第 3 章

面向对象设计

软件系统离不开硬件系统，一个基于计算机的系统离不开使用该系统的客户和整个社会。所有这些元素：硬件设备、操作系统、网络、应用系统、业务过程、各种机构和社会形成了复杂的“社会技术系统”，如图 3-1 所示。这个系统的建立、维护是一个系统工程问题。软件工程只是整个社会技术系统中的一部分。

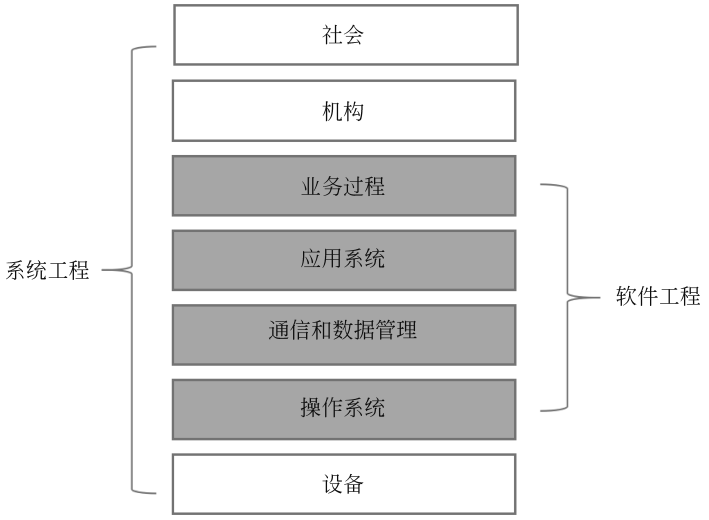

图 3-1　社会技术系统（萨默维尔，《软件工程》（第 9 版））

处于社会技术系统中的软件设计应当展示坚固性（Firmness）、适用性（Commodity）和愉悦（Delight），以满足“社会性”要求。坚固性指程序不应含有缺陷；适用性指程序应当满足其愿景；愉悦指用户体验应当是愉悦的。

软件工程中的设计包括类/数据设计、体系结构设计、接口设计和组件设计。类设计是对分析模型中的类模型进行补充和修订；体系结构设计定义软件主要构造元素之间的关系；接口设计描述软件与协作系统之间、软件与用户之间如何进行交互；组件级设计对软件组件进行过程性描述。所以，设计过程就是把分析模型映射到设计模型的过程，如图 3-2 所示。

分析模型中包括从功能、结构和行为三个方面描述的系统逻辑模型，如果有必要，还可以包括面向数据流的模型。分析模型是软件开发者与客户沟通的结果，也是软件设计者的出发点，所以分析模型是需求与设计之间的桥梁。由于概念的一致性，面向对象的分

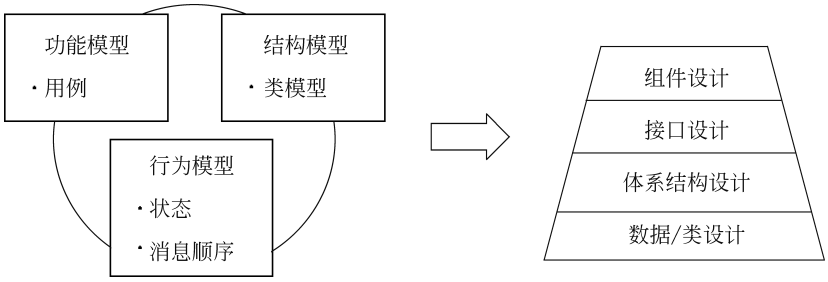

图 3-2　从分析模型到设计模型的映射

析与设计是一个无缝衔接和过渡的过程。

软件体系结构(Architecture)也称为架构、构架,是计算组件以及组件之间的交互。这里的组件可以是子系统、框架等。

一个复杂的软件系统,往往分解成若干个子系统,以完成相对独立的功能。多个子系统相互配合满足整个系统的需求。一个大型企业往往部署了多套系统,通过系统间的互操作,把这些业务系统集成起来,就形成了“集成系统”。子系统的基本组成单元是类,一组相关的类通常被组织在类库中。

框架(Framework)是一个半成品,是一个可实例化的、部分完成的软件子系统,为构造完成的系统提供了基础设施。在面向对象的环境中,框架由接口、抽象类和实现类组成。所以,从设计角度看,框架是一组相互协作的类,是形成某类软件的可复用的设计。开发者通过继承框架类中的类和组合其实例来定制该框架以得到特定的应用。框架也是按照一定的架构开发的。

类库是类的集合,有些类之间可能是相互独立的。框架中的类并不是孤立的,存在协作关系。

软件架构设计的内容包括:规划目标系统的子系统,为子系统分配不同的职责,并使这些子系统通过协作完成功能需求;深入研究预测运行期间系统应满足的质量需求,如响应时间、吞吐量、并发、负载、可用性等运行时刻质量属性,制定相应的设计决策;为满足目标系统可测试性、可维护性、可扩展性、可复用性等开发时期质量属性,做出相应设计决策。约束也是一类特殊的需求,带有一定的强制性,架构的设计决策应满足这些限制。

针对不同的需求类型,产生不同的架构视图,如图 3-3 所示。

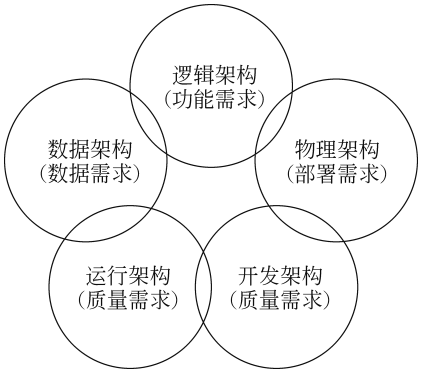

图 3-3　架构视图

逻辑架构关注于业务功能需求,即目标系统的行为以及目标系统行为的分解。逻辑架构关心的是如何将系统分解为弱耦合的不同部分以及各部分之间如何交互,也就是规定软件系统由哪些宏观逻辑元素组成以及这些逻辑元素之间的关系。逻辑元素有层、子系统、模块等。关系包括交互接口和交互机制等,

如使用 JDBC 中间件连接数据库服务器、使用消息中间件 Kafka 等。

数据架构关注于持久化数据的组织、传递、复制和同步等策略。侧重于从业务数据流的角度描述本系统数据与上下游系统数据之间的关系，以及针对本系统承载的业务处理，设计了哪些与关键实体对象对应的实体数据表。要明确本系统处理的数据在整个业务数据流链条上的位置，说明数据初始化方式、数据冗余策略、分库分表方法和数据库备份方案等。

