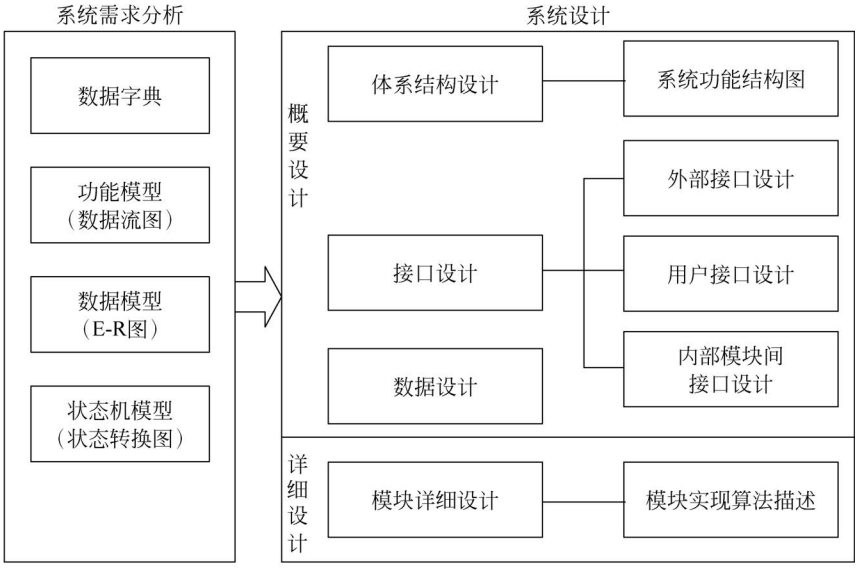
第5章 CHAPTER 5 系统设计

第4章系统需求分析围绕“系统要做什么”展开需求分析和建模，在此基础上，进入软件生命周期的下一个重要阶段：系统设计。系统设计阶段的基本任务是回答“系统要怎么做”这个问题。

本章将重点介绍两种系统设计方法：结构化设计方法和面向对象设计方法，主要内容如图5-1所示。

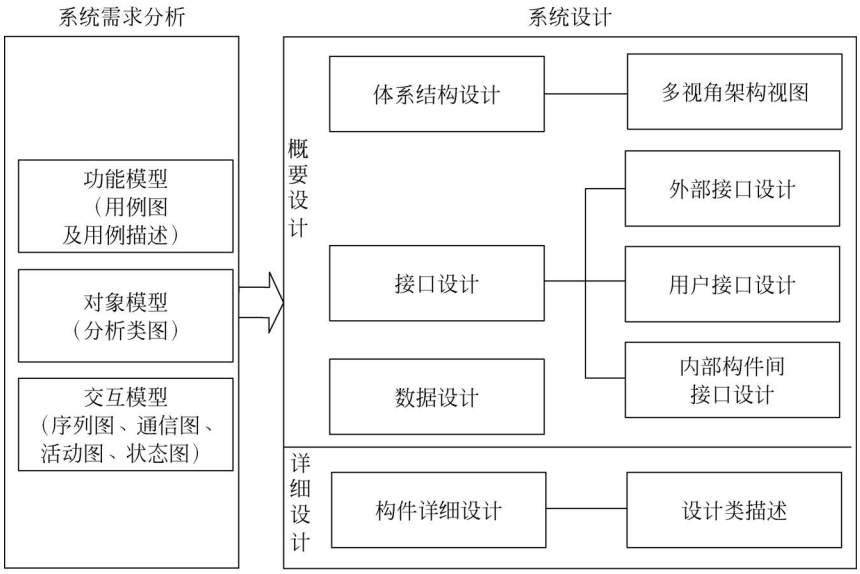
图5-1(a)展示了结构化系统设计方案由结构化系统需求分析阶段的成果推导而来；图5-1(b)展示了面向对象系统设计方案由面向对象系统需求分析阶段的成果推导而来。无论采用何种软件工程方法，系统设计的主要内容都包括体系结构设计、接口设计、数据设计和模块/构件详细设计。

本章将以上述四部分内容为主线，首先介绍软件设计的相关概念、通用的软件体系结构、接口设计和数据库设计基本概念；然后介绍结构化设计与面向对象设计中体系结构设计和详细设计所使用的特定概念和方法；最后选用面向对象系统设计方法，以智慧社区养老系统为例，阐述案例的体系结构设计、接口设计、数据设计和模块/构件详细设计。



(a) 结构化设计

图5-1　系统设计主要内容



(b) 面向对象设计

图 5-1　（续）

5.1　系统设计概述

1. 软件设计的概念

软件工程中的系统设计特指软件系统设计，也称为软件设计。在运用各种软件设计技术之前，应首先理解软件设计的概念。软件设计是从软件需求规格说明书出发，形成具体软件设计方案的过程。软件设计是将系统需求转换成软件制品的必经途径，在软件需求和软件实现之间起到了桥梁作用。也就是说在需求分析阶段明确软件是“做什么”的基础上，软件设计主要解决软件“怎么做”的问题。在真正理解用户需求之前，不可能得到正确的系统设计。需求确认之后，系统设计是由需求自然推导而来的结果，系统设计方案就藏在用户需求里。

软件设计过程使用的一个关键技术是“分解”：把一个较大的问题分解成一些较小的、可管理的单元，每一个单元都可以单独处理。分解技术是许多软件工程方法的核心。例如：结构化设计使用分解技术后产生的单元称为模块，面向对象设计使用分解技术后产生的单元称为构件。

软件设计是一个逐步迭代的过程。在这个过程中，首先在较高抽象层次上描绘软件系统的概貌，构建出一幅软件系统的“蓝图”，之后随着迭代的不断深入，在较低的抽象层次对系统蓝图的组成部分进行细化，为下一阶段的编码提供依据。

从软件工程管理的角度看，软件设计包括两个阶段：概要设计阶段和详细设计阶段。概要设计阶段主要完成体系结构设计、接口设计、数据设计；详细设计阶段主要完成构件/模块的细化。

2. 软件设计方法对比

下面从不同方面对结构化设计和面向对象设计两种方法作对比。

（1）从设计的主要内容上比较。两种方法在进行体系结构设计时所采用的方法及成果不同；接口设计和数据设计基本相同；结构化详细设计完成模块的设计，面向对象详细设计完成构件的设计；模块设计着重于对函数的描述，构件设计着重于对类的描述。

(2) 从设计思想上比较。结构化设计以数据流图为推演基础,使用自顶向下、模块化、逐步求精的方法,通过逐层分解来构建系统结构图,因此模块间的关系局限于信息流,无法体现模块间的继承、关联关系等;同时结构化设计从系统功能需求的实现入手,随着用户需求和软、硬件技术的不断发展变化,功能模块的划分更多地依赖于经验,模块改动容易引起系统的根本性变化。在面向对象设计中,设计的核心是描述现实世界事物属性和行为的对象,对象间的关系有丰富的表达方式,能够体现事物之间复杂的关系;面向对象的设计通过设计类与类之间的关系来解决实际问题,设计良好的类图,可以有效地提升系统对于需求变化的适用性,减缓系统的腐化。

(3) 共同特点。系统设计模型都是从需求模型转化而来的;两种设计都描述了功能性构件和它们之间的接口;两种设计都采用了分割和逐步求精的方法;具有相同的设计质量评估原则。

3. 软件设计的任务

软件设计阶段主要包括如下几方面任务。

1) 体系结构设计

体系结构设计描述了软件系统的框架,定义了满足系统需求的软件结构元素(构件/模块)和元素之间的关系。首先根据分析模型选择一种适用于目标软件系统的体系结构风格,然后将系统划分为若干个子系统,并将这些子系统划归体系结构中的某部分。

2) 接口设计

接口描述了软件和外部系统及外部设备之间、软件和用户之间及软件内部构件/模块间的通信联系。

3) 数据设计

系统设计阶段应对要存储的数据及其结构进行设计,需求分析阶段建立的领域模型和分析类中的实体类都为数据设计提供了依据。数据存储的方式既可以选择文件,也可以选择数据库。大部分情况下,数据存储都会采用数据库,本章的数据设计部分将主要介绍数据库设计。

4) 构件/模块设计

构件/模块设计详细描述了每个构件/模块内部的数据结构、处理逻辑的算法细节,细化每个构件/模块的接口。在结构化设计方法中,一般将体系结构设计中的一个功能部件称为模块,模块可以是子程序、过程、函数等不同层次的表示,底层元素为函数;在面向对象设计方法中,一般将体系结构组成部分中能够独立运行的部件称为构件,底层元素为类。

4. 软件设计的指导原则

软件设计主要的指导原则如下所述。

1) 模块化

在解决复杂问题时,“关注点分离”是被广泛使用的一种系统思维方法,是计算科学和软件工程在长期实践中确立的方法论之一,在业界更多的时候以分而治之的形式出现,即将整体看成是各部分的组合体并对各部分分别加以处理。大体思路是:先将复杂问题做合理的分解,再分别研究问题的不同侧面(关注点),最后综合各方面的结果,组合成整体的解决方案。

模块化原则是“关注点分离”最有代表性的具体设计原则之一。按照模块化原则,软件系统可划分为独立命名的、可以独立访问的构件(在传统软件工程中称为模块),把这些构件集成到一起可以满足用户各种需求。

在进行软件模块化分解的过程中，应注意模块划分层次和数量的平衡点，避免出现模块划分过少或过度模块化的问题。“信息隐蔽”原则认为一个设计良好的模块应该具有的特征是：“每个模块对其他所有模块都隐蔽自己的设计决策”。也就是说，应该把实现独立功能所必需的数据结构和算法都包含在模块内，其他模块无需访问这些信息，模块之间的联系仅限于实现软件功能所必需的信息交流。

2）模块功能独立

模块功能独立是“关注点分离”“模块化”“信息隐蔽”等概念的直接产物。模块功能独立性是指软件设计在划分模块时，应使模块的功能专一，尽量避免和其他模块有过多的交互，可以实现模块功能独立。

模块功能独立性可以通过内聚性和耦合性进行评估。内聚性用来度量一个模块内部元素之间结合的紧密程度。模块内部元素间的联系越紧密，内聚性就越高，与其他模块之间的耦合性就越低，模块功能独立性就越高。耦合性用来度量模块之间互相连接的紧密程度。模块与模块之间的连接越紧密，耦合性就越高，模块功能独立性就越低。

3）逐步求精

“逐步求精”是一种自顶向下的设计策略，软件设计从顶部的软件体系结构开始，对体系结构中的各个组成元素逐步细化，直到提供足够的数据细节和过程算法细节，用程序设计语言能够实现为止。逐步求精大致遵循下述过程：将系统分解为子系统→将子系统划分为各个构件/模块→细化各个构件/模块中的类/函数。每次细化都会提供实现阶段所需的更多细节。

4）复用性

软件复用是提高软件生产力和质量的一种重要技术。早期的软件复用主要是代码级复用，被复用的知识专指程序，后来扩大到领域知识、开发经验、体系结构、需求、设计、代码和文档等软件开发的各方面。

软件复用的实现有三种途径：

第一种途径是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同一系统的其他部分或是新系统的设计中；

第二种途径是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中；

第三种途径是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，以便为将来的复用提供服务。

采用适当的设计方法能够有效地提高构件的可复用性。出发点是将构件进行封装，尽可能减少构件对外部的依赖。在这方面，传统的结构化功能分解方法强调构造低耦合、高内聚的模块，保持模块之间清晰的接口；面向对象方法中的开-闭原则、里氏代换原则、依赖倒转原则、接口隔离原则、合成/聚合复用原则等设计原则能够为构件复用提供更好的支持。

5. 软件设计质量评估

在整个软件设计过程中，应采用一系列技术评审手段来评估软件设计质量，以减少软件实现阶段修正错误所付出的成本。一个良好的软件设计应具备 3 个特征：

（1）应该能回溯所有在需求模型中明确提出的需求，并能满足干系人所期望的相关利益。

（2）应该提供软件系统的全貌，并从实现的角度对数据、功能、行为做出说明。

（3）对于软件编码人员、软件测试人员和软件维护人员，设计应该是可读的、可理解的指南。

5.2 软件体系结构

体系结构的概念来自“Architecture”,对这个单词的翻译有不同的译法,包括架构、构架、体系结构等。对于“Software Architecture”一词,学术界一般译为“软件体系结构”,在业界更习惯称之为“软件架构”,对应的有“软件架构师”的职位。软件架构师会针对开发系统的特点,选择合适的体系结构风格。软件体系结构作为从软件设计抽象出来的一门新兴学科,目前已经成为软件工程一个重要的研究领域。

作为软件体系结构最早的研究者,Mary Shaw 和 David Garlan 在《软件体系结构》中提到:“从第一个程序被划分为模块开始,软件系统就有了体系结构。同时,程序员已经开始负责模块间的交互和模块装配的全局属性。优秀的软件开发人员经常采用一个或多个体系结构模式作为系统组织策略。”

Len Bass、Paul Clements 和 Rick Kazman 在《软件架构实践(第二版)》中对软件体系结构给出了如下定义:“程序或计算系统的软件体系结构是指系统的一个或者多个结构,它包括软件构件、构件的外部可见属性以及它们之间的相互关系。外部可见属性则是指软件构件提供的服务、性能、使用特性、错误处理、共享资源使用等。”这一定义强调了“软件构件”在体系结构表述中的重要作用。

5.2.1 体系结构风格

软件体系结构风格(Software Architecture Style)是用于描述某一特定应用领域中系统组织方式的惯用模式,它促进了设计复用与代码复用。体系结构风格定义了一个系统家族,即一个体系结构定义一个词汇表和一组约束。词汇表中包含一些构件和连接件类型,而这组约束指出系统是如何将这些构件和连接件组合起来的。体系结构风格反映了领域中众多系统所共有的结构和语义特性,并指导如何将各个模块和子系统有效地组织成一个完整的系统。

在进行体系结构设计时,遵循某种体系结构风格的好处体现在以下几方面:

(1) 能够使各种背景的系统参与者更易理解体系结构设计方案,便于沟通、建立共识、加快体系结构选型。

(2) 可以快速地明确体系结构需要复用的构件,形成框架。

(3) 可以提升开发效率,规避风险。

Mary Shaw 和 David Garlan 总结了 5 种被广泛接受的体系结构风格类型,如表 5-1 所示。

表 5-1 典型的体系结构风格分类

序号	风格类型	体系结构子风格
1	数据流风格	批处理、管道/过滤器
2	调用/返回风格	主程序/子程序、面对对象风格、层次风格
3	独立构建风格	进程通信、事件系统
4	虚拟机风格	解释器、基于规则的系统
5	仓库风格	数据库系统、超文本系统、黑板系统

下面从表 5-1 中选取几种典型的体系结构子风格进行介绍。

1. 管道/过滤器风格

管道/过滤器风格为处理数据流的系统提供了一种结构,每个处理步骤封装在一个过滤器模块中,数据通过相邻过滤器之间的管道进行传输。当输入数据经过一系列模块的变换形成

输出数据时，可以应用这种体系结构。管道/过滤器风格如图 5-2 所示。

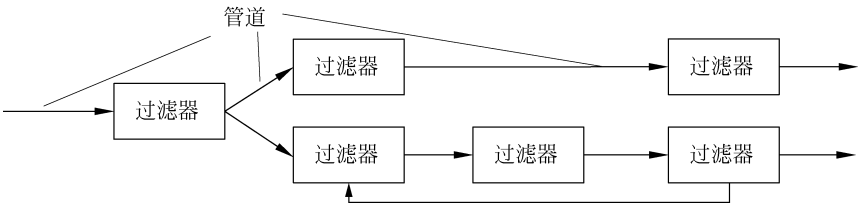


图 5-2　管道/过滤器风格

在图 5-2 中，每个模块都有一组输入和一组输出。每个模块从它的输入端接收输入数据流，在其内部经过处理后，将结果数据流送到输出端。每个模块称为“过滤器”，各模块之间的连接器充当了数据流的导管，将一个过滤器的输出传到下一个过滤器的输入端，这种连接器称为“管道”。

例如：Servlet 2.3 规范中提供的过滤器(Filter)就是管道/过滤器体系结构在 J2EE 中的具体应用。过滤器的工作机制如图 5-3 所示。

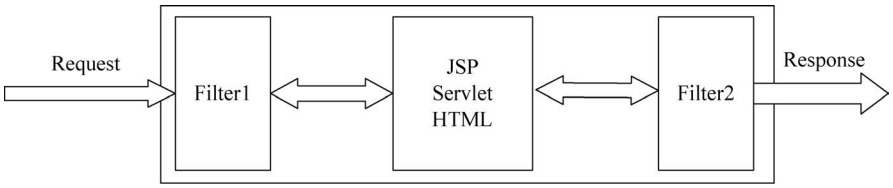


图 5-3　管道/过滤器风格应用示例

在图 5-3 中，有两个位置设置了过滤器：第一个是在客户端使用过滤器(Filter1)，使得请求(Request)在到达 Web 资源(JSP、Servlet、HTML)前被拦截并进行相应处理；第二个是在请求资源反馈之前使用过滤器(Filter2)，拦截响应(Response)进行处理，将处理后的结果反馈给客户端。因此，过滤器的特性为处理某些特殊的问题提供了很好的解决方案。例如：防盗链、字符编码处理、日志记录、数据加密、过滤黑词等。

2. 主程序/子程序风格

主程序/子程序风格是结构化程序设计的一种典型风格，它从功能分解的角度来设计系统，通过把问题逐步分解和细化，形成整个系统的体系结构，如图 5-4 所示。

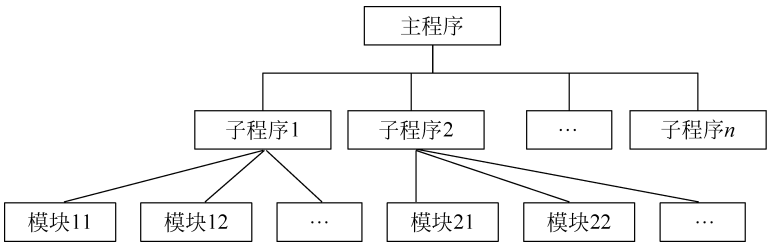


图 5-4　主程序/子程序风格

在图 5-4 中，主程序调用子程序，子程序将调用结果返回给主程序。同时，子程序也可以继续划分为模块，增加调用过程的层次性。主程序运行结果的正确性取决于其下属的子程序和模块执行结果的正确性。

3. 层次风格

层次风格是在系统设计和开发时，把系统组织成一系列的层次结构，如图 5-5 所示。系统层次间工作界限清晰，按适当次序放置。每一层为上层提供一组服务，并调用下层提供的服

务,不允许较高层次直接越级访问较低层次。

在图 5-5 中,层与层之间通过接口联系,一个接口发生改变,只有毗邻的层会受到影响。由于每一层的变化最多只影响两层,同时只要给相邻层提供相同的接口,就允许它用不同的方法实现,为软件复用提供了强大的支持。

4. 数据库系统风格

数据库系统风格包含一个数据库和若干其他构件,数据库位于体系结构的中心,其他构件访问数据库并对其中的数据进行增删改等操作,以数据存储为中心的数据库系统风格如图 5-6 所示。

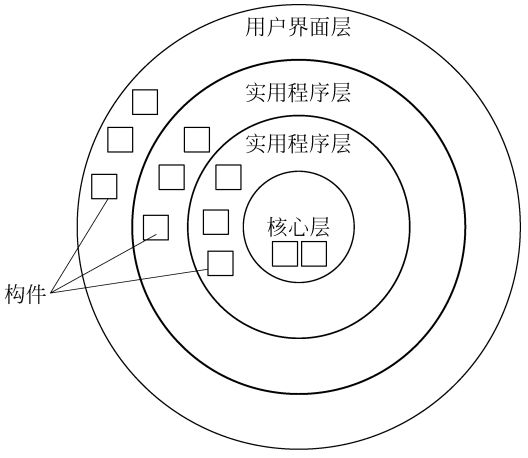


图 5-5 层次风格

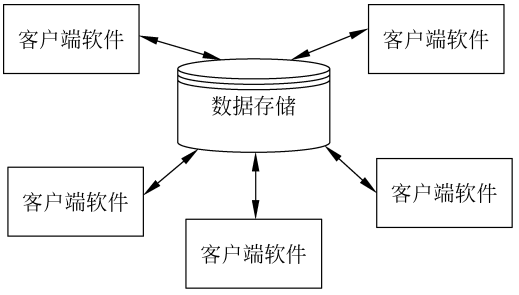


图 5-6 数据库系统风格

在图 5-6 中,不同客户端软件共享数据模型,访问数据存储中心。客户端软件之间无需数据转换,无需考虑数据如何集中管理;客户端软件在访问数据时,独立于其他客户端软件的动作。因此,新的客户端软件在加入到体系结构中时,无需考虑其他的客户端,从而促进了系统的可集成性。

5.2.2 体系结构模式

1. 模式定义

Dirk Riehle 和 Heinz Zullighoven 在《在软件开发中理解和使用模式》一文中,给出了这样一个定义:模式是从解决具体问题抽象出来的,这种具体问题在特定的上下文中重复出现。也就是说,模式是解决某一类问题的方法论。模式描述了一个在某种上下文环境中不断出现的问题,以及该问题的解决方案核心。通过这种方式,可以对一种重复的问题采用重复的解决方案。

然而,模式不仅仅是解决方案。在模式中,问题出现在某种特定的上下文环境中,并且包含各种互相竞争的关切。目标解决方案包括对这些关切的平衡,这种关切在模式中称为“作用力”,这些作用力互相牵制。因此,模式不能简单地看作是“特定场景的解决方案”,其实质是:在特定场景下,对各方面关切进行平衡后得出的解决方案。

值得注意的是,解决一个问题可能有多个可选择的解决方案,这些解决方案各有偏重,针对不同的关切可能有不同的选择,没有哪个方案是万能的。从模式的定义可以看出,模式是一种在平衡了各方关切、权衡了各种利弊后的解决方案,一旦作用力间的平衡被打破,这个解决方案就可能不再成立。

2. 软件模式的层次分类

不同的领域有不同的模式，建筑领域有建筑模式，软件设计领域有软件模式。软件模式按照不同的层次类型分为如下 3 种。

1）体系结构模式

体系结构模式是系统的高层次策略，涉及系统整体结构和大粒度的构件。它是开发一个软件系统的基本设计决策。体系结构模式的好坏影响系统总体布局和框架结构。

目前广泛使用的一种体系结构模式是 MVC 模式，其目标是将软件的用户界面和业务逻辑分离，使代码具有更高的可扩展性、可复用性、可维护性以及灵活性。

将 MVC 模式应用于基于 Java Web 的系统，其工作原理如图 5-7 所示。下面以产品信息管理为例，说明 MVC 模式在 Java Web 应用中的工作原理。

（1）用户通过浏览器访问服务器中的视图（View）部分，通过视图发起各种业务处理请求（产品信息的增删改查请求）。在 Java Web 开发中，常用的视图技术包括 HTML、CSS 和 JSP 等。

（2）通过视图发起的请求被控制器（Controller）接收。在 Java Web 开发中，控制器的底层为 Servlet。本例由 ProductServlet 类作为控制器接收来自视图的请求。

（3）控制器调用相应的模型（Model）处理请求。在图 5-7 中，模型包含了业务逻辑（Service）、数据操作（Dao）和数据对象实体（Entity）三部分。本例由 ProductService 类、ProductDao 类和 Product 类组成模型部分。

（4）模型访问后台数据库，根据请求完成相关数据操作（产品信息的增删改查操作），获取操作结果。

（5）模型将执行完毕后的操作结果返回给控制器，控制器再将结果返回给相应的视图。

（6）视图渲染数据后，将最终响应结果通过浏览器展示给用户，实现人机交互的过程。

在 MVC 模式下，控制器的作用是“接收请求，返回响应”，是视图和模型间的桥梁，控制器实现了视图和模型的代码分离。

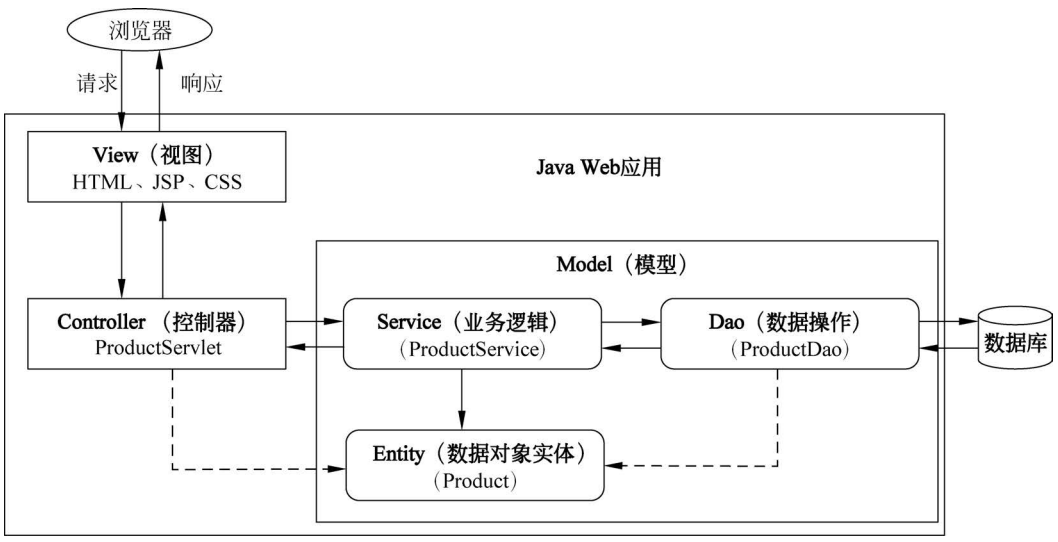


图 5-7　Java Web 应用中的 MVC 模式

深入思考 5.1　在图 5-7 中，业务逻辑（Service）和数据操作（Dao）的区别在哪里？控制器是否能直接调用数据操作（Dao）？

参考答案：请参见微课视频 5-1。

2）设计模式

设计模式是中等尺度的结构策略。设计模式定义了一些大粒度组件的行为和组件之间的关系。设计模式的好坏不会影响到系统的总体布局和总体框架。

经典的设计模式分为三大类：

- 创建型模式，共五种：工厂方法模式、抽象工厂模式、单例模式、建造者模式、原型模式。
- 结构型模式，共七种：适配器模式、装饰器模式、代理模式、外观模式、桥接模式、组合模式、享元模式。
- 行为型模式，共十一种：策略模式、模板方法模式、观察者模式、迭代子模式、责任链模式、命令模式、备忘录模式、状态模式、访问者模式、中介者模式、解释器模式。

一个体系结构模式常常可以分解成多个设计模式的联合使用。例如：MVC 体系结构模式常常包括中介者模式、策略模式、组合模式、观察者模式等。

3）代码模式

代码模式是特定的范例和与特定语言有关的编程技巧，是处理特定设计问题的实现。代码模式的好坏会影响中等粒度组件内部、外部的结构或行为的底层细节。较为典型的代码模式为双检索模式等。

3. 体系结构模式与体系结构风格的区别

体系结构风格与体系结构模式在概念上通常可以互用，但是它们仍然存在不同。

（1）体系结构风格是施加在整个系统设计上的，为整合系统所有构件建立的一种结构。体系结构风格的转换会导致软件结构的根本性改变，包括对构件功能的重新分配。

（2）体系结构模式涉及的范围要小一些，它更多应用在体系结构的某一方面，是为了解决某个特定的问题而形成的解决方案模板。

大多数应用系统都符合特定领域或特定类型，适用于某一种或多种体系结构风格。在某种体系结构风格中遇到的一系列常见问题，可以用具体的体系结构模式来处理。

4. 体系结构（架构）与框架的区别

在企业实践中，通常把学术界中的“软件体系结构”称为“软件架构”。

1）架构和框架处于不同的抽象层次上

软件架构处于较高的抽象层次上。架构是系统的顶层设计，它将系统划分为一些各自独立的构件，这些构件通过规定的接口传递信息。

而框架则是定义好的一套系统结构，这套结构包含了具体的类和对象，规定了它们的主要责任及类之间的协作、控制关系。所以框架是针对某个问题领域的通用解决方案，对开发工作起到提高效率，指导和规范开发活动的作用。

2）架构不是软件，而框架是软件

架构不是软件，它描述的是软件系统的总体设计决策，更多的是以模型、图文说明的表现形式存在。软件架构决策涉及如何将软件系统分解成不同的部分、各部分之间的静态结构关系和动态交互关系等。

