Dalvik虚拟机

罗升阳

http://weibo.com/shengyangluo

http://blog.csdn.net/luoshengyang

About Me

- 《老罗的Android之旅》博客作者
- 《Android系统源代码情景分析》书籍作者
- 博客：http://blog.csdn.net/Luoshengyang
- 微博：http://weibo.com/shengyangluo

Agenda

- Dalvik虚拟机概述
- Dalvik虚拟机的启动过程
- Dalvik虚拟机的运行过程
- JNI函数的注册过程
- Dalvik虚拟机进程
- Dalvik虚拟机线程

Dalvik虚拟机概述

- Dalvik虚拟机由Dan Bornstein开发，名字来源于他的祖先曾经居住过的位于冰岛的同名小渔村
- Dalvik虚拟机起源于Apache Harmony项目，后者是由Apache软件基金会主导的，目标是实现一个独立的、兼容JDK 5的虚拟机，并根据Apache License v2发布

Dalvik虚拟机概述

- Dalvik虚拟机与Java虚拟机的区别

	Java Virtual Machine	Dalvik Virtual Machine
Instruction Set	Java Bytecode (Stack Based)	Dalvik Bytecode (Register Based)
File Format	.class file (one file, one class)	.dex file (one file, many classes)

Dalvik虚拟机概述

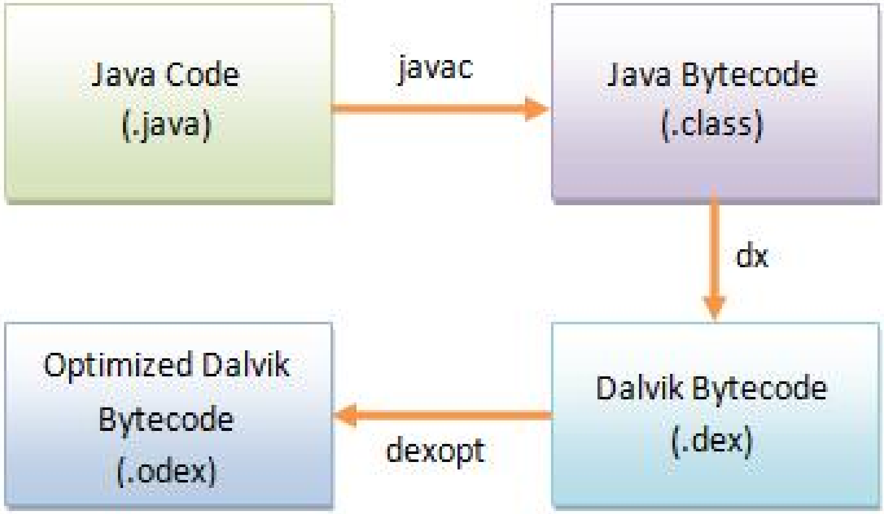
- 基于堆栈的Java指令(1个字节)和基于寄存器的Dalvik指令(2、4或者6个字节)各有优劣
- 一般而言，执行同样的功能， Java虚拟机需要更多的指令（主要是load和store指令），而Dalvik虚拟机需要更多的指令空间
- 需要更多指令意味着要多占用CPU时间，而需要更多指令空间意味着指令缓冲（i-cache）更易失效

Dalvik虚拟机概述

- Dalvik虚拟机使用dex（Dalvik Executable）格式的类文件，而Java虚拟机使用class格式的类文件
- 一个dex文件可以包含若干个类，而一个class文件只包括一个类
- 由于一个dex文件可以包含若干个类，因此它可以将各个类中重复的字符串只保存一次，从而节省了空间，适合在内存有限的移动设备使用
- 一般来说，包含有相同类的未压缩dex文件稍小于一个已经压缩的jar文件

Dalvik虚拟机概述

- Dex文件的生成



Dalvik虚拟机概述

- Dex文件的优化
  - 将invoke-virtual指令中的method index转换为vtable index – *加快虚函数调用速度*
  - 将get/put指令中的field index转换为byte offset – *加快实例成员变量访问速度*
  - 将boolean/byte/char/short变种的get/put指令统一转换为32位的get/put指令 – *减小VM解释器的大小，从而更有效地利用CPU的i-cache*
  - 将高频调用的函数，例如String.length，转换为inline函数 – *消除函数调用开销*
  - 移除空函数，例如Object.<init> -- *消除空函数调用*
  - 将可以预先计算的数据进行预处理，例如预先生成VM根据class name查询class的hash table – *节省Dex文件加载时间以及内存占用空间*

Dalvik虚拟机概述

- 将invoke-virtual指令中的method index转换为vtable index

```
invoke-virtual {v1, v2}, method@BBBB
➔
invoke-virtual-quick {v1,v2},vtable #0xhh
```

- 将get/put指令中的field index转换为byte offset

```
iget-object v0, v2, field@BBBB
➔
iget-object-quick v0,v2,[obj+0x100]
```

Dalvik虚拟机概述

- Dex文件的优化时机
  - VM在运行时即时优化，例如使用DexClassLoader动态加载dex文件时。这时候需要指定一个当前进程有写权限的用来保存odex的目录。
  - APP安装时由具有root权限的installd优化。这时候优化产生的odex文件保存在特权目录/data/dalvik-cache中。
  - 编译时优化。这时候编译出来的jar/apk里面的classes.dex被提取并且优化为classes.odex保存在原jar/apk所在目录，打包在system image中。

Dalvik虚拟机概述

- 内存管理
  - Java Object Heap
    - 大小受限，16M/24M/32M/48M/…
  - Bitmap Memory(External Memroy)：
    - 大小计入Java Object Heap
  - Native Heap
    - 大小不受限

Dalvik虚拟机概述

- Java Object Heap
  - 用来分配Java对象。Dalvik虚拟机在启动的时候，可以通过-Xms和-Xmx选项来指定Java Object Heap的最小值和最大值。
  - Java Object Heap的最小和最大默认值为2M和16M。但是厂商会根据手机的配置情况进行调整，例如，G1、Droid、Nexus One和Xoom的Java Object Heap的最大值分别为16M、24M、32M 和48M。
  - 通过ActivityManager.getMemoryClass可以获得Dalvik虚拟机的Java Object Heap的最大值。

Dalvik虚拟机概述

- Bitmap 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。这就是为什么我们在调用BitmapFactory相关的接口来处理大图像时，会抛出一个OutOfMemoryError异常的原因：***java.lang.OutOfMemoryError: bitmap size exceeds VM budget***
  - 在HoneyComb以及更高的版本中，Bitmap Memory就直接是在Java Object Heap中分配了，这样就可以直接接受GC的管理。

Dalvik虚拟机概述

- Native Heap
  - 在Native Code中使用malloc等分配出来的内存，这部分内存不受Java Object Heap的大小限制。
  - 注意，不要因为Native Heap可以自由使用就滥用，因为滥用Native Heap会导致系统可用内存急剧减少，从而引发系统采取激进的措施来Kill掉某些进程，用来补充可用内存，这样会影响系统体验。

Dalvik虚拟机概述

- 垃圾收集(GC)
  - Step 1: Mark，使用RootSet标记对象引用
  - Step 2: Sweep，回收没有被引用的对象
- GingerBread之前
  - Stop-the-word，也就是垃圾收集线程在执行的时候，其它的线程都停止
  - Full heap collection，也就是一次收集完全部的垃圾
  - 一次垃圾收集造成的程序中止时间通常都大于100ms
- GingerBread之后
  - Cocurrent，也就是大多数情况下，垃圾收集线程与其它线程是并发执行的
  - Partial collection，也就是一次可能只收集一部分垃圾
  - 一次垃圾收集造成的程序中止时间通常都小于5ms

Dalvik虚拟机概述

- Dalvik虚拟机执行完成一次垃圾收集之后，我们通常可以看到类似以下的日志输出：

```
D/dalvikvm(9050): GC_CONCURRENT freed 2049K, 65% free 3571
K/9991K, external 4703K/5261K, paused 2ms+2ms
```

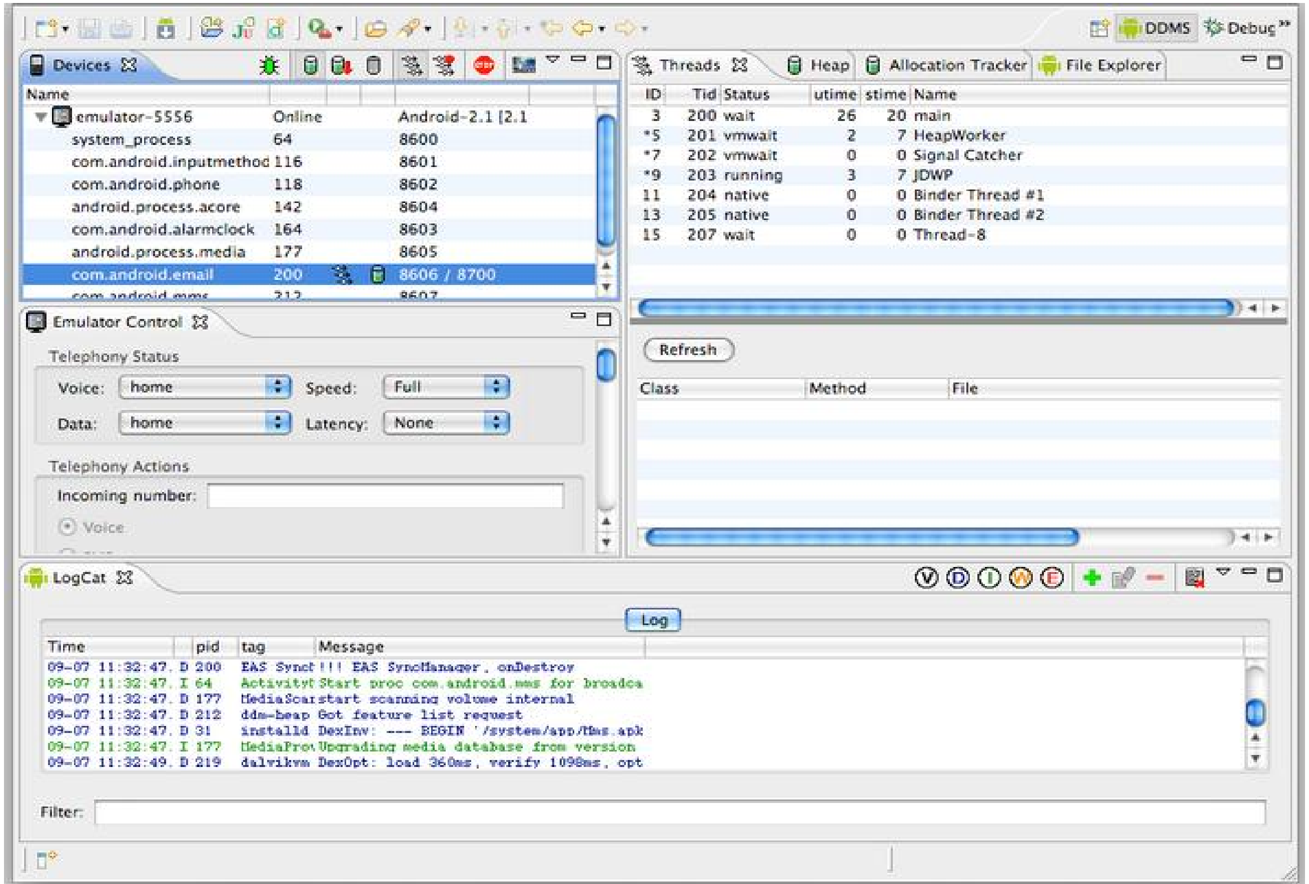
- GC_CONCURRENT表示并行GC，2049K表示总共回收的内存，3571K/9991K表示Java Object Heap统计，即在9991K的Java Object Heap中，有3571K是正在使用的，4703K/5261K表示External Memory统计，即在5261K的External Memory中，有4703K是正在使用的，2ms+2ms表示垃圾收集造成的程序中止时间。

Dalvik虚拟机概述

- 即时编译(JIT)
  - 从2.2开始支持JIT，并且是可选的，编译时通过WITH_JIT宏进行控制
  - 基于执行路径(Executing Path)对热门的代码片断进行优化(Trace JIT)，传统的Java虚拟机以Method为单位进行优化(Method JIT)
  - 可以利用运行时信息进行激进优化，获得比静态编译语言更高的性能，如Lazy Unlocking机制，可以参考《Oracle JRockit: The Definitive Guide》一书
  - 实现原理：http://blog.reverberate.org/2012/12/hello-jit-world-joy-of-simple-jits.html

Dalvik虚拟机概述

- 支持JDWP（Java Debug Wire Protocol）协议
  - 每一个Dalvik虚拟机进程都都提供有一个端口来供调试器连接
  - DDMS提供有一个转发端口8870，通过它可以同时调试多个Dalvik虚拟机进程



Name				
emulator-5556	Online			Android-2.1 [2.1
system_process	64			8600
com.android.inputmethod	116			8601
com.android.phone	118			8602
android.process.acore	142			8604
com.android.alarmclock	164			8603
android.process.media	177			8605
com.android.email	200			8606 / 8700
com.android.mms	212			8607

Telephony Status

Voice: home　Speed: Full

Data: home　Latency: None

Telephony Actions

Incoming number:

○ Voice

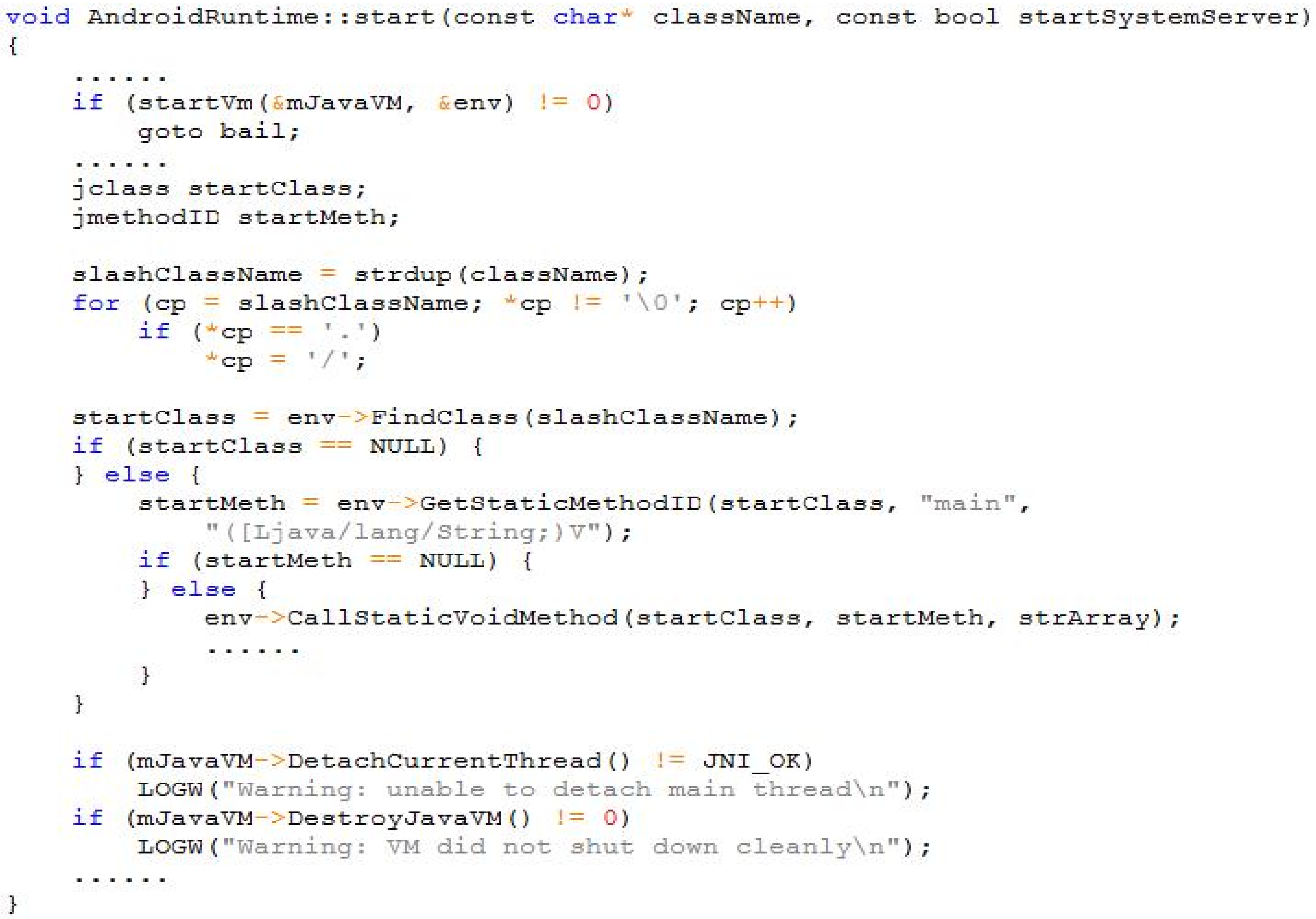
ID	Tid	Status	utime	stime	Name
3	200	wait	26	20	main
*5	201	vmwait	2	7	HeapWorker
*7	202	vmwait	0	0	Signal Catcher
*9	203	running	3	7	JDWP
11	204	native	0	0	Binder Thread #1
13	205	native	0	0	Binder Thread #2
15	207	wait	0	0	Thread-8

```
Time               pid   tag        Message
09-07 11:32:47. D 200    EAS Synct !!! EAS SyncManager, onDestroy
09-07 11:32:47. I 64     Activityt Start proc com.android.mms for broadca
09-07 11:32:47. D 177    MediaScar start scanning volume internal
09-07 11:32:47. D 212    ddm-heap Got feature list request
09-07 11:32:47. D 31     installd DexInv: --- BEGIN '/system/app/Mms.apk
09-07 11:32:47. I 177    MediaPro\ Upgrading media database from version
09-07 11:32:49. D 219    dalvikvm DexOpt: load 360ms, verify 1098ms, opt
```

Filter:

Dalvik虚拟机的启动过程

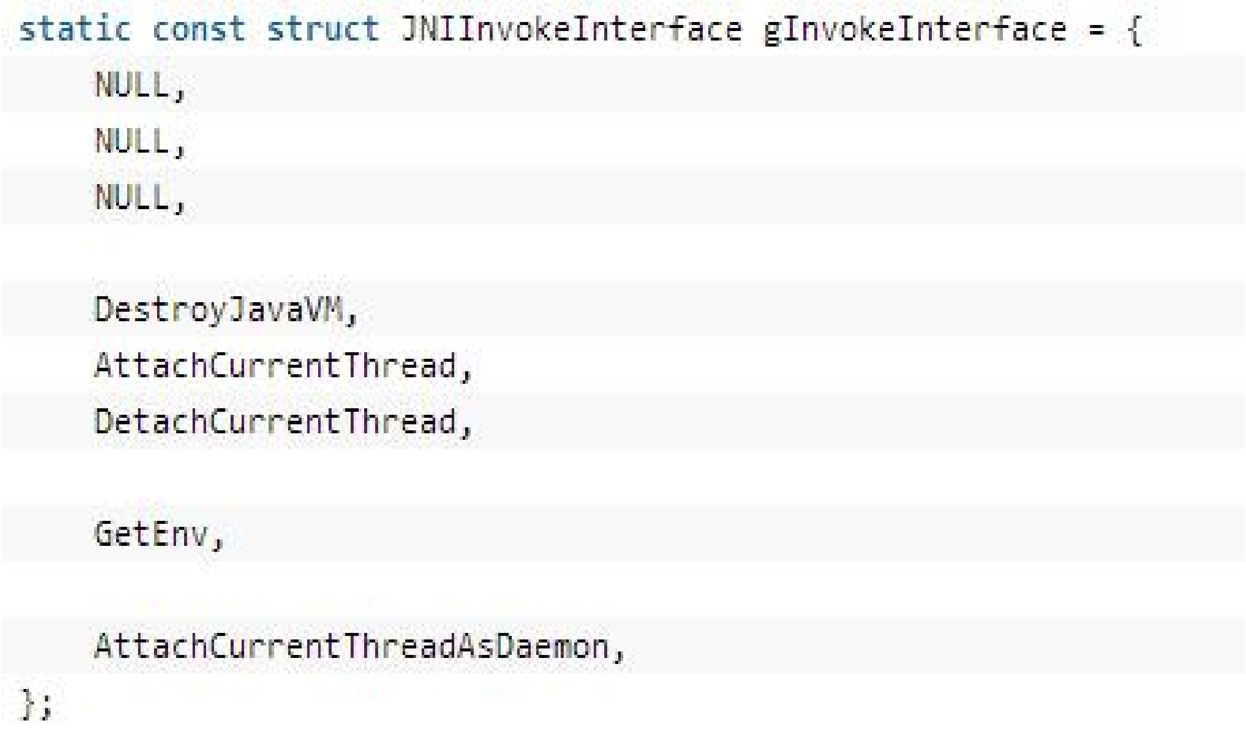
- Dalvik虚拟机由Zygote进程启动，然后再复制到System Server进程和应用程序进程



```
void AndroidRuntime::start(const char* className, const bool startSystemServer)
{
    ......
    if (startVm(&mJavaVM, &env) != 0)
        goto bail;
    ......
    jclass startClass;
    jmethodID startMeth;

    slashClassName = strdup(className);
    for (cp = slashClassName; *cp != '\0'; cp++)
        if (*cp == '.')
            *cp = '/';

    startClass = env->FindClass(slashClassName);
    if (startClass == NULL) {
    } else {
        startMeth = env->GetStaticMethodID(startClass, "main",
            "([Ljava/lang/String;)V");
        if (startMeth == NULL) {
        } else {
            env->CallStaticVoidMethod(startClass, startMeth, strArray);
            ......
        }
    }

    if (mJavaVM->DetachCurrentThread() != JNI_OK)
        LOGW("Warning: unable to detach main thread\n");
    if (mJavaVM->DestroyJavaVM() != 0)
        LOGW("Warning: VM did not shut down cleanly\n");
    ......
}
```

