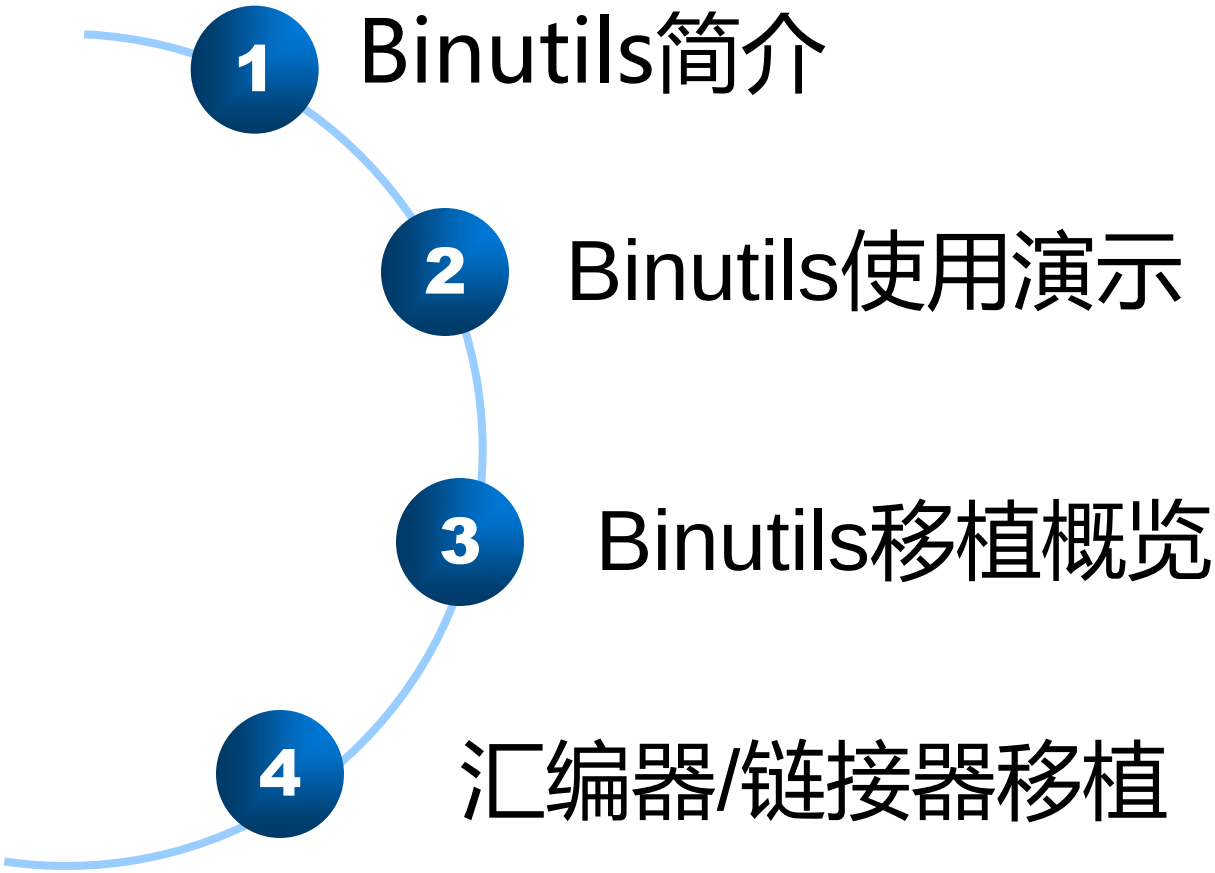
针对嵌入式CPU的Binutils移植

袁新宇

Yuan_xin_yu@hotmail.com

Yuanxinyu.hangzhou@gmail.com

目录



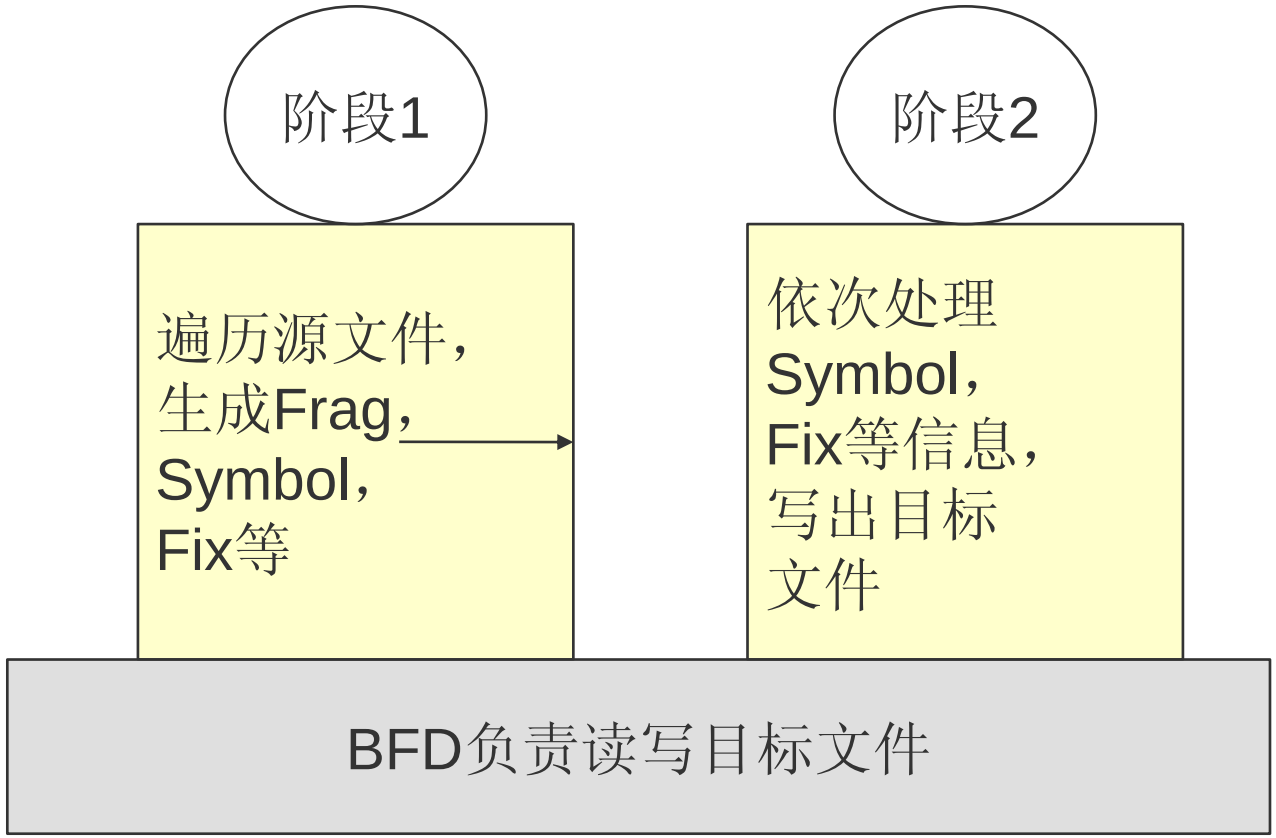
1.Binutils简介

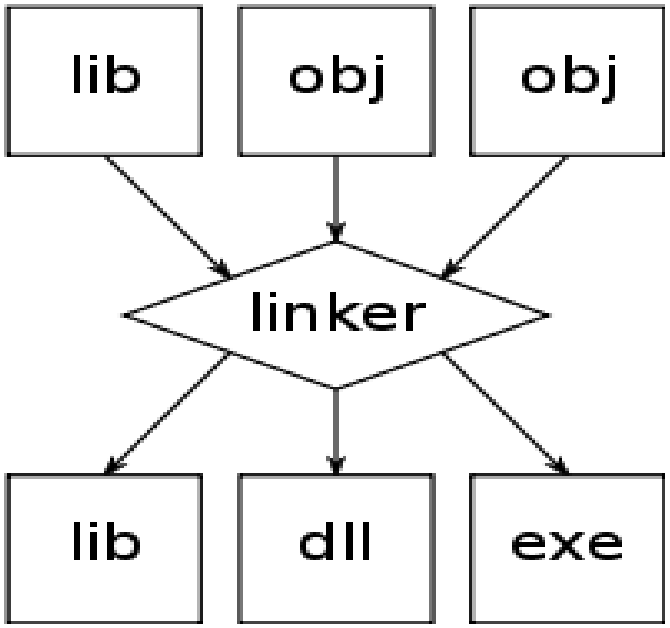
The GNU Binutils are a collection of binary tools. The main ones are:

- **ld** - the GNU linker.
- **as** - the GNU assembler.

- Most of these programs use **BFD**, the Binary File Descriptor library, to do low-level manipulation. Many of them also use the **opcodes** library to assemble and disassemble machine instructions.

GAS程序结构


阶段1
阶段2
