嵌入式软件与机器一起完全替换人们的工作。这样考虑的话，感觉像是现实世界被替换为了软件。而执行实际工作的是机器，嵌入式软件只是发送指令而已。嵌入式软件也绝不是在软件世界中直接临摹现实世界。

9.15 建模蕴含着软件开发的乐趣

本章以业务应用程序和嵌入式软件为例，介绍了建模的内容和成果。面向对象能够成为软件开发的综合技术，也可以说是因为建模技术不断发展，覆盖了整个上流工程。

建模并不像编程那样有编译器，而是人类对人类的思想进行整理的技术，而验证建模的也是人类。

笔者深入学习面向对象的契机就是感受到了建模的魅力。通过聆听用户，根据不确定的现实世界来编写固定逻辑的软件，虽然这项工作很困难，但却是一项有价值的、令人开心的工作。这也可以说是软件开发的最大乐趣。希望大家都能掌握建模技术，感受到与编程工作不同的业务分析和需求定义的乐趣。

深入学习的参考书籍

[1] 渡边幸三. 業務別データベース設計のためのデータモデリング入門 [M]. 东京：日本实业出版社，2001.

☆☆☆

[2] 渡边幸三. 販売管理システムで学ぶモデリング講座 [M]. 东京：翔泳社，2008.

☆☆☆

这两本书介绍了很多业务应用程序的建模实例和技术窍门。虽然不是介绍面向对象的书，作者在书中使用的也是独自设计的表示方法，但是对于从事业务应用程序上流工程相关工作的人来说，这两本是必读书。

[3] 椿正明. データ中心システムの概念データモデル [M]. 静冈：OHM 社，1997.

☆☆☆

作者常年从事业务应用程序领域的数据建模工作，该书总结了作者在多年的工作经验中积累的理论和表示方法。其中，“资源”“事件”“库存”“剖面”和“摘要”等概念文件类型（相当于 UML 中的 stereotype）思想非常恰当地描述了业务应用程序的本质。对于从事业务应用程序上流工程相关工作的人来说，这是一本必读书。

[4] 儿玉公信. UML モデリングの本質 [M]. 东京：日经 BP 社，2004.

☆☆

该书介绍了概念模型的创建步骤和技术窍门。通过列举图书馆、自动检票、库存管理和机票预约等 4 个示例，为读者介绍了概念模型和用例模型的创建过程和思考流程。此外，该书还提到了模式、重构、业务建

模及 OCL（Object Constraint Language，对象约束语言）等话题。

[5] 平泽章. UML モデリングレッスン——21 のパターンでわかる要求モデルのつくり方 [M]. 东京：日经 BP 社，2008.

☆☆

该书主要通过一些简单的练习题来讲解业务应用程序中 21 个典型的数据模型的模式。书中使用 UML 类图和对象图的变形版、状态机图表示模型，内容主要限定于概念层次的数据建模。

第10章 面向对象设计：拟人化和职责分配

本章的关键词

内聚度、耦合度、依赖关系、拟人化

热身问答

在阅读正文之前，请挑战一下下面的问题来热热身吧。

内聚度（cohesion）和耦合度（coupling）是评判软件构件独立性的两个标准，下面哪一项可被评判为优秀的设计？

A. 内聚度高、耦合度也高
B. 内聚度高、耦合度低
C. 内聚度低、耦合度高
D. 内聚度低、耦合度也低

B. 内聚度高、耦合度低

解 析

内聚度和耦合度是在面向对象成为主流之前的 20 世纪 70 年代，在结构化设计方法中提出的用于评判软件构件独立性的标准。

内聚度是评判软件构件所提供的功能互相结合的紧密程度的标准，结合得越紧密，内聚度越高，设计就越好。

耦合度是评判多个软件构件之间互相依赖的程度的标准，依赖程度越低，耦合度越低，设计就越好。

本章重点

本章以设计为主题，来介绍用于提高可维护性和可重用性的思想。这里首先介绍设计易于维护和重用的软件结构的三个目标。然后介绍以面向对象的思维方式进行这种设计的技巧，即将作为逻辑集合的软件拟人化，并进行职责分配。

