C#编程风格

目录

第一章

一般原则

编写性能良好的软件固然重要，但专业的开发者还应注意其他许多问题。好多的软件能完成任务，但以一致风格写成的卓越的软件则明晰、健壮、可维护性、可支持且可扩展。

1． 保持原有风格

修改别人编写的软件时，应遵循原始代码的风格。修改时不要引入新风格，也不要仅为了吻合新风格而重写就旧软件。在一个源代码文件中存在多种不同风格会使代码更难读懂。为修改风格而重写代码将产生本不可避免的缺陷，增加软件成本。

2． 坚持最小惊奇原则

最小惊奇原则建议你避免做出可能让其他软件开发人员吃惊的事情。这意味着，软件所展示的互动及行为方式必须可预料并保持一致。如果不是这样，文档就必须清晰地指出所有非通常的用法或行为。

为了降低其他人在使用你的软件时遭遇惊奇的可能性，你应当在软件的设计、实现、打包和文档中强调以下原则：

简单性 用简单的类和简单的方法满足用户期望。

清晰性 确保么个类、接口、方法、变量和对象都有清晰地目的。
阐明何时、何处、如何使用它们。

完整性 提供任一可能的用户期望找到和使用的最小功能。创建完成整的文档，描述所有特性和功能。

一致性 相似实体的外观和行为应该相同，不同实体的外观和行为应该不同。
应该尽可能制定并遵守相关标准。

健壮性 对软件中可能出现的错误和异常做出预测，并将解决方法记入文档。
不要隐藏错误，也不要等着用户去发现错误。

3． 第一次就作对

对所有代码实施这条规则，不仅限于正式产品的代码。原型或实验性代码多半会用到最终产品中，所以你改洞悉先机。即便代码永远不会采用到正式产品中，别人也可能读到它。任何阅读你的代码的人，都会从你的专业性和贯彻这些规则的先见之明中获益匪浅。

4． 记录所有非规范行为

没有十全十美、普适一切的标准。有时你会有偏离某条既定标准的需要。无人如何也要努力保持清晰和一致。

在决定忽略某条规则之前，你应该先确信自己了解该条规则存在之理由，以及不采用该规则会引起的后果。如果你决定要违反某条规则，记下这么做的原因。

这是第一守则。

5. 考虑采用代码检查工具强制遵循编码标准

可采用源代码分析工具检查嗲吗是否符合编号标准和最佳实践。例如，FxCop 是一种流行的.NET 代码分析工具，其利用反射技术、MSIL 解析和调用图分析来检查代码是否符合.NET 框架设计的指导原则。FxCop 可扩展，所以能加入你所在组织的特殊编码标准。

第 2 章

格式

2.1 空白

6. 使用空白

空白时页面上没有可见字符的区域。只有少量空白的代码很难阅读和理解，故应多用空白来凸显方法、注释、代码块和表达式。

使用单个空格分隔符控制流程语句中的关键字、圆括号以及括号：

```
for（…）
{
   //…….
}
while(…)
{
   //…
}
do
{
   //….
}while(…)

switch(…)
{
  //…
}

if(…
{
```

```
   //…
}
else if(…)
{
 //…
}
else
{
 //…
}

try
{
   //…
}
catch(Exception)
{
 //...
}
finally
{
 //…
}
```

在二元操作符左右分别放一个空格，“.”操作符除外：

```
double length=Math.Sqrt(x * x+y * y);
double xNorm=(length > 0.0) ? (x / length) : x;
double currentTemperature=engineBlock.Temperatuer;
```

在逗号和分号后面放一个空格：

```
Vector normalizedVector=NormalizeVector(x, y, z);

for(int i=0; i<100; i++)
{
 //…
}
```

在方法声明中，括号内的参数列表的各项间加上空格：

```
Vector NormalizeVector(double x, double y, double z)
{
 //…
}
```

使用空行分隔方法体中的逻辑块：

```
public void HandleMessage(Message message)
{
    string content=message.ReadContent();
    switch(message.ErrorLevel)
```

```
        {
            case ErrorLevel.Warning:
            //…进行一些操作…
            break;
            case ErrorLevel.Severe:
            //…进行一些操作…
            break;
            default:
            //…进行一些操作…
            break;


        }

}
```

使用空行隔开类中每个方法定义：

```
public void SendEmail()
{
 //…
}

public void SendFax()
{
 //…
}
```

7.使用缩进的语句块

改进代码可读性的方法之一是将多条独立的语句组合为语句块，统一缩进，使之与其他代码区分开。

如果使用集成开发环境（如 Visual Stodio）生成代码应确认组内每个成员均采用相同的缩进规则。如果手工编写代码，使用两个空格的缩进方式，在确保可读性的同时也不至于浪费过多空间：

```
void   PesterCustomer(Customer customer)
{
   customer.SendLetter();
   if(customer.HasEmailAddress())
   {
       customer.SendEmail();
       if(customer.IsForgetful())
       {
          Customer.ScheduleReminderEmail();
       }
   }
   if(customer.HasFaxNumber())
   {
```

```
        Customer.SendFox();
    }
}
```

如果你负责管理开发团队，别让团队中的每个开发人员自行选择缩进量和缩进风格，应设定标准缩进策略，确保所有人以此行事。

8.缩进标记后的语句

除了缩进语句块内容外，还应缩进标记后的语句，让标记更易于阅读：

```
void DoSomethingUseful(int arg)
{
  loop:
      for(int index=0;index<=arg; index++)
   {
      switch(index)
      {
          case 0:
              //…
              Break;//退出 switch 语句
           defualt:
              //…
              Break;//退出 switch 语句

      }
   }
}
```

9.不用“硬”制表符

许多开发人员使用制表符（Tab）做缩进和代码对其，而没有意识到在不同环境中制表符会有不同解释。在最初使用的编辑环境中看似正常的格式，迁移到以不同方式解释制表符的其他环境，或者由其他开发人员查看时，可能会显得毫无格式或极不可读。

要避免这种情况发生必须总是使用空格符，而不用制表符缩进和对齐源代码。可以用空格键代替制表符键，也可以配置编辑器，使之自动用空格符代替制表符。有些编辑器也具有“智能”缩进功能。如果编辑器使用制表符做自动缩进，则应该禁止智能缩进功能。

10.切分长语句为多行语句

基于视窗的现代编辑器能够通过水平滚动条轻易处理长语句，但如果能在一屏内显示而不用滚动的话，开发人员阅读代码就会更有效率、更不不易出错。另外，在超出最长打印行长度时，打印机会自动截断、换行或在另一张纸上打印代码。要保证代码在打印出来后任可阅读，应将代码行限制在打印环境支持的最大长度内，通常会是 80~132 个字符。

首先，如果可能超出允许的最大长度，则不要把多个语句放在一行以内。如果两个表达式语句放在同一行：

```
double x=random.NextDouble();double y=random.NextDouble(); //太长！
```

就应该分成两行：

```
double x=random.NextDouble();
double y=random.NextDouble();
```

其次，如果行中包括复杂的表达式，导致一行过长：

```
Double distance=Math.Sqrt(Math.Pow((x1-x2),2.0) +
```

Math.Pow((y1-y2),2.0)+Math.Pow(z1-z2),2.0));

//太长

则将表达式清晰地分割为数个表达式。在单独一个代码行中使用临时变量保存每个字表达式的运算结果：

double dx=Math.Pow((x1-x2),2.0);

double dy=Math.Pow((y1-y2),2.0);

double dz=Maht.Pow(z1-z2),2.0);

double distance=Math.Sqrt(dx+dy-dz);

最后，如果长代码不能按照上述指导原则缩短，则采取以下规则切分、换行、和缩进代码。

第一步

如果行中最顶层表达式包含一个多个逗号：

double value=Foo(Math.Pow(x,2.0),Math.Pow(y,2.0),Math.Pow(z,2.0));

则在每个逗号后换行。将每个逗号后的表达式与该逗号前面表达式的第一个字符对齐：

```
double value=Foo(Math.Pow(x,2.0),
                 Math.Pow(y,2.0),
                 Math.Pow(z,2.0));
```

第二步

如果行中最顶层表达式不包含逗号：

return person1.Name==person2.Name && person1.Address==person2.Addess && person1.Phone=person2.Phone; //太长

则在最低优先级的操作符之前换行；后者，如果有多个优先级相同的操作符，像这样对齐：

```
return person1.Name==person2.Name &&
       person1.Address==person2.Address &&
       person1.Phone==person2.Phone ;
```

第三步

按需要重复进行第一步和第二步，直至从原始表达式语句转换而来的每一行代码都少于允许的最大长度。

2.2 花括号

1.1 按同一风格放置花括号

放置语句块左花括号的方式有两种，可以把左花括号放在控制语句块入口代码行的行尾，也可以放在下一行，使之和上行首字符对齐。应当始终把右花括号单放一行，并使之与包含左花括号那行对齐：

```
void SameLine(){

}
void NextLine()
{
}
```

虽然许多程序员混用这两种方式，但你的组织应该二中选一，并且坚持使用。

本书适用第二种放置花括号的方式，下例展示了如果在不同 c#定义和控制结构中应

用此规则。

类声明

```
namespace MyOrganization
{
    public class Outer
    {
        public Outer()
        {
          //…
        }

       public class Inner
    {
          public Inner()
          {
            //…
          }
    }
}
```

方法声明

```
  public void Display()
  {
   //…
  }
```

for 循环语句

```
  for(int i=0;i<=j; i++)
  {
     //…
  }
```

if 和 else 语句

```
if (j<0)
{
   //…
}
else if(j>0)
{
  //…
}
else
{
  //…
}
```

try 块和 catch 块

```
try
{
 //…
}
catch(…)
{
   //…
}
finally
{
//…
}
```

switch 语句

```
switch （value）
{
   case 0:
      //…
      break;
   default:
      //…
      break;
}
```

while 语句

```
while( ++k <=j)
  {
      //…
  }
```

do-while 语句

```
do
{
 //…
}while(++k<=j);
```

12.在流程控制结构中始终使用语句块

组合语句，或称语句块，提供了一种将一系列语句看作单个组合语句的机制。不同的流程控制语句，如 if…else、for、while 和 do…whle 都能根据条件执行单个语句或语句块。

如果要在流程控制语句中根据条件执行多个语句，就必须使用块。在嵌套使用 if..else 语句时，也可能需要使用块，避免常说的“悬挂 else 问题”这种可能的歧义情况出现。

```
if (x>=0)
      If(x>0) Positive();
else //Oops！匹配最近的 if!
      NegativeX();
```

下面的方式看似累赘，但逻辑流程却清晰易于维护：

```
if(x>=0)
{
      if(x>0)
      {
          PositiveX();
      }
      else
      {
        //什么都不做…..
      }
}
else
{
    NegativeX(); //这才是我们想要的！
}
```

使用语句块更便于往既有流程控制结构中添加新语句：

```
for( int i=n;i>=0;i++)
    for(int j=0;j=n;j--)
        Foo(i,j);
        Goo(i,j);//为何 i 和 j 超出了作用域？
for( int i=n;i>=0;i++)
{
    for(int j=0;j=n;j--)
     {
        Foo(i,j);
        Goo(i,j);//这才是我们想要的！
     }
}
```

2.3 类的组织

13.在源文件开始分组放置 using 指示符

如果源文件中使用了 using 指示符，在文件开始分组放置这些指示符。先依字母顺序列出系统定义的命名空间，后面加上一个空行；然后依字母顺序列出第三方命名空间，后面加上一个空行；最后再依字母顺序列出用户定义的命名空间：

```
using System;
using System.IO;
using System.Xml;

using CenterSpace.Nmath.Core;
using CenterSpace.Nmath.Staus;

using MyOrganization.BusinessUtilities;
```