物理架构则关注于将通过编译的目标系统安装和部署到服务器、网络、嵌入式设备等硬件设备上，建立软件单元和硬件单元的映射。物理架构关注系统分为几部分、各个部分如何进行物理部署；部署系统各个部分之间的各服务器的协作关系；明确硬件服务器的型号、数量等配置，如多核数目、内存容量、硬盘热插拔、网络端口数及网络带宽要求，软件方面对操作系统的类型、版本要求，服务器软件版本要求、参数的调优设置，各个软件之间的协同配置等；明确整体部署的网络区域，如外联区、DMZ 区或内网区等。物理架构还关注可用性、可伸缩性、安全性等质量因素，需要描述清楚通过怎样的集群或热备部署保证可用性、系统做了何种设计或优化以满足伸缩性要求、设计了何种校验机制保障安全。每类非功能性应能够追溯到需求，与业务实际相匹配。

开发架构关注于软件的可扩展性、可移植性、可复用性、可测试性等质量属性，从技术的角度描述本系统在实现过程中用到的关键技术、核心技术组件，包括成熟商业套件以及开源技术产品。架构中的元素包括源文件、配置文件、包、第三方类库等。在程序员看来，开发架构就是基于什么框架。可复用性是技术架构的关键，无论是历史遗留组件还是开源框架，都是复用。要说明类库的版本、功能、适用场景。如果是商业套件，需要说明使用限制、升级支持等；如果是开源框架，需要说明开源协议要求。通过描述所有与外部系统的接口定义系统与外部系统之间的外部接口关系。详细说明每一个外部接口的名称（如××消息推送接口）、所交互的系统名称（如一卡通系统向 kafka 推送消息）、交互方式（如 Web Service）、交互风格（如 RESTful 风格）、通信方式（如异步）、接口描述（如一卡通系统通过此接口从 kafka 中获取××业务数据）。

运行架构则关注易用性、响应时间、负载等质量属性。

软件架构设计从大局着手，就技术方面重大问题做出决策，构造了一个具有一定抽象层次的解决方案，而没有深入到细节，从而控制了“技术复杂性”。有了架构设计，不同的开发小组就可以在不同的子系统上按照子系统间的契约并行开发，形成了大规模开发的基础。负责界面开发的团队只需研究用户界面工具包，而不必关心负责数据库访问的 SQL 语句如何设计；数据库开发人员不再关心界面如何设计，而只需按照约定的应用程序接口设计程序。

3.1 软件体系结构风格

风格是事物的形态特征，是文化、思想的表征。软件体系结构风格定义了组件和组件连接的策略。针对复杂系统的体系结构，先后形成了 Layer 和 Tier 两种基本风格。Layer 风格面向同族系统，形成紧密的垂直访问体系；Tier 风格面向异族系统，形成松散

的水平协作体系。另外，还有面向服务的体系结构风格等。

3.1.1　Layer 风格

Layer 风格是面向同族系统的一种软件体系结构风格。所谓同族系统，指内部各个层次之间的关系对外部系统来说是透明的。外部系统只能与该系统的顶层交互。Layer 风格也可称为垂直型层次风格。

垂直型分层结构是一种广泛应用的软件体系结构风格。各个子系统按照层次的形式组织，上层访问下层的各种服务，而下层对上层一无所知。每一层都对自己的上层隐藏其下层的细节。一般是下层为上层提供服务，而且一般不会跨层服务。操作系统的设计就采用了分层架构，如图 3-4 所示。其中定义了一系列不同的层次，每个层次各自完成操作，这些操作逐步接近机器的指令集。在最外层，图形用户界面、命令行窗口等程序完成用户界面的操作，与用户交互；浏览器、字处理软件、图像处理软件等应用软件形成应用程序层；用户通过界面访问应用程序。应用程序访问通过操作系统提供的预定应用程序接口（API）完成存盘、网络传输等功能；而这些 API 则通过核心代码与硬件交互。

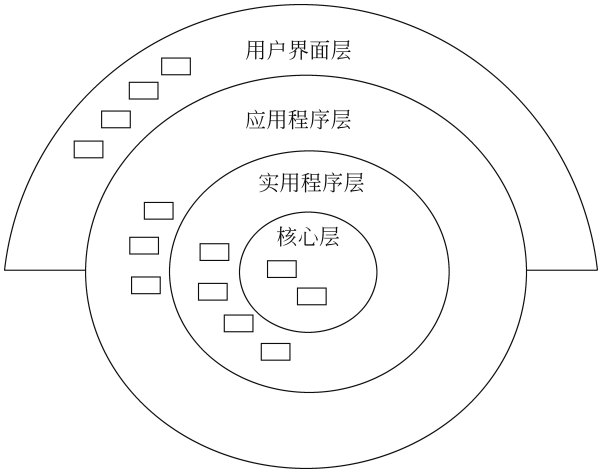

图 3-4　操作体系的层次体系结构

TCP/IP 协议簇是互联网中基本的协议，设计为一个四层的体系结构：应用层、传输层、网络层和数据链路层。应用层的主要协议有远程登录（Telnet）、文件传输协议（FTP）、简单邮件传输协议（SMTP）等，是用来接收来自传输层的数据或者按不同应用要求与方式将数据传输至传输层；传输层的主要协议有传输控制协议（TCP）、用户数据报协议（UDP），可以实现面向连接的或无连接的端到端数据传输；网络层的主要协议有网际报文控制协议（ICMP）、网际协议（IP）、互联网组管理协议（IGMP），主要负责网络中数据包的传输路径选择等；而网络访问层，也叫网络接口层或数据链路层，主要功能是提供点到点的链路管理等。例如，地址解析协议（ARP）、反向地址解析协议（RARP）用来管理 IP 地址和物理地址映射。图 3-5 以应用层协议（FTP）为例，展示了这四个层次之间的关系：FTP 客户端将数据打包成报文交给 TCP；TCP 建立目标进程的连接，将报文拆分成

“包”交给 IP 存储转发;IP 层再根据数据链路层协议将包封装到数据帧中传递给下个节点。目标计算机接收数据帧,恢复成 IP 包再交给传输层。等所有包到齐后传输层恢复给 FTP 完整的报文。

