第 5 章 软件构造

软件的详细设计完成，就表示完成了软件的过程性的描述，接下来进入软件构造阶段。

软件构造（Software Construction）指通过编码、验证、单元测试、集成测试和排错的组合，创建一个可以工作的、有意义的软件。在本章中，我们将聚焦在该领域当中的程序设计语言、设计、编码和复用这几个话题，其余的将在后续章节中讨论。

5.1 程序设计语言的选择

如果说人类的自然语言是人与人交流的工具，那么可以说，程序设计语言就是人和计算机进行通信的一种重要工具。不同的程序设计语言在不同层面反映了人们解决问题的思路，以及问题解决的方式和质量。同时，也会在程序可重用性上带来较大的差别。由此可见，程序实现之前的一项重要工作就是，选择适当的程序设计语言。

5.1.1 程序设计语言的分类

程序设计语言原本是被设计成专门使用在计算机上的，但它们也可以用来定义算法或数据结构。从 20 世纪 60 年代发展至今，已经先后设计和实现了种类繁多的程序设计语言。在使用的过程中，它们被改进、重组、演化，在各种不同的场合被广泛地使用。

按程序设计语言的级别，可以将语言分为两大类：低级语言和高级语言。

1. 低级语言

低级语言并不意味着语言的功能低下，而是指语言“距离”硬件的程度比较近。反之，高级语言则抽象程度高，不直接操纵硬件。低级语言依赖于特定的硬件（主要是 CPU），其使用复杂、烦琐、费时、易出差错，因而程序编写也有一定的难度。

常用的低级语言包括机器语言和汇编语言。

- 机器语言是直接用 CPU 二进制基本指令集或者是用经过符号化的基本指令集写成的代码，其存储由语言本身决定。无疑，用机器语言编写程序是所有程序员的噩梦。
- 汇编语言是机器语言中地址部分符号化的结果，比机器语言更直观。汇编语言程序中可以包括一些宏构造，使其显得比机器语言更容易理解和使用。不过，即使汇编语言存在着生产效率低、维护困难、容易出错等缺点，但是在实现与硬件系统接口部分时，因其易于实现且效率高，仍然被频繁使用。

2. 高级语言

高级语言的表示方法要比低级语言更接近于待求解的问题，其特点是在一定程度上与具体硬件无关，易学、易用、易维护。高级语言的使用极大地提高了软件的生产效率。

目前在世界范围内被使用的高级语言种类非常多。这些高级语言可根据不同的标准有不同的分类方式：

(1) 按语义基础分类

- 命令式语言：这种语言的语义基础是模拟“数据存储/数据操作”的图灵机可计算模型，十分符合现代计算机体系结构的自然实现方式。目前大多数的流行语言，例如 C、C++、Java、C#等都属于这类语言；
- 函数式语言：这类语言基于属于函数概念，例如 Lisp、Haskell、F#等；
- 逻辑式语言：这类语言基于一组已知规则的形式逻辑系统，例如 Prolog。

(2) 按数据类型检查的时机分类

- 静态语言：数据类型的检查在编译阶段进行。这类语言有 C、C++、Java 等；
- 动态语言：数据类型的检查在运行阶段进行。这里语言有 Perl、Python、Ruby、PHP、JavaScript 等。动态语言的特点是需要解释器的支持。

(3) 按语言对类型的约束分类

- 强类型语言：数据类型必须强制声明，即在编译阶段就必须确定每一个变量/对象的类型。在强类型语言程序中，给定数据对象的类型在这个对象的生命期内是不能改变的。所以说，这类语言是“类型安全”的。
- 弱类型语言：数据类型的声明是可以忽略的（甚至是不支持的）。在弱类型语言写成的程序中，变量的类型由初始化或者运算结果决定。例如，在 JavaScript 中，如果有以下变量初始化

  var a = (“12” + 3) * 4

  那么变量 a 的类型是整型，其值为 492。当然，在此后的计算过程中，a 的类型还可以发生改变。

对类型的弱约束使得弱类型语言的程序容易编写（因为程序员可以不必过分关心变量的类型。要知道，有时由程序员来确定类型是比较困难的）。但也正是这个特点，也增加了出错的可能性，程序的排错也比较困难。

(4) 按思维方式分类

- 面向过程：面向过程的语言以函数为基本语法单位，例如 C、Pascal 等；
- 面向对象：面向对象的语言以类为基本语法单位，例如 Smalltalk、Java 等；
- 混合型：面向过程和面向对象的特点兼而有之，例如 C++等。

5.1.2 高级程序设计语言的特点

目前，很多的软件系统选用的程序设计语言都属于高级语言类。这里，我们就高级程序语言的特点进行一些讨论。

1. 高级程序设计语言的基本组成

虽然不同的程序设计语言，在其各自的用途和实现上有很大的区别，但是它们之间在其基本的组成成分上却大同小异。其基本的组成成分如下：

(1) 数据成分

数据成分用于描述程序所涉及的数据。

① 变量/对象命名

变量/对象的名字在程序中有非常重要的作用，因为它能直接反应程序员的设计意图。因此，为其取一个有意义的名字是非常必要的。例如，设计一个时钟变量，那么将其命名为 clock 就明显好于将其简单名为 c。

对于 C 这样强类型的语言，对象必须在使用前命名。这可以方便编译器进行名字查找，以检查出潜在的错误。

② 数据类型声明

程序中命名的对象必定属于某种数据类型。数据类型的一种规范性的声明，是对对象的一种约束。它说明了该类型的对象将占据的内存大小，以及能参与的运算。编译器会对任何类型不符的运算报出警告。隐式类型转换和强制类型转换可以避免这类警告，但程序员应当清楚地认识到这些转换的隐患。

强类型语言常用的数据类型有整型、浮点型、字符型、用户自定义型（如结构体、类）等。弱类型语言没有明显的类型说明过程，但在其内部实现中，对象仍然属于某种特定类型。

⑵ 运算成分

运算成分用以描述程序中所包含的运算。

① 初始化

初始化使变量有一个运算的起点。这与赋值是不同的。例如：

int a = 0;

这是初始化过程。而在变量声明后的语句

a = 0;

则是赋值。

程序设计实现时最常出现的一种错误就是没有对要运算的数据进行初始化，即没有赋予一个合适的初始数据，结果造成了编译或运行时出错。

【例 5-1】编码错误示例：直接使用未初始化的指针

```
void foo()
{
    int *p;
    *p = 10; //错误，指针未初始化，它没有指向某个存储单元
}
```

类似的错误应当在编码时尽量避免。

② 运算对象

运算对象是程序执行时要运行的对象，包括一个算术表达式或者一个逻辑表达式，或者是一条完整的语句，如赋值语句等。

⑶ 控制成分

控制成分用以描述程序中所包含的控制，它们决定了程序的流程。

高级语言包含三种控制结构：

① 顺序控制结构。顺序执行的语句构成了顺序结构。

② 分支控制结构。分支结构将程序流程导向不同的分支，以覆盖需要选择、判断的场合。常见的分枝语句有 if 语句、if…else if 语句和 switch-case 语句三种，其中前者属于单

/双路分支语句，后二者属于多分支语句。

③ 循环控制结构。被包含在此类结构中的语句序列要被重复执行。循环控制结构一般用语句实现。常见的循环语句有 for 语句和 while 语句。使用循环语句需要注意两点：一是循环的初始化，二是循环必须有一个终点。

图 5.1 是以上三种控制结构的框图。

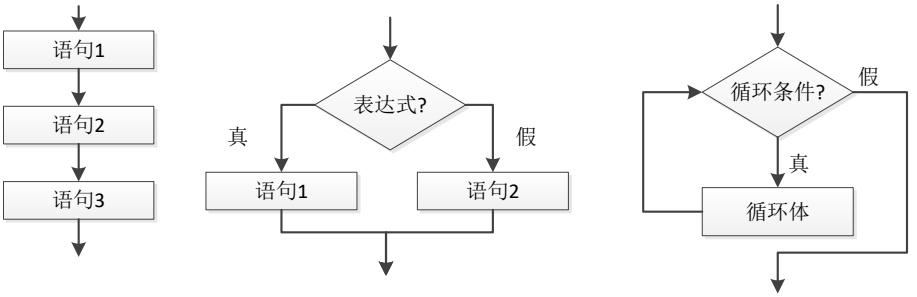

图 5.1 高级语言的三种控制结构

⑷ 传输成分

传输成分用以表达程序中数据的传输。

传输成分包括基本的输入和输出。在一些语言（如 C）中，输入和输出不是以语句的形式呈现的，而是用库函数的方式来实现。

5.1.3 程序设计语言选择准则

在程序设计阶段所遇到的首要问题是：如何选择程序设计语言？通常应根据软件系统的应用特点、程序设计语言的内在特性，以及系统的性能要求等方面来进行选择。

根据高级语言的内在特性，编码时选择语言应该考虑以下因素：

- 项目的应用领域。应尽量选取适合某个应用领域的语言。
- 算法和计算复杂性。要根据不同语言的特点来选取能够适应软件项目算法和计算复杂性的语言。
- 软件的执行环境。要选取机器上能运行且具有相应支持软件的语言。
- 性能因素。应结合工程具体性能来考虑。例如，实时系统对响应速度有特殊要求，就应选择汇编语言、C 语言等。
- 数据结构的复杂性。要根据不同语言构造数据结构类型的能力选取合适的语言。
- 软件开发人员的知识水平及心理因素。知识水平包括开发人员的专业知识，程序设计能力；心理因素是指开发人员对某种语言或工具的熟悉程度。要特别注意在选择语言时，尽量避免受外界的影响，盲目追求高、新的语言。

