第四章　C++面向对象程序设计方法概述

会用 C++的程序员一定懂得面向对象程序设计吗？

不会用 C++的程序员一定不懂得面向对象程序设计吗？

两者都未必。

我曾经和很多 C++程序员一样，在享用到 C++语法的好处时便以为自己已经明白了面向对象程序设计方法。我就这样糊里糊涂地编写了十几万行 C++程序，如此使用 C++，就象挤掉牙膏卖牙膏皮那样，真是暴殄天物呀。

本章目的不是阐述面向对象的理论，而是用浅显的示例来解说面向对象程序设计的一些重要概念，如封装、继承、组合、虚函数、抽象基类、动态联编、多态性等。本章对本书的后面十章有指导意义。

4.1 漫谈面向对象

在第一次世界大战结束前夕，一个叫做路德维希·维特根斯坦的维也纳人，在意大利的战俘营里写了一本《逻辑哲学论》。这本 75 页的小册子提出了对象的观点：

世界可以分解为事实；

事实是由原子事实组成的；

一个原子事实是多个对象的组合；

对象是简单的；

对象形成了世界的基础。

五十年之后，面向对象（Object-Oriented, OO）方法论火起来了，现在“对象”真的成为了软件世界的基础。

面向对象分析设计（OOAD）方法兴起于 20 世纪 80 年代，从 90 年代起至今它已经在分析设计领域占据了无可争议的主流地位。

我在读本科（90 年至 94 年）时就充分地感受到了人们对“面向对象”的狂热。关于“面向对象”的课堂、学术报告会常常人满为患。搞软件研发的人都“言必谈对象”，并引以为荣。

面向对象分析设计领域有一些比较著名的学派，如：

- ✧ Coad 和 Yourdon 学派
- ✧ Booch 学派
- ✧ Jocobson 学派
- ✧ Rumbaugh 学派

有趣的是，这些学派的掌门人就像上帝、真主、如来佛，他们用各自的方式定义了这个世界，并留下一堆经书来解释这个世界。这种混乱的局面被学术界称为百家争鸣，每年诞生了许多论著和教授。叫苦的是软件企业和开发人员：没有统一的方法，不好干活啊！

终于等到了那一天，Rational 公司招纳了 Booch, Jocobson, Rumbaugh，这三位“面向对象”业界的老大强强联手，制定了“统一建模语言”（UML）。1997 年 11 月，UML 被国际对象管理组织（OMG）采纳，此后 UML 成为 OOAD 建模语言的国际标准。

有趣的是，面向对象编程语言比 OOAD 方法论更早地问世。最早的面向对象编程语言是 Smalltalk，由施乐公司研究中心于 1970 年研制。在软件开发领域，编程实践往往先行于相应的理论。就如人类的进化：先学会讲话，后来才产生文字。用程序员的行话讲，这叫“编程是硬道理”。

六七年前，我刚“热恋”面向对象时，急切地想知道什么是面向对象，于是买了一堆书来阅读。

不少书籍建议这样找“对象”：分析一个句子的语法，找出名词和动词，名词就是对象，动词则是对象的方法（即函数）。

天哪，这不是程序员的做法！我除了发现自己有些“弱智”之外别无收获。

当年国民党的文人为了对抗毛泽东的《沁园春·雪》，特意请清朝遗老们写了一些对仗工整的诗，请蒋介石过目。老蒋看了气得大骂：“娘希匹，全都有一股棺材里腐尸的气味。”

不好意思，我初读面向对象理论书籍的感觉与老蒋的有点相似。

现在我有些心得体会了，我建议程序员应当先学习用 C++或者 Java 编写程序，当他们对面向对象程序设计有了感性认识之后，再阅读面向对象理论书籍，这样才能深入理解面向对象方法。

面向对象编程语言很多，如 Smalltalk、Ada、Eiffel、C++、Java 等等。C++语言最受程序员喜欢，因为它兼容 C 语言，所以应用最广泛。Java 是一种纯面向对象语言，它诞生之初曾红极一时，不少人叫喊着要用 Java 革 C++的命。我认为 Java 好比是 C++的外甥，虽然不是直接遗传的，但也有几分象样。外甥在舅舅身上玩耍时洒了一泡尿，俩人不该为此而争吵。

4.2 信息隐藏与类的封装特性

在一节不和谐的课堂里，老师叹气道：“要是坐在后排聊天的同学能象中间打牌的同学那么安静的话，就不会影响到前排睡觉的同学了。”

