百度鹰眼 Android SDK v3.0 开发指南

概述

自 2015 年 10 月百度鹰眼 Android SDK v2 版发布以来，开发者们提出了很多宝贵的建议和意见，但受限于 v2 版整体架构以及历史遗留问题，使用上不便的接口无法从根本上解决。

根据去其糟粕、取其精华的指导精神，在保留 v2 版内部逻辑的基础上，对外接口做了全面调整，全新升级推出百度鹰眼 Android SDK v3.0 版（以下简称 SDK），也因此 v3 版将无法兼容 v2 版，由此给各位开发者在升级过程中带来的不便，还请见谅，百度鹰眼将全力支持解决开发者在升级过程中遇到的任何问题。

一、接口说明

1、服务控制

①开启服务【startTrace()】和停止服务【stopTrace()】接口由控制服务和采集调整为只控制服务。

开启服务【startTrace()】只负责建立与维持 TCP 长连接，接收服务端的推送信息。若开启轨迹服务时，SDK 已缓存有轨迹数据，则上传缓存轨迹数据。

②新增开启采集【startGather()】和停止采集【stopGather()】接口。

开启采集后，SDK 将会采集定位依据，若网络连接正常，则达到打包周期后，上传采集的轨迹数据，若网络不可用，则缓存轨迹，待网络状态发生变化，SDK 会发起重连。

停止采集后，SDK 将会停止采集定位依据，并将已采集的轨迹数据上传到服务端，若上传过程中出现网络断开的情况，SDK 会监听网络变化，且在网络状态变化时，发起重连，继续上传轨迹数据。

在停止采集后，若没有停止服务【stopTrace()】，则 SDK 会维持与服务端的 TCP 长连接或监听网络状态，以保证采集的轨迹数据能及时上传到服务端。

③开启服务监听器【OnStartTraceListener】和停止服务监听器【OnStopTraceListener】合并为服务监听器【OnTraceListener】，服务监听器位于 com.baidu.trace.model 包下。

④开启/停止服务、开启/停止采集、推送消息均由 OnTraceListener 监听器接收。

⑤新增自定义数据监听器【OnCustomAttributeListener】，轨迹附加数据通过该监听器的 onTrackAttributeCallback()接口上传，该监听器位于 com.baidu.trace.model 包下。

⑥v2 版中提供了服务类型(TraceType)，分别为：

NO_CONNECT（不建立长连接）；

NO_UPLOAD_LOCATION（只建立长连接）；

UPLOAD_LOCATION（建立长连接，上传轨迹并接收报警推送消息）。

v3 版中去除了服务类型(TraceType)，改为通过接口调用实现服务类型转换，当前开启服务【startTrace()】只负责建立长连接，相当于 v2 中的 NO_UPLOAD_LOCATION。若要实现上传轨迹，则只需再调用开启采集【startGather()】，若想切回只建立长连接模式，则只需调用停止采集【stopGather()】。即通过控制采集，实现在 NO_UPLOAD_LOCATION 和 UPLOAD_LOCATION 模式间的自由切换。

2、对象存储服务（BOS）

①新增对象存储服务，开发者可以根据业务需求来确定是否使用。上传对象接口 putObject()、获取（下载）对象接口 getObject()、生成签名路径接口 generatePresignedUrl()。

②若需要使用 BOS，请导入 bos-android-sdk-1.0.2.jar，并在创建 Trace 时设置对象存储服务标识（isNeedObjectStorage）。

③若不需要使用 BOS，可以不导包，但在创建 Trace 时一定要将对象存储服务标识置为 false，否则会导致初始化存储服务时，由于找不到相应类而 crash。

④存储服务依赖轨迹服务，轨迹服务开启成功后，才会初始化存储服务。因此，上传 getObject()、下载 putObject()两个接口只能在开启服务成功后调用才有效，否则会返回存储服务未初始化；生成签名路径接口 generatePresignedUrl()则不受影响，无论是否开启服务均可调用。

3、api 接口

①SDK 封装了大部分 web api 接口，各接口使用到请求、响应、实体等类定义均在 com.baidu.trace.api 包下，分为轨迹分析(analysis)、对象存储(bos)、终端(Entity)、围栏(fence)、轨迹(track)五大接口。

