IBM

developerWorks 中国 技术主题 Linux 文档库

Intel平台下Linux中ELF文件动态链接的加载、解析及实例分析（一）：加载

动态链接，一个经常被人提起的话题。但在这方面很少有文章来阐明这个重要的软件运行机制，只有一些关于动态链接库编程的文章。本系列文章就是要从源代码的层次来探讨这个问题。

王瑞川 linux爱好者，愿与志同道合者一起探讨，联系方式 jeppeterone@163.com

2003 年 10 月 01 日

在 IBM Bluemix 云平台上开发并部署您的下一个应用。

开始您的试用

当然从文章的题目就可以看出，intel平台下的linux ELF文件的动态链接。一则是因为这一方面的资料查找比较方便，二则也是这个讨论的意思比其它的动态链接要更为重要（毕竟现在是intel的天下）。当然，有了这么一个例子，其它的平台下的ELF文件的动态链接也就大同小异。你可以在阅读完了本文之后"举一隅，而反三隅"了。

由于这是一个系列的文章，我计划分三部分来写，第一部分主要分析加载，涉及dl_open这个函数的内容，但由于这个函数所包含的内容实在太多。这里主要是它的_dl_map_object与_dl_init这两个部分，因为这里是把动态链接文件通过在ELF文件中的得到信息映射到内存空间中，而_dl_init中是一个特殊的初始化。这是对面向对象的函数实现的。

第二部分我将分析函数解析与卸载，这里要讲的内容会比较多，但每一个内容都不会多。首先是在前一篇中没有说完的dl_open中的涉及的_dl_map_object_deps和_dl_relocate_object两个函数内容，因为这些都与函数解析的内容直接相关，所以安排在这里。而下面的函数解析过程_dl_runtime_resolve是在程序运行中的动态解析过程。这里从本质上来讲没有太多的代码，但它的精巧程度却是最多的（正是我这三篇文章的核心之处）。最后是一个dl_close的实现。这里是一个结尾的工作，顺带一下是_dl_signal_cerror，与_dl_catch_error的错误例外处理。

第三部将给出injectso实例分析与应用，会介绍一个应用了动态链接的实例，并可以在日后的程序调试过程中使用的injectso实例，它不仅可以让我们对前面所说的动态链接原理有一个更感性的认识，而且就这个实例而言，还可以在以后的代码开发过程中来作为一种动态打补丁的工具，甚至有可能，我会在以后的文章中会用这个工具来介绍新的技术。

一、历史问题

关于动态链接，可以说由来已久。如果追溯，最早的思想就在五十年代就有了，那时就想把一些公用的代码放在内存中的一个地方上，在别的地址用call便是了。到后来又发展到了 loading overlays（就是把在程序运行生命期不同的代码在不同的时间段被加入内存），这是在六十年代的事。但这只能算是"滥觞"时期。接近于我们现在所说的动态链接是在unix操作系统之后，因为从unix的设计结构而言，本身就是分成模块来实现一个复杂的功能的操作系统。但这些还不是现代意义上的动态链接，原因是现代意义上的动态链接要符合两个特点：

1、 动态的加载，就是当这个运行的模块在需要的时候才被映射入运行模块的虚拟内存空间中，如一个模块在运行中要用到mylib.so中的myget函数，而在没有调用mylib.so这个模块中的其它函数之前，是不会把这个模块加载到你的程序中（也就是内存映射），这些内容在内核中实现，用的是页面异常机制（我可能在另一篇文章中提到这个问题）。

2、 动态的解析，就是当要调用的函数被调用的时候，才会去把这个函数在虚拟内存空间的起始地址解析出来，再写到专门在调用模块中的储存地址内，如前面所说的你已经调用了myget，所以mylib.so模块肯定

已经被映射到了程序虚拟内存之中，而如果你再调用mylib.so中的myput函数，那它的函数地址就在调用的时候才会被解析出来。

（注：这里用的程序就是一般所说的进程process，而模块既可能是你的程序的二进制代码，也可能是被你的程序所依赖的别的共享链接文件-------同样ELF格式。）

在这两点中很有点像现在的操作系统中对内存的操作，也就是只有当要用到一个内存空间中的时候才会进行虚拟空间映射，而不是过早的把所有的空间映射好，而只有当要从这个内存空间读的时候才分配物理空间。这有点像第一条。而只有当对这个内存空间进行写的时候产生一个COW（copy on write）。这就有点像第二条。

这样的好处就是充分避免不必要的开销。因为任何一个程序在运行的时候，大部分情况下，不可能用到所有的调用函数。

这样的思想方法提出与实现都是在八十年代的sun公司的SunOS的系统上。

关于这一段历史，请你参见资料[1]。

ELF二进制格式文件与现代的动态链接思想大致是在同一时段形成的，它的来源是AT&T公司的最早的unix中的a.out二进行文件格式。Bell labs的工作人员为了使这种在unix的早期主要的文件格式适应当时新的软件与操作系统的要求（如aix,SunOS,HP-UX这样的unix变种，对更广泛的应用程序的扩展要求，对面向对象的支持等等），就发明了ELF文件格式。

我在这里并不详细讨论ELF文件的具体细节，这本来就可以写一篇很长的文章，你可以参看资料[2]来得到关于它的ABI（application binary interface的规范）。但在ELF文件所采用的那种分层的管理方式却不仅在动态链接中起着重要的作用，而且这一思想可以说是我们计算机中的最古老，也是最经典的思想。

对每个ELF文件，都有一个ELF header，在这里的每个header有两个数据成员，就是

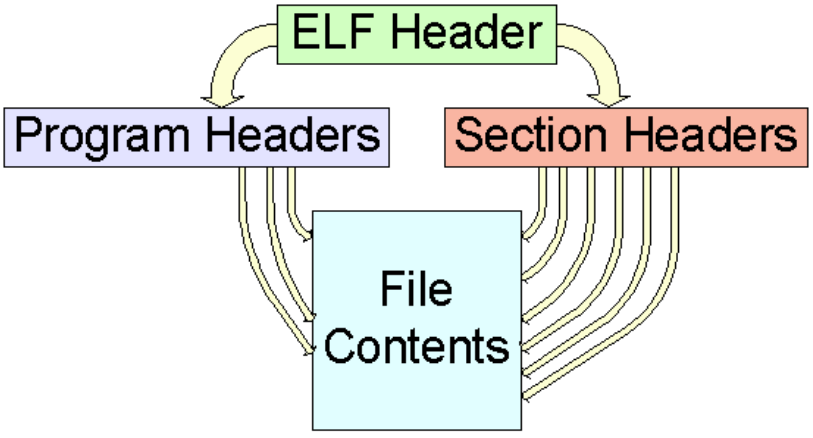
```
Elf32_Off  e_phoff;
Elf32_Off  e_shoff;
```

