Traffic Decider

```
Status TrafficDecider::Execute(
    Frame *frame, ReferenceLineInfo *reference_line_info,
    const std::shared_ptr<DependencyInjector> &injector) {
  CHECK_NOTNULL(frame);
  CHECK_NOTNULL(reference_line_info);

  for (const auto &rule_config : rule_configs_.config()) {
    if (!rule_config.enabled()) {
      ADEBUG << "Rule " << rule_config.rule_id() << " not enabled";
      continue;
    }
    //generate current scenario decider
    auto rule = s_rule_factory.CreateObject(rule_config.rule_id(), rule_config,
                                            injector);

    //concrete scenario decider to apply its rule
    if (!rule) {
      AERROR << "Could not find rule " << rule_config.DebugString();
      continue;
    }
    rule->ApplyRule(frame, reference_line_info);
    ADEBUG << "Applied rule "
           << TrafficRuleConfig::RuleId_Name(rule_config.rule_id());
  }

  BuildPlanningTarget(reference_line_info);
  return Status::OK();
}
```

对交通规则的整个梳理流程在上图所示的 for 循环中进行，通过 for 循环来遍历配置文件 /apollo/modules/planning/conf/traffic_rule_config.pb.txt 中设置的交通规则，主要涉及的交通规则有以下九类：

BACKSIDE_VEHICLE（后向来车）

CROSSWALK（人行横道）

DESTINATION（目的地）

KEEP_CLEAR（禁停区）

REFERENCE_LINE_END（参考线结束）

REROUTING（重新路由）

STOP_SIGN（停车）

TRAFFIC_LIGHT（交通灯）

YIELD_SIGN（让行）

目录

1. Desitination

到达目的地的情况下,障碍物促使本车采取靠边停车或者寻找合适的停车点。若存在有效的靠边停车点,则即创建虚拟障碍物,并封装成新的 PathObstacle 加入该 ReferenceLineInfo 的 pathdecision 中。

算法步骤：

- check frame 和 reference line 非空
- check reference 是否有终点
- 转换 destination point from x-y to s-l，判断终点是否被设置到自车后面，但是逻辑上还未过终点。
- 如果 pull_over 被允许，计算 pull_over_sl，并结合汽车参数以及 config 中 destiantion 下的 stop_distance 参数，计算出 stop_line_s，根据 stop_line_s 设置停车点。

```
const double stop_line_s = pull_over_sl.s() +
                           VehicleConfigHelper::GetConfig()
                               .vehicle_param()
                               .front_edge_to_center() +
                           config_.destination().stop_distance();
util::BuildStopDecision(
    stop_wall_id, stop_line_s, config_.destination().stop_distance(),
    StopReasonCode::STOP_REASON_PULL_OVER, wait_for_obstacle_ids,
    TrafficRuleConfig::RuleId_Name(config_.rule_id()), frame,
    reference_line_info);
```

- 默认：（如果靠边停车功能没有激活,或者靠边停车停车情况下没有有效的停车点。需要注意的是,此场景下车辆的停车点并不是我们所选择的目的地,而是距离一小段距离）。取 dest_lans_s 为 0

和 rounting 终点减去虚拟墙长度减去 stop_distance 的最大值，建立停止点。

```
const double dest_lane_s =
    std::fmax(0.0, routing_end.s() - FLAGS_virtual_stop_wall_length -
                       config_.destination().stop_distance());
util::BuildStopDecision(stop_wall_id, routing_end.id(), dest_lane_s,
                        config_.destination().stop_distance(),
                        StopReasonCode::STOP_REASON_DESTINATION,
                        wait_for_obstacle_ids,
                        TrafficRuleConfig::RuleId_Name(config_.rule_id()),
                        frame, reference_line_info);
```

2. Reference_line_end

当参考线结束时,一般需要重新路由查询,所以需要停车,这种情况下如果程序正常,一般是前方没有路了,需要重新查询当前点到目的地新的路由。

算法步骤：

- 获取 reference line info
- 计算 remain_s，如果还未靠近终点，则不生成虚拟停止墙。

```
//step 2, calculate remain s which reference line to current ad-vehicle
// if remain_s is smaller than min reference line remain length, return OK
double remain_s =
    reference_line.Length() - reference_line_info->AdcSlBoundary().end_s();
if (remain_s >
    config_.reference_line_end().min_reference_line_remain_length()) {
  return Status::OK();
}
```

