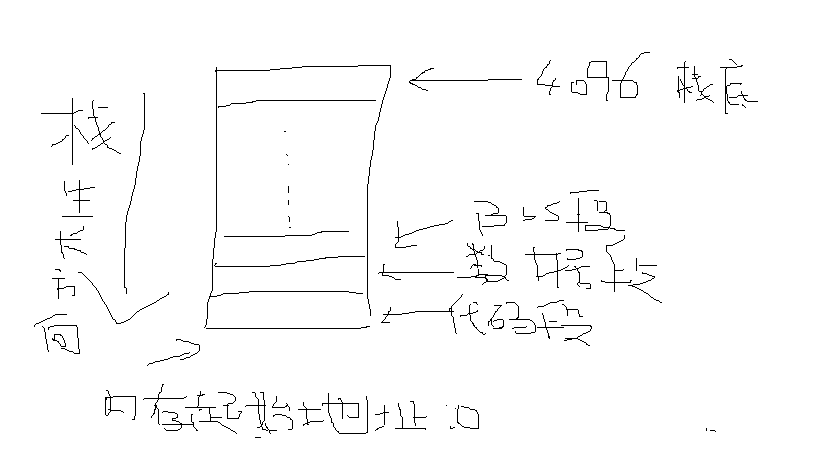
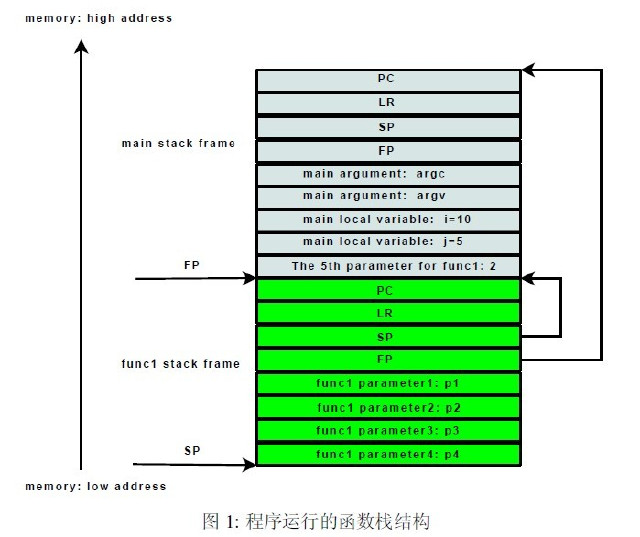
1. 栈结构

ARM的栈是自减栈，栈是向下生长的，也就是栈底处于高地址处，栈顶处于低地址处，所以栈区一般都放在内存的顶端，我这里偷懒，直接使用了2440 4K的片内内存，所以栈底就是片内内存的顶端地址：4096。



1. 函数的压栈与入栈操作



当函数main调用func1的时候其栈的过程如上图所示，每个函数都有自己的栈空间，这一部分我们称为栈帧，在函数被调用的时候创建，在函数返回后销毁。当main函数调用func1函数时，func1函数会先将PC、LR、SP、FP四个寄存器压到栈上边，其中SP和FP的值分别指向main函数栈帧的两个边界。LR的值保存的是func1调用结束之后的返回值，PC值表示的是当前执行到的指令地址，放置的是进入func1后的指令地址。紧接着就会在栈上分配一片区域，用于放置局部变量等。

