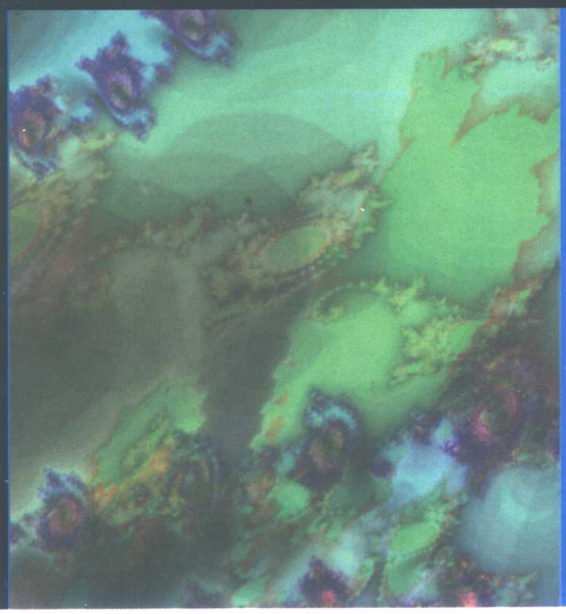
BOOKS FOR PROFESSIONALS BY PROFESSIONALS™



Distributed .NET Programming in C#

.NET 分布式编程

——C# 篇

- 了解分布式编程和.NET 的发展史
- 用.NET Remoting 和 Web 服务构建快速、可伸缩的以及健壮的分布式应用程序
- 在.NET 中开发服务组件和使用 MSMQ

(美) TOM BARNABY 著
黎 媛 王小锋 等译

清华大学出版社

.NET 分布式编程

——C#篇

(美) TOM BARNABY 著

黎媛 王小锋 等译

清华大学出版社

北 京

内 容 简 介

分布式编程和.NET 平台这两个主题都需要花费大量笔墨才能描述清楚，但在本书中，作者择其精要，深入浅出地介绍了在构建分布式应用时需用到的一些主要的.NET 技术，如.NET Remoting、Web 服务、串行化、COM+和 MSMQ 等。对于每一项技术，都首先进行详细的分析，然后再在实际应用中体会该技术解决问题的能力。

本书适用的对象为希望利用.NET 技术来构建分布式应用程序的并且具有 C#和面向对象编程经验的程序员。

EISBN：1-59059-039-2
Distributed .NET Programming in C#
Tom Barnaby

北京市版权局著作权合同登记号　图字：01-2003-7803

图书在版编目(CIP)数据

.NET 分布式编程——C#篇/(美)巴纳比(Barnaby,T.)著；黎媛等译.—北京：清华大学出版社，2004
书名原文：Distributed .NET Programming in C#
ISBN 7-302-08443-2

Ⅰ.N… Ⅱ.①巴…②黎… Ⅲ.C 语言—程序设计 Ⅳ.TP312

中国版本图书馆 CIP 数据核字(2004)第 029618 号

出 版 者：清华大学出版社　　**地　　址**：北京清华大学学研大厦
http://www.tup.com.cn　　**邮　　编**：100084
社 总 机：010-62770175　　**客户服务**：010-62776969
组稿编辑：曹康
文稿编辑：王军
封面设计：康博
版式设计：康博
印 装 者：北京鑫海金澳胶印有限公司
发 行 者：新华书店总店北京发行所
开　　本：185×260　**印张**：22.25　**字数**：569 千字
版　　次：2004 年 4 月第 1 版　2004 年 4 月第 1 次印刷
书　　号：ISBN 7-302-08443-2/TP·6068
印　　数：1～4000
定　　价：43.00 元

前　言

对于进行分布式编程的广大程序员来说，将 COM 技术扩展到网络上的 DCOM 无疑是一种福祉。在 DCOM 模型的帮助下，客户端可以与位于任何地方的 COM 对象进行精确的互操作，而无需改变 CodeBase。在 AppID、存根、代理以及通道的间接帮助下可以实现分布式的目的，而且其最终结果与使用像 dcomcnfg.exe 和 Component Services snap-in 这样的工具并无什么区别。不过，在 DCOM 中并不是都能事事如意，就这点而言，在 COM 中同样如此。虽然从表面上看，只需单击一些复选框就可以轻松地实现基于 COM 的远程调用，但实际上，还需要面对注册冲突、通过引用传递接口指针时所花费的时间、以及跨防火墙时的艰难等诸多问题。

