



注意 许多人奇怪事件模式为什么要求 **sender** 参数是 **Object** 类型。毕竟,只有 **MailManager** 才会引发传递了 **NewMailEventArgs** 对象的事件,所以回调方法更合适的原型似乎是下面这个:

```
void MethodName(MailManager sender, NewMailEventArgs e);
```

要求 **sender** 是 **Object** 主要是因为继承。例如,假定 **MailManager** 成为 **SmtpMailManager** 的基类,那么回调方法的 **sender** 参数应该是 **SmtpMailManager** 类型而不是 **MailManager** 类型。但这不可能发生,因为 **SmtpMailManager** 继承了 **NewMail** 事件。所以,如果代码需要由 **SmtpMailManager** 引发事件,还是要将 **sender** 实参转型为 **SmtpMailManager**。反正都要进行类型转换,这和将 **sender** 定为 **Object** 类型没什么两样。

将 **sender** 参数的类型定为 **Object** 的另一个原因是灵活性。它使委托能由多个类型使用,只要类型提供了一个会传递 **NewMailEventArgs** 对象的事件。例如,即使 **PopMailManager** 类不是从 **MailManager** 类派生的,也能使用这个委托。

此外,事件模式要求委托定义和回调方法将派生自 **EventArgs** 的参数命名为 **e**。这个要求唯一的作用就是加强事件模式的一致性,使开发人员更容易学习和实现这个模式。注意,能自动生成源代码的工具(比如 Microsoft Visual Studio)也知道将参数命名为 **e**。

最后,事件模式要求所有事件处理程序^①的返回类型都是 **void**。这很有必要,因为引发事件后可能要调用好几个回调方法,但没办法获得所有方法的返回值。将返回类型定为 **void**,就不允许回调(方法)返回值。遗憾的是,FCL 中的一些事件处理程序(比如 **ResolveEventHandler**)没有遵循 Microsoft 自定的模式。例如,**ResolveEventHandler** 事件处理程序会返回 **Assembly** 类型的对象。

11.1.3 第三步: 定义负责引发事件的方法来通知事件的登记对象

按照约定,类要定义一个受保护的虚方法。引发事件时,类及其派生类中的代码会调用该方法。方法只获取一个参数,即一个 **NewMailEventArgs** 对象,其中包含了传给接收通知的对象的信息。方法的默认实现只是检查一下是否有对象登记了对事件的关注。如果有,就引发事件来通知事件的登记对象。该方法在 **MailManager** 类中看起来像下面这样:

```
internal class MailManager {
    ...
    // 第三步: 定义负责引发事件的方法来通知已登记的对象。
    // 如果类是密封的,该方法要声明为私有和非虚
    protected virtual void OnNewMail(NewMailEventArgs e) {
        // 出于线程安全的考虑,现在将对委托字段的引用复制到一个临时变量中
        EventHandler<NewMailEventArgs> temp = Volatile.Read(ref NewMail);
        // 任何方法登记了对事件的关注,就通知它们
        if (temp != null) temp(this, e);
    }
    ...
}
```

① 本书按约定俗成的译法将 **event handler** 翻译成“事件处理程序”,但请把它理解成“事件处理方法”(在 VB 中,则理解成“事件处理 Sub 过程”)。——译注

以线程安全的方式引发事件

.NET Framework 刚发布时建议开发者用以下方式引发事件:

```
// 版本 1
protected virtual void OnNewMail(NewMailEventArgs e) {
    if (NewMail != null) NewMail(this, e);
}
```

OnNewMail 方法的问题在于, 虽然线程检查出 **NewMail** 不为 **null**, 但就在调用 **NewMail** 之前, 另一个线程可能从委托链中移除一个委托, 使 **NewMail** 成了 **null**。这会抛出 **NullReferenceException** 异常。为了修正这个竞态问题, 许多开发者都像下面这样写 **OnNewMail** 方法

```
// 版本 2
protected virtual void OnNewMail(NewMailEventArgs e) {
    EventHandler<NewMailEventArgs> temp = NewMail;
    if (temp != null) temp(this, e);
}
```

它的思路是, 将对 **NewMail** 的引用复制到临时变量 **temp** 中, 后者引用赋值发生时的委托链。然后, 方法比较 **temp** 和 **null**, 并调用 **(invoke)temp**; 所以, 向 **temp** 赋值后, 即使另一个线程更改了 **NewMail** 也没有关系。委托是不可变的(**immutable**), 所以这个技术理论上行得通。但许多开发者没有意识到的是, 编译器可能“擅做主张”, 通过完全移除局部变量 **temp** 的方式对上述代码进行优化。如果发生这种情况, 版本 2 就和版本 1 就没有任何区别。所以, 仍有可能抛出 **NullReferenceException** 异常。

要想整个修正这个问题, 应该像下面这样重写 **OnNewMail**:

```
// 版本 3
protected virtual void OnNewMail(NewMailEventArgs e) {
    EventHandler<NewMailEventArgs> temp = Volatile.Read(ref NewMail);
    if (temp != null) temp(this, e);
}
```

对 **Volatile.Read** 的调用强迫 **NewMail** 在这个调用发生时读取, 引用真的必须复制到 **temp** 变量中(编译器别想走捷径)。然后, **temp** 变量只有在不为 **null** 时才会被调用(**invoke**)。第 29 章“基元线程同步构造”将详细讨论 **Volatile.Read** 方法。

虽然最后一个版本很完美, 是技术正确的版本, 但版本 2 实际也是可以使用的, 因为 JIT 编译器理解这个模式, 知道自己不该将局部变量 **temp** “优化”掉。具体地说, Microsoft 的所有 JIT 编译器都“尊重”那些不会对堆内存的新的读取动作的不变量(**invariant**)。所以, 在局部变量中缓存一个引用, 可确保堆引用只被访问一次。这一点并未在文档中反映, 理论上说将来可能改变, 这正是为什么应该使用最后一个版本的原因。但实际上, Microsoft 的 JIT 编译器永远没有可能真的进行修改来破坏这个模式, 否则太多的应用程序都会“遭殃”^①。此外, 事件主要在单线程的情形 (WPF 和 Windows Store 应用) 中使用, 所以线程安全不是问题。

还要注意, 考虑到线程竞态条件^②, 方法有可能在从事件的委托链中移除之后得到调用。

① 这是 Microsoft 的 JIT 编译器团队的人告诉我的。

② 文档翻译成“争用状态”或“争用条件”。——译注

为方便起见，可定义扩展方法(参见第 8 章“方法”)来封装这个线程安全逻辑。如下所示：

```
public static class EventArgsExtensions {
    public static void Raise<TEventArgs>(this TEventArgs e,
        Object sender, ref EventHandler<TEventArgs> eventDelegate) {

        // 出于线程安全的考虑，现在将对委托字段的引用复制到临时字段
        EventHandler<TEventArgs> temp = Volatile.Read(ref eventDelegate);

        // 任何方法登记了对事件的关注就通知它们
        if (temp != null) temp(sender, e);
    }
}
```

现在可以像下面这样重写 **OnNewMail** 方法：

```
protected virtual void OnNewMail(NewMailEventArgs e) {
    e.Raise(this, ref m_NewMail);
}
```

以 **MailManager** 作为基类的类可自由重写 **OnNewMail** 方法。这使派生类能控制事件的引发，以自己的方式处理新邮件。一般情况下，派生类会调用基类的 **OnNewMail** 方法，使登记的方法能收到通知。但是，派生类也可以不允许事件转发。

11.1.4 第四步：定义方法将输入转化为期望事件

类还必须有一个方法获取输入并转化为事件的引发。在 **MailManager** 的例子中，是调用 **SimulateNewMail** 方法来指出一封新的电子邮件已到达 **MailManager**：

```
internal class MailManager {

    // 第四步：定义方法将输入转化为期望事件
    public void SimulateNewMail(String from, String to, String subject) {

        // 构造一个对象来容纳想传给通知接收者的信息
        NewMailEventArgs e = new NewMailEventArgs(from, to, subject);

        // 调用虚方法通知对象事件已发生，
        // 如果没有类型重写该方法，我们的对象将通知事件的所有登记对象
        OnNewMail(e);
    }
}
```

SimulateNewMail 接收关于邮件的信息并构造 **NewMailEventArgs** 对象，将邮件信息传给它的构造器。然后调用 **MailManager** 自己的虚方法 **OnNewMail** 来正式通知 **MailManager** 对象收到了新的电子邮件。这通常会导致事件的引发，从而通知所有已登记的方法。(如前所述，以 **MailManager** 为基类的类可能重写这个行为。)

11.2 编译器如何实现事件

知道如何定义提供了事件成员的类之后，接着研究一下事件是什么，以及它是如何工作的。**MailManager** 类用一行代码定义了事件成员本身：

```
public event EventHandler<NewMailEventArgs> NewMail;
```

C#编译器编译时把它转换为以下 3 个构造:

```
// 1. 一个被初始化为 null 的私有委托字段
```

```
private EventHandler<NewMailEventArgs> NewMail = null;
```

```
// 2. 一个公共 add_Xxx 方法 (其中 Xxx 是事件名)
```

```
// 允许方法登记对事件的关注
```

```
public void add_NewMail(EventHandler<NewMailEventArgs> value) {
```

```
    // 通过循环和对 CompareExchange 的调用, 可以
```

```
    // 以一种线程安全的方式向事件添加委托
```

```
    EventHandler<NewMailEventArgs> prevHandler;
```

```
    EventHandler<NewMailEventArgs> newMail = this.NewMail;
```

```
    do {
```

```
        prevHandler = newMail;
```

```
        EventHandler<NewMailEventArgs> newHandler =
```

```
            (EventHandler<NewMailEventArgs>) Delegate.Combine(prevHandler, value);
```

```
        newMail = Interlocked.CompareExchange<EventHandler<NewMailEventArgs>>(
```

```
            ref this.NewMail, newHandler, prevHandler);
```

```
    } while (newMail != prevHandler);
```

```
}
```

```
// 3. 一个公共 remove_Xxx 方法 (其中 Xxx 是事件名)
```

```
// 允许方法注销对事件的关注
```

```
public void remove_NewMail(EventHandler<NewMailEventArgs> value) {
```

```
    // 通过循环和对 CompareExchange 的调用, 可以
```

```
    // 以一种线程安全的方式从事件中移除一个委托
```

```
    EventHandler<NewMailEventArgs> prevHandler;
```

```
    EventHandler<NewMailEventArgs> newMail = this.NewMail;
```

```
    do {
```

```
        prevHandler = newMail;
```

```
        EventHandler<NewMailEventArgs> newHandler =
```

```
            (EventHandler<NewMailEventArgs>) Delegate.Remove(prevHandler, value);
```

```
        newMail = Interlocked.CompareExchange<EventHandler<NewMailEventArgs>>(
```

```
            ref this.NewMail, newHandler, prevHandler);
```

```
    } while (newMail != prevHandler);
```

```
}
```

第一个构造是具有恰当委托类型的字段。该字段是对一个委托列表的头部的引用。事件发生时通知这个列表中的委托。字段初始化为 **null**, 表明无侦听器(listener)登记对该事件的关注。一个方法登记对事件的关注时, 该字段会引用 **EventHandler<NewMailEventArgs>** 委托的实例, 后者可能引用更多的 **EventHandler<NewMailEventArgs>** 委托。侦听器登记对事件的关注时, 只需将委托类型的一个实例添加到列表中。显然, 注销(对事件的关注)意味着从列表中移除委托。

注意, 即使原始代码行将事件定义为 **public**, 委托字段(本例是 **NewMail**)也始终是 **private**。将委托字段定义为 **private**, 目的是防止类外部的代码不正确地操纵它。如果字段是 **public**, 任何代码都能更改字段中的值, 并可能删除已登记了对事件的关注的委托。

C#编译器生成的第二个构造是一个方法, 允许其他对象登记对事件的关注。C#编译器在事件名(**NewMail**)之前附加 **add_**前缀, 从而自动命名该方法。C#编译器还自动为方法生成代码。生成的代码总是调用 **System.Delegate** 的静态 **Combine** 方法, 它将委托实例添加到委托列表中, 返回新的列表头(地址), 并将这个地址存回字段。

C#编译器生成的第三个构造是一个方法, 允许对象注销对事件的关注。同样地, C#编译器

在事件名(**NewMail**)之前附加 **remove_** 前缀,从而自动命名该方法。方法中的代码总是调用 **Delegate** 的静态 **Remove** 方法,将委托实例从委托列表中删除,返回新的列表头(地址),并将这个地址存回字段。



警告 试图删除从未添加过的方法, **Delegate** 的 **Remove** 方法在内部不做任何事情。也就是说,不会抛出任何异常,也不会显示任何警告;事件的方法集合保持不变。



注意 **add** 和 **remove** 方法以线程安全的一种模式更新值。该模式的详情将在 29.3.4 节“**Interlocked Anything 模式**”讨论。

在本例中, **add** 和 **remove** 方法的可访问性都是 **public**。这是因为源代码将事件声明为 **public**。如果事件声明为 **protected**,编译器生成的 **add** 和 **remove** 方法也会被声明为 **protected**。因此,在类型中定义事件时,事件的可访问性决定了什么代码能登记和注销对事件的关注。但无论如何,只有类型本身才能直接访问委托字段。事件成员也可声明为 **static** 或 **virtual**。在这种情况下,编译器生成的 **add** 和 **remove** 方法分别标记为 **static** 或 **virtual**。

除了生成上述 3 个构造,编译器还会在托管程序集的元数据中生成一个事件定义记录项。这个记录项包含了一些标志(flag)和基础委托类型(underlying delegate type),还引用了 **add** 和 **remove** 访问器方法。这些信息的作用很简单,就是建立“事件”的抽象概念和它的访问器方法之间的联系。编译器和其他工具可利用这些元数据信息,并可通过 **System.Reflection.EventInfo** 类获取这些信息。但是,CLR 本身并不使用这些元数据信息,它在运行时只需要访问器方法。

11.3 设计侦听事件的类型

最难的部分已经完成了,接下来是一些较为简单的事情。本节将演示如何定义一个类型来使用另一个类型提供的事件。先来看看 **Fax** 类型的代码:

```
internal sealed class Fax {
    // 将 MailManager 对象传给构造器
    public Fax(MailManager mm) {

        // 构造 EventHandler<NewMailEventArgs> 委托的一个实例,
        // 使它引用我们的 FaxMsg 回调方法
        // 向 MailManager 的 NewMail 事件登记我们的回调方法
        mm.NewMail += FaxMsg;
    }

    // 新电子邮件到达时, MailManager 将调用这个方法
    private void FaxMsg(Object sender, NewMailEventArgs e) {

        // 'sender' 表示 MailManager 对象, 便于将信息传回给它

        // 'e' 表示 MailManager 对象想传给我们的附加事件信息
    }
}
```

```

// 这里的代码正常情况下应该传真电子邮件,
// 但这个测试性的实现只是在控制台上显示邮件
Console.WriteLine("Faxing mail message:");
Console.WriteLine(" From={0}, To={1}, Subject={2}",
    e.From, e.To, e.Subject);
}

// 执行这个方法, Fax 对象将向 NewMail 事件注销自己对它的关注,
// 以后不再接收通知
public void Unregister(MailManager mm) {

    // 向 MailManager 的 NewMail 事件注销自己对这个事件的关注
    mm.NewMail -= FaxMsg;
}
}

```

电子邮件应用程序初始化时首先构造 **MailManager** 对象, 并将对该对象的引用保存到变量中。然后构造 **Fax** 对象, 并将 **MailManager** 对象引用作为实参传递。在 **Fax** 构造器中, **Fax** 对象使用 C# 的 += 操作符登记它对 **MailManager** 的 **NewMail** 事件的关注:

```
mm.NewMail += FaxMsg;
```

C# 编译器内建了对事件的支持, 会将 += 操作符翻译成以下代码来添加对象对事件的关注:

```
mm.add_NewMail(new EventHandler<NewMailEventArgs>(this.FaxMsg));
```

C# 编译器生成的代码构造一个 **EventHandler<NewMailEventArgs>** 委托对象, 其中包装了 **Fax** 类的 **FaxMsg** 方法。接着, C# 编译器调用 **MailManager** 类的 **add_NewMail** 方法, 向它传递新的委托对象。为了对此进行验证, 可编译代码并用 **ILDasm.exe** 这样的工具查看 IL 代码。

即使使用的编程语言不直接支持事件, 也可显式调用 **add** 访问器方法向事件登记委托。两者效果一样, 只是后者的源代码看起来没那么优雅。两者最终都是用 **add** 访问器将委托添加到事件的委托列表中, 从而完成委托向事件的登记。

MailManager 对象引发事件时, **Fax** 对象的 **FaxMsg** 方法会被调用。调用这个方法时, 会传递 **MailManager** 对象引用作为它的第一个参数, 即 **sender**。该参数大多数时候会被忽略。但如果 **Fax** 对象希望在响应事件时访问 **MailManager** 对象的成员, 它就能派上用场了。第二个参数是 **NewMailEventArgs** 对象引用。对象中包含 **MailManager** 和 **NewMailEventArgs** 的设计者认为对事件接收者来说有用的附加信息。

FaxMsg 方法可从 **NewMailEventArgs** 对象中轻松访问邮件的发件人、收件人以及主题。真实的 **Fax** 对象应将这些信息传真到某处。本例只是在控制台窗口显示。

