**嵌入式学习笔记**

# 目录

[目录 1](#_Toc100500286)

[第一章 ARM内核与架构 3](#_Toc100500287)

[第二章 嵌入式开发基础 6](#_Toc100500288)

[**2.1 C语言常用技巧** 6](#_Toc100500289)

[**2.2 C/C++程序编译过程** 14](#_Toc100500290)

[**2.3 ELF文件** 17](#_Toc100500291)

[**2.3.1 原理介绍** 17](#_Toc100500292)

[**2.3.1 相关工具** 22](#_Toc100500293)

[**2.4 工具** 24](#_Toc100500294)

[**2.4.1 linux 串口工具** 24](#_Toc100500295)

[**2.4.2 linux 调试工具** 25](#_Toc100500296)

[**2.5 交叉编译环境** 31](#_Toc100500297)

[**2.5.1 交叉编译器** 31](#_Toc100500298)

[**2.5.2 构建交叉编译工具链待整理** 34](#_Toc100500299)

[**2.5.3 gcc llvm 和 clang** 34](#_Toc100500300)

[第三章 u-Boot 原理及移植 35](#_Toc100500301)

[**2.1 u-boot介绍** 35](#_Toc100500302)

[**2.2 常用指令及环境变量** 35](#_Toc100500303)

[**2.2.1 常用指令** 35](#_Toc100500304)

[**2.2.2 环境变量** 43](#_Toc100500305)

[**2.2.2 从uboot网络启动嵌入式设备** 44](#_Toc100500306)

[**2.3 u-boot编译架构** 47](#_Toc100500307)

[**2.4 源码分析** 47](#_Toc100500308)

[**2.4.1 链接脚本（lds文件）** 47](#_Toc100500309)

[**2.4.2 基本汇编语法** 51](#_Toc100500310)

[**2.4.3 复位之后函数调用流程** 57](#_Toc100500311)

[**2.4.4 uboot启动过程分析** 59](#_Toc100500312)

[**2.5 u-boot 移植** 61](#_Toc100500313)

[第四章 Linux内核 62](#_Toc100500314)

[**3.1 架构介绍** 62](#_Toc100500315)

[**3.2 内核裁剪** 62](#_Toc100500316)

[**3.3 源码（驱动）修改** 62](#_Toc100500317)

[第五章 设备树修改及使用 63](#_Toc100500318)

[第六章 根文件系统构建 64](#_Toc100500319)

[第七章 buildroot构建Linux系统 65](#_Toc100500320)

[第八章 yocto构建Linux系统 66](#_Toc100500321)

# 第一章 ARM内核与架构

任何一款ARM芯片都由两大部分组成：ARM内核，外设。

**ARM内核**

**ARM内核：** 包括了寄存器组、指令集、总线、存储器映射规则、中断逻辑和调试组件等。 内核是由ARM公司设计并以销售方式授权给个芯片厂商使用的（ARM公司本身不做芯片）。 比如为高速度设计的Cortex A8、A9都是ARMv7a 架构;Cortex M3、M4是ARMv7m架构;**前者是处理器（就是内核），后者是指令集的架构（也简称架构）**。

**外设部分** 包括计时器、A/D转换器、存储器、i2c、UART、SPI、ROM...等等，则完全由各芯片厂商自己设计并与ARM内核衔接配套。不同的芯片厂商就有不同的外设，因此构成了数量和规格庞大的ARM芯片产业。

**ARM指令集架构**

指令集的设计是处理器结构中最重要的一个部分，用ARM的术语称之为ISA（Instruction Set Architecture）。

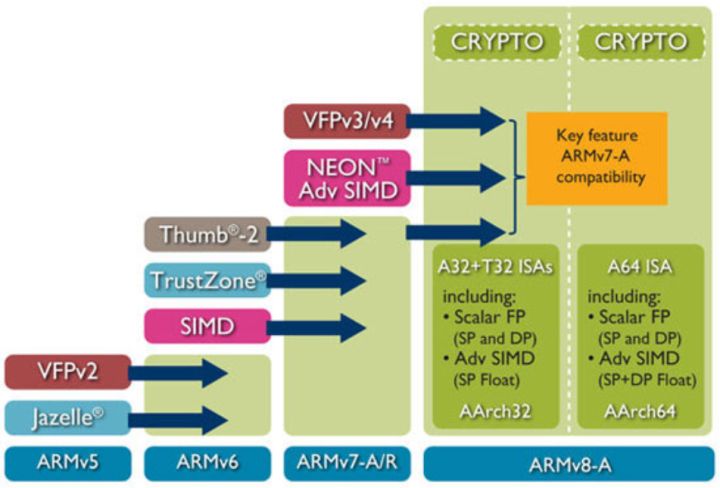
指令集可以说是cpu设计的灵魂，是打开CPU这个潘多拉魔盒的咒语，要想使用cpu，我们只能通过这些指令来操作cpu。

对于32位的cpu，这些指令就是一个个32位的01的序列，不同的值就代表了不同的机器指令，cpu的硬件能完美的解析并执行这些指令，比如寻址、运算、异常处理等等。

当我们用手机玩着王者荣耀的时候，要知道我们的每发的一招，其实最终都是被翻译成了一系列机器指令。

从1985年ARMv1架构诞生起，到2011年，ARM架构已经发展到了第八代ARMv8。

Cortex-A32/35/53/57/72/73/77/78采用的都是ARMv8架构，这是ARM公司的首款支持64位指令集的处理器架构。



**ARM11之前的处理器和指令集架构**

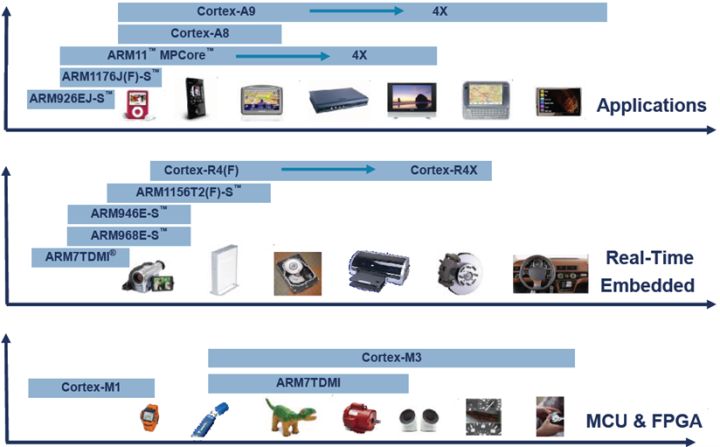
ARM11芯片之前，每一个芯片对应的架构关系如下：



**ARM11之后处理器和指令集架构**

ARM11芯片之后，也就是从ARMv7架构开始，ARM的命名方式有所改变。

新的处理器家族，改以Cortex命名，并分为三个系列，分别是Cortex-A，Cortex-R，Cortex-M。很巧合，又是这三个字母**A、R、M**。



**Cortex-A系列（A：Application）**

针对日益增长的消费娱乐和无线产品设计，用于具有高计算要求、运行丰富操作系统及提供交互媒体和图形体验的应用领域，如智能手机、平板电脑、汽车娱乐系统、数字电视，智能本、电子阅读器、家用网络、家用网关和其他各种产品。

**Cortex-R系列 （R：Real-time）**

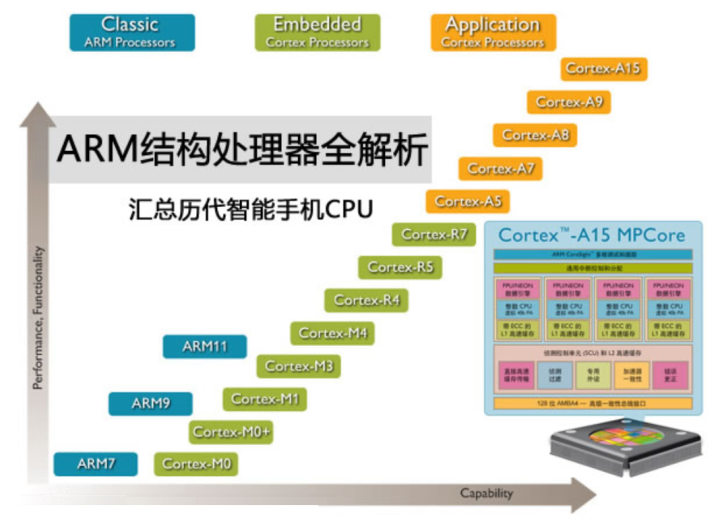
针对需要运行实时操作的系统应用，面向如汽车制动系统、动力传动解决方案、大容量存储控制器等深层嵌入式实时应用。

**Cortex-M系列（M：Microcontroller）**

该系列面向微控制器领域，主要针对成本和功耗敏感的应用，如智能测量、人机接口设备、汽车和工业控制系统、家用电器、消费性产品和医疗器械等。

**Cortex-SC系列（SC：SecurCore）**

其实，除了上述三大系列之外，还有一个主打安全的Cortex-SC系列（SC：SecurCore），主要用于政府安全芯片。



ARM11系列包括了ARM11MPCore处理器、ARM1176处理器、ARM1156处理器、ARM1136处理器，它们是基于ARMv6架构。

ARM Cortex-A5处理器、Cortex-A7处理器、Cortex-A8处理器、Cortex-A9处理器、Cortex-A15处理器隶属于Cortex-A系列，基于ARMv7-A架构。

Cortex-A53、Cortex-A57两款处理器属于Cortex-A50系列，首次采用64位ARMv8架构。

2020年ARM最近发布了一款全新的CPU架构Cortex-A78，是基于ARMv8.2指令集。

# 第二章 嵌入式开发基础

**2.1 C语言常用技巧**

**1. 关于#和##**

在C语言的宏中，#的功能是将其后面的宏参数进行字符串化操作（Stringfication），简单说就是在对它所引用的宏变量通过替换后在其左右各加上一个双引号。

例如：

#define WARN\_IF(EXP) \

do{ \

if (EXP) \

fprintf(stderr, "Warning: **" #EXP** "\n"); \

}while(0)

那么实际使用中会出现下面所示的替换过程：

WARN\_IF (divider == 0);

被替换为

do {

if (divider == 0)

fprintf(stderr, "Warning" **"divider == 0"** "\n");

} while(0);

这样每次divider（除数）为0的时候便会在标准错误流上输出一个提示信息。

而##被称为连接符（concatenator），用来将两个Token连接为一个Token。注意这里连接的对象是Token就行，而不一定是宏的变量。比如你要做一个菜单项命令名和函数指针组成的结构体的数组，并且希望在函数名和菜单项命令名之间有直观的、名字上的关系。那么下面的代码就非常实用：

struct command

{

char \*name;

void (\*function) (void);

};

#define COMMAND(NAME) { NAME, NAME ## \_command }

// 这里NAME##\_command 里的NAME会被替换为给定变量的名字，而不是值

// 然后你就用一些预先定义好的命令来方便的初始化一个command结构的数组了：

struct command commands[] = {

COMMAND(quit),

COMMAND(help),

...

}

COMMAND宏在这里充当一个代码生成器的作用，这样可以在一定程度上减少代码密度，间接地也可以减少不留心所造成的错误。我们还可以n个##符号连接 n+1个Token，这个特性也是#符号所不具备的。比如：

#define LINK\_MULTIPLE(a,b,c,d) a##\_##b##\_##c##\_##d

typedef struct \_record\_type LINK\_MULTIPLE(name,company,position,salary);

// 这里这个语句将展开为：

// typedef struct \_record\_type name\_company\_position\_salary;

**2.1 变参列表**

可变参数列表是通过宏来实现的，这些宏定义于 stdarg.h 头文件，他是标准库的一部分。这个头文件声明了一个类型 vs\_list 和三个宏： va\_start 、 va\_arg 和 va\_end 。我们可以声明一个类型为 va\_list 的变量，与这几个宏配合使用，访问参数的值。

/\*

\* 计算指定数量的数据的平均值

\*/

#include <stdarg.h>

float average(int n\_values, ...)

{

**va\_list** var\_arg;

int sum;

int count;

/\*

\* 准备访问可变参数

\*/

**va\_start**(var\_arg, n\_values);

/\*

\* 添加取自可变参数列表的值

\*/

for(count = 0; count < n\_values; count++){

sum += **va\_arg**(var\_arg, int);

}

/\*

\* 完成处理可变参数

\*/

**va\_end**(var\_arg);

return sum / n\_values;

}

int main(int argc, char \*argv)

{

printf("average is: %d \n", average(1, 2, 3));

}

注意参数列表中的省略号，他提示此处可能传递数量和类型未知的参数。

在函数中声明了一个名为 var\_arg 的变量，它用于访问参数列表的未确定部分。这个变量通过 va\_start 来初始化，它的第一个参数是 va\_lista 变量的名字（即程序中的var\_arg），第二个参数是省略号前最后一个有名字的参数。初始化过程把 var\_arg 变量设置为指向可变参数部分的第一个参数（n\_values）。

|  |
| --- |
| 栈底 |
| … |
| param3 |
| param2 |
| param1 (n\_values) |
| var\_arg |
| 栈顶 |

为了访问参数，需要使用 va\_arg，这个宏接受两个参数： va\_list 变量和参数列表当前参数的类型。在有些函数中可能要通过前面获得的数据来判断当前参数的类型（例如，printf检查格式字符串中的字符来判断他需要打印的参数类型），va\_arg 返回这个参数的值，并使 var\_arg 指向下一个可变参数。

最后，当访问完毕最后一个可变参数之后，需要调用va\_end。

**2.2 变参宏**

**标识符 \_\_VA\_ARGS\_\_**

\_\_VA\_ARGS\_\_ 是在 C99 中增加的新特性。虽然 C89 引入了一种标准机制，允许定义具有可变数量参数的函数，但是 C89 中不允许这种定义可变数量参数的方式出现在宏定义中。C99 中加入了 \_\_VA\_ARGS\_\_ 关键字，用于支持在宏定义中定义可变数量参数，用于接收 ... 传递的多个参数。

\_\_VA\_ARGS\_\_ 只能出现在使用了省略号的像函数一样的宏定义里。例如 #define myprintf(...) fprintf(stderr, \_\_VA\_ARGS\_\_)。

**带 ‘#’ 的标识符 #\_\_VA\_ARGS\_\_**

预处理标记 ‘#’ 用于将宏定义参数转化为字符串，因此 #\_\_VA\_ARGS\_\_ 会被展开为参数列表对应的字符串。

示例：

#define showlist(...) put(#\_\_VA\_ARGS\_\_)

测试如下：

showlist(The first, second, and third items.);

showlist(arg1, arg2, arg3);

输出结果分别为：

The first, second, and third items.

arg1, arg2, arg3

**带 ‘##’ 的标识符 ##\_\_VA\_ARGS\_\_**

##\_\_VA\_ARGS\_\_ 是 GNU 特性，不是 C99 标准的一部分，C 标准不建议这样使用，但目前已经被大部分编译器支持。

标识符 ##\_\_VA\_ARGS\_\_ 的意义来自 ‘##’，主要为了解决一下应用场景：

#define myprintf\_a(fmt, ...) printf(fmt, \_\_VA\_ARGS\_\_)

#define myprintf\_b(fmt, ...) printf(fmt, ##\_\_VA\_ARGS\_\_)

应用：

myprintf\_a("hello");

myprintf\_b("hello");

myprintf\_a("hello: %s", "world");

myprintf\_b("hello: %s", "world");

这个时候，编译器会报错，如下所示：

applications\main.c: In function 'main':

applications\main.c:26:57: error: expected expression before ')' token

#define myprintf\_a(fmt, ...) printf(fmt, \_\_VA\_ARGS\_\_)

^

applications\main.c:36:5: note: in expansion of macro 'myprintf\_a'

myprintf\_a("hello");

为什么呢？

我们展开 myprintf\_a("hello"); 之后为 printf("hello",)。因为没有不定参，所以，\_\_VA\_ARGS\_\_ 展开为空白字符，这个时候，printf 函数中就多了一个 ‘,’（逗号），导致编译报错。而 ##\_\_VA\_ARGS\_\_ 在展开的时候，因为 ‘##’ 找不到连接对象，会将 ‘##’ 之前的空白字符和 ‘,’（逗号）删除，这个时候 printf 函数就没有了多余的 ‘,’（逗号）。

**3. main函数入参**

main函数入参有两种写法：

**int main (int argc, char \*argv[])**

// [ ] 的优先级高于 \*，这样第二个参数可以解释为定义了一个数组，该数组中的所有元素都是指向C风格字符串的指针（即 char \*）。

另一种写法：

**int main (int argc, char \*\*argv)**

**4. C 语言中的constructor与destructor**

可以给一个函数赋予constructor或destructor，其中constructor在main开始运行之前被调用，destructor在main函数结束后被调用。如果有多个constructor或destructor，可以给每个constructor或destructor赋予优先级，对于constructor，优先级数值越小，运行越早。destructor则相反。

例如：

#include <stdio.h>

\_\_attribute\_\_((constructor(101))) void foo()

{

printf("in constructor of foo\n");

}

\_\_attribute\_\_((constructor(102))) void foo1()

{

printf("in constructor of foo1\n");

}

\_\_attribute\_\_((destructor)) void bar()

{

printf("in constructor of bar\n");

}

int main()

{

printf("in main\n");

return 0;

}

其中constructor后边括号中为优先级，也可以不指定优先级。

**5. \_\_FUNCTION\_\_ \_\_func\_\_等**

**\_\_FUNCTION\_\_**： 当前函数名

**\_\_func\_\_**：当前函数名

**\_\_PRETTY\_FUNCTION\_\_**： 非标准宏。这个宏比\_\_FUNCTION\_\_功能更强, 若用g++编译C++程序, \_\_FUNCTION\_\_只能输出类的成员名,不会输出类名;而\_\_PRETTY\_FUNCTION\_\_则会以 <return-type> <class-name>::<member-function-name>(<parameters-list>) 的格式输出成员函数的详悉信息(注: 只会输出parameters-list的形参类型, 而不会输出形参名).若用gcc编译C程序,\_\_PRETTY\_FUNCTION\_\_跟\_\_FUNCTION\_\_的功能相同。

**\_\_LINE\_\_**：当前程序行的行号，表示为十进制整型常量，#line指令可以改变它的值。

例如： #line 200 指定下一行的\_\_LINE\_\_为200,重点是line的下一行是200

**\_\_FILE\_\_**：当前源文件名，表示字符串型常量

**\_\_DATE\_\_**：转换的日历日期，表示为Mmm dd yyyy 形式的字符串常量，Mmm是由asctime产生的。

**\_\_TIME\_\_**：转换的时间，表示"hh:mm:ss"形式的字符串型常量，是由asctime产生的。

**\_\_SIZEOF\_POINTER\_\_**：当前平台指针的byte数

**\_\_WORDSIZE**： 这个好像针对嵌入式平台

**\_\_SIZEOF\_INT\_\_**

**\_\_SIZEOF\_LONG\_\_**

**\_\_SIZEOF\_LONG\_LONG\_\_**

**\_\_SIZEOF\_SHORT\_\_**

**\_\_SIZEOF\_POINTER\_\_**

**\_\_SIZEOF\_FLOAT\_\_**

**\_\_SIZEOF\_DOUBLE\_\_**

**\_\_SIZEOF\_LONG\_DOUBLE\_\_**

**\_\_SIZEOF\_SIZE\_T\_\_**

**\_\_SIZEOF\_WCHAR\_T\_\_**

**\_\_SIZEOF\_WINT\_T\_\_**

**\_\_SIZEOF\_PTRDIFF\_T\_\_**

更多宏定义可以参考：https://gcc.gnu.org/onlinedocs/cpp/Common-Predefined-Macros.html

6. do{}while(0)

为了保证宏定义的使用者能无编译错误地使用宏，可以考虑使用 do{}while(0)的形式。

7. 数组和指针

（1）编译器为每个变量分配一个地址（左值）。这个地址在编译时可知，而且该变量在运行时一直保存于这个地址。相反，存储与变量中的值（他的右值）只有在运行时才可知。所以定义一个数组，例如：int num[5]; 。这时候num跟一个地址绑定在一起，如果编译器需要一个地址（可能还要加上偏移量）来执行某种操作，他就可以直接进行操作，不需要增加指令首先取得具体的地址。相反，对于指针，必须首先在运行时取得他的当前值，然后才能对他进行解除引用操作。

（2）在表达式中，指针和数组是可以互换的，因为他们在编译器里的最终形式都是指针，并且都可以进行取下标操作。

（3）C语言把数组下标作为指针的偏移量。

（4）在函数参数的声明中，数组名被编译器当作指向该数组第一个元素的指针。编译器只向函数传递数组的地址，而不是整个数组的拷贝。

8. const

int \*pi; 普通指针

int const \*pci; 可以修改指针指向的地址，但是不能修改指针指向的值。值是常量

等价于 const int \*cpi; 被称为***常量指针***（底层），指针指向的内容不能更改。

int \* const cpi; 可以修改指针指向的值，但不能修改指针指向的地址，指针是常量

被称为***指针常量***（顶层），指针指向的地址不能更改。

int const \* const cpci; 指针的值和指针指向的值均不能修改，指针和值都是常量

注意：当用实参初始化函数形参时会忽略掉顶层const（值是常量）。换句话说，形参的顶层const被忽略掉了。当形参有顶层const时，传给它常量对象或者非常量对象都是可以的。这就导致下边两个定义是一样的，因为顶层const被忽略掉了。

void fcn(const int i){…}

void fcn(int i){…}

9. 友元

类可以允许其他类或者函数访问他的非公有成员，方法是另其他类或者函数成为他的友元。如果类想把一个函数作为他的友元，只需要增加一条以friend关键字开始的函数声明语句即可。

