**C#编码规范**

**内容：**

**1．** **关于 C#编码风格指南…………………………2**

**2． 文件组织………………………………………2**

**2.1 C#源文件**

**2.2 目录规划**

**3． 缩进……………………………………………2**

**3.1换行**

**3.2 空白符**

**4． 注释……………………………………………3**

**4.1 块注释**

**4.2** **单行注释**

**4.3** **文档注释**

**5． 声明……………………………………………4**

**5.1 每一行声明的数量**

**5.2 初始化**

**5.3 类、借口和命名空间的声明**

**5.4 方法的声明**

**5.5 属性、索引器和事件的声明**

**6． 语句……………………………………………6**

**6.1 简单语句**

**6.2 返回语句**

**6.3 If, if-else, if else-if else 语句**

**6.4 For / Foreach 语句**

**6.5 While/do-while 语句**

**6.6 Switch 语句**

**6.7 Try-catch 语句**

**7． 空白符…………………………………………8**

**7.1 空白行**

**7.2 相邻项间的空白符**

**7.3 表格式格式化**

**8． 命名规范………………………………………9**

**8.1 大写风格**

**8.2 命名指南**

**9． 编程实践………………………………………11**

**9.1 可见性**

**10．代码举例………………………………………11**

**10.1 括号放置位置举例**

**10.2 量命名举例**

**11.C#代码举例……………………………………13**

1. **关于 C#编码风格指南**

本文档可以作为编写健壮和可靠程序的指南来阅读。本文档主要关注于 C#编程，但其中很多规则和原则对于其他编程语言也是很有用的。

1. **文件组织**

**2.1 C#源文件**

保持你的类和文件简短，不要超过 2000 行，拆分你的代码，使结构更清晰。把每一个类放到单独的文件里，并以类名来命名文件（当然是用.cs 作为扩展名）。这个约定会使我们的编程变得更简单。

**2.2 目录规划**

为每一个命名空间创建目录。

1. **缩进**

**3.1 换行**

如果一个表达式不合适放在一行中的时候，可以根据一下原则把它进行断行处理：

* + 在逗号后断行
  + 在操作符后断行
  + 在低优先级别处断行，而保留高优先级完整性
  + 把新一行的表达式和前面的表达式在开始的地方要对齐排列方法调用断行的例子：

longMethodCall(expr1, expr2,

expr3, expr4, expr5);

算术表达式的断行举例：

好的断行风格：

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

4 \* z;

不好的断行风格– 要避免：

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

f) + 4 \* z;

第一种断行方式更好，因为它断行处在括号表达式（高优先级）的外边。注意，你先用制表符缩进到上一行的缩进层级，再用空格填充到断行位置，我们上边的例子应该这样：

* var = a \* b / (c - g + f) +
* ......4 \* z;

这里，“>”表示 tab 字符，“.”表示空格（tab 后面的空格，也是用 tab 键敲出的空格缩进）。一个好的编码练习，是在编辑器里让 tab 和空格都能看得到。

**3.2** **空白符**

一个缩进标准是用空格无法达到的。有人喜欢用两个空格，有人喜欢用四个空格，而其他人更喜欢用八个空格，设置更多的空格。最好使用制表符。制表符有以下有点：

* 每个人都可以设定他们喜欢的缩进层级
* 它只是一个字符，而不是二、四、八…因此可以减少敲键盘的次数（即使使用智能缩进，有时你也需要手动设定缩进，或者删除智能缩进或者其他的问题）
* 如果你想增加缩进（或者减少），标记光标位置，然后可以用 tab 增加缩进，用 shift+tab 减少缩进。这对绝大多数文本编辑器都适用。

这里，我们定义制表符作为标准缩进字符。

* 不要用空格做缩进，用制表符做缩进！
* **4 注释**

**4.1 块注释**

应经常避免使用块注释，因为使用///注释作为 C#标准注释是被推荐的。如果你要使用块注释，应该用下面的样式：

/\* Line 1

* 1. Line 2
  2. Line 3

\*/

因为这种样式使（人）读者很容区分块注释和编码。你也可以使用旧的 C 语言的样式风格来做单行注释，尽管这并被推荐。一旦你使用了这种样式，注释后便就应该断行，因为如果放在一行，很难分清楚注释和后边的代码。

