Android 5.0 Camera 系统源码分析(2)：Camera 打开流程

标签： MTKandroidcamera 源码分析调用流程
2016-08-05 10:09 1400 人阅读 评论(3) 收藏 举报

1. 前言

本文将分析 **Android** 系统源码，从 frameworks 层到 hal 层，暂不涉及 app 层和 kernel 层。由于某些函数比较复杂，在贴出代码时会适当对其进行简化。

本文属于自己对源码的总结，仅仅是贯穿代码流程，不会深入分析各个细节。欢迎联系讨论，QQ：1026656828

2. app 层

从 apk 开始，简单列出各个入口函数

[cpp] view plain copy

```cpp
1.  private void initCamera()
2.  {
3.      Camera mCamera = Camera.open();
4.      Camera.Parameters mParameters = mCamera.getParameters();
5.      mParameters.setPictureFormat(PixelFormat.JPEG);
6.      mCamera.setParameters(mParameters);
7.      mCamera.setPreviewDisplay(mSurfaceHolder);
8.      mCamera.startPreview();
9.      mCamera.takePicture(null, null , mJpegCallback);
10. }
```

3. frameworks 层

这里将重点介绍 Camera.open 函数，其余函数将在后续博文分析。先来看看 Camera.open 函数在 frameworks 层的实现，代码路径为: frameworks/base/core/java/android/hardware/Camera.java

[cpp] view plain copy

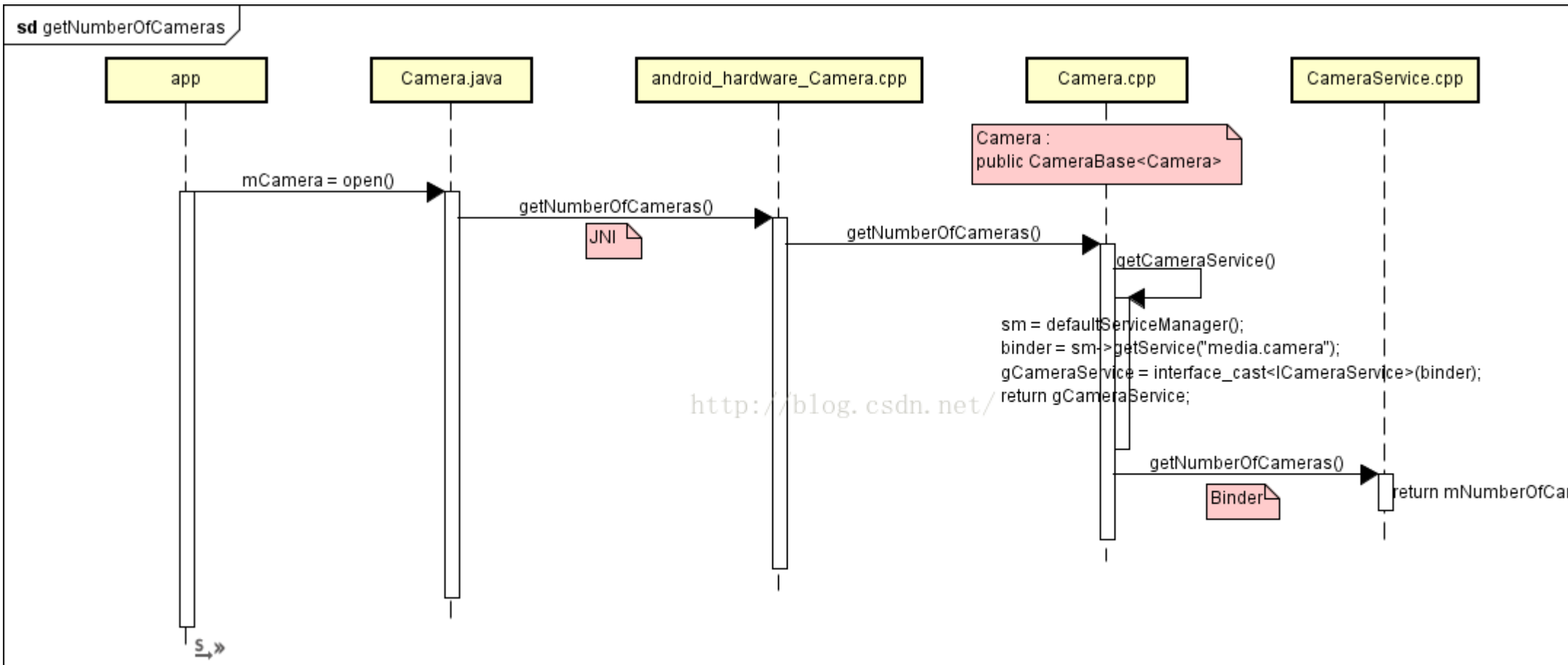
```cpp
1.  public static Camera open() {
2.      if (!isPermissionGranted()) {
3.          return null;
4.      }
5.      int numberOfCameras = getNumberOfCameras();
6.      CameraInfo cameraInfo = new CameraInfo();
7.      for (int i = 0; i < numberOfCameras; i++) {
8.          getCameraInfo(i, cameraInfo);
9.          if (cameraInfo.facing == CameraInfo.CAMERA_FACING_BACK) {
10.             return new Camera(i);
11.         }
12.     }
13.     return null;
14. }
```

