**1. 软件编码规范**

良好的编码规范是IT从业人员必备的素质，C++(包括C)和python语言是当前比较主流的编程语言。请参照google等编码规范并结合自己的编码习惯，完成C++或 python或C#或其它语言的编码规范，以供今后使用。具体要求：

(1) 提交一种编程语言的编码规范文档；

(2) 按照自己的规范提交对应的实例代码，**代码行必须50行以上，必须保证规范和实例的统一**。

1. **C#编码规范文档**
2. **C#编码规范**
3. 1. 简介
4. 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。
5. 2. 适用范围
6. 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。
7. 3. 文体
8. 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：
9. **要**：描述必须遵循的规范。例如：
10. Ö 异常类**要**以“Exception”做为后缀；
11. **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：
12. Ö 强制类型转换时，在类型和变量之间**建议**加一空格。
13. **不要**：描述一些几乎绝对绝不应该违反的规范。例如：
14. ´ 每个函数有效代码（不包括注释和空行）长度**不要**超过50行。
15. **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：
16. ´ **避免**块内部的变量与它外部的变量名相同。
17. 对一些规范内容一并提供了示例代码。
18. 4. 代码组织与风格
19. 4.1. Tab
20. Ö **要**使一个Tab为4个空格长。
21. 4.2. 缩进
22. Ö **要**使一个代码块内的代码都统一缩进一个Tab长度。
23. 4.3. 空行
24. Ö **建议**适当的增加空行，来增加代码的可读性。
25. Ö 在在类，接口以及彼此之间**要**有两行空行：
26. Ö 在下列情况之间**要**有一行空行：
27. 方法之间；
28. 局部变量和它后边的语句之间；
29. 方法内的功能逻辑部分之间；
30. 4.4. 函数长度
31. ´ 每个函数有效代码（不包括注释和空行）长度**不要**超过50行。
32. 4.5. {”，“}”
33. Ö 开括号“{”**要**放在块的所有者的下一行，单起一行；
34. Ö 闭括号“}”**要**单独放在代码块的最后一行，单起一行。
35. 4.6. 行宽
36. ´ 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。
37. 4.7. 空格
38. ´ 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) {};
39. ´ 但是方法名和左括号之间**不要**有空格。
40. Ö 参数之间的逗号后**要**加一空格。如：method1(int i1, int i2)
41. Ö for语句里的表达式之间**要**加一空格。如：for (expr1; expr2; expr3)
42. Ö 二元操作符和操作数之间**要**用空格隔开。如：i + c;
43. Ö 强制类型转换时，在类型和变量之间**要**加一空格。如：(int) i ;
44. 5. 注释
45. 5.1. 注释的基本约定
46. Ö 注释应该增加代码的清晰度；
47. 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。
48. ´ 注释**不要**包括其他的特殊字符。
49. Ö **建议**先写注释，后写代码，注释和代码一起完成
50. 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。
51. 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；
52. 5.2. 注释类型
53. 5.2.1. 块注释
54. 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：
55. /// <Remark>作者，创建日期，修改日期</ Remark >
56. 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
57. 5.2.2. 行注释
58. 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。
59. 5.2.3. 尾随注释
60. 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：
61. int m = 4 ; /／ 注释
62. 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。
63. 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 |

1. 5.4. 程序修改注释
2. 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；
3. 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
4. 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；
5. 6. 命名
6. 6.1. 命名的基本约定
7. 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 ”。
8. Ö **要**尽量采用项目所涉及领域的术语。
9. 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：
10. PasalCasing：标识符的第一个单词的字母大写；
11. camelCasing：标识符的第一个单词的字母小写。
12. 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; |

1. ´ **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。
2. 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）；
3. ´ **避免**使用长名字（最好不超过 15 个字母）。
4. ´ **避免**使用相似或者仅在大小写上有区别的名字。
5. 6.2. 各种标示符类型的命名约定
6. 6.2.1. 程序集命名
7. Ö 公司域名（Techstar）+ 项目名称 + 模块名称（可选），例如：
8. 中心系统程序集：Techstar.ProductionCenter；
9. 中心系统业务逻辑程序集：Techstar. ProductionCenter.Business；
10. 6.2.2. 命名空间命名
11. 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：
12. 中心系统：Techstar.ProductionCenter；
13. 中心系统下的用户控件：Techstar.ProductionCenter.UserControl；
14. 中心系统业务逻辑：Techstar. ProductionCenter.Business；
15. 中心系统数据访问：Techstar. ProductionCenter.Data；
16. 6.2.3. 类和接口命名
17. Ö 类的名字**要**用名词；
18. ´ 避免使用单词的缩写，除非它的缩写已经广为人知，如HTTP。
19. 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；
20. 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：
21. public class List<T>
22. public class MyClass<TSession>
23. ´ 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；
24. 6.2.4. 方法命名
25. Ö 第一个单词一般是动词
26. 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;
27. 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；
28. 6.2.5. 变量命名
29. 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：
30. i. 类的公有变量按通常的方式命名，无特殊要求；
31. ii. 类的私有变量采用两种方式均可：采用加“m”前缀，例如mWorkerName;
32. iii. 方法的参数变量采用camalString，例如workerName；
33. iv. 方法内部的局部变量采用camalString，例如workerName；
34. ´ **不要**用\_或&作为第一个字母；
35. Ö 尽量**要**使用短而且具有意义的单词；
36. 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
37. 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；
38. 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
39. 6.3. 组件名称缩写列表
40. 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 |

