**C# 编 码 规 范**

**石家庄铁道大学**

**20170600 赵浩森 土1703-1**

1. 简介

本规范为一套编写高效可靠的 C# 代码的标准、约定和指南。它以安全可靠的软件工程原则为基础，使代码易于理解、维护和增强，提高生产效率。同时，将带来更大的一致性，使软件开发团队的效率明显提高。

2. 适用范围

本规范适用于公司所有的C#源代码，为详细设计，代码编写和代码审核提供参考和依据。

3. 文体

本规范中的建议分为四种：**要，建议，避免，不要**，表示需要遵循的级别。文档中会以粗体表示。对于应遵循的规范，前面会以“Ö”来表示，对不好的做法前面会以“´”来表示：

**要**：描述必须遵循的规范。例如：

Ö 异常类**要**以“Exception”做为后缀；

**建议**：描述在一般情况下应该遵循的规范，但如果完全理解规范背后的道理，并有很好的理由不遵循它时，也不畏惧打破常规。例如：

Ö 强制类型转换时，在类型和变量之间**建议**加一空格。

**不要**：描述一些几乎绝对绝不应该违反的规范。例如：

´ 每个函数有效代码（不包括注释和空行）长度**不要**超过50行。

**避免**：与**建议**相对，一般情况下应该遵循，但有很好的理由时也可以打破。例如：

´ **避免**块内部的变量与它外部的变量名相同。

对一些规范内容一并提供了示例代码。

4. 代码组织与风格

4.1. Tab

Ö **要**使一个Tab为4个空格长。

4.2. 缩进

Ö **要**使一个代码块内的代码都统一缩进一个Tab长度。

4.3. 空行

Ö **建议**适当的增加空行，来增加代码的可读性。

Ö 在在类，接口以及彼此之间**要**有两行空行：

Ö 在下列情况之间**要**有一行空行：

方法之间；

局部变量和它后边的语句之间；

方法内的功能逻辑部分之间；

4.4. 函数长度

´ 每个函数有效代码（不包括注释和空行）长度**不要**超过50行。

4.5. {”，“}”

Ö 开括号“{”**要**放在块的所有者的下一行，单起一行；

Ö 闭括号“}”**要**单独放在代码块的最后一行，单起一行。

4.6. 行宽

´ 每行代码和注释**不要**超过70个字符或屏幕的宽度，如超过则应换行，换行后的代码应该缩进一个Tab。

4.7. 空格

´ 括号和它里面的字符之间**不要**出现空格。括号应该和它前边的关键词留有空格，如：while (true) {};

´ 但是方法名和左括号之间**不要**有空格。

Ö 参数之间的逗号后**要**加一空格。如：method1(int i1, int i2)

Ö for语句里的表达式之间**要**加一空格。如：for (expr1; expr2; expr3)

Ö 二元操作符和操作数之间**要**用空格隔开。如：i + c;

Ö 强制类型转换时，在类型和变量之间**要**加一空格。如：(int) i ;

5. 注释

5.1. 注释的基本约定

Ö 注释应该增加代码的清晰度；

Ö 保持注释的简洁，不是任何代码都需要注释的，过多的注释反而会影响代码的可读性。

´ 注释**不要**包括其他的特殊字符。

Ö **建议**先写注释，后写代码，注释和代码一起完成

Ö 如果语句块（比如循环和条件分枝的代码块）代码太长，嵌套太多，则在其结束“｝”**要**加上注释，标志对应的开始语句。如果分支条件逻辑比较复杂，也**要**加上注释。

