第六章 功能定制

在上一章中，我们了解到系统内置的过程与Android系统的编译流程密切相关。而本章的主题是定制功能，这与安卓源码的执行有着紧密的联系。通过理解源码运行过程，在执行过程中添加需求功能并插入自己的业务逻辑，比如对其进行插桩输出，可以帮助我们更好地理解源码的执行过程。

在本章中，我们将首先开始分析所需实现功能，并深入研究其原理。然后逐步实现这些功能。

6.1 如何进行功能定制

在开始实际操作之前，我们必须明确需求，并规划所要实现功能的具体表现。根据预定的目标方向，我们需要将可提取的业务部分与源码隔离开来。通过先开发一个普通的App来实现业务逻辑，而不是直接在AOSP中修改源码。这样做可以避免在排查细节时反复编译和耗费大量时间成本。

此外，尽可能使用源码版本管理工具来维护代码，以避免长期迭代后无法找到自己修改过的相关代码导致维护困难以及代码迁移不便利。如果无法搭建AOSP作为源码版本管理工具，则应保持一致的代码开发风格，并对自己所做修改处统一打上标识，在功能达到一定阶段时进行备份，以防因修改导致系统异常但又无法回退代码解决问题。

在进行功能定制时，首先需要对目标执行流程和实现过程有一定了解，并找到合适的切入点。在分析源码时，请注意AOSP中提供的各种常用功能函数。如果AOSP已经提供了类似功能的实现方式，则直接参考官方实现即可，不必重新编写。

事实上，在功能开发过程中就是不断熟悉源码和理解源码的过程。接下来，在本章中我们将完成以下几个目标：

- 学习插桩技术，加深对源码执行过程的印象。
- 模仿AOSP自身的系统服务，添加一个自己的系统服务。
- 在应用启动过程中注入Java代码。
- 修改默认权限，了解Android是如何加载解析AndroidManifest.xml文件。

6.2 插桩

在Android逆向中，插桩是一种非常常见的技术手段，它能够帮助开发人员检测和诊断代码问题。插桩指的是在程序运行时向代码中插入额外的指令或代码段，以收集与程序执行相关的信息。这些信息可以用于分析程序的执行流程、性能瓶颈等问题。常见的App插桩方式包括静态插桩和动态插桩两种。

静态插桩是指将额外的指令或代码段直接嵌入到源码中，并通过编译生成修改后的可执行文件。这种方式需要重新编译源码，并且对于已经发布的应用来说不太实际。

动态插桩则是在应用运行时通过注入代码来实现，在不改变原始源码结构和重新编译应用的情况下进行操作。这样做既方便又灵活，可以针对具体需求选择合适位置进行插入。

无论是静态还是动态插桩，它们都为开发人员提供了强大而有力的工具来深入理解和调试复杂程序。在进行Android逆向工程时，掌握并善于使用这些技术将会极大地提高我们解决问题和优化代码质量的能力。

除了App级别的插桩，对于系统来说，还有一种ROM级别的插桩，这种可以算作源码级别的插桩技术。

6.2.1 静态插桩

安卓App的静态插桩通常是指smali反编译文件的静态插桩。这种技术是指在应用程序的dex文件中直接修改smali代码，以实现对应用程序行为的改变或扩展。Smali是一种类似于Java字节码的低级语言，它是Android平台上Dalvik虚拟机所使用的指令集。

通过进行smali静态插桩，我们可以在目标应用程序中添加新的方法、修改现有方法体、注入特定逻辑等操作。这样做能够提供更大程度的控制，并且不需要重新编译整个应用。

下面给出一个简单例子来说明smali静态插桩：

假设我们有一个目标应用程序，其中存在一个名为calculateSum()的方法，该方法接受两个参数并返回它们之和。我们想要在调用calculateSum()前后打印日志以便跟踪其执行。

首先，在目标应用程序中找到相应类文件对应的smali文件（通常位于/smali/com/example/YourClass.smali）。

然后，在YourClass.smali文件中找到包含calculateSum()方法定义部分，并在其前后添加以下代码：

```
.method public calculateSum(II)I
    .locals 2

    ; 在调用calculateSum()之前打印日志
    const-string v0, "Before calculateSum()"
    invoke-static {v0}, Landroid/util/Log;->d(Ljava/lang/String;)I

    ; ... 其他原始的calculateSum()方法代码 ...

    ; 在调用calculateSum()之后打印日志
    const-string v0, "After calculateSum()"
    invoke-static {v0}, Landroid/util/Log;->d(Ljava/lang/String;)I

    ; ... 其他原始的calculateSum()方法代码 ...

.end method
```

以上代码在calculateSum()方法前后分别添加了打印日志的逻辑。这样，在每次调用calculateSum()时，我们都可以看到相应的日志信息。

需要注意的是，进行smali静态插桩需要对Dalvik虚拟机指令集和smali语法有一定了解，并且需要小心操作以避免引入错误或破坏应用程序结构。因此，在实际使用中，建议参考相关文档和教程，并谨慎处理目标应用程序的smali文件。如果读者对这一块内容感兴趣，可以参考阅读笔者另外一本安卓软件逆向工程相关的书籍。

6.2.2 动态插桩

安卓App的动态插桩是指在应用程序运行时通过注入代码来修改或扩展应用程序的行为。与静态插桩不同，动态插桩不需要对源码进行修改或重新编译，而是在应用程序加载和执行过程中实时注入代码。

Frida是一种常用的动态插桩工具，它可以帮助我们在Android设备上进行运行时的代码注入和修改。下面给出一个简单例子来说明Frida动态插桩：

首先，在你的Android设备上安装好Frida，并确保设备与计算机处于相同网络环境。

然后，创建一个Python脚本（例如frida_script.py），并使用以下代码示例：

```
import frida

# 定义要注入的JavaScript代码
js_code = """
Java.perform(function() {
    // 找到目标类及方法
    var targetClass = Java.use('com.example.YourClass');
    var targetMethod = targetClass.calculateSum;

    // 将目标方法替换为新逻辑
    targetMethod.implementation = function(a, b) {
        console.log('Before calculateSum()');

        var result = this.calculateSum(a, b);  // 调用原始方法

        console.log('After calculateSum(), Result: ' + result);

        return result;
    };
});
"""

# 连接到目标进程，并将JavaScript代码注入
def on_message(message, data):
     print(message)

process_name = "com.example.yourapp" # 替换为目标应用程序的进程名
session = frida.get_usb_device().attach(process_name)
script = session.create_script(js_code)
script.on('message', on_message)  # 设置消息监听器
script.load()  # 加载并执行注入的代码

# 持续运行，直到手动中断脚本
frida.resume(pid)
```

以上代码使用Frida在运行时动态插桩了一个名为calculateSum()的方法。它首先找到目标类和方法，然后将原始方法替换为新逻辑，在调用前后打印日志。

你需要将示例代码中的com.example.YourClass和com.example.yourapp替换为实际目标类和应用程序的名称。另外，请确保你已经安装了必要的Python依赖项（如frida、frida-tools等）。

通过执行上述Python脚本，Frida会自动连接到指定进程，并在运行时进行动态插桩操作。当你启动或触发相关操作时，可以在控制台或日志输出中看到相应的信息。

需要注意的是，使用Frida进行动态插桩可能涉及一些安全风险，并且对于某些防护机制可能无法正常工作。因此，在进行实际应用程序测试之前，请确保遵循合法和道德准则，并仔细研究相关文档和教程。

6.2.3 ROM插桩

安卓的ROM插桩是指在Android操作系统的固件（ROM）级别上，通过修改系统代码来实现功能扩展或行为修改。与应用程序层面的动态插桩不同，ROM插桩涉及对底层系统组件和服务进行修改，以实现更广泛、更深入的影响。

由于ROM插桩需要直接修改Android操作系统的代码，因此它通常需要具备特定技术知识和足够权限才能进行。一些常见的用例包括：

- 修改设备启动流程。
- 动态调整CPU频率和性能参数。
- 添加自定义模块或驱动。
- 实施反编译保护机制。
- 实现App代码方法与参数跟踪。

以下是一个简单示例来说明如何在安卓ROM中进行代码插桩：

1. 首先，在你的计算机上设置好Android开发环境，并获取到目标设备所使用的ROM源码。
2. 找到要进行插桩操作的源码文件，在适当位置添加你想要注入执行逻辑或修改原有逻辑。

   例如，在`frameworks/base/core/java/android/widget/Button.java`文件中找到`Button`类，并在其中添加以下代码：

   ```
   // 插入前置逻辑
   Log.d("Button", "Before onClick()");

   // 调用原始方法
   super.onClick(v);

   // 插入后续逻辑
   Log.d("Button", "After onClick()");
   ```

3. 构建并编译ROM，确保修改后的代码被正确集成到系统中。
4. 将编译好的新ROM安装到目标设备上，并验证插桩逻辑是否按预期生效。

在以上示例中，我们向`Button`类的`onClick()`方法添加了前置和后续逻辑。每当按钮被点击时，在日志输出中将显示相应的信息。

需要注意的是，ROM插桩操作属于底层系统级别，因此如果不了解相关技术或没有足够权限进行操作，则可能会导致设备无法正常工作甚至变砖。因此，在进行ROM插桩之前，请务必谨慎行事，并确保遵循官方文档、参考其他资源以及与经验丰富的开发者交流。

6.3 监控Native方法注册

`Native`函数是指在`Android`开发中，`Java`代码调用的由`C、C++`编写的函数。`Native`函数通常用来访问底层系统资源，或进行高性能的计算操作。和普通`Java`函数不一样，`Native`函数需要通过`JNI (Java Native Interface)`进行调用。而`Native`函数能被调用到的前提是需要先进行注册，有两种方式进行注册，分别是静态注册和动态注册。

静态注册是指在编译时就将Native函数与Java方法进行绑定。这种方式需要手动编写C/C++代码，并在编译时生成一个共享库文件，然后在Java程序中加载该库文件并通过JNI接口调用其中的函数。

动态注册是指在程序运行时将Native函数与Java方法进行绑定。这种方式可以在Java程序中动态地加载Native函数，避免了在编译时生成共享库文件的过程。通过JNI接口提供的相关函数，可以在Java程序中实现动态注册的功能。

下面开始了解两种注册方式的实现原理，最终在系统执行过程中找到一个共同调用处进行插桩，将所有App的静态注册和动态注册进行输出，打印出进行注册的目标函数名，以及注册对应的C++函数的偏移地址。

