中国科学技术大学
University of Science and Technology of China

编译和运行系统

《编译原理和技术(H)》

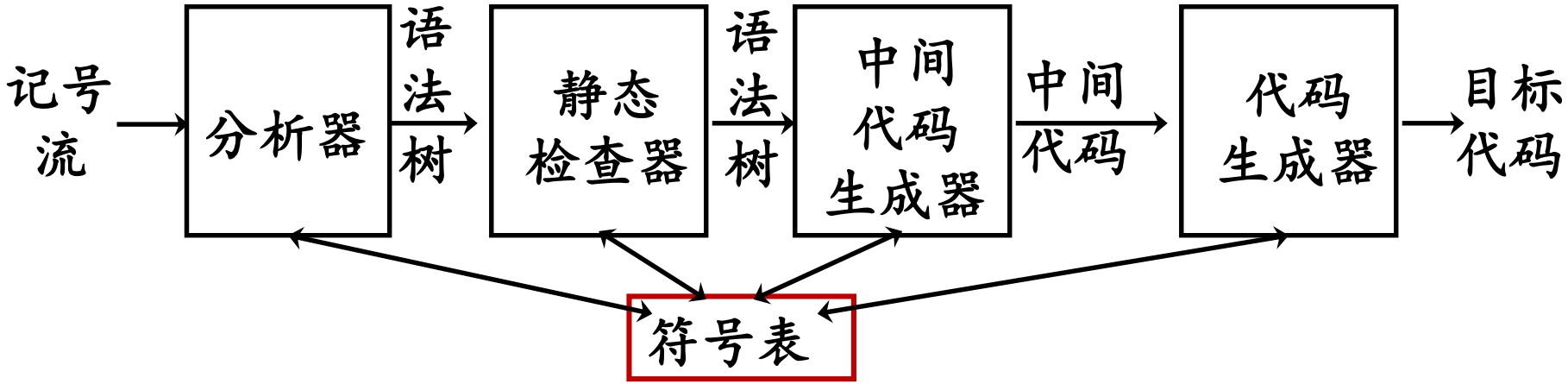
张昱

0551-63603804，yuzhang@ustc.edu.cn

中国科学技术大学

计算机科学与技术学院

本章内容



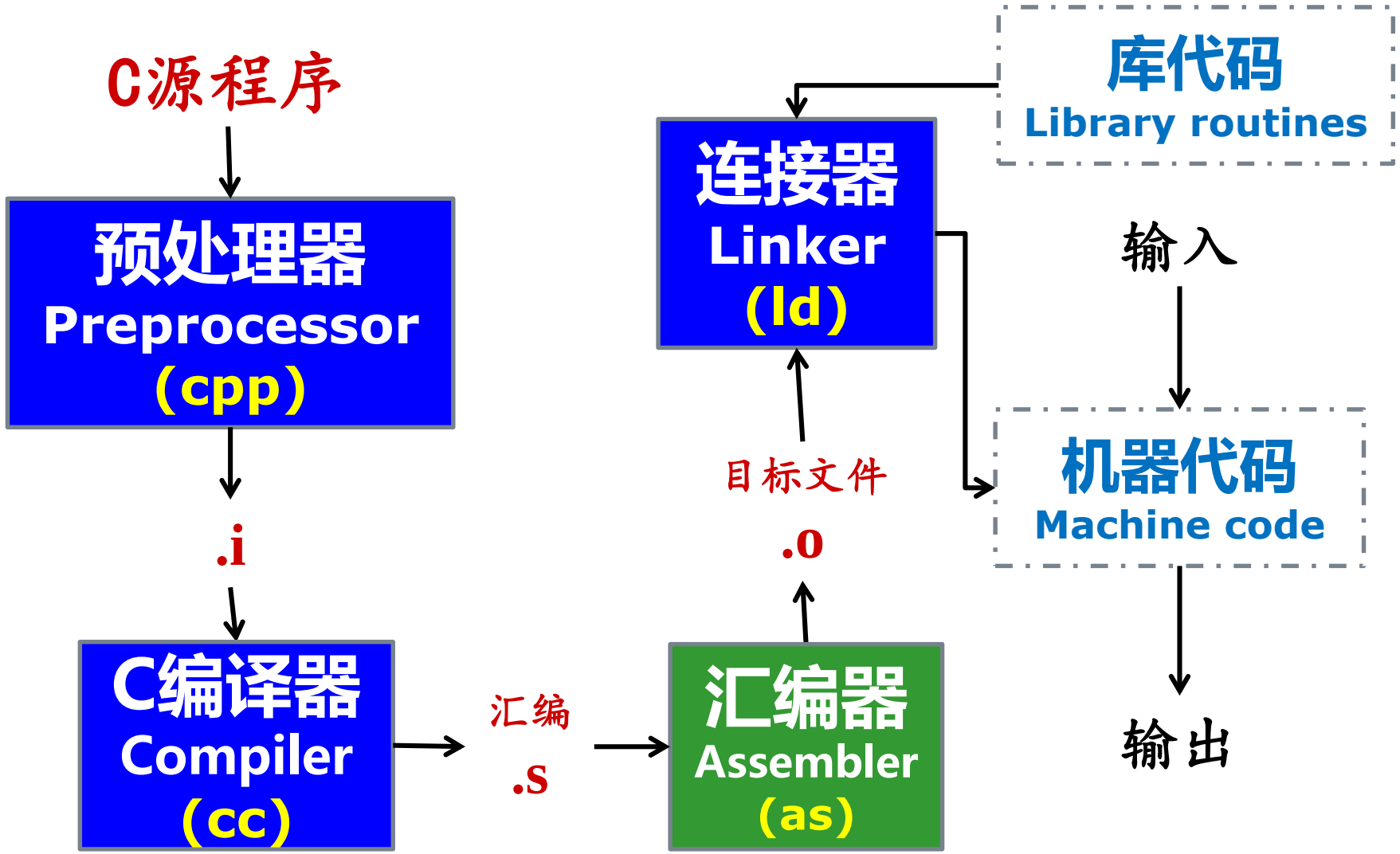
- (传统C)编译系统
  - 宏、汇编器、连接器
- Java运行系统

中国科学技术大学
University of Science and Technology of China

1. 编译系统

- C语言编译系统
- 宏
- 汇编器
- 连接器

C 编译系统

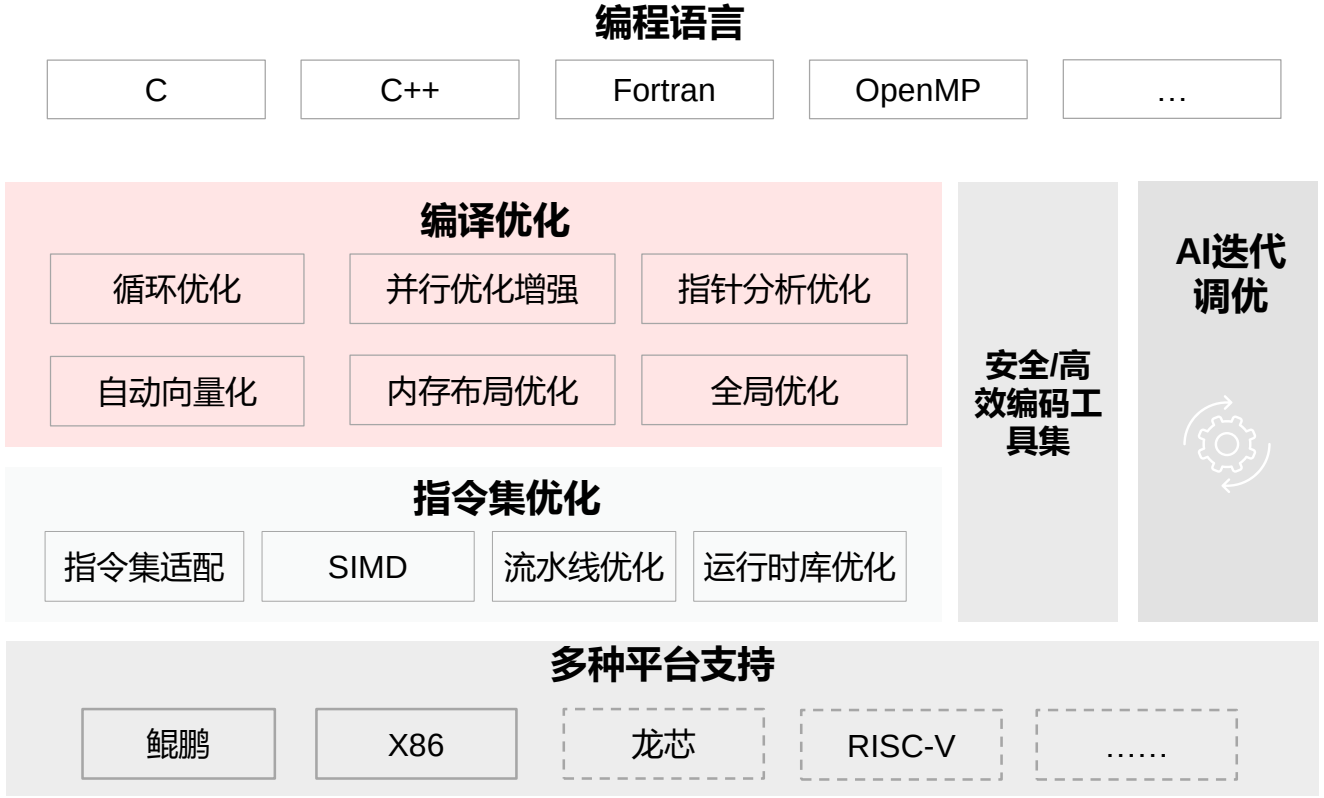

C源程序
预处理器
Preprocessor
(cpp)
.i
C编译器
Compiler
(cc)
汇编
.s
汇编器
Assembler
(as)
目标文件
.o
连接器
Linker
(ld)
库代码
Library routines
输入
机器代码
Machine code
输出

C/C++编译系统

- GCC：https://gcc.gnu.org/
  - 龙芯： http://www.loongnix.cn/index.php/GCC
  - 神威：http://www.nsccwx.cn/
- Clang/LLVM：https://llvm.org/
  - 毕昇编译器：针对鲲鹏平台的高性能编译器
    - 基于LLVM开发
    - 对中端及后端的关键技术点进行了深度优化
    - 集成Auto-tuner特性支持编译器自动调优
  - 龙芯： http://www.loongnix.cn/index.php/LLVM
- 闭源：ICC (oneAPI)

毕昇编译器的架构和关键特性

- 毕昇编译器是**华为编译器实验室**针对通用处理器架构构建
- 支持C/C++/Fortran编程语言及对应的OpenMP扩展
- 除LLVM已有功能外，对中、后端进行深度优化，并集成Auto-tuner
- 致力于打造高性能、高可信及易扩展的编译器工具链



关键特性

- 支持鲲鹏920(-march=tsv110)、X86-64等
- 支持C11、C++14/17语言标准，支持OpenMP 5.0 API标准
- 支持Fortran 2008语言标准，支持OpenMP 4.5 API标准
- **优化**：代码体积优化选项、多样化的浮点精度选项/模式调优
- **安全/高效编码**：静态检查、重构
- **AI迭代调优**：自动优化编译配置，迭代提升程序性能，完成最优编译
- 提供针对通用处理器架构的各类高性能编译优化技术

毕昇编译器信息获取

网页（获取软件包）

https://www.hikunpeng.com/zh/developer/devkit/compiler/bisheng

