第十一章 编译系统和运行系统

本章内容

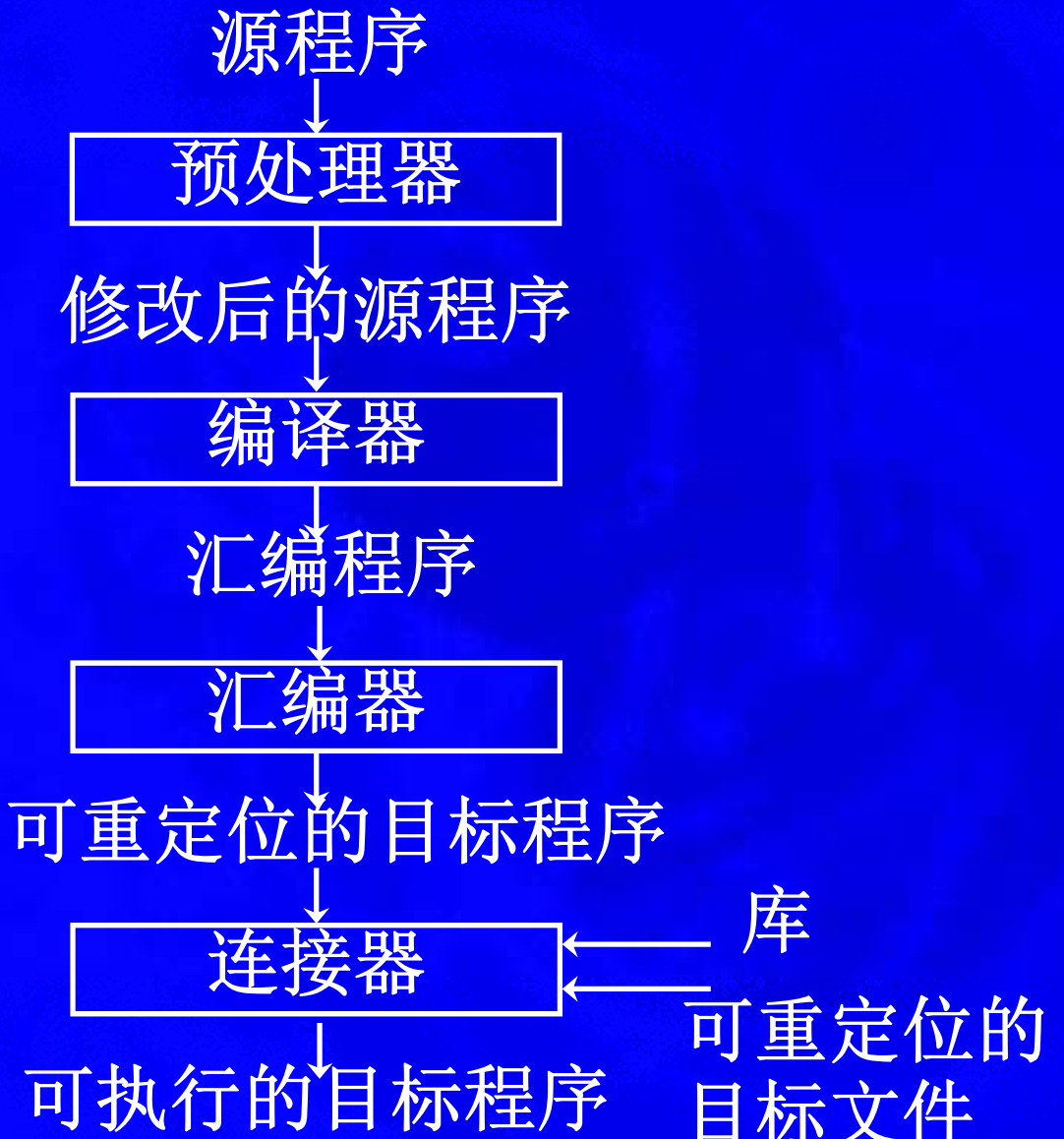
- **C**语言编译系统
  - 预处理器、编译器、汇编器、连接器
  - 目标文件的格式、静态库、动态连接
- **Java**运行系统
- 无用单元收集（垃圾收集）

引入本章的目的

- 掌握从源程序到可执行目标程序的实际处理过程
- 对实际参与软件开发是直接有用的

11.1 C语言编译系统

- C源程序可以分成若干个模块(文件)
- 分别进行预处理、编译和汇编、形成可重定位的目标文件
- 目标文件和必要的库文件连接成一个可执行的目标文件
- gcc和cc是编译驱动程序的名字



11.1 C语言编译系统

main.c

```
(1) #if 1
(2) int buf[2];
(3) #else
(4) int buf[2] = {10,20};
(5) #endif
(6) void swap();
(7) #define A buf[0]
(8) int main()
(9) {
(10)scanf("%d, %d", buf, buf+1);
(11)swap();
(12)printf("%d, %d",A, buf[1]);
(13)return 0;
(14) }
```

swap.c

```
(1) extern int buf[2];
(2) int *bufp0 = buf;
(3) int *bufp1;
(4) void swap()
(5) {
(6) int temp;
(7)bufp1 = buf+1;
(8)temp = *bufp0;
(9)*bufp0 = *bufp1;
(10)*bufp1 = temp;
(11) }
```

11.1 C语言编译系统

11.1.1 预处理器

- gcc首先调用预处理器cpp，将源程序文件翻译成一个ASCII中间文件，它是经修改后的源程序
- cpp实现以下功能
  - 文件包含
  - 宏展开
  - 条件编译

11.1 C语言编译系统

main.c

```
(1) #if 1
(2) int buf[2];
(3) #else
(4) int buf[2] = {10,20};
(5) #endif
(6) void swap();
(7) #define A buf[0]
(8) int main()
(9) {
(10)scanf("%d, %d", buf, buf+1);
(11)swap();
(12)printf("%d, %d",A, buf[1]);
(13)return 0;
(14) }
```

main.i

```
(1) # 1 “main.c”
(2)
(3) int buf[2];
(4)
(5)
(6) void swap();
(7)
(8) int main()
(9) {
(10) scanf("%d, %d", buf, buf+1);
(11) swap();
(12) printf("%d,%d",buf[0], …);
(13) return 0;
(14) }
```

11.1 C语言编译系统

11.1.2 汇编器

- GCC系统的编译器cc1产生汇编代码
- 最简单的汇编器对输入进行两遍扫描

11.1 C语言编译系统

- 例 一段汇编代码

```
.L2:
        cmpl $0,-4(%ebp)
        jne .L6
        jmp .L11
.L11:
        cmpl $0,-8(%ebp)
        jne .L6
        jmp .L12
.L12:
        jmp .L5
        .p2align 4,,7
.L6:
```

第一编扫描建立符号表，包括代码标号.L2、.L11等

第二遍扫描依据符号表中的信息来产生可重定位代码

11.1 C语言编译系统

11.1.2 汇编器

- GCC系统的编译器cc1产生汇编代码
- 最简单的汇编器对输入进行两遍扫描
- 一遍扫描完成汇编代码到可重定位目标代码的翻译也是完全可能的

11.1 C语言编译系统

- 例 一段汇编代码

```
.L2:
        cmpl $0,-4(%ebp)
        jne .L6                    建.L6的回填链
        jmp .L11
.L11:
        cmpl $0,-8(%ebp)
        jne .L6                    加入.L6的回填链
        jmp .L12
.L12:
        jmp .L5
        .p2align 4,,7
.L6:                               顺.L6回填链进行回填
```

11.1 C语言编译系统

11.1.2 汇编器

- GCC系统的编译器cc1产生汇编代码
- 最简单的汇编器对输入进行两遍扫描
- 一遍扫描完成汇编代码到可重定位目标代码的翻译也是完全可能的
- 用 **gcc –S main.c**

可以得到汇编文件**main.s**

- 用 **as –o main.o main.s**

可以将**main.s**汇编成可重定位目标文件**main.o**

11.1 C语言编译系统

11.1.3 连接器

目标模块或目标文件的形式

- 可重定位的目标文件
- 可执行的目标文件
- 共享目标文件
  - 一种特殊的可重定位目标文件
  - 在装入程序或运行程序时，动态地装入到内存并连接

11.1 C语言编译系统

- 连接是一个收集、组织程序所需的不同代码和数据的过程，以便程序能被装入内存并被执行
- 连接的时机
  - 编译时，装入时，或运行时
- 静态连接器
- 动态连接器
- 可重定位目标模块的组成？（回顾例子）

