第七章 类加载和函数调用

在上一章中，我们学习了分析Android运行的执行流程，并找到合适的时机来插入业务逻辑代码，以实现特定功能。例如，在应用启动流程中，我们可以通过注入DEX文件或动态库文件来实现某些功能。通过native函数的注册流程，我们可以对静态注册和动态注册信息进行打桩输出。而通过解析AndroidManifest.xml文件的过程，则可以额外添加默认权限。

在本章中，将详细介绍Android源码中加载类的执行流程。了解Android中类加载机制以及函数调用流程是非常重要的基础知识。通过学习这些执行流程原理，在定制功能时能为我们提供更多方向和思路。

7.1 双亲委派机制

在Android系统中，应用程序运行在Dalvik或ART虚拟机上。当应用启动时，Android系统会根据应用程序包中的AndroidManifest.xml文件确定需要启动哪些组件，并在启动过程中加载所需的类。

Android中的类加载器遵循双亲委派模型。即每个类加载器在尝试加载一个类之前，都会先委托其父类加载器去加载该类。只有当父类加载器无法完成任务时，子类加载器才会尝试自己来进行加载。这个模型保证了不同的类只会被加载一次，并且保护了核心Java API不被恶意代码篡改。

在Android应用程序中，每个类都分配到一个特定的DEX文件（即Dalvik Executable）中。DEX文件包含该类所有方法和属性的字节码。当应用程序启动时，它的DEX文件将被加载到内存并由虚拟机执行其中的代码。

在函数调用流程中，当一个函数被调用时，虚拟机会保存当前线程状态，并跳转到被调函数入口地址开始执行该函数。虚拟机对函数指令进行执行，并维护执行过程所需数据结构（如栈帧）。当函数执行完毕后，虚拟机将结果返回给调用方并恢复之前保存的线程状态。

深入学习Android的类加载机制和函数执行调用流程可以更好地理解应用程序的运行机制。

在Android中，类通常保存在DEX文件中，而ClassLoader则负责加载DEX文件。每个应用程序包（APK）都包含一个或多个DEX文件，这些DEX文件包含应用程序的所有类信息。当需要使用某个类时，ClassLoader会从相应的DEX文件中加载该类，并将其转换为可执行的Java类。因此，ClassLoader和DEX密切相关，ClassLoader是DEX文件的载体和管理者。

Android 中的 ClassLoader 类型分为两种：

1. 系统类加载器。系统类加载器主要包括BootClassLoader、PathClassLoader和DexClassLoader。
2. 自定义加载器。

一些常见的加载器的用途如下：

1. `BootClassLoader`：位于 `ClassLoader` 层次结构中的最顶层。负责加载系统级别的类，如 `Java` 核心库和一些基础库。
2. `PathClassLoader`：从应用程序的 `APK` 文件中加载类和资源。继承自`BaseDexClassLoader`类，它能够加载已经被优化的 `Dex` 文件和未经过优化的 `Dex` 文件。`PathClassLoader` 主要用于加载已经打包在 `APK` 文件中的代码和资源。
3. `DexClassLoader`：从 `.dex` 或 `.odex` 文件中加载类。继承自`BaseDexClassLoader`类，它支持动态加载 `Dex` 文件，并且可以在运行时进行优化操作。`DexClassLoader` 主要用于加载未安装的 `APK` 文件中

的代码。

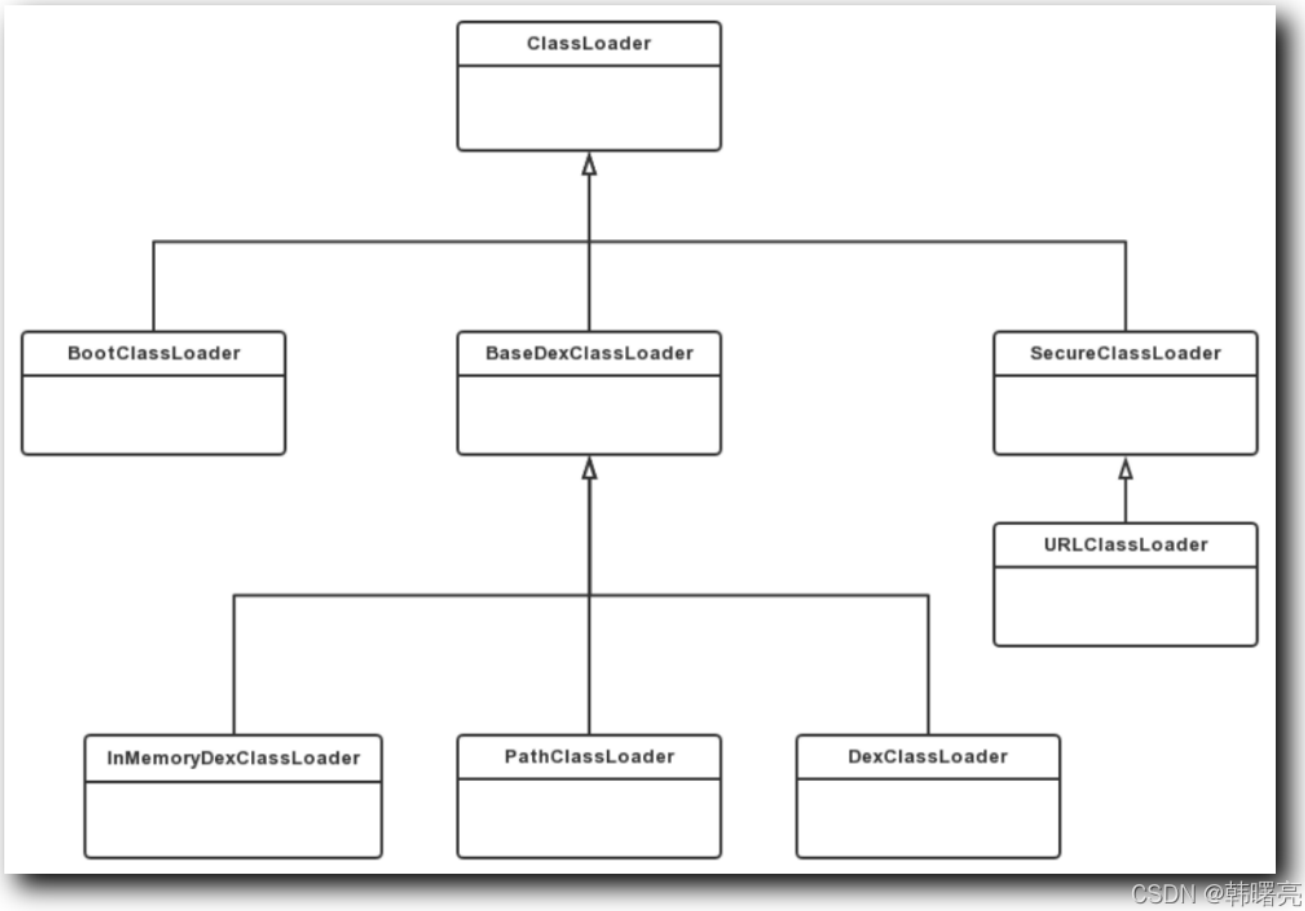
4. `InMemoryDexClassLoader`：用于从内存中加载已经存在于内存中的`dex`文件。继承自`BaseDexClassLoader`，并且可以处理多个`dex`文件。`InMemoryDexClassLoader` 可以在运行时动态加载 `dex` 文件，并且不需要将文件保存到磁盘上，从而提高应用程序的性能。

5. `BaseDexClassLoader`：`DexClassLoader`、`InMemoryDexClassLoader` 和 `PathClassLoader` 的基类，封装了加载 `dex` 文件的基本逻辑，包括创建 `DexPathList` 对象、打开 `dex` 文件、查找类等操作。`BaseDexClassLoader` 实现了双亲委派模型，即在自身无法加载类时，会委派给父类加载器进行查找。`BaseDexClassLoader` 还支持多个 `dex` 文件的加载，并且可以在运行时进行优化操作。

6. SecureClassLoader：继承自ClassLoader抽象类，该类主要实现了一些权限相关的功能。

7. URLClassLoader：SecureClassLoader的子类，其可以使用url路径加载JAR文件中的类。

整个类加载器的继承结构如下图所示：



类加载器采用了双亲委派机制（Parent Delegation Model），这是一种经典的Java类加载机制。

双亲委派机制是指当一个类加载器收到请求去加载一个类时，它并不会自己去加载，而是把这个任务委托给父类加载器去完成。如果父类加载器还存在父类加载器，这个请求就会向上递归，直到达到最顶层的BootClassLoader为止。也就是说，最先调用加载的ClassLoader是最顶层的，最后尝试加载的是当前的ClassLoader。

采用双亲委派机制可以有效地避免类的重复加载，并保证核心API的安全性。具体表现为：

- 在类加载时，首先从当前加载器的缓存中查找是否已经加在了该类，如果已经加在，则直接返回；

- 如果没有在缓存中找到该累，则将加在任务委派给父累加，在者完成；
- 父累加如果也没有找道该累，则将会递归向上委派, 直道BootClassLoader;
- BootCLassLoader无法代理添加和发生错误之前所做过得努力, 则会让子类加载器自行加载。

7.2 类的加载流程

在Android中，ClassLoader类是双亲委派机制的主要实现者。该类提供了findClass和loadClass方法，其中findClass是ClassLoader的抽象方法，需要由子类实现。接下来将跟踪源码实现，详细了解ClassLoader是如何进行类加载流程的。

在前文中曾经介绍过如何使用DexClassLoader加载一个类，并调用其中的函数，下面是当时的加载样例代码。

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

        String dexPath = "/system/framework/kjar.jar";
        String dexOutputDir = getApplicationInfo().dataDir;
        ClassLoader classLoader = new DexClassLoader(dexPath, dexOutputDir, null,
                getClass().getClassLoader());

        Class<?> clazz2 = null;
        try {
            clazz2 = classLoader.loadClass("cn.rom.myjar.MyCommon");
            Method addMethod = clazz2.getDeclaredMethod("add",
int.class,int.class);
            Object result = addMethod.invoke(null, 12,25);
            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();
        }
}
```

ClassLoader 加载类时，loadClass 和 findClass都可以完成对类的加载工作，它们在加载类时有着不同的作用和执行流程。

首先看看loadClass的特征，它的方法签名如下。

```
protected Class<?> loadClass( final String class_name, final boolean resolve )
throws ClassNotFoundException;
```

其中name 参数表示要加载的类的全名；resolve 参数表示是否需要在加载完成后进行链接操作。如果 resolve 参数为 true，则会尝试在加载完成后对该类进行链接操作，包括验证、准备和解析等步骤。如果 resolve 参数为 false，则不会进行链接操作。

在执行loadClass方法时，ClassLoader 会先检查自身是否已经加载过该类，如果已经加载过，则直接返回该类的 Class 对象。如果没有加载过，则将任务委托给父类加载器进行处理，如果父类加载器无法加载该类，则再次调用自身的 findClass 方法进行加载。如果 findClass 方法仍然无法找到该类，则抛出 ClassNotFoundException 异常。

接下来再了解下findClass 方法，它 是 BaseClassLoader 类中定义的一个抽象方法，用于在特定的数据源（如文件、内存等）中查找指定名称的类，并返回对应的 Class 对象。下面是方法签名。

```
protected abstract Class<?> findClass(String name) throws ClassNotFoundException;
```

与 loadClass 不同，findClass 方法并不会先委派给父类加载器进行处理，而是直接在当前 ClassLoader 中进行查找。如果能够找到指定的类，则通过 defineClass 方法将其转换成Class对象，并返回该对象；否则，抛出 ClassNotFoundException 异常。

明白了两者的区别后，接下来开始跟踪源码，了解在AOSP具体是如何加载类的。首先找到DexClassLoader中loadClass的实现代码。

```
public class DexClassLoader extends BaseDexClassLoader {
    public DexClassLoader(String dexPath, String optimizedDirectory,
            String librarySearchPath, ClassLoader parent) {
        super(dexPath, null, librarySearchPath, parent);
    }
}
```

发现内部并没有任何代码，说明该实现来自于父类中，接着来查看父类BaseDexClassLoader

```
public class BaseDexClassLoader extends ClassLoader {
    public BaseDexClassLoader(String dexPath, File optimizedDirectory, String
librarySearchPath, ClassLoader parent) {
        ...
    }
    protected Class<?> findClass(String name) throws ClassNotFoundException {
        ...
    }

    protected URL findResource(String name) {
        ...
    }

    protected Enumeration<URL> findResources(String name) {
        ...
    }
```

```
    public String findLibrary(String name) {
        ...
    }

    protected synchronized Package getPackage(String name) {
        ...
    }

    public String toString() {
        ...
    }
}
```

同样没有找到loadClass的实现，继续看它的父类ClassLoader的实现。

```
public abstract class ClassLoader {
    ...