Dalvik虚拟机的启动过程

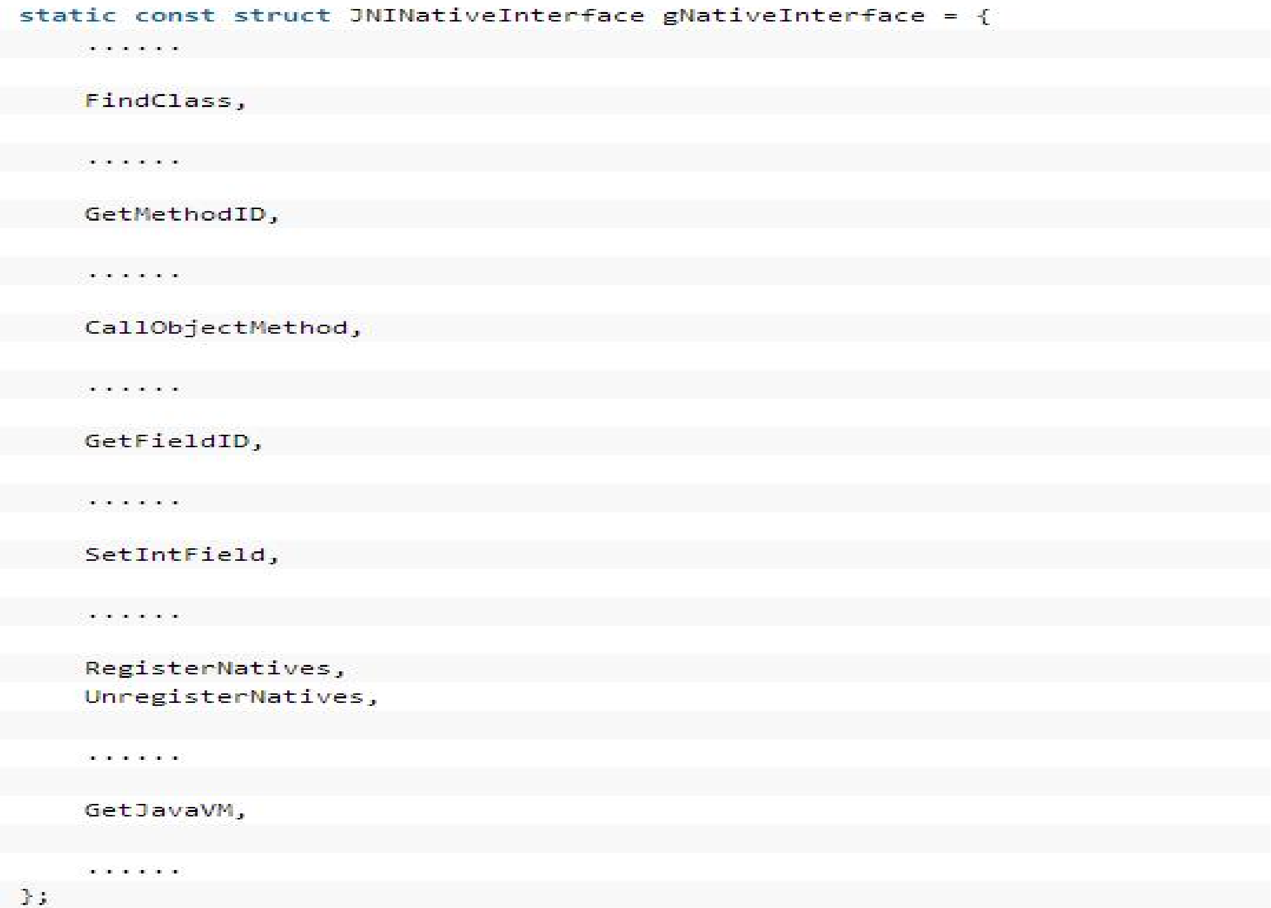
- startVM的过程中会创建一个JavaVMExt，并且该JavaVMExt关联有一个JNIInvokeInterface，Native Code通过它来访问Dalvik虚拟机



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

    DestroyJavaVM,
    AttachCurrentThread,
    DetachCurrentThread,

    GetEnv,

    AttachCurrentThreadAsDaemon,
};
```

Dalvik虚拟机的启动过程

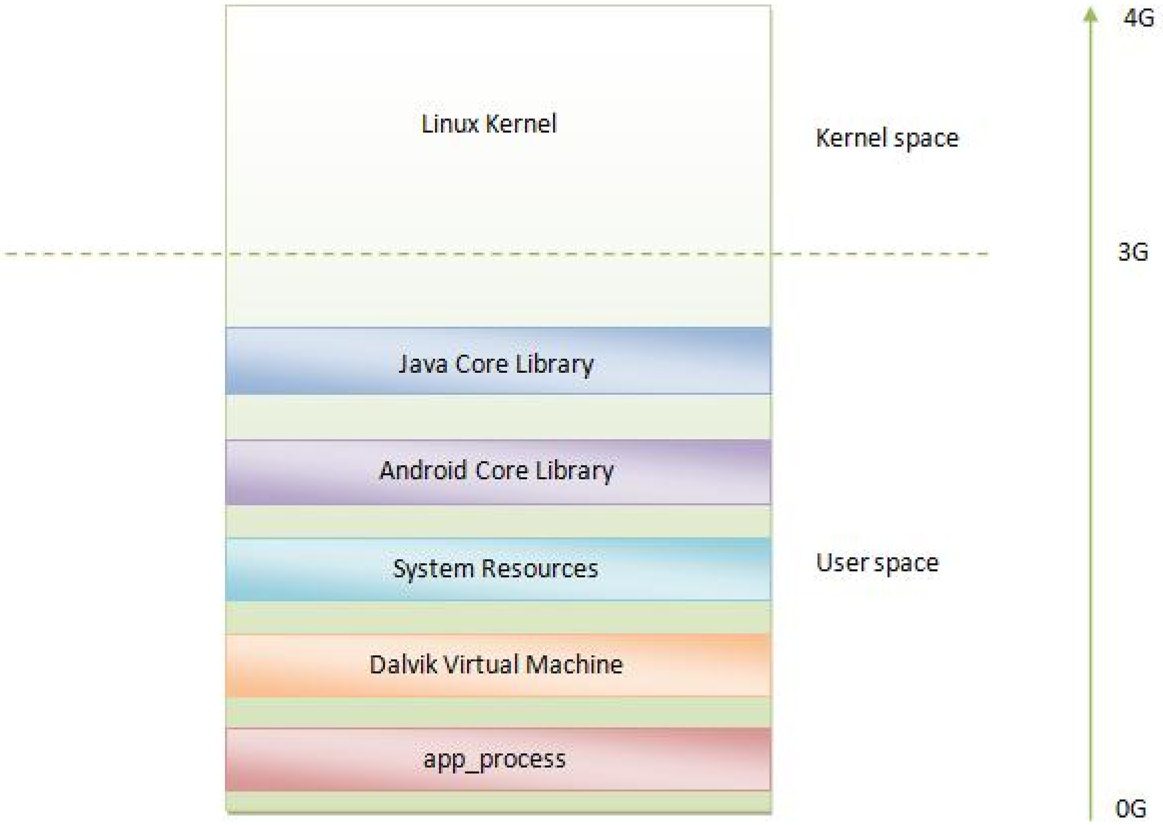
- startVM的过程中还会为当前线程关联有一个JNIEnvExt，并且该JNIEnvExt 关联有一个JNINativeInterface， Native Code通过它来调用Java函数或者访问Java对象



```
static const struct JNINativeInterface gNativeInterface = {
    ......

    FindClass,

    ......

    GetMethodID,

    ......

    CallObjectMethod,

    ......

    GetFieldID,

    ......

    SetIntField,

    ......

    RegisterNatives,
    UnregisterNatives,

    ......

    GetJavaVM,

    ......
};
```

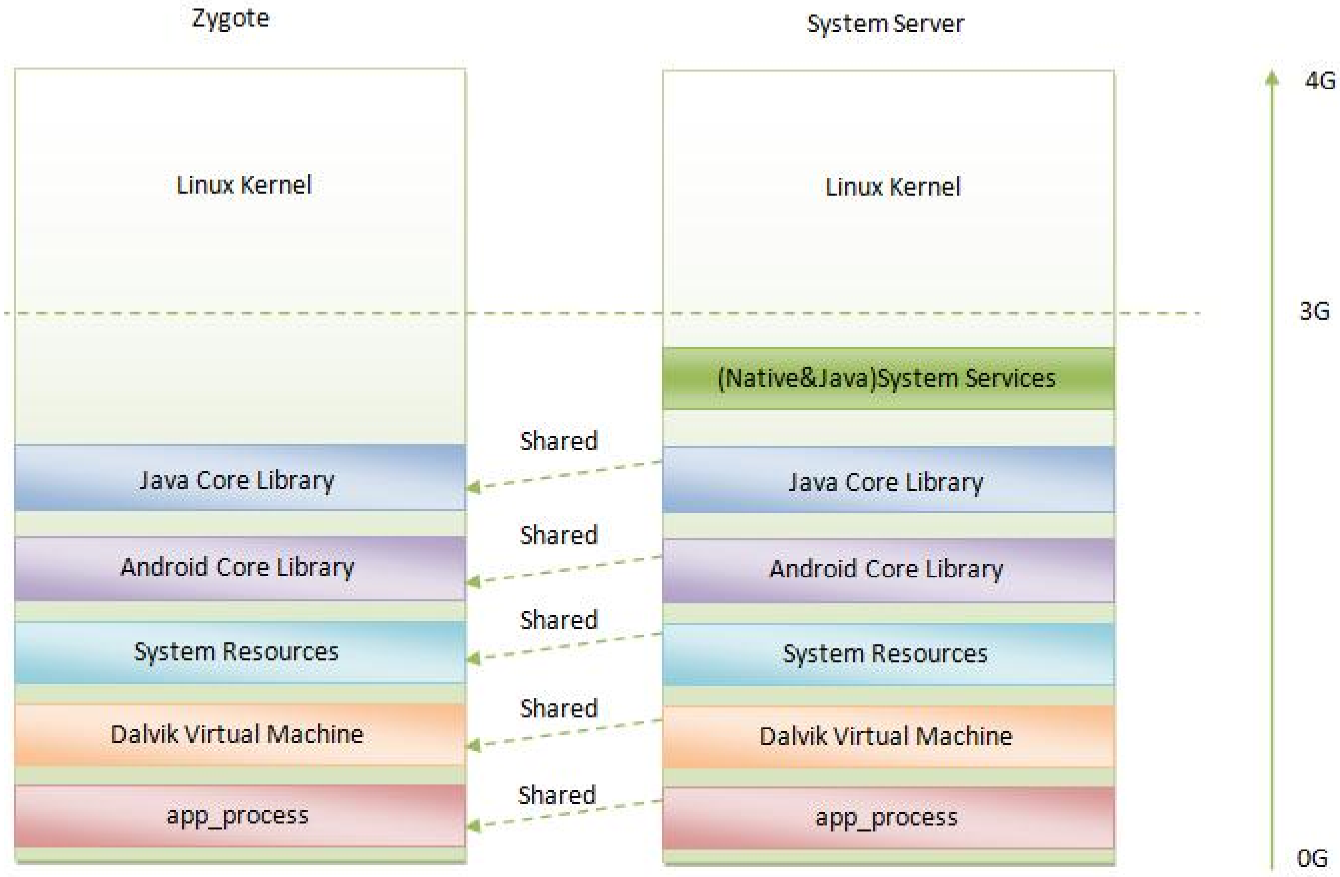
Dalvik虚拟机的启动过程

- Dalvik虚拟机在Zygote进程启动的过程中，还会进一步预加载Java和Android核心类库以及系统资源



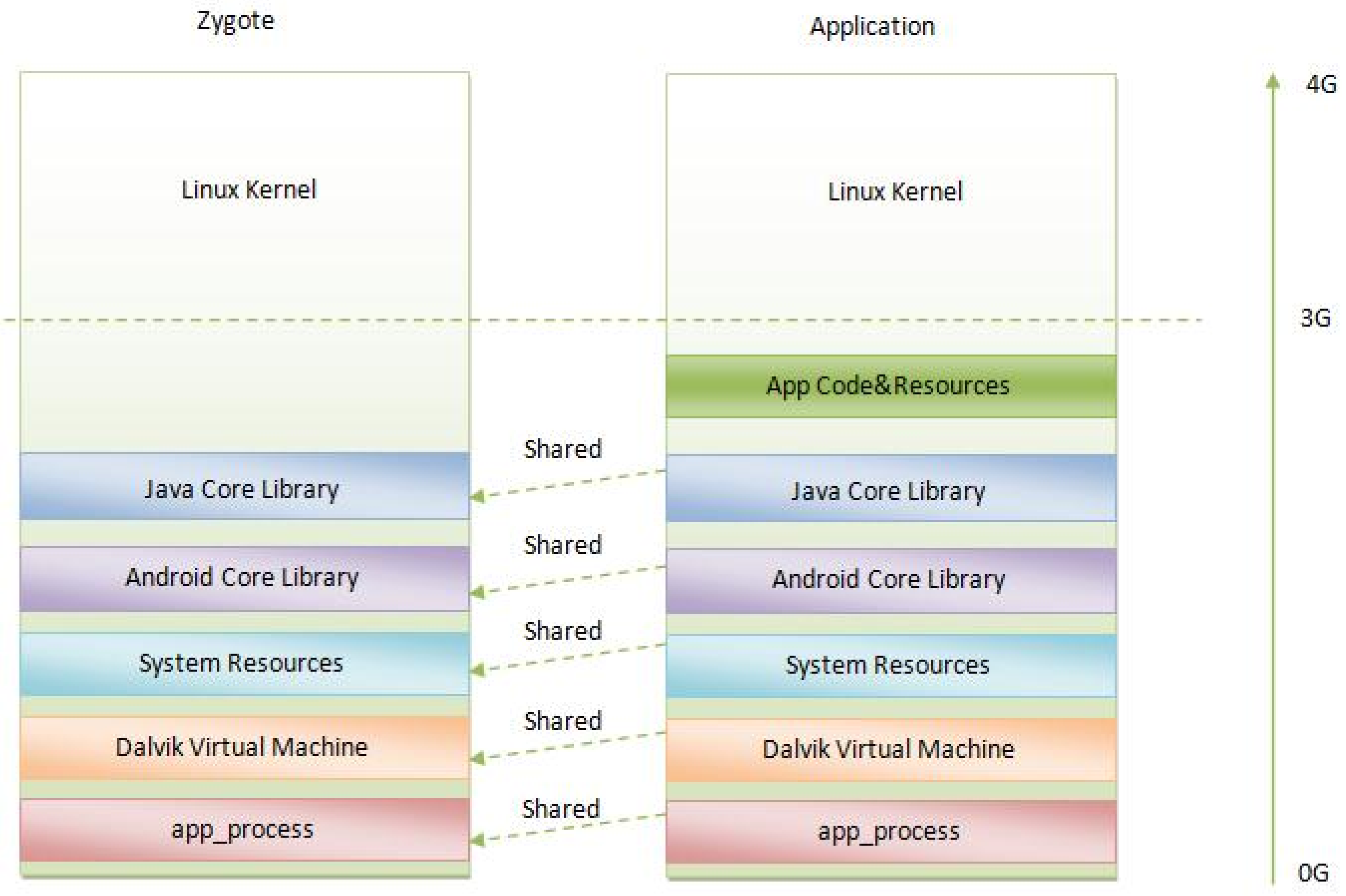
Dalvik虚拟机的启动过程

- Dalvik虚拟机从Zygote进程复制到System Server进程之后，它们就通过COW(Copy On Write)机制共享同一个Dalvik虚拟机实例以及预加载类库和资源



Dalvik虚拟机的启动过程

- Dalvik虚拟机从Zygote进程复制到应用程序进程之后，它们同样会通过COW(Copy On Write)机制共享同一个Dalvik虚拟机实例以及预加载类库和资源

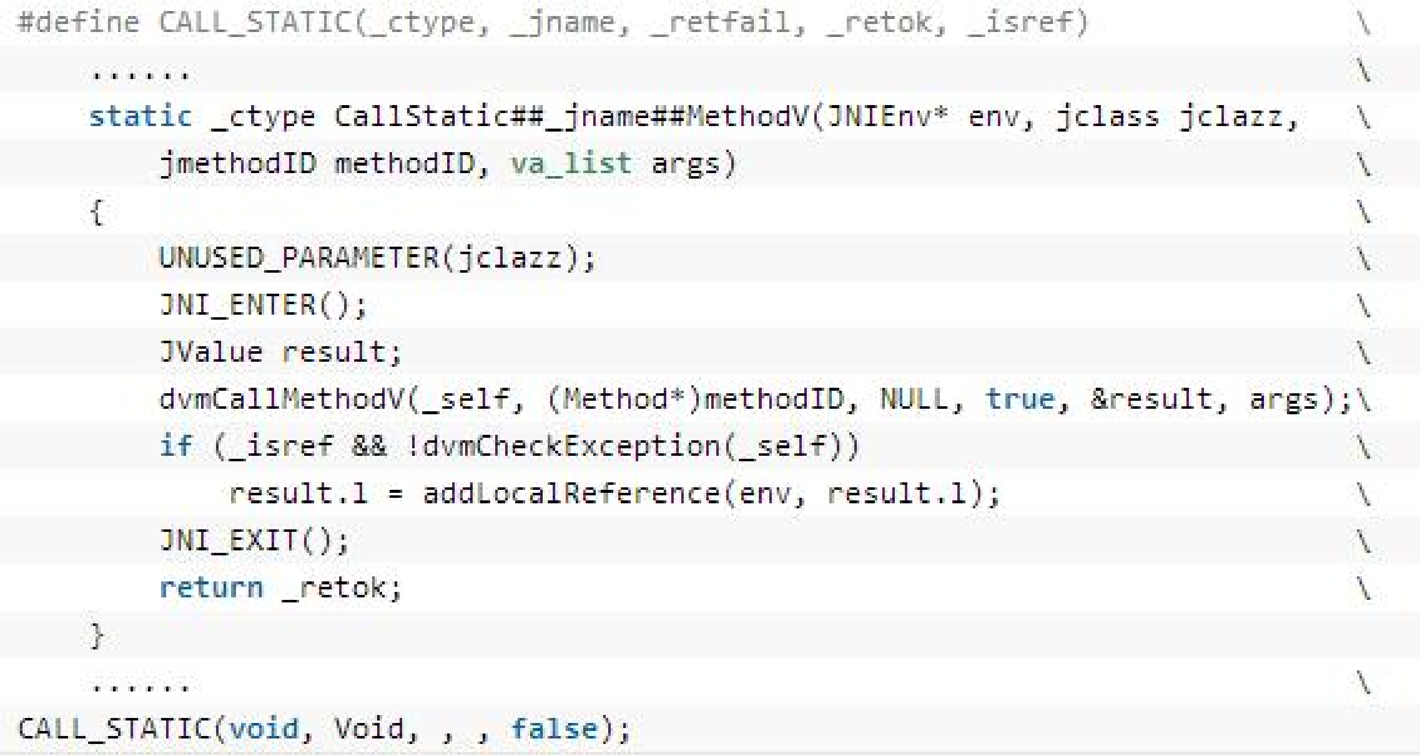


Dalvik虚拟机的运行过程

- Dalvik虚拟机在Zygote进程中启动之后，就会以ZygoteInit.main为入口点开始运行
- Dalvik虚拟机从Zygote进程复制到System Server进程之后，就会以SystemServer.main为入口点开始运行
- Dalvik虚拟机Zygote进程复制到应用程序进程之后，就会以ActivityThread.main为入口点开始运行
- 上述入口点都是通过调用JNINativeInterface接口的成员函数CallStaticVoidMethod来进入的

Dalvik虚拟机的运行过程

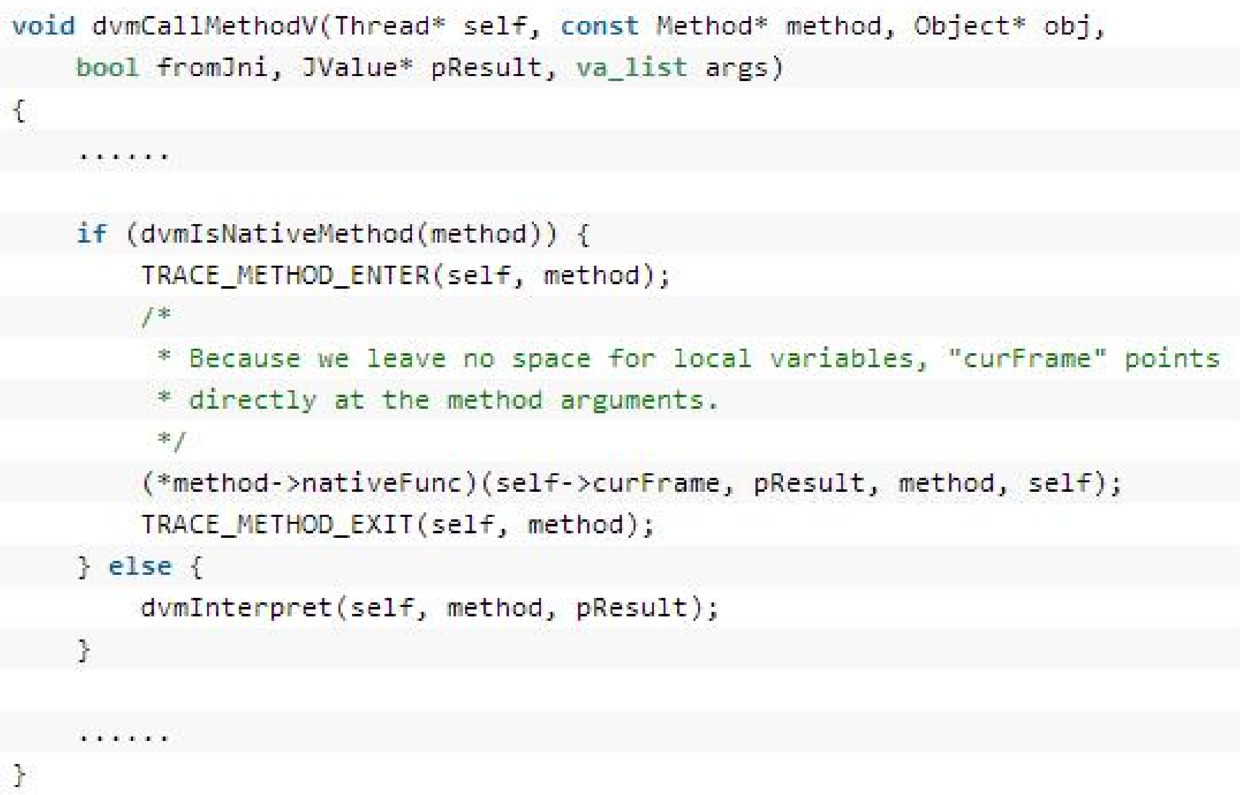
- JNINativeInterface->CallStaticVoidMethod对应的实现为CallStaticVoidMethodV



