

从语法、程序架构和设计、编码习惯和编程规范3个层面深入探讨
编写高质量C#代码的技巧、禁忌和最佳实践



陆敏技 著

Writing Solid C# Code: 157 Suggestions to Improve Your C# Program

编写高质量代码 改善C#程序的157个建议



机械工业出版社
China Machine Press



Writing Solid C# Code: 157 Suggestions to Improve Your C# Program

编写高质量代码 改善C#程序的157个建议

陆敏技 著



机械工业出版社
China Machine Press

本书是 C# 程序员进阶修炼的必读之作，所有建议都是 C# 编码的最佳实践，从语言本身、程序的设计和架构、编码规范和编程习惯三大方面对 C# 程序员遇到的经典问题给出了经验性的解决方案，为 C# 程序员提供了 157 条极为宝贵的建议。对于每一个问题，不仅以建议的方式给出了被实践证明为十分优秀的解决方案，而且还给出了经常被误用或被错误理解的反例，从正反两个方面进行分析和对比。

全书一共三个部分，第一部分专注于 C# 语言本身，一共 89 条建议，涵盖了 C# 语言基本要素、集合、LINQ、泛型、委托、事件、资源管理、序列化、异常处理、异步、多线程、任务和并行编程等与 C# 语法相关的核心内容；第二部分重点讲解了 C# 程序的设计和架构，一共 32 条建议，涉及成员设计、面向对象的类型设计、安全性设计等重要方面的内容；第三部分探讨了 C# 的编码规范及编程习惯，一共 36 条建议，包含 C# 命名规范、如何使代码更整洁以及如何规范开发行为等方面的内容。

本书是一本关于如何编写高质量 C# 代码的工具书，列举的问题非常典型，给出的建议也非常实用，其中的每一条建议都有可能在我们编写下一行代码的时候用到。你可以将此书搁置在案头，以便有需要的时候随时查阅。

封底无防伪标均为盗版

版权所有，侵权必究

本书法律顾问 北京市展达律师事务所

图书在版编目 (CIP) 数据

编写高质量代码：改善 C# 程序的 157 个建议 / 陆敏技著. —北京：机械工业出版社，2011.9

ISBN 978-7-111-35649-3

I. 编… II. 陆… III. C 语言—程序设计 IV. TP312

中国版本图书馆 CIP 数据核字 (2011) 第 166703 号

机械工业出版社 (北京市西城区百万庄大街 22 号 邮政编码 100037)

责任编辑：白 宇

北京京师印务有限公司印刷

2011 年 10 月第 1 版第 1 次印刷

186mm×240mm • 22.5 印张

标准书号：ISBN 978-7-111-35649-3

定价：59.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换

客服热线：(010) 88378991；88361066

购书热线：(010) 68326294；88379649；68995259

投稿热线：(010) 88379604

读者信箱：hzjsj@hzbook.com



前言

为什么写这本书

事实上，我在写本书之前就一直在思考一个问题：到底什么样的编程书籍能够帮助入门者快速进阶？所谓“入门者”指的是已经可以使用一门语言编写程序，但是不明白如何编写高质量代码的人。于是我开始回忆自己在开发生涯中的入门阶段，那时候，我常常被以下三类问题所困扰。第一类来自于语言本身，如：

- ❑ 如何格式化字符串才是最高效的？
- ❑ 基本类型间或其他 CLR 类型间的转换怎样才算是最高效的？
- ❑ for 和 foreach 有什么区别，何时该用 for，何时该用 foreach？
- ❑ 什么是 Dispose 模式，为什么要释放资源，如何释放资源？
- ❑ 多线程应选择何种方式来开启和结束，各线程之间为什么要同步，如何同步，如何锁定资源？

第二类来自于设计架构，如果你对编码充满热情，相信你的大脑里很快就会充满以下这些问题：

- ❑ 使用单例模式还是静态类，为什么有了静态类还需要单例模式？
- ❑ 该使用静态方法还是实例方法，它们的本质区别是什么？
- ❑ 如何使用异常才是最正确的，什么时候抛出异常，什么时候“吃掉”异常，为什么需要自定义异常？
- ❑ 如何避免过多的条件判断分支？
- ❑ 如何保证程序的数据安全和通信安全？