遍历源文件，
生成Frag，
Symbol，
Fix等
依次处理
Symbol，
Fix等信息，
写出目标
文件
BFD负责读写目标文件



链接器的作用

链接器将多个目标文件链接起来，生成一个可执行文件。

链接器完成了符号解析、重定位处理等工作。

二进制目标文件格式

- a.out
  assembler and link editor output
  汇编器和链接编辑器的输出
- coff
  common object file format
  一种通用的对象文件格式
- ELF
  excutive linked file
  Linux系统所采用的一种通用文件格式，支持动态连接。
  ELF格式可以比COFF格式包含更多的调试信息

目标文件

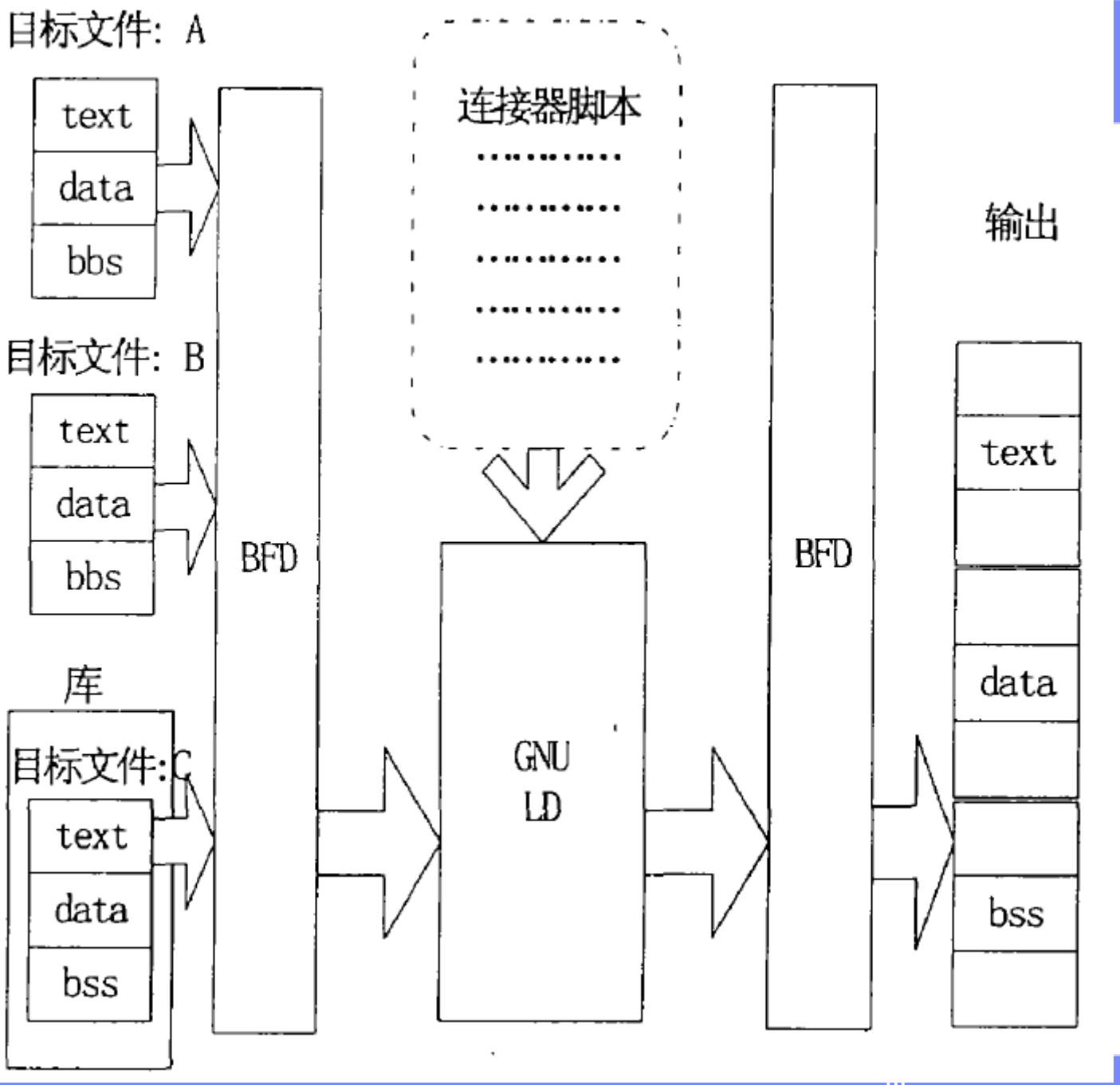
- ld通过BFD库可以读取和操作coff、elf、a.out等各种执行文件格式的目标文件
  - BFD（Binary File Descriptor）
- 目标文件（object file）
  - 由多个节(section)组成，常见的节有：
    - text节保存了可执行代码，
    - data节保存了有初值的全局标量，
    - bss节保存了无初值的全局变量。