- 创建虚拟墙

```
// step 3 create a virtual stop wall at the end of reference line to stop the adc
std::string virtual_obstacle_id =
    REF_LINE_END_VO_ID_PREFIX + reference_line_info->Lanes().Id();
double obstacle_start_s =
    reference_line.Length() - 2 * FLAGS_virtual_stop_wall_length;
auto* obstacle = frame->CreateStopObstacle(
    reference_line_info, virtual_obstacle_id, obstacle_start_s);

// check if obstacle is created
if (!obstacle) {
  return Status(common::PLANNING_ERROR,
                "Failed to create reference line end obstacle");
}
```

- 将障碍物添加到 path_decision

```
// step 4 add obstacle to reference line info, add it into path decision
Obstacle* stop_wall = reference_line_info->AddObstacle(obstacle);
if (!stop_wall) {
  return Status(
      common::PLANNING_ERROR,
      "Failed to create path obstacle for reference line end obstacle");
}
```

- 生成 stop_decision 并将其加入 path_decision

```
// stop 5 build stop decision
const double stop_line_s =
    obstacle_start_s - config_.reference_line_end().stop_distance();
auto stop_point = reference_line.GetReferencePoint(stop_line_s);

ObjectDecisionType stop;
auto stop_decision = stop.mutable_stop();
stop_decision->set_reason_code(StopReasonCode::STOP_REASON_DESTINATION);
stop_decision->set_distance_s(-config_.reference_line_end().stop_distance());
stop_decision->set_stop_heading(stop_point.heading());
stop_decision->mutable_stop_point()->set_x(stop_point.x());
stop_decision->mutable_stop_point()->set_y(stop_point.y());
stop_decision->mutable_stop_point()->set_z(0.0);

// add stop decision to path decision
auto* path_decision = reference_line_info->path_decision();
path_decision->AddLongitudinalDecision(
    TrafficRuleConfig::RuleId_Name(config_.rule_id()), stop_wall->Id(), stop);

return Status::OK();
```

3. Rerouting

重新路由的处理思想就是根据车辆当前的位置等信息判断是否需要重新请求路由

算法步骤：

- 接近/到达终点，不需要重新路由

```
//set rerouting threshold
//if has reach destination or distance to destination is smaller than threshold, return true
static constexpr double kRerouteThresholdToEnd = 20.0;
for (const auto& ref_line_info : frame_->reference_line_info()) {
  if (ref_line_info.ReachedDestination() ||
      ref_line_info.SDistanceToDestination() < kRerouteThresholdToEnd) {
    return true;
  }
}
```

- 如果当前是直行道，则不需要重新申请路由

```
const auto& segments = reference_line_info_->Lanes();
// 1. If current reference line is drive forward, no rerouting.
if (segments.NextAction() == routing::FORWARD) {
  // if not current lane, do not check for rerouting
  return true;
}
```

- 如果车辆不在当前路由段上，则不需要重新申请路由

```
// 2. If vehicle is not on current reference line yet, no rerouting
if (!segments.IsOnSegment()) {
  return true;
}
```

- 如果车辆即将退出当前 passage，则不需要重新申请路由、

```
// 3. If current reference line can connect to next passage, no rerouting
if (segments.CanExit()) {
  return true;
}
```

- 若当前路由段的终点不在当前参考线所在的车道上，则不需要重新申请路由

```
// 4. If the end of current passage region not appeared, no rerouting
const auto& route_end_waypoint = segments.RouteEndWaypoint();
if (!route_end_waypoint.lane) {
  return true;
}

//get end point of route,if transfer from XY to SL fail, return false
auto point = route_end_waypoint.lane->GetSmoothPoint(route_end_waypoint.s);
const auto& reference_line = reference_line_info_->reference_line();
common::SLPoint sl_point;
if (!reference_line.XYToSL(point, &sl_point)) {
  AERROR << "Failed to project point: " << point.ShortDebugString();
  return false;
}

//if point is not in current lane, no rerouting
if (!reference_line.IsOnLane(sl_point)) {
  return true;
}
```

- 如果车辆在变更路由之前不能到达当前 passage 的终点，则不需要重新申请路由

```
// 5. If the end of current passage region is further than kPrepareRoutingTime
// * speed, no rerouting
double adc_s = reference_line_info_->AdcSlBoundary().end_s();
const auto vehicle_state = injector_->vehicle_state();
double speed = vehicle_state->linear_velocity();
const double prepare_rerouting_time =
    config_.rerouting().prepare_rerouting_time();
const double prepare_distance = speed * prepare_rerouting_time;
if (sl_point.s() > adc_s + prepare_distance) {
  ADEBUG << "No need rerouting now because still can drive for time: "
         << prepare_rerouting_time << " seconds";
  return true;
}
```