6.3.1 静态注册

通过前文的介绍，了解到Native函数必须要进行注册才能被找到并调用，接下来看两个例子，展示了如何对Native函数进行静态注册和动态注册的。

当使用Android Studio创建一个Native C++的项目，其中默认使用的就是静态注册，在这个例子中，Java函数与C++函数的绑定是通过Java和C++函数名的约定来实现的。具体地说，在Java代码中声明的native方法的命名规则为：Java_+全限定类名+_+方法名，将所有的点分隔符替换为下划线。例如，在这个例子中，Java类的全限定名为cn.rom.nativecppdemo.MainActivity，方法名为stringFromJNI，因此对应的C++函数名为Java_cn_rom_nativecppdemo_MainActivity_stringFromJNI，静态注册例子如下。

```
// java文件
public class MainActivity extends AppCompatActivity {
    static {
        System.loadLibrary("nativecppdemo");
    }
    ...
    public native String stringFromJNI();
}
// c++文件
extern "C" JNIEXPORT jstring JNICALL
Java_cn_rom_nativecppdemo_MainActivity_stringFromJNI(
        JNIEnv* env,
        jobject /* this */) {
    std::string hello = "Hello from C++";
    return env->NewStringUTF(hello.c_str());
}
```

静态注册函数必须使用JNIEXPORT和JNICALL修饰符，这两个修饰符是JNI中的预处理器宏。其中，JNIEXPORT会将函数名称保存到动态符号表，在注册时通过dlsym函数找到该函数。

JNICALL宏主要用于消除不同编译器和操作系统之间的调用规则差异。在不同平台上，本地方法的参数传递、调用约定和名称修饰等方面可能存在差异。这些差异可能导致在一个平台上编译的共享库无法在另一个平台上运行。为了解决这个问题，JNI规范定义了一种标准的本地方法命名方式，即"Java_包名_类名_方法名"的格式。使用JNICALL宏可以让编译器根据规范自动生成符合要求的本地方法名，从而确保能够正确调用本地方法。

需要注意的是，尽管JNICALL可以帮助我们消除平台差异，在某些情况下仍然需要手动指定本地方法名称。例如当我们需要使用JNI反射机制来动态调用本地方法时。此时，我们需要显式指定注册本地方法时所需绑定到Java代码中相应方法上去。

对于静态注册而言，并未看到直接使用RegisterNative进行注册操作，但实际内部已经进行了隐式注册。具体来说，当Java类被加载时，会调用LoadMethod将方法加载到虚拟机中，并随后通过LinkCode将Native函数与Java函数进行链接。下面是相关代码片段：

```
void ClassLinker::LoadClass(Thread* self,
                            const DexFile& dex_file,
                            const dex::ClassDef& dex_class_def,
                            Handle<mirror::Class> klass) {
    ...
    // 遍历一个 Java 类的所有字段和方法，并对它们进行操作
    accessor.VisitFieldsAndMethods([&](
        const ClassAccessor::Field& field) REQUIRES_SHARED(Locks::mutator_lock_) {
          ...
          // 所有字段
          LoadMethod(dex_file, method, klass, art_method);
          LinkCode(this, art_method, oat_class_ptr, class_def_method_index);
          ...
        }, [&](const ClassAccessor::Method& method)
REQUIRES_SHARED(Locks::mutator_lock_) {
          // 所有方法
          ArtMethod* art_method = klass->GetVirtualMethodUnchecked(
              class_def_method_index - accessor.NumDirectMethods(),
              image_pointer_size_);
          LoadMethod(dex_file, method, klass, art_method);
          LinkCode(this, art_method, oat_class_ptr, class_def_method_index);
          ++class_def_method_index;
        });
    ...
}
```

下面继续看看LinkCode的实现，如果已经被编译就会有Oat文件，就可以获取到quick_code，直接从二进制中调用来快速执行，否则走解释执行。

```
static void LinkCode(ClassLinker* class_linker,
                     ArtMethod* method,
                     const OatFile::OatClass* oat_class,
                     uint32_t class_def_method_index)
REQUIRES_SHARED(Locks::mutator_lock_) {
  ...
  const void* quick_code = nullptr;
  if (oat_class != nullptr) {
    const OatFile::OatMethod oat_method = oat_class-
>GetOatMethod(class_def_method_index);
    quick_code = oat_method.GetQuickCode();
  }
  // 是否使用解释执行
  bool enter_interpreter = class_linker->ShouldUseInterpreterEntrypoint(method,
quick_code);
  // 为指定的java函数设置二进制的快速执行入口
  if (quick_code == nullptr) {
```

```
    method->SetEntryPointFromQuickCompiledCode(
        method->IsNative() ? GetQuickGenericJniStub() :
GetQuickToInterpreterBridge());
  } else if (enter_interpreter) {
    method->SetEntryPointFromQuickCompiledCode(GetQuickToInterpreterBridge());
  } else if (NeedsClinitCheckBeforeCall(method)) {
    method->SetEntryPointFromQuickCompiledCode(GetQuickResolutionStub());
  } else {
    method->SetEntryPointFromQuickCompiledCode(quick_code);
  }

  if (method->IsNative()) {
    // 为指定的java函数设置JNI入口点，IsCriticalNative表示java中带有@CriticalNative标
记的native函数。一般的普通函数会调用后面的GetJniDlsymLookupStub
    method->SetEntryPointFromJni(
        method->IsCriticalNative() ? GetJniDlsymLookupCriticalStub() :
GetJniDlsymLookupStub());

    if (enter_interpreter || quick_code == nullptr) {
      DCHECK(class_linker->IsQuickGenericJniStub(method-
>GetEntryPointFromQuickCompiledCode()));
    }
  }
}
```

上面可以看到JNI设置入口点有两种情况，Critical Native方法通常用于需要高性能、低延迟和可预测行为的场景，例如音频处理、图像处理、网络协议栈等。一般情况开发者使用的都是普通Native函数，所以会调用后者GetJniDlsymLookupStub，接着继续看看实现代码。

```
static inline const void* GetJniDlsymLookupStub() {
  return reinterpret_cast<const void*>(art_jni_dlsym_lookup_stub);
}
```

这里看到就是将一个函数指针转换后返回，这个函数指针对应的是一段汇编代码，下面看看汇编代码实现。

```
ENTRY art_jni_dlsym_lookup_stub
    // spill regs.
    ...
    bl    artFindNativeMethod
    b     .Llookup_stub_continue
    .Llookup_stub_fast_or_critical_native:
    bl    artFindNativeMethodRunnable
    ...

1:
    ret             // restore regs and return to caller to handle exception.
END art_jni_dlsym_lookup_stub
```

能看到里面调用了artFindNativeMethod和artFindNativeMethodRunnable继续查看相关函数。

```
extern "C" const void* artFindNativeMethod(Thread* self) {
  DCHECK_EQ(self, Thread::Current());
  Locks::mutator_lock_->AssertNotHeld(self);  // We come here as Native.
  ScopedObjectAccess soa(self);
  return artFindNativeMethodRunnable(self);
}

extern "C" const void* artFindNativeMethodRunnable(Thread* self)
    REQUIRES_SHARED(Locks::mutator_lock_) {
  Locks::mutator_lock_->AssertSharedHeld(self);  // We come here as Runnable.
  uint32_t dex_pc;
  ArtMethod* method = self->GetCurrentMethod(&dex_pc);
  DCHECK(method != nullptr);
  ClassLinker* class_linker = Runtime::Current()->GetClassLinker();
  // 非静态函数的处理
  if (!method->IsNative()) {
    ...
  }
  // 如果注册过了，这里就会直接获取到，返回对应的地址
  const void* native_code = class_linker->GetRegisteredNative(self, method);
  if (native_code != nullptr) {
    return native_code;
  }
  // 查找对应的函数地址
  JavaVMExt* vm = down_cast<JNIEnvExt*>(self->GetJniEnv())->GetVm();
  native_code = vm->FindCodeForNativeMethod(method);
  if (native_code == nullptr) {
    self->AssertPendingException();
    return nullptr;
  }
  // 最后通过Linker进行注册
  return class_linker->RegisterNative(self, method, native_code);
}
```

FindCodeForNativeMethod执行到内部最后是通过dlsym查找符号，并且成功在这里看到了前文所说的隐式调用的RegisterNative。

6.3.2 动态注册

动态注册一般是写代码手动注册，将指定的符号名与对应的函数地址进行关联，在AOSP源码中Native函数大部分都是使用动态注册方式的，动态注册例子如下。

```
// java文件
public class MainActivity extends AppCompatActivity {
    static {
        System.loadLibrary("native-lib");
    }
```

```
    public native String stringFromJNI2();

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        TextView tv = findViewById(R.id.sample_text);
        tv.setText(stringFromJNI());
    }
}

//c++文件
jstring stringFromJNI2(JNIEnv* env, jobject /* this */) {
    return env->NewStringUTF("Hello from C++");
}

// 在 JNI_OnLoad 中进行动态注册
JNIEXPORT jint JNICALL
JNI_OnLoad(JavaVM* vm, void* reserved) {
    JNIEnv* env;
    if (vm->GetEnv(reinterpret_cast<void**>(&env), JNI_VERSION_1_6) != JNI_OK) {
        return -1;
    }

    // 手动注册 stringFromJNI 方法
    jclass clazz = env->FindClass("com/example/myapplication/MainActivity");
    JNINativeMethod methods[] = {
            {"stringFromJNI2", "()Ljava/lang/String;", reinterpret_cast<void *>
(stringFromJNI2)}
    };
    env->RegisterNatives(clazz, methods, sizeof(methods)/sizeof(methods[0]));

    return JNI_VERSION_1_6;
}
```

动态注册中是直接调用JniEnv的RegisterNatives进行注册的，找到对应的实现代码如下。

```
  static jint RegisterNatives(JNIEnv* env,
                              jclass java_class,
                              const JNINativeMethod* methods,
                              jint method_count) {
    ...
    // 遍历所有需要注册的函数
    for (jint i = 0; i < method_count; ++i) {
      // 取出函数名，函数签名，函数地址
      const char* name = methods[i].name;
      const char* sig = methods[i].signature;
      const void* fnPtr = methods[i].fnPtr;
      ...
      // 遍历Java对象的继承层次结构，也就是所有父类，来获取函数
      for (ObjPtr<mirror::Class> current_class = c.Get();
```

```
           current_class != nullptr;
           current_class = current_class->GetSuperClass()) {
        m = FindMethod<true>(current_class, name, sig);
        if (m != nullptr) {
          break;
        }
        m = FindMethod<false>(current_class, name, sig);
        if (m != nullptr) {
          break;
        }
        ...
      }

      if (m == nullptr) {
        ...
        return JNI_ERR;
      } else if (!m->IsNative()) {
        ...
        return JNI_ERR;
      }
      ...
      // 内部也是调用了Linker的RegisterNative
      const void* final_function_ptr = class_linker->RegisterNative(soa.Self(), m,
fnPtr);
      UNUSED(final_function_ptr);
    }
    return JNI_OK;
  }
```

