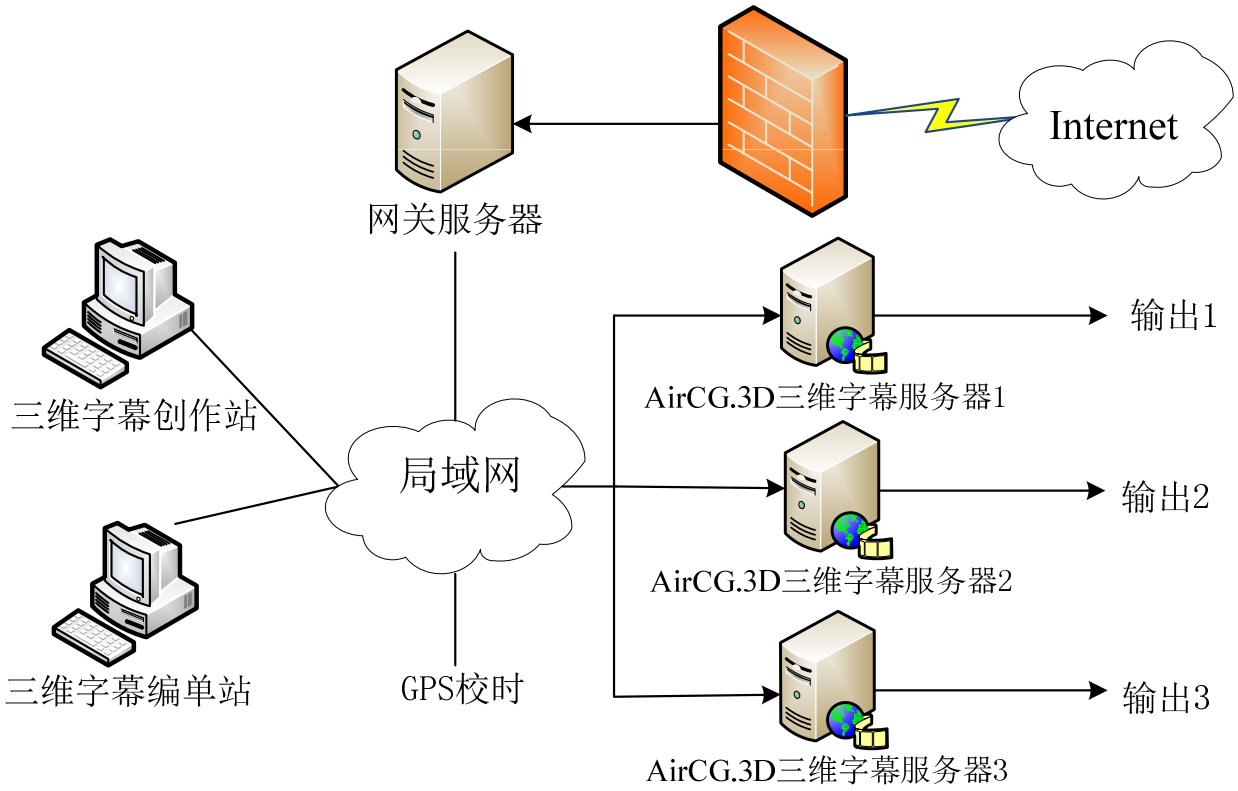
4.3.11 客户机/服务器风格

- AirCG.3D三维网络字幕系统是一套对电视台字幕部分进行集中管理及播放的系统。

AirCG.3D演播室在线包装

4.3.11 客户机/服务器风格

- 应用实例
  - AirCG.3D三维网络字幕系统的结构



4.3.12 浏览器/服务器风格

- 浏览器/服务器风格是三层C/S风格的一种实现方式。
- 主要包括浏览器，Web服务器和数据库服务器。

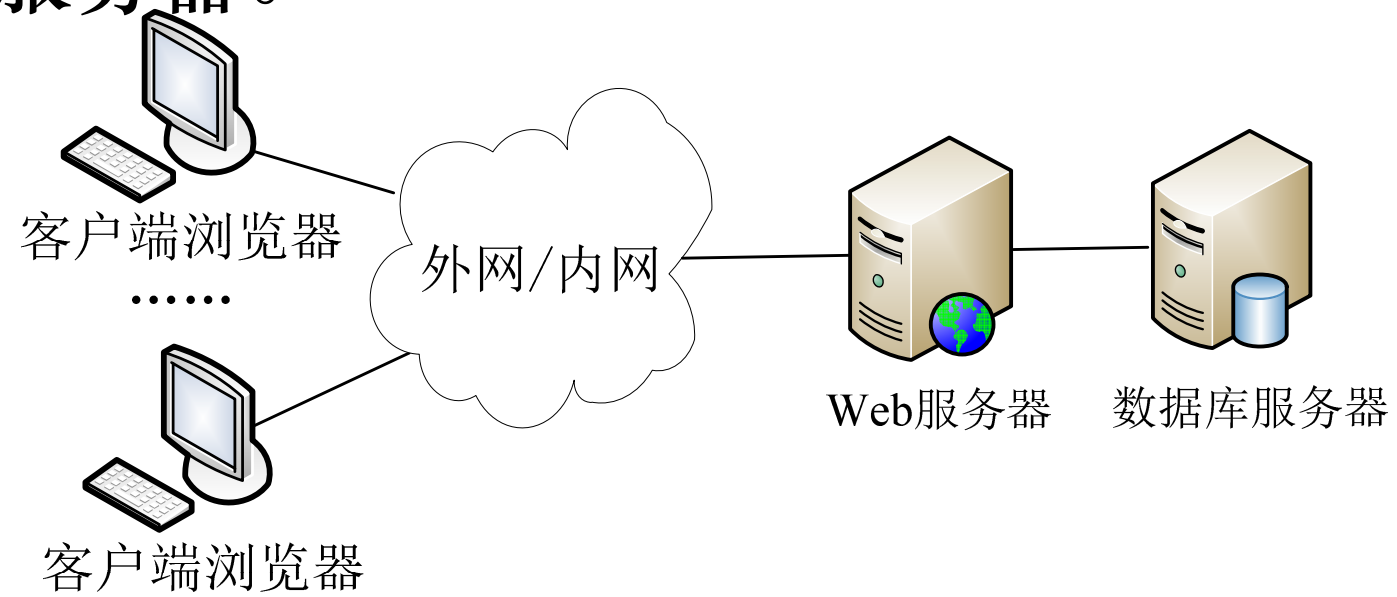


图4.34 B/S架构

4.3.12 浏览器/服务器风格

- 与三层C/S结构的解决方案相比，B/S架构在客户机上采用了WWW浏览器，将Web服务器作为应用服务器。
- B/S架构核心是Web服务器，数据请求、网页生成、数据库访问和应用程序执行全部由Web服务器来完成。
- 在B/S架构中，系统安装、修改和维护全在服务器端解决，客户端无任何业务逻辑。

4.3.12 浏览器/服务器风格

- 优点
  - （1）客户端只需要安装浏览器，操作简单，能够发布动态信息和静态信息。
  - （2）运用HTTP标准协议和统一客户端软件，能够实现跨平台通信。
  - （3）开发成本比较低，只需要维护Web服务器程序和中心数据库。客户端升级可以通过升级浏览器来实现。

4.3.12 浏览器/服务器风格

- 缺点
  - （1）个性化程度比较低，所有客户端程序的功能都是一样的。
  - （2）客户端数据处理能力比较差，加重了Web服务器的工作负担，影响系统的整体性能。（3）在B/S架构中，数据提交一般以页面为单位，动态交互性不强，不利于在线事物处理（Online Transaction Processing，OLTP）。
  - （4）B/S架构的可扩展性比较差，系统安全性难以保障。
  - （5）B/S架构的应用系统查询中心数据库，其速度要远低于C/S架构。

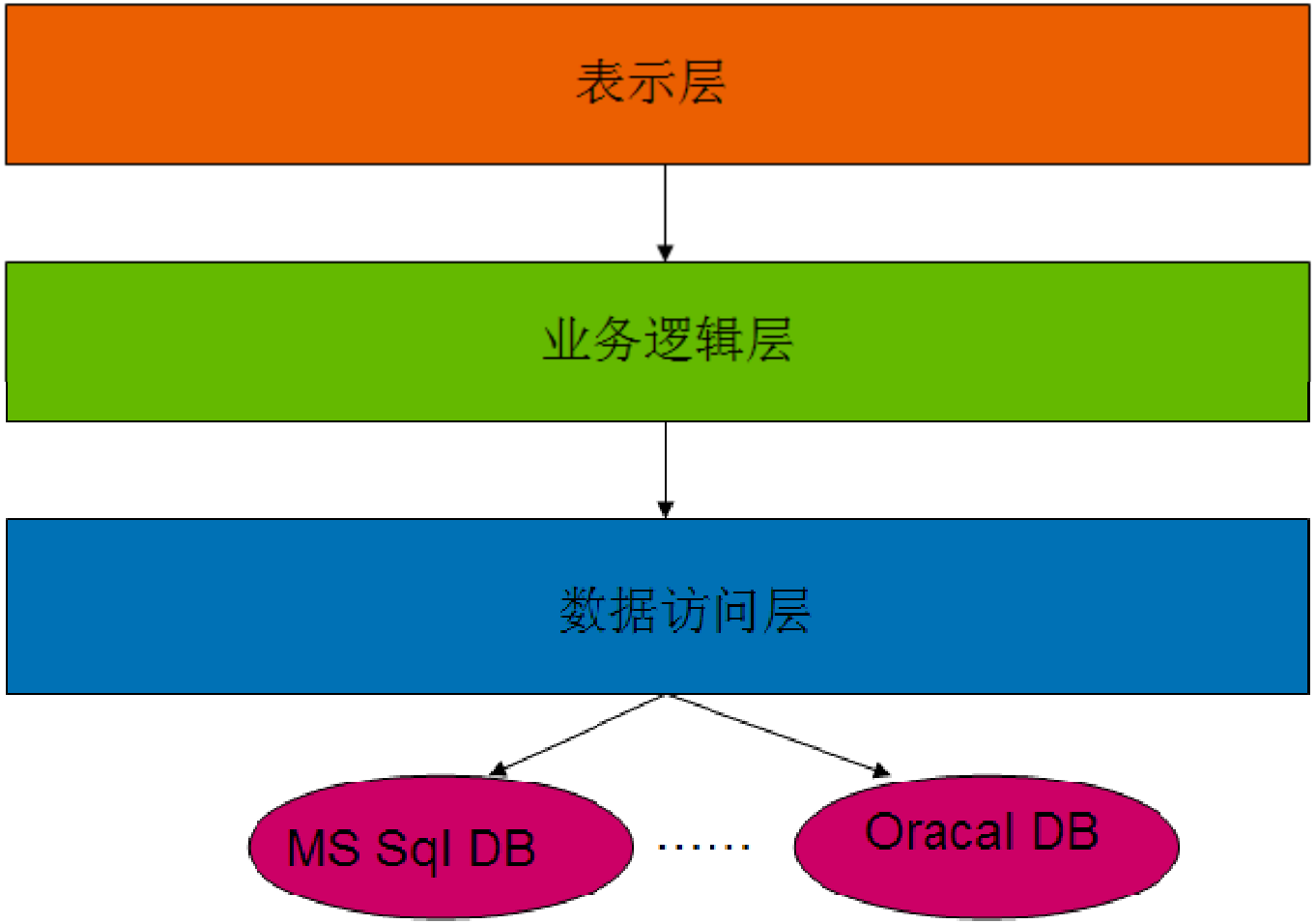
4.3.12 浏览器/服务器风格

- 虽然B/S结构有许多优越性，但是C/S结构起步较早，技术很成熟，网络负载也非常小，因而未来一段时间内将会出现B/S结构和C/S结构共存的现象。
- 然而，计算模式的未来发展趋势将向B/S结构转变。