Binutils-BFD

- BFD库的设计目标是针对不同体系结构、不同格式目标文件提供一种统一的操作界面，使用户能够使用相同的函数对不同格式的目标文件进行操作。

- BFD库，从层次上分为两部分:前端和后端。前端的功能是提供用户统一的界面，管理内存和各种内部数据结构，同时还决定使用哪一套后端以及何时使用该后端的函数。后端的主要功能是实现具体机器的目标文件表示和BFD内部表示的互相转换。

- BFD对目标文件进行抽象，在内部使用一种统一的格式来表示目标文件。


目标文件: A
text
data
bbs
目标文件: B
text
data
bbs
库
目标文件: C
text
data
bss
BFD
连接器脚本
GNU
LD
BFD
输出
text
data
bss

Binutils还包括：

- **addr2line** -把程序地址转换为文件名和行号.
- **ar** - 建立、修改、提取归档文件.
- **c++filt** - 解码(Demangle)C++符号名.
- **gold** - A new, faster, ELF only linker, still in beta test.
- **gprof** - 显示采样信息.
- **nm** - 列出目标文件中的符号.
- **objcopy** - 一种目标文件中的内容复制到另一种类型的目标文件中.
- **objdump** -显示一个或者更多目标文件的信息.
- **ranlib** - 产生归档文件索引.
- **readelf** - 显示ebf格式可执行文件的信息.
- **size** -列出目标文件每一段的大小以及总体的大小.
- **strings** -打印某个文件的可打印字符串，这些字符串最少4个字符长.
- **strip** -丢弃目标文件中的全部或者特定符号.

2.Binutils使用演示

binutils开发工具使用举例

- ar
- nm
- Objcopy
- Objdump
- readelf

ar

- ar用于建立、修改、提取归档文件(archive)，一个归档文件，是包含多个被包含文件的单个文件（也可以认为归档文件是一个库文件）。
- 被包含的原始文件的内容、权限、时间戳、所有者等属性都保存在归档文件中，并且在提取之后可以还原

使用ar建立库文件（1）

- 源程序add.c和minus.c

```
// add.c
int Add(int a, int b)
{
        int result;
        result = a+b;
        return result;
}
```

```
// minus.c
int Minus(int a, int b)
{
        int result;
        result = a-b;
        return result;
}
~
```

使用ar建立库文件（2）

```
[donger@donger gcctest]$ gcc -c add.c minus.c
[donger@donger gcctest]$ ar rv libtest.a add.o minus.o
ar: 正在创建 libtest.a
a - add.o
a - minus.o
[donger@donger gcctest]$ ls
add.c  gcctest.c  minus.c  mytest.c
add.o  libtest.a  minus.o
[donger@donger gcctest]$ cp libtest.a /usr/lib
cp: 无法创建一般文件‘/usr/lib/libtest.a’:  权限不够
[donger@donger gcctest]$ su
Password:
[root@donger gcctest]# cp libtest.a /usr/lib
[root@donger gcctest]#
```

编译成目标文件

建立库文件

Ar的rv参数的说明：

r：将多个文件组成一个文件

v：输出信息

将库文件拷贝到/usr/lib目录下

库文件使用举例
在代码中使用Add和Minus函数

```
// test.c
#include <stdio.h>

int main()
{
        int a = 8;
        int b = 3;
        printf("a=%d\tb=%d\n",a,b);
        int sum = Add(a,b);
        printf("a+b=%d\n",sum);
        int cha = Minus(a,b);
        printf("a-b=%d\n",cha);
        return 0;
}
```

在编译时指定库文件

```
[root@donger gcctest]# ls
add.c  gcctest.c  minus.c  mytest.c  test.c
[root@donger gcctest]# gcc -o test test.c -ltest
[root@donger gcctest]# ./test
a=8     b=3
a+b=11
a-b=5
[root@donger gcctest]#
```

指明将libtest.a链接进来

运行结果

nm

- nm的主要功能是列出目标文件中的符号，这样程序员就可以定位和分析执行程序和目标文件中的符号信息和它的属性

nm显示的符号类型

A：符号的值是绝对值，并且不会被将来的链接所改变

B：符号位于未初始化数据部分（BSS段）

C：符号是公共的。公共符号是未初始化的数据。在链接时，多个公共符号可能以相同的名字出现。如果符号在其他地方被定义，则该文件中的这个符号会被当作引用来处理

D：符号位于已初始化的数据部分

T：符号位于代码部分

U：符号未被定义

？：符号类型未知，或者目标文件格式特殊

nm使用举例

```
[root@donger gcctest]# ls
add.c  gcctest.c  minus.o  test     test.o
add.o  minus.c    mytest.c  test.c
[root@donger gcctest]# nm test.o
         U Add
00000000 T main
         U Minus
         U printf
[root@donger gcctest]# nm add.o
00000000 T Add
[root@donger gcctest]# nm minus.o
00000000 T Minus
[root@donger gcctest]#
```

如果对test可执行文件使用nm，会有什么结果呢？

objcopy

- 可以将一种格式的目标文件内容进行转换，并输出为另一种格式的目标文件。它使用GNU BFD(binary format description)库读/写目标文件，通过这个BFD库，objcopy能以一种不同于源目标文件的格式生成新的目标文件
- $objcopy -h
- 在makefile里面用-O binary 选项来生成原始的二进制文件,即通常说的image文件

Objcopy使用举例

```
[root@donger gcctest]# ls
add.c  gcctest.c  minus.o   test     test.o
add.o  minus.c    mytest.c  test.c
[root@donger gcctest]# file test
test: ELF 32-bit LSB executable, Intel 80386, version 1 (SY
SV), for GNU/Linux 2.2.5, dynamically linked (uses shared l
ibs), not stripped
[root@donger gcctest]#
[root@donger gcctest]#
[root@donger gcctest]# objcopy -O srec test ts
[root@donger gcctest]# ls
add.c  gcctest.c  minus.o   test     test.o
add.o  minus.c    mytest.c  test.c   ts
[root@donger gcctest]# file ts
ts: Motorola S-Record; binary data in text format
[root@donger gcctest]#
```

使用file命令查看文件类型

生成srec格式的目标文件

使用file命令查看新文件的类型

objdump

- 显示一个或多个目标文件的信息，由其选项来控制显示哪些信息。一般来说，objdump只对那些要编写编译工具的程序员有帮助，但是我们通过这个工具可以方便的查看执行文件或者库文件的信息

Objdump使用举例（1）

```
[root@donger gcctest]# objdump -f test

test:     文件格式 elf32-i386
体系结构: i386, 标志 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
起始地址 0x080482d8

[root@donger gcctest]# objdump -f ts

ts:     文件格式 srec
体系结构: UNKNOWN!, 标志 0x00000000:

起始地址 0x080482d8

[root@donger gcctest]#
```