先 前 例 题

一个C语言程序及其在X86/Linux操作系统上的编译结果如下。根据所生成的汇编程序来解释程序中四个变量的存储分配、生存期、作用域和置初值方式等方面的区别

```
static long aa = 10;
short bb = 20;
func( ) {
    static long cc = 30;
    short dd = 40;
}
```

先 前 例 题

```
static long aa = 10;
short bb = 20;
```

```
func( ) {
static long cc = 30;
short dd = 40; }
```

```
.data
.align 4
   .type aa,@object
   .size aa,4
aa:
   .long 10
.globl bb
   .align 2
   .type bb,@object
   .size bb,2
bb:
   .value 20
```

```
   .align 4
   .type  cc.2,@object
   .size   cc.2,4
cc.2:
   .long 30
.text
   .align 4
.globl func
func:
   . . .
   movw $40,-2(%ebp)
   . . .
```

11.1 C语言编译系统

- 一个重定位模块M可能定义和引用的符号
  - 全局符号 指那些在模块M中定义，可以被其它模块引用的符号
  - 局部符号 指那些在模块M中定义，且只能在本模块中引用的符号
  - 外部符号 指那些由模块M引用并由其它模块定义符号
- 符号解析
  - 识别各个目标模块中定义和引用的符号，为每一个符号引用确定它所关联的一个同名符号的定义
- 重定位

11.1 C语言编译系统

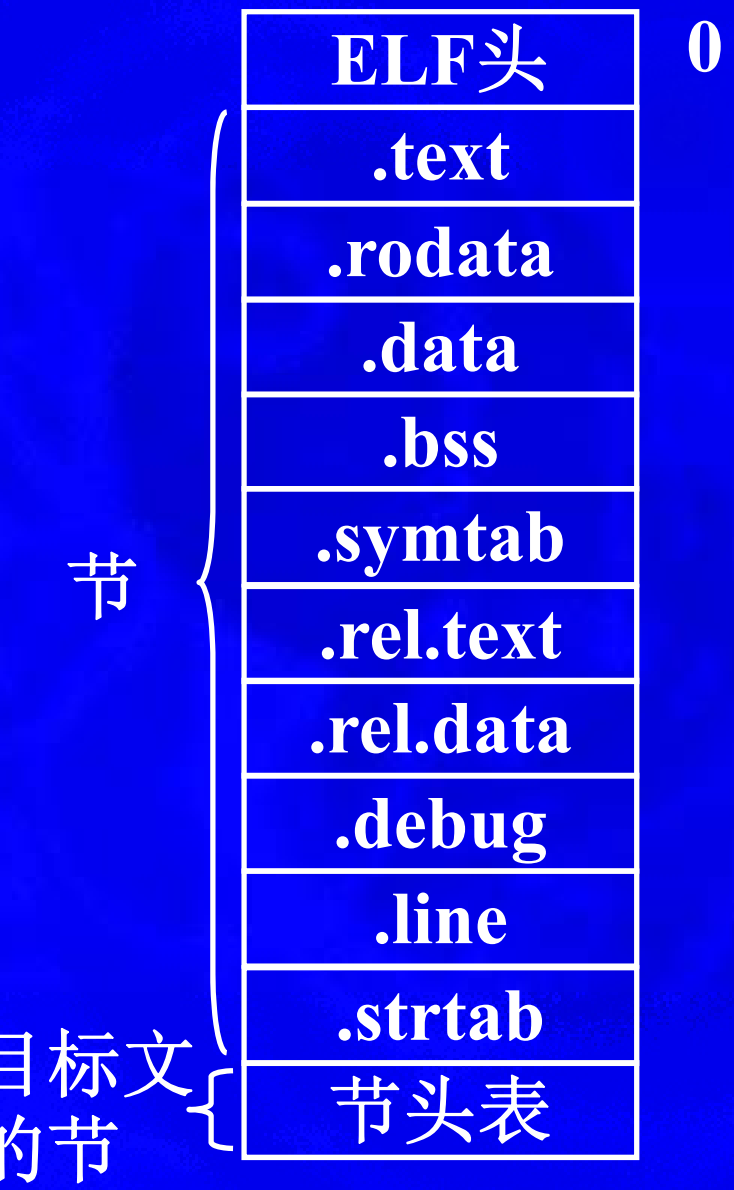
11.1.4 目标文件的格式

- 目标文件格式随系统不同而不同
- 介绍**Unix**的**ELF**（***Executable and Linkable Format***）格式
- **Linux**、**System V Unix**的后期版本、**BSD Unix**变体和**Sun Solaris**，都使用**Unix**的**ELF**格式

11.1 C语言编译系统

ELF头

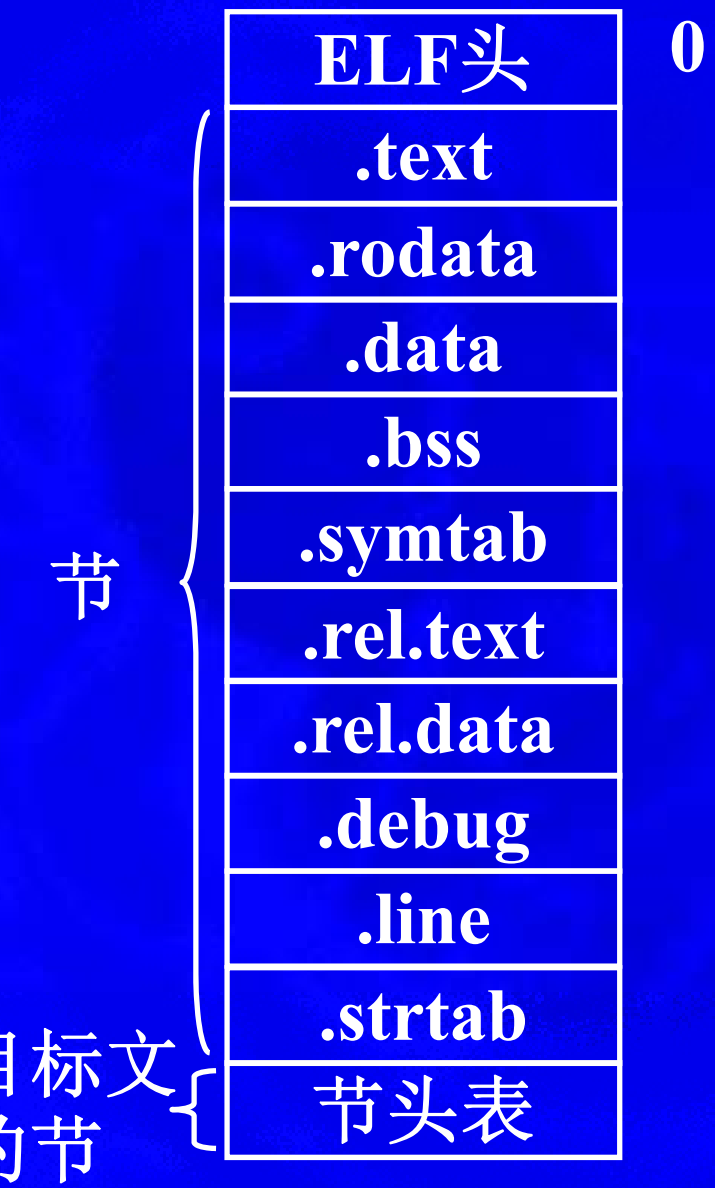
- 描述了字的大小
- 产生此文件的系统的字节次序
- 目标文件的类型
- 机器类型
- 节头表的位置、条目多少
- 其它