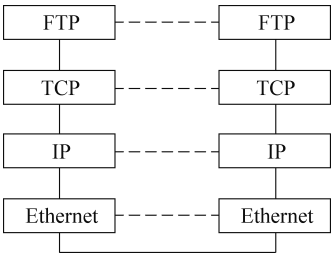

图 3-5　TCP/IP 层次

3.1.2　Tier 风格

Tier 风格是面向异族系统的一种体系结构风格。异族系统指系统中各个子系统之间的关系对外部系统来说是不透明的,外部系统可以与任何子系统交互。Tier 风格一般采用水平型层次以体现这种松耦合的特征。Tier 风格的案例包括客户机/服务器、三层体系结构风格、浏览器/服务器等。

1. 客户机/服务器

客户机/服务器体系结构风格中,整个软件系统分解成两种角色:客户机和服务器。客户机和服务器一般是多对一的关系,即 N 个客户机访问 1 个服务器。用户仅与客户机交互,由客户机向服务器发出访问请求,服务器响应客户机请求完成相应功能。

基于局域网的管理信息系统广泛采用客户机/服务器体系结构,例如,会计核算系统、工资管理系统。一般来说,服务器上部署的一个数据库管理系统,称为数据库服务器。业务处理程序部署在桌面计算机上,作为客户机;若干客户机通过局域网与数据库服务器连接。客户机所有访问数据库的需求都通过服务器完成:客户机发出请求(SQL 语句),服务器接收请求并执行,把执行结果返回给客户机。客户机/服务器架构如图 3-6 所示。

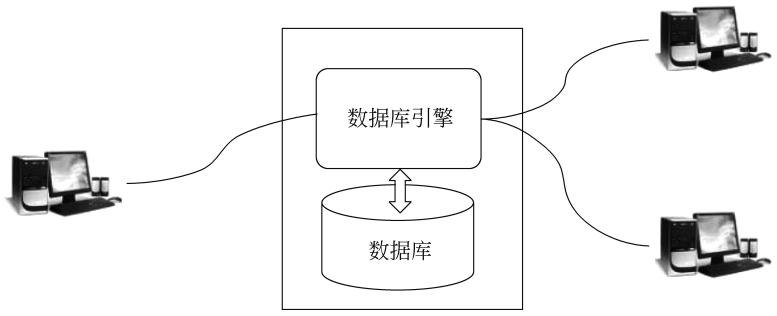

图 3-6　客户机/服务器架构

客户机和服务器实际上是角色的名称。服务器可能指具有接收和处理请求能力的计算机硬件,也可能指进程。客户机也未必一定是一台计算机,也可能是一部手机、一个嵌

入式设备，或者一个进程。为了一般起见，后面不再使用“客户机”，而使用“客户端”或者简单将其简称为“客户”。最简单、最常见的客户/服务器体系结构有多个客户端和一个服务器，如图 3-7 所示，称为多客户端-单服务器架构。

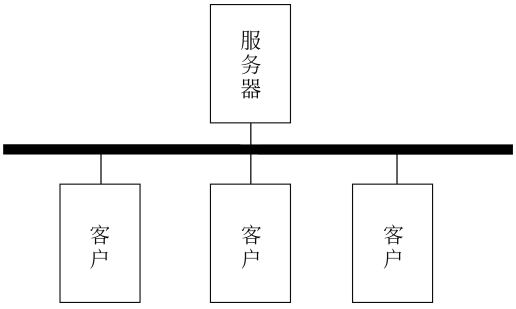

图 3-7　多客户端-单服务器架构

在 TCP/IP 的应用层协议中，FTP、SMTP、HTTP 等都是多客户端-单服务器体系结构风格。

客户/服务器体系结构中允许有多个服务器，每个客户端允许与多个服务器通信，同时服务器之间也允许通信，形成“多客户端-多服务器”架构，如图 3-8 所示。

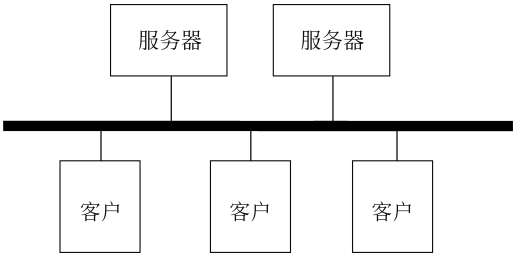

图 3-8　多客户端-多服务器架构

客户端与服务器之间的通信方式有：同步消息通信、异步消息通信、中介模式等。

“请求/响应”通信方式是一种典型的同步消息通信：客户端向服务器发送消息并等待回复。单个客户端和单个服务器之间不需要消息队列，在多客户端和单服务器情形中，服务器端会有一个消息队列。

在异步消息通信模式中，客户端向服务器发送消息后并不等待回复，继续做自己的事情；如果消息到达服务器而服务器正忙，则消息进入等待队列。

如果客户/服务器体系结构中，客户能够承担服务器角色，服务器也能承担客户角色，则形成了对等体系结构风格。

在多对多的客户/服务器体系结构中，有很多服务器提供多种服务，客户可能知道使用哪个服务但是不知道需要的服务在哪里，中介模式即可解决这类问题：客户端首先向中介发送服务请求；中介查询服务的位置并将请求转发；服务器处理请求并将结果回复给中介；中介再把回复转发给客户。

如果客户端知道请求的服务类型而不是特定的服务实例，那么可以使用服务发现通

信模式。在这种模式中，客户端发送一个查询请求给中介，请求给定类型的所有服务，中介回复一个服务清单；客户端从中选择一个服务，然后与服务器直接通信。

2. 浏览器/服务器

当 Web 中的服务器不仅能够响应浏览器发出的页面、图片、文件、视频、音频等静态资源请求时，还能够响应浏览器对服务器端脚本的请求时，那么在这种服务器上就能够部署完成应用功能的代码，形成基于 Web 的应用。这种应用的特点是：浏览器是统一的客户端，对业务逻辑的处理主要集中在服务器上，只需部署一次，如图 3-9 所示。

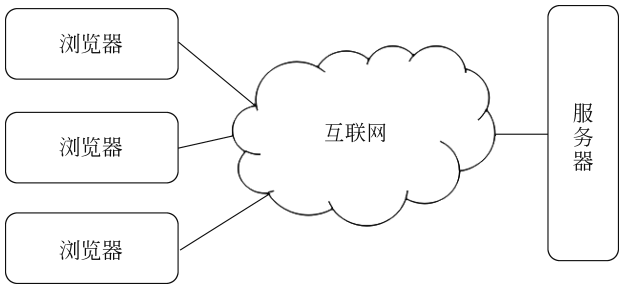

图 3-9　浏览器/服务器架构

浏览器也是一种客户端。由于客户机/服务器体系中的客户端既负责用户界面，又负责应用逻辑，承担较多的责任，所以一般把将客户机/服务器体系中的客户端称为“胖客户端”；而浏览器只负责人机界面，所以称为“瘦客户端”。

