中软国际公司内部技术规范

C#语言编程规范

中软国际科技服务有限公司

修订声明

参考:微软.Net 规范，C/C++编码规范，华为 C#语言编码规范等。

日期	修订版本	修订描述	作者
2014-08-15	1.00	整理创建初稿	徐庆阳+x00108368
2014-10-25	1.01	格式整理	徐庆阳+x00108368
2014-11-3	1.02	整合华为线2012实验室C#规范	徐庆阳+x00108368
2014-11-10	1.03	根据项目组意见修改	徐庆阳+x00108368
2017-8-2	1.1	统一整理格式并修改部分错误	陈丽佳 cwx435329

目录

1. 前言

规范制定的目的

为了统一公司软件开发的设计过程中关于代码编写时的编写规范和具体开发工作时的编程规范，保证 代码的一致性，便于交流和维护，特制定此规范。

1 方便代码的交流和维护。

2 不影响编码的效率，不与大众习惯冲突。

3 使代码更美观、阅读更方便。

4 使代码的逻辑更清晰、更易于理解。

5 有效地保持程序的运行性能。

术语定义

Pascal 大小写

将标识符的首字母和后面连接的每个单词的首字母都大写。可以对三字符或更多字符的标识符使用 Pascal 大小写。

例如：BackGround

Camel 大小写

标识符的首字母小写，而每个后面连接的单词的首字母都大写。例如：backGround

2. 命名规范

2.1 命名概述

名称应该说明“是什么”而不是“怎么样”。通过避免使用公开基础实现（它们会发生改变）的名称， 可以保留简化复杂性的抽象层。例如，可以使用 GetNextStudent()，而不是 GetNextArrayElement()。

2.2 大小写规则

大写

标识符中的所有字母都大写。仅对于由两个或者更少字母组成的标识符使用该约定。例如： System.IO System.Web.UI 下表汇总了大写规则，并提供了不同类型的标识符的示例。

标识符	大小写	示例
类	Pascal	AppDomain
枚举类型	Pascal	ErrorLevel
枚举值	Pascal	FatalError
~~事件~~委托	Pascal	ValueChange
属性类	Pascal	ValueChangeAttribute 注意 总是以 Attribute后缀结尾。
异常类	Pascal	WebException 注意 总是以 Exception 后缀结尾。
只读的静态变量	Pascal	RedValue
接口	Pascal	IDisposable 注意 总是以 I 前缀开始。
方法	Pascal	ToString
命名空间	Pascal	System.Drawing
属性	Pascal	BackColor
公共实例变量	Pascal	RedValue 注意 很少使用。属性优于使用公共实例变量。
受保护的实例变量	Pascal	RedValue 注意 很少使用。属性优于使用受保护的实例变量。
私有的实例变量	Camel	redValue
参数	Camel	typeName
方法内的变量	Camel	backColor

2.3 文件命名

1、文件名遵从Pascal命名法，无特殊情况，扩展名小写；
2、使用统一而又通用的文件扩展名： C# 类 .cs；

2.4 命名空间

1、命名命名空间时的一般性规则是使用公司名称，后跟项目名称和可选的功能与设计，如下所示。

CompanyName.ProductName[.Feature][.Design]

例如：namespace Huawei.Emergency　　//华为调度台命名空间

2、命名空间使用Pascal大小写，用小圆点分隔开。

3、命名空间和类不能使用同样的名字。例如，有一个类被命名为Debug后，就不要再使用Debug作 为一个名称空间名。

2.5 类

1、使用 Pascal 大小写。 2、用名词或名词短语命名类。

3、使用全称避免缩写，除非缩写已是一种公认的约定，如URL、HTML。

2.6 接口

以下规则概述接口的命名指南：

1、用名词或名词短语，或者描述行为的形容词命名接口。例如，接口名称 IComponent 使用描述性名词。接口名称 ICustomAttributeProvider 使用名词短语。名称 IPersistable 使用形容词。 2、使用 Pascal 大小写。

3、给接口名称加上字母 I 前缀，以指示该类型为接口。在定义类/接口对（其中类是接口的标准 实现）时使用相似的名称。两个名称的区别应该只是接口名称上有字母 I 前缀。 以下是正确命名的接口的示例。

public interface IServiceProvider public interface IFormatable

以下代码示例阐释如何定义 IComponent 接口及其标准实现 Component 类。

```
public interface IComponent
{
        // Implementation code goes here.
}

public class Component: IComponent
{
        // Implementation code goes here.
}
```

