软件架构与中间件

涂志莹 tzy_hit@hit.edu.cn

苏统华 thsu@hit.edu.cn

哈尔滨工业大学

软件架构与中间件

Software Architecture and Middleware

第2章

软件架构的传统风格

课程内容


软件架构与中间件的内涵及要点
软件架构的传统风格
软件架构的评审、测试与优化
软件架构与中间件
计算层的软件架构技术
表示层的软件架构技术
数据层的软件架构技术

第2章 软件架构的传统风格

2.1 软件架构风格概述

2.2 主程序-子过程风格

2.3 面向对象风格

2.4 数据流风格

2.5 事件驱动风格

2.6 解释器风格

2.7 分层结构

2.8 模型-视图-控制器

2.9 本章作业

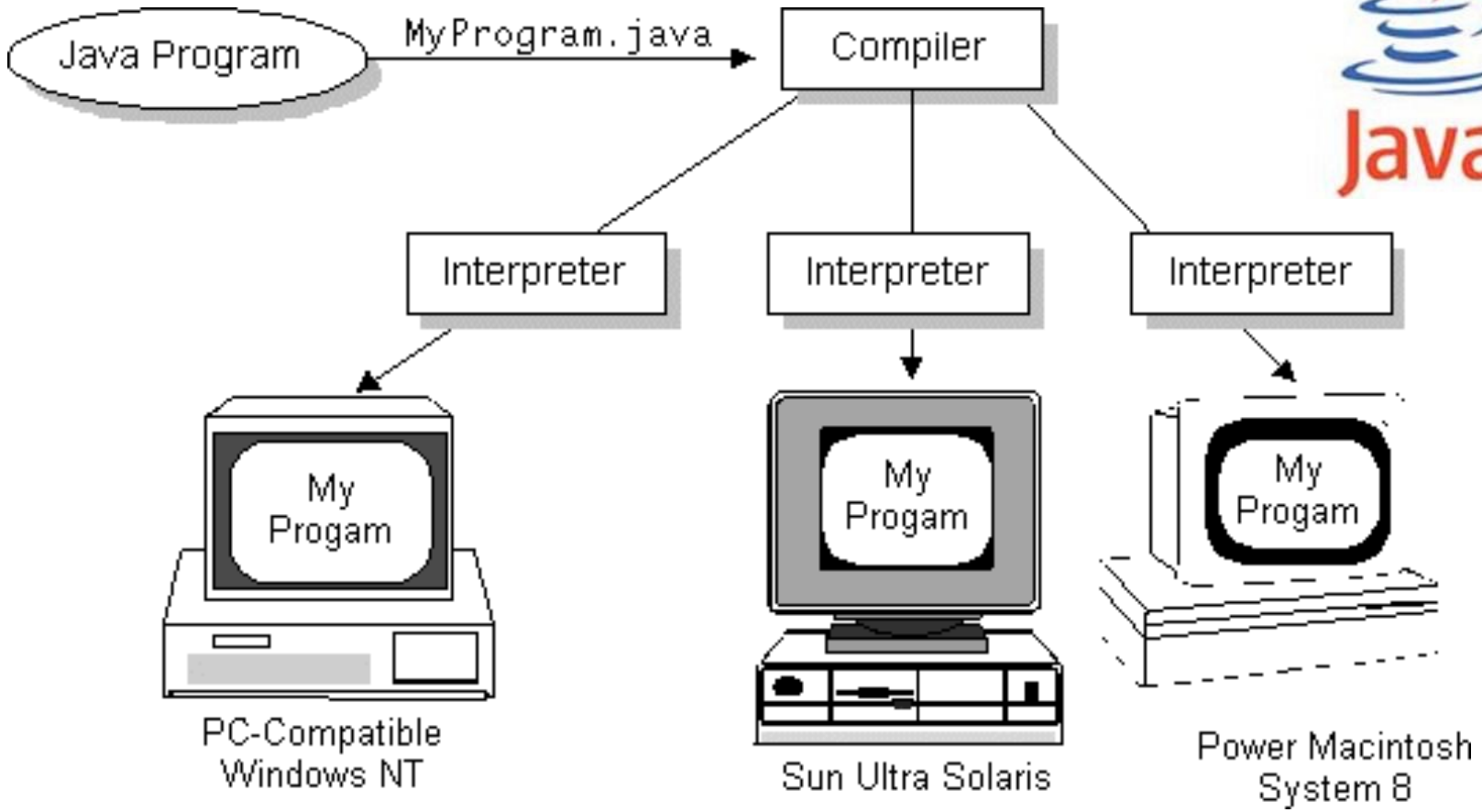
2.6
解释器风格

1、解释器风格的概念

2、解释器的分类

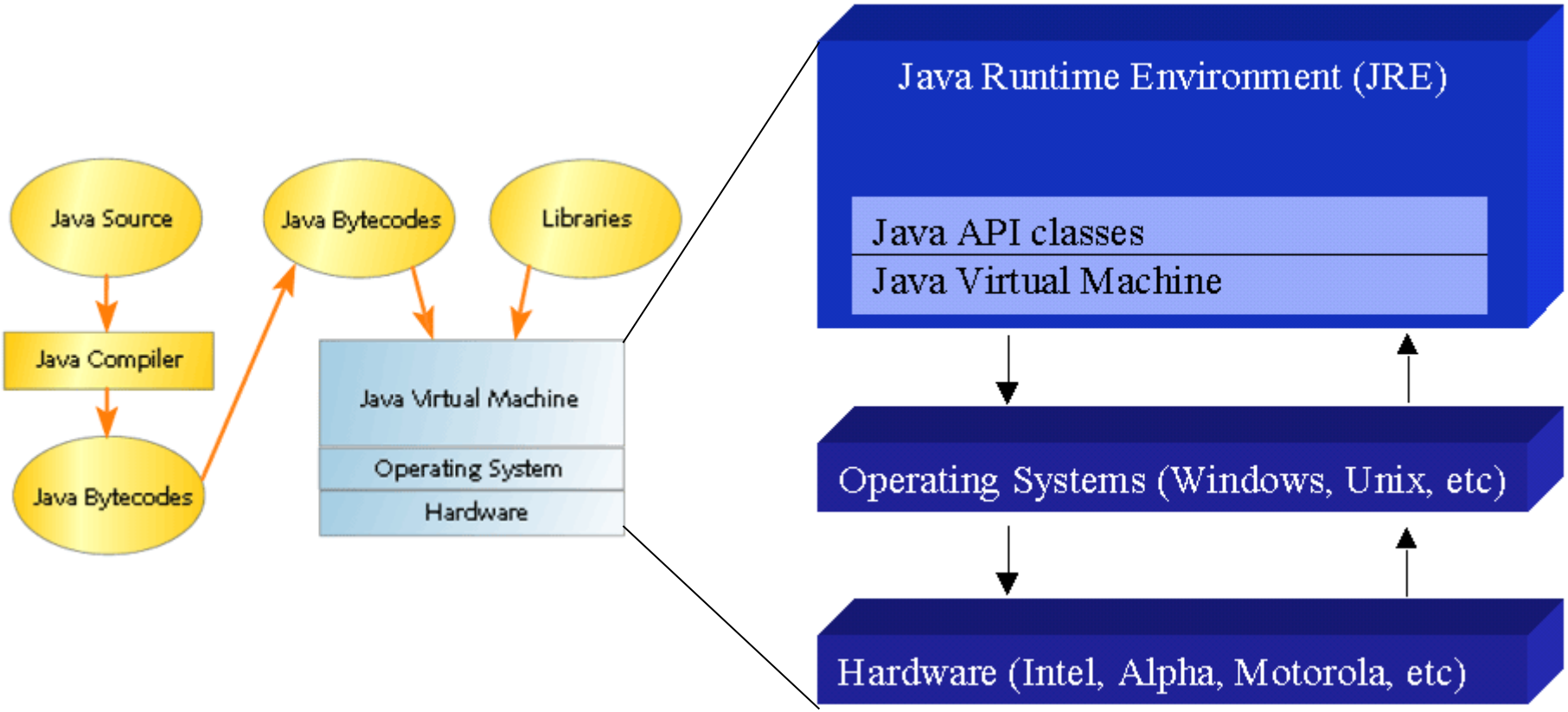
2.6.1 解释器风格概念

Java: Write Once, Run Anywhere

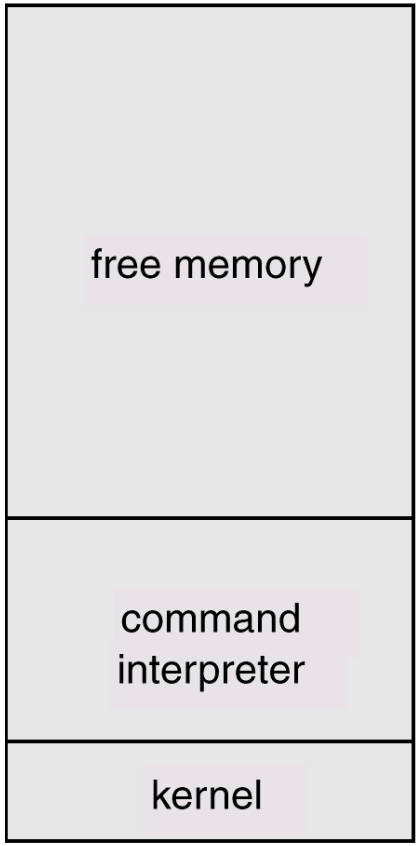


JAVA如何支持Platform Independence

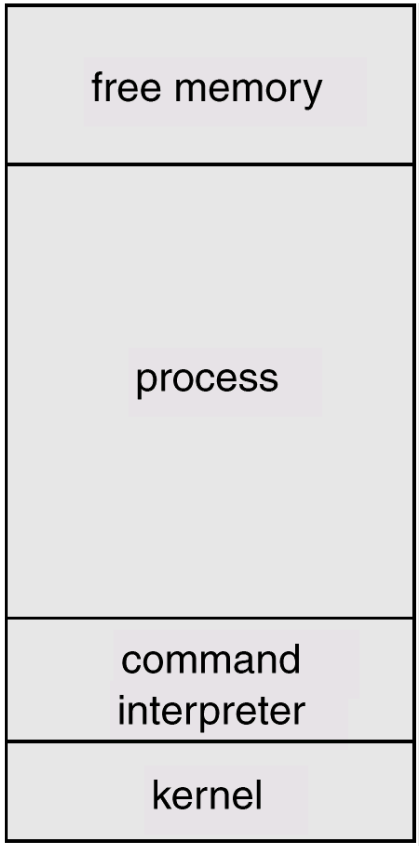
- Java虚拟机



MS-DOS的命令解释器



(a)



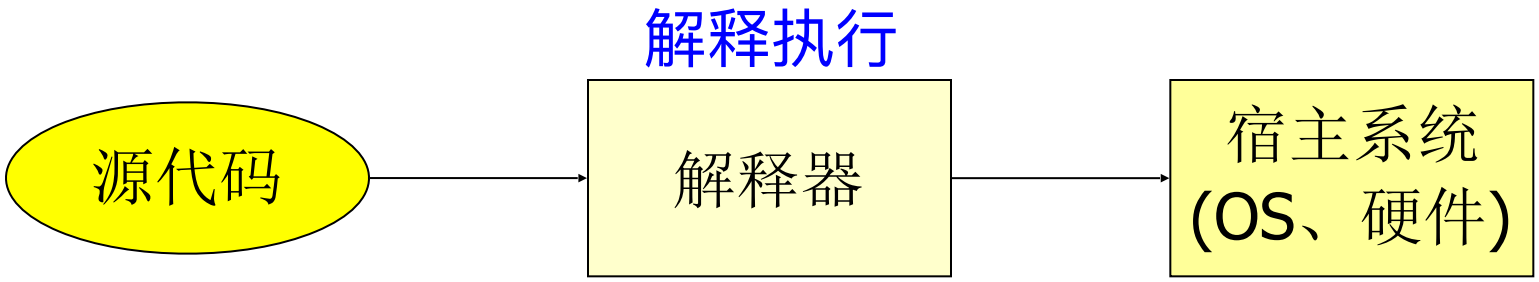
(b)

