C#编码规范

前言

本文是一套面向C# programmer 和C# developer 进行开发所应遵循的开发规范。

按照此规范来开发C#程序可带来以下益处：

• 代码的编写保持一致性，

• 提高代码的可读性和可维护性，

• 在团队开发一个项目的情况下，程序员之间可代码共享

• 易于代码的回顾，

本规范是初版，只适用于一般情况的通用规范，并不能覆盖所有的情况。

产品中所有的代码需遵循统一的标准。

    a. 指定统一编码风格文档。

    b. 重要的在于让每个开发人员都遵守。

    c. 将不符合规范的代码当作错误处理。

第1章 格式说明

1.1 基本命名格式

1.1.1 Pascal

每个单词首字母大写，如BaseClass。

1.1.2 camel

第一个单词首字母小写，其余单词首字母大写，如：baseClass。

1.1.3 Hungarian

类型缩写+Pascal，其中类型缩写一般使用类型中关键辅音字母，如：btnSubmit。

1.1.4 \_camel

下划线 加 camel 格式，如：\_wordDocument。

1.1.5 ACL（ALL\_CAPITAL\_LETTER）

所有字母大写，单词间用下划线连接，如：MESSAGE\_MAX\_SIZE。

1.1.6 其它原则

除字段外，尽量不使用下划线“\_”，如果需要使用，下划线应该表示语意上的分级。

第2章 一般规范

2.1 命名规则

一个好的名字应该要表达它的意义。

2.1.1 变量、参数

λ 名词、名词短语。

λ 格式：camel。

一般情况下应该取有意义的名字。但是计数变量当用在琐碎的计数循环式更适宜叫i, j, k, l, m, n。

2.1.2 常量

λ 名词、名词短语。

λ 格式：ACL。

λ 必须使用限定符。（注：限定符为private,public,internal,protected等。）

2.1.3 字段

λ 名词、名词短语。

λ 格式：\_camel或m\_camel。如：private int \_itemCount; private int m\_itemCount;

这样的格式，可以在方法中把字段和参数、变量等区分开。

λ GUI控件字段的格式：Hungarian。

GUI控件字段不同于一般的字段，这些控件字段中控件类型是重要的信息，所以这里使用Hungarian格式。

如：

frm Form

btn Buton

cb  下拉列表框

txt 文本输入框

lbl 标签

pic 图像控件

grd  DataGrid

sb  滚动条

lst 列表框

cb  CheckBox

rb  RadioButton

pb  进度条

tv  TreeView

lv  ListView

λ 除了常量及静态只读字段外，其它字段只能使用private、internal限定符。

λ 必须使用限定符。

2.1.4 属性

λ 名词、名词短语。

λ 格式：Pascal。

λ 必须使用限定符。

一般情况下，应该取有意义的名字。偶尔可以考虑使用与其类型相同的名字命名一个属性。如：

public TreeView TreeView

{

get

{

...

}

set

{

...

}

}

λ 除非是十分简单的类或者纯数据类型（贫血数据模型），否则不要使用属性的缩写格式（public TreeView TreeView{get;set;}）。

这样可以在复杂类中只有字段才能表示数据，所以可以很清楚的看到当前有哪些数据构成该类。

2.1.5 方法（函数）

λ 动词、动词短语。

λ 格式：Pascal。

λ 必须使用限定符。

常用的介词有：Of、In、By、DependOn等。如：

User GetByUserName(string username);

2.1.6 事件

λ 考虑用一个动词命名事件。用现在和过去时态命名有前缀和复制概念的事件名字。

λ 格式：Pascal。

λ 必须使用限定符。

λ 多使用EventHandler<T>进行定义。

如：public event EventHandler<MouseMoveEventArgs> MouseMove;

