

计算机组成和体系结构

第十二讲

四川大学网络空间安全学院

2020年5月18日

封面来自pcquest.com

Java

Dart
Scala
Max/MSP
cT
SAS
Haskell
Groovy
Tcl
Lisp
Mathematica

Erlang c# D Scala F# ML OpenEdge ABAP Visual Basic Transact-SQL Clarion Python Q Scheme R Haskell Go PHP Pascal Swift Objective-C FoxPro Perl Logo Prolog PostScript C++ Fortran MATLAB C Ada ActionScript Forth PL/SQL Delphi TCL PL/I ColdFusion Forth PL/SQL C Lua Io Yacc

版权声明

课件中所使用的图片、视频等资源版权归原作者所有。

课件原创内容采用 [创作共用署名—非商业使用—相同方式共享4.0国际版许可证\(Creative Commons BY-NC-SA 4.0 International License\)](#) 授权使用。

Copyright@四川大学网络空间安全学院计算机组成与体系结构课程组，2020



上期内容回顾

- 操作系统
 - 操作系统的发展、分类和标志技术
- 保护环境
 - 虚拟机、子系统和逻辑分区
 - XEN、KVM、AIX和Docker
 - Intel VT-x和VT-d

本期学习目标

- 编程工具
 - 汇编器、加载和链接器
 - 编程语言
 - 编译过程
 - JVM
- 数据库工具
 - 数据库视图
 - 数据库事务

中英文缩写对照表

英文缩写	英文全称	中文全称
DLL	Dynamic Link Library	动态链接库
GC	Garbage Collection	垃圾回收（空余内存回收）
JIT	Just-in-time	即时编译
JVM	Java Virtual Machine	Java虚拟机
ACID	Atomicity, consistency, isolation, durability	原子性、一致性、隔离性和持久性

编程工具

- 编程工具包含了从创建程序到最终执行的各个步骤
 - 编译时 (Compile time) : 从源代码编译为可执行二进制文件的过程
 - 加载时 (Load time) : 将执行二进制文件载入内存, 准备执行的过程
 - 运行时 (Run time) : 可执行二进制文件实际执行的过程
- 一个最简单的流程:
 - 使用汇编器将汇编语言编译为可执行二进制指令
 - 操作系统将可执行二进制指令加载到内存中并执行

汇编器 (ASSEMBLER)

- 汇编器是最基本的编程工具
- 汇编器将助记指令翻译为机器代码
 - 通常采用两次通读的方法
 - 第一次进行部分的翻译，并构造符号表
 - 第二次通读根据符号表完成指令翻译

可重定位代码和绝对代码

- 汇编器通常输出**可重定位** (Relocatable) 的二进制指令
 - 程序中数据和代码采用相对位置表示，实际地址是可以由操作系统在加载的时候决定的。例：一般的用户程序
 - 与之相对的，控制某些设备和操作系统的指令通常是不可重定位的，称为绝对代码（Absolute Code）。例：中断处理程序
- 重定位的实现原理
 - 程序中所使用地址表示偏移值
 - 在加载的时候，在特殊的寄存器中保存加载的基地址

重定位示例

Load x	1+004
Add y	3+005
Store z	2+006
Halt	7000
x, DEC 35	0023
y, DEC -23	FFE9
z, HEX 0000	0000

Address	Memory Contents
0x250	1254
0x251	3255
0x252	2256
0x253	7000
0x254	0023
0x255	FFE9
0x256	0000

Loaded Starting at Address 0x250

Address	Memory Contents
0x400	1404
0x401	3405
0x402	2406
0x403	7000
0x404	0023
0x405	FFE9
0x406	0000

Loaded Starting at Address 0x400

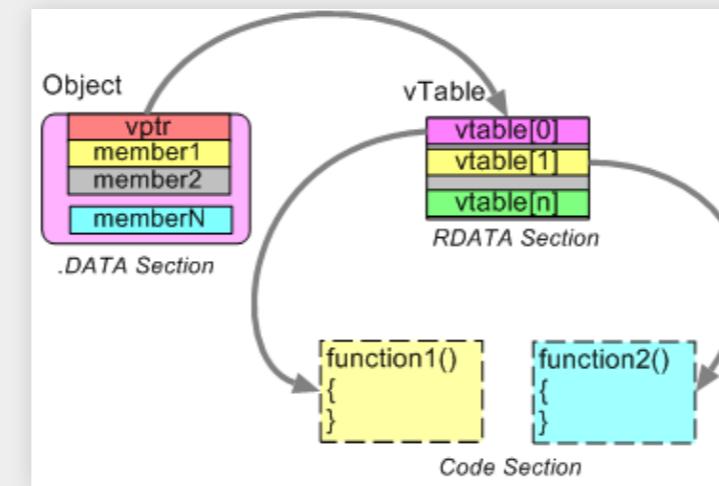
注意：MARIE体系结构不支持基址寻址，因此加载的时候会根据加载位置计算实际地址。在其它一些体系结构中（例如x86），通常是直接设置各段的基地址。

地址绑定

- 确定程序中的变量对应的内存地址的过程也叫做“绑定” (Binding)
- 绑定过程可以发生在编译、加载和运行时
- 编译时绑定生成的是绝对代码
- 加载时绑定在程序载入内存的时候为变量指定内存位置（加载之后的地址就不会改变了）
- 运行时绑定也叫做动态绑定，通常采用基址寻址加间接寻址的方式实现，程序中记录的是一个偏移值，在程序执行过程中根据实际基址寄存器中的值读取对应的地址中保存的实际地址（例：C++中的虚函数）

动态绑定示例：虚函数

- 根据对象得到指向虚表的指针
- 在虚表中查找对应的函数地址
- 调用最终的函数地址



图片来源：<http://www.equestionanswers.com/cpp/vptr-and-vtable.php>

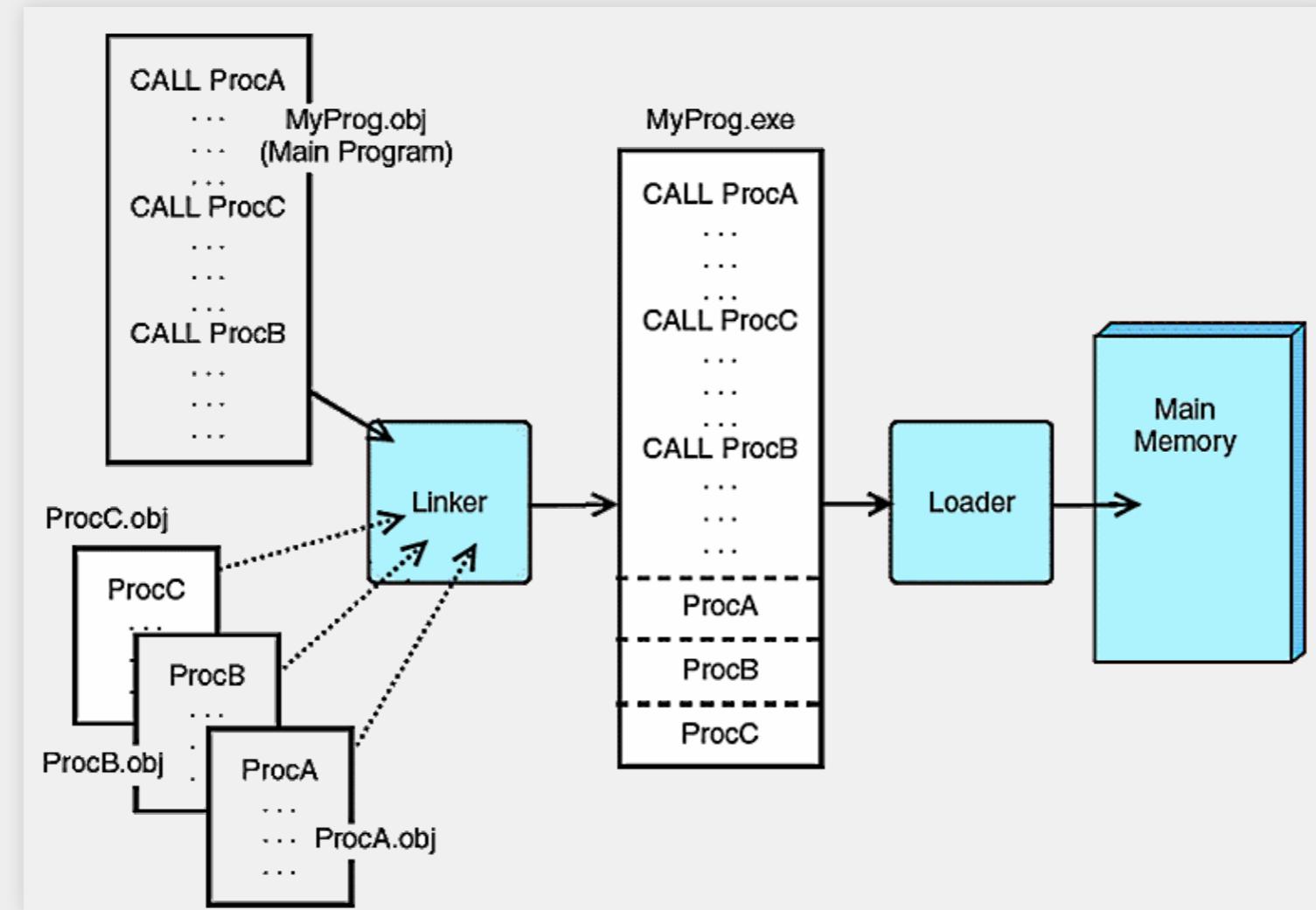
链接器(LINKER)

