



●第4章 软件架构的风格与模式

王璐璐 wanglulu@seu.edu.cn

廖力 lliao@seu.edu.cn

第4章 软件架构的风格与模式

- 4.0 引言
- 4.1 软件架构风格定义
- 4.2 软件架构风格的分类
- 4.3 典型的软件架构风格
- 4.4 软件架构模式

4.0 引言

- 软件架构的风格 VS 模式
 - 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，至今为止的**最佳解决方案**。
- 本章从软件架构风格的基本思想、优缺点和应用实例三个方面介绍20种最典型的架构风格。

4.1 软件架构风格的定义

- 软件架构风格 (software architecture style) , 又称软件架构惯用模式 (software architecture idiomatic paradigm)
- 是描述**某一特定应用领域**中系统组织方式的**惯用模式**, 作为“**可复用的组织模式和习语**”, 为设计人员的交流提供了公共的术语空间, 促进了设计复用与代码复用。

4.1 软件架构风格的定义

- 架构风格的基本属性
 - 设计元素的词汇表（包括组件、连接件的类型以及数据元素，例如：管道、过滤器、对象、服务等）
 - 配置规则：决定元素组合的拓扑约束
 - 例如：限制某一风格中的组件至多与其它两个组件相连。
 - 元素组合的语义解释以及使用某种风格构建的系统的分析

4.1 软件架构风格的定义

- 使用架构风格的优势
 - 可以极大地促进设计的重用性和代码的**重用性**，并且使得系统的组织结构**易被理解**。
 - 即使没有给出具体实现细节，依然可以通过“客户/服务器”架构风格大致推测出系统的组成结构和工作方式。
 - 使用标准的架构风格可较好地支持系统内部的**互操作性**以及**针对特定风格的分析**
 - 如：“管道/过滤器”风格可用来分析调度、吞吐量、延迟，和死锁等问题。

4.2 软件架构风格的分类

- (1) 数据流风格：批处理序列、管道/过滤器；
- (2) 调用/返回风格：主程序/子程序、面向对象风格、层次结构；
- (3) 独立组件风格：进程通讯、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、黑板系统等。

4.2 软件架构风格的分类

- 根据上述架构风格分类发现，架构风格是基于不同的视角或层次抽象出来的。不同风格之间常常具有交叉现象。
 - 主程序和子程序风格以及面向对象风格几乎是目前一切软件构件、连接件的设计和实现基础；
 - 分层系统的层次结构风格几乎是一切复杂系统的基本结构方法；
 - 事件系统是一种受操作系统管理控制的部件连接方式；
 - 仓库是部件的设计结构；
 - 管道/过滤器、客户/服务器、解释器等则是特殊的系统结构方法。
 - 即：对于同一种软件架构，从不同的视角和层次可能抽象出不同的架构风格。

4.3 典型的软件架构风格

1. 管道过滤器风格
2. 主程序/子程序风格
3. 面向对象风格
4. 层次化风格
5. 事件驱动风格
6. 解释器风格
7. 基于规则的系统风格
8. 仓库风格
9. 黑板系统风格
10. C2风格
11. 客户机/服务器风格
12. 浏览器/服务器风格
13. 平台/插件风格
14. 面向Agent风格
15. 面向方面软件架构风格
16. 面向服务架构风格
17. 正交架构风格
18. 异构风格
19. 基于层次消息总线的架构风格
20. 模型-视图-控制器风格

4.3.1 管道过滤器风格

- 基本思想
 - 在管道过滤器风格下，每个功能模块都有一组输入和输出。
 - 功能模块称作**过滤器 (filters)**；功能模块间的连接可以看作输入、输出数据流之间的通路，所以称作**管道 (pipes)**。
 - 管道-过滤器风格的特性之一在于**过滤器的相对独立性**，即过滤器独立完成自身功能，相互之间无需进行状态交互。

4.3.1 管道-过滤器风格

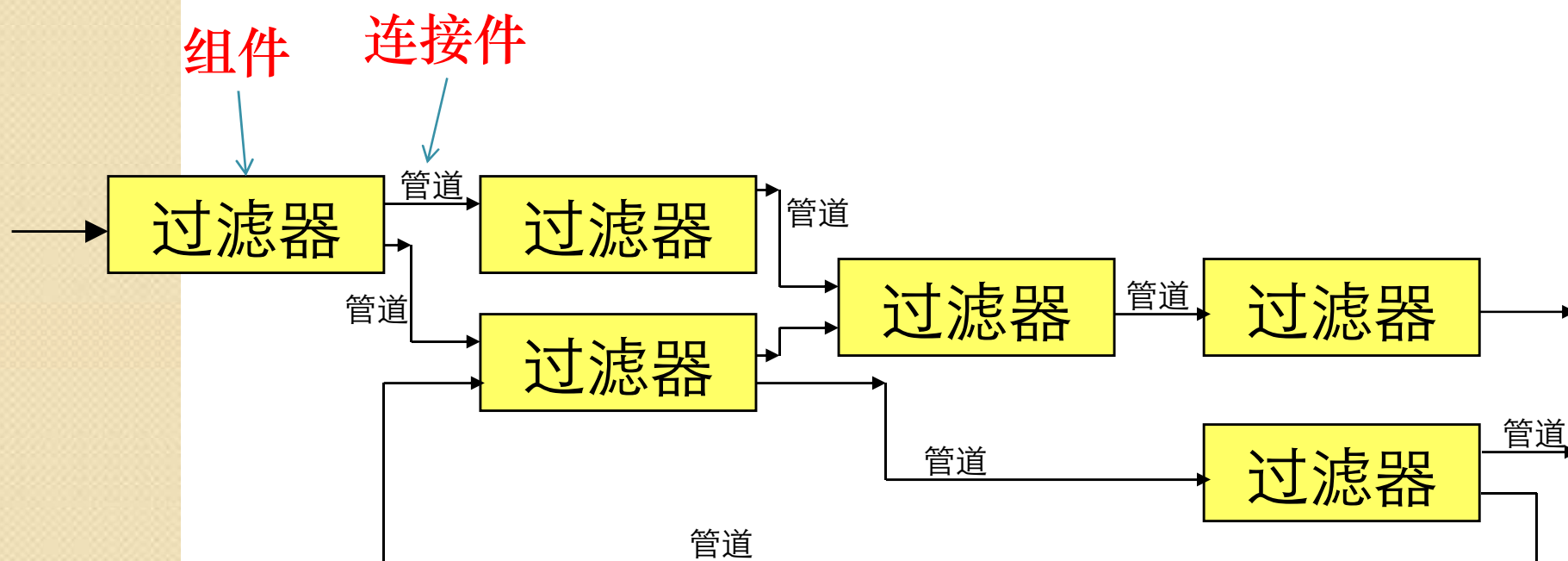


图4.1 管道 - 过滤器风格的体系结构

4.3.1 管道-过滤器风格

- 过滤器是独立运行的组件
 - 非临近的过滤器之间不共享状态
 - 过滤器自身无状态
- 过滤器对其处理上下连接的过滤器“无知”
 - 对相邻的过滤器不施加任何限制
- 结果的正确性不依赖于各个过滤器运行的先后次序
 - 各过滤器在输入具备后完成自己的计算。完整的计算过程包含在过滤器之间的拓扑结构中。

从解耦的角度看管道过滤器风格

- 耦合度表示模块之间（例如：类与类之间、子程序与子程序之间）关系的紧密程度，“低耦合度”是软件设计的目标。
- 低耦合模块（类或子程序）之间的关系尽可能简单，彼此之间的相互依赖性小，也就是“松散耦合”。
- **解耦（decouple）** 就是降低模块之间的耦合度，也就是尽可能使得模块之间的耦合是松散耦合。
- 解耦能够保持组件之间的自主和独立性。它的直接结果就是改动成本低，维护成本低，可读性高。

从解耦的角度看管道过滤器风格

- 引入管道过滤器的一个好处是它可以使得每个过滤器之间都是解耦的，每个过滤器都专注于自己的职责。
 - 过滤器是独立运行的组件
 - 过滤器对其处理上下连接的过滤器“无知”
 - 结果的正确性不依赖于各个过滤器运行的先后次序

管道-过滤器风格优点

- (1) 由于每个组件行为不受其他组件的影响，整个系统的行为易于理解。
 - 使得软件组件具有良好的隐蔽性和高内聚、低耦合的特点，允许设计者将整个系统的输入/输出行为看成是多个过滤器的行为的简单合成；

管道-过滤器风格优点

- (2) 管道-过滤器风格支持功能模块的复用
 - 任何两个过滤器，只要它们之间传送的数据遵守共同的规约，就可以相连接。每个过滤器都有自己独立的输入输出接口，如果过滤器间传输的数据遵守其规约，只要用管道将它们连接就可以正常工作。

管道-过滤器风格优点

- (3) 基于管道-过滤器风格的系统具有较强的可维护性和可扩展性。
 - 旧的过滤器可以被替代，新的过滤器可以添加到已有的系统上。软件的易于维护和升级是衡量软件系统质量的重要指标之一，在管道-过滤器模型中，只要遵守输入输出数据规约，任何一个过滤器都可以被另一个新的过滤器代替，同时为增强程序功能，可以添加新的过滤器。这样，系统的可维护性和可升级性得到了保证。

管道-过滤器风格优点

- (4) 支持一些特定的分析，如吞吐量计算和死锁检测等。
 - 利用管道-过滤器风格的视图，可以很容易得到系统的资源使用和请求的状态图。然后，根据操作系统原理等相关理论中的死锁检测方法就可以分析出系统目前所处的状态，是否存在死锁可能及如何消除死锁等问题。

管道-过滤器风格优点

- (5) 管道-过滤器风格具有并发性
 - 每个过滤器作为一个单独的执行任务，可以与其它过滤器并发执行。过滤器的执行是独立的，不依赖于其它过滤器的。在实际运行时，可以将存在并发可能的多个过滤器看作多个并发的任务并行执行，从而大大提高系统的整体效率，加快处理速度。

管道-过滤器风格不足

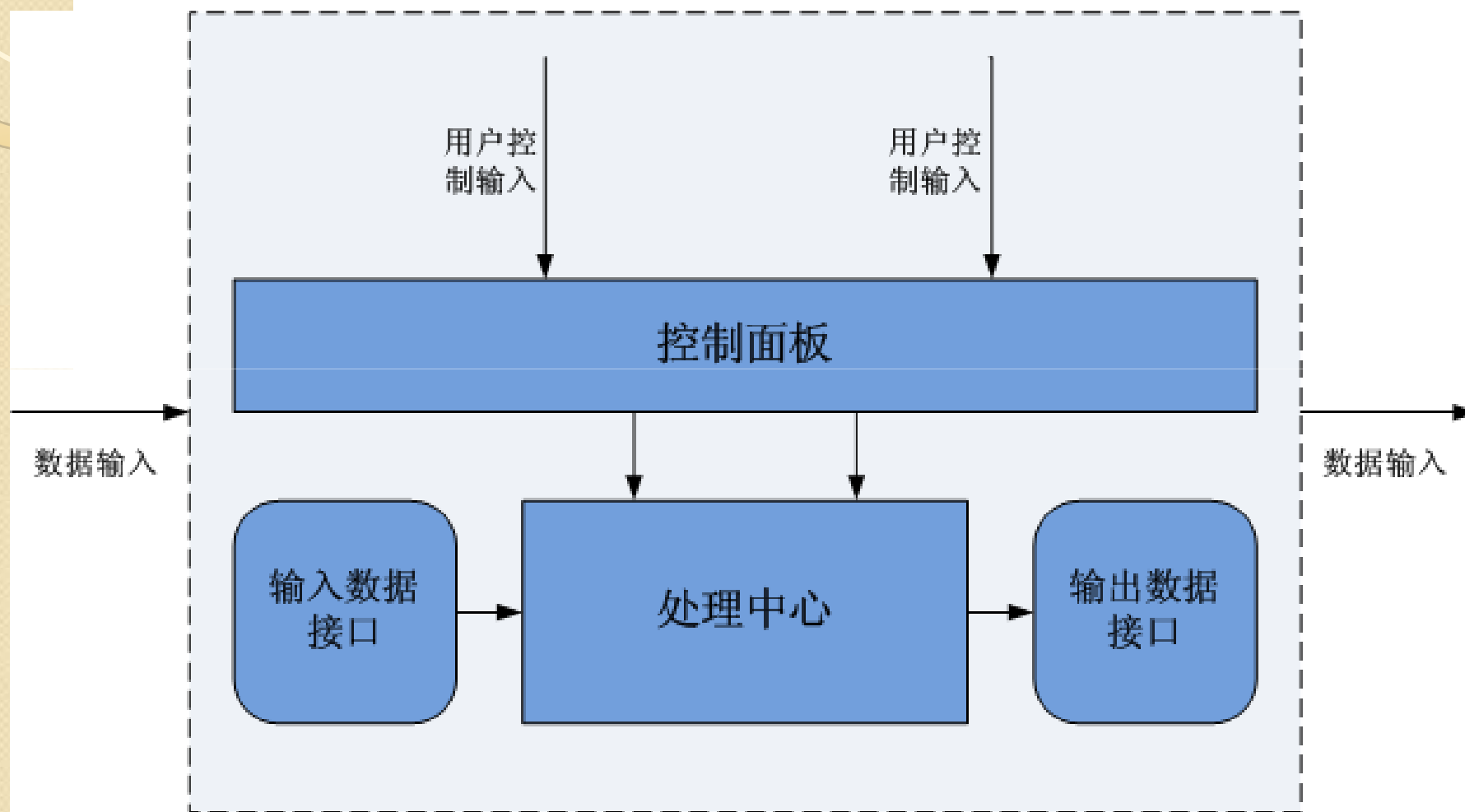
- (1) 管道-过滤器风格往往导致系统处理过程的成批操作。
- (2) 根据实际设计的需要，设计者也需要对数据传输进行特定的处理（如为了防止数据泄漏而采取加密等手段，或者使用了底层公共命名），导致过滤器必须对输入、输出管道中的数据流进行解析或反解析，增加了过滤器具体实现的复杂性。

管道-过滤器风格不足

- (3) 交互式处理能力弱

- 管道-过滤器模型适于数据流的处理和变换，不适合为与用户交互频繁的系统建模。在这种模型中，每个过滤器都有自己的数据，这些数据或者是从磁盘存储器中读取来，或者是由另一个过滤器的输出导入进来，或整个系统没有一个共享的数据区。这样，当用户要操作某一项数据时，要涉及到多个过滤器对相应数据的操作，其实现较为复杂。由以上的缺点，可以对每个过滤器增加相应的用户控制接口，使得外部可以对过滤器的执行进行控制。

管道-过滤器风格不足



改进的过滤器

管道-过滤器风格实例

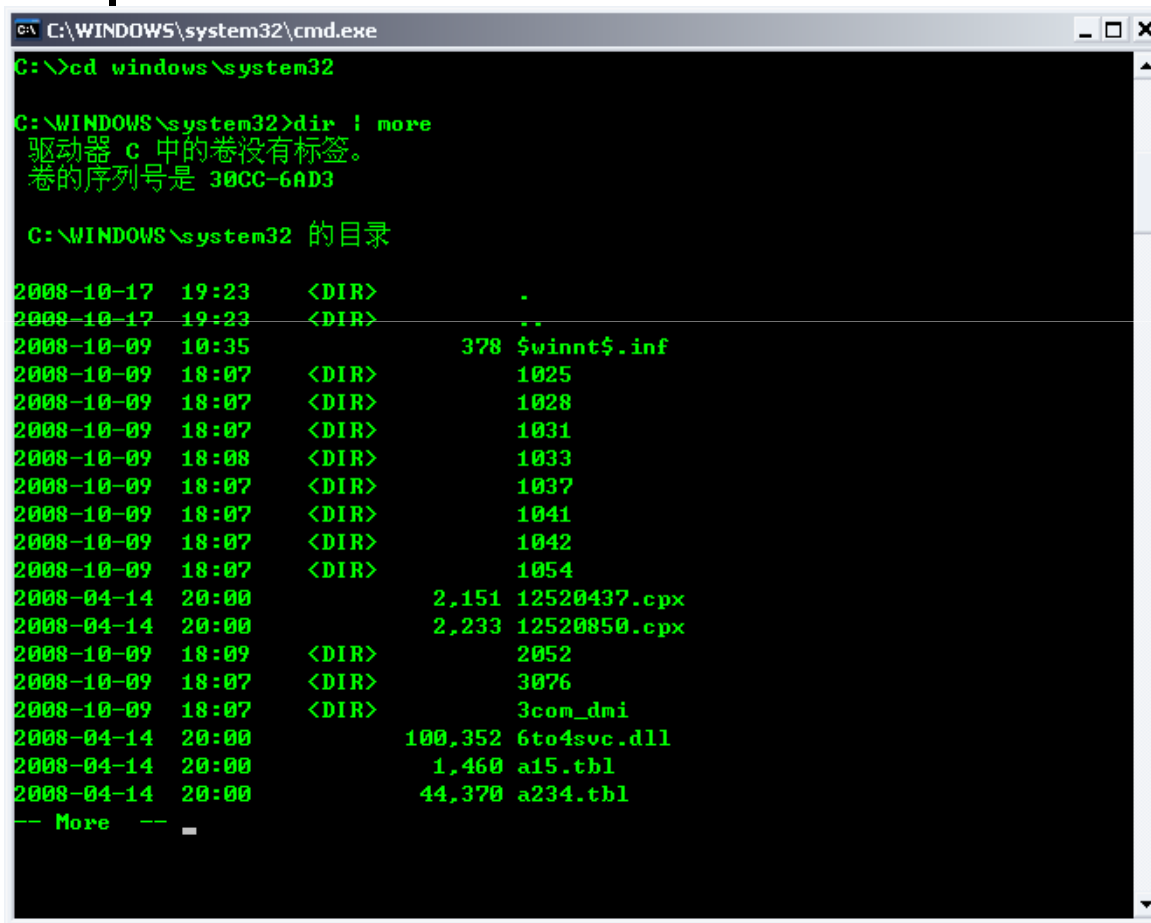
- 在Unix和Dos中，允许在命令中出现用竖线字符“|”分开的多个命令，将符号“|”之前的命令的输出，作为“|”之后命令的输入，这就是“管道功能”，竖线字符“|”是管道操作符。
- 例如： Unix shell中：
 - `cat file1|grep xyz|sort|uniq>out`
 - 即连接文件file1，找到含xyz的行，排序、去掉相同的行，最后输出