3. 三层体系结构风格

三层(3－tier)体系结构中有三个子系统：用户接口层(User Interface Layer，UI)，业务逻辑层(Business Logic Layer，BLL)和数据访问层(Data Access Layer，DAL)。用户接口层也称为表现层。这种划分体现了“高内聚，低耦合”的思想。用户接口层负责与用户交互，包括表单、脚本、样式等元素；业务逻辑层包含实体对象和控制逻辑，完成业务功能；数据访问层实现对象的持久化和查询。该层直接操作数据库，对数据增添、删除、修改、更新、查找等，如图 3-10 所示。

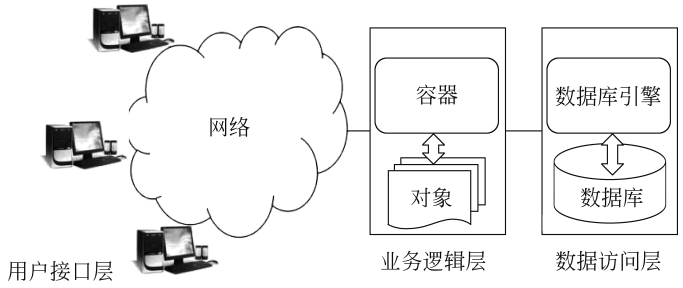

图 3-10　三层体系结构

注意这里的层是 tier 而不是 layer，tier 指的是子系统之间的独立性，每个子系统都是可替换的。每一层之间通过变量或对象作为参数传递数据，这样就构造了三层之间的联系，完成了功能的实现。

图 3-11 展示了现实生活中的三层架构到软件体系结构的映射。在饭店就餐服务业务中，服务员只负责接待客人；厨师只负责烹饪客人点的菜；采购员只管采购食材。顾客直接和服务员(UI)打交道，顾客和服务员说：我要一个醋溜土豆丝。服务员就把请求传递给厨师(BLL)，厨师需要土豆，就把请求传递给采购员(DAL)，采购员从仓库里取来土豆传回给厨师，厨师切丝炒菜装盘后，传回给服务员，服务员把菜品上桌呈现给顾客。在此过程中，土豆作为参数在三层中传递。这个模型的最大好处就是各层之间耦合度很低，每一层的变化不影响其他层以及整个业务。例如，更换服务员不会影响厨师。

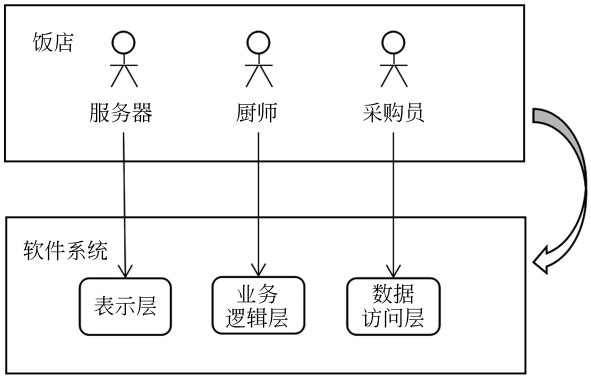

图 3-11　现实世界中的三层架构

再例如，天气预报的手机 APP 首先通过数据访问层得到温度、空气污染指数(API)等数据；业务逻辑层按照一定的逻辑进行处理，如把 API 值映射到“优”“良”“轻度”等状态；用户接口层则以图形、图标、颜色等形式显现给用户，如图 3-12 所示。

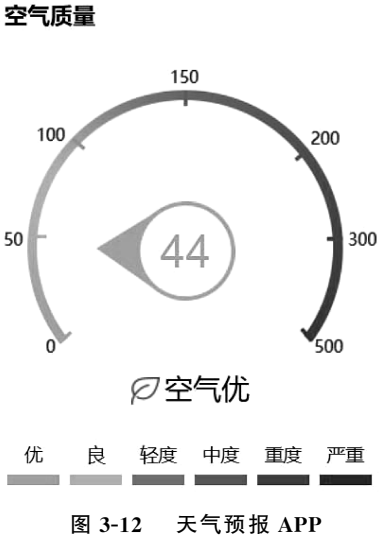

图 3-12　天气预报 APP

三层体系结构必须通过中间层完成，有时会导致级联的修改，对系统的性能有负面影响。如果在用户接口层中需要增加一个功能，为保证其设计符合分层式结构，可能需要在相应的业务逻辑层和数据访问层中都增加相应的代码，从而也会影响可维护性。

4. 四层体系结构

四层体系结构由表示层、请求处理层、业务逻辑层和数据访问层组成，如图 3-13 所示。

表示层是负责终端上的界面渲染并显示的层，当前主要技术包括 Java 脚本、HTML、样式表等；请求处理层(Web 层)主要负责对访问请求进行转发，对各类基本参数校验，或者对业务进行简单处理等；业务逻辑层(Service 层)是相对具体的业务逻辑服务层；数据访问层(DAO 层)与 MySQL、Oracle、HBase 等进行数据交互。

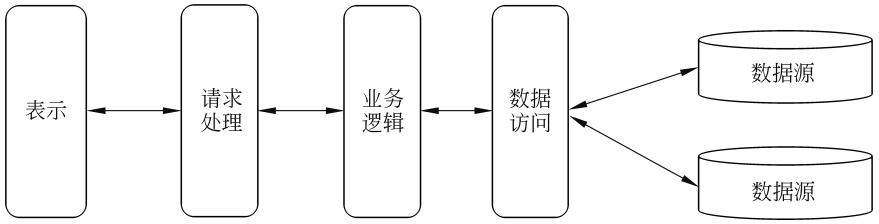

图 3-13　四层体系结构

这种分层的结构很容易扩展，如增加开放接口层，把 Service 方法封装成远程过程调用(RPC)接口或者封装成 Web 服务接口等，就可以让其他系统直接使用业务逻辑层的功能和接口。若要访问外部接口或第三方平台包括其他部门 RPC 开放接口、基础平台、其他公司的 Web 服务接口，则可以引入一个通用处理层(Manager 层)对第三方平台封装，处理返回结果及转换异常信息，通过缓存、中间件增加对 Service 层的通用能力。这样，通用处理层向业务逻辑层提供了服务，也可以与 DAO 层交互，使用 DAO 层的服务。