    // 调用了另外一个重载，resolve参数不传的情况默认为false
    public Class<?> loadClass(String name) throws ClassNotFoundException {
        return loadClass(name, false);
    }

    protected Class<?> loadClass(String name, boolean resolve)
        throws ClassNotFoundException
    {
            // 尝试在已经加载过的里面查找
            Class<?> c = findLoadedClass(name);
            if (c == null) {
                try {
                    // 有父类的情况，就让父类来加载
                    if (parent != null) {
                        c = parent.loadClass(name, false);
                    } else {
                        // 到达父类顶端后，则使用这个函数查找，通常来查找引导类和扩展类
                        c = findBootstrapClassOrNull(name);
                    }
                } catch (ClassNotFoundException e) {
                    // ClassNotFoundException thrown if class not found
                    // from the non-null parent class loader
                }

                if (c == null) {
                    // 父类没有找到的情况，再通过findClass查找
                    c = findClass(name);
                }
            }
            return c;
    }
    ...
    protected Class<?> findClass(String name) throws ClassNotFoundException {
```

```
            throw new ClassNotFoundException(name);
        }
    }
```

通过这里的代码，能够很清晰的看到前文中ClassLoader的双亲委派机制，接着继续跟踪findClass分析当前ClassLoader是如何加载类的，由于ClassLoader是一个抽象类，而findClass在该类中并未实现具体代码，所以该方法是在子类中实现，上面在BaseDexClassLoader的类中，就已经看到的findClass的函数，下面是具体实现。

```
public class BaseDexClassLoader extends ClassLoader {
    ...
    private final DexPathList pathList;
    ...
    protected Class<?> findClass(String name) throws ClassNotFoundException {
        // 首先检查当前ClassLoader是否有共享库，如果有则遍历每个共享库的ClassLoader去尝试加载该类
        if (sharedLibraryLoaders != null) {
            for (ClassLoader loader : sharedLibraryLoaders) {
                try {
                    return loader.loadClass(name);
                } catch (ClassNotFoundException ignored) {
                }
            }
        }
        List<Throwable> suppressedExceptions = new ArrayList<Throwable>();
        // 当前ClassLoader操作的dex文件中查找该类
        Class c = pathList.findClass(name, suppressedExceptions);
        if (c == null) {
            ClassNotFoundException cnfe = new ClassNotFoundException(
                    "Didn't find class \"" + name + "\" on path: " + pathList);
            for (Throwable t : suppressedExceptions) {
                cnfe.addSuppressed(t);
            }
            throw cnfe;
        }
        return c;
    }
    ...
}
```

pathList是一个DexPathList对象，表示当前ClassLoader所管理的一组dex文件的路径列表。findClass()方法通过调用DexPathList.findClass()方法来查找指定名称的类。继续跟进查看。

```
public final class DexPathList {
    ...
    private Element[] dexElements;
    ...
    public Class<?> findClass(String name, List<Throwable> suppressed) {
```

```
        for (Element element : dexElements) {
            Class<?> clazz = element.findClass(name, definingContext, suppressed);
            if (clazz != null) {
                return clazz;
            }
        }

        if (dexElementsSuppressedExceptions != null) {
            suppressed.addAll(Arrays.asList(dexElementsSuppressedExceptions));
        }
        return null;
    }
    ...
}
```

dexElements的数组存放着所有已经加载的dex文件中的类信息。具体来说，每个dex文件都被解析为一个DexFile对象，而dexElements数组中的每个元素实际上就是一个Element对象，代表了一个dex文件和其中包含的类信息。这些Element对象按照优先级顺序排列，以便ClassLoader可以根据它们的顺序来查找类定义。继续查看Element的findClass方法实现。

```
static class Element {
        ...
        // 管理着一个dex文件
        private final DexFile dexFile;

        ...
        private String getDexPath() {
            if (path != null) {
                return path.isDirectory() ? null : path.getAbsolutePath();
            } else if (dexFile != null) {
                // DexFile.getName() returns the path of the dex file.
                return dexFile.getName();
            }
            return null;
        }

        @Override
        public String toString() {
            if (dexFile == null) {
              return (pathIsDirectory ? "directory \"" : "zip file \"") + path +
"\"";
            } else if (path == null) {
              return "dex file \"" + dexFile + "\"";
            } else {
              return "zip file \"" + path + "\"";
            }
        }

        public Class<?> findClass(String name, ClassLoader definingContext,
                List<Throwable> suppressed) {
            return dexFile != null ? dexFile.loadClassBinaryName(name,
```

```
    definingContext, suppressed)
                        : null;
            }

            ...
        }
```

可以看到这里实际就是管理一个对应的DexFile对象，该对象关联着一个对应的dex文件，这里通过调用DexFile对象的loadClassBinaryName去加载这个类，继续跟踪它的实现。

```
public final class DexFile {
    ...
    public Class loadClassBinaryName(String name, ClassLoader loader,
List<Throwable> suppressed) {
        return defineClass(name, loader, mCookie, this, suppressed);
    }
    ...

    private static Class defineClass(String name, ClassLoader loader, Object
cookie,
                                     DexFile dexFile, List<Throwable> suppressed)
{
        Class result = null;
        try {
            result = defineClassNative(name, loader, cookie, dexFile);
        } catch (NoClassDefFoundError e) {
            if (suppressed != null) {
                suppressed.add(e);
            }
        } catch (ClassNotFoundException e) {
            if (suppressed != null) {
                suppressed.add(e);
            }
        }
        return result;
    }
    ...
    private static native Class defineClassNative(String name, ClassLoader loader,
Object cookie,
                                                  DexFile dexFile)
            throws ClassNotFoundException, NoClassDefFoundError;
}
```

这里看到经过几层调用后，进入了native实现了，根据AOSP中native注册的命名规则，直接搜索DexFile_defineClassNative找到对应的实现代码如下。

```
static jclass DexFile_defineClassNative(JNIEnv* env,
                                        jclass,
                                        jstring javaName,
```

```
                                          jobject javaLoader,
                                          jobject cookie,
                                          jobject dexFile) {
  std::vector<const DexFile*> dex_files;
  const OatFile* oat_file;
  // cookie转换成一组c++中的DexFile对象以及OatFile
  if (!ConvertJavaArrayToDexFiles(env, cookie, /*out*/ dex_files, /*out*/
oat_file)) {
    VLOG(class_linker) << "Failed to find dex_file";
    DCHECK(env->ExceptionCheck());
    return nullptr;
  }
  ...
  // 将类名转换为c++的string存放在了descriptor中
  // 这里会将java中的类描述符转换为c++使用的类描述符，例如类中的.转换为\
  const std::string descriptor(DotToDescriptor(class_name.c_str()));
  const size_t hash(ComputeModifiedUtf8Hash(descriptor.c_str()));
  for (auto& dex_file : dex_files) {
    // 根据类描述符找到对应的类
    const dex::ClassDef* dex_class_def =
        OatDexFile::FindClassDef(*dex_file, descriptor.c_str(), hash);
    if (dex_class_def != nullptr) {
      ScopedObjectAccess soa(env);
      ClassLinker* class_linker = Runtime::Current()->GetClassLinker();
      ...
      // 使用类加载器和 DEX 文件定义一个新的 Java 类，并返回一个描述该类的 Class 对象指
针
      ObjPtr<mirror::Class> result = class_linker->DefineClass(soa.Self(),
                                                               descriptor.c_str(),
                                                               hash,
                                                               class_loader,
                                                               *dex_file,
                                                               *dex_class_def);
      // 将DexFile插入到ClassLoader中。
      class_linker->InsertDexFileInToClassLoader(soa.Decode<mirror::Object>
(dexFile),
                                                 class_loader.Get());
      if (result != nullptr) {
        VLOG(class_linker) << "DexFile_defineClassNative returning " << result
                           << " for " << class_name.c_str();
        return soa.AddLocalReference<jclass>(result);
      }
    }
  }
  VLOG(class_linker) << "Failed to find dex_class_def " << class_name.c_str();
  return nullptr;
}
```

代码中看到cookie中能拿到所有DexFile，最终的Class对象是由DefineClass方法定义后返回的。继续看其实现过程。

```
ObjPtr<mirror::Class> ClassLinker::DefineClass(Thread* self,
                                               const char* descriptor,
                                               size_t hash,
                                               Handle<mirror::ClassLoader>
class_loader,
                                               const DexFile& dex_file,
                                               const dex::ClassDef& dex_class_def)
{
  ...
  DexFile const* new_dex_file = nullptr;
  dex::ClassDef const* new_class_def = nullptr;
  // 类被加载前的预处理
  Runtime::Current()->GetRuntimeCallbacks()->ClassPreDefine(descriptor,
                                                            klass,
                                                            class_loader,
                                                            dex_file,
                                                            dex_class_def,
                                                            &new_dex_file,
                                                            &new_class_def);
  // 将dex文件加载到内存中
  ObjPtr<mirror::DexCache> dex_cache = RegisterDexFile(*new_dex_file,
class_loader.Get());
  if (dex_cache == nullptr) {
    self->AssertPendingException();
    return sdc.Finish(nullptr);
  }
  klass->SetDexCache(dex_cache);

  // 初始化类
  SetupClass(*new_dex_file, *new_class_def, klass, class_loader.Get());
  ...

  // 向类表中插入类对象
  ObjPtr<mirror::Class> existing = InsertClass(descriptor, klass.Get(), hash);
  ...

  // 加载并初始化类，在必要时创建新的类对象
  LoadClass(self, *new_dex_file, *new_class_def, klass);
  ...

  MutableHandle<mirror::Class> h_new_class = hs.NewHandle<mirror::Class>(nullptr);

  // 链接类及其相关信息
  if (!LinkClass(self, descriptor, klass, interfaces, &h_new_class)) {
    // Linking failed.
    if (!klass->IsErroneous()) {
      mirror::Class::SetStatus(klass, ClassStatus::kErrorUnresolved, self);
    }
    return sdc.Finish(nullptr);
  }
  return sdc.Finish(h_new_class);
}
```

`ClassPreDefine`是一个回调函数，它在类被加载之前被调用，用于进行一些预处理工作。具体来说，`ClassPreDefin`会被调用以执行以下任务：

- 对新定义的类进行验证和解析，以确保类结构的正确性。
- 为新定义的类分配内存空间，并构造新对象的实例。
- 设置类的访问控制权限并更新关联的缓存信息。

`RegisterDexFile`用于注册 `DEX` 文件。该函数负责将 `DEX` 文件加载到内存中，并将其中包含的类和相关信息注册到运行时环境中，以供后续的程序使用。该函数的主要负责：

- 将 `DEX` 文件加载到内存中，并为其分配一段连续的内存空间。
- 在运行时环境中创建`mirror::DexFile`对象，该对象包含了 `DEX`文件的元数据信息，例如文件名、`MD5`哈希值等。
- 为`DEX`文件中包含的每个类创建相应的 `mirror::Class` 对象，并将其添加到类表中进行管理。
- 为新创建的 `mirror::Class` 对象设置其访问权限和其他属性，例如类标志、字段、方法等。
- 创建并返回一个 `mirror::DexCache` 对象，该对象表示已注册的`DEX`文件的缓存信息。

`SetupClass` 函数用于初始化类。该函数的主要作用：

- 解析类定义，并为其分配内存空间。
- 为新创建的类对象设置相关信息，例如类名、超类、接口信息等。
- 设置类对象的访问修饰符和标志。
- 将类对象添加到运行时环境中进行管理。
- 在必要的情况下，执行与类加载生命周期有关的回调函数。

`InsertClass`函数用于向类表中插入新的类对象，并确保在插入之前对其进行必要的验证和初始化工作。该函数的主要作用：

- 根据类描述符和哈希值查找类表中是否已经存在相同的类对象。
- 如果已经存在相同的类对象，则返回其指针，否则将新的类对象插入到类表中，并返回其指针。
- 在插入新的类对象之前，会先进行一些验证工作，例如检查类的访问权限，以及确保类的结构和超类的继承关系正确等。
- 在需要时，执行与类加载生命周期有关的回调函数。

`LoadClass` 函数用于加载并初始化类。并将其插入到类表中进行管理。主要作用：

- 根据类描述符查找类表中是否已经存在相同的类对象，如存在则直接返回其指针。
- 如果类表中不存在相同的类对象，则先使用 `SetupClass()` 函数创建新的类对象，并将其插入到类表中。此处调用了 `InsertClass()` 函数。

- 加载并初始化类的超类及接口信息，以确保类的继承关系正确。
- 执行与类加载生命周期有关的回调函数。

`LinkClass` 函数是在用于链接类，该函数会返回一个新的类对象指针，以供调用者使用。主要作用：

- 链接类的超类，并执行与超类有关的初始化工作。
- 链接类实现的接口，并执行与接口有关的初始化工作。
- 链接类的字段，并执行与字段有关的初始化工作。
- 链接类的方法，并执行与方法有关的初始化工作。
- 在必要时创建新的类对象，并将其返回给调用者。

将加载类的过程中几个关键的步骤搞清楚后，继续深入查看`LoadClass`是如何实现的，重点关注最后一个参数`kclass`做了些什么。