这个故事告诉我们，如果不想让坏事传播开来，就应该把坏事隐藏起来，“家丑不可外扬”就是这个道理。

对于软件设计而言，为了尽量避免某个模块的行为干扰同一系统中的其它模块，应该让模块仅仅公开必须要让外界知道的内容，而隐藏其它一切内容。

“信息隐藏”这种设计理念产生了 C++类（Class）的封装特性。

类可以将数据和函数封装在一起，其中函数表示了类的行为（或称服务）。类提供关键字 public、protected 和 private 用于声明哪些数据和函数是公有的、受保护的或者是私有的。这样可以达到信息隐藏的目的，即让类仅仅公开必须要让外界知道的内容，而隐藏其它一切内容。例如：

```
class WhoAmI
```

```
{
    public:
        void GetMyName(void);  // 名字是可以公开的

    protected:
        void GetMyAsset(void);  // 财产是受保护的，只有我和继承者可以使用

    private:
        void GetMyGuilty(void);  // 罪过是要保密的，只有我自己才能偷看
    …
};
```

类的封装特性是 C++的基本语法之一，易学易用。要注意的是，我们不可以滥用类的封装特性，不要把毫不相干的数据和函数封装到类里头，不要把类当成火锅，什么东西都往里扔。

4.3 类的继承特性

对象是类的一个实例（Instance）。如果将对象比作房子，那么类就是房子的设计图纸。所以面向对象设计的重点是类的设计，而不是对象的设计。对于 C++程序而言，设计孤立的类是比较容易的，难的是正确设计基类及其派生类。

如果 A 是基类，B 是 A 的派生类，那么 B 将继承 A 的数据和函数。例如：

```
class A
{
    public:
        void  Func1(void);
        void  Func2(void);
};

class B : public A
{
    public:
        void  Func3(void);
        void  Func4(void);
};

main()
{
    B  b;
```

```
    b.Func1();      // B 从 A 继承了函数 Func1
    b.Func2();      // B 从 A 继承了函数 Func2
    b.Func3();
    b.Func4();
}
```

这个简单的示例程序说明了这样一个事实：C++的“继承”特性可以提高程序的可复用性。正因为“继承”太有用、太容易用，才要防止乱用“继承”。我们应当给“继承”立一些使用规则：

- **【规则 4-3-1】 如果类 A 和类 B 毫不相关，不可以为了使 B 的功能更多些而让 B 继承 A 的功能和属性。**不要觉得“白吃白不吃”，让一个好端端的健壮青年无缘无故地吃人参补身体。

- **【规则 4-3-2】 若在逻辑上 B 是 A 的“一种”（a kind of ），则允许 B 继承 A 的功能和属性。**例如男人（Man）是人（Human）的一种，男孩（Boy）是男人的一种。那么类 Man 可以从类 Human 派生，类 Boy 可以从类 Man 派生。

```
class Human                 // Human 是基类
{
    …
};

class Man : public Human    // Man 是 Human 的派生类
{
    …
};

class Boy : public Man      // Boy 是 Man 的派生类
{
    …
};
```

◆ **注意事项**

【规则 4-3-2】看起来很简单，但是实际应用时可能会有意外，继承的概念在程序世界与现实世界并不完全相同。

例如从生物学角度讲，鸵鸟（Ostrich）是鸟（Bird）的一种，按理说类 Ostrich 应该可以从类 Bird 派生。但是鸵鸟不能飞，那么 Ostrich::Fly 是什么东西？

```
class Bird
{
    public:
        virtual void Fly(void);    // 鸟能飞行
```

```
    …
};

class Ostrich : public Bird       // 鸵鸟是鸟的一种
{
    public:
        virtual void Fly(void);    // 如何让鸵鸟飞起来？
    …
};
```

再例如，从数学角度讲，圆（Circle）是一种特殊的椭圆（Ellipse），按理说类 Circle 应该可以从类 Ellipse 派生。但是椭圆有长轴和短轴之分，如果圆继承了椭圆的长轴和短轴，岂非画蛇添足？

所以更加严格的继承规则应当是：

- **【规则 4-3-3】 若在逻辑上 B 是 A 的“一种”，并且 A 的所有功能和属性对 B 而言都有意义，则允许 B 继承 A 的功能和属性。**

4.4 类的组合特性

组合（Composition）用于表示类的“整体/部分”关系。例如主机、显示器、键盘、鼠标组合成一台计算机。继承则表示类的“一般/特殊”关系。继承与组合显然不是相似的概念，但奇怪的是，程序员经常在编程时把继承与组合混为一谈。

