理解 GNU Libtool

 **garfileo** 发布于 2015-11-16

这篇文章与『理解 GLib 的单元测试框架』一文有些渊源，因为后者在几个示例中使用了 libtool 产生库文件与应用程序文件。

田园时代

我要写一个叫做 foo 的库，它提供一个什么也不做的函数。这个库的头文件为 foo.h：

```
#ifndef FOO_H
#define FOO_H

void foo(void);

#endif
```

foo.c 是这个库的实现：

```
#include "foo.h"

void foo(void)
{
}
```

用 gcc 编译生成共享库文件 libfoo.so：

```
$ gcc -shared -fPIC foo.c -o libfoo.so
```

如果用 clang，可以这样：

```
$ clang -shared -fPIC foo.c -o libfoo.so
```

如果是在 Windows 环境中（例如 mingw 或 cygwin 之类的环境），可以这样：

```
$ gcc -shared -fPIC foo.c -o libfoo.dll
```

于是，问题就出现了……如果我想让 foo 库能够跨平台运行，那么我就不得不为每一个特定的平台提供相应的编译命令或脚本。这意味着，你必须知道各个平台在共享库支持方面的差异及处理方式。这通常是很烦琐很无趣的事，何况我还没说构建静态库的事。

这时候，一个 10000 多行的 Bash Shell 脚本 libtool 站了出来，这些破事，我来做！

消除环境差异的方法

要有效的消除各个环境差异性，往往有三种办法。

第一种办法是**革命**……不要害怕，不是革程序猿的命，而是革环境的命。譬如 Windows（我更愿意是 Linux）扫清寰宇，一统天下，那么环境的差异性也就不存在了。但是，人类的历史已经证明了这条路是走不通的。因为，一旦某个环境绝对的统治了一切，那么它下一步要面对的问题就是自身的分裂……整部中国历史记录的都是这种事！

第二种办法是**改良**……有一批人仁志士成立了某个团体，颁布了一些标准，并号召大家都遵守这个标准，别再自行其是。C 语言标准，C++ 标准，scheme 标准……都挺成功的。现在似乎还没有共享库或动态链接库标准。

第三种办法是**和谐**——不要害怕，这里没有 GFW——就是承认现实就是这么个狗血的现实，然后追求**和而不同**。

libtool 选择了第三种办法。

libtool 的『和』

gcc 编译生成共享库的命令可以粗略的拆分为两步——编译与连接：

```
$ gcc -fPIC foo.c -o libfoo.o  #编译
$ gcc -shared libfoo.o -o libfoo.so  #连接
```

与之相应，libtool 说，如果你要用 gcc 编译生成一个共享库，不管它是在 Linux 里，还是在 Solaris，还是在 Mac OS X 里，或者是在 Windows 里（Cygwin 或 MinGW），可以使用同样的命令：

```
$ libtool --tag=CC --mode=compile gcc -c foo.c -o libfoo.lo # 编译
$ libtool --tag=CC --mode=link gcc libfoo.lo -rpath /usr/local/lib -o libfoo.la  # 连接
```

似乎 libtool 把问题弄得更复杂了！不过，仔细观察一下，可以发现一些规律。比如这两个命令的前面一半都是：

```
$ libtool --tag=CC --mode=
```

`--tag` 选项用于告诉 libtool 要编译的库是用什么语言写的，`CC` 表示 C 语言。libtool 目前支持以下语言：

```
语言                Tag 名称
---------------------------
C                   CC
C++                 CXX
Java                GCJ
```

```
Fortran 77          F77
Fortran             FC
Go                  GO
Windows Resource    RC
--------------------------
```

`--mode` 选项用于设定 libtool 的工作模式。上面的例子中，`--mode=compile` 就是告诉 libtool 要做的工作是编译，而 `--mode=link` 就是连接。

libtool 支持 7 种模式，除了上述两种模式之外，还有执行、安装、完成、卸载、清理等模式。每个模式都对应于库的某个开发阶段。这 7 种模式抽象了大部分平台上的库的开发过程。这是 libtool 和而不同的第一步：**库开发过程的抽象。**