```
void ClassLinker::LoadClass(Thread* self,
                            const DexFile& dex_file,
                            const dex::ClassDef& dex_class_def,
                            Handle<mirror::Class> klass) {
  ...
  Runtime* const runtime = Runtime::Current();
  {
    ...
    // 获取类加载器的线性内存分配器
    LinearAlloc* const allocator = GetAllocatorForClassLoader(klass-
>GetClassLoader());
    // 为类中的静态字段分配内存空间
    LengthPrefixedArray<ArtField>* sfields = AllocArtFieldArray(self,
                                                                allocator,

accessor.NumStaticFields());
    // 为类中的实例字段分配内存空间
    LengthPrefixedArray<ArtField>* ifields = AllocArtFieldArray(self,
                                                                allocator,

accessor.NumInstanceFields());

    ...

    // 设置类的方法列表指针
    klass->SetMethodsPtr(
        AllocArtMethodArray(self, allocator, accessor.NumMethods()),
        accessor.NumDirectMethods(),
        accessor.NumVirtualMethods());
    size_t class_def_method_index = 0;
    uint32_t last_dex_method_index = dex::kDexNoIndex;
    size_t last_class_def_method_index = 0;

    // 遍历类的所有方法和字段
```

```
    accessor.VisitFieldsAndMethods([&](
        const ClassAccessor::Field& field) REQUIRES_SHARED(Locks::mutator_lock_) {
          ...
          // 遍历所有字段，由last_static_field_idx判断是否正在处理的是静态字段
          if (num_sfields == 0 || LIKELY(field_idx > last_static_field_idx)) {
            // 加载字段信息
            LoadField(field, klass, &sfields->At(num_sfields));
            ++num_sfields;
            last_static_field_idx = field_idx;
          }
        }, [&](const ClassAccessor::Field& field)
REQUIRES_SHARED(Locks::mutator_lock_) {
          ...
          // 加载实例字段信息
          if (num_ifields == 0 || LIKELY(field_idx > last_instance_field_idx)) {
            LoadField(field, klass, &ifields->At(num_ifields));
            ++num_ifields;
            last_instance_field_idx = field_idx;
          }
        }, [&](const ClassAccessor::Method& method)
REQUIRES_SHARED(Locks::mutator_lock_) {
          // 获取实例方法
          ArtMethod* art_method = klass-
>GetDirectMethodUnchecked(class_def_method_index,
              image_pointer_size_);
          // 将dex_file参数中指向Java方法字节码的指针(method)解析为机器码，并将它存储到
art_method参数对应的内存区域中，完成对Java方法实现代码的加载
          LoadMethod(dex_file, method, klass, art_method);
          // 将art_method参数对应的实现代码链接到oat_class_ptr参数对应的oat文件中
          LinkCode(this, art_method, oat_class_ptr, class_def_method_index);
          ...
        }, [&](const ClassAccessor::Method& method)
REQUIRES_SHARED(Locks::mutator_lock_) {

          // 和上面差不多的，不过这里处理的是虚方法
          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;
        });

    ...
    // 将加载好的字段保存到kclass
    klass->SetSFieldsPtr(sfields);
    DCHECK_EQ(klass->NumStaticFields(), num_sfields);
    klass->SetIFieldsPtr(ifields);
    DCHECK_EQ(klass->NumInstanceFields(), num_ifields);
  }
  // Ensure that the card is marked so that remembered sets pick up native roots.
  WriteBarrier::ForEveryFieldWrite(klass.Get());
  self->AllowThreadSuspension();
```

```
}
```

然后再了解一下LoadField和LoadMethod是如何加载的。

```
void ClassLinker::LoadField(const ClassAccessor::Field& field,
                            Handle<mirror::Class> klass,
                            ArtField* dst) {
  // 可以看到实际就是将值填充给了dst
  const uint32_t field_idx = field.GetIndex();
  dst->SetDexFieldIndex(field_idx);
  dst->SetDeclaringClass(klass.Get());
  dst->SetAccessFlags(field.GetAccessFlags() |
hiddenapi::CreateRuntimeFlags(field));
}


void ClassLinker::LoadMethod(const DexFile& dex_file,
                             const ClassAccessor::Method& method,
                             Handle<mirror::Class> klass,
                             ArtMethod* dst) {
  const uint32_t dex_method_idx = method.GetIndex();
  const dex::MethodId& method_id = dex_file.GetMethodId(dex_method_idx);
  const char* method_name = dex_file.StringDataByIdx(method_id.name_idx_);

  ScopedAssertNoThreadSuspension ants("LoadMethod");
  dst->SetDexMethodIndex(dex_method_idx);
  dst->SetDeclaringClass(klass.Get());

  ...
  // 如果加载的是finalize方法
  if (UNLIKELY(strcmp("finalize", method_name) == 0)) {
    ...
  } else if (method_name[0] == '<') {
    // 处理构造函数
    bool is_init = (strcmp("<init>", method_name) == 0);
    bool is_clinit = !is_init && (strcmp("<clinit>", method_name) == 0);
    if (UNLIKELY(!is_init && !is_clinit)) {
      LOG(WARNING) << "Unexpected '<' at start of method name " << method_name;
    } else {
      if (UNLIKELY((access_flags & kAccConstructor) == 0)) {
        LOG(WARNING) << method_name << " didn't have expected constructor access
flag in class "
            << klass->PrettyDescriptor() << " in dex file " <<
dex_file.GetLocation();
        // access_flags存储了Java方法的访问标志，如public、private、static等。
kAccConstructor是一个常量，表示Java构造函数的访问标志
        access_flags |= kAccConstructor;
      }
    }
  }
  // 判断是否为native函数
```

```
  if (UNLIKELY((access_flags & kAccNative) != 0u)) {
    // Check if the native method is annotated with @FastNative or
@CriticalNative.
    access_flags |= annotations::GetNativeMethodAnnotationAccessFlags(
        dex_file, dst->GetClassDef(), dex_method_idx);
  }
  // 设置该方法的访问标志
  dst->SetAccessFlags(access_flags);

  // 判断是否为接口类的抽象方法
  if (klass->IsInterface() && dst->IsAbstract()) {
    // 计算并设置抽象方法的IMT索引。IMT(Interface Method Table)是一个虚拟表，用于存储
接口类中的所有方法索引。
    dst->CalculateAndSetImtIndex();
  }
  // 这个java方法是否有可执行代码，也就是java字节码，方法的具体执行指令集
  if (dst->HasCodeItem()) {
    DCHECK_NE(method.GetCodeItemOffset(), 0u);
    // 根据当前是否采用AOT编译器来进行不同的方式填充可执行代码。
    if (Runtime::Current()->IsAotCompiler()) {
      dst->SetDataPtrSize(reinterpret_cast32<void*>(method.GetCodeItemOffset()),
image_pointer_size_);
    } else {
      dst->SetCodeItem(dst->GetDexFile()-
>GetCodeItem(method.GetCodeItemOffset()));
    }
  } else {
    dst->SetDataPtrSize(nullptr, image_pointer_size_);
    DCHECK_EQ(method.GetCodeItemOffset(), 0u);
  }

  // 检查该方法的参数类型和返回值类型是否符合要求
  const char* shorty = dst->GetShorty();
  bool all_parameters_are_reference = true;
  bool all_parameters_are_reference_or_int = true;
  bool return_type_is_fp = (shorty[0] == 'F' || shorty[0] == 'D');

  for (size_t i = 1, e = strlen(shorty); i < e; ++i) {
    if (shorty[i] != 'L') {
      all_parameters_are_reference = false;
      if (shorty[i] == 'F' || shorty[i] == 'D' || shorty[i] == 'J') {
        all_parameters_are_reference_or_int = false;
        break;
      }
    }
  }
  // Java方法设置是否启用Nterp快速路径，如果该函数非native的，并且参数全部为引用类型，则
设置该方法的entry_point_from_interpreter_为Nterp快速路径
  if (!dst->IsNative() && all_parameters_are_reference) {
    dst->SetNterpEntryPointFastPathFlag();
  }
  // 返回值类型非浮点型，并且所有参数类型都是引用类型或整型，则设置该方法的
invocation_count_为Nterp快速路径
  if (!return_type_is_fp && all_parameters_are_reference_or_int) {
```

```
      dst->SetNterpInvokeFastPathFlag();
    }
  }
```

finalize是Java中的一个方法，定义在Object类中，用于执行垃圾回收前的资源清理工作。当某个对象不再被引用时，垃圾回收器会调用该对象的finalize方法来完成一些特定的清理操作，如释放非托管资源等。

Nterp快速路径 (Nterp Fast Path) 是ART虚拟机的一种执行模式，可以在不进行线程切换的情况下快速执行Java方法。具体来说，Nterp快速路径使用一种特殊的、基于指令计数器的执行模式来处理Java方法，以实现更高效的性能。

Nterp快速路径的作用是提高Java方法的执行速度和效率，特别是在热点代码部分，可以获得更高的吞吐量和更低的延迟。另外，由于采用了一些特殊的优化技术，如参数传递方式改变、返回值处理流程优化等，Nterp快速路径还可以减少JNI开销，从而提升整个应用程序的性能表现。

在前文介绍native的动态注册时，曾经简单的讲解LinkCode，这里再次对这个重点函数进行详细的了解。

```
bool ClassLinker::ShouldUseInterpreterEntrypoint(ArtMethod* method, const void*
quick_code) {
  ...
  if (quick_code == nullptr) {
    return true;
  }
  ..
  return false;
}

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);
    // 获取一个方法的快速代码（Quick Code），用于设置该方法的入口点地址
    quick_code = oat_method.GetQuickCode();
  }

  // 如果有方法的快速代码，否则使用解释器执行，在下一节的函数调用中会详细讲到
  bool enter_interpreter = class_linker->ShouldUseInterpreterEntrypoint(method,
quick_code);

  if (quick_code == nullptr) {
    // 设置一个方法的入口点位置，可以是编译成机器码的快速执行入口、解释器入口，或者
native函数的入口地址
    method->SetEntryPointFromQuickCompiledCode(
        method->IsNative() ? GetQuickGenericJniStub() :
```

```
GetQuickToInterpreterBridge());
  } else if (enter_interpreter) {
    // 设置解释器入口为该方法的入口点位置
    method->SetEntryPointFromQuickCompiledCode(GetQuickToInterpreterBridge());
  } else if (NeedsClinitCheckBeforeCall(method)) {
    DCHECK(!method->GetDeclaringClass()->IsVisiblyInitialized());
    method->SetEntryPointFromQuickCompiledCode(GetQuickResolutionStub());
  } else {
    // 已经编译好的机器码所在的快速执行入口
    method->SetEntryPointFromQuickCompiledCode(quick_code);
  }

  // 给native设置入口地址的，在第六章动态注册中讲到。
  if (method->IsNative()) {
    ...
  }
}
```

快速代码是指一种优化后的本地机器代码，它可以直接执行Java字节码对应的指令，从而实现更快的函数调用和执行。快速代码通常是通过即时编译器（JIT）或预编译技术生成的，并保存在Oat文件中。在运行时，如果一个方法已经被编译为快速代码，则LinkCode函数可以直接使用Oat文件中的方法描述符获取快速代码的地址，并将其设置为该方法的入口点地址。

接下来看看设置的解释器入口是什么，跟踪方法GetQuickToInterpreterBridge的实现。

```
static inline const void* GetQuickToInterpreterBridge() {
  return reinterpret_cast<const void*>(art_quick_to_interpreter_bridge);
}
```

这里和native动态注册分析时看到入口设置非常类似，GetQuickToInterpreterBridge是一个静态内联函数，它将全局变量art_quick_to_interpreter_bridge的地址强制转换为const void*类型，然后返回该地址。art_quick_to_interpreter_bridge是一个指向解释器入口点的函数指针，它在链接器启动时被初始化，是由汇编进行实现。

```
ENTRY art_quick_to_interpreter_bridge
    SETUP_SAVE_REFS_AND_ARGS_FRAME         // Set up frame and save arguments.

