

# 库 (http://casatwy.com/ku.html)

**Date** 📅 Thu 05 March 2015 **Tags** shared library (<http://casatwy.com/tag/shared-library.html>) / dynamic linked library (<http://casatwy.com/tag/dynamic-linked-library.html>) / static library (<http://casatwy.com/tag/static-library.html>) / dynamic load library (<http://casatwy.com/tag/dynamic-load-library.html>)

## 啰嗦一下

模块化开发不光要求代码级的模块化，比如区分各种功能，然后把功能的实现分散在各个对象或文件。大部分情况也要求部署级的模块化，使之能够通过使用库的方式模块化加载，模块化部署。在这一波"模块化"大潮中，各种不同类型的库被广泛使用着。如果你希望你的程序能够支持插件功能，其本质也依旧是通过库来实现。

其实库的本质是一堆函数实现的集合，他们被编译在某一个文件里，然后根据使用方式的不同，分为 静态库 和 动态库 两类。在 动态库 里，又分为 动态链接库 和 动态加载库。

## 静态库 vs 动态库

一般来说，build一个项目的过程是先compile然后再link，然后才有一个可执行文件。link的时候要做的一件事情就是把各种函数符号转换成函数调用地址，然后最终生成的可执行文件就能够直接调用到函数了。

静态库是在build的时候就把库里面的代码链接进可执行文件

动态库的做法跟静态库的做法不一样，不会在build的时候就把代码link进可执行文件。然而对于两种不同的动态库而言，它们又有所区别：

- 对于动态链接库而言，build可执行文件的时候需要指定它依赖哪些库，当可执行文件运行时，如果操作系统没有加载过这些库，那就会把这些库随着可执行文件的加载而加载进内存中，供可执行程序运行。如果多个可执行文件依赖同一个动态链接库，那么内存中只会有一份动态链接库的代码，然后把它共享给所有相关可执行文件的进程使用，所以它也叫共享库(shared library)。比如pthread就是

### 🏠 Social

📺 RSS (<http://casatwy.com/feeds/all.atom.xml>)

🐙 github (<http://github.com/casatwy>)

📘 facebook (<https://www.facebook.com/taloyum>)

👤 google+ (<https://plus.google.com/u/0/108264119649922067163>)

👤 weibo (<http://weibo.com/casatwy>)

### 🔖 Tags

(<http://casatwy.com/>)

### 🔗 Links

casatwy (<http://casatwy.com/>)

刘坤的技术博客 (<http://blog.cnbluebox.com>)

齐道长的博客 (<http://qitaos.github.io>)

一个这样的库。

- 对于动态加载库而言，build可执行文件的时候就不需要指定它依赖哪些库，当可执行文件运行时，如果需要加载某个库，就用 `dlopen`、`dlsym`、`dlclose` 等函数来动态地把库加载到内存，并调用库里面的函数。各大软件的插件模块基本上就都是这样的库。事实上，静态库和动态链接库也可以被动态加载，只是由于使用方式的不同，才多了一个 动态加载库 这样的类别。

他们的区别说白了就是加载库的时机，动态链接库在可执行文件得到运行的时候就加载，动态加载库在可执行文件运行期间的任何一个阶段都可以动态加载。大部分情况业界推荐使用动态库，至于动态链接还是动态加载，那就可以根据具体需要来。推荐更多使用动态库的原因如下：

- 在静态库方案下，库的版本更新之后，需要重新编译程序，才能使得更新后的代码起作用。而动态库只需要编译对应的库代码，然后重启程序即可。
- 使用动态库能够减少可执行文件的体积，因为公共功能被独立了出来，使得每个相关可执行文件不必把这部分代码也编入，从而减小了体积。

但是使用动态库也会带来一个缺点，那就是debug的时候特么超级麻烦。我个人是喜欢在debug的时候不使用库方案来生成可执行文件去debug，直接用 `.o` 文件链接生成可执行文件。部署的时候才会使用动态链接方案生成可执行文件。这两者的区别就只是几条编译指令的区别而已，在makefile里面写好就好了。

## 孔乙己

我在翻阅各种资料的时候，发现关于各种库的术语有特别多，我觉得有必要在这里做个辨析，如果大家都已经熟悉了那就可以跳过这一节。

关于常见的 DLL，有的时候大家会把它理解成 Dynamic Linked Library (动态链接库)，有的时候会把它理解成 Dynamic Load Library (动态加载库)，这个就要自己看上下文去区分了。

关于 Shared Library，那就是指 Dynamic Load Library，`so` (Shared Object)也是 Dynamic Load Library。