/\* blah blah blah \*/

块注释也许在某些案例中有用，请参考技术笔记“注释的艺术”作为例子。一般，块注释在注释掉大块代码的时候有用。

**4.2** **单行注释**

你应该使用//注释样式来注释掉代码（SharpDevelop 有快捷键 Alt+/）。它也可以用来注释掉代码片段。

单行注释在代码文件中使用时，一定要缩进到缩进位上。

应该在第一行注释掉代码，这样能够使注释掉的代码和为注释的代码很容易区分。

一条经验法则：通常，注释的长度不应该超过代码，如果解释太多，就说明这是一段复杂的、有潜在 bug 的代码。

**4.3** **文档注释**

在.net framework 中，微软提出了一种基于 xml 注释的文档生成系统。这些注释形式上是单行带有 xml 标签的 C#注释。单行注释样式如下：

* <summary>
* This class...
* </summary>

多行 xml 注释样式如下：

* <exception cref=”BogusException”>
* This exception gets thrown as soon as a
* Bogus flag gets set.
* </exception>

所有的行都必须以三个/做前缀，这样可以作为 xml 注释行被接受。

标签分为两类：

* 文档条目
* 格式/参考

第一个分类包含诸如<summary>, <param> 或者 <exception>样的标签。这些条目

描述程序 API 的元素，它们必须被写到程序的文档中，这样对其他编程者会很有用。这些标签通常都有属性，例如 name 或者 cref，就像上面多行例子中描述的。这些属性会被编译器检查，所以它们都是有效的。后面的那个分类管理文档的规划，使用标记

像<code>, <list> 或者 <para>。

使用#develop（译者注：SharpDevelop 软件）的 build 菜单中的文档选项可以生成文档。生成的文档是 HTML 帮助的格式。

在微软的.net framework SDK 文档中可以查看 xml 注释的完整解释。在技术笔记“注释的艺术”可以查看关于注释的最佳实践和深入话题。

1. **声明**

**5.1 每一行声明的数量**

推荐一行一个声明，这样有利于注释 1。换句话说，

int level; // 缩进级别 int size; // 表格大小

不要在一行中声明多个变量或者不同类型的变量，例如： int a, b; // 'a'表示什么？ 'b'表示什么?

上面的例子同时也说明了没有语义的变量名字的缺点。当命名变量的时候一定要清楚。

**5.2 初始化**

尽量在局部变量声明的时候就初始化它们，例如： string name = myObject.Name;

或者

int val = time.Hours;

注意：如果你初始化一个对话框，尽量使用 using 语句：

using (OpenFileDialog openFileDialog = new OpenFileDialog())

{

...

}

**5.3** **类、借口和命名空间的声明**

当编写 C#类、接口和命名空间的时候，应该遵循下边的格式规则：

* 开始的大括号“{”出现在声明语句的下一行
* 结束的大括号“}”自己在一行，缩进到和它对应的大括号的位置上

例如：

4

class MySample : MyClass, IMyInterface {

int myInt;

}

namespace MyNamespace

{

// namespace contents

}

关于一个大括号放置位置的例子请参看10.1章节

**5.4** **方法的声明**

关于方法的声明，大括号和类的大括号放置规则一样，并且方法名字和它参数列表前边的圆括弧“(”不应该有空白符。

例如：

public MySample(int myInt)

{

this.myInt = myInt ;

}

void Inc()

{

++myInt;

}

**5.5** **属性、索引器和事件的声明**

对于属性、索引器和事件，你**不**能把开始的大括号自己单独放一行期盼最后的结束大括号“}”。例如：

public int Amount {

get {

...

}

set {

...

}

}

public this[string index] {

get;

set;

} public EventHandler MyEvent {

add {

...

}

remove {

...

}

}

1. **语句**

**6.1 简单语句**

每行只有一条语句。

**6.2 返回语句**

不应该用圆括号把返回语句都包括起来。

不使用：return (n \* (n + 1) / 2);

使用： return n \* (n + 1) / 2;

**6.3** **If, if-else, if else-if else 语句**

if, if-else and if else-if else 语句应该是这样：

if (condition) {

DoSomething();

...

}

if (condition) {

DoSomething();

...

} else { DoSomethingOther();

...

}

if (condition) {

DoSomething();

...

} else if (condition) { DoSomethingOther();

...

} else { DoSomethingOtherAgain();

...

}

**6.4** **For / Foreach 语句**

for 语句应该是下边的格式：

for (int i = 0; i < 5; ++i) {

...