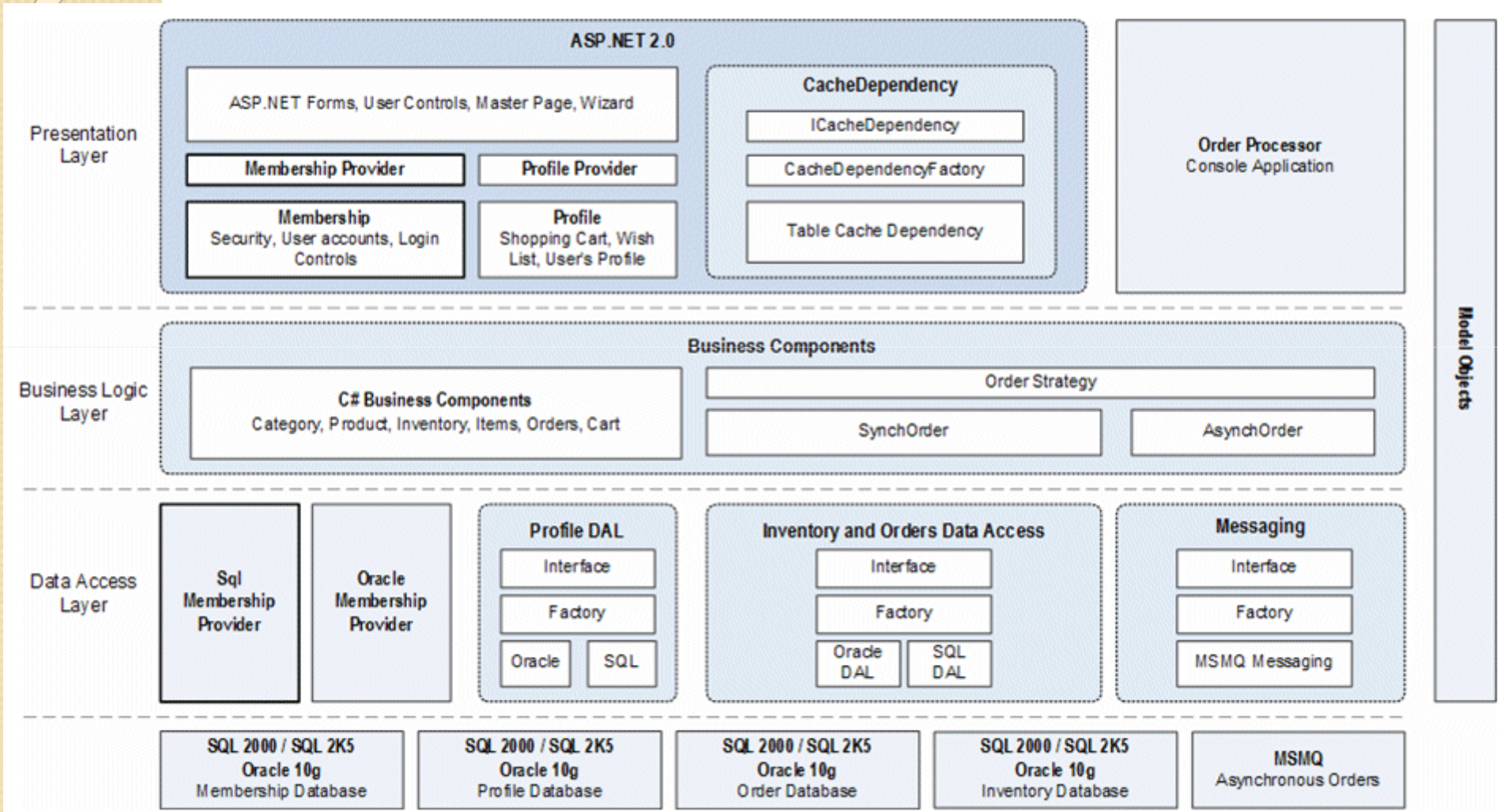
4.3.12 浏览器/服务器风格

- 应用实例：PetShop
- PetShop 是一个范例，微软用它来展示.Net企业系统开发的能力。
- PetShop是一个小型的项目，系统架构与代码都比较简单，但凸现了许多颇有价值的设计与开发理念。

PetShop的系统架构设计



PetShop 4的系统架构设计



4.3.12 浏览器/服务器风格

PetShop架构分为三层：

（1）数据访问层：其功能主要是负责数据库的访问。

（2）业务逻辑层：是整个系统的核心，它与这个系统的业务（领域）有关。PetShop的业务逻辑层的相关设计，均和网上宠物店特有的逻辑相关。

（3）表示层：是系统的UI部分，负责使用者与整个系统的交互。在这一层中，理想的状态是不应包括系统的业务逻辑。表示层中的逻辑代码，仅与界面元素有关。

4.3.13 平台/插件风格

- 插件（Plug-in）是一种遵循统一的预定义接口规约编写出来的程序，应用程序在运行时通过接口规约对插件进行调用，以扩展应用程序的功能。
- 插件的本质在于不修改程序主体（或者程序运行平台）的情况下对软件功能进行扩展与加强。

4.3.13 平台/插件风格

- 这意味着软件开发者可以通过公布插件的预定义接口规约，从而允许第三方的软件开发者通过开发插件对软件的功能进行扩展，而无须对整个程序代码进行重新编译。

4.3.13 平台/插件风格

- “平台/插件”软件结构将待开发的目标软件分为两部分：
  - 程序的主体或主框架，可定义为平台
  - 功能扩展或补充模块，可定义为插件

4.3.13 平台/插件风格

- 平台所完成的功能应为一个软件系统的核心和基础,可以把平台基本功能分为两个部分:
  - 内核功能：是整个软件的重要功能，一个软件的大部分功能因由内核功能完成。
  - 插件处理功能：用于扩展平台和管理插件，为插件操纵平台和与插件通信提供标准平台扩展接口。
  - 如：Eclipse IDE是Eclipse的运行主体平台，编辑c/c++程序可以应用CDT插件；使用SVN可以安装SVN Repository Exploring。

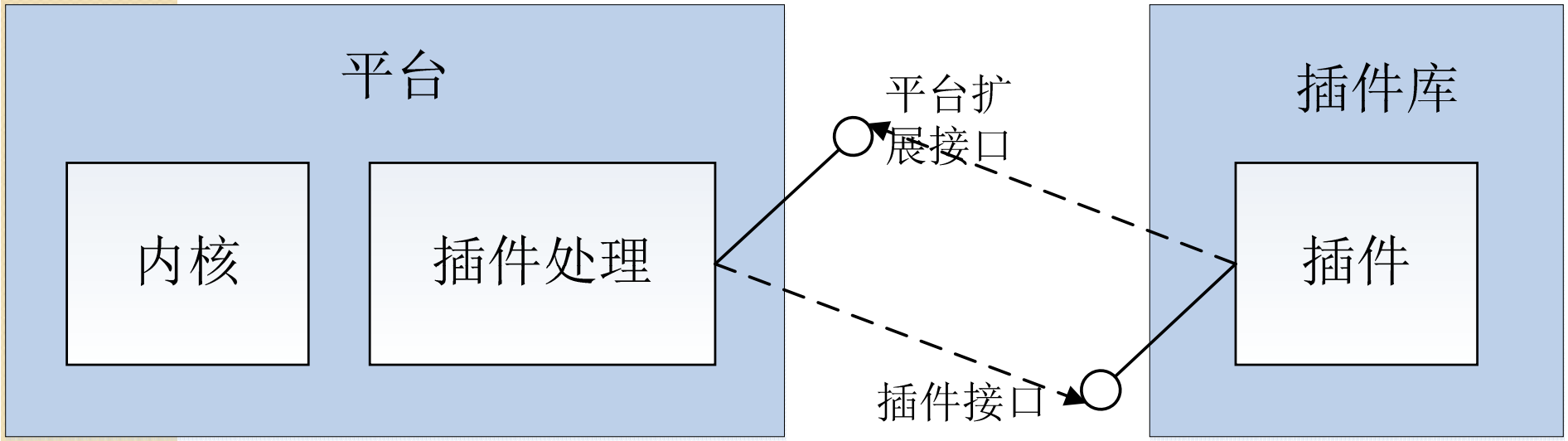
4.3.13 平台/插件风格

- 插件是能动态插入到平台中的程序模块，提供给系统某一方面的功能，但多个插件能使系统功能完善，完成多个复杂的处理。
- 插件受到的约束是:
  - 1)插件必须能在运行过程中动态地插入平台和从平台中注销，且不影响系统的运行;
  - 2)当在系统中插入插件后，系统的功能得到扩展或升级;
  - 3)多个插件之间、插件和平台之间不会发生冲突。

4.3.13 平台/插件风格

- 实现“平台/插件”(Platform / Plug-in) 结构软件需要定义两个标准接口
  - 平台扩展接口：完全由平台实现，插件只是调用和使用
  - 插件接口：完全由插件实现，平台也只是调用和使用。
  - 平台扩展接口实现插件向平台方向的单向通信，插件通过平台扩展接口可获取主框架的各种资源和数据，可包括各种系统句柄，程序内部数据以及内存分配等。
  - 插件接口为平台向插件方向的单向通信，平台通过插件接口调用插件所实现的功能，读取插件处理数据等。