2.7 常量

以下规则概述常量的命名指南： 所有单词大写，多个单词之间用 "_" 隔开。

例如：public const string PAGE_TITLE = "Welcome";

2.8 变量

实例变量

以下规则概述变量的命名指南：

1、private 表示法： m_ + Camel 大小写。（属性不相关变量）_ + Camel 大小写。（属性相关变量）

2、Public、Protected 或者 Internal 使用 Pascal 大小写。

3、拼写出变量名称中使用的所有单词。仅在开发人员一般都能理解时使用缩写。下面是正确 命名的变量的示例。

```
Public class SampleClass
{
    Protected int Salary; Private String m_url;
    Private String m_destinationUrl; Private String _name
    Public String Name
  {
          Get{ return _name; }
  }
}
```

2.8.1 静态变量

以下规则概述静态变量的命名指南：

1、命名规则同实例变量；

2、建议尽可能使用静态属性而不是公共静态变量。

2.8.2 局部变量

以下规则概述局部变量的命名指南：Camel 大小写。 例如：int intValue = 0;

2.8.3 循环变量

以下规则概述循环变量的命名指南： 即使对于可能仅出现在几个代码行中的生存期很短的变量，仍然使用有意义的名称。仅对于短 循环使用单字母变量名，如 i 或 j。

2.8.4 布尔变量

以下规则概述布尔变量的命名指南： 对布尔变量进行命名时，用 is,has 等词开头。

例如：Boolean hasFound = false; //是否已经找到了

2.9 属性 （ property）

以下规则概述属性的命名指南：

1 使用名词或名词短语命名属性。

2 使用 Pascal 大小写。 例如：

```
Private String _userName = "";
Public String UserName
{
        Get{Return _userName;}
        Set{_userName = Value;}
}
```

2.10 属性类 （Attribute）

应该总是将后缀 Attribute 添加到自定义属性类。以下是正确命名的属性类的示例。

```
 public class ObsoleteAttribute
{
}
```

2.11 枚举

枚举（Enum）值类型从 Enum 类继承。以下规则概述枚举的命名指南： 1、对于 Enum 类型和值名称使用 Pascal 大小写。

2、少用缩写。

3、不要在 Enum 类型名称上使用 Enum 后缀。

4、对大多数 Enum 类型使用单数名称，但是对作为域的 Enum 类型使用复数名称。 5、总是将 FlagsAttribute 添加到位域 Enum 类型。

例如：

```
[FlagsAttribute] Public Enum UserType
{
  Type1 = 1,
  Type2 = 2,
  Type3 = 4,
  Type4 = 8,
}
```

2.12 参数

以下规则概述参数的命名指南： 1、使用描述性参数名称。参数名称应当具有足够的描述性，以便参数的名称及其类型可用于

在大多数情况下确定它的含义。 2、参数名称表示法：Camel 大小写。

以下是正确命名的参数的示例

```
Type GetType(String strTypeName)

String Format(String strFormat, object[] args)
```

2.13 函数

以下规则概述方法的命名指南：

1、使用动词或动词短语命名方法。

2、使用 Pascal 大小写。

3、以下是正确命名的方法的示例。

```
Private void RemoveAll(){}

Private ArrayList GetCharArray(){}

 Private void Invoke(){}
```

2.14 事件

以下规则概述事件的命名指南： 1、对事件处理程序名称使用 EventHandler 后缀。

2、指定两个名为 sender 和 e 的参数。sender 参数表示引发事件的对象。sender 参数始 终是 object 类型的，即使在可以使用更为特定的类型时也如此。与事件相关联的状态封 装在名为 e 的事件类的实例中。对 e 参数类型使用适当而特定的事件类。

3、用 EventArgs 后缀命名事件参数类。 4、考虑用动词命名事件。

5、使用动名词（动词的“ing”形式）创建表示事件前的概念的事件名称，用过去式表示事 件后。例如，Close 事件应当具有 Closing 事件和 Closed 事件。不要使用 BeforeXxx/AfterXxx 命名方式。

6、不要在类型的事件委托声明上使用前缀或者后缀。例如，使用 Close，而不要使用 OnClose。

7、通常情况下，对于可以在派生类中重写的事件，应在类型上提供一个受保护的方法（称为 OnXxx）。此方法只应具有事件参数 e，因为发送方总是类型的实例。

以下示例阐释具有适当名称和参数的事件处理程序。

```
public delegate void MouseEventHandler(object sender, MouseEventArgs e);
```