}

或者是单行（考虑使用 while 语句代替）

for (initialization; condition; update) ;

foreach 语句应该像这样：

foreach (int i in IntList) {

...

}

注意：一般使用大括号，即使在循环中只有一条语句。

**6.5** **While/do-while 语句**

while 语句应该像下边这样写：

while (condition) {

...

}

空 while 语句应该是下边这样的格式：

while (condition) ;

do-while 语句是下边这样的格式：

do {

...

} while (condition);

**6.6** **Switch 语句**

switch 语句是下边这样的格式：

switch (condition) {

case A:

...

break;

case B:

...

break;

default:

...

break;

}

**6.7** **Try-catch 语句**

try-catch 语句是这样的格式：

try {

...

} catch (Exception) {}

或者

try {

...

} catch (Exception e) {

...

}

或者

try {

...

} catch (Exception e) {

...

} finally {

...

}

1. **空白符**

**7.1 空白行**

空白行可以提高可读性。他们可以帮助区分逻辑相关联的代码块。两个空白行应该用在下列情形中：

* + 一个源文件的逻辑代码片段之间
  + 类和接口定义之间（尽量将一个类或者接口放到一个文件中，这样可以避免这种情形）

单空白行应该用在下列情形中：

* 方法之间
* 属性之间
* 方法中的局部变量和它的第一个语句之间
* 一个方法中的逻辑片段之间，来提高可读性

注意，你应该总是把空白行也缩进到正确的位置上，而不要留着空白行什么都不做或者缩进到一个错误的位置上。这样以来，在空白行中插入新的语句就变得容易些。

**7.2 相邻项间的空白符**

在逗号或者分号后便应该有一个空白符，例如：

TestMethod(a, b, c); 不要使用 : TestMethod(a,b,c)

或者

TestMethod( a, b, c );

操作符前后要有单个空白符（一元运算符除外，像++或者逻辑反运算），例如：

a = b; // 不要使用 a=b;

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

* 或者
* for(int i=0;i<10;++i)

**7.3** **表格式格式化**

多行的逻辑块代码应该像一个表格一样格式化：

|  |  |  |  |
| --- | --- | --- | --- |
| string name | | = "Mr. Ed"; | |
| int | myValue | = | 5; |
| Test | aTest | = | Test.TestYou; |

在表格样式化的时候要使用空白符，不要使用制表符，因为在特殊的制表符缩进层级上表格格式化会看起来很奇怪。

1. **命名规范**

**8.1 大写风格**

**8.1.1 Pascal 命名规则**

这种方式是每个单词第一个字母大写（例如 **T**est**C**ounter）。

**8.1.2 骆驼式命名规则**

这种方式是除了第一个单词之外的每个单词首字母大写，例如 test**C**ounter。

**8.1.3 大写字母命名规则**

全大写标识符仅用在由一个或者两个字母组成的缩写的情况下，由三个或者更多的字母组成的标识符应该使用 Pascal 命名规则来代替，例如：

public class Math

{

public const PI = ...

public const E = ...

public const FeigenBaumNumber = ...

}

**8.2** **命名指南**

一般，在命名中使用下划线和以匈牙利命名规则命名变量被认为是不好的习惯。

匈牙利命名规则定义了一套前缀和后缀来命名变量，反映出变量的类型。这种命名方式在以前的 windows 编程中被广泛使用，但是现在却不被使用，或者不建议使用。如果你遵循本规范，那么你不能使用匈牙利命名规则。记住，一个好的变量名字是描述的语义而不是类型。

但是 GUI 编码是一个例外。所有包含 GUI 元素的字段和变量名字，像按钮，都应该带有类型名字的后缀，这种后缀不是缩写的。例如：

System.Windows.Forms.Button cancelButton;

System.Windows.Forms.TextBox nameTextBox;

**8.2.1 类命名指南**

* 类的名字必须为名词或者名词短语。
* 使用 Pascal 命名规则，参见 8.1.1
* 不要使用类前缀

**8.2.2 借口命名指南**

* 使用名次、名词短语或者描述行为的形容词命名接口(例如：IComponent or IEnumberable)
* 使用 Pascal 命名规则，参见 8.1.1
* 使用“I”作为名字的前缀，它后面是一个大写的字母（接口名字的第一个字母）