4.3.13 平台/插件风格



4.3.13 平台/插件风格

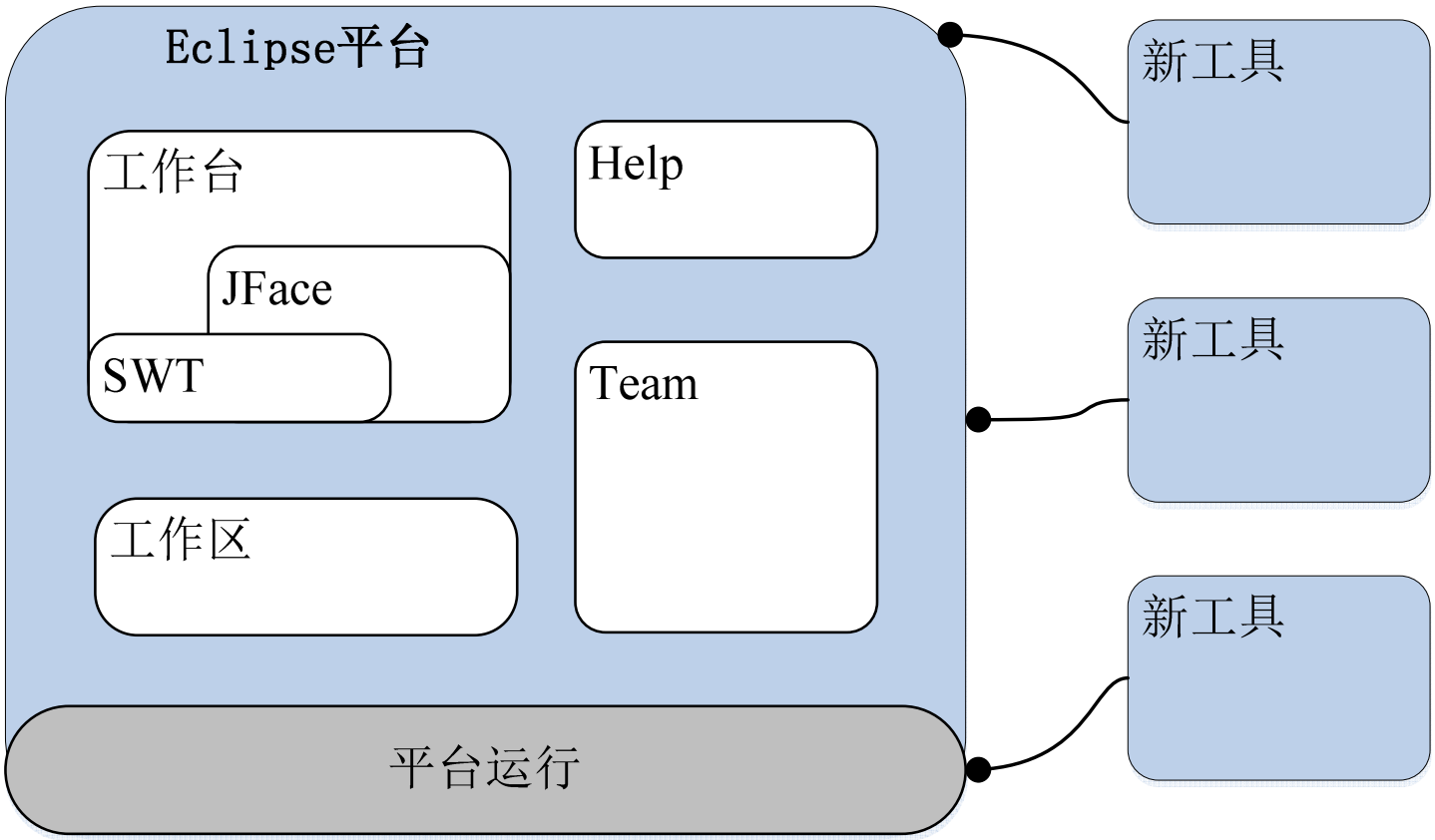

电脑管家 - 系统优化
清理垃圾
清除痕迹
插件清理
文件清理
系统加速
已信任插件(0) 清理记录(2)
发现 44 款插件，清理不需要的插件可提高系统性能
插件名称
公司
评价
清理建议
操作
支付宝安全支付控件
支付宝提供在线安全支付保护功能相关控件。
详情 支付宝 中评 按需要清理 投票 信任
搜狗拼音语言栏支持模块
搜狗输入法在win7、vista系统下的辅助模块；清理以后不影响输入法...
详情 搜狗 中评 按需要清理 投票 信任
WMP录制组件
Windows系统自带的播放器WindowsMediaPlayer附带的媒体录制...
详情 微软 中评 按需要清理 投票 信任
百度云管家检测功能控件
百度云管家监测功能相关控件。
详情 百度 中评 按需要清理 投票 信任
腾讯电脑管家辅助插件
腾讯电脑管家的浏览器辅助插件，提供监控页面、对搜索页嵌入软件...
详情 腾讯 中评 按需要清理 投票 信任
推荐选项
重新扫描
立即清理

4.3.13 平台/插件风格

- 优点：
  - （1）降低系统各模块之间的互依赖性
  - （2）系统模块独立开发、部署、维护
  - （3）根据需求动态的组装、分离系统
- 缺点
  - 插件是别人开发的可以用到某主程序中的，只服务于该主程序，可重用性差。

4.3.13 平台/插件风格

- 应用实例
  - Eclipse开发平台



4.3.14 面向Agent风格

- 面向Agent风格的一个重要特点，就是编程时不再仅仅只是用对象的思维来考虑问题，而是要更进一步，用Agent智能体的思维去考虑问题。

4.3.14 面向Agent风格

- 面向Agent风格（Agent-oriented）的基本思想是认为事物的属性，特别是动态特性在很大程度上受到与其密切相关的人和环境的影响，将影响事物的主观与客观特征相结合抽象为系统中的Agent，作为系统的基本构成单位，通过Agent之间的合作实现系统的整体目标。

4.3.14 面向Agent风格

- Agent：一个能够根据它对其环境的感知，主动采取决策和行为的软件实体。
- Agent组件：对系统处理的高度抽象，具有高度灵活和高度智能特色的软件实体。
- Agent连接件：对复合型组件的连接，该连接能够提供通信、协调、转换、接通等服务。

4.3.14 面向Agent风格

- Agent组件有别于以往任何系统的组件类型，其所具有的自主性、智能性、交互性等特性是传统架构对象所不具备的。
- 多Agent系统中的连接件并非显式地将两个不同的组件联系起来。不同Agent之间的联系是根据运行时状态来决定的。

4.3.14 面向Agent风格

- 优点
  - 面向Agent的软件工程方法对于解决复杂问题是一种好的技术,特别是对于分布开放异构的软件环境。
- 缺点
  - 大多数结构中Agent 自身缺乏社会性结构描述和与环境的交互。

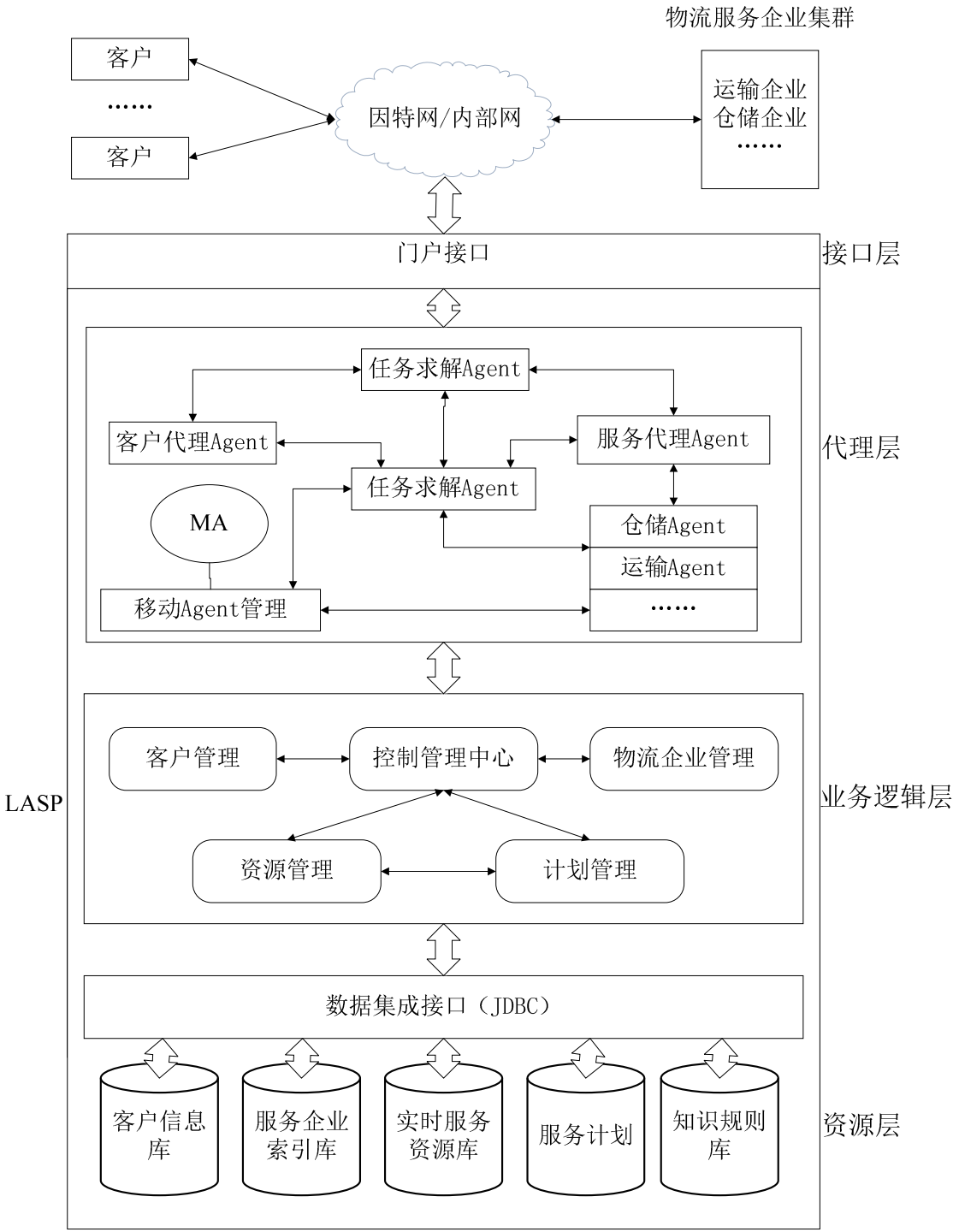


图4-38LASP物流管理系统的架构

4.3.15 面向方面软件架构风格

- 面向方面的编程(AOP-- -Aspect- -Oriented Programming)
  - 一般认为AOP在传统软件架构基础上增加了方面组件(Aspect Component)这一新的构成单元，通过方面组件来封装系统的横切关注点。
  - 系统的有些特性和需求是横切于系统的每一个层面中，并融于系统的每一个组件中，这种特性称为系统的方面(Aspect)需求特性或关注点
    - 如系统中的时间要求、业务逻辑、性能、安全性和错误检测、QoS监测等。

4.3.15 面向方面软件架构风格

- 应用AOP的主要目的----尽量分离“技术问题实现”和“业务问题实现”
  - 它允许开发者能够对横切关注点进行模块化设计；
  - 能够实现分散关注，将通用需求功能从不相关类之中分离出来；
  - 能够实现代码重用。

4.3.15 面向方面软件架构风格

- 例如：日志功能
  - 日志代码往往水平地散布在所有对象层次中----在控制层、业务层和数据访问层都需要日志功能，并且而与它所散布到的对象的核心功能毫无关系。

4.3.15 面向方面软件架构风格

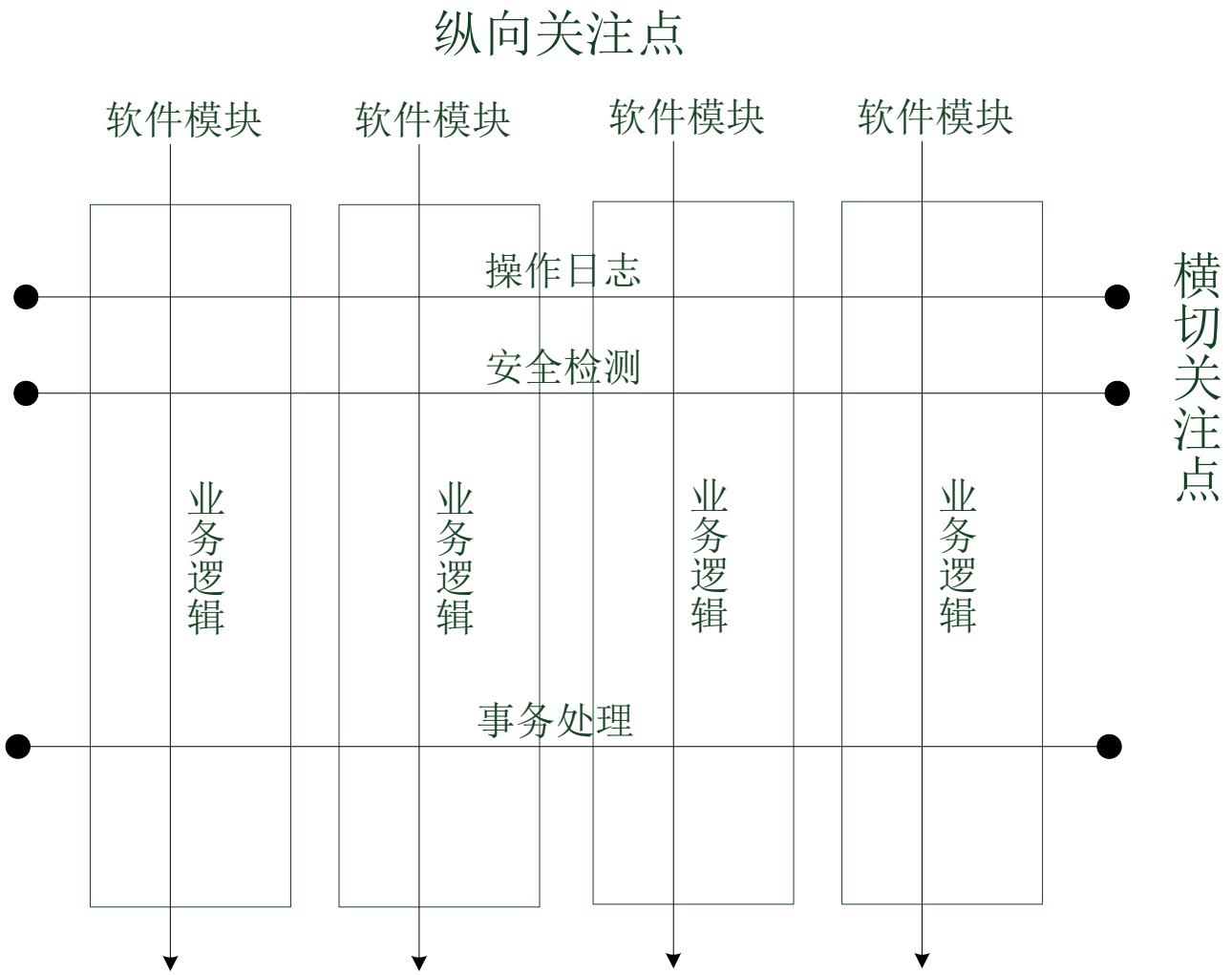


图4.39 AOP示意图

4.3.15 面向方面软件架构风格

- 优缺点分析
  - 可以定义交叉的关系，并将这些关系应用于跨模块的、彼此不同的对象模型
  - AOP 同时还可以让我们层次化功能性而不是嵌入功能性，从而使得代码有更好的可读性和易于维护。
  - 它会和面向对象编程可以很好地合作，互补