用户的命令行请求
(e.g., dir *.jsp /a /p)

操作系统内核的
执行指令？

解释器

- An interpreter is a program that executes another program (解释器是一个用来执行其他程序的程序).
  - An interpreter implements a virtual machine, which may be different from the underlying hardware platform. (解释器针对不同的硬件平台实现了一个虚拟机)
  - To close the gap between the computing engine expected by the semantics of the program and the computing engine available in hardware. (将高抽象层次的程序翻译为低抽象层次所能理解的指令，以消除在程序语言与硬件之间存在的语义差异)

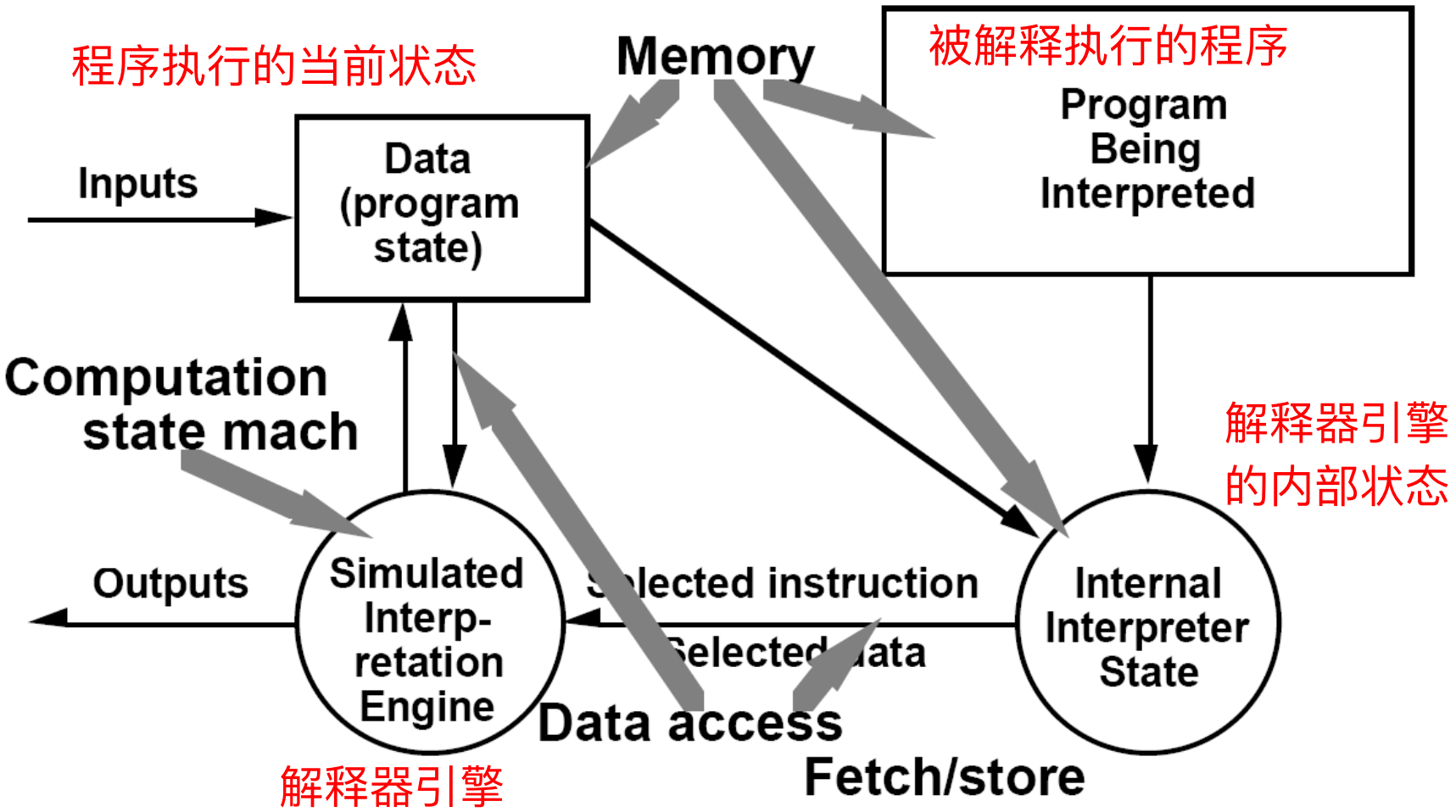


解释器

- 解释器通常用来在程序语言定义的计算和有效硬件操作确定的计算之间建立对应和联系。
  - 简单和小规模的解释器只完成基本的信息识别和转换
  - 复杂的解释器需要从词法到句法、到语法的复杂识别和处理

- 作为一种架构风格，解释器已经被广泛应用在从系统软件到应用软件的各个层面，
  - 包括各类语言环境、Internet浏览器、数据分析与转换等；
  - LISP、Prolog、JavaScript、VBScript、HTML、Matlab、数据库系统(SQL解释器)、各种通信协议等。

解释器的组成


程序执行的当前状态
Memory
被解释执行的程序
Program
Being
Interpreted
Inputs
Data
(program
state)
Computation
state mach
解释器引擎
的内部状态
Outputs
Simulated
Interp-
retation
Engine
lected instruction
elected ata
Internal
Interpreter
State
Data access
Fetch/store
解释器引擎