以下示例阐释正确命名的事件参数类。

```
public class MouseEventArgs : EventArgs
{
        int _X = 0;
        int _Y = 0;
        public MouseEventArgs(int intX, int intY)
        {
                this.m_X = intX;
```

```
                this.m_Y = intY;
            }

        public int X
        {
                get{return _X;}
            }
            public int Y
        {
                get{return _Y; }
            }
    }
```

2.15 集合

如果实现了ICollection接口，应该总是将后缀 Collection 添加到类名称后面，如果实现了IList 接口，应该总是将后缀 List添加到类名称后面，集合是一组组合在一起的类似的类型化对象，如哈希表、 查询、堆栈、字典和列表，集合的命名建议用复数。

2.16 控件

命名方法:控件名简写 + 英文描述，英文描述首字母大写

主要控件名简写对照表

数据类型	数据类型简写	标准命名举例
Label	lbl	lblMessage
LinkLabel	llbl	llblToday
Button	btn	btnSave
TextBox	txt	txtName
MainMenu	mmnu	mmnuFile
CheckBox	chk	chkStock
RadioButton	rbtn	rbtnSelected
GroupBox	gbx	gbxMain
PictureBox	pic	pic Image
Panel	pnl	pnlBody
DataGrid	dgrd	dgrdView
ListBox	lst	lstProducts
CheckedListBox	clst	clstChecked

ComboBox	cbo	cboMenu
ListView	lvw	lvwBrowser
TreeView	tvw	tvwType
TabControl	tctl	tctlSelected
DateTimePic ker	dtp	dtpStartDate
HscrollBar	hsb	hsbImage
VscrollBar	vsb	vsbImage
Timer	tmr	tmrCount
ImageList	ilst	ilstImage
ToolBar	tlb	tlbManage
StatusBar	stb	stbFootPrint
OpenFileDialog	odlg	odlgFile
SaveFileDialog	sdlg	sdlgSave
FoldBrowserDialog	fbdlg	fgdlgBrowser
FontDialog	fdlg	fdlgFoot
ColorDialog	cdlg	cdlgColor
PrintDialog	pdlg	pdlgPrint

说明：由于.NET 平台提供了大量控件，每一个定义命名前缀难以记忆而且容易重名，也可以尝试系统控件（主要位于 System.Windows.Forms 下面）统一使用 ctrl 开头，用户控件使用 ucl 开头。

2.17 关键字

避免使用与常用的 .NET 框架命名空间重复的类名称。例如，不要将以下任何名称用作类名称：System、Collections、Forms 或 UI。有关 .NET 框架命名空间的列表，请参阅类库。 另外，避免使用和以下关键字冲突的标识符。

AddHandler	AddressOf	Alias	And	Ansi
As	Assembly	Auto	Base	Boolean
ByRef	Byte	ByVal	Call	Case
Catch	CBool	CByte	Cchar	CDate
CDec	CDbl	Char	Cint	Class
CLng	CObj	Const	Cshort	CSng
CStr	CType	Date	Decimal	Declare
Default	Delegate	Dim	Do	Double
Each	Else	ElseIf	End	Enum
Erase	Error	Event	Exit	ExternalSour
False	Finalize	Finally	Float	For
Friend	Function	Get	GetType	Goto
Handles	If	Implement	Imports	In
Inherits	Integer	Interface	Is	Let
Lib	Like	Long	Loop	Me

Mod	Module	MustInher	MustOver	MyBase
MyClass	Namespace	New	Next	Not
Nothing	NotInherita	NotOverri	Object	On
Option	Optional	Or	Overload	Overridable
Overrides	ParamArray	Preserve	Private	Property
Protected	Public	RaiseEven	ReadOnly	ReDim
Region	REM	RemoveHan	Resume	Return
Select	Set	Shadows	Shared	Short
Single	Static	Step	Stop	String
Structure	Sub	SyncLock	Then	Throw
To	True	Try	TypeOf	Unicode
Until	volatile	When	While	With
WithEvents	WriteOnly	Xor	Eval	extends
instanceof	package	var		

2.18 声明

2.18.1 每行声明数

一行只建议作一个声明，并按字母顺序排列。

```
如 int intLevel;  //推荐
int intSize; //推荐
int intX, intY;   //不推荐
```

2.18.2 初始化

在变量声明时就对其做初始化。

```
 例如：int intValue = 0;
String strValue = "" ;
Object obj = null;
```

