迪拜小王子

Read the fucking source code

首页 归档 关于 订阅

Android 6.0省电模式研究（二）

Jul 14, 2016 | 深入调研 | 144 Hits

接着上一篇，回顾问题要点：
省电模式下：

- 实现高频网络
- 尽可能不要用户干预
- 尽可能不要添加权限（可以放宽）

调研思路

- 从PowerManager入手，深入了解如何做到添加白名单
- 从JobScheduler入手，深入了解它是如何设置控制网络的
- 从ConnectivityManager入手，深入了解它是如何控制网络连接的（最后发现这个思路和第一个汇合了）

跟踪的时候，直接贴核心源码，描述会放到注释里。

PowerManager思路

```
/**
 * Return whether the given application package name is on the device's power whitelist.
 * Apps can be placed on the whitelist through the settings UI invoked by
 * {@link android.provider.Settings#ACTION_IGNORE_BATTERY_OPTIMIZATION_SETTINGS}.
 */
public boolean isIgnoringBatteryOptimizations(String packageName) {
    synchronized (this) {
        if (mIDeviceIdleController == null) {
            mIDeviceIdleController = IDeviceIdleController.Stub.asInterface(
                ServiceManager.getService(Context.DEVICE_IDLE_CONTROLLER));
        }
    }
    try {
        return mIDeviceIdleController.isPowerSaveWhitelistApp(packageName);
    } catch (RemoteException e) {
        return false;
    }
}

/** 从上面我们看到，和 mIDeviceIdleController 有关，我们瞅瞅 mIDeviceIdleController 干啥的，且怎传到这里的 */
/** DeviceIdleController.java 的 onStart 函数 */
@Override
public void onStart() {

    /** 省略无关 */

    publishBinderService(Context.DEVICE_IDLE_CONTROLLER, new BinderService());
    publishLocalService(LocalService.class, new LocalService());
}
```

很明显，Google相当聪明，没有把 DeviceIdleController 这个对象往外传，明显知道可以用java反射去执行一些函数，然而他们抽象了一层接口，只是可调用，无法有具体的对象。这个我们也可以看到我们如果有些东西不想被用户反射得到，传到给外面的对象应该只有接口，里面尽可能不要有对象。这样