    //  x0 will contain mirror::ArtMethod* method.
    mov x1, xSELF                          // How to get Thread::Current() ???
    mov x2, sp

    // uint64_t artQuickToInterpreterBridge(mirror::ArtMethod* method, Thread*
self,
    //                                      mirror::ArtMethod** sp)
    bl   artQuickToInterpreterBridge

    RESTORE_SAVE_REFS_AND_ARGS_FRAME
    REFRESH_MARKING_REGISTER
```

```
    fmov d0, x0

    RETURN_OR_DELIVER_PENDING_EXCEPTION
END art_quick_to_interpreter_bridge
```

查看汇编代码时，可以注意到关键是使用bl指令调用artQuickToInterpreterBridge函数。这个函数就是解释器的入口函数。

解释器（Interpreter）是一种Java字节码执行引擎，它能够直接解释和执行Java字节码指令。与预编译的本地机器代码不同，解释器以Java字节码为基础，通过逐条解释执行来完成函数的执行过程。

当应用程序需要执行一个Java方法时，链接器会将该方法的字节码读入内存，并利用解释器逐条指令执行。解释器会根据Java字节码类型进行相应的操作，包括创建对象、读取/写入局部变量和操作数栈、跳转操作等。同时，解释器还会处理异常、垃圾回收、线程同步等方面的操作，从而保证Java程序的正确性和稳定性。

尽管解释器的执行速度比本地机器代码要慢一些，但它具有许多优点。例如，解释器可以实现更快的程序启动时间、更小的内存占用和更好的灵活性；同时，它还可以避免因硬件平台差异、编译器优化等问题导致代码执行异常和安全隐患。

当一个方法第一次被调用时，在进行初步解释和执行之后，解释器会生成相应的Profile数据。后续调用将根据Profile数据决定是否使用JIT编译器或AOT编译器进行优化。这种混合的执行方式可以有效地平衡运行效率和内存开销之间的关系，提高Java程序的整体性能和响应速度。

当类加载完成后，对应的类数据将会存储在相应的DexFile中。在后续使用中，可以通过DexFile来访问类中的成员和函数。下面简单了解一下DexFile结构。

```
class DexFile {
 public:
  // dex文件魔数的字节数。
  static constexpr size_t kDexMagicSize = 4;
  // dex文件版本号的字节数。
  static constexpr size_t kDexVersionLen = 4;

  static constexpr uint32_t kClassDefinitionOrderEnforcedVersion = 37;
  // SHA-1消息摘要的长度
  static constexpr size_t kSha1DigestSize = 20;
  // dex文件的大小端标志
  static constexpr uint32_t kDexEndianConstant = 0x12345678;

  // 无效索引的值
  static constexpr uint16_t kDexNoIndex16 = 0xFFFF;
  static constexpr uint32_t kDexNoIndex32 = 0xFFFFFFFF;

  // 表示dex文件头结构
  struct Header {
    uint8_t magic_[8] = {};      // 魔数
    uint32_t checksum_ = 0;     // 校验和
    uint8_t signature_[kSha1DigestSize] = {};    // SHA-1签名
    uint32_t file_size_ = 0;  // 文件总大小
    uint32_t header_size_ = 0;  // 偏移量到下一部分的起始位置
    uint32_t endian_tag_ = 0;    // 大小端标志
```

```
    uint32_t link_size_ = 0;
    uint32_t link_off_ = 0;
    uint32_t map_off_ = 0;
    uint32_t string_ids_size_ = 0;  // 字符串ID的数量
    uint32_t string_ids_off_ = 0;  // 字符串ID数组的文件偏移量
    uint32_t type_ids_size_ = 0;  // 类型ID数，不支持超过65535个
    uint32_t type_ids_off_ = 0;  // 类型ID数组的文件偏移量
    uint32_t proto_ids_size_ = 0;  // ProtoId的数量，不支持超过65535个
    uint32_t proto_ids_off_ = 0;  // ProtoId数组的文件偏移量
    uint32_t field_ids_size_ = 0;  // FieldIds的数量
    uint32_t field_ids_off_ = 0;  // FieldIds数组的文件偏移量
    uint32_t method_ids_size_ = 0;  // MethodIds的数量
    uint32_t method_ids_off_ = 0;  // MethodIds数组的文件偏移量
    uint32_t class_defs_size_ = 0;  // ClassDefs的数量
    uint32_t class_defs_off_ = 0;  // ClassDef数组的文件偏移量
    uint32_t data_size_ = 0;  // 数据部分的大小
    uint32_t data_off_ = 0;  // 数据部分的文件偏移量

    // 解码dex文件版本号。
    uint32_t GetVersion() const;
  };
  ...

 protected:
  // 支持默认方法的第一个Dex格式版本。
  static constexpr uint32_t kDefaultMethodsVersion = 37;

  ...
  // dex文件数据起始位置
  const uint8_t* const begin_;

  // 内存分配的字节数。
  const size_t size_;

  // 数据节的基地址（对于标准dex，与Begin()相同）。
  const uint8_t* const data_begin_;

  // 数据节的大小。
  const size_t data_size_;

  const std::string location_;

  const uint32_t location_checksum_;

  // Dex文件头的指针
  const Header* const header_;

  // 字符串标识符列表的指针
  const dex::StringId* const string_ids_;

  // 类型标识符列表的指针
  const dex::TypeId* const type_ids_;

  // 字段标识符列表的指针
```

```
  const dex::FieldId* const field_ids_;

  // 方法标识符列表的指针
  const dex::MethodId* const method_ids_;

  // 原型标识符列表的指针
  const dex::ProtoId* const proto_ids_;

  // 类定义列表的指针
  const dex::ClassDef* const class_defs_;

  // 方法句柄列表的指针
  const dex::MethodHandleItem* method_handles_;

  // 方法句柄列表中元素的数量
  size_t num_method_handles_;
  ...
};
```

接着查看DexFile的构造函数实现。

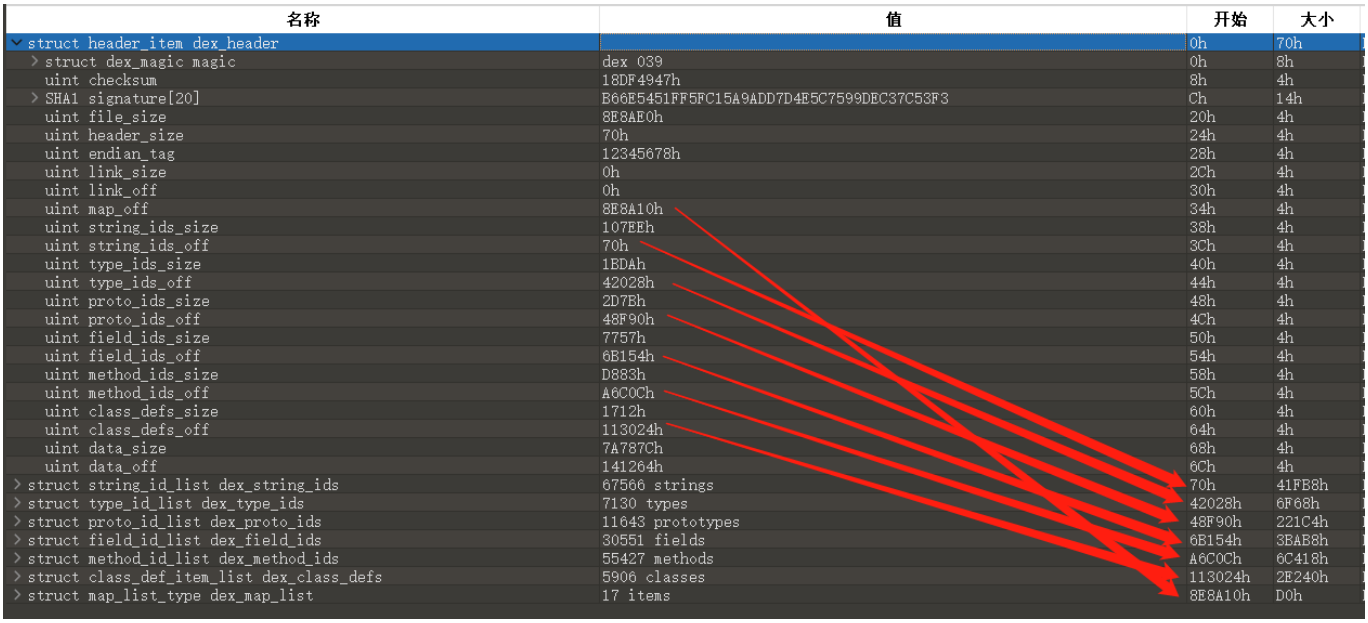
```
DexFile::DexFile(const uint8_t* base,
                 size_t size,
                 const uint8_t* data_begin,
                 size_t data_size,
                 const std::string& location,
                 uint32_t location_checksum,
                 const OatDexFile* oat_dex_file,
                 std::unique_ptr<DexFileContainer> container,
                 bool is_compact_dex)
    : begin_(base),
      size_(size),
      data_begin_(data_begin),
      data_size_(data_size),
      location_(location),
      location_checksum_(location_checksum),
      header_(reinterpret_cast<const Header*>(base)),
      string_ids_(reinterpret_cast<const StringId*>(base + header_-
>string_ids_off_)),
      type_ids_(reinterpret_cast<const TypeId*>(base + header_->type_ids_off_)),
      field_ids_(reinterpret_cast<const FieldId*>(base + header_-
>field_ids_off_)),
      method_ids_(reinterpret_cast<const MethodId*>(base + header_-
>method_ids_off_)),
      proto_ids_(reinterpret_cast<const ProtoId*>(base + header_-
>proto_ids_off_)),
      class_defs_(reinterpret_cast<const ClassDef*>(base + header_-
>class_defs_off_)),
      method_handles_(nullptr),
      num_method_handles_(0),
      call_site_ids_(nullptr),
      num_call_site_ids_(0),
```

```
        hiddenapi_class_data_(nullptr),
        oat_dex_file_(oat_dex_file),
        container_(std::move(container)),
        is_compact_dex_(is_compact_dex),
        hiddenapi_domain_(hiddenapi::Domain::kApplication) {
    CHECK(begin_ != nullptr) << GetLocation();
    CHECK_GT(size_, 0U) << GetLocation();
    // Check base (=header) alignment.
    // Must be 4-byte aligned to avoid undefined behavior when accessing
    // any of the sections via a pointer.
    CHECK_ALIGNED(begin_, alignof(Header));

