

Android so 加载深入分析

—从载入到链接

作者: ManyFace

Github: <https://github.com/ManyFace>

2016.2.7

目录

1 Java 层.....	3
2 Native 层	4
2.1 find_libraries()第一部分：初始化阶段.....	9
2.2 find_libraries()第二部分：载入 so 到内存	9
2.2.1 load_library()第一部分：打开 so 文件	12
2.2.2 load_library()第二部分：映射 so 文件到内存	12
2.2.3 load_library()第三部分：解析 dynamic section	21
2.3 find_libraries()第三部分：链接阶段.....	28
3 附录	37

本文对 Android 中 so 的加载进行了深入分析。通过分析相关源码，了解 linker 是如何将 so 文件加载到内存、如何进行链接操作。本文涉及的 Android 源码版本是 Android L。阅读本文之前，希望读者了解一点 ELF 的文件格式，如果对 ELF 完全不了解，可以参考我之前对 OAT 的分析 (<https://github.com/ManyFace/ExtractDexFromOat>)。由于水平有限，难免有所疏忽，分析不够到位或者错误的地方，还请各位指正。

1 Java 层

Android 在 java 层加载 so 的接口是 `System.loadLibrary()`，本文以此为突破口，逐步向下分析，得到 java 层函数的调用关系如图 1 所示。下面对每个函数进行详细分析。

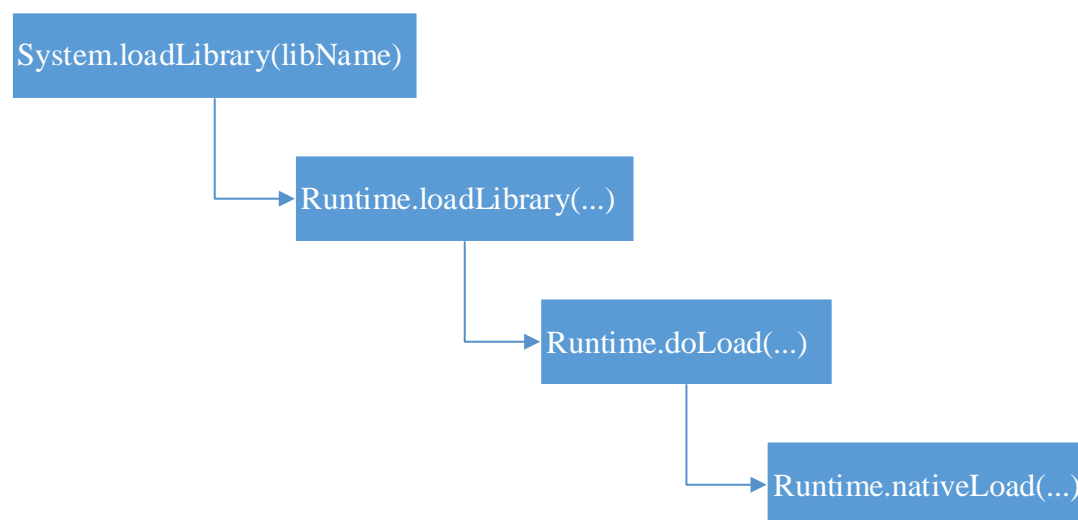


图 1 java 层函数调用关系图

`System.loadLibrary()` 的源码如下：

```
987 public static void loadLibrary(String libName) {  
988     Runtime.getRuntime().loadLibrary(libName, VMStack.getCallingClassLoader());  
989 }
```

可以看到它是调用了 `Runtime` 类的 `loadLibrary()`，其源码如下：

```
351 public void loadLibrary(String nickname) {  
352     loadLibrary(nickname, VMStack.getCallingClassLoader());  
353 }
```

从上可知，`loadLibrary(String nickname)` 调用了它的一个重载函数 `loadLibrary(String libraryName, ClassLoader loader)`，其关键代码如下：

```
358 void loadLibrary(String libraryName, ClassLoader loader) {  
359     if (loader != null) {  
360         String filename = loader.findLibrary(libraryName); //so 路径  
361         ... ..
```

```

369         String error = doLoad(filename, loader); //加载
370         ... ..
373         return;
374     }
376     ... ..
396 }

```

360 行获得 so 文件的绝对路径 filename，369 行调用 doLoad()来加载 so 文件。doLoad(String name, ClassLoader loader)的源码如下：

```

400     private String doLoad(String name, ClassLoader loader) {
420         String ldLibraryPath = null;
421         if (loader != null && loader instanceof BaseDexClassLoader) {
422             ldLibraryPath = ((BaseDexClassLoader) loader).getLdLibraryPath();
423         }
427         synchronized (this) {
428             return nativeLoad(name, loader, ldLibraryPath);
429         }
430     }

```

428 行调用 nativeLoad()来加载 name 指向的.so 文件，nativeLoad()是 Runtime 类的一个 native 函数，在 native 层对应的函数为 Runtime_nativeLoad()。

2 Native 层

大家在看 native 层的分析时，可以结合附录中的图 11(字有点小⊗)。从图 11 可知，本文主要是分析 dlopen()的实现。下面开始分析 native 层的代码。

Runtime_nativeLoad()的关键代码如下：

```

46 static jstring Runtime_nativeLoad(JNIEnv* env, jclass, jstring javaFilename,
  jobject javaLoader, jstring javaLdLibraryPath) {
47     ScopedUtfChars filename(env, javaFilename);
51     ... ..
67     std::string detail;
68     {
69         ScopedObjectAccess soa(env);
70         StackHandleScope<1> hs(soa.Self());
71         Handle<mirror::ClassLoader> classLoader(
72             hs.NewHandle(soa.Decode<mirror::ClassLoader*>(javaLoader)));
73         JavaVMExt* vm = Runtime::Current()->GetJavaVM();
74         bool success = vm->LoadNativeLibrary(filename.c_str(), classLoader,
  &detail); //加载 so
75         if (success) {
76             return nullptr;

```

```
77     }  
78 }
```

74 行调用 JavaVMExt 类的 LoadNativeLibrary() 函数来加载 .so 文件，filename 是 .so 文件的路径，detail 用于存储加载过程中的 log 信息。LoadNativeLibrary() 的关键代码如下：

```
bool JavaVMExt::LoadNativeLibrary(const std::string& path,  
3226                                     Handle<mirror::ClassLoader> class_loader,  
3227                                     std::string* detail) {  
3228     detail->clear();  
3229 //首先检查是否加载过该 .so，加载过就不需要重复加载  
3234     SharedLibrary* library;  
3235     Thread* self = Thread::Current();  
3236     {  
3238         MutexLock mu(self, libraries_lock);  
3239         library = libraries->Get(path);  
3240     }  
3241     if (library != nullptr) {  
3242         if (library->GetClassLoader() != class_loader.Get()) {  
3246             StringAppendF(detail, "Shared library \"%s\" already opened by "  
3247                             "ClassLoader %p; can't open in ClassLoader %p",  
3248                             path.c_str(), library->GetClassLoader(), class_loader.Get());  
3249             LOG(WARNING) << detail;  
3250             return false;  
3251         }  
3252         ... ..  
3254         if (!library->CheckOnLoadResult()) {  
3255             StringAppendF(detail, "JNI_OnLoad failed on a previous attempt "  
3256                             "to load \"%s\"", path.c_str());  
3257             return false;  
3258         }  
3259         return true;  
3260     }  
3261 //之前没有加载过  
3275     self->TransitionFromRunnableToSuspended(kWaitingForJniOnLoad);  
3276     const char* path_str = path.empty() ? nullptr : path.c_str();  
3277     void* handle = dlopen(path_str, RTLD_LAZY); //用 dlopen 加载 .so 文件  
3279     ... ..  
3285     self->TransitionFromSuspendedToRunnable();  
3286     ... ..  
3297     bool created_library = false;  
3298     {  
3299         MutexLock mu(self, libraries_lock);
```

```

3300     library = libraries->Get(path);
3301     if (library == nullptr) {
3302         //加载完成后，新建 SharedLibrary 对象，并以 path 为键值将其存入 libraries.
3303         library = new SharedLibrary(path, handle, class_loader.Get());
3304         libraries->Put(path, library);
3305         created_library = true;
3306     }
3307 }
3308 ... ..
3309 bool was_successful = false;
3310 void* sym = nullptr;
3311 if (UNLIKELY(needs_native_bridge)) {
3312     library->SetNeedsNativeBridge();
3313     sym = library->FindSymbolWithNativeBridge("JNI_OnLoad", nullptr);
3314 } else {
3315     sym = dlsym(handle, "JNI_OnLoad"); //找到 JNI_OnLoad 函数
3316 }
3317
3318 if (sym == nullptr) {
3319     VLOG(jni) << "[No JNI_OnLoad found in \"" << path << "\"]";
3320     was_successful = true;
3321 } else {
3322     typedef int (*JNI_OnLoadFn)(JavaVM*, void*);
3323     JNI_OnLoadFn jni_on_load = reinterpret_cast<JNI_OnLoadFn>(sym);
3324     ... ..
3325     int version = 0;
3326     {
3327         ScopedThreadStateChange tsc(self, kNative);
3328         VLOG(jni) << "[Calling JNI_OnLoad in \"" << path << "\"]";
3329         version = (*jni_on_load)(this, nullptr); //调用 JNI_OnLoad 函数
3330     }
3331     ... ..
3332     return was_successful;
3333 }

```

从上可知，LoadNativeLibrary() 函数执行的主要流程为：

1. 3234-3260 行，判断该 .so 文件是否已经加载了，如果已经加载了，检查 class_loader 是否一样；
2. 3277 行，如果没有加载，调用 dlopen() 函数加载该 .so 文件；
3. 3322 行，调用 dlsym() 找到 JNI_OnLoad 函数的地址；
4. 3343 行，调用 JNI_OnLoad 函数。

至此，一个 .so 文件就加载完成了。接下来，我们详细分析 dlopen() 函数，

了解一个 .so 文件是如何载入内存、如何链接的。

dlopen() 函数的源码如下：

```
82 void* dlopen(const char* filename, int flags) {
83     return dlopen_ext(filename, flags, nullptr);
84 }
```

