编译与链接

1 编译与解释

编程语言可以分为编译型语言和解释型语言，编译型语言就是把源码通过编译器编译成机器码即二进制可执行文件，直接运行可执行文件就可以运行程序，例如c语言、golang、rust都是编译型语言。而解释型语言原则上不需要处理代码源文件，直接由解释器逐行解释代码来运行，例如python、js和java都是解释型语言。

对于java稍微说一句，java由javac“编译”成 `.class` 文件，并不是机器码，而是jvm的一种规范的文件形式，本质上是实现了一种统一的跨平台的文件格式规范，让解释器更统一的解释代码。但是jvm运行时的 `JIT` 及时编译，可以对常用的代码块在运行时编译成机器码，因为是运行时编译，可以更好的跨平台。所以严格来讲java是一种半解释半编译型语言。

2 动态链接与静态链接

以前学c语言老师让用vc++6.0，里面有三个重要的步骤：编译->连接->运行。链接是很重要的步骤，链接的本质是将用到的c语言库进行引入。两种连接方式分别是动态和静态的链接，其中最常用的是动态链接。

例如最简单的打印 `hello world` 程序如下， `stdio` 是标准的io库，他在libc这个库中，需要链接进来。

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

int main() {
    printf("hello world\n");
}
```

我们通过指令 `gcc -o gnu_c_dyn_hello hello.c` 将源码进行编译得到文件 `gnu_c_dyn_hello` 这里文件名是为了和后面的其他方式区分。如下图，我们很容易得到了编译和链接后的二进制文件，并将其运行起来了。其实我们没有特别去指定链接，gcc回到系统默认的libc库中去进行动态链接。通过 `ldd` 指令也可以看到动态链接的库名，和对应的所在的文件路径，这里默认是 `/lib/xxx` 这个路径下的so文件，linux下的动态链接库后缀是so，windows下是dll。

```
gitpod /workspace/notebook/22.11/compiler (master) $ gcc -o gnu_c_dyn_hello hello.c
gitpod /workspace/notebook/22.11/compiler (master) $ ./gnu_c_dyn_hello
hello world
gitpod /workspace/notebook/22.11/compiler (master) $ ldd gnu_c_dyn_hello
        linux-vdso.so.1 (0x00007ffeaf578000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fd6df6c1000)
        /lib64/ld-linux-x86-64.so.2 (0x00007fd6df8c2000)
```

动态链接的好处是，像 `libc.so` 的库是非常常见的，很多程序中都要用到，如果每个程序都将该功能引入，文件就会变大，且直接加载到内存也会占用较多的内存空间，而通过动态链接的方式，动态库是在共享的内存空间，所有的进程都用的函数在内存中是共享的，节省内存。但是动态链接的坏处也很显著，即如果把二进制文件放到没有安装对应的libc的linux中就无法运行了，会报共享库缺失的错误，此外共享库需要有很好的兼容性。

动态链接的缺点与优点刚好就和静态链接相反了，静态链接执行文件变大，内存占用变多，但是所有的功能都封到二进制文件中了，不需要依赖系统是否有合适的lib库。我们通过 `gcc -static -o gnu_c_static_hello hello.c` 得到静态链接的二进制文件，如下同样可以执行，并且我们看到静态链接的文件有870k而动态链接的只有16k。

```
gitpod /workspace/notebook/22.11/compiler (master) $ gcc -static -o gnu_c_static_hello hello.c
gitpod /workspace/notebook/22.11/compiler (master) $ ./gnu_c_static_hello
hello world
gitpod /workspace/notebook/22.11/compiler (master) $ ll | grep gnu
-rwxr-xr-x 1 gitpod gitpod  16696 Nov 26 10:20 gnu_c_dyn_hello*
-rwxr-xr-x 1 gitpod gitpod 871896 Nov 26 10:31 gnu_c_static_hello*
```

静态链接的文件可以在没有gnu库的linux下也能运行，例如我们启动一个alpine的docker来执行下这两个文件，动态链接的文件报错说 `not found` 并不是文件不存在，是这个动态链接的二进制找不到他链接的so文件或者找到了但是不太匹配，因而无法运行。