```
#define CALL_STATIC(_ctype, _jname, _retfail, _retok, _isref)               \
    ......                                                                  \
    static _ctype CallStatic##_jname##MethodV(JNIEnv* env, jclass jclazz,   \
        jmethodID methodID, va_list args)                                   \
    {                                                                       \
        UNUSED_PARAMETER(jclazz);                                           \
        JNI_ENTER();                                                        \
        JValue result;                                                      \
        dvmCallMethodV(_self, (Method*)methodID, NULL, true, &result, args);\
        if (_isref && !dvmCheckException(_self))                            \
            result.l = addLocalReference(env, result.l);                    \
        JNI_EXIT();                                                         \
        return _retok;                                                      \
    }
    ......                                                                  \
CALL_STATIC(void, Void, , , false);
```

Dalvik虚拟机的运行过程

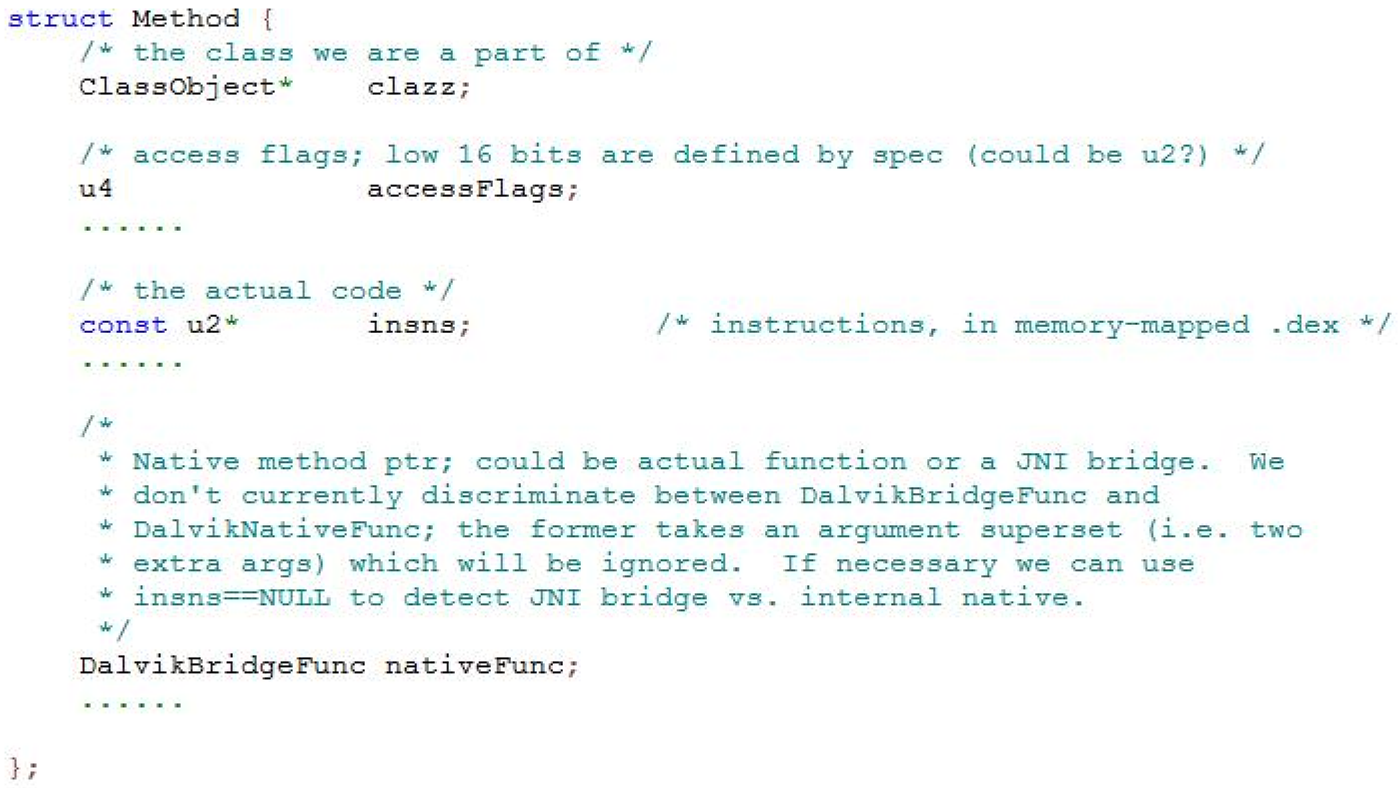
- CallStaticVoidMethodV调用dvmCallMethodV



```
void dvmCallMethodV(Thread* self, const Method* method, Object* obj,
    bool fromJni, JValue* pResult, va_list args)
{
    ......

    if (dvmIsNativeMethod(method)) {
        TRACE_METHOD_ENTER(self, method);
        /*
         * Because we leave no space for local variables, "curFrame" points
         * directly at the method arguments.
         */
        (*method->nativeFunc)(self->curFrame, pResult, method, self);
        TRACE_METHOD_EXIT(self, method);
    } else {
        dvmInterpret(self, method, pResult);
    }

    ......
}
```

Dalvik虚拟机的运行过程

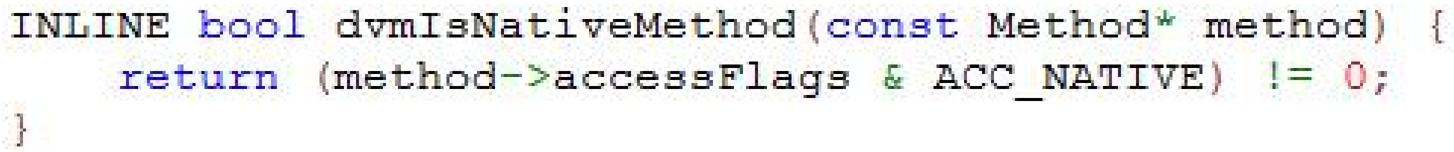
- 在Dalvik虚拟机中，无论是Java函数，还是Native函数，都是通过Method结构体来描述的



```
struct Method {
    /* the class we are a part of */
    ClassObject*    clazz;

    /* access flags; low 16 bits are defined by spec (could be u2?) */
    u4              accessFlags;
    ......

    /* the actual code */
    const u2*       insns;          /* instructions, in memory-mapped .dex */
    ......

    /*
     * Native method ptr; could be actual function or a JNI bridge.  We
     * don't currently discriminate between DalvikBridgeFunc and
     * DalvikNativeFunc; the former takes an argument superset (i.e. two
     * extra args) which will be ignored.  If necessary we can use
     * insns==NULL to detect JNI bridge vs. internal native.
     */
    DalvikBridgeFunc nativeFunc;
    ......

};
```

Dalvik虚拟机的运行过程

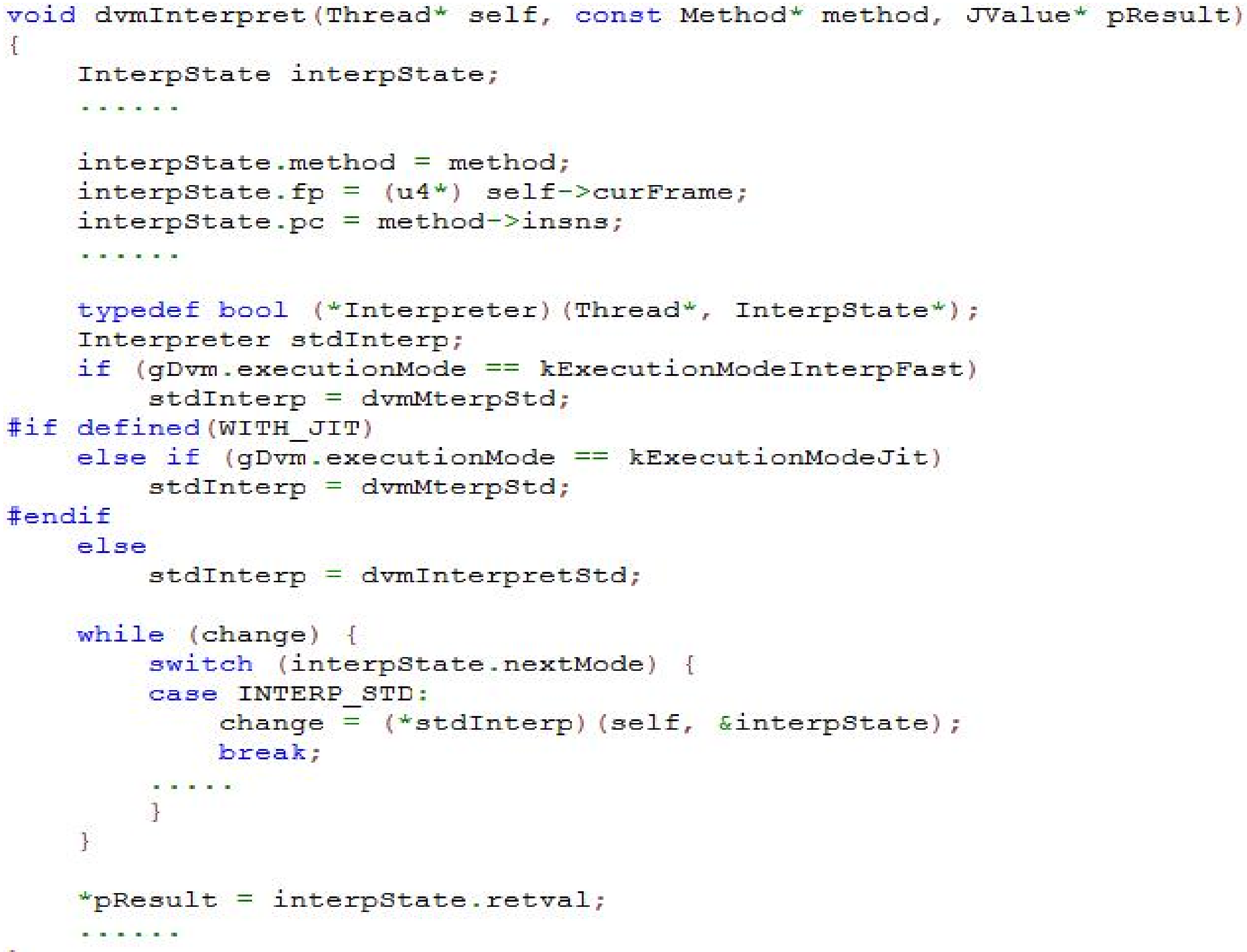
- 在Dalivk虚拟机中，通过dvmIsNativeMethod判断一个函数是Java函数还是Native函数



```
INLINE bool dvmIsNativeMethod(const Method* method) {
    return (method->accessFlags & ACC_NATIVE) != 0;
}
```

Dalvik虚拟机的运行过程

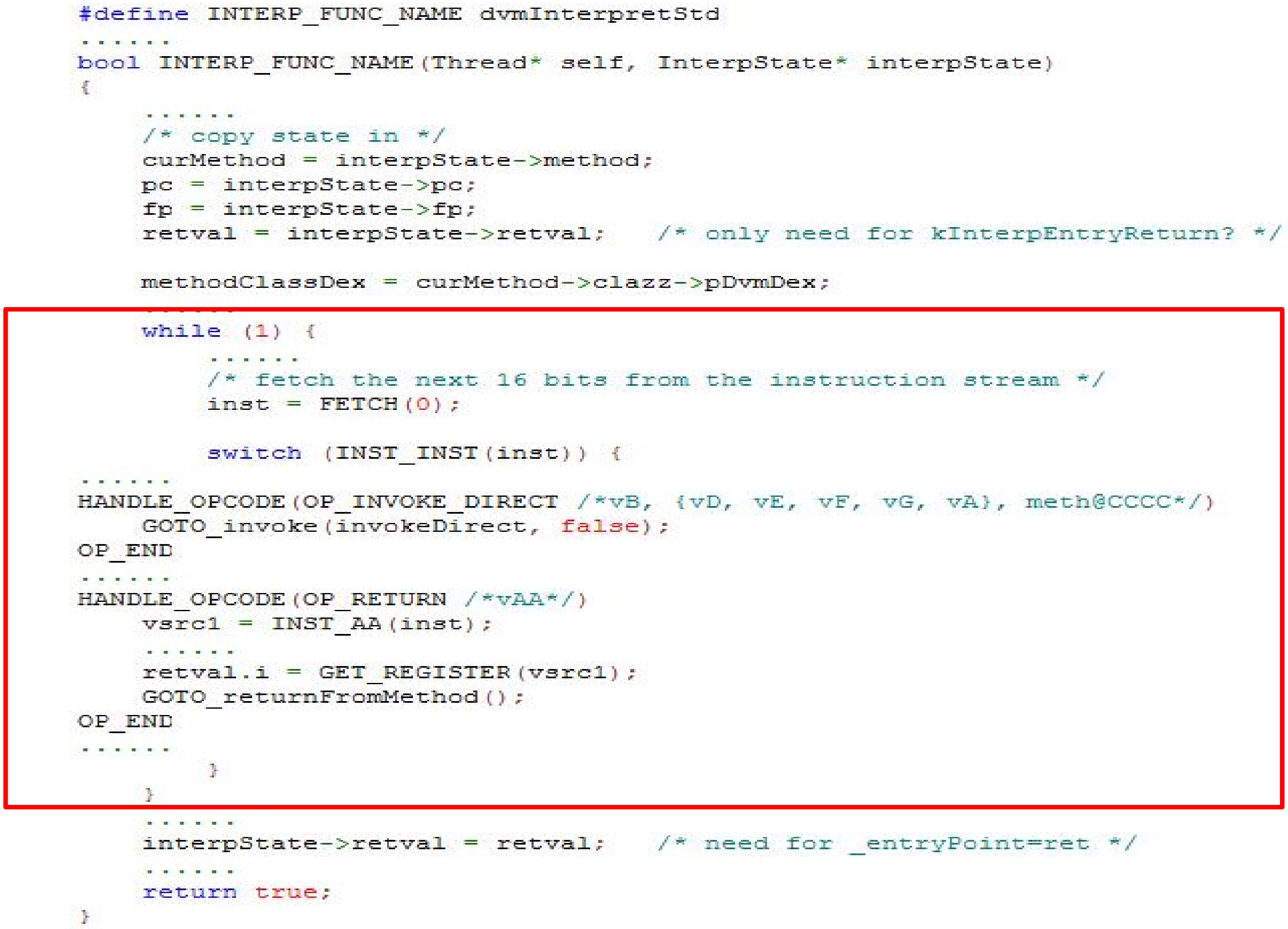
- Native函数直接由CPU执行，Java函数由Dalvik虚拟机解释执行，即通过dvmInterpret函数执行



```
void dvmInterpret(Thread* self, const Method* method, JValue* pResult)
{
    InterpState interpState;
    ......

    interpState.method = method;
    interpState.fp = (u4*) self->curFrame;
    interpState.pc = method->insns;
    ......

    typedef bool (*Interpreter)(Thread*, InterpState*);
    Interpreter stdInterp;
    if (gDvm.executionMode == kExecutionModeInterpFast)
        stdInterp = dvmMterpStd;
#if defined(WITH_JIT)
    else if (gDvm.executionMode == kExecutionModeJit)
        stdInterp = dvmMterpStd;
#endif
    else
        stdInterp = dvmInterpretStd;

    while (change) {
        switch (interpState.nextMode) {
        case INTERP_STD:
            change = (*stdInterp)(self, &interpState);
            break;
        .....
        }
    }

    *pResult = interpState.retval;
    ......
```

Dalvik虚拟机的运行过程

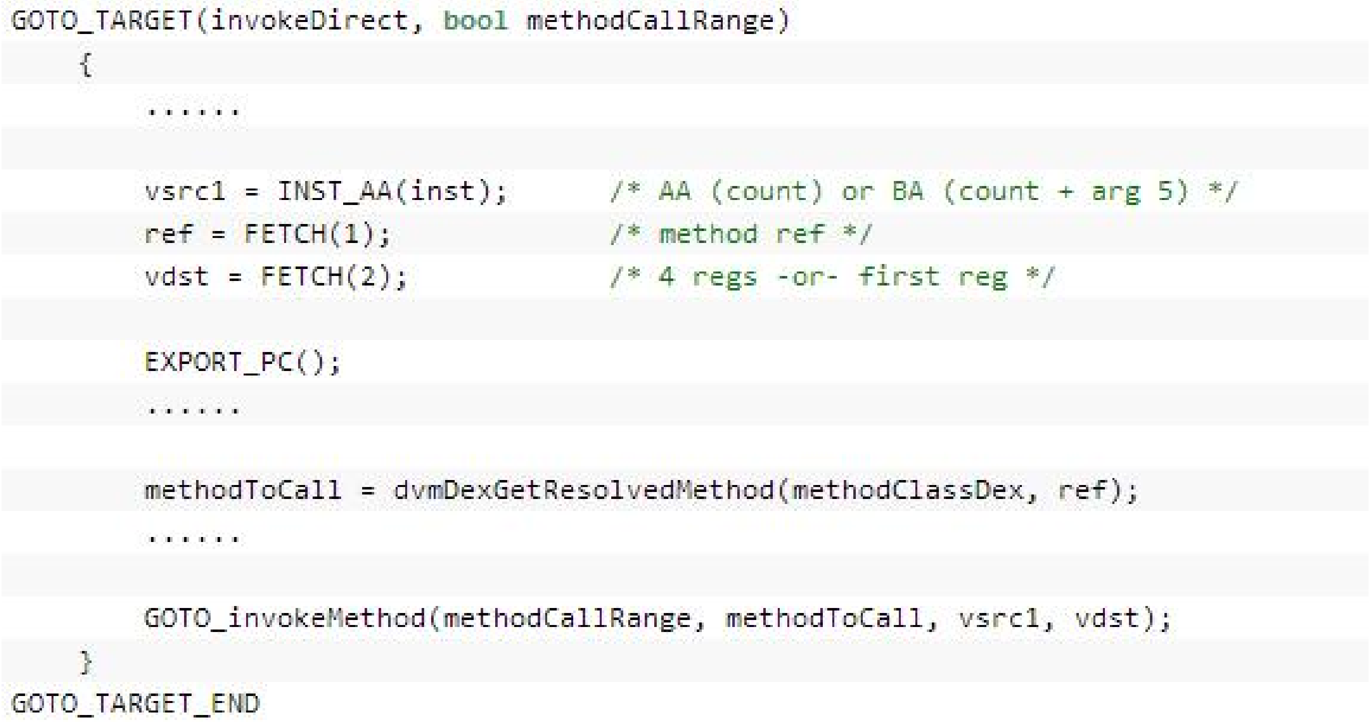
- Dalvik虚拟机标准解释器：dvmInterpretStd



