基 础 篇

本篇是全书的基础知识讲解，共包括 3 章。

第 1 章主要介绍 Android 虚拟机的技术原理，重点学习 Dalvik 虚拟机和 ART 虚拟机；第 2 章主要介绍 Native 开发和 ARM 汇编语言，学习移动应用和原生层的交互机制；第 3 章主要介绍 iOS 相关的基础知识，重点学习 iOS 软件包结构以及 iOS 应用的启动和运行流程。

本篇作为全书的基础，主要对主流的移动端操作系统（Android 与 iOS）的核心原理进行剖析，对移动应用从应用层到原生层的运行机制进行了全面的梳理，使读者掌握移动安全攻防的基础知识，为后续理论篇中对核心理论的讲解打下基础。

第 1 章 Android 虚拟机

CHAPTER 1

Android 系统是开源项目，攻防双方可以直接从系统底层的源码内找到可以利用的关键点，因此本章将会围绕 Android 虚拟机的演化、功能以及虚拟机运行应用代码等方面，帮助读者更好地了解 Android 虚拟机的结构与功能。

1.1 Dalvik 虚拟机


视频讲解

Dalvik 虚拟机由 Dan Bornstein 开发，起源于 Apache Harmony 项目。Apache Harmony 项目是根据 Apache License v2 协议发布的，由 Apache 软件基金会主导，目标是实现一个独立的、兼容 JDK 5 的虚拟机。

Google 公司在 2007 年年底正式发布了 Android SDK，同时，作为 Android 系统的支撑，Dalvik 虚拟机的存在也第一次被人们知晓。一方面，Dalvik 虚拟机对内存的高效使用，以及在低速 CPU 上表现出的高性能，令人印象深刻。另一方面，虚拟机底层的操作系统与 Posix 兼容，使虚拟机可以简单地完成进程隔离和线程管理。每一个 Android 应用在系统底层都有一个对应的 Dalvik 虚拟机实例，Android 应用的代码在虚拟机的解释下得以执行。

1.1.1 DVM 的特点


视频讲解

与在桌面系统和服务器上运行的虚拟机相比，Dalvik 虚拟机不需要很快的 CPU 速度和大量的内存空间，因此非常适合运行在移动设备上。根据 Google 公司的测算，64MB 的 RAM 就足以支撑系统的正常运转。其中，24MB 用于底层系统的初始化和启动，20MB 用于启动高层服务，剩余的 20MB 可用于应用运行。但是随着系统服务的增多和应用功能的扩展，虚拟机运行时所消耗的内存也一定会越来越大。

Dalvik 虚拟机与 Java 虚拟机最显著的区别是，两者具有不同的类文件格式以及指令集。Dalvik 虚拟机使用的是 Dex(Dalvik Executable)格式的类文件，而 Java 虚拟机使用的是 Class 格式的类文件。一个 Dex 文件可以包含若干 Java 类，而一个 Class 文件只能包括一个 Java 类。Dex 文件内部包含的各个 Java 类中重复的字符串和常数只需要保存一次，从而节省了空间，适合在内存和处理器速度有限的移动设备中使用。一般来说，包含有相同类的未压缩 Dex 文件的体积比一个已经压缩的 Jar 文件的体积要稍微小一些。

Dalvik 虚拟机使用的指令是基于寄存器的，Java 虚拟机使用的指令集是基于堆栈的。基于堆栈的指令很紧凑，而基于寄存器的指令由于需要指定源地址和目标地址，因此需要占

用更多的指令空间。Java 虚拟机使用的指令只占用一个字节，Dalvik 虚拟机的某些指令需要占用两个字节。基于堆栈和基于寄存器的指令集各有优劣，一般而言，如果执行同样的功能，Java 虚拟机需要更多的指令，这意味着要占用更多的 CPU 时间；而 Dalvik 虚拟机需要更多的指令空间，这意味着数据缓冲更容易失效。

基于寄存器的指令对目标机器的寄存器进行了假设，所以它更有利于进行 AOT 优化。所谓 AOT(Ahead Of Time)，是指在解释语言指令运行之前，先将它编译成本地机器语言指令，所以 AOT 本质上是一种静态编译(在程序运行前进行编译)。与之相对的是 JIT(在程序运行时进行编译)。JIT 可以利用程序在运行时产生的信息得到比较优化的代码，但是因为优化的过程太耗时了，不能进行某些高级优化。另一方面，AOT 不占用程序运行时间，因此就可以不计时间成本来优化代码。无论是 AOT，还是 JIT，它们的最终的目标都是将解释语言编译为本地机器语言，机器语言都是基于寄存器来执行的。因此，从某种角度上说，基于寄存器的指令更有利于进行编译以及优化。

以上是 DVM 与 JVM 之间的区别，接下来介绍 DVM 的其他特征。

1. 内存管理

Dalvik 虚拟机的内存大体上可以分为 Java Object Heap、Bitmap Memory 和 Native Heap 这 3 种。

Java Object Heap 用来分配 Java 对象，Java 代码中通过 new 关键字创建出来的类对象都位于 Java Object Heap 上。Dalvik 虚拟机启动时，可以通过-Xms 和-Xmx 选项指定 Java Object Heap 的最小值和最大值。为了避免 Dalvik 虚拟机在运行的过程中对 Java Object Heap 的大小进行调整而影响性能，通常会通过-Xms 和-Xmx 选项将 Java Object Heap 的最小值和最大值设置为相等。