例如：

class Sales\_data{

friend Sales\_data add();

public:

…

private:

…

};

友元的声明只能出现在类定义的内部，但是在类内出现的具体位置不限。友元不是类的成员，也不受他所在区域访问控制级别的约束。

另外，类还可以把其他的类定义为友元，也可以把其他类（之前已经定义过的）的成员函数定义为友元。此外，友元函数能定义在类的内部，这样的函数是隐式的。

10. 用单引号括起的一个字符代表一个整数，整数值对应于该字符在编译器采用的字符集中的序列值。因此，对于采用ASCII字符集的编译器而言，‘a’的含义与0141（八进制）或者97（十进制）严格一致。

用双引号括起的一个字符代表一个指向无名数组起始字符的指针，该数组被双引号之间的字符以及一个额外的二进制值为零的字符‘\0’初始化。

因此如下的语句时错误的，因为单引号括起来的是一个整数，而不是char指针：

char \*slash = ‘/’;

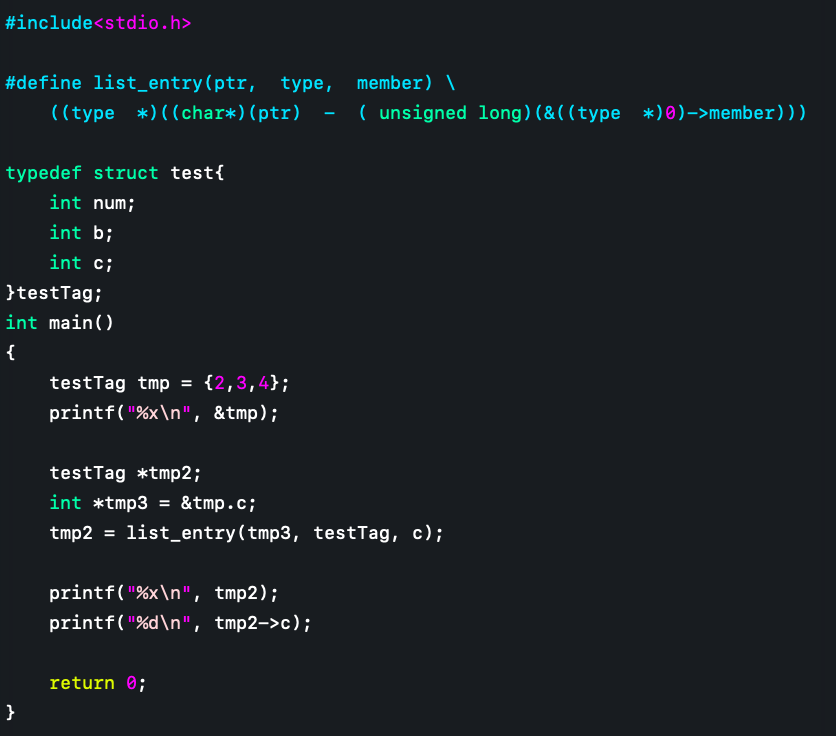
11. ANSIC标准允许值为0的常量被强制转换成任何一种类型的指针，并且转换的结果是个NULL，因此((type \*)0)的结果就是一个类型为type \*的NULL指针。

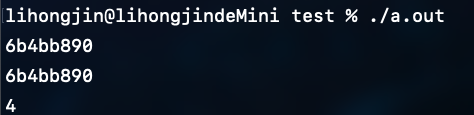
如果利用这个NULL指针来访问type的成员当然是非法的，但typeof( ((type \*)0)->member )是想取该成员的类型，所有编译器不会生成访问type成员的代码，类似的代码&( ((type \*)0)->member )在最前面有个取地址符&，它的意图是想取member的地址，所以编译器同样会优化为直接取地址。

如下宏是根据结构体成员获取结构体地址：

#define list\_entry(ptr, type, member) \

((type \*)((char\*)(ptr) - ( unsigned long)(&((type \*)0)->member)))





12. 判断大端小端

大端：数据高位在低地址，低位在高地址

小端：数据地位在低地址，高位在高地址

Linux和mac默认栈从低地址向高地址增长

判断大小端的代码：

|  |
| --- |
| #include <stdio.h>  int main() {  int i = 0x11223344;  char \*p;  p = (char \*) &i;  if (\*p == 0x44) {  printf("Little endian\n");  }  else {  printf("Big endian\n");  }  return 0;  } |

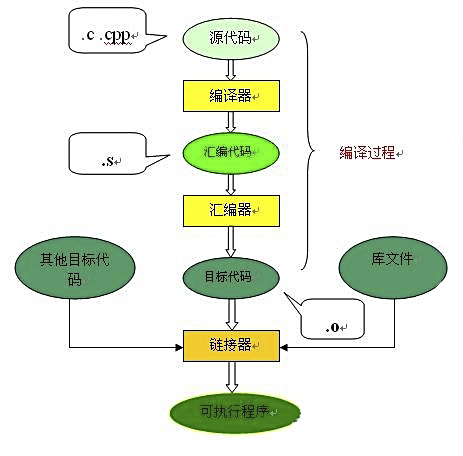
13. \_\_attribute\_\_((packed)) 和 \_\_attribute\_\_((aligned(4)))

packed 在用在结构体里时，指的是所有元素紧密排列，不进行填充

aligned(4) 在用在结构体里时，指的是结构体存放的起始地址是4字节对齐的，自然结构体也占4字节对齐的内存空间

**2.2 C/C++程序编译过程**

整个代码的编译过程分为编译和链接两个过程，编译对应图中的大括号括起的部分，其余则为链接过程。



**1. 编译过程**

编译过程又可以分成两个阶段：编译和汇编。

**编译**

编译是读取源程序（字符流），对之进行词法和语法的分析，将高级语言指令转换为功能等效的汇编代码，源文件的编译过程包含两个主要阶段：

***编译预处理***

读取c源程序，对其中的伪指令（以# 开头的指令）和特殊符号进行处理。

伪指令主要包括以下四个方面：

1) 宏定义指令，如# define Name TokenString，# undef等。

对于前一个伪指令，预编译所要做的是将程序中的所有Name用TokenString替换，但作为字符串常量的 Name则不被替换。对于后者，则将取消对某个宏的定义，使以后该串的出现不再被替换。

2) 条件编译指令，如# ifdef，# ifndef，# else，# elif，# endif等。

这些伪指令的引入使得程序员可以通过定义不同的宏来决定编译程序对哪些代码进行处理。预编译程序将根据有关的文件，将那些不必要的代码过滤掉。

3) 头文件包含指令，如# include "FileName" 或者# include < FileName> 等。

在头文件中一般用伪指令# define定义了大量的宏（最常见的是字符常量），同时包含有各种外部符号的声明。

采用头文件的目的主要是为了使某些定义可以供多个不同的C源程序使用。因为在需要用到这些定义的C源程序中，只需加上一条# include语句即可，而不必再在此文件中将这些定义重复一遍。预编译程序将把头文件中的定义统统都加入到它所产生的输出文件中，以供编译程序对之进行处理。

包含到c源程序中的头文件可以是系统提供的，这些头文件一般被放在/ usr/ include目录下。在程序中# include它们要使用尖括号（< >）。另外开发人员也可以定义自己的头文件，这些文件一般与c源程序放在同一目录下，此时在# include中要用双引号（""）。

4) 特殊符号，预编译程序可以识别一些特殊的符号。

例如在源程序中出现的\_\_LINE\_\_标识将被解释为当前行号（十进制数），\_\_FILE\_\_则被解释为当前被编译的C源程序的名称。预编译程序对于在源程序中出现的这些串将用合适的值进行替换。

预编译程序所完成的基本上是对源程序的“**替代**”工作。经过此种替代，生成一个没有宏定义、没有条件编译指令、没有特殊符号的输出文件。这个文件的含义同没有经过预处理的源文件是相同的，但内容有所不同。下一步，此输出文件将作为编译程序的输入而被翻译成为机器指令。

***编译、优化阶段***

经过预编译得到的输出文件中，只有常量；如数字、字符串、变量的定义，以及C语言的关键字，如main, if , else , for , while , { , } , + , - , \* , \ 等等。

**编译程序**所要作得工作就是通过词法分析和语法分析，在确认所有的指令都符合语法规则之后，将其翻译成等价的中间代码表示或汇编代码。

**优化处理**是编译系统中一项比较艰深的技术。它涉及到的问题不仅同编译技术本身有关，而且同机器的硬件环境也有很大的关系。优化一部分是对中间代码的优化。这种优化不依赖于具体的计算机。另一种优化则主要针对目标代码的生成而进行的。

对于前一种优化，主要的工作是删除公共表达式、循环优化（代码外提、强度削弱、变换循环控制条件、已知量的合并等）、复写传播，以及无用赋值的删除，等等。

后一种类型的优化同机器的硬件结构密切相关，最主要的是考虑是如何充分利用机器的各个硬件寄存器存放有关变量的值，以减少对于内存的访问次数。另外，如何根据机器硬件执行指令的特点（如流水线、RISC、CISC、VLIW等）而对指令进行一些调整使目标代码比较短，执行的效率比较高，也是一个重要的研究课题。

经过优化得到的汇编代码必须经过汇编程序的汇编转换成相应的机器指令，方可能被机器执行。

**汇编**

汇编过程实际上指把汇编语言代码翻译成目标机器指令的过程。对于被翻译系统处理的每一个C语言源程序，都将最终经过这一处理而得到相应的目标文件。目标文件中所存放的也就是与源程序等效的目标的机器语言代码。

**目标文件由段组成。通常一个目标文件中至少有两个段：**

1) 代码段：该段中所包含的主要是程序的指令。该段一般是可读和可执行的，但一般却不可写。

2) 数据段：主要存放程序中要用到的各种全局变量或静态的数据。一般数据段都是可读，可写，可执行的。

UNIX环境下主要有三种类型的目标文件：

1) 可重定位文件

其中包含有适合与其它目标文件链接来创建一个可执行的或者共享的目标文件的代码和数据。通过指令 readelf -S 可以发现，可重定位文件会多一个.rel.text段。通过指令 readelf -r 可以读取重定义表。

2) 共享的目标文件

这种文件存放了适合于在两种上下文里链接的代码和数据。

第一种是链接程序可把它与其它可重定位文件及共享的目标文件一起处理来创建另一个目标文件；

第二种是动态链接程序将它与另一个可执行文件及其它的共享目标文件结合到一起，创建一个进程映象。

3) 可执行文件

它包含了一个可以被操作系统创建一个进程来执行之的文件。

汇编程序生成的实际上是第一种类型的目标文件。对于后两种还需要其他的一些处理方能得到，这个就是链接程序的工作了。

**2. 链接过程**

由汇编程序生成的目标文件并不能立即就被执行，其中可能还有许多没有解决的问题。

例如，某个源文件中的函数可能引用了另一个源文件中定义的某个符号（如变量或者函数调用等）；在程序中可能调用了某个库文件中的函数，等等。所有的这些问题，都需要经链接程序的处理方能得以解决。

链接程序的主要工作就是将有关的目标文件彼此相连接，也即将在一个文件中引用的符号同该符号在另外一个文件中的定义连接起来，使得所有的这些目标文件成为一个能够被操作系统装入执行的统一整体。

根据开发人员指定的同库函数的链接方式的不同，链接处理可分为两种：

1) 静态链接

在这种链接方式下，函数的代码将从其所在的静态链接库中被拷贝到最终的可执行程序中。这样该程序在被执行时这些代码将被装入到该进程的虚拟地址空间中。**静态链接库实际上是一个目标文件的集合**，其中的每个文件含有库中的一个或者一组相关函数的代码。

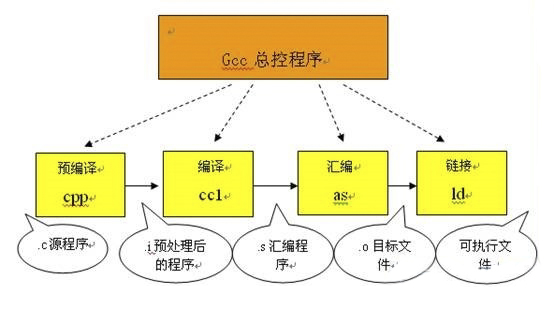
2) 动态链接

在此种方式下，函数的代码被放到称作是动态链接库或共享对象的某个目标文件中。链接程序此时所作的只是在最终的可执行程序中记录下共享对象的名字以及其它少量的登记信息。在此可执行文件被执行时，动态链接库的全部内容将被映射到运行时相应进程的虚地址空间。动态链接程序将根据可执行程序中记录的信息找到相应的函数代码。

对于可执行文件中的函数调用，可分别采用动态链接或静态链接的方法。使用动态链接能够使最终的可执行文件比较短小，并且当共享对象被多个进程使用时能节约一些内存，因为在内存中只需要保存一份此共享对象的代码。但并不是使用动态链接就一定比使用静态链接要优越。在某些情况下动态链接可能带来一些性能上损害。

**3. GCC的编译链接**

我们在linux使用的gcc编译器便是把以上的几个过程进行捆绑，使用户只使用一次命令就把编译工作完成，这的确方便了编译工作，但对于初学者了解编译过程就很不利了，下图便是gcc代理的编译过程：



从上图可以看到：

1) 预编译

将.c 文件转化成 .i文件

使用的gcc命令是：gcc –E

对应于预处理命令cpp

2) 编译

将.c/.h文件转换成.s文件

使用的gcc命令是：gcc –S

对应于编译命令 cc –S

3) 汇编

将.s 文件转化成 .o文件

使用的gcc 命令是：gcc –c

对应于汇编命令是 as

4) 链接

将.o文件转化成可执行程序

使用的gcc 命令是： gcc

对应于链接命令是 ld

总结起来编译过程就上面的四个过程：预编译处理(.c) －－> 编译、优化程序（.s、.asm）－－> 汇编程序(.obj、.o、.a、.ko) －－> 链接程序（.exe、.elf、.axf等）。

参考博客：<https://www.cnblogs.com/mickole/articles/3659112.html>

在博客后半部分有对C++的编译说明，与C大致类似，但有差异

**2.3 ELF文件**

**2.3.1 原理介绍**

**ELF文件：** https://blog.csdn.net/feglass/article/details/51469511

<https://blog.csdn.net/liugaigai427/article/details/86742062>

https://www.jianshu.com/p/dd5aec5826da

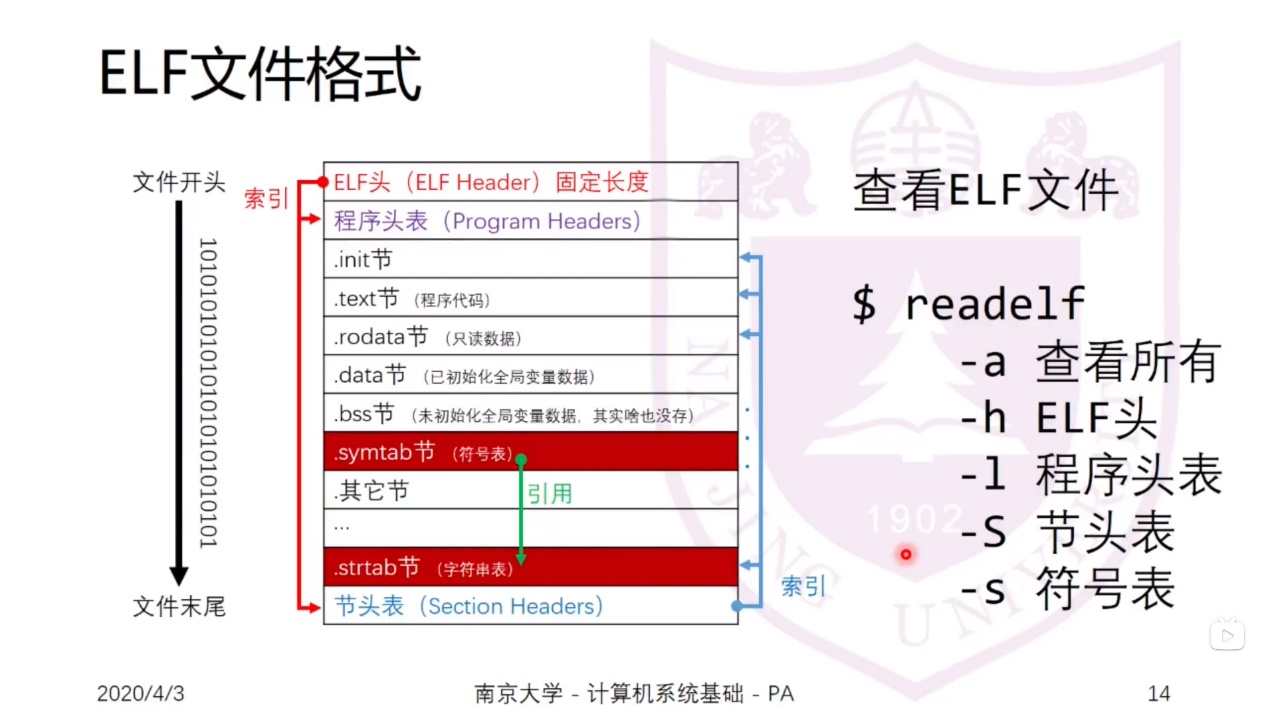
<https://blog.csdn.net/weixin_44316996/article/details/107396385?utm_medium=distribute.pc_relevant.none-task-blog-BlogCommendFromBaidu-1.control&depth_1-utm_source=distribute.pc_relevant.none-task-blog-BlogCommendFromBaidu-1.control>

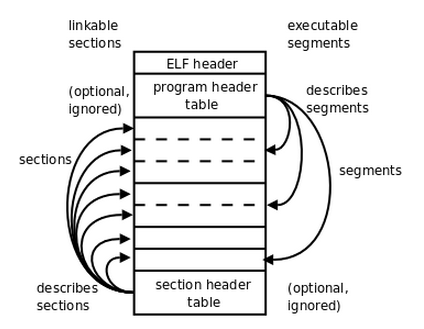
**ELF文件(B站视频 推荐 已下载在当前目录下：elf文件及符号表解析)：**

<https://www.bilibili.com/video/BV1e54y1d74j?from=search&seid=157787288945585334>

**该up主有好多相关视频，可以关注：**

<https://space.bilibili.com/284613991/channel/detail?cid=103368>





**可重定位文件（Relocatable File）** 包含适合于与其他目标文件链接来创建*可执行文件*或者*共享目标文件*的代码和数据。

**可执行文件（Executable File）** 包含适合于执行的一个程序，此文件*规定了 exec() 如何创建一个程序的进程映像*。

**共享目标文件（Shared Object File）** 包含可在两种上下文中链接的代码和数据。*首先链接编辑器可以将它和其它可重定位文件和共享目标文件一起处理，生成另外一个目标文件。其次，动态链接器（Dynamic Linker）可能将它与某个可执行文件以及其它共享目标一起组合，创建进程映像。*

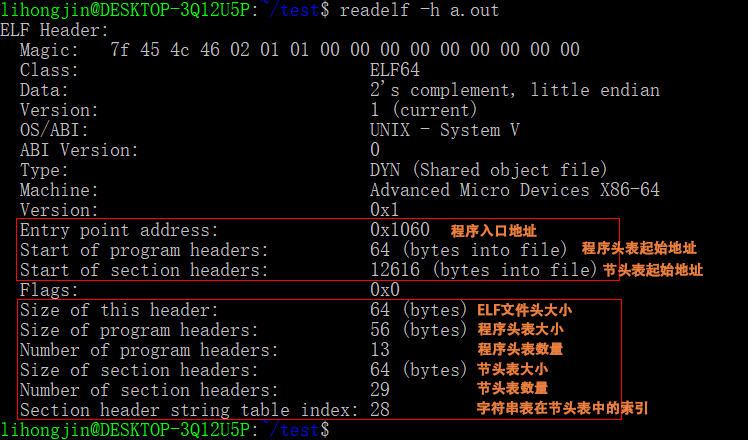
文件开始处是一个**ELF 头部**（ELF Header），用来描述整个文件的组织。节区部分包含链接视图的大量信息：指令、数据、符号表、重定位信息等等。

