

开发者说 | Apollo感知分析之如何进行HM对象跟踪

原创：Apollo社区开发者 Apollo开发者社区 2018-12-12



任何一个系统的感知算法里，仅仅有深度学习是不够的，一定要有数据驱动模型计算（模式较重，需要收集数据），同时也有面向下游的、后处理的计算（模式轻，见效快）。

后处理的计算主要分3种，分别是2D-to-3D的几何计算、时序计算、多相机环视融合。

2D-to-3D的几何计算（分割的方法）：

- 受相机pose的影响（相机自身仰俯、消失点判断）
- 接地点（前提：相机水平、地面水平、相机高度已知情况下，可通过三角形解答，同时也可看出相机pitch角对回3D的影响），2D框（假设每辆车有8个顶点，将每个顶点都投到图像当中，最终形成的最小外接矩阵即2D框，可用于给障碍物约束边界），绝对尺寸多条途径回3D（最终融合得出精准的结果）的影响
- 稳定性至关重要（安全驾驶的保证）

时序信息计算（跟踪要求至少15fps，高速情况下要求更高）：

- 对相机帧率和延时有要求（必须很低）（总耗时=分耗时×障碍物个数）
- 充分利用检测模型的输出信息：特征（feature map可理解为对图像高层语义的刻画，再将feature map做roi pooling后可得到障碍物级别的feature，将这些feature用来做跟踪和关联，就可以消化掉传统跟踪中计算量很大的提取特征步骤）、类别（输出是人还是车，在跟踪关联也是有幫助的，因相应来说这一帧后下一帧的类别不会发生跳变，所以这种信息要加以利用）
- 可以考虑轻量级Metric Learning（因为CNN feature并未对instance做出区分，所以在其中可加入一些轻量级Metric Learning来做专门针对产品的feature）

多相机的环视融合：

- 相机布局决定融合策略（环式布局严重影响融合策略，做好视野重叠，即做好冗余才能保证回3D和识别质量）

在后处理的计算中，2D-to-3D的几何计算、时序计算、多相机环视融合等因素密不可分，模块之间的算法需相通。这也要求开发团队积极配合，形成一个有机的整体。

以下是Apollo社区开发者朱炎亮在Github-Apollo-Note上分享的《HM对象跟踪》，感谢他为我们tracking这一步所做的详细注解和释疑。

面对复杂多变、快速迭代的开发环境，只有开放才会带来进步，Apollo社区正在被开源的力量唤醒。



HM对象跟踪器跟踪分段检测到的障碍物。通常，它通过将当前检测与现有跟踪列表相关联，来形成和更新跟踪列表，如不再存在，则删除旧的跟踪列表，并在识别出新的检测时生成新的跟踪列表。更新后的跟踪列表的运动状态将在关联后进行估计。在HM对象跟踪器中，匈牙利算法(Hungarian algorithm)用于检测到跟踪关联，并采用鲁棒卡尔曼滤波器(Robust Kalman Filter)进行运动估计。

上述是Apollo官方文档对HM对象跟踪的描述，这部分意思比较明了，主要的跟踪流程可以分为：

- **预处理**；(lidar->local ENU坐标系变换、跟踪对象创建、跟踪目标保存)
- **卡尔曼滤波器滤波**，预测物体当前位置与速度；(卡尔曼滤波阶段1：Predict阶段)
- **匈牙利算法匹配**，关联检测物体和跟踪物体；
- **卡尔曼滤波**，更新跟踪物体位置与速度信息。(卡尔曼滤波阶段2：Update阶段)

进入HM物体跟踪的入口依旧在 `LidarProcessSubnode::OnPointCloud` 中：

```
1  /// file in apollo/modules/perception/obstacle/onboard/lidar_process_subnode.cc
2  void LidarProcessSubnode::OnPointCloud(const sensor_msgs::PointCloud2& message) {
```

```
3    /// call hdmap to get ROI
4    ...
5    /// call roi_filter
6    ...
7    /// call segmentor
8    ...
9    /// call object builder
10   ...
11   /// call tracker
12   if (tracker_ != nullptr) {
13       TrackerOptions tracker_options;
14       tracker_options.velodyne_trans = velodyne_trans;
15       tracker_options.hdmap = hdmap;
16       tracker_options.hdmap_input = hdmap_input_;
17       if (!tracker_>Track(objects, timestamp_, tracker_options, &(out_sensor_objects->objec
18       ...
19   }
20   }
21 }
```

在这部分，总共有三个比较绕的对象类，分别是**Object**、**TrackedObject**和**ObjectTrack**，在这里统一说明一下区别：

- **Object类**：常见的物体类，里面包含物体原始点云、多边形轮廓、物体类别、物体分类置信度、方向、长宽、速度等信息。[全模块通用](#)。
- **TrackedObject类**：封装Object类，记录了跟踪物体类属性，额外包含了中心、重心、速度、加速度、方向等信息。
- **ObjectTrack类**：封装了TrackedObject类，实际的跟踪解决方案，不仅包含了需要跟踪的物体(TrackedObject)，同时包含了跟踪物体滤波、预测运动趋势等函数。

所以可以看到，跟踪过程需要将原始Object封装成TrackedObject，创立跟踪对象；最后跟踪对象创立跟踪过程ObjectTrack，可以通过ObjectTrack里面的函数来对ObjectTrack所标记的TrackedObject进行跟踪。