下面来看编译过程，当 libtool 的 `--mode` 选项设为 `compile` 时，那么随后便是具体的编译命令，本例中是 `gcc -c foo.c -o libfoo.lo`。这条 gcc 编译命令，会被 `libtool --tag=CC --mode=compile` 变换为：

```
$ gcc -c foo.c  -fPIC -DPIC -o .libs/libfoo.o
$ gcc -c foo.c -o libfoo.o >/dev/null 2>&1
```

注意，注意，注意！事实上，libtool 命令中的 `gcc -c foo.c -o libfoo.lo`，并非真正的 gcc 的编译命令（gcc 输出的目标文件默认的扩展名是 `.o` 而非 `.lo`），它只是 libtool 对编译器工作方式的一种抽象。在 libtool 看来，它所支持的编译器，都应该这样工作：

```
$ 编译器 -c 源文件 -o 目标文件
```

如果 libtool 所支持的编译器并不支持 `-c` 与 `-o` 选项，那么 libtool 也会想办法让它们像这样工作！这是 libtool 和而不同的第二步：**库编译过程的抽象。**

下面观察一下执行 libtool 命令前后文件目录的变化。假设 foo.h 与 foo.c 位于 foo 目录，并且 foo 目录里只有这两个文件：

```
$ cd foo
$ tree -a
.
├── foo.c
└── foo.h

0 directories, 2 files
```

现在，执行 libtool 编译命令：

```
$ libtool --tag=CC --mode=compile gcc -c foo.c -o libfoo.lo
```

然后再查看一下 foo 目录：

```
$ tree -a
.
├── foo.c
├── foo.h
├── libfoo.lo
├── libfoo.o
└── .libs
    └── libfoo.o

1 directory, 5 files
```

执行 libtool 命令后，多出来一个隐藏目录 `.libs`，以及三份文件 `libfoo.o`, `.libs/libfoo.o`, `libfoo.lo`。有点诡异的就是有两份 `libfoo.o` 文件，虽然它们位于不同的目录，但是它们的内容相同吗？`libfoo.lo` 文件说，它们不相同。因为 `libfoo.lo` 是一份人类可读的文本文件，用文本编辑器打开它，可以看到以下内容：

```
# libfoo.lo - a libtool object file
# Generated by libtool (GNU libtool) 2.4.6
#
# Please DO NOT delete this file!
# It is necessary for linking the library.

# Name of the PIC object.
pic_object='.libs/libfoo.o'

# Name of the non-PIC object
non_pic_object='libfoo.o'
```

位于 `foo/.libs` 目录中的 libfoo.o，是 PIC 目标文件，而位于 `foo` 目录中的 libfoo.o 则是非 PIC 目标文件。在 gcc 看来，PIC 目标文件就是共享库的目标文件，而非 PIC 目标文件就是静态库的目标文件。也就是说，libtool 的目标不仅仅要生成共享库文件，也要生成静态库文件。这是 libtool 和而不同的第三步：**目标文件的抽象**。

接下来，再执行以下 libtool 的连接命令：

```
$ libtool --tag=CC --mode=link gcc libfoo.lo -rpath /usr/local/lib -o libfoo.la
```