83 行调用 dlopen_ext()，其实现为：

```
68 static void* dlopen_ext(const char* filename, int flags, const
android_dlextinfo* extinfo) { //extinfo 为 null
69     ScopedPthreadMutexLocker locker(&g_dl_mutex);
70     soinfo* result = do_dlopen(filename, flags, extinfo);
71     if (result == nullptr) {
72         __bionic_format_dLError("dlopen failed", linker_get_error_buffer());
73         return nullptr;
74     }
75     return result;
76 }
```

70 行调用 do_dlopen 来加载 filename 指向的 .so 文件，返回值为 soinfo 对象的指针，因而 dlopen() 函数的返回的指针指向一个 soinfo 对象。

do_dlopen() 函数的关键代码如下：

```
1041 soinfo* do_dlopen(const char* name, int flags, const android_dlextinfo*
extinfo) { //extinfo 为 null
1042     ... ..
1057     protect_data(PROT_READ | PROT_WRITE);
1058     soinfo* si = find_library(name, flags, extinfo); //加载链接 name
1059     if (si != nullptr) {
1060         si->CallConstructors(); //初始化
1061     }
1062     protect_data(PROT_READ);
1063     return si;
1064 }
```

1058 行调用 find_library() 函数得到 soinfo 的对象，1060 行调用 si->CallConstructors() 进行初始化。我们首先分析 find_library()，由于 find_library() 的分析占了绝大数篇幅，导致 CallConstructors() 几乎在文末的时候分析。find_library() 函数的关键代码如下：

```
968 static soinfo* find_library(const char* name, int dlflags, const
android_dlextinfo* extinfo) {
969     ... ..
974     soinfo* si;
976     if (!find_libraries(&name, 1, &si, nullptr, 0, dlflags, extinfo)) {
977         return nullptr;

```

```
978 }  
980 return si;  
981}
```

976 行调用 find_libraries(), 其关键代码如下:

```
896 static bool find_libraries(const char* const library_names[], size_t  
library_names_size, soinfo* soinfos[], soinfo* ld_preloads[], size_t  
ld_preloads_size, int dlflags, const android_dlexthinfo* extinfo) {  
    //library_names 是 .so 文件名的数组, 当然这里它只有一个元素  
    //library_names_size=1  
    //soinfos 也只有一个 soinfo 对象的指针, 将加载 library_names 中 so 的结果存入 soinfos  
    //ld_preloads=null  
    //ld_preloads_size=0  
    //extinfo=null  
898    // Step 0: prepare.  
        //part 1  
        //宽度优先搜索的栈, 父节点的依赖库为其子节点, 根结点是待加载的 .so 文件  
899    LoadTaskList load_tasks;  
900    for (size_t i = 0; i < library_names_size; ++i) {  
901        const char* name = library_names[i];  
902        load_tasks.push_back(LoadTask::create(name, nullptr));  
903    }  
904  
905    // Libraries added to this list in reverse order so that we can  
906    // start linking from bottom-up - see step 2.  
907    SoinfoLinkedList found_libs; //该 .so 文件和其所有依赖库的列表  
908    size_t soinfos_size = 0;  
909    ... ..  
        //part 2  
920    //采用宽度优先搜索加载该 .so 文件和其依赖库  
921    // Step 1: load and pre-link all DT_NEEDED libraries in breadth first order.  
922    for (LoadTask::unique_ptr task(load_tasks.pop_front()); task.get() !=  
nullptr; task.reset(load_tasks.pop_front())) {  
923        soinfo* si = find_library_internal(load_tasks, task->get_name(), dlflags,  
extinfo); //extinfo==null  
924        ... ..  
928        soinfo* needed_by = task->get_needed_by(); //needed_by 依赖 si  
929  
930        if (is_recursive(si, needed_by)) { //判断是否出现递归依赖关系  
931            return false;  
932        }  
933  
934        si->ref_count++; //si 的引用数加 1
```



```

935     if (needed_by != nullptr) {
936         needed_by->add_child(si); //needed_by 依赖 si
937     }
938     found_libs.push_front(si);
939     ... ...
946     if (soinfos_size < library_names_size) {
947         //只将 library_names 中对应的 soinfo 存入 soinfos
948         soinfos[soinfos_size++] = si;
949     }
950     //part 3
951     //链接加载的库
952     // Step 2: link libraries.
953     soinfo* si;
954     while ((si = found_libs.pop_front()) != nullptr) {
955         if ((si->flags & FLAG_LINKED) == 0) { //如果 si 没有链接, 对 si 进行链接
956             if (!si->LinkImage(extinfo)) { //extinfo=null
957                 return false;
958             }
959             si->flags |= FLAG_LINKED;
960         }
961     }
962     ... ...
965     return true;
966 }

```

find_libraries() 将数组 library_names[] 中的 so 文件加载到内存, 并进行链接。这里将 find_libraries() 分为三个部分来进行分析。第一部分 (899-908 行): 初始化阶段; 第二部分 (922-949 行): 采用宽度优先搜索加载 so; 第三部分 (952-960 行): 对加载的 so 进行链接。下面对这三个部分进行详细分析。

2.1 find_libraries() 第一部分: 初始化阶段

要加载的 so 可能依赖于其他库, linker 采用宽度优先搜索依次加载 so 及其依赖库。搜索树中父节点的依赖库为其子节点, 根节点是待加载的 .so 文件。

899 行 load_tasks 是用于宽度优先搜索的栈, 900-903 行对其进行初始化。907 行 found_libs 是 .so 文件和其依赖库的列表。

2.2 find_libraries() 第二部分: 载入 so 到内存

这部分对 .so 文件及其依赖库按照宽度优先的顺序依次进行加载, 最关键的是 923 行调用 find_library_internal() 函数, 用于加载 so, 其实现如下:

```

865 static soinfo* find_library_internal(LoadTaskList& load_tasks, const char*
name, int dlflags, const android_dlexinfo* extinfo) { //extinfo=null
867     soinfo* si = find_loaded_library_by_name(name); //检查是否被加载过
871     if (si == nullptr) { //加载过就直接返回 si, 否则, 调用 load_library 加载
872         TRACE("[ '%s' has not been found by name. Trying harder...]", name);
873         si = load_library(load_tasks, name, dlflags, extinfo);
874     }
876     return si;
877 }

```

find_library_internal() 首先会检查 name 指向的 .so 是否已经加载, 如果没有, 就调用 load_library() 加载, 其关键代码如下:

```

777 static soinfo* load_library(LoadTaskList& load_tasks, const char* name, int
dlflags, const android_dlexinfo* extinfo) { //extinfo=null
778     int fd = -1; // .so 的文件描述符
779     off64_t file_offset = 0;
780     ScopedFd file_guard(-1);
781     //part 1
782     if (extinfo != nullptr && (extinfo->flags & ANDROID_DLEX_USE_LIBRARY_FD) !=
0) { ... ..
787 } else {
788     // Open the file.
789     fd = open_library(name); //打开 .so 文件
790     ... ..
796 }
797 //文件偏移必须是 PAGE_SIZE 的整数倍, 这里 file_offset=0
798 if ((file_offset % PAGE_SIZE) != 0) {
799     ... ..
800     return nullptr;
801 }
802
803 struct stat file_stat;
804 if (TEMP_FAILURE_RETRY(fstat(fd, &file_stat)) != 0) { //获取 .so 文件的状态
805     ... ..
806     return nullptr;
807 }
809 // Check for symlink and other situations where
810 // file can have different names.
    //linux 下可以生成文件的链接文件, 这里检查 .so 文件是否以不同的名字加载
811 for (soinfo* si = solist; si != nullptr; si = si->next) {
812     if (si->get_st_dev() != 0 &&
813         si->get_st_ino() != 0 &&
814         si->get_st_dev() == file_stat.st_dev &&

```

```

815     si->get_st_ino() == file_stat.st_ino &&
816     si->get_file_offset() == file_offset) {
817     TRACE("library \"%s\" is already loaded under different name/path \"%s\" -
will return existing soinfo", name, si->name);
818     return si;
819 }
820 }
821 ... ...

//part 2
827 // Read the ELF header and load the segments.
//读取 ELF 头，加载段
828 ElfReader elf_reader(name, fd, file_offset); //file_offset=0
829 if (!elf_reader.Load(extinfo)) {
830     return nullptr;
831 }

//part 3
832 //为 soinfo 分配空间
833 soinfo* si = soinfo_alloc(SEARCH_NAME(name), &file_stat, file_offset);
834 if (si == nullptr) {
835     return nullptr;
836 }
837 si->base = elf_reader.load_start(); //加载 so 文件时，mmap 得到的空间的首地址
838 si->size = elf_reader.load_size(); //ReserveAddressSpace 中开辟的内存空间的大小
//加载段时的基址，load_bias+p_vaddr 为段的实际内存地址
839 si->load_bias = elf_reader.load_bias();
840 si->phnum = elf_reader.phdr_count(); //program header 的个数
841 si->phdr = elf_reader.loaded_phdr(); //program header table 在内存中的起始地址
842
843 if (!si->PrelinkImage()) { //解析.dynamic section
844     soinfo_free(si);
845     return nullptr;
846 }
847 //将该 so 文件依赖的库添加到待加载队列中
848 for_each_dt_needed(si, [&] (const char* name) {
849     load_tasks.push_back(LoadTask::create(name, si)); //si 依赖于名为 name 的库
850 });
851
852 return si;
853}

