概述

SDK简介

百度鹰眼轨迹iOS SDK是一套基于iOS 8.0及以上版本设备的轨迹服务应用程序接口。配合鹰眼轨迹产品，您可以开发适用于移动设备的轨迹追踪应用，轻松实现实时轨迹追踪、历史轨迹查询、地理围栏报警等功能。该套SDK提供的服务是免费的，任何非营利性程序均可使用，您需要先申请密钥，才可使用该套SDK接口。任何非营利性产品请直接使用，商业目的产品使用前请参考使用须知 。
在您使用百度鹰眼轨迹iOS SDK之前，请先阅读百度地图API使用条款。

功能总览

轨迹追踪与上传

鹰眼iOS SDK可以根据开发者自定义的定位周期和上传周期，自动地采集设备的位置信息，回传到鹰眼服务端，形成连续的轨迹。开发者可以为每个轨迹点添加附加信息，也可以对定位选项、采集周期与上传周期、缓存容量等进行自定义设置。SDK支持后台运行，断网时自动缓存，网络恢复后自动重连上传。SDK与服务端采用TCP长连接与HTTPS通信，所有数据经过压缩和加密，保障数据安全，省电省流量。

轨迹纠偏与查询

鹰眼iOS SDK支持实时位置查询、历史轨迹查询、里程计算等功能。所有轨迹查询都支持纠偏，解决定位漂移问题，。历史轨迹和里程计算还支持里程补偿，还原真实轨迹。

终端管理与检索

鹰眼iOS SDK提供了针对终端实体的增、删、改、查等接口对其进行管理，同时支持按照关键字、矩形区域、圆形区域等方式对终端实体的实时位置进行检索。

地理围栏

鹰眼iOS SDK提供了客户端地理围栏和服务端地理围栏两种围栏服务。客户端和服务端的地理围栏都可以提供了增、删、改、查等操作对围栏进行管理，同时还可以查询指定监控对象和围栏的位置关系、历史报警等信息。

版本说明

自2015年10月百度鹰眼iOS SDK v2版本发布以来，开发者们提出了很多宝贵的意见和建议。v2版本使用Swift语言开发，由于其ABI不稳定，导致开发者在导入鹰眼SDK时遇到了各种各样的问题，也经常无法第一时间适配最新版的Xcode和iOS版本，v3版SDK使用ObjC语言进行了重构。同时，针对开发者们反馈的问题，v3版SDK对接口也进行了重新设计，使其能适应更广泛地使用场景。因此v3版本SDK将无法完全兼容v2版本，由此给各位开发者在升级过程中带来的不便，还请见谅。百度鹰眼团队将全力解决开发者在升级过程中遇到的问题。

使用步骤

申请密钥

开发者在使用鹰眼iOS SDK前需要获取百度地图移动版开发密钥（ak），该密钥与开发者的百度账户相关联。因此开发者需要先有百度账户，才能申请密钥（ak）。

基础概念

- ak 从API控制台申请应用时分配的密钥，形式为由大小写字母与数字组成的字符串
- mcode 申请iOS类型的应用时填写的安全码，和Xcode中APP的Bundle Identifier一致即可。
- serviceID 从鹰眼轨迹管理台创建鹰眼服务时由系统分配的鹰眼服务的唯一标识符

申请步骤

1. 使用开发者的百度账户登录API控制台。
2. 点击创建应用，开始申请开发密钥。
3. 填写应用名称、应用类型注意选择“iOS SDK”、启用服务中勾选【鹰眼API】。
4. 将Xcode中APP的Bundle Identifier

创建鹰眼轨迹服务

使用任何鹰眼轨迹服务前，每个开发者必须先创建自有的鹰眼轨迹服务空间并获取 service_id，用于存储、访问和管理自己的终端和轨迹。 一个service对应开发者的一套轨迹管理系统，如：一个物流公司的车辆管理系统。每个service最多可管理100万终端（人、车等），一个开发者最多可创建10个 service。创建轨迹服务分为2步：

1. 进入鹰眼轨迹管理台，点击"创建服务"按钮，在弹窗中填写信息后完成服务创建。
2. 获取 serviceID。创建服务成功后，在"我创建的服务"列表中即可看到新增的 service。列表左侧的"系统 ID（service_id）"，如：128658，即为访问该service的唯一标识，在后续的接口调用中均要使用。

配置工程

与V2版SDK相比，V3版SDK不再区分开发版和上架版，而只提供一个版本，支持所有的模拟器和真机CPU架构，开发者可以使用`lipo`命令，根据自身的需要合成自己所需的SDK。同时与V2版相比，V3版SDK的导入过程也简化了很多。

合成自己需要的CPU架构（可选）

可以直接使用鹰眼iOS SDK进行开发测试，但是上架到APP STORE时，由于苹果公司的要求，只能使用真机对应CPU架构的SDK，所以需要开发者需要使用`lipo`命令合成自身需要的CPU架构版本的SDK。`lipo`命令的使用方法如下：

第1步：检查当前库支持的CPU架构

> 输入命令：

```
lipo -info ./BaiduTraceSDK.framework/BaiduTraceSDK
```

> 终端回显：

```
Architectures in the fat file: ./BaiduTraceSDK.framework/BaiduTraceSDK are: i386 x86_64
armv7 armv7s arm64
```

第2步：剥离你需要的CPU架构，这里以你需要arm64和armv7为例：

> 输入命令：

```
mkdir armv7
mkdir arm64
lipo ./BaiduTraceSDK.framework/BaiduTraceSDK -thin arm64 -output ./arm64/BaiduTraceSDK
lipo ./BaiduTraceSDK.framework/BaiduTraceSDK -thin armv7 -output ./armv7/BaiduTraceSDK
```

第3步：检查上一步剥离出来的库支持的CPU架构是否符合你的要求：

> 输入命令：

```
lipo -info ./arm64/BaiduTraceSDK
```

> 终端回显：

```
Non-fat file: ./arm64/BaiduTraceSDK is architecture: arm64
```

> 输入命令：

```
lipo -info ./armv7/BaiduTraceSDK
```

> 终端回显：

```
Non-fat file: ./armv7/BaiduTraceSDK is architecture: armv7
```

第4步：将剥离出来的库合并成你需要的库，并替换framework中的库

> 输入命令：

```
lipo -create ./armv7/BaiduTraceSDK ./arm64/BaiduTraceSDK -output
./BaiduTraceSDK.framework/BaiduTraceSDK
```

第5步：清理刚才过程中的中间产物：

> 输入命令：

```
rm -rf ./arm64
rm -rf ./armv7
```

第6步：检查framework中的库支持的CPU架构

> 输入命令：

```
lipo -info ./BaiduTraceSDK.framework/BaiduTraceSDK
```

> 终端回显：

```
Architectures in the fat file: ./BaiduTraceSDK.framework/BaiduTraceSDK are: armv7 arm64
```

第7步：完成。

导入SDK

选择需要的Target，将 BaiduTracekSDK.framework 拖入其General选项卡中的Embedded Binaries栏即可。拖入之后Linked Framework and Libraries栏也会显示 BaiduTraceSDK.framework 项，同时检查 Build Phases 选项卡中的 Embed Frameworks 栏和 Link Binary With Libraries 栏也都显示成功导入 BaiduTraceSDK.framework 项。

plist文件设置

鹰眼iOS SDK需要使用后台定位权限，因此请在APP 的info.plist文件源码中增加以下声明：

```
<key>NSLocationAlwaysUsageDescription</key>
    <string>需要持续定位</string>
<key>UIBackgroundModes</key>
<array>
    <string>location</string>
</array>
```

编译选项设置

- 鹰眼iOS SDK目前暂不支持Bitcode，请确保 Build Settings 选项卡中的 Enable Bitcode 项设置为No。
- 鹰眼iOS SDK最低支持iOS 8.0系统，请确保 Build Settings 选项卡中的 iOS Deployment Target 不低于iOS8.0。

功能介绍

鹰眼iOS SDK中主要包括5大类接口，涵盖了服务（service）与采集（gather）、终端实体（entity）、轨迹（track）、地理围栏（fence）、轨迹分析（analysis）这5大主题。

响应的格式

每个主题中的各个功能，都通过各自的Action类发起请求，通过各自的Delegate回调获取响应。除了轨迹服务和轨迹的采集的控制接口外，其他的响应都为NSData类型，开发者可以按照 `JSON` 格式将其解析为 `NSDictionary` 类型，如下面代码片段所示：

```
-(void)onQueryTrackLatestPoint:(NSData *)response {
    NSDictionary *dict = [NSJSONSerialization JSONObjectWithData:response
options:NSJSONReadingAllowFragments error:nil];
    NSLog(@"track latestpoint response: %@", dict);
}
```

而轨迹服务的开启和结束、采集的开始和停止的操作结果，通过错误码的形式返回。

轨迹服务相关操作执行结果的错误码由 `BTKServiceErrorCode` 枚举定义：

```
/**
 轨迹服务相关操作执行结果的错误码

 - BTK_START_SERVICE_SUCCESS: 服务开启成功，并与服务端连接成功
 - BTK_START_SERVICE_SUCCESS_BUT_OFFLINE: 服务开启成功，但与服务端连接失败，SDK会尝试重连
 - BTK_START_SERVICE_PARAM_ERROR: 参数错误
 - BTK_START_SERVICE_INTERNAL_ERROR: 内部错误
 - BTK_START_SERVICE_NETWORK_ERROR: 网络异常
 - BTK_START_SERVICE_AUTH_ERROR: 鉴权不通过导致失败(ak mcode等信息错误)
 - BTK_START_SERVICE_IN_PROGRESS: 正在开启服务
 - BTK_SERVICE_ALREADY_STARTED_ERROR: 已经开启服务，请勿重复开启
 - BTK_STOP_SERVICE_NO_ERROR: 停止服务成功
 - BTK_STOP_SERVICE_NOT_YET_STARTED_ERROR: 服务还未开启，无法停止
 - BTK_STOP_SERVICE_IN_PROGRESS: 正在停止服务
 */
typedef NS_ENUM(NSUInteger, BTKServiceErrorCode) {
    BTK_START_SERVICE_SUCCESS,
    BTK_START_SERVICE_SUCCESS_BUT_OFFLINE,
    BTK_START_SERVICE_PARAM_ERROR,
    BTK_START_SERVICE_INTERNAL_ERROR,
    BTK_START_SERVICE_NETWORK_ERROR,
```

```
    BTK_START_SERVICE_AUTH_ERROR,
    BTK_START_SERVICE_IN_PROGRESS,
    BTK_SERVICE_ALREADY_STARTED_ERROR,
    BTK_STOP_SERVICE_NO_ERROR,
    BTK_STOP_SERVICE_NOT_YET_STARTED_ERROR,
    BTK_STOP_SERVICE_IN_PROGRESS,
};
```

