Android USB通信

随着安卓手机的市场份额逐步增加，安卓系统的功能也日益强大。使用手机与各类设备通信的场景也越来越多，本篇文章将介绍如何使用安卓手机与USB设备进行通信。

1、USB简介

USB是英文Universal Serial Bus（通用串行总线）的缩写，是一个外部总线标准，用于规范电脑与外部设备的连接和通讯。是应用在PC领域的接口技术。现在智能手机也加入了对于USB设备的支持。

安卓手机支持USB accessory模式和USB host模式。USB host模式是手机充当主机，为总线提供电力支持。USB accessory模式正好相反，将手机当作附件把USB设备当作主机。本文中介绍的为Host模式。

2、相关API

class	说明
UsbManager	USB管理器，与连接的USB设备通信。
UsbDevice	USB设备的抽象，每个UsbDevice都代表一个USB设备。
UsbInterface	定义了设备的功能集，一个UsbDevice可能包含一个或多个UsbInterface，每个Interface都是独立的。
–	–
UsbEndpoint	UsbEndpoint是Interface的通信通道。
UsbDeviceConnection	host与device 建立的连接，并在endpoint 传输数据。
–	–
UsbRequest	USB 请求包。
UsbConstants	USB常量的定义

3、配置AndroidManifest.xml文件

在进行USB开发时，需要在AndroidManifest.xml文件中配置相应的属性。配置代码如下：

```
<!--添加权限-->
<uses-feature android:name="android.hardware.usb.host"/>

<activity
            android:name=".MainActivity"
            android:screenOrientation="landscape">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <!-- 如果这里是启动Activity的话，点击USB接入的弹窗会启动该页面 -->
                <category android:name="android.intent.category.LAUNCHER" />
                <action
android:name="android.hardware.usb.action.USB_DEVICE_ATTACHED" />
            </intent-filter>
```

```
            <meta-data
                android:name="android.hardware.usb.action.USB_DEVICE_ATTACHED"
                android:resource="@xml/device_filter" />
</activity>
```

4、过滤设备

在XML资源文件中，声明要过滤的USB设备的元素。通常，如果要过滤特定设备并使用类，子类和协议（如果要过滤一组USB设备（如大容量存储设备或数码相机）），请使用供应商（vendor-id）和产品（product-id）ID，在开发中这些过滤ID一般可以在文档中找到，或者在设备管理器中查看。

将资源文件保存在res/xml/目录中。资源文件名（不带.xml扩展名）必须与您在元素中指定的文件名相同。配置格式如下：

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <usb-device
        class="255"
        product-id="5678"
        protocol="1 "
        subclass="66"
        vendor-id="1234" />
</resources>
```

5、监听USB设备的插拔

Android 系统中，USB 设备的插入和拔出是以系统广播的形式发送的，我们只要注册监听这个广播就好。

注册广播代码如下：

```
public class USBReceiver extends BroadcastReceiver {
    public static final String ACTION_USB_PERMISSION =
"com.android.example.USB_PERMISSION";

    @Override
    public void onReceive(Context context, Intent intent) {
        String action = intent.getAction();
        if (ACTION_USB_PERMISSION.equals(action)) {
            // 获取权限结果的广播
            synchronized (this) {
                UsbDevice device = (UsbDevice)
intent.getParcelableExtra(UsbManager.EXTRA_DEVICE);
                if (device != null) {
                    //call method to set up device communication
                    if
(intent.getBooleanExtra(UsbManager.EXTRA_PERMISSION_GRANTED, false)) {
                        Log.e("USBReceiver", "获取权限成功：" +
device.getDeviceName());
                    } else {
                        Log.e("USBReceiver", "获取权限失败：" +
device.getDeviceName());
                    }
                }
            }
```

```
        }else if (UsbManager.ACTION_USB_DEVICE_ATTACHED.equals(action)) {
            // 有新的设备插入了，在这里一般会判断这个设备是不是我们想要的，是的话就去请求权
限
        } else if (UsbManager.ACTION_USB_DEVICE_DETACHED.equals(action)) {
            // 有设备拔出了
        }
    }
}
```

6、获取UsbManager

UsbManager类是安卓系统提供的用于管理USB设备的类，其中对于USB设备的操作大多数需要调用此类对象中的方法实现。UsbManager类通过获取系统服务的方式获取。

```
usbManager = (UsbManager) context.getSystemService(Context.USB_SERVICE);
```

7、获取目标UsbDevice

UsbDevice标识着搜索到的USB设备，目标UsbDevice设备需要通过pid和vid进行区别。

```
/**
     * @param vendorId  厂商ID
     * @param productId 产品ID
     * @return  device
     */
    public UsbDevice getUsbDevice(int vendorId, int productId) {
        HashMap<String, UsbDevice> deviceList = usbManager.getDeviceList();
        Iterator<UsbDevice> deviceIterator = deviceList.values().iterator();
        while (deviceIterator.hasNext()) {
            UsbDevice device = deviceIterator.next();
            if (device.getVendorId() == vendorId && device.getProductId() ==
productId) {
                Log.e("USBUtil", "getDeviceList: " + device.getDeviceName());
                return device;
            }
        }
        Toast.makeText(context, "没有对应的设备", Toast.LENGTH_SHORT).show();
        return null;
    }
```

8、申请USB设备使用权限

```
 /**
     * 判断对应 USB 设备是否有权限
     */
    public boolean hasPermission(UsbDevice device) {
        return usbManager.hasPermission(device);
    }

    /**
     * 请求获取指定 USB 设备的权限
     */
    public void requestPermission(UsbDevice device) {
        if (device != null) {
```

```
            if (usbManager.hasPermission(device)) {
                Toast.makeText(context, "已经获取到权限",
Toast.LENGTH_SHORT).show();
            } else {
                if (mPermissionIntent != null) {
                    usbManager.requestPermission(device, mPermissionIntent);
                    Toast.makeText(context, "请求USB权限",
Toast.LENGTH_SHORT).show();
                } else {
                    Toast.makeText(context, "请注册USB广播",
Toast.LENGTH_LONG).show();
                }
            }
        }
    }
```

9、USB设备收发数据

9.1打开通信端口

```
 public boolean openPort(UsbDevice device) {
        //获取设备接口，一般只有一个
        usbInterface = device.getInterface(0);

        // 判断是否有权限
        if (hasPermission(device)) {
            // 打开设备，获取 UsbDeviceConnection 对象，连接设备，用于后面的通讯
            usbConnection = usbManager.openDevice(device);

            if (usbConnection == null) {
                return false;
            }
            if (usbConnection.claimInterface(usbInterface, true)) {
                Toast.makeText(Utils.getContext(), "找到 USB 设备接口",
Toast.LENGTH_SHORT).show();
            } else {
                usbConnection.close();
                Toast.makeText(Utils.getContext(), "没有找到 USB 设备接口",
Toast.LENGTH_SHORT).show();
                return false;
            }
        } else {
            Toast.makeText(Utils.getContext(), "没有 USB 权限",
Toast.LENGTH_SHORT).show();
            return false;
        }

        //获取接口上的两个端点，分别对应OUT和IN
        for (int i = 0; i < usbInterface.getEndpointCount(); ++i) {
            UsbEndpoint end = usbInterface.getEndpoint(i);
            if (end.getDirection() == UsbConstants.USB_DIR_IN) {
                usbEndpointIn = end;//获取读入数据的UsbEndpoint
            } else {
                usbEndpointOut = end;//获取发送的数据的UsbEndpoint
            }
        }
```

```
        return true;
    }
```

9.2收发数据

调用UsbDeviceConnection的bulkTransfer方法与USB设备通信，向USB设备发送数据用usbEndpointOut，接受USB设备的数据用usbEndpointIn。 发送数据：

```
int ret = usbDeviceConnection.bulkTransfer(usbEndpointOut, data, data.length,
100);
```

接收数据：USB接收数据需开启数据读取线程。

```
//开线程读取数据
private void startReading() {
    new Thread(new Runnable() {
        @Override
        public void run() {
            while (isReading) {
                synchronized (this) {
                    //创建接收数据的数组
                    byte[] data = new byte[usbEndpointIn.getMaxPacketSize()];
                    //读取数据
                    int ret = usbConnection.bulkTransfer(usbEndpointIn, data,
data.length, 100);
                }
            }
        }
    }).start();
}
```

10、结语

安卓USB通信具有即插即用，可热插拔，具有自动配置能力，用户只要简单地将外设插人到手机就能自动识别和配置USB设备。目前安卓手机、平板都具备USB接口，连接灵活，易扩展。

USB通信速率相对较快，USB2.0理论速度约每秒480Mbps（约每秒60MB），USB3.0的理论速度能够达到每秒5Gbps（约为每秒625MB）。

Android配件协议AOA

AOA协议是Google公司推出的用于实现Android设备与外围设备之间USB通信的协议。该协议拓展了Android设备USB接口的功能，为基于Android系统的智能设备应用于数据采集和设备控制领域提供了条件。介绍了Android系统下USB通信的两种模式，并给出了USB配件模式下基于AOA协议实现Android手机控制步进电机的实例。

1、Android Open Accessory Protocol 1.0（AOA 协议 1.0）

Android USB 配件必须遵从 Android Open Accessory（AOA）协议，该协议定义了配件如何检测和建立与 Android 设备的通信。配件应执行以下步骤：

1. 等待并检测连接的设备
2. 确定设备的配件模式支持
3. 尝试以配件模式下启动设备（如果需要）
4. 如果设备支持 AOA，与设备建立通信

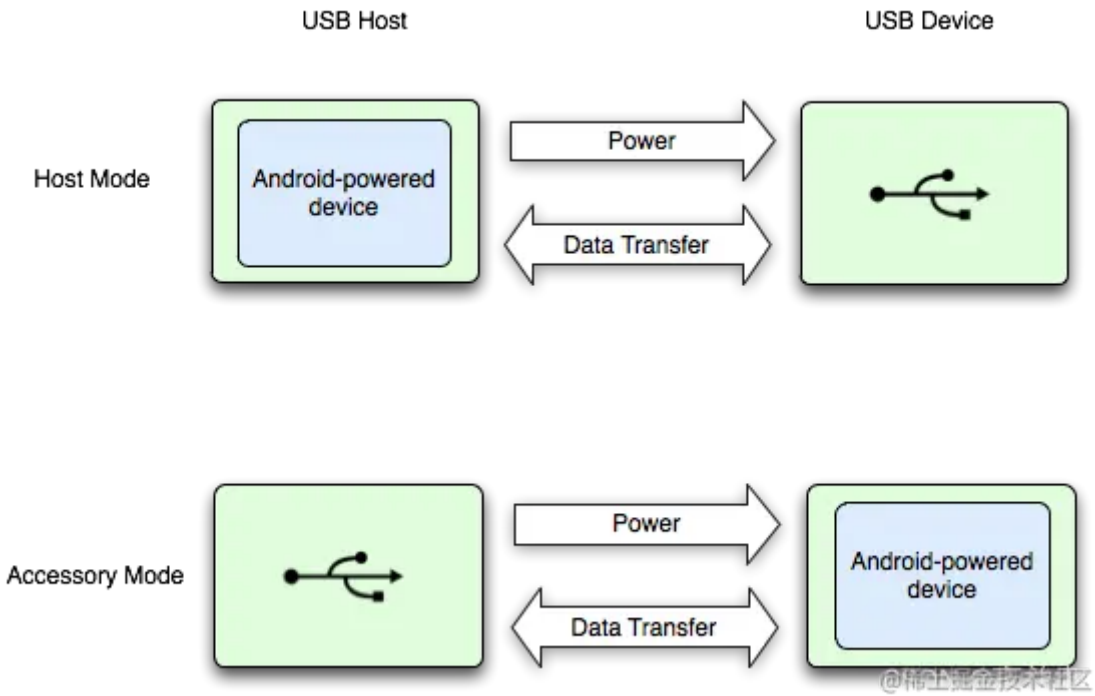
以下部分介绍如何实现这些步骤。

Note：在开发通过 USB 连接到 Android 设备的新配件时，请使用 AOAv2。

2、USB 主机和配件概览

Android 通过 USB 配件和 USB 主机两种模式支持各种 USB 外围设备和 Android USB 配件（实现 Android 配件协议的硬件）。在 USB 配件模式下，外部 USB 硬件充当 USB 主机。配件示例可能包括机器人控制器、扩展坞、诊断和音乐设备、自助服务终端、读卡器等等。这样，不具备主机功能的 Android 设备就能够与 USB 硬件互动。Android USB 配件必须设计为与 Android 设备兼容，并且必须遵守 Android 配件通信协议。在 USB 主机模式下，Android 设备充当主机。设备示例包括数码相机、键盘、鼠标和游戏控制器。针对各类应用和环境设计的 USB 设备仍可与能够与设备正常通信的 Android 应用互动。

下图展示了这两种模式之间的差异。当 Android 设备处于主机模式时，它会充当 USB 主机并为总线供电。当 Android 设备处于 USB 配件模式时，所连接的 USB 硬件（本例中为 Android USB 配件）充当主机并为总线供电。



Android 3.1（API 级别 12）或更高版本的平台直接支持 USB 配件和主机模式。USB 配件模式还作为插件库向后移植到 Android 2.3.4（API 级别 10）中，以支持更广泛的设备。设备制造商可以选择是否在设备的系统映像中添加该插件库。 注意：配件模式取决于设备硬件；部分设备可能不支持配件模式。您可以在相应应用的 Android 清单中使用 元素过滤出支持配件模式的设备。

3、Android 开放配件 (AOA)

Android 开放配件 (AOA) 支持功能可让外部 USB 硬件（Android USB 配件）与处于配件模式下的 Android 设备进行交互。当某台 Android 设备处于配件模式时，所连接的配件会充当 USB 主机（为总线供电并列举设备），而 Android 设备则充当 USB 配件。 Android USB 配件专门用于和 Android 设备相连。这些配件遵循 AOA 要求，从而能够检测到支持配件模式的 Android 设备，并且必须提供 500 毫安（电压为 5 伏）的充电电流。之前发布的部分 Android 设备只能充当 USB 设备，无法发起与外部 USB 设备的连接。AOA 支持功能打破了这一局限，让您能够构建可以与各种 Android 设备建立连接并与其进行交互的配件。

4、AOA 设备握手过程

AOA 协议规定 Android 设备和配件之间握手的大致过程如下：（1）通过 USB 线连接 Android 设备与配件。（2）配件枚举连接过来的设备。根据 VID 和 PID 判断当前 Android 设备是否处于 Accessory Mode。如果否，则配件会向 Android 设备发出切换到 AOA 模式的请求，进入第（3）步； 如果是，则直接调转到第（5）步。

判断 Android 设备处于 Accessory Mode 时，VID 和 PID 值必须满足如下条件如下：

VID	模式
0X18D1	
PID	模式
0x2D00	accessory
0x2D01	accessory + adb
0x2D02	audio
0x2D03	audio + adb
0x2D04	accessory + audio
0x2D05	accessory + audio + adb

（3）配件发送查询指令给 Android 设备。绝大多数 Android 设备，在缺省情况下都不挂载Accessory 驱动，即不会默认处于 Accessory Mode，在配件与 Android 设备建立 USB 连接时，配件会通过握手协议查询该设备是否为Android 设备且是否支持 AOA 协议以及支持的版本号。配件向 endpoint 0 端口中写入 51 号指令，如果返回值为1 或者 2，则说明Android 设备支持 AOA1.0 或者 AOA2.0，如果小于 1 或者大于 2，则说明连接设备不支持AOA协议或者支持的 AOA 协议版本号不正确。

（4） Android 配件发送认证信息给 Android 设备，并发送开始通信的指令。认证信息可以包含以下属性：manufacturer（厂商）、model（型号）、version（版本）、description（描述信息）、URL（Web页面）。

配件也会向Android设备发出53号指令，请求Android设备切换到AOA模式，Android 设备会执行请求，将 USB 切换到 AOA 模式；在这个过程中，USB 连接会出现一次逻辑插拔，配件会重新枚举设备，进入第（2）步。Android 设备接收到配件通过 52 指令发送过来的参数信息以后，使用其中的三个参数 Manufacturer、Model 、Version 来确定绑定到该配件的 App。如果系统内无任何 App 可以匹配配件设备发来的上述三个参数，则 Android 设备会弹出一个对话框，向用户提供 Accessory 设备发送过来的描述信息和 URL 信息，用户可以点击 URL 访问它指向的 Web 页面。 如果系统内有 App 可以匹配 Accessory 设备发来的握手信息，则 Android 系统会弹出一个对话框询问用户是否立刻启动该 App。如果用户选择 OK 则启动该App；同时该对话框提供一个勾选框，勾选之后每次 Accessory 设备连接后会自动启动该 App。应该要求用户勾选该对话框，否则 App 启动后向 USB Manager获取 Accessory 设备后可能因为 Permission 问题无法打开文件描述符建立通讯连接。

（5）建立连接

指令	作用
51	判断设备是否支持 AOA 协议，并获取支持 AOA 协议的版本号。
52	发送配件相关设备认证信息，Android 设备可 以据此来确定绑定到该配件的 APP。
53	请求 Android 设备切换到 Accessory Mode
54~57	HID支持功能
58	音频支持（ Android 8.0 中已被弃用）

AOA 配件端开发

在Android车机或者Android平板开发中，以车机或平台为配件端，与Android手机通过数据线进行通信。 Android 官方是提供 USB 的相关接口用来进行 AOA 连接，主要有UsbManager, UsbDevice, UsbInterface, UsbDeviceConnection 等。具体参考官方文档：USB 主机概览 以下代码段是执行同步数据传输的一种简单方式:

```
private Byte[] bytes;
private static int TIMEOUT = 0;
private boolean forceClaim = true;

...

UsbInterface intf = device.getInterface(0);
UsbEndpoint endpoint = intf.getEndpoint(0);
UsbDeviceConnection connection = usbManager.openDevice(device);
connection.claimInterface(intf, forceClaim);
connection.bulkTransfer(endpoint, bytes, bytes.length, TIMEOUT); //do in another
thread
```

5、AOA 主机端开发

在与Android车机或Android平板USB通信过程中，手机作为主机端，默认不开启配件模式，需要与配件建立AOA通信才会开启启动配件模式。 具体参考Google官方文档：USB 配件模式 与配件通信代码如下：

```
UsbAccessory accessory;
ParcelFileDescriptor fileDescriptor;
FileInputStream inputStream;
FileOutputStream outputStream;
...

private void openAccessory() {
    Log.d(TAG, "openAccessory: " + accessory);
    fileDescriptor = usbManager.openAccessory(accessory);
    if (fileDescriptor != null) {
        FileDescriptor fd = fileDescriptor.getFileDescriptor();
        inputStream = new FileInputStream(fd);
        outputStream = new FileOutputStream(fd);
        Thread thread = new Thread(null, this, "AccessoryThread");
        thread.start();
    }
}
```

6、建立与设备的通信

如果配件检测到处于配件模式的 Android 设备，则配件可以查询设备接口和端点描述符，以获取与设备通信的批量端点。 接口和批量端点的数量取决于产品 ID。 Android 设备有产品 ID：

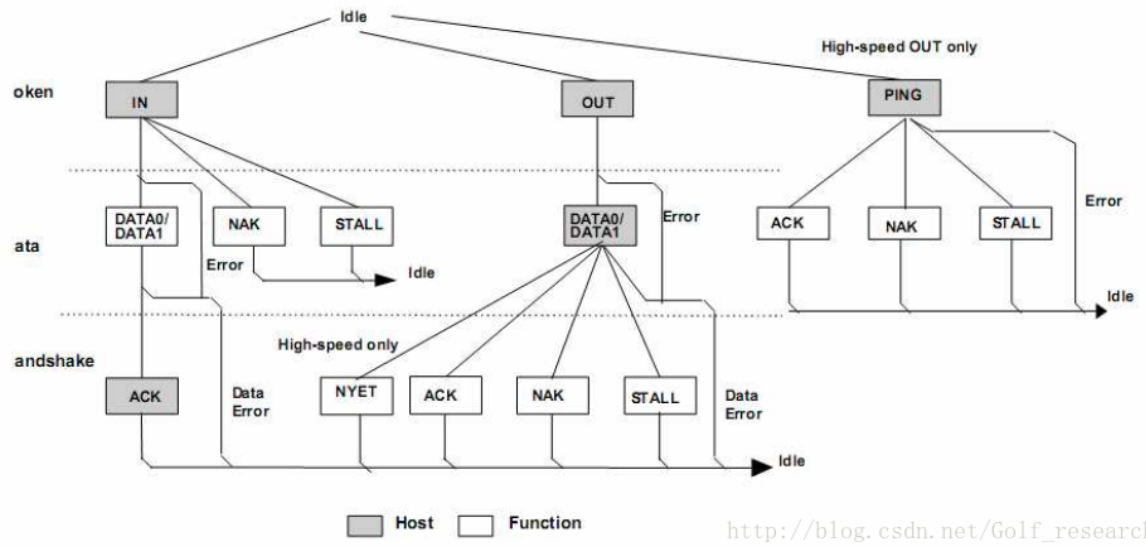
1. 0x2D00 具有一个有两个批量端点的接口，用于输入和输出通信。
2. 0x2D01 具有两个接口，每个接口具有两个批量端点，用于输入和输出通信。 第一个接口处理标准通信，第二个接口处理 ADB 通信。 要使用接口，请找到第一个批量输入和输出端点，使用 SET_CONFIGURATION（0x09）设备请求将设备配置设置为 1，然后使用端点进行通信。

USB的四种传输类型

usb协议中的4中传输类型
usb协议有4中传输类型：批量传输、同步传输、中断传输、控制传输；

1、批量传输

批量传输是一种可靠的单向传输，但延迟没有保证，它尽量利用可以利用的带宽完成传输，时候数据量比较大的传输；低速usb设备不支持批量传输，高速批量端点的最大包长度为512，全速批量端点的最大包长度为8、16、32、64；批量传输在访问usb总线时，相对其他传输类型具有最低的优先级，usb HOST总是优先安排其他类型的传输，当总线带宽有富余时才安排批量传输；高速的批量端点必须支持PING操作，向主机报告端点的状态，ACK表示肯定应答，已经准备好接受下一个数据包；NYET表示否定应答，没有准备好接受下一个数据包；
以下是批量传输的过程图：
图中一个方框代表一个包，灰色的包表示主机发送的包，白色的包表示设备发送的包；



一次正确的批量写传输的过程如下：

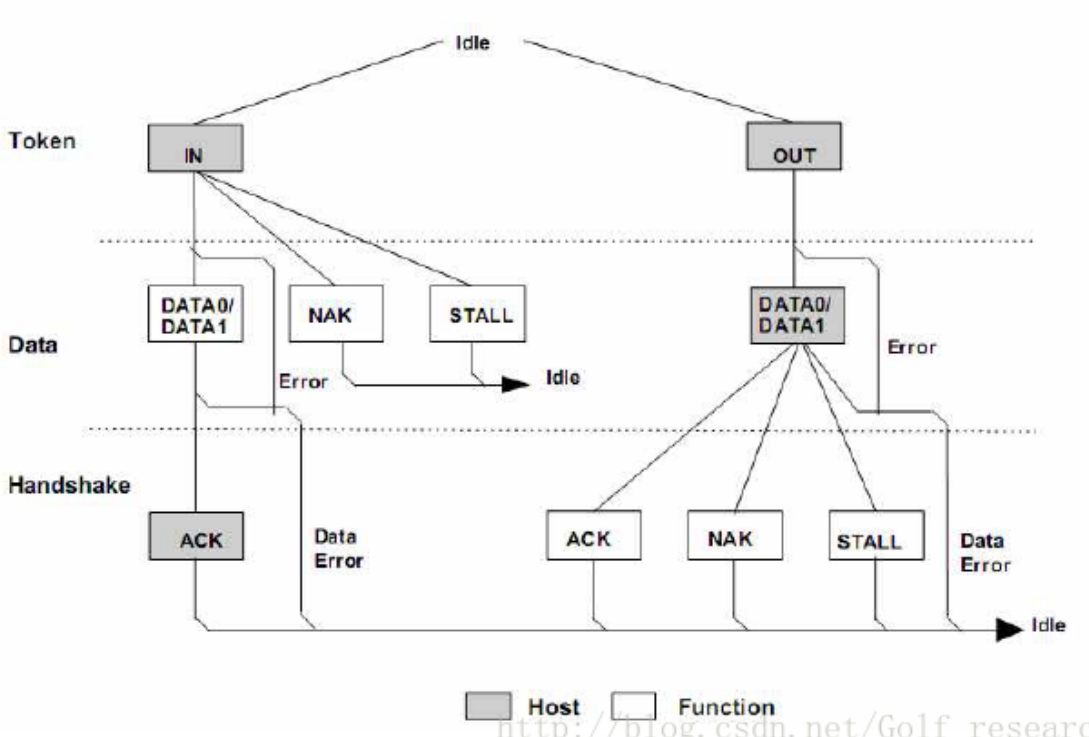
sync	OUT	address	endpoint	crc5	EOP	发送OUT令牌包
sync	DATA	byte0	byte1	.....	byte6	crc16	EOP	发送6个字节数据
sync	ACK	EOP	设备发送ACK给主机

一次正确的批量读传输的过程如下：

sync	IN	address	endpoint	crc5	EOP	发送IN令牌包
sync	DATA	byte0	byte1	.....	byte6	crc16	EOP	设备发送6个字节数据
sync	ACK	EOP	主机发送ACK给设备

2、中断传输

中断传输是一种轮询的传输方式，是一种单向的传输，主机通过固定的间隔对中断端点进行查询，若有数据传输或可以接受数据则返回数据或发送数据，否则返回NAK，表示设备没有准备好；中断传输的延迟没有保证，但并非实时传输，是一种延迟有限的可靠传输，支持错误重传；对于高速、去哪苏、低速端点，最大包长度分别可以达到1024/64/8bytes；中断传输有较高的优先级，仅次于同步传输；
以下是中断传输的过程图：



一次正确的中断写传输过程图：

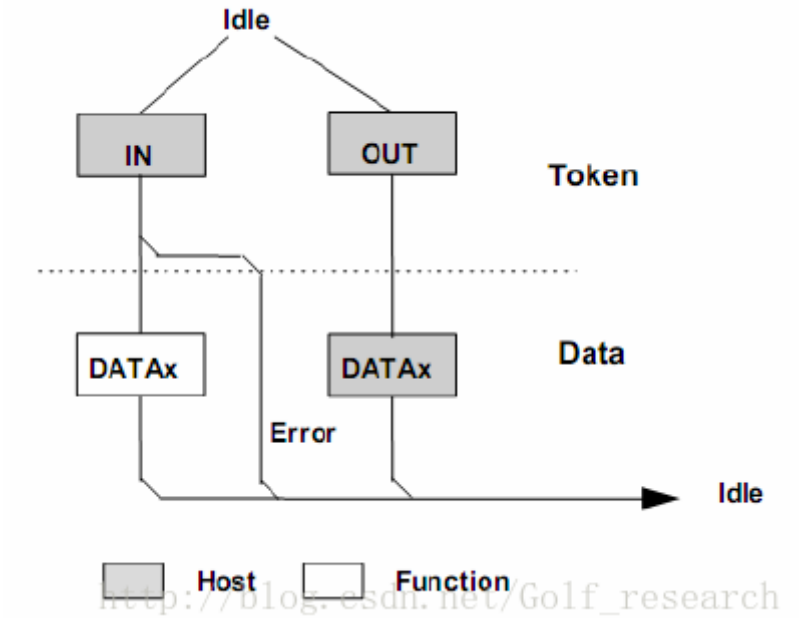
sync	OUT	address	endpoint	crc5	EOP	发送OUT令牌包
sync	DATA	byte0	byte1	.....	byte6	crc16	EOP	发送6个字节数据
sync	ACK	EOP	设备发送ACK给主机

一次正确的中断读传输过程图：

sync	IN	address	endpoint	crc5	EOP	发送IN令牌包
sync	DATA	byte0	byte1	.....	byte6	crc16	EOP	设备发送6个字节数据
sync	ACK	EOP	主机发送ACK给设备

3、同步传输

同步传输是一种实时的、不可靠的传输，不支持错误重发机制，没有握手包；只有高速和全速端点支持同步传输，高速同步端点的最大包长为1024，低速的未1023；同步传输具有最高的优先级；用于对可靠性要求不高的实时数据传输，如摄像头、USB音响；
同步传输过程图：



同步写传输过程：

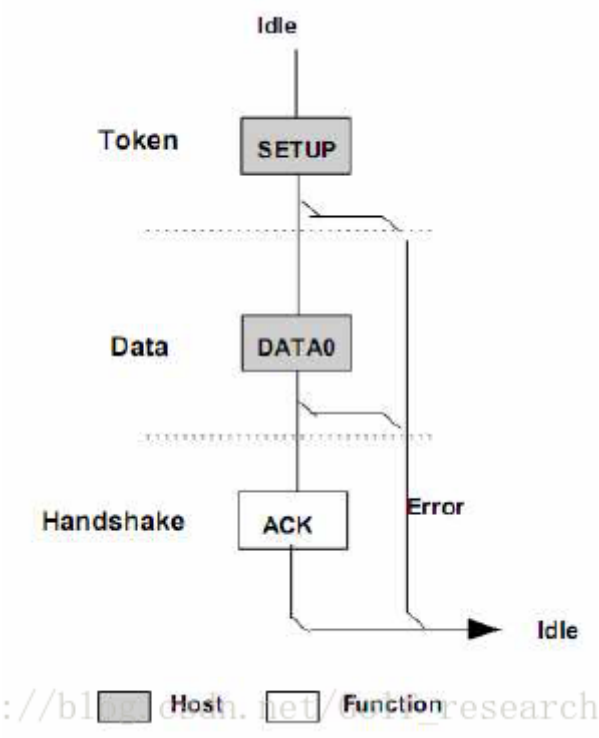
sync	OUT	address	endpoint	crc5	EOP	发送IN令牌包
sync	DATA	byte0	byte1	.....	byte6	crc16	EOP	设备发送6个字节数据

同步读传输过程：

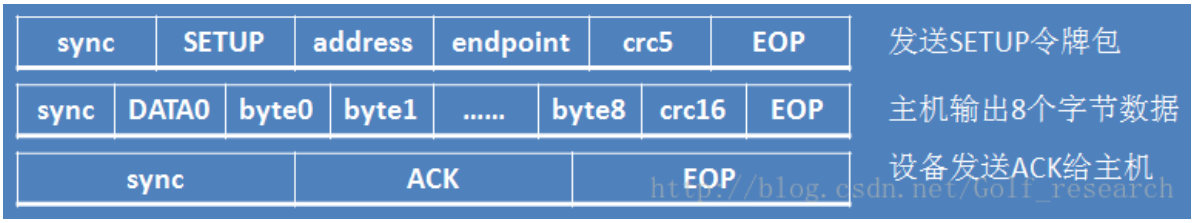
sync	IN	address	endpoint	crc5	EOP	发送IN令牌包
sync	DATA	byte0	byte1	.....	byte6	crc16	EOP	设备发送6个字节数据

4、控制传输

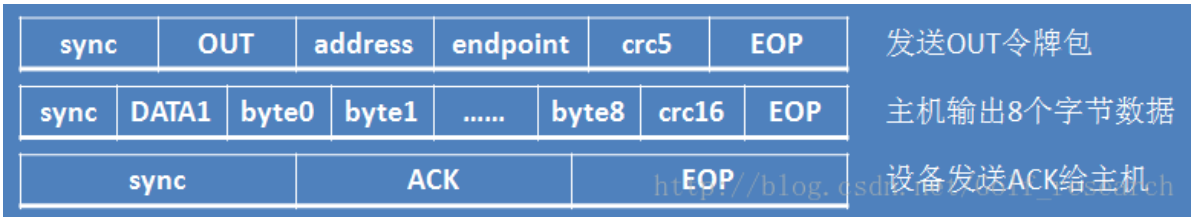
控制传输分为三个过程：第一个过程是建立过程、第二个过程是可选的建立过程、第三个过程是状态过程；
建立过程是一个建立事务，建建立事务的结构是：SETUP令牌包+DATA0数据包+ACK握手包；建立过程的流程图如下：



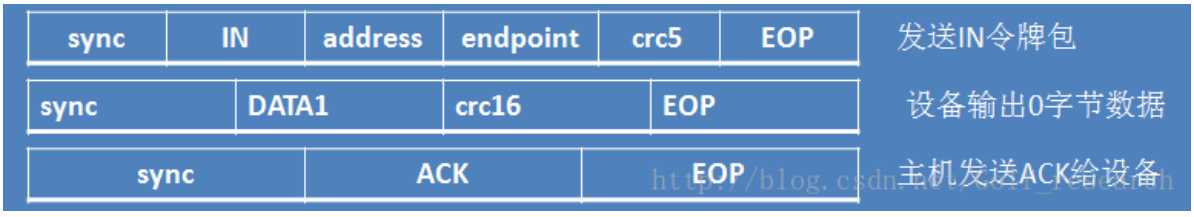
数据过程是一笔或者多笔数据事务，数据事务和批量传输中的批量事务是一样的，不过，数据过程中的所有数据事务的传输方向都是一致的，一旦数据传输方向发生变化，就会认为进入到状态过程，数据过程中第一个数据事务的数据包类型必须是DATA1；
状态过程是一次批量事务，传输方向和数据阶段的传输方向相反，状态过程志使用DATA1数据包格式；
控制写传输过程图：
建立过程



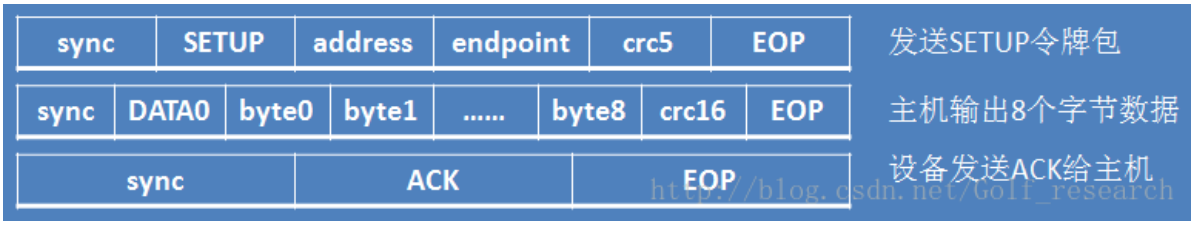
数据过程：一笔或者多笔写数据事务



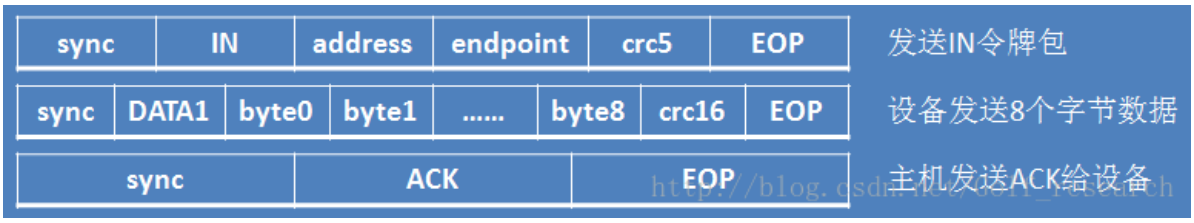
状态过程



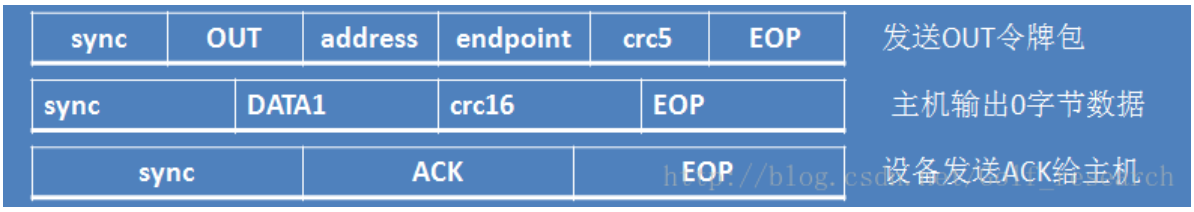
控制读传输过程图：
建立过程：



数据过程：一笔或者多笔读数据事务

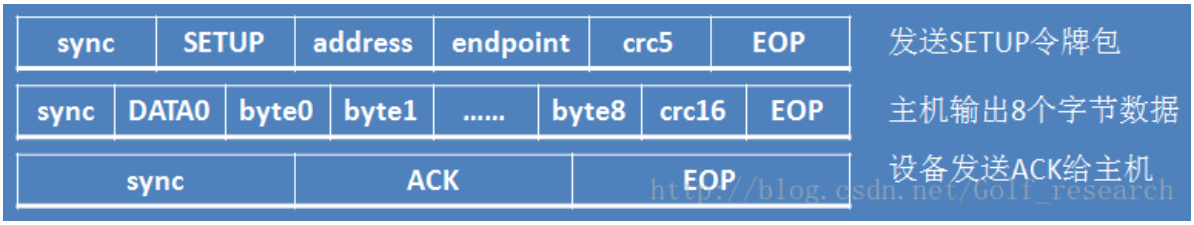


状态过程

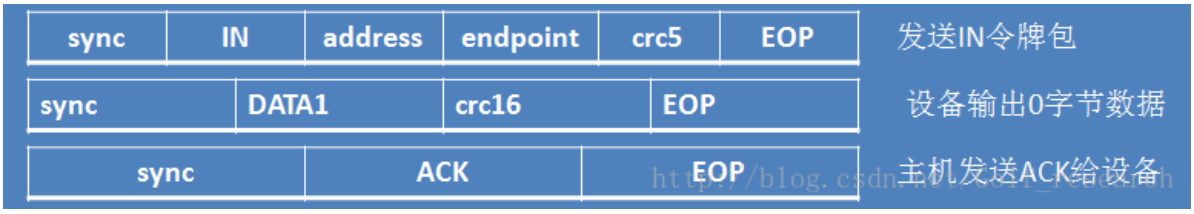


无数据控制写过程图：

建立过程



状态过程



USB结构

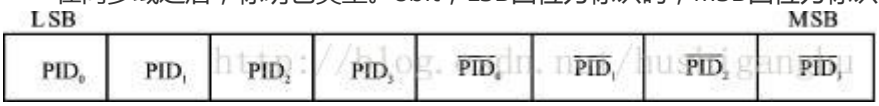
1、域

1.1同步序列域（SYNC）：

位于一个包的最开始处，8bit，固定0000 0001

1.2标识域（PID）：