```

我们将 `load_library()` 分为三个部分来进行分析。第一部分(782-820 行)：主要作用是打开 .so 文件，并判断是否已经加载；第二部分(828-831 行)：加载 .so 文件的可加载段；第三部分(832-850 行)：创建 `soinfo` 对象，解析 `.dynamic`

section，并将该.so 文件的依赖库添加到待加载的队列中。下面对这三个部分进行详细分析。

2.2.1 load_library() 第一部分：打开 so 文件

798 行内存页的大小 PAGE_SIZE 为 4096，定义位于头文件 /bionic/libc/include/limits.h，现在最好通过 sysconf(_SC_PAGE_SIZE) 来获取 PAGE_SIZE 的值，sysconf() 位于 /bionic/libc/bionic/sysconf.cpp，从 sysconf 的实现可以知道，sysconf(_SC_PAGESIZE) 也可以用来获取 PAGE_SIZE 的值。

803-820 行代码的主要用途是检查.so 文件是否以不同的文件名被加载过了。Linux 下一个文件可以有多个链接文件，因而不同的文件名可能指向的是同一个文件。

2.2.2 load_library() 第二部分：映射 so 文件到内存

828-831 行代码用 ElfReader 类解析 ELF 头，并根据 program header table 加载段。其成员函数 Load() 实现如下：

```
135 bool ElfReader::Load(const android_dlextinfo* extinfo) {
136     return ReadElfHeader() &&
137         VerifyElfHeader() &&
138         ReadProgramHeader() &&
139         ReserveAddressSpace(extinfo) &&
140         LoadSegments() &&
141         FindPhdr();
142 }
```

ReadElfHeader() 用于读取 ELF 的头，并将结果赋给 ElfReader 的成员变量 Elf32_Ehdr header_，Elf32_Ehdr 的定义可以在 /art/runtime/elf.h 中找到，自动生成的文件 /bionic/libc/kernel/uapi/linux/elf.h 中也有相关 elf 的定义。

VerifyElfHeader() 用于检查 ELF 头某些字段是否合法，其实现如下：

```
159 bool ElfReader::VerifyElfHeader() {
    //检查 magicNum 是否为 "\177ELF"
160     if (memcmp(header_.e_ident, ELF_MAG, SELF_MAG) != 0) {
161         DL_ERR("\'%s\' has bad ELF magic", name_);
162         return false;
163     }
164     //检查其位数是否与目前系统同为 32 位(1)或 64 位(2)
165     int elf_class = header_.e_ident[EI_CLASS]; // EI_CLASS=4
166 #if defined(__LP64__)
167     if (elf_class != ELFCLASS64) {
168         if (elf_class == ELFCLASS32) {
```

```

171     DL_ERR("\'%s\' is 32-bit instead of 64-bit", name_);
172 } else {
173     DL_ERR("\'%s\' has unknown ELF class: %d", name_, elf_class);
174 }
175 return false;
176 }
177 #else
178 if (elf_class != ELFCLASS32) {
179     if (elf_class == ELFCLASS64) {
180         DL_ERR("\'%s\' is 64-bit instead of 32-bit", name_);
181     } else {
182         DL_ERR("\'%s\' has unknown ELF class: %d", name_, elf_class);
183     }
184     return false;
185 }
186 #endif
187 //该.so文件必须是小端字节序
188 if (header_.e_ident[EI_DATA] != ELFDATA2LSB) { //EI_DATA=5, ELFDATA2LSB=1
189     DL_ERR("\'%s\' not little-endian: %d", name_, header_.e_ident[EI_DATA]);
190     return false;
191 }
192 //该.so文件必须是共享目标文件
193 if (header_.e_type != ET_DYN) { //ET_DYN=3
194     DL_ERR("\'%s\' has unexpected e_type: %d", name_, header_.e_type);
195     return false;
196 }
197 //版本号必须为1
198 if (header_.e_version != EV_CURRENT) { //EV_CURRENT=1
199     DL_ERR("\'%s\' has unexpected e_version: %d", name_, header_.e_version);
200     return false;
201 }
202 //如果目标平台是 arm, 那么 ELF_TARG_MACH=40
203 if (header_.e_machine != ELF_TARG_MACH) {
204     DL_ERR("\'%s\' has unexpected e_machine: %d", name_, header_.e_machine);
205     return false;
206 }
207
208 return true;
209 }

```

VerifyElfHeader() 校验的部分如上面的红色注释所示, 可见 ELF 头中, byte e_ident[16] 字段的后 10 位并没有进行校验。

ReadProgramHeader() 将 program header table 从 .so 文件通过 mmap64 映

射到只读私有匿名内存，其实现如下：

```
213 bool ElfReader::ReadProgramHeader() {
214     phdr_num_ = header_.e_phnum; //phdr 的数目
216     // Like the kernel, we only accept program header tables that
217     // are smaller than 64KiB.
218     if (phdr_num_ < 1 || phdr_num_ > 65536/sizeof(ElfW(Phdr))) {
219         DL_ERR(“\” %s\” has invalid e_phnum: %zd”, name_, phdr_num_);
220         return false;
221     }
222
223     ElfW(Addr) page_min = PAGE_START(header_.e_phoff); //0
224     ElfW(Addr) page_max = PAGE_END(header_.e_phoff + (phdr_num_ *
sizeof(ElfW(Phdr))));
225     ElfW(Addr) page_offset = PAGE_OFFSET(header_.e_phoff); //pht 在页中的偏移
226
227     phdr_size_ = page_max - page_min; //需要为 pht 映射内存的大小
228
229     void* mmap_result = mmap64(nullptr, phdr_size_, PROT_READ, MAP_PRIVATE, fd_,
file_offset_ + page_min);
230     ... ..
235     phdr_mmap_ = mmap_result;
236     phdr_table_ =
reinterpret_cast<ElfW(Phdr)*>(reinterpret_cast<char*>(mmap_result) + page_offset);
237     return true;
238 }
```

223-227 行代码用于计算映射 program header table 需要的内存大小，以及偏移。宏 PAGE_START，PAGE_END，PAGE_OFFSET 定义的文件位于 /bionic/linker/linker.h，分别为：

- #define PAGE_START(x) ((x) & PAGE_MASK)

由前面的分析可以知道，内存页的大小是 $4096(2^{12})$ ，因而内存页的起始地址低 12 位应该为 0。PAGE_START(x) 就是计算地址 x 所在内存页的起始地址，其中 PAGE_MASK 的低 12 位为 0，其他位是 1。

- #define PAGE_OFFSET(x) ((x) & ~PAGE_MASK)

PAGE_OFFSET(x) 用于计算地址 x 在其所在内存页中的偏移。

- #define PAGE_END(x) PAGE_START((x) + (PAGE_SIZE-1))

PAGE_END(x) 用于计算地址 x 所在页的结束地址(注意不包含 PAGE_END(x))，其实就是地址 x 所在内存页的下一页的起始地址。例如地址 $x=0x1243B3C1$ ，那么它所在内存页 A 的起始地址 $PAGE_START(x)=0x1243B000$ ，它在页 A 中的偏移 $PAGE_OFFSET(x)=0x3C1$ ，页 A 的结束地址 $PAGE_END(x)=0x1243C000$ ，即页 A 最后

一个字节的地址为 0x1243BFFF。

`ReserveAddressSpace()` 通过 `mmap` 创建足够大的匿名内存空间,以便能够容纳所有可以加载的段,其关键代码如下:

```
292 bool ElfReader::ReserveAddressSpace(const android_dlexthinfo* extinfo) {
293     ElfW(Addr) min_vaddr;
294     //加载所有段所需要的内存空间
295     load_size_ = phdr_table_get_load_size(phdr_table_, phdr_num_, &min_vaddr);
296     if (load_size_ == 0) {
297         DL_ERR("\"%s\" has no loadable segments", name_);
298         return false;
299     }
300     uint8_t* addr = reinterpret_cast<uint8_t*>(min_vaddr);
301     void* start;
302     size_t reserved_size = 0;
303     bool reserved_hint = true;
304     ... ..
314     if (load_size_ > reserved_size) {
315         ... ..
320         int mmap_flags = MAP_PRIVATE | MAP_ANONYMOUS;
321         start = mmap(addr, load_size_, PROT_NONE, mmap_flags, -1, 0); //分配空间
322         ... ..
324         return false;
325     }
326 }
329 ... ..
330 load_start_ = start; //分配的匿名内存空间的首地址
331 load_bias_ = reinterpret_cast<uint8_t*>(start) - addr;
332 return true;
333 }
```

为了理解 `load_bias_` 的作用,以及 linker 是如何将 .so 文件映射到内存的,我们先看一下 `/bionic/linker/linker_phdr.cpp` 中的一段注释,大致翻译并结合了一些自己的理解,具体内容如下:

ELF 文件的 program header table 中包含了一个或者多个可加载段,可加载段的标志为 `PT_LOAD`,这些段需要被映射到该进程的地址空间中。

可加载段重要的一些属性如下:

- `p_offset`: 段在文件中的偏移
- `p_filesz`: 段在文件中的大小
- `p_memsz`: 段在内存中的大小,总是大于或等于 `p_filesz`。
- `p_vaddr`: 段的虚拟地址

- `p_flags`: 段 flags(读、写、执行等)

目前我们暂时忽略了 `p_paddr` 和 `p_align` 字段。

所有的可加载段可以看做虚拟地址范围的列表: `[p_vaddr ... p_vaddr+p_memsz)`, 满足的条件如下:

- 虚拟地址范围不能够重叠
- 如果段的 `p_filesz` 小于 `p_memsz`, 内存中多出的部分初始化为 0
- 地址的边界不需要位于页的边界。如果同一个页中包含两个不同的段, 那么这个页的属性继承于后一个段的映射标志

例如, 考虑如下的可加载段:

```
[ offset:0,      filesz:0x4000, memsz:0x4000, vaddr:0x30000 ],  
[ offset:0x4000, filesz:0x2000, memsz:0x8000, vaddr:0x40000 ],
```

这两个可加载段的虚拟地址范围为:

`0x30000...0x34000`

`0x40000...0x48000`

如果 loader 在地址 `load_start_=0xa0000000` 处加载第一个段, 那么这两个段在内存中实际加载的地址范围是:

`0xa0030000...0xa0034000`

`0xa0040000...0xa0048000`

也就是说, 所有的段在本进程的虚拟内存中的起始地址分别减去各自的 `p_vaddr`, 得到的偏移应该是相同的, 比如上面的例子得到的偏移都是 `0xa0000000`。