框架是一种特殊的软件，它不能提供完整的解决方案，而是为构建解决方案提供良好的基础。框架不能直接使用，需要二次开发，框架中的服务可以被应用系统直接调用，框架的表现形式除了模型及图文说明之外，还有实实在在的可复用的代码。例如：Java Web 开发前端常用的框架有 JQuery、Angular、React、Vue，后端常用的框架有 Spring、SpringMVC、MyBatis、

Spring Boot 等。这些框架都有可下载的源码，安装后即可使用框架提供的各种服务。

总之，架构是系统蓝图，是对软件系统高层次的定义和描述；框架是解决方案，是提高系统质量和开发效率的半成品。框架比架构更具体，通过框架实现了架构的落地，对于同一个软件体系结构可以通过多种框架实现。

5.2.3　常见的软件架构

从软件架构设计师的角度来看，软件架构是一套构建软件应用系统整体结构的各种设计准则。通过这套设计准则，软件架构设计师可以把一个复杂的软件应用系统划分为一些相对独立的子系统，通过系统架构设计对各种系统应用要求和功能实现问题提出合理的解决方案。

1. 分层架构

分层架构是运用最为广泛的架构模式。分层架构体现了软件设计指导原则中的“关注点分离”原则，也就是通过层(Layer)将软件系统分离为不同的关注点。分层架构里的层是平行的层次，每一层都承担了软件系统运行所需的功能(例如：展示层承担人机交互、业务逻辑层承担业务功能实现等)。分层架构中层与层之间是一种松散耦合的关系，层与层间的依赖性较低，层与层之间一旦定义好统一的接口，就可以被各个模块所调用，有利于各层逻辑的复用。

如果没有分层的体系结构设计，所有代码都在同一层，系统仍然可以运行。那么为什么仍然要对系统分层呢？这是因为，随着系统规模逐步扩大，层次分离可以让不同的工程师或开发团队只关注架构中的某一层，实现在系统的不同层次并行工作，而不用担心影响其他人的工作，便于实现分工合作。

在分工合作的场景下，各开发团队之间不再需要对系统的其他层有深入了解，只需要完成三项任务：调用接口从上一层获取本层所需的数据；使用这些数据；数据处理完毕后，提供接口将处理后的结果数据传递给下一层。

分层架构没有规定系统应分成几层，下面按照分层架构的历史演变进行介绍。

1) C/S 架构

C/S(Client/Server)架构称为客户机/服务器架构。在这种架构下，不同的应用需要在客户端安装不同的软件。随着技术发展，C/S 架构也不断演变，先后出现了二层 C/S、三层 C/S 架构，如图 5-8 所示。

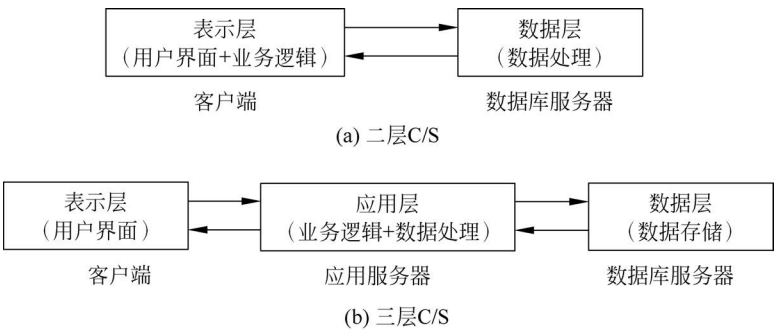


图 5-8　C/S 架构

早期的二层 C/S 架构主要特点是胖客户端，如图 5-8(a)所示。客户端是表示层，包含用户界面的人机交互和业务逻辑处理两大部分；服务端是数据层，主要负责数据处理。二层 C/S 架构交互性强、网络通信量低、响应速度快、利于处理大量数据。但是这种架构不利于扩展，升级、维护和管理的难度较大，客户端软件每次更新后都需要重新安装。

后来出现的三层 C/S 架构主要特点是瘦客户端,如图 5-8(b)所示。客户端是表示层,只包含用户界面部分和少量的业务逻辑;主要的业务逻辑以及与后台数据库间的数据交互处理被独立为应用层,部署在应用服务器上;服务端是数据层,主要负责数据存储。三层 C/S 架构使得软件的分层更加明晰,便于开发和维护。美工人员可以只针对表示层进行用户界面(User Interface,UI)设计,而程序开发人员则可以专注于应用层的功能设计和编码实现。由于业务逻辑从客户端分离出来,三层 C/S 架构的升级维护相对二层 C/S 架构更容易些。

2) B/S 架构

B/S(Browser/Server)架构称为浏览器/服务器架构。在这种架构下,不同的应用不需要在客户端安装软件,只需要客户端具备浏览器即可。它是随着网络技术的兴起,对 C/S 架构的一种变化或者改进的架构,如图 5-9 所示。

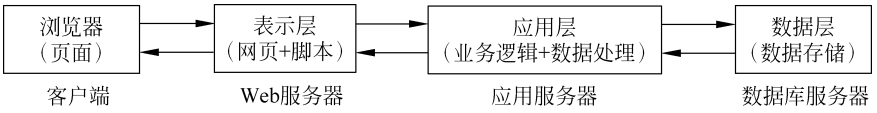


图 5-9 B/S 架构

在 B/S 架构下,用户通过浏览器访问网页完成人机交互;用户界面一般由网页和脚本语言组成,部署在 Web 服务器上;大部分的业务逻辑以及与后台数据库间的数据交互处理被独立为应用层,部署在应用服务器上;服务端是数据层,主要负责数据存储。B/S 架构下,所有的客户端只是浏览器,不需要做任何的维护,软件系统维护升级的工作只针对运行在服务器上的软件,基本可以做到对用户不产生影响,同时大大减少了人力、时间成本,因此成为目前的主流架构。

在实践中,多层 C/S 架构或多层 B/S 架构实际上是对应用层的细分。例如 Java 开发的系列 Spring 框架把应用层又细分为了控制层(Controller)、业务层(Service)、数据访问层(Dao)。

2. 面向切面编程架构

面向切面编程(Aspect Oriented Programming,AOP)是一种通过预编译方式和运行期间动态代理实现功能统一维护的技术。利用 AOP 对各业务逻辑进行隔离,从而降低业务逻辑之间的耦合度,可提高程序的可复用性,同时提高开发效率,AOP 原理如图 5-10 所示。

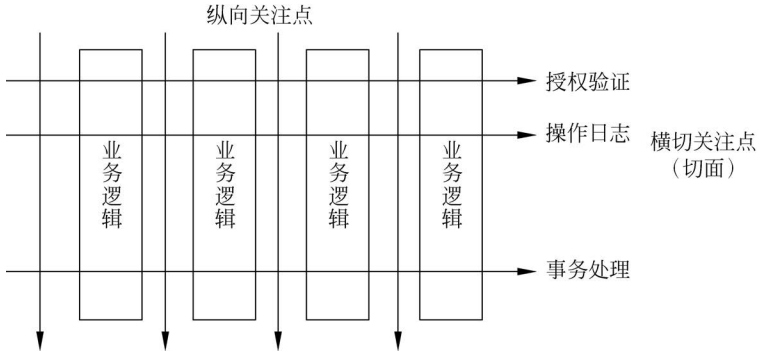


图 5-10 AOP 原理

在图 5-10 中,AOP 将软件系统分为两个关注点:纵向关注点和横切关注点。业务逻辑是纵向关注点;与业务逻辑关系不大、但各个业务逻辑模块都需要的共有功能部分是横切关注点。AOP 中所谓的切面指的就是横切关注点,切面将那些与业务无关、却被业务逻辑模块所

共同调用的逻辑或责任封装起来，便于减少系统的重复代码，降低模块间的耦合度，有利于提高系统的可操作性和可维护性。例如：授权验证、操作日志、事务处理等都是每个业务逻辑模块需要的，它们都属于横切关注点。以授权验证为例，每个业务逻辑基本都需要进行授权验证，授权验证也不属于具体的业务，它会横跨多个业务模块。不可能在每个业务逻辑模块里面都重复编写授权验证代码，因此授权验证将被提取出来，作为各业务逻辑模块的公用功能。

简言之，AOP 将各业务逻辑公用的功能提取出来，当公用的功能发生变化时，只需要修改公用功能的代码即可。AOP 以一个新的视角来看待软件的架构，即使不使用 AOP 技术，只使用 AOP 的某些观念和现有的技术来搭建系统架构，对于分离某些本来是紧耦合的关注点，也是非常有益的。

3. 面向服务架构

面向服务的架构(Service-Oriented Architecture，SOA)是一种应用程序架构，它是在企业内部 IT 系统重复构建，效率低下的背景下提出的。在 SOA 模型中，所有的功能都被定义成了独立的服务，所有的服务通过服务总线或流程管理器来连接，这种松散耦合的架构使得各服务在交互过程中无需考虑双方的内部实现细节以及部署的平台。各服务带有定义明确的可调用接口，能够以定义好的顺序调用这些服务来形成业务流程。因此，SOA 本质上是服务的集合，服务之间彼此通信，这种通信可能是简单的数据传送，也可能是两个或更多的服务协调进行某些活动。服务之间需要某些方法进行连接。图 5-11 描述了一个 SOA 参考模型。

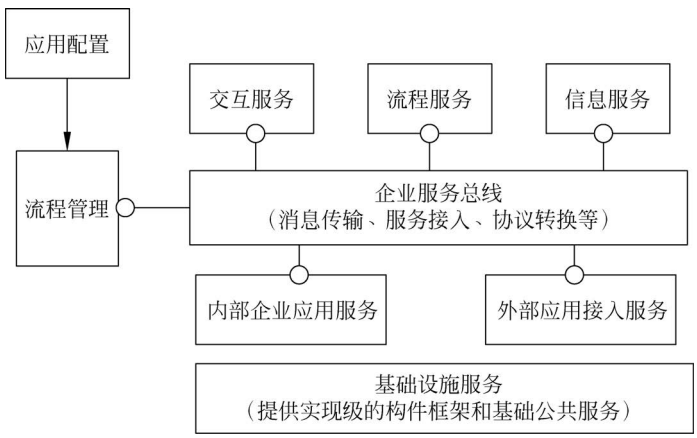


图 5-11　SOA 架构参考模型

在图 5-11 中，企业服务总线提供消息传输、服务接入、协议转换、数据格式转换、基于内容的路由等功能，是构建基于 SOA 解决方案所用基础架构的关键部分，由中间件技术实现并支持 SOA 的一组基础架构功能。各类服务从功能上分为交互服务、流程服务、信息服务、内部企业应用服务和外部应用接入服务等。基础设施服务提供实现级的构件框架和基础公共服务。流程管理是指服务注册、服务管理、业务流程编排等业务支持服务。应用配置包括：根据实际应用场景的需要或通信中间件类型配置对应的服务接口实例化参数，配置对应的软件应用组件参数等。

通过服务接口的标准化描述，使得服务可以提供给任何异构平台和任何用户接口使用。SOA 服务间的交互方式是：各个服务根据统一标准向企业服务总线进行注册；服务调用其他服务时，企业服务总线根据服务接口提供的统一标准寻找其他服务，服务请求者无需了解服务的运行所在地、编写语言以及消息的传输路径，只需要提出服务请求即可获得正确的响应。

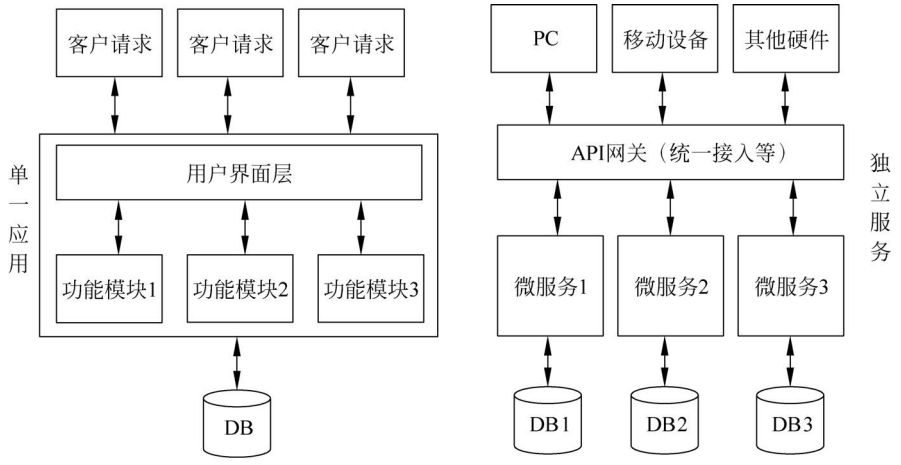
基于面向服务的架构思想将重复公用的功能抽取为组件,能够提高软件开发效率、复用性和可维护性。由于以组件服务的方式给各子系统提供服务,因此可针对不同组件服务特点制定集群及优化方案。企业服务总线作为子系统之间通信的桥梁,可以减少系统中的接口耦合。

4. 微服务架构

微服务架构是 SOA 的一种变体,它提倡将单一应用程序划分成一组小的服务,服务之间互相协调、互相配合,为用户提供最终价值。每个服务运行在其独立的进程中,服务与服务间采用轻量级的通信机制互相沟通(通常是基于 HTTP 的 RESTful API)。每个服务都围绕着具体业务进行构建,并且能够独立地部署到生产环境中,从而降低系统的耦合性,并提供更加灵活的服务支持。图 5-12 对比了传统 Web 应用开发模型与微服务架构模型的区别。

在图 5-12(a)中,传统的 Web 应用是单一应用,Web 工程将所有的功能模块打包到一起并放在一个 Web 容器中运行。当应用不太复杂时,这种开发模式具有架构简单、部署成本低的优点。但是随着大规模的复杂应用场景越来越多,这种单一应用会显得很笨重:当要修改一处代码时就要将整个应用全部部署,不利于更新技术框架,除非重写全部的应用。

在图 5-12(b)中,从粒度方面考虑,每一个微服务可以是单一的功能模块,也可以是一个代表单一功能的应用程序(例如:提供天气预报或者文件存储)。因此,设计合适的微服务粒度在实践中是一个很大的挑战。由于微服务单独部署且通常不依赖其他服务,因此在需求改变时,可以敏捷地重写微服务,可以对某个微服务单独尝试使用最新的技术、语言和框架等。微服务中经常使用 API 网关的主要目的不是重用代码,而是减少客户端和服务间的往来,完成统一身份认证和权限校验、服务路由和负载均衡、请求限流等。同时,微服务架构的应用使得软件开发组织可围绕业务领域组件来创建应用,这些应用可独立地进行开发、管理和迭代,每个微服务所访问的后台数据库允许根据数据性质不同而不同。例如:数据库可以是 MySQL、MongoDB、ES 等不同类型。



(a) 传统Web应用开发模型　　(b) 微服务架构模型

图 5-12　传统 Web 应用开发模型与微服务架构模型对比

微服务架构的特点是将系统服务层完全独立出来,并将业务服务抽象为一个个的微服务。它的本质是面向更细粒度的服务,即:使用小的业务功能服务级别去解决更大、更实际的问题。业务服务拆分粒度越细,越有利于资源重复利用,提高开发效率,项目版本迭代周期更短,可更加灵活地为每个服务制定不同的优化方案,提高系统可维护性,使产品交付变得更加简单。微服务架构采用去中心化思想,各服务可采用不同的技术栈实现,适用于复杂的互联网中

大型项目。目前很多企业开始利用微服务架构实施 IT 架构转型，与此同时，也出现了很多微服务框架。例如：Spring Cloud 就是微服务架构实施中常用的一个实现技术框架。

5. 服务网格架构

虽然微服务架构可带来一系列好处，但同时也面临一些问题：需要解决网络通信的问题；维护不同语言开发和非业务代码带来的成本；服务拆分的细化虽然实现了轻量级解耦，但是维护成本却更高了。解决上述问题的思路是：将非业务代码从业务代码中拆分出来，解决服务之间的通信问题，也就是说在客户端请求调用相关服务时，中间的网络通信应尽量与业务代码无关。

服务网格(Service Mesh)架构可用于解决上述分布式微服务开发与运维部署时面临的问题，它能够实现面向业务功能的服务和非功能服务(系统控制服务)的彻底解耦拆分。它强调了业务逻辑与服务治理逻辑的分层及解耦，在业务逻辑和服务治理基础设施间划分出清晰的边界，图 5-13 给出了一个服务网格架构参考模型。

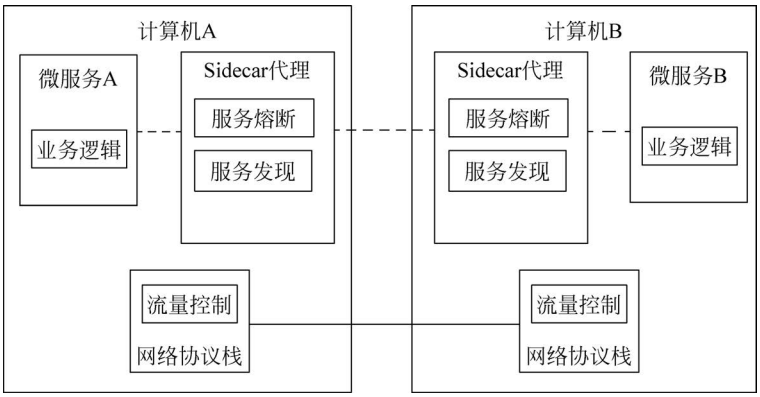


图 5-13　服务网格架构参考模型

服务网格架构下，通过网格代理进行服务间通信，不仅强调业务逻辑的解耦和复用，更强调基础设施的解耦和复用。服务网格架构的本质是微服务架构下的服务治理平台，包含服务治理的所有方面。例如，Spring Cloud 也能实现服务治理，但是它的服务治理实现与业务逻辑耦合在一起，部署、运维的同时耦合了微服务本身的操作，这样给开发人员在开发、测试、回归、发布等环节带来了大量的重复工作。通过将与业务逻辑无关的服务治理逻辑下沉到基础设施，服务网格让业务开发人员与基础技术开发人员的关注点分离，各司其职，大大提升了研发效率。业务开发人员可以更关注对业务的理解、建模，集中精力实现业务开发。目前比较流行的服务网格架构实现是基于云原生的 Istio。

5.3　接口设计

5.3.1　接口分类

接口的主要类型包括外部接口、内部接口和用户接口。

1. 外部接口

外部接口是本系统与外部系统或外部设备的接口。当业务运行需要多个系统的支持时，不同系统间的数据交互不可避免。例如：高校毕业管理系统在毕业生办理毕业手续时，需要调用各部门的数据：财务系统的学费缴纳数据、图书管理系统的借阅数据、后勤管理系统的退宿信息

等。上述示例展示了高校组织内部系统间的数据交互需求,除此之外,在不同组织间也会存在系统数据交互。例如:学生缴费系统需要通过与银行系统间的接口完成缴纳学费等相关业务。

在结构化设计方法中,外部接口设计的依据来自顶层数据流图所确定的系统边界,由穿过系统边界的数据流来定义。

在面向对象设计方法中,外部接口设计的依据来自系统需求分析阶段的系统用例图,由位于系统外部的参与者(包括外部系统、外部设备)与系统间的交互来定义。

当系统要从外部系统获取资源或信息时,外部系统通常不会将数据库直接共享,更多的是提供一个封装的方法作为对外提供数据的接口,系统通过调用接口获取所需的数据。

2. 内部接口

内部接口是系统内部构件(或模块)间的接口。

存在调用关系的两个构件/模块之间应定义内部接口。对于每一个构件/模块,应给出接口描述,即提供获取该模块数据的方法。每一个接口描述至少包括:接口名、接口参数类型及接口返回值。

3. 用户接口

用户接口指系统与用户间的交互界面。用户接口在有些书籍中被划分为外部接口的一种类型。由于用户界面设计与系统外部接口设计工作内容差别较大,本书将用户接口单独划分为一种接口类型。用户界面是人机交互的主要方式,用户界面的设计质量将直接影响软件产品的用户体验,从而直接影响软件产品的竞争力和使用寿命。

接口设计在系统体系结构中扮演着非常重要的角色。对于一个良好的接口设计,在接口需要变动时,应该尽可能少地改变目前的系统设计结构。接口设计不应涉及模块或子系统内部的实现细节,以减少接口变动造成的影响。

5.3.2 接口的定义与访问

首先对于接口从不同角度进行划分,了解接口的相关概念、含义及常用接口协议,然后介绍接口的定义及访问方法。

1. 接口划分

接口是两个独立系统/模块之间互相访问或者同步数据的途径。

1) 从服务供需的角度划分

从服务双方供需的角度划分,接口可分为服务提供方和服务使用方。服务提供方是服务的提供者,需要提供接口;服务使用方是服务的调用者,需要调用服务提供方给出的接口。

例如:智慧社区养老系统(本书案例系统)需要从户籍系统(第三方系统)获取本社区满足年龄限定的老年人信息,为本系统的养老服务提供基础数据。在此场景下,智慧社区养老系统是服务使用方,户籍系统是服务提供方。户籍系统需要提供接口,智慧社区养老系统通过访问该接口实现服务调用。

上述示例指的是两个系统之间通过接口调用相互访问,属于外部接口调用。对于一个前后端分离的 Web 应用程序开发,前后端子系统之间的交互也是通过接口调用完成的。后端子系统作为服务提供方提供服务接口,前端子系统作为服务使用方访问服务接口,这里属于内部接口调用。

2) 从数据获取方式划分

从数据获取方式划分,接口可分为主动推送和主动拉取。

主动推送是指数据提供方一旦更新,则触发推送,将所需字段对应值传递给数据使用方。

主动推送场景由数据使用方提供接口，数据提供方调用接口推送数据。

主动拉取是指数据使用方传递请求参数，数据提供方按照协议响应，将满足请求参数条件的数据返回到数据使用方。主动拉取场景由数据提供方提供接口，数据使用方调用接口拉取数据。

适用场合推荐：对实时性要求高的应用场景，建议由数据提供方主动推送；对于实时性要求不高或者数据传输量较大的应用场景，则建议数据使用方按一定频率主动拉取，有利于系统负荷压力稳定。

例如：

(1) 志愿者管理系统(第三方系统)需要从智慧社区养老系统获取志愿者的志愿服务记录信息，这种数据获取方式属于主动拉取。智慧社区养老系统作为数据提供方，需要提供外部接口。

(2) 户籍系统需要主动将人口的异动信息同步给智慧社区养老系统，为准确发放养老券提供依据，这种数据获取方式属于主动推送。智慧社区养老系统作为数据使用方，需要提供外部接口。

3) 从调用方式划分

从调用方式划分，接口可分为同步调用和异步调用。

同步调用是指服务使用方在调用接口时，要等待服务提供方的操作执行完毕，返回响应结果后才可以继续执行下面的操作。

异步调用是指服务使用方在调用接口时，不用等待服务提供方返回响应，可以继续执行下面的操作，后续通过服务使用方轮询或者服务提供方回调的方式获取接口调用结果。

例如：

(1) 当老年人/老年人亲属通过公众号前端查询订单进度时，需要等待后端提供的查询接口处理完毕后，返回查询结果。此时的查询订单操作属于前端同步调用后端接口。

(2) 第三方支付平台在处理各种渠道提交的支付请求时，一般都是采用异步处理的方式，在其接口文档上都会说明支付结果以异步通知为准。当一个支付请求被发送到支付渠道方，支付渠道会很快返回一个支付受理成功的通知，但是这个通知只能说明支付受理接口调用成功，并不是扣款成功。这里的“调用支付接口”属于同步调用；支付渠道在支付受理成功后，在同步请求参数里会有一个回调地址，这个地址对应着通过网联转给发卡行进行扣款的请求，这里的“调用发卡行扣款接口”属于异步调用。在这个支付过程中，同步接口仅检查参数是否正确，签名是否无误等；异步接口才反馈扣款结果，支付渠道异步通知商户支付成功，商户后台也可以主动查询支付结果。

2. 常见的接口协议

常见的接口协议如表 5-2 所示。

表 5-2　常见的接口协议

接口	HTTP 接口	RPC 接口	MQ 接口	Web Service 接口
描述	HTTP 接口是基于 HTTP 的开发接口，通过 HTTP 接口，开发人员可以发送特定的请求到服务器，并获取所需的数据或执行相应的操作	RPC 技术是指远程过程调用，可以像调用本地方法一样去调用远程服务器上的方法	MQ 接口是指消息队列，是一种应用程序对应用程序的通信方法。应用程序通过写和检索出入列队的数据(消息)来通信，而无需专用连接来链接它们	Web Service 是指以 Web 形式提供的服务。Web 服务提供者开放一系列的 API，开发人员通过调用这些 API 来集成 Web 服务，构建自己的应用程序

续表

接口	HTTP 接口	RPC 接口	MQ 接口	Web Service 接口
基于或支持的协议	HTTP	HTTP、TCP、UDP、自定协议	常见协议有在 HTTP 级别工作的 AMQP(高级消息队列协议)和 STOMP(一种面向文本的简单消息协议)	基于 HTTP 协议的 SOAP 协议的封装和补充
传输的数据格式	JSON	JSON、XML 等,一般是 XML	JSON	XML
应用场景	使用广泛、跨平台、跨语言; 通常第三方提供的 API 都会有 HTTP 版本的接口; Spring Cloud 框架的各个服务间是通过 HTTP 方式来实现; 目前流行的 REST 风格,是通过 HTTP 来实现的	对于要立即等待返回处理结果的场景,首选 RPC; 当项目较大,需要解耦服务、灵活部署时就要用到 RPC,主要解决分布式系统中,服务与服务之间的调用问题。目前热门的 DUBBO 框架是 RPC 的典型代表	消息传递指的是程序之间通过在消息中发送数据进行通信。因此,MQ 接口主要用于不是通过直接调用彼此来通信的应用程序(直接调用通常是用于诸如远程过程调用的技术)	Web Service 解决了过去的中间件平台要求服务的客户端与系统提供的服务本身之间紧密耦合的问题,采用松散耦合的基本结构,更加适合于互联网运用

在早期 Java Web 系统开发过程中,当作为接口提供商给第三方提供接口以及作为客户端去调用第三方提供的接口时,大部分时候都是使用 Web Service 接口。随着技术不断进步,目前很多第三方接口都改成了基于 HTTP 接口直接传递 JSON 数据的方式,来代替 Web Service。

关于传输数据格式的选择,目前 JSON 比 XML 更加流行,这是由于 JSON 的可读性比 XML 强,解析规则也更简单。但是 JSON 支持的数据类型较少,且不精确,所以在一些业务要求较高的领域,使用 XML 更合适。

3. 接口定义

接口定义是指应撰写相应的接口文档对接口相关信息进行说明,以便来自接口双方的工程师在项目开发过程中,进行沟通交流、使用及维护。以目前常见的 Web 开发应用为例,对 Web 开发前后端接口的定义主要包含以下几部分:接口名称、接口 URL、请求方法、请求参数和响应参数。

1) 接口名称

接口名称用于区分不同的接口。接口命名后应对接口的功能作一个简单说明。例如:“OCR 接口:本接口提供基于小程序或 H5 的身份证、银行卡、行驶证的 OCR 识别”。

2) 接口 URL

接口的调用地址。URL 格式为:

protocol://hostname[:port]/path/[;parameters][?query]#fragment

其中,

- protocol:指定使用的传输协议,常见的 protocol 属性包括:
  - ◆ “http://”:HTTP 是目前万维网中应用最广的协议;
  - ◆ “file:///”:支持访问本地资源;
  - ◆ “ftp://”:支持通过 FTP 访问资源;

◆ “https://”：支持通过安全的 HTTPS 访问资源；

◆ “mms://”：支持 MMS(流媒体)协议访问资源。

- hostname：存放资源服务器的主机名或 IP 地址。
- port：端口号，可选。各种传输协议都有默认的端口号，如 HTTP 的默认端口号为 80。在端口号省略时使用传输协议的默认端口号。有时出于安全的考虑，可对端口号重新定义，采用非标准端口号，此时 URL 中的端口号不能省略。
- path：表示路径，由零个或多个反斜杠符号(/)隔开的字符串表示，一般用来表示主机上的一个目录或文件地址。
- parameters：参数，可选。用于指定特殊的参数。
- query：查询，可选。用于将用户提供的信息从浏览器传递到服务器。当传递多个参数时，每对参数间用 & 隔开，每对参数采用 key=value 的表示方法。
- fragment：信息片段，字符串。用来标识 URL 所标识资源中的某个资源片段。