管道-过滤器风格实例

- 例如：DOS 中：
 - 命令 `dir | more` 使得当前目录列表在屏幕上逐屏显示。
 - `dir` 的输出是整个目录列表，它不出现在屏幕上而是由于符号 “|” 的规定，成为下一个命令 `more` 的输入，`more` 命令则将其输入，`more` 命令则将其输入一屏一屏地显示，成为命令行的输出。

管道-过滤器风格 实例

- dir | more



```
C:\WINDOWS\system32\cmd.exe
C:\>cd windows\system32

C:\WINDOWS\system32>dir | more
驱动器 C 中的卷没有标签。
卷的序列号是 30CC-6AD3

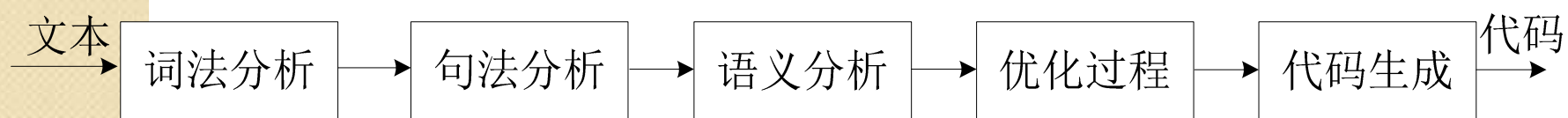
C:\WINDOWS\system32 的目录

2008-10-17  19:23    <DIR>          .
2008-10-17  19:23    <DIR>          ..
2008-10-09  10:35             378 $winnt$.inf
2008-10-09  18:07    <DIR>          1025
2008-10-09  18:07    <DIR>          1028
2008-10-09  18:07    <DIR>          1031
2008-10-09  18:08    <DIR>          1033
2008-10-09  18:07    <DIR>          1037
2008-10-09  18:07    <DIR>          1041
2008-10-09  18:07    <DIR>          1042
2008-10-09  18:07    <DIR>          1054
2008-04-14  20:00           2,151 12520437.cpx
2008-04-14  20:00           2,233 12520850.cpx
2008-10-09  18:09    <DIR>          2052
2008-10-09  18:07    <DIR>          3076
2008-10-09  18:07    <DIR>          3com_dni
2008-04-14  20:00          100,352 6to4svc.dll
2008-04-14  20:00           1,460 a15.tbl
2008-04-14  20:00          44,370 a234.tbl
-- More --
```

管道-过滤器风格实例

- 实例2:

- 传统的编译器，一个阶段的输出是另一个阶段的输入（包括词法分析、语法分析、语义分析和代码生成）

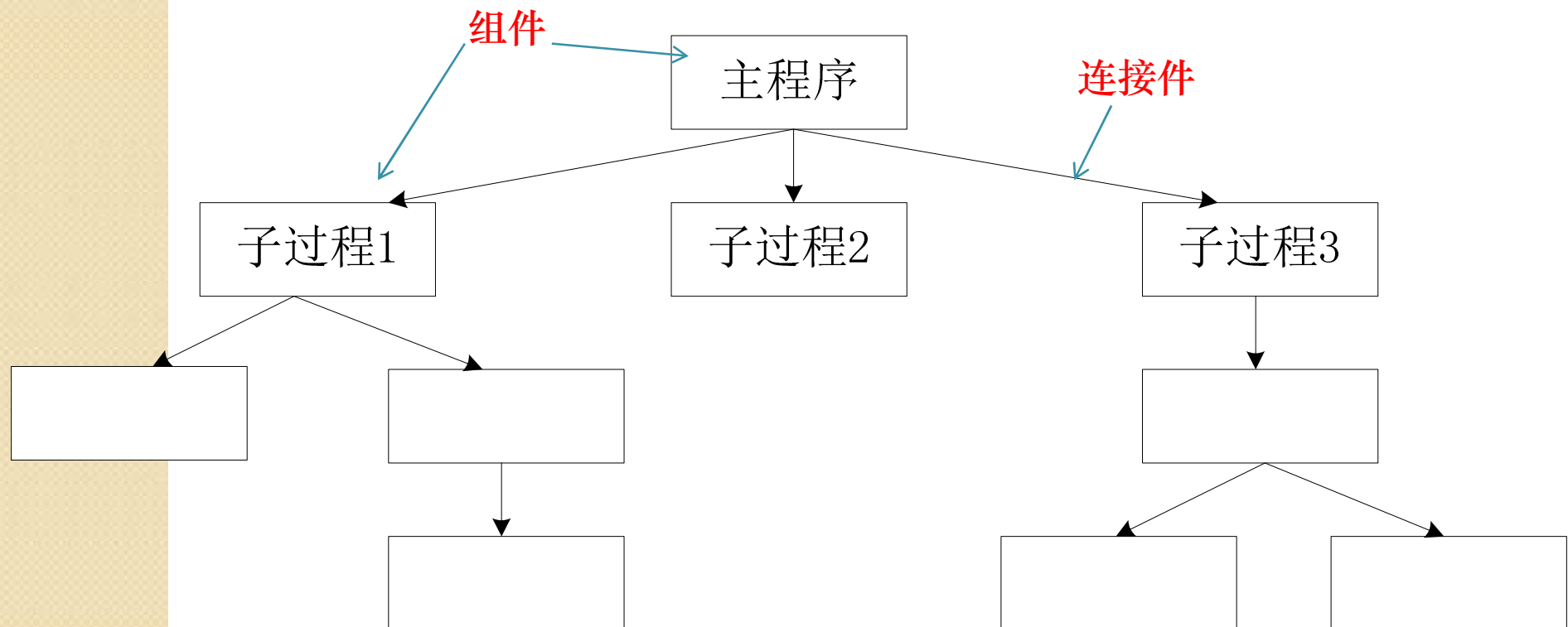


4.3.2 主程序/子程序风格

- 20世纪70年代，结构化程序设计方式出现，主程序/子程序结构(Main Program/Subroutines) 成为结构化设计的一种典型风格。
 - 该架构风格从功能的观点设计系统
 - 通过逐步分解和逐步细化得到系统架构，即将大系统分解为若干模块(模块化)，主程序调用这些模块实现完整的系统功能
 - 主程序的正确性依赖于它所调用的子程序的正确性。

4.3.2 主程序/子程序风格

- 其中，组件为主程序和子程序，连接件为调用-返回机制，拓扑结构为层次化结构。



4.3.2 主程序/子程序风格

- 优缺点分析

- 优点:

- (1) 具有很高的数据访问效率，因为计算共享同一个存储区。
 - (2) 不同的计算功能被划分在不同的模块中。

4.3.2 主程序/子程序风格

- 优缺点分析

- 缺点：该方案在处理变更的能力上有许多严重的缺陷。
- (1) 对数据存储格式的变化将会影响几乎所有的模块。
- (2) 对处理流程的改变与系统功能的增强也很难适应，依赖于控制模块内部的调用次序。
- (3) 这种分解方案难以支持有效的复用。

4.3.2 主程序/子程序风格

- 应用实例

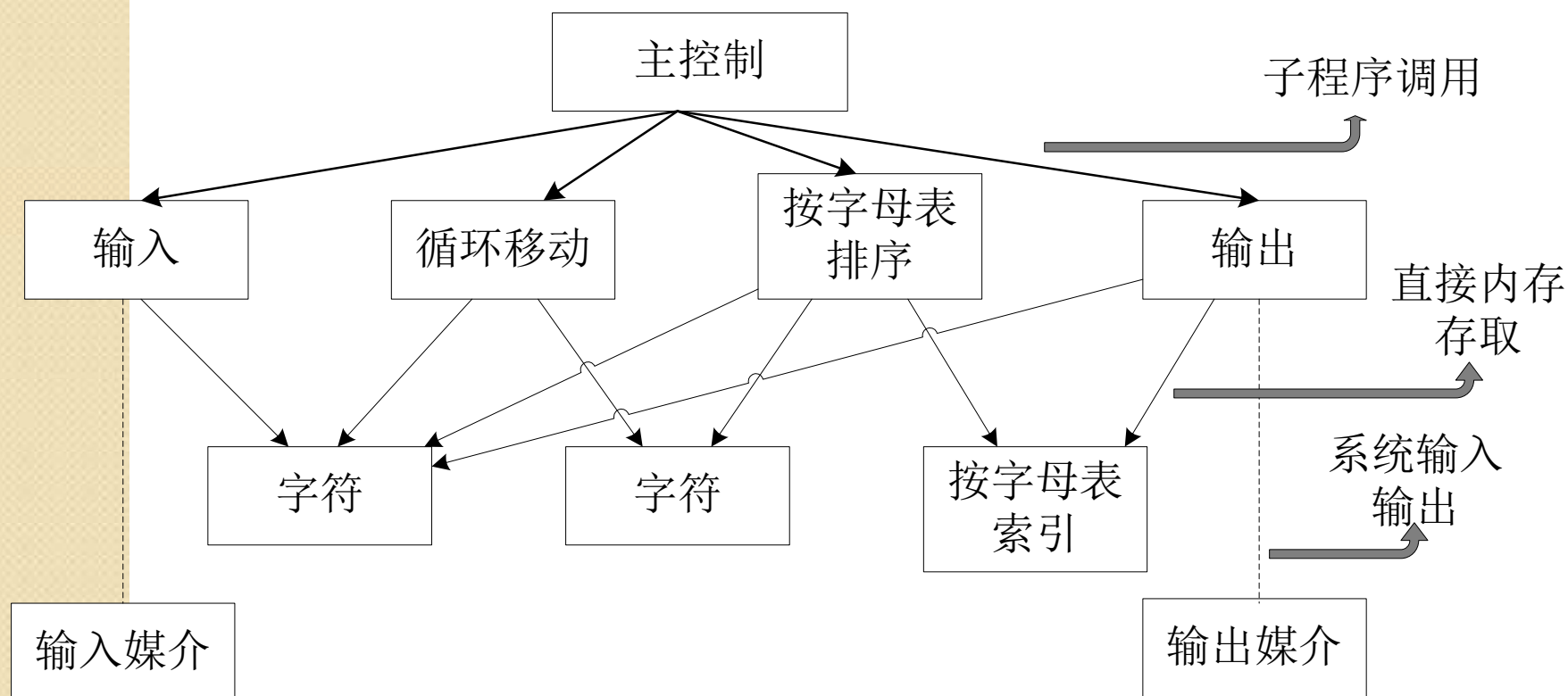


图4.6 KWIC (Key Word in Context) 解决方案

作业

- 思考题：

- KWIC还可以采用哪些风格？分析各自的架构特点和运行特点。

4.3.3 面向对象风格

- 面向对象（Object-Oriented）方法是80年代初期提出的一种新型的程序设计方法，它彻底改变了过去数据流、事物流分析方式的缺点，采用直接对问题域进行自然抽象的方法，并逐渐发展成包括面向对象分析、设计、编程、测试、维护等一整套内容的完整体系。
- 该架构风格从现实世界中客观存在的事物出发，强调直接以问题域中的事物为中心来思考问题、认识问题，根据这些事物的本质特征，将其抽象为系统中的对象，并作为系统中的基本构成单位。

4.3.3 面向对象风格

- 面向对象组织结构有两个重要特点：
 - (1)对象负责维护其表示的完整性（通常是通过保持其表示上的一些不变式来实现的）；
 - (2)对象的表示对其他对象而言是隐蔽的。抽象数据类型的使用，以及面向对象系统的使用已经非常普遍。

4.3.3 面向对象风格

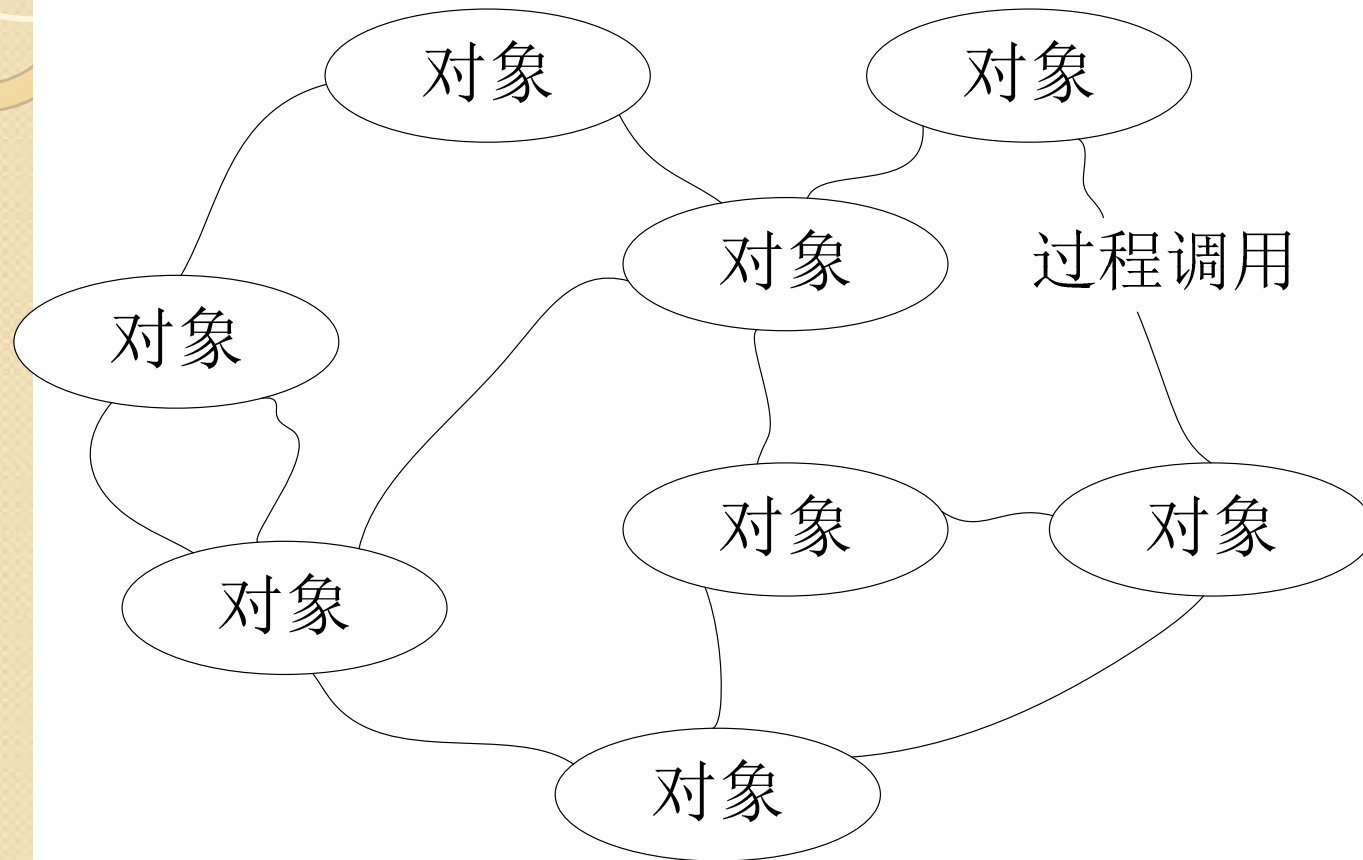


图4-7面向对象风格

4.3.3 面向对象风格

- 优点：

- (1) 对象隐藏了其实现细节，所以可以在不影响其它对象的情况下改变对象的实现，不仅使得对象的使用变得简单、方便，而且具有很高的安全性和可靠性。
- (2) 设计者可将一些数据存取操作的问题分解成一些交互的代理程序的集合。

