火龙果•整理
uml.org.cn

学习小结

Android 驱动框架学习

Sensor 部分代码分析

目录

第一章　Android 体系结构分析 ……2
1.1　Android 源码目录简介 ……2
1.1.1 核心工程部分……2
1.1.2 扩展工程部分……3
1.1.3 包部分……5
1.2　Android 分层结构 ……5
1.2.1 Linux Kernel 层……6
1.2.2 Libraries&Android Runtime 层……6
1.2.3 Framework 层……6
1.2.4 Applications 层……7
第二章　Android 系统启动过程 ……8
2.1　INIT 进程的启动 ……8
2.1.1 rc 文件解析……9
2.1.2 服务启动……9
2.2 ServiceManager 的启动 ……11
2.3 Zygote 进程的启动 ……11
2.3.1 在.rc 脚本文件中 zygote 的描述……11
2.3.2 ZygoteInit……12
2.4 SystemServer 的启动 ……13
2.5 HOME 界面的启动 ……16
第三章 Sensor 体系结构 ……18
3.1 Android Sensor 架构 ……18
3.2 Sensor HAL 层接口 ……19
3.3 JNI/HAL 编程实现 ……21
3.3.1 JNI 层代码简析……21
3.3.2 HAL 层代码简析……22
3.4 Sensor 驱动代码简析 ……23
第四章　陀螺仪装配方位说明 ……27
4.1 方位矩阵 ……27
4.2 方位确定步骤 ……27
第五章　工具命令学习 ……29
5.1 ADB 常用命令 ……29
5.2 LINUX 下 SVN 常用命令 ……31

第一章　Android 体系结构分析

1.1　Android 源码目录简介

Android代码的工程分为如下3个部分：
核心工程（Core Project）：建立Android 系统的基础，在根目录的各个文件夹中。
扩展工程（External Project）：使用其他开源项目扩展的功能，在external 文件夹中。
包工程（Package Project）：提供Android 的应用程序和服务，在package 文件夹中。

1.1.1 核心工程部分

Android 的核心工程包含了对 Android 系统基本运行的支持，以及 Android 系统的编译系统，工程的内容如下表：

Android 的核心工程

工程名称	工程描述
bionic	[Build 系统] C 运行时支持：libc、libm、libdl、动态 linker
bootloader/legacy	Bootloader 参考代码（内核加载器，在内核运行之前运行）
build	[Build 系统] Build 系统
dalvik	Dalvik 虚拟机
delelopment	高层的开发和调试工具
framework/base	Android 核心的框架库
framework/policies/base	框架配置策略
hardware/libhardware	硬件抽象层库
hardware/ril	无线接口层(Radio Interface Layer)
kernel	Linux 内核
prebuilt	[预编译内核] 对 Linux 和 Mac OS 编译的二进制支持
system/core	最小化可启动的环境
system/extras	底层调试和检查工具

除了上表列出的工程外，还有其他的工程，但大都是 Android 系统第 2 个层次的源代码。

1.1.2 扩展工程部分

Android 的扩展工程包含在 external 文件夹中，是一些经过修改后适应 Android 系统的开源工程。有一些工程在主机上运行，也有些在目标机上运行。

external 目录结构及简介如下：

```
external 目录
.
|-- aes                    （AES 加密）
|-- apache-http            （网页服务器）
|-- astl                   （ASTL (Android STL) is a slimmed-down version of the regular C++ STL.）
|-- bison                  （自动生成语法分析器，将无关文法转换成 C、C++）
|-- blktrace               （blktrace is a block layer IO tracing mechanism）
|-- bluetooth              （蓝牙相关、协议栈）
|-- bsdiff                 （diff 工具）
|-- bzip2                  （压缩工具）
|-- clearsilver            （html 模板系统）
|-- dbus                   （低延时、低开销、高可用性的 IPC 机制）
|-- dhcpcd                 （DHCP 服务）
|-- dosfstools             （DOS 文件系统工具）
|-- dropbear               （SSH2 的 server）
|-- e2fsprogs              （EXT2 文件系统工具）
|-- elfcopy                （复制 ELF 的工具）
|-- elfutils               （ELF 工具）
|-- embunit                （Embedded Unit Project）
|-- emma                   （java 代码覆盖率统计工具）
|-- esd                    （Enlightened Sound Daemon，将多种音频流混合在一个设备上播放）
|-- expat                  （Expat is a stream-oriented XML parser.）
|-- fdlibm                 （FDLIBM (Freely Distributable LIBM)）
|-- freetype               （字体）
|-- fsck_msdos             （dos 文件系统检查工具）
|-- gdata                  （google 的无线数据相关）
|-- genext2fs              （genext2fs generates an ext2 filesystem as a normal (non-root) user）
|-- giflib                 （gif 库）
|-- googleclient           （google 用户库）
|-- grub                   （This is GNU GRUB, the GRand Unified Bootloader.）
|-- gtest                  （Google C++ Testing Framework）
|-- icu4c                  （ICU(International Component for Unicode)在 C/C++下的版本）
|-- ipsec-tools            （This package provides a way to use the native IPsec functionality ）
|-- iptables               （防火墙）

|-- jdiff                  （generate a report describing the difference between two public Java APIs.）
|-- jhead                  （jpeg 头部信息工具）
```

```
|-- jpeg                    （jpeg 库）
|-- junit                   （JUnit 是一个 Java 语言的单元测试框架）
|-- kernel-headers          （内核的一些头文件）
|-- libffi                  （libffi is a foreign function interface library.）
|-- libpcap                 （网络数据包捕获函数）
|-- libpng                  （png 库）
|-- libxml2                 （xml 解析库）
|-- mtpd                    （一个命令）
|-- netcat                  （simple Unix utility which reads and writes dataacross network connections）
|-- netperf                 （网络性能测量工具）
|-- neven                   （看代码和 JNI 相关）
|-- opencore                （多媒体框架）
|-- openssl                 （SSL 加密相关）
|-- openvpn                 （VPN 开源库）
|-- oprofile                （OProfile 是 Linux 内核支持的一种性能分析机制。）
|-- ping                    （ping 命令）
|-- ppp                     （pppd 拨号命令，好像还没有 chat）
|-- proguard                （Java class file shrinker, optimizer, obfuscator, and preverifier）
|-- protobuf                （a flexible, efficient, automated mechanism for serializing structured data）
|-- qemu                    （arm 模拟器）
|-- safe-iop                （functions for performing safe integer operations  ）
|-- skia                    （skia 图形引擎）
|-- sonivox                 （sole MIDI solution for Google Android Mobile Phone Platform）
|-- speex                   （Speex 编/解码 API 的使用(libspeex)）
|-- sqlite                  （数据库）
|-- srec                    （Nuance  公司提供的开源连续非特定人语音识别）
|-- strace                  （trace 工具）
|-- svox                    （Embedded Text-to-Speech）
|-- tagsoup                 （TagSoup 是一个 Java 开发符合 SAX 的 HTML 解析器）
|-- tcpdump                 （抓 TCP 包的软件）
|-- tesseract               （Tesseract Open Source OCR Engine.）
|-- tinyxml                 （TinyXml is a simple, small, C++ XML parser）
|-- tremor                  （I stream and file decoder provides an embeddable,integer-only library）
|-- webkit                  （浏览器核心）
|-- wpa_supplicant          （无线网卡管理）
|-- xmlwriter               （XML  编辑工具）
|-- yaffs2                  （yaffs 文件系统）
`-- zlib                    （a general purpose data compression library）
```

1.1.3 包部分

Android 中的 Java 程序包是 Android 系统架构第 4 层的内容，主要包括应用程序(Application)和内容提供器(Content Providers)两个部分，还有一个目录 inputmethods 是输入法的部分。

应用程序（Application）在 package/apps 目录中，主要包括：

AlarmClock，Browser，Calculator，Calendar，Camera，Contacts，E-mail，GoogleSearch HTML Viewer，IM，Launcher，Mms，Music，PackageInstaller，Phone Settings，SoundRecorder，Stk，Sync，Updater，VoiceDialer。

内容提供者（Content Providers）在 package/providers 目录中，主要包括：

CalendarProvider，ContactsProvider，DownloadProvider，DrmProvider GoogleContactsProvider，GoogleSubsribedFeedsProvidercImProvider，MediaProvider SettingsProvider，SubscribedFeedsProvider，TelephonyProvider。

1.2 Android 分层结构

Android 的系统架构采用了分层架构的思想，如图 1 所示。从底层到上层共包括四层，分别是 Linux Kernel 层、Libraries&Android Runtime 层、Framework 层和 Applications 层。