2.18.3 位置

变量建议置于块的开始处，不要总是在第一次使用它们的地方做声明。

```
如 Private void MyMethod()
{
      int int1 = 0;    // beginning of method block
      if (condition)
      {
          int int2 = 0;    // beginning of "if" block
```

```
            ...
        }
    }
```

不过，对于以下情况把声明放在第一次使用之前能更清楚的指明初始值，便于理解 1）用于循环内计数、操作的变量定义

2）使用函数、属性等返回值初始化

```
int counter = GetCounter();
FirstFunction(counter);
SecondFunction(counter);
```

2.18.4 类和接口的声明

1、在方法名与其后的左括号间没有任何空格。

2、左花括号 “{” 出现在声明的下行并与之对齐，单独成行。

3、方法间用一个空行隔开。

2.18.5 变量的声明

不要使用是 public 或 protected 的实例变量。如果避免将变量直接公开给开发人员，可以更轻松 地对类进行版本控制，原因是在维护二进制兼容性时变量不能被更改为属性。考虑为变量提供 get 和 set 属性访问器，而不是使它们成为公共的。 get 和 set 属性访问器中可执行代码的存在使得 可以进行后续改进，如在使用属性或者得到属性更改通知时根据需要创建对象。下面的代码示例阐 释带有 get 和 set 属性访问器的私有实例变量的正确使用。 示例：

```
public class Control: Component
{
   private int _handle;
   public  int Handle
   {
      get
      {
         return _handle;
      }
   }
}
```

2.19 其他

其它主要事项有：

1、避免容易被主观解释的难懂的名称，如函数名 AnalyzeThis()，或者属性名 xxK8。这样的名称会 导致多义性。

2、在类属性的名称中包含类名是多余的，如 Book.BookTitle。而是应该使用 Book.Title。 3、不要出现魔鬼数字，字符文字等“硬编码”在编码中，而是使用命名常数，以便于维护和理解。

例如：

for(i=1; i<7; i++) 应变成

for(i=1; i< NUM_DAYS_IN_WEEK; i++)

3、查询多条记录建议使用 Find、List ，查询单条记录建议使用 Retriev。 如：

```
public void  ListAllRecords()
{
          ……
}

public void FindAllRecords()
{
          ……
}

public void  RetrieveOneRecord()
{
          ……
}
```

3、尽量使各种名称具有对称性。 如：

Add / Remove

Insert / Delete

Get / Set

Start / Stop

Begin / End

First / Last

Get / Release

Put / Get

Up / Down

Show / Hide

Source / Target

Open / Close

Source / Destination

Increment / Decrement

Lock / Unlock

Old / New

Next / Previous

Min / Max

3. 程序注释

3.1 注释概述

1、修改代码时，总是使代码周围的注释保持最新。

2、在每个例程的开始，提供标准的注释以指示例程的用途、假设和限制很有帮助。注释应该是解释它为什么存在和可以做什么的简短介绍。

3、避免在代码行的末尾添加注释；行尾注释使代码更难阅读。不过在批注变量声明时，行尾注释是合适的； 在这种情况下，将所有行尾注释在公共制表位处对齐。

4 、避免杂乱的注释，如一整行星号。而是应该使用空白将注释同代码分开。

5 、避免在块注释的周围加上印刷框。这样看起来可能很漂亮，但是难于维护。

6 、在部署发布之前，移除所有临时或无关的注释，以避免在日后的维护工作中产生混乱。

7 、如果需要用注释来解释复杂的代码节，请检查此代码以确定是否应该重写它。尽可能不注释难以理解 的代码，而应该重写它。尽管一般不应该为了使代码更简单以便于人们使用而牺牲性能，但必须保持性能和可维护性之间的平衡。

8 、在编写注释时使用完整的句子。注释应该阐明代码，而不应该增加多义性。

9 、在编写代码时就注释，因为以后很可能没有时间这样做。另外，如果有机会复查已编写的代码，在今 天看来很明显的东西六周以后或许就不明显了。

10、 使用注释来解释代码的意图。它们不应作为代码的联机翻译。

11、 注释代码中不十分明显的任何内容。

12 、为了防止问题反复出现，对错误修复和解决方法代码总是使用注释，尤其是在团队环境中。

13 、对由循环和逻辑分支组成的代码使用注释。这些是帮助源代码读者的主要方面。

14 、在整个应用程序中，使用具有一致的标点和结构的统一样式来构造注释。

15 、用空白将注释同注释分隔符分开。在没有颜色提示的情况下查看注释时，这样做会使注释很明显且容 易被找到。

16 、在所有的代码修改处加上修改标识的注释。

17 、采用**文档型注释**。该类注释采用.Net 已定义好的 Xml 标签来标记，在声明接口、类、方法、属性、变量都应该使用该类注释，以便代码完成后直接生成代码文档，让别人更好的了解代码的实现和接口。