2.1.7 类型

一般情况下，类型都使用Pascal格式来进行命名。

必须使用限定符。

2.1.7.1 接口

λ 使用I字母打头，然后接Pascal形式，即格式为：IPascal。如ICollection。

2.1.7.2 枚举

λ 格式：Pascal。

λ 对于位枚举(Flags)用复数名字。

2.1.7.3 异常类

λ 格式：Pascal + Exception

自定义异常类以Exception结尾，并且在类名中能清楚的描述出该异常的原因。比如FileNotFoundException，描述出了某个实体（文件、内存区域等）无法被找到。

2.1.7.4 事件委托

λ 格式：Pascal + EventHandler

如：比如public delegate void MouseEventHandler (object sender, MouseEventArgs e);    //用于处理与鼠标相关的事件或委托。

对于自定义的委托，其参数第一个建议仍然使用object sender，sender代表触发这个时间或委托的源对象。而第二个参数继承于EventArgs类，并且在派生类中实现自己的业务逻辑。

2.2 代码格式及逻辑

2.2.1 声明

2.2.1.1 类成员声明

λ 所有类成员都应该明确写上限定符：private，protected，public，internal。

2.2.1.2 每行的声明数

λ 每行只有一个声明，因为它可以方便注释。

int level; // indentation level

int size; // size of table

λ 当声明变量时，不要把多个变量或不同类型的变量放在同一行，例如：

int a, b; //What is 'a'? What does 'b' stand for?

2.2.1.3 初始化

λ 局部变量一旦被声明就要初始化。例如：

string name = myObject.Name;

或

int val = time.Hours;

2.2.1.4 方法声明

λ 如果可以的话，方法的参数尽量不要使用out及ref。

2.2.2 语句

λ 大括号的格式：大括号内表示一个语句块(block)，前大括号和后大括号可以只占一行。

λ 每行都应该只包含一条语句。

2.2.2.1 If语句

λ 可以使用单行if。

λ 其余情况应该使用带括号的if语句。

如，使用：

If(argument <= 0) throw new ArgumentException(“argument必须是正整数！”);

或

If(value)

{

//do something

}

而不是

If(value)

This.Calculate();

λ 推荐不要把逻辑上不是一类的判断组合在一起。

λ 条件尽量简单，不宜过长。

λ 如果条件过长，可采用两种方式：

1.可以先把条件的值计算出，并存入一个临时的变量。

2.可对条件进行折行，折行的方式见：2.4.5.2换行。

2.2.2.2 For语句

λ 带上两个括号，即采用以下格式：

for (int i = 0; i < array.Length; i++)

{

//do sth

}

λ 当判定条件比较耗时时，应该采用缓存变量的形式，即以下格式：

for( int i = 0, c = list.Count; i < c; i++)

{

//do sth

}

2.2.2.3 While语句

采用以下格式：

while(value)

{

}

或

do

{

}

while(value)

2.2.2.4 Switch语句

switch应该处理所有的可能分支。如果不处理时，应该抛出异常。如：

switch(value)//value 是string类型

{

case “1”:

//case 1;

break;

case “2”:

//case 2;

break;

default:

   throw new NotSupportedException();

}

2.2.2.5 Try语句

格式如下：

try

{

}

catch(NotSupportedException ex)//子异常

{

}

catch(Exception e)//基异常

{

}

finally//finally语句可以省略

{

}

2.2.2.6 小技巧

以上所有语句，在VS环境下，都已经默认携带了snippet方便快捷输入。如在书写if语句时，可输入“if”，然后按两下tab键即可。

另外，它的用处在于它可以进行自定义你经常输入的代码段。具体内容见：http://www.cnblogs.com/WilsonWu/archive/2010/02/25/1673745.html

2.2.3 属性

λ 不要连续地对同一属性进行调用。

如：

var prop2 = this.Prop1.Prop2;

var prop3 = this.Prop1.Prop3;

而是应该用变量进行缓存，如：

var prop1 = this.Prop1;

var prop2 = prop1.Prop2;

var prop3 = prop1.Prop3;

