技术文档 | Apollo3.5 软件架构

知识点

敲黑板，本文需要学习的知识点有

RTK　底盘状态

人机界面　调试

控制模块　路由

感知模块

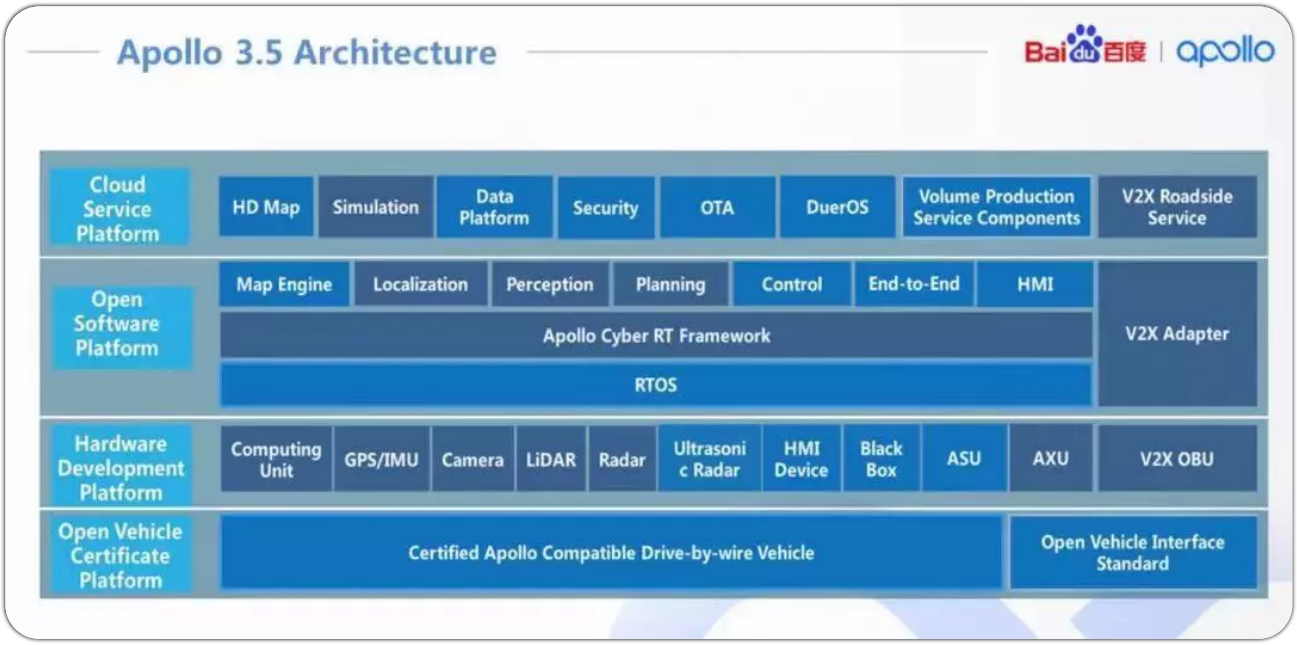
Apollo平台架构包括:

云服务平台: HD地图，仿真，数据平台，安全，云更新，指令控制。

软件平台：地图工程，定位，感知，预测，规划，安全，控制，自主规划，人机交互。

硬件平台：计算单元，GPS，相机，激光雷达，毫米波雷达，超声波雷达，人机接口设备，黑盒（数据记录）。

车辆平台：线控车辆。



Apollo3.5 平台架构

面向自动驾驶的Apollo 平台3.5版本主要包括以下软件模块：

- **感知**—感知模块识别自动驾驶车辆的周围环境。该模块包含两个重要的子模块：障碍物检测和交通灯检测。
- **预测**—预测模块预测感知障碍物的未来运动轨迹。
- **路由**—路由模块告诉自动驾驶车辆如何经过一系列车道或道路到达目的地。
- **规划**—规划模块规划自动驾驶车辆所采取的时空轨迹。
- **控制**—控制模块通过生成控制命令（如加速、刹车和转向）来执行规划模块提供的时空轨迹。
- **Can总线**—Can总线是传递控制命令到车辆硬件的接口。它也负责将硬件系统信息传递给软件系统。
- **高精地图**—该模块类似于一个库。它更多作为查询引擎的功能提供有关道路的临时结构化信息，而不是发布和订阅消息。
- **定位**—定位模块利用GPS、Lidar和IMU等各种信息来估计自动驾驶车辆本身的位置。
- **人机接口**—Apollo平台的人机接口或者DreamView是一个查看自动驾驶车辆状态的模块，同时也可用来测试其他模块和实时控制车辆。
- **监测**—车辆中包括硬件在内的所有模块的监控系统。