3.2 文件注释

在每个文件头必须包含以下注释说明：

```
/*****************************************************************************
// 版权所有 2010xx 技术有限公司
// 程 序 集：Emergency CC Dispatch
// 文 件 名：Meeting.cs
// 文件说明：临时会议控件
// 版　本：
// 作　者：KF88888
```

```
// 创建日期：2010/01/25
//
// 修改标识：
// 修改说明：
*********************************************************************************/
```

文件说明只需简述，具体详情在类的注释中描述。

3.3 类和接口注释

```
/// <summary>
/// 功能描述：调度员类
///    包括了调度员的信息，以及基本的调度员的操作（取得信息，签入，签出等）
/// 作者：   KF23565
/// 创建日期：2010-01-25
/// </summary>
public class OrderSynchronous : IOrderStrategy
{
          …
}
```

3.4 方法注释

推荐使用 XML 的格式对函数进行注释

```
/// <summary>
/// 功能描述：******************
/// 作者：   KF8888
/// 创建日期：2010-02-01
/// </summary>
/// <param name="user">用户信息</param>
/// <returns>True;已经存在;False:不存在</returns>
 private Boolean IsUserInTempMeet(String user)
{
          ...
}
```

3.5 方法内注释

采用单行注释，如下所示：

```
// Get an instance of the Product DAL using the DALFactory
 IProduct dal = PetShop.DALFactory.Product.Create();
```

3.6 变量注释

在行尾进行注释，并使注释对齐。

3.7 错误修改注释

在代码发布之后，若发现 Bug 修改代码时，则应用如下注释标识被修改的代码。

```
// BEGIN: Added by 员工号, 2010/01/14 问题单号:XXXXXXXX
          Source code
// END: Added by 员工号, 2010/01/14 问题单号:XXXXXXXXX
// BEGIN: Modified by 员工号, 日期 问题单号: XXXXXXXXX
          Source code
// END: Modified by 员工号, 日期 问题单号: XXXXXXXXX

// BEGIN: Delete by 员工号, 日期 问题单号: XXXXXXXXX
// END: Delete by 员工号, 日期   问题单号: XXXXXXXXX
```

也可以通过版本控制工具记录，或者在方法块内的头部单独说明。

4. 代码布局

4.1 概述

统一规范的代码布局，可以使代码更美观，让阅读者赏心悦目。虽然下面有些规定比较细致，但一旦 严格遵守并养成习惯，会有意想不到的效果。

4.2 列宽

代码列宽控制在 80 字符左右。以代码行在 VisualStudio 的默认布局的编辑框里能完全显示为标准（17 寸显示器）。

4.3 换行

当表达式超出或即将超出规定的列宽，遵循以下规则进行换行

1、在逗号后换行。

2、在操作符前换行。

3、规则 1 优先于规则 2。 当以上规则会导致代码混乱的时候自己采取更灵活的换行规则。

4.4 缩进

缩进应该是每行一个 Tab(4 个空格)，不要在代码中使用 Tab 字符。

Visual Studio.Net 设置：工具->选项->文本编辑器->C#->制表符->插入空格

4.5 空行

空行是为了将逻辑上相关联的代码分块，以便提高代码的可阅读性。 在以下情况下使用两个空行

1、接口和类的定义之间。

2、枚举和类的定义之间。

3、类与类的定义之间。

在以下情况下使用一个空行

1、属性与属性之间。

2、方法与方法之间。

3、属性与方法、属性与变量、方法与变量之间。

4、方法中变量声明与语句之间。

5、方法中不同的逻辑块之间。

6、方法中的返回语句与其他的语句之间。

7、注释与它注释的语句间不空行，但与其他的语句间空一行。

4.6 空格

在以下情况中要使用到空格

1、 关键字和左括符 “(” 应该用空格隔开。如

```
while (true)
```

注意在方法名和左括符 “(” 之间不要使用空格，这样有助于辨认代码中的方法调用与关键字。

2、 多个参数用逗号隔开，每个逗号后都应加一个空格。

3、 除了 . 之外，所有的二元操作符都应用空格与它们的操作数隔开。一元操作符、++及-- 与操作数间不需要空格。如

```
a += c + d;
a = (a + b) / (c * d);
 while (d++ = s++)
{
      n++;
}
PrintSize(“size is “ + size + “\n”);
```