Ö 在VS2005环境中通过配置工程编译时输出XML文档文件可以检查注释的完整情况，如果注释不完整会报告编译警告；

5.2. 注释类型

5.2.1. 块注释

Ö 主要用来描述文件，类，方法，算法等，放在所描述对象的前边。具体格式以IDE编辑器输入“///”自动生成的格式为准，另外再附加我们自定义的格式，如下所列：

/// <Remark>作者，创建日期，修改日期</ Remark >

对类和接口的注释必须加上上述标记，对方法可以视情况考虑

5.2.2. 行注释

Ö 主要用在方法内部，对代码，变量，流程等进行说明。整个注释占据一行。

5.2.3. 尾随注释

Ö 与行注释功能相似，放在代码的同行，但是要与代码之间有足够的空间，便于分清。例：

int m = 4 ; /／ 注释

Ö 如果一个程序块内有多个尾随注释，每个注释的缩进**要**保持一致。

5.3. 注释哪些部分

|  |  |
| --- | --- |
| **项目** | **注释哪些部分** |
| 参数 | 参数用来做什么  任何约束或前提条件 |
| 字段/属性 | 字段描述 |
| 类 | 类的目的  已知的问题  类的开发/维护历史 |
| 接口 | 目的  它应如何被使用以及如何不被使用 |
| 局部变量 | 用处/目的 |
| 成员函数注释 | 成员函数做什么以及它为什么做这个  哪些参数必须传递给一个成员函数  成员函数返回什么  已知的问题  任何由某个成员函数抛出的异常  成员函数是如何改变对象的  包含任何修改代码的历史  如何在适当情况下调用成员函数的例子适用的前提条件和后置条件 |
| 成员函数内部注释 | 控制结构  代码做了些什么以及为什么这样做  局部变量  难或复杂的代码  处理顺序 |

5.4. 程序修改注释

Ö 新增代码行的前后**要**有注释行说明，对具体格式不作要求，但必须包含作者，新增时间，新增目的。在新增代码的最后必须加上结束标志；

Ö 删除代码行的前后**要**用注释行说明，删除代码用注释原有代码的方法。注释方法和内容同新增；删除的代码行**建议**用#region XXX #endregion 代码段折叠，保持代码文件干净整洁

Ö 修改代码行**建议**以删除代码行后再新增代码行的方式进行（针对别人的代码进行修改时，必须标明，对于自己的代码进行修改时，酌情进行）。注释方法和内容同新增；

6. 命名

6.1. 命名的基本约定

Ö **要**使用可以准确说明变量/字段/类的完整的英文描述符，如firstName。对一些作用显而易见的变量可以采用简单的命名，如在循环里的递增（减）变量就可以被命名为 ” i ”。

Ö **要**尽量采用项目所涉及领域的术语。

Ö **要**采用大小写混合，提高名字的可读性。为区分一个标识符中的多个单词，把标识符中的每个单词的首字母大写。不采用下划线作分隔字符的写法。有两种适合的书写方法，适应于不同类型的标识符：

PasalCasing：标识符的第一个单词的字母大写；

camelCasing：标识符的第一个单词的字母小写。

下表描述了不同类型标识符的大小写规则：

|  |  |  |
| --- | --- | --- |
| **标识符** | **大小写** | **示例** |
| 命名空间 | Pascal | namespace Com.Techstar.ProductionCenter |
| 类型 | Pascal | public class DevsList |
| 接口 | Pascal | public interface ITableModel |
| 方法 | Pascal | public void UpdateData() |
| 属性 | Pascal | Public int Length{…} |
| 事件 | Pascal | public event EventHandler Changed; |
| 私有字段 | Camel | private string fieldName; |
| 非私有字段 | Pascal | public string FieldName； |
| 枚举值 | Pascal | FileMode{Append} |
| 参数 | Camel | public void UpdateData(string fieldName) |
| 局部变量 | Camel | string fieldName; |

´ **避免**使用缩写，如果一定要使用，就谨慎使用。同时，应该保留一个标准缩写的列表，并且在使用时保持一致。

Ö 对常见缩略词，两个字母的缩写**要**采用统一大小写的方式（示例：ioStream，getIOStream）；多字母缩写采用首字母大写，其他字母小写的方式（示例：getHtmlTag）；