②各接口监听器均位于对应 api 包下，如围栏监听器 OnFenceListener 位于 com.baidu.trace.api.fence 包下，其他接口监听器依次类推。

③回调接口均运行在 UI 线程中，请不要在回调接口中执行耗时任务。

④接口详情

终端管理:

addEntity()、deleteEntity()、updateEntity()、queryEntityList()

实时位置搜索:

searchEntity()、boundSearchEntity()、aroundSearchEntity()

轨迹分析:

queryStayPoint()、queryDrivingBehavior()
轨迹纠偏和里程:
queryHistoryTrack()、queryDistance()、queryLatestPoint()
地理围栏管理：
createFence()、updateFence()、deleteFence()、queryFenceList()
地理围栏报警：
queryMonitoredStatus()、queryMonitoredStatusByLocation()、
queryFenceHistoryAlarmInfo()

附 web api v3.0 接口总览链接:
http://lbsyun.baidu.com/index.php?title=yingyan/api/v3/all

二、状态说明

1、开启服务（startTrace）

①若未开启网络，则回调消息为【网络未开启】，状态码为【10004】，此时 sdk 会监听网络状态，若网络发生变化则会进行重连。

②若网络不稳定或者网络不通，则回调消息为【网络连接失败】，状态码为【10003】，此时 sdk 也会监听网络状态，待网络联通或者稳定后发起重连。

③若 ak、serviceId 不匹配，导致校验失败，则回调消息为【服务开启失败】，状态码为【10001】，此时 sdk 不会进行任何重连操作。

④若服务已开启，再次开启服务时，当 entityName、serviceId 没有发生变化，此时回调消息为【服务已开启】，状态码为【10006】；当 entityName 或 serviceId 发生改变，则重新登录，回调消息为【成功】，状态码为【0】。

2、停止服务（stopTrace）

①若服务还未开启，则回调消息为【服务未开启】，状态码为【11002】。

②若服务正在停止，则回调消息为【服务正在停止】，状态码为【11003】。

③若网络未开启或者连接不可用，当缓存有轨迹数据时，则回调消息为【服务停止成功，有缓存数据未上传】，状态码为【11004】；若没有缓存轨迹，则回调消息为【成功】，状态码为【0】。

3、开启采集（startGather）

①若已开启服务，首次开启采集时，轨迹服务处理成功，则回调消息为【成功】，状态码为【0】；反之，轨迹服务处理失败，则回调消息为【采集开启失败】，状态码【12001】。

②若服务还未开启，则回调消息为【服务未开启】，状态码为【12002】。

③若已开启采集，再次开启采集时回调消息为【采集已开启】，状态码为【12003】。

4、停止采集（stopGather）

①若已开启采集，首次停止采集时，轨迹服务处理成功，则回调消息为【成功】，状态码为【0】；反之，轨迹服务处理失败，则回调消息为【采集停止失败】，状态码【13001】。

②若服务还未开启，则回调消息为【服务未开启】，状态码为

【13002】。

③若已停止采集，再次停止采集时回调消息为【采集已停止】，状态码为【13003】。

三、使用说明

1、轨迹追踪

对应 demo 中的 TracingActivity。

①初始化轨迹服务，若未创建鹰眼服务，请先在鹰眼轨迹管理台创建服务：http://lbsyun.baidu.com/trace/admin/service，服务创建成功后，系统会生成服务 ID(serviceId)。

```
// 轨迹服务 ID
long serviceId = 0;
// 设备标识
String entityName = "myTrace";
// 是否需要对象存储服务，注意：若需要对象存储服务，一定要导入
bos-android-sdk-1.0.2.jar。
boolean isNeedObjectStorage = false;
// 初始化轨迹服务
Trace mTrace = new Trace(serviceId, entityName,
isNeedObjectStorage);
```

②初始化轨迹服务客户端

```
// 初始化轨迹服务客户端
LBSTraceClient mTraceClient = new
LBSTraceClient(getApplicationContext());
```

③设置采集和打包周期

```
// 采集周期(单位:秒)
int gatherInterval = 5;
// 打包周期(单位:秒)
int packInterval = 5;
// 设置采集和打包周期
mTraceClient.setInterval(gatherInterval, packInterval);
```