Apollo感知模块3.5版具有以下新的特性：

- 支持VLS-128线激光雷达
- 使用多摄像头方法检测障碍物
- 可配置的传感器融合方法

感知模块使用5个摄像头（2个前置、2个侧面和后方各1个），2台雷达（车头和车尾）以及3个16线激光雷达（2个在车尾和1个在车头）和 1 个128 线激光雷达来识别障碍并融合各个传感器的信息来得到最后的跟踪列表。

障碍物检测子模块对障碍物进行检测、分类和跟踪。同时预测障碍物的运动和位置信息(例如：方向和速度)。对于车道线，我们通过后期对道路的像素级分割，并计算出其与参考车辆的相对位置

（如L0、L1、R0、R1），来构建道路实例。

预测模块将对感知到的障碍物在未来一段时间内的运动轨迹进行估计。输出的预测消息将会打包感知信息。预测模块将订阅定位、规划和感知障碍等消息，如下所示。

```
void PredictionComponent::OnLocalization(
    const LocalizationEstimate& localization_msg) {
  auto ptr_ego_pose_container =
      ContainerManager::Instance()->GetContainer<PoseContainer>(
          AdapterConfig::LOCALIZATION);
  CHECK(ptr_ego_pose_container != nullptr);
  ptr_ego_pose_container->Insert(localization_msg);

  ADEBUG << "Received a localization message ["
         << localization_msg.ShortDebugString() << "].";
}

void PredictionComponent::OnPlanning(
    const planning::ADCTrajectory& planning_msg) {
  auto ptr_ego_trajectory_container =
      ContainerManager::Instance()->GetContainer<ADCTrajectoryContainer>(
          AdapterConfig::PLANNING_TRAJECTORY);
  CHECK(ptr_ego_trajectory_container != nullptr);
  ptr_ego_trajectory_container->Insert(planning_msg);

  ADEBUG << "Received a planning message ["
         << planning_msg.ShortDebugString() << "].";
}

void PredictionComponent::OnPerception(
    const PerceptionObstacles& perception_msg) {
  // Insert obstacle
  auto end_time1 = std::chrono::system_clock::now();
  auto ptr_obstacles_container = ContainerManager::Instance()->GetContainer
      ObstaclesContainer>(AdapterConfig::PERCEPTION_OBSTACLES);
  CHECK(ptr_obstacles_container != nullptr);
```

当接收到定位更新之后，预测模块会更新自己的内部状态。当感知模块发出障碍物信息时，会触发真正的预测行为。

定位模块融合各种数据对自动驾驶车辆进行定位。通常有两种定位模式：OnTimer和多传感器融合。

第一种定位方法基于RTK，使用一个基于时间的回调函数OnTimer，具体代码如下所示。

```
bool RTKLocalizationComponent::Proc(
    const std::shared_ptr<localization::Gps>& gps_msg) {
  localization_->GpsCallback(gps_msg);

  if (localization_->IsServiceStarted()) {
    LocalizationEstimate localization;
    localization_->GetLocalization(&localization);
    LocalizationStatus localization_status;
    localization_->GetLocalizationStatus(&localization_status);

    // publish localization messages
    PublishPoseBroadcastTopic(localization);
    PublishPoseBroadcastTF(localization);
    PublishLocalizationStatus(localization_status);
    ADEBUG << "[OnTimer]: Localization message publish success!";
  }

  return true;
}
```