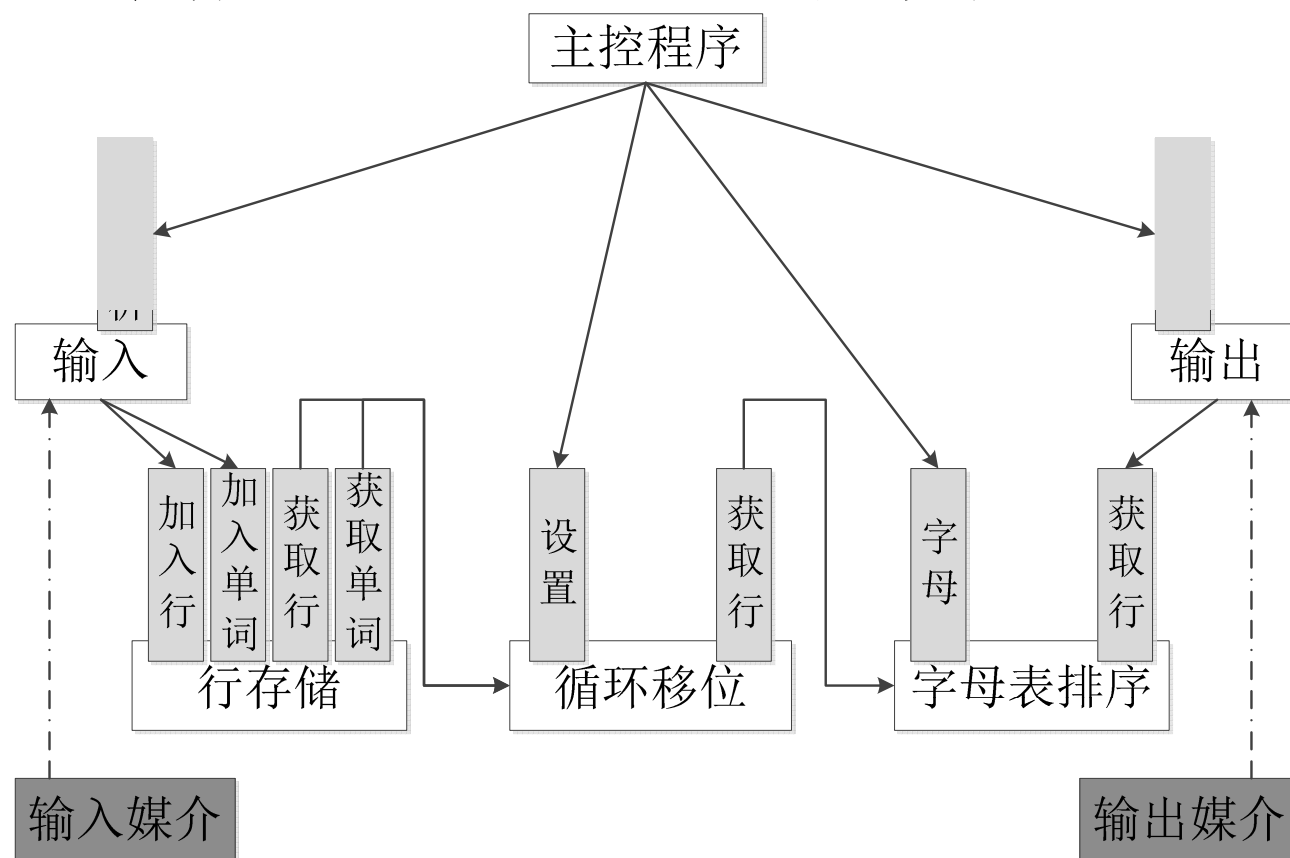
4.3.3 面向对象风格

- 缺点

- 当一个对象和其它对象通过过程调用等方式进行交互时，必须知道其它对象的标识。无论何时改变对象的标识，都必须修改所有显式调用它的其它对象，并消除由此带来的一些副作用。

4.3.3 面向对象风格

- 应用实例：KWIC检索系统

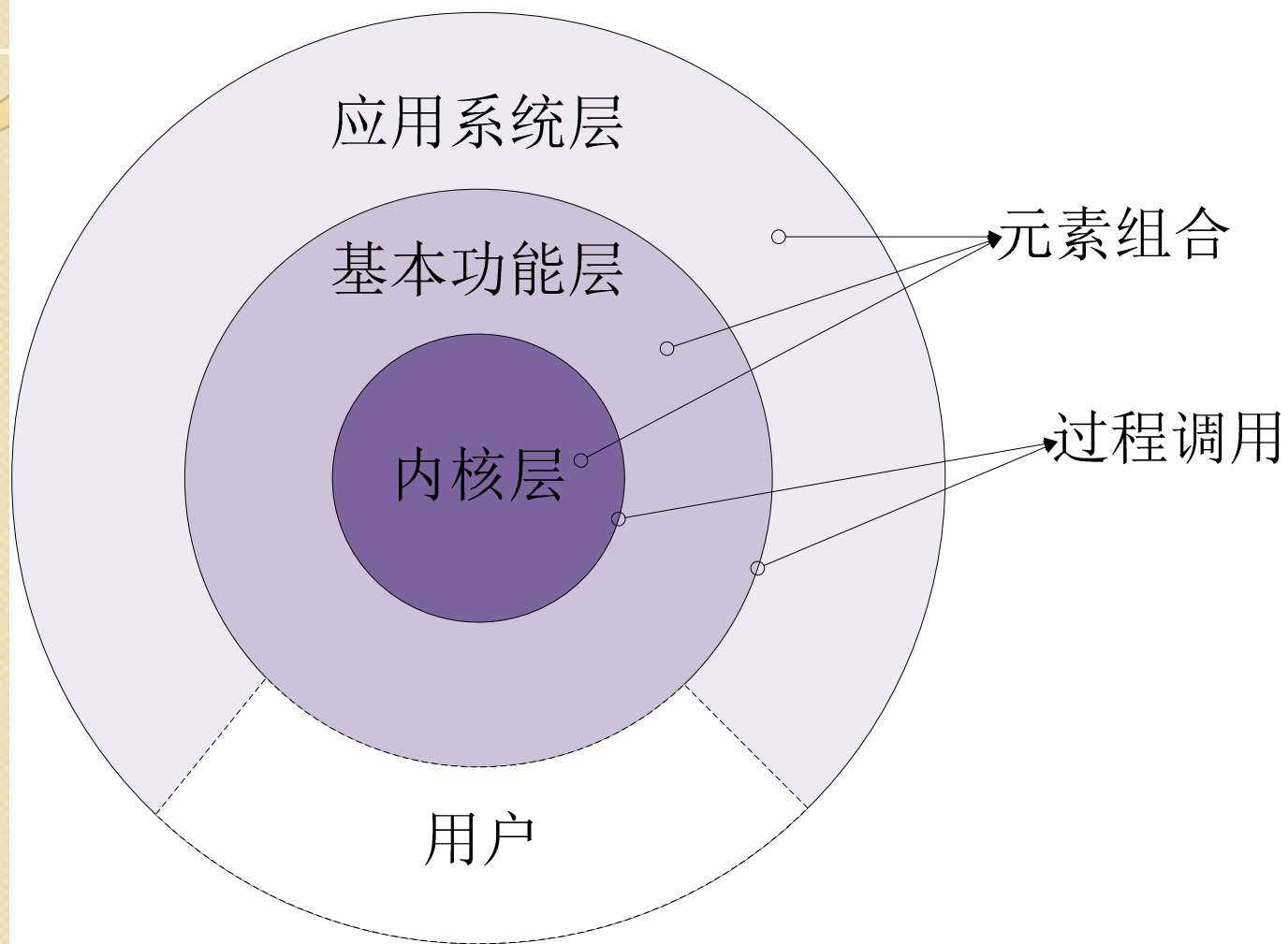


图例： □ 模块（对象） ■ 公共接口 ■ 输入/输出媒介
——→ 方法调用 - - - -> 系统输入/输出

4.3.4 层次化风格

- 基本思想
 - 在分层系统（Layered System）中，系统被组织成若干个层次，每个层次由一系列组件组成；层次之间存在接口，通过接口形成call/return的关系——下层组件向上层组件提供服务，上层组件被看作是下层组件的客户端。
- 层次化早已经成为一种复杂系统设计的普遍性原则。

4.3.4 层次化风格



4.3.4 层次化风格

- 在分层架构中，组件被划分成几个水平层，每个层在应用中执行特定角色。
- 大多数分层架构由四个标准层组成：
 - 表现层 (presentation)
 - 业务层 (business)
 - 持久层 (persistence)
 - 数据库层 (database)
 - (有些情况下将业务层和持久层结合在一起成为一个单独的业务层)

从解耦角度看分层架构

- 分层架构解耦
 - 分层架构中的每个层都标记为关闭（closed），即：当请求从一个层移动到另一个层时，它必须经过它正下方的层，以到达该层下面的下一层。
 - 例如，源自表现层的请求必须首先经过业务层，然后在最终到达数据库层之前到达持久层。

从解耦角度看分层架构

- 分层架构解耦
- 在经典分层架构设计之前，有两个基本概念需要牢记：
 - (1) 高层模块不应该依赖于底层模块，两者都应该依赖于抽象。
 - (2) 抽象不应该依赖于细节，细节应该依赖于抽象。
- 这也是为了达到高内聚低耦合的设计要求，起到解耦的作用。

4.3.4 层次化风格

- 优点:

- (1) 支持基于可增加抽象层的设计，允许将一个复杂问题分解成一个增量步骤序列的实现。
- (2) 支持扩展。每一层的改变最多只影响相邻层。
- (3) 支持重用。只要给相邻层提供相同的接口，它允许系统中同一层的不同实现相互交换使用。

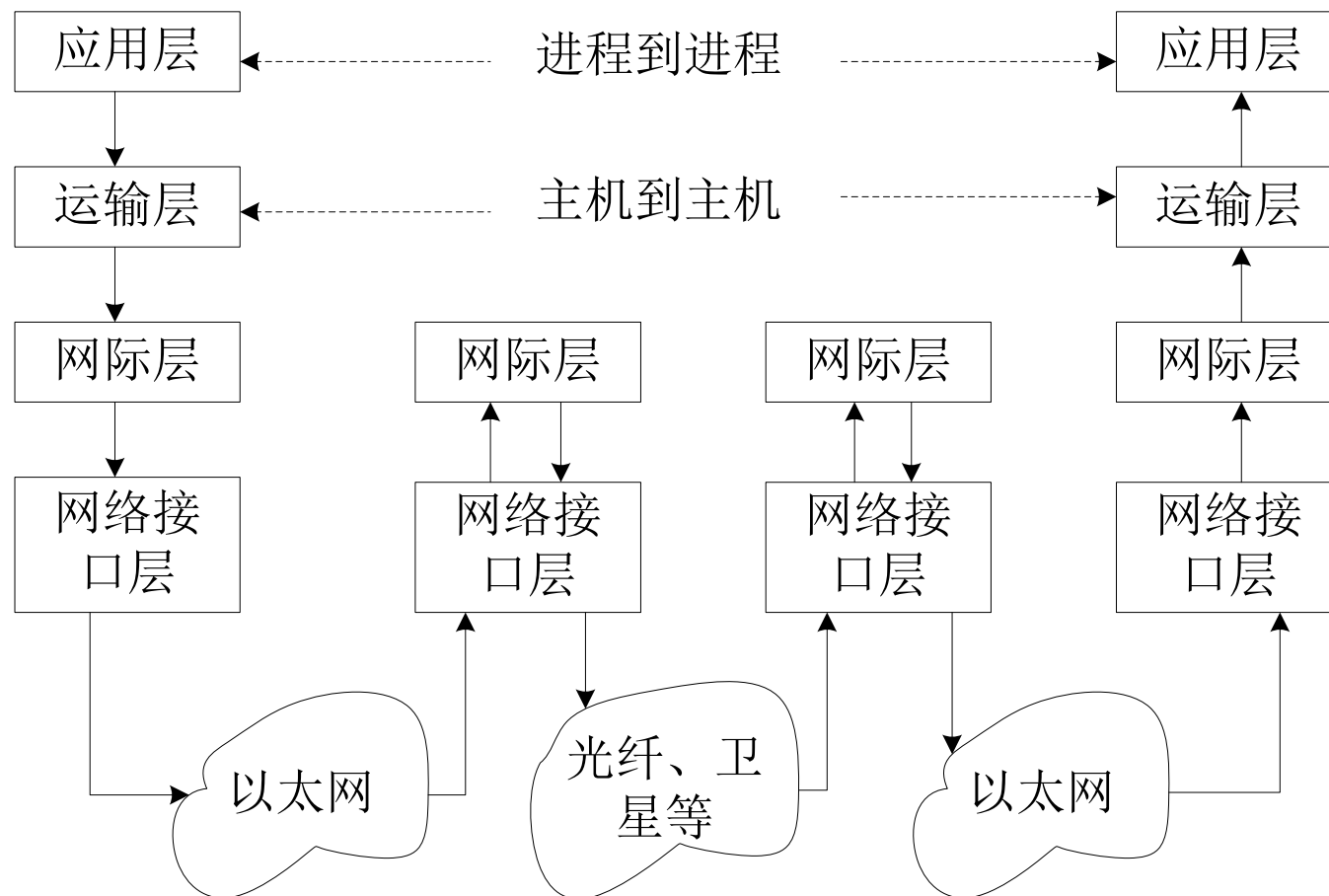
4.3.4 层次化风格

- 缺点

- (1) 不是所有系统都容易用这种模式来构建；
- (2) 定义一个合适的抽象层次可能会非常困难，特别是对于标准化的层次模型。
 - 例如，实际的通信协议体就很难映射到ISO框架中，因为其中许多协议跨多个层。

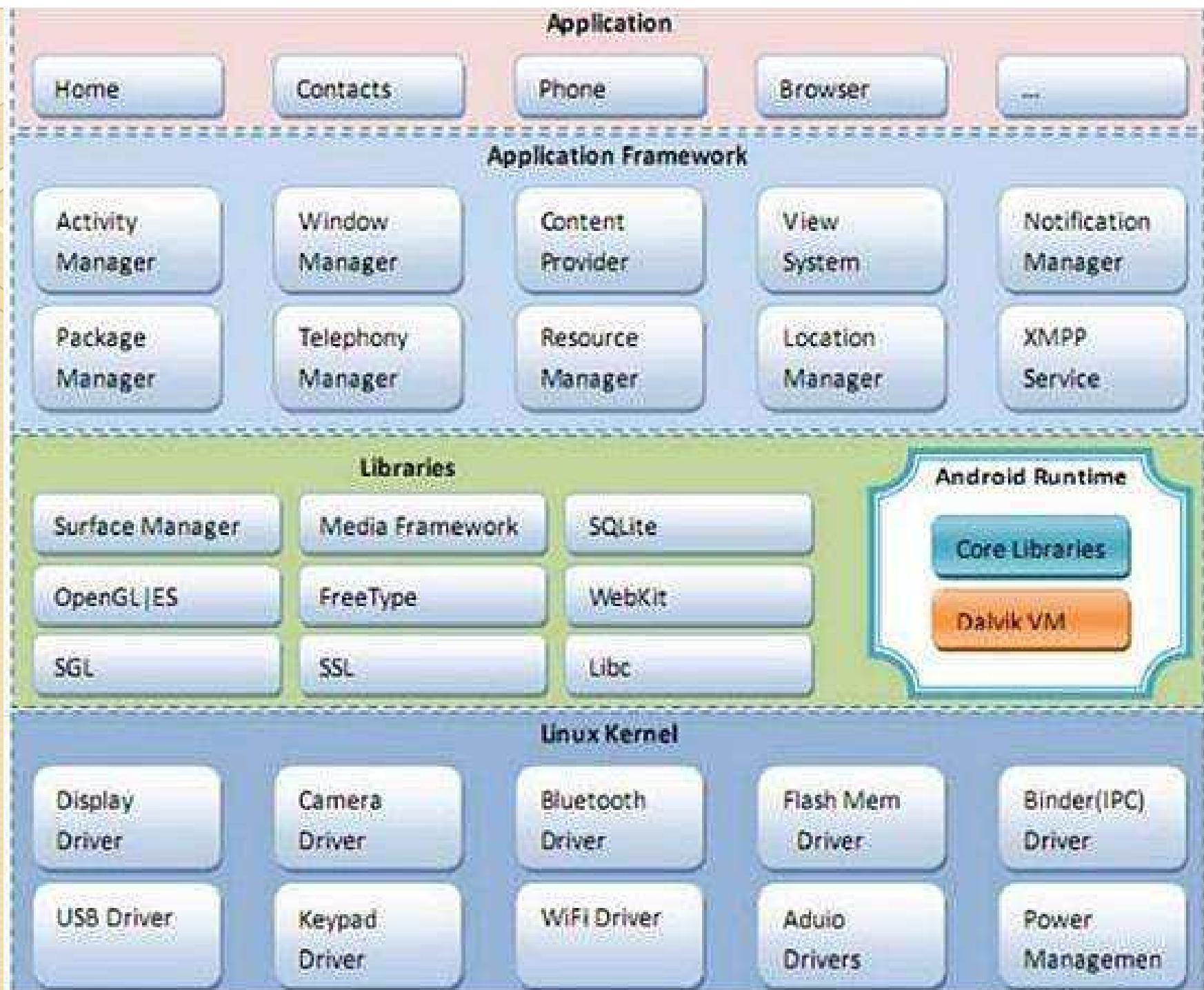
4.3.4 层次化风格

- 应用实例：TCP/IP分层通信协议



作业

- 分析了解安卓的分层架构。



4.3.5 事件驱动风格

- 通常，在一个系统中，组件接口提供了访问过程或函数的端口的集合，组件通过显式地调用这些过程或函数来与其他组件交互。
- 然而，一种基于隐式调用(implicit invocation)的集成技术非常受关注，该技术就是事件驱动(Event-based) 的软件架构风格。

