第24章 从面向对象到SOA

本章导读：

本书前面用了 23 章的篇幅向读者深入介绍了面向对象的许多知识与编程技巧，然而，了解这些知识和技巧是一回事，会灵活应用它们则是另一回事。在实际开发中，所有的软件产品或项目都是综合应用多种技术的结果。我们不仅要深入探索和把握具体的技术领域，更要掌握“组合的艺术”，要注意培养出依据实际情况选择合适的技术、设计合理的方案、采取正确的方法、遵循严格的流程来开发软件的能力。

开发大型的功能复杂的软件系统（比如 Windows）是一项浩大的工程，需要采用严格的管理手段来保证项目的成功，有关这方面话题的探讨，属于软件工程领域，超过了本书所介绍的范畴。

本书更关注那些规模较小的，一个人或者是最多几个人就可以完成的软件，这种软件的开发过程，在笔者看来与“炒菜”非常类似。我们可以把本书中介绍的许多技术看成是各种食物原料，对具体技术的学习可看成是对食物原料进行的初步加工与处理，应用这些技术开发一个程序就是将这些已经加工好的原料“下锅烹调”，而最终端上桌子的“菜”，就是我们劳动的成果——一个可以解决实际问题的软件。

做的菜好不好吃，固然与原料有关，但更关键因素的是厨师本人，一个能在五星级酒店里掌勺的“戴着高高的帽子”的大厨，他的烹调水平，不是大多数普通人所能达到的。采用同样的原料、遵循同样的步骤，我们做出来的菜可能就是没有大厨做得好吃，这里面的原因太多了，要说清楚并不容易。

但我们也不用对大厨“顶礼膜拜”，因为没有人能生而知之，相信大厨自己也是一步一步走过来的。对于大多数人而言，只要能选准一个最适合的领域，勤奋努力，成为这个领域内的“大厨”也是有可能的。

本章所展示的，是笔者本人开发一个小小的四则运算器程序的过程实录。

四则运算是小学生都必须掌握的基本技能，然而要编写一个功能完备的程序来计算四则运算表达式的值，却并不像看上去的那么简单。事实上，笔者看到过一些计算机专业的学生，学了四年之后，毕业时却连这样一个小小的计算器程序也编不好。

这个小程序看上去不值一提，而且似乎用处有限，但“麻雀虽小，五脏俱全”，笔者认为：“小”和“大”的区别是相对的，“小”和“大”也不是能截然分开的，有许多开发方法和原则，其实是普遍的、相通的，与软件规模大小无关。

本章通过仔细剖析这样一个小程序，可以帮助读者将已学过的知识编织成一个知识的网络，为将知识转化为能力打下良好的基础，同时，读者还可以直观地

了解到一个真实的软件的成形过程。

希望本章能对读者更好地理解面向对象的软件开发过程有所帮助。

24.1 面向对象软件的开发过程

24.1.1 OOA、OOD、OOP和OOT

“面向对象的分析（Object Oriented Analysis，OOA）”指的是运用面向对象方法对将要开发的软件系统功能进行分析。

OOA 的核心问题是区分清楚“问题域与系统责任”（图 24-1）。

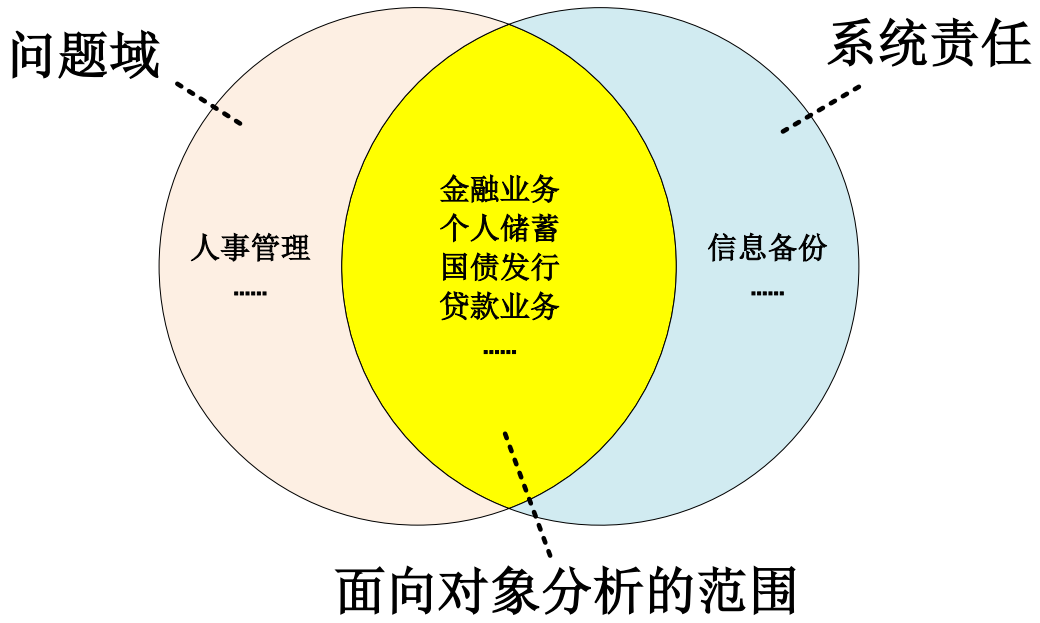

图 24-1 问题域与系统责任

所谓**“问题域”**，指的是软件系统所应用的具体业务领域，比如“银行业务”就是一个典型的“问题域”。

而**“系统责任”**，则是指开发出来的软件系统所应具备的功能。

并非“问题域”中所有的功能都需要实现，而“系统责任”中的部分功能也不属于问题域，但两者之间的交集属于面向对象分析的范围。

OOA 的结果被称为**“领域模型”**。

OOA 的侧重点是业务领域，与此软件系统所要应用的行业领域直接相关，而与软件技术本身关系不大，这一部分的工作被称为**“需求分析”**，通常需要由系统分析员和相关业务领域专家来承担。

“面向对象的设计（Object Oriented Design，OOD）”指用面向对象的方法为真实世界建立一个“活”在计算机中的虚拟模型。它是 OOA 所得到的“领域模型”的接收者，而它的输出为可以直接指导开发的“软件设计文档”。

OOD 的主要任务是跨越业务领域模型与可实际运行的软件系统之间的鸿沟。负责 OOD 工作的人被称为“架构设计师”。

在 OOD 完成之后，进入**“面向对象编程（Object Oriented Program，OOP）”**阶段，

其主要任务是用面向对象的语言来实现 OOD 完成的系统设计，其承担者为软件工程师。OOP 是本书着重介绍的内容。

图 24-2 展示了 OOA、OOD 和 OOP 三者之间的关系。

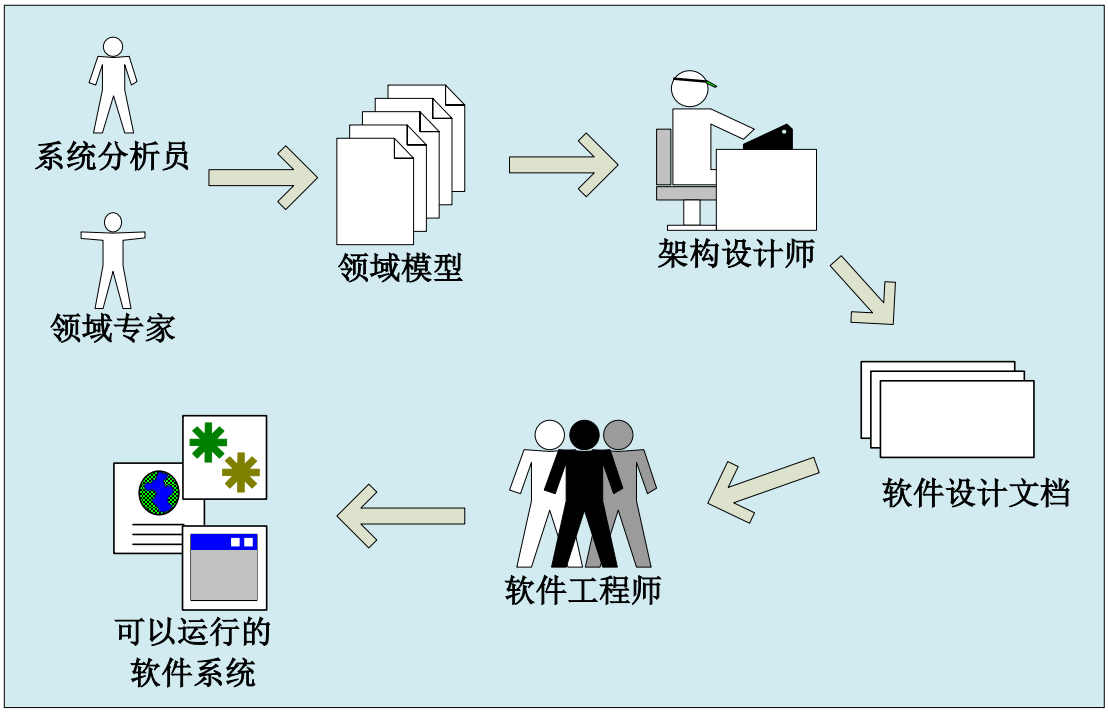

图 24-2 OOA、OOD 和 OOP

“面向对象的测试（Object-oriented Test， OOT）”是指针对 OOP 所得到的软件系统（源码和可执行的组件或子系统）进行测试的过程，其目的在于尽可能地找出软件系统中存在的 Bug，反馈给程序员进行修正。

交叉链接：

本书第 21 章《基于 Attribute 的开发与应用》中介绍了如何使用 Visual Studio 2010 进行单元测试，以及.NET 4.0 所引入的“代码协定（Code Contracts）”特性，这两项技术应用得当，有助于得到高质量的代码。

24.1.2 增量与迭代开发

面向对象软件系统的开发过程通常可以使用一个“喷泉”来描述（图 24-3）。

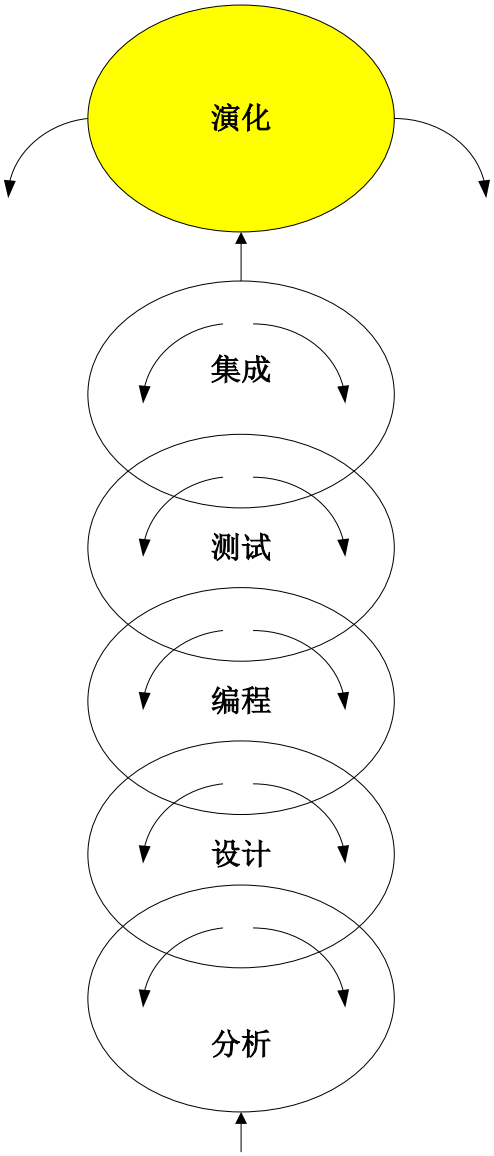

图 24-3 面向对象软件系统开发的“喷泉”模型

如图 24-3 所示，在面向对象软件系统的开发过程中，从下往上，依次经历分析、设计等各个开发阶段，而每个阶段又可以“回归”到前一阶段，构成一种“迭代”的开发过程。