-f选项：显示文件头中的内容

Objdump使用举例（2）

```
[root@donger gcctest]# objdump -d add.o

add.o:     文件格式 elf32-i386

反汇编 .text 节:

00000000 <Add>:
   0:   55                      push   %ebp
   1:   89 e5                   mov    %esp,%ebp
   3:   83 ec 10                sub    $0x10,%esp
   6:   8b 45 0c                mov    0xc(%ebp),%eax
   9:   03 45 08                add    0x8(%ebp),%eax
   c:   89 45 fc                mov    %eax,0xfffffffc(%ebp)
   f:   8b 45 fc                mov    0xfffffffc(%ebp),%eax
  12:   c9                      leave
  13:   c3                      ret
[root@donger gcctest]#
```

-d选项进行反汇编

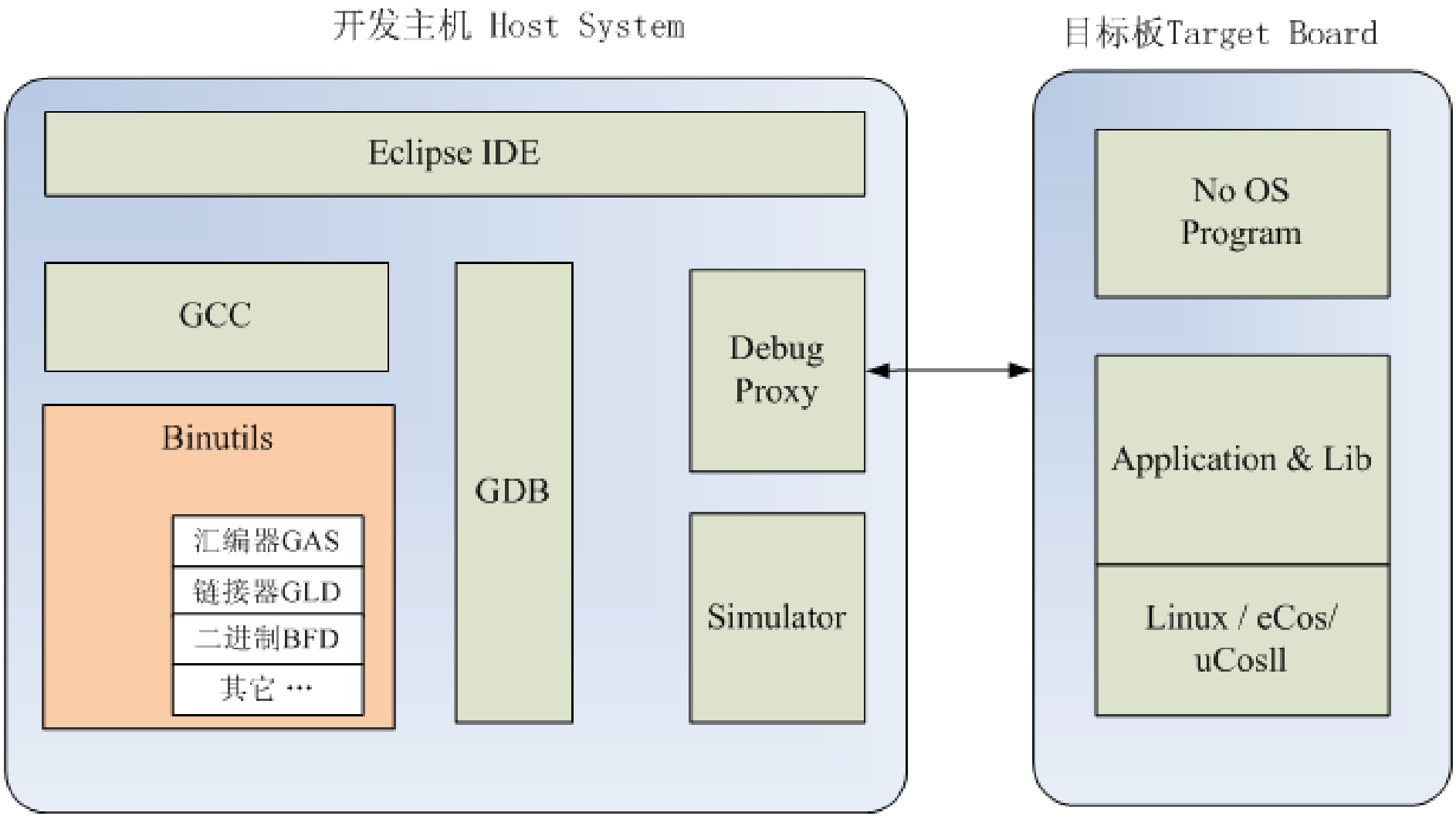
readelf

- readelf软件显示一个或多个ELF格式的目标文件信息。

Readelf使用举例

```
[root@donger gcctest]# readelf -h test
ELF 头:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (可执行文件)
  Machine:                           Intel 80386
  Version:                           0x1
  入口点地址:               0x80482d8
  程序头起点:               52 (bytes into file)
  Start of section headers:          2120 (bytes into file)
  标志:                     0x0
  本头的大小:               52 (字节)
  程序头大小:               32 (字节)
  程序头数量:               7
  节头大小:                 40 (字节)
  节头数量:                 28
  字符串表索引节头:         25
[root@donger gcctest]#
```

3.Binutils移植概览


开发主机 Host System
目标板Target Board
Eclipse IDE
GCC
Binutils
汇编器GAS
链接器GLD
二进制BFD
其它 …
GDB
Debug Proxy
Simulator
No OS Program
Application & Lib
Linux / eCos/ uCosll