第三类问题最常见，可能来自于编码习惯和编程思想，我在入门阶段经常会问自己下面这些问题：

- ☐ 一个文件只包含一个类比较好，还是一个文件可以包含多个类？
- ☐ 如何命名才是专业级别的？
- ☐ 应该使用抽象类还是接口？
- ☐ 到底什么才是真正的面向对象编码，我这样编程够面向对象吗？
- ☐ 什么是单元测试，如何编写单元测试？

如果你也曾经问过自己类似的问题，说明你已经为成为专业程序员做好了准备；如果你还苦于找不到问题的答案，那么本书正是为你准备的。本书为那些普遍存在于初级开发者脑海中的问题给出了经验性的解决方案。我尽可能多地罗列问题并给出解决方法（同时给出错误的方法和正确的方法，以及好的方法和更好的方法，并进行分析和比较）和建议，并且尽量覆盖 .NET 4.0 中 C# 的新特性。我已经用 Visual Studio 2010 编写并调试和运行（在 Release 状态下）了书中所有的代码，我期望你也这样做。相信当你调试完书中所有的示例时，你会对 .NET 和 C# 有更深入的认识，同时你也会对自己的代码更加有自信：没错，这就是我想要的代码，它规范，并且优雅而稳定。

如何阅读本书

本书适合那些有一定 C# 基础，并希望在技术上得到大幅提升的程序员。

本书并没有讲述 C# 中的基础概念，而是将使用 C# 过程中可能遇到的疑问或障碍一一罗列，并给出了建议。书中的大多数建议实战性很强，要完全理解其中的奥妙，首先应该动手写一写示例程序，或许在调试程序的过程中就会得到启发。

本书的每一个章节都比较独立，下一章的阅读并不需要前面章节作为铺垫。本书更像一本工具书，根据知识点各个建议进行了分类，方便随时查阅。

资源及勘误

通常情况下，一个问题的解决方案往往不止一种，你可能会不同意本书中的一些观点，甚至会强烈反对。没有关系，你可以通过 luminji@hotmail.com (E-mail) 与我分享你的宝贵意见，同时也可以到 <http://www.cnblogs.com/luminji> 下载书中的源码。我也经常在那里发表博客。当然，你一定也会在书中找到一些错误，我已经在博客上放置了一篇勘误表，我会在第一时间公布这些勘误。

致谢

最后，我想说的是，写作是一项耗时的工程，它不但压缩了我的工作时间，也完全耗掉了我的午休时间。首先要感谢我的同事，因为我时常因为思考本书的内容而工作不在状态，或者总在饭桌上讨论书中的一些技术细节。他们不但没有抱怨，反而提出了很多好的意见，他们是：胡昌俊、于文广、肖昌、樊鑫、王文壮、来伟、栗志辉、赵中海、王领军，还有其他很多朋友不再一一列出。他们是如此优秀和宽容，能和他们一起工作是我莫大的荣幸。

在漫长的写作过程中，难免让我情绪波动，是机械工业出版社华章公司的杨福川先生和杨绣国女士给我一如既往的支持与鼓励。当我想要偷懒的时候，是你们的敦促让我对写作时刻保持着热情。同时还要强调一下，没有福川，本书不会出版。另外，没有白宇女士的编辑工作，本书的文字看上去不会像现在这样流畅。

还要感谢我的家人，尤其是我的父母和妻子胡忠华。在过去这段时间里，我总是一回到家就打开电脑。陪妻子散步的时间没有了，聊天少了，家务活儿也很少做了。妻子的宽容让我有了更多的时间去写作。难得的是，中文系毕业的你在写作方面给我提出了很多好的建议。你的第一标准是：“如果我都看不懂，你还写什么书。”没有你的支持，本书不会诞生。

陆敏技

2011年7月



目 录

前 言

第一部分 语言篇

第 1 章 基本语言要素 / 2

- 建议 1: 正确操作字符串 / 2
- 建议 2: 使用默认转型方法 / 6
- 建议 3: 区别对待强制转型与 as 和 is / 9
- 建议 4: TryParse 比 Parse 好 / 12
- 建议 5: 使用 int? 来确保值类型也可以为 null / 15
- 建议 6: 区别 readonly 和 const 的使用方法 / 16
- 建议 7: 将 0 值作为枚举的默认值 / 19
- 建议 8: 避免给枚举类型的元素提供显式的值 / 20
- 建议 9: 习惯重载运算符 / 22
- 建议 10: 创建对象时需要考虑是否实现比较器 / 23
- 建议 11: 区别对待 == 和 Equals / 27
- 建议 12: 重写 Equals 时也要重写 GetHashCode / 29
- 建议 13: 为类型输出格式化字符串 / 32
- 建议 14: 正确实现浅拷贝和深拷贝 / 36
- 建议 15: 使用 dynamic 来简化反射实现 / 40

