**SCB代码规范——CodeReview指南**

**[一：宗旨](#_Toc30137)** [4](#_Toc30137)

**[二：环境与工具介绍](#_Toc11081)** [4](#_Toc11081)

**[1. Vs工具](#_Toc8269)** [4](#_Toc8269)

**[1.1 使用Tab作为缩进，并设置缩进大小为4](#_Toc18518)** [4](#_Toc18518)

**[1.2 代码使用半展开](#_Toc8815)** [5](#_Toc8815)

**[2. ReSharper](#_Toc10032)** [6](#_Toc10032)

**[三：术语定义](#_Toc697)** [6](#_Toc697)

**[1. Pascal 大小写](#_Toc2539)** [6](#_Toc2539)

**[2. Camel 大小写](#_Toc24932)** [6](#_Toc24932)

**[3. 匈牙利命名法](#_Toc9347)** [6](#_Toc9347)

**[3.1属性部分](#_Toc3188)** [6](#_Toc3188)

**[3.2类型部分](#_Toc22242)** [7](#_Toc22242)

**[3.3描述部分](#_Toc28231)** [7](#_Toc28231)

**[3.4举例说明](#_Toc3005)** [7](#_Toc3005)

**[四：代码规范](#_Toc29020)** [7](#_Toc29020)

**[1.整体结构](#_Toc2992)** [7](#_Toc2992)

**[1.1工程及命名空间分级区分](#_Toc2959)** [7](#_Toc2959)

**[1.2所有命名空间、类型名称使用Pascal风格（单词首字母大写）](#_Toc19982)** [7](#_Toc19982)

**[1.3类型名称和源文件名称必须一致](#_Toc29740)** [7](#_Toc29740)

**[1.4一个.cs源文件至多定义三个类型](#_Toc23857)** [7](#_Toc23857)

**[1.5 合理的访问控制](#_Toc25463)** [8](#_Toc25463)

**[2.代码外观](#_Toc21075)** [8](#_Toc21075)

**[2.1列宽](#_Toc28578)** [8](#_Toc28578)

**[2.2换行](#_Toc19773)** [8](#_Toc19773)

**[2.3缩进](#_Toc31303)** [8](#_Toc31303)

**[2.4空行（重要点）](#_Toc11846)** [8](#_Toc11846)

**[2.5空格](#_Toc31457)** [8](#_Toc31457)

**[2.6括号](#_Toc19837)** [- () 9](#_Toc19837)

**[2.7花括号 - {}](#_Toc5590)** [9](#_Toc5590)

**[3. 程序注释](#_Toc32754)** [9](#_Toc32754)

**[3.1注释概述](#_Toc11540)** [9](#_Toc11540)

**[3.2文档型注释](#_Toc1148)** [10](#_Toc1148)

**[3.3类c注释](#_Toc15949)** [10](#_Toc15949)

**[3.4单行注释](#_Toc31595)** [10](#_Toc31595)

**[3.5声明](#_Toc17492)** [11](#_Toc17492)

**[3.6类和接口的声明](#_Toc26881)** [11](#_Toc26881)

**[3.7字段的声明](#_Toc15405)** [11](#_Toc15405)

**[4. 命名规范](#_Toc187)** [12](#_Toc187)

**[4.1命名概述](#_Toc13260)** [12](#_Toc13260)

**[4.2命名原则](#_Toc10141)** [12](#_Toc10141)

**[4.3大小写规则](#_Toc14228)** [12](#_Toc14228)

[4.3.1大写 12](#_Toc19999)

[4.3.2缩写 13](#_Toc23834)