这是因为属性其实就是方法。所以就算是最简单的属性，对它进行多次调用都会浪费一定的资源。

λ 属性的内部代码，不要设计得太过复杂。方便调试。

λ 最好不要访问易中断的外部资源。因为很可能会导致无法对其调试。

2.2.4 事件

监听的事件处理方法取名按照“一定规则”，如btnSubmit\_Click。监听的事件处理方法里面，不应该写比较复杂的代码。如果比较复杂，应该提取函数。

2.2.5 方法

λ 原则上单个方法的长度不能太长，如不超过50行、不超过一屏。

λ 程序编码力求简洁，结构清晰，避免太多的分支结构及太过于技巧性的程序。

λ 程序代码要严格，要把所有情况都考虑到。抛出的异常，应该都是在意料范围内的。如，以下代码是不严格的：

int[] array = this.GetIntArray();//这个方法可能会返回一个空数组。

int first = array[0];//由于数组可能没有元素，所以这里可能会抛出异常。

好的代码应该是：

if(array.Length > 0)

{

    int first = array[0];

}

或断言：

if(array.Length == 0) throw new InvalidException(“返回的数组没有元素。”)

λ 如果是未完成的方法，必须抛出NotImplementException。

λ 参数的检测，使用正常的代码进行检测。而调试期代码则使用Debug.Assert方法。

λ 方法逻辑：

按照以下顺序编写：

1.参数检查。（private方法在参数传入时已经肯定参数正确时，可以忽略此步。）

2.参数转换。（private方法在参数传入时已经肯定参数正确时，可以忽略此步。）

3.核心逻辑。

4.返回值转换。

5.返回。

2.3 注释

λ 注释尽量使用中文。

λ 添加注释的目的：使代码易读、易写、易维护 。

λ 如何添加注释：

    a. 代码、数据、算法的解释

    b. 做标记（时间、所做的改动等）

    c. 标识代码的功能和目的

    d. 代码如何调用

λ 避免

    a. 对显而易见的内容进行注释

    b. 添加大段注释

    c. 注释的拼写错误

2.3.1 文件头

为了清晰的说明代码功能，跟踪版本纪录，应该在源代码文件中加入相应的描述信息，本文将列举相应的文件头格式。

需要说明的是，文件头中的历史纪录位置应该详细记录每次所作修改的内容。每次修改版本号都应该作相应的变更。

如对版本号无特殊要求，请遵照如下通用原则：

λ 版本号形如 X.Y，其中X为大版本号，Y为小版本号。

λ bug修正，变更小版本号。

λ 新增功能或重大bug修正，变更大版本号。

2.3.1.1 C#代码文件头

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

 \*

 \* 作者：胡庆访

 \* 创建日期：20150801

 \* 说明：此文件只包含一个类，具体内容见类型注释。

 \* 运行环境：.NET 4.0

 \* 版本号：1.0.0

 \*

 \* 历史记录：

 \* 创建文件 胡庆访 20150801 17:13

 \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

2.3.1.2 小技巧

可以在 Visual Studio 中搜索 HxyUtils 插件扩展，如下图：

安装后，点击工具->添加文件头，即可为当前打开的代码文件加入可配置的文件头。如图：

生成：

2.3.2 类和函数

λ 每个类及函数必须写上注释。

λ 格式：利用VS自带的注释块生成格式。在需要的地方键入"///"。

λ 类型注释、文件头、复杂函数、复杂属性、接口全部内容必须写清楚注释。

2.3.3 块注释

块注释通常应该是被避免的。推荐使用//注释作为C#的标准声明。如果希望用块注释时你应该用以下风格：

因为样可以为读者将注释块与代码块区分开。虽然并不提倡使用C风格的单行注释，但你仍然可以使用。一旦用这种方式，那么在注释行后应有断行，因为很难看清在同一行中前面有注释的代码：