该“软件的拟人化”与本书前面介绍的“现实世界与软件结构是似是而非的”从某种含义上来说是矛盾的。而在设计阶段，面向对象的两方面——归纳整理法和编程技术都必不可少。本书通篇讨论的现实世界和软件之间的沟壑的话题到本章就结束了。

10.1 设计的目标范围很广

我们先来思考一下设计整体是做什么的。

第 9 章中介绍了编程之前需要完成的三个阶段的工作，即业务分析、需求定义和设计。在最后的设计阶段，我们将讨论如何通过软件来实现需求定义中确定的计算机承担的工作范围。

虽然简单地称为设计，但是实际的工作还要分为几个阶段。典型的设计作业的流程是定义运行环境、定义软件的整体结构，以及设计各个软件构件。

首先应该定义运行环境，也就是选择并确定所采用的硬件和软件产品。在选择产品时，我们要考虑可靠性和运行效率等系统性能、技术和产品的成熟度、费用，运维的便捷性等各种因素。特别是在软件方面，我们需要确定操作系统、通信和数据库等各个领域中采用的技术和产品。

然后应该定义软件整体的结构。这里将整体分割为多个子系统。在嵌入式软件等应用程序中，在该阶段通常也会大致讨论线程的结构。

特别是在 OOP 的情况下，为了确保应用程序整体的标准化和质量，在确定整体共同的软件结构之后，通常会准备第 6 章介绍的框架。

接着在下一个阶段，我们会逐个确定用于实现应用程序各个功能的软件构件的规格和接口。在 OOP 中，这相当于确定类和方法的规格和接口（图 10-1）。

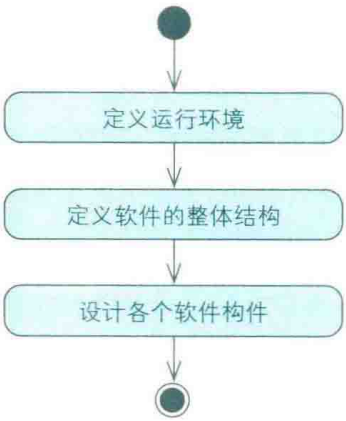

图 10-1　典型的设计作业的流程

设计作业的具体内容和技术窍门根据所采用的技术、编程语言或产品的不同而不同，也根据业务应用程序和嵌入式软件等应用程序性质的不同而不同。像这样，针对不同的运行环境和应用程序性质，设计的推进方法和技术窍门会有相应的变化，这一内容非常深奥。因此，这里将依赖于运行环境和应用程序性质的设计手法排除在外，只介绍定义软件的整体结构和设计各个软件构件时常见的思想。

10.2　相比运行效率，现在更重视可维护性和可重用性

设计的目标是什么呢？

最重要的是按照需求规格说明正确运行吧。如果无法实现用户期待的功能，那么，即使运行效率再高，即使使用非常优秀的设计模式实现了高扩展性的软件结构，也毫无意义。

而第二重要的目标，在过去应该是运行效率。在硬件性能低下的时代，

我们追求的是创建出运行速度尽量快、内存和硬盘等资源消耗尽量少的软件。而现在硬件性能显著提高，另外，应用程序的规模变大了，软件的寿命也变长了，因此，与运行效率相比，可维护性和可重用性变得更加重要。

那么，怎样才能提高可维护性和可重用性呢？

OOP 的类、多态和继承等结构就是用来提高可维护性和可重用性的。不过，它们终究只是工具，如果只是使用这些工具，并不会提高可维护性和可重用性，重要的是如何运用这些优秀的工具。