解释器风格

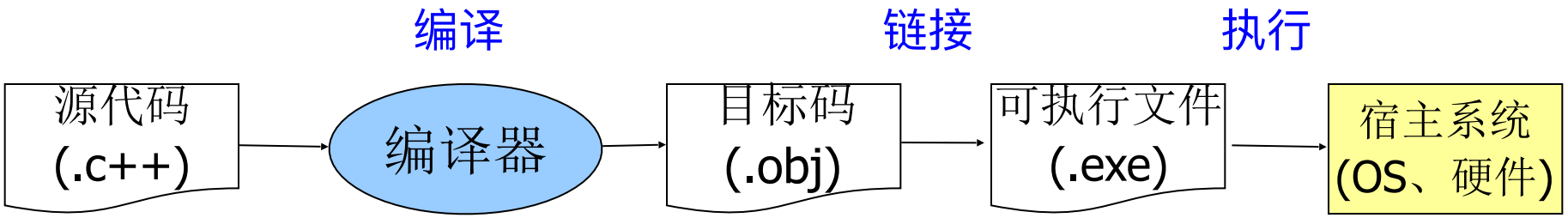
- 基本构件：
  - An interpretation engine (**解释器引擎**)
  - A Memory that contains(存储区):
    - The pseudo-code to be interpreted (**被解释的源代码**)
    - A representation of the control state of the interpretation engine (**解释器引擎当前的控制状态的表示**)
    - A representation of the current state of the program being simulated. (**程序当前执行状态的表示**)
- 连接器：
  - Data access (**对存储区的数据访问**)

解释器和编译器

- 解释器在软件中的应用由来已久，早期的程序语言环境就分为编译(Compilation)和解释(Interpretation)两大类。

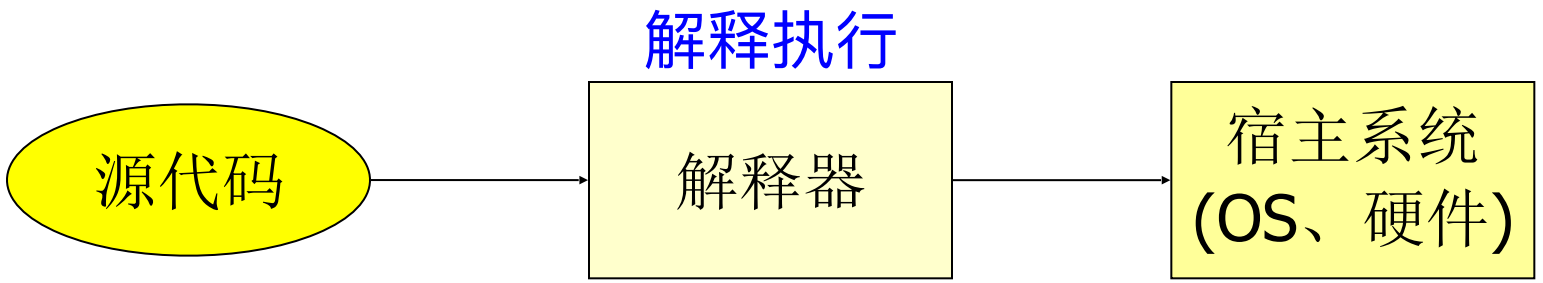
Compiler (编译器)

- This is in contrast to a compiler which does not execute its input program (the source code) but translates it into another language, usually executable machine code (also called object code) which is output to a file for later execution. (编译器不会执行输入的源程序代码，而是将其翻译为另一种语言，通常是可执行的机器码或目标码，并输出到文件中以便随后链接为可执行文件并加以执行)



Interpreter (解释器)

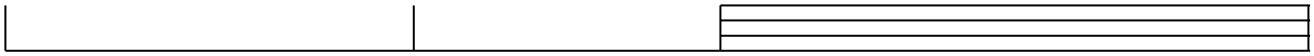
- It may be possible to execute the same source code either directly by an interpreter or by compiling it and then executing the machine code produced.（在解释器中，程序源代码被解释器直接加以执行。）



解释器和编译器

编译器

编译　　链接　　执行

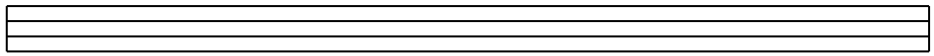


分析程序结构
源代码→目标码
确定内存分配与访问方案

分配内存
执行目标码(机器码)

解释器

解释执行



分析程序结构
确定内存分配与访问方案
分配内存
解释并执行程序

解释器和编译器

- It takes longer to run a program under an interpreter than to run the compiled code but it can take less time to interpret it than the total time required to compile and run it. (解释器的执行速度要慢于编译器产生的目标代码的执行速度，但是可能低于编译器“编译+链接+执行”的总时间)
- Interpreters are generally slower to run, but more flexible than compilers. Interpreters usually skip a linking and compilation step, enabling faster turn-around and decreasing cost of programmer time. (解释器通常省略了链接与编译的步骤，从而降低编程时间)
  - edit-interpret-debug (编辑源代码-解释-调试) vs
  - edit-compile-link-run-debug (编辑源代码-编译-链接-运行-调试)

解释器和编译器

注意二者区别

- Interpreting code is slower than running the compiled code because the interpreter must analyze each statement in the program each time it is executed and then perform the desired action whereas the compiled code just performs the action. (解析器执行速度之所以慢，是因为每次解释执行的时候，都需要分析程序的结构，而编译代码则直接执行而无需重复编译)

- Access to variables is also slower in an interpreter because the mapping of identifiers to storage locations must be done repeatedly at run-time rather than at compile time.(解释器对内存的分配是在解释时才进行的；而编译器则是在编译时进行，因此运行时直接将程序代码装入内存并执行即可)

2.6.2 解释器风格分类

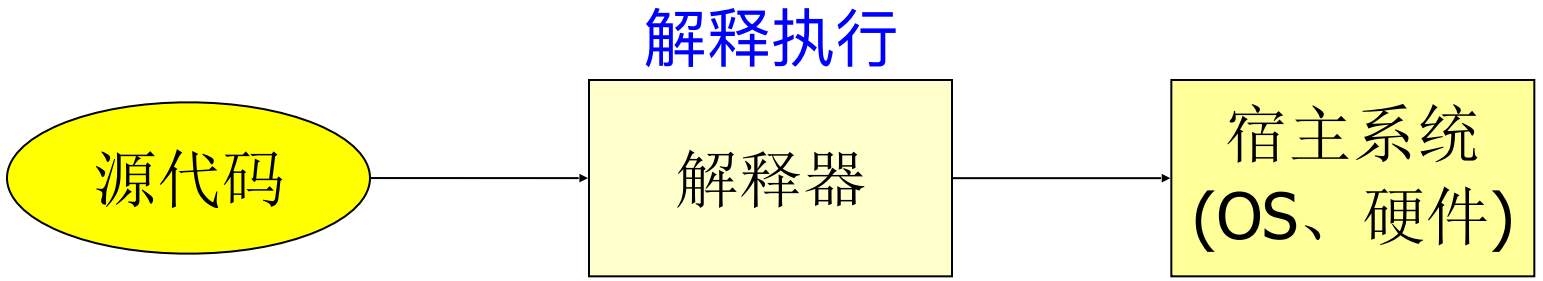
解释器的三种策略

- 传统解释器(traditionally interpreted)
  - ➢纯粹的解释执行
- 基于字节码的解释器 (compiled to bytecode which is then interpreted)
  - ➢编译→解释执行
- Just-in-Time (JIT)编译器
  - ➢编译||解释执行

