编码规范补充

.NET 代码规范旨在提高代码质量的规则。 这些规则分为设计、全球化、性能和安全性等领域。 某些规则特定于 .NET API 用法，而其他规则与通用代码质量相关。

# 设计规则

设计规则支持遵从 .NET Framework 设计准则。

## 不要在泛型类型中声明静态成员

调用泛型类型的 static 成员时，必须指定该类型的类型参数。 当调用不支持推理的泛型实例成员时，必须指定该成员的类型参数。 在上述两种情况下，用于指定类型参数的语法不同且容易混淆。 通常，应避免前两个声明，以便在调用成员时不必指定类型参数。 这导致用于调用泛型中的成员的语法与用于非泛型的语法没有区别。如下情况：

class Example<T>

{

public static void StaticMethod()

{

}

public void GenericMethod<T>()

{

}

}

var example = new Example<int>();

Example<int>.StaticMethod();

example.GenericMethod<int>();

## 抽象类型不应具有公共构造函数

抽象类型的构造函数只能由派生类型调用。 由于公共构造函数可创建类型的实例，但无法创建抽象类型的实例，因此具有公共构造函数的抽象类型在设计上是错误的。

## 枚举应具有零值

像其他值类型一样，未初始化枚举的默认值为零。 无标志特性的枚举应定义值为零的成员，这样默认值即为该枚举的有效值。 如果可行，请将成员命名为“None”。 否则，将零赋给最常使用的成员。 默认情况下，如果未在声明中设置第一个枚举成员的值，则其值为零。如果应用了 FlagsAttribute 的枚举定义值为零成员，则该成员的名称应为“None”，以指示枚举中尚未设置值。 将值为零的成员用于任何其他目的与使用 FlagsAttribute 存在冲突，因为 AND 和 OR 位运算符对成员没有意义。 这意味着，只应为一个成员分配零值。 如果有多个零值成员在标志特性的枚举中出现，对于不为零的成员，Enum.ToString() 将返回不正确的结果。

## 集合应实现泛型接口

若要扩大集合的用途，应实现某个泛型集合接口。 然后，可以使用该集合来填充泛型集合类型，如下所示：

* System.Collections.Generic.List<T>
* System.Collections.Generic.Queue<T>
* System.Collections.Generic.Stack<T>

请实现某个泛型集合接口：

* System.Collections.Generic.IEnumerable<T>
* System.Collections.Generic.ICollection<T>
* System.Collections.Generic.IList<T>

## 在适用处使用属性

在大多数情况下，属性表示数据，方法执行操作。 访问属性的方式类似于访问字段，这使得它们更易于使用。 如果一个方法具备以下条件之一，则该方法可能很适合成为属性：

* 方法不采用任何自变量，并返回对象的状态信息。
* 方法接受单个自变量，以设置对象的部分状态。
* 一个方法的名称以 Get 开头，不采用任何参数，并返回一个非数组的值。（性能规则第7条）

## 在适用处使用事件

如果在对象状态发生变化时调用该方法，请考虑更改设计以使用 .NET 事件模型。事件遵循“观察者”或“发布-订阅”设计模式；当必须将一个对象的状态更改传达给其他对象时，它们适用。 如果为响应明确定义的状态更改而调用一个方法，则应由事件处理程序调用该方法。 调用该方法的对象应引发事件而不是直接调用该方法。用户界面应用程序中发现了一些常见事件示例，其中用户操作（如单击按钮）会导致执行一段代码。 .NET 事件模型并不局限于用户界面。 它应在必须将状态更改传达给一个或多个对象的任何位置使用。

## 不要捕捉一般异常类型

捕获一般异常类型可隐藏库用户的运行时问题，并且可能会使调试变得更加困难。如System.Exception 或 System.SystemException等，请捕捉更具体的异常，或者在执行 catch 块中的最后一条语句时重新引发一般异常。

## 属性不应是只写的

Get 访问器提供对属性的读取访问权限，而 set 访问器提供写入访问权限。 虽然可以接受且经常需要使用只读属性，但设计准则禁止使用只写属性。 这是因为允许用户设置值但又禁止该用户查看这个值不能提供任何安全性。 而且，如果没有读访问，将无法查看共享对象的状态，使其用处受到限制。

## 静态容器类型应是 Static 或NotInheritable(sealed)

假定仅包含不设计为继承的静态成员的类型，因为该类型不提供任何可在派生类型中重写的功能。 未计划继承的类型应该用 C# 中的 static 修饰符进行标记，以便禁止其作为基类型使用。 此外，应删除其默认构造函数。不包括抽象类。

