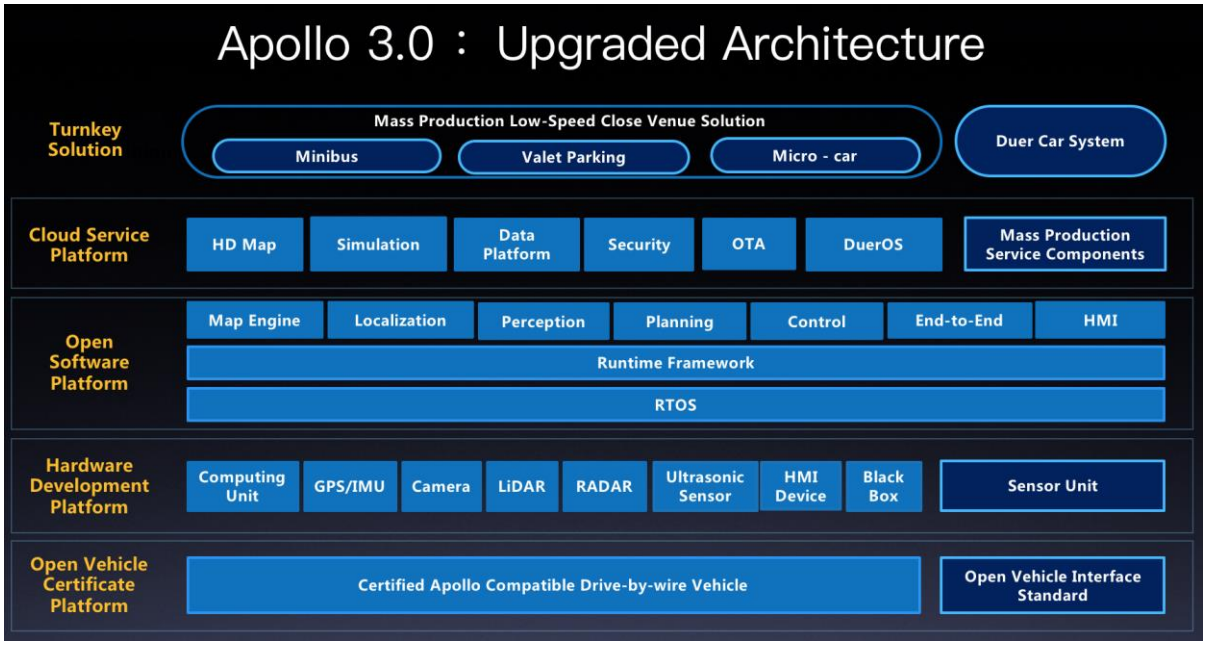
Apollo 学习笔记

- 简介
- 目录结构
- 编译

简介



apollo 是百度的自动驾驶开源框架，根据自动驾驶的功能划分为不同的模块，下面会根据目录结构和功能模块分别介绍和学习"apollo 模块"。下面简单介绍下各个模块的作用:

- **定位** - 知道汽车在哪里,这里的定位可能涉及方方面面,比如 GPS,但是 GPS 的精度只有米级别,有下面几个场景会不太合适,比如你在桥洞里,GPS 信号不好的情况,另外还有一种情况是,停车的时候,你要知道前后车的距离,另外比如在雪天或者路况复杂的情况,这些情况下,仅有的 GPS 信号可能不能满足我们的要求,因此无人车又增加了激光雷达来测量周围环境的距离,而且可以精确到厘米。
  因此产生了高精地图的需求,高精地图可以把周围的 3D 环境都记录下来,这样我们就可以通过 3D 图像来匹配周围的场景,来找到自己的位置,而且高精地图还可以记录比地图多的多的东西,比如红绿灯的位置,交通标志,左转还是右转道.通过高精地图我们不仅可以知道道路情况,还可以知道车辆需要获取的一些其他信息,让车辆知道自己的实时位置。
- **感知** - 我们总是希望车辆行驶在马路中间,这样更加安全,这就需要追踪到道路的路牙线,而道路随时会出现拐弯,那么追踪路牙线就用到了图像处理技术,另外还要感知到什么是车辆,什么是行人,主要涉及到图像的语义分割,感知是自动驾驶中最难,而且最具有挑战性的一块,因为只有感知到周围的行人,车辆,以及突发状况,才能为后面规划线路。

- **规划** - 目前已经知道当前道路情况,而且也已经感知到前面的车辆或者行人,如何去规划我们的行驶线路呢,这里需要解决的就是 2 个点之间的线路,而且行驶中途可能会出现新的情况,又需要重新规划线路,还有一种情况是,通过高精地图,我们已经知道前面需要转弯了,我们可以提前调整线路来适应这种需求。
- **控制** - 现在已经规划出了一条线路,剩下的就是控制汽车,按照已经规划好的线路行驶,而且如果遇到突发状况,需要立即停车,而且控制汽车能够按照预定的线路不会出现很大的偏离,这就是控制要做的事情。

目录结构

apollo 是一个**全栈**(包含编程语言，框架，应用，代码等)的自动驾驶框架，下面是整个 apollo 代码的目录结构，主要是按照功能模块划分：

|-cyber 消息中间件， 替换 ros 作为消息层
|-docker 容器相关
|-docs 文档相关
|-modules 自动驾驶模块， 主要的定位， 预测， 感知， 规划都在这里
    |-calibration 校准， 主要用于传感器坐标的校准， 用于感知模块做传感器融合
    |-**canbus** 通讯总线， 工业领域的标准总线， 鉴于工业界的保守， 我估计后面会有新的总线来取代
    |-common
    |-contrib
    |-control 控制模块， 根据 planning 生成的路径对车辆轨迹进行控制， 再底层就是发送命令到 can 总线， 实现车辆的控制。
    |-data 地图等生成好的数据放在这里（其他数据待补充）
    |-dreamview 仿真， 能够对自动驾驶过程中的数据进行回放， 其他厂家也有推出一些仿真平台， 后面有机会再介绍下
    |-**drivers** 雷达，lidar，**GPS，canbus，camera** 等驱动
    |-guardian 监护程序
    |-**localization 定位，获取汽车的当前位置**
    |-**map 地图模块**
    |-monitor 监控模块， 主要是监控汽车状态， 并且记录， 用于故障定位， 健康检查等
    |-perception 感知，获取汽车当前的环境， 行人， 车辆， 红绿灯等， 给 planning 模块规划线路
    |-planning 规划，针对感知到的情况，对路径做规划，短期规划，只规划100-200M 的距离，生成好的路径给 control 模块
    |-prediction 预测，属于感知模块，对运动物体的轨迹做预测
    |-routing 导航线路，就是百度地图上查询2点之间的线路，生成的线路短期规划还是 planning 模块
    |-third_party_perception 第三方感知模块
    |-tools 工具，这里面的工具倒是很多，后面再详细介绍下
    |-transform 转换，主要是？
    |-v2x 顾名思义就 vehicle-to-everything，其希望实现车辆与一切可能影响车辆的实体实现信息交互，
            目的是减少事故发生，减缓交通拥堵，降低环境污染以及提供其他信息服务。
|-scripts 脚本
|-third_party 第三方库
|-tools 工具目录，基本就是个空目录

编译

apollo 采用的是 bazel 来进行编译， JAVA 的 maven 包管理的功能特别好用，第一能解决包自动下载，第二解决依赖传递的问题，第三解决了编译的问题。也就是说你只要引用对应的包，就可以把包的依赖全部解决。而 bazel 就是 c++对应的编译管理，bazel 主要是通过 WORKSPACE 和 BUILD 文件来进行编译。