传统解释器

- 解释器直接读取源代码并加以执行；
  - ASP
  - Excel
  - JavaScript
  - MATLAB
  - etc

记住这些例子



JVM中的解释器

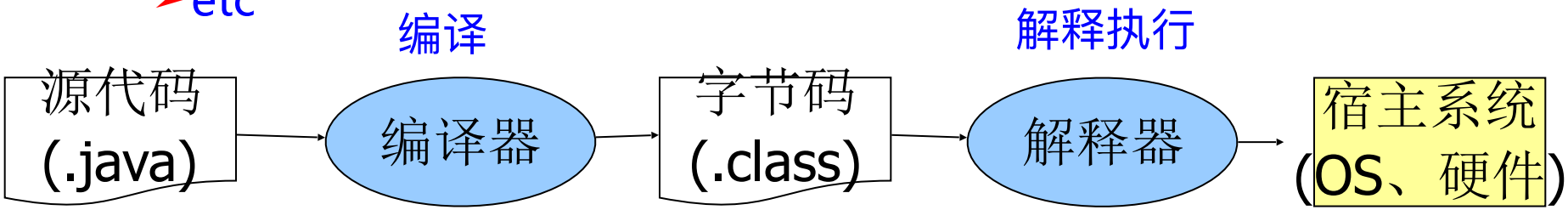
- Java的源程序不是直接交给解释器解释，而是先经过一个编译过程，把Java源程序翻译成一种特定的二进制字节码文件(Bytecode)，再把这个字节码文件交给Java解释器来解释执行；
  - The javac compiles Java source code to Java bytecode. (javac程序将Java源代码编译为字节码)
- Java编译器所生成的可执行代码可以不基于任何具体的硬件平台，而是基于JVM。
  - C/C++要的源程序要在不同的平台上运行，必须重新进行编译。

Bytecode interpreter (字节码解释器)

- 在该类解释器下，源代码首先被“编译”为高度压缩和优化的字节码，但并不是真正的机器目标代码，因而与硬件平台无关；
- 编译后得到的字节码然后被解释器加以解释；
- 例如：
  - Java
  - Perl
  - PHP
  - Python
  - etc

一个依赖具体硬件平台，一个不依赖

Quiz：
字节码解释器中所产生的中间字节码与编译器中产生的目标码有何区别?

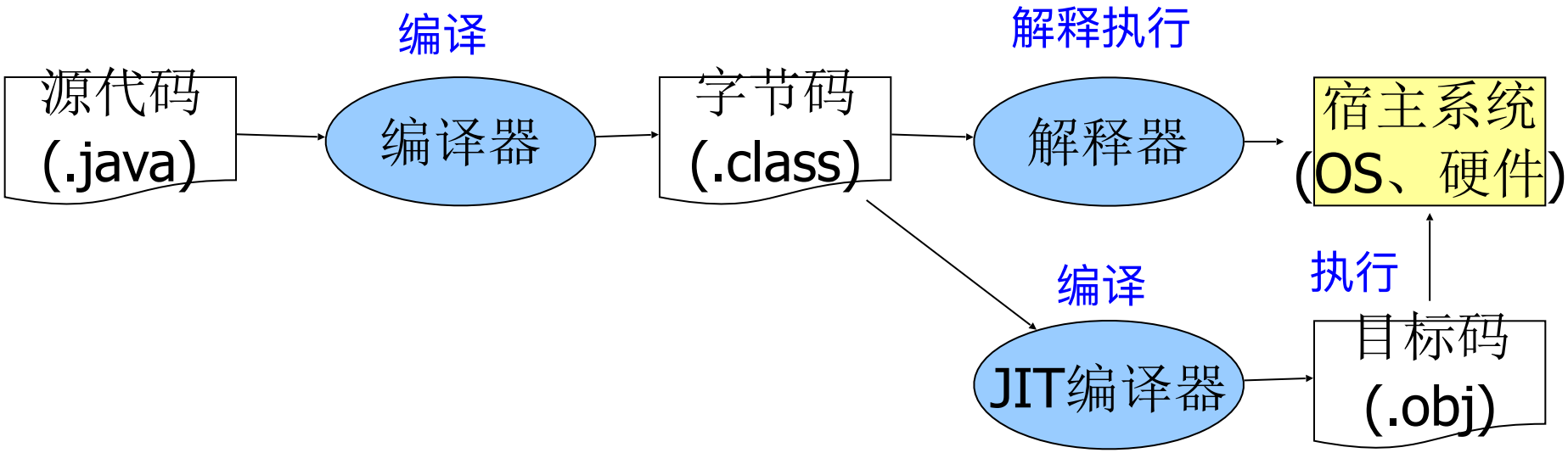


Just-in-time compilation (实时编译)

- Just-in-time compilation (JIT), refers to a technique where bytecode is compiled to native machine code at runtime (实时编译JIT中，字节码在运行时被编译为本机的目标代码)
  - In a JIT environment, bytecode compilation is the first step, reducing source code to a portable and optimizable intermediate representation. (第一步是编译得到字节码)
  - The bytecode is deployed onto the target system. (字节码被配置到目标系统中)
  - When the code is executed, the runtime environment 's compiler translates it into native machine code. (**当字节码被执行时，运行环境下的编译器将其翻译为本地机器码**)

Just-in-time compilation (实时编译)

- It has gained attention in recent years, which further blurs the distinction between interpreters, byte-code interpreters and compilation. (JIT模糊了解释器、字节码解释器和编译器之间的边界与区分)
  - ➢JIT is available for both the .NET and Java platforms.