## 将 P/Invoke 移动到 NativeMethods 类

平台调用方法应该放在合适的 NativeMethods 类中。 对于大多数应用程序，将 P/Invoke 移动到名为 NativeMethods 的新类便足够了。但是，如果要开发在其他应用程序中使用的库，应考虑定义两个名为 SafeNativeMethods 和 UnsafeNativeMethods 的其他类。 这些类与 NativeMethods 类相似；但是，它们使用名为 SuppressUnmanagedCodeSecurityAttribute 的特殊属性进行标记 。 应用此属性时，运行时不会执行完整堆栈审核来确保所有调用方都具有 UnmanagedCode 权限。 运行时通常会在启动时检查是否具有此权限。 因此可极大地提高对这些非托管方法的调用的性能，还使具备有限权限的代码可以调用这些方法。不过，应非常小心地使用此属性。 如果未正确实现，则可能会产生严重的安全隐患。

## 正确实现 IDisposable

实现IDisposable接口，请使用Dispose 模式。

public class Example : IDisposable

{

private bool isDisposed;

private IntPtr nativeResource = Marshal.AllocHGlobal(100);

private Managed managed = new Managed();

public void Dispose()

{

//Dispose调用Dispose(true)

Dispose(true);

GC.SuppressFinalize(this);

}

protected virtual void Dispose(bool disposing)

{

if (isDisposed) return;

if (disposing)

{

// 释放托管资源

managed.Dispose();

}

// 释放非托管资源

if (nativeResource != IntPtr.Zero)

{

Marshal.FreeHGlobal(nativeResource);

nativeResource = IntPtr.Zero;

}

isDisposed = true;

}

~Example()

{

// 析构调用 Dispose(false)

Dispose(false);

}

}

## 不要在意外的位置引发异常

不应引发异常的方法可分成以下几类：

1. 属性 Get 方法

属性基本上都是智能字段。 因此，其行为应尽可能类似于字段。 字段不会引发异常，属性也不应引发异常。 如果有一个引发异常的属性，可考虑将其设为方法。属性 Get 方法可引发以下异常：

* System.InvalidOperationException 和所有派生项（包括 System.ObjectDisposedException）
* System.NotSupportedException 和所有派生项
* System.ArgumentException（仅从带有索引的 Get）
* KeyNotFoundException（仅从带有索引的 Get）

1. 事件访问器方法

事件访问器应是不会引发异常的简单操作。 尝试添加或删除事件处理程序时，事件不应引发异常。事件访问器可引发以下异常：

* System.InvalidOperationException 和所有派生项（包括 System.ObjectDisposedException）
* System.NotSupportedException 和所有派生项
* ArgumentException 和派生项

1. Equals 方法

Equals 方法应返回 true 或 false 而不是引发异常。 例如，如果 Equals 传递两个不匹配的类型，则应只返回 false 而不是引发 ArgumentException。

1. GetHashCode 方法

GetHashCode 应始终返回值。 否则，可能会丢失哈希表中的项。采用参数的 GetHashCode 版本可能会引发 ArgumentException。 但是，Object.GetHashCode 应始终不会引发异常。

1. ToString 方法

调试器使用 System.Object.ToString 来帮助以字符串格式显示有关对象的信息。 因此，ToString 不应更改对象的状态，也不应引发异常。

1. 静态构造函数

从静态构造函数引发异常将导致该类型在当前应用程序域中不可用。 从静态构造函数引发异常应具备充分的理由（如安全问题）。

1. 终结器（析构函数）

从终结器引发异常将导致 CLR 快速失败，从而中断过程。 因此，应始终避免在终结器中引发异常。

1. Dispose 方法

System.IDisposable.Dispose 方法不应引发异常。 Dispose 通常作为 finally 子句中清理逻辑的一部分调用。 因此，从 Dispose 显式引发异常将强制用户在 finally 子句内添加异常处理。Dispose (false) 代码路径应始终不会引发异常，因为 Dispose 几乎都是从终结器调用的。

1. 相等运算符

与 Equals 方法一样，相等运算符应返回 true 或 false，而不应引发异常。

1. 隐式强制转换运算符

由于用户通常不知道已调用了隐式强制转换运算符，因此对它引发的异常会感到意外。 因此，隐式强制转换运算符不应引发异常。

## 重写 Equals 时实现 IEquatable

值类型重写 Equals 方法指示它可支持对类型的两个实例进行比较以确定二者的值是否相等。 请考虑实现 IEquatable<T> 接口以支持强类型相等性测试。 这可确保执行相等性检查的调用方调用强类型 System.IEquatable<T>.Equals 方法，避免对参数进行装箱，从而提高性能。

## CancellationToken 参数必须最后出现

执行长时间运行操作或异步操作并可取消的方法，通常采用取消令牌参数。 每个取消令牌都有一个 CancellationTokenSource，以创建令牌并将其用于可取消的计算。 通常的做法是使用一长的方法调用链，将取消令牌从调用方传递到被调用方。 因此，参与可取消计算的大量方法最终都具有取消令牌参数。 但是，取消令牌本身通常与大多数这些方法的核心功能无关。