5.2 程序设计方法

目前流行的程序设计方法有两种：结构化设计和面向对象设计。

5.2.1 结构化程序设计

结构程序设计的概念最早是由 E.W.Dijkstra 提出来的，他指出：结构程序设计是创立一种新的程序设计思想、方法和风格，以显著提高软件生产效率和质量。

提高程序可读性的关键是使程序结构简单清晰，结构化程序设计（SP）方法是达到这一目标的重要手段。

1. 结构化设计概念和特点

结构化程序设计是一种程序设计技术，它采用自顶向下、逐步求精的程序设计方法及单入口和单出口的控制结构。具体来说结构化程序设计技术，具有以下主要特点：

(1) 自顶而下，逐步求精

这种逐步求精的思想符合人类解决复杂问题的普遍规律，从而可以显著提高软件开发的效率。而且这种思想还体现了先全局、后局部，先抽象、后具体的方法，使开发的程序层次结构清晰，易读、易理解，还易验证，因而提高了程序的质量。

将程序自顶向下逐步细化的分解过程用一个树型结构来描述，如图 5.2 所示。

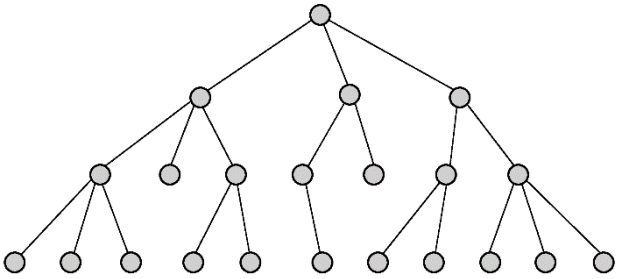

图 5.2 自顶而下的程序分解过程

(2) 单入口和单出口的控制结构

结构化的程序由且仅由顺序、选择、循环三种基本控制结构组成，既保证了程序结构清晰，又提高了程序代码的可重用性。这三种基本结构可以组成所有的各种复杂程序。

在一些程序中常用的递归可以被转换为循环结构。实际上，递归是非必须的，且是不易理解的。但递归可以使代码更加简洁，因此仍被有节制地使用。

2. 结构化程序设计的基本原理

结构化程序设计的基本原理中一个重要的概念是“模块化”。因为要实现结构化的程序设计总的指导思想是：自顶向下，逐步求精，模块化程序设计，分而治之思想和最终的结构化编程。相应地，结构化程序设计的步骤如图 5.3 所示。

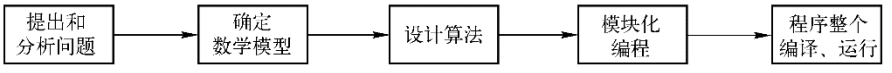

图 5.3 结构化程序设计步骤

模块是由边界元素限定的相邻的程序元素的序列，而且有一个总体标识符来代表它。

所谓模块化，即是将一个大任务分成若干个较小的任务，较小的任务又细分为更小的任务，直到更小的任务只能解决功能单一的任务为止。一个小任务称为一个模块。各个模块可以分别由不同的人编写和调试。把大任务逐步分解成小任务的过程可以称为是“自顶向下，逐步细化”的过程。

对于模块的设计和实现有以下五条基本的标准：可分解性，可组装性，可理解性，连续性，保护性。

遵循以上标准是进行结构化程序设计时运用模块化原理的基本准则，这样设计出来的程序不但软件结构清晰，而且代码也有很好的可读性和可维护性。

3. 优化设计

结构化设计中对于优化设计的要求是比较高的。考虑优化设计，基于软件开发的可靠性、高效性，在有效模块化的前提下尽量使用最少的模块。同时优化设计应该在软件结构设计的早期就开始进行，力求结构的精简，并能满足相同的功能需求。

优化设计有多方面的内容，如结构的优化、功能的优化、算法的优化和时间、效率的优化等，这里介绍对时间起决定性作用的软件的优化方法。

① 在不考虑时间因素的前提下开发并精简软件结构。

② 寻求软件设计结构中的“关键路径”和“关键事件”。所谓“关键路径”，是指影响整个软件开发的主要事件和模块系列；而“关键事件”是指在这个关键路径中，最耗费时间的模块。然后仔细地设计该模块的实现算法。

③ 选择合适的高级编程语言，提高程序的编译效率。

④ 在效率和实现功能之间寻求平衡点。所谓的平衡点是指不要求为了一些不必要的功能而耗费大量的时间，从而降低效率，以致得不偿失。

5.2.2 面向对象程序设计

结构化程序设计的特点是“自顶向下，逐步求精”，是一种先全局、后局部的程序设计方法。在高级程序设计语言产生的早期，结构化程序设计方法占了主导地位。然而，随着时间的推移，软件变得越来越庞大，设计过程也变得越来越复杂，对软件的可重用和可扩展性的需求越来越强烈。而结构化程序设计的先天不足导致这样的需求要满足起来非常困难。在这种形式下，面向对象的程序设计方法应运而生。

依据面向对象技术的观点，客观世界是由大量对象构成的，每一个对象都有自己的运动规律和内部状态，不同对象之间的相互作用和互相通信构成了完整的客观世界。因此，从思维模型的角度，面向对象很自然的与客观世界相对应。

实际上，计算是一种仿真。如果每个被仿真的对象都由一个特定的数据结构来表示，并且将相关的操作信息封装进去，那么仿真将被简化，可以更方便地刻画对象的内部状态和运动规律。面向对象就是这样一种适用于直观模型化的设计方法。这意味着现实世界问题空间中的模型与程序设计者在计算机中的建立的模型有近乎一对一的对应关系。这一思想非常利于实现大型的软件系统。

1. 面向对象的核心概念

面向对象程序设计方法从诞生其就致力于解决软件生产过程中的可重用和可扩展的问题。

为此，所有面向对象的程序设计语言都支持面向对象技术的四个核心概念：数据封装、继承、多态和泛型编程。

(1) 数据封装

数据封装使用抽象原则来描述客观事物。抽象是指对于一个系统的简化的描述。对于使用系统的人员，不会去关心该系统的组成和工作的原理；他们所关心的是该系统具有什么样的功能，如何去使用该系统（即系统提供什么样的接口让人们使用）。当然，对于该系统的实现人员，需要关心的是该系统的一切情况。

数据封装将一组数据和这组数据有关的操作集合封装在一起，形成一个能动的实体，这类实体的内存映像称为“对象”。在使用时，用户不必知道对象行为的实现细节，只需根据对象提供的外部特性接口方法访问对象。这里举一个复数例子来对比一下结构化思想和面向对象的思想的不同。

【例 5-2】复数的 C 语言实现。身为 C 语言程序员 A 可能使用直角坐标系中的（实部，虚部）模型来描述复数，其数据结构如下：

```
typedef struct
{
    float real, img;
} Complex, *PComplex;
```

再设计一系列接口函数来操作这个数据结构，例如：

```
Complex add(Complex c1, Complex c2);
Complex mul(Complex c1, Complex c2);
```

完成后，他将代码交付给程序员 B 使用。程序员 B 在了解了 Complex 结构体的构造后，为了提高效率，在他的程序中绕开了接口函数，而直接使用了 Complex 的成员。另一方面，当程序员 A 了解到应用程序使用复数的乘除法多于加减法，因此，为了提高效率，他重新设计了数据结构，采用了利于复数乘除法的（模，幅角）模型表示法：

```
typedef struct
{
    float mod, theta;
} Complex, *PComplex;
```

接下来发生的情况可想而知：这种改变对程序员 B 来说是灾难性的，他不得不重写他的代码。

造成这种困境的根本原因是：C 语言虽然在一定程度上做到了数据封装，但没能完整地实现信息隐蔽。

面向对象技术试图通过建立一个合适的数据类型，将复数的内部数据（称为数据成员）和函数（称为成员函数）结合在一起，形成一个新的抽象数据类型，称为类类型（class）。以下代码就是一个描述复数的 C++类，其他面向对象语言的类语法大同小异。

【例 5-3】复数的 C++实现

```
class Complex
{
private:
    float real, imag;
```

```
public:
    Complex add(Complex c);
    Complex mul(Complex c);
};
```

在这样建立的类 Complex 中，能确保私有数据只能由类中的成员函数进行访问和处理。在任何时候，都可以自由地改变内部数据的组织形式，只需改变成员函数的实现细节。由于这些成员函数的接口不改变，系统其他部分的程序（及使用者）就不会由于改动而受到影响。