14.将源代码组织到不同区域中

15.依可访问性排列类元素
16.单独声明每个变量和特性

第三章

命名

一致地使用一种命名约定，能给那些阅读代码的人留下极有价值的直观线索。本章给出一些命名约定，可以让代码更为可读。

3.1 一般原则

17.使用有意义的名称

使用对读代码的人始终有意义的名称。使用有意义的单词创建名称。使用可被未来读者所理解的一致的语言。避免使用单个字符或太一般性的名称，那样对定义所命名的实体无所助益。

在下面的代码中，变量 a 和常量 65 的目的不清晰:

```
if(a<65)
{
        // 'a'的属性是什么?
        //这里 j 正在计算什么?
 }
  else
 {
    y=0;
 }
```

改用有意义的名称后，下面这段代码更易于理解:

```
if(age<RetirementAge)
{
    yearsToRetirement=RetirementAge – age;
}
Else
{
   yearsToRetirement=0;
}
```

本条规则的唯一例外是，当足以从上下文中判断出其目的时，可用简约方式命名临时变量，例如在循环内部用作计数器或索引的变散（讲规则 47）：

```
for(int i=0;i<numberOfStudents;++i)
{
    EncrollStudent(i);
}
```

18.根据含意而非类型来命名

类型信息一般可从其用法利使用场景来推断出来。有意义的名称才有用。例如，使用 Customer 二而不用 CustomerClass。

本条规则的一个例外是 GUI 控件的命名。一有时，以名称来区分 GUI 元素的类型非常有用。例如，区分 customerNameLabel(窗体上的标记控件)和 customerNameTextbox(窗体上的文本框控件)。

19.使用熟悉的名称

使用目标领域术语表中存在的单词。如果用户喜欢用“customer(顾客)“，则用 Custoemer 命名类，而不用 Client(客户)。许多开发人员会错误地在目标行业或锁域中已存在常用术语时新创术语。

20.不要用大小写来区分名称

编译器能够区别仅大小写不同的名称，但人可能注意不到其差异。
这等同于名字隐藏（name hiding）。

例如，如果已经有名为 XMLStream 的类存在，就别将其他类命名为 XmlStrem。假若两个类在同一作用城内出现，从人阅读和理解代码的角度来看，其中一个就会把另一个隐藏掉。

21.避免使用过长的名称

对象的名称应足以描述其目的。如果类、接口、变量或方法的名称过长，则孩实体可能企图实现太多功能。

不要简单地用包含更少意义的名称重命名实体，首先要考虑其设计或目的二通过实体的重构可以得到功能更集中，用更简洁的名称即可涵括其意义的新类、新接口、新方法或新变量。

22.加上元音一使用完整的单词

切勿通过去除元音来缩短名称，这种做法降低了代码的可读性，如果有多个本来其有实际意义的名称被缩减为同一类形式，就会产生歧义。

下面的代码不好:

```
public class Msg
{
    public Msg AppendSig(string sig)
    {
        ….
    }
}
```

最好改成:

```
public class Message
```

```
{
    public Message AppendSignature(string signature)
    {
        ….
    }
}
```

这样，偶然读到代码的人才能看懂后而的实现。

如果只为了缩短名称而去除元音，那么需要考虑一下原来的名称是否合适（见规则 21）。

3.2 缩略形式

23.除非全称太长，否到不用缩路形式

坚决不用不必要的缩略词迷惑人。没有必要用 Grph 取代 Graphical，但 GuiListener 比 GraphicalUserInterfaceListener 要好。如果必须使用缩略词，就用广为使用和接受的缩略词。

24.像普通词一样书写缩略词

如果缩略词是类型或常量名称的首个单词，只大写缩略词的第一个字母。

当大写字母用作隔离符时，这种写法能消除名称称中的含混之处。如果缩略词后面还有一个缩略词时，这一点尤为重要:

XMLString > XmlString

LoadXMLDocument() >LoadXmlDocument()

条件编译指示符的名称中的缩略词不适用这一规则，因为这类名称只能用大写字母写出(见规则 25):

[conditional(GUI)]

本规则不适用于在变里或参数名开始处的缩略词，因为这些名称总以小写字母开头：

Document xmlDocument ;

3.3 预处理器符号

25.用大写字母和下划线表示预处理器符号

用大写字母表示预处理器符号，使之与用 C#语法定义的符号区分开来：

#define EVAL_VERSION

26.给预处理器名称添加唯一前缀

给预处理器名称添加前缀，避免与用户定义或第三方软件中的预处理器名称相冲突。建议使用你所在组织名称的缩略形式，可自行选择是否加上产品名称的缩略形式，例如 ACME_DB_ USER。

3.4 类型和常量

27．使用 Pascal 写法给命名空间、类、结构、属性、枚举、常最及函数命名

每个单词的首字母大写，区分名称中每个独立的单词。第一个字母大写提供了一种使其与参数或变量相区分的机制〔见规则 43) 。

```
public enum BackgroundColor
{
   None= 0
   Red=1，
   Green=2.
   Bule = 3
};
const int FixedWidth=10;
class BankAccount
{
    //…
}

Public double CalculatePercentitle(double percent)
{
   //…
}
```

28.使用名词命名复合类型

应该用名词来命名定义了对象或其他事物的类、结构或属性：

```
public class Customer
{
    public string Name
    {
        get
        {
            Return name_;
        }

    }
}
```

29．用复数形式书写集合名称

对象集合的名称应该有能反映集合中对象的类型的复数形式，这样阅读你的代码的人才能区分表示多个值的变量和表示单个值的变量：

```
List<Shape> shapes =…
Shape shape=shapes[index];
```

30．给抽象基类型加上“Base“后缀

清晰定义的基类更易于管理。

```
Public abstract class AccountBase
Public class PersonalAccount ： AccountBase
Public class BusinessAccount ： AccountBase
```

31.给实现一种设计模式的类添加模式名称

例如，名为 MessageFactory 的类对熟悉设计模式的开发人员来说有某种特定含义。

32.使用单个大写字母命名泛型参数

```
public static List<T> Unquify(List<T>)
{
   //…
}
```

3.5 枚举

33.用单数形式为枚举命名

枚举类型通常用于相互独立的元素组成的列表，应使用单数形式：

```
pubilc    enum SortOrder
```

34.用复数形式给位域命名

位域通常用于可以组合形式出现的元素的列表，应使用复数形式：

```
[Flags]
public enum PrintSettings
{
   Draft=0,
   Duplex=1.
   Color=2
};
```

参见规则 136。

3.6 接口

35.用大写字母“|”作为接口名称的前缀

如果真的某个类是继承自父类还是实现一个接口，这就做会比较方便。

```
public class Worker:Iworkable
```

36.使用名词或形容词给接口命名

接口声明了对象提供的服务，或描述了对象的能力。

使用名词给用于声明服务的接口命名：

```
public interface ImessageListener
{
      public void MessageReceived(Message message);
}
```

使用形容词给用于描述能力的接口命名。大多数描述能力的接口使用在动词后面加上

一 able 或一 ible 后缀创造的形容词来命名：

```
public interface Ireversible
{
    public Icollection Reverse();
}
```

3.7 属性

37. 依取值或赋值项给属性命名

例如，对于一个取得过期日的属性：

```
public Date ExpirationDate
{
    get
    {
        return expirationDate_;
    }
}
```

38.避免冗长的属性名称

使用

```
public ICollection Customers
```

不用

```
public   ICollection CustomerCollection
```

39.用能体现布尔值特性的名称给布尔型属性命名

如果某个属性返回一个布尔值，给其名称加上“is”、“has”或“are”前缀。

```
public bool IsGood
public bool HasCompleted
```

3.8 方法

40.使角 Pascal 写法为方法命名

函数名中第一个单词首字母大写，后面每个单词首字母大写，可区分开名称中每个单词。

```
public class DataManipulator
{
    public void ComputeStatistics(DoubleMatrix m);
}
```

41.用动词命名方法

方法通常定义动作，应该用动词来描述：

```
public class   Account
{
    public void Withdraw(double amount)
    {
        balance_-=amount;
```

```
    }
    public    void Deposit(double amount)
    {
        Withdraw(-amount);
    }
}
```

42.避免冗长的方法名

在名为 Book 的类中，使用

public void Open()

不用

public void OpenBook()

3.9 变量和参数

43.使用骆驼写法给变量和方法参数命名

变量名中第一个单词的首字母小写，后面每个单词的首字母大写，区分开名称中每个单词。

```
public class Customer
{
 public    string    firstName_;
 public    string    1astName_;
 public    string ToString()
 {
   return lastName_+”,” + firstName_;
 }
}
```

第一个小写的字母把变量与常量区分开来：

```
Const int ConstantValue =10;
```

44.用名词命名变量

变量代指对象或类似事物，应该用名词描述。复数形式的变量名表示集合：

```
public class Customer
{
    private    string    billingAddress_;
    private    string     ShippingAddress_;
    private    string     daytimePhone_;
    private    Order[]    openOrders_;
  }
```

45．给成员变量名称加上前缀或后缀，使之与其他变量区分开

这样能减少名称隐藏(name-hiding)的可能性，提高代码可读性。选择前缀或后缀形式，并在产品中贯彻使用。如果是扩展或修改第三方框架，则遵循原框架的命名方式。例如，下面的代码在名称后面加下划线前缀，以指示出成员字段：

```
public class Customer
{
    private string homePhone_;
```

```
        private string workPhone_;
    }
```

46．依所赋值的字段名称给构造函数和属性参数命名

依成员变量名称给方法参数命名，向阅读者指出参数赋值给成员：

```
public class Customer
{
    private Customer(string name)
    {
       name_=name;
    }
    public string Name
    {
       get
       {
           return name_;
       }
    }
}
```

47.用一系列标准名称为“一次性“变量和参数命名

应使用足具描述性的名称给大多数变量命名，但在 C#代码中频繁出现的许多变量类型拥有共同的短名称，你也可以使用这些名称。下表列出一些例子：

循环指示符（通常是 int）	i,j,k
Object	o
String	s
Exception	e 或 ex
EventArgs	ea
Graphics	g

3.10 特性

48．给自定义特性实现加上“Attribute“后缀

在 C#中，给特性类名称加上“Attribute”后缀是标准做法。

```
public class   MyFavoriteAttribute:Attribute
```

在后面的代码中这样用：

```
[MyFavorite]
pubic class   Facilitator
```

3.11　命名空间

49．用机构名称给根命名空间命名，加上项目、产品或小组名来缩小范围

```
namespace Company.Group.Project
```

```
{
  //…
}
```

3.12 事件处理

50．使用适当的名称清晰区分事件处理部分

事件类的名称应包括对动作的描述，如 MessageReceived。

引发事件的类的名称该是名词，如 Messenger。

为事件定义数据的类的名称应该类 MessageReceivedEventArgs。

MessageReceivedEventHandler 是委托。

MessageReceiver 是一个类，有一个名为 OnMessageReceived()的方法，该方法处理 MessageReceived 事件。

3.13 异常

51．给自定义异常类型添加“Exception”后缀

异常当然要特殊一些。用易于识记的名称命名：

```
public BadArgumentException:ApplicationException
{
    //…
}
```

第四章

文档

开发人员经常忘记软件的目的是满足最终用户的需求。他们常常关注解决方案本身，而不普于指导他人如何使用解决方案。

良好的软件文档不但要告诉别人如何使用软件，而且还为帮助你开发软件和以后负责维护、增强软件的工程师提供了接口和行为的规格说明。尽管你应当一直尽力编写无需说明的软件。但最终用户也许不能访问源代码，而且也总有编程语言无法表达的关于用法和行为的重要信息。