```
#define INTERP_FUNC_NAME dvmInterpretStd
......
bool INTERP_FUNC_NAME(Thread* self, InterpState* interpState)
{
    ......
    /* copy state in */
    curMethod = interpState->method;
    pc = interpState->pc;
    fp = interpState->fp;
    retval = interpState->retval;   /* only need for kInterpEntryReturn? */

    methodClassDex = curMethod->clazz->pDvmDex;
    ......
    while (1) {
        ......
        /* fetch the next 16 bits from the instruction stream */
        inst = FETCH(0);

        switch (INST_INST(inst)) {
......
HANDLE_OPCODE(OP_INVOKE_DIRECT /*vB, {vD, vE, vF, vG, vA}, meth@CCCC*/)
    GOTO_invoke(invokeDirect, false);
OP_END
......
HANDLE_OPCODE(OP_RETURN /*vAA*/)
    vsrc1 = INST_AA(inst);
    ......
    retval.i = GET_REGISTER(vsrc1);
    GOTO_returnFromMethod();
OP_END
......
        }
    }
    ......
    interpState->retval = retval;   /* need for _entryPoint=ret */
    ......
    return true;
}
```

Dalvik虚拟机的运行过程

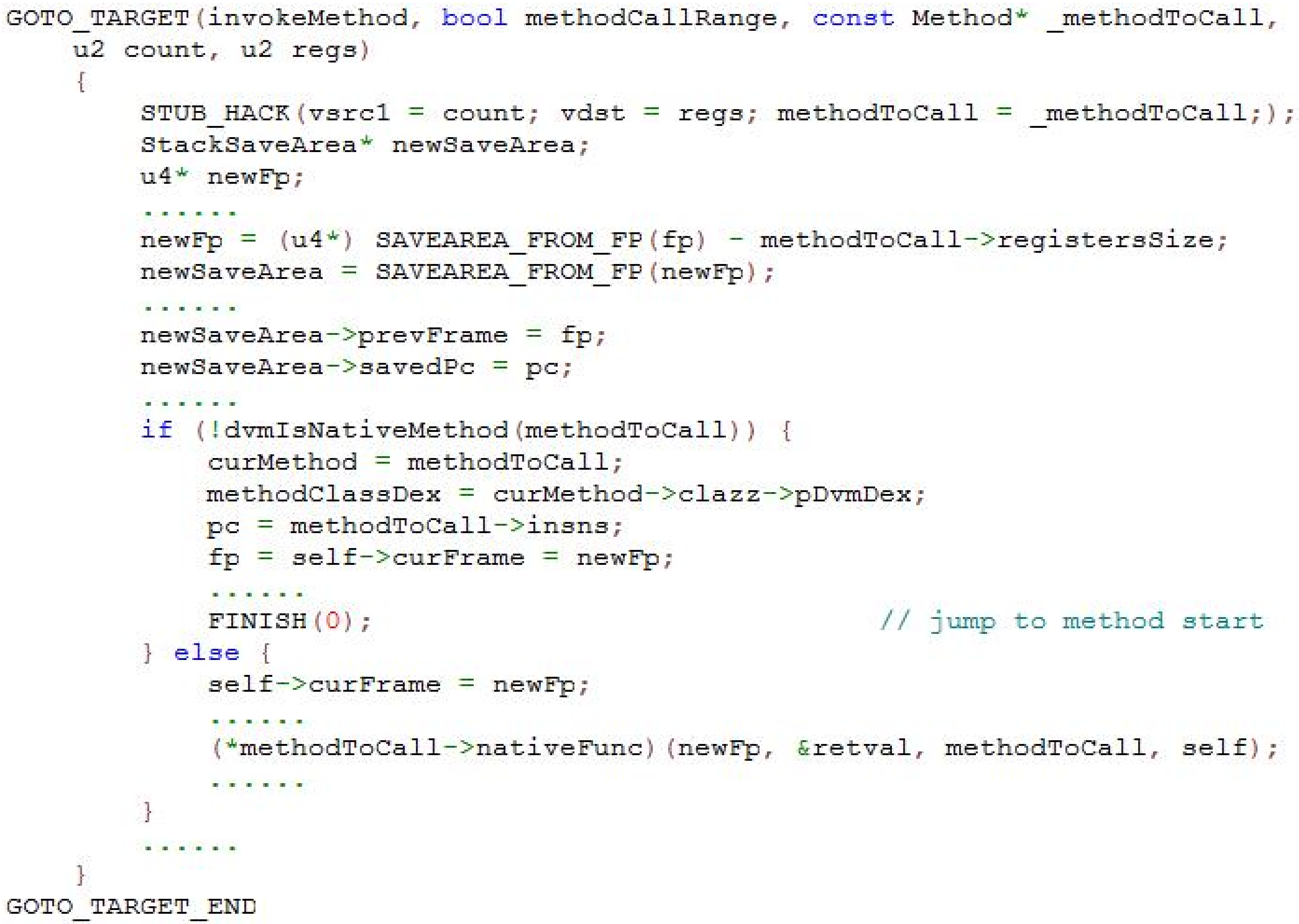
- Invoke-direct指令由函数invokeDirect执行



```
GOTO_TARGET(invokeDirect, bool methodCallRange)
    {
        ......

        vsrc1 = INST_AA(inst);      /* AA (count) or BA (count + arg 5) */
        ref = FETCH(1);             /* method ref */
        vdst = FETCH(2);            /* 4 regs -or- first reg */

        EXPORT_PC();
        ......

        methodToCall = dvmDexGetResolvedMethod(methodClassDex, ref);
        ......

        GOTO_invokeMethod(methodCallRange, methodToCall, vsrc1, vdst);
    }
GOTO_TARGET_END
```

Dalvik虚拟机的运行过程

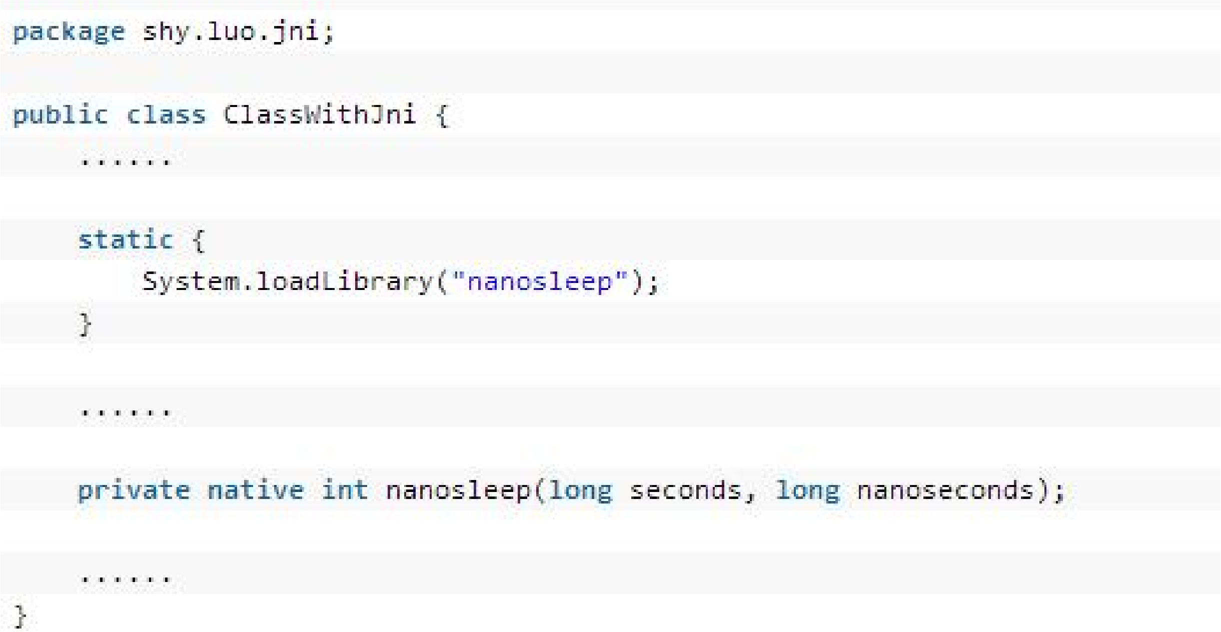
- 函数invokeDirect调用 invokeMethod执行



```
GOTO_TARGET(invokeMethod, bool methodCallRange, const Method* _methodToCall,
    u2 count, u2 regs)
    {
        STUB_HACK(vsrc1 = count; vdst = regs; methodToCall = _methodToCall;);
        StackSaveArea* newSaveArea;
        u4* newFp;
        ......
        newFp = (u4*) SAVEAREA_FROM_FP(fp) - methodToCall->registersSize;
        newSaveArea = SAVEAREA_FROM_FP(newFp);
        ......
        newSaveArea->prevFrame = fp;
        newSaveArea->savedPc = pc;
        ......
        if (!dvmIsNativeMethod(methodToCall)) {
            curMethod = methodToCall;
            methodClassDex = curMethod->clazz->pDvmDex;
            pc = methodToCall->insns;
            fp = self->curFrame = newFp;
            ......
            FINISH(0);                              // jump to method start
        } else {
            self->curFrame = newFp;
            ......
            (*methodToCall->nativeFunc)(newFp, &retval, methodToCall, self);
            ......
        }
        ......
    }
GOTO_TARGET_END
```

JNI函数的注册过程

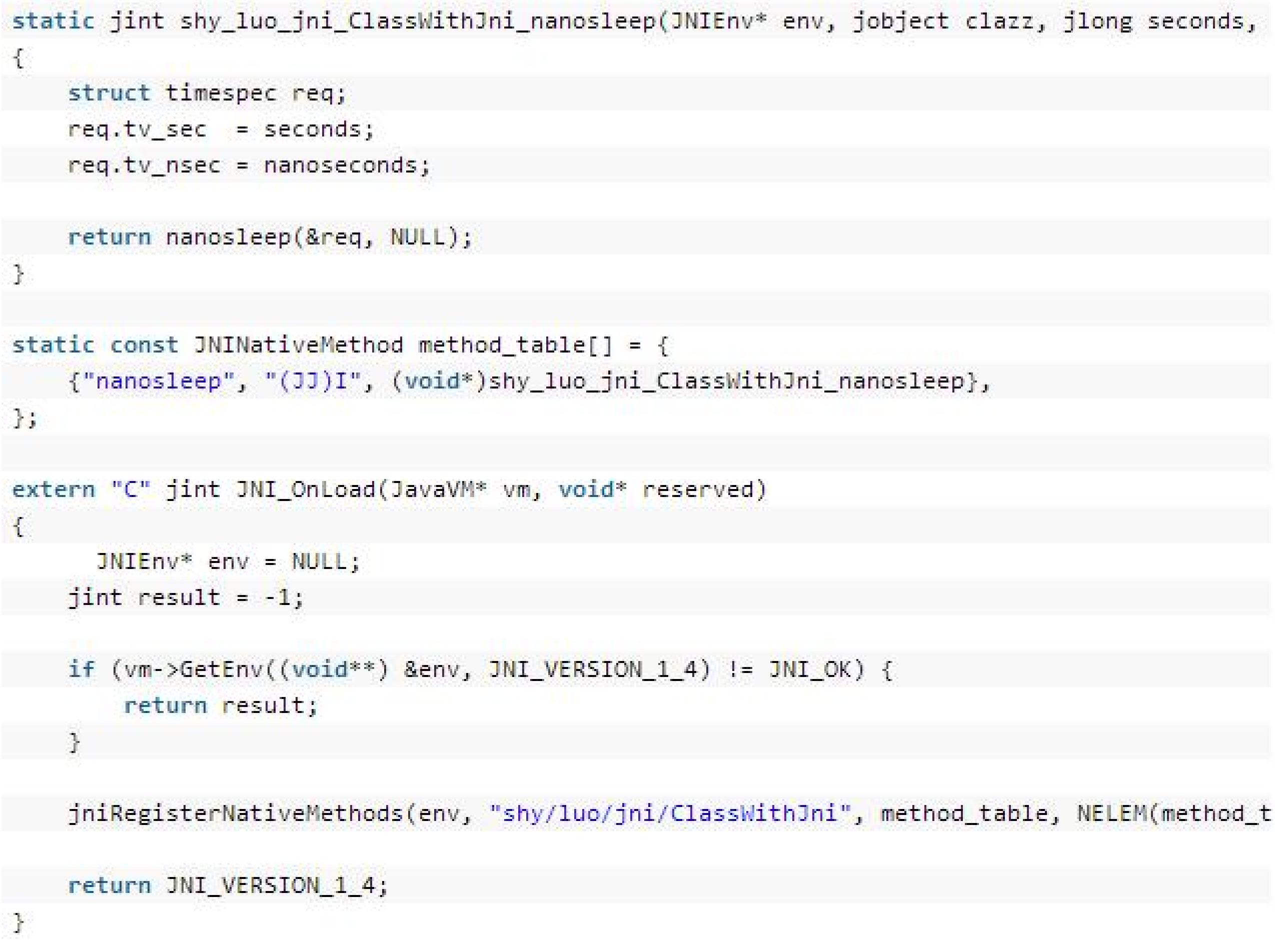
- JNI函数注册示例 -- ClassWithJni



```
package shy.luo.jni;

public class ClassWithJni {
    ......

    static {
        System.loadLibrary("nanosleep");
    }

    ......

    private native int nanosleep(long seconds, long nanoseconds);

    ......
}
```

JNI函数的注册过程

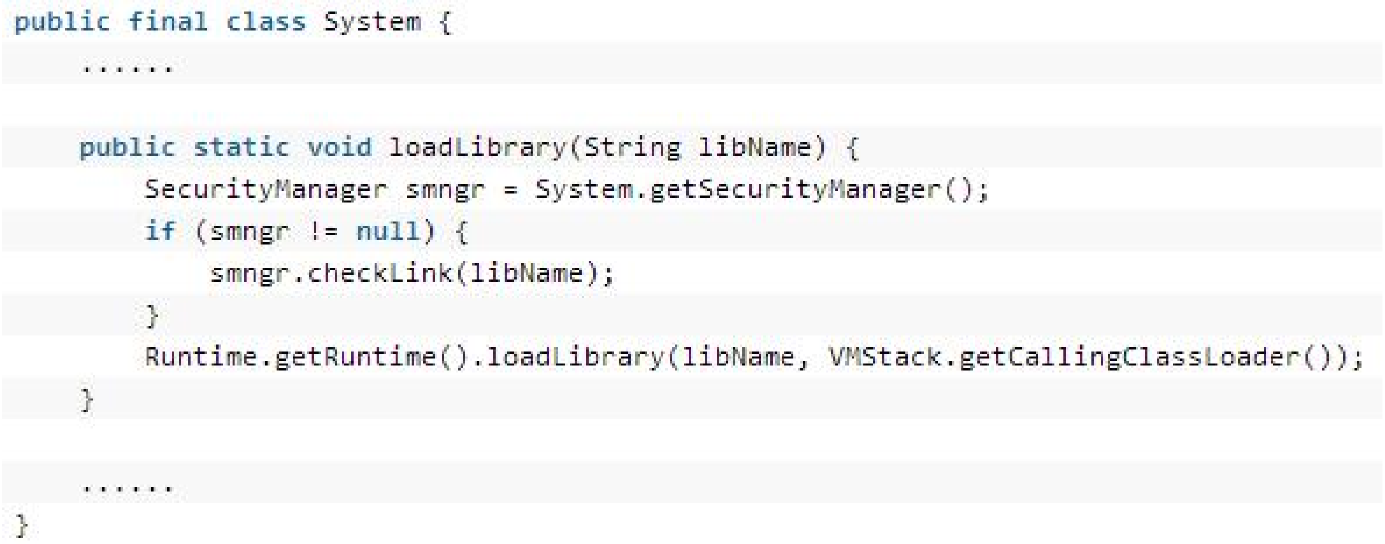
- JNI函数注册示例 -- shy_luo_jni_ClassWithJni_nanosleep



```
static jint shy_luo_jni_ClassWithJni_nanosleep(JNIEnv* env, jobject clazz, jlong seconds,
{
    struct timespec req;
    req.tv_sec  = seconds;
    req.tv_nsec = nanoseconds;

    return nanosleep(&req, NULL);
}

static const JNINativeMethod method_table[] = {
    {"nanosleep", "(JJ)I", (void*)shy_luo_jni_ClassWithJni_nanosleep},
};

extern "C" jint JNI_OnLoad(JavaVM* vm, void* reserved)
{
      JNIEnv* env = NULL;
    jint result = -1;

    if (vm->GetEnv((void**) &env, JNI_VERSION_1_4) != JNI_OK) {
        return result;
    }

    jniRegisterNativeMethods(env, "shy/luo/jni/ClassWithJni", method_table, NELEM(method_t

    return JNI_VERSION_1_4;
}
```

JNI函数的注册过程

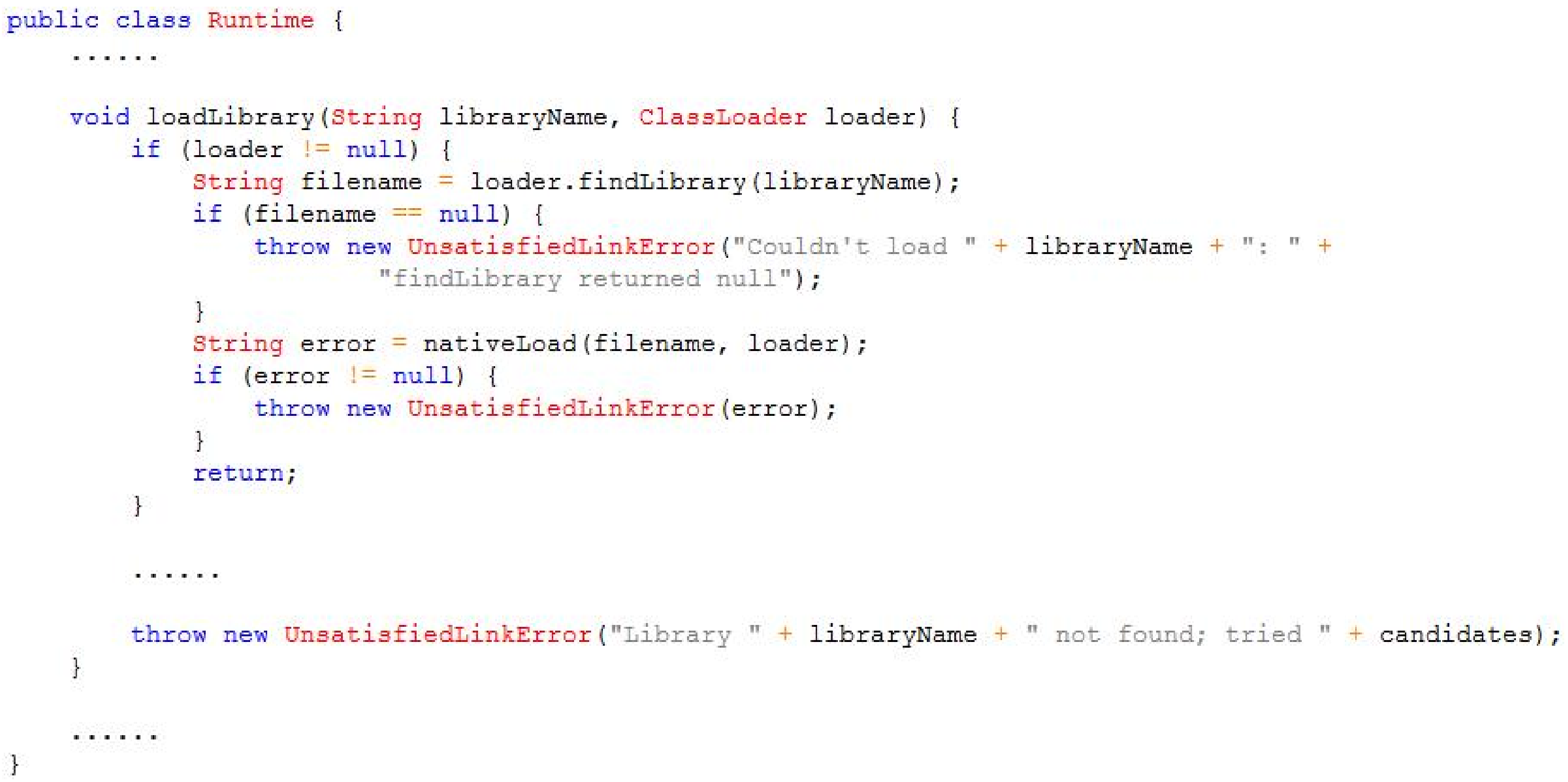
- System.loadLibrary



```
public final class System {
    ......

    public static void loadLibrary(String libName) {
        SecurityManager smngr = System.getSecurityManager();
        if (smngr != null) {
            smngr.checkLink(libName);
        }
        Runtime.getRuntime().loadLibrary(libName, VMStack.getCallingClassLoader());
    }

    ......
}
```

JNI函数的注册过程

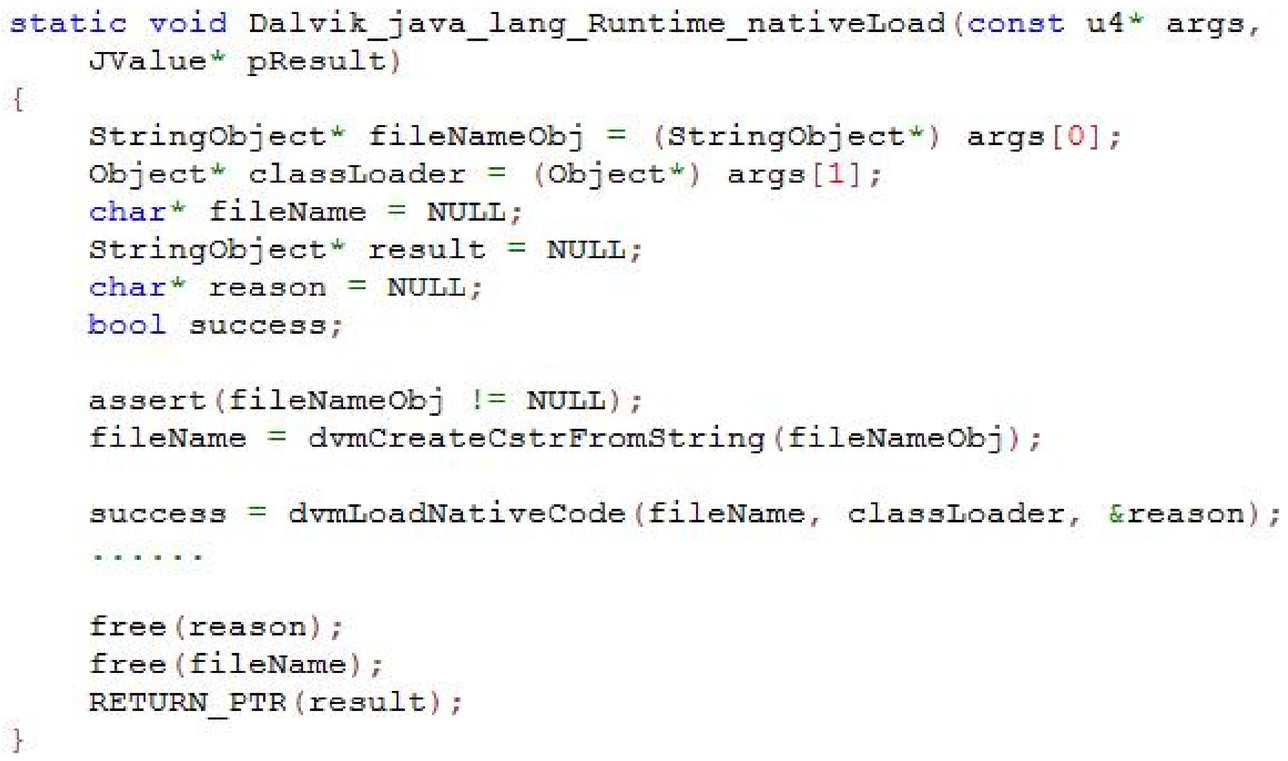
- Runtime.loadLibrary