STEP 1. 预处理



```
1  /// file in apollo/modules/perception/obstacle/onboard/lidar_process_subnode.cc
2  void LidarProcessSubnode::OnPointCloud(const sensor_msgs::PointCloud2& message) {
3      /// call hdmap to get ROI
4      ...
5      /// call roi_filter
6      ...
7      /// call segmentor
8      ...
9      /// call object builder
10     ...
11     /// call tracker
12     if (tracker_ != nullptr) {
13         TrackerOptions tracker_options;
14         tracker_options.velodyne_trans = velodyne_trans;
15         tracker_options.hdmap = hdmap;
16         tracker_options.hdmap_input = hdmap_input_;
17         if (!tracker_->Track(objects, timestamp_, tracker_options, &(out_sensor_objects->object
18         ...
19     }
20 }
21 } /// file in apollo/modules/perception/obstacle/lidar/tracker/hm_tracker/hm_tracker.cc
22 bool HmObjectTracker::Track(const std::vector& objects,
23                             double timestamp, const TrackerOptions& options,
24                             std::vector* tracked_objects) {
25     // A. track setup
26     if (!valid_) {
27         valid_ = true;
28         return Initialize(objects, timestamp, options, tracked_objects);
29     }
30     // B. preprocessing
31     // B.1 transform given pose to local one
32     TransformPoseGlobal2Local(&velo2world_pose);
33     // B.2 construct objects for tracking
34     std::vector transformed_objects;
35     ConstructTrackedObjects(objects, &transformed_objects, velo2world_pose, options);
36     ...
```

```
37 }
38
39 bool HmObjectTracker::Initialize(const std::vector& objects,
40                                 const double& timestamp,
41                                 const TrackerOptions& options,
42                                 std::vector* tracked_objects) {
43     global_to_local_offset_ = Eigen::Vector3d(-velo2world_pose(0, 3), -velo2world_pose(1, 3)
44     // B. preprocessing
45     // B.1 coordinate transformation
46     TransformPoseGlobal2Local(&velo2world_pose);
47     // B.2 construct tracked objects
48     std::vector transformed_objects;
49     ConstructTrackedObjects(objects, &transformed_objects, velo2world_pose, options);
50     // C. create tracks
51     CreateNewTracks(transformed_objects, unassigned_objects);
52     time_stamp_ = timestamp;
53     // D. collect tracked results
54     CollectTrackedResults(tracked_objects);
55     return true;
56 }
```

预处理阶段主要分两个模块：A.跟踪建立(track setup)和B.预处理(preprocess)。跟踪建立过程，主要是对上述得到的物体对象进行跟踪目标的建立，这是Track第一次被调用的时候进行的，后续只需要进行跟踪对象更新即可。建立过程相对比较简单，主要包含：

1. 物体对象坐标系转换(原先的lidar坐标系-->lidar局部ENU坐标系/有方向性)
2. 对每个物体创建跟踪对象，加入跟踪列表
3. 记录现在被跟踪的对象

从上面代码来看，预处理阶段两模块重复度很高，这里我们就介绍 Initialize 对象跟踪建立函数。



第一步是进行坐标系的变换

这里我们注意到一个平移向量 `global_to_local_offset_`，他是lidar坐标系到世界坐标系的变换矩

阵 `velo2world_trans` 的平移成分，前面高精地图ROI过滤器小节我们讲过：**local局部ENU坐标系跟world世界坐标系之间只有平移成分，没有旋转。所以这里取了转变矩阵的平移成分，其实就是world世界坐标系转换到lidar局部ENU坐标系的平移矩阵(变换矩阵)。**
$$P_local = P_world + global_to_local_offset_$$

```
1  /// file in apollo/modules/perception/obstacle/lidar/tracker/hm_tracker/hm_tracker.cc
2  void HmObjectTracker::TransformPoseGlobal2Local(Eigen::Matrix4d* pose) {
3      (*pose)(0, 3) += global_to_local_offset_(0);
4      (*pose)(1, 3) += global_to_local_offset_(1);
5      (*pose)(2, 3) += global_to_local_offset_(2);
6  }
```

从上面的 `TransformPoseGlobal2Local` 函数代码我们可以得到一个没有平移成分，只有旋转成分的变换矩阵 `velo2world_pose`，这个矩阵有什么作用？很简单，**这个矩阵就是lidar坐标系到lidar局部ENU坐标系的转换矩阵。**

02

第二步中需要根据前面CNN检测到的物体来创建跟踪对象

也就是将 `Object` 包装到 `TrackedObject` 中，那我们先来看一下 `TrackedObject` 类里面的成分：

名称	备注
<code>ObjectPtr object_ptr</code>	<code>Object</code> 对象指针
<code>Eigen::Vector3f barycenter</code>	重心，取该类所有点云xyz的平均值得到
<code>Eigen::Vector3f center</code>	中心，bbox4个角点外加平均高度计算得到
<code>Eigen::Vector3f velocity</code>	速度，卡尔曼滤波器预测得到
<code>Eigen::Matrix3f velocity_uncertainty</code>	不确定速度
<code>Eigen::Vector3f acceleration</code>	加速度
<code>ObjectType type</code>	物体类型，行人、自行车、车辆等

名称	备注
float association_score	--

从上面表格可以看到， TrackedObject 封装了 object ， 并且只增加了少量速度， 加速度等额外信息。

```

6  /// file in apollo/modules/perception/obstacle/lidar/tracker/hm_tracker/hm_tracker.cc
7  void HmObjectTracker::ConstructTrackedObjects(
8      const std::vector& objects,
9      std::vector* tracked_objects, const Eigen::Matrix4d& pose,
10     const TrackerOptions& options) {
11     int num_objects = objects.size();
12     tracked_objects->clear();
13     tracked_objects->resize(num_objects);
14     for (int i = 0; i < num_objects; ++i) {
15         ObjectPtr obj(new Object());
16         obj->clone(*objects[i]);
17         (*tracked_objects)[i].reset(new TrackedObject(obj));           // create new Tr
18         // Computing shape feature
19         if (use_histogram_for_match_) {
20             ComputeShapeFeatures(&((*tracked_objects)[i]));           // compute shape
21         }
22         // Transforming all tracked objects
23         TransformTrackedObject(&((*tracked_objects)[i]), pose);         // transform coc
24         // Setting barycenter as anchor point of tracked objects
25         Eigen::Vector3f anchor_point = (*tracked_objects)[i]->barycenter;
26         (*tracked_objects)[i]->anchor_point = anchor_point;
27         // Getting lane direction of tracked objects
28         pcl_util::PointD query_pt;                                     // get lidar's v
29         query_pt.x = anchor_point(0) - global_to_local_offset_(0);
30         query_pt.y = anchor_point(1) - global_to_local_offset_(1);
31         query_pt.z = anchor_point(2) - global_to_local_offset_(2);
32         Eigen::Vector3d lane_dir;
33         if (!options.hdmap_input->GetNearestLaneDirection(query_pt, &lane_dir)) {
34             lane_dir = (pose * Eigen::Vector4d(1, 0, 0, 0)).head(3);   // get nearest l
35         }
36         (*tracked_objects)[i]->lane_direction = lane_dir.cast<float>();
37     }
38 }

```

`ConstructTrackedObjects` 是由物体对象来创建跟踪对象的代码，这个过程相对来说比较简单易懂，没大的难点，下面就解释一下具体的功能。

- 针对 `vector& objects` 中的每个对象，创建对应的 `TrackedObject`，并且计算他的shape feature，这个特征计算比较简单，先计算物体xyz三个坐标轴上的最大和最小值，分别将其

