Android init进程之如何进入java世界

作者:程姚根 联系方式:chengyaogen123@163.com

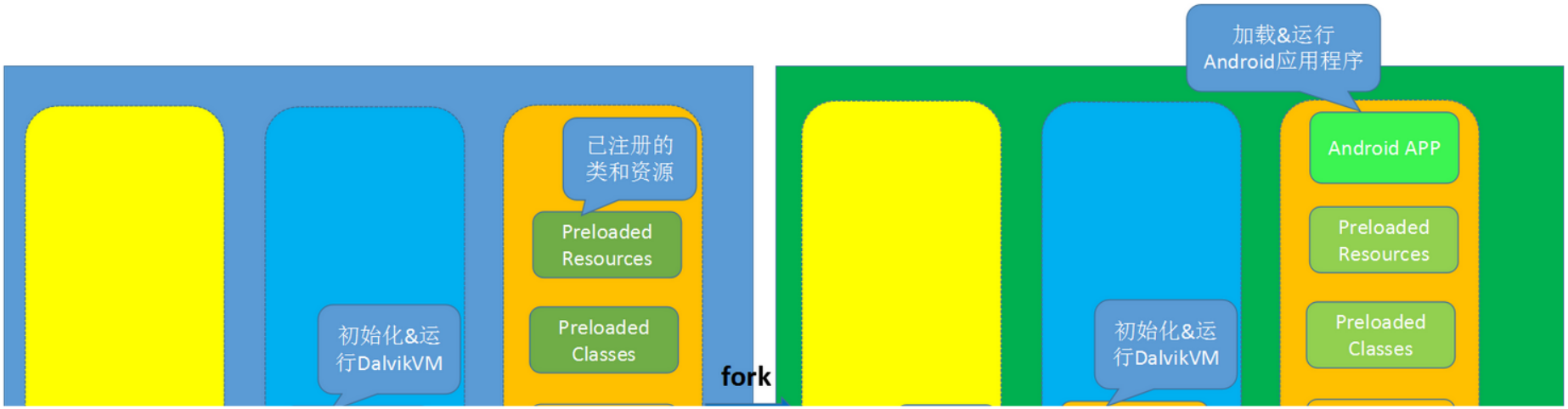
上一节我们了解了一下Android的init启动流程，我们知道在init进程启动的过程中，开启了很多服务。我们也知道Android上层的应用程序是用Java语言编写的，Java语言编写的程序需要放在Java虚拟机中运行，为了提高Android App应用程序的效率，google公司提供了经过优化之后的Dalvik虚拟机来运行上层Java语言编写的程序。

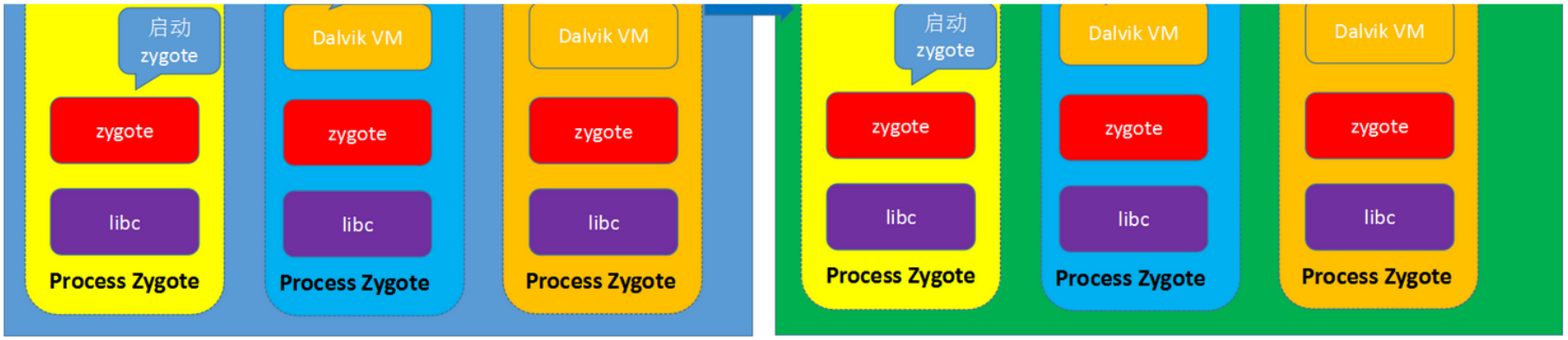
好了，抛出两个问题:

1、Android系统是如何从C/C++世界进入Java世界的呢?
2、Android系统是如何运行上层的APP应用程序的呢?