**程序头部表（Program Header Table）**，如果存在的话，告诉系统如何创建进程映像。用来构造进程映像的目标文件必须具有程序头部表，可重定位文件不需要这个表。

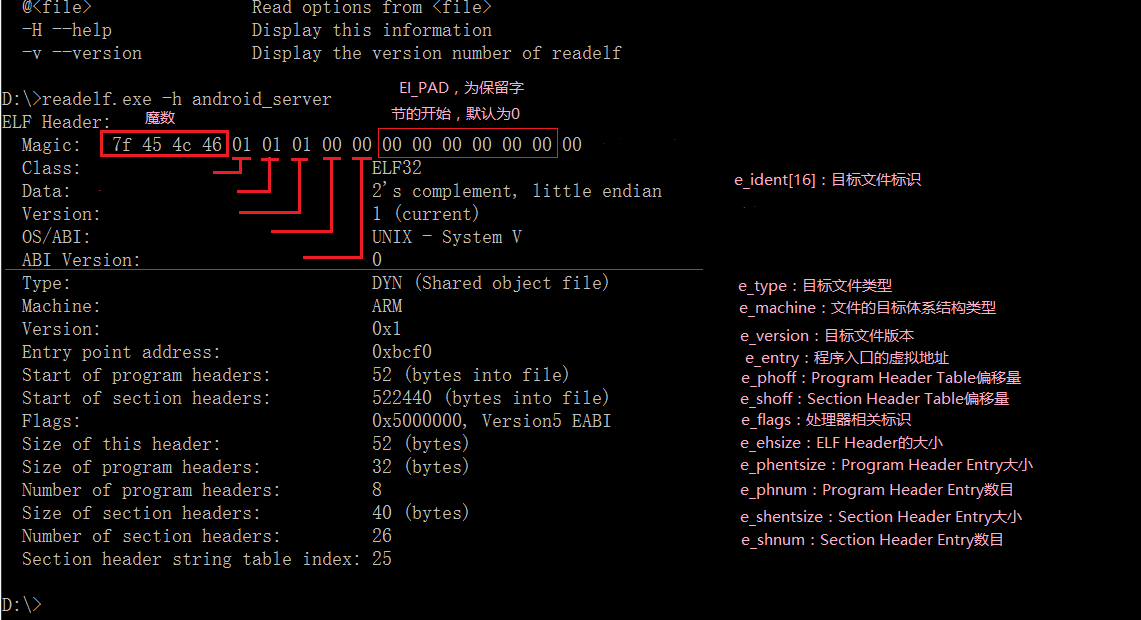
**节区头部表（Section Heade Table）**包含了描述文件节区的信息，每个节区在表中都有一项，每一项给出诸如节区名称、节区大小这类信息。用于链接的目标文件必须包含节区头部表，其他目标文件可以有，也可以没有这个表。

注意：尽管图中显示的各个组成部分是有顺序的，实际上除了 ELF 头部表以外，其他节区和段都没有规定的顺序。

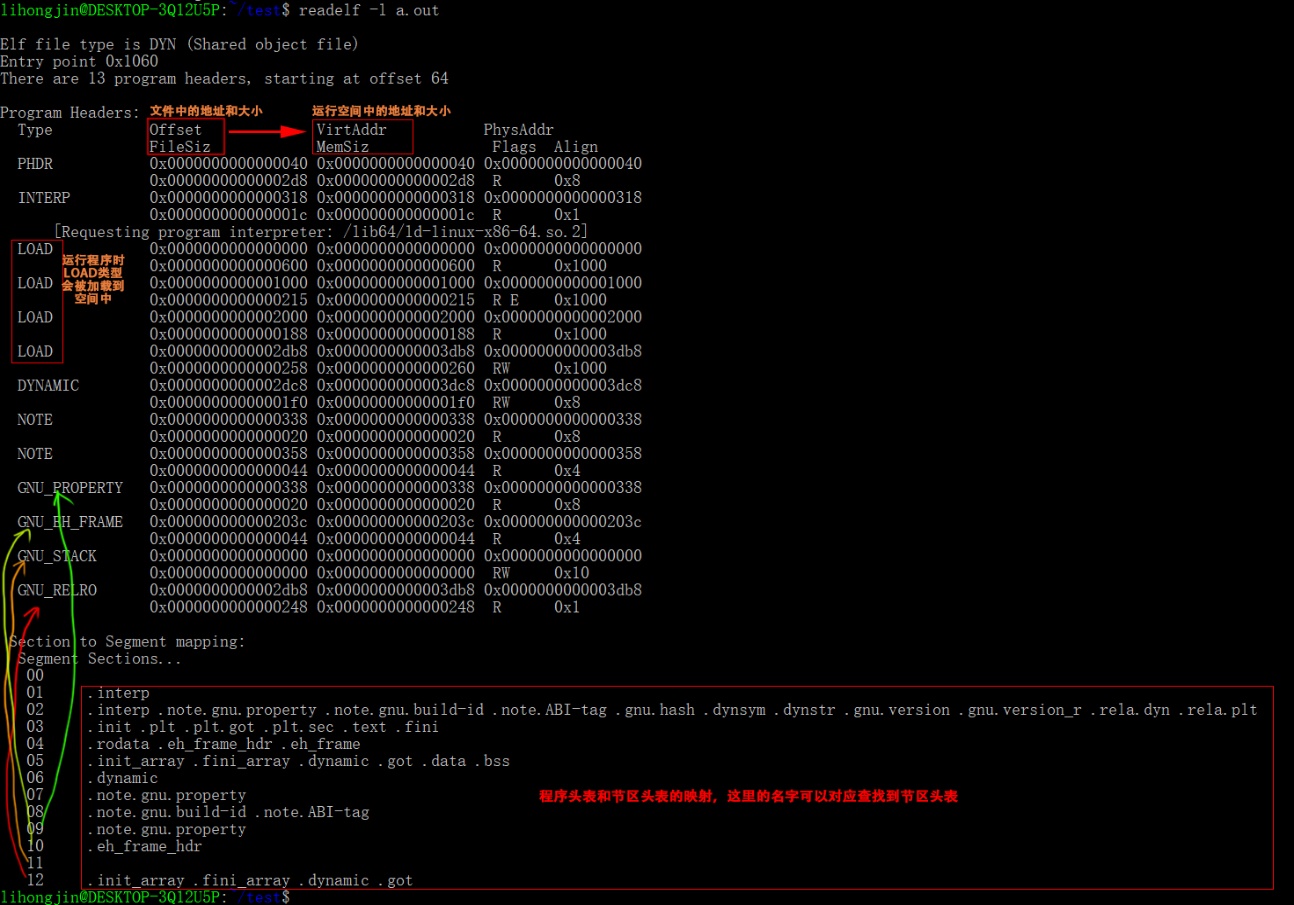
**readelf -h <elf文件>可以查询elf头信息：**



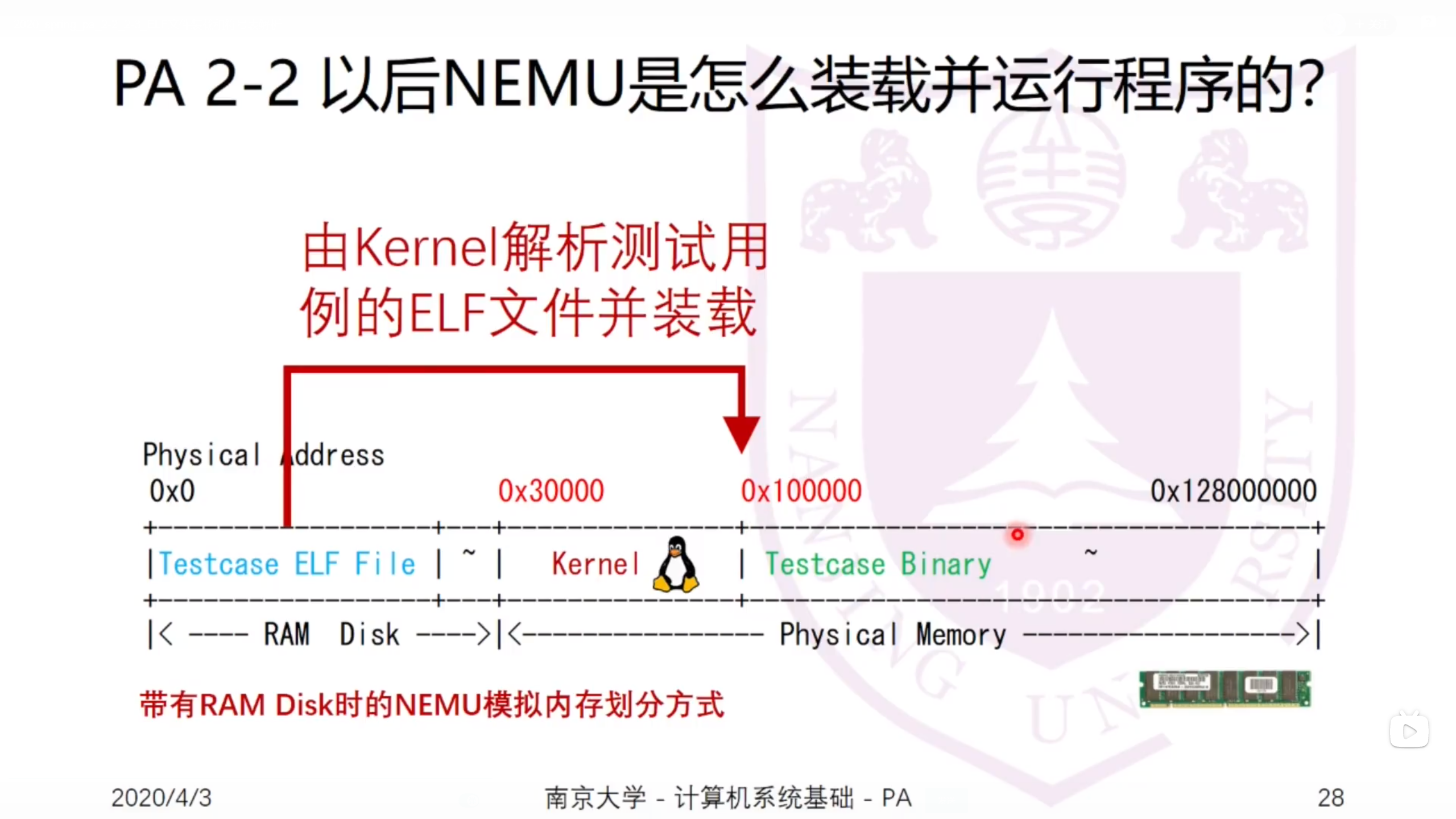
网上的一张图，有别的相关注释，留作记录：



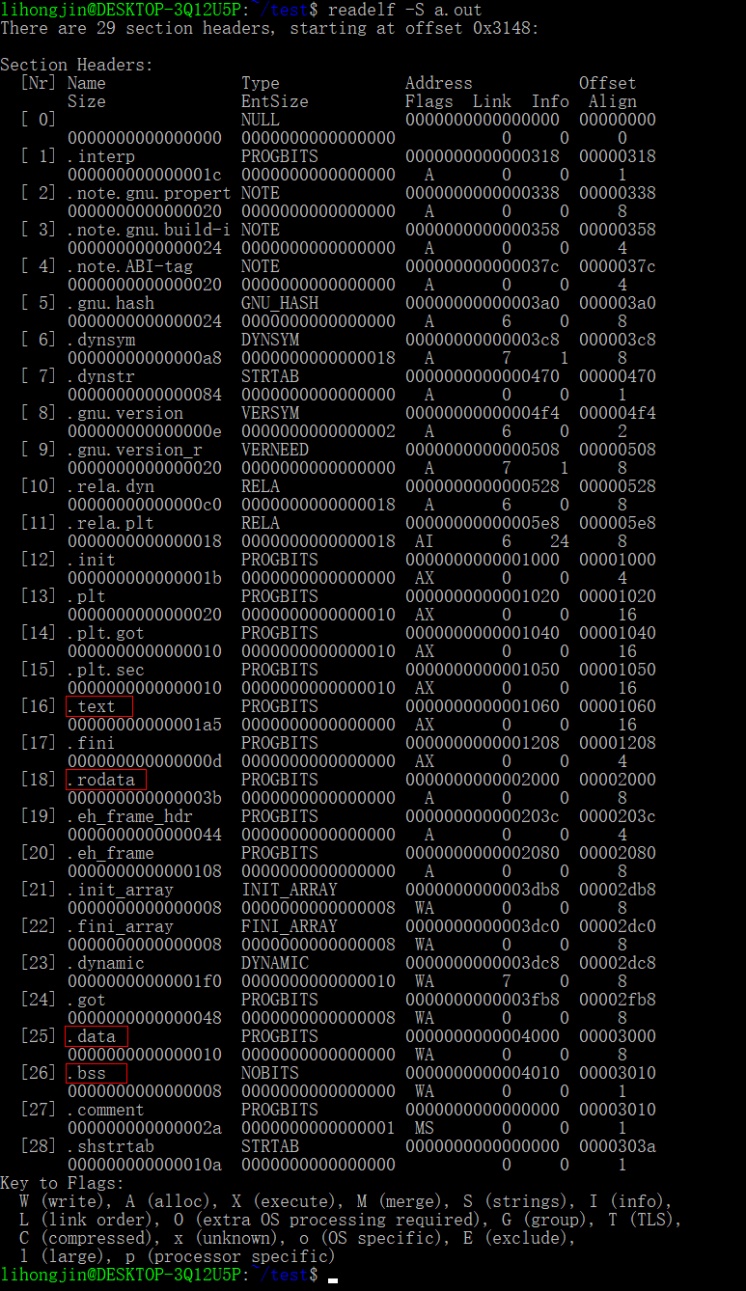
**readelf -l <elf文件>可以查询elf程序头表信息：**



MemSiz可能比FileSiz大，这一般是由于存在定义未初始化的全局变量(.bss)引入的，这样的话多出来的空前全都需要初始化为0。



**readelf -S <elf文件>可以查询elf程序头表信息：**



**2.3.1 相关工具**

* **readelf**

readelf命令用来显示一个或者多个elf格式的目标文件的信息，可以通过它的选项来控制显示哪些信息。

readelf <option(s)> elf-file(s)

**常用的选项：**

**-a --all 显示全部信息,等价于 -h -l -S -s -r -d -V -A -I.**

**-h --file-header 显示elf文件开始的文件头信息.**

**-l --program-headers 显示程序头（段头）信息(如果有的话)。 别称： --segments**

**-S --section-headers 显示节头信息(如果有的话)。 别称：--sections**

-g --section-groups 显示节组信息(如果有的话)。

-t --section-details 显示节的详细信息(-S的)。

**-s --syms --symbols 显示符号表段中的项（如果有的话）。**

-e --headers 显示全部头信息，等价于: -h -l -S

-n --notes 显示note段（内核注释）的信息。

-r --relocs 显示可重定位段的信息。

-u --unwind 显示unwind段信息。当前只支持IA64 ELF的unwind段信息。

-d --dynamic 显示动态段的信息。

-V --version-info 显示版本段的信息。

-A --arch-specific 显示CPU构架信息。

-D --use-dynamic 使用动态段中的符号表显示符号，而不是使用符号段。

-x <number or name> --hex-dump=<number or name> 以16进制方式显示指定段内内容。number指定段表中段的索引,或字符串指定文件中的段名。

-w[liaprmfFsoR] or --debug-dump [=line, =info, =abbrev, =pubnames, =aranges, =macro, =frames, =frames-interp, =str, =loc, =Ranges] 显示调试段中指定的内容。

-I --histogram 显示符号的时候，显示bucket list长度的柱状图。

-v --version 显示readelf的版本信息。

-H --help 显示readelf所支持的命令行选项。

-W --wide 宽行输出。

* **nm**

nm命令是linux下自带的特定文件分析工具，一般用来检查分析二进制文件、库文件、可执行文件中的符号表，返回二进制文件中各段的信息。

<https://zhuanlan.zhihu.com/p/363014233>

**常用选项：**

-A 或-o或 --print-file-name：打印出每个符号属于的文件

-a或--debug-syms：打印出所有符号，包括debug符号

-B：BSD码显示

-C或--demangle[=style]：对低级符号名称进行解码，C++文件需要添加

--no-demangle：不对低级符号名称进行解码，默认参数

-D 或--dynamic：显示动态符号而不显示普通符号，一般用于动态库

-f format或--format=format：显示的形式，默认为bsd，可选为sysv和posix

-g或--extern-only：仅显示外部符号

-h或--help：国际惯例，显示命令的帮助信息

-n或-v或--numeric-sort：显示的符号以地址排序，而不是名称排序

-p或--no-sort：不对显示内容进行排序

-P或--portability：使用POSIX.2标准

-V或--version：国际管理，查看版本

--defined-only：仅显示定义的符号，这个从英文翻译过来可能会有偏差，故贴上原文：

Display only defined symbols for each object file

**nm输出内容解析**

* 首先，前面那一串数字，指的就是地址，如果是二进制目标文件，指的就是逻辑地址，并非程序最后运行时的地址。
* 然后，我们发现，每一个条目前面还有一个字母，类似'U','B','D等等，其实这些符号代表的就是当前条目所对应的内存所在部分
* 最右边的就是对应的符号内容了

首要的需要讲解的就是第二点中字符所对应的含义，同样在还是在linux命令行下man nm指令可以得到：

A ：符号的值是绝对值，不会被更改

B或b ：未被初始化的全局数据，放在.bss段

D或d ：已经初始化的全局数据

G或g ：指被初始化的数据，特指small objects

I ：另一个符号的间接参考

N ：debugging 符号

p ：位于堆栈展开部分

R或r ：属于只读存储区

S或s ：指为初始化的全局数据，特指small objects

T或t ：代码段的数据，.test段

U ：符号未定义

W或w ：符号为弱符号，当系统有定义符号时，使用定义符号，当系统未定义符号且定义了弱符号时，使用弱符号。

？ ：unknown符号

* **objdump**

objdump命令是用查看目标文件或者可执行的目标文件的构成的gcc工具。可以用来反汇编代码。

objdump <option(s)> <file(s)>

**常用选项：**

**objdump -d <file(s)>: 将代码段反汇编；**

objdump -D <file(s)>: 将所有段反汇编；

**objdump -S <file(s)>: 将代码段反汇编的同时，将反汇编代码与源代码交替显示，编译时需要使用-g参数，即需要调试信息；**

objdump -C <file(s)>: 将C++符号名逆向解析

objdump -l <file(s)>: 反汇编代码中插入文件名和行号

objdump -j section <file(s)>: 仅反汇编指定的section

-T --dynamic-syms

显示文件的动态符号表入口，仅仅对动态目标文件意义，比如某些共享库。它显示的信息类似于 nm -D|–dynamic 显示的信息。

对于 arm 平台的反汇编：

sudo apt-get install gcc-arm-none-eabi

arm-linux-gnueabihf-objdump

arm-linux-gnueabi-objdump

* **pmap**

**查看进程的内存映像信息**

**2.4 工具**

**2.4.1 linux 串口工具**

***minicom***

minicom -s 修改设置，包括预置场景

打开串口后 ctrl+a z 打开帮助，敲z时不加ctrl

在帮助里按 o 可以进行端口配置

打开串口后 ctrl+a q 退出

使用功能菜单时按 Esc 基本都可以返回

对于mac，meta键默认是esc，但可以修改

***picocom*** minicom的精简版

**2.4.2 linux 调试工具**

* **GDB基本用法**

**基本指令：参考宋宝华的书**

编译需要添加参数-g

1. 启动

$ gdb <app>

$ (gdb) file <app>

attach:

$ sudo gdb <app> -p <pid>

$ (gdb) file <app>

$ (gdb) attach <pid >

**运行参数处理：**

1. 进入gdb之前设置：gdb --args <app> <paras>
2. 进入gdb之后设置：set args <paras>
3. 进入gdb后，运行时设置：r <paras>
4. --batch 在处理完所有用“-x”指定的命令文件后以状态 0 退出。 如果在命令文件中执行 GDB 命令时发生错误，则以非零状态退出。例如：

gdb -ex=r --batch --args <paras> 其中 r 是 run的简写

1. 使用command文件

进入gdb之前：

gdb --command=<commandfile.gdb> [--batch] --args <app> [<paras>]

进入gdb之后：

source <commandfile.gdb>

commandfile的编写按照命令行敲命令的顺序即可

注意：无论在哪里设置参数，只要设置完就可以直接使用r运行，默认带有参数，除非另作修改更新，才会使用新的参数

2. 列出代码

$(gdb) list <show number>

$(gdb) list <function>

$(gdb) list # 显示当前行后边的源程序

$(gdb) show listsize # 显示现在 list 显示的行数

$(gdb) set listsize <num> # 修改 list 显示的行数

$(gdb) tui enable # 使用独立的代码窗口

$(gdb) layout src # 使用独立的代码窗口

$(gdb) layout asm # 使用独立的代码窗口，汇编语言

**layout的使用：**

layout：用于分割窗口，可以一边查看代码，一边测试。主要有以下几种用法：

layout src：显示源代码窗口

layout asm：显示汇编窗口

layout regs：显示源代码/汇编和寄存器窗口

layout split：显示源代码和汇编窗口

layout next：显示下一个layout

layout prev：显示上一个layout

Ctrl + L：刷新窗口

Ctrl + x，再按1：单窗口模式，显示一个窗口

Ctrl + x，再按2：双窗口模式，显示两个窗口

Ctrl + x，再按a：回到传统模式，即退出layout，回到执行layout之前的调试窗口。

3. 调试控制

$(gdb) run # 运行，简写 r，后边可跟命令行参数，如果没有设置断点，不会停下。

$(gdb) start # 开始执行程序，在main 函数处会停下来

$(gdb) starti # 开始执行程序，在第一条(机器)指令处会停下来

$(gdb) continue [<ignore-cnt>]# 可简写为 c，恢复程序运行直到程序结束或到达下一个断点，ignore-cnt 表示忽略其后多少次断点

$(gdb) next # 下一步，可简写为 n

$(gdb) break # 中断，可简写为 b

$(gdb) where # 查看程序当前位置

$(gdb) print # 打印，可简写为 p

$(gdb) jump <linespec> # GDB 提供了乱序执行的功能，<inespee>可以是文件的行号，可以是file:line格式，也可以是fnum 这种偏移量格式，表示下一条运行语句从哪里开始。