采集相关操作执行结果的错误码，由 `BTKGatherErrorCode` 枚举定义：

```
/**
 采集相关操作执行结果的错误码

 - BTK_START_GATHER_SUCCESS: 开始采集成功
 - BTK_GATHER_ALREADY_STARTED_ERROR: 已经在采集，请勿重复开始
 - BTK_START_GATHER_BEFORE_START_SERVICE_ERROR: 开始采集必须在开始服务之后调用
 - BTK_START_GATHER_LOCATION_SERVICE_OFF_ERROR: 开始采集由于系统定位服务未开启而失败
 - BTK_START_GATHER_LOCATION_ALWAYS_USAGE_AUTH_ERROR: 开始采集由于没有后台定位权限而失败
 - BTK_START_GATHER_INTERNAL_ERROR: 开始采集由于内部错误而失败
 - BTK_STOP_GATHER_NO_ERROR: 停止采集成功
 - BTK_STOP_GATHER_NOT_YET_STARTED_ERROR: 停止采集必须在开始采集之后调用
 */
typedef NS_ENUM(NSUInteger, BTKGatherErrorCode) {
    BTK_START_GATHER_SUCCESS,
    BTK_GATHER_ALREADY_STARTED_ERROR,
    BTK_START_GATHER_BEFORE_START_SERVICE_ERROR,
    BTK_START_GATHER_LOCATION_SERVICE_OFF_ERROR,
    BTK_START_GATHER_LOCATION_ALWAYS_USAGE_AUTH_ERROR,
    BTK_START_GATHER_INTERNAL_ERROR,
    BTK_STOP_GATHER_NO_ERROR,
    BTK_STOP_GATHER_NOT_YET_STARTED_ERROR,
};
```

下面详细介绍每个主题中各个接口的用法。

服务与采集

服务与采集控制相关的功能是鹰眼SDK中最基础、最核心的功能。通过类BTKAction中相应的接口，控制轨迹服务与轨迹采集的开启和停止，操作结果通过BTKTraceDelegate协议中相应的方法回调给开发者。下面首先对涉及到的基本概念进行介绍，之后会对各功能进行阐述。

基础概念

轨迹服务与轨迹采集

1. 轨迹服务的含义：轨迹服务负责和服务端建立并保持长连接，接收服务端的推送消息，并将客户端的轨迹数据尽快地上传到服务端。
2. 轨迹采集的含义：SDK通过访问iOS系统定位服务获取当前设备的位置信息。
3. 服务与采集的关系：与采集相比，服务是一个更广的概念。虽然采集的开启与结束和服务的开启与结束是独立控制的，但他们的调用顺序也有一定的要求：调用startGather和stopGather之前必须先调用startService，并且调用stopService时，SDK会自动stopGather。一个典型的使用场景是：

   > startService -> startGather -> stopGather -> startGather -> stopGather -> ... -> stopService

   也就是开启服务之后，在需要采集轨迹的时候就调用startGather，不需要采集轨迹的时候就调用stopGather，最后停止服务。

采集周期与上传周期

SDK以采集周期为间隔，周期性地采集当前设备的位置信息；以上传周期为间隔，周期性地将若干轨迹点上传至鹰眼服务端。

后台保活

保活指的是当SDK在后台运行时不被系统杀死。如果选择不保活，当SDK只是开启了服务而没有开启采集时，由于没有访问定位服务，在后台运行时，可能会被系统杀死；如果选择保活，开启服务之后，即使没有开启采集，SDK也会访问系统的定位服务，使其在后台运行时不被系统杀死。有3点需要注意的是：

1. 保活时由于会访问系统定位服务，因此在系统的状态栏上会显示定位图标，但此时SDK并不会采集设备的位置。只要没有开启采集，当前设备的位置就不会被上传到鹰眼服务端。
2. 保活目的下访问系统定位服务的定位精度较低、距离阈值较大，因此几乎不会额外增加耗电量。
3. 由于iOS优先保证前台APP的资源，保活只是减小后台运行被杀的概率，不保证系统资源紧张时，SDK在后台运行一定不会被杀掉。

服务初始化

通过BTKAction类的 `-(BOOL)initInfo:(BTKServiceOption *)option;` 方法设置SDK运行所需要的基础信息。包括ak、mcode、轨迹服务的ID以及是否需要保活。调用SDK的任何接口之前，必须先调用此方法，否则SDK的各功能无法正常运行。

以下代码片段表示，设置SDK运行时需要的基础信息。

```
BTKServiceOption *sop = [[BTKServiceOption alloc] initWithAK:@"asdf1234asdf1234"
mcode:@"com.yingyan.sdk" serviceID:100000 keepAlive:false];
[[BTKAction sharedInstance] initInfo:sop];
```

轨迹服务控制

轨迹服务的控制分为开启轨迹服务（startService）和停止轨迹服务（stopService）两个接口。

开启服务

通过`BTKAction`类的 `-(void)startService:(BTKStartServiceOption *)option delegate:(id <BTKTraceDelegate>)delegate;` 方法开启轨迹服务，操作结果通过 `BTKTraceDelegate` 协议的 `-(void)onStartService:(BTKServiceErrorCode) error;` 方法回调给开发者。
开启服务时需要指定`BTKStartServiceOption`类型的配置信息，目前`BTKStartServiceOption` 类中只有entityName一个属性。该entityName在SDK的运行过程中有举足轻重的作用：

- 开启轨迹服务之后，SDK会以此entityName的身份登录到服务端；
- 在轨迹服务运行期间，收到的地理围栏报警信息都来自于被监控对象为此entityName的地理围栏；
- 在轨迹服务运行期间，采集到的轨迹数据也隶属于此entityName名下；

开启服务之后，如果有之前遗留的缓存数据，也会将其全部上传至服务端（所有entity的缓存数据都会上传）。

以下代码片段表示，开启轨迹服务。

```
// 设置开启轨迹服务时的服务选项，指定本次服务以“entityA”的名义开启
BTKStartServiceOption *op = [[BTKStartServiceOption alloc]
initWithEntityName:@"entityA"];

// 开启服务
[[BTKAction sharedInstance] startService:op delegate:self];
```

停止服务

停止轨迹服务时，SDK会和服务端断开连接。在此之前，如果当前登录的entity有断网缓存的数据，则将其上传至服务端。以上操作均要求网络畅通，否则只是停止轨迹服务，缓存数据会由SDK持久化保存，待下次开启轨迹服务后，网络畅通的情况下，由SDK自动继续上传。

以下代码片段表示，停止轨迹服务。

```
1 [[BTKAction sharedInstance] stopService:self];
```

轨迹采集控制

轨迹采集的控制分为开启采集(startGather)和结束采集(stopGather)两个接口。开启采集和结束采集都必须在开启轨迹服务之后才可以调用。开启采集之后，SDK会在每个采集周期获取当前设备的位置信息，停止采集之后，SDK 不再获取当前设备的位置信息。

开始采集

通过`BTKAction`类的`-(void)startGather:(id <BTKTraceDelegate>)delegate;`方法开始采集轨迹，操作结果通过`BTKTraceDelegate`协议的`-(void)onStartGather:(BTKGatherErrorCode) error;`方法回调给开发者。

以下代码片段表示，开始采集。

```
1 [[BTKAction sharedInstance] startGather:self];
```

停止采集

通过`BTKAction`类的`-(void)stopGather:(id <BTKTraceDelegate>)delegate;`方法停止采集轨迹，操作结果通过`BTKTraceDelegate`协议的`-(void)onStopGather:(BTKGatherErrorCode) error;`方法回调给开发者。

以下代码片段表示，停止采集。

```
1 [[BTKAction sharedInstance] stopGather:self];
```

自定义字段上传

每个轨迹点都可能有一些和具体业务相关的数据，鹰眼iOS SDK支持开发者将这些数据作为轨迹点的自定义数据，“附加”到这个轨迹点上。SDK在每个采集周期会回调`BTKTraceDelegate`协议中的`-(NSDictionary *)onGetCustomData;`方法，将其返回值作为当前采集周期采集的轨迹点的自定义字段的值。SDK会解析此方法返回的字典，将key认为是自定义字段的名称，将value认为是自定义字段的值。需要注意的是，字典中的key必须是已经存在的`track属性字段`。开发者可以通过鹰眼轨迹管理台，设置某个service的`track属性字段`。下面我们通过一个例子，来进一步说明如何上传轨迹点的自定义字段值：

比如当前在开发一款跑步类的APP，可能需要记录每个轨迹点所对应的心率信息，那么可以通过以下几个步骤实现这一点。

1. 通过 鹰眼轨迹管理台，为当前service添加一个`track属性字段`，其`字段名称`为 'heart_rate'，`字段描述`为“心率”，`字段类型`为 int。
2. 通过心率带或带有心率检测功能的外部设备获取心率，并通过其相应的接口将心率数据传入APP。
3. 定义一个字典类型的变量，包含一个名为heart_rate的键，并设置其初始值。假设为`additional_data['heart_rate'] = 60`。
4. 每当APP获取到最新的心率数据时，更新这个字典中heart_rate的值为最新的心率数据。
5. `-(NSDictionary *)onGetCustomData;`方法的实现非常简单，只需要return这个字典即可。

6. 通过以上步骤，SDK在每个采集周期回调这个方法时，就获取到了最新的心率数据，并将此心率数据“附加”在了当前的轨迹点上。

无论是通过 queryEntityList方法查询实时位置，还是通过getTrackHistory方法查询历史轨迹，这些心率数据都会作为轨迹的“附加”数据呈现出来。

补充说明一点，每个采集周期对应轨迹点的速度、方向、海拔高度、定位精度的值，作为轨迹数据的一部分，SDK会自动采集上传，不需要额外实现此方法去指定。

以下代码示例表示，当前开启轨迹服务时指定的serviceID下，已经在轨迹管理台中添加了名为“intTest”、“doubleTest”、“stringTest” 这3个track属性字段，在每个采集周期采集的轨迹点，都将这3个字段赋予一个随机值，“附加”在该轨迹点上。

```
-(NSDictionary *)onGetCustomData {
    NSMutableDictionary *customData = [NSMutableDictionary dictionaryWithCapacity:3];
    NSArray *intPoll = @[@5000, @7000, @9000];
    NSArray *doublePoll = @[@3.5, @4.6, @5.7];
    NSArray *stringPoll = @[@"aaa", @"bbb", @"ccc"];
    int randomNum = arc4random() % 3;
    // intTest doubleTest stringTest这3个自定义字段需要在轨迹管理台提前设置才有效
    customData[@"intTest"] = intPoll[randomNum];
    customData[@"doubleTest"] = doublePoll[randomNum];
    customData[@"stringTest"] = stringPoll[randomNum];
    return [NSDictionary dictionaryWithDictionary:customData];
}
```

