# 前言

## 1.1 编写目的

为了保证企业编写出的程序都符合相同的规范，保证一致性、统一性。程序员可以了解任何代码，弄清程序的状况，新人可以很快的适应环境。防止新接触.net的人出于节省时间的需要，自创一套风格并养成终生的习惯；防止新接触.net的人一次次的犯同样的错误而建立的程序编码规范。

## 1.2 范围

适用于企业所有基于.NET平台的软件开发工作。

# 2. 规范内容

## 2.1 代码格式

* 所有的缩进为4个空格，使用VS.NET的默认设置。
* 在代码中垂直对齐左括号和右括号。

if(x==0)

{

Response.Write("用户编号必须输入！");

}

**不允许以下情况：**

if(x==0) {

Response.Write("用户编号必须输入！");

}

或者：

if(x==0){ Response.Write("用户编号必须输入！");}

* 为了防止在阅读代码时不得不滚动源代码编辑器，每行代码或注释在1024\*800的显示频率下不得超过一显示屏
* 当一行被分为几行时，通过将串联运算符放在每一行的末尾而不是开头，清楚地表示没有后面的行是不完整的。
* 每一行上放置的语句避免超过一条。
* 在大多数运算符之前和之后使用空格，这样做时不会改变代码的意图却可以使代码容易阅读。

例：

int j = i + k;

**而不应写为**

int j=i+k;

* 将大的复杂代码节分为较小的、易于理解的模块。

## 2.2 命名指南

### 2.2.1 大写样式

使用下面的三种大写标识符约定。

**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：

**B**ack**C**olor

**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：

**b**ack**C**olor

**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**

可能还必须大写标识符以维持与现有非托管符号方案的兼容性，在该方案中所有大写字母经常用于枚举和常数值。一般情况下，在使用它们的程序集之外这些字符应当是不可见的。

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 | Pascal | **ValueChange** |
| 异常类 | Pascal | WebException 注意 总是以 Exception 后缀结尾。 |
| 只读的静态字段 | 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** |
| 参数 | Camel | **typeName** |
| 属性 | Pascal | **BackColor** |
| 变量 | Camel | **backColor** |
| 受保护的实例字段 | Camel | **redValue**  **注意** 很少使用。属性优于使用受保护的实例字段。 |
| 公共实例字段 | Pascal | **RedValue**  **注意** 很少使用。属性优于使用公共实例字段。 |

### 2.2.2 区分大小写

为了避免混淆和保证跨语言交互操作，请遵循有关区分大小写的使用的下列规则：

* 不要使用要求区分大小写的名称。对于区分大小写和不区分大小写的语言，组件都必须完全可以使用。不区分大小写的语言无法区分同一上下文中仅大小写不同的两个名称。因此，在创建的组件或类中必须避免这种情况。
* 不要创建仅是名称大小写有区别的两个命名空间。例如，不区分大小写的语言无法区分以下两个命名空间声明。

namespace ee.cummings;

namespace Ee.Cummings;

* 不要创建具有仅是大小写有区别的参数名称的函数。下面的示例是不正确的。

void MyFunction(string a, string A)

* 不要创建具有仅是大小写有区别的类型名称的命名空间。在下面的示例中，Point p 和 POINT p 是不适当的类型名称，原因是它们仅是大小写有区别。

System.Windows.Forms.Point p

System.Windows.Forms.POINT p

* 不要创建具有仅是大小写有区别的属性名称的类型。在下面的示例中，int Color 和 int COLOR 是不适当的属性名称，原因是它们仅是大小写有区别。

int Color {get, set}

int COLOR {get, set}

* 不要创建具有仅是大小写有区别的方法名称的类型。在下面的示例中，calculate 和 Calculate 是不适当的方法名称，原因是它们仅是大小写有区别。

void calculate()

void Calculate()