/\* blah blah blah \*/

λ 块注释在逻辑比较复杂时进行注释是非常有用的。通常块注释用于注释掉大的代码段。

2.3.4 单行注释

你应该用//注释风格“注释掉”代码（快捷键，Ctrl+K,Ctrl+C）。它也可以被用于代码的注释部分。

单行注释被用于代码说明时必须缩进到相应的编进层级。

（一条经验，注释的长度不应该超过被解释代码的长度太长，因为这表示代码过于复杂，有潜在的bug。）

2.4 代码组织

2.4.1 命名空间

命名空间的格式也是Pascal命名法。

逻辑上可分为子空间的，尽量分子空间。

2.4.2 目录结构

为每一个命名空间创建一个目录。（用MyProject/TestSuite/TestTier作为MyProject.TestSuite.TestTier的路径，而不用带点的命名空间名做路径）这样可以更容易地将命名空间映射到目录层次划分。

如果一个命名空间下的文件过多，也可以添加子目录进行管理。这时不需要和空间名保持一致。

2.4.3 C# 源文件

类名或文件名要尽量简短，将代码分割开，使结构清晰。将每个类尽量放在一个单独的文件中，使用类名来命名文件名（当然扩展名是.cs）。这种约定会使大家工作更简单。

如果是很简单的一些同类型的类，也可以把多个类写在一个文件当中。这时，文件的名字使用最主要的类的类名，或者是为它们起一个类型名。例如，多个类可以放一个文件的情况：枚举、常量、命令、Attribute。

如果某个类是分部类，分别写在多个文件中。则需要其中一个文件是主文件，其它文件扩展名分级别。推荐让它们文件依赖于主文件。（这里的依赖是指project中的文件依赖关系，如1.aspx.cs依赖于1.aspx文件，这样vs就会把它们折叠起来。）如：

（

推荐使用文件依赖：

对应csproject代码：

    <Compile Include="Control.cs" />

    <Compile Include="Control.MouseEvent.cs" >

      <DependentUpon>Control.cs</DependentUpon>

    </Compile>

）

2.4.4 类成员排序

λ 成员按以下次序排序：

常量 static readonly 或 const

静态私有字段 private static int \_count

静态构造函数 static ClassName(){}

公共静态属性 public static int Count{get;set;}

公共静态方法 public static void AddCount(){}

私有字段 private int \_count

构造函数 public ClassName(){}

公共属性 public int Count{get;set;}

公共方法 public void AddCount(){}

事件 public event EventHandler CountChanged;

保护方法 protect void OnCountChanged(){}

私有方法 private void AddCount();

保护静态方法 protected static void HelperMethod(){}

私有静态方法 private static void HelperMethod(){}

内部类 private class InnerClass{}。

λ 不同类型的成员，使用区域或空格进行分隔。

λ 同种类型的成员，先按该成员的重要程度进行排序，同时兼顾它们之间的关系。

λ 排序原理：

常量命名从类内部的第1行开始。

紧接着是字段，字段表示数据，能够一目了然地表现类的结构。

然后是：构造函数、属性、事件、方法。

保护的静态方法和私有静态方法一般被本类的实体方法当作公共方法调用，所以放在最后。

修饰符按可见度进行排序，即：public internal protected private。

2.4.5 空白与分隔

2.4.5.1 空白

2.4.5.1.1 缩进

缩进不使用制表符，而是使用空白符（这是因为制表符在不同的编辑器下，显示的效果不一样，所以都使用空白）。每次缩进的空白字符数是4。

2.4.5.1.2 内部空格

在一个逗号或一个分号之后应该由一个空格，例如：

TestMethod(a, b, c);

不要用：

TestMethod(a,b,c)

或

TestMethod( a, b, c );

单个空格包围操作符（除了像加的一元操作符和逻辑非），例：

a = b; // don't use a=b;

for (int i = 0; i < 10; ++i) // don't use for (int i=0; i<10; ++i)

// or

// for(int i=0;i<10;++i)

