Console.WriteLine("performing operation with timeout {0}", _
            timeout.ToString())
    End Sub

    Public Shared ReadOnly Property DefaultOperationTimeout As TimeSpan
        Get
            Return defaultTimeout
        End Get
    End Property
End Class

```

C#

 [复制代码](#)

```

class ServerWithDefault
{
    static TimeSpan defaultTimeout = new TimeSpan(1000);

    public void PerformOperation()
    {
        this.PerformOperation(DefaultOperationTimeout);
    }

    public void PerformOperation(TimeSpan timeout)
    {
        // Insert code here.
        Console.WriteLine("performing operation with timeout {0}",
            timeout.ToString());
    }

    public static TimeSpan DefaultOperationTimeout
    {
        get
        {
            return defaultTimeout;
        }
    }
}

```

不能将超时解析为 **TimeSpan** 解析的类型应将超时舍入到可以提供的最接近的间隔。例如，只能按一秒的增量等待的类型应该舍入到最接近的秒。此规则的例外情况是当值向下舍入到零时。这种情况下，超时应向上舍入到可能的最小超时。避免舍入为零可防止出现“繁忙等待”循环，即零超时值导致 100% 的处理器使用。

另外，建议在超时过期时引发异常而不是返回错误代码。超时过期意味着操作未能成功完成，因此应该按任何其他运行时错误处理。有关更多信息，请参见[异常设计准则](#)。

对于具有超时的异步操作，在首次访问操作结果时，应该调用回调函数并引发异常。下面的代码示例中阐释了这一点。

Visual Basic

```

Sub OnReceiveCompleted(ByVal sender As System.Object, ByVal asyncResult As ReceiveCompletedEventArgs)
    Dim queue As MessageQueue = CType(sender, MessageQueue)
    ' The following code will throw an exception
    ' if BeginReceive has timed out.
    Dim message As Message = queue.EndReceive(asyncResult.AsyncResult)
    Console.WriteLine("Message: " + CStr(message.Body))
    queue.BeginReceive(New TimeSpan(1, 0, 0))
End Sub

```

C#

 [复制代码](#)

```
void OnReceiveCompleted(Object sender, ReceiveCompletedEventArgs asyncResult)
{
    MessageQueue queue = (MessageQueue) sender;
    // The following code will throw an exception
    // if BeginReceive has timed out.
    Message message = queue.EndReceive(asyncResult.AsyncResult);
    Console.WriteLine("Message: " + (string)message.Body);
    queue.BeginReceive(new TimeSpan(1,0,0));
}
```

部分版权所有 2005 Microsoft Corporation。保留所有权利。

部分版权所有 Addison-Wesley Corporation。保留所有权利。

有关设计指南的更多信息, 请参见 Krzysztof Cwalina 和 Brad Abrams 编著、Addison-Wesley 于 2005 年出版的“Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries”(《框架设计指南: 可重用 .NET 库的约定、术语和模式》)。

■ 请参见

概念

[类库开发的设计准则](#)

[使用准则](#)

[异步编程设计模式](#)