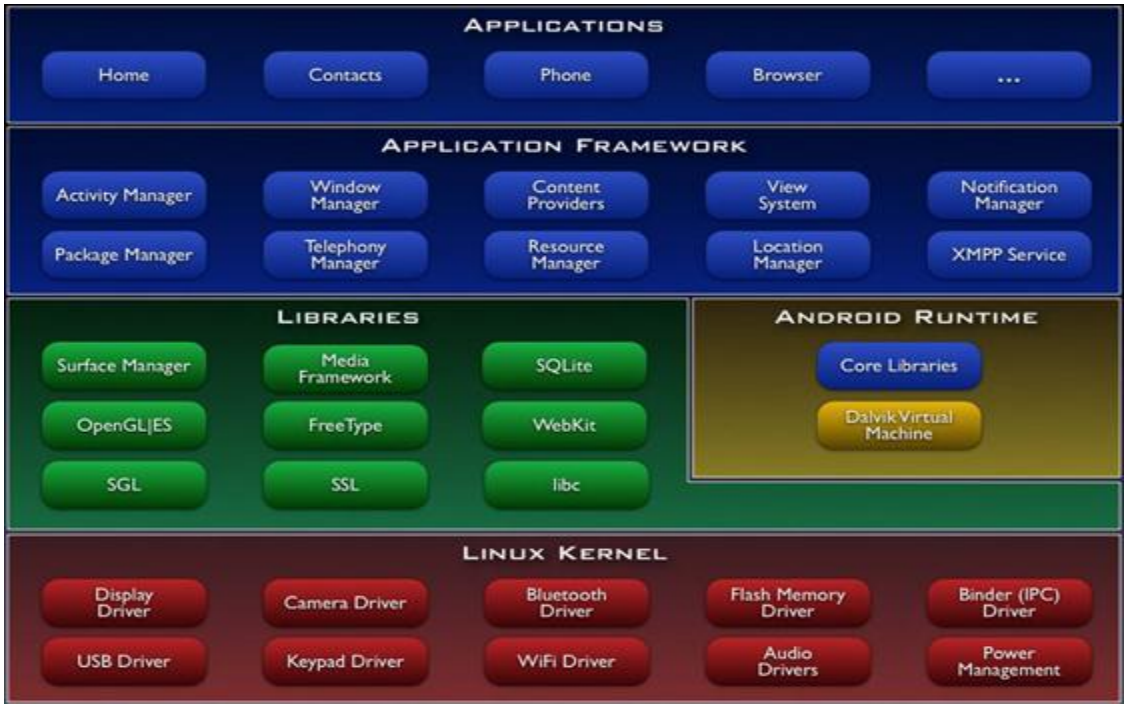


图 1　Android 系统架构图

1.2.1 Linux Kernel 层

Android 核心系统服务依赖于 Linux2.6 内核，如安全性、内存管理、进程管理、网络协议栈和驱动模型。Linux 内核也是作为硬件与软件栈的抽象层。

驱动程序模型(Driver Model)包含以下这些常规的驱动程序：显示驱动、摄像头驱动、键盘驱动、WiFi 驱动、Audio 驱动、flash 内存驱动、Binder（IPC）驱动、电源管理等。

1.2.2 Libraries&Android Runtime 层

这部分内容开始涉及底层，开发普通的应用不会直接对这个层进行操作。这层中包含了一组类库（Libraries）和运行时（RunTime)，主要包含如下：

1） 系统 C 库（libc），一个从 BSD 集成来的标准 C 系统函数库（libc）它是专门为基于嵌入式 Linux 的设备定制的。
2） 媒体库（Media Framework），基于 PackeVideo OpenCore，该库支持多种常用的音频，视频格式以及回放和录制，同时支持静态图像文件。编码格式包括：MPEG4、H.264，MP3、AAC、AMR、JPG、PNG。
3） Surface Manager 对显示子系统的管理，并且为多个应用提供 2D，3D 图层的无缝融合。
4） WebKit，一个最新的 web 浏览器引擎，用来支持 Andiroid 浏览器或者嵌入的 web 视图。
5） SGL 底层的 2D 图形引擎。
6） OPENGL|ES,基于 OpenGL ES 1.0 APIs 实现，该库可以使用硬件 3D 加速或者使用高度优化的 3D 软加速。
7） FreeType，位图（bitmap）和适量（vector）字图显示支持。
8） SQLite，一个对所有应用程序可用，功能强大的轻量级关系型数据库引擎。
9） Core Libraries 该核心库提供了 Java 编程语言核心库的大多数功能。
10） Dalvik VM， android 平台的一个虚拟机，它相当于 PC 中 Java 的虚拟机 JVM。

1.2.3 Framework 层

普通开发者可以使用 Android 基本应用程序使用的系统 API，android 应用框架中的各个模块都可以被复用，各种服务也可以被复用，理解了这个机制，开发人员可以更好的更轻松的开发出优秀的 android 应用。开发过程中常用 到的基本框架组件如下：

1） 一组 View（UI 组件），这些 UI 组件包括 Button(按钮），EidtView（文本框），TextView（标签），List（列表）等等，灵活运用这些组件可以快速方便的创建良好的用户交互界面。
2） Content Providers（内容提供者），听起来有点抽象，通俗的理解 Content Provider 就是提供一种服务，通过这种服务应用程序之间可以实现数据的互相 访问和共享，比如通讯录的存储就使用了这种服务，其它应用也可以访问通讯录中存储的联系人信息。

3） Resource Manager 顾名思义就是资源管理，android 中的资源很多包括图片，用户界面（Layout xml），字体，颜色，UI 组件的 Id 等等都可以称之为资源，这些丰富的资源，都是通过 Resource Manager 来统一进行管理。
4） Notification Manager(消息通知管理），它在不影响用户正常操作和使用设备的时候在状态栏上提供消息的提示。比如有短信来的时候，android 自动会将这个消息提示在状态栏上，用户可以及时的看到。
5） Activity Manager（活动管理），Activity 管理着应用程序的生命周期，并且控制着应用的导航，扮演控制器的角色。每个 Activity 类似于 Windows 应用中的一个 wendow。一般的应用都是通过一个个 Activity 交互构成的。
6） Window Manager(窗口管理），管理所有启动的窗口。
7） Location Manager(位置管理），用来管理地图服务的相关功能。
8） Telephoney Manager(电话管理），用来管理有关的电话的相关功能。
9） Package Manager（包管理），管理所有的安装在 android 系统内的应用程序。

1.2.4 Applications 层

该层提供一些核心应用程序包，例如电子邮件、短信、日历、地图、浏览器和联系人管理等。同时，开发者可以利用 Java 语言设计和编写属于自己的应用程序，而这些程序与那些核心应用程序彼此平等、友好共处。