Linux/Unix下动态库文件的后缀名大多是 `so`，`s`，`dylib`。静态库文件后缀名多数是 `a`。windows我不熟悉也不知道也不愿意去查。

## 准备工作

会写C会写编译命令。

## 静态库

```
...  
clang -c log.c -o log.o  
clang -c memory.c -o memory.o  
ar rcs libmylib.a log.o memory.o  
...
```

静态库本质上就是一堆函数的集合，所以把相关文件编译成 .o 文件之后，就可以使用 ar 来把这些文件集成 .a 文件。ar rcs libmylib.a log.o memory.o 的意思就是把 log.o 和 memory.o 塞进 libmylib.a 里面，事实上一个静态库就是各种 .o 文件的集合。

当你生成好 libmylib.a 之后，就可以把它加进去了：

```
...  
clang -o demo.run demo.c -lmylib -L/your/lib/path  
or  
clang -o demo.run demo.c /path/to/your/static/library.a  
...
```

要注意的地方：

- 要是不加 -L 来添加库的搜索路径的话，会找不到你的静态库。因为编译器会跑去系统默认的路径去找，然而默认路径并不包括你的当前目录。
- -l 和 -L 要放在 demo.c 的后面，否则就会报莫名其妙的错误。
- -l 参数会自动帮你在前面补上 lib，在后面补上 .a。
- 你要是直接用 ld 来链接也可以，但是 ld 的参数接口经常会变，所以还是让编译器去链接吧。

其实使用静态库的场合不多，大多数时候是作为第三方提供SDK给别人，但又不希望别人看到源代码，才会用静态库交付。但是在iOS开发领域，静态库用得还是蛮多的，当一个app是由很多业务线组成的时候，最好还是交付静态库而不是代码，这样能节省很多编译的时间。

## 动态链接库

动态链接库是动态库的一种(先这么区分吧, 因为静态库也能动态加载), 我们也习惯叫它共享库(Shared Library), 当程序加载进内存的时候, 动态加载库也会跟着被加载进内存。当动态加载库加载到内存之后, 如果后面的程序也起来了, 而且也依赖这个动态加载库的话, 就不会重复加载。

动态链接库相对于静态库来说更加灵活和复杂, 因为在实际应用的时候对动态链接库会有以下要求:

- 更新动态库之后, 依然需要支持那些需要依赖旧版本动态库的程序的正常运行。
- 当程序运行的时候, 需要允许覆盖特定的库, 甚至特定的函数。
- 在程序使用现有库运行时, 依然能够需要支持以上两点。

为了达到以上目的, 业界制定了一套规范, 这套规范主要从两个方面着手:

- 命名规范。一个动态库会有不同的名字, 他们分别起到了不同的作用。
- 路径规范。一个动态库要放在特定路径下, 内核才能够在加载的时候去这个特定路径找到这个动态链接库。

## 命名规范

一个库有三个分别起到不同作用的名字: `soname`, `real name`, `link name`。在 Mac OSX 系统下, `soname` 又被称为 `install_name`

### **soname**

其实就是 `Shared Object NAME`, 在 MAC OSX 下是 `install_name`。

这个名字的规范就是 `lib + 库名 + so + 大版本号`, 它是用来标示动态链接库的主版本的, 用于给内核加载动态库选择版本时提供参考。`soname`是在编译的时候传递给链接器的, 如果你不在编译命令里面设置`soname`的话, 默认会用生成的文件名作为`soname`, 这不是个好的做法, 原则上都是要设置一下`soname`的。编译之后, `soname`就会被写入到你的库文件里面去, 你用 `vim` 直接打开库文件就能看到这个库的`soname`, 它跟你 `ls` 时候看到的文件名不是同一个东西。

另外, 可执行文件具体依赖哪样的库也是在编译的时候被编译进可执行文件的, 编译命令会找到对应的库, 并把这个库的`soname`转存到可执行文件的依赖列表中。

举个例子: 你有一个库, 库名叫做`casa`, 然后当前版本号是1, 那么它的`soname`就应该是 `libcasa.so.1`。

`soname`是怎么起作用的呢, 先不着急, 我会在实例的地方讲这个。

### **real name**

其实就是真实的文件名，命名规范就是 soname + 小版本号。它是你动态库的文件名，也就是你 ls 的时候看到的那个。以上面的 libcasa.so.1 为例，它的real name就是 soname再加小版本号，可以是 libcasa.so.1.0 或者 libcasa.so.1.1 这样。

它的具体作用就是让内核加载动态库时使用的名字，所以必须是文件名。

### link name

我们在build一个可执行文件的时候，需要在可执行文件里面记录这个可执行文件依赖于哪些动态库，这样内核在加载可执行文件的时候，才知道有哪些动态库需要加载。在写这条编译命令的时候，是不需要带版本号的。

