HZ BOOKS
华章科技

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

實戰

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

實戰

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。

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

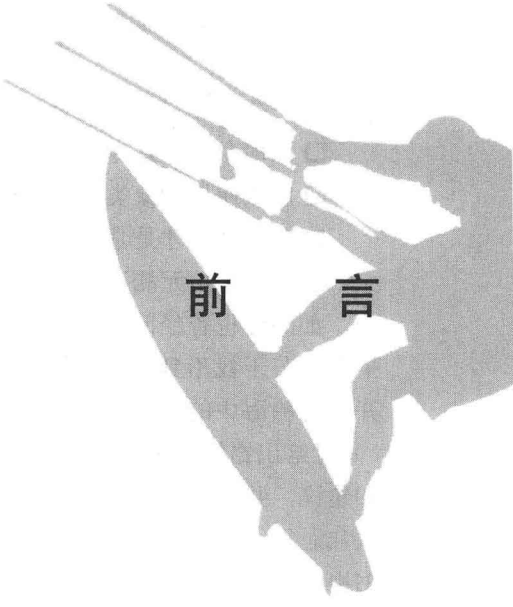
ISBN 978-7-111-35649-3

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，根据知识点对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，同时也可以在 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

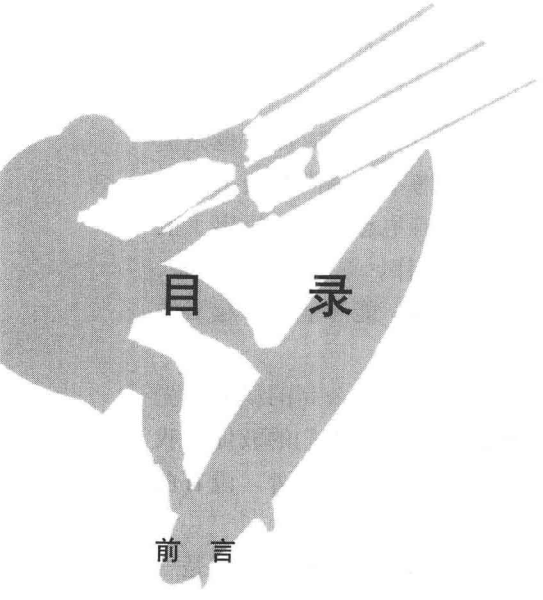
最后，我想说的是，写作是一项耗时的工程，它不但压缩了我的工作时间，也完全耗掉了我的午休时间。首先要感谢我的同事，因为我时常因为思考本书的内容而工作不在状态，或者总在饭桌上讨论书中的一些技术细节。他们不但没有抱怨，反而提出了很多好的意见，他们是：胡昌俊、于文广、肖昌、樊鑫、王文壮、来伟、粟志辉、赵中海、王领军，还有其他很多朋友不再一一列出。他们是如此优秀和宽容，能和他们一起工作是我莫大的荣幸。

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

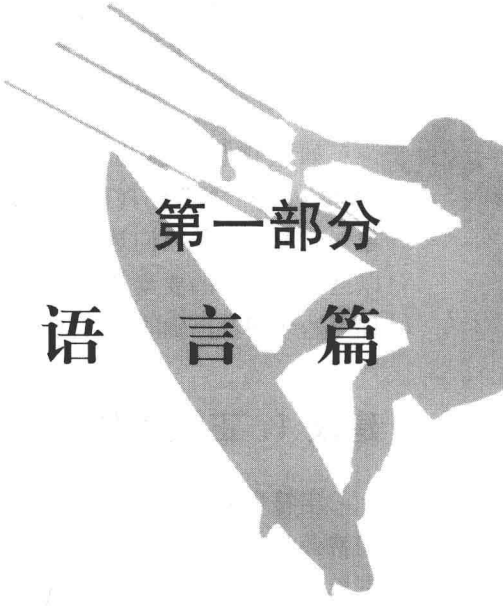
第 2 章　集合和 LINQ / 43

第 3 章　泛型、委托和事件 / 86

第三部分　编码规范及习惯

第 10 章　命名规范 / 296

第 11 章　代码整洁 / 311



第一部分

语　言　篇

本部分内容

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：也许会存在。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

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.Contact 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.Contact 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