´ **避免**使用长名字（最好不超过 15 个字母）。

´ **避免**使用相似或者仅在大小写上有区别的名字。

6.2. 各种标示符类型的命名约定

6.2.1. 程序集命名

Ö 公司域名（Techstar）+ 项目名称 + 模块名称（可选），例如：

中心系统程序集：Techstar.ProductionCenter；

中心系统业务逻辑程序集：Techstar. ProductionCenter.Business；

6.2.2. 命名空间命名

Ö 采用和程序集命名相同的方式：公司域名（Techstar）+ 项目名称 + 模块名称。 另外，一般情况下建议命名空间和目录结构相同。例如：

中心系统：Techstar.ProductionCenter；

中心系统下的用户控件：Techstar.ProductionCenter.UserControl；

中心系统业务逻辑：Techstar. ProductionCenter.Business；

中心系统数据访问：Techstar. ProductionCenter.Data；

6.2.3. 类和接口命名

Ö 类的名字**要**用名词；

´ 避免使用单词的缩写，除非它的缩写已经广为人知，如HTTP。

Ö 接口的名字**要**以字母I开头。保证对接口的标准实现名字只相差一个“I”前缀，例如对IComponent的标准实现为Component；

Ö 泛型类型参数的命名：命名**要**为T或者以T开头的描述性名字，例如：

public class List<T>

public class MyClass<TSession>

´ 对同一项目的不同命名空间中的类，命名**避免**重复。避免引用时的冲突和混淆；

6.2.4. 方法命名

Ö 第一个单词一般是动词

Ö 如果方法返回一个成员变量的值，方法名一般为Get+成员变量名，如若返回的值 是bool变量，一般以Is作为前缀。另外，如果必要，考虑用属性来替代方法，具 体建议见10.1.2节;

Ö 如果方法修改一个成员变量的值，方法名一般为：Set + 成员变量名。同上，考虑 用属性来替代方法；

6.2.5. 变量命名

Ö 按照使用范围来分，我们代码中的变量的基本上有以下几种类型，类的公有变量；类的私有变量（受保护同公有）；方法的参数变量；方法内部使用的局部变量。这些变量的命名规则基本相同，见标识符大小写对照表。区别如下：

i. 类的公有变量按通常的方式命名，无特殊要求；

ii. 类的私有变量采用两种方式均可：采用加“m”前缀，例如mWorkerName;

iii. 方法的参数变量采用camalString，例如workerName；

iv. 方法内部的局部变量采用camalString，例如workerName；

´ **不要**用\_或&作为第一个字母；

Ö 尽量**要**使用短而且具有意义的单词；

Ö 单字符的变量名一般只用于生命期非常短暂的变量。i,j,k,m,n一般用于integer；c,d,e 一般用于characters；s用于string

Ö 如果变量是集合，则变量名**要**用复数。例如表格的行数，命名应为：RowsCount；

Ö 命名组件**要**采用匈牙利命名法，所有前缀均应遵循同一个组件名称缩写列表

6.3. 组件名称缩写列表

缩写的基本原则是取组件类名各单词的第一个字母，如果只有一个单词，则去掉其中的元音，留下辅音。缩写全部为小写。

|  |  |  |
| --- | --- | --- |
| 组件类型 | 缩写 | 例子 |
| Label | Lbl | lblNote |
| TextBox | Txt | txtName |
| Button | Btn | btnOK |
| ImageButton | Ib | ibOK |
| LinkButton | Lb | lbJump |
| HyperLink | Hl | hlJump |
| DropDownList | Ddl | ddlList |
| CheckBox | Cb | cbChoice |
| CheckBoxList | Cbl | cblGroup |
| RadioButton | Rb | rbChoice |
| RadioButtonList | Rbl | rblGroup |
| Image | Img | imgBeauty |
| Panel | Pnl | pnlTree |
| TreeView | Tv | tvUnit |
| WebComTable | Wct | wctBasic |
| ImageDateTimeInput | Dti | dtiStart |
| ComboBox | Cb | cbList |
| MyImageButton | Mib | mibOK |
| WebComm.TreeView | Tv | tvUnit |
| PageBar | Pb | pbMaster |

7. 声明

Ö 每行**要**只有一个声明，如果是声明i,j,k之类的简单变量可以放在一行;

Ö 除了for循环外，声明**要**放在块的最开始部分。for循环中的变量声明可以放在for语句中。如：for(int i = 0; I < 10; i++) 。

´ **避免**块内部的变量与它外部的变量名相同。

8. 表达式和语句

Ö 每行建议只有一条语句。

Ö if-else,if-elseif语句，任何情况下，都应该有“{”，“}”，格式如下：

if (condition)

{

statements;

}

else if (condition)

{

statements;

}

else

{

statements;

}

Ö for语句格式如下：

for (initialization; condition; update)

{

statements;

}

如果语句为空：

for (initialization; condition; update) ;

Ö while语句格式如下：

while (condition)

{

statements;

}

如果语句为空:

while (condition);