- **【规则 4-4-1】**若在逻辑上 A 是 B 的“一部分”（a part of），则不允许 B 从 A 派生，而是要用 A 和其它东西组合出 B。

例如眼（Eye）、鼻（Nose）、口（Mouth）、耳（Ear）是头（Head）的一部分，所以类 Head 应该由类 Eye、Nose、Mouth、Ear 组合而成，不是派生而成。

```
class Eye
{
    public:
        void  Look(void);
};

class Nose
{
    public:
        void  Smell(void);
};

class Mouth
```

```
{
    public:
        void  Eat(void);
};

class Ear
{
    public:
        void  Listen(void);
};

// 正确的设计，虽然代码冗长。
class Head
{
    public:
        void    Look(void)      {  m_eye.Look();  }
        void    Smell(void)     {  m_nose.Smell(); }
        void    Eat(void)       {  m_mouth.Eat();  }
        void    Listen(void)    {  m_ear.Listen(); }
     private:
        Eye     m_eye;
        Nose    m_nose;
        Mouth   m_mouth;
        Ear     m_ear;
};
```

如果允许 Head 从 Eye、Nose、Mouth、Ear 派生而成，那么 Head 将自动具有 Look、Smell、Eat、Listen 这些功能。程序如下：

```
class Head : public Eye, public Nose, public Mouth, public Ear
{
};
```

采用继承方法来实现的 Head 程序十分简短并且运行正确，但是这种设计方法却是不对的。

一只公鸡使劲地追打一只刚下了蛋的母鸡，你知道为什么吗？

因为母鸡下了鸭蛋。

许多刚刚接触 C++的程序员恨不得在所有的地方都使用继承，然后得意洋洋的宣称已经充分利用了面向对象的好处。“运行正确”的程序不见得是高质量的程序，此处就是一个例证。

4.5 动态特性

在绝大多数情况下，程序的功能是在编译的时候就确定下来了，我们称之为静态特性。反之，如果程序的功能是在运行时刻才确定下来的，称之为动态特性。

动态特性是面向对象语言最强大的功能之一，因为它在语言级别上支持程序的可扩展性，而可扩展性则是软件设计追求的重要目标之一。

C++的虚函数、抽象基类、动态联编和多态性（Polymorphism）构成了出色的动态特性。

4.5.1 虚函数

假定几何形状的基类为 Shape，其派生类有 Circle、Rectangle、Ellipse 等，每个派生类都能够用绘制自己的形状。不管派生类的形状如何，我们希望用统一的方式来调用绘制函数，最好是使用 Shape 定义的函数接口 Draw，并让程序在运行时动态地确定应该使用那个派生类的 Draw 函数。

为了使这种行为可行，我们把基类 Shape 中的函数 Draw 声明为虚函数，然后在派生类中重新定义 Draw 使之绘制正确的形状。虚函数的声明方法是在基类的函数原型之前加上关键词 virtual。

一旦类的一个函数被声明为虚函数，那么其派生类的对应函数也成为虚函数。虽然函数在类层次结构的高层中声明为虚函数将会使它在底层自动（隐式）地成为虚函数，但是为了提高程序的清晰性，建议在每一层中将它显式地声明为虚函数（即加 virtual）。

例如：

```
class Shape
{
    public:
        virtual void Draw(void);  // Draw 为虚函数
};

class Rectangle : public Shape
{
    public:
        virtual void Draw(void);  // Draw 为虚函数
    …
}
```

4.5.2 抽象基类

当我们把类看作是一种数据类型时，通常会认为该类肯定是要被实例化为对象的。但是在很多情况下，定义那些不被实例化为对象的类是很有用的，这种类称为抽象类（Abstract Class）。能够被实例化为对象的类称为具体类（Concrete Class）。抽象类的唯

一目的就是让其派生类继承它的函数接口，因此它通常也被称为抽象基类（Abstract Base Class）。[Deitel, p175]

如果将基类的虚函数声明为纯虚函数，那么该类就成为抽象基类。纯虚函数是在声明时其“初始化值”为 0 的函数，例如：

```
class Shape     // Shape 是抽象基类
{
    public:
        virtual void Draw(void)=0; // Draw 为纯虚函数
};
```

抽象基类 Shape 的纯虚函数 Draw 根本不知道它自己能干什么，具体功能必须由派生类的 Draw 函数来实现。

很多良好的面向对象系统中，其类层次结构的顶部通常是抽象基类，甚至可以有好几层的抽象类。例如几何形状的类结构可分三层（如图 4-1 所示），顶层是抽象基类 Shape，第二层也是抽象类 Shape2D 和 Shape3D，在第三层才是可以被实例化为对象的具体类，如二维形状类 Circle、Rectangle 和 Ellipse，三维形状类 Cube、Cylinder 和 Sphere。