正如 ADO.NET 与典型 ADO 并无多大联系一样，.NET Remoting 与典型 DCOM 也没有什么联系。这一点表现得最明显的是.NET 程序集并不进行系统注册，因此，我们也没有 AppID。没有了 AppID，我们也就没有 RemoteServerName 的值，这就进一步意味着没有对 oleaut32.dll 的引用，也就没有了基于 COM 的存根和代理。简而言之，过去我们所了解的跨网络进行的交互方式全都发生了翻天覆地的变化。

.NET 为我们提供了许多新的远程结构。除了要面对如 WKO、CAO 这样的众多 TLA(3 个字母的首字母缩写)外，还需要处理围绕旧概念所引发的新问题(例如实际代理和透明代理之间的区别)、以及 XML 配置文件的任务。

许多准备学习.NET 分布式编程的程序员一般都会求助于 MSDN。这时，他们就会学习许多代码示例、微软的一些白皮书、还需要一台 21 寸的监视器来查看各式图表。这样一来，必定会给程序员们造成不少麻烦，而且也会使知识结构之间产生脱节。现在最需要的是要有一种实际的方法，能够彻底地解决在企业级应用环境中如何合理运用这些新技术的问题。

Tom 最新出版的图书(也就是您现在手上拿的这本书)就提供了这种解决方式。在本书中，对于.NET Remoting 层的许多具体内容，您都可以看到逻辑分明清晰易懂的说明和评述。在 Tom 的这本书中，不但经常性地将这一系列新的 TLA 反复提出来，还提出许多以企业为中心的技术，例如构建配置组件(亦称 COM+)、.NET 通信、Web 服务和典型的 COM 类型间的交互，从而可以加深您对相关技术的理解。

多年来，Tom 和我都一直在 Intertech 公司(http://intertech-inc.com)共事，亲眼目睹他进行了许多关于典型 COM 和.NET 方面的教学。我很荣幸能够与他在许多开发研究工作中进行合作。对于想进行分布式编程的程序员们来说，以我的经验来看，本书确实不错。

祝愉快！

Andrew Troelsen

简　介

分布式编程这个话题很大。为了能恰当地实现分布式应用程序，您必须理解从低级的连网细节到高级的体系结构问题这所有环节中的方方面面。而且.NET 是一个新平台，需要花费几千页的文档才能讲述清楚。所以在编写这本书的时候我所面临的挑战是：如何才能将这两个都非常大的话题在一本并不算厚的书中讲得通俗易懂呢？

我的回答是：真的很难！换而言之，我必须假想读者的具体经验水平如何，但想要了解这一点很难，毕竟.NET 也是一个全新的平台。不过，即使再困难，我也必须对书的内容进行艰难的取舍。对于有些问题我还是能够肯定下来的，例如本书并不是技术文档的翻版，也不是洋洋洒洒谈几十个无关紧要话题的大全。

还是谈谈本书的大致情况吧。我把它看成是一次有导游进行讲解的旅游，利用.NET 中的基本技术，例如.NET Remoting、Web 服务、串行化、COM+和 MSMQ，来构建分布式的应用程序。而这些技术本身就足够复杂，值得深入研究。本书的着重点在于每一种技术的应用，以及该技术在分布式应用程序中所担当的角色。对于每项技术，首先对其进行分析，然后再从实际应用中来体会它能解决的问题。和其他工作一样，学习分布式编程最好的方法还是通过实践，在使用一种技术代替另一种技术时，我会尽可能地从正反两个方面来讨论利弊之处。