例如：

https://api.weixin.qq.com/cv/ocr/idcard?img_url=ENCODE_URL&access_token=ACCESS_TOCKEN

在上述 URL 地址中，“https”是采用的协议；“api.weixin.qq.com”是 OCR 接口资源所在的主机地址；“cv/ocr/idcard?img_url=ENCODE_URL&access_token=ACCESS_TOCKEN”是 OCR 接口资源的具体地址；“?”后面是接口调用时需传入的请求参数：img_url 是要检测的图片地址，access_token 是接口调用凭证。

一般情况下，URL 地址中的“协议+地址[:端口号]”这部分内容在接口文档里面不要固定，因为未来项目可能会部署在多个域名下面，URL 地址会变化。因此，在接口文档中只需要提供方法映射所需的 URL 部分即可。以 OCR 接口为例，在接口文档中只需要提供 URL 地址中的“/cv/ocr/idcard?img_url=ENCODE_URL&access_token=ACCESS_TOCKEN”部分即可。

3) 请求方法

常用的 HTTP 请求方法包括 GET、POST、PUT 和 DELETE。可以认为：一个 URL 地址用于描述一个网络上的资源，HTTP 中的 GET、POST、PUT 和 DELETE 对应着对该资源的查询、增加、修改和删除 4 种类型的操作。GET 一般用于获取/查询资源信息，POST 一般用于创建资源，PUT 一般用于更新资源信息，DELETE 一般用于删除资源信息。

4) 请求参数

描述接口需要传递的参数。每个参数必须包含参数名称、参数类型、是否必选和参数说明 4 项内容。

5) 响应参数

描述接口的返回值。一般包含返回值名称、返回值数据类型和返回值说明 3 项内容。目前主流的数据返回格式都是以键值对 { key：value }(JSON 格式)的形式成对出现的。其中，key 是用来读取指定数据的唯一标识，value 是这个 key 所对应的数据信息。

4. 接口访问示例

下面的 URL 是由中国气象局提供的，一个面向外部免费的访问接口。

http://t.weather.sojson.com/api/weather/city/101010100

URL 中的参数“101010100”是北京的城市代码，访问该接口可查询北京市的天气状况。查询其他城市时只需要把 URL 中代表城市代码的参数修改成其他城市代码即可。

在浏览器中输入上面的访问接口地址,返回结果如图 5-14 所示。图 5-14 是接口返回的 JSON 格式的数据,前端开发工程师只需从中取出项目所需的数据显示到人机交互界面,即可实现通过访问外部接口获取数据。

{"message":"success感谢又拍云(upyun.com)提供CDN赞助","status":200,"date":"20230717","time":"2023-07-17 17:30:15","cityInfo":
{"city":"北京市","citykey":"101010100","parent":"北京","updateTime":"16:31"},"data":
{"shidu":"32%","pm25":18.0,"pm10":40.0,"quality":"轻度","wendu":"35","ganmao":"儿童、老年人及心脏、呼吸系统疾病患者人群应减少
长时间或高强度户外锻炼","forecast":[{"date":"17","high":"高温 35℃","low":"低温 23℃","ymd":"2023-07-17","week":"星期
一","sunrise":"05:00","sunset":"19:41","aqi":106,"fx":"南风","fl":"2级","type":"晴","notice":"愿你拥有比阳光明媚的心情"},
{"date":"18","high":"高温 36℃","low":"低温 24℃","ymd":"2023-07-18","week":"星期
二","sunrise":"05:01","sunset":"19:40","aqi":94,"fx":"南风","fl":"2级","type":"晴","notice":"愿你拥有比阳光明媚的心情"},
{"date":"19","high":"高温 36℃","low":"低温 24℃","ymd":"2023-07-19","week":"星期
三","sunrise":"05:01","sunset":"19:40","aqi":102,"fx":"南风","fl":"2级","type":"晴","notice":"愿你拥有比阳光明媚的心情"},
{"date":"20","high":"高温 35℃","low":"低温 23℃","ymd":"2023-07-20","week":"星期
四","sunrise":"05:02","sunset":"19:39","aqi":97,"fx":"南风","fl":"2级","type":"多云","notice":"阴晴之间,谨防紫外线侵扰"},
{"date":"21","high":"高温 26℃","low":"低温 23℃","ymd":"2023-07-21","week":"星期
五","sunrise":"05:03","sunset":"19:38","aqi":94,"fx":"东南风","fl":"2级","type":"小雨","notice":"雨虽小,注意保暖别感冒"},
{"date":"22","high":"高温 30℃","low":"低温 23℃","ymd":"2023-07-22","week":"星期

图 5-14　天气预报访问接口返回结果

5.3.3　用户界面设计

用户界面是用户与软件系统进行信息交互的媒介。用户界面设计主要内容包括交互流设计、数据输入界面设计、数据输出界面设计和控制界面设计。为了做好用户界面设计,界面设计工程师需要了解用户界面应具备的特性,掌握用户界面设计的基本步骤。

1. 用户界面应有的特性

1) 可用性

可用性是用户界面设计的基本目标,包括使用简单、界面一致、拥有帮助功能、必要的系统响应和具有容错能力等。使用简单是指用较少的操作步骤就能执行目标功能;界面一致是指软件系统各个用户界面应采用一致的界面风格、术语及活动步骤;拥有帮助功能是指在不阅读用户操作说明文档的情况下,软件应为复杂功能提供联机帮助;必要的系统响应是指用户在向系统发出请求后,系统应及时给出反馈信息,尤其在等待系统执行操作的过程中,应给出当前的执行进度;具有容错能力是指当用户操作错误时,应尽可能地提供恢复功能。

2) 灵活性

对于不同的用户或不同的接入方式,系统应有不同的界面形式,但整体应保持风格一致性。用户通常可以分为外行型、初学型、熟练型和专家型,界面设计工程师可以根据用户所属的类型设计适合于大部分用户使用的界面。

3) 可靠性

用户界面的可靠性是指无故障使用的间隔时间。用户界面应该保证用户可靠地、正确地使用系统,保证有关程序和数据的安全性。

2. 页面流程图

页面流程图是一个进行用户界面系统性规划的工具,它能够展示应用系统的功能和实现功能所需的页面间的跳转关系,借助它可以直观地体现用户与系统的交互过程。

页面流程图的目标是规划用户的行为路径,而不是单页面交互设计,所以在绘制页面流程图时无需考虑页面内容、布局,每个页面都不必追求精细,而应聚焦于用户业务流程的界面设计。

如何着手绘制页面流程图呢? 通过前期需求分析中的用例图和业务流程图,已经明确了系统应具备的功能和提供的数据,接下来就需要将这些功能及数据分配到不同的页面。面向对象设计方法选择基于需求分析获取的用例图,针对每一个用例,绘制用例描述中涉及的所有页面,然后标识出页面间的流转。注意:不同用例可能会使用相同的页面(尤其是主页面)。

绘制页面流程图和页面初步设计的工具可以选用专业的快速原型设计工具，目前常见的原型设计工具有 Axure RP、InVision、Mockplus、Flinto、墨刀等。

3. 用户界面设计步骤

(1) 在对每一个具体页面进行详细设计前，应先进行界面设计的总体规划。从需求分析阶段的成果(例如：用例图及用例描述、业务流程图)中获得信息，绘制页面流程图，描述引起用户界面跳转的事件和动作。

(2) 以页面流程图为依据，规划页面流程图中每个界面的布局，明确实现界面功能的界面对象，绘制用户界面粗略布局图。

(3) 在用户界面粗略布局图的基础上，从美学角度进行优化，以获得更加良好的用户体验。

5.4　数据库设计

常用的数据存储方法包括两种：文件存储和数据库存储。大多数情况下，软件系统更多使用数据库存储数据，因为数据库不仅提供了多种存储策略，还可以满足数据一致性的要求，基于数据库还能很方便地完成数据计算。与数据库存储数据相对的是使用文件存储数据，文件可以借助文件系统的树状目录进行管理，不同项目、模块按照树形结构存储，管理和使用都很方便。但文件本身没有计算能力，也无法保证数据一致性，更适用于一些非结构化数据(例如：视频、图片等)的存放。本节将重点介绍数据库设计。

数据库设计是指针对待开发的软件系统，根据用户的各种信息管理要求和数据操作要求，设计出优化的数据库逻辑模式及其应用程序，并据此建立数据库及其应用系统，使之能够有效地存储和管理数据。

数据库的设计过程包括 6 个阶段：需求分析、概念结构设计、逻辑结构设计、物理结构设计、数据库实施、数据库运行与维护。

1) 需求分析

通过详细调查组织的业务流程，由数据库设计人员和用户双方共同收集信息需求和处理需求，明确系统运行过程中需要管理的数据。需求分析是数据库设计的起点。

2) 概念结构设计

概念结构设计是整个数据库设计的关键，它通过对用户的需求进行综合、归纳与抽象，形成一个独立于具体数据库管理系统的概念模型。对于关系型数据库管理系统，概念结构设计的任务是建立 E-R 模型。

3) 逻辑结构设计

逻辑结构设计是指将概念模型转换成某个数据库管理系统所支持的数据模型，并对其进行优化。对于关系型数据库管理系统，逻辑结构设计的任务是将 E-R 模型转换为关系模式。

4) 物理结构设计

物理结构设计是指为逻辑数据模型选取一个最适合应用环境的物理结构，即设计数据库的存储结构和存取方法。对于关系型数据库管理系统，物理结构设计的任务是：定义数据库、定义表及字段、定义索引等数据库文件的物理存储结构；选择合适的存储引擎。

5) 数据库实施

在数据库实施阶段，设计人员运用数据库管理系统提供的数据语言及其宿主语言，根据物

理结构设计的结果创建数据库,编制与调试数据库应用程序,组织数据入库,并进行数据库试运行。

6) 数据库运行与维护

数据库运行与维护是指数据库应用系统正式投入运行后,必须不断地对数据库系统进行评价、调整与修改。

对于上述 6 个数据库设计阶段,需求分析、概念结构设计阶段独立于数据库管理系统,即与具体选用何种数据库管理系统无关;逻辑结构设计、物理结构设计、数据库实施、数据库运行与维护设计阶段依赖于具体所采用的数据库管理系统。

数据库的需求分析工作实际上包含在前期的系统需求分析工作中,在此不再赘述。物理结构设计、数据库实施、数据库运行与维护主要是由数据库管理员和数据库应用程序开发人员来负责。对于软件设计人员,最关心的是概念结构设计和逻辑结构设计阶段,这也是本节介绍的重点内容。

5.4.1 概念结构设计

概念结构设计是将需求分析得到的用户需求抽象为信息结构(即概念模型)的过程。概念结构设计阶段的产出成果是待开发软件系统的概念模型。

概念模型是一种面向问题的数据模型,是按照用户的观点对数据建立的模型,它能真实、充分地反映现实世界,包括事物和事物之间的联系,是从现实世界映射到信息世界的一个数据模型。

概念模型的重要性在于它易于理解,能够成为开发人员和用户之间沟通业务、交换意见的桥梁。概念模型独立于数据库管理系统使得它能够向各种数据模型转换,它是各种数据模型的共同基础。描述概念模型的工具包括传统的 E-R 模型和 UML 概念模型。

E-R 模型是用实体-联系(Entity-Relationship,E-R)图来描述现实世界的概念模型。E-R 图通过图形刻画现实世界实体与实体之间的联系,为客观事物建立概念模型。一个 E-R 图示例如图 5-15 所示。

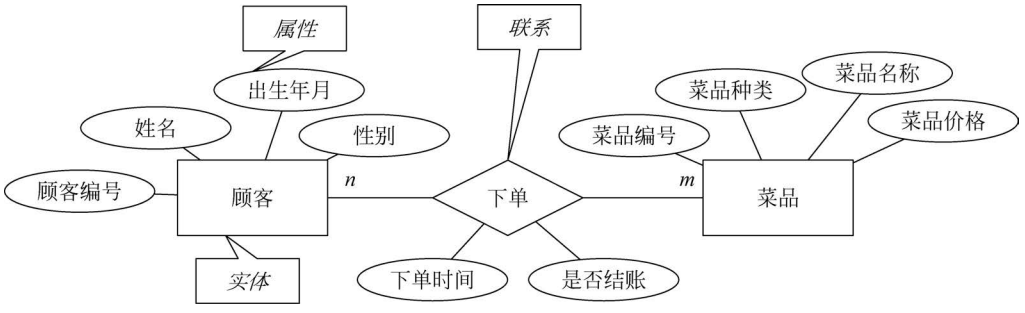


图 5-15 一个 E-R 图示例

在图 5-15 中,E-R 图包含三类基本元素:实体、联系和属性。实体与实体属性间、实体与联系间、联系与联系特有的属性间都是用实线相连。

1. 实体(Entity)

实体用矩形框表示,矩形框内标识实体的名称。

实体是首要的数据对象,是现实世界中客观存在的、可以被区分的事物,可以用来表示人、物或事件。实体既可以是实际能够触摸的对象,也可以是存在于头脑中的抽象概念。实体的命名一般为名词,例如:教师、学生、课程等。

实体集是指同一类实体构成的集合。一般将实体、实体集等概念统称为实体。E-R 模型中提到的实体往往是指实体集。

在图 5-15 中，顾客、菜品是实体，每一位顾客、每一个菜品都是一个具体的实体对象。

2. 联系（Relationship）

联系用菱形框表示，菱形框内标识联系的名称。

联系体现了现实世界中事物之间的联系，在信息世界中表示实体之间或实体内部的关联关系，联系在 E-R 模型中扮演了非常重要的角色。联系的命名一般为动词，例如：老师与学生之间存在“授课”联系。

按照联系所关联的实体数量，将联系分为以下 3 种类型。

1）两个实体之间的联系

两个实体之间的联系（又称二元联系）是 E-R 图中最常见的联系形式。按照联系两端所关联的实体对象数量的约束，两个实体之间的联系又分为以下三种类型。

（1）一对一联系（1∶1）。

如果对于实体集 A 中的每一个实体，实体集 B 中至多有一个（也可以没有）实体与之联系，反之亦然，则称实体集 A 与实体集 B 具有一对一联系，记为 1∶1。

例如：若一个老年人只能拥有一台智能终端，同时，一个智能终端也只能属于一个老年人，则称老年人与智能终端之间是一对一的联系，对应的 E-R 模型如图 5-16 所示。

图 5-16　一对一联系（1∶1）示例

（2）一对多联系（1∶n）。

如果对于实体集 A 中的每一个实体，实体集 B 中有 n 个实体（$n \geqslant 0$）与之联系，反之，对于实体集 B 中的每一个实体，实体集 A 中至多只有一个实体与之联系，则称实体集 A 与实体集 B 具有一对多联系，记为 1∶n。

例如：若一个社区可以管理多个老年人，而一个老年人只能属于一个社区，则称社区和老年人之间是一对多的联系，对应的 E-R 模型如图 5-17 所示。

图 5-17　一对多联系（1∶n）示例

（3）多对多联系（m∶n）。

如果对于实体集 A 中的每一个实体，实体集 B 中有 n 个实体（$n \geqslant 0$）与之联系，反之，对于实体集 B 中的每一个实体，实体集 A 中也有 m 个实体（$m \geqslant 0$）与之联系，则称实体集 A 与实体集 B 具有多对多联系，记为 m∶n。

例如：若一个社区可以有多家养老服务提供商，而一家养老服务提供商可以服务多个社区，则称社区和养老服务提供商之间是多对多的联系，对应的 E-R 模型如图 5-18 所示。

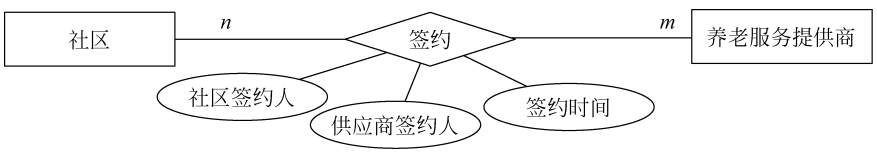


图 5-18　多对多联系（m∶n）示例

深入思考 5.2 上述社区、老年人、养老服务提供商等实体间的联系示例是否无论放在哪个社区养老类的系统中都成立？例如：社区和养老服务提供商是不是一定是多对多的联系？

参考答案：请参见微课视频 5-2。

深入思考 5.3 在实际 E-R 图绘制中，对于两个实体之间的联系，如何分析才能快速且准确地标注联系某一端的实体约束数值为 1 还是 n？

参考答案：请参见微课视频 5-3。

2) 多个实体之间的联系

当现实世界中事物之间的联系牵涉的实体较多，用两个实体之间的联系无法表达清楚时，则可以用多个实体之间的联系来表达，最为常见的是三个实体之间的联系(又称三元联系)。不建议一个联系关联太多实体，这样会使得 E-R 图过于复杂，不易理解。

按照联系所关联的三端实体对象数量的约束，三元联系又分为以下 4 种类型。

(1) 一对一对一联系(1∶1∶1)。

例如：对于产品线、分公司、产品经理三个实体集，若一个分公司的一条产品线只由一位产品经理负责；一条产品线的一位产品经理只能向一个分公司负责；一个分公司的一位产品经理只负责一条产品线，则称产品线、分公司、产品经理三个实体之间是 1∶1∶1 的三元联系，对应的 E-R 模型如图 5-19 所示。

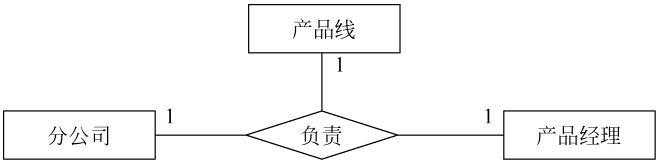


图 5-19 三元联系(1∶1∶1)示例

(2) 一对一对多联系(1∶1∶n)。

例如：对于教师、教材、课程三个实体集，若一个教师所讲授的一门课程只能使用一种教材；一个教师所使用的一种教材只能用于讲授一门课程；一门课程所使用的一种教材可以被多个教师讲授，则称教师、教材、课程三个实体之间是 1∶1∶n 的三元联系，对应的 E-R 模型如图 5-20 所示。

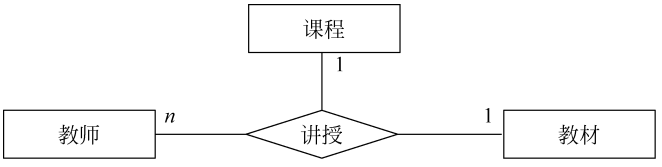


图 5-20 三元联系(1∶1∶n)示例

(3) 一对多对多联系(1∶m∶n)。

例如：对于志愿者、老年人、养老服务三个实体集，若一个志愿者为一位老年人能提供多项养老服务；一个志愿者的一项养老服务能提供给多位老年人；一个老年人得到的一项养老服务只能由一位志愿者提供，则称志愿者、老年人、养老服务三个实体之间是 1∶m∶n 的三元联系，对应的 E-R 模型如图 5-21 所示。

(4) 多对多对多联系(m∶n∶p)。

对于供应商、零件、项目三个实体集，若在一个项目中，一个供应商能供应多种零件；一个项目中的一个零件可以有多家供应商；一个供应商供应的一种零件可以应用在多个项目中，则称项目、供应商、零件三个实体之间是 m∶n∶p 的三元联系，对应的 E-R 模型如图 5-22 所示。

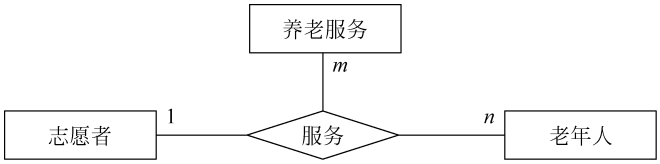


图 5-21　三元联系(1∶*m*∶*n*)示例

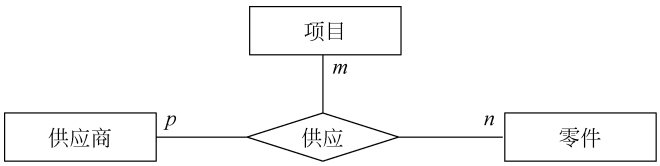


图 5-22　三元联系(*m*∶*n*∶*p*)示例

深入思考 5.4　在实际 E-R 图绘制中,对于三个实体之间的联系,如何解析才能快速且准确地标注联系某一端的实体约束数值为 1 还是 *n*?

参考答案:请参见微课视频 5-4。

3) 实体内部之间的联系

同一个实体集内部实体之间也可以存在联系,又称一元联系或递归联系。

例如:一个员工由一个部门经理管理,一个部门经理可以管理多个员工,但部门经理实质上也是员工实体集中的一个对象,此时称员工和员工实体之间存在 1∶*n* 的一元联系,对应的 E-R 模型如图 5-23 所示。

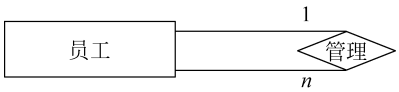


图 5-23　一元联系示例

同样地,一元联系的类型也分为三种类型:1∶1、1∶*n*、*m*∶*n*,具体需要根据语义确定。

3. 属性(Attribute)

属性用椭圆表示,椭圆框内标识属性的名称。

实体的属性是指实体的某一个特性。例如:图 5-15 中,顾客编号、姓名、性别、出生年月都是“顾客”实体的属性;菜品编号、菜品种类、菜品名称、菜品价格都是“菜品”实体的属性。属性为实体提供了详细的描述信息。

联系也可以有属性。例如:图 5-15 中,“下单”联系特有的属性为:下单时间、是否结账。联系的属性一般出现在“多对多”的二元联系或三元联系中。

在确定属性时,要注意两条原则:

1) 属性必须是不可再分的数据项

关系数据库中,属性不可再分体现了属性的原子性,意味着属性不能再由另一些属性组成。例如:如果需要按省、市、区查找“顾客”的住址,那么在设计时就不能只以一个字符串类型的“住址”作为“顾客”实体的属性,而应将“住址”拆分为省、市、区和街道等几个独立的属性。这样处理虽然满足了属性不可再分的原则,但是却和人们的思维习惯不吻合。解决思路是:用面向对象的思想来处理复杂属性,把复杂属性“住址”单独抽象为一个实体,在“顾客”和“住址”之间建立一个联系,从而既符合人们的思维习惯,又可以解决查询不便的问题,如图 5-24 所示。不仅如此,修改后的设计允许每个顾客提供多个地址,还能够增加应用的灵活性。

2) 属性不能与实体有联系

在 E-R 模型中,联系只能发生在实体之间。例如:学生居住在某个宿舍,如果一个简单的宿舍号能清楚标识学生的住所,那么宿舍号就可以作为学生的属性。但是如果学校有不同的

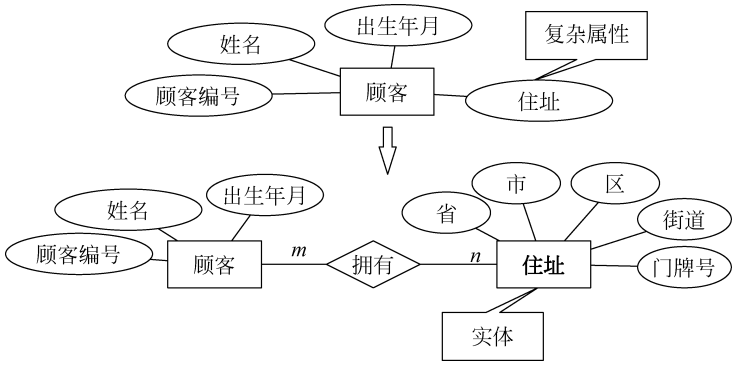


图 5-24 “住址”复杂属性的处理

校区，若要表达属性“宿舍号”与实体“校区”之间的关联，则无法实现。解决思路是：将“宿舍”抽取为一个实体，使其与“校区”实体间产生联系，如图 5-25 所示。

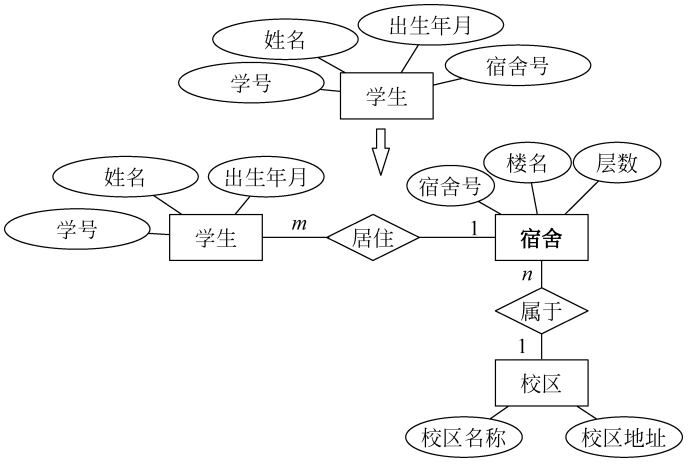


图 5-25 属性与实体有关联的处理方法

4. E-R 图的设计原则

E-R 概念模型的设计原则是：先局部、后综合。

1）设计局部 E-R 图

从需求分析阶段得到的多层数据流图中，选择一个适当层次的数据流图，作为设计局部 E-R 图的出发点，通常选取子系统的中层数据流图作为设计局部 E-R 图的依据。设计 E-R 图主要包括两个步骤。

（1）确定实体及其属性。

将各子系统涉及的数据从数据字典中抽取出来，参照数据流图，确定各个子系统中的实体、实体的属性和实体的码（唯一标识一个实体对象的属性或属性集合）。

（2）确定实体之间的联系及其类型。

首先确定实体之间是一元联系、二元联系还是多元联系，然后根据用户需求传递出的语义进行解析，进一步确定联系的类型，标识联系所关联的各端的实体数量约束。

2）集成得到全局 E-R 图

由于各个局部 E-R 图之间必定会存在许多不一致的地方，因此在合并局部 E-R 图时，需要消除属性冲突、命名冲突、结构冲突等，得到初步的全局 E-R 图。在此基础上，消除不必要的冗余，得到最终的全局 E-R 图。对局部 E-R 图集成后得到的全局 E-R 图就是系统的数据库

概念模型。

5. E-R 图的绘制原则

图 5-15 所示的 E-R 图模型完整包含了实体、属性和联系三种元素。当对一个软件系统进行建模时，由于牵涉的实体较多，每个实体的属性也较多，如果按图 5-15 的表示方法进行绘制，整个 E-R 图会显得内容繁杂，反而失去了用 E-R 图表达现实世界事物之间联系的作用。因此，在绘制 E-R 图时，推荐通过两张图来表达 E-R 图的全局及细节。

1）实体及联系图（全局）

从把握系统全局的数据模型结构出发，只画出实体及实体之间的联系，在这张图中暂不体现各实体的属性，但是联系特有的属性要标识出来。以图 5-15 为例，得到的简化 E-R 图如图 5-26 所示。

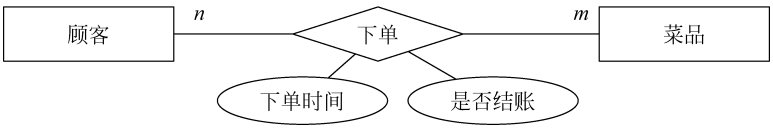


图 5-26　简化 E-R 图

2）实体及属性图（细节）

单独绘制另一张图，把描述各个实体详细信息的属性标识在各实体中，如图 5-27 所示。

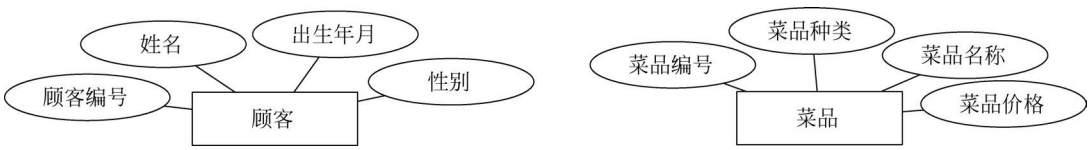


图 5-27　实体及属性图

UML 主要用于面向对象建模，但是也可用于数据库建模。

UML 概念模型与 E-R 模型相似，但是不提供多元联系。表 5-3 给出了 UML 概念模型和 E-R 模型中的术语对比。UML 概念模型中的类和 E-R 模型中的实体集都是现实世界的事物在信息世界的不同表示，它们最大的不同在于：类封装了属性和方法，既有静态数据，又包含动态行为；而实体只有属性，没有方法。当 UML 用于数据建模时，删除类中的方法并增加主键即可。整体比较两者来看，UML 概念模型跟 E-R 模型的整体概念基本相似，只是换了一种形式而已。