**8.2.3 枚举命名指南**

* 使用 Pascal 命名枚举值名字和类型名字
* 不要为枚举类型或者值的名字添加前缀（或后缀）
* 对枚举使用单数名字
* 对位字段使用复数名字

**8.2.4 只读和常量字段名**

* 对于静态的字段使用名词、名词短语或者名词缩写
* 使用 Pascal 命名规则，参见 8.1.1

**8.2.5 参数和非常量字段名**

* 要使用描述性的名字，名字足以描述变量的语义和类型。但是更倾向于基于参数语义的名字。
* 使用骆驼式命名规则，参见 8.1.2

**8.2.6 变量命名**

* 计数型变量倾向于使用 i, j, k, l, m, n 这几个字母，一般都用于常见的的循环计数中。（参见 10.2 中关于全局计数器等的更加智能的命名的例子。）
* 更喜欢在布尔型变量前加前缀，如 Is, Has 或者 Can。一般你应该给布尔型变量命名能意味真或假的名字（例如：fileFound, done, success

或者带 is 前缀：isFileFound, isDone, isSuccess，但是不要使用

IsName 这样根本没有意义的名字）。

* 使用骆驼式命名规则，参见 8.1.2

**8.2.7 方法命名**

* 用动词或者动词短语命名方法名字
* 使用 Pascal 命名规则

**8.2.8 属性命名**

* 使用名词或者名词短语命名属性的名字
* 使用 Pascal 命名规则
* 考虑使用属性类型的名字来命名属性

**8.2.9 事件命名**

* 使用 EventHandler 作为命名事件句柄的后缀
* 使用两个参数，分别是 sender 和 e
* 使用 Pascal 命名规则
* 命名事件参数类使用 EventArgs 后缀
* 对于有“之前”或“之后”概念的事件，要使用现在时或过去时命名
* 考虑使用动词命名事件

**8.2.10 大写摘要**

|  |  |  |
| --- | --- | --- |
| ***类型*** | ***命名规则*** | ***备注*** |
|  |  |  |
| 类 / 结构 | Pascal |  |
|  |  |  |
| 接口 | Pascal | 以 I 开头 |
| 枚举值 | Pascal |  |
|  |  |  |
| 枚举类型 | Pascal |  |
|  |  |  |
| 事件 | Pascal |  |
|  |  |  |
| 异常类 | Pascal | 以 Exception 结尾 |
| 公开字段 | Pascal |  |
|  |  |  |
| 方法 | Pascal |  |
|  |  |  |
| 命名空间 | Pascal |  |
|  |  |  |
| 属性 | Pascal |  |
|  |  |  |
| 受保护的 / 私有的 | 骆驼式 |  |
| 字段 |  |  |
|  |  |  |
| 参数 | 骆驼式 |  |
|  |  |  |

1. **编程实践**

**9.1 可见性**

不要给实例或者类变量公开的属性，使它们具有私有的属性，要避免使用“private”关键字，这是标准修饰符，C#编程者应该知道这一点，因此什么都不需要写。

使用属性来代替类变量。你可以使用 public static 字段（或者 const）作为这个规则的一个例外，但是要谨慎使用。**9.2** **没有“魔法”的数字**

不要使用魔法数字，也就是说，把不变的数字直接放到你的源文件中。以后万一因为变化需要替换这些数字是错误的和没有好处的。解决方法就是，你把这个不变的数声明为一个常量。

public class MyMath

{

public const double PI = 3.14159...

}

1. **代码举例**

**10.1 大括号放置位置举例**

namespace ShowMeTheBracket

{

public enum Test {

TestMe,

TestYou

}

public class TestMeClass

{

Test test;

public Test Test {

get {

return test;

}

set {

test = value;

}

}

void DoSomething()

{

if (test == Test.TestMe) {

//...stuff gets done

} else {

//...other stuff gets done

}

}

}

}

一个大括号“{”应该开始在一个新行上，仅当它在以下条件之后：

* 命名空间声明
* 类、接口、结构的声明
* 方法的声明

**10.2** **变量命名举例**

代替：

for (int i = 1; i < num; ++i) {

meetsCriteria[i] = true;

}

for (int i = 2; i < num / 2; ++i) { int j = i + i;

while (j <= num) {

meetsCriteria[j] = false;

j += i;

}

}

for (int i = 0; i < num; ++i) {

if (meetsCriteria[i]) {

Console.WriteLine(i + " meets criteria");

