第3章 软件体系结构风格

内容

- 3.1 概述
- 3.2 数据流风格
- 3.3 过程调用风格
- 3.4 独立构件风格
- 3.5 层次风格
- 3.6 虚拟机风格
- 3.7 客户/服务器风格
- 3.8 表示分离风格
- 3.9 插件风格
- 3.10 微内核风格
- 3.11 SOA风格

虚拟机风格

- 3.6.1 虚拟机(Virtual Machine)
- 3.6.2 解释器(Interpreter)
- 3.6.3 基于规则的系统(Rule-based System)

3.6.1 虚拟机(Virtual Machine)

从Java入手...

问题

你认为JAVA最吸引人的特性是什么？

Object Oriented

Automatic Memory Management

Security

Built-in Networking

Robust

Compiler/Interpreter Combo

Write Once, Run Anywhere

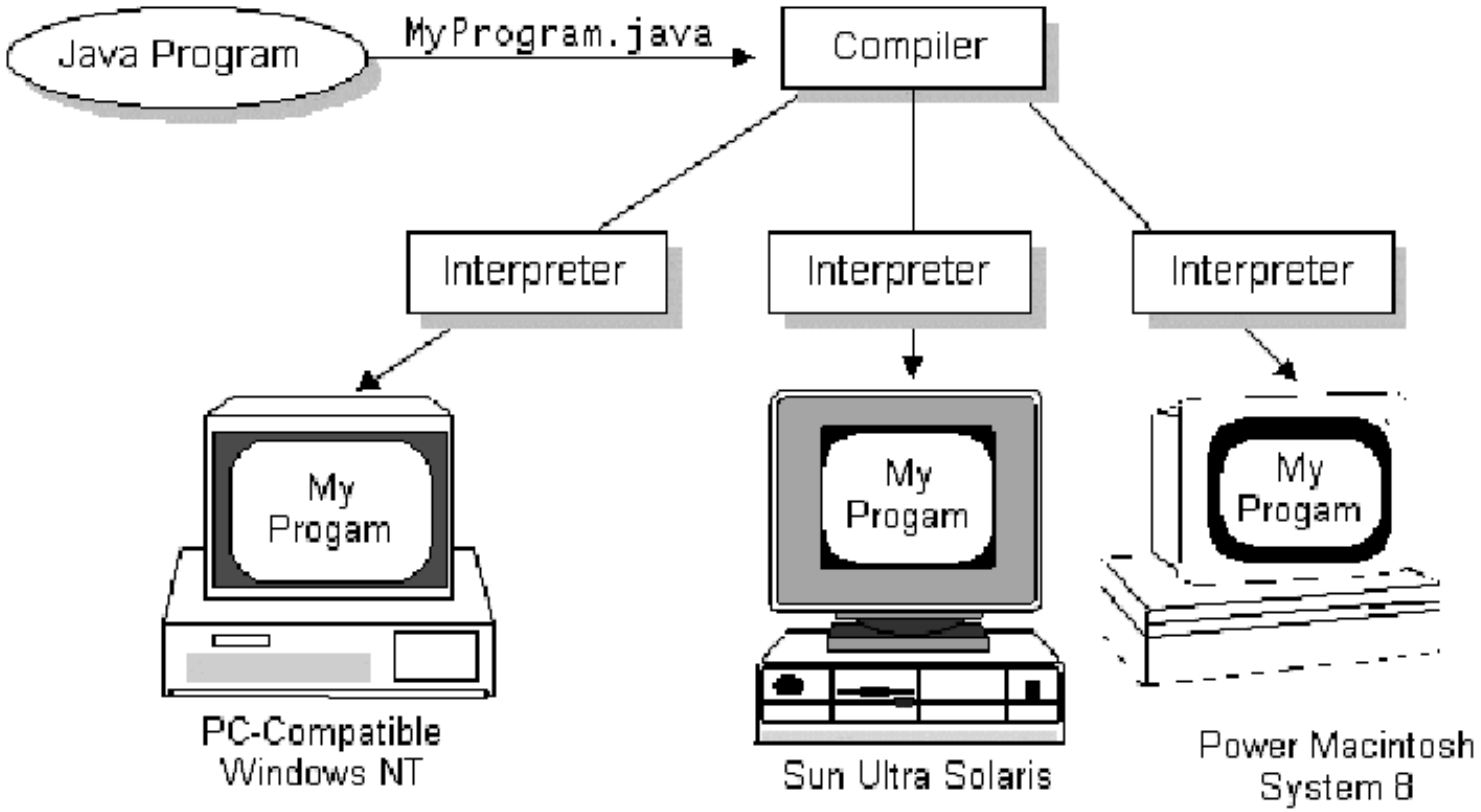
Platform Independence

Dynamic Binding

Good Performance

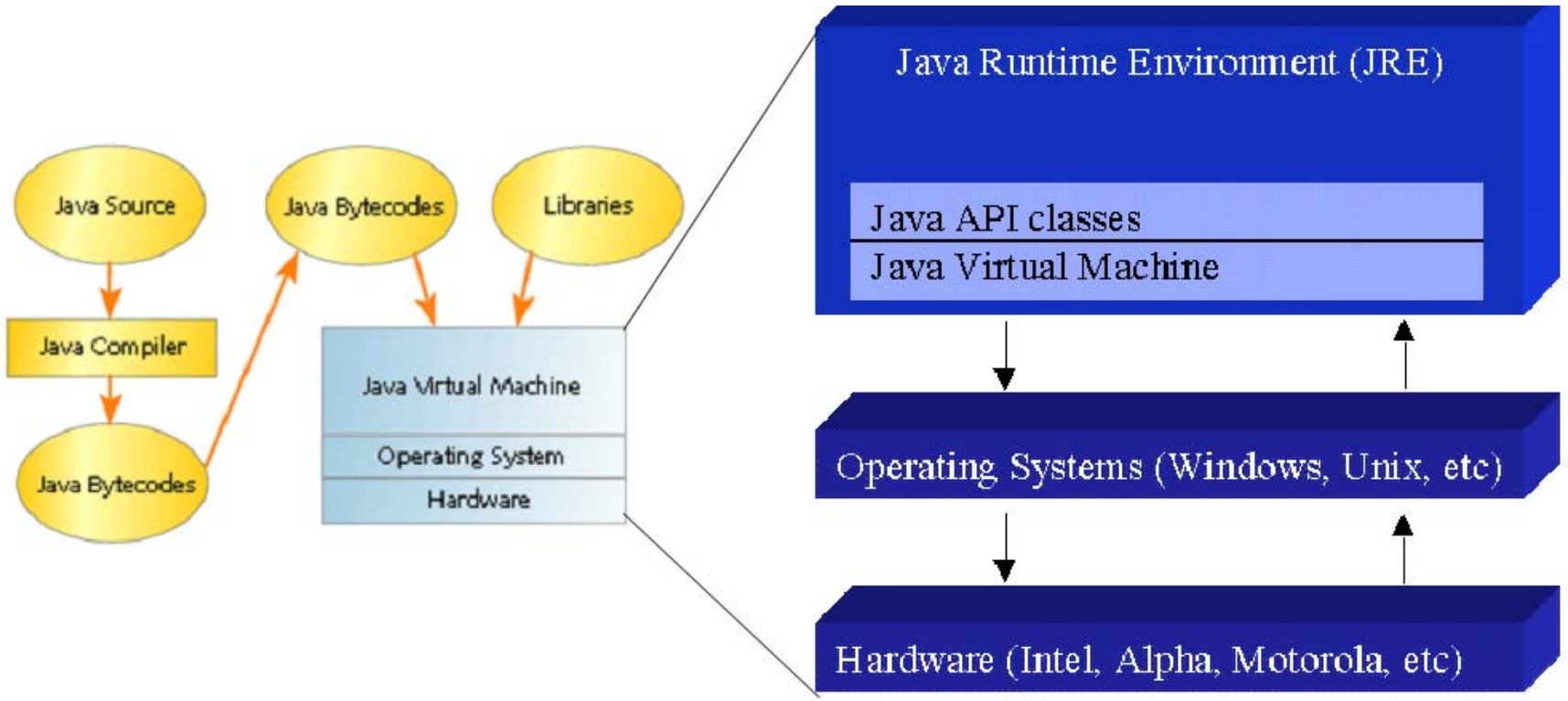
Several dangerous features of C&C++ eliminated