这些应用都比较复杂，可以保证能够进行深入的研究。

读者对象

如果您从书架上拿了这本书，我就假定您是位有兴趣使用.NET 技术来构建分布式应用程序的程序员。我还会假定您有 C#和面向对象编程方面的知识背景。如果具备.NET 的基本知识会有很大的帮助，不具备也没关系，本书的第 2 章谈到了这方面的一些内容。最重要的一点是：我假定您愿意花费一些时间来下载(或输入)并运行书中的示例；愿意从 MSDN 中找些资料来学习，愿意多次阅读某些章节以消化吸收所提出的一些概念。这些假设帮助我控制了本书的厚度。

运行示例所必备的条件

您可从 Apress Web 站点(http://www.apress.com)中下载本书所示的几乎所有代码。为了运行代码，至少需要安装.NET Framework 的最新发布版本，您可以从微软的 Web 站点(http://msdn.microsoft.com/netframework)进行免费下载。只要利用.NET Framework 所提供的编译器和工具，以及像 Notepad 这样的文本编辑器，都可以实现和测试书中的许多示例。不过，我假定 Visual Studio .NET 是首选的开发工具，联机代码包括了 Visual Studio .NET 解决方案文件。

在后面的几章中，您需要用其他的软件来运行示例，如 COM+、IIS 和 MSMQ。这些示例都是在专业版 Windows XP 下开发的，但我相信它们在 Windows 2000 下也能运行。

没有提供真实应用代码的原因

实际上，本书满篇都是可以进行真实应用的代码。也就是说，这些代码可以帮助您解决在利用.NET 构建分布式应用时遇到的常见问题。但我知道您的意思是我没有提供像比萨外卖服务、联系服务或可以工作的电子商务网站这种真实案例。我的观点是像这种示例的要求会比较高，并不适合目前学习这么多新的基本概念的阶段。如果在开发电子商务系统时陷入细节问题中时，需要花费很多时间精力和其他帮助手段才能脱离困境，而不仅仅是讨论或学习一些基本的概念就能解决的。因此，本书中的示例代码比较简短扼要。

书中为什么没有列出所有选项、方法、工具参数、类和方法的表

本书的定位首先是一本教学用书，其次的作用是教导具体如何操作，最后才是作为参考资料。关于.NET 最权威的参考用书已经面世了，这就是 MSDN。我觉得没有必要来重复微软已经做过的工作，没有必要再将每个工具的每个选项、每个类的每个方法、每个方法的每个参数等进行分类归档。不过，我确实发现有必要出这样一本书：能够通过一系列有逻辑关联的主题，来帮助读者澄清一些复杂的概念。还希望这本书不要太厚，能够放到您的公文包、背包、电脑包甚至手提袋中，我希望这本书能够满足这些目的。

为什么在运行示例代码时经常出现“找不到文件”异常

本书中的有些示例项目比较复杂，需要一些自定义的相关程序集。为使项目运行起来，这些程序集必须保存在特定的位置中。您必须阅读并熟悉一下第 2 章中所讲到的程序集绑定过程。特别要注意在“查看程序集绑定日志”一节中所谈到的程序集绑定日志查看器，以及在“绑定过程小结”中谈到的程序集绑定流程图。这两节内容中有您诊断问题时所需要的信息。

结束语

在我的感觉中，Spinal Tap 是以往最伟大的一支摇滚乐队。但由于经营不善、女友们的干扰以及许多鼓手离奇地去世，到了 80 年代的早期，这支乐队就逐渐被人淡忘了。电影 This Is Spinal Tap 讲的就是这支乐队的悲伤往事。在软件开发过程中，也极需要乐队成员的这种聪明才智，愿我能与我的读者们分享他们这种深厚的洞察力。

如果您不准备将这本书放回书架上，那就让我们开始这一段探索之旅吧！

目　　录

第1章　分布式编程的发展

"It's like,how much more black can this be?
and the answer is none. None more black."