    InitializeSectionsFromMapList();
  }
```

可以看出header_这个dex文件头的结构体中存储着最重要的信息，初始化时先是填充了header_中的数据，然后再根据header_文件头，将其他重要信息初始化。当需要对这个Dex进行访问时，只需要通过文件头信息，就可以为我们索引找到任何一段信息了。它提供了整个文件的索引。

使用010 Editor工具，通过模板库在线安装DEX.bt模板，然后打开之前的样例文件，查看在例子中header_的真实数据。



名称	值	开始	大小
struct header_item dex_header		0h	70h
struct dex_magic magic	dex 039	0h	8h
uint checksum	18DF4947h	8h	4h
SHA1 signature[20]	B66E5451FF5FC15A9ADD7D4E5C7599DEC37C53F3	Ch	14h
uint file_size	8E8AE0h	20h	4h
uint header_size	70h	24h	4h
uint endian_tag	12345678h	28h	4h
uint link_size	0h	2Ch	4h
uint link_off	0h	30h	4h
uint map_off	8E8A10h	34h	4h
uint string_ids_size	107EEh	38h	4h
uint string_ids_off	70h	3Ch	4h
uint type_ids_size	1BDAh	40h	4h
uint type_ids_off	42028h	44h	4h
uint proto_ids_size	2D7Bh	48h	4h
uint proto_ids_off	48F90h	4Ch	4h
uint field_ids_size	7757h	50h	4h
uint field_ids_off	6B154h	54h	4h
uint method_ids_size	D883h	58h	4h
uint method_ids_off	A6C0Ch	5Ch	4h
uint class_defs_size	1712h	60h	4h
uint class_defs_off	113024h	64h	4h
uint data_size	7A787Ch	68h	4h
uint data_off	141264h	6Ch	4h
struct string_id_list dex_string_ids	67566 strings	70h	41FB8h
struct type_id_list dex_type_ids	7130 types	42028h	6F68h
struct proto_id_list dex_proto_ids	11643 prototypes	48F90h	221C4h
struct field_id_list dex_field_ids	30551 fields	6B154h	3BAB8h
struct method_id_list dex_method_ids	55427 methods	A6C0Ch	6C418h
struct class_def_item_list dex_class_defs	5906 classes	113024h	2E240h
struct map_list_type dex_map_list	17 items	8E8A10h	D0h

7.3 函数调用流程

在Android中，**Java**函数和**native**函数的调用方式略有不同。

对于**Java**函数，它们的执行是由*Android Runtime*虚拟机完成的。具体来说，当应用程序需要调用一个**Java**函数时，*Android Runtime*会根据该函数的状态和类型进行相应的处理，包括解释器执行、JIT编译器动态生成机器码等；当函数执行完毕后，结果会被传递回应用程序。

而对于**native**函数，则是由操作系统内核直接执行的。应用程序需要通过JNI（Java Native Interface）来调用 native 函数。首先将**Java数据结构转换为C/C++类型**, 然后将参数传递给 native 函数, 最终再将结果转换为**Java数据结构并返回给应用程序。**

在这个过程中, JNI提供了一系列的函数和接口来实现Java与本地代码/数据之间的交互和转换。

下面使用反编译工具jadx打开前文中的样例程序，样例程序的代码如下。

```
public class MyCommon {
    public static String getMyJarVer() {
        return "v1.0";
    }
    public static int add(int a, int b) {
        return a + b;
    }
}
```

切换为展示smali指令，并右键选择显示Dalvik字节码，看到如下代码。

```
.method public static add(II)I
    .registers 3

    .param p0, "a":I
    .param p1, "b":I

                            .line 11
    002bf89c: 9000 0102                 0000: add-int             v0, p0, p1
                        .end local v1 # "a":I
                        .end local v2 # "b":I
    002bf8a0: 0f00                      0002: return              v0

.end method
```

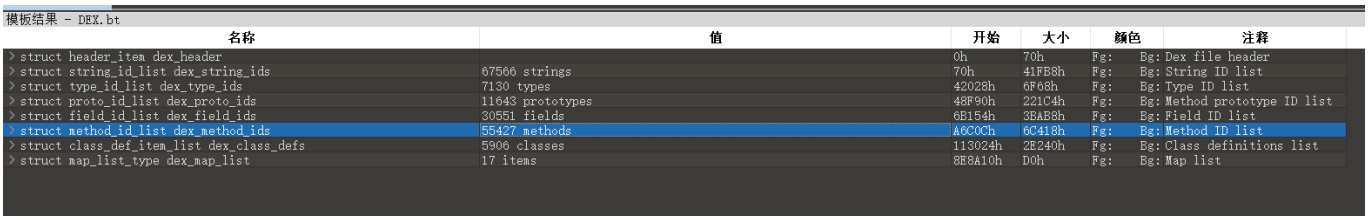
.registers 3表示该函数中使用到了3个寄存器，分别为两个参数和返回值。

002bf89c 表示的是当前指令的偏移地址。

9000 0102 就是该函数中的java字节码，其中前两个字节9000表示的是操作码，由于字节码是大端序存储的，所以这里实际操作码解读为0x0090，下面在AOSP中对Opcodes的定义中也能看到0x90操作码对应的操作是add-int。

```
public interface Opcodes {
    ...
    int OP_ADD_INT                      = 0x0090;
    ...
}
```

使用010 Editor工具，将样例程序解压后获得的classes.dex拖入010 Editor打开。看到结果如下。

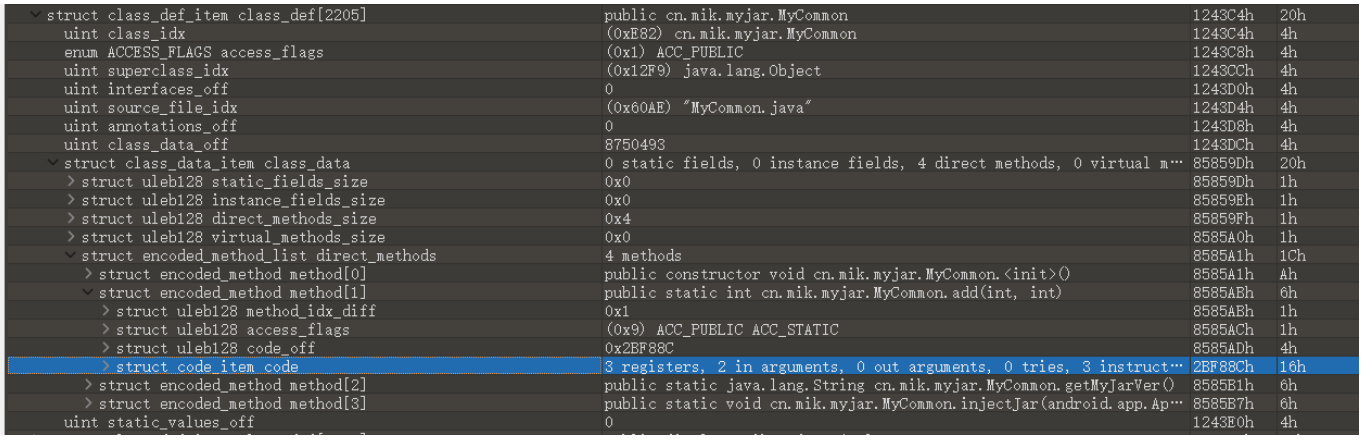

模板结果 - DEX.bt

名称	值	开始	大小	颜色	注释
struct header_item dex_header		0h	70h	Fg:	Bg: Dex file header
struct string_id_list dex_string_ids	67566 strings	70h	41FB8h	Fg:	Bg: String ID list
struct type_id_list dex_type_ids	7130 types	42028h	6F68h	Fg:	Bg: Type ID list
struct proto_id_list dex_proto_ids	11643 prototypes	48F90h	221C4h	Fg:	Bg: Method prototype ID list
struct field_id_list dex_field_ids	30551 fields	6B154h	3BAB8h	Fg:	Bg: Field ID list
struct method_id_list dex_method_ids	55427 methods	A6C0Ch	6C418h	Fg:	Bg: Method ID list
struct class_def_item_list dex_class_defs	5906 classes	113024h	2E240h	Fg:	Bg: Class definitions list
struct map_list_type dex_map_list	17 items	8E8A10h	D0h	Fg:	Bg: Map list

接下来在dex_class_defs中寻找刚刚分析的目标类MyCommon。

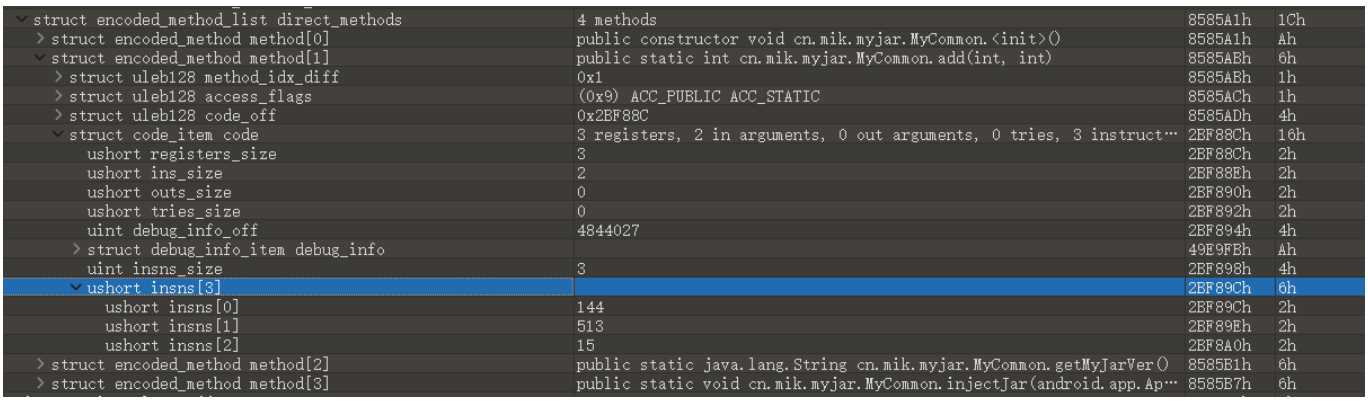
```
struct class_def_item class_def[2205]    public cn.rom.myjar.MyCommon     1243C4h
20h Fg: Bg:0xE0E0E0 Class ID
```

将其展开后，能看到该class的详细信息，在上一节的类加载中，当DEX被解析后，加载的类在内存中就是以这样的结构存储着数据。



struct class_def_item class_def[2205]	public cn.mik.myjar.MyCommon	1243C4h	20h
uint class_idx	(0xE82) cn.mik.myjar.MyCommon	1243C4h	4h
enum ACCESS_FLAGS access_flags	(0x1) ACC_PUBLIC	1243C8h	4h
uint superclass_idx	(0x12F9) java.lang.Object	1243CCh	4h
uint interfaces_off	0	1243D0h	4h
uint source_file_idx	(0x60AE) "MyCommon.java"	1243D4h	4h
uint annotations_off	0	1243D8h	4h
uint class_data_off	8750493	1243DCh	4h
struct class_data_item class_data	0 static fields, 0 instance fields, 4 direct methods, 0 virtual m…	85859Dh	20h
struct uleb128 static_fields_size	0x0	85859Dh	1h
struct uleb128 instance_fields_size	0x0	85859Eh	1h
struct uleb128 direct_methods_size	0x4	85859Fh	1h
struct uleb128 virtual_methods_size	0x0	8585A0h	1h
struct encoded_method_list direct_methods	4 methods	8585A1h	1Ch
struct encoded_method method[0]	public constructor void cn.mik.myjar.MyCommon.<init>()	8585A1h	Ah
struct encoded_method method[1]	public static int cn.mik.myjar.MyCommon.add(int, int)	8585ABh	6h
struct uleb128 method_idx_diff	0x1	8585ABh	1h
struct uleb128 access_flags	(0x9) ACC_PUBLIC ACC_STATIC	8585ACh	1h
struct uleb128 code_off	0x2BF88C	8585ADh	4h
struct code_item code	3 registers, 2 in arguments, 0 out arguments, 0 tries, 3 instruct…	2BF88Ch	16h
struct encoded_method method[2]	public static java.lang.String cn.mik.myjar.MyCommon.getMyJarVer()	8585B1h	6h
struct encoded_method method[3]	public static void cn.mik.myjar.MyCommon.injectJar(android.app.Ap…	8585B7h	6h
uint static_values_off	0	1243E0h	4h

在其中的函数结构体下面的code_item类型的数据，就存储着该函数要执行的java字节码，继续展开该结构。



struct encoded_method_list direct_methods	4 methods	8585A1h	1Ch
struct encoded_method method[0]	public constructor void cn.mik.myjar.MyCommon.<init>()	8585A1h	Ah
struct encoded_method method[1]	public static int cn.mik.myjar.MyCommon.add(int, int)	8585ABh	6h
struct uleb128 method_idx_diff	0x1	8585ABh	1h
struct uleb128 access_flags	(0x9) ACC_PUBLIC ACC_STATIC	8585ACh	1h
struct uleb128 code_off	0x2BF88C	8585ADh	4h
struct code_item code	3 registers, 2 in arguments, 0 out arguments, 0 tries, 3 instruct…	2BF88Ch	16h
ushort registers_size	3	2BF88Ch	2h
ushort ins_size	2	2BF88Eh	2h
ushort outs_size	0	2BF890h	2h
ushort tries_size	0	2BF892h	2h
uint debug_info_off	4844027	2BF894h	4h
struct debug_info_item debug_info		49E9FBh	Ah
uint insns_size	3	2BF898h	4h
ushort insns[3]		2BF89Ch	6h
ushort insns[0]	144	2BF89Ch	2h
ushort insns[1]	513	2BF89Eh	2h
ushort insns[2]	15	2BF8A0h	2h
struct encoded_method method[2]	public static java.lang.String cn.mik.myjar.MyCommon.getMyJarVer()	8585B1h	6h
struct encoded_method method[3]	public static void cn.mik.myjar.MyCommon.injectJar(android.app.Ap…	8585B7h	6h

这里就能看到对该函数结构的描述了，insns中则存储着函数要执行的指令。每个指令的单位是ushort，即两个字节存储，将这里的三个指令转换为16进制表示则是。