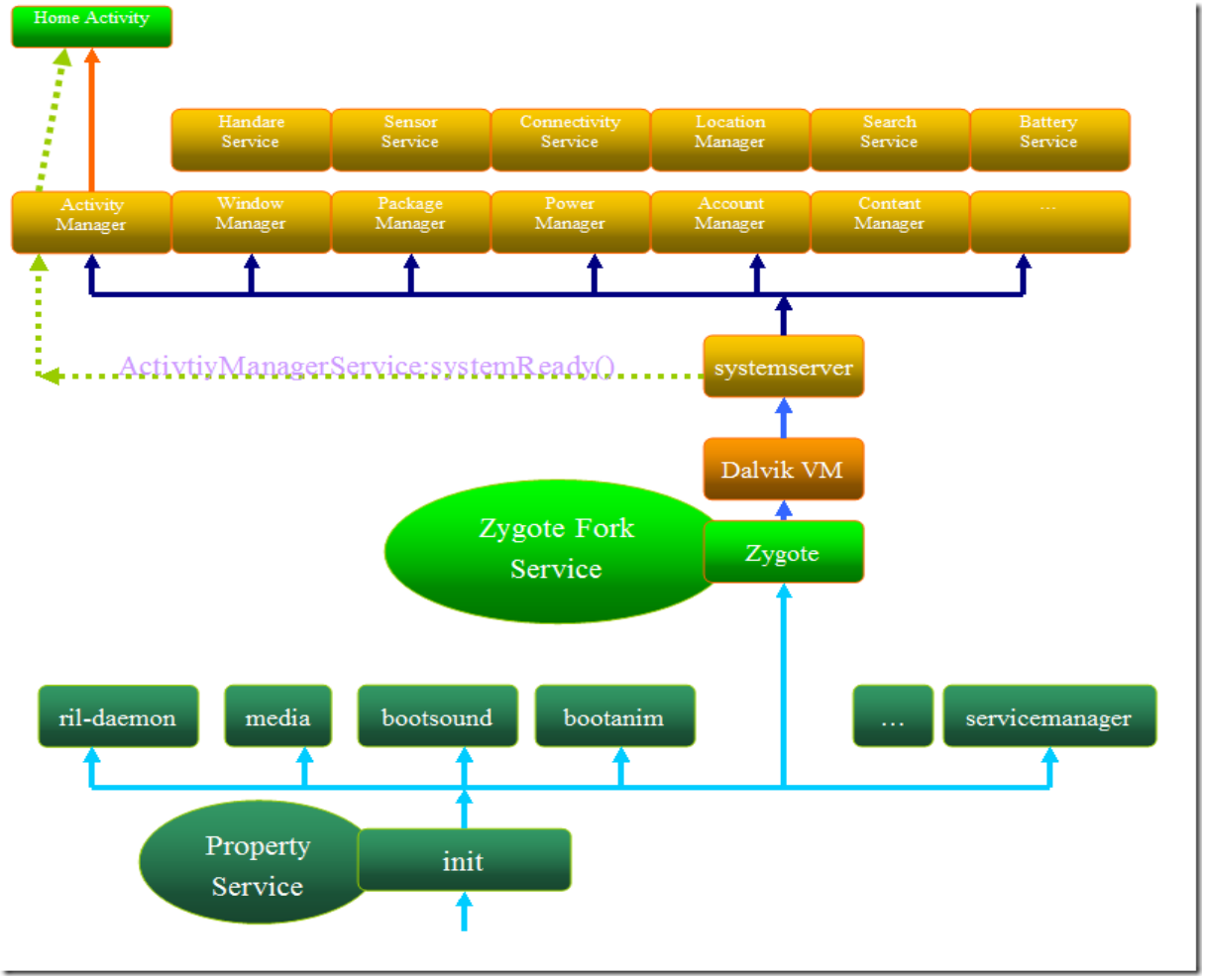
总结：

1、Android 的系统架构采用分层架构的思想，架构清晰，层次分明，协同工作。

2、Android 的系统架构不仅从宏观上认识了 Android 系统，同时，也给我们的学习与实践指明了方向。若是从事 Android 应用开发，那应该研究 Android 的应用框架层和应用程序层；若是从事 Android 系统开发，那应该研究 Android 的系统库和 Android 运行时；若是从事 Android 驱动开发，那应该研究 Android 的 Linux 内核。总之，找准切入点，实践出真知。

第二章　Android 系统启动过程

Linux 内核启动之后就到 Android Init 进程，进而启动 Android 相关的服务和应用。启动的过程如下图所示：



2.1　Init 进程的启动

init 进程，它是一个由内核启动的用户级进程。内核自行启动（已经被载入内存，开始运行，并已初始化所有的设备驱动程序和数据结构等）之后，就通过启动一个用户级程序 init 的方式，完成引导进程。init 始终是第一个进程。

启动过程就是代码 init.c 中 main 函数执行过程：system\core\init\init.c，在函数中执行了：文件夹建立，挂载，rc 文件解析，属性设置，启动服务，执行动作，socket 监听……

下面看两个重要的过程：rc 文件解析和服务启动。

2.1.1 rc 文件解析

.rc 文件是 Android 使用的初始化脚本文件（System/Core/Init/readme.txt 中有描述：four broad classes of statements which are Actions, Commands, Services, and Options.）

其中 Command 就是系统支持的一系列命令，如：export，hostname，mkdir，mount，等等，其中一部分是 linux 命令，还有一些是 android 添加的，如：class_start <serviceclass>：启动服务，class_stop <serviceclass>：关闭服务，等等。

其中 Options 是针对 Service 的选项的。

系统初始化要触发的动作和要启动的服务及其各自属性都在 rc 脚本文件中定义，具体看一下启动脚本：\system\core\rootdir\init.rc。

在解析 rc 脚本文件时，将相应的类型放入各自的 List 中：\system\core\init\Init_parser.c：init_parse_config_file( )存入到 action_queue、action_list、service_list 中，解析过程可以看一下 parse_config 函数。这其中包含了服务：adbd、servicemanager、vold、ril-daemon、debuggerd、surfaceflinger、zygote、media……

2.1.2 服务启动

文件解析完成之后将 service 放入到 service_list 中。

Service 的启动是在 do_class_start 函数中完成：

```
int do_class_start(int nargs, char **args)
{
    service_for_each_class(args[1], service_start_if_not_disabled);
    return 0;
}
```

遍历所有名称为 classname，状态不为 SVC_DISABLED 的 Service 启动

```
void service_for_each_class(const char *classname,
                            void (*func)(struct service *svc))
{
        ……
}

static void service_start_if_not_disabled(struct service *svc)
{
    if (!(svc->flags & SVC_DISABLED)) {
        service_start(svc, NULL);
    }
}
```

do_class_start 对应的命令：

KEYWORD(class_start, COMMAND, 1, do_class_start)

init.rc 文件中搜索 class_start：class_start main 、class_start core、……

main、core 即为 do_class_start 参数 classname

init.rc 文件中 Service class 名称都是 main：
service drm /system/bin/drmserver
class main
service surfaceflinger /system/bin/surfaceflinger
class main

于是就能够通过 main 名称遍历到所有的 Service，将其启动。
do_class_start 调用：
init.rc 中
on boot //action
class_start core //执行 command 对应 do_class_start
class_start main
Init 进程 main 函数中：
system/core/init/init.c 中：

```
int main(){
        //挂在文件
       //解析配置文件: init.rc……
       //初始化化action queue
        ……
       for(;;){

              execute_one_command();

              restart_processes();

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

            if (ufds[i].revents == POLLIN) {

                if (ufds[i].fd == get_property_set_fd())

                    handle_property_set_fd();

                else if (ufds[i].fd == get_keychord_fd())

                    handle_keychord();

                else if (ufds[i].fd == get_signal_fd())

                    handle_signal();
            }
        }

       }
}
```

循环调用 service_start，将状态 SVC_RESTARTING 启动， 将启动后的 service 状态设置为 SVC_RUNNING。

pid=fork();
execve();

在消息循环中：Init 进程执行了 Android 的 Command，启动了 Android 的 NativeService，监听 Service 的变化需求，Signal 处理。Init 进程是作为属性服务（Property service），维护这些 NativeService。

2.2 ServiceManager 的启动

在.rc 脚本文件中 zygote 的描述：

```
service servicemanager /system/bin/servicemanager
    class core
    user system
    group system
    critical
    onrestart restart zygote
    onrestart restart media
    onrestart restart surfaceflinger
    onrestart restart drm
```

ServiceManager 用来管理系统中所有的 binder service，不管是本地的 c++实现的还是 java 语言实现的都需要这个进程来统一管理，最主要的管理就是，注册添加服务，获取服务。所有的 Service 使用前都必须先在 servicemanager 中进行注册。

do_find_service( )

do_add_service( )

svcmgr_handler( )

代码位置：frameworks\base\cmds\servicemanager\Service_manager.c

2.3 Zygote 进程的启动

Zygote 这个进程是非常重要的一个进程，Zygote 进程的建立是真正的 Android 运行空间，初始化建立的 Service 都是 Navtive service。

2.3.1 在.rc 脚本文件中 zygote 的描述

```
service zygote /system/bin/app_process -Xzygote /system/bin --zygote --start-system-server
    class main
    socket zygote stream 666
    onrestart write /sys/android_power/request_state wake
    onrestart write /sys/power/state on
    onrestart restart media
    onrestart restart netd

参数: --zygote --start-system-server
```

代码位置：frameworks/base/cmds/app_process/app_main.cpp

上面的参数在这里就会用上，决定是否要启动和启动那些进程。

```
int main( ){
        AppRuntime runtime;
        if (zygote) {
                runtime.start("com.android.internal.os.ZygoteInit",
                  startSystemServer ? "start-system-server" : "");
        }
}

class AppRuntime : public AndroidRuntime{};
```

接着到了 AndroidRuntime 类中：frameworks\base\core\jni\AndroidRuntime.cpp

```
void start(const char* className, const char* options){

        // start the virtual machine Java在虚拟机中运行的
        JNIEnv* env;
        if (startVm(&mJavaVM, &env) != 0) {
                return;
        }

        //向刚刚新建的虚拟机注册JNI本地接口
        if (startReg(env) < 0) {
                return;
        }

        // jni 调用 java 方法，获取对应类的静态main方法
        jmethodID startMeth = env->GetStaticMethodID(startClass,
            "main","([Ljava/lang/String;)V");

        // jni调用 java方法，调用到ZygoteInit类的main函数

        jclass startClass = env->FindClass(className);

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

到了 ZygoteInit.java 中的静态 main 函数中，从 C++ —>JAVA

2.3.2 ZygoteInit

真正Zygote 进程: frameworks\base\core\java\com\android\internal\os\ZygoteInit.java

```
public static void main(String argv[]) {
        //Registers a server socket for zygote command connections
        registerZygoteSocket();