- 实际中一个可执行文件通常是由多个可执行文件组合得到的，并可能包含对外部符号的引用
- 需要通过链接器生成最终的可执行文件
 - 示例：ld - the GNU linker
- 链接的过程同样可以发生在编译时、加载时和运行时

静态链接

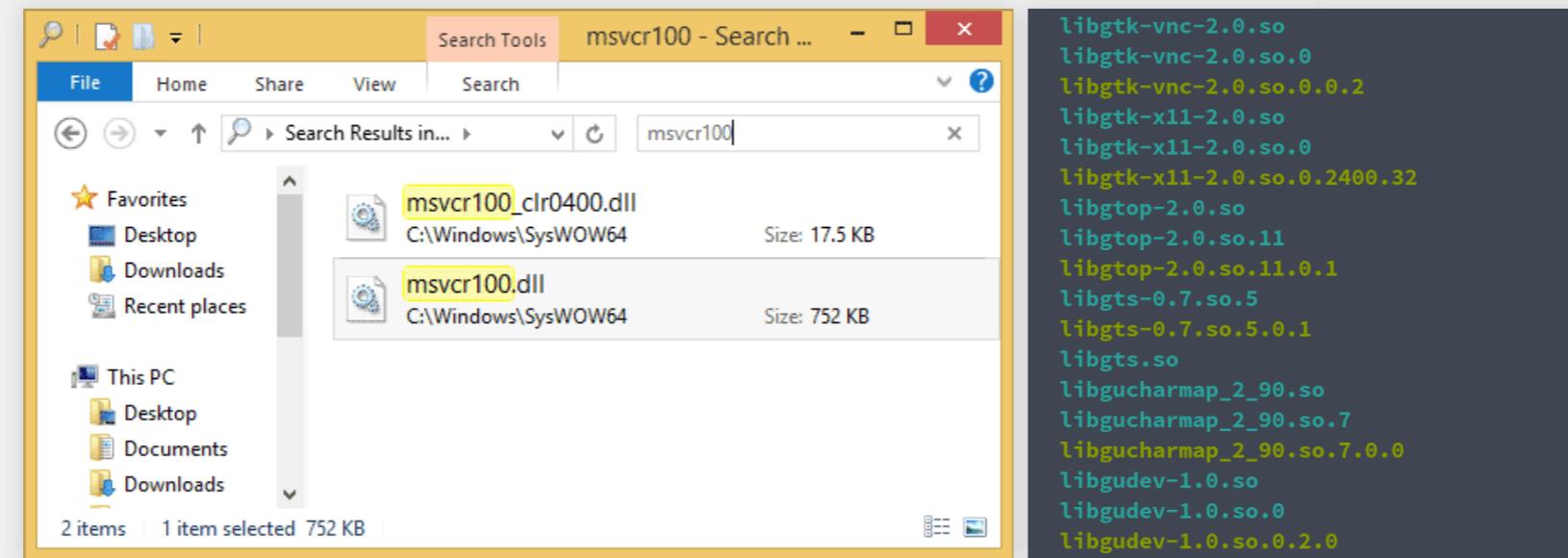
- 编译时链接也称为静态链接（Static Linking）：链接器将多个二进制指令文件进行合并，并且处理其中的符号引用
- 链接器通常也需要进行两次通读
 - 第一次建立全局符号表，记录每个符号在合并的文件中实际存储的位置
 - 第二次将各文件中引用的符号替换为实际的位置

静态链接示例



动态链接

- 外部引用在加载时或者运行时被绑定到对应的地址称为动态链接（Dynamic Linking），对应的库文件在Windows中称为动态链接库（Dynamic Link Library, DLL），在*nix系统中称为共享对象（Shared Object）

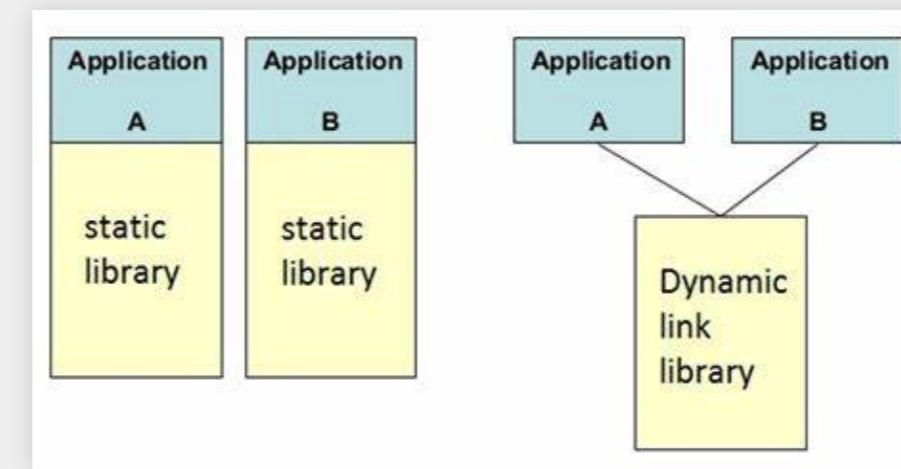


动态链接类型

- 加载时动态链接：在程序被加载的时候，操作系统在特定的路径中寻找选择对应的动态链接库中的符号地址，如果依赖的动态链接库未加载则需要先加载动态链接库
- 运行时动态链接：也叫做动态加载（dynamic loading），通常用于支持Plugin机制，调用动态链接库的函数（例如 `dlopen`）从动态链接库文件中查找对应的地址进行绑定
- 加载时动态链接会增加程序加载的时间（因为需要递归加载依赖的动态链接库），但是调用DLL函数的速度更快
- 运行时动态链接需要在调用函数之前才进行动态链接库的加载和符号查找，因此执行速度较慢

动态链接示例

静态链接和（加载时） 动态链接在生成最终二进制代码时的区别



图片来源：<http://codetutorial.blog.ir/1396/05/14/Dynamic-link-library-PE-Portable-Executable>