### 2.2.3 缩写

为了避免混淆和保证跨语言交互操作，请遵循有关区缩写的使用的下列规则：

* 不要将缩写或缩略形式用作标识符名称的组成部分。例如，使用 GetWindow，而不要使用 GetWin。
* 不要使用计算机领域中未被普遍接受的缩写。
* 在适当的时候，使用众所周知的缩写替换冗长的词组名称。例如，用 UI 作为 User Interface 的缩写，用 OLAP 作为 On-line Analytical Processing 的缩写。
* 在使用缩写时，对于超过两个字符长度的缩写请使用 Pascal 大小写或 Camel 大小写。例如，使用 HtmlButton 或 HTMLButton。但是，应当大写仅有两个字符的缩写，如，System.IO，而不是 System.Io。
* 不要在标识符或参数名称中使用缩写。如果必须使用缩写，对于由多于两个字符所组成的缩写请使用 Camel 大小写，虽然这和单词的标准缩写相冲突。

### 2.2.4 措词

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** | **ExternalSource** |
| **False** | **Finalize** | **Finally** | **Float** | **For** |
| **Friend** | **Function** | **Get** | **GetType** | **Goto** |
| **Handles** | **If** | **Implements** | **Imports** | **In** |
| **Inherits** | **Integer** | **Interface** | **Is** | **Let** |
| **Lib** | **Like** | **Long** | **Loop** | **Me** |
| **Mod** | **Module** | **MustInherit** | **MustOverride** | **MyBase** |
| **MyClass** | **Namespace** | **New** | **Next** | **Not** |
| **Nothing** | **NotInheritable** | **NotOverridable** | **Object** | **On** |
| **Option** | **Optional** | **Or** | **Overloads** | **Overridable** |
| **Overrides** | **ParamArray** | **Preserve** | **Private** | **Property** |
| **Protected** | **Public** | **RaiseEvent** | **ReadOnly** | **ReDim** |
| **Region** | **REM** | **RemoveHandler** | **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** |  |  |

### 2.2.5 避免类型名称混淆

不同的编程语言使用不同的术语标识基本托管类型。类库设计人员必须避免使用语言特定的术语。请遵循本节中描述的规则以避免类型名称混淆。

使用描述类型的含义的名称，而不是描述类型的名称。如果参数除了其类型之外没有任何语义含义，那么在这种罕见的情况下请使用一般性名称。例如，支持将各种数据类型写入到流中的类可以有以下方法。

[Visual Basic]

Sub Write(value As Double);

Sub Write(value As Single);

Sub Write(value As Long);

Sub Write(value As Integer);

Sub Write(value As Short);

[C#]

void Write(double value);

void Write(float value);

void Write(long value);

void Write(int value);

void Write(short value);

不要创建语言特定的方法名称，如下面的示例所示。

[Visual Basic]

Sub Write(longValue As Long);

Sub Write(integerValue As Integer);

Sub Write(shortValue As Short);

[C#]

void Write(double doubleValue);

void Write(float floatValue);

void Write(long longValue);

void Write(int intValue);

void Write(short shortValue);

如果有必要为每个基本数据类型创建唯一命名的方法，那么在这种极为罕见的情况下请使用通用类型名称。下表列出基本数据类型名称和它们的通用替换。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **C# 类型名称** | **Visual Basic 类型名称** | **JScript 类型名称** | **Visual C++ 类型名称** | **Ilasm.exe 表示形式** | **通用类型名称** |
| **sbyte** | **SByte** | **sByte** | **char** | **int8** | **SByte** |
| **byte** | **Byte** | **byte** | **unsigned char** | **unsigned int8** | **Byte** |
| **short** | **Short** | **short** | **short** | **int16** | **Int16** |
| **ushort** | **UInt16** | **ushort** | **unsigned short** | **unsigned int16** | **UInt16** |
| **int** | **Integer** | **int** | **int** | **int32** | **Int32** |
| **uint** | **UInt32** | **uint** | **unsigned int** | **unsigned int32** | **UInt32** |
| **long** | **Long** | **long** | **\_\_int64** | **int64** | **Int64** |
| **ulong** | **UInt64** | **ulong** | **unsigned \_\_int64** | **unsigned int64** | **UInt64** |
| **float** | **Single** | **float** | **float** | **float32** | **Single** |
| **double** | **Double** | **double** | **double** | **float64** | **Double** |
| **bool** | **Boolean** | **boolean** | **bool** | **bool** | **Boolean** |
| **char** | **Char** | **char** | **wchar\_t** | **char** | **Char** |
| **string** | **String** | **string** | **String** | **string** | **String** |
| **object** | **Object** | **object** | **Object** | **object** | **Object** |