另外，第 6 章介绍的设计模式是应用 OOP 结构时的技术窍门集。而这些设计模式也只有在非常适用的情况下才会发挥威力，并不是可以适用所有情况。

为了提高软件的可维护性，需要让修改位置的定位变简单，使修改的影响范围变小。另外，为了便于重用，需要能够以子系统或构件为单位对软件进行分割。

这里，我们将易于维护和重用的软件结构的目标汇总为如下三点。

① 去除重复。
② 提高构件的独立性。
③ 避免依赖关系发生循环。

接下来，我们依次介绍这三个目标。

10.3 设计目标之一：去除重复

第一个目标是去除重复。

如果功能重复，那么相应地规模就会变大，测试就会变得很辛苦，理解起来也很困难。特别是修改重要问题时，可能会造成遗漏。在最开始编写程序时还好，如果过了一段时间后需要修改该程序，就容易忽视重复的地方，造成修改遗漏。

因此，在设计阶段，我们需要尽可能地避免功能重复。在新编写程序

时是这样，之后修改程序时也是如此。为了不影响既有程序，有时会进行“复制 / 粘贴式编程”，即将程序的一部分整体复制下来进行修改，但对于长期使用的软件来说，应该尽力避免这种简单的修改。

为了去除重复，传统编程语言提供的结构只有子程序（函数），用来实现步骤的公用化。而 OOP 中还提供了实现调用端公用化的多态、汇总类的共同部分并使用的继承等结构。这样一来，我们便可以创建之前无法实现的类库和框架等大规模的可重用构件，这在第 6 章中已经介绍过。

10.4　设计目标之二：提高构件的独立性

第二个目标是提高构件的独立性。

为了在日后能修改已经完成的软件，我们首先需要理解该软件，但仅凭一己之力是根本记不住大规模软件多达几万、几十万行的命令群的。那么，为了便于理解这样复杂的软件，我们应该怎么做呢？

一般来说，轻松理解复杂内容的诀窍就是“分割”，即将复杂的内容分割成较小的部分。

在软件中运用这个诀窍的话，就可以将整体看成由多个子系统或构件组成的。不过，只是简单地将软件分割成子系统或构件还不够，重点在于各个子系统或构件要独立性高。

如果能提高构件的独立性，那么子系统或构件的功能就会变得比较清晰，修改时也更容易确定修改位置。即使对某个构件或子系统进行了修改，也可以将该修改对其他部分的影响控制在最小限度。另外，我们也能很容易地将独立的构件拿出来，并在其他应用程序中重用。