它们分别代表了program header 与section header 在ELF文件中的偏移量。Program header 是总纲，而section header 则是第一个小目。

```
Elf32_Addr  sh_addr;
Elf32_Off  sh_offset;
```

Sh_addr这个section 在内存中的映射地址（对动态链接库而言，这是一个相对量，它与整个ELF文件被加载的l_addr形成绝对地址）。Sh_offset是这个section header在文件中的偏移量。

用一图来表示就是这样的，它就是用elf header 来管理了整个ELF文件：



举个例子，如果要从一个**ELF**动态链接库文件中，根据已知的函数名称，找到相应的函数起始地址，那么过程是这样的。

先从前面的**ELF** 的**ehdr**中找到文件的偏移**e_phoff**处，在这其中找到为**PT_DYNAMIC** 的**d_tag**的**phdr**，从这个地址开始处找到**DT_DYNAMIC**的节，最后从其中找到这样一个**Elf32_Sym**结构，它的**st_name**所指的字符串与给定的名称相符，就用**st_value**便是了。

这种的管理模式，可以说很复杂，有时会看起来是繁琐。如找一个**function** 的起始地址就要从 **elf header >>program header >>symbol section >>function address** 这样的四个步骤。但这里的根本的原因是我们的计算机是线性寻址的，并且冯*诺依曼提出的计算机体系结构相关，所以在前面说这是一个古老的思想。但同样也是由于这样的一个**ELF**文件结构，很有利于**ELF**文件的扩充。我们可以设想，如果有一天，我们的**ELF**文件为了某种原因，对它进行加密。这时如果要在**ELF**文件中保存密钥，这时候可以在**ELF**文件中开辟一个专门的**section encrypt** ，这个**section** 的**type** 就是**ST_ENCRYPT**，那不就是可以了吗？这一点就可以看出**ELF**文件格式设计者当初的苦心了（现在这个真的有这么一个节了）。

二、代码举例

讲了这么多，还没有真正讲到在intel 32平台下linux动态链接库的加载与调用。在一般的情况下，我们所编写的程序是由编译器与ld.so这个动态链接库来完成的。而如果要显式的调用某一个动态链接库中的程序，则下面是一个例子。

```
#include <dlfcn.h>
#include <stdio.h>
main()
{
void *libc;
void (*printf_call)();
char* error_text;
if(libc=dlopen("/lib/libc.so.5",RTLD_LAZY))
  {
   printf_call=dlsym(libc,"printf");
   (*printf_call)("hello, world\n");
dlclose(libc);
return 0;
}
error_text= dlerror();
printf(error_test);
return -2;
}
```

在这里先用dlopen来打开一个动态链接库文件，而这个过程比我们这里看到的内容多的多，我会在下面用很大的篇幅来说明这一点，而它返回的参数是一个指针，确切的说是struct link_map*，而dlsym就是在这个struct link_map* 与函数名称一起决定这个函数在这个进程中的地址，这个过程用术语来说就是函数解析(function resolution)。而最后的dlclose就是释放刚才在dlopen中得到的资源，这个过程与我们在加载的share object file module，内核中的程序是大概相同的，只不过这里是在用户态，而那个是在内核态。从函数的复杂性而言这里还要复杂一些（最后有一点要说明，如果你想编译上面的文件-------文件名如果是test那就不能用一般的gcc -o test test.c ，而应该是gcc -c test test.c -ldl这样才能编译通过，因为不这样编译器会找不到dlopen 与dlsym dlclose这些特别函数的库文件libdl.so.2， -ldl 就是加载它的标志的）。

三、**_dl_open**加载过程分析

本文以及以后的两篇文章将都以上面的程序所展示的而讲解。也就是以dlopen >> dlsym >> dlclose 的方式来讲解这个过程，但有几点先要说明： 我在这里所展示的源代码来自glibc 2.3.2版本。但由于原来的代码，从代码的移植与健壮的考虑，而有许多的防止出错，与关于不同平台的代码，在这里大部分是出错处理代码，我把这些的代码都删除。并且只以intel 32平台下的代码为准。还有，在这里的还考虑到了多线程情况下的动态链接库加载，这里也不予以包括在内（因为现在的linux内核中没有对内核线程的支持）。所

以你所看到的代码，在尽量保证说明动态链接加载与函数解析的情况作了多数的删减，代码量大概只有原来的四分之一左右，同时最大程度保持了原来代码的风格，突出核心功能。尽管如此，还是有高达2000行以上的代码，请大家耐心的解读。我也会对其中可能的难解之处作出详细的说明。让大家真正体会到代码设计与动态解析的真谛。

第一个函数在dl-open.c中

```
   2672  void* internal_function
  2673  _dl_open (const char *file, int mode, const void *caller)
  2674  {
  2675     struct dl_open_args args;
  2676
  2677       __rtld_lock_lock_recursive (GL(dl_load_lock));
  2678
  2679     args.file = file;
  2680     args.mode = mode;
  2681     args.caller = caller;
  2682     args.map = NULL;
  2683
  2684     dl_open_worker(&args);
  2685        __rtld_lock_unlock_recursive (GL(dl_load_lock));
  2686
  2687  }
```

这里的internal_function是表明这个函数从寄存器中传递参数，而它的定义在configure.in中得到的。

define internal_function __attribute__ ((regparm (3), stdcall))

这其中的regparm就是gcc的编译选项是从寄存器传递3个参数，而stdcall表明这个函数是由调用函数来清栈，而一般的函数是由调用者来负责清栈，用的是cdecl。 __rtld_lock_lock_recursive (GL(dl_load_lock)); 与__rtld_lock_unlock_recursive (GL(dl_load_lock));在现在还没有完全定义，至少在linux中是没有的，但可以参考在linux/kmod.c 中的request_module中为了防止过度嵌套而加的一个锁。

