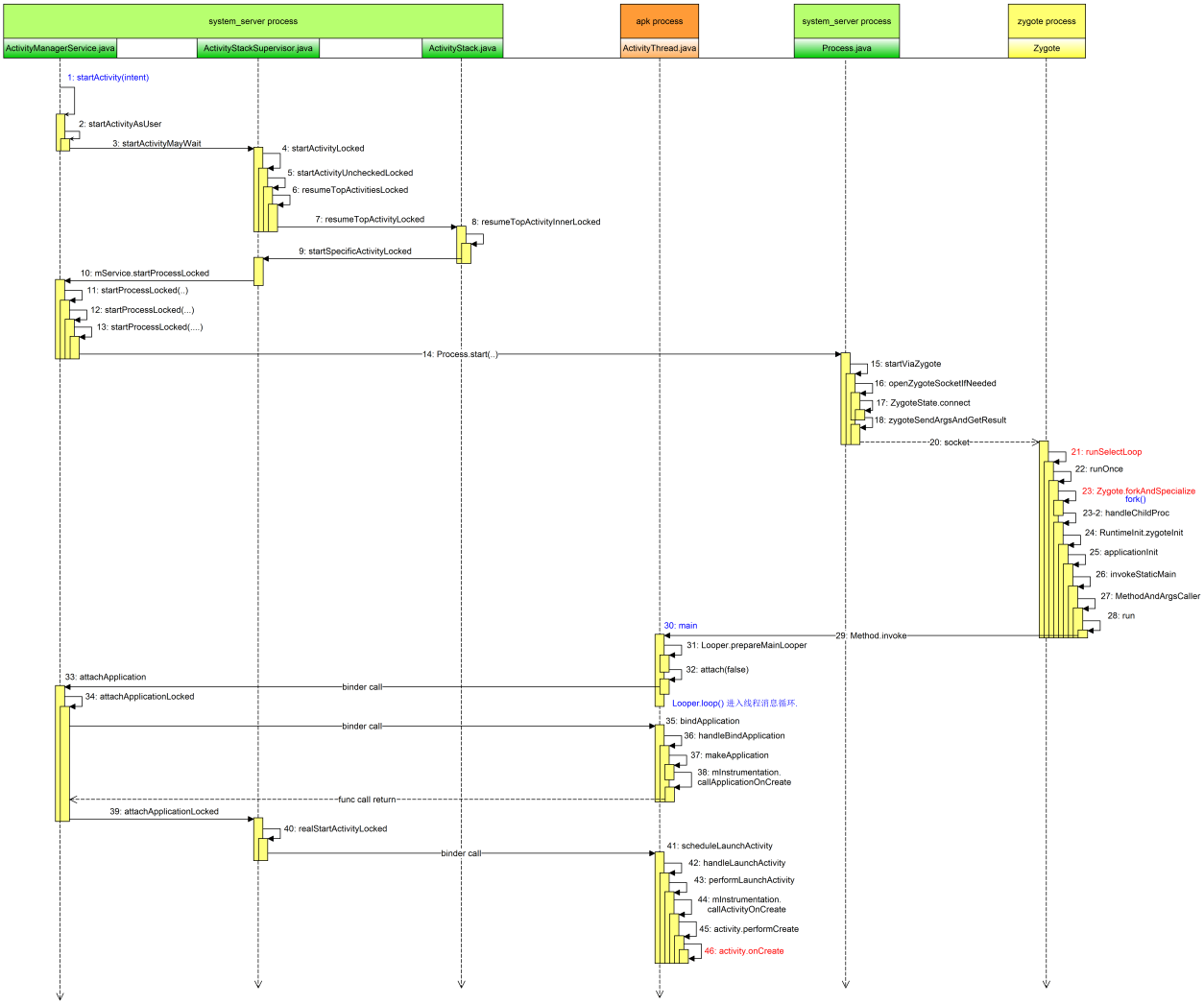
Android M 启动源码分析笔记之 - App 进程

以Home应用启动为例,一图胜千言 啊~



源码分析，Here we go!