- 如果车辆在短时间内已经重新申请过路由，则不需要再重新申请路由

```
// 6. Check if we have done rerouting before
//rerouting is nullptr, rerouting
auto* rerouting = injector_->planning_context()
                           ->mutable_planning_status()
                           ->mutable_rerouting();
if (rerouting == nullptr) {
  AERROR << "rerouting is nullptr.";
  return false;
}

//rerouting cooldown time is not beyond, not rerouting
double current_time = Clock::NowInSeconds();
if (rerouting->has_last_rerouting_time() &&
    (current_time - rerouting->last_rerouting_time() <
     config_.rerouting().cooldown_time())) {
  ADEBUG << "Skip rerouting and wait for previous rerouting result";
  return true;
}
```

- 默认情况，重新路由。如果给 frame 发 rerouting request 失败，则返回 false

```
 //default, rerouting
//if failed to send rerouting request to frame, return false
if (!frame_->Rerouting(injector_->planning_context())) {
  AERROR << "Failed to send rerouting request";
  return false;
}

// store last rerouting time.
rerouting->set_last_rerouting_time(current_time);
return true;
```

4. StopSign

看到这个 Stop_sign 必须得停车，所以对这个停止牌的处理逻辑就是生成停止墙。

算法逻辑

- 检查 frame 和 reference line info 非空，检查是否 enable stop sign

```
CHECK_NOTNULL(frame);
CHECK_NOTNULL(reference_line_info);

if (!config_.stop_sign().enabled()) {
  return;
}
```

- 遍历 stop_sign_overlap region，检查汽车是否已经驶过 stop sign，已经走过则不再处理

```
for (const auto& stop_sign_overlap : stop_sign_overlaps) {
  if (stop_sign_overlap.end_s <= adc_back_edge_s) {
    continue;
  }
```

- 检查当前 stop sign 的状态是否为 done，为 done 则不再处理

```
    //CHECK IF it has finished for the current stop sign
    if (stop_sign_overlap.object_id ==
        stop_sign_status.done_stop_sign_overlap_id()) {
      continue;
    }
```

- 生成虚拟墙

```
// build stop decision
ADEBUG << "BuildStopDecision: stop_sign[" << stop_sign_overlap.object_id
       << "] start_s[" << stop_sign_overlap.start_s << "]";
const std::string virtual_obstacle_id =
    STOP_SIGN_VO_ID_PREFIX + stop_sign_overlap.object_id;
const std::vector<std::string> wait_for_obstacle_ids(
    stop_sign_status.wait_for_obstacle_id().begin(),
    stop_sign_status.wait_for_obstacle_id().end());
 util::BuildStopDecision(virtual_obstacle_id, stop_sign_overlap.start_s,
                         config_.stop_sign().stop_distance(),
                         StopReasonCode::STOP_REASON_STOP_SIGN,
                         wait_for_obstacle_ids,
                         TrafficRuleConfig::RuleId_Name(config_.rule_id()),
                         frame, reference_line_info);
```

5. TrafficLight

对交通灯的处理是红灯停，绿灯行，黄灯也停

算法逻辑

- 检查 frame 和 reference line info 非空，检查是否 enable traffic light

首先从 map 中获取存储所有信号灯的 vector，然后对每一个信号灯进行遍历；

- 如果根据车辆当前的位置判断已经驶过红绿灯，则当前遍历的红绿灯则不作任何处理

```
//Step 2 check if vehicle has passed traffic light
const std::vector<PathOverlap>& traffic_light_overlaps =
    reference_line_info->reference_line().map_path().signal_overlaps();
for (const auto& traffic_light_overlap : traffic_light_overlaps) {
  if (traffic_light_overlap.end_s <= adc_back_edge_s) {
    continue;
  }
```

- 根据红绿灯状态判断当前红绿灯是否已经完成

```
  //step 3: check if traffic-light-stop already finished, set by scenario/stage
  bool traffic_light_done = false;
  for (const auto& done_traffic_light_overlap_id :
       traffic_light_status.done_traffic_light_overlap_id()) {
    if (traffic_light_overlap.object_id == done_traffic_light_overlap_id) {
      traffic_light_done = true;
      break;
    }
  }
```

