开发者说 | Apollo 3.5 车辆配置方案

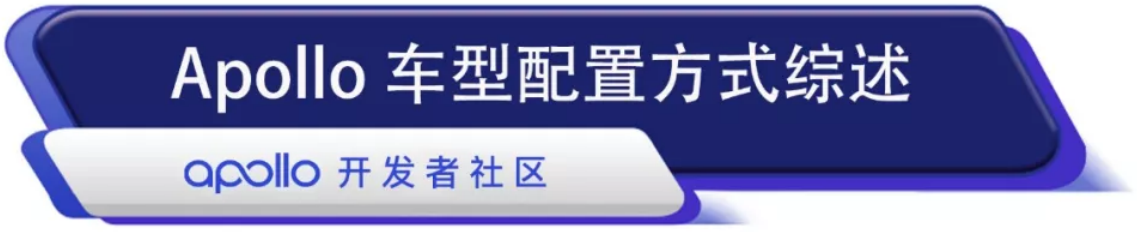
Apollo 开放车辆的接口标准主要涉及到两大部分，即**线控系统**和**车辆系统**。Apollo 对这两者的功能指标、性能指标、安全指标进行一系列的约定并提出了相关标准。以常见的刹车和油门为例，Apollo 对这两者的控制精度、控制力度、系统的周期时间、响应时间都有着严格的规定。

线控系统对指令越界保护和控制的处理等安全指标都有着明确约定以及标准化的要求。而**车辆系统**要求有相对稳定的CAN信号通道，同时对于车辆电源，包括电压、功率、最大波动、输出误差都有一系列的规定，以够保证在整个自动驾驶过程中电源输出稳定。

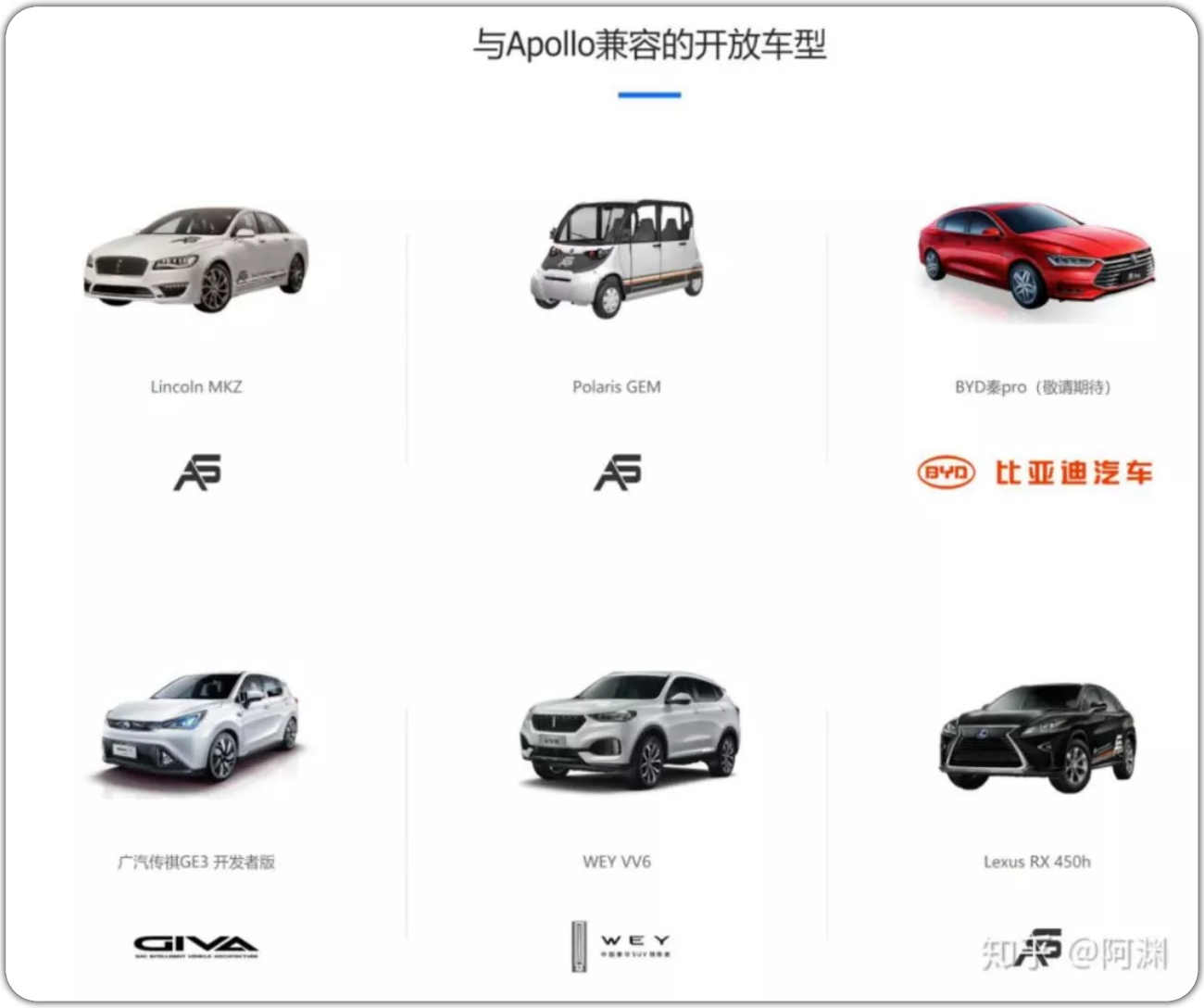
本文由**Apollo开发者社区认证布道师-阿渊**撰写，对**Apollo 3.5 车辆配置方案**进行了详细讲解，希望这篇文章给感兴趣的同学带来更多帮助。