类的概念将数据和与这个数据有关的操作集合封装在一起，建立了一个定义良好的接口，人们只关心其使用，不关心其实现细节。这反应了抽象数据类型的思想。

(2) 继承

继承是面向对象语言的另一个重要的概念，是软件可重用和可扩充问题的基石。

要了解什么是继承，首先从对象的关系入手。在客观世界中，可以将对象之间的关系分为两种：

- 整体和部分的关系。举个例子：车轮是汽车的一部分，而车轮不是一辆汽车。车轮和汽车体现了一种 Has-A 的关系。
- 一般和特殊的关系。猫和哺乳动物之间就构成了 Is-A 的关系。我们一般不说：猫是哺乳动物的一部分，因为这种逻辑关系是不确切的。

在这个意义上，继承实现了一般和特殊的关系。例如，一种对交通工具进行分类的可能如图 5.4 的树状结构所示：

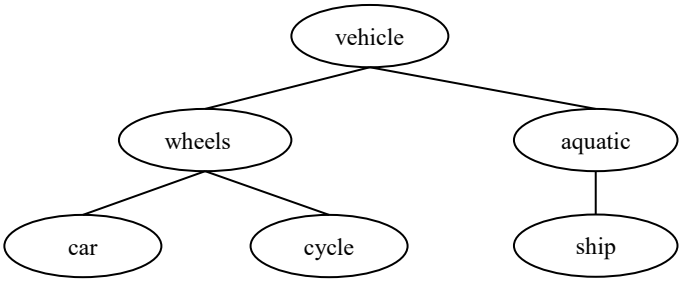

图 5.4 交通工具的分类

在这棵类别树上，最高层（根结点）是具有最普遍的特性。在其下的每一层中，每个节点拥有的描述都比它之前的层更具体，例如 wheels，它明确说明此属于类别的交通工具是有轮子的。而在最下层，一旦确定 car 为 wheels 的一种，那么就不必指出它是有轮子的，因为 car 沿着它的继承链自动继承了上层祖先所有的特性。

在面向对象语言中，类的继承机制仿真了自然界中的这种继承机制：除了根结点外，每个类都有它的基类/父类(base/parent class)；除了叶结点外，每个类都有它的派生类/子类(derived/child class)。基类抽象出共同特征（共性），派生类表达其差别（个性）。

有了类的层次结构和继承性，不同对象的共同性质只需定义一次。一个子类可以从它的父类那里继承所有的数据和操作而不必重复说明，并能扩充自己的特殊数据和操作。以下 C++代码示意了继承机制。

【例 5-4】继承和多态的 C++示例代码

```
class vehicle
{
protected:
    string name;

public:
    virtual void drive() = 0;
    virtual void brake() = 0;
};

class wheels : public vehicle
{
protected:
    int wheelNum;
};

class car : public wheels
{
public:
    void drive() { ... }
    void brake() { ... }
};
```

从示例代码可以清晰地看到：子类无须重复定义父类中已有的属性和操作，而每一个子类都实现了上层父类未实现的操作。这种特性使得用户可以充分利用已有的类扩展出心累，从而达到软件重用的目标。

(3) 多态性

面向对象另外一个核心概念是多态性。所谓多态，是指一个名字（或符号）具有多种含义。这对仿真客观世界以及提升软件的灵活性有相当重要的意义。

在面向对象的程序设计语言中，多态是通过函数重载(overload)来实现的。函数重载系指两个或多个函数具有相同的名字，它们通过参数（个数、类型）加以区分。

下面让我们来考察多态性问题的一个类比问题。当一位汽车司机为避免撞车时刹车，他关心的是快速刹车（效果），而不关心刹车是鼓式刹车还是盘式刹车（实现方法的细节）。这里，刹车的使用与刹车的结构是分离的概念，可能有多种结构的刹车，它们的使用方法是相同的。相同的使用方法（相同界面）对应于不同种类的刹车结构（多种实现），这反映了多态性的思想。

与此类似，用户在使用别的程序员提供的函数时，关心的是该函数的功能及其使用接口，而并不需要了解该函数的使用接口与函数的哪一种实现方法相匹配（binding）。也就是说，在设计这一级上，软件人员只关心“施加在对象上的动作是什么”，而不必牵涉“如何实现这个动作”以及“实现这个动作有多少种方法”等细节。

对于函数重载，若函数调用（接口）与对应的函数体（函数实现）相匹配，是在编译时确定的，称为“早期匹配（early binding）”；如果函数调用与对应的函数体的匹配是在运行时动态进行的，称之为“晚期匹配（lately binding）”。一般来说，早期匹配执行速度比较快，晚期匹配提供灵活性和高度的问题抽象。在面向对象的语言中，同名函数一般体现了早期匹配，而继承链中的虚函数（virtual function）提供了晚期匹配带来的良好特征。在例 5-4 的第 7、8 两行代码中，函数声明前标有 virtual 字样的函数即为虚函数。特别的，这些函数后的=0 标记说明它们是纯虚函数，在该类中没有函数体（即没有实现），而在继承链的最终子类中实现它们。

普通函数重载强调的是函数名相同，函数参数和函数体不相同（编译能根据参数的差别进行识别和匹配）。虚特性的函数则强调单接口多实现版本的方法，亦即函数名、返回类型、函数参数的类型、顺序、个数完全相同，但函数体可以完全不同。在继承链中，父类及其子类中都可以定义这个虚函数的不同实现，例如：car 类的 brake()和 truck 类的 brake()是完全不同的：

```
void car::brake() { std::cout << "stamp brake pedal" << std::endl; }
void cycle::brake() { std::cout << "pinch hand brake" << std::endl; }
```

在使用时，系统能在运行时刻动态地寻找所需的实现版本。例如代码：

```
wheels * p = new car();
p->brake();
wheels * q = new cycle();
q->brake();
```

的输出是不同的。

从代码中可以看到，多态机制使得相同类型的方法调用能根据对象的不同产生不同的效果。这种机制使得软件可扩充性更为自然。

(4) 泛型编程

泛型编程(generic programming)不是面向对象语言的专属，但却在其中体现得更加淋漓尽致。

所谓泛型编程，就是以独立于任何特定类型的方式编写代码。这使得程序员在编写充分可重用代码时有了趁手的工具。

我们知道，像 C++这样的高级程序设计语言都是强类型语言。所谓强类型，就是指编译器要在编译时对数据的类型进行严格的检查和匹配，以免类型失配时带来的严重隐患，甚至错误。这对保证程序的健壮性和安全性有很大的好处，但却以损失灵活性为代价。

考虑这样一个问题：编写求一个数的绝对值函数。由于类型的限制，我们不得不为不同的数据类型编写一系列代码极为相似的函数，如例 5-5 所示。

【例 5-5】函数重载

```
int abs(int a) { return a > 0 ? a : -a; }
long abs(long a) { return a > 0 ? a : -a; }
double abs(double a) { return a > 0 ? a : -a; }
```

显然，这些都是重载的函数。编译器会根据调用时参数的类型仔细选择匹配的版本。

然而问题是，当某个函数的实现发生了改变，其它的函数也必须发生相应的变化。这无

疑对代码的维护造成了困难。那么，可不可以只用一个函数应对不同类型呢？

仔细分析上面的那些代码，可以发现，三个版本的函数除了类型不同外，其余部分都是完全相同的。那么，我们自然会想到，能否将类型作为参数传递呢？

在 C 这样的语言中，类型参数化是不容易实现的，或者实现起来很笨拙，效率不高。而在面向对象的语言中却是手到擒来，其原因就是面向对象技术引入了泛型(generics)编程的思想。

泛型其实是一种形式的静态多态，实现方式是类型参数化。一旦类型本身被参数化，那么我们就可以跃过类型限制带来的鸿沟，用一个类或者函数操纵多种类型不同的对象，并且不需要知道实现的细节。这无疑为代码的编写和维护带来的巨大的好处。

在面向对象的程序设计语言中，泛型编程主要依托模板(template)来实现。模板有函数模板和类模板之分，它们的存在为泛型编程打下了坚实的基础。例如，前面的三个求绝对值的函数可以使用一个 C++函数模板来统一：

```
template <typename T>
T abs(T a) { return a > 0 ? a : -a; }
```

其中，用 typename 修饰的符号 T 就是类型参数。在用 int c = max(1, 2)这样的方式使用函数模板时，编译器会将 int 作为参数传递给 max 函数模板并生成一个 int 版本的实例函数。其他的类型与此类似。

除了函数模板，程序员们还可以使用功能更强大的类模板。关于类模板的内容这里不再赘述。

总的来说，面向对象的设计方法使用对象将信息局部化、并使程序结构与设计结构相吻合的优点，有利于在完善和维护阶段对软件进行修改，也有利于其它人（非设计人员）来清除软件错误。程序员容易确定程序的哪些部分依赖于正要修改的片断，而且正在修改的部分对其他部分影响很小。这对大型、复杂软件的维护和改进是很重要的。

面向对象设计非常注重设计方法，因为它要产生一种与现实具有自然关系的软件系统，而现实就是一种模型。实际上，用面向对象方法编程的关键是模型化。程序员的责任是构造现实的软件模型。此时，计算机的观点是不重要的，而现实生活的观点才是最重要的。