Write Once, Run Anywhere
一次书写，多处运行



JAVA如何支持
Platform Independence

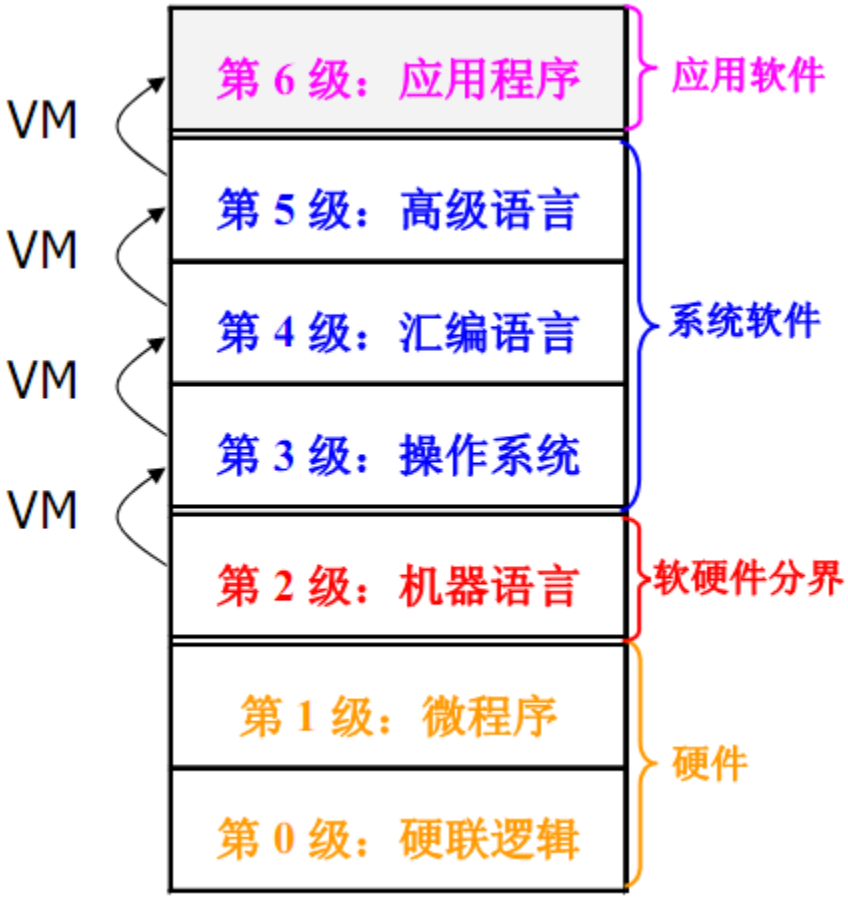
Java虚拟机



虚拟机
Virtual Machine

What is Virtual Machine?

- A virtual machine is software that creates a virtualized environment between the computer platform and the end user in which the end user can operate software.



- 虚拟机是一种软件；
- 它创建了一种虚拟的环境；
- 将用户与底层平台隔离开来。

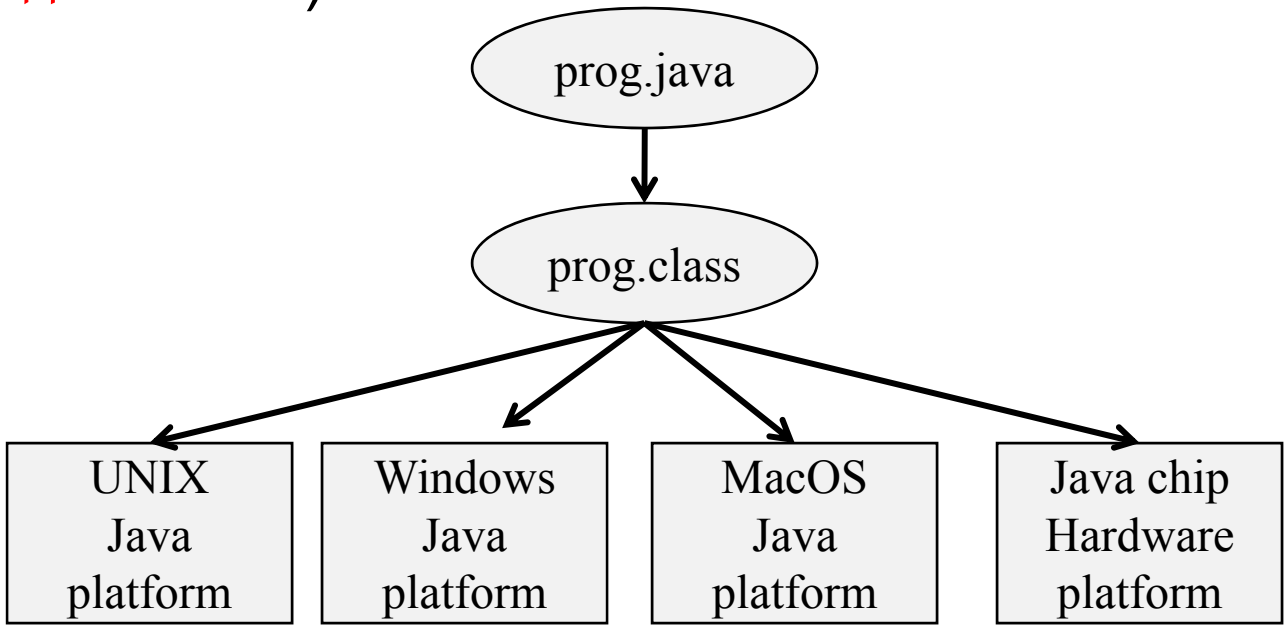
Java Virtual Machine (JVM)

- A Java Virtual Machine (JVM), originally developed by Sun Microsystems, is a virtual machine that executes Java byteCode. (JVM：执行Java字节码的虚拟机)

- The availability of JVMs on many types of hardware and software platforms enables Java to function both as middleware and a platform in its own right. Hence the expression “Write once, run anywhere.” (JVM可适应所有的硬件与OS平台，从而使得Java具有“一次书写，到处运行”的能力)

JVM的标准输入：Java Class

- Programs intended to run on a JVM must be compiled into a standardized portable binary format, which typically comes in the form of .class files. (在JVM上运行的程序必须首先被编译为标准的二进制格式的文件：.class)

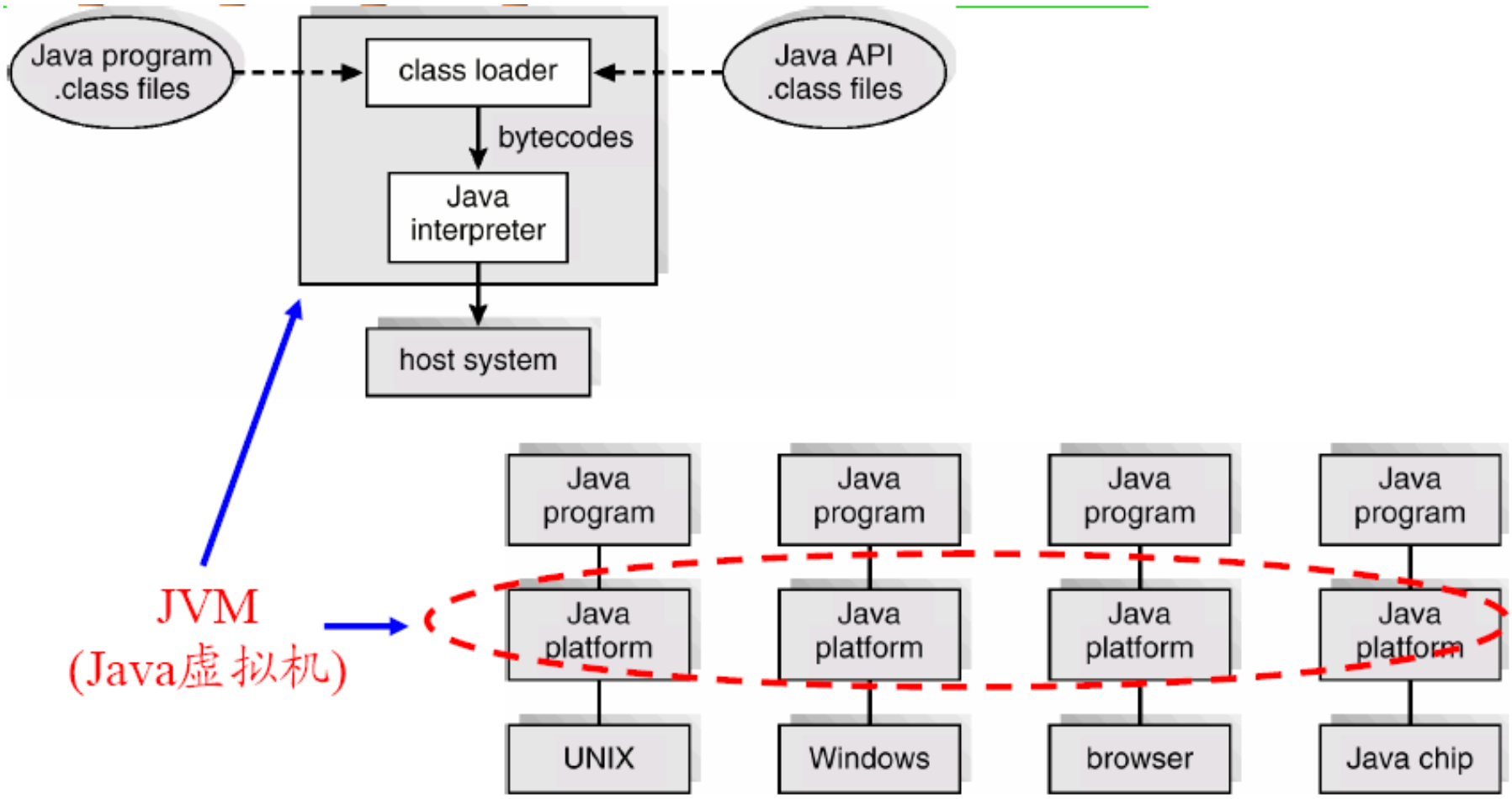


Java Class在JVM上的执行

- 注意：Java class文件并不是机器代码或目标代码，而是一种具有标准中间格式的二进制文件，无法直接在任何OS平台上执行；
- Java Class必须在JVM的支持下才能真正执行；