4.3.16 面向服务架构风格

- 1996年，Gartner Group 提出面向服务架构模型SOA
- SOA 是一个组件模型，它将应用程序的不同功能单元（服务）通过这些服务之间定义良好的接口和契约联系起来。
  - 服务(service)是一个粗粒度的、可发现的软件实体。
  - 接口是采用中立的方式进行定义的，应独立于实现服务的硬件平台、操作系统和编程语言，便于不同系统中的服务以一种统一和通用的方式进行交互。

4.3.16 面向服务架构风格

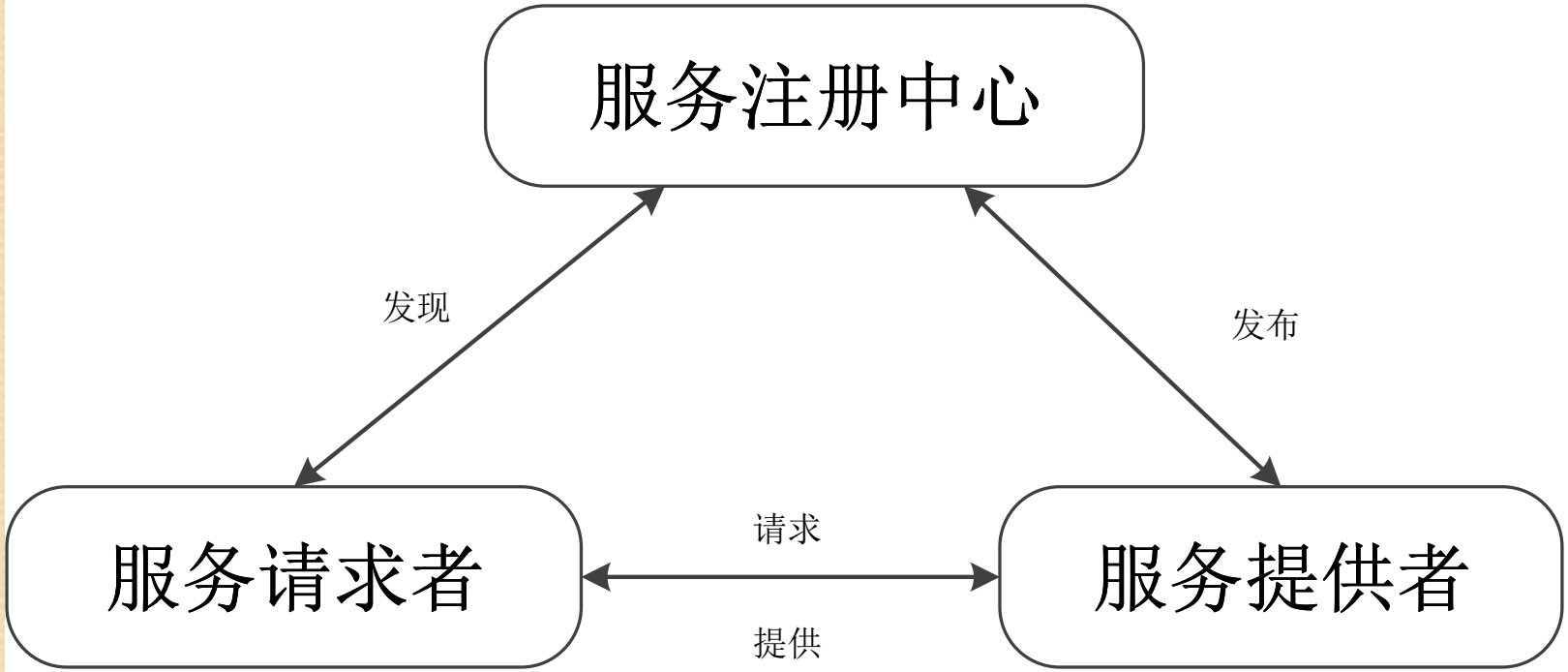


图4.41 SOA架构

4.3.16 面向服务架构风格

- 服务请求者：可以是服务或者第三方的用户，通过查询服务提供者在服务注册中心发布的服务接口的描述，通过服务接口描述来通过RPC或者SOAP进行绑定调用服务提供者所提供的业务或服务。
- 服务提供者：作为服务管理者和创建者，必须将服务描述的接口发布到服务注册中心才能被潜在的服务请求 发现，能够为合适的服务请求者提供服务 。
- 服务注册中心：相当于服务接口的管理中心，服务请求者者 能够通过查询服务注册中心的数据库来找到需要的服务调用方式和接口描述信息。

4.3.16 面向服务架构风格

- 发布：为了便于服务请求者发现，服务提供者将对服务接口的描述信息发布到服务注册中心上。
- 发现：服务请求者通过查询服务注册中心的数据库来找到需要的服务，服务注册中心能够通过服务的描述对服务进行分类，使服务提供者更快定位所需要的服务范围。
- 绑定和调用：服务请求者在查询到所需要服务描述信息，根据这些信息服务请求者能够调用服务。

4.3.16 面向服务架构风格

- 面向服务软件架构风格在于具有基于标准、松散耦合、共享服务和粗粒度等优势，表现为易于集成现有系统、具有标准化的架构、提升开发效率、降低开发维护复杂度等。
- 通过采用SOA架构，在进行一次开发成本急剧减少的同时，由于系统具有松散耦合的特征使得维护成本也大大减少。

4.3.16 面向服务架构风格

- 优点
  - （1）灵活性，根据需求变化，重新编排服务。
  - （2）对IT资产的复用。
  - （3）使企业的信息化建设真正以业务为核心。业务人员根据需求编排服务，而不必考虑技术细节。

4.3.16 面向服务架构风格

- 缺点
  - （1）服务的划分很困难。
  - （2）服务的编排是否得当。
  - （3）如果选择的接口标准有问题，如主流的Web service之类，会带来系统的额外开销和不稳定性。
  - （4）对IT硬件资产还谈不上复用。
  - （5）目前主流实现方式接口很多，很难统一。
  - （6）目前主流实现方式只局限于不带界面的服务的共享。

4.3.16 面向服务架构风格

- 应用实例
  - 金蝶EAS (Enterprise Application Suite)

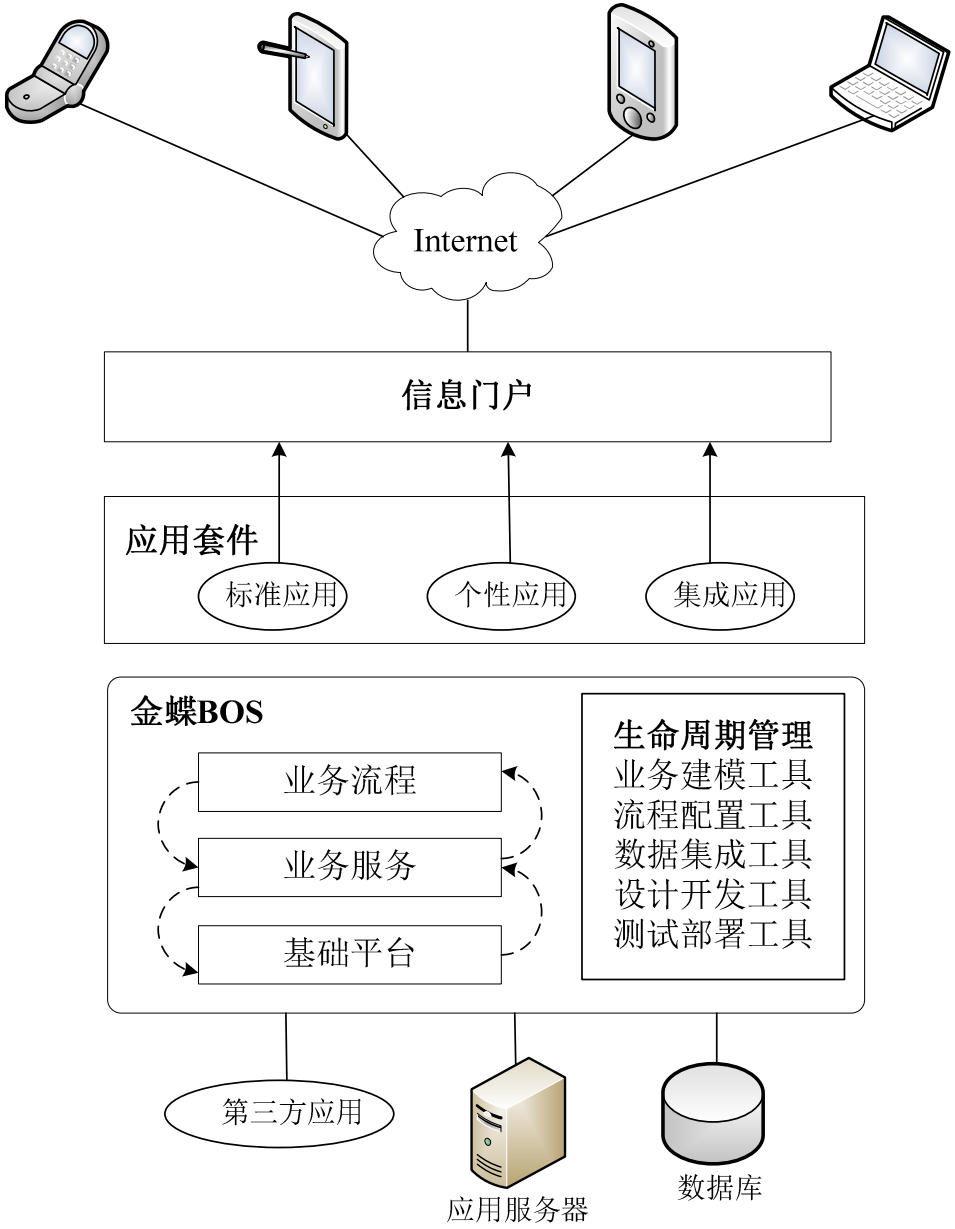
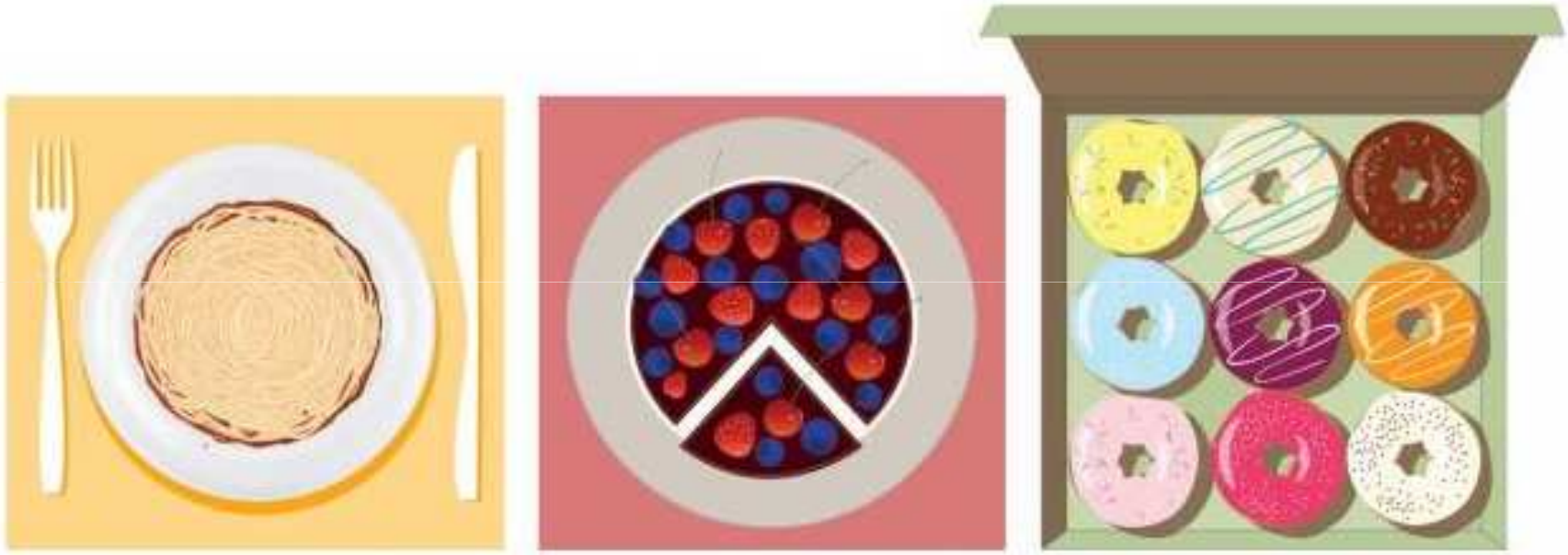