## 不要将事件字段声明为“虚拟”

遵循这些 .NET 设计指南，在派生类中引发基类事件。 不要在基类中声明虚拟事件。 派生类中的重写事件具有未定义的行为。 C# 编译器不会正确处理此事件，并且无法预知派生事件的订阅者是否实际上会订阅基类事件。

# 文档规则

文档规则支持通过正确为外部可见的 API 使用 XML 文档注释来编写记录详尽的库。

# 全球化规则

全球化规则支持世界通用库和应用程序。

# 性能规则

性能规则支持高性能库和应用程序。

## 在合适的位置使用文本

当调用声明类型的静态构造函数时，将在运行时计算 static readonly 字段的值。 如果 static readonly 字段在声明时被初始化并且静态构造函数不是显式声明的，编译器将发出一个静态构造函数来初始化该字段。const 字段的值是在编译时计算的，并存储在元数据中，这与 static readonly 字段相比，运行时性能提高了。因为赋给目标字段的值可在编译时计算，所以，请将声明更改为 const 字段，以便在编译时（而非运行时）计算该值。/// <summary>

/// 违反规则的代码示例

/// </summary>

public class Example

{

static readonly int x = 3;

static readonly double y = x + 2.1;

static readonly string s = "readonly";

}

/// <summary>

/// 符合规则的代码示例

/// </summary>

public class Example

{

const int x = 3;

const double y = x + 2.1;

const string s = "const";

}

## 避免不必要的初始化

在运行构造函数之前，.NET 运行时将引用类型的所有字段初始化为其默认值。 在大多数情况下，在构造函数中将字段显式初始化为其默认值是多余的，这会增加维护成本，并可能会降低性能（例如，随着程序集大小的增加），可以删除显式初始化。在大多数情况下，正确的解决方法是删除不必要的初始化。

class Example

{

/// <summary>

/// 违反规则的代码示例

/// </summary>

int \_value1 = 0;

/// <summary>

/// 符合规则的代码示例

/// </summary>

int \_value1;

}

在某些情况下，由于字段会永久保留其默认值，因此删除初始化可能会导致发出后续的 [CS0649](https://docs.microsoft.com/zh-cn/dotnet/csharp/misc/cs0649) 警告。 在这种情况下，更好的解决方法是完全删除该字段，或将其替换为属性

class Example

{

/// <summary>

/// 违反规则的代码示例

/// </summary>

private static readonly int \_value = 0;

/// <summary>

/// 符合规则的代码示例

/// </summary>

private static int Value => 0;

}

## 不要忽略方法结果

不必要的对象创建和未使用对象的关联垃圾回收会降低性能。字符串是不可变的，并且 String.ToUpper 等方法返回字符串的新实例，而不是在调用方法中修改字符串的实例。忽略 HRESULT 或错误代码可能导致在错误情况下或资源不足的情况下发生异常行为。已知 LINQ 方法不具有副作用，因此不应忽略其结果。

/// <summary>

/// 违反规则的代码示例

/// </summary>

public class Example

{

private readonly string \_title;

public Example(string title)

{

if (title is not null)

{

title.Trim();

}

\_title = title;

}

public string Title => \_title;

}

/// <summary>

/// 违反规则的代码示例

/// </summary>

public class Example

{

public static Example CreateExample()

{

new Example();

return new Example();

}

}

/// <summary>

/// 符合规则的代码示例

/// </summary>

public class Example

{

private readonly string \_title;

public Example(string title)

{

if (title is not null)

{

title = title.Trim();

}

\_title = title;

}

public string Title => \_title;

}

/// <summary>

/// 符合规则的代码示例

/// </summary>

public class Example

{

public static Example CreateExample()

{

return new Example();

}

}

## 以内联方式初始化引用类型的静态字段

当一个类型声明显式静态构造函数时，实时 (JIT) 编译器会向该类型的每个静态方法和实例构造函数中添加一项检查，以确保之前已调用该静态构造函数。 访问任何静态成员或创建该类型的实例时，将触发静态初始化。 但是，如果声明一个类型的变量，但不使用它，则不会触发静态初始化；这在初始化会更改全局状态的情况下非常重要。当所有静态数据都以内联方式初始化并且未声明显式静态构造函数时，Microsoft 中间语言 (MSIL) 编译器会将 beforefieldinit 标志和隐式静态构造函数（该构造函数初始化静态数据）添加到 MSIL 类型定义。 JIT 编译器遇到 beforefieldinit 标志时，大多数情况下不会添加静态构造函数检查。 静态初始化可以保证在访问任何静态字段之前的某个时间发生，但不能在调用静态方法或实例构造函数之前发生。 请注意，在声明类型的变量后，可能会随时发生静态初始化。

静态构造函数检查会降低性能。 通常，静态构造函数仅用于初始化静态字段，在这种情况下，必须确保仅在首次访问静态字段之前发生静态初始化。 beforefieldinit 行为适用于这些类型和大多数其他类型。 仅当静态初始化影响全局状态并且满足以下任一条件时，它才是不适当的：