从AMS的startActivity入口开始，时序图【1-3】：

```
@Override
    public final int startActivity(IApplicationThread caller, String callingPackage,
            Intent intent, String resolvedType, IBinder resultTo, String resultWho, int requestCode,
            int startFlags, ProfilerInfo profilerInfo, Bundle options) {
        // 很简单就是封装了一层，继续走.
        return startActivityAsUser(caller, callingPackage, intent, resolvedType, resultTo,
            resultWho, requestCode, startFlags, profilerInfo, options,
            UserHandle.getCallingUserId());
    }

    // 通过Activity栈管理者ASS切换到user app栈
    public final int startActivityAsUser(IApplicationThread caller, String callingPackage,
            Intent intent, String resolvedType, IBinder resultTo, String resultWho, int requestCode,
            int startFlags, ProfilerInfo profilerInfo, Bundle options, int userId) {
...
        // TODO: Switch to user app stacks here.
        // 这个注释已经很明显了.
        return mStackSupervisor.startActivityMayWait(caller, -1, callingPackage, intent,
                resolvedType, null, null, resultTo, resultWho, requestCode, startFlags,
                profilerInfo, null, null, options, false, userId, null, null);
    }
```

然后ASS（ActivityStackSupervisor）会依次调用这些方法： startActivityLocked ->startActivityUncheckedLocked->

resumeTopActivitiesLocked->resumeTopActivityInnerLocked->startSpecificActivityLocked. 另外注意这里面的方法很长很绕，很容易迷失在里面,需要非常耐心仔细看清楚抓主流程分析.

时序图【4-9】

重点就在下面这个函数，如果app不存在就创建一个新app进程，否则就直接启activity.

```
void startSpecificActivityLocked(ActivityRecord r,
            boolean andResume, boolean checkConfig) {
...
/* 如果该 app 已经存在了就直接调用realStartActivityLocked 启动最后
   跑Activity的onCreate流程 */
        if (app != null && app.thread != null) {
            try {
...
                realStartActivityLocked(r, app, andResume, checkConfig);
                return;
            } catch (RemoteException e) {

            }
        }

/* 否则调用AMS新创建一个app process ！ */
        mService.startProcessLocked(r.processName, r.info.applicationInfo, true, 0,
                "activity", r.intent.getComponent(), false, false, true);
    }
```

开机启动Home应用肯定是需要新创建的，所以我们按新创建这条路分析：

看到mService字眼其实就知道代码执行路径又回到了AMS中：

```
final ProcessRecord startProcessLocked(String processName,
            ApplicationInfo info, boolean knownToBeDead, int intentFlags,
            String hostingType, ComponentName hostingName, boolean allowWhileBooting,
            boolean isolated, boolean keepIfLarge) {
        return startProcessLocked(processName, info, knownToBeDead, intentFlags, hostingType,
                hostingName, allowWhileBooting, isolated, 0 /* isolatedUid */, keepIfLarge,
                null /* ABI override */, null /* entryPoint */, null /* entryPointArgs */,
                null /* crashHandler */);
    }
```

AMS中的startProcessLocked这个函数有好几个，要一步步细致的跟入分析，抓住主线忽略细节，然后会发现最终是去调用Process.start接口，传入了包括app进程启动入口等的各种信息参数，继续分析：

```
private final void startProcessLocked(ProcessRecord app, String hostingType,
            String hostingNameStr, String abiOverride, String entryPoint, String[] entryPointArgs) {

        ...
            /* 指定新进程的入口：android.app.ActivityThread，所以每一个app进程的执行都是从该进程的
android.app.ActivityThread.main 开始的.
            */
            if (entryPoint == null) entryPoint = "android.app.ActivityThread";

            /* 进入process start发请求给zygote进程要求fork新进程 */
            Process.ProcessStartResult startResult = Process.start(entryPoint,
                    app.processName, uid, uid, gids, debugFlags, mountExternal,
                    app.info.targetSdkVersion, app.info.seinfo, requiredAbi, instructionSet,
                    app.info.dataDir, entryPointArgs);
        ...

    }
```

Process.start 以及AMS等都是运行在system_server进程，通过socket机制跟父进程zygote通信.

而AcitivtyThread则是运行在app进程,所以ActivityThread跟AMS之间的通信需要借助binder call来实现.