以下，ENJOY

最近在研究百度无人车 Apollo 的工厂模式及车辆配置方式，有一些小心得希望和大家一起分享。



Apollo 无人驾驶平台支持 Lincoln MKZ、WEY VV6 等来自多个 OEM 的不同车型。



Apollo 兼容的开放车型，来源： http://apollo.auto/vehicle/certificate_cn.html

众所周知，各车厂车型的配置方式、接口、信号都各不相同。那么 Apollo 是如何兼容各个车型的呢？本文将从以下三个层次来回答这个问题。

平台构架

apollo 开发者社区

从平台构架上看，Apollo 借助“**开放车辆认证平台 (Open Vehicle Certificate Platform)**"完成与汽车的交互，其他上层平台无需关注底层实现。

Apollo 3.5 架构图， 来源：https://github.com/ApolloAuto/apollo

Apollo 的平台架构如上图所示，Apollo 开放平台包括了以下几个部分：

- 云端服务平台
- 开源软件平台
- 硬件开发平台
- 开放车辆认证平台

这里我们着重了解一下“开放车辆认证平台”。

来源：http://apollo.auto/developer_cn.html

目前各个 OEM 厂商的大多使用 CAN 总线协议来进行车辆内部各个 ECU 节点之间的通讯。CAN 总线通讯协议中各节点的信息使用 DBC（Database Can）文件来进行来进行描述。

> The DBC file describes the communication of a single CAN network. This information is sufficient to monitor and analyze the network and to simulate nodes not physically available.
>
> DBC文件描述了单个CAN网络的通信。 此信息足以监视和分析网络并模拟物理上不可用的节点。[1]

各车厂的 DBC 文件定义通常并不相同，并且是严格保密的。为了解决开发者在开发无人驾驶系统中与车辆交互的问题，Apollo 搭建了 **《开放汽车认证平台》**，并提出了开放车辆认证计划。

> 开放车辆认证计划第一次在业内提出标准化的无人驾驶系统与车辆接口，透过这个计划，车企/车辆提供商可以更方便的将车辆平台接入到Apollo开放平台，从而覆盖更广泛的无人驾驶开发者人群，加速无人驾驶能力的上车部署。[2]

该平台作为软硬件中间层，**提出了开放车辆接口标准，定义了系统与汽车的线控接口，负责完成系统与汽车的具体交互。该平台抽象出了与车型无关的信号作为上层算法模块的输入，使得上层平台可以与底层车辆信号解耦。**

Apollo 的开放车辆接口标准定义了 Apollo 需要的诸多用于控制车辆和接收反馈的信号。大体而言， Apollo 需要车企提供**线控转向**、**驱动**、**制动**、**档位**、**驻车**、**灯光**、**雨刮控制**、**喇叭控制**等控制及故障反馈等接口。Apollo 乘用车的线控需求具体的详细信息可参见下列规范。

https://link.zhihu.com/?target=http%3A//apollo-homepage.bj.bcebos.com/Apollo_by_wire_requirement.xlsx

此外，根据**《开放车辆认证车企认证流程》**，想要接入到 Apollo 开放平台，车企需要遵循 Apollo 的接口规范，向 Apollo 开放平台提供对应的 DBC 文件。

Apollo 在与**开放车辆的信号交互**上和**开放车辆配置**上均使用了**Protobuf**。

> Protocol Buffers 是一种轻便高效的结构化数据存储格式，可以用于结构化数据串行化，或者说序列化。它很适合做数据存储或 RPC 数据交换格式。可用于通讯协议、数据存储等领域的语言无关、平台无关、可扩展的序列化结构数据格式。目前提供了 C++、Java、Python 三种语言的 API。**[3]**

通常车企会使用 DBC 文件来完成 CAN 信号的定义和解析， Apollo 则大量使用了 Protobuf 来进行模块间的通信和配置，因此车企需要使用 Apollo 提供的工具基于 DBC 来生成 Apollo 可用的 Proto 文件 (如下所示)。

```
// modules/canbus/proto/wey.proto
message Wey {
optional Ads_shifter_115 ads_shifter_115 = 1; // control message
```

```
4   optional Ads_eps_113 ads_eps_113 = 2; // control message
5   optional Status_310 status_310 = 3; // report message
6   optional Vin_resp3_393 vin_resp3_393 = 4; // report message
7   optional Vin_resp2_392 vin_resp2_392 = 5; // report message
8   optional Vin_resp1_391 vin_resp1_391 = 6; // report message
9   optional Ads_req_vin_390 ads_req_vin_390 = 7; // control message
10  optional Ads1_111 ads1_111 = 8; // control message
11  optional Fbs2_240 fbs2_240 = 9; // report message
12  optional Fbs1_243 fbs1_243 = 10; // report message
13  optional Fbs4_235 fbs4_235 = 11; // report message
14  optional Fail_241 fail_241 = 12; // report message
15  optional Fbs3_237 fbs3_237 = 13; // report message
16  optional Ads3_38e ads3_38e = 14; // control message
17  }
```