**[4.4命名空间](#_Toc1664)** [13](#_Toc1664)

**[4.5类](#_Toc23740)** [13](#_Toc23740)

**[4.6接口](#_Toc8820)** [14](#_Toc8820)

**[4.7属性类 (Attribute)](#_Toc30969)** [14](#_Toc30969)

**[4.8枚举 (Enum)](#_Toc15248)** [14](#_Toc15248)

**[4.9参数](#_Toc10692)** [14](#_Toc10692)

**[4.10方法](#_Toc25580)** [15](#_Toc25580)

**[4.11属性 (property)](#_Toc29469)** [15](#_Toc29469)

**[4.12事件](#_Toc26813)** [15](#_Toc26813)

**[4.13常量 (const)](#_Toc18867)** [16](#_Toc18867)

**[4.14字段](#_Toc7941)** [16](#_Toc7941)

**[4.15静态字段](#_Toc15711)** [17](#_Toc15711)

**[4.16集合](#_Toc12942)** [17](#_Toc12942)

**[4.17措词](#_Toc8378)** [18](#_Toc8378)

**[4.18类型成员的排列顺序](#_Toc29116)** [18](#_Toc29116)

**[5. 语句](#_Toc24090)** [18](#_Toc24090)

**[5.1每行一个语句](#_Toc9472)** [18](#_Toc9472)

**[5.2复合语句](#_Toc28089)** [19](#_Toc28089)

**[5.3 return 语句](#_Toc14077)** [19](#_Toc14077)

**[5.4 if、 if-else、if else-if 语句](#_Toc15829)** [19](#_Toc15829)

**[5.5 for、foreach 语句](#_Toc23185)** [19](#_Toc23185)

**[5.6 while 语句](#_Toc9477)** [20](#_Toc9477)

**[5.7 do - while 语句](#_Toc1631)** [20](#_Toc1631)

**[5.8 switch - case 语句](#_Toc30754)** [20](#_Toc30754)

**[5.9 try - catch 语句](#_Toc6552)** [21](#_Toc6552)

**[5.10 using 块语句](#_Toc26729)** [21](#_Toc26729)

**[5.11调用类型内部其他成员，需加this；调用父类成员，需加base](#_Toc23566)** [21](#_Toc23566)

**[6. 控件命名规则](#_Toc13253)** [21](#_Toc13253)

**[6.1命名方法](#_Toc23511)** [21](#_Toc23511)

**[6.2主要控件名简写对照表](#_Toc16490)** [21](#_Toc16490)

**[7. 其他](#_Toc32008)** [22](#_Toc32008)

**[7.1表达式](#_Toc17945)** [22](#_Toc17945)

**[7.2类型转换](#_Toc29779)** [22](#_Toc29779)

**[8. 代码扩展性，复用性](#_Toc31439)** [22](#_Toc31439)

**[8.1合理使用设计模式，进行分层设计](#_Toc27533)** [22](#_Toc27533)

**[8.2 避免使用大文件](#_Toc9648)** [22](#_Toc9648)

**[8.3 避免大方法/函数](#_Toc18276)** [22](#_Toc18276)

**[9. XHTM代码风格要求](#_Toc21089)** [23](#_Toc21089)

**[9.1 如果XHTML标记有层次，那么代码也要有层次](#_Toc32720)** [23](#_Toc32720)

**[9.2 所有标记必须闭合](#_Toc28730)** [23](#_Toc28730)

**[9.3 如果标记中间代码超过20行，则应在标记末尾加注标识](#_Toc18696)** [23](#_Toc18696)

**[10. CSS代码风格要求](#_Toc17392)** [23](#_Toc17392)

**[10.1 代码使用半展开](#_Toc30963)** [24](#_Toc30963)

**[10.2 使用Tab作为缩进，并设置缩进大小为4](#_Toc1960)** [24](#_Toc1960)

**[10.3 代码注释](#_Toc13692)** [24](#_Toc13692)

**[10.4 代码编写](#_Toc13462)** [24](#_Toc13462)

**[10.5 嵌入式样式的比例不超过样式表代码总量的10%](#_Toc6706)** [24](#_Toc6706)

**[10.6 内联式样式的比例不超过样式表代码总量的30%](#_Toc2705)** [24](#_Toc2705)

**[10.7 外联式样式表的比例不少于样式表代码总量的60%](#_Toc14350)** [24](#_Toc14350)

**[11. JavaScirpt代码风格要求](#_Toc19776)** [25](#_Toc19776)

**[11.1 代码使用半展开](#_Toc10335)** [25](#_Toc10335)

**[11.2 使用Tab作为缩进，并设置缩进大小为4](#_Toc11220)** [25](#_Toc11220)

**[11.3 代码注释](#_Toc29741)** [25](#_Toc29741)

**[11.4 不得出现内嵌式代码](#_Toc12137)** [25](#_Toc12137)

**[11.5 内联式代码占JavaScript的总量不得超过40%](#_Toc23438)** [25](#_Toc23438)

**[11.6 外联式代码占JavaScript的总量至少为60%](#_Toc11647)** [25](#_Toc11647)

**[12. 待补充](#_Toc9410)** [25](#_Toc9410)

**[五：性能优化及开发注意（\*重点\*）](#_Toc26409)** [25](#_Toc26409)

**[1. 代码基础及细节](#_Toc6417)** [26](#_Toc6417)

**[1.1 慎用异常捕捉机制抛出异常到客服端，改用bool或者异常代号返回](#_Toc751)** [26](#_Toc751)

**[1.2 绝对不要在循环中使用try-Catch](#_Toc17859)** [26](#_Toc17859)

**[1.3 不要在循环内部使用New操作来创建变量](#_Toc15983)** [26](#_Toc15983)

**[1.4 循环中的公共计算应该剥离到循环外](#_Toc2471)** [26](#_Toc2471)

**[1.5 减少重复代码](#_Toc24121)** [26](#_Toc24121)

**[1.6 减少不必要的初始化](#_Toc31748)** [27](#_Toc31748)

**[1.7 不要迷信正则表达式，使用后要做测试。](#_Toc21898)** [27](#_Toc21898)

**[1.8 字符串，非常常用](#_Toc24560)** [27](#_Toc24560)

**[1.8.1 字符串判断](#_Toc11465)** [27](#_Toc11465)

**[1.8.2字符串拼接](#_Toc12113)** [28](#_Toc12113)

**[2. 合理使用缓存](#_Toc9455)** [28](#_Toc9455)

**[2.1访问量大的接口或功能，采用索引或缓存](#_Toc25380)** [28](#_Toc25380)

**[2.2使用ICache接口（Foundation中有该接口，采用Memcache实现）](#_Toc18249)** [28](#_Toc18249)

**[2.3内存缓存，需了解服务器内存使用情况](#_Toc2747)** [28](#_Toc2747)

**[2.4 常用表的数据，若对数据变化实时性要求不高，可简单缓存](#_Toc19510)** [28](#_Toc19510)

**[3. 合理使用数据结构](#_Toc4613)** [29](#_Toc4613)

**[3.1 合理使用Dictionary](#_Toc22725)** [29](#_Toc22725)

**[3.2 合理使用Lookup](#_Toc22319)** [31](#_Toc22319)

**[3.3 使用其他，如HashSet等](#_Toc21093)** [32](#_Toc21093)

**[4.关于数据库方面](#_Toc29202)** [32](#_Toc29202)

**[4.1数据持久化](#_Toc16652)** [32](#_Toc16652)

**[4.2优化读取数据库的语句](#_Toc25677)** [32](#_Toc25677)

**[4.2.1去除重复读写同一数据的过程](#_Toc15217)** [33](#_Toc15217)

**[4.2.2 Linq下，多表联合查询与分开单表查询性能](#_Toc24318)** [33](#_Toc24318)

**[4.2.3 mongodb客户端并不支持所有linq写法](#_Toc30297)** [33](#_Toc30297)

**[4.2.4给表建立合适的索引](#_Toc29558)** [33](#_Toc29558)

**[5.避免情况](#_Toc5890)** [33](#_Toc5890)

**[5.1程序搜索大文件夹下的某文件](#_Toc28256)** [33](#_Toc28256)

**一：宗旨**

**简洁美观，逻辑清晰，性能高效，方便阅读理解，易于交流维护。**

**二：环境与工具介绍**

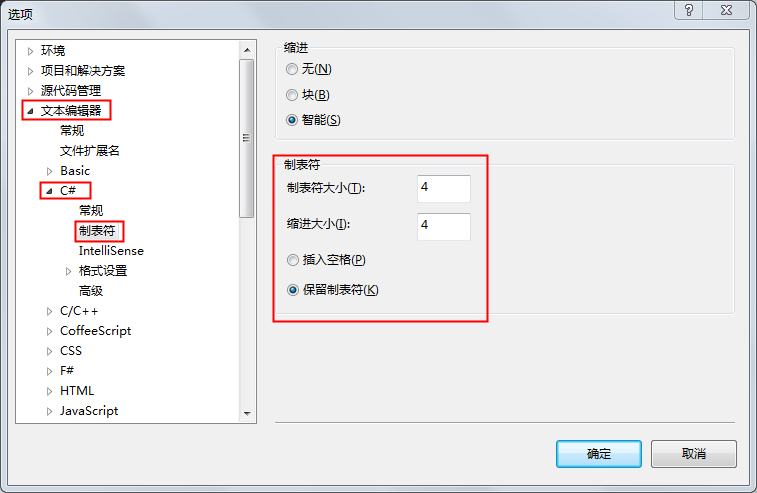
