第5章 CarPropertyService——车辆属性服务

本章涉及的源码文件名及位置：

- CarPropertyManager.java
  packages/services/Car/car-lib/src/android/car/hardware/property/
- CarInfoManager.java
  packages/services/Car/car-lib/src/android/car/
- CarHvacManager.java
  packages/services/Car/car-lib/src/android/car/hardware/hvac/
- PropertyHalService.java
  packages/services/Car/service/src/com/android/car/hal/
- VehicleHal.java
  packages/services/Car/service/src/com/android/car/hal/
- CarPropertyService.java
  packages/services/Car/service/src/com/android/car/
- HalClient.java
  packages/services/Car/service/src/com/android/car/hal/
- PropertyHalServiceIds.java
  packages/services/Car/service/src/com/android/car/hal/

读者可通过 http://cs.android.com/在线或下载完整 Android 源码进行查看，具体内容可参考第 3 章阅读准备。

第一个不得不提的重要服务就是 CarPropertyService，说它重要是因为绝大部分与车辆硬件功能相关联的属性，如空调、车舱功能、车辆传感器等都是通过 CarPropertyService 来读取或者设置的。

CarPropertyManager 是 CarPropertyService 在客户端的代理，通过 CarPropertyManager 中提供的 API，可以设置和获取车辆各个属性的状态。但在实际使用时，CarPropertyManager 却未必是开发者使用最频繁的对象。尤其是在 Android 9 平台上开发时，当开发者想控制空调相关的功能的时候，也许会使用 CarHvacManager；当想获取车辆信息的时候，也许会使用 CarInfoManager；当想获取车辆传感器数据的时候，会使用 CarSensorManager。但其实无论是 CarHvacManager 还是 CarInfoManager 或是 CarSensorManager，它们最后都会通过 CarPropertyManager 来获取设置属性，在服务端对应的都是 CarPropertyService。通过 ICarImpl 中 getCarService 方法也可以很清楚地发现这一点。源码片段如下：

```
[ICarImpl.java]

   public IBinder getCarService(String serviceName) {
      switch (serviceName) {
         ...

         case Car.CABIN_SERVICE:
         case Car.HVAC_SERVICE:
         case Car.INFO_SERVICE:
         case Car.PROPERTY_SERVICE:
         case Car.SENSOR_SERVICE:
         case Car.VENDOR_EXTENSION_SERVICE:
            return mCarPropertyService;

         ...
      }
   }
```

可以看到 CarPropertyService 同时对应着 CarCabinManager、CarHvacManager、CarInfoManager、CarPropertyManager、CarSensorManager 和 CarVendorExtensionManager 这六个对象,可以说是一个服务分饰多角了。所以在 Android 10 中,谷歌官方直接推荐使用 CarPropertyManager。

5.1 CarInfoManager——车辆信息服务

虽然 CarPropertyService 分饰着多个角色,但是在 Android 官方的 API 参考手册[①]中,只提供了 CarInfoManager 和 CarSensorManager 相关的 API 说明。这是因为其他的 Manager 都是标明为 hide(隐藏)的 API,其中 CarCabinManager、CarHvacManager、CarVendorExtensionManager 同时被标注为 SystemApi,在 Android 中被标注为 hide 和 SystemApi 的类是不会出现在公开的 SDK 中的。

而由于在 Android 9 的时候,谷歌并未正式将 Car API 加入官方 SDK 中,而到了 Android 10,又不再推荐使用 CarSensorManager 等对象,并公开了 CarPropertyManager。因此就造成了有趣的一幕,在官方的 API 文档上,CarSensorManager 在 API Level 29 刚被加入的同时,也被标记为弃用的,出现了“还未被使用就遭弃用”的情况。其实,是因为 Android 9 中以上 API 就已存在了,并且是 Android 9 平台的标准方法,而到了 Android 10 为了更方便地拓展属性(随着越来越多的功能加入,将其分类就更困难了,而如果创建太多的 Manager 对象,维护和使用成本就将变高),才将 CarSensorManager、CarCabinManager、CarHvacManager、CarVendorExtensionManager 弃用的。