如果func1中还调用了func2子函数，那么也会为func2创建一个栈帧，并且func2的SP和FP会指向func1栈帧的两个边界。这样当函数返回的时候，参数进行出栈，也能找到Caller函数，这个也就是backtrace的原理了。

pc, lr, sp, fp是处理器的寄存器，其含义如下：

pc, program counter，程序计数器。程序当前运行的指令会放入到pc寄存器中

fp, 即frame pointer,帧指针。通常指向一个函数的栈帧底部，表示一个函数栈的开始位置。

sp, stack pointer，栈顶指针。指向当前栈空间的顶部位置，当进行push和pop时会一起移动。

lr, link register。在进行函数调用时，会将函数返回后要执行的下一条指令放入lr中，对应x86架构下的返回地址。

调用栈从高地址向低地址增长，当函数调用时，分别将分别将pc, lr, ip和 fp寄存器压入栈中，然后移动sp指针，为当前程序开辟栈空间。

其中我们看到这其中涉及到四个比较关键的寄存器：PC、LR、SP、FP。需要注意的是，每个栈帧中的PC、LR、SP、FP都是寄存器的历史值，而不是当前值。

PC寄存器和LR寄存器均指向代码段，PC表示当前的代码指向到何处，LR表示当前函数返回后要到哪里去继续执行。

SP和FP用于维护函数的栈空间，其中SP指向栈顶，FP指向上一个函数栈帧的栈顶。

依次为当前函数指针PC、返回指针LR、栈指针SP、栈基址FP、传入参数个数及指针、本地变量和临时变量。如果函数准备调用另一个函数，跳转之前临时变量区先要保存另一个函数的参数。

1. 栈回溯

1、基于frame pointer进行unwind：

在栈回溯的过程中，我们主要是利用的是这个FP寄存器进行回溯，因为根据FP寄存器就可以找到下一个FP寄存器的栈底，获得PC指针，然后固定偏移，又可以回溯到上个PC指针，这样回溯下去，然后就可以完全的跟踪到函数的运行过程了。然后利用addr2line工具，就可以详细跟踪到函数的执行过程了。

当程序出现异常或者死机的时候，我们可以读取当前寄存器的状态，找到当前pc指针的情况，但是这些往往还不能说明问题，我们有时需要跟踪函数的执行过程。使用unwind的原理是记录每个函数的入栈指令到特殊的段.ARM.unwind\_idx .ARM.unwind\_tab。unwind回溯的过程可以总结为三部分：

1.根据pc找到函数unwind的段内存地址

2.根据unwind段中信息找到指令相关的编码数据

3.根据入栈地址，分析函数上一级的栈底保存的sp和lr。

对于arm32体系架构的backtrace基本原理可以参考如上的描述，其中最核心的部分是每个函数的栈中寄存器地址指向的是上个函数的地址，所以利用这个特性，就可以一级一级的跟踪下去，从而实现栈的回溯功能。这样我们在分析和定位问题的时候，就会更加的高效。