④初始化监听器

```
// 初始化轨迹服务监听器
OnTraceListener mTraceListener = new OnTraceListener() {
    // 开启服务回调
    @Override
    public void onStartTraceCallback(int status, String
message) {}
    // 停止服务回调
    @Override
    public void onStopTraceCallback(int status, String
message) {}
    // 开启采集回调
    @Override
    public void onStartGatherCallback(int status, String
message) {}
    // 停止采集回调
    @Override
    public void onStopGatherCallback(int status, String
message) {}
    // 推送回调
    @Override
    public void onPushCallback(byte messageNo, PushMessage
message) {}
};
```

⑤开启服务

```
// 开启服务
mTraceClient.startTrace(mTrace, mTraceListener);
```

⑥开启采集

```
// 开启采集
mTraceClient.startGather(mTraceListener);
```

⑦停止采集

```
// 停止采集
mTraceClient.stopGather(mTraceListener);
```

⑧停止服务

```
// 停止服务
mTraceClient.stopTrace(mTrace, mTraceListener);
```

⑨注意事项

开启服务、开启采集、停止采集、停止服务四个接口共用 OnTraceListener 监听器，每次调用时，请务必传入同一个监听器实例；或者在初始化时调用 LBSTraceClient.setOnTraceListener()设置监听器，后续调用以上四个接口时无须再次传入，如

开启服务 LBSTraceClient.startTrace(mTrace, null)，

开启采集 LBSTraceClient.startGather(null)，

此时回调消息由 setOnTraceListener()时传入的监听器接收。

2、轨迹查询

对应 demo 中的 TrackQueryActivity。

1）查询历史轨迹

①创建历史轨迹请求实例

```
// 请求标识
int tag = 1;
// 轨迹服务 ID
long serviceId = 0;
// 设备标识
String entityName = "myTrace";
// 创建历史轨迹请求实例
HistoryTrackRequest historyTrackRequest = new
HistoryTrackRequest(tag, serviceId, entityName);
```

②设置轨迹起止时间

```
// 开始时间(单位：秒)
long startTime = System.currentTimeMillis() / 1000 - 12 * 60
* 60;
// 结束时间(单位：秒)
long endTime = System.currentTimeMillis() / 1000;
// 设置开始时间
historyTrackRequest.setStartTime(startTime);
// 设置结束时间
historyTrackRequest.setEndTime(endTime);
```

③设置是否纠偏

```
// 设置纠偏
historyTrackRequest.setProcessed(true);
```

④设置纠偏选项（若第三步设置为 true）

```
// 创建纠偏选项实例
ProcessOption processOption = new ProcessOption();
// 设置需要去噪
processOption.setNeedDenoise(true);
// 设置需要抽稀
processOption.setNeedVacuate(true);
// 设置需要绑路
```

```
processOption.setNeedMapMatch(true);
// 设置精度过滤值(定位精度大于 100 米的过滤掉)
processOption.setRadiusThreshold(100);
// 设置交通方式为驾车
processOption.setTransportMode(TransportMode.driving);

// 设置纠偏选项
historyTrackRequest.setProcessOption(processOption);
```

⑤设置里程补充方式

```
// 设置里程填充方式为驾车
historyTrackRequest.setSupplementMode(SupplementMode.driv
ing);
```

⑥初始化轨迹监听器

```
// 初始化轨迹监听器
OnTrackListener mTrackListener = new OnTrackListener() {
    // 历史轨迹回调
    @Override
    public void onHistoryTrackCallback(HistoryTrackResponse
response) {}
};
```

⑦查询历史轨迹

```
// 查询历史轨迹
mTraceClient.queryHistoryTrack(historyTrackRequest,
mTrackListener);
```

2）查询里程

①创建里程请求实例

```
// 请求标识
int tag = 2;
```

```
// 轨迹服务 ID
long serviceId = 0;
// 设备标识
String entityName = "myTrace";

DistanceRequest distanceRequest = new DistanceRequest(tag,
serviceId, entityName);
```

②设置里程起止时间

```
// 开始时间(单位：秒)
long startTime = System.currentTimeMillis() / 1000 - 12 * 60
* 60;
// 结束时间(单位：秒)
long endTime = System.currentTimeMillis() / 1000;
// 设置开始时间
distanceRequest.setStartTime(startTime);
// 设置结束时间
distanceRequest.setEndTime(endTime);
```