Ö do-while语句格式如下：

do

{

statements;

}

while (condition);

Ö switch语句，每个switch里都应包含default子语句,格式如下：

switch (condition)

{

case ABC:

statements;

/\* falls through \*/

case DEF:

statements;

break;

case XYZ:

statements;

break;

default:

statements;

break;

}

Ö try-catch语句格式如下：

try

{

statements;

}

catch (ExceptionClass e)

{

statements;

}

finally

{

statements;

}

9. 类型设计规范

Ö **要**确保每个类型由一组定义明确，相互关联的成员组成，而不仅仅是一些无关功能的随 机集合；

9.1. 类型和命名空间

Ö **要**用命名空间把类型组织成相关域的层次结构。例如：

界面层：Techstar.ProductionCenter；

业务逻辑层：Techstar.ProductionCenter.Business；

数据访问层：Techstar.ProductionCenter.Data；

´ **避免**过深的命名空间；

´ **避免**太多的命名空间；

9.2. 类型和接口的选择

Ö **要**优先采用类而不是接口。

接口的缺点在于语义变化时改变困难。注意接口并不是协定，把协定和实现分开并非一 定用接口实现，用基类和抽象类同样可以表达；

Ö **建议**使用抽象类而不是接口来解除协定与实现间的偶合；

Ö **要**定义接口，来实现类似多重继承的效果；

精心定义接口的标志是一个接口只做一件事情。关键是接口的协定需要保持不变， 如果一个接口包含太多功能，那么这个胖接口产生变化的机会就会大得多。

9.3. 抽象类设计：

´ **不要**在抽象类中定义公有的或内部受保护的构造函数。因为抽象类无法实例化，所以这 种设计会误导用户；

Ö **要**为抽象类定义受保护的构造函数或内部构造函数；

9.4. 静态类设计

静态类是一个只包含静态成员的类，它提供了一种纯面向对象设计和简单性之间的一个权衡，广泛用来提供类似于全局变量或一些通用功能。

Ö **要**少用静态类。静态类应该仅用作辅助类；

´ **避免**把静态类当作杂物箱。每个静态类都应该有其明确目的；

Ö **不要**在静态类中声明或覆盖实例成员；

9.5. 枚举设计

Ö **要**用枚举来加强那些表示值的集合的参数，属性以及返回值的类型性；

Ö **要**优先使用枚举而不是静态常量。例如：

//不好的写法

public static class Color

{

public static int Red = 0;

public static int Green = 1;

public static int Blue = 2;

}

//好的写法

public enum Color

{

Red,

Green,

Blue

}

´ **不要**把枚举用于开放的场合，例如操作系统的版本，朋友的名字等；

´ **枚举最后一个值不要加逗号；**

´ 枚举中**不要**提供为了今后使用而保留的枚举值；

10. 成员设计规范

方法，属性，事件，构造函数以及字段等统称为成员。

10.1. 成员设计的一般规范

10.2. 方法的重载规范；

´ **避免**在重载中随意的给参数命名。如果两个重载中的某个参数表示相同的输入，那么该参数的名字应该相同。例如：

public class String

{

//好的写法

public int IndexOf(string value) { ...}

public int IndexOf(string value, int startIndex) { ...}

//不好的写法

public int IndexOf(string value) { ...}

public int IndexOf(string str, int startIndex) { ...}

}

´ **避免**使重载成员的参数顺序不一致。在所有的重载中，同名参数应该出现在相同的位置。 例如：

public class EventLog

{

public EventLog();

public EventLog(string logName);

public EventLog(string logName, string machineName);

public EventLog(string logName, string machineName, string source);

}

Ö 较短的重载应该仅仅调用较长的来实现。另外，重载如果需要扩展性，把最长重载 做成虚函数。例如：

public class String

{

public int IndexOf(string s)

{

//调用

return IndexOf(s, 0);

}

public int IndexOf(string s, int startIndex)

{

//调用

return IndexOf(s, startIndex, s.Length);

}

public virtual int IndexOf(string s, int startIndex, int Count)

{

//实际的代码

}

}

Ö **要**允许可选参选为null。这样做是为了避免调用者调用之前需要检查参数是否null。例 如：

//允许为null时的调用

DrawGeometry(brush, pen, geometry);