Bitmap Memory 也称为 External Memory，用来处理图像。在 HoneyComb 之前，Bitmap Memory 是在 Native Heap 中分配的，但是这部分内存同样计入 Java Object Heap 中，所以 Bitmap 占用的内存和 Java Object 占用的内存加起来不能超过 Java Object Heap 的最大值。

Native Heap 就是在 Native Code 中使用 malloc()函数分配出来的内存，这部分内存不受 Java Object Heap 大小的限制，而是由系统进行限制。需要注意的是，不要因为 Native Heap 不受 Java Object Heap 的限制就滥用，滥用 Native Heap 会导致系统可用内存急剧减少，系统会采取激进的措施来强制停止某些进程，用来补充可用内存，这样会影响 Android 系统的用户体验。

2. 垃圾收集

Dalvik 虚拟机可以自动回收那些不再被引用的 Java 对象。垃圾自动收集机制将开发者从内存问题中解放出来，极大地提高了开发效率和程序的可维护性。

在 Android 2.3 以后的版本中，垃圾收集机制的特性为：

(1) 在大多数情况下，垃圾收集线程与其他线程是并发执行的；

(2) 一次可能只收集一部分垃圾；

(3) 一次垃圾收集造成的程序中止时间通常都小于 5ms。

3. 即时编译

即时(Just In Time，JIT)编译与运行前(Ahead Of Time，AOT)编译的概念相对应，是

指解释语言的代码在程序运行的过程中进行编译。即时编译的优点是编译器可以利用程序运行时产生的信息对编译出来的代码进行优化，缺点是占用程序的运行时间。

为了解决占用运行时间的问题，JIT 只会选择那些热点代码进行编译或者优化。根据二八原则，运行一个程序时，80％的时间可能都是在重复执行该程序中 20％的代码。因此，JIT 就可以选择这 20％经常执行的代码来进行编译和优化。

4. Java 本地调用(JNI)

JNI(Java Native Interface，Java 本地调用)由 SUN 公司发布，该方案用于将 Java 与 C/C++代码进行集成。Java 语言可以通过 JNI 与 C/C++代码进行交互，但是由于 C/C++代码依赖于处理器平台，所以它在一定程度上降低了 Java 代码的可移植性。

由于 Dalvik 虚拟机是运行在目标机器上的，所以有些功能需要调用底层 Linux 系统的接口，这时就需要一种机制，使得方法调用可以从 Java 层穿越到 Native 层，也就是 C/C++层，这个机制就是 JNI。JNI 机制是双向的——既可以在 Java 方法中调用 C/C++函数，也可以在 C/C++函数中调用 Java 方法。

事实上，Dalvik 虚拟机提供 Java 运行时库，大部分都是通过调用 Linux 系统接口来实现的。例如，当调用 android.os.Process 类的成员函数 start()来创建一个进程时，最终会调用到 Linux 系统提供的 fork 系统来创建一个进程。

同时，为了方便开发者使用 C/C++语言开发应用程序，Android 官方提供了 NDK。通过 NDK，开发者可以在 Android 应用中编写 C/C++代码。

5. 进程与线程管理

一般来说，虚拟机的进程和线程与机器本地操作系统的进程和线程一一对应，这样就可以由本地操作系统来对进程和线程进行调度。进程和线程调度是操作系统的核心模块，它的实现是非常复杂的，尤其是考虑到多核的情况，所以没有必要在虚拟机中提供进程和线程库，可以借用底层系统的进程和线程机制。

Dalvik 虚拟机运行在 Linux 操作系统之上。在 Linux 操作系统中没有线程的概念，如果两个进程共享同一个地址空间，则可以认为它们是同一个进程的两个线程。Linux 操作系统提供了 fork()和 clone()两个函数，其中，fork()用来创建进程，clone()用来创建线程。

每一个 Android 应用程序进程都有一个 Dalvik 虚拟机实例。这样，Android 应用程序进程之间不会相互影响，如果一个 Android 应用程序进程意外中止，不会影响到其他 Android 应用程序进程的正常运行。

每一个 Android 应用程序进程都是被 Zygote 进程调用 fork()函数创建出来的，而 Zygote 进程是由 init 进程在系统启动阶段启动的。Zygote 进程在启动的时候，会创建一个虚拟机实例，并且在这个虚拟机实例中加载所有的 Java 核心库。每当 Zygote 进程调用 fork()函数创建一个新的 Android 应用程序进程的时候，虚拟机实例就可以复制自身。这样，被孵化出来的 Android 应用程序进程，一方面复制了 Zygote 进程中的虚拟机实例，另一方面可以和 Zygote 进程共享同一套 Java 核心库。这样不仅加快了 Android 应用程序的进程创建过程，而且由于所有的 Android 应用程序进程都共享同一套 Java 核心库，还可以节省内存空间。

1.1.2 DVM 虚拟机启动流程


视频讲解

1.1.1 节提到，在 Android 系统中，应用程序进程都是由 Zygote 进程孵化出来的。

Zygote 进程在启动时会创建一个 Dalvik 虚拟机实例，每当它孵化一个新的应用程序进程时，都会将这个 Dalvik 虚拟机实例复制到新的应用程序进程中，从而使得每一个应用程序进程都有一个独立的 Dalvik 虚拟机实例。本节将分析 Dalvik 虚拟机在 Zygote 进程中的启动过程。