```
gitpod /workspace/notebook/22.11/compiler (master) $ docker run -it -v `pwd`:/app alpine sh
Unable to find image 'alpine:latest' locally
latest: Pulling from library/alpine
c158987b0551: Pull complete
Digest: sha256:8914eb54f968791faf6a8638949e480fef81e697984fba772b3976835194c6d4
Status: Downloaded newer image for alpine:latest
/ # /app/gnu_c_static_hello
hello world
/ # /app/gnu_c_dyn_hello
sh: /app/gnu_c_dyn_hello: not found
```

在alpine中运行ldd得到输出如下，可以看到动态链接的文件libc链接到了alpine系统中的so文件了，但是我们在ubuntu下编译用的是glibc和alpine中的musl libc其实并不相同，所以so文件并不匹配导致无法运行，下面会说gnu和musl。

```
/app # ldd gnu_c_static_hello
/lib/ld-musl-x86_64.so.1: gnu_c_static_hello: Not a valid dynamic program
/app # ldd gnu_c_dyn_hello
        /lib64/ld-linux-x86-64.so.2 (0x7f1aceb93000)
        libc.so.6 => /lib64/ld-linux-x86-64.so.2 (0x7f1aceb93000)
```

3 gnu与musl

c的库有很多gnu是名气最大的，很多发行版的linux，例如ubuntu、debian、centos等都基于gnu构建，gnu其实是一个大的组织，glibc是他的c库。

musl则是另一套，他的libc就叫musl libc，除了这俩还有其他的c库和toolchain，名气最大的肯定是gnu，而musl则是小巧精致，被内置在alpine系统中，在云原生时代，alpine才开始被广泛使用。所以我们就以这俩比较常见的libc为例，来对比相同平台，但是不同libc下构建产生的现象和原因吧。

3.1 gnu编译出来的文件->musl linux运行

这个其实我们在上面的例子已经看到了，即在alpine这个镜像中，运行gnu的两个可执行文件，现象是

- gnu静态编译的，可以在musl平台运行
- gnu动态编译的，因为依赖glibc的so文件，而无法再musl平台运行

3.2 musl编译出来的文件->gnu linux运行

先下载个muslcc的docker镜像，我们在镜像中进行操作。

```
docker pull muslcc/x86_64:x86_64-linux-musl
```

我们在docker中编译，静态和动态链接都能在本平台运行成功（如下图）

```
/app # gcc -o musl_c_dyn_hello hello.c
/app # ./musl_c
musl_c             musl_c_dyn_hello
/app # ./musl_c_dyn_hello
hello world
/app # gcc -static -o musl_c_static_hello hello.c
/app # ./musl_c_static_hello
hello world
```

我们把可执行文件放到宿主机GNU/Linux上运行如下

```
gitpod /workspace/notebook/22.11/compiler (master) $ ./musl_c_static_hello
hello world
gitpod /workspace/notebook/22.11/compiler (master) $ ./musl_c_dyn_hello
bash: ./musl_c_dyn_hello: No such file or directory
gitpod /workspace/notebook/22.11/compiler (master) $ ldd musl_c_static_hello
        statically linked
gitpod /workspace/notebook/22.11/compiler (master) $ ldd musl_c_dyn_hello
./musl_c_dyn_hello: error while loading shared libraries: /lib/x86_64-linux-gnu/libc.so: invalid ELF header
```

从上图可以得到和之前一样的结论，静态链接的放到其他libc环境的linux也能运行，其实没有libc的平台上也是能运行的，因为所有的库都封到二进制文件中了。

而动态链接的必须当前系统安装编译时候指定的libc动态链接so文件才行。

3.3 同时安装多个libc

从ldd指令能看到对应的so文件和映射的系统库的文件，一般都是 `/lib` 目录下，这也是系统的c库的默认安装的地方，我们能否同时安装多个libc呢，例如我们在ubuntu下能否同时安装glibc和musl libc，是不是就能同时运行两种动态编译出来的文件了呢？其实是这样。