$(gdb) signal <signal> # 使用singal 命令，可以产生一个信号量给被调试的程序，如中断信号 ctrl+C。于是，可以在程序运行的任意位置处设置断点，并在该断点处用 GDB 产生一个信号量，这种精确地在某处产生信号的方法非常有利于程序的调试。UINIX 的系统信号量通常为 1-15，因此 signals 的取值也在这个范围内。

$(gdb) return [<expr>] # 如果在函数中设置了调试断点，在断点后还有语向没有执行完，这时候我们可以使用 return 命令强制函数忽略还没有执行的语句并返回。

$(gdb) call <func> # 强制调用某函数，也可以使用p <func> 的形式调用函数

在运行前，可以设置如下的环境变量：

1. 程序运行参数：

$(gdb) set args 1 2 3 4 5

$(gdb) r <paras>

$(gdb) show args

1. 运行环境：

$(gdb) path <dir> 设定程序的运行路径

$(gdb) show paths 查看程序的运行路径

$(gdb) set environment varname[=value] # 设定环境变量

$(gdb) show environment[varname] # 查看环境变量

1. 工作目录：

$(gdb) cd <dir> 相当于 shell 的 cd 命令

$(pwd) pwd 显示当前所在目录

1. 程序的输入输出：

info terminal 用于显示程序用到的终端模式

在gdb中也可以使用重定向控制程序输出，如 run > outfile

用tty命令可以指定输入输出的终端设备，如 tty /dev/ttyS1

4. 单步调试：

1) next [<count>] # 可简写为 n，单步执行，不会进入函数

2) step [<count>] # 可简写为 s，单步执行，会进入函数内部

3) set step-mode

set step-mode on 用于打开step-mode 模式，这样在进行单步跟踪(step)时，若跨于某个没有调试信息的函数，程序的执行则会在该函数的第一条指令处停住，而不会跳过整个函数。这样我们可以查看该函数的机器指令

4) finish

运行程序，直到当前函数完成返回，并打印函数返回时的堆栈地址、返回值及参数值等信息

5) untile (缩写为 u) # 运行程序直到退出循环体

6) setpi (缩写为si) 和 nexti (缩写为ni)

stepi 和 nexiti 用于单步跟踪一条机器指令。比如，一条c程序代码有可能由数条机器指令完成，stepi 和 nexiti 可以单步执行机器指令，相反，setp和next是c语言级别的命令。另外，运行 display/i $pc 命令后，单步跟踪会在打出程序代码的同时打出机器指令，即汇编代码。

5. 断点调试：

1) break <func> # 进入函数时停住，c++ 可以使用 class::function 或 function(type,type)

2) break <linenum> # 在指定行号停住

3) break +offset / break -offset # 在当前行号的前面或后面的 offset 行停住

4) break filename:linenum # 在源文件flename 的 linenum 行处停住。

5) break filename: function # 在源文件 fliename 的 function 函数的人口处停住。

6) break \*address # 在程序运行的内存地址处停住。

7) break # break 命令没有参数时，表示在下一条指令处停住。

8) break …… if <condition>

…… 可以是上述的 break<linenum>、break +offset / break -offset 中的参数，condition表示条件，在条件成立时停住。比如在循环体中，可以设置 break if i = 100，表示 i 100时停住程序

9) info breakpoints[n]、info break[n] # 查看断点，n表示断点号

10) delete <num> # 删除断点

11) disable <num> # 去使能断点

12) enable <num> # 使能断点

13) commands 设置在给定断点被击中时要执行的命令，命令格式：

commands [break point idx]

cmd 1

cmd 2

end

如果不填 break point idx 默认使用最后一个中断点

6. 调试过程中的信息查询和处理

1）print[/<f>] <expr>

<expr> 是表达式，或者是变量，<f>是输出的格式，例如：/x 表示十六进制格式，当使用gdb的print查看程序运行时数据时，每个print都会被gdb记录下来。gdb会以$1,$2,$3这样的形式为每一个print命令编号。我们可以使用这个编号访问以前的表达式。如果要修改变量，可以使用类似 print x=4 的格式

print 的输出格式如下：

x：按十六进制格式显示变量

d：按十进制格式显示变量

u：按十六进制格式显示无符号整型

o：按八进制格式显示变量

t：按二进制格式显示变量

a：按十六进制格式显示变量

c：按宇符格式显示变量

f：按浮点数格式显示变量

2) display</fmt> <expr>

和 print 命令一样，display 命令也用于调试阶段查看某个变量或表达式的值，它们的区别是，使用 display 命令查看变量或表达式的值，每当程序暂停执行（例如单步执行）且作用域能达到时，GDB 调试器都会自动帮我们打印出来，而 print 命令则不会。

关闭则使用 disable/delete display <idx> 其中idx可以通过info display查询

3) watch

watch 一般用来观察某个表达式的值是否有了变换，如果有了变化，马上停止运行程序，使用方法如下：

watch <expr>：为表达式（变量）expr 设置一个观察点。一旦表达式值有变化时，马上停止程序运行。

rwatch <expr>：当表达式（变量）expr 被读时，停止程序运行。

awatch <expr>：当表达式（变量）的值被读或被写时，停止程序运行。

如果想观察内存的值变化可以按如下操作：

1. 得到内存的地址：print &val
2. watch \*<addr> // 观察一个byte

watch \*(int\*)<addr> // 观察int长度的byte

4) examine (缩写为 x) 查看内存地址中的值

格式：x/<n/f/u> <addr>

n:是正整数，表示需要显示的内存单元的个数，即从当前地址向后显示n个内存单元的内容，一个内存单元的大小由第三个参数u定义。

f:表示addr指向的内存内容的输出格式，s对应输出字符串，此处需特别注意输出整型数据的格式：

x:按十六进制格式显示变量。

d:按十进制格式显示变量。

u:按十六进制格式显示无符号整型。

o:按八进制格式显示变量。

t:按二进制格式显示变量。

a:按十六进制格式显示变量。

c:按字符格式显示变量。

f:按浮点数格式显示变量。

u:就是指以多少个字节作为一个内存单元-unit,默认为4。当然u还可以用被一些字符表示，如b=1 byte, h=2 bytes,w=4 bytes,g=8 bytes.

<addr>:表示内存地址。

整合这个命令的诠释：就是以addr为起始地址，返回n个单元的值，每个单元对应u个字节，输出格式是f。

如：x/ 3uh 0x54320表示：以地址0x54320为起始地址，返回3个单元的值，每个单元有两个字节，输出格式为无符号十六进制。

也就是说返回了3\*2=6个字节的数据，以十六进制输出，这6个字节的数据，每两个字节为一个单元输出，共输出3个单元。

5) examine 命令用于查看内存，而 set 命令用于修改内存。它的命令格式是“set\* 有类型的指针=value”

6) info 命令

info 命令可以用来在调试时查看奇存器、断点、观察点和信号等信息。要查看寄存器的值可以使用如下命令

info registers（查看除了浮点寄存器以外的寄存器）

info a11-registers（查看所有寄存器）

info registers ＜regname...> （查看所指定的寄存器）

要查看断点信息，可以使用如下命令：

info break

要列出当 前所设置的所有观察点，可使用如下命令：

info watchpoints

要查看有哪些信号正在被 GDB 检测，可使用如下命令：

info signals

info handle

也可以使用 info line 命令来查看源代码在内存中的地址。info line 后面可以跟行号、函数名、文件名:行号、文件名:函数名等各种形式，例如用下面的命令会打印出所指定的源码在运行时的内存地址：

info line tst.c: func

**查看进程(包括进程号)：**

**info inferiors**

**查看线程：**

**info threads**

**查看线程栈结构：**

**bt**

**打印所有线程堆栈：**

**thread apply all bt**

**切换线程：**

**thread n # n 表示第几个线程**

7) 历史命令

mkdir -p ~/.config/gdb

echo 'set history save on' >> ~/.config/gdb/gdbinit

或

# 如果不设置文件的话，会在当前路径下生成.gdb\_history文件

set history filename gdb.history

set history save on

set history size 1000

7. 查看帧/栈相关的信息

1）查看调用栈信息：

（1）backtrace: 显示程序的调用栈信息，可以用bt缩写

（2）backtrace n: 显示程序的调用栈信息，只显示栈顶n帧(frame)

（3）backtrace –n: 显示程序的调用栈信息，只显示栈底部n帧(frame)

（4）set backtrace limit n: 设置bt显示的最大帧层数

（5）where, info stack：都是bt的别名，功能一样

2）查看帧信息：

（1）frame n: 查看第n帧的信息， frame可以用f缩写

（2）frame addr: 查看pc地址为addr的帧的相关信息

（3）up [n]: 查看当前帧上面第n帧的信息，跳转后可以打印上一个函数的相关信息

注意这里的上指外层函数，不写n表示跳一层

（4）down [n]: 查看当前帧下面第n帧的信息，跳转后可以打印下一个函数的相关信息

3）查看更加详细的信息：

（1）info frame、info frame n或者info frame addr

（2）info args：查看当前帧中的参数

（3）info locals：查看当前帧中的局部变量

（4）info catch：查看当前帧中的异常处理器

8. disassemble <func> # 命令用于反汇编，可用它来查看当前执行时的源代码的机器码，实际上只是把目前内存中的指令冲刷出来。

* **嵌入式GDB远程调试**
* 调试主机需要安装gdb-multiarch，这里需要与嵌入式设备匹配，我是用udooer的开发板的时候发现gdb-multiarch可以使用。
* 嵌入式设备需要安装gdbserver
* 嵌入式设备与调试主机在同一网段
* 嵌入式平台作为服务端执行：

gdbserver <host\_ip>:<port> <app> 例如： gdbserver localhost:8888 test

* 调试主机作为客户端执行：

gdb-multiarch <app> 例如：gdb-multiarch test

(gdb) target remote <target\_ip>:<port>

例如：target remote 192.168.2.107:8888

* 之后就可以在上位机敲指令调试了
* 同样如果是使用系统中空闲的串口作为GDB调试器和GDBserver的底层通信手段的话各条指令如下：

gdbserver /dev/ttyS0 test 其中/dev/ttyS0是嵌入式平台的通信设备

gdb-multiarch <app>

(gdb) target remote /dev/ttyUSB0 其中/dev/ttyUSB0是上位机的通信设备

* **LLDB 基本用法**

lldb 与 gdb 使用方法大同小异，可以参考gdb使用，如果无法实现效果再细究是否有差异

lldb 与 gdb 的关系映射

<https://lldb.llvm.org/use/map.html>

lldb 常用命令：

<https://www.jianshu.com/p/81278ade3b80>

**环境相关**

(lldb) platform status # 打印基本状态

(lldb)

(lldb) platform list # 列举当前支持的平台，如果调试android，则执行 platform select remote-android

(lldb) platform connect connect://<deviceName>:<remotePort>

(lldb) platform settings -w <path: local or remote>

(lldb) file <fileName>

(lldb) r <input params>

$ lldb -s <debug.lldb> <app> # 相当于 gdb 的—command

$ lldb -- <args> # -- 相当于 gdb 的 --args

***远程设备执行：***

lldb-server p --server --listen "\*:<remotePort>"

**使用独立的代码窗口**

(lldb) gui

**运行控制**

run (r) - 启动目标程序，如果遇到断点则暂停

next (n) - 运行当前箭头指向行

step (s) - 进入下一条指令中的函数内部

continue (c) - 继续运行程序直到遇到断点。

thread backtrace (bt) - 显示程序的调用栈信息，可以用bt缩写

frame (f) - 默认显示当前栈的内容，可以通过 `frame arg` 进入特定的 frame（用作输出本地变量）

**断点相关**

(lldb) breakpoint list # 列出所有断点，可简写 br l

(lldb) b <funcName> # 添加断点

(lldb) b <fileNmae>:<line> # 添加断点

(lldb) br del <num> # 删除断点

(lldb) br dis <num> # 去使能断点

(lldb) br en <num> # 使能断点

**线程相关**

process interrupt # 暂停当时程序的运行

thread list # 列出当前线程

thread select 1 # 选择线程 1，可简写为 t 1

thread backtrace # 打印当前线程的调用战信息，可简写为 bt

thread backtrace all # 打印所有线程的调用栈信息

**变量相关**

* **嵌入式LLDB远程调试**

**上位机：**

$ lldb

(lldb) platform list # 列出当前支持的远程平台

(lldb) platform select remote-android # 选择嵌入式系统的平台

(lldb) platform connect connect://<device name>:8888 # 设备名可以用adb devices 查询

(lldb) platform settings -w /usr/local/bin # 修改工作路径，使用file加载文件时用的是该路径

(lldb) file <local host> # 加载work路径下的文件，可以通过paltform status 查看work路径，被加载的文件会在运行时推到嵌入式系统的server工作路径

(lldb) target

***attach 到远程进程***

(lldb) platform process list # 列出远程系统上的进程

(lldb) attach 68639 # attach 到进程上

**嵌入式系统：**

lldb-server 基本使用方法，（server可以启动一次循环调试使用）：

Usage:

lldb-server v[ersion]

lldb-server g[dbserver] [options]

lldb-server p[latform] [options]

当lldb工作再 platform 模式时：

Usage:

lldb-server p [--log-file log-file-name] [--log-channels log-channel-list] [--port-file port-file-path] --server --listen port

例如：lldb-server p --server --listen "\*:8888"

注意：这里需要在一个可以运行文件的路径下启动server，例如在 /vendor/bin 下执行就可以正常调试，但是如果在/sdcard下执行就不能正常调试，会返回process launch failed: 'a' packet returned an error: 8错误

使用demo：

lldb-server p --server --listen "\*:8888" # 在可执行文件的路径下

platform select remote-android

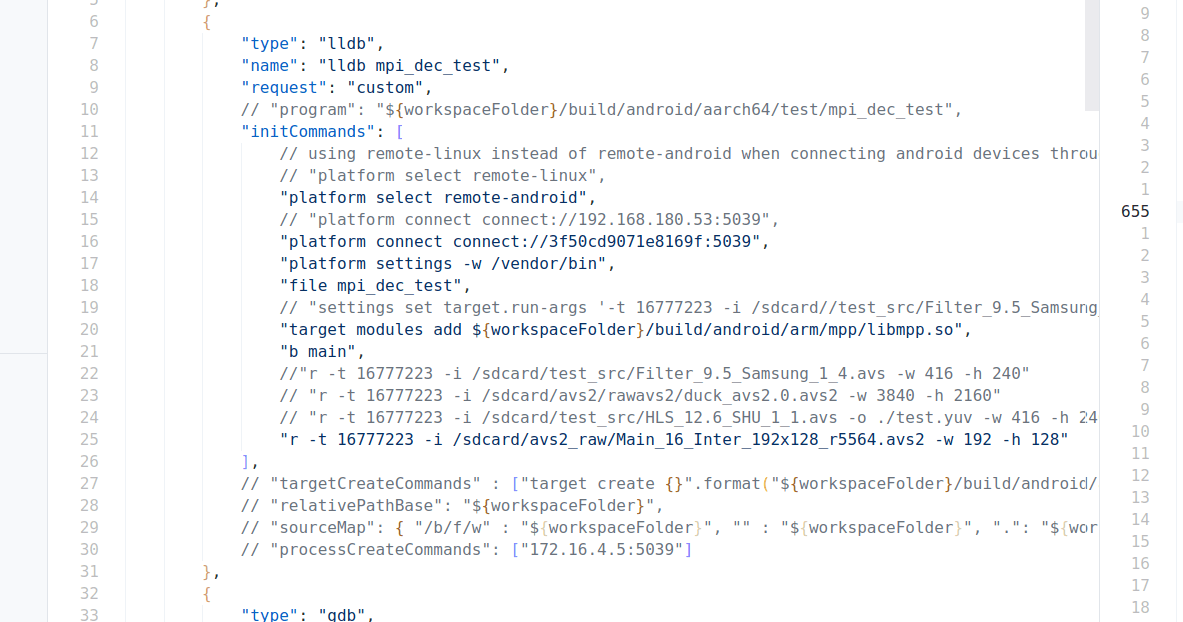
platform connect connect://0:8888

platform settings -w /vendor/bin

file mpi\_dec\_test

r -i output.h264 -t 7

海强的 vscode 配置



* **ADB调试基本指令**
* 连接设备的adb接口
* 电脑上安装adb工具
* 执行 adb devices 可以看到连接的设备
* Android系统的文件系统可能为只读系统，无法放入文件，需要执行一下命令：

    adb root

    adb remount

* 然后在Android的路径 /vendor 下就是可写的了
* **文件传输：**
* 调试机 --> Android： adb push <localFile> <remoteDir>
* Android --> 调试机： adb pull <remoteFile> <localDir>
* **端口相关**

#将PC的<hostPort>端口收到的数据，转发到手机中<remotePort>端口。

adb forward tcp:<hostPort> tcp:<remotePort>

例如：adb forward tcp:8887 tcp:8888

#查看端口转发

adb forward --list

#删除建立的转发端口

adb forward --remove tcp:<hostPort>

* **可能会存在Linux系统无法检测到 adb 设备的情况，可以如下操作：**

$ sudo adb kill-server

$ sudo adb devices

之后直接用 adb devices 就可以检测到设备了

* **adb 网络模式**

切换到网络模式：

adb tcpip <localPort>

adb connect <remoteIp>:<remotePort>

直接执行 adb connect <remoteIP> 也可以

切换回usb模式：

adb usb

例如：

adb tcpip 5555

adb connect 192.168.0.101:5555

* **ADB+gdb调试(命令行)**
* 嵌入式设备执行： gdbserver :<port> <app> <param>

例如： gdbserver :8888 mpi\_dec\_test -i /sdcard/0.origin.h264

其中：<param>在嵌入式设备指定就可以了，调试主机可以不指定，软件实际执行是在嵌入式设备

* 调试主机使用adb做端口映射：adb forward tcp:<hostPort> tcp:<remotePort>

例如：adb forward tcp:8887 tcp:8888

* 调试主机启动gdb开始调试：

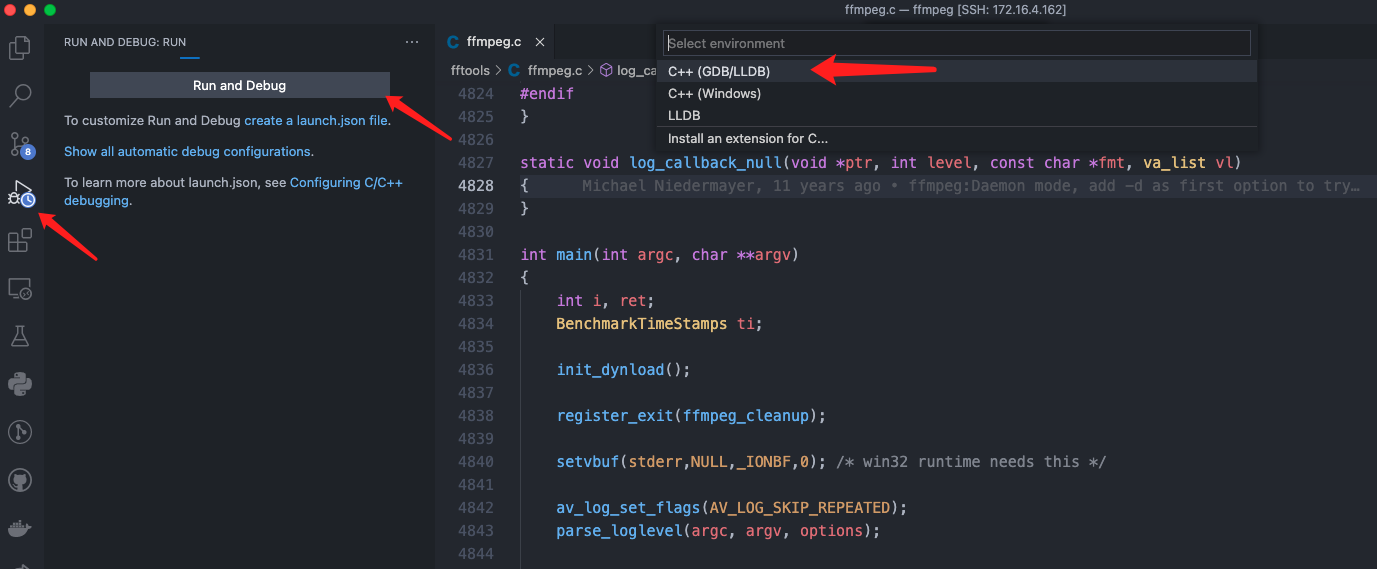
1. gdb <app>
2. (gdb) target remote 127.0.0.1:<hostPort> # 这里指向adb指定的主机端口