论坛

https://www.hikunpeng.com/forum/forum-0105101360563095011-1.html

在线课程

https://education.huaweicloud.com/courses/course-v1:HuaweiX+CBUCNXK068+Self-paced/about

沙箱实验

https://www.hikunpeng.com/learn/experiments

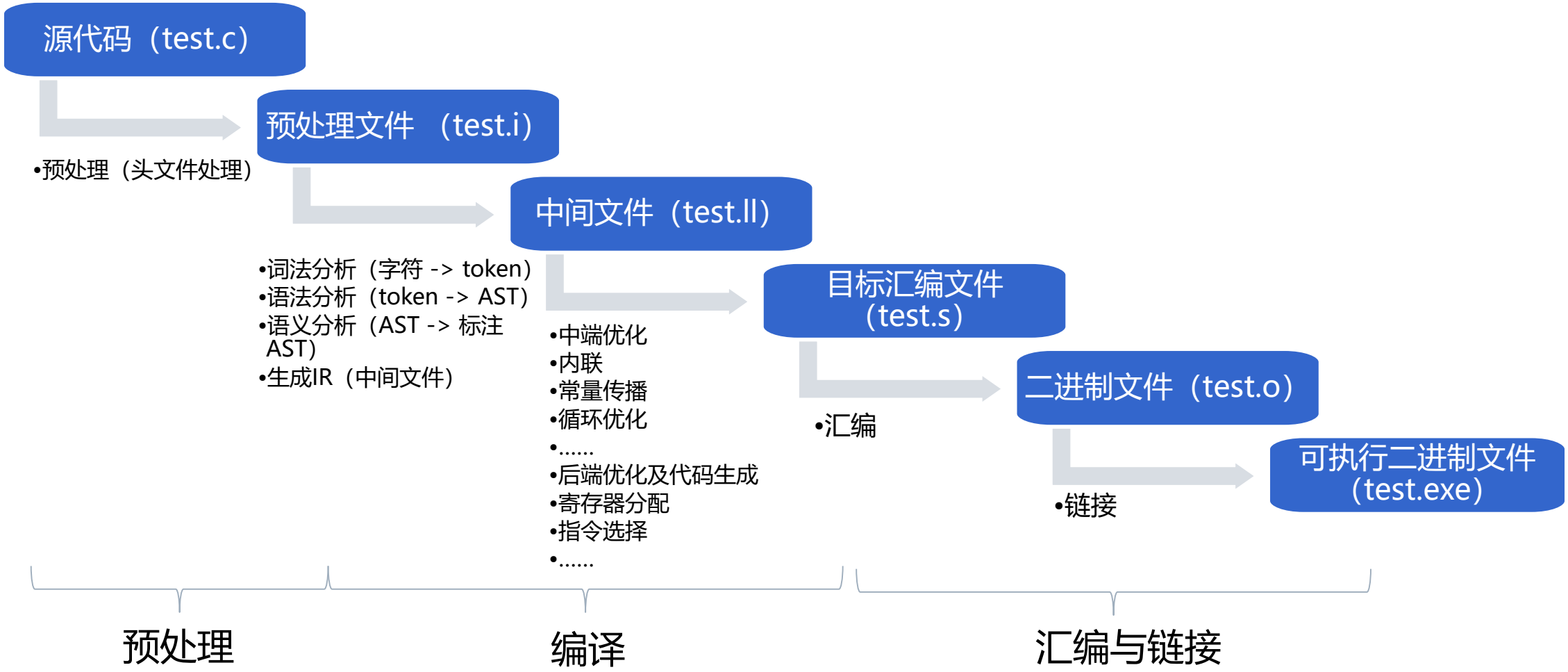
文档-毕昇编译器用户指南

https://www.hikunpeng.com/document/detail/zh/kunpengdevps/compiler/ug-bisheng

文档-AutoTuner特性指南

https://www.hikunpeng.com/document/detail/zh/kunpengdevps/compiler/fg-autotuner

编译的过程与架构划分


源代码（test.c）
•预处理（头文件处理）
预处理文件 （test.i）
•词法分析（字符 -> token）
•语法分析（token -> AST）
•语义分析（AST -> 标注 AST）
•生成IR（中间文件）
中间文件（test.ll）
•中端优化
•内联
•常量传播
•循环优化
•……
•后端优化及代码生成
•寄存器分配
•指令选择
•……
目标汇编文件 (test.s)
•汇编
二进制文件（test.o）
•链接
可执行二进制文件 (test.exe)
预处理
编译
汇编与链接

宏

https://en.wikipedia.org/wiki/Macro_(computer_science)

- **词法级别的宏：**如C语言
- **语法级别的宏：**如类Lisp的语言、Rust、Scala 等
- **Hygienic Macro** [ESOP2008]

  若宏调用处有个名字**name1**，同时宏内部也有一个**name1**，**hygienic**宏在展开时会把自己内部的**name1**改名成**name2**

  （注：普通宏则不改名）

- **Macrofication** [ESOP2016]

  识别（JavaScript）代码中与宏定义的模式相匹配的代码片段，将其替换为宏

Rust 具有卫生性**Hygiene**

```
macro_rules! foo {
    () => {
        let x = 1;
    }
}

let x = 10;
foo!();
println!("{}", x); // 10
```

宏内部定义的x不会污染调用者作用域

宏

```
#define M 10

#define SWAP(a, b)  int t; t=a; a=b; b=t;

#define SWAP(a,b)  {int t; t=a; a=b; b=t;}
```

会有问题吗

宏有哪些用处?