第2章 集合和 LINQ / 43

- 建议 16: 元素数量可变的情况下不应使用数组 / 43
- 建议 17: 多数情况下使用 foreach 进行循环遍历 / 45
- 建议 18: foreach 不能代替 for / 51
- 建议 19: 使用更有效的对象和集合初始化 / 53
- 建议 20: 使用泛型集合代替非泛型集合 / 54
- 建议 21: 选择正确的集合 / 57
- 建议 22: 确保集合的线程安全 / 61
- 建议 23: 避免将 List<T> 作为自定义集合类的基类 / 64
- 建议 24: 迭代器应该是只读的 / 67
- 建议 25: 谨慎集合属性的可写操作 / 68
- 建议 26: 使用匿名类型存储 LINQ 查询结果 / 70
- 建议 27: 在查询中使用 Lambda 表达式 / 73
- 建议 28: 理解延迟求值和主动求值之间的区别 / 75
- 建议 29: 区别 LINQ 查询中的 IEnumerable<T> 和 IQueryable<T> / 78
- 建议 30: 使用 LINQ 取代集合中的比较器和迭代器 / 80
- 建议 31: 在 LINQ 查询中避免不必要的迭代 / 83

第3章 泛型、委托和事件 / 86

- 建议 32: 总是优先考虑泛型 / 86
- 建议 33: 避免在泛型类型中声明静态成员 / 88
- 建议 34: 为泛型参数设定约束 / 90
- 建议 35: 使用 default 为泛型类型变量指定初始值 / 92
- 建议 36: 使用 FCL 中的委托声明 / 94
- 建议 37: 使用 Lambda 表达式代替方法和匿名方法 / 96
- 建议 38: 小心闭包中的陷阱 / 99
- 建议 39: 了解委托的实质 / 103
- 建议 40: 使用 event 关键字为委托施加保护 / 106
- 建议 41: 实现标准的事件模型 / 108
- 建议 42: 使用泛型参数兼容泛型接口的不可变性 / 109
- 建议 43: 让接口中的泛型参数支持协变 / 111

