Android ClassLoader

Last edited by **CaoQuanli** 2 weeks ago

Android ClassLoader

Table of Contents

Android classpath记录

1. Android ClassLoader分类

Android的ClassLoader和Java的ClassLoader工作机制类似，都是负责加载类的实现到内存中。 不同的是Android加载的是dex文件，而Java虚拟机加载的是class字节码。

加载类文件到内存中，那么就存在一个问题：类的文件存放在哪里，从何处加载？在Android中存 在三种不同的类加载，分别用来加载不同的类文件。

1. boot classloader
2. system server classloader
3. application classloader

1.1. Boot ClassLoader

Boot ClassLoader负责加载初始化的类库。这些类库有哪一些呢？我们可以通过进程的环境变量 看看这些类库有哪一些。

运行adb shell登录机器，运行如下命令：

```
 generic_x86_64:/ # env
_=/system/bin/env
ANDROID_DATA=/data
DOWNLOAD_CACHE=/data/cache
LOGNAME=root
HOME=/
TERM=xterm-256color
ANDROID_ROOT=/system
ANDROID_BOOTLOGO=1
TMPDIR=/data/local/tmp
ANDROID_ASSETS=/system/app
SHELL=/system/bin/sh
BOOTCLASSPATH=/system/framework/core-oj.jar:/system/framework/core-libart.jar:/system/frame
ASEC_MOUNTPOINT=/mnt/asec
ANDROID_SOCKET_adbd=9
HOSTNAME=generic_x86_64
EXTERNAL_STORAGE=/sdcard
ANDROID_STORAGE=/storage
USER=root
PATH=/sbin:/system/sbin:/system/bin:/system/xbin:/vendor/bin:/vendor/xbin
SYSTEMSERVERCLASSPATH=/system/framework/services.jar:/system/framework/ethernet-service.jar
```

有上述输出结果可以得知：

- BOOTCLASSPATH 包含：
  - core-oj.jar
  - core-libart.jar
  - conscrypt.jar
  - okhttp.jar
  - bouncycastle.jar

- apache-xml.jar
- ext.jar
- framework.jar
- telephony-common.jar
- voip-common.jar
- ims-common.jar
- android.hidl.base-V1.0-java.jar
- android.hidl.manager-V1.0-java.jar
- framework-oahl-backward-compatibility.jar
- android.test.base.jar

那么BOOTCLASSPATH的环境变量又是哪里设置的呢?

1. 在system/core/rootdir/Android.mk文件中有如下代码片段

```
include $(CLEAR_VARS)
LOCAL_MODULE_CLASS := ETC
LOCAL_MODULE := init.environ.rc
LOCAL_MODULE_PATH := $(TARGET_ROOT_OUT)
……

include $(BUILD_SYSTEM)/base_rules.mk

# Regenerate init.environ.rc if PRODUCT_BOOTCLASSPATH has changed.
bcp_md5 := $(word 1, $(shell echo $(PRODUCT_BOOTCLASSPATH) $(PRODUCT_SYSTEM_SERVER_CLAS
bcp_dep := $(intermediates)/$(bcp_md5).bcp.dep
$(bcp_dep) :
        $(hide) mkdir -p $(dir $@) && rm -rf $(dir $@)*.bcp.dep && touch $@

$(LOCAL_BUILT_MODULE): $(LOCAL_PATH)/init.environ.rc.in $(bcp_dep)
        @echo "Generate: $< -> $@"
        @mkdir -p $(dir $@)
        $(hide) sed -e 's?%BOOTCLASSPATH%?$(PRODUCT_BOOTCLASSPATH)?g' $< >$@
        $(hide) sed -i -e 's?%SYSTEMSERVERCLASSPATH%?$(PRODUCT_SYSTEM_SERVER_CLASSPATH)
        $(hide) sed -i -e 's?%EXPORT_GLOBAL_ASAN_OPTIONS%?$(EXPORT_GLOBAL_ASAN_OPTIONS)
        $(hide) sed -i -e 's?%EXPORT_GLOBAL_GCOV_OPTIONS%?$(EXPORT_GLOBAL_GCOV_OPTIONS)

bcp_md5 :=
bcp_dep :=
```

上述代码的意思是：以system/core/rootdir/init.environ.rc 生成
out/target/product/generic_x86_64/obj/ETC/init.environ.rc_intermediates/init.environ.rc

- 在 **build/make/core/config.mk** 中对于hide变量有如下定义：hide := @ . @ 在makefile语法中的意义： 通常makefile会将其执行的命令行在执行前输出到屏幕上。如果将‘@’添加到命令行前，这个命令将不被make回显出来。
- $< 表示第一个依赖文件,$@ 表示目标文件
- sed命令使用$(PRODUCT_BOOTCLASSPATH)替换掉%BOOTCLASSPATH%
- sed命令使用$(PRODUCT_SYSTEM_SERVER_CLASSPATH)替换掉%SYSTEMSERVERCLASSPATH%