还是举soname是 libcasa.so.1 的库为例子，它的link name就是 libcasa.so。具体编译指令长这样：

```
```\n    clang -o demo.run -c demo.c -L./ -lcasa\n```\n
```

因为有命名规范，所以链接器会自动在前面加上 lib，在后面加上 .so，这样我们就只要给出库名字 casa 就好了。

## 路径规范

由于动态加载库是动态加载的，当一个可执行文件被执行的时候，系统就要能够找得到它依赖的动态库在哪里。一般情况下，GNU标准要求系统去 /usr/local/lib 这个地方找。在FHS(Filesystem Hierarchy Standard)中，对这个又做了很多规定：大多数内核依赖的动态库应该被放在 /usr/lib 下，在内核启动时就依赖的动态库应该放在 /lib 下，然后不属于内核依赖的动态库才放在 /usr/local/lib 下。

这两个规范其实不冲突，我们按照最详细的那个来就好了，FHS。

## 生成shared library和实例

demo.c:

```
#include "stdio.h"
#include "libcasa.h"

int main(int argc, char *argv[]) {
    return print_a_message("hello world");
}
```

libcasa.c:

```
#include "libcasa.h"
int print_a_message(const char *data) {
    int i = 0;

    printf("here i am\n\n");

    for (i = 0; data[i] != '\0'; i++) {
        printf("%c", data[i]);
    }

    printf("\n");

    return 0;
}
```

libcasa.h:

```
#include "stdio.h"
int print_a_message(const char *data);
```

makefile:

```
shared:
    clang -Wall -fPIC -c libcasa.c -o libcasa.o
    clang -shared -Wl,-install_name,libcasa.so.1 -o libcasa.so.1.0
    ln -sf libcasa.so.1.0 libcasa.so.1
    ln -sf libcasa.so.1.0 libcasa.so
    clang -Wall -fPIC -g demo.c -o demo.run -lcasa -L./

static:
    clang -c libcasa.c -o libcasa.o
    ar rcs libcasa.a libcasa.o
    clang -o demo.run demo.c libcasa.a

clean:
    rm -rf *.o *.dSYM *.run *.a *.so *.so.1 *.so.1.0
```

写好这些文件之后，在命令行输入 `make shared` 就会生成基于动态链接库的可执行文件 `demo.run`。输入 `make static` 就会生成基于静态库的可执行文件 `demo.run`。

你会发现，决定一个库是静态还是动态，其实就仅仅取决于它的编译指令的不同。下面我着重解释一下动态库的编译指令。

```
clang -Wall -fPIC -c libcasa.c libcasa.o
```

- `-fPIC`：这是一个编译器指令，让编译器生成定位无关的代码，具体的定位操作由可执行文件加载时再重定位。要编译动态链接库，必须要有这个参数。`-fpic` 也可以用，但这两者不等价，`-fpic` 生成的文件会比 `-fPIC` 小一些，但牺牲了平台兼容性。所以用 `-fPIC` 生成的文件大就大一点了，比较保险。

```
clang -shared -Wl,-install_name,libcasa.so.1 -o libcasa.so.1.0  
libcasa.o
```

- `-shared`：告诉编译器你要生成的是一个 `shared library`，也就是动态加载库。
- `-Wl,-install_name,libcasa.so.1`：这里就是给你的动态库设置 `soname` 的地方。`-Wl` 表示后面跟的字符串是传递给连接器的参数，设置了 `install_name` 为 `libcasa.so.1`，这将会出现在编译之后的库文件里面。因为我当前使用的是 Mac，如果你用的是 Linux 的话，就应该是 `-Wl,-soname,libcasa.so.1`。对的，`install_name` 就是 `soname`，不同的系统不同而已。编译完之后，你可以使用 `vim` 直接打开 `libcasa.so.1.0`，你会找到 `libcasa.so.1` 这个字样。如果你不传递这个参数也能编译通过，它就会把文件名作为 `soname`。但是原则上不能省略这个 `soname` 的配置，它会引起库版本管理错乱。
- `-o libcasa.so.1.0`：这就是动态库的 `real name`，表示生成出来的库的文件名为 `libcasa.so.1.0`，但它的 `soname` 是 `libcasa.so.1` 哦。
- `ln -sf libcasa.so.1.0 libcasa.so.1`：这里做了一个符号链接，把 `soname` 跟 `real name` 做了关联。这样在程序运行要加载动态库的时候，它会去加载 `libcasa.so.1`，由于有了符号链接，实际上内核会加载 `libcasa.so.1.0`，以后如果有更新的版本，但 `soname` 还是没变的话，内核就能自动加载最新版本了。
- `-sf libcasa.so.1.0 libcasa.so`：这里做了一个符号链接，把 `link name` 跟 `real name` 做了关联。这样在后面的 `-lcasa` 的时候，就会按照命名规范组装成 `libcasa.so`，然后找到 `libcasa.so` 做链接。

