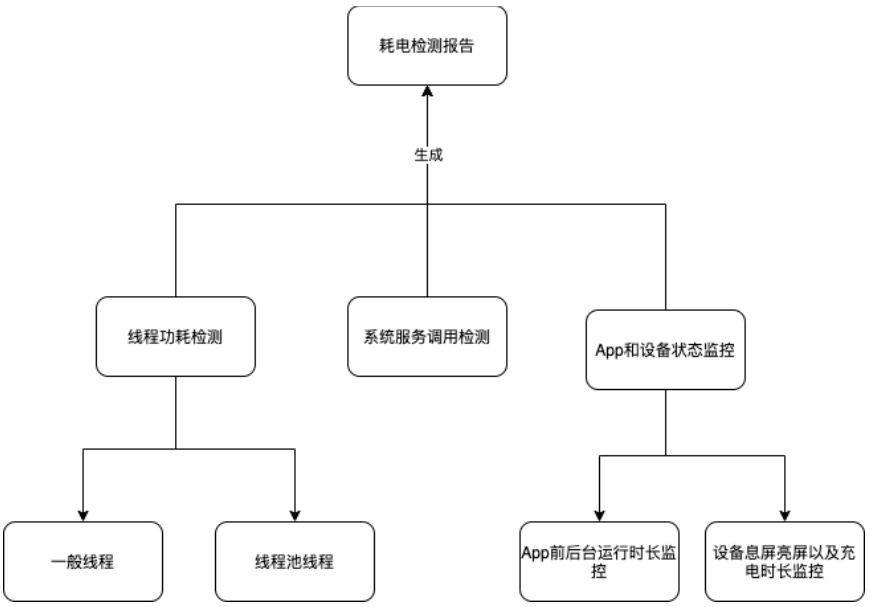
Dokit耗电检测技术方案

主要从**线程功耗异常**、**系统服务调用异常和App**、**设备状态**三个方面来实现对异常耗电情况的监控。功能模块图如下：



1. 线程功耗异常

线程可以分为**一般线程**和**线程池线程**。

1.1 一般线程

首先通过Linux 命令 `proc/[pid]/stat` 和 `proc/[pid]/task/[tid]/stat` 可以 Dump 当前 App 进程和线程的统计信息。（信息中会包含线程运行的用户时间、系统时间等等，根据时间可以转化成线程的

jiffy消耗）。根据每个线程的Jiffy消耗和线程的运行状态，可以得到当前线程在这段时间内的运行状况，是否发生异常。

目前初步实现的核心逻辑如下。会在开始监控的时间和结束监控的时间点对系统做一次快照，在`getAllThreadInfo()`方法中通过Linux 命令 proc/[pid]/stat 和 proc/[pid]/task/[tid]/stat 输出并记录系统当前时间点App的线程的状态和jiffy消耗。然后当监控结束时会触发`calculateDiff`方法通过对比开始和结束时线程状态的差异来计算这一时间段内线程的消耗。

```
1 个用法  Truth360
public void start() {
    startTime = System.currentTimeMillis();
    // 对开始时间的系统状态做快照

    // 线程
    ProcStateUtil procStateUtil = new ProcStateUtil();
    preProcState = procStateUtil.getAllThreadInfo();
}


1 个用法  Truth360
public void finish() {
    // 对结束时间的系统状态做快照
    this.endTime = System.currentTimeMillis();
    // 线程
    ProcStateUtil procStateUtil = new ProcStateUtil();
    curProcState = procStateUtil.getAllThreadInfo();
    ThreadConsumptionDiff threadConsumptionDiff = new ThreadConsumptionDiff();
    this.threadDiffList = threadConsumptionDiff.calculateDiff(this.preProcState, this.curProcState);
    SimpleDateFormat format = new SimpleDateFormat( pattern: "yyyy年MM月dd日-HH时mm分ss秒");

    for (ThreadConsumptionDiff.ThreadDiff threadDiff : threadDiffList) {
        Date date = new Date(this.startTime);
        threadDiff.startTime = format.format(date);
        date = new Date(this.endTime);
        threadDiff.endTime = format.format(date);
    }
}
```