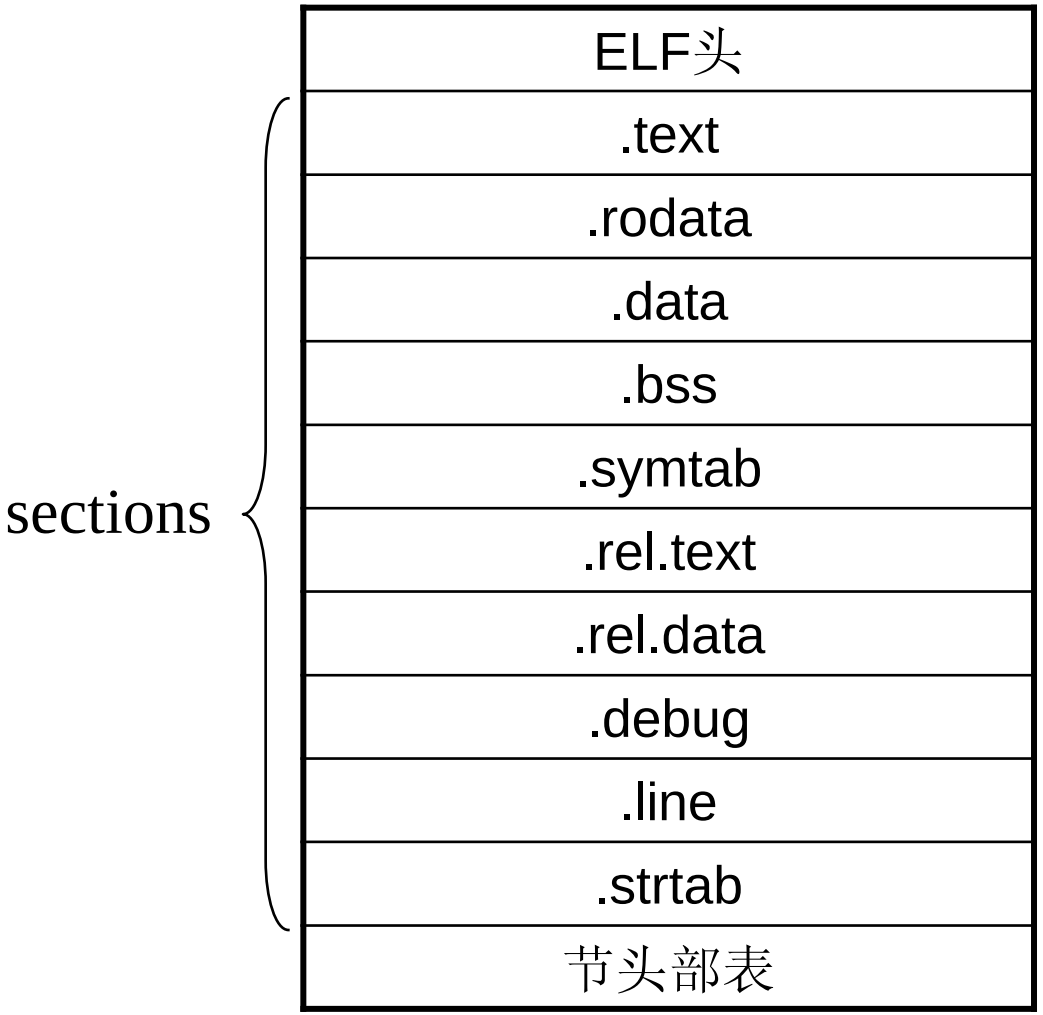
移植目标

- 针对嵌入式CSKY CPU的16位/32位混编指令集
- 只支持ELF目标文件格式
- 兼容原有16位指令集
- 高性能与高代码密度
- 当前只支持静态链接，以后扩展支持动态链接
- 采用DejaGun测试框架进行测试

ELF格式

- 根据用途的的不同，ELF文件可以分为三种类型:
- 1.可重定位文件:包含用十和其他的ELF文件一起连接生成可执行文件或共享目标文件的代码和数据。
- 2.可执行文件:用十被执行，它指出了如何在内存中生成程序的进程映象。
- 3.共享目标文件:可以作为库和其他的可重定位文件或共享目标文件一起连接生成新的ELF文件，也可以在动态连接时与可执行文件及其他共享目标文件一起生成程序的进程映象。

ELF可重定位目标文件

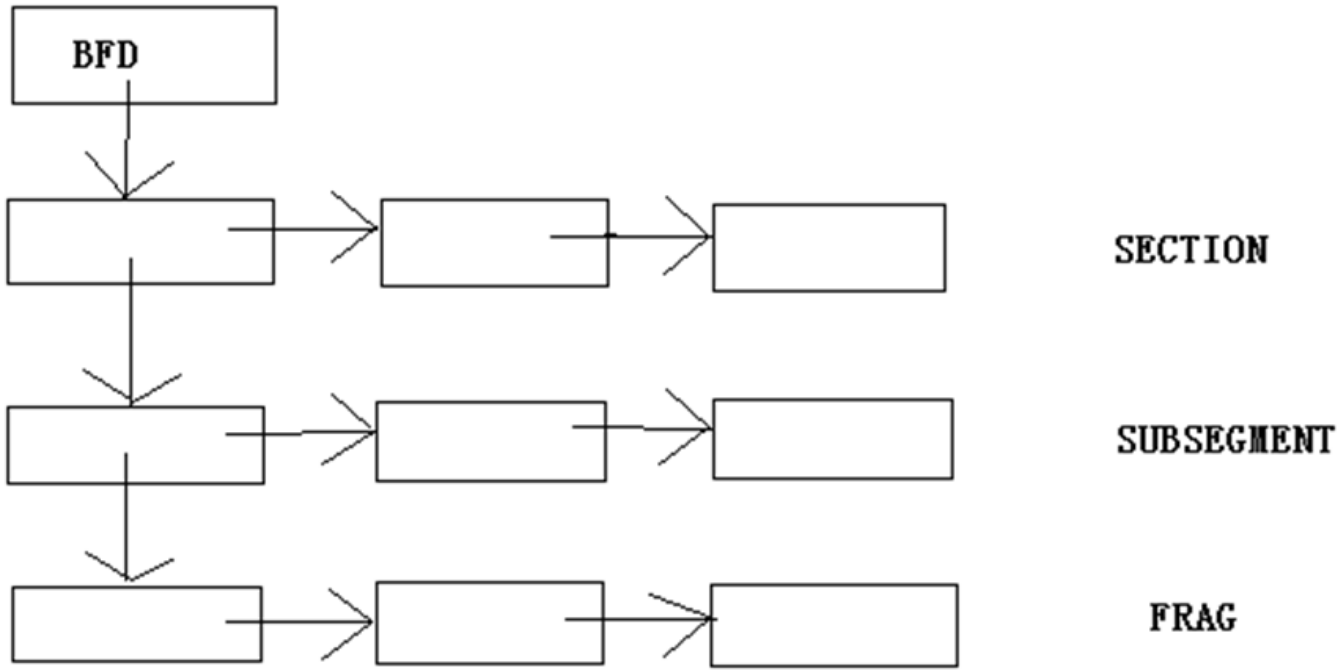


Binutils移植的任务

- 汇编器（GAS）
  - 正确解析新的指令
  - 为链接器、反汇编器等后端生成合适的信息
- 二进制文件操作函数库（BFD）
  - 实现读写目标文件的各种函数接口
  - 为链接器脚本中的函数提供实现
- 链接器（LD）
  - 选择或写出合适的链接控制脚本
- 其他二进制工具（OBJDUMP等）
  - 实现反汇编等必须功能

