Android的启动过程-zygote进程

在前一章里介绍了Android runtime的Init进程的启动，在Init进程里，我们知道了runtime环境是如何搭建的和系统属性设置在什么时候设置的，并了解了使用配置文件Init.rc来启动系统Service和其他操作处理。

在这一章里，我们将继续跟进zygote进程的启动过程。在zygote进程启动过程中,虚拟机的创建、jni functions注册及java层ZygoteInit是我们需要关注和理解的，这有助于我们开发更加高效的Android程序。

本文主要以android-7.1.2\_r11源码作为参考

主要代码路径： \frameworks\base\cmds\app\_process\app\_main.cpp

\frameworks\base\core\jni\AndroidRuntime.cpp

\frameworks\base\include\android\_runtime\AndroidRuntime.h

\libnativehelper\JniInvocation.cpp

\libnativehelper\include\nativehelper\JniInvocation.h

\frameworks\base\core\java\android\os\ZygoteInit.java

**zygote进程的启动**

在《Android runtime机制（一）init进程》篇中我们知道，zygote进程是在Init进程中通过执行如下命令启动的

/system/bin/app\_process64//64位系统使用

/system/bin/app\_process32//32位系统使用

执行上述命令后，程序执行到了app\_main.c的main()函数。

**AndroidRuntime**

在介绍zygote进程的启动流程之前，我们先来看看两个重要的类AppRuntime 和AndroidRuntime。这两个类是Android runtime运行环境的接口类，在Android runtime中起着至关重要的作用。

下面是这两个类之间的关系：

class AppRuntime : public AndroidRuntime

从上面的类继承关系中可以看到，AppRuntime类继承自AndroidRuntime。从后续处理中，在Android 创建运行时环境的时候，主要是通过AppRuntime实例操作。

**AndroidRuntime类**

AndroidRuntime类是AppRuntime的父类，基本承载着Android runtime的绝大部分操作处理。

下面是AndroidRuntime类的声明，从其声明中，我们可以看到Android runtime的几个重要的接口：

**公共接口**

* registerNativeMethods函数。JNI编程时，应用程序向虚拟机注册本地方法时使用。JNI编程时非常重要的一个函数接口。
* getRuntime函数。提供给外部获取当前进程AndroidRuntime类实例的接口。
* getJavaVM函数。提供给外部获取当前进程android虚拟机的接口。
* getJNIEnv函数。提供给外部获取当前进程JNI环境的接口。

**私有接口**

* startReg函数。系统级服务的JNI注册函数。
* startVm函数。创建并启动Android虚拟机。
* mJavaVM。指向虚拟机的指针，用以访问Android虚拟机。

具体请参考AndroidRuntime类的声明：\frameworks\base\core\jni\AndroidRuntime.h

**AppRuntime类**

AppRuntime类继承自AndroidRuntime类，重载了AndroidRuntime类的onVmCreated()、onStarted()、onZygoteInit()和onExit()函数，是zygote进程处理时实际runtime入口。

**Android runtime类的功能**

从上面AppRuntime类和AndroidRuntime类的实现声明中可知，AndroidRumtime主要实现如下功能：

* 创建并启动虚拟机
* 注册JNI服务
* 提供外部访问虚拟机的接口
* 提供外部访问JNI环境的接口，并提供JNI程序开发的功能支持。

**zygote进程的功能处理**

zygote进程的入口处理函数是app\_main.cpp中的main()函数。Android run time环境的创建和启动，主要是通过AndroidRuntime来实现的。

zygote进程主要处理：

* app\_main.cpp中的main()处理：解析传入参数，实例化AppRuntime，启动Android runtime运行时环境等
* Android虚拟机的创建
* 向Android虚拟机注册Android native处理函数
* java层Zygote初始化处理

下面我们按照程序执行流程来具体梳理一下整个zygote进程的启动处理过程。

**app\_main.cpp中的main()处理**

**1. 实例化AndroidRuntime类**

///通过执行system/bin/app\_process命令，根据service zygote /system/bin/app\_process -Xzygote /system/bin --zygote --start-system-server传入参数