Canbus

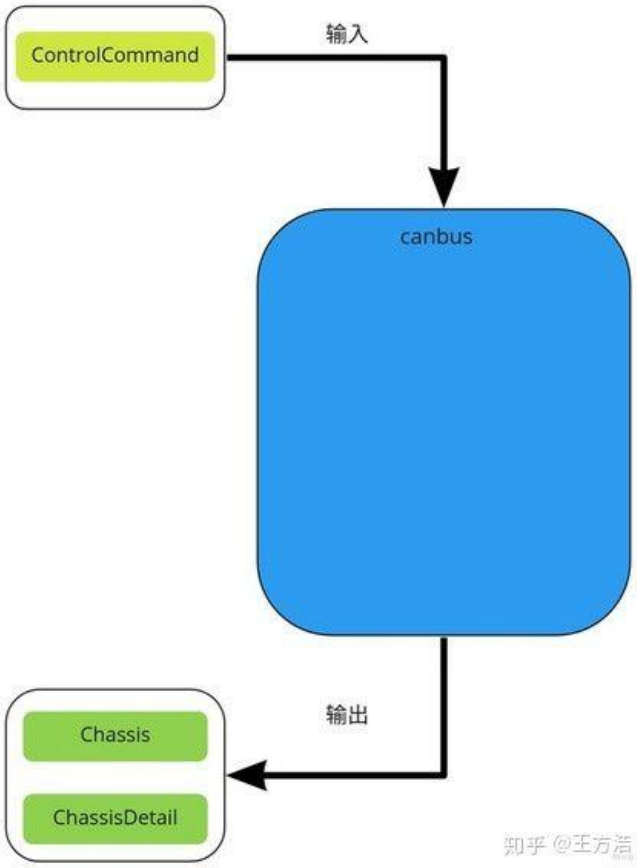
Table of Contents

- Canbus 模块介绍
- Canbus 模块主流程
    - 车辆工厂模式(VehicleFactory)
    - 车辆控制器(LincolnController)
- Canbus(驱动程序)
    - 消息管理器(MessageManager)
    - 消息接收(CanReceiver)
    - 消息发送(CanSender)
    - canbus 客户端(CanClient)
- Reference

Canbus 模块介绍

我们先看下什么是 Canbus： 控制器局域网（Controller Area Network，简称 CAN 或者 CAN bus）是一种车用总线标准。被设计用于在不需要主机（Host）的情况下，允许网络上的节点相互通信。采用广播机制，并利用标识符来定义内容和消息的优先顺序，使得 canbus 的扩展性良好，同时不基于特殊类型（Host）的节点，增加了升级网络的便利性。
这里的 **Canbus 模块其实可以称为 Chassis 模块**，主要的作用是反馈车当前的状态（航向，角度，速度等信息），并且发送控制命令到车线控底盘，**可以说 Canbus 模块是车和自动驾驶软件之间的桥梁**。由于这个模块和"drivers/canbus"的联系紧密，因此也一起在这里介绍。
Canbus 模块是车和自动驾驶软件之间的桥梁，通过 canbus 驱动(drivers/canbus)来实现将车身信息发送给 apollo 上层软件，同时接收控制命令，发送给汽车线控底盘实现对汽车的控制。

那么 canbus 模块的输入是什么？输出是什么呢？



可以看到 canbus 模块：

- **输入** - 1. ControlCommand（控制命令）
- **输出** - 1. Chassis（汽车底盘信息）, 2. ChassisDetail（汽车底盘信息详细信息）

Canbus 模块的输入是 control 模块发送的控制命令，输出汽车底盘信息，这里 apollo 的上层模块被当做一个 can_client 来处理，实现接收和发送 canbus 上的消息。

Canbus 模块的目录结构如下：

```
├── BUILD                   // bazel 编译文件
├── canbus_component.cc      // canbus 主入口
├── canbus_component.h
├── canbus_test.cc           // canbus 测试
├── common                  // gflag 配置
├── conf                    // 配置文件
├── dag                     // dag 依赖
├── launch                  // launch 加载
├── proto                   // protobuf 文件
├── testdata                // 测试数据
├── tools                   // 遥控汽车和测试 canbus 总线工具
└── vehicle                 //
```

接着我们来分析下 Canbus 模块的执行流程。

Canbus 模块主流程

周期性任务

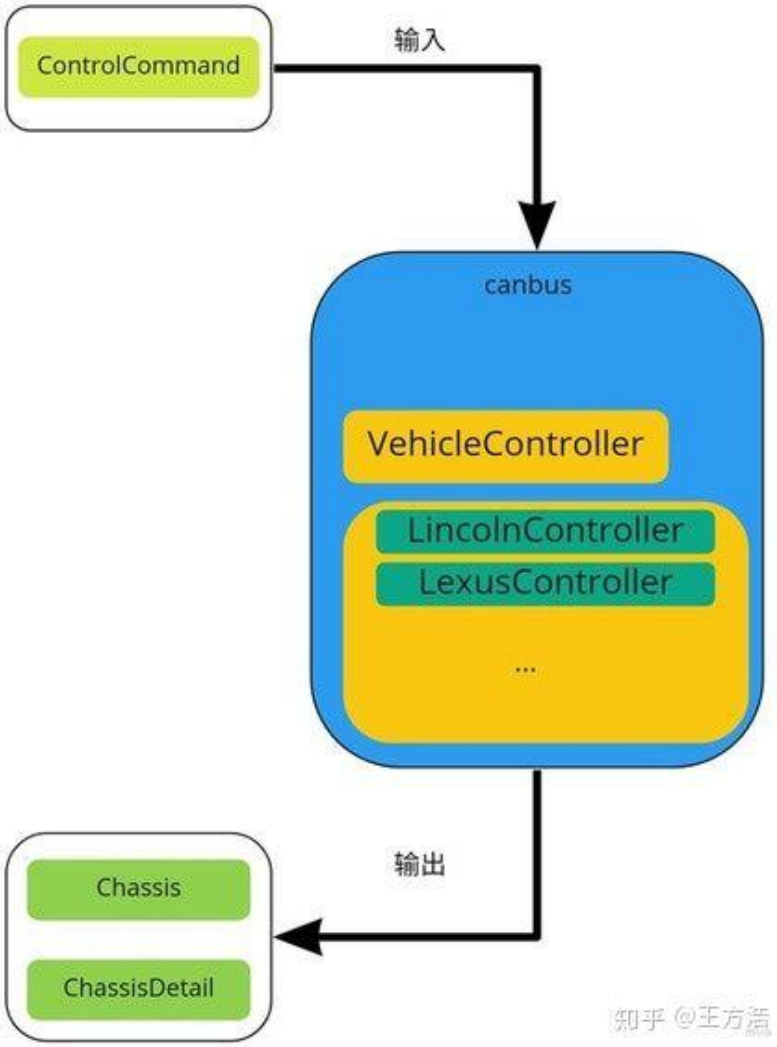
Canbus 模块的主流程在文件"canbus_component.cc"中，canbus 模块为定时触发，每 10ms 执行一次，发布 chassis 信息，而 ControlCommand 则是每次读取到之后触发回调"OnControlCommand"，发送"control_command"到线控底盘。

用户触发任务

```
1 bool CanbusComponent::Proc()
2 {
3     PublishChassis();
4     if (FLAGS_enable_chassis_detail_pub)
5     {
6         PublishChassisDetail();
7     }
8     return true;
9 }
```

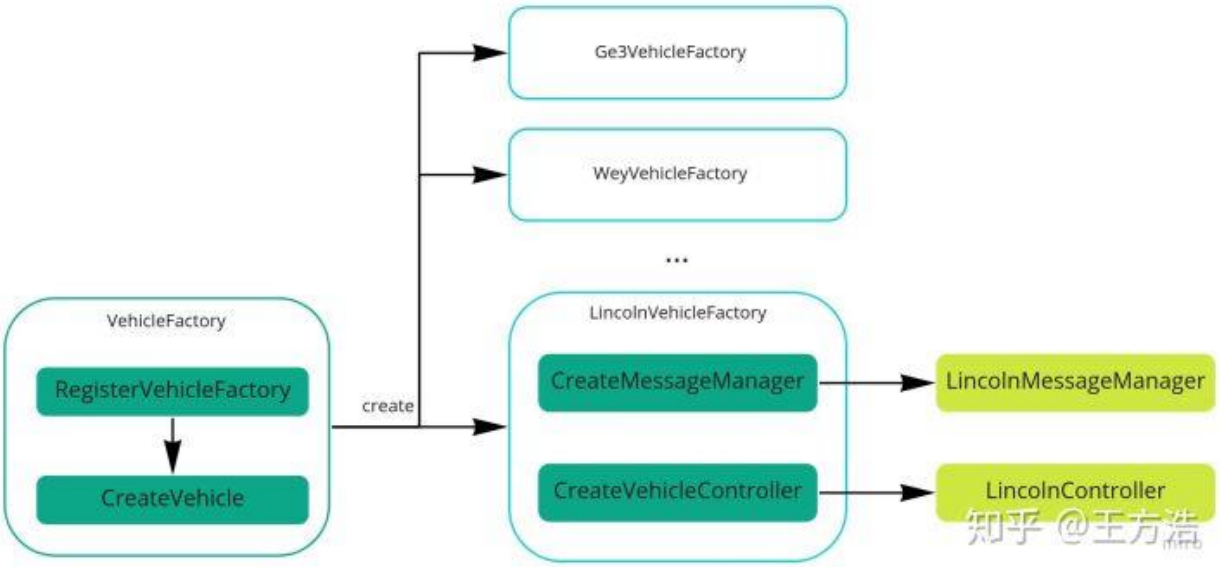
由于不同型号的车辆的 canbus 命令不一样，在"/vehicle"中适配了不同型号车辆的 canbus 消息格式，所有的车都继承自 Vehicle_controller 基类，通过对 Vehicle_controller 的抽象来