* + 影响全局状态的成本非常昂贵，如果不使用该类型，则不需要这样做。
  + 可以在不访问该类型的任何静态字段的情况下访问全局状态效果。

/// <summary>

/// 违反规则的代码示例

/// </summary>

public class Example

{

private static readonly int \_number;

private static readonly string \_text;

static Example()

{

\_number = 10;

var textManager = new TextManager();

\_text = textManager.GetString();

}

}

/// <summary>

/// 符合规则的代码示例

/// </summary>

public class Example

{

private static readonly int \_number = 10;

private static readonly string \_text = Init();

private static string Init()

{

var textManager = new TextManager();

return textManager.GetString();

}

}

## 避免使用非密封特性

.NET 提供用于检索自定义特性的方法。 默认情况下，这些方法搜索特性继承层次结构。 例如，System.Attribute.GetCustomAttribute 搜索指定的特性类型或扩展指定特性类型的所有特性类型。 密封特性后，无需通过继承层次结构进行搜索，且能够提高性能。

/// <summary>

/// 违反规则的代码定义

/// </summary>

[AttributeUsage(AttributeTargets.Class | AttributeTargets.Struct)]

public sealed class ExampleAttribute : Attribute

{

private readonly string \_name;

public ExampleAttribute(string name)

{

\_name = name;

}

public string Name => \_name;

}

/// <summary>

/// 符合规则的代码示例

/// </summary>

[AttributeUsage(AttributeTargets.Class | AttributeTargets.Struct)]

public sealed class ExampleAttribute : Attribute

{

private readonly string \_name;

public ExampleAttribute(string name)

{

\_name = name;

}

public string Name => \_name;

}

## 与多维数组相比，首选使用交错数组

在多维数组中，每个维度中的每个元素都具有与该维度中的其他元素相同的固定大小。 在交错数组（即多个数组构成的数组）中，每个内部数组的大小可以不同。 通过仅使用给定数组所需的空间，就不会浪费空间。 建议优先使用交错数组以节省内存。

public class Example

{

/// <summary>

/// 交错数组

/// </summary>

int[][] jaggedArray =

{

new int[] {1,2,3,4},

new int[] {5,6,7},

new int[] {8},

new int[] {9}

};

/// <summary>

/// 多维数组

/// </summary>

int[,] multiDimArray =

{

{1,2,3,4},

{5,6,7,0},

{8,0,0,0},

{9,0,0,0}

};

}

## 属性不应返回数组

即使属性是只读的，该属性返回的数组也不受写入保护。 若要使数组不会被更改，属性必须返回数组的副本。

## 使用字符串长度测试是否有空字符串

使用 String.Length 属性或 String.IsNullOrEmpty 方法比较字符串比使用 Equals 更快。 这是因为 Equals 执行的 MSIL 指令比 IsNullOrEmpty 或执行以用于检索 Length 属性值并将其与零进行比较的指令数要多得多。对于 NULL 字符串，Equals 和 <string>.Length == 0 的行为不同。 如果尝试获取 NULL 字符串的 Length 属性值，则公共语言运行时将引发 System.NullReferenceException。 如果在 NULL 字符串和空字符串之间执行比较，则公共语言运行时不会引发异常，并将返回 false。 测试 NULL 不会对这两种方法的相对性能产生显著影响。 面向 .NET Framework 2.0 或更高版本时，请使用 IsNullOrEmpty 方法。 否则，请尽可能使用 Length == 0 比较。

public class Example

{

private readonly string \_str = "test";

public void EqualsTest()

{

// 违反规则的代码示例

if (\_str == "")

{

Console.WriteLine("\_str equals empty string.");

}

}

public void LengthTest()

{

// .NET Framework 1.0 and 1.1. 符合规则的代码示例

if (\_str != null && \_str.Length == 0)

{

Console.WriteLine("\_str.Length == 0.");

}

}

public void NullOrEmptyTest()

{

// .NET Framework 2.0.及以上符合规则的代码示例

if (!string.IsNullOrEmpty(\_str))

{

Console.WriteLine($"{nameof(\_str)} != null and {nameof(\_str)}.Length != 0.");

}

}

}

## 将成员标记为 static

可以将不访问实例数据或不调用实例方法的成员标记为静态。 在将这些方法标记为 static 之后，编译器将向这些成员发出非虚拟调用站点。 发出非虚拟调用网站将禁止在运行时检查每个调用，以确保当前对象指针为非 NULL。 这会使性能敏感的代码的性能得到显著提高。 在某些情况下，访问当前对象实例失败表示存在正确性问题。