初步的输出效果如下。可以输出线程在监控时间段内的jiffy消耗。

```
2022-08-17 20:02:42.141 10642-10642/com.example.androidpowercomsumption D/ProcStateUtil: ThreadDiff{jiffiesDiff=25, comm='owercomsumption', state='R', tid=10642,
 startTime=2022年08月17日-20时02分39秒, endTime=2022年08月17日-20时02分42秒}
2022-08-17 20:02:42.141 10642-10642/com.example.androidpowercomsumption D/ProcStateUtil: ThreadDiff{jiffiesDiff=0, comm='Runtime worker ', state='S', tid=10648,
 startTime=2022年08月17日-20时02分39秒, endTime=2022年08月17日-20时02分42秒}
2022-08-17 20:02:42.141 10642-10642/com.example.androidpowercomsumption D/ProcStateUtil: ThreadDiff{jiffiesDiff=0, comm='Runtime worker ', state='S', tid=10649,
 startTime=2022年08月17日-20时02分39秒, endTime=2022年08月17日-20时02分42秒}
2022-08-17 20:02:42.141 10642-10642/com.example.androidpowercomsumption D/ProcStateUtil: ThreadDiff{jiffiesDiff=0, comm='Runtime worker ', state='S', tid=10650,
 startTime=2022年08月17日-20时02分39秒, endTime=2022年08月17日-20时02分42秒}
```

> **Jiffy消耗：**
> 一般来说，当线程的 Jiffy 开销约等于 6000 jiffies / 分钟的时候，就说线程在这段时间内一直在吃 CPU 资源（CPU 中断），换算下来线程的 CPU Load 约为 100%（App 的 CPU Load 为所有线程的 CPU Load 累加，具体值的区间在 [0, 100 × CPU Core Num]）。当 App 在后台的时候，线程持续这种状态超过 10 分钟以上，就说明已经出现待机功耗异常；当时间超过 30min 甚至 1

> hour 以上，就可以在系统电量统计排行上面看到明显的电量曲线的变化。（可以当作判断线程异常的一种规则）

1.2 线程池线程

对于线程池线程的Jiffy 开销异常于普通线程的统计方式不太一样，需要把线程池整体的 Jiffy 消耗分摊到每个 Runnable 上面，而且考虑到同一个 Runable 会反复被执行，还需要把相同 Runnable 的 Jiffy 累计起来。

具体来说，就是对线程池中的每一个Runnable执行的前后统计Jiffy并计算差值，得到当前Runnable的实际开销。同时根据之前提到的Event Slice的统计方案得出每个线程池线程在某一段出现异常的时间窗口里，执行的 Runnable 的占比信息，从而推断出哪些 Runnable 是导致线程耗电的元凶。

2. 检测系统服务调用异常

通过Hook SystemService或者ASM方案来检测系统服务调用异常

调用系统服务的流程如下：
先通过 ServiceManager 的 getService 方法获取远端服务的 IBinder 对象，然后在通过指定服务的 Stub 类的 asInterface 方法转化成本地可使用对象，然后才会去调用系统服务的方法。

根据系统服务调用流程得出实现方案如下：

1. 通过如下代码获得ServiceManager里的原始的系统服务的binder对象。

```Java
Class<?> serviceManager = Class.forName("android.os.ServiceManager");
Method getService = serviceManager.getDeclaredMethod("getService", String.class);
this.baseServiceBinder = (IBinder) getService.invoke(null, serviceName);
```