图4.42金蝶EAS技术架构

微服务架构



With monolithic, tightly coupled applications, all changes must be pushed at once, making continuous deployment impossible.

Traditional SOA allows you to make changes to individual pieces. But each piece must be carefully altered to fit into the overall design.

With a microservices architecture, developers create, maintain and improve new services independently, linking info through a shared data API.

图片来源：http://www.kanbansolutions.com/blog/microservices-architecture-friend-or-foe/

微服务架构

- 微服务架构（Microservice Architect）是一种将单个应用程序开发为一组小型服务的方法，每个小型服务都在自己的进程中运行，并与轻量级机制（通常是HTTP资源API）进行通信。
- 这些服务围绕业务功能构建，可通过全自动部署机制独立部署。
- Amazon、Netflix、LinkedIn、Spotify、SoundCloud和其他公司已将其应用架构演变为微服务架构（MSA）。

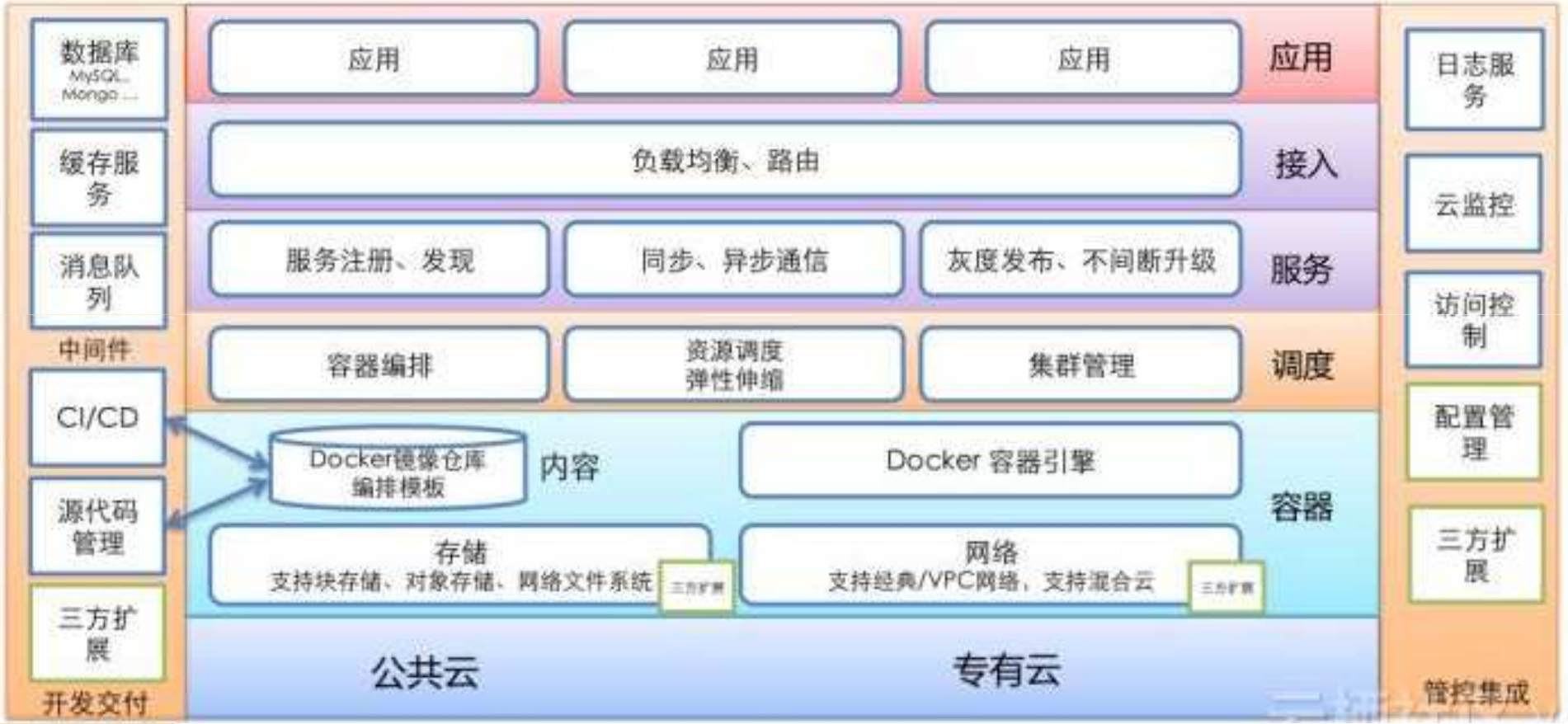
微服务架构

- 微服务架构风格的特点是：
  - 围绕业务能力组织系统；
  - 自动化部署；
  - 智能化；
  - 对程序和数据的去中心化控制。
- 这种风格可以设计出灵活、模块化且易扩展的架构。

微服务架构

- 微服务架构是面向服务架构的一个变形，由于它注重于单个服务的构建，能够轻松的增加、改进或删除服务，因此具有松耦合的特点。
- 使用微服务进行架构设计并非易事，因为使用微服务意味着需要管理分布式架构及接受其所带来的一系列挑战，包括管理网络延迟和不可靠性、容错能力、复杂服务的编排、数据一致性、事务管理以及负载均衡。
- 云基础架构和新技术在实现微服务架构以及克服相关挑战和复杂性方面发挥着重要作用。

基于Docker的微服务架构技术栈



微服务架构

- 优点：
  - 每个微服务模块都是独立部署独立维护的，每次更新都不会对整体业务有所影响，而且是基于共享的通讯机制的。
  - 在部署或者扩容的时候并不用对整个应用进行整体扩容，只需要将应用中遇到性能瓶颈的模块进行扩容就可以了，这样的更细粒度的扩容方式可以节省资源。

微服务架构

- 缺点：
  - 分布性系统天生的复杂性，如网络延迟、容错性、不可靠的网络、异步机制等
  - 运维开销及成本增加，一个单体应用被拆分部署成十几个服务，增大了运维的压力。
  - 隐式接口及接口匹配问题，不同组件间协作需要统一可管理的接口。
  - 代码重复，为了尽量实现松耦合，相同功能的底层功能会在不同服务内多次实现。
  - 异步机制，微服务往往使用异步编程、消息与并行机制，会引入新的故障点。
  - 其他的像不可变基础设施、管理难度、微服务架构的推进等。

4.3.17 正交架构风格

- 正交软件架构(Orthogonal Software Architecture) 由组织层(Layer)和线索(Thread)的组件(Component)构成。
  - 层是由一组具有相同抽象级别(Level of Abstraction)的组件构成。
  - 线索是子系统的特例，它是由完成不同层次功能的组件组成(通过相互调用来关联)，每一条线索完成整个系统中相对独立的一部分功能。
  - 如果线索是相互独立的，即不同线索中的组件之间没有相互调用，那么这个结构就是完全正交的。

4.3.17 正交架构风格

- 正交软件架构是一种以垂直线索组件族为基础的层次化结构
- 其基本思想是把应用系统的结构按功能的正交相关性，垂直分割为若干个线索(子系统)，线索又分为几个层次，每个线索由多个具有不同层次功能和不同抽象级别的组件构成。

4.3.17 正交架构风格

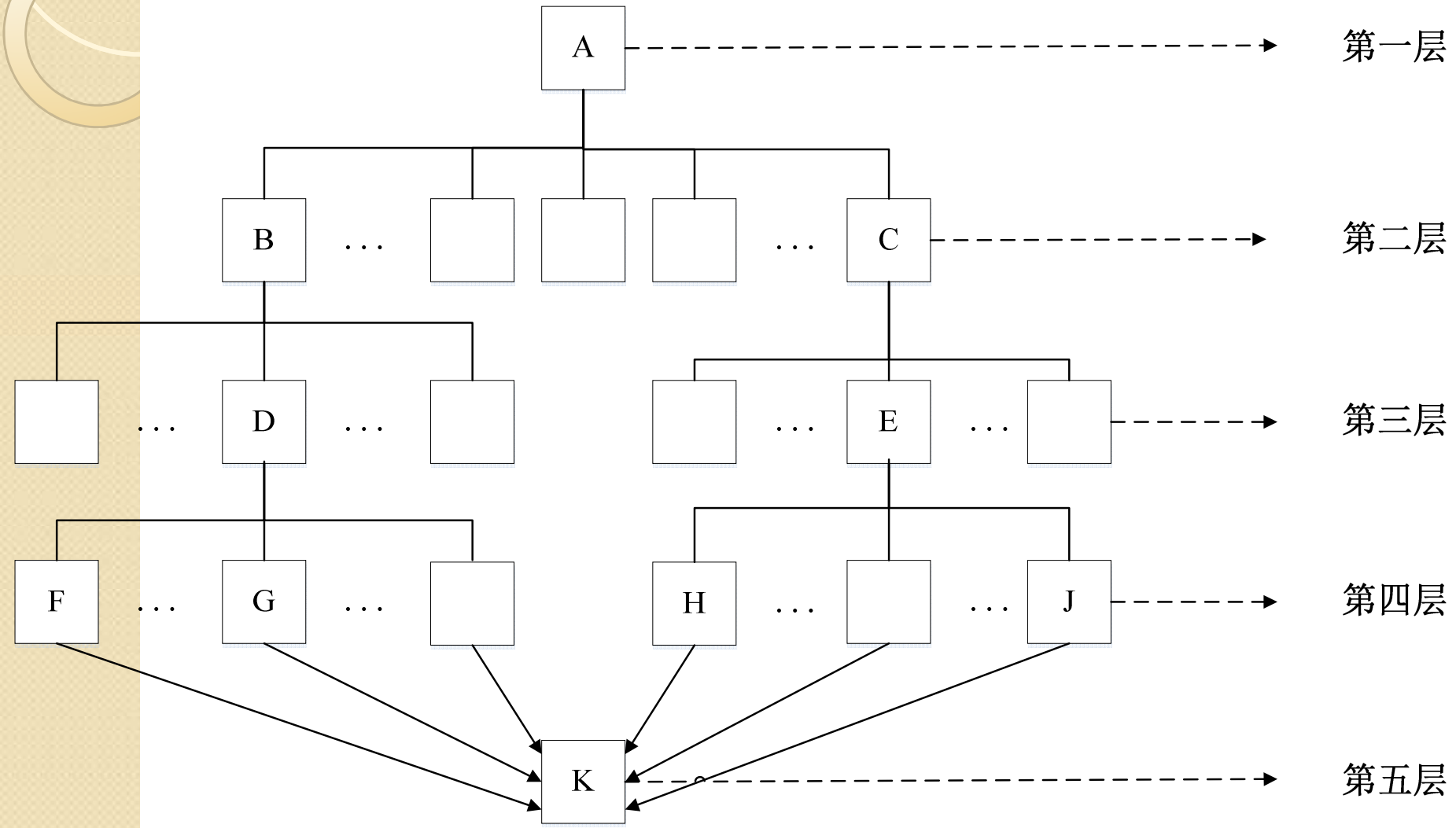


图4.43 正交软件架构

4.3.17 正交架构风格

- 特点：
  - （1）由完成不同功能的n(n>1)个线索(子系统)组成；
  - （2）系统具有m(m >1)个不同抽象级别的层；
  - （3）线索之间是相互独立的(正交的)；
  - （4）系统有一个公共驱动层(一般为最高层)和公共数据结构(一般为最低层)。