2.4.5.1.3 小技巧

可以使用VS IDE自带的格式化功能，（快捷键：Ctrl+K、Ctrl+D）即可自动自成缩进、空格。

2.4.5.2 换行

λ 逻辑换行：

空行提高可读性。它们分开那些逻辑上自身相关联的代码块。所以，每一个空行，都应该表示逻辑上的分隔关系，

类的成员之间，都必须空行。

λ 过长换行：

当一个表达式超过一行时，根据这些通用原则进行处理：

1. 在逗号后换行。

2. 在操作符后换行。

3. 在高层换行而不要在低层处换行。

4. 一次或多次折行后，都对应此行语句最上层的表达式起始位置，进行一次缩进。

方法调用换行示例：

longMethodCall(expr1, expr2,

    expr3, expr4, expr5);

算术表达式换行示例：

推荐：

var x = a \* b / (c - g + f) +

      4 \* z;

不好的格式——应避免：

var x = a \* b / (c - g +

      f) + 4 \* z;

推荐使用第一种方法，因为是在括号表达式之外折行（高层次折行原则）。注意要用制表符到缩进的位置，然后用用空格到折行的位置。在我们的例子中是：

> var x = a \* b / (c - g + f) +

> ......4 \* z;

'>'表示是制表符，'.'表示是空格符。（制表符后是空白是用制表符缩进）。一个好的编码习惯就是在所用的编辑器中显示制表符和空格符。

2.4.5.3 Region

λ 逻辑上的独立块，都推荐使用Region进行分隔。

λ 方法内不要使用Region，如果太长，就提取方法。

λ 空行可进行逻辑块的分隔。但是当逻辑块较大时，应该使用Region。

λ 格式：#region 和 #endregion的前后都需要有一行空行。如果添加的空行是在block的开始和结尾，则可以省略。

如：

public class Class1

{

#region 私有字段

字段定义

#end region

#region 公共属性

属性的定义

#endregion

}

2.4.5.4 空代码块

大括号内的代码块为空时，应把两个大括号都和前一行代码放在一行。如：

protected virtual void OnMove(EventArgs e) { }

try

{

    //…

}

catch{ }//放在一行。

第3章 推荐基类编写规范

此规范较为严格，仅为推荐。适用于编写较好的框架基类，特殊情况下可不遵守部分规范。

3.1 代码格式及逻辑

3.1.1 事件

监听某个事件后，要记住释放监听。

3.1.1.1 一般编写模式（推荐使用）

λ 为每个事件都写一个保护的虚方法。虚方法名为On + 事件名，并只带一个对应事件的事件参数。

λ 在虚方法中触发外部监听的事件。

如：

public event EventHandler<MouseMoveEventArgs> MouseMove;

protected virtual void OnMouseMove(MouseMoveEventArgs e)

{

var handler = this.MouseMove;

if(handler != null) handler(this, e);

}

λ 子类在处理事件时，直接重写该虚方法。

如：

protected override void OnMouseMove(MouseMoveEventArgs e)

{

base.OnMouseMove(e);

//do sth.

}

3.1.2 异常

3.1.2.1 描述

异常信息应该有如下的描述

存在的问题。

违反的规律。

可能的解决方案。

如：Nod无法打开文件nod.config：请确认该文件存在并且当前用户拥有足够的权限

如：OPath解析查询语句错误：[和]必须匹配，请检查[]的完整性

3.1.2.2 处理

发生异常时只能向外抛出。

第4章 DBI应用编写规范

4.1 代码组织

4.1.1 命名空间

命名空间使用以下格式：

实体类：DBI.(可选子模块缩写).(可选命名空间)；

服务：DBI.Services.(可选子模块缩写)；

命令：DBI.WPF. (可选子模块缩写).Commands；

界面：DBI.WPF. (可选子模块缩写)。

例如，在用户模块中的实体类都声明在 DBI.Accounts 空间下，命令则声明在 DBI.WPF.Accounts.Commands 空间下，界面其它类型则声明 DBI.WPF.Accounts 中。