在动态注册中，同样看到内部是调用了Linker的RegisterNative进行注册的，最后我们看看Linker中的实现。

```
const void* ClassLinker::RegisterNative(
    Thread* self, ArtMethod* method, const void* native_method) {
  CHECK(method->IsNative()) << method->PrettyMethod();
  CHECK(native_method != nullptr) << method->PrettyMethod();
  void* new_native_method = nullptr;
  Runtime* runtime = Runtime::Current();
  runtime->GetRuntimeCallbacks()->RegisterNativeMethod(method,
                                                       native_method,
                                                       /*out*/&new_native_method);
  if (method->IsCriticalNative()) {
    ...
  } else {
    // 给指定的java函数设置对应的Native函数的入口地址。
    method->SetEntryPointFromJni(new_native_method);
  }
  return new_native_method;
}
```

分析到这里，就已经明白两个目标需求如何实现了： ClassLinker::RegisterNative是静态注册和动态注册执行流程中的共同点，该函数的返回值就是Native函数的入口地址。接下来可以开始进行插桩输出了。

6.3.3 RegisterNative实现插桩

前文简单介绍ROM插桩其实就是输出日志，找到了合适的时机，以及要输出的内容，最后就是输出日志即可。在函数ClassLinker::RegisterNative调用结束时插入日志输出如下

```
#inclue
const void* ClassLinker::RegisterNative(
    Thread* self, ArtMethod* method, const void* native_method) {
  ...
  LOG(INFO) << "[ROM] ClassLinker::RegisterNative "<<method-
>PrettyMethod().c_str()<<" native_ptr:"<<new_native_method<<" method_idx:"
<<method->GetMethodIndex()<<" baseAddr:"<<base_addr;
  return new_native_method;
}
```

刷机编译后，安装测试demo，输出结果如下，成功打印出静态注册和动态注册的对应函数以及其函数地址。

```
rom.nativedem: [ROM] ClassLinker::RegisterNative java.lang.String
cn.rom.nativedemo.MainActivity.stringFromJNI2() native_ptr:0x7983a918c8
method_idx:632
rom.nativedem: [ROM] ClassLinker::RegisterNative java.lang.String
cn.rom.nativedemo.MainActivity.stringFromJNI() native_ptr:0x7983a916e8
method_idx:631
```

这里尽管已经输出了函数地址，但是可以再进行细节的优化，比如将函数地址去掉动态库的基址，获取到文件中的真实函数偏移。在这个时机已知了函数地址，只需要遍历已加载的所有动态库，计算出动态库结束地址，如果函数地址在某个动态库范围中，则返回动态库基址，最后打桩时，使用函数地址减掉基址即可拿到真实偏移了。实现代码如下。

```
#include "link.h"
#include "utils/Log.h"

// 遍历输出所有已经加载的动态库
int dl_iterate_callback(struct dl_phdr_info* info, size_t , void* data) {
    uintptr_t addr = reinterpret_cast<uintptr_t>(*(void**)data);
    // 计算出结束地址
    void* endptr=  (void*)(info->dlpi_addr + info->dlpi_phdr[info->dlpi_phnum -
1].p_vaddr + info->dlpi_phdr[info->dlpi_phnum - 1].p_memsz);
    uintptr_t end=reinterpret_cast<uintptr_t>(endptr);
    ALOGD("[ROM] native: %p\n", (void*)addr);
    ALOGD("[ROM] Library name: %s\n", info->dlpi_name);
    ALOGD("[ROM] Library base address: %p\n", (void*) info->dlpi_addr);
    ALOGD("[ROM] Library end address: %p\n\n",endptr);
    // 函数地址在动态库范围则返回该动态库的基址
```

```
    if(addr >= info->dlpi_addr && addr<=end){
        ALOGD("[ROM] Library found address: %p\n\n",(void*)info->dlpi_addr);
        reinterpret_cast<void**>(data)[0] = reinterpret_cast<void*>(info->dlpi_addr);
    }
    return 0;
}

// 根据函数地址获取对应动态库的基址
void* FindLibraryBaseAddress(void* entry_addr) {
    void* lib_base_addr = entry_addr;
    // 遍历所有加载的动态库，设置回调函数
    dl_iterate_phdr(dl_iterate_callback, &lib_base_addr);
    return lib_base_addr;
}


const void* ClassLinker::RegisterNative(
    Thread* self, ArtMethod* method, const void* native_method) {
    ...
    void * native_ptr=new_native_method;
    void* base_addr=FindLibraryBaseAddress(native_ptr);
    // 指针尽量转换后再进行操作，避免出现问题。
    uintptr_t native_data = reinterpret_cast<uintptr_t>(native_ptr);
    uintptr_t base_data = reinterpret_cast<uintptr_t>(base_addr);
    uintptr_t offset=native_data-base_data;
    ALOGD("[ROM] ClassLinker::RegisterNative %s native_ptr:%p method_idx:%p offset:0x%lx",method->PrettyMethod().c_str(),new_native_method,method->GetMethodIndex(),(void*)offset);
    return new_native_method;
}
```

优化后的输出日志如下

```
rom.nativedem: [ROM] native: 0x7a621108c8
rom.nativedem: [ROM] Library name:
/data/app/~~sm_GZ36XVwW9zZJGRl1ABg==/cn.rom.nativedemo-
VJiQEEQ3s9XXRMp6pkOKqA==/base.apk!/lib/arm64-v8a/libnativedemo.so
rom.nativedem: [ROM] Library base address: 0x7a62102000
rom.nativedem: [ROM] Library end address: 0x7a62136000
rom.nativedem: [ROM] Library found address: 0x7a62102000
rom.nativedem: [ROM] ClassLinker::RegisterNative java.lang.String
cn.rom.nativedemo.MainActivity.stringFromJNI2() native_ptr:0x7a621108c8
method_idx:0x278 offset:0xe8c8

rom.nativedem: [ROM] native: 0x7a621106e8
rom.nativedem: [ROM] Library name:
/data/app/~~sm_GZ36XVwW9zZJGRl1ABg==/cn.rom.nativedemo-
VJiQEEQ3s9XXRMp6pkOKqA==/base.apk!/lib/arm64-v8a/libnativedemo.so
rom.nativedem: [ROM] Library base address: 0x7a62102000
rom.nativedem: [ROM] Library end address: 0x7a62136000
```

```
rom.nativedem: [ROM] Library found address: 0x7a62102000
rom.nativedem: [ROM] ClassLinker::RegisterNative java.lang.String
cn.rom.nativedemo.MainActivity.stringFromJNI() native_ptr:0x7a621106e8
method_idx:0x277 offset:0xe6e8
```

尽管这里已经成功获取到了偏移地址，但是这里并不能确定拿到的end地址是正确的，对于end地址的计算方式，是获取最后一个段的地址+偏移+段大小计算出来的。下面简单修改一下函数对于段进行详细打印。

```
int dl_iterate_callback(struct dl_phdr_info* info, size_t , void* entry) {
    void* endptr=  (void*)(info->dlpi_addr + info->dlpi_phdr[info->dlpi_phnum -
1].p_vaddr + info->dlpi_phdr[info->dlpi_phnum - 1].p_memsz);
    LOGD("mikrom name:%s base:%p end:%p\n",info->dlpi_name,(void*)info-
>dlpi_addr,endptr);
    for (int j = 0; j < info->dlpi_phnum; j++) {
        LOGD("mikrom    %2d: [%14p; memsz:%7lx] flags: 0x%x; ", j,
               (void *) (info->dlpi_addr + info->dlpi_phdr[j].p_vaddr),
               info->dlpi_phdr[j].p_memsz,
               info->dlpi_phdr[j].p_flags);
    }
    return 0;
}
```

输出的结果如下所示，根据base和end的大小，可以看出来该结束地址是错误的。

```
mikrom name:/system/bin/linker64 base:0x7b21c17000 end:0x7b21c17290
mikrom     0: [  0x7b21c17040; memsz:    230] flags: 0x4;
mikrom     1: [  0x7b21c17000; memsz:  367b4] flags: 0x4;
mikrom     2: [  0x7b21c4e000; memsz:  e2940] flags: 0x5;
mikrom     3: [  0x7b21d31000; memsz:   7ca0] flags: 0x6;
mikrom     4: [  0x7b21d39ca0; memsz:   c0f0] flags: 0x6;
mikrom     5: [  0x7b21d381a8; memsz:    120] flags: 0x6;
mikrom     6: [  0x7b21d31000; memsz:   8000] flags: 0x4;
mikrom     7: [  0x7b21c2eacc; memsz:   5e14] flags: 0x4;
mikrom     8: [  0x7b21c17000; memsz:      0] flags: 0x6;
mikrom     9: [  0x7b21c17270; memsz:     20] flags: 0x4;
```

接下来查看maps中的映射动态库的地址，印证计算的结果是否正确。

```
7b21c17000-7b21c4e000 r--p 00000000 07:48 16
/apex/com.android.runtime/bin/linker64
7b21c4e000-7b21d31000 r-xp 00037000 07:48 16
/apex/com.android.runtime/bin/linker64
7b21d31000-7b21d39000 r--p 0011a000 07:48 16
/apex/com.android.runtime/bin/linker64
7b21d39000-7b21d3b000 rw-p 00121000 07:48 16
/apex/com.android.runtime/bin/linker64
```

为什么会出现这种错误呢，根据上面的段详细打印，以及前面的计算方式，是使用最后一个段的地址+段大小，得到的结束地址，也就是0x7b21c17270+0x20=end(0x7b21c17290)，所以这里需要将逻辑进行优化，应该取段中，最大的地址+段大小=end。同时优化代码，将动态库名称一起返回展示。代码如下。

```
// add mikrom
typedef struct{
    uintptr_t addr;
    uintptr_t baseAddr;
    std::string moduleName="";
}ModuleStruck;

int dl_iterate_callback(struct dl_phdr_info* info, size_t , void* entry) {
    ModuleStruck* mod=reinterpret_cast<ModuleStruck*>(entry);
    if(strlen(mod->moduleName.c_str())>0){
        return 0;
    }
    uintptr_t addr = mod->addr;
//    void* endptr=  (void*)(info->dlpi_addr + info->dlpi_phdr[info->dlpi_phnum -
1].p_vaddr + info->dlpi_phdr[info->dlpi_phnum - 1].p_memsz);
    uint64_t maxAddr=0;
    uint64_t maxMemsz=0;
    for (int j = 0; j < info->dlpi_phnum; j++) {
        if((info->dlpi_addr + info->dlpi_phdr[j].p_vaddr)>maxAddr){
            maxAddr=info->dlpi_addr + info->dlpi_phdr[j].p_vaddr;
            maxMemsz=info->dlpi_phdr[j].p_memsz;
        }
    }
    uintptr_t end=maxAddr+maxMemsz;
//    ALOGD("mikrom native:%p name:%s base:%p end:%p\n", (void*)addr,info-
>dlpi_name,(void*)info->dlpi_addr,(void*)end);
    if(addr >= info->dlpi_addr && addr<=end){
//        ALOGD("mikrom Library found native:%p base:%p\n\n",(void*)addr,
(void*)info->dlpi_addr);
        mod->baseAddr=info->dlpi_addr;
        mod->moduleName=info->dlpi_name;
    }
    return 0;
}

void FindLibraryBaseAddress(ModuleStruck* entry) {
    dl_iterate_phdr(dl_iterate_callback, entry);
}
// addend

const void* ClassLinker::RegisterNative(
    Thread* self, ArtMethod* method, const void* native_method) {
  ...
  if (method->IsCriticalNative()) {
    ...
  } else {
    method->SetEntryPointFromJni(new_native_method);
    if(Runtime::Current()->GetConfigItem().isRegisterNativePrint){
```

```
        void * native_ptr=new_native_method;
        ModuleStruck mod;
        mod.addr=(uintptr_t)native_ptr;
        FindLibraryBaseAddress(&mod);
        uintptr_t base_addr=mod.baseAddr;

        uintptr_t native_data = reinterpret_cast<uintptr_t>(native_ptr);
        uintptr_t base_data = reinterpret_cast<uintptr_t>(base_addr);
        uintptr_t offset=native_data-base_data;
        ALOGD("mikrom ClassLinker::RegisterNative %s native_ptr:%p method_idx:0x%x
offset:%p module_name:%s",method->PrettyMethod().c_str(),new_native_method,method-
>GetMethodIndex(),(void*)offset,mod.moduleName.c_str());
    }
  }
  return new_native_method;
}
```

