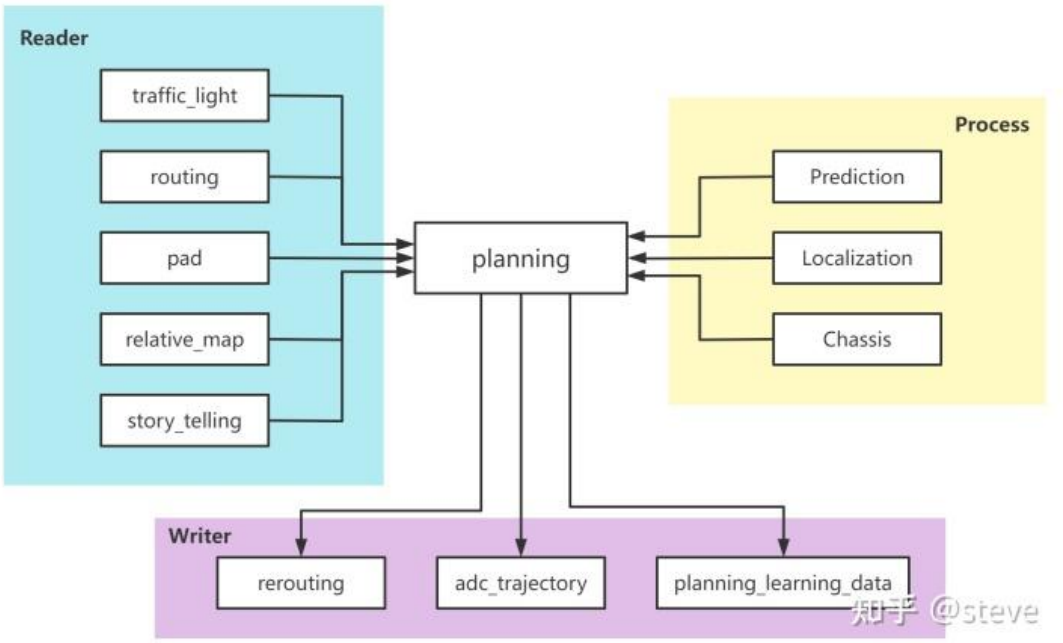
Apollo planning 模块梳理

1.简介

实时规划出车辆的行驶轨迹，要求：安全、能够避障、遵守交通规则、驾乘体验好。

2.模块架构



2.1 数据的输入和输出

2.1.1 输入

Routing：定义了概念性问题“我想去哪儿”，给出全局路径。消息定义在 routing.proto 文件中。RoutingResponse 包含了 RoadSegment，RoadSegment 指明了车辆到达目的地应该遵循的路线。

感知和预测：关于“我周围有什么”的消息定义在 perception_obstacles.proto 和 traffic_light_detection.proto 中。perception_obstacles.proto 定义了表示车辆周围的障碍物的数据，车辆周围障碍物的数据由感知模块提供。traffic_light_detection 定义了信号灯状态的数据。除了已被感知的障碍物外，动态障碍物的路径预测对 Planning 模块也是非常重要的数据，因此 prediction.proto 封装了 perception_obstacle 消息来表示预测路径。

车辆状态和定位：另外一个重要的概念性问题是“我在哪”。关于该问题的数据通过高清地图和定位模块获得。定位信息和车辆车架信息被封装在 VehicleState 消息中，该消息定义在 vehicle_state.proto

高清地图：实际上还有高精度地图信息，不在参数中传入，而是在函数中直接读取的。

2.1.2 输出

planning.proto 介绍了 ADCTrajectory 包含的内容（总时间、总长度、路径点等）（trajectory point 包含 path_point、v、a、Kappa 等）

// next id: 24

```
message ADCTrajectory {
  optional apollo.common.Header header = 1;

  optional double total_path_length = 2;  // in meters
  optional double total_path_time = 3;     // in seconds

  // path data + speed data
  repeated apollo.common.TrajectoryPoint trajectory_point = 12;

  optional EStop estop = 6;

  // path point without speed info
  repeated apollo.common.PathPoint path_point = 13;

  // is_replan == true mean replan triggered
  optional bool is_replan = 9 [default = false];
  optional string replan_reason = 22;

  // Specify trajectory gear
  optional apollo.canbus.Chassis.GearPosition gear = 10;

  optional apollo.planning.DecisionResult decision = 14;

  optional LatencyStats latency_stats = 15;

  // the routing used for current planning result
  optional apollo.common.Header routing_header = 16;
  optional apollo.planning_internal.Debug debug = 8;

  enum RightOfWayStatus {
    UNPROTECTED = 0;
    PROTECTED = 1;
  }
  optional RightOfWayStatus right_of_way_status = 17;

  // lane id along current reference line
  repeated apollo.hdmap.Id lane_id = 18;

  // set the engage advice for based on current planning result.
  optional apollo.common.EngageAdvice engage_advice = 19;

  // the region where planning cares most
  message CriticalRegion {
    repeated apollo.common.Polygon region = 1;
```

```
  }

  // critical region will be empty when planning is NOT sure which region is
  // critical
  // critical regions may or may not overlap
  optional CriticalRegion critical_region = 20;

  enum TrajectoryType {
    UNKNOWN = 0;
    NORMAL = 1;
    PATH_FALLBACK = 2;
    SPEED_FALLBACK = 3;
    PATH_REUSED = 4;
  }
  optional TrajectoryType trajectory_type = 21 [default = UNKNOWN];

  // lane id along target reference line
  repeated apollo.hdmap.Id target_lane_id = 23;

  // output related to RSS
  optional RSSInfo rss_info = 100;
  // set the steering, throttle, brake zero when complete the parking
  optional bool complete_parking = 25;
}
```

除了路径信息，Planning 模块还输出了多种注释信息。主要的注释数据包括：Estop、DecisionResult、调试信息。

2.2 重要的数据结构

2.2.1LocalView

LocalView 这个结构体中包含了 planning 的所有上游输入数据：包含了预测障碍物及其轨迹、底盘信息、定位信息、交通灯信息、导航信息、相对地图等，一帧规划中所会用到的所有外部输入都被包含进了该结构体中, 其会在 component 的 Process()函数的一开始便进行赋值。

```
/**
 * @struct local_view
 * @brief LocalView contains all necessary data as planning input
 */
struct LocalView {
  std::shared_ptr<prediction::PredictionObstacles> prediction_obstacles;
  std::shared_ptr<canbus::Chassis> chassis;
  std::shared_ptr<localization::LocalizationEstimate> localization_estimate;
  std::shared_ptr<perception::TrafficLightDetection> traffic_light;
  std::shared_ptr<routing::RoutingResponse> routing;
```

```
  std::shared_ptr<relative_map::MapMsg> relative_map;
  std::shared_ptr<PadMessage> pad_msg;
  std::shared_ptr<storytelling::Stories> stories;
};
```