}

}

尝试智能命名：

for (int primeCandidate = 1; primeCandidate < num; ++primeCandidate) {

isPrime[primeCandidate] = true;

}

for (int factor = 2; factor < num / 2; ++factor) { int factorableNumber = factor + factor; while (factorableNumber <= num) {

isPrime[factorableNumber] = false;

factorableNumber += factor;

}

}

for (int primeCandidate = 0; primeCandidate < num; ++primeCandidate) {

if (isPrime[primeCandidate]) { Console.WriteLine(primeCandidate + " is prime.");

}

}注意：索引变量一般使用 i, j, k 等，但是像这种情况下，考虑使用智能命名很有意义。总之，当相同的计数或者索引被再次使用，那就给他们一个有意义的名字。

**注释：**

当然，使用有语义的变量的名字例如 indentLevel，就没必要使用注释了。

**C#代码举例**

**public double fangweijiao(double[] sdr, double [] cr)**

**{**

**double sum = 0;**

**for (int i = 1; i < sdr.Length ; i++)**

**//从第二行开始循环计算坐标方位角、观测角度累加值**

**{**

**cr[i] = cr[i - 1] + sdr[i] - Math.PI;**

**//计算坐标方位角/左角if (cr[i] >= Math.PI \* 2)//判断坐标方位角是否在0到2PI之间**

**cr[i] -= Math.PI \* 2;**

**else if (cr[i] < 0.0) cr[i] += Math.PI \* 2;**

**sum += sdr[i];**

**}**

**return sum;**

**}**

**public double dmstorad (string s)**

**{**

**r=dmstorad(d)/开始调用**

**string[] ss = s.Split(new char[3] { '°', '′', '″' }, StringSplitOptions.RemoveEmptyEntries);**

**//用°、 ′、 ″分割字符串s**

**double[] d = new double[ss.Length];**

**//新建一个双精度数值数组for (int i = 0; i < d.Length; i++)**

**//将度分秒存入这个双精度数值数组中d[i] = Convert.ToDouble(ss[i]);**

**double sign = d[0] >= 0.0 ? 1.0 : -1.0;**

**//判断角度值是否为负值**

**double rad =0;** **if (d.Length == 1)**

**//根据数组的长度进行判断计算**

**rad = Math.Abs(d[0]) \* Math.PI / 180;**

**//将度取绝对值，并转换为弧度else if (d.Length == 2)**

**rad = (Math.Abs(d[0]) + d[1] / 60) \* Math.PI /180;**

**//将度取绝对值，分化为度，并最终转换为弧度**

**else**

**rad = (Math.Abs(d[0])+ d[1]/60+d[2]/60/60)\*Math.PI/180;**

**//将度取绝对值，分化为度，秒化为度，并最终转换为弧度**

**rad = sign \* rad;**

**//弧度前边添加正负号**

**}**

**{ double sign = rad >= 0.0 ? 1.0 : -1.0; //判断弧度值是否为负值**

**rad = Math.Abs(rad) \* 180 / Math.PI; //将弧度取绝对值，并转换为度**

**double[] d = new double[3];//新建一个长度为3的数组**

**d[0] = (int)rad;//取整获取度**

**d[1] = (int)((rad - d[0]) \* 60);//取整获取分**

**d[2] = (rad - d[0] - d[1] / 60) \* 60 \* 60;**

**//获取秒，不取整**

**d[2] = Math.Round(d[2], 2); //将秒保留两位小数**

**if (d[2] == 60) //检查秒是否等于60**

**{**

**d[1] += 1; d[2] -= 60;** **if (d[1] == 60)**

**{**

**d[0] += 1; d[1] -= 60;**

**}**

**}**

**d[0] = sign \* d[0];//度前边添加正负号**

**string s = Convert.ToString(d[0]) + "°" + Convert.ToString(d[1])**

**+ "′" + Convert.ToString(d[2]) + "″";**

**//将度分秒赋值给文本框，并添加°、 ′、 ″**

**}**

**for (int i = 1; i < sd.Length; i++)**

**//从第二行开始循环，将观测角度的原始值放入sd[]数组中,并转换成弧度值存放在sdr数组中**

**{**

**sd[i] = Convert.ToString(dataGridView1.Rows[i].Cells[1].Value); sdr[i] = dmstorad(sd[i]);**

**}**

**sum = fangweijiao(sdr, cr); //计算改正前坐标方位角和观测角度总和，分别存储在cr数组和sum中**

**dataGridView1.Rows[dataGridView1.RowCount - 4].Cells[1].Value = radtodms(sum);**

**//将观测角度总和放入表格中**

**double fd, fdx;**

**fd = cr[cr.Length - 1] - acd;//计算角度闭合差，单位弧度**

**fdx = 60 \* Math.Sqrt(sd.Length - 1);//计算角度闭合差限差，单位秒**

**dataGridView1.Rows[dataGridView1.RowCount - 3].Cells[1].Value = Convert.ToString(Math.Round(fd \* 180 / Math.PI \* 3600, 2))+"″";**