4.汇编器移植

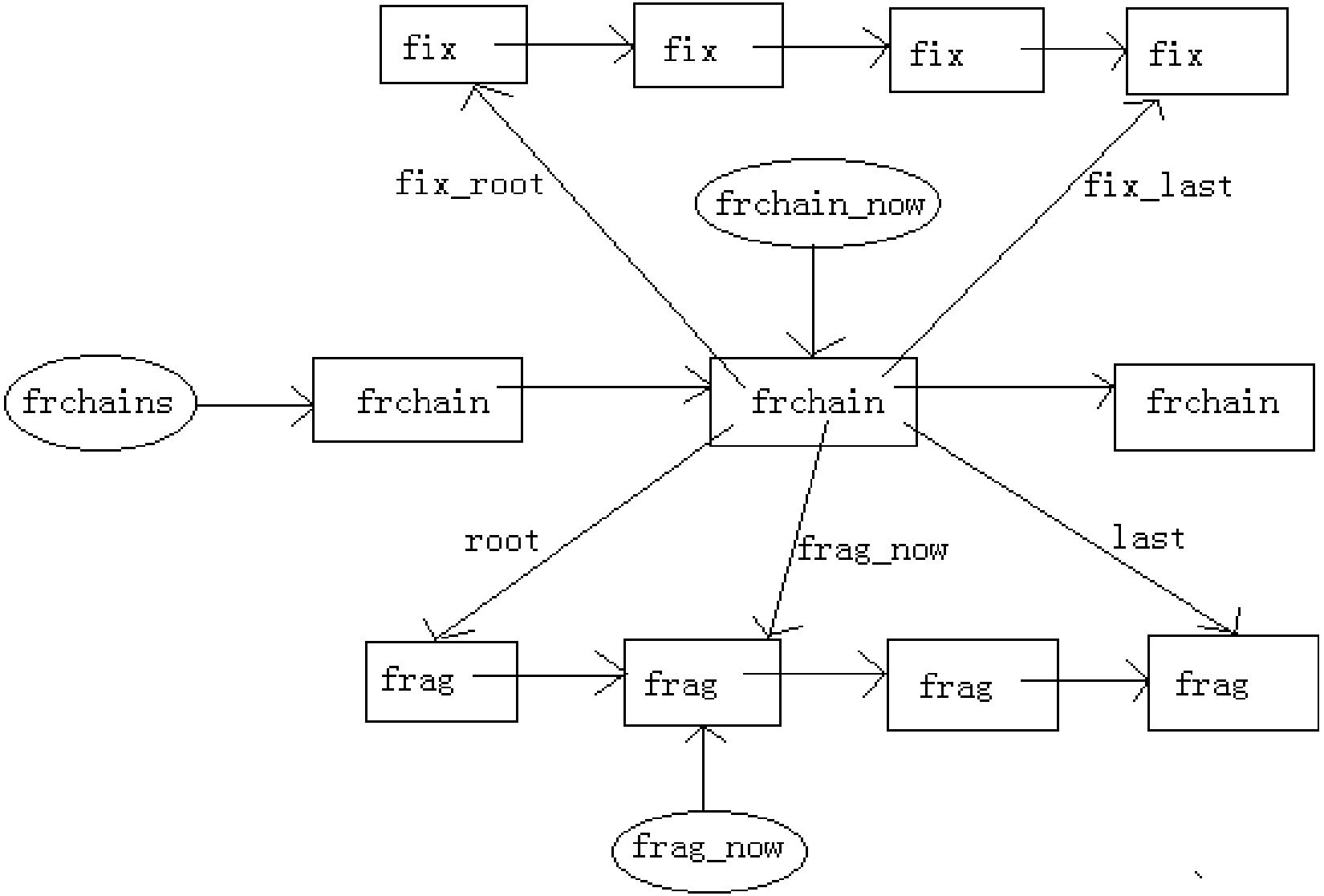
汇编器基本数据结构



Frag是GAS中基本的数据结构，它是基于汇编代码的最小结构，是最后构成目标输出文件的基本

Frag结构的字段包括：

- fr_address
- fr_fix
- fr_symbol
- fr_line
- fr_type
- …


fix
fix
fix
fix
fix_root
frchain_now
fix_last
frchains
frchain
frchain
frchain
root
frag_now
last
frag
frag
frag
frag
frag_now

汇编器扫描流程

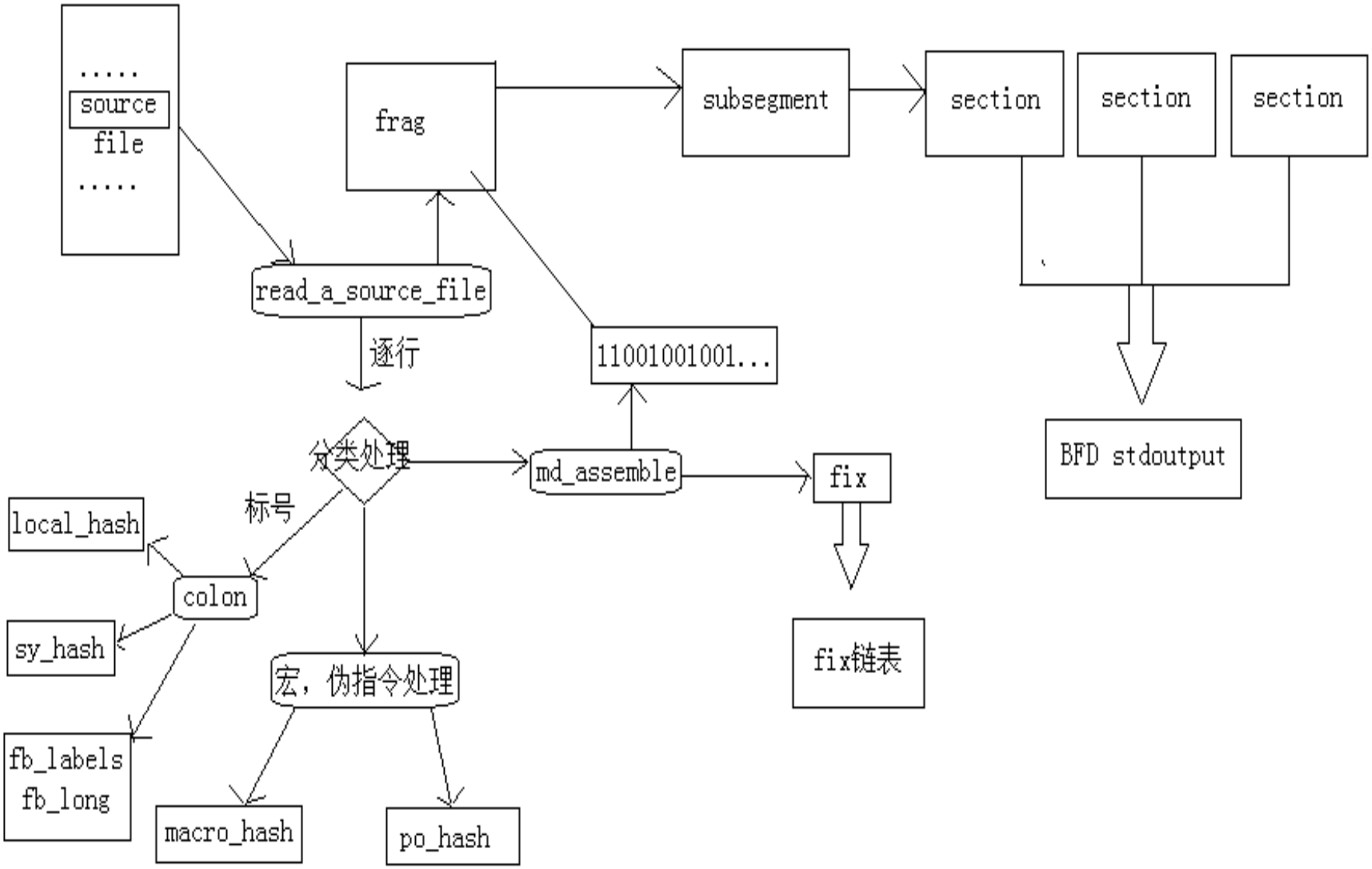
通过read_a_source_file这个函数进行扫描：

当扫描到标号（即<Label>:），会调用colon函数对标号进行处理，一般处理是将标号保存到相应的符号表（未解析符号表local_hash，已解析符号表sy_hash）和专门的标号表（fb_labels，fb_long），并且将标号所处的环境信息（当前位置，偏移及一些调试信息）保存到标号信息中；