和优良的设计和实现一样，良好的文档也是专业程序员的标志。

4.1 一般原则

52.为使用接口的人编写软件接口文档

编写代码中公共接口的文档，可让别人正确、高效地理解和使用接口。
编写文档注释的目的是在服务器的提供者(supplier)和客户(clicnt)之间定义一种编程契约（programming contract）。与方法相关的文档应描述该方法的调用者可依赖的行为的诸多方面，而不应试图描述其实现

53.为维护者编写代码

编写代码实现部分的文档，好让别人可以维护和增强代码。总是假定有完全不熟悉你的代码的人要阅读和理解你的代码。

54.保持注释和代码同步

如果代码和注释不一致，那么可能两者青错。

— Norm Schryes，贝尔实验室

. 修改代码时，也要确保更新相关注释。代码和文档一起构成软件产品，应对其同等重视。

55.尽早编写软件元素的文档

在实现之前或实现之中编写每个软件元素的文档，勿拖延至软件将近完成才编写。因为对代码太过熟悉或太过厌烦，在项目末尾做的文档往往缺少细节。

如果在实现之前编写软件参考文档，则可以使用用该文档为受托实现该软件的开发人员定义需求。

56．考虑全世界读者

鉴于软件市场的全球性，应假设你的软件将在全世界使用，考虑全世界读者，意味着让与你不同母语的读者能够轻易阅读和理解文档。而且当软件要为其他目标市场做本地化时，也易于璐译为其他语言。

要简洁，使用简单语法，并且术语标准一致。尽量少用缩略语、避免使用本地惯用语。尽可能请非母语人士审阅文档。使用恰当的货币及计算单位。避免使用攻击性语言。

57．在每个文件的开始添加版权、授权许可和作者信息

在每个源文件开始增加头注释块，包括版权或授权许可说明。多数计算机软件包含商业秘密，所以要保护你的工作免受未经授权的复制。即便源代码将自由分发，也应说明代码仲用、重新分发或修改的限制条件。

如果决定在头注释块中添加版本号和修改日期，应建立一种使该信息保持最新及同步

的机制，如配置管理系统扩展出来的关键字，或者容易找到和更换的标准字符串。

4. 2 API

58．尽量使用 C#语言内建的文档机制

C#允许在源代码中嵌入 XML 注释，然后自动生成 API 文档。自动生成的文档往往比在外部手工维护的文档更精确、更完整、更及时反应当前状态。

尽管编译器能处理任何有效的 XML 标签，微软还是提供了推荐使用 XML 元素列表。为所有公共（public）、保护（protected）和内部(internal)声明添加 XML 注释。总是使用<summary>标签，在适当的位置使用<param>、<return>和<exception>标签。在属性取值和赋值的数据处添加<value>标签。使用<example>段演示一般和适宜的用法，使用<seeecref=”” />及<seealso cref=”” />标签标示交义引用。

C#编译器将源代码中的文档注释导出为 XML 文件，但你要自己做最终处理。一般的做法是采用 XSLT 把 XML 转为帮助文件。Ndoc 是一种好用的开源工具，可以处理.NET 程序集（assembly）和 XML 文档注释文件，生成数据通用形式的 API 文档，包括 MSDN 式 HTML 帮助格式（.chm）、Visual Studio .NET 帮助格式（HTML Help2）、MSDN 在线 Web 页面以及 JavaDoc 式 Web 页面。

59.编写重要前置条件、后置条件和不变条件的文档

生成数种通用形式的 API 文档

前置条件(prcondition)是一种条件判断，如果某个方法运行正常，则它在改方法开始之前保持为真值。典型的前置条件可以限制方法参数的可接受值范围。

后置条件(postcondition)是一种条件判断，如果某个方法运行正常，则它在该方法结束之后保持为真值。典型的后置条件描述一个对象的状态，改对象应来自给定初始状态及调用参数来调用的方法。

不变条件（invariant）是一种条件判断，对于某种对象总保持为真值。典型的不变条件可能是限制表示在 1 到 12 之间的当前月份的一个整型字段。

编写前置条件、后置条件、不变条件的文档很重要，因为这些文档定义了用户与类交互的前提。例如，如果方法申请了必须由调用者释放的资源，则应在文档中写明。

参见规则 75 和规则 146。

60.编写线程同步需求

如果类或方法有可能在多线程环境中使用，.则应该在文档中写明线程安全级别。说明对象是否能在线程间共享，如果可以，则说明是否需要外部同步确保串行访问。完全线程安全

的对象或方法使用其自身的同步机制，在多线程中保护内部状态。

参见规则 82。

61.编写已知缺陷和不足

标识并描述与类或方法有关的突出问题。指出存在的替代方案或解决方法。如果有可能，则说明何时可以解决该问题。

虽然没人愿意公开自己代码中的问题，但你的同事和客户将乐见这些信息。这些信息让他们有机会加以解决，或隔离问题以将未来修改的影响降到最低。

62.使用主动语态描述操作者，使用被动语态描述动作

在用英语写文章时，应多使用主动语态，少用被动语态。不过，对于技术文档则不尽然。对于提供用法指导的文档尤为如此。

当操作者在场景中较为重要时，使用主动语态。

这样写：

A Time object represents a point in time..(Time 对象表示一个时刻。)

Use Time() to get the current system time.(使用 Time()方法获取当前系统时间。)

A point in time is represented by a Time object .(时刻由 Time 对象象表示。〕

The system time is returned by Time().(系统时间由 Time()方法返回。)

当父对象被操作，或操作比操作者重要时，使用被动语态：

The Guard object is destroyed upon exit from the enclosing block scope.

(Guard 对象在从封闭块中退出时被销毁。)

The Reset () method must be callled prior to invoking any other method.(Reset()方法必须在其他方法之前被调用。)

你可能会在对软件元素的单句式概述中忽略主语：

[Time()] Returns the Current time.

然而，在元素描述的主体部分不要使用这种语法方式。写出完整的句子，指明元素名称，或者以“this”代替名称：

The Time() method ignores time zones(Time()方法忽略时区。)

This method ignores time zones.(This 方法忽略时区。)

63.当指称当前类的实例时，使用“this”、不用“the”

当描述方法的目的或行为时，使用 this 而不用 the 来指称定义该方法的类的实例对象：

```
///<summary>
/// Reurns a formatted string representation   of this object .(返回这个对象的格式化字符串表示。)
//</summay>
public   override string ToString();
```

4.3 内部代码

64.只在需要帮助别人理解代码的时候才添加内部注释

避免强加无用或无关的注释：

```
public   int   OccurrencesOf(Object item)
{
    //结果变得很简单
    //这种实现比我想象的要简单一些.
    return (Find(item) != null ? 1: 0 ;
}
```

只在有助于别人理解代码运行机制时才添力注释：

```
public   int   OccurrencesOf(Object item)
{
  //不允许出现重复，所以行得通：
  return (Find(item) != null ? 1: 0 ;
}
```

如果内部注释未增加任何价值，最好让代码自己说话。

65.解释代码为什么要这么做

好的代码本身也是一种文档。其他开发者能从编写良好的代码中看出其功用。然而，他或她可能不知道为什么代码要这么做。

下面代码中的注释给出的附加信息很少：

```
//用向量长度去除向量的各个分量。
double   length =0;
for(int i=0; i<vectorCount; i++)
{
    double x= vector[i].x;
    double y=vector[i].y;
    length=Math.Sqrt( x* x, y* y);
    vector[i].x=vector[i].x./ length ;
    vector[i].y=vector[i].y / length ;
}
```

看完这段代码后，，普通的开发者大概还是不明白这段代码为什么要做。可以用内部注释提供这一信息：

```
//规范化每个向量，产生一个可在几何计算和转换中用作方向向量的单位向量。.
double    length =0;
for(int i=0; i<vectorCount; i++)
{
    double x= vector[i].x;
    double y=vector[i].y;
    length=Math.Sqrt( x* x, y* y);

    vector[i].x=vector[i].x./ length ;
    vector[i].y=vector[i].y / length ;
}
```

66．避免使用 C 风格的注释块

C#同时提供用/*和*/分隔的 C#风格注释块和用//指明的单行注释。避免使用 C 风格的注释块。如果注释块太大，在某些按行注释的编辑器里面就会看不到。Visual Studia.NET 中的“注释块/取消注释”工具，可以方便地使用多个单行注释将死代码注释掉。

67.使用单行注释描述实现细节

在以下场合使用一个或多个单行注释说明：
特定变量或表达式的月的。
实现层面的设计决定。
复杂算法的来源资料。
缺陷的修正或变通方法。
以后可能做优化或加工的代码。
任何所知的问题、局限或缺陷。

通过让代码更具可读性，尽可能减少内嵌注释。不要添加重复描述代码行为的注释。只在提供有用信息时添加注释。

```
//在打印总额前计算折扣
If (invoiceTotal > DiscountThreshold )
{
  //采用了硬编码的折扣率，
  //因为所有客户都用同一折扣率。
  //如果有客户需要不同折扣率，
  //或者需要用到多个折扣率，
  //我们就要用变量替换这个常量。
}
```

68.避免使用行末注释

避免在代码行末添加注释。这类注释很容易干扰代码的视觉结构。修改注释所在的代码行可能会将注释推到文本编辑器右侧不可见的地方。有些程序员为了让行末注释看起来整齐，会左对齐这些行末注释。要维护这种外观，每次修改代码时就要重新做对齐处理。

将一行的注释直接放在所属代码的下面另起一行。本规则亦有例外，例如注释标明修改的代码行，或者为了支持搜索一替换操作而放置注释。行末注释也可用来描述简单的局部变量，或标出多重嵌套的控制结(见下一条规则)。

69.在多重嵌套控制结构中标出结束花括号

通常应该避免创建过深的多重嵌套控制结构，但是也可以采用在每个结构的结束花括号后而添加行末注释的方式，改善这种代码的可读性：

```
for（int i=0; …）
{
    for(int j=0;…)
    {
        while( !done)
        {
            if(…)
            {
                switch(…)
                {
                    //…
                }//结束 switch
            }//结束 if
        }//结束 while
    }//结束  for j
}结束  for i
```

如果出现过深的嵌套控制结构，建议进行重构。

70.使用关键词标出待完成工作、未解决问题、错误和缺陷修正

制订一系列关键词或标签，你和其他开发者可以用这些标签创建特殊注释，标出和定位代码中的重要部分。在标出己知的未完成、不正确或留待以后检查的代码时，这些标志特别有用。

用于标出未解决问题的关键词应该包括提出问题的人的名字或简称，及发现或解决问题的日期。选用不太可能在代码其他部分出现的关键词字符串：

```
// **FIX** --添加刷新缓冲区代码。
// TODO John Smith 5/5/2005
//这些代码无法处理输入溢出内部缓冲区的情况！
While(everMoreInput)
{
  …
```

```
}
```

有些 IDE,如 Visual Studio，能够为这类关键词自动生成摘要。

71.标出空语句

当在 while 或 for 循环之类控制结构中设计了空语句时，添加一行注释，指出这是作者本意。

```
//消除开头空格
While( (c ==reader.Read() ) == ‘’);
//空！
```

第五章

设计

全面阐述设计原则显然超出了本书的范畴。本章只包括了一些我们发现在优秀软件工程中至关重要的核心原则。

5.1 工程

72．别怕做工程

专业软件开发的终极目标是创建某种有用之物———更像是一种工程任务，而非一种科学任务。〔科学更直接地关注了解周围做界，这对于工程诚然也属必要，但却不够充分。)