第 5 行，通过 getNumberOfCameras 函数来获取 Camera 的个数。从上一篇博文 CameraService 的启动流程可以看出，这个信息保存在 CameraService 中。

第 10 行，需重点关注，构造一个 Camera 对象，并将它返回给 app 层。

3.1 getNumberOfCameras 函数分析

getNumberOfCameras 函数进入到 CameraService 获取 Camera 个数的流程如下:



Camera.**Java** 调用的 getNumberOfCameras 函数是一个 JNI 接口，对应的函数是 android_hardware_Camera.cpp 里的

android_hardware_Camera_getNumberOfCameras 函数

[cpp] view plain copy

```cpp
static jint android_hardware_Camera_getNumberOfCameras(JNIEnv *env, jobject thiz)
{
    return Camera::getNumberOfCameras();
}
```

这里只是简单调用了 Camera.cpp 的 getNumberOfCameras 函数，Camera 继承了 CameraBase，该函数由它实现

[cpp] view plain copy

```cpp
template <typename TCam, typename TCamTraits>
int CameraBase<TCam, TCamTraits>::getNumberOfCameras() {
    const sp<ICameraService> cs = getCameraService();
    return cs->getNumberOfCameras();
}
```

第 3 行, getCameraService 函数用来获取 ICameraService 的 Bp 端，代码实现如下

[cpp] view plain copy

```cpp
const char* kCameraServiceName = "media.camera";

template <typename TCam, typename TCamTraits>
const sp<ICameraService>& CameraBase<TCam, TCamTraits>::getCameraService()
{
    if (gCameraService.get() == 0) {
        sp<IServiceManager> sm = defaultServiceManager();
        sp<IBinder> binder;
        binder = sm->getService(String16(kCameraServiceName));
        gCameraService = interface_cast<ICameraService>(binder);
    }
    return gCameraService;
}
```

Android 的 Binder 通讯机制

第 1 行, 获取的 ServiceName 为"media.camera"，结合上一篇博文 CameraService 的启动流程可以看出 Bn 端的实现在 CameraService.cpp

回到之前的 getNumberOfCameras 函数，在获取到 ICameraService 的 Bp 端后，就可以开始和 Bn 端通讯了。在第 4 行，当调用 cs->getNumberOfCameras 函数时，将会进入 CameraService.cpp 的 getNumberOfCameras 函数

[cpp] view plain copy

```cpp
int32_t CameraService::getNumberOfCameras() {
    return mNumberOfCameras;
}
```

代码很简单，返回上一篇博文讲到的，千辛万苦从 hal 层拿到的数据

3.2 Camera 构造函数分析

回到最开始的 Camera.open 函数，在第 10 行，将会构造一个 Camera 对象

[cpp] view plain copy

```cpp
1.  private int cameraInitVersion(int cameraId, int halVersion) {
2.      ......
3.      Looper looper;
4.      if ((looper = Looper.myLooper()) != null) {
5.          mEventHandler = new EventHandler(this, looper);
6.      } else if ((looper = Looper.getMainLooper()) != null) {
7.          mEventHandler = new EventHandler(this, looper);
8.      } else {
9.          mEventHandler = null;
10.     }
11.
12.     String packageName = ActivityThread.currentPackageName();
13.
14.     return native_setup(new WeakReference<Camera>(this), cameraId, halVersion, packageName);
15. }
16.
17. private int cameraInitNormal(int cameraId) {
18.     return cameraInitVersion(cameraId, CAMERA_HAL_API_VERSION_NORMAL_CONNECT);
19. }
20.
21. Camera(int cameraId) {
22.     int err = cameraInitNormal(cameraId);
23.     ......
24. }
```