发送和读取 canbus 信息。



车辆工厂模式(VehicleFactory)

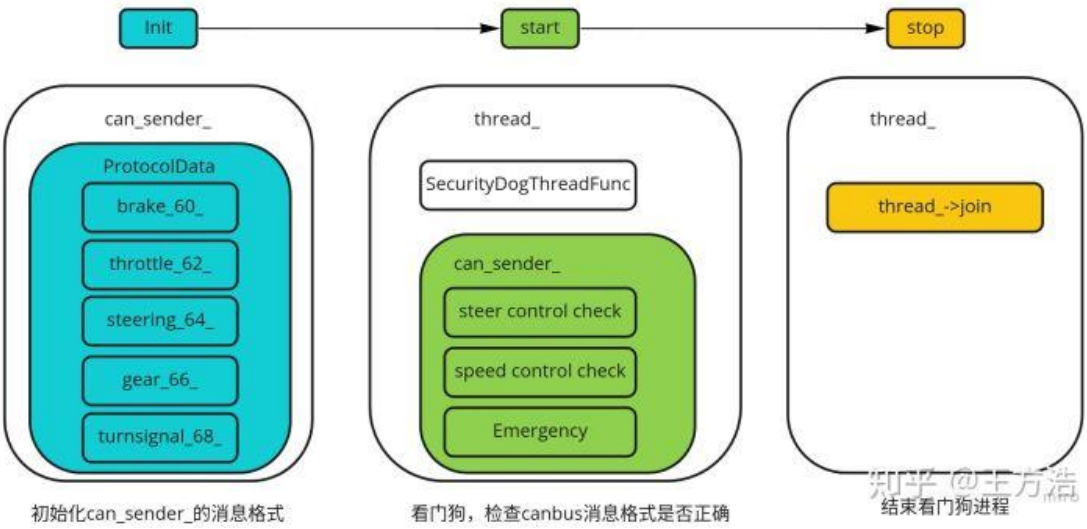
在 vehicle 中可以适配不同的车型，而每种车型都对应一个 vehicle_controller，创建每种车辆的控制器(VehicleController)和消息管理(MessageManager)流程如下：



VehicleFactory 类通过创建不同的类型 AbstractVehicleFactory，每个车型自己的 Factory 在创建出对应的 VehicleController 和 MessageManager，用林肯来举例子就是：
VehicleFactory 创建 LincolnVehicleFactory，之后通过 CreateMessageManager 和 CreateVehicleController 创建对应的控制器（LincolnController）和消息管理器（LincolnMessageManager）。
上述代码流程用到了设计模式的工厂模式，通过车辆工厂创造不同的车辆类型。

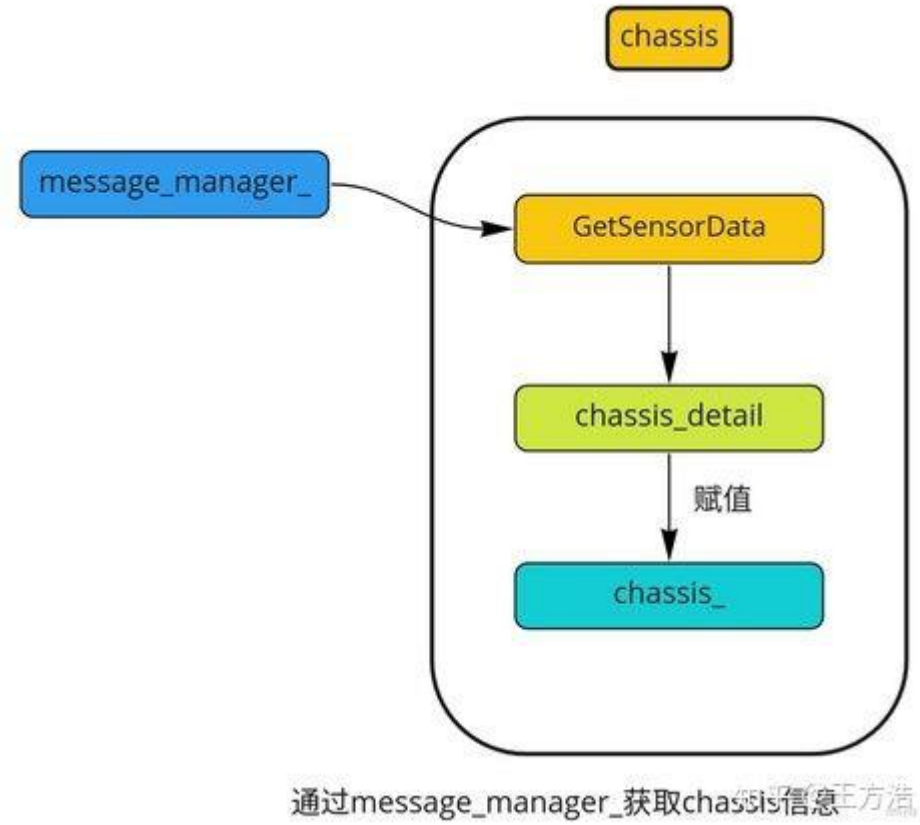
车辆控制器(LincolnController)

下面以林肯来介绍 LincolnController，以及如何接收 chassis 信息，其它的车型可以以此类推，下面主要分为 2 部分介绍，第一部分为 controller 的 init->start->stop 流程，第二部分为 chassis 信息获取：



可以看到 control 模块初始化(init)的过程获取了发送的消息的格式，通过 can_sender 应该发

送那些消息，而启动(start)之后启动一个看门狗，检查 canbus 消息格式是否正确，最后关闭(stop)模块则是结束看门狗进程。



通过message_manager_获取chassis信息

而 chassis 的获取则是通过 message_manager_获取 chassis_detail，之后对 chassis 进行赋值。

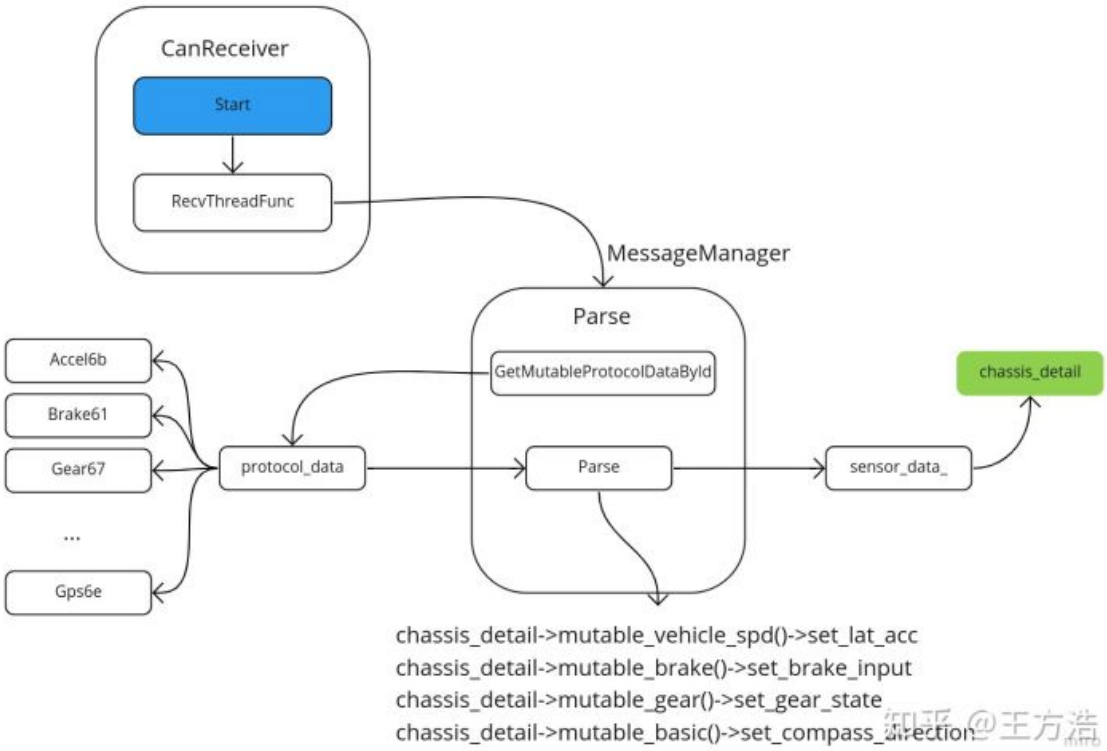
Canbus(驱动程序)