2. hook掉这个Binder代理的queryLocalInterface 方法。拦截的是 queryLocalInterface() 方法，这个方法返回的是一个远端的服务，还没有转化为本地对象，所以不能去拦截具体的服务方法。我们需要使用反射获取到 serviceClassName$Stub 类，然后反射调用它的 asInterface 方法就可以得到了 serviceClassName$Proxy 对象了，然后才可以去拦截服务具体的方法。

```java
@Override
public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
    if ("queryLocalInterface".equals(method.getName())) {
        Class<?> serviceManagerStubCls = Class.forName(serviceClassName + "$Stub");
        ClassLoader classLoader = serviceManagerStubCls.getClassLoader();
        Method asInterfaceMethod = serviceManagerStubCls.getDeclaredMethod("asInterface", IBinder.class);

        Class<?> serviceManagerCls = Class.forName(serviceClassName);
        final Object originManagerService = asInterfaceMethod.invoke(null, baseServiceBinder);

        return Proxy.newProxyInstance(classLoader,
                new Class[]{IInterface.class},
                new InvocationHandler() {
                    @Override
                    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
                        // todo hook需要的方法
                    }
                });
    }
    return method.invoke(baseServiceBinder, args);
}

```

在这一步中可以拦截服务具体的方法后，针对不同的系统服务，我们可以实现一个对应的InvocationHandler类来对方法进行操作。

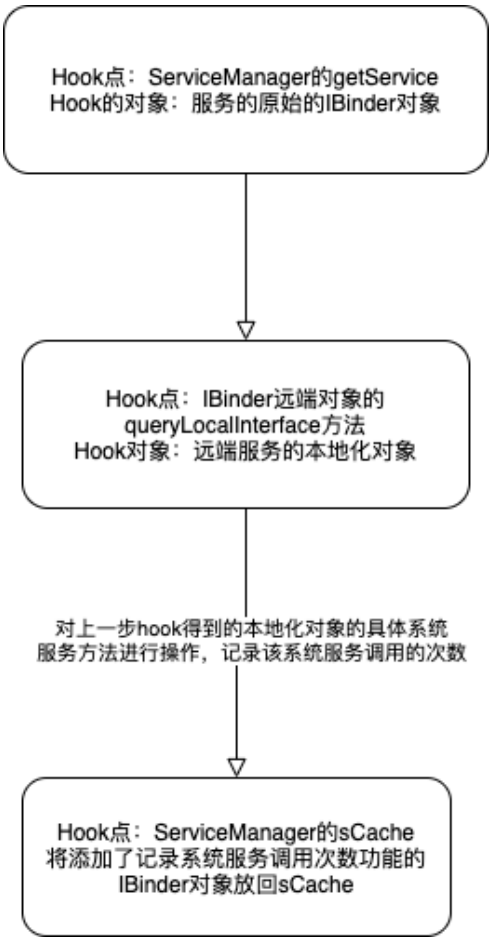
加一个监听器，记录该系统服务调用的次数，在每次调用该系统服务时，将次数加一。就可以达到检测系统服务调用的目的。（这部分在下一阶段会去开发）