而其它的内容就是一个封装了。

dl_open_worker是真正做动态链接库映射并构造一个struct link_map而这是一个绝对重要的数据结构它的定义由于太长，我会放在第二篇文章结束的附录中介绍，因为那时你可以回头再理解动态链接库加载与解析的过程，而在下面的具体函数中出现了作实用性的解释，下面我们分段来看：

```
_dl_open() >> dl_open_worker()
  2532  static void
  2533  dl_open_worker (void *a)
2534  {
……..
2547  args->map = new = _dl_map_object (NULL, file, 0, lt_loaded, 0, mode);
```

这里就是调用_dl_map_object 来把文件映射到内存中。原来的函数要从不同的路径搜索动态链接库文件，还要与SONAME（这是动态链接库文件在运行时的别名）比较，这些内容我在这里都删除了。

```
_dl_open() >> dl_open_worker() >> _dl_map_object()
  1693  struct link_map *
  1694  internal_function
  1695  _dl_map_object (struct link_map *loader, const char *name, int preloaded,
  1696      int type, int trace_mode, int mode)
  1697  {
  1698    int fd;
  1699    char *realname;
  1700    char *name_copy;
  1701    struct link_map *l;
  1702    struct filebuf fb;
  1703
  1704
  1705    /* Look for this name among those already loaded.  */
  1706    for (l = GL(dl_loaded); l; l = l->l_next)
  1707      {
  1708          if (!_dl_name_match_p (name, l))
……..
  1721          return l;
  1722      }
  1723
  1724     fd = open_path (name, namelen, preloaded, &env_path_list,
  1725        &realname, &fb);
  1726
  1727     l = _dl_new_object (name_copy, name, type, loader);
  1728
  1729     return _dl_map_object_from_fd (name, fd, &fb, realname, loader, type, mode);
  1730
  1731
1732  }/*end of _dl_map_object*/
```

这里先在已经被加载的一个动态链接库的链中搜索，在1706与1721行中就是作这一件事。想起来也很简单，因为可能在一个可执行文件依赖好几个动态链接库。而其中有几个动态链接库或许都依赖于同一个动态链接文件，可能早就加载了这样一个动态链接库，就是这样的情况了。

下面open_path是一个关键，这里要指出的是env_path_list得到的方式有几种，一是在系统环境变量，二就是DT_RUNPATH所指的节中的字符串（参见下面的 附录），还有更复杂的，是从其它要加载这个动态链接库文件的动态链接库中得到的环境变量-------这些问题我们都不说明了。

```
   _dl_open() >> dl_open_worker() >> _dl_map_object() >> open_path()
 1289  static int open_path (const char *name, size_t namelen, int preloaded,
 1290       struct r_search_path_struct *sps, char **realname,
 1291       struct filebuf *fbp)
 1292
 1293  {
 1294    struct r_search_path_elem **dirs = sps->dirs;
 1295    char *buf;
 1296    int fd = -1;
 1297    const char *current_what = NULL;
 1298    int any = 0;
 1299
 1300    buf = alloca (max_dirnamelen + max_capstrlen + namelen);
 1301
 1302    do
 1303      {
 1304        struct r_search_path_elem *this_dir = *dirs;
 1305        size_t buflen = 0;
.....
 1310       struct stat64 st;
 1311
 1312
 1313        edp = (char *) __mempcpy (buf, this_dir->dirname, this_dir->dirnamelen);
 1314        for (cnt = 0; fd == -1 && cnt < ncapstr; ++cnt)
 1315    {
 1316      /* Skip this directory if we know it does not exist.  */
 1317      if (this_dir->status[cnt] == nonexisting)
 1318        continue;
 1319
 1320      buflen = ((char *) __mempcpy (__mempcpy (edp, capstr[cnt].str,
 1321            capstr[cnt].len), name, namelen)- buf);
 1322
 1323
 1324      fd = open_verify (buf, fbp);
 1325
 1326
 1327      __xstat64 (_STAT_VER, buf, &st);
 1328
 1329
 1341    }
 1342
.....
    1358    }
```

在这上面的alloc是在栈上分配空间的函数，这样就不用担心在函数结束的时候出现内存泄漏的情况（好的程序员真的要对内存的分配熟谙于心）。1313行就是把r_search_path_elem的dirname copy过来，而在1320至1321行的内容就是为这个路径加上最后的'/'路径分隔号，而capstr就是根据不同的操作系统与体系得到的路径分隔号。这其实是一个很好的例子，因为__memcpy返回的参数是dest string所copy的最后的一个字节的地址，所以每copy之后就会得到新的地址，如果用strncpy来写的话，就要用这样的方法

```
strncpy(edp, capstr[cnt].str, capstr[cnt].len);
edp+=capstr[cnt].len;
strncpy(edp,name, namelen);
edp+=namelen;
buflen=edp-buf;
```

这就要用四句，而这里用了一句就可以了。

下面的open_verify是打开这个buf所指的文件名，fbp是从这个文件得到的文件开时1024字节的内容,并对文件的有效性进行检查，这里最主要的是ELF_IMAGIC核对。如果成功，就返回一个大于-1的文件描述符。整个open_path就这样完成了打开文件的方法。

_dl_new_object是一个分配struct link_map* 数据结构并填充一些最基本的参数。

```
 _dl_open() >> dl_open_worker() >> _dl_map_object() >> _dl_new_object()
 2027  struct link_map *
 2028  internal_function
 2029  _dl_new_object (char *realname, const char *libname, int type,
 2030       struct link_map *loader)
 2031
 2032  {
 2033    struct link_map *l;
 2034    int idx;
 2035    size_t libname_len = strlen (libname) + 1;
 2036    struct link_map *new;
```

```
  2037    struct libname_list *newname;
  2038
  2039     new = (struct link_map *) calloc (sizeof (*new) + sizeof (*newname)
  2040               + libname_len, 1);
  2041
……..
  2046
  2047    new->l_name = realname;
  2048    new->l_type = type;
  2049    new->l_loader = loader;
  2050
  2051    new->l_scope = new->l_scope_mem;
  2052    new->l_scope_max = sizeof (new->l_scope_mem) / sizeof (new->l_scope_mem[0]);
  2053
  2054   if (GL(dl_loaded) != NULL)
  2055      {
  2056        l = GL(dl_loaded);
  2057        while (l->l_next != NULL)
  2058     l = l->l_next;
  2059        new->l_prev = l;
  2060        /* new->l_next = NULL;  Would be necessary but we use calloc.  */
  2061        l->l_next = new;
  2062
  2063        /* Add the global scope.  */
  2064        new->l_scope[idx++] = &GL(dl_loaded)->l_searchlist;
  2065      }
  2066    else
  2067      GL(dl_loaded) = new;
  2068    ++GL(dl_nloaded);
 ……..
  2080
  2081      return new;
  2082
  2083  }
```