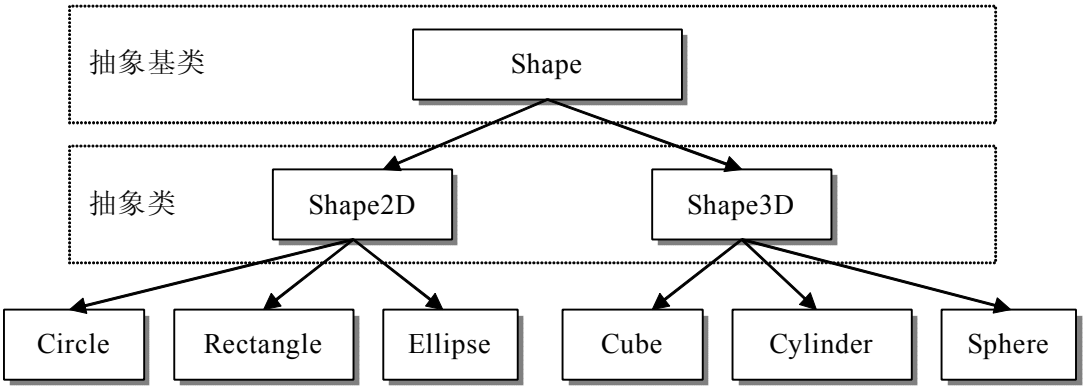

图 4-1 几何形状的类结构

4.5.3 动态联编

如果将基类 Shape 的函数 Draw 声明为 virtual，然后用指向派生类对象的基类指针调用 Draw，那么程序会动态地（即在运行时）选择该派生类的 Draw 函数，这种特性称为动态联编。例如：

```
Shape  *aShape;
Circle  aCircle;
Cube    aCube;
Sphere  aSphere;

aShape = &aCircle;
aShape->Draw();    // 绘制一个 circle
```

```
aShape = &aCube;
aShape->Draw();     // 绘制一个 cube

aShape = &aSphere;
aShape->Draw();     // 绘制一个 Shpere
```

动态联编可以使独立软件供应商（ISV）在不透露技术秘密的情况下发行软件包，即只发行头文件和二进制目标码，不必公开源代码。软件开发者可以利用继承机制从 ISV 提供的类库中派生出新的类。和 ISV 类库一起运行的软件也能够和新的派生类一起运行，并且能够通过动态联编使用新派生类的虚函数。

4.5.4 多态性

当许多派生类因为继承了共同的基类而发生关系时，每一个派生类的对象都可以被当成基类的对象来使用，这些派生类对象能对同一函数调用作出不同的反应，这就是多态性。多态性是通过虚函数和动态联编实现的。例如：

```
void Draw(Shape *aShape)   // 多态函数
{
    aShape->Draw();
}

main()
{
    Circle  aCircle;
    Cube    aCube;
    Sphere  aSphere;

    Draw(&aCircle);     // 绘制一个 circle
    Draw(&aCube);       // 绘制一个 cube
    Draw(&aSphere);     // 绘制一个 Sphere
}
```

综合 C++的“虚函数”和“多态”，有如下突出优点：

- ✧ 应用程序不必为每一个派生类编写功能调用，只需要对基类的虚函数进行处理即可。这一招叫“以不变应万变”，可以大大提高程序的可复用性和可扩展性。
- ✧ 派生类的功能可以被基类指针引用，这叫向后兼容。以前写的程序可以被将来写的程序调用不足为奇，但是将来写的程序可以被以前写的程序调用那可了不起，这正是动态特性的妙处。

4.6 小结

C++是应用最广泛的面向对象编程语言，在作者心目中C++/C是程序员的正宗语言。学好C++/C后，再学习其它编程语言如Visual Basic、Java就非常容易。面向对象不会是编程语言的终点。我们现在不知道OO之后的“XO”是什么东西，但至少可以推知，“XO”的核心概念必然高于并包容对象这一概念。正如对象高于并包容了函数和变量一样。

C++/C 程序设计如同少林寺的武功一样博大精深，作者练了十年，大概只有五成功力。所以无论什么时候，都不要觉得自己的编程水平很高，要虚心学习。

如果你会编写 C++/C 程序，不要因此得意洋洋，这只是程序员的基本技能而已。如果把系统分析和系统设计比作“战略和战术”，那么编程充其量只是“格斗技能”。如果指挥官是个大笨蛋，士兵再勇敢也会打败仗的。所以程序员不要只把眼光盯在程序上，要让自己博学多才。我们应该向北京胡同里的小孩们学习，他们小小年纪就能指点江山，评论世界大事。