该decider的大致作用是对记录ADC此时的换道状态，并不决定ADC是否进行lanechange。车的换道状态信息存于 injector\_->planning\_context()->mutable\_planning\_status()->mutable\_change\_lane()中

1：它的更新状态函数：

void LaneChangeDecider::UpdateStatus(double timestamp,ChangeLaneStatus::Status status\_code, const std::string& path\_id)

{

auto\* lane\_change\_status = injector\_->planning\_context()

->mutable\_planning\_status()

->mutable\_change\_lane();

lane\_change\_status->set\_timestamp(timestamp);

lane\_change\_status->set\_path\_id(path\_id);

lane\_change\_status->set\_status(status\_code);

}

2:PrioritizeChangeLane

//当is\_prioritize\_change\_lane为true,则遍历存储referenceLineInfo的链表，把

//当前车辆不位于的的那条（俗称换到参考线）放到链表的第一个位置

//当is\_prioritize\_change\_lane为false,则遍历存储referenceLineInfo的链表，把

//当前车辆所位于的的那条放到链表的第一个位置

//首先获取第一条参考线的迭代器，然后遍历所有的参考线

//注意，可变车道为按迭代器的顺序求取，一旦发现可变车道，即推出循环。

void LaneChangeDecider::PrioritizeChangeLane(

const bool is\_prioritize\_change\_lane,

std::list<ReferenceLineInfo>\* reference\_line\_info) const {

if (reference\_line\_info->empty()) {

AERROR << "Reference line info empty";

return;

}

const auto& lane\_change\_decider\_config = config\_.lane\_change\_decider\_config();

// 暂时禁用参考线的顺序更改

if (!lane\_change\_decider\_config.enable\_prioritize\_change\_lane()) {

return;

}

auto iter = reference\_line\_info->begin();

while (iter != reference\_line\_info->end()) {

ADEBUG << "iter->IsChangeLanePath(): " << iter->IsChangeLanePath();

/\* is\_prioritize\_change\_lane == true: prioritize change\_lane\_reference\_line

is\_prioritize\_change\_lane == false: prioritize

non\_change\_lane\_reference\_line(对应红色注释) \*/

if ((is\_prioritize\_change\_lane && iter->IsChangeLanePath()) ||

(!is\_prioritize\_change\_lane && !iter->IsChangeLanePath())) {

ADEBUG << "is\_prioritize\_change\_lane: " << is\_prioritize\_change\_lane;

ADEBUG << "iter->IsChangeLanePath(): " << iter->IsChangeLanePath();

break;

}

++iter;

}

reference\_line\_info->splice(reference\_line\_info->begin(),

\*reference\_line\_info, iter);

ADEBUG << "reference\_line\_info->IsChangeLanePath(): "

<< reference\_line\_info->begin()->IsChangeLanePath();

}

3：IsClearToChangeLane

//调选出位于该referenceline(参考线)上的动态障碍物，结合障碍物的运动方向和车的运动方向，

//检查每个障碍物与车的前后距离，看是否都满足安全阈值。只要有一个动态障碍物不满足条件则该referenceline就不满足换道条件。

prev\_status->set\_is\_clear\_to\_change\_lane(false)

bool LaneChangeDecider::IsClearToChangeLane(

ReferenceLineInfo\* reference\_line\_info) {

double ego\_start\_s = reference\_line\_info->AdcSlBoundary().start\_s();

double ego\_end\_s = reference\_line\_info->AdcSlBoundary().end\_s();

double ego\_v =

std::abs(reference\_line\_info->vehicle\_state().linear\_velocity());

for (const auto\* obstacle :

reference\_line\_info->path\_decision()->obstacles().Items()) {

// 排除虚拟障碍物和静态障碍物

if (obstacle->IsVirtual() || obstacle->IsStatic()) {

ADEBUG << "skip one virtual or static obstacle";

continue;

}

double start\_s = std::numeric\_limits<double>::max();

double end\_s = -std::numeric\_limits<double>::max();

double start\_l = std::numeric\_limits<double>::max();

double end\_l = -std::numeric\_limits<double>::max();

for (const auto& p : obstacle->PerceptionPolygon().points()) {

SLPoint sl\_point;

reference\_line\_info->reference\_line().XYToSL(p, &sl\_point);

start\_s = std::fmin(start\_s, sl\_point.s());

end\_s = std::fmax(end\_s, sl\_point.s());

start\_l = std::fmin(start\_l, sl\_point.l());

end\_l = std::fmax(end\_l, sl\_point.l());

}

if (reference\_line\_info->IsChangeLanePath()) {

double left\_width(0), right\_width(0);

reference\_line\_info->mutable\_reference\_line()->GetLaneWidth(

(start\_s + end\_s) \* 0.5, &left\_width, &right\_width);

//只考虑在reference\_line\_info（参考线）所在的车道的障碍物

if (end\_l < -right\_width || start\_l > left\_width) {

continue;