11.1 C语言编译系统

节头表

- 描述目标文件中各节的位置和大小
- 处于目标文件的末尾



11.1 C语言编译系统

.text节

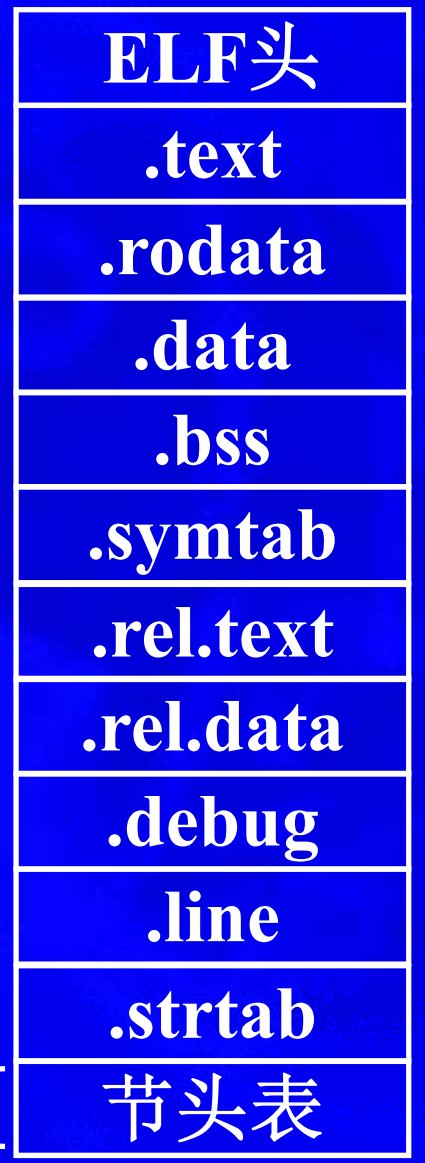
被编译程序的机器代码

.rodata节

诸如**printf**语句中的格式串和**switch**语句的跳转表等只读数据

.data节

已初始化的全局变量



11.1 C语言编译系统

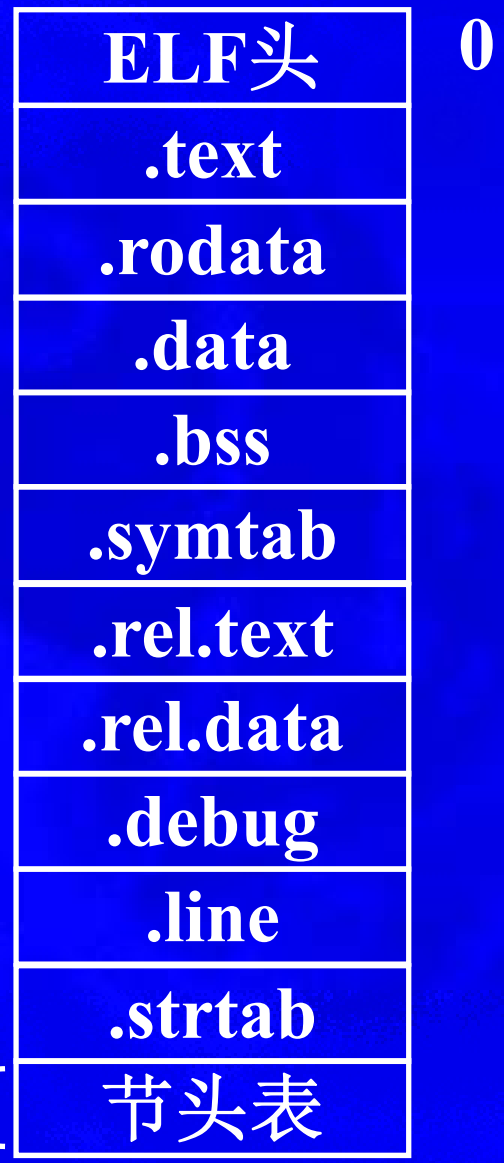
.bss节（.comm 节）

未初始化的全局变量

在目标文件中不占实际的空间

.symtab节

记录在该模块中定义和引用的函数和全局变量的信息的符号表



11.1 C语言编译系统

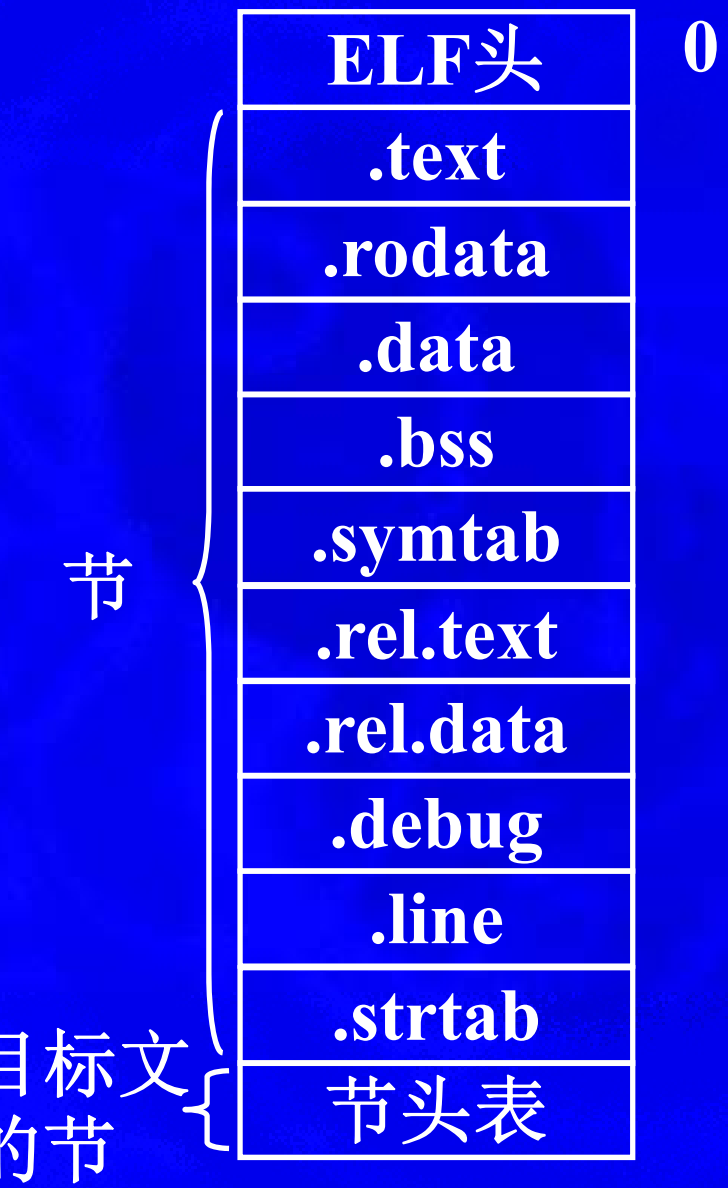
.symtab节

- Type
  - FUNC
  - OBJECT
- Bind
  - GLOBAL
  - LOCAL
  - EXTERN

11.1 C语言编译系统

.symtab节

- Name
- Value: 符号的地址
  - 偏移地址，或
  - 绝对地址
- Size
  - 字节数



11.1 C语言编译系统

.rel.text节

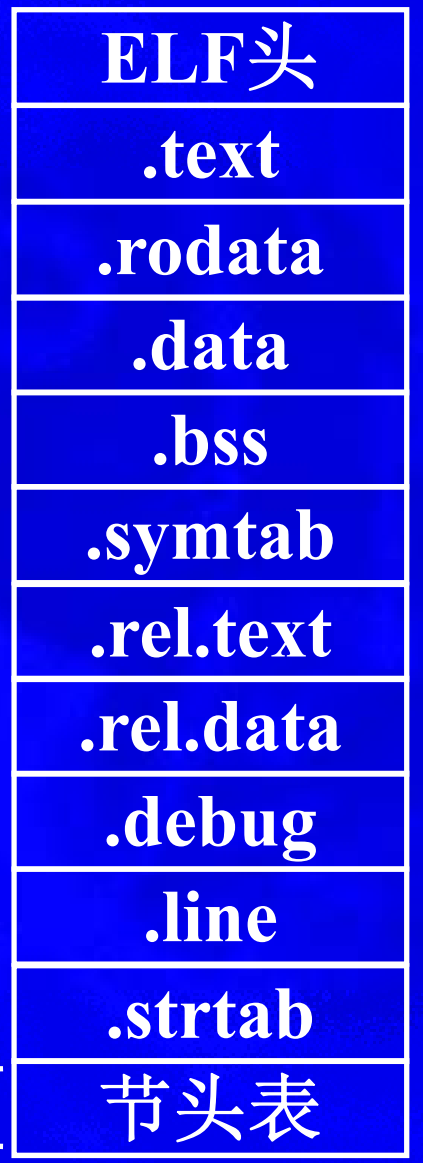
.text节中需要修改的单元的位置列表

如调用外部函数或引用全局变量的指令

.rel.data节

用于被本模块引用或定义的全局变量的重定位信息

要初始化的全局变量



11.1 C语言编译系统

.debug节

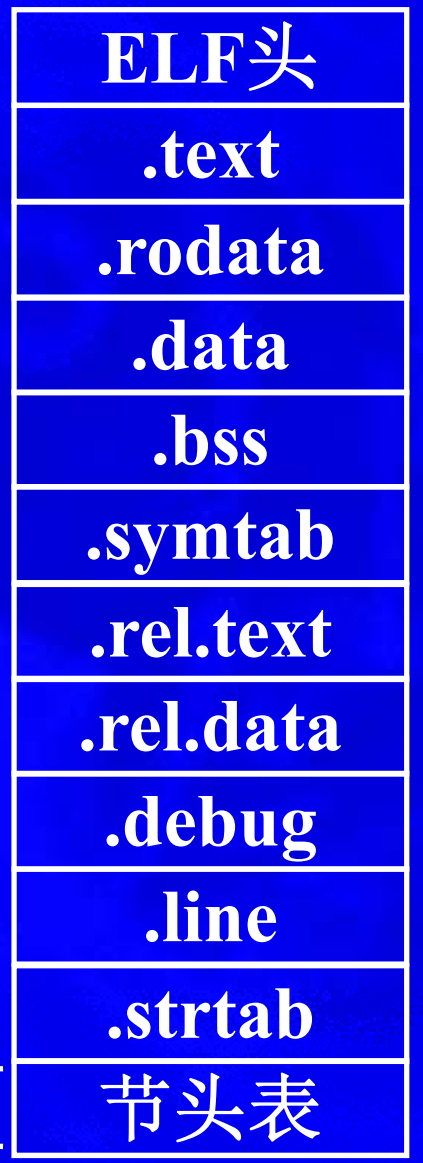
用于调试程序的调试符号表

.line节

源文件和**.text**节中的机器指令之间的行号映射

.strtab

一组有空结束符的串构成的串表



11.1 C语言编译系统

11.1.5 符号解析

- 将每个符号引用正确地与某可重定位模块的符号表中的一个符号定义相关联，从而确定各个符号引用的位置
- 在所有输入模块中都找不到被引用符号的定义，则打印错误消息并结束连接
- 需要定义解析规则

11.1 C语言编译系统

- 解析规则
  - 函数和已初始化的全局变量称为**强符号**；未初始化的全局变量称为**弱符号**
  - 不允许有多重的强符号定义
  - 出现一个强符号定义和多个弱符号定义时，选择强符号的定义
  - 出现多个弱符号定义时，选择任意一个弱符号的定义

