**软件编码规范**

（仅限内部开发人员）

北斗星测绘地理信息研发中心

**文档修改记录**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **版本** | **日期** | **修改理由** | **作者** | **备注** |
| 0.1 | 2017.7.1 | 初稿 | 庞立阳 |  |
| 1.0 | 2017.7.14 | 调整 | 范冬林 |  |

## 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，提高团队代码一致性，使代码易于理解、维护和增强，提高生产效率。

## 2. 适用范围

本规范适用于中心所有的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. 约定

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“”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：

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：

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

要使一个Tab为4个空格长。

### 4.2. 缩进

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

建议适当的增加空行，来增加代码的可读性。

在类，接口以及彼此之间要有两行空行：

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. “{”，“}”

开括号“{”要放在块的所有者的下一行，单起一行；

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。

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

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

二元操作符和操作数之间要用空格隔开。如：i + c;

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

注释应该增加代码的清晰度；

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。

建议先写**注释**，后写**代码**，注释和代码一起完成

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。

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

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

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

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

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

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；

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

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

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 ”。

要尽量采用项目所涉及领域的术语。

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.NorthStar.Pipeline |
| 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。

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

公司域名（NorthStar）+ 项目名称 + 模块名称（可选），例如：

中心系统程序集：NorthStar.Pipeline；

中心系统业务逻辑程序集：NorthStar. Pipeline.Business；

#### 6.2.2. 命名空间命名

采用和程序集命名相同的方式：中心简称（NorthStar）+ 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：NorthStar.Pipeline；

中心系统下的用户控件：NorthStar.Pipeline.UserControl；

中心系统业务逻辑：NorthStar.Pipeline.Business；

中心系统数据访问：NorthStar.Pipeline.Data；

#### 6.2.3. 类和接口命名

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。

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；

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

第一个单词一般是动词

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;

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

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”或“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；

尽量要使用短而且具有意义的单词；

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

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；

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

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;

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

每行建议只有一条语句。

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;

}

for语句格式如下：

for (initialization; condition; update)

{

statements;

}

如果语句为空：

for (initialization; condition; update) ;

while语句格式如下：

while (condition)

{

statements;

}

如果语句为空:

while (condition);

do-while语句格式如下：

do

{

statements;

}

while (condition);

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;

}

try-catch语句格式如下：

try

{

statements;

}

catch (ExceptionClass e)

{

statements;

}

finally

{

statements;

}

## 9. 类型设计规范

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

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：NorthStar.Pipeline；

业务逻辑层：NorthStar.Pipeline.Business；

数据访问层：NorthStar.Pipeline.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

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；

建议使用抽象类而不是接口来解除协定与实现间的偶合；

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；

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。

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；

不要在静态类中声明或覆盖实例成员；

### 9.5. 枚举设计

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；

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);

}

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

}

}

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

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。

要使用属性，如果该成员表示类型的逻辑attribute

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

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;

如果该操作比访问字段慢一个或多个数量级，要使用方法。

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：

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；

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；

要允许用户以任何顺序来设置属性的值；

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

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；

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；

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；

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；

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；

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);

}

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

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

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。

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：

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；

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

## 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；

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；

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；

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

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；

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

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；

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；

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；

进行清理工作时要用try-finally，避免使用try-catch；

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

## 13. 参考文档

[Krzysztof Cwalina,](https://book.douban.com/search/Krzysztof%20Cwalina%2C)[Brad Abrams](https://book.douban.com/search/Brad%20Abrams)，《.net 设计规范》，人民邮电出版社，2010