因此，可以将面向对象的目标归纳为：对试图利用计算机进行问题求解和信息处理的领域，尽量使用对象的概念，将问题空间中的现实模型映射到程序空间，由此所得到的自然性可望克服软件系统的复杂性，从而得到问题求解和信息处理的更高性能。

2. 面向对象程序设计方法

在充分理解了面向对象核心概念的基础上，可以对在设计阶段完成的对象/类进行编码。编码一定涉及到某种程序设计语言。我们可以根据应用的需求和系统的类型，选定某种或某几种程序设计语言作为系统的编程语言。

实际上，从 OOD 阶段到 OOP 阶段的转换是相当直观的，因为在 OOD 阶段已经建立了完整的对象模型，将这些类模型映射到程序设计语言中的类是相当容易的事情。当然，在 OOP 阶段，程序员可能因为系统的需要而有控制地修改已经设计好的类，以方便编码。

一般情况下，我们会选用某种建模工具来建立类模型，而这些工具往往提供了正向工程功能，即工具会根据模型，用选定的语言自动生成类框架源代码。这项有用的功能会极大地提高编码效率。

关于如何编码不是本书的任务。因此，这里我们简单讨论一下在编码过程中如何去运用所选定的程序设计语言。以下提到的只是对语言运用的实践性指南，而非绝对的标准。

(1) 使用“标准语”而非“方言”

有些厂商推出的程序设计语言产品包含一些非标准的扩展，我们可以视这种语言为一种“方言”。方言可能会对特定环境的程序设计带来好处，但却极大地降低了代码的可移植性。所以，要尽量采用语言的国际通用标准来进行编码，以免增加代码的维护成本。

(2) 深挖并运用语言的优良特性

每一种程序设计语言都有一些优良的特性，而这些特性往往与语言的新标准/新版本有关。使用新特性，例如 C++ 11 中的右值引用、move 语意和类型自动推导，可以使编码和程序的运行效率提高。随时关注所用语言的新标准/新版本总是对程序设计有帮助的，至少可以从中获得一些灵感。

需要注意的是：不是所有的编译器都支持新标准/新特性。例如：GCC 的 g++编译器支持几乎所有的 C++ 11 标准，而 VC 12 则是有保留的支持。所以编写需要交叉编译的应用时，要将这些因素考虑进去。

(3) 多使用标准库

几乎所有的程序设计语言都会提供一些标准库。使用这些标准库可以达到事半功倍的效果，极大地提高编码效率。例如在用 C++编写应用时，程序员不必从头编写链表类，可以直接使用 C++标准模板库 STL 中的 list 类。

5.3 程序设计风格

程序实际上也是一种供人阅读的“文章”，只不过它不是用自然语言而是用程序设计语言编写的；因此，一个逻辑上虽正确但杂乱无章的程序是没有什么价值的，因为它无法供人阅读，也难以测试、排错和维护。

讨论程序设计风格，力图从编码原则的角度来探讨提高程序的可读性、改善程序质量的方法和途径。

5.3.1 源程序文件

1. 符号的命名

① 尽量用与实际意义相同或接近的标识符为符号命名，这样可以“见名知意”。例如，把交换两个数的函数命名为 swap，明显好于仅仅把它命名为 f；
② 在程序中尽量使用人们习惯用的符号名；
③ 可以使用长一点的名字，但不要使名字太长，这样既增加了输入时间，又增加了出错的可能；
④ 不要让命名规则成为“法定”的，但一定要在命名时保持一致性。

2. 源程序中的注释

注释可分为序言性注释和解释性注释。序言性注释是在一个程序或模块的开头对本程序段的模块功能、接口信息等做必要的说明。解释性注释是对程序正文中说明语句段和程序段功能的解释性说明。

不要过多考虑注释的风格，但一定要写有用的注释。

3. 源程序的书写格式

将源程序缩进编排，并加上适当的空格和空行，可使程序的结构关系更清晰，以提高其可读性。

涉及源代码书写风格的一个明显例子是花括号的位置，它有两种常用的风格：

```
void foo() { //K & R 风格
    …
}
void boo() //每个括号一行
{
    …
}
```

采用哪种风格由程序员自定，但一定要在整个程序设计过程中保持一致。

5.3.2 语句构造方法

语句构造的技术，尤其是流程控制语句的构造技术，直接影响到程序的可读性及效率。应该采用直接、清晰的构造方式，而不要为了提高效率或者显示技巧而降低程序的清晰性和可读性。

【例 5-6】有以下 C 程序段，用其它表达式代替逻辑表达式：

```
int   *p = (int *)malloc(sizeof(int));
if (p)
    *p = 1;
```

在规范情况下，if 语句的判断部分应该是个逻辑表达式，而代码中却使用了元表达式。虽然二者的效果相同，但易读性差，特别是对初学者。为了改善程序的易读性，应采用直截了当的描述方式。改进的程序段如下：

```
if (p!=NULL) …
```

5.3.3 数据说明方法

要使程序中的数据说明更易于理解和维护，必须遵循以下原则：

① 数据说明的次序应当规范化，使数据的属性更易于查找，从而有利于测试、纠错与维护。例如可按照以下说明次序：常量说明；简单变量类型说明；复杂类型说明等。

② 一个语句说明多个变量时，各变量应该分类并且最好按字母顺序排列。例如应该把

```
int size, length, width, cost, price;
```

写成

```
int cost, price;
int length, size, width;
```

③ 对于复杂的数据结构，要加注释，说明在程序实现时的特点。例如，对 C 语言定义的链表结构和 Java 中用户自定义的类的类型，都应当在注释中做必要的补充说明。

5.3.4 输入/输出技术

对输入/输出技术的要求是：

① 输入和输出的格式应尽可能统一。

② 输出信息中应该反映输入的数据，这是检查程序运行正确性和用户输入数据正确性的需要。

③ 输入和输出应尽可能集中安排。

5.3.5 编码策略

实际应用中可采用的编码策略较多，这里仅录用一些具典型意义的来讨论。

1. 在高级别的警告模式下编译程序

在编译时出现的警告信息是不应该忽略的。一个有用的建议是将编译器的警告级别设为最高，以便能尽可能多地检测出程序中隐含的错误。我们编写的程序应该是“warning-free”的干净代码。要做到这一点，就要求程序员正确地理解警告信息，然后修改代码而不是降低警告级别来消除编译警告。

【例 5-7】编译警告。下面的程序片段在编译时将导致这样的警告：“Variable defined but never used.”

```
void foo()
{
    int forLaterUse; //为将来的编码预先定义的但从不使用的变量
}
```

为了消除警告信息，可以用这样的方法：

```
void foo()
{
    int forLaterUse;
     forLaterUse;
}
```

2. 尽量采用自动化的编程工具

一般高级语言程序的开发需要经过编辑、编译、链接的过程，而这些过程中的每一步都有可能出错，从而引起过程的反复。而我们开发的应用，往往由多个源程序构成，分离的开发工具将会使开发过程变得更加复杂，同时也增加了出错的概率。集成化的编程工具，例如 Visual Studio，可以自动地将上述过程“一键”完成，从而大大加快了开发进程。

3. 交叉审阅代码

邀请别的程序员来与您互相审阅对方的代码，这非常有助于发现您自己在思维定势的驱动下无法检测到的错误，同时提出改进代码的意见和建议。这能促进彼此的提高。

4. 只让一个功能模块完成一个任务

不要试图编写大而全的功能模块。一旦发生错误，错误极有可能在多个任务中蔓延，最终导致代码不可恢复。让模块聚焦在一件事上，让它具有高的内聚性。

5. 正确性、简单性和清晰性是第一位的

在编程时遵循 KISS(Keep It Simple Software)原则：正确凌驾速度，简单好过复杂，清楚优于酷炫。

6. 尽量避免全局和共享数据的使用

共享总是与竞争想成对出现。所以，尽量避免使用共享数据，尤其是全局数据。共享数据增加了模块间的耦合性，它使代码的可维护性和性能降低。

初级 C 程序员往往大量使用全局变量，因为它们可以在模块间方便地传递信息。但这是很危险的事情。全局变量的错误可能导致错误的全局性蔓延，在发现错误后也非常难以修改。通过深入学习编程技术和累积编程经验可以解决这个问题。

7. 总是初始化变量

使用未经初始化的变量，尤其是指针，是非常危险的，它们常常是程序的祸根。在定义它们的时候就完成初始化，以绝后患。

8. 避免编写行数很多的函数，避免过深的嵌套

长代码带来的缺点是维护不易，阅读理解都不易。长代码中总是能够找出可以复用的单元，应该将它们拆分出来。

5.4 算法与程序效率

设计逻辑结构清晰、高效的算法，是提高程序效率的关键。也就是说，源程序的效率与详细设计阶段确定的算法的效率直接相关。而在将详细设计的描述转换成源程序代码后，算法效率就反映为对程序的执行速度和存储容量的要求。