        //Loads and initializes commonly used classes and
        //used resources that can be shared across processes
        preload();

        // Do an initial gc to clean up after startup
        gc();

        if (argv[1].equals("start-system-server")) {
                startSystemServer();
        }

        /**
        * 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.
        */

        runSelectLoopMode();    //loop中
        /**
        * Close and clean up zygote sockets. Called on shutdown and on the
        * child's exit path.
        */
        closeServerSocket();
}
```

Zygote 就建立好了，利用 Socket 通讯，接收请求，Fork 应用程序进程，进入 Zygote 进程服务框架中。

2.4 SystemServer 的启动

（1）在 Zygote 进程进入循环之前，调用了 startSystemServer( );

（2）RuntimeInit 中：

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

```
//The main function called when started through the zygote process.
void zygoteInit(int targetSdkVersion, String[] argv){
        applicationInit(targetSdkVersion, argv);
}

void applicationInit(int targetSdkVersion, String[] argv){
    // Remaining arguments are passed to the start class's static main
    invokeStaticMain(args.startClass, args.startArgs);
}

void invokeStaticMain(String className, String[] argv){
    Class<?> cl;
    cl = Class.forName(className);

    //获取SystemServer的main方法，抛出MethodAndArgsCaller异常
    Method m;
    m = cl.getMethod("main", new Class[] { String[].class });
    int modifiers = m.getModifiers();
    throw new ZygoteInit.MethodAndArgsCaller(m, argv);
}
```

（3）从 startSystemServer 开始执行并没有去调用 SystemServer 的任何方法，只是通过反射获取了 main 方法，付给了 MethodAndArgsCaller，并抛出了 MethodAndArgsCaller 异常。

此异常是在哪里处理的呢？

回到 startSystemServer( )函数的调用处：

在 ZygoteInit 的 main 函数中：

```
public static void main(String argv[]) {
        try {
                ……
                if (argv[1].equals("start-system-server")) {
                    startSystemServer();        //这里如果抛出异常，跳过下面流程
                }

                runSelectLoopMode();     //loop中
                ……


        } catch (MethodAndArgsCaller caller) {
                caller.run();        //处理的异常
        }
}
```

如果 startSystemServer 抛出了异常，跳过执行 ZygoteInit 进程的循环，这是怎么回事呢？

在 startSystemServer 中异常是由 handleSystemServerProcess 抛出，而

```
pid = Zygote.forkSystemServer( )
/* For child process 仅对新的子进程设置 */
    if (pid == 0) {
        handleSystemServerProcess(parsedArgs);
    }
```

Zygote.forkSystemServer 根据参数 fork 出一个子进程，若成功调用，则返回两次：一次返回的是 zygote 进程的 pid ，值大于 0；一次返回的是子进程 pid，值等于 0 否则，出错返回-1；

caller.run();

MethodAndArgsCaller run 函数：调用前面所提到的 SystemServer main 方法

m = cl.getMethod("main", new Class[] { String[].class });

启动了进程 SystemServer。

（4）SystemServer 的执行 init1( )

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

```
public static void main(String[] args) {

        System.loadLibrary("android_servers");
        init1(args);    //native 完了回调init2( )
    }

//init1:
    frameworks/base/services/jni/com_android_server_SystemServer.cpp:: android_server_SystemServer_init1( )
    中调用: system_init
extern "C" status_t system_init()
{
    sp<ProcessState> proc(ProcessState::self());
    sp<IServiceManager> sm = defaultServiceManager();

    //启动SurfaceFlinger 和传感器
    property_get("system_init.startsurfaceflinger", propBuf, "1");
    SurfaceFlinger::instantiate();

    property_get("system_init.startsensorservice", propBuf, "1");
    SensorService::instantiate();

    AndroidRuntime* runtime = AndroidRuntime::getRuntime();

    //回调 com.android.server.SystemServer init2 方法

    JNIEnv* env = runtime->getJNIEnv();
    jclass clazz = env->FindClass("com/android/server/SystemServer");
    jmethodID methodId = env->GetStaticMethodID(clazz, "init2", "()V");
    env->CallStaticVoidMethod(clazz, methodId);

    //启动线程池 做为binder 服务
    ProcessState::self()->startThreadPool();
    IPCThreadState::self()->joinThreadPool();
    return NO_ERROR;
}
```

ProcessState：

每个进程在使用 binder 机制通信时，均需要维护一个 ProcessState 实例来描述当前进程在 binder 通信时的 binder 状态。

ProcessState 有如下 2 个主要功能：

1. 创建一个 thread，该线程负责与内核中的 binder 模块进行通信，称该线程为 Pool thread ；

2. 为指定的 handle 创建一个 BpBinder 对象，并管理该进程中所有的 BpBinder 对象。

Pool thread：

在 Binder IPC 中，所有进程均会启动一个 thread 来负责与 BD 来直接通信，也就是不停的读写 BD ，

这个线程的实现主体是一个 IPCThreadState 对象，下面会介绍这个类型。

下面是 Pool thread 的启动方式：

ProcessState::self()->startThreadPool();

IPCThreadState ：

IPCThreadState 也是以单例模式设计的。由于每个进程只维护了一个 ProcessState 实例，同时 ProcessState 只启动一个 Pool thread ，

也就是说每一个进程只会启动一个 Pool thread ，因此每个进程则只需要一个 IPCThreadState 即可。

Pool thread 的实际内容则为：

IPCThreadState::self()->joinThreadPool();

（5）SystemServer 的执行 init2( )

```
public static final void init2() {
        //建立线程来处理
        Thread thr = new ServerThread();
        thr.setName("android.server.ServerThread");
        thr.start();
}

//看看线程ServerThread里面都做了什么事情？
public void run() {
    addBootEvent(new String("Android:SysServerInit_START"));
    Looper.prepare();
    android.os.Process.setThreadPriority(
    android.os.Process.THREAD_PRIORITY_FOREGROUND);

    //初始化服务，创建各种服务实例，如：电源、网络、Wifi、蓝牙，USB等，
    //初始化完成以后加入到 ServiceManager中，
    //事我们用 Context.getSystemService (String name) 才获取到相应的服务
    PowerManagerService power = null;
    NetworkManagementService networkManagement = null;
    WifiP2pService wifiP2p = null;
    WindowManagerService wm = null;
    BluetoothService bluetooth = null;
    UsbService usb = null;
    NotificationManagerService notification = null;
    StatusBarManagerService statusBar = null;
    ……

    power = new PowerManagerService();
    ServiceManager.addService(Context.POWER_SERVICE, power);
    ……
```

```
    // ActivityManagerService作为ApplicationFramework最重要的服务
    ActivityManagerService.setSystemProcess();
    ActivityManagerService.installSystemProviders();
    ActivityManagerService.self().setWindowManager(wm);
   // We now tell the activity manager it is okay to run third party
   // code.  It will call back into us once it has gotten to the state
   // where third party code can really run (but before it has actually
   // started launching the initial applications), for us to complete our
   // initialization.
   //系统服务初始化准备就绪，通知各个模块
    ActivityManagerService.self().systemReady(new Runnable() {

          public void run() {
                startSystemUi(contextF);
                batteryF.systemReady();
                networkManagementF.systemReady();
                usbF.systemReady();
                ……

                // It is now okay to let the various system services start their
                // third party code...
                appWidgetF.systemReady(safeMode);
                wallpaperF.systemReady();
          }
    });