对象不再希望接收事件通知时, 应注销对事件的关注。例如, 如果不再希望将电子邮件转发到一台传真机, **Fax** 对象就应该注销它对 **NewMail** 事件的关注。对象只要向事件登记了它的一个方法, 便不能被垃圾回收。所以, 如果你的类型要实现 **IDisposable** 的 **Dispose** 方法, 就应该在实现中注销对所有事件的关注。**IDisposable** 的详情参见第 21 章“托管堆和垃圾回收”。

Fax 的 **Unregister** 方法示范了如何注销对事件的关注。该方法和 **Fax** 构造器中的代码十分

相似。唯一区别是使用 `-=` 而不是 `+=`。C# 编译器看到代码使用 `-=` 操作符向事件注销委托时，会生成对事件的 **remove** 方法的调用：

```
mm.remove_NewMail(new EventHandler<NewMailEventArgs>(FaxMsg));
```

和 `+=` 操作符一样，即使编程语言不直接支持事件，也可显式调用 **remove** 访问器方法向事件注销委托。**remove** 方法为了向事件注销委托，需要扫描委托列表来寻找一个恰当的委托（其中包装的方法和传递的方法相同）。找到匹配，现有委托会从事件的委托列表中删除。没有找到也不会报错，列表不发生任何变动。

顺便说一下，C# 要求代码使用 `+=` 和 `-=` 操作符在列表中增删委托。如果显式调用 **add** 和 **remove** 方法，C# 编译器会报告以下错误消息：**CS0571: 无法显式调用运算符或访问器。**

11.4 显式实现事件

System.Windows.Forms.Control 类型定义了大约 70 个事件。假如 **Control** 类型在实现事件时，允许编译器隐式生成 **add** 和 **remove** 访问器方法以及委托字段，那么每个 **Control** 对象仅为事件就要准备 70 个委托字段！由于大多数程序员只关心少数几个事件，所以每个从 **Control** 派生类型创建的对象都要浪费大量内存。顺便说一下，ASP.NET 的 **System.Web.UI.Control** 类型和 WPF 的 **System.Windows.UIElement** 类型也提供了大多数程序员都用不上的大量事件。

本节将讨论 C# 编译器如何允许类的开发人员显式实现一个事件，使开发人员能够控制 **add** 和 **remove** 方法处理回调委托的方式。我要演示如何通过显式实现事件来高效率地实现提供了大量事件的类，但肯定还有其他情形也需要显式实现事件。

为了高效率存储事件委托，公开了事件的每个对象都要维护一个集合（通常是字典）。集合将某种形式的事件标识符作为键(key)，将委托列表作为值(value)。新对象构造时，这个集合是空白的。登记对一个事件的关注时，会在集合中查找事件的标识符。如果事件标识符已在其中，新委托就和这个事件的委托列表合并。如果事件标识符不在集合中，就添加事件标识符和委托。

对象需要引发事件时，会在集合中查找事件标识符。如果集合中没有找到事件标识符，表明还没有任何对象登记对这个事件的关注，所以没有任何委托需要回调。如果事件标识符在集合中，就调用与它关联的委托列表。具体怎么实现这个设计模式，是定义事件的那个类型的开发人员的责任；使用类型的开发人员不知道事件在内部如何实现。

下例展示了如何完成这个模式。首先实现一个 **EventSet** 类，它代表一个集合，其中包含事件以及每个事件的委托列表。

```
using System;
using System.Collections.Generic;
using System.Threading;
```

```
// 这个类的目的是在使用 EventSet 时，提供
```

```

// 多一点的类型安全性和代码可维护性
public sealed class EventKey { }

public sealed class EventSet {
    // 该私有字典用于维护 EventKey -> Delegate 映射
    private readonly Dictionary<EventKey, Delegate> m_events =
        new Dictionary<EventKey, Delegate>();

    // 添加 EventKey -> Delegate 映射 (如果 EventKey 不存在),
    // 或者将委托和现有的 EventKey 合并
    public void Add(EventKey eventKey, Delegate handler) {
        Monitor.Enter(m_events);
        Delegate d;
        m_events.TryGetValue(eventKey, out d);
        m_events[eventKey] = Delegate.Combine(d, handler);
        Monitor.Exit(m_events);
    }

    // 从 EventKey (如果它存在) 删除委托, 并且
    // 在删除最后一个委托时删除 EventKey -> Delegate 映射
    public void Remove(EventKey eventKey, Delegate handler) {
        Monitor.Enter(m_events);
        // 调用 TryGetValue, 确保在尝试从集合中删除不存在的 EventKey 时不会抛出异常
        Delegate d;
        if (m_events.TryGetValue(eventKey, out d)) {
            d = Delegate.Remove(d, handler);

            // 如果还有委托, 就设置新的头部 (地址), 否则删除 EventKey
            if (d != null) m_events[eventKey] = d;
            else m_events.Remove(eventKey);
        }
        Monitor.Exit(m_events);
    }

    // 为指定的 EventKey 引发事件
    public void Raise(EventKey eventKey, Object sender, EventArgs e) {
        // 如果 EventKey 不在集合中, 不抛出异常
        Delegate d;
        Monitor.Enter(m_events);
        m_events.TryGetValue(eventKey, out d);
        Monitor.Exit(m_events);

        if (d != null) {
            // 由于字典可能包含几个不同的委托类型,
            // 所以无法在编译时构造一个类型安全的委托调用
            // 因此, 我调用 System.Delegate 类型的 DynamicInvoke
            // 方法, 以一个对象数组的形式向它传递回调方法的参数
            // 在内部, DynamicInvoke 会向调用的回调方法查证参数的
            // 类型安全性, 并调用方法
            // 如果存在类型不匹配的情况, DynamicInvoke 会抛出异常
            d.DynamicInvoke(new Object[] { sender, e });
        }
    }
}

```

接着定义一个类来使用 **EventSet** 类。在这个类中, 一个字段引用了一个 **EventSet** 对象, 而且这个类的每个事件都是显式实现的, 使每个事件的 **add** 方法都将指定的回调委托存储到 **EventSet** 对象中, 而且每个事件的 **remove** 方法都删除指定的回调委托(如果找得到的话)。

```

using System;

// 为这个事件定义从 EventArgs 派生的类型
public class FooEventArgs : EventArgs { }

public class TypeWithLotsOfEvents {

    // 定义私有实例字段来引用集合
    // 集合用于管理一组“事件/委托”对
    // 注意: EventSet 类型不是 FCL 的一部分, 它是我自己的类型
    private readonly EventSet m_eventSet = new EventSet();

    // 受保护的属性使派生类型能访问集合
    protected EventSet EventSet { get { return m_eventSet; } }

    #region 用于支持 Foo 事件的代码 (为附加的事件重复这个模式)
    // 定义 Foo 事件必要的成员
    // 2a. 构造一个静态只读对象来标识这个事件.
    // 每个对象都有自己的哈希码, 以便在对象的集合中查找这个事件的委托链表
    protected static readonly EventKey s_fooEventKey = new EventKey();

    // 2b. 定义事件的访问器方法, 用于在集合中增删委托
    public event EventHandler<FooEventArgs> Foo {
        add { m_eventSet.Add(s_fooEventKey, value); }
        remove { m_eventSet.Remove(s_fooEventKey, value); }
    }

    // 2c. 为这个事件定义受保护的虚方法 OnFoo
    protected virtual void OnFoo(FooEventArgs e) {
        m_eventSet.Raise(s_fooEventKey, this, e);
    }

    // 2d. 定义将输入转换成这个事件的方法
    public void SimulateFoo() { OnFoo(new FooEventArgs()); }
    #endregion
}

```

使用 **TypeWithLotsOfEvents** 类型的代码不知道事件是由编译器隐式实现, 还是由开发人员显式实现。它们只需用标准的语法向事件登记即可。以下代码进行了演示:

```

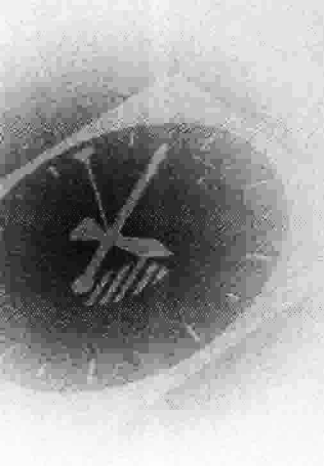
public sealed class Program {
    public static void Main() {
        TypeWithLotsOfEvents twle = new TypeWithLotsOfEvents();

        // 添加一个回调
        twle.Foo += HandleFooEvent;

        // 证明确实可行
        twle.SimulateFoo();
    }

    private static void HandleFooEvent(object sender, FooEventArgs e) {
        Console.WriteLine("Handling Foo Event here...");
    }
}

```



第 12 章 泛 型

本章内容:

- FCL 中的泛型
- 泛型基础结构
- 泛型接口
- 泛型委托
- 委托和接口的逆变和协变泛型类型实参
- 泛型方法
- 泛型和其他成员
- 可验证性和约束

熟悉面向对象编程的开发人员都深谙这种编程方式的好处。其中一个好处是“代码重用”，它极大提高了开发效率。也就是说，可以派生出一个类，让它继承基类的所有能力。派生类只需重写虚方法，或添加一些新方法，就可定制派生类的行为，使之满足开发人员的需求。泛型(generic)是 CLR 和编程语言提供了一种特殊机制，它支持另一种形式的代码重用，即“算法重用”。

简单地说，开发人员先定义好算法，比如排序、搜索、交换、比较或者转换等。但是，定义算法的开发人员并不设定该算法要操作什么数据类型；该算法可广泛地应用于不同类型的对象。然后，另一个开发人员只要指定了算法要操作的具体数据类型，就可以开始使用这个算法了。例如，一个排序算法可操作 **Int32** 和 **String** 等类型的对象，而一个比较算法可操作 **DateTime** 和 **Version** 等类型的对象。

大多数算法都封装在一个类型中，CLR 允许创建泛型引用类型和泛型值类型，但不允许创建泛型枚举类型。此外，CLR 还允许创建泛型接口和泛型委托。方法偶尔也封装有用的算法，所以 CLR 允许在引用类型、值类型或接口中定义泛型方法。

先来看一个简单的例子。Framework 类库(Framework Class Library, FCL)定义了一个泛型列表算法，它知道如何管理对象集合。泛型算法没有设定对象的数据类型。要在使用这个泛型列表算法时指定具体数据类型。

封装了泛型列表算法的 FCL 类称为 **List<T>**(读作 List of Tee)。这个类是在 **System.Collections.Generic** 命名空间中定义的。下面展示了类定义 (代码被大幅简化):

```
[Serializable]
public class List<T> : IList<T>, ICollection<T>, IEnumerable<T>,
    IList, ICollection, IEnumerable {

    public List();
    public void Add(T item);
    public Int32 BinarySearch(T item);
    public void Clear();
    public Boolean Contains(T item);
    public Int32 IndexOf(T item);
    public Boolean Remove(T item);
    public void Sort();
    public void Sort(IComparer<T> comparer);
    public void Sort(Comparison<T> comparison);
    public T[] ToArray();

    public Int32 Count { get; }
    public T this[Int32 index] { get; set; }
}
```

泛型 **List** 类的设计者紧接在类名后添加了一个 **<T>**，表明它操作的是一个未指定的数据类型。定义泛型类型或方法时，为类型指定的任何变量(比如 **T**)都称为**类型参数**(type parameter)。**T** 是变量名，源代码能使用数据类型的任何地方都能使用 **T**。例如，在 **List** 类定义中，**T** 被用于方法参数(**Add** 方法接受一个 **T** 类型的参数)和返回值(**ToArray** 方法返回 **T** 类型的一维数组)。另一个例子是索引器方法(在 C# 中称为 **this**)。索引器有一个 **get** 访问器方法，它返回 **T** 类型的值；一个 **set** 访问器方法，它接受 **T** 类型的参数。由于凡是能指定一个数据类型的地方都能使用 **T** 变量，所以在方法内部定义一个局部变量时，或者在类型中定义字段时，也可以使用 **T**。



注意 根据 Microsoft 的设计原则，泛型参数变量要么称为 **T**，要么至少以大写 **T** 开头(如 **TKey** 和 **TValue**)。大写 **T** 代表类型(Type)，就像大写 **I** 代表接口(Interface)一样，比如 **IComparable**。

定义好泛型 **List<T>** 类型之后，其他开发人员为了使用这个泛型算法，要指定由算法操作的具体数据类型。使用泛型类型或方法时指定的具体数据类型称为**类型实参**(type argument)。例如，开发人员可指定一个 **DateTime** 类型实参来使用 **List** 算法。以下代码对此进行了演示：

```
private static void SomeMethod() {
    // 构造一个 List 来操作 DateTime 对象
    List<DateTime> dtList = new List<DateTime>();

    // 向列表添加一个 DateTime 对象
    dtList.Add(DateTime.Now); // 不进行装箱

    // 向列表添加另一个 DateTime 对象
    dtList.Add(DateTime.MinValue); // 不进行装箱

    // 尝试向列表中添加一个 String 对象
    dtList.Add("1/1/2004"); // 编译时错误

    // 从列表提取一个 DateTime 对象
    DateTime dt = dtList[0]; // 不需要转型
}
```

从以上代码可看出泛型为开发人员提供了以下优势。

- **源代码保护**

使用泛型算法的开发人员不需要访问算法的源代码。然而，使用 C++ 模板的泛型技术时，算法的源代码必须提供给准备使用算法的用户。

- **类型安全**

将泛型算法应用于一个具体的类型时，编译器和 CLR 能理解开发人员的意图，并保证只有与指定数据类型兼容的对象才能用于算法。试图使用不兼容类型的对象会造成编译时错误，或在运行时抛出异常。在上例中，试图将 **String** 对象传给 **Add** 方法造成编译器报错。

- **更清晰的代码**

由于编译器强制类型安全性，所以减少了源代码中必须进行的强制类型转换次数，使代码更容易编写和维护。在 **SomeMethod** 的最后一行，开发人员不需要进行 **(DateTime)** 强制类型转换就能将索引器的结果(查询索引位置 0 处的元素)存储到 **dt** 变量中。

- **更佳的性能**

没有泛型的时候，要想定义常规化的算法，它的所有成员都要定义成操作 **Object** 数据类型。要用这个算法来操作值类型的实例，CLR 必须在调用算法的成员之前对值类型实例进行装箱。正如第 5 章“基元类型、引用类型和值类型”讨论的那样，装箱造成在托管堆上进行内存分配，造成更频繁的垃圾回收，从而损害应用程序的性能。由于现在能创建一个泛型算法来操作一种具体的值类型，所以值类型的实例能以传值方式传递，CLR 不再需要执行任何装箱操作。此外，由于不再需要进行强制类型转换(参见上一条)，所以 CLR 无需验证这种转型是否类型安全，这同样提高了代码的运行速度。

为了理解泛型的性能优势，我写了一个程序来比较泛型 **List** 算法和 FCL 的非泛型 **ArrayList** 算法的性能。我打算同时使用值类型的对象和引用类型的对象来测试这两个算法的性能。

下面是程序本身：

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Diagnostics;

public static class Program {
    public static void Main() {
        ValueTypePerfTest();
        ReferenceTypePerfTest();
    }

    private static void ValueTypePerfTest() {
        const Int32 count = 100000000;

        using (new OperationTimer("List<Int32>")) {
            List<Int32> l = new List<Int32>();
            for (Int32 n = 0; n < count; n++) {
                l.Add(n);           // 不发生装箱
                Int32 x = l[n];     // 不发生拆箱
            }
        }
    }
}
```

```

        l = null; //确保进行垃圾回收
    }

    using (new OperationTimer("ArrayList of Int32")) {
        ArrayList a = new ArrayList();
        for (Int32 n = 0; n < count; n++) {
            a.Add(n); // 发生装箱
            Int32 x = (Int32) a[n]; // 发生拆箱
        }
        a = null; // 确保进行垃圾回收
    }
}

private static void ReferenceTypePerfTest() {
    const Int32 count = 100000000;

    using (new OperationTimer("List<String>")) {
        List<String> l = new List<String>();
        for (Int32 n = 0; n < count; n++) {
            l.Add("X"); // 复制引用
            String x = l[n]; // 复制引用
        }
        l = null; // 确保进行垃圾回收
    }

    using (new OperationTimer("ArrayList of String")) {
        ArrayList a = new ArrayList();
        for (Int32 n = 0; n < count; n++) {
            a.Add("X"); // 复制引用
            String x = (String) a[n]; // 检查强制类型转换 & 复制引用
        }
        a = null; // 确保进行垃圾回收
    }
}

// 这个类用于进行运算性能计时
internal sealed class OperationTimer : IDisposable {
    private Stopwatch m_stopwatch;
    private String m_text;
    private Int32 m_collectionCount;

    public OperationTimer(String text) {
        PrepareForOperation();

        m_text = text;
        m_collectionCount = GC.CollectionCount(0);