在同步域之后，标明包类型。8bit，LSB四位为标识码，MSB四位为标识码的反码。类型见下图。

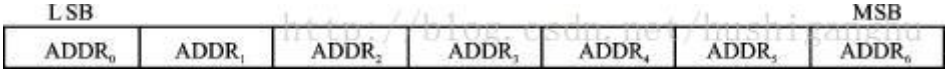


USB 协议1 .1 中, 定义了10 种不同类型的包。USB 协议2 .0 中则定义了全部16 种标识域。如下图所示，标橙色为2.0所有。

数据包类型	name	标识符PID<3:0>	Description	标识域
令牌包	OUT	0001B	启动一个主机到设备的传输，并包含地址加端点号	0x87
	IN	1001B	启动一个设备到主机的传输，并包含地址加端点号	0x96
	SOF	0101B	一个帧的开始并包含帧号	0xA5
	SETUP	1101B	启动一个设置事务，并包含地址和端点号	0xB4
数据包	DATA 0	0011B	偶 数据包	0xC3
	DATA 1	1011B	奇 数据包	0xD2
	DATA2	0111B	高速、高带宽同步 数据包	0xE1
	MDATA	1111B	高速、高带宽同步 数据包	0xF0
握手包	ACK	0010B	接收到没有错误的数据包	0x4B
	NAK	1010B	接收端无法接收数据或发送端无法发送数据	0x5A
	STALL	1110B	端点被禁止或不支持控制管道请求	0x78
	NYET	0110B	从接收机没有反应	0x69
特殊包	PRE	1100B	用于启动下行端口的低速设备的数据传输	0x3C
	ERR	1100B	(Handshake) Split Transaction Error Handshake (reuses PRE value)	0x3C
	SPLIT	1000B	(Token) High-speed Split Transaction Token	0x1E
	PING	0100B	(Token) High-speed flow control probe for a bulk/control endpoint	0x2D
	Reserved	0000B	Reserved	0x0F

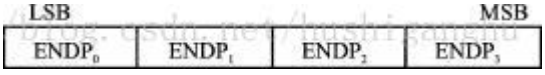
1.3地址域（ADDR）

由主机分配唯一地址，7bit，由于地址0保留用作初始化，因此地址最多有127个，也是usb结构中最大127个设备的原因。



1.4端点域（ENDP）：

4bit，端点0必须作为控制端点，端点可被定义为IN、SETUP、OUT、PING端点，所有设备必须支持端点0作为默认的控制管道。低速设备最大支持3个管道，1个0端点和2个其他端点（可以是都是控制，一个控制一个中断也可以是两个中断）。高速全速支持最大16个输入输出管道。

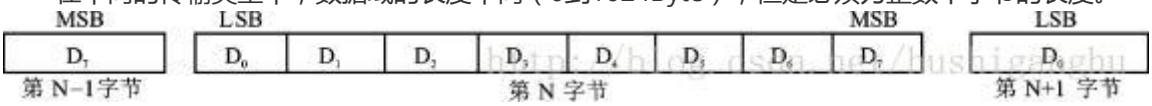


1.5帧号域（FRAM）

11bit，每个帧都有一个特定帧号，帧号域最大容量为0x7FF

1.6数据域（DATA）

在不同的传输类型中，数据域的长度不同（0到1024Byte），但是必须为整数个字节的长度。



1.7校验域（CRC）

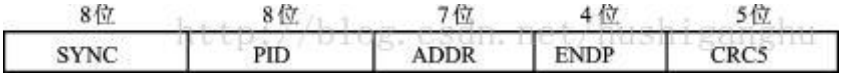
对令牌包和数据包中非PID域进行校验，令牌包使用CRC5（5bit），数据包使用CRC16（16bit）。

2包（Packet）

包(Packet)是最基本的USB 的数据单元, 由一系列的域组成。如前PID域所述, USB 中定义了4 种类型的包, 即令牌包、数据包、握手包和特殊包。
包是USB总线上数据传输的最小单位，不能被打断或干扰，否则会引发错误。若干个数据包组成一次事务传输，一次事务传输也不能打断，属于一次事务传输的几个包必须连续，不能跨帧完成。一次传输由一次到多次事务传输构成，可以跨帧完成。

2.1令牌包(Token Packet)

IN、OUT、SETUP包结构是一样的，如下图



帧起始包机构，如下图
- frame：全速设备一帧1.00 ms ±0.0005 ms
- microframe：高速设备一帧125 μs ±0.0625 μs。

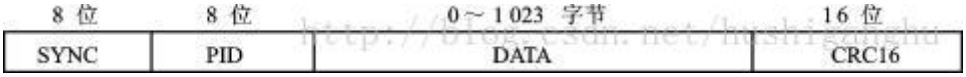
帧的起始由一个特定的包（SOF 包）表示，所有高速和全速还有hub都会收到SOF包，但是不会引起任何作用

split传输特殊令牌包（Split Token）

在主机控制器和 HUB之间执行，通过分离传输，可以允许全速/低速设备连接到高速主机。分为start-split transaction (SSPLIT)和complete-split transaction (CSPLIT)。具体可看USB2.0协议8.4.2。

2.2数据包（Data Packets）

USB 发送数据包的时候, 如果一次发送数据长度大于端点容量时候, 就要把数据分成几个包, 分批发送。如果第一个数据包被确定为DATA0 , 那么第二个发送的数据包就应该是DATA1……如此交替下去。而数据的接收方在接收数据时检查其类型是否是DA TA0、DATA1 交替的, 这是保证数据交换正确的机制之一。
但是当处于同步传输中，所有的数据包都是DATA0。



所有的四个数据包用于高带宽高速同步端点，三个数据包(MDATA DATA0,DATA1)用于split事务传输。
所有数据包都是整数发送，CRC校验不包括PID域。允许数据包最大大小为低速8Byte，全速1023Byte，高速1024Byte。

2.3握手包（Handshake Packets）

2.3.1ACK

- 标识域PID 被正确接收;
- 并且没有发生数据位错误;
- 没有发生数据域的CRC 校验错误等。
  一般由接收到数据的一方发出，主机和设备都可以发送ACK。

2.3.2NACK

- 在接到主机发来的OU T 命令后, 设备无法接收数据;
- 接到主机的IN 命令, 但是设备没有数据发送给主机。
  NAK 只能由设备发送, 而不能由主机发送。

2.3.3STALL

- 设备无法发送数据;
- 设备无法接收数据;
- 不支持某一种控制管道的命令
  STALL 也只能是由设备来发送，NAK 是USB 的一种暂时状态, 当设备处于“ 忙”的工作状态时, 就会发送NAK。等到设备处于空闲状态。这时, 如果主机再要求设备发送或接收数据时, 设备不再返回NAK, 而是进行正常的数据传输。而STALL 与NAK 不同, STALL所表示的传送失败的意义更加严重。当在控制传输中出现STALL 后, 主机就会丢弃所有发给设备的控制命令, 而且要重新进行控制传输后, 才能停止STALL 状态。

2.3.4NYET

2.3.5ERR

3事务

事务分为IN，OUT，SETUP3个事务。每个事务由令牌包，数据包，握手包3个阶段构成。事务3个阶段如下：

1）令牌包阶段：启动一个输入，输出或设置的事务。

2）数据包阶段：按输入，输出发送相应的数据。

3）握手包阶段：返回数据接收情况，在同步传输的IN事务和OUT事务中没有这个阶段。

3.1 事务的3种类型

3.1.1IN事务：

令牌包阶段–主机发送一个PID为IN的输入包给设备，通知设备要往主机发送数据。

数据包阶段–设备根据情况会做出3种反应：

a. 设备端点正常，设备往主机内发出数据，DATA0和DATA1交替发送。

b. 设备正在繁忙，无法往主机发出数据包，此时发送NAK无效包，IN事务提前结束，到了下一个IN事务才继续。

c. 相应设备端点被禁止，此时发送STALL错误包，事务提前结束，总线进入空闲状态。

握手包阶段–主机在正确接收到数据后，就会向设备发送ACK包。

3.1.2OUT事务：

令牌包阶段-主机发送一个PID为OUT的输出包给设备，通知设备要接收主机数据。
数据包阶段-主机交替发送数据DATA0和DATA1。
握手包阶段-设备根据情况做出3种反应：
a. 设备端点接收正确，设备向主机返回ACK包，通知主机可以发送新的数据，如果数据包发送了CRC校验数据，将不返回任何握手信息。
b. 设备正在忙碌，无法从主机接收数据包就发送NAK无效包，通知主机再次发送数据。
c. 相应设备端点被禁止，发送错误STALL包，事务提前结束，总线直接进入空闲状态。

3.1.3SETUP事务：

令牌包阶段-主机发送一个PID为SETUP的输出包给设备，通知设备要接收数据。
数据包阶段-主机设备发送数据，注意这里只有一个固定为8个字节的DATA0包，其中就是标准的USB设备请求指令。
握手包阶段-设备接收到主机的命令信息后，返回ACK包，此后总线进入空闲状态，并准备下一个传输。一般来说，在SETUP事务后面通常是一个IN或OUT事务构成的传输。

libusb交叉编译与使用

在开发板上如果想要显示jpeg格式的图片，必须用到libjpeg库，不可能自己去编写jpg的解码代码。

libjpeg是一个完全用C语言编写的库，包含了被广泛使用的JPEG解码、JPEG编码和其他的JPEG功能的实现。这个库由独立JPEG工作组维护。

1、安装编译步骤

下面介绍libjpeg库交叉编译器的详细步骤。