```
144   =   00 90 -> 大端序 -> 9000

513   =   02 01 -> 大端序 -> 0102

15    =   00 0F -> 大端序 -> 0f00
```

结果和上面smali展示中的一致。这就是java字节码，在调用过程中，系统会经过层层的转换和解析，最终通过对函数中的指令进行执行来完成函数的调用。

接下来根据之前的例子，开始对函数调用流程的代码进行跟踪分析。

```
protected void onCreate(Bundle savedInstanceState) {
    ...
    Object result = addMethod.invoke(null, 12,25);
    Log.i("MainActivity","getMyJarVer:"+result);
    ...
}
```

找到Method的invoke的实现，这是一个native函数，所以继续找对应的Method_invoke函数。

```
@FastNative
    public native Object invoke(Object obj, Object... args)
            throws IllegalAccessException, IllegalArgumentException,
InvocationTargetException;

static jobject Method_invoke(JNIEnv* env, jobject javaMethod, jobject
javaReceiver,
                             jobjectArray javaArgs) {
  ScopedFastNativeObjectAccess soa(env);
  return InvokeMethod<kRuntimePointerSize>(soa, javaMethod, javaReceiver,
javaArgs);
}

jobject InvokeMethod(const ScopedObjectAccessAlreadyRunnable& soa, jobject
javaMethod,
                     jobject javaReceiver, jobject javaArgs, size_t num_frames) {
  ...

  // Java方法和ArtMethod之间存在映射关系，SOA提供了一种方便的方式来将Java对象转换为Art
虚拟机中的数据对象
  ObjPtr<mirror::Executable> executable = soa.Decode<mirror::Executable>
(javaMethod);
  const bool accessible = executable->IsAccessible();
  ArtMethod* m = executable->GetArtMethod();

  ...

  if (!m->IsStatic()) {
    // Replace calls to String.<init> with equivalent StringFactory call.
    if (declaring_class->IsStringClass() && m->IsConstructor()) {
      ...
    } else {
      ...
      // 查找虚方法的真实实现
      m = receiver->GetClass()->FindVirtualMethodForVirtualOrInterface(m,
kPointerSize);
```

```
    }
  }

  // 对java方法的参数进行转换
  ObjPtr<mirror::ObjectArray<mirror::Object>> objects =
      soa.Decode<mirror::ObjectArray<mirror::Object>>(javaArgs);
  ...

  // 调用函数
  JValue result;
  const char* shorty;
  if (!InvokeMethodImpl(soa, m, np_method, receiver, objects, &shorty, &result)) {
    return nullptr;
  }
  return soa.AddLocalReference<jobject>
(BoxPrimitive(Primitive::GetType(shorty[0]), result));
}
```

在上面这个函数中，主要使用SOA将Java函数以及函数的参数转换为C++对象。

Structured Object Access (SOA) 用于优化Java对象在Native代码和Art虚拟机之间的传递和处理。SOA技术提供了一种高效的方式，将Java对象转换为基于指针的本地C++对象，从而避免了频繁的对象复制和GC操作，提高了程序的性能和执行效率。

在SOA技术中使用Handle和ObjPtr等类型的指针来管理Java对象和本地C++对象之间的映射关系。Handle是一种包装器，用于管理Java对象的生命周期，并确保其在被访问时不会被GC回收。ObjPtr则是一种智能指针，用于管理本地C++对象的生命周期，并确保其正确释放和销毁。

通过SOA可以在Native代码中高效地访问和操作Java对象，例如调用Java方法、读取Java字段等。在执行过程中，SOA技术会自动进行对象的内存分配和管理，以确保程序的正确性和性能表现。

接下来继续了解InvokeMethodImpl函数的实现。

```
ALWAYS_INLINE
bool InvokeMethodImpl(const ScopedObjectAccessAlreadyRunnable& soa,
                      ArtMethod* m,
                      ArtMethod* np_method,
                      ObjPtr<mirror::Object> receiver,
                      ObjPtr<mirror::ObjectArray<mirror::Object>> objects,
                      const char** shorty,
                      JValue* result) REQUIRES_SHARED(Locks::mutator_lock_) {
  // 将函数的参数转换后，存放到arg_array中。
  uint32_t shorty_len = 0;
  *shorty = np_method->GetShorty(&shorty_len);
  ArgArray arg_array(*shorty, shorty_len);
  if (!arg_array.BuildArgArrayFromObjectArray(receiver, objects, np_method,
soa.Self())) {
    CHECK(soa.Self()->IsExceptionPending());
    return false;
  }
  // 函数调用
```

```
  InvokeWithArgArray(soa, m, &arg_array, result, *shorty);
  ...
  return true;
}
```

ArgArray主要用于管理Java方法参数列表的类。ArgArray和Java中的类型对应如下：

1.基本类型：ArgArray中的基本类型分别对应Java中的八种基本类型

- boolean: 'Z'
- byte: 'B'
- short: 'S'
- char: 'C'
- int: 'I'
- long: 'J'
- float: 'F'
- double: 'D'

2.引用类型：ArgArray中的引用类型对应Java中的对象类型，包括String、Object、数组等。在ArgArray中，引用类型用字符'L'开头，并紧跟着完整类名和结尾的分号';'表示，例如'Landroid/content/Context;'表示android.content.Context类。

3.可变参数：可变参数在Java中使用“...”符号表示，而在ArgArray中，则需要将所有可变参数打包为一个数组，并使用‘[’和‘]’符号表示。例如，如果Java方法声明为“public void foo(int a, String... args)”，则在ArgArray中，参数列表的短类型描述符为“ILjava/lang/String;[”。

理解了C++如何存放参数数据后，继续看下一层的函数调用。

```
void InvokeWithArgArray(const ScopedObjectAccessAlreadyRunnable& soa,
                                ArtMethod* method, ArgArray* arg_array, JValue*
result,
                                const char* shorty)
    REQUIRES_SHARED(Locks::mutator_lock_) {
  // 获取java参数的数组指针
  uint32_t* args = arg_array->GetArray();
  if (UNLIKELY(soa.Env()->IsCheckJniEnabled())) {
    CheckMethodArguments(soa.Vm(), method-
>GetInterfaceMethodIfProxy(kRuntimePointerSize), args);
  }
  method->Invoke(soa.Self(), args, arg_array->GetNumBytes(), result, shorty);
}
```

调用到了ArtMethod的Invoke函数，这里将参数的数组指针，参数数组大小，返回值指针，调用函数的描述符号传递了过去。在开始进入关键函数前，先对返回值指针JValue* result进行简单介绍。

JValue是用于存储和传递Java方法返回值的联合体。包含了各种基本类型和引用类型的成员变量。下面是该联合体的定义。

```
union PACKED(alignof(mirror::Object*)) JValue {
  // We default initialize JValue instances to all-zeros.
  JValue() : j(0) {}

  template<typename T> ALWAYS_INLINE static JValue FromPrimitive(T v);

  int8_t GetB() const { return b; }
  void SetB(int8_t new_b) {
    j = ((static_cast<int64_t>(new_b) << 56) >> 56);  // Sign-extend to 64 bits.
  }

  uint16_t GetC() const { return c; }
  void SetC(uint16_t new_c) {
    j = static_cast<int64_t>(new_c);  // Zero-extend to 64 bits.
  }

  double GetD() const { return d; }
  void SetD(double new_d) { d = new_d; }

  float GetF() const { return f; }
  void SetF(float new_f) { f = new_f; }

  int32_t GetI() const { return i; }
  void SetI(int32_t new_i) {
    j = ((static_cast<int64_t>(new_i) << 32) >> 32);  // Sign-extend to 64 bits.
  }

  int64_t GetJ() const { return j; }
  void SetJ(int64_t new_j) { j = new_j; }

  mirror::Object* GetL() const REQUIRES_SHARED(Locks::mutator_lock_) {
    return l;
  }
  ALWAYS_INLINE
  void SetL(ObjPtr<mirror::Object> new_l) REQUIRES_SHARED(Locks::mutator_lock_);

  int16_t GetS() const { return s; }
  void SetS(int16_t new_s) {
    j = ((static_cast<int64_t>(new_s) << 48) >> 48);  // Sign-extend to 64 bits.
  }

  uint8_t GetZ() const { return z; }
  void SetZ(uint8_t new_z) {
    j = static_cast<int64_t>(new_z);  // Zero-extend to 64 bits.
  }

  mirror::Object** GetGCRoot() { return &l; }

 private:
  uint8_t z;
  int8_t b;
  uint16_t c;
```

```
  int16_t s;
  int32_t i;
  int64_t j;
  float f;
  double d;
  mirror::Object* l;
};
```

JValue结构体的大小为8个字节对齐，结构体提供了一些成员函数，例如GetXXX和SetXXX等函数，用于获取和设置不同类型的返回值。alignof(mirror::Object*)的具体值取决于编译器和操作系统的不同，一般为4或8。

对参数以及返回值的在C++中的表示有了初步的了解后，开始继续查看函数调用过程中的关键函数ArtMethod::Invoke，下面是具体实现代码。

```
void ArtMethod::Invoke(Thread* self, uint32_t* args, uint32_t args_size, JValue*
result,
                       const char* shorty) {

  ...

  // 将当前的环境（也就是函数调用时的程序计数器、堆栈指针等信息）保存到一个栈帧中。这个栈
帧通常会被分配在堆上，并且由垃圾回收器来管理。在函数返回时，这个栈帧会被弹出，恢复之前的环
境。
  ManagedStack fragment;
  self->PushManagedStackFragment(&fragment);

  Runtime* runtime = Runtime::Current();
  // IsForceInterpreter为true表示强制使用解释器执行函数
  // 这里的条件是，如果设置了强制走解释器执行，并且非native函数，并且非代理函数，并且可执
行的函数，则符合条件
  if (UNLIKELY(!runtime->IsStarted() ||
               (self->IsForceInterpreter() && !IsNative() && !IsProxyMethod() &&
IsInvokable()))) {
    if (IsStatic()) {
      // 静态函数调用
      art::interpreter::EnterInterpreterFromInvoke(
          self, this, nullptr, args, result, /*stay_in_interpreter=*/ true);
    } else {
      // 非静态函数调用
      mirror::Object* receiver =
          reinterpret_cast<StackReference<mirror::Object>*>(&args[0])-
>AsMirrorPtr();
      art::interpreter::EnterInterpreterFromInvoke(
          self, this, receiver, args + 1, result, /*stay_in_interpreter=*/ true);
    }
  } else {

    ...
    // 是否有已编译的快速执行代码的入口点
```

```
    bool have_quick_code = GetEntryPointFromQuickCompiledCode() != nullptr;
    if (LIKELY(have_quick_code)) {
      ...

      // 走快速调用方式，比解释器执行的性能高。
      if (!IsStatic()) {
        (*art_quick_invoke_stub)(this, args, args_size, self, result, shorty);
      } else {
        (*art_quick_invoke_static_stub)(this, args, args_size, self, result,
shorty);
      }
      ...
    } else {
      LOG(INFO) << "Not invoking '" << PrettyMethod() << "' code=null";
      if (result != nullptr) {
        result->SetJ(0);
      }
    }
  }

  // 从栈帧中还原当前环境
  self->PopManagedStackFragment(fragment);
}
```

根据以上代码得到的结论是，函数执行的路线有两条，`EnterInterpreterFromInvoke`由解释器执行和`art_quick_invoke_stub`快速执行通道。

`art_quick_invoke_stub`是由一段汇编完成对函数的执行，该函数充分利用寄存器并尽可能地减少堆栈访问次数，以提高`Java`方法的执行效率。虽然快速执行通道的效率会更加高，但是可读性差，但是对于学习执行过程和修改执行流程来说，解释器执行会更加简单易改。所以接下来跟进解释器执行，了解执行的细节。继续跟踪`EnterInterpreterFromInvoke`函数。

```
void EnterInterpreterFromInvoke(Thread* self,
                                ArtMethod* method,
                                ObjPtr<mirror::Object> receiver,
                                uint32_t* args,
                                JValue* result,
                                bool stay_in_interpreter) {
  ...