4.1.2 类型

命令类型：由于所有命令都已经放在 Commands 命名空间下，所以这些类型都不需要以 Command 结尾。

第5章 代码管理

5.1 版本控制

采用版本控制工具svn进行源代码管理。（未来将采用 Git 进行管理）

λ 每天打开解决方案后，先获取最新版本。

λ 每次提交都应该保证没有编译错误，同时在说明栏写上本次完成的内容。

λ 提交流程：更新最新版本；编译无错；提交。

5.2 版本号

给客户发布的所有版本需要使用版本号进行管理。

版本号由四个部分组成：主版本号、次版本号、内部版本号和修订号。形式如：

主版本号.次版本号.编译版本号.修正版本号

所有定义的部分都必须是大于或等于 0 的整数。

主版本号：具有相同名称但不同主版本号的程序集不可互换。例如，这适用于对产品的大量重写，这些重写使得无法实现向后兼容性。

次版本号：如果两个程序集的名称和主版本号相同，而次版本号不同，这指示显著增强，但照顾到了向后兼容性。例如，这适用于产品的修正版或完全向后兼容的新版本。

内部版本号：内部版本号的不同表示对相同源所作的重新编译。这适合于更改处理器、平台或编译器的情况。

修正版本号：名称、主版本号和次版本号都相同但修订号不同的程序集应是完全可互换的。这适用于修复以前发布的程序集中的安全漏洞。

程序集的只有内部版本号或修订号不同的后续版本被认为是先前版本的修补程序 (Hotfix) 更新。

5.3 Build策略

每日至少Build一次。

5.4 Code Review

暂无。

实例代码：

doublex2, y2, x3, y3; //存放已知两个点的x，y坐标

x2 = Convert.ToDouble(dataGridView1.Rows[1].Cells[12].Value);

y2 = Convert.ToDouble(dataGridView1.Rows[1].Cells[13].Value);

x3 = Convert.ToDouble(dataGridView1.Rows[sd.Length -1].Cells[12].Value);

y3 = Convert.ToDouble(dataGridView1.Rows[sd.Length -1].Cells[13].Value);

double[] sl = newdouble[sd.Length -1];//存放观测距离

double[] dx = newdouble[sl.Length];//存放坐标增量

double[] dy = newdouble[sl.Length];

doublesuml = 0, sumdx = 0, sumdy = 0;

for(inti=1;i<sl.Length;i++)

{

sl[i] = Convert.ToDouble(dataGridView1.Rows[i].Cells[5].Value);

//将观测距离放到sl数组中

suml += sl[i]; //计算距离总和

dx[i] = sl[i] \* Math.Cos (cr[i]); //利用距离和坐标方位角计算坐标增量

dy[i] = sl[i] \* Math.Sin (cr[i]);

sumdx += dx[i]; //计算坐标增量总和

sumdy += dy[i];

}

doublefx, fy, fxy, k1;

fx = sumdx -(x3 -x2); //计算坐标增量闭合差

fy = sumdy -(y3 -y2);

fxy = Math.Sqrt(fx \* fx + fy \* fy); //计算导线全长闭合差

k1 = suml / fxy; //计算导线全长相对闭合差分母

double[] vx = newdouble[sl.Length]; //定义数组用于存放坐标增量的改正数及总和

double[] vy = newdouble[sl.Length];

doublesumvx = 0, sumvy = 0;

double[] cx = newdouble[sl.Length]; //定义数组用于存放改正后的坐标增量及总和

double[] cy = newdouble[sl.Length];

doublesumcx = 0, sumcy = 0;

double[] x = newdouble[sl.Length+1]; //定义数组用于存放x，y坐标

double[] y = newdouble[sl.Length+1];

x[1] = x2;

y[1] = y2;

if(k1 < 2000) //判断导线全长相对闭合差是否超限

MessageBox.Show("导线全长相对闭合差超限！");

else

{

for(inti=1; i <vx.Length; i ++)

{

vx[i] = -fx \* sl[i] / suml; //计算坐标增量改正数

vy[i] = -fy \* sl[i] / suml;

sumvx += vx[i]; //计算坐标增量改正数总和

sumvy += vy[i];