- This binary is then executed by the JVM runtime which carries out emulation of the JVM instruction set by interpreting it or by applying a just-in-time compiler (JIT). (java class文件在JVM下运行，运行策略可能为：解释器或**JIT**编译器)

JVM的执行机制

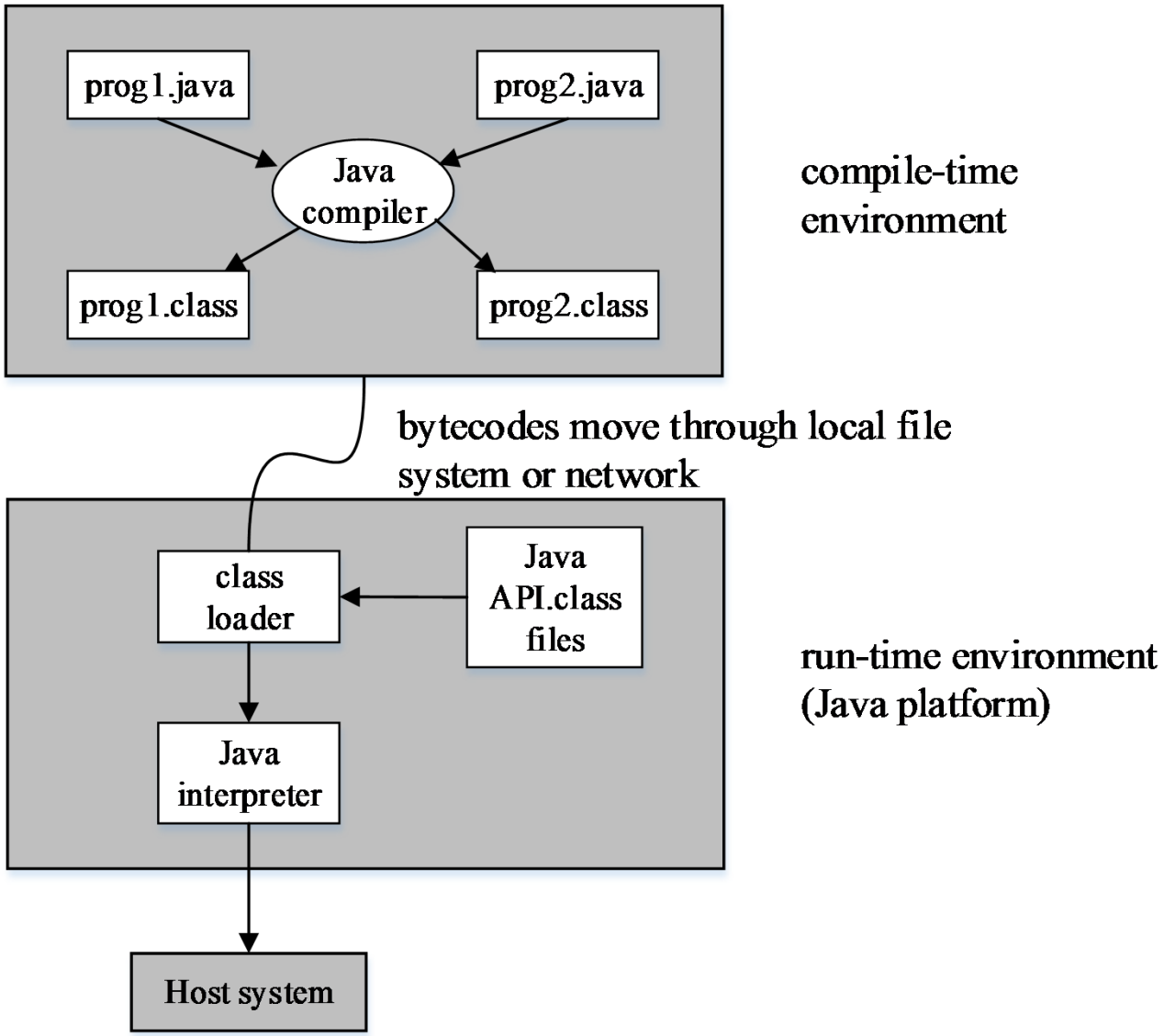

Java program
.class files
class loader
Java API
.class files
bytecodes
Java
interpreter
host system
JVM
(Java虚拟机)
Java
program
Java
program
Java
program
Java
program
Java
platform
Java
platform
Java
platform
Java
platform
UNIX
Windows
browser
Java chip

JVM的功能与作用

- Each particular host operating system needs its own implementation of the JVM and runtime. (每个特定的OS需要实现自己的JVM)
- The JVM has instructions for the following groups of tasks: (在执行具体的class代码之前，JVM需要针对特定的OS环境，进行以下转换)
  - **Load and store**
  - **Type conversion**
  - **Object creation and manipulation**
  - **Operand stack management (push / pop)**
  - **Control transfer (branching)**
  - **Method invocation and return**
  - **Throwing exceptions**

将JAVA CLASS代码转化为
特定OS所能支持的程序运行模式

总结：基于JVM的Java程序开发与执行过程



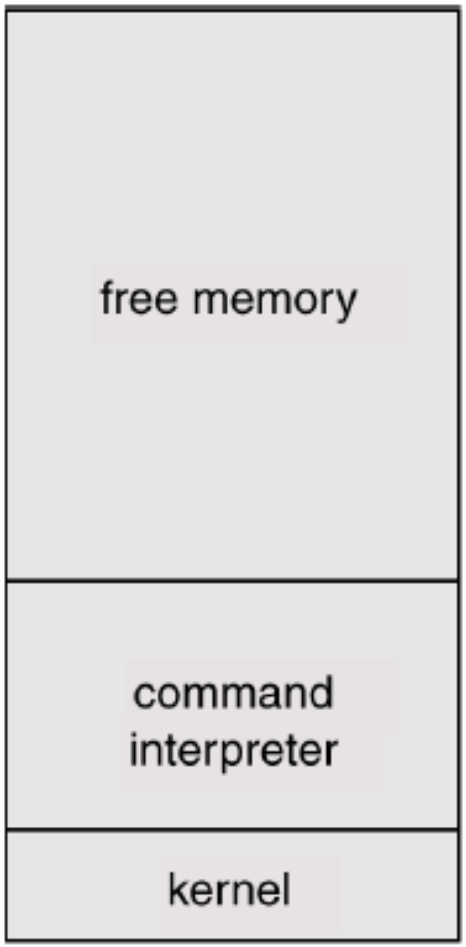
3.6.2 解释器(Interpreter)

问题

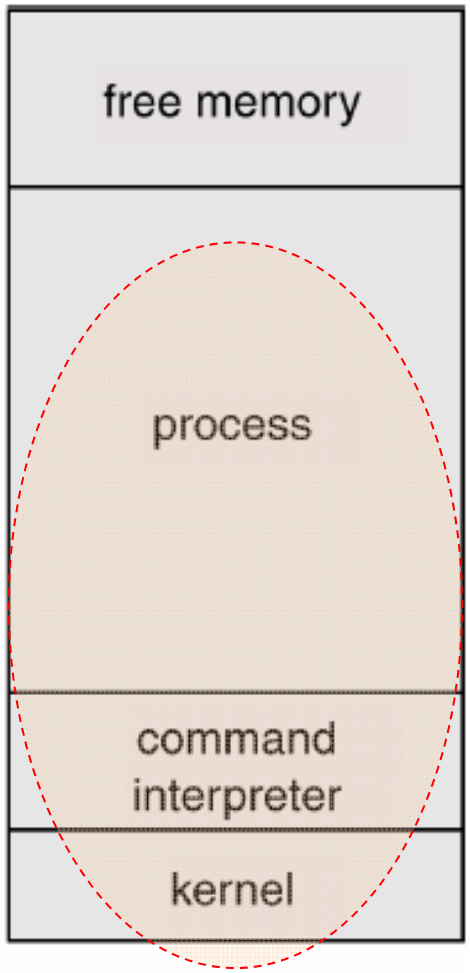
- 不管何种类别的虚拟机，本质上都是在高层次抽象的用户与低层次抽象的OS/硬件之间建立一道屏障。