宏定义时需要注意什么?

宏应用举例

C语言宏举例：wrapfuns.h 配置文件

```
enum {
#define fnxx(a, b, c, d) CALL_##b = d,
#include "wrapfuns.h"
        CALLLAST
};
#define CALL_ID(k)      CALL_##k
#define CALL_NAME(k)    #k
…
#define COUNT_CALLS(fn) \
{           \
    (s_tstat->ncalls[CALL_ID(fn)]) += 1;
            \
}
```

用宏定义来描述函数对应的符号常量

配置文件：描述什么样的函数需要统计调用次数、时间等

wrapfuns.h

```
fnxx(BASIC, spmctime, spmctime, 1)
……
```

宏应用举例

C语言宏举例：wrapfuns.h 配置文件

```
#define BEGIN_TIMING(fn)\
{                              \
   if ( CALL_ID(fn) > 0 ) {  \
      COUNT_CALLS(fn)      \
      s_tstat->tstart[CALL_ID(fn)] = sys_time();\
   }     \
}
#define END_TIMING(fn)         \
{         \
   if ( CALL_ID(fn) > 0 ) {       \
      double t = sys_time()-     \
         (s_tstat->tstart[CALL_ID(fn)]);          \
      s_tstat->calltime[CALL_ID(fn)] += t;      \
   }              \
}
```

宏应用举例

C语言宏举例：BEGIN_TIMING等的应用

```
#define _SPMC_DECL1(t1, a1)          t1 a1
#define _SPMC_DECL2(t2, a2, ...)     t2 a2, _SPMC_DECL1(__VA_ARGS__)
#define _SPMC_DECL3(t3, a3, ...)     t3 a3, _SPMC_DECL2(__VA_ARGS__)

#define _SPMC_ADECL1(t1, a1)         a1
#define _SPMC_ADECL2(t2, a2, ...)    a2, _SPMC_ADECL1(__VA_ARGS__)
#define _SPMC_ADECL3(t3, a3, ...)    a3, _SPMC_ADECL2(__VA_ARGS__)

#define SPMC_FUNDEF_RET(x, rettype, fn, ...)                    \
        rettype                                                 \
        swr_##fn(_SPMC_DECL##x(__VA_ARGS__)) {                  \
                BEGIN_TIMING(fn);                               \
                rettype r;                                      \
                r = fn(_SPMC_ADECL##x(__VA_ARGS__));            \
                END_TIMING(fn);                                 \
                return r;                                       \
        }
```

参数声明

实参

用宏定义一大类函数，来对函数调用计时

汇编器

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

第一遍扫描**建立符号表**，包括代码标号.L2、.L11

第二遍扫描依据符号表中的信息来**产生可重定位代码**

jne .L6

gas
llvm-as

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（建立标号的回填链）

连接器（链接器Linker）

目标文件形式

- 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

常见连接器

GNU ld
Gold
lld (LLVM)

发展趋势

- 更快（并行）
- 更智能（LTO, Link-Time Optimization）
- 更可组合（插件）

连接

收集、组织程序所需的不同代码和数据，以便程序能被装入内存并被执行

解析符号引用

连接器（链接器Linker）

- 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
- 符号解析（Symbol Resolution）

识别各个目标模块中定义和引用的符号，为每个符号引用确定它所关联的一个同名符号的定义

- 重定位模块M中可能定义和引用的符号(用nm命令获取）

  全局符号(M中定义，可被外部引用)

  局部符号(M中定义，只能在M中引用)

  外部符号((M外定义，在M中引用)

在C语言中怎么对应?

连接器

- 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
- 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

- 重定位模块M中可能定义和引用的符号(用nm命令获取）
  全局符号(M中定义，可被外部引用)、
  局部符号(M中定义，只能在M中引用)、
  外部符号((M外定义，在M中引用)

如static全局变量
注意：不是局部变量

使用库的问题举例

- gcc -S hello.c

  没有任何报错和警告

- gcc -nostdinc -S hello.c

```
#include<stdio.h>
int main()
{
        printf("hello world");
        return 0;
}
```

```
hello.c:1:18: error: no include path in which to search for stdio.h
 #include<stdio.h>
                  ^
hello.c: In function ‘main’:
hello.c:4:2: warning: implicit declaration of function ‘printf’ [-Wimplicit-fu
nction-declaration]
  printf("hello world");
  ^~~~~~
hello.c:4:2: warning: incompatible implicit declaration of built-in function
printf’
hello.c:4:2: note: include ‘<stdio.h>’ or provide a declaration of ‘printf’
```

编译的什么阶段检查出错误？ **预处理阶段：cpp**

编译的什么阶段检查出警告？ **语义检查（静态类型检查）：cc**

使用库的问题举例

- gcc -S hello.c
- gcc -nostdinc -S hello.c
- gcc -nostdlib hello.c

```
#include<stdio.h>
int main()
{
        printf("hello world");
        return 0;
}
```

```
/usr/bin/ld: 警告: 无法找到项目符号 _start; 缺省为 0000000000400144
/tmp/ccvNQfYa.o：在函数‘main’中：
hello.c:(.text+0xf)：对‘printf’未定义的引用
collect2: error: ld returned 1 exit status
```

编译的什么阶段检查出警告和错误？

连接阶段：ld

_start是什么？为什么无法找到它？

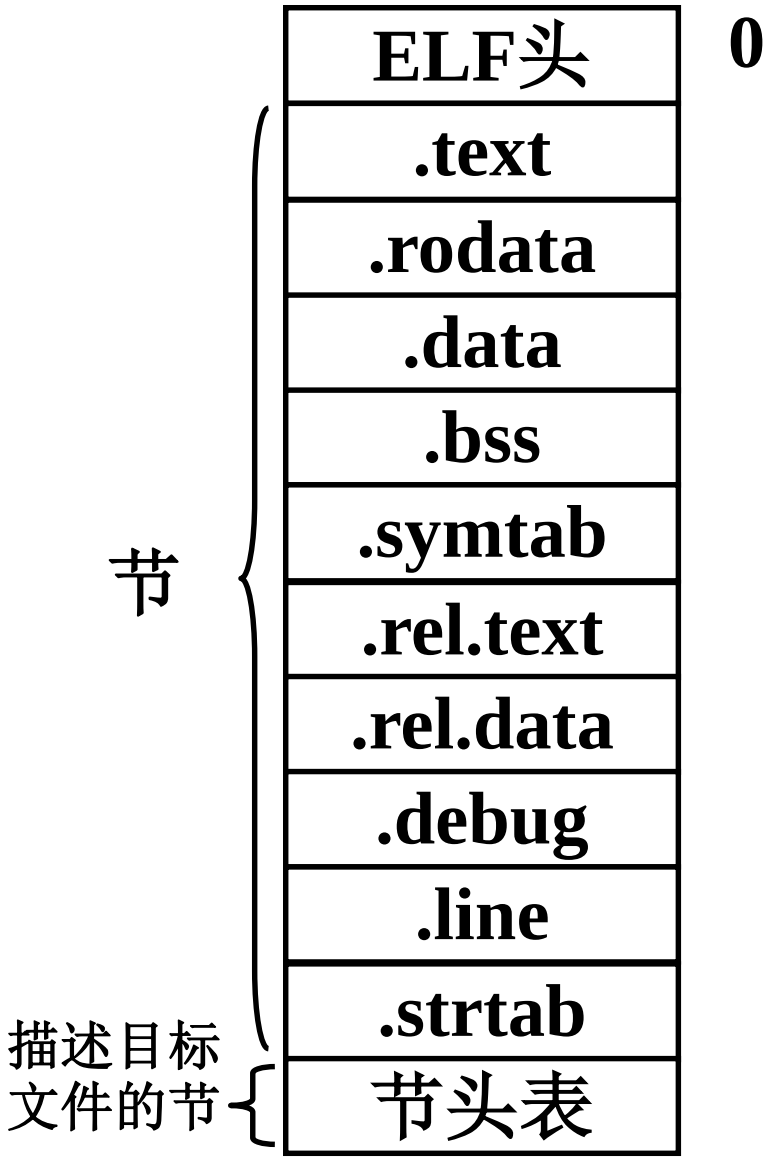
_start是程序的入口，在crt1.o定义（如位于/usr/lib/x86_64-linux-gnu）