2.2.2ReferenceLineInfo

ReferenceLineInfo 类是 planning 当中最重要的类之一，其包含了 ReferenceLine 类，为车辆行驶的参考线；

还包含了 blocking_obstacles_，即在该条参考线上阻塞车道的障碍物；还有优化出的 path_data、speed_data 以及最终融合后的轨迹 discretized_trajectory。在以参考线为主的规划算法中，该类包含了所有算法中用到的信息及结果。

ReferenceLineInfo 对 ReferenceLine 类进行了封装，为 Planning 模块提供了平滑的指令执行序列。

```
private:
  static std::unordered_map<std::string, bool> junction_right_of_way_map_;
  const common::VehicleState vehicle_state_;
  const common::TrajectoryPoint adc_planning_point_;
  ReferenceLine reference_line_;

  /**
   * @brief this is the number that measures the goodness of this reference
   * line. The lower the better.
   */
  double cost_ = 0.0;
  bool is_drivable_ = true;
  PathDecision path_decision_;
  Obstacle* blocking_obstacle_;
  std::vector<PathBoundary> candidate_path_boundaries_;
  std::vector<PathData> candidate_path_data_;
  PathData path_data_;
  PathData fallback_path_data_;
  SpeedData speed_data_;
  DiscretizedTrajectory discretized_trajectory_;
  RSSInfo rss_info_;

  /**
   * @brief SL boundary of stitching point (starting point of plan trajectory)
   * relative to the reference line
   */
  SLBoundary adc_sl_boundary_;
```

2.2.3ReferenceLine

ReferenceLine 类即为规划的参考线，其中包含了地图给出的原始中心线 map_path，以及后续优化的基准参考点，一系列的 reference_points。

```
private:
  struct SpeedLimit {
    double start_s = 0.0;
    double end_s = 0.0;
    double speed_limit = 0.0;   // unit m/s
    SpeedLimit() = default;
    SpeedLimit(double _start_s, double _end_s, double _speed_limit)
        : start_s(_start_s), end_s(_end_s), speed_limit(_speed_limit) {}
  };
  /**
   * This speed limit overrides the lane speed limit
   **/
  std::vector<SpeedLimit> speed_limit_;
  std::vector<ReferencePoint> reference_points_;
  hdmap::Path map_path_;
  uint32_t priority_ = 0;
```

2.2.4Frame

Frame 类包含了 planning 一次循环中所用到的所有信息，可以说这个数据结构贯穿了 planning 的主逻辑，上述讲到的所有结构体也都被包含在了这个类中，可以看下里面包含的变量。有 LocalView、地图信息 hd_map、规划初始点、车辆状态信息、ReferenceLineInfo 信息、障碍物信息、交通灯信息等等，它也是后续 planner 基类的入参，所有的输入信息、输出信息都包含其中。

```
private:
  static DrivingAction pad_msg_driving_action_;
  uint32_t sequence_num_ = 0;
  LocalView local_view_;
  const hdmap::HDMap *hdmap_ = nullptr;
  common::TrajectoryPoint planning_start_point_;
  common::VehicleState vehicle_state_;
  std::list<ReferenceLineInfo> reference_line_info_;
  bool is_near_destination_ = false;

  /**
   * the reference line info that the vehicle finally choose to drive on
   **/
  const ReferenceLineInfo *drive_reference_line_info_ = nullptr;
  ThreadSafeIndexedObstacles obstacles_;
  std::unordered_map<std::string, const perception::TrafficLight *>
      traffic_lights_;
  // current frame published trajectory
  ADCTrajectory current_frame_planned_trajectory_;
  // current frame path for future possible speed fallback
  DiscretizedPath current_frame_planned_path_;
```

```
const ReferenceLineProvider *reference_line_provider_ = nullptr;
OpenSpaceInfo open_space_info_;
std::vector<routing::LaneWaypoint> future_route_waypoints_;
common::monitor::MonitorLogBuffer monitor_logger_buffer_;
```

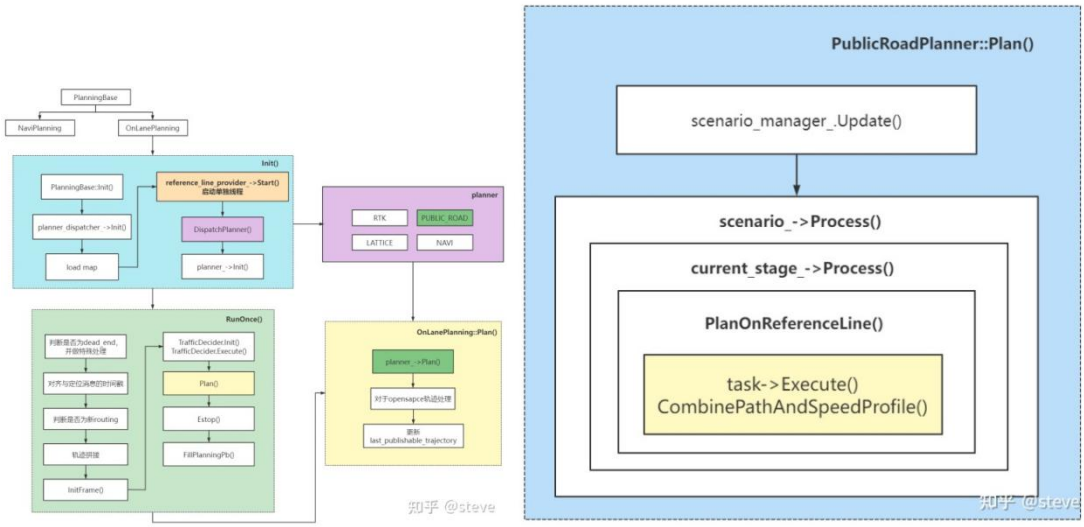
2.2.5HD-Map

HD-Map 在 Planning 模块内作为封装了多个数据的库使用，提供不同特点的地图数据查询需求。

2.2.6Planner

Planner 执行具体的 Planning 任务。

2.3 运行机制



主框架分为两个线程，子线程 ReferenceLineProvider 以 20HZ 的频率运行，用于计算 planning 中最重要的数据结构 reference_line；主线程上还是基于场景划分的思路，多数场景下还是采用基于 ReferenceLine 的规划算法，对于泊车相关场景，则利用 open space 算法。目前 Apollo 的场景划分为了 16 种，在 proto 文件中可以查看到。在 Apollo 7.0 中，新增了 deadend_turnaround 场景，用于无人车遇到断头路时，采用 openspace 的方法进行调头的轨迹规划。

2.3.1 模块入口

模块的入口是 PlanningComponent，在 Cyber 中注册模块，订阅和发布消息，并且注册对应的 Planning 类。

2.3.2 模块初始化

PlanningBase::Init()