```
① 下载源码包，将源码包拷贝到linux系统下。比如：jpegsrc.v9b.tar.gz

② 解码源码包
[root@xiaolong jpeg-9b]# tar xf jpegsrc.v9b.tar.gz

③ 配置源码
[root@xiaolong jpeg-9b]#
./configure --prefix=/usr/local/lib CC=arm-linux-gcc --host=arm-linux --enable-
shared --enable-static
注意：
/usr/local/lib 表示指定源码最终安装的路径。

④ 编译源码
[root@xiaolong jpeg-9b]# make

⑤ 安装源码
[root@xiaolong jpeg-9b]# make install

安装好的目录如下：（/usr/local/lib）
[root@xiaolong lib]# ls
bin  include  lib  share

文件结构：
[root@xiaolong lib]# pwd
/usr/local/lib
[root@xiaolong lib]# tree ./
```

```
./
├── bin
│   ├── cjpeg
│   ├── djpeg
│   ├── jpegtran
│   ├── rdjpgcom
│   └── wrjpgcom
├── include
│   ├── jconfig.h
│   ├── jerror.h
│   ├── jmorecfg.h
│   └── jpeglib.h
├── lib
│   ├── libjpeg.a
│   ├── libjpeg.la
│   ├── libjpeg.so -> libjpeg.so.9.2.0
│   ├── libjpeg.so.9 -> libjpeg.so.9.2.0
│   └── libjpeg.so.9.2.0
└── share
    └── man
        └── man1
            ├── cjpeg.1
            ├── djpeg.1
            ├── jpegtran.1
            ├── rdjpgcom.1
            └── wrjpgcom.1

6 directories, 19 files
```

2、使用步骤

```
1. 将以下几个头文件拷贝到需要编译的工程目录下：
jmorecfg.h、jpeglib.h、jerror.h、jconfig.h

2. 将以下头文件加到工程中：
#include "jpeglib.h"

3. /将usr/local/lib目录下的生成的库文件拷贝到开发板的lib目录下。

4. 编译选择--任意一种：
arm-linux-gcc -o app show_jpeg.c -L/usr/local/lib
arm-linux-gcc -o app show_jpeg.c -l:libjpeg.so.9
arm-linux-gcc show_jpeg.c -ljpeg -static -o app

show_jpeg.c是要编译的源文件
app 是生成的目标文件。
 -static 表示静态生成
#include <jpeglib.h>头文件定义解压缩使用的数据结构信息。
```

3、使用案例

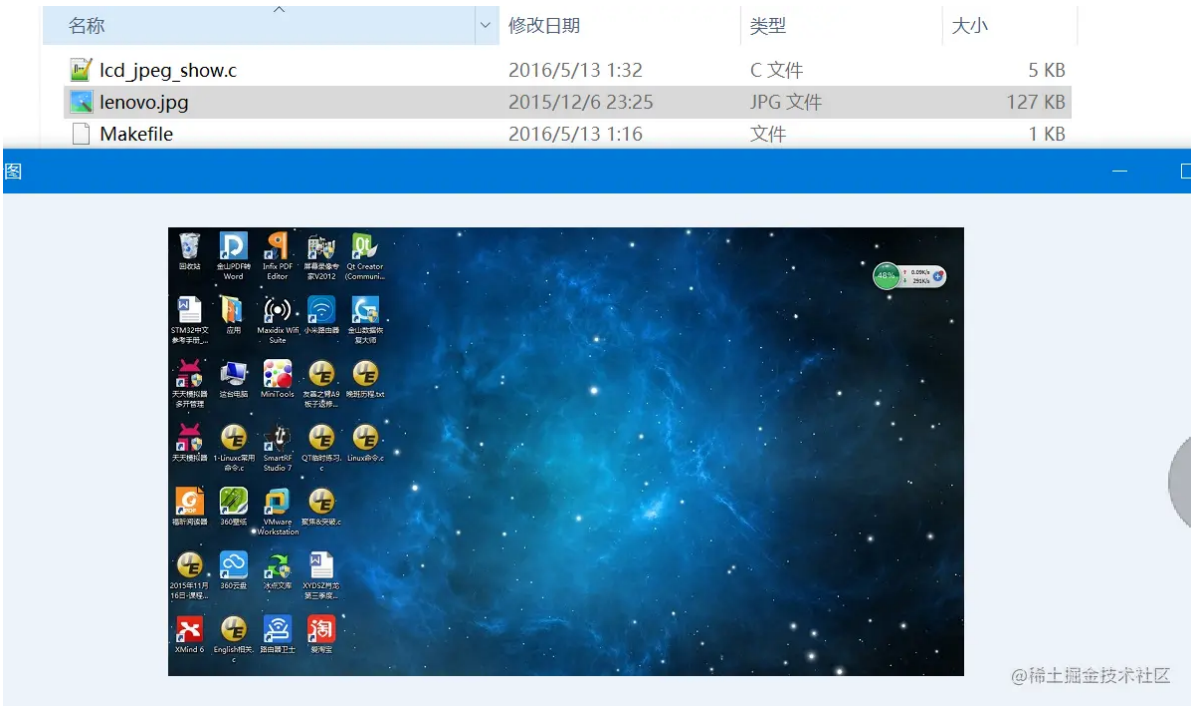
3.1 使用libjpg库编码-RGB数据保存为jpg图片

下面这个是利用libjpeg封装的一个方便函数，用于将传入的rgb数据压缩编码成jpg文件保存，一般用与屏幕截屏、相机拍照等地方。