上层的 canbus 就介绍完成了，而 canbus 的发送(CanSender)和接收(CanReceiver)，还有消息管理(MessageManager)都是在"drivers/canbus"中实现的。

消息管理器(MessageManager)

MessageManager 是如何获取消息的呢？

MessageManager 主要作用是解析和保存 canbus 数据，而具体的接收和发送则是在"CanReceiver"和"CanSender"中，拿接收消息举例子，也就是说 CanReceiver 收到消息后，会调用 MessageManager 中的 parse 去解析消息，消息的解析协议在"modules/canbus/vehicle/lincoln/protocol"中，每个消息把自己对应的信息塞到"chassis_detail"中完成了消息的接收。



消息接收(CanReceiver)

canbus 消息的接收在上面有介绍，在 CanReceiver 中的"Start"调用"RecvThreadFunc"实现消息的接收，这里会启动一个异步进程去完成接收。

```
template <typename SensorType>
::apollo::common::ErrorCode CanReceiver<SensorType>::Start()
{
    if (is_init_ == false)
    {
        return ::apollo::common::ErrorCode::CANBUS_ERROR;
    }
    is_running_.exchange(true);

    // 启动异步接收消息
    async_result_ = cyber::Async(&CanReceiver<SensorType>::RecvThreadFunc, this);
    return ::apollo::common::ErrorCode::OK;
}
```

```
/* RecvThreadFunc通过"can_client_"接收消息，然后通过"MessageManager"去解析消息，在MessageManager中有讲到。 */
template <typename SensorType>
void CanReceiver<SensorType>::RecvThreadFunc()
{
    ...
    while (IsRunning())
    {
        std::vector<CanFrame> buf;
        int32_t frame_num = MAX_CAN_RECV_FRAME_LEN;

        // 1. can_client_接收canbus数据
        if (can_client_->Receive(&buf, &frame_num) !=
                ::apollo::common::ErrorCode::OK)
        {

            cyber::USleep(default_period);
            continue;
        }
        ...
        for (const auto &frame : buf)
        {
            uint8_t len = frame.len;
            uint32_t uid = frame.id;
            const uint8_t *data = frame.data;

            // 2. MessageManager解析canbus数据
            pt_manager_->Parse(uid, data, len);
            if (enable_log_)
            {
                ADEBUG << "recv_can_frame#" << frame.CanFrameString();
            }
        }
        cyber::Yield();
    }
    AINFO << "Can client receiver thread stopped.";
}
```

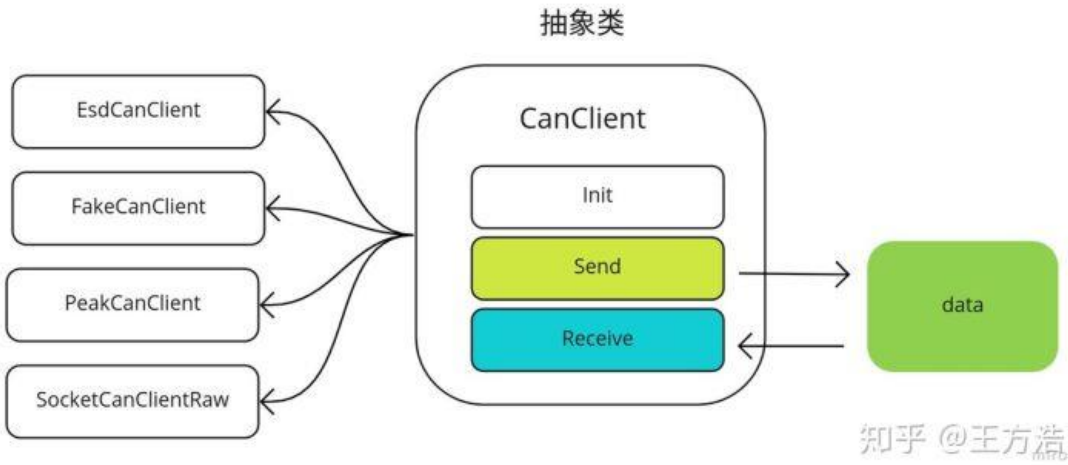
消息发送(CanSender)

消息发送对应的是在 CanSender 中的"Start"调用"PowerSendThreadFunc"，我们可以看具体实现:

```cpp
template <typename SensorType>
common::ErrorCode CanSender<SensorType>::Start()
{
    if (is_running_)
    {
        AERROR << "Cansender has already started.";
        return common::ErrorCode::CANBUS_ERROR;
    }
    is_running_ = true;

    // 启动线程发送消息
    thread_.reset(new std::thread([this] { PowerSendThreadFunc(); }));

    return common::ErrorCode::OK;
}
PowerSendThreadFunc再通过"can_client"发送消息：
std::vector<CanFrame> can_frames;
CanFrame can_frame = message.CanFrame();
can_frames.push_back(can_frame);

// 通过can_client发送消息
if (can_client_->SendSingleFrame(can_frames) != common::ErrorCode::OK)
{
    AERROR << "Send msg failed:" << can_frame.CanFrameString();
}
```

canbus 客户端(CanClient)

CanClient 是 canbus 客户端，同时也是 canbus 的驱动程序，针对不同的 canbus 卡，对发送和接收进行封装，并且提供给消息发送和接收控制器使用。



拿"EsdCanClient"来举例子，发送在"Send"函数中，调用的是第三方的硬件驱动，目录在"third_party/can_card_library/esd_can"，实现 can 消息的发送：

```
ErrorCode EsdCanClient::Send(const std::vector<CanFrame> &frames,
                             int32_t *const frame_num)
{
    ...
    // canWrite为第三方库的硬件驱动，third_party/can_card_library/esd_can
    // Synchronous transmission of CAN messages
    int32_t ret = canWrite(dev_handler_, send_frames_, frame_num, nullptr);
    if (ret != NTCAN_SUCCESS)
    {
        AERROR << "send message failed, error code: " << ret << ", "
               << GetErrorString(ret);
        return ErrorCode::CAN_CLIENT_ERROR_BASE;
    }
    return ErrorCode::OK;
}
```

其他的 can 卡可以参考上述的流程，至此整个 canbus 驱动就分析完成了。

如何在 Apollo 中添加新的车辆

简介

本文阐述了如何向 Apollo 中添加新的车辆。

注意：Apollo 控制算法将林肯 MKZ 配置为默认车辆。

添加新的车辆时，如果您的车辆需要不同于 Apollo 控制算法提供的属性，请参考：

使用适合您的车辆的其它控制算法。

修改现有算法的参数以获得更好的结果。

增加新车辆

按照以下步骤以实现新车辆的添加：

实现新的车辆控制器

实现新的消息管理器

实现新的车辆工厂

注册新的车辆

更新配置文件

实现新的车辆控制器

新的车辆控制器是从 VehicleController 类继承的。 下面提供了一个头文件示例。

```cpp
/**
 * @class NewVehicleController
 *
 * @brief this class implements the vehicle controller for a new vehicle.
 */
class NewVehicleController final : public VehicleController
{
public:
    /**
     * @brief initialize the new vehicle controller.
     * @return init error_code
     */
    ::apollo::common::ErrorCode Init(
        const VehicleParameter& params, CanSender* const can_sender,
        MessageManager* const message_manager) override;
    /**
     * @brief start the new vehicle controller.
     * @return true if successfully started.
     */
    bool Start() override;
    /**
     * @brief stop the new vehicle controller.
     */
    void Stop() override;
    /**
     * @brief calculate and return the chassis.
     * @returns a copy of chassis. Use copy here to avoid multi-thread issues.
     */
    Chassis chassis() override;
    // more functions implemented here
    ...
};