第 14 行, native_setup 同样是个 JNI 接口，对应 android_hardware_Camera.cpp 里的 android_hardware_Camera_native_setup 函数

[cpp] view plain copy

```cpp
1.  static jint android_hardware_Camera_native_setup(JNIEnv *env, jobject thiz,
2.      jobject weak_this, jint cameraId, jint halVersion, jstring clientPackageName)
3.  {
4.      camera = Camera::connect(cameraId, clientName, Camera::USE_CALLING_UID);
5.
6.  #if 1 // defined(MTK_CAMERA_BSP_SUPPORT)
7.      sp<JNICameraContext> context = new MtkJNICameraContext(env, weak_this, clazz, camera);
8.  #else
9.      sp<JNICameraContext> context = new JNICameraContext(env, weak_this, clazz, camera);
10. #endif
11. }
```

第 4 行，调用了 Camera.cpp 的 connect 函数，同时返回一个 Camera 对象，保存在 JNICameraContext 当中

[cpp] view plain copy

```cpp
1.  sp<Camera> Camera::connect(int cameraId, const String16& clientPackageName,
2.          int clientUid)
3.  {
4.      return CameraBaseT::connect(cameraId, clientPackageName, clientUid);
5.  }
```

先来看看 Camera 和 CameraBase 的类定义

[cpp] view plain copy

```cpp
1.  /* ---------- Camera.h ---------- */
2.  template <>
3.  struct CameraTraits<Camera>
4.  {
5.      ......
6.      static TCamConnectService fnConnectService;
7.  };
8.
9.  class Camera : public CameraBase<Camera>
10. {
11.     ......
12. }
13. /* -------- CameraBase.h -------- */
14. <pre name="code" class="cpp">template <typename TCam>
15. struct CameraTraits {
16. };
17.
18. template <typename TCam, typename TCamTraits = CameraTraits<TCam>>
19. class CameraBase
```

```
20. {
21.     ......
22.     typedef CameraBase<TCam>    CameraBaseT;
23. }
```

这里使用了 C++模版，其实就是调用 CameraBase::connect 函数

[cpp] view plain copy

```
1.  CameraTraits<Camera>::TCamConnectService CameraTraits<Camera>::fnConnectService = &ICameraService::connect;
2.
3.  template <typename TCam, typename TCamTraits>
4.  sp<TCam> CameraBase<TCam, TCamTraits>::connect(int cameraId,
5.                                                 const String16& clientPackageName,
6.                                                 int clientUid)
7.  {
8.      sp<TCam> c = new TCam(cameraId);
9.      sp<TCamCallbacks> cl = c;
10.
11.     const sp<ICameraService>& cs = getCameraService();
12.     if (cs != 0) {
13.         TCamConnectService fnConnectService = TCamTraits::fnConnectService;
14.         status = (cs.get()->*fnConnectService)(cl, cameraId, clientPackageName, clientUid,
15.                                                /*out*/ c->mCamera);
16.     }
17.
18.     return c;
19. }
```