基于FP的缺点：

依赖编译器选项，编译的实现需要增加编译参数-fno-omit-frame-pointer 。

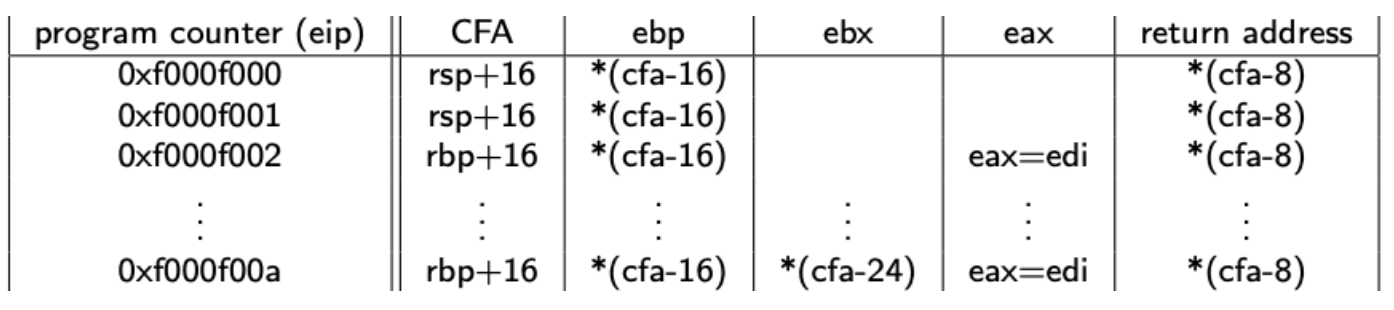
AArch32状态下，如果涉及到ARM和Thumb指令的时候，用FP不可靠。

1. 基于DWARF CFI进行unwind

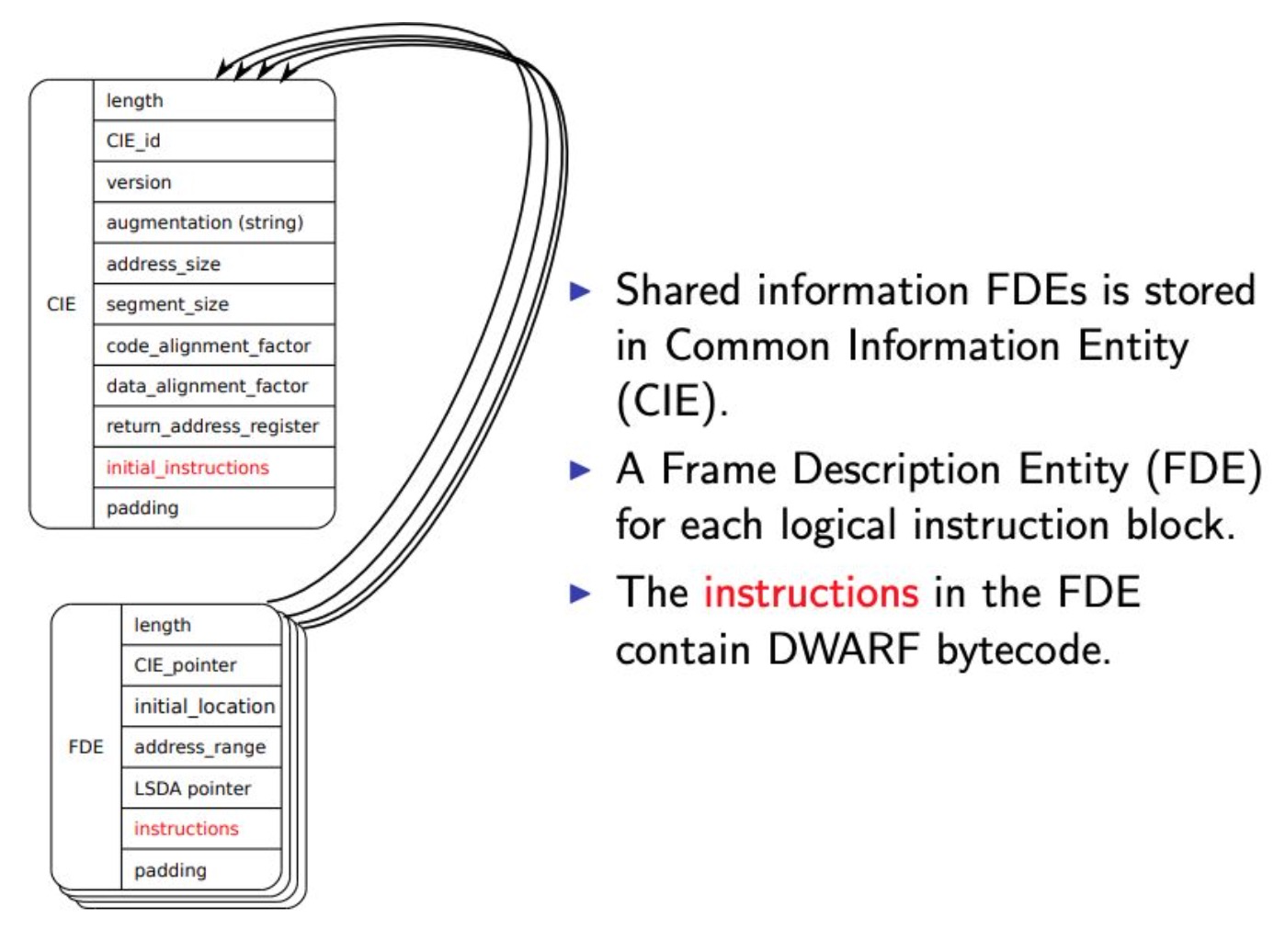
如果没有FP指针，那我们如何从stack上找的对应函数的调用链关系呢？在DWARF引入了一个新的 Call Frame Information段来辅助的调试信息。CFI信息被存储在.eh\_frame 段。从逻辑上来看，eh\_frame是一个table：