建议 44: 理解委托中的协变 / 112

建议 45: 为泛型类型参数指定逆变 / 114

第 4 章 资源管理和序列化 / 116

建议 46: 显式释放资源需继承接口 IDisposable / 116

建议 47: 即使提供了显式释放方法, 也应该在终结器中提供隐式清理 / 119

建议 48: Dispose 方法应允许被多次调用 / 120

建议 49: 在 Dispose 模式中应提取一个受保护的虚方法 / 121

建议 50: 在 Dispose 模式中应区别对待托管资源和非托管资源 / 123

建议 51: 具有可释放字段的类型或拥有本机资源的类型应该是可释放的 / 124

建议 52: 及时释放资源 / 125

建议 53: 必要时应将不再使用的对象引用赋值为 null / 127

建议 54: 为无用字段标注不可序列化 / 131

建议 55: 利用定制特性减少可序列化的字段 / 136

建议 56: 使用继承 ISerializable 接口更灵活地控制序列化过程 / 137

建议 57: 实现 ISerializable 的子类型应负责父类的序列化 / 140

第 5 章 异常与自定义异常 / 144

建议 58: 用抛出异常代替返回错误代码 / 144

建议 59: 不要在不恰当的场所下引发异常 / 147

建议 60: 重新引发异常时使用 Inner Exception / 150

建议 61: 避免在 finally 内撰写无效代码 / 151

建议 62: 避免嵌套异常 / 157

建议 63: 避免“吃掉”异常 / 160

建议 64: 为循环增加 Tester-Doer 模式而不是将 try-catch 置于循环内 / 161

建议 65: 总是处理未捕获的异常 / 162

建议 66: 正确捕获多线程中的异常 / 166

建议 67: 慎用自定义异常 / 168

建议 68: 从 System.Exception 或其他常见的基本异常中派生异常 / 170

建议 69: 应使用 finally 避免资源泄漏 / 172

建议 70: 避免在调用栈较低的位置记录异常 / 175

第6章 异步、多线程、任务和并行 / 177

- 建议 71: 区分异步和多线程应用场景 / 177
- 建议 72: 在线程同步中使用信号量 / 180
- 建议 73: 避免锁定不恰当的同步对象 / 184
- 建议 74: 警惕线程的 `IsBackground` / 188
- 建议 75: 警惕线程不会立即启动 / 189
- 建议 76: 警惕线程的优先级 / 191
- 建议 77: 正确停止线程 / 193
- 建议 78: 应避免线程数量过多 / 194
- 建议 79: 使用 `ThreadPool` 或 `BackgroundWorker` 代替 `Thread` / 196
- 建议 80: 用 `Task` 代替 `ThreadPool` / 198
- 建议 81: 使用 `Parallel` 简化同步状态下 `Task` 的使用 / 202
- 建议 82: `Parallel` 简化但不等同于 `Task` 默认行为 / 204
- 建议 83: 小心 `Parallel` 中的陷阱 / 205
- 建议 84: 使用 `PLINQ` / 208
- 建议 85: `Task` 中的异常处理 / 209
- 建议 86: `Parallel` 中的异常处理 / 214
- 建议 87: 区分 WPF 和 WinForm 的线程模型 / 216
- 建议 88: 并行并不总是速度更快 / 220
- 建议 89: 在并行方法体中谨慎使用锁 / 222

第二部分 架构篇

第7章 成员设计 / 226

- 建议 90: 不要为抽象类提供公开的构造方法 / 226
- 建议 91: 可见字段应该重构为属性 / 226
- 建议 92: 谨慎将数组或集合作为属性 / 227
- 建议 93: 构造方法应初始化主要属性和字段 / 228
- 建议 94: 区别对待 `override` 和 `new` / 229
- 建议 95: 避免在构造方法中调用虚成员 / 235

- 建议 96: 成员应优先考虑公开基类型或接口 / 236
- 建议 97: 优先考虑将基类型或接口作为参数传递 / 237
- 建议 98: 用 params 减少重复参数 / 237
- 建议 99: 重写时不应使用子类参数 / 238
- 建议 100: 静态方法和实例方法没有区别 / 239
- 建议 101: 使用扩展方法, 向现有类型“添加”方法 / 240

第 8 章 类型设计 / 243

- 建议 102: 区分接口和抽象类的应用场合 / 243
- 建议 103: 区分组合和继承的应用场合 / 245
- 建议 104: 用多态代替条件语句 / 248
- 建议 105: 使用私有构造函数强化单例 / 251
- 建议 106: 为静态类添加静态构造函数 / 253
- 建议 107: 区分静态类和单例 / 255
- 建议 108: 将类型标识为 sealed / 255
- 建议 109: 谨慎使用嵌套类 / 256
- 建议 110: 用类来代替 enum / 257
- 建议 111: 避免双向耦合 / 260
- 建议 112: 将现实世界中的对象抽象为类, 将可复用对象圈起来就是命名空间 / 262

第 9 章 安全性设计 / 264

- 建议 113: 声明变量前考虑最大值 / 264
- 建议 114: MD5 不再安全 / 265
- 建议 115: 通过 HASH 来验证文件是否被篡改 / 268
- 建议 116: 避免用非对称算法加密文件 / 269
- 建议 117: 使用 SSL 确保通信中的数据安全 / 273
- 建议 118: 使用 SecureString 保存密钥等机密字符串 / 284
- 建议 119: 不要使用自己的加密算法 / 289
- 建议 120: 为程序集指定强名称 / 289
- 建议 121: 为应用程序设定运行权限 / 291

第三部分 编码规范及习惯

第 10 章 命名规范 / 296

- 建议 122: 以 <Company>.<Component> 为命名空间命名 / 296
- 建议 123: 程序集不必与命名空间同名 / 296
- 建议 124: 考虑在命名空间中使用复数 / 297
- 建议 125: 避免用 FCL 的类型名称命名自己的类型 / / 297
- 建议 126: 用名词和名词组给类型命名 / 298
- 建议 127: 用形容词组给接口命名 / 299
- 建议 128: 考虑让派生类的名字以基类名字作为后缀 / 300
- 建议 129: 泛型类型参数要以 T 作为前缀 / 300
- 建议 130: 以复数命名枚举类型, 以单数命名枚举元素 / 301
- 建议 131: 用 PascalCasing 命名公开元素 / 302
- 建议 132: 考虑用类名作为属性名 / 302
- 建议 133: 用 camelCasing 命名私有字段和局部变量 / 303
- 建议 134: 有条件地使用前缀 / 304
- 建议 135: 考虑使用肯定性的短语命名布尔属性 / 305
- 建议 136: 优先使用后缀表示已有类型的新版本 / 306
- 建议 137: 委托和事件类型应添加上级后缀 / 307
- 建议 138: 事件和委托变量使用动词或形容词短语命名 / 308
- 建议 139: 事件处理器命名采用组合方式 / 309