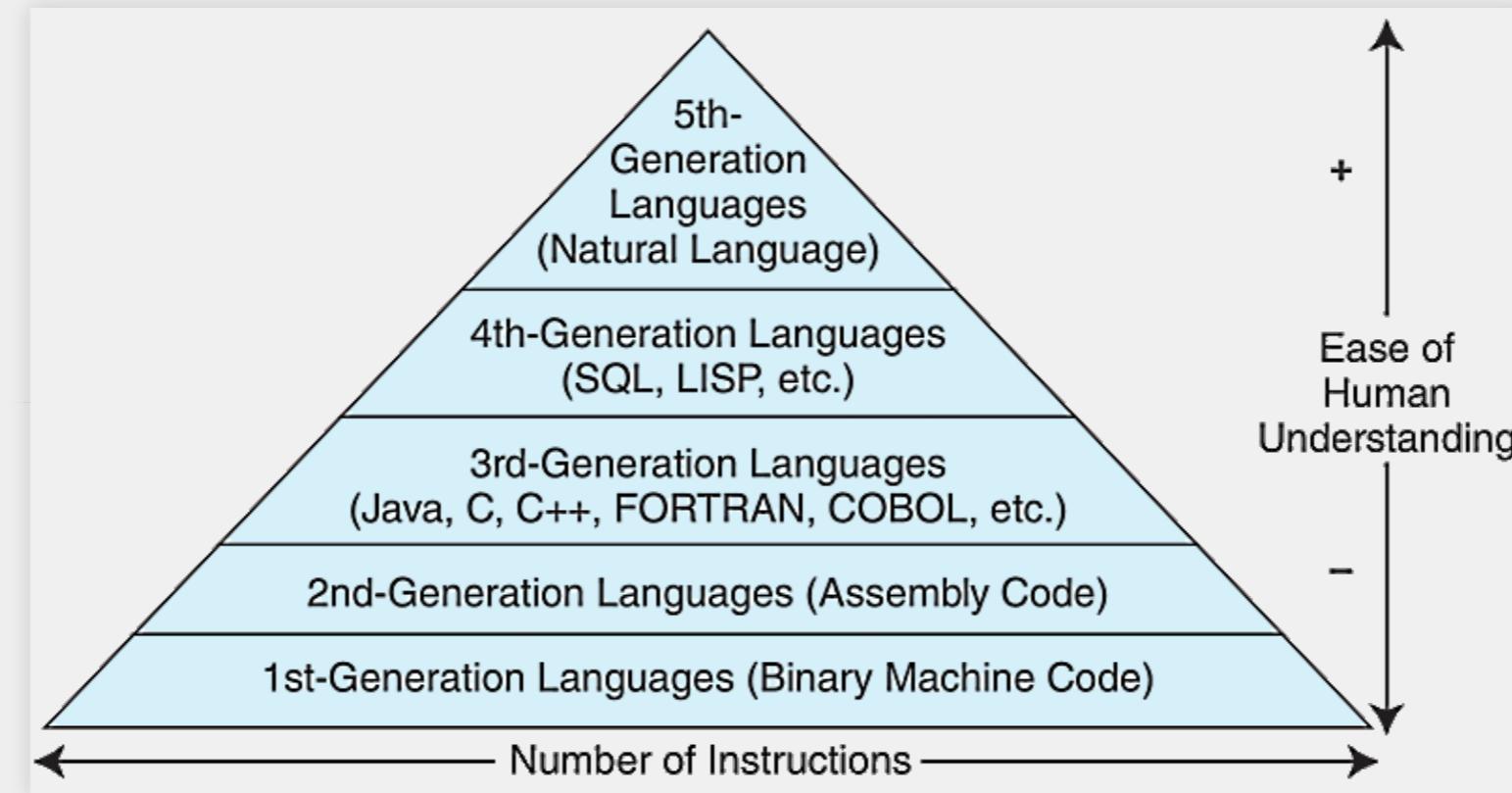
链接方式对比

项目	静态链接	加载时动态链接	运行时动态链接
二进制文件大小	大	小	小
加载速度	快	慢（冷启动）	快
调用速度	快	快	慢（冷启动）
依赖更新	重新链接	重新执行	程序内更新函数指针

编程语言的分类

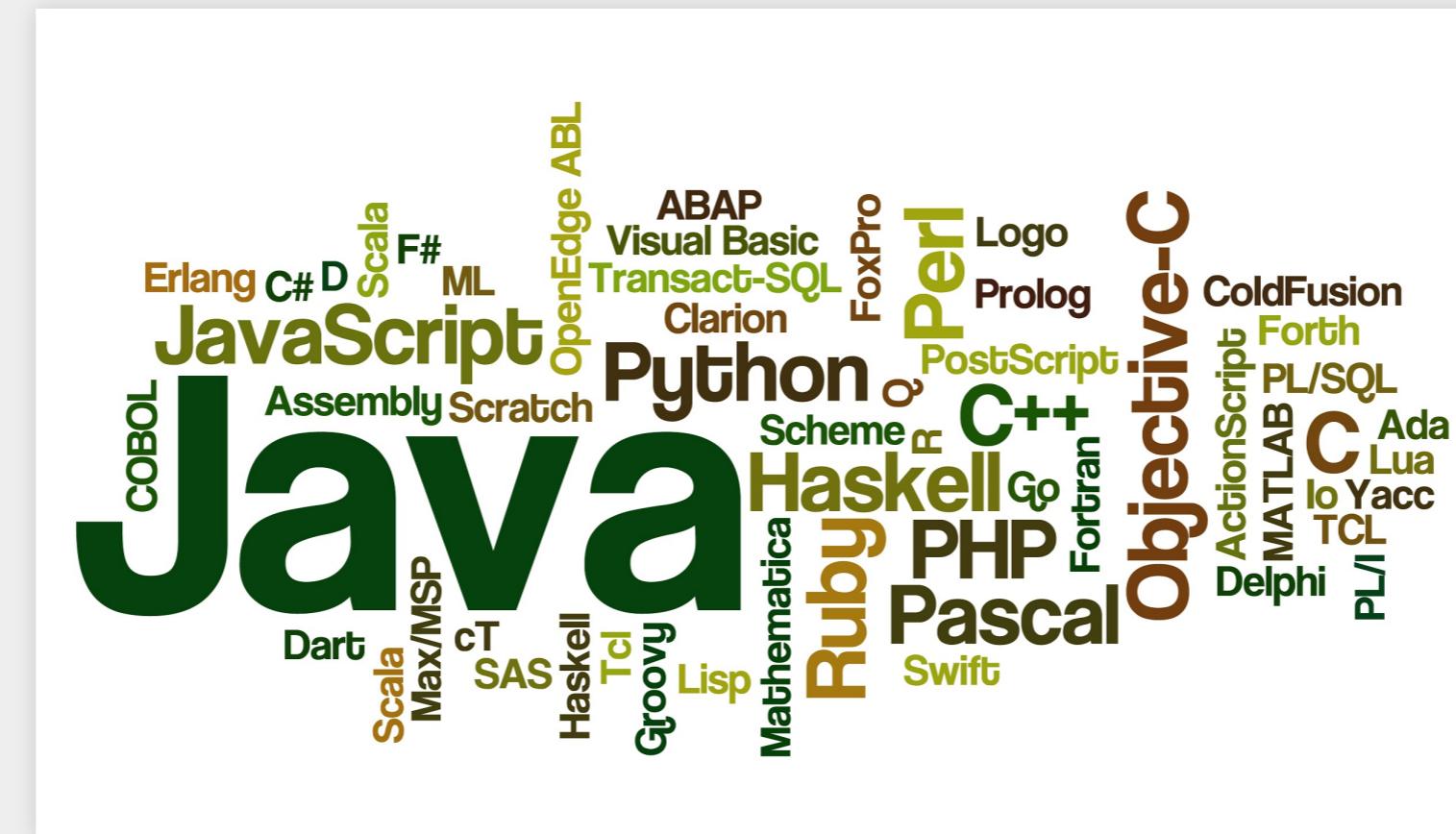
- 实际工作中，大部份程序员不会直接编写汇编语言
- 按照编程语言的抽象程度：
 - 抽象程度越低，编程语言越接近机器语言
 - 抽象程度越高，编程语言越接近人类自然语言
- 汇编语言通常被认为是第二代编程语言
- C/C++、Fortran、Java等被认为是第三代

编程语言图谱



编译语言分类

- 按照特点可以对编程语言进行分类
 - 面向对象编程语言，例如：C++，Java，Objective-C等等
 - 函数式编程语言，例如：Haskell，F#，LISP，Erlang等



编译器(COMPILER)

- 计算机硬件只能理解机器语言 (L1)
- 编译器的作用就是将高级语言进行翻译，最终转化为机器语言
- 编译过程通常包含六个阶段

编译过程 (1)

- 词法分析 (Lexical Analysis) : 从程序文本中提取出有意义的语言元素 (或者叫做单词) , 例如关键字、数字、字符串、空行等
- 语法分析/解析 (Syntax Analysis/Parsing) : 根据语法规则, 将单词组合为对应的语法树结构
- 有的工具中这两步可以同时完成

```
# Start Tokens
1 1 FN
1 4 ID csid
1 8 LPAREN
301
}

fn csid() -> i64 {
    1 14 I64
    1 18 LBRACE
    2 3 NUM 301
    3 1 RBRACE
    1 11 ARROW
# End Tokens

BINOP: ">"
| ">="
| "<"
| "<="
| "="
| "!="

select_statement: [opt_clause] select_clause [where_clause] [as_clause]

select_clause: SELECT ra_expr IN VARNAM
as_clause: AS VARNAM
ra_expr: (WAYPOINT RA_OP)+ WAYPOINT
```