然而, 实际上段的起始地址并不都是在页的边界, 如果上面第一个段的为 `[offset:0x31020, filesz:0x1000, memsz:0x1000, vaddr:0x31020]`, 按照如上的规则, 两个段在内存中的地址范围是:

`0xa0031020...0xa0032020`

`0xa0040000...0xa0048000`

从上可以看出, 地址空间 `[0xa0000000, 0xa0031020)` 是被浪费了。由于在进行内存映射的时候, 内存中的地址必须是一个页的起始地址, 同时文件偏移量也必须是页大小的整数倍。为了让第一个段映射在 `load_start_=0xa0000000` 后的第一个页中, 需要重新引入一个偏移量 `load_bias_`:

$$\text{load_bias_} = \text{load_start_} - \text{PAGE_START}(\text{p_vaddr})$$

这样, 段在内存中的起始虚拟地址 `seg_start = load_bias_ + p_vaddr`, 而 `seg_start` 所在页的起始地址 `seg_page_start=PAGE_START(seg_start)`, 这样就可以从 `seg_page_start` 处进行映射。从中可以看出, 如果段在文件中的偏移不是页的整数倍, 那么映射后, 内存中 `[seg_page_start, seg_start)` 区域是文

件中段前面的 `seg_start - seg_page_start` 个字节的内容，内存中的段内容是从 `seg_start` 开始。

注意 ELF 文件如果要通过 `mmap` 进行内存映射，需要满足条件：
`PAGE_OFFSET(phdr0->p_vaddr) == PAGE_OFFSET(phdr0->p_offset)`

通过上面的内容，知道了 `.so` 文件映射到内存的基本原理。下面继续分析 `LoadSegments()` 函数，理解加载的具体过程。`LoadSegments()` 函数的关键代码如下：

```
335 bool ElfReader::LoadSegments() {
336     for (size_t i = 0; i < phdr_num_; ++i) {
337         const ElfW(Phdr)* phdr = &phdr_table_[i];
338         //遍历 program header table 找到可加载段
339         if (phdr->p_type != PT_LOAD) {
340             continue;
341         }
342         // Segment addresses in memory.
343         ElfW(Addr) seg_start = phdr->p_vaddr + load_bias_; //段在内存中的起始地址
344         ElfW(Addr) seg_end   = seg_start + phdr->p_memsz; //段在内存中的结束地址
345         //seg_start 所在页的起始地址
346         ElfW(Addr) seg_page_start = PAGE_START(seg_start);
347         //seg_end 所在页的下一页的起始地址
348         ElfW(Addr) seg_page_end   = PAGE_END(seg_end);
349         //文件中段的结束位置在内存中的地址
350         ElfW(Addr) seg_file_end   = seg_start + phdr->p_filesz;
351
352         // File offsets.
353         ElfW(Addr) file_start = phdr->p_offset; //段在文件中的偏移
354         ElfW(Addr) file_end   = file_start + phdr->p_filesz; //段在文件中的结束地址
355         //file_start 所在页的起始地址
356         ElfW(Addr) file_page_start = PAGE_START(file_start);
357         //需要映射的文件长度，file_length>=phdr->p_filesz
358         ElfW(Addr) file_length = file_end - file_page_start;
359
360         if (file_length != 0) { //将文件中的段映射到内存
361             void* seg_addr = mmap64(reinterpret_cast<void*>(seg_page_start),
362                                     file_length,
363                                     PFLAGS_TO_PROT(phdr->p_flags),
364                                     MAP_FIXED|MAP_PRIVATE,
365                                     fd_,
366                                     file_offset_ + file_page_start);
367             ... ..
368         }
369     }
370 }
```

```

371
372 // if the segment is writable, and does not end on a page boundary,
373 // zero-fill it until the page limit.
374 if ((phdr->p_flags & PF_W) != 0 && PAGE_OFFSET(seg_file_end) > 0) {
375     memset(reinterpret_cast<void*>(seg_file_end), 0, PAGE_SIZE -
PAGE_OFFSET(seg_file_end)); //将最后一页中，不是段内容的数据置 0
376 }
377
378 seg_file_end = PAGE_END(seg_file_end);
380 // seg_file_end is now the first page address after the file
381 // content. If seg_end is larger, we need to zero anything
382 // between them. This is done by using a private anonymous
383 // map for all extra pages.
384 if (seg_page_end > seg_file_end) {
385     void* zeromap = mmap(reinterpret_cast<void*>(seg_file_end),
386                           seg_page_end - seg_file_end,
387                           PFLAGS_TO_PROT(phdr->p_flags),
388                           MAP_FIXED|MAP_ANONYMOUS|MAP_PRIVATE,
389                           -1,
390                           0); //额外的内容置 0
391     ... ..
395 }
396 }
397 return true;
398}

```

LoadSegments() 函数的执行流程如上面的红色注释所示：遍历 program header table，找到可加载段，并通过 mmap 将可加载段从文件映射到内存。这里我们来看一个实际的例子：libapp.so 有两个可加载段，第一个可加载段可读可执行，第二个可加载段可读可写。这两个段的基本信息如图 2 所示：

p_offset	0
p_vaddr	0
p_paddr	0
p_filesz	1D4F
p_mems	1D4F

a. 第一个可加载段

p_offset	1EA8
p_vaddr	2EA8
p_paddr	2EA8
p_filesz	168
p_mems	168

b. 第二个可加载段

图 2 libapp.so 中可加载段信息

通过对源码加入 log 信息，我们知道加载 libapp.so 时，load_start_ = load_bias_ = B43EE000。加载第一个可加载段的相关信息如图 3 所示，此时内存与文件的映射关系如图 4 所示(黄色表示可加载段，红线是页的边界)，文件内容[0, 2000)映射到内存[B43EE000, B43F0000)中，其中可加载段一[0, 1D4F)在

内存中位于[B43EE000, B43EFD4F)。

seg_page_start	B43EE000
seg_page_end	B43F0000
file_page_start	0
file_length	1D4F
seg_file_end	B43EFD4F
seg_start	B43EE000

图 3 第一个可加载段的加载信息

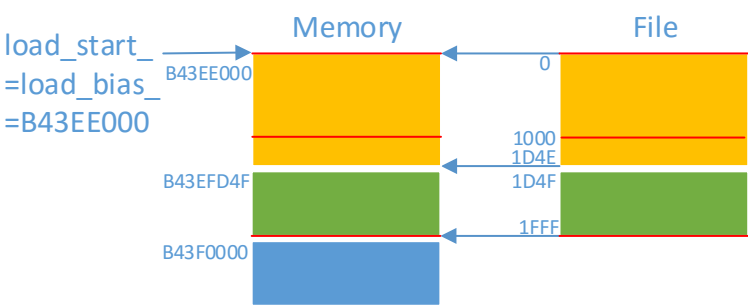


图 4 加载第一个可加载段时内存与文件的映射关系

加载第二个可加载段的相关信息如图 5 所示，此时内存与文件的映射关系如图 6 所示，文件内容[1000, 2010)映射到内存[B43F0000, B43F1010)中，其中可加载段二[1EA8, 2010)在内存中位于[B43F0EA8, B43F1010)。由于该段可写，因而内存[B43F1010, B43F2000)中的内容置 0。

seg_page_start	B43F0000
seg_page_end	B43F2000
file_page_start	1000
file_length	1010
seg_file_end	B43F1010
seg_start	B43F0EA8

图 5 第二个可加载段的加载信息

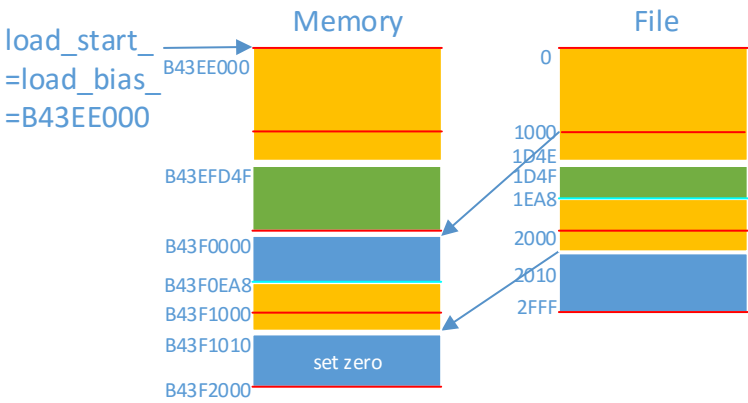


图 6 加载第二个可加载段时内存与文件的映射关系

在加载完 .so 文件后, Load() 继续调用 FindPhdr() 函数检查 program header table 是否已经在内存中了, 即检查可加载段中是否包含 program header table。

FindPhdr()实现如下:

```
728 bool ElfReader::FindPhdr() {
729     const ElfW(Phdr)* phdr_limit = phdr_table_ + phdr_num_;
730
731     //首先检查是否有类型是 PT_PHDR 的段, 即 program header table
732     for (const ElfW(Phdr)* phdr = phdr_table_; phdr < phdr_limit; ++phdr) {
733         if (phdr->p_type == PT_PHDR) {
734             //load_bias_ + phdr->p_vaddr 是 phdr 在内存中的起始地址
735             return CheckPhdr(load_bias_ + phdr->p_vaddr); //检查是否在内存中
736         }
737     }
738     //否则, 检查第一个可加载段。如果它在文件中的偏移是 0, 那么该段以 ELF 头
739     //开始, 我们通过 ELF 头能计算 program header table 的地址
740     for (const ElfW(Phdr)* phdr = phdr_table_; phdr < phdr_limit; ++phdr) {
741         if (phdr->p_type == PT_LOAD) {
742             if (phdr->p_offset == 0) {
743                 ElfW(Addr) elf_addr = load_bias_ + phdr->p_vaddr;
744                 const ElfW(Ehdr)* ehdr = reinterpret_cast<const ElfW(Ehdr)*>(elf_addr);
745                 ElfW(Addr) offset = ehdr->e_phoff; //ehdr->e_phoff 是 pht 在文件中的偏移
746                 return CheckPhdr((ElfW(Addr))ehdr + offset); //检查是否在内存中
747             }
748         }
749         break;
750     }
751 }
752 DL_ERR("can't find loaded phdr for \"%s\"", name_);
753 return false;
754 }
755 }
```