3. 放回ServiceManager中，替换掉原有的系统服务的binder对象。

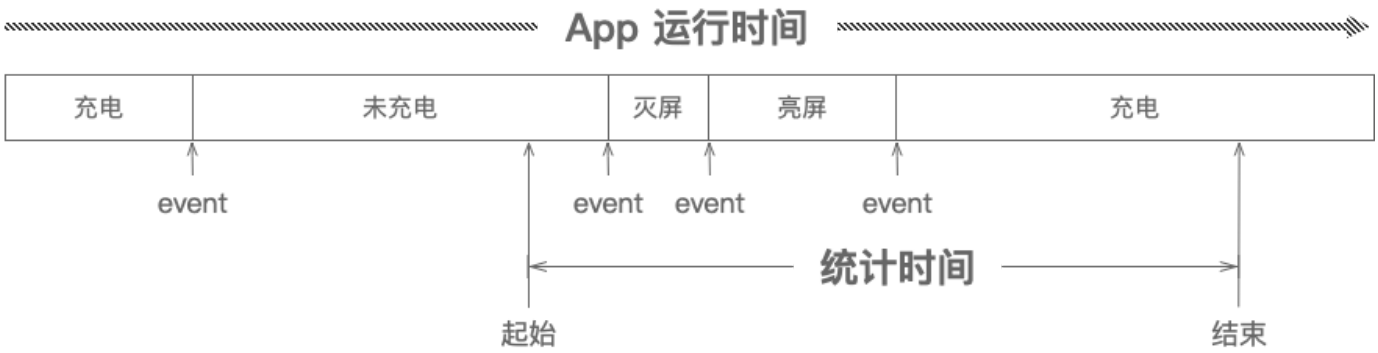
```java
Class<?> serviceManagerCls = Class.forName("android.os.ServiceManager");
Field cacheField = serviceManagerCls.getDeclaredField("sCache");
cacheField.setAccessible(true);
Map<String, IBinder> cache = (Map) cacheField.get(null);
cache.put(serviceName, this.proxyServiceBinder);
```

通过上述方法，我们可以获得在app运行时间内所调用的系统服务以及检测异常的系统服务调用。（比如在短时间内大量重复的调用相同服务，或者调用某个服务响应时间很长等等）

流程图如下：



3. App和设备状态统计



在设备和App状态发生改变的时候，记录下每一种状态的起始时间和结束时间。在监控的时间段内，根据监控窗口的起始时间和结束时间以及记录的这段时间内的app和设备状态，就可以计算出这段时长内 App 每个事件状态的占比。

App状态统计

关于统计 App 状态，核心的逻辑如下。通过ActivityLifecycleCallbacks拿到App所有Activity的生命周期回调。通过重写ActivityLifecycleCallbacks的onActivityStarted、onActivityStopped、onActivityDestroyed方法来监听App的运行状态。初步实现如下。AppStateController是一个控制类，它持有App的前后台运行时间、总运行时间，可以用来计算在监控的时间段内，app的前台运行时间和后台运行时间的占比。

```java
@Override
public void onActivityCreated(Activity activity, Bundle savedInstanceState) {
    // 只会在启动的时候触发一次
    appStateController.start();
    appStateController.status = true; // 前台状态
    appStateController.curStatusStartTime = appStateController.startTime; // 当前状态的开始时间
}

@Override
public void onActivityStarted(Activity activity) {
    Log.d(TAG, "APP进入前台");
    if (!appStateController.status) { // 后台进入前台
        appStateController.status = true;
        appStateController.curStatusEndTime = System.currentTimeMillis(); // 后台状态的结束时间
        appStateController.backgroundTime += (appStateController.curStatusEndTime - appStateController.curStatusStartTime);
        appStateController.curStatusStartTime = System.currentTimeMillis();// 后台进入前台，前台状态的开始时间

    }
}
@Override
public void onActivityStopped(Activity activity) {
    Log.d(TAG, "App进入后台");
    if (appStateController.status) { // 由前台进入后台
        appStateController.status = false;
        appStateController.curStatusEndTime = System.currentTimeMillis(); // 前台状态的结束时间
        appStateController.foregroundTime += (appStateController.curStatusEndTime - appStateController.curStatusStartTime);
        appStateController.curStatusStartTime = System.currentTimeMillis(); // 前台进入后台，后台状态的开始时间
    }
}

@Override
public void onActivityDestroyed(Activity activity) {
    appStateController.finish();
}
```