消息推送回调

在开启轨迹服务之后，停止轨迹服务之前，鹰眼iOS SDK在运行期间内会随时通过 `BTKTraceDelegate` 协议的 `-(void)onGetPushMessage:(BTKPushMessage *)message;` 回调方法，向开发者推送消息。推送的消息为 `BTKPushMessage` 类型，其中`type` 属性代表推送消息的类型，`content` 属性代表推送消息的内容。

消息分类

目前鹰眼iOS SDK只会推送服务端地理围栏报警和客户端地理围栏报警2种类型的消息，对应的 `type` 属性的值分别为 0x03 和 0x04。

消息内容

对于服务端地理围栏报警和客户端地理围栏报警这2种消息类型对应的消息内容，按照 `BTKPushMessageFenceAlarmContent` 类来解析。

以下代码示例展示了，如何解析消息推送：

```
-(void)onGetPushMessage:(BTKPushMessage *)message {
    NSLog(@"收到推送消息，消息类型: %@", @(message.type));
    if (message.type == 0x03) {
        BTKPushMessageFenceAlarmContent *content = (BTKPushMessageFenceAlarmContent *)message.content;
        if (content.actionType == BTK_FENCE_MONITORED_OBJECT_ACTION_TYPE_ENTER) {
            NSLog(@"被监控对象 %@ 进入 服务端地理围栏 %@ ", content.monitoredObject, content.fenceName);
        } else if (content.actionType == BTK_FENCE_MONITORED_OBJECT_ACTION_TYPE_EXIT) {
            NSLog(@"被监控对象 %@ 离开 服务端地理围栏 %@ ", content.monitoredObject, content.fenceName);
        }
    } else if (message.type == 0x04) {
```

```
        BTKPushMessageFenceAlarmContent *content = (BTKPushMessageFenceAlarmContent
*)message.content;
        if (content.actionType == BTK_FENCE_MONITORED_OBJECT_ACTION_TYPE_ENTER) {
            NSLog(@"被监控对象 %@ 进入 客户端地理围栏 %@ ", content.monitoredObject,
content.fenceName);
        } else if (content.actionType == BTK_FENCE_MONITORED_OBJECT_ACTION_TYPE_EXIT) {
            NSLog(@"被监控对象 %@ 离开 客户端地理围栏 %@ ", content.monitoredObject,
content.fenceName);
        }
    }
}
```

偏好设置

鹰眼iOS SDK支持开发者对SDK的某些行为偏好进行自定义设置。主要分为定位选项设置、采集周期与上传周期设置、缓存容量设置3个方面。

定位选项设置

鹰眼iOS SDK 使用iOS系统定位服务（CoreLocation），开发者可以通过 `BTKAction` 类中

```
-(void)setLocationAttributeWithActivityType:(CLActivityType)activityType
                                    desiredAccuracy:(CLLocationAccuracy)desiredAccuracy
                                    distanceFilter:(CLLocationDistance)distanceFilter;
```

方法设置CoreLocation中定位的活动类型、定位精度以及触发定位的距离阈值。该方法中的3个参数，将会透传给CoreLocation的LocationManager，3个参数的值请参考苹果官方文档中的用法，根据开发者自己的使用场景进行设置。

采集周期与上传周期设置

鹰眼iOS SDK默认的采集周期为5秒，上传周期为30秒，即SDK会每隔5秒采集一次当前设备的位置信息，每隔30秒将该周期内的若干个位置信息打包、压缩、加密后传输至鹰眼服务端。

1. SDK支持开发者通过 `BTKAction` 类中的 `-(void)changeGatherAndPackIntervals:(NSUInteger)gatherInterval packInterval:(NSUInteger)packInterval delegate:(id <BTKTraceDelegate>)delegate;` 方法自定义采集周期和上传周期。需要注意的是，该值一经设置，永久有效。即使是重新开启轨迹服务，也会按照上一次设置过的采集周期和上传周期进行采集和上传，SDK默认的采集周期和上传周期不再有效。如果想更改周期，再次调用此方法即可。该方法在开启轨迹服务之前和之后都可以调用。
2. **采集周期**一定是准确的，而**上传周期**只在网络畅通的时候才有意义。网络畅通的情况下，按照5秒采集30秒上传的默认值，查询历史轨迹时会发现，各轨迹点的loctime间隔为采集周期，而每6个连续的轨迹点的create_time是相同的，代表这6个点是同时上传至服务端的。而当断网时，SDK会自动缓存轨迹，等网络恢复后再批量上传至服务端。这种情况下，当查询历史轨迹时会发现，各轨迹点的loctime间隔仍然为采集周期，但create_time则没有什么规律了。但SDK会保证同一个entity终端的轨迹点，loctime较晚的点不会在loctime较早的点之前上传，以确保正确触发服务端地理围栏报警。

以下代码片段表示，将SDK默认的5秒采集一次位置信息、30秒上传一次位置信息，改为自定义的2秒采集一次位置信息、10秒上传一次位置信息。直到下次调用此方法，重新自定义采集周期和上传周期，SDK将一直使用本次自定义的值，即使停止轨迹服务后，重新开启轨迹服务也不受影响。

```
[[BTKAction sharedInstance] changeGatherAndPackIntervals:2 packInterval:10
delegate:self];
```

缓存容量设置

鹰眼iOS SDK支持开发者自定义SDK缓存所占磁盘空间的最大值。开发者可以通过 `BTKAction` 类的 `-(void)setCacheMaxSize:(NSUInteger)size delegate:(id <BTKTraceDelegate>)delegate;` 方法进行设置。关于此方法的行为有以下几点说明：

1. 如果设置了该阈值，当SDK缓存的数据占用磁盘超过该阈值时，将删除最早的缓存轨迹，直到满足该条件。
2. SDK默认情况下缓存占用空间没有限制。如果对于缓存占用空间没有非常强烈的要求，建议不要调用此方法。否则将会导致缓存轨迹数据被丢弃等情况，且数据无法找回。
3. 阈值一经设置，永久有效。如果想取消限制，只有再次设置一个非常大的值才行。

以下代码片段表示，设置客户端缓存占用磁盘空间的最大值为60MB，缓存轨迹数据占用超过60MB时将清空最早的缓存轨迹，直到占用空间满足要求。

```
[[BTKAction sharedInstance] setCacheMaxSize:60 delegate:self];
```

终端实体（Entity）

鹰眼iOS SDK提供了对终端实体的管理和检索功能。通过调用 `BTKEntityAction` 类中相应的接口发起请求，响应通过 `BTKEntityDelegate` 协议中对应的方法回调给开发者。

终端管理

类 `BTKEntityAction` 中提供了对终端实体的增、删、改、查，4种操作，满足开发者对终端实体的管理需求。

新建终端实体

通过 `-(void)addEntityWith:(BTKAddEntityRequest *)request delegate:(id<BTKEntityDelegate>)delegate;` 方法新建一个终端实体。

以下代码片段表示，在已经通过轨迹管理台为100000这个service创建过一个名为“city”的entity属性字段的情况下，为这个service新建一个名为 “entityA”的终端实体，它的描述信息为 “实体A”，其“city”属性的值为"bj"。

```
// 设置新建的终端实体的entity属性字段的值
NSDictionary *columnKey = @{@"city":@"bj"};

// 构造请求对象
BTKAddEntityRequest *request = [[BTKAddEntityRequest alloc]
initWithEntityName:@"entityA" entityDesc:@"实体A" columnKey:columnKey serviceID:100000
tag:31];

// 发起创建请求
[[BTKEntityAction sharedInstance] addEntityWith:request delegate:self];
```

删除终端实体

通过 `-(void)deleteEntityWith:(BTKDeleteEntityRequest *)request delegate:(id<BTKEntityDelegate>)delegate;` 方法删除一个已有的终端实体。

以下代码片段表示，删除100000这个service下名为 “entityA”的终端实体。

```
// 构造请求对象
BTKDeleteEntityRequest *request = [[BTKDeleteEntityRequest alloc]
initWithEntityName:@"entityA" serviceID:100000 tag:32];

// 发起删除请求
```

```objc
[[BTKEntityAction sharedInstance] deleteEntityWith:request delegate:self];
```

更新终端实体

通过 `-(void)updateEntityWith:(BTKUpdateEntityRequest *)request delegate:(id<BTKEntityDelegate>)delegate;` 方法更新一个终端实体的描述字段或属性信息。

以下代码片段表示，在100000这个service下，名称为“entityA”的终端实体，将描述字段的值改为“实体AAA”，将“city”这个entity属性字段的值改为“北京”。

```objc
// 设置需要更新的entity属性字段的值
NSDictionary *columnKey = @{@"city":@"北京"};

// 构造请求对象
BTKUpdateEntityRequest *request = [[BTKUpdateEntityRequest alloc] initWithEntityName:@"entityA" entityDesc:@"实体AAA" columnKey:columnKey serviceID:100000 tag:33];

// 发起更新请求
[[BTKEntityAction sharedInstance] updateEntityWith:request delegate:self];
```

查询终端实体

通过 `-(void)queryEntityWith:(BTKQueryEntityRequest *)request delegate:(id<BTKEntityDelegate>)delegate;` 方法，检索符合指定过滤条件的终端实体。

以下代码片段表示，列出100000这个service下，名称为“entityA”或“entityB”的，且在24小时之内有轨迹点上传的，活跃终端实体的最新位置。

```objc
// 设置过滤条件
BTKQueryEntityFilterOption *filter = [[BTKQueryEntityFilterOption alloc] init];
// 设置过滤条件中的entityName，若不设置则检索该entity下全部的终端实体
filter.entityNames = @[@"entityA", @"entityB"];
// 设置过滤条件中的活跃时间
filter.activeTime = [[NSDate date] timeIntervalSince1970] - 24 * 60 * 60;

// 构造请求对象
BTKQueryEntityRequest *request = [[BTKQueryEntityRequest alloc] initWithFilter:filter outputCoordType:BTK_COORDTYPE_BD09LL pageIndex:1 pageSize:100 serviceID:100000 tag:34];

// 发起查询请求
[[BTKEntityAction sharedInstance] queryEntityWith:request delegate:self];
```

`BTKEntityAction` 类中的 `queryEntityWith:` 方法、 `searchEntityWith` 方法，以及 `BTKTrackAction` 中的 `queryTrackLatestPointWith`方法，都可以查询终端实体的实时位置，它们的区别可以用下面的表格来表示：