要抵住诱惑，不要试图用代码对包括了所有理论上可行的科学现实进行建模。写出有限制性的代码并非过错，只要你确信这些限制性不会影响产品系统的功用。

例如，设想你需要做一个树遍历的数据结构，而且打算写栈代码。栈至少要能保存与任何树的最大深度数相等的条目数。假设对树的深度没有理论限制。你也许会通过重新分配内存、按需复制条目来创建能无限扩展的栈。但是，团队对应用程序的理解可能是你想都想不到的，也许就十来层树的样子。如果是这样，最佳选择就是创建一个固定长度的栈，可能最多容纳 50 个元素即可。

你也许会听到这样的抱怨：“多少还是有可能出现需要能容纳 51 个元素的树，那样程序就会崩溃，代码也就算是写错了。”我们确信，如果编写更为复杂的类来处理所有可能性，你的程序有更多可能因为编程错误、未预期的副作用或对类语义的错误理解而崩溃。

话虽如此，在文档中写明你的所有假设和你留意到的所有实践限制还是很重要的。稍后可能有人会决定使用你写的栈，但 50 个元素可能不够用，这中情况也很现实。这个文档应该在代码中直接体现，也要在随系统发行的技术文档中体现。

73.简洁优于优雅

在设计编码时要追求优雅，但有时最好舍优雅而取简洁。虽然优雅常具有简洁性，但并非所有简洁方案都够优雅。包括 50 行代码的 if-else 语句说不上优雅，不过如果这是解决问题的最简单直接的方法，就没有道理还要画蛇添足、改之而后快。

74.了解重用的代价

人们常以重用为面向对象编程的圣杯，事实上毋庸置疑，重用确乎是一种奇妙之物。不过，增加了依赖性和复杂度等代价也不可小视，这些代价有时足以抵消其助益。

可重用的组件难以创建和维护是出了名的。设计师不得不大量不同的使用场景。一旦投入使用，因为要照顾向后兼容问题，维护就会变得十分困难。组件必须要能重用的假定也给开发机构带来了压力，迫使不同开发组之间的协调工作增加。用户越多，复杂度越高，代价也越大。

75.按约编程

方法是调用双发之间的契约。契约要求调用者必须遵守方法前置条件，而方法也应返回满足与子相关的后置条件的结果。

遵守方法前置条件通常表示按方法所期待的形式传入参数，也可能意味着按正确顺序调用一系列方法。要遵守方法的后置条件，方法必须正确完成调用它要完成的工作，并且保持对象处于一致的状态。

在适当的公共方法中，应以异常和断言(参见规则 146)检查前置条件和后置条件。在方法开始处、其他代码执行前检查前置条件，在方法结尾处、方法返回前检查后置条件(参见规则 59)。

在从覆盖了超类方法的类派生新类时，必须保留超类方法的前置条和后置条件，为了确保这一点，可采用模板方法设计模式(通过公共非虚拟调用提供了功能性实现的保护虚方法)。公共方法均测试前置条件，调用关联的虚方法，再测试后置条件。子类可能通过覆盖虚方法来覆盖超类中的公共行为：

```
public class LinkedList<T>
{
    public void    Prepend(T t)
    {
      //测试前置条件
      Debug.Assert(…);
      DoPrepend(t);
      //测试后置条件
      Debug.Assert(Object.Equals(this[0], t));
    }

    protected    virtual    void    DoPrepend(T t)
    {
      …
    }
}
```

```
public    class Stack<T> : LinkedList<T>
{
    protected override void DoPrepend(T t)
    {
      ….
   }
}
```

76．选用适宜的工程方法

考虑选用一种适合你项目的软件设计方法。各种设计方法总结了最佳实践，避免经常导致软件项目失败的缺陷和沟通不足。例如，敏捷软件开发是一系列相关的方法论，强调小型自组织团队、迭代式开发循环以及持续测试和集成。极限编程(Extreme Programming, XP)是敏捷开发家族中的流行一员。敏捷开发不同于强调严格计划先行和文档驱动生命周期的“传统”方法。

77.分隔不同的编程层

分隔不同的编程层，创建一种更为灵活且易于维护的架构。例如，三层架构传统上包括 3 个层次：表现层，包括用户界面；中间层，包括应用逻辑；还有数据访问层。表现层仅与中间层沟通。中间层处理业务逻辑，并与数据访问层沟通。如果要修改数据访问层，甚至全部替代之，也无需改变表现层代码。

注意，逻辑层和物理层不必相同。

5.2 类的设计

78.让类保持简单

如果不确定是否一定需要某个方法，就不要添加它。如果另一个方法或方法的组合可有效完成同样的功能，就不要添加方法。添加方法易，剔除方法难。

参见 Martin Fowler 关于“Humane Interface”的讨论。

79 定义派生类，使其可以用在任何可以使用其祖先类的地方

通过覆盖来修改或限制其父类行为的派生类，是对该类一种特化（specialization），但其实例可能仅能有限地代换祖先类的实例。可能并非在用到父类的所有地方均可使用特化类。

在行为上与祖先类兼容的派生类是一种子类型(subtype)，其实例可与父类的实例完全代换。实现子类型的派生类并不覆盖其祖先类的行为，它只扩展祖先类提供的服务。子类型拥有与父类型相同的特性和关联关系。

下列设计原则陈述了可代换性的问题。

里氏替代原则

用户引用超类的方法必须能够在不知道子类的情况下使用子类对象。

根据该原则，用派生类对象代换父类对象的能力是良好的体现。与不采用该原则的设计相比，这能提供更强的稳定性和可靠性。如果某份设计案坚持该原则，就表明设计者很好地识别了基础对象，并且很好地泛化了抽象的接口。

开放—封闭原则

软件实体，如类、模块、函教等应对扩展开放，但对修改封闭。

要修改代码才能实现新派生类的设计是恶劣的设计。当派生类破坏可其祖先类与祖先类的客户之间的契约关系时，就会导致修改既有代码。当方法接受祖先实例，而使用该实例的派生类型控制其行为时，需要修改才能引入新子类。这类修改违反了开放—封闭原则，应该避免。

考虑下例：

```
public abstract class Shape
{
    public    abstract void    Resize(double scale);
}
public class Rectangle :Shape
{
    protected    double    width_;
    protected    double    height_;
    public    double Width
    {
        get
        {
            return    width_;
        }
        set
        {
            width_=value;
        }
    }
    public double Height
    {
      get
      {
        return height_;
      }
      set
      {
        height_=value;
      }
    }

    public override void Resize(double    scale)
```

```
    {
        this.Width *=scale;
        this.Height *=scale;
    }
}
```

改类构成了某个虚构的绘图包中形状部分的简单层次结构。Rectangle(矩形)以宽和高描述。Resize()方法用来按比例同步矩形的宽和高。

现在假设你决定添加一个表示正方形的新类。正方形是矩形的特殊形式，因此你从 Rectangle 类派生一个名为 Square 的新类：

```
public  class  Square : Rectangle
{
  public  double  Size
  {
      get
      {
          return width_;
      }
      set
      {
          width_=value;
          height_=value;
      }
  }
}
```

由于正方形宽高不一致，所以在 Size 这个属性里面总是赋予相同的宽和高。然而，这种现实有个问题：如 Square 对象被当作 Rectangle 对象传递给另一软件实体，而该实体又设置了 Width 和 Height 属性值，结果可能导致 Square 对象不能满足 width== height 约束。这一行为破坏了里氏替代原则。

你也许会考虑将 Width 和 Height 转换为虚属性，在 Square 类中覆盖这些属性，以满足该约束：

```
public class Rectangle :Shape
{
    protected  double  width_;
    protected  double  height_;
    public  virtual  double Width
    {
        get
        {
            return  width_;
        }
```

```
        set
        {
            width_=value;
        }
    }
    public virtual double Height
    {
      get
      {
        return height_;
      }
      set
      {
        height_=value;
      }
    }
}

public class Square :Shape
{
    protected  double  width_;
    protected  double  height_;
    public  override  double Width
    {
        get
        {
            return  width_;
        }
        set
        {
            width_=value;
        }
    }
    public override double Height
    {
      get
      {
        return height_;
      }
      set
      {
        height_=value;
      }
    }
```

```
}
```

尽管解决了上面的问题，但还是需要修改 Rectangle 父类，而你有可能没有权限修改。由于该方案需要修改既有代码，就违背了开放—封闭原则。

里氏替代原则和开放—封闭原则对方法同样适用。在下例中，为了处理新的形状类，比如 Square，需要修改 Canvass 类的 DrawShape()方法：

```
public class Canvas
{
    public void    DrawShape(Shape shape)
    {
        //
        if( shape is Circle)
        {
            DrawCircle( shape as Circle);
        }
        else if (shape is Rectangle )
        {
           DrawRectangle(shape as Rectangle);
        }
    }
    public void DrawCirclr( Circle circle) { …. }
    public void DrawRectangle(Rectangle rect ) {    …. }
}
```

当开发人员想要添加 Shape 类的新子类时，不得不修改 Canva s 类和 DrawShape 方法。要解决这个问题，可以往 Shape 子类添加一个名为 DrawSelf()的方法，用一系列 Shape 对象可用于绘制自己的简单绘制操作来替代与形状绑定的方法。Shape 类的每个子类都覆盖 DrawSelf() 方法，调用必要的图板绘制操作，画出特定形状：

```
public abstract class Shape
{
   …
   public abstract void DrawSelf(Canvas    canvas);
}

public class Circle : Shape
{
    …
    public override void DrawSelf(Canvas canvas)
    {
       ….
    }
    …
}
public class Canvas
```

```
{
   …
   public void DrawShapes(Ienumerable<Shape> shapes)
   {
       foreach(Shape shape in shapes)
    {
        shape.DrawSelf(this);
    }
   }
  //定义 shape 会用到的操作
  public void    DrawSelf(int x, int y1, int x2, int y2)
  {
      …
  }
  public void DrawCircle(int x, int y , int radius)
  {
      …
  }
   …

}
```

80.对于“is-a“关系使用继承，对于“has-a”关系使用包含

在做面向对象设计时，在继承与包含之间做选择是最重要的决策之一。用继承来给“is -a”关系建模，用包含来给“has-a”关系建模，这是一种经验法则。

例如，卡车有一(has-a)组轮胎，运送冰激凌的卡车是一(is-a)种特殊种类的卡车：

```
public       class    Wheel
{
    …
}

public class Truck
{
       private Wheel[] wheel_;
}

public class IceCreamTruck :Truck
{
    ….
}
```

81.对于“is -a”关系使用抽象基类，对于“实现“关系使用接口

根据经验，使用基类指明类之间的“is-a”关系，使用接口指明实现关系。

在类与接口之间做选择时，记住以下几点。

类可以包括字段及方法的默认实现，接口只定义方法签名。

类比接口易于扩展：加入新成员时可以不破坏派生类，而扩展接口时会破坏实现该接口的既有类型。

类只能有一个基类型，但能实现任意数量的接口。

关于这些要素的详细讨论，参加《《.NET 设计规范》》一书（人民邮电出版社，2006）。

5.3 线程安全和并发

当两个或多个线程同时执行指令时，并发(concurrency)就产生了。单处器系统可以通过在两个或多个线程间切换执行来模拟并发。多处理器系统能够通过在每个处理器上执行单个线程来支持并行并发。

许多应用在实现户采用并发而获益。在并发执行模型中，应用被分为两个或多个进程或线程，每个进程或线程按自己的语句或指令顺序执行。一个应用可能包含一个或多个进程，而每个进程可能包括一个或多个线程。执行可以分布到网络上的两台或多台计算机上，或单台机器的两个或多个处理器上，或在单个处理器上做交叉处理。

被隔离开的执行进程或线程通常必须争抢共享资源和数据，并且必须同完成总任务。

并发应用程序开发是一项复杂的工作。设计并发应用程序牵涉判断进程或线程的需要数量、单个进程或线程的职责以及互动的方法。还牵涉判断良好、合法、不变的程序状态，以及不好或非法的程序状态。关键问题是，找到和实现一种即便出现两个或多个线程对同一资源进行操作的情况时，也能维护或保证良好程序状态、防止不好程序状态出现的方案。

在并发环境中，开发者通过同步来限制或协调对共享资源的访问，从而维护期望的程序状态。同步的主要角色是防止多个线程同时执行时不合需要或未预料到的冲突。

同步描述了一组防止不合需要的交叉执行操作或者并发线程间冲突的机制或进程。这主要通过串行化对共享程序状态的访问来实现。程序员应该在两种同步技术中择一而为：互斥（mutual exclusion)或条件同步(condition synchronization)。

互斥是将细密的原子操作与粗放的操作合成起来，并使得合并操作具有原子特性。

.