最终在实际应用中效果展示如下。

```
2023-08-01 15:24:08.816  5468-5659  com.UCMobile            com.UCMobile
D  mikrom ClassLinker::RegisterNative void
com.alibaba.mbg.unet.internal.UNetRequestJni.nativeSetExtraInfo(long, int,
java.lang.String[]) native_ptr:0xb60d7ae9 method_idx:0x11 offset:0x108ae9
module_name:/data/app/~~JHAIRSSsk6jBdgjb90EHSQ==/com.UCMobile-aUSjEzsqxD6w8l-
eJW7kWw==/lib/arm/libunet.so
2023-08-01 15:24:08.816  5468-5659  com.UCMobile            com.UCMobile
D  mikrom ClassLinker::RegisterNative void
com.alibaba.mbg.unet.internal.UNetRequestJni.nativeAddLogScene(long,
java.lang.String, java.lang.String, java.lang.String) native_ptr:0xb60d7c95
method_idx:0x1 offset:0x108c95
module_name:/data/app/~~JHAIRSSsk6jBdgjb90EHSQ==/com.UCMobile-aUSjEzsqxD6w8l-
eJW7kWw==/lib/arm/libunet.so
2023-08-01 15:24:08.816  5468-5659  com.UCMobile            com.UCMobile
D  mikrom ClassLinker::RegisterNative void
com.alibaba.mbg.unet.internal.UNetRequestJni.nativeSetEnableDeepPrefetch(long,
boolean) native_ptr:0xb60d7d5d method_idx:0x10 offset:0x108d5d
module_name:/data/app/~~JHAIRSSsk6jBdgjb90EHSQ==/com.UCMobile-aUSjEzsqxD6w8l-
eJW7kWw==/lib/arm/libunet.so
2023-08-01 15:24:08.816  5468-5659  com.UCMobile            com.UCMobile
D  mikrom ClassLinker::RegisterNative void
com.alibaba.mbg.unet.internal.UNetRequestStatJni.nativeDestroy(long)
native_ptr:0xb60d8895 method_idx:0x2 offset:0x109895
module_name:/data/app/~~JHAIRSSsk6jBdgjb90EHSQ==/com.UCMobile-aUSjEzsqxD6w8l-
eJW7kWw==/lib/arm/libunet.so
```

6.4 自定义系统服务

自定义系统服务是指在操作系统中创建自己的服务，以便在需要时可以使用它。系统服务可以在系统启动时自动运行且没有UI界面，在后台执行某些特定任务或提供某些功能。由于系统服务有着`system`身份的权限，所以自定义系统服务可以用于各种用途。

以下是一些例子：

1. **系统监控与管理**：通过定期收集和分析系统数据，自动化报警和管理，保证系统稳定性和安全性；
2. **自动化部署和升级**：通过编写脚本和程序实现软件的自动化部署和升级，简化人工干预过程；
3. **数据备份与恢复**：通过编写脚本和程序实现数据备份和恢复，保证数据安全性和连续性；
4. **后台任务处理**：例如定时清理缓存、定时更新索引等任务，减轻人工干预压力，并提高系统效率。

第三章已经简单介绍了如何启动一个系统服务。要添加一个新的自定义系统服务，请参考 AOSP 源码中的添加方式来逐步完成。接下来我们将参考源码来添加一个最简单的名为 ROM_SERVICE 的自定义系统服务。

首先，在文件frameworks/base/core/java/android/content/Context.java中可以找到定义了各种系统服务的名称。在这里，我们将参考POWER_SERVICE服务的添加方式，在其下面添加自定义的服务。同时，在该文件中寻找其他处理POWER_SERVICE的代码段，并将自定义的服务同样进行处理。以下是相关代码示例：

```
public abstract class Context {
    @StringDef(suffix = { "_SERVICE" }, value = {
            POWER_SERVICE,
            ...
            [ROM]_SERVICE,
    })
    ...
    public static final String POWER_SERVICE = "power";
    public static final String [ROM]_SERVICE = "[ROM]";
}
```

接着搜索POWER_SERVICE找到该服务注册的地方，找到了文件frameworks/base/core/java/android/app/SystemServiceRegistry.java中进行了注册，所以在注册该服务的下方，模仿源码添加对自定义服务的注册。

```
public final class SystemServiceRegistry {
    ...
    static {
        ...
        //POWER_SERVICE服务注册
        registerService(Context.POWER_SERVICE, PowerManager.class,
                new CachedServiceFetcher<PowerManager>() {
            @Override
            public PowerManager createService(ContextImpl ctx) throws
ServiceNotFoundException {
                IBinder powerBinder =
ServiceManager.getServiceOrThrow(Context.POWER_SERVICE);
                IPowerManager powerService =
IPowerManager.Stub.asInterface(powerBinder);
                IBinder thermalBinder =
ServiceManager.getServiceOrThrow(Context.THERMAL_SERVICE);
                IThermalService thermalService =
IThermalService.Stub.asInterface(thermalBinder);
                return new PowerManager(ctx.getOuterContext(), powerService,
thermalService,
```

```
                        ctx.mMainThread.getHandler());
            }});
        //新增的自定义服务注册
        registerService(Context.[ROM]_SERVICE, [ROM]Manager.class,
                        new CachedServiceFetcher<[ROM]Manager>() {
                    @Override
                    public [ROM]Manager createService(ContextImpl ctx) throws
ServiceNotFoundException {
                        IBinder [ROM]Binder =
ServiceManager.getServiceOrThrow(Context.[ROM]_SERVICE);
                        I[ROM]Manager [ROM]Service =
I[ROM]Manager.Stub.asInterface([ROM]Binder);
                        return new [ROM]Manager(ctx.getOuterContext(),
[ROM]Service, ctx.mMainThread.getHandler());
                    }});
        ...
    }
    ...
}
```

PowerManager的功能中用到了THERMAL_SERVICE系统服务，所以这里不必完全照搬，省略掉这个参数即可。接下来发现注册时用到的I[ROM]Manager、[ROM]Manager并不存在，所以继续参考PowerManager的实现，先寻找IPowerManager在哪里定义的，通过搜索，发现该接口在文件frameworks/base/core/java/android/os/IPowerManager.aidl中。在同目录下新建文件I[ROM]Manager.aidl并添加简单的接口内容如下。

```
package android.os;

interface I[ROM]Manager
{
    String hello();
}
```

AIDL（Android接口定义语言）是一种Android平台上的IPC机制，用于不同应用程序组件之间进行进程通信。要使用AIDL实现进程间通信，首先需要创建一个.aidl文件来定义接口。将.aidl文件编译成Java接口，并在服务端和客户端中分别实现该接口。最后，在服务端通过bindService 方法绑定服务并向客户端返回IBinder对象。使用AIDL可以轻松地实现跨进程通信。

添加完毕后还需要找到在哪里将这个文件添加到编译中的，搜索IPowerManager.aidl后，找到文件frameworks/base/core/java/Android.bp中进行处理的。所以跟着加上刚刚定义的aidl文件。修改如下。

```
filegroup {
    name: "libpowermanager_aidl",
    srcs: [
        ...
        "android/os/IPowerManager.aidl",
        "android/os/I[ROM]Manager.aidl",
```

```
    ],
}
```

然后继续寻找IPowerManager.aidl在哪里进行实现的，搜索IPowerManager.Stub，找到文件frameworks/base/services/core/java/com/android/server/power/PowerManagerService.java实现的具体的逻辑。该服务的路径是在power目录下，并不适合存放自定义的服务，所以选择在更上级目录创建一个对应的新文件frameworks/base/services/core/java/com/android/server/[ROM]ManagerService.java，代码如下。

```
public class [ROM]ManagerService extends I[ROM]Manager.Stub {
    private Context mContext;
    private String TAG="[ROM]ManagerService";
    public [ROM]ManagerService(Context context){
        mContext=context;
    }

    @Override
    public String hello(){
        return "hello [ROM] service";
    }
}
```

继续找到PowerManager的实现，在文件 frameworks/base/core/java/android/os/PowerManager.java 中，所以在这个目录中创建文件[ROM]Manager.java，代码实现如下。

```
package android.os;

@SystemService(Context.[ROM]_SERVICE)
public final class [ROM]Manager {
    private static final String TAG = "[ROM]Manager";
    final Context mContext;
    @UnsupportedAppUsage
    final I[ROM]Manager mService;
    @UnsupportedAppUsage(maxTargetSdk = Build.VERSION_CODES.P)
    final Handler mHandler;
    public [ROM]Manager(Context context, I[ROM]Manager service,Handler handler) {
            mContext = context;
            mService = service;
            mHandler = handler;
    }

    public String hello(){
        return mService.hello();
    }
}
```

到这里注册一个自定义的系统服务基本完成了，最后是启动这个自定义的服务，而启动的流程在第三章中有详细的介绍，在文件frameworks/base/services/java/com/android/server/SystemServer.java中启动，这里选择系统准备就绪后的时机再拉起这个服务，参考其他任意服务启动的方式即可。

```
private void startOtherServices(@NonNull TimingsTraceAndSlog t) {

    ...
    // 参考其他服务是如何拉起的
    t.traceBegin("StartNetworkStatsService");
    try {
        networkStats = NetworkStatsService.create(context, networkManagement);
        ServiceManager.addService(Context.NETWORK_STATS_SERVICE, networkStats);
    } catch (Throwable e) {
        reportWtf("starting NetworkStats Service", e);
    }
    t.traceEnd();

    // 启动自定义的服务
    t.traceBegin("Start[ROM]ManagerService");
    try {
        [ROM]ManagerService [ROM]Service = new [ROM]ManagerService(context);
        ServiceManager.addService(Context.[ROM]_SERVICE,[ROM]Service);
    } catch (Throwable e) {
        reportWtf("starting [ROM] Service", e);
    }
    t.traceEnd();
    ...
}
```