③设置是否纠偏

```
// 设置纠偏
distanceRequest.setProcessed(true);
```

④设置纠偏选项（若第三步设置为 true）

```
// 创建纠偏选项实例
ProcessOption processOption = new ProcessOption();
// 设置需要去噪
processOption.setNeedDenoise(true);
// 设置需要绑路
processOption.setNeedMapMatch(true);
// 设置交通方式为驾车
processOption.setTransportMode(TransportMode.driving);
```

```
// 设置纠偏选项
distanceRequest.setProcessOption(processOption);
```

⑤设置里程补充方式

```
// 设置里程填充方式为驾车
distanceRequest.setSupplementMode(SupplementMode.driving)
;
```

⑥初始化轨迹监听器

```
// 初始化轨迹监听器
OnTrackListener mTrackListener = new OnTrackListener() {
    // 里程回调
    @Override
    public void onDistanceCallback(DistanceResponse response)
{}
};
```

⑦查询里程

```
// 查询里程
mTraceClient.queryDistance(distanceRequest,
mTrackListener);
```

3、地理围栏

对应 demo 中的 FenceActivity。

1）创建本地圆形围栏

①初始化请求参数

```
// 请求标识
int tag = 3;
// 轨迹服务 ID
long serviceId = 0;
// 围栏名称
```

```
String fenceName = "local_circle";
// 监控对象
String monitoredPerson = "myTrace";
// 围栏圆心
com.baidu.trace.model.LatLng center = new
com.baidu.trace.model.LatLng(39.9151190000,
116.4039630000);
// 围栏半径（单位 : 米）
double radius = 2000;
// 去噪精度
int denoise = 200;
// 坐标类型
CoordType coordType = CoordType.bd09ll;
```

②创建本地圆形围栏请求实例

```
// 创建本地圆形围栏请求实例
CreateFenceRequest localCircleFenceRequest =
CreateFenceRequest.buildLocalCircleRequest(tag, serviceId,
       fenceName, monitoredPerson, center, radius, denoise,
coordType);
```

③初始化围栏监听器

```
// 初始化围栏监听器
OnFenceListener mFenceListener = new OnFenceListener() {
    // 创建围栏回调
    @Override
    public void onCreateFenceCallback(CreateFenceResponse
response) {}
    // 更新围栏回调
    @Override
    public void onUpdateFenceCallback(UpdateFenceResponse
```

```
response) {}
    // 删除围栏回调
    @Override
    public void onDeleteFenceCallback(DeleteFenceResponse
response) {}
    // 围栏列表回调
    @Override
    public void onFenceListCallback(FenceListResponse
response) {}
    // 监控状态回调
    @Override
    public void
onMonitoredStatusCallback(MonitoredStatusResponse
response) {}
    // 指定位置监控状态回调
    @Override
    public void
onMonitoredStatusByLocationCallback(MonitoredStatusByLoca
tionResponse response) {}
    // 历史报警回调
    @Override
    public void onHistoryAlarmCallback(HistoryAlarmResponse
response) {}
};
```

④请求创建本地圆形围栏

```
// 创建本地圆形围栏
mTraceClient.createFence(localCircleFenceRequest,
mFenceListener);
```

2）创建服务端圆形围栏

①初始化请求参数

```
// 请求标识
int tag = 4;
// 轨迹服务 ID
long serviceId = 0;
// 围栏名称
String fenceName = "server_circle";
// 监控对象
String monitoredPerson = "myTrace";
// 围栏圆心
com.baidu.trace.model.LatLng center = new
com.baidu.trace.model.LatLng(39.9151190000,
116.4039630000);
// 围栏半径（单位 : 米）
double radius = 2000;
// 去噪精度
int denoise = 200;
// 坐标类型
CoordType coordType = CoordType.bd0911;
```

②创建服务端圆形围栏请求实例

```
// 创建服务端圆形围栏请求实例
CreateFenceRequest serverCircleFenceRequest =
CreateFenceRequest.buildServerCircleRequest(tag, serviceId,
       fenceName, monitoredPerson, center, radius, denoise,
coordType);
```

③初始化围栏监听器

同本地围栏。

④请求创建服务端圆形围栏

```
// 创建服务端圆形围栏
mTraceClient.createFence(serverCircleFenceRequest,
mFenceListener);
```