    //
    //BOOTPROF
    addBootEvent(new String("Android:SysServerInit_END"));
    Looper.loop();
}
```

到这里系统 ApplicationFramework 层的 XxxServiceManager 准备就绪，可以开始跑上层应用了，我们的第一个上层应用 HomeLauncher。

HomeActivity 又是如何启动的呢？

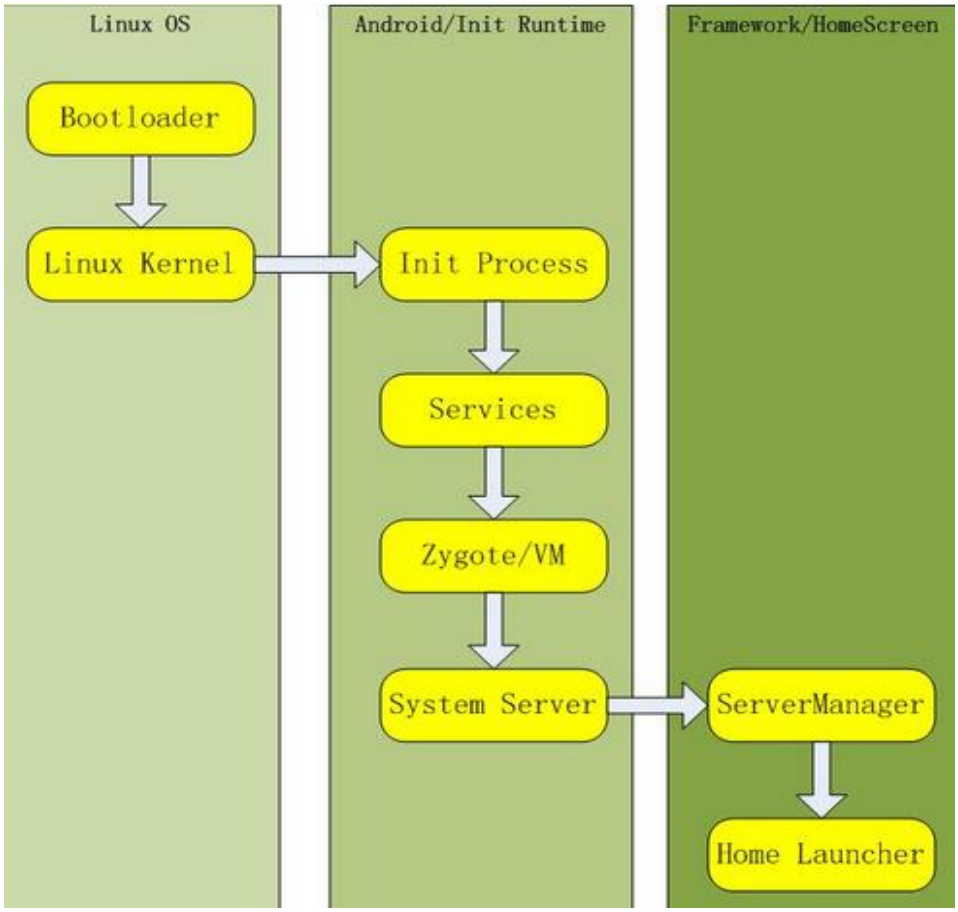
Activity 的启动必然和 ActivityManagerService 有关，我们需要去看看 ActivityManagerService.systemReady( )中都干了些什么。

2.5 Home 界面的启动

```
 public void systemReady(final Runnable goingCallback) {
        ……
        //ready callback
        if (goingCallback != null)
                goingCallback.run();

        synchronized (this) {
                // Start up initial activity.
                // ActivityStack mMainStack;
                mMainStack.resumeTopActivityLocked(null);
        }
……

}

final boolean resumeTopActivityLocked(ActivityRecord prev) {
    // Find the first activity that is not finishing.
    ActivityRecord next = topRunningActivityLocked(null);
    if (next == null) {
        // There are no more activities!  Let's just start up the
        // Launcher...
        if (mMainStack) {
            //ActivityManagerService mService;
            return mService.startHomeActivityLocked();
        }
    }
    ……
}
```

然后就启动了 Home 界面，完成了整个 Android 启动流程。

整个过程如下：


Linux OS
Android/Init Runtime
Framework/HomeScreen
Bootloader
Linux Kernel
Init Process
Services
Zygote/VM
System Server
ServerManager
Home Launcher

第三章 sensor 体系结构

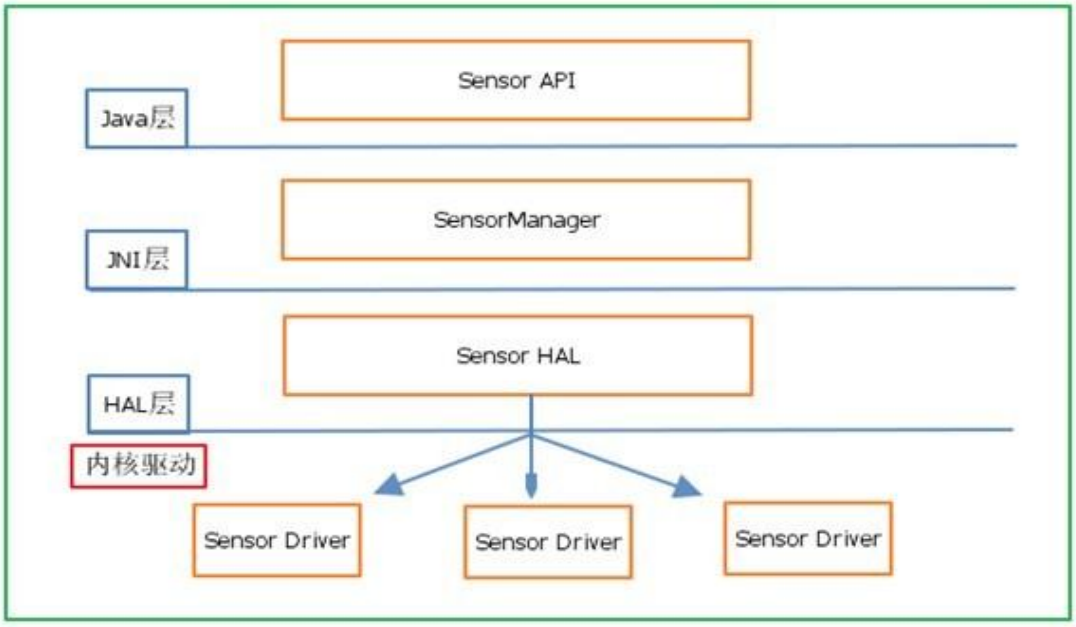
3.1 Android Sensor 架构

Android4.0 系统内置对传感器的支持达 13 种，它们分别是：加速度传感器(accelerometer)、磁力传感器(magnetic field)、方向传感器(orientation)、陀螺仪(gyroscope)、环境光照传感器(light)、压力传感器(pressure)、温度传感器(temperature)和距离传感器(proximity)等。

Android 实现传感器系统包括以下几个部分：

类别	名称	代码
用户空间	Java 应用程序	用户实现
	Java framework 框架层	SensorManager.java SensorListener.java SensorEvent.java …
	JNI 层	adroid_hardware_SensorManager.cpp com_android_server_SensorService.cpp
	HAL 硬件抽象层	用户实现（sensor.c）
内核空间	设备驱动程序	用户实现
具体硬件	重力传感器、温度传感器 LM75、电压测量传感器	用户实现

各部分之间架构图如下：



3.2 Sensor HAL 层接口

Google 为 Sensor 提供了统一的 HAL 接口，不同的硬件厂商需要根据该接口来实现并完成具体的硬件抽象层，Android 中 Sensor 的 HAL 接口定义在：hardware/libhardware/include/hardware/sensors.h.

对传感器类型的定义如下：

```
#define SENSOR_TYPE_ACCELEROMETER        1
#define SENSOR_TYPE_MAGNETIC_FIELD       2
#define SENSOR_TYPE_ORIENTATION          3
#define SENSOR_TYPE_GYROSCOPE            4
#define SENSOR_TYPE_LIGHT                5
#define SENSOR_TYPE_PRESSURE             6
#define SENSOR_TYPE_TEMPERATURE          7
#define SENSOR_TYPE_PROXIMITY            8
#define SENSOR_TYPE_GRAVITY              9
#define SENSOR_TYPE_LINEAR_ACCELERATION 10
#define SENSOR_TYPE_ROTATION_VECTOR     11
#define SENSOR_TYPE_RELATIVE_HUMIDITY   12
#define SENSOR_TYPE_AMBIENT_TEMPERATURE 13
```

传感器模块的定义结构体如下：

```
struct sensors_module_t {
    struct hw_module_t common;

    /**
     * Enumerate all available sensors. The list is returned in "list".
     * @return number of sensors in the list
     */
    int (*get_sensors_list)(struct sensors_module_t* module,
            struct sensor_t const** list);
};
```

该接口的定义实际上是对标准的硬件模块 hw_module_t 的一个扩展，增加了一个 get_sensors_list 函数，用于获取传感器的列表。

对任意一个 sensor 设备都会有一个 sensor_t 结构体，其定义如下：

```
struct sensor_t {
    /* name of this sensors */
    const char*     name;
    /* vendor of the hardware part */
    const char*     vendor;
    /* version of the hardware part + driver. The value of this field
     * must increase when the driver is updated in a way that changes the
     * output of this sensor. This is important for fused sensors when the
     * fusion algorithm is updated.
     */
    int             version;
    /* handle that identifies this sensors. This handle is used to activate
     * and deactivate this sensor. The value of the handle must be 8 bits
     * in this version of the API.
     */
    int             handle;
    /* this sensor's type. */
    int             type;
    /* maximaum range of this sensor's value in SI units */
    float           maxRange;
    /* smallest difference between two values reported by this sensor */
    float           resolution;
    /* rough estimate of this sensor's power consumption in mA */
    float           power;
    /* minimum delay allowed between events in microseconds. A value of zero
     * means that this sensor doesn't report events at a constant rate, but
     * rather only when a new data is available */
    int32_t         minDelay;
    /* reserved fields, must be zero */
    void*           reserved[8];
};
```

每个传感器的数据由 sensors_event_t 结构体表示，定义如下：