11.1 C语言编译系统

11.1.6 静态库

- 将相关的可重定位目标模块打包成一个文件，作为连接器的输入
- 连接器仅复制库中被应用程序引用的模块

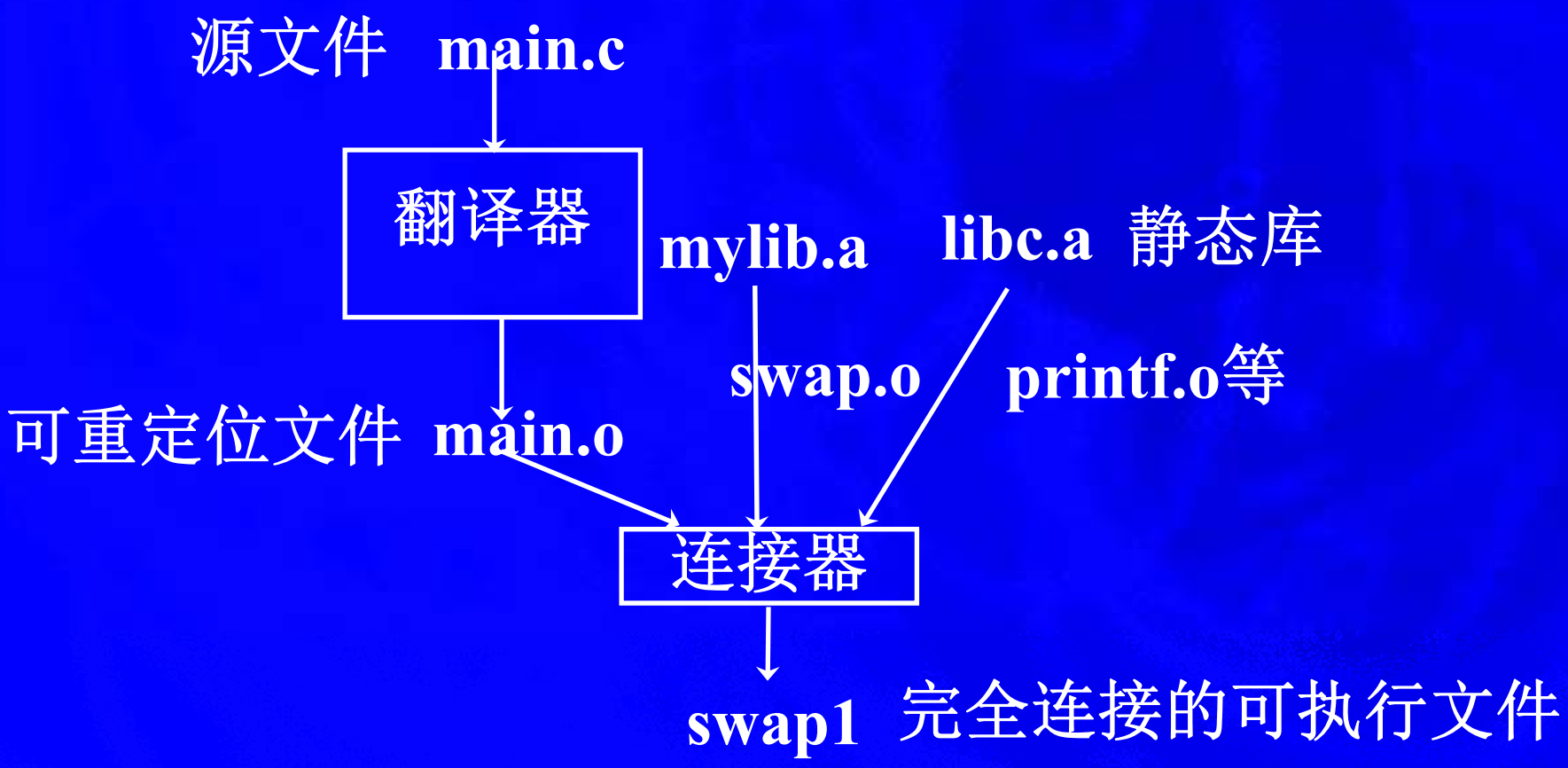
gcc –c swap.c ——编译

ar rcs mylib.a swap.o ——建库

gcc –static –o swap1 main.c /usr/lib/libc.a mylib.a ——生成可执行文件

11.1 C语言编译系统

和静态库连接



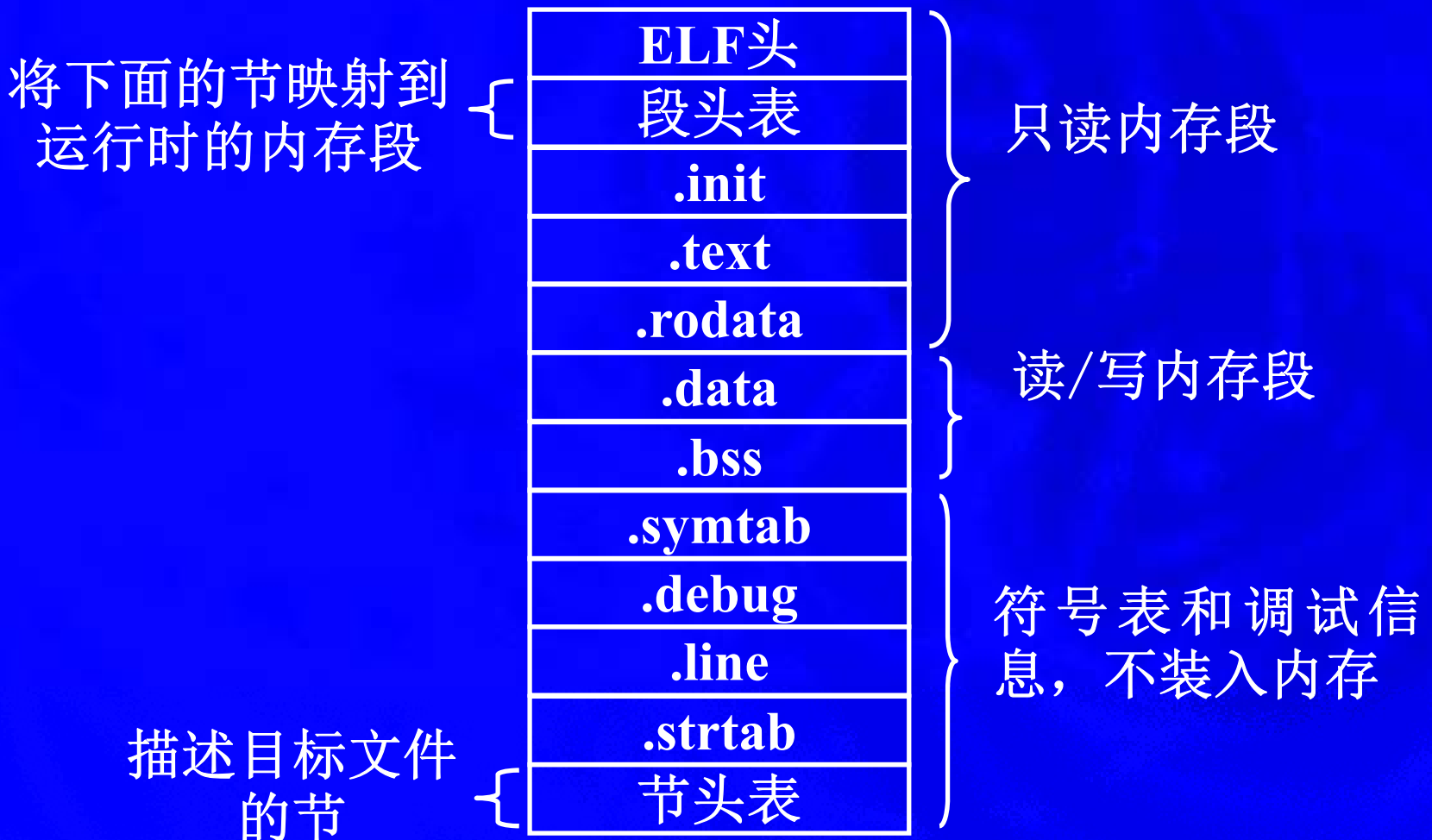
11.1 C语言编译系统

11.1.7 可执行目标文件及装入

- 可执行目标文件与可重定位目标文件格式类似
- 可执行目标文件的装入由加载器完成

11.1 C语言编译系统

典型的ELF可执行目标文件



11.1 C语言编译系统

Linux运行时的内存映像

11.1 C语言编译系统

- 这里描述的装入过程从概念上来说是正确的
- 若需要了解装入过程真正是怎样工作的，必须在理解了进程、虚拟内存和内存分页等概念以后

11.1 C语言编译系统

11.1.8 动态连接

- 静态库
  - 周期性地被维护和更新
  - 内存可能有多份**printf**和**scanf**的代码
- 共享库
  - 在运行时可以装到任意的内存位置，被内存中的进程共享

11.1 C语言编译系统

共享库以两种不同的方式被共享

- 共享库的代码和数据被所有引用该库的可执行目标文件所共享
- 共享库的.text节在内存中的一个副本可以被正在运行的不同进程共享

11.1 C语言编译系统


源文件
main.c
翻译器
（cpp,cc1,as）
libc.so
mylib.so
可重定位文件
main.o
重定位和符号表信息
连接器（ld）
部分连接的可执行目标文件
swap2
加载器（execve）
libc.so
mylib.so
代码和数据
动态连接器（ld-linux.so）
此时，动态连接器是内存中已完全连接的可执行代码

11.1 C语言编译系统

加载器通常装入和运行动态连接器