- 根据自车与交通灯的欧氏距离以及 s 差值，判断交通灯是否偏离，如果是，则不做处理；此处主要判断的是红绿灯是否沿着参考线（因为有时候难以保证当前红绿灯是正对着的还是侧面的）

```
//step 4 check if current traffic light is along reference line
// work around incorrect s-projection along round routing
static constexpr double kSDiscrepanceTolerance = 10.0;
const auto& reference_line = reference_line_info->reference_line();
common::SLPoint traffic_light_sl;
traffic_light_sl.set_s(traffic_light_overlap.start_s);
traffic_light_sl.set_l(0);
common::math::Vec2d traffic_light_point;
reference_line.SLToXY(traffic_light_sl, &traffic_light_point);
common::math::Vec2d adc_position = {injector_->vehicle_state()->x(),
                                    injector_->vehicle_state()->y()};
const double distance =
    common::util::DistanceXY(traffic_light_point, adc_position);
const double s_distance = traffic_light_overlap.start_s - adc_front_edge_s;
ADEBUG << "traffic_light[" << traffic_light_overlap.object_id
       << "] start_s[" << traffic_light_overlap.start_s << "] s_distance["
       << s_distance << "] actual_distance[" << distance << "]";

// here we need check if s-distance is very close to L2 distance, if not, it means the traffic
// reference line, it may be the traffic light in the left or right, so we can skip the current
if (s_distance >= 0 &&
    fabs(s_distance - distance) > kSDiscrepanceTolerance) {
  ADEBUG << "SKIP traffic_light[" << traffic_light_overlap.object_id
         << "] close in position, but far away along reference line";
  continue;
}
```

- 剩下的就是根据交通灯的颜色判断处理，如果是绿色，则不做处理，如果是红/黄/未知的颜色色，并且计算出的停车减速度大于最大减速度，则说明速度太高或者距离太近，无法处理，则直接让其通过（此处貌似有明显的问题，对于红黄灯，假如说速度已经无法停下来，应该先减速再说）。

```
// step 6 check if green light or stop deceleration beyond normal range which the adv can reac
if (signal_color == perception::TrafficLight::GREEN) {
  continue;
}

// Red/Yellow/Unknown: check deceleration
if (stop_deceleration > config_.traffic_light().max_stop_deceleration()) {
  AWARN << "stop_deceleration too big to achieve.  SKIP red light";
  continue;
}
```

- 生成虚拟墙

```
// build stop decision
ADEBUG << "BuildStopDecision: traffic_light["
       << traffic_light_overlap.object_id << "] start_s["
       << traffic_light_overlap.start_s << "]";
std::string virtual_obstacle_id =
    TRAFFIC_LIGHT_VO_ID_PREFIX + traffic_light_overlap.object_id;
const std::vector<std::string> wait_for_obstacles;

//it was put into commom/utils package

//step 7 decisiion make here a stop wall is set
util::BuildStopDecision(virtual_obstacle_id, traffic_light_overlap.start_s,
                        config_.traffic_light().stop_distance(),
                        StopReasonCode::STOP_REASON_SIGNAL,
                        wait_for_obstacles,
                        TrafficRuleConfig::RuleId_Name(config_.rule_id()),
                        frame, reference_line_info);
```

6. Yield Sign

停车让行的处理逻辑与交通灯一致。

算法逻辑：

- 检查 frame 和 reference line info 非空，检查是否 enable yield sign

遍历数组：

- 检查汽车是否已经超过当前 yield sign

```
//step 2 : check if yield sign is before adv
for (const auto& yield_sign_overlap : yield_sign_overlaps) {
  if (yield_sign_overlap.end_s <= adc_front_edge_s) {
    continue;
  }
```

- 检查当前 yield sign 的状态，如果已经完成，则不做处理

```
  // check if yield-sign-stop already finished, set by scenario/stage
  bool yield_sign_done = false;
  for (const auto& done_yield_sign_overlap_id :
       yield_sign_status.done_yield_sign_overlap_id()) {
    if (yield_sign_overlap.object_id == done_yield_sign_overlap_id) {
      yield_sign_done = true;
      break;
    }
  }
  if (yield_sign_done) {
    continue;
  }
```

- 建立虚拟墙

```
// build stop decision
ADEBUG << "BuildStopDecision: yield_sign[" << yield_sign_overlap.object_id
       << "] start_s[" << yield_sign_overlap.start_s << "]";
const std::string virtual_obstacle_id =
    YIELD_SIGN_VO_ID_PREFIX + yield_sign_overlap.object_id;
const std::vector<std::string> wait_for_obstacle_ids(
    yield_sign_status.wait_for_obstacle_id().begin(),
    yield_sign_status.wait_for_obstacle_id().end());
util::BuildStopDecision(virtual_obstacle_id, yield_sign_overlap.start_s,
                        config_.yield_sign().stop_distance(),
                        StopReasonCode::STOP_REASON_YIELD_SIGN,
                        wait_for_obstacle_ids,
                        TrafficRuleConfig::RuleId_Name(config_.rule_id()),
                        frame, reference_line_info);
```

7. backside_vehicle

在对后向来车的处理决策中,会对 path_decision 中的每一个障碍物进行遍历,在遍历时,如果是后向车辆,则选择忽略,即通过调用函数 PathDecision::AddLongitudinalDecision()和函数 PathDecision::AddLateralDecision()为障碍物打上"ignore” 的标签,这两个函数的处理逻辑比较简单,这里就不再展开介绍.需要"ignore” 的后向车辆主要是下面的 2,3,4,三种情况,在 4 中,如果后向车辆的横向距离较大,其可能会选择超车,不能忽略,因此在此处不作处理.