**1. Vs工具**

**1.1 使用Tab作为缩进，并设置缩进大小为4**

第一步，打开Visual Studio，进入“工具”，“选项...”，如下图所示：



第二步，进入“文本编辑器”，“C#”，“制表符”，如下图所示，设置制表符。



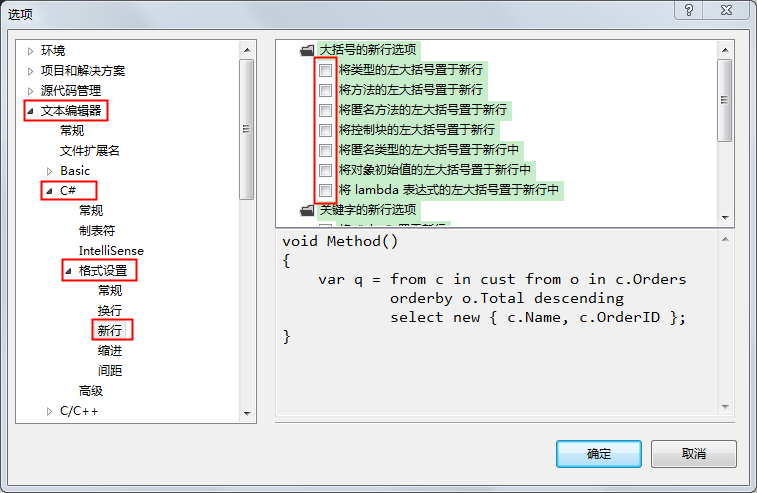
第三步，点击“确定”，完成设置。

**1.2 代码使用半展开**

第一步，打开Visual Studio，进入“工具”，“选项...”，如下图所示：



第二步，进入“文本编辑器”，“C#”，“格式设置”，“新行”，取消掉右侧所有复选框中的对号，如下图所示：



第三步，点击“确定”，完成设置。

1. **ReSharper**

ReSharper是一个JetBrains公司出品的著名的代码生成工具，其能帮助Microsoft Visual Studio成为一个更佳的IDE。它包括一系列丰富的能大大增加C#和Visual Basic .net开发者生产力的特征。使用ReSharper，你可以进行深度代码分析，智能代码协助，实时错误代码高亮显示，解决方案范围内代码分析，快速代码更正，一步完成代码格式化和清理，业界领先的自动代码重构，高级的集成单元测试方案，和强大的解决方案内导航和搜索。实质上，ReSharper特征可用于C#，VB.NET，XML，ASP.NET，XAML，和构建脚本。ReSharper还为C#和VB.NET提供了增强的交叉语言功能，它使开发者可以有效的控制.net混合项目。

这是一款非常好的代码编写辅助工具，有兴趣的同事请移步学习！

地址：http://www.cnblogs.com/luminji/p/3285505.html

**三：术语定义**

**1. 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：BackColor

**2. 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：backColor

**3. 匈牙利命名法**

匈牙利命名法是一名匈牙利程序员发明的，而且他在微软工作了多年。此命名法就是通过微软的各种产品和文档传出来的。多数有经验的程序员，不管他们用的是哪门儿语言，都或多或少在使用它。

这种命名法的基本原则是：**变量名＝属性＋类型＋对象描述**

即一个变量名是由三部分信息组成，这样，程序员很容易理解变量的类型、用途，而且便于记忆。

下边是一些推荐使用的规则例子，你可以挑选使用，也可以根据个人喜好作些修改再用之。

**3.1属性部分**

全局变量： g\_

常量 ： c\_

类成员变量： m\_

**3.2类型部分**

指针： p

句柄： h

布尔型： b

浮点型： f

无符号： u

**3.3描述部分**

初始化： Init

临时变量： Tmp

目的对象： Dst

源对象： Src

窗口： Wnd

**3.4举例说明**

hwnd： h表示句柄，wnd表示窗口，合起来为“窗口句柄”。

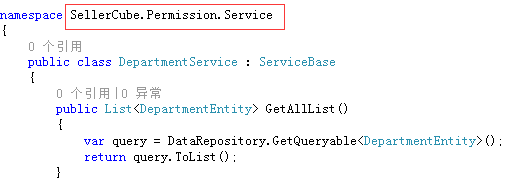
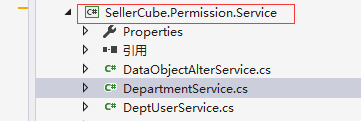
m\_bFlag： m表示成员变量，b表示布尔，合起来为：“某个类的成员变量，布尔型，是一个状态标志”。

**四：代码规范**

**1.整体结构**

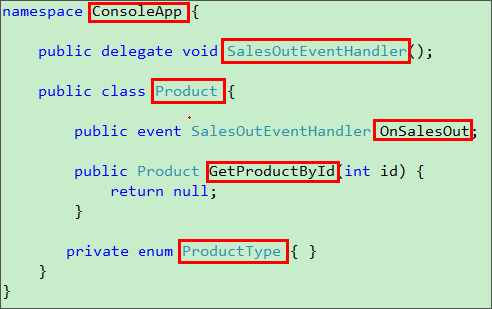
**1.1工程及命名空间分级区分**

工程命名：一级使用：SCB/SellerCube；二级使用：业务系统/模块名称；三级：功能名称；四级：设计名称...



**1.2所有命名空间、类型名称使用Pascal风格（单词首字母大写）**

如下图所示，红色标记的为使用Pascal风格的类型：



**1.3类型名称和源文件名称必须一致**

当类型命名为Product时，其源文件命名只能是Product.cs。

**1.4一个.cs源文件至多定义三个类型**

如果两个类型的关系是紧密相关的，比如 产品、产品类型，此时Product类，和ProductType枚举可以定义在同一个Product.cs文件中。

但不能在一个.cs文件中出现两个不相关的类型定义，例如将 Product类和Reseller类（分销商）定义在一个BasicInfo.cs文件中。

**1.5合理的访问控制**

给类，方法，属性，变量添加适当的访问控制，而不是所有都是 public

**2.代码外观**

**2.1列宽**

代码列宽控制在120字符左右。

**2.2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：

在逗号后换行。

在操作符前换行。

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。

**2.3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 设置：工具->选项->文本编辑器->C#->制表符->插入空格，制表符大小=4，缩进大小=4

**2.4空行（重要点）**

**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

接口和类的定义之间。

枚举和类的定义之间。

类与类的定义之间。

在以下情况下使用一个空行

方法与方法、属性与属性之间。

方法中变量声明与语句之间。

方法与方法之间。

**方法中不同的逻辑块之间**。

方法中的返回语句与其他的语句之间。

属性与方法、属性与字段、方法与字段之间。

注释与它注释的语句间不空行，但与其他的语句间空一行。

**2.5空格**

在以下情况中要使用到空格

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。

多个参数用逗号隔开，每个逗号后都应加一个空格。

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");

语句中的表达式之间用空格隔开。如：

for (expr1; expr2; expr3)

**2.6括号** - ()

左括号“(”不要紧靠关键字，中间用一个空格隔开。

左括号“(”与方法名之间不要添加任何空格。

没有必要的话不要在返回语句中使用()。

**2.7花括号 - {}**

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)

{

}

左花括号 “{” 要与相应的右花括号 “}”对齐。

通常情况下左花括号 “{”单独成行，不与任何语句并列一行。

if、while、do语句后一定要使用{}，即使{}号中为空或只有一条语句。如：

if (somevalue == 1)

{

somevalue = 2;

}

右花括号 “}” 后建议加一个注释以便于方便的找到与之相应的 {。如:

while(1)

{

if(valid)

{

} // if valid

else

{

} // not valid

} // end forever

1. **程序注释**

**3.1注释概述**

修改代码时，总是使代码周围的注释保持最新。

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

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。

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。

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。

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。

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。尽一切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。

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。

避免多余的或不适当的注释，不应包含个人情绪内容，如幽默的不主要的备注。

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。

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。

注释代码中不十分明显的内容。

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。

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。

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。

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。

代码修改变更记录不应使用注释标明修改日期和修改人，注释应只针对代码不记录版本，代码版本应该使用代码版本系统进行管理

为了使层次清晰，在闭合的右花括号后注释该闭合所对应的起点。

namespace SCB.Framework.Web

{

} // namespace SCB.Framework.Web

**3.2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、字段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。如

///MyMethod is a method in the MyClass class.