划分成10等份，对每个点xyz坐标进行bins投影与统计。最后的到的特征就是
[x_bins,y_bins,z_bins]一共30维，归一化(除点云数量)后得到最终的shape feature

- TransformTrackedObject 函数进行跟踪物体的方向、中心、原始点云、多边形角点、重心等进行坐标系转换。Lidar坐标系变换到local ENU坐标系
- 根据lidar的世界坐标系坐标查询高精地图HD Map计算车道线方向

03 建的跟踪对象(TrackedObject)建立跟踪, 正式进行跟踪(加入进ObjectTrack)

```

1  /// file in apollo/modules/perception/obstacle/lidar/tracker/hm_tracker/hm_tracker.cc
2  void HmObjectTracker::CreateNewTracks(
3      const std::vector& new_objects,
4      const std::vector<int>& unassigned_objects) {
5      // Create new tracks for objects without matched tracks
6      for (size_t i = 0; i < unassigned_objects.size(); i++) {
7          int obj_id = unassigned_objects[i];
8          ObjectTrackPtr track(new ObjectTrack(new_objects[obj_id]));
9          object_tracks_.AddTrack(track);
10     }
11 }
```

同时函数 CollectTrackedResults 会将当前正在跟踪的对象(世界坐标系坐标形式)保存到向量中，该部分代码比较简单就不贴出来了。

STEP 2. 卡尔曼滤波, 跟踪物体对象 (卡尔曼滤波阶段1: Predict)



在预处理阶段，每个物体Object类经过封装以后，产生一个对应的ObjectTrack过程类，里面封装了对应要跟踪的物体(TrackedObject，由

Object封装而来)。这个阶段的工作就是对跟踪物体TrackedObject进行卡尔曼滤波并预测其运动方向。

首先，在这里我们简单介绍一下卡尔曼滤波的一些基础公式，方便下面理解。

一个系统拥有一个状态方程和一个观测方程。观测方程是我们能宏观看到的一些属性，在这里比如说汽车重心xyz的位置和速度；而状态方程是整个系统里面的一些状态，包含能观测到的属性(如汽车重心xyz的位置和速度)，也可能包含其他一些看不见的属性，这些属性甚至我们都不能去定义它的物理意义。**因此观测方程的属性是状态方程的属性的一部分**现在有：

状态方程： $X_t = A_{t,t-1}X_{t-1} + W_t$ ，其中 $W_t \sim N(0, Q)$

观测方程： $Z_t = C_tX_t + V_t$ ，其中 $V_t \sim N(0, R)$

卡尔曼滤波分别两个阶段，分别是预测Predict与更新Update：

- **Predict预测阶段**
 - 利用上时刻t-1最优估计 X_{t-1} 预测当前时刻状态 $X_{t,t-1} = A_{t,t-1}X_{t-1}$ ，这个 $X_{t,t-1}$ 不是t时刻的最优状态，只是估计出来的状态
 - 利用上时刻t-1最优协方差矩阵 P_{t-1} 预测当前时刻协方差矩阵 $P_{t,t-1} = A_{t,t-1}P_{t-1}A_{t,t-1}^T + Q$ ，这个 $P_{t,t-1}$ 也不是t时刻最优协方差
- **Update更新阶段**
 - 利用 $X_{t,t-1}$ 估计出t时刻最优状态 $X_t = X_{t,t-1} + H_t[Z_t - C_tX_{t,t-1}]$ ，其中 $H_t = P_{t,t-1}C_t^T[C_tP_{t,t-1}C_t^T + R]^{-1}$
 - 利用 $P_{t,t-1}$ 估计出t时刻最优协方差矩阵 $P_t = [I - H_tC_t]P_{t,t-1}$

最终t从1开始递归计算k时刻的最优状态 X_k 与最优协方差矩阵 P_t

```
1  /// file in apollo/modules/perception/obstacle/lidar/tracker/hm_tracker/hm_tracker.cc
2  bool HmObjectTracker::Track(const std::vector& objects,
3                               double timestamp, const TrackerOptions& options,
4                               std::vector* tracked_objects) {
5      // A. track setup
6      ...
7      // B. preprocessing
8      // B.1 transform given pose to local one
```

```
9    ...
10   // B.2 construct objects for tracking
11   ...
12   // C. prediction
13   std::vector tracks_predict;
14   ComputeTracksPredict(&tracks_predict, time_diff);
15   ...
16 }
17
18 void HmObjectTracker::ComputeTracksPredict(std::vector* tracks_predict, const double& time
19     // Compute tracks' predicted states
20     std::vector& tracks = object_tracks_.GetTracks();
21     for (int i = 0; i < no_track; ++i) {
22         (*tracks_predict)[i] = tracks[i]->Predict(time_diff);    // track every tracked object
23     }
24 }
```

从代码中我们可以看到，这个过程其实就是对object_tracks_列表中每个物体调用其Predict函数进行滤波跟踪(object_tracks_是上阶段Object--TrackedObject--ObjectTrack的依次封装)。接下来我们就对这个Predict函数进行深层次的挖掘和分析，看看它实现了卡尔曼过滤器的哪个阶段工作。

```
1  /// file in apollo/modules/perception/obstacle/lidar/tracker/hm_tracker/object_track.cc
2  Eigen::VectorXf ObjectTrack::Predict(const double& time_diff) {
3      // Get the predict of filter
4      Eigen::VectorXf filter_predict = filter_->Predict(time_diff);
5      // Get the predict of track
6      Eigen::VectorXf track_predict = filter_predict;
7      track_predict(0) = belief_anchor_point_(0) + belief_velocity_(0) * time_diff;
8      track_predict(1) = belief_anchor_point_(1) + belief_velocity_(1) * time_diff;
9      track_predict(2) = belief_anchor_point_(2) + belief_velocity_(2) * time_diff;
10     track_predict(3) = belief_velocity_(0);
11     track_predict(4) = belief_velocity_(1);
12     track_predict(5) = belief_velocity_(2);
13     return track_predict;
14 }
15
16 /// file in apollo/modules/perception/obstacle/lidar/tracker/hm_tracker/kalman_filter.cc
17 Eigen::VectorXf KalmanFilter::Predict(const double& time_diff) {
```

```

18     // Compute predict states
19     Eigen::VectorXf predicted_state;
20     predicted_state.resize(6);
21     predicted_state(0) = belief_anchor_point_(0) + belief_velocity_(0) * time_diff;
22     predicted_state(1) = belief_anchor_point_(1) + belief_velocity_(1) * time_diff;
23     predicted_state(2) = belief_anchor_point_(2) + belief_velocity_(2) * time_diff;
24     predicted_state(3) = belief_velocity_(0);
25     predicted_state(4) = belief_velocity_(1);
26     predicted_state(5) = belief_velocity_(2);
27     // Compute predicted covariance
28     Propagate(time_diff);
29     return predicted_state;
30 }
31
32 void KalmanFilter::Propagate(const double& time_diff) {
33     // Only propagate tracked motion
34     ity_covariance_ += s_propagation_noise_ * time_diff * time_diff;
35 }