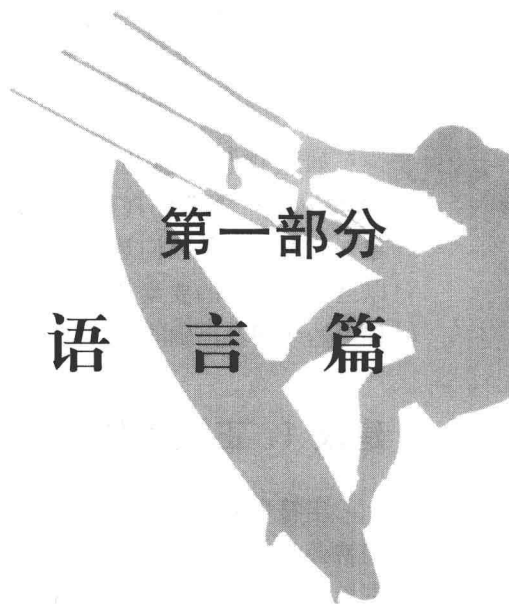
第 11 章 代码整洁 / 311

- 建议 140: 使用默认的访问修饰符 / 311
- 建议 141: 不知道该不该用大括号时, 就用 / 312
- 建议 142: 总是提供有意义的命名 / 314
- 建议 143: 方法抽象级别应在同一层次 / 315
- 建议 144: 一个方法只做一件事 / 316
- 建议 145: 避免过长的方法和过长的类 / 317
- 建议 146: 只对外公布必要的操作 / 318

- 建议 147: 重构多个相关属性为一个类 / 319
- 建议 148: 不重复代码 / 320
- 建议 149: 使用表驱动法避免过长的 if 和 switch 分支 / 321
- 建议 150: 使用匿名方法、Lambda 表达式代替方法 / 324
- 建议 151: 使用事件访问器替换公开的事件成员变量 / 325
- 建议 152: 最少, 甚至是不要注释 / 326
- 建议 153: 若抛出异常, 则必须要注释 / 326

第 12 章 规范开发行为 / 327

- 建议 154: 不要过度设计, 在敏捷中体会重构的乐趣 / 327
- 建议 155: 随生产代码一起提交单元测试代码 / 336
- 建议 156: 利用特性为应用程序提供多个版本 / 342
- 建议 157: 从写第一个界面开始, 就进行自动化测试 / 344



第一部分 语言篇

本部分内容

- 第 1 章 基本语言要素
- 第 2 章 集合和 LINQ
- 第 3 章 泛型、委托和事件
- 第 4 章 资源管理和序列化
- 第 5 章 异常与自定义异常
- 第 6 章 异步、多线程、任务和并行

第 1 章 基本语言要素

如何操作字符串？如何进行转型？什么是克隆？什么是相等型？为什么需要 HashCode？当我们真正开始使用一门语言进行编程时，就会遇到这些问题。这些问题看起来简单，可是我们是否想过：为什么要这样处理，这样做是最好的吗？本章的内容将以这些最基础的要素开篇，给出 C# 编码中的一些最佳实践，让我们在 C# 进阶学习的过程中始终朝着正确的方向前进。

建议 1：正确操作字符串

字符串应该是所有编程语言中使用最频繁的一种基础数据类型。如果使用不慎，我们就会为一次字符串的操作所带来的额外性能开销而付出代价。本条建议将从两个方面来探讨如何规避这类性能开销：

- ❑ 确保尽量少的装箱

- ❑ 避免分配额外的内存空间

先来介绍第一个方面，请看下面的两行代码：

```
1.      String str1 = "str1"+ 9;
2.      String str2 = "str2"+ 9.ToString();
```

为了清楚这两行代码的执行情况，我们来比较两者生成的 IL 代码。

第一行代码对应的 IL 代码如下：

```
.maxstack 8
IL_0000: ldstr      "str1"
IL_0005: ldc.i4.s    9
IL_0007: box       [mscorlib]System.Int32
IL_000c: call       string [mscorlib]System.String::Concat(object, object)
IL_0011: pop
IL_0012: ret
```