printf为什么是未定义的引用？

因为用nostdlib，不链接C标准库

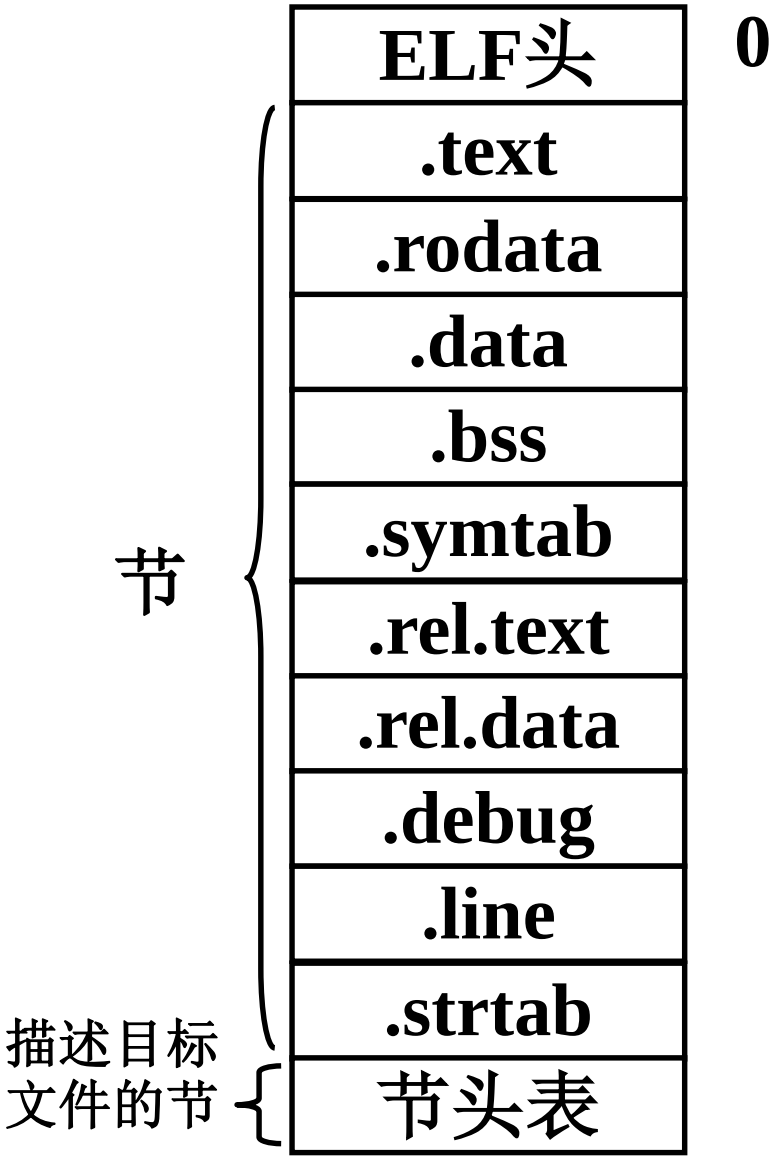
目标文件的格式

- Unix：ELF
  - ELF头
    - 描述了字的大小
    - 字节次序
    - 目标文件的类型
    - 机器类型
    - 节头表的位置及条目数
    - 其它



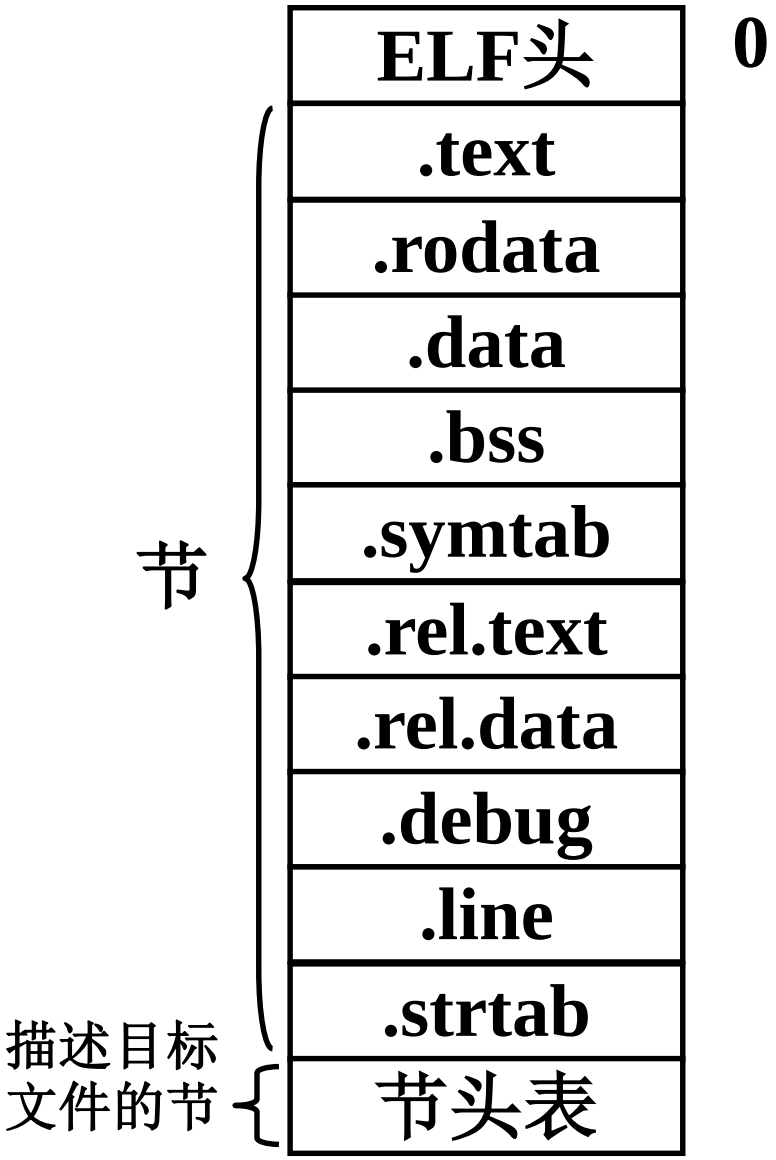
目标文件的格式

- Unix：ELF
  - 节头表
    - 描述各节的位置和大小
    - 位于目标文件的末尾
  - .text节
    - 被编译程序的机器代码
  - .rodata节
    - 只读数据
  - .data节
    - 已初始化的全局变量