从『形状』上来看，这条命令与

```
$ gcc -shared libfoo.o -o libfoo.so
```

相似，libfoo.lo 对应 libfoo.o，而 `libfoo.la` 对应 `libfoo.so`。事实上就是这样对应的。 libfoo.lo 是对 libfoo.o 的抽象，而 libfoo.la 是对 libfoo.so 的抽象。libfoo.lo 抽象的是共享库与静态库的目标文件，而 libfoo.la 抽象的就是共享库与静态库。libfoo.la 也是人类可读的。用文本编辑器打开 libfoo.la 文件，可以看到：

```
# libfoo.la - a libtool library file
# Generated by libtool (GNU libtool) 2.4.6
```

```
#
# Please DO NOT delete this file!
# It is necessary for linking the library.

# The name that we can dlopen(3).
dlname='libfoo.so.0'

# Names of this library.
library_names='libfoo.so.0.0.0 libfoo.so.0 libfoo.so'

# The name of the static archive.
old_library='libfoo.a'

# Linker flags that cannot go in dependency_libs.
inherited_linker_flags=''

# Libraries that this one depends upon.
dependency_libs=''

# Names of additional weak libraries provided by this library
weak_library_names=''

# Version information for libfoo.
current=0
```

文件内容太多，要关注的内容是：

```
# The name that we can dlopen(3).
dlname='libfoo.so.0'

# Names of this library.
library_names='libfoo.so.0.0.0 libfoo.so.0 libfoo.so'

# The name of the static archive.
old_library='libfoo.a'

Directory that this library needs to be installed in:
libdir='/usr/local/lib'
```

显然，libfoo.la 包含了 libtool 生成的（其实是 gcc 生成的）共享库与静态库信息，并且它还包含了一个 `libdir` 变量。这个变量的值，显然是 libtool 的连接命令中的 `-rpath /usr/local/lib` 设定的。`libdir` 表示 libfoo.la, libfoo.so.*, libfoo.a 等文件最终都应该放到 `/usr/local/lib` 目录。

下面看一下 foo 目录中的文件变化：

```
$ tree -a
.
├── foo.c
├── foo.h
├── libfoo.la
├── libfoo.lo
```

```
├── libfoo.o
└── .libs
    ├── libfoo.a
    ├── libfoo.la -> ../libfoo.la
    ├── libfoo.lai
    ├── libfoo.o
    ├── libfoo.so -> libfoo.so.0.0.0
    ├── libfoo.so.0 -> libfoo.so.0.0.0
    └── libfoo.so.0.0.0

1 directory, 12 files
```

这就是 libtool 三步抽象的所有成果，libfoo.a 与含有 `.so` 的那些文件，就是最终生成的静态库与共享库文件，而 libfoo.la 是它们的抽象。

注意，还有一个 libfoo.lai 文件，它是一个临时文件，当我们使用 libtool 将 foo 库安装到 `/usr/local/lib` 目录时，它就变成了 libfoo.la。其实，libfoo.la 与 libfoo.lai 的区别是，前者的内容中有一个 `installed` 变量，它的值是 `no`，而在后者的内容中，这个变量的值是 `yes`。可以将此刻的 libfoo.la 视为安装前的库的抽象，而将 libfoo.lai 视为安装后的库的抽象。

对未安装的库进行抽象，有什么用？便于在库的开发过程中对其进行单元测试。

库的测试

为了显得不那么业余，我需要对 foo 目录中的文件进行一些变动，变动后的目录结构如下：

```
$ tree -a
.
├── lib
│   ├── foo.c
│   └── foo.h
└── test

2 directories, 2 files
```

就是将 foo.h 与 foo.c 放到 lib 目录中，另外新建了一个 test 目录。我要在 test 目录中建立测试程序，即 test.c，其内容如下：

```
#include <foo.h>

int main(void)
{
        foo();
        return 0;
}
```

然后使用 libtool 重新编译生成库文件：

```
$ cd lib
$ libtool --tag=CC --mode=compile gcc -c foo.c -o libfoo.lo
$ libtool --tag=CC --mode=link gcc libfoo.lo -rpath /usr/local/lib -o libfoo.la
```

现在，foo 目录结构变成：

```
$ cd .. # 返回 foo 目录，因为刚才是在 foo/lib 目录里
$ tree -a
.
├── lib
│   ├── foo.c
│   ├── foo.h
│   ├── libfoo.la
│   ├── libfoo.lo
│   ├── libfoo.o
│   └── .libs
│       ├── libfoo.a
│       ├── libfoo.la -> ../libfoo.la
│       ├── libfoo.lai
│       ├── libfoo.o
│       ├── libfoo.so -> libfoo.so.0.0.0
│       ├── libfoo.so.0 -> libfoo.so.0.0.0
│       └── libfoo.so.0.0.0
└── test
    └── test.c
```