```

实现新的消息管理器

新的消息管理器是从 MessageManager 类继承的。下面提供了一个头文件示例。

```
/**
 * @class NewVehicleMessageManager
 *
 * @brief implementation of MessageManager for the new vehicle
 */
class NewVehicleMessageManager : public MessageManager
{
public:
    /**
     * @brief construct a lincoln message manager. protocol data for send and
     * receive are added in the construction.
     */
    NewVehicleMessageManager();
    virtual ~NewVehicleMessageManager();

    // define more functions here.
    ...
};
```

实施新的车辆工厂

新的车辆工厂是从 AbstractVehicleFactory 类继承的。下面提供了一个头文件示例。

```
/**
 * @class NewVehicleFactory
 *
 * @brief this class is inherited from AbstractVehicleFactory. It can be used to
 * create controller and message manager for lincoln vehicle.
 */
class NewVehicleFactory : public AbstractVehicleFactory
{
public:
    /**
    * @brief destructor
    */
    virtual ~NewVehicleFactory() = default;

    /**
     * @brief create lincoln vehicle controller
     * @returns a unique_ptr that points to the created controller
     */
    std::unique_ptr<VehicleController> CreateVehicleController() override;

    /**
     * @brief create lincoln message manager
     * @returns a unique_ptr that points to the created message manager
     */
    std::unique_ptr<MessageManager> CreateMessageManager() override;
};
```

一个.cc 示例文件如下：

```
std::unique_ptr<VehicleController>
NewVehicleFactory::CreateVehicleController()
{
    return std::unique_ptr<VehicleController>(new lincoln::LincolnController());
}

std::unique_ptr<MessageManager> NewVehicleFactory::CreateMessageManager()
{
    return std::unique_ptr<MessageManager>(new lincoln::LincolnMessageManager());
}
```

Apollo 提供可以用于实现新车辆协议的基类 ProtocolData。

注册新的车辆

在 modules/canbus/vehicle/vehicle_factory.cc 里注册新车辆。 下面提供了一个头文件示例。

```
void VehicleFactory::RegisterVehicleFactory()
{
    Register(VehicleParameter::LINCOLN_MKZ, []() -> AbstractVehicleFactory*
    {
        return new LincolnVehicleFactory();
    });
    // register the new vehicle here.
    Register(VehicleParameter::NEW_VEHICLE_BRAND, []() -> AbstractVehicleFactory*
    {
        return new NewVehicleFactory();
    });
}
```

更新配置文件

在 modules/canbus/conf/canbus_conf.pb.txt 中更新配置，在 Apollo 系统中激活车辆。

```
vehicle_parameter {
brand:
    NEW_VEHICLE_BRAND
    // put other parameters below
    ...
}
```

Localization

Table of Contents

- Localization 模块简介
- 代码目录
- RTK 定位流程
- Reference

Localization 模块简介

localization 模块主要实现了以下 2 个功能：

1. 输出车辆的位置信息（planning 模块使用）
2. 输出车辆的姿态，速度信息（control 模块使用）

其中 apollo 代码中分别实现了 3 种定位方法：

1. GNSS + IMU 定位
2. NDT 定位（点云定位）
3. MSF（融合定位）

MSF 方法参考论文"Robust and Precise Vehicle Localization Based on Multi-Sensor Fusion in Diverse City Scenes"

代码目录

下面是 localization 的目录结构，在查看具体的代码之前最好看下定位模块的 readme 文件:

```
├── common          // 声明配置(flags)，从 conf 目录中读取相应的值
├── conf            // 配置文件存放目录
├── dag             // cyber DAG 流
├── launch          // cyber 的配置文件，依赖 DAG 图（这 2 个和 cyber 有关的后面再分析）
├── msf             // 融合定位（gnss,点云,IMU 融合定位）
│   ├── common
│   │   ├── io
│   │   ├── test_data
│   │   └── util
│   ├── local_integ
│   ├── local_map
│   │   ├── base_map
│   │   ├── lossless_map
│   │   ├── lossy_map
│   │   ├── ndt_map
│   │   └── test_data
│   ├── local_tool
│   │   ├── data_extraction
│   │   ├── local_visualization
│   │   └── map_creation
│   └── params
│       ├── gnss_params
│       ├── vehicle_params
│       └── velodyne_params
├── ndt                         // ndt 定位
│   ├── map_creation
│   ├── ndt_locator
│   └── test_data
│       ├── ndt_map
│       └── pcds
├── proto                   // 消息格式
├── rtk                     // rtk 定位
└── testdata                // imu 和 gps 的测试数据
```

通过上述目录可以知道，定位模块主要实现了 rtk，ndt，msf 这 3 个定位方法，分别对应不同的目录。proto 文件夹定义了消息的格式，common 和 conf 主要是存放一些配置和消息 TOPIC。下面我们逐个分析 RTK 定位、NDT 定位和 MSF 定位。

RTK 定位流程

RTK 定位是通过 GPS 和 IMU 的信息做融合然后输出车辆所在的位置。RTK 通过基准站的获取当前 GPS 信号的误差，用来校正无人车当前的位置，可以得到厘米级别的精度。IMU 的输出频率高，可以在 GPS 没有刷新的情况下（通常是 1s 刷新一次）用 IMU 获取车辆的位置。下面是 RTK 模块的目录结构。

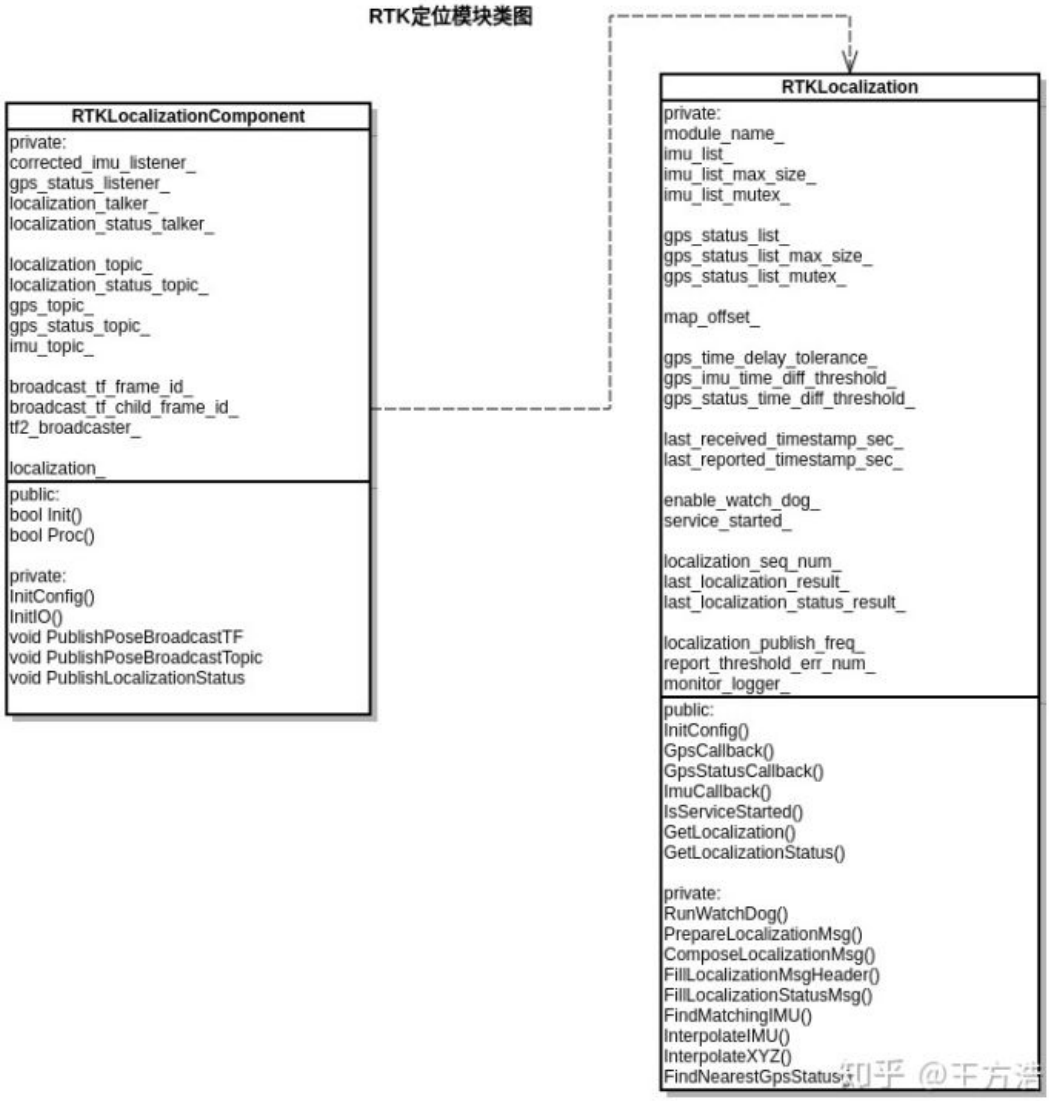
```
├──── BUILD                          // bazel 编译文件
├──── rtk_localization.cc            // rtk 定位功能实现模块
├──── rtk_localization_component.cc  // rtk 消息发布模块
├──── rtk_localization_component.h
├──── rtk_localization.h
└──── rtk_localization_test.cc       // 测试
```