第 1 行，将 CameraTraits::fnConnectService 赋为 ICameraService::connect

第 7 行，构造一个 Camera 对象

第 10 行，获取 ICameraService 的 Bp 端

第 13 行，从上面的解释可以看出，实际就是调用 CameraService.cpp 的 connect 函数

第 17 行，将 Camera 对象返回给 JNI 层

[cpp] view plain copy

```
1.  status_t CameraService::connectHelperLocked(
2.          /*out*/
3.          sp<Client>& client,
4.          /*in*/
5.          const sp<ICameraClient>& cameraClient,
6.          int cameraId,
7.          const String16& clientPackageName,
8.          int clientUid,
9.          int callingPid,
10.         int halVersion,
11.         bool legacyMode) {
12.     ......
13.     int deviceVersion = getDeviceVersion(cameraId, &facing);
14.
15.     switch(deviceVersion) {
16.         case CAMERA_DEVICE_API_VERSION_1_0:
17.             client = new CameraClient(this, cameraClient,
18.                     clientPackageName, cameraId,
19.                     facing, callingPid, clientUid, getpid(), legacyMode);
20.             break;
21.         case CAMERA_DEVICE_API_VERSION_2_0:
22.         case CAMERA_DEVICE_API_VERSION_2_1:
23.         case CAMERA_DEVICE_API_VERSION_3_0:
24.         case CAMERA_DEVICE_API_VERSION_3_1:
25.         case CAMERA_DEVICE_API_VERSION_3_2:
26.             client = new Camera2Client(this, cameraClient,
27.                     clientPackageName, cameraId,
28.                     facing, callingPid, clientUid, getpid(), legacyMode);
29.             break;
30.     }
31.
32.     status_t status = connectFinishUnsafe(client, client->getRemote());
33.     mClient[cameraId] = client;
34. }
35.
36. status_t CameraService::connect(
```

```
37.         const sp<ICameraClient>& cameraClient,
38.         int cameraId,
39.         const String16& clientPackageName,
40.         int clientUid,
41.         /*out*/
42.         sp<ICamera>& device) {
43.     ......
44.     sp<Client> client;
45.     status = connectHelperLocked(/*out*/client,
46.                                  cameraClient,
47.                                  cameraId,
48.                                  clientPackageName,
49.                                  clientUid,
50.                                  callingPid);
51.     device = client;
52.     return OK;
53. }
```

忽略细节之后 connect 函数就只是调用 connectHelperLocked 函数

第 13 行，获取 api 版本信息，这个函数比较简单，不细说。这里的版本为 CAMERA_DEVICE_API_VERSION_1_0

第 15-30 行，根据不同的 api 版本选择构造 CameraClient 或 Camera2Client，这里是 CameraClient

第 32 行，调用 connectFinishUnsafe 函数，实现如下

[cpp] view plain copy

```
1.  status_t CameraService::connectFinishUnsafe(const sp<BasicClient>& client,
2.                                              const sp<IBinder>& remoteCallback) {
3.      status_t status = client->initialize(mModule);
4.  }
```

这里的 client 就是上一个函数的 CameraClient，mModule 就是在上一篇博文 CameraService 的启动流程里提到的 hal 层的接口

[cpp] view plain copy

```
1.  status_t CameraClient::initialize(camera_module_t *module) {
2.      mHardware = new CameraHardwareInterface(camera_device_name);
3.      res = mHardware->initialize(&module->common);
4.      mHardware->setCallbacks(notifyCallback,
5.              dataCallback,
6.              dataCallbackTimestamp,
7.              (void *)(uintptr_t)mCameraId);
8.      return OK;
9.  }
```