        // 这应该是方法的最后一个语句, 从而最大程度保证计时的准确性
        m_stopwatch = Stopwatch.StartNew();
    }

    public void Dispose() {
        Console.WriteLine("{0} (GCs={1,3}) {2}", (m_stopwatch.Elapsed),
            GC.CollectionCount(0) - m_collectionCount, m_text);
    }

    private static void PrepareForOperation() {
        GC.Collect();
        GC.WaitForPendingFinalizers();
        GC.Collect();
    }
}

```

在我的机器上编译并运行这个程序 (打开优化开关), 得到以下输出:

```
00:00:01.6246959 (GCs= 6) List<Int32>
00:00:10.8555008 (GCs=390) ArrayList of Int32
00:00:02.5427847 (GCs= 4) List<String>
00:00:02.7944831 (GCs= 7) ArrayList of String
```

这证明在操作 **Int32** 类型时，泛型 **List** 算法比非泛型 **ArrayList** 算法快得多。1.6 秒和 11 秒，约 7 倍的差异！此外，用 **ArrayList** 操作值类型(**Int32**)会造成大量装箱，最终要进行 390 次垃圾回收。对应地，**List** 算法只需进行 6 次。

不过，用引用类型来测试，差异就没那么明显了，时间和垃圾回收次数基本相同。所以，泛型 **List** 算法在这里表面上没什么优势。但要注意的是，泛型算法的优势还包括更清晰的代码和编译时的类型安全。所以，虽然性能提升不是很明显，但泛型算法的其他优势也不容忽视。



注意 应该意识到，首次为特定数据类型调用方法时，CLR 都会为这个方法生成本机代码。这会增大应用程序的工作集(working set)大小，从而损害性能。12.2 节“泛型基础结构”将进一步探讨这个问题。

12.1 FCL 中的泛型

泛型最明显的应用就是集合类。FCL 在 **System.Collections.Generic** 和 **System.Collections.ObjectModel** 命名空间中提供了多个泛型集合类。**System.Collections.Concurrent** 命名空间则提供了线程安全的泛型集合类。Microsoft 建议使用泛型集合类，不建议使用非泛型集合类。这是出于几方面的考虑。首先，使用非泛型集合类，无法像使用泛型集合类那样获得类型安全性、更清晰的代码以及更佳的性能。其次，泛型类具有比非泛型类更好的对象模型。例如，虚方法数量显著变少，性能更好。另外，泛型集合类增添了一些新成员，为开发人员提供了新的功能。

集合类实现了许多接口，放入集合中的对象可实现接口来执行排序和搜索等操作。FCL 包含许多泛型接口定义，所以使用接口时也能享受到泛型带来的好处。常用接口在 **System.Collections.Generic** 命名空间中提供。

新的泛型接口不是为了替代旧的非泛型接口。许多时候两者都要使用(为了向后兼容)。例如，如果 **List<T>** 类只实现了 **ICollection<T>** 接口，代码就不能将一个 **List<DateTime>** 对象当作一个 **ICollection** 来处理。

还要注意，**System.Array** 类(所有数组类型的基类)提供了大量静态泛型方法，比如 **AsReadOnly**, **BinarySearch**, **ConvertAll**, **Exists**, **Find**, **FindAll**, **FindIndex**, **FindLast**, **FindLastIndex**, **ForEach**, **IndexOf**, **LastIndexOf**, **Resize**, **Sort** 和 **TrueForAll** 等。下面展示了部分方法：

```
public abstract class Array : ICloneable, IList, ICollection, IEnumerable,
    IStructuralComparable, IStructuralEquatable {
```

```

public static void Sort<T>(T[] array);
public static void Sort<T>(T[] array, IComparer<T> comparer);

public static Int32 BinarySearch<T>(T[] array, T value);
public static Int32 BinarySearch<T>(T[] array, T value,
    IComparer<T> comparer);
...
}

```

以下代码展示了如何使用其中一些方法：

```

public static void Main() {
    // 创建并初始化字节数组
    Byte[] byteArray = new Byte[] { 5, 1, 4, 2, 3 };

    // 调用 Byte[] 排序算法
    Array.Sort<Byte>(byteArray);

    // 调用 Byte[] 二分搜索算法
    Int32 i = Array.BinarySearch<Byte>(byteArray, 1);
    Console.WriteLine(i); // 显示"0"
}

```

12.2 泛型基础结构

泛型在 CLR 2.0 中加入。为了在 CLR 中加入泛型，许多人花费了大量时间来完成这个大型任务。具体地说，为了使泛型能够工作，Microsoft 必须完成以下工作。

- 创建新的 IL 指令，使之能够识别类型实参。
- 修改现有元数据表的格式，以便表示具有泛型参数的类型名称和方法。
- 修改各种编程语言(C#, Microsoft Visual Basic .NET 等)来支持新语法，允许开发人员定义和引用泛型类型和方法。
- 修改编译器，使之能生成新的 IL 指令和修改的元数据格式。
- 修改 JIT 编译器，以便处理新的支持类型实参的 IL 指令来生成正确的本机代码。
- 创建新的反射成员，使开发人员能查询类型和成员，以判断它们是否具有泛型参数。另外，还必须定义新的反射成员，使开发人员能在运行时创建泛型类型和方法定义。
- 修改调试器以显示和操纵泛型类型、成员、字段以及局部变量。
- 修改 Microsoft Visual Studio 的“智能感知”(IntelliSense)功能。将泛型类型或方法应用于特定数据类型时能显示成员的原型。

现在让我们花些时间讨论 CLR 内部如何处理泛型。这一部分的知识可能影响你构建和设计泛型算法的方式。另外，还可能影响你是否使用一个现有泛型算法的决策。

12.2.1 开放类型和封闭类型

贯穿全书，我讨论了 CLR 如何为应用程序使用的各种类型创建称为**类型对象**(type object)的内部数据结构。具有泛型类型参数的类型仍然是类型，CLR 同样会为它创建内部的对象。这一点适合引用类型(类)、值类型(结构)、接口类型和委托类型。然而，具有泛型类型参数的类型称为**开放类型**，CLR 禁止构造开放类型的任何实例。这类似于 CLR 禁止构造接口类型的实例。

代码引用泛型类型时可指定一组泛型类型实参。为所有类型参数都传递了实际的数据类型，类型就成为**封闭类型**。CLR 允许构造封闭类型的实例。然而，代码引用泛型类型的时候，可能留下一些泛型类型实参未指定。这会在 CLR 中创建新的开放类型对象，而且不能创建该类型的实例。以下代码更清楚地说明了这一点：

```
using System;
using System.Collections.Generic;

// 一个部分指定的开放类型
internal sealed class DictionaryStringKey<TValue> :
    Dictionary<String, TValue> {
}

public static class Program {
    public static void Main() {
        Object o = null;

        // Dictionary<,>是开放类型，有 2 个类型参数
        Type t = typeof(Dictionary<,>);

        // 尝试创建该类型的实例(失败)
        o = CreateInstance(t);
        Console.WriteLine();

        // DictionaryStringKey<>是开放类型，有 1 个类型参数
        t = typeof(DictionaryStringKey<>);

        // 尝试创建该类型的实例(失败)
        o = CreateInstance(t);
        Console.WriteLine();

        // DictionaryStringKey<Guid>是封闭类型
        t = typeof(DictionaryStringKey<Guid>);

        // 尝试创建该类型的一个实例(成功)
        o = CreateInstance(t);

        // 证明它确实能够工作
        Console.WriteLine(" 对象类型 = " + o.GetType());
    }

    private static Object CreateInstance(Type t) {
        Object o = null;
        try {
            o = Activator.CreateInstance(t);
            Console.Write("已创建 {0} 的实例。", t.ToString());
        }
        catch (ArgumentException e) {
            Console.WriteLine(e.Message);
        }
    }
}
```

```

    }
    return o;
}
}

```

编译并运行上述代码得到以下输出:

无法创建 `System.Collections.Generic.Dictionary'2[TKey,TValue]` 的实例,
因为 `Type.ContainsGenericParameters` 为 `True`

无法创建 `DictionaryStringKey'1[TValue]` 的实例,
因为 `Type.ContainsGenericParameters` 为 `True`

已创建 `DictionaryStringKey'1[System.Guid]` 的实例。
对象类型 = `DictionaryStringKey'1[System.Guid]`

可以看出, **Activator** 的 **CreateInstance** 方法会在试图构造开放类型的实例时抛出 **ArgumentException** 异常。注意, 异常的字符串消息指明类型中仍然含有一些泛型参数。

从输出可以看出, 类型名以 “'” 字符和一个数字结尾。数字代表类型的元数, 也就是类型要求的类型参数个数。例如, **Dictionary** 类的元数为 2, 要求为 **TKey** 和 **TValue** 这两个类型参数指定具体类型。**DictionaryStringKey** 类的元数为 1, 只要求为 **TValue** 指定具体类型。

还要注意, CLR 会在类型对象内部分配类型的静态字段(本书第 4 章“类型基础”对此进行了讨论)。因此, 每个封闭类型都有自己的静态字段。换言之, 假如 **List<T>** 定义了任何静态字段, 这些字段不会在一个 **List<DateTime>** 和一个 **List<String>** 之间共享; 每个封闭类型对象都有自己的静态字段。另外, 假如泛型类型定义了静态构造器(参见第 8 章“方法”), 那么针对每个封闭类型, 这个构造器都会执行一次。泛型类型定义静态构造器的目的是保证传递的类型实参满足特定条件。例如, 我们可以像下面这样定义只能处理枚举类型的泛型类型:

```

internal sealed class GenericTypeThatRequiresAnEnum<T> {
    static GenericTypeThatRequiresAnEnum() {
        if (!typeof(T).IsEnum) {
            throw new ArgumentException("T must be an enumerated type");
        }
    }
}

```

CLR 提供了名为约束的功能, 可以更好地指定有效的类型实参。本章稍后会详细讨论。遗憾的是, 约束无法将类型实参限制为“仅枚举类型”。正是因为这个原因, 所以上例需要用静态构造器来保证类型是一个枚举类型。

12.2.2 泛型类型和继承

泛型类型仍然是类型, 所以能从其他任何类型派生。使用泛型类型并指定类型实参时, 实际是在 CLR 中定义一个新的类型对象, 新的类型对象从泛型类型派生自的那个类型派生。换言之, 由于 **List<T>** 从 **Object** 派生, 所以 **List<String>** 和 **List<Guid>** 也从 **Object** 派生。类似地, 由于 **DictionaryStringKey<TValue>** 从 **Dictionary<String, TValue>** 派生, 所以 **DictionaryStringKey<Guid>** 也从 **Dictionary<String, Guid>** 派生。指定类型实参不影响继承层次结构——理解这一点, 有助于你判断哪些强制类型转换是允许的, 哪些不允许。

假定像下面这样定义一个链表节点类:

```
internal sealed class Node<T> {
    public T m_data;
    public Node<T> m_next;

    public Node(T data) : this(data, null) {
    }

    public Node(T data, Node<T> next) {
        m_data = data; m_next = next;
    }

    public override String ToString() {
        return m_data.ToString() +
            ((m_next != null) ? m_next.ToString() : String.Empty);
    }
}
```

那么可以写代码来构造链表, 例如:

```
private static void SameDataLinkedList() {
    Node<Char> head = new Node<Char>('C');
    head = new Node<Char>('B', head);
    head = new Node<Char>('A', head);
    Console.WriteLine(head.ToString()); // 显示"ABC"
}
```

在这个 **Node** 类中, 对于 **m_next** 字段引用的另一个节点来说, 它的 **m_data** 字段必须包含相同的数据类型。这意味着在链表包含的节点中, 所有数据项都必须具有相同的类型(或派生类型)。例如, 不能使用 **Node** 类来创建这样一个链表: 其中一个元素包含 **Char** 值, 另一个包含 **DateTime** 值, 另一个元素则包含 **String** 值。当然, 如果到处都用 **Node<Object>**, 那么确实可以做到, 但会丧失编译时类型安全性, 而且值类型会被装箱。

所以, 更好的办法是定义非泛型 **Node** 基类, 再定义泛型 **TypedNode** 类(用 **Node** 类作为基类)。这样就可以创建一个链表, 其中每个节点都可以是一种具体的数据类型(不能是 **Object**), 同时获得编译时的类型安全性, 并防止值类型装箱。下面是新的类型定义:

```
internal class Node {
    protected Node m_next;

    public Node(Node next) {
        m_next = next;
    }
}

internal sealed class TypedNode<T> : Node {
    public T m_data;

    public TypedNode(T data) : this(data, null) {
    }

    public TypedNode(T data, Node next) : base(next) {
        m_data = data;
    }

    public override String ToString() {
        return m_data.ToString() +
            ((m_next != null) ? m_next.ToString() : String.Empty);
    }
}
```

现在可以写代码来创建一个链表，其中每个节点都是不同的数据类型。例如：

```
private static void DifferentDataLinkedList() {
    Node head = new TypedNode<Char>('.',');
    head = new TypedNode<DateTime>(DateTime.Now, head);
    head = new TypedNode<String>("Today is ", head);
    Console.WriteLine(head.ToString());
}
```

12.2.3 泛型类型同一性

泛型语法有时会将开发人员弄糊涂，因为源代码中可能散布着大量“<”和“>”符号，这有损可读性。为了对语法进行增强，有的开发人员定义了一个新的非泛型类类型，它从一个泛型类型派生，并指定了所有类型实参。例如，为了简化下面这样的代码：

```
List<DateTime> dtl = new List<DateTime>();
```

一些开发人员可能首先定义下面这样的类：

```
internal sealed class DateTimeList : List<DateTime> {
    // 这里无需放入任何代码！
}
```

然后就可以简化创建列表的代码(没有了“<”和“>”符号)：

```
DateTimeList dtl = new DateTimeList();
```

这样做表面上是方便了(尤其是要为参数、局部变量和字段使用新类型的时候)，但是，绝对不要单纯出于增强源码可读性的目的来定义一个新类。这样会丧失类型同一性(identity)和相等性(equivalence)，如以下代码所示：

```
Boolean sameType = (typeof(List<DateTime>) == typeof(DateTimeList));
```

上述代码运行时，**sameType** 会被初始化为 **false**，因为比较的是两个不同类型的对象。这也意味着如果方法的原型接受一个 **DateTimeList**，那么不可以将一个 **List<DateTime>** 传给它。然而，如果方法的原型接受一个 **List<DateTime>**，那么可以将一个 **DateTimeList** 传给它，因为 **DateTimeList** 从 **List<DateTime>** 派生。开发人员很容易被所有这一切搞糊涂。

幸好，C# 允许使用简化的语法来引用泛型封闭类型，同时不会影响类型的相等性。这个语法要求在源文件顶部使用传统的 **using** 指令，例如：

```
using DateTimeList = System.Collections.Generic.List<System.DateTime>;
```

using 指令实际定义的是名为 **DateTimeList** 的符号。代码编译时，编译器将代码中出现的**所有** **DateTimeList** 替换成 **System.Collections.Generic.List<System.DateTime>**。这样就允许开发人员使用简化的语法，同时不影响代码的实际含义。所以，类型的同一性和相等性得到了维持。现在，执行下面这行代码时，**sameType** 会被初始化为 **true**：

```
Boolean sameType = (typeof(List<DateTime>) == typeof(DateTimeList));
```

另外，可以利用 C# 的“隐式类型局部变量”功能，让编译器根据表达式的类型来推断方法的局部变量的类型：

```
using System;
```

```

using System.Collections.Generic;
...
internal sealed class SomeType {
    private static void SomeMethod () {

        // 编译器推断出 dt1 的类型
        // 是 System.Collections.Generic.List<System.DateTime>
        var dt1 = new List<DateTime>();
        ...
    }
}

```

12.2.4 代码爆炸

使用泛型类型参数的方法在进行 JIT 编译时, CLR 获取方法的 IL, 用指定的类型实参替换, 然后创建恰当的本机代码(这些代码为操作指定数据类型“量身定制”)。这正是你希望的, 也是泛型的重要特点。但这样做有一个缺点: CLR 要为每种不同的方法/类型组合生成本机代码。我们将这个现象称为**代码爆炸**。它可能造成应用程序的工作集显著增大, 从而损害性能。

幸好, CLR 内建了一些优化措施能缓解代码爆炸。首先, 假如为特定的类型实参调用了一个方法, 以后再用相同的类型实参调用这个方法, CLR 只会为这个方法/类型组合编译一次代码。所以, 如果一个程序集使用 **List<DateTime>**, 一个完全不同的程序集(加载到同一个 AppDomain 中)也使用 **List<DateTime>**, CLR 只为 **List<DateTime>** 编译一次方法。这样就显著缓解了代码爆炸。

CLR 还有另一个优化, 它认为所有引用类型实参都完全相同, 所以代码能够共享。例如, CLR 为 **List<String>** 的方法编译的代码可直接用于 **List<Stream>** 的方法, 因为 **String** 和 **Stream** 均为引用类型。事实上, 对于任何引用类型, 都会使用相同的代码。CLR 之所以能执行这个优化, 是因为所有引用类型的实参或变量实际只是指向堆上对象的指针(32 位 Windows 系统是 32 位指针; 64 位 Windows 系统是 64 位指针), 而所有对象指针都以相同方式操纵。

但是, 假如某个类型实参是值类型, CLR 就必须专门为那个值类型生成本机代码。这是因为值类型的大小不定。即使两个值类型大小一样(比如 **Int32** 和 **UInt32**, 两者都是 32 位), CLR 仍然无法共享代码, 因为可能要用不同的本机 CPU 指令来操纵这些值。

12.3 泛型接口

显然, 泛型的主要作用就是定义泛型的引用类型和值类型。然而, 对泛型接口的支持对 CLR 来说也很重要。没有泛型接口, 每次用非泛型接口(如 **IComparable**)来操纵值类型都会发生装箱, 而且会失去编译时的类型安全性。这将严重制约泛型类型的应用范围。因此, CLR 提供了对泛型接口的支持。引用类型或值类型可指定类型实参实现泛型接口。也可保持类型实参的未指定状态来实现泛型接口。下面来看一些例子。

以下泛型接口定义是 FCL 的一部分(在 `System.Collections.Generic` 命名空间中):