图片来源: <https://cs.wellesley.edu/~cs301/s19/project/lexer/>

编译过程 (2)

- 语义分析 (Semantics Analysis) : 检查数据类型以及运算符有效性等
- 生成中间代码(Intermediate Representation): 常见的中间代码包括三地址码 (three address code) 、 LLVM IR 等，中间代码通常是独立于体系结构的，可以进一步转换与优化

```
; Declare the string constant as a global constant...
%.LC0 = internal constant [13 x sbyte] c"hello world\0A\00"           ; [13 x sbyte]*
; External declaration of the puts function
declare int %puts(sbyte*)                                              ; int(sbyte)*

; Definition of main function
int %main() {                                                               ; int()
    ; Convert [13x sbyte]* to sbyte ...
    %cast210 = getelementptr [13 x sbyte]* %.LC0, long 0, long 0 ; sbyte*
    ; Call puts function to write out the string to stdout...
    call int %puts(sbyte* %cast210)                                     ; int
    ret int 0
}
```

```
def select_statement(self, ast):
    children = ast.children.copy()
    if children[0].data == 'opt_clause':
        opt_obj = children[0].opt_obj
        children = children[1:]
    else:
        opt_obj = None

    reactive = children[0].reactive
    ra_expr = children[0].ra_expr
    toponame = children[0].toponame
    children = children[1:]

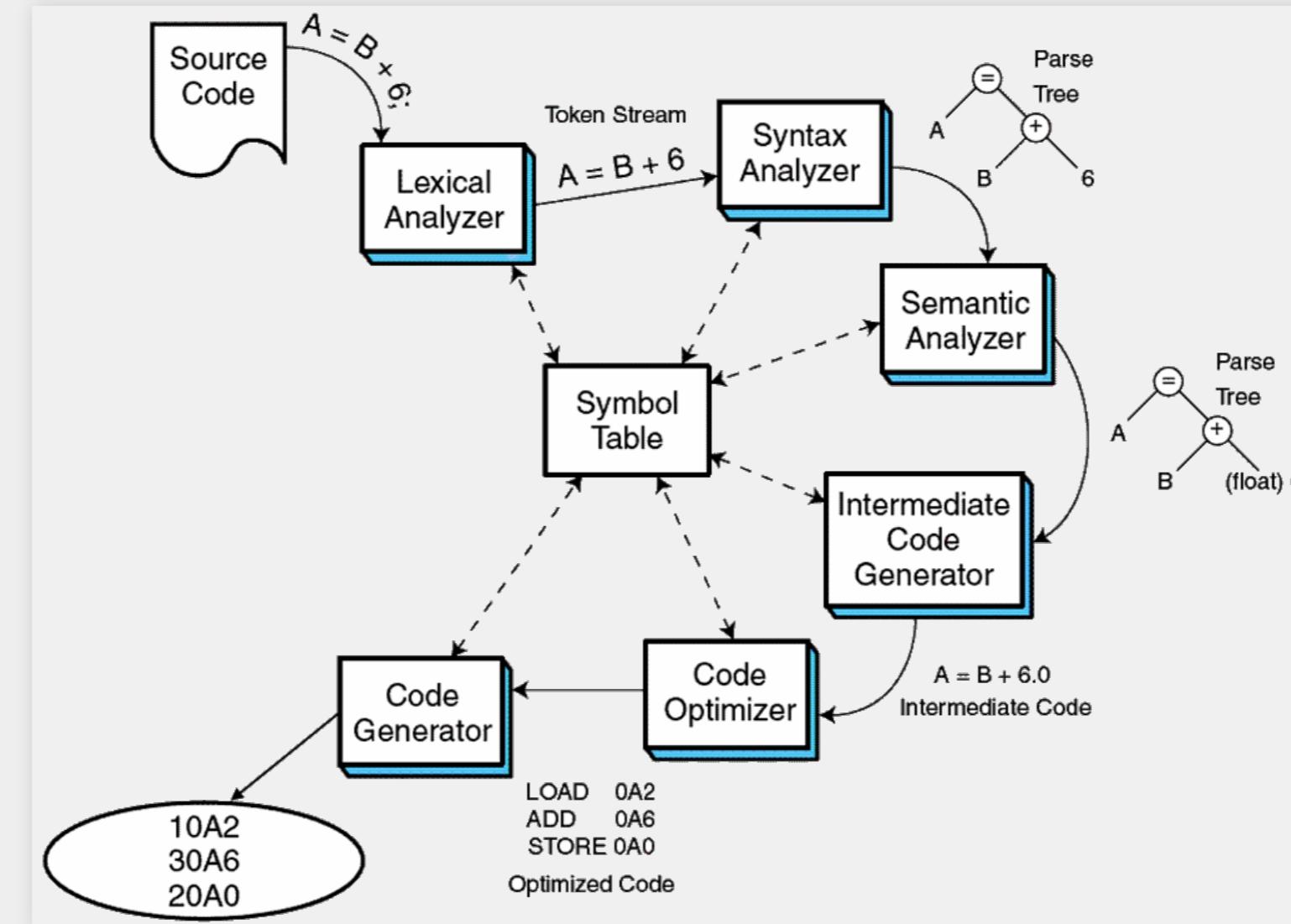
    if len(children) > 0 and children[0].data == 'where_clause':
        constraints = children[0].constraints
        children = children[1:]
    else:
        constraints = None

    if len(children) > 0 and children[0].data == 'as_clause':
        varname = children[0].varname
        children = children[1:]
    else:
        varname = None
    ast.cmd = SelectCommand(ra_expr, toponame, varname,
                           reactive, constraints, opt_obj)
```

编译过程 (3)

- 代码优化(Optimization)：利用流分析等优化手段对中间代码进行重新排列、替换、变换等，在不改变程序执行逻辑的前提下提高程序的性能
- 代码生成（Code Generation）：生成目标平台上的可执行文件，或者是下一层编程语言的源文件

编译过程示例



解释器(INTERPRETER)

- 解释器：支持对编程语言的实时执行
- 解释型语言一般比编译的语言执行速度更慢
- 解释型语言的最终结果可能不是机器语言，而是解释器可以识别的某种格式

```
def select_statement(self, ast):
    children = ast.children.copy()
    if children[0].data == 'opt_clause':
        opt_obj = children[0].opt_obj
        children = children[1:]
    else:
        opt_obj = None

    reactive = children[0].reactive
    ra_expr = children[0].ra_expr
    toponame = children[0].toponame
    children = children[1:]

    if len(children) > 0 and children[0].data == 'where_clause':
        constraints = children[0].constraints
        children = children[1:]
    else:
        constraints = None

    if len(children) > 0 and children[0].data == 'as_clause':
        varname = children[0].varname
        children = children[1:]
    else:
        varname = None
    ast.cmd = SelectCommand(ra_expr, toponame, varname,
                           reactive, constraints, opt_obj)
```

```
def dispatch(self, cmd):
    if isinstance(cmd, LoadCommand):
        self.load(cmd)
    elif isinstance(cmd, DropCommand):
        self.drop(cmd)
    elif isinstance(cmd, DefineCommand):
        self.define(cmd)
    elif isinstance(cmd, SetCommand):
        self.set_value(cmd)
    elif isinstance(cmd, SelectCommand):
        self.select(cmd)
    elif isinstance(cmd, ShowCommand):
        self.show(cmd)
```

