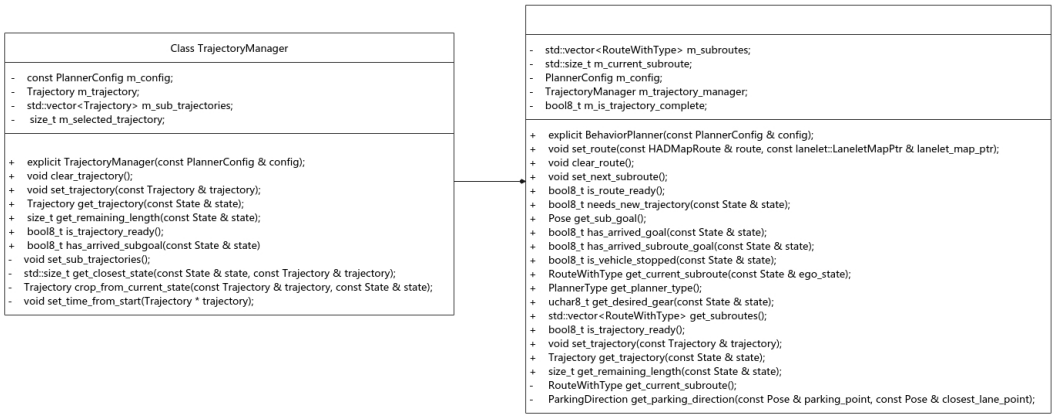
Behavior Planner



1. Trajectory Manager

```
void TrajectoryManager::clear_trajectory()
```

```
void TrajectoryManager::set_trajectory(const Trajectory & trajectory)
```

- clear_trajectory 是为了清空轨迹点，set_trajectory 实现了将一条轨迹拷贝给类成员的功能。

```
bool8_t TrajectoryManager::is_trajectory_ready()
{
  return !m_sub_trajectories.empty();
}
```

- 判断轨迹是否准备好->不为空。

```
std::size_t TrajectoryManager::get_closest_state(
  const State & current_state,
  const Trajectory & trajectory)
{
  const auto distance_from_current_state =
    [this, &current_state](const TrajectoryPoint & other_state) {
      const auto s1 = current_state.state, s2 = other_state;
      const auto distance = norm_2d(minus_2d(s1, s2));
      const auto angle_diff =
        std::abs(::motion::motion_common::to_angle(s1.pose.orientation - s2.pose.orientat
      return distance + m_config.heading_weight * static_cast<float32_t>(angle_diff);
    };
```

➢ 这个函数会返回当前距离 trajectory 最近的 state。distance_from_current_state 是一个内联函数，计算两个 state 之间的距离。

```
 const auto comparison_function =
   [&distance_from_current_state](const TrajectoryPoint & one, const TrajectoryPoint & t
   {return distance_from_current_state(one) < distance_from_current_state(two);};

 //NOTE:: std::min_element will return min element iterator in current compare rule,
 const auto minimum_index_iterator =
   std::min_element(
   std::begin(trajectory.points), std::end(trajectory.points),
   comparison_function);

 //NOTE:: std::distance will return num of elment from first iterator to second iterator
 auto minimum_idx = std::distance(std::begin(trajectory.points), minimum_index_iterator)

 return static_cast<std::size_t>(minimum_idx);
```

后半部分通过自定义的比较规则，从 trajectory points 中选出距离最近点的下标。std::min_element 返回该规则下最小值的 iterator， std::distance 返回当前 iterator 距离 begin iterator 的元素个数。

```
size_t TrajectoryManager::get_remaining_length(const State & state)
{
  // remaining length of current selected sub trajectory
  const auto & current_trajectory = m_sub_trajectories.at(m_selected_trajectory);
  const size_t closest_index = get_closest_state(state, current_trajectory);
  size_t remaining_length = current_trajectory.points.size() - closest_index;

  // remaining length including rest of sub trajectories
  for (size_t i = m_selected_trajectory + 1; i < m_sub_trajectories.size(); i++) {
    remaining_length += m_sub_trajectories.at(i).points.size();
  }

  return remaining_length;
}
```