目标文件的格式

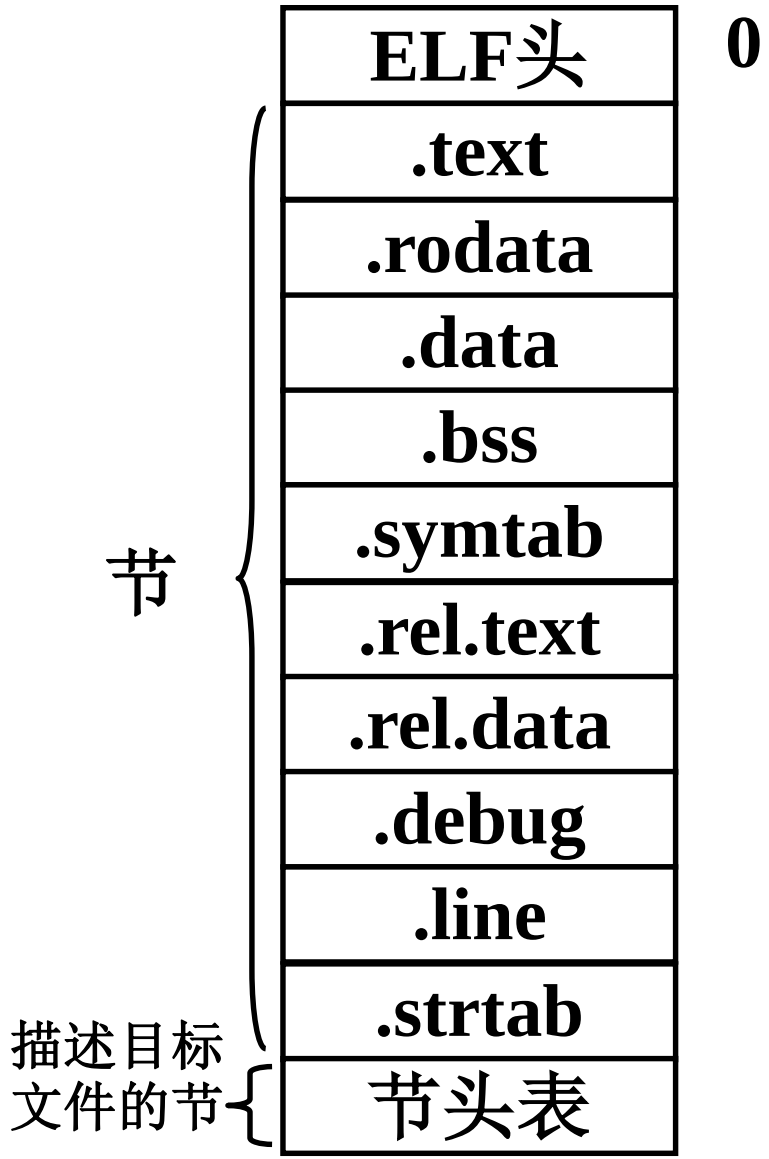
- **Unix：ELF**
  - **.bss节（.comm 节）**
    - 未初始化的全局变量
    - 位于目标文件的末尾
  - **.symtab节**
    - 在该模块中定义和引用的函数和全局变量的信息的符号表
    - **Type:** FUNC, OBJECT
    - **Bind**: GLOBAL, LOCAL, EXTERN
    - **Value**：地址
    - **Size**: 字节数
    - **Name**



目标文件的格式

- Unix：ELF
  - .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
    - 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



符号解析

解析规则

- 强符号：函数、已初始化的全局变量
- 弱符号：未初始化的全局变量、__attribute__((weak))修饰的函数
  - `__attribute__((weak)) int hook();`

同名符号的解析

- 不允许有同名的多重强符号定义
- 当出现同名的一个强符号定义和多个弱符号定义时，选择强符号的定义
- 出现同名的多个弱符号定义时，选择任意一个弱符号的定义

静态库

静态库

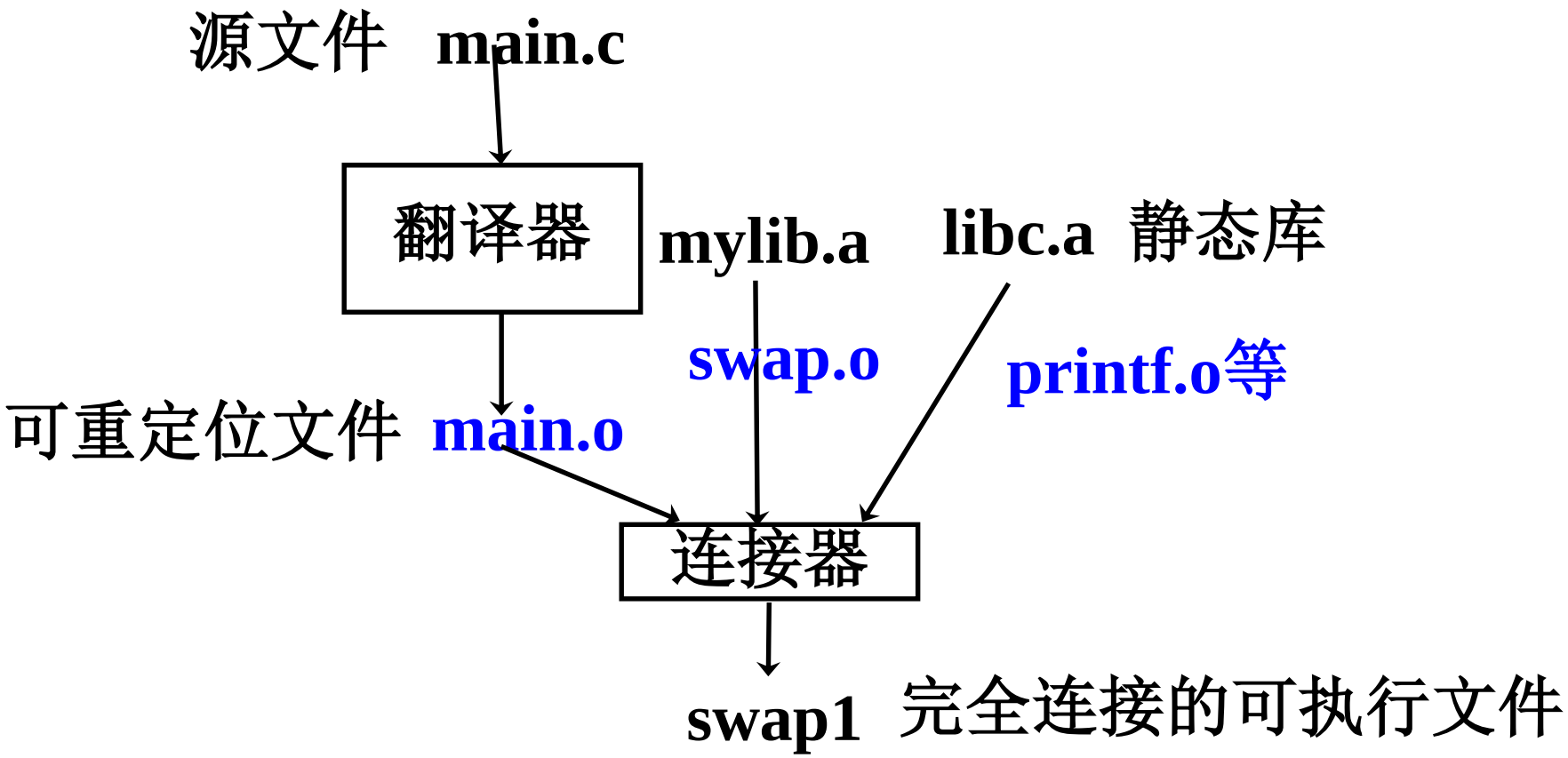
- 将相关的可重定位目标模块打包成一个文件，作为连接器的输入
- 连接器仅复制库中**被应用程序引用的模块**