算法逻辑:

对于每个在 path decision 中的 obstacle 遍历:

- 在对后向车辆的情况处理时,忽略前方障碍物及警戒级别的障碍物，不做处理。

```
for (const auto* obstacle : path_decision->obstacles().Items()) {
  //step 1,check if current obstacle is at the front of adv, if it is, ignore
  if (obstacle->PerceptionSLBoundary().end_s() >= adc_sl_boundary.end_s() ||
      obstacle->IsCautionLevelObstacle()) {
    // don't ignore such vehicles.
    continue;
  }
```

- 参考线上没有障碍物（后车）轨迹，则忽略

```
//step 2, if trajectory of obstacle is empty in reference line, continue
if (obstacle->reference_line_st_boundary().IsEmpty()) {
  path_decision->AddLongitudinalDecision("backside_vehicle/no-st-region",
                                         obstacle->Id(), ignore);
  path_decision->AddLateralDecision("backside_vehicle/no-st-region",
                                    obstacle->Id(), ignore);
  continue;
}
```

- 如果障碍物轨迹距离无人车的最近距离小于一个车长,则忽略

```
// step 3 Ignore the car comes from back of ADC
// if min s of the obstacle's trajectory is smaller than car length, ignore
if (obstacle->reference_line_st_boundary().min_s() < -adc_length_s) {
  path_decision->AddLongitudinalDecision("backside_vehicle/st-min-s < adc",
                                         obstacle->Id(), ignore);
  path_decision->AddLateralDecision("backside_vehicle/st-min-s < adc",
                                    obstacle->Id(), ignore);
  continue;
}
```

- 如果距离本车横向距离小于一定阈值的后向车辆,说明车辆不会超车.选择忽略

```
const double lane_boundary =
    config_.backside_vehicle().backside_lane_width();
if (obstacle->PerceptionSLBoundary().start_s() < adc_sl_boundary.end_s()) {
  if (obstacle->PerceptionSLBoundary().start_l() > lane_boundary ||
      obstacle->PerceptionSLBoundary().end_l() < -lane_boundary) {
    continue;
  }
  path_decision->AddLongitudinalDecision("backside_vehicle/sl < adc.end_s",
                                         obstacle->Id(), ignore);
  path_decision->AddLateralDecision("backside_vehicle/sl < adc.end_s",
                                    obstacle->Id(), ignore);
  continue;
}
```

8. CrossWalk

在对遇人行横道的情况进行处理时,首先从地图中获取所有的人行横道,然后其进行遍历

- 当车头驶过人行横道的距离超过阈值 min_pass_s_distance 时,如果当前的 crosswalk_status 中 crosswalk_id 信息,且 crosswalk_id 就是当前正在遍历的 crosswalk_id,则清除 rosswalk_status 中的 ID 及停止时间信息,即不需要停车;

```
// step 1 . skip crosswalk if master vehicle body already passes the stop line
if (adc_front_edge_s - crosswalk_overlap->end_s >
    config_.crosswalk().min_pass_s_distance()) {
  if (mutable_crosswalk_status->has_crosswalk_id() &&
      mutable_crosswalk_status->crosswalk_id() == crosswalk_id) {
    mutable_crosswalk_status->clear_crosswalk_id();
    mutable_crosswalk_status->clear_stop_time();
  }

  ADEBUG << "SKIP: crosswalk_id[" << crosswalk_id
         << "] crosswalk_overlap_end_s[" << crosswalk_overlap->end_s
         << "] adc_front_edge_s[" << adc_front_edge_s
         << "]. adc_front_edge passes crosswalk_end_s + buffer.";
  continue;
}
```

- 如果已经通过行人横道,则忽略;

```
// step 2-- check if crosswalk already finished
if (finished_crosswalks.end() != std::find(finished_crosswalks.begin(),
                                           finished_crosswalks.end(),
                                           crosswalk_id)) {
  ADEBUG << "SKIP: crosswalk_id[" << crosswalk_id << "] crosswalk_end_s["
         << crosswalk_overlap->end_s << "] finished already";
  continue;
}
```