```
clang -Wall -fPIC -g demo.c -o demo.run -lcasa -L./
```

- `-fPIC`：这里的 `-fPIC` 其实不写也可以，但是 GCC4.5 的手册要求你最好还是写上，具体原因倒是没细说。
- `-lcasa`：编译器会通过这条指令组装动态库的 `link name`，在前面加上 `lib`，在后面加上 `.so`，最终形成 `libcasa.so`，编译器就会根据这个名字找到这个库，然后把这个动态库的 `soname` 读取出来，加到可执行文件的依赖列表里面去。
- `-L./`：告诉编译器搜索动态库的地址在当前目录

这时我们终于可以强调一下soname的规范了：

- 如果你更新的动态库内容变化并没有添加或删除API，只是修改了API的实现，那么soname可以不用变，只要改变你的动态库的real name就好。然后把对应soname做符号链接到你的新版本动态库中，下次启动这个可执行文件时，内核就会加载到最新的动态库了。
- 如果你更新的动态库里面有添加新的API，可以再在soname后面添加一个小版本号，soname就可以变成 `libcasa.so.1.1`。即使原来的可执行文件还是依赖于 `libcasa.so.1`，那也不影响使用。
- 如果你更新的动态库已经不兼容就版本了，那么soname后面的数字就要改改了，比如改成 `libcasa.so.2`。

所以你一定要给你的动态库在编译的时候设置soname，不然soname就跟着文件名走了，到后面涉及版本管理的时候你就坑了。

MAC下直接跑生成成功的 `demo.run` 没问题，Linux下要做一些额外的操作，具体看下面的小贴士。

## 小贴士

- Linux下 `make shared` 之后要想正常运行 `demo.run`，还需要一些额外的操作

你需要把你的 `libcasa.so.1` (real name)拷贝到 `/usr/local/bin` 下面，然后运行一次 `ldconfig`。`ldconfig`帮你建立了一些必要的符号链接，刷新了你的 `ld.so.cache`，然后你再跑 `demo.run` 就没问题了。

另外，我习惯在 `/usr/local/lib` 里为单独的一套库创建新的文件夹来存放，比如我会新建一个 `libcasa` 这个文件夹，把跟 `libcasa` 相关的库都放到这里面去。这么说的话，在 `install` 的时候就需要在 `/etc/ld.so.conf` 里面添加一行 `/usr/local/lib/libcasa`，然后再跑 `ldconfig`了。这么做的好处就是 `/usr/local/lib` 看上去很干净。

因为去不同的路径下面找各自的动态库很影响效率，所以Linux专门有这个cache来记录动态库都在哪儿，这样加载的时候就快了。发展到后面就变成，你不建立这个cache，内核干脆就不找了，直接跟你说没这个动态库。

- 在Linux下使用 `ldd` 命令能够看出一个可执行文件它依赖于哪些动态库，在Mac下使用 `otool -L` 来查看：

首先是忠告：你不要拿 `ldd` 去看你不信任的可执行文件，因为它会执行这个可执行文件。

这里是采用动态链接库方案编译的结果：



```
→ $ make shared
clang -Wall -fPIC -c libcasa.c -o libcasa.o
clang -shared -Wl,-install_name,libcasa.so.1 -o libcasa.so.1.0
ln -sf libcasa.so.1.0 libcasa.so.1
ln -sf libcasa.so.1.0 libcasa.so
clang -Wall -fPIC -g demo.c -o demo.run -lcasa -L./

→ $ otool -L demo.run
demo.run:
libcasa.so.1 (compatibility version 0.0.0, current version 0.0.0)
/usr/lib/libSystem.B.dylib (compatibility version 1.0.0, current version 1.0.0)
```

可以看到 demo.run 这个可执行文件依赖于 libcasa.so.1 这个soname的库。下面是采用静态库方案编译的结果：

```
→ $ make static
clang -c libcasa.c -o libcasa.o
ar rcs libcasa.a libcasa.o
clang -o demo.run demo.c libcasa.a

→ $ otool -L demo.run
demo.run:
/usr/lib/libSystem.B.dylib (compatibility version 1.0.0, current version 1.0.0)
```

可以看到此时 demo.run 这个可执行文件并不依赖于 libcasa.so 这个动态库，因为它已经被静态编译进这个可执行文件了。

- 使用 nm 能够看到一个库里面都有些啥。