## 避免数组分配长度为零

初始化长度为零的数组将导致不必要的内存分配。 请改为通过调用 Array.Empty 方法来使用静态分配的空数组实例。 内存分配在此方法的所有调用之间共享。

## 使用属性，而不是 Linq Enumerable 方法

此规则在具有等效但更高效的属性的类型集合上标记 Enumerable LINQ 方法调用，以提取相同的数据。此规则针对以下集合类型：

* 实现 IReadOnlyList<T> 但不实现 IList<T> 的类型

此规则针对这些集合类型对以下方法进行的调用：

* System.Linq.Enumerable.Count 方法
* System.Linq.Enumerable.First 方法
* System.Linq.Enumerable.FirstOrDefault 方法
* System.Linq.Enumerable.Last 方法
* System.Linq.Enumerable.LastOrDefault 方法

等等，具体以实际情况为准。

class Example

{

/// <summary>

/// 违反规则的代码示例

/// </summary>

/// <param name="list"></param>

public void Violates(IReadOnlyList<string> list)

{

Console.Write(list.First());

Console.Write(list.Last());

Console.Write(list.Count());

}

/// <summary>

/// 符合规则的代码示例

/// </summary>

/// <param name="list"></param>

public void Satisfies(IReadOnlyList<string> list)

{

Console.Write(list[0]);

Console.Write(list[list.Count - 1]);

Console.Write(list.Count);

}

}

## 如果可以使用 Any/AnyAsync，请勿使用 Count/LongCount/ CountAsync/LongCountAsync,

此规则将标记 Count / CountAsync和 LongCount/ LongCountAsync LINQ 方法调用，用于检查集合是否至少有一个元素。 这些方法调用需要枚举整个集合来计算计数。 使用 Any 方法进行相同的检查速度更快，因为它可以避免枚举集合。

## 使用 Length/Count 属性，而不是 Enumerable.Count 方法

此规则在具有等效但更高效的 Length 或 Count 属性以提取相同数据的类型的集合上标记 Count LINQ 方法调用。 Length 或 Count 属性不枚举集合，因此更高效。

## 在 StringBuilder 上优先使用强类型 Append 和 Insert 方法重载。

Append 和 Insert 为除 String 之外的多种类型提供重载。 在可能的情况下，请首先使用强类型重载，而不是使用 ToString () 和基于字符串的重载。使用单位长度字符串调用 StringBuilder.Append 时，请考虑使用 const char 而不是单位长度 const string 来提高性能

class Example

{

int \_value;

/// <summary>

/// 违反规则的代码示例

/// </summary>

/// <param name="sb"></param>

public void Violates(StringBuilder sb)

{

sb.Append("Value: “).Appeend(("a").Append(\_value.ToString()).AppendLine();

}

/// <summary>

/// 符合规则的代码示例

/// </summary>

/// <param name="destination"></param>

public void Satisfies(StringBuilder destination)

{

sb.Append("Value: ").Append('a').Append(\_value).AppendLine();

}

}

## 可用时最好使用 IsEmpty (而不是 Count)

当将 Count 和 Length 属性或 Count<TSource>(IEnumerable<TSource>) 和 LongCount<TSource>(IEnumerable<TSource>) LINQ 方法用于确定对象是否包含任何项以及对象是否具有更有效的 IsEmpty 属性时，优先使用此规则。

/// <summary>

/// 违反规则的代码示例

/// </summary>

class Example

{

ConcurrentQueue<int> \_queue;

public bool IsEmpty => \_queue.Count == 0;

}

/// <summary>

/// 符合规则的代码示例

/// </summary>

class Example

{

ConcurrentQueue<int> \_queue;

public bool IsEmpty => \_queue.IsEmpty;

}

## 首选字典包含方法

对 Keys 或 Values 集合调用 Contains 通常比对字典本身调用 ContainsKey 或 ContainsValue 开销更高：

* 许多字典实现会延迟对键值集合的实例化，这意味着访问 Keys 或 Values 集合可能导致额外的分配。
* 如果键/值集合使用显式接口实现来隐藏 ICollection<T> 上的方法，可能最终会对 IEnumerable<T> 上调用扩展方法。 这可能会降低性能，尤其是在访问键集合时。 大多数字典实现都能为键提供快速的 O(1) 包含检查，而 IEnumerable<T> 上的 Contains 扩展方法通常执行较慢的 O(n) 包含检查。

class Example

{

/// <summary>

/// 违反规则的代码示例

/// </summary>

void Violates()

{

var dictionary = new Dictionary<string, int>();

dictionary.Keys.Contains("hello world");

dictionary.Values.Contains(17);

}

/// <summary>

/// 符合规则的代码示例

/// </summary>

void Satisfies()

{

var dictionary = new Dictionary<string, int>();

dictionary.ContainsKey("hello world");

dictionary.ContainsValue(17);