1. 算法转换过程中的指导原则

算法转换过程中的指导原则是：

①算法在编码前，尽可能化简有关的算术表达式和逻辑表达式。

②仔细检查算法中嵌套的循环，尽可能将某些语句或表达式移到循环外面。

③尽量避免使用多维数组。

④尽量避免使用指针和复杂的表。

⑤采用“快速”的算术运算。

⑥不要混淆数据类型，避免在表达式中出现类型混杂。

⑦尽量采用整数算术表达式和布尔表达式。

⑧选用等效的高效率算法。

上述原则要在转换时统筹考虑，而不应该教条地使用。例如，程序员紧守的一条原则就是尽量不用 GOTO 语句。但是当要从一个嵌套很深的循环中直接跳出来时，GOTO 语句就能很好地发挥作用。

2. 影响效率的因素

基于以上原则，同时可以从以下三个方面进一步讨论效率问题。

(1) 算法对效率的影响

上面关于算法转换的指导原则，在一定程度上减小了算法对效率的影响。从这一点能够看出算法直接影响到的是程序，进而影响到了整个代码的效率问题。除在转换时需要注意效率之外，在程序设计和实现时同样需要考虑效率的问题。

【例 5-8】算法对效率的影响。这里我们用顺序表的查找来说明算法对效率的影响。顺序表指的是已经排好序的线性表，它的查找算法有多种，比较常见的有顺序查找法、二分查找法等。下面是两种算法的 C 语言表达(假设顺序表是从小到大排序的)。

顺序法	二分法

int find(int *list, int len, int key)
{
int i;
for (i = 0; i < len ; i++)
if (list[i] == key) return i;
return -1;/* Not found */
}
	int find(int *list, int len, int key)
	{
	int low = 0, high = len - 1, mid;
	while (low <= high)
	{
	mid = (low + high) / 2;
	if (list[mid] == key) return mid; /* Found */
	else if (list[mid] > key) high = mid – 1;
	else low = mid + 1;
	}
	return -1;/* Not found */
	}

二分法看起来要复杂一些。但是分析一下这两种算法的时间复杂度，就会有不同的结论。假设顺序表的长度为 N，很明显，顺序法的时间复杂度为 $O(N)$，而二分法的时间复杂度却只是 $O(\log_2 N)$。可以看出，后者的效率明显高于前者，尤其是表的长度较大时。因此，当程序要实现对较长顺序表的查找时，二分法是很好的选择。

(2) 存储效率

处理器的分页调度和分段调度的特点决定了文件的存储效率，同样对于代码也存在这个问题。提高效率的办法通常也是提高存储效率的方法。

(3) 输入/输出效率

输入/输出的效率决定的是人与计算机之间通信的效率，程序设计中输入和输出的简单清晰，是提高输入/输出效率的关键。

3. 代码的优化

进阶的程序员往往容易沉迷于代码优化，因为这项技术可以让代码变得非常漂亮，成为一种炫耀技巧和成熟的资本。然而，优化过的代码却是不容易理解和维护的，同时也是逻辑错误的藏身之地。所以，不要过早地、贸然地去优化代码。如果一定要这么做，请仔细阅读下面的优化原则。

优化的第一原则是：不要去做。优化的第二原则是：还是不要去做。三思而后行。

请牢记这条原则：“让一个正确的程序变快远比让一个快的程序变正确容易”。因此，尽可能编写清晰而可读性好的代码。仅仅在必须的时候优化代码。

5.5 软件代码审查

用高级语言进行应用开发时，为了保证代码开发的质量，制定了这样的政策：代码审查或者叫 Review。也就是说，在代码完成后，要通过其他人的审查，确认没有问题，才能提交进入最终版本。

代码审查是软件排错的有效方法，应该从哪些方面进行审查呢？表 5-1 给出了对 C 代码进行软件代码审查的相关内容。对于其他语言，可以参考表中的项目。

表 5-1　软件代码审查

审查项	审查内容
程序的版式	空行是否得体
	代码行内的空格是否得体
	长行拆分是否得体
	“{”和“}”是否各占一行并且对齐于同一列
	一行代码是否只做一件事，如果定义一个变量，只写一条语句
	if、for、while、do 等语句自占一行，不论执行语句多少都要加“{ }”
	在定义变量（或参数）时，是否将修饰符*和&紧靠变量
	注释是否清晰并且必要
	注释是否有错误或者可能导致误解
文件结构	头文件和定义文件的名称是否合理
	头文件和定义文件的目录建构是否合理
	版权和版本声明是否完整

审查项	审查内容
	头文件是否使用了预处理块
	头文件中是否只存放“声明（declaration）”而不存放“定义（definition）”
命名规则	命名规则是否与所采用的操作系统或开发工具的风格保持一致
	标识符的长度应当符合“ min-length && max-information”原则
	标识符是否直观且可以拼读
	程序中是否出现相同的局部变量和全部变量
	类名、函数名、变量和参数、常量的书写格式是否遵循一定的规则
	静态变量、全局变量、类的成员变量是否加前缀
表达式与 基本语句	代码行中的运算符较多时，是否已经用括号清楚地确定表达式的操作顺序
	是否将复合表达式与“真正的数学表达式”混淆
	是否编写太复杂或者多用途的复合表达式
表达式与 基本语句	是否用隐含错误的方式写 if 语句，例如 ①将布尔变量直接与 TRUE、FALSE 或者 1、0 进行比较 ②将浮点变量用“==”或“!=”与任何数字比较 ③将字符串变量用“==”进行比较
	如果循环体内存在逻辑判断，并且循环次数很大，是否已经将逻辑判断移到循环体的外面
	case 语句的结尾是否忘了加 break
	是否忘记写 switch 的 default 分支
	使用 goto 语句时是否留下隐患，例如跳过了某些对象的构造、变量的初始化、重要的计算等

代码审查的目的是帮助开发团队标准化编码风格，并确保实施最佳的代码实践。它与编译器一起确保代码满足一定级别的质量要求。如果将自动代码审查集成到开发环境，就能够在开发阶段便发现并处理许多错误，以免将这些错误蔓延到后面的开发阶段。

5.6 软件复用

在软件生产过程中可以发现，许多软件产品之间存在着相当大的共性，特别是在同一个应用领域的软件更是如此。显然对相同或相似的软件产品从第一行代码开始进行重复的开发会造成人力、财力巨大浪费。在软件开发过程中使用现成的可以重复使用的软件产品，是非常明智的做法，这就是软件复用思想。

软件复用是指重复使用已有的软件产品用于开发新的软件系统，以达到提高软件系统的开发质量与效率，降低开发成本的目的。在软件复用中重复使用的软件产品不仅仅局限于程序代码，而是包含了在软件生产的各个阶段所得到的各种软件产品，这些软件产品包括：领域知识、体系结构、需求分析、设计文档、程序代码、测试用例和测试数据等。将这些已有的软件产品在软件系统开发的各个阶段重复使用，这就是软件复用的原理。

最早用于软件复用的软件产品是程序代码，这些程序代码最初是以子程序库的形式进行组织和管理的。软件开发人员通过使用相应的子程序名和参数，就可以在软件开发过程中重复使用这些程序代码。子程序库所代表的早期的软件复用主要是程序代码的复用，这是软件复用的一种原始形态。

随着软件复用技术的不断发展，软件复用的范围已经从最初的程序代码的复用，扩展到了更为广阔的范围，其中包含了体系结构、需求分析、设计文档、测试用例和测试数据的复用。

5.6.1 软件复用的级别

可复用的软件成分，也称可复用构件（Reusable Component），可从旧软件中提取，也可以专门为复用而开发。

将可用于软件复用的软件产品，按照其抽象程度的高低，划分为图 5.5 所示的复用级别。

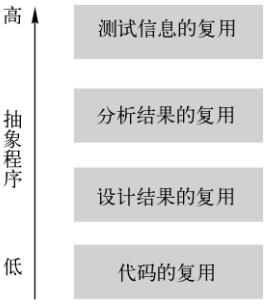

图 5.5　软件复用的级别

1. 代码的复用

这里的代码既包括二进制数形式的经过编译产生的目标代码，也包括文本形式的源代码。其中目标代码复用的抽象程度是最低的。目前大多数高级程序设计语言的开发环境都以库文件的形式向编程人员提供对许多基本功能的支持，如输入/输出、文件访问等功能。编程人员可以通过链接（Link）将库文件和自己编写的代码合并成为一个可执行的文件，通过这一方式实现对库文件中的目标代码的复用，从而避免编程人员重复地开发一些会被反复使用的程序代码。随着软件技术的发展，出现了许多目标代码一级的复用技术。

2. 设计结果的复用

设计结果比源程序的抽象级别更高，因为它的复用受实现环境的影响较小，从而使可复用构件被复用的机会更多，并且所需的修改更少。这种复用有三种途径：

① 从现有系统的设计结果中提取一些可复用的设计构件，并把这些构件应用于新系统的设计中；

② 把一个现有系统的全部设计文档在新的软硬件平台上重新实现，也就是把一个设计运用于多个具体的实现；

③ 独立于任何具体的应用，有计划地开发一些可复用的设计构件。

3. 分析结果的复用

这是比设计结果的复用抽象程度更高的复用，可被复用的分析结果是针对问题域的某些事物或某些问题的抽象程度更高的解法，受设计技术及实现条件的影响非常小，所以可复用的机会更大。复用的途径也有三种：