到这里基本准备就绪了，可以开始尝试编译，由于添加了aidl文件，所以需要先调用make update-api进行编译，编译过程如下，最后出现编译报错。

```
source ./build/envsetup.sh

lunch aosp_blueline-userdebug

make update-api -j8

// 出现下面的错误
frameworks/base/core/java/android/os/[ROM]Manager.java:10: error: Method parameter
type `android.content.Context` violates package layering: nothin
g in `package android.os` should depend on `package android.content`
[PackageLayering]
frameworks/base/core/java/android/os/[ROM]Manager.java:16: error: Managers must
always be obtained from Context; no direct constructors [ManagerCon
structor]
frameworks/base/core/java/android/os/[ROM]Manager.java:16: error: Missing
nullability on parameter `context` in method `[ROM]Manager` [MissingNull
ability]
```

```
frameworks/base/core/java/android/os/[ROM]Manager.java:16: error: Missing
nullability on parameter `service` in method `[ROM]Manager` [MissingNull
ability]
```

这是由于Android 11以后谷歌强制开启lint检查来提高应用程序的质量和稳定性。Lint检查是Android Studio中的一个静态分析工具，用于检测代码中可能存在的潜在问题和错误。它可以帮助开发人员找到并修复代码中的bug、性能问题、安全漏洞等。可以设置让其忽略掉对这个android.os目录的检查，修改文件framewoks/base/Android.bp文件如下。

```
metalava_framework_docs_args = "--manifest $(location
core/res/AndroidManifest.xml) " +
    ...
    "--api-lint-ignore-prefix android.os."
```

根据上面另一个错误提示知道Managers必须是单例模式，并且String的参数火返回值需要允许为null值的，也就是要携带@Nullable注解，调用service函数时，需要捕获异常。针对以上的提示对[ROM]Manager进行调整如下。

```
package android.os;

import android.annotation.NonNull;
import android.annotation.Nullable;
import android.compat.annotation.UnsupportedAppUsage;
import android.content.Context;
import android.annotation.SystemService;
import android.os.I[ROM]Manager;

@SystemService(Context.[ROM]_SERVICE)
public final class [ROM]Manager {
    private static final String TAG = "[ROM]Manager";
    I[ROM]Manager mService;
    public [ROM]Manager(I[ROM]Manager service) {
            mService = service;
    }
    private static [ROM]Manager sInstance;
    /**
     *@hide
     */
    @NonNull
    @UnsupportedAppUsage
    public static [ROM]Manager getInstance() {
        synchronized ([ROM]Manager.class) {
            if (sInstance == null) {
                try {
                    IBinder [ROM]Binder =
ServiceManager.getServiceOrThrow(Context.[ROM]_SERVICE);
                    I[ROM]Manager [ROM]Service =
I[ROM]Manager.Stub.asInterface([ROM]Binder);
                    sInstance= new [ROM]Manager([ROM]Service);
```

```
                } catch (ServiceManager.ServiceNotFoundException e) {
                    throw new IllegalStateException(e);
                }
            }
            return sInstance;
        }
    }
    @Nullable
    public String hello(){
        try{
            return mService.hello();
        }catch (RemoteException ex){
            throw ex.rethrowFromSystemServer();
        }
    }
}
```

除此之外，在注册该服务的地方也要对应的调整初始化的方式。调整如下

```
public final class SystemServiceRegistry {
    ...
    static {
        ...
        registerService(Context.[ROM]_SERVICE, [ROM]Manager.class,
                        new CachedServiceFetcher<[ROM]Manager>() {
                    @Override
                    public [ROM]Manager createService(ContextImpl ctx) throws
ServiceNotFoundException {
                        return [ROM]Manager.getInstance();
                    }});
        ...
    }
    ...
}
```

经过修改后，再重新编译就能正常编译完成了，最后还需要对selinux进行修改，对新增的服务设置权限。找到文件system/sepolicy/public/service.te，参考其他的服务定义，在最后添加一条类型定义如下。

```
type [ROM]_service, system_api_service, system_server_service,
service_manager_type;
```

然后找到文件system/sepolicy/private/service_contexts，在最后给我们Context中定义的[ROM]服务设置使用刚刚定义的[ROM]_service类型的权限，修改如下。

```
[ROM]                                     u:object_r:[ROM]_service:s0
```

为自定义的系统服务设置了selinux权限后，还需要给应用开启权限访问这个系统服务，找到system/sepolicy/public/untrusted_app.te文件，添加如下策略开放让其能查找该系统服务。

```
allow untrusted_app [ROM]_service:service_manager find;
allow untrusted_app_27 [ROM]_service:service_manager find;
allow untrusted_app_25 [ROM]_service:service_manager find;
```

这时直接编译会出现下面的错误。

```
FAILED:
~/android_src/out/target/product/blueline/obj/FAKE/sepolicy_freeze_test_intermedia
tes/sepolicy_freeze_test
/bin/bash -c "(diff -rq -x bug_map system/sepolicy/prebuilts/api/31.0/public
system/sepolicy/public ) && (diff -rq -x bug_map system/sepolicy/prebui
lts/api/31.0/private system/sepolicy/private ) && (touch
~/android_src/out/target/product/blueline/obj/FAKE/sepolicy_freeze_test_int
ermediates/sepolicy_freeze_test )"
```

在前文介绍selinux时有说到系统会使用prebuilts中的策略进行对比，这是因为prebuilts中包含了在Android设备上预置的 sepolicy策略和规则。所以当改动策略时，要将prebuilts下对应的文件做出相同的修改。因为对应要调整system/sepolicy/prebuilts/api/31.0/public/service.te和system/sepolicy/prebuilts/api/31.0/private/service_contexts进行和上面相同的调整。这里需要注意的是untrusted_app.te文件只需要修改prebuilts/api/31.0的即可，而service.te和service_contexts，需要将prebuilts/api/目录下所有版本都添加定义，否则会出现如下错误。

```
SELinux: The following public types were found added to the policy without an
entry into the compatibility mapping file(s) found in private/compat/V
.v/V.v[.ignore].cil, where V.v is the latest API level.
```

selinux策略修改完毕，成功编译后，刷入手机，检查服务是否成功开启。

```
adb shell

service list|grep [ROM]

// 成功查询到自定义的系统服务
120 [ROM]: [android.os.I[ROM]Manager]
```

最后开发测试的app对这个系统服务调用hello函数。创建一个Android项目，在java目录下创建package路径android.os，然后在该路径下创建一个文件I[ROM]Manager.aidl，内容和前文添加系统服务时一至，内容如下。

```
package android.os;

interface I[ROM]Manager
{
    String hello();
}
```

通过反射获取ServiceManager类，调用该类的getService函数得到[ROM]的系统服务，将返回的结果转换为刚刚定义的接口对象，最后调用目标函数拿到结果。实现代码如下。

```
public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        Class localClass = null;
        try {
            // 使用反射拿到系统服务
            localClass = Class.forName("android.os.ServiceManager");
            Method getServiceMethod = localClass.getMethod("getService", new
Class[] {String.class});
            if(getServiceMethod != null) {
                // 获取自定义的服务
                Object objResult = getServiceMethod.invoke(localClass, new
Object[]{"[ROM]"});
                if (objResult != null) {
                    IBinder binder = (IBinder) objResult;
                    I[ROM]Manager i[ROM] = I[ROM]Manager.Stub.asInterface(binder);
                    // 调用服务中的实现
                    String msg= i[ROM].hello();
                    Log.i("MainActivity", "msg: " + msg);
                }
            }
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        } catch (InvocationTargetException e) {
            e.printStackTrace();
        } catch (NoSuchMethodException e) {
            e.printStackTrace();
        } catch (IllegalAccessException e) {
            e.printStackTrace();
        } catch (RemoteException e) {
            e.printStackTrace();
        }
    }
}
```

最后成功输出结果如下。

```
cn.rom.myservicedemo I/MainActivity: msg: hello [ROM] service
```

6.5 APP权限修改

Android中的权限是指应用程序访问设备功能和用户数据所需的授权。在Android系统中，所有的应用程序都必须声明其需要的权限，以便在安装时就向用户展示，并且在运行时需要获取相应的授权才能使用。

这一节将介绍APP的权限，以及在源码中是如何加载AndroidManifest.xml文件获取到权限，最后尝试在加载流程中进行修改，让App默认具有一些权限，无需APP进行申请。

6.5.1 APP权限介绍

Android系统将权限分为普通权限和危险权限两类，其中危险权限需要用户明确授权才能使用，而普通权限则不需要。普通权限通常不涉及到用户隐私和设备安全问题，例如访问网络、读取手机状态等。而危险权限则可能会涉及到用户隐私和设备安全问题，例如读取联系人信息、访问摄像头等。 在AndroidManifest.xml文件中声明权限，可以使用<uses-permission>标签来声明需要的权限，例如：

```
<manifest package="com.example.app">
    <uses-permission android:name="android.permission.READ_CONTACTS" />
    <uses-permission android:name="android.permission.CAMERA" />
    ...
</manifest>
```

Android中的常见权限列表如下。

1. 日历权限：android.permission.READ_CALENDAR、android.permission.WRITE_CALENDAR
2. 相机权限：android.permission.CAMERA
3. 联系人权限：android.permission.READ_CONTACTS、android.permission.WRITE_CONTACTS、android.permission.GET_ACCOUNTS
4. 定位权限：android.permission.ACCESS_FINE_LOCATION、android.permission.ACCESS_COARSE_LOCATION
5. 麦克风权限：android.permission.RECORD_AUDIO
6. 手机状态和电话权限：android.permission.READ_PHONE_STATE、android.permission.CALL_PHONE、android.permission.READ_CALL_LOG、android.permission.WRITE_CALL_LOG、android.permission.ADD_VOICEMAIL、android.permission.USE_SIP、android.permission.PROCESS_OUTGOING_CALLS
7. 传感器权限：android.permission.BODY_SENSORS
8. 短信权限：android.permission.READ_SMS、android.permission.RECEIVE_SMS、android.permission.SEND_SMS、android.permission.RECEIVE_WAP_PUSH、android.permission.RECEIVE_MMS
9. 存储权限：android.permission.READ_EXTERNAL_STORAGE、android.permission.WRITE_EXTERNAL_STORAGE
10. 联网权限：android.permission.INTERNET

androidManifest.xml文件是Android应用程序的清单文件，它在应用程序安装和运行过程中都会被解析。Android系统启动时也会解析每个已安装应用程序的清单文件，以了解应用程序所需的权限、组件等信息，并将这些信息记录在系统中。而这项解析工作是由PackageManagerService系统服务来完成的。开始分析的入手点可以从该系统服务的启动开始。