➢ 根据当前选中轨迹和状态，匹配到当前最近的下标，返回当前轨迹到最终轨迹的元素的总个数。

```cpp
Trajectory TrajectoryManager::crop_from_current_state(
  const Trajectory & trajectory,
  const State & state)
{
  if (trajectory.points.empty()) {return trajectory;}
  auto index = get_closest_state(state, trajectory);

  // we always want trajectory to start from front of vehicle so increment index
  if (index + 1 < trajectory.points.size()) {
    index += 1;
  }

  Trajectory output;
  output.header = trajectory.header;
  auto current_state = state.state;
  current_state.longitudinal_velocity_mps = trajectory.points.at(index).longitudinal_velocity_mps;
  output.points.push_back(current_state);
  for (size_t i = index; i < trajectory.points.size(); i++) {
    output.points.push_back(trajectory.points.at(i));
  }
  return output;
}
```

- 根据当前 state 和轨迹匹配到最近的 index，同时返回剩余轨迹

```cpp
void TrajectoryManager::set_time_from_start(Trajectory * trajectory)
{
  if (trajectory->points.empty()) {
    return;
  }

  float32_t t = 0.0;

  // special operation for first point
  auto & first_point = trajectory->points.at(0);
  first_point.time_from_start.sec = 0;
  first_point.time_from_start.nanosec = 0;

  for (std::size_t i = 1; i < trajectory->points.size(); ++i) {
    auto & p0 = trajectory->points[i - 1];
    auto & p1 = trajectory->points[i];
    auto v = 0.5f * (p0.longitudinal_velocity_mps + p1.longitudinal_velocity_mps);
    t += norm_2d(minus_2d(p0, p1)) / std::max(std::fabs(v), 0.5f);
    float32_t t_s = 0;

    //NOTE:: std::modf is used to divide fraction and int, t_s will be int and t_ns will be fraction
    float32_t t_ns = std::modf(t, &t_s) * 1.0e9f;
    trajectory->points[i].time_from_start.sec = static_cast<int32_t>(t_s);
    trajectory->points[i].time_from_start.nanosec = static_cast<uint32_t>(t_ns);
  }
}
```

- 这是一个时间匹配函数，初始时间设为 0。首先根据轨迹点计算点之间的距离，取速度为两点速度平均值和速度绝对值的最大值，计算时间，sec 部分是 t 的整数部分，nanosec 是小数部分。std::modf 则是一个分离整数和小数的函数。

```cpp
Trajectory TrajectoryManager::get_trajectory(const State & state)
{
  // select new sub_trajectory when vehicle is at stop
  if (std::abs(state.state.longitudinal_velocity_mps) < m_config.stop_velocity_thresh) {
    const auto & last_point = m_sub_trajectories.at(m_selected_trajectory).points.back();
    const auto distance = norm_2d(minus_2d(last_point, state.state));

    // increment index to select new sub_trajectory if vehicle has arrived the end of sub_trajectory
    if (distance < m_config.goal_distance_thresh) {
      m_selected_trajectory++;
      m_selected_trajectory = std::min(m_selected_trajectory, m_sub_trajectories.size() - 1);
    }
  }

  // TODO(mitsudome-r) implement trajectory refine functions if needed to integrate with controller
  const auto & input = m_sub_trajectories.at(m_selected_trajectory);
  auto output = crop_from_current_state(input, state);
  set_time_from_start(&output);
  return output;
}
```