```
typedef struct sensors_event_t {
    /* must be sizeof(struct sensors_event_t) */
    int32_t version;
    /* sensor identifier */
    int32_t sensor;
    /* sensor type */
    int32_t type;
    /* reserved */
    int32_t reserved0;
    /* time is in nanosecond */
    int64_t timestamp;
    union {
        float           data[16];
        /* acceleration values are in meter per second per second (m/s^2) */
        sensors_vec_t   acceleration;
        /* magnetic vector values are in micro-Tesla (uT) */
        sensors_vec_t   magnetic;
        /* orientation values are in degrees */
        sensors_vec_t   orientation;
        /* gyroscope values are in rad/s */
        sensors_vec_t   gyro;
        /* temperature is in degrees centigrade (Celsius) */
        float           temperature;
        /* distance in centimeters */
        float           distance;
        /* light in SI lux units */
        float           light;
        /* pressure in hectopascal (hPa) */
        float           pressure;
        /* relative humidity in percent */
        float           relative_humidity;
    };
    uint32_t        reserved1[4];
} sensors_event_t;
```

其中，sensor 为传感器的标志符，而不同的传感器则采用 union 方式来表示，sensors_vec_t 结构体用来表示不同传感器的数据，sensors_vec_t 定义如下：

```
typedef struct {
    union {
        float v[3];
        struct {
            float x;
            float y;
            float z;
        };
        struct {
            float azimuth;
            float pitch;
            float roll;
        };
    };
    int8_t status;
    uint8_t reserved[3];
} sensors_vec_t;
```

Sensor 设备结构体 sensors_poll_device_t，对标准硬件设备 hw_device_t 结构体的扩展，主要完成读取底层数据，并将数据存储在 struct sensors_poll_device_t 结构体中，poll 函数用来获取底层数据，调用时将被阻塞定义如下：

```
struct sensors_poll_device_t {
    struct hw_device_t common;

    int (*activate)(struct sensors_poll_device_t *dev,
            int handle, int enabled);

    int (*setDelay)(struct sensors_poll_device_t *dev,
            int handle, int64_t ns);

    int (*poll)(struct sensors_poll_device_t *dev,
            sensors_event_t* data, int count);
};
```

控制设备打开/关闭结构体定义如下：

```
static inline int sensors_open(const struct hw_module_t* module,
        struct sensors_poll_device_t** device) {
    return module->methods->open(module,
            SENSORS_HARDWARE_POLL, (struct hw_device_t**)device);
}

static inline int sensors_close(struct sensors_poll_device_t* device) {
    return device->common.close(&device->common);
}
```

3.3 JNI/HAL 编程实现

3.3.1 JNI 层代码简析

SensorManager 的构造函数调用 sensors_module_init();，以及间接调用 get_sensors_list 来获取传感器的列表。

sensors_module_init 是一个 JNI 函数，其实现如下：

```
static jint sensors_module_init(JNIEnv *env, jclass clazz)
{
    int err = 0;
    sensors_module_t const* module;
    err = hw_get_module(SENSORS_HARDWARE_MODULE_ID, (const hw_module_t **)&module);
    if (err == 0)
        sSensorModule = (sensors_module_t*)module;
    return err;
}
```

hw_get_module 函数是一个 HAL 层的标准 API 函数调用，它通过 SENSORS_HARDWARE_MODULE_ID 名字得到一个 hw_module_t 对象，这样就得到了 sensors_module_t 对象。

3.3.2 HAL 层代码简析

首先在 mediatek\hardware\sensor\hwmsen\ hwmsen_chip_info.c 中定义传感器数组 sSensorList[MAX_NUM_SENSOR]，其实就是初始化 struct sensor_t 结构体，初始化如下：

```
struct sensor_t sSensorList[MAX_NUM_SENSOR] =
{
#ifdef CUSTOM_KERNEL_ACCELEROMETER
    {
        .name          = ACCELEROMETER,
        .vendor        = ACCELEROMETER_VENDER,
        .version       = 1,
        .handle        = ID_ACCELEROMETER,
        .type          = SENSOR_TYPE_ACCELEROMETER,
        .maxRange      = ACCELEROMETER_RANGE,//32.0f,
        .resolution = ACCELEROMETER_RESOLUTION,//4.0f/1024.0f,
        .power         = ACCELEROMETER_POWER,//130.0f/1000.0f,
        .reserved      = {}
    },
#endif

    ……
};
```

然后在 mediatek\hardware\sensor\hwmsen\ sensors_hwmsen.c 定义接口，以供 JNI 层调用，其实现原理如下所述。

1、定义传感器模块结构体

```
struct sensors_module_t HAL_MODULE_INFO_SYM = {
    .common = {
        .tag = HARDWARE_MODULE_TAG,
        .version_major = 1,
        .version_minor = 0,
        .id = SENSORS_HARDWARE_MODULE_ID,
        .name = "MTK SENSORS Module",
        .author = "The Android Open Source Project",
        .methods = &sensors_module_methods,
    },
    .get_sensors_list = sensors__get_sensors_list,
};
```

2、实现传感器结构体中的 methods

```
static struct hw_module_methods_t sensors_module_methods = {
    .open = open_sensors
};
```

实现 open_sensors 方法：

```
static int open_sensors(const struct hw_module_t* module, const char* name,
        struct hw_device_t** device)
{
    int status = -EINVAL;
    ALOGD("%s: name: %s!\r\n", __func__, name);

    struct sensors_data_context_t *dev;
    dev = malloc(sizeof(*dev));
    memset(dev, 0, sizeof(*dev));
    dev->events_fd = open_input(O_RDONLY);
    if (dev->events_fd < 0)
    {
        ALOGE("%s: Open input device error!",__func__);
        return -1;
    }
    dev->device_io_fd = -1;
    if(dev->device_io_fd <= 0)
    {
        dev->device_io_fd = open(HWM_SENSOR_DEV, O_RDONLY);
        ALOGD("%s: device handle %d",__func__, dev->device_io_fd);
    }
    dev->activate =  hwm__activate;
    dev->set_delay = hwm__set_delay;
    dev->poll = hwm__poll;
    dev->device.common.tag = HARDWARE_DEVICE_TAG;
    dev->device.common.version  = 0;
    dev->device.common.module   = (struct hw_module_t*)module;
    dev->device.common.close    = data__close;
    dev->device.activate        = control__activate;
    dev->device.setDelay        = control__setDelay;
    dev->device.poll            = data__poll;
    *device = &dev->device.common;
    status = 0;
    return status;
}
```

由以上 open_sensors 函数中可知，最后*device = &dev->device.common；即将创建并初始化好的 sensors_data_context_t 结构体给第三个形参 struct hw_device_t** device，使得 device 指针指向一个 device 对象，该对象含有各种已经初始化好的属性和方法。这些方法就是在 Linux 驱动层中进行实现的，所以通过操作这些接口，可以实现和实际的 sensor 硬件进行交互。

在 JNI 层中通过调用 hw_get_module 函数获取到 sensors_module_t 对象。从而得到 sensors_module_t 对象中的 open 方法，即 open_sensors 方法。而由 HAL 层代码分析可知，调用 open_sensors 方法的同时会将创建好的 sensors_data_context_t 结构体（含有 device 对象）给第三个形参 struct hw_device_t** device，即通过 device 指针的方法可以控制相应的 sensor 硬件。

3.4 Sensor 驱动代码简析

以陀螺仪为例说明。

mediatek\custom\common\kernel\gyroscope\mpu3050c\mpu3050c.c

模块加载函数 mpu3000_init()进行初始化工作，函数实现如下：

```
static int __init mpu3000_init(void)
{
    struct gyro_hw *hw = get_cust_gyro_hw();
    GYRO_FUN();
    GYRO_LOG("%s: i2c_number=%d\n", __func__,hw->i2c_num);
    i2c_register_board_info(hw->i2c_num, &i2c_mpu3000, 1);
    if(platform_driver_register(&mpu3000_gyro_driver))
    {
        GYRO_ERR("failed to register driver");
        return -ENODEV;
    }
    return 0;
}
```

由代码可知，函数主要进行 i2c 板级信息注册和平台驱动注册。

接下来来看一下平台总线的定义，如下：

```
struct bus_type platform_bus_type = {        //定义了一条虚拟的平台总线
    .name        = "platform",
    .dev_attrs   = platform_dev_attrs,
    .match       = platform_match,
    .uevent      = platform_uevent,
    .pm      = &platform_dev_pm_ops,
};
```

其中，match 函数的实现如下：