另一种定位方式是多传感器融合方法，在该方法中，将会注册一组事件驱动的回调函数，代码如下所示。

```
bool MSFLocalizationComponent::Proc(
    const std::shared_ptr<drivers::gnss::Imu>& imu_msg) {
  localization_.OnRawImu(imu_msg);
  return true;
}

LocalizationMsgPublisher::LocalizationMsgPublisher(
    const std::shared_ptr<cyber::Node>& node)
    : node_(node), tf2_broadcaster_(node) {}

bool LocalizationMsgPublisher::InitConfig() {
  localization_topic_ = FLAGS_localization_topic;
  broadcast_tf_frame_id_ = FLAGS_broadcast_tf_frame_id;
  broadcast_tf_child_frame_id_ = FLAGS_broadcast_tf_child_frame_id;
  lidar_local_topic_ = FLAGS_localization_lidar_topic;
  gnss_local_topic_ = FLAGS_localization_gnss_topic;
  localization_status_topic_ = FLAGS_localization_msf_status;

  return true;
}
```

要计算出自动驾驶车辆经过的车道和道路，路由模块需要知道起点和终点。通常情况下路由的起点是车辆所在的位置。RoutingResponse的计算和发布过程如下所示。

```cpp
bool RoutingComponent::Proc(const std::shared_ptr<RoutingRequest>& request)
  auto response = std::make_shared<RoutingResponse>();
  if (!routing_.Process(request, response.get())) {
    return false;
  }
  common::util::FillHeader(node_->Name(), response.get());
  response_writer_->Write(response);
  {
    std::lock_guard<std::mutex> guard(mutex_);
    response_ = std::move(response);
  }
  return true;
}
```

Apollo3.5使用多源信息来规划一条安全和无碰撞的运行轨迹，因此规划模块几乎与所有其他模块都有交互。

随着Apollo平台的成熟并应用在不同路况和驾驶用例下，规划已经演化为一个更加模块化、场景指定和全局的方法。

在该方法中，每个驾驶用例都被视为不同驾驶场景。这种方法是很有用的，因为在当前这种方式下，修复一个特定场景下的问题不会影响其他场景，在以前版本中，所有的驾驶用例都被认为是一个驾驶场景，一个问题的修复会影响其他的驾驶用例。

首先，规划模块会获取预测模块的输出。由于预测输出打包了最开始感知的障碍物信息，规划模块订阅的是红绿灯检测输出，而不是感知障碍物输出。

然后，规划模块获取路由输出。在某些情况下，如果当前路由无法执行的，规划模块可能会发送路由请求来触发新的路由计算。

最后，规划模块需要知道位置（定位：我在哪里）以及当前的自动驾驶车辆的信息（底盘：我的状态是什么）。

```
// process fused input data
local_view_.prediction_obstacles = prediction_obstacles;
local_view_.chassis = chassis;
local_view_.localization_estimate = localization_estimate;
{
  std::lock_guard<std::mutex> lock(mutex_);
  if (!local_view_.routing ||
      hdmap::PncMap::IsNewRouting(*local_view_.routing, routing_)) {
    local_view_.routing =
        std::make_shared<routing::RoutingResponse>(routing_);
    local_view_.is_new_routing = true;
  } else {
    local_view_.is_new_routing = false;
  }
}
```

控制模块将规划轨迹作为输入，并产生控制指令给Can总线。它主要有5个数据接口函数：OnPad(中控面板)，OnMonitor（监控模块），OnChassis（底盘），OnPlanning（规划）和OnLocalization（定位）。

```cpp
void ControlComponent::OnPad(const std::shared_ptr<PadMessage> &pad) {
  pad_msg_.CopyFrom(*pad);
  ADEBUG << "Received Pad Msg:" << pad_msg_.DebugString();
  AERROR_IF(!pad_msg_.has_action()) << "pad message check failed!";

  // do something according to pad message
  if (pad_msg_.action() == DrivingAction::RESET) {
    AINFO << "Control received RESET action!";
    estop_ = false;
    estop_reason_.clear();
  }
  pad_received_ = true;
}

void ControlComponent::OnChassis(const std::shared_ptr<Chassis> &chassis)
  ADEBUG << "Received chassis data: run chassis callback.";
  std::lock_guard<std::mutex> lock(mutex_);
  latest_chassis_.CopyFrom(*chassis);
}

void ControlComponent::OnPlanning(
    const std::shared_ptr<ADCTrajectory> &trajectory) {
  ADEBUG << "Received chassis data: run trajectory callback.";
  std::lock_guard<std::mutex> lock(mutex_);
  latest_trajectory_.CopyFrom(*trajectory);
}

void ControlComponent::OnLocalization(
    const std::shared_ptr<LocalizationEstimate> &localization) {
  ADEBUG << "Received control data: run localization message callback.";
  std::lock_guard<std::mutex> lock(mutex_);
  latest_localization_.CopyFrom(*localization);
}

void ControlComponent::OnMonitor(
    const common::monitor::MonitorMessage &monitor_message) {
  for (const auto &item : monitor_message.item()) {
    if (item.log_level() == common::monitor::MonitorMessageItem::FATAL) {
      estop_ = true;
      return;
    }
  }
}
```