例如，支持将从流读取各种数据类型的类可以有以下方法。

[Visual Basic]

[C#]

double ReadDouble();

float ReadSingle();

long ReadInt64();

int ReadInt32();

short ReadInt16();

前面的示例优于下面的语言特定的替换。

[Visual Basic]

[C#]

double ReadDouble();

float ReadFloat();

long ReadLong();

int ReadInt();

short ReadShort();

### 2.2.6 命名空间命名指南

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

CompanyName.TechnologyName[.Feature][.Design]

例如：

Microsoft.Media

Microsoft.Media.Design

给命名空间名称加上公司名称或者其他知名商标的前缀可以避免两个已经发布的命名空间名称相同的可能性。例如，Microsoft.Office 是由 Microsoft 提供的 Office Automation Classes 的一个适当的前缀。

在第二级分层名称上使用稳定的、公认的技术名称。将组织层次架构用作命名空间层次架构的基础。命名一个命名空间，该命名空间包含为具有 .Design 后缀的基命名空间提供设计时功能的类型。例如，System.Windows.Forms.Design 命名空间包含用于设计基于 System.Windows.Forms 的应用程序的设计器和相关的类。

嵌套的命名空间应当在包含它的命名空间中的类型上有依赖项。例如，System.Web.UI.Design 中的类依赖于 System.Web.UI 中的类。但是，**System.Web.UI** 中的类不依赖于 **System.UI.Design** 中的类。

应当对命名空间使用 Pascal 大小写，并用句点分隔逻辑组件，如 Microsoft.Office.PowerPoint 中所示。如果您的商标使用非传统的大小写，请遵循您的商标所定义的大小写，即使它与规定的 Pascal 大小写相背离。例如，命名空间 NeXT.WebObjects 和 ee.cummings 阐释了对于 Pascal 大小写规则的适当背离。

如果在语义上适当，使用复数命名空间名称。例如，使用 System.Collections而不是 System.Collection。此规则的例外是商标名称和缩写。例如，使用 System.IO而不是 System.IOs。

不要为命名空间和类使用相同的名称。例如，不要既提供 Debug 命名空间也提供 Debug 类。

最后，请注意命名空间名称不必非得与程序集名称相似。例如，如果命名程序集 MyCompany.MyTechnology.dll，它没有必要非得包含 MyCompany.MyTechnology 命名空间。

### 2.2.7 类命名指南

以下规则概述命名类的指南：

* 使用名词或名词短语命名类。
* 使用 Pascal 大小写。
* 少用缩写。
* 不要使用类型前缀，如在类名称上对类使用 C 前缀。例如，使用类名称 FileStream，而不是 CFileStream。
* 不要使用下划线字符 (\_)。
* 有时候需要提供以字母 I 开始的类名称，虽然该类不是接口。只要 I 是作为类名称组成部分的整个单词的第一个字母，这便是适当的。例如，类名称 IdentityStore 是适当的。
* 在适当的地方，使用复合单词命名派生的类。派生类名称的第二个部分应当是基类的名称。例如，ApplicationException 对于从名为 Exception 的类派生的类是适当的名称，原因是 ApplicationException 是一种 Exception。请在应用该规则时进行合理的判断。例如，Button 对于从 Control 派生的类是适当的名称。尽管按钮是一种控件，但是将 Control 作为类名称的一部分将使名称不必要地加长。

下面是正确命名的类的示例。

[Visual Basic]

Public Class **FileStream**

Public Class **Button**

Public Class **String**

[C#]

public class **FileStream**

public class **Button**

public class **String**

### 2.2.8 接口命名指南

以下规则概述接口的命名指南：

* 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。
* 使用 Pascal 大小写。
* 少用缩写。
* 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。
* 不要使用下划线字符 (\_)。

以下是正确命名的接口的示例。

[Visual Basic]

Public Interface IServiceProvider

Public Interface IFormatable

[C#]

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。

[Visual Basic]

Public Interface IComponent

' Implementation code goes here.