- 如果汽车停下来了，则需要重新选择轨迹（index），首先判断当前是否已经走完了子轨迹，走完了下一条，否则继续走。重新选择轨迹之后，根据当前 state 返回最新轨迹（trajectory），并进行时间匹配。

```
void TrajectoryManager::set_sub_trajectories()
{
  // return sign with hysterysis buffer

  //Note: following inline function will show the sign of velocity is positive or negative

  const auto is_positive = [](const TrajectoryPoint & pt) {
      // using epsilon instead to ensure change in sign.
      static bool8_t is_prev_positive = true;
      bool8_t is_positive;
      if (is_prev_positive) {
        is_positive = pt.longitudinal_velocity_mps > -std::numeric_limits<float32_t>::epsilon();
      } else {
        is_positive = pt.longitudinal_velocity_mps > std::numeric_limits<float32_t>::epsilon();
      }
      is_prev_positive = is_positive;
      return is_positive;
    };

  if (m_trajectory.points.empty()) {
    m_sub_trajectories.push_back(m_trajectory);
    return;
  }
```

- 这个内联函数返回当前轨迹点的速度方向，

```
  Trajectory sub_trajectory;
  sub_trajectory.header = m_trajectory.header;

  //step 1
  //logic: put all points into sub_trajectory
  //       if sign is changed and sub_trajectory is not empty, put it in m_sub_trajectory
  //        because when sign is changed, it means the vehicle's direction is changed
  auto prev_sign = is_positive(m_trajectory.points.front());
  for (auto & pt : m_trajectory.points) {
    const auto sign = is_positive(pt);
    if (prev_sign != sign && !sub_trajectory.points.empty()) {
      m_sub_trajectories.push_back(sub_trajectory);
      sub_trajectory.points.clear();
    }
    sub_trajectory.points.push_back(pt);
    prev_sign = sign;
  }
  //if all points sign is same, finally put it in m_sub_trajectory
  if (!sub_trajectory.points.empty()) {
    m_sub_trajectories.push_back(sub_trajectory);
  }
```

这个函数会将 m_trajectory 中的点 push 到 sub_trajectory 中，当速度方向发生改变时并且 sub_trajectory 不为空时，将 sub_trajectory 放入 m_sub_trajectories 中。默认是所有点都被放入 sub_trajectory 中，当 sub_trajectory 不为空时，将 sub_trajectory 放入 m_sub_trajectories 中。

2. Behavior Planner

```
PlannerType get_planner_type_from_primitive(
  const MapPrimitive & map_primitive)
{
  static const std::unordered_map<std::string, PlannerType> primitive_to_planner_type({
        {"parking", PlannerType::PARKING},
        {"drivable_area", PlannerType::PARKING},
        {"lane", PlannerType::LANE}});

  const std::string & primitive_type = map_primitive.primitive_type;

  if (primitive_to_planner_type.find(primitive_type) != primitive_to_planner_type.end()) {
    return primitive_to_planner_type.at(primitive_type);
  } else {
    return PlannerType::UNKNOWN;
  }
}
```

➢ 根据 MapPrimitive 信息返回当前的 plannertype，默认 UNKNOWN。

```
bool8_t BehaviorPlanner::is_route_ready()
{
  return !m_subroutes.empty();
}

void BehaviorPlanner::set_next_subroute()
{
  m_current_subroute = std::min(m_current_subroute + 1, m_subroutes.size() - 1);
}
```

➢ 这两个函数主要：返回 route 是否 ready 以及设置下一条子路径（index）。

```
RouteWithType BehaviorPlanner::get_current_subroute(const State & ego_state)
{
  if (!is_route_ready()) {
    return RouteWithType();
  }
  auto updated_subroute = m_subroutes.at(m_current_subroute);
  updated_subroute.route.header = ego_state.header;
  if (updated_subroute.planner_type == PlannerType::LANE) {
    updated_subroute.route.start_pose = ego_state.state.pose;
  }
  return updated_subroute;
}
```

➢ 返回一个带类型的路径，根据当前路径下标更新路径，如果类型为 LANE,将路径起点更新为汽车位置。

```
RouteWithType BehaviorPlanner::get_current_subroute()
{
  if (!is_route_ready()) {
    return RouteWithType();
  }
  return m_subroutes.at(m_current_subroute);
}
```