一、zygote进程介绍

回顾上一节我们知道init进程在启动过程中启动了一个叫做**zygote**服务。在Android中，zygote是整个系统创建新进程的核心装置。从字面上看，zygote是受精卵的意思，它的主要工作就是进行细胞分裂。zygote进程内部会先启动Dalvik虚拟机，继而加载一些必要的系统资源和系统类，最后进入一种监听状态。在后续的运作中，当其他系统模块(比如AMS)希望创建新进程时，只需要向zygote进程发出请求，zygote进程监听到请求后，会相应地"分裂"出新的进程，于是这个新进程在出生之时，就先天具有了自己的Dalvik虚拟机及系统资源。





zygote服务在init.zygote32.rc文件中描述如下:

```
service zygote /system/bin/app_process -Xzygote /system/bin --zygote --start-system-server
    class main
    socket zygote stream 660 root system
    onrestart write /sys/android_power/request_state wake
    onrestart write /sys/power/state on
    onrestart restart media
    onrestart restart netd
```

init进程在运行app_process时根据如下规则传递参数，app_process参数形式如下:

app_process [java-options] cmd-dir start-class-name [options]

(1)[java-options] ：传递给虚拟机的选项，必须以"-"开始

(2)cmd-dir ：所要运行的进程所在的目录

(3)start-class-name：要在虚拟机中运行的类的名称。app_process会将制定的类加载到虚拟机中，而后调用类的main()方法。

(4)[options] ：要传递给类的选项

根据参数规则，可以知道-Xzygote是指要传递给VM的选项。"-Xzygote"选项用来区分要在虚拟机中运行的类是Zygote,还是在Zygote中运行其他Android应用程序，“--zygote”表示加载com.android.internal.os.ZygoteInit类。最后一个参数"--start-system-server"作为选项传递给生成的类，用于启动运行系统服务器。

好了，了解完这些以后，我们来看看zygote的详细实现。

二、zygote服务创建过程分析

通过前面的分析，我们知道zygote对应的应用程序是/system/bin/app_process,它对应的源代码在**framework\base\cmds\app_process\app_main.cpp**文件中。

```
int main(int argc, char* const argv[])
{
```

```
    AppRuntime runtime(argv[0], computeArgBlockSize(argc, argv));
    bool zygote = false;
    bool startSystemServer = false;
    bool application = false;
    String8 niceName;
    String8 className;


    if (strcmp(arg, "--zygote") == 0) {
       zygote = true;
    } else if (strcmp(arg, "--start-system-server") == 0) {
       startSystemServer = true;
    } else if (strcmp(arg, "--application") == 0) {
        application = true;
    } else if (strncmp(arg, "--", 2) != 0) {
            className.setTo(arg);
    }

    Vector<String8> args;
    if (!className.isEmpty()) {
        // We're not in zygote mode, the only argument we need to pass
        // to RuntimeInit is the application argument.
        //
        // The Remainder of args get passed to startup class main(). Make
        // copies of them before we overwrite them with the process name.
        args.add(application ? String8("application") : String8("tool"));
        runtime.setClassNameAndArgs(className, argc - i, argv + i);
    } else {
        // We're in zygote mode.
        maybeCreateDalvikCache();

        if (startSystemServer) {
            args.add(String8("start-system-server"));
        }
    }

    if (zygote) {
        runtime.start("com.android.internal.os.ZygoteInit", args);
    } else if (className) {
        runtime.start("com.android.internal.os.RuntimeInit", args);
    } 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;
    }
}
```

可以看到在运行app_process应用程序的时候，传递了"--zygote"参数，所以这里会调用到**runtime.start("com.android.internal.os.ZygoteInit","args");**,这句话的含义是加载**com.android.internal.os.ZygoteInit**类运行。我们先来看看**runtime.start()**函数的实现。

```
/*
 * Start the Android runtime.  This involves starting the virtual machine
 * and calling the "static void main(String[] args)" method in the class
```

```
 * named by "className".
 *
 * Passes the main function two arguments, the class name and the specified
 * options string.
 */
void AndroidRuntime::start(const char* className, const Vector<String8>& options)
{
    static const String8 startSystemServer("start-system-server");
    /* start the virtual machine */
    JniInvocation jni_invocation;
    jni_invocation.Init(NULL);
    JNIEnv* env;

    if (startVm(&mJavaVM, &env) != 0) {
        return;
    }
    onVmCreated(env);

    /*
     * Register android functions.
     */
    if (startReg(env) < 0) {
        ALOGE("Unable to register all android natives\n");
        return;
    }

    /*
    /*
     * Start VM.  This thread becomes the main thread of the VM, and will
     * not return until the VM exits.
     */
    char* slashClassName = toSlashClassName(className);

    jclass startClass = env->FindClass(slashClassName);
    jmethodID startMeth = env->GetStaticMethodID(startClass, "main","([Ljava/lang/String;)V");
    env->CallStaticVoidMethod(startClass, startMeth, strArray);
}
```

这段代码主要用途如下如下:

(1)jni_invocation.Init(NULL)初始化jni接口

(2)startVM(&mJavaVM,&env)启动Dalvik虚拟机

(3)startReg(env)注册jni函数接口，方便Java世界与C/C++世界沟通

(4)FindClass(slashClassName)通过根据类名解析出来的路径查找指定的类

(5)env->GetStaticMethodID()获取指定指定类的main函数接口

(6)env->CallStaticVoidMethod()调用指定的函数接口

跋山涉水，终于构造出了Java世界(AndroidRuntime---Dalvik虚拟机),接下来我们就开始在Java世界中加载第一个类:**ZygoteInit**运行。

三、ZygoteInit类运行过程分析

好了，至此我们从苦逼的C/C++世界进入了高富帅的Java世界，下面我们来看看ZygoteInit类所做的事情，它对应的源代码在**frameworks\base\core\java\com\android\internal\os\ZygoteInit.java**中。

```
public static void main(String argv[]) {
    try {
        boolean startSystemServer = false;
        String socketName = "zygote";

        registerZygoteSocket(socketName);

        preload();

        if (startSystemServer) {
            startSystemServer(abiList, socketName);
        }

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

        closeServerSocket();
    } catch (MethodAndArgsCaller caller) {
        caller.run();
    } catch (RuntimeException ex) {
        Log.e(TAG, "Zygote died with exception", ex);
        closeServerSocket();
        throw ex;
    }
}
```

这段代码的主要用途如下:

(1)**registerZygoteSocket(socketName)**，这个函数用来绑定套接字,以便接收新Android应用程序的运行请求。Zygote使用UDS(Unix Domain Socket),为了从接收ActivityManagerService发来的新Android应用程序的运行请求。

(2)**preload()**函数实现如下:

```
static void preload() {
    Log.d(TAG, "begin preload");
    preloadClasses(); //加载
    preloadResources();
    preloadOpenGL();
    preloadSharedLibraries();
    // Ask the WebViewFactory to do any initialization that must run in the zygote process,
    // for memory sharing purposes.
    WebViewFactory.prepareWebViewInZygote();
    Log.d(TAG, "end preload");
```

}

可以看到**zygote进程在运行的过程中加载了应用程序框架中的类、平台资源(图像、XML信息、字符串等)预先加载到内存中**。新进程直接使用这些类与资源，而不需要重新加载他们，这大大加快了程序的执行速度。

加载应用程序Framework中包含的资源

在Android应用程序Framework中使用的字符串、颜色、图像文件、音频文件等都被称为资源。应用程序不能直接访问这些资源，需要通过Android开发工具自动生成的R类来访问。通过R类可访问的资源组成信息记录在XML中。Android资源大致可以分为两大类，如下:

<1>Drawable

这类资源是指北京画面、照片、图标等可在画面中绘画的资源。preloadResource会加载按钮图片、按钮组等基本主题图像。Android应用程序Framework中包含CheckBox、Button、Editbox、Call等图像文件。

<2>XML管理的资源

XML管理的资源有保存字符串的strings.xml、保存字符串数组的arrays.xml，以及保存颜色值得colors.xml等。此外，动画、布局等资源也是由XML文件管理的。

---

(3)**startSystemServer(abiList,socketName)**

```
/**
 * Prepare the arguments and fork for the system server process.
 */
private static boolean startSystemServer(String abiList, String socketName)
        throws MethodAndArgsCaller, RuntimeException {
    /* Hardcoded command line to start the system server */
    String args[] = {
        "--setuid=1000",
        "--setgid=1000",
        "--setgroups=1001,1002,1003,1004,1005,1006,1007,1008,1009,1010,1018,1032,3001,3002,3003,3006,3007",
        "--capabilities=" + capabilities + "," + capabilities,
        "--runtime-init",
        "--nice-name=system_server",
        "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);
    }
```

```
    /* For child process */
    if (pid == 0) {
        handleSystemServerProcess(parsedArgs);
    }

    return true;
}
```

创建了一个子进程，然后在子进程中加载"com.android.server.SystemServer"类运行。下面我们接着看看，它是如何运行SystemServer类运行的。

```
/**
 * Finish remaining work for the newly forked system server process.
 */
private static void handleSystemServerProcess(
        ZygoteConnection.Arguments parsedArgs)
        throws ZygoteInit.MethodAndArgsCaller {

    closeServerSocket();

    // set umask to 0077 so new files and directories will default to owner-only permissions.
    Os.umask(S_IRWXG | S_IRWXO);

    if (parsedArgs.niceName != null) {
        Process.setArgV0(parsedArgs.niceName);
    }

    /*
     * Pass the remaining arguments to SystemServer.
     */
    RuntimeInit.zygoteInit(parsedArgs.targetSdkVersion, parsedArgs.remainingArgs, cl);
    /* should never reach here */
}
```