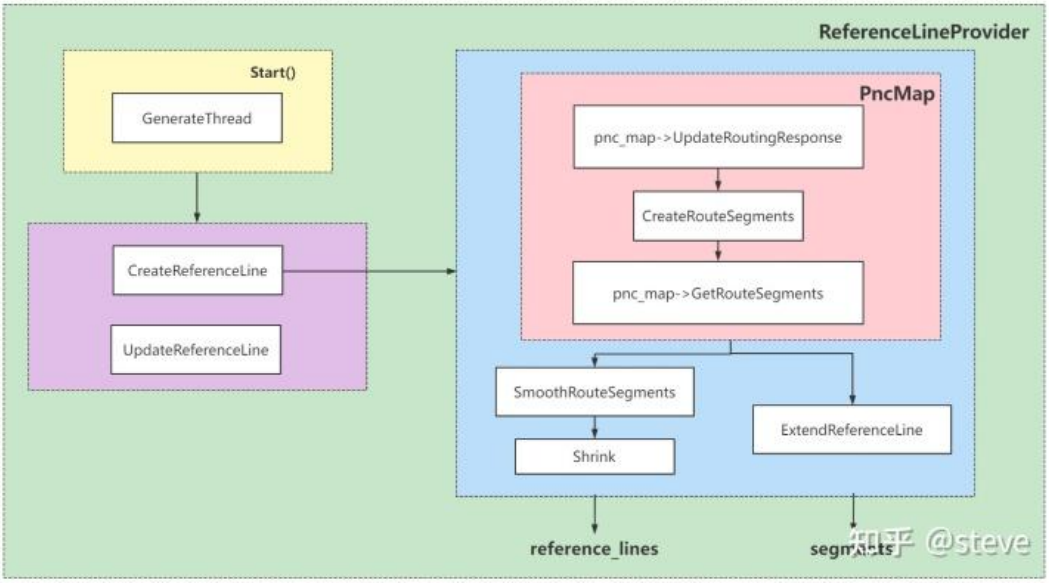
- OnLanePlanning（车道规划，可用于城区及高速公路各种复杂道路）
- NaviPlanning（导航规划，主要用于高速公路）
- OpenSpacePlanning （自主泊车和狭窄路段的掉头）

plannner_dispatcher_->Init()

- PublicRoadPlanner（默认规划器）
- LatticePlanner
- NaviPlanner（主要用于高速公路场景）
- RTKPlanner（循迹算法，一般不用）

2.3.3ReferenceLineProvider

具体进入 planner 的 Plan()函数之前，其中最重要的一个步骤就是子进程中对于 reference_line 的构造，可以说后续 planning 的算法逻辑全都是基于这些 reference_line 来完成的。



pnc_map 即规划控制地图，在 planning 中属于相对独立的一块功能，
其主要目的在于将 routing 给到的原始地图车道信息转换成规划可以用的 ReferenceLine 信息。

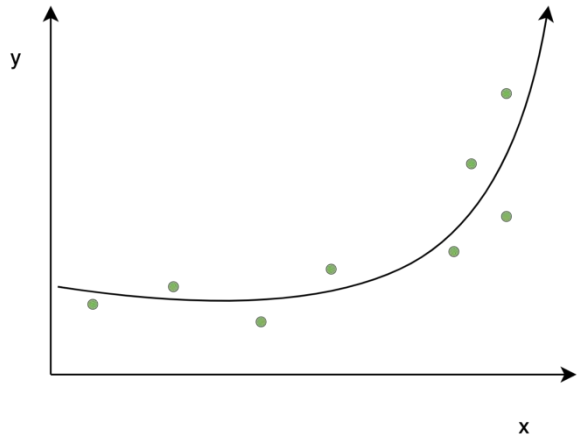
简单来说，实现 pnc_map 的功能分为两步

1.处理 routing 结果，将 routing 数据处理并存储在 map 相关的数据结构中，供后续使用。

2.根据 routing 结果及当前车辆位置，计算可行驶的 passage 信息，并转换成 RouteSegments 结构。

2.3.4 参考线平滑

拿到 pnc_map 的结果之后，下一步就要对原始的地图中心线信息进行平滑。这里分为两块逻辑，如果当前 routing 为新 routing 或者未启用 reference_line_stitching 功能的话，就走流程图中的左边分支，进行 SmoothRouteSegments 和 Shrink 操作；若不是，则走流程图中的右侧分支，进行 ExtendReferenceLine 操作，其中也包含了对参考线的平滑.



目标函数设计： $cost = cost_1 + cost_2 + cost_3$

其中： $cost_1$ **为平滑度代价，** $cost_2$ **为长度代价，** $cost_3$ **为相对原始点偏移量代价**

$$cost_1 = \sum_{i=1}^{n-1}(x_{i-1} + x_{i+1} - 2 \times x_i)^2 + (y_{i-1} + y_{i+1} - 2 \times y_i)^2$$

$$cost_2 = \sum_{i=0}^{n-1}(x_i - x_{i+1})^2 + (y_i - y_{i+1})^2$$

$$cost_3 = \sum_{i=0}^{n-1}(x_i - x_{i-ref})^2 + (y_i - y_{i-ref})^2$$

约束条件：

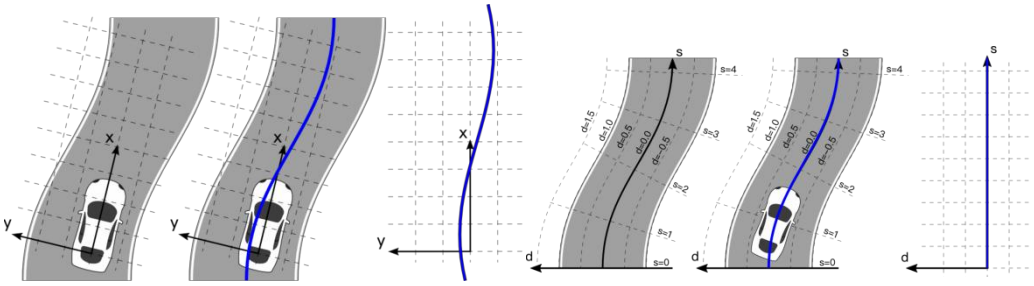
$$\begin{cases} x_{i-ref} - x_l \le x_i \le x_{i-ref} + x_u & i = 0, 1, 2, \cdots, n \\ y_{i-ref} - y_l \le x_i \le y_{i-ref} + y_u & i = 0, 1, 2, \cdots, n \\ 0 \le stack_i \le +\infty & i = 1, 2, 3, \cdots, n-1 \\ (x_{i-1} + x_{i+1} - 2 \times x_i)^2 + (y_{i-1} + y_{i+1} - 2 \times y_i)^2 - stack_i \le (\Delta s^2 \times cur_{cstr})^2 & i = 1, 2, 3, \cdots, n-1 \end{cases}$$

Apollo 里的高精地图里车道线中心点可能稀疏，可能密集，质量是不确定的，如果以这样的点生成路径，路径也会崎岖不平，不连续。所以首先就需要对车道中心线的点做平滑处理。

Apollo 平滑默认用的是散点平滑算法，每个点作为优化变量，会允许优化点在参考点一定范围内变化，设计目标函数，包括平滑度，就是三个点组成的夹角越大越平滑，两点之间的长度越小越好（点分布的越平均越好），还有相对原始点的偏移量，不希望相对原始点偏移太多。