——Nigel Tufnel(This Is Spinal Tap)
关于软件开发的演讲

今天，诸如企业编程、分布式编程、n 层和可扩展性等流行词汇出现在每一个产品的宣传中。所以，要抓住.NET 中分布式开发的细微区别，就不能从字面上考虑这些术语，而应该考虑这些特殊词汇的真实含义和上下文环境。而且，由于这本书主要是一本"操作指南"，所以，清楚地理解为什么要分布应用程序以及如何设计一个分布式应用是非常重要的。在本章结尾提出了五项原则，它们可以指导您在.NET 平台及其他平台上进行分布式开发。

最后，为了了解分布式编程的历史，本章回顾了原有的分布式开发模型，以及这些旧模型被新模型取代的原因。如同您将要看到的，要解释清楚为什么微软创造出新的开发平台.NET 来取代 COM 需要花费很长时间。

1.1　分布式编程概述

什么是分布式编程？现在几乎很少有人再敢问这个问题。这个术语现在是如此普及，以至于去询问它的含义会让人觉得非常尴尬。而其他人则认为没有必要再去询问这个术语的含义。当我在按照惯例让学生定义分布式编程时，却很少能得到相同的答案。

分布式编程的特点是让几个物理上独立的组件作为一个单独的系统协同工作。在这里，"物理上独立的组件"可能指多个 CPU，或者更普遍的是指网络中的多台计算机。分布式编程可用于解决很多类型的问题，从预测天气到购买图书。作为分布式编程的核心，它做了如下的假定：如果一台计算机能够在 5 秒钟内完成一项任务，那么 5 台计算机以并行的方式一起工作时就能在 1 秒钟内完成一项任务。

当然，分布式编程不会如此简单。问题就在于"以并行方式协同工作"，很难让网络中的 5 台计算机高效协作。实际上，要达到如此高效，应用软件必须经过特殊设计。对此可以举一个只有一匹马拉车的例子。马是强大的动物，但是从力量与重量之比来说，蚂蚁要比马强壮很多倍(这里仅假设强壮 10 倍)。这样，如果聚集了与一匹马质量相同的一堆蚂蚁并利用它们来工作，则可以拉动 10 倍于一匹马所能拉的物质。这是一个非常好的分布负载示例。这种计算是合理的，但让数百万的蚂蚁身上都套着细小的缰绳去拉动货物却是不现实的。

1.1.1 应用程序的分层

通过马与蚂蚁的类比说明，分布式计算提出几台计算机协同工作的问题。这也是将应用程序分解为几个可被分布式处理的任务的问题。幸运的是，我们可以利用从以前的应用程序中学到的知识。经过这些年的发展，可以清楚地了解到大多数业务应用程序是由 3 个主要逻辑部分构成：表示逻辑，业务逻辑和数据源逻辑。

- 表示逻辑。表示逻辑是应用程序的一部分，终端用户可以通过它输入订单、查找用户信息和查看业务报表。对于用户来说，这部分逻辑就是应用程序。
- 业务逻辑。这部分是应用程序的中心，开发人员将在这里花费大部分的时间和精力。它包括定义业务运行方式的业务规则。例如，业务逻辑规定客户何时收到折扣、如何计算运费以及订单上所必需的信息。
- 数据源逻辑。这部分逻辑用于保存将来可能用到的订单、客户信息以及其他一些信息。幸运的是，SQL Server 和 Oracle 等数据库产品会实现大部分工作。不过您仍然需要设计数据层以及检索数据所使用的查询。

设计任何商业应用程序的首要之处是将应用程序的各个部分逻辑划分为不同的层次。换句话说，不能将业务逻辑代码与表示逻辑代码混在一起。然而，不要想当然地认为每一层必须运行在单独的机器上或单独的进程中。除此之外，每一层的代码只能通过定义良好的界面与另一层的代码进行交互。典型的情况是在独立的代码库(DLLs)中从物理上实现某些层。

1.1.2 分布式设计的 5 个原则

分层结构允许在不影响其他层的情况下修改某一层的实现。同时，它也允许将来从物理上灵活地分隔各个层。但是，正如下面紧接着的部分中所述，不应该轻易决定在独立进程或机器上执行每一层。如果您决定对某一层进行分布处理，那么必须对它进行特殊的分布设计。令人迷惑的是，某些设计策略实际上与传统的面向对象原则相矛盾。为弄清这些问题，这一部分阐述了几项用于有效分布应用程序的原则以及使用这些原则的理由。

原则 1：保守分布

对于分布式编程的书籍来说，这个原则看起来让人有些惊奇。然而，这项原则却是基于计算中一个简单且不可否认的事实：调用不同进程上对象的方法要比调用进程内对象的方法慢数百倍；将对象移动到网络中的另一台计算机上，这种方法调用又会慢数十倍。