外面拿到这个对象也完全没辙。

```
/** 接着看看 BinderService 怎么实现白名单的 */

private final class BinderService extends IDeviceIdleController.Stub {

    /** 省略无关 */

    @Override public boolean isPowerSaveWhitelistApp(String name) {
        return isPowerSaveWhitelistAppInternal(name);
    }

    @Override public void addPowerSaveWhitelistApp(String name) {
        getContext().enforceCallingOrSelfPermission(android.Manifest.permission.DEVICE_POWER,
            null);
        addPowerSaveWhitelistAppInternal(name);
    }

    @Override public void addPowerSaveTempWhitelistApp(String packageName, long duration,
                int userId, String reason) throws RemoteException {
        getContext().enforceCallingPermission(Manifest.permission.CHANGE_DEVICE_IDLE_TEMP_WHITELIST,
            "No permission to change device idle whitelist");
        final int callingUid = Binder.getCallingUid();
        userId = ActivityManagerNative.getDefault().handleIncomingUser(
                    Binder.getCallingPid(),
                    callingUid,
                    userId,
                    /*allowAll=*/ false,
                    /*requireFull=*/ false,
                    "addPowerSaveTempWhitelistApp", null);
        final long token = Binder.clearCallingIdentity();
        try {
            DeviceIdleController.this.addPowerSaveTempWhitelistAppInternal(callingUid,
                        packageName, duration, userId, true, reason);
        } finally {
            Binder.restoreCallingIdentity(token);
        }
    }
}

/** 真正添加白名单的是 addPowerSaveWhitelistAppInternal ， addPowerSaveTempWhitelistAppInternal */
public boolean addPowerSaveWhitelistAppInternal(String name) {
    synchronized (this) {
        try {
            ApplicationInfo ai = getContext().getPackageManager().getApplicationInfo(name, 0);
            if (mPowerSaveWhitelistUserApps.put(name, UserHandle.getAppId(ai.uid)) == null) {
                reportPowerSaveWhitelistChangedLocked();
                updateWhitelistAppIdsLocked();
                writeConfigFileLocked(); /** 写白名单文件 */
            }
            return true;
        } catch (PackageManager.NameNotFoundException e) {
            return false;
        }
    }
}

void writeConfigFileLocked() {
    mHandler.removeMessages(MSG_WRITE_CONFIG);
    mHandler.sendEmptyMessageDelayed(MSG_WRITE_CONFIG, 5000);
}

void handleWriteConfigFile() {
    final ByteArrayOutputStream memStream = new ByteArrayOutputStream();

    try {
        synchronized (this) {
        XmlSerializer out = new FastXmlSerializer();
        out.setOutput(memStream, StandardCharsets.UTF_8.name());
        writeConfigFileLocked(out);
        }
    } catch (IOException e) {
    }

    synchronized (mConfigFile) {
        FileOutputStream stream = null;
        try {
            stream = mConfigFile.startWrite();
            memStream.writeTo(stream);
            stream.flush();
            FileUtils.sync(stream);
            stream.close();
            mConfigFile.finishWrite(stream);
        } catch (IOException e) {
            Slog.w(TAG, "Error writing config file", e);
            mConfigFile.failWrite(stream);
        }
    }
}

void writeConfigFileLocked(XmlSerializer out) throws IOException {
    out.startDocument(null, true);
    out.startTag(null, "config");
    for (int i = 0; i < mPowerSaveWhitelistUserApps.size(); i++) {
        String name = mPowerSaveWhitelistUserApps.keyAt(i);
        out.startTag(null, "wl");
        out.attribute(null, "n", name);
        out.endTag(null, "wl");
    }
    out.endTag(null, "config");
    out.endDocument();
}

public final AtomicFile mConfigFile;

public DeviceIdleController(Context context) {
    super(context);
    mConfigFile = new AtomicFile(new File(getSystemDir(), "deviceidle.xml"));
    mHandler = new MyHandler(BackgroundThread.getHandler().getLooper());
}
```

大概清楚了，写到系统目录下的deviceidle.xml，而且要更新 mPowerSaveWhitelistUserApps。
读取系统文件应该可以读，但是写，这个肯定没有权限的，但是这个可以作为一个解决方案尝试的一个点。

下面看看临时增加网络请求的代码

```
/** 添加临时网络白名单 */

public void addPowerSaveTempWhitelistAppInternal(int callingUid, String packageName,
            long duration, int userId, boolean sync, String reason) {
    try {
        int uid = getContext().getPackageManager().getPackageUid(packageName, userId);
        int appId = UserHandle.getAppId(uid);
        addPowerSaveTempWhitelistAppDirectInternal(callingUid, appId, duration, sync, reason);
    } catch (NameNotFoundException e) {
    }
}
/**
 * Adds an app to the temporary whitelist and resets the endTime for granting the
 * app an exemption to access network and acquire wakelocks.
 */
public void addPowerSaveTempWhitelistAppDirectInternal(int callingUid, int appId,
                long duration, boolean sync, String reason) {
    final long timeNow = SystemClock.elapsedRealtime();
    Runnable networkPolicyTempWhitelistCallback = null;
    synchronized (this) {
        int callingAppId = UserHandle.getAppId(callingUid);
        if (callingAppId >= Process.FIRST_APPLICATION_UID) {
            if (!mPowerSaveWhitelistSystemAppIds.get(callingAppId)) {
                throw new SecurityException("Calling app " + UserHandle.formatUid(callingUid)
                        + " is not on whitelist");
            }
        }
        duration = Math.min(duration, mConstants.MAX_TEMP_APP_WHITELIST_DURATION);
        Pair<MutableLong, String> entry = mTempWhitelistAppIdEndTimes.get(appId);
```

```
            final boolean newEntry = entry == null;
            // Set the new end time
            if (newEntry) {
                entry = new Pair<>(new MutableLong(0), reason);
                mTempWhitelistAppIdEndTimes.put(appId, entry);
            }
            entry.first.value = timeNow + duration;
            if (DEBUG) {
                Slog.d(TAG, "Adding AppId " + appId + " to temp whitelist");
            }
            if (newEntry) {
                // No pending timeout for the app id, post a delayed message
                try {
                    mBatteryStats.noteEvent(BatteryStats.HistoryItem.EVENT_TEMP_WHITELIST_START,
                            reason, appId);
                } catch (RemoteException e) {
                }
                postTempActiveTimeoutMessage(appId, duration);
                updateTempWhitelistAppIdsLocked();
                if (mNetworkPolicyTempWhitelistCallback != null) {
                    if (!sync) {
                        mHandler.post(mNetworkPolicyTempWhitelistCallback);
                    } else {
                        networkPolicyTempWhitelistCallback = mNetworkPolicyTempWhitelistCallback;
                    }
                }
                reportTempWhitelistChangedLocked();
            }
        }
        if (networkPolicyTempWhitelistCallback != null) {
            networkPolicyTempWhitelistCallback.run();
        }
    }
/** 我们注意一下，这个临时到底能有多长 */
MAX_TEMP_APP_WHITELIST_DURATION = mParser.getLong(
        KEY_MAX_TEMP_APP_WHITELIST_DURATION, 5 * 60 * 1000L);
/** 放弃吧，只有5分钟，没有实用性，而且要求是系统AppId，但是我们可以看看用什么机制搞定的 */
```