- 遍历 path decision 中的 obstalce

1. 计算减速度

```
for (const auto* obstacle : path_decision->obstacles().Items()) {
  //step 3.1  calculate stop deceleration
  const double stop_deceleration = util::GetADCStopDeceleration(
      injector_->vehicle_state(), adc_front_edge_s,
      crosswalk_overlap->start_s);
```

2. 检查是否需要停车

```
// step 3.2 check if it's needed to stop
bool stop = CheckStopForObstacle(reference_line_info, crosswalk_ptr,
                                 *obstacle, stop_deceleration);
```

2.1 检查类型是否是 pedestrain, bicycle, unknown movable or unknown。此时不需要停车。

```
// step 3.2.1 check type, if it's pedestrain, bicycle, unknown movable or unknown, not stop
if (obstacle_type != PerceptionObstacle::PEDESTRIAN &&
    obstacle_type != PerceptionObstacle::BICYCLE &&
    obstacle_type != PerceptionObstacle::UNKNOWN_MOVABLE &&
    obstacle_type != PerceptionObstacle::UNKNOWN) {
  ADEBUG << "obstacle_id[" << obstacle_id << "] type[" << obstacle_type_name
         << "]. skip";
  return false;
}
```

2.2 判断障碍物是否在扩展后的人行道范围之内，不在则不停车。

```
// step 3.2.2 if point is not in expanded polygon region, not stop
Vec2d point(perception_obstacle.position().x(),
            perception_obstacle.position().y());
const Polygon2d crosswalk_exp_poly =
    crosswalk_ptr->polygon().ExpandByDistance(
        config_.crosswalk().expand_s_distance());
bool in_expanded_crosswalk = crosswalk_exp_poly.IsPointIn(point);

if (!in_expanded_crosswalk) {
  ADEBUG << "skip: obstacle_id[" << obstacle_id << "] type["
         << obstacle_type_name << "] crosswalk_id[" << crosswalk_id
         << "]: not in crosswalk expanded area";
  return false;
}
```

2.3 如果 obstacle L 大于 loose L， 同时轨迹有重合，需要停车

```
//step 3.2.3
if (obstacle_l_distance >= config_.crosswalk().stop_loose_l_distance()) {
  // (1) when obstacle_l_distance is big enough(>= loose_l_distance),
  //     STOP only if paths crosses
  if (is_path_cross) {
    stop = true;
    ADEBUG << "need_stop(>=l2): obstacle_id[" << obstacle_id << "] type["
           << obstacle_type_name << "] crosswalk_id[" << crosswalk_id << "]";
  }
```

2.4 如果 obstacle L 小于 strict L， 障碍物在当前道路前方，需要停车

```
//step 3.2.4
else if (obstacle_l_distance <=
         config_.crosswalk().stop_strict_l_distance()) {
if (is_on_road) {
  // (2) when l_distance <= strict_l_distance + on_road
  //     always STOP
  if (obstacle_sl_point.s() > adc_end_edge_s) {
    // if obstacle is on road and at the front of vehicle, need stop
    stop = true;
    ADEBUG << "need_stop(<=l1): obstacle_id[" << obstacle_id << "] type["
           << obstacle_type_name << "] s[" << obstacle_sl_point.s()
           << "] adc_end_edge_s[ " << adc_end_edge_s << "] crosswalk_id["
           << crosswalk_id << "] ON_ROAD";
  }
```

2.5 如果 obstacle L 小于 strict L， 障碍物轨迹有重合，需要停车

```
//step 3.2.5 -- if obstale is not on road,but the trajectories are in cross, need stop
if (is_path_cross) {
  stop = true;
  ADEBUG << "need_stop(<=l1): obstacle_id[" << obstacle_id << "] type["
         << obstacle_type_name << "] crosswalk_id[" << crosswalk_id
         << "] PATH_CRSOSS";
```

2.6 如果 obstacle L 小于 strict L， 障碍物轨迹没有重合但是在接近，需要停车

```
else {
// (4) when l_distance <= strict_l_distance
//     + NOT on_road(i.e. on crosswalk/median etc)
//     STOP if he pedestrian is moving toward the ego vehicle
const auto obstacle_v = Vec2d(perception_obstacle.velocity().x(),
                              perception_obstacle.velocity().y());
const auto adc_path_point =
    Vec2d(injector_->ego_info()->start_point().path_point().x(),
          injector_->ego_info()->start_point().path_point().y());
const auto ovstacle_position =
    Vec2d(perception_obstacle.position().x(),
          perception_obstacle.position().y());
auto obs_to_adc = adc_path_point - ovstacle_position;
const double kEpsilon = 1e-6;
// v and delta_x innerprod,
//step 3.2.6 if obstacle is moving towards adv, need stop
if (obstacle_v.InnerProd(obs_to_adc) > kEpsilon) {
  stop = true;
  ADEBUG << "need_stop(<=l1): obstacle_id[" << obstacle_id << "] type["
         << obstacle_type_name << "] crosswalk_id[" << crosswalk_id
         << "] MOVING_TOWARD_ADC";
}
```