Protobuf 提供的 Codegen 工具会根据 Proto 文件中定义的变量生成可直接使用的 C++ 代码，十分便捷。

车辆配置

Protobuf 提供了一种名为**TextFormat**的序列化格式，该格式类似于Json，清晰易懂。配合事先定义的 Proto 文件， 开发者可以轻易实现从**可读的配置文件**到**具体对象的实例**的反射，配置文件经过反序列化后可以作为业务代码类的输入。这种方式使得配置变得便捷，不易出错，且具有很好的向后兼容性。

Apollo 的代码中大量使用了这种方式来管理配置。Apollo 激活车辆的配置文件为 `modules/canbus/conf/canbusconf.pb.txt`， 开发者可以在这里定义车型及对应的 CAN card 的参数，开发者只需修改 "`vehicle_parameter`" 相应的字段，即可使 Apollo 支持对应的车型。

```
1   # modules/canbus/conf/canbus_conf.pb.txt
2   vehicle_parameter {
3   brand: LINCOLN_MKZ
4   max_enable_fail_attempt: 5
5   driving_mode: COMPLETE_AUTO_DRIVE
6   }
7
8   can_card_parameter {
9   brand: ESD_CAN
10  type: PCI_CARD
11  channel_id: CHANNEL_ID_ZERO
```

```
}

enable_debug_mode: false
enable_receiver_log: false
enable_sender_log: false
```

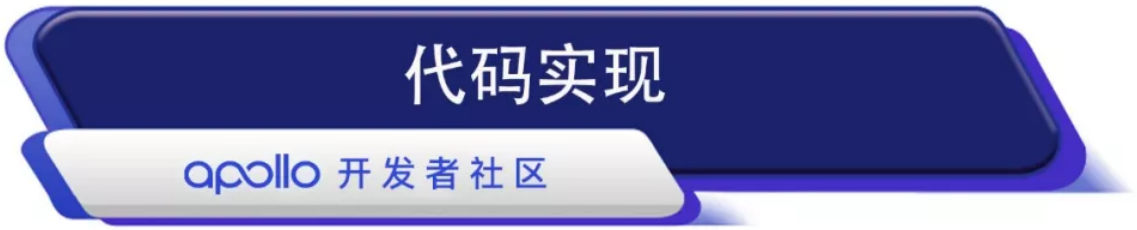
上述配置文件的参数的含义是由下面的 Proto 文件决定的。

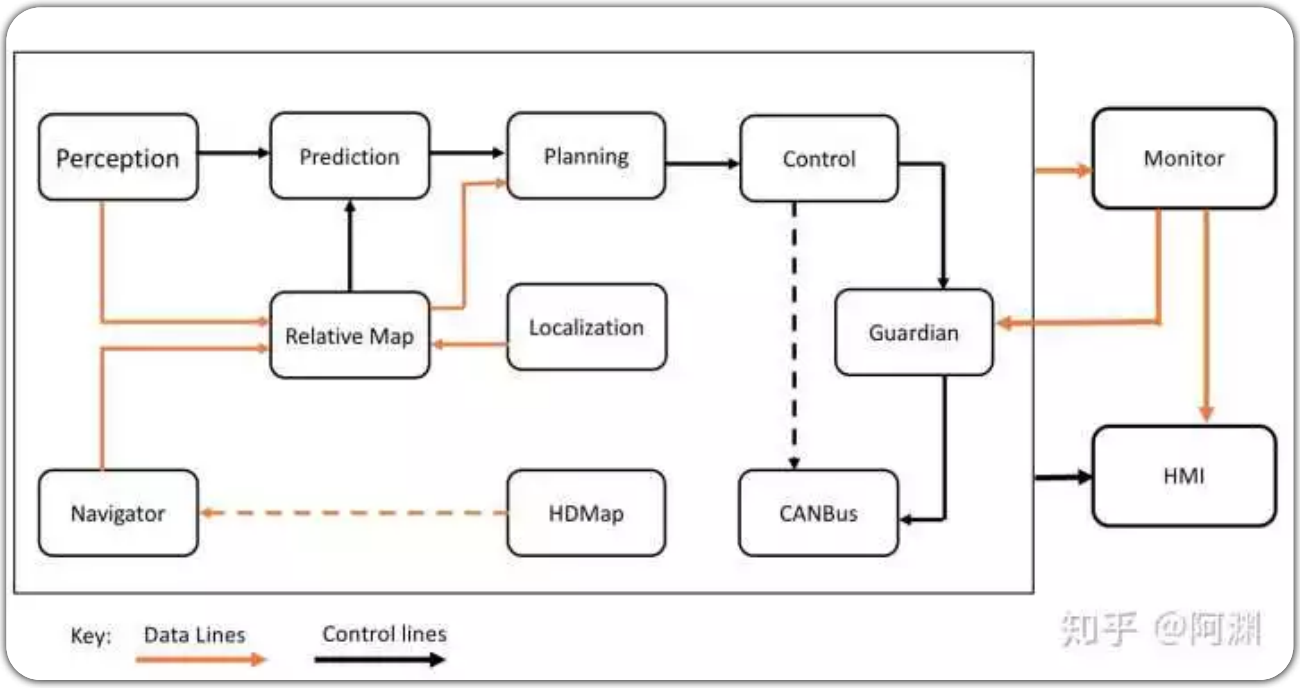
```
// modules/canbus/proto/canbus_conf.proto
message CanbusConf {
optional apollo.canbus.VehicleParameter vehicle_parameter = 1;
optional apollo.drivers.canbus.CANCardParameter can_card_parameter = 2;
optional bool enable_debug_mode = 3 [default = false];
optional bool enable_receiver_log = 4 [default = false];
optional bool enable_sender_log = 5 [default = false];
}

//modules/canbus/proto/vehicle_parameter.proto
// Apollo 支持了 LINCON_MKZ, GEM, LEXUS 等多种车型
message VehicleParameter {
enum VehicleBrand {
LINCOLN_MKZ = 0;
GEM = 1;
LEXUS = 2;
TRANSIT = 3;
GE3 = 4;
WEY = 5;
}
optional VehicleBrand brand = 1;
optional double max_engine_pedal = 2;
optional int32 max_enable_fail_attempt = 3;
optional Chassis.DrivingMode driving_mode = 4;
}

// modules/drivers/canbus/proto/can_card_parameter.proto
message CANCardParameter {
enum CANCardBrand {
FAKE_CAN = 0;
ESD_CAN = 1;
SOCKET_CAN_RAW = 2;
HERMES_CAN = 3;
}
...
}
```