条件同步描述了一种延迟线程执行直至程序满足某种预设或条件的进程或机制。

处于线程机制等待状态的线程称为阻塞（blocked)的。当线程是非阻塞（unblocked）、唤醒的〔awakened) 或告知的(notifiedd) ，就被重新调度准备后面的执行。

线程同步有两种基本用途：保护共享数据完整性，在协同线程间沟通程序状态的变动。

如果多个线程能同时访问某个实体，则改实体是多线程安全（multithread-safe,MT-safe）的。静态类方法可以与该类实例的方法有不同的线程安全级别。如果某个类或方法需要创建额外的线程完成任务，则该类或方法被看作是

multithread-hot(MT-hot)的。

82.设计可重入的方案

总是编写可重入的代码，即在被单线程循环调用或被多线程并发调用时均工作正常的代码。要编写可重入的代码，就不要使用静态分配的资源，除非采用了某种互斥机制来确保对该资源的串行访问。静态资源包括共享对象、I/O 设备以及其他硬件资源等。

83.只在合适的地方使用线程

使用多线程并不必然等同于改善应用程序性能。有些应用程序不使用，反而会因为引入多线程后在管理线程上开销过大而变慢。

在采用多线程之前，考虑应用程序是否会从中受益。如果应用程序有以下需求，可以采用多线程。

同时响应许多事件。例如，Web 浏览器或者服务器。

提供极高响应能力。例如，当应用程序在进行其他计算时也要能持续响应用户操作的用户界面实现。

尽多处理器之用。

84.避免不必要的同步

同步可能代价昂贵。同步是为了串行化对一个对象昂的访问，以此将并发可能性降至最低。在将代码修改为支持同步之前，先看着一该段代码是否存取了共享的状态信息。如果某个方法只操作独立同步的对象、局部变量或非易失的数据成员，即在构造时初始化的实体，则同步并非必要。

不要同步基础数据类型或结构，例如列表、向显等。让这些对象的用户自行判断是否有必要进行外部同步。

5.4 效率

85.使用做懒惰值和懒惰初始化

在需要结果之前，不要进行复杂的计算。总是在最靠近嵌套边缘的地方执行计算。如有可能，缓存结果。

此概念亦可适用于对象构造及初始化——在需要对象之前，不构造对象。使用简单专一的函数访问对象。该函数必须在初次调用时构造对象，并在该点返回对象的引用。需要访问对象的代码均需使用该函数。可能需要串行化（加锁）来防止并发初始化。在下例中，我们假定 LoanCalculatr 是在必须创建之前不做创建的对象。

```
public class LoanCalculator
{
    //…
}
```

```
public class PersonalFinance
{
    private LoanCalculate loanCalculator_=null;
    public double CalculateIntereset()
    {
        //使用双重检查模式以
        //防止并发构造发生
        if( loanCalculator_==null)
        {
            //加锁…
            if(loanCalculator_==null)
            {
                loanCulculator_=new LoanCulculator();
            }
            //释放锁…
        }
    }
    return loanCalculator_.CalculateIntereset();
}
```

86.重用对像以避免再次分配

缓存并重用频繁创建而且生命周期有限的对象。

使用访问器而不是构造函数来重新初始化对象。

使用工厂设计模式来封装缓存和重用对象的机制。为了恰当地管理这些机制，必须将从某个对象工厂获取的对象返回到同一个工厂。这表示对象及其二厂之间的关联必须在某处得到维护。

在类中———单个静态工厂与该对象的类有关联，而且该工厂管理该类的所有对象。

在对象中———对象维护着到管理自己的工厂的引用。

在对象所有者中———对象的“所有者”维护着从中获取该对象的工厂的引用。

留心选择无需创建自己的对象的实现来管理要缓存的对象。否则会导致失败！

87.最后再优化

优化的第一原则：

不要优化。

优化的第二原则〔只对专家有效)：

还是不要优化.

—麦克·杰克森，Michael Jackson 系统公司

在确认需要做优化之前，不要花时间做优化。

采用 80-20 原则：系统中 20%的代码使用 80%的资源(平均而言)。如果要做优化，确认从那 20%的代码开始。

88.避免创建不必要的对象

如果新对象生命周期较短，或者构造后从来不引用，则此规则尤其重要。这不仅会浪费时间创建对象，还会在垃圾回收上耗费时间。

下面代码演示的多余初始化操作极为常见，也极为浪费：

```
public Color TextColor
{
    get
    {
       Color c=new Color (…);
       if( this.state <2)
       {
              c=new Color(…);
       }
       return c ;
    }
}
```

在知道自己需要什么之前，避免创建对象：

```
public Color TextColor
{
    get
    {
       Color c=null;
       if( this.state <2)
       {
              c=new Color(…);
       }
       return c ;
    }
}
```

89.让 CLR 处理垃圾回收

公共语言运行库(Common Language Runtime, CLR)中的垃圾回收器让你免于跟踪内存使用，也不必知道何时释放内存。一般而言，避免调用 GC.Collect()方法，让垃圾回收器自行运作。在多数情况下，垃圾回收器的忧化引擎要比你更善于判断执行回收的最佳时机。

第六章

编程

本章阐述了一些推荐采用的 C#编程约定。

6.1 类型

90.使用内建的 C#数据类型别名

所有 C#基本类型均有别名。例如，int 是 System.Int32 的别名。别名最具可读性。即，与其写

```
System.Int32 i=new System.Int32(4);
```

不如用更简洁的同等语句替换。

```
int i =4;
```

91.避免使用内联字面量

避免这样的代码：

```
if( size > 45)
{
   //…
}
```

声明一个常量，不但增加可读性，而且便于在一个位置修改。

```
const int    Limit =45;
…
if( size >Limit)
{
   //…
}
```

92．邀免不必要的值类型装箱

将值类型转换为引用类型的操作被称作装箱(boxing) 。当编写类似以下代码时，装箱会隐式发生：

```
object obj =4;
```

拆箱（unboxing）是从引组类型向值类型的转换，如下：

```
int i=(int)obj;
```

如果在引用类型环境中使用值类型，则装箱/拆箱不可避免。这会明显增加性能上的开销，所以应尽可能避免重复的装箱/拆箱操作。例如，如果你想保存一组整数，宁可用 List 也不要用基于对象的 ArrayList。

93.使用标准形式书写浮点字面量

在浮点数小数点的前后均应是数字。

下而的语句难以阅读。

const double Foo =0.042e2;

下面的语句至少符合标准。

const double Foo =4.2e0;

但下面的语句更好。

const double Foo=4.2;

94．为“值”语义使用结构

多情形下，可以将对象设计为 class 或 struct。选用 struct 表示该对象将在值语义的栈上创建。

注意，struct 不能从基类派生，是密封的(sealed)(因此它们无派生〕，而且没有默认的构造函数。

95.考虑在结构中覆盖等同方法和操作符

struct 应遵循值语义。因此，应覆盖等同方法和操作符，并且实现值行为而不是从 System.Object 继承引用行为。

96.使用“@”前缀转义整个字符串

使用“@”前缀转义字符串中的所有字符较为方便和易读。

```
string path= “C:\\My Folder\\Acme”;
```

上一行代码可以用字符串文本的形式实现：

```
string path= @“C:\My Folder\Acme”;
```

97.避免代价昂贵的隐藏字符串分配

如果下面的代码行位于循环之中，则将在每次循环中进行两次字符串分配操作。

```
if( str1.ToUpperCase() == str2.ToUpperCase())
```

可以用下面的方式避免这一开销：

```
if( string.Compare(str1,str2,str3) )
```

同样，字符串拼接也需要做一次字符串分配：

```
string str=””;
while( …)
{
    str+=”.”;
}
```

使用 StringBuilder 类并在拼接结束后转换为 string 会更为有效。另外，如果几个字符串总被重复使用，则应该创建为一个 String。

```
StringBuilder buffer =new StringBuilder();
string temp=””;
while( …)
{
    buffer.Append(temp);
}
string str=buffer.ToString();
```

这种方法省下了每次循环所需的额外字符串分配操作。

98.采用有效的空字符串检侧方法

类似下面的代码需要创建一个新的空字符串：

```
if(str.Equals(“”) )
```

而检测长度则不必创建字符串：

```
if(str.length==0)
```

另外，.NET2.0 引入了静态方法 String.IsNullOrEmpty()，用一次高效测试就能检测空引用和空字符串。

99.只在必要时使用空值

.NET2.0 引入了可空值，允许你得到可设置为空的内建数据类型。

例如：

```
int ? a=null;
```

尽管 int?可以在所有使用 int 的地方正常工作，但使用可空使会有性能上的影响，所以不应该无条件地使用。只在确认自己有必要用空值的情况下使用空值。

100.仅为支持机器生成代码使用部分类型

.NET 2.0 中引入部分类型，主要是为了支持代码生成工具。代码生成是一种很有用的技术，但将类中非代码生成的部分单拆成多个文件可能会让阅读代码者感到闲惑。

6.2 语句和表达式

101.在复杂表达式中不要依赖操作符优先级

复杂的表达式很难用肉眼分析和理解，特别是那些依靠操作符优先级做子表达式计算和排序的表达式更是如此。在编写或修改依赖操作符优先级的表达式时，你和以后的维护者都很容易出错。使用括号定义和控制子表达式求值会让代码易于理解和维护。

```
//试着得到 60
int j=10 *2 << 1 +20 ; //不好！  j==41943040
//添加圆括号
j=(10 *  (2 << 1) ) + 20 ; //好！  j==60
```

102.不用 true 或 false 测试相等

没必要用 fa1se 或 true 比较布尔值。

```
bool b=false;
if(b==false)
{
    //…
}
```

直接使用布尔值：

```
bool b=false;
if(!b)
{
    //…
}
```

103.用等价方法替换重复出现的非普通表达式

在编写代码时，找出可以拆分为单个方法的重复表达式或操作。

用函数调用替换重复代码，能简化和缩减代码的长度。这会让你的代码更易读。

用有意义的函数名称替换重复代码，改善了代码的自说明性。这让你的代码更易于理解。

把代码拆分为单个方法，通过将行为局部化简化了测试，通过将变动局部化简化了维护工作。

104.在三元条件判断中避免使用复杂语句

对于简单的情况， 三元形式是一种方便、易读的控制流程机制：

```
Short i=(d<0 ) ? -1 : 1 ;
```

别在复杂的情况下使用:

```
Customer customer =(! busy || flag ==3) ?
                find(name) :
                CreateCustomer(name,address, birthdate);
```