4.3.17 正交架构风格

- 优点
  - （1）结构清晰，易于理解。由于线索功能相互独立，组件的位置可以清楚地说明它所实现的抽象层次和负担的功能。
  - （2）易修改，可维护性强。由于线索之间是相互独立的，所以对一个线索的修改不会影响到其他线索。
  - （3）可移植性强，重用粒度大。因为正交结构可以为一个领域内的所有应用程序所共享，这些软件有着相同或类似的层次和线索，可以实现架构级的重用。

4.3.17 正交架构风格

- 缺点
  - 在实际应用中，并不是所有软件系统都能完全正交化，或者有时完全正交化的成本太高。因此，在进行应用项目的软件架构设计时，必须反复权衡进一步正交化的额外开销与所得到的更好的性能之间的关系。

4.3.17 正交架构风格

- 应用实例：汽修服务管理系统

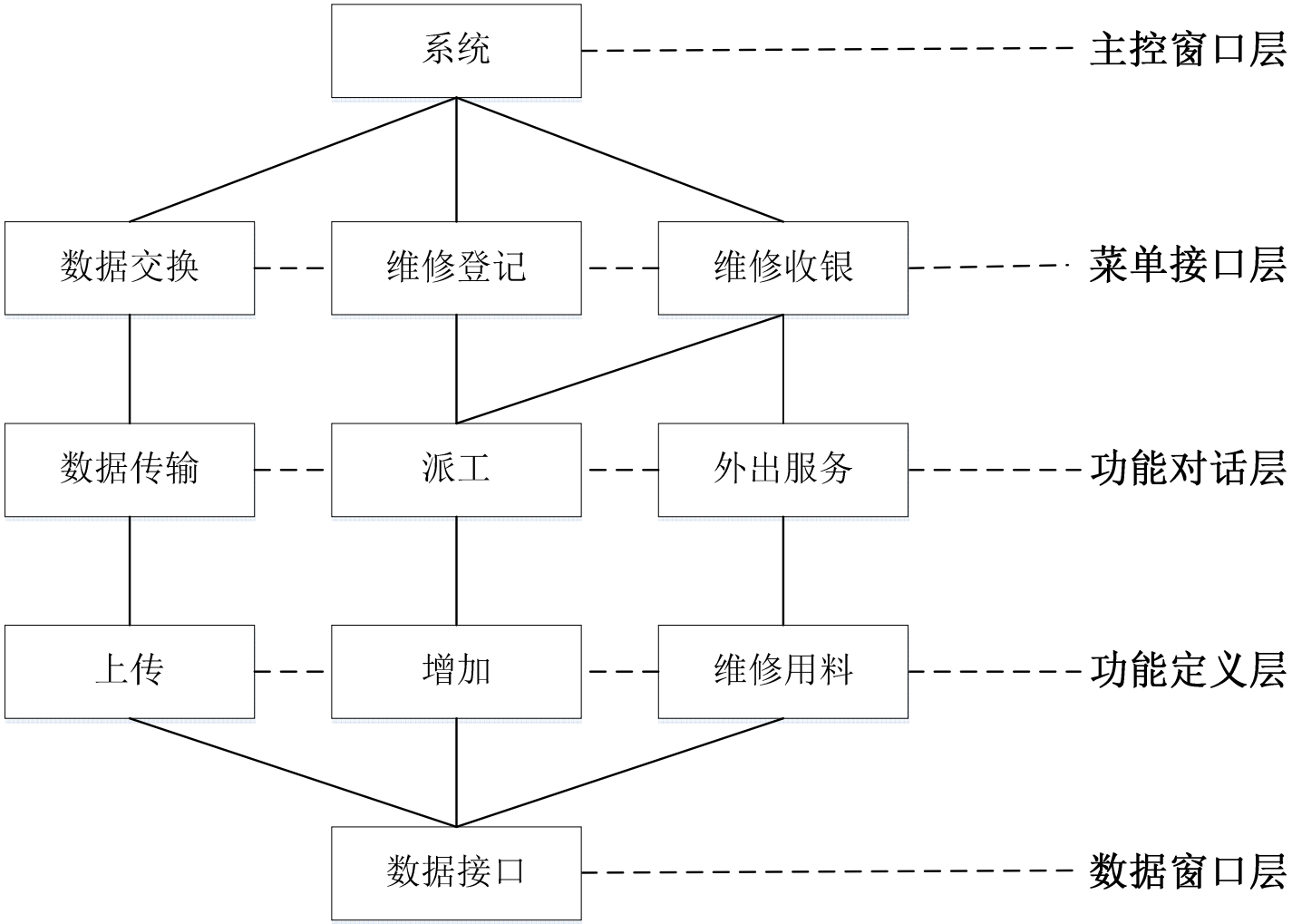


图4.44汽修服务管理系统的框架结构

4.3.18 异构风格

- 在设计软件系统时，从不同角度来观察和思考问题，会对架构风格的选择产生影响。
- 每一种架构风格都有不同的特点，适用于不同的应用问题，因此，架构风格的选择是多样化的和复杂的。
- 在实际应用中，各种软件架构并不是独立存在的，在一个系统中，往往会有多种架构共存和相互融合，形成更复杂的框架结构，即异构架构。

4.3.18 异构风格

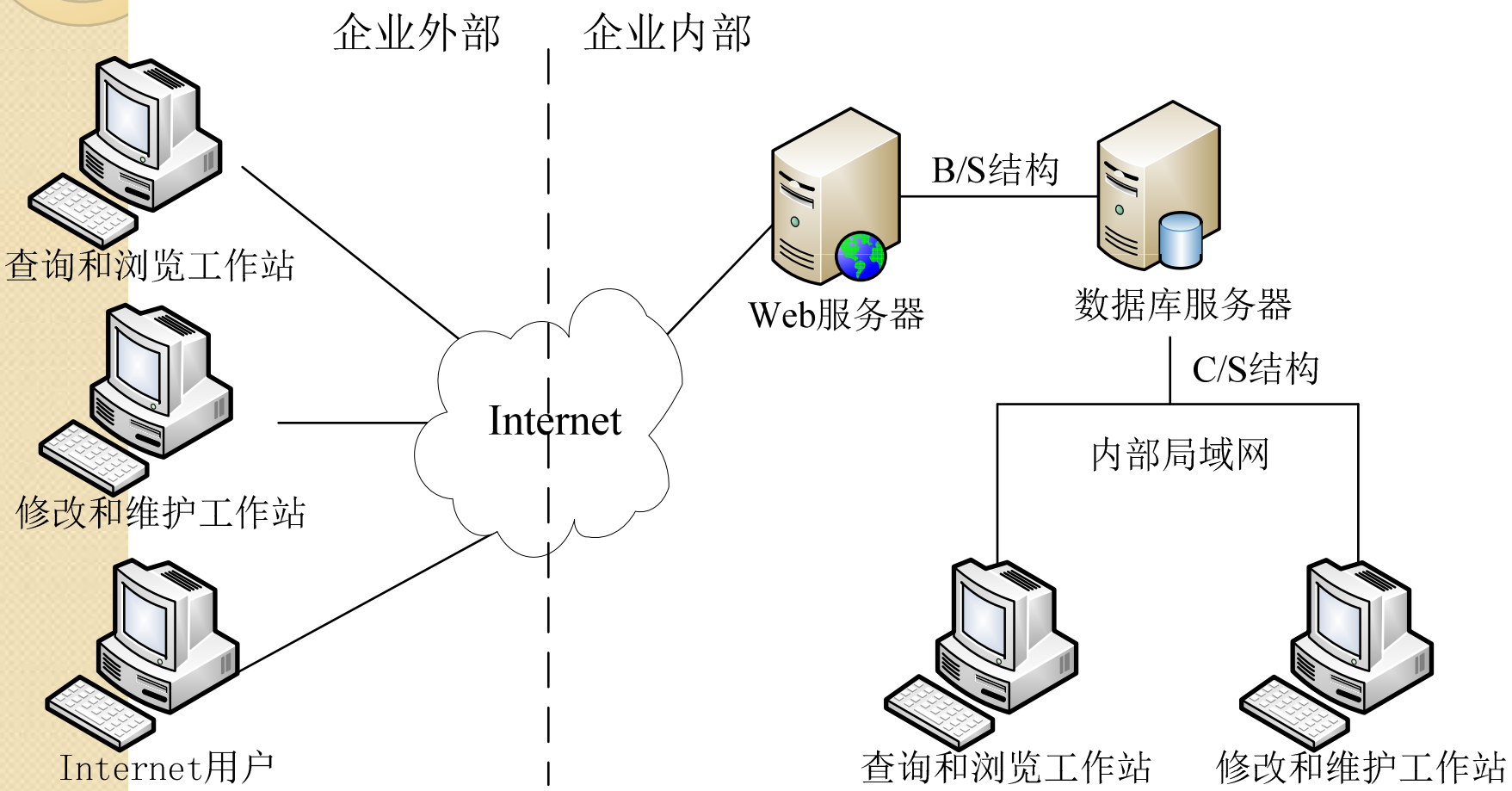
- 异构架构是几种风格的组合，组合方式可能有如下几种：
  - （1）使用层次结构。一个系统组件被组织成某种架构风格，但它的内部结构可能是另一种完全不同的风格。
  - （2）允许单一组件使用复合的连接件。

4.3.18 异构风格

- 优点
  - （1）选择异构架构风格，可以实现遗留代码的重用。
  - （2）在某一单位中，规定了共享软件包和某些标准，但仍会存在解释和表示习惯上的不同。选择异构架构风格，可以解决这一问题。
- 缺点
  - 不同风格之间的兼容问题有时很难解决