构造一个 CameraHardwareInterface 对象，并调用它的 initalize 函数，直接看 initalize 函数

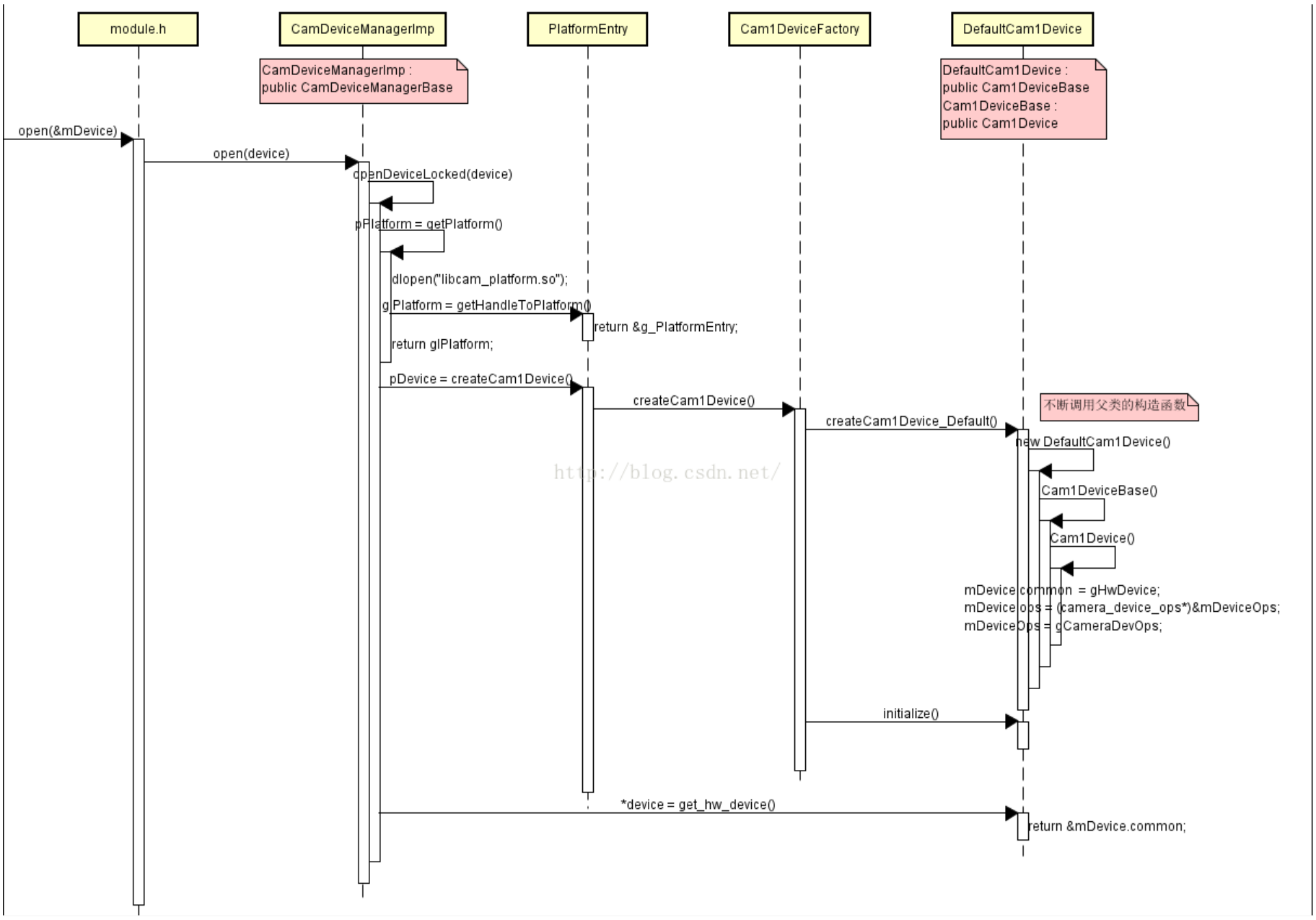
[cpp] view plain copy

```
1.  status_t initialize(hw_module_t *module)
2.  {
3.      module->methods->open(module, mName.string(), (hw_device_t **)&mDevice)
4.      initHalPreviewWindow();
5.  }
```

第 4 行，从这里进入到了 hal 层，hal 层主要对 Camera 硬件进行初始化，并将操作集保存在 mDevice 当中

4. hal 层-基于 MTK 平台

hal 层对 Camera 硬件进行初始化以及返回 Device 操作集的流程如下



4.1 open 函数分析

这里再看一次 module 的定义

[cpp] view plain copy

```cpp
static
hw_module_methods_t*
get_module_methods()
{
    static
    hw_module_methods_t
    _methods =
    {
        open: open_device
    };

    return  &_methods;
}

static
camera_module
get_camera_module()
{
    camera_module module = {
        common:{
            tag                 : HARDWARE_MODULE_TAG,
            #if (PLATFORM_SDK_VERSION >= 21)
            module_api_version  : CAMERA_MODULE_API_VERSION_2_3,
            #else
            module_api_version  : CAMERA_DEVICE_API_VERSION_1_0,
            #endif
            hal_api_version     : HARDWARE_HAL_API_VERSION,
            id                  : CAMERA_HARDWARE_MODULE_ID,
            name                : "MediaTek Camera Module",
            author              : "MediaTek",
            methods             : get_module_methods(),
            dso                 : NULL,
            reserved            : {0},
        },
        get_number_of_cameras   : get_number_of_cameras,
        get_camera_info         : get_camera_info,
```

```
        set_callbacks               : set_callbacks,
        get_vendor_tag_ops          : get_vendor_tag_ops,
        #if (PLATFORM_SDK_VERSION >= 21)
        open_legacy                 : open_legacy,
        #endif
        reserved                    : {0},
    };
    return  module;
};
```