在2039行的内存分配是一个把libname 与name的数据结构也一同分配，是一种零用整取的策略。从2043-2053行都是为struct link_map 的成员数据赋值。从2054-2067行则是把新的struct link_map* 加入到一个单链中，这是在以后是很有用的，因为这样在一个执行文件中如果要整体管理它相关的动态链接库，就可以以单链遍历。

如果要加载的动态链接库还没有被映射到进程的虚拟内存空间的话，那只是准备工作，真正的要点在_dl_map_object_from_fd()这个函数开始的。因为这之后，每一步都有关动态链接库在进程中发挥它的作用而必须的条件。

这上段比较长，所以分段来看，

```
_dl_open() >> dl_open_worker() >> _dl_map_object() >> _dl_map_from_fd()
  1391  struct link_map *
  1392  _dl_map_object_from_fd (const char *name, int fd, struct filebuf *fbp,
  1393        char *realname, struct link_map *loader, int l_type,
  1394        int mode)
  1395
  1396  {
  1397
  1398    struct link_map *l = NULL;
  1399    const ElfW(Ehdr) *header;
  1400    const ElfW(Phdr) *phdr;
  1401    const ElfW(Phdr) *ph;
  1402    size_t maplength;
  1403    int type;
  1404    struct stat64 st;
  1405
  1406    __fxstat64 (_STAT_VER, fd, &st);
……..
  1413    for (l = GL(dl_loaded); l; l = l->l_next)
  1414      if (l->l_ino == st.st_ino && l->l_dev == st.st_dev)
  1415        {
……..
  1418    __close (fd);
……..
  1422    free (realname);
  1423    add_name_to_object (l, name);
  1424
  1425    return l;
1426  }
```

这里先开始就要从再找一遍，如果找到了已经有的struct link_map* 要加载的libname（的而比较的依据是它的与st_ino，这是物理文件在内存中编号，且文件的设备号st_dev相同，这是从比较底层来比较文件，具体的原因，你可以参看我将要发表的《从linux的内存管理看文件共享的实现》）。之所以采取这样再查一遍，因为如果进程从要开始打开动态链接库文件，走到这里可能要经过很长的时间（据我作的实验来看，对第一次打开的文件大概也就在200毫秒左右---------主要的时间是硬盘的寻道与读盘，但这对于计算机的进程而言已经是很长的时间了。）所以，有可能别的线程已经读入了这个动态链接库，这样就没有必要再做下去了。这与内核在文件的打开文件所用的思想是一致的。

```
_dl_open() >> dl_open_worker() >> _dl_map_object() >> _dl_map_from_fd()
  1427
  1428    /* This is the ELF header.  We read it in `open_verify'.  */
  1429    header = (void *) fbp->buf;
  1430
  1431    l->l_entry = header->e_entry;
  1432    type = header->e_type;
  1433    l->l_phnum = header->e_phnum;
  1434
  1435    maplength = header->e_phnum * sizeof (ElfW(Phdr));
  1436
```

这一段所作的为下面的ELF文件的分节映射入内存做一点准备（要读写phdr的数组）。

```
_dl_open() >> dl_open_worker() >> _dl_map_object() >> _dl_map_from_fd()
  1438       /* Scan the program header table, collecting its load commands.  */
  1439      struct loadcmd
  1440        {
  1441    ElfW(Addr) mapstart, mapend, dataend, allocend;
  1442    off_t mapoff;
  1443    int prot;
  1444        } loadcmds[l->l_phnum], *c;
1445  size_t nloadcmds = 0;
```

这里把数据结构定义在函数内部，能保证这是一个局部变量定义，与面向对象中的private的效果是一样的。

```
_dl_open() >> dl_open_worker() >> _dl_map_object() >> _dl_map_from_fd()
  1448      for (ph = phdr; ph < &phdr[l->l_phnum]; ++ph)
  1449        switch (ph->p_type)
  1450    {
……..
  1454    case PT_DYNAMIC:
  1455      l->l_ld = (void *) ph->p_vaddr;
  1456      l->l_ldnum = ph->p_memsz / sizeof (ElfW(Dyn));
  1457      break;
  1458
  1459    case PT_PHDR:
  1460      l->l_phdr = (void *) ph->p_vaddr;
  1461      break;
  1462
  1463    case PT_LOAD:
 ……..
  1467      c = &loadcmds[nloadcmds++];
  1468      c->mapstart = ph->p_vaddr & ~(ph->p_align - 1);
  1469      c->mapend = ((ph->p_vaddr + ph->p_filesz + GL(dl_pagesize) - 1)
  1470             & ~(GL(dl_pagesize) - 1));
  1471      c->dataend = ph->p_vaddr + ph->p_filesz;
  1472      c->allocend = ph->p_vaddr + ph->p_memsz;
  1473      c->mapoff = ph->p_offset & ~(ph->p_align - 1);
……..
  1480      c->prot = 0;
  1481      if (ph->p_flags & PF_R)
  1482        c->prot |= PROT_READ;
  1483      if (ph->p_flags & PF_W)
  1484        c->prot |= PROT_WRITE;
  1485      if (ph->p_flags & PF_X)
  1486        c->prot |= PROT_EXEC;
    1488      break;
  ……..
1493  }
```

在ELF文件的规范中，根据不同的program header 不同，要实现不同的功能，采用不同的处理策略，具体的内容请参看 附录2中的说明。这里没有出现一般的default 但实际运行与下面的语句是等价的：

```
default:
   continue;
```

真是达到程序简洁的特点。

但有一个特别要指出的是PT_LOAD的那些，把所有的可以加载的节都在加载的数据结构中loadcmds中构建完成，是一个好的想法。特别是指针的妙用，值得学习(1467 c = &loadcmds[nloadcmds++];)。

```
_dl_open() >> dl_open_worker() >> _dl_map_object() >> _dl_map_from_fd()
  1498      c = loadcmds;
  ……..
  1501      maplength = loadcmds[nloadcmds - 1].allocend - c->mapstart;
  1502
  1503      if (__builtin_expect (type, ET_DYN) == ET_DYN)
  1504        {
……..
  1521    l->l_map_start = (ElfW(Addr)) __mmap ((void *)0, maplength,
  1522                  c->prot, MAP_COPY | MAP_FILE,
  1523                  fd, c->mapoff);
  1524
  1525          l->l_map_end = l->l_map_start + maplength;
  1526    l->l_addr = l->l_map_start - c->mapstart;
……..
  1535    __mprotect ((caddr_t) (l->l_addr + c->mapend),
  1536          loadcmds[nloadcmds - 1].allocend - c->mapend,
  1537          PROT_NONE);
```

```
  1538
  1539    goto postmap;
1540  }
```