那什么时候才应当进行分布式处理呢？以前的观点是只有必须进行分布时才这样做。但是您可能想了解更多的细节，所以让我们从数据层开始考虑几个示例。通常，应用的数据库运行在独立的专用服务器上——换句话说，它相对于其他层是分布式的。这样做有几个很好的理由：

- 数据库软件复杂而昂贵，而且通常需要高性能的硬件，所以分布数据库软件的多个副本将导致开销太大。
- 数据库可包含和关联由许多应用程序共享的数据。但是，只有当每个应用程序正在访问单独的数据库服务器而不是自己的本地副本时，这种情况才可能发生。
- 数据库被设计为运行在独立的物理层上。它们提供最终的“chunky”接口：结构化查询语言(SQL)。(请参考原则 3 以获得与 chunky 接口相关的细节。)

因此，当您决定使用数据库时，一般需要决定分布数据源逻辑。然而，决定分布表示逻辑会更复杂一些。首先，除非所有应用程序用户都使用公共的终端(例如 ATM)，否则表示层的某些部分就必须分布到每个用户机上。但问题是分布到什么程度。当然，近来的趋势是在服务器上执行大部分逻辑，而将简单的 HTML 发送到客户端 Web 浏览器。实际上，这正是遵守了保守分布的原则。然而，它也需要每个用户交互都遍历服务器，从而这些用户交互才能产生正确的响应。

在 Web 迅速发展之前，普遍的情况是在每个客户机上执行整个表示逻辑(遵守原则 2)。这样可与用户更快地交互，因为它最小化了服务器上的遍历，但它也需要更新用户接口并被部署到整个用户群。最后，选择使用哪一个客户机基本上与分布设计原则无关，但都与期望的用户经验和部署问题有关。

数据逻辑几乎总是在一个独立的计算机上执行，表示层通常也是如此。现在只剩下业务逻辑，它是整个问题组中最复杂的部分。业务层有时被部署到每个用户，而其他时候则被保存到服务器上。在许多情况下，业务层被分解成两个或更多个组件。与用户接口交互相关的组件被部署到客户机处，而与数据访问相关的组件则被保存到服务器上。这就遵守了下一个原则，即相关内容本地化。

可以看到，您有许多分布选项。分布的时间、原因以及如何分布等受到很多因素的影响——其中的许多因素又互相竞争。下面几个原则会提供进一步的指导。

原则 2：本地化相关内容

如果决定或被迫分布全部或部分业务逻辑层，那么应当保证经常交互的组件被放置在一起。换句话说，您应当本地化相关内容。例如，参考图 1-1 所示的电子商务应用。这个应用程序将客户组件、商品组件和购物车组件分隔到指定的服务器上，这会在表面上允许并行执行。然而，当一件商品添加到购物车时，组件之间就会进行多次交互。每一次交互都会带来跨网络方法调用的系统开销。因此，这种跨网络的行为会抵消掉并行处理带来的好处。再考虑到几千用户会同时使用，可想其后果是破坏性的。如果还用前面提及的马和马车的类比，这种情况就等同于利用马的每条腿而不是整匹马。