表 5-3　UML 概念模型和 E-R 模型中的术语对比

UML 概念模型	E-R 模型
类	实体集
属性	属性
方法	无
关联	联系

关于 UML 概念模型中类的相关概念及绘制方法已在第 3 章详细讲述，在此不再赘述。

5.4.2　逻辑结构设计

逻辑结构设计的任务是将概念结构设计阶段得到的概念模型转换成具体的数据库管理系统所支持的逻辑结构模型。下面重点讲解 E-R 模型如何转换为逻辑结构模型，UML 模型的

转换原理基本相同,不再赘述。

对于关系型数据库,逻辑结构设计的任务是将 E-R 模型转换为关系模式。E-R 图的核心元素是“实体”和“联系”,它们在转换为关系模式时,按照各自不同的原则进行转换。下面在描述关系模式时,关系模式的主码用下划线标识,体现实体之间联系的部分及联系特有属性用粗体标识。

1. 实体的转换原则

实体的转换原则:E-R 图中的每一个实体都将转换为一个独立的关系模式,实体的属性转换为关系的属性,实体的码转换为关系的码(唯一标识一条记录的属性或属性集合)。

以图 5-15 的 E-R 图示例为例,图中有 2 个实体“顾客”和“菜品”,转换后得到如下两个关系模式:

- 顾客(顾客编号、姓名、性别、出生年月);
- 菜品(菜品编号、菜品种类、菜品名称、菜品价格)。

2. 联系的转换原则

在概念模型设计中,E-R 图中的联系分为多种类型。在向逻辑结构模型转换时,不同的联系类型需要根据不同的转换规则进行。

1) 1∶1 联系的转换

1∶1 联系在转换时,有两种选择:转换为一个独立的关系模式;与任意一端对应的关系模式合并。

下面以图 5-16 一对一联系(1∶1)示例为例,分别使用上述两种转换方式进行转换。

(1) 联系转换为一个独立的关系模式。

转换规则为:将与该联系相连的各实体的码以及联系本身的属性转换为关系的属性;每个实体的码均是该关系的候选码。

在图 5-16 中,“拥有”联系关联了“老年人”和“智能终端”实体,“老年人”的码为身份证号,“智能终端”的码为智能终端编号。本例中的联系无属性。联系本身是动词,转换为关系模式后,应使用名词来命名关系模式,按照业务语义将其命名为“智能终端发放清单”。联系转换后的关系模式为

智能终端发放清单(老年人身份证号,智能终端编号)或智能终端发放清单(老年人身份证号,智能终端编号)

在这种规则下,图 5-16 所示的 E-R 模型转换得到的关系模型为三个关系模式:

- 老年人(老年人身份证号,姓名,性别,出生年月.....);
- 智能终端(智能终端编号,型号,单价.....);
- 智能终端发放清单(**老年人身份证号,智能终端编号**)或智能终端发放清单(**老年人身份证号,智能终端编号**)。

(2) 联系与任意一端实体对应的关系模式合并(推荐)。

转换规则为:将一端实体对应的关系模式的码和联系本身的属性合并到另一端实体对应的关系模式中,合并后关系模式的码不变。

在这种规则下,图 5-16 所示的 E-R 模型转换得到的关系模型为两个关系模式:

//将智能终端关系模式的码和联系的属性(本例无)合并到老年人关系模式中

- 老年人(老年人身份证号,姓名,性别,出生年月,**智能终端编号**);
- 智能终端(智能终端编号,型号,单价.....);

或

//将老年人关系模式的码和联系的属性(本例无)合并到智能终端关系模式中

- 老年人(老年人身份证号,姓名,性别,出生年月.....);
- 智能终端(智能终端编号,型号,单价.....,**老年人身份证号**)。

2) 1∶n 联系的转换

1∶n 联系在转换时,有两种选择:转换为一个独立的关系模式;与 n 端对应的关系模式合并。

下面以图 5-17 的一对多联系(1∶n)示例为例,分别使用上述两种转换方式进行转换。

(1) 联系转换为一个独立的关系模式。

转换规则为:与该联系相连的各实体的码以及联系本身的属性将转换为关系的属性;n 端实体的码是关系的码。

在图 5-17 中,"管理"联系关联了 1 端实体"社区"和 n 端实体"老年人","社区"的码为社区编号,"老年人"的码为老年人身份证号。本例中的联系无属性。联系转换为关系模式后,按照业务语义将其命名为"社区老年人名单"。联系转换后的关系模式为

社区老年人名单(老年人身份证号,社区编号)

在这种规则下,图 5-17 所示的 E-R 模型转换得到的关系模型为三个关系模式:

- 社区(社区编号,社区名称,所属行政区,街道.....);
- 老年人(老年人身份证号,姓名,性别,出生年月.....);
- 社区老年人名单(**老年人身份证号,社区编号**)。

(2) 联系与 n 端对应的关系模式合并(推荐)。

转换规则为:在 n 端关系中加入 1 端关系的码和联系本身的属性,成为合并后关系的属性;合并后关系的码仍是 n 端的码。

在这种规则下,图 5-17 一对多联系(1∶n)示例的 E-R 模型转换得到的关系模型为两个关系模式:

社区(社区编号,社区名称,所属行政区,街道.....);

老年人(老年人身份证号,姓名,性别,出生年月.....**社区编号**)。

3) m∶n 联系的转换

这种情况是最简单的,一个 m∶n 联系转换为一个独立的关系模式。转换规则为:与该联系相连的各实体的码及联系本身的属性作为关系的属性;各实体码的组合作为关系的码。

下面以图 5-18 的多对多联系(m∶n)示例为例进行转换。

在图 5-18 中,"管理"联系关联了 m 端实体"养老服务提供商"和 n 端实体"社区","养老服务提供商"的码为提供商编号,"社区"的码为社区编号。本例中的联系有一个属性:"签约时间"。联系转换为关系模式后,按照业务语义将其命名为"签约记录"。联系转换后的关系模式为

签约记录(供应商编号,社区编号,签约时间)

在这种规则下,图 5-18 所示的 E-R 模型转换得到的关系模型为三个关系模式:

- 社区(社区编号,社区名称.....);
- 养老服务提供商(供应商编号,供应商名称.....);
- 签约记录(**供应商编号,社区编号,签约时间**)。

深入思考 5.5　对于多元联系(主要考虑常见的三元联系)的 E-R 模型,在转换为关系模式时,应遵循何种转换规则?

参考答案:请参见微课视频 5-5。

5.5 结构化设计

5.5.1 模块与结构图

1. 模块概念

模块是构成程序的基本构件。一个软件系统可以划分为多个子系统,每个子系统中包含多个业务功能,每个业务功能需要多个函数来解决。子系统、业务功能、函数都可以称为模块,只是粒度和层次不同而已。

对于大的模块,可以继续划分为功能独立的小模块,直到划分为不能再分解的原子模块。将系统划分为功能相对独立且可独立访问的模块的过程,称为模块化。

在结构化体系结构设计实践中,软件系统的设计一般不会划分到直接可以拿来编码的原子模块,而是划分到一个模块大小适度的层级,使得系统的开发成本最小。因为,当模块数量增加时,虽然模块的复杂程度减小使得单个模块的开发成本减小,但是模块间接口增多会使得接口开发所需的成本增大,因此不能无限地进行模块的分解,应找到一个模块数量和接口数量间的一个平衡点。

在软件体系结构设计过程中,模块有良好的独立性是系统设计良好的关键,即:模块应具有相对独立的功能且与其他模块间有简单的接口关系。那么如何衡量模块的独立程度呢?有两个定性的度量标准:耦合和内聚。耦合用于衡量不同模块间互相依赖的紧密程度,内聚用于衡量一个模块内部各个元素彼此结合的紧密程度。

2. 耦合

耦合是对模块间关联程度的度量。耦合的强弱取决于模块间接口的复杂性、调用模块的方式以及数据的传送量。模块间的耦合度越强,表明模块独立性越差,在软件设计时应尽量采取低耦合的系统设计。

根据控制关系、调用关系和数据传递关系的不同,耦合可以分为以下几类。

1) 非直接耦合

如果两个模块都能各自独立工作,相互之间没有直接关系,模块之间的交互不是通过直接的接口或通信方式进行,而是通过中间件或消息传递来实现,这种耦合称为非直接耦合。非直接耦合的耦合度最弱,模块独立性最强。但是在一个软件系统中,不可能所有模块之间都是非直接耦合关系,主模块与子模块之间必然存在控制和调用关系。

2) 数据耦合

如果两个模块之间只传递简单的数据参数信息(相当于高级编程语言中的值传递),这种耦合称为数据耦合。数据耦合是低耦合,系统中至少且必须存在这种耦合,因为只有某模块的输出数据作为另一模块的输入数据时,系统才能通过模块间的数据传递完成特定功能。

3) 标记耦合

如果两个模块之间传递的数据不是简单类型数据,而是由多个数据元素组成的复合数据结构,但是模块在调用过程中只使用到了其中一部分数据元素,这种耦合称为特征耦合或标记耦合。例如:模块 A 在调用模块 B 时只需要使用学号,但是在参数传递时,却传递了包含学号、姓名、性别、籍贯等多个数据信息的学生数据结构,此时就称发生了标记耦合。

4) 控制耦合

如果两个模块之间传递的信息是控制类型的数据信息(例如:标志、开关量,有时是以数

据参数的形式出现），这种耦合称为控制耦合。控制耦合增加了系统的复杂性，调用模块必须知道被调用模块的内部逻辑，增加了相互依赖。将被调用模块适当分解后，可以用数据耦合代替控制耦合。

5）外部耦合

外部耦合是指两个模块共享一个外部强加的数据结构、通信协议或者设备接口。外部耦合基本上与外部工具和设备通信有关。例如：在微服务框架 Dubbo 中，服务提供者与服务消费者通信时，共享一个外部强加的数据结构：包名、类名、字段名、字段类型、字段个数以及序列化版本号，要求它们都一致。

6）公共耦合

公共耦合是指多个模块访问同一个公共数据环境。公共数据环境可以是全局数据结构、共享的通信区、内存的公共覆盖区等。当全局数据或者某些模块发生变化时，公共耦合可能会导致产生不受控制的错误。公共耦合的复杂程度随着耦合模块个数的增加而显著增加。

7）内容耦合

如果一个模块与另一个模块的内部数据有关，不经调用直接使用另一个模块的程序代码或内部数据，那么这两个模块之间就存在内容耦合。如果模块 A 与模块 B 存在内容耦合，则当修改模块 B 引起程序代码或内部数据出错时，必然会引起模块 A 出错。另外，此时模块 A 的出错原因很难查找，给模块的修改、维护带来极大困难。内容耦合的内聚程度最低，在设计时应避免使用。

上面 7 种耦合类型的耦合性和模块独立性程度变化如图 5-28 所示。非直接耦合、数据耦合、标记耦合属于弱耦合，控制耦合属于中耦合，外部耦合、公共耦合属于较强耦合，内容耦合属于强耦合。

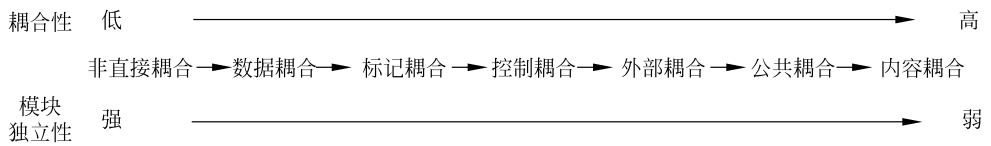


图 5-28　不同耦合类型的耦合性和模块独立性

3. 内聚

内聚是对模块内部各个元素彼此结合的紧密程度的度量。模块的内聚性越强，通常意味着模块间的耦合性越低，软件设计应尽量采用高内聚。

根据内聚性的高低不同，内聚可以分为以下 7 种类型。

1）功能内聚

功能内聚是指模块内各部分处理元素都围绕同一项功能协同工作，紧密联系、不可分割。功能内聚是内聚性最强的内聚。

2）顺序内聚

顺序内聚又称信息内聚，是指模块内的各处理元素都和同一个功能密切相关，而且这些处理必须顺序执行，通常前一个处理元素的输出数据是后一个处理元素的输入数据。

例如，某模块的功能为：逐个读取供应商的各项评估指标得分，计算供应商的总评分数，根据等级评价标准，评定出“优秀”“合格”“不合格”的等级。模块示意图如图 5-29 所示。该模块包含了“计算供应商评分”和“评定供应商等级”两个处理元素，它们彼此紧密联系，并且前一个处理的输出“供应商总评分数”是后一个处理的输入，它们是顺序执行的，则称该模块满足顺序内聚。

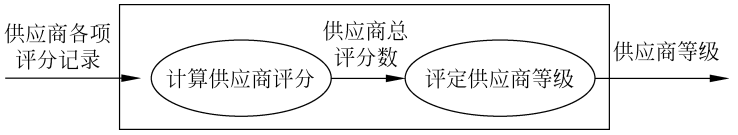


图 5-29　顺序内聚示例

3）通信内聚

通信内聚是指模块内的各处理元素都使用相同的输入数据或产生相同的输出数据。

例如，某模块的功能为：根据年接单记录生成个人接单记录、计算月平均接单量。模块示意图如图 5-30 所示。

该模块包含了“生成个人接单记录”和“计算月平均接单量”两个处理元素，它们的输入相同，都使用了“年接单记录”，并且它们之间没有顺序关系，彼此独立，则称该模块满足通信内聚。

4）过程内聚

过程内聚是指模块内的处理元素是相关的，而且必须以特定次序执行。

例如，某模块的功能为：生成并打印个人接单记录。模块示意图如图 5-31 所示。该模块包含了“生成个人接单记录”和“打印个人接单记录”两个处理元素，它们彼此相关，且必须按照特定的次序执行，则称该模块满足过程内聚。

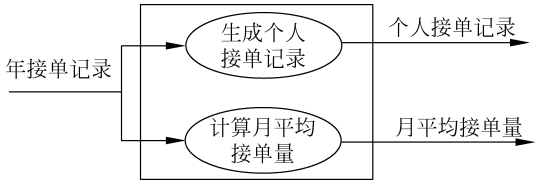


图 5-30　通信内聚示例

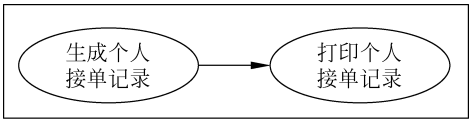


图 5-31　过程内聚示例

过程内聚与顺序内聚的区别是：在顺序内聚中，从一个处理单元流到另一个处理单元的是“数据流”；而在过程内聚中，从一个处理单元流到另一个处理单元的是“控制流”。

5）时间内聚

时间内聚又称经典内聚，是指一个模块内有多个功能，并且各个功能在特定时刻执行。例如：初始化系统模块、系统退出处理模块、紧急故障处理模块。

6）逻辑内聚

逻辑内聚是指一个模块把几种相关的功能组合在一起，每次被调用时，由传送给模块的控制型参数来确定该模块应执行哪一种功能。

例如，一个模块可以生成月报表、季度报表、年报表，具体生成哪个报表由传入的控制参数来决定，则称该模块满足逻辑内聚。

逻辑内聚的特点是：一个模块完成的任务在逻辑上属于相同或相似的一类，通过参数确定该模块完成哪一个功能。

7）巧合内聚

巧合内聚又称偶然内聚，是指一个模块内部各处理元素之间没有联系，或者即使有联系也是很松散的联系。

例如，一个模块中有两个方法：一个方法用于将字符串中的所有字母都变成大写；一个方法用于完成两个整数的求和。两个方法之间没有关系，只是简单地将一组功能放在同一个模块中，则称该模块满足巧合内聚。

巧合内聚模块是内聚程度最低的模块。它的缺点是模块的内容不易理解、修改和维护。上面 7 种内聚类型的内聚性和模块独立性程度变化示意图如图 5-32 所示。

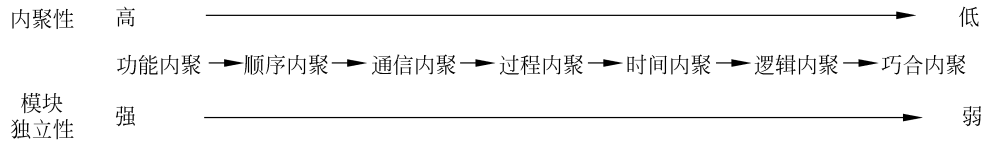
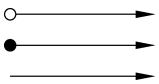
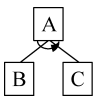


图 5-32　不同内聚类型的内聚性和模块独立性

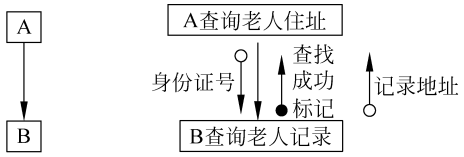
4. 结构图及组成

结构图(Structure Chart,SC)由 Yourdon 于 20 世纪 70 年代提出,早期被广泛应用于软件结构设计中,是准确表达软件结构的图形化工具,用来描述软件模块之间的层次调用关系和联系。软件结构图中的基本图形符号如表 5-4 所示。

表 5-4　结构图中的基本图形符号

图　例	含　义
	表示模块
	表示模块间的调用关系 表示调用模块指向被调用模块 表示直线上方的模块调用直线下方的模块,可以不用箭头表示模块间的调用关系
	表示模块间的信息传递 表示传递的是数据信息 表示传递的是控制信息 表示传递的信息不区分是数据信息还是控制信息
	表示模块 A 可以有条件地调用模块 B 或模块 C
	表示模块 A 可以反复地调用模块 B 和模块 C

利用表 5-4 中的基本符号表示结构图模块间的调用关系和信息传递,如图 5-33 所示。



(a) 模块调用关系　　(b) 模块间的信息传递

图 5-33　模块间的调用关系和信息传递

在图 5-33(a)中,用单向箭头连接两个模块,箭头方向由模块 A 指向模块 B,表示模块 A 调用了模块 B。在图 5-33(b)中,查询老人住址的模块 A 调用查询老人记录的模块 B,在调用过程中,模块 A 把身份证号作为数据信息传递给模块 B 作为查询条件,当模块 B 查询结束后,把是否查找成功的标记作为控制信息,把记录老人信息的记录地址作为数据信息,将两种信息都回送给模块 A。

5. 结构图中的指标

图 5-34 给出了一个常用的软件结构图层次结构，软件模块间的关系均可以表示为层次结构。

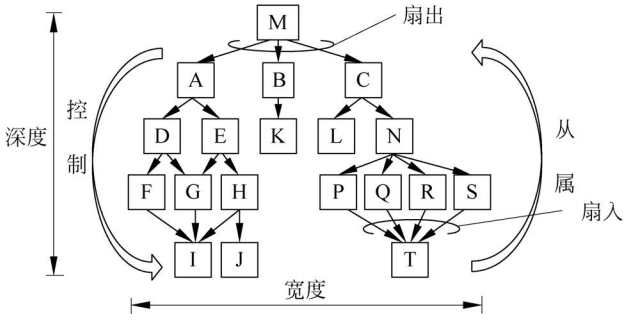


图 5-34　分层模块结构图示例

图 5-34 中的相关概念描述如下：

1) 深度

模块结构的控制层数称为结构图的深度，结构图的深度在一定程度上反映了系统的复杂程度。在图 5-34 中，结构图的深度为 5。

2) 宽度

在结构图中，位于同一层次的模块数的最大值称为结构图的宽度。在图 5-34 中，自上而下各层的模块数分别为 1、3、5、7、3，最大值为 7，所以此结构图的宽度为 7。

3) 扇出和扇入

扇出和扇入的概念都是针对结构图中某一个模块定义的。扇出表示一个模块直接调用的下级模块的数目；扇入表示直接调用一个模块的上层模块的数目。例如，在图 5-34 中，模块 M 直接调用了下级模块 A、B、C，那么模块 M 的扇出数为 3。模块 T 被上级模块 P、Q、R、S 直接调用，那么模块 T 的扇入数为 4。

结构图的扇出数和扇入数都不宜过高。扇出数太大，表明该模块需要调用的下级模块数量过多，控制过于复杂，可以通过增加中间层次来减小扇出数。扇入数大的模块因为被多个上级模块调用，所以通常为共用模块。扇入数越大，表示该模块被越多的上级模块共享，说明模块的复用率高，但不应片面追求高扇入数。例如，把彼此无关的功能放入同一个模块中，虽然扇入数提高了，但模块的内聚程度必然降低，违反了模块设计的高内聚原则。图 5-35 给出了一个扇入扇出数过大的改善方法示例。

在图 5-35(a)中，模块 M 的扇出数为 10。扇出改善后的模块结构如图 5-35(c)所示，增加了中间层次，使得模块 M 的扇出数为 2，模块 A、B 的扇出数为 4，不存在扇出数过大的模块。在图 5-35(b)中，模块 T 的扇入数为 10。扇入改善后的模块结构如图 5-35(d)所示，增加了中间层次，使得模块 T 的扇入数为 2，模块 I、J 的扇入数为 4，不存在扇入数过大的模块。

在模块设计过程中，一般扇出数平均为 3～4，通常不超过 7。扇入数可以较高，但不能为了追求高扇入数而违反模块的设计原则。一个好的软件结构图形态应整体均衡：顶层宽度小，中层宽度大，底层宽度次之；顶层扇出数较高，中层扇出数较少，底层扇入数较高。

5.5.2　基于数据流的体系结构设计

结构化体系结构设计的主要任务是：定义软件模块及其之间的关系，将模块组织为设计良好的层次系统。本书采用结构图来描述软件系统的体系结构，上层模块调用它的下层模块，

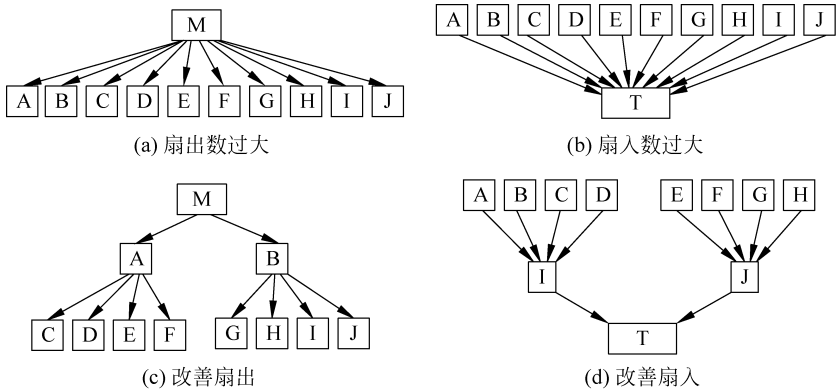


图 5-35　扇入扇出数过大的改善方法示例

下层模块继续调用其下属的更下层模块，以此类推，直到完成一个完整的程序功能。

在结构化设计方法中，软件体系结构通常由数据流图推导而来。基于数据流的体系结构设计方法的主要任务是：将数据流图映射为软件结构。

1．面向数据流方法的设计过程

典型的数据流类型包括：变换型数据流和事务型数据流。不同的数据流类型映射的系统结构也不同。通常情况下，一个系统中的所有数据流都可以认为是变换型数据流，若数据流具有明显的事务特征，则建议采用事务型数据流映射方法进行设计。面向数据流方法的设计过程如图 5-36 所示。

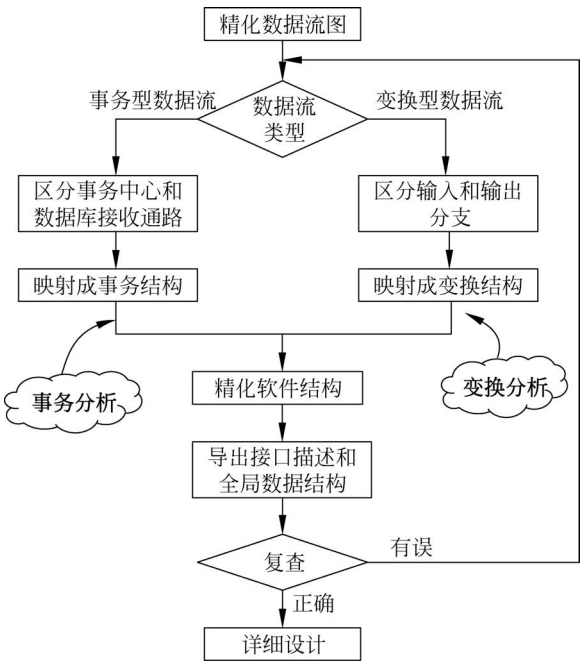


图 5-36　面向数据流方法的设计过程

在图 5-36 中，面向数据流方法首先对需求分析阶段得到的数据流图进行精化，以确保数据流图中的每一个处理都对应一个规模适当、相对独立的功能。然后根据不同的数据流图类型，采用不同的分析映射方法将其转换为初始的软件结构图。最后对软件结构作进一步分解和精化，针对软件结构图中的每一个模块，导出接口描述和全局数据结构描述。经过复查后正

确的体系结构设计方案将用于详细设计。

2. 变换型体系结构设计

1) 变换型数据流图

在变换型数据流图中,信息沿输入通路由外部进入系统,同时由外部形式变换成内部形式,经变换中心加工处理后再沿输出通路变换成外部形式离开软件系统。变换型数据流图整体结构可划分为三部分:输入、变换中心和输出,如图 5-37 所示。

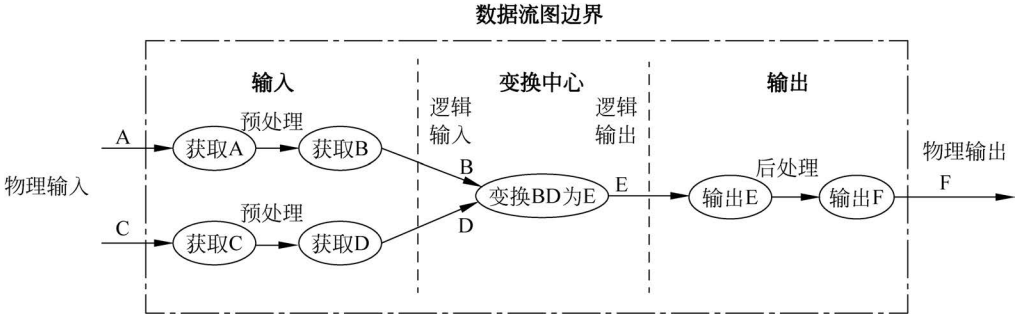


图 5-37 变换型数据流图整体结构

在图 5-37 中,从数据流图边界外部进入数据流图的数据 A 和数据 C 称为物理输入。数据 A 和数据 C 进入数据流图内部后,经过一系列的预处理,在到达变换中心前变换为能够直接加以处理的数据 B 和数据 D,数据 B 和数据 D 称为逻辑输入。变换中心将数据 B 和数据 D 加工处理为数据 E,数据 E 称为逻辑输出。数据 E 经过一系列的后处理变换为适合输出的数据 F,向外部输出的数据 F 称为物理输出。

2) 变换型数据流图映射为变换型系统结构图

一个数据流图对应着一个加工,而一个加工对应着某一个功能模块。因此,每一个数据流图将被映射为一个结构化功能模块,不同层级的数据流图对应着粒度不同的模块。

以图 5-37 的数据流图为例,将变换型数据流图映射为变换型系统结构图,如图 5-38 所示。

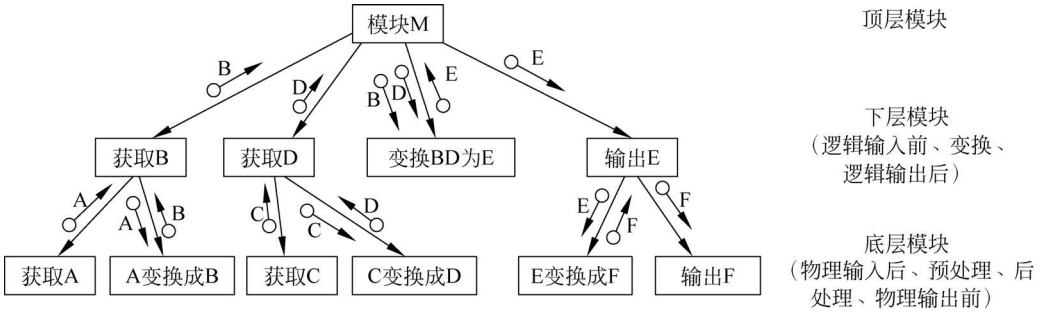


图 5-38 变换型系统结构图

在图 5-38 中,用自上而下的连线表示模块间的调用关系,在连线两侧,用空心尾部箭头表示数据传递的方向和内容。系统结构图的运行过程如下所述:

(1) 整个数据流图所描述的上层处理被映射为顶层模块 M,顶层模块 M 的操作为:首先获取逻辑输入数据 B 和 D,然后变换 BD 为 E,最后输出逻辑数据 E。这些操作被映射为模块 M 的下层模块:逻辑输入前模块“获取 B”“获取 D”、变换模块“变换 BD 为 E”和逻辑输出后模块“输出 E”。顶层模块 M 通过调用下层模块,实现将输入数据 B、D 转换为输出数据 E。

（2）每个下层模块继续调用的处理被映射为再下一层模块，直到数据流图边界处，此处的模块被称为底层模块。例如：逻辑输入模块“获取 B”分解后，由物理输入后模块“获取 A”和预处理模块“A 变换成 B”组成。模块“获取 B”通过调用下层模块，实现将物理输入 A 转换为逻辑输入数据 B。

从上述描述中可以看到，结构图是以模块的调用关系为线索，能够从宏观上反映软件的体系结构。

3. 事务型体系结构设计

1）事务型数据流图

在事务型数据流图中，信息沿输入通路由外部进入系统，在事务处理中心处进行分析判断后，选择执行事务分支中的某一项事务处理加工。事务型数据流图整体结构至少包括三部分：输入、事务处理中心和事务分支，如图 5-39 所示。至于事务执行完毕后是否向外部输出，则根据实际情况而定。

在图 5-39 中，从数据流图边界外部获取数据 A，进入数据流图内部后，经过一系列的预处理，在到达事务处理中心前变换为能够直接加以处理的数据 B，数据 B 进入事务处理中心后，进行分析判断，选择执行事务分支中的某一项事务处理 Tn。

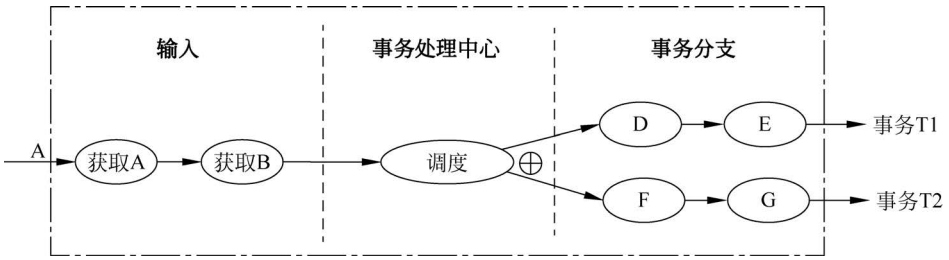


图 5-39　事务型数据流图

2）事务型数据流图映射为事务型系统结构图

以图 5-39 的数据流图为例，将事务型数据流图映射为事务型系统结构图，如图 5-40 所示。

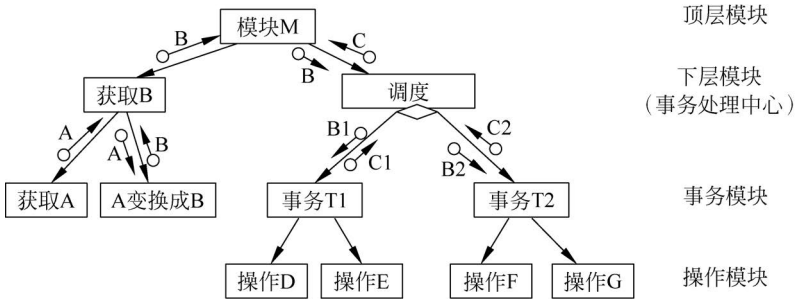


图 5-40　事务型系统结构图

在图 5-40 中，系统结构图的运行过程如下所述：

（1）整个数据流图被映射为顶层模块 M，顶层模块 M 的操作为：首先调用逻辑输入数据 B，然后在调度中心对数据 B 做出判断选择。这些操作被映射为对应的下层模块“获取 B”和“调度”，每个下层模块继续调用的操作被映射为再下一层模块，直到数据流图边界处。

（2）模块“获取 B”的分解类似于变换型数据流图对输入数据流的变换处理。

（3）在“调度”模块下，为每条事务处理分支设计了一个事务模块“事务 Tn”，每个事务模

块按照操作分解为下一层模块。例如：事务 T1 为事务模块，对应的“操作 D”和“操作 E”为操作模块。

深入思考 5.6 在图 5-40 中，如果调度中心得到的数据 C 作为模块 M 的输出，进行后处理之后，再向外部输出，则该图如何进行扩展？

参考答案：请参见微课视频 5-6。

深入思考 5.7 在实际应用中，系统的数据流图往往是变换型和事务型的混合结构。在图 5-40 中，如果调度中心得到的数据 C 作为上一层模块的输入，且上一层模块为变换型系统结构图，则该图如何进行扩展？

参考答案：请参见微课视频 5-7。

4. 基于数据流的体系结构设计案例

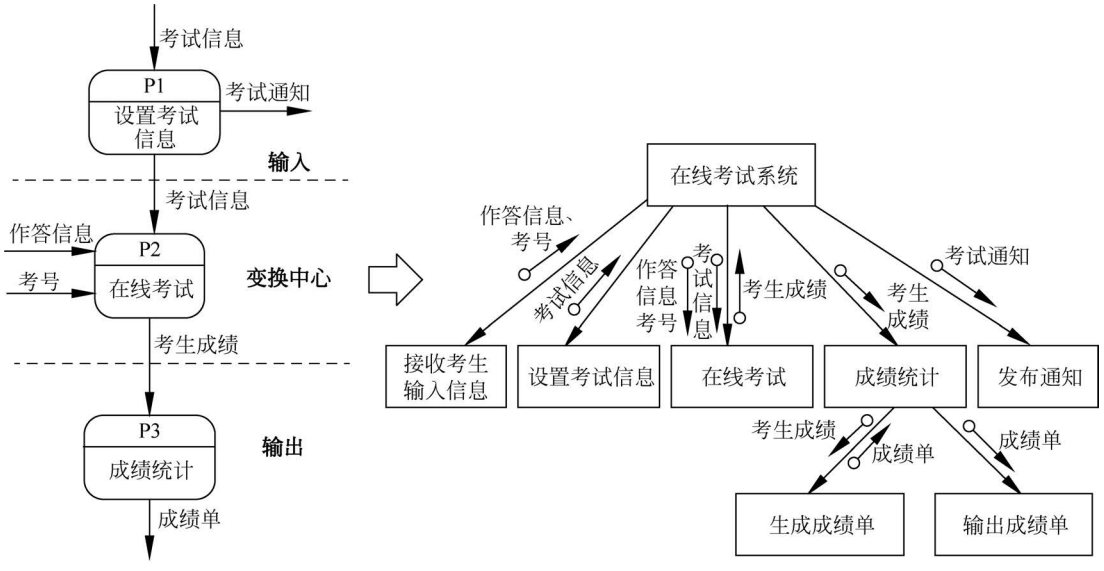
在结构化体系结构设计过程中，一个大型软件系统体系结构的设计通常以变换型映射方法为主，其他方法(主要是事务型映射方法)为辅，得到的系统结构图是一个变换型和事务型并存的混合结构。

下面在第 4 章分层绘制数据流图案例基础上，完成基于数据流的“在线考试系统”体系结构设计。转换过程分为以下几个步骤：

1) 精化一层数据流图，设计上层模块

在精化的一层数据流图中划分出输入、输出和变换三部分，根据变换型映射确定顶层模块和第一层模块。

将“在线考试系统”一层数据流图精化后转换为上层模块结构图，过程如图 5-41 所示。



(a) 精化后的“在线考试系统”一层数据流图

(b) 上层结构图

图 5-41 “在线考试系统”一层数据流图精化后转换为上层模块结构图

将图 4-15“在线考试系统”一层数据流图中的数据加工细节去除，只描述数据在系统中的流动，得到精化后的一层数据流图，如图 5-41(a)所示。按照变换型数据流图映射为变换型系统结构图的方法，图 4-14 顶层数据流图中的处理“在线考试系统”被映射为图 5-41(b)系统结构图中的顶层模块“在线考试系统”，图 5-41(a)中的输入 P1、变换中心 P2 和输出 P3 被映射为顶层模块的下层模块：“设置考试信息”“在线考试”“成绩统计”。

当数据流图并不满足标准时，映射过程可以灵活处理，以应对数据流图中不同的变化。在图 5-41 中，存在以下两种情形：

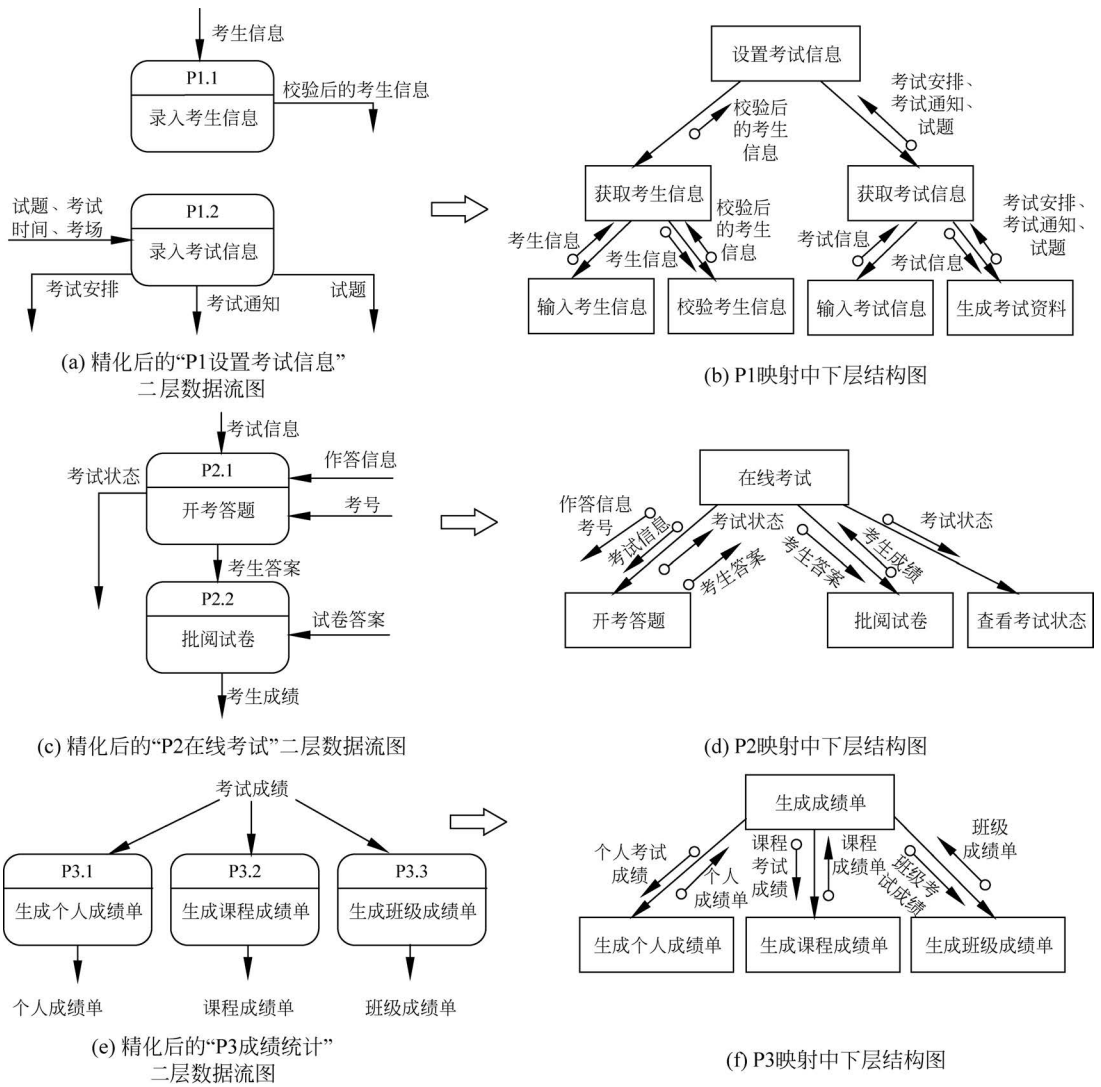
（1）输入模块的输出数据未经变换，直接输出。在图 5-41(a)中，P1“设置考试信息”并不是一个纯粹的输入，它在录入各种考试相关信息后，直接将考试通知发布给考生，而考试通知是一个直接对外的输出数据，因此并不符合一个标准的变换型数据流图。在此情形下，可以在图 5-41(b)中的输出部分增加一个“发布通知”输出模块。

（2）变换中心的数据直接来自外部输入。变换中心 P2“在线考试”未经输入模块，直接接收了来自外部的“作答信息”和“考号”，也不符合一个标准的变换型数据流图。在此情形下，可以在图 5-41(b)中的输入部分增加一个“接收考生输入信息”输入模块。经过上述处理后，变换后的系统结构图从整体上仍然保持自左向右为输入、变换和输出三部分。

2）精化二层数据流图，设计中层模块

根据输入、变换和输出三部分所细分的二层数据流图特点，区分变换型数据流图和事务型数据流图，利用相应的映射方法，确定其对应的模块结构图。

“在线考试系统”二层数据流图精化后转换为中下层模块结构图，如图 5-42 所示。



(a) 精化后的“P1设置考试信息”二层数据流图

(b) P1映射中下层结构图

(c) 精化后的“P2在线考试”二层数据流图

(d) P2映射中下层结构图

(e) 精化后的“P3成绩统计”二层数据流图

(f) P3映射中下层结构图

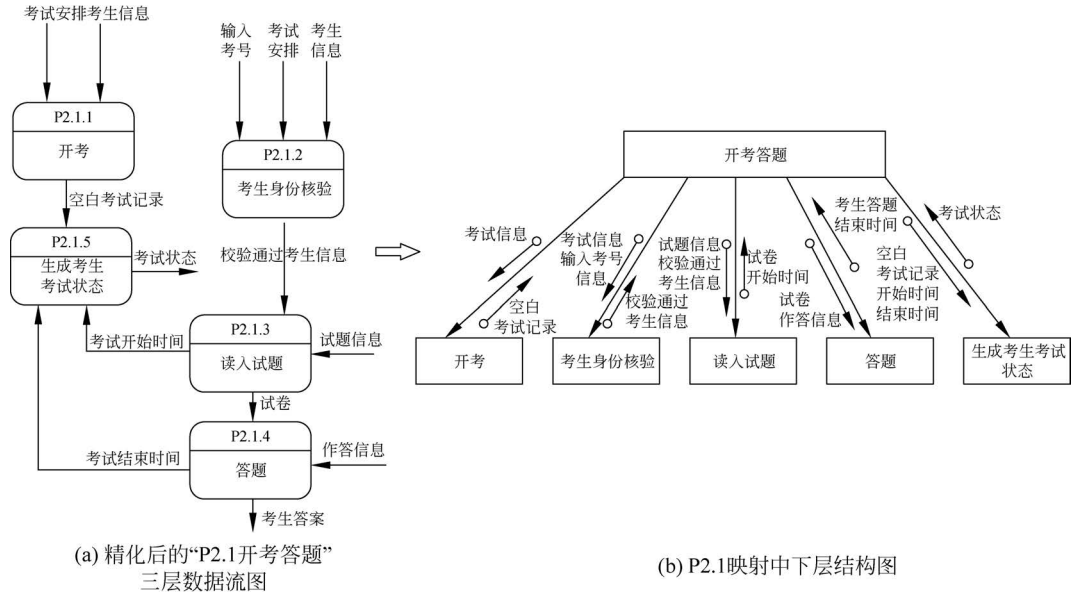
图 5-42　“在线考试系统”二层数据流图精化后转换为中下模块结构图

图 5-42(a)、(c)、(e)为“在线考试系统”精化后的各二层数据流图,映射转换后得到的变换型结构图分别对应图 5-42(b)、(d)、(f)。其中,(c)中的 P2.1 处理生成的数据“考试状态”是向外部输出的,映射转换处理为:在(d)所示的映射结构图中添加一个输出模块为“查看考试状态”。

3) 自顶向下,逐层分解转换

以“在线考试系统”中的“P2.1 开考答题”三层数据流图为例,将三层数据流图精化后转换为中下层模块结构图,如图 5-43 所示。

图 5-43(a)为精化后的“P2.1 开考答题”三层数据流图,映射转换后的变换型结构图为图 5-43(b)。在图 5-43 中,某些模块获取的输入数据并不一定来自同一层的输入模块,而是来自上层模块。例如:图 5-43(b)中的“答题”模块,“作答信息”由“开考答题”模块传输而来。在绘制结构图的过程中,要根据数据流图所表达的数据流向,正确地映射出输入和输出数据。



(a) 精化后的“P2.1开考答题”三层数据流图

(b) P2.1映射中下层结构图

图 5-43 “在线考试系统”三层数据流图精化后转换为中下模块结构图

4) 生成整体系统结构图

将上述各层数据流图逐级映射后得到的各部分结构图进行合并、优化,得到“在线考试系统”整体系统结构图,如图 5-44 所示。

图 5-44 并不是“在线考试系统”的完整系统结构图,例如:“批阅试卷”模块就应继续进行分解。得到初步的系统结构图之后,应在此基础上进行调整优化,例如:“查看考试状态”属于输出部分,可以将该模块调整为与“发布通知”“成绩统计”同一层次的输出模块。

5.5.3 模块详细设计

模块详细设计是在需求分析和概要设计的基础上,对概要设计阶段将系统分解后得到的模块进行细致和具体的设计。模块详细设计主要包括模块功能、接口、数据结构和算法等方面的设计。详细设计的主要工具包括程序流程图、盒图、PAD 图和伪代码等。通过图形化的方式或自然语言来指明控制流程、处理功能等算法实现细节,为后续的系统实现阶段提供编码依据。

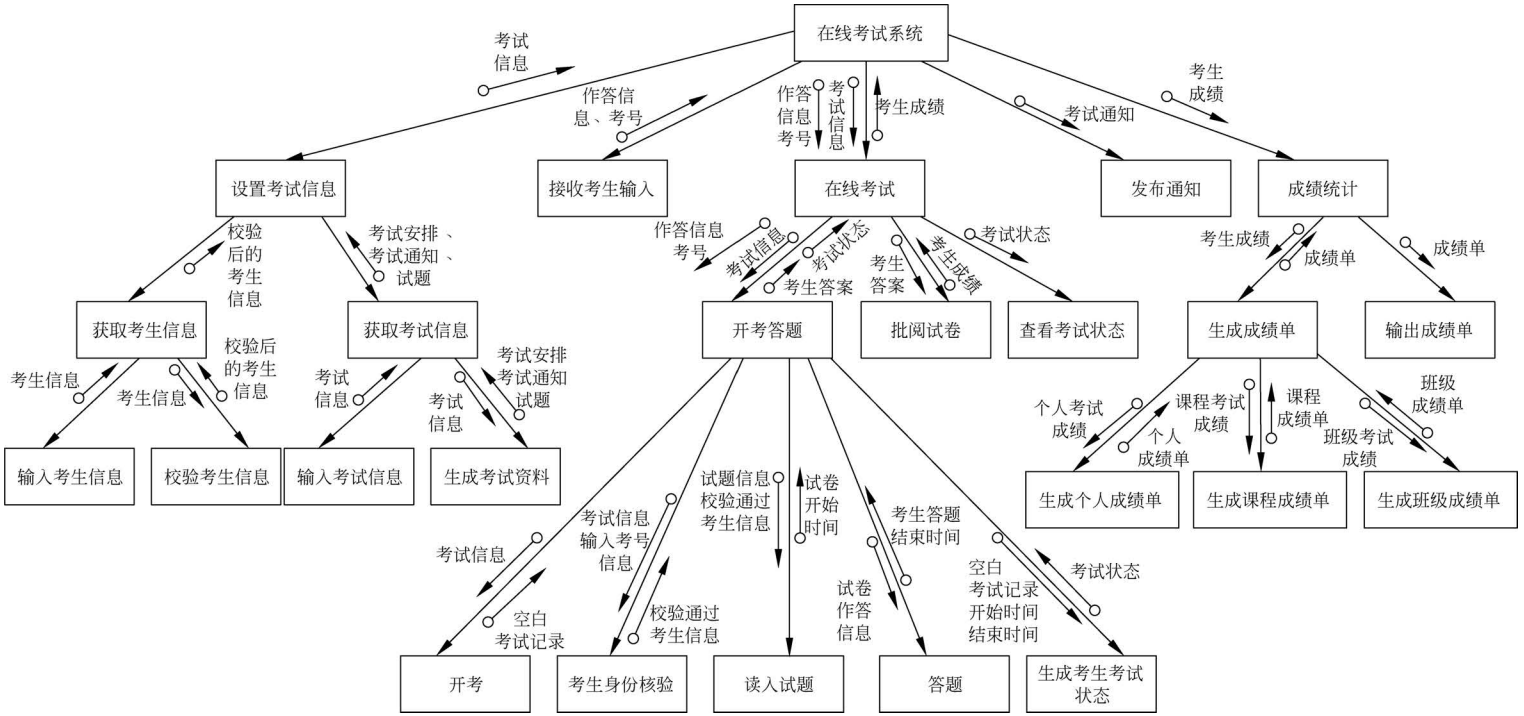


图 5-44 “在线考试系统”整体系统结构图

1. 程序流程图

程序流程图也称为程序框图,它将程序流程图形化,使程序流程的内容更加直观、清晰和易于理解,是一种常用的算法表达工具。当程序流程较为复杂时,通常会绘制一张反映程序控制逻辑的程序流程图来描述算法。

图 5-45 列出了程序流程图常用的基本符号。

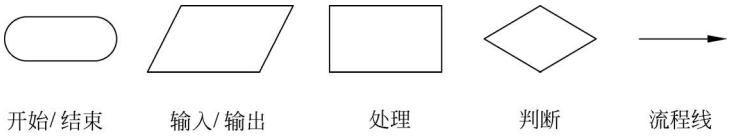


图 5-45 程序流程图基本符号

程序流程图自 20 世纪 40 年代到 70 年代中期,一直是主要的算法表达工具。任何程序逻辑都可用顺序、分支和循环这三种基本结构来表示。使用程序流程图描述结构化程序设计中的三种基本控制结构如图 5-46 所示。

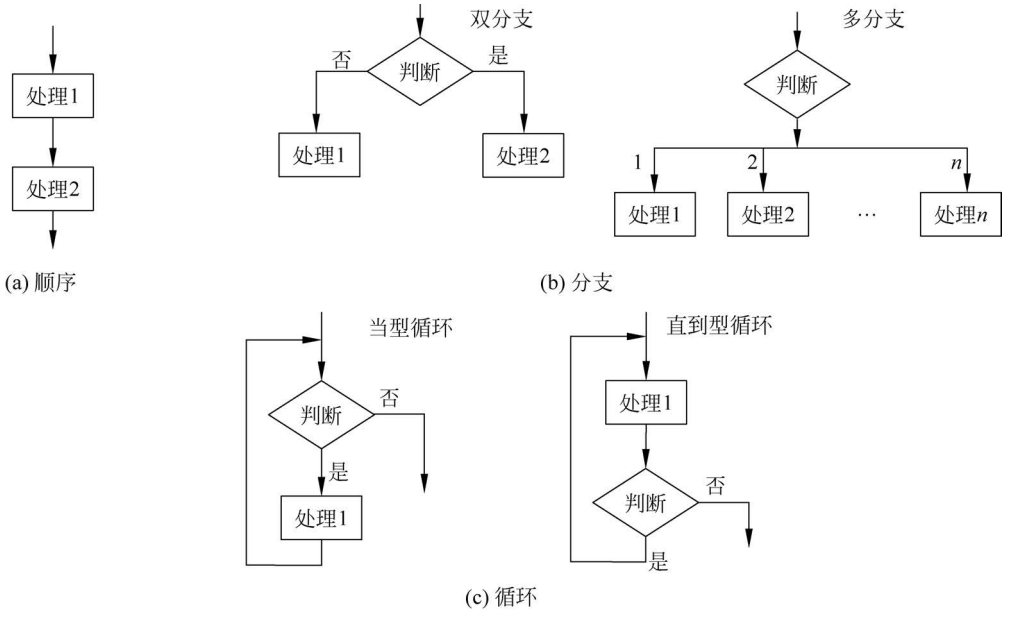


图 5-46 程序流程图的三种基本控制结构

1) 顺序结构

图 5-46(a)表示顺序结构,它是三种基本控制结构中最简单的一种,由几个连续的处理组成,表示处理按照流程线箭头所指的顺序依次执行。

2) 分支结构

图 5-46(b)表示分支结构,包括双分支和多分支两种形式。分支结构在菱形处判断给定的条件是否成立,然后由判断结果决定流程走向。

3) 循环结构

图 5-46(c)表示循环结构,分为当型和直到型两种循环方式。当型循环是指先判断给定条件,当条件满足时重复执行某一处理,当条件不满足时退出循环(处理执行次数为 0～n,处理可能一次都不执行);直到型循环是指先执行处理,然后对给定条件进行判断,当条件满足时重复执行该处理,当条件不满足时退出循环(处理执行次数为 1～n,处理至少执行一次)。

下面给出“智慧社区养老服务系统”案例中的养老券发放规则,试用程序流程图描述养老

券发放算法。

例：政府养老部门决定为社区内每一位适龄老年人发放养老券。养老券发放规则为：70～80 岁(包括 70 岁,不包括 80 岁)的老年人每月发放 10 张养老券；80 岁及以上的老年人每月发放 20 张养老券。

假设社区内老年人的总人数用 n 表示,系统当前日期用 $d1$ 表示,老年人的生日用 $d2$ 表示,老年人的养老券数量用 x 表示,老年人的年龄用 y 表示,getYear(起始日期,结束日期)方法能够获取两个日期之间的年数。

图 5-47 是养老券发放算法的程序流程图。图中涵盖了程序流程图的三种基本控制结构。

程序流程图的缺点是：用箭头代表控制流可以随意转移控制,容易造成非结构化的程序结构。

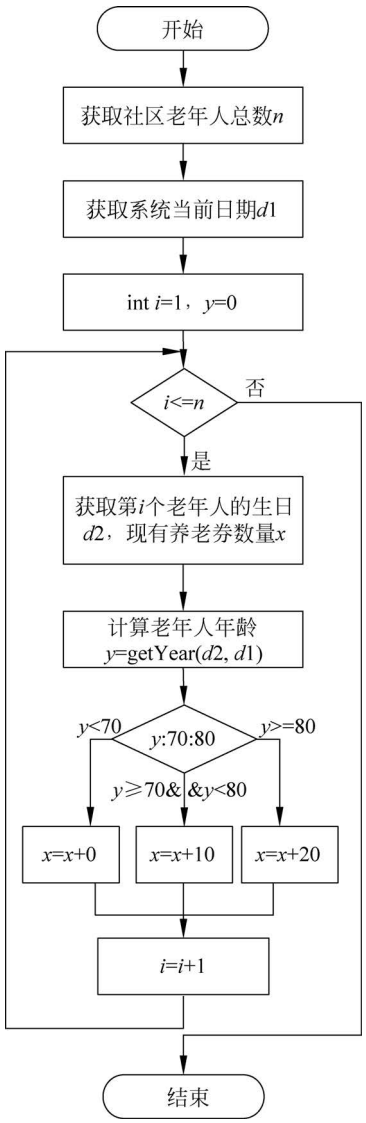


图 5-47　养老券发放算法的程序流程图

2. 盒图

盒图,又称为 N-S 图(N-S 由两个提出者 Nassi 和 Shneiderman 名字的第一个字母组成)。在 N-S 图中,全部算法写在一个大框图内,大框图又由若干个小的基本框图构成。同样地,盒图也可以有表示顺序、分支和循环的三种基本控制结构,如图 5-48 所示。

1) 顺序结构

图 5-48(a)表示顺序结构,按顺序先执行处理 1,再执行处理 2。

2) 分支结构

图 5-48(b)表示分支结构,包括双分支和多分支两种形式。双分支结构中,若条件成立,则执行 T(True)下面的处理 1,若条件不成立,则执行 F(False)下面的处理 2；多分支结构中,给出了多个执行出口,根据控制条件的取值相应地执行对应的处理。