通过 module->methods 获取到的函数为 open_device，

[cpp] view plain copy

```
static
int
open_device(hw_module_t const* module, const char* name, hw_device_t** device)
{
    return  NSCam::getCamDeviceManager()->open(device, module, name);
}
```

CamDeviceManagerImp 继承了 CamDeviceManagerBase。这里直接调用了 CamDeviceManagerBase 的 open()

[cpp] view plain copy

```
status_t
CamDeviceManagerBase::
openDeviceLocked(
    hw_device_t** device,
    hw_module_t const* module,
    int32_t const i4OpenId,
    uint32_t device_version
)
{
    // [2] get platform
    IPlatform*const pPlatform = getPlatform();
    pDevice = pPlatform->createCam1Device(s8ClientAppMode.string(), i4OpenId);
    *device = const_cast<hw_device_t*>(pDevice->get_hw_device());
}

status_t
CamDeviceManagerBase::
open(
    hw_device_t** device,
    hw_module_t const* module,
    char const* name,
    uint32_t device_version
)
{
    return  openDeviceLocked(device, module, i4OpenId, device_version);
}
```

第 11 行, getPlatform 函数用来加载 libcam_platform.so，并获取 PlatformEntry 接口

第 12 行，构造一个 Cam1Device 对象，并调用它的 init 函数

第 13 行，获取 camera device 的操作集

4.2 getPlatform 函数分析

[cpp] view plain copy

```
static PlatformEntry g_PlatformEntry;

IPlatform*
getHandleToPlatform()
{
    return &g_PlatformEntry;
}

IPlatform*
CamDeviceManagerBase::
getPlatform()
{
    char const szModulePath[] = "libcam_platform.so";
    char const szEntrySymbol[] = "getHandleToPlatform";
    void* pfnEntry = NULL;
    IPlatform* pIPlatform = NULL;

```

```
18.     mpLibPlatform = ::dlopen(szModulePath, RTLD_NOW);
19.     pfnEntry = ::dlsym(mpLibPlatform, szEntrySymbol);
20.     pIPlatform = reinterpret_cast<IPlatform*(*)()>(pfnEntry)();
21.     gIPlatform = pIPlatform;
22.
23.     return gIPlatform;
24. }
```

第 18 行，加载 libcam_platform.so

第 19 20 行，获取 getHandleToPlatform 函数入口，并调用，最后返回 PlatformEntry 接口

4.3 pPlatform->createCam1Device 函数分析

[cpp] view plain copy

```
1.  NSCam::Cam1Device*
2.  createCam1Device(
3.      String8 const   s8ClientAppMode,
4.      int32_t const   i4OpenId
5.  )
6.  {
7.      NSCam::Cam1Device* pdev = NULL;
8.
9.      String8 const s8LibPath = String8::format("libcam.device1.so");
10.     void *handle = ::dlopen(s8LibPath.string(), RTLD_GLOBAL);
11.
12.     String8 const s8CamDeviceInstFactory = String8::format("createCam1Device_Default");
13.     void* pCreateInstance = ::dlsym(handle, s8CamDeviceInstFactory.string());
14.     pdev = reinterpret_cast<NSCam::Cam1Device* (*)(String8 const&, int32_t const)>
15.                     (pCreateInstance)(s8ClientAppMode, i4OpenId);
16.
17.     pdev->initialize();
18. }
19.
20. ICamDevice*
21. PlatformEntry::
22. createCam1Device(
23.     char const*         szClientAppMode,
24.     int32_t const       i4OpenId
25. )
26. {
27.     return  ::createCam1Device(String8(szClientAppMode), i4OpenId);
28. }
```

pPlatform->createCam1Device 函数调用的是 Cam1DeviceFactory.cpp 里的 createCam1Device 函数

第 10 行，加载 libcam.device1.so

第 12-15 行，获取 createCam1Device_Default 函数入口并调用

第 17 行, Cam1Device 初始化

先来看 createCam1Device_Default 函数，以及类的继承关系

[cpp] view plain copy

```
1.  class Cam1DeviceBase : public Cam1Device
2.  {
3.  }
4.
5.  class DefaultCam1Device : public Cam1DeviceBase
6.  {
7.  }
8.
9.  NSCam::Cam1Device*
10. createCam1Device_Default(
11.     String8 const&          rDevName,
12.     int32_t const           i4OpenId
13. )
14. {
15.     return new DefaultCam1Device(rDevName, i4OpenId);
16. }
```