105.使用 Object.Equals()测试引用类型的对象等同

某个类型的非静态方法 Equals()和==操作符可能会被另外一个使用值等同的类型所覆盖。如果想引用等同，那么使用 Object. Equals()方法显式地做检查。

```
if( Object.Equals(string1,stirng2) )
{
  //…
}
```

6.3 控制流程

在看到一段代码时，我们通常会假设代码内部的语句会顺序执行。虽然制流程语句可能会选择不同的语句或语句块来执行，我们还是期望流程在第一个语句处进入代码块，在最后一个语句后退出代码块一——假设每个代码块都有唯一的入口和出口。软件开发社区长久以认为遵循此模型编写代码是一种良好的编程实践。使用这种方式编写的代码易于调试，因为我们只需要寻找单个出口。

C#程序员可以利用 goto、break、 continue、 return 和 throw 语句来破坏这种控制流程模型。尽管在许多情况下建议使用甚至必须使用这类语句，但这些语句还是会让代码难以阅读、理解、测试和维护。

106．避免在循环语句中使用 break 和 continue 语句

break 和 continue 语句中断了执行的正常流程。break 语句会立即退出:最近的循环语句及所有在其中的语句块，并继续执行后续语句。continue 语句会立即跳到最近的循环语句控制表达式，退出所有其中的语句块。这两个语句的行为很像是 goto 语句。

如果最近的循环块过大或太复杂，读者很容易忽略这些语句的出现，读者也许会错误地配对循环语句中的 break 和 continue 语句，从误解代码。有多个出口的代码块更难于调试，因为需要在每个出口设置断点。

尽量组合使用 if…else 语句和控制表达式产生使用 break 和 continue 完成的功能。如果你发现，为了避免使用 break 和 continue 语句需要创建和求值一堆额外的状态变量或复杂的表达式，那么可以选择忽略此规则。在复杂性和性能上付出的代价会抵消风格上的考虑。如果一应要使用 break 和 continue 语句，就添加明显的注释来突出这些特殊的出口。

107 在方法中避免使用多个 return 语句

在函数块垠后一个语句之外的任何地方使用 return 语句都会打破正常的执行流程。读者通常会在函数尾部寻找返回语句，所以如果在别处放置返回语句，应添加明显的注释来突出这些出口。

108.不要使用 goto

是否使用 goto 是计算机编程领域中最大的争论之一，最初由 Edsger Dijkstra 的著名论文 Go To Statement Considered Harmful 引起。

在缺乏有力控制构造的编程语言中，goto 语句可能有一席之地，但如果你觉得在 C#中需要使用 goto 语句，那么你可能做错了(但有时也没错)。

109．不要使用 try…throw...catch 来管理控制流程

尽可能只用 C#的异常机制处理异常情，.不要将其用作条件表达式 if…else、for、while、和 do…while 或类似 return、break 和 continue 等控制流程语句的替代品。

参见“TryParse”模式。

110.在 for 语句内部声明循环变量

循环变量应该在 for 语句中声明。这样可以将变量的作用域限于 for 语句之内：

```
for( int count=0; count < length; count++)
{
  //仅在此块中可见才计数…

}
```

111.给所有 switch 语句的结尾添加 default 标记

并解释默认情况如何或如何不能运行到。

```
swith( color)
{
    case Color.Red:
        //..
        break;
    case Color.Bule:
        //..
        break;
    case Color.Green:
        //..
        break;
    default::
      //这可能无法获得，因为假定颜色只是红、蓝或者绿色。.
        Debug.Assert(false,”Invalid color value”);

}
Swith ( day )
{
  Case Day.Saturday:
      //…
      Break;
  Case Day.Sunday:
      //…
      Break;
  default:
      //这是星期中的某一天...
```

}

6.4.类

112.定义小类和小方法

较小的类和方法易于设计、编码、测试、编写文档、阅读、理解和使用。因为较小的类通常拥有更少的方法，能表达更简单的概念，所以其接口倾向于表现出更好的内聚性。

尽量将每个类的接口限制在提供必要功能所需的最少方法之内。当只有一种通用形式的方法足够使用时，应避免添加该方法的“方便”形式。

各种非正式的指导都为创建最大的类或方法而存在 ———— 好自为之。如果某个类或方法看来太大，则要考虑重构该类或方法，以多写一个类或方法。不要不情愿把较大的方法改写为较小的私有方法，即便你在其他方法中只会调用这些小方法一次。切割开的代码更易于阅读，而且你可能会发现可以重用其中一些新方法。此外，如果拆分成小方法，CLR 也能更好地优化代码。

113 从标准类型构造基础类

在设计底层的基础类型或具体类型时，应该尽量减少对非标准类型得依赖。每次在基础类型的接口中包含非标准类型时。都引入了新的不秘定依赖。这种依赖可能不仅让你的代码易受改动的影响，而且还增加了代码的编译和执行“蓝图”。

尽可能限制自己只使用.NET 定义的类型。

114.避免在用户可扩展的类层次结构中使用虚基类

用户在扩展拥有虚基类的类层次结构时，可能会没留意自己也许需要在派生类构造函数中调用虚父构造函数。如果你希望用户扩展有虚拟类的类，那么你需要详细阐述是否必须初始化父类，这可能会迫使你公开一些最好是隐藏起来的实现细节。

如果一定要用虚父类，好好设计你的实现方式，以便虚父类可以通过默认构造函数初始化———如果用户没有指定，编译器就会自动调用这个构造洒函数。

115.声明所有成员的访问级别

不要假设别人会记得默认的访问级别。只要有可能，就把同等访问级别的声明放到同一组里面。许多程序员循以下顺序放置声明：public,然后是 protected，跟着是 private。

116.将类标记为 sealed,防止不想要的派生

如果想禁止从“封闭”类派生新类，就使用 sealed 关键字。所有访问修饰符都应为 public 或 private；在这种情况下，protected 没有任何意义。

```
public sealed class ClosedClass
{
    private int id;
    public ClosedClass()
    {
        //…
    }
}
//非法！无法从 sealed 类派生
public   class Derived :ClosedClass
{
```

```
        //…
    }
```

关于 sealed 关键字的使用，编程社区中存在激烈争论。

117．避免使用 internal 声明

internal 声明常标志着不好的设计，因为它绕过了访问限制，而且隐藏了类和函数之间的依赖关系。只应在需要防止子类获得特定超类方法时允许特定辅助类、操作符和函数访问这些方法的情况下使用 internal 声明。

118.避免使用 new 来隐藏派生类型的成员

如果子类拥有与父类函数签名相同的函数，但该函数并没有覆盖虚函数，那么你的设计可能有问题了，应该避免只为了对付编译器错误而使用 new 关键字。

119.限制 base 关健字在子类构造函数和覆盖方法中的使用

base 关键可用于调用基类的任意函数。只应在必须时使用，例如：

```
public Child() : base()
{
    //…
}

public override object Clone()
{
    Foo foo =(Foo) base.Clone();
    foo.field=field;
    return foo;
}
```

下面代码中 base 的用法令人迷惑：

```
public bool IsGood()
{
    //只调用 Solve()
    return    (base.Solve() !=null);
}
```

120. 在覆盖 Equals()方法时也覆盖 Operator==.和 Operator!=

如果等价操作符与 Equals()方法保持同步的话，代码就会清晰许多。

```
    public    boolean    operator==( Student lhs,
                                     Student rhs)
    {
        return lhs==null ?
                    Rhs ==null : lhs.Equals(this);
    }

    public bool Equals(Student rhs)
    {
        return    ((rhs!=null )&& (id_==rhs.id_));
```

```
}
```

121.在覆盖 ToString()方法时，考虑覆盖隐式字符串转换操作符

覆盖某个类的 ToString()方法，展示有意义的字符串，会对调试和记录颇有帮助。

如果对字符串的显式转换有意思，或许隐式转换也同样有用。然而，如果这么做了的话，小心你的类型可以在字符串能用的任何地方被用到，这可能会导致不希望出现的后果。

122. 用相反功能的方法来实现方法

用相反功能的方法来实现方法有两个目的：明示方法的含义（参见规则 2），以及简化代码。

```
public void    Deposit(double amount)
{
    balance_+=amount;
}

public void Withdraw(double amount )
{
   Deposit(-amount);
}
```

代码的阅读者总期望 Deposit()方法与 Withdraw()方法有逻辑相反关系，这合乎逻辑。这样实现方法就能确保语义的呈现，甚至以后其中的实现之一有改动也是如此。例如，Operator+ 与 Operator – 可以说是相同的。

6.5 生命周期

123．初始化所有变

未初始化的变量是所有软件缺陷中最常见和最隐蔽的一种。包含未初始化变量的软件只在其内存位置刚巧有变量值的时候才能运行。这可能保持数年之久，直至进行一次新编译或调用顺序变动在变量的内存位置上留下了不同的、不期望出现的值。

这类风险在类似 CLR 这样的托管代码环境中已极大降低：在多数情况下，初始化变量失败将导致编泽错。养成显式初始化所有用到的变量的习惯，是良好的实践，而且也最为易读。

124.总是构造在有效状态的对象

不允许构造无效对象。如果为了在多步初始化过程中使用而实现不产出有效对象的构造函数，就将该构造函数声明为 protected 或 private，并使用一个静态方法协调构造过程。

为了保持构造函数简而快速，考虑采用懒惰求值来推迟初始化(参见规则 85)。

125.为增加的透明性和 COM 互操作性声明显式默认构造函数

应声明和实现公共默认构造函数，明确你的类支持默认构造。即便没有初始化且函数体空置，也应定义构造函数。如果此编译器生成构造函数，则可能你没考虑周全类的所有初始化需求。提供默认构造函数时，你应该提供能用于设置调试断点和执行跟踪的源代码。

如果你的类不支持默认构造，则应声明保护的默认构造函数。直接实例化默认对象将会导致编译错误。如果想要为子类提供默认构造功能，则只需要实现保护的构造函数即可(参见规则 126）。

如果你的类不支持默认构造，或只包含静态成员，则应声明私有的默认构造函数。直接实例化默认对象将会导致编译错误。在这种情形下，不要提供实现，因为该构造函数永远不会被调用。

还要注意，如果你自行提供了任何其他构造雨数，编译器不会生成默认认构造函数。

126.声明构造函数为保护的，禁止直接实例化

要禁止直接实例化一个抽象类，应该只允许对构造函数的受保护访问。派生类仍然能够构造抽象类，但其他实体则不能。

127．总是在派生构造汤数的初始化列表中列出全部基构造函数

派生类不应直接初始化父类成员———初始化父类成员是父类构造函数的职责。反之，应在初始化列表中包括合适的父类构造函数，即便它是默认构造角数也应如此。

```
public    class Base
{
        protected int foo_;
        public Base(int foo)
        {
           foo_=foo;
        }

}

public class A : Base
{
      public A（int foo）
      {
          //不好！让 Base()实现这个
          foo_=foo;
      }
}

public    class B : Base
{
     public B(int foo) :base (foo) //好
     {
     }
}
```

128.使用嵌套的构造函数消除冗余代码

为了避免写出冗余的构造函数代码，在高层构造函数中调用低层构造函数。

下列代码在两个地方实现了同样的低层初始化过程。

```
public    class Account
{
     private    string name_;
     private    double balance_;
     const      double DefaultBalance=0.0d;
```

```
        public Account( string name, double balance)
        {
            name_=name;
            balance_=balance;
        }
        public Account(string name)
        {
            name_=name;
            balance_=DefaultBalance;
        }
    }
```

下面的代码只在一个地方实现了低层初始化。

```
    Public class    Account
    {
        private    string name_;
        private    double balance_;
        const      double DefaultBalance=0.0d;
        public Account( string name, double balance)
        {
            name_=name;
            balance_=balance;
        }
        public Account(string name)
        {
            this(name,DefaultBalance);
        }

    }
```

如果正在使用断言这种方法也会有用，它通常会减少特定构造函数参数出现的次数，从而减少该参数的有效性检查的数量。

129.在引用外部资源的类中实现 Idisposable

如果类提供了 Dispose()和 Close() 方法，应总是调用。

如果你的类拥有外部或非受管资源，则总应实现 IDisposable 接口和模式。

为 Idisposable 提供 Dispose()方法，调用你自己的终止代码。本例中取消了.NET 垃圾收集。

```
    public void Dispose()
    {
        Dispose(true);
        GC.SuppressFinalize(this);
    }
```

实现一个保护方法，来做清理工作。

```
    protected    void    Dispose(bool disposing)
    {
```

```
    //如果已调用请检查
    if( !disposed_)
    {
        //如果 disposing 为 true,释放所有受管和非受管资源
        if(disposing)
        {
            //释放受管资源
        }
        //调用合适的方法清理非受管资源
        disposed_=true;
    }
}
```

参见规制 150.