在具体实施时，经常采用的是“**基于软件组件的增量开发模型**”。

使用这种方式开发软件，整个软件系统都是由软件组件以类似于“搭积木”的形式装配而成。系统的开发工作可以简化为“开发新的组件”，“复用已有的组件”和“组件装配”三大部分。系统使用的各个软件组件都完成某种特定的功能，许多组件可以在多个项目中重复使用。

在.NET Framework 中，软件组件以“程序集（Assembly）”的形式存在，.NET Framework 是一个支持组件化开发的优秀软件平台。

所谓“增量开发模型”，具体来说，就是先做出软件系统的一个尽可能简单、但能运行的版本。它不必接受真实的输入，也无须对数据进行真正的处理，更不用产生真实的输出——它仅仅需要构成一个足够强壮的骨架，支撑起未来将要开发的真实系统。

在骨架形成之后，开始一点点地完善它，逐步增加功能。

整个开发过程如同滚雪球，每给系统增加一个功能，相当于“软件雪球”滚一圈，增大一圈，直到最后得到一个满足需求的软件系统（图 24-4）。

图 24-4 开发软件类似于“滚雪球”

在整个增量开发过程中，采用的是“步步为营”的策略，这就是说：每给系统增加一个功能，给就马上对其进行测试，保证其工作正常，然后再开发新的功能。

在 Visual Studio 中，引入了多项提升软件开发效率的工具，比如单元测试就是实施增量开发的技术手段之一。

在本章后面的章节中，将以四则运算器的开发为例，向读者展示一个典型的面向对象软件开发的全过程

此示例程序的功能很简单，唯一的功能就是完成一个四则运算表达式的解析和计算工作。程序有两个界面，标准界面与精简界面（图 24-5）。

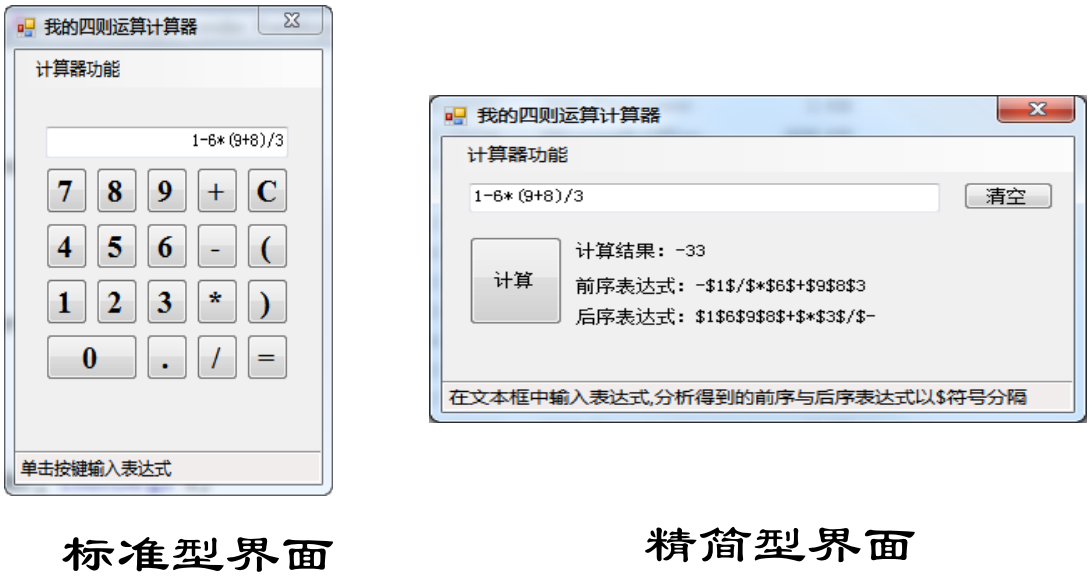

图 24-5 计算器的标准界面

标准型界面下用户使用鼠标单击相应按钮输入一个四则运算表达式之后，单击“＝”按

钮计算出结果。

在精简型界面下用户可直接用键盘在文本框中输入表达式，敲回车键计算出结果。同时，程序显示表达式前序与后序解析的结果，显示此结果是出于方便读者学习的目的。

示例程序可以动态选择解析表达式的算法（图 24-6）。

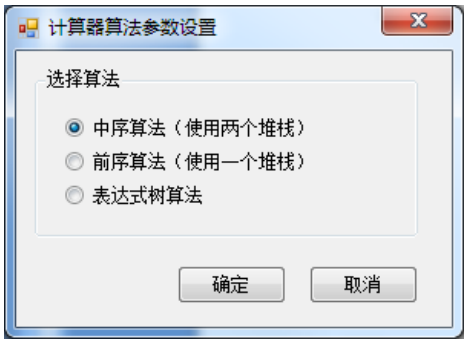

图 24-6 算法选择

此程序的特色在于：

（1）具有一定的健壮性，对各种非法的表达式可以检测出错误并给出相关的提示；

（2）具有较好的可扩充性，可以方便地在程序中用新算法替换掉老算法。

一个真正实用的程序是解决生活中现实问题的程序，像本章示例，虽然不复杂，但的确可以做一些“实实在在”的事，这就是与本书前面以展示技术特性为目的的示例的根本区别。

软件是帮助人们更好更快地完成某项工作的，要编写一个软件，首先就要知道人是如何完成某项工作的；接着分析如何用计算机来完成这个工作，其结果是形成一个计算机算法，这要求有相关领域的理论知识；之后进入设计阶段；最后是编码和测试工作。

在本章的后几节中，将从计算机算法、软件架构设计与实际软件开发过程三个方面介绍这个四则运算计算器的开发过程。

24.2 四则运算数学模型的建立

本节建立解析四则运算表达式的数学模型。

24.2.1 四则运算表达式

诸如“9＋5×（4—3）”之类的数学表达式称为“四则运算表达式”。

四则运算表达式具有以下特点：

（1）只有加减乘除四种运算符；

（2）运算符具有优先级：乘除优先于加减；

（3）可以有多个括号，左右括号必须配对，括号的优先级最高。

上述三点为四则运算表达式的基本特性。

手工计算四则运算表达式非常简单，小学生都能很轻松地完成，然而，要编程序来让计算机完成这个工作则并不容易，人脑习惯的手工计算四则运算表达式的方法不能直接“搬到”计算机中来，必须设计出专用的计算机算法。

24.2.2 四则运算表达式解析算法分析

四则运算表达式在计算机中原始的表达方式是“字符串”。因此，四则运算表达式解析的过程就是边扫描字符串边进行分析的过程。

1 将字符串划分为字符单元

在扫描字符串过程中，首先要将字符串中的字符分成有意义的单元，比如以下表达式：

```
1.5 + 3.5 * 2
```

共有 9 个字符，应分为“1.5”，“+”，“3.5”，“*”，“2”共五个字符单元。

将字符串划分为有意义的字符单元之后，再将这些单元分为“操作数”和“运算符”两组（表 24-1）。

表 24-1 字符单元类别表

种类	字符单元
操作数	“1.5”、“3.5”、“2”
运算符	“+”、“*”

接着是进行计算的问题。

2 前序、中序和后序表达式

我们常见的四则运算表达式都是这种形式的：

```
X + Y
```

即运算符在两个操作数中间，这种形式的表达式称为“**中序（Infix）表达式**”。

对应地，“**前序（Prefix）表达式**”运算符在前，操作数在后：

```
+ X Y
```

“**后序（Postfix）表达式**”操作数在前，运算符在后：

```
X Y +
```

中序表达式看起来非常自然，而前序和后序表达式就显得非常“怪异”，请看表 24-2。

表 24-2 四则运算表达式的三种形式

种类	表达式
中序表达式	2 * 3 / (2 - 1) + 5 * (4 – 1)
前序表达式	+ / * 2 3 – 2 1 * 5 – 4 1
后序表达式	2 3 * 2 1 - / 5 4 1 - * +

这就很奇怪了，中序表达式看上去多么自然，为何要设计出前序和后序表达式呢？

请仔细观察表 24-2，可以发现前序与后序表达式中不再有括号，而且在计算过程中可以是非常机械的单向扫描，不需要考虑运算符的优先级。

3 使用前序算法计算四则运算表达式

以前序表达式“+/*23-21*5-41”为例，从右到左，先取出两个操作数“1”、“4”和一个运算符“-”，计算“4-1”，其结果为“3”，将其回填到字符串中（为方便区分，回填的数字用中括号与原字符串隔开，现在字符串变成：“+/*23-21*5[3]”。

继续从右到左取两个数“3”和“5”，一个运算符“*”，“3*5”=15，回填字符串，得“+/*23-21[15]”。

再取数，这时，需连续取出 3 个数：“15”、“1”、“2”，直到取出一个运算符“-”为止，将与运算符“-”最接近的两个操作数进行计算，“2-1”＝1，回填到字符串中，现在字符串变成：“+/*23[1][15]”。

重复上述过程，这次取出“2”、“3”和“*”，计算“2×3”＝6，回填：“＋/[6][1][15]”。

接着取出“6”、“1”和“/”，计算“6/1”=6，回填：“＋[6] [15]”。

现在只剩下两个操作数和一个运算符了，结果已经出来：“6＋15”＝21。

上述过程虽然烦琐，规则却很简单：

从右到左取数，直到取出一个运算符，将紧挨着运算符的两个操作数按运算符进行计算，结果回填回字符串。直到最后只剩两个操作数和一个运算符，即可计算出结果。

后序表达式的字符串扫描方式与前序刚好相反，是从左到右扫描，规则也是类似的。

4 表达式树

“**表达式树（Expression Tree）**”是另一种计算四则运算表达式的方法。其特点是将四则运算表达式转化为一棵二叉树，以“2*3/(2-1)+5*(4-1)”为例，此表达式生成以下表达式树（图 24-7）。

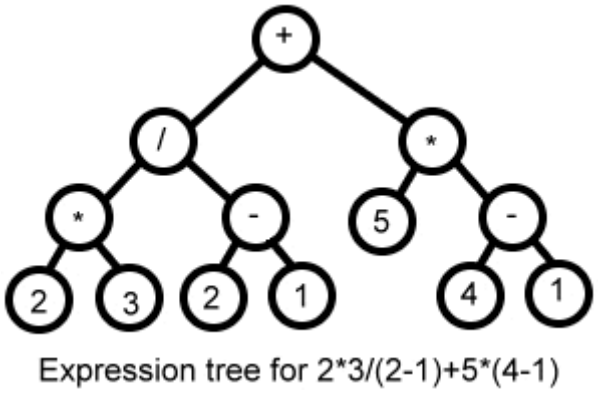

图 24-7 表达式树

求值时，从最底层的叶子节点着手，逐级计算，其结果放在中间的运算符结点中，最后，放在根节点中的值就是表达式计算的结果。

表达式树与表达式的前序、中序和后序表示法有着密切的联系，读者只要试一下，对图 24-7 的树进行前序遍历（即先输出根节点，再输出左、右子节点），即可生成表达式的

前序表示形式。生成中序与后序表达式的过程类似。

24.2.3 数据结构与算法的面向对象设计

在从数学和计算机算法两个角度解决了四则运算表达式的解析问题之后，就可以考虑如何编程来实现得到的计算机表达式解析算法。

1 表达式对象

表达式用一个类 MathExpression 表达（图 24-8）。

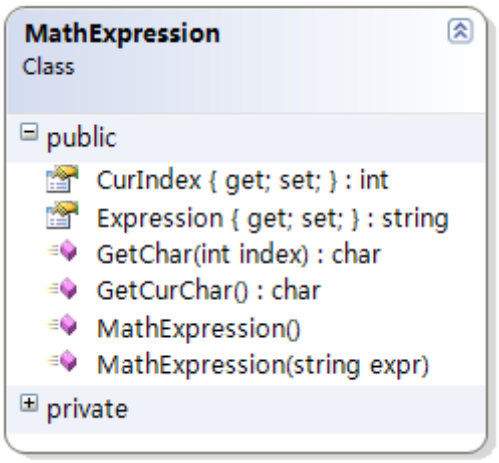

图 24-8 表达式类

表达式是一个字符串，保存在 Expression 属性中。由于在算法中需要不断地扫描此字符串并从中取出字符，因此，要求表达式对象能“记住”当前正在访问哪个字符，故设计一个 CurIndex 属性用于表明当前字符索引。GetCurChar 方法则根据此属性值取出当前字符。