2.7 obstacle L 大于 strict L 小于 Losse L，轨迹有重合，需要停车

```
//step 3.2.7, if l bigger than strict l and smaller than loose l, and trajectories are cross, need stop
if (is_path_cross) {
  stop = true;
}
ADEBUG << "need_stop(between l1 & l2): obstacle_id[" << obstacle_id
       << "] type[" << obstacle_type_name << "] obstacle_l_distance["
       << obstacle_l_distance << "] crosswalk_id[" << crosswalk_id
       << "] USE_PREVIOUS_DECISION";
```

2.8 最后得到需要停车，但停车加速度大于 config 中在 crosswalk 场景下可以达到的最大减速度，同时，obstcle L 大于 Strict L， 则不需要停车（减速度过大但忽略也足够安全）

```
// check stop_deceleration
if (stop) {
  if (stop_deceleration >= config_.crosswalk().max_stop_deceleration()) {
    if (obstacle_l_distance > config_.crosswalk().stop_strict_l_distance()) {
      // SKIP when stop_deceleration is too big but safe to ignore
      //step 3.2.8, if handle stop and stop dec is bigger than max stop dec, and obstalc l is bigger than stop l, no need stop
      stop = false;
    }
    AWARN << "crosswalk_id[" << crosswalk_id << "] stop_deceleration["
          << stop_deceleration << "]";
  }
}

return stop;
```

3. 如果需要停车，障碍物不在当前车道，人行道到车的距离大于阈值（40）

3.1 如果障碍物速度小于最大停止速度（0.3），判断当前 obstacle id 是否在 crosswalk_stop_timer 中，不在则插入。在的花，判断停止时间三否大于 config.crosswalk.stop_timeout,大于则不需要停车

```
//step 3.3, if need stop, obstacle is not on current lane and obstacle to adv distance is smaller than 40,
if (stop && !is_on_lane &&
    crosswalk_overlap->start_s - adc_front_edge_s <=
        kStartWatchTimerDistance) {
  // check on stop timer for static pedestrians/bicycles
  // if NOT on_lane ahead of adc
  const double kMaxStopSpeed = 0.3;
  auto obstacle_speed = std::hypot(perception_obstacle.velocity().x(),
                                   perception_obstacle.velocity().y());
  //step 3.4
  if (obstacle_speed <= kMaxStopSpeed) {
    //if osbtacle speed is smaller than stop speed, and obstacle id isn't in stop_timer, insert it
    if (crosswalk_stop_timer[crosswalk_id].count(obstacle_id) < 1) {
      // add timestamp
      ADEBUG << "add timestamp: obstacle_id[" << obstacle_id
             << "] timestamp[" << Clock::NowInSeconds()
             << "]";
      crosswalk_stop_timer[crosswalk_id].insert(
          {obstacle_id, Clock::NowInSeconds()});
    } else {
      //else no need stop
      double stop_time = Clock::NowInSeconds() -
                         crosswalk_stop_timer[crosswalk_id][obstacle_id];
      ADEBUG << "stop_time: obstacle_id[" << obstacle_id << "] stop_time["
             << stop_time << "]";
      if (stop_time >= config_.crosswalk().stop_timeout()) {
        stop = false;
      }
    }
  }
}
```

4. 如果需要停车，将 obstacle_id 放入 pedestrains 中

```
  //step 3.5  if  need stop, push back pedestrians
  if (stop) {
    pedestrians.push_back(obstacle_id);
    ADEBUG << "wait for: obstacle_id[" << obstacle_id << "] type["
           << obstacle_type_name << "] crosswalk_id[" << crosswalk_id
           << "]";
  } else {
    ADEBUG << "skip: obstacle_id[" << obstacle_id << "] type["
           << obstacle_type_name << "] crosswalk_id[" << crosswalk_id
           << "]";
  }
```

5. pedestrains 非空，则将 crosswalk_overlap 和 pedestrains 放入 corsswalks_to_stop

```
if (!pedestrians.empty()) {
  //step 3.6 if not empty, put crosswalk_overlap and pedestrians into crosswalks_to_stop
  crosswalks_to_stop.emplace_back(crosswalk_overlap, pedestrians);
  ADEBUG << "crosswalk_id[" << crosswalk_id << "] STOP";
}
```