```
#include <jpeglib.h>
#define JPEG_QUALITY 100 //图片质量

int savejpg(uchar *pdata, char *jpg_file, int width, int height)
{  //分别为RGB数据，要保存的jpg文件名，图片长宽
    int depth = 3;
    JSAMPROW row_pointer[1];//指向一行图像数据的指针
    struct jpeg_compress_struct cinfo;
    struct jpeg_error_mgr jerr;
    FILE *outfile;

    cinfo.err = jpeg_std_error(&jerr);//要首先初始化错误信息
    //* Now we can initialize the JPEG compression object.
    jpeg_create_compress(&cinfo);

    if ((outfile = fopen(jpg_file, "wb")) == NULL)
    {
        fprintf(stderr, "can't open %s\n", jpg_file);
        return -1;
    }
    jpeg_stdio_dest(&cinfo, outfile);

    cinfo.image_width = width;              //* image width and height, in
pixels
    cinfo.image_height = height;
    cinfo.input_components = depth;     //* # of color components per pixel
    cinfo.in_color_space = JCS_RGB;      //* colorspace of input image
    jpeg_set_defaults(&cinfo);

    jpeg_set_quality(&cinfo, JPEG_QUALITY, TRUE ); //* limit to baseline-JPEG
values
    jpeg_start_compress(&cinfo, TRUE);

    int row_stride = width * 3;
    while (cinfo.next_scanline < cinfo.image_height)
    {
            row_pointer[0] = (JSAMPROW)(pdata + cinfo.next_scanline *
row_stride);//一行一行数据的传，jpeg为大端数据格式
            jpeg_write_scanlines(&cinfo, row_pointer, 1);
    }

    jpeg_finish_compress(&cinfo);
    jpeg_destroy_compress(&cinfo);//这几个函数都是固定流程
    fclose(outfile);
    return 0;
}
```

3.2 LCD显示jpg格式图片

下面代码利用libjpeg库解码传入的jpg文件，得到rgb数据，再绘制到LCD屏上显示。




```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <sys/ioctl.h>
#include <linux/fb.h>
#include <sys/mman.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <stdlib.h>
#include <jpeglib.h>
#include <jerror.h>

// 24位色和16位色转换宏
// by cheungmine
#define RGB888_TO_RGB565(r,g,b)  ((WORD)(((WORD(r)<<8)&0xF800)|((WORD(g)
<<3)&0x7E0)|((WORD(b) >> 3))))
#define RGB_TO_RGB565(rgb)       ((WORD)(((((WORD)((rgb)>>3))&(0x1F))<<11)|
((((WORD)((rgb)>>10))&(0x3F))<<5)|(((WORD)((rgb)>>19))&(0x1F))))
#define RGB888_TO_RGB555(r,g,b)  ((WORD)(((WORD(r)<<7)&0x7C00)|((WORD(g)
<<2)&0x3E0)|((WORD(b)>>3))))
#define RGB_TO_RGB555(rgb)       ((WORD)(((((WORD)((rgb)>>3))&(0x1F))<<10)|
((((WORD)((rgb)>>11))&(0x1F))<<5)|(((WORD)((rgb)>>19))&(0x1F))))
#define RGB555_TO_RGB(rgb555)    ((DWORD)(((BYTE)(((rgb555)>>7)&0xF8)|((WORD)
((BYTE)(((rgb555)>>2)&0xF8))<<8))|(((DWORD)(BYTE)(((rgb555)<<3)&0xF8))<<16)))
#define RGB565_TO_RGB(rgb565)    ((DWORD)(((BYTE)((((rgb565)&0xF800)>>11)<<3)|
((WORD)((BYTE)((((rgb565)&0x07E0)>>5)<<2))<<8))|(((DWORD)(BYTE)
(((rgb565)&0x001F)<<3))<<16)))
unsigned short  rgb888_to_rgb555(unsigned char red,unsigned char green,unsigned
char blue);
```

```
 unsigned short  rgb888_to_rgb565(unsigned char red,unsigned char green,unsigned char blue);


 /*-------------------------------------------------------------
                          JPEG图片显示
 -------------------------------------------------------------*/
 static unsigned char *fbmem = NULL;
 static struct fb_var_screeninfo var;//定义可变参数结构体来接收驱动传过来的可变参数结构体
 static struct fb_fix_screeninfo fix;//定义固定参数结构体来接收驱动传过来的固定参

 //显示JPEG
 int show_jpeg(unsigned char *file)
 {
     struct jpeg_decompress_struct cinfo; //存放图像的数据
     struct jpeg_error_mgr jerr; //存放错误信息
     FILE           *infile;
     unsigned int *dst=fbmem;
     unsigned char  *buffer;
     unsigned int    x;
     unsigned int    y;
     /*
     * 打开图像文件
     */
     if ((infile = fopen(file, "rb")) == NULL) {
         fprintf(stderr, "open %s failed\n", file);
         exit(-1);
     }

     /*
      * init jpeg压缩对象错误处理程序
      */
     cinfo.err = jpeg_std_error(&jerr); //初始化标准错误，用来存放错误信息
     jpeg_create_decompress(&cinfo);    //创建解压缩结构信息


     /*
      * 将jpeg压缩对象绑定到infile
      */
     jpeg_stdio_src(&cinfo, infile);

     /*
      * 读jpeg头
      */
     jpeg_read_header(&cinfo, TRUE);

     /*
         *开始解压
         */
     jpeg_start_decompress(&cinfo);

     printf("JPEG高度: %d\n",cinfo.output_height);
     printf("JPEG宽度: %d\n",cinfo.output_width);
     printf("JPEG颜色位数（字节单位）: %d\n",cinfo.output_components);

     //为一条扫描线上的像素点分配存储空间
```

```
    buffer = (unsigned char *) malloc(cinfo.output_width
*cinfo.output_components);
    y = 0;

    //将图片内容显示到framebuffer上
    while (cinfo.output_scanline < cinfo.output_height)
    {

         //读取一行的数据
        jpeg_read_scanlines(&cinfo, &buffer, 1);

        //判断LCD屏的映射空间像素位数
        if (var.bits_per_pixel == 32)
        {
            unsigned int  color;
            for (x = 0; x < cinfo.output_width; x++) {
                color = buffer[x * 3 + 0] << 16  |
                        buffer[x * 3 + 1] << 8   |
                        buffer[x * 3 + 2] << 0;
                dst = ((unsigned int *) fbmem + y * var.xres + x);
                *dst = color;
            }
        }
        y++;                                    // 显示下一个像素点
    }

    /*
     * 完成解压,摧毁解压对象
     */
    jpeg_finish_decompress(&cinfo); //结束解压
    jpeg_destroy_decompress(&cinfo); //释放结构体占用的空间

    /*
     * 释放内存缓冲区
     */
    free(buffer);

    /*
     * 释放内存缓冲区
     */
    fclose(infile);
    return 0;
}

/*映射LCD显示的内存空间*/
unsigned char * fmem(unsigned char *fbname)
{
    int fb;
    unsigned char *mem;
    fb = open(fbname,2);
    if(fb<0)
    {
        printf("open fbdev is error!!!\n");
        return NULL;
    }
    ioctl(fb,FBIOGET_VSCREENINFO,&var);//获取固定参数结构体放在var结构体中
    ioctl(fb,FBIOGET_FSCREENINFO,&fix);//获取固定参数，存放在fix结构体中
```

```
    mem = (unsigned char
*)mmap(NULL,fix.smem_len,PROT_READ|PROT_WRITE,MAP_SHARED,fb,0);
    if(mem == (unsigned char *)-1)
    {
        printf("fbmmap is error!!!\n");
        munmap(mem,fix.smem_len);
        return NULL;
    }
    return mem;
}