**//将角度闭合差存入表格中**

**dataGridView1.Rows[dataGridView1.RowCount - 2].Cells[1].Value =Convert.ToString(Math.Round(fdx, 2))+"″";**

**//将角度闭合差限差存入表格中**

**if (Math.Abs(fd \* 180 / Math.PI \* 3600) > fdx)**

**//检查角度闭合差是否满足要求MessageBox.Show("角度闭合差超限！");**

**else**

**{**

**double vd = -fd / (sd.Length - 1);//分配角度闭合差**

**double sumvd = 0;**

**for (int i = 1; i < sd.Length; i++)**

**{**

**sdr[i] += vd;//计算改正后的观测角度，并存入sdr数组中**

**sumvd += vd;**

**dataGridView1.Rows[i].Cells[2].Value =**

**Convert.ToString(Math.Round(vd \* 180 / Math.PI \* 3600, 2))+"″";**

**//将角度改正数存入表格中**

**dataGridView1.Rows[i].Cells[3].Value = radtodms(sdr[i]);**

**}**

**if (Math.Round(sumvd, 8) != Math.Round(-fd , 8))**

**MessageBox.Show("角度改正数分配有误！");**

**else**

**dataGridView1.Rows[dataGridView1.RowCount-4].Cells[2].Value = =Convert.ToString(Math.Round(sumvd \* 180 / Math.PI \* 3600, 2)) + "″";**