关闭了从父进程那边继承过来的套接字文件描述符，然手调用了RuntimInit.zygoteInit()方法。这个方法在frameworks\base\core\java\com\android\internal\os\RuntimeInit.java文件中定义,我们来看看它的具体实现

```
public static final void zygoteInit(int targetSdkVersion, String[] argv, ClassLoader classLoader)
        throws ZygoteInit.MethodAndArgsCaller {

    applicationInit(targetSdkVersion, argv, classLoader);
}
```

```
private static void applicationInit(int targetSdkVersion, String[] argv, ClassLoader classLoader)
        throws ZygoteInit.MethodAndArgsCaller {

    // Remaining arguments are passed to the start class's static main
    invokeStaticMain(args.startClass, args.startArgs, classLoader);
}
```

```
private static void invokeStaticMain(String className, String[] argv, ClassLoader classLoader)
        throws ZygoteInit.MethodAndArgsCaller {
    Class<?> cl;

    cl = Class.forName(className, true, classLoader);
    m = cl.getMethod("main", new Class[] { String[].class });
    /*
     * This throw gets caught in ZygoteInit.main(), which responds
```

```
     * by invoking the exception's run() method. This arrangement
     * clears up all the stack frames that were required in setting
     * up the process.
     */
    throw new ZygoteInit.MethodAndArgsCaller(m, argv);
}
```

通过Class.forName加载SystemServer类，然后通过cl.getMethod()方法获取SystemServer类的main方法。在这里我们并没有看到直接调用这个"main"方法，而是抛出了一个一个异常。通过注释我们可以知道，这个异常最终会被ZygtoeInit.main()函数捕获。

```
  public static void main(String argv[]) {
      try {
          boolean startSystemServer = false;
          String socketName = "zygote";

          registerZygoteSocket(socketName);

          preload();

          if (startSystemServer) {
              startSystemServer(abiList, socketName);
          }

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

          closeServerSocket();
      } catch (MethodAndArgsCaller caller) {
          caller.run();
      } catch (RuntimeException ex) {
          Log.e(TAG, "Zygote died with exception", ex);
          closeServerSocket();
          throw ex;
      }
  }
```

SystemServer进程最终走到这里

我们来看看caller.run()方法的实现。

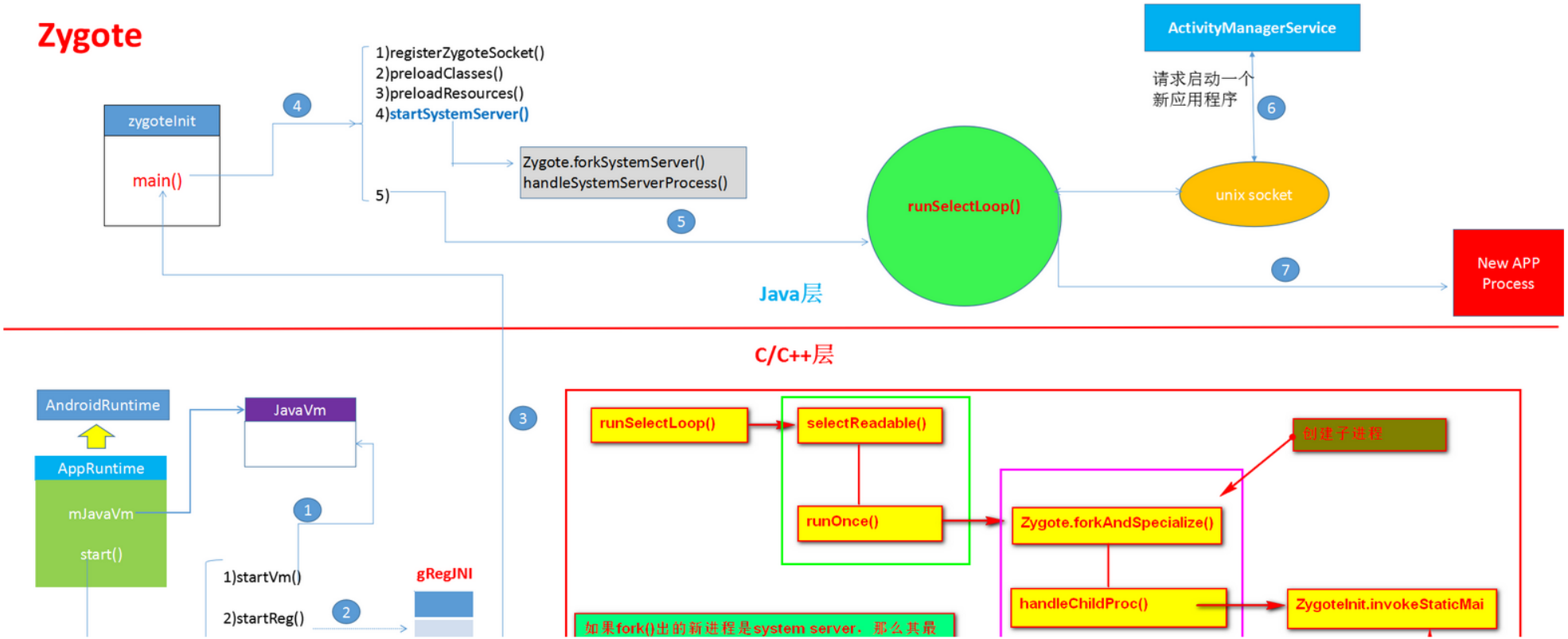
```
public void run() {
    try {
        mMethod.invoke(null, new Object[] { mArgs });
    } catch (IllegalAccessException ex) {
        throw new RuntimeException(ex);
    } catch (InvocationTargetException ex) {
        Throwable cause = ex.getCause();
        if (cause instanceof RuntimeException) {
            throw (RuntimeException) cause;
        } else if (cause instanceof Error) {
            throw (Error) cause;
        }
        throw new RuntimeException(ex);
    }
}
```