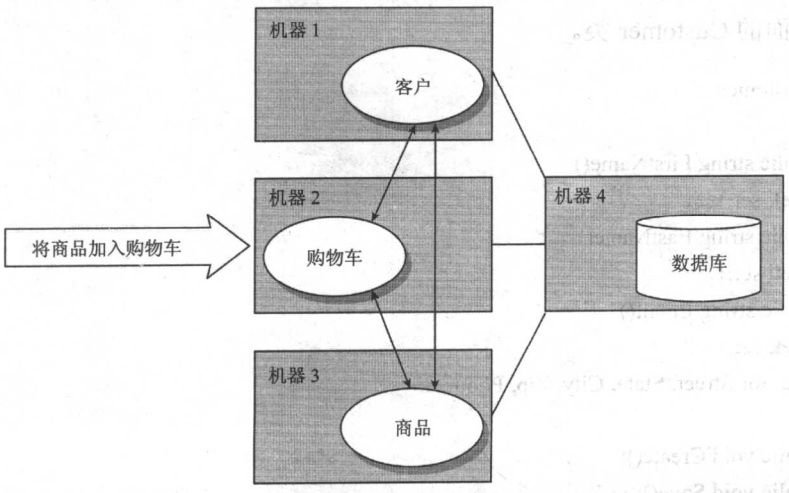


图 1-1　一个不成功的分布式应用例子

在本地化相关内容的同时，如何利用分布式编程(也就是并行处理)的作用呢？再买一匹马？那就意味着复制整个应用程序并使它运行在另一台专门的服务器上。可以使用负载平衡方法将每个客户机请求路由到特殊的服务器，即如图 1-2 所示的这种结构。基于 Web 的应用程序经常通过在几个 Web 服务器上驻留相同的 Web 站点来使用这种模型，有时该设置被称为 Web 场。

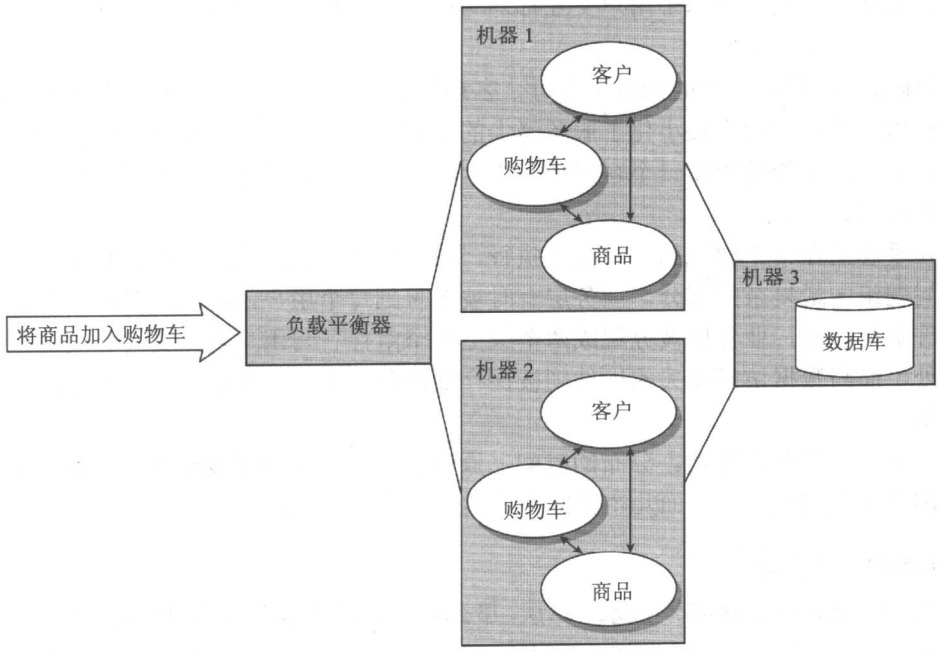


图 1-2　一个成功的分布式应用示例

对应用程序服务器进行复制和均衡负载可以很好地提高应用程序的容量或不伸缩性。然而，您需要非常清楚如何管理状态。更详细的信息请参考原则 4。

原则 3：使用 Chunky 接口，而不是 chatty 接口

面向对象编程的思想之一是创建具有许多简单方法的对象，每一个方法都专注于一个特殊的行为。考虑下面的 Customer 类。

```
Class Customer
{
    public string FirstName()
    { get; set;}
    public string LastName()
    { get; set;}
    public string Email()
    { get; set;}
    //etc. for Street, State, City, Zip, Phone ...

    public void Create();
    public void Save();
}
```

这种实现会受到大多数面向对象专家的肯定。但是，我的第一反应是：相对于调用代码而言该对象在何处运行。如果直接在过程中访问 Customer 类，即使从大多数标准来看，这种设计也是非常正确的。但是，如果这个类被执行在其他过程或机器中的代码所调用，则无论现在或是将来，这种设计都是非常糟糕的。要了解具体原因，可考虑下面的代码，并且设想它正运行在纽约的客户机上，而 Customer 对象运行在伦敦的服务器上。

```
Static void ClientCode()
{
    Customer cust = new Customer();
    cust.Create();
    cust.FirstName = "Nigel ";
    cust.LastName = "Tufnel ";
    cust.Email = "ntufnel@spinaltap.com ";
    //etc. for Street, State, City, Zip, Phone...

    cust.Save();
}
```