另外还有一些约束，比如优化点在参考点多远范围内变化，以及曲率的约束。关于曲率的计算，Apollo 的主要思想是三点构成一个圆来求解半径。stack 是松弛变量。

2.3.4frenet 坐标系



平滑后的点不会直接用，将平滑后的参考线点的切线方向作为 s 轴，法线方向作为 d 轴，建立坐标系，就是把这条曲线拉直，这样其实更方便路径规划的计算。这个就作为后面整个规划过程是用的坐标系，相应地，障碍物的位置也会投影到这个坐标系上来。

2.3.5Plan

场景注册、场景转换、场景运行

Scenario

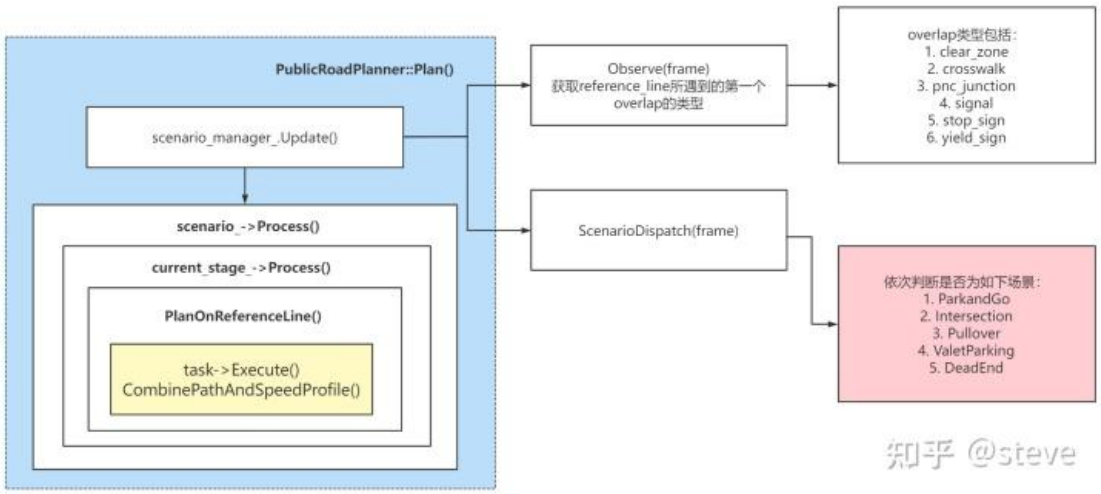
stage

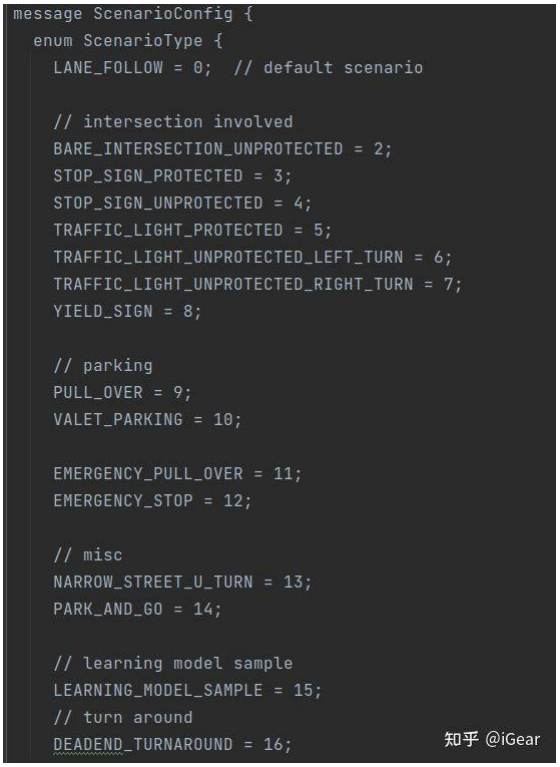
Task

3.场景 scenario

3.1 场景管理

Apollo 中的规划里的一个重要思想就是基于场景划分来调用不同的 task 处理，而其中如何进行场景分配便是实现该思想的核心。从上面的配置参数可以看到目前 Apollo 设定了 16 种场景，而场景下的具体切换逻辑也比较复杂，scenario_manager 的具体流程逻辑如下图所示：





在上图红框中的场景判断中，只画了第一步的场景判断，红框内的 5 种场景为大类，剩余的场景在这 5 大类中再细分做出判断。另外需要注意的是，红框内的 5 种场景是有优先级顺序的，即如果判断为某种场景后，后续的场景也就不再判断。下面从这 5 种场景出发，介绍一下 Apollo 中的场景判断条件，以及每个大类场景下包含哪些细分的场景小类。

场景管理器主要就是做了一个有限状态机，在每一帧 planning 运行过程中，根据参考线和车辆状态来判断在哪个场景，以及是否需要切换场景。

默认场景是 lane follow，当每个场景完成后最后都会回到 lane follow 的场景。每个场景都可以单独配置参数，单独配置算法，这样也就实现了场景和算法的解耦。

3.2 场景确认

3.2.1LANE_FOLLOW

默认驾驶场景，包括本车道保持、变道、基本转弯

3.2.2PARK_AND_GO

停车和启动。该场景的判断条件为车辆是否静止，并且距离终点 10m 以上，并且当前车辆已经 off_lane 或者不在城市道路上，在该场景下采用的是 open_space 相关的算法。个人感觉该场景在驶离目标车道并正常规划失败导致的停车时会触发，利用 open_space 方法使其重新回到正常道路上，因此也是场景判断中首先需要 check 的。

3.2.3Intersection

STOP_SIGN 停止标志

TRAFFIC_LIGHT 交通灯

YIELD_SIGN 让路标志

BARE_INTERSECTION 裸露交叉路口

3.2.4PullOver

靠边停车，需要满足以下条件才可切换到该场景：不在 change_line 的时候，即 reference_line 只有一条当前位置距离终点在一定范围内并且满足 pullover 可以执行的最短

距离地图中能够找到 pullover 的位置终点的位置不在交叉路口附近能查找到最右边车道的 lane_type，并且该车道允许 pullover 只有从 lane_follow 场景下才能切换到 pullover 大体逻辑如上述所示，具体的参数设置可以查看代码。