3.1.3　模型-视图-控制器

模型-视图-控制器(Model-View-Controller，MVC)是一种体系结构风格，如图 3-14 所示。视图是用户看到并与之交互的界面。对 Web 应用程序来说，视图就是由 HTML 元素组成的界面。当用户单击 Web 页面中的超链接或者提交 HTML 表单时，控制器负责接收请求，但本身不输出任何东西和做任何处理，它只是接收请求并决定调用哪个模型处理请求，然后再确定用哪个视图显示返回的数据。所以，在 MVC 中，模型提供了要展示的数据，以及访问数据的行为，是领域模型。也就是模型提供了模型数据查询和模型数据的状态更新等功能。视图负责进行模型的展示，一般就是用户界面。控制器负责接收

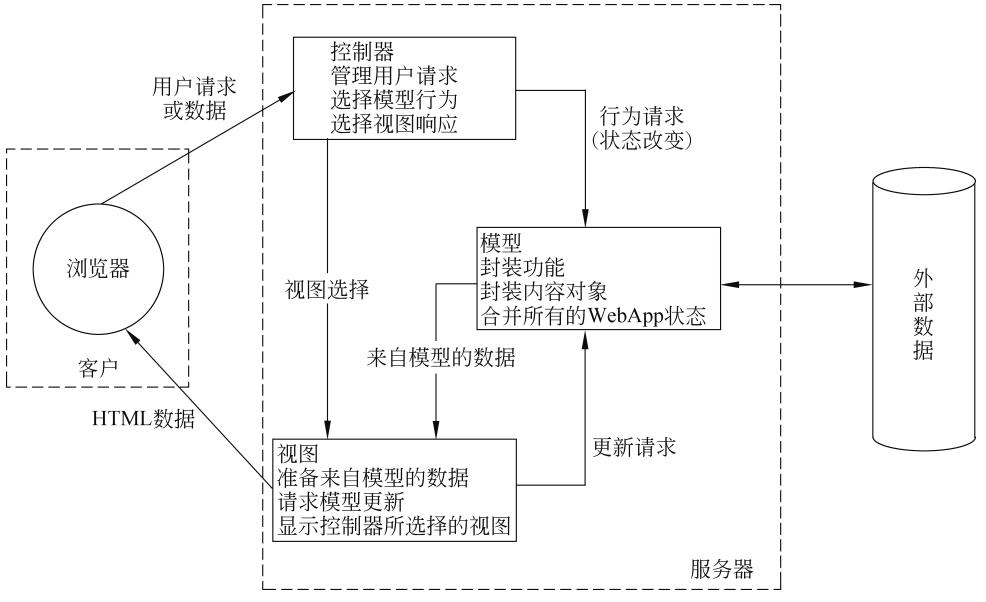

图 3-14　Web 应用中的 MVC 风格

用户请求，委托给模型进行处理，处理完毕后把返回的模型数据返回给视图，由视图负责展示。也就是说，控制器承担调度员角色。

下面使用一个简单的例子进行说明。假设有如下 JSP 界面。

```
<html>
<head>
    <title>Menu</title>
</head>
<body>
    <a href = "$ {pageContext.request.contextPath}/AddServlet">列表</a><br>
</body>
</html>
```

这个页面就是一个视图。当用户在 Web 页面上单击“列表”超级链接时，这个请求就发送给了控制器 ListServlet，这个 Servlet 控制器定义如下。

```
@WebServlet("/ListServlet")
public class ListServlet extends HttpServlet {
    //…
    protected void doGet(HttpServletRequest request
    , HttpServletResponse response) throws ServletException, IOException {
        request.getRequestDispatcher("/WEB-INF/jsp/list.jsp")
                .forward(request, response);
    }
    //…
}
```

该控制器是通过继承 HttpServlet 实现的，其中的 doGet()方法把请求转发给了 list.jsp，这就是“选择视图响应”。list.jsp 页面通过 DbAccess 访问数据库，向模型发出“更新请求”，模型访问数据库，返回结果是实体 Professor 类的对象，这些对象由容器 professors 管理。JSP 把“来自模型的数据”遍历后，逐个在浏览器中显示所有对象，即把“HTML 数据”发送给浏览器渲染。

```
<html>
<head>
    <meta http-equiv= "Content-Type" content= "text/html; charset= UTF-8">
    <title>列表</title>
</head>
<body>
<h1>Professors</h1>
<%
    for (Professor p : DbAccess.professors) {
        out.println(p);
        out.write("<br>");
    }
```

```
% >
<br>
</html>
```

3.1.4 面向服务的架构

面向服务的架构(Service-Oriented Architecture,SOA)是由若干自治的服务组成的分布式软件体系结构。各个服务可以运行在不同平台上,可以用不同的语言实现。通过标准的协议注册、发现和协调服务。SOA 是一种粗粒度、松耦合的服务架构,服务之间通过简单、精确定义的接口进行通信,不涉及底层编程接口和通信模型。

尽管面向服务的架构在概念上与平台无关,但是 Web 服务的技术是面向服务架构的成功实现。从客户角度看,Web 服务是部署在 Web 上的对象,具备完好的封装性、松散耦合、自包含、互操作、动态、独立于实现技术、可集成、使用标准协议等特征。从实施角度看,Web 服务把资源、计算能力提供给用户,需要以服务的形式完成,Web 服务是 Web 上可寻址的应用程序接口。如果把 Web 服务看作类库,那么类库不是位于本地的,而是位于远程机器上。从设计角度看,Web 服务通过 WSDL 描述,通过 SOAP 访问,在注册中心发布,从而使客户可以搜索并定位到该服务。在 Web 服务体系结构中有三个角色:服务提供者(Service Provider),服务请求者(Service Requester)和服务中介(Service Broker),图 3-15 显示了它们之间的关系。