很简单啦，调用指定的方法。我们这里就是SystemServer类的main方法。嗯，至此SystemServer这个进程就创建成功了。创建子进程成功后，我们的父进程zygote从startSystemServer()函数返回后，会调用runSelectLoop()方法。

```
/**
 * Runs the zygote process's select loop. Accepts new connections as
 * they happen, and reads commands from connections one spawn-request's
 * worth at a time.
 *
 * @throws MethodAndArgsCaller in a child process when a main() should
 * be executed.
 */
private static void runSelectLoop(String abiList) throws MethodAndArgsCaller {
```

从注释中我们就可以知道，zygote进程最终一直在runSelectLoop函数工作。这个函数最终就会调用到C/C++层的select函数，探测socket文件描述附是否就绪。如果有,则说明ActivityManagerService向它发出了"启动新应用进程"的命令，zygote进程收到命令后，就会fork一个子进程，然后在子进程中抛出一个MethodAndArgsCaller异常。最终会被ZygoteInit.main()函数捕获，然后调用call.run()方法，最终调用需要运行类的main方法，这样应用程序就跑起来啦。

好了，下面我们画一幅图总结一下zygote进程做的的事情:



四、SystemServer服务分析

通过前面的分析我们知道SystemServer是zygote进程孵化出来的第一个进程，它在Android的运行环境中扮演了"神经中枢"的作用，APK应用中能够直接交互的大部分系统服务都在该进程中运行，常见的比如WindowManagerServer(WMS)、ActivityManagerSystemService(AMS)、PackageManagerServer(PMS)等，这些系统服务都是以一个线程的方式存在于SystemServer进程中。**所以SystemServer关系了整个Java世界的生死存亡,如果SystemServer进程异常退出，zygote进程知道后就会"自杀"，接着init进程重新启动zygote进程，从而再次开启Java世界。**

下面我们简单分析一下SystemServer的运行过程:

frameworks\base\services\java\com\android\server\SystemServer.java

```
/**
 * The main entry point from zygote.
 */
public static void main(String[] args) {
    new SystemServer().run();
}
```

```
private void run() {

    // Initialize native services.
    System.loadLibrary("android_servers");
    nativeInit();

    // Start services.
    try {
        startBootstrapServices();
        startCoreServices();
        startOtherServices();
    } catch (Throwable ex) {
        Slog.e("System", "******************************************");
        Slog.e("System", "************ Failure starting system services", ex);
        throw ex;
    }
}
```

从上述代码中我们可以看到SystemServer中启动了Java世界中所需要的服务,它分为核心平台服务与硬件服务

(1)核心平台服务(Core Platform Service)

一般而言，核心平台服务不会直接与Android应用程序进行交互，但它们是Android Framework运行所必须的服务，其包含的主要服务如下表所示:

核心平台服务	功能
Activity Manager Service	管理所有 Activity 的生命周期与堆栈（Stack）

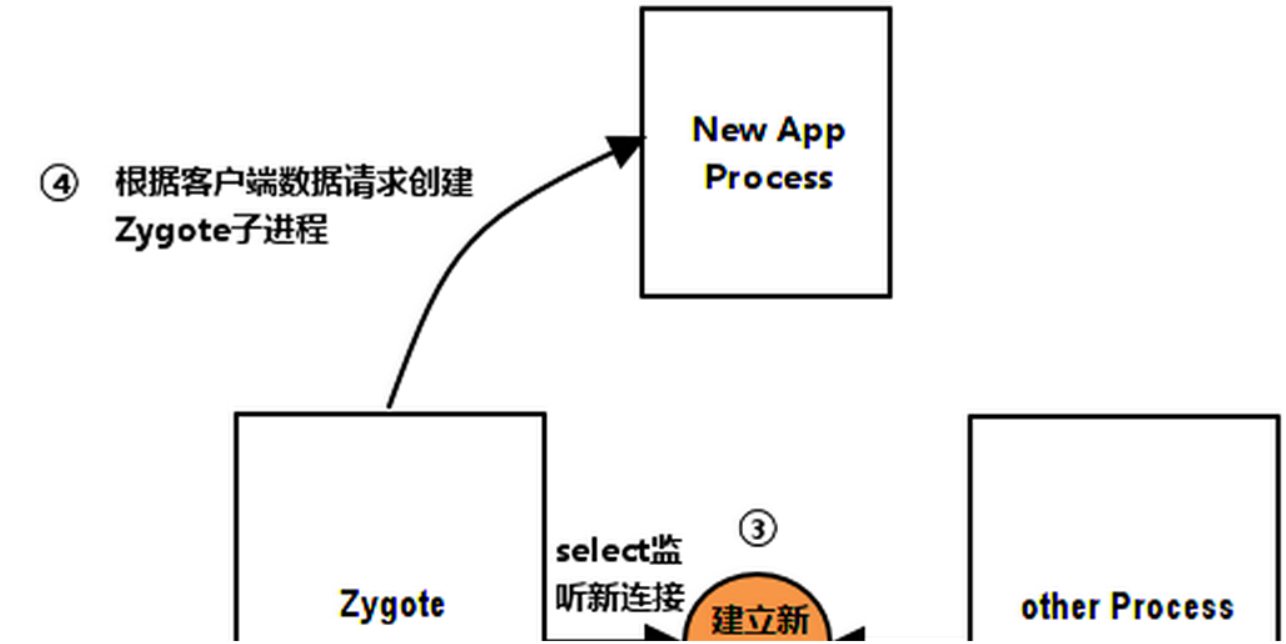
Window Manager Service	位于 Surface Flinger 之上，将要绘制到机器画面上的内容传递给 Surface Flinger
Package Manager Service	加载 apk 文件（Android 包文件）的信息，提供信息显示系统中设置了哪些包，以及加载了哪些包

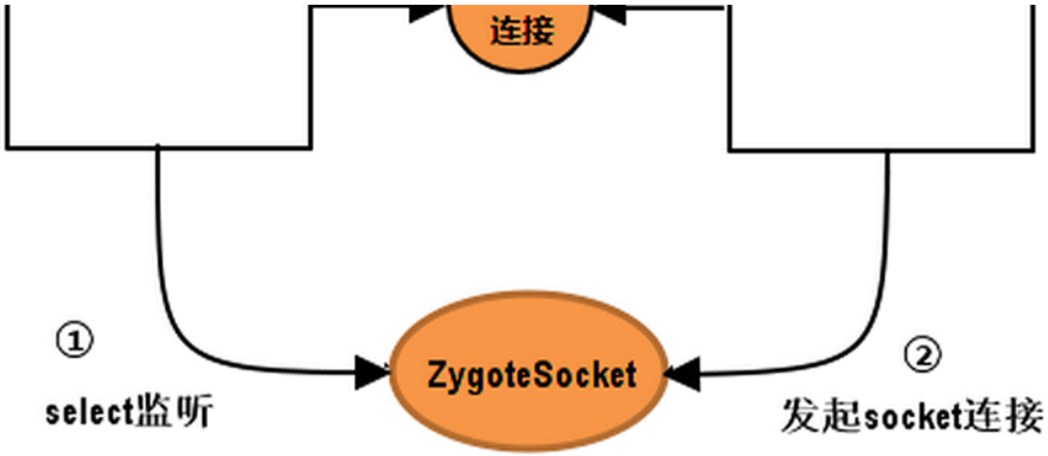
(2)硬件服务(Hardware Service)

该服务提供了一系列的API,用户控制底层硬件， 主要服务包含如下

硬件服务	功能
Alarm Manager Service	在特定时间后运行指定的应用程序，就像定时器
Connectivity Service	提供有关网络当前状态的信息
Location Service	提供终端当前的位置信息
Power Service	设备电源管理
Sensor Service	提供 Android 中各种传感器（磁力感应器、加速度传感器）的感应值
Telephony Service	提供话机状态及电话服务
Wifi Service	控制无线网络连接，如 AP 搜索、连接列表管理等

五、监听zygote socket分析





ZygoteInit的main()函数在调用完startSystemServer()之后，就会调用runSelectLoop()函数，它的代码如下:

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

```
/**
 * Runs the zygote process's select loop. Accepts new connections as
 * they happen, and reads commands from connections one spawn-request's
 * worth at a time.
 *
 * @throws MethodAndArgsCaller in a child process when a main() should
 * be executed.
 */
private static void runSelectLoop(String abiList) throws MethodAndArgsCaller {
    ArrayList<FileDescriptor> fds = new ArrayList<FileDescriptor>();
    ArrayList<ZygoteConnection> peers = new ArrayList<ZygoteConnection>();
    FileDescriptor[] fdArray = new FileDescriptor[4];

    fds.add(sServerSocket.getFileDescriptor());
    peers.add(null);

    int loopCount = GC_LOOP_COUNT;
    while (true) {
        int index;

        index = selectReadable(fdArray);

        if (index < 0) {
            throw new RuntimeException("Error in select()");
        } else if (index == 0) {
            ZygoteConnection newPeer = acceptCommandPeer(abiList);
            peers.add(newPeer);
            fds.add(newPeer.getFileDescriptor());
        } else {
            boolean done;
            done = peers.get(index).runOnce();
        }
```

```
        }
    }
```