接着看 DefaultCam1Device 的构造函数

[cpp] view plain copy

```
1.  Cam1Device::
2.  Cam1Device()
```

```
3.  {
4.      ::memset(&mDevice, 0, sizeof(mDevice));
5.      mDevice.priv    = this;
6.      mDevice.common  = gHwDevice;
7.      mDevice.ops     = (camera_device_ops*)&mDeviceOps;
8.      mDeviceOps      = gCameraDevOps;
9.  }
10.
11. Cam1DeviceBase::
12. Cam1DeviceBase(
13.     String8 const&          rDevName,
14.     int32_t const           i4OpenId
15. )
16.     : Cam1Device()
17.     , mDevName(rDevName)
18.     , mi4OpenId(i4OpenId)
19. {
20.     MY_LOGD("");
21. }
22.
23. DefaultCam1Device::
24. DefaultCam1Device(
25.     String8 const&          rDevName,
26.     int32_t const           i4OpenId
27. )
28.     : Cam1DeviceBase(rDevName, i4OpenId)
29. {
30. }
```

删除了一些暂不关注的代码，DefaultCam1Device 的构造函数会不断调用父类的构造函数，需要关注的是它的父类 Cam1Device 的构造函数。其中的 gCameraDevOps 结构体很重要，是 Camera Device 的操作集，预览、拍照、录像都是通过它来操作，来看下它的定义

[cpp] view plain copy

```
1.  static mtk_camera_device_ops const
2.  gCameraDevOps =
3.  {
4.      #define OPS(name) name: camera_##name
5.
6.      {
7.          OPS(set_preview_window),
8.          OPS(set_callbacks),
9.          OPS(enable_msg_type),
10.         OPS(disable_msg_type),
11.         OPS(msg_type_enabled),
12.         OPS(start_preview),
13.         OPS(stop_preview),
14.         OPS(preview_enabled),
15.         OPS(store_meta_data_in_buffers),
16.         OPS(start_recording),
17.         OPS(stop_recording),
18.         OPS(recording_enabled),
19.         OPS(release_recording_frame),
20.         OPS(auto_focus),
21.         OPS(cancel_auto_focus),
22.         OPS(take_picture),
23.         OPS(cancel_picture),
24.         OPS(set_parameters),
25.         OPS(get_parameters),
26.         OPS(put_parameters),
27.         OPS(send_command),
28.         OPS(release),
29.         OPS(dump)
30.     },
31.     OPS(mtk_set_callbacks),
32.
33.     #undef  OPS
34. };
```

回到 createCam1Device 函数，最后调用了 pdev->initialize 函数，这个函数过程比较复杂，在它的父类 Cam1DeviceBase 中实现

[cpp] view plain copy

```
1.  bool
2.  DefaultCam1Device::
```

```
onInit()
{
    //  (1) power on sensor
    if( pthread_create(&mThreadHandle, NULL, doThreadInit, this) != 0 )
    {
        goto lbExit;
    }

    //  (2) Open 3A
    mpHal3a = NS3A::IHal3A::createInstance(
            NS3A::IHal3A::E_Camera_1,
            getOpenId(),
            LOG_TAG);

    //  (3) Init Base.
    if  ( ! Cam1DeviceBase::onInit() )
    {
        goto lbExit;
    }
}

status_t
Cam1DeviceBase::
initialize()
{
    onInit();
    return  OK;
}
```

initialize 函数只是简单的回调了 onInit 函数，如注释所示，主要做了 3 件事情。其中(2)和(3)主要是初始化 3A 和 CamClient，这两个这里暂时不会关注，所以暂时不进行分析。重点关注(1)，也就是 doThreadInit 函数