- 但是，如何把上层应用的请求映射到下层OS/硬件系统的执行？
  - 解释器(Interpreter)
  - 基于规则的系统(Rule-based System)

MS-DOS的命令解释器



(a)



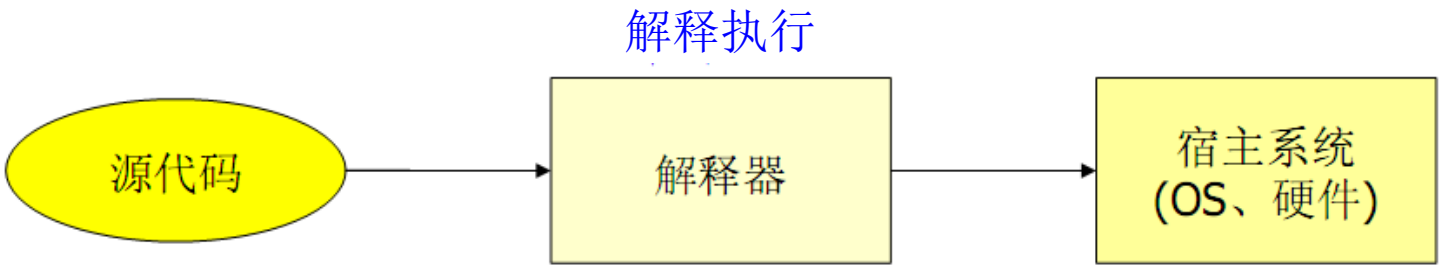
(b)

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

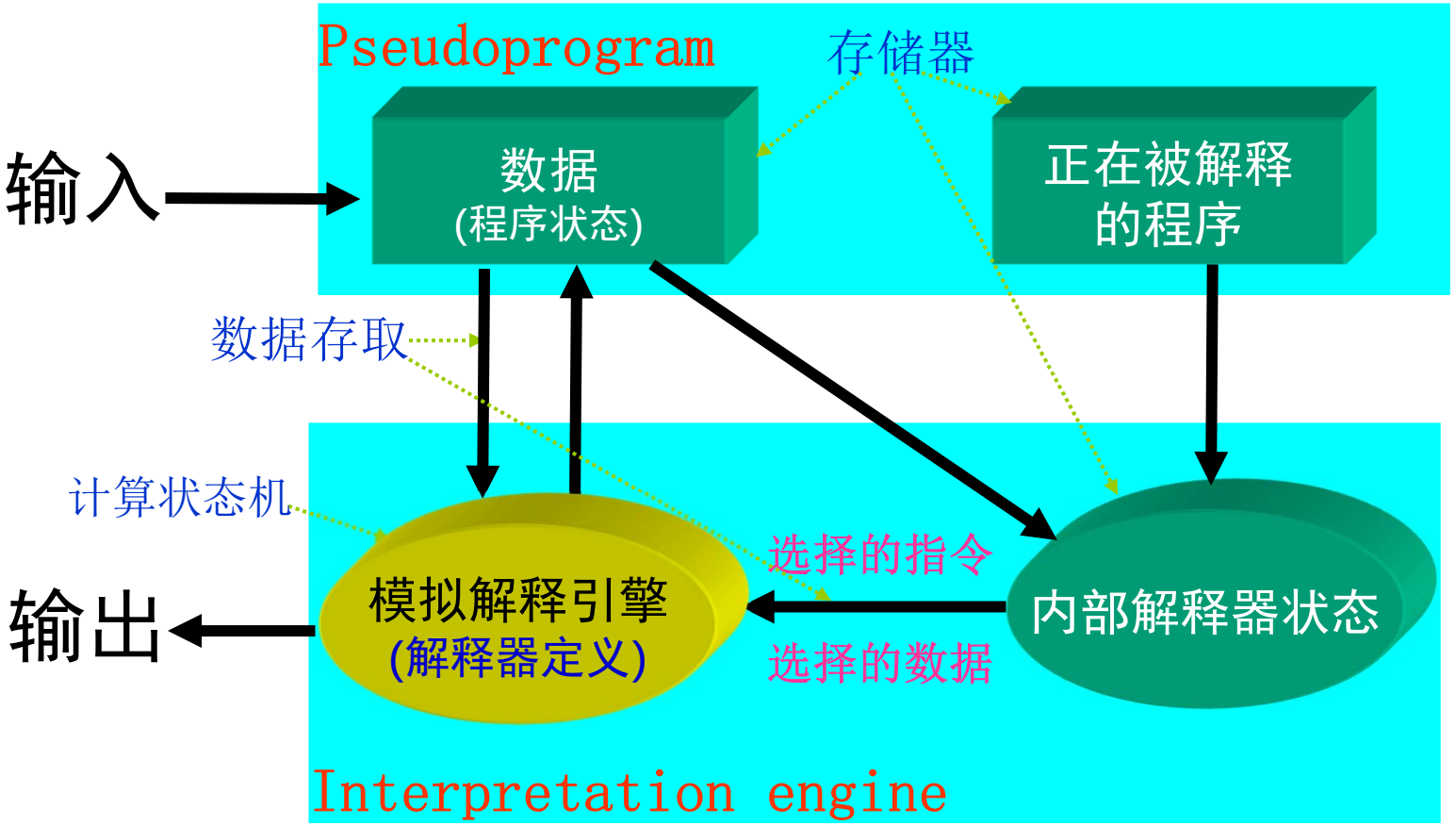
- 解释器通常用来在程序语言定义的计算和有效硬件操作确定的计算之间建立对应和联系。
  - 简单和小规模的解释器只完成基本的信息识别和转换
  - 复杂的解释器需要从词法到句法、到语法的复杂识别和处理
- 作为一种体系结构，解释器已经被广泛应用在从系统到应用软件的各个层面，
  - 包括各类语言环境、Internet浏览器、数据分析与转换等；
  - LISP、Prolog、JavaScript、VBScript、HTML、产生式系统、数据库系统(SQL解释器)、各种通信协议等。

解释器的组成

一个解释器就是一个虚拟机

- 就解释器来说，一个解释器包括：伪程序(pseudoprogram)和解释引擎(interpretation engine)两个部分
- 就虚拟机来说，包括：计算状态机CSM(Computation State Machine)和存储器(Memory)两个部分
- 也可以说，一个解释器包括：一个模拟解释引擎(CSM)和一个存储器。而一个存储器又包括：被解释的伪码、解释引擎控制状态的表示、以及被模拟程序的当前状态表示。

解释器的组成



Interpreters/Virtual Machine

解释器的组成

- 基本构件：
  - An interpretation engine to do the work (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 (解释器引擎当前的内部控制状态的表示：在某个时刻需要执行哪些指令)
    - 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)

Interpreter versus compiler
(解释器和编译器)

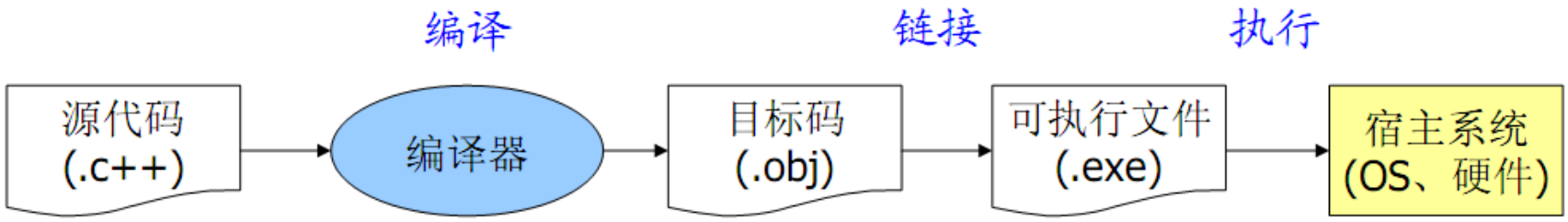
Interpreter versus compiler (解释器和编译器)

- 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。

- 二者在目标、功能与实现上有何差别？
- 二者的性能有何差别？

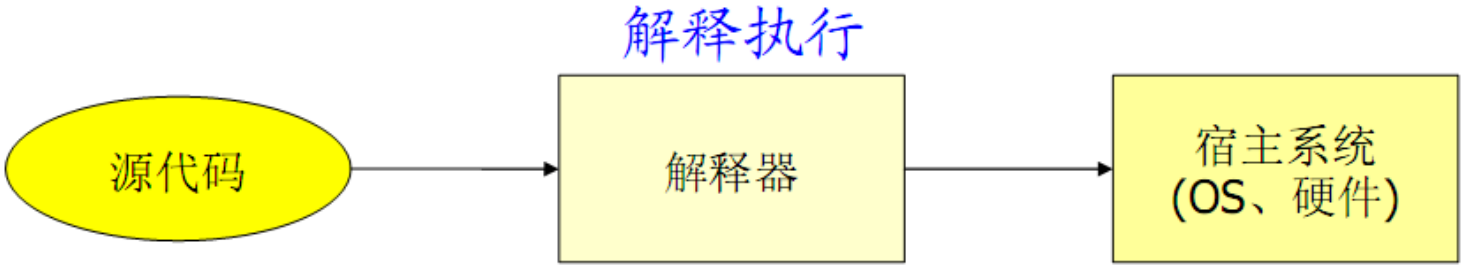
Compiler (编译器)