}

}

## 对“流”进行子分类时，提供异步方法的基于内存的重写

添加了基于内存的 ReadAsync 和 WriteAsync 方法来提高性能，这些方法的实现方式有多种：

* 它们分别返回 ValueTask 和 ValueTask<int>，而不是 Task 和 Task<int>。
* 它们允许传入任意类型的缓冲区，而无需对数组执行额外的复制。

为了实现这些性能优势，派生自 Stream 的类型必须提供自己的基于内存的实现。 否则，将强制默认实现将内存复制到数组中，以便调用基于数组的实现，从而降低性能。 当调用方传入不受数组支持的 Memory<T> 或 ReadOnlyMemory<T> 实例时，性能会受到影响。

## 使用基于跨度的“string.Concat” 。首选 AsSpan 次选 Substring

调用 Substring 会生成提取的子字符串的副本。 通过使用 AsSpan 代替 Substring，并调用接受跨度的 string.Concat 重载，可以消除不必要的字符串分配。Substring 在堆上分配一个新的 string 对象并执行提取文本的完整副本。 字符串操作是多个程序的性能瓶颈。 在热路径上分配很多生存期较短的小型字符串可以产生足够的集合压力来影响性能。 当子字符串变大时，Substring 创建的 O(n) 副本将变得举足轻重。 为了解决这些性能问题，创建了 Span<T> 和 ReadOnlySpan<T> 类型。许多接受字符串的 API 也具有接受 ReadOnlySpan<System.Char> 参数的重载。 当此类重载可用时，可以通过调用 AsSpan 而不是 Substring 来提升性能。

class Example

{

/// <summary>

/// 违法规则的示例

/// </summary>

void Violates()

{

string text = "11111 abc !@#$%^&\*";

// Violation: allocations by Substring are wasteful.

string s1 = text.Substring(10) + "---" + text.Substring(0, 5);

int.TryParse(text.Substring(10), out int x);

int.TryParse(text.Substring(2, 3), out int y);

}

/// <summary>

/// 符合规则的示例

/// </summary>

void Satisfies()

{

string text = "11111 abc !@#$%^&\*";

string s2 = string.Concat(text.AsSpan(10), "---", text.AsSpan(0, 5));

int.TryParse(text.AsSpan(10), out int x);

int.TryParse(text.AsSpan(2, 3), out int y);

}

}

## 对单个字符使用 string.Contains(char) 而不是 string.Contains(string)

在搜索单个字符时，使用 string.Contains(char) 可获得比使用 string.Contains(string) 时更好的性能。

# 安全规则

安全规则可实现更安全的库和应用程序。 这些规则有助于防止程序中出现安全漏洞。如果书写的代码不符合其中的规则，你应该在代码中清楚的标记原因，并通知开发项目的指定安全负责人。

## 检查 SQL 查询是否存在安全漏洞

此规则假定无法在编译时确定值的任何字符串都可能包含用户输入。 基于用户输入生成的 SQL 命令字符串易于受到 SQL 注入式攻击。 在 SQL 注入攻击中，恶意用户会提供改变查询设计的输入，企图破坏基础数据库或对该数据库进行未经授权的访问。 典型方法包括注入一个单引号或撇号（这是 SQL 文本字符串分隔符）、两个短划线（表示 SQL 注释）和一个分号（指示后跟一个新命令）。 如果用户输入必须是查询的一部分，请按照以下方法之一（按有效性排列）来降低遭受攻击的风险。

* 使用存储过程。
* 使用参数化命令字符串。
* 在生成命令字符串之前，先验证用户输入的类型和内容。

下面的 .NET 类型实现 CommandText 属性，或提供使用字符串参数设置属性的构造函数。

System.Data.Odbc.OdbcCommand 和 System.Data.Odbc.OdbcDataAdapter

System.Data.OleDb.OleDbCommand 和 System.Data.OleDb.OleDbDataAdapter

System.Data.OracleClient.OracleCommand 和 System.Data.OracleClient.OracleDataAdapter

System.Data.SqlClient.SqlCommand 和 System.Data.SqlClient.SqlDataAdapter

public class SqlQueries

{

public object UnsafeQuery(string connectionString, string name, string password)

{

using var connection = new SqlConnection(connectionString);

var cmdText = "SELECT Id FROM Users WHERE Username='" + name + "' AND Password='" + password + "'";

SqlCommand cmd = new(cmdText, connection);

connection.Open();

object id = cmd.ExecuteScalar();

connection.Close();

return id;

}

public object SaferQuery(string connectionString, string name, string password)

{

using var connection = new SqlConnection(connectionString);

var cmdText = "SELECT Id FROM Users WHERE Username=@username AND Password=@password";

SqlCommand cmd = new(cmdText, connection);

cmd.Parameters.Add(new SqlParameter[]

{

new SqlParameter("@username", SqlDbType.VarChar,20,name),

new SqlParameter("@password", SqlDbType.VarChar,20,password)

});

connection.Open();

object id = cmd.ExecuteScalar();

connection.Close();

return id;