① 从现有系统的分析结果中提取可复用构件用于新系统的分析；

② 用一份完整的分析文档作为输入，产生针对不同软硬件平台和其他实现条件的多项设计；

③ 独立于具体应用，专门开发一些可复用的分析构件。

4. 测试信息的复用

它主要包括测试用例的复用和测试过程信息的复用。测试用例的复用指多次的软件系统的测试过程中重复使用同一测试用例，以降低测试工作的成本，提高软件测试的效率。

测试过程信息是在测试过程中记录的测试人员的操作信息、软件系统的输入/输出信息、软件系统的运行环境信息等与测试工作有关的信息。这些信息可以在对同一软件进行修改后的后续测试工作中重复使用。

5.6.2 软件复用过程

软件复用是一项复杂的系统工程，对开发单位和开发人员的技术、组织和管理水平都提出了较高的要求。根据许多软件复用的成熟经验，要实施高效合理的软件复用，充分发挥软件复用技术带来的巨大效益，需要具备这样一些基本条件：

① 高层管理人员的重视和有效的领导与组织，同时具备长期的经费支持；
② 先进行软件体系结构的规划，然后在此架构下进行软件复用；
③ 采用度量方法评价复用过程，并优化复用程序；
④ 坚持对软件开发人员在软件复用上的教育和技术培训；
⑤ 将软件系统中的可复用构件，作为一个独立的软件产品来进行控制和管理。

软件的复用通常有两种实施方式：系统地采用复用技术（简称系统复用）；渐进地采用复用技术（简称渐进复用）。

1. 系统复用

即在软件企业的开发活动中全面地采用软件复用技术。要把一个软件企业从传统的开发方式转换到基于复用的软件开发方式上来并非易事，它要求对整个企业的业务、人员、过程、工具、技术、组织机构、体系结构进行调整和变革。显然这样一种实施方式具有较大的风险。

2. 渐进复用

即在软件企业的开发活动中逐步渐增地采用复用技术。正如前面所述，全面系统地采用软件复用技术具有较大的风险。从传统机制到复用机制的转换并非易事，往往需要较长的时间，因此以逐步过渡的方式较为稳妥。不断地总结经验教训，逐步地扩展复用的覆盖面，直至贯穿整个企业的所有开发活动。

5.6.3 可复用构件

可复用构件是指可以在多个软件系统的开发过程中被重复使用的软件产品。它可以是需求分析、系统设计、程序代码、测试用例、测试数据、软件文档，以及软件开发过程中产生的其他软件产品。

1. 可复用构件的标准

可复用构件是一种特殊的软件产品，它与只在一个软件系统中使用的软件产品相比具有较大的差异。为了使可复用构件在软件开发过程中能被高效、方便地重复使用，以达到提

高软件开发的效率和质量、降低开发成本的目的，对可复用构件一般有以下的要求：

(1) 功能上的独立性与完整性

一个可复用构件应该具有相对独立的完整功能，构件与构件之间的联系应该尽可能少，彼此之间应该具有较为松散的耦合度，并且构件与构件之间的交互应该通过良好定义的接口进行。

(2) 较高的通用性

构件的通用性（一般性）越高，它的适用范围就越广，相应的可复用程度就越高，也就越能充分发挥软件复用的优势。所以在开发构件时，应该尽量提高构件的通用性（一般性），使其可以在更多的软件系统的开发中被重复使用。

(3) 较高的灵活性

可复用构件应该允许构件的用户根据具体情况对构件进行适当的调整，以适应不同用户和环境的具体要求。

(4) 严格的质量保证

可复用构件的可靠质量是其被复用的基础。所以对于构件的测试工作应该在不同的软硬件环境中进行。

(5) 较高的标准化程度

用于组装一个软件系统的可复用构件可能是由不用的组织或个人开发的，甚至可能是采用不同编程语言编写的，这就要求这些异质的构件具有定义良好的接口。目前，常用的构件技术规范有构件对象模型（COM）、公共对象的请求代理体系结构（CORBA）、EJB（Enterprise Java Bean），以及目前广为流行的 Web 服务。

2. 可复用构件的开发

在构件的编码阶段，需要充分考虑到可复用构件与一般应用程序的显著区别。为了使构件能够被较为广泛地复用，构件应该具有较强的通用性和灵活性。构件应该具有相当的一般性和抽象性，能够用于满足某一类相似的需求。一个过于特殊的构件是很难被重复使用的。即使一个通用性很高的构件也不可能完全适应用户的需求和运行环境。所以在一个构件被不同的应用复用时，对它的某些部分进行修改是不可避免的，需要为用户对构件的调整和修改留出余地。例如，继承、参数化、模板和宏都是典型的提高构件灵活性和可调整性的机制。同时为了保证不同的构件能够被正确地组装和交互，构件与外界之间的联系应该通过标准化的、定义良好的接口进行，而将构件的实现和内部数据结构加以隐藏。构件的封装性和彼此之间松散的耦合性对于降低构件系统开发难度和提高开发效率起着关键的作用。

进入测试阶段的构件应该经历比普通应用更为严格和充分的测试。同时，在测试过程中，要考虑可复用构件可能被应用于不同的运行环境中，所以在不同的软硬件环境中对构件进行多次测试，对于保证构件的质量和可靠性是非常必要的。

通过测试的构件就可以被提交到构件库中，由后者对其进行存储和管理，以备开发人员选用。除了构件本身需要被提交外，对构件的特征和属性进行描述的文档也需要一并提交，

以保证对该构件能够进行科学的管理和高效的检索。

3. 建立构件库

要在软件系统的开发过程中有效地实现复用，必须要求复用达到一定的规模，必须有大量的可供开发人员选择的可复用构件。构件的数量越多，找到合适构件的可能性也就越大，应用系统的复用程度也就越高。但是随着构件数量的增加，如何有效地对这些构件进行组织和管理就成为构件复用技术成败的关键。如果大量的构件没有被有效地组织和管理起来，那么要在一堆没有任何结构、散乱的构件中，找到满足特定需求的构件是一件十分困难的事情。因此，当构件的数量达到一定规模时，采用构件库对其进行组织和管理是十分必要的，构件库的组织和管理水平直接决定着构件复用的效率。构件库是用于存储、检索、浏览和管理可复用构件的基础设施，构件库的组织和管理形式要有利于构件的存储和检索，其最关键的目标是支持构件的使用者可以高效而准确地发现满足其需要的可复用构件。

为了达到有效地进行软件复用的目的，构件库一般应具备以下功能：

① 支持对构件库的各种基本的维护操作，如在构件库中增加、删除、更新构件。

② 支持对构件的分类存储，根据构件的分类标准和模型将构件置于合适的构件类型中。

③ 支持对构件的高效检索，可以根据用户的需求从构件库中发现合适的构件。在这里对用户需求的匹配，既包括精确匹配，也包括模糊或者近似的匹配。

④ 支持方便的、友好的用户管理和使用界面。

5.6.4 基于复用的开发过程和模型

1. 基于复用的开发过程

软件复用的运用在为软件开发带来巨大效益的同时，也改变了传统的软件开发过程。基于复用的软件开发过程由以下四个过程共同组成。

(1) 创建

软件复用的创建过程主要是指界定并提供可供复用的软件产品，以满足复用者的需要。这里所指的可供复用的软件产品是多种类型的，如源代码、测试用例、体系结构等。这些可供复用的软件产品的来源既可以是新开发的，也可以是从第三方软件生产商处购买的。

(2) 复用

复用过程是指使用由创建过程提供的或另外购买的可复用产品来开发新的软件系统。该过程主要包括如下几个步骤：

① 收集和分析用户的需求；

② 确定需要的可复用产品；

③ 在复用库中根据需要的可复用产品的特征属性查找相应产品；

④ 对可复用产品进行必要的调整和修改；

⑤ 设计并实现需要另加的软件部件；

⑥ 组装出完整的软件系统；

⑦ 对组装的软件系统进行测试。

(3) 支持

支持过程是指对可复用软件产品的获取、管理、维护、调整工作。此过程主要包括如下几个步骤：

① 对可复用产品使用复用库进行组织、管理，以方便用户的检索；
② 通告和分发可复用产品；
③ 为可复用产品提供必要的文档及使用说明；
④ 根据用户的反馈意见和缺陷报告对可复用产品进行修改、调整或创建新版本的可复用产品。

(4) 管理

管理过程包括计划、协调、资源管理、过程跟踪等工作。同时，管理过程还涉及安排新的可复用产品的开发工作，针对新的可复用产品的教育、培训工作，与软件开发中其他活动的协调工作等。