```
public class Runtime {
    ......

    void loadLibrary(String libraryName, ClassLoader loader) {
        if (loader != null) {
            String filename = loader.findLibrary(libraryName);
            if (filename == null) {
                throw new UnsatisfiedLinkError("Couldn't load " + libraryName + ": " +
                        "findLibrary returned null");
            }
            String error = nativeLoad(filename, loader);
            if (error != null) {
                throw new UnsatisfiedLinkError(error);
            }
            return;
        }

        ......

        throw new UnsatisfiedLinkError("Library " + libraryName + " not found; tried " + candidates);
    }

    ......
}
```

JNI函数的注册过程

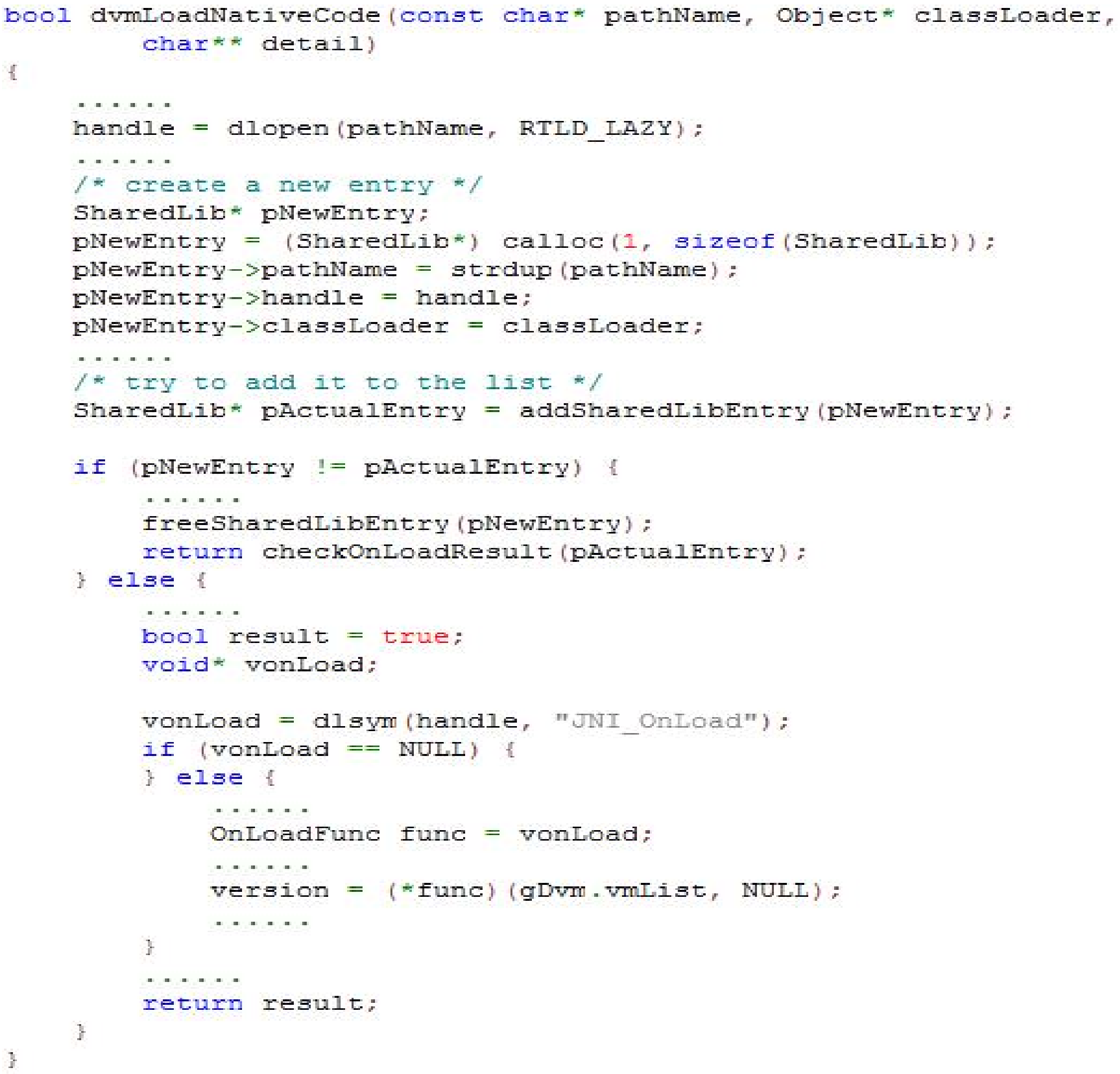
- Runtime.nativeLoad



```
static void Dalvik_java_lang_Runtime_nativeLoad(const u4* args,
    JValue* pResult)
{
    StringObject* fileNameObj = (StringObject*) args[0];
    Object* classLoader = (Object*) args[1];
    char* fileName = NULL;
    StringObject* result = NULL;
    char* reason = NULL;
    bool success;

    assert(fileNameObj != NULL);
    fileName = dvmCreateCstrFromString(fileNameObj);

    success = dvmLoadNativeCode(fileName, classLoader, &reason);
    ......

    free(reason);
    free(fileName);
    RETURN_PTR(result);
}
```

JNI函数的注册过程

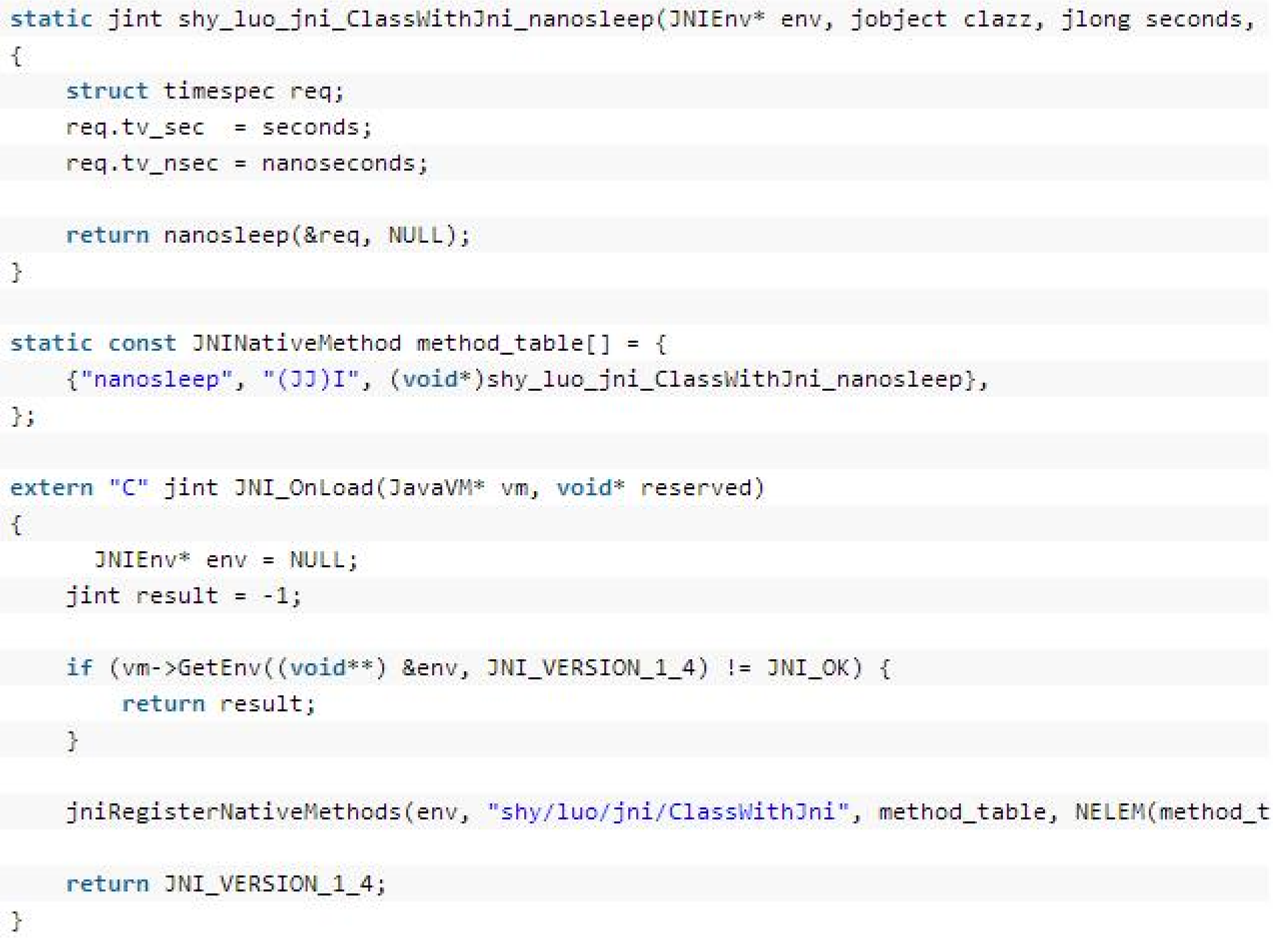
- dvmLoadNativeCode



```
bool dvmLoadNativeCode(const char* pathName, Object* classLoader,
        char** detail)
{
    ......
    handle = dlopen(pathName, RTLD_LAZY);
    ......
    /* create a new entry */
    SharedLib* pNewEntry;
    pNewEntry = (SharedLib*) calloc(1, sizeof(SharedLib));
    pNewEntry->pathName = strdup(pathName);
    pNewEntry->handle = handle;
    pNewEntry->classLoader = classLoader;
    ......
    /* try to add it to the list */
    SharedLib* pActualEntry = addSharedLibEntry(pNewEntry);

    if (pNewEntry != pActualEntry) {
        ......
        freeSharedLibEntry(pNewEntry);
        return checkOnLoadResult(pActualEntry);
    } else {
        ......
        bool result = true;
        void* vonLoad;

        vonLoad = dlsym(handle, "JNI_OnLoad");
        if (vonLoad == NULL) {
        } else {
            ......
            OnLoadFunc func = vonLoad;
            ......
            version = (*func)(gDvm.vmList, NULL);
            ......
        }
        ......
        return result;
    }
}
```

JNI函数的注册过程

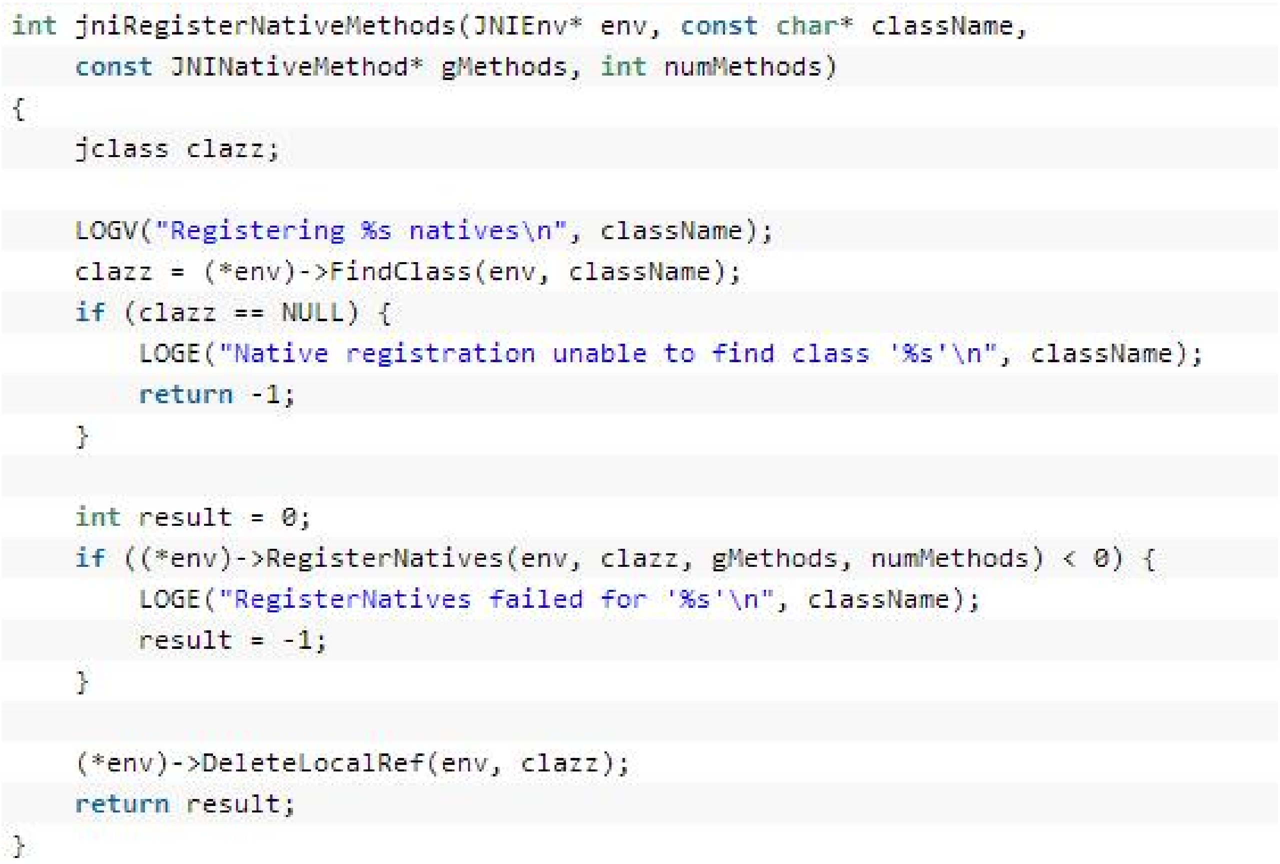
- JNI_OnLoad



```
static jint shy_luo_jni_ClassWithJni_nanosleep(JNIEnv* env, jobject clazz, jlong seconds,
{
    struct timespec req;
    req.tv_sec  = seconds;
    req.tv_nsec = nanoseconds;

    return nanosleep(&req, NULL);
}

static const JNINativeMethod method_table[] = {
    {"nanosleep", "(JJ)I", (void*)shy_luo_jni_ClassWithJni_nanosleep},
};

extern "C" jint JNI_OnLoad(JavaVM* vm, void* reserved)
{
      JNIEnv* env = NULL;
    jint result = -1;

    if (vm->GetEnv((void**) &env, JNI_VERSION_1_4) != JNI_OK) {
        return result;
    }

    jniRegisterNativeMethods(env, "shy/luo/jni/ClassWithJni", method_table, NELEM(method_t

    return JNI_VERSION_1_4;
}
```

JNI函数的注册过程

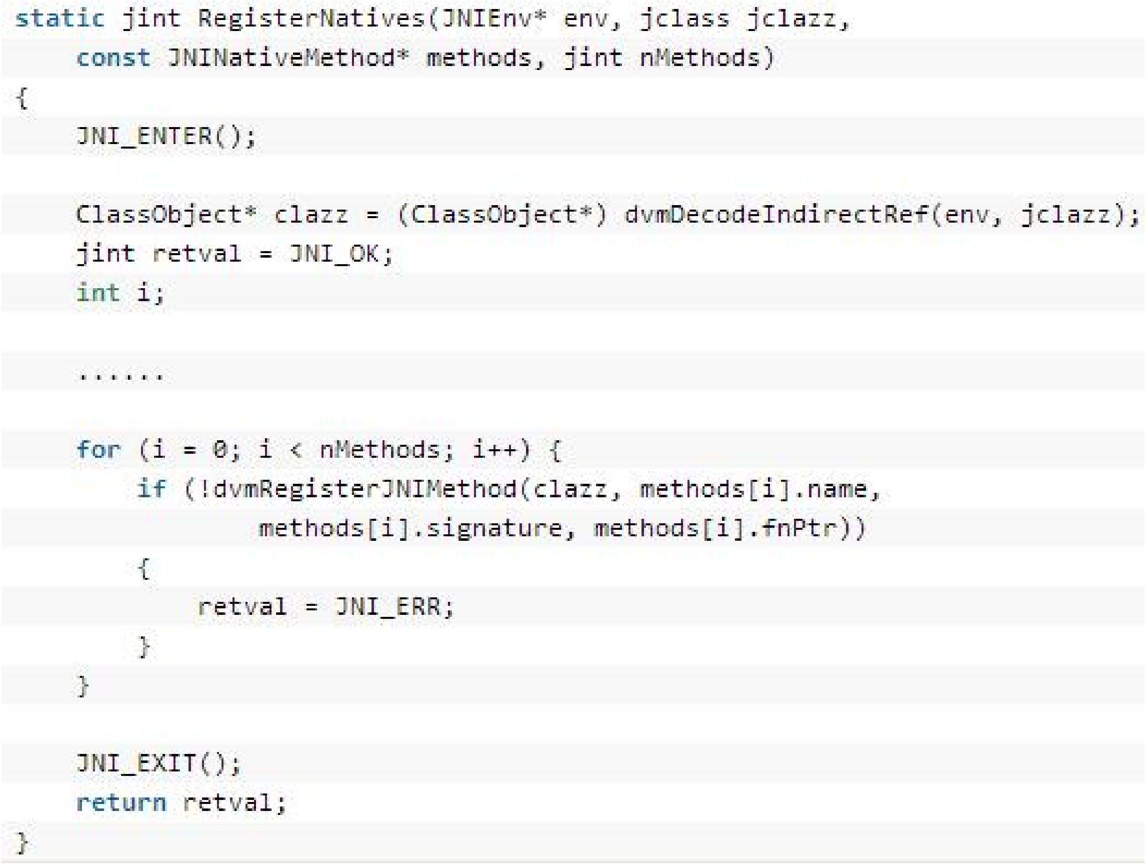
- jniRegisterNativeMethods



```
int jniRegisterNativeMethods(JNIEnv* env, const char* className,
    const JNINativeMethod* gMethods, int numMethods)
{
    jclass clazz;

    LOGV("Registering %s natives\n", className);
    clazz = (*env)->FindClass(env, className);
    if (clazz == NULL) {
        LOGE("Native registration unable to find class '%s'\n", className);
        return -1;
    }

    int result = 0;
    if ((*env)->RegisterNatives(env, clazz, gMethods, numMethods) < 0) {
        LOGE("RegisterNatives failed for '%s'\n", className);
        result = -1;
    }

    (*env)->DeleteLocalRef(env, clazz);
    return result;
}
```

JNI函数的注册过程

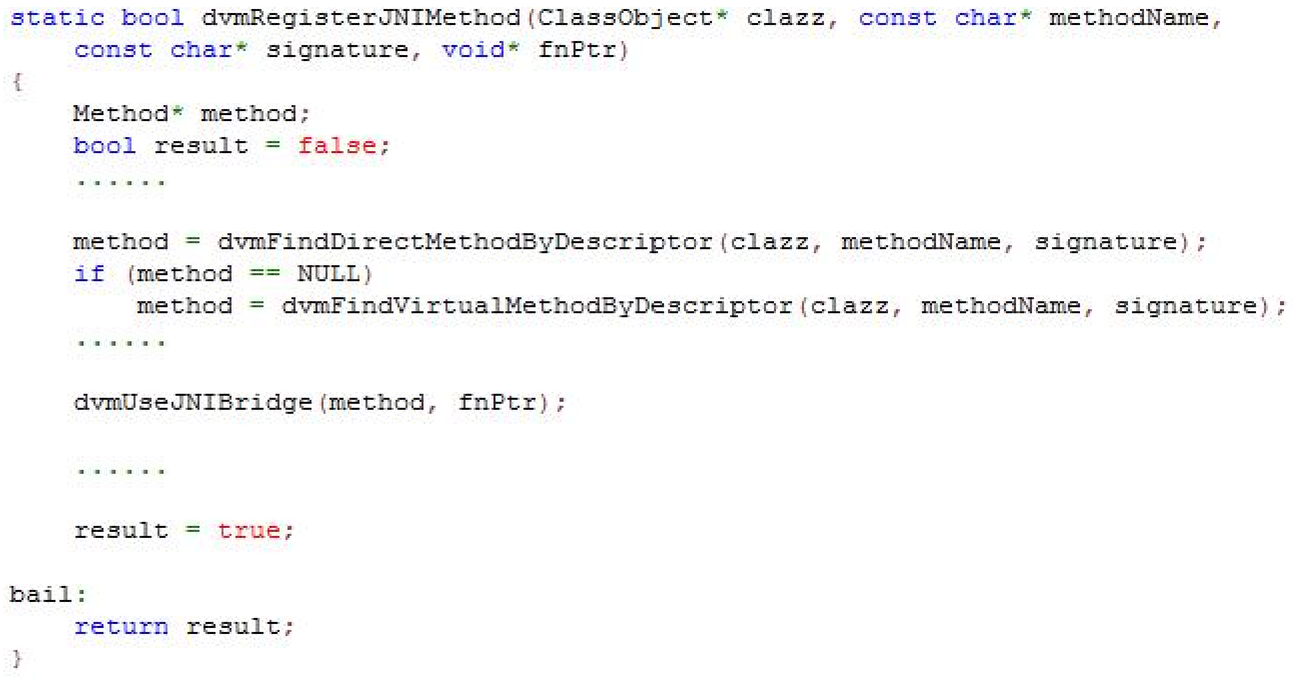
- RegisterNatives



```
static jint RegisterNatives(JNIEnv* env, jclass jclazz,
    const JNINativeMethod* methods, jint nMethods)
{
    JNI_ENTER();

    ClassObject* clazz = (ClassObject*) dvmDecodeIndirectRef(env, jclazz);
    jint retval = JNI_OK;
    int i;

    ......

    for (i = 0; i < nMethods; i++) {
        if (!dvmRegisterJNIMethod(clazz, methods[i].name,
                methods[i].signature, methods[i].fnPtr))
        {
            retval = JNI_ERR;
        }
    }