Zygote 进程在启动的过程中，除了创建一个 Dalvik 虚拟机实例之外，还会将 Java 运行时库加载到进程中，以及将一些 Android 核心类的 JNI 方法注册到前面创建的 Dalvik 虚拟机实例中去。这里需要注意，当一个应用程序进程被 Zygote 进程孵化出来时，不仅会获得 Zygote 进程中的 Dalvik 虚拟机实例副本，还会与 Zygote 进程一起共享 Java 运行时库，这完全得益于 Linux 内核的进程创建机制。这种 Zygote 孵化机制的优点是快速地启动一个应用程序进程，同时节省整体的内存消耗。但是孵化机制的缺点是会影响系统的开机速度，因为 Zygote 进程是在系统开机的过程中启动的。总体来说，孵化机制是利大于弊的，整个系统只有一个 Zygote 进程，但是可能会有无数个应用程序进程被孵化，而且手机在绝大多数时间内都是处于息屏休眠的状态，只在极少情况下需要重新启动手机。

接下来结合 Android 4.4 源码分析 DVM 虚拟机启动过程。

1. AndroidRuntime::start()

这个函数定义在文件 frameworks/base/core/jni/AndroidRuntime.cpp 中，属于 AndroidRuntime 类的成员函数。该函数主要完成以下 4 个任务。

(1) 调用 AndroidRuntime 类的另一个成员函数 startVm()来创建一个 DVM 实例，并且保存在变量 mJavaVM 中。

```
void AndroidRuntime::start(const char * className, const char * options)
{
…
    if (startVm(&mJavaVM, &env) != 0) {
        return;
    }
    onVmCreated(env);
…
}
```

(2) 调用 AndroidRuntime 类的成员函数 startReg()来注册部分 Android 核心类的 JNI 方法。

```
void AndroidRuntime::start(const char * className, const char * options)
{
…
    onVmCreated(env);
    /*
     * Register Android functions.
     */
    if (startReg(env) < 0) {
        ALOGE("Unable to register all android natives\n");
        return;
    }
…
}
```

(3) 调用参数 className 所描述的一个 Java 类的 main()函数,来初始化 Zygote 进程,className 所表示的类就是 Zygote Init。该类的 main()函数会加载大量的 Android 核心类和系统资源文件。此时,Zygote 进程中的 Dalvik 虚拟机实例就开始正式运作了。其中,预加载的 Android 核心类可以参考 frameworks/base/preloaded-classes,而预加载的系统资源就包含在/system/framework/framework-res.apk 中。

```
void AndroidRuntime::start(const char * className, const char * options)
{
…
    char * slashClassName = toSlashClassName(className);
    jclass startClass = env->FindClass(slashClassName);
    if (startClass == NULL) {
        ALOGE("JavaVM unable to locate class '%s'\n", slashClassName);
        /* keep going */
    } else {
        jmethodID startMeth = env->GetStaticMethodID(startClass, "main",
            "([Ljava/lang/String;)V");
        if (startMeth == NULL) {
            ALOGE("JavaVM unable to find main() in '%s'\n", className);
            /* keep going */
        } else {
            env->CallStaticVoidMethod(startClass, startMeth, strArray);
        }
    }
free(slashClassName);
…
}
```

(4) 当完成对 Zygote Init 类的 main()函数的执行,并完成 Android 核心类和系统资源文件的预加载后,Zygote 进程就准备退出了。在退出之前,Zygote 进程会调用两个前面创建的 Dalvik 虚拟机实例的成员函数:

① DetachCurrentThread()用来将 Zygote 的主线程脱离前面创建的 Dalvik 虚拟机实例;

② DestroyJavaVM()用来销毁前面创建的 Dalvik 虚拟机实例。

```
void AndroidRuntime::start(const char * className, const char * options)
{
…
    ALOGD("Shutting down VM\n");
    if (mJavaVM->DetachCurrentThread() != JNI_OK)
        ALOGW("Warning: unable to detach main thread\n");
    if (mJavaVM->DestroyJavaVM() != 0)
        ALOGW("Warning: VM did not shut down cleanly\n");
}
```

2. AndroidRuntime::startVm()

在启动 Dalvik 虚拟机的时候,可以通过特定的系统属性来指定一系列选项,以实现不同的功能。下面简要介绍几个可能有用的选项。

(1) -Xcheck:jni：用来启动 JNI 方法检查。在 C/C++代码中，可以修改 Java 对象的成员变量或者调用 Java 对象的成员函数。加了-Xcheck:jni 选项之后，就可以对要访问的 Java 对象的成员变量或者成员函数进行合法性检查，例如，检查类型是否匹配。可以通过 dalvik.vm.checkjni 或者 ro.kernel.android.checkjni 这两个系统属性来指定是否要启用 -Xcheck:jni 选项。注意：加了-Xcheck:jni 选项之后，会使得 JNI 方法执行速度变慢。

```
int AndroidRuntime::startVm(JavaVM ** pJavaVM, JNIEnv ** pEnv, bool zygote)
{
…
    ALOGV("CheckJNI is %s\n", checkJni ? "ON" : "OFF");
    if (checkJni) {
        /* extended JNI checking */
        addOption("-Xcheck:jni");

        /* with -Xcheck:jni, this provides a JNI function call trace */
        //addOption("-verbose:jni");
    }
…
}
```