6.6 字段和属性

130.声明所有字段为私有访问级别，使用属性提供访问

声明所有类数据成员(在 C#中称作字段)为私有访问，使用属性让类的用户和派生类能访问这些私有数据成员。将实现细节看作私有信息，降低在实现改动时对依赖类的影响，并且让数据持续有效。

下面的代码为私有的 month 字段提供了读写访问功能。

```
 public Date
 {
     private int month_;
     public    int Month
     {
       get
       {
           return month_;
       }
       set
       {
           month_=value;
       }
     }
}
```

如果类的用户应该能读取 month 值，但不能修改，就省略 set 属性。

```
   public Date
  {
     private int month_;
     public    int Month
     {
       get
       {
```

```
            return month_;
        }
    }
}
```

可以在属性的实现中包括数据确认。这样，所有对属性的修改都能保持有效。

```
public Date
{
    private int month_;
    public    int Month
    {
        get
        {
            return month_;
        }
        set
        {
            if( value <1 || value >12)
            {
                throw new ArgumentOutOfRangeException();
            }
            month_=value;
        }
    }
}
```

131.只为简单、低成本、与顺序无关的访问使用属性

对于 API 用户，属性就像是数据字段，而且通常会依此对待。因此，属性的实现不应昂贵或有不期望的副作用。

例如、属性返回私有成员的值或是完成简单计算，都是合适的;而让 set 属性将信息保存到数据库或让 set 属性下载网页，就可能是不合适的。这样的操作应暴露为显式方法，这样调用者就能明白起作用。

6.7 方法

132．避免传递过多参数

如果某个方法获取大量参数，那么考虑是否应该重新设计它。参数太多标志着方法做了太多事，或者数个相关参数可被抽象到另一个类后面。

这样的代码：

```
public    bool IsPolisyValid(int originalYear，
                             int originalMonth,
                             int originalDay,
                             int originalHour,
```

```
                                    int renewalYear,
                                    int renewalMonth,
                                    int renewalDay,
                                    int renewalHour,
                                    int effectiveYear，
                                    int effectiveMonth,
                                    int effectiveDay,
                                    int effectiveHour)
{
    //…
}
```

用日期抽象重写成下面这样可能会好些。

```
public   bool IsPolisyValid(DateTime originalDate，
                            DateTime renewalDate,
                            DateTime effectiveDate)
{
    //…
}
```

133．检验参数值有效性

如果方法对参数的值做了些假设(例如。表示月份的整数在 1 到 12 之间)，在方法开始处验证这些参数

很据期望的运行时行为，以及在发布和调试版本中是否需要进行验证，可以利用 System.Diagnostics.Debug 类或抛出 System.Exception 类层结构中的异常。

参见规则 148 和规则 158。

6.8 特性

134.使用 SystemObsoleteAttribute 弃用 API

如果类库的不同版本导致了接口的改动，使用 SystemObsoleteAttribute 弃用旧接口，而不是简单地移除或改动接口。这样会向类的用户显示一条编译时消息，解释改动，或许还建议解决方法。

SystemObsoleteAttribute 的一个构造函数允许类的实现者指定是允许继续使用弃用的元素〔编译器警告)还是不允许(编译器错误)。

135．考虑新类是否可被串行化

考虑新类是否可被串行化(Serializable)。查阅《《Microsoft .NET Framework 开发者指南》中的 Serialization .Guideline，其中列出了对特定类做出此决定的原因。

有多种途径让类可串行化：用 Serializable 特性标注类是最简单的方式，因为无需增加代码。作为替代方案，实现工 Serializable 接口可获得更好的性能，因为它不像 Serializable 一般使用反射，同时也允对输出的更多控制。

136.使用 System.FlagsAtttribute 指明位域

用 System.FlagsAtttribute 标注枚举，向编译器指出这些值可以无需警示按位做或运算。

```
[Flags]
enum Access
{
```

```
        None=0,
        Read=1,
        Write=2,
        Admin=4
    };
    //…
    Access userPermissions =
                Access.Read| Access.Write ;
```

另外，这还将为枚举的 ToString() 方法提供更友好的实现。对于上例，userPermissions.ToString()将返回。

Read， Write

果没有应用 F1agsAttribute，上列调用将返回。

6.9 泛型

137.泛型类型胜过未指定类型或强类型类

.NET2.0 引入了泛型(generic)，允许在运行时才进行类型特化。这样就能定义类型安全的数据结构，且无需在代码中指定真实类型。

在.NET 1.1 中,集合通常以松散类型〔例如 Ar rayList ，可容纳 Object 类型的条目)实现，或为每种具体类型创建强类型类(可能使用某种代码成工具)。松散类型在类型安全和性能之间求得折中，强类型则需要创建和维护额外代码。

与此不同，泛型让你能在任何类型上重用代码，并提供完整的编译器支持和类型安全性。

6.10 枚举

138.使用枚举而非布尔型来改善参数可读性

当开发人员遇到对函数的调用时，布尔型函数参数的意思常会丢失。用枚举替代布尔型参数能够提升代码可读性，让读者不必为了找出参数的意思而去查阅文档。

```
    //这些参数是什么意思?
    customer.ShowReport(true,true);
    //将输出送往屏幕和打印机
    customer.ShowReport(Report.Screen |
                        Report.Printer);
```

139.使用枚举值而非整型常

使用 enum 语句指明描述了特定概念或属性的整型顺序值，例如颜色、方向、类别、模式等。枚举提供了整型常量不能提供的类型安全性和清晰度。

140.创建 0 值枚举元素表示未初始化、无效、未指定或默认的状态

用类似 None、Unspecified、Unknown 或 Invalid 之类的名字定义一个枚举元素，为可变枚举的有效性验证提供类型安全的方法。

```
enum Color
{
    None=0,
    Red=1,
    Green=2,
    Blue=3
};
```

将该枚举元素置于枚举中第一个位置。这样，当别人添加新元素时，就不太容易忽视这个枚举元素，而且也让编译器自动给该枚举元素赋 0 值，使之易于用运行时断言捕捉无效值。

141．验证枚举值

不要假定 enum 值总会在定义的范围之内，而且将整型值转换为 enum 类型也是合法的，即便该整数没有对应枚举中的任何值。例如，给定：

```
enum Color
{
    None=0,
    Red=1,
    Green=2,
    Blue=3
};
```

尽管下面的代码不是我们期望的，但却是合法的写法。

```
Color c=(Color) 42 ;
```

遵循合适步骤验证枚举值的正确性，特别是当它是从类似 XML 文档或数据库等外部资源构造而来时尤其如此。

可以用库函数 Enum.IsDefined()来判断枚举中是否存在某个值。然而，要小心 Enum.IsDefined()是一种昂贵的运行时函数调用(使用反射)，而且对于[Flags]特性中值的组合并不能如愿工作，最好显式地检查期望的值。

```
switch( color)
 {
    case Red :
    case Blue:
    case Green:
        …
        break;
    default :
```

```
            throw new ArgumentOutOfRangeException();
            break;
    }
```

6.11 类型安全、强制转换与转换

142.避免强制转换，并且不要强迫别人使用强制转换

显式强制转换与语言提供的一般类型安全机制有关。转换操作可能产生细微而不易察觉的错误，直至某人修改或扩展你操作的类型和值。

让编译器生成类型安全的隐式转换，只把强制转换操作符当作最后一招。此规则的最常见的例外是在基于 Object 的集合中引用对象的时候(虽然在使用.NET2.0 的泛型集合时这一点并不必要)，以及访问通过一般 event:委托参数引用的对象的时候。

当在类型之间转换对象时，尽量使用 System.Convert 类中的类型安全函数，而不是直接强制转换。

143.as 操作符胜过直接强制转换

如果对象实例与给定类型不兼容，则直接强制转换将抛出一个 InvalidCastException 异常。C#的 as 操作符将尝试把对象强制转换为指定类型，如果失败则返回 null。

如果存在强制转换失败的可能性(例如，正在使用基于 object 的集合，其中容纳了混合类型的元素)，则使用 as 操作符可以使用直接条件语进行检测，而无需处理异常。参见规则 144。

144 使用多态而不频繁使用 is 或 as

C#的 is 操作符将检测对象实例是否与给定类型兼容，并返回一个布尔值。C#的 as 操作符尝试将对象强制转换为指定类型，如果失败则返回 null。

尽管使用这些操做作符有充分理由，但如果频繁使用这些运行时类型检侧，就该考虑是否要修改设计了。对继承和多态的明智使用也许能充分简化设计。
辘侧

6.12 错误处理和调试

145.使用返回码报告预期的状态改变

可使用返回码、标记或方法报告预期的状态改变。这让代码更为可读，控制流程更直接。例如，在从文件读入的过程中，可预料到在某一点将读至文件尾———不要用异常来报

告到达文件尾部。

146.使用异常强迫获得编程契约

如果传递给方法的参数无效，或者调用方法时相关对象处于无效状态，则使用前置条件异常(例如，ArgumentOutOfRangeException)。

在检测方法产出结果的有效性或在方法返问前测试相关对象是否处于有效状态时，使用后置条件异常。

通常，如果异常与方法调用者有关才使用该异常。如果检测出方法内部的逻辑错误，脱离了调用者的控制范围，则应使用断言（Debug.Assert）。

147.不要静默地接受或忽略非预期的运行时错误

要创建健壮的软件，需要甄别出运行时错误的潜在来源。如果有可能从错误中恢复过来，就应该这么做，至少要让客户有机会这么做。

总是确认操作成功完成。如果方法返回表示失败的标识，应找到失败原因并采取恰当的措施。检验返回值，并使用 try...catch 代码块检测并回应错误。不要为了抑消异常而创建空的 catch 代码块(参见规则 152)。

如果检测到不可恢复的错误，使用断言或异常报告这一错误〔参见规则 148) 。如果你能访问某种形式的执行记录工具，就在采取采取其他行动之前先记录一下错误的原因、类型和位置。

148.使用断言或异常报告非预期的或未处理的运行时错误

你很少能预知或恢复所有运行时错。有些错误，例如硬件错误、第三方软件错误或内部程序逻辑错误太过严重，完全无法恢复。可能只终止程序。

可以抛出标识非预期运行时错误的异常。但在软件的调试版本中采用 Debug.Assert() 会有帮助：断言产生一条开发人员自定义的信息，可以按需提供说明， 而且能在失败点直接中断执行。

枪查代码中的所有假设。例如.如果某个算法假设列表中只会容纳单个元素，则用断言测试之。同样，也用 Debug.Assert()检测非预期状态、逻辑错误、索引越界等。