- 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)

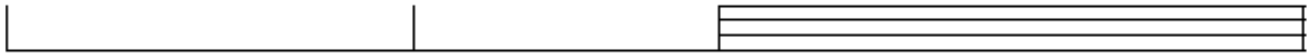
- It may be possible to execute the same source code either directly by an interpreter or by compiling it and then executing the machine code produced. (在解释器中，程序源代码被解释器直接加以执行。)



Interpreter versus compiler
(解释器和编译器)

编译器

编译　　链接　　执行

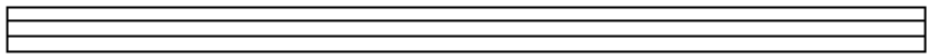


分析程序结构
源代码→目标码
确定内存分配与访问方案

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

Interpreter versus compiler (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，但是却低于编译器“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. (解释器通常省略了编译与链接的步骤，从而降低编程时间)
  - edit-interpret-debug (编辑源代码-解释-调试)
  - edit-compile-link-run-debug (编辑源代码-编译-链接-运行-调试)

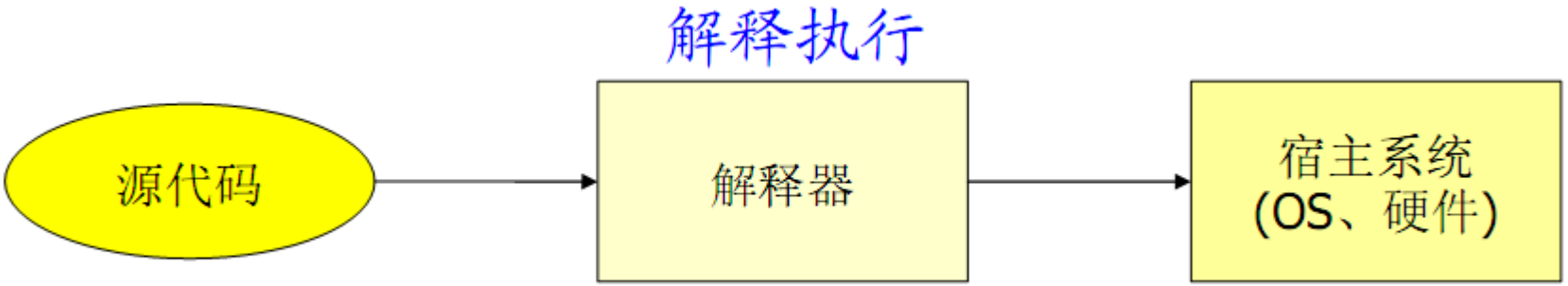
Interpreter versus compiler (解释器和编译器)

- 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 themapping 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)

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



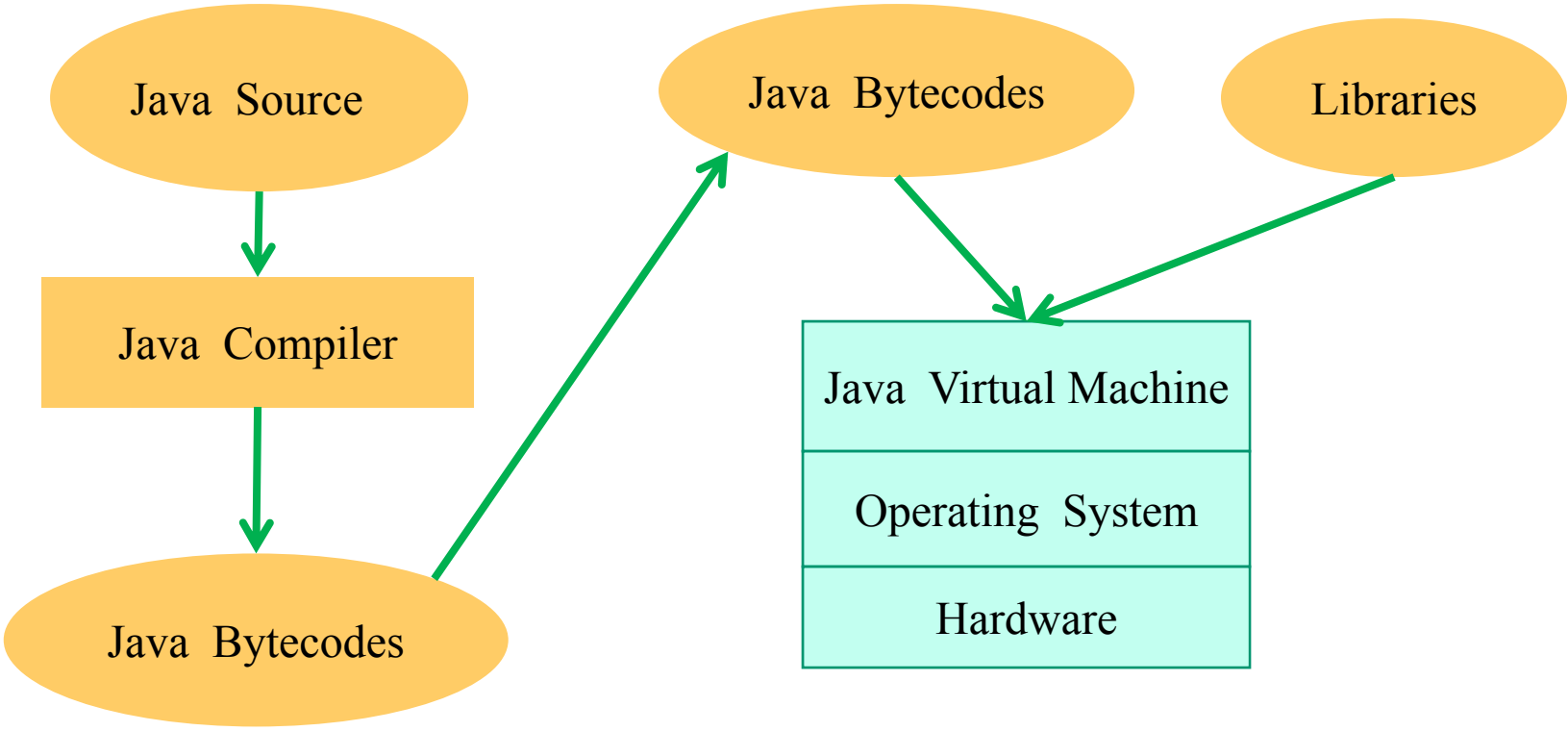
字节码解释器
Bytecode interpreter

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 Java compiles Java source code to Java bytecode. (Java 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++源程序要在不同的平台上运行，必须重新进行编译。

JVM中的解释器

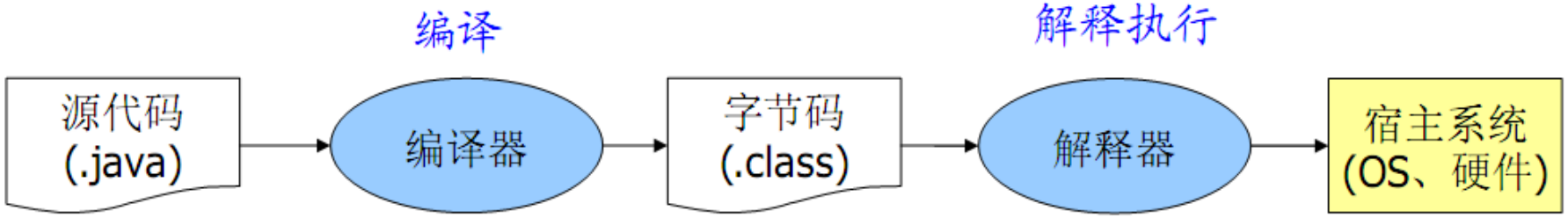

Java Source
Java Compiler
Java Bytecodes
Java Bytecodes
Libraries
Java Virtual Machine
Operating System
Hardware