索引：每条指令的代码地址（every address in program text）

值： how to set registers to restore the previous call frame.



这个表是从逻辑上讲的，但是物理上是不存在的，因为如果按照这个表格存储占用的空间太大了。eh\_frame其实存储的是一种compact bytecode，需要一个解释器（例如android libunwindstack）去解释执行这些字节码，按需产生对应的表项。



CFI包括两类信息：CIE和FDE。在FDE中instructions即bytecode。

查看eh\_frame：

readelf --debug-dump=frames <xxx.so>

//产生上面那个表

readelf -wF <xxx.so>

基本unwind算法：

输入：当前线程pc寄存器

处理过程：

从.eh\_frame找的对应的 CIE/FDE的bytecode，依赖解释器去获取调用点各个寄存器的值找到return address所在的stack location，获取caller的pc

依次递归

输出：call-chain上的各个pc值。

Android上libunwindstack用C++实现了一个解释器算法。

在.eh\_frame只描述了unwind的必要信息，对于调试器和异常处理机制，还需要额外的一些调试信息，相关如下：

.debug\_line: code address -> source code locations

.debug\_info: source variables -> registers or stack locations

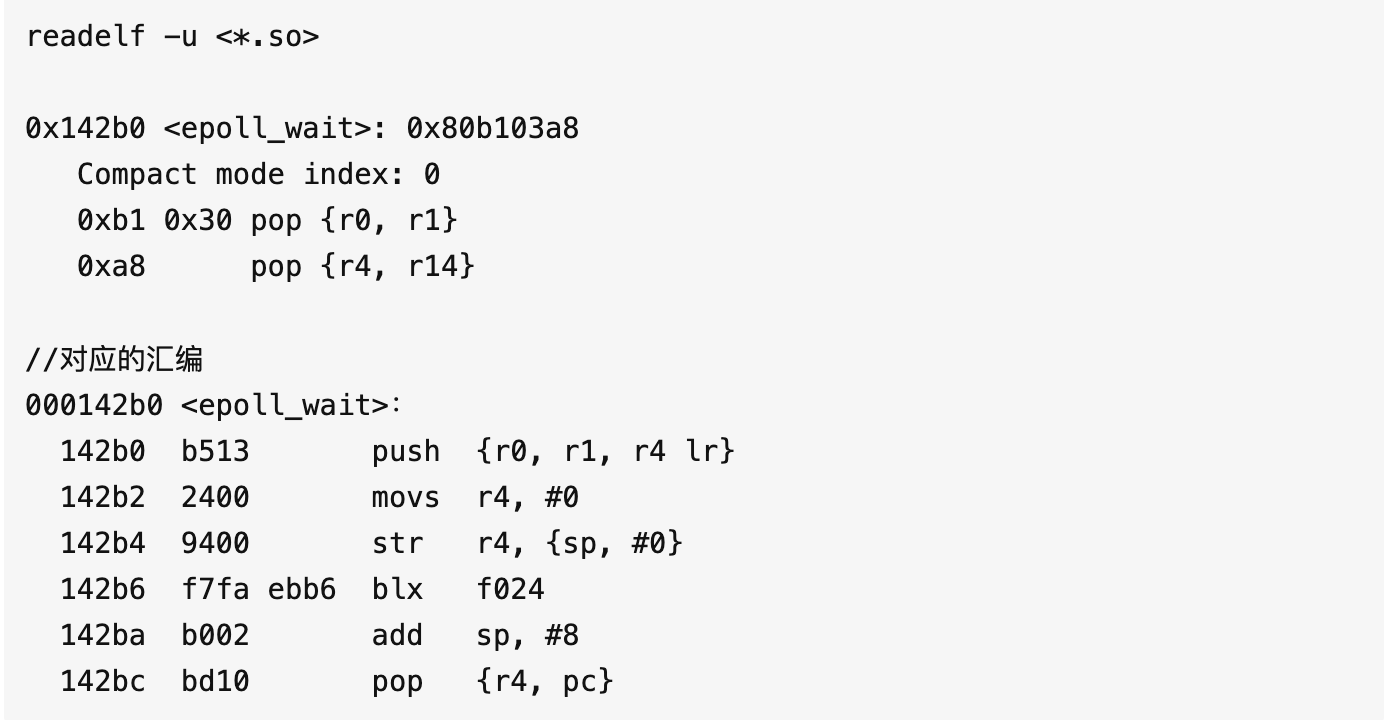
.gcc\_except\_table: a collection of language-specific data areas (LSDAs)