[cpp] view plain copy

```
bool
DefaultCam1Device::
powerOnSensor()
{
    IHalSensorList* pHalSensorList = IHalSensorList::get();
    mpHalSensor = pHalSensorList->createSensor(USER_NAME, getOpenId());
    sensorIdx = getOpenId();
    if( !mpHalSensor->powerOn(USER_NAME, 1, &sensorIdx) )
    {
        MY_LOGE("sensor power on failed: %d", sensorIdx);
        goto lbExit;
    }
    ......
}

void*
DefaultCam1Device::
doThreadInit(void* arg)
{
    DefaultCam1Device* pSelf = reinterpret_cast<DefaultCam1Device*>(arg);
    pSelf->mRet = pSelf->powerOnSensor();
    pthread_exit(NULL);
    return NULL;
}
```

doThreadInit 函数只是回调自身的了 powerOnSensor 函数，而 powerOnSensor 函数先调用 pHalSensorList->createSensor 函数创建一个 HalSensor 实例，然后再调用它的 PowerOn 函数来开始相关的硬件操作，来看 powerOn 的实现

[cpp] view plain copy

```
MBOOL
HalSensor::
powerOn(
    char const* szCallerName,
    MUINT const uCountOfIndex,
    MUINT const*pArrayOfIndex
)
{
    ......
    ret = mpSeninfDrv->init();
    ret = mpSensorDrv->init(sensorDev);
```

```
12.     ret = setTgPhase(sensorDev, pcEn);
13.     ret = setSensorIODrivingCurrent(sensorDev);
14.     ret = mpSensorDrv->open(sensorDev);
15.     ......
16. }
```

powerOn 函数比较长，这里暂时只关注 SensorDrv 的 init 和 open 函数

[cpp] view plain copy

```
1.  MINT32
2.  ImgSensorDrv::init(MINT32 sensorIdx)
3.  {
4.      m_fdSensor = ::open("dev/kd_camera_hw", O_RDWR);
5.
6.      //set driver
7.      ret = ioctl(m_fdSensor, KDIMGSENSORIOC_X_SET_DRIVER,sensorDrvInit);
8.
9.      //init resolution
10.     pSensorResInfo[0] = &m_SenosrResInfo[0];
11.     pSensorResInfo[1] = &m_SenosrResInfo[1];
12.     ret = getResolution(pSensorResInfo);
13.
14.     if(SENSOR_MAIN & sensorIdx ) {
15.         sensorDevId = SENSOR_MAIN;
16.
17.         FeatureParaLen = sizeof(MUINTPTR);
18.         ret = featureControl((CAMERA_DUAL_CAMERA_SENSOR_ENUM)sensorDevId, SENSOR_FEATURE_GET_PIXEL_CLOCK_FREQ,  (MUINT8*)&FeaturePara32,(MUINT32*)&F
    eatureParaLen);
19.
20.         FeatureParaLen = sizeof(pFeaturePara16);
21.         ret = featureControl((CAMERA_DUAL_CAMERA_SENSOR_ENUM)sensorDevId, SENSOR_FEATURE_GET_PERIOD,  (MUINT8*)pFeaturePara16,(MUINT32*)&FeaturePara
    Len);
22.     }
23.     ......
24. }
25.
26. MINT32
27. ImgSensorDrv::open(MINT32 sensorIdx)
28. {
29.     MINT32 err = SENSOR_NO_ERROR;
30.
31.     ......
32.     err = ioctl(m_fdSensor, KDIMGSENSORIOC_X_SET_CURRENT_SENSOR, &sensorIdx);
33.     err = ioctl(m_fdSensor, KDIMGSENSORIOC_T_OPEN);
34.
35.     return err;
36. }
```

这两个函数逻辑比较简单，就是通过 ioctl 进入到 kernel 层来对 sensor 硬件进行初始化和获取硬件相关的信息。kernel 层的代码暂不分析

4.4 get_hw_device 函数分析

回到 4.1 的 openDeviceLocked 函数，最后调用了 pDevice->get_hw_device 函数，并将它的返回值赋给*device

[cpp] view plain copy

```
1.  class Cam1Device : public ICamDevice
2.  {
3.      virtual hw_device_t const*      get_hw_device() const { return &mDevice.common; }
4.  }
```

这个函数很简单，就是获取 4.3 里面提到的 mDevice，这个 mDevice 最终将被保存在 frameworks 层的 CameraHardwareInterface.h 的 mDevice 变量当中，以便日后访问

5. 总结

Camera 打开流程的重点工作在 4.3 和 4.4 章节，也就是对 Camera 硬件进行初始化和将 gCameraDevOps 操作集返回给 frameworks 层。