从上可知, FindPhdr()通过两种策略来确定 program header table 是否在内存中, 如上面的注释所示。最终确定 program header table 是否在内存中是通过 CheckPhdr()来实现的。CheckPhdr()的逻辑很简单: 检查 program header table 的地址范围是否包含在被加载的段中, CheckPhdr()的实现如下:

```
760 bool ElfReader::CheckPhdr(ElfW(Addr) loaded) {
761     const ElfW(Phdr)* phdr_limit = phdr_table_ + phdr_num_;
762     //loaded_end 是 pht 在内存中的结束地址
763     ElfW(Addr) loaded_end = loaded + (phdr_num_ * sizeof(ElfW(Phdr)));
764     for (ElfW(Phdr)* phdr = phdr_table_; phdr < phdr_limit; ++phdr) {
765         if (phdr->p_type != PT_LOAD) {
766             continue;
767         }
768         ElfW(Addr) seg_start = phdr->p_vaddr + load_bias_;
769         ElfW(Addr) seg_end = phdr->p_filesz + seg_start;
```

```

769     if (seg_start <= loaded && loaded_end <= seg_end) {
        //遍历每一个可加载段，检查 phdr 的地址范围是否在可加载段中
770         loaded_phdr_ = reinterpret_cast<const ElfW(Phdr)*>(loaded);
771         return true;
772     }
773 }
774 DL_ERR("\'%s\' loaded phdr %p not in loadable segment", name_,
        reinterpret_cast<void*>(loaded));
775 return false;
776}

```

到此，我们分析完了 ElfReader.Load() 是如何加载 .so 文件的可加载段，这里简单总结一下：

1. ReadElfHeader(): 从 .so 文件中读取 ELF 头；
2. VerifyElfHeader(): 校验 ELF 头；
3. ReadProgramHeader(): 将 .so 文件的 program header table 映射到内存；
4. ReserveAddressSpace(): 开辟匿名内存空间；
5. LoadSegments(): 将可加载段加载到 ReserveAddressSpace 开辟的空间中；
6. FindPhdr(): 校验 program header table 是否在内存中。

2.2.3 load_library() 第三部分：解析 dynamic section

load_library() 函数中，833-841 行创建一个 soinfo 对象，并对相关字段进行赋值：

- a. si->base: 加载 so 文件时，mmap 得到的内存空间的首地址。由于 .so 文件中第一个可加载段的偏移通常是 0，从 ReserveAddressSpace() 的实现可知，si->load_bias 等于 si->base，因而 si->base 也是第一个可加载段的起始地址；
- b. si->size: ReserveAddressSpace 中开启的内存空间的大小；
- c. si->load_bias: 加载的偏移地址，对于一个可加载段来说，si->load_bias+p_vaddr 是它在内存中的地址；
- d. si->phnum: program header 的个数；
- e. si->phdr: program header table 在内存中的起始地址。

843 行调用 PrelinkImage() 解析 .so 文件的 dynamic section；848-850 行将该 .so 文件依赖的库添加到待加载的队列中。在分析 PrelinkImage() 之前，我们先看一下 dynamic section 中 entry 的数据结构，如下：

```

1620// Dynamic table entry for ELF32.
1621struct Elf32_Dyn

```

```

1622{
1623    Elf32_Sword d_tag;           // Type of dynamic table entry.
1624    union
1625    {
1626        Elf32_Word d_val;        // Integer value of entry.
1627        Elf32_Addr d_ptr;        // Pointer value of entry.
1628    } d_un;
1629};

```

其中 d_un 值的意义与 d_tag 的取值有关。下面分析 PrelinkImage(), 看它是如何解析 *Elf32_Dyn*, PrelinkImage() 的关键代码如下:

```

1858 bool soinfo::PrelinkImage() {
1859     /* Extract dynamic section */
1860     ElfW(Word) dynamic_flags = 0;
1861     phdr_table_get_dynamic_section(phdr, phnum, load_bias, &dynamic,
    &dynamic_flags); //根据 program header table 找到 dynamic section
1862     ... ...
1881 #if defined(__arm__) //找到 ARM.exidx section 在内存中的地址
1882     (void) phdr_table_get_arm_exidx(phdr, phnum, load_bias,
1883                                     &ARM_exidx, &ARM_exidx_count);
1884 #endif
1885
1886     // Extract useful information from dynamic section.
1887     uint32_t needed_count = 0; //该 .so 依赖库的数目
1888     for (ElfW(Dyn)* d = dynamic; d->d_tag != DT_NULL; ++d) { //遍历 dynamic
1889         ... ...
1891         switch (d->d_tag) {
1892             ... ...
1897             case DT_HASH: //hash 表相关信息
1898                 nbucket = reinterpret_cast<uint32_t*>(load_bias + d->d_un.d_ptr)[0];
1899                 nchain = reinterpret_cast<uint32_t*>(load_bias + d->d_un.d_ptr)[1];
1900                 bucket = reinterpret_cast<uint32_t*>(load_bias + d->d_un.d_ptr + 8);
1901                 chain = reinterpret_cast<uint32_t*>(load_bias + d->d_un.d_ptr + 8 +
    nbucket * 4);
1902                 break;
1903             //字符串表的偏移, 与 .dynstr section 对应, d_un.d_ptr 与 s_addr 相等
1904             case DT_STRTAB:
1905                 strtabs = reinterpret_cast<const char*>(load_bias + d->d_un.d_ptr);
1906                 break;
1907
1908             case DT_STRSZ: //字符串表的大小(字节)

```

```

1909     strtab_size = d->d_un.d_val;
1910     break;
1911
1912     case DT_SYMTAB: //符号表的偏移
1913         symtab = reinterpret_cast<ElfW(Sym)*>(load_bias + d->d_un.d_ptr);
1914         break;
1915
1916     case DT_SYMENT: //符号表项的大小(字节)
1917         if (d->d_un.d_val != sizeof(ElfW(Sym))) {
1918             DL_ERR("invalid DT_SYMENT: %zd",
static_cast<size_t>(d->d_un.d_val));
1919             return false;
1920         }
1921         break;
1922         ... ..
1937     case DT_JMPREL: //与过程链接表相关的重定位表的偏移，与.rel.plt section 对应
1938 #if defined(USE_RELA)
1939         plt_rela = reinterpret_cast<ElfW(Rela)*>(load_bias + d->d_un.d_ptr);
1940 #else
1941         plt_rel = reinterpret_cast<ElfW(Rel)*>(load_bias + d->d_un.d_ptr);
1942 #endif
1943         break;
1944
1945     case DT_PLTRELSZ: //与过程链接表相关的重定位表的大小(字节)
1946 #if defined(USE_RELA)
1947         plt_rela_count = d->d_un.d_val / sizeof(ElfW(Rela));
1948 #else
1949         plt_rel_count = d->d_un.d_val / sizeof(ElfW(Rel));
1950 #endif
1951         break;
1952
1953     case DT_PLTGOT:
1954 #if defined(__mips__)
1955         // Used by mips and mips64.
1956         plt_got = reinterpret_cast<ElfW(Addr)**>(load_bias + d->d_un.d_ptr);
1957 #endif
1958         // Ignore for other platforms... (because RTLD_LAZY is not supported)
1959         break;
1960         ... ..
1974 #if defined(USE_RELA)
1975         ... ..
2001 #else

```

```

2002     case DT_REL: //重定位表的偏移, 与.rel.dyn section 对应
2003         rel = reinterpret_cast<ElfW(Rel)*>(load_bias + d->d_un.d_ptr);
2004         break;
2005
2006     case DT_RELSZ: //重定位表的总大小(字节)
2007         rel_count = d->d_un.d_val / sizeof(ElfW(Rel));
2008         break;
2009
2010     case DT_RELENT: //重定位表项的大小(字节)
2011         if (d->d_un.d_val != sizeof(ElfW(Rel))) {
2012             DL_ERR("invalid DT_RELENT: %zd",
static_cast<size_t>(d->d_un.d_val));
2013             return false;
2014         }
2015         break;
2016         ... ...
2017 #endif
2018
2019     case DT_INIT: //初始化函数 init 的偏移
2020         init_func = reinterpret_cast<linker_function_t>(load_bias +
d->d_un.d_ptr);
2021         DEBUG("%s constructors (DT_INIT) found at %p", name, init_func);
2022         break;
2023
2024     case DT_FINI: //结束函数的偏移
2025         fini_func = reinterpret_cast<linker_function_t>(load_bias +
d->d_un.d_ptr);
2026         DEBUG("%s destructors (DT_FINI) found at %p", name, fini_func);
2027         break;
2028
2029     case DT_INIT_ARRAY: //初始化函数数组 init_array 的偏移
2030         init_array = reinterpret_cast<linker_function_t*>(load_bias +
d->d_un.d_ptr);
2031         DEBUG("%s constructors (DT_INIT_ARRAY) found at %p", name, init_array);
2032         break;
2033
2034     case DT_INIT_ARRAYSZ: // init_array 的大小(字节)
2035         init_array_count = ((unsigned)d->d_un.d_val) / sizeof(ElfW(Addr));
2036         break;
2037
2038     case DT_FINI_ARRAY:
2039         fini_array = reinterpret_cast<linker_function_t*>(load_bias +
d->d_un.d_ptr);