在 rtk_localization_component.cc 中可以看到

```
RTKLocalizationComponent::RTKLocalizationComponent()
    : localization_(new RTKLocalization()) {}
```

即 class RTKLocalization()实际上是 RTKLocalizationComponent()中的一个属性 localization_，我们可以发现 apollo 模块的架构大部分都是这样，一个模块负责发布接收消息，一个模块负责实现具体的功能，后面的模块在前面的模块中注册为一个属性。

具体的类图如下：



其中 RTKLocalization 首先读取驱动模块发布的 gnss 消息，然后再调用 GpsCallback 输出位置。

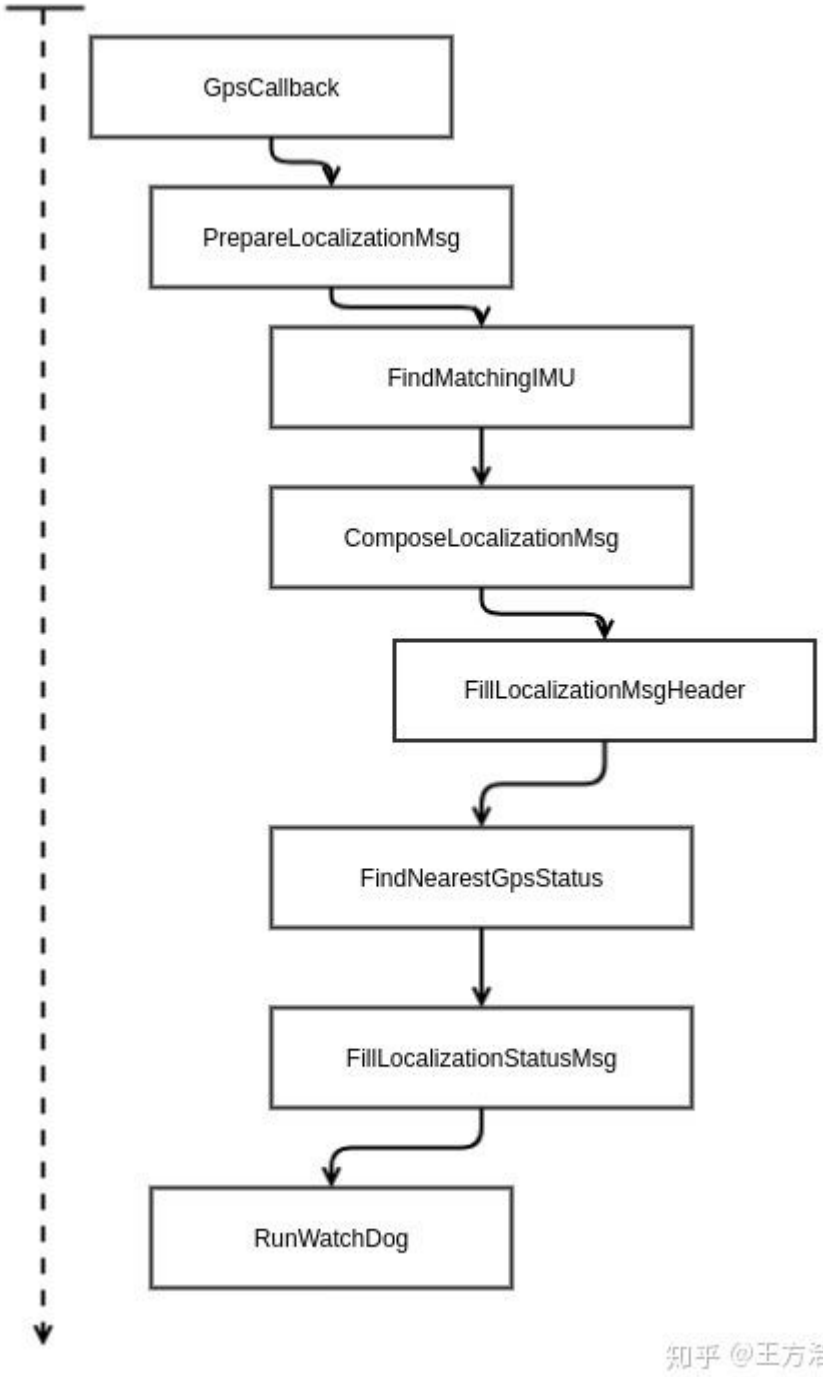
1.通过回调读取 gnss 驱动发布的消息

在 gnss driver 中 DataParser 类通过 PublishCorrimu 发布 IMU 的消息，而 RTKLocalizationComponent::InitIO 中绑定了回调

```
1 corrected_imu_listener_ = node_->CreateReader<localization::CorrectedImu>(
2                                 imu_topic_,
std::bind(&RTKLocalization::ImuCallback, localization_.get(),
3                                         std::placeholders::_1));
```

当读取 imu_topic_的消息时候，调用 ImuCallback 回调，把数据放到 imu_list_中。

2. 通过 GpsCallback 返回位置信息，下面是 GpsCallback 的函数调用顺序



其中"rtk_localization_component.cc"注册为标准的 cyber 模块，RTK 定位模块在"Init"中初始化，每当接收到"localization::Gps"消息就触发执行"Proc"函数。

```
class RTKLocalizationComponent final
        : public cyber::Component<localization::Gps>
{
public:
    RTKLocalizationComponent();
    ~RTKLocalizationComponent() = default;
    bool Init() override;
    bool Proc(const std::shared_ptr<localization::Gps> &gps_msg) override;
```

下面我们分别查看这 2 个函数。

1. Init 函数
   Init 函数实现比较简单，一是初始化配置信息，二是初始化 IO。初始化配置信息主要是读取一些配置，例如一些 topic 信息等。下面主要看下初始化 IO。

```
bool RTKLocalizationComponent::InitIO()
{
    // 1.读取IMU信息，每次接收到localization::CorrectedImu消息，则回调执行“RTKLocalization::ImuCallback”
    corrected_imu_listener_ = node_->CreateReader<localization::CorrectedImu>(
                                    imu_topic_,
std::bind(&RTKLocalization::ImuCallback, localization_.get(),
                                        std::placeholders::_1));
    CHECK(corrected_imu_listener_);
    // 2.读取GPS状态信息，每次接收到GPS状态消息，则回调执行"RTKLocalization::GpsStatusCallback"
    gps_status_listener_ = node_->CreateReader<drivers::gnss::InsStat>(
                                gps_status_topic_,
std::bind(&RTKLocalization::GpsStatusCallback,
                                    localization_.get(),
std::placeholders::_1));
    CHECK(gps_status_listener_);

    // 3.发布位置信息和位置状态信息
    localization_talker_ =
        node_->CreateWriter<LocalizationEstimate>(localization_topic_);
    CHECK(localization_talker_);

    localization_status_talker_ =
        node_->CreateWriter<LocalizationStatus>(localization_status_topic_);
    CHECK(localization_status_talker_);
    return true;
}
```