```

从上面两个函数可以明显看到这个阶段就是卡尔曼滤波器的Predict阶段。同时可以看到：

1. track_predict/predicted_state 相当于卡尔曼滤波其中的 $X_{t,t-1}$ ， belief_anchor_point_ 和 belief_velocity_ 相当于 X_t ， ity_covariance_ 交替存储 P_t 和 P_{t-1} (Why?可以从上面的卡尔曼滤波器公式看到 P_t 在估测完 P_{t-1} 以后就没用了，所以可以覆盖存储，节省部分空间)
2. 状态方程和观测方程其实本质上是一样，也就是相同维度的。都是6维，分别表示重心的xyz坐标和重心xyz的速度。同时在这个应用中，短时间间隔内。当前时刻重心位置=上时刻重心位置 + 上时刻速度*时间差，所以可知卡尔曼滤波器中 $A_{t,t-1} \equiv 1$ ， $Q = I * \Delta t^2$
3. 该过程工作:首先利用上时刻的最优估计 belief_anchor_point_ 和 belief_velocity_ (等同于 X_{t-1})估计出t时刻的状态 predicted_state (等同于 $X_{t,t-1}$)； 然后估计当前时刻的协方差矩 ity_covariance_ (P_{t-1} 和 $P_{t,t-1}$ 交替存储)

STEP 3. 匈牙利算法匹配, 关联检测物体和跟踪物体



```
1  /// file in apollo/modules/perception/obstacle/lidar/tracker/hm_tracker/hm_tracker.cc
2  bool HmObjectTracker::Track(const std::vector& objects,
3                               double timestamp, const TrackerOptions& options,
4                               std::vector* tracked_objects) {
5      // A. track setup
6      ...
7      // B. preprocessing
8      // B.1 transform given pose to local one
9      ...
10     // B.2 construct objects for tracking
11     ...
12     // C. prediction
13     ...
14     // D. match objects to tracks
15     std::vector assignments;
16     std::vector<int> unassigned_objects;
17     std::vector<int> unassigned_tracks;
18     std::vector& tracks = object_tracks_.GetTracks();
19     if (matcher_ != nullptr) {
20         matcher_->Match(&transformed_objects, tracks, tracks_predict, &assignments, &unassigned_
21     }
22     ...
23 }
24
25 /// file in apollo/modules/perception/obstacle/lidar/tracker/hm_tracker/hungarian_matcher.
26 void HungarianMatcher::Match(std::vector* objects,
27                               const std::vector& tracks,
28                               const std::vector& tracks_predict,
29                               std::vector* assignments,
30                               std::vector<int>* unassigned_tracks,
31                               std::vector<int>* unassigned_objects) {
32     // A. computing association matrix
33     Eigen::MatrixXf association_mat(tracks.size(), objects->size());
34     ComputeAssociateMatrix(tracks, tracks_predict, (*objects), &association_mat);
35 }
```

```

36 // B. computing connected components
37 std::vector<std::vector<int>> object_components;
38 std::vector<std::vector<int>> track_components;
39 ComputeConnectedComponents(association_mat, s_match_distance_maximum_,
40                             &track_components, &object_components);
41 // C. matching each sub-graph
42 ...
43 }

```

这个阶段主要的工作是**匹配CNN分割+MinBox检测到的物体和当前ObjectTrack的跟踪物体**。主要的工作为：

- A. Object和TrackedObject之间关联矩阵 `association_mat` 计算
- B. 子图划分，利用上述的关联矩阵和设定的阈值(两两评分小于阈值则互相关联，即节点之间链接)，将矩阵分割成一系列子图
- C. 匈牙利算法进行二分图匹配，得到cost最小的(Object,TrackedObject)连接对

01 关联矩阵`association_mat`计算

- 重心位置坐标距离差异评分
- 物体方向差异评分
- 标定框尺寸差异评分
- 点云数量差异评分
- 外观特征差异评分

最终以0.6, 0.2, 0.1, 0.1, 0.5的权重加权求和得到关联评分。

```

1 /// file in apollo/modules/perception/obstacle/lidar/tracker/hm_tracker/track_object_distance.cpp
2 float TrackObjectDistance::ComputeDistance(const ObjectTrackPtr& track,
3                                             const Eigen::VectorXf& track_predict,
4                                             const TrackedObjectPtr& new_object) {
5     // Compute distance for given track & object
6     float location_distance = ComputeLocationDistance(track, track_predict, new_object);

```

```

7     float direction_distance = ComputeDirectionDistance(track, track_predict, new_object);
8     float bbox_size_distance = ComputeBboxSizeDistance(track, new_object);
9     float point_num_distance = ComputePointNumDistance(track, new_object);
10    float histogram_distance = ComputeHistogramDistance(track, new_object);
11
12    float result_distance = s_location_distance_weight_ * location_distance +           //
13                           s_direction_distance_weight_ * direction_distance +       //
14                           s_bbox_size_distance_weight_ * bbox_size_distance +       //
15                           s_point_num_distance_weight_ * point_num_distance +      //
16                           s_histogram_distance_weight_ * histogram_distance;       //
17    return result_distance;
18 }

```

各个子项的计算方式，这里以文字形式描述，假设：

Object重心坐标为 (x_1, y_1, z_1) ，方向为 (dx_1, dy_1, dz_1) ，bbox尺寸为 (l_1, w_1, h_1) ，shape feature为30维向量 sf_1 ，包含原始点云数量 n_1 。

TrackedObject重心坐标为 (x_2, y_2, z_2) ，方向为 (dx_2, dy_2, dz_2) ，bbox尺寸为 (l_2, w_2, h_2) ，shape feature为30维向量 sf_2 ，包含原始点云数量 n_2 。