//argc是传入参数的参数数目：argc=5

//argv是传入的参数值：

// argv[0]="/system/bin/app\_process64"

// argv[1]="-Xzygote"

// argv[2]="/system/bin"

// argv[3]="--zygote"

// argv[4]="--start-system-server"

AppRuntime runtime(argv[0], computeArgBlockSize(argc, argv));

**2. 解析启动参数**

* 忽略命令行参数：”/system/bin/app\_process64”

// Process command line arguments

// ignore argv[0]

//执行此语句块后：

//argc=4

//"/system/bin/app\_process64"被忽略，此时argv剩下的值为：

// argv[1]="-Xzygote"

// argv[2]="/system/bin"

// argv[3]="--zygote"

// argv[4]="--start-system-server"

argc--;

argv++;

* 把以“-”打头的参数传递给虚拟机，对于zygote，是把”-Xzygote”传入虚拟机

//执行此语句块后：

//argc=3

//"-Xzygote"被传递给虚拟机，被忽略，此时argv剩下的值为：

// argv[2]="/system/bin"

// argv[3]="--zygote"

// argv[4]="--start-system-server"

int i;

for (i = 0; i < argc; i++) {

//只有第一个参数"-Xzygote"满足，第二个参数是"/system/bin"，不是以"-"打头的，所以退出for循环。

if (argv[i][0] != '-') {

break;

}

if (argv[i][1] == '-' && argv[i][2] == 0) {

++i; // Skip --.

break;

}

runtime.addOption(strdup(argv[i]));

}

* 解析剩余传入的参数，根据上一个语句块的处理，此时剩余参数值为：”/system/bin”、”–zygote”和”–start-system-server”

// Parse runtime arguments. Stop at first unrecognized option.

bool zygote = false;

bool startSystemServer = false;

bool application = false;

String8 niceName;

String8 className;

++i; // Skip unused "parent dir" argument.

while (i < argc) {

const char\* arg = argv[i++];

//根据参数"--zygote",表示当前进程是zygote进程，zygote=true，并设置nicename为zygote64（64位）或zygote。

if (strcmp(arg, "--zygote") == 0) {

zygote = true;

//static const char ZYGOTE\_NICE\_NAME[] = "zygote64";或

//static const char ZYGOTE\_NICE\_NAME[] = "zygote";

niceName = ZYGOTE\_NICE\_NAME;

}

//根据参数"--start-system-server"，需要启动System Server。

else if (strcmp(arg, "--start-system-server") == 0) {

startSystemServer = true;

} else if (strcmp(arg, "--application") == 0) {

application = true;

} else if (strncmp(arg, "--nice-name=", 12) == 0) {

niceName.setTo(arg + 12);

} else if (strncmp(arg, "--", 2) != 0) {

className.setTo(arg);

break;

} else {

--i;

break;

}

}

**3. 变更zygote进程名为zygote或zygote64**

//niceName在参数解析时被设置。此处变更了app\_process启动的进程名为zygote。

if (!niceName.isEmpty()) {

runtime.setArgv0(niceName.string());

set\_process\_name(niceName.string());

}

**4. 调用AndroidRuntime.start()来创建和启动Android 运行时环境**

//zygote在参数解析时被设置为true。

//"com.android.internal.os.ZygoteInit"是Android Java运行时环境的初始化类。

//zygote=true。

if (zygote) {

runtime.start("com.android.internal.os.ZygoteInit", args, zygote);

} else if (className) {

runtime.start("com.android.internal.os.RuntimeInit", args, zygote);

} else {

fprintf(stderr, "Error: no class name or --zygote supplied.\n");

app\_usage();

LOG\_ALWAYS\_FATAL("app\_process: no class name or --zygote supplied.");

return 10;

}

**Android虚拟机的创建**

Android虚拟机的创建是在AndroidRuntime.start()中处理的。先初始化JniInvocation，然后通过startVm()->JNI\_CreateJavaVM()来创建Android虚拟机。

/\* start the virtual machine \*/

JniInvocation jni\_invocation;

jni\_invocation.Init(NULL);

JNIEnv\* env;

if (startVm(&mJavaVM, &env, zygote) != 0) {

return;