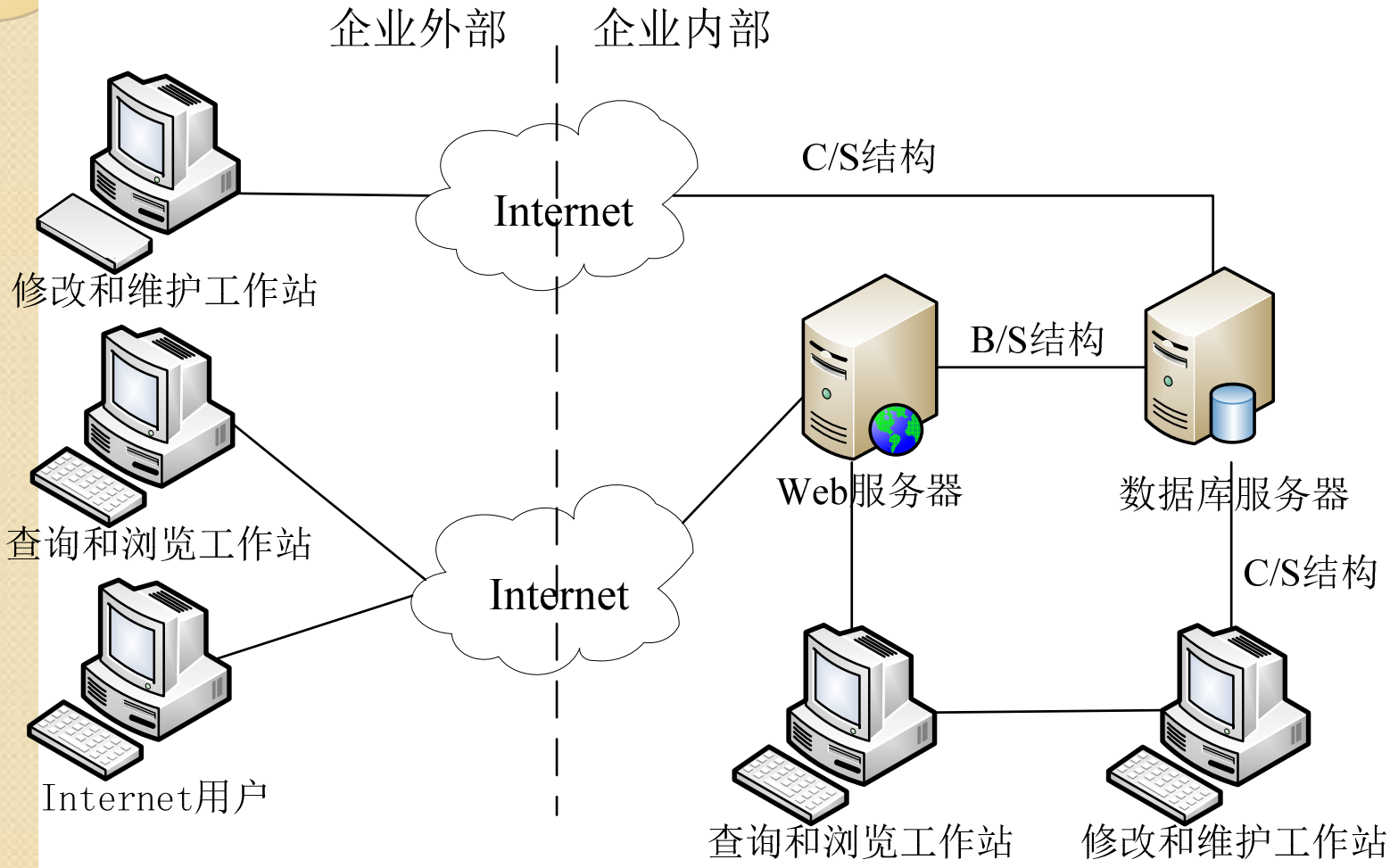
4.3.18 异构风格

- 应用实例：B/S结构和C/S结构组合——“内外有别”模型



4.3.18 异构风格

- 应用实例：B/S结构和C/S结构组合——“查改有别”模型



4.3.18 异构风格

- 应用实例：供电管理系统——B/S结构和C/S结构组合

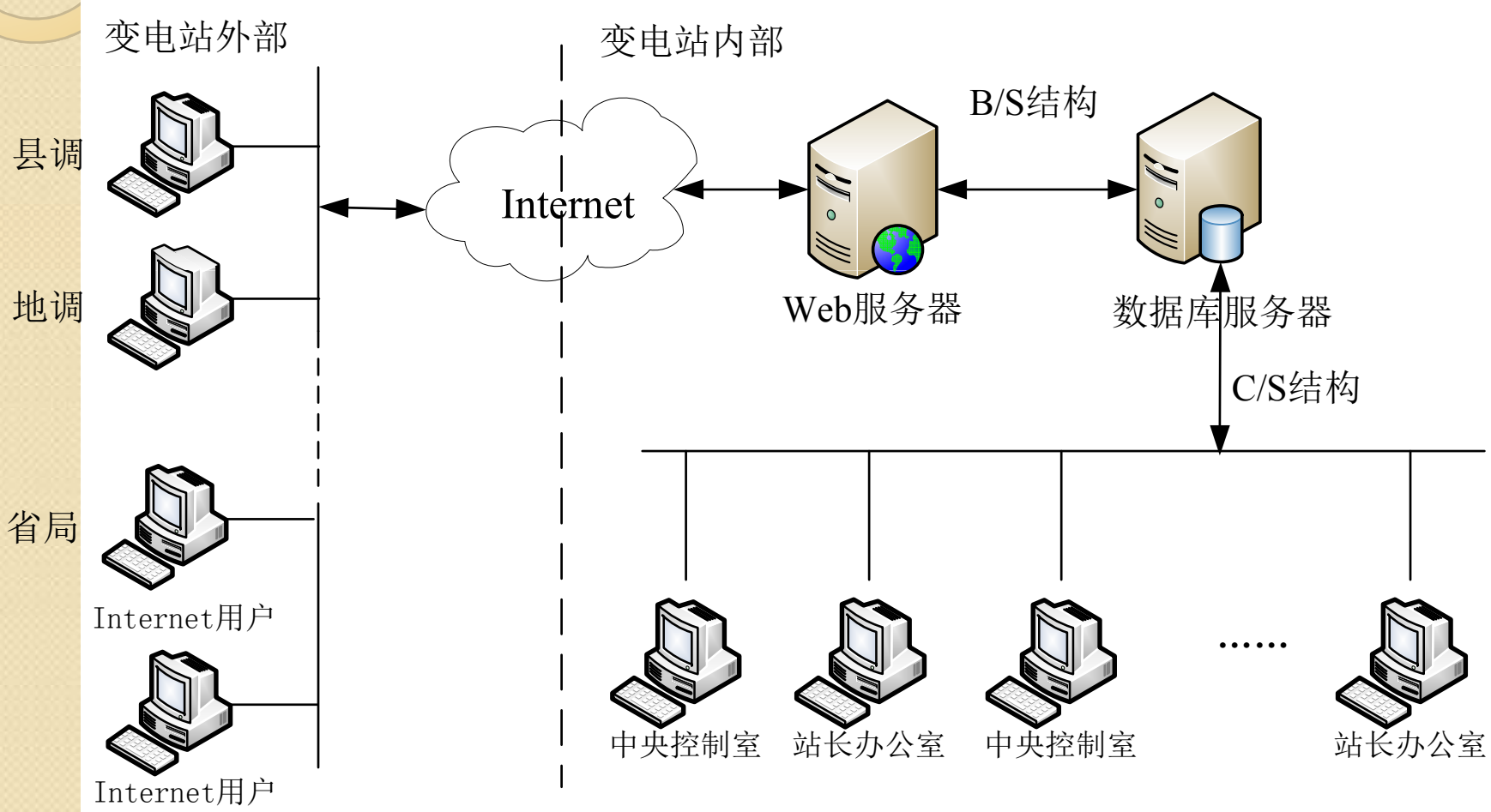


图4.47供电管理系统的架构

4.3.19 基于层次消息总线的架构风格（JB/HMB风格）

- 北大杨芙清院士等提出；
- 以青鸟软件生产线的实践为背景，提出了基于层次消息总线的软件架构(Jade bird hierarchical message bus based style)

4.3.19 基于层次消息总线的架构风格（JB/HMB风格）

- 背景：
  - 网络和分布式构件技术日趋成熟，具有分布和并发特点的软件系统成为普遍需求;
  - 基于事件驱动的编程模式在图形用户界面程序设计中得到广泛应用;
  - 硬件总线体系结构具有良好的扩展性和适应性。

4.3.19 基于层次消息总线的架构风格（JB/HMB风格）

- JB/HMB风格基于层次消息总线、支持组件的分布和并发，组件之间通过消息总线进行通讯。
- 消息总线是系统的连接件，负责消息的分派、传递和过滤以及处理结果的返回。
- 各个组件挂接在消息总线上，向总线登记感兴趣的消息类型。
- 不要求各个构件具有相同的地址空间或局限在同台机器上，可较好地描述分布式并发系统。

4.3.19 基于层次消息总线的架构风格（JB/HMB风格）

- 基本思想

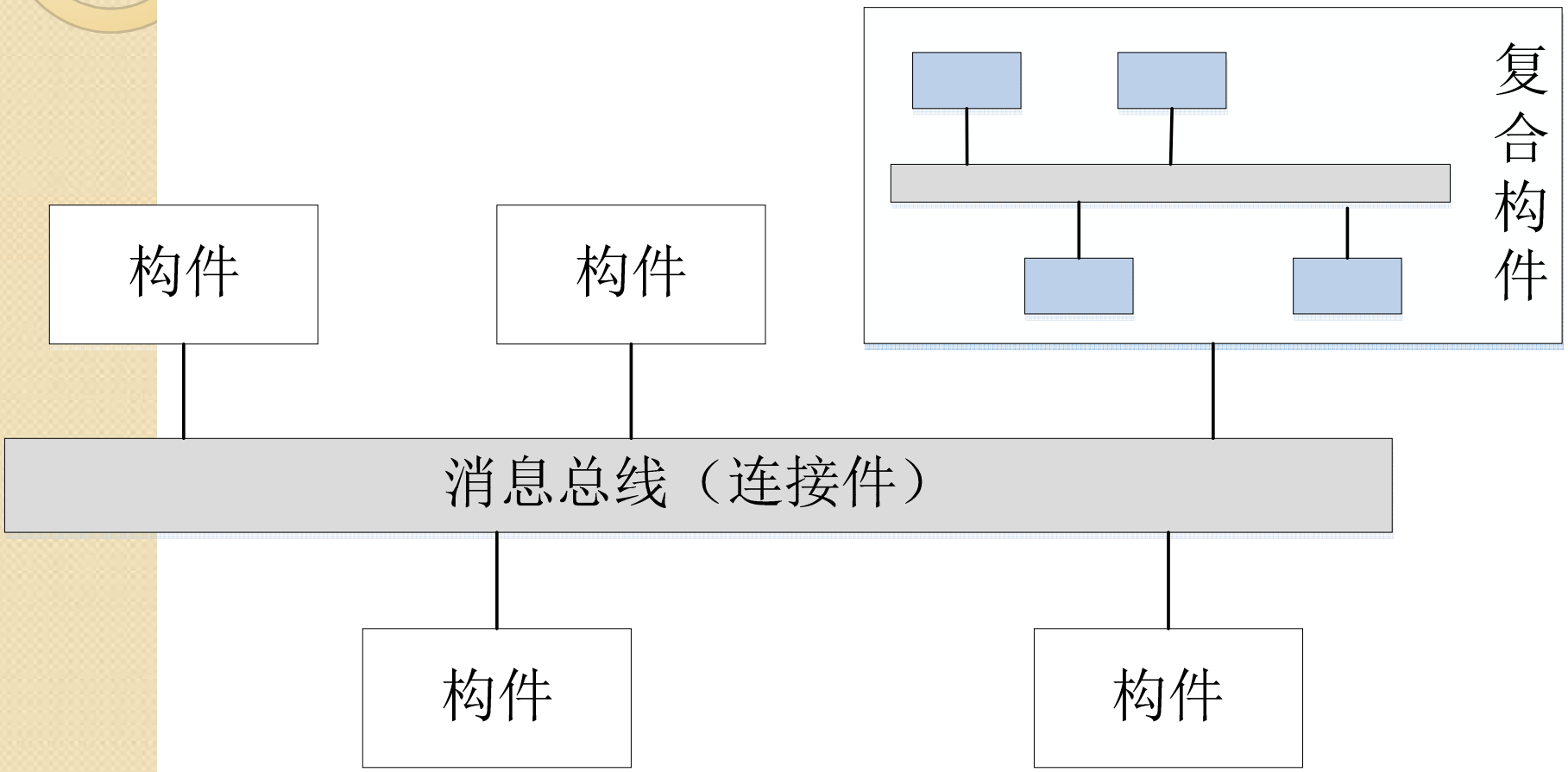


图4.48基于层次消息总线的架构风格

4.3.19 基于层次消息总线的架构风格

- 优点
  - （1）JB/HMB风格的构件接口是一种基于消息的互联接口，可以较好地支持架构设计。降低了构件之间的耦合性，增强了构件的重用性。
  - （2）JB/HMB风格支持运行时系统演化，主要体现在可动态增加和删除构件，动态改变构件所响应的消息以及消息过滤这三个方面。
- 缺点是重用要求高，可重用性差。