MathExpression 类的另一个公有方法 GetChar 依据其参数值取出指定索引处的字符。

程序中对字符串扫描的过程可以看成是 CurIndex 属性值不断变化的过程，让 CurIndex 属性值从 0 依次增长到“字符串的长度值 - 1”，不断调用 GetCurChar 方法就可完成对字符串的扫描过程。

2 数字的提取

本章示例程序使用一个有穷状态自动机[1]来实现从表达式字符串中取出一个合法数字的功能（图 24-9），一个有穷状态自动机拥有若干个状态（状态数有限），状态之间可以相互转换。

[1] 有穷状态自动机（Finite Automata）分为确定型（Deterministic）和非确定型（Nondeterministic）两大类，常简称为 DFA 和 NFA。自动机理论是编译器设计与算法分析重要的理论基础，也是《编译原理》、《计算理论》这两门计算机专业骨干课程的讲授内容，感兴趣的读者可以自行参阅相关教材和技术书籍。

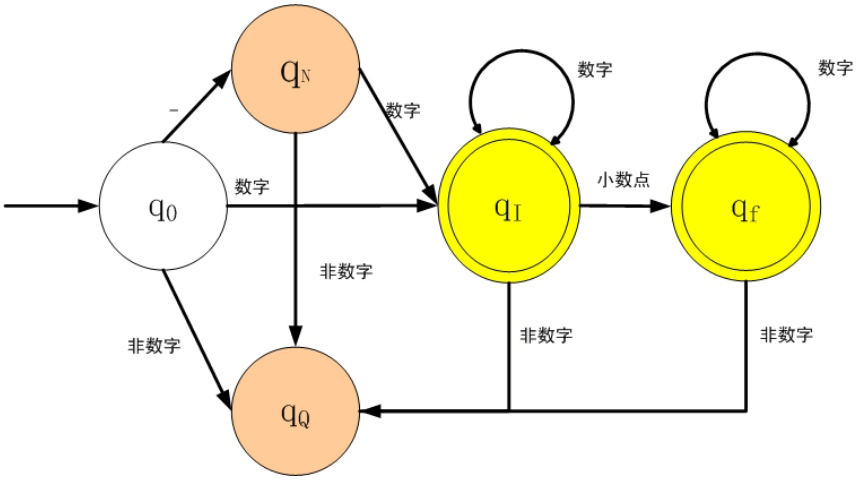

图 24-9 有穷状态自动机

图 24-9 中的有穷状态自动机有五个状态，其意义如表 24-3 所示。

表 24-3 自动机状态说明

状态标识	含义
q0	初始状态
qN	负数的状态：N = Negative
qI	整数的状态：I = Integer
qF	浮点数的状态：F = Float
qQ	退出状态：Q = Quit

图 24-9 中的自动机完成的工作是从字符串中取出数字。其过程简述如下：

当表达式字符串扫描开始时，自动机居于 q0 状态，如果当前字符是数字，则转入 qI 状态（即处于整数状态），继续读入字符，如果还是数字，则自动机停在 qI 状态不动，如果读入一个小数点，则转入 qF 状态（即处于浮点数状态），可继续读入数字。

在 q0 状态下，如果当前字符为减号，则转入 qN 状态（表明现在居于负数状态），这时，若再读入数字转入 qI 状态，以后的状态转换就如上所述。

在上述过程中，只要读入一个非数字，则自动机将会退出到 qQ 状态，抽取数字过程结束。

此自动机可以读取由多个数字字符组成的整数和小数。

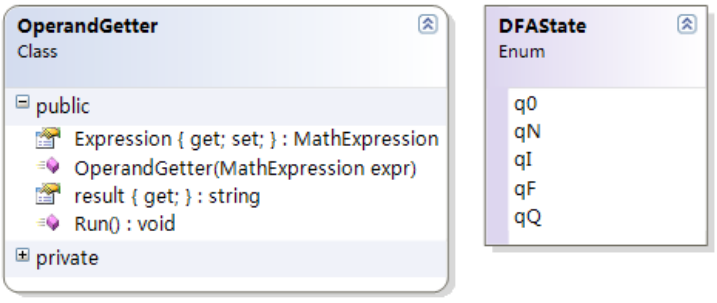

图 24-10 应用自动机取出数字

如图 24-10 所示，自动机的五个状态用一个枚举类型 DFAState 表达，OperandGetter 类完成取数字工作，先将四则运算表达式对象传入 OperandGetter 类的 Expression 属性，调用其 Run 方法之后，在 Result 属性中即为提取出来的数字，四则运算表达式对象的 curIndex 属性会自动指向下一个待读取的字符。

反复调用 GetOperand 类的 Run 方法，即可抽取出表达式字符串中的所有数字。

这里要注意，Run 方法要想正常工作，必须要求它所读入的第一个字符是数字或减号。不满足此要求则 Run 方法自动退出。

3 表达式树

仔细观察图 24-7 所示的表达式树，可以看到有两种类型的节点：操作数节点（OperandNode）与运算符节点（OperatorNode）。所以可设计一个表达式树节点基类 ExpressTreeNode，其他两种类型的节点由它派生出来（图 24-11）。

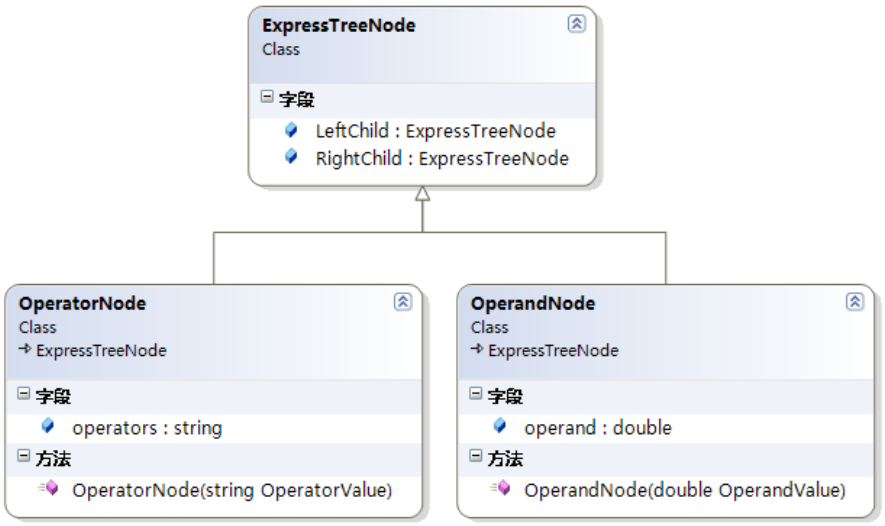

图 24-11 树节点的继承体系

可以看到，ExpressTreeNode 类中用两个公有字段 LeftChild 和 RightChild 表达其左右子树，操作数节点 OperandNode 拥有一个 Double 类型的字段用于存贮操作数，运算符节点 OperatorNode 则用一个 String 类型的字段存贮运算符。

创建树、遍历树等方法被封装到一个 ExpressTree 类中（图 24-12）。

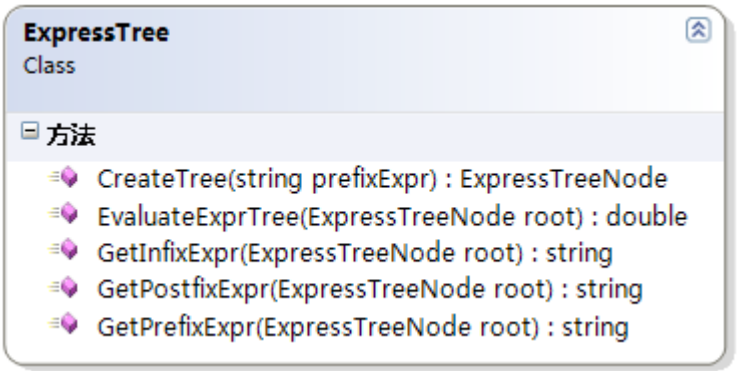

图 24-12 表达式树

如图 24-12 所示，CreateTree 方法创建一棵树，并返回其根节点，此方法接收一个前序表达式字符串，项目中的另一个表达式转换类 Converter（图 24-13）负责将中序表达式转为前序表达式。

EvaluateExprTree 方法递归遍历整个表达式树，并求解表达式的值。

当表达式树创建完成后，GetInfixExpr、GetPostfixExpr 和 GetPrefixExpr 三个方法根据表达式树生成中序、后序和前序表达式。

4 表达式类型转换类

类 Converter 集中了所有表达式类型转换的方法（图 24-13）。

图 24-13 表达式转换器

其中，最关键的静态方法是 InfixToPrefix，由它来完成中序表达式转前序表达式的工作，然后，私有方法 CreateExprTreeFromPrefix 调用 ExpressTree 类的 CreateTree 方法创建表达式树。

一旦表达式树创建完毕，前序、中序和后序这三种表现形式的转换就简单地体现为对树的“前序”，“中序”和“后序”遍历。而这些功能都已经封装到前面介绍的 ExpressTree 类中了。

由中序转为前序是一个比较复杂的算法，许多《数据结构》教材都没介绍，故具体说明如下：

输入： 中序表达式字符串

数据结构： 需要一个运算符栈

处理步骤：

1) 反转输入字符串（如“abcd”被反转为“dcba”）；

2) 从字符串中取下一个字符；

3) 如果是操作数，直接输出；

4) 如果是右括号“)”，将其压入运算符栈中；

5)如果是运算符：

i)若运算符栈空，此运算符进栈；

ii)若运算符栈顶是一个右括号“)”，将其弹出；

iii)如果此运算符与栈顶优先级相同或更高，此运算符进栈；

iv)否则，栈顶运算符出栈输出，然后重复第 5 步。

6)如果是操作数，出栈一个运算符并输出，重复此过程直到遇见一个“)”，“)”出栈并丢弃它；

7) 还有更多的待处理字符，转到第 2 步；

8) 不再有未处理的字符了，将栈中剩余元素出栈输出；

9) 再次反转输出字符串得到最终结果。

本书不是专门介绍算法设计的，因此不对此算法做详细解析，Converter 类中的 InfixToPrefix 方法实现了此算法。

对算法不感兴趣的读者可以跳过这一部分，不影响对后面内容的阅读。

5 四则运算表达式计算算法

在生成了前序和后序表达式后，计算表达式的值就是很机械的扫描过程。

类 PrefixAlgorithm 封装了前序表达式求值的过程（图 24-14）。

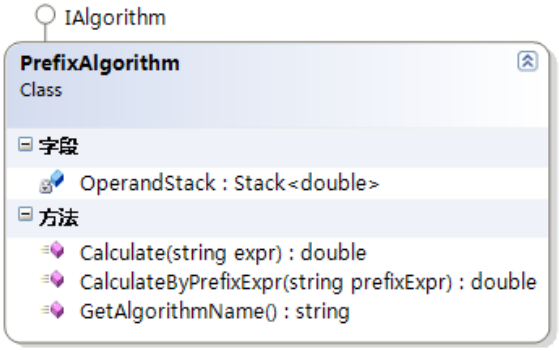

图 24-14 前序算法类

图 24-14 中，CalculateByPrefixExpr 方法根据前序表达式计算结果，其详细计算过程前已详述。

请注意本类的Calculate方法接收一个**中序表达式**作为其参数 [1]，在其内部先调用Converter类的InfixToPrefix方法将中序表达式转为前序表达式，再调用CalculateByPrefixExpr方法计算。