3.2.5VALET_PARKING

代客泊车，判断逻辑如下：

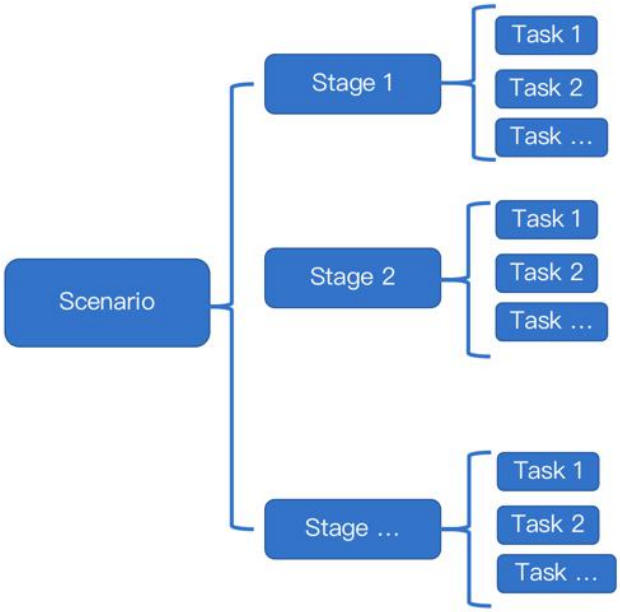
- 从 routing 中得到 target_parking_spot_id
- 从地图中搜索是否存在 path 能够抵达该 parking_spot
- 查询当前位置至 parking_spot 的距离，满足条件即可切换至该场景

3.2.6DeadEnd

Apollo 7.0 中新增的断头路场景，增加了"三点掉头"功能，增加了驶入驶出的能力，扩展了城市路网运营边界。"三点掉头"功能基于 open space planner 框架，包含以下几个部分：断头路场景转换、开放空间 ROI 构建、掉头轨迹规划。

4.stage

在 planning 模块确定 scenarios 之后就会进入 stage 阶段，stage 是 plannin 衔接 scenarios 与后具体执行器 task 的中间阶段。



```
scenario_type: VALET_PARKING
valet_parking_config: {
  parking_spot_range_to_start: 20.0
  max_valid_stop_distance: 1.0
}

stage_type: VALET_PARKING_APPROACHING_PARKING_SPOT
stage_type: VALET_PARKING_PARKING

stage_config: {
  stage_type: VALET_PARKING_APPROACHING_PARKING_SPOT
  enabled: true
  task_type: PARKING_SPACE_DECIDER
  task_type: OPEN_SPACE_PRE_STOP_DECIDER
  task_type: PATH_LANE_BORROW_DECIDER
  task_type: PATH_BOUNDS_DECIDER
  task_type: PIECEWISE_JERK_PATH_OPTIMIZER
  task_type: PATH_ASSESSMENT_DECIDER
  task_type: PATH_DECIDER
  task_type: RULE_BASED_STOP_DECIDER
  task_type: ST_BOUNDS_DECIDER
  task_type: SPEED_BOUNDS_PRIORI_DECIDER
  task_type: SPEED_HEURISTIC_OPTIMIZER
  task_type: SPEED_DECIDER
  task_type: SPEED_BOUNDS_FINAL_DECIDER
  task_type: PIECEWISE_JERK_SPEED_OPTIMIZER
  task_config: {
    task_type: PARKING_SPACE_DECIDER
  }
  task_config: {
    task_type: OPEN_SPACE_PRE_STOP_DECIDER
    open_space_pre_stop_decider_config {
      stop_type: PARKING
```

```
stage_config: {
  stage_type: VALET_PARKING_PARKING
  enabled: true
  task_type: PARKING_SPACE_DECIDER
  task_type: OPEN_SPACE_ROI_DECIDER
  task_type: OPEN_SPACE_TRAJECTORY_PROVIDER
  task_type: OPEN_SPACE_TRAJECTORY_PARTITION
  task_type: OPEN_SPACE_FALLBACK_DECIDER
  task_config: {
    task_type: PARKING_SPACE_DECIDER
  }
  task_config: {
    task_type: OPEN_SPACE_ROI_DECIDER
    open_space_roi_decider_config {
      roi_type: PARKING
    }
  }
  task_config: {
    task_type: OPEN_SPACE_TRAJECTORY_PROVIDER
  }
  task_config: {
    task_type: OPEN_SPACE_TRAJECTORY_PARTITION
  }
  task_config: {
    task_type: OPEN_SPACE_FALLBACK_DECIDER
  }
}
```

场景下面还分出了多个 stage，每个 stage 一般按照默认的顺序工作，stage 下面还会分出 task，就是 planning 具体执行的任务，比如这是泊车场景的一个配置文件，泊车场景分成了两个 stage，一个是接近停车位的过程，一个是泊车的过程。Planning 会首先判断是否要进入泊车场景，如果 routing 里有泊车的指令，或者主车距离泊车位到达一定距离，则要进入泊车场景。

进入泊车场景后，首先会进入第一个 stage 接近停车位的过程。可以看到接近停车位过程第一个 task 是 pre stop decider，就是确定要在车位前哪个位置停下来，接下来是路径规划和速度规划的 task。然后是开始泊车，泊车也分为 4 个 task，第一个 ROIdecider 是确定泊车的边界，因为是开放空间规划，需要知道哪里能走，第二个是轨迹生成，就是生成一条从起点到终点无碰撞的轨迹，第三个是轨迹分割，就是分成前进和后退的轨迹，根据自车当前位置来选择走哪条轨迹，最后一个是 fallback，如果轨迹有碰撞或者失败生成一条降级的刹车轨迹。

5.task

在 apollo/modules/planning/tasks 文件夹中，Task 分为 3 类：deciders(决策)，optimizers（规划），learning_model。

```
scenario_type: LANE_FOLLOW
stage_type: LANE_FOLLOW_DEFAULT_STAGE
stage_config: {
  stage_type: LANE_FOLLOW_DEFAULT_STAGE
  task_type: LANE_CHANGE_DECIDER
  task_type: PATH_LANE_BORROW_DECIDER
  task_type: PATH_BOUNDS_DECIDER
  task_type: PIECEWISE_JERK_PATH_OPTIMIZER
  task_type: PATH_ASSESSMENT_DECIDER
  task_type: PATH_DECIDER
  task_type: RULE_BASED_STOP_DECIDER
  task_type: ST_BOUNDS_DECIDER
  task_type: SPEED_BOUNDS_PRIORI_DECIDER
  task_type: SPEED_HEURISTIC_OPTIMIZER
  task_type: SPEED_DECIDER
  task_type: SPEED_BOUNDS_FINAL_DECIDER
  # task_type: PIECEWISE_JERK_SPEED_OPTIMIZER
  task_type:
PIECEWISE_JERK_NONLINEAR_SPEED_OPTIMIZER
  task_type: RSS_DECIDER
}
```

车道内路径限制

nudge路径限制

这个是默认 lane follow 场景的 task 配置列表，第一个 task 是一个确定是否换道的 task，在这个 task 里会根据 routing 判断是否要变道，以及旁边车道障碍物车情况决策是否要换道，第二个是判断是否借道的决策，比如前方有车挡在了前面，会根据旁边的车道线型，障碍物阻塞时间确定是否应该借道，第三个是确定路径边界，会根据前面两个 decider 确定路径的边界，正常的路径边界就是自车道内的路径限制，如果旁边有障碍物车辆，但是自车还能通过，会将障碍物车的边界包含进来。