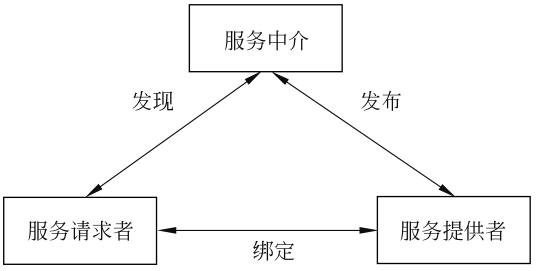

图 3-15 Web 服务体系结构

Web 服务提供者创建服务,发布 Web 服务,并且对服务请求进行响应。Web 服务请求者也就是 Web 服务功能的消费者,它通过 Web 服务中介查找到所需要的服务,再向 Web 服务提供者发送请求以获得服务。Web 服务中介,也称为服务代理,用来注册已经发布的 Web 服务,并对其进行分类,提供搜索服务,把一个 Web 服务请求者和合适的 Web 服务提供者进行匹配。

在这个架构中有三种操作:发布、发现和绑定。通过发布操作,可以使 Web 服务提供者向 Web 服务中介注册自己的功能以及访问的接口。发现(查找)使得 Web 服务请求者可以通过 Web 服务中介查找所需的 Web 服务。绑定就是实现让服务请求者能够使用服务提供者提供的服务了。

Web 服务架构中用到的协议、语言和规范有 UDDI、WSDL、SOAP、XML、HTTP 等。

UDDI 是统一描述、发现和集成(Universal Description, Discovery, and Integration)的缩写。它是一个基于 XML 的跨平台的描述规范,可以使世界范围内的企业在互联网上发布自己所提供的服务。WSDL(Web Services Description Language,Web 服务描述语言)是一个基于 XML 的关于如何与 Web 服务通信和使用的服务描述语言。简单对象访问协议(SOAP)是一种轻量的、简单的、基于 XML 的协议,可以和现存的许多因特网协议结合使用,包括超文本传输协议(HTTP)、简单邮件传输协议(SMTP)、多用途网际邮件扩充协议(MIME)等。

例如,国家气象局可以实时以 Web 服务的形式公开发布全国各地的天气状况,各种应用系统、手机 APP 便可以通过这个 Web 服务来访问到天气状况。

3.1.5　微服务架构

微服务提倡将单一应用程序划分成更小的服务,每个服务运行在独立的自己的进程中,服务之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,并且能够被独立地部署,如图 3-16 所示。

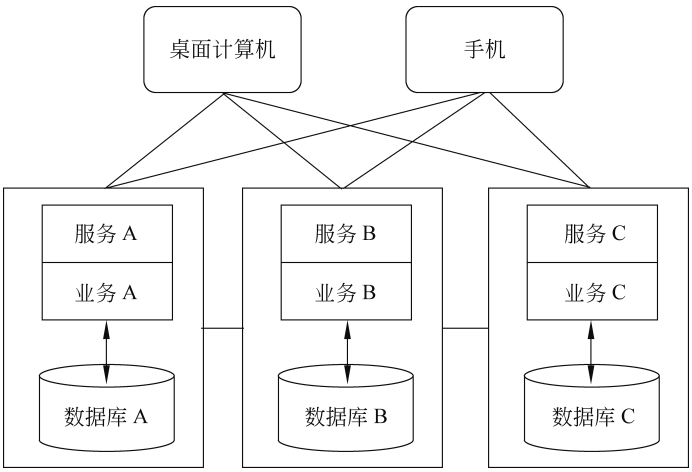

图 3-16　微服务架构

在单体式(Monolithic)的架构中,如三层应用,所有的功能打包在一个 WAR 包里,部署在一个 Java EE 容器(Tomcat、JBoss、WebLogic 等)里,包含数据访问、业务处理和用户界面等所有程序,除了容器基本没有外部依赖。其优点是单点部署,没有分布式的管理和调用代价。但缺点是扩展性不够,无法满足高并发下的业务变更需求。例如,数据库模式被多个应用依赖,无法重构和优化。所有应用都在一个数据库上操作,数据库出现性能瓶颈。开发、测试、部署、维护愈发困难。即使只改动一个小功能,也需要整个应用一起构建和发布。基于微服务架构的设计目的是有效地拆分应用,实现敏捷开发和部署。

3.2 组消息通信模式

分布式系统中相互作用的进程集合称为组。一个发送进程在一次操作中将一个消息发送给组内其他进程的通信,称为组消息通信。组内每个成员都是平等的。进程可以加入或离开组。

3.2.1 消息队列

在消息队列模型中有两种角色:消息的生产者(Producer)和消费者(Consumer)。消息生产者生产消息发送到消息队列中,然后消费者从消息队列中取出并且消费消息。消息从队列中取出被消费以后,队列中不再有存储,所以这个模型中虽然存在多个消费者,但是对一个消息而言,只会有一个消费者可以消费,如图 3-17 所示。

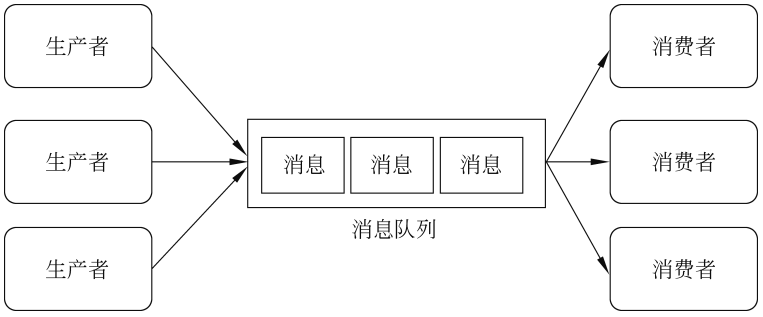

图 3-17 消息队列模型

3.2.2 发布/订阅

发布/订阅模型中有三个角色:消息发布者(Publisher)、中介代理(Broker)、消息订阅者(Subscriber)。发布者发布消息到一个消息的中介代理,对消息感兴趣的订阅者向该代理注册和订阅,由消息代理进行过滤。消息代理执行存储转发的功能将消息从发布者发送到订阅者,如图 3-18 所示。

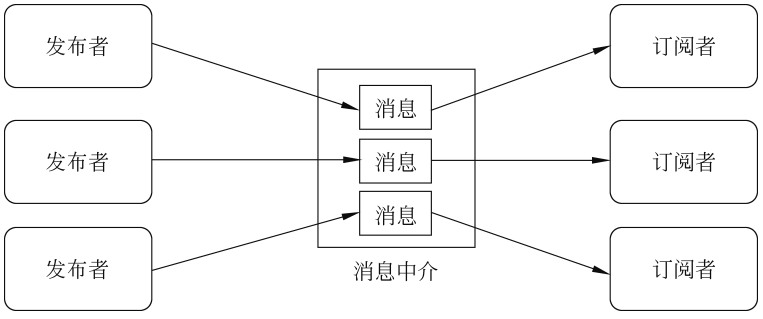

图 3-18 发布/订阅模式

在发布-订阅系统中，发布者和订阅者之间不需要互操作；消息的发布者异步地发送到所有对消息感兴趣的订阅者。例如，《读者》杂志社负责出版期刊《读者》，每月发布一刊。但是读者并不直接从杂志社购买，而是通过中国邮政或者其他渠道订阅，一般订阅一年，预交费用。中国邮政则是中介角色，负责接收读者订阅，从杂志社逐月批量获取杂志（消息），然后按照订阅信息派送到读者手中。