//不允许为null时的调用

if (geometry == null) DrawGeometry(brush, pen);

else DrawGeometry(brush, pen, geometry);

10.3. 属性和方法的选择

Ö 基本原则是方法表示操作，属性表示数据。如果其他各方面都一样，优先使用属性而不 是方法。

Ö **要**使用属性，如果该成员表示类型的逻辑attribue

Ö 如果属性的值存储在内存中，而提供属性的目的仅仅是为了访问该值，**要**使用属性而不 要使用方法

Ö 如果该操作每次返回的结果不同，那么**要**使用方法。例如来自于.net framework的例子:

//好的写法

Guid.NewGuid();

//不好的写法

DateTime.Now;

Ö 如果该操作比访问字段慢一个或多个数量级，**要**使用方法。

Ö 如果该操作有严重的副作用，**要**使用方法。

10.4. 属性的设计规范：

Ö 如果不应该让调用方法改变属性值，**要**创建只读属性；

´ **不要**提供只写属性；

Ö **要**为所有的属性提供合理的默认值，这样可以确保默认值不会导致漏洞或效率低的代 码；

Ö **要**允许用户以任何顺序来设置属性的值；

Ö **避免**在属性的获取方法抛出异常。

属性的获取方法应该是个简单的操作，不应该有任何的条件。如果一个获取方法会抛出 异常，按么可能它更应该设计为方法。

10.5. 构造函数的设计规范

Ö **建议**提供简单的构造函数，最好是默认构造函数。简单的构造函数增强易用性；

Ö 考虑扩展性，如果构造函数设计的不自然，**建议**用静态的工厂方法来替代构造函数；

Ö **要**把构造函数的参数用作设置主要属性的便捷方法。如果构造函数参数仅用来设置属 性，应和属性名称相同。仅有大小写的区别；

Ö **要**在构造函数中做最少的工作。任何其他处理应该推迟到需要的时候；

Ö **要**在类中显示的声明公用的默认构造函数，如果这样的构造函数是必须的。

如果没有显示默认构造函数，填加有参数构造函数时往往会破坏已有使用默认构造函数 的代码；

´ **避免**在对象的构造函数内部调用虚成员。这样在扩展设计的时候会导致难以理解的现 象；

10.6. 字段设计规范

´ **不要**提供公有的或受保护的字段。代之以属性来访问字段；

Ö **要**只用常量字段来表示永远不会改变的量。否则会导致兼容性问题。下面是正确的例子：

public struct Int32

{

public const int MaxValue = 0x7fffffff;

public const int MinValue = unchecked((int)0x80000000);

}

Ö **要**用公有的静态只读字段来定义预定义的对象实例。例如：

public struct Color

{

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

}

10.7. 参数的设计规范

Ö **要**用类结构层次中最接近基类类型来作为参数的类型，同时要保证该类型能够提供成员 所需的功能。例如：

要设计一个集合遍历的方法，那么参数应该是IEnbumerable为参数，而不应该是IList， 这样方法具有更强的适应性。

´ **不要**使用保留参数。如果将来需要更多的参数，那么可以增加重载成员。例如：

//不好的写法

public void Method(string reserved, SomeOption option);

//好的写法

public void Method(SomeOption option);

//将来填加

public void Method(SomeOption option, string path);

10.7.1. 参数设计中枚举和布尔参数的选择规范

Ö **要**用枚举。在代码阅读，书写中，枚举都比布尔的可读性好很多。例如：

//使用布尔型，阅读的时候不会轻易了解参数的含义

FileStream f = File.Open(“1.txt”, true, false);

//使用枚举型

FileStream f = File.Open(“1.txt”，CasingOptions.CaseSenstive, FileMode.Open);