int main (int argc,char** argv) //./a.out /dev/fb0 xxx.bmp
{
    int fb ,i=4;
    char key;
    unsigned char * bmpmem;
    if(argc!=3)
    {
        printf("Usage: ./%s <fbdev>  <bmpname> \n",argv[0]);
        return -1;
    }
    fbmem =  fmem(argv[1]);         //将缓冲设备映射到内存进行写入
    memset(fbmem,0x00,fix.smem_len);//清屏函数 往映射的地址填充fix.sem_len大小的0xff
颜色进去
    show_jpeg(argv[2]);            //程序运行时显示主界面
    return 0;
}
```

串口通信原理

1、串口通信（Serial Communications）的概念非常简单，串口按位（bit）发送和接收字节。尽管比按字节（byte）的并行通信慢，但是串口可以在使用一根线发送数据的同时用另一根线接收数据。

2、它很简单并且能够实现远距离通信。比如IEEE488定义并行通行状态时，规定设备线总长不得超过20米，并且任意两个设备间的长度不得超过2米。

3、而对于串口而言，长度可达1200米。典型地，串口用于ASCII码字符的传输。通信使用3根线完成，分别是地线、发送、接收。

4、由于串口通信是异步的，端口能够在一根线上发送数据同时在另一根线上接收数据。其他线用于握手，但不是必须的。

5、串口通信最重要的参数是波特率、数据位、停止位和奇偶校验。对于两个进行通信的端口，这些参数必须匹配。

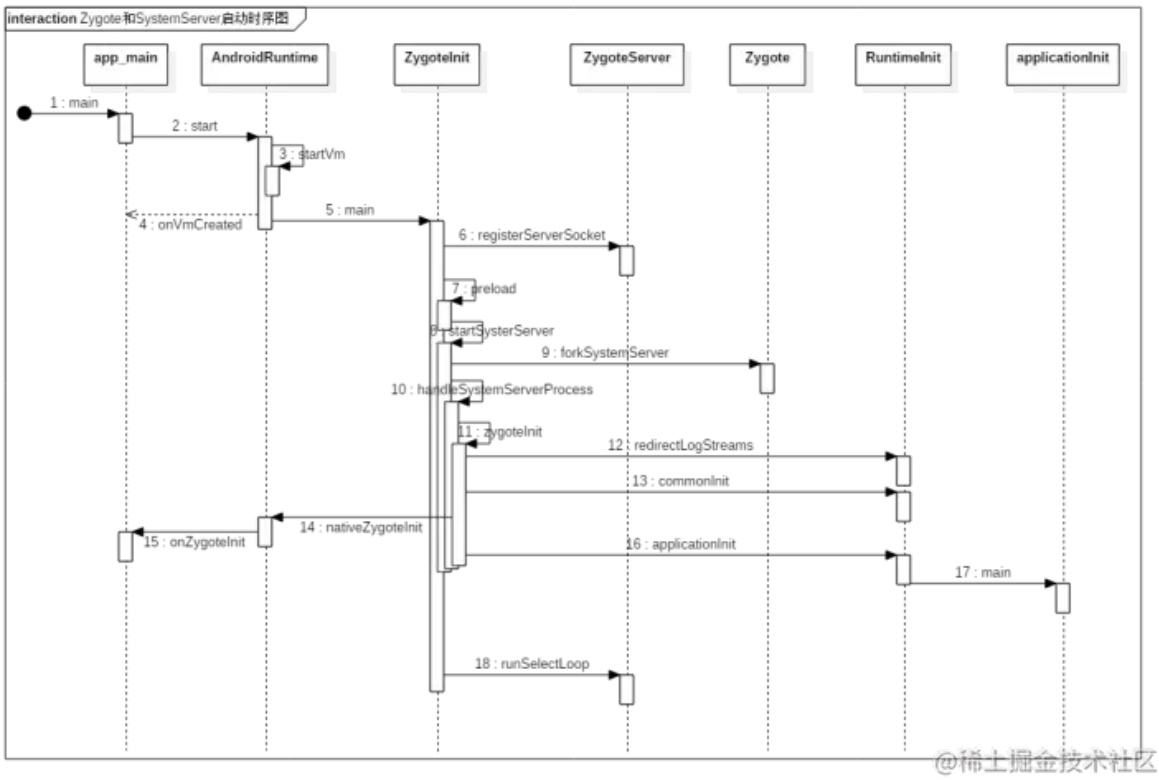
CarFramework框架

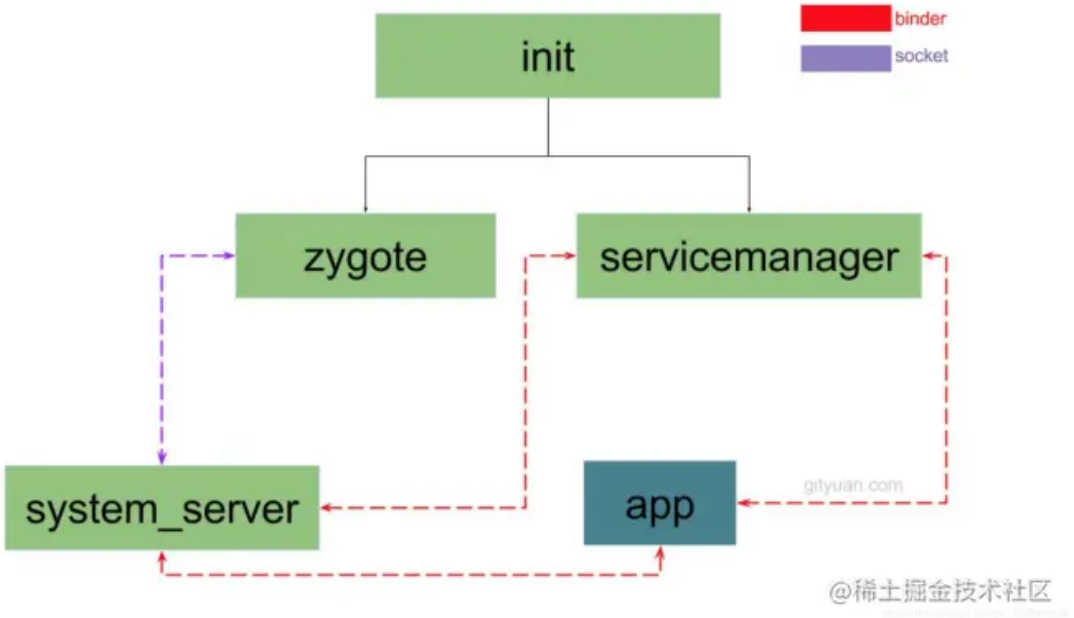
1、CarFramework与Android Framework

在Android系统中，所有的应用程序进程以及系统服务进程SystemServer都是由Zygote进程孕育（fork）出来的，这也许就是为什么要把它称为Zygote（受精卵）的原因吧。由于Zygote进程在Android系统中有着如此重要的地位，本文将详细分析它的启动过程

1.1总体时序

先概述一下总体运行流程，当按电源键，首先是加载系统引导程序BootLoader，然后启动linux内核，再启动init进程，最后Zygote进程启动完成。理论上Android系统中的所有应用程序理论上都是由Zygote启动的。Zygote前期启动启动服务，后期主要fork程序。





1.2init启动流程

- 用户空间的第一个进程，进程号为1（在《深入理解安卓内核思想》的257页里面写的是0，在这记录一下）
- 职责
- 创建Zygote
- 初始化属性服务
- init文件位于源码目录system/core/init中

1.3init进程的启动三个阶段

- 启动电源以及系统的启动，加载引导程序BootLoader。
- 启动Linux内核
- 启动init进程。
- 启动Zygote进程
- 初始化启动属性服务。

1.3Zygote进程

- 所有App的父进程，ZygoteInit.main
- Zygote进程，是由init进程通过解析init.rc文件后fork生成的，Zygote进程主要包括
- 加载Zygoteinit类，注册Zygote Socket服务端套接字
- 加载虚拟机
- 提前加载类PreloadClasses
- 提前加载资源PreLoadResouces
- system_server进程，是由Zygote fork而来，System Server是Zygote孵化出的第一个进程，System Server 负责启动和管理整个Java FrameWork，包含ActivityManagerService，WorkManagerService，PagerManagerService，PowerManagerService等服务

1.4system_server进程

系统各大服务的载体， SystemServer.main system_server进程从源码角度来看可以分为，引导服务，核心服务和其他服务

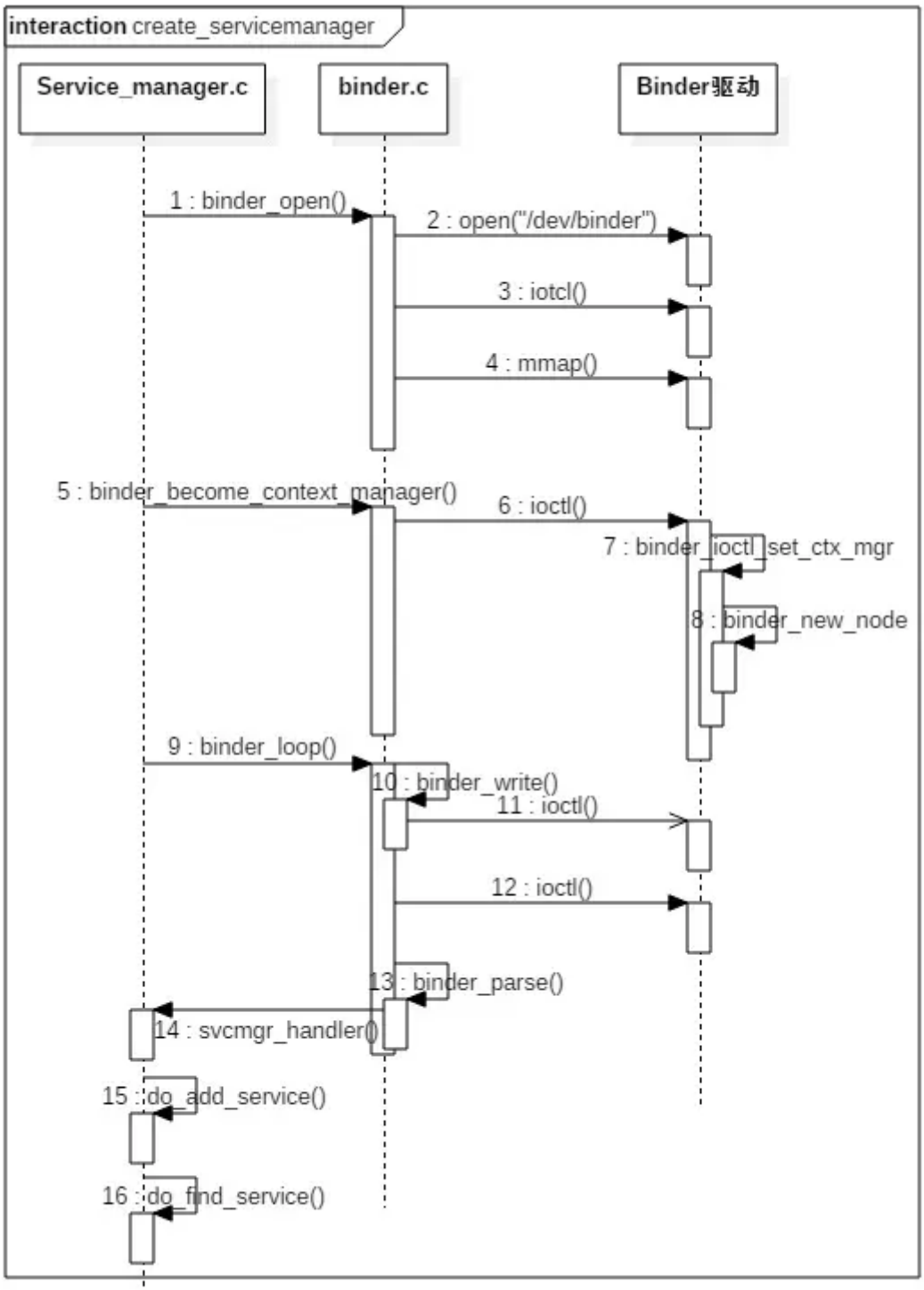
- 引导服务(7个)：ActivityManagerService、PowerManagerService、LightsService、DisplayManagerService、PackageManagerService、UserManagerService、SensorService；
- 核心服务(3个)：BatteryService、UsageStatsService、WebViewUpdateService；
- 其他服务(70个+)：AlarmManagerService、VibratorService等。

1.5ServiceManger进程

bInder服务的大管家

ServiceManager 是Binder IPC通信过程中的守护进程，本身也是一个Binder，但是并没有采用多线程模型来跟Binder通信，而是自行编写了binder.c直接和Binder驱动来通信，并且只有一个binder_loop来读取和处理事务，这样做的好处是简单和高效 ServiceManager本身工作相对简单，其工能查询和注册服务

1.6流程图



ServiceManager 集中管理系统内的所有服务，通能过权限控制进程是否有权注册服务，通过字符串来查找是否有对应的Service，由于ServiceManager进程注册了Service的死亡通知，那么服务所在的进程死亡后，只需告诉ServiceManager，每个Client通过查询ServiceManager可以获取Service的情况

启动主要包括以下几个阶段

- 打开Binder驱动，并调用mmap()方法分配128k的内存映射空间，binder_open
- 注册成为Binder服务的大管家binder_become_context_manager
- 验证selinux权限，判断进程是否有权注册查看指定服务
- 进入无限循环，处理Client发来的请求 binder_loop
- 根据服务的名称注册服务，重复注册会移除之前的注册信息
- 死亡通知，当所在进程死亡后，调用binder_release方法，然后调用binder_node_release，这个过程发出死亡通知回调

1.7App进程

- 通过Process.start启动的App进程ActivityThread.main
- Zygote 孵化出的第一个App进程是Launcher，这是用户看到的桌面App
- Zygote 还会创建出Browser，Phone，Email等App进程，每个App至少运行在一个进程上
- 所有的App进程都是由Zygote fork而成

2、Zygote进程的启动

> Zygote进程, 一个在Android系统中扮演重要角色的进程. 我们知道Android系统中的两个重要服务PackageManagerService和ActivityManagerService, 都是由SystemServer进程启动的, 而这个SystemServer进程本身是Zygote进程在启动的过程中fork出来的. 这样一来, 想必我们就知道Zygote进程在Android系统中的重要地位了.

从图中可得知Android系统中各个进程的先后顺序为：

> init进程 –> Zygote进程 –> SystemServer进程 –>应用进程

链接

1. 在init启动Zygote时主要是调用app_main.cpp的main函数中的AppRuntime.start()方法来启动Zygote进程的；
2. 接着到AndroidRuntime的start函数：使用JNI调用ZygoteInit的main函数，之所以这里要使用JNI，是因为ZygoteInit是java代码。最终，Zygote就从Native层进入了Java FrameWork层。在此之前，并没有任何代码进入Java FrameWork层面，因此可以认为，Zygote开创了java FrameWork层。
3. /frameworks/base/core/java/com/android/internal/os/ZygoteInit.java

```
    @UnsupportedAppUsagepublic static void main(String argv[]) {
    ZygoteServer zygoteServer = null;
    // Mark zygote start. This ensures that thread creation will throw
    // an error.ZygoteHooks.startZygoteNoThreadCreation();
    // Zygote goes into its own process group.try {Os.setpgid(0, 0);}
    catch (ErrnoException ex) {
    throw new RuntimeException("Failed to setpgid(0,0)", ex);
    }Runnable caller;try {
    // Report Zygote start time to tron unless it is a runtime restartif
(!"1".equals(SystemProperties.get("sys.boot_completed"))) {
    MetricsLogger.histogram(null, "boot_zygote_init",(int)
SystemClock.elapsedRealtime());}
    String bootTimeTag = Process.is64Bit() ? "Zygote64Timing" :
"Zygote32Timing";
    TimingsTraceLog bootTimingsTraceLog = new
TimingsTraceLog(bootTimeTag,Trace.TRACE_TAG_DALVIK);
    bootTimingsTraceLog.traceBegin("ZygoteInit");
    RuntimeInit.enableDdms();
    boolean startSystemServer = false;
    String zygoteSocketName = "zygote";
    String abiList = null;
    boolean enableLazyPreload = false;
    for (int i = 1; i < argv.length; i++) {
    if ("start-system-server".equals(argv[i])) {
    startSystemServer = true;}
    else if ("--enable-lazy-preload".equals(argv[i])) {enableLazyPreload = true;
    }
    else if (argv[i].startsWith(ABI_LIST_ARG)) {
```