下面，编译测试程序，即编译 test.c：

```
$ cd test
$ libtool --tag=CC --mode=compile gcc -I../lib -c test.c
$ libtool --tag=CC --mode=link gcc ../lib/libfoo.la test.lo -o test
```

执行 test 程序：

```
$ ./test
```

结果什么也不显示，这是正确的。因为 `foo()` 函数本来就是什么也不做的函数。

再看一下 foo 的目录结构的变化：

```
$ cd ..  # 因为刚才在 foo/test 目录中
$ tree -a
.
├── lib
```

```
│   ├── foo.c
│   ├── foo.h
│   ├── libfoo.la
│   ├── libfoo.lo
│   ├── libfoo.o
│   └── .libs
│       ├── libfoo.a
│       ├── libfoo.la -> ../libfoo.la
│       ├── libfoo.lai
│       ├── libfoo.o
│       ├── libfoo.so -> libfoo.so.0.0.0
│       ├── libfoo.so.0 -> libfoo.so.0.0.0
│       └── libfoo.so.0.0.0
└── test
    ├── .libs
    │   ├── test
    │   └── test.o
    ├── test
    ├── test.c
    ├── test.lo
    └── test.o
```

结果，在 test 目录中生成了可执行文件 test，但是 test 目录也有个 .libs 目录，而这个 .libs 目录里也包含了一份可执行文件 test……这是 libtool 的第四部抽象：**可执行文件的抽象**。结果，在 test 目录中生成了可执行文件 test，但是 test 目录也有个 .libs 目录，而这个 .libs 目录里也包含了一份可执行文件 test……这是 libtool 的第四步抽象：**可执行文件的抽象**。不知你有没有注意到，生成 test 的过程与生成 libfoo.la 的过程几乎是一样的！其实也没什么好奇怪的，因为共享库或静态库本身就是可执行文件。

`foo/test/test` 文件，其实是一份 Bash 脚本，而 `foo/test/.libs/test` 才是真正的 test 程序。为了便于描述，我将前者称为 test 脚本，将后者称为 test 程序。test 脚本就是对 test 程序的抽象！

test 脚本所做的工作就是为 test 程序的运行提供正确的环境。因为运行 test 程序，需要加载 foo 库。按照 Linux 的共享库加载逻辑，系统会自动去 `/usr/lib` 目录为 test 程序搜索共享库 libfoo.so，或者去环境变量 `LD_LIBRARY_PATH` 所定义的路径去搜索 libfoo.so。但是，但是，但是，此刻我们的 foo 库还没有被安装，我们也没有设置 `LD_LIBRARY_PATH` 变量，test 程序是运行不起来的，所以，需要一个 test 脚本来抽象它！

用文本编辑器打开 test 脚本，在 200 多行 Bash 代码中可以看到以下内容：

```
# Add our own library path to LD_LIBRARY_PATH
LD_LIBRARY_PATH="/tmp/foo/lib/.libs:$LD_LIBRARY_PATH"

# Some systems cannot cope with colon-terminated LD_LIBRARY_PATH
# The second colon is a workaround for a bug in BeOS R4 sed
LD_LIBRARY_PATH=`$ECHO "$LD_LIBRARY_PATH" | /bin/sed 's/::*$//'`

export LD_LIBRARY_PATH
```

我看着这份死活也看不懂的 test 脚本，深深感到 libtool 为了让这个什么也不做的 test 程序能够正确的运行，呕心沥血，很拼的！

库的安装与卸载

foo 库，经过我的精心测试，没发现它有什么 bug，现在我要将它安装到系统中：

```
$ cd lib  # 因为刚才跑到 foo 目录下查看了目录结构
$ sudo libtool --mode=install install -c libfoo.la /usr/local/lib
```