///Here's how you could make a second paragraph in a description.

////// for information about output statements.

/////////public static void MyMethod(int Int1)

{

}

注释标签的使用请参考:http://msdn.microsoft.com/zh-cn/library/5ast78ax.aspx

**3.3类c注释**

该类注释用于复杂程序逻辑说明与技术事项

用法

/\*

动态路由算法使用Round-robin算法，原理是...

\*/

**3.4单行注释**

该类注释用于

方法内的代码注释。如变量的声明、代码或代码段的解释。例：

//

// 注释语句

//

private int number;

或

// 注释语句

private int number;

方法内变量的声明或花括号后的注释, 例：

if (1 == 1) // always true

{

statement;

} // always true

**3.5声明**

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 level; // 推荐

int size; // 推荐

int x, y; // 不推荐

初始化：建议在变量声明时就对其做初始化。

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：

void MyMethod()

{

int int1 = 0;

if (condition)

{

int int2 = 0;

...

}

}

例外情况

for (int i = 0; i < maxLoops; i++) { ... }

应避免不同层次间的变量重名，如：

int count;

...

void MyMethod()

{

if (condition)

{

int count = 0; // 避免

...

}

...

}

**3.6类和接口的声明**

在方法名与其后的左括号间没有任何空格。

左花括号 “{” 出现在声明的下行并与之对齐，单独成行。

方法间用一个空行隔开。

**3.7字段的声明**

不要使用是 public 或 protected 的实例字段。如果避免将字段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字段不能被更改为属性。考虑为字段提供 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字段的正确使用。 例：

public class Control: Component

{

private int handle;

public int Handle

{

get

{

return handle;

}

}

}

1. **命名规范**

**4.1命名概述**

名称应该说明“什么”而不是“如何”。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()。

**4.2命名原则**

选择正确名称时的困难可能表明需要进一步分析或定义项的目的。使名称足够长以便有一定的意义，并且足够短以避免冗长。唯一名称在编程上仅用于将各项区分开。表现力强的名称是为了帮助人们阅读；因此，提供人们可以理解的名称是有意义的。不过，请确保选择的名称符合适用语言的规则和标准。

以下几点是推荐的命名方法：

避免容易被主观解释的难懂的名称，如方面名AnalyzeThis()，或者属性名xxK8。这样的名称会导致多义性。

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。

只要合适，在变量名的末尾或开头加计算限定符（Avg、Sum、Min、Max、Index）。

在变量名中使用互补对，如min/max、begin/end和open/close。

布尔变量名应该包含Is，这意味着Yes/No或True/False值，如fileIsFound。

在命名状态变量时，避免使用诸如Flag的术语。状态变量不同于布尔变量的地方是它可以具有两个以上的可能值。不是使用documentFlag，而是使用更具描述性的名称，如documentFormatType。

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。仅对于短循环索引使用单字母变量名，如 i 或 j。 可能的情况下，尽量不要使用原义数字（幻数）或原义字符串，如for (int i = 1; i < 7; i++)。而是使用命名常数，如for (int i = 1; i < NUM\_DAYS\_IN\_WEEK; i++)，以便于维护和理解。

**4.3大小写规则**

4.3.1大写

组织名缩写使用大写

两个或者更少字母组成的标识符使用大写。例：

System.IO

System.Web.UI

SCB.Framework.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 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

属性 Pascal BackColor

公共实例字段 Pascal RedValue

注意: 应优先使用属性。

受保护的实例字段 Camel redValue

注意: 应优先使用属性。

私有的实例字段 Camel redValue

参数 Camel typeName

方法内的变量 Camel backColor

4.3.2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。

**4.4命名空间**

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：

CompanyName.TechnologyName[.Feature][.Design]

例如：

namespace SCB.SupplierChain // 赛酷比公司的供应链系统

namespace SCB.SupplierChain.DataRules // 赛酷比公司的供应链系统的业务规则模块

命名空间使用Pascal大小写。

TechnologyName指的是该项目的英文缩写或软件名。

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。

**4.5类**

使用 Pascal 大小写。

用名词或名词短语命名类。

使用全称避免缩写，除非缩写已是一种公认的约定，如URL、HTML

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。

public class FileStream

public class Button

public class String

**4.6接口**

用名词或名词短语，或者描述行为的形容词命名接口。例：

接口名称IComponent使用描述性名词

接口名称ICustomAttributeProvider使用名词短语

接口名称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 (\_)。

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.

}

**4.7属性类 (Attribute)**

应该总是将后缀 Attribute 添加到自定义属性类。例：

public class ObsoleteAttribute

{

}

**4.8枚举 (Enum)**

枚举 (Enum) 值类型从 Enum 类继承。

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。

**4.9参数**

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。

不要给参数名称加匈牙利语类型表示法的前缀，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。

Type GetType(string typeName)

string Format(string format, object args)

**4.10方法**

使用动词或动词短语命名方法。

使用 Pascal 大小写。

RemoveAll()

GetCharArray()

Invoke()

如果方法返回的类型是bool类型，则其前缀为Is，例如：IsHidden。

如果某个属性的类型为bool类型，则其前缀为Can，例如：CanHidden。

**4.11属性 (property)**

使用名词或名词短语命名属性。

使用 Pascal 大小写。

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。

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。

public enum Color

{

// Insert code for Enum here.

}

public class Control

{

public Color Color

{

get

{

// Insert code here.

}

set

{

// Insert code here.

}

}

}

**4.12事件**

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。

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

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;

}

}

}

**4.13常量 (const)**

所有单词大写，多个单词之间用 "\_" 隔开。 如：

public const string PAGE\_TITLE = "Welcome";

**4.14字段**

private、protected 使用 Camel 大小写。

public 使用 Pascal 大小写。

拼写出字段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。

class SampleClass

{

string url;

string destinationUrl;

}

不要对字段名使用匈牙利语表示法。好的名称描述语义，而非类型。

不要对字段名或静态字段名应用前缀。具体说来，不要对字段名称应用前缀来区分静态和非静态字段。例如，应用g\_或s\_前缀是不正确的。

不能出现公有字段，如果需要公有字段，使用属性进行包装。

对预定义对象实例使用公共静态只读字段。如果存在对象的预定义实例，则将它们声明为 对象本身的公共静态只读字段。使用 Pascal 大小写，原因是字段是公共的。

public struct Color

{

public static readonly Color Red = new Color(0x0000FF);

public Color(int rgb)

{

// Insert code here.