JIT: Deciding what to Compile

- This can be done on a per-file or per-function basis: functions can be compiled only when they are about to be executed (hence the name "just-in-time"). (只有当某个函数要被执行时，才被编译，因此称为JIT)
- JIT compiles only those methods that contain frequently executed ("hot") code: (而且，JIT并不是编译全部的代码，而是只编译那些被频繁执行的代码段)
  - methods that are called a large number of times (被执行多次的方法);
  - methods containing loops with large iteration counts (包含多次循环的方法).

总结

- 解释器的三种策略：
  - 传统解释器(traditionally interpreted)
  - 基于字节码的解释器 (compiled to bytecode which is then interpreted)
  - Just-in-Time (JIT)编译器

2.7
分层结构

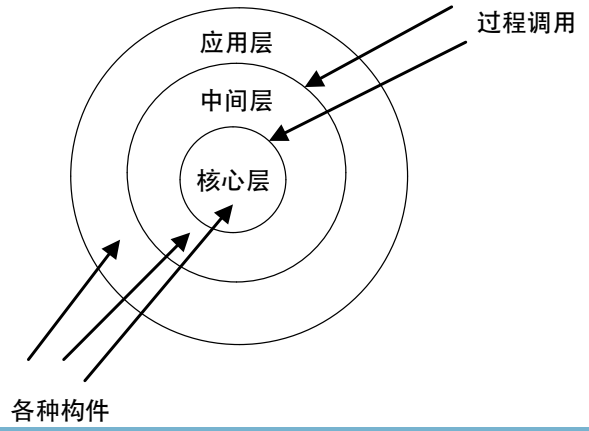
1、分层风格概述

2、分层类型

3、案例：C/S、B/S

Layered System

- A layered system is a system in which
  - components are grouped, i.e., layered, in a hierarchical arrangement (在层次系统中，系统被组织成若干个层次，每个层次由一系列构件组成)
  - lower layers provide functions and services that support the functions and services of higher layers (**下层构件向上层构件提供服务**)
  - higher layers serves as a client to the layer below (上层构件被看作是下层构件的客户)



Layered System

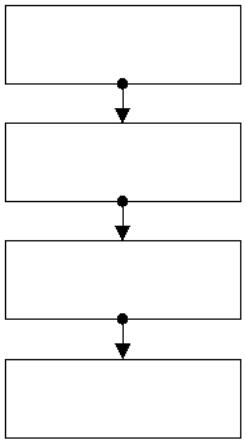
- Layering Principles(分层原则)
  - Separation of concerns(分离关注), Break your application into distinct features that overlap in functionality as little as possible.尽量减少功能重叠
  - Abstraction(抽象),Concentrating on the essential, force out the irrelevant.删枝节留主干
  - Hiding(隐藏),Restriction to the visibility of details to those parts of a system that need them.只暴露需要访问的接口

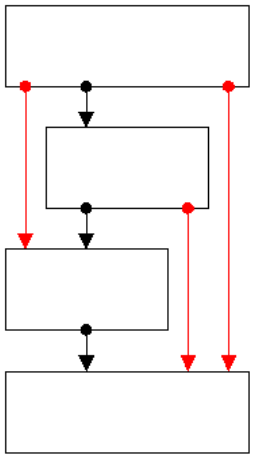
Layered System

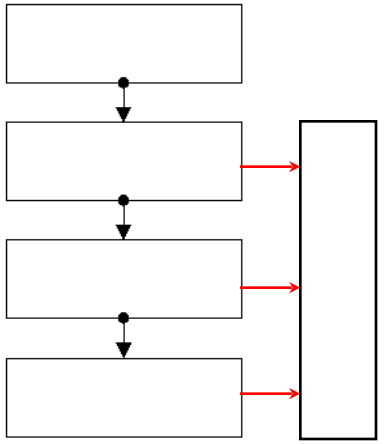
- The components are defined in each layer. (层次系统的基本构件：各层次内部包含的构件)
- The connectors are defined by the protocols that determine how the layers will interact.(连接件：层间的交互协议)
- Topology: layered structure. (拓扑结构：分层)
- Topological constraints include limiting interactions to adjacent layers.(拓扑约束：对相邻层间交互的约束)
  - Mainframe(集中式部署)
  - Distributed(分布式部署)

分层模式

- 某一层中的构件一般只与同一级别中的对等实体或较低级别中的构件交互，这种单向交互有助于减少不同级别中的构件之间的依赖性。(思考：下层构件如何与上层构件交互？)
- 分层模式：
  - 严格分层(Strict System Layering)
  - 松散分层(Loosely System Layering)
  - 横切关注(Cross-Cutting Concerns)