    JNI_EXIT();
    return retval;
}
```

JNI函数的注册过程

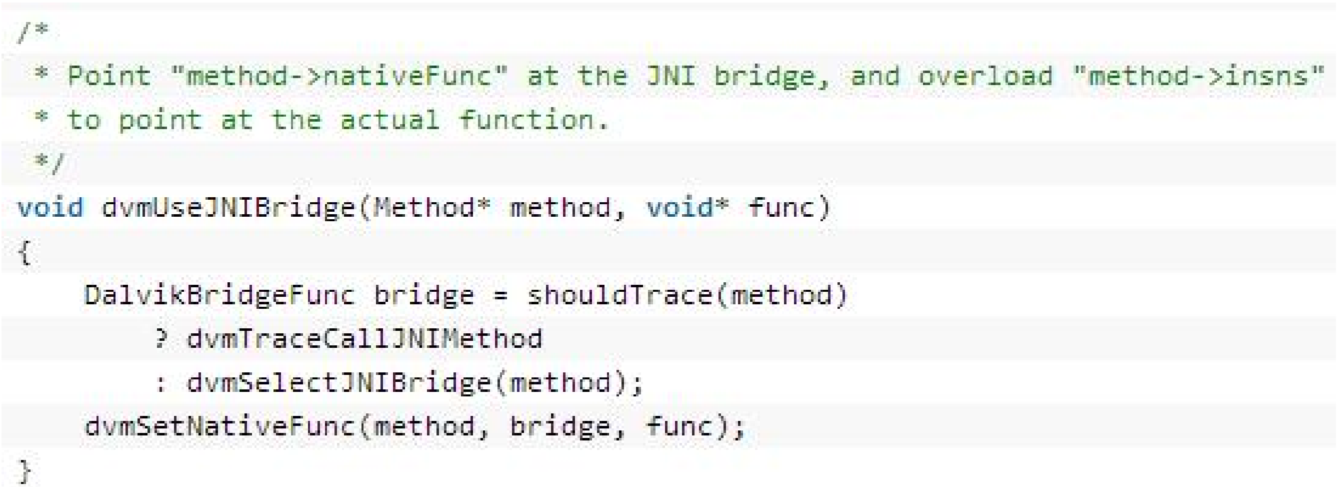
- dvmRegisterJNIMethod



```
static bool dvmRegisterJNIMethod(ClassObject* clazz, const char* methodName,
    const char* signature, void* fnPtr)
{
    Method* method;
    bool result = false;
    ......

    method = dvmFindDirectMethodByDescriptor(clazz, methodName, signature);
    if (method == NULL)
        method = dvmFindVirtualMethodByDescriptor(clazz, methodName, signature);
    ......

    dvmUseJNIBridge(method, fnPtr);

    ......

    result = true;

bail:
    return result;
}
```

JNI函数的注册过程

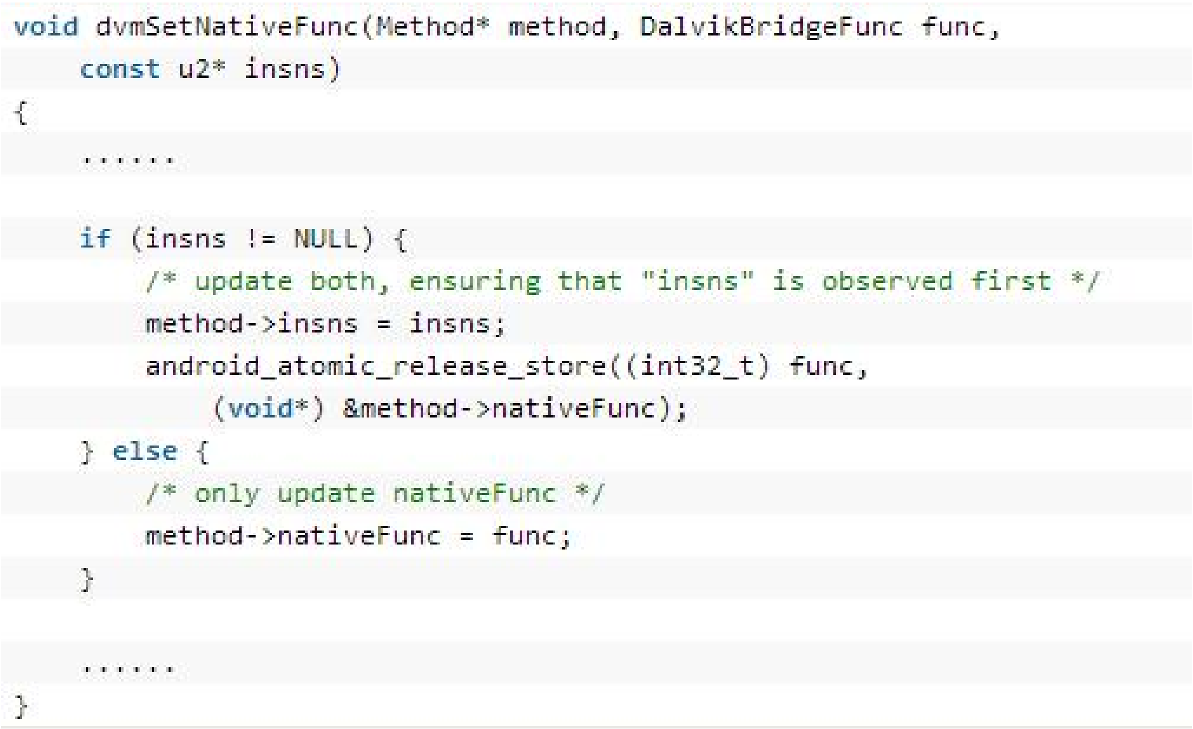
- dvmUseJNIBridge



```
/*
 * Point "method->nativeFunc" at the JNI bridge, and overload "method->insns"
 * to point at the actual function.
 */
void dvmUseJNIBridge(Method* method, void* func)
{
    DalvikBridgeFunc bridge = shouldTrace(method)
        ? dvmTraceCallJNIMethod
        : dvmSelectJNIBridge(method);
    dvmSetNativeFunc(method, bridge, func);
}
```

JNI函数的注册过程

- dvmSetNativeFunc



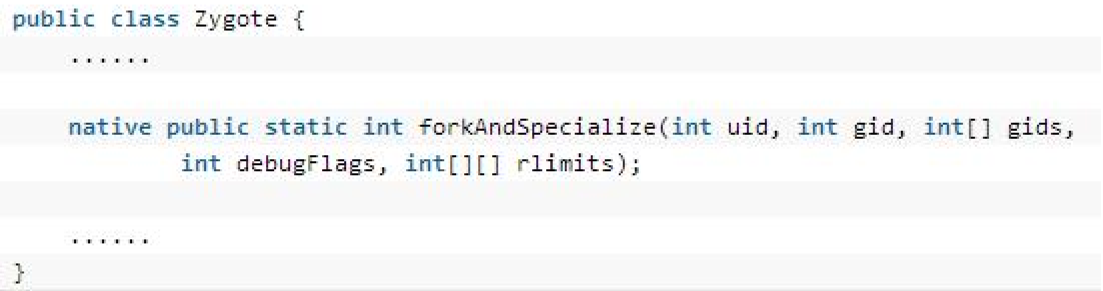
```
void dvmSetNativeFunc(Method* method, DalvikBridgeFunc func,
    const u2* insns)
{
    ......

    if (insns != NULL) {
        /* update both, ensuring that "insns" is observed first */
        method->insns = insns;
        android_atomic_release_store((int32_t) func,
            (void*) &method->nativeFunc);
    } else {
        /* only update nativeFunc */
        method->nativeFunc = func;
    }

    ......
}
```

Dalvik虚拟机进程

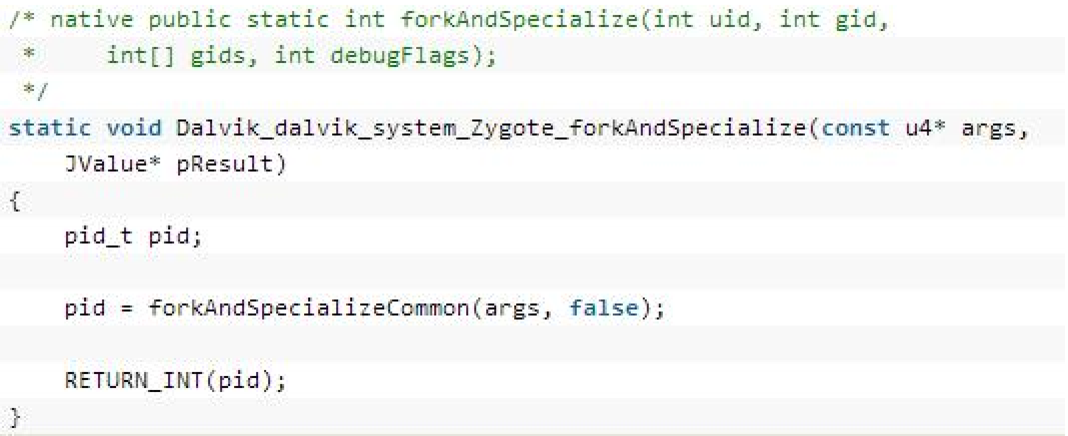
- Dalvik虚拟机进程与下层的Linux进程是一一对应的
- 当ActivityManagerService启动一个组件的时候，发现用来运行该组件的应用程序进程不存在，就会请求Zygote进程创建
- Zygote进程通过调用Zygote类的成员函数forkAndSpecialize来创建

Dalvik虚拟机进程

- Zygote.forkAndSpecialize



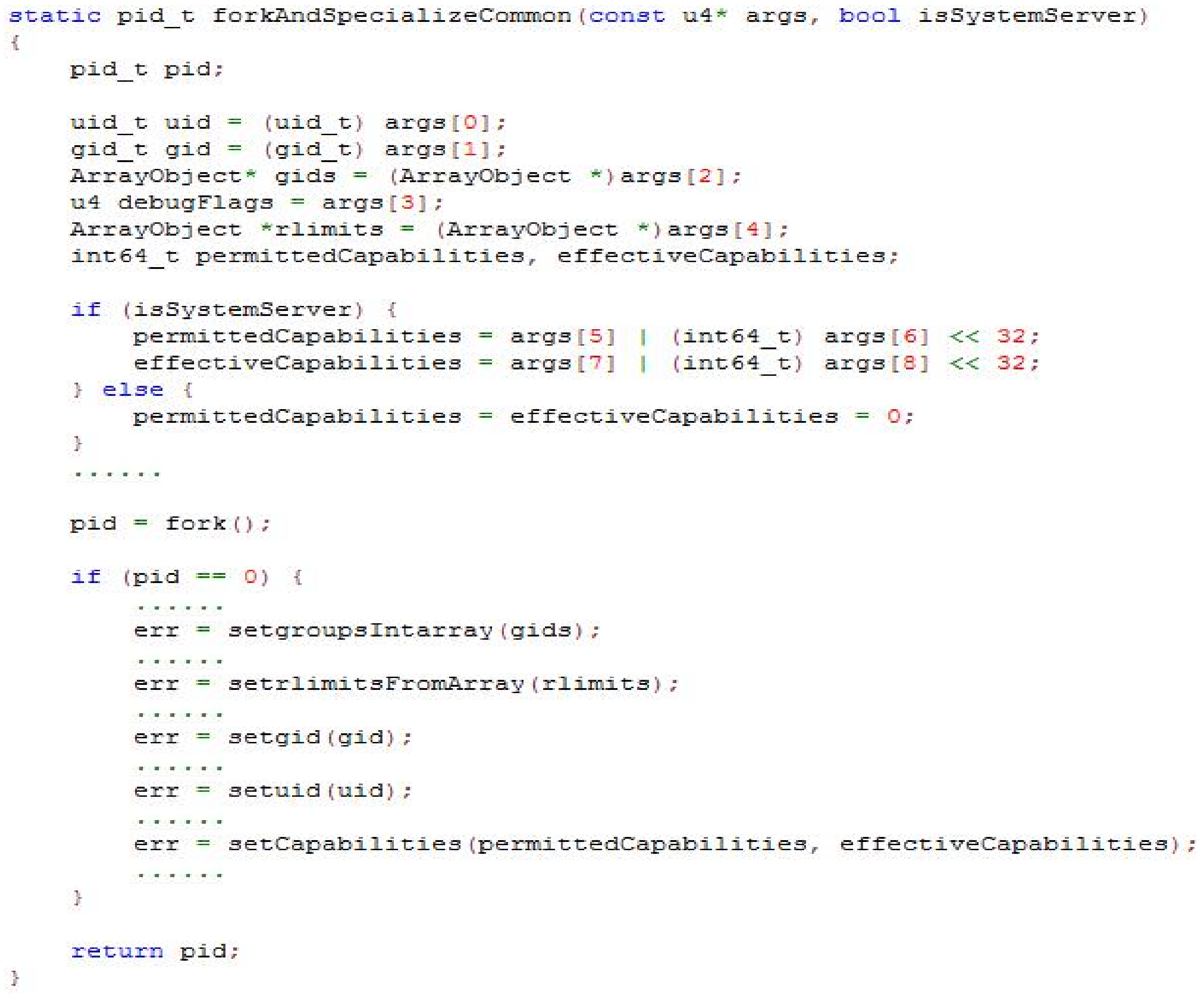
```
public class Zygote {
    ......

    native public static int forkAndSpecialize(int uid, int gid, int[] gids,
            int debugFlags, int[][] rlimits);

    ......
}
```



```
/* native public static int forkAndSpecialize(int uid, int gid,
 *     int[] gids, int debugFlags);
 */
static void Dalvik_dalvik_system_Zygote_forkAndSpecialize(const u4* args,
    JValue* pResult)
{
    pid_t pid;

    pid = forkAndSpecializeCommon(args, false);

    RETURN_INT(pid);
}
```

Dalvik虚拟机进程

- forkAndSpecializeCommon



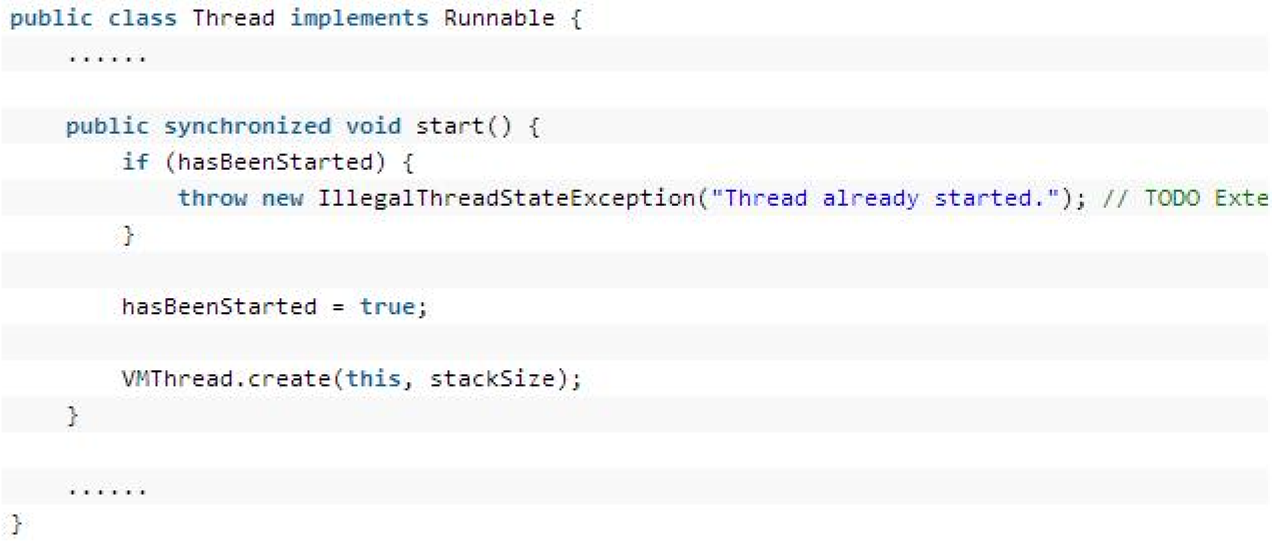
```
static pid_t forkAndSpecializeCommon(const u4* args, bool isSystemServer)
{
    pid_t pid;

    uid_t uid = (uid_t) args[0];
    gid_t gid = (gid_t) args[1];
    ArrayObject* gids = (ArrayObject *)args[2];
    u4 debugFlags = args[3];
    ArrayObject *rlimits = (ArrayObject *)args[4];
    int64_t permittedCapabilities, effectiveCapabilities;

    if (isSystemServer) {
        permittedCapabilities = args[5] | (int64_t) args[6] << 32;
        effectiveCapabilities = args[7] | (int64_t) args[8] << 32;
    } else {
        permittedCapabilities = effectiveCapabilities = 0;
    }
    ......

    pid = fork();

    if (pid == 0) {
        ......
        err = setgroupsIntarray(gids);
        ......
        err = setrlimitsFromArray(rlimits);
        ......
        err = setgid(gid);
        ......
        err = setuid(uid);
        ......
        err = setCapabilities(permittedCapabilities, effectiveCapabilities);
        ......
    }

    return pid;
}
```

Dalvik虚拟机线程

- Dalvik虚拟机线程与下层的Linux线程是一一对应的
- 在Java层中，可以创建一个Thread对象，并且调用该Thread对象的成员函数start来启动一个Dalvik虚拟机线程
- 在Native层中，也可以通过创建一个Thread对象，并且调用该Thread对象的成员函数run来启动一个Dalvik虚拟机线程

Dalvik虚拟机线程

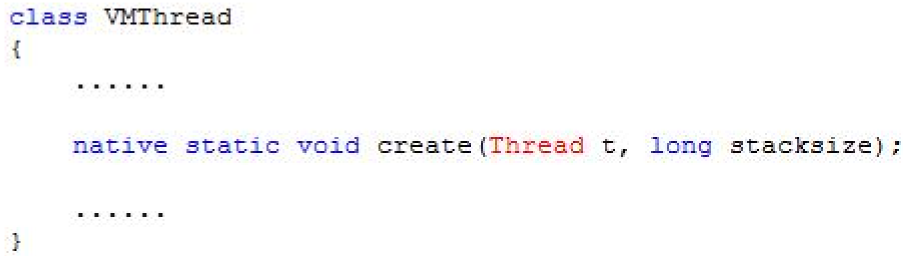
- 在Java层创建Dalvik虚拟机线程--Thread.start



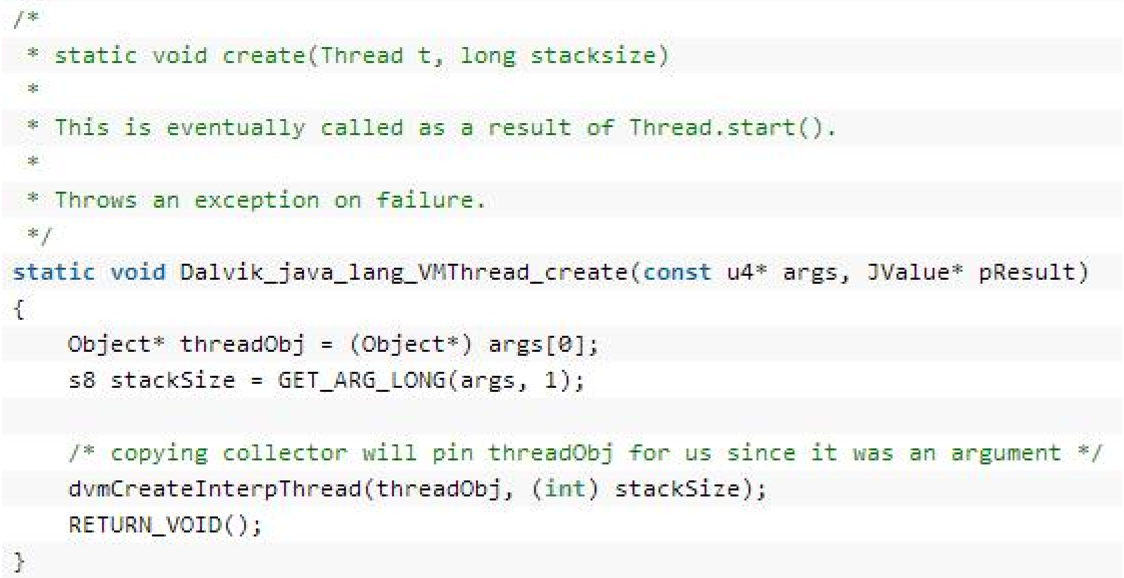
```
public class Thread implements Runnable {
    ......

    public synchronized void start() {
        if (hasBeenStarted) {
            throw new IllegalThreadStateException("Thread already started."); // TODO Exte
        }

        hasBeenStarted = true;