4、 语句中的表达式之间用空格隔开。如

```
for(expr1; expr2; expr3)
```

4.7 括号 - ()

1、 左括号“(” 不要紧靠关键字，中间用一个空格隔开。

2、 左括号“(” 与方法名之间不要添加任何空格。

3、 没有必要的话不要在返回语句中使用()。如

```
 if (condition)
Array.Remove(1)
 return 1
```

4.8 花括号 - {}

1、 左花括号 “{” 放于关键字或方法名的下一行并与之对齐。如

```
if (condition)
{
}
public int Add(int x, int y)
{
}
```

2、 左花括号 “{” 要与相应的右花括号 “}” 对齐。

3、 通常情况下左花括号 “{”单独成行，不与任何语句并列一行。

4、 if、while、do 语句后一定要使用{}，即使{}号中只有一条语句。如

```
if (somevalue == 1)
{
    somevalue = 2;
}
```

针对 4.6/7/8 关于代码中的空格:
建议使用 VS 的默认格式化设置，通过 Edit->Advanced->Format Document 自动格式化。

4.9 语句相关的布局

4.9.1 每行一个语句

每行最多包含一个语句。如

```
a++;     //推荐
b--;     //推荐
a++; b--;  //不推荐
```

4.9.2 复合语句

复合语句是指包含"父语句{子语句;子语句;}"的语句，使用复合语句应遵循以下几点

1 子语句要缩进。

2 左花括号“{” 在复合语句父语句的下一行并与之对齐，单独成行。

3 即使只有一条子语句要不要省略花括号“ {}”。 如

```
while (d + =  s++)
{
      n++;
 }
```

4.9.3 return 语句

return 语句中不使用括号，除非它能使返回值更加清晰。

```
return;
return myDisk.size();
return (size ? size : defaultSize);
```

4.9.4 if、if-else、if else-if 语句

if、 if-else、if else-if 语句使用格式

```
 if (condition)
```

```
{
    statements;
}

if (condition)
{
statements;
}
else
{
    statements;
}

if (condition)
{
    statements;
}
else if (condition)
{
    statements;
}
else
{
    statements;
}
```

4.9.5 switch - case 语句

switch - case 语句使用格式

```
switch (condition)
{
    case 1:
         statements;
         break;
    case 2:
         statements;
          break;
     default:
          statements;
```

```
            break;
    }
```

注意：

1、语句 switch 中的每个 case 各占一行。

2、语句 switch 中的 case 按字母顺序排列。

3、为所有 switch 语句提供 default 分支。

4、所有的非空 case 语句必须用 break，如果有未加 break 语句的，必须加特殊说明。

4.9.6 for、foreach 语句

for 语句使用格式

```
for (initialization; condition; update)
{
    statements;
}
```

空的 for 语句(所有的操作都在 initialization、condition 或 update 中实现)使用格式 for (initialization; condition; update);

foreach 语句使用格式

```
foreach (object obj in array)
{
    statements;
}
```

注意

1 在循环过程中不要修改循环计数器。

2 对每个空循环体给出确认性注释。

4.9.7 while 语句

while 语句使用格式

```
while (condition)
{
    statements;
}
```

空的 while 语句使用格式

```
while (condition);
```

4.9.8 do - while 语句

do－while 语句使用格式 do

```
{
```

```
    statements;
} while (condition);
```

4.9.9 try - catch 语句

try - catch 语句使用格式

```
try
{
    statements;
}
catch (ExceptionClass e)
{
    statements;
}
finally
{
  statements;
}
```

4.9.10 using 块语句

using 块语句使用格式

```
using (object)
{
    statements;
}
```

名称空间的“using”语句必须置于文件顶端，并且.NET名称空间置于客制化名称空间之前

4.9.11 goto 语句

goto 语句使用格式

```
goto Label1:
     statements;
Label1:
     statements;
```

5. 开发规范

5.1 一般原则

1、 必须显式地使用访问修饰符，而不是隐含地使用缺省的访问修饰符；

```
//Bad!
void WriteEvent(string message)
{…}

//Good!
Private void WriteEvent(string message)
{…}
```