动态连接器接着完成连接任务

- 把**libc.so**的文本和数据装入内存并进行重定位
- 把**mylib.so**的文本和数据装入内存并进行重定位
- 重定位**swap2**中任何对**libc.so**或**mylib.so**定义的符号的引用
- 将控制传递给应用程序

11.1 C语言编译系统

11.1.9 处理目标文件的一些工具

ar 创建静态库，插入、删除、罗列和提取成

strings 列出包含在目标文件中的所有可打印串

strip 从一个目标文件中删除符号表信息

nm 列出一个目标文件的符号表中定义的符号

size 列出目标文件中各段的名字和大小

readelf 显示目标文件的完整结构，包括编码在ELF头中的所有信息。它包括了size和nm的功能

objdump 可以显示目标文件中的所有信息。其最有用的功能是反汇编.text节中的二进制指令

ldd 列出可执行目标文件在运行时需要的共享库

11.2 Java语言的运行系统

- **Java语言**
  - 简单性、分布性、安全性、可移植性等
  - 关心：平台无关性
- **Java虚拟机技术是实现Java平台无关性特点的关键**
- **Java运行系统就是Java虚拟机的一个实现**

11.2 Java语言的运行系统

11.2.1 Java虚拟机语言简介

Java程序首先由Java编译器把它编译成字节码，也就是JVML程序

- 常量池：存放各种字符串常量，类似于传统程序设计语言中的符号表
- 类成员信息：域信息表和方法信息表
- JVML指令序列

11.2 Java语言的运行系统

Java源程序中的方法：

```
int calculate (int i){

    int j =2;
    return ((i+j) ×(j-1));

}
```

对应的JVML指令序列：

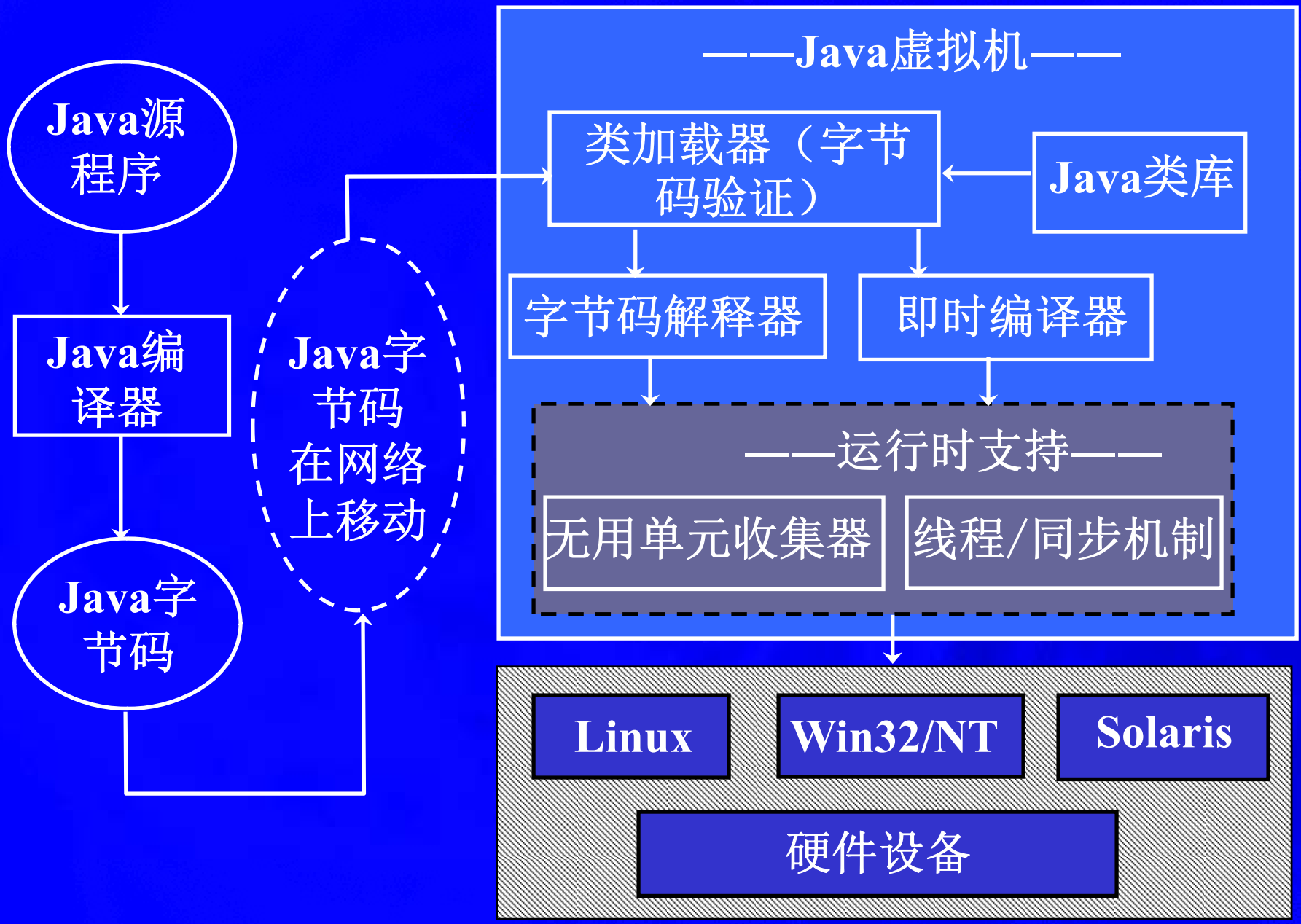
```
int calculate (int i)
iconst_2
istore_2
iload_1
iload_2
iadd
iload_2
iconst_1
isub
imul
ireturn
```