也就是说，RTK 模块同时还接收 IMU 和 GPS 的状态信息，然后触发对应的回调函数。具体的实现在"RTKLocalization"类中，我们先看下回调的具体实现。以"GpsStatusCallback"为例，

每次读取到 gps 状态信息之后，会把信息保存到"gps_status_list_"列表中。"ImuCallback"类似，也是接收到 IMU 消息后，保存到"imu_list_"列表中。

```
void RTKLocalization::GpsStatusCallback(
    const std::shared_ptr<drivers::gnss::InsStat> &status_msg)
{
    std::unique_lock<std::mutex> lock(gps_status_list_mutex_);
    if (gps_status_list_.size() < gps_status_list_max_size_)
    {
        gps_status_list_.push_back(*status_msg);
    }
    else
    {
        gps_status_list_.pop_front();
        gps_status_list_.push_back(*status_msg);
    }
}
```

2. Proc
   在每次接收到"localization::Gps"消息后，触发执行"Proc"函数。这里注意如果需要接收多个消息，这里是 3 个消息，则选择最慢的消息作为触发，否则，如果选择比较快的消息作为触发，这样会导致作为触发的消息刷新了，而其它的消息还没有刷新。所以这里采用的是 GPS 消息作为触发消息，IMU 的消息刷新快。下面我们看具体的实现。

```cpp
bool RTKLocalizationComponent::Proc(
    const std::shared_ptr<localization::Gps>& gps_msg)
{
    // 1. 通过RTKLocalization处理GPS消息回调
    localization_->GpsCallback(gps_msg);

    if (localization_->IsServiceStarted())
    {
        LocalizationEstimate localization;
        // 2. 获取定位消息
        localization_->GetLocalization(&localization);
        LocalizationStatus localization_status;
        // 3. 获取定位状态
        localization_->GetLocalizationStatus(&localization_status);

        // publish localization messages
        // 4. 发布位置信息
        PublishPoseBroadcastTopic(localization);
        // 5. 发布位置转换信息
        PublishPoseBroadcastTF(localization);
        // 6. 发布位置状态信息
        PublishLocalizationStatus(localization_status);
        ADEBUG << "[OnTimer]: Localization message publish success!";
    }

    return true;
}
```

具体的执行过程如下图所示。
主要的执行过程在"GpsCallback"中，然后通过"GetLocalization"和"GetLocalizationStatus"获取结果，最后发布对应的位置信息、位置转换信息和位置状态信息。
由于"GpsCallback"主要执行过程在"PrepareLocalizationMsg"中，因此我们主要分析"PrepareLocalizationMsg"的实现。

获取定位信息

PrepareLocalizationMsg 函数的具体实现如下。

```cpp
void RTKLocalization::PrepareLocalizationMsg(
    const localization::Gps &gps_msg, LocalizationEstimate *localization,
    LocalizationStatus *localization_status)
{
    // find the matching gps and imu message
    double gps_time_stamp = gps_msg.header().timestamp_sec();
    CorrectedImu imu_msg;
    // 1.寻找最匹配的IMU信息
    FindMatchingIMU(gps_time_stamp, &imu_msg);
    // 2.根据GPS和IMU信息，给位置信息赋值
    ComposeLocalizationMsg(gps_msg, imu_msg, localization);

    drivers::gnss::InsStat gps_status;
    // 3.查找最近的GPS状态信息
    FindNearestGpsStatus(gps_time_stamp, &gps_status);
    // 4.根据GPS状态信息，给位置状态信息赋值
    FillLocalizationStatusMsg(gps_status, localization_status);
}
```

下面我们逐个分析上述 4 个过程。

FindMatchingIMU

在队列中找到最匹配的 IMU 消息，其中区分了队列的第一个，最后一个，以及如果在中间位置则进行插值。插值的时候根据距离最近的原则进行反比例插值。

```cpp
bool RTKLocalization::FindMatchingIMU(const double gps_timestamp_sec,
                                      CorrectedImu *imu_msg)
{

    // 加锁，这里有疑问，为什么换个变量就没有锁了呢？
    std::unique_lock<std::mutex> lock(imu_list_mutex_);
    auto imu_list = imu_list_;
    lock.unlock();

    // 在IMU队列中找到最新的IMU消息
    // scan imu buffer, find first imu message that is newer than the given
    // timestamp
    auto imu_it = imu_list.begin();
    for (; imu_it != imu_list.end(); ++imu_it)
    {
        if ((*imu_it).header().timestamp_sec() - gps_timestamp_sec >
                std::numeric_limits<double>::min())
        {
            break;
        }
    }

    if (imu_it != imu_list.end())    // found one
    {
        if (imu_it == imu_list.begin())
        {
            AERROR << "IMU queue too short or request too old. "
             << "Oldest timestamp[" << imu_list.front().header().timestamp_sec()
             << "], Newest timestamp["
             << imu_list.back().header().timestamp_sec() << "], GPS timestamp["
                   << gps_timestamp_sec << "]";
            *imu_msg = imu_list.front();  // the oldest imu
        }
        else
        {
            // here is the normal case
            auto imu_it_1 = imu_it;
            imu_it_1--;
            if (!(*imu_it).has_header() || !(*imu_it_1).has_header())
            {
                AERROR << "imu1 and imu_it_1 must both have header.";
                return false;
            }
            // 根据最新的IMU消息和它之前的消息做插值。
```

接下来我们看线性插值

1. InterpolateIMU
   根据上述函数得到 2 个 IMU 消息分别对角速度、线性加速度、欧拉角进行插值。原则是根据比例，反比例进行插值。

```
bool RTKLocalization::InterpolateIMU(const CorrectedImu &imu1,
                                     const CorrectedImu &imu2,
                                     const double timestamp_sec,
                                     CorrectedImu *imu_msg)
{
    if (timestamp_sec - imu1.header().timestamp_sec() <
            std::numeric_limits<double>::min())
    {
        AERROR << "[InterpolateIMU1]: the given time stamp[" << timestamp_sec
               << "] is older than the 1st message["
               << imu1.header().timestamp_sec() << "]";
        *imu_msg = imu1;
    }
    else if (timestamp_sec - imu2.header().timestamp_sec() >
             std::numeric_limits<double>::min())
    {
        AERROR << "[InterpolateIMU2]: the given time stamp[" << timestamp_sec
               << "] is newer than the 2nd message["
               << imu2.header().timestamp_sec() << "]";
        *imu_msg = imu1;
    }
    else
    {
        // 线性插值
        *imu_msg = imu1;
        imu_msg->mutable_header()->set_timestamp_sec(timestamp_sec);

        double time_diff =
            imu2.header().timestamp_sec() - imu1.header().timestamp_sec();
        if (fabs(time_diff) >= 0.001)
        {
            double frac1 =
                (timestamp_sec - imu1.header().timestamp_sec()) / time_diff;
            // 1. 分别对角速度、线性加速度、欧拉角进行插值
            if (imu1.imu().has_angular_velocity() &&
                    imu2.imu().has_angular_velocity())
            {
                auto val = InterpolateXYZ(imu1.imu().angular_velocity(),
                                          imu2.imu().angular_velocity(), frac1);

imu_msg->mutable_imu()->mutable_angular_velocity()->CopyFrom(val);
            }
            ...
        }
```

2. InterpolateXYZ
   根据距离插值，反比例，即 frac1 越小，则越靠近 p1，frac1 越大，则越靠近 p2

```
template <class T>
T RTKLocalization::InterpolateXYZ(const T &p1, const T &p2,
                                  const double frac1)
{
    T p;
    double frac2 = 1.0 - frac1;
    if (p1.has_x() && !std::isnan(p1.x()) && p2.has_x() && !std::isnan(p2.x()))
    {
        p.set_x(p1.x() * frac2 + p2.x() * frac1);
    }
    if (p1.has_y() && !std::isnan(p1.y()) && p2.has_y() && !std::isnan(p2.y()))
    {
        p.set_y(p1.y() * frac2 + p2.y() * frac1);
    }
    if (p1.has_z() && !std::isnan(p1.z()) && p2.has_z() && !std::isnan(p2.z()))
    {
        p.set_z(p1.z() * frac2 + p2.z() * frac1);
    }
    return p;
}
```