```
/*匹配函数,匹配原理*/
static int platform_match(struct device *dev, struct device_driver *drv)
{
    struct platform_device *pdev = to_platform_device(dev);
    struct platform_driver *pdrv = to_platform_driver(drv);

    /* match against the id table first */
    if (pdrv->id_table)     //如果驱动中实现了id_table则按id匹配
        return platform_match_id(pdrv->id_table, pdev) != NULL;
            while (id->name[0]) {
                if (strcmp(pdev->name, id->name) == 0) {    //由此处可知,其实是按id中的名字来匹配的
                    pdev->id_entry = id;
                    return id;
                }
                id++;
            }

    /* fall-back to driver name match */
    return (strcmp(pdev->name, drv->name) == 0);            //若没id_table，则按设备与驱动的名字进行匹配
}
```

由 platform_match 函数可知，平台设备和平台驱动优先以 id_table 进行匹配，匹配成功后，调用对应的 probe 函数，即 mpu3000_probe 函数，其实现如下：

```
static int mpu3000_probe(struct platform_device *pdev)
{
    struct gyro_hw *hw = get_cust_gyro_hw();
    GYRO_FUN();

    MPU3000_power(hw, 1);
    //mpu3000_force[0] = hw->i2c_num;
    if(i2c_add_driver(&mpu3000_i2c_driver))
    {
        GYRO_ERR("add driver error\n");
        return -1;
    }
    return 0;
}
```

由 mpu3000_probe 函数可知，其主要工作是将 i2c 驱动加入到 i2c 驱动链表，当 i2c 驱动与 i2c 设备匹配成功后调用对应的 probe 函数，即 mpu3000_i2c_probe 函数：

```
static int mpu3000_i2c_probe(struct i2c_client *client, const struct i2c_device_id *id)
{
    struct i2c_client *new_client;
    struct mpu3000_i2c_data *obj;
    struct hwmsen_object sobj;
    int err = 0;
    GYRO_FUN();
    if(!(obj = kzalloc(sizeof(*obj), GFP_KERNEL)))
    {
        err = -ENOMEM;
        goto exit;
    }
    memset(obj, 0, sizeof(struct mpu3000_i2c_data));
    obj->hw = get_cust_gyro_hw();
    err = hwmsen_get_convert(obj->hw->direction, &obj->cvt);
    if(err)
    {
        GYRO_ERR("invalid direction: %d\n", obj->hw->direction);
        goto exit;
    }
    GYRO_LOG("gyro_default_i2c_addr: %x\n", client->addr);
    GYRO_LOG("gyro_custom_i2c_addr: %x\n", obj->hw->addr);
    if(0!=obj->hw->addr)
    {
      client->addr = obj->hw->addr >> 1;
      GYRO_LOG("gyro_use_i2c_addr: %x\n", client->addr);
    }
    obj_i2c_data = obj;
    obj->client = client;
    new_client = obj->client;
    i2c_set_clientdata(new_client,obj);

    atomic_set(&obj->trace, 0);
    atomic_set(&obj->suspend, 0);

    mpu3000_i2c_client = new_client;
    err = mpu3000_init_client(new_client, false);
    if(err)
    {
        goto exit_init_failed;
    }


    err = misc_register(&mpu3000_device);
    if(err)
    {
        GYRO_ERR("mpu3000_device misc register failed!\n");
        goto exit_misc_device_register_failed;
    }

    err = mpu3000_create_attr(&mpu3000_gyro_driver.driver);
    if(err)
    {
        GYRO_ERR("mpu3000 create attribute err = %d\n", err);
        goto exit_create_attr_failed;
    }


    sobj.self = obj;
    sobj.polling = 1;
    sobj.sensor_operate = mpu3000_operate;
    err = hwmsen_attach(ID_GYROSCOPE, &sobj);
    if(err)
    {
        GYRO_ERR("hwmsen_attach fail = %d\n", err);
        goto exit_kfree;
    }
```

```
#ifdef CONFIG_HAS_EARLYSUSPEND
    obj->early_drv.level    = EARLY_SUSPEND_LEVEL_STOP_DRAWING - 2,
    obj->early_drv.suspend  = mpu3000_early_suspend,
    obj->early_drv.resume   = mpu3000_late_resume,
    register_early_suspend(&obj->early_drv);
#endif

    GYRO_LOG("%s: OK\n", __func__);
    return 0;

    exit_create_attr_failed:
    misc_deregister(&mpu3000_device);
    exit_misc_device_register_failed:
    exit_init_failed:
    //i2c_detach_client(new_client);
    exit_kfree:
    kfree(obj);
    exit:
    GYRO_ERR("%s: err = %d\n", __func__, err);
    return err;
}
```

分析以上代码可知，mpu3000_i2c_probe 函数进行一系列的初始化工作，并最终创建出设备属性和设备文件，之后上层就可以通过操作设备文件或设备属性，使用 file_operation 结构体提供的方法，对硬件进行相应的操作。

其中 mpu3000_i2c_probe 函数做了几件比较重要的工作是：

1、 misc_register(&mpu3000_device);

将设备注册到设备链表

```
static struct file_operations mpu3000_fops = {
//  .owner = THIS_MODULE,//modified
    .open = mpu3000_open,
    .release = mpu3000_release,
    .unlocked_ioctl = mpu3000_unlocked_ioctl,
};
/*--------------------------------------------------------------------------*/
static struct miscdevice mpu3000_device = {
    .minor = MISC_DYNAMIC_MINOR,
    .name = "gyroscope",
    .fops = &mpu3000_fops,
};
```

由上图可知，在进行设备注册的同时，提供了一些操作设备的方法，即 file_operation 中的方法。

2、 mpu3000_create_attr(&mpu3000_gyro_driver.driver);

创建设备属性，之后上层可以通过设备属性对设备进行操作。

3、 hwmsen_attach(ID_GYROSCOPE, &sobj);

hwmsen_attach 函数，把 sensor_operate 接口加到 hwmsen_dev 的列表里，这样 hwmsen_dev 里就能调用所有 sensor 的 sensor_operate 函数。

第四章　陀螺仪装配方位说明

4.1 方位矩阵

在 InvenSense 的 Motion Processing Library 种装配方位。在转换矩阵中可以调整传感器轴向的定义。这种方法使用户只是修改矩阵就可以适应传感器的各种装配位置件做出修改。
方位矩阵对传感器轴向的改变如下所示

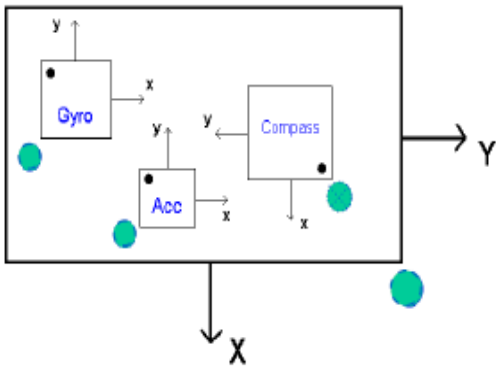
$$\begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} -y \\ x \\ z \end{bmatrix}$$

所以通过方位矩阵可以对陀螺仪方向进行调整，使得陀螺仪的正方向和参考坐标系方向一致。

4.2 方位确定步骤

1、设定参考坐标系，如图所示，这是按照 OpenGL 定义出来的参考坐标系：

2、在 PCB 板中找出陀螺仪芯片的正方向，确定其坐标系。
3、按照系统（手机）的参考坐标系，找出陀螺仪芯片坐标系与参考坐标系的关系，例如下图：



由上图有，参考坐标系的 x 轴向下，y 轴向右，根据右手坐标系定则，z 轴垂直指出纸面。用参考坐标系坐标来表示陀螺仪的坐标有（y，-x，z），再由下图：

- The value could be defined as 0 ~ 7

Value	Description
0	{x, y, z} => { x, y, z}
1	{x, y, z} => {-y, x, z}
2	{x, y, z} => {-x, -y, z}
3	{x, y, z} => { y, -x, z}
4	{x, y, z} => {-x, y, -z}
5	{x, y, z} => { y, x, -z}
6	{x, y, z} => { x, -y, -z}
7	{x, y, z} => {-y, -x, -z}

可以确定其方向值为 3。

所以可以在配置文 mediatek\custom\$(Project)\kernel\gyroscope\mpu3050c\cust_gyro.c 中将方位调整值. Direction 修改为 3，如下：