6.5.2 权限解析源码跟踪

PackageManagerService系统服务的启动也是在SystemServer进程中，所以在SystemServer.java中搜索就能该进程启动的入口，相关代码如下。

```
private void startBootstrapServices() {
    ...
    t.traceBegin("StartPackageManagerService");
    try {
        Watchdog.getInstance().pauseWatchingCurrentThread("packagemanagermain");
        mPackageManagerService = PackageManagerService.main(mSystemContext,
installer,

domainVerificationService, mFactoryTestMode != FactoryTest.FACTORY_TEST_OFF,
                                                              mOnlyCore);
    } finally {
        Watchdog.getInstance().resumeWatchingCurrentThread("packagemanagermain");
    }
    ...
}
```

继续跟进看该服务的PackageManagerService.main函数

```
public static PackageManagerService main(Context context, Installer installer,
            @NonNull DomainVerificationService domainVerificationService, boolean
factoryTest,
            boolean onlyCore) {
        ...
        PackageManagerService m = new PackageManagerService(injector, onlyCore,
factoryTest,
                Build.FINGERPRINT, Build.IS_ENG, Build.IS_USERDEBUG,
Build.VERSION.SDK_INT,
                Build.VERSION.INCREMENTAL);
        ...
        ServiceManager.addService("package", m);
        final PackageManagerNative pmn = m.new PackageManagerNative();
        ServiceManager.addService("package_native", pmn);
        return m;
    }
```

然后这里调用了PackageManagerService的构造函数，继续查看构造函数代码。

```
public PackageManagerService(Injector injector, boolean onlyCore, boolean
factoryTest,
            final String buildFingerprint, final boolean isEngBuild,
            final boolean isUserDebugBuild, final int sdkVersion, final String
incrementalVersion) {
        ...
        synchronized (mInstallLock) {
        // writer
        synchronized (mLock) {
            ...
            // 遍历系统应用程序目录列表
            for (int i = mDirsToScanAsSystem.size() - 1; i >= 0; i--) {
                final ScanPartition partition = mDirsToScanAsSystem.get(i);
                if (partition.getOverlayFolder() == null) {
                    continue;
                }
                scanDirTracedLI(partition.getOverlayFolder(), systemParseFlags,
                        systemScanFlags | partition.scanFlag, 0,
                        packageParser, executorService);
            }

            scanDirTracedLI(frameworkDir, systemParseFlags,
                    systemScanFlags | SCAN_NO_DEX | SCAN_AS_PRIVILEGED, 0,
                    packageParser, executorService);
            ...

        } // synchronized (mLock)
        } // synchronized (mInstallLock)
        // CHECKSTYLE:ON IndentationCheck
        ...
    }
```

mDirsToScanAsSystem是PackageManagerService类中的一个成员变量，用于存储系统应用程序目录列表。

系统应用程序存储在多个目录中，例如/system/app、/system/priv-app等。当系统启动时，PackageManagerService类会扫描这些目录以查找系统应用程序，并将其添加到应用程序列表中。这个列表中的每个元素都是一个File对象，表示一个系统应用程序目录。

当系统启动时，PackageManagerService会遍历mDirsToScanAsSystem列表并扫描其中的所有目录以查找系统应用程序。如果发现新的应用程序，则将其添加到应用程序列表中；如果发现已删除或升级的应用程序，则将其添加到possiblyDeletedUpdatedSystemApps列表中进行后续处理。下面看看scanDirTracedLI方法的实现。

```
private void scanDirTracedLI(File scanDir, final int parseFlags, int scanFlags,
            long currentTime, PackageParser2 packageParser, ExecutorService
executorService) {
        Trace.traceBegin(TRACE_TAG_PACKAGE_MANAGER, "scanDir [" +
scanDir.getAbsolutePath() + "]");
        try {
            scanDirLI(scanDir, parseFlags, scanFlags, currentTime, packageParser,
```

```
executorService);
        } finally {
            Trace.traceEnd(TRACE_TAG_PACKAGE_MANAGER);
        }
    }

private void scanDirLI(File scanDir, int parseFlags, int scanFlags, long
currentTime,
            PackageParser2 packageParser, ExecutorService executorService) {
        final File[] files = scanDir.listFiles();
        if (ArrayUtils.isEmpty(files)) {
            Log.d(TAG, "No files in app dir " + scanDir);
            return;
        }

        if (DEBUG_PACKAGE_SCANNING) {
            Log.d(TAG, "Scanning app dir " + scanDir + " scanFlags=" + scanFlags
                    + " flags=0x" + Integer.toHexString(parseFlags));
        }

        ParallelPackageParser parallelPackageParser =
                new ParallelPackageParser(packageParser, executorService);

        // Submit files for parsing in parallel
        int fileCount = 0;
        // 遍历所有文件
        for (File file : files) {
            final boolean isPackage = (isApkFile(file) || file.isDirectory())
                    && !PackageInstallerService.isStageName(file.getName());
            if (!isPackage) {
                // Ignore entries which are not packages
                continue;
            }
            // 使用parallelPackageParser.submit()方法异步地将其提交给PackageParser类
来解析
            parallelPackageParser.submit(file, parseFlags);
            fileCount++;
        }
        ...
    }
```

以上代码可以看到scanDirLI方法主要是遍历所有文件筛选是Apk文件，或者是一个目录，isStageName方法是判断当前文件是否为分阶段安装的数据，parallelPackageParser.submit()方法异步地将其提交给PackageParser类来解析。跟踪查看submit。

```
public void submit(File scanFile, int parseFlags) {
        mExecutorService.submit(() -> {
            ParseResult pr = new ParseResult();
            Trace.traceBegin(TRACE_TAG_PACKAGE_MANAGER, "parallel parsePackage ["
+ scanFile + "]");
            try {
```

```
                pr.scanFile = scanFile;
                // 解析应用程序包
                pr.parsedPackage = parsePackage(scanFile, parseFlags);
            } catch (Throwable e) {
                pr.throwable = e;
            } finally {
                Trace.traceEnd(TRACE_TAG_PACKAGE_MANAGER);
            }
            try {
                // 返回数据
                mQueue.put(pr);
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
                // Propagate result to callers of take().
                // This is helpful to prevent main thread from getting stuck
waiting on
                // ParallelPackageParser to finish in case of interruption
                mInterruptedInThread = Thread.currentThread().getName();
            }
        });
    }
```

继续跟踪parsePackage是如何解析的

```
protected ParsedPackage parsePackage(File scanFile, int parseFlags)
            throws PackageParser.PackageParserException {
        return mPackageParser.parsePackage(scanFile, parseFlags, true);
    }
```

这里需要留意mPackageParser的类型是PackageParser2，而在AOSP10中，它的类型是PackageParser。继续查看parsePackage方法的实现。

```
public ParsedPackage parsePackage(File packageFile, int flags, boolean useCaches)
            throws PackageParserException {
        // 尝试从缓存中找解析结果
        if (useCaches && mCacher != null) {
            ParsedPackage parsed = mCacher.getCachedResult(packageFile, flags);
            if (parsed != null) {
                return parsed;
            }
        }

        long parseTime = LOG_PARSE_TIMINGS ? SystemClock.uptimeMillis() : 0;
        ParseInput input = mSharedResult.get().reset();
        // 解析
        ParseResult<ParsingPackage> result = parsingUtils.parsePackage(input,
packageFile, flags);
        if (result.isError()) {
            throw new PackageParserException(result.getErrorCode(),
```

```
    result.getErrorMessage(),
                        result.getException());
            }
            ...
            return parsed;
        }
```

继续跟踪parsingUtils.parsePackage的实现。

```
    public ParseResult<ParsingPackage> parsePackage(ParseInput input, File
    packageFile,
                int flags)
                throws PackageParserException {
            if (packageFile.isDirectory()) {
                return parseClusterPackage(input, packageFile, flags);
            } else {
                return parseMonolithicPackage(input, packageFile, flags);
            }
        }
```

如果是一个目录，则说明这是一个集群版本 (cluster package) 的应用程序包，可能由多个应用程序组成。在这种情况下，它调用parseClusterPackage方法对应用程序包进行解析，并返回解析结果。

parseClusterPackage方法会遍历该目录下的所有文件，解析其中的每个应用程序，并将它们打包成一个PackageParser.Package集合返回。每个PackageParser.Package对象表示单独的一个应用程序。

如果packageFile不是一个目录，则说明这是一个单体版本 (monolithic package) 的应用程序包，只包含一个应用程序。在这种情况下，它调用parseMonolithicPackage方法对应用程序包进行解析，并返回解析结果。

parseMonolithicPackage方法会读取应用程序包的内容，并解析其中的AndroidManifest.xml文件和资源文件等信息，然后创建一个PackageParser.Package对象来表示整个应用程序，并返回该对象作为解析结果。

跟踪一条路线即可，接下来查看parseMonolithicPackage的实现代码。

```
    private ParseResult<ParsingPackage> parseMonolithicPackage(ParseInput input, File
    apkFile,
                int flags) throws PackageParserException {
            ...
            try {
                // 解析应用程序
                final ParseResult<ParsingPackage> result = parseBaseApk(input,
                        apkFile,
                        apkFile.getCanonicalPath(),
                        assetLoader, flags);
                if (result.isError()) {
                    return input.error(result);
                }
```

```
            return input.success(result.getResult()
                    .setUse32BitAbi(lite.isUse32bitAbi()));
        } catch (IOException e) {
            return input.error(INSTALL_PARSE_FAILED_UNEXPECTED_EXCEPTION,
                    "Failed to get path: " + apkFile, e);
        } finally {
            IoUtils.closeQuietly(assetLoader);
        }
    }
```

继续查看parseBaseApk的实现代码。

```
private ParseResult<ParsingPackage> parseBaseApk(ParseInput input, File apkFile,
            String codePath, SplitAssetLoader assetLoader, int flags)
            throws PackageParserException {
        final String apkPath = apkFile.getAbsolutePath();
        ...
        // 读取AndroidMannifest.xml文件
        try (XmlResourceParser parser = assets.openXmlResourceParser(cookie,
                ANDROID_MANIFEST_FILENAME)) {
            final Resources res = new Resources(assets, mDisplayMetrics, null);
            // 调用另一个重载进行解析
            ParseResult<ParsingPackage> result = parseBaseApk(input, apkPath,
codePath, res,
                    parser, flags);
            ...
            return input.success(pkg);
        } catch (Exception e) {
            return input.error(INSTALL_PARSE_FAILED_UNEXPECTED_EXCEPTION,
                    "Failed to read manifest from " + apkPath, e);
        }
    }
```