11.2 Java语言的运行系统

11.2.2 Java虚拟机

Java虚拟机一般由以下几个部分构成：

- 类加载器（字节码验证器）
- 解释器或/和编译器
- 包括无用单元收集器和线程控制模块在内的运行支持系统
- 另外还有一些标准类和应用接口的class文件库


Java源程序
Java编译器
Java字节码
Java字节码在网络上移动
——Java虚拟机——
类加载器（字节码验证）
Java类库
字节码解释器
即时编译器
——运行时支持——
无用单元收集器
线程/同步机制
Linux
Win32/NT
Solaris
硬件设备

11.2 Java语言的运行系统

- C语言
  - 数据栈用来存放生存期和过程一次活动的生存期一致的局部变量
  - 数据堆用来存放生存期和过程一次活动的生存期不一定一致的动态变量
    - 程序员通过**malloc**和**free**函数参与堆的管理
    - 不安全的一个根源（悬空指针、内存泄漏）
- Java语言
  - 数据栈：除对象和数组外，都分配在栈上
  - 数据堆：对象和数组分配在堆上
    - 出于安全的要求，程序员不参与堆管理

11.2 Java语言的运行系统

无用单元收集（俗称垃圾收集）

- 无用单元（理论上）
  - 那些在继续运行过程中不会再使用的数据单元
- 收集器采用稳妥策略
  - 实际上并非总能判断出一个数据记录的值以后是否还需要
  - 通过根集（***roots set***，在栈上）以及从根集开始的可达性来定义变量的活跃性
- 无用单元（实现上）
  - 通常指那些不可能从程序变量经指针链到达的堆分配记录

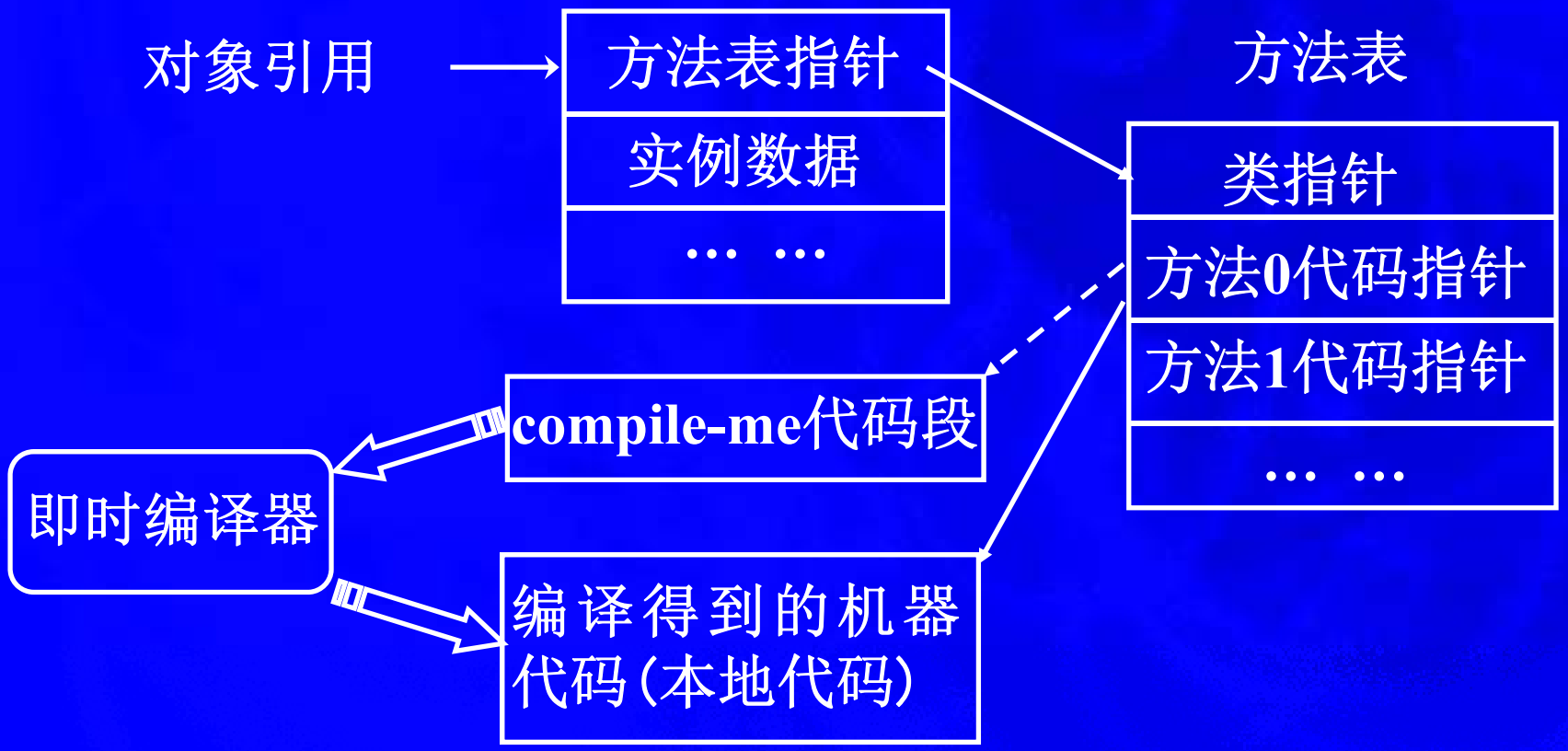
11.2 Java语言的运行系统

11.2.3 即时编译器

- 当一个类的某个方法第一次被调用时，虚拟机才激活即时编译器将它编译成机器代码
- 生成的代码的执行速度可以达到解释执行的10倍
- 但是执行过程不得不等待编译的结束，因而使得执行时间变长
- 很多虚拟机都会使用快速解释器和优化编译器的组合或者是简单编译器和复杂编译器的组合