4.3.5 事件驱动风格

- 基本思想：
 - 不直接调用一个过程，而是发布或广播一个或多个事件。系统中的其它组件通过注册与一个事件关联起来的过程，来表示对某一个事件感兴趣。当这个事件发生时，系统本身会调用所有注册了这个事件的过程。这样一个事件的激发会导致其它模块中过程的隐式调用。
 - 事件：是指可以被系统识别的、发生在界面上的用户动作或者状态变化。

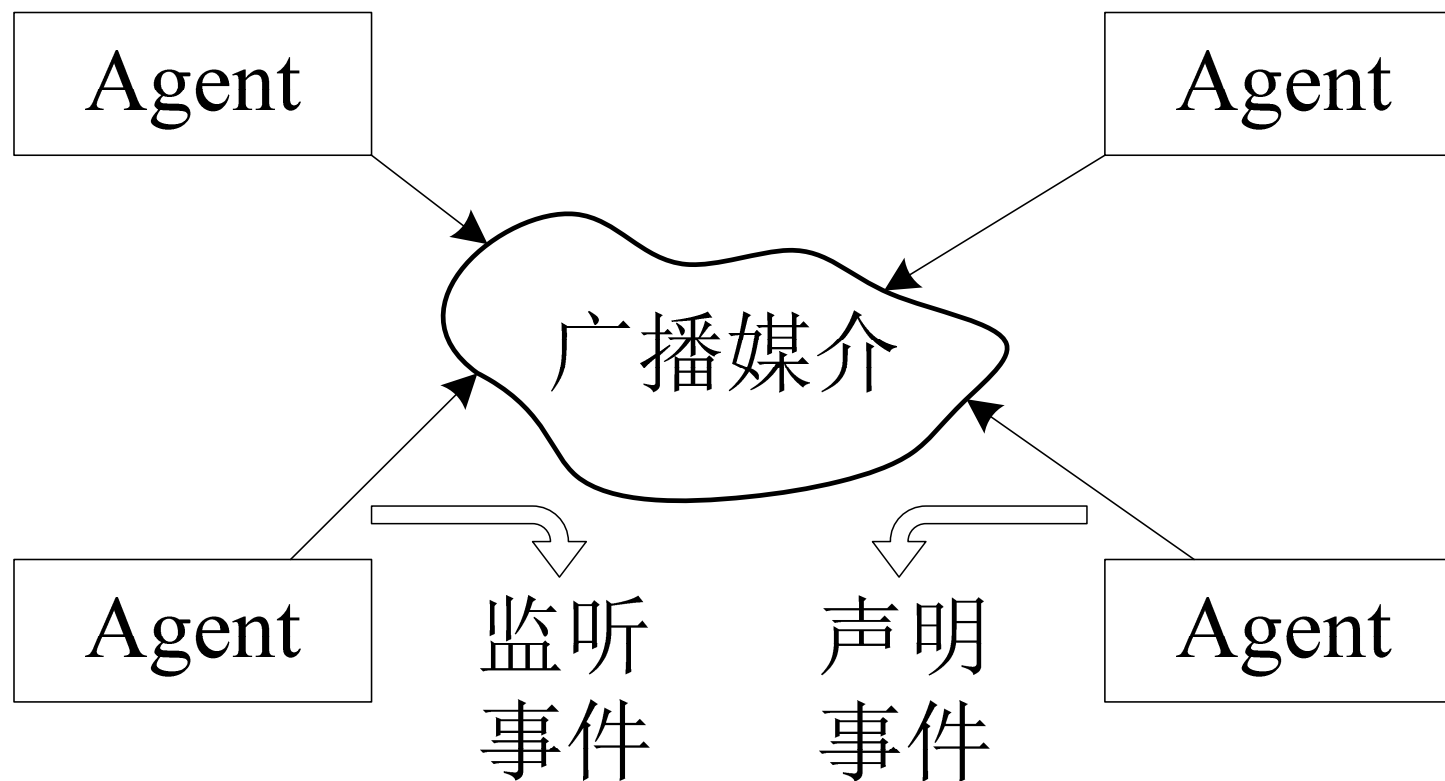
4.3.5 事件驱动风格

- 事件驱动编程的一般步骤：
 - 1) 确定响应事件的元素；
 - 2) 为指定元素确定需要响应的事件类型；
 - 3) 为该元素的相应事件编写事件处理程序；
 - 4) 将事件处理程序绑定到指定元素的指定事件。

从解耦角度看事件驱动风格

- 系统组件松耦合
- 在系统执行过程中，可以并行处理不可预期的事件发生
- 观察者模式
 - 让多个观察者对象同时监听某一主题对象，当该对象状态发生变化时，会通知所有观察者对象，使它们能够自动更新自己。

4.3.5 事件驱动风格



4.3.5 事件驱动风格

- 特点：

- 事件发布者不知道哪些组件会受到事件的影响；组件不能对事件的处理顺序，或者事件发生后的处理结果做任何假设。
- 从架构上来说，事件驱动系统的组件提供了一个过程集合和一组事件。
- 过程可以使用显示的方法进行调用，也可以由组件在系统事件中注册。当触发事件时，会自动引发这些过程的调用。
- 连接件既可以是显示过程调用，也可以是一种绑定事件声明和过程调用的手段。

4.3.5 事件驱动风格

- 优点

- (1) 事件声明者不需要知道哪些组件会影响事件，组件之间关联较弱。
- (2) 提高软件复用能力。只要在系统事件中注册组件的过程，就可以将该组件集成到系统中。
- (3) 系统便于升级。只要组件名和事件中所注册的过程名保持不变，原有组件就可以被新组件替代。

4.3.5 事件驱动风格

- 缺点
- (1) 组件放弃了对计算的控制权，完全由系统来决定。
- (2) 存在数据交换问题。
- (3) 该风格中，正确性验证成为一个问题。

4.3.5 事件驱动风格

- 应用实例：Field系统

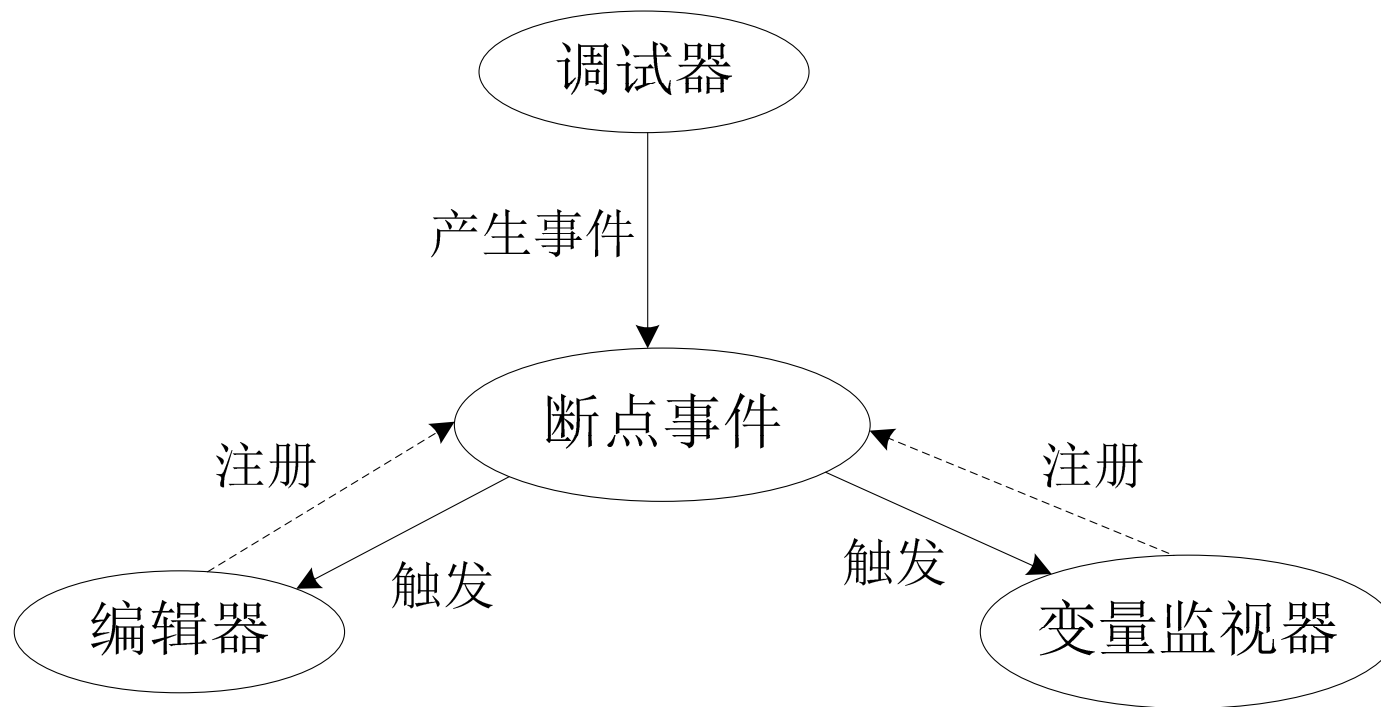
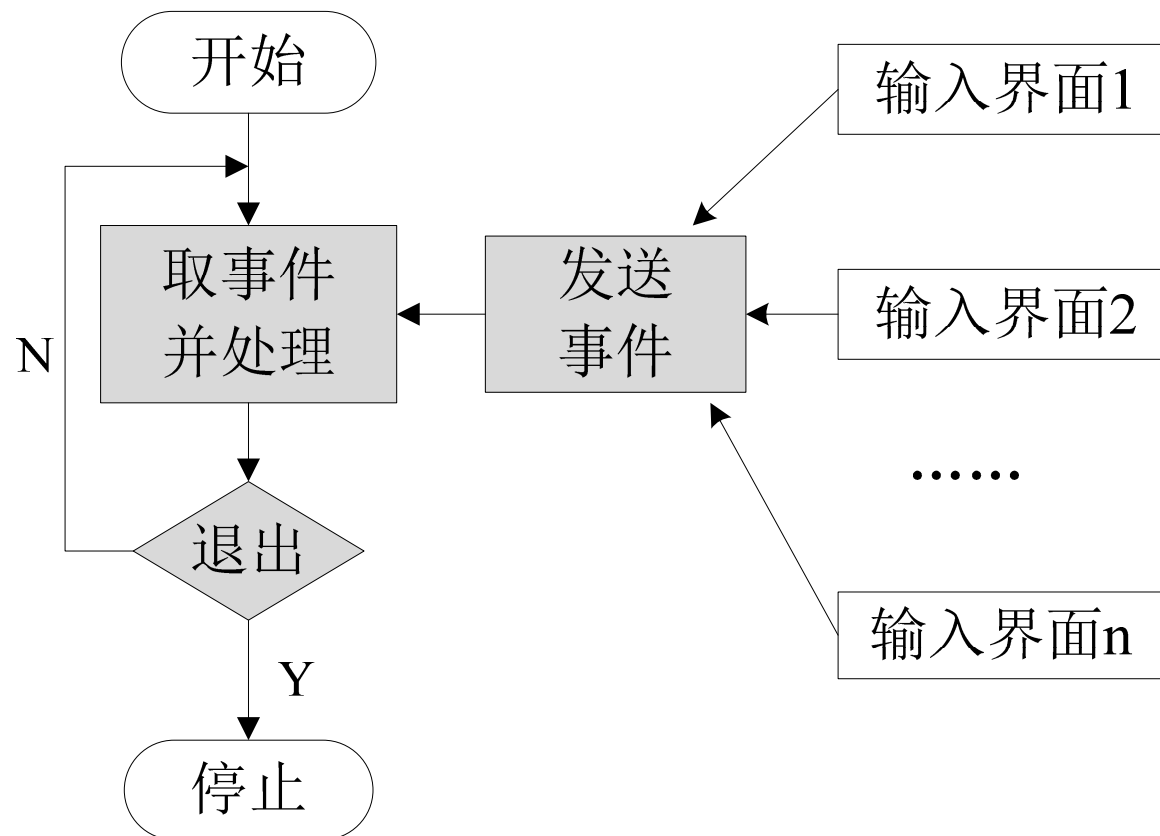


图4.12调试器中的断点处理

4.3.5 事件驱动风格

- 应用实例： Win32 GUI程序



4.3.6 解释器风格

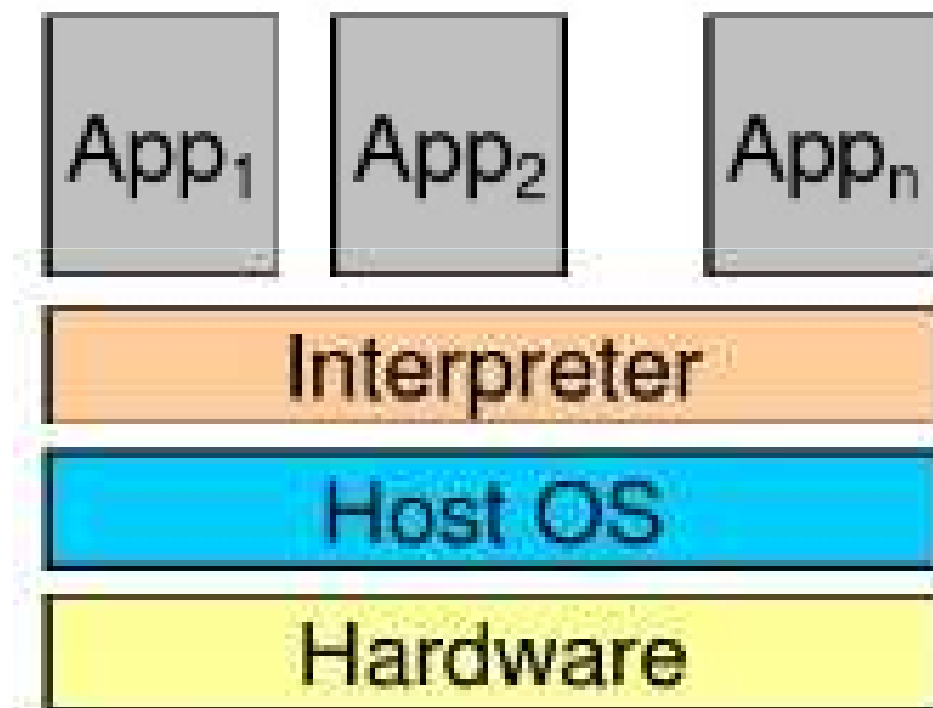
- 基本思想

- 解释器(Interpreter) 是一个用来执行其他程序的程序，它针对不同的硬件平台实现了一个虚拟机，将高抽象层次的程序翻译为低抽象层次所能理解的指令，以弥合程序语义所期望的与硬件提供的计算引擎之间的差距。