在这里看到读取AndroidMannifest.xml配置文件了，随后调用另一个重载进行解析。代码如下。

```
private ParseResult<ParsingPackage> parseBaseApk(ParseInput input, String apkPath,
            String codePath, Resources res, XmlResourceParser parser, int flags)
            throws XmlPullParserException, IOException {
        ...
        final TypedArray manifestArray = res.obtainAttributes(parser,
R.styleable.AndroidManifest);
        try {
            final boolean isCoreApp =
                    parser.getAttributeBooleanValue(null, "coreApp", false);
            final ParsingPackage pkg = mCallback.startParsingPackage(
                    pkgName, apkPath, codePath, manifestArray, isCoreApp);
            // 解析Apk文件中xml的各种标签
            final ParseResult<ParsingPackage> result =
                    parseBaseApkTags(input, pkg, manifestArray, res, parser,
```

```
flags);
            if (result.isError()) {
                return result;
            }

            return input.success(pkg);
        } finally {
            manifestArray.recycle();
        }
    }
```

继续查看parseBaseApkTags的实现代码。

```
private ParseResult<ParsingPackage> parseBaseApkTags(ParseInput input,
ParsingPackage pkg,
            TypedArray sa, Resources res, XmlResourceParser parser, int flags)
            throws XmlPullParserException, IOException {
        ...
        while ((type = parser.next()) != XmlPullParser.END_DOCUMENT
                && (type != XmlPullParser.END_TAG
                || parser.getDepth() > depth)) {
            ...
            // <application> has special logic, so it's handled outside the
general method
            if (TAG_APPLICATION.equals(tagName)) {
                if (foundApp) {
                    ...
                } else {
                    foundApp = true;
                    result = parseBaseApplication(input, pkg, res, parser, flags);
                }
            } else {
                result = parseBaseApkTag(tagName, input, pkg, res, parser, flags);
            }

            if (result.isError()) {
                return input.error(result);
            }
        }
        ...
        return input.success(pkg);
    }
```

检查tagName是否为<application>标记。如果是<application>标记，则表示当前正在解析应用程序包的主要组件，在该标记中会定义应用程序的所有组件、权限等信息。如果没有发现<application>标记，则继续递归调用处理其他标记。所以接下来查看parseBaseApplication方法的实现。

```
private ParseResult<ParsingPackage> parseBaseApplication(ParseInput input,
            ParsingPackage pkg, Resources res, XmlResourceParser parser, int
```

```
flags)
            throws XmlPullParserException, IOException {
        final String pkgName = pkg.getPackageName();
        int targetSdk = pkg.getTargetSdkVersion();

        TypedArray sa = res.obtainAttributes(parser,
R.styleable.AndroidManifestApplication);
        try {
            ...
            // 解析应用程序包中基本APK文件的标志
            parseBaseAppBasicFlags(pkg, sa);
            ...
            // 根据xml配置，对pkg的值做相应的修改
            if (sa.getBoolean(R.styleable.AndroidManifestApplication_persistent,
false)) {
                // Check if persistence is based on a feature being present
                final String requiredFeature = sa.getNonResourceString(R.styleable

.AndroidManifestApplication_persistentWhenFeatureAvailable);
                pkg.setPersistent(requiredFeature == null ||
mCallback.hasFeature(requiredFeature));
            }

            if
(sa.hasValueOrEmpty(R.styleable.AndroidManifestApplication_resizeableActivity)) {
                pkg.setResizeableActivity(sa.getBoolean(
                        R.styleable.AndroidManifestApplication_resizeableActivity,
true));
            } else {
                pkg.setResizeableActivityViaSdkVersion(
                        targetSdk >= Build.VERSION_CODES.N);
            }
            ...

        } finally {
            sa.recycle();
        }
        ...
        // 根据xml中的tag进行对应的处理
        while ((type = parser.next()) != XmlPullParser.END_DOCUMENT
                && (type != XmlPullParser.END_TAG
                || parser.getDepth() > depth)) {
            if (type != XmlPullParser.START_TAG) {
                continue;
            }
            final ParseResult result;
            String tagName = parser.getName();
            boolean isActivity = false;
            switch (tagName) {
                case "activity":
                    isActivity = true;
                    // fall-through
                case "receiver":
                    ParseResult<ParsedActivity> activityResult =
```

```
ParsedActivityUtils.parseActivityOrReceiver(mSeparateProcesses, pkg,
                                    res, parser, flags, sUseRoundIcon, input);

                    if (activityResult.isSuccess()) {
                        ParsedActivity activity = activityResult.getResult();
                        if (isActivity) {
                            hasActivityOrder |= (activity.getOrder() != 0);
                            pkg.addActivity(activity);
                        } else {
                            hasReceiverOrder |= (activity.getOrder() != 0);
                            pkg.addReceiver(activity);
                        }
                    }

                    result = activityResult;
                    break;
                case "service":
                    ParseResult<ParsedService> serviceResult =
                            ParsedServiceUtils.parseService(mSeparateProcesses,
pkg, res, parser,
                                    flags, sUseRoundIcon, input);
                    if (serviceResult.isSuccess()) {
                        ParsedService service = serviceResult.getResult();
                        hasServiceOrder |= (service.getOrder() != 0);
                        pkg.addService(service);
                    }

                    result = serviceResult;
                    break;
                case "provider":
                    ParseResult<ParsedProvider> providerResult =
                            ParsedProviderUtils.parseProvider(mSeparateProcesses,
pkg, res, parser,
                                    flags, sUseRoundIcon, input);
                    if (providerResult.isSuccess()) {
                        pkg.addProvider(providerResult.getResult());
                    }

                    result = providerResult;
                    break;
                case "activity-alias":
                    activityResult = ParsedActivityUtils.parseActivityAlias(pkg,
res,
                            parser, sUseRoundIcon, input);
                    if (activityResult.isSuccess()) {
                        ParsedActivity activity = activityResult.getResult();
                        hasActivityOrder |= (activity.getOrder() != 0);
                        pkg.addActivity(activity);
                    }

                    result = activityResult;
                    break;
                default:
```

```
                    result = parseBaseAppChildTag(input, tagName, pkg, res,
parser, flags);
                    break;
            }

            if (result.isError()) {
                return input.error(result);
            }
        }
        ...
        return input.success(pkg);
    }
```

基本大多数的解析都在这里实现了，最后看看基本APK标志是如何解析处理的。parseBaseAppBasicFlags的实现如下。

```
private void parseBaseAppBasicFlags(ParsingPackage pkg, TypedArray sa) {
        int targetSdk = pkg.getTargetSdkVersion();
        //@formatter:off
        // CHECKSTYLE:off
        pkg
                // Default true
                .setAllowBackup(bool(true,
R.styleable.AndroidManifestApplication_allowBackup, sa))
                .setAllowClearUserData(bool(true,
R.styleable.AndroidManifestApplication_allowClearUserData, sa))
                .setAllowClearUserDataOnFailedRestore(bool(true,
R.styleable.AndroidManifestApplication_allowClearUserDataOnFailedRestore, sa))
                .setAllowNativeHeapPointerTagging(bool(true,
R.styleable.AndroidManifestApplication_allowNativeHeapPointerTagging, sa))
                .setEnabled(bool(true,
R.styleable.AndroidManifestApplication_enabled, sa))
                .setExtractNativeLibs(bool(true,
R.styleable.AndroidManifestApplication_extractNativeLibs, sa))
                .setHasCode(bool(true,
R.styleable.AndroidManifestApplication_hasCode, sa))
                // Default false
                .setAllowTaskReparenting(bool(false,
R.styleable.AndroidManifestApplication_allowTaskReparenting, sa))
                .setCantSaveState(bool(false,
R.styleable.AndroidManifestApplication_cantSaveState, sa))
                .setCrossProfile(bool(false,
R.styleable.AndroidManifestApplication_crossProfile, sa))
                .setDebuggable(bool(false,
R.styleable.AndroidManifestApplication_debuggable, sa))
                .setDefaultToDeviceProtectedStorage(bool(false,
R.styleable.AndroidManifestApplication_defaultToDeviceProtectedStorage, sa))
                .setDirectBootAware(bool(false,
R.styleable.AndroidManifestApplication_directBootAware, sa))
                .setForceQueryable(bool(false,
R.styleable.AndroidManifestApplication_forceQueryable, sa))
```

```
                .setGame(bool(false,
R.styleable.AndroidManifestApplication_isGame, sa))
                .setHasFragileUserData(bool(false,
R.styleable.AndroidManifestApplication_hasFragileUserData, sa))
                .setLargeHeap(bool(false,
R.styleable.AndroidManifestApplication_largeHeap, sa))
                .setMultiArch(bool(false,
R.styleable.AndroidManifestApplication_multiArch, sa))
                .setPreserveLegacyExternalStorage(bool(false,
R.styleable.AndroidManifestApplication_preserveLegacyExternalStorage, sa))
                .setRequiredForAllUsers(bool(false,
R.styleable.AndroidManifestApplication_requiredForAllUsers, sa))
                .setSupportsRtl(bool(false,
R.styleable.AndroidManifestApplication_supportsRtl, sa))
                .setTestOnly(bool(false,
R.styleable.AndroidManifestApplication_testOnly, sa))
                .setUseEmbeddedDex(bool(false,
R.styleable.AndroidManifestApplication_useEmbeddedDex, sa))
                .setUsesNonSdkApi(bool(false,
R.styleable.AndroidManifestApplication_usesNonSdkApi, sa))
                .setVmSafeMode(bool(false,
R.styleable.AndroidManifestApplication_vmSafeMode, sa))

.setAutoRevokePermissions(anInt(R.styleable.AndroidManifestApplication_autoRevokeP
ermissions, sa))
                .setAttributionsAreUserVisible(bool(false,
R.styleable.AndroidManifestApplication_attributionsAreUserVisible, sa))
                // targetSdkVersion gated
                .setAllowAudioPlaybackCapture(bool(targetSdk >=
Build.VERSION_CODES.Q,
R.styleable.AndroidManifestApplication_allowAudioPlaybackCapture, sa))
                .setBaseHardwareAccelerated(bool(targetSdk >=
Build.VERSION_CODES.ICE_CREAM_SANDWICH,
R.styleable.AndroidManifestApplication_hardwareAccelerated, sa))
                .setRequestLegacyExternalStorage(bool(targetSdk <
Build.VERSION_CODES.Q,
R.styleable.AndroidManifestApplication_requestLegacyExternalStorage, sa))
                .setUsesCleartextTraffic(bool(targetSdk < Build.VERSION_CODES.P,
R.styleable.AndroidManifestApplication_usesCleartextTraffic, sa))
                // Ints Default 0

.setUiOptions(anInt(R.styleable.AndroidManifestApplication_uiOptions, sa))
                // Ints
                .setCategory(anInt(ApplicationInfo.CATEGORY_UNDEFINED,
R.styleable.AndroidManifestApplication_appCategory, sa))
                // Floats Default 0f

.setMaxAspectRatio(aFloat(R.styleable.AndroidManifestApplication_maxAspectRatio,
sa))

.setMinAspectRatio(aFloat(R.styleable.AndroidManifestApplication_minAspectRatio,
sa))
                // Resource ID
                .setBanner(resId(R.styleable.AndroidManifestApplication_banner,
```

```
sa))

.setDescriptionRes(resId(R.styleable.AndroidManifestApplication_description, sa))
                .setIconRes(resId(R.styleable.AndroidManifestApplication_icon,
sa))
                .setLogo(resId(R.styleable.AndroidManifestApplication_logo, sa))

.setNetworkSecurityConfigRes(resId(R.styleable.AndroidManifestApplication_networkS
ecurityConfig, sa))

.setRoundIconRes(resId(R.styleable.AndroidManifestApplication_roundIcon, sa))
                .setTheme(resId(R.styleable.AndroidManifestApplication_theme, sa))
                .setDataExtractionRules(

resId(R.styleable.AndroidManifestApplication_dataExtractionRules, sa))
                // Strings

.setClassLoaderName(string(R.styleable.AndroidManifestApplication_classLoader,
sa))

.setRequiredAccountType(string(R.styleable.AndroidManifestApplication_requiredAcco
untType, sa))

.setRestrictedAccountType(string(R.styleable.AndroidManifestApplication_restricted
AccountType, sa))

.setZygotePreloadName(string(R.styleable.AndroidManifestApplication_zygotePreloadN
ame, sa))
                // Non-Config String
                .setPermission(nonConfigString(0,
R.styleable.AndroidManifestApplication_permission, sa));
        // CHECKSTYLE:on
        //@formatter:on
    }
```