另外要提到一点的是，Protobuf 提供了两个版本的库，即精简版 ("`libprotobuf-lite.so`") 和完整版 ("`libprotobuf.so`")。

> The "lite" library is much smaller than the full library, and is more appropriate for resource-constrained systems such as mobile phones.
>
> 精简版体积远小于完整版，因此更适合使用在诸如移动电话等资源受限的系统上。[4]

精简版的 Protobuf 常用于嵌入式设备，但精简版的库并不支持 TextFormat 的反射功能。开发者如果想兼具代码体积和功能的话，可以考虑自己写一套格式化语言的反射机制，有兴趣的同学可以参考**《简单的 C++ 结构体字段反射》**。





Apollo Software Overview, 来源：https://github.com/ApolloAuto/apollo

从软件实现上看，Apollo 通过**CANBus**模块来实现对车辆的管理和通讯 。

CANBus 模块接收并执行来自 Control 模块的指令，同时收集汽车底盘的状态，这些状态是 Apollo 抽象出的一组与车型无关的信号。Canbus 模块处理这些状态与各个汽车底盘信号的映射关系，随后将这些状态反馈回 Control 模块。基于这样的设计，Apollo 得以兼容多个不同的车型。

chassis.proto文件对 Apollo 抽象出的信号进行了定义，大体包括下列信息：

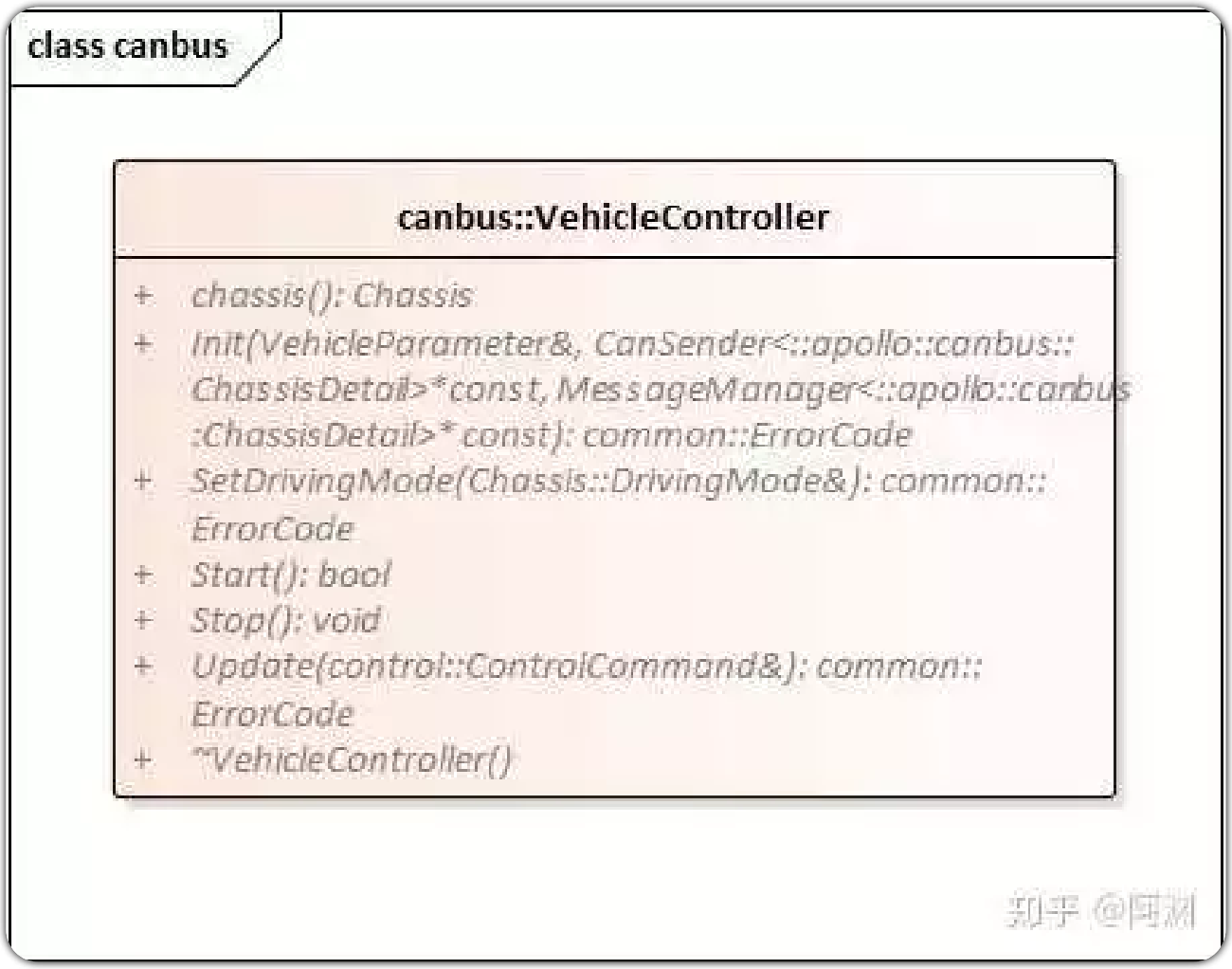
Chassis 信号

CANBus 模块主要由以下两个部件组成

> **Vehicle:** the vehicle itself, including its **controller** and **message manager**
>
> **CAN Client** - CAN client has been moved to `/modules/drivers/canbus` since it is shared by different sensors utilizing the canbus protocol **[5]**

在这里着重介绍一下**Vehicle** 部分。

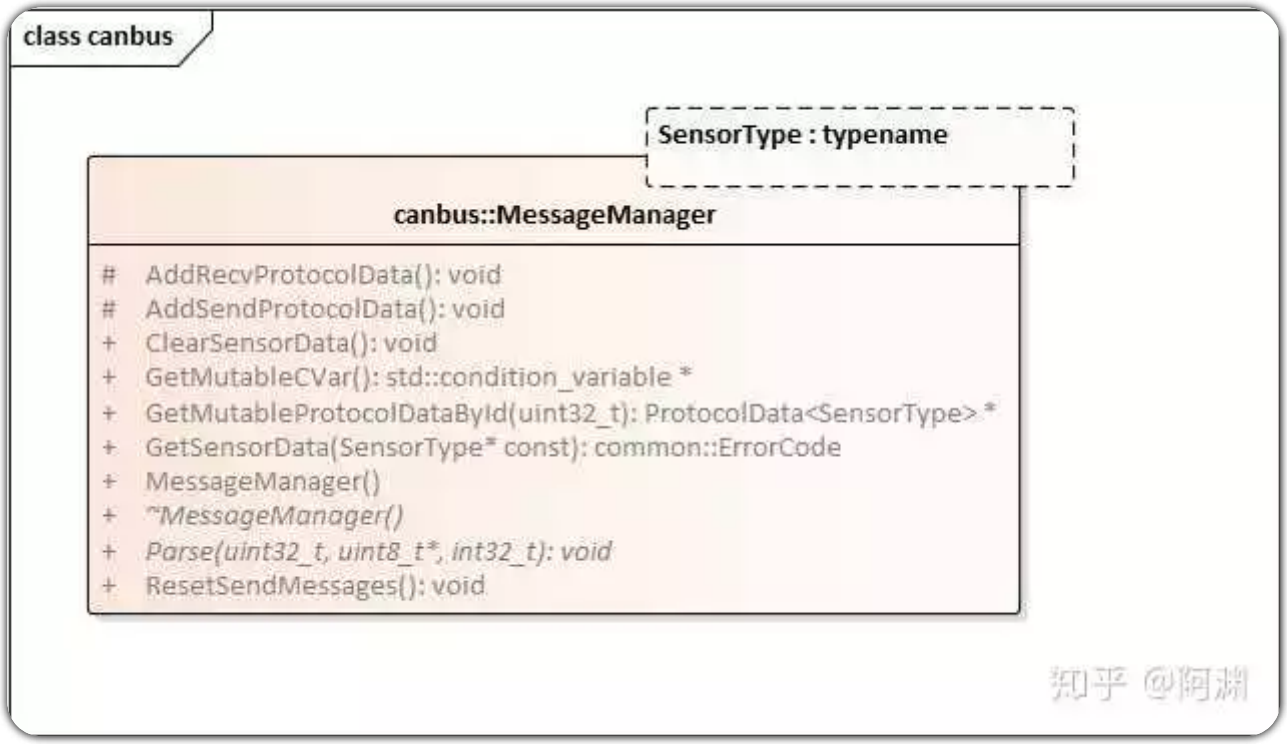
Vehicle的Controller(`modules/canbus/vehicle/vehicle_controller.h`) 的类图如下(有简化)：