}

}

// Raw estimation on whether same direction with ADC or not based on

// prediction trajectory

// 基于预测轨迹的与ADC方向是否相同的原始估计

bool same\_direction = true;

if (obstacle->HasTrajectory()) {

double obstacle\_moving\_direction =

obstacle->Trajectory().trajectory\_point(0).path\_point().theta();

const auto& vehicle\_state = reference\_line\_info->vehicle\_state();

double vehicle\_moving\_direction = vehicle\_state.heading();

if (vehicle\_state.gear() == canbus::Chassis::GEAR\_REVERSE) {

vehicle\_moving\_direction =

common::math::NormalizeAngle(vehicle\_moving\_direction + M\_PI);

}

double heading\_difference = std::abs(common::math::NormalizeAngle(

obstacle\_moving\_direction - vehicle\_moving\_direction));

same\_direction = heading\_difference < (M\_PI / 2.0);

}

// 计算所有的安全域值

static constexpr double kSafeTimeOnSameDirection = 3.0;

//相同方向的安全时间

static constexpr double kSafeTimeOnOppositeDirection = 5.0;

//相反方向的

static constexpr double kForwardMinSafeDistanceOnSameDirection = 10.0;

//想同方向的前方最小安全距离

static constexpr double kBackwardMinSafeDistanceOnSameDirection = 10.0;

//想同方向的后方最小安全距离

static constexpr double kForwardMinSafeDistanceOnOppositeDirection = 50.0;

//想反方向的前方最小安全距离

static constexpr double kBackwardMinSafeDistanceOnOppositeDirection = 1.0;

//想反方向的后方最小安全距离

static constexpr double kDistanceBuffer = 0.5;

//缓冲距离

double kForwardSafeDistance = 0.0;

//前方安全距离

double kBackwardSafeDistance = 0.0;

//后方

if (same\_direction) {

kForwardSafeDistance =

std::fmax(kForwardMinSafeDistanceOnSameDirection,

(ego\_v - obstacle->speed()) \* kSafeTimeOnSameDirection);

kBackwardSafeDistance =

std::fmax(kBackwardMinSafeDistanceOnSameDirection,

(obstacle->speed() - ego\_v) \* kSafeTimeOnSameDirection);

} else {

kForwardSafeDistance =

std::fmax(kForwardMinSafeDistanceOnOppositeDirection,

(ego\_v + obstacle->speed()) \* kSafeTimeOnOppositeDirection);

kBackwardSafeDistance = kBackwardMinSafeDistanceOnOppositeDirection;

}

/\*

根据前面计算的阈值，判断障碍物是否安全，采用的是滞回区间的方法，如果障碍物小于安全距离，laneChangeBlocking为true。

如果障碍物大于安全距离，laneChangeBlocking为false。通过滞回区间进行滤波。

一旦发现有block的障碍物，函数就返回，就认为该Reference 非clear(安全)。

\*/

if (HysteresisFilter(ego\_start\_s - end\_s, kBackwardSafeDistance,

kDistanceBuffer, obstacle->IsLaneChangeBlocking()) &&

HysteresisFilter(start\_s - ego\_end\_s, kForwardSafeDistance,

kDistanceBuffer, obstacle->IsLaneChangeBlocking())) {

reference\_line\_info->path\_decision()

->Find(obstacle->Id())

->SetLaneChangeBlocking(true);

ADEBUG << "Lane Change is blocked by obstacle" << obstacle->Id();

return false;

} else {

reference\_line\_info->path\_decision()

->Find(obstacle->Id())

->SetLaneChangeBlocking(false);

}

}

return true;

}

lane\_change\_decider的代码表示：

在\modules\planning\tasks\deciders\lane\_change\_decider目录下，

且process()是处理lane\_change\_decider的主要处理逻辑函数：

// 添加了一个伪参数以在ExecuteTaskOnReferenceLine中启用此任务

// 这个ExecuteTaskOnReferenceLine在\modules\planning\scenarios\stage.cc目录下有具体内容 参考线在Frame类中可以得到

Status LaneChangeDecider::Process(Frame\* frame, ReferenceLineInfo\* const current\_reference\_line\_info)

{

// Sanity checks.

CHECK\_NOTNULL(frame);//检查frame是否为空

const auto& lane\_change\_decider\_config = config\_.lane\_change\_decider\_config();

// 通过frame拿到车辆此时所在的区域参考线，reference\_line\_info即表示参考//线。

std::list<ReferenceLineInfo>\* reference\_line\_info =

frame->mutable\_reference\_line\_info();

// 如果没有参考线则提示错误

if (reference\_line\_info->empty()) {

const std::string msg = "Reference lines empty.";

AERROR << msg;

return Status(ErrorCode::PLANNING\_ERROR, msg);

}

// 是否进行强制换道，如果是进入此函数，（详情见2）

if (lane\_change\_decider\_config.reckless\_change\_lane()) {

PrioritizeChangeLane(true, reference\_line\_info);

return Status::OK();

}

auto\* prev\_status = injector\_->planning\_context()

->mutable\_planning\_status()

->mutable\_change\_lane();

double now = Clock::NowInSeconds();