临时申请白名单的原理很简单，更新本地的白名单，通知电池管理，发送开始广播。
注意到，临时申请白名单的时候，设置完了回调了一个网络策略线程 mNetworkPolicyTempWhitelistCallback
通过追踪，找到实现，这个有点惊喜，因为是网络策略管理。

```
/** NetworkPolicyManagerService.java */

final private Runnable mTempPowerSaveChangedCallback = new Runnable() {
    @Override
    public void run() {
        synchronized (mRulesLock) {
            updatePowerSaveTempWhitelistLocked();
            updateRulesForTempWhitelistChangeLocked();
            purgePowerSaveTempWhitelistLocked();
        }
    }
};

/** 直接设置防火墙规则，现在也明白了省电模式是如何禁用网络的 */

void updateRuleForAppIdleLocked(int uid) {
    if (!isUidValidForRules(uid)) return;

    int appId = UserHandle.getAppId(uid);
    if (!mPowerSaveTempWhitelistAppIds.get(appId) && isUidIdle(uid)) {
        setUidFirewallRule(FIREWALL_CHAIN_STANDBY, uid, FIREWALL_RULE_DENY);
    } else {
        setUidFirewallRule(FIREWALL_CHAIN_STANDBY, uid, FIREWALL_RULE_DEFAULT);
    }
}

void updateRuleForDeviceIdleLocked(int uid) {
    if (mDeviceIdleMode) {
        int appId = UserHandle.getAppId(uid);
        if (mPowerSaveTempWhitelistAppIds.get(appId) || mPowerSaveWhitelistAppIds.get(appId)
            || isProcStateAllowedWhileIdle(mUidState.get(uid))) {
            setUidFirewallRule(FIREWALL_CHAIN_DOZABLE, uid, FIREWALL_RULE_ALLOW);
        } else {
            setUidFirewallRule(FIREWALL_CHAIN_DOZABLE, uid, FIREWALL_RULE_DEFAULT);
        }
    }
}

private void setUidFirewallRule(int chain, int uid, int rule) {
    try {
        mNetworkManager.setFirewallUidRule(chain, uid, rule);
    } catch (IllegalStateException e) {
        Log.wtf(TAG, "problem setting firewall uid rules", e);
    } catch (RemoteException e) {
        // ignored; service lives in system_server
    }
}

private final INetworkManagementService mNetworkManager;

public static final int FIREWALL_RULE_DEFAULT = 0;
public static final int FIREWALL_RULE_ALLOW = 1;
public static final int FIREWALL_RULE_DENY = 2;

public static final int FIREWALL_TYPE_WHITELIST = 0;
public static final int FIREWALL_TYPE_BLACKLIST = 1;

public static final int FIREWALL_CHAIN_NONE = 0;
public static final int FIREWALL_CHAIN_DOZABLE = 1;
public static final int FIREWALL_CHAIN_STANDBY = 2;

public static final String FIREWALL_CHAIN_NAME_NONE = "none";
public static final String FIREWALL_CHAIN_NAME_DOZABLE = "dozable";
public static final String FIREWALL_CHAIN_NAME_STANDBY = "standby";

private static final boolean ALLOW_PLATFORM_APP_POLICY = true;
```