> 为了提高可维护性和可重用性，重要的是使用独立性高的构件来组成软件。

作为提高构件和子系统独立性的思想，下面将介绍内聚度和耦合度两

个标准，它们在面向对象成为主流之前就已经被提出来了。

内聚度是评判软件构件所提供的功能互相结合的紧密程度的标准，结合得越紧密，内聚度越高，设计就越好。

耦合度是评判多个软件构件之间互相依赖的程度的标准，依赖程度越低，耦合度越低，设计就越好。

通俗地说，就是各个构件内部紧密结合，构件和构件之间不互相依赖（图 10-2）。打个比方，就是排除外人，内部紧密团结，将与周围人的交往控制在最低限度。

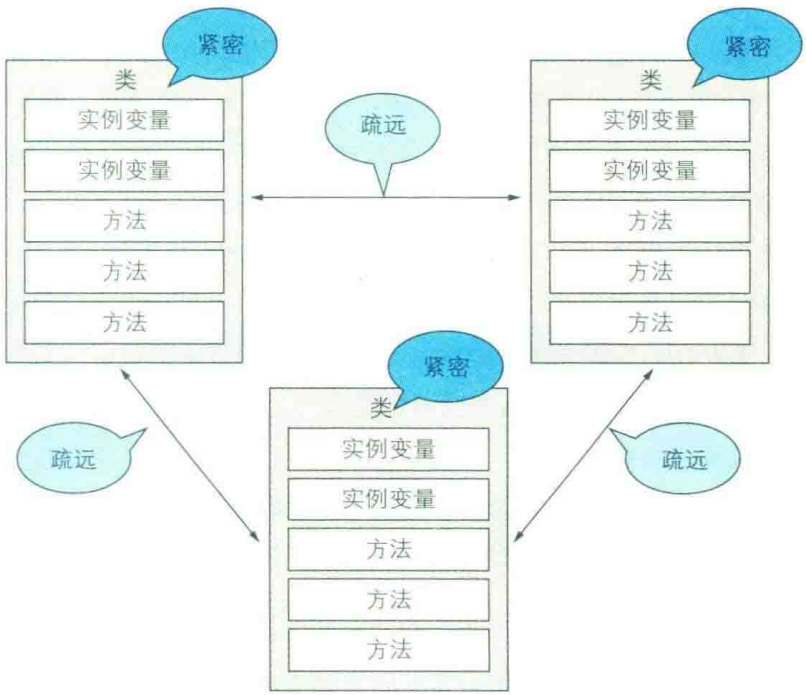

图 10-2　内聚度高、耦合度低的情形（以类为例）

在面向对象中，软件构件的基本单位是类，因此，内聚度高、耦合度低就相当于一个类中定义的功能（方法和变量）的含义密切相关，而类之间的交互较少。这种内聚度和耦合度的思想也适用于方法或包。

10.5　提高构件独立性的诀窍

不过，如果只是坚持强化内聚度、弱化耦合度的方针，在实际设计时会很难判断。下面，我们针对类和方法，来介绍几个提高构件独立性的具体窍门。

- **起一个能用一句话表示的名称**

  强化内聚度的最大诀窍就是起一个能用一句话明确表示类功能的名称（包和方法中也是如此）。如果命名不恰当，类中就可能包含结合不紧密的功能，在这种情况下，我们就需要讨论类的分割。俗话说“名如其人”，只有构件结合紧密，才能够起一个简洁明了的名称。通过起一个容易理解的名称，日后他人在确认该程序时，也容易理解。

- **创建许多秘密**

  类会将向外部公开的信息控制在最小限度。具体来说，就是使用第 4 章中介绍的类的隐藏功能。隐藏实例变量自不必说，那些看起来并不会直接使用的方法也应该隐藏。

- **创建得小一点**

  还有一个诀窍是，所创建的各个类和方法要尽可能地小。如果在一个类中定义过多的方法和变量，就难以理解整体是做什么的。

另外，有时使用 OOP 编写的方法只有几行，甚至一行。虽然并不是所有方法都要这么小，但是在使用 OOP 的情况下，一个方法的上限应该是二三十行。

相反，即使最开始创建得并不小，为了提高构件的独立性而对功能加以限制后，方法和类也就自然而然地变小了。

10.6 设计目标之三：避免依赖关系发生循环

内聚度和耦合度在面向对象成为主流之前就已经被提出，其重要性至今也没有发生改变，但在以 OOP 为前提的情况下，还有一个重要目标，那就是避免依赖关系发生循环。

其中，避免包的依赖关系发生循环尤为重要。这是设计时必须遵守的规则，其目的就是维持大规模软件中的秩序。

> 为了维持软件中的秩序，避免包和类的依赖关系发生循环非常重要。

下面我们来稍微详细地介绍一下依赖关系。

所谓依赖关系，就是某个构件使用其他构件。图 10-3 表示构件 A 依赖于构件 B，这也可以说，在编译构件 A 时需要构件 B。

图 10–3 构件 A 依赖于构件 B

而如果这种依赖关系发生循环，就会有问题。我们以三个构件的依赖关系发生循环为例来思考一下。如图 10-4 所示，如果存在依赖关系的循环，在修改或重用这些构件时就会发生问题。那么，会发生什么问题呢？

