编译命令

生成目标文件：

目标文件包含了机器指令代码、数据，链接时需要的信息，符号表、调试信息，字符串表。

1. 不指定 `target` ，默认是 `Mach-O 64-bit object x86_64` ：

```
clang -x c -g -c a.c -o a.o
-x: 指定编译文件语言类型
-g: 生成调试信息
-c: 生成目标文件，只运行preprocess，compile，assemble，不链接
-o: 输出文件
-I<directory> 在指定目录寻找头文件
-L <dir> 指定库文件路径（.a\.dylib库文件）
-l<library_name> 指定链接的库文件名称（.a\.dylib库文件）
-F<directory> 在指定目录寻找framework头文件
-framework <framework_name> 在指定链接的framework名称
生成相应的LLVM文件格式，来进行链接时间优化
当我们配合着-S使用时，生成汇编语言文件。否则生成bitcode格式的目标文件
-flto=<value> 设置LTO的模式：full or thin
-flto  设置LTO的模式：full
-flto=full，默认值，单片（monolithic）LTO通过将所有输入合并到单个模块中来实现此目的
-flto=thin，使用ThinLTO代替
-emit-llvm
-install_name 指定动态库初次安装时的默认路径，向'LC_ID_DYLIB'添加安装路径，该路径作为dyld定位该库。
```

`clang -o` 是将 `.c` 源文件编译成为一个可执行的二进制代码(`-o` 选项其实是指定输出文件文件名，如果不加 `-c` 选项， `clang` 默认会编译链接生成可执行文件，文件的名称由 `-o` 选项指定)。

`clang -c` 是使用 `LLVM` 汇编器将源文件转化为目标代码。

2. 指定生成 `Mach-O 64-bit x86-64` 目标文件格式：

```
clang -x c -target x86_64-apple-macos10.15 -g -c a.c -o a.o
```

3. 如果指定 `target` 不带 `apple` 系统版本（包括 `macOS`， `ipadOS`， `iOS`，真机和模拟器）。例如 `x86_64`，那么生成的目标文件是 `Linux` 的 `ELF 64-bit`：

```
clang -x c -target x86_64  -g -c a.c -o a.o
```

4. 编译 `.m`：

```
clang -x objective-c -target x86_64-apple-macos10.15 -fobjc-arc -fmodules -isysroot /Applications/Xcode.app/Contents/Developer/Platforms/MacOSX.platform/Developer/SDKs/MacOSX10.15.sdk -c test.m -o test.o
clang -x c -g -target arm64-apple-ios13.5 -isysroot /Applications/Xcode.app/Contents/Developer/Platforms/iPhoneOS.platform/Developer/SDKs/iPhoneOS13.6.sdk -c a.c -o a.o
```

5. 编译 `.mm`：

   在 `mac` 上编译：

```
clang -x objective-c++  -target x86_64-apple-macos10.15 -std=c++11 -stdlib=libc++ -fobjc-arc -fmodules -isysroot /Applications/Xcode.app/Contents/Developer/Platforms/MacOSX.platform/Developer/SDKs/MacOSX10.15.sdk -c test.mm -o test.o
```

   在模拟器上编译：

```
clang -x objective-c -target x86_64-apple-ios13.5-simulator -fobjc-arc -fmodules -isysroot /Applications/Xcod
```

```
e.app/Contents/Developer/Platforms/iPhoneSimulator.platform/Developer/SDKs/iPhoneSimulator13.6.sdk -c test.m
-o test.o
```

在模拟器上链接其他三方库：

```
clang -x objective-c -target x86_64-apple-ios13.5-simulator -fobjc-arc -fmodules -isysroot /Applications/Xcod
e.app/Contents/Developer/Platforms/iPhoneSimulator.platform/Developer/SDKs/iPhoneSimulator13.6.sdk -I/Users/w
s/Desktop/课程/Library/代码-库/AFNetworking.framework/Headers  -F/Users/ws/Desktop/课程/Library/代码-库 -c test.
m -o test.o
clang -target x86_64-apple-ios13.5-simulator -isysroot /Applications/Xcode.app/Contents/Developer/Platforms/i
PhoneSimulator.platform/Developer/SDKs/iPhoneSimulator13.6.sdk -F/Users/ws/Desktop/课程/Library/代码-库 -fobjc-
arc -framework AFNetworking -v  test.o -o test
clang -target x86_64-apple-ios13.5-simulator -isysroot /Applications/Xcode.app/Contents/Developer/Platforms/i
PhoneSimulator.platform/Developer/SDKs/iPhoneSimulator13.6.sdk -L/Users/ws/Desktop/课程/Library/代码-库 -fobjc-
arc -lAFNetworking -dead-strip  test.o -o test
```