我们赶紧看看防火墙设置， INetworkManagementService 实现类

```
/** NetworkManagementService.java */

public void setFirewallUidRule(int chain, int uid, int rule) {
    enforceSystemUid(); /** 要求是系统权限 */
    setFirewallUidRuleInternal(chain, uid, rule);
}
private void setFirewallUidRuleInternal(int chain, int uid, int rule) {
    synchronized (mQuotaLock) {
        SparseIntArray uidFirewallRules = getUidFirewallRules(chain);

        final int oldUidFirewallRule = uidFirewallRules.get(uid, FIREWALL_RULE_DEFAULT);
        if (DBG) {
            Slog.d(TAG, "oldRule = " + oldUidFirewallRule
                        + ", newRule=" + rule + " for uid=" + uid);
        }
        if (oldUidFirewallRule == rule) {
            if (DBG) Slog.d(TAG, "!!!!! Skipping change");
            // TODO: eventually consider throwing
            return;
        }

        try {
            String ruleName = getFirewallRuleName(chain, rule);
            String oldRuleName = getFirewallRuleName(chain, oldUidFirewallRule);

            if (rule == NetworkPolicyManager.FIREWALL_RULE_DEFAULT) {
                uidFirewallRules.delete(uid);
            } else {
                uidFirewallRules.put(uid, rule);
            }

            if (!ruleName.equals(oldRuleName)) {
                mConnector.execute("firewall", "set_uid_rule", getFirewallChainName(chain), uid,
                        ruleName);
            }
        } catch (NativeDaemonConnectorException e) {
            throw e.rethrowAsParcelableException();
        }
    }
}
/**
 * connector object for communicating with netd
 */
private final NativeDaemonConnector mConnector;
```

实现真正更改是用mConnector发命令。mConnector是封装的一个LocalSocket，实现的类在android.jar是获取不到的。
看源码的时候同时发现网络策略那一块，到最后执行的是

```
@Override
public void setUidNetworkRules(int uid, boolean rejectOnQuotaInterfaces) {
    mContext.enforceCallingOrSelfPermission(CONNECTIVITY_INTERNAL, TAG);

    // silently discard when control disabled
    // TODO: eventually migrate to be always enabled
    if (!mBandwidthControlEnabled) return;

    synchronized (mQuotaLock) {
        final boolean oldRejectOnQuota = mUidRejectOnQuota.get(uid, false);
        if (oldRejectOnQuota == rejectOnQuotaInterfaces) {
            // TODO: eventually consider throwing
            return;
        }

        try {
            mConnector.execute("bandwidth",
                    rejectOnQuotaInterfaces ? "addnaughtyapps" : "removenaughtyapps", uid);
            if (rejectOnQuotaInterfaces) {
                mUidRejectOnQuota.put(uid, true);
            } else {
                mUidRejectOnQuota.delete(uid);
            }
        } catch (NativeDaemonConnectorException e) {
            throw e.rethrowAsParcelableException();
        }
    }
}
```

所以，mConnector 值得研究以下，后面有空专门看看。

思路一总结

看完这些核心代码，总结为以下几个方向：

- 深入研究防火墙如何去修改
- 修改系统文件，添加白名单

Android 省电模式 分享到

内存泄露分析（一） Android 6.0省电模式研究（一）

分享到： 微博 QQ空间 腾讯微博 微信

0条评论

还没有评论，沙发等你来抢

社交帐号登录: 微信 微博 QQ 人人 更多»

说点什么吧...

发布

Kinva正在使用多说