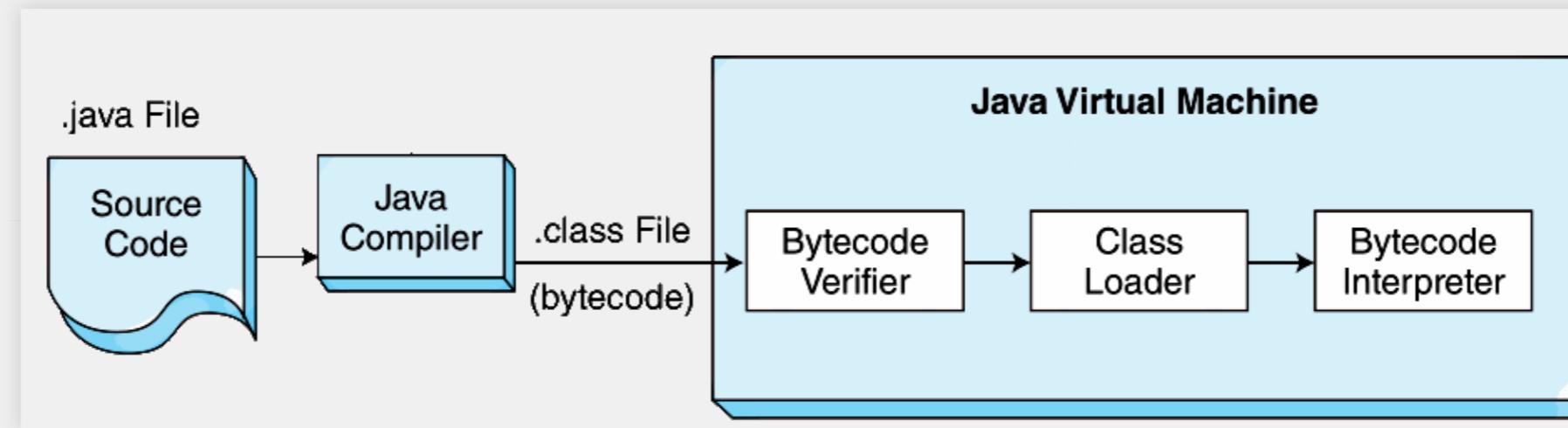
JAVA

- Java为我们前面提到的许多概念提供了一个示例
- Java的最终编译目标是Java虚拟机（Java Virtual Machine）
- Java虚拟机采用了一套特殊的指令集，称为Java字节码（Java Bytecode）
- Java本身只支持编译模式
- JVM是一个解释器，因此可以基于JVM开发解释型的语言（例如Scala）

JVM

- JVM可以看作一个微型操作系统
- JVM可以完成字节码的链接、加载，以及进程管理、资源管理等功能
- JVM的执行效率通常没有原生应用快

JAVA编译执行过程

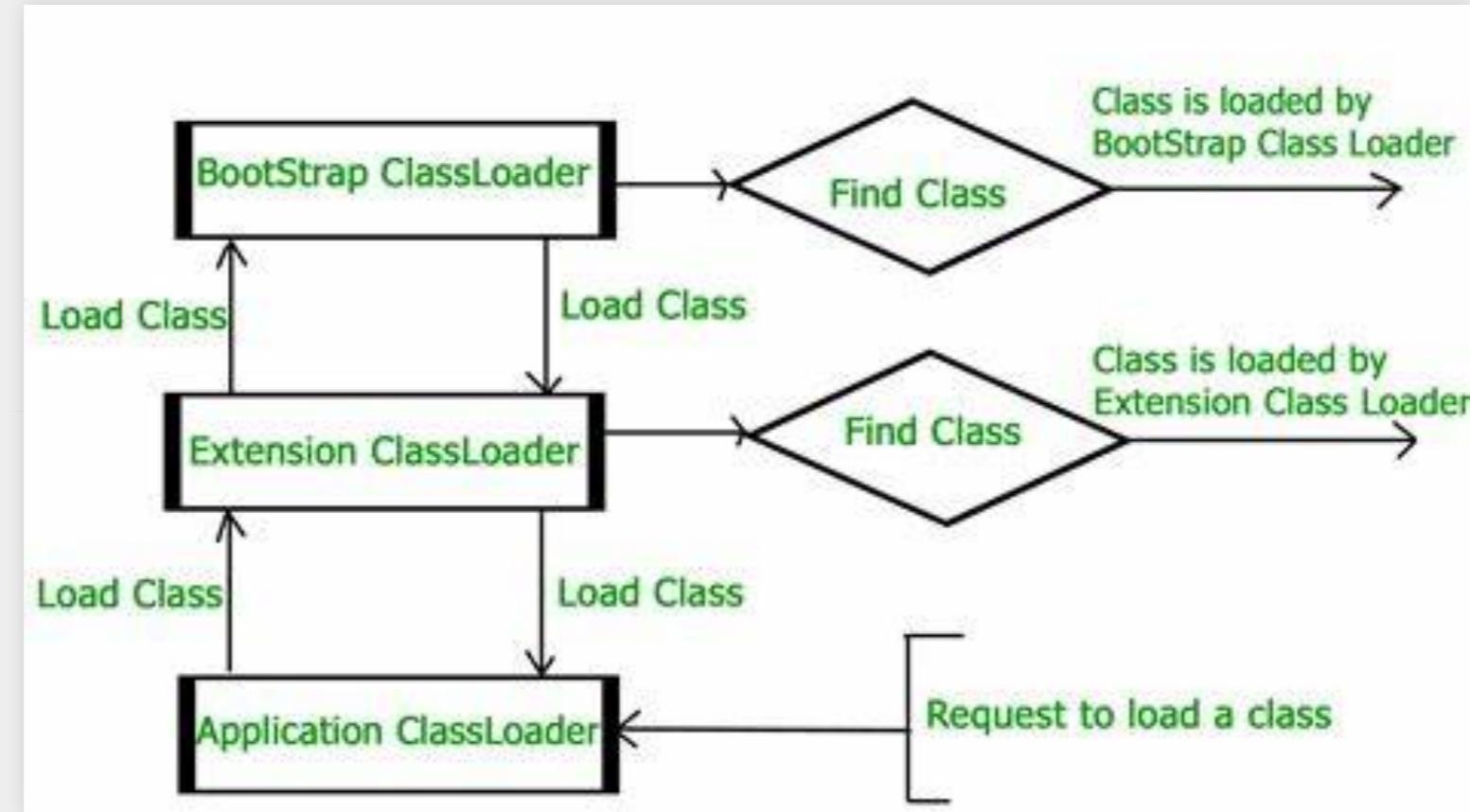


- Java源程序编译为字节码（.class文件）
- JVM载入字节码并对其进行解释执行

JVM

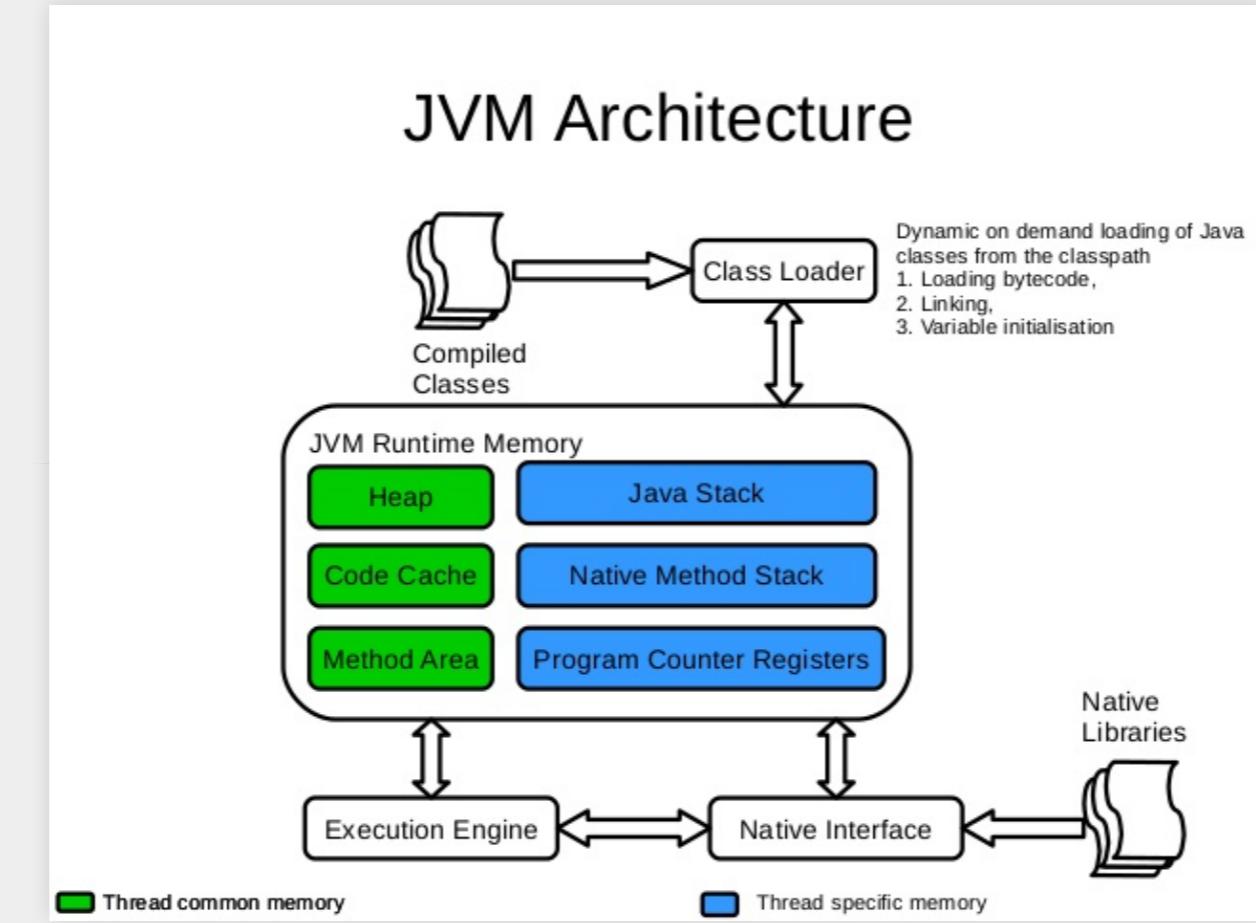
- 根据classpath查找依赖的类并进行加载，完成链接功能
- 创建主方法对应的栈帧和局部变量
- 启动一个线程执行主程序
- 管理堆中的资源，对不再使用的内存进行回收。这个功能叫做垃圾回收（Garbage collection, GC）
- 当线程终止时回收对应的资源
- 程序终止时，杀死所有剩余的线程并退出JVM