Vehicle Controller 类负责完成与汽车底盘的具体交互，下面对部分公有接口做一些解释。

```
/**   * @brief start the vehicle controller.   * 注：该函数会在内部起一个名   为"SecurityDog
virtual bool Start() = 0;

/**   * @brief stop the vehicle controller.   */
virtual void Stop() = 0;

/**   * @brief calculate and return the chassis.   * 注：该函数完成了汽车底盘信号和 Apollo 内
virtual Chassis chassis() = 0;

/**   * @brief update the vehicle controller.   * 注：该函数负责执行来自 Control 模块的具体的
virtual common::ErrorCode Update(const control::ControlCommand &command);
```

Vehicle 的 MessageManager 类负责完成对具体信号的接收、发送、解析等，其类图如下：




```
// modules/drivers/canbus/can_comm/message_manager.h
// 用于指定系统向汽车底盘发送的控制型号
template <typename SensorType>
template <class T, bool need_check>
void MessageManager::AddSendProtocolData();

// 用于指定系统接收的信号
template <typename SensorType>
template <class T, bool need_check>
void MessageManager::AddRecvProtocolData();
```

接下来我们以 Wey VV6 车型为例，来分析 Apollo 是如何在代码层面上完成配置任务的。

Wey 文件夹包含有如下文件：

根据 Apollo 的官方文件`how_to_add_a_new_vehicle`， 想要为 Apollo 添加 Wey 车型需要完成以下内容：

- 实现新的车辆控制器--**wey_controller.cc**，继承VehicleController类
- 实现新的消息管理器--**wey_message_manager.cc** 继承MessageManager 类
- 实现新的车辆工厂类--**wey_vehicle_factory.cc**, 继承AbstractVehicleFactory类
- 更新配置文件
  - 在`modules/canbus/vehicle/vehicle_factory.cc`中进行注册
  - 更新配置文件`modules/canbus/conf/canbus_conf.pb.txt`

通过上述方式可以增加新车型的原因在于 Apollo 的配置是基于工厂模式实现的。

> 工厂方法模式（Factory method pattern）是一种实现了“工厂”概念的**面向对象设计模式**。就像其他**创建型模式**一样，它也是处理在不指定**对象**具体**类型**的情况下创建对象的问题。工厂方法模式的实质是“定义一个创建对象的接口，但让实现这个接口的类来决定实例化哪个类。工厂方法让类的实例化推迟到子类中进行。”[6]

Canbus 模块中 Vehicle 相关的内容使用工厂模式抽象出了 VehicleController，MessageManager，AbstractVehicleFactory 三个接口。Canbus 的业务代码（`canbus_component.cc`）通过以上接口来操纵具体的对象，用户无需关心具体的对象是什么，从而实现了业务逻辑和目标对象的解耦。

工厂方法模式的定义和实现的相关讲解有很多，本文就不再赘述，可参考下列链接和书籍:

> - **https://en.wikipedia.org/wiki/Factory_method_pattern**
> - **《设计模式：可复用面向对象软件的基础》**
> - **《Head First 设计模式》**

Apollo 社区布道师贺志国老师曾对 Apollo 的工厂模式进行过介绍，接下来本文会在此基础上继续延伸。

https://blog.csdn.net/davidhopper/article/details/79197075

Apollo 提供了一个工厂模版(`modules/common/util/factory.h`)，该模版可支持任何类型的输入，类图如下：

工厂模版

> Factory 类包含了 `Register()`、`Unregister()`、`Empty()`、`CreateObjectOrNull()`、`CreateObject()`等公有函数，其中`Register()`、`Unregister()`函数用于注册和反注册产品类，其作用与经典模式中抽象工厂接口类的功能类似，`Empty()`函数用于判断当前工厂类中是否包含产品创建函数，`CreateObjectOrNull()`、`CreateObject()`函数用于创建可能包含空指针和不包含空指针的产品类对象。[7]

Factory 工厂模版维护了一个 Map 用来管理 IdentifierType 和 ProductCreator 的键值对，根据输入的 IdentifierType， 模版可返回 ProductCreator 生产的产品，从而实现了从 IdentifierType 到 Product 的“映射“。

在Canbus 模块中，工厂类为 "VehicleFactory"， 该类继承于工厂模版 "Factory"。VehicleFactory 工厂维护了键值对为 VehicleParameter::VehicleBrand 和 AbstractVehicleFactory 的 Map。

如下所示，每新注册一种车型，该 Map 中就会插入一条汽车品牌(VehicleBrand)和该品牌汽车生产工厂(AbstractVehicleFactory)的键值对。

```
void VehicleFactory::RegisterVehicleFactory() {
Register(apollo::common::LINCOLN_MKZ, []() -> AbstractVehicleFactory * {
return new LincolnVehicleFactory();
});
Register(apollo::common::GEM, []() -> AbstractVehicleFactory * {
return new GemVehicleFactory();
});
Register(apollo::common::LEXUS, []() -> AbstractVehicleFactory * {
return new LexusVehicleFactory();
});
Register(apollo::common::TRANSIT, []() -> AbstractVehicleFactory * {
return new TransitVehicleFactory();
```

```
13   });
14   Register(apollo::common::GE3, []() -> AbstractVehicleFactory * {
15   return new Ge3VehicleFactory();
16   });
17   Register(apollo::common::WEY, []() -> AbstractVehicleFactory * {
18   return new WeyVehicleFactory();
19   });
20   }
```