  // 获取函数中的指令信息
  CodeItemDataAccessor accessor(method->DexInstructionData());
  uint16_t num_regs;
  uint16_t num_ins;
  if (accessor.HasCodeItem()) {
    // 获取寄存器的数量和参数的数量
    num_regs =  accessor.RegistersSize();
    num_ins = accessor.InsSize();
  } else if (!method->IsInvokable()) {
    self->EndAssertNoThreadSuspension(old_cause);
    method->ThrowInvocationTimeError();
```

```
    return;
  } else {
    DCHECK(method->IsNative()) << method->PrettyMethod();
    // 从函数描述符中计算出寄存器数量和参数数量
    // 这里将num_regs和num_ins都赋值的原因是，方法的前几个参数通常会存储在寄存器中，而不是堆栈中。因此，num_regs和num_ins的值应该是相同的，都代表了当前方法使用的寄存器数量，也就是用于存储参数和局部变量等数据的寄存器数量。
    num_regs = num_ins = ArtMethod::NumArgRegisters(method->GetShorty());
    // 非静态函数的情况，会多一个this参数，所以寄存器数量和参数数量+1
    if (!method->IsStatic()) {
      num_regs++;
      num_ins++;
    }
  }

  // 创建一个新的ShadowFrame作为当前栈，将当前环境保存在其中，并且推入栈帧，供当前线程调用方法时使用
  ShadowFrame* last_shadow_frame = self->GetManagedStack()->GetTopShadowFrame();
  ShadowFrameAllocaUniquePtr shadow_frame_unique_ptr =
      CREATE_SHADOW_FRAME(num_regs, last_shadow_frame, method, /* dex pc */ 0);
  ShadowFrame* shadow_frame = shadow_frame_unique_ptr.get();
  self->PushShadowFrame(shadow_frame);
  // 计算出将要使用的第一个寄存器
  size_t cur_reg = num_regs - num_ins;
  // 非静态函数的情况，第一个寄存器的值为this，所以设置其为引用类型
  if (!method->IsStatic()) {
    // receiver变量表示方法调用的第一个参数
    CHECK(receiver != nullptr);
    shadow_frame->SetVRegReference(cur_reg, receiver);
    ++cur_reg;
  }
  uint32_t shorty_len = 0;
  const char* shorty = method->GetShorty(&shorty_len);
  // 遍历所有参数
  for (size_t shorty_pos = 0, arg_pos = 0; cur_reg < num_regs; ++shorty_pos, ++arg_pos, cur_reg++) {
    DCHECK_LT(shorty_pos + 1, shorty_len);
    switch (shorty[shorty_pos + 1]) {
      //L 表示这个参数是个引用类型，比如Ljava/lang/String;
      case 'L': {
        ObjPtr<mirror::Object> o =
            reinterpret_cast<StackReference<mirror::Object>*>(&args[arg_pos])->AsMirrorPtr();
        // 将转换好的数据设置到当前栈中
        shadow_frame->SetVRegReference(cur_reg, o);
        break;
      }
      case 'J': case 'D': {
        // J或者D的数据类型要占用两个寄存器存放。
        uint64_t wide_value = (static_cast<uint64_t>(args[arg_pos + 1]) << 32) | args[arg_pos];
        // 合并后的数据设置到栈中
        shadow_frame->SetVRegLong(cur_reg, wide_value);
        cur_reg++;
```

```
          arg_pos++;
          break;
        }
        default:
          // 普通的整型数据设置到栈中
          shadow_frame->SetVReg(cur_reg, args[arg_pos]);
          break;
      }
    }
    self->EndAssertNoThreadSuspension(old_cause);
    // 静态函数的情况需要检查所在的类是否已经正常初始化。
    if (method->IsStatic()) {
      ObjPtr<mirror::Class> declaring_class = method->GetDeclaringClass();
      if (UNLIKELY(!declaring_class->IsVisiblyInitialized())) {
        StackHandleScope<1> hs(self);
        Handle<mirror::Class> h_class(hs.NewHandle(declaring_class));
        if (UNLIKELY(!Runtime::Current()->GetClassLinker()->EnsureInitialized(
                          self, h_class, /*can_init_fields=*/ true,
/*can_init_parents=*/ true))) {
          CHECK(self->IsExceptionPending());
          self->PopShadowFrame();
          return;
        }
        DCHECK(h_class->IsInitializing());
      }
    }
    // 非native函数执行
    if (LIKELY(!method->IsNative())) {
      // 解释执行的关键函数
      JValue r = Execute(self, accessor, *shadow_frame, JValue(),
stay_in_interpreter);
      if (result != nullptr) {
          *result = r;
      }
    } else {
      // native函数的解释执行
      args = shadow_frame->GetVRegArgs(method->IsStatic() ? 0 : 1);
      if (!Runtime::Current()->IsStarted()) {
        UnstartedRuntime::Jni(self, method, receiver.Ptr(), args, result);
      } else {
        InterpreterJni(self, method, shorty, receiver, args, result);
      }
    }
    // 弹出栈帧，还原到执行后的栈环境
    self->PopShadowFrame();
}
```

在这个函数中，为即将执行的函数准备好了栈帧环境，将参数填入了`shadow_frame`栈帧中。并且获取出了函数要执行的指令信息`accessor`。最后通过`Execute`执行该函数。

```
static inline JValue Execute(
    Thread* self,
    const CodeItemDataAccessor& accessor,
    ShadowFrame& shadow_frame,
    JValue result_register,
    bool stay_in_interpreter = false,
    bool from_deoptimize = false) REQUIRES_SHARED(Locks::mutator_lock_) {
  ...

  // 是否需要从解释器模式切换到编译模式。
  if (LIKELY(!from_deoptimize)) {
    ...

    instrumentation::Instrumentation* instrumentation = Runtime::Current()-
>GetInstrumentation();
    // 从当前线程栈帧中获取要执行的函数
    ArtMethod *method = shadow_frame.GetMethod();
    // 是否有注册Method Entry 监听器
    if (UNLIKELY(instrumentation->HasMethodEntryListeners())) {
      // 触发 Method Entry 监听器，并传递相应的参数
      instrumentation->MethodEnterEvent(self,

shadow_frame.GetThisObject(accessor.InsSize()),
                                        method,
                                        0);
      ...
      // 是否有未处理的异常
      if (UNLIKELY(self->IsExceptionPending())) {
        ...
        return ret;
      }
    }
    // stay_in_interpreter 表示是否需要停留在解释器模式，self->IsForceInterpreter()
表示是否强制使用解释器模式。所以内部是不走解释器执行的处理，走编译模式执行
    if (!stay_in_interpreter && !self->IsForceInterpreter()) {
      jit::Jit* jit = Runtime::Current()->GetJit();
      if (jit != nullptr) {
        // 判断当前方法是否可以使用 JIT 编译后的机器码执行
        jit->MethodEntered(self, shadow_frame.GetMethod());
        if (jit->CanInvokeCompiledCode(method)) {
          JValue result;
          // 直接栈帧推出
          self->PopShadowFrame();

          uint16_t arg_offset = accessor.RegistersSize() - accessor.InsSize();
          // 调用该函数的机器码实现
          ArtInterpreterToCompiledCodeBridge(self, nullptr, &shadow_frame,
arg_offset, &result);
          // 重新推入栈帧
          self->PushShadowFrame(&shadow_frame);

          return result;
        }
```

```
      }
    }
  }
  // 从栈帧中获取要执行的当前函数
  ArtMethod* method = shadow_frame.GetMethod();
  ...
  // kSwitchImplKind：表示当前实现是否使用基于 switch 语句的解释器实现。
  if (kInterpreterImplKind == kSwitchImplKind ||
      UNLIKELY(!Runtime::Current()->IsStarted()) ||
      !method->IsCompilable() ||
      method->MustCountLocks() ||
      Runtime::Current()->IsActiveTransaction()) {
    // 使用switch解释器执行
    return ExecuteSwitch(
        self, accessor, shadow_frame, result_register,
/*interpret_one_instruction=*/ false);
  }

  CHECK_EQ(kInterpreterImplKind, kMterpImplKind);
  // 编译执行函数
  while (true) {
    // 是否支持Mterp解释器执行
    if (!self->UseMterp()) {
      return ExecuteSwitch(
          self, accessor, shadow_frame, result_register,
/*interpret_one_instruction=*/ false);
    }
    // 执行目标函数
    bool returned = ExecuteMterpImpl(self,
                                     accessor.Insns(),
                                     &shadow_frame,
                                     &result_register);
    if (returned) {
      return result_register;
    } else {
      // 失败的情况继续采用switch解释器执行
      result_register = ExecuteSwitch(
          self, accessor, shadow_frame, result_register,
/*interpret_one_instruction=*/ true);
      if (shadow_frame.GetDexPC() == dex::kDexNoIndex) {
        return result_register;
      }
    }
  }
}
```

看完该函数后，在继续深入前，先将其中的几个知识点进行介绍。

编译模式（Compiled Mode）是一种执行方式，它将应用程序代码编译成机器码后再执行。相较于解释器模式，编译模式具有更高的执行效率和更好的性能表现。

在Android应用程序中，编译模式采用的是 Just-In-Time (JIT) 编译技术。当一个方法被多次调用时，系统会自动将其编译成本地机器码，并缓存起来以备下次使用。

当一个方法被编译成本地机器码后，其执行速度将显著提高。因为与解释器模式相比，编译模式不需要逐条解释代码，而是直接执行编译好的机器码。

由于编译过程需要一定的时间，因此在程序启动或者第一次运行新方法时，可能会出现一些额外的延迟。所以，在实际应用中，系统通常会采用一些策略，如预热机制等，来优化编译模式的性能表现。编译模式是一种性能更高、效率更好的执行方式，可以帮助应用程序在运行时获得更好的响应速度和用户体验。

Method Entry 监听器是Android系统中的一种监听器，它可以用来监听应用程序的方法入口。当一个方法被调用时，系统会触发Method Entry监听器，并将当前线程、当前方法和调用栈信息等相关数据传递给监听器。

Android Studio在调试模式下会自动为每个线程启动一个监听器，并在方法进入和退出时触发相应的事件。这些事件包括 Method Entry （方法入口）、Method Exit （方法出口）等。

下面将分别介绍ExecuteMterpImpl和ExecuteSwitch是如何实现指令流的执行。

7.4 ExecuteMterpImpl

ExecuteMterpImpl是基于Mterp (Method Interpreter) 技术实现。Mterp技术使用指令集解释器来执行应用程序的代码，相比于JIT编译模式可以更快地启动和执行短小精悍的方法，同时也可以避免JIT编译带来的额外开销。

在Mterp模式下，Dex 指令集被转化成了一组C++的函数，这些函数对应Dex指令集中的每一条指令。ExecuteMterpImpl实际上就是调用这些函数来逐条解释执行当前方法的指令集。

在Android 4.4中，系统首次引入了 Mterp 技术来加速应用程序的解释执行。在此之后的 Android版本中，Mterp 技术得到了不断优化和完善，并逐渐成为Android平台的主要方法执行方式之一。

从Android 6.0开始，Dalvik 运行时环境被弃用，取而代之的是ART运行时环境。ART 运行时环境可以通过JIT编译、AOT 编译和Mterp等多种方式来执行应用程序的代码，其中Mterp技术被广泛使用于 Android 应用程序的解释执行过程中。但是对于某些特定的场景和应用程序，系统可能还是会选择其他的执行方式来获得更好的性能和效率。

ExecuteMterpImpl使用了汇编语言和 C++语言混合编写，需要有一定的汇编和C++编程经验才能理解其含义和功能。该代码主要实现了以下功能：

1.保存当前方法的返回值寄存器和指令集

2.设置方法执行的环境和参数，包括vregs数组、dex_pc 寄存器等

3.为当前方法设置热度倒计时，并根据热度值来判断是否需要启用Mterp技术

4.执行当前方法的指令集，逐条解释执行 Dex 指令

下面看ExecuteMterpImpl实现代码。