GetAlgorithmName 方法用于向外界返回算法的名称。

6 算法接口

由于本程序提供多个表达式解析算法，而且希望以后可以动态扩充，因此，设计了一个算法接口 IAlgorithm（图 24-15）。

[1] 之所以要让使用前序算法的 PrefixAlgorithm 类的 Calculate 方法接收一个中序表达式，是因为在日常生活中，人们使用的多为中序表达式。

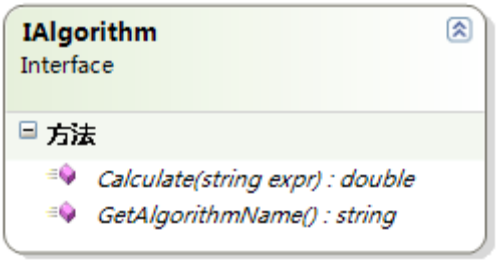

图 24-15 算法接口

IAlgorithm 接口只定义了两个方法，规定所有的算法都必须实现此接口（图 24-16）。

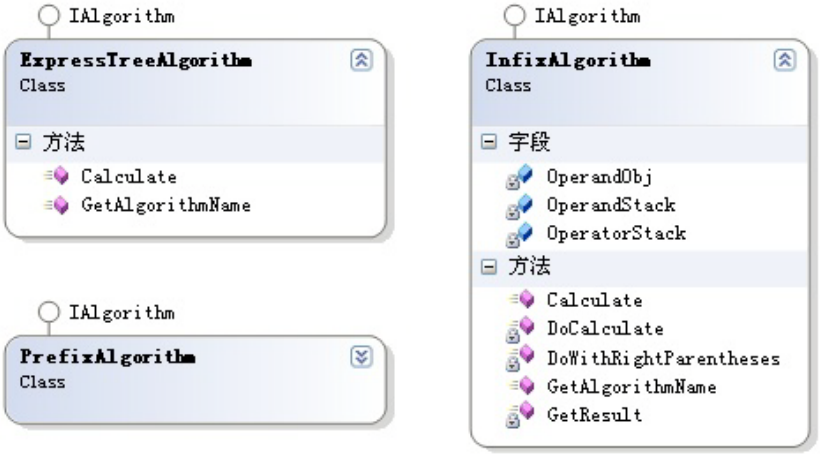

图 24-16 示例程序实现的算法

图 24-16 中，InfixAlgorithm 类封装的是中序算法，使用两个堆栈：一个操作数堆栈，一个运算符堆栈。很多数据结构的教材中都有此算法，本书不再介绍。

PrefixAlgorithm 类封装的是前序算法，其过程前文已有叙述。

ExpressTreeAlgorithm 类封装的是表达式树算法，采用由底向上递归遍历树的方法求值。

7 算法辅助类

笔者将一些常用到的功能组织为静态函数，统一封装到一个类 AlgorithmHelper 中（图 24-17）。

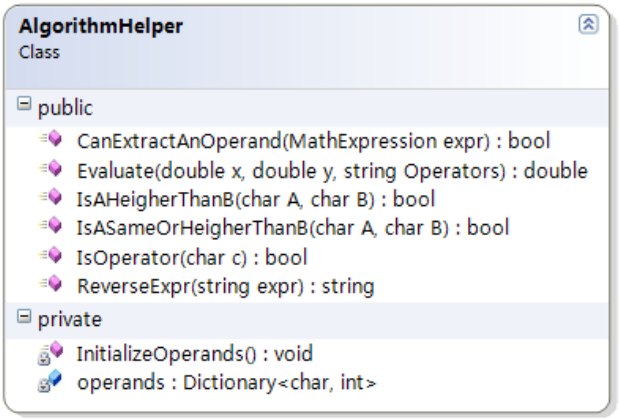

图 24-17 算法辅助类

此类各方法的详细说明请参看源码中的注释。

8 异常处理

从.NET Framework 提供的 ApplicationException 类中派生出一个自定义异常类 CalculatorException，用于捕获各种表达式解析错误（图 24-18）。

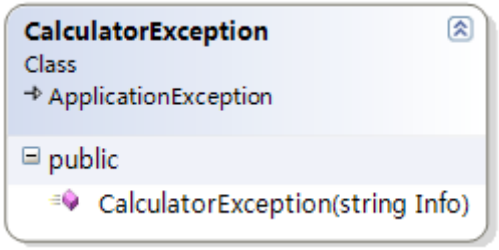

图 24-18 自定义异常

9 小结

现在对示例程序中的类进行一下汇总。

表 24-4 所示几个类为对数据结构的封装。

表 24-4 实现数据结构的类

类名	说明
MathExpression	代表四则运算表达式
ExpressTree	代表表达式树
ExpressTreeNode,OperatorNode,OperandNode	代表表达式树的节点

表 24-5 所示的三个类和一个接口，用于实现表达式求值算法。

表 24-5 实现算法的类

类名	说明
IAlgorithm	算法接口

InfixAlgrorithm	中序算法
PrefixAlgorithm	前序算法
ExpressTreeAlgorithm	表达式树算法

还有一个后序算法未实现，这个任务留给读者作为练习。

表 24-6 所示的两个类功能相对独立。

表 24-6 功能独立的算法类

类名	说明
Converter	前序、中序、后序表达式的相互转换
GetOperand	利用有穷状态自动机从字符串中抽取数字

表 24-7 所示的两个类起着辅助作用。

表 24-7 算法辅助类

类名	说明
AlgorithmHelper	算法辅助类
CalculatorException	自定义异常类

上述就是整个程序总体架构的核心部分，由于文字的局限，读者形成的印象可能是零乱的，请读者务必打开项目的各个源代码文件，仔细阅读每个类和方法的说明，并理解清楚这些类的职责与相互调用关系。

事实上，阅读面向对象的软件，最关键的就是弄清楚整个软件的架构。笔者的体会是首先弄清楚整个软件中有几个类，每个类是干什么的，这些类之间有什么关系（比如哪几个类间是继承关系，哪几个类是关联的，哪几个类又是依赖的，是否有多个类共同实现一个接口等等），将类按逻辑功能分为几个“群”，然后“各个突破”。千万不要一上来就一行行地“死啃”源代码，那很有可能陷入“只见树木不见森林”的窘境。

提示：

Visual Studio 2010 提供的类关系图（Class Diagram）和依赖图（Dependency Graph）是两个很好的工具，在阅读较大的程序可以帮助从整体上理解整个程序的架构。

有关这两个工具的使用方法，请读者自行查询 MSDN，本书不做详细介绍。

本节介绍的这些类都是属于中间层的软件模块，下一节将介绍整个软件的总体设计思路，以及用户界面层开发技巧。

24.3 软件体系结构设计方案

确定算法并用面向对象的方法将其实现，这些工作虽然十分重要，也是整个程序的核心，但这也仅仅是“万里长城走完了第一步”，要想真正开发出一个好的软件，还需要完成更多的工作。

24.3.1 确定软件处理流程

在设计四则运算表达式求值算法时，十分自然地假设四则运算表达式都是合法的，但在实际上，由于无法控制用户的输入，非法的表达式，比如“)9*1-3..5”之类是完全可能出现的。

对于这些非法的表达式，我们的算法不仅有可能算不出结果或得到错误的结果，还有可能出现死循环或无限递归这样的严重错误。

那么是否可由四则运算表达式求值算法本身来判断表达式的合法与否？这是一个可行的方案，但由于非法表达式的情况实在太多了，这就造成了算法类的“不务正业”，将大量的工作用在了校验表达式上，真正的表达式求值工作反而成了细枝末节。

在这点上可以看看编译器是怎样做的。

编译器在对付可能有着多种语法错误的源代码时，采用的策略是**多遍扫描**，一遍干一种特定的工作（当然可以一遍扫描干多个工作，但那是编译器工作流程优化的问题）。于是，我们也可以采用两遍扫描的方法，第一遍的主要目的是检验表达式的有效性，第二遍才开始解析表达式并求值。

经过这样的考虑，可以确定整个程序的处理流程（图 24-19）。

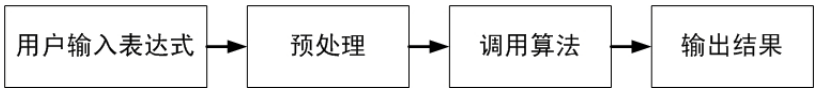

图 24-19 程序处理流程

如图 24-19 所示，“预处理”这一阶段完成第一遍扫描，“调用算法”这一阶段完成第二遍扫描。

预处理工作由类 PreProcess 完成（图 24-20）。

图 24-20 预处理类

PreProcess 类只有两个方法，其中的 ClearAllSpace 方法先将表达式中的所有空格删除。接着在 CheckExprValidate 方法中对表达式进行检测。

这里有一个问题，CheckExprValidate 方法检测出的表达式的错误如何处理？很明显，一旦表达式有错，应中止表达式解析工作，同时通知用户改正。在给用户的通知信息中，出错信息应越具体越好。

第 6 章《异常捕获与处理》介绍过的异常处理知识现在可以派上用场了。

先设计一个自定义的异常类 CalculatorException，在预处理类 PreProcess 中设计一个 CheckExprValidate 方法负责检测表达式，若发现错误就抛出一个 CalculatorException 对象，并给这一异常对象附加一个有意义的信息。外界可捕获这一异常对象做进一步的处理。

下面是四则运算计算器“精简界面”窗口（frmExpr）中预处理的代码片段：

```
try
{
    PreProcess.CheckExprValidate(txtExpr.Text);
    //……
}
catch (CalculatorException ex)
{
      ShowError(ex.Message); //向用户提交有意义的信息
}
finally
{
    txtExpr.SelectAll();
    txtExpr.Focus();//焦点回到文本输入框
}
```

24.3.2 软件可扩展性

一个软件一编完不等于就是“万事大吉”了，在设计时还应该考虑到软件的日后扩充问题。

1 扩充对表达式错误的智能检测功能

笔者在编写预处理类 PreProcess 的 CheckExprValidate 方法时，首先列出了一些对合法四则运算表达式的基本判断准则，如：

- 不能出现连续的两个小数点；
- 字符串不能全由运算符组成，至少要有一个操作数；
- 括号必须配对；
- 小数点前必须为数字；
- 不能出现运算符、数字或小数点之外的非法字符；
- ……

然后再根据上述判断准则编程实现。

在本章示例项目中，所有的规则均由 CheckExprValidate 方法实现，这导致此方法拥有近百行的代码和很深的嵌套层次，不易于阅读和调试。请读者对这一方法进行重构。

提示：

读者可能会想出多种方式对 CheckExprValidate 方法进行重构，笔者建议的一个方法是新建立一个专门的类来完成对表达式错误的检测功能，将 CheckExprValidate 方法内部用到的变量转换为此类的私有字段，然后再把那些嵌套层次较多的 if 语句转换为一个个的私有方法，从而有效地简化代码，提升代码的可阅读性。