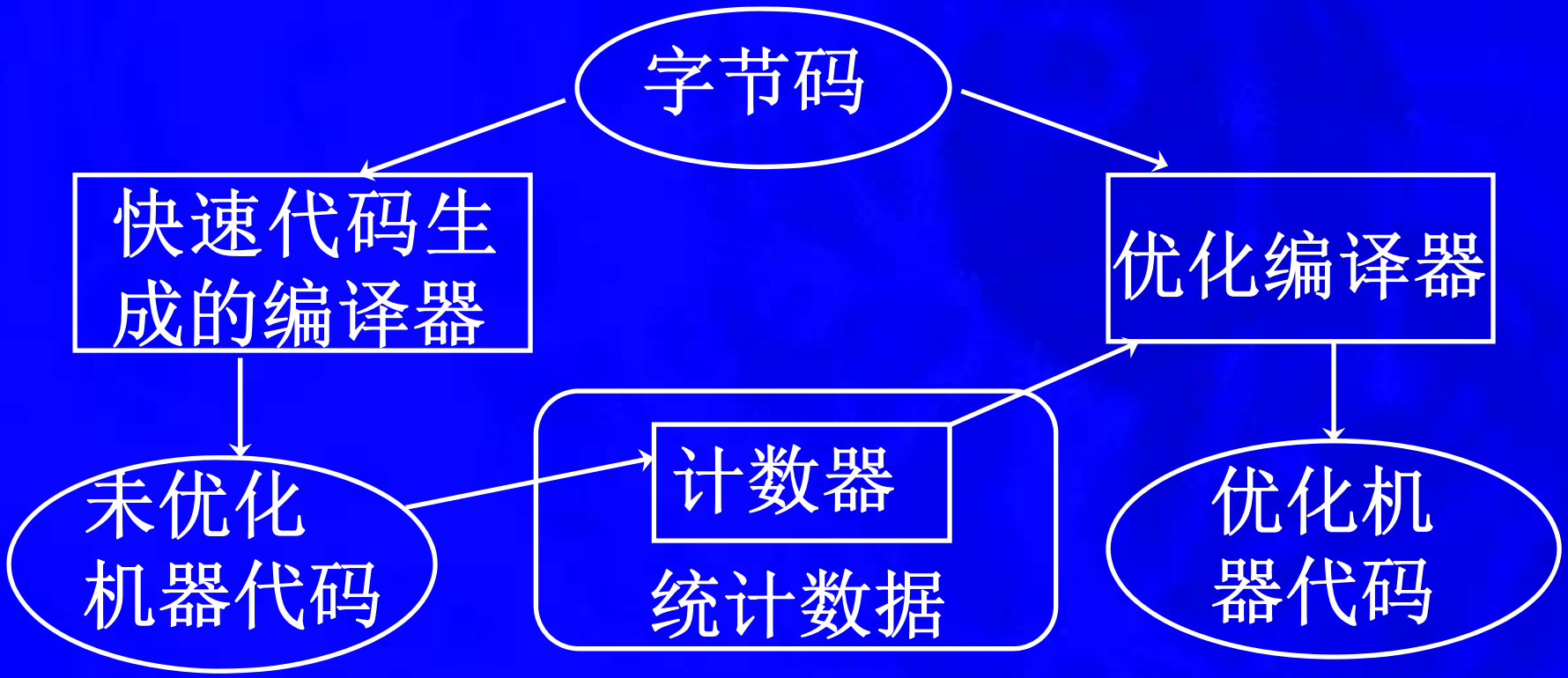
11.2 Java语言的运行系统

即时编译



11.2 Java语言的运行系统

重编译机制



11.3 无用单元收集

- 无用单元收集器需要根据数据的活跃性来判断哪些是无用单元
- 活跃性分析采用稳妥策略，是通过根集以及从根集开始的可达性来定义活跃性
- 从实现的角度，无用单元是那些不可能从程序变量经指针链到达的堆分配记录
- 无用单元收集可能需要来自编译器、操作系统和硬件方面的支持

- 本节简要介绍一些主要的无用单元收集方法，并且描述编译器和收集器之间的一些相互影响

11.3 无用单元收集

11.3.1 标记和清扫

- 方法概述：首先标记堆上所有可达记录，然后回收未被标记的记录
  - 根集包含了全局变量、活动记录栈中的局部变量和被活跃着的过程使用的寄存器
  - 堆上活跃记录的集合是从根集开始的任何一条指针路径上的记录的集合
  - 任何图遍历算法，都可用于标记所有的可达记录
  - 清扫从堆的首地址开始，逐个记录地考察整个堆，寻找未被标记的记录，把它们链成一个空闲链表

11.3 无用单元收集

11.3.1 标记和清扫

- 传统的标记和清扫方法的问题
  - 碎片问题：当要分配一个*n*字节大小的记录时，发现有很多小于*n*字节的空闲块存在，但就是没有合适的空闲块可分配给这个记录
  - 引用局部性问题：使用块和空闲块相互交织，使得当前要使用的各个活跃记录被分散到很多的虚拟内存页中，这些页在内存中可能被频繁地换进换出

11.3 无用单元收集

11.3.2 引用计数

- 方法概述：通过记住有多少指针指向每个记录来直接完成标记，引用计数存在各记录中
  - 编译器需要在每个出现指针存储的地方生成额外的指令，以调整一些引用计数器的值
  - 当一个记录的引用计数值为0的时候，就可以把该记录加入空闲链表
  - 被回收记录本身的指针域都要一一检查，它们所指向的记录的引用计数值也都要减1

11.3 无用单元收集

11.3.2 引用计数

- 引用计数方法的问题
  - 碎片和引用局部性问题仍然存在
  - 并非总有效：对于循环的数据结构会失效，因为这些记录的引用计数也永远不可能减到零
  - 代价高，因为每当执行指针存储的时候，都需要执行额外的指令来调整一些引用计数

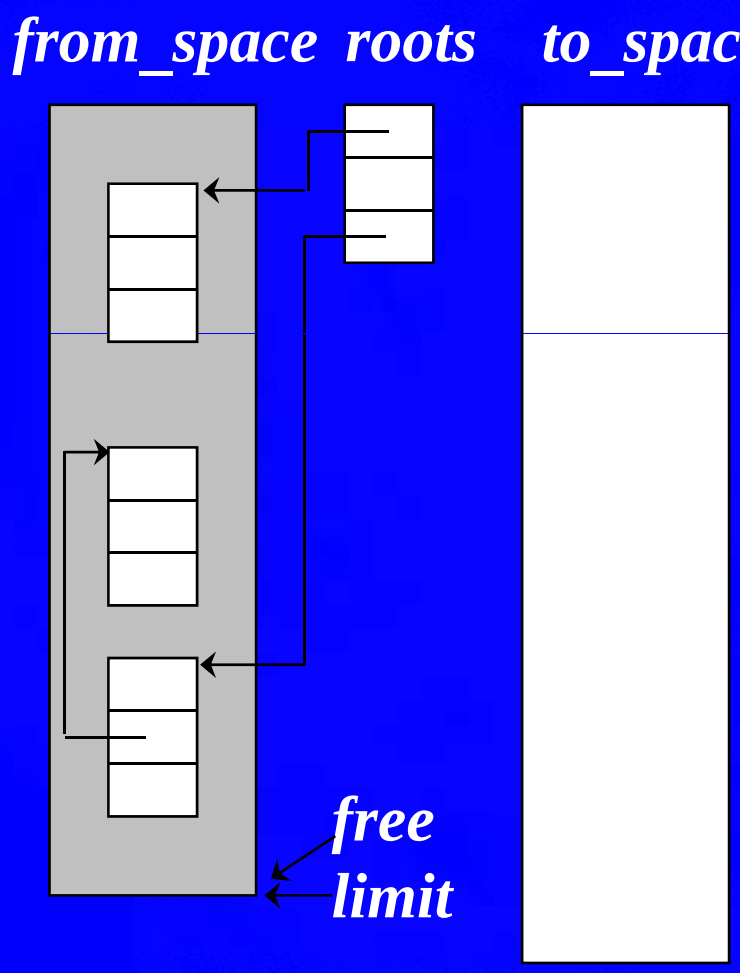
  - 引用计数收集已经被跟踪型收集代替，标记和清扫收集方法就是一种跟踪型收集方法

