交叉编译原理及应用

一、程序的编译过程

什么是GCC：GNU Compiler Collection (GCC)

具体内容参考：程序的编译、链接、装载及运行

一段C/C++程序，经过编译器处理后变为可执行的二进制程序，都要经过下面4个过程：

> 1. 预处理：删除注释信息，宏展开，头文件文本插入源文件；使用的工具为cpp(C-compatible macro preprocessor)
> 2. 编译：对预处理的后的文件，进行词法、语法、语义分析，生成汇编文件；使用的工具为cc1(GNU C Compiler Internals)
> 3. 汇编：把汇编文件处理成为二进制文件，成为目标文件，目标文件与可执行文件结构只有细微差异，无法被直接执行，需要经过**链接**才能生成可执行文件；使用的工具为as(The GNU Assembler)
> 4. 链接：简单来说就是把所有需要的目标文件组装起来，形成一个可执行文件；使用的工具为ld(The GNU linker)；具体的：
>   - 合并不同目标文件中同类型的段
>   - 对于目标文件中的符号引用，在其它目标文件中找到可以引用的符号
>   - 对目标文件中的地址变量进行重定位

因此，我们平时说的“编译”，其实指的是以上4个步骤依次执行的过程，常说的编译“工具链”，其实是对“编译”过程中这4个工具依次链式执行的形象称呼。

二、交叉编译

交叉编译就是将**【高级语言S】**在支持**【机器语言A】**的机器上编译出**【机器语言B】**的程序的过程

问题：

> 假设这么一个场景，已知a机器支持的机器语言为A，b机器支持的机器语言为B。a机器拥有能把S>语言编译成A语言的编译器和一段S语言编写的服务程序。
>
> 问题1：如何在a机器生成b机器上能运行的服务程序?
> 问题2：如何在a机器生成b机器的S语言的编译器?

答：

我们约定这样描述一个编译器：<编译器自身使用的语言>（<源语言> -> <目标语言>）

已知a机器上存在一个S语言翻译到A语言的编译器，那么这个编译器本身是用A语言构成的，因此可以将a机器上的编译器表示为：A(S->A)

现在问题1可以转化为，**如何通过A(S->A)得到A(S->B)**；问题2转化为，**如何通过A(S->A)得到B(S->B)**

以上2个问题可以通过如下步骤解答：

1. 使用S语言编写编译器S(S->B)
2. 使用A(S->A)编译S(S->B)，得到A(S->B)，即A(S(S->B)->A) => A(S->B)，至此问题1解决
3. 使用A(S->B)编译S(S->B)，得到B(B->S)，即A(S(S->B)->B) => B(S->B)，至此问题2解决

类比x86交叉编译ARM架构程序的场景：S：C++，A：x86，B：ARM

1. 用C++编写编译器C++(C++->ARM)
2. 用x86(C++->x86)编译C++(C++->ARM)，得到x86(C++->ARM)
3. 用x86(C++->ARM)编译C++(C++->ARM)，得到ARM(C++->ARM)

三、交叉编译器的制作

详细流程请参见（需要梯子）：[如何制作ARM的交叉编译器]

必要的原材料(源码包)：

1. [Binutils(操作二进制文件的工具)]
2. [GCC]
3. [Linux kernel]
4. [Glibc(linux系统的底层运行时库)]

编译顺序：

1. 【Binutils包】编译交叉版本Bintuils
2. 【GCC包】编译交叉版本GCC
3. 【Glibc包+Linux kernel包】编译ARM版本内核
4. 【GCC包】编译ARM版本libgcc
5. 【Glibc包】编译ARM版本Glibc
6. 【GCC包】编译ARM版本libstdc++

虽然，目前大部分平台都会有现成交叉编译器供下载使用：
[x86-ARM交叉编译器下载]

但是理解上述步骤后，会对交叉编译器目录下的文件夹有深入的理解，如：

```
[dapp@armcompilevm gcc-linaro-10.2.1-2021.02-x86_64_aarch64-linux-gnu]$ ls
aarch64-linux-gnu  bin  gcc-linaro-10.2.1-2021.02-linux-manifest.txt  include
lib  libexec  share
```

aarch64-linux-gun文件夹下的是ARM平台下的各种库文件、头文件，用于程序的引用、链接
bin文件夹下是交叉编译版本的二进制工具如 aarch64-linux-gnu-gcc、 aarch64-linux-gnu-ld等
include 、lib、libexec是交叉编译工具需要依赖的二进制库头文件等

四、Nginx交叉编译

详情参见：
[Nginx交叉编译步骤]

常见问题：

1. 运行可执行文件后找不到动态链接库，怎么办?