```



```

2049     DEBUG("%s destructors (DT_FINI_ARRAY) found at %p", name, fini_array);
2050     break;
2051
2052     case DT_FINI_ARRAYSZ:
2053         fini_array_count = ((unsigned)d->d_un.d_val) / sizeof(ElfW(Addr));
2054         break;
2055
2056     case DT_PREINIT_ARRAY:
2057         preinit_array = reinterpret_cast<linker_function_t*>(load_bias +
2058         d->d_un.d_ptr);
2059         DEBUG("%s constructors (DT_PREINIT_ARRAY) found at %p", name,
2060         preinit_array);
2061         break;
2062
2063     case DT_PREINIT_ARRAYSZ:
2064         preinit_array_count = ((unsigned)d->d_un.d_val) / sizeof(ElfW(Addr));
2065         break;
2066
2067     case DT_TEXTREL:
2068         #if defined(__LP64__)
2069         DL_ERR("text relocations (DT_TEXTREL) found in 64-bit ELF file \"%s\"",
2070         name);
2071         return false;
2072         #else
2073         has_text_relocations = true;
2074         break;
2075         #endif
2076
2077     case DT_SYMBOLIC:
2078         has_DT_SYMBOLIC = true;
2079         break;
2080
2081     case DT_NEEDED: //d->d_un.d_val 是依赖库名字在字符串表中的索引
2082         ++needed_count;
2083         break;
2084     ... ..
2085 }
2086 }
2087 ... ..
2088 return true;
2089 }

```

1861 行调用 `phdr_table_get_dynamic_section()` 来获取 `dynamic section`

在内存中的地址。`.dynamic` section 对应的段的类型是 `PT_DYNAMIC`，且类型为 `PT_DYNAMIC` 的段中只有 `.dynamic` 一个 section。因而只需要遍历 `program header table`，找到类型为 `PT_DYNAMIC` 的段即可，其在内存中的地址就是 `load_bias+p_vaddr`。这里就不贴 `phdr_table_get_dynamic_section()` 的代码了。

与 `phdr_table_get_dynamic_section()` 类似，1881-1884 行调用 `phdr_table_get_arm_exidx()` 来获取 `.ARM.exidx` section 在内存中的地址，类型为 `PT_ARM_EXIDX` 的段中只有 `.ARM.exidx` 一个 section。

1888-2150 行是个 `for` 循环，遍历 `.dynamic` section 中每一条 entry，根据 `d_tag` 的值，用 `d_un` 做相应的操作。我们这里选择几个来讲解：

1. 1897-1902 行，如果 `d_tag` 为 `DT_HASH` (4)，那么 `d->d_un.d_ptr` 是 `.hash` section (哈希表) 的偏移，`load_bias + d->d_un.d_ptr` 是哈希表在内存中的起始地址。哈希表的结构如图 7 所示。哈希表用于快速访问符号表，其中 `bucket` 数组有 `nbucket` 个元素，`chain` 数组有 `nchain` 个元素。`bucket` 数组和 `chain` 数组中都保存着符号表的索引。`chain[i]` 与 `symbolTable[i]` 对应，`nchain` 等于符号表的项数。哈希函数的输入是符号名，返回一个哈希值 `X`，`index = bucket[X % nbucket]` 是 `chain` 和符号表的索引，如果 `symbolTable[index]` 不是要找的符号，那么 `chain[index]` 的值是具有相同哈希值的下一个符号的索引，这样我们可以沿着 `chain` 找到所需要的符号，直到 `chain[index] = 0`。

<code>nbucket</code>	4bytes
<code>nchain</code>	4bytes
<code>bucket[0]</code>	4bytes
... ..	
<code>bucket[nbucket-1]</code>	4bytes
<code>chain[0]</code>	4bytes
... ..	
<code>chain[nchain-1]</code>	4bytes

图 7 哈希表结构图

2. 1904-1906 行，如果 `d_tag` 为 `DT_STRTAB` (5)，那么 `d->d_un.d_ptr` 是字符串表的偏移，`load_bias + d->d_un.d_ptr` 是字符串表在内存中的地址，图 8 是一个 `.so` 文件的 `.dynamic` section，红色框出的是 `d_tag` 为 `DT_STRTAB` 的一条 entry，从图中可以知道，字符串表的偏移是 `0x04E8`。该 `.so` 文件的 `.dynstr` section header 如图 9 所示，可知 `d->d_un.d_ptr` 与 `s_offset` 是相同的。

1EB0h:	00 00 00 00	03 00 00 00	D0 2F 00 00	02 00 00 00/.....
1EC0h:	48 00 00 00	17 00 00 00	94 0B 00 00	14 00 00 00	H.....".
1ED0h:	11 00 00 00	11 00 00 00	3C 0B 00 00	12 00 00 00<.....
1EE0h:	58 00 00 00	13 00 00 00	08 00 00 00	FA FF FF 6F	X.....úÿÿo
1EF0h:	09 00 00 00	06 00 00 00	DT STRTAB	01 00 00 00	DT STRSZ
1F00h:	10 00 00 00	05 00 00 00	E8 04 00 00	0A 00 00 00è.....
1F10h:	CE 04 00 00	04 00 00 00	B8 09 00 00	01 00 00 00	ï.....
1F20h:	94 04 00 00	01 DT_NEEDED	04 00 00 00	01 00 00 00	".....ž.....
1F30h:	AB 04 00 00	01 00 00 00	B3 04 00 00	01 00 00 00	«.....'.....
1F40h:	BB 04 00 00	0E 00 00 00	C4 04 00 00	1A 00 00 00	».....Ä.....
1F50h:	A8 2E 00 00	1C 00 00 00	08 00 00 00	19 00 00 00
1F60h:	B0 2E 00 00	1B 00 00 00	04 00 00 00	10 00 00 00	o.....
1F70h:	00 00 00 00	1E 00 00 00	0A 00 00 00	FB FF FF 6Fúÿÿo
1F80h:	01 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
1F90h:	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
1FA0h:	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
1FB0h:	00 00 00 00	85 14 00 00	89 14 00 00	8D 14 00 00%.....

图 8 .dynamic section 实例

struct section_table_entry32_t section_table_element[3]	.dynstr
▷ struct s_name32_t s_name	.dynstr
enum s_type32_e s_type	SHT_STRTAB (3h)
enum s_flags32_e s_flags	SF32_Alloc (2h)
Elf32_Addr s_addr	0x000004E8
Elf32_Off s_offset	4E8h
Elf32_Xword s_size	4CEh
Elf32_Word s_link	0h
Elf32_Word s_info	0h
Elf32_Xword s_addralign	1h
Elf32_Xword s_entsize	0h
▷ char s_data[1230]	

图 9 .dynstr section header

- 1904-1906 行，如果 d_tag 为 DT_STRSZ (10)，那么 d->d_un.d_val 是字符串表的大小，如图 8，黄色框出的是 d_tag 为 DT_STRSZ 的一条 entry，可知字符串表的大小为 04CE，与图 9 中的 s_size 相等。
- 与字符串表类似，1912-1921 行与 .dynsym section（即符号表）相关，如果 d_tag 为 DT_SYMTAB(6)，d->d_un.d_ptr 是符号表的偏移，load_bias + d->d_un.d_ptr 是符号表在内存中的地址。如果 d_tag 为 DT_SYMENT(11)，d->d_un.d_val 是符号表每一条 entry 的大小，必须为 sizeof(ElfW(Sym))。
- 2078-2080 行，如果 d_tag 为 DT_NEEDED (1)，那么 d->d_un.d_val 是字符串表的索引，其指向的值是该 .so 文件的一个依赖库的名字。如图 8，黑色框出的是 d_tag 为 DT_NEEDED 的一条 entry，可知其中一个依赖库的名字在字符串表中的索引是 0x04B3，由图 9 可知，字符串表的偏移是 0x04E8，因而该依赖库的名字在文件中的地址为 0x04B3+0x04E8=0x099B，如图 10 所示，可知该依赖库为 libc.so。

0980h:	6F 67 2E 73	6F 00 6C 69	62 73 74 64	63 2B 2B 2E	og.so.libstdc++.
0990h:	73 6F 00 6C	69 62 6D 2E	73 6F 00 6C	69 62 63 2E	so.libm.so.libc.
09A0h:	73 6F 00 6C	69 62 64 6C	2E 73 6F 00	6C 69 62 61	so.libdl.so.liba
09B0h:	70 70 2E 73	6F 00 00 00	25 00 00 00	3A 00 00 00	pp.so...%.....

图 10 字符串表

至此，我们分析完了 `load_library()` 函数，也就是完成了 `find_libraries()` 第二部分的分析，知道了 linker 是如何将 .so 文件加载到内存，并对 dynamic section 进行解析处理。

2.3 find_libraries() 第三部分：链接阶段

这部分对加载的 so 进行链接操作，链接的顺序与加载的顺序刚好相反。955 行调用 `LinkImage()` 进行链接，其关键代码如下：

```

2175 bool soinfo::LinkImage(const android_dlextinfo* extinfo) {
2177     if (!defined(__LP64__))
2178     {
2179         if (has_text_relocations) {
2180             DL_WARN("%s has text relocations. This is wasting memory and prevents "
2181                 "security hardening. Please fix.", name);
2182             //使段可读写，通过系统调用 mprotect() 来设置
2183             if (phdr_table_unprotect_segments(phdr, phnum, load_bias) < 0) {
2184                 DL_ERR("can't unprotect loadable segments for \"%s\": %s",
2185                     name, strerror(errno));
2186                 return false;
2187             }
2188         }
2189     }
2190     if defined(USE_RELA)
2191     ...
2204 else
2205     if (rel != nullptr) {
2206         DEBUG("[ relocating %s ]", name);
2207         if (Relocate(rel, rel_count)) { //对重定位表中所指的符号进行重定位
2208             return false;
2209         }
2210     }
2211     if (plt_rel != nullptr) { //与调用导入函数相关
2212         DEBUG("[ relocating %s plt ]", name);
2213         if (Relocate(plt_rel, plt_rel_count)) { //对重定位表中所指的符号进行重定位
2214             return false;
2215         }
2216     }
2217 }