}

}

class Malicious

{

static void Main(string[] args)

{

var queries = new SqlQueries();

// SELECT AccountNumber FROM Users WHERE Username='' OR 1=1

queries.UnsafeQuery(args[0], "' OR 1=1 --", "password");

// SELECT AccountNumber FROM Users WHERE Username=''' OR 1=1 --' AND Password='password'

queries.SaferQuery(args[0], "' OR 1=1 --", "password");

}

}

## 请勿使用不安全的反序列化程序 BinaryFormatter

反序列化不受信任的数据时，不安全的反序列化程序易受攻击。 攻击者可能会修改序列化数据，使其包含非预期类型，进而注入具有不良副作用的对象。 例如，针对不安全反序列化程序的攻击可以在基础操作系统上执行命令，通过网络进行通信，或删除文件。反序列化漏洞是指不安全地处理请求有效负载的威胁类别。 成功利用这些漏洞攻击应用的攻击者可导致目标应用内出现拒绝服务 (DoS)、信息泄露或远程代码执行。成功的攻击可能导致攻击者能够在目标进程的上下文中运行代码。更简单的比喻是，假设在有效负载上调用 BinaryFormatter.Deserialize 相当于将该有效负载解释为独立的可执行文件并启动它。

## 反序列化对象图中存在潜在引用循环

反序列化不受信任的数据时，处理反序列化对象图的任何代码都需要在处理引用循环时不进入无限循环。 这包括反序列化回叫中的一部分代码和在反序列化完成后处理对象图的代码。 否则攻击者可能会使用包含引用循环的恶意数据执行拒绝服务攻击。

## 请勿使用 None 以外的 TypeNameHandling 值

原因同上，可能造成信息泄露或远程代码执行，若要使用None以外的之，请设置SerializationBinder，此时，该值不应该是null

## 确保 DataTable.ReadXml() 的输入受信任

反序列化具有不受信任输入的 DataTable 时，攻击者可创建恶意输入来实施拒绝服务攻击。 有可能存在未知的远程代码执行漏洞。

## 可序列化类型中的不安全 DataSet 或 DataTable 容易受到远程代码执行攻击

当反序列化具有不受信任输入且反序列化的对象图（Object image）包含 DataSet 或 DataTable 时，攻击者可能创建执行远程代码执行攻击的恶意有效负载。如果可能，请使用实体框架，而不是 DataSet 和 DataTable。或者使序列化的数据免被篡改。 序列化后，对序列化的数据进行加密签名。 在反序列化之前，验证加密签名。

## 请勿序列化具有 Pointer 字段的类型

Pointer 不是类型安全的，这意味着你无法保证它们所指向的内存的正确性。 因此，序列化具有 Pointer 字段的类型会带来安全风险，它可能允许攻击者控制指针。

## 查看 XSS 漏洞的代码

在处理来自 Web 请求的不受信任的输入时，请注意防范跨站脚本 (XSS) 攻击。 XSS 攻击会将不受信任的输入注入原始 HTML 输出，使攻击者可以执行恶意脚本或恶意修改网页中的内容。 一个典型的技术是将包含恶意代码的 <script> 元素放入输入中。

## 查看文件路径注入漏洞的代码

在处理来自 Web 请求的不受信任的输入时，请谨慎使用用户控制的输入指定文件路径。 攻击者可能能够读取非预期文件，从而导致敏感数据出现信息泄漏。 或者，攻击者可能能够写入非预期文件，从而导致在未经授权的情况下修改敏感数据，或者降低服务器的安全性。 常见的攻击者技术是使用路径遍历访问预期目录之外的文件。

## 查看信息泄露漏洞的代码

泄漏异常信息可让攻击者深入了解应用程序的内部机制，从而帮助攻击者找到其他漏洞并利用这些漏洞。

## 查看进程命令注入漏洞的代码

处理不受信任的输入时，请注意防范命令注入攻击。 命令注入攻击可在基础操作系统上执行恶意命令，从而降低服务器的安全和完整性。

## 查看公开重定向漏洞的代码

处理不受信任的输入时，请注意防范开放重定向漏洞。 攻击者可以利用开放重定向漏洞，使用你的网站提供合法 URL 的外观，但将毫不知情的访客重定向到钓鱼网页或其他恶意网页。

## 查看 XPath 注入漏洞的代码

处理不受信任的输入时，请注意防范 XPath 注入攻击。 使用不受信任的输入构造 XPath 查询可能会允许攻击者恶意控制查询，使其返回一个意外的结果，并可能泄漏查询的 XML 的内容。

## 查看 XML 注入漏洞的代码

处理不受信任的输入时，请注意防范 XML 注入攻击。 攻击者可以使用 XML 注入向 XML 文档中插入特殊字符，使文档变成无效的 XML。 或者，攻击者可能会恶意插入其所选的 XML 节点。