   答：首先程序在链接分为2种：编译时链接和运行时链接。编译时链接用到的是链接器ld，而运行时链接用到的是动态链接加载器ld.so。

   可执行文件找不到动态链接库，其实也就是ld.so找不到对应的动态链接库，那也就是说只要在ld.so的搜索路径上把库加上即可。

   ld.so的搜索路径如下：rpath(编译时指定) -> LD_LIBRARY_PATH(环境变量) -> /etc/ld.so.conf -> /lib -> /usr/lib。

   rpath的作用就是指定程序运行时动态库的搜索路径，rpath的设置方式如下：

```
gcc -Wl,-rpath=/myLib main.c -o main
```

特别的，假如某程序涉及到了使用的Glibc与系统Glibc库不兼容，在不更换系统Glibc的情况下，个人理解只能使用rpath进行指定。因为其它的修改方式，都是所有程序可见的，会导致系统某些依赖系Glibc的程序不能正常执行。

2. 如何知道程序需要哪些动态库?

答：使用ldd命令，如下就展示出了myapp用的动态链接器为/lib64/ld-linux-x86-64.so.2，依赖的linux内核linux-vdso.so.1，依赖的动态链接库为libc.so.6

```
ldd myapp
    linux-vdso.so.1 (0x00007ffc34725000)
    libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f2761476000)
    /lib64/ld-linux-x86-64.so.2 (0x00007f2761753000)
```

3. 程序链接时报ABI相关的错误，怎么办?

   答：ABI全称为Application Binary Interface，与机器的位数相关，32位对应32位的ABI，64位对应64位的ABI。出现类似错误一般是目标文件之间的ABI类型不一致导致的，可在编译选项中使用-mabi指定。在龙芯交叉编译64位程序场景下，要注意的是需要在**编译**和**链接**过程中都要添加选项，因为该交叉编译默认以32位abi进行编译链接的。

4. OpenSSL编译出的二进制版本总是宿主平台的版本而不是目标平台版本?

   答：解决方案见问题链接中的问题6，即将脚本中的./config改成./Configure这一步。首先Configure是一份生成Makefile的shell脚本，它是GNU工具集的一员，有统一的标准；而config是Confiure的傻瓜版本，即该脚本能自动识别当前平台配置，并使用该配置来生成Makefile。OpenSSL的默认脚本中，使用的是config脚本，因此默认配置下，编译出来的二进制总是宿主平台的。

五、其他知识

5-1、编译器的自举（bootstrapping）

自举的定义:

> In computer science, bootstrapping is the technique for producing a self-compiling compiler — that is, a compiler (or assembler) written in the source programming

具体步骤:

> 已知A语言是一门成熟的语言，B语言是一门新开发的语言，下面展示B语言的自举过程:
>
> 1. 用A语言编写编译器A(B->二进制)，生成可执行文件记作A.exe，并留下一组测试用例
> 2. 用B语言编写编译器B(B->二进制)，用A.exe编译生成B.exe，并用B.exe验证所有测试用例
> 3. 用B.exe编译B(B->二进制)，生成B2.exe，并用B2.exe验证所有测试用例
> 4. 多次迭代验证后，最后用A.exe编译一遍B(B->二进制)得到B.exe，直接用B.exe编译B语言即可

5-2、程序的装载

程序的装载现在不再是直接装载到物理内存中，因为会由2个问题:

- 程序编写困难，需要判断当前地址是否被其他程序占用
- 其它程序修改内存数据，导致程序崩溃

因此虚拟内存机制应运而生，虚拟内存的地址与物理地址采用页映射的方式进行，通过页表进行管理。在虚拟内存的机制下，程序的装载步骤如下：

> 1. 为程序创建虚拟地址空间
> 2. 读取程序文件头，创建虚拟地址空间与可执行文件的映射。原因是，当程序执行发生页错误时，操作系统将从物理内存中分配一个物理页，然后将该“缺页”从磁盘中读取到内存中，再设置缺页的虚拟页和物理页的映射关系，这样程序才得以正常运行。
> 3. MMU翻译入口地址，并将翻译后的地址写入CPU指令寄存器，启动运行