```

```

2218 ... ...
2225 DEBUG("[ finished linking %s ]", name);
2226
2227 #if !defined(__LP64__)
2228     if (has_text_relocations) {
2229         // All relocations are done, we can protect our segments back to read-only.
2230         if (phdr_table_protect_segments(phdr, phnum, load_bias) < 0) {
2231             DL_ERR("can't protect segments for \"%s\": %s",
2232                 name, strerror(errno));
2233             return false;
2234         }
2235     }
2236 #endif
2237
2238 /* We can also turn on GNU RELRO protection */
2239 if (phdr_table_protect_gnu_relro(phdr, phnum, load_bias) < 0) {
2240     DL_ERR("can't enable GNU RELRO protection for \"%s\": %s",
2241         name, strerror(errno));
2242     return false;
2243 }
2244 ... ...
2263 return true;
2264 }

```

从 2207 行和 2213 行可知，对 .rel.dyn 和 .rel.plt 两个重定位表都是调用 Relocate() 来进行重定位的。在分析 Relocate() 之前，先看一下重定位表项的数据结构，如下：

```

164 typedef struct elf32_rel {
165     Elf32_Addr r_offset; //在文件中的偏移或者虚拟地址
166     Elf32_Word r_info; //符号表的索引和重定位类型
167 } Elf32_Rel;
159 #define ELF32_R_SYM(x) ((x) >> 8)
160 #define ELF32_R_TYPE(x) ((x) & 0xff)

```

其中符号表的索引是 r_info 的高 24 位，可通过宏 ELF32_R_SYM 获取，表示重定位的符号；重定位类型是 r_info 的低 8 位，可通过宏 ELF32_R_TYPE 获取，r_offset 根据重定位类型的不同有不同的解释。

知道了重定位表项的结构，下面看 Relocate() 是如何对重定位表中的每一项进行重定位的，其关键代码如下：

```

1359 int soinfo::Relocate(ElfW(Rel)* rel, unsigned count) {
1360     for (size_t idx = 0; idx < count; ++idx, ++rel) { //遍历重定位表
1361         unsigned type = ELF32_R_TYPE(rel->r_info); //重定位类型

```

```

1363 unsigned sym = ELFW(R_SYM)(rel->r_info); //符号表索引
//重定位的地址，即 reloc 处的值需要重新计算
1364 ElfW(Addr) reloc = static_cast<ElfW(Addr)>(rel->r_offset + load_bias);
1365 ElfW(Addr) sym_addr = 0; //符号的地址
1366 const char* sym_name = nullptr; //符号的名称
1367
1368 DEBUG("Processing '%s' relocation at index %zd", name, idx);
1369 if (type == 0) { // R_*_NONE
1370     continue;
1371 }
1372
1373 ElfW(Sym)* s = nullptr; //该符号在其定义 so 中的记录
1374 soinfo* lsi = nullptr; //定义该符号的 so
1375
1376 if (sym != 0) {
1377     sym_name = get_string(symtab[sym].st_name); //得到符号的名称
1378     s = soinfo_do_lookup(this, sym_name, &lsi); //查找 sym_name 定义在哪个 so
1379     if (s == nullptr) { //如果该符号没有定义，那么它的绑定类型必须是弱引用
1380         // We only allow an undefined symbol if this is a weak reference...
1381         s = &symtab[sym];
1382         if (ELF_ST_BIND(s->st_info) != STB_WEAK) {
1383             DL_ERR("cannot locate symbol \"%s\" referenced by \"%s\"...",
sym_name, name);
1384             return -1;
1385         }
1400         switch (type) {
//没有定义的弱引用，它的 sym_addr 是 0，或者重定位的时候不关心 sym_addr 的值
1401 #if defined(__arm__)
1402         case R_ARM_JUMP_SLOT:
1403         case R_ARM_GLOB_DAT:
1404         case R_ARM_ABS32:
1405         case R_ARM_RELATIVE: /* Don't care. */
1406             // sym_addr was initialized to be zero above or relocation
1407             // code below does not care about value of sym_addr.
1408             // No need to do anything.
1409             break;
1411             ... ..
1423 #endif
1424             ... ..
1432         } //end switch
1433     } else { //找到了符号的定义 so，计算该符号的地址
1435         sym_addr = lsi->resolve_symbol_address(s);

```

```

1436     }
1437     count_relocation(kRelocSymbol);
1438 } //end if (sym != 0)
1439
1440     switch (type) { //根据重定位类型修改 reloc 处的值
1441 #if defined(__arm__)
1442         case R_ARM_JUMP_SLOT:
1443             ... ..
1444             *reinterpret_cast<ElfW(Addr)*>(reloc) = sym_addr;
1445             break;
1446         case R_ARM_GLOB_DAT:
1447             ... ..
1448             *reinterpret_cast<ElfW(Addr)*>(reloc) = sym_addr;
1449             break;
1450         case R_ARM_ABS32:
1451             ... ..
1452             *reinterpret_cast<ElfW(Addr)*>(reloc) += sym_addr;
1453             break;
1454         case R_ARM_REL32:
1455             ... ..
1456             *reinterpret_cast<ElfW(Addr)*>(reloc) += sym_addr - rel->r_offset;
1457             break;
1458         case R_ARM_COPY:
1459             DL_ERR("%s R_ARM_COPY relocations are not supported", name);
1460             return -1;
1461         ... ..
1462 #endif
1463
1464 #if defined(__arm__)
1465         case R_ARM_RELATIVE:
1466 #elif defined(__i386__)
1467         case R_386_RELATIVE:
1468 #endif
1469
1470     count_relocation(kRelocRelative);
1471     MARK(rel->r_offset);
1472     if (sym) {
1473         DL_ERR("odd RELATIVE form...");
1474         return -1;
1475     }
1476     TRACE_TYPE(RELO, "RELO RELATIVE %p <- +%p",
1477         reinterpret_cast<void*>(reloc),
1478         reinterpret_cast<void*>(base));

```

```

1543     *reinterpret_cast<ElfW(Addr)*>(reloc) += base;
1544     break;
1546     ... ...
1554     default:
1555         DL_ERR("unknown reloc type %d @ %p (%zu)", type, rel, idx);
1556         return -1;
1557 } //end switch
1558 } //end for
1559 return 0;
1560}

```

1364 行变量 `reloc` 是重定位的地址，即 `reloc` 处的值需要重新计算，对于导入函数来说，地址 `reloc` 在 `got` 表中，`reloc` 处应该是函数的实际地址，代码中函数的地址其实是其在 `got` 表中的偏移，再从 `got` 表中跳转到函数的实际地址。

1378 行调用 `soinfo_do_lookup()` 查找符号的定义 `so`。 `soinfo_do_lookup()` 的关键代码如下：

```

482 static ElfW(Sym)* soinfo_do_lookup(soinfo* si, const char* name, soinfo** lsi)
{
483     unsigned elf_hash = elfhash(name); //计算符号的哈希值
484     ElfW(Sym)* s = nullptr;
497     if (si->has_DT_SYMBOLIC) {
498         DEBUG("%s: looking up %s in local scope (DT_SYMBOLIC)", si->name, name);
499         s = soinfo_elf_lookup(si, elf_hash, name);
500         if (s != nullptr) {
501             *lsi = si;
502         }
503     }
504     ... ...
516     // 2. Look for it in the ld_preloads
517     if (s == nullptr) {
518         for (int i = 0; g_ld_preloads[i] != NULL; i++) {
519             s = soinfo_elf_lookup(g_ld_preloads[i], elf_hash, name);
520             if (s != nullptr) {
521                 *lsi = g_ld_preloads[i];
522                 break;
523             }
524         }
525     }
526 }
527
538 if (s == nullptr && !si->has_DT_SYMBOLIC) {

```



```

539     DEBUG("%s: looking up %s in local scope", si->name, name);
540     s = soinfo_elf_lookup(si, elf_hash, name);
541     if (s != nullptr) {
542         *lsi = si;
543     }
544 }
545
546 if (s == nullptr) { //在其依赖库(子结点)中递归查找符号
547     si->get_children().visit([&](soinfo* child) {
548         DEBUG("%s: looking up %s in %s", si->name, name, child->name);
549         s = soinfo_elf_lookup(child, elf_hash, name);
550         if (s != nullptr) {
551             *lsi = child;
552             return false;
553         }
554         return true;
555     });
556 }
557 ... ..
566 return s;
567}

```

从上可知，soinfo_do_look() 分别在其自身、预加载库和依赖库中查找符号的定义，具体的查找函数是 soinfo_elf_lookup()，其关键代码如下：

```

418 static ElfW(Sym)* soinfo_elf_lookup(soinfo* si, unsigned hash, const char*
name) {
419     ElfW(Sym)* symtab = si->symtab; //符号表
420     ... ..
423     //通过哈希表在符号表中快速查找 name
424     for (unsigned n = si->bucket[hash % si->nbucket]; n != 0; n = si->chain[n]) {
425         ElfW(Sym)* s = symtab + n;
426         if (strcmp(si->get_string(s->st_name), name)) continue; //符号名字需相同
427
428         // only concern ourselves with global and weak symbol definitions
429         switch (ELF_ST_BIND(s->st_info)) {
430             case STB_GLOBAL:
431             case STB_WEAK:
432                 if (s->st_shndx == SHN_UNDEF) { //符号未定义
433                     continue;
434                 }
435
436                 TRACE_TYPE(LOOKUP, "FOUND %s in %s (%p) %zd",
437                     name, si->name, reinterpret_cast<void*>(s->st_value),

```

```

438         static_cast<size_t>(s->st_size));
439     return s; //在 si 中找到符号的定义
440 case STB_LOCAL:
441     continue;
442 default:
443     __libc_fatal("ERROR: Unexpected ST_BIND value: %d for '%s' in '%s'",
444                 ELF_ST_BIND(s->st_info), name, si->name);
445 }
446 }
447 ... ..
452 return nullptr;
453}

```

从上可知，soinfo_elf_lookup() 用于确定符号是否在 si 中定义并且符号的绑定类型是否为 STB_GLOBAL 或 STB_WEAK (430-439 行)。该函数通过哈希表来快速定位符号，大家可以参看前面分析 PrelinkImage() 时对哈希表的解释，以便理解查找过程。