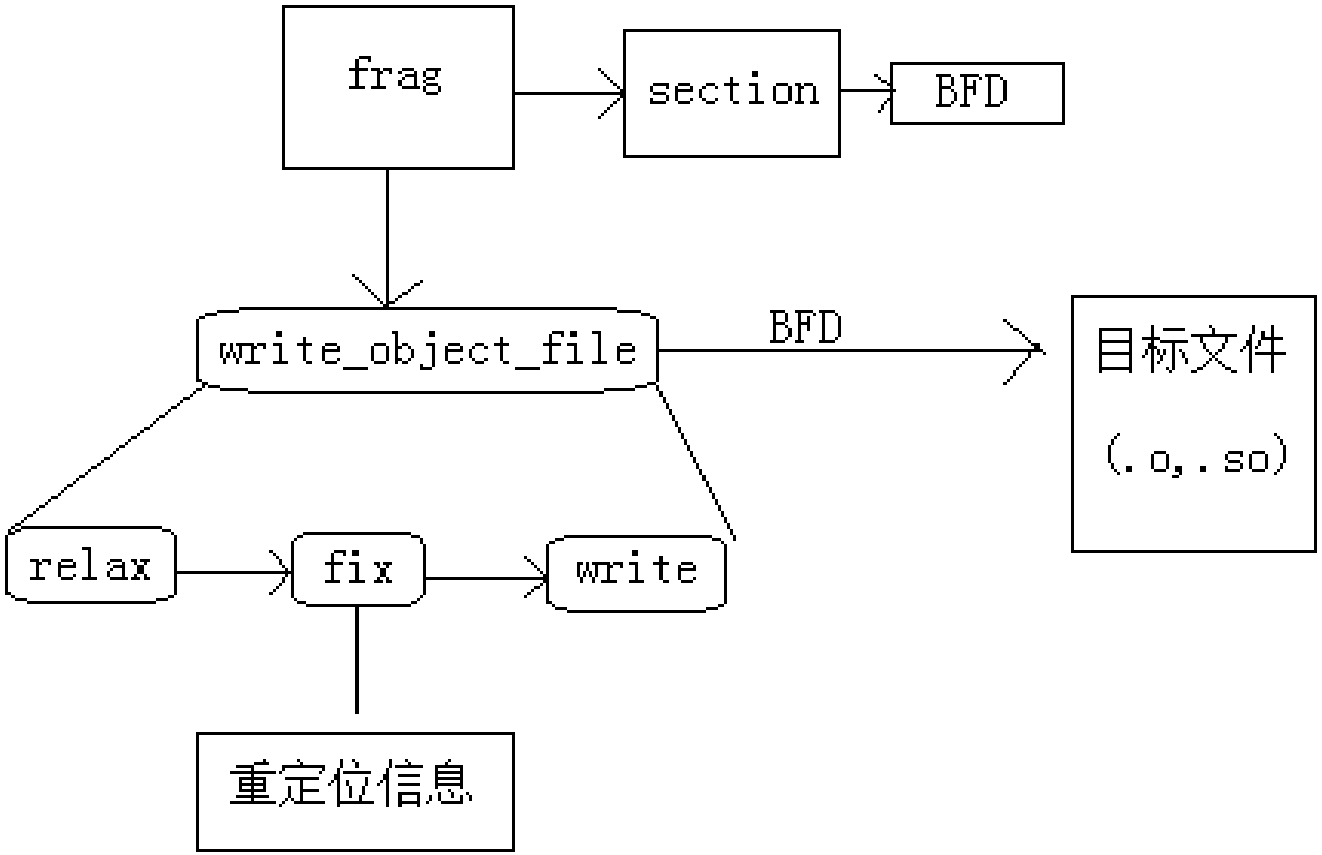
当扫描到宏指令时，将宏指令定义保存到macro_hash宏表中，或者遇到宏指令调用时，从macro_hash取得相关信息对源文件进行扩展，并重新开始扫描扩充信息；当扫描到伪指令时，同于宏指令的处理，只是信息保存到po_hash伪指令表中；

当扫描到其他信息即常规信息时，会调用机器相关汇编处理程序md_assemble进行汇编处理，这个函数的主要功能就是编译汇编代码为二进制信息，并且在需要重定位信息时，构建新的fix结构，填充到相应frag中，用于在后文当中的重定位工作（需要重定位信息的一般是跳转指令）。

经过如此的扫描处理之后，对应的BFD类型变量stdoutput中相关信息，已经填充完毕，即可对目标文件进行输出。


.....
source
file
.....
frag
subsegment
section
section
section
read_a_source_file
逐行
11001001001...
分类处理
md_assemble
fix
BFD stdoutput
标号
local_hash
colon
sy_hash
fb_labels
fb_long
宏，伪指令处理
macro_hash
po_hash
fix链表

目标文件输出流程


frag
section
BFD
write_object_file
BFD
目标文件
(.o,.so)
relax
fix
write
重定位信息

Fix用于解决什么问题？

- 一条指令转换为Frag后，部分内容可能无法在第一轮的扫描中确定，如：jsri label1.
- 这时候，生成一个Fix结构，指向该Frag.
- 汇编最后阶段，遍历Fix链表，修改Frag.