编译成 `arm64` 真机：

```
clang -target arm64-apple-ios13.5 -isysroot /Applications/Xcode.app/Contents/Developer/Platforms/iPhoneOS.pla
tform/Developer/SDKs/iPhoneOS13.6.sdk  -L/Users/ws/Desktop/课程/Library/代码-库 -fobjc-arc -lAFNetworking  test
.o -o test
clang -target arm64-apple-ios13.5 -isysroot /Applications/Xcode.app/Contents/Developer/Platforms/iPhoneOS.pla
tform/Developer/SDKs/iPhoneOS13.6.sdk -F/Users/ws/Desktop/课程/Library/代码-库 -fobjc-arc -framework AFNetworki
ng test.o -o test
```

6. 生成 `dSYM` 文件：

```
clang -x c -g1 a.c -o a.o
-g1: 将调试信息写入`DWARF`格式文件
```

查看调试信息

`dwarfdump` 取出并验证 `DWARF` 格式调试信息：

```
dwarfdump a.o
dwarfdump a.dSYM
dwarfdump --lookup 0x100000f20 --arch=x86_64 a.dSYM

--lookup 查看地址的调试信息。将显示出所在的目录，文件，函数等信息
```

查看文件内容

`otool` 用来查看 `Mach-o` 文件内部结构:

```
otool -l liba.dylib
otool -h libTest.a

-l: 显示解析后的mach header和load command
-h: 显示未解析的mach header
-L: 打印所有链接的动态库路径
-D: 打印当前动态库的`install_name`
```

`objdump` 用来查看文件内部结构，包括 `ELF` 和 `Mach-o`：

```
objdump --macho -h a.o
objdump --macho -x a.o
objdump --macho -s -d a.o
objdump --macho --syms a.o

```

```
--macho: 指定Mach-o类型
-h: 打印各个段的基本信息
-x: 打印各个段更详细的信息
-d: 将所有包含指定的段反汇编
-s: 将所有段的内容以16进制的方式打印出来
--lazy-bind: 打印lazy binding info
--syms 打印符号表
```

静态库的压缩和解压缩

`ar` 压缩目标文件，并对其进行编号和索引，形成静态库。同时也可以解压缩静态库，查看有哪些目标文件：

```
ar -rc a.a a.o
-r: 添加or替换文件
-c: 不输出任何信息
-t: 列出包含的目标文件
```

创建静态库

创建库命令： `libtool` 。可以创建静态库和动态库:

```
libtool -static -arch_only x86_64 a.o -o a.a

libtool -static -arch_only arm64 -D -syslibroot /Applications/Xcode.app/Contents/Developer/Platforms/iPhoneOS.platform/Developer/SDKs/iPhoneOS13.6.sdk test.o -o libTest.a
```

创建动态库

```
clang -dynamiclib -target arm64-apple-ios13.5 -isysroot /Applications/Xcode.app/Contents/Developer/Platforms/iPhoneOS.platform/Developer/SDKs/iPhoneOS13.6.sdk a.o -o a.dylib
```

查看符号表

`nm` 命令：

```
nm -pa a.o
-a: 显示符号表的所有内容
-g: 显示全局符号
-p: 不排序。显示符号表本来的顺序
-r: 逆转顺序
-u: 显示未定义符号
```

生成dSYM文件

`dsymutil` 可以被理解为是调试信息链接器。它按照上面的步骤执行：

- 读取 `debug map`
- 从.o文件中加载DWARF
- 重新定位所有地址
- 最后将全部的 `DWARF` 打包成 `dSYM Bundle`
  有了 `dSYM` 后，我们就拥有了最标准的 `DWARF` 的文件，任何可以 `dwarf` 读取工具（可以处理 `Mach-O` 二进制文件）都可以处理该标准 `DWARF` 。