}

public Color(byte r, byte g, byte b)

{

// Insert code here.

}

public byte RedValue

{

get

{

return Color;

}

}

}

**4.15静态字段**

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。

建议尽可能使用静态属性而不是公共静态字段。

**4.16集合**

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。

凡符合下表所列的集合类型，应添加相应的后缀。



**4.17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，避免使用与C#语言关键字冲突的标识符。

**4.18类型成员的排列顺序**

类型成员的排列顺序自上而下依次为：

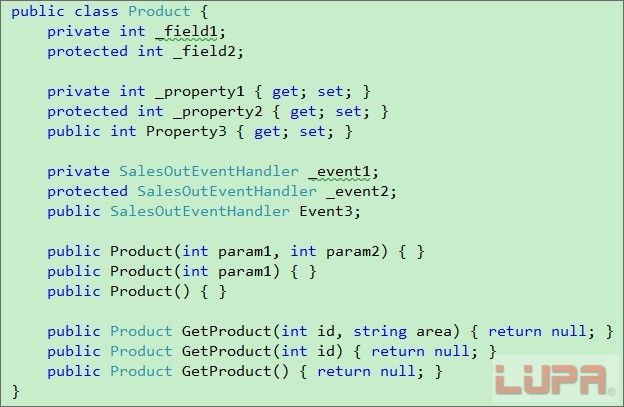
字段：私有字段、受保护字段

属性：私有属性、受保护属性、公有属性

事件：私有事件、受保护事件、公有事件

构造函数：参数数量最多的构造函数，参数数量中等的构造函数，参数数量最少的构造函数

方法：重载方法的排列顺序与构造函数相同，从参数数量最多往下至参数最少



1. **语句**

**5.1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

**5.2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:

子语句要缩进。

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。

即使只有一条子语句要不要省略花括号“ {}”。 如：

while(d += s++)

{

n++;

}

**5.3 return 语句**

return语句中不使用括号，除非它能使返回值更加清晰。如：

return;

return myDisk.size();

return (size ? size : defaultSize);

**5.4 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;

}

**5.5 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); // update user id

foreach 语句使用格式

foreach (object obj in array)

{

statements;

}

注意

在循环过程中不要修改循环计数器。

对每个空循环体给出确认性注释。

**5.6 while 语句**

while 语句使用格式

while (condition)

{

statements;

}

空的 while 语句使用格式

while (condition);

**5.7 do - while 语句**

do - while 语句使用格式

do

{

statements;

} while (condition);

**5.8 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：

语句switch中的每个case各占一行。

为所有switch语句提供default分支。

所有的非空 case 语句必须用 break; 语句结束。

**5.9 try - catch 语句**

try - catch语句使用格式

try

{

statements;

}

catch (ExceptionClass e)

{

statements;

}

finally

{

statements;

}

**5.10 using 块语句**

using 块语句使用格式

using (object)

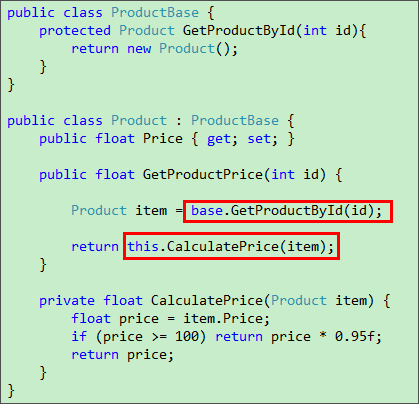
{

statements;