值班人员	多个entity	模糊检索	轨迹纠偏	排序条件
queryEntityWith	支持	不支持	不支持	不支持
searchEntityWith	支持	支持	不支持	支持
queryTrackLatestPointWith	不支持	不支持	支持	不支持

接口选择建议：

- 如果需要模糊检索，只能选择 `searchEntityWith:` 接口。
- 如果需要返回纠偏后的实时位置，只能选择 `queryTrackLatestPointWith` 接口。
- 如果需要返回多个entity甚至所有entity的实时位置，且不需要模糊检索，使用 `queryEntityWith:` 接口即可。
- 如果需要返回多个entity甚至所有entity的实时位置，且需要模糊检索，或者需要指定返回的多个entity的排序规则，只能选择 `searchEntityWith:` 接口。

实时位置检索

类 `BTKEntityAction` 中支持检索终端实体的实时位置，包括3种类型的检索方式：

关键字检索

通过 `-(void)searchEntityWith:(BTKSearchEntityRequest *)request delegate:(id<BTKEntityDelegate>)delegate;` 方法，根据关键字检索终端实体，并返回其实时位置。与 `-(void)queryEntityWith:(BTKQueryEntityRequest *)request delegate:(id<BTKEntityDelegate>)delegate;` 方法相比，该方法增加了关键字检索的功能，支持对终端实体的名称和描述的联合模糊检索。

以下代码片段表示，查找100000这个service下，名称或描述字段含有“entity”的，在过去7天有轨迹上传的，终端实体的实时位置。若有多个entity满足条件，则按照它们最后定位时间 “loc_time” 字段的倒序返回，即定位时间离现在越近越靠前。

```
// 设置过滤条件
BTKQueryEntityFilterOption *filterOption = [[BTKQueryEntityFilterOption alloc] init];
filterOption.activeTime = [[NSDate date] timeIntervalSince1970] - 7 * 24 * 3600;

// 设置排序条件，返回的多个entity按照，定位时间'loc_time'的倒序排列
BTKSearchEntitySortByOption * sortbyOption = [[BTKSearchEntitySortByOption alloc] init];
sortbyOption.fieldName = @"loc_time";
sortbyOption.sortType = BTK_ENTITY_SORT_TYPE_DESC;

// 构造请求对象
BTKSearchEntityRequest *request = [[BTKSearchEntityRequest alloc]
initWithQueryKeyword:@"entity" filter:filterOption sortby:sortbyOption
outputCoordType:BTK_COORDTYPE_BD09LL pageIndex:1 pageSize:10 ServiceID:100000 tag:34];

// 发起检索请求
[[BTKEntityAction sharedInstance] searchEntityWith:request delegate:self];
```

矩形检索

通过 `-(void)boundSearchEntityWith:(BTKBoundSearchEntityRequest *)request delegate:(id<BTKEntityDelegate>)delegate;` 方法，在指定的矩形地理范围内，检索符合条件的终端实体，并返回其实时位置。

以下代码片段表示，在西南角为东经116.311046度、北纬40.046667度，东北角为东经116.315264度、北纬40.048973度，所构成的矩形区域内，检索名称为 “entityA” 、 “entityB”、“entityC” 这3个终端实体，在过去的7天内是否有轨迹点上传。如果有，则返回每个终端实体最后一次上传的轨迹点，如果返回结果包含不止一个轨迹点，则按照它们最后一次上传的时间戳 “loc_time” 字段的倒序排列，即离当前时间越近的轨迹点越靠前。

```
// 设置矩形的区域
NSMutableArray *bounds = [NSMutableArray arrayWithCapacity:2];
// 矩形左下角的顶点坐标
CLLocationCoordinate2D point1 = CLLocationCoordinate2DMake(40.046667, 116.311046);
```

```objc
[bounds addObject:[NSValue valueWithBytes:&point1
objCType:@encode(CLLocationCoordinate2D)]];
// 矩形右上角的顶点坐标
CLLocationCoordinate2D point2 = CLLocationCoordinate2DMake(40.048973, 116.315264);
[bounds addObject:[NSValue valueWithBytes:&point2
objCType:@encode(CLLocationCoordinate2D)]];

// 设置检索的过滤选项
BTKQueryEntityFilterOption *filterOption = [[BTKQueryEntityFilterOption alloc] init];
filterOption.entityNames = @[@"entityA", @"entityB", @"entityC"];
filterOption.activeTime = [[NSDate date] timeIntervalSince1970] - 7 * 24 * 3600;

// 设置检索结果的排序选项
BTKSearchEntitySortByOption * sortbyOption = [[BTKSearchEntitySortByOption alloc] init];
sortbyOption.fieldName = @"loc_time";
sortbyOption.sortType = BTK_ENTITY_SORT_TYPE_DESC;

// 构造检索请求
BTKBoundSearchEntityRequest *request = [[BTKBoundSearchEntityRequest alloc]
initWithBounds:bounds inputCoordType:BTK_COORDTYPE_BD09LL filter:filterOption
sortby:sortbyOption outputCoordType:BTK_COORDTYPE_BD09LL pageIndex:1 pageSize:10
ServiceID:100000 tag:44];

// 发起检索请求
[[BTKEntityAction sharedInstance] boundSearchEntityWith:request delegate:self];
```

圆形检索

通过 `-(void)aroundSearchEntityWith:(BTKAroundSearchEntityRequest *)request delegate:(id<BTKEntityDelegate>)delegate;` 方法，在指定的圆形地理范围内，检索符合条件的终端实体，并返回其实时位置。

以下代码片段表示，在以东经116.312964度、北纬40.047772度为圆心，半径100米的范围内，检索100000这个service下所有在过去7天有轨迹点上传的活跃终端，如果有满足条件的终端实体，则返回其最后一次上传的轨迹点。若有多个终端实体符合以上检索条件，则返回结果按照它们的定位时间戳"loc_time" 字段倒序排列，即离当前时间越近的轨迹点越靠前。

```objc
// 设置圆形的圆心
CLLocationCoordinate2D center = CLLocationCoordinate2DMake(40.047772, 116.312964);

// 设置检索的过滤条件
BTKQueryEntityFilterOption *filterOption = [[BTKQueryEntityFilterOption alloc] init];
filterOption.activeTime = [[NSDate date] timeIntervalSince1970] - 7 * 24 * 3600;

// 设置检索结果的排序方式
BTKSearchEntitySortByOption * sortbyOption = [[BTKSearchEntitySortByOption alloc] init];
sortbyOption.fieldName = @"loc_time";
sortbyOption.sortType = BTK_ENTITY_SORT_TYPE_DESC;

// 构造检索请求对象
BTKAroundSearchEntityRequest *request = [[BTKAroundSearchEntityRequest alloc]
initWithCenter:center inputCoordType:BTK_COORDTYPE_BD09LL radius:100 filter:filterOption
sortby:sortbyOption outputCoordType:BTK_COORDTYPE_BD09LL pageIndex:1 pageSize:10
ServiceID:100000 tag:55];

// 发起检索请求
[[BTKEntityAction sharedInstance] aroundSearchEntityWith:request delegate:self];
```

轨迹（Track）

功能总览

鹰眼iOS SDK提供了一系列与轨迹相关的功能。通过调用 `BTKTrackAction` 类中相应的接口发起请求，响应通过 `BTKTrackDelegate` 协议中对应的方法回调给开发者。轨迹相关的功能主要包括以下几个方面：

- 轨迹查询
  - 查询某终端实体的实时位置，支持轨迹纠偏
  - 查询某终端实体在一段时间内的轨迹，支持轨迹纠偏，支持里程补偿
  - 查询某终端实体在一段时间内的里程，支持轨迹纠偏，支持里程补偿
- 自定义轨迹点上传
  - 上传单个entity的单个自定义轨迹点
  - 批量上传多个entity的多个自定义轨迹点
- 缓存轨迹的管理
  - 查询客户端缓存的轨迹信息
  - 清空客户端缓存的轨迹信息

概念解释

在功能总览中，出现了轨迹纠偏、里程补偿、自定义轨迹点、缓存轨迹等概念，本节对这几个概念进行解释。

轨迹纠偏

各种定位方式或多或少都存在着一定的误差。如果是在室外，如果GPS信号比较好，定位结果会比较准确。当GPS信号不好的时候（例如高架桥下、隧道、高层建筑遮挡等），可能就会使用WIFI或基站定位，特别是当周边WIFI热点比较少的时候会使用基站定位，定位误差会有所加大，产生轨迹漂移的现象。

为了更好地帮助开发者管理轨迹和展现轨迹，鹰眼提供了轨迹纠偏功能，达到优化轨迹、校正里程等效果。（注：纠偏轨迹与原始轨迹数据相互独立，原始轨迹数据仍被保留并可查询。）

鹰眼轨迹纠偏包括以下步骤：

1. 去噪：对于明显的噪点进行识别并去除。
2. 抽稀：对于冗余的数据点进行去除，如一条直线上的多个轨迹点，减少数据量，提升展示效率。
3. 绑路：将轨迹点绑定至道路，达到纠正偏移轨迹、补充中断轨迹点（如：轨迹不连续、进入隧道导致的丢点）、补充道路拐点等效果。

鹰眼iOS SDK中使用 `BTKQueryTrackProcessOption` 类来设置轨迹纠偏选项：

- `denoise` 设置纠偏时是否需要去噪。
- `vacuate` 设置纠偏时是否需要抽稀。
- `mapMatch` 设置纠偏时是否需要绑路。
- `radiusThreshold` 设置定位精度过滤阈值，用于过滤掉定位精度较差的轨迹点。
- `transportMode` 设置轨迹对应的交通方式，鹰眼纠偏模块将根据不同的交通方式采用不同的轨迹纠偏处理。

纠偏选项的默认值为去噪、不绑路、不过滤噪点、交通方式为驾车。

里程补偿

在查询某时间段内的轨迹或里程时，除了指定纠偏选项外，还支持里程补偿。两个轨迹点定位时间间隔5分钟以上，被认为是中断，鹰眼支持对中断5分钟以上的轨迹区间进行里程补偿。里程补偿只是对里程数进行补偿纠正，并不会补充具体的轨迹点。