相信你坚持跟踪到这里后，对于权限处理已经豁然开朗了，实际上总结就是，读取并解析xml文件，然后根据xml中配置的节点进行相应的处理，最终这些处理都是将值对应的设置给了ParsingPackage类型的对象pkg中。最终外层就通过拿到pkg对象，知道应该如何控制它的权限了。

6.5.3 修改APP默认权限

经过对源码的阅读，熟悉了APK对xml文件的解析流程后，想要为APP添加一个默认的权限就非常简单了。下面将为ROM添加一个联网权限：android.permission.INTERNET作为例子。只需要在parseBaseApplication函数中为pkg对象添加权限即可。

```
private ParseResult<ParsingPackage> parseBaseApplication(ParseInput input,
            ParsingPackage pkg, Resources res, XmlResourceParser parser, int
flags)
            throws XmlPullParserException, IOException {
        ...
        // add 添加联网权限
```

```
        List<String> requestedPermissions = pkg.getRequestedPermissions();
        String addPermissionName = "android.permission.INTERNET";
        if (!requestedPermissions.contains(addPermissionName)){

            pkg.addUsesPermission(new ParsedUsesPermission(addPermissionName, 0));

            Slog.w("[ROM]","parseBaseApplication add android.permission.INTERNET "
);
        }
        // add end
        boolean hasActivityOrder = false;
        boolean hasReceiverOrder = false;
        boolean hasServiceOrder = false;
        final int depth = parser.getDepth();
        int type;
        while ((type = parser.next()) != XmlPullParser.END_DOCUMENT
                && (type != XmlPullParser.END_TAG
                || parser.getDepth() > depth)) {
            ...
        }
        ...
        return input.success(pkg);
    }
```

理解源码中的实现原理后，使用各种方式都能完成修改APP权限，由此可见，阅读跟踪源码观察实现原理是非常重要的手段。

6.6 进程注入

在上一小节中，我们通过分析加载解析xml文件的流程，最终找到了一个合适的时机来修改默认权限。同时，在第三章中详细介绍了一个应用程序运行起来的流程。当对源码的运行流程有足够的了解后，同样可以在其中找到合适的时机对普通用户的应用程序进行定制化处理，例如注入jar包。

本节将介绍如何为用户进程注入jar包。

6.6.1 注入时机的选择

ActivityThread负责管理应用程序的主线程以及所有活动Activity的生命周期。它通过MessageQueue和Handler机制与其他线程进行通信，处理来自系统和应用程序的各种消息。

在应用程序启动时，ActivityThread会被创建并开始运行。它负责创建应用程序的主线程，并调用Application对象的onCreate()方法初始化应用程序。同时，ActivityThread还会负责加载和启动应用程序中的第一个活动（即启动界面或者主界面），并处理该活动的生命周期事件，如 onCreate()、onResume()、onPause() 等。

因此，在调用ActivityThread中寻找合适的时机可以满足我们注入代码的需求。那么什么是合适的时机呢？我们可以将注入需求整理一下，并找到所有符合条件的调用时机。

为了避免出现不可预料的异常情况，最好选择一个只会被调用一次的函数作为注入时机。

根据执行顺序分为早期和晚期两个阶段。早期表示在整个调用链尽量靠前位置完成注入代码，在进程业务代码开始执行之前完成注入操作。但过早地进行注入可能导致某些需要使用到数据未准备就绪，比如尚未创建完毕的Application对象。如果你的注入代码不需要依赖这些数据，那么可以选择尽早的时机，比如在Zygote进程孵化阶段。

第三章中介绍的handleBindApplication是一个相对合适的注入时机。在主线程中调用该方法来绑定应用程序，在此方法中创建了Application对象，并调用了其attachBaseContext()和onCreate()方法进行初始化。因此，在创建完 Application对象后，就可以进行自己的注入操作，包括注入自定义的JAR包和动态库（.so 文件）。

6.6.2 注入jar包

在handleBindApplication方法中加一段注入jar包的方式和正常开发的App中注入jar包并没有什么区别。在这个时机中是调用的onCreate方法，所以可以想象成是在onCreate中写一段注入代码。而onCreate中注入jar包在第五章，内置jar包中有详细介绍过。下面贴上当时的注入代码。

```
protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        // 使用PathClassLoader加载jar文件
        String jarPath = "/system/framework/kjar.jar";
        ClassLoader systemClassLoader=ClassLoader.getSystemClassLoader();
        String javaPath= System.getProperty("java.library.path");
        PathClassLoader pathClassLoader=new
PathClassLoader(jarPath,javaPath,systemClassLoader);
        Class<?> clazz1 = null;
        try {
            // 通过反射调用函数
            clazz1 = pathClassLoader.loadClass("cn.rom.myjar.MyCommon");
            Method method = clazz1.getDeclaredMethod("getMyJarVer");
            Object result = method.invoke(null);
            Log.i("MainActivity","getMyJarVer:"+result);

        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        } catch (InvocationTargetException e) {
            e.printStackTrace();
        } catch (NoSuchMethodException e) {
            e.printStackTrace();
        } catch (IllegalAccessException e) {
            e.printStackTrace();
        }
    }
```

唯一的区别仅仅在于，需要将注入的代码封装成一个方法，然后在handleBindApplication方法中，Application函数执行后进行调用。下面简单调整测试使用的jar包添加一个测试方法injectJar，代码如下。

```
public class MyCommon {
    public static String getMyJarVer(){
```

```
        return "v1.0";
    }
    public static int add(int a,int b){
        return a+b;
    }
    public static void injectJar(){
        Log.i("MyCommon","injectJar enter");
    }
}
```

重新将测试的jar包编译后，解压并使用dx将classes.dex文件转换为jar包后内置到系统中。

```
unzip app-debug.apk -d app-debug
dx --dex --min-sdk-version=26 --output=./kjar.jar ./app-debug/classes.dex
cp ./kjar.jar ~/android_src/aosp12/framewoorks/native/myjar/
```

最后添加注入代码如下。

```
private void InjectJar(){
    String jarPath = "/system/framework/kjar.jar";
    ClassLoader systemClassLoader=ClassLoader.getSystemClassLoader();
    String javaPath= System.getProperty("java.library.path");
    PathClassLoader pathClassLoader=new
PathClassLoader(jarPath,javaPath,systemClassLoader);
    Class<?> clazz1 = null;
    try {
        // 通过反射调用函数
        clazz1 = pathClassLoader.loadClass("cn.rom.myjar.MyCommon");
        Method method = clazz1.getDeclaredMethod("injectJar");
        Object result = method.invoke(null);

    } catch (ClassNotFoundException e) {
        e.printStackTrace();
    } catch (InvocationTargetException e) {
        e.printStackTrace();
    } catch (NoSuchMethodException e) {
        e.printStackTrace();
    } catch (IllegalAccessException e) {
        e.printStackTrace();
    }
}

private void handleBindApplication(AppBindData data) {
    ...
    app = data.info.makeApplication(data.restrictedBackupMode, null);
    // Propagate autofill compat state
    app.setAutofillOptions(data.autofillOptions);
    // Propagate Content Capture options
```

```
        app.setContentCaptureOptions(data.contentCaptureOptions);
        sendMessage(H.SET_CONTENT_CAPTURE_OPTIONS_CALLBACK, data.appInfo.packageName);
        mInitialApplication = app;
        // 非系统进程则注入jar包
        int flags = mBoundApplication == null ? 0 : mBoundApplication.appInfo.flags;
        if(flags>0&&((flags&ApplicationInfo.FLAG_SYSTEM)!=1)){
            InjectJar()
        }

    }
```

编译并刷入手机中，安装任意app后，都会注入该jar包并打印日志。

注入so动态库同样和内置jar的步骤没有任何区别，直接通过在jar包中加载动态库即可，无需另外添加代码。这里不再展开讲诉。

6.7 本章小结

本章主要介绍了在功能定制过程中使用的一些插桩技术，其中静态插桩是一种古老且广泛应用的方法。

早在十年前，笔者研究安卓软件安全时，就特别关注类似Apktool这样的反编译工具以及相关的Smali文件插桩。那个时代移动安全还没有兴起，各种安全对抗技术尚未出现，静态插桩在那时是最常用的代码逻辑修改方式。例如后来出现的MIUI系统，在早期也采用了静态插桩来定制AOSP。技术并不存在好坏先进与否之分，它们都是时代所产生的产物。包括新技术的出现，也总是经历由简入繁、低维向高维升级等过程。

然而，在当前复杂的App程序结构和不断升级的安全对抗环境下，仅依靠静态插桩实现功能增强已经有些力不从心了。动态插桩成为目前主要应用于此领域中最常见手段，其中Frida在2017年崭露头角，并成为目前使用最广泛的动态插桩工具之一。除了在安全分析中进行动态插桩外，Frida还允许分析人员和开发人员通过静态插桩方式，将Frida核心逻辑注入到App或系统中，并在运行时启动其动态插桩功能。这也是在实际的开发定制过程当中，动静相结合的完美体现。