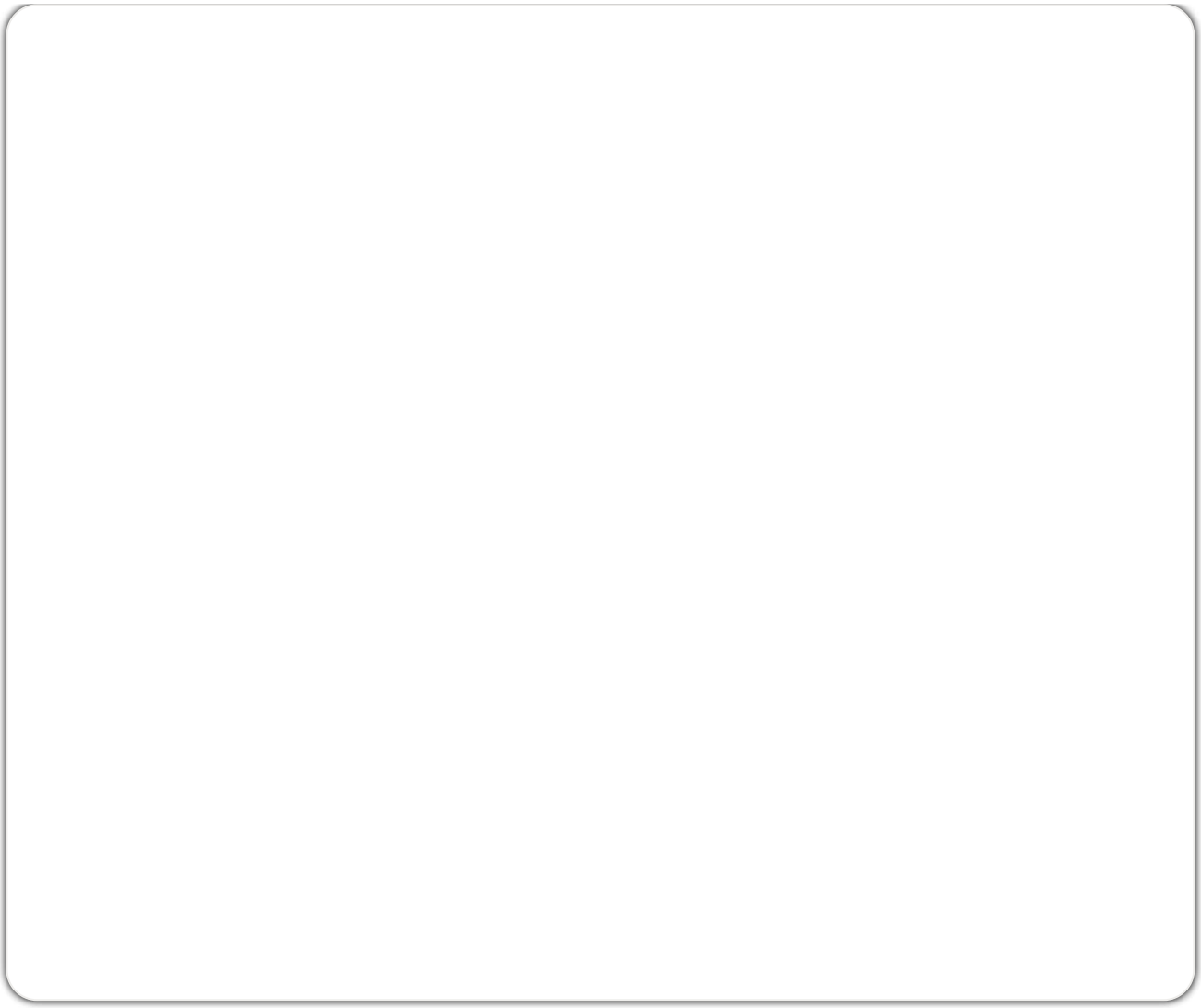
当VehicleFactory 类的"CreateVehicle" 方法被调用时， VehicleFactory 会根据输入的汽车品牌，在 Map 中查找并返回可以生产这种汽车的工厂 。

例如输入汽车品牌"WEY" ， VehicleFactory 会返回 WeyVehicleFactory ，WeyVehicleFactory 继承于 AbstractVehicleFactory 。

```
1   /**
2   * @brief Creates an AbstractVehicleFactory object based on vehicle_parameter
3   * @param vehicle_parameter is defined in vehicle_parameter.proto
4   */
5   std::unique_ptr CreateVehicle(
6   const VehicleParameter &vehicle_parameter);
```

AbstracVehicleFactory 工厂会产出一组适用于该品牌车型的产品即 MessageManager 和 Vehicle controller。

以 “Wey” 为例， WeyVehicleFactory 会产出 WeyMessageManager 和 WeyController 用于实现 “Wey”车型的通讯和控制。
完整的类图如下所示：