(2) -Xint:portable、-Xint:fast、-Xint:jit：用来指定 Dalvik 虚拟机的执行模式。Dalvik 虚拟机支持 3 种运行模式，分别是 Portable、Fast 和 Jit。Portable 是指 Dalvik 虚拟机以可移植的方式来进行编译，也就是说，编译出来的虚拟机可以在任意平台上运行。Fast 是针对当前平台对 Dalvik 虚拟机进行编译，这样编译出来的 Dalvik 虚拟机可以进行特殊的优化，从而使得它能更快地运行程序。Jit 不是解释执行代码，而是将代码动态编译成本地语言后再执行。可以通过 dalvik.vm.execution-mode 系统属性来指定 Dalvik 虚拟机的解释模式。

```
int AndroidRuntime::startVm(JavaVM ** pJavaVM, JNIEnv ** pEnv, bool zygote)
{
…
    if (executionMode == kEMIntPortable) {
        addOption("-Xint:portable");
    } else if (executionMode == kEMIntFast) {
        addOption("-Xint:fast");
    } else if (executionMode == kEMJitCompiler) {
        addOption("-Xint:jit");
    }
…
}
```

(3) -Xstacktracefile：用来指定调用堆栈输出文件。Dalvik 虚拟机接收到 SIGQUIT (Ctrl-\或者 kill -3)信号之后，会将所有线程的调用堆栈输出(默认输出到日志中)。指定了-Xstacktracefile 选项之后，就可以将线程的调用堆栈输出到指定的文件中。通过 dalvik.vm.stack-trace-file 系统属性可以指定调用堆栈输出文件。

```
int AndroidRuntime::startVm(JavaVM ** pJavaVM, JNIEnv ** pEnv, bool zygote)
{
    …
```

```
    // If dalvik.vm.stack-trace-dir is set, it enables the "new" stack trace
    // dump scheme and a new file is created for each stack dump. If it isn't set,
    // the old scheme is enabled.
    property_get("dalvik.vm.stack-trace-dir", propBuf, "");
    if (strlen(propBuf) > 0) {
        addOption("-Xusetombstonedtraces");
    } else {
        parseRuntimeOption("dalvik.vm.stack-trace-file", stackTraceFileBuf,
"-Xstacktracefile:");
    }
…
}
```

(4) -Xmx：用来指定 Java 对象堆的最大值。Dalvik 虚拟机的 Java 对象堆的默认最大值是 16M，也可以通过 dalvik.vm.heapsize 系统属性来指定为其他值。

设置好 Dalvik 虚拟机的启动选项之后，AndroidRuntime 的成员函数 startVm()就会调用另外一个函数 JNI_CreateJavaVM()来创建以及初始化一个 Dalvik 虚拟机实例。

```
int AndroidRuntime::startVm(JavaVM** pJavaVM, JNIEnv** pEnv, bool zygote)
{
    …
    parseCompilerRuntimeOption("dalvik.vm.image-dex2oat-Xmx", dex2oatXmxImageFlagsBuf,
"-Xmx", "-Ximage-compiler-option");
    …
    // Extra options for DexClassLoader.
    parseCompilerRuntimeOption("dalvik.vm.dex2oat-Xms", dex2oatXmsFlagsBuf, "-Xms",
"-Xcompiler-option");
    parseCompilerRuntimeOption("dalvik.vm.dex2oat-Xmx", dex2oatXmxFlagsBuf, "-Xmx",
"-Xcompiler-option");
    …
}
```

3. JNI_CreateJavaVM()

```
jint JNI_CreateJavaVM(JavaVM** p_vm, JNIEnv** p_env, void* vm_args) {
    …
    /*
     * Set up structures for JNIEnv and VM.
     */
    JavaVMExt* pVM = (JavaVMExt*) calloc(1, sizeof(JavaVMExt));
    pVM->funcTable = &gInvokeInterface;
    pVM->envList = NULL;
    dvmInitMutex(&pVM->envListLock);

    UniquePtr<const char*[]> argv(new const char*[args->nOptions]);
    memset(argv.get(), 0, sizeof(char*) * (args->nOptions));
    …
    /*
     * Create a JNIEnv for the main thread.  We need to have something set up
     * here because some of the class initialization we do when starting
     * up the VM will call into native code.
     */
```

```
    JNIEnvExt * pEnv = (JNIEnvExt * ) dvmCreateJNIEnv(NULL);

    /* Initialize VM. */
    gDvm.initializing = true;
    std::string status = dvmStartup(argc, argv.get(), args->ignoreUnrecognized, (JNIEnv * )pEnv);
    gDvm.initializing = false;
…
    /*
     * Success!  Return stuff to caller.
     */
    dvmChangeStatus(NULL, THREAD_NATIVE);
    *p_env = (JNIEnv * ) pEnv;
    *p_vm = (JavaVM * ) pVM;
    ALOGV("CreateJavaVM succeeded");
    return JNI_OK;
}
```

JNI_CreateJavaVM()主要完成以下 4 项工作。

(1) 为当前进程创建一个 Dalvik 虚拟机实例,即一个 JavaVMExt 对象。

(2) 为当前线程创建和初始化一个 JNI 环境,即一个 JNIEnvExt 对象,这是通过调用函数 dvmCreateJNIEnv()来完成的。

(3) 将参数 vm_args 所描述的 Dalvik 虚拟机启动选项复制到变量 argv 所描述的一个字符串数组中去,并且调用函数 dvmStartup()来初始化前面所创建的 Dalvik 虚拟机实例。