End Interface

Public Class Component

Implements IComponent

' Implementation code goes here.

End Class

[C#]

public interface IComponent

{

}

public class Component: IComponent

{

// Implementation code goes here.

}

### 2.2.9 枚举类型命名指南

枚举 (**Enum**) 值类型从 Enum 类继承。以下规则概述枚举的命名指南：

* 对于 **Enum** 类型和值名称使用 Pascal 大小写。
* 少用缩写。
* 不要在 **Enum** 类型名称上使用 Enum 后缀。
* 对大多数 **Enum** 类型使用单数名称，但是对作为位域的 **Enum** 类型使用复数名称。
* 总是将 **FlagsAttribute** 添加到位域 **Enum** 类型。

### 2.2.10 静态字段命名指南

以下规则概述静态字段的命名指南：

* 使用名词、名词短语或者名词的缩写命名静态字段。
* 使用 Pascal 大小写。
* 对静态字段名称使用匈牙利语表示法前缀。
* 建议尽可能使用静态属性而不是公共静态字段。

### 2.2.11 参数命名指南

以下规则概述参数的命名指南：

* 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。
* 对参数名称使用 Camel 大小写。
* 使用描述参数的含义的名称，而不要使用描述参数的类型的名称。开发工具将提供有关参数的类型的有意义的信息。因此，通过描述意义，可以更好地使用参数的名称。少用基于类型的参数名称，仅在适合使用它们的地方使用它们。
* 不要使用保留的参数。保留的参数是专用参数，如果需要，可以在未来的版本中公开它们。相反，如果在类库的未来版本中需要更多的数据，请为方法添加新的重载。
* 不要给参数名称加匈牙利语类型表示法的前缀。

以下是正确命名的参数的示例。

[Visual Basic]

GetType(**typeName** As String)As Type

Format(**format** As String, object [] **args**)As String

[C#]

Type GetType(string **typeName**)

string Format(string **format**, **args()** As object)

### 2.2.12 方法命名指南

以下规则概述方法的命名指南：

* 使用动词或动词短语命名方法。
* 使用 Pascal 大小写。

以下是正确命名的方法的实例。

RemoveAll()

GetCharArray()

Invoke()

### 2.2.13 属性命名指南

以下规则概述属性的命名指南：

* 使用名词或名词短语命名属性。
* 使用 Pascal 大小写。
* 不要使用匈牙利语表示法。
* 考虑用与属性的基础类型相同的名称创建属性。例如，如果声明名为 Color 的属性，则属性的类型同样应该是 Color。请参阅本主题中后面的示例。

以下代码示例阐释正确的属性命名。

[Visual Basic]

Public Class SampleClass

Public Property **BackColor** As Color

' Code for Get and Set accessors goes here.

End Property

End Class

[C#]

public class SampleClass

{

public Color **BackColor**

{

// Code for Get and Set accessors goes here.

}

}

以下代码示例阐释提供其名称与类型相同的属性。

[Visual Basic]

Public Enum Color

' Insert code for Enum here.

End Enum

Public Class Control

Public Property Color As **Color**

Get

' Insert code here.

End Get

Set

' Insert code here.

End Set

End Property

End Class

[C#]

public enum Color

{

// Insert code for Enum here.

}

public class Control

{

public **Color** Color

{

get {// Insert code here.}

set {// Insert code here.}

}

}

以下代码示例不正确，原因是 Color 属性是 Integer 类型的。

[Visual Basic]

Public Enum Color

' Insert code for Enum here.

End Enum

Public Class Control

Public Property Color As **Integer**

Get

' Insert code here.

End Get

Set

' Insert code here.