// 默认设置false

prev\_status->set\_is\_clear\_to\_change\_lane(false);

//此处判断传进来的referenceLineinfo是否是变道参考线，如果是则通过

// IsChangeLanePath():判断是否是可变车道，如果车不在车道片段上，则该车道为可变道车道。

if (current\_reference\_line\_info->IsChangeLanePath()) {

//IsClearToChangeLane()检查该参考线是否满足变道条件，

//IsClearToChangeLane只考虑传入的参考线上的动态障碍物，不考虑虚的和静态的障碍物。详情见3；

prev\_status->set\_is\_clear\_to\_change\_lane(

IsClearToChangeLane(current\_reference\_line\_info));

}

//头次进入task，车道换道状态应该为空，默认设置为换道结束状态

if (!prev\_status->has\_status()) {

UpdateStatus(now, ChangeLaneStatus::CHANGE\_LANE\_FINISHED,

GetCurrentPathId(\*reference\_line\_info));

prev\_status->set\_last\_succeed\_timestamp(now);

return Status::OK();

}

// 判断参考线数量

bool has\_change\_lane = reference\_line\_info->size() > 1;

ADEBUG << "has\_change\_lane: " << has\_change\_lane;

// 如果只有一条参考线（比如往某个方向只有一条车道），那就通过updatestatus将车辆状态设置为CHANGE\_LANE\_FINISHED，不用换道；

if (!has\_change\_lane) {

//没有换道参考线（参考线数量小于1条）：如果上个周期状态是已经换道完成或者换道失败，

//则返回进入下个task或者下个周期；如果上个周期状态是正在换道，更新换道状态

const auto& path\_id = reference\_line\_info->front().Lanes().Id();

if (prev\_status->status() == ChangeLaneStatus::CHANGE\_LANE\_FINISHED) {

} else if (prev\_status->status() == ChangeLaneStatus::IN\_CHANGE\_LANE) {

UpdateStatus(now, ChangeLaneStatus::CHANGE\_LANE\_FINISHED, path\_id);

} else if (prev\_status->status() == ChangeLaneStatus::CHANGE\_LANE\_FAILED) {

} else {

const std::string msg =

absl::StrCat("Unknown state: ", prev\_status->ShortDebugString());

AERROR << msg;

return Status(ErrorCode::PLANNING\_ERROR, msg);

}

return Status::OK();

}

//

//下面的else处理不止一条参考线的情况，正常道路都不止一条参考线，

//主要逻辑为状态切换，实际操作还是通过updatestatus来实时更新车辆的换道状态。

else { // has change lane in reference lines.

auto current\_path\_id = GetCurrentPathId(\*reference\_line\_info);

if (current\_path\_id.empty()) {

const std::string msg = "The vehicle is not on any reference line";

AERROR << msg;

return Status(ErrorCode::PLANNING\_ERROR, msg);

}

if (prev\_status->status() == ChangeLaneStatus::IN\_CHANGE\_LANE) {

if (prev\_status->path\_id() == current\_path\_id) {

PrioritizeChangeLane(true, reference\_line\_info);（见1）

} else {

// RemoveChangeLane(reference\_line\_info);（移除该参考线的改变车道）

PrioritizeChangeLane(false, reference\_line\_info);

ADEBUG << "removed change lane.";

UpdateStatus(now, ChangeLaneStatus::CHANGE\_LANE\_FINISHED,

current\_path\_id);

}

return Status::OK();

} else if (prev\_status->status() == ChangeLaneStatus::CHANGE\_LANE\_FAILED) {

// 添加一个 optimization\_failure计数器去加入change\_lane\_failed 状态

if (now - prev\_status->timestamp() <

lane\_change\_decider\_config.change\_lane\_fail\_freeze\_time()) {

// RemoveChangeLane(reference\_line\_info);

PrioritizeChangeLane(false, reference\_line\_info);

ADEBUG << "freezed after failed";

} else {

//更新当前的车道为车的换道状态

UpdateStatus(now, ChangeLaneStatus::IN\_CHANGE\_LANE, current\_path\_id);

ADEBUG << "change lane again after failed";

}

return Status::OK();

} else if (prev\_status->status() ==

ChangeLaneStatus::CHANGE\_LANE\_FINISHED) {

if (now - prev\_status->timestamp() <

lane\_change\_decider\_config.change\_lane\_success\_freeze\_time()) {

// RemoveChangeLane(reference\_line\_info);

PrioritizeChangeLane(false, reference\_line\_info);

ADEBUG << "freezed after completed lane change";

} else {

PrioritizeChangeLane(true, reference\_line\_info);

UpdateStatus(now, ChangeLaneStatus::IN\_CHANGE\_LANE, current\_path\_id);

ADEBUG << "change lane again after success";

}

} else {

const std::string msg =

absl::StrCat("Unknown state: ", prev\_status->ShortDebugString());

AERROR << msg;

return Status(ErrorCode::PLANNING\_ERROR, msg);

}

}

return Status::OK();

}

lane\_change\_decider的工作流程图：（结合这上面的代码一一对应着观看更好的理解其工作流程）