2 提供更有效的表达式求值算法

本章示例采用的表达式求值算法都是一些经典的算法，比如前序和中序表达式求值、表达式树遍历等算法，每本《数据结构》教材中几乎都有介绍。

但表达式解析算法远不止这三种，也许希望以后能以更有效的算法代替现有算法，因此在设计整个软件架构时，遵循一个基本原则：**将算法与界面“剥离”开**。为此引入了一个算法的抽象接口 IAlgorithm。

当引入新的算法时，只需新算法实现 IAlgorithm 接口，就可以无缝地“插入”到本软件中。

在目前情况下，当引入一个新的算法时，程序还需要重新编译。后面将要介绍使用 MEF 改造程序架构，通过将算法独立编译成 DLL 程序集，只需将新算法的 DLL 文件复制到计算器示例程序的文件夹下，不需要重新编译，计算器程序就可以直接使用新的算法。

24.3.3 设计用户界面

用户是通过软件的界面来使用软件的，界面设计的好坏直接影响到用户对软件质量好坏的评价。

本节以计算器界面设计的过程为例，介绍几个用户界面设计的基本原则。

1 一切以用户为中心

用户界面设计的最基本原则就是“一切以用户为中心”。在实际开发中程序员最容易犯的毛病就是“一切以自己为中心”，想当然地认为“我是这样用软件的，这很自然啊”，于是设计出来的软件只有他自己用得舒服。

另外，软件界面要努力设计成“不需要看使用说明书也可以使用”，为此，要采用用户所熟悉的软件操作方式，不要自己弄一套“有自己特色的界面与软件使用方法”。

在设计本章示例程序时，笔者就采用了标准的计算器界面，对于这个界面，大多数人一看就会用。

当用户输入错误的表达式时，示例程序给出的信息细化到了第几个字符位置。示例程序还采取了更为严格的输入限制——不允许用户直接在文本框中直接输入表达式，从而降低了表达式中出现非法字符的机率（图 24-21）。

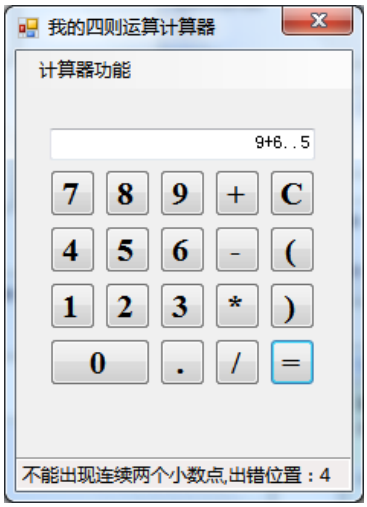

图 24-21 标准的计算器界面

这种对用户输入进行“限制”的手段是很重要的用户界面设计原则。图 24-22 所示为为 Windows 7 附件中的计算器程序，请注意当选择“八进制”之后，界面中的部分按钮就“灰”掉了。这也应用了对用户输入进行“限制”的界面设计方法。

图 24-22 Windows 计算器

在设计界面时，还应该尽可能地方便用户，以下是在设计程序交互特性时的考虑（参看图 24-23）。

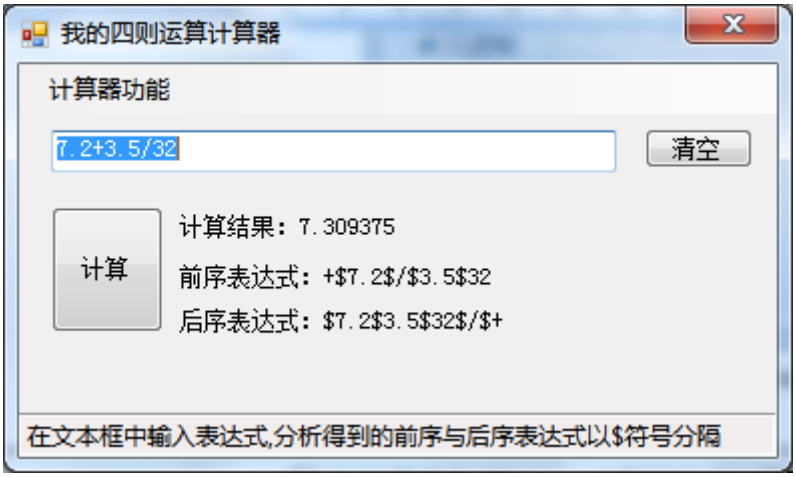

图 24-23 方便用户的设计

（1）在计算器的“精简型”界面中，用户在输入完表达式后，直接敲回车即可计算表达式，而不需要用户手动地去用鼠标点击“计算”按钮。

（2）在计算完毕后，自动地将输入文本框中的表达式全选，不再需要用户使用删除键删除原有的表达式就可以直接键入新的。同时，还在界面上增加了一个“清空”按钮。

提示：

有关用户界面设计的方法与原则，还有许多，均属于“软件人机交互设计”这一领域的研究内容，详细介绍超出了本书的范围，请感兴趣的读者自行探索。

2 使用“窗体嵌入”实现界面切换

计算器程序中有两个窗体（标准计算器窗体和精简表达式窗体），如何实现界面切换？

对于这类问题，有的读者可能想到创建两个窗体，在需要显示另一个窗体时，将自己隐藏，然后再显示另一个窗体。

这样做是可以的，却不是最优方案。这种方案主要存在着一个问题：如何处理两个窗体都有的公用元素，比如菜单？

由于菜单是依附于窗体的，因此，为保持界面的一致性，不得不在两个窗体中都设计同样的菜单，类似地，完成同样代码的功能也要写两份。

示例程序采用了 12.2 节《用对象组合实现可视化界面的嵌套》介绍的方法，将标准型界面与精简型界面分别放在两个窗体中，独立进行设计，然后设计一个“框架”窗体 frmMain，在程序运行时依据用户选择将标准型与精简型两个窗体之一嵌入到主框架窗体中，实现了界面的动态切换。

24.4 规划软件的开发流程

在整个软件架构设计完成之后，必须高度注意各个软件模块开发次序的确定。

24.4.1 各模块开发次序的确定

整个软件拥有 4 可视化的窗体和 14 个非可视化的类，应该先开发哪个后开发哪个？

如果读者是一个“老”Visual Basic 程序员，可能非常习惯先把窗体“画”出来再编码。但在本章所示的软件项目中，先不要去管窗体，而应先致力于实现整个程序的“灵魂”——算法。算法实现之后，再设计界面。

在设计阶段我们得到了 14 个与算法直接相关的类，该如何确定开发次序呢？

对于本项目而言，应该首先给数据结构相关的类编码。所以，第一步是先开发用于代表四则运算表达式对象的 MathExpression 类，此类开发完成，其他算法相关的类就有了可以操作的对象。

接着，开始实现算法相关的类。程序中计划实现 3 个算法：中序、前序和表达式树。其中中序算法是相对独立的，而前序与表达式树算法彼此之间有一定的关联。毛主席说过，**“打仗要先打弱敌孤敌”**，因此，先开发中序算法。

但仔细看看中序算法，会发觉在调用中序算法之前必须扫描字符串以提取数字，因此，要先开发实现数字提取功能的 OperandGetter 类，之后才可以实现中序算法。

有的读者可能会问：那对表达式进行预处理的 PreProcess 类为何不安排在第二步开发？道理很简单，做事要先选择最重要的完成。我们这个软件要完成的主要工作就是计算表达式，预处理工作只不过算是外围的工作，不应主次颠倒。在开发算法类时，只处理合乎规范的表达式，对非法表达式不予理会。

等到三个算法类开发完成，才进行预处理类 PreProcess 的开发。

在非可视化的类开发完成之后，最后才开始设计窗体，并基于已有的算法类“装配程序”。

整个开发过程可以规划出一个大致的流程（图 24-24）。

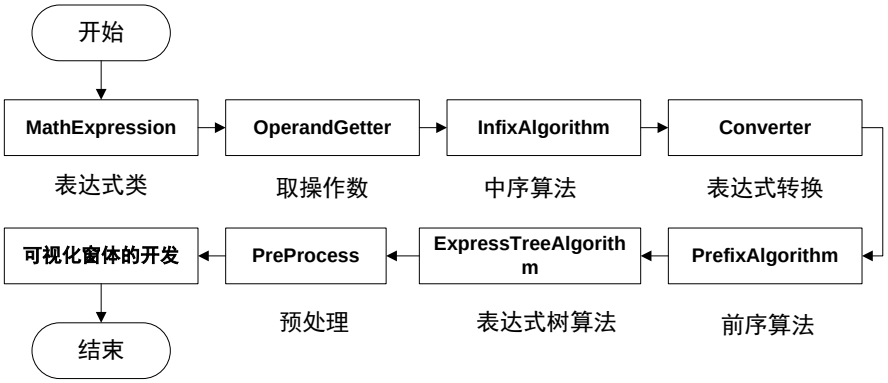

图 24-24 软件开发流程

需要指出的是，图 24-24 只是给出了一个大致的流程，其目的是对整个开发过程能有一个总体的控制，由于这个小软件是由笔者一个人“单枪匹马”完成的，因此采用了“单线程”的开发模式。真正的软件开发任务多由一个团队协作完成，因此，整个开发过程会复杂得多，是“多线程并行”的，其中还有许多的“同步点”和“里程碑”。

有关软件开发过程管理的理论可谓汗牛充栋，各种各样的具体实施框架和相关工具也是百花齐放，本书就不再赘述了，读者可以自行阅读相关技术资料。

24.4.2 实施迭代的软件开发方法

软件开发需要经历一个完整的软件开发周期：先设计出软件的总体技术方案，然后编写代码，接着由测试人员对此代码进行测试，发现的任何错误都将提交给原开发者进行更正，最后发布此系统（参看图 24-25）。

图 24-25 软件开发周期

一个复杂的软件系统开发过程并不是如图 24-25 所示一次循环就可以完成的，而是需要经历多次的循环迭代（图 24-26）。

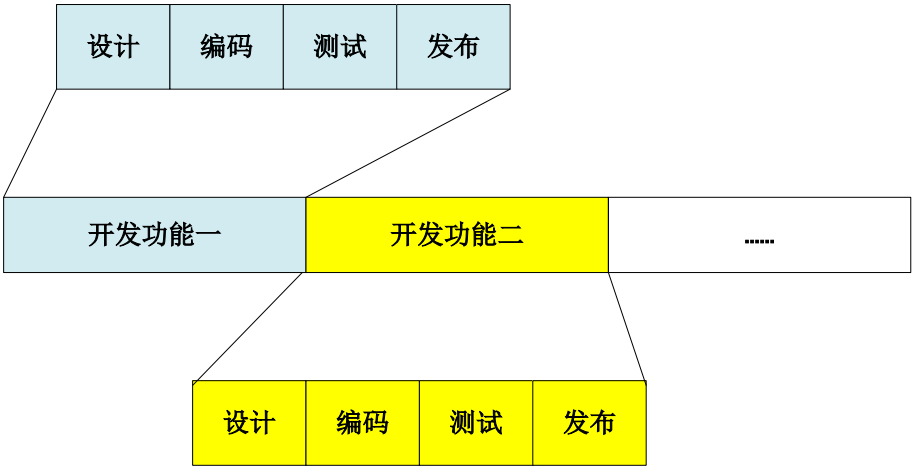

图 24-26 迭代开发

图 24-26 所示的软件开发方法可称为“**功能驱动的迭代开发**”，其特点是采用“蚂蚁啃骨头”的方法，每次迭代完成一个系统功能的开发。在开发每个系统功能的过程中，又经历一次完整的从设计到发布的软件开发周期。