在1521-1526行之间就是把整个文件都进行了映射，妙处在1498行与1501行，是把头与尾的两个 PT_LOAD program header 的内容都计算在内了。而1503行就是我们这里的情景，因为这是动态链接库的加载。而1535行的修改虚拟内存的属性，就是把映射在最高地址的空白失效。这是一种保护。为了防止有人利用这里大做文章。

```
_dl_open() >> dl_open_worker() >> _dl_map_object() >> _dl_map_from_fd()
  1546       while (c < &loadcmds[nloadcmds])
  1547        {
  1548
  1549        postmap:
  1550    if (l->l_phdr == 0
  1551        && (ElfW(Off)) c->mapoff <= header->e_phoff
  1552        && ((size_t) (c->mapend - c->mapstart + c->mapoff)
  1553      >= header->e_phoff + header->e_phnum * sizeof (ElfW(Phdr))))
……
  1555      l->l_phdr = (void *) (c->mapstart + header->e_phoff - c->mapoff);
  1556
  1557    if (c->allocend > c->dataend)
  1558      {
……..
  1561        ElfW(Addr) zero, zeroend, zeropage;
  1562
  1563        zero = l->l_addr + c->dataend;
  1564        zeroend = l->l_addr + c->allocend;
  1565        zeropage = ((zero + GL(dl_pagesize) - 1)
  1566        & ~(GL(dl_pagesize) - 1));
  1567
  1568        if (zeroend < zeropage)
…….
  1571          zeropage = zeroend;
  1572
  1573        if (zeropage > zero)
  1574          {
…….
  1576      if ((c->prot & PROT_WRITE) == 0)
  1577        {
  1578          /* Dag nab it.  */
  1579        __mprotect ((caddr_t) (zero & ~(GL(dl_pagesize)
  1580            - 1)),  GL(dl_pagesize),
   1581                      c->prot|PROT_WRITE) < 0);
  1582
  1583        }
  1584      memset ((void *) zero, '\0', zeropage - zero);
  1585      if ((c->prot & PROT_WRITE) == 0)
  1586        __mprotect ((caddr_t) (zero & ~(GL(dl_pagesize) - 1)),
  1587             GL(dl_pagesize), c->prot);
  1588          }
  1589
  1590        if (zeroend > zeropage)
  1591          {
……..
  1593      caddr_t mapat;
  1594      mapat = __mmap ((caddr_t) zeropage, zeroend - zeropage,
  1595          c->prot, MAP_ANON|MAP_PRIVATE|MAP_FIXED,
  1596          ANONFD, 0);
  1597
  1598          }
  1599      }
  1600
  1601    ++c;
1602  }
```

这里所作的与上面的相类似，根据在前面从PT_LOAD program header 得到的文件映射的操作属性进行修改，但在zeroend>zerorpage的时候不同，把它映射成为进程独享的数据空间。这也就是一般的初始化数据区BSS的地方。因为zeroend是在文件中的映射的页面对齐尾地址，而zeropage是文件中的内容映射的页面对齐尾地址，这其中的差就是为未初始化数据准备的，这在1593-1597行之间体现，要把它的属性改成可写的，且全为0。

```
_dl_open() >> dl_open_worker() >> _dl_map_object() >> _dl_map_from_fd()
  1606  if (l->l_phdr == NULL)
  1607        {
……..
  1611    ElfW(Phdr) *newp = (ElfW(Phdr) *) malloc (header->e_phnum
  1612                * sizeof (ElfW(Phdr)));
  1613
  1614    l->l_phdr = memcpy (newp, phdr,
  1615            (header->e_phnum * sizeof (ElfW(Phdr))));
  1616    l->l_phdr_allocated = 1;
  1617        }
  1618      else
  1619        /* Adjust the PT_PHDR value by the runtime load address.  */
1620  (ElfW(Addr)) l->l_phdr += l->l_addr;
```

把phdr 就是program header 也纳入struct link_map的管理之中，一般的情况是不会有的，所以要copy过来。

```
  _dl_open() >> dl_open_worker() >> _dl_map_object() >> _dl_map_from_fd()
  1625       elf_get_dynamic_info (l);
```

这里调用的函数elf_get_dynamic_info是在加载过程中最重要的一个之一，因为在这之后的几乎所有的对动态链接管理的内容都要用要与这里的l_info数据组相关。

```
 _dl_open() >> dl_open_worker() >> _dl_map_object() >>
  _dl_map_from_fd() >> elf_get_dynamic_info()
  2826  static inline void __attribute__ ((unused, always_inline))
  2827  elf_get_dynamic_info (struct link_map *l)
  2828  {
  2829    ElfW(Dyn) *dyn = l->l_ld;
  2830    ElfW(Dyn) **info;
  2831
  2832
  2833    info = l->l_info;
  2834
  2835    while (dyn->d_tag != DT_NULL)
  2836      {
  2837        if (dyn->d_tag < DT_NUM)
  2838    info[dyn->d_tag] = dyn;
 ..............
  2853        ++dyn;
  2854      }
..............
  2858    if (l->l_addr != 0)
  2859      {
  2860        ElfW(Addr) l_addr = l->l_addr;
  2861
  2862        if (info[DT_HASH] != NULL)
  2863    info[DT_HASH]->d_un.d_ptr += l_addr;
  2864        if (info[DT_PLTGOT] != NULL)
  2865    info[DT_PLTGOT]->d_un.d_ptr += l_addr;
  2866        if (info[DT_STRTAB] != NULL)
  2867    info[DT_STRTAB]->d_un.d_ptr += l_addr;
  2868        if (info[DT_SYMTAB] != NULL)
  2869    info[DT_SYMTAB]->d_un.d_ptr += l_addr;
..................
  2874
..............
  2876        if (info[DT_REL] != NULL)
  2877    info[DT_REL]->d_un.d_ptr += l_addr;
..............
  2879
  2880        if (info[DT_JMPREL] != NULL)
  2881    info[DT_JMPREL]->d_un.d_ptr += l_addr;
  2882        if (info[VERSYMIDX (DT_VERSYM)] != NULL)
  2883    info[VERSYMIDX (DT_VERSYM)]->d_un.d_ptr += l_addr;
2884  }
..............
2889  }
```