}

**5.11调用类型内部其他成员，需加this；调用父类成员，需加base**

示例代码如下：



1. **控件命名规则**

**6.1命名方法**

控件名简写+英文描述，英文描述首字母大写

**6.2主要控件名简写对照表**

控件名 => 简写

Label => lbl

TextBox => txt

Button => btn

LinkButton => lnkbtn

ImageButton => imgbtn

DropDownList => ddl

ListBox => lst

DataGrid => dg

DataList => dl

CheckBox => chk

CheckBoxList => chkls

RadioButton => rdo

RadioButtonList => rdolt

Image => img

Panel => pnl

Calender => cld

AdRotator => ar

Table => tbl

RequiredFieldValidator => rfv

CompareValidator => cv

RangeValidator => rv

RegularExpressionValidator => rev

ValidatorSummary => vs

CrystalReportViewer => rptvew

1. **其他**

**7.1表达式**

避免在表达式中用赋值语句

避免对浮点类型做等于或不等于判断

**7.2类型转换**

尽量避免强制类型转换。

如果不得不做类型转换，尽量用显式方式。

1. **代码扩展性，复用性**

**8.1合理使用设计模式，进行分层设计**

熟练掌握常用的设计模式，如：创建型模式的5种，组合模式中的适配器模式，代理模式，组合模式等。

使用合理的设计模式来架构你的代码层次。

**8.2 避免使用大文件**

如果一个文件里的代码超过500-600行，必须考虑将代码分开到不同的类中。

**8.3 避免大方法/函数**

原则是能够细化尽量细化，单个函数不超过100行代码。

对功能或者任务要学会拆解，避免写太长的方法，如果一个方法代码过长（暂时没有明确指出方法的行数），应该考虑将其分解为不同的方法。

一个方法只完成一个任务。不要把多个任务组合到一个方法中，即使那些任务非常小。如此，可以提高代码的复用性，也方便维护和阅读。

举例：对某字符串（有英文有中文）提取其中的英文单词，写入文件

拿到该问题，我们首先要细化子功能，从整体设计考虑，共分几个子函数。

函数1：字符串去除中文，以空格代替

函数2：对无中文的字符串进行拆分成英文单词块

函数3：对每个单词进行验证是否为标准英文单词

函数4：将得到的英文单词写入文件

切忌对这样一个问题不做任何切分，就一个函数搞定。

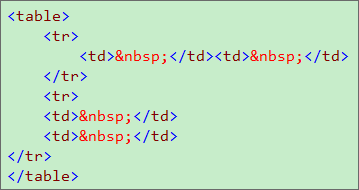
1. **XHTM代码风格要求**

**9.1 如果XHTML标记有层次，那么代码也要有层次**

下面是书写符合要求的例子：



下面是书写 不符合要求 的例子：



**9.2 所有标记必须闭合**

示例代码如下：

IMG_256

**9.3 如果标记中间代码超过20行，则应在标记末尾加注标识**

标注方式如下：



1. **CSS代码风格要求**

**10.1 代码使用半展开**

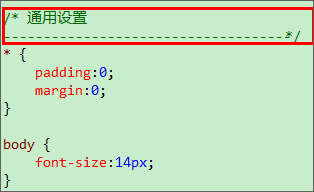
设置方法参考1.2节。

**10.2 使用Tab作为缩进，并设置缩进大小为4**

设置方法参考1.1节。

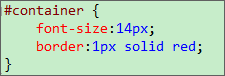
**10.3 代码注释**

注释主要说明该样式应用于页面的哪个部分，而非说明样式的应用效果，代码注释风格如下所示：



**10.4 代码编写**

每一个样式设置必须独占一行，不能位于同一行，下面是符合要求的写法：



下面是 不符合要求 的写法：

IMG_256

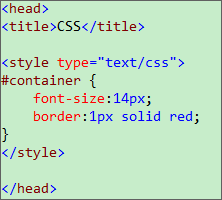
**10.5 嵌入式样式的比例不超过样式表代码总量的10%**

嵌入式样式为直接写在HTML标记内部的样式，如下图所示：

IMG_256

**10.6 内联式样式的比例不超过样式表代码总量的30%**

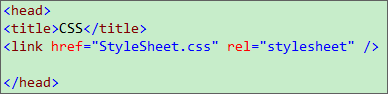
内联式样式为写在<head></head>中的样式，如下图所示：



内联式样式，不能 写在<body></body>之间。

**10.7 外联式样式表的比例不少于样式表代码总量的60%**

外联式样式表为写在.css文件中的样式，通过link引入到XHTML页面中，如下图所示：



1. **JavaScirpt代码风格要求**

**11.1 代码使用半展开**

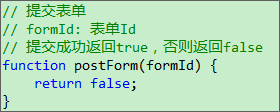
设置方法参考1.2节。

**11.2 使用Tab作为缩进，并设置缩进大小为4**

设置方法参考1.1节。

**11.3 代码注释**

代码注释需要说明“函数功能”、“入口参数”、“返回值”，注释范例如下：



其中第一行说明函数功能；第二行说明入口参数；最后一行说明返回值

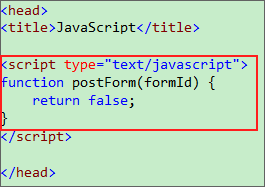
**11.4 不得出现内嵌式代码**

内嵌式代码是指写在XHTML标记中的JavaScript代码，下面的写法是 不符合要求 的：

IMG_256

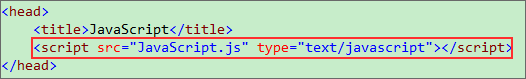
**11.5 内联式代码占JavaScript的总量不得超过40%**

内联式代码是指写在<head />或者<body />之间的代码：



**11.6 外联式代码占JavaScript的总量至少为60%**

外联式代码指写在单独的.js文件中，然后通过script标记连接到XHTML页面中的代码。



1. **待补充**

**五：性能优化及开发注意（\*重点\*）**

**优雅的代码，美观易读是基础；作为程序的本质，运行效率是根本！**

**当你觉得页面打开比较慢，而涉及数据又不多的情况下，您的代码一定有优化空间！**

1. **代码基础及细节**

**1.1 慎用异常捕捉机制抛出异常到客服端，改用bool或者异常代号返回**

经测试，下列代码，返回异常到客服端，性能下降几十倍。

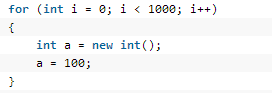
1. **class** BusinessLogcCheck
2. {
3. **public** **void** Check()
4. {
5. **try**
6. {
7. //Your validation code is here
9. }
10. **catch** (Exception ex)
11. {
12. **throw** **new** Exception("My own exception");
13. }
15. }
16. }

**1.2 绝对不要在循环中使用try-Catch**

经测试，在循环外面比在循环里面处理异常会快上2倍。

**1.3 不要在循环内部使用New操作来创建变量**

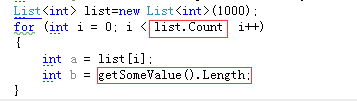
大量在循环内部new变量，对代码效率的影响非常大。例：



应尽量将New移到循环外。

**1.4 循环中的公共计算应该剥离到循环外**

经常可以看到类似如下代码，可能影响不大，但是养成一个好习惯很重要。

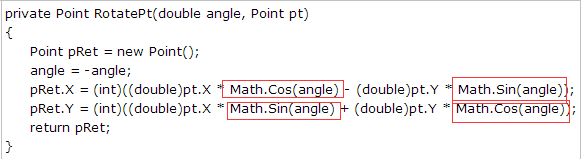


在循环中，应尽量将公有调用移到循环外。

**1.5 减少重复代码**

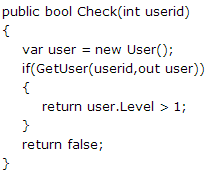
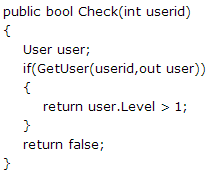
这是最基本的优化方案,尽可能减少那些重复做的事,让他们只做一次

比较常见是这种代码,同样的Math.Cos(angle) 和Math.Sin(angle)都做了2次



**1.6 减少不必要的初始化**

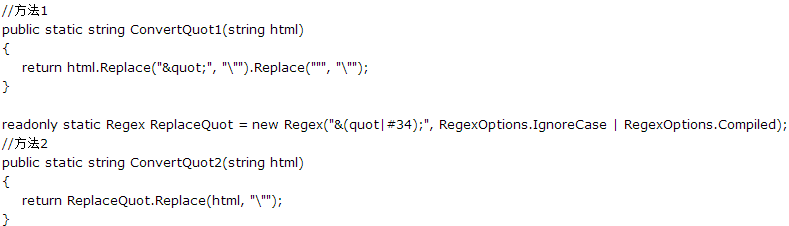
比如调用out参数之前,是不需要初始化的

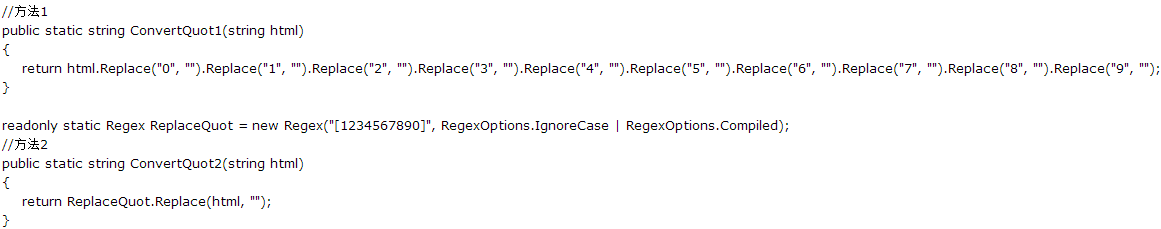
优化前 优化后

**1.7 不要迷信正则表达式，使用后要做测试。**

简单替换可以做到的效果尽量不用正则，例子



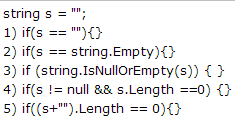
10万次的测试结果表明，1的效率高于2，且差距甚大。



仍然是1的效率高于2。

**1.8 字符串，非常常用**

**1.8.1 字符串判断**



1,2最慢 3较快 4,5最快

1,2几乎没区别 4,5几乎没区别

不过这个只适用于比较null和空字符串,如果是连续的空白就是string.IsNullOrWhiteSpace最快了,不过这个方法2.0里面没有，所以2.0可以这样 (s+””).trim() == 0。