```
# 安装gnu这一套，默认ubuntu已经有了
$ apt install build-essential gcc

# 安装musl相关的工具和库
$ apt install musl-tools
```

此时在ubuntu上运行musl的dyn文件也可以运行成功了!

```
gitpod /workspace/notebook/22.11/compiler (master) $ ./musl_c_dyn_hello
hello world
```

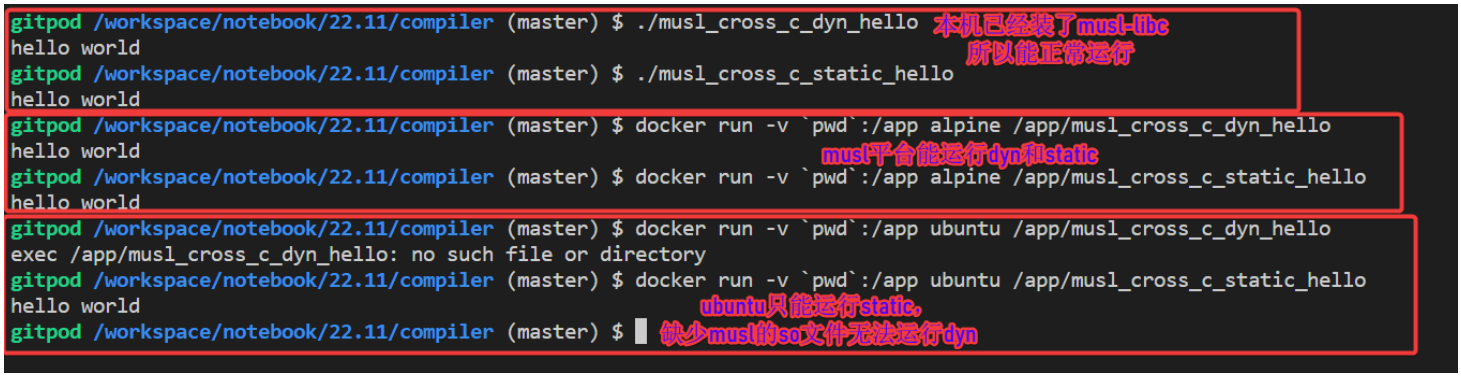
4 跨libc交叉编译

交叉编译一般是指跨平台，例如windows上开发编译出能在linux上运行的程序。但是这里我们要说的是在ubuntu上编译出动态依赖musl libc的二进制文件。

刚才我们已经安装了 `musl-tools` ， 其实不光安装了 `musl-libc` ， 也安装了 `musl-gcc`

```
$ musl-gcc -o musl_cross_c_dyn_hello hello.c
$ musl-gcc -static -o musl_cross_c_static_hello hello.c
```

因为已经安装了 `musl-libc` 所以在本机直接运行肯定是已经可以了。我们在单独的ubuntu里其实是无法运行动态链接的文件的。



5 golang的编译

golang编译和libc以及链接库也有关系，golang本身是c语言写的，本身也会用到很多c库，按理说通过静态链接可以把libc的依赖给屏蔽的，但是除了goSDK还会有其他第三方库可能也会用到c函数，因为golang提供了 CGO 机制，来使得golang能直接调用c的动态库，这为很多native的开发提供了方便。也有较高的效率，但是 CGO 会依赖动态so文件，不是很云原生。

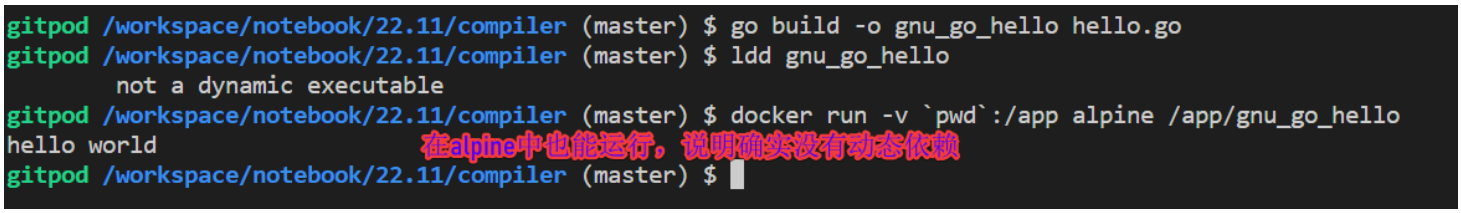
例如goSDK中的 os/user 还有 net 库默认就是调用了libc，尤其是 net 在很多上层应用场景还是很常用的，因而使用这两个库的时候，默认方式编译出来的二进制文件会依赖系统libc，我们来看一下吧。

先来看下不用上述两个库的时候，已经静态链接到goSDK中了，所以是不会依赖libc的。