```
static struct gyro_hw cust_gyro_hw = {
    .i2c_num = 2,
    .direction = 3,
    .power_id = MT65XX_POWER_NONE,  /*!< LDO is not used */
    .power_vol= VOL_DEFAULT,        /*!< LDO is not used */
    .firlen = 16,                   /*!< don't enable low pass fileter */
   // .power = cust_gyro_power,
};
```

第五章　工具命令学习

5.1 adb 常用命令

adb 的全称为 Android Debug Bridge，就是起到调试桥的作用。通过 adb 我们可以在 Eclipse 中方便通过 DDMS 来调试 Android 程序，说白了就是 debug 工具。adb 的工作方式比较特殊，采用监听 Socket TCP 5554 等端口的方式让 IDE 和 Qemu 通讯，默认情况下 adb 会 daemon 相关的网络端口，所以当我们运行 Eclipse 时 adb 进程就会自动运行。

adb 是 android sdk 里的一个工具，用这个工具可以直接操作管理 android 模拟器或者真实的 andriod 设备(如 G1 手机).　它的主要功能有:

* 运行设备的 shell(命令行)
* 管理模拟器或设备的端口映射
* 计算机和设备之间上传/下载文件
* 将本地 apk 软件安装至模拟器或 android 设备

ADB 是一个 客户端-服务器端 程序，其中客户端是你用来操作的电脑，服务器端是 android 设备.其常用命令有：

1）查看设备情况

adb devices

该命令返回的结果为连接的 android 设备序列号及状态。序列号用于唯一标示一台 android 设备，

通常为<设备名称>-<端口号>

2）获取设备的 SN 号

adb get-serialno

3）安装软件

adb install xxx.apk　(xxx.apk 需要包含软件包的路径)

---这个命令将指定的 apk 文件安装到设备上。

adb install –r xxx.apk

---表示重新安装软件的意思。

4）卸载软件

adb uninstall <包名>

adb uninstall　– k <包名>

---如果加 -k 参数，表示卸载软件但是保留配置和缓存文件.

5）进入手机 shell

adb shell

---进入后通过 ls，例出当前的文件目录

在这个 Linux　Shell 中，你可以执行各种 Linux 的命令，另外如果另想执行一条 shell 命令，可以采用以下的方式：adb shell [command]

如：adb shell dmesg 就会打印出内核的调试信息。

6）发布端口

可以设置任意的端口号，做为主机向模拟器或设备的请求端口。如：

adb forward tcp: 5555 tcp:8000

7）从电脑上发送文件到设备

adb push <本地路径> <远程路径>

---用 push 命令可以把本机电脑上的文件戒者文件夹复制到设备(手机)，此处的本地就是指电脑，远程就是指手机戒者模拟器。

8）从设备上下载文件到电脑

adb pull <远程路径> <本地路径>

---用 pull 命令可以把设备(手机)上的文件戒者文件夹复制到本机电脑中。

9）查看 bug 报告

adb bugreport

10）查看手机上的运行日志，此项可以用来查错

adb logcat

11）重新挂载文件系统

adb remount

12）重启手机

adb reboot

13）重启到 Recovery 界面

adb reboot recovery

14）重启到 bootloader 界面

adb reboot bootloader

15）进行 adb 运行环境

adb shell

16.）设置超级用户模式

su

17）映射出目录

mount -o remount,rw -t yaffs2 /dev/block/mtdblock3 /system

18）退出 su

exit

19）退出 shell 准备接受下一个 adb 命令
exit

20）增加 flash_image 文件到手机上
adb push flash_image.zip /system/bin/flash_image

21）recovery 先复制到卡上
adb push recovery-RA-heroc-v1.6.2.img /sdcard

22）再次进入 adb 环境
adb shell

23）修改文件属性
chmod 0755 /system/bin/flash_image

24）重启系统
reboot

5.2 linux 下 svn 常用命令

Subversion（svn）是一个免费的、开放源码的版本控制系统。就是说，Subversion 管理随时改动的文件和目录。文件树被放在一个中央资料库（Repository）中。这个资料库象一个普通的文件服务器，但是它可以记录文件和目录的每次改动。它允许你恢复以前旧版本的数据，或者检查数据变化的历史。因此，很多人把版本控制系统看成某种“时光机”。

Linux 下 svn 的常用命令如下：

1）将文件 checkout 到本地目录
svn checkout path（path 是服务器上的目录）
例如：svn checkout svn://192.168.1.1/pro/domain
简写：svn co

2）往版本库中添加新的文件
svn add file
例如：svn add test.php(添加 test.php)
svn add *.php(添加当前目录下所有的 php 文件)

3）将改动的文件提交到版本库
svn commit -m “LogMessage“ [-N] [--no-unlock] PATH(如果选择了保持锁，就使用－no-unlock 开关)
例如：svn commit -m “add test file for my test“ test.php
简写：svn ci

4）加锁/解锁

svn lock -m “LockMessage“ [--force] PATH

例如：svn lock -m “lock test file“ test.php

svn unlock PATH

5）更新到某个版本

svn update -r m path

例如：

svn update 如果后面没有目录，默认将当前目录以及子目录下的所有文件都更新到最新版本。

svn update -r 200 test.php(将版本库中的文件 test.php 还原到版本 200)

svn update test.php (更新，于版本库同步。如果在提交的时候提示过期的话，是因为冲突，需要先 update，修改文件，然后清除 svn resolved，最后再提交 commit)

简写：svn up

6）查看文件或者目录状态

svn status path（目录下的文件和子目录的状态，正常状态不显示）

【?：不在 svn 的控制中；M：内容被修改；C：发生冲突；A：预定加入到版本库；K：被锁定】

svn status -v path(显示文件和子目录状态)

第一列保持相同，第二列显示工作版本号，第三和第四列显示最后一次修改的版本号和修改人。

注：svn status、svn diff 和 svn revert 这三条命令在没有网络的情况下也可以执行的，原因是 svn 在本地的.svn 中保留了本地版本的原始拷贝。

简写：svn st

7）删除文件

svn delete path -m “delete test fle“

例如：svn delete svn://192.168.1.1/pro/domain/test.php -m “delete test file”

或者直接 svn delete test.php 然后再 svn ci -m ‘delete test file‘，推荐使用这种

简写：svn (del, remove, rm)

8）查看日志

svn log path

例如：svn log test.php 显示这个文件的所有修改记录，及其版本号的变化

9）查看文件详细信息

svn info path

例如：svn info test.php

10）比较差异

svn diff path(将修改的文件与基础版本比较)

例如：svn diff test.php
svn diff -r m:n path(对版本 m 和版本 n 比较差异)
例如：svn diff -r 200:201 test.php
简写：svn di

11）将两个版本之间的差异合并到当前文件
svn merge -r m:n path
例如：svn merge -r 200:205 test.php（将版本 200 与 205 之间的差异合并到当前文件，但是一般都会产生冲突，需要处理一下）

12）SVN 帮助
svn help
svn help ci

13）版本库下的文件和目录列表
svn list path
显示 path 目录下的所有属于版本库的文件和目录
简写：svn ls

14）创建纳入版本控制下的新目录
svn mkdir: 创建纳入版本控制下的新目录。
用法:
mkdir PATH…
每一个以工作副本 PATH 指定的目录，都会创建在本地端，并且加入新增调度，以待下一次的提交。

mkdir URL…
每个以 URL 指定的目录，都会透过立即提交于仓库中创建。
在这两个情况下，所有的中间目录都必须事先存在。

15）恢复本地修改
svn revert: 恢复原始未改变的工作副本文件 (恢复大部份的本地修改)。
revert:
用法: revert PATH…
注意: 本子命令不会存取网络，并且会解除冲突的状况。但是它不会恢复被删除的目录

16）代码库 URL 变更
svn switch (sw): 更新工作副本至不同的 URL。
用法:
switch URL [PATH]
更新工作副本，映射到一个新的 URL，其行为跟“svn update”很像，也会将服务器上文件与本地文件合并。这是将工作副本对应到同一仓库中某个分支或者标记的方法。

switch －relocate FROM TO [PATH...]

改写工作副本的 URL 元数据，以反映单纯的 URL 上的改变。当仓库的根 URL 变动(比如方案名或是主机名称变动)，但是工作副本仍旧对映到同一仓库的同一目录时使用这个命令更新工作副本与仓库的对应关系。

17）解决冲突

svn resolved: 移除工作副本的目录或文件的“冲突”状态。

用法: resolved PATH…

注意: 本子命令不会依语法来解决冲突或是移除冲突标记；它只是移除冲突的相关文件，然后让 PATH 可以再次提交。

18）输出指定文件或 URL 的内容。

svn cat 目标[@版本]…如果指定了版本，将从指定的版本开始查找。

svn cat -r PREV filename > filename (PREV 是上一版本,也可以写具体版本号,这样输出结果是可以提交的)

19）创建版本库

svnadmin create /home/svn/repo

注意，各个子版本库的上层目录需要自己手动创建。

20）开启版本库

svn serve -d -r /home/svn/repo