鹰眼iOS SDK中使用 `BTKTrackProcessOptionSupplementMode` 枚举类型表示里程补偿的方式：

- `BTK_TRACK_PROCESS_OPTION_NO_SUPPLEMENT` 代表不进行补充
- `BTK_TRACK_PROCESS_OPTION_SUPPLEMENT_MODE_STRAIGHT` 代表使用直线距离补充
- `BTK_TRACK_PROCESS_OPTION_SUPPLEMENT_MODE_DRIVING` 代表使用最短驾车路线距离补充
- `BTK_TRACK_PROCESS_OPTION_SUPPLEMENT_MODE_RIDING` 代表使用最短骑行路线距离补充
- `BTK_TRACK_PROCESS_OPTION_SUPPLEMENT_MODE_WALKING` 代表使用最短步行路线距离补充

里程补偿的默认值为不补充，中断两点间距离不记入里程。

这里需要特别说明的是：纠偏选项中的交通方式和里程补偿中的交通方式并无关系，纠偏选项中交通方式只会影响纠偏策略；而里程补偿中的交通方式只有当轨迹点定位间隔超过5分钟时，才起作用。

自定义轨迹点

鹰眼iOS SDK除了可以自动采集并上传轨迹信息外，还支持开发者上传自定义的轨迹点。比如在两个采集周期之间，上传某个轨迹点作为补充；或者上传非当前登陆的entity的其他终端的轨迹点等使用场景。自定义轨迹点有如下几点说明：

- 自定义轨迹点除了可以上传位置坐标以及速度、方向、海拔高度、定位精度等系统预定义的track属性字段外，同样支持开发者自定义的track属性字段的上传，类似通过 `BTKTraceDelegate` 协议中的 `-(NSDictionary *)onGetCustomData;` 方法设置每个轨迹点的附加数据。
- 自定义轨迹点上传至服务端之后，和SDK自动采集上传的点没有区别。都会触发服务端地理围栏的计算、查询实时位置和轨迹时，也会返回。
- 自定义轨迹点的上传依赖网络。
- 自定义轨迹点的上传不会参与客户端地理围栏的计算，也就不会触发客户端地理围栏报警。

轨迹缓存

鹰眼iOS SDK在网络不畅时，将采集到的轨迹持久化在客户端本地，称为轨迹缓存。在网络畅通的情况下，SDK基本不会存在轨迹数据的缓存。开发者可以查询SDK缓存的轨迹数据所属的终端实体名称、时间段、数量等信息，但无法获取具体的位置信息。也可以清空指定entity在指定时间段内的轨迹缓存。轨迹缓存清除之后，将不再上传至鹰眼服务端，且无法还原，清空之前，请谨慎确认对应的终端实体名称和时间段无误。

- 通过 `BTKTrackAction` 类中的 `-(void)queryTrackCacheInfoWith:(BTKQueryTrackCacheInfoRequest *)request delegate:(id<BTKTrackDelegate>)delegate;` 方法，查询客户端缓存的轨迹数据所属的终端实体名称、时间段、数量等信息。
- 通过 `BTKTrackAction` 类中的 `-(void)clearTrackCacheWith:(BTKClearTrackCacheRequest *)request delegate:(id<BTKTrackDelegate>)delegate;` 方法，清空满足指定条件的轨迹缓存数据。

具体用法

查询实时位置

通过 `-(void)queryTrackLatestPointWith:(BTKQueryTrackLatestPointRequest *)request delegate:(id<BTKTrackDelegate>)delegate;` 方法，查询某终端实体的经过轨迹纠偏后的实时位置。

以下代码片段表示查询名称为 “entityA” 的终端，经过纠偏之后的实时位置，去噪、绑路、定位精度大于10米的点将被认为是噪点。

```
// 设置纠偏选项
BTKQueryTrackProcessOption *option = [[BTKQueryTrackProcessOption alloc] init];
option.denoise = TRUE;
```

```
option.mapMatch = TRUE;
option.radiusThreshold = 10;

// 构造请求对象
BTKQueryTrackLatestPointRequest *request = [[BTKQueryTrackLatestPointRequest alloc]
initWithEntityName:@"entityA" processOption: option outputCootdType:BTK_COORDTYPE_BD09LL
serviceID:100000 tag:11];

// 发起查询请求
[[BTKTrackAction sharedInstance] queryTrackLatestPointWith:request delegate:self];
```

查询轨迹

通过 `-(void)queryHistoryTrackWith:(BTKQueryHistoryTrackRequest *)request delegate:(id<BTKTrackDelegate>)delegate;` 方法，查询某终端实体在某段时间内的轨迹。

以下代码片段表示查询名称为 “entityA” 的终端，在过去24小时内经过纠偏后的轨迹，纠偏选项采用默认值（去噪、不绑路、不过滤噪点、交通方式为驾车）；选择最短步行路线距离进行里程补偿。如前文中的解释，纠偏选项中的驾车交通方式代表按照驾车的行驶行为进行纠偏，而对定位时间间隔大于5分钟的轨迹点间，采用最短步行路线距离进行里程补偿，正常的轨迹点不受里程补偿方案的影响。

```
// 构造请求对象
NSUInteger endTime = [[NSDate date] timeIntervalSince1970];
BTKQueryHistoryTrackRequest *request = [[BTKQueryHistoryTrackRequest alloc]
initWithEntityName:@"entityA" startTime:endTime - 84400 endTime:endTime isProcessed:TRUE
processOption:nil supplementMode:BTK_TRACK_PROCESS_OPTION_SUPPLEMENT_MODE_WALKING
outputCoordType:BTK_COORDTYPE_BD09LL sortType:BTK_TRACK_SORT_TYPE_DESC pageIndex:1
pageSize:10 serviceID:103044 tag:13];

// 发起查询请求
[[BTKTrackAction sharedInstance] queryHistoryTrackWith:request delegate:self];
```

查询里程

通过 `-(void)queryTrackDistanceWith:(BTKQueryTrackDistanceRequest *)request delegate:(id<BTKTrackDelegate>)delegate;` 方法，查询某终端实体在某段时间内的里程。

以下代码片段表示查询名称为 “entityA” 的终端，在过去24小时内经过轨迹纠偏后的里程，纠偏时去噪、绑路，定位精度超过15米的轨迹点被认为是噪点、使用驾车的行驶行为进行纠偏；定位时间间隔超过5分钟的轨迹点之间，采用最短步行距离进行里程补偿。

```
// 设置纠偏选项
BTKQueryTrackProcessOption *option = [[BTKQueryTrackProcessOption alloc] init];
option.denoise = TRUE;
option.mapMatch = TRUE;
option.radiusThreshold = 15;
option.transportMode = BTK_TRACK_PROCESS_OPTION_TRANSPORT_MODE_DRIVING;
NSUInteger endTime = [[NSDate date] timeIntervalSince1970];

// 构造请求对象
BTKQueryTrackDistanceRequest *request = [[BTKQueryTrackDistanceRequest alloc]
initWithEntityName:@"entityA" startTime:endTime - 84400 endTime:endTime isProcessed:TRUE
processOption:nil supplementMode:BTK_TRACK_PROCESS_OPTION_SUPPLEMENT_MODE_WALkING
serviceID:100000 tag:12];

// 发起查询请求
[[BTKTrackAction sharedInstance] queryTrackDistanceWith:request delegate:self];
```

缓存管理

鹰眼iOS SDK提供了对缓存轨迹信息的查询和删除操作。

查询缓存信息

通过 `-(void)queryTrackCacheInfoWith:(BTKQueryTrackCacheInfoRequest *)request delegate:(id<BTKTrackDelegate>)delegate;` 方法，查询客户端缓存轨迹数据所属的终端实体名称、时间段、数量等信息；

以下代码片段表示，查询缓存在本机上的所有entity的时间段和轨迹点数量信息。

```
// 构造请求对象
BTKQueryTrackCacheInfoRequest *request = [[BTKQueryTrackCacheInfoRequest alloc] initWithEntityNames:nil serviceID:100000 tag:333];

// 发起请求
[[BTKTrackAction sharedInstance] queryTrackCacheInfoWith:request delegate:self];
```

清空缓存信息

通过 `-(void)clearTrackCacheWith:(BTKClearTrackCacheRequest *)request delegate:(id<BTKTrackDelegate>)delegate;` 方法，清空客户端缓存的轨迹信息。

以下代码片段表示，清空缓存在本机上，属于 “entityA” 的前天的轨迹，以及属于 “entityB” 的昨天的轨迹。

```
NSUInteger now = [[NSDate date] timeIntervalSince1970];

// 设置entityA名下，要清空的轨迹缓存的起止时间
BTKClearTrackCacheOption *op1 = [[BTKClearTrackCacheOption alloc] initWithEntityName:@"entityA" startTime:(now - 84400 * 3) endTime:(now - 84400 * 2)];

// 设置entityA名下，要清空的轨迹缓存的起止时间
BTKClearTrackCacheOption *op2 = [[BTKClearTrackCacheOption alloc] initWithEntityName:@"entityB" startTime:(now - 84400 * 2) endTime:(now - 84400)];

// 设置清空的条件
NSMutableArray *options = [NSMutableArray arrayWithCapacity:2];
[options addObject:op1];
[options addObject:op2];

// 构造请求对象
BTKClearTrackCacheRequest *request = [[BTKClearTrackCacheRequest alloc] initWithOptions:options serviceID:100000 tag:33];

// 发起请求
[[BTKTrackAction sharedInstance] clearTrackCacheWith:request delegate:self];
```

自定义轨迹点上传

上传某个entity的单个轨迹点

通过 `-(void)addCustomPointWith:(BTKAddCustomTrackPointRequest *)request delegate:(id<BTKTrackDelegate>)delegate;` 方法，上传单个的自定义轨迹点；

以下代码片段表示，100000这个service下，已经通过轨迹管理台创建了3个叫做"someIntColumn"、"someDoubleColumn"、 "someStringColumn" 的自定义track属性。名称为 “entityA” 的终端，在10秒钟之前，出现在了东经111.123456度，北纬44.654321度的位置，当时它的 “someIntColumn”属性的值为50，someDoubleColumn属性的值为44.55，someStringColumn属性的值为“aaa”，且方向为北偏东11度，海拔高

度58米，定位精度为3米，速度为68米/秒。

```objc
// 设置轨迹点的坐标
CLLocationCoordinate2D coordinate = CLLocationCoordinate2DMake(44.654321, 111.123456);

// 设置轨迹点的定位时间戳
NSUInteger timestamp = [[NSDate date] timeIntervalSince1970] - 10;

// 设置轨迹点的自定义track属性字段的值
NSMutableDictionary *customData = [NSMutableDictionary dictionaryWithCapacity:3];
customData[@"someIntColumn"] = @50;
customData[@"someDoubleColumn"] = @44.55;
customData[@"someStringColumn"] = @"aaa";

// 构造自定义轨迹点
BTKCustomTrackPoint *point = [[BTKCustomTrackPoint alloc] initWithCoordinate:coordinate coordType:BTK_COORDTYPE_BD09LL loctime:timestamp direction:11 height:58 radius:3 speed:68 customData:customData entityName:@"entityA"];

// 使用自定义轨迹点构造请求对象
BTKAddCustomTrackPointRequest *request = [[BTKAddCustomTrackPointRequest alloc] initWithCustomTrackPoint:point serviceID:100000 tag:444];

// 发起上传请求
[[BTKTrackAction sharedInstance] addCustomPointWith:request delegate:self];
```