ComposeLocalizationMsg

填充位置信息，这里实际上涉及到姿态解算，具体是根据 GPS 和 IMU 消息对位置信息进行赋值。需要注意需要根据航向对 IMU 的信息进行转换。

```
void RTKLocalization::ComposeLocalizationMsg(
    const localization::Gps &gps_msg, const localization::CorrectedImu &imu_msg,
    LocalizationEstimate *localization)
{
    localization->Clear();

    FillLocalizationMsgHeader(localization);

    localization->set_measurement_time(gps_msg.header().timestamp_sec());

    // combine gps and imu
    auto mutable_pose = localization->mutable_pose();
    // GPS消息包含位置信息
    if (gps_msg.has_localization())
    {
        const auto &pose = gps_msg.localization();
        // 1. 获取位置
        if (pose.has_position())
        {
            // position
            // world frame -> map frame
            mutable_pose->mutable_position()->set_x(pose.position().x() -
                                                    map_offset_[0]);
            mutable_pose->mutable_position()->set_y(pose.position().y() -
                                                    map_offset_[1]);
            mutable_pose->mutable_position()->set_z(pose.position().z() -
                                                    map_offset_[2]);
        }
        // 2. 获取方向
        // orientation
        if (pose.has_orientation())
        {
            mutable_pose->mutable_orientation()->CopyFrom(pose.orientation());
            double heading = common::math::QuaternionToHeading(
            pose.orientation().qw(), pose.orientation().qx(),
            pose.orientation().qy(), pose.orientation().qz());
            mutable_pose->set_heading(heading);
        }
        // linear velocity
        // 3. 获取速度
        if (pose.has_linear_velocity())
        {

mutable_pose->mutable_linear_velocity()->CopyFrom(pose.linear_velocity());
```

FindNearestGpsStatus

获取最近的 Gps 状态信息，这里实现的算法是遍历查找离"gps_time_stamp"最近的状态，GPS 状态信息不是按照时间顺序排列的？

```
bool RTKLocalization::FindNearestGpsStatus(const double gps_timestamp_sec,
        drivers::gnss::InsStat *status)
{
    ...
    // 1. 遍历查找最近的GPS状态信息
    double timestamp_diff_sec = 1e8;
    auto nearest_itr = gps_status_list.end();
    for (auto itr = gps_status_list.begin(); itr != gps_status_list.end();
            ++itr)
    {
        double diff = std::abs(itr->header().timestamp_sec() - gps_timestamp_sec);
        if (diff < timestamp_diff_sec)
        {
            timestamp_diff_sec = diff;
            nearest_itr = itr;
        }
    }
    ...
}
```

FillLocalizationStatusMsg

获取位置状态，一共有 3 种状态：稳定状态(INS_RTKFIXED)、浮动状态(INS_RTKFLOAT)、错误状态(ERROR)。由于代码比较简单，这里就不分析了。

发布消息

最后通过以下几个函数发布消息。

1. PublishPoseBroadcastTopic // 发布位置信息
2. PublishPoseBroadcastTF // 发布位置转换 transform 信息
3. PublishLocalizationStatus // 发布位置状态信息

以上就是整个 RTK 的定位流程，主要的思路是通过接收 GPS 和 IMU 信息结合输出无人车的位置信息，这里还有一个疑问是为什么最后输出的定位信息的位置是直接采用的 GPS 的位置信息，没有通过 IMU 信息对位置信息做解算，还是说在其它模块中实现的？

如何添加新的 GPS 接收器

简介

GPS 接收器是一种从 GPS 卫星上接收信息,然后根据这些信息计算设备地理位置、速度和精确时间的设备。这种设备通常包括一个接收器，一个 IMU（Inertial measurement unit，惯性测量单元），一个针对轮编码器的接口以及一个将各传感器获取的数据融合到一起的融合引擎。Apollo 系统中默认使用 Novatel 板卡，该说明详细介绍如何添加并使用一个新的 GPS 接收器。

添加 GPS 新接收器的步骤

请按照下面的步骤添加新的 GPS 接收器.

通过继承基类“Parser”，实现新 GPS 接收器的数据解析器

在 Parser 类中为新 GPS 接收器添加新接口

在文件 config.proto 中，为新 GPS 接收器添加新数据格式

在函数 create_parser（见文件 data_parser.cpp），为新 GPS 接收器添加新解析器实例

下面让我们用上面的方法来添加 u-blox GPS 接收器。

步骤 1

通过继承类“Parser”，为新 GPS 接收器实现新的数据解析器:

```
class UbloxParser : public Parser
{
public:
    UbloxParser();
    virtual MessageType get_message(MessagePtr& message_ptr);
private:
    bool verify_checksum();
    Parser::MessageType prepare_message(MessagePtr& message_ptr);
    // The handle_xxx functions return whether a message is ready.
    bool handle_esf_raw(const ublox::EsfRaw* raw, size_t data_size);
    bool handle_esf_ins(const ublox::EsfIns* ins);
    bool handle_hnr_pvt(const ublox::HnrPvt* pvt);
    bool handle_nav_att(const ublox::NavAtt *att);
    bool handle_nav_pvt(const ublox::NavPvt* pvt);
    bool handle_nav_cov(const ublox::NavCov *cov);
    bool handle_rxm_rawx(const ublox::RxmRawx *raw);
    double _gps_seconds_base = -1.0;
    double _gyro_scale = 0.0;
    double _accel_scale = 0.0;
    float _imu_measurement_span = 0.0;
    int _imu_frame_mapping = 5;
    double _imu_measurement_time_previous = -1.0;
    std::vector<uint8_t> _buffer;
    size_t _total_length = 0;
    ::apollo::drivers::gnss::Gnss _gnss;
    ::apollo::drivers::gnss::Imu _imu;
    ::apollo::drivers::gnss::Ins _ins;
};
```

步骤 2

在 Parser 类中，为新 GPS 接收器添加新的接口:

在 Parser 类中添加函数‘create_ublox‘:

```
class Parser
{
public:
    // Return a pointer to a NovAtel parser. The caller should take ownership.
    static Parser* create_novatel();

    // Return a pointer to a u-blox parser. The caller should take ownership.
    static Parser* create_ublox();

    virtual ~Parser() {}

    // Updates the parser with new data. The caller must keep the data valid until
get_message()
    // returns NONE.
    void update(const uint8_t* data, size_t length)
    {
        _data = data;
        _data_end = data + length;
    }

    void update(const std::string& data)
    {
        update(reinterpret_cast<const uint8_t*>(data.data()), data.size());
    }

    enum class MessageType
    {
        NONE,
        GNSS,
        GNSS_RANGE,
        IMU,
        INS,
        WHEEL,
        EPHEMERIDES,
        OBSERVATION,
        GPGGA,
    };

    // Gets a parsed protobuf message. The caller must consume the message before
calling another
    // get_message() or update();
    virtual MessageType get_message(MessagePtr& message_ptr) = 0;

protected:
```

步骤 3

在 config.proto 文件中，为新的 GPS 接收器添加新的数据格式定义:

在配置文件（modules/drivers/gnss/proto/config.proto）中添加 UBLOX_TEXT and UBLOX_BINARY

```
message Stream {
    enum Format {
        UNKNOWN = 0;
        NMEA = 1;
        RTCM_V2 = 2;
        RTCM_V3 = 3;

        NOVATEL_TEXT = 10;
        NOVATEL_BINARY = 11;

        UBLOX_TEXT = 20;
        UBLOX_BINARY = 21;
    }
    ... ...
```

步骤 4

在函数 create_parser（见 data_parser.cpp），为新 GPS 接收器添加新解析器实例. 我们将通过添加处理 config::Stream::UBLOX_BINARY 的代码实现上面的步骤，具体如下。

```
Parser* create_parser(config::Stream::Format format, bool is_base_station = false)
{
    switch (format)
    {
    case config::Stream::NOVATEL_BINARY:
        return Parser::create_novatel();

    case config::Stream::UBLOX_BINARY:
        return Parser::create_ubloxl();

    default:
        return nullptr;
    }
}
```