上面的__attribute__ 中的unused 是为了消除编译器在-Wall 情况下对于其中可能没有用到在函数中的局部变量发出警告，而alwayse_inline，很好解释，就是内联函数的强制标志。

2829行的l->l_ld是在前面的__dl_map_object_from_fd中的1455被给定的。也就是所有关于动态链接节的所在地址（参看 附录B中的解释）。

很明显在2835至2854行之间的循环就是把l_info的内容都填充好。 这为之后有很大的作用，因为这些节是可以找到如函数名与定位信息的，这里的的妙处是把数组的偏移量与d_tag相关联，代码简洁。

2856至2885便是对动态链接库的调整过程（这里调整的每一个节都是与函数解析有重要关系的，详细内容可参看 附录A），如果我们考虑的更远一点，在前面的函数中的1521行一开始把整个文件连续的映射入内存，在这里就很好的得到解释，如果不是连续的，就没有办法在这里作一个统一的调整了。

```
  _dl_open() >> dl_open_worker() >> _dl_map_object() >> _dl_map_from_fd()
  1662    /* Finally the file information.  */
  1663    l->l_dev = st.st_dev;
  1664    l->l_ino = st.st_ino;
  1667    return l;
  1670  }
```

最后就是把设备号与节点号加入就完成了最后的dl_map_object就行了，回头看1414行中对已经加载的文件的搜索，就可以明白这里的作用了。

再回到dl_open_worker中

```
_dl_open() >> dl_open_worker()
  2550  /* It was already open.  */
  2551    if (new->l_searchlist.r_list != NULL)
```

```
  2552      {
……
  2556        if ((mode & RTLD_GLOBAL) && new->l_global == 0)
  2557    (void) add_to_global (new);
  2558
  2559        /* Increment just the reference counter of the object.  */
  2560        ++new->l_opencount;
  2561
  2562        return;
2563  }
```

这就是对已经被打开了的，就对l_opencount加一返回了。但为什么要在2551行之后作出这一判断呢，那是在下面的代码有关，_dl_map_object_deps会把l_searchlist加载入。

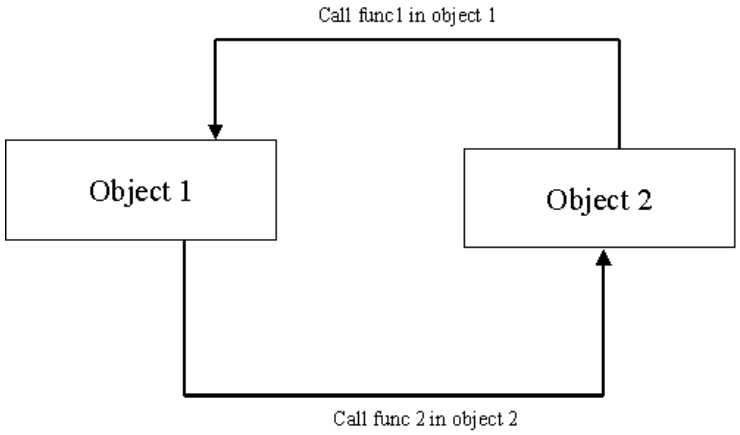
```
    _dl_open() >> dl_open_worker()
  2565    /* Load that object's dependencies.  */
  2566    _dl_map_object_deps (new, NULL, 0, 0, mode & __RTLD_DLOPEN);
……
  2573    l = new;
  2574    while (l->l_next)
  2575      l = l->l_next;
  2576    while (1)
  2577      {
  2578        if (! l->l_relocated)
  2579    {
  2580        _dl_relocate_object (l, l->l_scope, lazy, 0);
  2581    }
  2582
  2583        if (l == new)
  2584    break;
  2585        l = l->l_prev;
  2586      }
```

在这里的_dl_map_object_deps会填充l_searchlist.r_list，对于这个函数与下面的_dl_relocate_object由于与函数的解析关系比较大，所以我放在《Intel平台下linux中ELF文件动态链接的加载、解析及实例分析（中）----------函数解析与卸载篇》讲解。但可以把这个当作这个新加载的动态链接库的所依赖的动态链接库的struct link_map* 放入这个指针的列表中(就是l_search_list中)，_dl_relocate_object是对这个动态链接库中的函数重定位，而这里用的，这里之所以用的是while (1) 2576行，是因为在前面用的_dl_map_object_deps会把这个动态链接库所依赖的动态链接库也加载进来，这其中就会有没有重定位的。

```
_dl_open() >> dl_open_worker()
  2592    for (i = 0; i < new->l_searchlist.r_nlist; ++i)
  2593      if (++new->l_searchlist.r_list[i]->l_opencount > 1
  2594    && new->l_searchlist.r_list[i]->l_type == lt_loaded)
  2595        {
  2596    struct link_map *imap = new->l_searchlist.r_list[i];
  2597    struct r_scope_elem **runp = imap->l_scope;
  2598    size_t cnt = 0;
  2599
  2600    while (*runp != NULL)
  2601      {
 ……
  2605        if (*runp == &new->l_searchlist)
  2606          break;
  2607
  2608        ++cnt;
  2609        ++runp;
  2610      }
  2611
  2612    if (*runp != NULL)
  2613      /* Avoid duplicates.  */
  2614      continue;
……
  2642    imap->l_scope[cnt++] = &new->l_searchlist;
  2643    imap->l_scope[cnt] = NULL;
2644  }
```

这段代码如果从实现功能上来讲是很简单的，就是在我们刚新加入的动态链接库new中的l_searchlist中(这些都是在前面被dl_object_deps加载入的被依赖的动态链接库数组)imap->l_scope查找，如果里面runp有&new->l_searchlist，就不用对原来的imap->l_scope扩充了，但如果没有就要完成2616到2644行的扩充工作。