JVM加载



图片来源：<https://www.geeksforgeeks.org/jvm-works-jvm-architecture/>

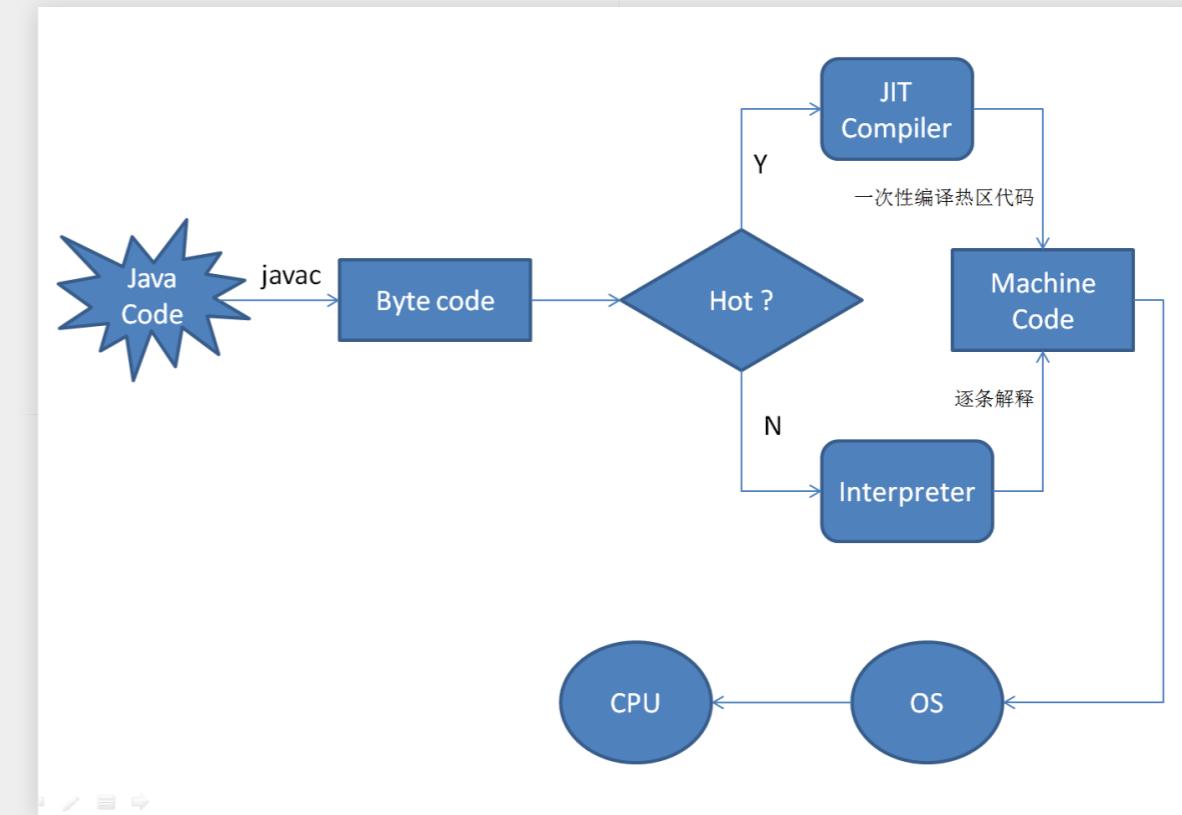
JVM体系结构



图片来源：<https://www.slideshare.net/OmarBashir2/an-introduction-to-java-compiler-and-runtime>

即时编译

- 即时编译（Just-in-time）是一种提高Java字节码执行速度的方法
- 基本原理：将代码或者字节码编译成二进制文件
- 实际过程：JVM统计字节码的执行热度，仅仅执行会多次执行的代码块/字节码



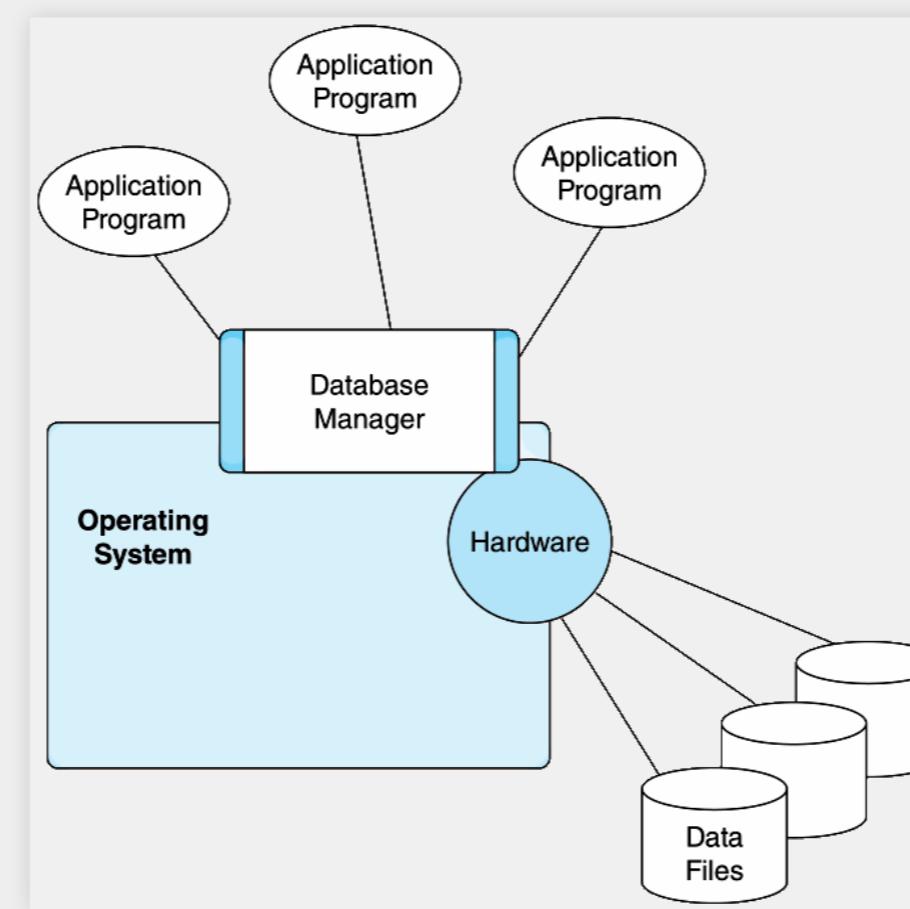
图片来源：<https://www.ibm.com/developerworks/cn/java/j-lo-just-in-time/>

JVM与二进制执行的对比

- JVM执行速度通常不如原生的二进制执行文件
- 但是JVM可以运行在任何体系结构和操作系统上
- 因此，大部份Java代码编写一次之后可以运行在不同的平台上，降低了开发和维护的复杂性