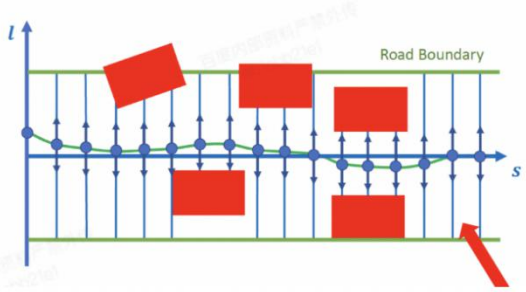
```
scenario_type: LANE_FOLLOW
stage_type: LANE_FOLLOW_DEFAULT_STAGE
stage_config: {
  stage_type: LANE_FOLLOW_DEFAULT_STAGE
  enabled: true
  task_type: LANE_CHANGE_DECIDER
  task_type: PATH_REUSE_DECIDER
  task_type: PATH_LANE_BORROW_DECIDER
  task_type: PATH_BOUNDS_DECIDER
  task_type: PIECEWISE_JERK_PATH_OPTIMIZER
  task_type: PATH_ASSESSMENT_DECIDER
  task_type: PATH_DECIDER
  task_type: RULE_BASED_STOP_DECIDER
  task_type: ST_BOUNDS_DECIDER
  task_type: SPEED_BOUNDS_PRIORI_DECIDER
  task_type: SPEED_HEURISTIC_OPTIMIZER
  task_type: SPEED_DECIDER
  task_type: SPEED_BOUNDS_FINAL_DECIDER
  # task_type: PIECEWISE_JERK_SPEED_OPTIMIZER
  task_type:
PIECEWISE_JERK_NONLINEAR_SPEED_OPTIMIZER
  task_type: RSS_DECIDER
}
```

借道路径限制

换道路径限制

如果产生了借道的决策，就是前车阻塞了自车车道，也会把旁边车道边界和障碍物车边加进来，如果产生了换道决策，就会把换道的路径边界加进来。

```
scenario_type: LANE_FOLLOW
stage_type: LANE_FOLLOW_DEFAULT_STAGE
stage_config: {
  stage_type: LANE_FOLLOW_DEFAULT_STAGE
  enabled: true
  task_type: LANE_CHANGE_DECIDER
  task_type: PATH_REUSE_DECIDER
  task_type: PATH_LANE_BORROW_DECIDER
  task_type: PATH_BOUNDS_DECIDER
  task_type: PIECEWISE_JERK_PATH_OPTIMIZER
  task_type: PATH_ASSESSMENT_DECIDER
  task_type: PATH_DECIDER
  task_type: RULE_BASED_STOP_DECIDER
  task_type: ST_BOUNDS_DECIDER
  task_type: SPEED_BOUNDS_PRIORI_DECIDER
  task_type: SPEED_HEURISTIC_OPTIMIZER
  task_type: SPEED_DECIDER
  task_type: SPEED_BOUNDS_FINAL_DECIDER
  # task_type: PIECEWISE_JERK_SPEED_OPTIMIZER
  task_type:
PIECEWISE_JERK_NONLINEAR_SPEED_OPTIMIZER
  task_type: RSS_DECIDER
}
```

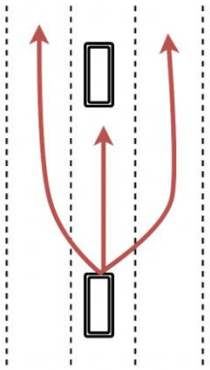


- $f = w_l \sum_{i=0}^{n-1} l_i^2 + w_{l'} \sum_{i=0}^{n-1} l_i'^2 + w_{l''} \sum_{i=0}^{n-1} l_i''^2 + w_{l'''} \sum_{i=0}^{n-2} l_{i\to i+1}'''^2$

优化目标函数

接下来是对每个边界做路径优化，路径优化就是优化每个路径点的 l l 的一阶导 二阶导目标函数就是期望 l l 的一阶导 二阶导尽可能的小，因为一阶导二阶导越小，路径也就越平滑，l 越小路径也就会越贴近车道中心线，约束对于 l 的话，就是定义的边界，一阶导、二阶导的约束就是根据车辆运动学关系，轮胎的最大转角，最大转速和道路的曲率确定。

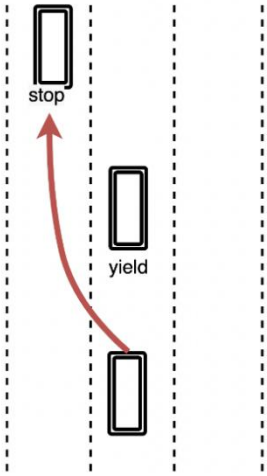
```
scenario_type: LANE_FOLLOW
stage_type: LANE_FOLLOW_DEFAULT_STAGE
stage_config: {
  stage_type: LANE_FOLLOW_DEFAULT_STAGE
  enabled: true
  task_type: LANE_CHANGE_DECIDER
  task_type: PATH_REUSE_DECIDER
  task_type: PATH_LANE_BORROW_DECIDER
  task_type: PATH_BOUNDS_DECIDER
  task_type: PIECEWISE_JERK_PATH_OPTIMIZER
  task_type: PATH_ASSESSMENT_DECIDER
  task_type: PATH_DECIDER
  task_type: RULE_BASED_STOP_DECIDER
  task_type: ST_BOUNDS_DECIDER
  task_type: SPEED_BOUNDS_PRIORI_DECIDER
  task_type: SPEED_HEURISTIC_OPTIMIZER
  task_type: SPEED_DECIDER
  task_type: SPEED_BOUNDS_FINAL_DECIDER
  # task_type: PIECEWISE_JERK_SPEED_OPTIMIZER
  task_type:
PIECEWISE_JERK_NONLINEAR_SPEED_OPTIMIZER
  task_type: RSS_DECIDER
}
```



路径是否和障碍物碰撞
路径长度
路径是否会停在对向车道
路径离自车远近
哪个路径更早回自车道
…

最后会优化出多条路径，比如在自车道的、左侧超车的、右侧超车的路径，会根据一定规则选择最优路径，比如路径是否会和障碍物碰撞、路径长度、路径是否会停在对向车道、路径离自车远近、哪个路径更早回自车道。

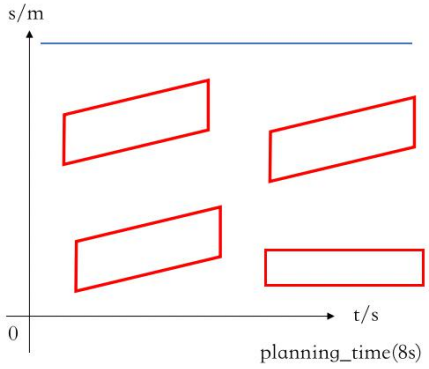
```
scenario_type: LANE_FOLLOW
stage_type: LANE_FOLLOW_DEFAULT_STAGE
stage_config: {
  stage_type: LANE_FOLLOW_DEFAULT_STAGE
  enabled: true
  task_type: LANE_CHANGE_DECIDER
  task_type: PATH_REUSE_DECIDER
  task_type: PATH_LANE_BORROW_DECIDER
  task_type: PATH_BOUNDS_DECIDER
  task_type: PIECEWISE_JERK_PATH_OPTIMIZER
  task_type: PATH_ASSESSMENT_DECIDER
  task_type: PATH_DECIDER
  task_type: RULE_BASED_STOP_DECIDER
  task_type: ST_BOUNDS_DECIDER
  task_type: SPEED_BOUNDS_PRIORI_DECIDER
  task_type: SPEED_HEURISTIC_OPTIMIZER
  task_type: SPEED_DECIDER
  task_type: SPEED_BOUNDS_FINAL_DECIDER
  # task_type: PIECEWISE_JERK_SPEED_OPTIMIZER
  task_type:
PIECEWISE_JERK_NONLINEAR_SPEED_OPTIMIZER
  task_type: RSS_DECIDER
}
```