3) 循环结构

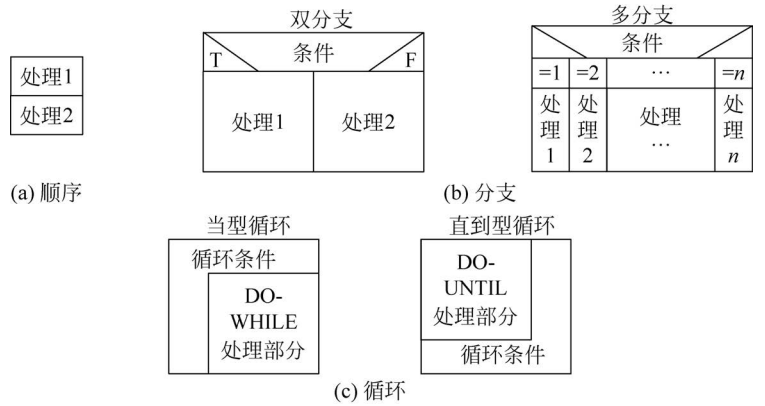
图 5-48(c)表示循环结构,分为当型和直到型两种循环方式。执行原理同程序流程图中的循环原理。

用盒图工具来描述上面案例中的养老券发放算法,如图 5-49 所示。

图 5-49(a)使用盒图基本控制结构绘制了养老券发放算法的盒图表示。图 5-49(b)是其扩展盒图表示。当算法逻辑较复杂或者在第一个盒图中暂不考虑更多算法细节时,可使用一个命名的椭圆框代替复杂逻辑部分,在其他图中展开描述椭圆框部分的算法。例如：在图 5-49(b)中,循环部分的处理逻辑用椭圆 k 标记,在右图中展开表示 k 的盒图。

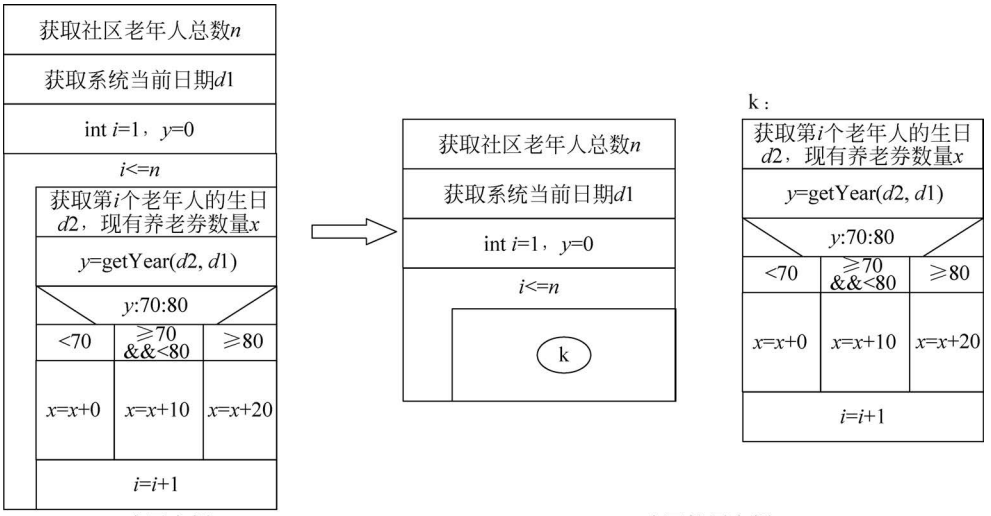
3. PAD 图

PAD 图又称问题分析图,是在程序流程图的基础上演化而来,用结构化程序设计思想表现程序逻辑结构的图形工具。同样地,PAD 图也可以有表示顺序、分支和循环的三种基本控制结构,如图 5-50 所示。



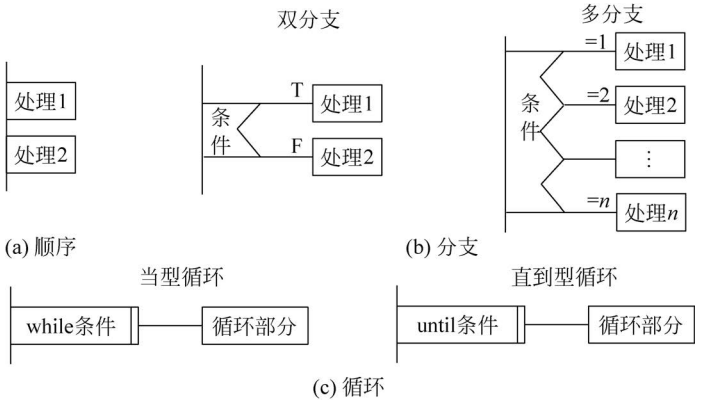
(a) 顺序　(b) 分支　(c) 循环

图 5-48　盒图的三种基本控制结构



(a) 盒图案例　(b) 盒图扩展案例

图 5-49　养老券发放算法的盒图

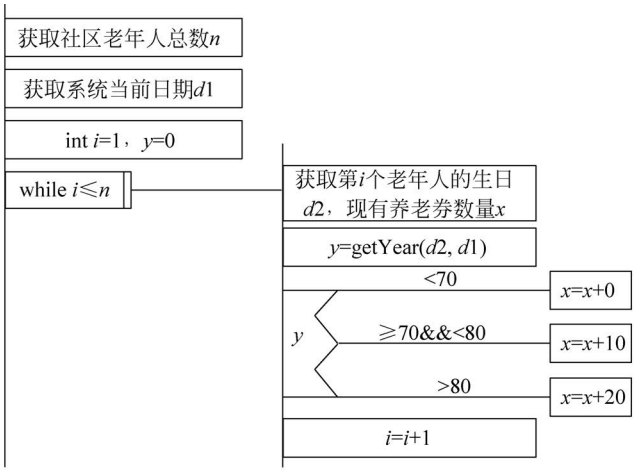


(a) 顺序　(b) 分支　(c) 循环

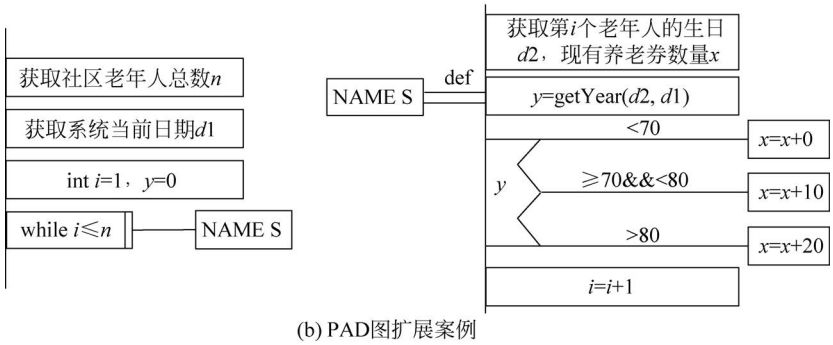
图 5-50　PAD 图的三种基本控制结构

在图 5-50 中，纵向表示系统的层次结构，横向表示系统的嵌套结构。最左侧的纵线是程序的主干流程，执行顺序是自上而下依次执行；当主干流程上的某一个处理存在分支或循环结构，则自左向右横向转入下一层；当横向嵌套结构执行完毕后，则返回上一层的纵线转入处，继续执行纵向流程，直到主干流程的末端为止。

用 PAD 图工具来描述养老券发放算法，如图 5-51 所示。



(a) PAD图案例



(b) PAD图扩展案例

图 5-51　养老券发放算法的 PAD 图

图 5-51(a)使用 PAD 图基本控制结构绘制了养老券发放算法的 PAD 图。图 5- 51(b)是图 5-51(a)的扩展 PAD 图表示。与盒图不同的是，PAD 图使用一个命名的矩形框代替复杂逻辑部分，用 def 及双线来定义子 PAD 图。例如，在图 5-51(b)左图中，循环部分的处理逻辑用矩形 NAME S 标记，NAME S 的 PAD 图在右图中进一步展开。

4. 伪代码

伪代码是一种非正式的、类似于英语结构的、用于描述模块算法设计和加工细节的语言。它的书写形式介于自然语言与编程语言之间。使用伪代码的目的是使被描述的算法可以更方便地以任何一种编程语言(C、Java、Python 等)实现，而不用拘泥于具体的实现语言。因此，伪代码必须结构清晰、代码简单、可读性好。

1) 伪代码的书写规则

伪代码没有技术规则。但为了阅读方便，伪代码仍然有一些约定俗成的书写规则。

(1) 通常情况下，每一条指令占一行，描述不出的指令可以用简单陈述句表示，语句后不接任何符号。

(2) 分支、循环语句应有严格的缩进，并有结束标记。

(3) 赋值可以使用 x←a，简单赋值也可以使用 x＝a。

(4) 输入用 input，输出用 output 或 return，推荐用 return。

2) 伪代码的程序控制结构

用伪代码编写的程序控制结构也有三种。

(1) 顺序结构。

顺序结构实际上是一个指令序列集合。可以用 Begin 作为开始,用 End 作为结束;也可以用“{”作为开始,用“}”作为结束,推荐用“{ }”;

(2) 分支结构。

```
if (条件) then
{
  ……
}
else if (条件) then
{
  ……
}
else
{
  ……
}
end if
```

其中,else if(条件) then 和 else 分支是可选的,可以根据实际需要灵活运用。

(3) 循环结构。

循环结构有两种表示方式。

```
for (var from s to f by incr) do
{
    ……
}
while(条件) do
{
    ……
}
```

其中,var 是循环变量;s 是循环变量初始值;f 是循环变量结束值;incr 是每次循环时,循环变量的倍增。

用伪代码工具来描述发放养老券的算法如下所示。

```
sendylq( ){
    n←获取社区老年人总数;
    d1←获取系统当前日期;
    int i = 1,y = 0;
    while(i < = n) do
    {
        d2←获取第 i 个老年人的生日;
        x←现有养老券数量;
        //getYear()方法能够获得两个日期之间的差值,精确获得老年人的年龄 y
        y = getYear(d2,d1);
        if (y < 70) then
        {
            x = x + 0;
        }
        else if (y < 80) then
        {
            x = x + 10;
        }
        else
        {
```

```
                x = x + 20;
            }
            end if
        }
    }
```

5.6　面向对象设计

5.6.1　基于多视图的体系结构设计

体系结构作为高层抽象，提供能被来自各种背景的系统参与者所接受的描述。系统参与者包括管理者、用户、客户、分析设计人员、软件实现人员和测试人员等。不同的参与者从不同的角度来理解体系结构，因此，体系结构应该有很多的维度，分别表述不同背景参与者的关注点。

在构建软件系统的体系结构模型时，单靠一个图形无法表示和系统体系结构相关的所有信息，因为不同的视角所关心的内容不同，所以通常需要使用多个不同的视图来呈现软件的体系结构。

1. RUP 4＋1 视图方法概述

Philippe Kruchten 于 1995 年提出了软件体系结构视图的 4＋1 视图方法，并最终被 RUP 采纳，演变为许多架构师所熟知的“RUP 4＋1 视图”，如图 5-52 所示。Philippe Kruchten 在其著作《Rational 统一过程引论》中写道：“一个架构视图是对于从某一视角或某一点上看到的系统所做的简化描述，描述中涵盖了系统的某一特定方面，而省略了与此方面无关的实体。”

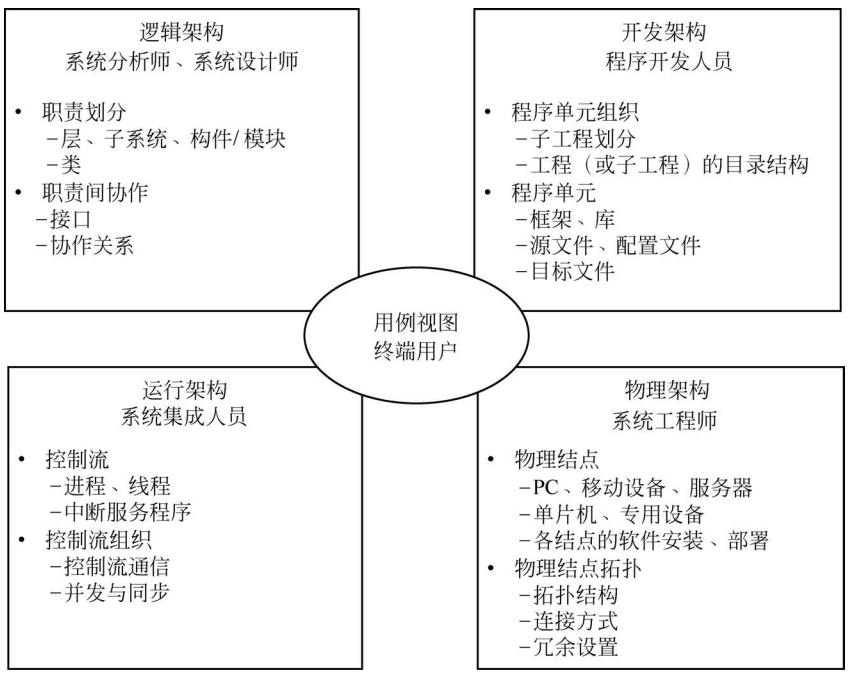


图 5-52　软件体系结构视图的“4＋1 视图”

图 5-52 中的“4＋1 视图”指的是“逻辑架构、开发架构、运行架构、物理架构”4 个架构视图和 1 个用例视图。

1）逻辑架构

逻辑架构从系统分析师和系统设计师的视角出发，主要关注系统体系结构如何支持系统功能需求，如何为最终用户提供服务。在逻辑架构视图中，根据职责划分，系统被分解成一系列的功能抽象。在面向对象方法中，这些抽象可能是逻辑层、子系统、构件/模块、类等。其中，构件为体系结构中能够独立运行的部件。功能间的协作通过接口或其他协作关系完成。

2）开发架构

开发架构从程序开发人员的视角出发，关注软件开发环境下程序单元组织和程序单元，侧重于软件构件/模块的组织管理。程序单元组织包括子工程的划分、工程(或子工程)的目录结构。程序单元包括开发技术和框架、程序库、源文件、配置文件、目标文件等。

开发架构是各种开发活动的基础，如：需求分配(将分解后的设计需求分配到具体的设计模块)、团队工作的分配、成本评估和计划、项目进度的监控、软件复用性、移植性和安全性等。

3）运行架构

运行架构从系统集成人员的视角出发，关注进程、线程和中断服务程序等运行时的概念，以及相关的并发、同步和通信等问题。运行架构考虑一些非功能性的需求，它解决并发性、分布性、系统完整性和容错性的问题。

4）物理架构

物理架构在 UML 中称为部署视图，从负责部署和运营维护的系统工程师视角出发，呈现由物理结点组成的拓扑结构。常见的物理结点包括计算机、移动设备、服务器、单片机、专用设备等；还需要关注各物理结点上支持系统正常运行所需的软件安装及部署，包括软件系统运行于何种操作系统之上，依赖于哪些软件中间件，是否有群集或备份等部署要求，驻留在不同机器上的软件部分之间采用何种通信协议等决策；物理结点的拓扑结构、连接方式、冗余设置等则用于明确物理结点间的关系。

5）用例视图

用例视图使上述四种视图有机联系起来，四种视图中的元素通过描述应用场景的用例协同工作。一方面，用例可以作为架构设计过程中发现架构元素的驱动因素；另一方面，用例可以作为架构设计结束后系统功能的验证和说明。因此可以说，用例是架构原型测试的出发点。

2. 逻辑架构分解

1）逻辑架构分解的原则

逻辑架构关注系统体系结构如何支持系统功能需求。从对应关系上，RUP 4＋1 视图方法中的逻辑架构与结构化方法中的系统功能结构图具有同等的作用，都用来描述系统的功能需求。

逻辑架构设计需要完成的任务是：将系统分解为子系统、将子系统划分为各个构件/模块。为了正确地进行系统分解，进而识别出各个构件/模块，需要遵循一些分解原则。常用架构分解原则有如下几种：

(1) 低耦合、高内聚：莱布尼兹指出："分解的主要难点在于怎么分。分解策略之一是按容易求解的方式来分，之二是在弱耦合处下手，切断联系"。高内聚、低耦合也是软件设计的基本原则，软件设计中的很多设计原则(例如：单一职责原则、依赖倒置原则、模块化封装原则)在架构分解中也是适用的。

(2) 层次性：分解通常是先业务后技术，循序渐进；先逻辑后物理，从上到下逐级进行分解展开：系统→子系统→构件/模块→类。

(3) 正交原则：和物理学中的正交分解类似，架构分解出的架构元素应是相互独立的，在

职责上没有重叠。

(4) 抽象原则：架构元素在较大程度上是架构师抽象思维的结果，架构师应该具备在抽象概念层面进行架构构思和架构分解的能力。

(5) 稳定性原则：将稳定部分和易变部分分解为不同的架构元素，稳定部分不应依赖易变部分。根据稳定性原则，将通用部分和专用部分分解为不同的元素；将动态部分和静态部分分解为不同的元素；将机制和策略分解为不同的元素；将应用和服务分离。

(6) 复用性原则：尽量重用系统已有的架构设计、设计经验、成熟的架构模式或参考模型、设计模式、领域模型、架构思想等。因为它们已经在不同的层次上分解识别出了许多架构元素，或者指出了一些分解方向，对架构分解具有借鉴和指导作用。例如：SOA 对 SOA 服务化具有重要的指导意义，可以参照它对系统进行初步的架构分解。

2) SOA 的分解

对于不同的架构，分解的构件元素是否合理也会有不同的评判标准。以 SOA 为例，在架构分解过程中，通过观察构件是否是独立的服务提供者来考察一个构件分解是否适当。当系统需要某一服务时，会调用相应服务的构件，而无需知道此构件位于何处，也无需知道该构件是用何种程序设计语言开发的。

在 SOA 中，子系统分解的重点工作是：从业务流程出发，从子系统的业务流程和用例模型中抽取服务，用服务描述子系统的业务能力。服务仍然是粗粒度的构件，它体现了内聚在服务模块中的业务逻辑。在进行服务模块识别的时候，应遵循业务逻辑高内聚的设计原则。

在识别服务的过程中，需要注意以下两点：

(1) 服务的划分要适应多种业务流程分支的情形，并考虑未来的需求扩展，以实现服务的可复用性。

(2) 识别服务时最易犯的错误是：把前期数据模型中的各个实体管理作为一个个服务模块，然后把实体对应的数据库表的 CRUD(增加(Create)、读取(Read)、更新(Update)、删除(Delete))操作作为接口向外暴露，这样做显著增加了服务模块间的调用频率，增加了模块间的耦合性，违反了模块设计高内聚的基本要求。

服务是比子系统粒度更小的构件，比服务粒度更小的构件是需求分析模型中的分析类。将构件分解到分析类粒度，系统功能概要设计就基本完成了。下一步将在构件详细设计中，完成将分析类转换为设计类的过程，也就是构件细化的过程。

3) 软件系统逻辑架构分解示例

以基于 Web 的自助下单系统为例，运用上述逻辑架构分解原则，得到自助下单系统的逻辑架构如图 5-53 所示。

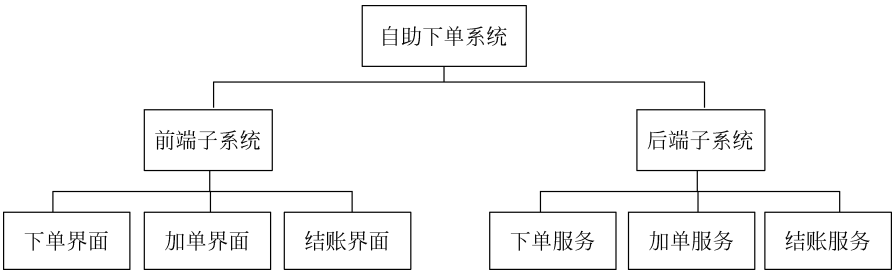


图 5-53　自助下单系统的逻辑架构

在图 5-53 中，自助下单系统分为前端子系统和后端子系统两个大的构件。前端子系统主要为顾客提供“下单”“加单”“结账”的操作界面，对应着分析类中的边界类。后端子系统主要

实现“下单”“加单”“结账”的业务逻辑,提供相应的服务。每个服务都被认为是后端子系统继续分解后得到的构件。各项服务的控制逻辑由分析类中的控制类和实体类交互协作完成。

图 5-53 是一个较为简单的系统逻辑架构图。对于功能复杂、规模较大的系统,逻辑架构图可以分为多个逻辑架构图。首先将系统划分为子系统,绘制一个由子系统组成的总的逻辑架构图;然后将每个子系统展开为更小粒度的构件结构,绘制多个由构件组成的子系统的逻辑架构图。

3. 开发架构的组织管理

在开发实践中,一些初学者会将程序开发人员比较熟悉的框架当作软件系统架构,例如:SSM 框架、Spring Boot 框架、Spring Cloud 微服务框架等。但开发中使用的框架只是技术架构中的框架选型,是软件系统架构模型中的一个侧面,因此,不能将框架等同于架构。框架的使用能够帮助程序开发人员完成系统运行的一些基础性技术工作。框架和业务无关,但是架构不能脱离业务,架构需要业务驱动去解决业务场景和问题。

开发架构侧重于程序单元的组织管理,与选定的技术框架密切相关。在图 5-52 中,开发架构的重要工作是确定工程的目录结构,也就是包。包结构是一个项目的骨架,包的组织是一个良好开发架构的关键要素之一,包结构设计的两种策略如下:

(1) 按业务功能组织包。根据不同的包名可以找到对应的功能或服务。将实现同一个功能所需的、位于不同分层中进行协同工作的代码放在一个包中,能够降低包与包之间的耦合度。

(2) 按分层结构组织包。根据不同的包名可以找到与技术框架对应的层。通常情况下,包中的元素间没有什么联系,降低了包的内聚性,提高了包之间的耦合度。实现同一个功能所需的代码按照技术框架的层级分布在不同包中。因此,修改一个功能需要同时修改多个包下的代码文件。

在实际开发中,上述两种策略通常会组合使用。无论采用何种策略,当项目组分工完成同一个系统时,各项目组所使用的开发架构应使用相同的程序单元组织管理策略,以便后期进行整合。

5.6.2 构件详细设计

1. 面向对象方法中的构件

构件是软件体系结构中可复用的软件组成部分,是设计、实现以及维护基于构件构造的系统的基础。构件有不同的粒度:类、类库、包、子程序等。但是,操作集合、过程、函数即使可以复用也不能称为一个构件,因为它们不能独立运行。

面向对象的观点认为,构件是一个协作类集合。面向对象详细设计不仅要阐述构件中每个类的属性及其相关操作,还应定义类与其他协作类之间的通信接口。

构件设计是面向对象详细设计的核心工作内容,构件的详细设计为下一步程序开发人员实现软件提供了依据,为程序开发人员提供充分的细节以便具体实现。

2. 类、构件和服务

类的概念是 20 世纪 80 年代初在面向对象分析与设计方法中最先提出的。面向对象观点认为,软件的运行是由类之间的交互协作完成的。20 世纪 90 年代末,开发人员认为类的粒度太小,提出了构件的概念和面向构件的设计方法。面向构件的设计方法认为软件可以分解成各个构件,在各个构件开发完毕后,通过整合形成软件系统,构件之间的关系是紧耦合的。21 世纪初,随着软件更新速度加快,用户需求也随之变动频繁,出现了面向服务的软件设计方法,强调应站

在用户需求的角度，从业务流程中提取识别各种服务。面向服务的软件系统架构是基于服务的设计，服务之间的关系是松耦合的。松耦合能够提高服务的复用性，也让业务逻辑变得可组合，每个服务可以根据使用情况做出合理的分布式部署，能灵活适应不断变更的用户需求。

总的来说，服务的概念源自用户需求，是从业务流程中抽取的业务能力，存在于业务架构中；构件是接近代码抽象级的一个模块化的构造块，存在于技术架构中；类是面向对象编程的基础，面向对象程序是由类组成的。

3. 构件设计的任务

在面向对象的软件工程方法中，构件设计的任务主要如下。

1）构件的静态模型设计

在系统需求分析阶段，对象模型定义了一组分析类，分析类的抽象级别相对较高。进入软件设计阶段后，对于需要细化的分析类，软件设计师应从代码实现的角度将其转化为一个由协作的设计类集合组成的构件，从而展开构件设计。

在构件设计过程中，分析类转化为设计类，抽象级别降低了。分析类使用业务领域术语描述数据对象以及数据对象所使用的相关服务；设计类更多地表现技术细节，用于指导实现。因此，从分析类细化为设计类的过程与选用的实现技术紧密联系，设计类的细化要在选定的技术框架下展开。

面向对象设计中，类是构件的组成要素，构件的静态模型设计内容主要包括两部分。

（1）对内描述组成构件的类。将需求模型中产生的分析类细化为设计类；定义设计类的属性和操作；描述操作的算法细节。在此过程中，使用类图来体现设计类之间的关系；使用包图来展现模型的逻辑组织；使用结构化设计中的程序流程图、伪代码等工具描述操作的算法。

（2）对外定义允许访问构件的接口。构件是独立可执行的部件，构件所提供的服务通过其接口获取。构件接口表示为参数化的操作，对构件接口的说明按照接口文档规定的内容进行。

2）构件的动态模型设计

UML 序列图、通信图和活动图等可以用来表示设计类之间的交互过程。在构件设计阶段，使用何种工具构建动态模型，并没有统一的标准，只要有助于设计人员理解系统和构件的工作过程即可。序列图是最易理解且最常用的动态建模工具，应该为每个重要的交互创建序列图。如果已经开发了用例模型，就应该为每个用例进行序列图建模。

4. 构件设计的原则

为了使构件设计在需求发生变更时能够更好地适应变更，最大限度地减少代码变动，类设计时有四个常用的基本设计原则，它们同样适用于构件设计。

1）开闭原则

开闭原则是指“对扩展开放，对修改关闭”，即尽量保持软件架构的稳定。开闭原则使用“接口和抽象类”保持抽象定义不变，使用从接口或抽象类中派生的实现类扩展需求的变化部分。

2）里氏代换原则

里氏代换原则是指子类可以替换它们的基类。换言之，子类可以扩展父类的功能，但不能改变父类原有的功能。也就是说，子类继承父类时，除了添加新的方法完成新增功能外，尽量不要重写父类的方法。

3）依赖倒转原则

依赖倒转原则是指构件依赖于抽象，而非具体实现。简单来说，依赖倒转的核心思想是

“面向抽象编程”。这里的抽象指的是接口或者抽象类,实现是指体现具体细节的实现类。遵循此原则会降低实现模块与调用模块之间的耦合。

4) 接口分离原则

接口分离原则是指一个类对另一个类的依赖应该建立在最小的接口上,即要为各个类建立它们需要的专用接口。不要试图建立一个庞大的接口,为所有依赖它的类提供调用服务。要尽量将庞大的接口拆分得更小更具体,让接口中只包含调用方感兴趣的方法。

5.7 面向对象系统设计的案例

5.7.1 案例的体系结构设计

面向服务的架构设计以业务为中心,在用例分析得到的系统功能需求基础上,对业务逻辑进行抽象和封装。从业务角度寻找服务,从架构角度强调服务的可复用性和可扩展性。

下面将在案例系统需求分析成果的基础上,选用面向服务的软件架构,采用“RUP 4+1视图”方法,对智慧社区养老服务系统进行体系结构设计。

1. 系统总体功能架构

将智慧社区养老服务系统的业务功能进行分析、汇总,绘制系统总体功能架构图如图5-54所示。

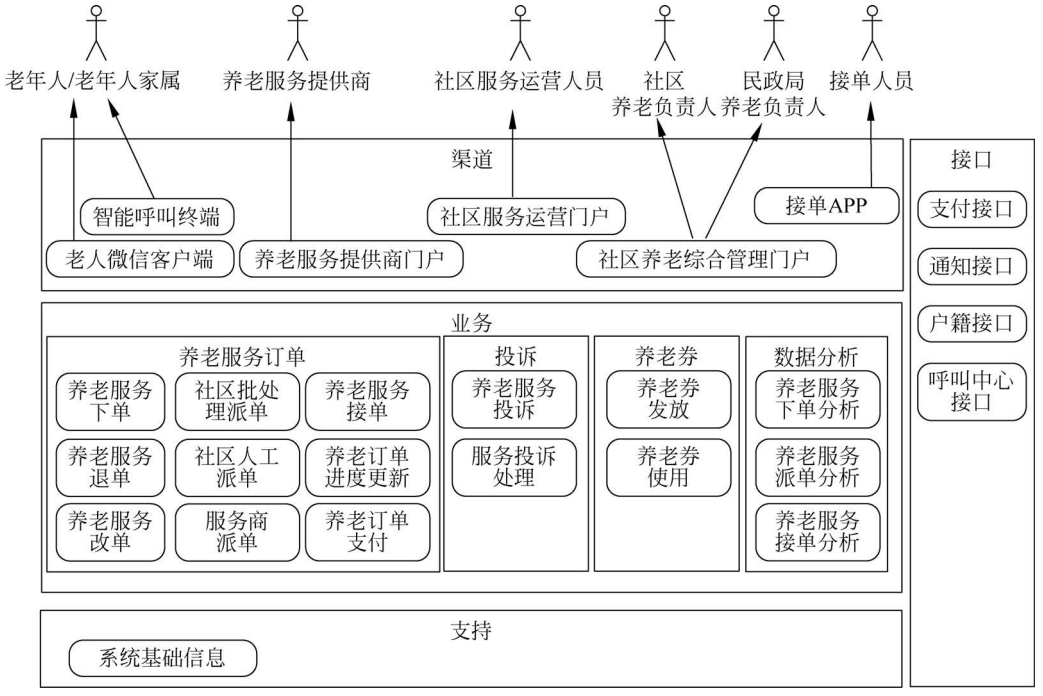


图 5-54 智慧社区养老服务系统总体功能架构图

在图5-54中,总体功能架构图采用了业界普遍使用的“上渠道、中业务、下支持、右接口”的风格。“渠道”为用户提供了访问系统的入口;“业务”是指系统为用户提供的各项功能;“支持”是指为了保障业务功能正常运转所需的系统基础功能模块;“接口”是指系统与外部交互所依赖的第三方接口。