(4) 调用函数 dvmChangeStatus()将当前线程的状态设置为正在执行 Native 代码,并且将前面创建和初始化好的 JavaVMExt 对象和 JNIEnvExt 对象通过输出参数 p_vm 和 p_env 返回给调用者。

gDvm 是一个类型为 DvmGlobals 的全局变量,用来收集当前进程中所有与虚拟机相关的信息,其中,它的成员变量 vmList 指向的就是当前进程中的 Dalvik 虚拟机实例,即一个 JavaVMExt 对象。以后每当需要访问当前进程中的 Dalvik 虚拟机实例时,就可以通过全局变量 gDvm 的成员变量 vmList 来获得,避免了在函数之间传递该 Dalvik 虚拟机实例。

每一个 Dalvik 虚拟机实例都有一个函数表,保存在对应的 JavaVMExt 对象的成员变量 funcTable 中,而这个函数表又被指定为 gInvokeInterface。gInvokeInterface 是一个类型为 JNIInvokeInterface 的结构体,它定义在文件 dalvik/vm/Jni.c 中,如下所示。

```
static const struct JNIInvokeInterface gInvokeInterface = {
    NULL,
    NULL,
    NULL,

    DestroyJavaVM,
    AttachCurrentThread,
    DetachCurrentThread,

    GetEnv,

    AttachCurrentThreadAsDaemon,
};
```

有了Dalvik虚拟机函数表之后，就可以将当前线程附加到Dalvik虚拟机中或者从Dalvik虚拟机分离或者销毁当前进程的Dalvik虚拟机等。

4. dvmCreateJNIEnv()

```
JNIEnv * dvmCreateJNIEnv(Thread * self) {
    JavaVMExt * vm = (JavaVMExt * ) gDvmJni.jniVm;

    //if (self != NULL)
    //     ALOGI("Ent CreateJNIEnv: threadid = %d %p", self->threadId, self);

    assert(vm != NULL);

    JNIEnvExt * newEnv = (JNIEnvExt * ) calloc(1, sizeof(JNIEnvExt));
    newEnv->funcTable = &gNativeInterface;
    if (self != NULL) {
        dvmSetJniEnvThreadId((JNIEnv * ) newEnv, self);
        assert(newEnv->envThreadId != 0);
    } else {
        /* make it obvious if we fail to initialize these later */
        newEnv->envThreadId = 0x77777775;
        newEnv->self = (Thread * ) 0x77777779;
    }
    if (gDvmJni.useCheckJni) {
        dvmUseCheckedJniEnv(newEnv);
    }

    ScopedPthreadMutexLock lock(&vm->envListLock);

    /* insert at head of list */
    newEnv->next = vm->envList;
    assert(newEnv->prev == NULL);
    if (vm->envList == NULL) {
        // rare, but possible
        vm->envList = newEnv;
    } else {
        vm->envList->prev = newEnv;
    }
    vm->envList = newEnv;

    //if (self != NULL)
    //    ALOGI("Xit CreateJNIEnv: threadid = %d %p", self->threadId, self);
    return (JNIEnv * ) newEnv;
}
```

函数dvmCreateJNIEnv()主要执行了以下3个操作。

(1) 创建一个JNIEnvExt对象，用来描述一个JNI环境，并且设置这个JNIEnvExt对象的宿主Dalvik虚拟机，以及所使用的本地接口表，即设置这个JNIEnvExt对象的成员变量funcTable和vm。这里的宿主Dalvik虚拟机即为当前进程的Dalvik虚拟机，它保存在全局变量gDvm的成员变量vmList中。本地接口表由全局变量gNativeInterface来描述。

(2) 参数self描述的是前面创建的JNIEnvExt对象要关联的线程，可以通过调用函数

dvmSetJniEnvThreadId()来将它们关联起来。注意,当参数 self 的值等于 NULL 的时候,就表示前面的 JNIEnvExt 对象是要与主线程关联的,但是要等到后面再关联,因为现在用来描述主线程的 Thread 对象还没有准备好。通过将一个 JNIEnvExt 对象的成员变量 envThreadId 和 self 的值分别设置为 0x77777775 和 0x77777779 来表示它还没有与线程关联。

(3) 在一个 Dalvik 虚拟机中可以运行多个线程。所有关联有 JNI 环境的线程都有一个对应的 JNIEnvExt 对象,这些 JNIEnvExt 对象相互连接在一起,保存在用于描述其宿主 Dalvik 虚拟机的一个 JavaVMExt 对象的成员变量 envList 中。因此,前面创建的 JNIEnvExt 对象需要连接到其宿主 Dalvik 虚拟机的 JavaVMExt 链表中。

gNativeInterface 是一个类型为 JNINativeInterface 的结构体,它定义在文件 dalvik/vm/Jni.cpp 中。

```
static const struct JNINativeInterface gNativeInterface = {

    GetVersion,

    DefineClass,
    FindClass,

    FromReflectedMethod,
    FromReflectedField,
    ToReflectedMethod,

    …

    NewDirectByteBuffer,
    GetDirectBufferAddress,
    GetDirectBufferCapacity,

    GetObjectRefType
};
```

5. dvmStartup()

第 4 步执行完成之后,返回到前面的 JNI_CreateJavaVM 中,接下来就会继续调用函数 dvmStartup()来初始化前面所创建的 Dalvik 虚拟机实例。