2. 开发模型

在经过领域分析和需求分析以及系统设计之后，可以在此基础上，初步确定系统的框架，划分出系统将要使用的构件及其功能。图 5.6 是一种开发模型的示意图。

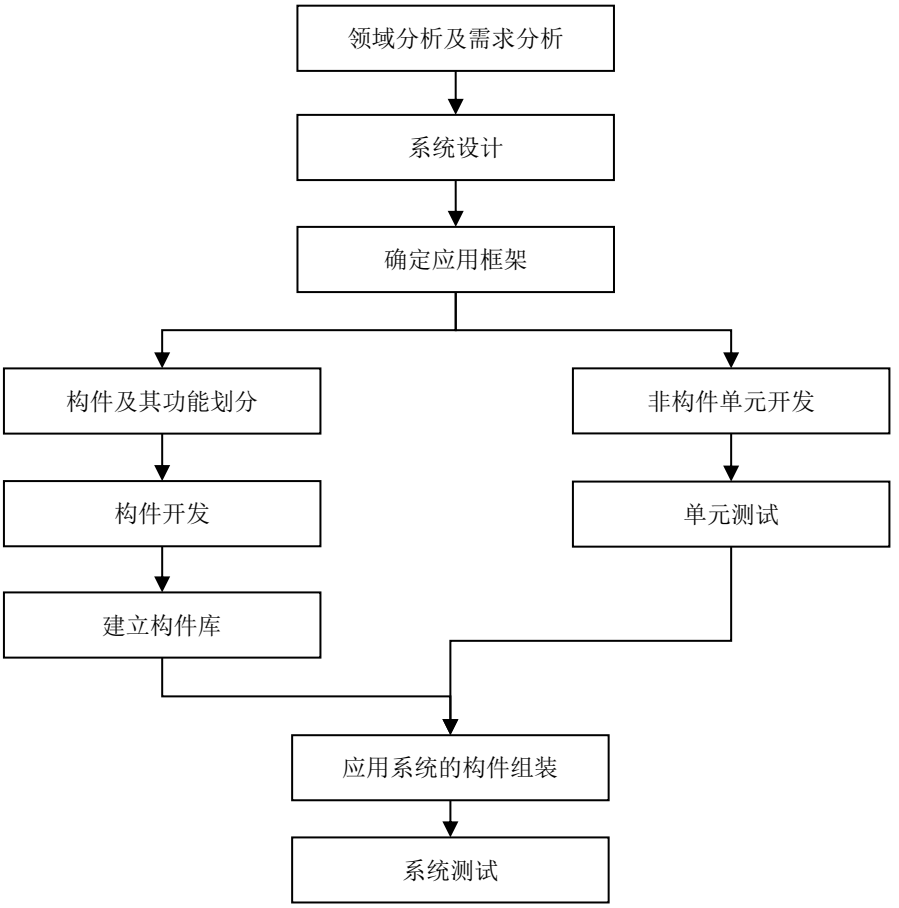

图 5.6 基于构件的开发过程模型

5.6.5 构件的组装和复用

构件的组装就是构件通过其接口交互而创建应用系统的过程。构件组装的过程主要包括三个步骤：

1. 构件的选取

根据软件系统的需求，在已有的构件库中选择适当的构件，或者自己开发构件。

2. 构件的自适应

选取的构件不一定满足复用上下文的需求，所以，在它们与其他构件连接之前，需要根据复用上下文对其作相应的适应性调整。

3. 体系结构的配置

通过相应的通信机制可以将构件连接起来。不同类型的通信机制，其具体实现也是不同的。

在基于构件的软件开发中，相关的两个基本的活动是面向构件的开发和基于构件的开发。前者是生产构件的过程，后者是利用构件组装新的应用系统的过程。例如，构件的选取属于前者，而构件的自适应和体系结构的配置则属于后者。

在构件组装的过程中，构件的自适应和体系结构的配置两个阶段是交织在一起的，而不是分开的、独立的。此外，构件的组装过程中常常会遇到组装不匹配的问题。我们可以从软件体系结构层次找出组装不匹配的问题所在并在实现中解决。

5.7 软件构造实例

在第三章中我们完成一个实际案例的设计部分。在本节中，我们将编码实现这个案例。

5.7.1 实现环境

1. 系统支撑平台

(1) 服务器端

- 操作系统：Windows 10
- Web 服务器：Apache 2.4
- 数据库管理系统（DBMS）：因为此应用系统的简单性，所以没有选用任何 DBMS，只用一系列 XML 文件保存数据。这些 XML 文件模拟了数据库中的表（Table）。

(2) 客户端

- 操作系统：无要求
- 浏览器：需要支持 HTML 5 和 CSS 3

2．设计模式框架

因为本系统很简单，所以本系统虽然采用了 MVC 设计模式，但没有选择复杂的 MVC 框架。我们选择一款实用的模板引擎 Smarty 来支持 MVC 设计模式。Smarty 用 PHP 实现，可以让程序员和界面设计者快速编写应用。

实际上，因为本系统非常简单，所以选择 Smarty 也略显笨重。不过，选择它主要是做一个示范，使读者能够领略到 MVC 设计模式的优点。

3．程序设计语言

本系统只针对服务器端脚本和页面的编码实现选择程序设计语言。

(1) 脚本

脚本设计语言选择 PHP。PHP 是一种面向对象的语言，易懂易学，容易上手，可以快速地搭建应用。

(2) 页面

编制页面用到三种语言或脚本：

- HTML。这是编写页面的标准语言。本系统采用了 HTML 5，它支持诸如<nav>、

<canvas>这样的新标签；

- JavaScript。用于改善用户交互体验。本系统采用了流行的 JS 库：jQuery，它的设计初衷是“write less, do more”，可以使代码编写更加快捷方便；
- Less。Less（http://lesscss.org/）可以算是一种脚本语言，是对 CSS 文档的一种动态包装。Less 文档的主体仍然沿用 CSS 的语法，但支持变量定义、混入函数(mixed-in)等高级特性，可以使系统设计更加灵活，例如实现界面换肤等。Less 文档需要编译方可使用。它的编译器是用 JavaScript 写成的小巧插件。可以事先将 Less 文档编译成静态的 CSS 文档，也可以直接下载到浏览器后在动态编译。如果是后者，则需要配置 Web 服务器的 MIME 类型。

4. 其他系统运行时支持

本系统的图表采用 SVG 格式，因此需要浏览器支持。目前流程的浏览器都支持 SVG 格式图形的显示。

5. 系统目录结构

本系统将源代码和数据分别存放在不同的目录/文件夹下：

root/

|----controller：控制器类源码（文件夹）

|----database：数据库文件。本系统实际应用的只是一个 XML 文档（文件夹）

|----model：模型类源码（文件夹）

|----renderer：渲染器类（文件夹）

|----script：JS 脚本（文件夹）

|----style：CSS/Less 文档（文件夹）

|----templates：页面模板（文件夹）

|----templates_c：编译后的页面模板脚本，由 Smarty 引擎自动生成（文件夹）

|----view：视图类源码（文件夹）

|----chart.class.php：用于产生 SVG 图片的脚本

|----index.html：系统入口文档

|----route.php：路由器启动脚本

|----router.class.php：路由器类脚本

|----router.json：JSON 格式的路由表

5.7.2 系统编码实现

本系统需要对 3.7.3 中提到的所有类、所有页面模板进行编码。以下内容主要展示一些主要的类和脚本。

1. 数据表

数据表由如下 XML 文档表示：

```
<sales>
    <data>
```

```
        <annual>
            <year>2012</year>
            <Q1>10</Q1>
            <Q2>9</Q2>
            <Q3>15</Q3>
            <Q4>6</Q4>
        </annual>
        … //省略了其他年份的数据
    </data>
</sales>
```

2. 模型类

(1) Model 接口类

Model 接口类是所有模型类的父类，它描述了模型的工作接口，其编码如下：

```
<?php
    interface Model
    {
        public function &query($queryCriteria = null); //返回数据集的引用
    }
?>
```

(2) Model_chart 类

Model_chart 类是专用于图表系统的模型类，它实现的 query()方法用于实际的数据查询，其编码如下：

```
<?php
    final class Model_chart implements Model
    {
        protected $dataset;

        public function __construct()
        {
            $xmldata = simplexml_load_file("database/sales.xml");
            foreach ($xmldata->data->annual as $data)
            {
                $year = (int)$data->year;
                $this->dataset[$year]['Q1'] = (int)$data->Q1;
                //其余数据处理以此类推
            }
        }
```

```
        public function &query($queryCriteria = null)
        {
            return $this->dataset;
        }
    }
?>
```

2.视图类

(1) View 抽象类

```
<?php
    abstract class View
    {
        protected $model;
        protected $renderer;

        public function __construct(&$model, &$renderer)
        {
            $this->model = $model;
            $this->renderer = $renderer;
        }

        final public function update()
        {
            $dataset = $this->model->query();
            $this->renderer->render($this->formatData($dataset));
        }

        abstract public function formatData($dataset);
    }
?>
```

View 抽象类描述了所有视图类的工作接口：

- 构造方法为其实例对象添加对指定模型对象和页面渲染器对象的引用；
- update()方法高度抽象了视图类与模型类、渲染器类的交互，其流程是：从模型对象查询数据→将数据转换为显示格式→将转换后的数据交给渲染器显示。这个方法不需要子类修改或覆盖；
- formatData()方法将其参数指定的数据集转换为渲染用数据集，其子类必须覆盖此方法。