```
public interface IEnumerator<T> : IDisposable, IEnumerator {
    T Current { get; }
}
```

下面的示例类型实现了上述泛型接口, 而且指定了类型实参。注意, **Triangle** 对象可枚举一组 **Point** 对象。还要注意, **Current** 属性具有 **Point** 数据类型:

```
internal sealed class Triangle : IEnumerator<Point> {
    private Point[] m_vertices;

    // IEnumerator<Point>的 Current 属性是 Point 类型
    public Point Current { get { ... } }

    ...
}
```

下例实现了相同的泛型接口, 但保持类型实参的未指定状态:

```
internal sealed class ArrayEnumerator<T> : IEnumerator<T> {
    private T[] m_array;

    // IEnumerator<T>的 Current 属性是 T 类型
    public T Current { get { ... } }

    ...
}
```

注意, **ArrayEnumerator** 对象可枚举一组 **T** 对象(**T** 未指定, 允许代码使用泛型 **ArrayEnumerator** 类型在以后为 **T** 指定具体类型)。还要注意, **Current** 属性现在具有未指定的数据类型 **T**。第 13 章“接口”将更深入地讨论泛型接口。

12.4 泛型委托

CLR 支持泛型委托, 目的是保证任何类型的对象都能以类型安全的方式传给回调方法。此外, 泛型委托允许值类型实例在传给回调方法时不进行任何装箱。第 17 章“委托”会讲到, 委托实际只是提供了 4 个方法的一个类定义。4 个方法包括一个构造器、一个 **Invoke** 方法, 一个 **BeginInvoke** 方法和一个 **EndInvoke** 方法。如果定义的委托类型指定了类型参数, 编译器会定义委托类的方法, 用指定的类型参数替换方法的参数类型和返回值类型。

例如, 假定像下面这样定义泛型委托:

```
public delegate TReturn CallMe<TReturn, TKey, TValue>(TKey key, TValue value);
```

编译器会将它转换成如下所示的类:

```
public sealed class CallMe<TReturn, TKey, TValue> : MulticastDelegate {
    public CallMe(Object object, IntPtr method);
    public virtual TReturn Invoke(TKey key, TValue value);
    public virtual IAsyncResult BeginInvoke(TKey key, TValue value,
        AsyncCallback callback, Object object);
    public virtual TReturn EndInvoke(IAsyncResult result);
}
```



注意 建议尽量使用 FCL 预定义的泛型 **Action** 和 **Func** 委托。这些委托的详情将在 17.6 节“委托定义不要太多(泛型委托)”讲述。

12.5 委托和接口的逆变和协变泛型类型实参

委托的每个泛型类型参数都可标记为协变量或逆变量^①。利用这个功能,可将泛型委托类型的变量转换为相同的委托类型(但泛型参数类型不同)。泛型类型参数可以是以下任何一种形式。

- **不变量(invariant)** 意味着泛型类型参数不能更改。到目前为止,你在本章看到的全是不变量形式的泛型类型参数。
- **逆变量(contravariant)** 意味着泛型类型参数可以从一个类更改为它的某个派生类。在 C# 是用 **in** 关键字标记逆变量形式的泛型类型参数。逆变量泛型类型参数只出现在输入位置,比如作为方法的参数。
- **协变量(covariant)** 意味着泛型类型参数可以从一个类更改为它的某个基类。C# 是用 **out** 关键字标记协变量形式的泛型类型参数。协变量泛型类型参数只能出现在输出位置,比如作为方法的返回类型。

例如,假定存在以下委托类型定义(顺便说一下,它真的存在):

```
public delegate TResult Func<in T, out TResult>(T arg);
```

其中,泛型类型参数 **T** 用 **in** 关键字标记,这使它成为逆变量;泛型类型参数 **TResult** 用 **out** 关键字标记,这使它成为协变量。

所以,如果像下面这样声明一个变量:

```
Func<Object, ArgumentException> fn1 = null;
```

就可将它转型为另一个泛型类型参数不同的 **Func** 类型:

```
Func<String, Exception> fn2 = fn1; // 不需要显式转型
Exception e = fn2("");
```

上述代码的意思是说: **fn1** 变量引用一个方法,获取一个 **Object**,返回一个 **ArgumentException**。而 **fn2** 变量引用另一个方法,获取一个 **String**,返回一个 **Exception**。由于可将一个 **String** 传给期待 **Object** 的方法(因为 **String** 从 **Object** 派生),而且由于可以

① 本书采用目前约定俗成的译法: covariant=协变量, contravariant=逆变量; covariance=协变性, contravariance=逆变性。另外, variance=可变性。至于两者更详细的区别,推荐阅读 Eric Lippert 的 Covariance and Contravariance in C# 系列博客文章。简而言之,协变性指定返回类型的兼容性,而逆变性指定参数的兼容性。——译注

获取返回 **ArgumentException** 的一个方法的结果，并将这个结果当成一个 **Exception** (因为 **Exception** 是 **ArgumentException** 的基类)，所以上述代码能正常编译，而且编译时能维持类型安全性。



注意 只有编译器能验证类型之间存在引用转换，这些可变性才有用。换言之，由于需要装箱，所以值类型不具有这种可变性。我个人认为正是因为存在这个限制，这些可变性的用处不是特别大。例如，假定定义以下方法：

```
void ProcessCollection(IEnumerable<Object> collection) { ... }
```

我不能在调用它时传递一个 **List<DateTime>** 对象引用，因为 **DateTime** 值类型和 **Object** 之间不存在引用转换——虽然 **DateTime** 是从 **Object** 派生的。为了解决这个问题，可像下面这样声明 **ProcessCollection**：

```
void ProcessCollection<T>(IEnumerable<T> collection) { ... }
```

另外，**ProcessCollection(IEnumerable<Object> collection)** 最大的好处是 JIT 编译得到的代码只有一个版本。但如果使用 **ProcessCollection<T>(IEnumerable<T> collection)**，那么只有在 **T** 是引用类型的前提下，才可共享同一个版本的 JIT 编译代码。对于 **T** 是值类型的情况，每个值类型都有一份不同的 JIT 编译代码；不过，起码能在调用方法时传递一个值类型集合了。

另外，对于泛型类型参数，如果要将该类型的实参传给使用 **out** 或 **ref** 关键字的方法，便不允许可变性。例如，以下代码会造成编译器报告错误消息：**无效的可变性：类型参数“T”在“SomeDelegate<T>.Invoke(ref T)”中必须是不变量。现在的“T”是逆变量。**^①

```
delegate void SomeDelegate<in T>(ref T t);
```

使用要获取泛型参数和返回值的委托时，建议尽量为逆变性和协变性指定 **in** 和 **out** 关键字。这样做不会有不良反应，并使你的委托能在更多的情形中使用。

和委托相似，具有泛型类型参数的接口也可将类型参数标记为逆变量和协变量。下面的示例接口有一个逆变量泛型类型参数：

```
public interface IEnumerator<in T> : IEnumerator {
    Boolean MoveNext();
    T Current { get; }
}
```

由于 **T** 是逆变量，所以以下代码可顺利编译和运行：

① Visual Studio 2012/2013 实际显示的消息有点让人摸不着头脑：变体无效：类型参数“**T**”必须为对于“**SomeDelegate<T>.Invoke(ref T)**”有效的固定式。“**T**”为逆变。——译注

```
// 这个方法接受任意引用类型的一个 IEnumerable
Int32 Count(IEnumerable<Object> collection) { ... }

...
// 以下调用向 Count 传递一个 IEnumerable<String>
Int32 c = Count(new[] { "Grant" });
```



重要提示 开发人员有时会问：“为什么必须显式用 **in** 或 **out** 标记泛型类型参数？”他们认为编译器应该能检查委托或接口声明，并自动检测哪些泛型类型参数能够逆变和协变。虽然编译器确实能，但 C# 团队认为必须由你订立协定(contract)，明确说明想允许什么。例如，假定编译器判断一个泛型类型参数是逆变量(用在输入位置)，但你将来向某个接口添加了成员，并将类型参数用在了输出位置。下次编译时，编译器将认为该类型参数是不变量。但在引用了其他成员的所有地方，只要还以为“类型参数是逆变量”就可能出错。

因此，编译器团队决定，在声明泛型类型参数时，必须由你显式使用 **in** 或 **out** 来标记可变性。以后使用这个类型参数时，假如用法与声明时指定的不符，编译器就会报错，提醒你违反了自己订立的协定。如果为泛型类型参数添加 **in** 或 **out** 来打破原来的协定，就必须修改使用旧协定的代码。

12.6 泛型方法

定义泛型类、结构或接口时，类型中定义的任何方法都可引用类型指定的类型参数。类型参数可作为方法参数、方法返回值或方法内部定义的局部变量的类型使用。然而，CLR 还允许方法指定它自己的类型参数。这些类型参数也可作为参数、返回值或局部变量的类型使用。

在下例中，类型定义了一个类型参数，方法也定义了自己的：

```
internal sealed class GenericType<T> {
    private T m_value;

    public GenericType(T value) { m_value = value; }

    public TOutput Converter<TOutput>() {
        TOutput result = (TOutput) Convert.ChangeType(m_value, typeof(TOutput));
        return result;    // 返回类型转换之后的结果
    }
}
```

在这个例子中，**GenericType** 类定义了类型参数(T)，**Converter** 方法也定义了自己的类型参数(TOutput)。这样的 **GenericType** 可以处理任意类型。**Converter** 方法能将 **m_value** 字段引用的对象转换成任意类型——具体取决于调用时传递的类型实参是什么。泛型方法的存在，为开发人员提供了极大的灵活性。

泛型方法的一个很好的例子是 **Swap** 方法：

```
private static void Swap<T>(ref T o1, ref T o2) {
    T temp = o1;
    o1 = o2;
    o2 = temp;
}
```

现在可以这样调用 **Swap**：

```
private static void CallingSwap() {
    Int32 n1 = 1, n2 = 2;
    Console.WriteLine("n1={0}, n2={1}", n1, n2);
    Swap<Int32>(ref n1, ref n2);
    Console.WriteLine("n1={0}, n2={1}", n1, n2);

    String s1 = "Aidan", s2 = "Grant";
    Console.WriteLine("s1={0}, s2={1}", s1, s2);
    Swap<String>(ref s1, ref s2);
    Console.WriteLine("s1={0}, s2={1}", s1, s2);
}
```

为获取 **out** 和 **ref** 参数的方法使用泛型类型很有意思，因为作为 **out/ref** 实参传递的变量必须具有与方法参数相同的类型，以防止损害类型安全性。涉及 **out/ref** 参数的这个问题已在 9.3 节“以传引用的方式向方法传递参数”讨论。事实上，**Interlocked** 类的 **Exchange** 和 **CompareExchange** 方法就是因为这个原因才提供泛型重载的^①：

```
public static class Interlocked {
    public static T Exchange<T>(ref T location1, T value) where T: class;
    public static T CompareExchange<T>(
        ref T location1, T value, T comparand) where T: class;
}
```

泛型方法和类型推断

C# 泛型语法因为涉及大量“<”和“>”符号，所以开发人员很容易被弄得晕头转向。为了改进代码的创建，增强可读性和可维护性，C# 编译器支持在调用泛型方法时进行**类型推断**。这意味着编译器会在调用泛型方法时自动判断(或者说推断)要使用的类型。以下代码对类型推断进行了演示：

```
private static void CallingSwapUsingInference() {
    Int32 n1 = 1, n2 = 2;
    Swap(ref n1, ref n2);    // 调用 Swap<Int32>

    String s1 = "Aidan";
    Object s2 = "Grant";
    Swap(ref s1, ref s2);    // 错误，不能推断类型
}
```

在上述代码中，对 **Swap** 的调用没有在一对“<”和“>”中指定类型实参。在第一个 **Swap** 调用中，C# 编译器推断 **n1** 和 **n2** 都是 **Int32**，所以应该使用 **Int32** 类型实参来调用 **Swap**。

推断类型时，C# 使用变量的数据类型，而不是变量引用的对象的实际类型。所以在第二个

① **where** 子句将在本章稍后的 12.8 节“可验证性和约束”讨论。

Swap 调用中, C#发现 **s1** 是 **String**, 而 **s2** 是 **Object**(即使它恰好引用一个 **String**)。由于 **s1** 和 **s2** 是不同数据类型的变量, 编译器拿不准要为 **Swap** 传递什么类型实参, 所以会报告以下消息: **error CS0411: 无法从用法推断出方法"Program.Swap<T>(ref T, ref T)"的类型参数。请尝试显式指定类型参数。**

类型可定义多个方法, 让其中一个方法接受具体数据类型, 让另一个接受泛型类型参数, 如下例所示:

```
private static void Display(String s) {
    Console.WriteLine(s);
}

private static void Display<T>(T o) {
    Display(o.ToString()); // 调用 Display(String)
}
```

下面展示了 **Display** 方法的一些调用方式:

```
Display("Jeff"); // 调用 Display(String)
Display(123); // 调用 Display<T>(T)
Display<String>("Aidan"); // 调用 Display<T>(T)
```

在第一个调用中, 编译器可调用接受一个 **String** 参数的 **Display** 方法, 也可调用泛型 **Display** 方法(将 **T** 替换成 **String**)。但 C#编译器的策略是先考虑较明确的匹配, 再考虑泛型匹配。所以, 它会生成对非泛型 **Display** 方法的调用, 也就是接受一个 **String** 参数的版本。对于第二个调用, 编译器不能调用接受 **String** 参数的非泛型 **Display** 方法, 所以必须调用泛型 **Display** 方法。顺便说一句, 编译器优先选择较明确的匹配, 开发人员应对此感到庆幸。假如编译器优先选择泛型方法, 那么由于泛型 **Display** 方法会再次调用 **Display**(但传递由 **ToString** 返回的一个 **String**), 所以会造成无限递归。

对 **Display** 的第三个调用明确指定了泛型类型实参 **String**。这告诉编译器不要尝试推断类型实参。相反, 应使用显式指定的类型实参。在本例中, 编译器还假定我肯定是想调用泛型 **Display** 方法, 所以会毫不犹豫地调用泛型 **Display** 方法。在内部, 泛型 **Display** 方法会为传入的字符串调用 **ToString** 方法, 然后将转换所得的字符串传给非泛型 **Display** 方法。

12.7 泛型和其他成员

在 C#中, 属性、索引器、事件、操作符方法、构造器和终结器本身不能有类型参数。但它们能在泛型类型中定义, 而且这些成员中的代码能使用类型的类型参数。

C#之所以不允许这些成员指定自己的泛型类型参数, 是因为 Microsoft C#团队认为开发人员很少需要将这些成员作为泛型使用。除此之外, 为这些成员添加泛型支持的代价是相当高的, 因为必须为语言设计足够的语法。例如, 在代码中使用一个+操作符时, 编译器可能要调用一个操作符重载方法。而在代码中, 没有任何办法能伴随+操作符指定类型实参。

12.8 可验证性和约束

编译泛型代码时，C#编译器会进行分析，确保代码适用于当前已有或将来可能定义的任何类型。看看下面这个方法：

```
private static Boolean MethodTakingAnyType<T>(T o) {
    T temp = o;
    Console.WriteLine(o.ToString());
    Boolean b = temp.Equals(o);
    return b;
}
```

这个方法声明了 **T** 类型的临时变量(**temp**)。然后，方法执行两次变量赋值和几次方法调用。这个方法适用于任何类型。无论 **T** 是引用类型，是值类型或枚举类型，还是接口或委托类型，它都能工作。这个方法适用于当前存在的所有类型，也适用于将来可能定义的任何类型，因为所有类型都支持对 **Object** 类型的变量的赋值，也支持对 **Object** 类型定义的方法的调用(比如 **ToString** 和 **Equals**)。

再来看看下面这个方法：

```
private static T Min<T>(T o1, T o2) {
    if (o1.CompareTo(o2) < 0) return o1;
    return o2;
}
```

Min 方法试图使用 **o1** 变量来调用 **CompareTo** 方法。但是，许多类型都没有提供 **CompareTo** 方法，所以 C#编译器不能编译上述代码，它不能保证这个方法适用于所有类型。强行编译上述代码会报告消息：

error CS0117: “T” 不包含 “CompareTo” 的定义，并且找不到可接受类型为 “T” 的第一个参数的扩展方法 “CompareTo” (是否缺少 using 指令或程序集引用?)。

所以从表面看，使用泛型似乎做不了太多事情。只能声明泛型类型的变量，执行变量赋值，再调用 **Object** 定义的方法，如此而已！显然，假如泛型只能这么用，可以说它几乎没有任何用。幸好，编译器和 CLR 支持称为**约束**的机制，可通过它使泛型变得真正有用！

约束的作用是限制能指定成泛型实参的类型数量。通过限制类型的数量，可以对那些类型执行更多操作。以下是新版本的 **Min** 方法，它指定了一个约束(加粗显示)：