这个函数定义在文件 dalvik/vm/Init.cpp 中,用来初始化 Dalvik 虚拟机。

```
std::string dvmStartup(int argc, const char * const argv[], bool ignoreUnrecognized, JNIEnv *
pEnv)
{
    ScopedShutdown scopedShutdown;

    assert(gDvm.initializing);

    ALOGV("VM init args (%d):", argc);
    for (int i = 0; i < argc; i++) {
        ALOGV("   %d: '%s'", i, argv[i]);
    }
```

```
    setCommandLineDefaults();

    /*
     * Process the option flags (if any).
     */
    int cc = processOptions(argc, argv, ignoreUnrecognized);
    if (cc != 0) {
        if (cc < 0) {
            dvmFprintf(stderr, "\n");
            usage("dalvikvm");
        }
        return "syntax error";
    }
    …
}
```

这段代码用来处理 Dalvik 虚拟机的启动选项，这些启动选项保存在参数 argv 中，并且个数等于 argc。在处理这些选项之前，先调用函数 setCommandLineDefaults()来给 Dalvik 虚拟机设置默认参数，因为启动选项不一定会指定 Dalvik 虚拟机的所有属性。之后就可以调用函数 ProcessOptions()来处理参数 argv 和 argc 所描述的启动选项了，也就是根据这些选项值来设置 Dalvik 虚拟机的属性，例如，设置 Dalvik 虚拟机的 Java 对象堆的最大值。

```
std::string dvmStartup(int argc, const char* const argv[], bool ignoreUnrecognized, JNIEnv*
pEnv)
{
    …
    /* configure signal handling */
    if (!gDvm.reduceSignals)
        blockSignals();
    …
}
```

如果没有在 Dalvik 虚拟机的启动选项中指定-Xrs，那么 gDvm.reduceSignals 的值就会被设置为 false，表示要在当前线程中屏蔽掉 SIGQUIT 信号。在这种情况下，会有一个线程专门用来处理 SIGQUIT 信号。这个线程在接收到 SIGQUIT 信号的时候，就会将各个线程的调用堆栈打印出来。因此，这个线程又称为 dump-stack-trace 线程。

```
std::string dvmStartup(int argc, const char* const argv[], bool ignoreUnrecognized, JNIEnv*
pEnv)
{
    …
    /*
     * Initialize components.
     */
    dvmQuasiAtomicsStartup();
    /*
     * 初始化 Davlik 虚拟机的对象分配记录子模块，可以通过 DDMS 工具查看
     * Davlik 虚拟机的对象分配情况
     */
    if (!dvmAllocTrackerStartup()) {
```

```
        return "dvmAllocTrackerStartup failed";
    }
    /*
     * 用来初始化 Davlik 虚拟机的垃圾收集(GC)子模块
     */
    if (!dvmGcStartup()) {
        return "dvmGcStartup failed";
    }
    /*
     * 用来初始化 Davlik 虚拟机的线程列表,为主线程创建一个 Thread 对象
     */
    if (!dvmThreadStartup()) {
        return "dvmThreadStartup failed";
    }
    /*
     * 初始化 Davlik 虚拟机的内建 Native 函数表
     * 内建 Native 函数替换了一些 Java 类某些成员函数的实现
     */
    if (!dvmInlineNativeStartup()) {
        return "dvmInlineNativeStartup";
    }
    /*
     * 初始化寄存器映射集(Register Map)子模块
     * 用来辅助 DVM 进行精确垃圾收集
     */
    if (!dvmRegisterMapStartup()) {
        return "dvmRegisterMapStartup failed";
    }
    /*
     * 用来初始化 instanceof 操作符子模块
     */
    if (!dvmInstanceofStartup()) {
        return "dvmInstanceofStartup failed";
    }
    /*
     * 初始化启动类加载器
     */
    if (!dvmClassStartup()) {
        return "dvmClassStartup failed";
    }
    …
    /*
     * 查找必要的类和成员函数
     */
    if (!dvmFindRequiredClassesAndMembers()) {
        return "dvmFindRequiredClassesAndMembers failed";
    }
    /*
     * 用来初始化 java.lang.String 类内部的私有字符串池
     */
    if (!dvmStringInternStartup()) {
        return "dvmStringInternStartup failed";
    }
    /*
```

```
        * 用来初始化 Native Shared Object 库加载表,也就是 SO 库加载表
        */
       if (!dvmNativeStartup()) {
            return "dvmNativeStartup failed";
       }
       /*
        * 初始化一个内部 Native 函数表
        * 所有需要直接访问 Dalvik 虚拟机内部函数或者数据结构的 Native 函数
        * 都定义在这张表中
        */
       if (!dvmInternalNativeStartup()) {
            return "dvmInternalNativeStartup failed";
       }
       /*
        * 初始化全局引用表,以及加载一些与 Direct Buffer 相关的类
        */
       if (!dvmJniStartup()) {
            return "dvmJniStartup failed";
       }
       /*
        * 用来初始化 Dalvik 虚拟机的性能分析子模块,
        * 以及加载 dalvik.system.VMDebug 类等
        */
       if (!dvmProfilingStartup()) {
            return "dvmProfilingStartup failed";
       }
       …
}
```