这里的关键就是 s + “”  这个操作可以把null转换为””

注意第二个参数只能是””或string.Empty 这样的累加几乎是不消耗时间的,如果第二个参数是” “(一个空格)这个时间就远远不止了

**1.8.2字符串拼接**

大量字符串拼接，用Stringbuilder，少数几个直接使用string+string

1. **合理使用缓存**

**2.1访问量大的接口或功能，采用索引或缓存**

直接查询关系数据库，容易导致查询超时。解决办法：根据具体情况可以尝试把数据缓存到内存或分布式缓存，不能缓存的可以针对查询条件建合适的索引或迁移到mongodb。

**2.2****使用ICache接口（Foundation中有该接口，采用Memcache实现）**

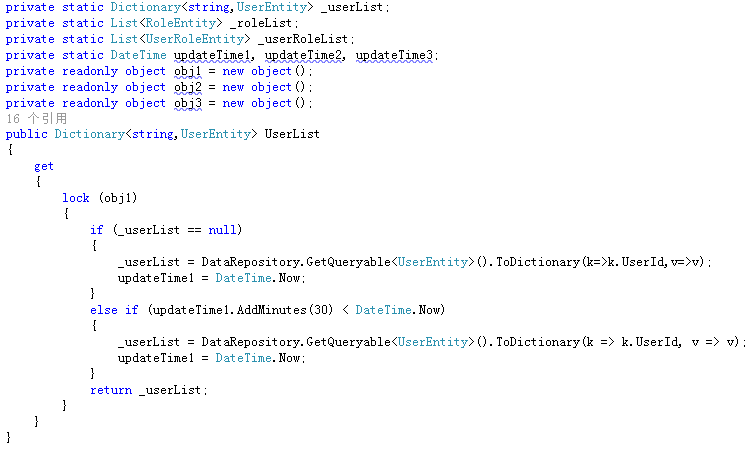
使用ICache接口，要设置过期时间，不设置默认是24分钟；缓存过期或丢失后要有替代方案获取数据。

**2.3内存缓存，需了解服务器内存使用情况**

当决定把数据缓存到内存时，要了解当前服务器内存使用情况，判断当前数据的大小和预估未来的增长情况，不能把服务器内存撑爆。

**2.4 常用表的数据，若对数据变化实时性要求不高，可简单缓存**

缓存在静态变量中或者.net Cache对象中。 如下：数据30分钟取一次，时间可调整。



1. **合理使用数据结构**

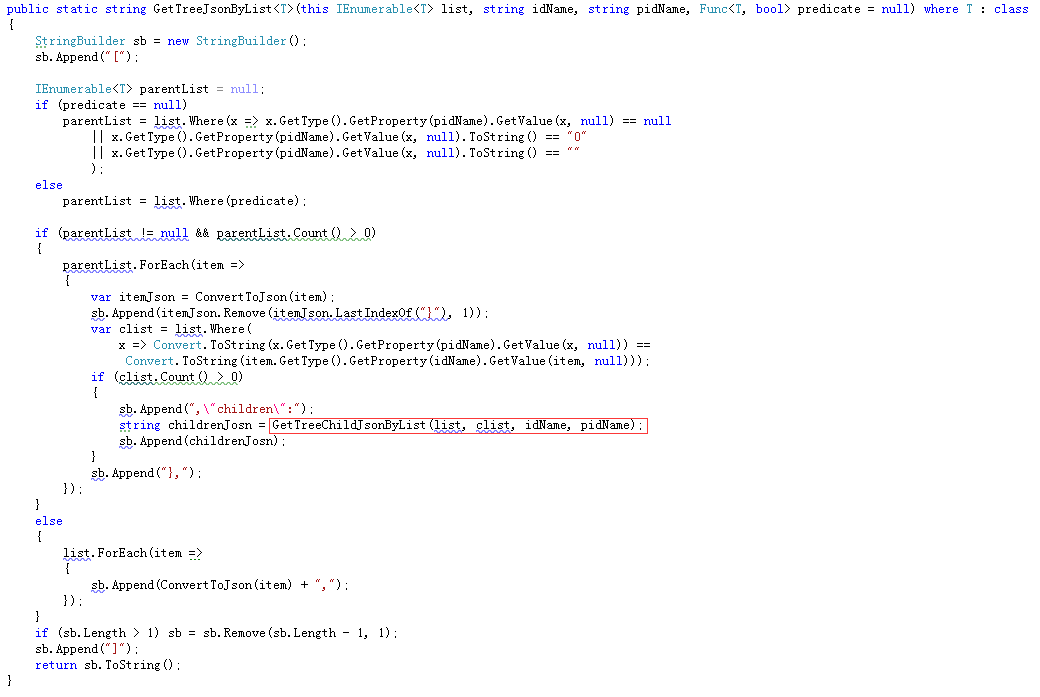
C#中有很多集合的数据结构，合理的使用这些数据结构，对我们的代码性能的提升呈几倍增长。

**3.1 合理使用Dictionary**

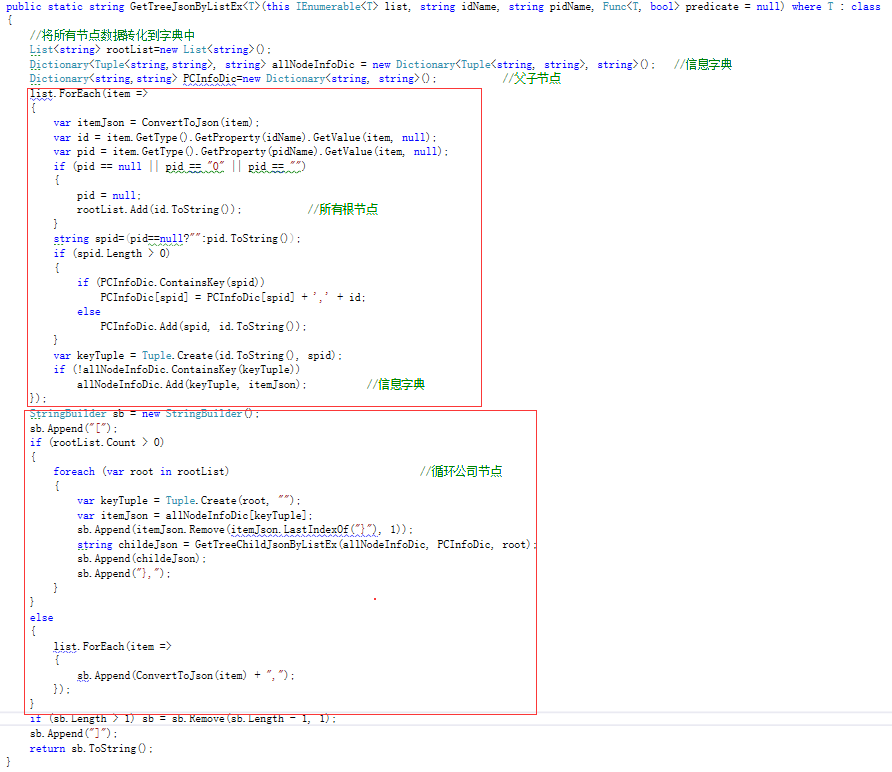
举个例子：曾经线上站点容器，要展现下面的权限树，在15分钟内还无法加载出来，而且此时功能还比较简单，只是展现树；后期经过优化后，不但增加了一些功能处理，数据量也更多，但是性能一直稳定在3s左右。