## 查看 DLL 注入漏洞的代码

处理不受信任的输入时，请谨慎加载不受信任的代码。 如果你的 Web 应用加载不受信任的代码，攻击者可能能够将恶意 DLL 注入到你的进程中，并执行恶意代码。

## 查看正则表达式注入漏洞的代码

处理不受信任的输入时，请注意防范正则表达式注入攻击。 攻击者可以使用正则表达式注入恶意修改正则表达式，让正则表达式匹配非预期结果，或者让正则表达式占用过多 CPU，从而形成拒绝服务攻击。

## 尽可能减少使用弱加密算法,损坏的加密算法

DES, RC2, TripleDES, DSA 等加密算法和 SHA1 及 RIPEMD160, MD5 等哈希算法被视为弱加密算法。可能会带来重大风险，并可能通过普通的攻击技术（如暴力攻击和哈希冲突）导致暴露敏感信息。对于 DES, RC2, TripleDES 加密，请使用 Aes 加密。对于 SHA1 或 RIPEMD160 , MD5哈希函数，请从 SHA-2 系列（例如 SHA512、 SHA384、 SHA256）中选择使用。尽管具体的漏洞因使用环境的不同而异，但 MD5 哈希算法仍易遭到已知的冲突攻击。 用于确保数据完整性的哈希算法（例如文件签名或数字证书）特别容易受到攻击。 在这种情况下，攻击者可能会生成两个独立的数据块，以便在不更改哈希值或使相关数字签名无效的情况下，将良性数据替换为恶意数据。对于加密算法：

* DES 加密使用的密钥强度较低，可能在一天内被暴力破解。
* RC2 加密容易遭受与密钥相关的攻击，攻击者可以通过这些攻击找出所有密钥值之间的数学关系。
* DSA 是一种弱非对称加密算法。请改为使用密钥大小至少为 2048 的 RSA、ECDH 或 ECDsa 算法。小于 2048 位的 RSA 密钥更容易受到暴力攻击。
* 对称加密应始终使用不可重复的初始化向量，以防止字典攻击。

## 请勿使用不安全的密码模式

ECB, OFB, CFB这些模式容易受到攻击，并可能导致敏感信息泄露。 例如，使用 ECB 对纯文本块进行加密始终都会生成相同的密码文本，因此它可以轻松判断两个加密消息是否相同。 使用批准的模式(CBC,CTS)可以避免这些不必要的风险。

## 使用弱密钥派生功能时，请确保迭代计数足够大

通过 System.Security.Cryptography.Rfc2898DeriveBytes.GetBytes 派生加密密钥时,应保证迭代计数不小于100,000 。 较高的迭代计数有助于缓解尝试猜测已生成的加密密钥的字典攻击。

class Example

{

/// <summary>

/// 违法规则的代码示例

/// </summary>

/// <param name="password"></param>

/// <param name="salt"></param>

/// <param name="cb"></param>

public void Violates(string password, byte[] salt, int cb)

{

var rfc2898DeriveBytes = new Rfc2898DeriveBytes(password, salt);

rfc2898DeriveBytes.GetBytes(cb);

}

/// <summary>

/// 符合规则的代码示例

/// </summary>

/// <param name="password"></param>

/// <param name="salt"></param>

/// <param name="cb"></param>

public void Satisfies(string password, byte[] salt, int cb)

{

var rfc2898DeriveBytes = new Rfc2898DeriveBytes(password, salt)

{

IterationCount = 100000

};

rfc2898DeriveBytes.GetBytes(cb);

}

}

## 请勿使用不安全的随机性

如果使用加密较弱的伪随机数生成器，攻击者可以预测将要生成的安全敏感值。

将Random.Next改为RandomNumberGenerator. GetInt32

## 请勿编码加密密钥

要成功使用对称算法，密钥必须只有发送方和接收方知道。 如果密钥是硬编码的，就容易被发现。 即使使用编译的二进制文件，恶意用户也容易将其提取出来。 私钥泄露后，密码文本可直接被解密并且不再受保护。

# 代码样式规则

代码样式规则支持应用程序的可维护性和阅读性，尽可能减少不必要的维护成本使应用程序源码更规范标准

# 附录：

## string.IndexOf(string) 从 .NET Core 3.0 -> .NET 5.0的中断性更改

在不同平台下,string.IndexOf会产生不同结果，具体原因参考以下链接

1. [Breaking change with string.IndexOf(string) from .NET Core 3.0 -> .NET 5.0 · Issue #43736 · dotnet/runtime (github.com)](https://github.com/dotnet/runtime/issues/43736)
2. [全球化和 ICU | Microsoft Docs](https://docs.microsoft.com/zh-cn/dotnet/core/extensions/globalization-icu)
3. [在 .NET 5 及更高版本中比较字符串时的行为更改 | Microsoft Docs](https://docs.microsoft.com/zh-cn/dotnet/standard/base-types/string-comparison-net-5-plus)