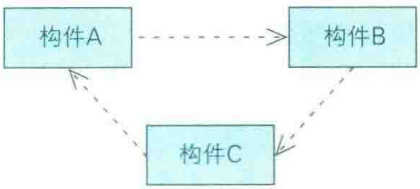

图 10–4 依赖关系的循环

我们先来介绍一下重用的问题。在依赖关系存在循环的情况下，这三个构件都将无法单独编译。因此，这些构件都无法单独重用，必须组合使用。

图 10-4 的结构在修改时也会发生问题。对某个构件有依赖关系，也就是会以某种形式使用该构件，因此，在修改该构件的情况下，就需要确认是否对使用端有影响。在图 10-4 中，无论对哪一个构件进行修改，都必须确认对 A、B、C 所有构件的影响。

我们再来看一个例子，如图 10-5 所示，构件 A 与构件 C 的依赖关系与图 10-4 相反。

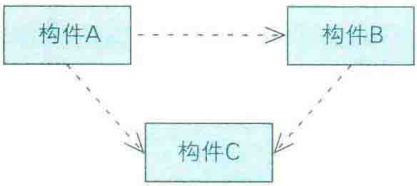

图 10-5　避免依赖关系发生循环

虽然只是掉转了一个依赖关系的方向，但是这样依赖关系就不循环了。这里，构件 C 是完全独立的，因此，我们可以将构件 C 单独拿出来重用。另外，虽然构件 A 依赖于构件 B 和构件 C，但是其他地方并未使用构件 A，因此，当修改构件 A 时，无须确认对其他构件的影响。像这样，为了提高可维护性和可重用性，避免构件的依赖关系的循环是非常重要的。

这种依赖关系的思想在结构化语言时代是不需要的。这是因为，如果循环调用子程序（函数），就容易形成无限循环，所以那时基本上不会出现循环结构。而在 OOP 中，由于引入了汇总多个方法的类、汇总多个类的包等结构，类之间或者包之间容易出现依赖关系的循环，所以我们需要谨慎地设计包和类，避免依赖关系发生循环。

另外，第 8 章中介绍的 UML 的图形表示规则中就非常注意这种依赖关系。UML 类图和包图中使用的四种关系如图 10-6 所示。它们的箭头形状不同，但其方向都表示依赖关系。虽然本书中没有详细介绍 UML 的绘制方法，但是依赖关系的方向在面向对象设计中非常重要，建议大家牢记该规则。

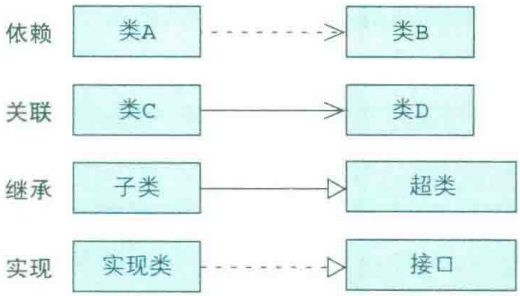

图 10-6 UML 类图和包图中的箭头表示依赖关系的方向

10.7 面向对象设计的“感觉”是拟人化和职责分配

前面我们介绍了易于维护和重用的软件结构的三个目标。

为了设计这样的软件，除了具体的技术之外，我们还需要某种“感觉”。由于是“感觉”，所以很难用文字来表达，但以笔者的经验来说，就是**拟人化**和**职责分配**[1]。

大规模软件非常复杂。当人们从事复杂的工作时，就会将人或组织的职责分开，与此相同，实现复杂功能的软件也会将职责分配给各个子系统或构件。

实际上，正如第 2 章中讨论的那样，现实世界和软件结构是似是而非的。尽管如此，在设计阶段，我们还是会考虑将作为逻辑集合的软件拟人化，参照现实世界中的人和组织，进行职责分配。

这并不是介绍 OOP 结构时的比喻，而是进行面向对象设计时的“感觉”。

在评审设计时，我们有时会听到“该类知道这个信息，但不知道那个信息，因此，无法拥有那个功能”“这家伙是这种作用，所以调用这种方法”等。这些说法，特别是后一种将类称为“这家伙”的拟人化表示，在

① 在面向对象的术语中，确定软件中类的作用一般用“责任（responsibility）分配”来表示，但由于该词让人感觉比较死板，所以本书中改用“职责分配”一词。