图4.14解释执行过程

4.3.6 解释器风格



4.3.6 解释器风格

- 组成部分

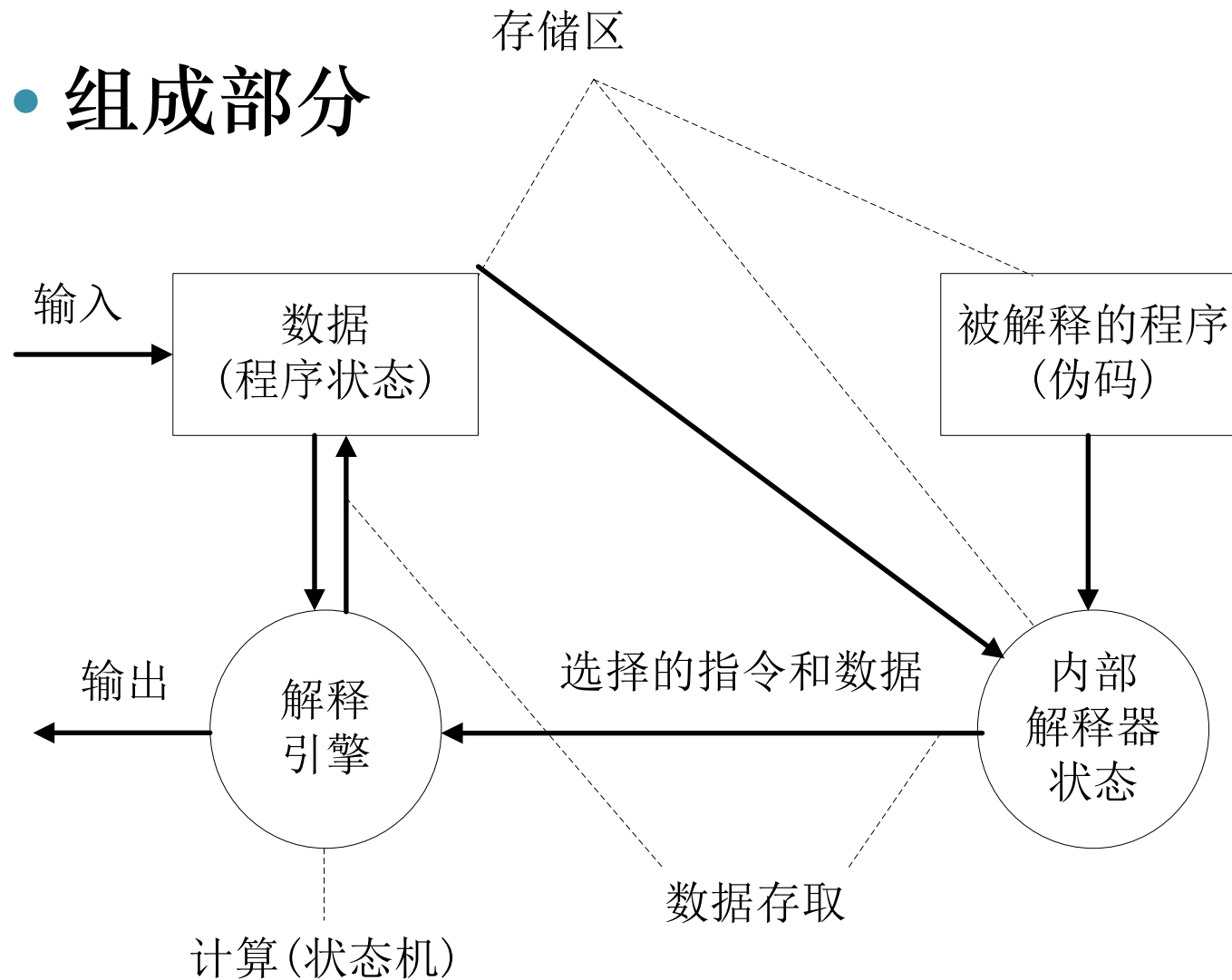


图4.15解释器风格

4.3.6 解释器风格

- 优点

- (1) 它有利于实现程序的可移植性和语言的跨平台能力；
- (2) 可以对未来的硬件进行模拟和仿真，能够降低测试所带来的复杂性和昂贵花费

4.3.6 解释器风格

- 缺点

- 额外的间接层次导致了系统性能的下降，如在不引入JIT（Just In Time）技术的情况下，Java应用程序的运行速度相当慢。

4.3.6 解释器风格

- 应用实例

- JVM

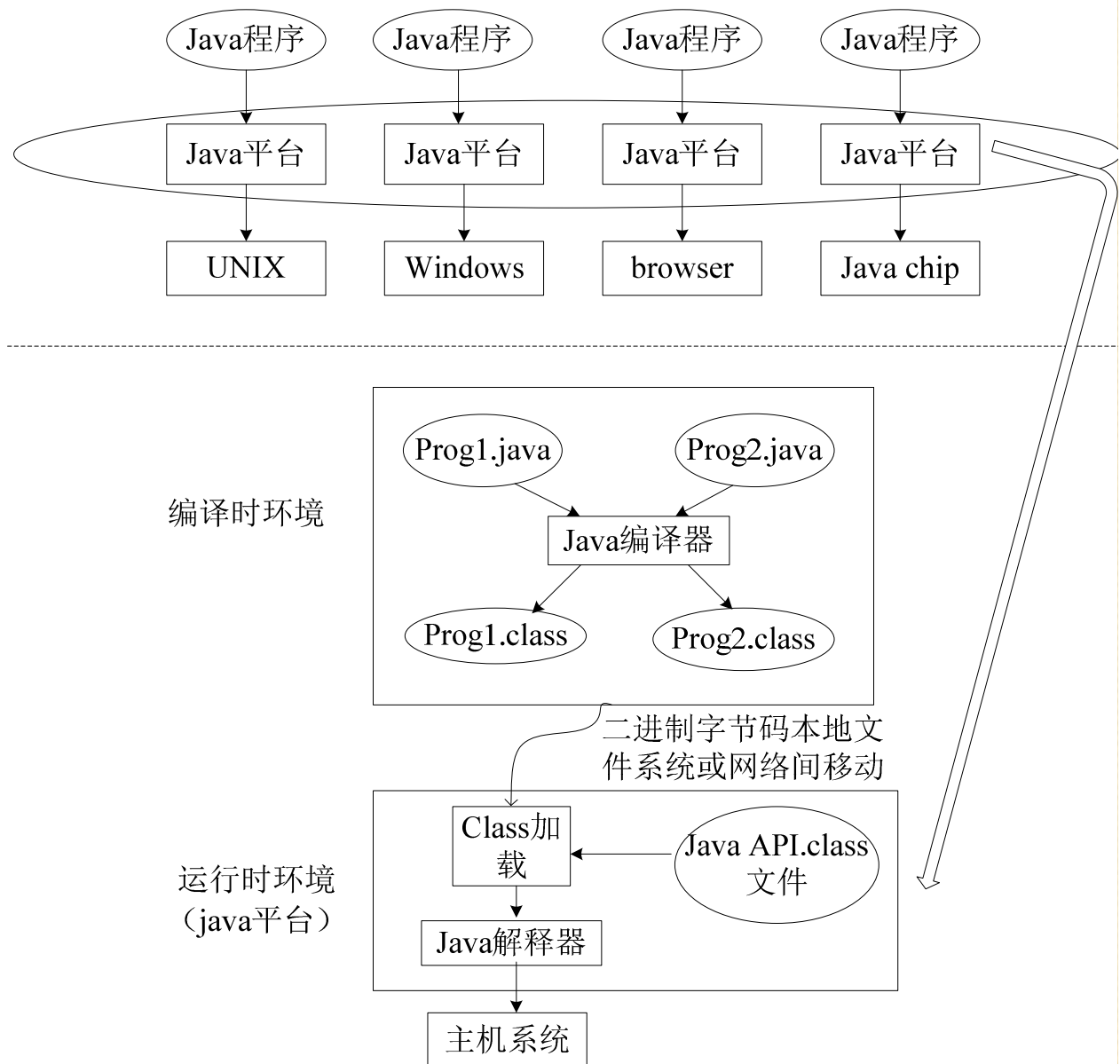


图4.16基于JVM的
Java程序执行过程

4.3.6 解释器风格

- 应用实例： java class文件在JVM下通过解释器运行

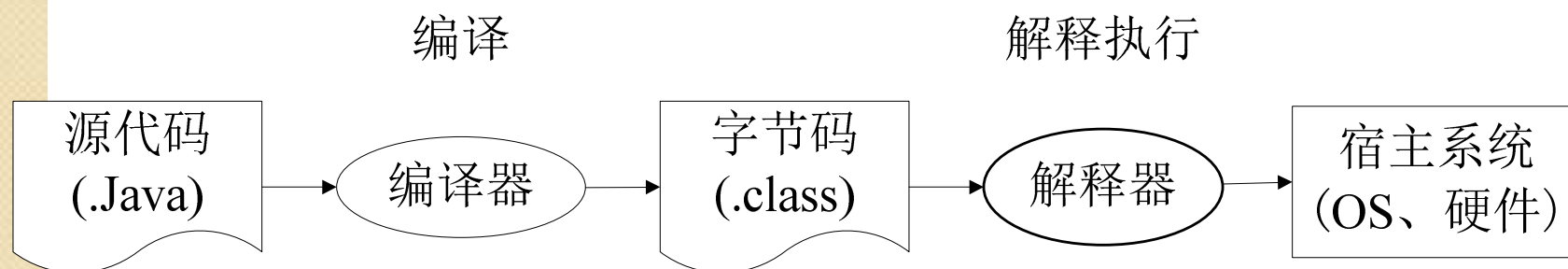


图4-17通过解释器执行.class文件

4.3.6 解释器风格

- 应用实例java class文件在JVM下通过JIT (just-in-time) 编译器运行

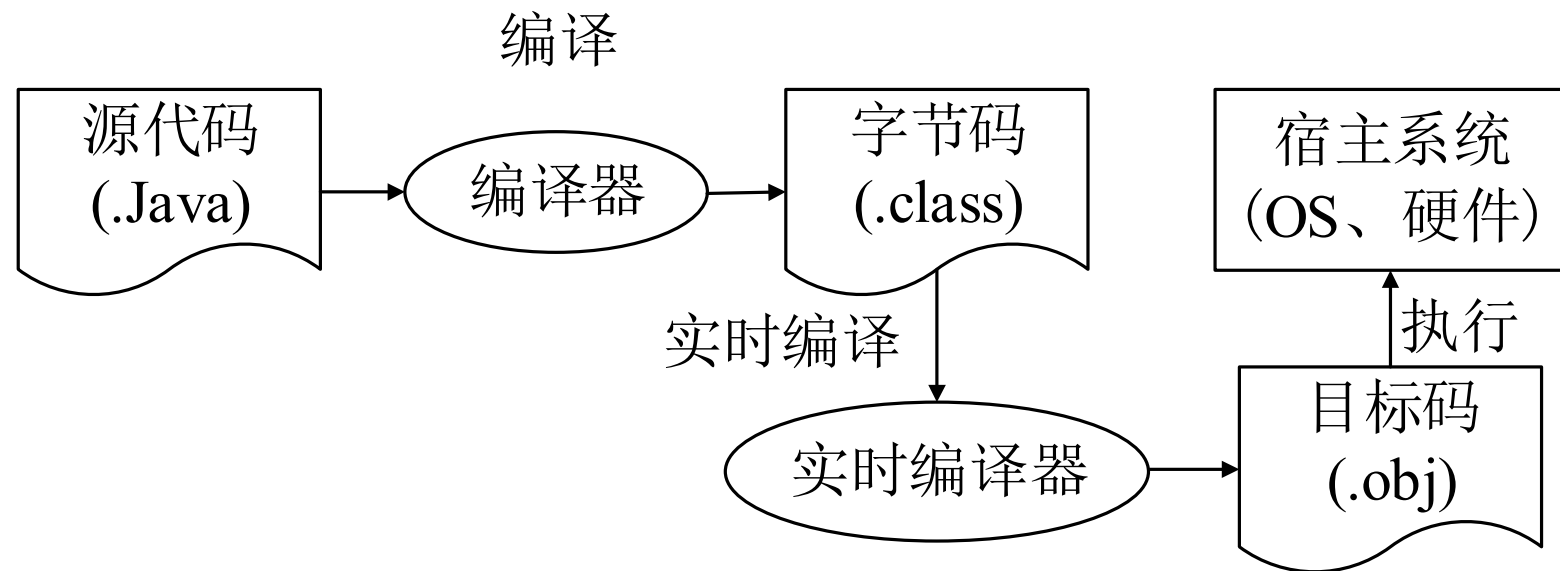


图4-18通过实时编译方式执行.class文件

4.3.7 基于规则的系统风格

- 现实里的业务需求经常频繁的发生变化，不断修改代码效率低、成本高。最好的办法是把频繁变化的业务逻辑抽取出来，形成独立的规则库。
- 这些规则可独立于软件系统而存在，可被随时的更新。
- 系统运行时，读取规则库，依据当前运行状态，从规则库中选择与之匹配的规则，对规则进行解释，根据结果控制系统运行的流程。

4.3.7 基于规则的系统风格

- 业务逻辑=固定业务逻辑+可变业务逻辑(规则)+规则引擎
- 基于规则系统 (Rule-based System)
 - 一个使用模式匹配搜索来寻找规则并在正确的时候应用正确的逻辑知识的虚拟机。
 - 基于规则系统提供了一种基于专家系统解决问题的手段，将知识表示为“条件-行为”的规则，当满足条件时触发相应的行为，而不是将这些规则直接写在程序源代码中。