在图 24-24 中给出的软件开发顺序图，就遵循了上述迭代开发的基本原则。

其实笔者在开发此程序时，还采用了“步步为营”的增量开发方法，每写一个类，就马上编写测试代码检测其是否工作正常[1]。

笔者希望每个程序员在参加正式软件项目开发时，都应同步地为自己的代码编写测试代码。这看似费事，其实是省事，编写测试代码所花费的时间将在对代码的排错与维护过程中成倍地节省回来。

24.5 示例程序的组件化重构

经过编码和测试，我们这个四则运算计算器开发完成了。这个小小的程序虽然功能不多，但毕竟开发它也花了不少时间与精力，自然就想到应该想法充分地挖掘它潜在的价值，今后如果在新项目中需要用到四则运算的，就可以直接复用过去的劳动成果而不需要“重新发明轮子”。

为了实现复用的目的，就需要对项目进行组件化重构。

24.5.1 一分钟实现示例的组件化重构

进行组件化重构的第一步，就是把用户界面与表达式解析代码给分离出来，由于我们在一开始设计时就考虑到了这个基本原则，因此，这一步事实上已经完成了。请看图 24-27，不难发现所有四则运算表达式解析的相关代码都放到了 Algorithms 和 ExpressTree 这两个文件夹下。

[1] 本书 21.5 节介绍了如何使用 Visual Studio 2010 对本示例程序进行单元测试。

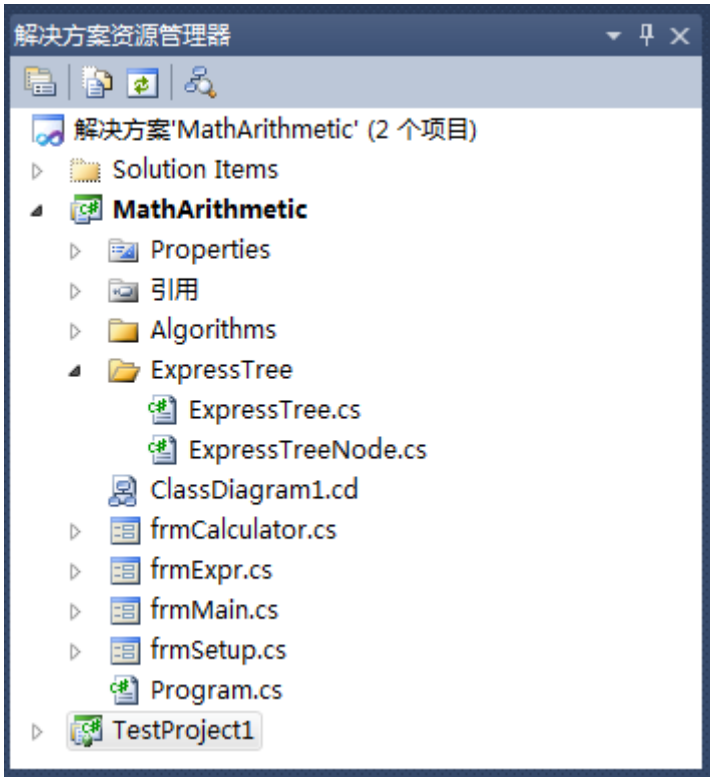

图 24-27 组件化前的四则运算器示例程序

所以，最简单的重构方法就是创建一个单独的类库项目（假设取名 MathFuncLib），将 Algorithms 和 ExpressTree 这两个文件夹下的所有文件移到此项目中，编译之后会得到一个程序集 MathFuncLib.dll，这时，原有的 MathArithmetic 项目中将只剩下几个窗体，给它添加对 MathFuncLib.dll 的引用之后，它已有的功能将不受任何影响。

通过“重构”，我们就将原有一个“包容所有功能”的单个程序集（MathArithmetic.exe）分割为两个程序集（图 24-28），其中 MathFuncLib.dll 包容了所有的四则运算表达式解析功能代码，而 MathArithmetic.exe 只包容用户界面，它在“后台”调用 MathFuncLib.dll 完成相应的表达式解析工作。

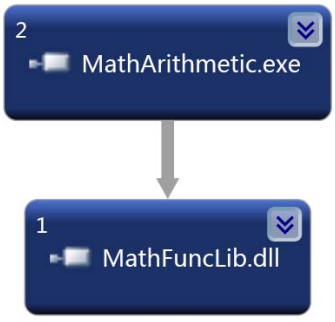

图 24-28 组件化重构后的系统架构

由于最初设计考虑得比较周到，所以，整个组件化过程所费时间不到一分钟。

对于我们这个小程序而言，组件化到这个程度其实就已经足够了，当我们在其他项目中需要四则运算表达式解析功能时，只需添加对 MathFuncLib.dll 的引用，此项目就可以从此

程序集所提供的中序、前序和表达式树三种算法中任选一个解析四则运算表达式。

如果需要“更酷”的界面，读者完全可以使用 WPF 取代 Windows Forms 设计一个全新的界面， 原有的 MathArithmetic.exe 程序集就可以“完全扔掉”了，因为里面只有 Windows Forms 窗体，并没什么复用价值。

下面我们再来点更有趣的尝试：

第 22 章不是介绍过 MEF 吗？为什么不尝试着将这个程序改造成支持插件的架构？

我们可以把各种表达式解析算法作成插件，由主程序在启动时动态装载和组合这些插件，从而实现这样的目标：

当有更新更好更快的四则运算表达式解析算法出现时，只需编写一个插件，我们的四则运算计算器无需做任何修改，就可以直接使用这个新算法解析四则运算表达式。

24.5.2 使用MEF将示例转换为插件架构

在最初设计四则运算计算器时，已经将各算法的本质特征抽象出来定义了一个 IAlgorithm 接口，因此，只需让新算法实现这一接口，就可以无缝地融入到已有的程序中。

我们前面实现了前序、中序和表达式树三个算法，很自然地想到可以设计三个插件，每插件实现一种算法。然而很遗憾，在本例中由于各算法相关类型间的依赖关系，我们做不到将这三个算法划分为三个插件，因为各插件之间是不应该有依赖关系的。

我们怎么知道应当将现有项目划分为几个插件？

这需要分析各个类之间的依赖关系入手。很幸运，Visual Studio 2010 给我们提供了强大的工具，让这一工作很容易完成。

请读者打开“原始版”的未组件化的 MathArithmetic 项目，从“体系结构(Architecture)”菜单中选取“生成依赖项关系图（Generate Dependency Graph)”→ “按类（By Class)”命令，将会看到整个解决方案中各类型之间的依赖关系（图 24-29)。

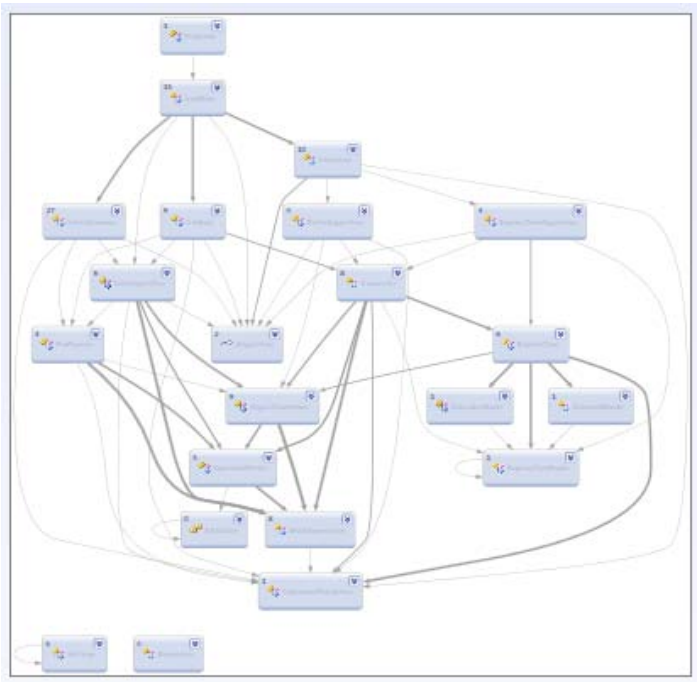

图 24-29 使用 Visual Studio 生成类型依赖图

提示：

图 24-29 由于显示了太多的类型，因此看不清楚。不过不要紧，Visual Studio 2010 使用WPF开发的，因此具有良好的缩放图形能力，只需压住Ctrl键，再滚动鼠标中部的“滚轮”即可[1]，往前滚放大，往后滚缩小。

为了清晰起见，请从类型依赖图中删除与插件化无关的类（比如几个窗体和 Settings、Resources 等），其方法很简单，鼠标点击选中后按 Del 键。

现在仔细看看 CalculatorException 类，会发现它的调用箭头“只入不出”，而且现有的三个算法类都依赖于它（图 24-30）。

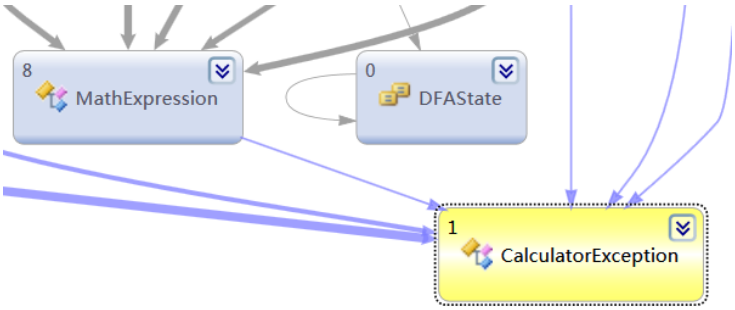

图 24-30 “只入不出”的 CalculatorException 类

这就提示我们，应该将所有算法都用到的类型放置到一个独立的程序集中，由此程序集为所有的算法插件提供支撑。我们给此程序集取名 AlgorithmFoundation。

AlgorithmFoundation 这一程序集中到底应该包容哪些类型？

很明显，诸如 IAlgorithm、MathExpression、CalculatorException、PreProcess 等类型应该放到 AlgorithmFoundation 程序集里面，因为它们的功能非常明确，而且都是应用程序调用四则运算功能所必须用到的。作出这些决定之后，将这些已经确定归属的类型从类型依赖图中移除，再从中选择一个类型，跟踪它的依赖关系，将依赖关系紧密无法分割的几个类型归作一组，作出是否将它们作为一个整体加入到 AlgorithmFoundation 程序集中的决策，之后再将它们从图中移除，由此不断进行下去，慢慢地，原先“犬牙交错”的“乱麻”一般的类型依赖图不见了，所有的类型都找到了它应该“呆”的地方（即程序集），至此我们的重构工作完成。

仔细分析我们的小小计算器中的三个算法类，不难发现前序算法类 PrefixAlgorithm、表达式算法类 ExpressTreeAlgorithm 和其他类型之间的依赖关系非常强，因此无法将这两个算法独立地抽取为插件，只能将它们一起并入到 AlgorithmFoundation 程序集中。

另一个中序算法 InfixAlgorithm 则独立性较强，可以独立抽取为插件。

图 24-31 展示了示例程序插件化的结果。

[1] 如果读者的鼠标上没有“滚轮”，可以在“定向关系图（Directed Graph）”工具栏上找到缩放显示当前图形的按钮。

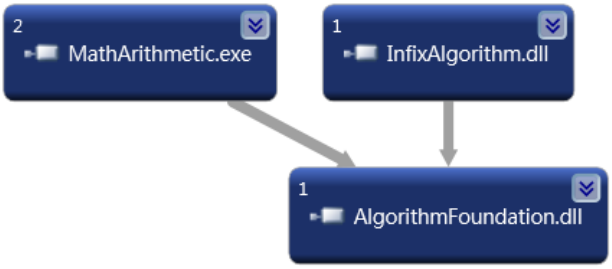

图 24-31 示例程序插件化的结果

如图 24-31 所示，InfixAlgorithm.dll 将作为独立的插件而存在，MathArithmetic.exe 现在成为了插件宿主程序，它只引用 AlgorithmFoundation.dll 程序集，与插件间没有任何的依赖关系。