4.3.19 基于层次消息总线的架构风格

- 应用实例：青鸟工程软件开发过程

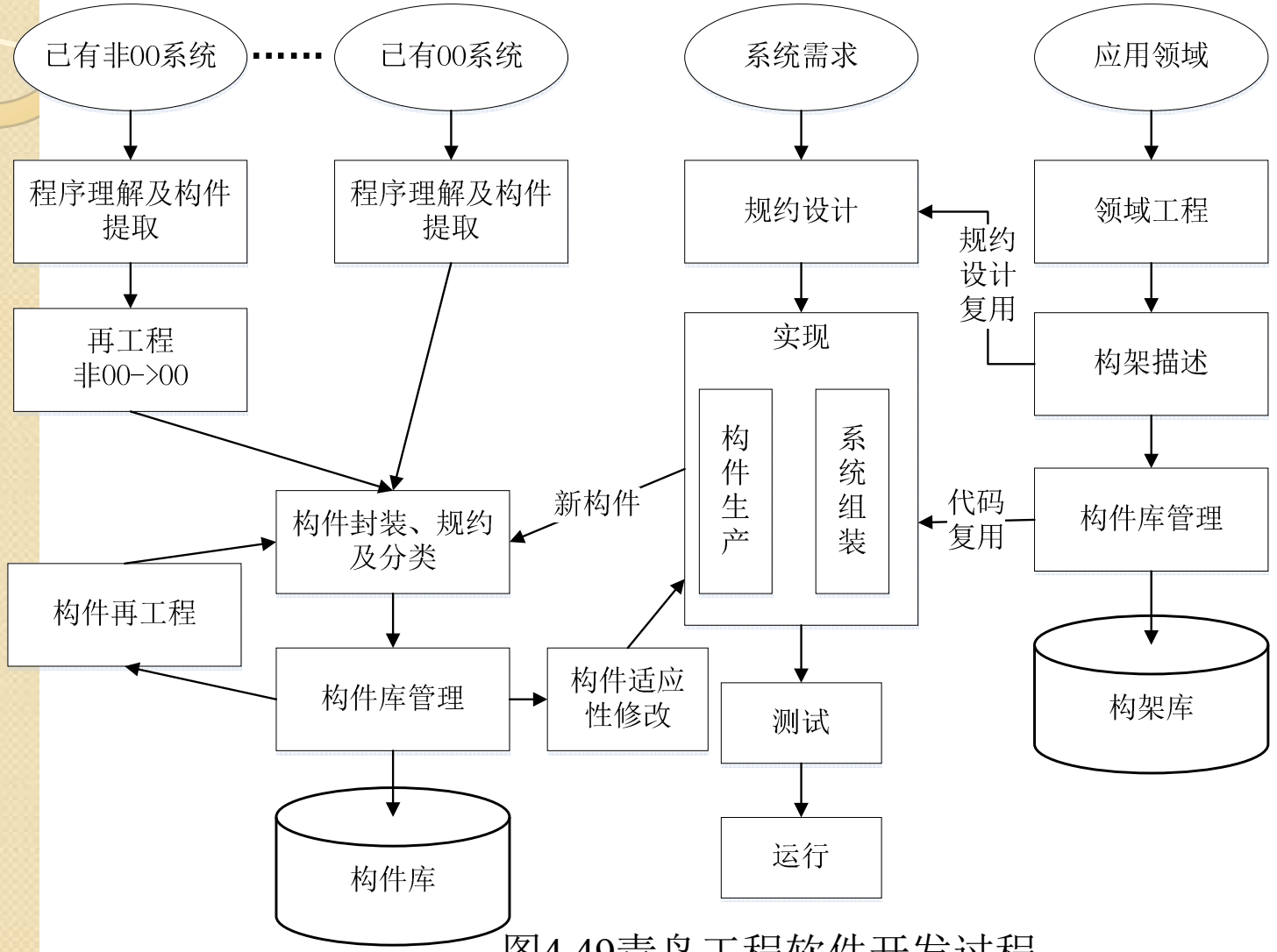


图4.49青鸟工程软件开发过程

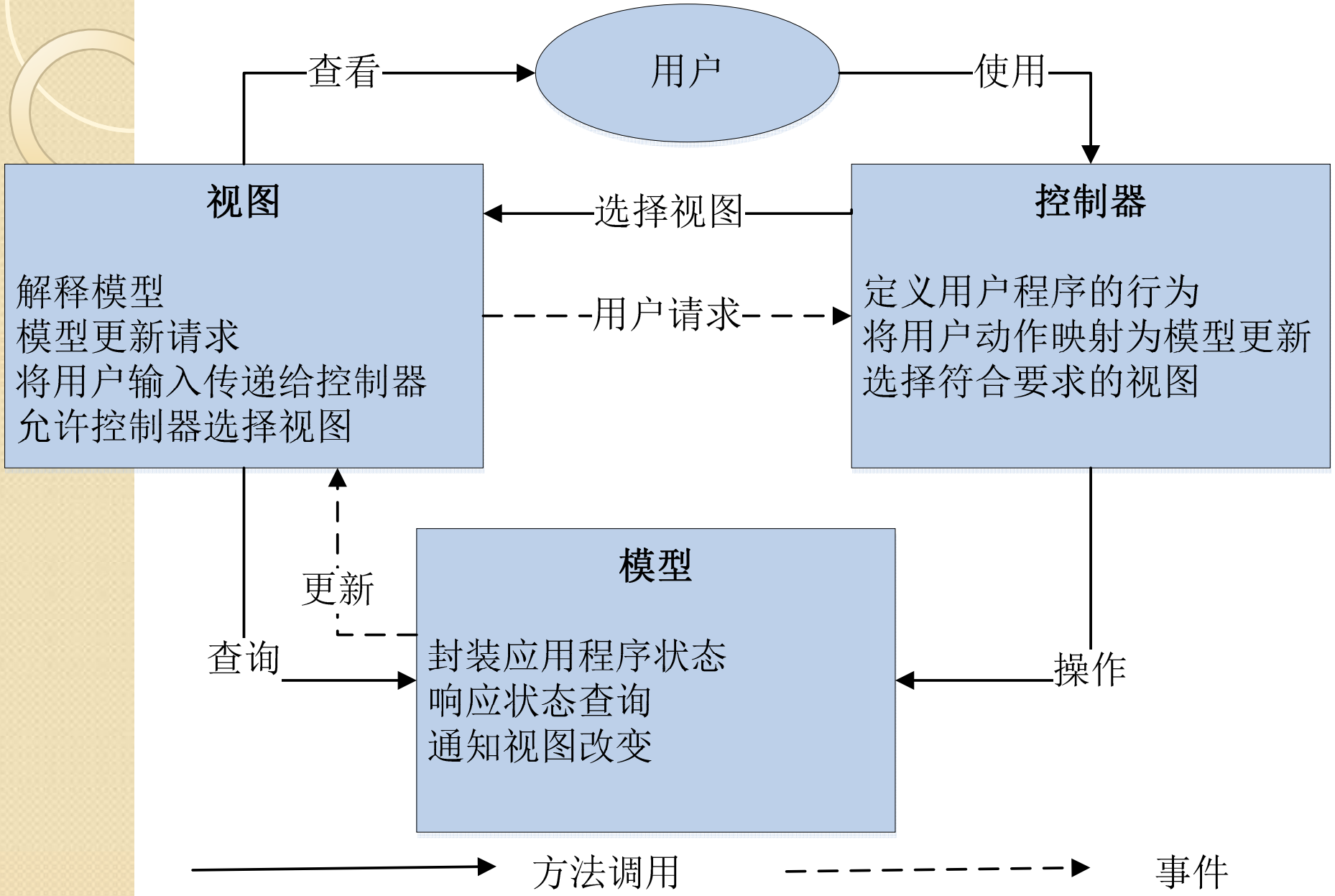
4.3.20 模型-视图-控制器风格

- 模型-视图-控制器风格（Model-View-Controller，MVC）主要是针对编程语言Smalltalk 80所提出的一种软件设计模式。
- MVC被广泛的应用于用户交互程序的设计中。

4.3.20 模型-视图-控制器风格

- MVC结构主要包括模型、视图和控制器三部分。
  - （1）模型(Model, M)：模型是应用程序的核心，它封装了问题的核心数据、逻辑关系和计算功能，提供了处理问题的操作过程。
  - （2）视图(View,V)：视图是模型的表示，提供了交互界面，为用户显示模型信息。
  - （3）控制器(Controller, C)：控制器负责处理用户与系统之间的交互，为用户提供操作接口。

4.3.20 模型-视图-控制器风格



4.3.20 模型-视图-控制器风格

- 优点
  - （1）多个视图与一个模型相对应。变化——传播机制确保了所有相关视图都能够及时地获取模型变化信息，从而使所有视图和控制器同步，便于维护。
  - （2）具有良好的移植性。由于模型独立于视图，因此可以方便的实现不同部分的移植。
  - （3）系统被分割为三个独立的部分，当功能发生变化时，改变其中的一个部分就能满足要求。

4.3.20 模型-视图-控制器风格

- 缺点
  - （1）增加了系统设计和运行复杂性。
  - （2）视图与控制器连接过于紧密，妨碍了二者的独立重用。
  - （3）视图访问模型的效率比较低。由于模型具有不同的操作接口，因此视图需要多次访问模型才能获得足够的数据。
  - （4）频繁访问未变化的数据，也将降低系统的性能。

4.3.20 模型-视图-控制器风格

- 应用实例
  - 目前比较好的MVC
    - 老牌的有Struts，Webwork。
    - 新兴的MVC框架有Spring MVC，Tapestry，JSF等。
    - 还有，如Dinamica，VRaptor等
  - 这些框架都提供了较好的层次分隔能力。在实现良好的MVC分隔的基础上，提供一些现成的辅助类库，同时也促进了生产效率的提高。

4.4 软件架构模式

- Dwayne E. Perry和Alexander L. Wolf从组件的角度给出了软件架构模式的定义：根据系统的结构组织定义了软件系统族，以及构成系统族的组件之间的关系。

4.4 软件架构模式

- 在一般意义上，大多数人认为模式即为风格
- Frank Buschmann 认为体系结构的模式与风格是有区别的，它们的区别和联系主要在于：
  - （1）体系结构风格主要描述应用系统的总体结构框架。
  - （2）架构风格相对独立。
  - （3）模式比架构风格更加面向问题。

4.4 软件架构模式

- 架构风格与模式的区别
  - 软件架构风格是对软件架构整体方案的展现形式，反映的是整体方案实施之后的效果；
  - 软件架构模式仍然是软件设计模式的一种，也是一种问题解决方案对（problem-solution pair），反映的是针对某个需求问题，至今为止最佳的解决方案是什么。
  - 虽然软件架构风格也与解决方案有关，但它并没有强调一定是最佳方案，而架构模式具有这种潜在要求。