4.3.7 基于规则的系统风格

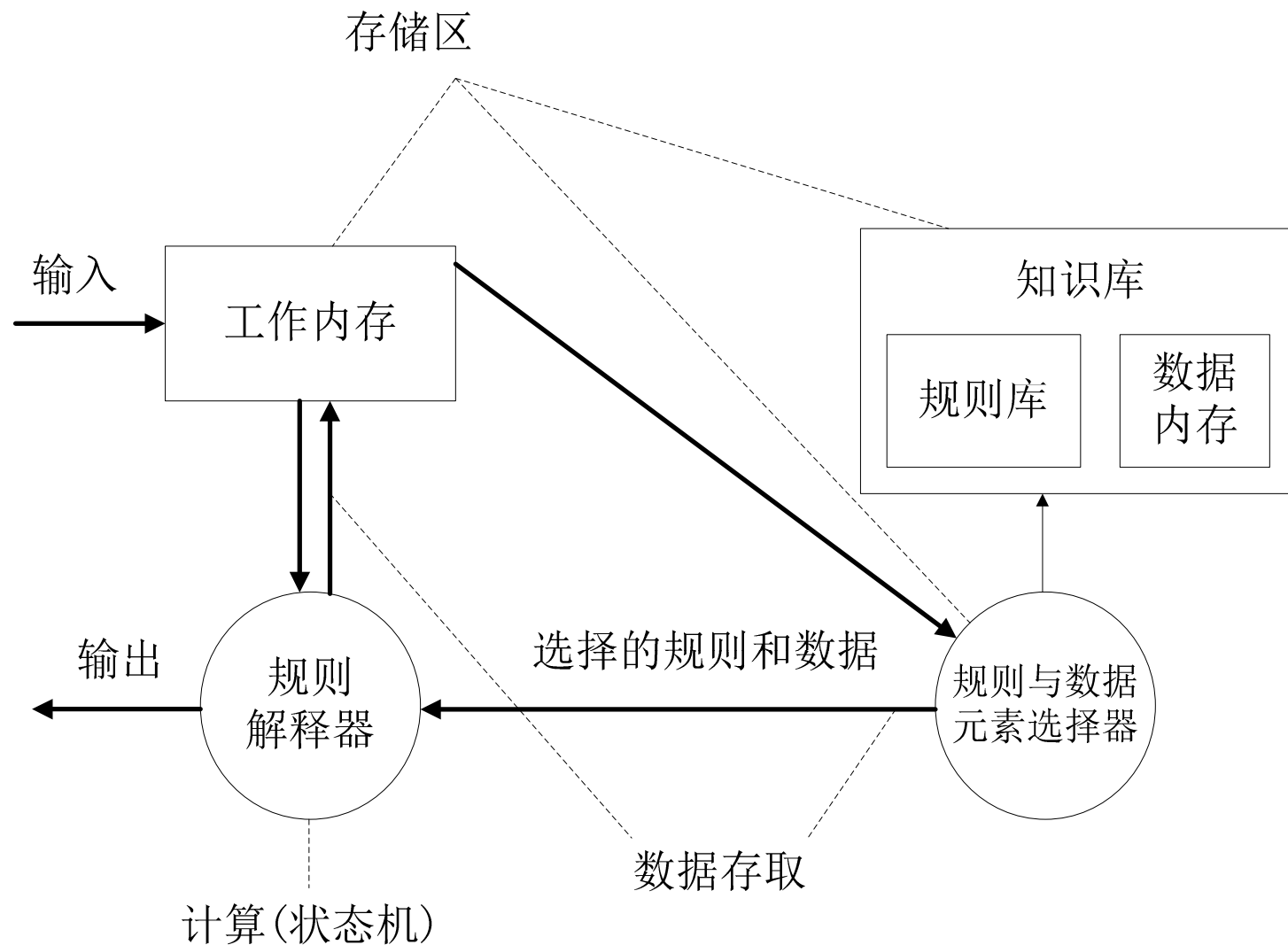


图4-19基于规则的系统

4.3.7 基于规则的系统风格

- 基于规则的系统风格的基本组件与解释器风格的组件相似

基于规则的系统	解释器风格
知识库	待解释的程序（伪码
规则解释器	解释器引擎
规则与数据元素选择	解释器引擎内部的控制状态
工作内存	程序当前的运行状态

4.3.7 基于规则的系统风格

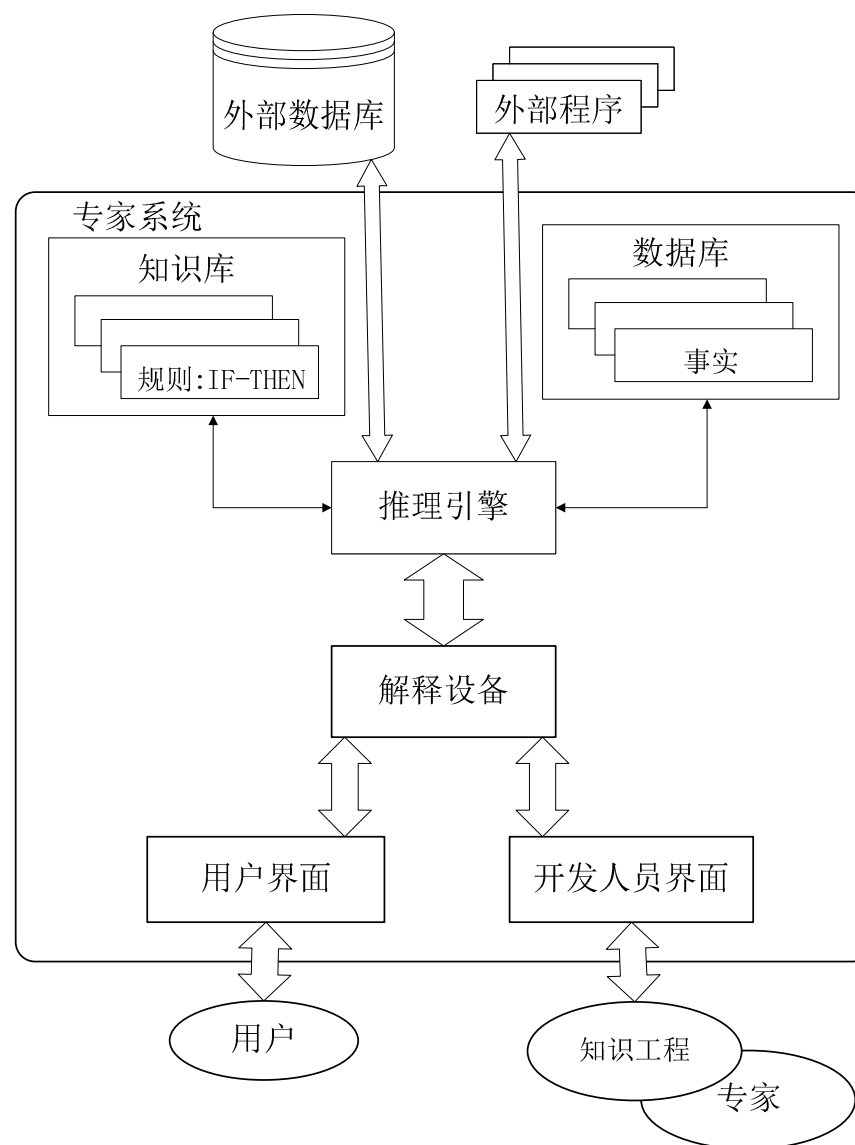
- 基于规则的系统和解释器风格的系统比较

	基于规则的系统	解释器风格的系统
相同点	基于规则的系统本质上是与解释器风格一致的，都是通过“解释器”(“规则引擎”)，在两个不同的抽象层次之间建立起一种虚拟的环境。	
不同点	在自然语言/XML的规则和高级语言的程序源代码之间建立虚拟机环境	在高级语言程序源代码和OS/硬件平台之间建立虚拟机环境

- 基于规则的系统与解释器风格的系统类似。

4.3.7 基于规则的系统风格

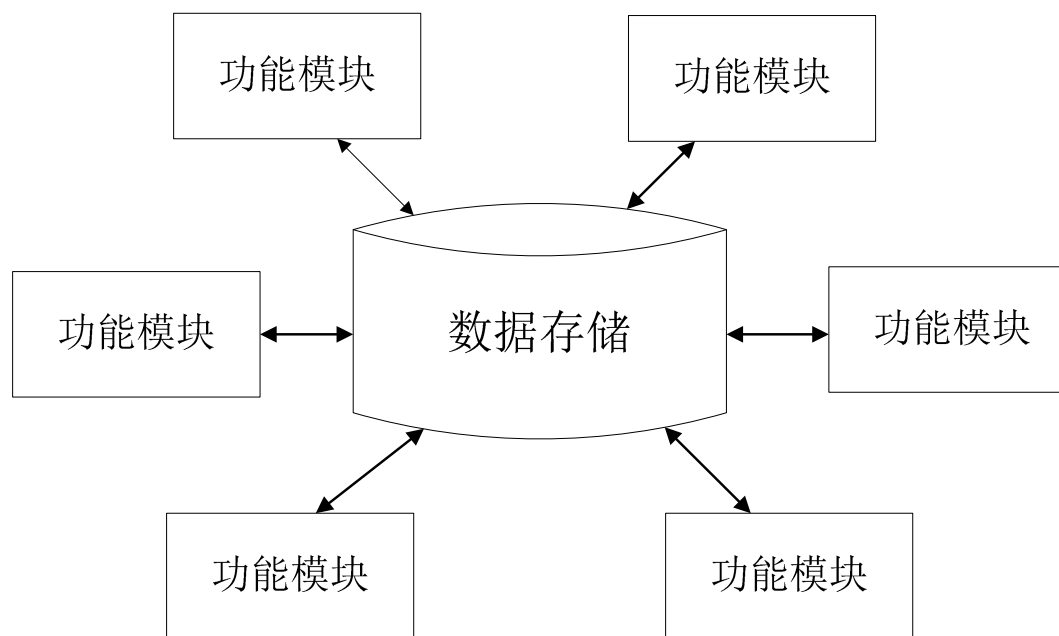
- 应用实例
 - 基于规则的专家系统



4.3.8 仓库风格

- 基本思想

- 仓库是存储和维护数据的中心场所。
- 仓库式风格的两种组件：中央数据结构组件、相对独立的组件集合



4.3.8 仓库风格

- 优点

- 便于模块间的数据共享
- 方便模块的添加、更新和删除
- 避免了知识源的不必要的重复存储等。

- 缺点

- 对于各个模块，需要一定的同步/加锁机制保证数据结构的完整性和一致性等。

4.3.8 仓库风格

- 应用实例：现代编译器

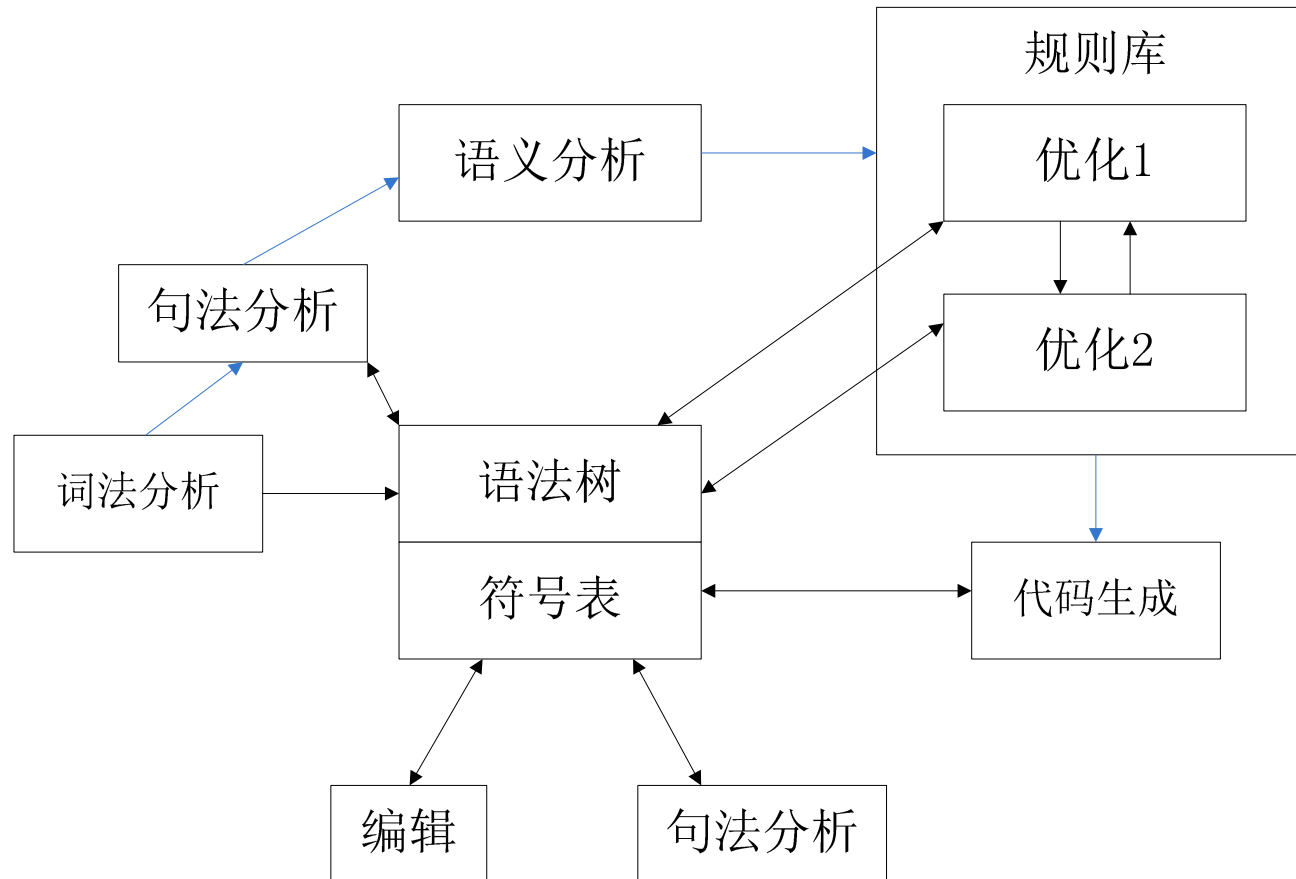


图4.22现代的规范编译器结构

4.3.8 仓库风格

- 应用实例：基于仓库风格的软件开发环境Eclipse

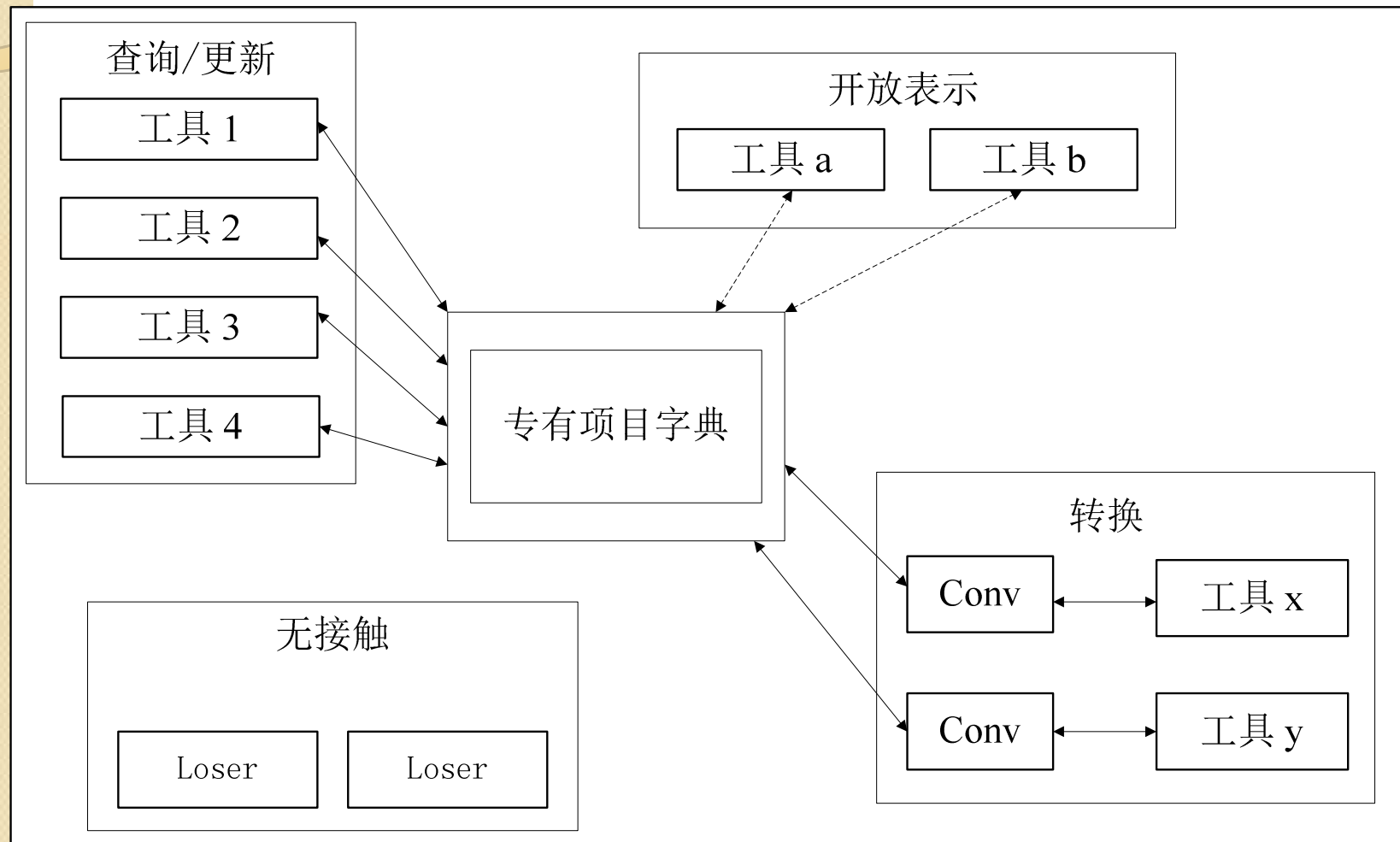


图4-23基于仓库风格的Eclipse软件开发环境

4.3.9 黑板系统风格

- 黑板系统（Blackboard System）是传统上被用于信号处理方面进行复杂解释的应用程序，以及松散耦合的组件访问共享数据的应用程序。
- 黑板架构实现的基本出发点是已经存在一个对公共数据结构进行协同操作的独立程序集合。

4.3.9 黑板系统风格

- 组成部分

- (1) 知识源 (2) 黑板数据结构 (3) 控制器

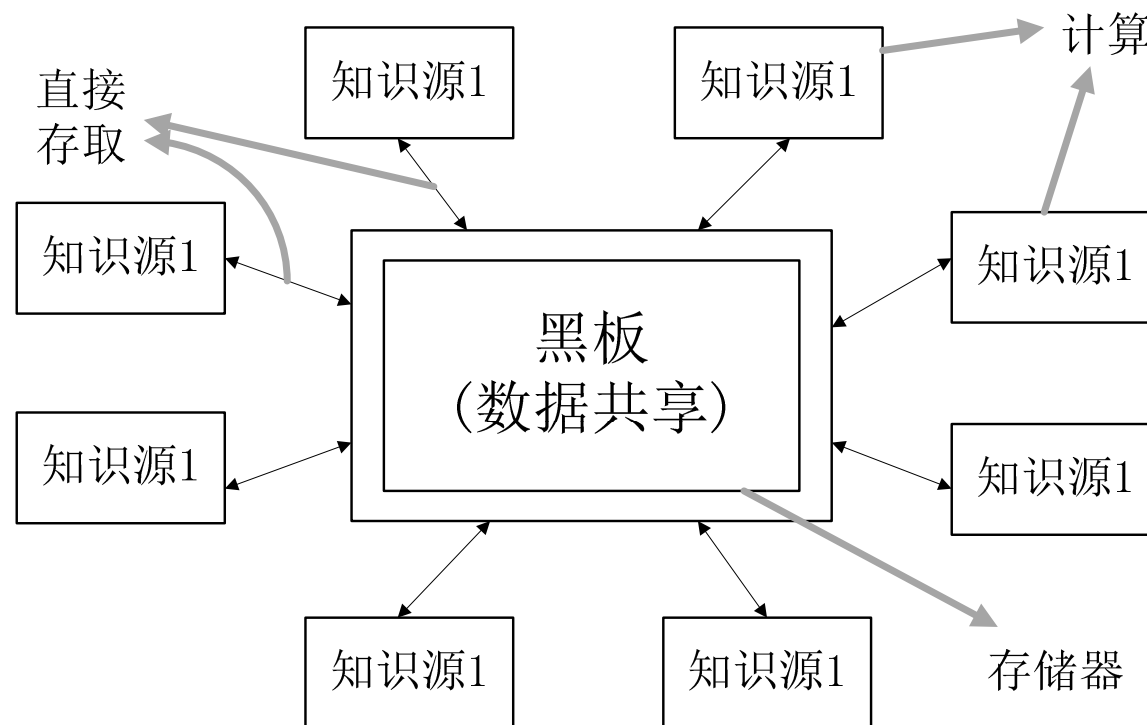


图4.24黑板架构风格

4.3.9 黑板系统风格

- 优点

- (1) 便于多客户共享大量数据，他们不关心数据何时有的、谁提供的、怎样提供的。
- (2) 既便于添加新的作为知识源代理的应用程序，也便于扩展共享的黑板数据结构。
- (3) 知识源可重用。
- (4) 支持容错性和健壮性。

4.3.9 黑板系统风格

- 缺点

- (1) 不同的知识源代理对于共享数据结构要达成一致，而且，这也造成对黑板数据结构的修改较为困难——要考虑到各个代理的调用。
- (2) 需要一定的同步/加锁机制保证数据结构的完整性和一致性，增大了系统复杂度。