2. 系统逻辑架构

1）系统划分为子系统

在图 5-54 总体功能架构的基础上，将功能架构图中的功能进行整合并映射到一组协同的应用程序中，完成子系统或构件的划分，得到系统的逻辑架构，如图 5-55 所示。

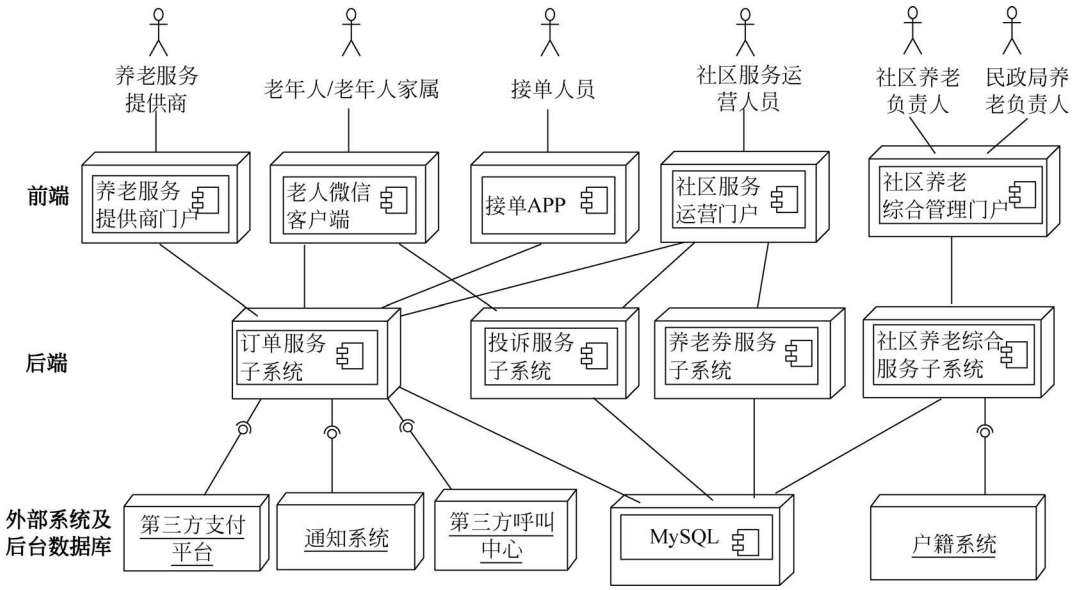


图 5-55　智慧社区养老服务系统逻辑架构图

在图 5-55 中，智慧社区养老服务系统被划分为 9 个子系统，与子系统有交互的元素包括 6 个参与者、1 个后台数据库 MySQL 和 4 个用下划线标识的外部系统，涵盖图 5-54 中的功能。9 个子系统划分如下。

（1）5 个前端子系统。

5 个前端子系统包括老人微信客户端、接单 APP、养老服务提供商门户、社区服务运营门户、社区养老综合管理门户。

（2）4 个后端子系统。

- 订单服务子系统：涵盖与养老服务订单相关的所有业务能力，包括养老服务下单、养老服务退单、养老服务改单、社区批处理派单、社区人工派单、服务商派单、养老服务接单、养老订单进度更新、养老订单支付。
- 投诉服务子系统：涵盖与投诉相关的所有业务能力，包括养老服务投诉、服务投诉处理。
- 养老券服务子系统：涵盖与养老券相关的所有业务能力，包括养老券发放、养老券使用。
- 社区养老综合服务子系统：涵盖与数据分析和系统基础信息相关的所有业务能力。数据分析包括养老服务下单分析、养老服务派单分析、养老服务接单分析。系统基础信息包括老年人基本信息、养老服务提供商基本信息、接单人员基本信息、社区服务运营人员基本信息、社区养老负责人和民政局养老负责人基本信息等。

2）基于子系统的工作分配

将系统分解为子系统后，软件开发组织的项目经理就可以将系统拆分为多个独立的开发项目（对应子系统或者组件）进行工作分配，每个项目可以进行独立的构建和编译。

案例的项目分组如表 5-5 所示。项目组被分为微信开发组、手机 APP 组、前端页面开发

组、后端开发组和接口组，分别负责不同的工作包，覆盖系统分解后的所有目标产品。

表 5-5 项目分组

序号	目标产品	工作包	项目组
1	老人微信客户端	微信开发工作包	微信开发组
2	接单 APP	手机接单 APP 工作包	手机 APP 组
3	养老服务提供商门户	养老服务提供商前端工作包	前端页面开发组
4	社区服务运营门户	社区服务运营前端工作包	
5	社区养老综合管理门户	社区养老综合管理前端工作包	
6	订单服务子系统	订单服务工作包	后端开发组
7	投诉服务子系统	投诉服务工作包	
8	养老券服务子系统	养老券服务工作包	
9	社区养老综合服务子系统	系统基础信息工作包	
		数据分析工作包	
10	与外部系统的接口	接口工作包	接口组

3) 子系统的分解

从图 5-55 中选取“订单服务子系统”，根据子系统所涵盖的业务能力(即系统提供的服务)，完成子系统的构件分解，如图 5-56 所示。

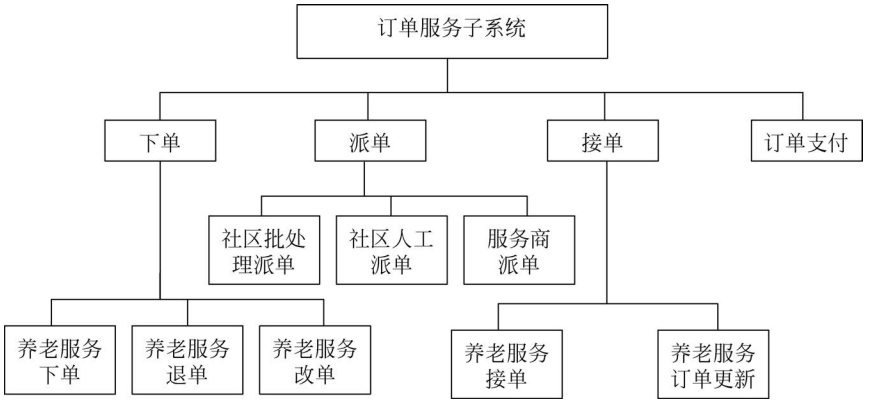


图 5-56 “订单服务子系统”分解

3. 子系统开发架构

系统逻辑架构完成了从业务场景层到应用服务层的映射，开发架构则主要完成从应用服务层到应用程序层的映射，从系统实现的角度对系统开发架构进行设计。开发架构侧重于程序单元的组织管理，与选定的技术框架密切相关。案例将“订单服务子系统”看作一个构件，技术选型选用层结构的 Spring Boot 框架。“订单服务子系统”包的设计采用先按业务领域划分外层包，再按技术框架划分内层包的混合包组织策略，包的组织如表 5-6 所示。

表 5-6 “订单服务子系统”包的设计

序号	外层包名	外层包的功能	内层包名(与框架分层对应)
1	com. oldman. area. order	订单领域服务	controller(对应控制层)
			service service. impl(对应服务层)
			dao mapper(对应数据操作层)
			pojo(对应实体层)
2	com. oldman. area. util	工具类库	
3	com. oldman. area. config	配置类库	

1）外层包

首先按照“域名类型.域名.项目名称”的规范，将项目总的工程包命名为“com.oldman.area”，然后外层包按“订单服务子系统”所属的领域类 order 命名。另外，创建 util 包存放工具类，config 包存放配置类，为服务实现提供公共支持。

2）内层包

内层包按照 Spring Boot 技术框架的分层结构来组织。controller 包对应控制层，用来放置控制类；service 包对应服务层，用来放置服务接口；service.impl 包用来放置服务实现类；dao 包对应数据操作层，用来放置数据访问接口；mapper 包用来放置数据访问接口实现的 xml 文件；pojo 包对应实体层，用来放置数据实体类。注意：框架中的接口是面向对象程序设计语言中的接口概念，与接口设计中的接口含义不同。

5.7.2　案例的接口设计

1. 外部接口设计

在图 5-55 中，外部接口两端用“棒棒糖”符号—◎—连接，规则为：**接口提供方—◎—接口调用方**。从外部接口连接符号所连接的接口两端可知，本节案例系统与 4 个外部系统有交互，包括户籍系统、第三方支付平台、通知系统、第三方呼叫中心。其中，社区养老综合服务子系统与户籍系统的接口、订单服务子系统与第三方支付平台的接口和订单服务子系统与通知系统的接口，都是由外部系统提供接口，本案例的子系统负责直接调用外部系统提供的接口，无需进行外部接口设计；订单服务子系统与第三方呼叫中心的接口是由本案例的订单服务子系统提供接口，需要进行外部接口设计。

在订单服务子系统与第三方呼叫中心的交互中，订单服务子系统是紧急呼叫自助下单业务的下游系统，第三方呼叫中心是上游系统，流程的实时性要求高，不能有延迟。这种情况下，因为下游系统无法知道何时需要紧急自助下单，这时最好的方式是由订单服务子系统提供接口，让呼叫中心主动将发出紧急呼叫的智能终端数据同步过来，完成自助下单。外部接口设计如表 5-7 所示。

表 5-7　“订单服务子系统”与“第三方呼叫中心”外部接口设计列表

接口名称	**紧急呼叫自助下单**			
接口描述	接收第三方呼叫中心推送的呼叫智能终端编号，完成自动下单			
接口应用场景	老年人紧急呼叫时，按下智能终端按钮，通过第三方呼叫中心连接订单服务子系统，自动生成紧急服务订单			
方法定义	public JsonResult sosOrder (String deviceId)			
请求参数 body	**参数名**	**参数类型**	**是否必填**	**说明**
deviceId	智能终端 id	String	Y	智能终端的标识
返回参数	**参数名**	**参数类型**	**说明**	
返回状态码	code	int	返回值为 200：成功；返回值非 200：失败	
返回消息	msg	String	code 非 200 时，给出的错误提示	
返回数据	data	可选		

2. 内部接口设计

在图 5-55 中，系统内部前端子系统与后端子系统之间的交互用直线连接。对于前后端分离的 Web 应用程序开发，前端负责向后端发起请求，由后端提供业务逻辑实现的接口，供前端子系统调用。选取“派单给服务商”用例，绘制在子系统级别的序列图，如图 5-57 所示。通过描述子系统与外部构件或系统的交互，识别“订单服务子系统”与其他子系统间的接口。

在图 5-57 中,社区服务运营人员通过“社区服务运营门户”前端向“订单服务子系统”后端发起各种请求;“订单服务子系统”通过查询后台数据库、调用“社区养老综合服务子系统”和“通知系统”,完成前端的各种请求并返回响应信息。

从子系统序列图中寻找接口的规则为:凡是由其他子系统指向目标系统的消息,都是目标系统应该提供接口。按照这个规则,观察在图 5-57 序列图中由其他子系统指向“订单服务子系统”的消息,获取其接口设计列表,如表 5-8 所示。

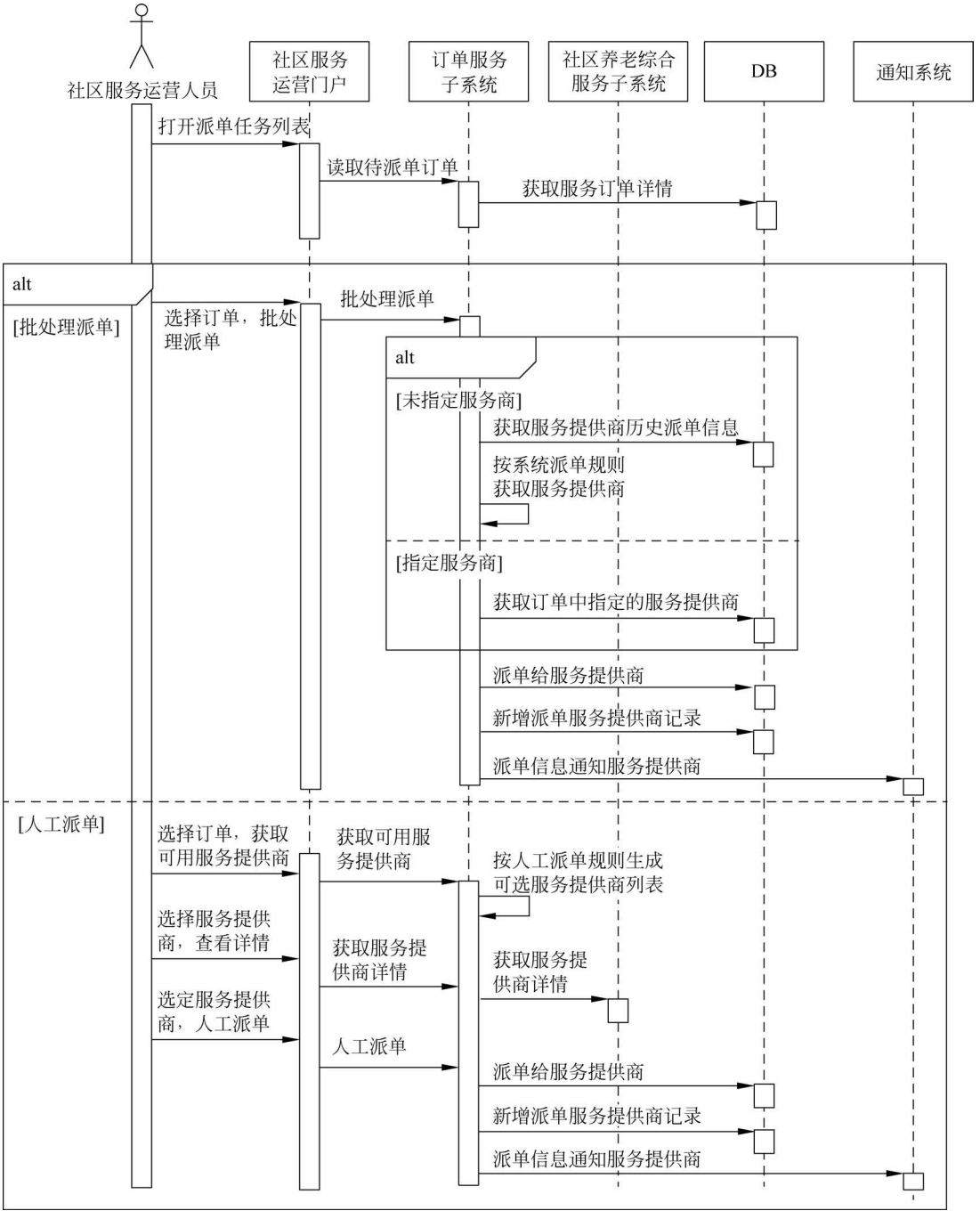


图 5-57　子系统交互序列图

表 5-8　“订单服务子系统”接口设计列表

接口名称	**1. 读取待派单订单**
接口描述	读取订单状态为“未派单”的养老服务订单列表信息
方法所在类	com. oldman. area. order. controller. DispatchController
方法定义	public JsonResult getUndispatchOrderList( )
映射 URL 地址	/order/dispatch/getUndispatchOrderList
请求方式	GET
接口名称	**2. 批处理派单**
接口描述	完成服务申请的批处理派单
方法所在类	com. oldman. area. order. controller. DispatchController
方法定义	public JsonResult batchDispatch(String[] orderid)
映射 URL 地址	/order/dispatch/batchDispatch
请求方式	POST
接口名称	**3. 获取可用服务提供商**
接口描述	获取能提供订单指定服务类型的服务提供商列表
方法所在类	com. oldman. area. order. controller. DispatchController
方法定义	public JsonResult getProviderListBySid(String serviced)
映射 URL 地址	/order/dispatch/getProviderListBySid
请求方式	GET
接口名称	**4. 获取服务提供商详情**
接口描述	获取指定服务提供商详情
方法所在类	com. oldman. area. order. controller. DispatchController
方法定义	public JsonResult getProviderByPid(String providerid)
映射 URL 地址	/order/dispatch/getProviderByPid
请求方式	GET
接口名称	**5. 人工派单**
接口描述	完成服务申请的人工派单
方法所在类	com. oldman. area. order. controller. DispatchController
方法定义	public JsonResult personDispatch (String orderid)
映射 URL 地址	/order/dispatch/personDispatch
请求方式	POST

表 5-8 中的“方法所在类”“方法定义”“映射 URL 地址”在接口设计中的意义是：后端通过将控制层中控制类的方法映射为 URL 地址的方式，向前端提供接口。前端通过访问包装为 URL 地址的接口，实现对后端服务的调用。

表 5-8 并不是完整的接口列表，列出的只是在完成“派单给服务商”用例过程中，“订单服务子系统”需要提供的接口。通过绘制子系统中其他各用例的子系统交互序列图，可以识别出完整的子系统接口列表。

表 5-8 所示的接口列表是一个简单的接口介绍概览，对于每一个接口应提供更加详细的接口说明。选取“获取服务提供商详情”接口，给出一个 Web 应用开发前端界面调用后端服务器的接口说明文档示例，如表 5-9 所示。

表 5-9 接口说明文档示例

接口名称	4. 获取服务提供商详情			
接口描述	获取指定服务提供商详情			
方法所在类	com. oldman. area. order. controller. DispatchController			
方法定义	public JsonResult getProviderByPid(int providerid)			
映射 URL 地址	/order/dispatch/getProviderByPid			
请求方式	GET			
请求参数 body	**参数名**	**参数类型**	**是否必填**	**说明**
providerId	服务提供商 id	String	Y	服务提供商的标识
返回参数	**参数名**	**参数类型**	**说明**	
返回状态码	code	int	返回值为 200：成功；返回值非 200：失败	
错误消息	msg	String	code 非 200 时,给出的错误提示	
返回数据	data	JSON	code 为 200 时,返回数据的具体信息	
返回数据具体信息	**返回值类型：JSON**			
data				
id	主键 id	String	标识数据 data 的唯一性	
serviceId	服务 id	String	服务标识	
serviceName	服务名称	String	服务名称	
providerId	服务提供商 id	String	服务提供商标识	
providerName	服务提供商名称	String	服务提供商名称	
areaId	社区 code	String	社区编码	
providerAddress	详细地址	String	服务提供商详细地址	
coordinateMap	坐标	JSON	服务提供商地址坐标	

请求格式：

```
{
   " providerId ":1029
}
```

返回格式：

```
{
  "success":true,
  "code":200,
  "msg":"成功",
  "data":{
     "id":30,
     " serviceId ":"10009",
     " serviceName ":"配餐",
        " providerId ":"1029",
      " providerName ":"山东怡老公司",
      "areaId":"110101",
      " providerAddress ":"山东省济南市历下区舜耕路 138 号天鸿广场 6 层 617 号",
      "coordinateMap":{
          " latitude ":"116.412549",
          " longitude ":"39.913736"
      }
    }
  }
```

深入思考 5.8　图 5-57 中，在人工派单过程中，订单服务子系统需要获取服务提供商详情时，为什么不直接从后台数据库获取相关数据，而是从社区养老综合服务子系统获取服务提供商基本信息？

企业观点：如果订单服务子系统直接从后台数据库获取服务提供商详情，其他子系统也采用同样的访问方式获取服务提供商信息，那么当养老服务提供商的表数据结构改变时，所有与该表有数据交互的子系统都需要修改相应的访问代码。这种做法会带来两个问题：一是代码变更量大，二是遗留未修改的代码可能为确保系统运行中的数据一致性带来隐患。如果把服务提供商信息的增删改查操作统一交由社区养老综合服务子系统，把对服务提供商的数据操作封装为对外开放的接口，那么当服务提供商表结构调整时，只需要改变相应的接口实现代码，其他子系统的调用代码都可以保持不变。

详情参见微课视频 5-8。

3. 用户界面设计

以“派单给服务商”用例为例，设计该用例的外部参与者“社区服务运营人员”的业务界面，重点介绍描述用例执行过程中页面跳转关系的页面流程图。根据用例描述，绘制“派单给服务商”用例的页面流程图如图 5-58 所示。

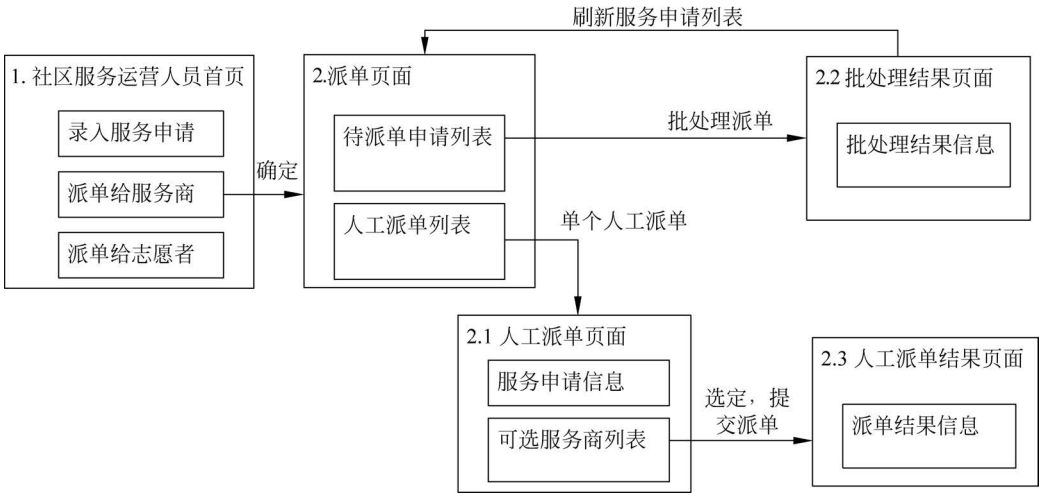


图 5-58　“派单给服务商”用例页面流程图

在图 5-58 中，“派单给服务商”用例共涉及 5 个页面，用例执行过程所涉及的页面跳转流程如下：

（1）社区服务运营人员登录后进入该角色的任务操作界面“1. 社区服务运营人员首页”。在该操作界面中放置了三个导航对象，对应着用例图中社区服务运营人员发起的三个用例。这三个导航对象可能是菜单，也可能是按钮。在单个页面具体设计时选择导航对象的类型。

（2）选择“派单给服务商”操作后，页面跳转到“2. 派单页面”。首先对所有的待派单服务申请进行批处理派单：若申请中没有指定养老服务提供商，则系统根据“系统派单规则”自动派单给某养老服务提供商；若申请中有指定养老服务提供商，则系统根据养老服务申请中的要求，派单给指定的养老服务提供商。选定一批服务申请，提交批处理派单，系统处理完毕后，将弹出“2.2 批处理结果页面”。系统将批处理结果反馈给社区服务运营人员并刷新“2. 派单页面”中的待派单列表，那些批处理派单未成功的申请将被加入到人工派单列表中。

（3）社区服务运营人员从“人工派单养老服务申请列表”中，选中某一条申请，提交“人工

派单”,页面跳转到“2.1 人工派单页面”。系统显示该服务申请的详情及派单操作界面。根据“人工派单规则”选择某养老服务提供商后,提交派单申请,将弹出“2.3 人工派单结果页面”。系统将人工派单处理结果反馈给社区服务运营人员。

绘制完页面流程图后,遵循用户界面应有的特性,可以着手开始单个页面的设计。由于这部分内容涉及一些美工专业知识,本书不展开描述。

5.7.3 案例的数据库设计

1. 智慧社区养老服务系统数据库的概念设计

根据案例的需求分析调研,绘制“智慧社区养老服务系统”的数据库概念设计模型,如图 5-59 所示。在图 5-59 中,每个实体及其属性的 E-R 图不再绘制,重点关注实体及实体之间的联系,各实体间联系的语义如下:

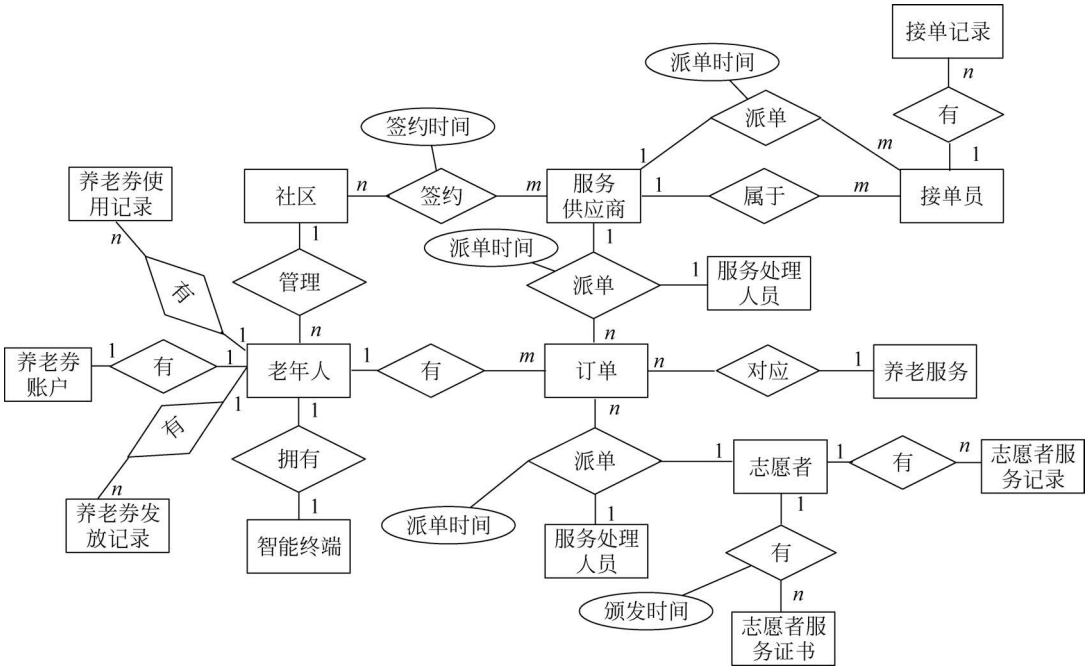


图 5-59 “智慧社区养老服务系统”E-R 图

(1) 在社区和养老服务提供商实体之间,一个社区可以签约多个养老服务提供商,一个养老服务提供商可以签约多个社区。

(2) 在社区和老年人实体之间,一个社区可以管理多个老年人,一个老年人只能属于一个社区管理。

(3) 在老年人和智能终端实体之间,一个老年人只能匹配一个智能终端,一个智能终端只能属于一个老年人。

(4) 在老年人和订单实体之间,一个老年人可以有多条订单,一条订单只能属于一个老年人。

(5) 在老年人和养老券发放记录实体之间,一个老年人可以有多条养老券发放记录,一条养老券发放记录只能属于一个老年人。

(6) 在老年人和养老券使用记录实体之间,一个老年人可以有多条养老券使用记录,一条养老券使用记录只能属于一个老年人。

(7) 在老年人和养老券账户实体之间,一个老年人只能有一个养老券账户,一个养老券账户只能属于一个老年人。

(8) 在社区服务运营人员、养老服务提供商和订单实体之间,一个社区服务运营人员为一个养老服务提供商可以派发多个订单,一个社区服务运营人员派发的一个订单只能属于一个养老服务提供商;一个养老服务提供商的某一个订单只能由一个社区服务运营人员派发。