}

if(Math.Round(sumvx, 4) != Math.Round(-fx, 4)

|| Math.Round(sumvy, 4) != Math.Round(-fy, 4))

MessageBox.Show("坐标增量分配有误！");

else

{

for(inti = 1; i < vx.Length; i ++)

{

cx[i] = dx[i] + vx[i]; //计算改正后坐标增量

cy[i] = dy[i] + vy[i];

sumcx += cx[i]; //计算改正后坐标增量总和

sumcy += cy[i];

}

if(Math.Round(sumcx, 4) != Math.Round(x3 -x2, 4)

|| Math.Round(sumcy, 4) != Math.Round(y3 -y2, 4))

MessageBox.Show(“改正后的坐标增量计算有误！");

else

{

for(inti = 2; i < x.Length; i++)

{

x[i] = x[i-1] + cx[i-1]; //计算x,y坐标

y[i] = y[i-1] + cy[i-1];

}

if(Math.Round(x[x.Length -1], 4) != Math.Round(x3, 4)

|| Math.Round(y[y.Length -1], 4) != Math.Round(y3, 4))

MessageBox.Show("坐标计算有误！");

else

{

for(inti=1;i<sl.Length;i++)

{

dataGridView1.Rows[i].Cells[6].Value = Convert.ToString(Math.Round(dx[i],4));

//将坐标增量放入表格

dataGridView1.Rows[i].Cells[7].Value = Convert.ToString(Math.Round(dy[i],4));

dataGridView1.Rows[i].Cells[8].Value = Convert.ToString(Math.Round(vx[i],4));

//将坐标增量改正数放入表格

dataGridView1.Rows[i].Cells[9].Value = Convert.ToString(Math.Round(vy[i],4));

dataGridView1.Rows[i].Cells[10].Value = Convert.ToString(Math.Round(cx[i], 4));

dataGridView1.Rows[i].Cells[11].Value = Convert.ToString(Math.Round(cy[i], 4));

//将改正后坐标增量放入表格

dataGridView1.Rows[i].Cells[12].Value = Convert.ToString(Math.Round(x[i], 3));

dataGridView1.Rows[i].Cells[13].Value = Convert.ToString(Math.Round(y[i], 3));

//将x,y坐标放入表格

}

dataGridView1.Rows[dataGridView1.RowCount -4].Cells[5].Value =

Convert.ToString(Math.Round(suml,4));

dataGridView1.Rows[dataGridView1.RowCount -4].Cells[6].Value =

Convert.ToString(Math.Round(sumdx,4));

dataGridView1.Rows[dataGridView1.RowCount -4].Cells[7].Value =

Convert.ToString(Math.Round(sumdy,4));

//将距离总和、坐标增量总和放入表格中

dataGridView1.Rows[dataGridView1.RowCount -4].Cells[8].Value =

Convert.ToString(Math.Round(sumvx,4));

dataGridView1.Rows[dataGridView1.RowCount -4].Cells[9].Value =

Convert.ToString(Math.Round(sumvy,4));

//将坐标增量改正数总和、改正后坐标增量总和放入表格中

dataGridView1.Rows[dataGridView1.RowCount -4].Cells[10].Value =

Convert.ToString(Math.Round(sumcx,4));

dataGridView1.Rows[dataGridView1.RowCount -4].Cells[11].Value =

Convert.ToString(Math.Round(sumcy,4));

dataGridView1.Rows[dataGridView1.RowCount -3].Cells[7].Value =

Convert.ToString(Math.Round(fx,4));

dataGridView1.Rows[dataGridView1.RowCount -2].Cells[7].Value =

Convert.ToString(Math.Round(fy,4));

dataGridView1.Rows[dataGridView1.RowCount -3].Cells[10].Value =

Convert.ToString(Math.Round(fxy,4));

dataGridView1.Rows[dataGridView1.RowCount -2].Cells[11].Value =

Convert.ToString((int)k1); //导线全长相对闭合差分母取整

}

}

}

}