例如： target remote 127.0.0.1:8887

注意：这里的gdb应该使用nkd里的gdb，

例如：android-ndkbuilt/linux-x86\_64/bin/gdb

* **vscode 本地调试**



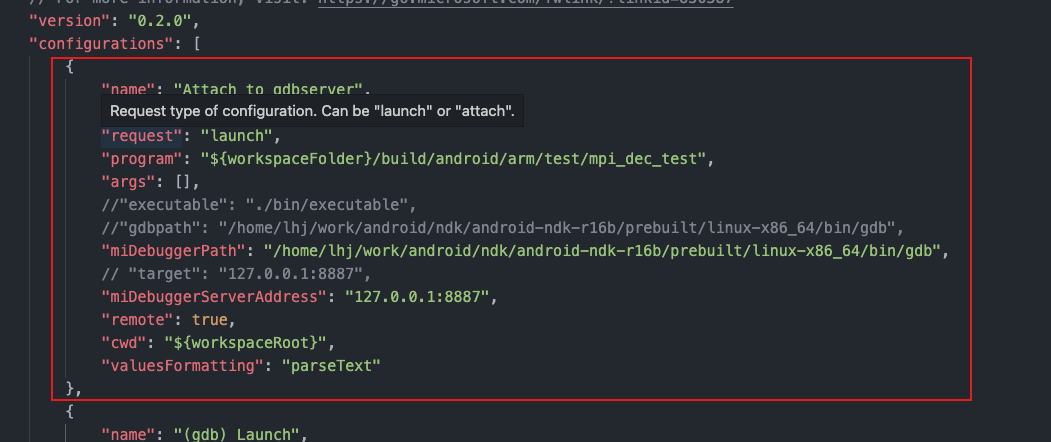




* **vscode 调试+gdb 远程调试（adb环境）**

**launch 模式：**

1. 在嵌入式平台上启动gdbserver，方法与**ADB+gdb调试(命令行)** 一致
2. 在调试主机使用adb forward做端口映射，方法与**ADB+gdb调试(命令行)** 一致
3. 在调试主机加入如下配置，然后开始调试



|  |
| --- |
| {  "name": "Attach to gdbserver",  "type": "cppdbg",  "request": "launch",  "program": "${workspaceFolder}/build/android/arm/test/mpi\_dec\_test",  "args": [],  "miDebuggerPath":"/home/lhj/work/android/ndk/android-ndk-r16b/prebuilt/linux-x86\_64/bin/gdb",  "miDebuggerServerAddress": "127.0.0.1:8887",  "remote": true,  "cwd": "${workspaceRoot}",  "valuesFormatting": "parseText"  }, |

**attach 模式：**

只需要在嵌入式启动的时候将指令改为：

gdbserver :<port> --attach <pid>

例如：

rk3566\_r:/ # ps -elf | grep mediaserver

media 357 1 0 04:38:15 ? 00:00:01 mediaserver

root 2324 2321 0 07:02:00 pts/3 00:00:00 grep mediaserver

rk3566\_r:/ # gdbserver :8888 –attach 357

* **top 使用技巧**

top -H 按照线程查看，或者启动top之后按 H 也可以

top 按照进程查看

top -H | grep <> 按照筛选项查看

参考博客： <http://c.biancheng.net/view/1065.html>

**选项：**

-d 秒数：指定 top 命令每隔几秒更新。默认是 3 秒；

-b：使用批处理模式输出。一般和"-n"选项合用，用于把 top 命令重定向到文件中；

-n 次数：指定 top 命令执行的次数。一般和"-"选项合用；

-p 进程PID：仅查看指定 ID 的进程；

-s：使 top 命令在安全模式中运行，避免在交互模式中出现错误；

-u 用户名：只监听某个用户的进程；

在 top 命令的显示窗口中，还可以使用如下按键，进行一下交互操作：

? 或 h：显示交互模式的帮助；

P：按照 CPU 的使用率排序，默认就是此选项；

M：按照内存的使用率排序；

N：按照 PID 排序；

T：按照 CPU 的累积运算时间排序，也就是按照 TIME+ 项排序；

k：按照 PID 给予某个进程一个信号。一般用于中止某个进程，信号 9 是强制中止的信号；

r：按照 PID 给某个进程重设优先级（Nice）值；

q：退出 top 命令；

1：将cpu核心展开显示，而不是统计显示

H：切换到线程状态，再按切回到任务状态

F：选择指定的列显示

U：筛选用户

V：按照树状图显示

**第一部分的作用**

第一行为任务队列信息

12:26:46 系统当前时间

up 1 day, 13:32 系统的运行时间.本机己经运行 1 天 13 小时 32 分钟

2 users 当前登录了两个用户

load average: 0.00,0.00，0.00 系统在之前 1 分钟、5 分钟、15 分钟的平均负载。如果 CPU 是单核的，则这个数值超过 1 就是高负载：如果 CPU 是四核的，则这个数值超过 4 就是高负载 （这个平均负载完全是依据个人经验来进行判断的，一般认为不应该超过服务器 CPU 的核数）

第二行为进程信息

Tasks: 95 total 系统中的进程总数

1 running 正在运行的进程数

94 sleeping 睡眠的进程数

0 stopped 正在停止的进程数

0 zombie 僵尸进程数。如果不是 0，则需要手工检查僵尸进程

第三行为 CPU 信息

Cpu(s): 0.1 %us 用户模式占用的 CPU 百分比

0.1%sy 系统模式占用的 CPU 百分比

0.0%ni 改变过优先级的用户进程占用的 CPU 百分比

99.7%id 空闲 CPU 占用的 CPU 百分比

0.1%wa 等待输入/输出的进程占用的 CPU 百分比

0.0%hi 硬中断请求服务占用的 CPU 百分比

0.1%si 软中断请求服务占用的 CPU 百分比

0.0%st st（steal time）意为虚拟时间百分比，就是当有虚拟机时，虚拟 CPU 等待实际 CPU 的时间百分比

第四行为物理内存信息

Mem: 625344k total 物理内存的总量，单位为KB

571504k used 己经使用的物理内存数量

53840k&ee 空闲的物理内存数量。我们使用的是虚拟机，共分配了 628MB内存，所以只有53MB的空闲内存

65800k buffers 作为缓冲的内存数量

第五行为交换分区（swap）信息

Swap: 524280k total 交换分区（虚拟内存）的总大小

Ok used 已经使用的交换分区的大小

524280k free 空闲交换分区的大小

409280k cached 作为缓存的交换分区的大小

我们还要解释一下缓冲（buffer）和缓存（cache）的区别：

缓存（cache）是在读取硬盘中的数据时，把最常用的数据保存在内存的缓存区中，再次读取该数据时，就不去硬盘中读取了，而在缓存中读取。

缓冲（buffer）是在向硬盘写入数据时，先把数据放入缓冲区,然后再一起向硬盘写入，把分散的写操作集中进行，减少磁盘碎片和硬盘的反复寻道，从而提高系统性能。

简单来说，缓存（cache）是用来加速数据从硬盘中"读取"的，而缓冲（buffer）是用来加速数据"写入"硬盘的。

**第二部分的作用**

再来看 top 命令的第二部分输出，主要是系统进程信息，各个字段的含义如下：

PID：进程的 ID。

USER：该进程所属的用户。

PR：优先级，数值越小优先级越高。

NI：优先级，数值越小、优先级越高。

VIRT：该进程使用的虚拟内存的大小，单位为 KB。

RES：该进程使用的物理内存的大小，单位为 KB。

SHR：共享内存大小，单位为 KB。

S：进程状态。

%CPU：该进程占用 CPU 的百分比。

%MEM：该进程占用内存的百分比。

TIME+：该进程共占用的 CPU 时间。

COMMAND：进程的命令名。

Linux系统可以使用 htop ，与 top 功能类似

* **addr2line工具**

addr2line工具是一个可以将指令的地址和可执行映像转换为文件名、函数名和源代码行数的工具。这在内核执行过程中出现崩溃时，可用于快速定位出出错的位置，进而找出代码的bug。

**常用参数：**

|  |  |
| --- | --- |
| -a | 在函数名、文件和行号信息之前，显示地址，以十六进制形式 |
| -b | 指定目标文件的格式为bfdname |
| -e | 指定需要转换地址的可执行文件名 |
| -j | 给出的地址代表指定section的偏移，而非绝对地址 |
| -C | 将低级别的符号名解码为用户级别的名字 |
| -f | 在显示文件名、行号输出信息的同时显示函数名信息 |
| -p | 使得该函数的输出信息更加人性化：每一个地址的信息占一行 |

示例：addr2line -f -e <elfFile> <Num>

**使用方法**

1. 编译可执行文件时要注意添加-g参数，保留调试信息
2. 查找系统信息，然后定位代码

dmesg

|  |
| --- |
| [150100.451504] traps: test[7593] trap divide error ip:**4005f5** sp:7ffeebd4ba70 error:0 in test[400000+1000] |

这条信息里，ip（指令指针寄存器）字段后面的数字就是test程序出错时程序执行的位置。使用addr2line就可以将4005f5定位到代码的位置：

addr2line 4005f5 -e test -f -s -C

**原理**

addr2line如何找到的这一行呢。在可执行程序中都包含有调试信息（所以编译的时候需要加-g选项），其中很重要的一份数据就是源程序的行号和编译后的机器代码之间的对应关系Line Number Table。Line Number Table存储在可执行程序的.debug\_line域。

使用如下命令

readelf -w test | grep "advance Address"

|  |
| --- |
| [0x000000ca] Special opcode 7: advance Address by 0 to 0x4005e7 and Line by 2 to 3  [0x000000cb] Special opcode 146: advance Address by 10 to 0x4005f1 and Line by 1 to 4  [0x000000cc] Special opcode 104: advance Address by 7 to 0x4005f8 and Line by 1 to 5  [0x000000cd] Special opcode 36: advance Address by 2 to 0x4005fa and Line by 3 to 8  [0x000000ce] Special opcode 118: advance Address by 8 to 0x400602 and Line by 1 to 9  [0x000000cf] Special opcode 146: advance Address by 10 to 0x40060c and Line by 1 to 10  [0x000000d0] Special opcode 104: advance Address by 7 to 0x400613 and Line by 1 to 11  [0x000000d1] Special opcode 104: advance Address by 7 to 0x40061a and Line by 1 to 12  [0x000000d3] Special opcode 20: advance Address by 1 to 0x40062c and Line by 1 to 13  [0x000000d5] Special opcode 132: advance Address by 9 to 0x400646 and Line by 1 to 14  [0x000000d6] Special opcode 76: advance Address by 5 to 0x40064b and Line by 1 to 15 |

观察第二行和第三行，源代码的第4行的指令起始地址是0x4005f1， 第5行的起始地址是0x4005f8，可以知道0x4005f5位置的指令是属于第4行代码的。

* **hexdump**

hexdump是Linux下的一个二进制文件查看工具，它可以将二进制文件转换为ASCII、八进制、十进制、十六进制格式进行查看。

hexdump: [-bcCdovx] [-e fmt] [-f fmt\_file] [-n length] [-s skip] [file ...]

此命令参数是Red Hat Enterprise Linux Server release 5.7下hexdump命令参数，不同版本Linux的hexdump命令参数有可能不同。

|  |  |  |
| --- | --- | --- |
| 参数 | 长参数 | 描叙 |
| -b |  | 每个字节显示为8进制。一行共16个字节，一行开始以十六进制显示偏移值 |
| -c |  | 每个字节显示为ASCII字符 |
| -C |  | 每个字节显示为16进制和相应的ASCII字符 |
| -d |  | 两个字节显示为10进制 |
| -e |  | 格式化输出 |
| -f |  | Specify a file that contains one or more newline separated format strings.  Empty lines and lines whose first non-blank character is a hash mark (#) are ignored. |
| -n |  | 只格式前n个长度的字符 |
| -o |  | 两个字节显示为8进制 |
| -s |  | 从偏移量开始输出 |
| -v |  | The -v option causes hexdump to display all input data.  Without the -v option, any number of groups of output lines, which would be identical to the immediately preceding group of output lines |
| -x |  | 双字节十六进制显示 |

* **nc**

nc [-hlnruz][-g<网关...>][-G<指向器数目>][-i<延迟秒数>][-o<输出文件>][-p<通信端口>][-s<来源位址>][-v...][-w<超时秒数>][主机名称][通信端口...]

**使用示例：**

nc <ip> <port> 连接指定IP和端口

nc -l <port> 监听本地端口

-u: 使用udp协议传输，不带该参数默认使用tcp协议

传输文件：

nc <ip> <port> < <fileName>

nc -l <port> > <fileName>

* **ldd**

ldd命令用于打印可执行文件或者库文件所依赖的共享库列表。

ldd(选项)(参数)

参数：

--version：打印指令版本号；

-v：详细信息模式，打印所有相关信息；

-u：打印未使用的直接依赖；

-d：执行重定位和报告任何丢失的对象；

-r：执行数据对象和函数的重定位，并且报告任何丢失的对象和函数；

--help：显示帮助信息。

* **ar**

ar命令可以用来创建、修改库，也可以从库中提出单个模块。

* **netstat**

netstat -atulnp会显示所有端口和所有对应的程序，用grep管道可以过滤出想要的字段

-a ：all，表示列出所有的连接，服务监听，Socket资料

-t ：tcp，列出tcp协议的服务

-u ：udp，列出udp协议的服务

-n ：port number， 用端口号来显示

-l ：listening，列出当前监听服务

-p ：program，列出服务程序的PID

* **ssh 隧道**

**正向隧道**

client --> server

localssh <-> remotessh

ssh -N -f [-g] -L <locIP>:<locPort>:<remoteIP>:<remotePort> <remoteUserName>@<remoteIP> -p <sshPort>

其中：

-N: 表示不登陆 remote 设备

-f: 表示后台运行，终端退出也在后台运行

-g: 开启网关功能，开启该选项可以监听当前主机所有IP的 <locPort>

<locIP> ： locIP可以缺省，此时默认 127.0.0.1，但这种情况下只能在本机访问 <locPort> 端口，来访问远程主机

如果想其他设备访问 本机，进而转发到 远程主机，则 <locIP> 需要改为本机在网络中的IP

**反向隧道**

场景：

在 client 无法访问到 server 的 IP 时，但是 server 可以访问到 client时，可能需要从 server 建立反向隧道。

典型的情况时有两台公网IP的主机，一台在 server 的环境中（IPA），另一台在其他环境中（IPB），server可以访问到IPA和IPB，但是IPA没有权限，IPB有权限，反之，IPB无法访问 server，此时需要建立反向隧道

client --> server

remotessh <-> localssh

ssh -N -f -R <remoteIP>:<remotePort>:<locIP>:<locPort> <remoteUserName>@<remoteIP> -p <sshPort>

其中：

<remoteIP>: 可以缺省不写，因为无论写不写都只能监听 remote 主机的 127.0.0.1 即：remote 主机不能被当作跳板，即便开启 -g 也不行

* **iperf3**

（1）-s,--server：iperf服务器模式，默认启动的监听端口为5201，eg：iperf -s

（2）-c,--client host：iperf客户端模式，host是server端地址，eg：iperf -c 222.35.11.23

（3）-i，--interval：指定每次报告之间的时间间隔，单位为秒，eg：iperf3 -c 192.168.12.168 -i 2

（4）-p，--port：指定服务器端监听的端口或客户端所连接的端口，默认是5001端口。

（5）-u，--udp：表示采用UDP协议发送报文，不带该参数表示采用TCP协议。

（6）-l，--len：设置读写缓冲区的长度，单位为 Byte。TCP方式默认为8KB，UDP方式默认为1470字节。通常测试 PPS 的时候该值为16，测试BPS时该值为1400。

（7）-b，--bandwidth [K|M|G]：指定UDP模式使用的带宽，单位bits/sec，默认值是1 Mbit/sec。

（8）-t，--time：指定数据传输的总时间，即在指定的时间内，重复发送指定长度的数据包。默认10秒。

（9）-A：CPU亲和性，可以将具体的iperf3进程绑定对应编号的逻辑CPU，避免iperf进程在不同的CPU间调度。

举例：

#iperf3 -s -p <serPort> -i 1

#iperf3 -c <serIP> -p <serPort> -i 1 -t 60

Interval表示时间间隔。

Transfer表示时间间隔里面转输的数据量。

Bandwidth是时间间隔里的传输速率。

* **网络问题总结**

**软件**

断连问题ping,telnet,nc,websocat，postman

端口问题telnet,netstat,nmap

流量问题nethogs

udp问题iptraf

带宽问题iperf3,speedtest

DNS问题ping,tcping,dig,nslookup

路由问题traceroute,route

延时问题ping,tcping

运营商问题:切换不同的代理出口

抓包问题：tcpdump，wireshark，fiddler

**硬件**

替换法，替换稳定的器件或者稳定的通路

验证原先的通路是否有问题

ifconfig

ethtool

nmcli

* **perf （Linux性能分析工具）**

<https://blog.csdn.net/cyq6239075/article/details/104371328>

Perf包含22种子工具的工具集，以下是最常用的5种：

Perf-list：用来查看perf所支持的性能事件，有软件的也有硬件的。

perf-stat：用于分析指定程序的性能概况。

perf-top：对于一个指定的性能事件(默认是CPU周期)，显示消耗最多的函数或指令。

perf-record：收集采样信息，并将其记录在数据文件中。随后可以通过其它工具(perf-report)对数据文件进行分析，结果类似于perf-top的。

perf-report：读取perf record创建的数据文件，并给出热点分析结果。

Tips:

任务调度追踪

sudo perf record -e context-switches -ag

sudo perf report -n --stdio -f

生成timechart

sudo perf sched record -a

sudo perf timechart

* **火焰图**

<https://zhuanlan.zhihu.com/p/54276509>

<https://blog.csdn.net/u013919153/article/details/110559888>

<https://blog.csdn.net/tugouxp/article/details/120165100>

**第一步：使用perf采样**

**方法一：**直接使用perf启动服务

# sudo perf record -g ls

--------------------------------

**方法二：**挂接到已启动的进程

*# 使用PID监控程序*

# sudo perf record -e cpu-clock -g -p pid

*# perf record 表示采集系统事件*

*# 没有使用 -e 指定采集事件, 则默认采集 cycles(即 CPU clock 周期)*

*# -F 99 表示每秒 99 次*

*# -p 13204 是进程号, 即对哪个进程进行分析*

*# -g 表示记录调用栈*

*# sleep 30 则是持续 30 秒*

*# -F 指定采样频率为 99Hz(每秒99次), 如果 99次 都返回同一个函数名, 那就说明 CPU 这一秒钟都在执行同一个函数, 可能存在性能问题.*

*# 如果svg图出现unknown函数，使用如下*

# sudo perf record -e cpu-clock --call-graph dwarf -p pid

*# 使用程序名监控程序*

# sudo perf record -e cpu-clock -g -p `pgrep your\_program`

------------------------------------

使用ctrl+c中断perf进程，或者在命令最后加上参数 -- sleep n (n秒后停止-- 和sleep中间有空格)

perf record表示记录到文件，perf top直接会显示到界面

如果record之后想直接输出结果，使用perf report即可:

sudo perf report

或

sudo perf report -n --stdio // 树状图

**第二步：生成火焰图**

*从github下载分析脚本*

git clone https://github.com/brendangregg/FlameGraph.git

*用 perf script 工具对 perf.data 进行解析,生成折叠后的调用栈*

sudo perf script -i perf.data &> perf.unfold

*用 stackcollapse-perf.pl 将 perf 解析出的内容 perf.unfold 中的符号进行折叠*

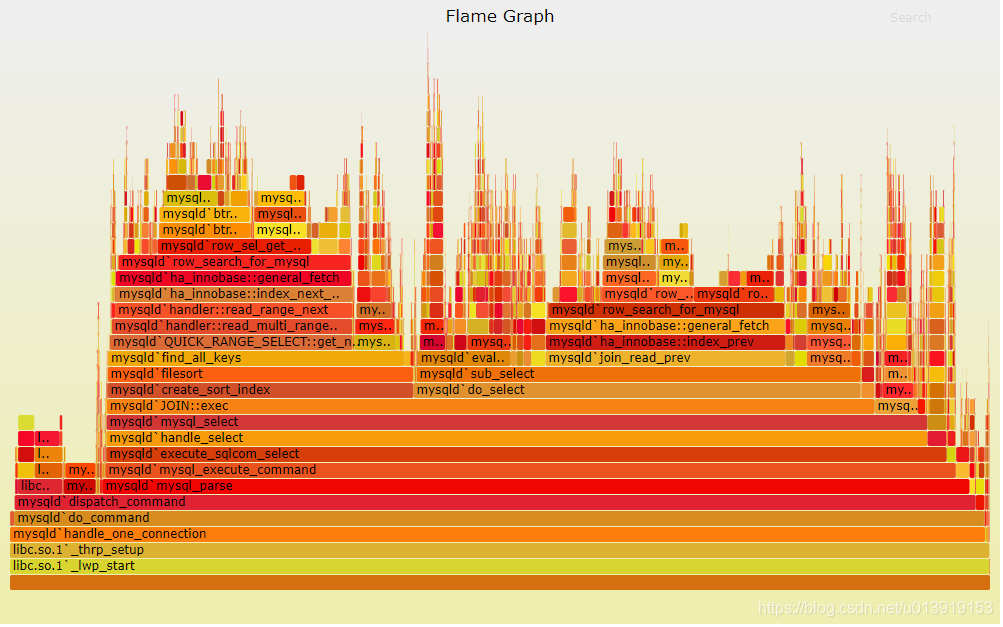
./FlameGraph/stackcollapse-perf.pl perf.unfold &> perf.folded

*生成 svg 图*

./FlameGraph/flamegraph.pl perf.folded > perf.svg

*可以利用shell 管道将上面三条命令合为一条*

perf script | FlameGraph/stackcollapse-perf.pl | FlameGraph/flamegraph.pl > process.svg



y 轴表示调用栈，每一层都是一个函数。调用栈越深，火焰就越高，顶部就是正在执行的函数，下方都是它的父函数。

x 轴表示抽样数，如果一个函数在 x 轴占据的宽度越宽，就表示它被抽到的次数多，即执行的时间长。注意，x 轴不代表时间，而是所有的调用栈合并后，按字母顺序排列的。