(2) View_chart 抽象类

```
<?php
```

```
abstract class View_chart extends View
{
    protected $colors;

    public function __construct(&$model, $tpl)
    {
        $renderer = new Renderer_chart("$tpl.tpl");
        parent::__construct($model, $renderer);
        //获取渲染参数的代码略
    }
}
?>
```

此类对象在其构造方法中实例化指定的渲染器对象，该对象与指定页面模板关联。此外，还获取系统渲染参数，用于在渲染页面是指定主题颜色。

⑶ View_chart_table 类

这里只列出用于显示表格的视图子类。

```
<?php
    final class View_chart_table extends View_chart
    {
        public function __construct(&$model)
        {
            parent::__construct($model, __CLASS__);
        }

        public function formatData($dataset)
        {
            return array("data"=>$dataset, "color"=>$this->colors['color3']);
        }
    }
?>
```

在此类的 formatData 方法中，将数据集（来自于对 Model_chart 对象的查询）和渲染参数合成在一个新的数据集中以用于页面显示。因为 View_chart_table 视图只是用表格的形式简单呈现数据，所以没有对数据集做更多的处理。如果要绘制图形，则须按图表插件的格式要求对其进行处理。

⑷ 与视图类相关的代码

- 以下是配色方案文档的内容。这是一个 Less 文档，其中包含了三色方案中主色的定义，不包含其他颜色（白色、灰色）的定义。

```
@color1:rgb(0,106,115);
@color2:rgb(0,139,156);
```

```
@color3:rgb(49,165,185);
```

- 以下是与此视图类 View_chart_table 相关联的页面模板，其中包含了 Smarty 引擎要求的类 PHP 语法：

```
<style>
    th {background-color: {$color}; }
</style>

<br /><div style="font-size:1.2em;">近三年销售量表格</div><br />
<table class='tv'>
    <tr>
        <th width='25%'>年份\季度</th><th>Q1</th><th>Q2</th><th>Q3</th><th>Q4</th>
    </tr>
    {foreach $data as $rowkey => $row}
    <tr>
        <td>{$rowkey}</td>
        {foreach $row as $colkey => $colval}
            <td>{$colval}</td>
        {/foreach}
    </tr>
    {/foreach}
</table>
```

在实际应用的编写过程中，页面模板是由界面设计者而非程序员编写的。这样可以使二者更关注自己的工作。但这种模式的缺点是：界面设计者需要学习一种新的语言。即使这种语言比较简单，但对他们来说也许这仍是一项挑战。

3.控制器类

(1)Controller 抽象类

```
<?php
    abstract class Controller
    {
        protected $model;
        protected $view;

        public function __construct(&$model, &$view)
        {
            $this->model = $model;
            $this->view = $view;
        }

        final public function updateView()
```

```
        {
            $this->view->update();
        }
    }
?>
```

(2)Controller_chart 类

```
<?php
    class Controller_chart extends Controller
    {
        public function __construct($route_entry)
        {
            $m = new $route_entry["model"]();
            $v = new $route_entry["view"]($m);
            parent::__construct($m, $v);
        }
    }
?>
```

(5) Controller_chart_table 类

这里只列出用于显示表格的控制器子类。

```
<?php
    final class Controller_chart_table extends Controller_chart
    {
        public function __construct($route_entry)
        {
            parent::__construct($route_entry);
        }
    }
?>
```

4.渲染器类

(1)Renderer 接口类

```
<?php
    interface Renderer
    {
        public function render($dataset);
    }
?>
```

⑵Renderer_chart 类

Renderer_chart 类继承自具体的模板引擎类，同时实现了父接口类。

```
<?php
    final class Renderer_chart extends Smarty implements Renderer
    {
        protected $tpl;

        public function __construct($tpl)
        {
            $this->tpl = $tpl;
            parent::__construct();
        }

        public function render($dataset)
        {
            foreach ($dataset as $key=>$data)   $this->assign("$key", $data);
            $this->display($this->tpl);
        }
    }
?>
```

5. 路由器相关

⑴路由表

```
{
    "Controller_chart_table" : {
        "model" : "Model_chart",
        "view" : "View_chart_table"
    },
    "Controller_chart_bar" : {
        "model" : "Model_chart",
        "view" : "View_chart_bar"
    },
    "Controller_chart_line" : {
        "model" : "Model_chart",
        "view" : "View_chart_line"
    }
}
```

这是一个 JSON 格式的文档。

(2) 路由器类

```
<?php
    class Router
    {
        public function route()
        {
            $route_table = json_decode(file_get_contents("router.json"), true);
            $controller = $_GET['controller'] ?? "table";
            $controllerClass = "Controller_chart_{$controller}";
            $controller = new $controllerClass($route_table[$controllerClass]);
            $controller->updateView();
        }
    }
?>
```

路由器的功能是：根据用户的选择，为 Controller 查找到关联的 Model 和 View，然后启动该 Controller 去触发 View 的更新。可以看到，路由器实际上是一个类工厂。

路由器由如下代码启动：

```
<?php
    //其它辅助代码（PHP 的类自动载入功能等）略
    $router = new Router();
    $router->route();
?>
```

6.其他文档

其他文档包括 index.html、动态样式（.less）和 JavaScript 脚本，这里就不再展示了。

5.7.3 用户界面

图 5.7-5.9 分别是不同视图的显示结果。

图 5.7 表格视图

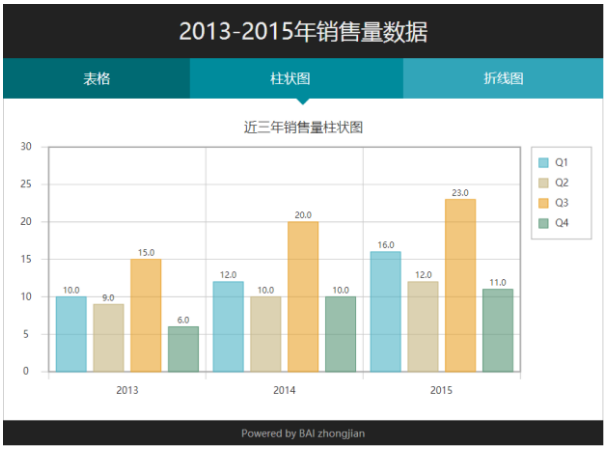

图 5.8 柱状图视图

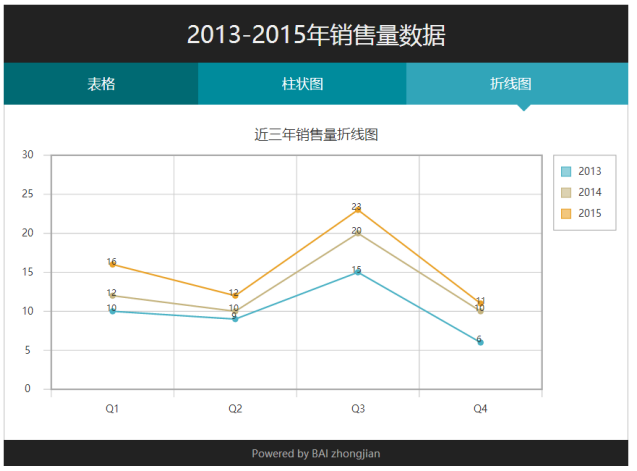

图 5.9 折线图视图

小 结

编码是软件实现的重要环节。一旦设计定稿之后，所有的编码工作全部都应遵循设计方案所定下的框架。这不仅可以使工作规范，同时也可以提高代码维护的效率。但软件编码又是一件容易带上浓厚个人色彩的工作，所以如何在这个阶段解决好个人编码风格和编码规范之间的关系是每个程序员都应当高度重视的问题。这是提高程序的可读性、可理解性，提高程序质量的关键。最后用实例来说明系统实现的过程。

习题五

1. 程序设计语言分为哪几类？
2. 程序设计语言由哪些成分组成？

3. 程序设计语言的选择对应用程序的开发有什么样的影响？

4.结构化程序设计的特点是什么？为什么要采用结构化程序设计？

5.对比面向对象程序设计，结构化程序设计有什么样的优势和劣势？

6.面向对象程序设计的优势是什么？

7.算法转换的指导原则是什么？

8.影响软件代码效率的因素有哪些？

9.请仿照 C 的代码审查项目提出针对于 Java 的审查项目表。

10.请找出你和其他同学/同事的一些程序作品，然后互相审查对方的代码，写出一份审查报告。

11.根据代码审查报告来修改你自己的代码，然后再做一次审查来检验你的结果。

12.软件复用有哪些优点？

13.请自行查阅关于 COM/COM+、CORBA、EJB 和 Web Service 相关的文献，看看他们各有什么特点。

14.如果你正在开发一个软件，那么你改如何考虑复用？

15.如果你恰好熟悉 PHP 开发，那么请你沿袭笔者的思路，将 5.7 中的案例完整的编码实现。