批量上传多个entity的多个轨迹点

通过 `-(void)batchAddCustomPointWith:(BTKBatchAddCustomTrackPointRequest *)request delegate:(id<BTKTrackDelegate>)delegate;` 方法，批量上传若干个开发者自定义轨迹点，这些轨迹点可以属于不同的终端实体。

以下代码片段表示，一次性上传2个自定义轨迹点，分别属于entityA和entityB，每个轨迹点的设置与上面上传单个轨迹点时相同。

```objc
// 第一个轨迹点point1
CLLocationCoordinate2D coordinate1 = CLLocationCoordinate2DMake(44.44, 111.11);
NSUInteger timestamp1 = [[NSDate date] timeIntervalSince1970] - 10;
NSMutableDictionary *customData1 = [NSMutableDictionary dictionaryWithCapacity:3];
customData1[@"someIntColumn"] = @50;
customData1[@"someDoubleColumn"] = @44.55;
customData1[@"someStringColumn"] = @"aaa";
BTKCustomTrackPoint *point1 = [[BTKCustomTrackPoint alloc] initWithCoordinate:coordinate1 coordType:BTK_COORDTYPE_BD09LL loctime:timestamp1 direction:11 height:11 radius:11 speed:11 customData:customData1 entityName:@"entityA"];

// 第二个轨迹点point2
CLLocationCoordinate2D coordinate2 = CLLocationCoordinate2DMake(55.55, 122.22);
NSUInteger timestamp2 = [[NSDate date] timeIntervalSince1970] - 20;
NSMutableDictionary *customData2 = [NSMutableDictionary dictionaryWithCapacity:3];
customData2[@"someIntColumn"] = @50;
customData2[@"someDoubleColumn"] = @44.55;
customData2[@"someStringColumn"] = @"aaa";
BTKCustomTrackPoint *point2 = [[BTKCustomTrackPoint alloc] initWithCoordinate:coordinate2 coordType:BTK_COORDTYPE_BD09LL loctime:timestamp2 direction:22 height:22 radius:22 speed:22 customData:customData2 entityName:@"entityB"];

// 构造需要上传的多个轨迹点
NSMutableArray *points = [NSMutableArray arrayWithCapacity:2];
```

```
[points addObject:point1];
[points addObject:point2];

// 构造请求对象
BTKBatchAddCustomTrackPointRequest *request = [[BTKBatchAddCustomTrackPointRequest alloc] initWithCustomTrackPoints:points serviceID:100000 tag:444];

// 发起请求
[[BTKTrackAction sharedInstance] batchAddCustomPointWith:request delegate:self];
```

分析（Analysis）

鹰眼iOS SDK提供了轨迹分析相关的功能。通过 `BTKAnalysisAction` 类中相应的接口发起请求，响应通过 `BTKTrackDelegate` 协议中对应的方法回调给开发者。包括停留点分析、驾驶行为分析功能。

停留点分析

通过 `-(void)analyzeStayPointWith:(BTKStayPointAnalysisRequest *)request delegate:(id<BTKAnalysisDelegate>)delegate;` 方法，查询指定终端实体在指定时间段内的停留点。

以下代码片段表示，使用默认的纠偏规则（不绑路、交通工具为驾车）情况下，如果认为在某个半径为20米的圆形区域内停留了超过100秒为一次**停留**的话，查询名称为 “entityA” 的终端实体，在过去12小时内，所有的停留点。

```
NSUInteger endTime = [[NSDate date] timeIntervalSince1970];

// 构造请求对象
BTKStayPointAnalysisRequest *request = [[BTKStayPointAnalysisRequest alloc] initWithEntityName:@"entityA" startTime:endTime - 12 * 60 * 60 endTime:endTime stayTime:100 stayRadius:20 processOption:nil outputCoordType:BTK_COORDTYPE_BD09LL serviceID:100000 tag:222];

// 发起请求
[[BTKAnalysisAction sharedInstance] analyzeStayPointWith:request delegate:self];
```

驾驶行为分析

通过 `-(void)analyzeDrivingBehaviourWith:(BTKDrivingBehaviourAnalysisRequest *)request delegate:(id<BTKAnalysisDelegate>)delegate;` 方法，查询指定终端实体在指定时间段内的驾驶行为。驾驶行为包括：起终点信息、里程、耗时、平均速度、最高速度等总体信息，以及超速、急加速、急刹车、急转弯等异常信息。

以下代码片段表示，使用默认的纠偏规则（不绑路、交通工具为驾车）情况下，如果认为 50km/h 为超速的话，查询名称为 “entityA” 的终端实体，在过去12小时内的驾驶行为。

该方法返回的驾驶行为包括起终点信息、里程、耗时、平均速度、最高速度等总体信息；以及超速、急加速、急刹车、急转弯等异常信息。返回各字段的具体含义请参考Web API文档中 drivingbehavior接口 对应的介绍。

```
NSUInteger endTime = [[NSDate date] timeIntervalSince1970];

// 构造请求对象
BTKDrivingBehaviourAnalysisRequest *request = [[BTKDrivingBehaviourAnalysisRequest alloc] initWithEntityName:@"zhubei" startTime:endTime - 12 * 60 * 60 endTime:endTime speedingThreshold:50 processOption:nil outputCoordType:BTK_COORDTYPE_BD09LL serviceID:103044 tag:223];

```

```
// 发起请求
[[BTKAnalysisAction sharedInstance] analyzeDrivingBehaviourWith:request delegate:self];
```

地理围栏（Fence）

鹰眼iOS SDK提供了一系列与地理围栏相关的功能。通过 `BTKFenceAction` 类中相应的接口发起请求，响应通过 `BTKFenceDelegate` 协议中对应的方法回调给开发者。

功能简介

鹰眼iOS SDK中的地理围栏分为客户端地理围栏和服务端地理围栏，二者都支持创建、删除、修改、查询操作。使用鹰眼iOS SDK，还可以查询指定终端实体与指定地理围栏的位置关系，查询指定终端实体的历史报警信息。当某终端实体的轨迹触发了地理围栏之后，报警信息会通过 `BTKTraceDelegate` 的 `-(void)onGetPushMessage:(BTKPushMessage *)message;` 方法回调给开发者。

使用场景

鹰眼服务中，地理围栏主要有以下3种使用场景

1. 物流时效监控：司机APP集成鹰眼SDK持续追踪轨迹。车辆管理人员通过服务端为每个司机创建出发地和目的地站点围栏，一旦监控对象进出围栏，鹰眼支持向鹰眼SDK 和 开发者服务端 推送报警信息。
2. 路线偏离报警：鹰眼v3.0新增路线围栏，可设定车辆行驶路线和偏离距离，一旦车辆偏离超过设定的距离，鹰眼围栏将推送报警。
3. 用车行业运营区域监控：利用鹰眼多边形地理围栏和行政区围栏功能，设置运营区域围栏，一旦判断车辆进/出运营区触发报警。

概念说明

围栏的种类

鹰眼iOS SDK提供了两个端的地理围栏：客户端地理围栏和服务端地理围栏。

- **客户端地理围栏**的管理、计算、报警触发均在客户端的鹰眼iOS SDK内部完成，无需联网即可完成地理围栏的运算。
- **服务端地理围栏**的管理、计算、报警触发都在鹰眼服务端完成，依赖于轨迹点及时上传至服务端才能完成围栏的各种操作。因此要想完整地使用服务端地理围栏的功能，使用SDK的设备必须时刻保持联网状态，否则将无法及时触发服务端地理围栏，报警信息也无法及时推送至客户端。与客户端地理围栏相比，服务端地理围栏支持将报警信息推送。

开发者可根据业务场景选择两类围栏中的一类，或两类同时使用，互为补充。

围栏的形状

目前鹰眼iOS SDK的服务端地理围栏支持**圆形**、**多边形**、**线形**、**行政区**4种围栏形状，客户端地理围栏支持圆形围栏。四种形状的围栏报警行为如下：

1. 圆形围栏：支持设置圆形围栏，一旦进出圆形范围则推送报警。
2. 多边形围栏：支持设置多边形围栏，一旦进出多边形围栏则推送报警。
3. 线形围栏：支持设置折线围栏，一旦偏离或回到设定路线则推送报警。
4. 行政区围栏：支持通过传入行政区名称，创建以行政区边界为界的围栏。

围栏报警去噪的说明

无论是GPS定位还是网络定位都存在误差（也就是常说的定位漂移问题），噪点会造成围栏误报警。
目前鹰眼围栏进行了去噪处理，同时开放了 `denoiseAccuracy` 去噪精度参数供开发者在创建围栏时设置，围栏运算时，一旦判断轨迹点的定位精度大于此去噪精度，则不参与围栏运算。比如设置 denoiseAccuracy=30，则定位精度大于30米的轨迹点都不会参与围栏计算。
在此提供各定位模式的平均精度供开发者参考：

- 设备在空旷的室外时定位精度均值为几米
- 设备在室内打开Wi-Fi开关的情况下，Wi-Fi定位的精度均值为十几米到几十米
- 设备在室内但关闭Wi-Fi开关的情况下，基站定位的精度均值为几百米到几千米。

坐标系说明

目前中国主要有以下三种坐标系：

- WGS84：为一种大地坐标系，也是目前广泛使用的GPS全球卫星定位系统使用的坐标系。
- GCJ02：是由中国国家测绘局制订的地理信息系统的坐标系统。由WGS84坐标系经加密后的坐标系。
- BD09：为百度坐标系，在GCJ02坐标系基础上再次加密。其中bd09ll表示百度经纬度坐标，bd09mc表示百度墨卡托米制坐标

非中国地区地图，统一使用WGS84坐标。
鹰眼iOS SDK默认的输入输出坐标均为百度经纬度坐标系（bd09ll），同时可通过"coordType"，"coordTypeInput"，"coordTypeOutput"（以各方法参数介绍为准）控制输入输出的坐标类型，鹰眼将自动完成转换。