分析完了符号的查找过程，我们再回到 Relocate() 函数。在找到符号的定义后，1435 行调用 resolve_symbol_address() 来计算符号的地址。resolve_symbol_address() 的实现如下：

```

1781 ElfW(Addr) soinfo::resolve_symbol_address(ElfW(Sym)* s) {
1782     if (ELF_ST_TYPE(s->st_info) == STT_GNU_IFUNC) { //符号的类型是 gnu indirect
func
1783         return call_ifunc_resolver(s->st_value + load_bias);
1784     }
1786     return static_cast<ElfW(Addr)>(s->st_value + load_bias);
1787}

```

从上可知，如果符号的类型不是 STT_GNU_IFUNC (GNU indirect function)，如 STT_FUNC (可执行代码，如函数)、STT_OBJECT (数据对象，如变量) 等，直接返回符号的地址，即 s->st_value + load_bias，否则调用 call_ifunc_resolver() 计算符号的地址（关于 GNU indirect function，简单来说就是根据函数名字字符串来调用函数，感觉和反射类似，如果想进一步了解，还请自行 google）。call_ifunc_resolver() 的实现如下：

```

1072 static ElfW(Addr) call_ifunc_resolver(ElfW(Addr) resolver_addr) {
1073     typedef ElfW(Addr) (*ifunc_resolver_t)(void);
1074     ifunc_resolver_t ifunc_resolver =
reinterpret_cast<ifunc_resolver_t>(resolver_addr); //将 resolver_addr 转为函数指针
1075     ElfW(Addr) ifunc_addr = ifunc_resolver(); //执行 resoler_addr 处的函数

```

```

1076 TRACE_TYPE(RELO, "Called ifunc_resolver@%p. The result is %p", ifunc_resolver,
reinterpret_cast<void*>(ifunc_addr));
1078 return ifunc_addr;
1079}

```

从上可知，resolver_addr 其实是一个函数的地址，在 1075 行执行这个函数，其返回值就是符号的地址。

在得到符号的地址(sym_addr)后, Relocate() 函数中 1440-1557 行根据符号的重定位类型重新计算 reloc 处的值。重定位类型和 reloc 处值的计算方式对应关系如表 1 所示。注意重定位类型为 R_ARM_RELATIVE 的重定位项，其 r_info 中符号表的索引必须为 0，即不需要搜索符号，其重定位值的计算也与 sym_addr 没有关系。

表 1 重定位类型与重定位值的计算方式对应表

重定位类型	reloc 处的值
R_ARM_JUMP_SLOT	*reloc = sym_addr
R_ARM_GLOB_DAT	*reloc = sym_addr
R_ARM_ABS32	*reloc += sym_addr
R_ARM_REL32	*reloc += sym_addr -rel->r_offset
R_ARM_RELATIVE	*reloc += base

至此，find_libraries() 的第三部分分析完了，这里对链接过程总结一下：遍历重定位表，根据重定位项的 r_info 获得重定位类型和重定位项对应的符号在符号表中的索引；然后利用 so 中的 hash 表，根据符号名快速地查找符号在哪个 so 中定义；当找到了符号的定义，计算符号的地址 sym_addr；最后根据符号的重定位类型，结合 sym_addr 计算重定位值。

so 文件加载到内存，并链接完成后，就开始调用 so 中的初始化函数。这里回到 do_dlopen() 继续分析。为方便大家阅读，这里重复帖一下 do_dlopen() 的关键代码，如下：

```

1041 soinfo* do_dlopen(const char* name, int flags, const android_dlexthinfo*
extinfo) { //extinfo 为 null
1042 ... ..
1057 protect_data(PROT_READ | PROT_WRITE);
1058 soinfo* si = find_library(name, flags, extinfo); //加载链接 name
1059 if (si != nullptr) {
1060     si->CallConstructors(); //初始化
1061 }

```

```
1062 protect_data(PROT_READ);
1063 return si;
1064}
```

1060 行调用 `CallConstructors()` 进行初始化操作。`CallConstructors()` 关键代码如下：

```
1656 void soinfo::CallConstructors() {
1657     if (constructors_called) {
1658         return;
1659     }
1660     ... ..
1679     get_children().for_each([] (soinfo* si) {
1680         si->CallConstructors();
1681     });
1682     ... ..
1685     // DT_INIT should be called before DT_INIT_ARRAY if both are present.
1686     CallFunction("DT_INIT", init_func); //调用 init_func 函数
        //调用 init_array 数组中的函数
1687     CallArray("DT_INIT_ARRAY", init_array, init_array_count, false);
1688 }
```

`CallConstructors()` 主要是执行了两段初始化代码：`init_func` 和 `init_array`，这两个变量是在 `PrelinkImage()` 中解析 `dynamic section` 时赋值的。通常加壳逻辑就放在 `init_func` 或 `init_array` 中，它们先于 `jni_onLoad` 执行。

至此，完成了 `so` 的加载分析，希望能对大家有所帮助，分析不到位的地方，还请见谅。

3 附录

表 2 相关函数对应源码路径表

类名	函数名	源码路径
System	loadLibrary	/libcore/luni/src/main/java/java/lang/System.java
Runtime	loadLibrary	/libcore/luni/src/main/java/java/lang/Runtime.java
	doLoad	
-	Runtime_nativeLoad	/art/runtime/native/java_lang_Runtime.cc
JavaVMExt	LoadNativeLibrary	/art/runtime/jni_internal.cc
-	dlopen	/bionic/linker/dlfcn.cpp
-	dlopen_ext	
-	do_dlopen	/bionic/linker/linker.cpp
-	find_library	
-	find_libraries	
-	find_library_internal	
-	load_library	
soinfo	PrelinkImage	
	LinkImage	
	Relocate	
	resolve_symbol_address	
	CallConstructors	
-	soinfo_do_lookup	
-	soinfo_elf_lookup	
-	call_ifunc_resolver	
ElfReader	Load	/bionic/linker/linker_phdr.cpp
	ReadElfHeader	
	VerifyElfHeader	
	ReadProgramHeader	
	ReserveAddressSpace	
	LoadSegments	
	FindPhdr	
	CheckPhdr	

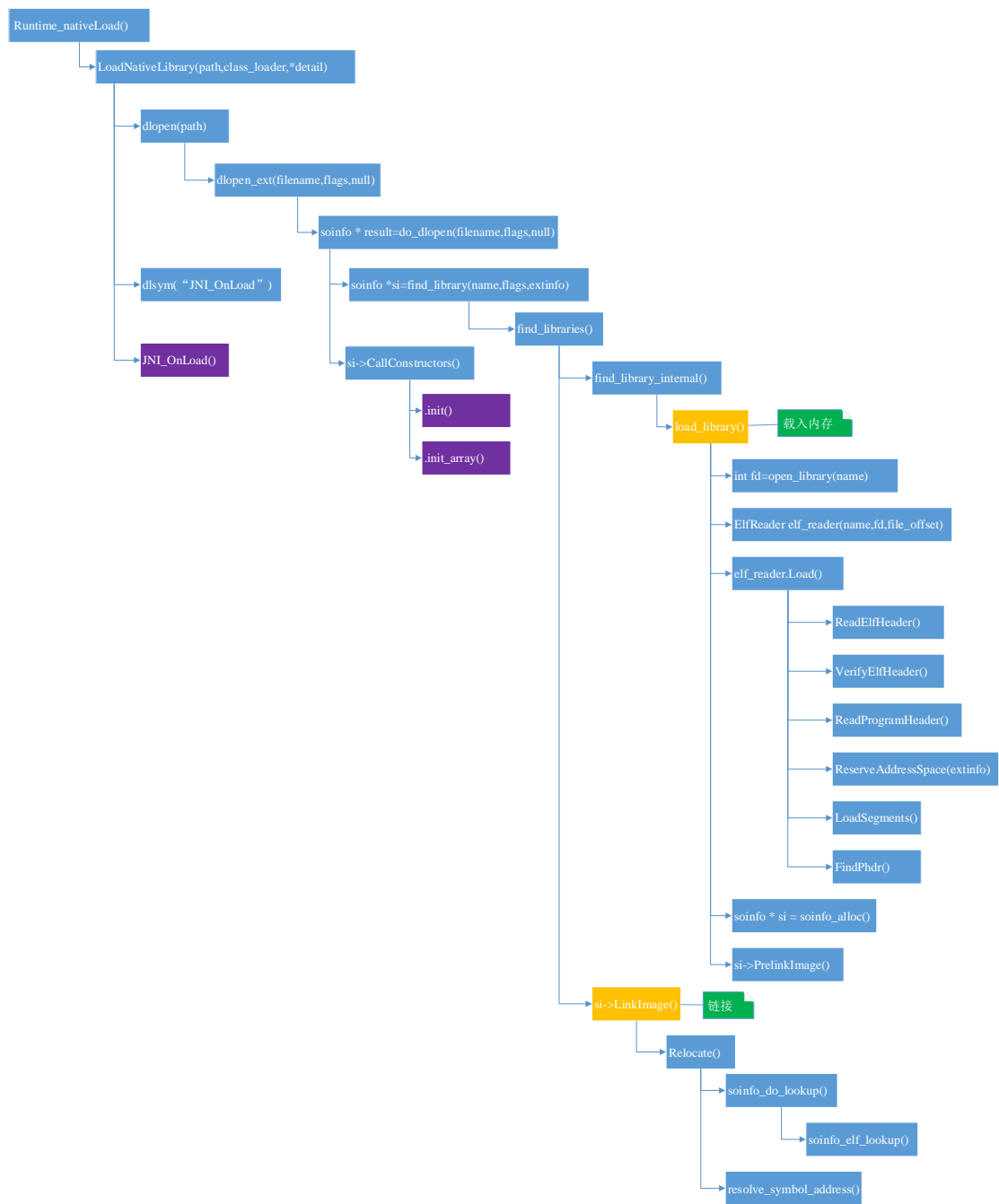


图 11 native 层函数调用关系图