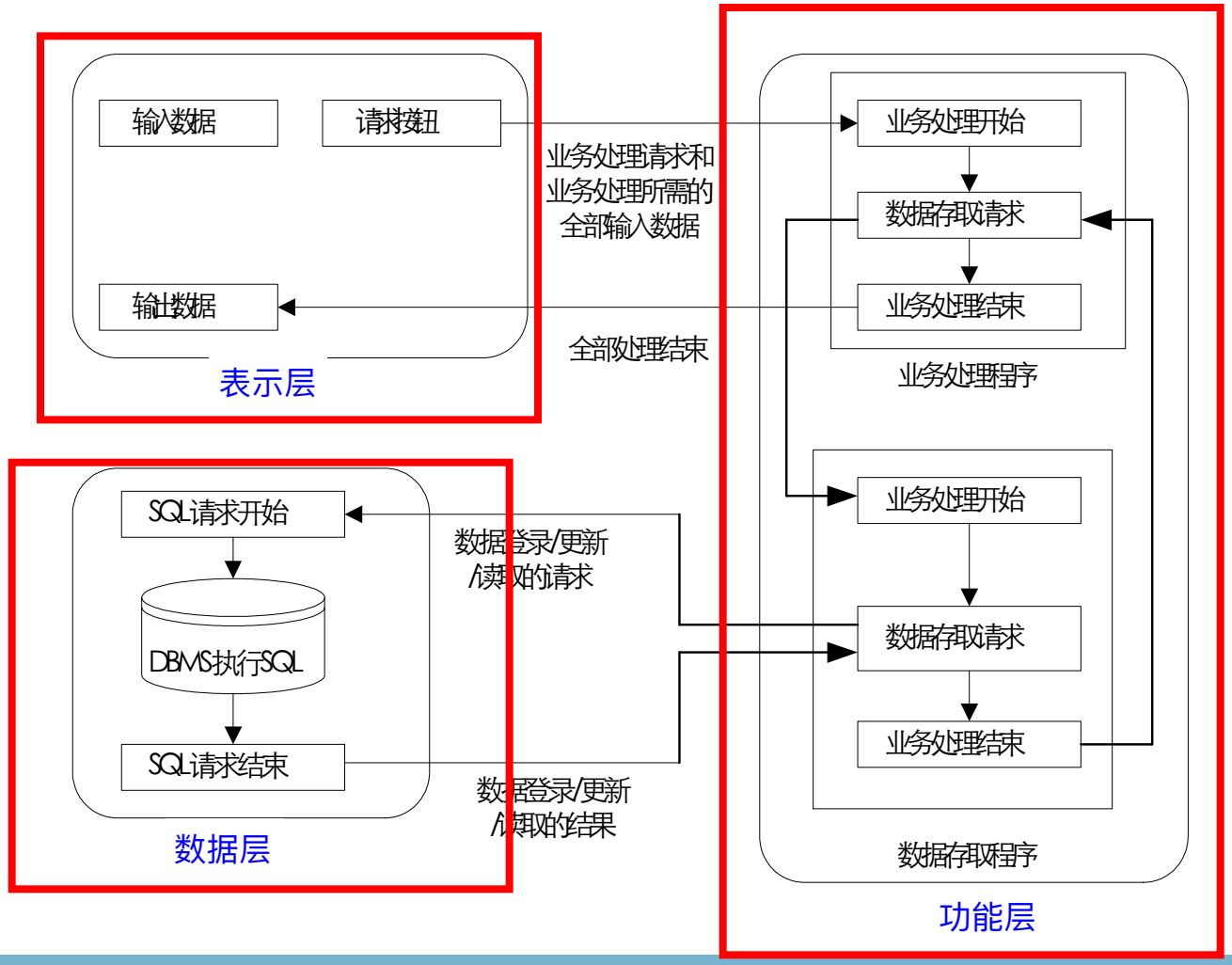




三层C/S结构(3-Tier C/S Architecture)

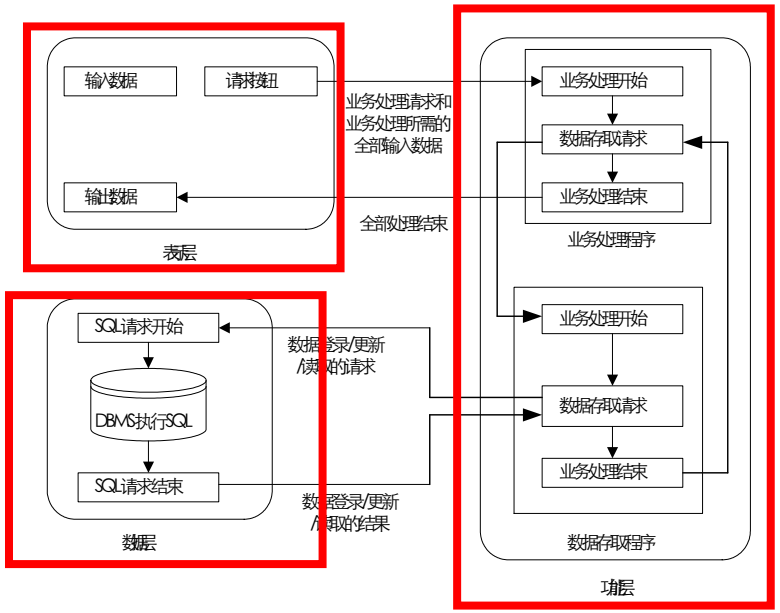
- In the three tier architecture, a middle tier was added between the user system interface client environment and the database management server environment. (在客户端与数据库服务器之间增加了一个中间层)
  - There are a variety of ways of implementing this middle tier, such as transaction processing monitors, message servers, or application servers. (中间层可能为：事务处理监控服务器、消息服务器、应用服务器等)
  - The middle tier can perform queuing, application execution, and database staging. (中间层负责消息排队、业务逻辑执行、数据中转等功能)

三层C/S结构(3-Tier C/S Architecture)



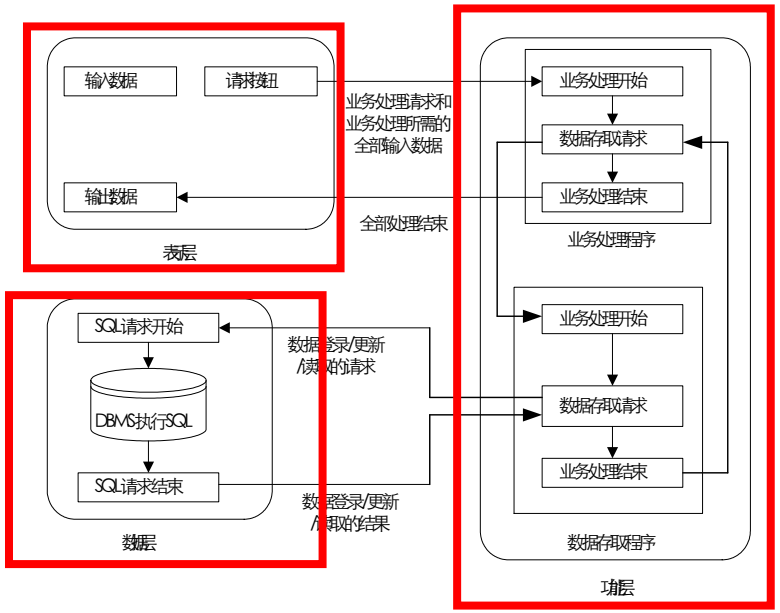
表示层

- 用户接口部分，担负着用户与应用之间的对话功能；
- 检查用户的输入，显示应用的输出；
- 通常使用GUI；
- 在变更时，只需要改写显示控制和数据检查程序，而不影响其他层；
- 不包含或包含一部分业务逻辑。