则有：

- 重心位置坐标距离差异评分 $location_distance$ 计算

$$location_distance = \sqrt{\{(x_1 - x_2)\}^2 + \{(y_1 - y_2)\}^2}$$

如果速度太大，则需要用方向向量去正则惩罚，具体可以参考代码

- 物体方向差异评分 $direction_distance$ 计算

方向差异其实就是计算两个向量的夹角：

$$\cos\theta = a \cdot b / (|a| \cdot |b|)$$

夹角越大，差异越大， \cos 值越小；夹角越大，差异越大， \cos 值越大

最后使用 $1 - \cos$ 计算评分，差异越小，评分越大。

- 标定框尺寸差异评分 $bbox_size_distance$ 计算

代码中首先计算两个量 dot_val_00 和 dot_val_01 ：

```

1  /// file in apollo/master/modules/perception/obstacle/lidar/tracker/hm_tracker/track_objec
2  float TrackObjectDistance::ComputeBboxSizeDistance(const ObjectTrackPtr& track, const Trac
3      double dot_val_00 = fabs(old_bbox_dir(0) * new_bbox_dir(0) + old_bbox_dir(1) * new_bbox_
4      double dot_val_01 = fabs(old_bbox_dir(0) * new_bbox_dir(1) - old_bbox_dir(1) * new_bbox_
5      bool bbox_dir_close = dot_val_00 > dot_val_01;
6
7      if (bbox_dir_close) {
8          float diff_1 = fabs(old_bbox_size(0) - new_bbox_size(0)) / std::max(old_bbox_size(0),
9          float diff_2 = fabs(old_bbox_size(1) - new_bbox_size(1)) / std::max(old_bbox_size(1),
10         size_distance = std::min(diff_1, diff_2);
11     } else {
12         float diff_1 = fabs(old_bbox_size(0) - new_bbox_size(1)) / std::max(old_bbox_size(0),
13         float diff_2 = fabs(old_bbox_size(1) - new_bbox_size(0)) / std::max(old_bbox_size(1),
14         size_distance = std::min(diff_1, diff_2);
15     }
16     return size_distance;
17 }

```

这两个量有什么意义？这里简单解释一下，从计算方式可以看到：

其实 `dot_val_00` 是两个坐标的点积，数学计算形式上是方向1投影到方向2向量上得到向量3，最后向量3模乘以方向2模长，这么做可以估算方向差异。因为，当方向1和方向2两个向量夹角靠近0或180度时，投影向量很长，`dot_val_00` 这个点积的值会很大。**`dot_val_00` 越大说明两个方向越接近。**

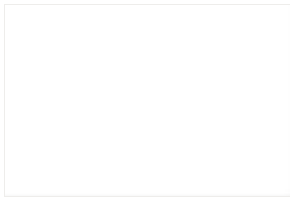
同理 `dot_val_01` 上文我们提到过，差积的模可以衡量两个向量组成的四边形面积大小，这么做也可以估算方向差异。因为，当方向1和方向2两个向量夹角靠近90度时，组成的四边形面积最大，`dot_val_01` 这个差积的模会很大。**`dot_val_00` 越大说明两个方向越背离。**

- 点云数量差异评分 `point_num_distance` 计算

$$\text{point-num_distance} = |n1 - n2| / \max(n1, n2)$$

- 外观特征差异评分 `histogram_distance` 计算

$$\text{histogram-distance} = \sum_{m=0}^{30} |sf1[m] - sf2[m]|$$



子图划分首先根据上步骤计算的 `association_mat` 矩阵，利用超参数 `s_match_distance_maximum_4`，关联值小于阈值的判定为连接，否则不连接。最终得到的连接矩阵大小为 $(N+M) \times (N+M)$

```
1 void HungarianMatcher::ComputeConnectedComponents(  
2     const Eigen::MatrixXf& association_mat, const float& connected_threshold,  
3     std::vector<std::vector<int>>* track_components,  
4     std::vector<std::vector<int>>* object_components) {  
5     // Compute connected components within given threshold  
6     int no_track = association_mat.rows();  
7     int no_object = association_mat.cols();  
8     std::vector<std::vector<int>> nb_graph;  
9     nb_graph.resize(no_track + no_object);  
10    for (int i = 0; i < no_track; i++) {  
11        for (int j = 0; j < no_object; j++) {  
12            if (association_mat(i, j) <= connected_threshold) {  
13                nb_graph[i].push_back(no_track + j);  
14                nb_graph[j + no_track].push_back(i);  
15            }  
16        }  
17    }  
18  
19    std::vector<std::vector<int>> components;  
20    ConnectedComponentAnalysis(nb_graph, &components); // sub_graph segment  
21    ...  
22 }
```

主要的子图划分工作在 `ConnectedComponentAnalysis` 函数完成，具体的可以参考代码，一个比较简单地广度优先搜索。最后得到的 `components` 二维向量中，每一行为一个子图的组成元素。

03 匈牙利算法对每个子图匹配

匹配的算法主要还是匈牙利算法的矩阵形式，跟wiki百科的基本描述一致，可以参考主页[匈牙利算](#)

法。

STEP 4.

卡尔曼滤波， 更新跟踪物体位置与速度信息 (卡尔曼滤波阶段2:Update阶段)



这个阶段做的工作比较重要，对上述Hungarian Matcher得到的追踪对。

- 计算真实的观测变量，包括真实观测到的车速度与加速度 $\{Zv_t\}$ 、 $\{Za_t\}$
- 由上时刻最优速度与加速度 $\{Xv_{t-1}\}$ 、 $\{Xa_{t-1}\}$ 估测当前时刻的速度与加速度 $\{Xv_{t,t-1}\}$ 、 $\{Xa_{t,t-1}\}$
- 由估测速度与加速度 $\{Xv_{t,t-1}\}$ 、 $\{Xa_{t,t-1}\}$ ，更新得到t时刻最优速度与加速度 $\{Xv_t\}$ 、 $\{Xa_t\}$

另外这里还要一个说明，ObjectTrack类不仅封装了TrackedObject类，同时也封装了KalmanFilter类，KalmanFilter自己保存了上时刻最优状态，上时刻最优协方差矩阵，当前时刻最优状态等信息。TrackedObject、ObjectTrack里面也有这些状态。

注意：KalmanFilter里面是滤波后的原始数据(没有实际应用的限制条件加入)；TrackedObject和ObjectTrack类同样保存了一份状态信息，这些状态信息是从KalmanFilter中得到的原始信息，并且加入实际应用限制滤波以后的状态信息；ObjectTrack类里面的状态是需要依赖TrackedObject类的，所以务必先更新TrackedObject再更新ObjectTrack的状态。