只将断言用于检测非预期或不可恢复的错误。如果调用者能够从错误条件中恢复，则代码应返回错误码或抛出异常。

由于断言中用到的表达式仅编译到调试版，绝对不要在这些表达式中使用“活”代码———有其他作用的代码。

149．使用异常来报告可恢复错误

使用异常来报告非预期但有可能恢复的错误。典型的例子包括：
由于磁盘已满导致的文件写操作失败。
由于磁盘已移除或卸载导致的文件访问操作失败。
系统无法满足内存分配请求。
系统通信软件遇到无效协议、序列或格式。
请求或回应通信在非预期的情况下被客户打断或取消。

150.使用 try . . . f inally 代码块或 using 语句管理资源

在代码释放之前获得资源(例如数据库连接)非常霍要的情况下，使用 try…finally 代码块或者 using 语句。

finally 代码块中的代码一定会执行，无论执行路径是否通过相应的 try 代码块，在 using 代码块的结束处，会调用被引用对象的 Dispose()方法。当退出代码块时，这两种手段都保证了清理代码的执行，即使退出的方式是 throw 异常也是如此。

看看下面使用了一个数据库连接的代码。

```
SqlConnection connection=
                new SqlConnection(connectionString);
 connection.Open();
…
connection.Dispose();
```

释放数据库资源是很重要的，即使发生了异常改变了控制流也是如此。为了确保这一点，Open(}与 Dispose()可以在 try. . . finally 块中包装起来。

```
SqlConnection connection=
                new SqlConnection(connectionString);
 try
 {
      connection.Open();
…
}
Finally
{
      connection.Dispose();
}
```

可以用 using 代码块编写更为简洁的等价逻辑。

```
  using(SqlConnection connection=
                new SqlConnection(connectionString))
  {
      connection.Open();
      ….
```

```
}
```

以上任何一种方法都可以确保 SqlConnection 对象上的 Dispose（）方法会被调用，即便代码块内部出现了异常也是如此。

151.尽可能抛出最具体的异常

在抛出异常时，尽量采用最具体的异常类，这样就可以为异常处理函数提供有助于恰当处理异常的额外信息。不要抛出 baseException 类中定义的异常，要考虑使用 NullReferenceException、ArgumentOutOfException 和 FileNotFoundException 等更具说明性的类，或者从 ApplicationException 类派生的自定义异常类。参见规则 156 和规则 157。

152.只捕获你能处理的异常

只在你的代码要对异常进行具体操作的情况下捕获异常，否则，让其他异常处理机制有机会捕获异常。如果你的处理机制仅应付特定情况（例如没找到文件），就只捕获特定的异常类（比如 FileNotFoundException），而不要使用通用的 try(Exception e)。参见规则 156。

153．如果在 catch 代码块中抛出新异常，不要丢弃异常信息

在 catch()代码块中抛出新异常时，用你的异常扩展原异常中提供的信息。出于这个目的，Exception 类提供了 InnerException 属性，以及接受一个内部 Exception 作为参数的构造函数。如果忽略掉捕获到的异常提供的信息，你可能丢弃了可能对上一层有价值的信息(参见规则 152)。

154.依异常类型特殊性级别排列 catch 块

异常发生时，系统按 Catch 块在应用程序中的出现顺序查找有关的 catch 块，直至找到第一个处理该异常的 catch 块。如果 catch 块指定了异常的类型或从该类型派生的类型，就能处理这些异常，所以小心类似下面的代码：

```
 catch(Exception )
 {
   // 这里所有异常会被捕捉….

 }
catch(ApplicationException )
{
    //绝不可能到这里，因为较常见情况首先将匹配。
}
```

155．不要在 finally 块中抛出异常

从 finally 块中抛出异常会导致同时激活多个异常，很难为这种情况做正确设计。

156.仅在认为调用者会做特别处理时才创建自定义异常

自定义异常类的目的是为特殊异常处理函数提供一种机制。如果你预料到不同类别的异常将以相当不同的方式处理(例如，ArithmeticException).，就为每种情况提供一种自定义异常类。然而，为每种特定异常提供自定义类可能做过了头，不如用合适的诊断信息批注异常实例。

157．从 ApplicationException 而不是 Exception 派生自定义异常

如果要定义新异常类。就从 ApplicationException 而不是 Exception 派生。这一约定用于区分应用程序抛出的异常和系统抛出的异常。

158．使用内建的调试类调试代码

Dubug 类提供了许多有用的方法打印诊断信息和检查逻辑。这些调用仅在调试版中可用，所以，使用该类能提供一种让代码更健壮、且不会影响到发布版本性能和代码规模的机制。

6.13 事件、委托和线程

159. 使用 lock()而不是 Monitor.Enter()

lock 语句提供了一种获取对象互斥锁的捷径，且能确保锁会被正确释放。语句

```
lock( x) …
```

等价于

```
System.Threading.Monitor.Eneter(x);
try
{
      ….
}
finally
{
   System.Threading.Monitor.Exit(x);
}
```

参见规则 150。

160.只锁定私有对象

一般来说，不要锁定代码控制范围之外的公共类型或实例。最好只锁定类的私有对象实例。

第七章

打包

包（package）可能是单个源文件、包含数个相关类的命名空间或者包含一个或多个命名空间的程序集。

7.1 文件

161．在单个文件中放置命名空间作用域内的每个元素

用其容纳的元素名为文件命名，大小写也要一致。这样做有几个好处。用更高级别的粒度管理元素经常能简化打包策略。不用依赖工具也能容易地定位实现的位置。多个开发组成贝之间源代码签出、编辑、合并导致的文件冲突能被减少。最后要注意，这符合多数 C++ 和 Java 程序员的做法，因此这可能在混合语言编程环境中成为优势。

162.用元素名作为文件名

通常，在每个文件中只放置一个公共类，而且用该类的名称为文件命名（Classname.cs）。在现代开发环境中寻找类、浏览文件较为容易。尽管如此，让类与文件名保持一致仍然可以让开发组工作起来更为容易，特别对于大开发组更是如此。

7.2 命名空间

163．不要污染框架命名空间

不要在 System 命名空间下创建类型。代码的用户会感到迷惑，而且会导致与未来.NET 中的类相冲突。

164.为每个命名空间创建单独的目录

在命名空间和目录之间维持一一对应关系。这种简单的组织结构使得查找类和命名空间变得非常容易。

165．将常被共同使用、修改和发布或互相依物的类型放到同一个命名空间下

本规则包括多个相关的打包设计原则，这些原则最初由 Robert Martin 提出。

共同重用原则

包由一起重用的类组成。如果使用包中的一个类，则也使用了其他类。

将经常一起使用的类和接口放到同一个包。这些类联系非常紧密，通常不可能只用

其中一个而不涉及其他。紧密相关类型的例子包括：

容器和迭代器。
数据库表、行和列。
日历、日期和时间。
点、线和多边形。

共同封闭原则

包中的类对同一种修改关闭。对包的修改影响到包中所有类。

将很可能为了同样原因同时修改的类放到同一个包中。如果两个类太过相关，对其中一个的修改会影响另一个，则将它们放到一个包中。

重用发布等价原则

重用单元就是版本发布单元。有效的重用需要跟踪来自变更控制系统的版本发布。该包是重用和版本发布的有效单元。

将每个类都作为一个版本发布单元对待不太实际。典型的应用程序可能包括数十个或数百个类，所以逐类发布代码使得集成及测试过程极大复杂化了，而且还极大地增加了软件内部的总体改动率。

包应为发布多个类和接口提供更为方便的机制。包中每个类或接口在不同版本之间可能经历多次独立修订，但包发布版本仅采集每个类和接口的最新版本。将包作为版本发布和分发的基础单元。

无环依赖原则

包之间的依赖结构必须是一个有向元环图，依赖结构中不能有环。

如果两个包直接或间接地互相依赖，就不能只独立发布其中一个，因为对其中一个的修改经常会导致修改另外一个。这种循环依赖极大地增加了系统脆弱性，而且可能抵消通过将每个包分配给单个开发人员或开发组而获得的开发日程缩短的努力。

逐步消除循环依赖，可以合并相互依赖的包，也可以引入新的包抽象，让原来的两个包依赖它，而不再互相依赖。

166.在分开的程序集中隔离不稳定的类

避免将不稳定的类和稳定的类放到同一个包中。如果把包作为发布和分发的主要单元，那么用户只有在你重新发布整个包时才能访问到对不稳定类的修改。每次发布包，用户都必须承担双新集成、重新测试包中所有类的成本，尽管其中许多类并未改动过。

把不稳定的类和稳定的类分开，减少了新代码版本发布带来的影响，从而也减少了对代码用户的影响。

167.最大化抽象以最大化稳定性

本规则来自下列设计原则。

稳定抽象原则

包呈现的稳定性与其抽象级别直接相称。包越抽象就越倾向于稳定。包越具体就越倾向于不稳定。

使用稳定的抽象来创建稳定的包。在类和接口中采用高级、稳定的概念，使用具体类提供实现。将抽象类和接口从具体类中分离出来，形成稳定和不稳定的包。这能确保不稳定包中的派生类依赖于稳定包中的抽象类和接口。

168．将高级设计和架构捕获为稳定抽象，组织到稳定命名空间中

要成功规划和管理软件开发工作，顶层设计必须快速稳定下来，而且保持稳定。如果系统架构不断改动，则任何开发经理也没希望精确规划、评估、调度和分配资源。

商层架构设计完成后，用包隔开设计中稳定的部分和易变动的实现。创建放置设计中的高层抽象的包。将这些抽象的详细实现放到单独的包中，这些包依赖于高层抽象包。

7.3 程序集

169．让程序集和命名空间的名字保持一致

使用相同的名字可以确保程序集名的唯一性。在增加引用时，程序集的内容对于用户也是透明的。

170.避免让难以修改的程序集依赖于易于修改的程序集

本规则来自下列设计原则。

稳定依赖原则

包之间的依赖关系应导向稳定性增加。包只应依赖于比自身稳定的包。

如果包括难以修改的类型的包依赖于包括容易活有可能修改的类型的包，则被依赖的包会有效地阻止对易变包的修改。

在软件系统中，特别是增量开发的软件系统中，有些包总是有些易变，这类系统的开发人员必须要能方便地修改或扩展这些易变的包，完成系统的实现，而且不必太过担心下游效应（downstream effect）。

不要创建依赖于不稳定包的包。如果必须这么做，创建新包，反转稳定代码和不稳定代码之间的关系。

171.手工增加程序集版本号

.NET 运行时用程序集版本号判别两种类型足否匹配。手工增加

AssemblyVersionAttribute，而不要依赖默认的版本号方案。这样就能执行机构的构建策略。例如指定某次发布是否主要版本、小版本、维护版本或工程版本。

172．只把单个类暴露给 COM，不暴露整个程序集

只暴露必要的部分。在程序集层次没定 ComVisible 特性为 false，暴露单个类。

```
[ComVisible(true)]
class MyClass
{
    ….
}
```

173.将有不安全代码的类放到单独程序集中

有些环境禁止使用不安全代码，如果把不安全代码隔离到单独的程序集中，就还能继续使用其他程序集。

174.静态链接本地代码

有时需要使用本地代码。一种做法是通过 DllImport 特性引用本地动态库（DLL）。这意味着部署.NET 程序集时必须包括被引用的 DLL。然而，DLL 不能被放到全局程序集缓存中，而必须放在用户路径的目录中。

在某些情况下，更好的方案是创建一个受管 C++程序集，链入本地静态库（LIB）。结果程序集能和其他.NET 程序集一样地部署。