}

onVmCreated(env);

**1. JniInvocation**

JniInvocation，是一个外部和虚拟机之间的一个中间层，是外部访问虚拟机的API接口，允许外部动态的调用虚拟机内部的实现。其主要实现如下功能：

* 指定加载的虚拟机。现在Android系统中有两种虚拟机：dalvik和art。现在已基本使用art虚拟机了。
* 封装虚拟机的对象接口

代码路径：

\libnativehelper\JniInvocation.cpp

\libnativehelper\include\nativehelper\JniInvocation.h

**JniInvocation的主要接口**

//获取默认的虚拟机初始化参数

jint (\*JNI\_GetDefaultJavaVMInitArgs\_)(void\*);

//创建虚拟机

jint (\*JNI\_CreateJavaVM\_)(JavaVM\*\*, JNIEnv\*\*, void\*);

//获取已创建的虚拟机对象

jint (\*JNI\_GetCreatedJavaVMs\_)(JavaVM\*\*, jsize, jsize\*);

**虚拟机库的加载**

在JniInvocation的Init()函数首先会加载虚拟机库文件。

虚拟机库文件有两种获取方式：

* 用户指定
* 系统默认

用户可以通过persist.sys.dalvik.vm.lib.2属性设定虚拟机的库文件，默认的库文件是libart.so。

**接口的初始化**

在虚拟机库文件加载成功后，需要对JniInvocation的接口进行初始化，提供虚拟机接口功能供外部来访问虚拟机。   
在init()函数里，有如下处理：

//初始化JNI\_GetDefaultJavaVMInitArgs\_，调用JNI\_GetDefaultJavaVMInitArgs\_实际上就是调用虚拟机库函数JNI\_GetDefaultJavaVMInitArgs。

if (!FindSymbol(reinterpret\_cast<void\*\*>(&JNI\_GetDefaultJavaVMInitArgs\_),

"JNI\_GetDefaultJavaVMInitArgs")) {

return false;

}

//初始化JNI\_CreateJavaVM\_，调用JNI\_CreateJavaVM\_实际上就是调用虚拟机库函数JNI\_CreateJavaVM。

if (!FindSymbol(reinterpret\_cast<void\*\*>(&JNI\_CreateJavaVM\_),

"JNI\_CreateJavaVM")) {

return false;

}

//初始化JNI\_GetCreatedJavaVMs\_，调用JNI\_GetCreatedJavaVMs\_实际上就是调用虚拟机库函数JNI\_GetCreatedJavaVMs。

if (!FindSymbol(reinterpret\_cast<void\*\*>(&JNI\_GetCreatedJavaVMs\_),

"JNI\_GetCreatedJavaVMs")) {

return false;

}

函数JNI\_CreateJavaVM()、JNI\_GetCreatedJavaVMs()、JNI\_GetDefaultJavaVMInitArgs()定义在art\runtime\java\_vm\_ext.cc下，是art虚拟机的内部处理函数。

**2. startVm的处理**

在JniInvocation初始化之后，虚拟机库文件被加载且接口函数也已被初始化。从JniInvocation的初始化中，我们知道，Android N的默认虚拟机已是ART虚拟机。   
startVm()函数就是用来创建ART虚拟机的。整个的处理过程就是配置虚拟机的属性然后调用虚拟机库函数JNI\_CreateJavaVM()来创建。

在虚拟机属性配置过程中，有如下几点值得我们注意：

* 虚拟机的heapstartsize和heapsize分别被设置为4M和16M。这表明在Java的进程空间中，内存堆栈只有16M的空间

parseRuntimeOption("dalvik.vm.heapstartsize", heapstartsizeOptsBuf, "-Xms", "4m");

parseRuntimeOption("dalvik.vm.heapsize", heapsizeOptsBuf, "-Xmx", "16m");

* 执行模式的配置。执行模式是在属性”dalvik.vm.execution-mode”里配置的

property\_get("dalvik.vm.execution-mode", propBuf, "");

if (strcmp(propBuf, "int:portable") == 0) {

executionMode = kEMIntPortable;

} else if (strcmp(propBuf, "int:fast") == 0) {

executionMode = kEMIntFast;