4、图片服务

对应 demo 中的 BosActivity。

1）上传对象（图片）

①初始化请求参数

```
// 请求标识
int tag = 5;
// 轨迹服务 ID
long serviceId = 0;
// 对象 key（即文件名称包括后缀，如 track.jpg、track.png）
String objectKey = "track.jpg";
// 对象类型
BosObjectType objectType = BosObjectType.image;
// 通过文件形式上传
String path = "图片路径";
File file = new File(path);
```

②初始化 BOS 监听器

```
// 初始化监听器
OnBosListener mBosListener = new OnBosListener() {
    // 上传对象（图片）回调
    @Override
    public void onPutObjectCallback(BosPutObjectResponse
response) {}
```

```
    // 获取对象（图片）回调
    @Override
    public void onGetObjectCallback(BosGetObjectResponse
response) {}
    // 生成签名对象（图片）URL
    @Override
    public void
onGeneratePresignedUrlCallback(BosGeneratePresignedUrlRes
ponse response) {}
};
```

③创建上传对象（图片）请求实例

```
// 创建上传对象（图片）请求实例
BosPutObjectRequest request =
BosPutObjectRequest.buildFileRequest(tag, serviceId,
       objectKey, objectType, file);
```

④请求上传对象（图片）

```
// 请求上传对象（图片）
mTraceClient.putObject(request, mBosListener);
```

2）获取对象（图片）

①初始化请求参数

```
// 请求标识
int tag = 6;
// 轨迹服务 ID
long serviceId = 0;
// 对象 key（即文件名称包括后缀，如 track.jpg、track.png）
String objectKey = "track.jpg";
```

```
// 对象类型
BosObjectType objectType = BosObjectType.image;
```

②创建获取对象（图片）实例

```
// 创建获取对象（图片）请求实例
BosGetObjectRequest request = new BosGetObjectRequest(tag,
       serviceId, objectKey, objectType);
```

③初始化 BOS 监听器

同上传对象（图片）。

④请求获取对象（图片）

```
// 请求获取对象（图片）
mTraceClient.getObject(request, mBosListener);
```

3）生成签名对象（图片）URL

①初始化请求参数

```
// 请求标识
int tag = 7;
// 轨迹服务 ID
long serviceId = 0;
// 对象 key（即文件名称包括后缀，如 track.jpg、track.png）
String objectKey = "track.jpg";
// 对象类型
BosObjectType objectType = BosObjectType.image;
```

②创建请求实例

```
// 创建请求实例
BosGeneratePresignedUrlRequest request = new
BosGeneratePresignedUrlRequest(tag, serviceId, objectKey,
objectType);
```

③设置图片处理命令

```
// 图片处理命令
ImageProcessCommand imageProcessCommand = new
ImageProcessCommand();
imageProcessCommand.setAngle(180);
request.setImageProcessCommand(imageProcessCommand);
```

④设置文字水印命令

```
// 文字水印命令
TextWatermarkCommand textWatermarkCommand = new
TextWatermarkCommand();
textWatermarkCommand.setText("百度鹰眼");
textWatermarkCommand.setFontFamily(FontFamily.KaiTi);
textWatermarkCommand.setAngle(45);
textWatermarkCommand.setFontColor("0000FF");
request.setTextWatermarkCommand(textWatermarkCommand);
```

⑤初始化 BOS 监听器

同上传对象（图片）。

⑥请求生成签名对象（图片）URL

```
// 请求生成签名对象（图片）URL
mTraceClient.generatePresignedUrl(request, mBosListener);
```

5、缓存管理

对应 demo 中的 CacheManageActivity。

1）查询缓存轨迹

①初始化请求参数

```
// 请求标识
int tag = 8;
// 轨迹服务 ID
long serviceId = 0;
// 设备标识
String entityName = "myTrace";
```

②初始化轨迹监听器

```
// 初始化轨迹监听器
OnTrackListener  mTrackListener = new OnTrackListener() {
    // 缓存轨迹回调
    @Override
    public void onQueryCacheTrackCallback(
QueryCacheTrackResponse response) {}
};
```

③创建缓存轨迹请求实例

```
// 创建缓存轨迹请求实例
QueryCacheTrackRequest request = new
QueryCacheTrackRequest(tag, serviceId, entityName);
```