4.3.9 黑板系统风格

- 应用实例
 - HEARSAY-II

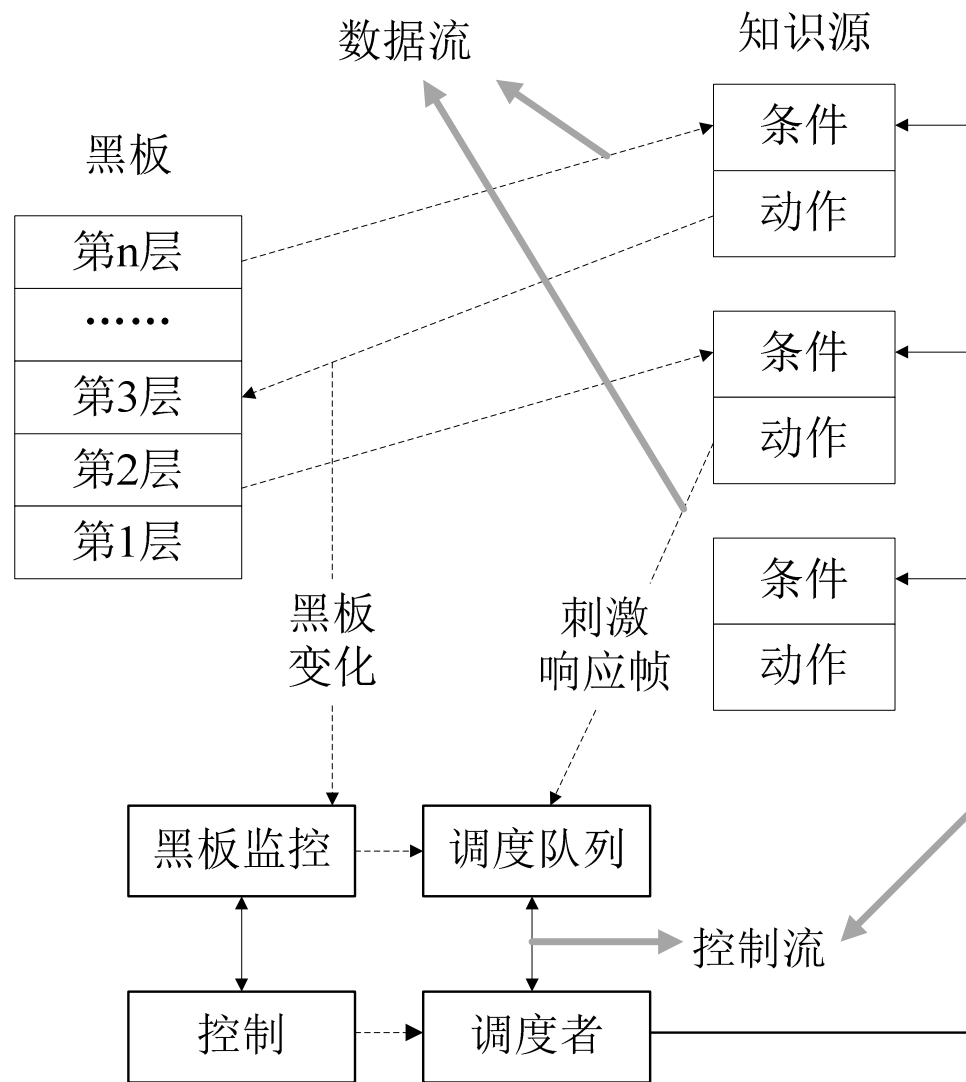


图4.25 HEARSAY-II架构

4.3.10 C2风格

- C2架构风格是一种常见的层次体系架构风格。
- 1995年由California大学Irvine分校的Richard. N. Taylor等人提出
- C2风格的主要思想来源于Chiron-1用户界面系统，因此又被命名为Chiron-2，简称C2。

4.3.10 C2风格

- C2架构风格可以概括为：通过连接件绑定在一起的按照一组规则运作的**并行组件网络**。该规则规定了所有组件之间的交互必须通过**异步消息机制**来实现
- C2是一种基于组件和消息的架构风格，适用于GUI软件开发，构建灵活和可扩展的应用系统。

4.3.10 C2风格

- C2风格的系统组织规则：
 - (1) 组件之间不能直接连接；
 - (2) 组件和连接件都有一个顶部和一个底部；
 - (3) 组件的顶部应连接到某连接件的底部，组件的底部则应连接到某连接件的顶部；
 - (4) 一个连接件可以和任意数目的其他组件和连接件连接；
 - (5) 当两个连接件进行直接连接时，必须由其中一个的底部到另一个的顶部。

4.3.10 C2风格

- 其结构如图所示:

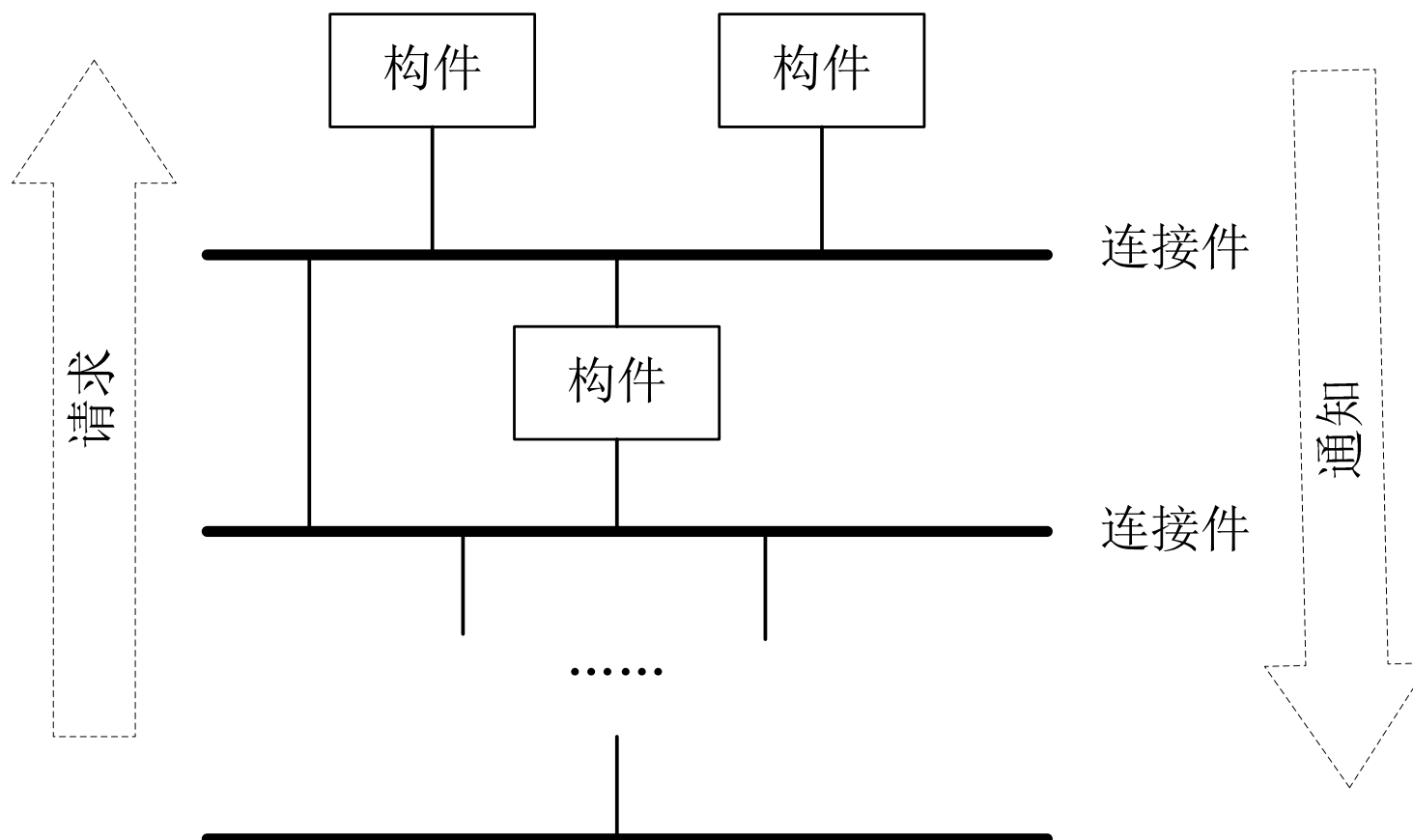


图4.26 C2风格

4.3.10 C2风格

- C2架构的内部，通信和处理是分开完成的。

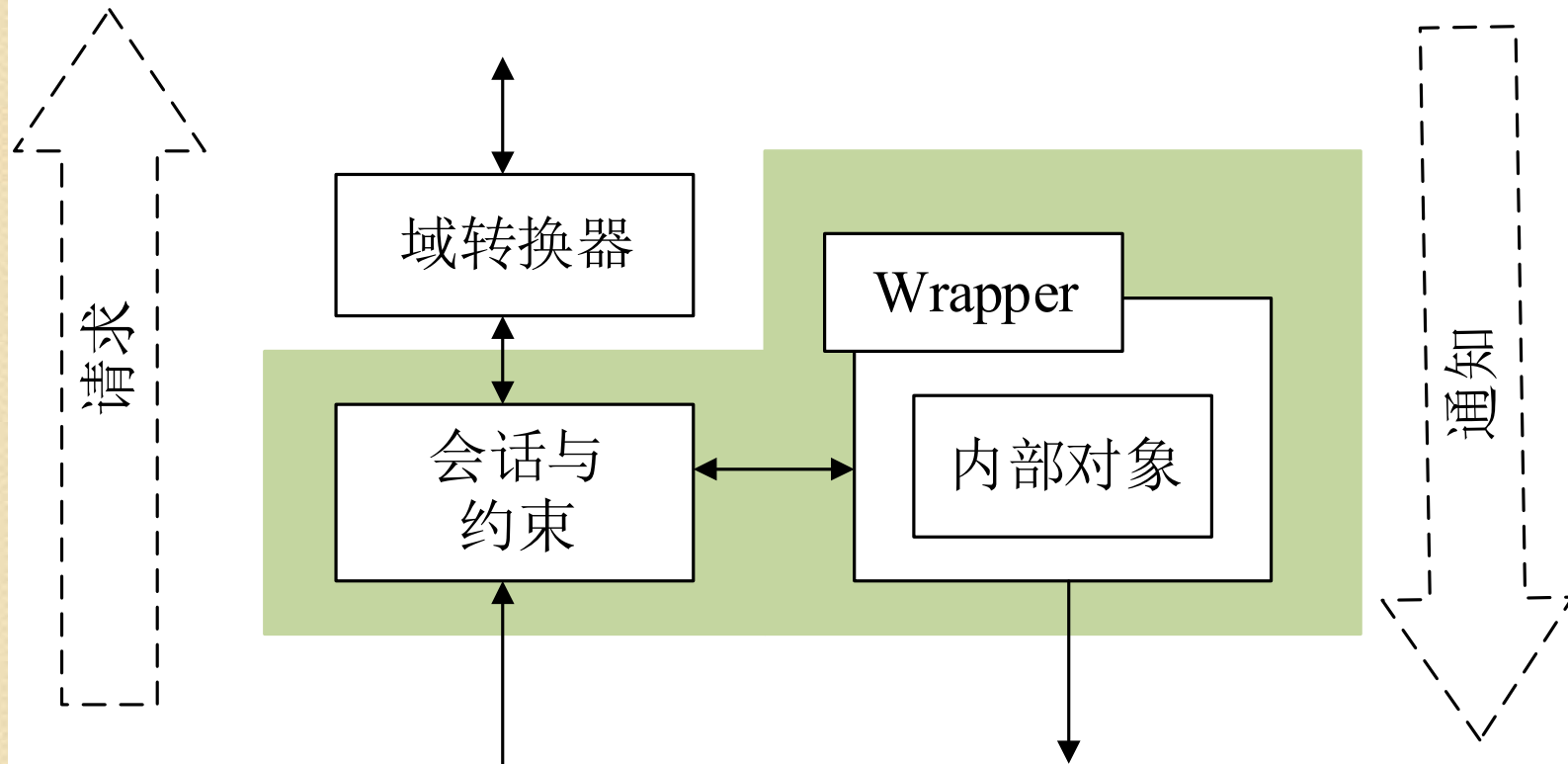


图4-27 C2组件的内部结构

4.3.10 C2风格

- 优点：

- (1) 可使用任何编程语言开发组件，组件重用和替换易实现；
- (2) 由于组件之间相对独立，依赖较小，因而该风格具有一定扩展能力，可支持不同粒度的组件；
- (3) 组件不需共享地址空间；
- (4) 可实现多个用户和多个系统之间的交互；
- (5) 可使用多个工具集和多种媒体类型，动态更新系统框架结构。

4.3.10 C2风格

- 缺点

- 不太适合大规模流式风格系统，以及对数据库使用比较频繁的使用。

4.3.10 C2风格

- 应用实例
 - KLAX游戏
 - 旋转方块，一种街机模拟游戏



游戏逻辑组件

Artist组件

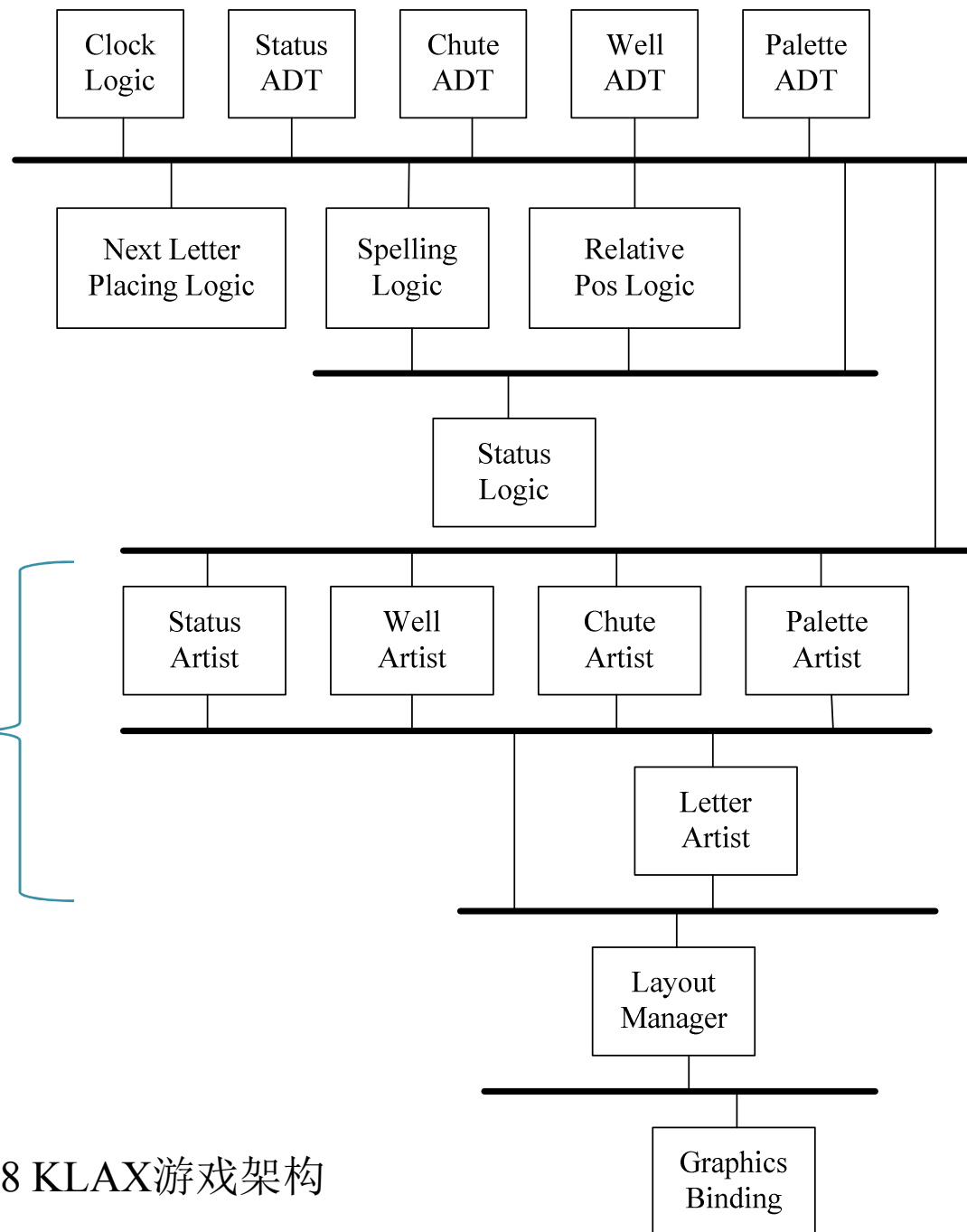


图4.28 KLAX游戏架构

4.3.11 客户机/服务器风格

- 客户机/服务器 (Client/Server)是20世纪90年代开始成熟的一项技术，主要针对资源不对等问题而提出的一种共享策略。
- 客户机和服务器是两个相互独立的逻辑系统，为了完成特定的任务而形成一种协作关系。
 - 客户机(前端，front-end)：业务逻辑、与服务器通讯的接口；
 - 服务器(后端：back-end)：与客户机通讯的接口、业务逻辑、数据管理。