但另一方面,Android 9 将不同属性按功能分类的设计,笔者认为也有其道理,一方面对于上层应用来说,使用 API 的时候更清晰;另一方面,更容易管理应用访问 API 的范围。

本章,会将与 CarPropertyService 相关的 API 都进行介绍。一方面,便于在不同平台上

① API 参考手册:https://developer.android.com/reference/android/car/Car。

开发的读者了解其用法；另一方面，也可以从中了解到 Android Automotive OS 的发展和变化过程。

CarInfoManager 的用法

关于如何获取 CarInfoManager 对象的实例，在前面介绍 Car API 的用法的时候已经介绍过了，这里就不再赘述了。

CarInfoManager 主要包含以下 API：

```
float    getEvBatteryCapacity()    // 电池容量
int[]    getEvConnectorTypes()     // 充电连接器类型
float    getFuelCapacity()         // 燃油容量
int[]    getFuelTypes()            // 燃料类型
String   getManufacturer()         // 制造商
String   getModel()                // 车型
String   getModelYear()            // 车型年份
String   getVehicleId()            // 车辆 Id
```

CarInfoManager 中的 API 不多，而且从方法的名称也很容易了解其具体的功能[①]。需要注意的是相关权限的申请，虽然 CarInfoManager 中的接口都是公开的，但依然需要申请相关的权限才可以访问，否则运行的时候就会抛出异常，因此需要在 AndroidManifest.xml 中增加对应权限：

```
<uses-permission android:name="android.car.permission.CAR_INFO"/>
```

声明了相应的权限，并且获取到了 CarInfoManager 的实例后就能调用相关的接口了，是不是很简单？如果想获得车辆的型号、能源类型、电池容量等信息，那就使用 CarInfoManager 吧。

5.2 CarSensorManager ——车辆传感服务

接着来看另一个公开的类 CarSensorManager 的功能及其用法。顾名思义，CarSensorManager 主要对应的就是和车辆传感器相关的一些功能了。相比手机，车辆相关的传感器种类要丰富得多，特别是随着汽车越来越智能化，自动驾驶等功能的加入，传感器的数量还会不断增加。很多车辆相关的智能控制功能都和传感器的数据有着密切的关系，如自动空调、自动控制灯光、驾驶座位自动调整等。

作为一款运行在 Android Automotive OS 上的应用，为了能与车内外场景做更好的融合，获取相关的传感器数据就变得很有必要了。CarSensorManager 作为 Android 提供的标准接口，应用可以通过它来获取车辆传感器数据并监听传感器数据的变化。

值得一提的是，如果之前接触过 Android 传感器相关的开发，应该对 Android 中的 SensorManager 并不陌生，现在又多出了 CarSensorManager，可能会让人觉得有一点疑惑，

① CarInfoManager 中的 getVehicleId 返回的并不是车辆的 VIN(车辆唯一识别码)。且始终返回为空字符串，无实际意义，使用的时候需要注意。

原来的 SensorManager 还有用吗？到底该使用谁来获取传感器的数据呢？其实，在 Android Automotive OS 中，可以将 CarSensorManager 看作是 SensorManager 的一个补充，与它相关联的是和车辆密切相关的传感器数据，如车速、发动机转速、油量、点火状态等。与此同时，开发者依然可以使用 SensorManager 来获取传感器的数据，当想要获取陀螺仪、加速度计、磁力传感器的相关数据时，依然需要使用 SensorManager。可以说，CarSensorManager 和 SensorManager 各司其职，为使用者提供不同种类的传感器数据。

下面介绍 CarSensorManager 的用法。

1. 获取 CarSensorManager 对象

在 Car 连接成功后通过 getCarManager 方法获取 CarSensorManager 的实例，示例源码如下：

```
mSensorManager = (CarSensorManager) mCar.getCarManager(Car.SENSOR_SERVICE);
```