最后对 CANBus 模块的**CanbusComponent**进行介绍 。该类继承于 " TimerComponent"，主要作用为处理来自控制模块的控制指令，并将信号消息发送至 Can card。

CanbusComponent 的初始化函数 (`init`)主要完成了以下工作：

1. 读取 CANBus 配置文件

```
1     if (!GetProtoConfig(&canbus_conf_)) {
2       AERROR << "Unable to load canbus conf file: " << ConfigFilePath();
3       return false;
4     }
```

2. 根据配置文件初始化 Can-client.

```
can_client_ = can_factory->CreateCANClient(canbus_conf_.can_card_parameter());
```

3. 根据配置文件获取汽车工厂

```
VehicleFactory vehicle_factory;
vehicle_factory.RegisterVehicleFactory();
auto vehicle_object =
  vehicle_factory.CreateVehicle(canbus_conf_.vehicle_parameter());
```

4. 获取该汽车工厂生产的 message_manager 和 Vehicle_contorller

```
message_manager_ = vehicle_object->CreateMessageManager();
...
// 初始化 can_receiver_ 和 can_sender_
if (can_receiver_.Init(can_client_.get(), message_manager_.get(),
                       canbus_conf_.enable_receiver_log()) != ErrorCode::OK) {...}
if (can_sender_.Init(can_client_.get(), canbus_conf_.enable_sender_log()) !=
  ErrorCode::OK) { ...}

 vehicle_controller_ = vehicle_object->CreateVehicleController();
```

5. 使能 Can 收发和 Vehicle_contorller

初始化完成之后，CanbusComponent 会周期性的报告车身状态，并执行来自 Control 模块和 Guardian 模块的命令。

```
bool CanbusComponent::Proc() {
// publish 底盘信息
PublishChassis();
if (FLAGS_enable_chassis_detail_pub) {
// Publish 底盘的细节信息
PublishChassisDetail();
}
return true;
}

 // 事件触发，执行来自 Control 模块的指令
 void CanbusComponent::OnControlCommand(const ControlCommand &control_command) {...}

 // 事件触发，执行来自 Gurdian 模块的指令
 void CanbusComponent::OnGuardianCommand(
 const GuardianCommand &guardian_command) {
 apollo::control::ControlCommand control_command;
 control_command.CopyFrom(guardian_command.control_command());
```

```
19   OnControlCommand(control_command);
20   }
```

1. **Apollo 开放车辆认证平台定义了系统与线控车辆的接口标准，并且从各个车型中抽象出了用于算法的与具体车型无关的信号。**
2. **在软件模块中， Canbus 模块负责处理这些信号与车辆底盘信号的映射。**
3. **Apollo 以 Protobuf 为基础使得车辆配置管理变得十分简洁易用。**
4. **Apollo 使用抽象工厂模式，使业务逻辑得以与具体的车辆解耦。**

上述方式的综合应用，使得 Apollo 得以支持多种不同的车辆。

本文部分内容参考链接：

*《开放汽车认证平台》
【https://link.zhihu.com/?target=http%3A//apollo.auto/vehicle/certificate_cn.html】
*《开放车辆认证车企认证流程》
【https://link.zhihu.com/?target=http%3A//apollo.auto/docs/procedure_cn.html】
*《简单的 C++ 结构体字段反射》
【https://link.zhihu.com/?target=https%3A//bot-man-jl.github.io/articles/%3Fpost%3D2018/Cpp-Struct-Field-Reflection%26nsukey%3D4g6o6B8COf08Pd%252FUh7E8erUlhov8qcM0lfvkGeYVsTJliShSUCJomyEn9h11RAtAE6OZ4i2Fc71cx7Jyo23Pab6I%252B2zQFvl2d1sHdveshnxHl8s35hNIWhcN4g0Xld9xh0mORjg7mN0WKZWIZaa1jG2a%252FWNjDsgETR0G4IzLwpk%253D】
*【1】《DBC File Format Documentation》
【https://link.zhihu.com/?target=http%3A//read.pudn.com/downloads766/ebook/3041455/DBC_File_Format_Documentation.pdf】

*【2】《开放汽车认证平台》
【https://link.zhihu.com/?target=http%3A//apollo.auto/vehicle/certificate_cn.html】
*【3】《如何在Apollo中添加新的车辆》
【https://link.zhihu.com/?target=https%3A//github.com/ApolloAuto/apollo/blob/master/docs/howto/how_to_add_a_new_vehicle_cn.md】
*【4】《 Protobuf: C++ Generated Code》
【https://link.zhihu.com/?target=https%3A//developers.google.com/protocol-buffers/docs/reference/cpp-generated%3Fcsw%3D1%23message】
*【5】《 CANBus》
【https://link.zhihu.com/?target=https%3A//github.com/ApolloAuto/apollo/tree/master/modules/canbus】
*【6】《工厂设计模式》
【https://link.zhihu.com/?target=https%3A//zh.wikipedia.org/wiki/%25E5%25B7%25A5%25E5%258E%2582%25E6%2596%25B9%25E6%25B3%2595%23cite_note-1】
*【7】《Apollo项目类对象创建之工厂模式分析》
【https://link.zhihu.com/?target=https%3A//blog.csdn.net/davidhopper/article/details/79197075】