大多数消息系统同时支持消息队列模型和发布/订阅模型，例如，Java 消息服务（Java Message Service，JMS）。JMS 是一个 Java 平台中关于面向消息中间件的应用程序接口，用于两个应用程序之间，或分布式系统中发送消息，进行异步通信。

微信订阅号（https://mp.weixin.qq.com/）是一种微信公众号，是为媒体和个人提供的一种“发布/订阅”信息传播方式，主要功能是在微信侧给用户传达资讯，类似报纸杂志，提供新闻信息或娱乐趣事。订阅号认证媒体 1 天内可群发 1 条消息。媒体就是“发布者”，而微信订阅号则是“消息中介”，读者通过微信订阅号订阅感兴趣的消息，从而省去了从互联网搜索引擎查找信息的负担。

3.3　设 计 模 式

模式是某种事物的标准形式或使人可以照着做的标准形式。设计模式（Design Pattern）是在软件开发中针对普遍发生的问题而总结的被学术界、企业界和教育界认同的、反复使用的、经过分类编目的代码设计经验。设计模式是在特定环境下为解决某一通用软件设计问题提供的一套有效的解决方案，该方案描述了对象和类之间的相互作用。设计模式的应用提高了代码复用、可理解和可靠的程度。一个设计模式包括以下四个要素。

（1）名称（Pattern Name）：每一个模式都有自己的名字，起到对该模式的标识作用，方便进行引用。

（2）问题（Problem）：在面向对象的系统设计过程中频繁出现的特定场景，如在期望系统运行期间希望只有一个实例。设计模式所解决的问题通常是可扩展性等非功能性需求。

（3）解决方案（Solution）：针对问题所设计的类、类之间的关系、职责划分和协作方式。

（4）效果（Consequence）：采用该模式对系统的扩充性、可移植性等方面的影响。

例如，适配器模式的描述如表 3-1 所示。

表 3-1　适配器模式

名　称	适　配　器
问题描述	使用客户期望的接口访问遗留类（Legacy Class），但不期望修改遗留类
解决方案	客户所期望接口的实现类（Adapter）访问遗留类并执行必要的转换
效果	客户使用遗留类工作而无须修改遗留类； 针对不同的客户期望，可设计不同的适配器

与面向对象相关的设计模式分为三类：创建型模式、结构型模式和行为型模式。创建型模式着眼于对象的“创建、组合及表示”。结构型模式着眼于有关如何将类和对象组织和集成起来，以创建更大结构的问题和解决方案。行为型模式解决与对象间任务分配以及影响对象间通信方式的有关问题。

创建型模式有单例模式（Singleton Pattern）、简单工厂模式（Simple Factory Pattern）、工厂方法模式（Factory Method Pattern）、抽象工厂模式（Abstract Factory Pattern）、建造者模式（Builder Pattern）、原型模式（Prototype Pattern）等。

结构型模式有适配器模式（Adapter Pattern）、桥接模式（Bridge Pattern）、组合模式（Composite Pattern）、装饰模式（Decorator Pattern）、外观模式（Facade Pattern）、享元模式（Flyweight Pattern）、代理模式（Proxy Pattern）等。

行为型模式有职责链模式（Chain of Responsibility Pattern）、命令模式（Command Pattern）、解释器模式（Interpreter Pattern）、迭代器模式（Iterator Pattern）、中介者模式（Mediator Pattern）、备忘录模式（Memento Pattern）、观察者模式（Observer Pattern）、状态模式（State Pattern）、策略模式（Strategy Pattern）、模板方法模式（Template Method Pattern）、访问者模式（Visitor Pattern）等。

3.3.1 单例模式

单例模式指一个类有且仅有一个实例，并且自行实例化向整个系统提供。一种常见的实现方案如下面的代码所示。

```
public class MySingleton {
    //静态私有成员变量
    private static final MySingleton instance = new MySingleton();
    //私有构造方法
    private MySingleton() {}
    //静态公共方法
    public static MySingleton getInstance() {
        return instance;
    }
}
```

当类被加载时，静态变量 instance 会被初始化，此时类的私有构造函数会被调用，单例类的唯一实例将被创建。需要该实例的客户通过公共的静态方法 getInstance()获取。

在这个方案中，只要类被加载，实例就会被创建。下面的代码是一种“按需实例化”的设计。

```
public class MySingleton {
    /* 私有构造方法，防止被实例化 */
    private MySingleton() {
    }
    /* 此处使用一个内部类维护单例 */
    private static class SingletonFactory {
```

```
        private static MySingleton instance = new MySingleton();
    }
    /* 获取实例 */
    public static MySingleton getInstance() {
        return SingletonFactory.instance;
    }
}
```

在这种设计中，类加载时不会实例化对象 MySingleton。因为第一次调用 getInstance()方法时才加载内部类，初始化在该内部类中定义的静态引用变量 instance，该成员变量只初始化一次。

3.3.2　抽象工厂模式

如果有一组提供相同功能的产品，使用产品的客户只关心产品功能而不关心具体产品的实现细节，当增加新的产品后不愿意修改客户代码，这种场景下适合采用抽象工厂模式。

例如，客户程序需要通过不同方式发送消息：电子邮件或者短信，因此需要一个负责邮件发送的对象(MailSender)和一个负责短信发送(SmsSender)的对象，将来也有可能又需要负责微信发送的对象。

把负责消息发送的对象称为产品，这些产品对客户的接口是相同的；不同的产品使用不同的工厂生产，即创建多个工厂类，这样一旦需要增加新的功能，直接增加新的工厂类就可以了，不需要修改现有的代码。

首先定义产品接口：

```
public interface Sender {
    void send();
}
```

类 MailSender 实现该接口：

```
public class MailSender implements Sender {
    @Override
    public void send() {
        System.out.println("this is a mail sender!"); //此处仅模拟发送行为
    }
}
```