2. CarSensorManager 中的属性

在 CarSensorManager 中包含了汽车特有的传感器类型，如车速、发动机转速等。表 5-1 列出了可供使用的传感器类型。

表 5-1　CarSensorManager 属性列表

属　　性	类　　型	权　　限	是否为系统权限
SENSOR_TYPE_CAR_SPEED	车速	CAR_SPEED	否
SENSOR_TYPE_RPM	发动机转速	CAR_ENGINE_DETAILED	是
SENSOR_TYPE_ODOMETER	里程数	CAR_MILEAGE	是
SENSOR_TYPE_FUEL_LEVEL	油量	CAR_ENERGY	否
SENSOR_TYPE_PARKING_BRAKE	驻车制动	CAR_POWERTRAIN	否
SENSOR_TYPE_GEAR	挡位	CAR_POWERTRAIN	否
SENSOR_TYPE_NIGHT	白天黑夜	CAR_EXTERIOR_ENVIRONMENT	否
SENSOR_TYPE_ENV_OUTSIDE_TEMPERATURE	车外环境温度	CAR_EXTERIOR_ENVIRONMENT	否
SENSOR_TYPE_IGNITION_STATE	点火状态	CAR_POWERTRAIN	否
SENSOR_TYPE_WHEEL_TICK_DISTANCE	轮距	CAR_SPEED	否
SENSOR_TYPE_ABS_ACTIVE	ABS 状态	CAR_DYNAMICS_STATE	是
SENSOR_TYPE_TRACTION_CONTROL_ACTIVE	牵引力控制	CAR_DYNAMICS_STATE	是
SENSOR_TYPE_FUEL_DOOR_OPEN	加油口状态	CAR_ENERGY_PORTS	否
SENSOR_TYPE_EV_BATTERY_LEVEL	电量值	CAR_ENERGY	否
SENSOR_TYPE_EV_CHARGE_PORT_OPEN	充电口状态	CAR_ENERGY_PORTS	否
SENSOR_TYPE_EV_CHARGE_PORT_CONNECTED	充电连接状态	CAR_ENERGY_PORTS	否
SENSOR_TYPE_EV_BATTERY_CHARGE_RATE	充电速率	CAR_ENERGY	否
SENSOR_TYPE_ENGINE_OIL_LEVEL	机油量	CAR_ENGINE_DETAILED	是

表 5-1 列出了当前 CarSensorManager 中支持的传感器数据类型，以及获得相关数据所需要的权限[①]。需要注意的是，部分权限是系统级别[②]的，普通第三方应用无法获取，从这点上可以看出目前 Android Automotive OS 对于车辆数据的开放还是相对比较谨慎的，出于安全性方面的考量，第三方应用能获得的传感器数据比较有限。

另外，从传感器的数量来说，目前来看，同样算不上丰富。实际上，车辆可以公开的传感器数据要比列表中的多得多，如油门深度、驾驶模式、胎压等，目前都不在列表中。可以预想到随着车辆相关功能进一步的发展以及 Android Automotive OS 系统方面的逐步完善和迭代，未来会支持更多的传感器类型。

3. 获取传感器数据

通过 CarSensorManager 中定义的不同的属性，可以获取相应的传感器数据。这里以车速为例，说明如何获取传感器数据，对于其他的属性，获取的方式也是一样的。通过 CarSensorManager 获取车速信息的源码如下：

```
if (mCarSensorManager
    .isSensorSupported(CarSensorManager.SENSOR_TYPE_CAR_SPEED)) {
  CarSensorEvent event = mCarSensorManager
    .getLatestSensorEvent(CarSensorManager.SENSOR_TYPE_CAR_SPEED);
  if (event != null) {
    CarSensorEvent.CarSpeedData data = null;
    data = event.getCarSpeedData(data);
    Log.d(TAG, "Speed = " + data.carSpeed);
  }
}
```

首先，通过调用 isSensorSupported 方法判断当前是否支持该传感器类型，如果返回为 false 说明当前车辆上是不提供该传感器的数据的。

如果该传感器是支持的，就可以调用 getLatestSensorEvent 方法来获取最近一次的传感器数据了。CarSensorEvent 中包含很多的内部类，对应不同的传感器类型，通过对应的方法就可以获取到数据了。在调用 getCarSpeedData 的时候只需传入一个空对象即可，在方法中会进行判断，如果为空则创建新的数据类。