根据路径对障碍物产生决策

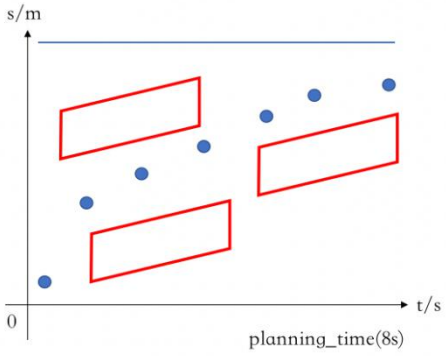
然后根据选出的最优路径作为最终轨迹的路径。Path decider 会根据这条路径对障碍物产生决策，比如对于阻塞的障碍物产生停止决策，对于越过的障碍物产生绕行的决策。

```
scenario_type: LANE_FOLLOW
stage_type: LANE_FOLLOW_DEFAULT_STAGE
stage_config: {
  stage_type: LANE_FOLLOW_DEFAULT_STAGE
  enabled: true
  task_type: LANE_CHANGE_DECIDER
  task_type: PATH_REUSE_DECIDER
  task_type: PATH_LANE_BORROW_DECIDER
  task_type: PATH_BOUNDS_DECIDER
  task_type: PIECEWISE_JERK_PATH_OPTIMIZER
  task_type: PATH_ASSESSMENT_DECIDER
  task_type: PATH_DECIDER
  task_type: RULE_BASED_STOP_DECIDER
  task_type: ST_BOUNDS_DECIDER
  task_type: SPEED_BOUNDS_PRIORI_DECIDER
  task_type: SPEED_HEURISTIC_OPTIMIZER
  task_type: SPEED_DECIDER
  task_type: SPEED_BOUNDS_FINAL_DECIDER
  # task_type: PIECEWISE_JERK_SPEED_OPTIMIZER
  task_type:
PIECEWISE_JERK_NONLINEAR_SPEED_OPTIMIZER
  task_type: RSS_DECIDER
}
```



接下来是速度规划，速度规划是在 st 曲线，所以要把障碍物预测的轨迹投影到 st 坐标系上，如果是向前行驶的车就会形成类似这样的曲线，纵向矩形的宽度就是车的宽度。然后左右的边就是障碍车进入和离开路径的时间，如果是停止的车，就是类似这样的边框。速度规划分为两次优化，一次是粗优化，一次是细优化。这个 prior decider 是确定粗优化边界，这里可以看道速度规划的曲线既可以从上面超车过去，也可以从下面让行过去，所以这是个非凸的优化问题，用二次规划的优化算法是无法求解的。

```
scenario_type: LANE_FOLLOW
stage_type: LANE_FOLLOW_DEFAULT_STAGE
stage_config: {
  stage_type: LANE_FOLLOW_DEFAULT_STAGE
  enabled: true
  task_type: LANE_CHANGE_DECIDER
  task_type: PATH_REUSE_DECIDER
  task_type: PATH_LANE_BORROW_DECIDER
  task_type: PATH_BOUNDS_DECIDER
  task_type: PIECEWISE_JERK_PATH_OPTIMIZER
  task_type: PATH_ASSESSMENT_DECIDER
  task_type: PATH_DECIDER
  task_type: RULE_BASED_STOP_DECIDER
  task_type: ST_BOUNDS_DECIDER
  task_type: SPEED_BOUNDS_PRIORI_DECIDER
  task_type: SPEED_HEURISTIC_OPTIMIZER
  task_type: SPEED_DECIDER
  task_type: SPEED_BOUNDS_FINAL_DEC
  task_type:
PIECEWISE_JERK_NONLINEAR_SPEED_OPTIMIZER
  task_type: RSS_DECIDER
}
```

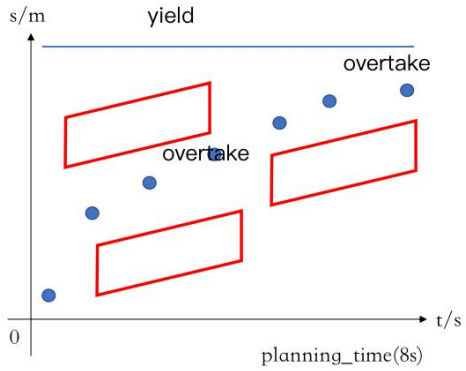


$$C(i+1,j+k) = min \begin{cases} C_{obs}(i+1,j+k) + C_{spatial}(i+1,j+k) + C(i,j) + C_{edge}(i,j,i+1,j+k) \\ C(i+1,j+k) \end{cases}$$

$$C_{edge} = C_{speed} + C_{acc} + C_{jerk}$$

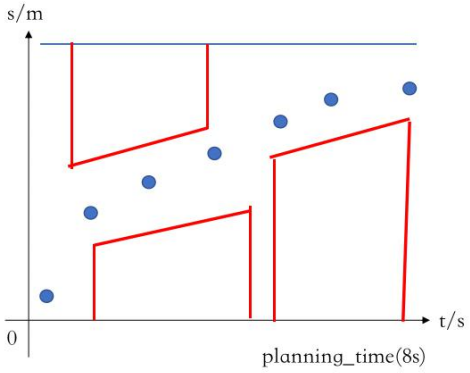
所以这里先用动态规划作为速度规划的粗优化，用来确定规划的边界，动态规划里可以看到 cost，首先是距离障碍物边界的距离，越大越好，然后是加速度和 jerk 越小越好。

```
scenario_type: LANE_FOLLOW
stage_type: LANE_FOLLOW_DEFAULT_STAGE
stage_config: {
  stage_type: LANE_FOLLOW_DEFAULT_STAGE
  enabled: true
  task_type: LANE_CHANGE_DECIDER
  task_type: PATH_REUSE_DECIDER
  task_type: PATH_LANE_BORROW_DECIDER
  task_type: PATH_BOUNDS_DECIDER
  task_type: PIECEWISE_JERK_PATH_OPTIMIZER
  task_type: PATH_ASSESSMENT_DECIDER
  task_type: PATH_DECIDER
  task_type: RULE_BASED_STOP_DECIDER
  task_type: ST_BOUNDS_DECIDER
  task_type: SPEED_BOUNDS_PRIORI_DECIDER
  task_type: SPEED_HEURISTIC_OPTIMIZER
  task_type: SPEED_DECIDER
  task_type: SPEED_BOUNDS_FINAL_DECIDER
  # task_type: PIECEWISE_JERK_SPEED_OPTIMIZER
  task_type:
PIECEWISE_JERK_NONLINEAR_SPEED_OPTIMIZER
  task_type: RSS_DECIDER
}
```