```
    abiList = argv[i].substring(ABI_LIST_ARG.length());
    }
    else if (argv[i].startsWith(SOCKET_NAME_ARG)) {zygoteSocketName =
argv[i].substring(SOCKET_NAME_ARG.length());
    }
    else {throw new RuntimeException("Unknown command line argument: " +
argv[i]);
    }
    }
    final boolean isPrimaryZygote =
zygoteSocketName.equals(Zygote.PRIMARY_SOCKET_NAME);
    if (abiList == null)
   {
   throw new RuntimeException("No ABI list supplied.");
   }
   // In some configurations, we avoid preloading resources and classes eagerly.
   // In such cases, we will preload things prior to our first fork.if
(!enableLazyPreload)
{bootTimingsTraceLog.traceBegin("ZygotePreload");EventLog.writeEvent(LOG_BOOT_PR
OGRESS_PRELOAD_START,SystemClock.uptimeMillis());preload(bootTimingsTraceLog);Ev
entLog.writeEvent(LOG_BOOT_PROGRESS_PRELOAD_END,SystemClock.uptimeMillis());boot
TimingsTraceLog.traceEnd();
   // ZygotePreload} else {Zygote.resetNicePriority();}
   // Do an initial gc to clean up after
startupbootTimingsTraceLog.traceBegin("PostZygoteInitGC");gcAndFinalize();bootTi
mingsTraceLog.traceEnd();
   // PostZygoteInitGCbootTimingsTraceLog.traceEnd();
   // ZygoteInit// Disable tracing so that forked processes do not inherit stale
tracing tags from
   // Zygote.Trace.setTracingEnabled(false,
0);Zygote.initNativeState(isPrimaryZygote);
   ZygoteHooks.stopZygoteNoThreadCreation();
   zygoteServer = new ZygoteServer(isPrimaryZygote);
   if (startSystemServer) {
// 使用了forkSystemServer()方法去创建SystemServer进程Runnable r =
forkSystemServer(abiList, zygoteSocketName, zygoteServer);
// {@code r == null} in the parent (zygote) process, and {@code r != null} in
the
// child (system_server) process.if (r != null) {r.run();return;
}
}
Log.i(TAG, "Accepting command socket connections");
// The select loop returns early in the child process after a fork and
// 这里调用了ZygoteServer的runSelectLoop方法来等等ActivityManagerService来请求创建新的
应用程序进程
// loops forever in the zygote.caller = zygoteServer.runSelectLoop(abiList);}
catch (Throwable ex) {
Log.e(TAG, "System zygote died with exception", ex);
throw ex;}
finally
{
if (zygoteServer != null) {zygoteServer.closeServerSocket();
}
}
// We're in the child process and have exited the select loop. Proceed to execute
the
// command.if (caller != null) {caller.run();
}
```

```
}
```

其中，在ZygoteInit的forkSystemServer()方法中启动了SystemServer进程，forkSystemServer()方法核心代码 ：

```
private static Runnable forkSystemServer(String abiList, String
socketName,ZygoteServer zygoteServer)
{
// 一系统创建SystemServer进程所需参数的准备工作try {...
/* Request to fork the system server process
*/// 3.1pid = Zygote.forkSystemServer(parsedArgs.uid,
parsedArgs.gid,parsedArgs.gids,parsedArgs.runtimeFlags,null,parsedArgs.permitted
Capabilities,parsedArgs.effectiveCapabilities);
}
catch (IllegalArgumentException ex)
{throw new RuntimeException(ex);}
/* For child process
*/if (pid == 0) {
if (hasSecondZygote(abiList)) {waitForSecondaryZygote(socketName);}
zygoteServer.closeServerSocket();
// 3.2return handleSystemServerProcess(parsedArgs);
}return null;
}
```

可以看到，forkSystemServer()方法中，注释3.1调用了Zygote的forkSystemServer()方法去创建SystemServer进程，其内部会执行nativeForkSystemServer这个Native方法，它最终会使用fork函数在当前进程创建一个SystemServer进程。如果pid等于0，即当前是处于新创建的子进程ServerServer进程中，则在注释3.2处使用handleSystemServerProcess()方法处理SystemServer进程的一些处理工作。

> 从以上的分析可以得知，Zygote进程启动中承担的主要职责如下：

- 1、创建AppRuntime，执行其start方法，启动Zygote进程。。
- 2、创建JVM并为JVM注册JNI方法。
- 3、使用JNI调用ZygoteInit的main函数进入Zygote的Java FrameWork层。
- 4、使用registerZygoteSocket方法创建服务器端Socket，并通过runSelectLoop方法等等AMS的请求去创建新的应用进程。
- 5、启动SystemServer进程。

1. 调用了handleSystemServerprocess()方法来启动SystemServer进程。
   handleSystemServerProcess()方法如下所示：

```
/*** Finish remaining work for the newly forked system server process.
*/
private static Runnable handleSystemServerProcess(ZygoteConnection.Arguments
parsedArgs) {...
if (parsedArgs.invokeWith != null) {...}
else {ClassLoader cl = null;
if (systemServerClasspath != null) {
// 1cl = createPathClassLoader(systemServerClasspath,
parsedArgs.targetSdkVersion);
Thread.currentThread().setContextClassLoader(cl);}
/** Pass the remaining arguments to SystemServer.
*/// 2return ZygoteInit.zygoteInit(parsedArgs.targetSdkVersion,
parsedArgs.remainingArgs, cl);
}
}
```

在注释1处，使用了systemServerClassPath和targetSdkVersion创建了一个PathClassLoader。接着，在注释2处，执行了ZygoteInit的zygoteInit()方法，该方法如下所示：

```
public static final Runnable zygoteInit(int targetSdkVersion, String[] argv,
ClassLoader classLoader)
{
if (RuntimeInit.DEBUG) {
Slog.d(RuntimeInit.TAG, "RuntimeInit: Starting application from zygote");
}
Trace.traceBegin(Trace.TRACE_TAG_ACTIVITY_MANAGER, "ZygoteInit");
RuntimeInit.redirectLogStreams();
RuntimeInit.commonInit();
// 1ZygoteInit.nativeZygoteInit();
// 2return RuntimeInit.applicationInit(targetSdkVersion, argv, classLoader);
}
```

1. zygoteInit()方法的注释2处，这里调用了RuntimeInit 的 applicationInit() 方法，代码如下所示：

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

```
protected static Runnable applicationInit(int targetSdkVersion, String[]
argv,ClassLoader classLoader) {...
// Remaining arguments are passed to the start class's static mainreturn
findStaticMain(args.startClass, args.startArgs, classLoader);
}
```

在applicationInit()方法中最后调用了findStaticMain()方法：

```
protected static Runnable findStaticMain(String className, String[]
argv,ClassLoader classLoader) {
Class<?> cl;
try {
// 1cl = Class.forName(className, true, classLoader);
}
catch (ClassNotFoundException ex) {
throw new RuntimeException("Missing class when invoking static main " +
className,ex);
}
Method m;try {
// 2m = cl.getMethod("main", new Class[] { String[].class });
}
catch (NoSuchMethodException ex) {
throw new RuntimeException("Missing static main on " + className, ex);}
catch (SecurityException ex) {throw new RuntimeException("Problem getting static
main on " + className, ex);}
int modifiers = m.getModifiers();
if (! (Modifier.isStatic(modifiers) && Modifier.isPublic(modifiers))) {
throw new RuntimeException("Main method is not public and static on " +
className);}
/** 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.
*/// 3return new MethodAndArgsCaller(m, argv);
}
```

首先，在注释1处，通过发射得到了SystemServer类。接着，在注释2处，找到了SystemServer中的main()方法。最后，在注释3处，会将main()方法传入MethodAndArgsCaller()方法中，这里的MethodAndArgsCaller()方法是一个Runnable实例，它最终会一直返回出去，直到在ZygoteInit的main()方法中被使用，如下所示：

```
if (startSystemServer) {
Runnable r = forkSystemServer(abiList, socketName, zygoteServer);
// {@code r == null}
in the parent (zygote) process, and {
@code r != null}
in the
// child (system_server) process.
if (r != null) {
r.run();
return;
}
}
```

可以看到，最终直接调用了这个Runnable实例的run()方法，代码如下所示：

```
/*** Helper class which holds a method and arguments and can call them. This is
used as part of
* a trampoline to get rid of the initial process setup stack frames.
*/
static class MethodAndArgsCaller implements Runnable {
/** method to call
*/private final Method mMethod;
/** argument array
*/private final String[] mArgs;public MethodAndArgsCaller(Method method,
String[] args) {
mMethod = method;mArgs = args;}
public void run() {try {
// 1mMethod.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);
}
}
}
```

在注释1处，这个mMethod就是指的SystemServer的main()方法，这里动态调用了SystemServer的main()方法，最终，SystemServer进程就进入了SystemServer的main()方法中了。这里还有个遗留问题，为什么不直接在findStaticMain()方法中直接动态调用SystemServer的main()方法呢？原因就是这种递归返回后再执行入口方法的方式会让SystemServer的main()方法看起来像是SystemServer的入口方法，而且，这样也会清除之前所有SystemServer相关设置过程中需要的堆栈帧。

> --------走到 SystemService 进程

1. /frameworks/base/services/java/com/android/server/SystemServer.java

接下来我们看看SystemServer的main()方法：

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

main()方法中调用了SystemServer的run()方法，如下所示：

```
private void run() {try {...
// 1Looper.prepareMainLooper();...
// Initialize native services.
// 2System.loadLibrary("android_servers");
// Check whether we failed to shut down last time we tried.
// This call may not return.performPendingShutdown();
// Initialize the system context.createSystemContext();
// Create the system service manager.
// 3mSystemServiceManager = new SystemServiceManager(mSystemContext);
mSystemServiceManager.setStartInfo(mRuntimeRestart,mRuntimeStartElapsedTime,
mRuntimeStartUptime);
LocalServices.addService(SystemServiceManager.class, mSystemServiceManager);
// Prepare the thread pool for init tasks that can be
parallelizedSystemServerInitThreadPool.get();}
finally {traceEnd();
// InitBeforeStartServices}
// Start services.try {
traceBeginAndSlog("StartServices");
// 4startBootstrapServices();
// 5startCoreServices();
//6startOtherServices();
SystemServerInitThreadPool.shutdown();}
catch (Throwable ex) {Slog.e("System",
"************************************************");
Slog.e("System", "************ Failure starting system services", ex);
throw ex;}
finally {
traceEnd();
}...
// Loop forever.
// 7Looper.loop();
throw new RuntimeException("Main thread loop unexpectedly exited");
}
```

在注释1处，创建了消息Looper。

在注释2处，加载了动态库libandroid_servers.so。

在注释3处，创建了SystemServerManager，它的作用是对系统服务进行创建、启动和生命周期管理。

在注释4处的startBootstarpServices()方法中使用SystemServiceManager启动了ActivityManagerService、PackageManagerService、PowerManagerService等引导服务。

在注释5处的startCoreServices()方法中则启动了BatteryService、WebViewUpdateService、DropBoxManagerService、UsageStatsService4个核心服务。

在注释6处的startOtherServices()方法中启动了WindowManagerService、InputManagerService、CameraService等其它服务。这些服务的父类都是SystemService。

可以看到，上面把系统服务分成了三种类型：引导服务、核心服务、其它服务。这些系统服务共有100多个，其中对于我们来说比较关键的有：

>

- 引导服务：ActivityManagerService，负责四大组件的启动、切换、调度。
- 引导服务：PackageManagerService，负责对APK进行安装、解析、删除、卸载等操作。
- 引导服务：PowerManagerService，负责计算系统中与Power相关的计算，然后决定系统该如何反应。
- 核心服务：BatteryService，管理电池相关的服务。
- 其它服务：WindowManagerService，窗口管理服务。
- 其它服务：InputManagerService，管理输入事件。

很多系统服务的启动逻辑都是类似的，这里我以启动ActivityManagerService服务来进行举例，代码如下所示：

```
mActivityManagerService =
mSystemServiceManager.startService(ActivityManagerService.Lifecycle.class).getSe
rvice();
```

SystemServiceManager 的 startService() 方法启动了ActivityManagerService，该启动方法如下所示：

```
@SuppressWarnings("unchecked")
public <T extends SystemService> T startService(Class<T> serviceClass) {
try {final String name = serviceClass.getName();
...try {Constructor<T> constructor = serviceClass.getConstructor(Context.class);
// 1service = constructor.newInstance(mContext);
}
catch (InstantiationException ex) {...
// 2startService(service);return service;
}
finally {Trace.traceEnd(Trace.TRACE_TAG_SYSTEM_SERVER);
}
}
```

在注释1处使用反射创建了ActivityManagerService实例，并在注释2处调用了另一个startService()重载方法，如下所示：

```
public void startService(@NonNull final SystemService service) {
// Register it.
// 1mServices.add(service);
// Start it.long time = SystemClock.elapsedRealtime();
try {
// 2service.onStart();
}
catch (RuntimeException ex)
{
throw new RuntimeException("Failed to start service " +
service.getClass().getName()+ ": onStart threw an exception", ex);
}
warnIfTooLong(SystemClock.elapsedRealtime() - time, service, "onStart");
}
```

在注释1处，首先会将ActivityManagerService添加在mServices中，它是一个存储SystemService类型的ArrayList，这样就完成了ActivityManagerService的注册。

在注释2处，调用了ActivityManagerService的onStart()方法完成了启动ActivityManagerService服务。

除了使用SystemServiceManager的startService()方法来启动系统服务外，也可以直接调用服务的main()方法来启动系统服务，如PackageManagerService：