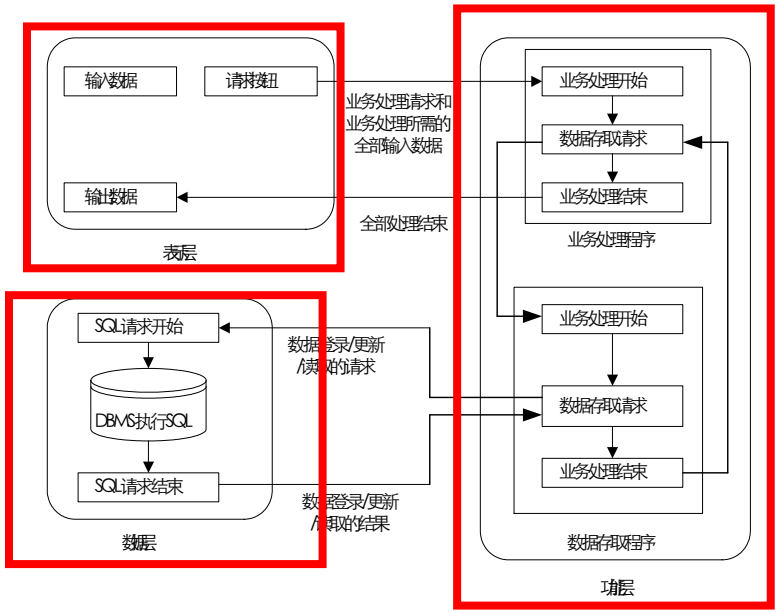
功能层

- 应用系统的主体，包括大部分业务处理逻辑 (通常以业务构件的形式存在，如JavaBean/EJB/COM等)；
- 从表示层获取用户的输入数据并加以处理；
- 处理过程中需要从数据层获取数据或向数据层更新数据；
- 处理结果返回给表示层。



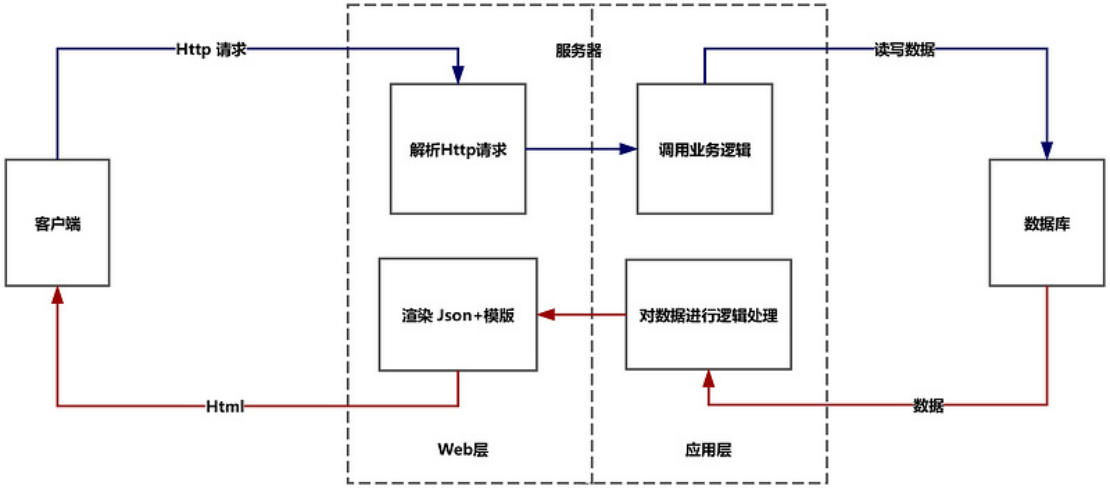
数据层

- DBMS；
- 接受功能层的数据查询请求，执行请求，并将查询结果返回给功能层；
- 从功能层接受数据存取请求，并将数据写入数据库；
- 请求的执行结果也要返回给功能层。



浏览器/服务器(Browser/Server Architecture)

- 浏览器/服务器(B/S)是三层C/S风格的一种实现方式。
  - ➢表现层：浏览器
  - ➢逻辑层：
    - Web服务器
    - 应用服务器
  - ➢数据层：数据库服务器



B/S结构的优点

- 基于B/S架构的软件，系统安装、修改和维护全在服务器端解决，系统维护成本低：
  - ➢客户端无任何业务逻辑，用户在使用系统时，仅仅需要一个浏览器就可运行全部的模块，很容易在运行时自动升级。
  - ➢良好的灵活性和可扩展性：对于环境和应用条件经常变动的情况，只要对业务逻辑层实施相应的改变，就能够达到目的。
- 较好的安全性：在这种结构中，客户应用程序不能直接访问数据，应用服务器不仅可控制哪些数据被改变和被访问，而且还可控制数据的改变和访问方式 。
- 三层模式成为真正意义上的“瘦客户端”，从而具备了很高的稳定性、延展性和执行效率。
- 三层模式可以将服务集中在一起管理，统一服务于客户端，从而具备了良好的容错能力和负载平衡能力。

B/S结构的缺点

- 客户端浏览器以同步的请求/响应模式交换数据，每请求一次服务器就要刷新一次页面；
- 受HTTP协议“基于文本的数据交换”的限制，在数据查询等响应速度上，要远远低于C/S架构；
- 数据提交一般以页面为单位，数据的动态交互性不强，不利于在线事务处理(OLTP)应用；
- 受限于HTML的表达能力，难以支持复杂GUI (如报表等)。

第2章

软件架构的传统风格

Thanks for listening

涂志莹、苏统华
哈尔滨工业大学计算机学院
企业与服务计算研究中心