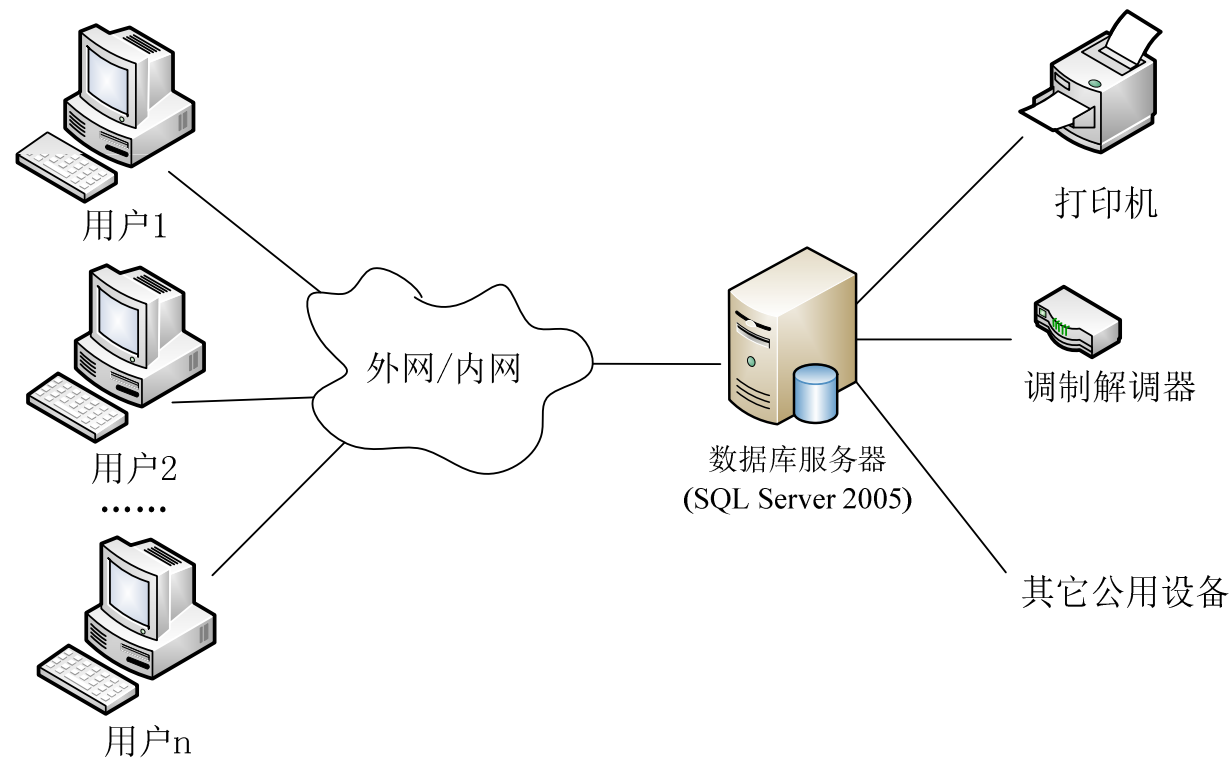
4.3.11 客户机/服务器风格

- 一般的，客户机为完成特定的工作向服务器发出请求；服务器处理客户机的请求并返回结果。
- 客户机程序和服务器程序配置在同一台计算机上时：采用消息、共享存储区和信号量等方法实现通信连接。
- 客户机程序和服务器程序配置在分布式环境中时：通过远程调用（Remote Produce Call, RPC）协议来进行通信。

4.3.11 客户机/服务器风格

- 基本思想

- 两层C/S架构



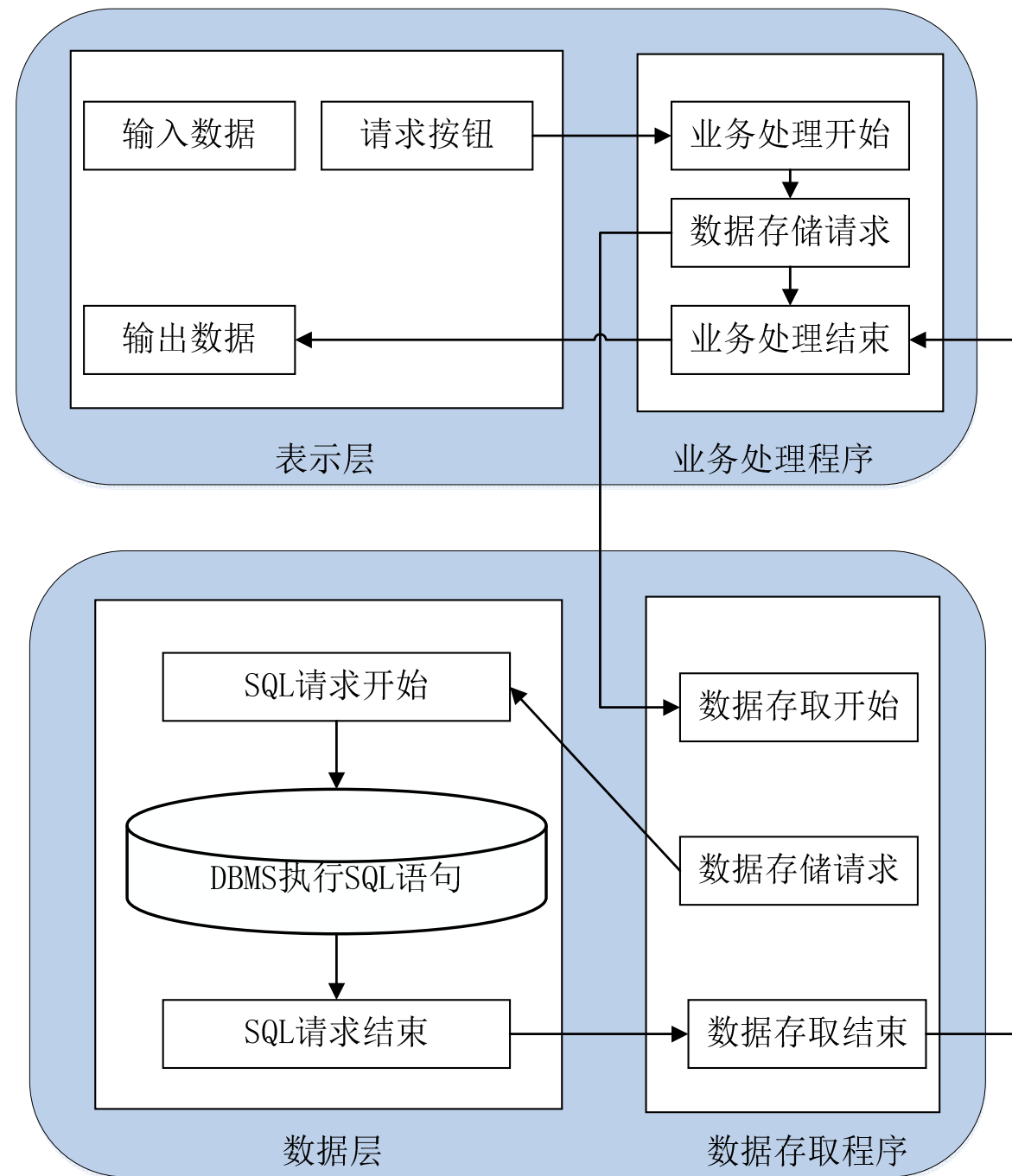


图4-30两层C/S架构的处理流程

4.3.11 客户机/服务器风格

- 两层C/S架构优点：
 - (1) 客户机组件和服务机组件分别运行在不同的计算机上，有利于分布式数据的组织和处理。
 - (2) 组件之间的位置是相互透明的
 - (3) 客户机程序和服务程序可运行在不同的操作系统上，便于实现异构环境和多种不同开发技术的融合。
 - (4) 软件环境和硬件环境的配置具有极大的灵活性，易于系统功能的扩展。
 - (5) 将大规模的业务逻辑分布到多个通过网络连接的低成本的计算机上，降低了系统的整体开销。

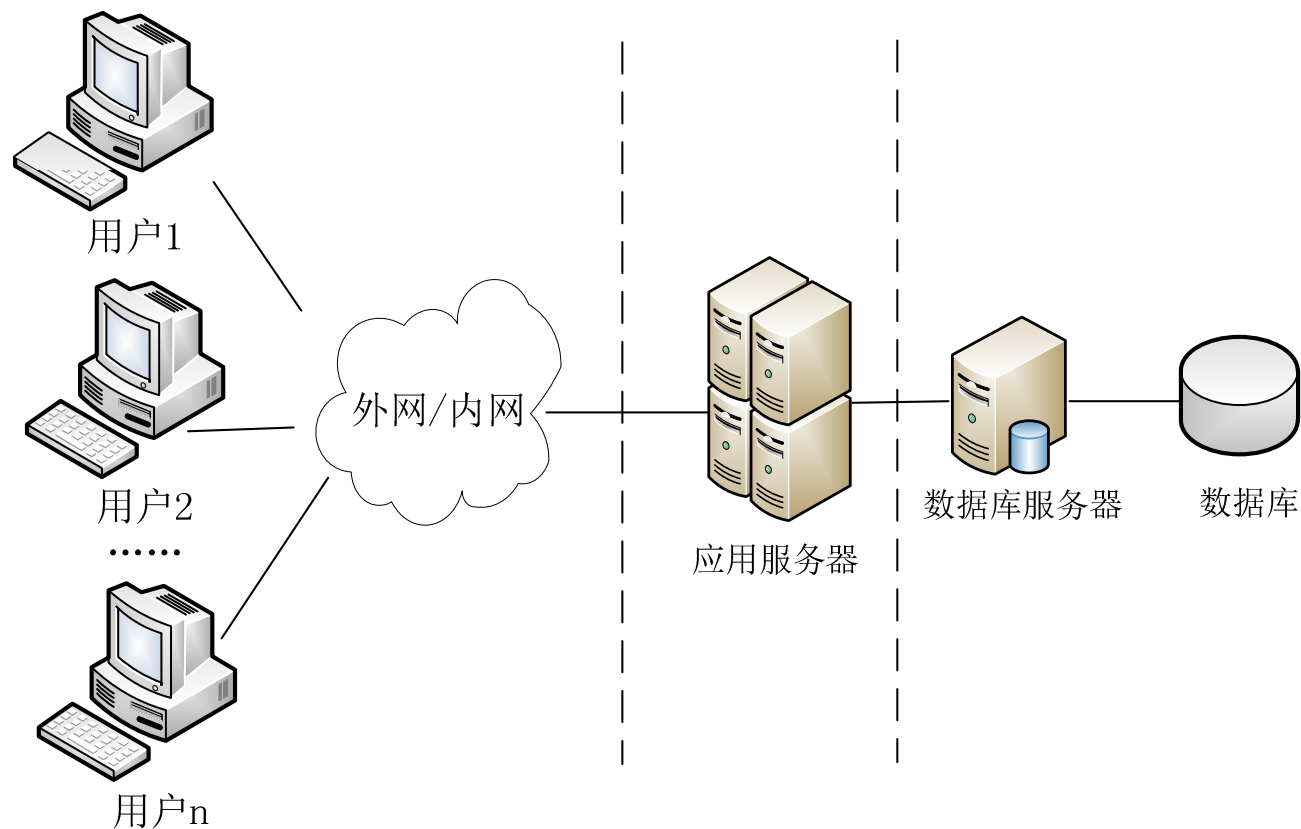
4.3.11 客户机/服务器风格

- 两层C/S架构缺点：
 - (1) 开发成本较高（客户机的软硬件要求高）。
 - (2) 客户机程序的设计复杂度大，客户机负荷重。
 - (3) 信息内容和形式单一。
 - (4) C/S架构升级需要开发人员到现场更新客户机程序，对运行环境进行重新配置，增加了维护费用。
 - (5) 两层C/S结构采用了单一的服务器，同时以局域网为中心，难以扩展到Internet。
 - (6) 数据安全性不高。

4.3.11 客户机/服务器风格

- 基本思想

- 三层C/S架构



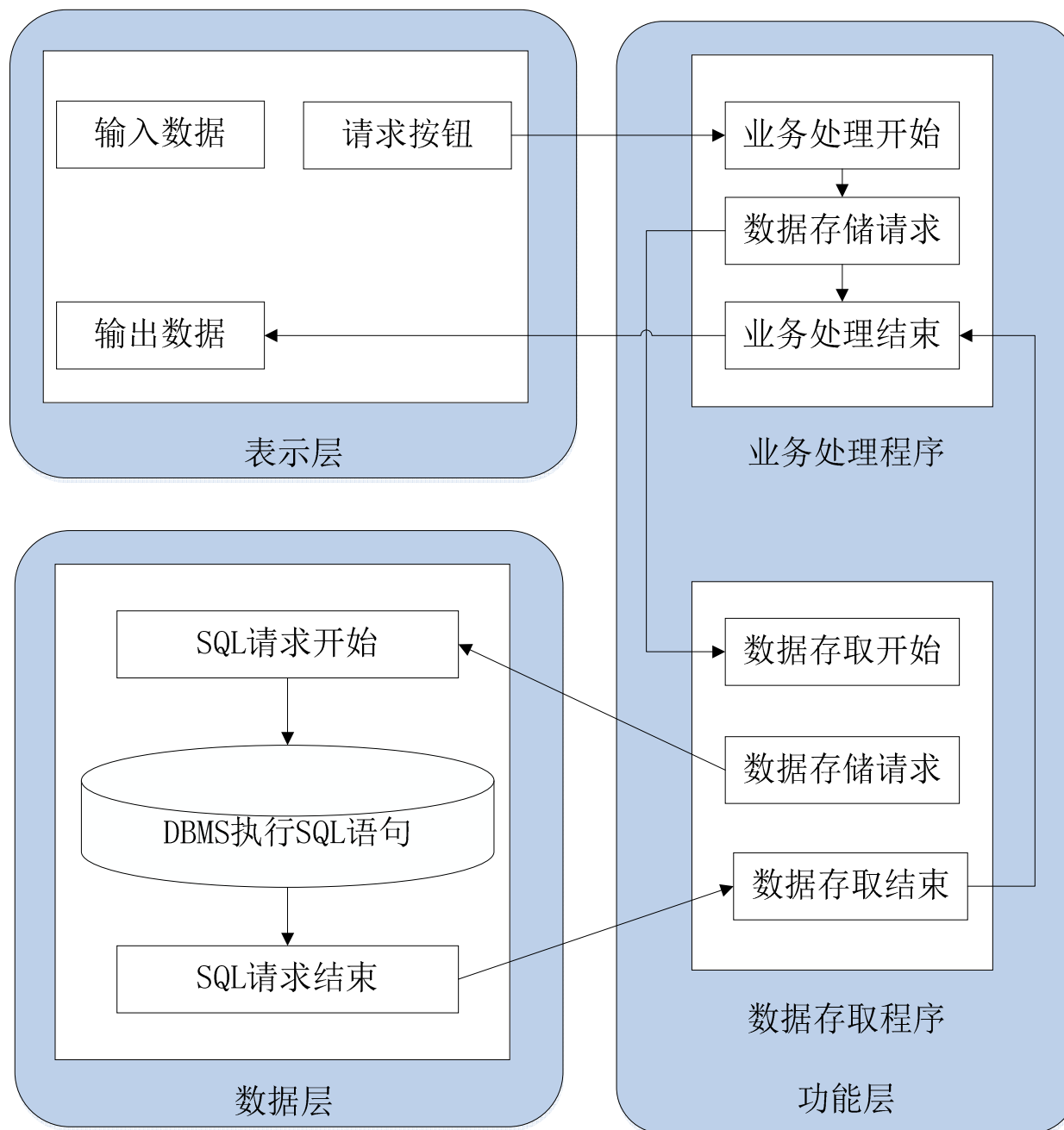


图4-32三层C/S架构的处理流程

4.3.11 客户机/服务器风格

- 三层C/S架构相比于两层C/S架构的优点：
 - (1) 合理地划分三层结构的功能，可以使系统的逻辑结构更加清晰，提高软件的可维护性和可扩充性。
 - (2) 在实现三层C/S架构时，可以更有效地选择运行平台和硬件环境，从而使每一层都具有清晰的逻辑结构、良好的负荷处理能力和较好的开放性。
 - (3) 在C/S架构中，可以分别选择合适的编程语言并行开发。
 - (4) 系统具有较高的安全性。

4.3.11 客户机/服务器风格

- 在使用三层C/S架构时需注意以下两个问题：
 - (1) 如果各层之间的通信效率不高，即使每一层的硬件配置都很高，系统的整体性能也不会太高。
 - (2) 必须慎重考虑三层之间的通信方法、通信频率和传输数据量，这和提高各层的独立性一样也是实现三层C/S架构的关键性问题。