`dsymutil` 操作 `DWARF` 格式的 `debug symbol` 。可以将可执行文件 `debug symbol` 的生成 `DWARF` 格式的文件：

```
dsymutil -f a -o a.dSYM

```

```
-f: .dwarf格式文件
-o <filename>: 输出.dSYM格式文件
```

移除符号

`strip` 用来移除和修改符号表:

```
strip -S a.o

-S 删除调试符号
-X 移除本地符号，‘L’开头的
-x 移除全部的本地符号，只保留全局符号
```

链接器

ld

```
-all_load 加载静态库的包含的所有文件。

-ObjC 加载静态库的包含的所有义的Objective-C类和Category.

-force_load <path_to_archive> 加载静态库中指定的文件
```

链接动态库与静态库

```
ld -dylib -arch x86_64 -macosx_version_min 10.13 a.dylib -o a

```

```
ld -static -arch x86_64 -e _main  a.a -o a
```

Xcode打印加载的库

`Pre-main Time` 指 `main` 函数执行之前的加载时间，包括 `dylib` 动态库加载， `Mach-O` 文件加载， `Rebase/Binding` ，`Objective-C Runtime` 加载等。

`Xcode` 自身提供了一个在控制台打印这些时间的方法：在 `Xcode` 中 `Edit Scheme -> Run -> Auguments` 添加环境变量 `DYLD_PRINT_STATISTICS` 并把其值设为 1。

`DYLD_PRINT_LIBRARIES` : 打印出所有被加载的库。

`DYLD_PRINT_LIBRARIES_POST_LAUNCH` : 打印的是通过 dlopen 调用返回的库，包括动态库的依赖库，主要发生在 main 函数运行之后。

二进制重排

链接order.file

```
    ld -o test test.o -lsystem -order_file test.order

    ld -o test test.o -lsystem -lc++ -framework Foundation  -order_file test.order

    ld -map output.map -lsystem  -o output a.o
```

生成Link Map 文件

```
ld -map output.map -lsystem -lc++  -framework Foundation  test.o  -o output
```

```
-map map_file_path 生成map文件。
主要包括三大部分：

Object Files：  生成二进制用到的 link 单元的路径和文件编号
Sections：  记录 Mach-O 每个 Segment/section 的地址范围
Symbols：  按顺序记录每个符号的地址范围
```

`install_name_tool`

更改动态共享库的安装名称并操纵运行路径.

```
install_name_tool -add_rpath <directory> libs_File
install_name_tool -delete_rpath <directory> libs_File
install_name_tool -rpath <old> <new> libs_File
```

生成目标文件的过程中发生了什么?

编译器（clang-cl）  汇编器（llvm-as）

链接器(llvm-ld)并没有被执行

所以输出的目标文件不会包含Unix程序在被装载和执行时所必须的包含信息，但它以后可以被链接到一个程序。

Mach-o File Format

一个Mach-o文件有两部分组成：header 和data。

header：代表了文件的映射，描述了文件的内容以及文件所有内容所在的位置。

data：紧跟header之后，由多个二进制组成，one by one。

Load Commands

进制文件加载进内存要执行的一些指令。

这里的指令主要在负责我们 APP 对应进程的创建和基本设置（分配虚拟内存，创建主线程，处理代码签名/加密的工作），然后对动态链接库（.dylib 系统库和我们自己创建的动态库）进行库加载和符号解析的工作。

Mach-o File Format

一个Mach-o文件有两部分组成：header 和data。

header：包含三种类型。Mach header，segment，sections

header内的section描述了对应的二进制信息。

注意⚠️：Mach header属于header的一部分，它包含了整个文件的信息和segment信息。

Mach-o File Format

一个Mach-o文件有两部分组成：header 和data。

Segments(segment commands): 指定操作系统应该将Segments加载到内存中的什么位置，以及为该Segments分配的字节数。还指定文件中的哪些字节属于该Segments，以及文件包含多少 sections。始终是4096字节或4 KB的倍数，其中4096字节是最小大小。

Section: 所有sections都在每个segment之后一个接一个地描述。sections里面定义其名称，在内存中的地址，大小，文件中section数据的偏移量和segment名称。