```
// 从xPC寄存器中获取一条指令
```

```
.macro FETCH_INST
    ldrh    wINST, [xPC]
.endm

// 从指令中获取操作码，指令最顶部2个字节就是操作码，所以这里拿操作码是 & 0xff的意思
.macro GET_INST_OPCODE reg
    and     \reg, xINST, #255
.endm

// 跳转到操作码处理逻辑
.macro GOTO_OPCODE reg
    add     \reg, xIBASE, \reg, lsl #${handler_size_bits}
    br      \reg
.endm

ENTRY ExecuteMterpImpl
    .cfi_startproc
    // 保存寄存器信息
    SAVE_TWO_REGS_INCREASE_FRAME xPROFILE, x27, 80
    SAVE_TWO_REGS                xIBASE, xREFS, 16
    SAVE_TWO_REGS                xSELF, xINST, 32
    SAVE_TWO_REGS                xPC, xFP, 48
    SAVE_TWO_REGS                fp, lr, 64

    // fp寄存器指向栈顶
    add     fp, sp, #64

    /* 记录对应返回值的寄存器 */
    str     x3, [x2, #SHADOWFRAME_RESULT_REGISTER_OFFSET]

    /* 记录dex文件中的指令的指针 */
    str     x1, [x2, #SHADOWFRAME_DEX_INSTRUCTIONS_OFFSET]

    mov     xSELF, x0
    ldr     w0, [x2, #SHADOWFRAME_NUMBER_OF_VREGS_OFFSET]
    add     xFP, x2, #SHADOWFRAME_VREGS_OFFSET     // 计算局部变量表的偏移地址
    add     xREFS, xFP, w0, uxtw #2                // 计算局部变量引用表的偏移地址
    ldr     w0, [x2, #SHADOWFRAME_DEX_PC_OFFSET]   // 获取当前Dex中的PC
    add     xPC, x1, w0, uxtw #1                   // 将Dex PC转换为地址，并保存到寄
存器xPC中
    CFI_DEFINE_DEX_PC_WITH_OFFSET(CFI_TMP, CFI_DEX, 0)
    EXPORT_PC                   // 将Dex PC导出

    /* Starting ibase */
    ldr     xIBASE, [xSELF, #THREAD_CURRENT_IBASE_OFFSET]

    /* Set up for backwards branches & osr profiling */
    ldr     x0, [xFP, #OFF_FP_METHOD]               // 获取当前方法的方法指针
    add     x1, xFP, #OFF_FP_SHADOWFRAME            // 计算拿到当前线程的栈帧
    mov     x2, xSELF                               // 将当前线程对象保存到寄存器x2
中
    bl      MterpSetUpHotnessCountdown              // 热度计数器调整
    mov     wPROFILE, w0                // 将热度计数器的宽度赋值给寄存器wPROFILE
```

```
    /* start executing the instruction at rPC */
    FETCH_INST                          // 获取下一条指令
    GET_INST_OPCODE ip                  // 从指令中获取操作码
    GOTO_OPCODE ip                      // 跳转到操作码处理逻辑
    /* NOTE: no fallthrough */
    // cfi info continues, and covers the whole mterp implementation.
    END ExecuteMterpImpl
```

这些操作码可以通过Opcodes找到其对应的对应，代码如下。

```
public interface Opcodes {
    ...
    int OP_IPUT_CHAR                    = 0x005e;
    int OP_IPUT_SHORT                   = 0x005f;
    int OP_SGET                         = 0x0060;
    int OP_SGET_WIDE                    = 0x0061;
    ...
}
```

而在在汇编文件中，会有其对应操作码的具体实现。

```
%def field(helper=""):
    .extern $helper
    mov      x0, xPC                        // arg0: 指令的地址
    mov      x1, xINST                      // arg1: 指令对应的16位数值
    add      x2, xFP, #OFF_FP_SHADOWFRAME   // arg2: ShadowFrame* sf
    mov      x3, xSELF                      // arg3: Thread* self
    PREFETCH_INST 2                         // 预备取下一条指令
    bl       $helper                        // 调用 $helper 函数
    cbz      x0, MterpPossibleException
    ADVANCE 2
    GET_INST_OPCODE ip                      // 从指令中获取操作码
    GOTO_OPCODE ip                          // 跳转到操作码处理逻辑

%def op_iput(helper="MterpIPutU32"):
%  field(helper=helper)

%def op_sget(helper="MterpSGetU32"):
%  field(helper=helper)

%def op_iput_char():
%  op_iput(helper="MterpIPutU16")

%def op_iput_short():
%  op_iput(helper="MterpIPutI16")

%def op_sget_wide():
%  op_sget(helper="MterpSGetU64")
```

到这里，就找到对应的执行C++函数将Dex的指令逐一进行执行处理，其对应的C++执行部分则在文件mterp.cc文件中找到。Mterp的执行流程到这里就非常清晰了。

7.5 ExecuteSwitch

ExecuteSwitch是基于 switch 语句实现的一种解释器，用于执行当前方法的指令集。在 Android 应用程序中，每个方法都会对应一组指令集，用于描述该方法的具体实现。当该方法被调用时，系统需要按照指令集来执行相应的操作，从而实现该方法的功能并计算出结果。

```
static JValue ExecuteSwitch(Thread* self,
                            const CodeItemDataAccessor& accessor,
                            ShadowFrame& shadow_frame,
                            JValue result_register,
                            bool interpret_one_instruction)
REQUIRES_SHARED(Locks::mutator_lock_) {
  // 是否处于事务中
  if (Runtime::Current()->IsActiveTransaction()) {
    // 是否跳过访问检查
    if (shadow_frame.GetMethod()->SkipAccessChecks()) {
      return ExecuteSwitchImpl<false, true>(
          self, accessor, shadow_frame, result_register,
interpret_one_instruction);
    } else {
      return ExecuteSwitchImpl<true, true>(
          self, accessor, shadow_frame, result_register,
interpret_one_instruction);
    }
  } else {
    if (shadow_frame.GetMethod()->SkipAccessChecks()) {
      return ExecuteSwitchImpl<false, false>(
          self, accessor, shadow_frame, result_register,
interpret_one_instruction);
    } else {
      return ExecuteSwitchImpl<true, false>(
          self, accessor, shadow_frame, result_register,
interpret_one_instruction);
    }
  }
}
```

在这个函数中，根据条件调整参数，最终都是调用ExecuteSwitchImpl，下面继续看解释器的实现。

```
template<bool do_access_check, bool transaction_active>
void ExecuteSwitchImplCpp(SwitchImplContext* ctx) {
  ...
  // 获取到当前正在执行的Dex指令在CodeItem中的索引位置
  uint32_t dex_pc = shadow_frame.GetDexPC();
  const auto* const instrumentation = Runtime::Current()->GetInstrumentation();
  // 获取指令流
```

```
  const uint16_t* const insns = accessor.Insns();
  // 将当前指令转换为专门用来操作指令的Instruction类
  const Instruction* next = Instruction::At(insns + dex_pc);

  DCHECK(!shadow_frame.GetForceRetryInstruction())
      << "Entered interpreter from invoke without retry instruction being
handled!";

  bool const interpret_one_instruction = ctx->interpret_one_instruction;
  while (true) {
    // 获取下一条待执行的指令
    const Instruction* const inst = next;
    dex_pc = inst->GetDexPc(insns);
    // 更新pc位置
    shadow_frame.SetDexPC(dex_pc);
    TraceExecution(shadow_frame, inst, dex_pc);
    // 从指令中获取到操作码
    uint16_t inst_data = inst->Fetch16(0);
    bool exit = false;
    bool success;  // Moved outside to keep frames small under asan.
    // 执行指令前的预处理
    if (InstructionHandler<do_access_check, transaction_active,
Instruction::kInvalidFormat>(
            ctx, instrumentation, self, shadow_frame, dex_pc, inst, inst_data,
next, exit).
            Preamble()) {
      DCHECK_EQ(self->IsExceptionPending(), inst->Opcode(inst_data) ==
Instruction::MOVE_EXCEPTION);
      // 这是一个超大的switch，根据操作码来选择如何执行
      switch (inst->Opcode(inst_data)) {
#define OPCODE_CASE(OPCODE, OPCODE_NAME, NAME, FORMAT, i, a, e, v)
\
        case OPCODE: {
\
          next = inst-
>RelativeAt(Instruction::SizeInCodeUnits(Instruction::FORMAT));             \
          success = OP_##OPCODE_NAME<do_access_check, transaction_active>(
\
              ctx, instrumentation, self, shadow_frame, dex_pc, inst, inst_data,
next, exit);     \
          if (success && LIKELY(!interpret_one_instruction)) {
\
            continue;
\
          }
\
          break;
\
        }
  DEX_INSTRUCTION_LIST(OPCODE_CASE)
#undef OPCODE_CASE
      }
    }
    ...
```

```
  }
}
```

switch解释器，就是指的这个函数中，使用switch来对不同的所有操作码进行对应的处理，但是这里并没有看到非常大的case条件，这是因为代码都在OPCODE_CASE定义中，找到这个定义的实现如下。

```
#define OPCODE_CASE(OPCODE, OPCODE_NAME, NAME, FORMAT, i, a, e, v)
\
template<bool do_access_check, bool transaction_active>
\
ASAN_NO_INLINE static bool OP_##OPCODE_NAME(
\
    SwitchImplContext* ctx,
\
    const instrumentation::Instrumentation* instrumentation,
\
    Thread* self,
\
    ShadowFrame& shadow_frame,
\
    uint16_t dex_pc,
\
    const Instruction* inst,
\
    uint16_t inst_data,
\
    const Instruction*& next,
\
    bool& exit) REQUIRES_SHARED(Locks::mutator_lock_) {
\
  InstructionHandler<do_access_check, transaction_active, Instruction::FORMAT>
handler(            \
      ctx, instrumentation, self, shadow_frame, dex_pc, inst, inst_data, next,
exit);              \
  return LIKELY(handler.OPCODE_NAME());
\
}
DEX_INSTRUCTION_LIST(OPCODE_CASE)
#undef OPCODE_CASE
```

可以看到内部是调用了初始化了一个InstructionHandler对象，然后handler.OPCODE_NAME()调用了对应的操作码函数。最后看看其实现。

```
class InstructionHandler {
    ...
    ALWAYS_INLINE InstructionHandler(SwitchImplContext* ctx,
                                     const instrumentation::Instrumentation*
instrumentation,
                                     Thread* self,
```

```
                                    ShadowFrame& shadow_frame,
                                    uint16_t dex_pc,
                                    const Instruction* inst,
                                    uint16_t inst_data,
                                    const Instruction*& next,
                                    bool& exit_interpreter_loop)
      : ctx_(ctx),
        instrumentation_(instrumentation),
        self_(self),
        shadow_frame_(shadow_frame),
        dex_pc_(dex_pc),
        inst_(inst),
        inst_data_(inst_data),
        next_(next),
        exit_interpreter_loop_(exit_interpreter_loop) {
    }

    ...

    HANDLER_ATTRIBUTES bool INVOKE_STATIC() {
      return HandleInvoke<kStatic, /*is_range=*/ false>();
    }

    HANDLER_ATTRIBUTES bool INVOKE_STATIC_RANGE() {
      return HandleInvoke<kStatic, /*is_range=*/ true>();
    }
    ...
  }
```

所有操作码对应的实现都是在`InstructionHandler`中进行实现，`switch`解释器的做法非常简单粗暴，尽量性能较差，但是可读性高，当需求是对调用流程进行打桩，或者定制修改时，可以选择强制其走`switch`解释器来执行该函数。

需要注意的是，在执行的优化中，当强制走解释器流程调用后，它会交给`JIT`编译器进行编译，生成本地机器码。在生成机器码的同时，`JIT`编译器会将该函数的入口地址设置为生成的机器码的地址。在下一次调用该函数时，虚拟机就会跳过解释器阶段，直接执行机器码，从而提高程序的执行效率。

7.6 本章小结

本章主要介绍了安卓系统中DEX文件的类的加载机制与细节。相比于实际操作动手修改代码，本章介绍的内容显示更加枯燥乏味，但是深入了解系统内部的运行机制，有助于更宏观视角的去理解的程序执行。掌握这一部分内容，在代码修改点的选择上，尤其是系统组件的部分代码，将会更加精准。而且，本章内容同样适合于二进制安全对抗研究领域，是研究软件加密与解密必不可少的基础知识。