1. 7. 声明
2. 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;
3. 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++) 。
4. ´ **避免**块内部的变量与它外部的变量名相同。
5. 8. 表达式和语句
6. Ö 每行建议只有一条语句。
7. Ö if-else,if-elseif语句，任何情况下，都应该有“{”，“}”，格式如下：
8. if (condition)
9. {
10. statements;
11. }
12. else if (condition)
13. {
14. statements;
15. }
16. else
17. {
18. statements;
19. }
20. Ö for语句格式如下：
21. for (initialization; condition; update)
22. {
23. statements;
24. }
25. 如果语句为空：
26. for (initialization; condition; update) ;
27. Ö while语句格式如下：
28. while (condition)
29. {
30. statements;
31. }
32. 如果语句为空:
33. while (condition);
34. Ö do-while语句格式如下：
35. do
36. {
37. statements;
38. }
39. while (condition);
40. Ö switch语句，每个switch里都应包含default子语句,格式如下：
41. switch (condition)
42. {
43. case ABC:
44. statements;
45. /\* falls through \*/
46. case DEF:
47. statements;
48. break;
49. case XYZ:
50. statements;
51. break;
52. default:
53. statements;
54. break;
55. }
56. Ö try-catch语句格式如下：
57. try
58. {
59. statements;
60. }
61. catch (ExceptionClass e)
62. {
63. statements;
64. }
65. finally
66. {
67. statements;
68. }
69. 9. 类型设计规范
70. 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；
71. 9.1. 类型和命名空间
72. Ö **要**用命名空间把类型组织成相关域的层次结构。例如：
73. 界面层：Techstar.ProductionCenter；
74. 业务逻辑层：Techstar.ProductionCenter.Business；
75. 数据访问层：Techstar.ProductionCenter.Data；
76. ´ **避免**过深的命名空间；
77. ´ **避免**太多的命名空间；
78. 9.2. 类型和接口的选择
79. Ö **要**优先采用类而不是接口。
80. 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；
81. 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；
82. Ö **要**定义接口，来实现类似多重继承的效果；
83. 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。
84. 9.3. 抽象类设计：
85. ´ **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；
86. Ö **要**为抽象类定义受保护的构造函数或内部构造函数；
87. 9.4. 静态类设计
88. 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。
89. Ö **要**少用静态类。静态类应该仅用作辅助类；
90. ´ **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；
91. Ö **不要**在静态类中声明或覆盖实例成员；
92. 9.5. 枚举设计
93. 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；
94. Ö **要**优先使用枚举而不是静态常量。例如：
95. //不好的写法
96. public static class Color
97. {
98. public static int Red = 0;
99. public static int Green = 1;
100. public static int Blue = 2;
101. }
102. //好的写法
103. public enum Color
104. {
105. Red,
106. Green,
107. Blue
108. }
109. ´ **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；
110. ´ **枚举最后一个值不要加逗号；**
111. ´ 枚举中**不要**提供为了今后使用而保留的枚举值；
112. 10. 成员设计规范
113. 方法，属性，事件，构造函数以及字段等统称为成员。
114. 10.1. 成员设计的一般规范
115. 10.2. 方法的重载规范；
116. ´ **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：
117. public class String
118. {
119. //好的写法
120. public int IndexOf(string value) { ...}
121. public int IndexOf(string value, int startIndex) { ...}
122. //不好的写法
123. public int IndexOf(string value) { ...}
124. public int IndexOf(string str, int startIndex) { ...}
125. }
126. ´ **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：
127. public class EventLog
128. {
129. public EventLog();
130. public EventLog(string logName);
131. public EventLog(string logName, string machineName);
132. public EventLog(string logName, string machineName, string source);
133. }
134. 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：
135. public class String
136. {
137. public int IndexOf(string s)
138. {
139. //调用
140. return IndexOf(s, 0);
141. }
142. public int IndexOf(string s, int startIndex)
143. {
144. //调用
145. return IndexOf(s, startIndex, s.Length);
146. }
147. public virtual int IndexOf(string s, int startIndex, int Count)
148. {
149. //实际的代码
150. }
151. }
152. 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：
153. //允许为null时的调用
154. DrawGeometry(brush, pen, geometry);
155. //不允许为null时的调用
156. if (geometry == null) DrawGeometry(brush, pen);
157. else DrawGeometry(brush, pen, geometry);
158. 10.3. 属性和方法的选择
159. 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。
160. Ö **要**使用属性，如果该成员表示类型的逻辑attribue
161. 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
162. 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:
163. //好的写法
164. Guid.NewGuid();
165. //不好的写法
166. DateTime.Now;
167. 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。
168. Ö 如果该操作有严重的副作用，**要**使用方法。
169. 10.4. 属性的设计规范：
170. Ö 如果不应该让调用方法改变属性值，**要**创建只读属性；
171. ´ **不要**提供只写属性；
172. 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；
173. Ö **要**允许用户以任何顺序来设置属性的值；