```
mPackageManagerService = PackageManagerService.main(mSystemContext,
installer,mFactoryTestMode != FactoryTest.FACTORY_TEST_OFF, mOnlyCore);
```

这里直接调用了PackageManagerService的main()方法：

```
public static PackageManagerService main(Context context, Installer
installer,boolean factoryTest, boolean onlyCore) {
// Self-check for initial
settings.PackageManagerServiceCompilerMapping.checkProperties();
// 1PackageManagerService m = new PackageManagerService(context,
installer,factoryTest, onlyCore);
m.enableSystemUserPackages();
// 2ServiceManager.addService("package", m);
// 3final PackageManagerNative pmn = m.new PackageManagerNative();
ServiceManager.addService("package_native", pmn);
return m;
}
```

在注释1处，直接新建了一个PackageManagerService实例，

注释2处将PackageManagerService注册到服务大管家ServiceManager中，ServiceManager用于管理系统中的各种Service，用于系统C/S架构中的Binder进程间通信，即如果Client端需要使用某个Servcie，首先应该到ServiceManager查询Service的相关信息，然后使用这些信息和该Service所在的Server进程建立通信通道，这样Client端就可以服务端进程的Service进行通信了。

CarServiceHelperService启动

1、SystemServer中在startOtherServices启动

> mActivityManagerService.systemReady()里启动
> 判断feature属性值PackageManager.FEATURE_AUTOMOTIVE是否支持
> （ FEATURE_AUTOMOTIVE = “android.hardware.type.automotive” ）
> SystemServiceManager启动CAR_SERVICE_HELPER_SERVICE_CLASS = "com.android.internal.car.CarServiceHelperService"，调用CarServiceHelperService构造函数、onStart()

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

```
if (mPackageManager.hasSystemFeature(PackageManager.FEATURE_AUTOMOTIVE)) {
    t.traceBegin("StartCarServiceHelperService");
    final SystemService cshs = mSystemServiceManager
            .startService(CAR_SERVICE_HELPER_SERVICE_CLASS);
    if (cshs instanceof Dumpable) {
        mDumper.addDumpable((Dumpable) cshs);
    }
    if (cshs instanceof DevicePolicySafetyChecker) {
        dpms.setDevicePolicySafetyChecker((DevicePolicySafetyChecker) cshs);
    }
    t.traceEnd();
}
```

2、CarServiceHelperService初始化

2.1 CarServiceHelperService构造函数

> mHandler在HandlerThread处理事件
> CarLaunchParamsModifier当Activity启动时控制管理Car显示。该政策只能影响乘客可以使用的显示器。当前用户可以始终启动任何显示。
> CarWatchdogDaemonHelper car看门狗守护程序Watchdog的帮助程序类。
> mCarServiceProxy管理CarServiceHelperService请求的CarService操作。它用于在连接、终止和重新连接时向CarService发送和重新发送活页夹调用。
> （ packages/services/Car/service/src/com/android/car/CarService.java ）

```
public CarServiceHelperService(Context context) {
    this(context,
            new CarLaunchParamsModifier(context),
            new CarWatchdogDaemonHelper(TAG),
            null
    );
}

@VisibleForTesting
CarServiceHelperService(
        Context context,
        CarLaunchParamsModifier carLaunchParamsModifier,
        CarWatchdogDaemonHelper carWatchdogDaemonHelper,
        CarServiceProxy carServiceOperationManager) {
    super(context);

    mContext = context;
    mHandlerThread.start();
    mHandler = new Handler(mHandlerThread.getLooper());
    mCarLaunchParamsModifier = carLaunchParamsModifier;
    mCarWatchdogDaemonHelper = carWatchdogDaemonHelper;
    mCarServiceProxy =
            carServiceOperationManager == null ? new CarServiceProxy(this)
                    : carServiceOperationManager;
    UserManagerInternal umi =
LocalServices.getService(UserManagerInternal.class);
    if (umi != null) {
        umi.addUserLifecycleListener(new UserLifecycleListener() {
            @Override
```

```
            public void onUserCreated(UserInfo user, Object token) {
                if (DBG) Slogf.d(TAG, "onUserCreated(): %s",
user.toFullString());
            }
            @Override
            public void onUserRemoved(UserInfo user) {
                if (DBG) Slogf.d(TAG, "onUserRemoved(): $s",
user.toFullString());
                mCarServiceProxy.onUserRemoved(user);
            }
        });
    } else {
        Slogf.e(TAG, "UserManagerInternal not available - should only happen on
unit tests");
    }
    mCarDevicePolicySafetyChecker = new CarDevicePolicySafetyChecker(this);
}
```

2.2 `onStart()`启动绑定CarService

> mShutdownEventReceiver注册监听广播，通知car watchdog daemon改变StateType
> mCarWatchdogDaemonHelper.connect()连接car watchdog daemon：
> ServiceManager.getService(CAR_WATCHDOG_DAEMON_INTERFACE)，其中
> CAR_WATCHDOG_DAEMON_INTERFACE = "carwatchdogd_system"
> mContext.bindServiceAsUser(intent, mCarServiceConnection, ...)绑定CarService
> loadNativeLibrary() 导入jni：System.loadLibrary("car-framework-service-jni")

public void onStart() {
EventLog.writeEvent(EventLogTags.CAR_HELPER_START);

```
IntentFilter filter = new IntentFilter(Intent.ACTION_REBOOT);
filter.addAction(Intent.ACTION_SHUTDOWN);
mContext.registerReceiverForAllUsers(mShutdownEventReceiver, filter, null,
null);
mCarWatchdogDaemonHelper.addOnConnectionChangeListener(mConnectionListener);
mCarWatchdogDaemonHelper.connect();
Intent intent = new Intent();
intent.setPackage("com.android.car");
intent.setAction(CAR_SERVICE_INTERFACE);
if (!mContext.bindServiceAsUser(intent, mCarServiceConnection,
Context.BIND_AUTO_CREATE,
        mHandler, UserHandle.SYSTEM)) {
    Slogf.wtf(TAG, "cannot start car service");
}
loadNativeLibrary();
}
```

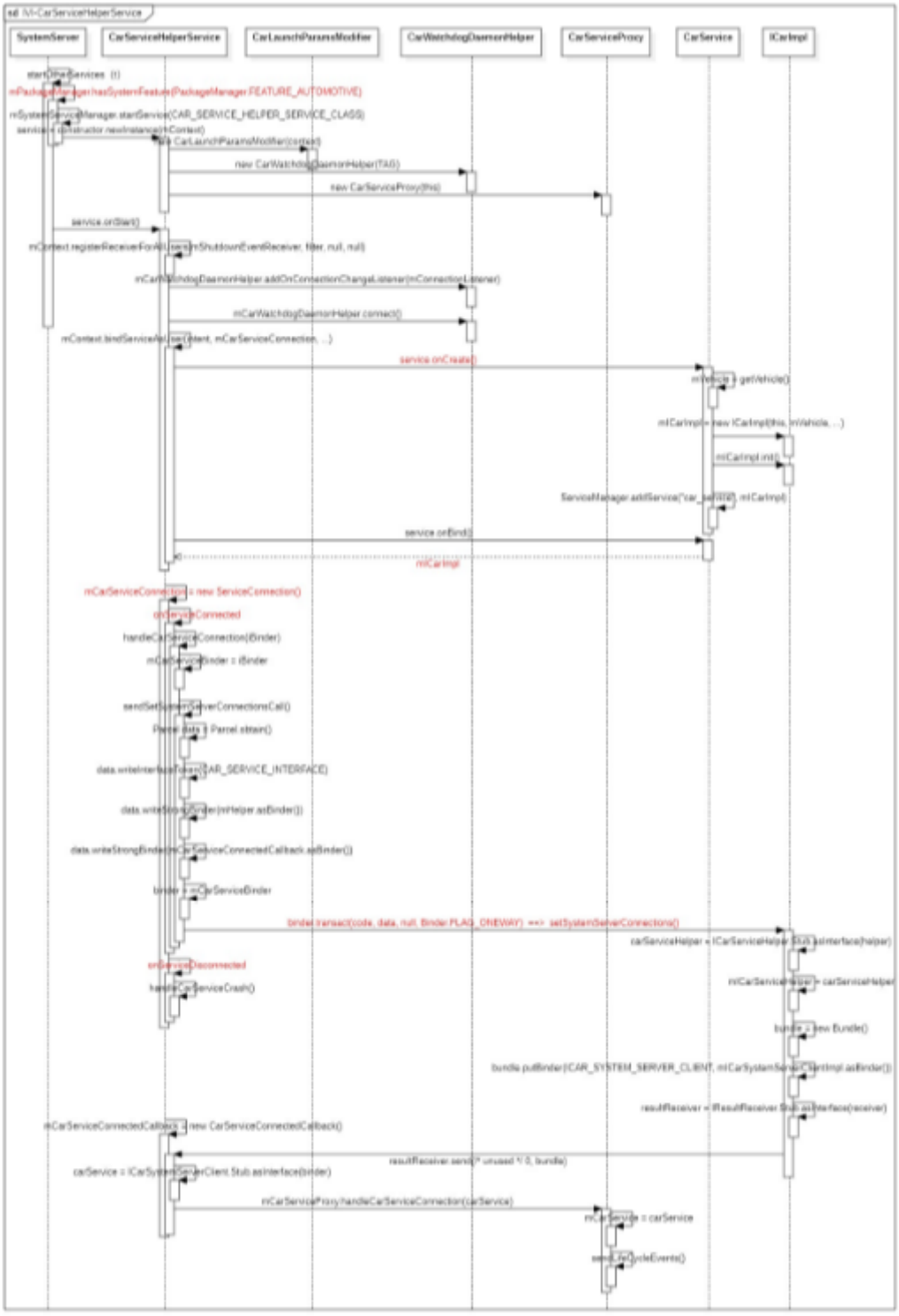
3、绑定CarService回调`mCarServiceConnection`

> 查看packages/services/Car/service/AndroidManifest.xml，绑定服务CarService服务
>
> handleCarServiceConnection(iBinder)连接CarService，通过Parcel数据传递，
> ICarServiceHelperImpl、CarServiceConnectedCallback响应处理

public static final String CAR_SERVICE_INTERFACE = “android.car.ICar”

```
private final ServiceConnection mCarServiceConnection = new ServiceConnection()
{
   @Override
   public void onServiceConnected(ComponentName componentName, IBinder iBinder)
{
       if (DBG) {
           Slogf.d(TAG, "onServiceConnected: %s", iBinder);
       }
       handleCarServiceConnection(iBinder);
   }

   @Override
   public void onServiceDisconnected(ComponentName componentName) {
       handleCarServiceCrash();
   }
};
```

4、简要时序图


sd IVI-CarServiceHelperService
SystemServer
CarServiceHelperService
CarLaunchParamsModifier
CarWatchdogDaemonHelper
CarServiceProxy
CarService
ICarImpl
startOtherServices ()
mPackageManager.hasSystemFeature(PackageManager.FEATURE_AUTOMOTIVE)
mSystemServiceManager.startService(CAR_SERVICE_HELPER_SERVICE_CLASS)
service = constructor.newInstance(mContext)
new CarLaunchParamsModifier(context)
new CarWatchdogDaemonHelper(TAG)
new CarServiceProxy(this)
service.onStart()
mContext.registerReceiverForAllUsers(mShutdownEventReceiver, filter, null, null)
mCarWatchdogDaemonHelper.addOnConnectionChangeListener(mConnectionListener)
mCarWatchdogDaemonHelper.connect()
mContext.bindServiceAsUser(intent, mCarServiceConnection, ...)
service.onCreate()
mVehicle = getVehicle()
mCarImpl = new ICarImpl(this, mVehicle, ...)
mCarImpl.init()
ServiceManager.addService("car_service", mCarImpl)
service.onBind()
mICarImpl
mCarServiceConnection = new ServiceConnection()
onServiceConnected
handleCarServiceConnection(IBinder)
mCarServiceBinder = IBinder
sendSetSystemServerConnectionsCall()
Parcel data = Parcel.obtain()
data.writeInterfaceToken(CAR_SERVICE_INTERFACE)
data.writeStrongBinder(mHelper.asBinder())
data.writeStrongBinder(mCarServiceConnectedCallback.asBinder())
binder = mCarServiceBinder
binder.transact(code, data, null, Binder.FLAG_ONEWAY) ==> setSystemServerConnections()
carServiceHelper = ICarServiceHelper.Stub.asInterface(helper)
mCarServiceHelper = carServiceHelper
bundle = new Bundle()
bundle.putBinder(CAR_SYSTEM_SERVER_CLIENT, mCarSystemServerClientImpl.asBinder())
resultReceiver = IResultReceiver.Stub.asInterface(receiver)
onServiceDisconnected
handleCarServiceCrash()
mCarServiceConnectedCallback = new CarServiceConnectedCallback()
resultReceiver.send(/* unused */ 0, bundle)
carService = ICarSystemServerClient.Stub.asInterface(binder)
mCarServiceProxy.handleCarServiceConnection(carService)
mCarService = carService
sendHalLifecycleEvents()