下面我们简要介绍一下插件版四则运算器示例程序中的关键代码。

很明显，三个算法类都必须定义为导出的部件，以前序算法为例：

```
[Export(typeof(IAlgorithm))]
public class PrefixAlgorithm : IAlgorithm
{ …… }
```

在宿主程序 MathArithmetic 项目中，添加一个 AlgorithmPlugIns 类作为集中保存插件的“仓库”，其 FindAllAlgorithmObjs 方法完成部件的装配工作：

```
public class AlgorithmPlugIns
{
    [ImportMany(typeof(IAlgorithm))]
    public List<IAlgorithm> AlgorithmObjs; //保存所有的插件

    // 查找所有的插件,并将其组合到AlgorithmObjs集合中
    public void FindAllAlgorithmObjs()
    {
        var dirCatalog = new
            DirectoryCatalog(Environment.CurrentDirectory);
        var assCatalog = new
            AssemblyCatalog(Assembly.GetExecutingAssembly());
        var aggCatalog = new AggregateCatalog(
            dirCatalog, assCatalog);
        var container = new CompositionContainer(aggCatalog);
        container.ComposeParts(this);
    }
}
```

为了能让宿主程序中的所有窗体都能访问到算法对象，在 Program 中定义了一个静态的 AlgorithmPlugIns 对象，然后在 Main 方法中完成扫描装载算法插件的工作：

```
static class Program
{
    public static AlgorithmPlugIns AlgorithmObjects =
        new AlgorithmPlugIns();

    [STAThread]
    static void Main()
    {
        Application.EnableVisualStyles();
        Application.SetCompatibleTextRenderingDefault(false);
        AlgorithmObjects.FindAllAlgorithmObjs(); //扫描装载算法插件
        Application.Run(new frmMain());
    }
}
```

需要对原有的各窗体代码做少量的修改，让 MathArithmetic 项目中的所有代码都只针对 IAlgorithm 接口进行编程，并移除所有使用具体算法类型的代码。

另外，还需要修改算法选择窗体，将原先固定的使用 RadioButton 表示可获取算法的界面改换为可方便地动态增减项目的 ListBox（图 24-31）。

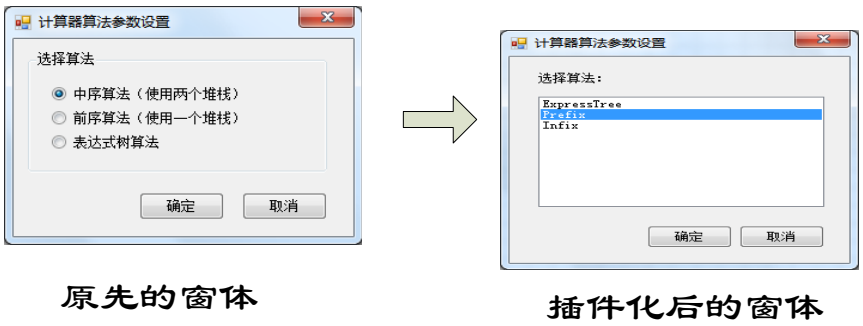

图 24-32 插件化的程序其界面必须是可动态调整选项的

读者可以试一试，在 MathArithmetic.exe 所在的文件夹中删除 InfixAlgorithm.dll，则图 24-32 所示的可选算法中将没有 Infix 这一项。

请读者自行阅读示例源码，体会一下“插件化”后示例程序的特点。

24.6 在 ASP.NET 网站中重用四则运算组件

将四则运算解析功能组件化之后，可以非常方便地在其他项目中重用。图 24-33 所示为一个 ASP.NET 网站，可以看到，我们将 24.5 节得到的 MathFuncLib.dll 程序集复制到网站的 bin 文件夹下，整个网站就可以使用四则运算解析功能了。

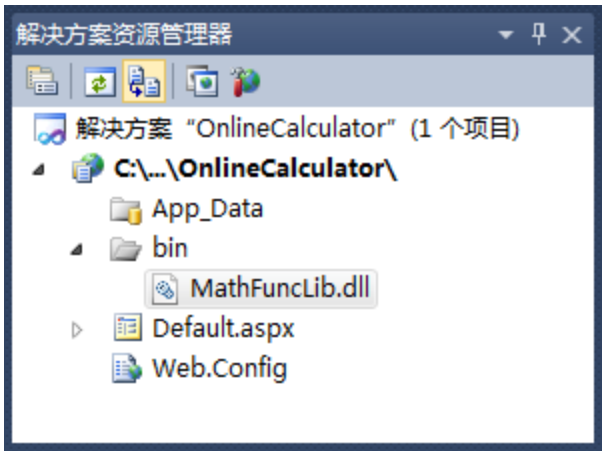

图 24-33　ASP.NET 网站中重用四则运算组件

图 24-34 所示为示例网站 OnlineCalculator 的运行截图。其代码非常简单，就是直接创建一个 InfixAlgorithm 对象完成计算工作，不再赘述。

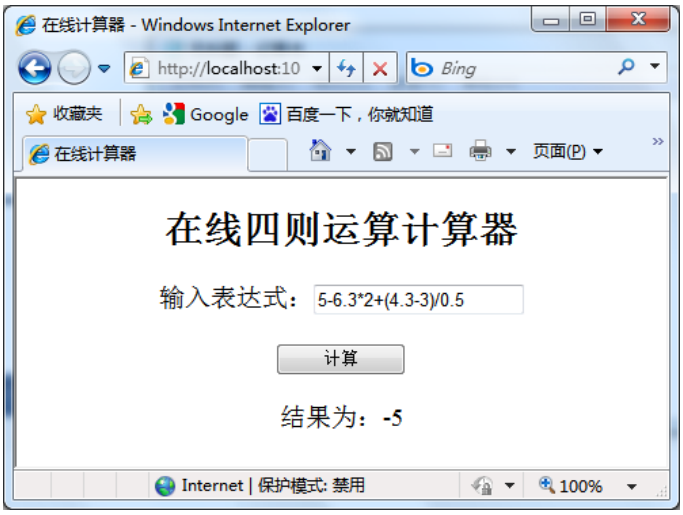

图 24-34 Web 上的四则运算计算器

24.7 无所不在的“四则运算”服务

通过将我们这个小小的四则运算计算器组件化，我们已经可以在各个项目中方便地重用代码，但这还是有局限的，必须提供程序集才能实现重用。“**SOA（Service-Oriented Architecture，面向服务的架构）**”则提出了一种新的开发方式，将“基于组件搭积木”的软件系统构造方式，转换为“选择服务来聚合”的方式，当聚合一个服务时，并不需要理会服务的内部技术细节，而只需要知道以下事实就够了：

（1）这个服务是干什么的？

（2）如果想使用这个服务，需要提供哪些信息？

（3）这个服务以哪种形式返回需要的结果？

只需拥有一个包容上述信息的服务描述，一个应用程序就可以“聚合”这个服务所提供

的功能。

在本节中，我们把四则运算的功能转换为一个“服务”，并将其发布在网络上，从而允许其他的应用程序通过网络来“聚合”这一服务。

使用 Visual Studio 创建一个 WCF 服务应用程序（图 24-35）。

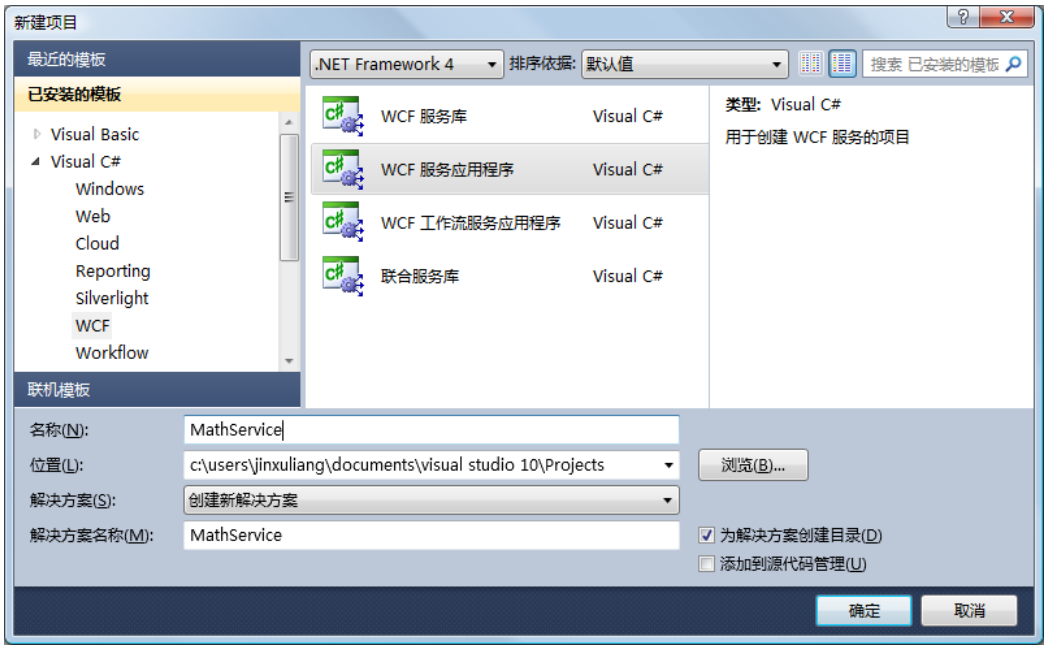

图 24-35 创建一个 WCF 服务应用程序

删除 Visual Studio 自动创建的 Service1.svc 等文件以得到一个“空的”WCF 服务应用程序，然后，向项目中添加一个名为“MyCalculatorService”的 WCF 服务（图 24-36）。

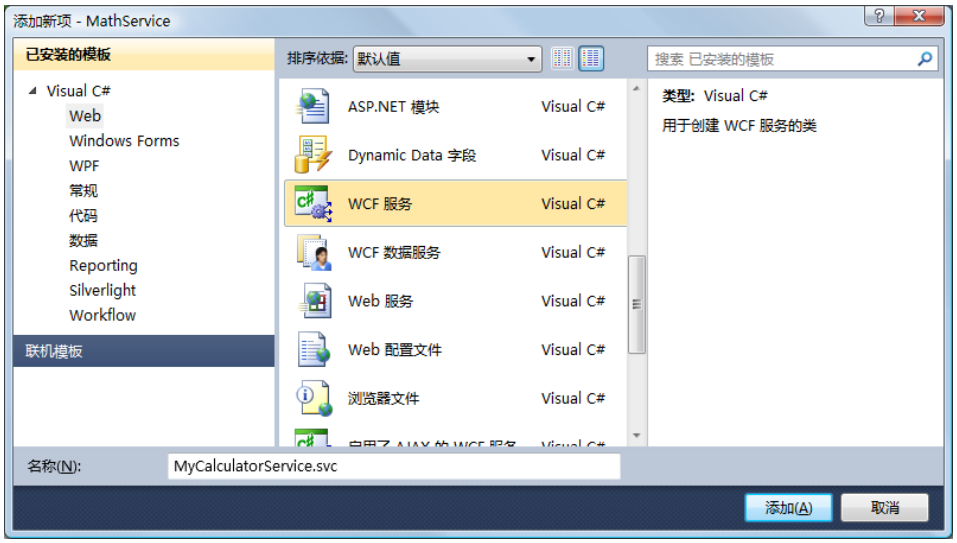

图 24-36 向项目中添加一个 WCF 服务

在 Visual Studio 自动生成的 IMyCalculatorService.cs 文件中修改服务协定如下：

```
[ServiceContract]
```

```
public interface IMyCalculatorService
{
    [OperationContract]
    double Calculator(string Expr);
}
```