174. Ö **避免**在属性的获取方法抛出异常。
175. 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。
176. 10.5. 构造函数的设计规范
177. 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；
178. 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；
179. 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；
180. 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；
181. 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。
182. 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；
183. ´ **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；
184. 10.6. 字段设计规范
185. ´ **不要**提供公有的或受保护的字段。代之以属性来访问字段；
186. 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：
187. public struct Int32
188. {
189. public const int MaxValue = 0x7fffffff;
190. public const int MinValue = unchecked((int)0x80000000);
191. }
192. 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：
193. public struct Color
194. {
195. public static readonly Color Red = new Color(0x0000FF);
196. }
197. 10.7. 参数的设计规范
198. 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：
199. 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。
200. ´ **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：
201. //不好的写法
202. public void Method(string reserved, SomeOption option);
203. //好的写法
204. public void Method(SomeOption option);
205. //将来填加
206. public void Method(SomeOption option, string path);
207. 10.7.1. 参数设计中枚举和布尔参数的选择规范
208. 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：
209. //使用布尔型，阅读的时候不会轻易了解参数的含义
210. FileStream f = File.Open(“1.txt”, true, false);
211. //使用枚举型
212. FileStream f = File.Open(“1.txt”，CasingOptions.CaseSenstive, FileMode.Open);
213. ´ **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。
214. 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；
215. 10.7.2. 参数验证的规范：
216. 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；
217. Ö **要**抛出System.ArgutmentNullException，如果传入的null，而该成员不支持null；
218. 10.7.3. 参数传递的规范：
219. ´ **避免**使用输出参数或引用参数；
220. 11. 扩展性设计规范
221. ´ 如果没有恰当理由，**不要**把类密封起来。这些理由包括：
222. A）类为静态类；
223. B）类的受保护成员保存了高度机密信息；
224. 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；
225. 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；
226. 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；
227. ´ **不要**使用虚成员，除非有合适的理由；
228. 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；
229. 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；
230. 12. 异常处理规范
231. 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；
232. ´ 一般情况下**不要使用异常实现来控制程序流程结构；**
233. ´ **使用异常而不要用错误代码来报告错误；**
234. 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；**
235. **12.1. 异常类型选择规范**
236. 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；**
237. 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
238. **12.2. 异常处理规范**
239. ´ 不是百分之百确定的情况，**不要**吞掉异常；
240. 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；
241. ´ **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；
242. Ö 进行清理工作时**要**用try-finally，避免使用try-catch；
243. 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
244. **12.3. 标准异常类的使用：**
245. 12.3.1. Exception与SystemException
246. ´ **不要**抛出这两种类型的异常；
247. ´ **避免**捕获这两种异常，除非是在顶层的异常处理器中；
248. 12.3.2. InvalidOperationException
249. Ö 对象处于不正确状态时抛出；
250. 12.3.3. ArgumentException，ArgumentNullException，ArgumentOutOfRangeException
251. 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；
252. Ö **要**在抛出异常时设置ParaName属性；
253. 12.3.4. NullRefernceException，IndexOutOfRangeException，AccessViolationException
254. ´ **不要**显示抛出或捕获；
255. 12.3.5. StackOverflowException：
256. ´ **不要**显示抛出或捕获；
257. 12.3.6. OutOfMemoryException：
258. ´ **不要**显示抛出或捕获；
259. **12.4. 自定义异常类型设计规则：**
260. ´ **避免**太深的继承层次；
261. Ö **要**从已有的异常基类继承；
262. Ö 异常类**要**以“Exception”做为后缀；
263. 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；
264. Ö **要**把与安全性有关的信息保存在私有的异常状态中
265. **12.5. 异常与性能**
266. 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；
267. 13. 其他规定
268. 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；
269. 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：
270. 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；
271. 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；
272. C）详设的时候打开开关，检查详设是否符合编程规范；
273. 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 |

1. **C#的实验封装方法代码及Excel文件导入代码**
2. //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！");  
     
     
           }