- 无参数时，返回当前下标返回的当前路径

```
ParkingDirection BehaviorPlanner::get_parking_direction(
  const Pose & parking_pose,
  const Pose & closest_lane_pose)
{
  // Calculate angle from parking pose to closest lane pose
  const auto direction_vector = minus_2d(closest_lane_pose.position, parking_pose.position);
  const auto diff_angle = std::atan2(direction_vector.y, direction_vector.x);

  // Get heading angle of parking pose
  const auto heading_angle = motion::motion_common::to_angle(parking_pose.orientation);

  // If absolute difference between angles is gt pi/2, the parking orientation is head-in
  if (
    static_cast<float32_t>(std::fabs(diff_angle - heading_angle)) > autoware::common::types::PI_2)
  {
    return ParkingDirection::HEAD_IN;
  } else {
    return ParkingDirection::TOE_IN;
  }
}
```

- 根据停车时的姿态和最近 lane 的姿态信息计算方向向量，根据 方向向量计算出一个 diff_angle，在根据停车时的姿态的朝向角计算出朝向角。如果两者相差大于 pi/2，则停车时的姿态是头进，否则是尾进。

```
bool8_t BehaviorPlanner::has_arrived_goal(const State & state)
{
  if (!is_route_ready()) {
    return false;
  }

  const auto satisfy_velocity_condition = is_vehicle_stopped(state);

  const auto & route = m_subroutes.back().route;
  const auto & state_route_pose = state.state.pose;
  const auto distance = norm_2d(minus_2d(route.goal_pose.position, state_route_pose.position));
  const auto satisfy_distance_condition = distance < m_config.goal_distance_thresh;

  // NOTE:: arrive goal must satisfy two condition: velocity condition and distance condition

  return satisfy_velocity_condition && satisfy_distance_condition;
}
```

- 在速度和距离上都满足到达终点的条件，则认为是到达了终点。、

```
bool8_t BehaviorPlanner::needs_new_trajectory(const State & ego_state)
{
  // we need trajectory if we don't have trajectory yet
  if (!is_trajectory_ready()) {
    return true;
  }

  // we need trajectory if trajectory does not reach to subroute goal
  const auto length = get_remaining_length(ego_state);
  const auto is_close_to_end = length < static_cast<size_t>(Trajectory::CAPACITY / 2);
  return is_close_to_end && !m_is_trajectory_complete;
}
```

根据轨迹是否为空，以及是否到达终点和 trajectory 完成来判断是否需要新轨迹。

```
geometry_msgs::msg::Pose get_closest_pose_on_lane(
  const Pose & point, const int64_t lane_id,
  const lanelet::LaneletMapPtr & lanelet_map_ptr, const float32_t offset)
{
  Pose closest_pose_on_lane;

  const auto lanelet = lanelet_map_ptr->laneletLayer.get(lane_id);

  //  we do planning based on 2D plane
  const auto & centerline = lanelet::utils::to2D(lanelet.centerline());

  if (centerline.size() < 2) {
    std::cerr << "Could not get closest pose on Lane due to invalid centerline of lane " <<
      lanelet.id() << std::endl;
    return closest_pose_on_lane;
  }
```