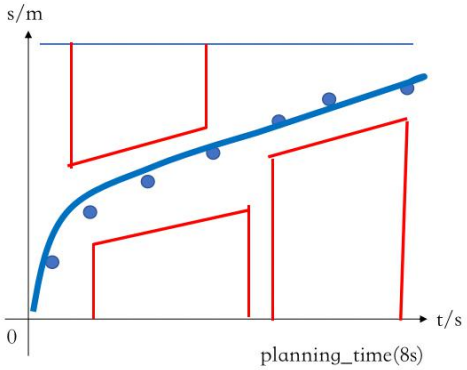
在 speed decider 中，就是来确定对于每个障碍物的决策。

```
scenario_type: LANE_FOLLOW
stage_type: LANE_FOLLOW_DEFAULT_STAGE
stage_config: {
  stage_type: LANE_FOLLOW_DEFAULT_STAGE
  enabled: true
  task_type: LANE_CHANGE_DECIDER
  task_type: PATH_REUSE_DECIDER
  task_type: PATH_LANE_BORROW_DECIDER
  task_type: PATH_BOUNDS_DECIDER
  task_type: PIECEWISE_JERK_PATH_OPTIMIZER
  task_type: PATH_ASSESSMENT_DECIDER
  task_type: PATH_DECIDER
  task_type: RULE_BASED_STOP_DECIDER
  task_type: ST_BOUNDS_DECIDER
  task_type: SPEED_BOUNDS_PRIORI_DECIDER
  task_type: SPEED_HEURISTIC_OPTIMIZER
  task_type: SPEED_DECIDER
  task_type: SPEED_BOUNDS_FINAL_DECIDER
  # task_type: PIECEWISE_JERK_SPEED_OPTIMIZER
  task_type:
PIECEWISE_JERK_NONLINEAR_SPEED_OPTIMIZER
  task_type: RSS_DECIDER
}
```



有了对于每一个障碍物的决策，就可以确定优化的边界。

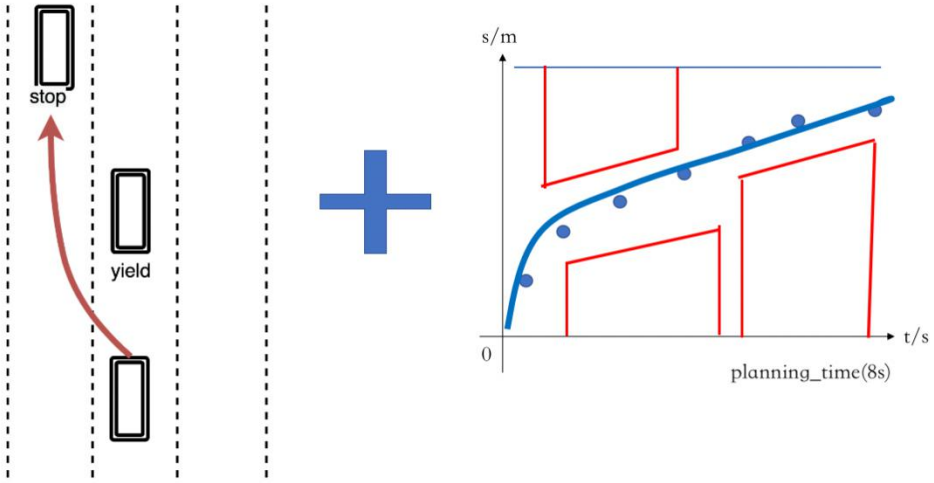
```
scenario_type: LANE_FOLLOW
stage_type: LANE_FOLLOW_DEFAULT_STAGE
stage_config: {
  stage_type: LANE_FOLLOW_DEFAULT_STAGE
  enabled: true
  task_type: LANE_CHANGE_DECIDER
  task_type: PATH_REUSE_DECIDER
  task_type: PATH_LANE_BORROW_DECIDER
  task_type: PATH_BOUNDS_DECIDER
  task_type: PIECEWISE_JERK_PATH_OPTIMIZER
  task_type: PATH_ASSESSMENT_DECIDER
  task_type: PATH_DECIDER
  task_type: RULE_BASED_STOP_DECIDER
  task_type: ST_BOUNDS_DECIDER
  task_type: SPEED_BOUNDS_PRIORI_DECIDER
  task_type: SPEED_HEURISTIC_OPTIMIZER
  task_type: SPEED_DECIDER
  task_type: SPEED_BOUNDS_FINAL_DECIDER
  # task_type: PIECEWISE_JERK_SPEED_OPTIMIZER
  task_type:
PIECEWISE_JERK_NONLINEAR_SPEED_OPTIMIZER
  task_type: RSS_DECIDER
}
```



$$cost = \sum_{i=0}^{n-1} w_{ref}(s_i - s\_ref_i)^2 + w_v(\dot{s}_i - v\_ref)^2 + w_a\ddot{s}_i^2 + w_j(\frac{\ddot{s}_{i+1} - \ddot{s}_i}{\Delta t})^2 + w_{lat\_acc}lat\_acc_i^2 + w_{soft}s\_lower_i + w_{soft}s\_upper_i$$

再对速度进行一次细优化，这里使用的非线性优化算法，可以看到目标函数的设计上包含了到参考曲线的距离，期望到参考曲线越小越好，到参考速度的距离，期望跟随参考速度的变化，以及加速度和 jerk 越小越好。

4. 轨迹组合



最后将路径曲线和速度曲线整合到一起就成轨迹。

6.simplecode 调参

参考《使用 Sim_Control 仿真调试 PnC 模块》

7.扩展知识

A*、Hybrid A*

frenet 坐标系

参考线平滑

二次规划

动态规划

8.参考链接

- apollo 规划能力汇总 - https://apollo.auto/document_cn.html?target=/Apollo-Homepage-Document/Apollo_Doc_CN_6_0/
- Apollo 规划模块详解 - https://mp.weixin.qq.com/s/TRY7_iAqSMAd2rgUZdELLw
- Apollo 规划模块之 Scenario - https://mp.weixin.qq.com/s/PjFzKqyFJUj9p_nbGRNTtQ
- Apollo 6.0 规划算法解析 - 知乎 (zhihu.com)
- 解析百度 Apollo 之决策规划模块 - 程十三 - 博客园 (cnblogs.com)
- apollo 介绍之 Routing 模块(六) - 知乎 (zhihu.com)
- apollo 介绍之 planning 模块(四) - 知乎 (zhihu.com)
- Apollo 规划模块详解（一）：整体架构+算法说明 - 知乎 (zhihu.com)
- 开发者说丨离散点曲线平滑原理 (qq.com)