} else if (strcmp(propBuf, "int:jit") == 0) {

executionMode = kEMJitCompiler;

}

* NativeBridge的配置，在属性”ro.dalvik.vm.native.bridge”里设置。如果没有被设置，则NativeBridge被disabled

// Native bridge library. "0" means that native bridge is disabled.

property\_get("ro.dalvik.vm.native.bridge", propBuf, "");

if (propBuf[0] == '\0') {

ALOGW("ro.dalvik.vm.native.bridge is not expected to be empty");

} else if (strcmp(propBuf, "0") != 0) {

snprintf(nativeBridgeLibrary, sizeof("-XX:NativeBridge=") + PROPERTY\_VALUE\_MAX,

"-XX:NativeBridge=%s", propBuf);

addOption(nativeBridgeLibrary);

}

还有其他一些属性配置，具体请参考int AndroidRuntime::startVm(JavaVM\*\* pJavaVM, JNIEnv\*\* pEnv, bool zygote)的实现。

**向Android虚拟机注册Android native处理函数**

在创建Android虚拟机成功后，Android 运行环境也已准备就绪并运行。在Android系统服务运行的过程中，需要Java层和C/C++层的相互访问。在Android系统中，虚拟机提供JNI机制来实现Java层和C/C++层代码的相互访问。

JNI机制的实现，主要依赖于虚拟机的native method注册机制，使Java类的方法和C/C++函数关联起来。在Java类中调用相应方法的时候能直接寻找到对应的native函数并调用之。

为了相关服务能正常运行，在zygote中就需要对相关服务的JNI功能进行初始化，以使在相关服务运行的时候能够使用底层提供的功能。

此阶段就是来处理系统级的JNI的native函数注册的。用户级别的native函数的注册是通过System.loadLibrary(libraryname)中处理，此部分将在JNI处理中介绍。

下面是AndroidRuntime::start(）中的处理，直接调用AndroidRuntime::startReg(env)函数进行注册。

/\*

\* Register android functions.

\*/

if (startReg(env) < 0) {

ALOGE("Unable to register all android natives\n");

return;

}

在AndroidRuntime::startReg(env)函数中，到底是如何注册JNI native functions的呢？注册了哪些JNI native functions呢？我们来看一下该函数的具体处理：

* 设置Android线程函数。Android所有的线程都是通过函数javaCreateThreadEtc()创建，并attach到虚拟机中

/\*

\* 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);

* 注册native funcitons。每一个native functions都是通过虚拟机的RegisterNativeMethods()函数接口进行注册的

/\*

\* 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);

在zygote进程中注册的native functions。请大家参考AndroidRuntime.cpp中的 static const RegJNIRec gRegJNI[]定义，gRegJNI[]是系统级JNI native functions列表。注册native functions详细处理，请参照JNI篇介绍，这里暂时做一个大致介绍：

* 在虚拟机中查找当前java类是否已经加载，如果已加载，则直接把当前native function 与java类的方法关联。如果当前java类没有加载，则先加载该java类，然后再把native function 与java类的方法关联。
* 在此注册过程中，我们可以看到，注册中涉及的java类已全部加载到虚拟机中，并与其他进程共享，因为Android java层中，其他的进程都是fork zygote进程的。

**java层Zygote初始化处理**

在完成虚拟机的创建及JNI native functions的注册后，程序就会执行到Android Java层上。

/\*

\* Start VM. This thread becomes the main thread of the VM, and will

\* not return until the VM exits.

\*/

//classname="com.android.internal.os.ZygoteInit"

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 {

//调用com.android.internal.os.ZygoteInit的main()方法。

env->CallStaticVoidMethod(startClass, startMeth, strArray);

}

}

下面我们来具体梳理一下Zygote Java层的处理过程

**1. com.android.internal.os.ZygoteInit类的main()方法**

文件目录：\frameworks\base\core\java\com\android\internal\os\ZygoteInit.java。

在ZygoteInit类的main()方法里，主要实现如下功能：

* 保证在Zygote进程中没有创建线程
* 创建zygote socket并监听
* preload Android资源
* GC初始化并启动
* 启动System server进程
* 运行zygote进程的select loop

**保证在Zygote进程中没有创建线程**

这个处理保证在Zygote Init处理的时候没有线程被创建，也就是说，在Zygote进程中是没有线程存在的。

// Mark zygote start. This ensures that thread creation will throw

// an error.

