vivo自定义接口方法使用规范

版本说明

版本号	作者	更新时间
1.0.0	蔡小波	2018/06/27
1.0.1	蔡小波	2018/10/15

1.入网和宣传机型名获取

入网型号获取：ro.product.model和Build.MODEL

宣传机型名获取 ：ro.vivo.market.name

获取属性是否含有vivo字段：ro.product.manufacturer

判断机型所属型号（Y、X、Xplay）：ro.vivo.market.name

详细示例如下：

PD1732_A_1.2.6 版本已经修改对应属性：

入网名 -- 全网通：

ro.product.model = V1732A

入网名 --移动全网通：

ro.product.model = V1732T

宣传名--全网通和移动全网通一样：

ro.vivo.market.name = vivo Y81s

注：PD1730CA、PD1730DA项目属于过渡项目比较特殊需要格外注意

2.外部机型名出现中文适配方式

公司app在与服务器交互时，会传递model参数给服务器，用以标识机型名。

近期的PD1816以及PD1730EA机器上，机型名将携带有中文，例如：ro.vivo.market.name = vivo X99 梦幻版

机型名加入中文后，服务器在跨网传递机型、数据过滤等功能上可能出现异常，所以需要对其进行适配，防止线上出现事故。

机型名中含有中文的机器，会在原有的ro.vivo.market.name基础上，新增ro.vivo.internet.name字段，新增的字段不会出现中文，以此规避风险。

参考代码如下：

```
/**
 * 获取系统相关属性类
 *
 * @author mejon
 */
public class SystemUtils {

    private static final String PROP_VIVO_NAME = "ro.vivo.market.name";
    private static final String PROP_VIVO_IN_NAME = "ro.vivo.internet.name";

    /**
     * 获取手机制造商
     *
     * @return 手机制造商
     */
    public static String getProductName() {

        String proName = getSystemProperties(PROP_VIVO_IN_NAME, Build.UNKNOWN);
        if (!TextUtils.isEmpty(proName) && !Build.UNKNOWN.equals(proName)) {
            if (!proName.toLowerCase().contains("vivo")) {
                proName = "vivo " + proName;
            }
            return proName;
        }
        proName = getSystemProperties(PROP_VIVO_NAME, Build.UNKNOWN);
        if (Build.UNKNOWN.equals(proName) || TextUtils.isEmpty(proName)) {
            proName = Build.MODEL;
        } else {
            if (!proName.toLowerCase().contains("vivo")) {
                proName = "vivo " + proName;
            }
        }
        return proName;
    }

    public static String getSystemProperties(String key, String def) {

        Method method = null;
        String value = def;
```

```
        try {
            method = Class.forName("android.os.SystemProperties").getMethod("get",
String.class);
            value = (String) method.invoke(null, key);
        } catch (Exception e) {
            e.printStackTrace();
        }

        if (value == null || value.length() == 0) {
            value = def;
        }
        return value;
    }

}
```

3.共享uid的应用so库集成

App应用有共享sharedUserId，且有集成so库的时候，在适配过程中需要内置32位和64位库。

如果你share的对象只内置了32位库，那么你就不能只内置64位的库；如果你share的对象只内置了64位库，那么你就不能只内置32位的库。总结起来其实就是要和share的对象保持一致才行。由于shareuid1000的应用实在太多了，大概有70个，这些APK在64位机器上，要么只内置了64位库，要么内置了32位加64位库（没有只内置32位库的情况），如果一个新开发的APK不和别的保持一致，而是只内置了32位库的话，就会有几率把其他所有shareuid1000的应用的primaryCpuAbi都强行改为32位，这样的话所有shareuid1000的应用都会出现crash的现象，这样的话这台手机直接就没法用了。

为避免此问题，需要注意：

- 64位机器，uid1000的app如果有库的话，同时内置32位和64位库；
- 否则概率性导致uid为1000的app所有app崩溃；
- 框架做了规避方案，但该方案会引起只配置了32位库（uid 1000）的应用功能异常。

4.Android 9.0 emmc id ufs id使用

Android P上由于google 更加严格执行treble架构，sepolicy中新增了新的neverallow规则，禁止 coredomain 访问没有明确label的sysfs以及proc file。

由于上述原因，导致vivo之前通过直接读取节点/sys/block/mmcblk0/device/cid， sys/ufs/ufsid的方案在9.0 上行不通 。

因为上述限制， 为了遵循treble 架构， 采用hidl的方式，来间接读取对应的emmc id， ufs id：

1）java 代码获取请直接调用如下方法：

```
android.util.FtDeviceInfo.getEmmcId()

android.util.FtDeviceInfo.getUFSId()
```

2) native 层通过如下调用获取：

```
sp<IOmnipotentService> service;

service = IOmnipotentService::getService();

hidl_string ret;

service->doOmnipotentTask("get_emmc_id",[&](const hidl_string& result) {

        ret = result;

});

hidl_string ufs;

service->doOmnipotentTask("get_ufs_id",[&](const hidl_string& result) {

        ufs = result;

});
```

注意事项:

（ 1 ） native方法编译需要添加相关依赖

至于依赖修改请参考如下目录：vendor/vivo/source/hardware/interfaces/omnipotentservice/tests 下 Android.bp

（ 2) 添加目标domain xxx对 该hal的访问，请参考如下修改：

hal_client_domain(xxx, hal_omnipotentservice)

allow xxx hal_omnipotentservice_hwservice:hwservice_manager find;