数据库软件

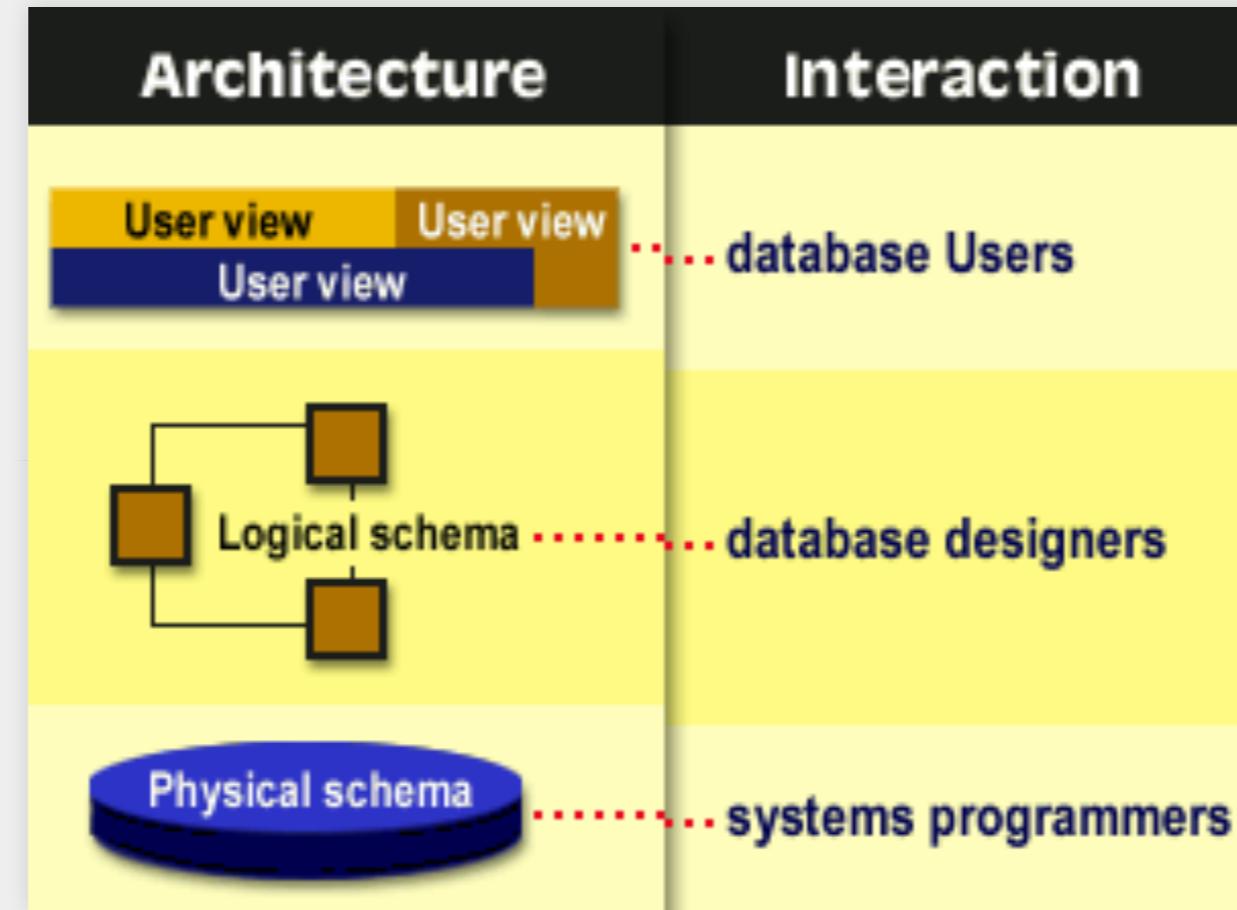
- 数据库为数据的组织、存储、访问和计算提供了抽象
- 数据库是许多应用程序的基础组成部分（网站、大数据计算等等）



数据库视图

- 数据库视图(Database schema)定义了数据库的访问
- 通常数据库包含两层视图，有的说法也加入用户视图：
 - 物理视图 (Physical Schema) : 数据库开发者所使用的数据视图
 - 逻辑视图 (Logical Schema) : 基于数据库的系统开发者所使用的数据视图
 - 用户视图 (User view) : 出于数据访问控制等原因，对不同用户创建的数据视图

数据库视图



图片来源：<https://www.relationaldbdesign.com/database-design/module3/schema-architecture.php>

数据库索引

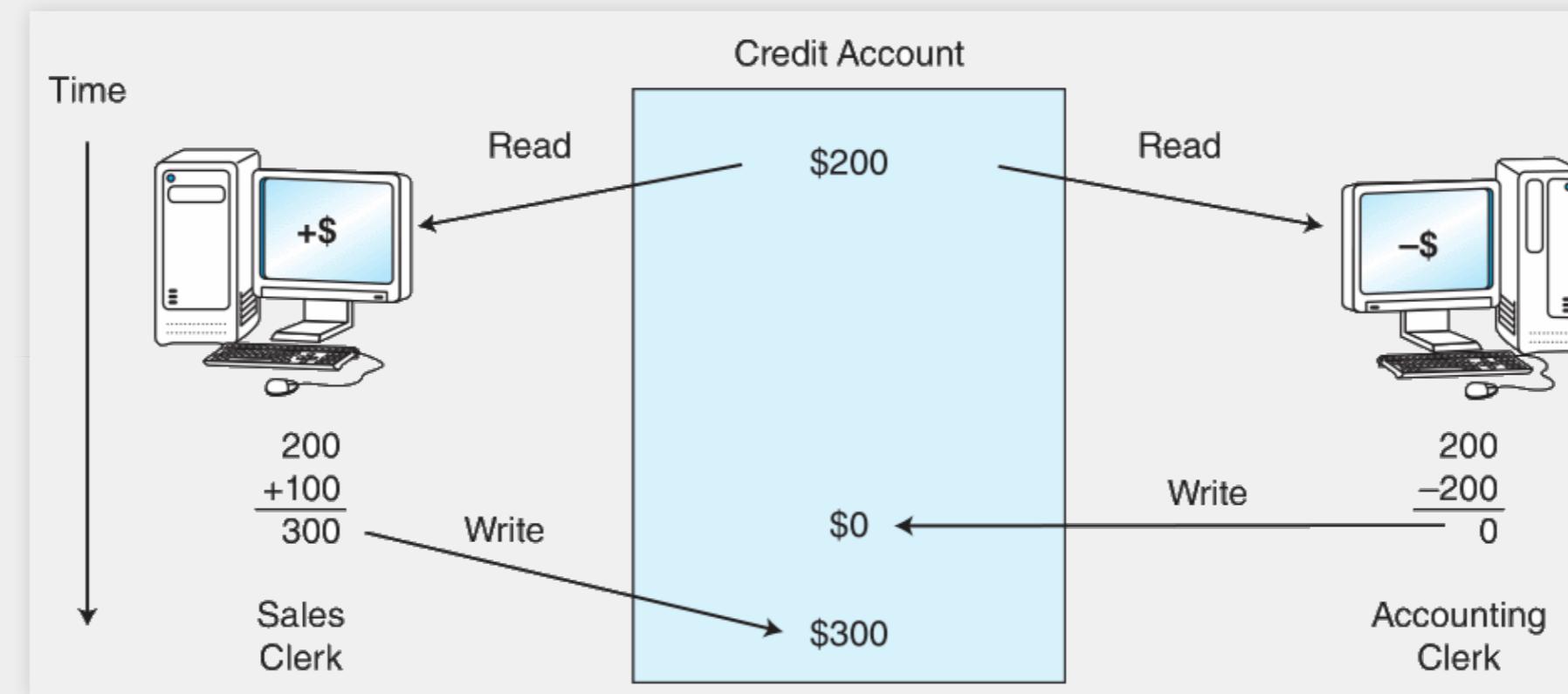
- 为了加速数据的查找和访问，数据库支持对某些数据建立索引
- 通常索引采用B+树实现，这是一种采用多叉树优化I/O访问的数据结构

事务

- 事务(Transaction)是数据库的一个重要功能，用于保证多用户并行处理时的系统一致性
- 事务具有下面四个性质，简记为ACID：
- 原子性 (Atomicity)：同一个事务中的操作要么全部执行，要么全部不执行
- 一致性 (Consistency)：所有的数据更新必须满足相关约束
- 隔离性 (Isolation)：任意两个事务之间的执行是不会互相影响的
- 持久性 (Durability)：成功的事务会尽快写入持久存储介质