基于dwarf的unwind的缺点：

比较慢，并且不太可靠性，因此Google在一些关键代码和Linux内核仍旧使用FramePointer。

1. 基于exidx和extab的进行unwind

在arm32上，也提供了一种基于arm.exidx段的unwind方法。 exidx存储的信息可以通过：



exidx和extab里面实现原理跟eh\_frame差不多，也是通过一种bytecode依赖解释器执行模拟操作从stack上找到对应caller的pc。

在Android的libunwindstack也实现了基于exidx的unwind算法。

4.关于\_Unwind\_Backtrace实现

函数\_Unwind\_Backtrace在Android上是libgcc.a中提供的一个函数，是Itanium C++ ABI: Exception Handling定义的一部分。本身实现这个函数也是依赖Dawarf/exidx信息进行回溯stack的。

//aarch64

gcc/libgcc/unwind-dw2.cpp

//arm32

gcc/libgcc/config/arm/libunwind.S

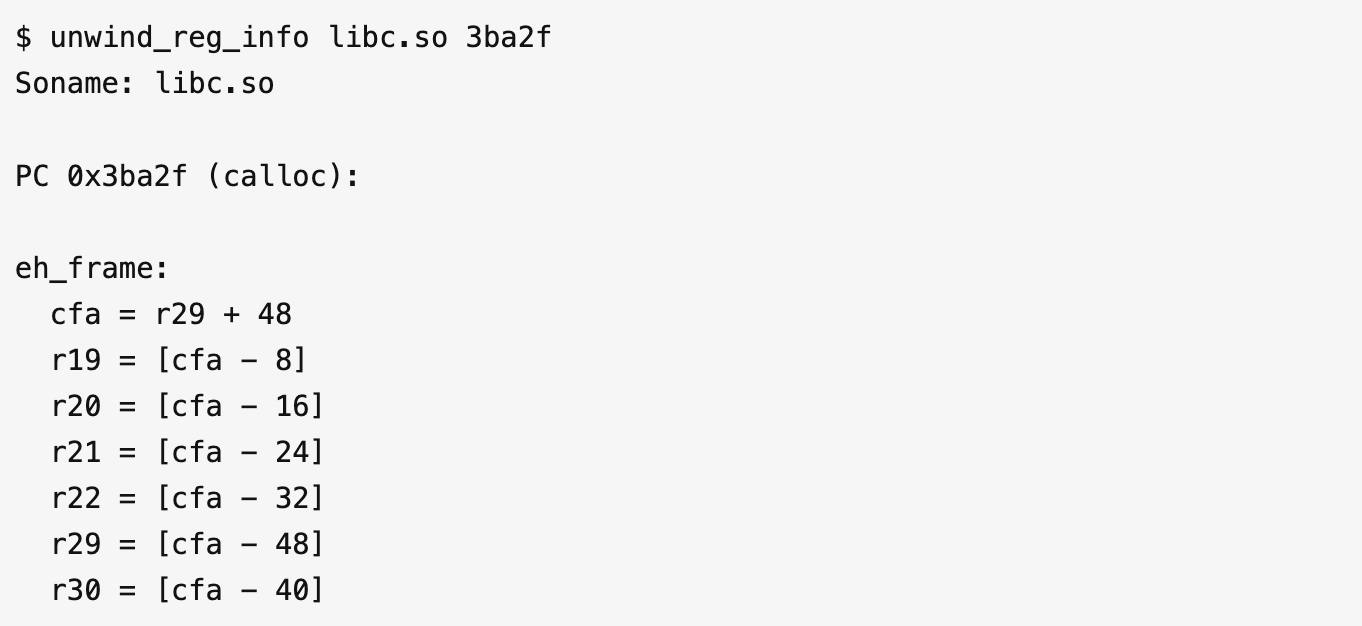
然后调用\_\_gnu\_Unwind\_Backtrace

1. libunwindstack

Android系统自己开发的unwind库，现在基本使用这个库进行stack unwind。

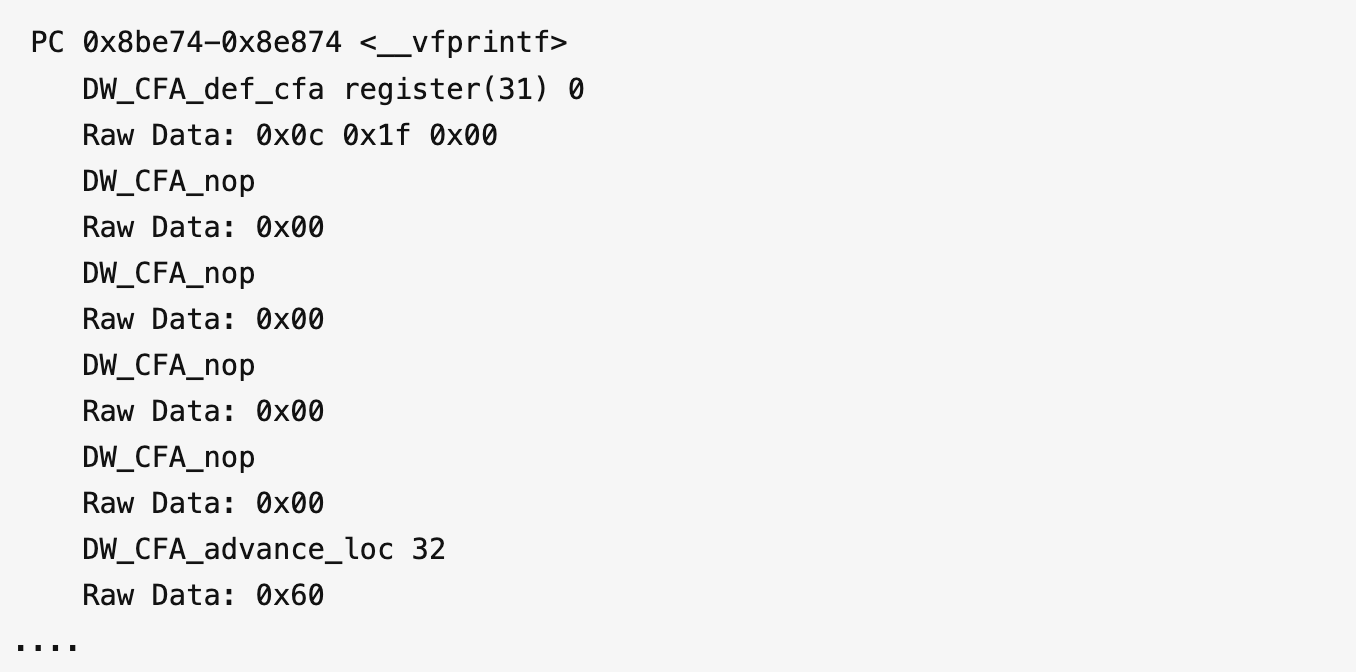