我觉得我没必要再说废话，简明扼要的说，这是 libtool 的第五步抽象：**库文件安装**抽象。

事实上，很少有人去用 libtool 来安装库。大部分情况下，libtool 是与 GNU Autotools 配合使用的。更正确的说法是，libtool 属于 GNU Autotools。我不知道在这里我将话题引到 GNU Autotools 是不是太唐突，因为有关 GNU Autotools 的故事，要差不多半个月才能讲完……

简单的说，GNU Autotools 就是产生两份文件，一份文件是 configure，用于检测项目构建（预处理、编译、连接、安装）环境是否完备；另一份文件是 Makefile，用于项目的构建。如果我们的项目是开发一个库，那么一旦有了 GNU Autotools 生成的 Makefile，编译与安装这个库的命令通常是：

```
$ ./configure          # 检测构建环境
$ make                 # 编译、连接
$ sudo make install    # 安装
```

也就是说，Makefile 中包含了 libtool 的编译、连接以及安装等命令。这篇文章的目的是帮助你理解 libtool，并非希望你使用 libtool 这个小木船来取代航母级别的 GNU Autotools。

既然以后很可能是用 GNU Autotools 来构建项目，因此可以用 libtool 命令卸载刚才所安装的库文件：

```
$ sudo libtool --mode=uninstall rm /usr/local/lib/libfoo.la
```

这是 libtool 的第六步抽象：**库文件卸载**抽象。它对应于 GNU Autotools 产生的 Makefile 中的 `make uninstall`。

归根复命

foo 库，经过我的精心测试，没发现它有什么 bug 了，我想将源代码分享给我的朋友……虽然我几乎没有这种朋友。

既然是分享，那么就得将一些对别人无用的东西都删掉。现在我的 foo 目录中有这些文件：

```
$ cd ..  # 因为刚才在 foo/lib 目录中
$ tree -a
.
├── lib
```

```
│   ├── foo.c
│   ├── foo.h
│   ├── libfoo.la
│   ├── libfoo.lo
│   ├── libfoo.o
│   └── .libs
│       ├── libfoo.a
│       ├── libfoo.la -> ../libfoo.la
│       ├── libfoo.lai
│       ├── libfoo.o
│       ├── libfoo.so -> libfoo.so.0.0.0
│       ├── libfoo.so.0 -> libfoo.so.0.0.0
│       └── libfoo.so.0.0.0
└── test
    ├── .libs
    │   ├── test
    │   └── test.o
    ├── test
    ├── test.c
    ├── test.lo
    └── test.o
```

我要删除 `*.o`，`*.lo`，`*.la`, `*.lai`, `*.a`, `*.so.*` 等文件，只保留源代码文件。手动删除有点繁琐，为此，libtool 提供了第七步抽象：**归根复命**的抽象。归根复命，这么高大上的概念，是老子创造的。他说，『夫物芸芸，各归其根。归根曰静，是谓复命』。

哲学的事，先放一边，libtool 让 foo 库归根复命的命令是：

```
$ cd lib
$ libtool --mode=clean rm libfoo.lo libfoo.la
$ cd ../test
$ libtool --mode=clean rm test.lo test
```

然后再看一下 foo 的目录结构：

```
$ cd ..  # 因为刚才在 test 目录中
$ tree -a
.
├── lib
│   ├── foo.c
│   └── foo.h
└── test
    └── test.c

2 directories, 3 files
```

对于 GNU Autotools 产生的 Makefile 而言，libtool 的 `clean` 模式对应于 `make clean`。

总结

将上面所讲的 libtool 命令的用法都忘了吧，只需要大致上理解它对哪些事物进行了**抽象**即可。因为，GNU Autotools 提供的 autoconf 与 automake 已将所有的 libtool 命令隐藏在它们的黑暗魔法中了。

libtool c 共享库 gcc