这段代码初始化了 DVM 的一些子模块。到这一步,系统就被初始化了,gDvm 实例中已经包含了 DVM 需要的全部类和类成员的引用。在各个子模块初始化完成之后,Dalvik 虚拟机继续执行其他的初始化和检查工作。

```
std::string dvmStartup(int argc, const char* const argv[],
            bool ignoreUnrecognized, JNIEnv* pEnv)
{
       …
       /*
        * 创建一个函数表,将表中的函数内联化以提升性能
        */
       if (!dvmCreateInlineSubsTable()) {
            return "dvmCreateInlineSubsTable failed";
       }

       /*
        * 验证 Dalvik 虚拟机中存在相应的装箱类
        * 比如 Boolean、Integer、Byte
        */
       if (!dvmValidateBoxClasses()) {
            return "dvmValidateBoxClasses failed";
       }
```

```
/*
 * 用来准备主线程的 JNI 环境
 * JNI_CreateJavaVM()函数中创建的 JNI 环境在这里与主进程关联
 */
if (!dvmPrepMainForJni(pEnv)) {
      return "dvmPrepMainForJni failed";
}

/*
 * 显式初始化 java.lang.Class,且必须在注册 JNI 方法之前
 * 因为如果被注册的类没有被初始化,则会抛出断言
 */
if (!dvmInitClass(gDvm.classJavaLangClass)) {
      return "couldn't initialized java.lang.Class";
}

/*
 * 为 Java 核心类注册 JNI 方法
 */
if (!registerSystemNatives(pEnv)) {
      return "couldn't register system natives";
}

/*
 * 创建一些与内存分配有关的 Exception 对象,并缓存,以方便将来使用
 */
if (!dvmCreateStockExceptions()) {
      return "dvmCreateStockExceptions failed";
}

/*
 * 为主线程创建 ThreadGroup 对象,Thread 对象和 VMThread 对象
 */
if (!dvmPrepMainThread()) {
      return "dvmPrepMainThread failed";
}

/*
 * 确保主线程在不引用任何 Java 对象的情况下执行程序入口
 */
if (dvmReferenceTableEntries(&dvmThreadSelf()->internalLocalRefTable) != 0)
{
      ALOGW("Warning: tracked references remain post-initialization");
      dvmDumpReferenceTable(&dvmThreadSelf()->internalLocalRefTable, "MAIN");
}

/*
 * 初始化 DVM 的调试环境
 */
if (!dvmDebuggerStartup()) {
      return "dvmDebuggerStartup failed";
}

if (!dvmGcStartupClasses()) {
```

```
            return "dvmGcStartupClasses failed";
        }
        …
}
```

接下来的代码执行最后的初始化工作。

```
std::string dvmStartup(int argc, const char * const argv[],
            bool ignoreUnrecognized, JNIEnv * pEnv)
{
        …
        /*
          * 完成 DVM 的最后一步的初始化工作,检查 DVM 是否指定了 -Xzygote 启动选项
          * 如果指定了,则说明当前是在 Zygote 进程中启动 DVM,
          * 调用函数 dvmInitZygote()
          * 否则调用 dvmInitAfterZygote()来执行最后一步的初始化工作
          */
         if (gDvm.zygote) {
            if (!initZygote()) {
                return "initZygote failed";
            }
         } else {
            if (!dvmInitAfterZygote()) {
                return "dvmInitAfterZygote failed";
            }
        }
        …
}
```

这部分代码负责 DVM 的初始化收尾工作,它会根据 DVM 虚拟机是否指定了-Xzygote 选项来判断是否是在 Zygote 进程中启动的 Dalvik 虚拟机,如果处于 Zygote 进程中,则调用函数 initZygote()来执行最后一步的初始化工作。如果不是处于 Zygote 进程中,则会调用另外一个函数 dvmInitAfterZygote()来执行最后一步的初始化工作。

6. initZygote()

```
static bool initZygote()
{
        /* zygote goes into its own process group */
        setpgid(0,0);

        // See storage config details at http://source.android.com/tech/storage/
        // Create private mount namespace shared by all children
        if (unshare(CLONE_NEWNS) == -1) {
            SLOGE("Failed to unshare(): %s", strerror(errno));
            return -1;
        }

        // Mark rootfs as being a slave so that changes from default
        // namespace only flow into our children.
        if (mount("rootfs", "/", NULL, (MS_SLAVE | MS_REC), NULL) == -1) {
            SLOGE("Failed to mount() rootfs as MS_SLAVE: %s", strerror(errno));
```

```
            return -1;
        }

        // Create a staging tmpfs that is shared by our children; they will
        // bind mount storage into their respective private namespaces, which
        // are isolated from each other.
        const char* target_base = getenv("EMULATED_STORAGE_TARGET");
        if (target_base != NULL) {
            if (mount("tmpfs", target_base, "tmpfs", MS_NOSUID | MS_NODEV,
                      "uid=0,gid=1028,mode=0751") == -1) {
                SLOGE("Failed to mount tmpfs to %s: %s", target_base, strerror(errno));
                return -1;
            }
        }

        // Mark /system as NOSUID | NODEV
        const char* android_root = getenv("ANDROID_ROOT");

        if (android_root == NULL) {
            SLOGE("environment variable ANDROID_ROOT does not exist?!?!");
            return -1;
        }

        std::string mountDev(getMountsDevDir(android_root));
        if (mountDev.empty()) {
            SLOGE("Unable to find mount point for %s", android_root);
            return -1;
        }

        if (mount(mountDev.c_str(), android_root, "none",
                  MS_REMOUNT | MS_NOSUID | MS_NODEV | MS_RDONLY | MS_BIND, NULL) == -1) {
            SLOGE("Remount of %s failed: %s", android_root, strerror(errno));
            return -1;
        }

    #ifdef HAVE_ANDROID_OS
        if (prctl(PR_SET_NO_NEW_PRIVS, 1, 0, 0, 0) < 0) {
            // Older kernels don't understand PR_SET_NO_NEW_PRIVS and return
            // EINVAL. Don't die on such kernels.
            if (errno != EINVAL) {
                SLOGE("PR_SET_NO_NEW_PRIVS failed: %s", strerror(errno));
                return -1;
            }
        }
    #endif

        return true;
    }
```