类 SmsSender 也实现该接口：

```
public class SmsSender implements Sender {
    @Override
    public void send() {
        System.out.println("this is a smssender!"); //此处仅模拟发送行为
    }
```

```
}
```

然后定义抽象工厂类：

```
public abstract class Provider{
    Sender produce();
}
```

生产 MailSender 对象的工厂类：

```
public class MailFactory extends Provider {
    @Override
    public Sender produce() {
        return new MailSender();
    }
}
```

生产 SmsSender 对象的工厂类：

```
public class SmsFactory extends Provider {
    @Override
    public Sender produce() {
        return new SmsSender();
    }
}
```

客户类首先创建工厂对象，然后让工厂对象生产产品(MailSender)，最后通过该产品实现消息发送(send)。

```
public class Client {
    public static void main(String[] args) {
        Factory factory = new MailFactory();
        Sender sender = factory.produce();
        sender.send();
    }
}
```

将来如果期望增加新的消息发送形式，如微信发送，那么新增一个产品类 WeChatSender 实现接口 Sender；新增工厂类 WeChatFactory 继承抽象工厂 Factory 即可。对现有代码无须做任何改动。

3.3.3 工厂方法模式

工厂方法模式(Factory Method Pattern)简称工厂模式(Factory Pattern)，又可称作虚拟构造器模式(Virtual Constructor Pattern)、多态工厂模式(Polymorphic Factory Pattern)。工厂父类负责定义创建产品对象的公共接口，而工厂子类则负责生成具体的产品对象。目的是将产品类的实例化操作延迟到工厂子类中完成。

例如，对于发送消息的客户来说，不管是使用电子邮件，还是使用短信，都使用实例方法 void send(String message)，只要工厂子类生产出满足这个接口要求的产品即可。

客户程序如下。

```
public class Client{
    public static void main(String[] args) {
        Sender sender = SendFactory.produceMail();
        sender.send("message");
    }
}
```

这里的产品对象是电子邮件，Sender 是其抽象类或者接口定义。

```
public interface Sender {
    public void Send(String msg);
}
```

电子邮件消息发送器 MailSender 和短信消息发送器 SmsSender 是具体的实现类。

```
public class MailSender implements Sender {
    @Override
    public void Send(String msg) {
        //…;
    }
}
public class SmsSender implements Sender {
    @Override
    public void Send(String msg) {
        //…;
    }
}
```

有了具体的产品，让工厂使用静态方法实例化产品即可。

```
public class Factory {
    public static Sender produceMail(){
        return new MailSender();
    }

    public static Sender produceSms(){
        return new SmsSender();
    }
}
```

可以看到，在工厂方法模式中，关键是客户和工厂事先约定好产品规格，即产品的抽象类或接口定义。然后客户按照这个定义使用产品；工厂按照这个定义生产产品。工厂方法用来创建客户所需要的产品，同时还向客户隐藏了哪种具体产品类将被实例化这一

细节。

3.3.4 原型模式

原型模式(Prototype Pattern)首先创建一个对象作为原型实例,然后应用对象克隆根据这个原型实例得到新的对象。通过克隆方法所创建的对象是全新的对象,它们在内存中拥有新的地址,每一个克隆对象都是独立的。在需要一个类的大量对象的时候,使用原型模式是最佳选择,因为原型模式是在内存中对这个对象进行复制,要比直接使用关键字 new 创建这个对象性能好很多,在这种情况下,需要的对象越多,原型模式体现出的优点越明显。

在 Java 语言中,Object 类的 clone()方法用于实现浅克隆,该方法使用起来很方便,直接调用 super.clone()方法即可实现克隆。浅克隆(Shallow Clone)指当原型对象被复制时,只复制它本身和其中包含的值类型的成员变量,而引用类型的成员变量并没有复制。而深克隆(Deep Clone)则除了对象本身被复制外,对象所包含的所有成员变量也将被复制。

假设孙悟空是类 Monkey 的对象,现在孙悟空需要克隆 100 万个跟自己一模一样的猴子对付妖怪。那么首先需要让类 Monkey 实现 Cloneable 接口。

```
public class Monkey implements Cloneable {
    public Monkey() {
        System.out.println("An object of Monkey.");
    }

    @Override
    protected Monkey clone(){
        Monkey a = null;
        try {
            a = (Monkey) super.clone();
        }catch (CloneNotSupportedException e){
            e.printStackTrace();
        }
        return a;
    }
}
```

然后孙悟空就可以变化了。

```
public class Test{
    public static void main(String[] args){
        Monkey wuKong = new Monkey();       //创建孙悟空
        Monkey[] a = new Monkey[1000000];
        int i= 1;
        while (i < 1000000){
            a[i] = wuKong.clone();          //克隆猴子
```

```
        }
    }
}
```

3.3.5　建造者模式

建造者模式将复杂对象的构建与它的表示分离，使得同样的构建过程可以创建不同的表示。当一个类构造方法的参数多于 4 个，例如，组装计算机类 Computer 需要 5 个参数：cpu、ram、usbCount、keyboard 和 display，其中，cpu 与 ram 是必填参数，而其他 3 个是可选参数，那么通常设计有如下构造方法。

```
public class Computer {
    //…
    public Computer(String cpu, String ram) {
        this(cpu, ram, 2);                              //默认两个 USB 接口
    }
    public Computer(String cpu, String ram, int usbCount) {
        this(cpu, ram, usbCount, "104");                //默认 104 键盘
    }
    public Computer(String cpu, String ram, int usbCount, String keyboard) {
        this(cpu, ram, usbCount, keyboard, "17");       //默认 17 英寸显示器
    }
    public Computer(String cpu, String ram, int usbCount, String keyboard,
                    String display) {
        this.cpu = cpu;
        this.ram = ram;
        this.usbCount = usbCount;
        this.keyboard = keyboard;
        this.display = display;
    }
}
```

这种设计要求程序员首先要决定使用哪一个构造方法，然后理解里面参数的含义，学习成本较高。

在 Computer 类中创建一个静态内部类 Builder，然后将构造 Computer 所需的参数都作为 Builder 类的成员变量，并在 Builder 类中设计实例方法 build()构建 Computer 类的实例并返回。这样对客户程序而言就隐藏了各种零件实例化和组装细节。

客户程序创建内部类 Computer.Builder 的实例，然后调用 build()方法返回 Computer 对象。代码如下。

```
public class Client{
    public static void main(String[] args){
Computer computer= new Computer.Builder("2.0GHz","256G")
                .setDisplay("24 英寸")
```