```
→ $ nm libcasa.so.1.0
00000000000000ed0 T _print_a_message
                 U _printf
                 U dyld_stub_binder
```

- 使用 /etc/ld.so.preload 来覆盖一些函数

你可以把针对某个库中某些函数的实现编译成.o文件，填进去。不过我试了几次没有成功，这种做法一般是在调试bug的时候会用到，发行出去的动态库要杜绝这种做法。

## 动态加载库

动态加载库(Dynamic Load Library)是在程序运行时，由程序自己调用系统API去加载的，而不是内核加载程序时顺便加载的。本质上跟静态库和动态链接库没什么区别，只

是使用方式有所不同而已。由于可以由程序自主调用，这种方案使得插件机制和动态模块机制的实现成为了可能。

## 涉及的API

```
#include <dlfcn.h>

void* dlopen(const char* path, int mode);
void* dlsym(void* handle, const char* symbol);
int dlclose(void* handle);
char* dlerror(void);
```

Link with `-ldl`

大致流程就是先dlopen一个库，然后用dlsym从这个库里面根据函数名找到对应的函数，然后返回一个指向这个函数的指针，拿到这个指针直接调用就好了。库用完之后调用dlclose关闭就可以。记得要在编译可执行文件时添加一个 `-ldl` 参数，其他就没什么了。

man里面写得非常好，还有用例，我就不搬运了。

## 结束

这篇文章主要讲了如何去实现不同类型的库，IBM的知识库也有一篇类似的文章 (<http://www.ibm.com/developerworks/library/l-shlibs/>)着重介绍了 `shared library`，这篇文章的文末提供了一些其他文章，那里有很多更深的内容，推荐大家去看一下。

评论系统我用的是Disqus，不定期被墙。所以如果你看到文章下面没有加载出评论列表，翻个墙就有了。

本文遵守CC-BY。请保持转载后文章内容的完整，以及文章出处。本人保留所有版权相关权利。

我的博客拒绝挂任何广告，如果您觉得文章有价值，可以通过支付宝扫描下面的二维码

捐助我。



---

## Comments

27 条评论 Casa Taloyum

 登录 Recommend 分享

按从新到旧排序




Join the discussion...



曹帅 · 2个月前


大神 请教一个iOS 库的问题，Xcode里面的embedded binaries 的作用是什么啊？我在stackoverflow 看了看 感觉讲的不是很详细 我不用 embedded binaries 好像对程序也没有什么影响啊？ embedded binaries ？ 嵌入到App Bundle 里面？ stackoverflow 地址 <http://stackoverflow.com/quest...> Bundle 里 有什么好处啊？

  · 回复 · 分享CasaTaloyum 管理员  曹帅 · 2个月前

这说明你Google的深度还不够~继续去Google~潜下心不要浮躁。根据你上一个问题，我觉得你还没有理解什么叫库，如果你直接这样寻找问题，是寻找不出答案的。

  · 回复 · 分享曹帅  CasaTaloyum · 2个月前


谢谢田大神啊，其实我大致学习了一下库是什么概念？自己感觉库的好处是别人不容易改你的代码，增强模块的隔离性，可以按需要加载 其实就是感觉会比较方便一点 是不是这样的好处？ 田大神

  · 回复 · 分享CasaTaloyum 管理员  曹帅 · 2个月前

还有就是可以动态部署~

  · 回复 · 分享曹帅  CasaTaloyum · 2个月前

田大神 你别逗我 之前看了你的那篇iOS应用架构谈 本地持久化方案及动态部署博客 明明说的iOS在正式的distribution下 因为代码签名机制是行不通的

  · 回复 · 分享CasaTaloyum 管理员  曹帅 · 2个月前

这篇文章又不是讲iOS的咯😂

  · 回复 · 分享曹帅  CasaTaloyum · 2个月前

田大神 请教你一个问题我这边 做一个动态库 这个动态库里面 要用到CMABPersonSimpleModel对象 但是 这个类又不止我这个库一个用 其他地方也有在用 如果我把这个类导入到库里 这样其他用到这个地方要不就是重复 要不就是把原来的删除 导入这个库 但是 这个CMABPersonSimpleModel 其实不属于 我这个库的业务范围之内 这个时候该如何解决这个问题啊

---

© 2016 Casa Taloyum · Powered by pelican-bootstrap3 (<https://github.com/DandyDev/pelican-bootstrap3>), Pelican (<http://docs.getpelican.com/>), Bootstrap (<http://getbootstrap.com>)

[↑ Back to top](#)