´ **不要**使用布尔参数，除非百分之百肯定绝对不需要两个以上的值。即使此时，采用枚举 往往也可以提供更好的可读性，如上例。

Ö 考虑在构造函数中，对确实只有两种状态值的参数以及用来初始化布尔属性的参数使用 布尔类型；

10.7.2. 参数验证的规范：

Ö **要**验证传给公有的，受保护的或显示成员的参数是否合法。如果验证失败，应该抛出 System.ArgutmentException或其子类；

Ö **要**抛出System.ArgutmentNullException，如果传入的null，而该成员不支持null；

10.7.3. 参数传递的规范：

´ **避免**使用输出参数或引用参数；

11. 扩展性设计规范

´ 如果没有恰当理由，**不要**把类密封起来。这些理由包括：

A）类为静态类；

B）类的受保护成员保存了高度机密信息；

C）类继承了许多虚成员，逐个密封的代价太高，不如密封整个类；

D）不要在密封类中声明保护成员或虚成员，因为无法覆盖其实现；

Ö **建议**用保护成员用于高级定制。它提供了扩展性，同时也避免了公用接口过于复杂；

´ **不要**使用虚成员，除非有合适的理由；

Ö **建议**只有在绝对必须的时候才用虚成员提供扩展性，并使用Template Method模式；

Ö **要**优先使用受保护的虚成员，而不是公有虚成员。公有成员通用调用受保护的虚成员的方式来提供扩展性；

12. 异常处理规范

Ö 异常的思想是只对错误采用异常处理：逻辑和编程错误，设置错误，被破坏的数据，资源耗尽，等等。通常的法则是系统在正常状态下以及无重载和硬件失效状态下，不应产生任何异常。**异常处理时**可以采用适当的日志机制来报告异常，包括异常发生的时刻；

´ 一般情况下**不要使用异常实现来控制程序流程结构；**

´ **使用异常而不要用错误代码来报告错误；**

Ö **要通过抛出异常的方式来报告操作失败。如果成员无法成功地完成它应该做的任务，那么应该抛出异常；**

**12.1. 异常类型选择规范**

Ö **优先考虑使用System命名空间中已有的异常，而不是自己创建新的异常类型；**

Ö **要使用最合理，最具针对性的异常。例如，对参数为空，应抛出**System.ArgutmentNullException，而不是System.ArgutmentException