Fix数据结构

- fx_size fx_where
- fx_frag
- fx_addsy fx_addnumber
- fx_done fix_r_type
- …

write.h:38

与Fix相关的接口函数

- md_assemble() md_estimate_size_before_relax() md_convert_frag()
- md_apply_fix()
- tc_gen_reloc()
- bfd_relocate_section()

<tc-csky.c>

如何处理重定位？

- tc_gen_reloc

  <tc-csky.c>

- bfd_relocate_section

  <elf32-csky.c>

- 链接器脚本控制链接过程

  <generic.em elf32csky.sh ...>

Relax用于解决什么问题？

- 汇编代码解析后，指令保存于Frag数据结构中。
- 所有Frag大小都确定后，才能一个接一个地把Frag保存到目标文件中。
- 但某些Frag的大小，却取决于其他Frag的大小。

需要Relax处理的实例

- Jbr指令

指令	指令大小	跳转范围
br_16	2B	-1KB～1KB
br_32	4B	-64MB～64MB
jsri	8B	绝对地址0～4GB

Relax的原理

- （1）遍历Frag链表，根据其初始大小，计算（估计）它在目标文件中的地址。
- （2）根据估计值，为每个Frag选取合适的指令（可能改变了Frag大小）。
- （3）再次遍历Frag链表，根据当前大小，计算（估计）它在目标文件中的地址。
- （4）重复2～3步，直到一次遍历中没有Frag改变大小为止。

与Relax相关的接口函数

- md_assemble()
- md_estimate_size_before_relax()
- md_convert_frag()

<tc-csky.c>

大端与小端

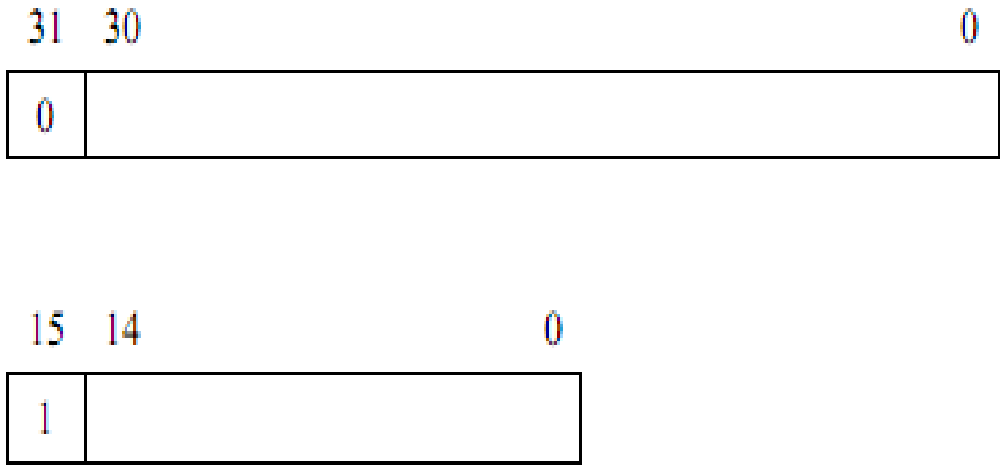
- 举例说明大小端，数据为0x01020304.
- 大端：

地址	0	1	2	3
数据	01	02	03	04

- 小端：

地址	0	1	2	3
数据	04	03	02	01

CSKY CPU通过指令编码中的最高位区分32/16指令，其中最高位为0代表32 位指令，最高位为 1代表16 位指令。具体的指令混编方式如图所示



大端与小端

- 举例说明大小端，数据为0x01020304.
- 大端：

地址	0	1	2	3
数据	01	02	03	04

- 小端：

地址	0	1	2	3
数据	04	03	02	01

- 我们需要的小端：

地址	0	1	2	3
数据	02	01	04	03

与大小端相关的接口函数

- md_number_to_chars()
- md_assemble() md_convert_frag()

<tc-csky.c>

- 反汇编器print_insn()

<csky-dis.c>

- 链接器bfd_relocate_section()

<elf32-csky.c>

链接器移植

文件	内容
emulparams/<format><target>.sh	链接器脚本参数
scripttempl/<format><target>.sc	链接器脚本
emultempl/<target><format>.em	仿真脚本文件
configure.tgt	目标信息
makefile.am	目标信息文件的生成规则

* 移植相关的文件

链接脚本（ Linker script ）

- 可以使用链接标本控制ld的链接过程。
- 用来控制ld的链接过程
  - 描述各输入文件的各节如何映射到输出文件的各节
  - 控制输出文件中各个节或者符号的内存布局
- 使用的语言为：
  - The ld command language，链接命令语言

链接脚本的命令

- 链接脚本的命令主要包括如下几类：
  - 设置入口点命令
  - 处理文件的命令
  - 处理文件格式的命令
  - 其他

常用的命令

- 设置入口点
  - 格式：ENTRY(symbol)
    设置symbol的值为执行程序的入口点。
- ld有多种方法设置执行程序的入口点，确定程序入口点的顺序如下：
  - ld命令的-e选项指定的值
  - Entry(symbol)指定的值
  - .text节的起始地址
  - 入口点为0

常用的命令

- INCULDE filename
  包含其他filename的链接描述文件
- INPUT(file,file,…)
  指定多个输入文件名

常用的命令

- OUTPUT_FORMAT(bfdname)
  指定输出文件的格式
- OUTPUT_ARCH (*bfdname*)
  - 指定目标机器体系结构，例如：
    OUTPUT_ARCH(arm)

常用的命令

- MEMORY：
  这个命令在用于嵌入式系统的链接描述文件中经常出现，它描述了各个内存块的起始地址和大小。格式如下：

```
MEMORY
{
    name [(attr)]:ORIGIN = origin,LENGTH = len
    …
}
```

- 例如：

```
MEMORY
{
    rom : ORIGIN=0x1000, LENGTH=0x1000
}
```

Memory举例

```
//标注嵌入式设备中各个内存块的地址划分情况
MEMORY
{
  //标注flash中断向量表的起始地址为0x01000000,长度为0x0400
  romvec: ORIGIN  = 0x01000000, LENGTH  = 0x0400
  //标注flash的起始地址为0x01000400,长度为0x011fffff-0x01000400
  flash: ORIGIN = 0x01000400, LENGTH = 0x011fffff-0x01000400
  //标注flash的结束地址在0x011fffff
  eflash : ORIGIN = 0x011fffff, LENGTH = 1
  ramvec:ORIGIN = 0x00000000, LENGTH = 0x0400
  ram : ORIGIN = 0x00000400, LENGTH = 0x0003ffff-0x00000400
  eram : ORIGIN = 0x0003ffff, LENGTH = 1
}
```

SECTIONS命令

- SECTIONS

  告诉ld如何把输入文件的各个节映射到输出文件的各个节中。

  - 在一个链接描述文件中只能有一个SECTIONS命令
- 在SECTIONS命令中可以使用的命令有三种：
  - 定义入口点
  - 赋值
  - 定义输出节

定位计数器

- 定位计数器，The Location Counter
  - 一个特殊的ld变量，使用“.”表示
  - 总是在SECTIONS中使用
  - 例如：

```
SECTIONS
{
    output :
    {
        file1(.text)
        . = . + 1000;
        file2(.text)
        . = . + 1000;
        file3(.text)
    } = 0x1234;
}
```

一个简单例子

- 下面是一个简单的例子：
  例中，输出文件包含text，data，bss三个节，而输入文件也只包含这3个节：

```
SECTIONS
{
    .=0x01000000;
    .text:{*(.text)};
    .=0x08000000;
    .data:{*(.data)};
    .bss:{*(.bss)};
}
```

谢谢！