        VMThread.create(this, stackSize);
    }

    ......
}
```

Dalvik虚拟机线程

- VMThread.create



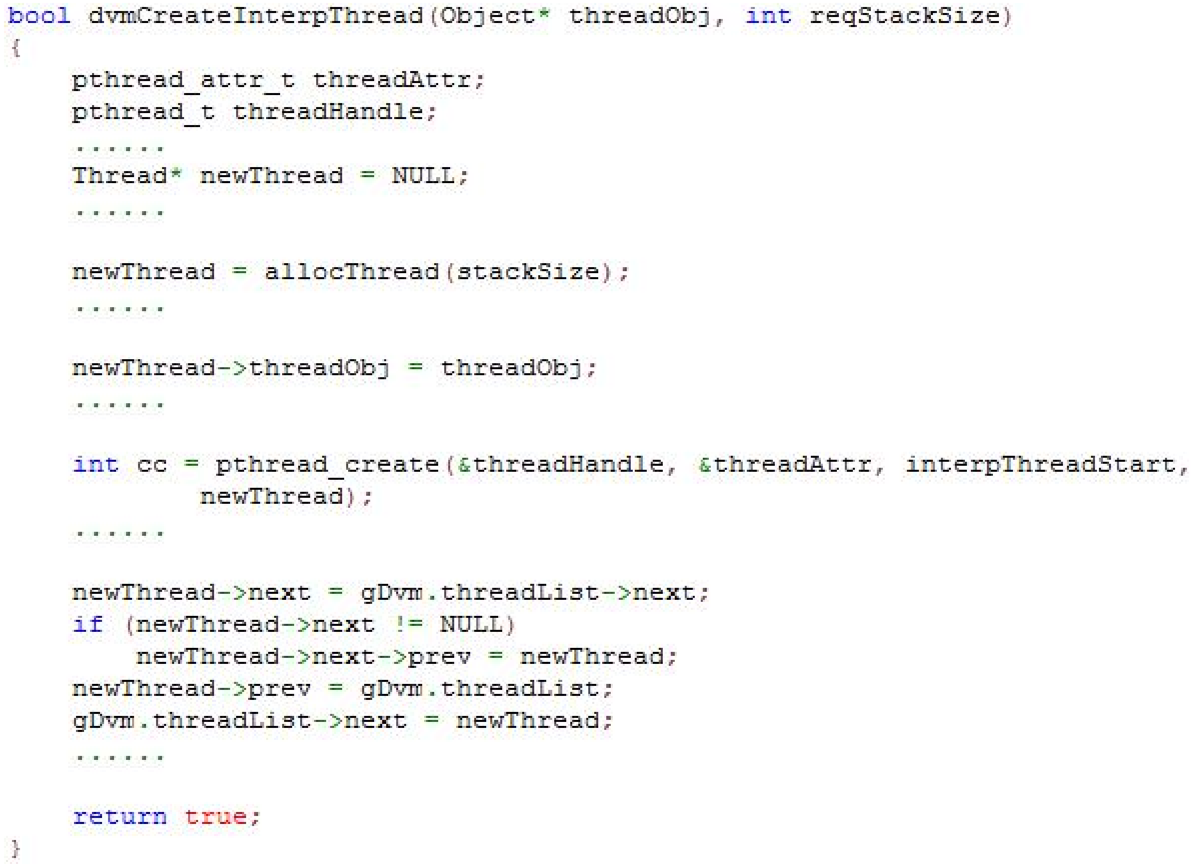
```
class VMThread
{
    ......

    native static void create(Thread t, long stacksize);

    ......
}
```



```
/*
 * static void create(Thread t, long stacksize)
 *
 * This is eventually called as a result of Thread.start().
 *
 * Throws an exception on failure.
 */
static void Dalvik_java_lang_VMThread_create(const u4* args, JValue* pResult)
{
    Object* threadObj = (Object*) args[0];
    s8 stackSize = GET_ARG_LONG(args, 1);

    /* copying collector will pin threadObj for us since it was an argument */
    dvmCreateInterpThread(threadObj, (int) stackSize);
    RETURN_VOID();
}
```

Dalvik虚拟机线程

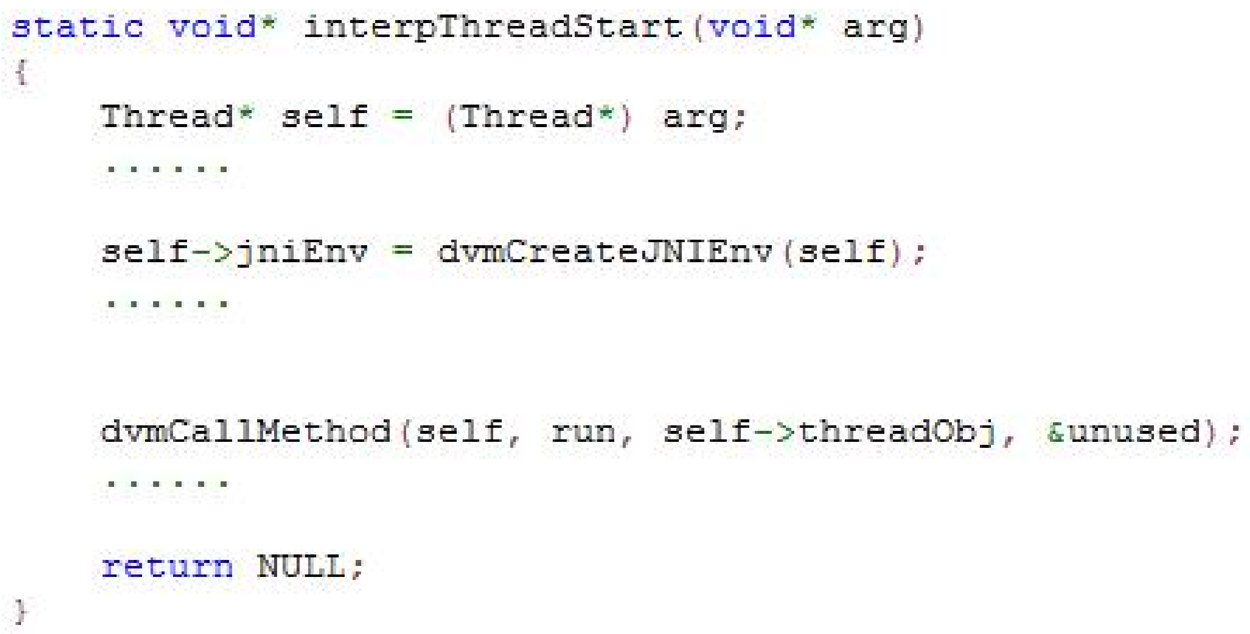
- dvmCreateInterpThread



```
bool dvmCreateInterpThread(Object* threadObj, int reqStackSize)
{
    pthread_attr_t threadAttr;
    pthread_t threadHandle;
    ......
    Thread* newThread = NULL;
    ......

    newThread = allocThread(stackSize);
    ......

    newThread->threadObj = threadObj;
    ......

    int cc = pthread_create(&threadHandle, &threadAttr, interpThreadStart,
            newThread);
    ......

    newThread->next = gDvm.threadList->next;
    if (newThread->next != NULL)
        newThread->next->prev = newThread;
    newThread->prev = gDvm.threadList;
    gDvm.threadList->next = newThread;
    ......

    return true;
}
```

Dalvik虚拟机线程

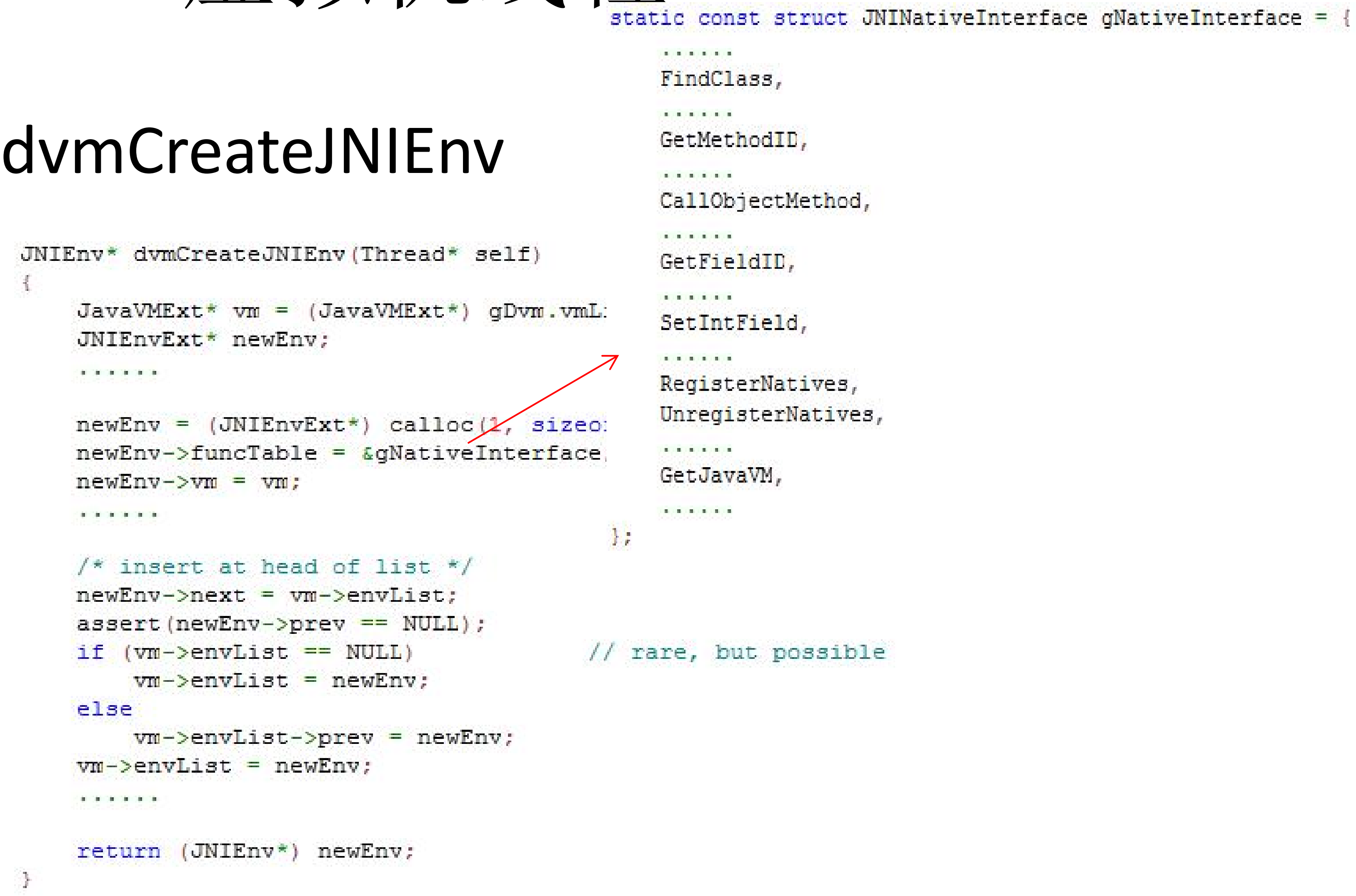
- 线程启动函数：interpThreadStart



```
static void* interpThreadStart(void* arg)
{
    Thread* self = (Thread*) arg;
    ......

    self->jniEnv = dvmCreateJNIEnv(self);
    ......


    dvmCallMethod(self, run, self->threadObj, &unused);
    ......

    return NULL;
}
```

Dalvik虚拟机线程

- dvmCreateJNIEnv

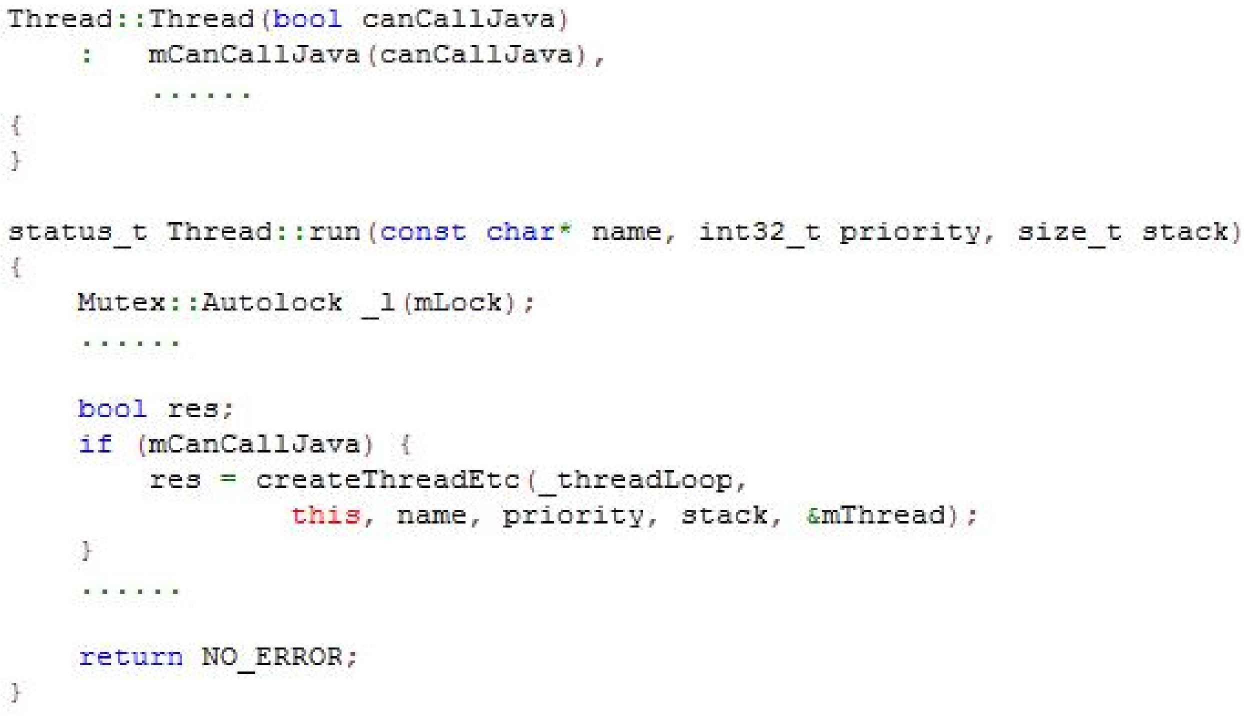


```
JNIEnv* dvmCreateJNIEnv(Thread* self)
{
    JavaVMExt* vm = (JavaVMExt*) gDvm.vmL
    JNIEnvExt* newEnv;
    ......

    newEnv = (JNIEnvExt*) calloc(1, sizeo
    newEnv->funcTable = &gNativeInterface,
    newEnv->vm = vm;
    ......

    /* 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;
    ......

    return (JNIEnv*) newEnv;
}
```

```
static const struct JNINativeInterface gNativeInterface = {
    ......
    FindClass,
    ......
    GetMethodID,
    ......
    CallObjectMethod,
    ......
    GetFieldID,
    ......
    SetIntField,
    ......
    RegisterNatives,
    UnregisterNatives,
    ......
    GetJavaVM,
    ......
};
```

Dalvik虚拟机线程

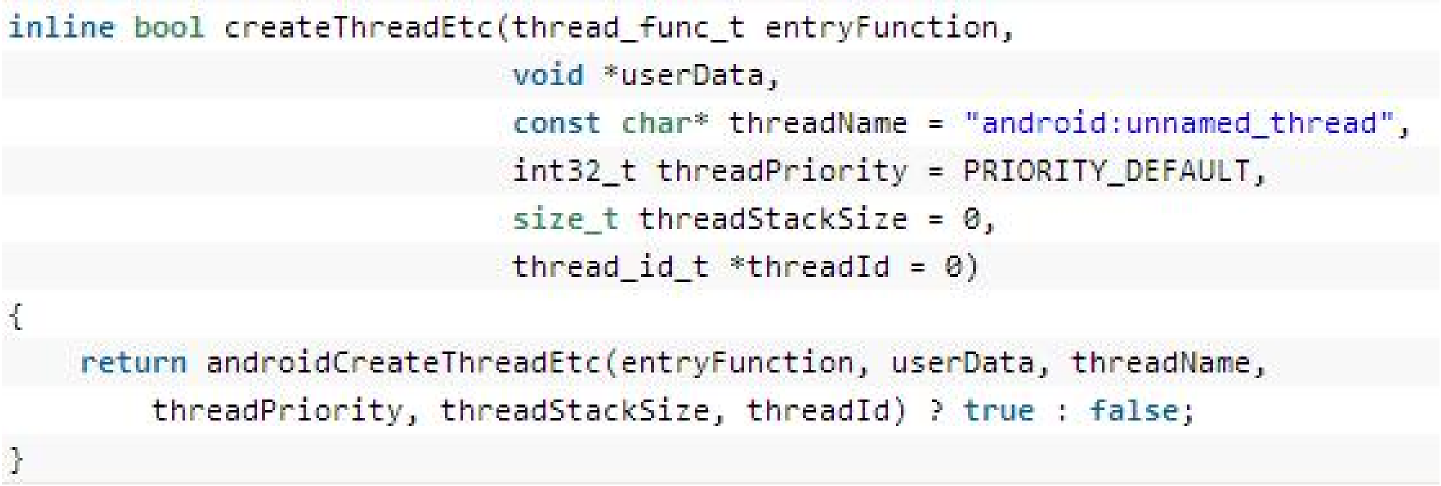
- 在Native层创建Dalvik虚拟机线程--Thread::run



```
Thread::Thread(bool canCallJava)
    :   mCanCallJava(canCallJava),
        ......
{
}

status_t Thread::run(const char* name, int32_t priority, size_t stack)
{
    Mutex::Autolock _l(mLock);
    ......

    bool res;
    if (mCanCallJava) {
        res = createThreadEtc(_threadLoop,
                this, name, priority, stack, &mThread);
    }
    ......

    return NO_ERROR;
}
```

Dalvik虚拟机线程

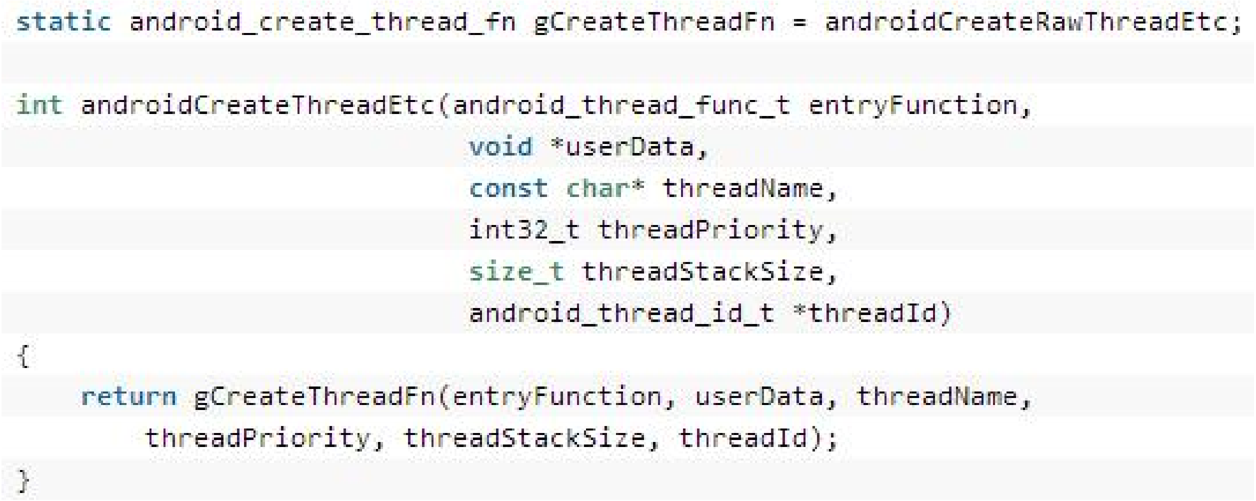
- createThreadEtc



```
inline bool createThreadEtc(thread_func_t entryFunction,
                            void *userData,
                            const char* threadName = "android:unnamed_thread",
                            int32_t threadPriority = PRIORITY_DEFAULT,
                            size_t threadStackSize = 0,
                            thread_id_t *threadId = 0)
{
    return androidCreateThreadEtc(entryFunction, userData, threadName,
        threadPriority, threadStackSize, threadId) ? true : false;
}
```

Dalvik虚拟机线程

- androidCreateThreadEtc

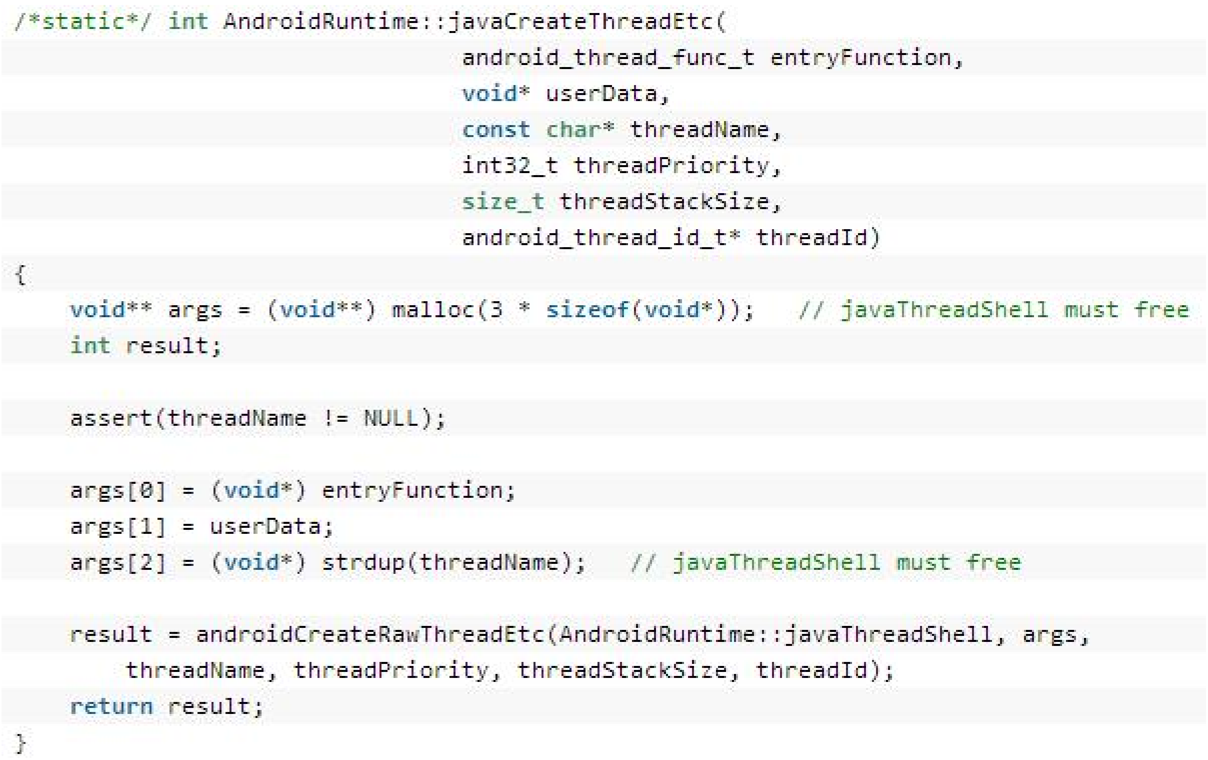