2. PRODUCT_BOOTCLASSPATH 和 PRODUCT_SYSTEM_SERVER_CLASSPATH变量 以上两个变量是在build/core/make/dex_preopt.mk 中赋值：

```
 DEXPREOPT_BOOT_JARS := $(subst $(space),:,$(PRODUCT_BOOT_JARS))
DEXPREOPT_BOOT_JARS_MODULES := $(PRODUCT_BOOT_JARS)
PRODUCT_BOOTCLASSPATH := $(subst $(space),:,$(foreach m,$(DEXPREOPT_BOOT_JARS_MODULES),
PRODUCT_SYSTEM_SERVER_CLASSPATH := $(subst $(space),:,$(foreach m,$(PRODUCT_SYSTEM_SERV
```

而**PRODUCT_BOOT_JARS**和**PRODUCT_SYSTEM_SERVER_JARS**变量是在build/make/target/product/core_minimal.mk中进行初始化赋值。

3. 最后把out/target/product/generic_x86_64/obj/ETC/init.environ.rc_intermediates/init.environ.rc 拷贝到 out/target/product/generic_x86_64/root/init.environ.rc

4. BOOTCLASS加载过程调用栈

```
01-02 20:53:04.264  3772  3772 D parser  : #00 pc 000000000045778c  /system/lib64/lib
art.so (art::ParsedOptions::DoParse(std::__1::vector<std::__1::pair<std::__1::basic_s
tring<char, std::__1::char_traits<char>, std::__1::allocator<char>>, void const*>, st
d::__1::allocator<std::__1::pair<std::__1::basic_string<char, std::__1::char_traits<c
har>, std::__1::allocator<char>>, void const*>>> const&, bool, art::RuntimeArgumentMa
p*)+6732)
01-02 20:53:04.264  3772  3772 D parser  : #01 pc 0000000000455cb2  /system/lib64/lib
art.so (art::ParsedOptions::Parse(std::__1::vector<std::__1::pair<std::__1::basic_str
ing<char, std::__1::char_traits<char>, std::__1::allocator<char>>, void const*>, st
d::__1::allocator<std::__1::pair<std::__1::basic_string<char, std::__1::char_traits<c
har>, std::__1::allocator<char>>, void const*>>> const&, bool, art::RuntimeArgumentMa
p*)+98)
01-02 20:53:04.264  3772  3772 D parser  : #02 pc 00000000004c02df  /system/lib64/lib
art.so (art::Runtime::ParseOptions(std::__1::vector<std::__1::pair<std::__1::basic_st
ring<char, std::__1::char_traits<char>, std::__1::allocator<char>>, void const*>, st
d::__1::allocator<std::__1::pair<std::__1::basic_string<char, std::__1::char_traits<c
har>, std::__1::allocator<char>>, void const*>>> const&, bool, art::RuntimeArgumentMa
p*)+63)
01-02 20:53:04.264  3772  3772 D parser  : #03 pc 00000000004c7e5e  /system/lib64/lib
art.so (art::Runtime::Create(std::__1::vector<std::__1::pair<std::__1::basic_string<c
har, std::__1::char_traits<char>, std::__1::allocator<char>>, void const*>, std::__
1::allocator<std::__1::pair<std::__1::basic_string<char, std::__1::char_traits<char>,
std::__1::allocator<char>>, void const*>>> const&, bool)+46)
01-02 20:53:04.264  3772  3772 D parser  : #04 pc 0000000000324c4e  /system/lib64/lib
art.so (JNI_CreateJavaVM+1278)
01-02 20:53:04.264  3772  3772 D parser  : #05 pc 00000000000b4b23  /system/lib64/lib
android_runtime.so (android::AndroidRuntime::startVm(_JavaVM**, _JNIEnv**, bool)+769
9)
01-02 20:53:04.264  3772  3772 D parser  : #06 pc 00000000000b50a5  /system/lib64/lib
android_runtime.so (android::AndroidRuntime::start(char const*, android::Vector<andro
id::String8> const&, bool)+309)
01-02 20:53:04.264  3772  3772 D parser  : #07 pc 00000000000021fd  /system/bin/app_p
rocess64 (main+1357)
```

1.2. System Server ClassLoader

由Boot ClassLoader中的输出可以知道，SYSTEMSERVERCLASSPATH 包含：

- services.jar
- ethernet-service.jar
- wifi-service.jar
- com.android.location.provider.jar