```
public static T Min<T>(T o1, T o2) where T : IComparable<T> {
    if (o1.CompareTo(o2) < 0) return o1;
    return o2;
}
```

C#的 **where** 关键字告诉编译器，为 **T** 指定的任何类型都必须实现同类型(**T**)的泛型 **IComparable** 接口。有了这个约束，就可以在方法中调用 **CompareTo**，因为已知 **IComparable<T>** 接口定义了 **CompareTo**。

现在，当代码引用泛型类型或方法时，编译器要负责保证类型实参符合指定的约束。例如，假如编译以下代码：

```
private static void CallMin() {
    Object o1 = "Jeff", o2 = "Richter";
    Object oMin = Min<Object>(o1, o2); // Error CS0311
}
```

编译器会报告以下消息：

error CS0311: 不能将类型"object"用作泛型类型或方法"System.Min<T>(T,T)"中的类型参数"T"。没有从"object"到"System.IComparable<object>"的隐式引用转换。

编译器报错是因为 **System.Object** 没有实现 **IComparable<Object>** 接口。事实上，**System.Object** 没有实现任何接口。

对约束及其工作方式有了一个基本的认识后，让我们更深入地研究一下它。约束可应用于泛型类型的类型参数，也可应用于泛型方法的类型参数(如 **Min** 方法所示)。CLR 不允许基于类型参数名称或约束来进行重载；只能基于元数(类型参数个数)对类型或方法进行重载。下例对此进行了演示：

```
// 可定义以下类型
internal sealed class AType {}
internal sealed class AType<T> {}
internal sealed class AType<T1, T2> {}

// 错误：与没有约束的 AType<T> 冲突
internal sealed class AType<T> where T : IComparable<T> {}

// 错误：与 AType<T1, T2> 冲突
internal sealed class AType<T3, T4> {}

internal sealed class AnotherType {
    // 可定义以下方法，参数个数不同：
    private static void M() {}
    private static void M<T>() {}
    private static void M<T1, T2>() {}

    // 错误：与没有约束的 M<T> 冲突
    private static void M<T>() where T : IComparable<T> {}

    // 错误：与 M<T1, T2> 冲突
    private static void M<T3, T4>() {}
}
```

重写虚泛型方法时，重写的方法必须指定相同数量的类型参数，而且这些类型参数会继承在基类方法上指定的约束。事实上，根本不允许为重写方法的类型参数指定任何约束。但类型参数的名称是可以改变的。类似地，实现接口方法时，方法必须指定与接口方法等量的类型参数，这些类型参数将继承由接口方法指定的约束。下例使用虚方法演示了这一规则：

```
internal class Base {
    public virtual void M<T1, T2>()
        where T1 : struct
        where T2 : class {
    }
}

internal sealed class Derived : Base {
    public override void M<T3, T4>()
        where T3 : EventArgs // 错误
        where T4 : class // 错误
}
```

```
    { }
}
```

试图编译上述代码，编译器会报告以下错误：

error CS0460: 重写和显式接口实现方法的约束是从基方法继承的，因此不能直接指定这些约束。

从 **Derived** 类的 **M<T3, T4>** 方法中移除两个 **where** 子句，代码就能正常编译了。注意，类型参数的名称可以更改，比如将 **T1** 改成 **T3**，将 **T2** 改成 **T4**；但约束不能更改(甚至不能指定)。

下面讨论编译器/CLR 允许向类型参数应用的各种约束。可用一个主要约束、一个次要约束以及/或者一个构造器约束来约束类型参数。接下来的三个小节分别讨论这些约束。

12.8.1 主要约束

类型参数可以指定零个或者一个**主要约束**。主要约束可以是代表非密封类的一个引用类型。不能指定以下特殊引用类型：**System.Object**，**System.Array**，**System.Delegate**，**System.MulticastDelegate**，**System.ValueType**，**System.Enum** 或者 **System.Void**。

指定引用类型约束时，相当于向编译器承诺：一个指定的类型实参要么是约束类型相同的类型，要么是从约束类型派生的类型。例如以下泛型类：

```
internal sealed class PrimaryConstraintOfStream<T> where T : Stream {
    public void M(T stream) {
        stream.Close();    // 正确
    }
}
```

在这个类定义中，类型参数 **T** 设置了主要约束 **Stream**(在 **System.IO** 命名空间中定义)。这就告诉编译器，使用 **PrimaryConstraintOfStream** 的代码在指定类型实参时，必须指定 **Stream** 或者从 **Stream** 派生的类型(比如 **FileStream**)。如果类型参数没有指定主要约束，就默认为 **System.Object**。但如果在源代码中显式指定 **System.Object**，C# 会报错：error CS0702: 约束不能是特殊类"object"。

有两个特殊的主要约束：**class** 和 **struct**。其中，**class** 约束向编译器承诺类型实参是引用类型。任何类类型、接口类型、委托类型或者数组类型都满足这个约束。例如以下泛型类：

```
internal sealed class PrimaryConstraintOfClass<T> where T : class {
    public void M() {
        T temp = null;    // 允许，因为 T 肯定是引用类型
    }
}
```

在这个例子中，将 **temp** 设为 **null** 是合法的，因为 **T** 已知是引用类型，而所有引用类型的变量都能设为 **null**。不对 **T** 进行约束，上述代码就通不过编译，因为 **T** 可能是值类型，而值类型的变量不能设为 **null**。

struct 约束向编译器承诺类型实参是值类型。包括枚举在内的任何值类型都满足这个约束。但编译器和 CLR 将任何 **System.Nullable<T>** 值类型视为特殊类型，不满足这个 **struct** 约束。

原因是 `Nullable<T>` 类型将它的类型参数约束为 `struct`，而 CLR 希望禁止像 `Nullable<Nullable<T>>` 这样的递归类型。可空类型将在第 19 章“可空值类型”讨论。

以下示例类使用 `struct` 约束来约束它的类型参数：

```
internal sealed class PrimaryConstraintOfStruct<T> where T : struct {
    public static T Factory() {
        // 允许。因为所有值类型都隐式有一个公共无参构造器
        return new T();
    }
}
```

这个例子中的 `new T()` 是合法的，因为 `T` 已知是值类型，而所有值类型都隐式地有一个公共无参构造器。如果 `T` 不约束，约束为引用类型，或者约束为 `class`，上述代码将无法通过编译，因为有的引用类型没有公共无参构造器。

12.8.2 次要约束

类型参数可以指定零个或者多个次要约束，次要约束代表接口类型。这种约束向编译器承诺类型实参实现了接口。由于能指定多个接口约束，所以类型实参必须实现了所有接口约束(以及主要约束，如果有的话)。第 13 章“接口”将详细讨论接口约束。

还有一种次要约束称为类型参数约束，有时也称为裸类型约束。这种约束用得比接口约束少得多。它允许一个泛型类型或方法规定：指定的类型实参要么是约束的类型，要么是约束的类型的派生类。一个类型参数可以指定零个或者多个类型参数约束。下面这个泛型方法演示了如何使用类型参数约束：

```
private static List<TBase> ConvertIList<T, TBase>(IList<T> list)
    where T : TBase {
    List<TBase> baseList = new List<TBase>(list.Count);
    for (Int32 index = 0; index < list.Count; index++) {
        baseList.Add(list[index]);
    }
    return baseList;
}
```

`ConvertIList` 方法指定了两个类型参数，其中 `T` 参数由 `TBase` 类型参数约束。意味着不管为 `T` 指定什么类型实参，都必须兼容于为 `TBase` 指定的类型实参。下面这个方法演示了对 `ConvertIList` 的合法调用和非法调用：

```
private static void CallingConvertIList() {
    // 构造并初始化一个 List<String> (它实现了 IList<String>)
    IList<String> ls = new List<String>();
    ls.Add("A String");

    // 1. 将 IList<String> 转换成一个 IList<Object>
    IList<Object> lo = ConvertIList<String, Object>(ls);

    // 2. 将 IList<String> 转换成一个 IList<IComparable>
    IList<IComparable> lc = ConvertIList<String, IComparable>(ls);

    // 3. 将 IList<String> 转换成一个 IList<IComparable<String>>
    IList<IComparable<String>> lcs =
        ConvertIList<String, IComparable<String>>(ls);
}
```

```
// 4. 将 IList<String> 转换成一个 IList<String>
IList<String> ls2 = ConvertIList<String, String>(ls);

// 5. 将 IList<String> 转换成一个 IList<Exception>
IList<Exception> le = ConvertIList<String, Exception>(ls); // 错误
}
```

在对 `ConvertIList` 的第一个调用中，编译器检查 `String` 是否兼容于 `Object`。由于 `String` 从 `Object` 派生，所以第一个调用满足类型参数约束。在对 `ConvertIList` 的第二个调用中，编译器检查 `String` 是否兼容于 `IComparable`。由于 `String` 实现了 `IComparable` 接口，所以第二个调用满足类型参数约束。在对 `ConvertIList` 的第三个调用中，编译器检查 `String` 是否兼容于 `IComparable<String>`。由于 `String` 实现了 `IComparable<String>` 接口，所以第三个调用满足类型参数约束。在对 `ConvertIList` 的第 4 个调用中，编译器知道 `String` 兼容于它自己。在对 `ConvertIList` 的第 5 个调用中，编译器检查 `String` 是否兼容于 `Exception`。由于 `String` 不兼容于 `Exception`，所以第 5 个调用不满足类型参数约束，编译器报告以下消息：

```
error CS0311: 不能将类型 "string" 用作泛型类型或方法 "Program.ConvertIList<T, TBase>
(System.Collections.Generic.IList<T>)" 中的类型参数 "T"。没有从 "string" 到 "System.Exception"
的隐式引用转换。
```

12.8.3 构造器约束

类型参数可指定零个或一个构造器约束，它向编译器承诺类型实参是实现了公共无参构造器的非抽象类型。注意，如果同时使用构造器约束和 `struct` 约束，C# 编译器会认为这是一个错误，因为这是多余的；所有值类型都隐式提供了公共无参构造器。以下示例类使用构造器约束来约束它的类型参数：

```
internal sealed class ConstructorConstraint<T> where T : new() {
    public static T Factory() {
        // 允许，因为所有值类型都隐式有一个公共无参构造器。
        // 而如果指定的是引用类型，约束也要求它提供公共无参构造器
        return new T();
    }
}
```

这个例子中的 `new T()` 是合法的，因为已知 `T` 是拥有公共无参构造器的类型。对所有值类型来说，这一点（拥有公共无参构造器）肯定成立。对于作为类型实参指定的任何引用类型，这一点也成立，因为构造器约束要求它必须成立。

开发人员有时想为类型参数指定一个构造器约束，并指定构造器要获取多个参数。目前，CLR（以及 C# 编译器）只支持无参构造器。Microsoft 认为这已经能满足几乎所有情况，我对此也表示同意。

12.8.4 其他可验证性问题

本节剩下的部分将讨论另外几个特殊的代码构造。由于可验证性问题，这些代码构造在和泛型共同使用时，可能产生不可预期的行为。另外，还讨论了如何利用约束使代码重新变得可以验证。

1. 泛型类型变量的转型

将泛型类型的变量转型为其他类型是非法的，除非转型为与约束兼容的类型：

```
private static void CastingAGenericTypeVariable1<T>(T obj) {
    Int32 x = (Int32) obj;    //错误
    String s = (String) obj;  //错误
}
```

上述两行会造成编译器报错，因为 **T** 可能是任意类型，无法保证成功转型。为了修改上述代码使其能通过编译，可以先转型为 **Object**：

```
private static void CastingAGenericTypeVariable2<T>(T obj) {
    Int32 x = (Int32) (Object) obj;    //无错误
    String s = (String) (Object) obj;  //无错误
}
```

虽然代码现在能编译，但 CLR 仍有可能在运行时抛出 **InvalidCastException** 异常。

转型为引用类型时还可使用 C# **as** 操作符。下面对代码进行了修改，为 **String** 使用了 **as** 操作符 (**Int32** 是值类型不能用)：

```
private static void CastingAGenericTypeVariable3<T>(T obj) {
    String s = obj as String; // 无错误
}
```

2. 将泛型类型变量设为默认值

将泛型类型变量设为 **null** 是非法的，除非将泛型类型约束成引用类型。

```
private static void SettingAGenericTypeVariableToNull<T>() {
    T temp = null; // error CS0403 - 无法将 null 转换为类型参数"T",
                  // 因为它可能是不可以为 null 的值类型。请考虑改用 default(T)
}
```

由于未对 **T** 进行约束，所以它可能是值类型，而将值类型的变量设为 **null** 是不可能的。如果 **T** 被约束成引用类型，将 **temp** 设为 **null** 就是合法的，代码能顺利编译并运行。

Microsoft 的 C# 团队认为有必要允许开发人员将变量设为它的默认值，并专门为此提供了 **default** 关键字：

```
private static void SettingAGenericTypeVariableToDefaultValue<T>() {
    T temp = default(T); // 正确
}
```

以上代码中的 **default** 关键字告诉 C# 编译器和 CLR 的 JIT 编译器，如果 **T** 是引用类型，就将 **temp** 设为 **null**；如果是值类型，就将 **temp** 的所有位设为 0。

3. 将泛型类型变量与 null 进行比较

无论泛型类型是否被约束，使用 **==** 或 **!=** 操作符将泛型类型变量与 **null** 进行比较都是合法的：

```
private static void ComparingAGenericTypeVariableWithNull<T>(T obj) {
    if (obj == null) { /* 对于值类型，永远都不会执行 */ }
}
```

由于 **T** 未进行约束，所以可能是引用类型或值类型。如果 **T** 是值类型，**obj** 永远都不会为 **null**。你或许以为 C# 编译器会报错。但 C# 编译器并不报错；相反，它能顺利地编译代码。调用这个方法时，如果为类型参数传递值类型，那么 JIT 编译器知道 **if** 语句永远都不会为 **true**，所以不会为 **if** 测试或者大括号内的代码生成本机代码。如果换用 **!=** 操作符，JIT 编译器不会为 **if** 测试生成代码(因为它肯定为 **true**)，但会为 **if** 大括号内的代码生成本机代码。

顺便说一句，如果 **T** 被约束成 **struct**，C# 编译器会报错。值类型的变量不能与 **null** 进行比较，因为结果始终一样。

4. 两个泛型类型变量相互比较

如果泛型类型参数不能肯定是引用类型，对同一个泛型类型的两个变量进行比较是非法的：

```
private static void ComparingTwoGenericTypeVariables<T>(T o1, T o2) {
    if (o1 == o2) { } //错误
}
```

在这个例子中，**T** 未进行约束。虽然两个引用类型的变量相互比较是合法的，但两个值类型的变量相互比较是非法的，除非值类型重载了 **==** 操作符。如果 **T** 被约束成 **class**，上述代码能通过编译。如果变量引用同一个对象，**==** 操作符会返回 **true**。注意，如果 **T** 被约束成引用类型，而且该引用类型重载了 **operator==** 方法，那么编译器会在看到 **==** 操作符时生成对这个方法的调用。显然，所有些讨论也适合 **!=** 操作符。

写代码来比较基元值类型(**Byte**, **Int32**, **Single**, **Decimal** 等)时，C# 编译器知道如何生成正确的代码。然而，对于非基元值类型，C# 编译器不知道如何生成代码来进行比较。所以，如果 **ComparingTwoGenericTypeVariables** 方法的 **T** 被约束成 **struct**，编译器会报错。

不允许将类型参数约束成具体的值类型，因为值类型隐式密封，不可能存在从值类型派生的类型。如果允许将类型参数约束成具体的值类型，那么泛型方法会被约束为只支持该具体类型，这还不如不要泛型呢！

5. 泛型类型变量作为操作数使用

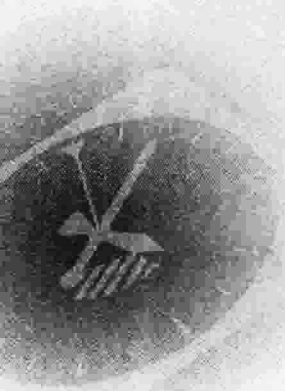
最后要注意，将操作符应用于泛型类型的操作数会出现大量问题。第 5 章讨论了 C# 如何处理它的基元类型：**Byte**, **Int16**, **Int32**, **Int64**, **Decimal** 等。我特别指出 C# 知道如何解释应用于基元类型的操作符(比如 **+**, **-**, ***** 和 **/**)。但不能将这些操作符应用于泛型类型的变量。编译器在编译时确定不了类型，所以不能向泛型类型的变量应用任何操作符。所以，不可能写一个能处理任何数值数据类型的算法。下面是我想写的一个示例泛型方法：