服务端围栏

使用鹰眼iOS SDK可以创建、删除、更新、查询服务端地理围栏。SDK将轨迹上传至服务端时，服务端会进行相应的计算，若有围栏报警被触发，则通过长连接将报警信息推送至SDK；SDK还可以主动发起请求来查询指定监控对象的状态以及报警历史纪录等。需要注意的是：SDK中使用服务端地理围栏的功能时，必须要求网络畅通。若网络不畅，轨迹信息无法及时上传至服务端，围栏计算和推送均无法正常运行。

创建围栏

通过 `BTKFenceAction` 类中的 `-(void)createServerFenceWith:(BTKCreateServerFenceRequest *)request delegate:(id<BTKFenceDelegate>)delegate;` 方法创建服务端地理围栏。该方法中的请求对象为 `BTKCreateServerFenceRequest` 类型，该类提供了4个构造方法，分别用于创建不同形状的地理围栏，其主要区别在于fence参数的类型：

- `BTKServerCircleFence` 代表服务端**圆形**地理围栏；
- `BTKServerPolygonFence` 代表服务端**多边形**地理围栏；
- `BTKServerPolylineFence` 代表服务端**线形**地理围栏；
- `BTKServerDistrictFence` 代表服务端**行政区域**地理围栏；

下面的代码片段表示，创建一个名称为“server_circle_fence” 的服务端圆形地理围栏，圆心坐标为东经116.3134度、北纬40.0478度，围栏半径为50米。它的监控对象为“entityA”，且当entityA这个终端实体的定位精度大于50米的轨迹点不参与此围栏的计算。

```
// 圆心
CLLocationCoordinate2D center = CLLocationCoordinate2DMake(40.0478, 116.3134);

// 构造将要创建的新的围栏对象
BTKServerCircleFence *fence = [[BTKServerCircleFence alloc] initWithCenter:center radius:50 coordType:BTK_COORDTYPE_BD09LL denoiseAccuracy:50
```

```
fenceName:@"server_circle_fence" monitoredObject:@"entityA"];

// 构造请求对象
BTKCreateServerFenceRequest *circleRequest = [[BTKCreateServerFenceRequest alloc]
initWithServerCircleFence:fence serviceID:100000 tag:21];

// 发起创建请求
[[BTKFenceAction sharedInstance] createServerFenceWith:circleRequest delegate:self];
```

删除围栏

通过 `BTKFenceAction` 类中的 `-(void)deleteServerFenceWith:(BTKDeleteServerFenceRequest *)request delegate:(id<BTKFenceDelegate>)delegate;` 方法删除服务端地理围栏。

以下代码片段表示，删除监控对象为终端实体 “entityA” 的所有服务端地理围栏。

```
// 构造请求对象
BTKDeleteServerFenceRequest *request = [[BTKDeleteServerFenceRequest alloc]
initWithMonitoredObject:@"entityA" fenceIDs:nil serviceID:100000 tag:22];

// 发起删除请求
[[BTKFenceAction sharedInstance] deleteServerFenceWith:request delegate:self];
```

更新围栏

通过 `BTKFenceAction` 类中的 `-(void)updateServerFenceWith:(BTKUpdateServerFenceRequest *)request delegate:(id<BTKFenceDelegate>)delegate;` 方法修改服务端地理围栏。围栏的形状类型无法更新，比如只能修改某个多边形围栏的顶点，而无法将其直接更新为圆形围栏。若想把某个多边形围栏更新为圆形围栏，应该先删除此多边形围栏，再新建一个圆形围栏。

以下代码片段表示，将fenceID为138的服务端圆形地理围栏的圆心修改为东经116.2134度、北纬40.1478度，半径修改为60米。且这个围栏的名称改为“server_fence_60”，监控对象改为终端实体“entityB”，去噪精度改为60米。

```
// 新的圆心
CLLocationCoordinate2D center = CLLocationCoordinate2DMake(40.1478, 116.2134);

// 新的圆形围栏
BTKServerCircleFence *fence = [[BTKServerCircleFence alloc] initWithCenter:center
radius:60 coordType:BTK_COORDTYPE_BD09LL denoiseAccuracy:60 fenceName:@"server_fence_60"
monitoredObject:@"entityB"];

// 构建请求对象
BTKUpdateServerFenceRequest *request = [[BTKUpdateServerFenceRequest alloc]
initWithServerCircleFence:fence fenceID:138 serviceID:100000 tag:23];

// 发起更新请求
[[BTKFenceAction sharedInstance] updateServerFenceWith:request delegate:self];
```

查询围栏

通过 `BTKFenceAction` 类中的 `-(void)queryServerFenceWith:(BTKQueryServerFenceRequest *)request delegate:(id<BTKFenceDelegate>)delegate;` 方法查询服务端地理围栏。该方法可以根据指定的监控对象或围栏ID，列出满足条件的地理围栏。

以下代码片段表示，查询监控对象为终端实体“entityA” 的所有服务端地理围栏。

```
// 构建请求对象
```

```
BTKQueryServerFenceRequest *request = [[BTKQueryServerFenceRequest alloc] initWithMonitoredObject:@"entityA" fenceIDs:nil outputCoordType:BTK_COORDTYPE_BD09LL serviceID:100000 tag:24];

// 发送查询请求
[[BTKFenceAction sharedInstance] queryServerFenceWith:request delegate:self];
```

实时状态查询

鹰眼iOS SDK支持查询指定监控对象的状态。监控对象即某个终端实体，监控对象的状态是指其相对其上的地理围栏的位置关系：是在圆形或多边形围栏的内部还是外部，是否偏离了线形围栏等。SDK提供了2个方法用于查询监控对象的状态：

使用`-(void)queryServerFenceStatusWith:(BTKQueryServerFenceStatusRequest *)request delegate:(id<BTKFenceDelegate>)delegate;`方法，查询被监控对象的状态时，被监控对象的位置以其上传至服务端的最新轨迹点为准。

以下代码片段表示，查询终端实体“entityA” 和所有监控该终端实体的服务端地理围栏的位置关系。

```
// 构建请求对象
BTKQueryServerFenceStatusRequest *request = [[BTKQueryServerFenceStatusRequest alloc] initWithMonitoredObject:@"entityA" fenceIDs:nil ServiceID:100000 tag:25];

// 发起查询请求
[[BTKFenceAction sharedInstance] queryServerFenceStatusWith:request delegate:self];
```

使用`-(void)queryServerFenceStatusByCustomLocationWith:(BTKQueryServerFenceStatusByCustomLocationRequest *)request delegate:(id<BTKFenceDelegate>)delegate;`方法，可以假设被监控对象处于某自定义的位置坐标时，其和地理围栏的位置关系。

以下代码片段表示，假设“entityA”这个终端实体在东经120.44度，北纬40.11度的话，该终端实体和其上的fenceID为17、23、29的这几个服务端地理围栏的位置关系，只有这几个地理围栏的监控对象是“entityA”这个终端实体时才有意义，如果不知道有哪些地理围栏在监控“entityA”这个终端实体，则fenceIDs属性传入nil即可。

```
// 被监控对象的模拟位置
CLLocationCoordinate2D customLocation = CLLocationCoordinate2DMake(40.11, 120.44);

// 地理围栏ID列表
NSArray *fenceIDs = @[@17, @23, @29];

// 构建请求对象
BTKQueryServerFenceStatusByCustomLocationRequest *request = [[BTKQueryServerFenceStatusByCustomLocationRequest alloc] initWithmonitoredObject:@"entityA" CustomLocation:customLocation coordType:BTK_COORDTYPE_BD09LL fenceIDs:fenceIDs serviceID:100000 tag:28];

// 发起查询请求
[[BTKFenceAction sharedInstance] queryServerFenceStatusByCustomLocationWith:request delegate:self];
```

历史报警查询

鹰眼iOS SDK支持查询指定时间段内，服务端地理围栏的历史报警记录。SDK提供了以下2个方法：

使用`-(void)queryServerFenceHistoryAlarmWith:(BTKQueryServerFenceHistoryAlarmRequest`

`*)request delegate:(id<BTKFenceDelegate>)delegate;` 方法，查询某监控对象的历史报警信息。

以下代码片段表示，查询“entityA”这个终端实体上的所有服务端地理围栏，在过去12小时内的所有报警纪录。

```
NSUInteger endTime = [[NSDate date] timeIntervalSince1970];

// 构建请求对象
BTKQueryServerFenceHistoryAlarmRequest *request =
[[BTKQueryServerFenceHistoryAlarmRequest alloc] initWithMonitoredObject:@"entityA"
fenceIDs:nil startTime:(endTime - 12 * 60 * 60) endTime:endTime
outputCoordType:BTK_COORDTYPE_BD09LL ServiceID:100000 tag:26];

// 发起查询请求
[[BTKFenceAction sharedInstance] queryServerFenceHistoryAlarmWith:request
delegate:self];
```

使用 `-(void)batchQueryServerFenceHistoryAlarmWith:(BTKBatchQueryServerFenceHistoryAlarmRequest *)request delegate:(id<BTKFenceDelegate>)delegate;` 方法，批量查询某个service下，所有的历史报警信息。

以下代码片段表示，查询100000这个service下，所有的终端实体上的所有服务端地理围栏，在过去12小时内的报警历史纪录。

```
NSUInteger endTime = [[NSDate date] timeIntervalSince1970];

// 构建请求对象
BTKBatchQueryServerFenceHistoryAlarmRequest *request =
[[BTKBatchQueryServerFenceHistoryAlarmRequest alloc] initWithStartTime:endTime - 12 * 60
* 60 endTime:endTime outputCoordType:BTK_COORDTYPE_BD09LL pageIndex:1 pageSize:10
ServiceID:100000 tag:27];

// 发起查询请求
[[BTKFenceAction sharedInstance] batchQueryServerFenceHistoryAlarmWith:request
delegate:self];
```

报警推送

当服务端围栏被触发之后，会通过长连接将报警信息推送给SDK，SDK会通过 `BTKActionDelegate` 协议的 `-(void)onGetPushMessage:(BTKPushMessage *)message;` 方法将报警信息推送给开发者。因此服务端围栏的报警推送要求网络畅通。当接收报警的手机断网或网络状态不好时，会导致报警推送失败，鹰眼服务端将在后续的10分钟之内每隔15s推送一次，直至收到成功响应。若10分钟之后仍未成功，将不再推送，但报警记录将存储在鹰眼服务端。为避免因此造成报警漏接收，开发者可定期使用历史报警查询接口同步报警信息。