11.3 无用单元收集

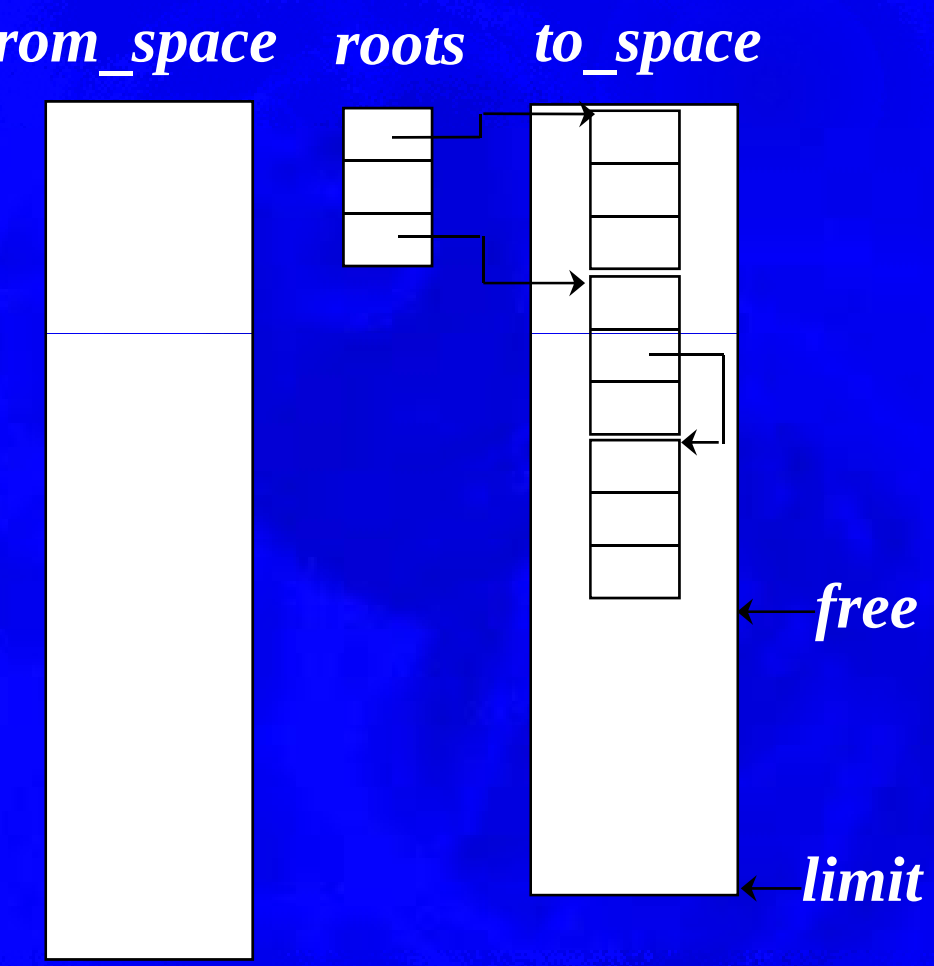
11.3.3 拷贝收集

- 方法概述
  - 这个算法和标记和清除算法一样，也遍历可达记录所组成的有向图，只不过它在遍历的同时进行清扫，并且这种清扫主要是拷贝活跃记录
  - 这种方法将堆空间分成大小相等的两块，每块都是连续的区域

11.3 无用单元收集



(a) 收集前



(b)收集后

11.3 无用单元收集

11.3.3 拷贝收集

- 优缺点
  - 从理论上来说，只要有足够的内存，可得到很高的效率
  - 可以将活跃数据紧缩在一起，碎片情况消失，引用局部性得到改善
  - 需要的空间是实际需要空间的两倍
  - 拷贝大记录的代价太大

11.3 无用单元收集

11.3.4 分代收集

- 原理
  - 很多程序在运行过程中，新建记录很可能很快死去，不会出现对它的拷贝；反过来，一个记录在几次收集后还可到达的话，那么很可能还会活跃到许多次收集后
  - 收集器应该将精力集中到“年轻”的数据上，因为这里有相对高的无用单元比率

11.3 无用单元收集

11.3.4 分代收集

- 基本做法
  - 堆被分成代，最年轻的记录在G_0代，G_i ($i > 0$)代中的记录比G_{i-1}代中的任何记录都要老。越年轻的代越被频繁地收集
  - 对G_0代进行收集时，这时的根集不仅仅是程序变量，它还包括G_1，G_2,…中指向G_0的指针。幸好，老记录指向年轻得多的记录的情况极少出现，通常是较新的记录指向老记录
  - 为了避免确定G_0的根集时在G_1，G_2,…中查找，需要编译器提供一些支持，方法有多种

11.3 无用单元收集

11.3.5 渐增式收集

- 缘由
  - 虽然收集时间的总和只占整个程序运行时间很小的百分比，但是收集器仍然有可能偶尔将运行程序中断相对长的时间，对于交互式程序和实时程序来说，这一点是难以接受的
- 改进
  - 渐增式收集：程序运行和无用单元收集交错进行
  - 困难在于，当收集器在做遍历以得到一个可达记录图时，运行程序并没有停止修改可达记录图

11.3 无用单元收集

11.3.6 编译器与收集器之间的相互影响

- 对收集器的底层有下面这些基本要求
  - 能确定堆上分配的记录大小
  - 能定位包含在堆记录里的指针
  - 能定位所有在全局变量中的指针
  - 能在程序中任何可进行收集的地方找到所有在活动记录栈中和寄存器中的指针
  - 能找到所有由指针运算所产生的值指向的记录
  - 能在记录被移动时，更新所有涉及到的指针值
- 很多所需信息只有在编译时能够获得

例　题　1

如果cfile是一个C语言源程序（注意，该文件名没有后缀），在X86/Linux机器上，命令

cc cfile

的结果是错误信息

/usr/bin/ld: cfile: file format not recognized:
treating as linker script
/usr/bin/ld: cfile: 1: parse error
collect2: ld returned 1 exit status

请解释为什么会是这样的错误信息

例　题　2

下面是C语言的一个程序：

```
long gcd(p,q) long p,q; {
    if (p%q == 0) return q;
     else return gcd(q, p%q);
}
main() {
     printf("\n%ld\n",gcdx(4,12));
}
```

在X86/Linux机器上，用gcc命令得到的编译结果如下

```
In function ‘main’:undefined reference to ‘gcdx’
ld returned 1 exit status.
```

请问，这个gcdx没有定义，是在编译时发现的，还是在连接时发现的？试说明理由

例　题　3

一些C程序设计的教材上指出“在需要使用标准I/O库中的函数时，应在程序前使用

#include <stdio.h>

预编译命令，但在用**printf**和**scanf**函数时，则可以不要。”但事实上并非仅限于这两个函数，如下面的C程序编译后运行时输出字符A并换行，它并没有预编译命令**#include <stdio.h>**。试解释为什么

```
main() {
   putchar('A');    putchar('\n');
}
```

例　题　4

C的一个源文件可以包含若干个函数，该源文件经编译可以生成一个目标文件；若干个目标文件可以构成一个函数库

如果一个用户程序引用某函数库中某文件的某个函数，那么，在连接时的做法是下面三种方式的哪一种，说明理由

- 将该函数的目标代码连到用户程序
- 将该函数的目标代码所在的目标文件连到用户程序
- 将该函数库全部连到用户程序

例　题　5

cc是UNIX系统上C语言编译命令，–l是连接库函数的选择项。某程序员自己编写了两个函数库libuser1.a和libuser2.a（库名必须以lib为前缀），当用命令

cc test.c –luser1.a –luser2.a

编译时，报告有未定义的符号，而改用命令

cc test.c –luser2.a –luser1.a

时，能得到可执行程序。试分析原因

（备注：库名中的lib在命令中省略。该命令和命令cc test.c libuser1.a libuser2.a的效果一致）

例　题　5

cc test.c –luser1.a –luser2.a

解答

test.c	**libuser1.a**	**libuser2.a**
引用a	**定义b**	**定义a**
		引用b

例　题　6

cc是UNIX系统上C语言编译命令，–l是连接库函数的选择项。两个程序员分别编写了函数库libuser1.a和libuser2.a，当用命令

cc test.c –luser1.a –luser2.a

编译时，报告有重复定义的符号。而改用命令

cc test.c –luser2.a –luser1.a

时，能得到可执行程序。试分析原因

例　题　6

cc test.c –luser1.a –luser2.a

一种情况

test.c	libuser1.a	libuser2.a
引用a	定义a	定义b
引用b		定义a

若干人一起开发软件时有可能发生

a的使用局部于文件，应加static而未加

例　题　7

两个C文件link1.c和link2.c的内容分别如下

```
int buf[1] ={100};
```

和

```
extern int *buf;
main() { printf("%d\n", *buf); }
```

在X86/Linux经命令cc link1.c link2.c编译后，运行时产生如下的出错信息

```
Segmentation fault (core dumped)
```

请说明原因

例　题　7

```
int buf[1] ={100};
```

和

```
extern int  *buf;
main() { printf("%d\n", *buf); }
```

- 连接时不检查名字的类型
  - 虽对buf的类型持不同观点，但能连接成目标程序
- 连接时让不同文件中同一名字的地址相同
  - 运行时，在link2.c中，由于buf的内容是100，取*buf的值就是取地址为100的单元的内容。该地址不在程序数据区内，报错
- 若把这些代码放在一个文件中，编译时报错