```
private static T Sum<T>(T num) where T : struct {
    T sum = default(T);
    for (T n = default(T); n < num; n++)
        sum += n;
    return sum;
}
```

我千方百计想让这个方法通过编译。我将 **T** 约束成一个 **struct**，而且使用 **default(T)** 将 **sum** 和 **n** 初始化为 **0**。但编译时得到以下错误：

- **error CS0019:** 运算符 "<" 无法应用于 "T" 和 "T" 类型的操作数
- **error CS0023:** 运算符 "++" 无法应用于 "T" 类型的操作数
- **error CS0019:** 运算符 "+=" 无法应用于 "T" 和 "T" 类型的操作数

这是 CLR 的泛型支持体系的一个严重限制,许多开发人员(尤其是科学、金融和数学领域的开发人员)对这个限制感到很失望。许多人尝试用各种技术来避开这一限制,其中包括反射(参见第 23 章“程序集加载和反射”)、**dynamic** 基元类型(5.5 节)和操作符重载等。但所有这些技术都会严重损害性能或者影响代码的可读性。希望 Microsoft 在 CLR 和编译器未来的版本中解决这个问题。



第 13 章 接 口

本章内容:

- 类和接口继承
- 定义接口
- 继承接口
- 关于调用接口方法的更多探讨
- 隐式和显式接口方法实现(幕后发生的事情)
- 泛型接口
- 泛型和接口约束
- 实现多个具有相同方法名和签名的接口
- 用显式接口方法实现来增强编译时类型安全性
- 谨慎使用显式接口方法实现
- 设计: 基类还是接口?

对于多继承(multiple inheritance)的概念,许多程序员并不陌生,它是指一个类从两个或多个基类派生的能力。例如,假定 **TransmitData** 类的作用是发送数据, **ReceiveData** 类的作用是接收数据。现在要创建 **SocketPort** 类,作用是发送和接收数据。在这种情况下,你会希望 **SocketPort** 从 **TransmitData** 和 **ReceiveData** 这两个类继承。

有的编程语言允许多继承,所以能从 **TransmitData** 和 **ReceiveData** 这两个基类派生出 **SocketPort**。但 CLR 不支持多继承(因此所有托管编程语言也支持不了)。CLR 只是通过接口提供了“缩水版”的多继承。本章将讨论如何定义和使用接口,还要提供一些指导性原则,以便你判断何时应该使用接口而不是基类。

13.1 类和接口继承

Microsoft .NET Framework 提供了 **System.Object** 类,它定义了 4 个公共实例方法: **ToString**, **Equals**, **GetHashCode** 和 **GetType**。该类是其他所有类的根或者说终极基类。换言之,所有类都继承了 **Object** 的 4 个实例方法。这还意味着只要代码能操作 **Object** 类的实例,就能操作任何类的实例。

由于 Microsoft 的开发团队已实现了 **Object** 的方法,所以从 **Object** 派生的任何类实际都继

承了以下内容。

- **方法签名**

使代码认为自己是在操作 **Object** 类的实例，但实际操作的可能其他类的实例。

- **方法实现**

使开发人员定义 **Object** 的派生类时不必手动实现 **Object** 的方法。

在 CLR 中，任何类都肯定从一个(而且只能是一个)类派生，后者最终从 **Object** 派生。这个类称为基类。基类提供了一组方法签名和这些方法的实现。你定义的新类可在将来由其他开发人员用作基类——所有方法签名和方法实现都会由新的派生类继承。

CLR 还允许开发人员定义接口，它实际只是对一组方法签名进行了统一命名。这些方法不提供任何实现。类通过指定接口名称来继承接口，而且必须显式实现接口方法，否则 CLR 会认为此类型定义无效。当然，实现接口方法的过程可能比较烦琐，所以我在前面说接口继承是实现多继承的一种“缩水版”机制。C#编译器 and CLR 允许一个类继承多个接口。当然，继承的所有接口方法都必须实现。

我们知道，类继承的一个重要特点是，凡是能使用基类型实例的地方，都能使用派生类型的实例。类似地，接口继承的一个重点特点是，凡是能使用具名接口类型的实例的地方，都能使用实现了接口的一个类型的实例。下面先看看如何定义接口。

13.2 定义接口

如前所述，接口对一组方法签名进行了统一命名。注意，接口还能定义事件、无参属性和有参属性(C#的索引器)。如前所述，所有这些东西本质上都是方法，它们只是语法上的简化。不过，接口不能定义任何构造器方法，也不能定义任何实例字段。

虽然 CLR 允许接口定义静态方法、静态字段、常量和静态构造器，但符合 CLS 标准的接口绝不允许，因为有的编程语言不能定义或访问它们。事实上，C#禁止接口定义任何一种这样的静态成员。

C#用 **interface** 关键字定义接口。要为接口指定名称和一组实例方法签名。下面是 FCL 中的几个接口的定义：

```
public interface IDisposable {
    void Dispose();
}

public interface IEnumerable {
    IEnumerator GetEnumerator();
}

public interface IEnumerable<T> : IEnumerable {
    new IEnumerator<T> GetEnumerator();
}

public interface ICollection<T> : IEnumerable<T>, IEnumerable {
    void Add(T item);
```

```

void      Clear();
Boolean   Contains(T item);
void      CopyTo(T[] array, Int32 arrayIndex);
Boolean   Remove(T item);
Int32     Count      { get; } // 只读属性
Boolean   IsReadOnly { get; } // 只读属性
}

```

在 CLR 看来，接口定义就是类型定义。也就是说，CLR 会为接口类型对象定义内部数据结构，同时可通过反射机制来查询接口类型的功能。和类型一样，接口可在文件范围中定义，也可嵌套在另一个类型中。定义接口类型时，可指定你希望的任何可见性/可访问性(**public**, **protected**, **internal** 等)。

根据约定，接口类型名称以大写字母 **I** 开头，目的是方便在源代码中辨认接口类型。CLR 支持泛型接口(前面几个例子已进行了演示)和接口中的泛型方法。本章稍后会讨论泛型接口的许多功能。另外，第 12 章“泛型”已全面讨论了泛型。

接口定义可从另一个或多个接口“继承”。但“继承”应打上引号，因为它并不是严格的继承。接口继承的工作方式并不完全和类继承一样。我个人倾向于将接口继承看成是将其其他接口的协定(contract)包括到新接口中。例如，**ICollection<T>**接口定义就包含了 **IEnumerable<T>**和 **IEnumerable** 两个接口的协定。这有下面两层含义。

- 继承 **ICollection<T>**接口的任何类必须实现 **ICollection<T>**，**IEnumerable<T>**和 **IEnumerable** 这三个接口所定义的方法。
- 任何代码在引用对象时，如果期待该对象的类型实现了 **ICollection<T>**接口，可以认为该类型还实现了 **IEnumerable<T>**和 **IEnumerable** 接口。

13.3 继承接口

本节介绍如何定义实现了接口的类型，然后介绍如何创建该类型的实例，并用这个对象调用接口的方法。C#将这个过程变得很简单，但幕后发生的事情还是有点复杂。本章稍后会详细解释。

下面是在 **mscorlib.dll** 中定义的 **System.IComparable<T>**接口：

```

public interface IComparable<in T> {
    Int32 CompareTo(T other);
}

```

以下代码展示了如何定义实现了该接口的类型，同时还展示了对两个 **Point** 对象进行比较的代码：

```

using System;

// Point 从 System.Object 派生，并实现了 IComparable<T>
public sealed class Point : IComparable<Point> {
    private Int32 m_x, m_y;

    public Point(Int32 x, Int32 y) {
        m_x = x;
    }
}

```

```

        m_y = y;
    }

    // 该方法为 Point 实现 IComparable<T>.CompareTo()
    public Int32 CompareTo(Point other) {
        return Math.Sign(Math.Sqrt(m_x * m_x + m_y * m_y)
            - Math.Sqrt(other.m_x * other.m_x + other.m_y * other.m_y));
    }

    public override String ToString() {
        return String.Format("{0}, {1}", m_x, m_y);
    }
}

public static class Program {
    public static void Main() {
        Point[] points = new Point[] {
            new Point(3, 3),
            new Point(1, 2),
        };

        // 下面调用由 Point 实现的 IComparable<T> 的 CompareTo 方法
        if (points[0].CompareTo(points[1]) > 0) {
            Point tempPoint = points[0];
            points[0] = points[1];
            points[1] = tempPoint;
        }
        Console.WriteLine("Points from closest to (0, 0) to farthest:");
        foreach (Point p in points)
            Console.WriteLine(p);
    }
}

```

C#编译器要求将实现接口的方法(后文简称为“接口方法”)标记为 **public**。CLR 要求将接口方法标记为 **virtual**。不将方法显式标记为 **virtual**，编译器会将它们标记为 **virtual** 和 **sealed**；这会阻止派生类重写接口方法。将方法显式标记为 **virtual**，编译器就会将该方法标记为 **virtual**(并保持它的非密封状态)，使派生类能重写它。

派生类不能重写 **sealed** 的接口方法。但派生类可重新继承同一个接口，并为接口方法提供自己的实现。在对象上调用接口方法时，调用的是该方法在该对象的类型中的实现。下例对此进行了演示：

```

using System;

public static class Program {
    public static void Main() {
        /***** 第一个例子 *****/
        Base b = new Base();

        // 用 b 的类型来调用 Dispose，显示: "Base's Dispose"
        b.Dispose();

        // 用 b 的对象的类型来调用 Dispose，显示: "Base's Dispose"
        ((IDisposable)b).Dispose();

        /***** 第二个例子 *****/
        Derived d = new Derived();

        // 用 d 的类型来调用 Dispose，显示: "Derived's Dispose"
        d.Dispose();
    }
}

```

```

// 用 d 的对象的类型来调用 Dispose, 显示: "Derived's Dispose"
((IDisposable)d).Dispose();

/***** 第三个例子 *****/
b = new Derived();

// 用 b 的类型来调用 Dispose, 显示: "Base's Dispose"
b.Dispose();

// 用 b 的对象的类型来调用 Dispose, 显示: "Derived's Dispose"
((IDisposable)b).Dispose();
}

// 这个类派生自 Object, 它实现了 IDisposable
internal class Base : IDisposable {
    // 这个方法隐式密封, 不能被重写
    public void Dispose() {
        Console.WriteLine("Base's Dispose");
    }
}

// 这个类派生自 Base, 它重新实现了 IDisposable
internal class Derived : Base, IDisposable {
    // 这个方法不能重写 Base 的 Dispose,
    // 'new' 表明该方法重新实现了 IDisposable 的 Dispose 方法
    new public void Dispose() {
        Console.WriteLine("Derived's Dispose");
    }

    // 注意, 下面这行代码展示了如何调用基类的实现 (如果需要的话)
    // base.Dispose();
}

```

13.4 关于调用接口方法的更多探讨

FCL 的 **System.String** 类型继承了 **System.Object** 的方法签名及其实现。此外, **String** 类型还实现了几个接口: **IComparable**, **ICloneable**, **IConvertible**, **IEnumerable**, **IComparable<String>**, **IEnumerable<Char>** 和 **IEquatable<String>**。这意味着 **String** 类型不需要实现(或重写)其 **Object** 基类型提供的方法, 但必须实现所有接口声明的方法。

CLR 允许定义接口类型的字段、参数或局部变量。使用接口类型的变量可以调用该接口定义的方法。此外, CLR 允许调用 **Object** 定义的方法, 因为所有类都继承了 **Object** 的方法。以下代码对此进行了演示:

```

// s 变量引用一个 String 对象
String s = "Jeffrey";
// 可以使用 s 调用在 String, Object, IComparable, ICloneable,
// IConvertible, IEnumerable 中定义的任何方法

// cloneable 变量引用同一个 String 对象
ICloneable cloneable = s;
// 使用 cloneable 只能调用 ICloneable 接口声明的
// 任何方法 (或 Object 定义的任何方法)

```

```
// comparable 变量引用同一个 String 对象
IComparable comparable = s;
// 使用 comparable 只能调用 IComparable 接口声明的
// 任何方法(或 Object 定义的任何方法)

// enumerable 变量引用同一个 String 对象
// 可在运行时将变量从一个接口转换成另一个, 只要
// 对象的类型实现了这两个接口
IEnumerable enumerable = (IEnumerable) comparable;
// 使用 enumerable 只能调用 IEnumerable 接口声明的
// 任何方法(或 Object 定义的任何方法)
```

在这段代码中, 所有变量都引用同一个“Jeffrey” **String** 对象。该对象在托管堆中; 所以, 使用其中任何变量时, 调用的任何方法都会影响这个“Jeffrey” **String** 对象。不过, 变量的类型规定了能对这个对象执行的操作。**s** 变量是 **String** 类型, 所以可以用 **s** 调用 **String** 类型定义的任何成员(比如 **Length** 属性)。还可用变量 **s** 调用从 **Object** 继承的任何方法(比如 **GetType**)。

cloneable 变量是 **ICloneable** 接口类型。所以, 使用 **cloneable** 变量可以调用该接口定义的 **Clone** 方法。此外, 可以调用 **Object** 定义的任何方法(比如 **GetType**), 因为 CLR 知道所有类型都继承自 **Object**。不过, 不能用 **cloneable** 变量调用 **String** 本身定义的公共方法, 也不能调用由 **String** 实现的其他任何接口的方法。类似地, 使用 **comparable** 变量可以调用 **CompareTo** 方法或 **Object** 定义的任何方法, 但不能调用其他方法。



重要提示 和引用类型相似, 值类型可实现零个或多个接口。但值类型的实例在转换为接口类型时必须装箱。这是由于接口变量是引用, 必须指向堆上的对象, 使 CLR 能检查对象的类型对象指针, 从而判断对象的确切类型。调用已装箱值类型的接口方法时, CLR 会跟随对象的类型对象指针找到类型对象的方法表, 从而调用正确的方法。

13.5 隐式和显式接口方法实现(幕后发生的事情)

类型加载到 CLR 中时, 会为该类型创建并初始化一个方法表(参见第 1 章“CLR 的执行模型”)。在这个方法表中, 类型引入的每个新方法都有对应的记录项; 另外, 还为该类型继承的所有虚方法添加了记录项。继承的虚方法既有继承层次结构中的各个基类型定义的, 也有接口类型定义的。所以, 对于下面这个简单的类型定义:

```
internal sealed class SimpleType : IDisposable {
    public void Dispose() { Console.WriteLine("Dispose"); }
}
```

类型的方法表将包含以下方法的记录项。

- **Object**(隐式继承的基类)定义的所有虚实例方法。

- **IDisposable**(继承的接口)定义的所有接口方法。本例只有一个方法, 即 **Dispose**, 因为 **IDisposable** 接口只定义了这个方法。
- **SimpleType** 引入的新方法 **Dispose**。

为简化编程, C#编译器假定 **SimpleType** 引入的 **Dispose** 方法是对 **IDisposable** 的 **Dispose** 方法的实现。之所以这样假定, 是由于 **Dispose** 方法的可访问性是 **public**, 而接口方法的签名和新引入的方法完全一致。也就是说, 两个方法具有相同的参数和返回类型。顺便说一句, 如果新的 **Dispose** 方法被标记为 **virtual**, C#编译器仍然认为该方法匹配接口方法。

C#编译器将新方法和接口方法匹配起来之后, 会生成元数据, 指明 **SimpleType** 类型的方法表中的两个记录项应引用同一个实现。为了更清楚地理解这一点, 下面的代码演示了如何调用类的公共 **Dispose** 方法以及如何调用 **IDisposable** 的 **Dispose** 方法在类中的实现:

```
public sealed class Program {
    public static void Main() {
        SimpleType st = new SimpleType();

        // 调用公共 Dispose 方法实现
        st.Dispose();

        // 调用 IDisposable 的 Dispose 方法的实现
        IDisposable d = st;
        d.Dispose();
    }
}
```

在第一个 **Dispose** 方法调用中, 调用的是 **SimpleType** 定义的 **Dispose** 方法。然后定义 **IDisposable** 接口类型的变量 **d**, 它引用 **SimpleType** 对象 **st**。调用 **d.Dispose()** 时, 调用的是 **IDisposable** 接口的 **Dispose** 方法。由于 C#要求公共 **Dispose** 方法同时是 **IDisposable** 的 **Dispose** 方法的实现, 所以会执行相同的代码。在这个例子中, 两个调用你看不出任何区别。输出结果如下所示:

```
Dispose
Dispose
```

现在重写 **SimpleType**, 以便于看出区别:

```
internal sealed class SimpleType : IDisposable {
    public void Dispose() { Console.WriteLine("public Dispose"); }
    void IDisposable.Dispose() { Console.WriteLine("IDisposable Dispose"); }
}
```

在不改动前面的 **Main** 方法的前提下, 重新编译并再次运行程序, 输出结果如下所示:

```
public Dispose
IDisposable Dispose
```

在 C#中, 将定义方法的那个接口的名称作为方法名前缀(例如 **IDisposable.Dispose**), 就会创建显式接口方法实现(Explicit Interface Method Implementation, EIMI^①)。注意, C#中不允

① 请记住 EIMI 的意思, 本书后面会大量使用这个缩写词。——译注

许在定义显式接口方法时指定可访问性(比如 **public** 或 **private**)。但是, 编译器生成方法的元数据时, 可访问性会自动设为 **private**, 防止其他代码在使用类的实例时直接调用接口方法。只有通过接口类型的变量才能调用接口方法。

还要注意, EIMI 方法不能标记为 **virtual**, 所以不能被重写。这是由于 EIMI 方法并非真的是类型的对象模型的一部分, 它只是将接口(一组行为或方法)和类型连接起来, 同时避免公开行为/方法。如果觉得这一点不好理解, 那么你的感觉没有错! 它就是不太好理解。本章稍后会介绍 EIMI 有用的一些场合。

13.6 泛型接口

C#和 CLR 所支持的泛型接口为开发人员提供了许多非常出色的功能。本节要讨论泛型接口提供的一些好处。

首先, 泛型接口提供了出色的编译时类型安全性。有的接口(比如非泛型 **IComparable** 接口)定义的方法使用了 **Object** 参数或 **Object** 返回类型。在代码中调用这些接口方法时, 可传递对任何类型的实例的引用。但这通常不是我们期望的。下面的代码对此进行了演示:

```
private void SomeMethod1() {
    Int32 x = 1, y = 2;
    IComparable c = x;

    // CompareTo 期待 Object; 传递 y(一个 Int32)没有问题
    c.CompareTo(y); // y 在这里装箱

    // CompareTo 期待 Object, 传递"2" (一个 String)可以编译,
    // 但在运行时抛出 ArgumentException 异常
    c.CompareTo("2");
}
```

接口方法理想情况下应该使用强类型。这正是 FCL 为什么包含泛型 **IComparable<in T>** 接口的原因。下面修改代码来使用泛型接口:

```
private void SomeMethod2() {
    Int32 x = 1, y = 2;
    IComparable<Int32> c = x;

    // CompareTo 期待 Int32, 传递 y(一个 Int32)没有问题
    c.CompareTo(y); // y 在这里不装箱

    // CompareTo 期待 Int32, 传递"2"(一个 String)造成编译错误,
    // 指出 String 不能被转换为 Int32
    c.CompareTo("2"); // 错误
}
```

泛型接口的第二个好处在于, 处理值类型时装箱次数会少很多。在 **SomeMethod1** 中, 非泛型 **IComparable** 接口的 **CompareTo** 方法期待获取一个 **Object**; 传递 **y(Int32 值类型)** 会造成 **y** 中的值装箱。但在 **SomeMethod2** 中, 泛型 **IComparable<in T>** 接口的 **CompareTo** 方法本来期待的就是 **Int32**; **y** 以传值的方式传递, 无需装箱。



注意 FCL 定义了 **Comparable**, **ICollection**, **ICollection** 和 **IDictionary** 等接口的泛型和非泛型版本。定义类型时要实现其中任何接口, 一般应实现泛型版本。FCL 保留非泛型版本是为了向后兼容, 照顾在 .NET Framework 支持泛型之前写的代码。非泛型版本还允许用户以较常规的、类型较不安全(more general, less type-safe)的方式处理数据。

有的泛型接口继承了非泛型版本, 所以必须同时实现接口的泛型和非泛型版本。例如, 泛型 **IEnumerable<out T>** 接口继承了非泛型 **IEnumerable** 接口, 所以实现 **IEnumerable<out T>** 就必须实现 **IEnumerable**。

和其他代码集成时, 有时必须实现非泛型接口, 因为接口的泛型版本并不存在。这时, 如果接口的任何方法获取或返回 **Object**, 就会失去编译时的类型安全性, 而且值类型将发生装箱。可利用本章 13.9 节“用显式接口方法实现来增强编译时类型安全性”介绍的技术来缓解该问题。

泛型接口的第三个好处在于, 类可以实现同一个接口若干次, 只要每次使用不同的类型参数。以下代码对这个好处进行了演示:

```
using System;

// 该类实现泛型 Comparable<T>接口两次
public sealed class Number: Comparable<Int32>, Comparable<String> {
    private Int32 m_val = 5;

    // 该方法实现 Comparable<Int32>的 CompareTo 方法
    public Int32 CompareTo(Int32 n) {
        return m_val.CompareTo(n);
    }

    // 该方法实现 Comparable<String>的 CompareTo 方法
    public Int32 CompareTo(String s) {
        return m_val.CompareTo(Int32.Parse(s));
    }
}

public static class Program {
    public static void Main() {
        Number n = new Number();

        // 将 n 中的值和一个 Int32(5) 比较
        Comparable<Int32> cInt32 = n;
        Int32 result = cInt32.CompareTo(5);

        // 将 n 中的值和一个 String("5") 比较
        Comparable<String> cString = n;
        result = cString.CompareTo("5");
    }
}
```

接口的泛型类型参数可标记为逆变和协变, 为泛型接口的使用提供更大的灵活性。欲知协

变和逆变的详情，请参见 12.5 节“委托和接口的逆变和协变泛型类型实参”。

13.7 泛型和接口约束

上一节讨论了泛型接口的好处。本节要讨论将泛型类型参数约束为接口的好处。

第一个好处在于，可将泛型类型参数约束为多个接口。这样一来，传递的参数类型必须实现全部接口约束。例如：

```
public static class SomeType {
    private static void Test() {
        Int32 x = 5;
        Guid g = new Guid();

        // 对 M 的调用能通过编译，因为 Int32 实现了
        // IComparable 和 IConvertible
        M(x);

        // 这个 M 调用导致编译错误，因为 Guid 虽然
        // 实现了 IComparable，但没有实现 IConvertible
        M(g);
    }

    // M 的类型参数 T 被约束为只支持同时实现了
    // IComparable 和 IConvertible 接口的类型
    private static Int32 M<T>(T t) where T : IComparable, IConvertible {
        ...
    }
}
```

这真的很“酷”！定义方法参数时，参数的类型规定了传递的实参必须是该类型或者它的派生类型。如果参数的类型是接口，那么实参可以是任意类类型，只要该类实现了接口。使用多个接口约束，实际是表示向方法传递的实参必须实现多个接口。

事实上，如果将 **T** 约束为一个类和两个接口，就表示传递的实参类型必须是指定的基类(或者它的派生类)，而且必须实现两个接口。这种灵活性使方法能细致地约束调用者能传递的内容。调用者不满足这些约束，就会产生编译错误。

接口约束的第二个好处是传递值类型的实例时减少装箱。上述代码向 **M** 方法传递了 **x**(值类型 **Int32** 的实例)。**x** 传给 **M** 方法时不会发生装箱。如果 **M** 方法内部的代码调用 **t.CompareTo(...)**，这个调用本身也不会引发装箱(但传给 **CompareTo** 的实参可能发生装箱)。

另一方面，如果 **M** 方法像下面这样声明：

```
private static Int32 M(IComparable t) {
    ...
}
```

那么 **x** 要传给 **M** 就必须装箱。

C#编译器为接口约束生成特殊 IL 指令，导致直接在值类型上调用接口方法而不装箱。不用

接口约束便没有其他办法让 C# 编译器生成这些 IL 指令，如此一来，在值类型上调用接口方法总是发生装箱。一个例外是如果值类型实现了一个接口方法，在值类型的实例上调用这个方法不会造成值类型的实例装箱。

13.8 实现多个具有相同方法名和签名的接口

定义实现多个接口的类型时，这些接口可能定义了具有相同名称和签名的方法。例如，假定有以下两个接口：

```
public interface IWindow {
    Object GetMenu();
}

public interface IRestaurant {
    Object GetMenu();
}
```

要定义实现这两个接口的类型，必须使用“显式接口方法实现”来实现这个类型的成员，如下所示：

```
// 这个类型派生自 System.Object,
// 并实现了 IWindow 和 IRestaurant 接口
public sealed class MarioPizzeria : IWindow, IRestaurant {

    // 这是 IWindow 的 GetMenu 方法的实现
    Object IWindow.GetMenu() { ... }

    // 这是 IRestaurant 的 GetMenu 方法的实现
    Object IRestaurant.GetMenu() { ... }

    // 这个 GetMenu 方法是可选的，与接口无关
    public Object GetMenu() { ... }
}
```

由于这个类型必须实现多个接口的 **GetMenu** 方法，所以要告诉 C# 编译器每个 **GetMenu** 方法对应的是哪个接口的实现。

代码在使用 **MarioPizzeria** 对象时必须将其转换为具体的接口才能调用所需的方法。例如：

```
MarioPizzeria mp = new MarioPizzeria();

// 这行代码调用 MarioPizzeria 的公共 GetMenu 方法
mp.GetMenu();

// 以下代码调用 MarioPizzeria 的 IWindow.GetMenu 方法
IWindow window = mp;
window.GetMenu();

// 以下代码调用 MarioPizzeria 的 IRestaurant.GetMenu 方法
IRestaurant restaurant = mp;
restaurant.GetMenu();
```

13.9 用显式接口方法实现来增强编译时类型安全性

接口很好用，它们定义了类型之间进行沟通的标准方式。前面曾讨论了泛型接口，讨论了它们如何增强编译时的类型安全性和减少装箱操作。遗憾的是，有时由于不存在泛型版本，所以仍需实现非泛型接口。接口的任何方法接受 **System.Object** 类型的参数或返回 **System.Object** 类型的值，就会失去编译时的类型安全性，装箱也会发生。本节将介绍如何用“显式接口方法实现”(EIMI)在某种程度上改善这个局面。

下面是极其常用的 **IComparable** 接口：

```
public interface IComparable {
    Int32 CompareTo(Object other);
}
```

该接口定义了接受一个 **System.Object** 参数的方法。可以像下面这样定义实现了该接口的类型：

```
internal struct SomeValueType : IComparable {
    private Int32 m_x;
    public SomeValueType(Int32 x) { m_x = x; }
    public Int32 CompareTo(Object other) {
        return m_x - ((SomeValueType) other).m_x;
    }
}
```

可用 **SomeValueType** 写下面这样的代码：

```
public static void Main() {
    SomeValueType v = new SomeValueType(0);
    Object o = new Object();
    Int32 n = v.CompareTo(v); // 不希望的装箱操作
    n = v.CompareTo(o);      // InvalidCastException 异常
}
```

上述代码存在两个问题。

- 不希望的装箱操作
v 作为实参传给 **CompareTo** 方法时必须装箱，因为 **CompareTo** 期待的是一个 **Object**。
- 缺乏类型安全性
代码能通过编译，但 **CompareTo** 方法内部试图将 o 转换为 **SomeValueType** 时抛出 **InvalidCastException** 异常。

这两个问题都可以用 EIMI 解决。下面是 **SomeValueType** 的修改版本，这次添加了一个 EIMI：

```
internal struct SomeValueType : IComparable {
    private Int32 m_x;
    public SomeValueType(Int32 x) { m_x = x; }

    public Int32 CompareTo(SomeValueType other) {
        return(m_x - other.m_x);
    }
}
```

```
// 注意以下代码没有指定 public/private 可访问性
Int32 IComparable.CompareTo(Object other) {
    return CompareTo((SomeValueType) other);
}
}
```

注意新版本的几处改动。现在有两个 **CompareTo** 方法。第一个 **CompareTo** 方法不是获取一个 **Object** 作为参数，而是获取一个 **SomeValueType**。这样就不必将 **other** 类型转换为 **SomeValueType** 了，所以用于强制类型转换的代码被去掉了。修改了第一个 **CompareTo** 方法使其变得类型安全之后，**SomeValueType** 还必须实现一个 **CompareTo** 方法来满足 **IComparable** 的协定。这正是第二个 **IComparable.CompareTo** 方法的作用，它是一个 EIMI。

经过这两处改动之后，就获得了编译时的类型安全性，而且不会发生装箱：

```
public static void Main() {
    SomeValueType v = new SomeValueType(0);
    Object o = new Object();
    Int32 n = v.CompareTo(v); // 不发生装箱
    n = v.CompareTo(o);      // 编译时错误
}
```

不过，定义接口类型的变量会再次失去编译时的类型安全性，而且会再次发生装箱：

```
public static void Main() {
    SomeValueType v = new SomeValueType(0);
    IComparable c = v;      // 装箱！

    Object o = new Object();
    Int32 n = c.CompareTo(v); // 不希望的装箱操作
    n = c.CompareTo(o);      // InvalidCastException 异常
}
```

事实上，如本章前面所述，将值类型的实例转换为接口类型时，CLR 必须对值类型的实例进行装箱。因此，前面的 **Main** 方法中会发生两次装箱。

实现 **IConvertible**、**ICollection**、**ICollection** 和 **IDictionary** 等接口时 EIMI 很有用。可利用它为这些接口的方法创建类型安全的版本，并减少值类型的装箱。

13.10 谨慎使用显式接口方法实现

使用 EIMI 也可能造成一些严重后果，所以应该尽量避免使用 EIMI。幸好，泛型接口可帮助我们在大多数时候避免使用 EIMI。但有时(比如实现具有相同名称和签名的两个接口方法时)仍然需要它们。EIMI 最主要的问题如下。

- 没有文档解释类型具体如何实现一个 EIMI 方法，也没有 **Microsoft Visual Studio** “智能感知”支持。
- 值类型的实例在转换成接口时装箱。
- EIMI 不能由派生类型调用。

下面详细讨论这些问题。

文档在列出一个类型的方法时，会列出显式接口方法实现(EIMI)，但没有提供类型特有的帮助，只有接口方法的常规性帮助。例如，**Int32** 类型的文档只是说它实现了 **IConvertible** 接口的所有方法。能做到这一步已经不错，它使开发人员知道存在这些方法。但也使开发人员感到困惑，因为不能直接在一个 **Int32** 上调用一个 **IConvertible** 方法。例如，下面的代码无法编译：

```
public static void Main() {
    Int32 x = 5;
    Single s = x.ToSingle(null); // 试图调用一个 IConvertible 方法
}
```

编译这个方法时，C#编译器会报告以下消息：**error CS0117: "int"不包含"ToSingle"的定义**。这个错误信息使开发人员感到困惑，因为它明显是说 **Int32** 类型没有定义 **ToSingle** 方法，但实际上定义了。

要在一个 **Int32** 上调用 **ToSingle**，首先必须将其转换为 **IConvertible**，如下所示：

```
public static void Main() {
    Int32 x = 5;
    Single s = ((IConvertible) x).ToSingle(null);
}
```

对类型转换的要求不明确，而许多开发人员自己看不出来问题出在哪里。还有一个更让人烦恼的问题：**Int32** 值类型转换为 **IConvertible** 会发生装箱，既浪费内存，又损害性能。这是本节开头提到的 EIMI 存在的第二个问题。

EIMI 的第三个也可能是最大的问题是，它们不能被派生类调用。下面是一个例子：

```
internal class Base : IComparable {
    // 显式接口方法实现
    Int32 IComparable.CompareTo(Object o) {
        Console.WriteLine("Base's CompareTo");
        return 0;
    }
}

internal sealed class Derived : Base, IComparable {
    // 一个公共方法，也是接口的实现
    public Int32 CompareTo(Object o) {
        Console.WriteLine("Derived's CompareTo");

        // 试图调用基类的 EIMI 导致编译错误：
        // error CS0117: "Base" 不包含 "CompareTo" 的定义
        base.CompareTo(o);
        return 0;
    }
}
```

在 **Derived** 的 **CompareTo** 方法中调用 **base.CompareTo** 导致 C#编译器报错。现在的问题是，**Base** 类没有提供一个可供调用的公共或受保护 **CompareTo** 方法，它提供的是一个只能用 **IComparable** 类型的变量来调用的 **CompareTo** 方法。可将 **Derived** 的 **CompareTo** 方

法修改成下面这样:

```
// 一个公共方法, 也是接口的实现
public Int32 CompareTo(Object o) {
    Console.WriteLine("Derived's CompareTo");

    // 试图调用基类的 EIMI 导致无穷递归
    IComparable c = this;
    c.CompareTo(o);

    return 0;
}
```

这个版本将 **this** 转换成 **IComparable** 变量 **c**, 然后用 **c** 调用 **CompareTo**。但 **Derived** 的公共 **CompareTo** 方法充当了 **Derived** 的 **IComparable.CompareTo** 方法的实现, 所以造成了无穷递归。这可以通过声明没有 **IComparable** 接口的 **Derived** 类来解决:

```
internal sealed class Derived : Base /*, IComparable */ { ... }
```

现在, 前面的 **CompareTo** 方法将调用 **Base** 中的 **CompareTo** 方法。但有时不能因为想在派生类中实现接口方法就将接口从类型中删除。解决这个问题的最佳方法是在基类中除了提供一个被选为显式实现的接口方法, 还要提供一个虚方法。然后 **Derived** 类可以重写虚方法。下面展示了如何正确定义 **Base** 类和 **Derived** 类:

```
internal class Base : IComparable {

    // 显式接口方法实现 (EIMI)
    Int32 IComparable.CompareTo(Object o) {
        Console.WriteLine("Base's IComparable.CompareTo");
        return CompareTo(o); // 调用虚方法
    }

    // 用于派生类的虚方法 (该方法可为任意名称)
    public virtual Int32 CompareTo(Object o) {
        Console.WriteLine("Base's virtual CompareTo");
        return 0;
    }
}

internal sealed class Derived : Base, IComparable {

    // 一个公共方法, 也是接口的实现
    public override Int32 CompareTo(Object o) {
        Console.WriteLine("Derived's CompareTo");