End Set

End Property

End Class

[C#]

public enum Color {// Insert code for Enum here.}

public class Control

{

public **int** Color

{

get {// Insert code here.}

set {// Insert code here.}

}

}

在不正确的示例中，不可能引用 Color 枚举的成员。Color.Xxx 将被解释为访问一个成员，该成员首先获取 **Color** 属性（在 Visual Basic 中为 **Integer** 类型，在 C# 中为 **int** 类型）的值，然后再访问该值的某个成员（该成员必须是 **System.Int32** 的实例成员）。

### 2.2.14 变量命名指南

以下规则概述变量的命名指南：

* 使用名词或名词短语命名属性。
* 使用Camel大小写。
* 不要使用匈牙利语表示法。

以下代码示例阐释正确的属性命名。

[Visual Basic]

Public Class SampleClass

Public Property **BackColor** As Color

' Code for Get and Set accessors goes here.

End Property

End Class

[C#]

public class SampleClass

{

private string **backColor** ;

}

### 2.2.16事件命名指南

以下规则概述事件的命名指南：

* 对事件处理程序名称使用 EventHandler 后缀。
* 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。
* 用 EventArgs 后缀命名事件参数类。
* 考虑用动词命名事件。
* 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，可以取消的 **Close** 事件应当具有 Closing 事件和 Closed 事件。不要使用 BeforeXxx/AfterXxx 命名模式。
* 不要在类型的事件声明上使用前缀或者后缀。例如，使用 Close，而不要使用 OnClose。
* 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**); [Visual Basic]

Public Delegate Sub **MouseEventHandler**(**sender** As Object, **e** As MouseEventArgs)

[C#]

x

以下示例阐释正确命名的事件参数类。

[Visual Basic]

Public Class MouseEventArgs

Inherits EventArgs

Dim x As Integer

Dim y As Integer

Public Sub New MouseEventArgs(x As Integer, y As Integer)

me.x = x

me.y = y

End Sub

Public Property X As Integer

Get

Return x

End Get

End Property

Public Property Y As Integer

Get

Return y

End Get

End Property

End Class

[C#]

public class MouseEventArgs : EventArgs

{

int x;

int y;

public MouseEventArgs(int x, int y)

{ this.x = x; this.y = y; }

public int X { get { return x; } }

public int Y { get { return y; } }

}

## 2.3    注释（Comment）规范

注释规范包括：模块（类）注释规范、类的属性、方法注释规范、代码间注释

### 2.3.1 模块（类）注释规范

模块开始必须以以下形式书写模块注释：

///<summary>

///模块编号：<模块编号，可以引用系统设计中的模块编号>

///作用：<对此类的描述，可以引用系统设计中的描述>

///作者：作者中文名

///编写日期：<模块创建日期，格式：YYYY-MM-DD>

///</summary>

如果模块有修改，则每次修改必须添加以下注释：

///<summary>

///Log编号：<Log编号,从1开始一次增加>

///修改描述：<对此修改的描述>

///作者：修改者中文名

///修改日期：<模块修改日期，格式：YYYY-MM-DD>

///</summary>

### 2.3.2 类属性注释规范

在类的属性必须以以下格式编写属性注释：

/// <summary>

///属性说明

/// </summary>

### 2.3.3 方法注释规范

在类的方法声明前必须以以下格式编写注释

/// <summary>

/// 说明：<对该方法的说明>

/// </summary>

/// <param name="<参数名称>"><参数说明></param>

/// <returns>

///<对方法返回值的说明，该说明必须明确说明返回的值代表什么含义>

/// </returns>

### 2.3.4 代码间注释规范

代码间注释分为单行注释和多行注释：

单行注释：

//<单行注释>

多行注释：

/\*多行注释1

多行注释2

多行注释3\*/

代码中遇到语句块时必须添加注释（if,for,foreach,……）,添加的注释必须能够说明此语句块的作用和实现手段（所用算法等等）。

### 2.3.5 Region的使用

命名空间的引用、私有成员字段、静态字段、常量、构造器方法、属性、私有方法、公有方法、公有静态方法。私有静态方法都使用region分割开来。

#region

private string backClor;

private string userName;

private int age;