初步输出效果如下，成功的计算出了app的前后台运行时间和占比。

```
2022-08-17 20:02:39.642 10642-10642/com.example.androidpowercomsumption D/AppStateApplication: APP进入前台
2022-08-17 20:02:42.065 10642-10642/com.example.androidpowercomsumption D/AppStateApplication: App进入后台
2022-08-17 20:02:42.808 10642-10642/com.example.androidpowercomsumption D/AppStateApplication: 前台运行时间:2790
2022-08-17 20:02:42.808 10642-10642/com.example.androidpowercomsumption D/AppStateApplication: 后台运行时间:0
2022-08-17 20:02:42.808 10642-10642/com.example.androidpowercomsumption D/AppStateApplication: 总运行时间:2022年08月17日-20时02分39秒~2022年08月17日-20时02分42秒
2022-08-17 20:02:42.809 10642-10642/com.example.androidpowercomsumption D/AppStateApplication: 0.7896971412397396
2022-08-17 20:02:42.809 10642-10642/com.example.androidpowercomsumption D/AppStateApplication: 0.0
```

设备状态统计

核心的逻辑如下。通过在 `onCreate` 生命周期方法中注册一个自己实现的DeviceStateListener监听器（它会接收设备屏幕状态的广播并调用相应的方法）用来监听屏幕点亮和屏幕熄灭的事件。DeviceStateController是一个控制类，它持有设备的息屏时间、亮屏时间、总的运行时间，根据这些时间可以计算出在监控的时间段内设备的息屏时间和亮屏时间的占比。

```java
listener.register(new DeviceStateListener.ScreenStateListener() {

    boolean isFirst = true; // 第一次启动

    @Override
    public void onScreenOn() {
        Log.d(TAG + "Device", "屏幕点亮");
        if (isFirst) {
            isFirst = false;
            deviceStateController.start();
            deviceStateController.status = true; // 亮屏状态
            deviceStateController.curStatusStartTime = deviceStateController.startTime;
        } else {
            if (!deviceStateController.status) { // 息屏进入亮屏
                deviceStateController.status = true;
                deviceStateController.curStatusEndTime = System.currentTimeMillis(); // 息屏状态的结束时间
                deviceStateController.screenOffTime += (deviceStateController.curStatusEndTime - deviceStateController.curStatusStartTime);
                deviceStateController.curStatusStartTime = System.currentTimeMillis(); // 息屏进入亮屏，亮屏状态的开始时间
            }
        }
    }

    @Override
    public void onScreenOff() {
        Log.d(TAG + "Device", "屏幕熄灭");
        if (deviceStateController.status) { // 亮屏进入息屏
            deviceStateController.status = false;
            deviceStateController.curStatusEndTime = System.currentTimeMillis(); // 亮屏状态的结束时间
            deviceStateController.screenOnTime += (deviceStateController.curStatusEndTime - deviceStateController.curStatusStartTime);
            deviceStateController.curStatusStartTime = System.currentTimeMillis(); // 亮屏进入息屏，息屏状态的开始时间

        }
    }
});
```

初步的输出效果如下。成功的计算出了在App运行阶段设备的息、亮屏时间和占比。

```
2022-08-18 20:26:30.147 13014-13014/? D/AppStateApplicationDevice: 息屏时间:1812
2022-08-18 20:26:30.147 13014-13014/? D/AppStateApplicationDevice: 亮屏时间:5187
2022-08-18 20:26:30.147 13014-13014/? D/AppStateApplicationDevice: 总运行时间:2022年08月18日-20时26分20秒~2022年08月18日-20时26分30秒
2022-08-18 20:26:30.147 13014-13014/? D/AppStateApplicationDevice: 息屏时间占比:0.25889412773253323
2022-08-18 20:26:30.148 13014-13014/? D/AppStateApplicationDevice: 亮屏时间占比:0.7411058722674668
```

4. 生成耗电检测报告

对宿主App从create到destory的生命周期进行实时监控，在App状态发生改变时综合1、2和3中检测到的信息，生成该状态时间段的耗电检测报告。最终以时间轴的形式展示在App的整个生命周期内的电量报告。预计效果如下。重点展示app的线程开销和系统服务调用，同时也会展示一些设备和app的状态。