**12.2. 异常处理规范**

´ 不是百分之百确定的情况，**不要**吞掉异常；

Ö **建议**捕获特定类型的异常，如果理解该异常在具体环境当中产生的原因；

´ **不要**捕获不应该捕获的异常，通常应该允许异常沿着调用栈传递；

Ö 进行清理工作时**要**用try-finally，避免使用try-catch；

Ö **要**在捕获并重新抛出异常时使用空的throw语句，这是保持调用栈的最好方法

**12.3. 标准异常类的使用：**

12.3.1. Exception与SystemException

´ **不要**抛出这两种类型的异常；

´ **避免**捕获这两种异常，除非是在顶层的异常处理器中；

12.3.2. InvalidOperationException

Ö 对象处于不正确状态时抛出；

12.3.3. ArgumentException，ArgumentNullException，ArgumentOutOfRangeException

Ö 如果传入的是无效参数，**要**抛出参数异常，尽可能使用位于继承层次末尾的类型；

Ö **要**在抛出异常时设置ParaName属性；

12.3.4. NullRefernceException，IndexOutOfRangeException，AccessViolationException

´ **不要**显示抛出或捕获；

12.3.5. StackOverflowException：

´ **不要**显示抛出或捕获；

12.3.6. OutOfMemoryException：

´ **不要**显示抛出或捕获；

**12.4. 自定义异常类型设计规则：**

´ **避免**太深的继承层次；

Ö **要**从已有的异常基类继承；

Ö 异常类**要**以“Exception”做为后缀；

Ö **要**使异常可序列化，使其能跨应用程序域和远程边界仍能正常使用；

Ö **要**把与安全性有关的信息保存在私有的异常状态中

**12.5. 异常与性能**

Ö 如果在普通场景都会抛出异常，**要**采用先效验合法性的方式来避免抛出异常引起的性能 问题；

13. 其他规定

Ö 为避免频繁改动代码，代码中只写比较简单的和不会经常发生变化的SQL，如果SQL 经常发生变化或是比较复杂，存到SysMisc中，比如统计用到的SQL；

Ö 在VS2005开发环境中，采用代码分析工具来做自动化的代码分析，以保证代码质量， 具体的使用建议如下：

A）启用代码分析，并设置当风格不符合要求时为错误而不是警告；

B）如果不是做代码审核，此开关应关闭。加上了这个选项的时候编译很慢；

C）详设的时候打开开关，检查详设是否符合编程规范；

D）所有的选项都应当打开。以下内容需要单独设置：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 编码 | 名称 | 大类 | 建议 | 使用等级 |
| CA2209 | 程序集应声明最小安全性 | 用法规则 | 不建议使用 | 警告 |
| CA1814 | 与多维数组相比，首选使用交错的数组 | 性能规则 | 使用，但降低等级 | 警告 |
| CA1822 | 将成员标记为 static | 性能规则 | 较繁锁，且影响代码质量 | 禁用 |
| CA2210 | 程序集应具有有效的强名称 | 设计规则 | 影响Xcopy部署 | 禁用 |
| CA1302 | 不要对区域设置特定的字符串进行硬编码 | 全球化规则 | 很繁琐，并且工具支持的不好。全球化规则全部禁用 | 禁用 |
| CA2100 | 检查 Sql 查询中是否有安全漏洞 | 安全性规则 | 都采用参数化查询，有可能会参数过长；如果是内部参数，也不会有安全问题 | 警告 |

**C#的实验封装方法代码及Excel文件导入代码**

//封装方法：弧度转角度，角度转弧度  
        public double dmstorad(string s)  
        {  
            string[] ss = s.Split(new char[3] { &apos;°&apos;, &apos;′&apos;, &apos;″&apos; }, StringSplitOptions.RemoveEmptyEntries);  
            double[] d = new double[ss.Length];  
            for (int i = 0; i < d.Length; i++)  
                d[i] = Convert.ToDouble(ss[i]);  
            double sign = d[0] >= 0.0 ? 1.0 : -1.0;  
            double rad = 0;  
            if (d.Length == 1)  
                rad = Math.Abs(d[0]) \* Math.PI / 180;  
            else if (d.Length == 2)  
                rad = (Math.Abs(d[0]) + d[1] / 60) \* Math.PI / 180;  
            else  
                rad = (Math.Abs(d[0]) + d[1] / 60 + d[2] / 60 / 60) \* Math.PI / 180;  
            rad = sign \* rad;  
            return rad;  
        }  
        public string radtodms(double rad)  
        {  
            double sign = rad >= 0.0 ? 1.0 : -1.0;  
            rad = Math.Abs(rad) \* 180 / Math.PI;  
            double[] d = new double[3];  
            d[0] = (int)rad;  
            d[1] = (int)((rad - d[0]) \* 60);  
            d[2] = (rad - d[0] - d[1] / 60) \* 60 \* 60;  
            d[2] = Math.Round(d[2], 2);  
            if (d[2] == 60)  
            {  
                d[1] += 1;  
                d[2] -= 60;  
                if (d[1] == 60)  
                {  
                    d[0] += 1;  
                    d[1] -= 60;  
                }  
            }  
            d[0] = sign \* d[0];  
            string s = Convert.ToString(d[0]) + "°" + Convert.ToString(d[1]) + "′" + Convert.ToString(d[2]) + "″";  
            return s;  
        }  
        //封装方法：坐标方位角推算  
        public double fangweijiao(double[] sdr, double[] cr)  
        {  
            double sum = 0;  
            for (int i = 1; i < sdr.Length; i++)  
            {  
                cr[i] = cr[i - 1] + sdr[i] - Math.PI;  
                if (cr[i] >= Math.PI \* 2)  
                    cr[i] -= Math.PI \* 2;  
                else if (cr[i] < 0.0)  
                    cr[i] += Math.PI \* 2;  
                sum += sdr[i];  
            }  
            return sum;  
        }  
  
//EXCE文件导入  
{  
         Excel.Application ex = new Excel.Application(); //声明一个Excel.Application对象 ex ex.Visible = true;   //使ex可见   
            ex.Application.Workbooks.Add(true); //在ex中增加一个工作簿  
        for (int i = 0; i < dataGridView1.ColumnCount; i++) //把dataGridView1中的列名存入表格中   
{   
        ex.Cells[1, i + 1] = dataGridView1.Columns[i].HeaderText;  
}  
       for (int i = 0; i < dataGridView1.RowCount; i++) //把dataGridView1中的数据存入表格中   
{   
       for (int j = 0; j < dataGridView1.ColumnCount; j++) ex.Cells[i + 2, j + 1] = dataGridView1.Rows[i].Cells[j].Value;  
}  
        MessageBox.Show("数据输入已完成！");  
  
  
        }