```
static android_create_thread_fn gCreateThreadFn = androidCreateRawThreadEtc;

int androidCreateThreadEtc(android_thread_func_t entryFunction,
                           void *userData,
                           const char* threadName,
                           int32_t threadPriority,
                           size_t threadStackSize,
                           android_thread_id_t *threadId)
{
    return gCreateThreadFn(entryFunction, userData, threadName,
        threadPriority, threadStackSize, threadId);
}
```

  - 注意，函数指针gCreateThreadFn所指向的函数在Dalvik虚拟机启动时已经被修改为javaCreateThreadEtc

Dalvik虚拟机线程

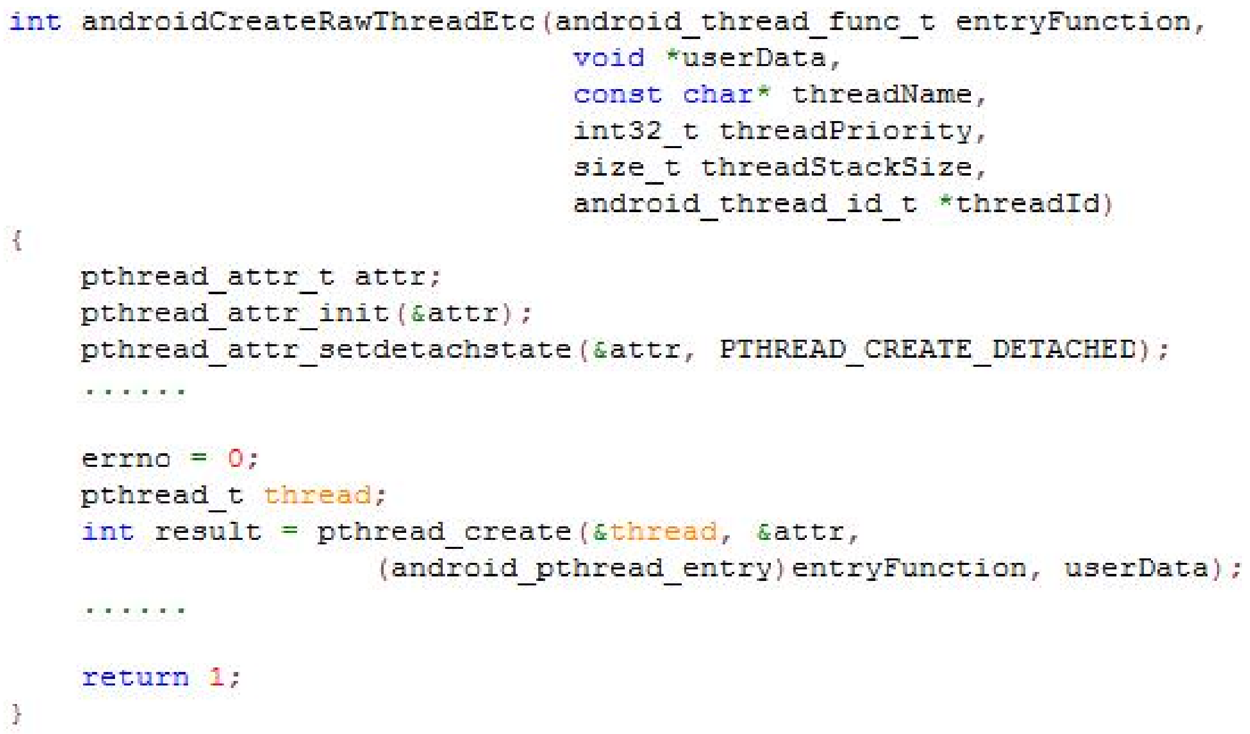
- javaCreateThreadEtc



```
/*static*/ int AndroidRuntime::javaCreateThreadEtc(
                                android_thread_func_t entryFunction,
                                void* userData,
                                const char* threadName,
                                int32_t threadPriority,
                                size_t threadStackSize,
                                android_thread_id_t* threadId)
{
    void** args = (void**) malloc(3 * sizeof(void*));   // javaThreadShell must free
    int result;

    assert(threadName != NULL);

    args[0] = (void*) entryFunction;
    args[1] = userData;
    args[2] = (void*) strdup(threadName);   // javaThreadShell must free

    result = androidCreateRawThreadEtc(AndroidRuntime::javaThreadShell, args,
        threadName, threadPriority, threadStackSize, threadId);
    return result;
}
```

Dalvik虚拟机线程

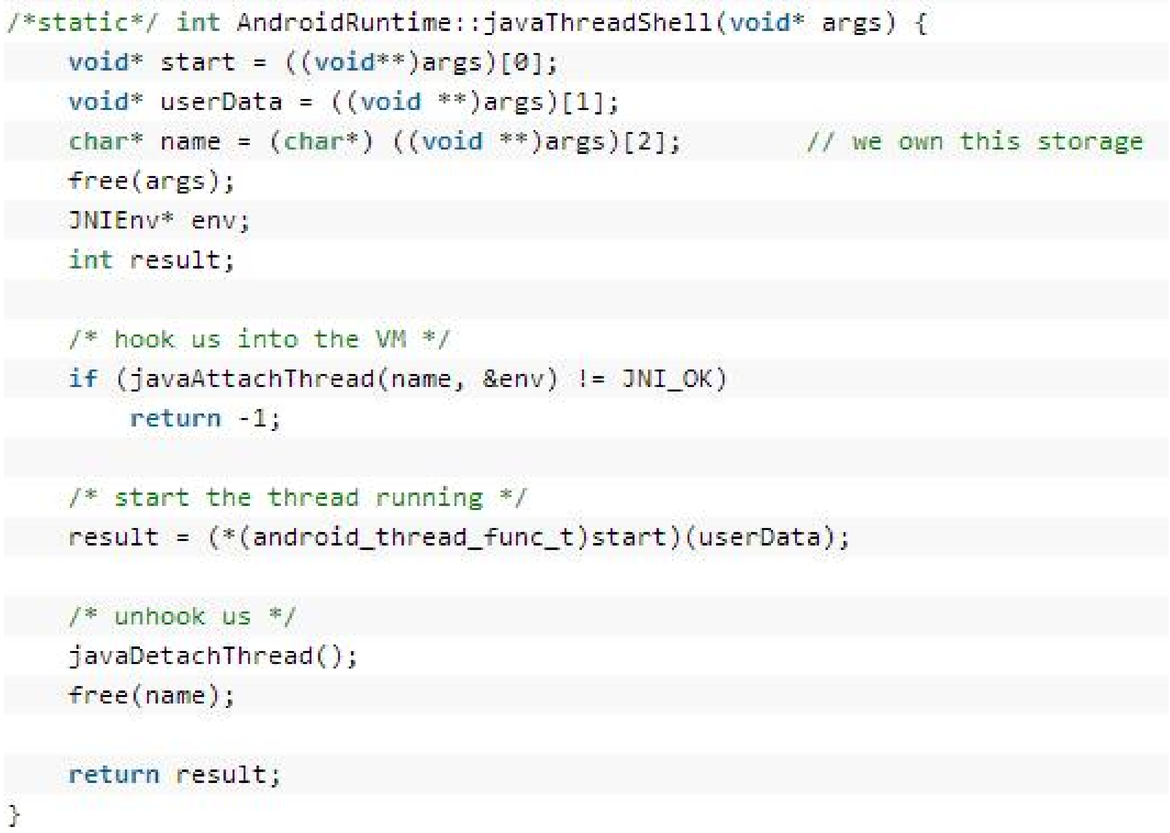
- androidCreateRawThreadEtc



```
int androidCreateRawThreadEtc(android_thread_func_t entryFunction,
                              void *userData,
                              const char* threadName,
                              int32_t threadPriority,
                              size_t threadStackSize,
                              android_thread_id_t *threadId)
{
    pthread_attr_t attr;
    pthread_attr_init(&attr);
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
    ......

    errno = 0;
    pthread_t thread;
    int result = pthread_create(&thread, &attr,
                    (android_pthread_entry)entryFunction, userData);
    ......

    return 1;
}
```

Dalvik虚拟机线程

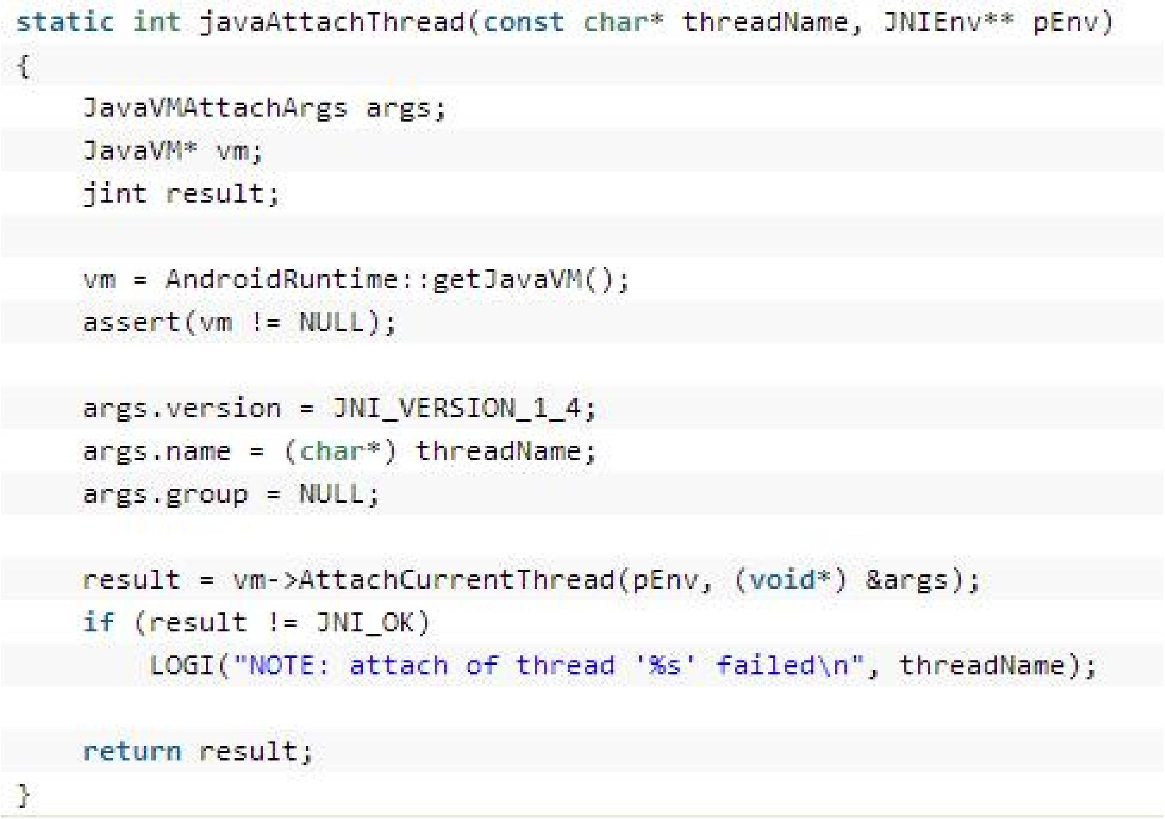
- AndroidRuntime::javaThreadShell



```
/*static*/ int AndroidRuntime::javaThreadShell(void* args) {
    void* start = ((void**)args)[0];
    void* userData = ((void **)args)[1];
    char* name = (char*) ((void **)args)[2];        // we own this storage
    free(args);
    JNIEnv* env;
    int result;

    /* hook us into the VM */
    if (javaAttachThread(name, &env) != JNI_OK)
        return -1;

    /* start the thread running */
    result = (*(android_thread_func_t)start)(userData);

    /* unhook us */
    javaDetachThread();
    free(name);

    return result;
}
```

Dalvik虚拟机线程

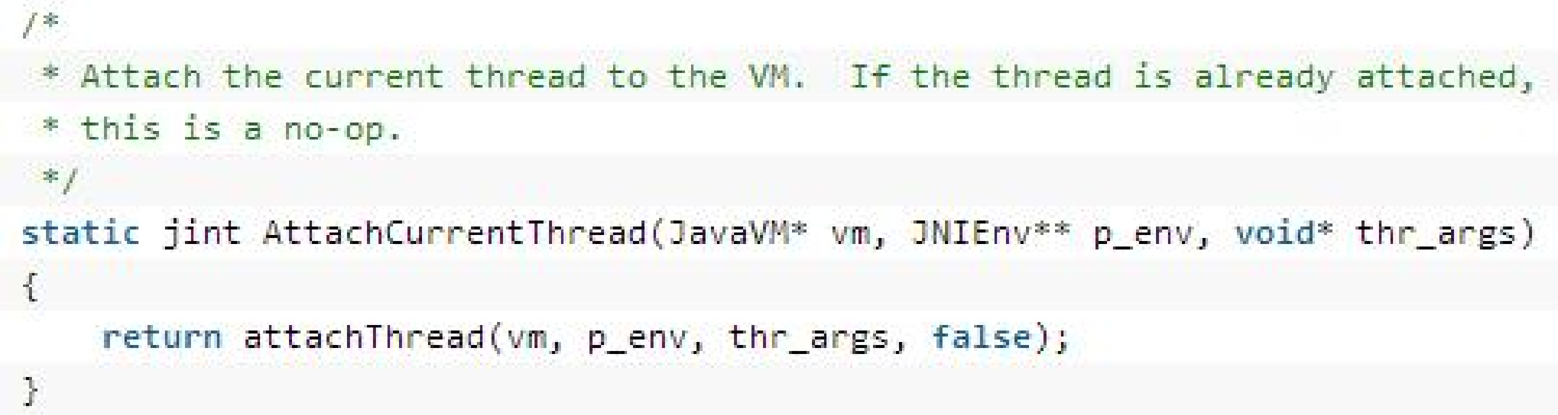
- javaAttachThread



```
static int javaAttachThread(const char* threadName, JNIEnv** pEnv)
{
    JavaVMAttachArgs args;
    JavaVM* vm;
    jint result;

    vm = AndroidRuntime::getJavaVM();
    assert(vm != NULL);

    args.version = JNI_VERSION_1_4;
    args.name = (char*) threadName;
    args.group = NULL;

    result = vm->AttachCurrentThread(pEnv, (void*) &args);
    if (result != JNI_OK)
        LOGI("NOTE: attach of thread '%s' failed\n", threadName);

    return result;
}
```

Dalvik虚拟机线程

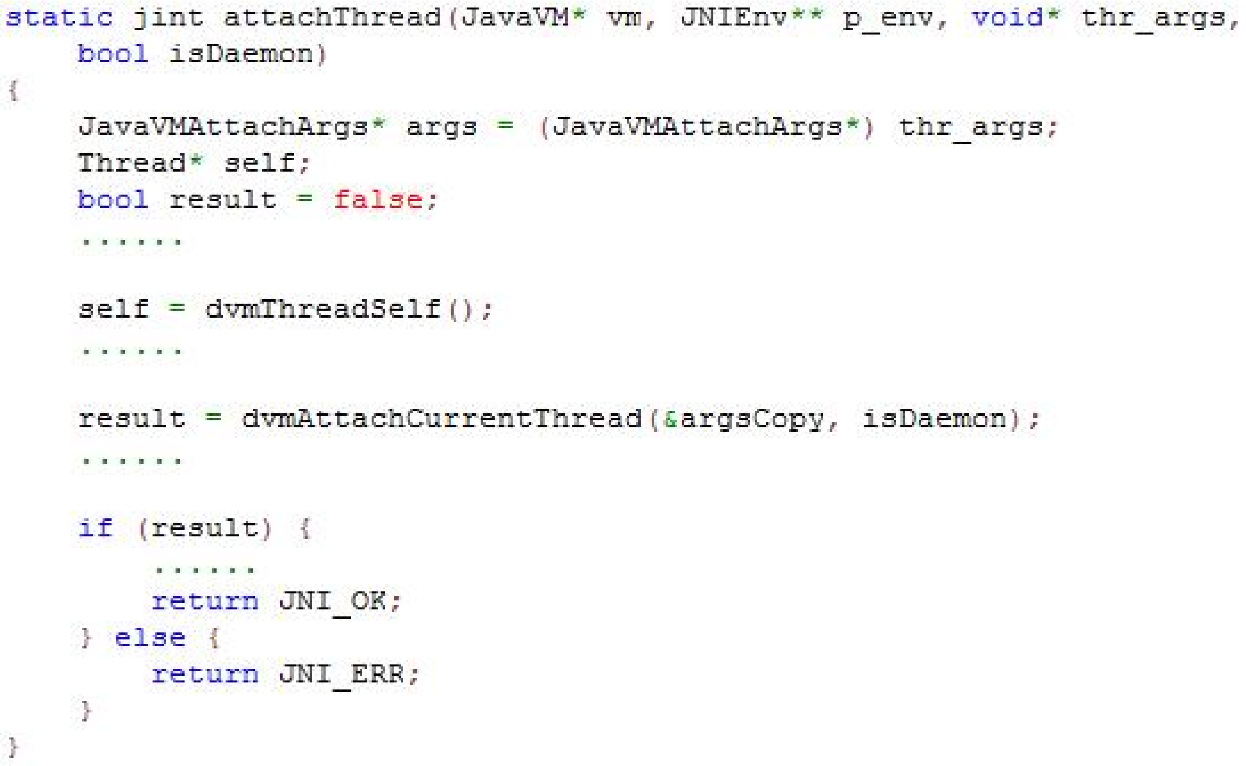
- AttachCurrentThread



```
/*
 * Attach the current thread to the VM.  If the thread is already attached,
 * this is a no-op.
 */
static jint AttachCurrentThread(JavaVM* vm, JNIEnv** p_env, void* thr_args)
{
    return attachThread(vm, p_env, thr_args, false);
}
```

Dalvik虚拟机线程

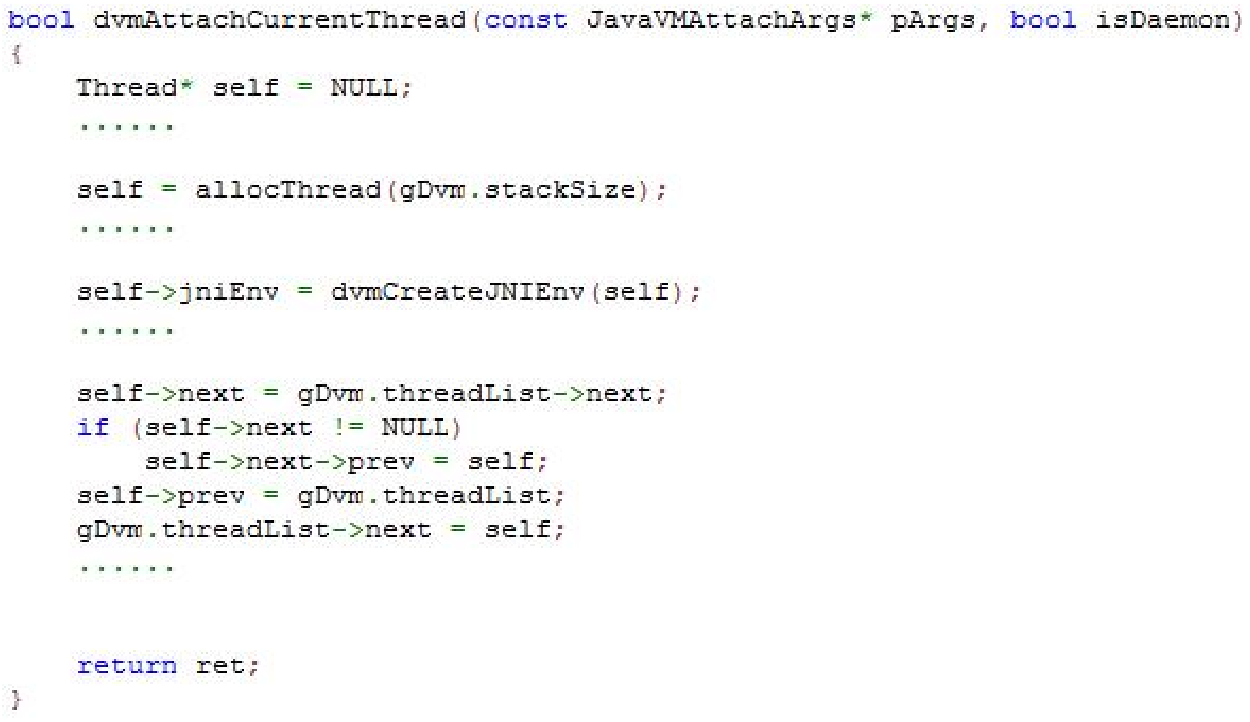
- attachThread



```
static jint attachThread(JavaVM* vm, JNIEnv** p_env, void* thr_args,
    bool isDaemon)
{
    JavaVMAttachArgs* args = (JavaVMAttachArgs*) thr_args;
    Thread* self;
    bool result = false;
    ......

    self = dvmThreadSelf();
    ......

    result = dvmAttachCurrentThread(&argsCopy, isDaemon);
    ......

    if (result) {
        ......
        return JNI_OK;
    } else {
        return JNI_ERR;
    }
}
```

Dalvik虚拟机线程

- dvmAttachCurrentThread



```
bool dvmAttachCurrentThread(const JavaVMAttachArgs* pArgs, bool isDaemon)
{
    Thread* self = NULL;
    ......

    self = allocThread(gDvm.stackSize);
    ......

    self->jniEnv = dvmCreateJNIEnv(self);
    ......

    self->next = gDvm.threadList->next;
    if (self->next != NULL)
        self->next->prev = self;
    self->prev = gDvm.threadList;
    gDvm.threadList->next = self;
    ......


    return ret;
}
```

Q&A

Thank You