```
// hello.go
package main

import "fmt"

func main() {
        fmt.Println("hello world")
}
```



接下来我们来使用下 os/user

```go
// user.go
package main

import "fmt"
import "os/user"

func main() {
        u, _ := user.Current()
        fmt.Println(u.Username)
}
```

```
gitpod /workspace/notebook/22.11/compiler (master) $ go build -o gnu_go_user user.go
gitpod /workspace/notebook/22.11/compiler (master) $ ./gnu_go_user
gitpod
gitpod /workspace/notebook/22.11/compiler (master) $ docker run -v `pwd`:/app alpine /app/gnu_go_user
exec /app/gnu_go_user: no such file or directory
gitpod /workspace/notebook/22.11/compiler (master) $ ldd gnu_go_user
        linux-vdso.so.1 (0x00007ffe21bf9000)
        libpthread.so.0 => /lib/x86_64-linux-gnu/libpthread.so.0 (0x00007fef4a091000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fef49e9f000)
        /lib64/ld-linux-x86-64.so.2 (0x00007fef4a0be000)
gitpod /workspace/notebook/22.11/compiler (master) $
```

此时确实动态链接了一些so，无法在alpine运行了

好在golang编译是可以关掉CGO的，并且强制指定编译时是静态链接，通过CGO_ENABLBED=0来关闭CGO，。

```
CGO_ENABLED=0 go build -o gnu_go_0_user user.go
```

一般来说静态链接还需要加上标志位 `--ldflags='-extldflags=-static'` ，他和cgo俩最好都加上，为啥要俩呢。参考这个Stack Overflow

```
gitpod /workspace/notebook/22.11/compiler (master) $ CGO_ENABLED=0 go build -o gnu_go_0_user user.go
gitpod /workspace/notebook/22.11/compiler (master) $ ldd gnu_go_0_user
        not a dynamic executable
gitpod /workspace/notebook/22.11/compiler (master) $ docker run -v `pwd`:/app alpine /app/gnu_go_0_user
root
gitpod /workspace/notebook/22.11/compiler (master) $
```

5.1 题外话：go的跨libc编译

CC环境变量可以指定c的编译工具链，例如在ubuntu默认的肯定是 `CC=x86_64-linux-gnu-gcc` ，可以指定为 `CC=x86_64-linux-musl-gcc` 来切换musl库，但是呢需要安装musl的工具链，我们之前虽然安装了 `musl-tool` ，但是似乎不太好使。我们自行下载musl的toolchain来支持跨CC编译。

在 `https://musl.cc/` 可以找到musl官方提供的toolchain，这个toolchain其实非常海纳百川，他不光提供了native支持，还提供了跨平台，比如linux x86想build出arm平台的也是可以的，我们暂不讨论跨平台编译方式。

下载解压

```
$ wget https://musl.cc/x86_64-linux-muslx32-native.tgz
$ tar zxvf x86_64-linux-muslx32-native.tgz
$ export PATH=$PATH:/path/to/x86_64-linux-muslx32-native/bin


CC=x86_64-linux-musl-gcc go build -o musl_go_user user.go
```

```
gitpod /workspace/notebook/22.11/compiler (master) $ export PATH=$PATH:/home/gitpod/x86_64-linux-musl-native/bin
gitpod /workspace/notebook/22.11/compiler (master) $ CC=x86_64-linux-musl-gcc go build -o musl_go_user user.go
gitpod /workspace/notebook/22.11/compiler (master) $ ./musl_go_user
gitpod 当前机器安装了musl所以能运行
gitpod /workspace/notebook/22.11/compiler (master) $ ldd musl_go_user
 /musl_go_user: error while loading shared libraries: /lib/x86_64-linux-gnu/libc.so: invalid ELF header
gitpod /workspace/notebook/22.11/compiler (master) $ docker run -v `pwd`:/app alpine /app/musl_go_user
root
gitpod /workspace/notebook/22.11/compiler (master) $ docker run -v `pwd`:/app ubuntu /app/musl_go_user
exec /app/musl_go_user: no such file or directory
gitpod /workspace/notebook/22.11/compiler (master) $ 在alpine上可以运行但ubuntu不能运行，
                                                     因为os/user用了cgo，默认开启，需要so
```

6 rust编译

rust是另一门编译型语言，他的toolchain安装更简单，rustup自己帮忙下载好，都不需要自己找musl库之类的。

默认cargo build使用的是动态链接:

```
gitpod /workspace/notebook/22.11/compiler/hello (master) $ cargo build --release
    Finished release [optimized] target(s) in 0.00s
gitpod /workspace/notebook/22.11/compiler/hello (master) $ ldd target/release/hello
        linux-vdso.so.1 (0x00007ffe56faa000)
        libgcc_s.so.1 => /lib/x86_64-linux-gnu/libgcc_s.so.1 (0x00007f7b301e4000)
        libpthread.so.0 => /lib/x86_64-linux-gnu/libpthread.so.0 (0x00007f7b301c1000)
        libdl.so.2 => /lib/x86_64-linux-gnu/libdl.so.2 (0x00007f7b301bb000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f7b2ffc9000)
        /lib64/ld-linux-x86-64.so.2 (0x00007f7b30259000)
```

添加标志位，使其静态链接，参考文档最后部分。

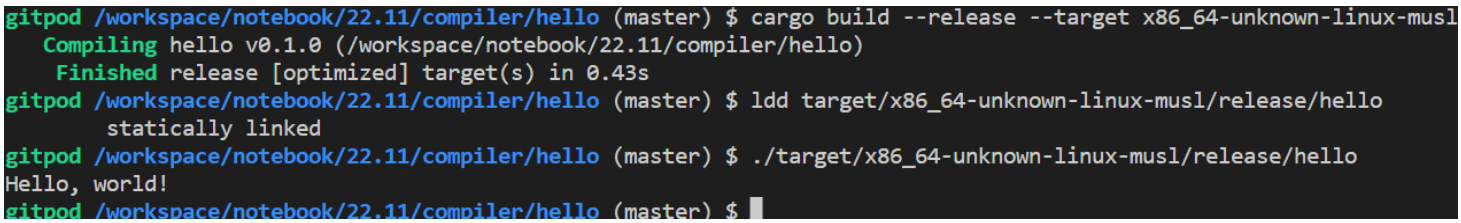
```
gitpod /workspace/notebook/22.11/compiler/hello (master) $ RUSTFLAGS='-C target-feature=+crt-static' cargo build --release
   Compiling hello v0.1.0 (/workspace/notebook/22.11/compiler/hello)
    Finished release [optimized] target(s) in 0.39s
gitpod /workspace/notebook/22.11/compiler/hello (master) $ ldd target/release/hello
        statically linked
gitpod /workspace/notebook/22.11/compiler/hello (master) $ docker run -v `pwd`:/app alpine /app/target/release/hello
Hello, world!
```

ubuntu下使用musl的toolchain默认也是静态链接的。

```
// 查看支持的目标平台列表，支持很多
$ rustup target list

// 添加musl平台相关的支持，细节不用自己管rustup自己把需要的东西下好
$ rustup target add x86_64-unknown-linux-musl

// 默认musl作为target就是静态链接的，不用再指定FLAG
$ cargo build --release --target x86_64-unknown-linux-musl
```



7 小结

libc是很重要的，因为libc是和linux交互的关键，可能不是写C语言，但大概率也会依赖libc。

gnu的glibc是一套性能不错也被广泛使用的libc库，musl在云原生时代逐渐被广泛使用。

将c库可以动态或者静态的链接进程序，动态链接是常见的也是建议的行为，但是云原生时代，一个容易内一般也就一个程序，似乎静态链接也成了小范围的一种趋势。

golang和rust是当前的编程界的宠儿，他们都是编译型语言，且都可以指定libc，当然也都支持交叉编译到其他平台。

比较程序的大小

```
// rust的静态链接的大概是四五M，动态链接是3.9M
5.6M    ./target/release/hello
4.1M    ./target/x86_64-unknown-linux-musl/release/hello

// c文件大小如下，都是1M以下
8.0K    musl_c_dyn_hello
20K     musl_cross_c_dyn_hello
28K     musl_cross_c_static_hello
16K     musl_c_static_hello
20K     gnu_c_dyn_hello
852K    gnu_c_static_hello

// golang只有不到2M，即使CGO=0的也不到2M
1.9M    gnu_go_0_user
1.8M    gnu_go_hello
1.9M    gnu_go_user
1.9M    musl_go_user
```