OnPad和OnMonitor是使用基于Pad的人机界面和模拟的日常交互。

Can总线有两个数据接口函数，如下所示。

```
void CanbusComponent::OnControlCommand(const ControlCommand &control_command) {
  int64_t current_timestamp =
      apollo::common::time::AsInt64<common::time::micros>(Clock::Now());
  // if command coming too soon, just ignore it.
  if (current_timestamp - last_timestamp_ < FLAGS_min_cmd_interval * 1000) {
    ADEBUG << "Control command comes too soon. Ignore.\n Required "
              "FLAGS_min_cmd_interval["
           << FLAGS_min_cmd_interval << "], actual time interval["
           << current_timestamp - last_timestamp_ << "].";
    return;
  }

  last_timestamp_ = current_timestamp;
  ADEBUG << "Control_sequence_number:"
         << control_command.header().sequence_num() << ", Time_of_delay:"
         << current_timestamp -
                static_cast<int64_t>(control_command.header().timestamp_sec());

  if (vehicle_controller_->Update(control_command) != ErrorCode::OK) {
    AERROR << "Failed to process callback function OnControlCommand because "
              "vehicle_controller_->Update error.";
    return;
  }
  can_sender_.Update();
}

void CanbusComponent::OnGuardianCommand(
    const GuardianCommand &guardian_command) {
  apollo::control::ControlCommand control_command;
  control_command.CopyFrom(guardian_command.control_command());
  OnControlCommand(control_command);
}
```

第一个是OnControlCommand，它是一个基于事件的带回调函数的发布者，当Canbus模块接收到控制命令时将会触发该函数。第二个是数据接口OnGuardianCommand。

在Apollo中，人机界面或者Dreamview是一个Web应用程序。它具有以下功能：

对相关自动驾驶模块的当前输出进行可视化处理，例如规划轨迹、汽车定位、底盘状态等。

为用户提供人机界面查看硬件状态，关闭/开启模块，启动自动驾驶车辆。

提供调试工具，如PNC监视器可以有效跟踪模块问题。

监控模块是车辆上包括硬件在内的所有模块的监控系统。监控模块从不同的模块接收数据并将数据传送到人机界面供驾驶员查看，保证所有的模块在没有任何问题的情况下运行。

在模块或硬件出现故障时，监控系统将向守护模块发送报警信号（守护模块是一个新的行动中心模块），由守护模块决定采取什么必要的措施以避免碰撞。

这个新模块基本上是一个操作中心，它根据监控模块发送来的数据做出决定。守护模块有两个主要功能:

所有模块工作正常情况——守护模块允许控制流正常工作。控制信号被发送到 Can总线，就好像守护模块不存在一样。

监控模块检测到有模块崩溃的情况——如果监控模块检测到故障，守护模块将阻止控制信号传送到Can总线，并执行停车操作。守护模块有三种方式来决定如何停车，如果要停车，守护模块将检查超声波传感器的状态。

如果超声波传感器工作正常且没有检测到障碍物，守护模块将缓慢停车。
如果该传感器没有反应，守护模块将采取紧急制动快速停车。

存在一种特殊情况，如果HMI通知驾驶员即将发生碰撞事故而驾驶员在10秒内没有采取干预措施，守护模块将采取紧急制动快速停车。

注意:

1. 在上述任何一种情况下，如果监控模块检测到任何模块或硬件出现故障，守护都将停止该车。

2. 监控和守护解耦以保证没有单点故障，并且使用模块化方法，可以在保证不影响监控系统功能的情况下，对操作中心做出增加额外操作的修改行为，因为监控模块仍然与HMI保持通信。