2、 不要使用缺省的程序集版本方案（“1.0.*”），要求手工增加 AssemblyVersionAttribute 的值；

3、 如果多段代码重复做同一件事情，那么在函数的划分上可能存在问题；

4、 要注意 private、protected、internal、public 的使用范围，不要随便将函数的范围定义为 public；

关于它们的使用范围，从小到大排列为：

1）private（主要用于类的内部）

2）protected（除了可以用于本类外，还可用于子类） 3）internal（在类库、组件的内部均可使用，比如：整个业务层都可适用）

4）public（系统的整个生命周期都能使用，范围最大）

5、 在类内部，将同样类型/访问级别的成员放在一起；

6、 代码文件（.cs)只包含一个类型对象（类，接口，结构，枚举等）；

7、 类的继承深度最好不要超过 5；

8、 函数的圈复杂度尽量控制在 7 以内；

9、 函数的代码行数不超过 100 行。

5.2 变量和类型

1、 尽量在声明变量的同时进行初始化；

2、 尽量使用最简单的数据类型（data type），列表（list）或者对象（object）。例如，除非确实需要存储 64 位的值，使用 int 而不是 long；

3、 使用 C＃内建的数据类型，而不是.NET 的公共类型系统（CTS）；

例：
使用short，不使用System.Int16；
使用int，不使用System.Int32； 使用long，不使用System.Int64； 使用string，不使用System.String；

4、 成员变量尽量设为 private，使用属性进行访问控制，根据需要设置为 public、protected 或者 internal；

5、 避免特别定义 enum 的类型，使用缺省的 int 类型；

6、 尽量避免强制类型转换，如果不得不做类型换，尽量用显式方式；

7、 只对简单类型使用 constants；如果是复杂类型，使用 readonly 或者 static readonly；

8、 避免直接做 casts，应该使用“as”关键字并且判空；

例：
```
// Good!
object dataObject = LoadData();
Dataset ds = dataObject as Dataset;
if (ds!=null)
{…}
```

9、 总是优先使用 C#的范型集合（ Generic Collection）类型 ，而不是标准（ standard）或者强类型（strong-typed）的集合；

10、总是使用 for 循环显式地初始化引用数组；

11、避免对值类型进行装箱（boxing）与拆箱（un-boxing）操作；

例：
```
// Bad!
int count = 1;
object refCount = count;    //  隐式的装箱
int newCount = (int)refCount;//  显式的拆箱
```

12、尽量在 string 字串前加“@”前缀代替转意符；

13、尽量使用 String.Format()或者 StringBuilder，而不是字符串直接相加（concatenation）；

14、避免在循环内做字符串的相加；

15、判断空字符串，使用 string.IsNullOrEmpty；

16、初始化空字符串使用常量 string.Empty，而不使用 strings = ″″；

17、在循环里避免隐含的 string 分配（allocation），使用 String.Compare()；

例：

```
// Bad!
int id = -1;
string name = “lance hunt”;
for(int i=0; i < customerList.Count; i++)
{
    if(customer.List[i].Name.ToLower() == name)
    {
        id = customerList[i].ID;
    }
}
// Good!
int id = -1;
string name = “lance hunt”;
for(int i=0; i < customerList.Count; i++)
{
    //  参数“ignoreCase = true”执行大小写不敏感的比较
    if(String.Compare(customerList[i].Name,  name,
    true)==0)
    {
        id = customerList[i].ID;
    }
```

5.3 流程控制

1、 避免在条件表达式中调用方法函数；

2、 避免在条件语句中赋值；

例：

```
// Bad!
if ((i = 2) == 2)
```

3、 避免对浮点类型做等于或不等于判断；

4、 避免使用递归调用的方法，尽量使用循环；

5、 禁止在 foreach 中修改枚举项；

6、 foreach 语句是对枚举数的包装，它只允许从集合中读取，不允许写入集合。也就是，不能在 foreach 里 遍历的时侯把它的元素进行删除或增加的操作的。

7、 三目条件操作符只对简单的条件判断使用；

例：

```
// Good!
int result = isValid  ? 9 : 4;
```

8、 避免对 ture 或者 false 做布尔条件判断；

例：

```
// BAD!
if (isValid  == true)
{…}

// GOOD!
if (isValid)
{…}
```

9、 避免组合条件表达式，使用 Boolean 变量将组合条件表达式分解为多个部分；

```
// BAD!
if (((value > _highScore)  && (value != _highScore))  && (value < _maxScore))
{…}

// GOOD!
isHighScore  =  (value  >=
_highScore); isTiedScore = (value ==
_highScore);  isValid  =  (value  <
_maxValue);
if ((isHighScore  && ! isTiedScore)  && isValid)
```