gcc –c swap.c ——编译

ar rcs mylib.a swap.o ——建库

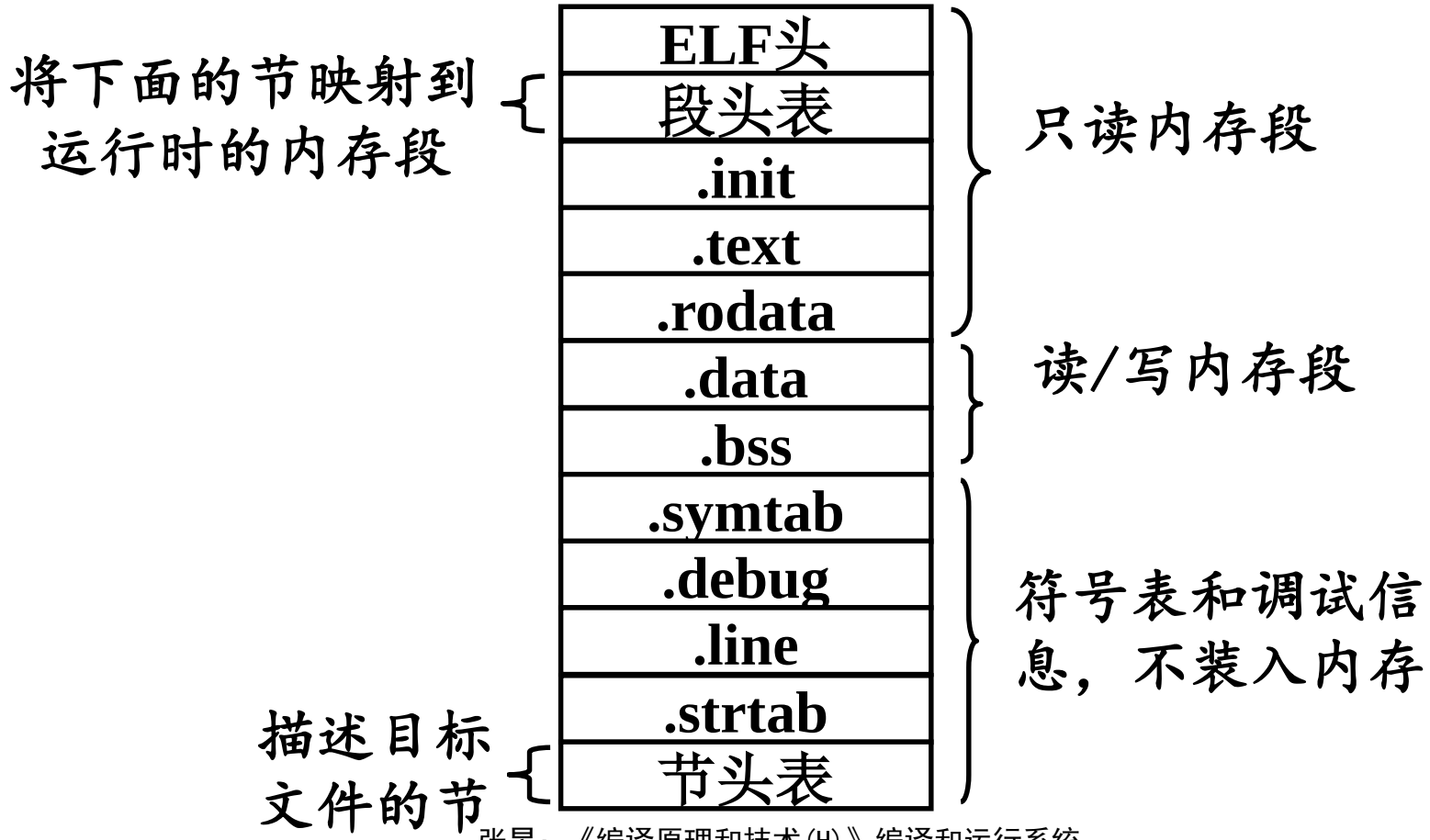
gcc –static –o swap1 main.c /usr/lib/libc.a mylib.a ——生成可执行文件

和静态库连接


源文件 main.c
翻译器
mylib.a
libc.a 静态库
swap.o
printf.o等
可重定位文件 main.o
连接器
swap1 完全连接的可执行文件

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



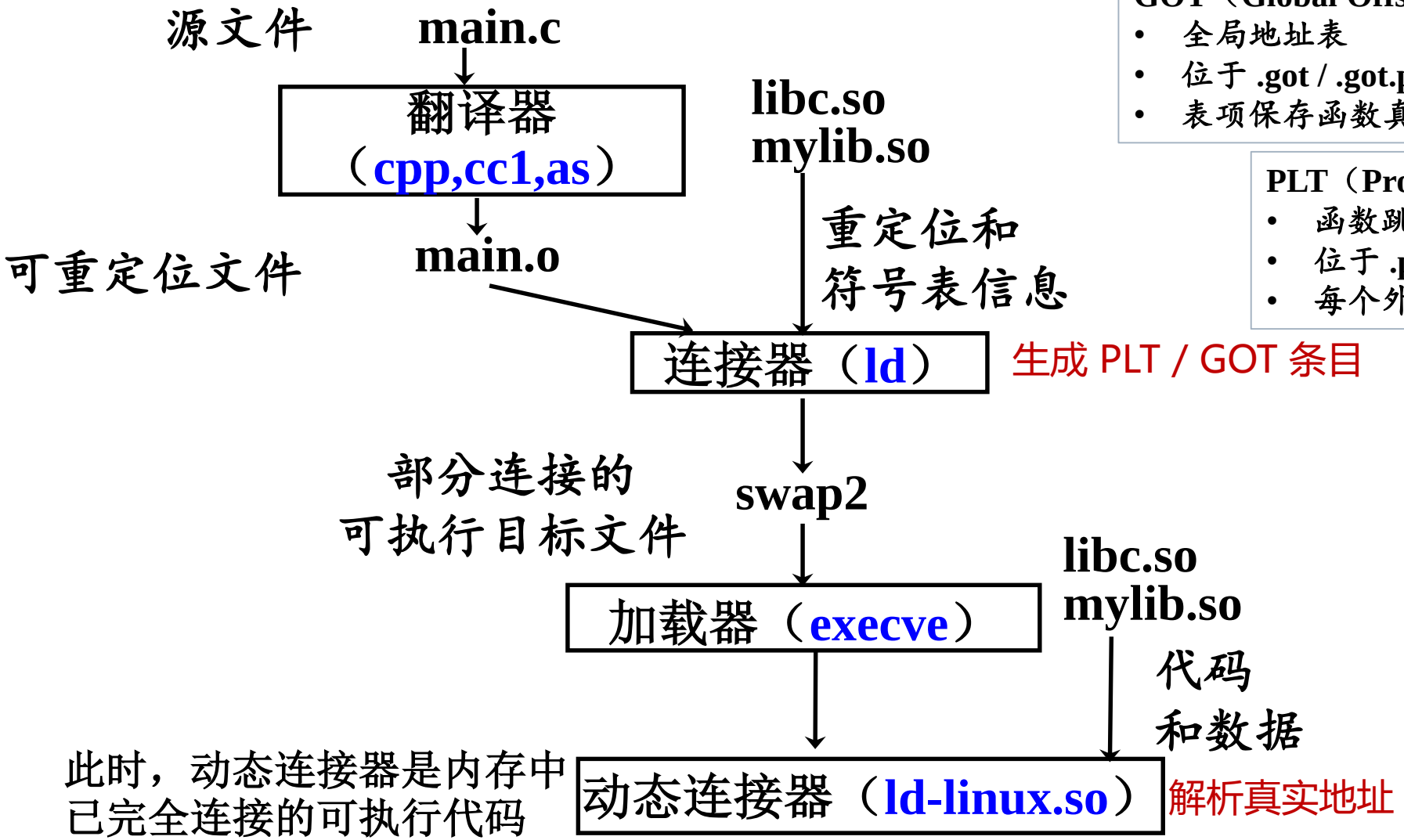
动态连接

- 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

  - 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

动态连接



GOT（Global Offset Table）
- 全局地址表
- 位于 .got / .got.plt 段
- 表项保存函数真实地址或全局变量地址

PLT（Procedure Linkage Table）
- 函数跳转表
- 位于 .plt 段
- 每个外部函数一个PLT条目

第一次调用 foo()：
1.call foo@plt
2.PLT 发现 GOT[foo]未绑定
3.跳转到 plt0
4.动态连接器：
- 查找 foo
- 得到真实地址
- **写回 GOT[foo]**