④查询缓存轨迹

```
// 查询缓存轨迹
mTraceClient.queryCacheTrack(request, mTrackListener);
```

2）清除缓存轨迹

①初始化请求参数

```
// 请求标识
int tag = 9;
// 轨迹服务 ID
long serviceId = 0;
// 要查询的 entityName
```

```
String entityName = "myTrace";
List<String> entityNames = new ArrayList<>();
entityNames.add(entityName);
```

②初始化轨迹监听器

```
// 初始化轨迹监听器
OnTrackListener  mTrackListener = new OnTrackListener() {
    // 清除缓存轨迹回调
    @Override
    public void
onClearCacheTrackCallback(ClearCacheTrackResponse response)
{}
};
```

③创建清除缓存轨迹请求实例

```
// 创建清除缓存轨迹请求实例
ClearCacheTrackRequest request = new
ClearCacheTrackRequest(tag, serviceId, entityNames);
```

④请求清除缓存轨迹

```
// 请求清除缓存轨迹
mTraceClient.clearCacheTrack(request, mTrackListener);
```

四、补充说明

1、doze 模式

Android M 中引入了 doze 模式，若手机厂商生产的定制机型中使用到该模式，需要申请将 app 添加进白名单，代码如下，demo 在 TracingActivity 的 onResume() 方法中实现：

```
// 在 Android 6.0 及以上系统，若定制手机使用到 doze 模式，请求将应用添加到白名单。
```

```
if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.M) {
    String packageName = trackApp.getPackageName();
    boolean isIgnoring =
powerManager.isIgnoringBatteryOptimizations(packageName);
    if (!isIgnoring) {
        Intent intent = new Intent(
Settings.ACTION_REQUEST_IGNORE_BATTERY_OPTIMIZATIONS);
        intent.setData(Uri.parse("package:" + packageName));
        try {
            startActivity(intent);
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }
}
```

清单文件中添加权限声明：

```
<uses-permission
android:name="android.permission.REQUEST_IGNORE_BATTERY_O
PTIMIZATIONS"/>
```

2、动态权限申请

Android M 中敏感权限需要动态申请，在 app 初始化完成后，需检测 app 是否有相应权限，具体代码可以参考 MainActivity 中的 onStart()方法。

3、状态码

com.baidu.trace.model.StatusCodes 类中定义了 api、服务、采集、GPS 等状态码及消息内容。

4、配置 AK

在 AndroidManifest.xml 中 application 标签下配置 meta-data，ak 申请地址：http://lbsyun.baidu.com/apiconsole/key

```
<application
    android:allowBackup="false"
    android:icon="@mipmap/app_icon"
    android:label="@string/app_name"
    android:theme="@style/AppTheme">
    <meta-data
        android:name="com.baidu.lbsapi.API_KEY"
        android:value="开发者申请的 Android 端 AK" />
</application>
```

5、监听器

v2 版中 api 接口共享 listener，LBSTraceClient 中会将 listener 保存为静态成员变量。而在 v3 版中，LBSTraceClient 不会再存储 api 接口对应的 listener，调用 api 接口时，均需要传入 listener，响应结果通过传入的 listener 进行回调。

再次提醒：OnTraceListener 仍然共享同一个，开启/停止服务、开启/停止采集时，若传入的 OnTraceListener 实例不为空，则更新 LBSTraceClient 中 OnTraceListener 成员变量，若传入的 OnTraceListener 实例为空，则使用 OnTraceListener 成员变量，若 OnTraceListener 成员变量也未设置，则不回调。因此可以通过 LBSTraceClient.setOnTraceListener()接口进行统一设置。

6、demo 工程说明

①com.baidu.mapapi.clusterutil 包为点聚合源码，用于将多个围栏聚

合在一起。

②com.baidu.track 包为鹰眼相关功能源码。

③轨迹服务 ID(serviceId)在 TrackApplication 类中设置。

7、问题反馈

开发过程中遇到任何问题，即可以在论坛百度鹰眼版块反馈：
http://bbs.lbsyun.baidu.com/forum.php?mod=forumdisplay&fid=26，
也可以发送邮件到 baiduyingyan <baiduyingyan@baidu.com>。