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

注意：
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？



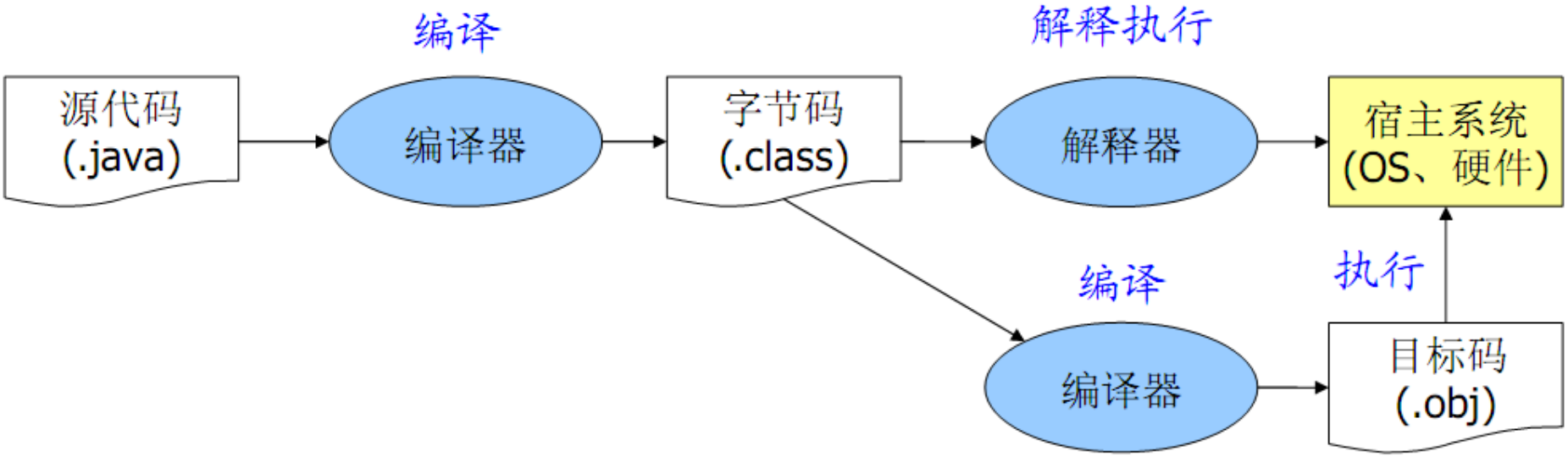
JIT (Just-in-time) Complier
实时编译器

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 (需编译哪些部分？)

- 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).

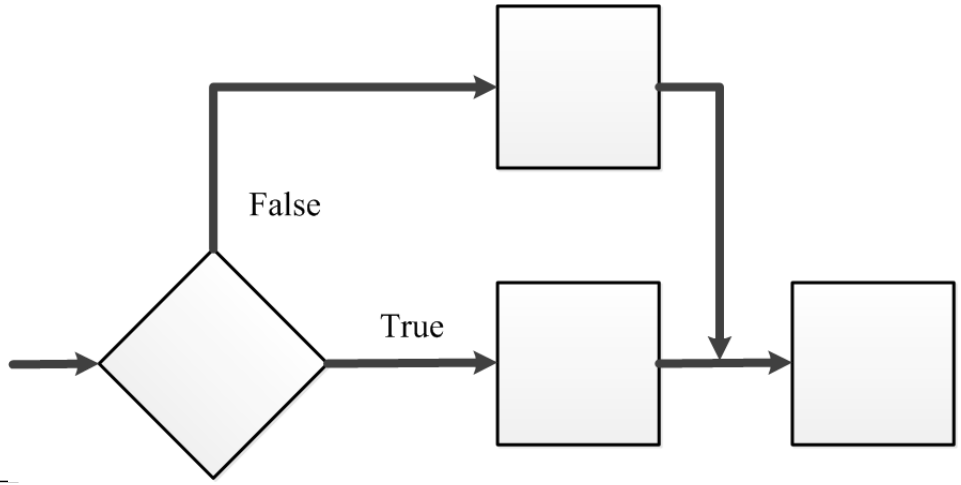
问题

- 针对解释器的三种策略：
  - 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
- 理解各自的工作原理，对比分析各自的优缺点。

3.6.3基于规则的系统
Rule-based System

if-else分支结构

- 解释器/编译器：对源程序/字节码中包含的每一条操作进行解释/编译并加以运行，所有的业务逻辑都被程序员书写在源代码中加以执行。
- 遇到不确定的业务逻辑，则采用if-else的分支结构进行判断，以控制程序的执行流向。如果条件为真，流向一边，如果不是，则流向另一边。



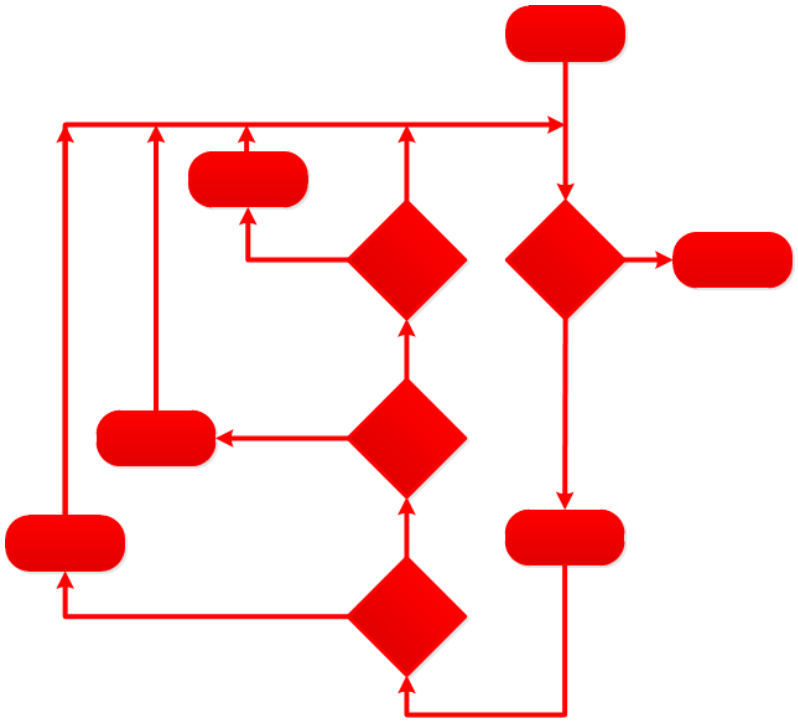
IF-ELSE-THEN

业务规则以if-else-then的形式散落在程序代码之中

当if-else结构越来越多时...

- 程序的结构变得越来越混乱，难以维护。

- 而且，多个分支结构之间并不存在简单的嵌套关系，因此，很多业务逻辑无法表达为if-else分支结构的形式。



业务逻辑与程序代码之间的“语义鸿沟”

- 现实中的业务需要通常采用自然语言来表达，而程序员则要将其“翻译”为程序代码，二者的语法之间存在非常大的区别，从而导致“语义鸿沟”。

- 这些业务需求一旦被编写为程序源代码，就不再能被容易理解了——丢失和埋没于代码之中。
  - 想想看，用汇编语言书写的程序，哪怕再简单的业务逻辑，对大多数人来说也是不可读的。

业务逻辑的频繁变化

- 现实里的业务需求经常频繁的发生变化，软件系统也要随之适应。如果每一次的需求变化都需要程序员来修改代码，那么，效率将会非常低，成本也非常高。

- 软件工程要求从“需求➡设计➡编码”，然而很多业务逻辑常常在需求阶段可能还没有明确，在设计和编码后还在变化。

如何解决上述问题？

- OO设计的一个重要原则：固定部分与可变部分分离；
  - OCP：the Open-Closed Principle（开放封闭原则）

- 将二者分离开来，当可变部分发生变化时，不会影响固定部分。

基于规则的系统

- 最好的办法是把频繁变化的、复杂的业务逻辑抽取出来，形成独立的规则库。这些规则可以独立于软件系统而存在，可被随时的更新；
  - 业务逻辑=固定业务逻辑+可变业务逻辑（规则）+规则引擎
  - 系统运行的时候，读取规则库，并根据模式匹配的原理，依据系统当前运行的状态，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。