ZygoteHooks.startZygoteNoThreadCreation();

同时，在Zygote init处理完成后，关闭保护功能。

ZygoteHooks.stopZygoteNoThreadCreation();

**创建zygote socket并监听**

zygote socket是应用程序通过Zygote进程来创建应用进程的一个通道，其主要被用在zygote select loop中来进行应用程序进程的创建处理。

//zygote socketName = "ANDROID\_SOCKET\_zygote"

String socketName = "zygote";

registerZygoteSocket(socketName);

**preload Android资源**

在zygote init中，Android的相关资源被预加载到Zygote进程空间中。这些资源在后续所有的应用程序进程中都是共享的，因为Android Java层进程都是Zygote的子进程。

//Android资源预加载的入口

preload();

下面是preload()方法的具体实现。

* preloadClasses：加载class类到虚拟机中，需要加载的类是由/system/etc/preloaded-classes文件指定。它是由相关preloaded-classes文件生成，如：   
  \frameworks\base\preloaded-classes   
  \frameworks\base\compiled-classes-phone   
  虚拟机会通过classloader把preloaded-classes文件中指定的类都加载到虚拟机中，方便应用程序的调用处理。此处会涉及到手机内存空间和手机开机性能问题，手机性能优化方面可以进一步深入研究。
* preloadResources：加载系统资源
* preloadOpenGL: 加载显示资源
* preloadSharedLibraries： 加载共享库，包含android.so、compiler\_rt.so和jnigraphics.so
* preloadTextResources：加载语言库

static void preload() {

Log.d(TAG, "begin preload");

Trace.traceBegin(Trace.TRACE\_TAG\_DALVIK, "BeginIcuCachePinning");

beginIcuCachePinning();

Trace.traceEnd(Trace.TRACE\_TAG\_DALVIK);

Trace.traceBegin(Trace.TRACE\_TAG\_DALVIK, "PreloadClasses");

preloadClasses();

Trace.traceEnd(Trace.TRACE\_TAG\_DALVIK);

Trace.traceBegin(Trace.TRACE\_TAG\_DALVIK, "PreloadResources");

preloadResources();

Trace.traceEnd(Trace.TRACE\_TAG\_DALVIK);

Trace.traceBegin(Trace.TRACE\_TAG\_DALVIK, "PreloadOpenGL");

preloadOpenGL();

Trace.traceEnd(Trace.TRACE\_TAG\_DALVIK);

preloadSharedLibraries();

preloadTextResources();

// Ask the WebViewFactory to do any initialization that must run in the zygote process,

// for memory sharing purposes.

WebViewFactory.prepareWebViewInZygote();

endIcuCachePinning();

warmUpJcaProviders();

Log.d(TAG, "end preload");

}

**GC初始化并启动**

// Do an initial gc to clean up after startup

Trace.traceBegin(Trace.TRACE\_TAG\_DALVIK, "PostZygoteInitGC");

gcAndFinalize();

Trace.traceEnd(Trace.TRACE\_TAG\_DALVIK);

**启动System server进程**

if (startSystemServer) {

startSystemServer(abiList, socketName);

}

在startSystemServer()方法里，通过Zygote.forkSystemServer() native函数调用来创建system server 进程。从创建中可以看到，system server进程是zygote进程的子进程。

/\* Hardcoded command line to start the system server \*/

String args[] = {

"--setuid=1000",

"--setgid=1000",

"--setgroups=1001,1002,1003,1004,1005,1006,1007,1008,1009,1010,1018,1021,1032,3001,3002,3003,3006,3007,3009,3010",

"--capabilities=" + capabilities + "," + capabilities,

"--nice-name=system\_server",

"--runtime-args",

"com.android.server.SystemServer",

};

ZygoteConnection.Arguments parsedArgs = null;

int pid;

try {

parsedArgs = new ZygoteConnection.Arguments(args);