SYSTEMSERVERCLASSPATH用来设置System Server进程的classpath，这个classpath只给System Server使用。 在 frameworks/base/core/java/com/android/internal/os/ZygoteInit.java中如下代码片段获取classpath然后创建类加载器：

```
 private static Runnable handleSystemServerProcess(ZygoteConnection.Arguments parsedArgs) {
……
final String systemServerClasspath = Os.getenv("SYSTEMSERVERCLASSPATH");
……
ClassLoader cl = null;
if (systemServerClasspath != null) {
    cl = createPathClassLoader(systemServerClasspath, parsedArgs.targetSdkVersion);

    Thread.currentThread().setContextClassLoader(cl);
}

}
```

1.3. Application ClassLoader

应用的ApplicationLoaders用来加载应用的代码，同时负责jni库的load工作。应用分为两种

- Bundled App

在 LoadedApk.java中createOrUpdateClassLoaderLocked方法用来创建应用的类加载器。在这个方法中 对于一个应用是否是Bundled App有判断

```
private void createOrUpdateClassLoaderLocked(List<String> addedPaths) {
        ……
    boolean isBundledApp = mApplicationInfo.isSystemApp()
        && !mApplicationInfo.isUpdatedSystemApp();

        final String defaultSearchPaths = System.getProperty("java.library.path");
        makePaths(mActivityThread, isBundledApp, mApplicationInfo, zipPaths, libPaths);
        ……
        final String librarySearchPath = TextUtils.join(File.pathSeparator, libPaths);

    mClassLoader = ApplicationLoaders.getDefault().getClassLoader(zip,
                mApplicationInfo.targetSdkVersion, isBundledApp, librarySearchPath,
        libraryPermittedPath, mBaseClassLoader,
        mApplicationInfo.classLoaderName);
}

public static void makePaths(ActivityThread activityThread,
                                boolean isBundledApp,
                                ApplicationInfo aInfo,
                                List<String> outZipPaths,
                                List<String> outLibPaths) {
……

        if (isBundledApp) {
                // Add path to system libraries to libPaths;
        // Access to system libs should be limited
        // to bundled applications; this is why updated
        // system apps are not included.
        outLibPaths.add(System.getProperty("java.library.path"));
        }
……

}
```

- createOrUpdateClassLoaderLocked用来创建类加载器
- makePaths确定lib的search路径。对于isBundledApp的App，将会包含虚拟机的系统属性java.library.path所对应的路径
- java.library.path 在makePaths方法中会获取java.library.path的值，那么这个值是怎么确定下来的呢?
  - libcore/ojluni/src/main/java/java/lang/System.java中initUnchangeableSystemProperties方法会设置系统属性
  - 在initUnchangeableSystemProperties中会调用private static native String[] specialProperties()获取特定属性的值
  - specialProperties方法由native实现，在libcore/ojluni/src/main/native/System.c中有如下代码片段:

```
static jobjectArray System_specialProperties(JNIEnv* env, jclass ignored) {
    const char* library_path = getenv("LD_LIBRARY_PATH");
#if defined(__ANDROID__)
    if (library_path == NULL) {
        android_get_LD_LIBRARY_PATH(path, sizeof(path));
        library_path = path;
    }
#endif
}
```

  - android_get_LD_LIBRARY_PATH方法在bionic 中实现，其返回的结果为默认的链接路径：

```
bionic/linker/linker.cpp

#if defined(__LP64__)
static const char* const kSystemLibDir     = "/system/lib64";
static const char* const kVendorLibDir     = "/vendor/lib64";
static const char* const kAsanSystemLibDir = "/data/asan/system/lib64";
static const char* const kAsanVendorLibDir = "/data/asan/vendor/lib64";
#else
static const char* const kSystemLibDir     = "/system/lib";
```

```
static const char* const kVendorLibDir     = "/vendor/lib";
static const char* const kAsanSystemLibDir = "/data/asan/system/lib";
static const char* const kAsanVendorLibDir = "/data/asan/vendor/lib";
#endif

static const char* const kAsanLibDirPrefix = "/data/asan";

static const char* const kDefaultLdPaths[] = {
  kSystemLibDir,
  kVendorLibDir,
  nullptr
};

static const char* const kAsanDefaultLdPaths[] = {
  kAsanSystemLibDir,
  kSystemLibDir,
  kAsanVendorLibDir,
  kVendorLibDir,
  nullptr
};
```

所以对于Bundled App，其jni的库链接路径是包括/system/lib64和/vendor/lib64。

- 非Bundled App

经过上面的分析，对于非Bundled App，也就是普通的后装App，其jni的链接路径仅仅包含应用自己安装目录的jni库路径，并不包含系统库路径