- 根据 lane_id，在 lanelet_map 中返回 lanelet, centerline.size 不大于 2，返回一个空的 pose，

```
// first find the closest point on line and fine length along lane
float32_t min_distance = std::numeric_limits<float32_t>::max();
float32_t length_along_line = 0.0f, accumulated_length = 0.0f;
for (size_t i = 1; i < centerline.size(); i++) {
  const auto prev_pt = convert_to_route_pose(centerline[i - 1]);
  const auto current_pt = convert_to_route_pose(centerline[i]);

  //this function will first return a vector which matched by (p,q,r),it will return (p->q)(p->r)/||p->q||
  //then it will calculate L2 value of difference vector of matched vector with r
  const auto distance = point_line_segment_distance_2d(
    prev_pt.position, current_pt.position, point.position);


  if (distance < min_distance) {
    min_distance = distance;
    const auto point_on_lane = closest_segment_point_2d(
      prev_pt.position, current_pt.position, point.position);

    length_along_line = accumulated_length + norm_2d(minus_2d(prev_pt.position, point_on_lane));
  }
  accumulated_length += norm_2d(minus_2d(prev_pt.position, current_pt.position));
}
```

接下来，根据 centerline（轨迹点集）上的每两个点（p，q）以及当前点 r 来计算：
首先计算出，pr 在 pq 方向上的投影，并计算出新的投影点 r’。
再计算出 r’和 r 的差向量的二范数。
最后计算沿着 centerline 的距离累加（根据匹配的点 r’）以及累积距离（根据 r）

```
  // we find a point from line length with offset
  float32_t length_with_offset = length_along_line + offset;
  length_with_offset = std::max(length_with_offset, 0.0f);
  length_with_offset = std::min(length_with_offset, accumulated_length);
```

取带 offset 的轨迹长度,其在 0 到 accumulated_length 之间。

```
accumulated_length = 0.0f;
for (size_t i = 1; i < centerline.size(); i++) {
  const auto prev_pt = convert_to_route_pose(centerline[i - 1]);
  const auto current_pt = convert_to_route_pose(centerline[i]);
  const auto distance = norm_2d(minus_2d(prev_pt.position, current_pt.position));
  if (accumulated_length + distance >= length_with_offset) {
    const auto direction_vector = minus_2d(current_pt.position, prev_pt.position);
    const auto ratio = (length_with_offset - accumulated_length) / distance;
    closest_pose_on_lane.position = plus_2d(prev_pt.position, times_2d(direction_vector, ratio));
    const float32_t angle = std::atan2(
      static_cast<float32_t>(direction_vector.y), static_cast<float32_t>(direction_vector.x));
    closest_pose_on_lane.orientation = motion::motion_common::from_angle(angle);
  }
  accumulated_length += distance;
}

return closest_pose_on_lane;
```

取 accumulate_length 为 0，
从 centerline 头部开始计算 accumulate_length。如果 accumulate_length 大于带 offset 的 length，则表明轨迹点应该匹配到这一段，根据要当前点的坐标以及上一个点的坐标计算出这一段的 distance（这两点都在 centerline 上），再根据带 offset 的 length 和 accumulated_length 的插值除以 distance，计算要被匹配的点在这一段上的占比。根据占比匹配出点，并计算其 angle 信息。

```
void BehaviorPlanner::set_route(
  const HADMapRoute & route,
  const lanelet::LaneletMapPtr & lanelet_map_ptr)
{
  // create subroutes from given global route
  clear_route();

  // initialization for before loop
  RouteWithType subroute;
  subroute.route.start_pose = route.start_pose;
  auto prev_type = PlannerType::UNKNOWN;
  autoware_auto_mapping_msgs::msg::HADMapSegment prev_segment;

  if (!route.segments.empty()) {
    const auto & first_segment = route.segments.front();
    prev_type = get_planner_type_from_primitive(first_segment.primitives.front());
    subroute.planner_type = prev_type;
    prev_segment = first_segment;
  }
```

- 首先清空路径信息。
  如果输入路径不为空，将第一段设为该路径的第一段，类型也保持一致。

```
  size_t i = 0;
  for (const auto & segment : route.segments) {
    const auto & primitive = segment.primitives.front();
    const auto & type = get_planner_type_from_primitive(primitive);

    auto new_segment = decltype(segment) {};
    new_segment.preferred_primitive_id = primitive.id;
    new_segment.primitives.push_back(primitive);
    subroute.route.segments.push_back(new_segment);
```