但在这之后的背景原因，却是&new->l_searchlist其实就是new本身。在一般情况下，如果这个依赖的动态链接库在new被加载之前已经加载（具体的原因会在下一篇文章关于动态链接库函数解析中说明），那就会遇到这种情况。而我们又不能保证两个动态链接库之间的互相依赖情况的发生，如下图，那这里的解决办法便是一个补救措施了。




```
_dl_open() >> dl_open_worker()
  2647    _dl_init (new, __libc_argc, __libc_argv, __environ);
```

这是要调用动态链接库自备的初始函数。这有点类似与**insmod**时调用的**init_module**的内容。至于这其中所传递的**__libc_argc, __libc_argv, __environ**三个参数是在你的可执行文件被运行的时候由**bash**引入的输入参数与环境变量，一般的动态链接库是没有什么用处了。

```
_dl_open() >> dl_open_worker()  >>  _dl_init()
  1118  void
  1119  internal_function
  1120  _dl_init (struct link_map *main_map, int argc, char **argv, char **env)
  1121  {
  1122
  1123    ElfW(Dyn) *preinit_array = main_map->l_info[DT_PREINIT_ARRAY];
  1124    ElfW(Dyn) *preinit_array_size = main_map->l_info[DT_PREINIT_ARRAYSZ];
  1125    unsigned int i;
  1126
  1127
  1128    ElfW(Addr) *addrs;
  1129    unsigned int cnt;
  1130
  1131
  1132    addrs = (ElfW(Addr) *) (preinit_array->d_un.d_ptr + main_map->l_addr);
  1133    for (cnt = 0; cnt < i; ++cnt)
  1134      (init_t) addrs[cnt]) (argc, argv, env);
……
  1146    i = main_map->l_searchlist.r_nlist;
  1147    while (i-- > 0)
  1148      call_init (main_map->l_initfini[i], argc, argv, env);
  1149
  1150
  1151
  1152
1153  }
```

先是调用 **DT_PREINIT**的内容，这是在**init**之的**init**方法。我想这个之所以要实现，不光是为让动态链接库的开发者有更好的开发接口，而且还是在以它所依赖的动态链接库之前进行一些初始化工作，借鉴于面向对象的构造函数。

```
   _dl_open() >> dl_open_worker()  >>  _dl_init()  >> call_init()
  1072  static void
  1073  call_init (struct link_map *l, int argc, char **argv, char **env)
  1074  {
  1075
  1076     if (l->l_init_called)
  1078      return;
  1079
  1082    l->l_init_called = 1;
……..
  1089    if (l->l_info[DT_INIT] != NULL)
  1090      {
  1091       init_t init = (init_t) DL_DT_INIT_ADDRESS(l, l->l_addr +
                                l->l_info[DT_INIT]->d_un.d_ptr);
  1092
  1093        /* Call the function.  */
  1094        init (argc, argv, env);
  1095      }
  1098    ElfW(Dyn) *init_array = l->l_info[DT_INIT_ARRAY];
  1099    if (init_array != NULL)
```

```
1100      {
1101        unsigned int j;
1102        unsigned int jm;
1103        ElfW(Addr) *addrs;
1104
1105        jm = l->l_info[DT_INIT_ARRAYSZ]->d_un.d_val / sizeof (ElfW(Addr));
1106
1107        addrs = (ElfW(Addr) *) (init_array->d_un.d_ptr + l->l_addr);
1108        for (j = 0; j < jm; ++j)
1109    ((init_t) addrs[j]) (argc, argv, env);
1110      }
1111
1112
1113  }
```

1076-1082行的内容一看便知，是防止两次初始化。下面是对**DT_INIT**与**DT_INIT_ARRAY**的函数调用，值得注意的是，前面调用**call_init**时是对**l_initfine**的数组进行的，这里就包括了这个新的动态链接库所依赖的。就这样完成了**dl_open_worker()**这个过程。

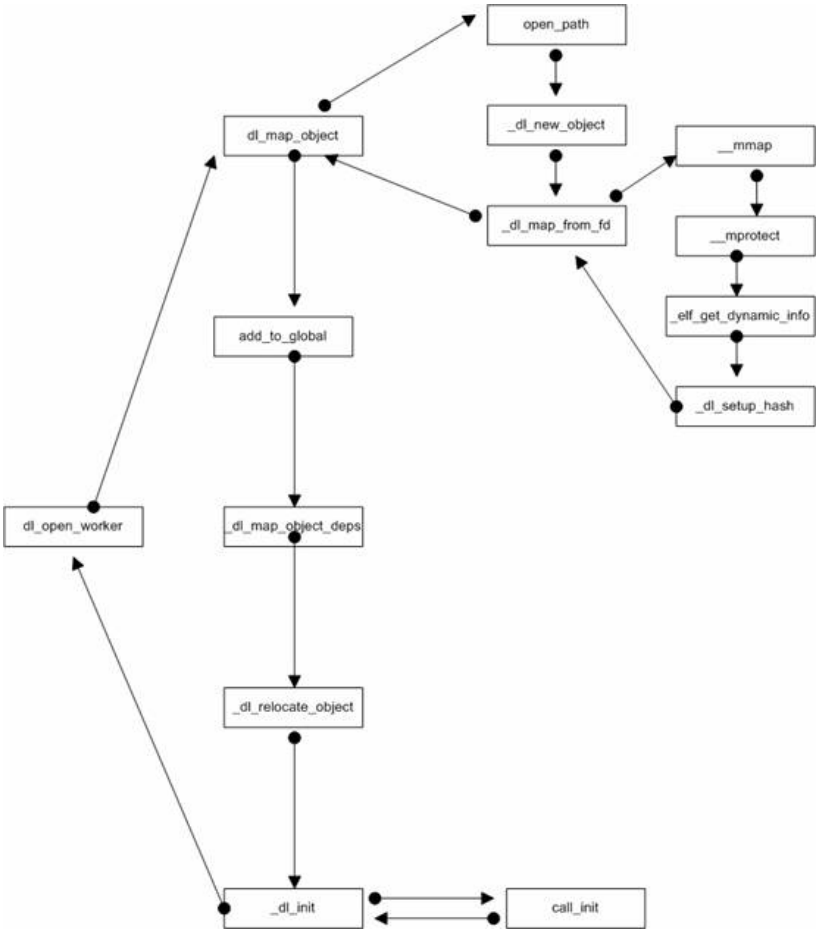
到此，我们至少大致上已经把动态链接库的过程说了一遍（当然，除了**_dl_map_object_deps**和**_dl_relocate_object**）到现在我们已经明白了以下几点：