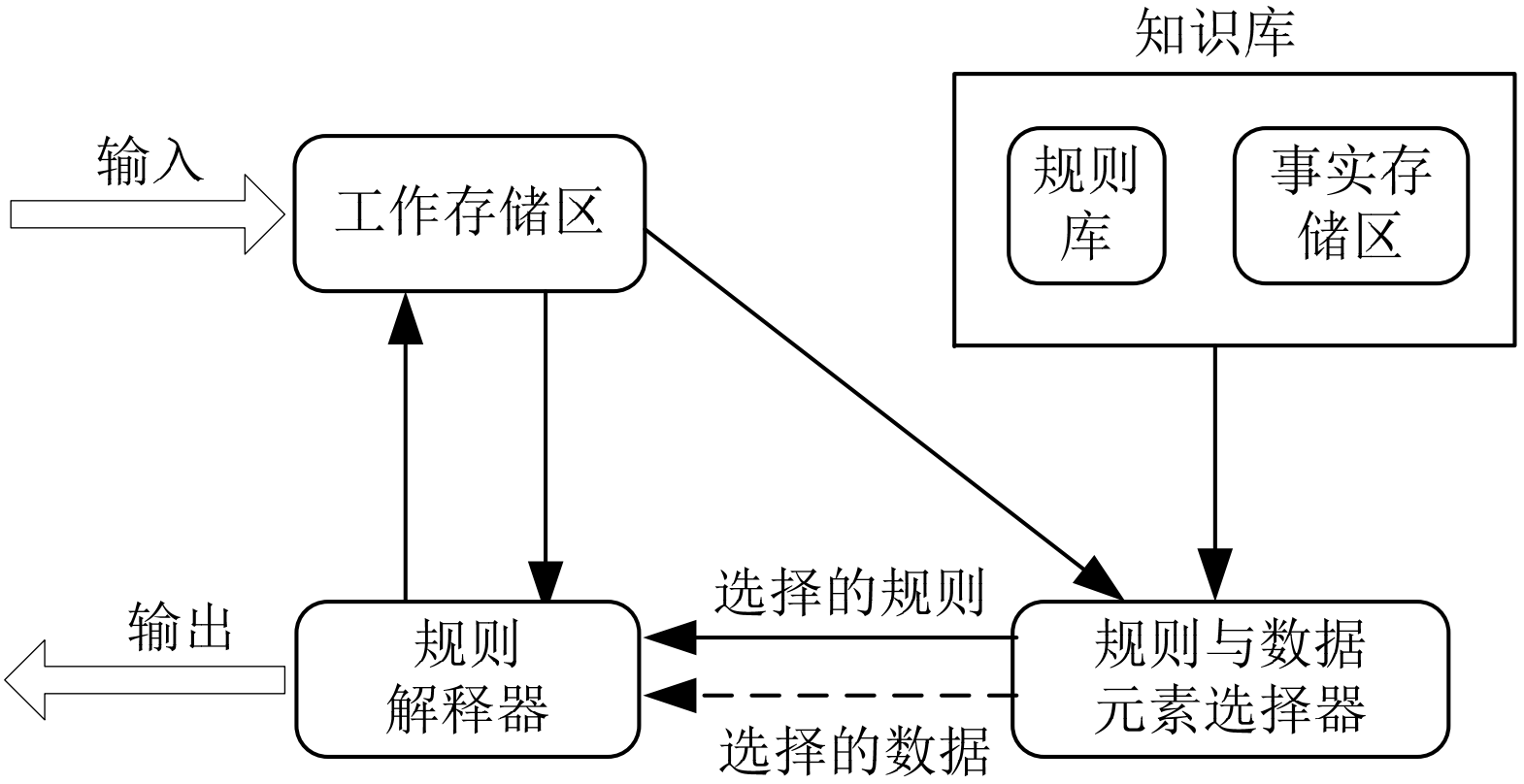
  ——“频繁变化的规则”与“较少发生变化的规则执行代码”分离

- 基于规则的系统：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。

基于规则的系统

- Rule-based systems provide a means of codifying the problem-solving know-how of human experts（基于规则的系统提供了一种将专家解决问题的知识与技术进行编码的手段）.
- These experts tend to capture problem-solving techniques as sets of situation-action rules whose execution or activation is sequenced in response to the conditions of the computation rather than by a predetermined scheme（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）.
- Since these rules are not directly executable by available computers, systems for interpreting such rules must be provided（一般的，规则都是用类似于自然语言的形式书写，无法被系统直接执行，故而需要提供解释规则执行的“解释器”）.

基于规则的系统



基于规则的系统

- 基本构件(与解释器风格中的构件类似):

基于规则的系统	解释器风格
The knowledge base 知识库(规则库)	the pseudo-code to be executed 待执行的伪代码
The rule interpreter 规则解释器	the interpretation engine 解释器引擎
The rule and data element selector 规则与数据元素选择器	The control state of the interpretation engine 解释器引擎的内部控制状态
the working memory 工作存储区	The current state of the program running on the virtual machine 程序当前的运行状态

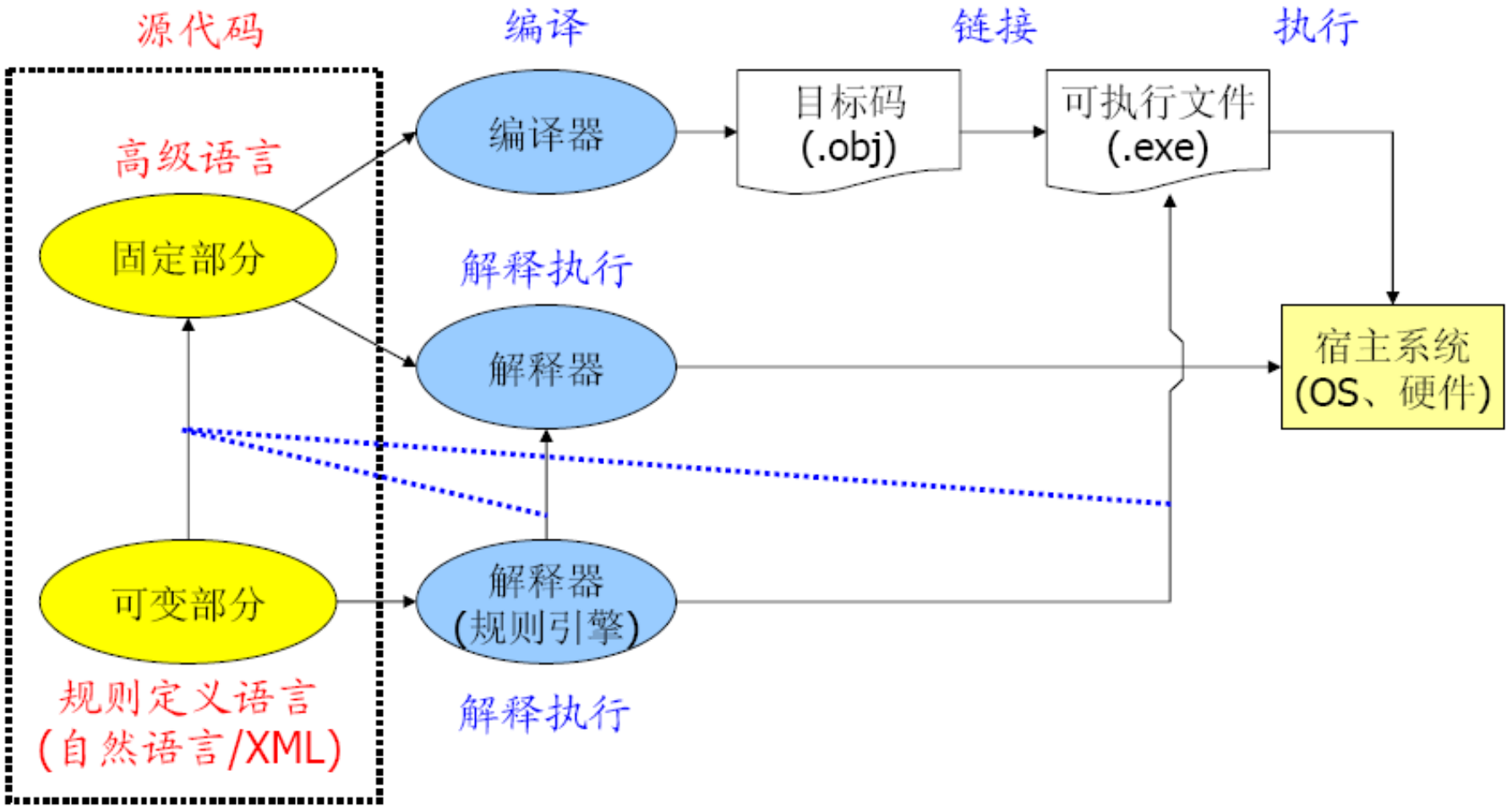
基于规则的系统

- 核心思想：
  - 将业务逻辑中可能频繁发生变化的代码从源代码中分离出来；
- 基本过程：
  - 使用规则定义语言（IF…THEN…的形式，通常基于XML或自然语言，但绝不是程序设计语言），将这些变化部分定义为“规则”；
  - 在程序运行的时候，规则引擎根据程序执行的当前状态，判断需要调用并执行那些规则，进而通过“解释器”的方式来解释执行这些规则。
  - 其他直接写在源代码里的程序，仍然通过传统的“编译”或“解释”的办法加以执行。

基于规则的系统

- 为什么将“基于规则的系统”归为虚拟机体系结构风格中？
  - 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。
- 二者的不同：
  - 解释器风格：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：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。

基于规则的系统


源代码
编译
链接
执行
高级语言
固定部分
编译器
目标码
(.obj)
可执行文件
(.exe)
解释执行
解释器
宿主系统
(OS、硬件)
可变部分
解释器
(规则引擎)
规则定义语言
(自然语言/XML)
解释执行

规则的表达形式

- Any rule consists of two parts(任何规则都包含两部分):
  - IF part, called the antecedent(premise or condition)(IF：规则的前提或条件)
  - THEN part, called the consequent(conclusion or action)(THEN：规则的结论或触发的行为)
    - IF<antecedent>THEN<consequent>
- A rule can have multiple antecedents joined by AND(),OR() or a combination of both.(一个规则可以有多个条件，使用AND或OR连接)
  - IF<antecedent 1>AND(<antecedent 2>OR<antecedent 3>) THEN<consequent>