给项目添加对四则运算组件 MathFuncLib.dll 的引用，然后在 Visual Studio 自动生成的 MyCalculatorService.svc.cs 文件中修改代码，调用 MathFuncLib.dll 组件完成四则运算解析功能：

```
public class MyCalculatorService : IMyCalculatorService
{
    public double Calculator(string expression)
    {
        //四则运算表达式预处理
        PreProcess.CheckExprValidate(expression);
        //创建中序算法对象
        InfixAlgorithm obj = new InfixAlgorithm();
        //计算表达式，返回结果
        return obj.Calculate(expression);
    }
}
```

现在可以测试此 WCF 服务了，在解决方案资源管理器中选中 MyCalculatorService.svc 文件，右击，从弹出菜单中选“在浏览器中查看（View in Browser）”命令，可以看到到 WCF 服务的测试页（图 24-37）。

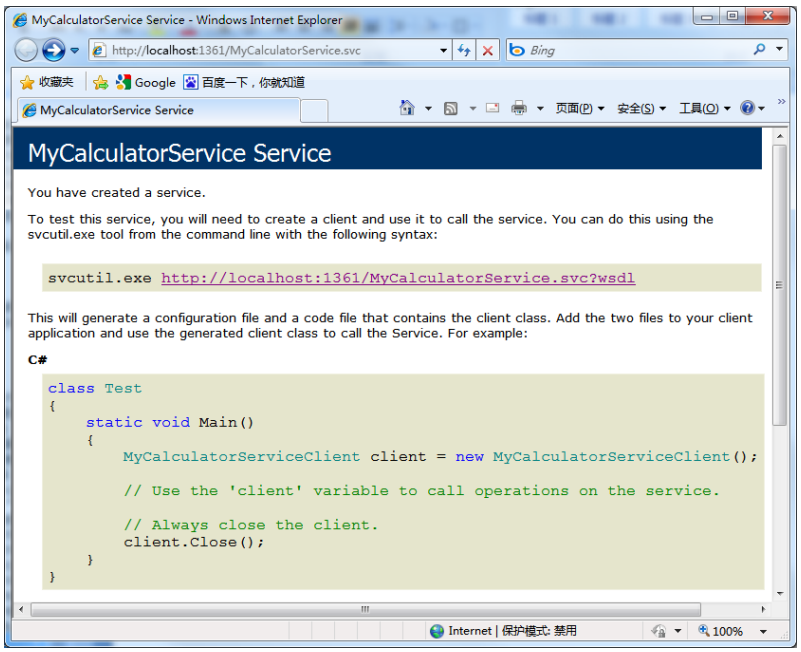

图 24-37 在浏览器中访问 WCF 服务

点击图 24-37 中的链接，可以看到以 XML 格式给出的 WCF 服务描述信息（图 24-38）。

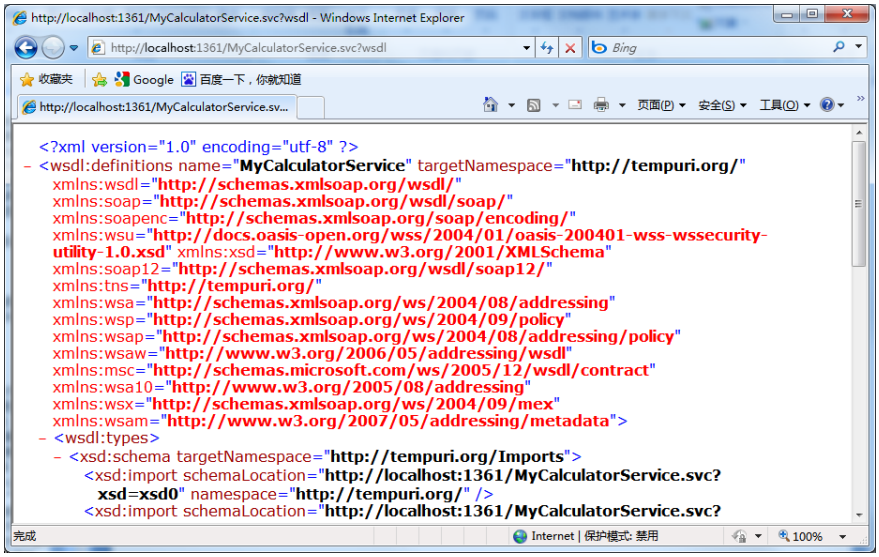

图 24-38 WCF 服务描述信息

Visual Studio 可以依据图 24-38 所给出的 WCF 服务描述信息为每一个需要“聚合”此服务的.NET 应用程序生成类型化客户端代理。本节我们以一个 Windows Forms 项目 UseMathWebService 为例，给此项目添加对 WCF 服务的引用（图 24-39）。

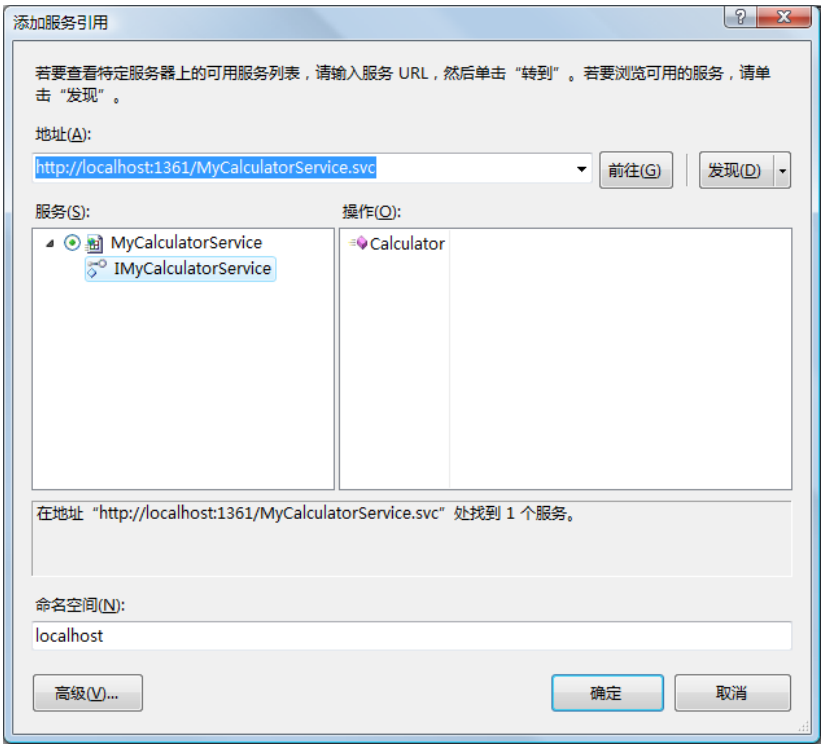

图 24-39 创建用于聚合服务的代理对象

注意：

WCF 服务的地址就是前面查看 WCF 服务描述信息的地址，在本例中为

```
http://localhost:1361/MyCalculatorService.svc
```

正式项目中，上述地址应当是服务发布到互联网上之后的那个地址。

当为一个.NET 应用程序添加 WCF 服务引用时，Visual Studio 会将用于访问远程服务的代码封闭到一个名字以“Client”结尾的类中，我们通常这个类称为“**WCF 类型化客户端**”，使用这个类，调用 WCF 服务的代码变得非常简单：

```
MyCalculatorServiceClient client = new MyCalculatorServiceClient();
double result = client.Calculator("一个有效的四则运算表达式");
```

示例项目 UseMathWebService 就是使用这一方法访问 WCF 服务的（图 24-40）。

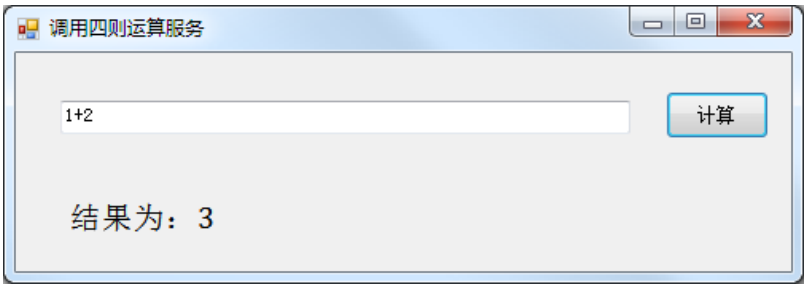

图 24-40 Windows Forms 程序聚合 WCF 四则运算服务

注意：

客户端程序的正确运行依赖于服务端程序是否已经运行。如果读者是在 Visual Studio 中调试示例程序的，请注意在 Windows 任务栏上是否有一个相应的 ASP.NET Development Server 小图标在显示（图 24-41）。

图 24-41 承载 WCF 服务的 ASP.NET 宿主必须启动

如果没有此图标，则应该按照前述查看 WCF 服务描述信息的方法，在 Visual Studio 中启动浏览器访问一次 WCF 服务的测试网页。

如果 WCF 服务已经发布到 IIS 上，只要 IIS 在运行，而且客户端添加服务引用时给出的服务地址是正确的（也可以直接在客户端项目的 app.config 中修改），客户端就能正确地调用 WCF 服务。

为了能向用户提示表达式错误信息，我们需要将 WCF 服务端对表达式的解析错误信息发回给客户端，最正式的用法是给 WCF 通讯协定设定一个故障协定：

```
[ServiceContract]
public interface IMyCalculatorService
{
    [OperationContract]
    [FaultContract(typeof(异常对象类型名称))]
    double Calculator(string Expr);
}
```

这样一来，在客户端生成代理后，就可以使用标准的 try…catch 结构捕获“服务端”的异常了。

```
try
{
    //使用类型化客户端访问远程WCF服务……
}
catch (FaultException<异常对象类型名称> ex)
{
    //编写代码处理服务端引发的异常……
}
```

但在本例中，有更简单的作法，只需对 WCF 服务应用程序项目的 Web.config 略作修改，设置 includeExceptionDetailInFaults="true"即可：

```
<serviceDebug includeExceptionDetailInFaults="true"/>
```

现在，客户端程序就拥有了“报告错误”的能力（图 24-42）。

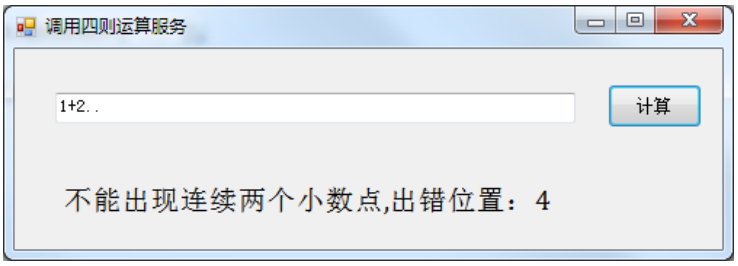

图 24-42 将服务端的错误“报告”给客户端用户

将“组合组件”的软件开发方式升级为“聚合服务”的新方式，拥有许多好处，比如客户端无需了解服务端的技术实现细节，任何一个可以解析服务描述信息（通常都是 XML 格式的数据）的程序都能聚合这一服务，从而使得客户端与服务端不再“绑定”在一起，比如完全可以使用 Java 编写一个聚合了 WCF 服务的客户端。

“聚合服务”并不能完全取代当前已经非常成熟的组件化开发方式，但可以肯定的是，我们一定会看到越来越多的以“服务”为作基础元素构造而成的软件系统，它是当前软件技术的一个重要发展方向。