        // 现在可以调用 Base 的虚方法
        return base.CompareTo(o);
    }
}
```

注意, 这里是将虚方法定义成公共方法, 但有时可能需要定义成受保护方法。把方法定义为受保护(而不是公共)是可以的, 但必须进行另一些小的改动。我们的讨论清楚证明了务必谨慎使用 EIMI。许多开发人员在最初接触 EIMI 时, 认为 EIMI 非常“酷”, 于是开始肆无忌惮地使用。千万不要这样做! EIMI 在某些情况下确实有用, 但应该尽量避免使用, 因为它们导致类型变得很不好用。

13.11 设计：基类还是接口

经常有人问：“应该设计基类还是接口？”这个问题不能一概而论，以下设计规范或许能帮你理清思路。

- **IS-A 对比 CAN-DO 关系^①**

类型只能继承一个实现。如果派生类型和基类型建立不起 IS-A 关系，就不用基类而用接口。接口意味着 CAN-DO 关系。如果多种对象类型都“能”做某事，就为它们创建接口。例如，一个类型能将自己的实例转换为另一个类型(**IConvertible**)，一个类型能序列化自己的实例(**ISerializable**)。注意，值类型必须从 **System.ValueType** 派生，所以不能从一个任意的基类派生。这时必须使用 CAN-DO 关系并定义接口。

- **易用性**

对于开发人员，定义从基类派生的新类型通常比实现接口的所有方法容易得多。基类型可提供大量功能，所以派生类型可能只需稍做改动。而提供接口的话，新类型必须实现所有成员。

- **一致性实现**

无论接口协定(contract)订立得有多好，都无法保证所有人百分之百正确实现它。事实上，COM 颇受该问题之累，导致有的 COM 对象只能正常用于 Microsoft Office Word 或 Microsoft Internet Explorer。而如果为基类型提供良好的默认实现，那么一开始得到的就是能正常工作并经过良好测试的类型。以后根据需要修改就可以了。

- **版本控制**

向基类型添加一个方法，派生类型将继承新方法。一开始使用的就是一个能正常工作的类型，用户的源代码甚至不需要重新编译。而向接口添加新成员，会强迫接口的继承者更改其源代码并重新编译。

FCL 中涉及数据流处理(streaming data)的类采用的是实现继承方案^②。**System.IO.Stream** 是抽象基类，提供了包括 **Read** 和 **Write** 在内的一组方法。其他类(**System.IO.FileStream**，**System.IO.MemoryStream** 和 **System.Net.Sockets.NetworkStream**)都从 **Stream** 派生。在这三个类中，每一个和 **Stream** 类都是 IS-A 关系，这使具体类^③的实现变得更容易。例如，派生类只需实现同步 I/O 操作，异步 I/O 操作已经从 **Stream** 基类继承了。

必须承认，为流类(**XXXStream**)选择继承的理由不是特别充分，因为 **Stream** 基类实际只提

① IS-A 是指“属于”，例如，汽车属于交通工具；CAN-DO 是指“能做某事”，例如，一个类型能将自己的实例转换为另一个类型。——译注

② 即继承基类的实现。——译注

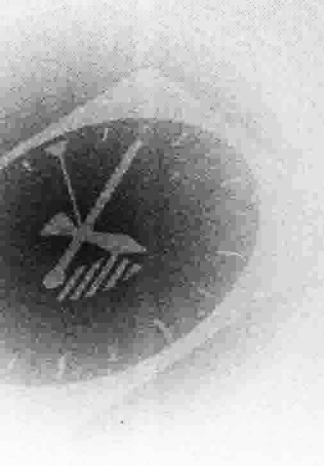
③ 对应于“抽象类”。——译注

供了很少的实现。那么就以 Microsoft Windows 窗体控件类为例好了。**Button**, **CheckBox**, **ListBox** 和其他所有窗体控件都从 **System.Windows.Forms.Control** 派生。**Control** 实现了大量代码, 各种控件类简单继承一下即可正常工作。这时选择继承应该没有疑问了吧?

相反, Microsoft 采用基于接口的方式来设计 FCL 中的集合。**System.Collections.Generic** 命名空间定义了几个与集合有关的接口: **IEnumerable<out T>**, **ICollection<T>**, **IList<T>** 和 **IDictionary<TKey, TValue>**。然后, Microsoft 提供了大量类来实现这些接口组合, 包括 **List<T>**, **Dictionary<TKey, TValue>**, **Queue<T>** 和 **Stack<T>** 等等。设计者在类和接口之间选择 CAN-DO 关系, 因为不同集合类的实现迥然各异。换句话说, **List<T>**, **Dictionary<TKey, TValue>** 和 **Queue<T>** 之间没有多少能共享的代码。

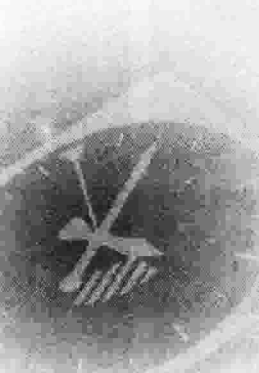
不过, 这些集合类提供的操作相当一致。例如, 都维护了一组可枚举的元素, 而且都允许添加和删除元素。假定有一个对象引用, 对象的类型实现了 **IList<T>** 接口, 就可在不需要知道集合准确类型的前提下插入、删除和搜索元素。这个机制太强大了!

最后要说的是, 两件事情实际能同时做: 定义接口, 同时提供实现该接口的基类。例如, FCL 定义了 **IComparer<in T>** 接口, 任何类型都可选择实现该接口。此外, FCL 提供了抽象基类 **Comparer<T>**, 它实现了该接口, 同时为非泛型 **IComparer** 的 **Compare** 方法提供了默认实现。接口定义和基类同时存在带来了很大的灵活性, 开发人员可根据需要选择其中一个。



第III部分 基本类型

- ▶ 第 14 章 字符、字符串和文本处理
- ▶ 第 15 章 枚举类型和位标志
- ▶ 第 16 章 数组
- ▶ 第 17 章 委托
- ▶ 第 18 章 定制特性
- ▶ 第 19 章 可空值类型



第 14 章 字符、字符串和文本处理

本章内容:

- 字符
- **System.String** 类型
- 高效率构造字符串
- 获取对象的字符串表示: **ToString**
- 解析字符串来获取对象: **Parse**
- 编码: 字符和字节的相互转换
- 安全字符串

本章将解释在 Microsoft .NET Framework 中处理字符和字符串的机制。首先讨论 **System.Char** 结构以及处理字符的多种方式。然后讨论更有用的 **System.String** 类, 它允许处理不可变(immutable)字符串(一经创建, 字符串便不能以任何方式修改)。探讨了字符串之后, 将介绍如何使用 **System.Text.StringBuilder** 类高效率地动态构造字符串。掌握了字符串的基础知识之后, 将讨论如何将对象格式化成字符串, 以及如何使用各种编码方案高效率地持久化或传输字符串。最后讨论 **System.Security.SecureString** 类, 它保护密码和信用卡资料等敏感字符串。

14.1 字符

在 .NET Framework 中, 字符总是表示成 16 位 Unicode 代码值, 这简化了国际化应用程序的开发。每个字符都是 **System.Char** 结构(一个值类型)的实例。**System.Char** 类型很简单, 提供了两个公共只读常量字段: **MinValue**(定义成 '\0')和 **MaxValue**(定义成 '\uffff')。

为 **Char** 的实例调用静态 **GetUnicodeCategory** 方法返回 **System.Globalization.UnicodeCategory** 枚举类型的一个值, 表明该字符是由 Unicode 标准定义的控制字符、货币符号、小写字母、大写字母、标点符号、数学符号还是其他字符。

为了简化开发, **Char** 类型还提供了几个静态方法, 包括 **IsDigit**, **IsLetter**, **IsUpper**, **IsLower**, **IsPunctuation**, **IsLetterOrDigit**, **IsControl**, **IsNumber**, **IsSeparator**, **IsSurrogate**,

IsLowSurrogate , **IsHighSurrogate** 和 **IsSymbol** 等。大多数都在内部调用了 **GetUnicodeCategory** , 并简单地返回 **true** 或 **false**。注意, 所有这些方法要么获取单个字符作为参数, 要么获取一个 **String** 以及目标字符在这个 **String** 中的索引作为参数。

另外, 可调用静态方法 **ToLowerInvariant** 或者 **ToUpperInvariant** , 以忽略语言文化(culture)的方式将字符转换为小写或大写形式。另一个方案是调用 **ToLower** 和 **ToUpper** 方法来转换大小写, 但转换时会使用与调用线程关联的语言文化信息(方法在内部查询 **System.Threading.Thread** 类的静态 **CurrentCulture** 属性来获得)。也可向这些方法传递 **CultureInfo** 类的实例来指定一种语言文化。**ToLower** 和 **ToUpper** 之所以需要语言文化信息, 是因为字母的大小写转换是一种依赖于语言文化的操作。比如在土耳其语中, 字母 U+0069(小写拉丁字母 i)转换成大写是 U+0130(大写拉丁字母 İ, 上面加一点), 而其他语言文化的转换结果是 U+0049(大写拉丁字母 I)。

除了这些静态方法, **Char** 类型还有自己的实例方法。其中, **Equals** 方法在两个 **Char** 实例代表同一个 16 位 Unicode 码位^①的前提下返回 **true**。**CompareTo** 方法(由 **IComparable** 和 **IComparable<Char>** 接口定义)返回两个 **Char** 实例的忽略语言文化的比较结果。**ConvertFromUtf32** 方法从一个 UTF-32 字符生成包含一个或两个 UTF-16 字符的字符串。**ConvertToUtf32** 方法从一对低/高代理项或者字符串生成一个 UTF-32 字符。**ToString** 方法返回包含单个字符的一个 **String**。与 **ToString** 相反的是 **Parse/TryParse** , 它们获取单字符的 **String** , 返回该字符的 UTF-16 码位。

最后一个方法是 **GetNumericValue** , 它返回字符的数值形式。以下代码演示了这个方法。

```
using System;

public static class Program {
    public static void Main() {
        Double d;
        d = Char.GetNumericValue('\u0033'); // '\u0033'是"数字 3"
        Console.WriteLine(d.ToString()); // 显示"3"

        // '\u00bc'是“普通分数四分之一(¼)”
        d = Char.GetNumericValue('\u00bc');
        Console.WriteLine(d.ToString()); // 显示"0.25"

        // 'A'是“大写拉丁字母 A”
        d = Char.GetNumericValue('A');
        Console.WriteLine(d.ToString()); // 显示"-1"
    }
}
```

最后, 可以使用三种技术实现各种数值类型与 **Char** 实例的相互转换。下面按照优先顺序列出这些技术。

① 在字符编码术语中, 码位或称编码位置, 即英文的 **code point** 或 **code position** , 是组成码空间(或代码页)的数值。例如, ASCII 码包含 128 个码位。——维基百科

- **转型(强制类型转换)**

将 **Char** 转换成数值(比如 **Int32**)最简单的办法就是转型。这是三种技术中效率最高的,因为编译器会生成中间语言(IL)指令来执行转换,而且不必调用方法。此外,有的语言(比如 C#)允许指定转换时是使用 **checked** 还是 **unchecked** 代码(参见 5.1 节“编程语言的基元类型”)。

- **使用 Convert 类型**

System.Convert 类型提供了几个静态方法来实现 **Char** 和数值类型的相互转换。所有这些转换都以 **checked** 方式执行,发现转换将造成数据丢失就抛出 **OverflowException** 异常。

- **使用 IConvertible 接口**

Char 类型和 FCL 中的所有数值类型都实现了 **IConvertible** 接口。该接口定义了像 **ToUInt16** 和 **ToChar** 这样的方法。这种技术效率最差,因为在值类型上调用接口方法要求对实例进行装箱——**Char** 和所有数值类型都是值类型。如果某个类型不能转换(比如 **Char** 转换成 **Boolean**),或者转换将造成数据丢失,**IConvertible** 的方法会抛出 **System.InvalidCastException** 异常。注意,许多类型(包括 FCL 的 **Char** 和数值类型)都将 **IConvertible** 的方法实现为显式接口成员^①。这意味着为了调用接口的任何方法,都必须先将实例显式转型为一个 **IConvertible**。**IConvertible** 的所有方法(**GetTypeCode** 除外)都接受对实现了 **IFormatProvider** 接口的一个对象的引用。如果转换时需要考虑语言文化信息,该参数就很有用。但大多数时候都可以忽略语言文化,为这个参数传递 **null** 值。

以下代码演示了如何使用这三种技术。

```
using System;

public static class Program {
    public static void Main() {
        Char c;
        Int32 n;

        // 通过 C# 转型(强制类型转换)实现数字与字符的相互转换
        c = (Char) 65;
        Console.WriteLine(c);           // 显示"A"

        n = (Int32) c;
        Console.WriteLine(n);           // 显示"65"

        c = unchecked((Char) (65536 + 65));
        Console.WriteLine(c);           // 显示"A"

        // 使用 Convert 实现数字与字符的相互转换
        c = Convert.ToChar(65);
        Console.WriteLine(c);           // 显示"A"

        n = Convert.ToInt32(c);
        Console.WriteLine(n);           // 显示"65"
```

① 参见 13.9 节“用显式接口方法实现来增强编译时类型安全性”。

```
// 演示 Convert 的范围检查
try {
    c = Convert.ToChar(70000);    // 对 16 位来说过大
    Console.WriteLine(c);        // 不执行
}
catch (OverflowException) {
    Console.WriteLine("Can't convert 70000 to a Char.");
}

// 使用 IConvertible 实现数字与字符的相互转换
c = ((IConvertible) 65).ToChar(null);
Console.WriteLine(c); // 显示"A"

n = ((IConvertible) c).ToInt32(null);
Console.WriteLine(n); // 显示"65"
}
```

14.2 System.String 类型

在任何应用程序中，**System.String** 都是用得最多的类型之一。一个 **String** 代表一个不可变 (immutable) 的顺序字符集。**String** 类型直接派生自 **Object**，所以是引用类型。因此，**String** 对象(它的字符数组)总是存在于堆上，永远不会跑到线程栈^①。**String** 类型还实现了几个接口 (**IComparable**/**IComparable<String>**，**ICloneable**，**IConvertible**，**IEnumerable**/**IEnumerable<Char>**和 **IEquatable<String>**)。

14.2.1 构造字符串

许多编程语言(包括 C#)都将 **String** 视为基元类型——也就是说，编译器允许在源代码中直接使用字面值(literal)字符串。编译器将这些字符串放到模块的元数据中，并在运行时加载和引用它们。

C#不允许使用 **new** 操作符从字面值字符串构造 **String** 对象：

```
using System;

public static class Program {
    public static void Main() {
        String s = new String("Hi there."); // 错误
        Console.WriteLine(s);
    }
}
```

相反，必须使用以下简化语法：

```
using System;

public static class Program {
    public static void Main() {
        String s = "Hi there.";
        Console.WriteLine(s);
    }
}
```

① 堆和线程栈的详情请参见 4.4 节“运行时的相互关系”。

```
}
}
```

编译代码并检查 IL(使用 ILDasm.exe), 会看到以下内容:

```
.method public hidebysig static void Main() cil managed
{
    .entrypoint
    // Code size 13 (0xd)
    .maxstack 1
    .locals init (string V_0)
    IL_0000: ldstr "Hi there."
    IL_0005: stloc.0
    IL_0006: ldloc.0
    IL_0007: call void [mscorlib]System.Console::WriteLine(string)
    IL_000c: ret
} // end of method Program::Main
```

用于构造对象新实例的 IL 指令是 **newobj**。但上述 IL 代码中并没有出现 **newobj** 指令, 只有一个特殊 **ldstr**(即 **load string**)指令, 它使用从元数据获得的字面值(literal)字符串构造 **String** 对象。这证明 CLR 实际是用一种特殊方式构造字面值 **String** 对象。

如果使用不安全的(unsafe)代码, 可以从一个 **Char***或 **Sbyte***构造一个 **String**。这时要使用 C#的 **new** 操作符, 并调用由 **String** 类型提供的、能接受 **Char***或 **Sbyte***参数的某个构造器。这些构造器将创建 **String** 对象, 根据由 **Char** 实例或有符号(signed)字节构成的一个数组来初始化字符串。其他构造器则不允许接受任何指针参数, 用任何托管编程语言写的安全(可验证)代码都能调用它们。^①

C#提供了一些特殊语法来帮助开发人员在源代码中输入字面值(literal)字符串。对于换行符、回车符和退格符这样的特殊字符, C#采用的是 C/C++开发人员熟悉的转义机制:

```
// 包含回车符和换行符的字符串
String s = "Hi\r\nthere.";
```



重要提示 上例虽然在字符串中硬编码了回车符和换行符, 但一般不建议这样做。相反, **System.Environment** 类型定义了只读 **NewLine** 属性。应用程序在 Microsoft Windows 上运行时, 该属性返回由回车符和换行符构成的字符串。**NewLine** 属性对平台敏感, 会根据底层平台来返回恰当的字符串。例如, 如果将公共语言基础结构(CLI)移植到 UNIX 系统, **NewLine** 属性将返回由单字符 **\n** 构成的字符串。以下才是定义上述字符串的正确方式, 它在任何平台上都能正确工作:

```
String s = "Hi" + Environment.NewLine + "there.";
```

可以使用 C#的+操作符将几个字符串连接成一个。如下所示:

```
// 三个字面值(literal)字符串连接成一个字面值字符串
```

① 记住, 除非指定了/unsafe 编译器开关, 否则 C#代码必须是安全的或者说具有可验证性, 确保代码不会引起安全风险和稳定性风险。详情参见 1.4.2 节“不安全的代码”。——译注