ZygoteConnection.applyDebuggerSystemProperty(parsedArgs);

ZygoteConnection.applyInvokeWithSystemProperty(parsedArgs);

/\* Request to fork the system server process \*/

pid = Zygote.forkSystemServer(

parsedArgs.uid, parsedArgs.gid,

parsedArgs.gids,

parsedArgs.debugFlags,

null,

parsedArgs.permittedCapabilities,

parsedArgs.effectiveCapabilities);

} catch (IllegalArgumentException ex) {

throw new RuntimeException(ex);

}

在system\_server进程创建之后，如果运行的是子进程，即system\_server进程，则通过命令执行到com.android.server.SystemServer类的main()方法。

//运行在子进程system\_server进程

/\* For child process \*/

if (pid == 0) {

if (hasSecondZygote(abiList)) {

waitForSecondaryZygote(socketName);

}

//跳转到com.android.server.SystemServer的main()方法继续处理。

//具体实现请参考handleSystemServerProcess()方法的实现。

handleSystemServerProcess(parsedArgs);

}

**运行zygote进程的select loop**

zygote select loop是在zygote进程中运行，通过zygote socket接受相关命令来创建zygote进程的子进程。

下面是select loop 在main()方法的入口：

Log.i(TAG, "Accepting command socket connections");

runSelectLoop(abiList);

在runSelectLoop()方法中

* 监听socket，通过Os.poll函数来等待POLLIN事件的到来。
* 通过ZygoteConnection来读取socket传过来的command并创建进程。
* zygote进程在运行select loop后，zygote进程就进入无限循环，一直等待socket的command，并做处理。

//zygote进程在此进入无限循环

while (true) {

StructPollfd[] pollFds = new StructPollfd[fds.size()];

for (int i = 0; i < pollFds.length; ++i) {

pollFds[i] = new StructPollfd();

pollFds[i].fd = fds.get(i);

pollFds[i].events = (short) POLLIN;

}

try {

//zygote进程被阻塞，直至以下条件达到时退出：

//a file descriptor becomes ready;

//the call is interrupted by a signal handler; or

//the timeout expires.

Os.poll(pollFds, -1);

} catch (ErrnoException ex) {

throw new RuntimeException("poll failed", ex);

}

for (int i = pollFds.length - 1; i >= 0; --i) {

if ((pollFds[i].revents & POLLIN) == 0) {

continue;

}

if (i == 0) {

ZygoteConnection newPeer = acceptCommandPeer(abiList);

peers.add(newPeer);

fds.add(newPeer.getFileDesciptor());

} else {

//调用ZygoteConnection的runOnce()方法来fork进程。

boolean done = peers.get(i).runOnce();

if (done) {

peers.remove(i);

fds.remove(i);

}

}

}

}

至此，zygote进程启动处理完成，最后一直停留在select loop中运行。

**总结及编程相关事项**

上面是Zygote进程的启动过程。从Zygote进程的启动过程中，我们可以获取以下知识点，能在我们的编程中给予一定的帮助。

* 每个虚拟机的内存堆栈只有16M，所以在Java处理中要特别注意内存的管理和使用。
* JNI的native function注册。如果想让自己的JNI处理在虚拟机中供其他进程调用，则可以把注册函数添加到gRegJNI[]数组里。但是这样会增加系统的开机时间及内存的开销。
* 虚拟机的属性配置大部分都是依据property属性的值，所以在系统级的优化方面可以根据自己的需求来修改配置虚拟机的属性，从而生成一个适合自己项目的虚拟机。
* 在Java层的ZygoteInit类中类及资源的加载，会占用大量的开机时间和内存开销，在开机时间优化及系统内存优化方面可以做进一步研究。
* Java进程都是Zygote进程的子进程，这个是Java进程的一个基本概念，有助于程序开发中性能及内存优化等方案的讨论。
* zygote进程的启动有许多好的编程技法和算法，如参数的解析、native function的注册、JniInvocation的接口定义和实现、socket的处理、通过命令来执行特定类的方法等，值得借鉴和参考。

转载至https://blog.csdn.net/zilx/article/details/72481767