1、 动态链接库的**struct link_map*** 的产生与组织过程（这个在**_dl_new_object**中实现）

2、 动态链接库是如何被提取信息入**struct link_map***中的，并被加载的**(**这个在**open_verify** 与**dl_map_object_from_fd**，**elf_get_dynamic_info**这三个函数中实现**)**

3、 动态链接库本身的初始化过程（这个在**_dl_init**中实现）

总体上函数调用结构在下图中一个示意图。



但还有几个问题没有被提到

1、 可执行文件中的函数被如何定位到动态链接库的函数体中的。

2、 一个动态链接库与依赖的动态链接库之间是什么关系，它们之间是如何联系。

3、 一个函数是怎样被动态解析，它又是使函数调用方与实现方成为一体的。

这些问题我会在《**Intel**平台下**linux**中**ELF**文件动态链接的加载、解析及实例分析（中）**----------**函数解析与卸载篇》进行阐明，敬请期待。

附录**A**：动态链接**section** 类型及说明

类型	数值	**d_un** 所指	**EXEC** 可选性	**DYN** 可选性	说明
DT_NULL	0	不用	必须	必须	这个表示动态链接section的结束标志
DT_NEEDED	1	d_val	可选	可选	这个节d_val是包含了以null结尾的字符串，这些字符串是这个动态链接文件或可执行文件的依赖文件名称与路径的节的开始地址
DT_PLTRELSZ	2	d_val	可选	可选	这里的d_val是过程链接表（procedure linkage table）的大小，它与DT_JMPREL结合使用
DT_PLTGOT	3	d_ptr	可选	可选	这里的d_ptr是过程链接表或全局偏移量表的起始地址。
DT_HASH	4	d_ptr	必须	必须	这里的d_val是符号哈希表的起始地址。
DT_STRTAB	5	d_ptr	必须	必须	这里d_ptr所给出的是符号名称字符串表的起始地址。
DT_SYMTAB	6	d_ptr	必须	必须	这里的d_ptr是Elf32_sym数据结构在的节表中的起始地址。
DT_STRSZ	10	d_val	必须	必须	这d_val是上面的DT_STRTAB节的大小。
DT_SYMENT	11	d_val	必须	必须	这里的d_val是DT_SYMTAB中的每个Elf32_Sym数据结构的大小
DT_INIT	12	d_ptr	可选	可选	这里的d_ptr是一个动态链接库被加载时调用的初始函数所在节的起始地址。
DT_FINI	13	d_ptr	可选	可选	这里的d_ptr是一个动态链接库被卸载时，调用解构函数所在节的起始地址。
DT_REL	17	d_ptr	必须	可选	这里的d_ptr与上面的DT_RELA相似，是Elf32_Rel数据结构所在节的起始地址，它在intel平台下用。
DT_RELSZ	18	d_val	必须	可选	这d_val与上面的DT_REL上面的相对应，表明上面的那个节的大小。
DT_RELENT	19	d_val	必须	可选	这里的d_val是DT_REL中的一个Elf32_Rel的数据结构的大小。
DT_PLTREL	20	d_val	可选	可选	这里的d_val是与过程链接表（procedure linkage table）有关的，就是DT_REL 或DT_RELA的值,也就是这个ELF文件用的是DT_REL的话那d_val就是17，而如果是DT_RELA的话就是7
DT_JMPREL	23	d_ptr	可选	可选	这是我们这里最重要的Elf_Dyn，因为d_ptr所指的就是GOT（global object table）全局对象表，这其实是一个导入函数与全局变量的地址表。
DT_INIT_ARRAY	25	d_ptr	可选	可选	这里的d_ptr是要初始化函数跳转表起始相对地址。
DT_FINI_ARRAY	26	d_ptr	可选	可选	这里的d_ptr是要解构时调用的函数跳转表起始相对地址。
DT_INIT_ARRAYSZ	27	d_val	可选	可选	这里的d_val表明前面的DT_INIT_ARRAY的大小。
DT_FINI_ARRAYSZ	28	d_val	可选	可	这里的d_val是前面的DT_FINI_ARRAY的大小。

				选	
DT_ENCODING	32	d_val或d_ptr	没有规定	没有规定	现在这个节还没有规定，但很明显就是为以后的加密而准备的。
DT_PREINIT_ARRAY	32	d_ptr	可选	不用	这里d_ptr是在调用main函数之前的调用初始函数跳转表的起始地址。
DT_PREINIT_ARRAYSZ	33	d_val	可选	不用	这里的d_val是前面的DT_PREINIT_ARRAY的大小

上面只列出了在我们这里要用到的项目，而ELF文件规范的设计者还为它留下了可以在不同的系统与平台中独自享用的项目，这里不列出了。

附录**B**：动态链接库**program header** 类型的说明

名称	值	说明
PT_NULL	0	这是program header 数组的分界标志符。
PT_LOAD	1	这个标志说明它所指的文件内容要被加载到内存单元，加载的内容由p_offset（在ELF文件中的偏移量）p_filesz（被加载的内容在文件中的大小）。而加载的要求是p_vaddr（被建议的加载的开始地址）p_memsz（被加载的建议内存大小）
PT_DYNAMIC	2	表示它所对应的dynamic section 内容，也就是在 附录A中所有的Elf32_Dyn数据结构所在的program heaer
PT_INTERP	3	这里所指的是一个字符串，它指的是为加载可执行文件而用的动态链接库名称，在linux下，这是/lib/ld-linux.so.2
PT_NOTE	4	为软件开发商加入标识而用的，表明软件的开发说明。
PT_SHLIB	5	这是为日后的扩充面预留。
PT_PHDR	6	表示program header array自身在内存中的映射地址与大小。

参考资料

John Levine "Linkers and Loaders" （是对动态链接的一般性理论作了一个概观介绍）可以在以下的网址上看到它的网络版 http://www.iecc.com/linker/

Executable and Linkable Format (ELF) （这专门介绍ELF文件格式的ABI的好文章，网络版在 www.skyfree.org/linux/references/ELF_Format.pdf可以得到）

glibc2-3-2版本 本文的源代码来源。可以在 ftp://ftp.gnu.org中下载而得。

IBM Bluemix 资源中心
文章、教程、演示，帮助您构建、部署和管理云应用。

developerWorks 中文社区
立即加入来自 IBM 的专业 IT 社交网络。

IBM 软件资源中心
免费下载、试用软件产品，构建应用并提升技能。