火焰图就是看顶层的哪个函数占据的宽度最大。只要有"平顶"（plateaus），就表示该函数可能存在性能问题。

颜色没有特殊含义，因为火焰图表示的是 CPU 的繁忙程度，所以一般选择暖色调。

* **kernelshark分析系统调度情况**

sudo trace-cmd record -e 'sched\_wakeup\*' -e sched\_switch -e 'sched\_migrate\*'

kernelshark trace.dat

* **查看程序搜索库的路径**

LD\_DEBUG1.LD\_DEBUG使用方法如下:

LD\_DEBUG=libs ./app

LD\_DEBUG=libs help 查看命令使用方法

**2.5 交叉编译环境**

**2.5.1 交叉编译器**

参考博客： https://blog.csdn.net/jpy1391/article/details/113798059

[交叉编译](https://so.csdn.net/so/search?q=%E4%BA%A4%E5%8F%89%E7%BC%96%E8%AF%91&spm=1001.2101.3001.7020)就是在一种平台上编译出能运行在体系结构不同的另一种平台上的程序，比如在PC平台（X86 CPU）上编译出能运行在以ARM为内核的CPU平台上的程序，编译得到的程序在X86 CPU平台上是不能运行的，必须放到ARM CPU平台上才能运行，虽然两个平台用的都是Linux系统。 交叉编译工具链是一个由编译器、连接器和解释器组成的综合开发环境，交叉编译工具链主要由binutils、gcc和glibc三个部分组成。有时出于减小 libc 库大小的考虑，也可以用别的 c 库来代替 glibc，例如 uClibc、dietlibc 和 newlib。

**分类和说明**

从授权上，分为免费授权版和付费授权版。

免费版目前有三大主流工具商提供，第一是GNU（提供源码，自行编译制作），第二是 Codesourcery，第三是Linora。

收费版有[ARM](https://so.csdn.net/so/search?q=ARM&spm=1001.2101.3001.7020)原厂提供的armcc、IAR提供的编译器等等，因为这些价格都比较昂贵，不适合学习用户使用，所以不做讲述。

* **arm-none-linux-gnueabi-gcc**：是 Codesourcery 公司（目前已经被Mentor收购）基于GCC推出的的ARM交叉编译工具。可用于交叉编译ARM（32位）系统中所有环节的代码，包括裸机程序、u-boot、Linux kernel、filesystem和App应用程序。
* **arm-linux-gnueabihf-gcc：**是由 Linaro 公司基于GCC推出的的ARM交叉编译工具。可用于交叉编译ARM（32位）系统中所有环节的代码，包括裸机程序、u-boot、Linux kernel、filesystem和App应用程序。
* **aarch64-linux-gnu-gcc：**是由 Linaro 公司基于GCC推出的的ARM交叉编译工具。可用于交叉编译ARMv8 64位目标中的裸机程序、u-boot、Linux kernel、filesystem和App应用程序。
* **arm-none-elf-gcc：**是 Codesourcery 公司（目前已经被Mentor收购）基于GCC推出的的ARM交叉编译工具。可用于交叉编译ARM MCU（32位）芯片，如ARM7、ARM9、Cortex-M/R芯片程序。
* **arm-none-eabi-gcc：**是 GNU 推出的的ARM交叉编译工具。可用于交叉编译ARM MCU（32位）芯片，如ARM7、ARM9、Cortex-M/R芯片程序。

**命名规则**

交叉编译工具链的命名规则为：arch [-vendor] [-os] [-(gnu)eabi]

arch – 体系架构，如ARM，MIPS（通过交叉编译工具生成的可执行文件或系统镜像的运行平台或环境）  
vendor – 工具链提供商  
os – 目标操作系统（host主要操作平台，也就是编译时的系统）  
eabi – 嵌入式应用二进制接口（Embedded Application Binary Interface）  
根据对操作系统的支持与否，ARM GCC可分为支持和不支持操作系统，如

arm-none-eabi：这个是没有操作系统的，自然不可能支持那些跟操作系统关系密切的函数，比如fork(2)。他使用的是newlib这个专用于嵌入式系统的C库。

arm-none-linux-eabi：用于Linux的，使用Glibc

**实例**  
**1、arm-none-eabi-gcc**

（ARM architecture，no vendor，not target an operating system，complies with the ARM EABI）  
用于编译 ARM 架构的裸机系统（包括 ARM Linux 的 boot、kernel，不适用编译 Linux 应用 Application），一般适合 ARM7、Cortex-M 和 Cortex-R 内核的芯片使用，所以不支持那些跟操作系统关系密切的函数，比如fork(2)，他使用的是 newlib 这个专用于嵌入式系统的C库。

**2、arm-none-linux-gnueabi-gcc**

(ARM architecture, no vendor, creates binaries that run on the Linux operating system, and uses the GNU EABI)

主要用于基于ARM架构的Linux系统，可用于编译 ARM 架构的 u-boot、Linux内核、linux应用等。arm-none-linux-gnueabi基于GCC，使用Glibc库，经过 Codesourcery 公司优化过推出的[编译器](https://so.csdn.net/so/search?q=%E7%BC%96%E8%AF%91%E5%99%A8&spm=1001.2101.3001.7020)。arm-none-linux-gnueabi-xxx 交叉编译工具的浮点运算非常优秀。一般ARM9、ARM11、Cortex-A 内核，带有 Linux 操作系统的会用到。

**3、arm-eabi-gcc**

Android ARM 编译器。

**4、armcc**

ARM 公司推出的编译工具，功能和 arm-none-eabi 类似，可以编译裸机程序（u-boot、kernel），但是不能编译 Linux 应用程序。armcc一般和ARM开发工具一起，Keil MDK、ADS、RVDS和DS-5中的编译器都是armcc，所以 armcc 编译器都是收费的（爱国版除外，呵呵~~）。

**5、arm-none-uclinuxeabi-gcc 和 arm-none-symbianelf-gcc**

arm-none-uclinuxeabi 用于uCLinux，使用Glibc。

arm-none-symbianelf 用于symbian，没用过，不知道C库是什么 。

**Codesourcery**  
Codesourcery推出的产品叫Sourcery G++ Lite Edition，其中基于command-line的编译器是免费的，在官网上可以下载，而其中包含的IDE和debug 工具是收费的，当然也有30天试用版本的。

目前CodeSourcery已经由明导国际(Mentor Graphics)收购，所以原本的网站风格已经全部变为 Mentor 样式，但是 Sourcery G++ Lite Edition 同样可以注册后免费下载。

Codesourcery一直是在做ARM目标 GCC 的开发和优化，它的ARM GCC在目前在市场上非常优秀，很多 patch 可能还没被gcc接受，所以还是应该直接用它的（而且他提供Windows下[mingw交叉编译的]和Linux下的二进制版本，比较方便；如果不是很有时间和兴趣，不建议下载 src 源码包自己编译，很麻烦，Codesourcery给的shell脚本很多时候根本没办法直接用，得自行提取关键的部分手工执行，又费精力又费时间，如果想知道细节，其实不用自己编译一遍，看看他是用什么步骤构建的即可，如果你对交叉编译器感兴趣的话。

**ABI 和 EABI**  
ABI：二进制应用程序接口(Application Binary Interface (ABI) for the ARM Architecture)。在计算机中，应用二进制接口描述了应用程序（或者其他类型）和操作系统之间或其他应用程序的低级接口。

EABI：嵌入式ABI。嵌入式应用二进制接口指定了文件格式、数据类型、寄存器使用、堆积组织优化和在一个嵌入式软件中的参数的标准约定。开发者使用自己的汇编语言也可以使用 EABI 作为与兼容的编译器生成的汇编语言的接口。

两者主要区别是，ABI是计算机上的，EABI是嵌入式平台上（如ARM，MIPS等）。

**arm-linux-gnueabi-gcc 和 arm-linux-gnueabihf-gcc**两个交叉编译器分别适用于 armel 和 armhf 两个不同的架构，armel 和 armhf 这两种架构在对待浮点运算采取了不同的策略（有 fpu 的 arm 才能支持这两种浮点运算策略）。

其实这两个交叉编译器只不过是 gcc 的选项 -mfloat-abi 的默认值不同。gcc 的选项 -mfloat-abi 有三种值 soft、softfp、hard（其中后两者都要求 arm 里有 fpu 浮点运算单元，soft 与后两者是兼容的，但 softfp 和 hard 两种模式互不兼容）：  
soft： 不用fpu进行浮点计算，即使有fpu浮点运算单元也不用，而是使用软件模式。  
softfp： armel架构（对应的编译器为 arm-linux-gnueabi-gcc ）采用的默认值，用fpu计算，但是传参数用普通寄存器传，这样中断的时候，只需要保存普通寄存器，中断负荷小，但是参数需要转换成浮点的再计算。  
hard： armhf架构（对应的编译器 arm-linux-gnueabihf-gcc ）采用的默认值，用fpu计算，传参数也用fpu中的浮点寄存器传，省去了转换，性能最好，但是中断负荷高。

把以下测试使用的C文件内容保存成 mfloat.c：

#include <stdio.h>

int main(void)

{

double a,b,c;

a = 23.543;

b = 323.234;

c = b/a;

printf(“the 13/2 = %f\n”, c);

printf(“hello world !\n”);

return 0;

}

1、使用 arm-linux-gnueabihf-gcc 编译，使用“-v”选项以获取更详细的信息：

arm-linux-gnueabihf-gcc -v mfloat.c

COLLECT\_GCC\_OPTIONS=’-v’ ‘-march=armv7-a’ ‘-mfloat-abi=hard’ ‘-mfpu=vfpv3-d16′ ‘-mthumb’ -mfloat-abi=hard

可看出使用hard硬件浮点模式。

2、使用 arm-linux-gnueabi-gcc 编译：

arm-linux-gnueabi-gcc -v mfloat.c

COLLECT\_GCC\_OPTIONS=’-v’ ‘-march=armv7-a’ ‘-mfloat-abi=softfp’ ‘-mfpu=vfpv3-d16′ ‘-mthumb’ -mfloat-abi=softfp

可看出使用softfp模式。

**2.5.2 构建交叉编译工具链待整理**

参考博客： https://zhuanlan.zhihu.com/p/25752954

**2.5.3 gcc llvm 和 clang**

https://www.jianshu.com/p/1367dad95445

https://zhuanlan.zhihu.com/p/357803433

# 第三章 u-Boot 原理及移植

**2.1 u-boot介绍**

待完善

**2.2 常用指令及环境变量**

**2.2.1 常用指令**

**help**

查看当前uboot版本所支持的所有命令，对于列出的命令并不是都能运行的，如果该命令并没有在板级文件中使能配置的话，那么直接在命令行中输入后按回车键，将会直接提示运行不了 。

如果我们想查看某个命令的详细用法的话，可以输入：

=>help command\_name 或 =>? command\_name

**信息查询相关命令**

**bdinfo**

查询当前板子的相关信息，可以查看板子的DRAM的大小以及DRAM的起始地址、当前使用的网络接口以及IP地址、波特率和uboot偏移地址以及偏移量等相关信息。

**version**

查询uboot版本和交叉编译工具的相关信息。

**环境变量相关命令**

**printenv**

打印环境变量，例如：当前串口的波特率baudrate、启动参数bootargs以及启动命令bootcmd等，这些环境变量都是字符串，能对其进行修改。

**setenv <name> <value>**

设置环境变量，setenv <name> <value>，当修改环境变量有空格值的时候，例如bootcmd或者bootargs等，这个时候的环境变量值需要使用单引号括起来。setenv命令也可以用来新建环境变量。

**editenv <value>** 编辑环境变量

**saveenv**  保存环境变量

**USB操作相关命令**

|  |  |
| --- | --- |
| 指令 | 功能 |
| usb start | 开始扫描USB控制器 |
| usb reset | 复位USB控制器 |
| usb stop [f] | 关闭USB控制器，[f]=force |
| usb tree | 已连接的USB设备树 |
| usb info [dev] | 显示USB设备[dev]的信息 |
| usb storage | 显示已连接的USB存储设备信息 |
| usb dev [dev] | 显示和设置当前USB存储设备 |
| usb part [dev] | 显示USB存储设备[dev]的分区信息 |
| usb read addr blk# cnt | 读取USB存储设备数据 |

**网络传输相关命令**

|  |  |
| --- | --- |
| dhcp | boot image via network using DHCP/TFTP protocol |
| rarpboot | boot image via network using RARP/TFTP protocol |
| nfs | boot image via network using NFS protocol |
| tftpboot | boot image via network using TFTP protocol |
| bootp | boot image via network using BOOTP/TFTP protocol |

这几个命令的格式都是：<指令> [目的SDRAM地址] [[主机IP:]文件名]

**注意：**

要使用dhcp、rarpboot或 bootp 等功能要路由器或Host的支持。

如果没有输入[目的SDRAM地址]，系统就是用编译时定义的CONFIG\_SYS\_LOAD\_ADDR作为目的SDRAM地址

如果tftpboot和nfs命令没有定义[主机IP:]，则使用ENV中的serverip

其它命令必需定义[主机IP:]，否则会使用提供动态IP服务的主机IP作为[主机IP:]。

**内存操作相关命令**

**md [.b, .w, .l] address [# of objects]**

命令用法中的[.b, .w, .l]对应着byte、word、long，分别以1个字节、2个字节、4个字节来进行内存值显示，address表示要查看的内存起始地址，[# of objects]表示要查看的数据长度，和显示的数据格式有关，并且需要注意的是uboot命令中的数字都是十六进制的。

例如： md.b 80000000 14 等价 md.b 80000000 0x14

**nm [.b, .w, .l] address**

nm命令用于修改指定地址的内存值。nm命令同样是使用[.b, .w, .l]来指定内存的操作格式，例如，想要使用修改0x80000000地址的数据为0x77，可使用下面的命令：

=> nm.b 80000000 或 => nm.b 0x80000000

**mm [.b, .w, .l] address**

mm命令也是可以用来修改内存值，但是使用mm命令修改内存值的时候，地址值将会自增，使用nm命令时，地址值将不会自增。

**mw [.b, .w, .l] address value [count]**

mw命令用来使用一个指定的数据填充一段内存，该命令同样使用[.b, .w, .l]来指定操作格式，address表示要填充的内存起始地址，value表示要填充的数据，count是要填充的长度。

例如，使用.b格式将以0x80000000为起始地址的0x14个内存块填充为0x33，命令如下：

=> mw.b 80000000 33 14

**cp [.b, .w, .l] source target count**

cp命令是数据拷贝命令，用于将DRAM中的数据从一段内存中拷贝到另一段内存中，该命令同样使用[.b, .w, .l]来指定操作格式，source表示为内存源地址，target表示为目标地址，count为拷贝的长度。

例如，使用.b格式将0x80000000开始地址处的0x14个字节拷贝到0x80000100地址处，命令如下所示：

=> cp.b 80000000 80000100 14

or

=> cp.b 0x80000000 0x80000100 14

**cmp [.b, .w, .l] addr1 addr2 count**

cmp命令用于比较两段内存的数据是否相等，该命令同样使用[.b, .w, .l]来指定操作格式，addr1为第一段内存首地址，addr2为第二段内存首地址，count表示要比较的长度。

例如，使用.b格式来比较0x80000000和0x80000100两个地址的数据是否相等，比较的长度为0x14个字节，命令如下：

=> cmp.b 80000000 80000100 14

**emmc和sd卡相关操作命令**

对于uboot来说是支持emmc或者sd卡，因此也需要提供给用户emmc和sd卡的相关操作命令，uboot中常用于操作mmc设备的命令为"mmc"，mmc是一系列的命令，它的后面可以跟不同的参数

|  |  |
| --- | --- |
| 命令 | 功能 |
| mmc info | 输出mmc设备的信息, SD卡的速率，容量,总线宽度 |
| mmc read | mmc read addr blk# cnt  读取mmc设备中的数据，其中addr是将数据读取到DRAM中的地址，blk是要读取的块起始地址，一块为512字节，cnt则是要读取的块的数量。  例如，从当前的SD卡设备的第2048块开始，读取20个块数据到DRAM的0x80000100地址处，该命令如下：  => mmc list  => mmc dev 0  => mmc read 80000100 800 14 |
| mmc write | mmc write addr blk# cnt  向mmc设备中写入数据，addr是要写入到mmc设备中的数据在DRAM中的起始地址，blk是要写入mmc的块起始地址，cnt是要写入的块数量，一个块的大小为512字节。  例如，将DRAM地址0x80000100开始的数据，从mmc设备的2048个块开始烧写，烧写20个块，命令如下：  => mmc list  => mmc part  => mmc write 80000100 2048 14 |
| mmc erase | mmc erase blk# cnt  其中blk是要擦除的起始块，cnt是要擦除的块数量。  例如，可以使用下面的命令擦除mmc设备从2048个块开始的20个块：  => mmc list  => mmc part  => mmc erase 800 14 |
| mmc rescan | 扫描当前目标板上所有的mmc设备，包括eMMC和SD卡 |
| mmc part | 查看当前mmc设备的分区 |
| mmc dev | 切换当前的mmc设备，mmc dev [dev] [part]  其中[dev]表示要切换的mmc设备号，[part]是mmc设备的分区号，如果不写分区号，则默认为分区0。 |
| mmc list | 查看当前目标板共有多少个mmc设备 |
| mmc hwpartition | 设置mmc设备的分区 |
| mmc bootbus | 设置指定mmc设备的BOOT\_BUS\_WIDTH的值 |
| mmc bootpart | 设置指定mmc设备的boot和RPMB分区大小 |
| mmc partconf | 设置指定mmc设备的PARTITION\_CONFIG的值 |
| mmc rst | mmc设备复位 |
| mmc setdsr | 设置mmc设备DSR寄存器的值 |

**Nand Flash相关操作命令**

uboot中除了有emmc子系统外，还具有nand子系统，所有uboot也是支持Nand Flash的。

**nand info**

打印目标板上Nand Flash的相关信息，页面大小和00B大小等相关信息。

**nand device**

显示Nand Flash的信息，也能用于切换目标板上的Nand Flash，如果目标板支持多块Nand Flash的话，可以使用该命令进行切换。

**nand erase**

擦除Nand Flash，在对Nand Flash中写入数据之前，必须要先对写的区域进行擦除，然后才能保证数据能写入进擦除的区域内，nand erase命令的形式有3种，如下：

***第一种形式如下：***

nand erase[.spread] [clean] off size

off表示Nand Flash的偏移地址，也就是要擦除区域的起始地址，size表示要擦除的区域大小。

例如，可以使用下面的命令将Nand Flash存储Linux内核镜像zImage的区域进行擦除：

=> nand erase 0x400000 0x1000000

***第二种形式如下：***

nand erase.part [clean] partition

表示擦除指定的区域。

***第三种形式如下：***

nand erase.chip [clean]

该形式将会将整个Nand Flash进行擦除，nand erase命令一般是与nand write命令进行配合使用。

**nand read**

nand read命令可以用于从Nand Flash中指定的地址读取指定大小的数据到DRAM中，该命令的使用格式如下：

nand read - addr off|partition size

命令使用格式中，addr表示DRAM的地址，off表示要读取的Nand Flash的区域的起始地址，size表示要读取的数据大小。

例如，可以使用下面的命令读取Linux内核镜像zImage文件到DRAM的0x80800000地址处：

=> nand read 0x80800000 0x400000 0x1000000

**nand write**

向Nand Flash中指定的地址写入指定大小的数据，一般和nand erase命令结合使用，还可以用来更新Nand Flash中的uboot、kernel和dtb等文件，该命令的使用格式如下：

nand write - addr off|partition size

和nand read命令类似，addr表示要写入的数据的开始地址，off表示Nand Flash中写入的起始地址，size表示要写入的数据大小。

例如，使用该命令并结合fatload命令进行Linux内核镜像zImage和dtb文件更新，将我们需要更新的文件放入到SD卡中，使用fatload命令将需要更新的zImage和dtb写入到DRAM中，然后使用nand write命令更新到Nand Flash中，如下：

将需要更新的zImage镜像文件读入到DRAM中：

=> fatload mmc 0:1 0x80800000 img/zImage

擦除Nand Flash中存储zImage的区域，并将DRAM中的zImage更新到Nand Flash中：

=> nand erase 0x400000 0x1000000

=> nand write 0x80800000 0x400000 0x1000000

接下来，按照类似的方法去更新dtb：

将需要更新的dtb设备树文件读入到DRAM中：

=> fatload mmc 0:1 0x83000000 img/imx6ul-14x14-evk.dtb

擦除Nand Flash中存储dtb设备树的区域，并将DRAM中的dtb文件更新到Nand Flash中：

=> nand erase 0x1400000 0x80000

=> nand write 0x83000000 0x1400000 0x80000

另外，还可以使用bootz命令启动Linux内核，使用下面命令即可：

=> bootz 0x80800000 - 0x83000000

**FAT格式文件系统相关操作命令**

**fatinfo <interface> [<dev[:part]>]**

查询mmc设备中指定分区的文件系统信息，<interface> 表示要查看的接口，例如mmc，[<dev[:part]>]中的dev表示要查询的设备号，part则表示要查询的分区。