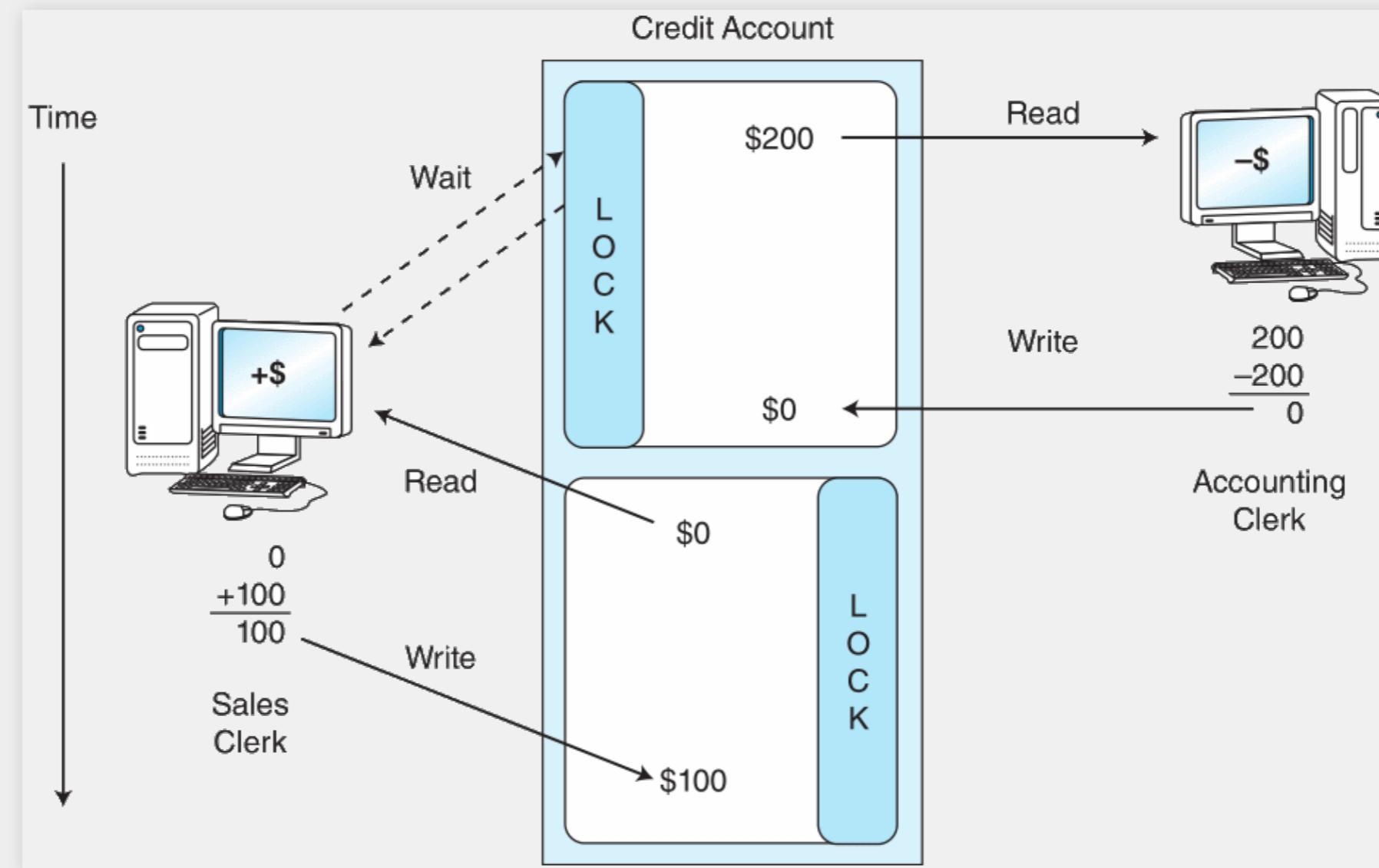
事务反例

- 缺少事务管理的数据库可能因为竞争而数据不一致的结果



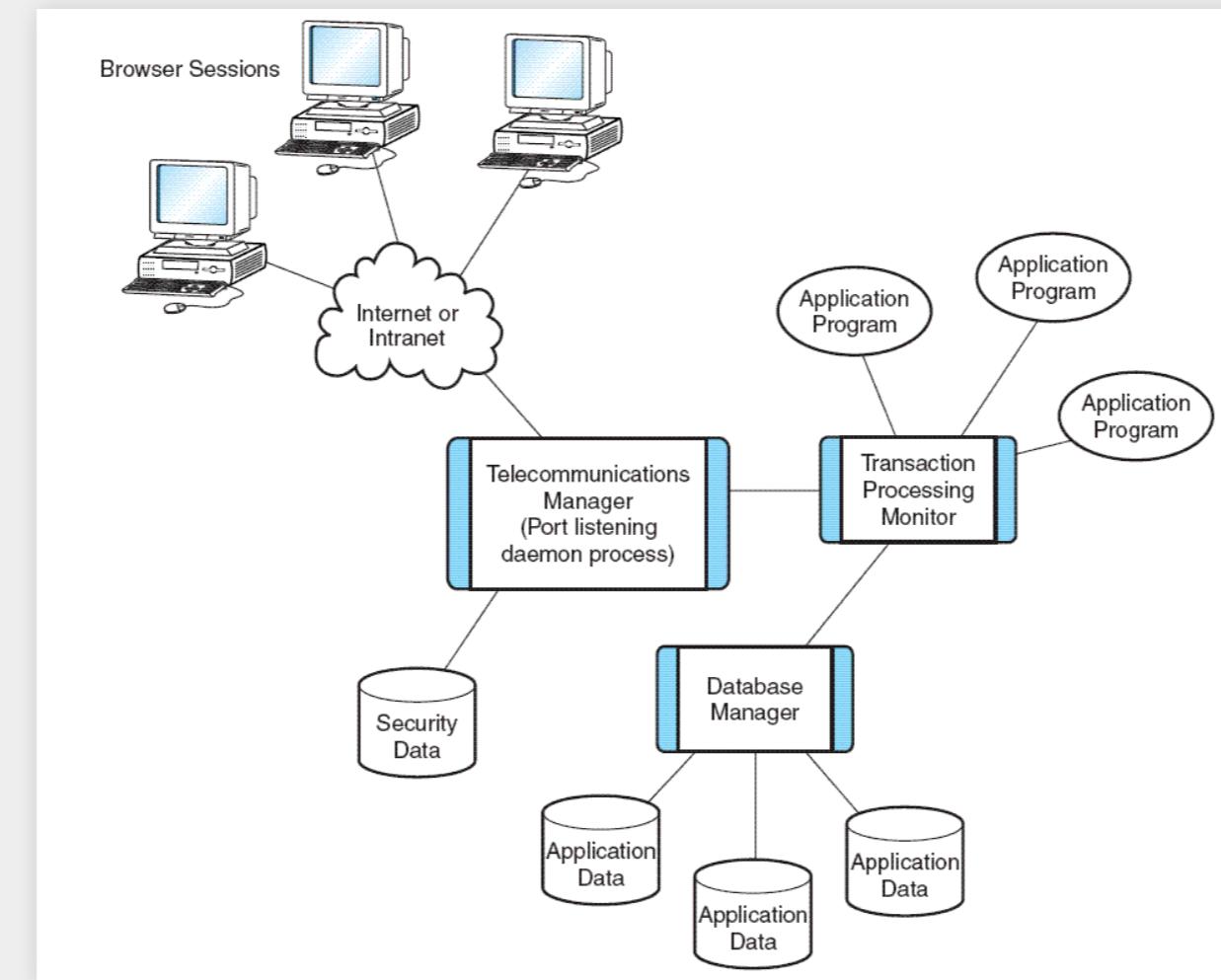
事务示例

- 加入事务管理之后，可以避免竞争，保证了数据的正确性



事务系统示例

- IBM CICS系统：大型机交易处理系统



第十一讲结束

本期内容总结

- 编程工具
 - 汇编器、加载和链接器
 - 编程语言
 - 编译过程
 - JVM
- 数据库工具
 - 数据库视图
 - 数据库事务



Q & A