总之三者状态更新顺序为：

KalmanFilter -> TrackedObject -> ObjectTrack

```
1  /// file in apollo/modules/perception/obstacle/lidar/tracker/hm_tracker/hm_tracker.cc
2  bool HmObjectTracker::Track(const std::vector& objects,
3                               double timestamp, const TrackerOptions& options,
4                               std::vector* tracked_objects) {
5      // E. update tracks
```

```
6 // E.1 update tracks with associated objects
7 UpdateAssignedTracks(&tracks_predict, &transformed_objects, assignments, time_diff);
8 // E.2 update tracks without associated objects
9 UpdateUnassignedTracks(tracks_predict, unassigned_tracks, time_diff);
10 DeleteLostTracks();
11 // E.3 create new tracks for objects without associated tracks
12 CreateNewTracks(transformed_objects, unassigned_objects);
13 // F. collect tracked results
14 CollectTrackedResults(tracked_objects);
15 return true;
16 }
17
18 void HmObjectTracker::UpdateAssignedTracks(
19     std::vector* tracks_predict,
20     std::vector* new_objects,
21     const std::vector& assignments, const double& time_diff) {
22 // Update assigned tracks
23 std::vector& tracks = object_tracks_.GetTracks();
24 for (size_t i = 0; i < assignments.size(); i++) {
25     int track_id = assignments[i].first;
26     int obj_id = assignments[i].second;
27     tracks[track_id]->UpdateWithObject(&(*new_objects)[obj_id], time_diff);
28 }
29 }
```

从上述的代码可以看到，更新过程有 `ObjectTrack::UpdateWithObject` 和

`ObjectTrack::UpdateWithoutObject` 两个函数完成，这两个函数间接调用kalman滤波器完成滤波更新，接下去我们简单地分析 `ObjectTrack::UpdateWithObject` 函数的流程。

```
1 /// file in apollo/modules/perception/obstacle/lidar/tracker/hm_tracker/object_track.cc
2 void ObjectTrack::UpdateWithObject(TrackedObjectPtr* new_object, const double& time_diff)
3
4 // A. update object track
5 // A.1 update filter
6 filter_->UpdateWithObject((*new_object), current_object_, time_diff);
7 filter_->GetState(&belief_anchor_point_, &belief_velocity_);
8 filter_->GetOnlineCovariance(&belief_velocity_uncertainty_);
9
10 (*new_object)->anchor_point = belief_anchor_point_;
11 (*new_object)->velocity = belief_velocity_;
```

```

12     (*new_object)->velocity_uncertainty = belief_velocity_uncertainty_;
13
14     belief_velocity_acceleration_ = ((*new_object)->velocity - current_object->velocity) /
15     // A.2 update track info
16     ...
17
18     // B. smooth object track
19     // B.1 smooth velocity
20     SmoothTrackVelocity((*new_object), time_diff);
21     // B.2 smooth orientation
22     SmoothTrackOrientation();
23 }

```

从代码中也可以间接看出更新的过程A.1和A.2是更新KalmanFilter和TrackedObject状态信息，B是更新ObjectTrack状态，这里必须按顺序来更新！

01 KalmanFilter滤波器状态更新

主要由 KalmanFilter::UpdateWithObject 函数完成，计算过成分下面几步：

- **Step1. 计算更新评分** ComputeUpdateQuality(new_object, old_object)

这个过程主要是计算更新力度，因为每个Object和对应的跟踪目标TrackedObject之间有一个关联系数 association_score，这个系数衡量两个目标之间的相似度，所以这里需要增加对目标的更新力度参数。

计算关联力度： $\text{update_quality_according_association_score} = 1 - \text{association_score} / \text{s_association_score_maximum_}$ 。默认s_association_score_maximum_ = 1，关联越大(相似度越大)，更新力度越大

计算点云变化力度： $\text{update_quality_according_point_num_change} = 1 - |n1 - n2| / \max(n1, n2)$ 。点云变化越小，更新力度越大

最终取两个值的较小值最为最终的更新力度。

- **Step2. 计算当前时刻的速度** `measured_velocity` **和加速度** `measured_acceleration` (这两个变量相当于卡尔曼滤波中的观测变量 Z_t)

首先计算重心速度: `measured_anchor_point_velocity = [NewObject_barycenter(x,y,z) - OldObject_barycenter(x,y,z)] / timediff`。timediff是两次计算的时间差, 很简单地计算方式

其次计算标定框(中心)速度: `measured_bbox_center_velocity = [NewObject_center(x,y,z) - OldObject_center(x,y,z)] / timediff`。这里的中心区别于上面的重心, 重心是所有点云的平均值; 重心是MinBox的中心值。还有一个需要注意的是, 如果求出来的中心速度方向和重心方向相反, 这时候有干扰, 中心速度暂时置为0。

然后计算标定框角点速度:

A. 根据NewObject的点云计算bbox(这不是MinBox), 并求出中心center, 然后根据反向求出4个点。如果NewObject方向是dir, 那么首先对dir进行归一化得到 $dir_normal = dir / |dir|^2 = (nx, ny, 0)$, 然后求他的正交方向 $dir_ortho = (-ny, nx, 0)$, 如果中心点坐标center, 那么左上角的坐标就是: $center + dir * size[0] * 0.5 + ortho_dir * size[1] * 0.5$ 。根据这个公式可以计算出其他三个点的坐标。

B. 计算标定框bbox四个角点的速度: `bbox_corner_velocity = ((new_bbox_corner - old_bbox_corner) / time_diff)` 公式与上面的重心、中心计算方式一样。

C. 计算4个角点的在主方向上的速度, 去最小的点最为标定框角点速度。只需要将B中的 `bbox_corner_velocity` 投影到主方向即可。

最后在重心速度、重心速度、bbox角速度中选择速度增益最小的最后最终物体的增益。增益=当前速度-上时刻最优速度

加速度 `measured_acceleration` 计算比较简单, 采用最近3次的速度 $(v_1, t_1), (v_2, t_2), (v_3, t_3)$, 然后加速度 $a = (v_3 - v_1) / (t_2 + t_3)$ 。注意 (v_2, t_2) 意思是某一时刻最优估计速度为 v_2 , 且距离上次的时间差为 t_2 , 所以三次测量的时间差为 $t_2 + t_3$ 。速冻变化为 $v_3 - v_1$ 。

- **Step3. 估算最优的速度与加速度**(卡尔曼滤波Update步骤)