例如，当前我的目标板有个sd卡设备，查看sd卡中分区1的文件系统信息，可以使用下面的命令：

=> mmc list

=> mmc part

=> fatinfo mmc 0:1

**fatls <interface> [<dev[:part]>] [directory]**

查询FAT格式文件系统的目录和文件信息，<interface>表示要查询的接口，[<dev[:part]>]中dev表示要查询的设备号，part表示要查询分区，[directory]表示要查询的目录，如果该参数不输入的话，默认为根目录。

例如，查询我当前sd卡中分区1中的目录和文件，可以输入下面命令：

=> fatls mmc 0:1

当我们再想查看img/目录下的目录或者文件时，可以使用下面的命令：

=> fatls mmc 0:1 img/

**fstype**

fstype命令可以用于查看mmc设备中某个分区的文件系统格式，fstyp命令具有两个用法，第一个用来查看mmc设备分区中的文件系统类型，第二个则是用来设置文件系统类型的环境变量，对于第一个命令用法，<interface>表示接口，例如mmc，<dev>:<part>中dev则表示要查询的设备号，part则是设备的分区。

例如，查看我当前目标板中sd设备的第一个分区的文件系统类型，可以使用下面命令：

=> fstype mmc 0:1

**fatload <interface> [<dev[:part]> [<addr> [<filename> [bytes [pos]]]]]**

fatload命令用来将指定的文件读取到DRAM内存中，<interface>表示设备接口，例如mmc，[<dev[:part]>中的dev表示设备号，part表示mmc设备的分区，<addr>则是文件读取到DRAM中的起始地址，<filename>则是要读取的文件的名字，bytes表示要读取多少字节的数据，如果该值为0或者未使用，则表示将要读取整个文件，pos表示要读的文件相对于文件首地址的偏移，如果为0或者未使用，则表示从文件首地址开始读取。

例如，在我当前的目标帮中，将sd卡中第一个分区中的img/u-boot-imx6ul14x14evk\_nand.imx文件读取到DRAM中0x80000100起始地址中，可以使用下面的命令：

=> fatls mmc 0:1 img/

=> fatload mmc 0:1 80000100 img/u-boot-imx6ul14x14evk\_nand.imx

**fatwrite**

当在uboot的板级配置文件中定义了#define CONFIG\_CMD\_FAT宏，fatinfo、fatls、fatload命令才会出现在uboot命令中，而fatwrite命令则需要定义#define CONFIG\_FAT\_WRITE宏才会出现，因此，如果想要在uboot中使用fatwrite命令，则需要定义宏CONFIG\_FAT\_WRITE。

fatwrite命令可以用于将DRAM中的数据写入到mmc设备中去，命令格式中，<interface>表示为接口，例如mmc，<dev[:part]>中dev表示为设备号，part表示为mmc设备的分区，<addr>则为要写入的数据在DRAM中的起始地址，<filename>表示写入的数据文件的名字，<bytes>表示要写入的字节数。

例如，在我的目标板系统中从DRAM地址0x80000100开始写20个字节，在sd卡设备的分区1中生成test.bin文件，可以使用下面命令：

=> mmc list

=> fatls mmc 0:1

=> fatwrite mmc 0:1 80000100 test.bin 14

=> fatls mmc 0:1

**EXT格式文件系统相关操作命令**

uboot中除了有FAT格式文件系统的相关操作命令外，还有EXT格式文件系统的相关操作命令，这些命令和实现的功能如下所示：

|  |  |
| --- | --- |
| 命令 | 功能 |
| ext2load | 从Ext2文件系统中加载二进制文件到DRAM中 |
| ext2ls | 列举目录中的文件 |
| ext4load | 从Ext4文件系统中加载二进制文件到DRAM中 |
| ext4ls | 列举目录中的文件 |
| ext4size | 修改文件大小 |
| ext4write | 在root目录下新创建文件 |

**BOOT启动相关操作命令**

**boot**

boot命令是用来启动Linux系统的，该命令将会运行bootcmd，也就是boot命令将会读取bootcmd这个环境变量，并运行这个环境变量中的命令，查看当前目标板中的bootcmd环境变量内容：print bootcmd

**bootm [addr [arg ...]]**

bootm命令用于启动uImage镜像，其中addr就是uImage镜像文件在DRAM的起始地址，另外它还有一些参数，如果要使用dtb或initrd的话，则就是在后面添加对应的DRAM地址。

**bootz [addr [initrd[:size]] [fdt]]**

bootz命令也是用来启动Linux系统，只不过启动的是Linux zImage镜像文件，命令中的addr是DRAM内存Linux镜像文件的起始地址，initrd是initrd文件在DRAM中的地址，fdt是设备树在DRAM中的地址，如果没有使用到initrd文件的话，则使用'-'进行替代，对于Linux镜像和设备树文件，我们可以通过读取eMMC或Nand Flash到DRAM中，当然，如果在网络接口可以使用的情况下，也可以使用NFS或TFTP服务将镜像下载到DRAM中，启动的原理是一样的。

例如，当Nand Flash中存储着我们需要启动的Linux系统镜像和设备树文件的话，可以使用下面命令进行启动：

先读取Linux zImage镜像到DRAM的0x80800000处：

=> nand read 0x80800000 0x400000 0x1000000

然后读取设备树文件到DRAM的0x83000000处：

=> nand read 0x83000000 0x1400000 0x80000

使用bootz启动引导Linux系统：

=> bootz 0x80800000 - 0x83000000

**其它常用uboot命令**

**reset**

reset命令能用来复位CPU

**go addr [arg ...]**

go命令能用于跳转到指定的内存地址处执行应用程序，命令用法中的addr就是内存的地址。

**run var [...]**

run命令能用来运行环境变量中定义的命令，例如通过run bootcmd来运行bootcmd中定义的启动命令，能将Linux系统进行启动，该命令能运行我们自己定义的环境变量

**mtest [start [end [pattern [iterations]]]]**

mtest命令能用于进行内存读写测试，例如可以用来测试目标板DDR的稳定性，命令用法中的start是DRAM内存的起始地址，end是内存的结束地址。

例如我们向测试0x80000000到0x87800000这段内存，可以使用下面命令：

=> mtest 80000000 87800000

测试的时候，如果想退出测试的话，可以使用键盘上的"Ctrl+C"组合键。

<https://www.cnblogs.com/Cqlismy/p/12214305.html>

<https://blog.csdn.net/ghostyu/article/details/6968681>

**2.2.2 环境变量**

|  |  |
| --- | --- |
| 变量 | 解释 |
| bootargs | 传递给内核的参数，设置了许多重要的参数 |
| bootcmd | 启动uboot会执行的命令 |
| bandrate | 串口速率 |
| ipaddr | 本地IP地址 |
| serverip | 服务端地址 |
| bootdelay | 启动前的延时等待 |
|  |  |

**2.2.2 从uboot网络启动嵌入式设备**

**搭建tftp服务器**

1. $sudo apt-get install tftp-hpa tftpd-hpa
2. 编辑 /etc/default/tftpd-hpa

|  |
| --- |
| # /etc/defaut/tftpd\_hpa  TFTP\_USERNAME = "tftp"  # 共享目录  TFTP\_DIRECTORY = "/var/lib/tftproot"  TFTP\_ADDRESS = ":69"  TFTP\_OPTIONS = "-l -c -s"  # TFTP\_OPTIONS = "--secure" |

1. 将共享目录权限设置为 777
2. 重启 tftp 服务： $: sudo service tftpd-hpa restart
3. 使用 $: tftp <服务IP> put/get 上传/下载

**关闭服务器防火墙**

Ubuntu防火墙设置

1.安装

sudo apt-get install ufw

2.启用

sudo ufw enable

sudo ufw default deny

运行以上两条命令后，开启了防火墙，并在系统启动时自动开启。

关闭所有外部对本机的访问，但本机访问外部正常。

3.开启/禁用

sudo ufw allow|deny [service]

打开或关闭某个端口，例如：

sudo ufw allow smtp　允许所有的外部IP访问本机的25/tcp (smtp)端口

sudo ufw allow 22/tcp 允许所有的外部IP访问本机的22/tcp (ssh)端口

sudo ufw allow 53 允许外部访问53端口(tcp/udp)

sudo ufw allow from 192.168.1.100 允许此IP访问所有的本机端口

sudo ufw allow proto udp 192.168.0.1 port 53 to 192.168.0.2 port 53

sudo ufw deny smtp 禁止外部访问smtp服务

sudo ufw delete allow smtp 删除上面建立的某条规则

4.查看防火墙状态

sudo ufw status

一般用户，只需如下设置：

sudo apt-get install ufw

sudo ufw enable

sudo ufw default deny

以上三条命令已经足够安全了，如果你需要开放某些服务，再使用sudo ufw allow开启。

补充：

开启/关闭防火墙 (默认设置是’disable’)

#ufw enable|disable

转换日志状态

#ufw logging on|off

设置默认策略 (比如 “mostly open” vs “mostly closed”)

#ufw default allow|deny

许可或者屏蔽某些入埠的包 (可以在“status” 中查看到服务列表［见后文］)。可以用“协议：端口”的方式指定一个存在于/etc/services中的服务名称，也可以通过包的meta-data。 ‘allow’ 参数将把条目加入 /etc/ufw/maps ，而 ‘deny’ 则相反。基本语法如下：

#ufw allow|deny [service]

显示防火墙和端口的侦听状态，参见 /var/lib/ufw/maps。括号中的数字将不会被显示出来。

#ufw status

［注意：上文中虽然没有使用 sudo，但是命令提示符号都是“#”］

UFW 使用范例：

允许 53 端口

$ sudo ufw allow 53

禁用 53 端口

$ sudo ufw delete allow 53

允许 80 端口

$ sudo ufw allow 80/tcp

禁用 80 端口

$ sudo ufw delete allow 80/tcp

允许 smtp 端口

$ sudo ufw allow smtp

删除 smtp 端口的许可

$ sudo ufw delete allow smtp

允许某特定 IP

$ sudo ufw allow from 192.168.254.254

删除上面的规则

$ sudo ufw delete allow from 192.168.254.254

而对于uboot 可以直接关闭防火墙，或者允许指定IP

**搭建NFS服务器**

1. 安装nfs服务：

$ sudo apt-get install nfs-kernel-server

2. 修改配置文件： /etx/exports

在配置最后加上：

/home/administrator/share/nfsRootfs \*(insecure, rw, sync, no\_subtree\_check, all\_squash, anonuid=1000, anongid=1000)

其中 anonuid 为用户 udi ，anongid 为用户 gid

3. 重启 nfs-ernel-server 服务

4. 客户端：查看指定IP的nfs共享目录： $ showmount -e <serverIP>

挂载：$ sudo mount <serverIP>:/path/to/share/ <localDir>

5. windows 挂载 nfs

1）控制面板->程序->程序和功能->打开或关闭windows功能

2）勾选nfs服务及相关子项

3）进入cmd执行：mount \\<serverIP>\path\to\share x:

其中 x 为一个挂载点，选择一个闲置的盘符即可

windows设置文件共享：

<1>右击网络->属性->更改高级共享设置 启用相关网络范围内的网络发现

<2>选择需要共享的文件夹右击->共享->共享->选择一个用户，如果有可以选择 Guest

或： 选择需要共享的文件夹右击->共享->高级共享->设置共享此文件夹

<3>使用：打开我的电脑，在地址栏输入 \\<serverIP>\

注意：第一次登陆需要使用服务端的用户账号和密码

**构建部署kernel、设备树、根文件系统**

这里以buildroot和 udoo-neo-full为例简要说明

git://git.busybox.net/buildroot

make list-defconfigs | grep udoo

make mx6sx\_udoo\_neo\_defconfig

make all

将 output/image 路径下的 zImage 和 设备树 放到 tftp 共享路径下

将 output/image 路径下的根文件系统解压到 nfs 共享目录下

**修改uboot环境变量**

通常需要修改 serverip、ipaddr、image(kernel)、fdt\_file(dtb)、nfsroot(rootfs) 这几个变量

例如：

serverip=192.168.2.21

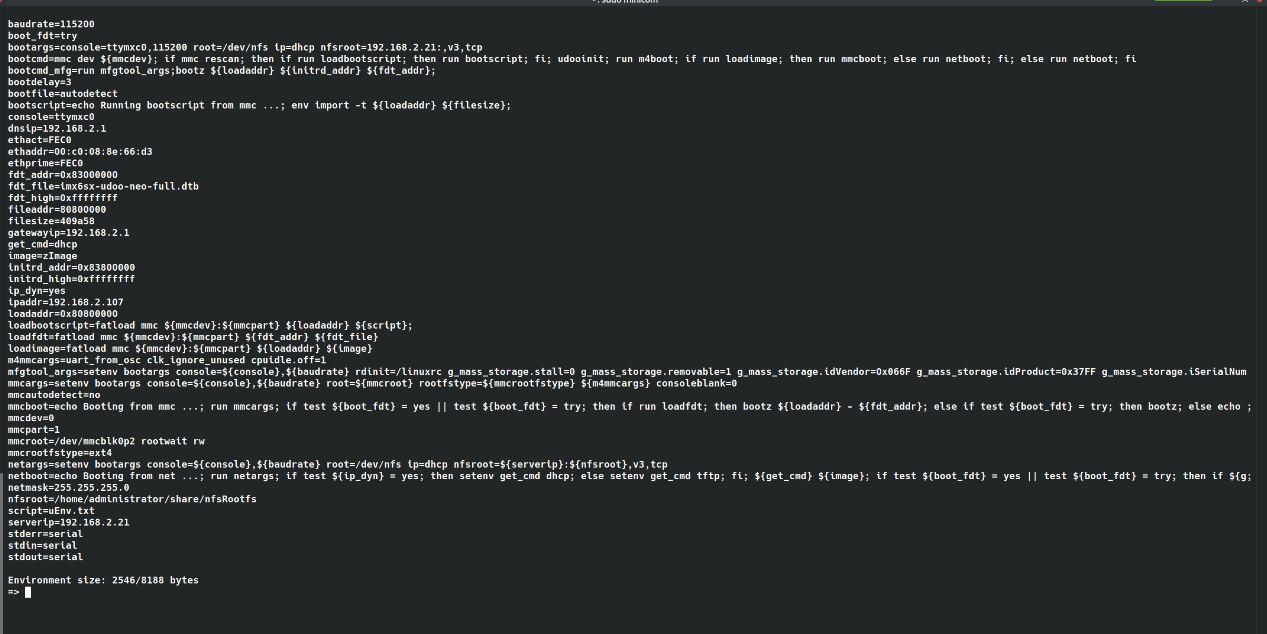
ipaddr=192.168.2.107

image=zImage

fdt\_file=imx6sx-udoo-neo-full.dtb

nfsroot=/home/administrator/share/nfsRootfs

以下为配置之后的参数：



**网络启动**

一般boot里会设置网络启动的参数，直接通过run命令执行即可，例如： run netboot

此时使用的参数为：­­­

**netargs**=setenv bootargs console=${console},${baudrate} root=/dev/nfs ip=dhcp nfsroot=${serverip}:${nfsroot},v3,tcp

**2.3 u-boot编译架构**

待完善

**2.4 源码分析**

**2.4.1 链接脚本（lds文件）**

**lds文件：**<https://www.jianshu.com/p/ec39403db315>

<https://www.jianshu.com/p/42823b3b7c8e>

<https://blog.csdn.net/u010833290/article/details/8991585>

<https://www.zhihu.com/question/36860638>

▲ <https://insidelinuxdev.net/article/a00r78.html>

<http://ftp.gnu.org/old-gnu/Manuals/ld-2.9.1/html_mono/ld.html>

<http://www.gnu.org/software/make/manual/make.html>

每一个链接都是链接脚本（linker script）控制的，脚本是按照链接器命令语言（linker command language）书写的。

链接脚本的主要目的是描述输入文件的section如何映射到输出文件，以及如何控制输出文件的内存布局。除此之外，大多数链接脚本不做什么其它的事。然而如果需要，链接脚本也可以指导链接器执行其它许多操作。

链接器总是使用链接脚本，如果没有提供，它使用缺省脚本，使用--verbose命令行选项，可以显示缺省的链接脚本。一些命令如-r或者-N，会影响缺省链接脚本。

可以使用-T选项指定自己的链接脚本，这么做时，你的脚本完全替换缺省链接脚本。

**基本的链接脚本概念：**

section：可参考本博客ELF分类中的文章。

input section：输入文件中的section。

output section：输出文件中的section。

loadable：当输出文件运行时，该section的内容加载内存中。

allocatable：该section没有内容，但是会分配内存，通常是.bss section。

每个可加载和可分配section有两个地址，第一个是VMA（virtual memory address），这是输出文件运行时，section的地址。第二个是LMA（load memory address），这是section要加载到的地址。大多数情况下，这两个地址相同。但是在一些嵌入式系统中，整个目标文件烧写到nor flash（指令可直接执行），只读段仍然在ROM中，但是变量加载到RAM中，此时VMA和LMA不同。

**链接脚本格式：**

链接脚本是文本文件，链接脚本由一系列命令组成。每一个命令或者是一个关键字，关键字可以带参数；或者是给符号赋值。可以用分号分开各个命令。注释的格式为 /\* \*/。文件名和格式名可直接输入，如果有特殊字符，使用双引号。

**简单的链接脚本例子：**

最简单的链接脚本只有一个命令："SECTIONS"，这个命令用来描述输出文件的内存布局。

SECTIONS

{

. = 0x10000;

.text : { \*(.text) }

. = 0x8000000;

.data : { \*(.data) }

.bss : { \*(.bss) }

}

.符号是location counter，用于指定section的地址，SECTIONS命令开始时，其值为0，可以显式设置，如果没有设置，则按照section大小自动增长。.text定义一个ouput section。后面跟一个冒号，现在可以省略。大括号用来指定input sections，\*是通配符，匹配任何文件名。表达式'\*(.text)'表示输入文件的所有.text section。链接器保证output section满足对齐要求。

**ENTRY命令**

程序执行的第一条指令叫做entry point。ENTRY命令用于设置入口点。链接器按照下面的顺序寻找入口点。

1. -e 命令行选项。
2. ENTRY(symbol)链接脚本命令。
3. 目标指定的符号，通常是start。
4. .text的第一个字节。
5. 0

**SECTIONS命令**

SECTIONS命令告诉链接器，如何映射输入section到输出section，如何将输出section放到内存。SECTIONS命令的格式：

SECTIONS

{

sections-command

sections-command

...

}

每个sections-command可能是下列之一：

1. 一个ENTRY命令
2. 一个符号赋值
3. 一个输出section描述
4. 一个overlay描述

ENTRY命令和符号赋值允许出现在SECTIONS命令中，是为了在这些命令中方便的使用location counter。这也使链接脚本更容易理解，因为你将这些命令放在输出文件布局中有意义的位置。

如果在链接脚本中，没有使用SECTIONS命令。链接器将每一个输入section放入一个唯一命名的输出section。按照其在第一个输入文件的顺序。当然也会合并输入section。第一个section的地址将为0。

其他更详细的信息、命令、学习资料参考： <https://insidelinuxdev.net/article/a00r78.html>



**一段典型脚本：（hello.lds）**

ENTRY(helloworld)

SECTIONS

{

. = 0x20008000;

.text :

{

start.o(.text);

\*(.text);

}

. = ALIGN(4);

.data :

{

\*(.data);

}

.bss :

{

\*(.bss);

}

}

----------------------------------

1、点号(.)：

点号在SECTIONS命令里称为位置计数器，代表了当前位置。如上面的0x20008000表示代码段的起始位置从这个值开始。

也可以不指定，如.data段和.bss段；代表紧挨着上面的段分布。如果一开始就不指定，则默认值为0。

2、输出段（.text/.data/.bss）：

各关键字代表了输出段的段名，花括号内定义了各段对应的内容。段名可任意定义，输出段的实际内容与段名无关，只与花括号内的具体内容有关。

如：.text，把列出的（\*代表任意）输入文件的.text段合并成一个.text（其中，start.o中的.text放在最前面）；

该段的地址由定位器符号的值指定，即0x20008000；

3、ALIGN（N）：

产生对齐的代码或数据，很多体系结构对对齐的代码或数据有严格的要求。

如 . = ALIGN（4）表示位置计数器向高地址方向取最近的4字节的整数倍。

4、ENTRY(SYMBOL)：

相等于-e，将符号SYMBOL的值设置成入口地址（进程执行的第一条用户空间的指令，在进程地址空间的地址）。

arm-linux-ld 有多种方法设置进程入口地址：按以下顺序，优先级依次之：

1、arm-linux-ld 命令行的 -e 选项；

2、连接脚本的ENTRY(SYMBOL)命令；

3、如果定义了start符号，使用start符号值；

4、如果存在.text section，使用.text section的第一字节的位置值；

5、使用值0；

-----------------------------------

使用：(运行链接命令时，借助于-T参数）

arm-linux-ld -T hello.lds -o file.eld

**u-boot入口：** <https://blog.csdn.net/weixin_43288201/article/details/107496580>

<https://zhuanlan.zhihu.com/p/132479321>

<https://blog.csdn.net/newnewman80/article/details/9948509>

程序的链接是由链接脚本来决定的，所以通过链接脚本可以找到程序的入口。

根据编译后的u-boot根目录下的u-boot.lds文件可以看到

入口点是 arch/arm/lib/vectors.S 文件中的\_start --- 中断向量表 然后跳转到

arch/arm/cpu/armv7/start.S 里面的reset函数

**2.4.2 基本汇编语法**

<https://www.crifan.com/files/doc/docbook/uboot_starts_analysis/release/htmls/summary_assembly.html>

**1. 汇编中的标号=C中的标号**

如：

reset:

/\*

\* set the cpu to SVC32 mode

\*/

mrs r0,cpsr

中的reset，就是汇编中的标号，相对来说，比较容易理解，就相当于C语言的标号。

比如，C语言中定义一个标号ERR\_NODEV：

ERR\_NODEV: /\* no device error \*/

... /\* c code here \*/

然后对应在别处，使用goto去跳转到这个标号ERR\_NODEV：

if (something)

goto ERR\_NODEV ;

**2. 汇编中的跳转指令=C中的goto**

对应地，和上面的例子中的C语言中的编号和掉转到标号的goto类似，汇编中，对于定义了标号，那么也会有对应的指令，去跳转到对应的汇编中的标号。这些跳转的指令，就是b指令，b是branch的缩写。

b指令的格式是：

b{cond} label

简单说就是跳转到label处。

用和上面的例子相关的代码来举例：

.globl \_start

\_start: b reset

就是用b指令跳转到上面那个reset的标号。

**3. 汇编中的.globl=C语言中的extern**

对于上面例子中：

.globl \_start

中的.global，就是声明\_start为全局变量/标号，可以供其他源文件所访问。

即汇编器，在编译此汇编代码的时候，会将此变量记下来，知道其是个全局变量，遇到其他文件是用到此变量的的时候，知道是访问这个全局变量的。

因此，从功能上来说，就相当于C语言用extern去生命一个变量，以实现本文件外部访问此变量。

**4.** **汇编中用bl指令和mov pc，lr来实现子函数调用和返回**

和b指令类似的，另外还有一个bl指令，语法是：

bl{cond} label

其作用是，除了b指令跳转到label之外，在跳转之前，先把下一条指令地址存到lr寄存器中，以方便跳转到那边执行完毕后，将lr再赋值给pc，以实现函数返回，继续执行下面的指令的效果。

用下面这个start.S中的例子来说明：

bl cpu\_init\_crit

......

cpu\_init\_crit:

......

mov pc, lr

其中，就是先调用bl掉转到对应的标号cpu\_init\_crit，其实就是相当于一个函数了，然后在cpu\_init\_crit部分，执行完毕后，最后调用 mov pc, lr，将lr中的值，赋给pc，即实现函数的返回原先 bl cpu\_init\_crit下面那条代码，继续执行函数。上面的整个过程，用C语言表示的话，就相当于：

......

cpu\_init\_crit();

......

void cpu\_init\_crit(void)

{

......