#endregion

# 3 成功的模式

作为一个软件开发者，在你的所有品中含有一份有关标准的文档并不会自动地使你更加地有效率。要成功，你必须选择变得更有效率，这意味着你必须将这些标准有效地应用起来。

## 3.1 有效地使用这些标准

以下的建议将帮助你更有效地使用本文所描述的 C# 编程标准和指南：

1. **理解标准。**花些时间去理解为什么每个标准和指南会使开发效率提高。比如说，不要仅仅是因为指南中要求你才在一行仅声明一个局部变量，而应该是因为你明白它能使你的代码更易懂你才这样做。
2. **信任这些标准。**理解每个标准是一个开始，但你还需要信任这些标准。遵守标准不应仅仅是当你有时间才做的事，而你应该一直遵守，因为你相信这是最好的程序设计方法。
3. **当你写代码时就应该遵守标准，而不应是一个事后的想法。**加了注释的代码不仅在你写程序时，而且在你写完程序时，都更容易理解。在程序开发阶段和维护阶段，一致性地命名成员函数和字段都使工作更加容易。在开发和维护阶段，整洁的代码让工作更加容易。概括起来说，遵守标准将提高你开发过程中的生产率，并且使你的代码更易维护（因此也使维护者的生产率提高了）。如果从一开始你就写出整洁的代码，你将在撰写过程中受益。
4. **使它们成为你的质量保证的过程。**代码检查的一部分应该是确保源码遵守你的机构所采用的标准。将标准作为你训练和指导开发员更有效率的基础。

## 3.2 其它导向成功代码的因素

1. **面向人而不是面向机器编程。**你的开发努力的主要目的应该是你的代码易被其它人理解。如果没人能理解它，它就一点儿优点也没有。使用命名约定。注释代码。给代码分段。
2. **首先设计，然后编写代码。**你是否曾遇到过这样的情况：一些你的程序所倚靠的代码需要修改？可能是要传一个新的参数给一个成员函数，或者是需要将一个类拆成几个类。为了确信你的代码与被重新设置修改的代码还能一起工作，你必须做多大的额外工作呢？你有多么乐意？你是否曾经问过自己，为什么有些人在开始写代码时不先停下来考虑一下，以避免这一切的发生？他们为什么不首先设计程序呢？当然你做了。如果你在实际开始动手写代码之前花时间想清楚你打算怎样写你的代码，你很可能可以少花些时间编写它。此外，仅通过开始时就将它们想好，你将潜在地减少将来修改代码所带来的影响。
3. **一小步一小步地开发。**一小步一小步地开发，先写几个成员函数，检测它们，再多写几个，这样开发比一次性地写完所有代码然后修改它要有效得多。检测和修改十行代码远比检修一百行代码要容易得多。实际上可以很有把握地说，同样是编写、测试和修改 100 行代码，十行十行地做所花的时间少于一口气做 100 行所花的时间的一半。理由很简单。当测试你的代码并发现问题时，问题几乎总是在刚写完的新代码中，当然这假定在剩下的那些旧代码上开始写是很可靠的。在一小部分代码中寻找问题会比在一大段代码中找问题要快得多。通过一小步一小步逐步地开发，减少了查找错误所需的平均时间，这转而又减少了整个的开发时间。
4. **让代码简洁。**复杂的代码或许让人在智力上获得满足，但是如果别人读不懂， 那就不好了。如果要求某人，甚至是你，第一次修改一段复杂的代码以纠正其中的错误或对它进行增强，那么很可能代码会被重写。实际上，你很可能已经因为代码难懂而重写过别人的代码。当你在重写代码时，你是怎样认为代码的最初开发者的呢？你认为那个人是天才还是怪物？写出那种后来要被重写的代码没什么可骄傲的，所以应该遵循 KISS 法则：要使代码简单直白。
5. **学习常用的模式、反模式和代码模式。**有大量的分析、设计和处理的模式和反模式以及编程代码模式供你提高开发效率。

你的开发工作的主要目的应该是你的代码易被其它人理解。如果没人能理解它，它就一点优点也没有。使用命名约定。注释代码。给它分段。