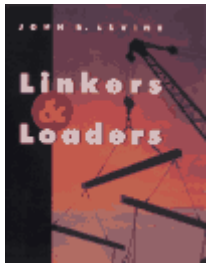
5.跳转到 foo 实现

后续调用：
不再进入动态连接器

相关资料

经典书籍

- *Linkers & Loaders* by John R. Levine, 2000



最近的研究

LTO（Link time optimization）

- Guided linking: dynamic linking without the costs, OOPSLA 2020, UIUC Vikram S. Adve, ALLVM 项目
  - 对动态链接行为预先已知的，在满足一定约束下将函数内联到其他库中的调用者(加速)
  - 将跨软件不同部分的相同函数去冗余（减少代码尺寸）

  https://github.com/allvm/allvm-tools
- BOLT: A Practical Binary Optimizer for Data Centers and Beyond, CGO 2018

  Meta (Facebook), github, bolt in llvm
  - 在不重新编译源码的情况下，对二进制进行性能优化——后连接(post-link)二进制优化器
  - 用“真实运行时 profile”，重排和优化最终二进制布局

毕昇编译器LTO优化

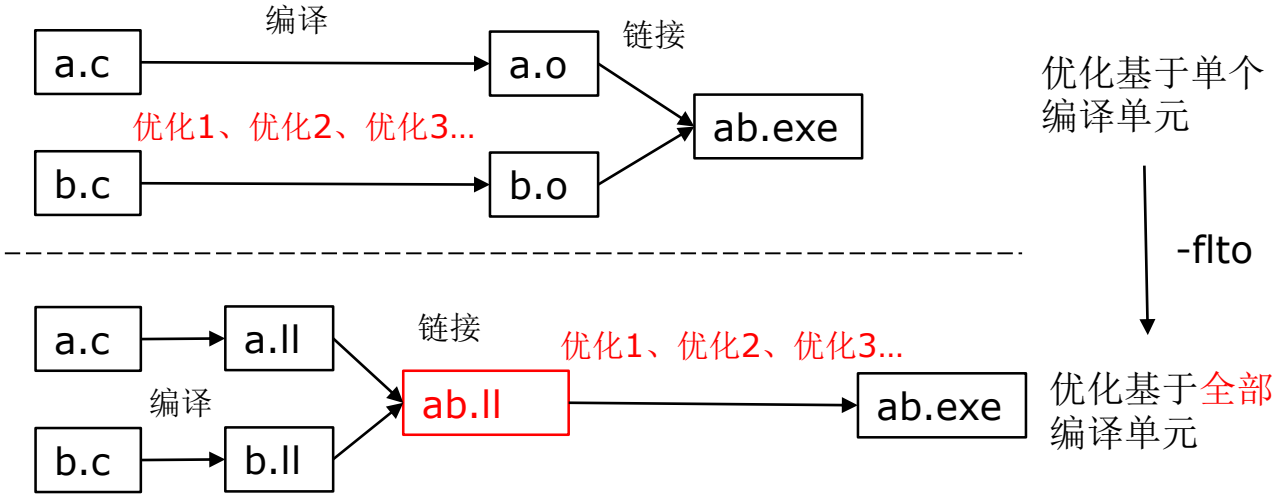
a.c

```
static int foo1(){
  return bar();
}

int foo2() {
  printf("foo2!");
}

int main(){
  return foo1();
}
```

b.c

```
static int g = 0;
int bar1() {
  g = 1;
}

int bar(){
  int t = 1;
  if (g > 0)
    t = g + t;
  else
    t = g – t;
  return t;
}
```

链接时优化 (LTO)原理：跨文件、过程间优化



ab.ll

```
int main(){
  return -1;
}
```

LTO效果

+ 跨文件函数内联，减小调用开销
+ 跨文件函数特化、常量传播，消除冗余代码
+ 可以跨语言优化
- 编译时间更长
- 需要修改Makefile

LTO是成熟的编译优化技术、业界已经广泛使用

- 谷歌Android kernel使用LTO编译
- Linux kernel 5.12版本引入LTO支持
- SPECCPU, Gauss数据库等

处理目标文件的一些工具

ar 创建静态库，插入、删除、罗列和提取成员

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

例题1

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2

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

- 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3

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3

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4

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4

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

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

a的使用局部于文件，
应加static而未加

例题5

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5

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

和

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

buf: array(1, int)

buf: pointer(int)

类型不同

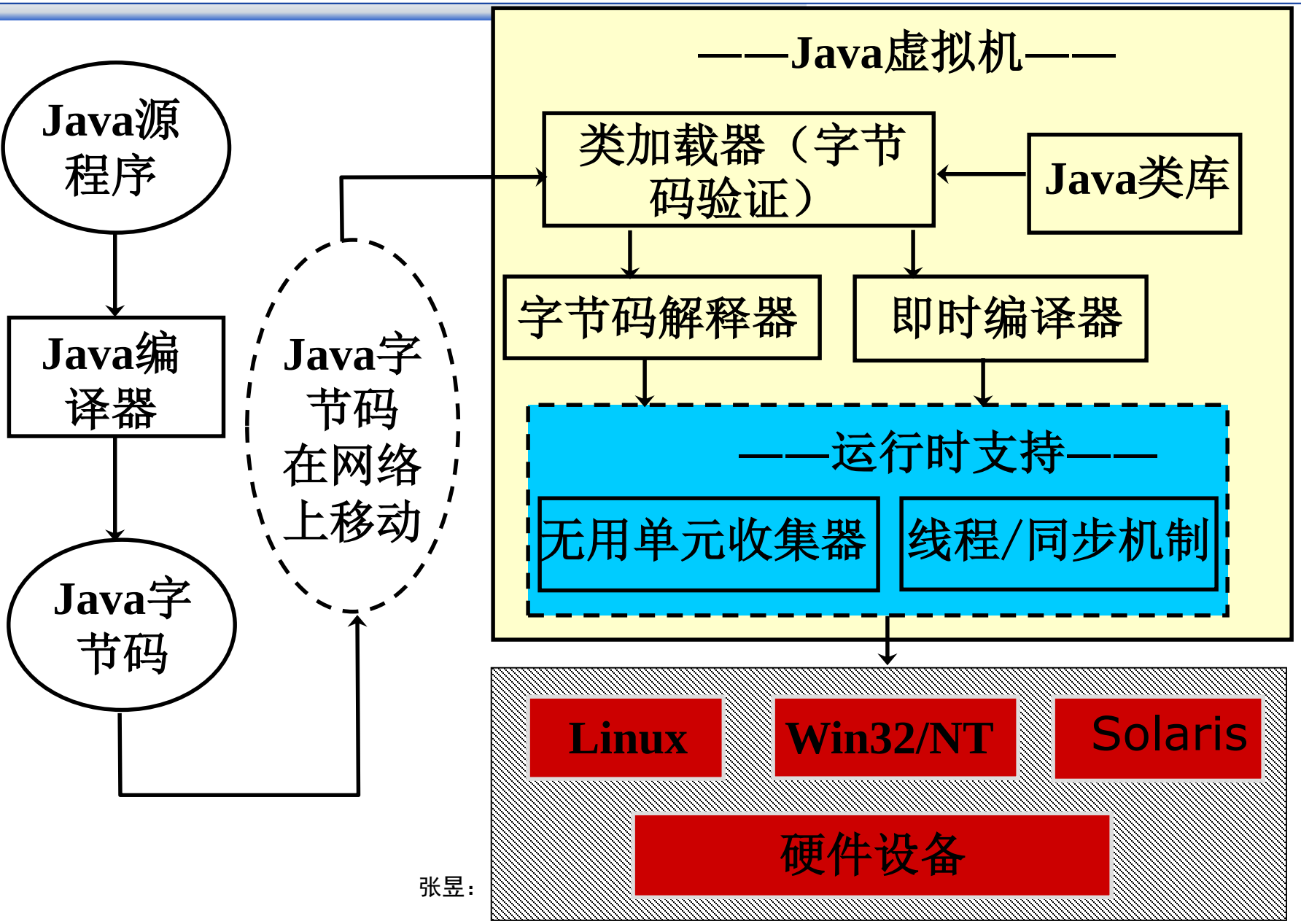
- 连接时不检查名字的类型
  - 但不同文件分别编译，每个文件对buf有不同的类型
- 连接时让不同文件中同一名字的地址相同
  - 运行时，在link2.c中，由于buf的内容是100，取*buf的值就是取地址为100的单元的内容。该地址不在程序数据区内，报错
- 若把这些代码放在同一文件中，编译时报错，Why?