几个例子

IF Customer.age<18
AND Withdraw.amount>1000
THEN signature of the parent is required

如果客户年龄小于18岁，
而且取款金额大于1000元，
那么，需要家长的签名；

IF CustomerOrder.TotalMoney>100
THEN CustomerOrder.Discount=10%

如果客户订单的总金额大于100元
，那么，该订单的折扣为10%；

IF Order.Sum>10,000
AND Order.Sum<=50,000
THEN
Order.AuditedBy(“DepartmentManager”
)

如果订单总金额大于1万元，
而且订单总金额小于5万元，
那么，该订单需要由部门经理审核

基于规则的系统的优点

- Lowers the cost incurred in the modification of business logic(降低了修改业务逻辑的成本)
- Shortens development time(缩短了开发时间)
- Rules are externalized and easily shared among multiple applications(将规则外部化，可在多个应用之间共享)
- Changes can be made faster and with less risk(对规则的改变将非常迅速并且具有较低的风险)

几个例子

IF ShoppingCart.purchaseAmount>200
THEN Customer.category="Gold"

如果，购物车里有货物总金额大于200元，
那么，该顾客为金卡顾客

IF Customer.category=="Gold"
THEN ShoppingCart.discount+=5%

如果，顾客为金卡顾客，
那么，该客户每次购物的折扣在原有基础上提高5%；

IF ShoppingCart.discount>15%
THEN ShoppingCart.discount=15%

如果，某次购物的折扣超过15%，
那么，调整折扣到15%。

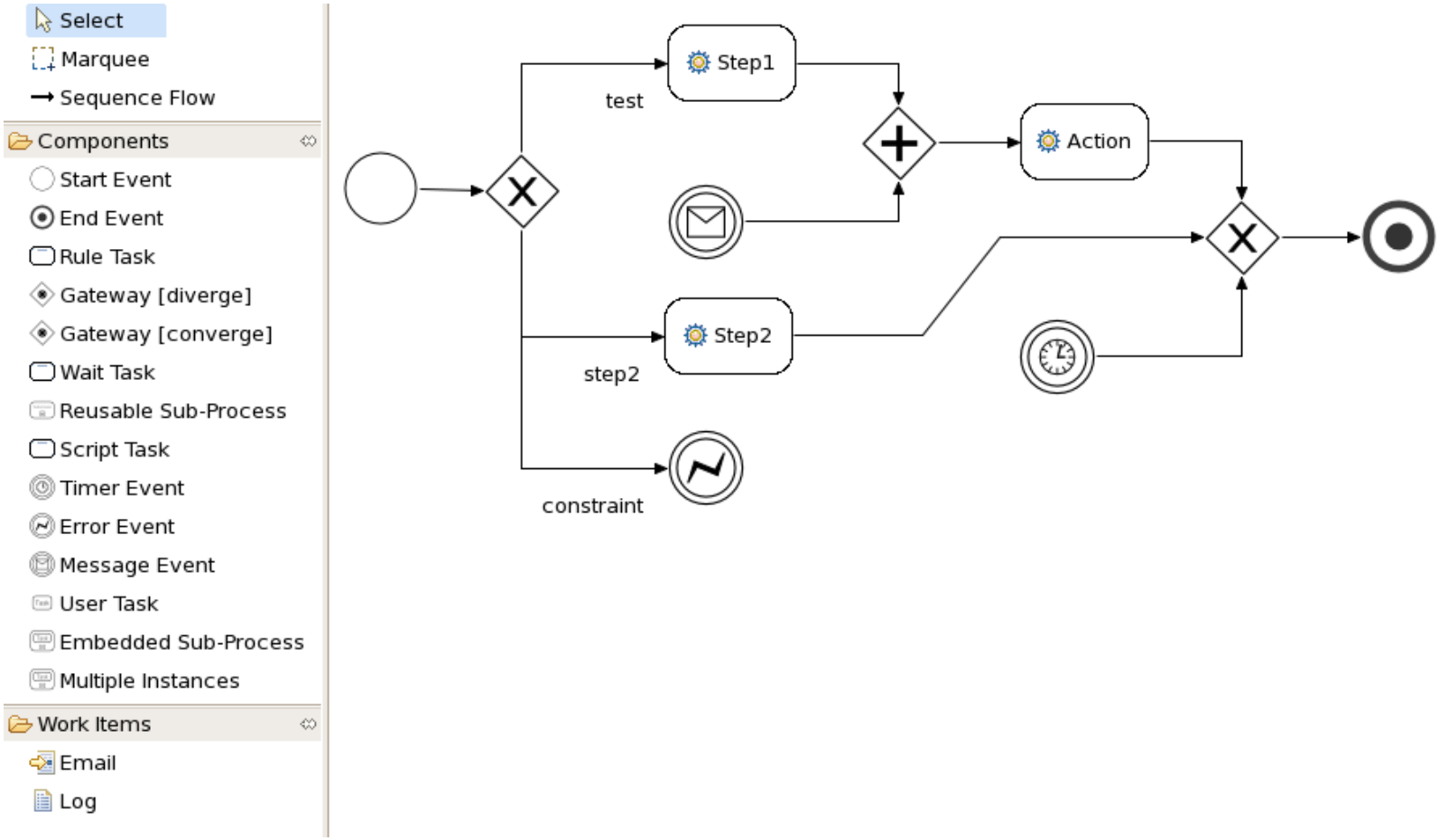
规则的分类

- UI输入域的合法性检查
- 数据库中的触发器和存储过程
- 安全性规则（权限控制规则）
- 组织中的权利/义务/责任的相关规则
- 业务过程中各活动的执行次序
- 业务策略（如折扣策略、风险/信用控制策略等）
- …

JBoss 规则引擎（Doors）

- JBoss Rules(Drools) 是一个基于Charles Forgy's的RETE算法的，易于访问企业策略、易于调整以及易于管理的开源业务规则引擎，符合业内标准，速度快、效率高。
- 与Drools功能类似的同类开源产品主要有：OpenRules、OpenLexicon等。

Drools


Select
Marquee
Sequence Flow
Components
Start Event
End Event
Rule Task
Gateway [diverge]
Gateway [converge]
Wait Task
Reusable Sub-Process
Script Task
Timer Event
Error Event
Message Event
User Task
Embedded Sub-Process
Multiple Instances
Work Items
Email
Log
Step1
test
Action
Step2
step2
constraint

Jess规则引擎简介

Jess(Java Expert System Shell)是java平台的规则引擎，由美国桑迪亚国家实验室使用java开发的rule-based专家系统程序语言，扩充了CLIPS(C Language Integrated Production System)的功能，是一个强大的专家系统发展语言。

Jess提供适合自动化专家系统的逻辑编程，采用声明式编程，通过pattern match的过程连续对一个事实的集合运用一系列的规则。

Jess规则引擎使用Rete算法来处理规则。可以使用forward-chaining（正向链接推理）和backward-chaining（反向链接推理）这两种推论方式。

Jess可以被用来构建使用规则定义形式的知识来推导结论和推论的Java Servlet、EJB、Applet和应用程序。因为不同的规则匹配不同的输入，所以有了一些有效的通用匹配算法。

规则引擎的比较（语言方面）

Drools vs Jess

相同点：均使用if-then句式与foreward-chaining的Rete引擎，按优先级匹配条件语句，执行规则语句。

不同点：针对开发人员

Drools：用XML的<Conditons>、<Consequence> 节点表达If--Then句式，而里面可以嵌入Java/Groovy/Python语言的代码作为判断语句和执行语句。用XML节点来规范If--Then句式和事实的定义，使引擎干起活来很舒服。 而使用Java,Groovy等原生语言来做判断和执行语句，让程序员很容易过渡、移植，学习曲线很低。

Jess：用 => 表达 If-Then句式。 CLIPS是真正的程序员专用语言，而且是很专业的程序员才习惯的东西。但这种本来就是用来做专家系统的AI语言，对规则的表达能力也应该是最强的。

规则引擎的比较（语言方面）

drools

```
<java:condition>
    hello.equals("Hello")
 </java:condition>

 <java:consequence>
    helloWorld( hello );
 </java:consequence>
```

jess

```
(defrule welcome-toolders
   "Give a special greeting to young chiledren!"
   (person {age < 3})
   =>
   (printout t "hello,little one!" crlf) )
```

作业

- 1.针对解释器的三种策略：
  - 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

理解各自的工作原理，对比分析各自的优缺点。

作 业

- 2.基于规则引擎Drools开发一个简单的专家系统：

（1）根据百分制成绩进行评级(A-F)，转化规则如下：

A (90-100)

B+ (87-89)

B (80-86)

C+ (77-79)

C (70-76)

D+ (67-69)

D (60-66)

F (0-59)

（2）分析基于规则的系统适用于什么情况，其优缺点是什么？