unwind\_reg\_info：

本身host和target上都可以用，可以根据pc展示出对应的寄存器的取值规则，例如calloc中：



unwind\_info：

显示一个elf中unwind信息：



unwind\_symbols：

显示一个elf中符号名称。

unwind：

可以显示一个进程的pid的调用stack，目前没有打印所有线程stack。

unwind执行流程：

for (; frames\_.size() < max\_frames\_;) {

1. 当前pc/sp

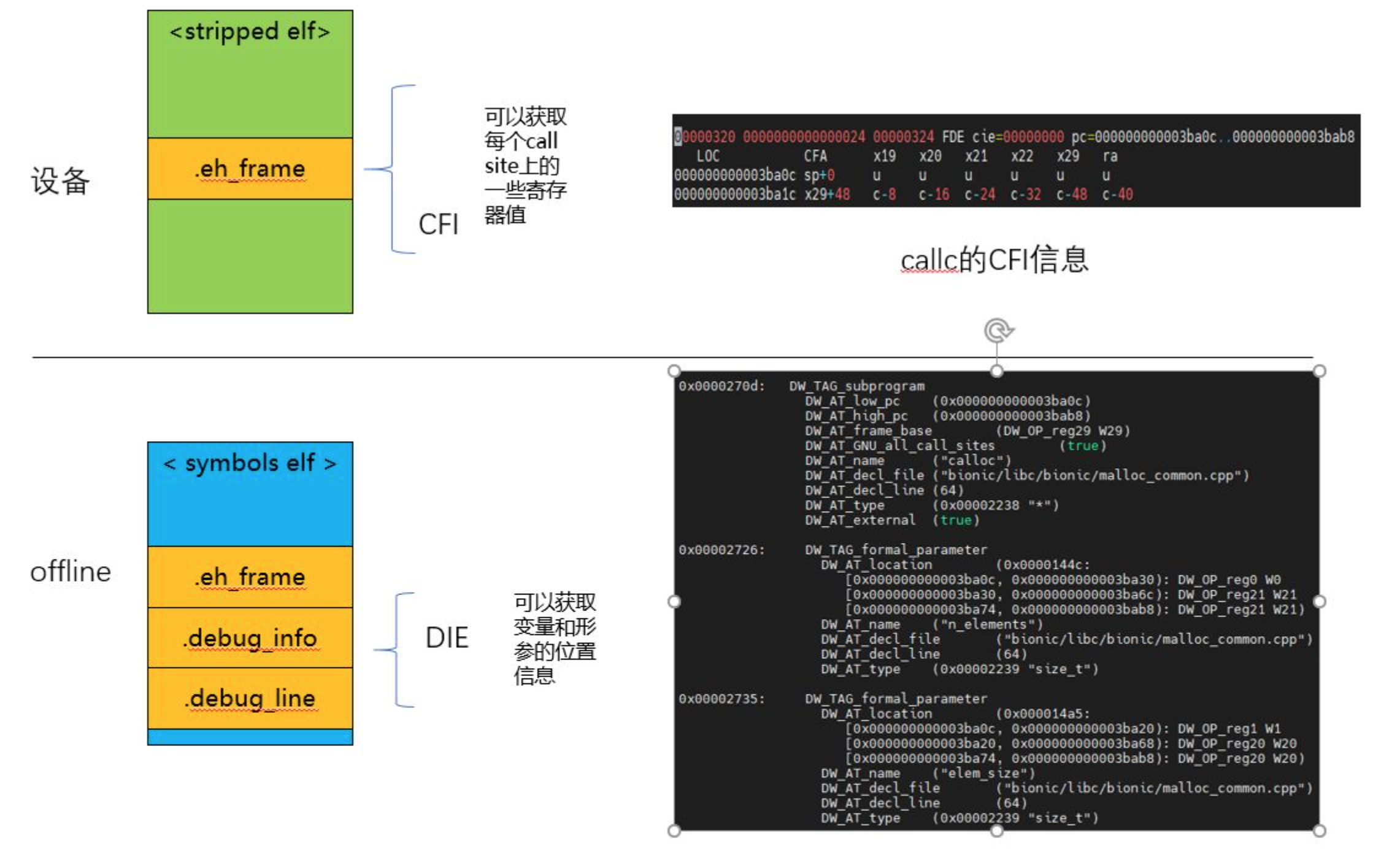
2. FillInDexFrame()和FillInFrame

3. 获取下一个frame的寄存器内容StepIfSignalHandler或者elf->Step

4. 如果没有错误，继续填充上一frame

}

android release版本下elf都有哪些信息：



设备上的elf文件，只携带eh\_frame/exidx信息，用于unwind库产生调用stack，并且可以恢复部分registers；symbols里面带着更多的信息，可以给addr2line等工具提供一些额外的调试信息，产生行号等。

我们期望根据CFI在unwind的时候可以获得部分寄存器值，然后更加DIE里面的一些变量TAG位置，找到部分形参和变量的值。现在Android平台上用的都是eh\_frame信息。

libunwindstack恢复寄存器过程：