这一步执行完成之后，Dalvik 虚拟机的创建和初始化工作就完成了，回到 AndroidRuntime 类的成员函数 start()中，接下来就会调用 AndroidRuntime 类的另外一个成员函数 startReg 来注册 Android 核心类的 JNI 方法。

7. AndroidRuntime::startReg

```
int AndroidRuntime::startReg(JNIEnv * env)
{
    /*
     * This hook causes all future threads created in this process to be
     * attached to the JavaVM. (This needs to go away in favor of JNI
     * Attach calls.)
     */
    androidSetCreateThreadFunc((android_create_thread_fn) javaCreateThreadEtc);

    ALOGV("--- registering native functions ---\n");

    /*
     * Every "register" function calls one or more things that return
     * a local reference (e.g. FindClass). Because we haven't really
     * started the VM yet, they're all getting stored in the base frame
     * and never released. Use Push/Pop to manage the storage.
     */
    env->PushLocalFrame(200);

    if (register_jni_procs(gRegJNI, NELEM(gRegJNI), env) < 0) {
        env->PopLocalFrame(NULL);
        return -1;
    }
    env->PopLocalFrame(NULL);

    //createJavaThread("fubar", quickTest, (void*) "hello");

    return 0;
}
```

AndroidRuntime类的成员函数startReg()首先调用函数androidSetCreateThreadFunc()设置线程创建指向javaCreateThreadEtc的函数指针。这个函数指针是用来初始化一个Native线程的JNI环境的，也就是说，当开发者在C++代码中创建一个Native线程的时候，函数javaCreateThreadEtc会被调用来初始化该Native线程的JNI环境。

AndroidRuntime类的成员函数startReg()接着调用函数register_jni_procs()来注册Android核心类的JNI方法。在注册JNI方法时会涉及一些Java对象的引用，但由于此时还没有启动虚拟机，因此将引用保存在Native堆栈中。当前线程的JNI环境是由参数env所指向的一个JNIEnv对象来描述的，通过调用它的成员函数PushLocalFrame和PopLocalFrame就可以手动向当前线程的Native堆栈压入和弹出一个帧。帧保存着Java对象在Native代码中的本地引用。

通过观察全局变量gRegJNI所描述的JNI方法注册函数表，可以了解注册了哪些Android核心类的JNI方法。

```
static const RegJNIRec gRegJNI[] = {
    REG_JNI(register_android_debug_JNITest),
    REG_JNI(register_com_android_internal_os_RuntimeInit),
```

```
    REG_JNI(register_android_os_SystemClock),
    REG_JNI(register_android_util_EventLog),
    REG_JNI(register_android_util_Log),
    REG_JNI(register_android_util_FloatMath),
    REG_JNI(register_android_text_format_Time),
    REG_JNI(register_android_content_AssetManager),
    REG_JNI(register_android_content_StringBlock),
    REG_JNI(register_android_content_XmlBlock),
…
    REG_JNI(register_android_content_res_ObbScanner),
    REG_JNI(register_android_content_res_Configuration),

    REG_JNI(register_android_animation_PropertyValuesHolder),
    REG_JNI(register_com_android_internal_content_NativeLibraryHelper),
    REG_JNI(register_com_android_internal_net_NetworkStatsFactory),
};
```

8. androidSetCreateThreadFunc()

```
void androidSetCreateThreadFunc(android_create_thread_fn func)
{
    gCreateThreadFn = func;
}
```

从上面的代码可以看到，androidSetCreateThreadFunc 将函数指针 gCreateThreadFn 指向了 javaCreateThreadEtc，而 gCreateThreadFn 默认指向 androidCreateRawThreadEtc。

至此，DVM 的启动流程基本分析完毕。整个流程完成了如下工作：

- 创建了一个 Dalvik 虚拟机实例；
- 加载了 Java 核心类及其 JNI 方法；
- 为主线程的设置了一个 JNI 环境；
- 注册了 Android 核心类的 JNI 方法。

Zygote 进程事先创建并初始化了一个 Dalvik 虚拟机实例，当 Zygote 进程创建新的 Android 应用进程时，可以直接将这个事先准备好的 DVM 实例复制到新的 Android 应用进程中，从而避免了每次新建 Android 进程都需要重新创建 DVM 实例，从而提升了 Android 应用的启动速度。另一方面，Java 与 Android 的核心类以及 JNI 方法都是映射到内存中的，因此新建的 Android 应用进程可以与 Zygote 进程共享这些类和 JNI 方法，节约内存空间。

当然，Zygote 进程在启动时必须加载大量的核心类，验证、优化以及注册大量的 Android 核心方法，这极大地降低了 Zygote 进程自身的启动速度。并且 Zygote 进程是在开机时由 init 进程启动的，也就意味着开机速度将变慢。但是日常生活中用户将手机关机或重启的频率极低，因此以牺牲开机时间为代价换取 Android 系统的流畅性与应用的启动速度是十分划算的。

1.1.3 DVM 虚拟机运行过程

Dalvik 虚拟机在 Zygote 进程中启动完成之后，就会获得一个 JavaVM 实例和一个