中国科学技术大学
University of Science and Technology of China

2. Java运行时系统

- Java虚拟机
- 无用单元收集
- 即时编译器

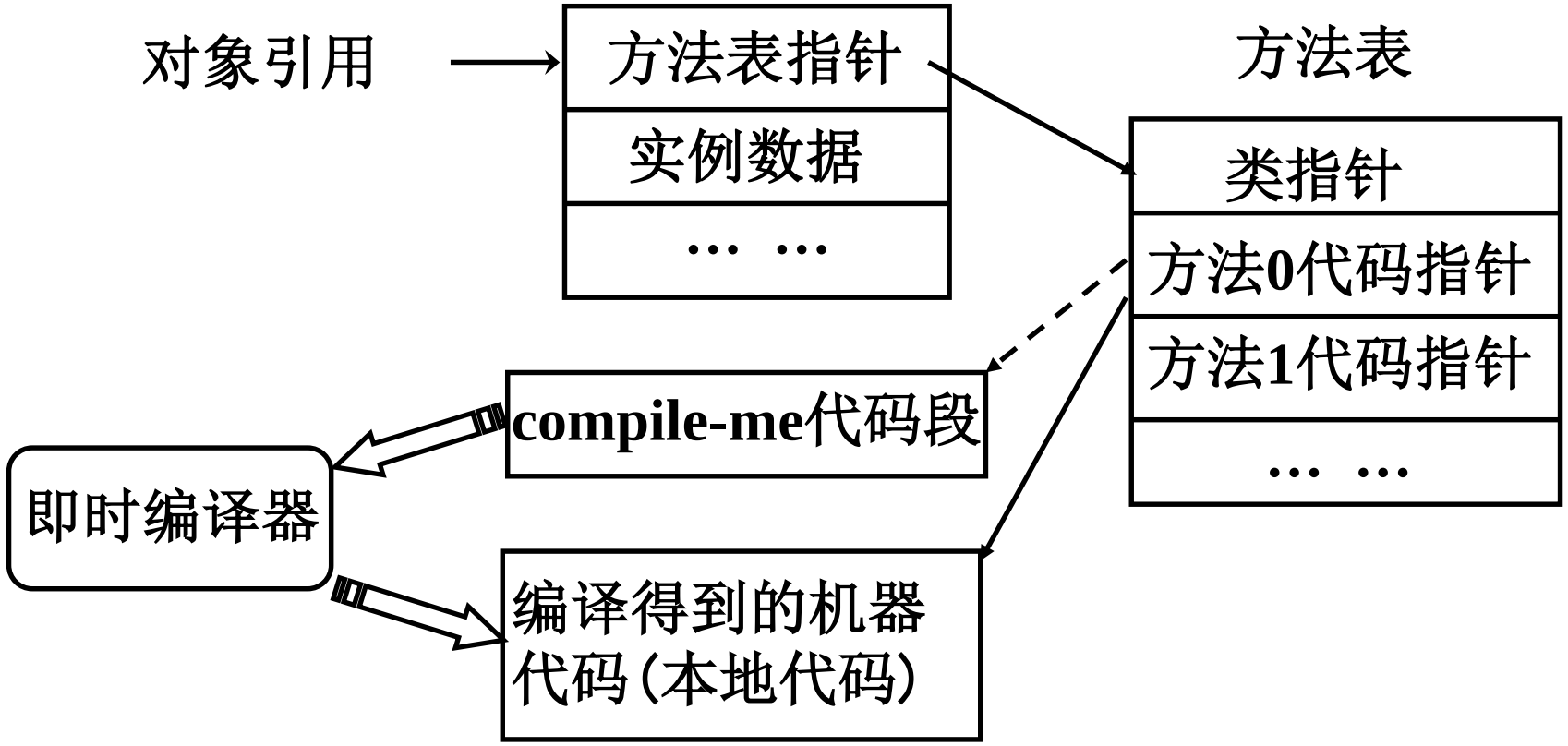
Java虚拟机（Java运行系统）



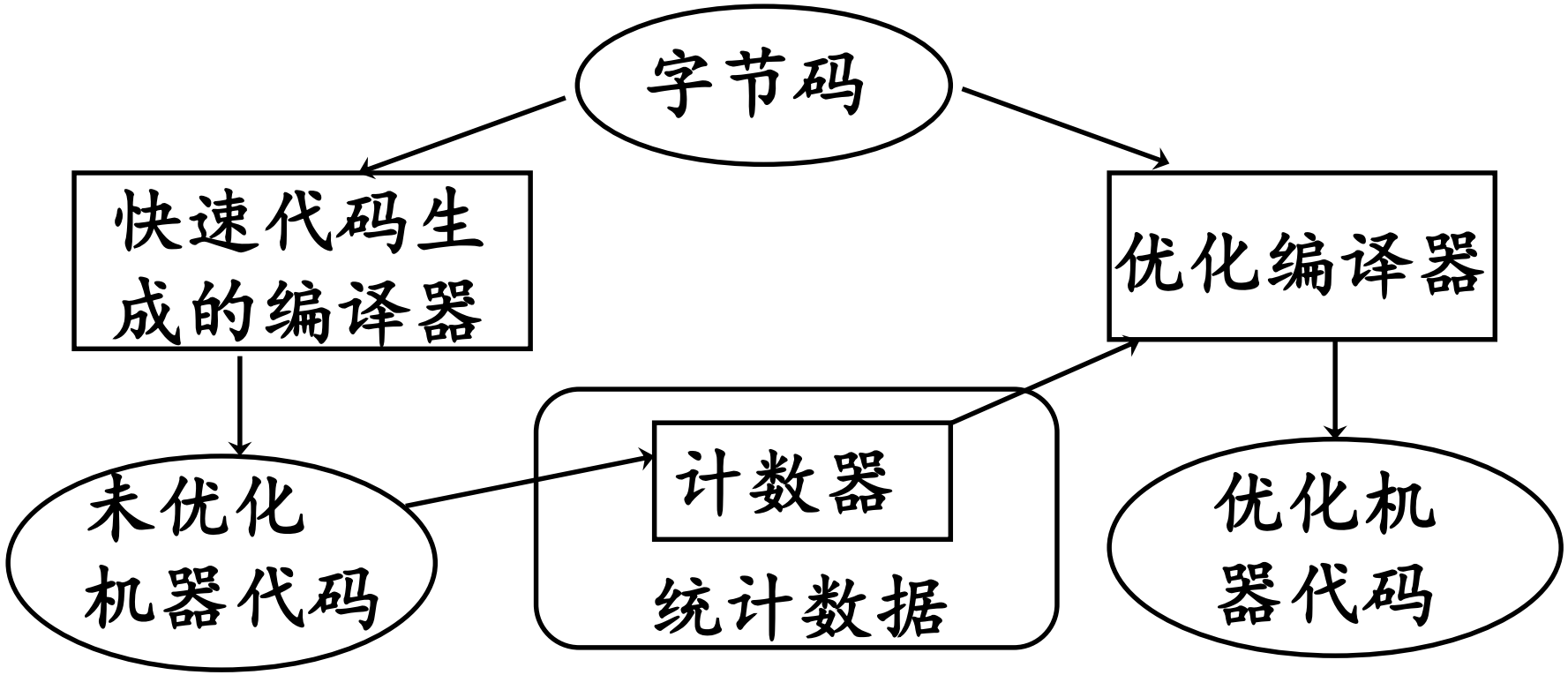
即时编译器

- 编译时机
  - 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
- 代码性能
  - 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
- 虚拟机的组织
  - 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

即时编译


对象引用
方法表指针
实例数据
… …
方法表
类指针
方法0代码指针
方法1代码指针
… …
compile-me代码段
即时编译器
编译得到的机器
代码(本地代码)

重编译机制



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

无用单元收集

- 标记和清扫
  - 标记堆上所有可达记录：从根集开始图遍历
  - 清扫从堆的首地址开始，寻找未被标记的记录，把它们链成一个空闲链表
- 引用计数
- 拷贝收集
- 分代收集