}

而关于C语言中，函数的跳转前后所要做的事情，都是C语言编译器帮我们实现好了，会将此C语言中的函数调用，转化为对应的汇编代码的。

其中，此处所说的，函数跳转前后所要做的事情，就是：

函数跳转前，要将当前指令的下一条指令的地址，保存到lr寄存器中

函数调用完毕后，将之前保存的lr的值给pc，实现函数跳转回来。继续执行下一条指令。

而如果你本身自己写汇编语言的话，那么这些函数跳转前后要做的事情，都是你程序员自己要关心，要实现的事情。

**5. 汇编中的对应位置有存储值的标号 = C语言中的指针变量**

像前文所解析的代码中类似于这样的：

LABEL1：.word Value2

比如：

\_TEXT\_BASE:

.word TEXT\_BASE

所对应的含义是，有一个标号\_TEXT\_BASE

而该标号中对应的位置，所存放的是一个word的值，具体的数值是TEXT\_BASE，此处的TEXT\_BASE是在别处定义的一个宏，值是0x33D00000。

所以，即为：

有一个标号\_TEXT\_BASE，其对应的位置中，所存放的是一个word的值，值为

TEXT\_BASE=0x33D00000

总的来说，此种用法的含义，如果用C语言来表示，其实更加容易理解：

int \*\_TEXT\_BASE = TEXT\_BASE = 0x33D00000

即：

int \*\_TEXT\_BASE = 0x33D00000

**C语言中如何引用汇编中的标号**

不过，对于这样的类似于C语言中的指针的汇编中的标号，在C语言中调用到的话，却是这样引用的：

/\* for the following variables, see start.S \*/

extern ulong \_armboot\_start; /\* code start \*/

extern ulong \_bss\_start; /\* code + data end == BSS start \*/

......

IRQ\_STACK\_START = \_armboot\_start - CFG\_MALLOC\_LEN - CFG\_GBL\_DATA\_SIZE - 4;

......

而不是我原以为的，直接当做指针来引用该变量的方式：

\*IRQ\_STACK\_START = \*\_armboot\_start - CFG\_MALLOC\_LEN - CFG\_GBL\_DATA\_SIZE - 4;

其中，对应的汇编中的代码为：

.globl \_armboot\_start

\_armboot\_start:

.word \_start

所以，针对这点，还是需要注意一下的。至少以后如果自己写代码的时候，在C语言中引用汇编中的global的标号的时候，知道是如何引用该变量的。

【总结】

汇编中类似这样的代码：

label1: .word value2

就相当于C语言中的：

int \*label1 = value2

但是在C语言中引用该标号/变量的时候，却是直接拿来用的，就像这样：

label1 = other\_value

其中label1就是个int型的变量。

**6. 汇编中的ldr+标号，来实现C中的函数调用**

接着上面的内容，继续解释，对于汇编中这样的代码：

**第一种：**

ldr pc, 标号1

......

标号1：.word 标号2

......

标号2：

......（具体要执行的代码）

或者是，

**第二种：**

ldr pc, 标号1

......

标号1：.word XXX（C语言中某个函数的函数名）

这里意思就是，将地址为标号1中内容载入到pc中。而地址为标号1中的内容，就类似第一种的标号2。

所以上面第一种的意思：就很容易看出来，就是把标号2这个地址值，给pc，即实现了跳转到标号2的位置执行代码，就相当于调用一个函数，该函数名为标号2。

第二种的意思，和上面类似，是将C语言中某个函数的函数名，即某个地址值，给pc，实现调用C中对应的那个函数。

两种做法，其含义用C语言表达，其实很简单：

PC = \*（标号1） = 标号2

例：汇编中的ldr加标号实现函数调用 示例

第一种：

......

ldr pc, \_software\_interrupt

......

\_software\_interrupt: .word software\_interrupt

......

software\_interrupt:

get\_bad\_stack

bad\_save\_user\_regs

bl do\_software\_interrupt

以上就是实现了将标号1，\_software\_interrupt，对应的位置中的值，标号2，software\_interrupt，给pc，即实现了将pc掉转到software\_interrupt的位置，即实现了调用函数software\_interrupt的效果。

第二种：

ldr pc, \_start\_armboot

\_start\_armboot: .word start\_armboot

含义就是，将标号1，\_start\_armboot，所对应的位置中的值，start\_armboot给pc，即实现了调用函数start\_armboot的目的。

其中，start\_armboot是C语言文件中某个C语言的函数。

[注意] 总结汇编中实现函数调用的方式

汇编中，实现函数调用的效果，有如下两种方法：

**方法1**

ldr pc, 标号1

......

标号1：.word 标号2

......

标号2：

......（具体要执行的代码）

**方法2**

ldr pc, 标号1

......

标号1：.word XXX（C语言中某个函数的函数名）

**7. 汇编中设置某个寄存器的值或给某个地址赋值**

在汇编代码start.S中，看到不止一处， 类似于这样的代码：

形式1：

# define pWTCON 0x53000000

......

ldr r0, =pWTCON

mov r1, #0x0

str r1, [r0]

或者是，

形式2：

# define INTSUBMSK 0x4A00001C

......

ldr r1, =0x3fff

ldr r0, =INTSUBMSK

str r1, [r0]

其含义，都是将某个值，赋给某个地址，此处的地址，是用宏定义来定义的，对应着某个寄存器的地址。

其中，形式1是直接通过mov指令来将0这个值赋给r1寄存器，和形式2中的通过ldr伪指令来将0x3ff赋给r1寄存器，两者区别是，前者是因为已经确定所要赋的值0x0是mov的有效操作数，而后者对于0x3ff不确定是否是mov的有效操作数，如果不是，则该指令无效，编译的时候，也无法通过编译，会出现类似于这样的错误：:

start.S: Assembler messages:

start.S:149: Error: invalid constant -- 'mov r1,#0xFFEFDFFF'

make[1]: \*\*\* [start.o] 错误 1

make: \*\*\* [cpu/arm920t/start.o] 错误 2

所以才用ldr伪指令，让编译器来帮你自动判断：

1. 如果该操作数是mov的有效操作数，那么ldr伪指令就会被翻译成对应的mov指令

例如：

汇编代码：

# define pWTCON 0x53000000

......

ldr r0, =pWTCON

被翻译后的真正的汇编代码：

33d00068: e3a00453 mov r0, #1392508928 ; 0x53000000

1. 如果该操作数不是mov的有效操作数，那么ldr伪指令就会被翻译成ldr指令

例如：

汇编代码：

ldr r1, =0x7fff

被翻译后的真正的汇编代码：

33d00080: e59f13f8 ldr r1, [pc, #1016] ; 33d00480 <fiq+0x60>

......

33d00480: 00007fff .word 0x00007fff

即把ldr伪指令翻译成真正的ldr指令，并且另外分配了一个word的地址空间用于存放该数值，然后用ldr指令将对应地址中的值载入，赋值给r1寄存器。

**总结汇编中给某个地址赋值的方法**

汇编中，一个常用的，用来给某个地址赋值的方法，类似如下形式：

#define 宏的名字 寄存器地址

......

ldr r1, =要赋的值

ldr r0, =宏的名字

str r1, [r0]

**2.4.3 复位之后函数调用流程**

**关键文件：**

u-boot/arch/arm/cpu/armv7/start.S 第一阶段文件

u-boot/arch/arm/cpu/armv7/lowlevel\_init.S 第一阶段文件

u-boot/arch/arm/lib/crt0.S 第二阶段文件

u-boot/common/board\_f.c

u-boot/common/board\_r.c

**重要变量：**

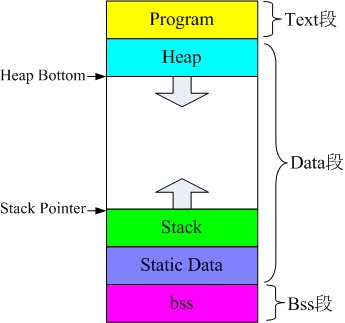
**global\_data gd\_t** u-boot/include/asm-generic/global\_data.h

保存很多有用信息的全局结构体－－**gd\_t**（global data缩写），其中包括了bd变量，可以说gd\_t结构体包括了u-boot中所有重要全局变量。

**代码段、数据段：**

一个程序本质上都是由 **bss段**、**data段**、**text段**（代码段）三个组成的。这在当前的计算机程序设计中是很重要的一个基本概念。在嵌入式系统的设计中也非常重要，牵涉到嵌入式系统运行时的内存大小分配，存储单元占用空间大小的问题。

**bss段：**在采用段式内存管理的架构中（比如x86系统），bss段通常是指用来存放程序中未初始化的全局变量的一块内存区域，一般在初始化时bss 段部分将会清零。bss段属于静态内存分配，即程序一开始就将其清零了。比如，在C语言之类的程序编译完成之后，已初始化的全局变量保存在.data 段中，未初始化的全局变量保存在.bss 段中。text和data段都在可执行文件，或镜像文件等，由系统从执行文件中加载，而bss段不在可执行文件中，由系统初始化。



图片来源：<https://zhuanlan.zhihu.com/p/28659560>

**text段:** 用于存放程序代码的区域， 编译时确定， 只读。更进一步讲是存放处理器的机器指令，当各个源文件单独编译之后生成目标文件，经连接器链接各个目标文件并解决各个源文件之间函数的引用，与此同时，还得将所有目标文件中的.text段合在一起，但不是简单的将它们“堆”在一起就完事，还需要处理各个段之间的函数引用问题。

在嵌入式系统中，如果处理器是带MMU（MemoryManagement Unit，内存管理单元），那么当我们的可执行程序被加载到内存以后，通常都会将.text段所在的内存空间设置为只读，以保护.text中的代码不会被意外的改写（比如在程序出错时）。当然，如果没有MMU就无法获得这种代码保护功能。

**data段:** 用于存放在编译阶段(而非运行时)就能确定的数据，可读可写。也是通常所说的静态存储区，赋了初值的全局变量、常量和静态变量都存放在这个域。

包含有代码的可执行程序至少有一个段，通常称为 .text。在 .data 段包含数据。可执行代码保存在 .text 段，读写 .data 段中的数据。此外，只读常数保存在 .rodata 段。 初始化为零的数据保存在 .bss 段。以 bss 段起始块定义了没有初始化的静态数据空间。

**函数调用流程：**

.globl reset u-boot/arch/arm/cpu/armv7/start.S

--> save\_boot\_params u-boot/arch/arm/cpu/armv7/start.S

--> save\_boot\_params\_ret u-boot/arch/arm/cpu/armv7/start.S

**检查是否支持 Hypervisor 模式**

--> switch\_to\_hypervisor u-boot/arch/arm/cpu/armv7/start.S

--> switch\_to\_hypervisor\_ret u-boot/arch/arm/cpu/armv7/start.S

**1）如果不支持 HYP 模式，设置为 SVC 模式**

**2）关闭 FIQ 和 IRQ**

**3）修改CP15 内的系统控制寄存器 SCTLR，将V设置为0，表示向量表在低**

**地址区**

**4）将处理器异常向量的入口地址设置为 \_start**

--> cpu\_init\_cp15 u-boot/arch/arm/cpu/armv7/start.S

**配置CP15处理器相关的寄存器来配置处理器的MMU、Cache及TLB如果**

**没有定义 CONFIG\_SYS\_ICACHE\_OFF，则打开ICACHE，关闭MMU和TLB**

--> cpu\_init\_crit u-boot/arch/arm/cpu/armv7/start.S

**跳至板子特定的初始化...**

**Mask ROM将已经初始化了基本存储器。 去这里提高时钟速率并处理唤醒**

**条件。**

--> lowlevel\_init u-boot/arch/arm/cpu/armv7/lowlevel\_init.S

**初始化堆栈后调用C函数继续进行初始化**

--> s\_init u-boot/arch/arm/cpu/armv7/kona-common/s\_init.c

**第一阶段结束**

--> \_main u-boot/arch/arm/lib/crt0.S **第二阶段开始**

**1) 设置初始C运行时环境并调用board\_init\_f(0)，即初始化堆栈指针sp**

**并且8字节对齐。GD\_SIZE 是指全局结构体gd的大小，在 GD\_SIZE 周**

**围有相关的宏设置，可以留意，文件：**

**u-boot\include\generated\generic-asm-offsets.h。**

**2) 设置中间环境（新的sp和gd）并调用relocate\_code（addr\_moni）。**

**这里的窍门是我们将返回“这里”但已重新定位**

--> board\_init\_f\_alloc\_reserve u-boot/common/init/board\_init.c

--> board\_init\_f\_init\_reserve u-boot/ common/init/board\_init.c

--> board\_init\_f u-boot/common/board\_f.c

**在SPL的上下文中，board\_init\_f（）准备从系统RAM（DRAM，DDR ...）**

**执行的硬件。 由于系统RAM可能尚不可用，因此board\_init\_f（）必须**

**使用当前的GD来存储必须传递到后续阶段的任何数据。 这些数据包括**

**重定位目标，将来的堆栈和将来的GD位置。 使用此功能后将清除BSS**

**（因此必须可访问）。**

--> relocate\_code u-boot/arch/arm/lib/relocate.S

**重定位监控程序代码**

--> relocate\_vectors u-boot/arch/arm/lib/ relocate.S

**将u-boot由当前位置重定位到由board\_inti\_f()函数所计算出的目的位置**

--> spl\_relocate\_stack\_gd u-boot/common/spl/spl.c

--> coloured\_LED\_init u-boot/ common/board\_f.c

--> red\_led\_on u-boot/ common / board\_f.c

--> board\_init\_r u-boot/common/board\_r.c

--> **init\_sequence\_r** u-boot/common/board\_r.c

**各种外围设备初始化，如：控制台初始化、网卡初始化等**

--> run\_main\_loop u-boot/common/board\_r.c

**实际执行在u-boot/include/initcall.h中**

--> main\_loop u-boot/common/main.c

**控制台命令操作体现在这里**

**2.4.4 uboot启动过程分析**

<https://www.cnblogs.com/linfeng-learning/p/9284060.html>

**重要** <https://blog.csdn.net/qq_16933601/article/details/106244510> 嵌入式Linux应用开发完全手册

<https://zhuanlan.zhihu.com/p/65853967>

<https://www.linuxidc.com/Linux/2017-02/141019.htm>

U-Boot的启动过程分为两个阶段：

第一阶段：主要是对处理器的内部资源的初始化(如时钟、串口)、内存(ddr)初始化，并进行 uboot 的重定位，板级的初始化比较少，所以移植的修改量比较小。此阶段由汇编语言编写，代码主体分布在start.S和lowlevel\_init.S中。

第二阶段：主要是板级的初始化，对处理器的外部资源(iNand、网卡芯片…)、uboot环境(uboot命令、环境变量..)等初始化，并等待命令输入，SOC内部的初始化比较少，移植的修改量主要在此。此阶段由c语言编写，代码主体分布在/uboot/lib\_arm/board.c中。

正常情况下，在 uboot 的初始化工作完毕后，会启动内核，在启动内核后结束 uboot 程序。

但是用户可以阻止 uboot 的结束，进入 uboot 的命令行模式，就是一个 uboot 中的死循环；uboot 在死循环中不断接受命令、解析命令、执行命令

uboot启动流程如下（该流程作为参看，以实际代码为准，如2.4.2的调用流程）:

1)设置CPU为管理模式

2)关看门狗

3)关中断

4)设置时钟频率

5)关mmu,初始化各个bank

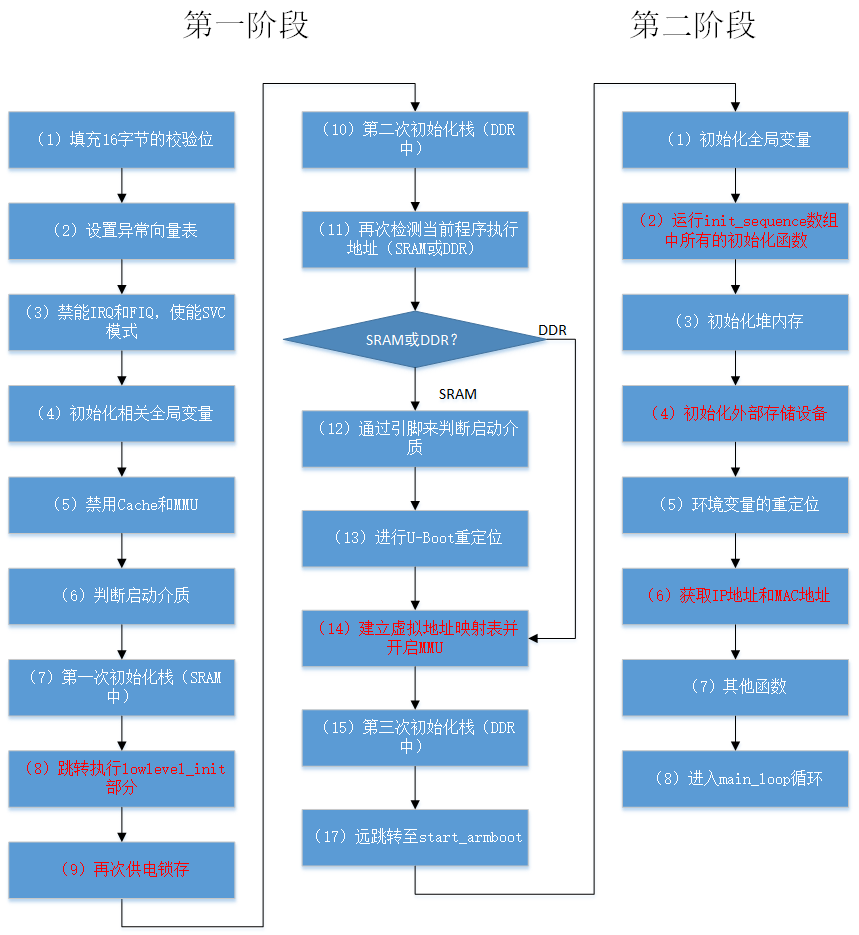
6)进入board\_init\_f()函数 (初始化定时器,GPIO,串口等,划分内存区域)

7)重定位 复制uboot,然后修改SDRAM上的uboot链接地址)

8)清bss

9)跳转到board\_init\_r()函数,启动流程结束

另一个可参考的启动流程：红色字体部分和板级关系较大，是移植的重点修改部分。



**2.5 u-boot 移植**

# 第四章 Linux内核

**3.1 架构介绍**

待完善

**3.1.1 字符设备**

设备注册

驱动注册

**3.1.2 platform 设备**

**3.1.3 网络设备**

**3.1.4 块设备**

**3.2 内核裁剪**

待完善

**3.3 源码（驱动）修改**

待完善

# 第五章 设备树修改及使用

# 第六章 根文件系统构建

# 第七章 buildroot构建Linux系统

以udoo-neo-full 为硬件平台的简单示例：

git://git.busybox.net/buildroot

make list-defconfigs | grep udoo

make mx6sx\_udoo\_neo\_defconfig

make all

# 第八章 yocto构建Linux系统