(9) 在养老服务提供商和接单员实体之间,一家养老服务提供商可以拥有多个接单员,一个接单员只能属于一家养老服务提供商;一家养老服务提供商可以多次派单接单员,一个接单员只能接受来自一家养老服务提供商的派单。

(10) 在接单员和接单记录实体之间,一个接单员可以有多条接单记录,一条接单记录只能属于一个接单员。

(11) 在社区服务运营人员、志愿者和订单实体之间,一个社区服务运营人员为一个志愿者可以派发多个订单,一个社区服务运营人员只能派发一个订单给一个志愿者;一个养老服务提供商的某一个订单只能由一个社区服务运营人员派发。

(12) 在志愿者和志愿者服务记录实体之间,一个志愿者可以有多条志愿者服务记录,一条志愿者服务记录只能属于一个志愿者。

(13) 在志愿者和志愿者服务证书实体之间,一个志愿者可以有多张志愿者服务证书,一张志愿者服务证书只能属于一个志愿者。

2. 智慧社区养老服务系统数据库的逻辑设计

对于关系型数据库,逻辑结构设计的任务是将 E-R 模型转换为关系模式。

1) 实体转换

根据“每一个实体都将转换为一个关系模式”的原则,图 5-59 中的每个实体都将转换为关系模式,转换结果如下(说明:每个实体中更多属性以……替代)。

- 社区(社区编号,社区名称,所属行政区,街道……);
- 老年人(老年人身份证号,姓名,性别,出生年月……);
- 养老服务(养老服务编号,养老服务名称,养老服务类型……);
- 养老服务提供商(养老服务提供商编号,养老服务提供商名称,企业类型……);
- 社区服务运营人员(社区服务运营人员编号,姓名,性别,出生年月……);
- 志愿者(志愿者编号,姓名,性别,出生年月……);
- 智能终端(智能终端编号,型号,单价……);
- 接单员(接单员编号,姓名,性别,出生年月……);
- 养老券发放记录(记录编号,发放年度,发放月份,发放养老券数量……);
- 养老券使用记录(记录编号,使用年度,使用月份,使用养老券数量……);
- 养老券账户(养老券账户编号,养老券数量……);
- 订单(订单编号,下单时间,订单来源……);
- 接单记录(订单编号,接单员编号,接单时间……);
- 志愿者服务记录(订单编号,接单时间……);
- 志愿者服务证书(证书编号,证书等级……)。

2) 联系转换

在图 5-59 中,各个实体间由于养老服务的业务需求产生了各种联系。根据联系转换规则,对联系进行相应的转换。下面从图 5-59 中挑选几个不同的联系类型加以说明。

(1) 老年人和智能终端之间存在 1∶1 的联系,联系的转换选择把 1 端的“智能终端”的码

加入另一个一端的“老年人”中。即把“老年人”的关系模式修改为

老年人(老年人身份证号,姓名,性别,出生年月……**智能终端编号**)

(2) 老年人和养老券发放记录之间存在 1∶n 的联系,联系的转换选择把 1 端的“老年人”的码加入 n 端的“养老券发放记录”中。即把“养老券发放记录”关系模式修改为

养老券发放记录(记录编号,发放年度,发放月份,发放养老券数量……**老年人身份证号**)

(3) 社区和养老服务提供商之间存在 m∶n 的联系,联系需要转换为一个独立的关系模式。即添加一个关系模式为

签约记录(记录编号,**社区编号,养老服务提供商编号,签约时间**……)

(4) 社区服务运营人员、志愿者、订单之间存在 1∶1∶n 的联系,联系的转换选择转换为一个独立的关系模式,即添加一个关系模式为

派单志愿者记录(记录编号,**志愿者编号,社区服务运营人员编号,派单时间**……)

根据上述规则进行逐步转换后,得到的智慧社区养老服务系统数据库逻辑模型为下面的关系模式集合:

- 社区(社区编号,社区名称,所属行政区,街道……);
- 老年人(老年人身份证号,姓名,性别,出生年月……**社区编号**);
- 养老服务(养老服务编号,养老服务名称,养老服务类型……);
- 养老服务提供商(养老服务提供商编号,养老服务提供商名称,企业类型……);
- 社区服务运营人员(社区服务运营人员编号,姓名,性别,出生年月……);
- 志愿者(志愿者编号,姓名,性别,出生年月……);
- 智能终端(智能终端编号,型号,单价……);
- 接单员(接单员编号,姓名,性别,出生年月……**养老服务提供商编号**);
- 养老券发放记录(记录编号,发放年度,发放月份,发放养老券数量……**老年人身份证号**);
- 养老券使用记录(记录编号,使用年度,使用月份,使用养老券数量……**老年人身份证号**);
- 养老券账户(养老券账户编号,养老券数量,**老年人身份证号**);
- 订单(订单编号,下单时间,订单来源……**老年人身份证号,养老服务编号**);
- 接单记录(记录编号,接单时间……**接单员编号**);
- 志愿者服务记录(订单编号,接单时间……**志愿者编号**);
- 志愿者服务证书(证书编号,证书等级……**志愿者编号**);
- 派单接单员记录(派单编号,**订单编号,接单员编号,养老服务提供商编号,派单时间**……);
- 派单志愿者记录(派单编号,**订单编号,志愿者编号,社区服务运营人员编号,派单时间**……);
- 派单养老服务提供商记录(派单编号,**订单编号,养老服务提供商编号,社区服务运营人员编号,派单时间**……);
- 签约记录(记录编号,**社区编号,养老服务提供商编号,签约时间**……)。

深入思考 5.9 在关系模式派单记录中,语义解析为:一个社区服务运营人员在处理一个订单时,只能分配一个志愿者,那么该关系模式的码可以是订单编号吗?为什么要添加一个派单编号作为关系模式的码?

参考答案:参见微课视频 5-9。

5.7.4 案例的构件设计

构件设计需要对子系统分解得到的每一个业务能力逐个展开细化,获取实现业务能力所

需的设计类，将设计类填充到开发架构确立的各个包中，最终完成子系统的设计。

选取“订单服务子系统”中的“社区批处理派单”构件，介绍构件的细化过程。

1. 将分析类转换为设计类集合

1）明确待细化的分析类

在需求分析模型中，与“派单服务商”后端运行相关的分析类为：负责执行派单服务商的控制类 DispatchProviderControl；实体类包括养老服务订单、养老服务提供商、系统派单规则和人工派单规则。“订单服务子系统”的技术架构选择 Spring Boot 框架作为后端开发框架。明确了分析类和开发框架之后，分析类的细化将在技术框架的基础上展开。

2）将分析类转化为设计类

在设计阶段，分析阶段的控制类 DispatchProviderControl 被细化为两个业务能力：“社区批处理派单”和“社区人工派单”。按照 Spring Boot 框架的分层结构，将分析类中的控制类细化为分布在控制层、服务层和数据访问层中交互协作的设计类集合。下面只讨论“社区批处理派单”的控制类细化。

分析阶段的实体类包括养老服务订单、养老服务提供商、系统派单规则和人工派单规则。“养老服务订单”对于“订单服务子系统”是核心实体，它也是设计阶段的核心实体，需要保留；“养老服务提供商”的基础信息管理在子系统划分时，划归“社区养老综合服务子系统”，在此不作保留；系统派单规则和人工派单规则在设计阶段以派单服务中的算法形式呈现，在此也不作保留。因为每一次社区执行派单给服务提供商都将生成一条派单记录，在设计阶段需增加“派单养老服务提供商记录”实体。

细化后的设计类集合描述如下。

- 控制层设计 1 个控制类：BatchDispatchController（负责批处理派单业务的控制类）。
- 服务层设计 2 个服务接口：BatchDispatchService（负责执行批处理派单业务）、SerOrderService（负责执行各种养老服务订单操作）和 2 个对应的服务接口实现类：BatchDispatchServiceImpl（批处理派单业务实现类）、SerOrderServiceImpl（养老服务订单操作实现类）。
- 数据访问层设计 2 个数据访问接口：ServiceOrderDao（负责访问服务订单实体数据）、DispatchProviderDao（负责访问派单服务提供商记录数据）和 2 个对应的数据访问接口实现的 Mapper 映射文件。
- 数据层设计了 2 个实体：ServiceOrder（养老服务订单）、DispatchProvider（派单养老服务提供商记录）。

3）设计类图

按照框架间的层间调用关系，细化后的设计类分布在各个包中，如图 5-60 所示。

在图 5-60 中，空心三角虚线箭头表示接口与实现类之间的实现关系，普通虚线箭头表示调用关系。各层之间通过接口完成调用。

（1）控制层调用服务层。批处理派单控制类 BatchDispatchController 调用服务层的各服务接口 XXService，完成控制层与服务层之间的通信，实现对服务的调用。

（2）服务层调用数据访问层。服务实现类 XXServiceImpl 调用数据访问层的各数据访问接口 XXDao，完成服务层与数据访问层之间的通信，实现对各实体数据的访问。

2. 定义包和设计类

在开发架构已经确定的包框架下，向“订单服务子系统”包 com.oldman.area.order 的各内层包填充图 5-60 中描述的设计类，如表 5-10 所示。

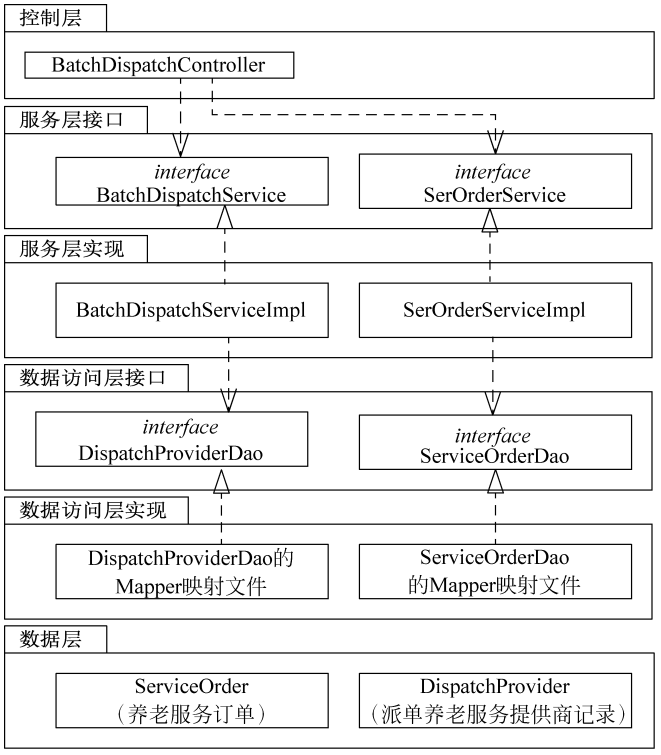


图 5-60 “社区批处理派单”类图

表 5-10 “社区批处理派单”服务包与类的设计

序号	包名	包中的类/接口	
		类/接口名	类/接口功能
1	controller	BatchDispatchController	批处理派单控制类
2	service	BatchDispatchService	批处理派单接口
		SerOrderService	养老服务订单管理接口
3	service/impl	BatchDispatchServiceImpl	批处理派单实现类
		SerOrderServiceImpl	养老服务订单管理实现类
4	dao	ServiceOrderDao	访问养老服务订单接口
		DispatchProviderDao	访问派单服务提供商记录数据
5	mapper	ServiceOrderDao. xml	访问服务订单实现
		DispatchProviderDao. xml	访问派单服务提供商记录数据实现
6	pojo	ServiceOrder	养老服务订单
		DispatchProvider	派单养老服务提供商记录

表 5-10 将分析阶段得到的分析类转化为设计阶段基于 Spring Boot 分层框架下的设计类，实现了从分析到设计的转换。

需要说明的是，软件设计各阶段都是一个迭代的过程。表 5-10 中包与类的设计只是一个初步的设计，在后期细化过程中，很有可能会不断进行更新。

3. 设计类之间的交互过程建模

以“社区批处理派单”构件的实现为例，绘制“社区批处理派单”服务的设计类交互序列图，

描述表 5-10 中设计类之间交互协作的过程，如图 5-61 所示。

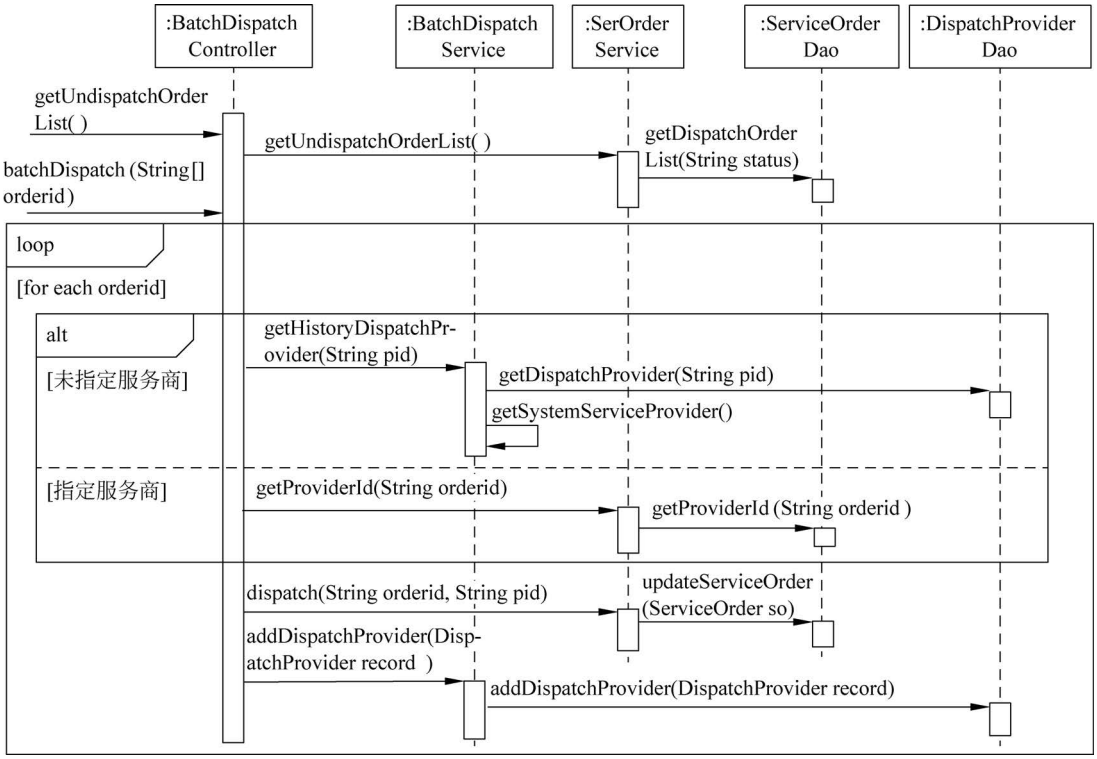


图 5-61　“社区批处理派单”服务的设计类交互序列图

图 5-61 所示的设计类交互序列图建模思路如下：观察图 5-57 所示的子系统交互图，由外部子系统“社区服务运营门户”发送给“订单服务子系统”的消息是控制层中的 BatchDispatchController 类应提供的访问接口，这部分设计已在接口设计中完成；由“订单服务子系统”发送给“DB”的消息和发送给自身的消息由服务层的 XXService 接口中的方法负责完成；由“订单服务子系统”发送给“DB”的消息又进一步被细化为服务层和数据访问层中的方法，XXService 发送给 XXDao 的消息由 XXDao 接口中的方法负责完成。

4. 定义设计类的属性和操作

1）服务层和数据访问层中的类

设计类中方法的定义来自交互过程中的消息传递，消息传递给谁，就由谁负责提供与消息对应的“方法”。从图 5-61 的描述中，得到“社区批处理派单”服务的设计类方法列表，如表 5-11 所示。

表 5-11　设计类描述示例

类名：SerOrderService			**所属包名：com. oldman. area. order. service**	
属性	编号	名称	数据类型	含义
	1	无		
方法	编号	名称		功能
	1	ServiceOrder[] getUndispatchOrderList( )		获取待派单养老服务订单
	2	String getProviderId(String orderid)		获取订单中指定的服务提供商编号
	3	int dispatch(String orderid,String pid)		派单给服务提供商

续表

类名：BatchDispatchService			所属包名：com.oldman.area.order.service	
属性	编号	名称	数据类型	含义
	1	无		
方法	编号	名称		功能
	1	DispatchProvider[] getHistoryDispatchProvider(String pid)		获取某养老服务提供商的社区派单历史数据
	2	ServiceProvider getSystemServiceProvider ()		根据历史派单信息及系统派单规则，设计派单算法，由系统自动生成指定养老服务提供商
	3	int addDispatchProvider(DispatchProvider record)		新增派单服务提供商记录

类名：ServiceOrderDao			所属包名：com.oldman.area.order.dao	
属性	编号	名称	数据类型	含义
	1	无		
方法	编号	名称		功能
	1	ServiceOrder[] getDispatchOrderList(String status)		读取订单状态为“未派单”的养老服务订单列表信息
	2	String getProviderId(String orderid)		根据订单编号获取指定的养老服务提供商编号
	3	int updateServiceOrder(ServiceOrder so)		更新服务订单

类名：DispatchProviderDao			所属包名：com.oldman.area.order.dao	
属性	编号	名称	数据类型	含义
	1	无		
方法	编号	名称		功能
	1	DispatchProvider[] getDispatchProvider(String pid)		根据养老服务提供商编号，查询其社区派单历史数据
	2	int addDispatchProvider(DispatchProvider record)		新增派单服务提供商记录

表 5-11 只列出了 service 包和 dao 包中接口的方法定义。service/impl 包中实现类的方法定义与 service 包中接口的方法定义是一致的，在此不再重复列出；mapper 包中存放的 xml 文件以 SQL 语句来实现 dao 包中接口的数据操作方法，在此也不作描述。

深入思考 5.10 根据图 5-57 的描述，考虑将“社区人工派单服务商”服务细化为设计类的集合，按照表 5-11 的模板完善实现“派单给服务商”用例的设计类。

参考答案：请参见微课视频 5-10。

2) 实体层中的类

在描述实体类的设计之前，首先来了解一下 ORM 技术。ORM(Object Relational Mapping，对象关系映射)是一种用来解决对象和关系型数据库表之间数据交互问题的技术。在传统开发过程中，把数据库表中的数据提取到对象中时，需要手动编写 SQL 语句。和自动生成 SQL 语句相比，手动编写 SQL 语句的缺点主要体现在：对象的属性名和数据表的字段

名往往不一致，在编写 SQL 语句时需要逐一核对属性名和字段名，确保它们不会出错，而且彼此之间要一一对应。ORM 为两者之间的数据映射提供了自动化的手段，解决了这个易出错的环节，同时也让源代码中不再出现 SQL 语句。

ORM 是一种双向数据交互技术，它不仅可以将对象中的数据存储到数据库中，也可以反过来将数据库中的数据提取到对象中。

（1）从对象到数据库。只要提前配置好对象和数据库之间的映射关系，ORM 就可以自动生成 SQL 语句，创建数据库的表结构并将对象中的数据自动存储到数据库中。

（2）从数据库到对象。只要提前做好数据库表的设计，就可以通过代码映射工具，将表自动映射为实体类以及对应的实体操作类，摆脱实体类及其通用数据操作的代码编写工作。例如：MyBatis-Plus 工具提供的方法能够将数据库表自动映射为 pojo 包中的数据实体类和 dao 包中数据访问类的基础操作方法。

本案例系统使用 ORM 技术，首先完成数据库表的设计，然后使用 MyBatis-Plus 工具自动生成相应的实体类和数据操作类，并在此基础上进行修改和细化。

5. 描述设计类中操作的处理逻辑

以 BatchDispatchController 派单控制类的 batchDispatch 批处理派单方法为例，使用程序流程图＋自然语言来描述其处理流程，如图 5-62 所示。

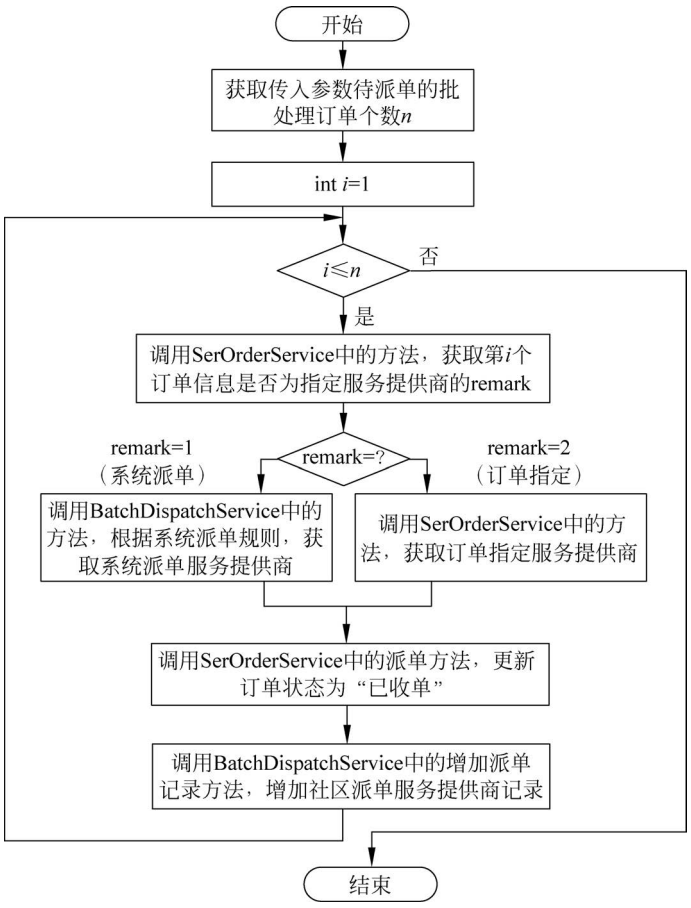


图 5-62　batchDispatch 方法的内部处理逻辑

在图 5-62 中，描述了控制层的批处理派单方法调用服务层各服务接口方法的过程，指明了方法实现时应遵循的控制流程。

习题

一、选择题

1. 系统软件设计阶段的任务是回答下列选项哪个问题？(　　)

A. 系统要做什么　　B. 系统要怎么做

C. 系统是什么　　D. 组织的价值

2. 下列不属于系统设计阶段内容的是(　　)。

A. 功能需求　　B. 体系结构设计

C. 接口设计　　D. 构件设计

3. 结构化体系结构设计常见的表达工具是(　　)。

A. 类图　　B. 系统结构图

C. 数据流图　　D. E-R 图

4. 下列哪一项不属于结构化概要设计的内容？(　　)

A. 体系结构设计　　B. 数据库设计

C. 接口设计　　D. 算法设计

5. 在解决复杂问题时，先将复杂问题做合理的分解，再分别仔细研究问题的不同侧面。这种系统思维方法被称为(　　)。

A. 自下而上　　B. 关注点分离

C. 复用性　　D. 信息隐蔽

6. “逐步求精”是下列哪一项设计策略？(　　)

A. 自下而上　　B. 由外至内　　C. 自顶向下　　D. 由内而外

7. 下列哪一项被称为系统的蓝图？(　　)

A. 用例图　　B. 数据结构　　C. 构件图　　D. 体系结构

8. 下列不属于调用/返回风格的体系结构是(　　)。

A. 主程序/子程序　　B. 面对对象风格

C. 层次风格　　D. 解释器

9. 下列哪一项描述了一个在某种上下文环境中不断出现的问题，以及该问题的解决方案？(　　)

A. 模式　　B. 体系结构　　C. 框架　　D. 构件

10. 下列哪一项模式是系统的高层次策略？(　　)

A. 数据模式　　B. 体系结构模式

C. 设计模式　　D. 代码模式

11. 下列哪一个概念是软件？(　　)

A. 模式　　B. 框架　　C. 架构　　D. 体系结构

12. AOP 中关于横切关注点，下列哪一项说法是不正确的？(　　)

A. 它是业务逻辑　　B. 是对公用功能的提取

C. 能够降低模块间耦合度　　D. 便于减少重复代码

13. 模块 A 在调用模块 B 时只需要用到学号，但是在参数传递时，却传递了包含学号、姓名、性别、籍贯等多个数据信息的学生数据结构，此时发生的耦合称为(　　)。

A. 非直接耦合　　B. 数据耦合

C. 标记耦合　　D. 控制耦合

14. 模块 A 向模块 B 传递一个控制类型的数据信息来控制模块 B，此时发生的耦合称为（　　）。

A. 非直接耦合　　B. 数据耦合

C. 标记耦合　　D. 控制耦合

15. 如果一个模块中各部分处理元素都是围绕同一项功能而协同工作的，紧密联系，不可分割，则称为（　　）。

A. 功能内聚　　B. 逻辑内聚　　C. 顺序内聚　　D. 通信内聚

16. 软件结构图中，模块结构的控制层数称为结构图的（　　）。

A. 宽度　　B. 扇出　　C. 深度　　D. 扇入

17. 在结构化设计阶段，可以基于下列哪一项将其映射变换成软件结构图？（　　）

A. 数据流图　　B. 状态图　　C. 用例图　　D. 序列图

18. 下列哪一项不属于 RUP“4＋1 视图”中 4 个视图中的一项？（　　）

A. 逻辑架构视图　　B. 开发架构视图

C. 用例视图　　D. 物理架构视图

19. 高校毕业管理系统在毕业生办理毕业手续时，需要调用财务系统的学费缴纳数据，此时的接口属于下列哪一项？（　　）

A. 用户接口　　B. 外部接口　　C. 内部接口　　D. 不确定

20. 下列哪一项不属于用户界面设计的内容？（　　）

A. 数据输入界面设计　　B. 数据输出界面设计

C. 外部设备交互设计　　D. 控制界面设计

21. 若一个志愿者为一位老人能提供多项养老服务；一个志愿者的一项养老服务能提供给多位老人；一个老人得到的一项养老服务只能由一位志愿者提供，则称志愿者、老人、养老服务三个实体之间是下列哪一项三元联系？（　　）

A. 1∶1∶1　　B. 1∶1∶m　　C. 1∶m∶n　　D. m∶n∶p

22. 下列哪一项不属于描述算法的常用工具？（　　）

A. 程序流程图　　B. 流图　　C. PAD 图　　D. 盒图

二、判断题

1. 软件设计在软件需求和软件实现之间起到了桥梁作用。（　　）
2. 模块功能独立是“关注点分离”等概念的直接产物。（　　）
3. 层次风格系统中的每一层为下层提供一组服务。（　　）
4. 视图负责控制器和模型之间的信息交互。（　　）
5. 一个体系结构模式通常对应一个设计模式。（　　）
6. 体系结构风格涉及的范围比体系结构模式要小一些。（　　）
7. 体系结构包含了具体的类和对象。（　　）
8. 将系统划分为模块时，模块分解得越细致越好。（　　）
9. 内聚是对模块间关联程度的度量。（　　）
10. 内容耦合属于强耦合。（　　）
11. 逻辑内聚模块是内聚程度最低的模块。（　　）
12. 扇入数越大表示该模块被越多的上级模块共享，所以扇入数越大越好。（　　）
13. 事务型数据流图整体结构可划分为三部分：输入、变换中心和输出。（　　）

14. 系统内部的接口是指方法与方法之间,模块与模块之间的交互。 ( )

15. 接口设计应描述涉及的模块或子系统内部的实现细节。 ( )

16. 在数据库概念模型设计中,属性必须是不可再分的数据项。 ( )

17. 任何程序逻辑都可用顺序、分支和循环三种基本结构来表示。 ( )

18. 伪代码是一种形式化的编程语言。 ( )

19. 通常只需要一个视图就可以呈现软件的体系结构。 ( )

20. 构件的概念主要存在于业务架构中。 ( )

三、综合题

1. 某商业集团数据库中有 4 个实体集。一是“商店”实体集,属性有商店编号、商店名、地址;二是“商品”实体集,属性有商品号、商品名、规格、单价;三是“职工”实体集,有职工编号、姓名、性别、业绩;四是“供应商”实体集,属性有供应商编号、供应商名、地址。

语义如下:每个商店可销售多种商品,每种商品也可放在多个商店销售,每个商店每销售一种商品就有月销售量;每个商店有许多职工,每个职工只能在一个商店工作,商店聘用职工有聘期和月薪;每个供应商可供应多种商品,每种商品可向多个供应商订购,供应商供应每种商品有月供应量。

根据上述描述回答以下问题:

(1) 画出该商业集团数据库的 E-R 图,并在图上注明属性、联系的类型。

(2) 将 E-R 图转换成关系模式集,写出转换后的关系模式。

2. 如果要求一个整数数组中各个元素的平均值,请用程序流程图、N-S 图和 PAD 图分别表示出求解该问题的算法。