以下的代码片段展示了报警信息的解析方法：

```
-(void)onGetPushMessage:(BTKPushMessage *)message {
    NSLog(@"收到推送消息，消息类型: %@", @(message.type));
    if (message.type == 0x03) {
        BTKPushMessageFenceAlarmContent *content = (BTKPushMessageFenceAlarmContent
*)message.content;
        if (content.actionType == BTK_FENCE_MONITORED_OBJECT_ACTION_TYPE_ENTER) {
            NSLog(@"被监控对象 %@ 进入 服务端地理围栏 %@ ", content.monitoredObject,
content.fenceName);
        } else if (content.actionType == BTK_FENCE_MONITORED_OBJECT_ACTION_TYPE_EXIT) {
            NSLog(@"被监控对象 %@ 离开 服务端地理围栏 %@ ", content.monitoredObject,
content.fenceName);
        }
```

```
    }
}
```

客户端围栏

客户端围栏的所有功能，均在SDK内部完成，不依赖网络。只要开启服务且开始采集之后，每个轨迹点参与客户端围栏的计算，若触发报警，会直接通过 `BTKActionDelegate` 协议的 `-(void)onGetPushMessage:(BTKPushMessage *)message;` 方法将报警信息推送给开发者。客户端围栏的历史报警信息也将存储在SDK中，开发者可以通过相应的方法查询客户端围栏的历史报警纪录。客户端历史报警纪录最多保存7天。

客户端地理围栏的管理，主要是指客户端围栏的新建、删除、修改、查询操作。

创建围栏

通过 `BTKFenceAction` 类中的 `-(void)createLocalFenceWith:(BTKCreateLocalFenceRequest *)request delegate:(id<BTKFenceDelegate>)delegate;` 方法创建客户端地理围栏。目前客户端地理围栏只有圆形，因此`BTKCreateLocalFenceRequest` 类只提供了1个构造方法 `-(instancetype)initWithLocalCircleFence:(BTKLocalCircleFence *)fence tag:(NSUInteger)tag;`，用于创建圆形地理围栏。

以下代码片段表示，创建一个名称为 “local_fence_A” 的客户端圆形地理围栏，圆心位于东经116.3134度、北纬40.0478度，半径为100米。它监控 “entityA” 这个终端实体，且没有设置去噪精度，也就是所有以“entityA”身份开启轨迹服务并开始采集后，采集到的轨迹点都会参与此地理围栏的计算。

```objc
// 围栏圆心
CLLocationCoordinate2D center = CLLocationCoordinate2DMake(40.0478, 116.3134);

// 新的客户端地理围栏对象
BTKLocalCircleFence *localFence = [[BTKLocalCircleFence alloc] initWithCenter:center radius:100 coordType:BTK_COORDTYPE_BD09LL denoiseAccuracy:0 fenceName:@"local_fence_A" monitoredObject:@"entityA"];

// 构建请求对象
BTKCreateLocalFenceRequest *request = [[BTKCreateLocalFenceRequest alloc] initWithLocalCircleFence:localFence tag:21];

// 创建客户端地理围栏
[[BTKFenceAction sharedInstance] createLocalFenceWith:request delegate:self];
```

删除围栏

通过 `BTKFenceAction` 类中的 `-(void)deleteLocalFenceWith:(BTKDeleteLocalFenceRequest *)request delegate:(id<BTKFenceDelegate>)delegate;` 方法删除客户端地理围栏。

以下代码片段表示，删除所有以“entityA” 这个终端实体为监控对象的客户端地理围栏。

```objc
// 构建请求对象
BTKDeleteLocalFenceRequest *request = [[BTKDeleteLocalFenceRequest alloc] initWithMonitoredObject:@"entityA" fenceIDs:nil tag:22];

// 删除客户端地理围栏
[[BTKFenceAction sharedInstance] deleteLocalFenceWith:request delegate:self];
```

修改围栏

通过 `BTKFenceAction` 类中的 `-(void)updateLocalFenceWith:(BTKUpdateLocalFenceRequest *)request delegate:(id<BTKFenceDelegate>)delegate;` 方法修改客户端地理围栏。

以下代码片段表示，将fenceID为15的客户端地理围栏的名称修改为“local_fence_A”，监控对象改为“entityB”，围栏的圆心坐标改为东经116.5167度，北纬40.1486度，围栏半径改为158米。同时将该客户端地理围栏的去噪精度改为10米，即定位精度超过10米的轨迹点将不再参与围栏计算。

```
// 新的围栏圆心
CLLocationCoordinate2D center = CLLocationCoordinate2DMake(40.1486, 116.5167);

// 新的客户端圆形地理围栏对象
BTKLocalCircleFence *localFence = [[BTKLocalCircleFence alloc] initWithCenter:center radius:158 coordType:BTK_COORDTYPE_BD09LL denoiseAccuracy:10 fenceName:@"local_fence_A" monitoredObject:@"entityB"];

// 构建请求对象
BTKUpdateLocalFenceRequest *request = [[BTKUpdateLocalFenceRequest alloc] initWithLocalFenceID:15 localCircleFence:localFence tag:23];

// 更新客户端地理围栏
[[BTKFenceAction sharedInstance] updateLocalFenceWith:request delegate:self];
```

查询围栏

通过 `BTKFenceAction` 类中的 `-(void)queryLocalFenceWith:(BTKQueryLocalFenceRequest *)request delegate:(id<BTKFenceDelegate>)delegate;` 方法查询客户端地理围栏。该方法可以根据指定的监控对象或围栏ID，列出满足条件的地理围栏。

以下代码片段表示，查询所有监控对象为“entityA”的客户端地理围栏。

```
// 构建请求对象
BTKQueryLocalFenceRequest *request = [[BTKQueryLocalFenceRequest alloc] initWithMonitoredObject:@"entityA" fenceIDs:nil tag:24];

// 查询客户端地理围栏
[[BTKFenceAction sharedInstance] queryLocalFenceWith:request delegate:self];
```

实时状态查询

鹰眼iOS SDK支持查询指定监控对象和客户端围栏的位置关系。开发者可以借此查询当前终端是在客户端地理围栏的内部还是外部。

使用`-(void)queryLocalFenceStatusWith:(BTKQueryLocalFenceStatusRequest *)request delegate:(id<BTKFenceDelegate>)delegate;` 方法，查询被监控对象的状态时，被监控对象的位置以其最后一次采集的位置为准。假设 "entityA" 和 “entityB” 均创建了客户端地理围栏，先使用 “entityA” 的身份开启轨迹服务后采集轨迹一段时间，在坐标A处停止轨迹服务，再以 “entityB” 的身份开启轨迹服务并开始采集轨迹，若当设备已经移动到坐标B处，查询被监控对象 “entityA” 的状态，则会按照A处的坐标来计算 “entityA” 相对其客户端地理围栏的位置关系。因为每个采集到的轨迹点，仅属于当前开启轨迹服务时，所指定的那个终端实体（entityName）。

以下代码片段表示，查询终端实体 “entityA” 和所有以 “entityA” 为监控对象的客户端地理围栏的位置关系。

```
// 构建请求对象
BTKQueryLocalFenceStatusRequest *request = [[BTKQueryLocalFenceStatusRequest alloc] initWithMonitoredObject:@"entityA" fenceIDs:nil tag:111];

// 发起查询请求
[[BTKFenceAction sharedInstance] queryLocalFenceStatusWith:request delegate:self];
```

使用`-(void)queryLocalFenceStatusByCustomLocationWith:`

`(BTKQueryLocalFenceStatusByCustomLocationRequest *)request delegate:(id<BTKFenceDelegate>)delegate;` 方法，可以假设被监控对象处于某自定义的位置坐标时，其和客户端地理围栏的位置关系。

以下代码片段表示，假设“entityA”这个终端实体在东经116.5167度，北纬40.1486度的话，该终端实体和其上所有客户端地理围栏的位置关系。

```
// 被监控对象的模拟位置
CLLocationCoordinate2D customLocation = CLLocationCoordinate2DMake(40.1486, 116.5167);

// 构建请求对象
BTKQueryLocalFenceStatusByCustomLocationRequest *request =
[[BTKQueryLocalFenceStatusByCustomLocationRequest alloc]
initWithmonitoredObject:@"entityA" CustomLocation:customLocation
coordType:BTK_COORDTYPE_BD09LL fenceIDs:nil tag:42];

// 发起查询请求
[[BTKFenceAction sharedInstance] queryLocalFenceStatusByCustomLocationWith:request
delegate:self];
```

历史报警查询

鹰眼iOS SDK支持查询指定监控对象，在指定时间段内，客户端地理围栏的历史报警记录。使用`-(void)queryLocalFenceHistoryAlarmWith:(BTKQueryLocalFenceHistoryAlarmRequest *)request delegate:(id<BTKFenceDelegate>)delegate;` 方法，查询某监控对象的历史报警信息。客户端地理围栏的历史报警纪录最多保存7天。SDK会不定时地清空7天以前的历史报警纪录，因此超出此时间范围的历史报警纪录不保证可以查询到。

以下代码片段表示，查询名称为 “entityA” 的终端实体，在过去1小时内的历史报警纪录。

```
NSUInteger endTime = [[NSDate date] timeIntervalSince1970] - 3600;

// 构建请求对象
BTKQueryLocalFenceHistoryAlarmRequest *request = [[BTKQueryLocalFenceHistoryAlarmRequest
alloc] initWithMonitoredObject:@"entityA" fenceIDs:nil startTime:endTime - 60 * 60
endTime:endTime tag:333];

// 发起查询请求
[[BTKFenceAction sharedInstance] queryLocalFenceHistoryAlarmWith:request delegate:self];
```

报警推送

客户端地理围栏不依赖网络，因此每个采集到的轨迹点，都会在第一时间参与客户端围栏的计算，若触发报警，则和服务端报警一样，通过 `BTKActionDelegate` 协议的 `-(void)onGetPushMessage:(BTKPushMessage *)message;` 方法将报警信息推送给开发者。不同的是，`BTKPushMessage` 中表示推送消息类型的 `type` 字段的值不同。

以下代码片段展示了，如何解析推送信息中的客户端地理围栏报警推送。

```
-(void)onGetPushMessage:(BTKPushMessage *)message {
    NSLog(@"收到推送消息，消息类型: %@", @(message.type));
    if (message.type == 0x04) {
        BTKPushMessageFenceAlarmContent *content = (BTKPushMessageFenceAlarmContent
*)message.content;
        if (content.actionType == BTK_FENCE_MONITORED_OBJECT_ACTION_TYPE_ENTER) {
            NSLog(@"被监控对象 %@ 进入 客户端地理围栏 %@ ", content.monitoredObject,
content.fenceName);
        } else if (content.actionType == BTK_FENCE_MONITORED_OBJECT_ACTION_TYPE_EXIT) {
```

```
            NSLog(@"被监控对象 %@ 离开 客户端地理围栏 %@ ", content.monitoredObject, content.fenceName);
        }
    }
}
```