与前面相同的是，如果 Customer 对象位于客户机进程中，则这个示例不会产生任何问题。可是，设想一下每个属性和方法调用都要跨越大西洋进行遍历，这将会产生很严重的性能问题。

这个 Customer 类具有典型的 chatty 接口，或被更专业地称作细粒度接口。相反，哪怕是被进程外代码偶尔访问的对象也应当被设计成具有 chunky 接口，或者说是粗粒度接口。下面是具有 chunky 接口的 Customer 类。

```
Class Customer
{
    public void Create(string FirstName, string LastName, string Email,
                       //etc for Street, State, City, Zip, Phone ...
                      );
    public void Save(string FirstName, string LastName, string Email,
                     //etc for Street, State, City, Zip, Phone ...
                    );
}
```

可以看出，这段代码不如第一个 Customer 类那么清晰。但是相对于前者所具有的更多面向对象的特点而言，当 Web 站点的访问量突然变大，以至于需要扩充容量来满足新用户的访问时，后者却会提供更多的保护。

顺带提一下，可以简化具有 chunky 接口的 Customer 类。不需要将每一份客户数据作为独立的参数来传输，可以将客户数据封装到一个客户类中，而只需传输这个客户类。下面就是这种情况的示例。

```
[Serializable] // <-- Explained in Chapter 2!
class CustomerData
```

```
{
    public string FirstName()
    { get; set;}
    public string LastName()
    { get; set;}
    public string Email()
    { get; set;}
    //etc for Street, State, City, Zip, Phone ...
}

class Customer
{
    public void Create(CustomerData data);
    public void Save(CustomerData data);
}
```

初看这段代码，它将 chatty 接口从 Customer 类移到了 Customerdata 类中。这样做有什么好处？关键之处是在 CustomerData 类定义前的 Serializable 特性。它告诉.NET 运行库，只要对象越出进程边界就复制整个对象。因此，当客户机代码调用 CustomerData 类的属性时，实际上在访问一个本地对象。在第 2 章和第 3 章中会进一步讨论串行化和可串行化对象。

原则 4：优先选用无状态对象，而不是有状态对象

如果上一个原则违背了面向对象拥护者的看法，那这个原则可能会激怒他们。与严格的面向对象定义相比，术语“无状态对象”就显得有些矛盾。然而，如果想利用在图 1-2 中显示的负载平衡体系结构，您就需要仔细管理分布式对象中的状态，或者干脆不使用状态。要记住，这条原则和原则 3 一样只能适用于分布在边界上并可能跨越进程边界的对象。而进程内的对象则可以自由地保存状态，而不会危及应用程序的可伸缩性。

无状态对象这个术语看起来会在开发人员中引起混淆。下面尽可能简洁地定义它：无状态对象是能够在方法调用之间被安全创建和销毁的对象。这是个简单的定义，但包含很多含义。首先，注意“能够”这个词。应用程序不需要在方法调用之后销毁无状态对象。但如果应用程序选择销毁它，这个动作也不会影响到其他用户。这个特性不是轻易就能实现的，您必须对类进行特殊实现，这样类才不会依赖于公共方法调用之后实例字段是否继续存在。因为对实例字段没有依赖关系，所以无状态对象倾向于使用 chunky 接口。

有两个原因可说明有状态对象对于可伸缩性具有负面的影响。首先，有状态对象通常会在服务器上存在很长一段时间。在它的生存期中，它会聚集并使用服务器资源。这样即使有状态对象不工作或正在等待其他用户的请求，它也会阻止其他对象使用这些资源。虽然一些人认为内存是资源竞争中的关键资源，但实际上这只是相对次要的因素。如图 1-3 所示，有状态对象是耗费稀有资源(例如数据库连接)的真正罪魁祸首。