这段代码完成的功能如下:

(1)通过selectReadable()函数探测是否有连接请求

(2)如果有则调用acceptCommandPeer()函数，提取添加请求，并且把已连接的文件描述符添加到文件描述符集合中

(3)如果有zygote socket中有数据到达，则调用peers.get(index).runOnce()函数。

runOnce()函数的实现代码如下:

frameworks\base\core\java\com\android\internal\os\ZygoteConnection.java:

```
boolean runOnce() throws ZygoteInit.MethodAndArgsCaller {

    String args[];
    Arguments parsedArgs = null;
    FileDescriptor[] descriptors;

    checkTime(startTime, "zygoteConnection.runOnce: preForkAndSpecialize");
    pid = Zygote.forkAndSpecialize(parsedArgs.uid, parsedArgs.gid, parsedArgs.gids,
                parsedArgs.debugFlags, rlimits, parsedArgs.mountExternal, parsedArgs.seInfo,
                parsedArgs.niceName, fdsToClose, parsedArgs.instructionSet,
                parsedArgs.appDataDir);

   if (pid == 0) {
            // in child
            IoUtils.closeQuietly(serverPipeFd);
            serverPipeFd = null;
            handleChildProc(parsedArgs, descriptors, childPipeFd, newStderr);

            // should never get here, the child is expected to either
            // throw ZygoteInit.MethodAndArgsCaller or exec().
            return true;
     }
}
```

这段代码的功能如下:

(1)调用Zygote.forkAndSpecialize()函数创建子进程

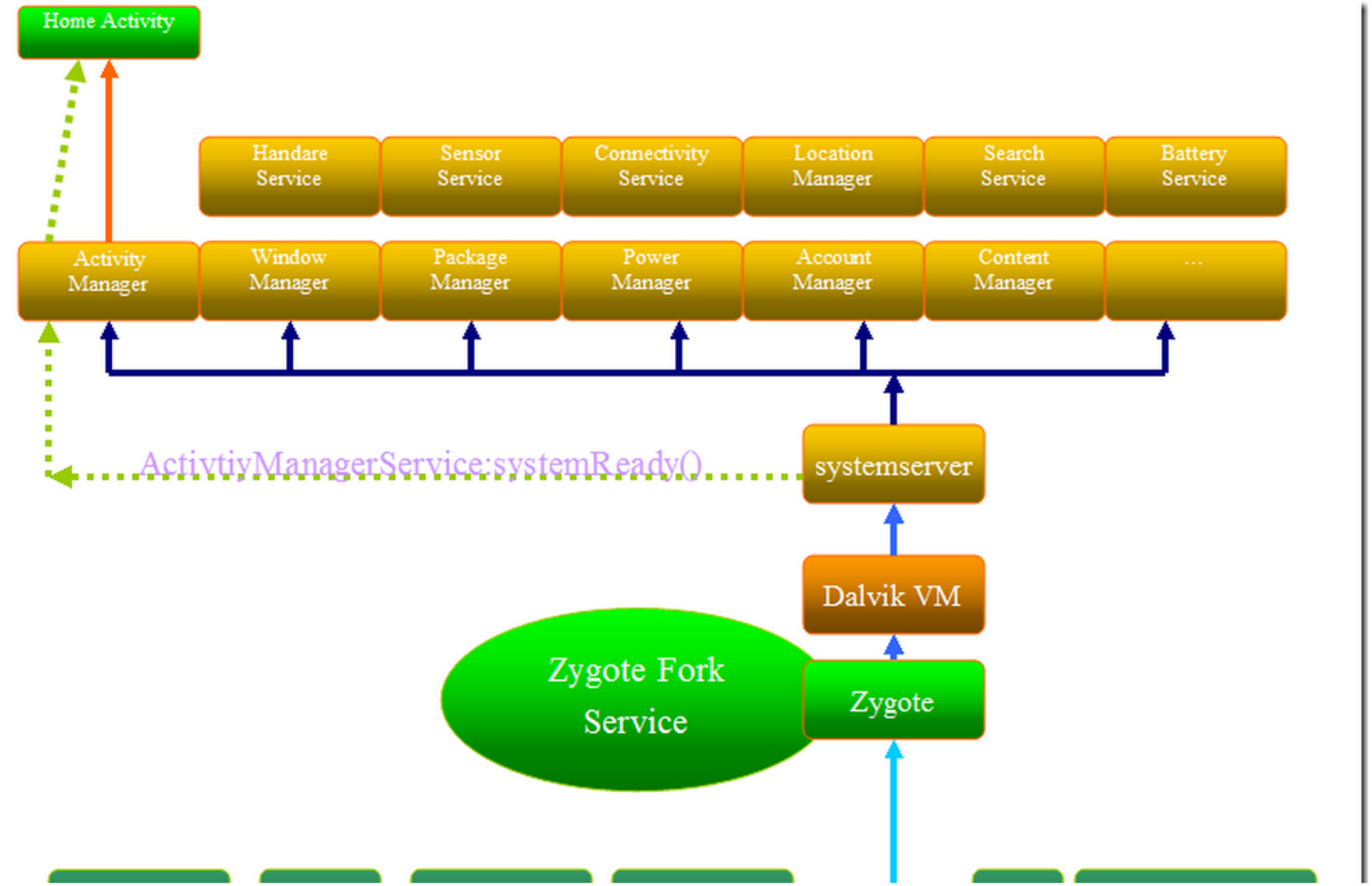
(2)调用handleChildProc()函数

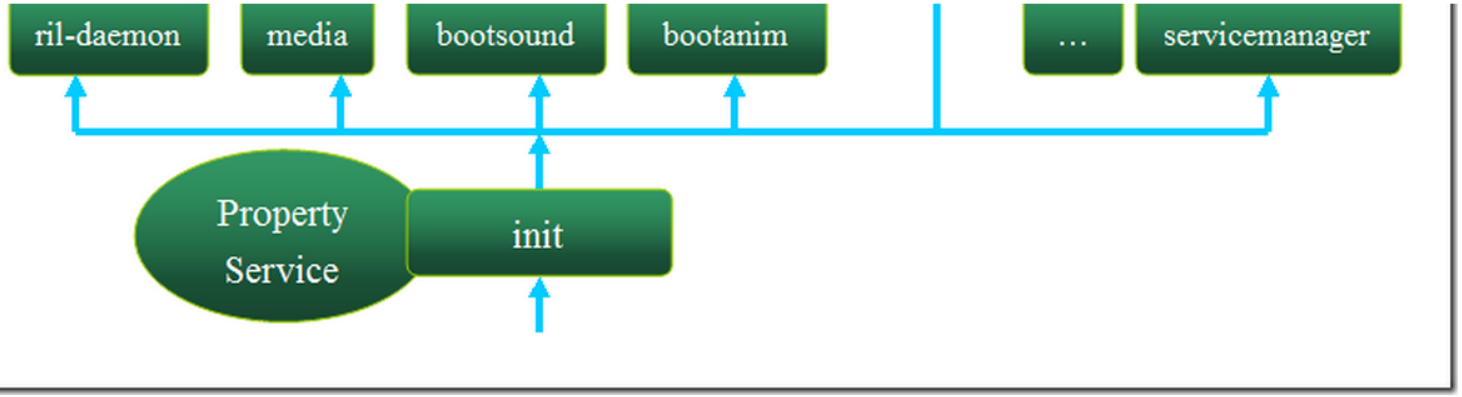
```
/**
 * Handles post-fork setup of child proc, closing sockets as appropriate,
 * reopen stdio as appropriate, and ultimately throwing MethodAndArgsCaller
 * if successful or returning if failed.
 *
 * @param parsedArgs non-null; zygote args
 * @param descriptors null-ok; new file descriptors for stdio if available.
 * @param pipeFd null-ok; pipe for communication back to Zygote.
 * @param newStderr null-ok; stream to use for stderr until stdio
 * is reopened.
 *
 * @throws ZygoteInit.MethodAndArgsCaller on success to
 * trampoline to code that invokes static main.
 */
private void handleChildProc(Arguments parsedArgs,
        FileDescriptor[] descriptors, FileDescriptor pipeFd, PrintStream newStderr)
        throws ZygoteInit.MethodAndArgsCaller {
```

```
            try {
                ZygoteInit.invokeStaticMain(cloader, className, mainArgs);
            } catch (RuntimeException ex) {
                logAndPrintError(newStderr, "Error starting.", ex);
            }
        }
    }
```

可以看到这个函数最终调用了ZygoteInit.invokeStaticMain()函数，这个函数间接抛出特殊的MethodAndArgsCaller异常，只不过此时抛出的异常携带的类名为ActivityThread。

注意:ActivityThread类在运行的时候，也标示着我们APK应用程序的运行，它就是APK应用程序的入口哦!等等，Android APP应用程序的入口不是onCreate()方法吗？呵呵！真正的入口是ActivityThread类，这个类在运行的时候会间接调用到相应类的onCreate()方法。




ril-daemon
media
bootsound
bootanim
…
servicemanager
Property Service
init