下面简单说下优化过程，证明合理采用数据结构很重要：

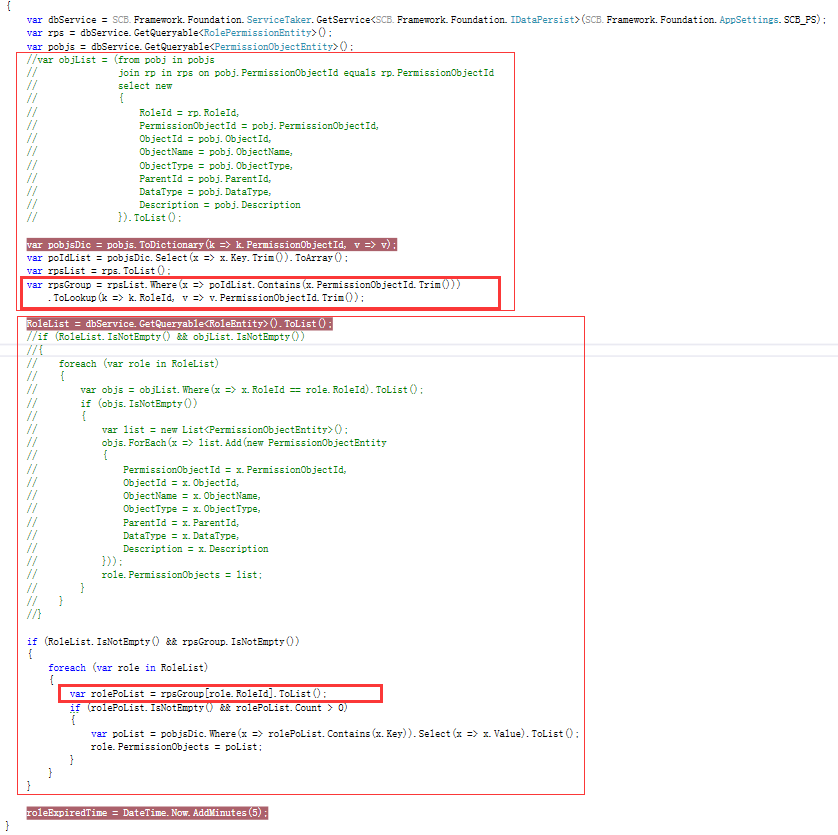


原函数，递归将List中存储的节点转化为Json字符串返回。属于Common中的一个公共函数，被很多地方使用。Vs2015测试到该处，发现爆慢。



这是优化后的代码，整体分为两个部分，先将List中的节点根据特征建立对应的Dic结构，后续所有的查找计算全部使用Dic信息。如此，效率稳定在1s左右，巨大的提升。

**3.2 合理使用Lookup**



注释的代码是原代码。优化分两部分，首先将原来Linq进行表联合的代码去掉，改成分开单表查询。测试证明此处分开，效率会稍好一些。但这条不是一条定律，需要根据实际测试情况。

同时，考虑到后需要使用角色分组数据，因此在内存加工中直接转成LookUp结构存储。这样，在后续的过程中可以减少一个内循环查找过程。

**3.3 使用其他，如HashSet等**

欢迎同事投稿。

**4.关于数据库方面**

**4.1数据持久化**

如果数据库使用SQL Server时，MapContex项目中映射TableName("表名").Schema("dbo")需使用“dbo”作为参数，不能使用“orm”；MySql使用 “orm”作为参数。

**4.2优化读取数据库的语句**

对于信息系统或者动态网站，很多时候性能瓶颈就在数据读取上。因此，代码中对数据的读取优化很重要。

**4.2.1去除重复读写同一数据的过程**

在过去很长一段时间内，我查阅很多小伙伴写的代码。发现一个问题，在处理同一个业务的同一个过程中，由于逻辑理的不够清晰，会出现重复读取同一张表的数据，也会分多次给某一张写入数据。

出现这种情况，需要厘清逻辑，整个过程一共有几次取数据，涉及哪些表，一共又有几次写数据，涉及哪些表。然后合并这些过程，达到一次读写的目的。

**4.2.2 Linq下，多表联合查询与分开单表查询性能**

在关系数据库要联表查询时，有一个或多个表数据量比较大时，联表查的性能可能不如根据条件分开单表查询再在内存中对数据进行加工的性能。这个过程需要进行测试比较。

个人觉得Linq通过Orm存取数据的效率，不如代码调用Sql语句或者存储过程。

**4.2.3 mongodb客户端并不支持所有linq写法**

mongodb客户端并不支持所有linq写法，如下面情况不支持：

var alist = mongo.GetQueryable<SellerCube.FMS.Report.SubjectPeriodSumReportEntity>()

.Where(x => plist.Contains(x.AccountantPeriodNum))

.GroupBy(g => new { g.AccountantPeriodNum, g.CompanyID, g.SubjectId })

.Select(x=>new {period=x.Key.AccountantPeriodNum,company=x.Key.CompanyID,total=x.Sum(g=>g.CreditAmount) }

).ToList();

用GroupBy时，Where不能写list.Contains或array.Contains

**4.2.4给表建立合适的索引**

索引分为聚集索引和非聚集索引，索引对查询效率的优化是明显的。因此，合适的索引很有必要。

设计一张表的时候，我们很多时候只考虑主键的设定即聚集索引。除此之外，我们也应该考虑，在哪些键上加上非聚集索引。一般来说，索引不超过3个字段，建立索引的字段也不宜过长，一张表的索引也不宜过多，2-3个即可。太多索引会影响表的插入效率。

多个字段组成的索引，称为复合索引，假如A,B,C三个字段组成复合索引IndexDemo，那么A出现在查询条件中才是有效的。单独在查询条件中使用B或者C字段，没有效果。

**5.避免情况**

**5.1程序搜索大文件夹下的某文件**

在文件很多的文件夹里用程序去搜索文件，容易撑爆CPU，要避免这种情况。