时序图【15-20】

```
// start 不复杂，封装一层.
    public static final ProcessStartResult start(final String processClass,
                                  final String niceName,
                                  int uid, int gid, int[] gids,
                                  int debugFlags, int mountExternal,
                                  int targetSdkVersion,
                                  String seInfo,
```

```
                                String abi,
                                String instructionSet,
                                String appDataDir,
                                String[] zygoteArgs) {
        try {
            return startViaZygote(processClass, niceName, uid, gid, gids,
                    debugFlags, mountExternal, targetSdkVersion, seInfo,
                    abi, instructionSet, appDataDir, zygoteArgs);
        } catch (ZygoteStartFailedEx ex) {
...
        }
    }

    /* 将AMS传过来的新进程名、uid、gid，abi（应用程序二进制接口，CPU体系结构相关），
       进程执行入口processClass等信息通过socket传入zygote .
    */
    private static ProcessStartResult startViaZygote(final String processClass,
                                final String niceName,
                                final int uid, final int gid,
                                final int[] gids,
                                int debugFlags, int mountExternal,
                                int targetSdkVersion,
                                String seInfo,
                                String abi,
                                String instructionSet,
                                String appDataDir,
                                String[] extraArgs)
                                throws ZygoteStartFailedEx {
...

            return zygoteSendArgsAndGetResult(openZygoteSocketIfNeeded(abi), argsForZygote);
        }
    }

    // jni调用connect接口获取ZygoteState对象，这里的jni继续往下本质就是系统调用了.因为系统调用是用户程序请求内核服务的标准形式.
    private static ZygoteState openZygoteSocketIfNeeded(String abi) throws ZygoteStartFailedEx {
        if (primaryZygoteState == null || primaryZygoteState.isClosed()) {
            try {
                primaryZygoteState = ZygoteState.connect(ZYGOTE_SOCKET);
            } catch (IOException ioe) {
                throw new ZygoteStartFailedEx("Error connecting to primary zygote", ioe);
            }
        }
        ...
        return primaryZygoteState;
    }

    /* 通过系统调用发起socket请求 */
        private static ProcessStartResult zygoteSendArgsAndGetResult(
            ZygoteState zygoteState, ArrayList<String> args)
            throws ZygoteStartFailedEx {
        try {
...
            final BufferedWriter writer = zygoteState.writer;
            final DataInputStream inputStream = zygoteState.inputStream;

            writer.write(Integer.toString(args.size()));
            writer.newLine();
        ...
    }
```

system_server进程通过system call向zygote进程发起soket connect请求， zygote进程接受到请求后fork新的process，然后经过一系列新进程的初始化，通过反射机制回调从system_server传下来的子进程入口：android.app.ActivityThread.main，正式开启新子进程之旅。 时序图【21-30】

接下来看看 ActivityThread.main() 具体干了什么事情：

```
public static void main(String[] args) {
      ...
// 初始化当前用户环境参数.
        Environment.initForCurrentUser();
...
// 创建进程中唯一一个 main thread.
```

```
        Looper.prepareMainLooper();

// new 唯一一个的ActivityThread 对象，执行attach
        ActivityThread thread = new ActivityThread();

/* 这里干的事情有点多，ActivityThread通过binder call AMS的attachApplication->attachApplicationLocked
   实现ProcessRecord相关初始化准备工作，然后再通过binder调用ActivityThread
   主线程中的bindApplication函数获取绑定数据 AppBindData.
*/
        thread.attach(false);

        if (sMainThreadHandler == null) {
            sMainThreadHandler = thread.getHandler();
        }

      ...

// 线程消息轮询循环，永不退出，直到进程死掉.
        Looper.loop();

        throw new RuntimeException("Main thread loop unexpectedly exited");
    }
```

ActivityThread获取到来自AMS的bind数据后就开始进行Application的创建等初始化工作，每一个子进程有且只有一个Application对象.mInstrumentation. callApplicationOnCreate通过binder调用AMS的attachApplicationLocked和ASS的realStartActivityLocked方法最终实现真正的开始Activity生命周期启动操作，而Activity生命周期启动操作都是在ActivityThread中完成的.
时序图【38-46】

每一个app进程的第一次启动都会通过system_server向zygote发起fork请求来创建子进程，app进程的启动的真正入口是ActivityThread.main函数，每一个app进程（包括system_server)都是zygote的子进程,zygote是所有app进程的父进程，体现了zygote进程对整个Android的Java world起到了孵化作用.