对于 route 上的每一段，记录其 primitive 和 type 信息，创建空的 new_segment 并将其 push 到 subroute.route.segments 中。

```cpp
if (prev_type == PlannerType::PARKING && type == PlannerType::LANE) {
  // Determine parking direction and set offset direction accordingly
  float32_t route_offset = m_config.subroute_goal_offset_parking2lane;
  const auto closest_lane_pose = get_closest_pose_on_lane(
    subroute.route.start_pose,
    primitive.id, lanelet_map_ptr, 0.0f);
  const auto parking_dir = get_parking_direction(
    subroute.route.start_pose, closest_lane_pose);

  if (parking_dir == ParkingDirection::HEAD_IN) {
    // Add extra distance for vehicle length
    route_offset = -(route_offset + m_config.cg_to_vehicle_center);
  }

  // create parking subroute
  // set goal to closest poin on lane from starting pose
  subroute.route.goal_pose = get_closest_pose_on_lane(
    subroute.route.start_pose,
    primitive.id, lanelet_map_ptr, route_offset);
  m_subroutes.push_back(subroute);

  // reinitialize for next subroute
  subroute.planner_type = type;
  subroute.route.segments.clear();
  subroute.route.segments.push_back(segment);
  subroute.route.start_pose = subroute.route.goal_pose;
}
```

如何 planner 的 type 发生了改变：

并且之前的 type 是 parking，而现在是 lane，那么根据 start_pose 和地图匹配出一个 close_lane_pose，再确定停车时的朝向。

如果汽车朝向是 HEAD_IN,计算 headin 方向的 offset。在根据 offset 匹配一个新的点。并将这个 subroute 放入 m_subroutes 中。(从 parking 状态到 LANE 状态 subroute)

在重新初始化，以用于下一段的 subroute.

```
  if (prev_type == PlannerType::LANE && type == PlannerType::PARKING) {
    // Determine parking direction and set offset direction accordingly
    float32_t route_offset = m_config.subroute_goal_offset_lane2parking;
    const auto closest_lane_pose = get_closest_pose_on_lane(
      route.goal_pose, prev_segment.primitives.front().id,
      lanelet_map_ptr, 0.0f);
    const auto parking_dir = get_parking_direction(
      route.goal_pose, closest_lane_pose);

    if (parking_dir == ParkingDirection::HEAD_IN) {
      // Add extra distance for vehicle length
      route_offset = -(route_offset + m_config.cg_to_vehicle_center);
    }

    // Currently, we assume that final goal is close to lane.
    subroute.route.goal_pose = get_closest_pose_on_lane(
      route.goal_pose, prev_segment.primitives.front().id,
      lanelet_map_ptr, route_offset);
    m_subroutes.push_back(subroute);

    // reinitialize for next subroute
    subroute.planner_type = type;
    subroute.route.segments.clear();
    subroute.route.segments.push_back(prev_segment);
    subroute.route.segments.push_back(segment);
    subroute.route.start_pose = subroute.route.goal_pose;
  }
```

如果之前的 type 是 LANE,而现在我们要 PARKING，

在 lane 上匹配到最终 goal_pose 的匹配点，计算汽车朝向。

如果汽车朝向是 HEAD_IN,计算 headin 方向的 offset。根据朝向再匹配一个点，将这个 subroute 放入 m_subroute.在重新初始化，以用于下一段的 subroute.

```
  // add final subroute
  // note that prev_type is actually type of final primitive after the loop is done
  if (prev_type == PlannerType::LANE) {
    subroute.route.goal_pose = get_closest_pose_on_lane(
      route.goal_pose,
      prev_segment.primitives.front().id, lanelet_map_ptr, 0.0f);
  } else {
    subroute.route.goal_pose = route.goal_pose;
  }
  m_subroutes.push_back(subroute);
}
```

最后匹配最终的 subroute 的 goal_pose 即可。