libunwindstack库里面依赖Elf类去解析和读取对应的section：

eh\_frame的数据结构构成：CIE和FDE，一个ELF中有多少CIE/FDE组成通过.eh\_frame\_hdr 看多大的size计算。

ElfInterface::ReadSectionHeaders和ElfInterface::ReadProgramHeaders

根据PC获取FDE的相关函数：

DwarfSectionImpl<AddressType>::GetFdes(std::vector<const DwarfFde\*>\* fdes)

计算CFA location地址：根据Cie/Fde信息，获取各个pc的location loc\_reg。

bool DwarfSection::Step(uint64\_t pc, Regs\* regs, Memory\* process\_memory, bool\* finished)

->DwarfSectionImpl<AddressType>::GetCfaLocationInfo(pc, fde, dwarf\_loc\_regs\_t\* loc\_regs)

->DwarfCfa<AddressType>::GetLocationInfo

这个函数就是一个基于stack的cfa\_instructions解释器，从函数base pc开始算，

算出出pc对应寄存器地址位置信息及其操作数

根据DwarfLocation loc\_reg计算actual registers：

DwarfSection::Step(.....) {

.....

return Eval(it->second.cie, process\_memory, it->second, regs, finished);

}

根据cfa地址及其计算法则，计算真正的地址：

DwarfSectionImpl<AddressType>::Eval

1. 先获取CFA地址

-> DWARF\_LOCATION\_REGISTER

-> DWARF\_LOCATION\_VAL\_EXPRESSION

2. 再获取其他寄存器EvalRegister

目前libunwindstack库时候没有打印调用点寄存器值，那我们打印一下寄存器值：

1. FrameData增加记录每个Frame上面的寄存器：

std::unique\_ptr<unwindstack::Regs> regs

1. Unwinder::Unwind(）记录

frame->regs.reset(regs\_->Clone());

1. 打印

frame.regs->IterateRegisters([&data](const char\* name, uint64\_t value) {

data += android::base::StringPrintf(" %-3s %0\*" PRIx64, name,

static\_cast<int>(2 \* sizeof(void\*)),

static\_cast<uint64\_t>(value));

});