首先，计算卡尔曼增益 $H_t = P_{\{t,t-1\}}C_t^T[C_tP_{\{t,t-1\}}C_t^T + R]^{-1}$ ，在apollo代码中计算代码如下：

```
1 // Compute kalman gain
2 Eigen::Matrix3d mat_c = Eigen::Matrix3d::Identity(); // C_t
3 Eigen::Matrix3d mat_q = s_measurement_noise_ * Eigen::Matrix3d::Identity(); // R_t
4 Eigen::Matrix3d mat_k = velocity_covariance_ * mat_c.transpose() * // H_t
5     (mat_c * velocity_covariance_ * mat_c.transpose() + mat_q).inverse();
```

从上面可知，代码和我们给出的结果是一致的。

然后，由当前时刻的估算速度 $X_{\{t,t-1\}}$ 、观测变量 Z_t 以及卡尔曼增益 H_t ，得到当前时刻的最优速度估计 $X_t = X_{\{t,t-1\}} + H_t[Z_t - C_tX_{\{t,t-1\}}]$ ，在apollo代码中计算了速度增益，也就是 $X_t - X_{\{t,t-1\}}$ ：

```
1 // Compute posterior belief
2 Eigen::Vector3d measured_velocity_d = measured_velocity.cast<double>();
3 Eigen::Vector3d priori_velocity = belief_velocity_ + belief_acceleration_gain_ * time_diff;
4 Eigen::Vector3d velocity_gain = mat_k * (measured_velocity_d - mat_c * priori_velocity);
```

然后对速度增益进行平滑并且保存当前t时刻最优速度以及最优加速度。

```
1 // Breakdown
2 ComputeBreakdownThreshold();
3 if (velocity_gain.norm() > breakdown_threshold_) {
4     velocity_gain.normalize();
5     velocity_gain *= breakdown_threshold_;
6 }
7
8 belief_anchor_point_ = measured_anchor_point_d;
9 belief_velocity_ = priori_velocity + velocity_gain; // Xv_t = Xv_{t,t-1} + Gain
10 belief_acceleration_gain_ = velocity_gain / time_diff; // Acc_t = Xv_t / timediff
```

最后就是速度整流并且修正估计协方差矩阵 $P_{t,t-1}$ ，得到当前时刻最优协方差矩阵 $P_t = [I - H_t C_t] P_{t,t-1}$ ，在这个应用中 $C_t \equiv 1$ 。

```

1  // Adaptive
2  if (s_use_adaptive_) {
3      belief_velocity_ -= belief_acceleration_gain_ * time_diff;
4      belief_acceleration_gain_ *= update_quality_;
5      belief_velocity_ += belief_acceleration_gain_ * time_diff;
6  }
7
8  // Compute posterior covariance
9  velocity_covariance_ = (Eigen::Matrix3d::Identity() - mat_k * mat_c) * velocity_covariance_

```

加速度更新与上述速度更新方法一致。

• Step4. 缓存更新信息

将观测变量 `measured_velocity` 和时间差 `time_diff` 缓存，同时使用观测速度

`measured_velocity` 对实时协方差矩阵 `online_velocity_covariance_` 进行更新。

02 TrackedObject状态更新

设置TrackedObject的重心，速度(滤波器得到的t时刻最优速度 `belief_velocity_`)，加速度([最优速度 `belief_velocity_` - OldObject的最优速度]/时间差)，更新跟踪时长(++age)，目标可见次数(++total_visible_count_)，跟踪总时长(`period_ += time_diff`)，连续不可见时长置0(`consecutive_invisible_count_ = 0`)。

03 ObjectTrack状态更新 (速度与方向平滑滤波)

由原始KalmanFilter中的各个状态信息，加入实际应用中的限制进行滤波得到ObjectTrack的状态信息，这些信息才是真实被使用的。

对跟踪物体的速度整流过程如下(`ObjectTrack::SmoothTrackVelocity`):

- 如果物体的加速度增益查过一定阈值(`s_acceleration_noise_maximum_`, 默认为5), 那么当前速度保持上时刻的速度
- 否则, 对小速度物体进行修建。计算物体的速度, 默认 `s_speed_noise_maximum_ = 0.4`
 - 如果 `velocity_is_noise = speed < (s_speed_noise_maximum_ / 2)`, 则判定为噪声
 - 如果 `velocity_is_small = speed < (s_speed_noise_maximum_ / 1)`, 则判定为小速度
 - 计算物体两个时刻角度的变化, `fabs(velocity_angle_change) > M_PI / 4.0`, 如果 `cos`值小于 $\pi/4$ (45度), 说明物体没有角度变化 最终判断: `if(velocity_is_noise || (velocity_is_small && is_velocity_angle_change))` 如果速度是噪声, 或者速度很小方向不变, 那么认定车是静止的。对于车是静止的, 真实速度和加速度都设置为0

这里需要注意:

```
// NEED TO NOTICE: claping small velocity may not reasonable when the true velocity of
target object is really small. e.g. a moving out vehicle in a parking lot. Thus, instead of
clapping all the small velocity, we clap those whose history trajectory or performance is
close to a static one.
```

按照官方代码提醒, 其实这样对小速度物体进行修剪时不太合理, 因为某些情况下物体速度确实很小, 但是他确实是在运动。E.g. 汽车倒车的时候, 速度小, 但是不能被忽略。所以最好的方法是根据历史的轨迹(重心, `anchor_point`)来判断物体在小速度的情况下是否是运动的。

对跟踪物体的方向整流过程如下(`ObjectTrack::SmoothTrackOrientation`), 如果物体运动比较明显 `velocity_is_obvious = current_speed > (s_speed_noise_maximum_ * 2)` (大于0.4m/s), 那么当前运动方向为物体速度的方向; 否则设定为车道线方向。

就这样, 经过三步骤, 跟踪配对的物体(`Object-TrackedObject`存在)完成了状态信息的更新, 主要包括当前时刻最优速度、方向、加速等信息。

如果跟踪物体中没有找到对应的Object与之匹配, 就需要使用 `UpdateUnassignedTracks` 函数来更新跟踪物体的信息。从上面我们可以看到, 匹配成功的可以用Object的属性来计算观测变量, 间接估算出t时刻的最优状态。但是未匹配的TrackedObject无法因为找不到Object, 所以无法了解当前时刻真实能测量到的位置、速度与加速度信息, 因此只能依赖自身上时刻的最优状态来推算出当

前时候的状态信息(注意, 这个推算出来的不是最优状态)。

对未找到Object的跟踪物体, 更新过程如下:

1. 使用2.4.2节中的估算数据来预测当前时刻的状态;

```
1 Eigen::Vector3f predicted_shift = predict_state.tail(3) * time_diff;
2 new_obj->anchor_point = current_object->anchor_point + predicted_shift;
3 new_obj->barycenter = current_object->barycenter + predicted_shift;
4 new_obj->center = current_object->center + predicted_shift;
```

2. 其中 `predicted_shift` 是利用卡尔曼滤波Predict阶段预测到的当前时刻物体重心位置与速度状态 $Xp_{t,t-1}$ 和 $Xv_{t,t-1}$, 乘以时间差就可以得到这个时间差内的位移, 去更新中心, 重心;
3. 上时刻TrackedObject里面的原始点云和多边形坐标也加上这个位移, 完成更新;
4. 更新KalmanFilter里面的原始状态信息, `KalmanFilter::UpdateWithoutObject`, KalmanFilter只更新重心坐标, 不需要更新速度和加速度(因为无法更新, 缺少观测数据Z, 不能使用卡尔曼滤波器的Update过程去更新);
5. 更新TrackedObject状态信息, 更新跟踪时长(++age), 跟踪总时长(`period_ += time_diff`), 更新连续不可见时长置(++consecutive_invisible_count_=0);
6. 更新历史缓存。

更新完匹配成功和不成功的跟踪物体以后, 下一步就是从跟踪列表中删掉丢失的跟踪物体。遍历整个跟踪列表:

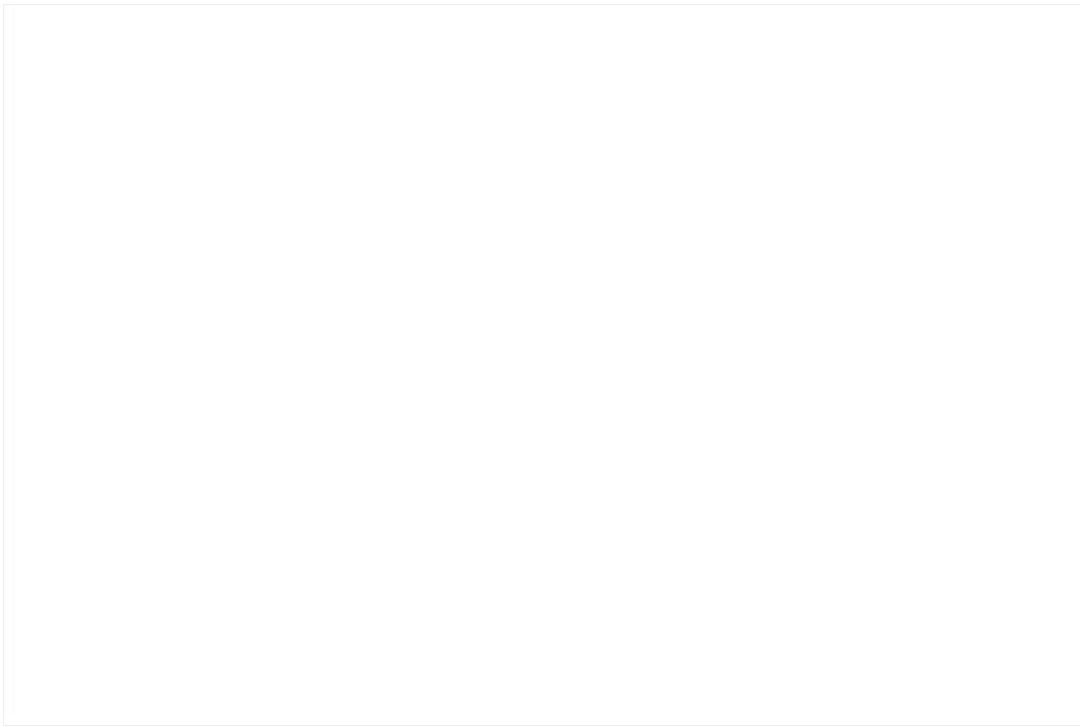
1. 可见次数/跟踪时长小于阈值(`s_track_visible_ratio_minimum_`, 默认0.6), 删除;
2. 连续不可见次数大于阈值(`s_track_consecutive_invisible_maximum_`, 默认1), 删除。

如果Object没有找到对应的TrackedObject与之匹配, 那么就创建新的跟踪目标, 并且加入跟踪队

列。

最终对HM物体跟踪做一个总结与梳理，物体跟踪主要是对上述CNN分割与MinBox边框构建产生的Object对一个跟踪与匹配，主要流程是：

- **Step1.** 预处理，Object里面的中心，重心，点云，多边形凸包从lidar坐标系转换成局部ENU坐标系；
- **Step2.** 将坐标转换完成Object封装成TrackedObject，方便后续加入跟踪列表；
- **Step3.** 使用卡尔曼滤波Predict阶段，对正处于跟踪列表中的跟踪物体进行当前时刻重心位置、速度的预测；
- **Step4.** 使用当前检测到的Object(封装成了TrackedObject)，去和跟踪列表中的物体进行匹配：
 - 计算Object与TrackedObject的关联矩阵
 - 重心位置坐标距离差异评分
 - 物体方向差异评分
 - 标定框尺寸差异评分
 - 点云数量差异评分
 - 外观特征差异评分
 - 根据关联矩阵，配合阈值，划分子图
 - 对于每个子图使用匈牙利匹配算法(Hungarian Match)进行匹配，得到<Object,TrackedObject>、<Object,None>、<None,TrackedObject>
 - <Object,TrackedObject>成功匹配(有Object计算观测数据)，更新KalmanFilter状态(Update阶段)，更新TrackedObject状态，更新ObjectTrack状态
 - <None,TrackedObject>没有对应的Object(无法得到观测数据，无法使用卡尔曼滤波估算最优速度)，更新部分KalmanFilter状态(仅重心)，更新TrackedObject状态，更新ObjectTrack状态
 - <Object,None>创建新的TrackedObject，加入跟踪列表
 - 删除丢失的跟踪目标
 - 可见次数/跟踪时长过小
 - 连续不可见次数过大



自Apollo平台开放以来，我们收到了大量开发者的咨询和反馈，越来越多开发者基于Apollo擦出了更多的火花，并愿意将自己的成果贡献出来，这充分体现了Apollo『**贡献越多，获得越多**』的开源精神。为此我们开设了『**开发者说**』板块，希望开发者们能够踊跃投稿，更好地为广大自动驾驶开发者营造一个共享交流的平台！

* 以上内容为开发者原创，不代表百度官方言论。

已获开发者授权，原文地址请戳阅读原文。

您可能感兴趣的其它内容

More



[阅读原文](#)