10、只对有并行条件逻辑的简单操作使用 switch/case 语句，如果可以尽量使用嵌套的 if/else 语句；

5.4 事件、代理与线程

1、 在调用事件与代理实例之前，必须判空；

2、 对于大多数简单的事件，使用缺省的 EventHandler 和 EventArgs；

3、 如果要提供额外的数据，总是要继承 EventArgs 类；

4、 使用现成的 CancelEventArgs 类，让事件订阅者可以控制事件

5、 总是使用“lock”关键字，而不使用 Monitor 类型；

6、 只对一个私有或者私有静态的对象执行“lock”操作；

7、 避免对一个类型加“lock”操作；

例：

```
// Bad!
lock(typeof(MyClass));
```

8、 避免对当前对象实例执行“lock”操作；

例：

```
// Bad!
lock(this);
```

5.5 对象编写

1、 总是在某一名称空间中显式地申明类型，不要使用缺省的“{global}”名称空间；

2、 避免申明超过 5 个参数的方法，可以考虑使用结构或者类传递参数；

3、 避免使用“new”关键字隐藏派生类成员；

4、 只在调用基类构造函数或者在覆盖（override）方法中调用基类的实现时，才使用“base”；

5、 在密封（sealed）类中不要使用 protected 访问限制符；

6、 尽量使用方法重载（overloading），不使用 params attribute；

7、 在使用枚举变量或者参数前，要校验；

例：

```
// Good!
public void Test(BookCategory cat)
{
    if (Enum.IsDefined(typeof(BookCategory), cat))
        {...}
}
```

8、 通常对于 struct，需要重写（overriding）Equals()；

9、 在重写（override）Equals()方法的同时，通常需要重写 Equality Operator（==）；

10、在重写（override）ToString()方法的同时，通常需要重写 String Implicit Operator；

11、通常要调用类的 Close()或者 Dispose()方法，如果有的话；

12、在初始化实现了 IDsiposable 接口的对象时，用“using”语句，确保其 Dispose()函数可以被自动调用；

例：

```
// Good!
using(SqlConection   cn = new SqlConnection(_connectionString))
{…}
```

13、在实现引用外部资源的类时，总是要实现 IDisposable 接口和模式；

14、避免定义和实现 Finalize 方法，使用 C#的析构关键字；

例：

```
// GOOD!
~MyClass{…}

// BAD!
void Finalize(){…}
```

5.6 例外处理

1、 禁止使用 try/catch 做流程控制；

2、 只捕捉可以处理的异常（Only catch exceptions that you can handle.）；

3、 永远不要声明一个空的 catch 块；

4、 避免在 catch 块中嵌套使用 try/catch；

5、 尽可能使用例外过滤器（exception filters）；

6、 避免重复抛出（re-throwing）例外，可以让例外冒泡（bubble-up）；

7、 如果重复抛出异常，在 thow 语句中省略异常参数，可以将原先的调用栈得以保留；

8、 使用 finally 块释放 try 语句中使用的资源；

9、 尽量使用有效性判断以避免 try/catch；

例：

```
// BAD!
try
{
    conn.Close();
}
catch(Exception ex)
{
    //  处理如果连接已经关闭的异常
}

// GOOD!
```

10、尽量使用现有的例外类型，避免定义客制化的例外类型；

11、如果需要定义客制化例外类型：

- ✧ 总是从 Exception 继承，而不是从 ApplicationException 继承；
- ✧ 总是覆盖 ToString()方法和 String implicit 操作符，提供序列化功能；

总是实现“例外构造器模式”：

例：

```
// Good!
public MyCustomException();
public MyCustomException(stringmessage);
public MyCustomException(stringmessage, Exception innerException);
```

12、当抛出一个新的例外，总是要传递内部例外（innterException），以保持例外树和内部的调用栈；

5.7 对象模型设计

1、 优先使用代理，而不是继承；

2、 优先使用接口，而不是抽象类；

3、 展示层要和业务逻辑层分离；

4、 务必要使对象行为对 API 消费者透明；

5、 尽量在类名后添加所使用的设计模式名；

6、 对于为了扩展性良好而设计的成员，要加 virtual 关键字。