**//将角度改正数总和存入表格中**

**sum = fangweijiao(sdr, cr);//推算改正后的坐标方位角**

**if (Math.Round(cr[cr.Length - 1], 8) != Math.Round(acd, 8))**

**MessageBox.Show("坐标方位角推算有误！");**

**else**

**{**

**dataGridView1.Rows[dataGridView1.RowCount - 4].Cells[3].Value =radtodms(sum);**

**//将改正后观测角度总和放入表格中**

**for (int i = 0; i < sd.Length; i++)//将改正后坐标方位角存入表格**

**dataGridView1.Rows[i].Cells[4].Value = radtodms(cr[i]);**

**}**

**}**

**//至此角度调整和计算完毕**

**double[] sl = new double[sd.Length - 1];**

**double suml = 0;**

**double[] dx = new double[sd.Length - 1];**

**double[] dy = new double[sd.Length - 1];**

**double sumdx = 0;**

**double sumdy = 0;**

**for (int i=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 C#中函数**

**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]; //计算坐标增量总和保留小数位数：计算过程 VS 表格显示**

**sumdy += dy[i];**

**dataGridView1.Rows[i].Cells[6].Value = Convert.ToString(Math.Round (dx[i],3));**

**//将坐标增量放入表格**

**dataGridView1.Rows[i].Cells[7].Value = Convert.ToString(Math.Round (dy[i],3));**

**}** **dataGridView1.Rows[dataGridView1.RowCount - 4].Cells[5].Value = Convert.ToString(suml);**

**//将距离总和放入表格中**

**dataGridView1.Rows[dataGridView1.RowCount - 4].Cells[6].Value = Convert.ToString(Math .**

**Round (sumdx,3)); //将坐标增量总和放入表格中**

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

**double fx, 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**

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

**Convert.ToString(Math.Round(fx,3));**

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

**Convert.ToString(Math.Round(fy,3));**

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

**Convert.ToString(Math.Round(fxy, 3));**

**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**

**double[] vx = new double[sd.Length - 1]; //定义数组用于存放坐标增量的改正数及总和**

**double[] vy = new double[sd.Length - 1];**

**double sumvx = 0;**

**double sumvy = 0;**

**double[] cx = new double[sd.Length - 1]; //定义数组用于存放改正后的坐标增量及总和**

**double[] cy = new double[sd.Length - 1];**

**double sumcx = 0;**

**double sumcy = 0;**

**double[] x = new double[sd.Length - 1]; //定义数组用于存放x，y坐标**

**double[] y = new double[sd.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(int j=1;j<vx.Length;j++)**

**{**

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

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

**sumvx += vx[j]; //计算坐标增量改正数总和保留小数位数**

**sumvy += vy[j];**

**dataGridView1.Rows[j].Cells[8].Value =Convert.ToString(Math.Round(vx[j], 4));//将坐标增量改正数放入表格**

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

**cx[j] = dx[j] + vx[j]; //计算改正后坐标增量**

**cy[j] = dy[j] + vy[j];**

**sumcx += cx[j]; //计算改正后坐标增量总和**

**sumcy+= cy[j];**

**dataGridView1.Rows[j].Cells[10].Value = Convert.ToString(Math.Round(cx[j], 3));**

**//将改正后坐标增量放入表格**

**dataGridView1.Rows[j].Cells[11].Value = Convert.ToString(Math.Round(cy[j], 3));**

**}** **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！");**

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

**Math.Round(y3 - y2, 4))**

**MessageBox.Show("改正后坐标增量计算有误！");**

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

**Convert.ToString(Math.Round(sumvx, 3)); //将坐标增量改正数总和放入表格中**

**dataGridView1.Rows[dataGridView1.RowCount-4].Cells[9].Value = Convert.ToString(Math.Round(sumvy, 3));**

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

**Convert.ToString(Math.Round(sumcx, 3)); //将改正后坐标增量总和放入表格中**

**dataGridView1.Rows[dataGridView1.RowCount-4].Cells[11].Value = Convert.ToString(Math.Round(sumcy, 3));**

**for(int j=1;j<x.Length -1;j++ )**

**{**

**x[j + 1] = x[j] + cx[j];**

**//计算x,y坐标**

**y[j + 1] = y[j] + cy[j];**

**dataGridView1.Rows[j+1].Cells[12].Value= Convert.ToString(Math.Round(x[j+1], 3));**

**//将x,y坐标放入表格**

**dataGridView1.Rows[j+1].Cells[13].Value= Convert.ToString(Math.Round(y[j+1], 3));**

**}**

**if(Math.Round (x[x.Length -1]+cx[cx.Length -1],3)!=Math.Round (x3,3) || Math.Round (y[y.Length -1]+cy[cy.Length -1],3)!=Math .Round ( y3,3))**

**MessageBox.Show("坐标计算有误！");**

**}** **private void txt文件ToolStripMenuItem\_Click(object sender, EventArgs e)**

**{**

**dataGridView1.DataSource = null; //清除数据源**

**dataGridView1.Rows.Clear(); //清空数据表格的行**

**dataGridView1.Columns.Clear();//清空数据表格的列**

**OpenFileDialog file = new OpenFileDialog(); //实例化 打开文件类 file file.Filter = "文本文件|\*.txt"; //文件过滤器，只显示txt文件**

**if (file.ShowDialog() == DialogResult.OK) //如果文件正常打开**

**{**

**StreamReader sr = new StreamReader(file.FileName, System.Text.Encoding.Default);**

**//实例化文本读取流，并以文本编码格式读取 textBox1.Text = sr.ReadToEnd(); //将sr中的内容全部放到textBox中 sr.Close();**

**}**

**else**

**return;**

**string[] str = textBox1.Text.Split(new string[] { "\r\n" }, StringSplitOptions.RemoveEmptyEntries);**

**//将textBox1.Text中按行分割，并放在一维字符串数组中**

**string[][] k = new string[str.Length][];**

**//定义字符串交错数组，行数与str的长度相同**

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

**k[i] = str[i].Split(‘,’);//将str中对应字符串以逗号分割，并放在k中**

**dataGridView1.RowCount = str.Length;//定义表格控件的行数，与str长度相同 dataGridView1.ColumnCount = k[0].Length; //定义表格列数，与k[0]长度相同**

**for (int i = 0; i < k[0].Length; i++) //将k中第0行元素放入表格的表头 dataGridView1.Columns[i].Name = k[0][i];**

**for (int i = 1; i < str.Length; i++) //将k中数据元素放入对应的表格中**

**{**

**for (int j = 0; j < k[i].Length; j++) dataGridView1.Rows[i-1].Cells[j].Value = k[i][j];**

**}**

**}**