第二行代码对应的 IL 代码如下：

```
.maxstack 2
.locals init ([0] int32 CS$0$0000)
IL_0000: ldstr      "str2"
```

```

IL_0005: ldc.i4.s    9
IL_0007: stloc.0
IL_0008: ldloca.s    CS$0$0000
IL_000a: call        instance string [mscorlib]System.Int32::ToString()
IL_000f: call        string [mscorlib]System.String::Concat(string, string)
IL_0014: pop
IL_0015: ret

```

可以看出，第一行代码“str1”+9在运行时会完成一次装箱行为（IL 代码中的 box）；而第二行代码中的 9.ToString() 并没有发生装箱行为，它实际调用的是整型的 ToString 方法。ToString 方法的原型为：

```

public override String ToString()
{
    return Number.FormatInt32(m_value, null, NumberFormatInfo.CurrentInfo);
}

```

可能有人会问，是不是原型中的 Number.FormatInt32 方法会发生装箱行为呢？实际上，Number.FormatInt32 方法是一个非托管的方法，其原型如下：

```

[MethodImpl(MethodImplOptions.InternalCall), SecurityCritical]
public static extern string FormatInt32(int value, string format,
    NumberFormatInfo info);

```

它是通过直接操作内存来完成从 int 到 string 的转换，效率要比装箱高很多。所以，在使用其他值引用类型到字符串的转换并完成拼接时，应当避免使用操作符“+”来完成，而应该使用值引用类型提供的 ToString 方法。

也许有人还会提出疑问：上文所举的示例中，即使 FCL 提供的方法没有发生装箱行为，但在其他情况下，FCL 方法内部会不会含有装箱的行为呢？答案是：也许会有。不过，我们这里有一个指导原则：

在自己编写的代码中，应当尽可能地避免编写不必要的装箱代码。

注意 装箱之所以会带来性能损耗，因为它需要完成下面三个步骤：

- 1) 首先，会为值类型在托管堆中分配内存。除了值类型本身所分配的内存外，内存总量还要加上类型对象指针和同步块索引所占用的内存。
 - 2) 将值类型的值复制到新分配的堆内存中。
 - 3) 返回已经成为引用类型的对象的地址。
-

第二个方面：避免分配额外的内存空间。对 CLR 来说，string 对象（字符串对象）是个很特殊的对象，它一旦被赋值就不可改变。在运行时调用 System.String 类中的任何

4 ❖ 第一部分 语 言 篇

方法或进行任何运算（如“=”赋值、“+”拼接等），都会在内存中创建一个新的字符串对象，这也意味着要为该新对象分配新的内存空间。像下面的代码就会带来运行时的额外开销。

```
private static void NewMethod1()
{
    string s1 = "abc";
    s1 = "123" + s1 + "456";    // 以上两行代码创建了3个
    // 字符串对象，并执行了一次 string.Concat 方法
}

private static void NewMethod6()
{
    string re6 = 9 + "456";    // 该代码发生一次装箱，并调
    // 用一次 string.Concat 方法
}
```

而在以下代码中，字符串不会在运行时拼接字符串，而是会在编译时直接生成一个字符串。

```
private static void NewMethod2()
{
    string re2 = "123" + "abc" + "456"; // 该代码等效于
    // string re2 = "123abc456";
}

private static void NewMethod9()
{
    const string a = "t";
    string re1 = "abc" + a;    // 因为 a 是一个常量，所以
    // 该行代码等效于 string re1 = "abc" + "t";
    // 最终等效于 string re1 = "abct";
}
```

由于使用 System.String 类会在某些场合带来明显的性能损耗，所以微软另外提供了一个类型 StringBuilder 来弥补 String 的不足。

StringBuilder 并不会重新创建一个 string 对象，它的效率源于预先以非托管的方式分配内存。如果 StringBuilder 没有先定义长度，则默认分配的长度为 16。当 StringBuilder 字符长度小于等于 16 时，StringBuilder 不会重新分配内存；当 StringBuilder 字符长度大于 16 小于 32 时，StringBuilder 又会重新分配内存，使之成为 16 的倍数。在上面的代码中，如果预先判断字符串的长度将大于 16，则可以为其设定一个更加合适的长度（如 32）。StringBuilder 重新分配内存时是按照上次的容量加倍进行分