6. 作停止决策

```
double min_s = std::numeric_limits<double>::max();
hdmap::PathOverlap* firsts_crosswalk_to_stop = nullptr;
for (auto crosswalk_to_stop : crosswalks_to_stop) {
  //step 3.7  build stop decision
  const auto* crosswalk_overlap = crosswalk_to_stop.first;
  ADEBUG << "BuildStopDecision: crosswalk[" << crosswalk_overlap->object_id
         << "] start_s[" << crosswalk_overlap->start_s << "]";
  std::string virtual_obstacle_id =
      CROSSWALK_VO_ID_PREFIX + crosswalk_overlap->object_id;
  util::BuildStopDecision(virtual_obstacle_id, crosswalk_overlap->start_s,
                          config_.crosswalk().stop_distance(),
                          StopReasonCode::STOP_REASON_CROSSWALK,
                          crosswalk_to_stop.second,
                          TrafficRuleConfig::RuleId_Name(config_.rule_id()),
                          frame, reference_line_info);

  //get first crosswalk to stop and min s
  if (crosswalk_to_stop.first->start_s < min_s) {
    firsts_crosswalk_to_stop =
        const_cast<PathOverlap*>(crosswalk_to_stop.first);
    min_s = crosswalk_to_stop.first->start_s;
  }
}

if (firsts_crosswalk_to_stop) {
```

7. 更新 crosswalk 状态

```
if (firsts_crosswalk_to_stop) {
  //step 3.8 update CrosswalkStatus
  std::string crosswalk = firsts_crosswalk_to_stop->object_id;
  mutable_crosswalk_status->set_crosswalk_id(crosswalk);
  mutable_crosswalk_status->clear_stop_time();
  for (const auto& timer : crosswalk_stop_timer[crosswalk]) {
    auto* stop_time = mutable_crosswalk_status->add_stop_time();
    stop_time->set_obstacle_id(timer.first);
    stop_time->set_stop_timestamp_sec(timer.second);
    ADEBUG << "UPDATE stop_time: id[" << crosswalk << "] obstacle_id["
           << timer.first << "] stop_timestamp[" << timer.second << "]";
  }

  // update CrosswalkStatus.finished_crosswalk
  mutable_crosswalk_status->clear_finished_crosswalk();
  for (auto crosswalk_overlap : crosswalk_overlaps_) {
    if (crosswalk_overlap->start_s < firsts_crosswalk_to_stop->start_s) {
      mutable_crosswalk_status->add_finished_crosswalk(
          crosswalk_overlap->object_id);
      ADEBUG << "UPDATE finished_crosswalk: " << crosswalk_overlap->object_id;
    }
  }
}

ADEBUG << "crosswalk_status: " << mutable_crosswalk_status->DebugString();
```

9 KeepClear

禁停区分为两类，第一类是传统的禁停区，第二类是交叉路口。对于禁停区的处理和对人行横道上障碍物构建虚拟墙很相似。具体做法是在参考线上构建一块禁停区，从纵向的 start_s 到 end_s(这里的 start_s 和 end_s 是禁停区 start_s 和 end_s 在参考线上的投影点)。禁停区宽度在配置文件中设定(4 米)。

算法逻辑

- 检查 frame 和 reference line info 非空
- 检查自车是否已经驶入禁停区,如果是,则忽略

```
// step 1 check if has driven into no-stop zone
const double adc_front_edge_s = reference_line_info->AdcSlBoundary().end_s();
if (adc_front_edge_s - keep_clear_start_s >
    config_.keep_clear().min_pass_s_distance()) {
  ADEBUG << "adc inside keep_clear zone[" << virtual_obstacle_id << "] s["
         << keep_clear_start_s << ", " << keep_clear_end_s
         << "] adc_front_edge_s[" << adc_front_edge_s
         << "]. skip this keep clear zone";
  return false;
}
```

- 对于还未经过的禁停区,生成虚拟障碍物,并封装到 path_obstacle 中

```
// create virtual static obstacle
auto* obstacle =
    frame->CreateStaticObstacle(reference_line_info, virtual_obstacle_id,
                                keep_clear_start_s, keep_clear_end_s);
if (!obstacle) {
  AERROR << "Failed to create obstacle [" << virtual_obstacle_id << "]";
  return false;
}
auto* path_obstacle = reference_line_info->AddObstacle(obstacle);
if (!path_obstacle) {
  AERROR << "Failed to create path_obstacle: " << virtual_obstacle_id;
  return false;
}
path_obstacle->SetReferenceLineStBoundaryType(
    STBoundary::BoundaryType::KEEP_CLEAR);
```

交叉口

与传统禁停区相比,对交叉口进行处理时,需要根据交叉口的类型来计算交叉口开始的 s 值(pnc_junction_start_s),用来生成虚拟障碍物的标定框