4. 监听传感器数据变化

除了主动获取传感器的数据外，还可以注册监听器接收传感器数据变化的通知。方法也很简单，大致如下：

```
CarSensorManager.OnSensorChangedListener listener =
  new CarSpeedSensorListener();
mCarSensorManager.registerListener(listener,
  CarSensorManager.SENSOR_TYPE_CAR_SPEED,
  CarSensorManager.SENSOR_RATE_NORMAL);
```

① 列表中的权限省略了包名部分：android.car.permission。在 AndroidManifest.xml 使用权限时应使用完整的声明，如“android.car.permission.CAR_ENERGY”。

② 这里所说的系统级别权限指代的是 Android 中保护级别为“signature|privileged”及以上的权限。关于保护级别的具体分类及定义请参考：https://developer.android.com/reference/android/R.attr#protectionLevel。

调用 CarSensorManager 中的 registerListener 方法以注册监听器,参数中传入需要监听的传感器类型和接收频率。

关于接收频率,根据传感器的功能和类型不同,事件上报的特点也不尽相同,大致可分为以下三类。

(1) 持续上报的传感器事件(如当前车速);

(2) 变化时上报的传感器事件(如油箱盖打开);

(3) 一次性上报的传感器事件(如低电量)。

在注册监听器时传入的频率参数只有当传感器上报类型为持续上报的时候才有意义。同时,数据的实际上报频率未必和传入的参数相一致,一般来说,一个传感器只能以一种频率上报数据,也就是说,如果多个监听器以不同的 rate 值监听同一个传感器数据,那么系统可能只会按照其中一种频率上报数据,而其他的会被忽略。

监听器的实现非常简单,在监听器的回调方法中接收数据变化的通知,并获取最新的数值。以下源码是一个简单的监听器实现:

```
class CarSpeedSensorListener
      implements CarSensorManager.OnSensorChangedListener {
        @Override
        public void onSensorChanged(CarSensorEvent event) {
          Log.d(TAG, "onSensorChanged, event: " + event);
          if (event.sensorType
                == CarSensorManager.SENSOR_TYPE_CAR_SPEED) {
              CarSensorEvent.CarSpeedData data = null;
              data = event.getCarSpeedData(data);
              Log.d(TAG, "Speed = " + data.carSpeed);
          }
        }
      }
```

CarSensorManager 很好地支持了主动获取和订阅这两种获取传感器数据的方式,同时支持设置传感器事件上报的频率。在大多数情况下,相信可以满足客户端的使用场景了。除了 CarSensorManager,类似的使用方式也会出现在其他服务中。

5.3 CarHvacManager——车内空调系统服务

本节介绍另一个很有用的对象 CarHvacManager——管理空调系统相关功能的服务。近几年,随着语音交互在车内的快速普及,越来越多的车主已经习惯了用语音来控制车内的一些功能,其中通过语音控制空调相关功能是最常见的应用场景之一。例如,“升高温度”“我很热”,车内的语音助手就能理解用户所说的,并完成空调的开关和对温度的调节,既方便又安全,无须驾驶人员动手操作。要实现类似的功能,就意味着软件需要提供对应的 API,Android Automotive OS 定义了标准的 CarHvacManager API 来提供相关的功能,通过该服务可以实现对空调系统的监听和控制。比较可惜的是,CarHvacManager 所涉及的属性都需要系统级别的权限,所以第三方应用目前是无法直接使用 CarHvacManager 的。

1. CarHvacManager 的用法

通过以下方式获取 CarHvacManager 对象实例：

```
mHvacManager = (CarHvacManager) mCarApi.getCarManager(Car.HVAC_SERVICE);
```

2. CarHvacManager 中的属性

与 CarSensorManager 一样，CarHvacManager 中同样也定义了许多属性。不同的是，CarHvacManager 中的属性不仅是“只读”的，不少属性同时还是“可写”的，也就是说，通过设置特定值可以对相关功能进行控制。

表 5-2 列出了 CarHvacManager 中包含的属性。

表 5-2 CarHvacManager 属性列表

属　　性	类　　型	功　　能
ID_MIRROR_DEFROSTER_ON	bool	后视镜除霜
ID_STEERING_WHEEL_HEAT	int	方向盘加热
ID_OUTSIDE_AIR_TEMP	float	车外温度
ID_TEMPERATURE_DISPLAY_UNITS	int	温标(华氏度或摄氏度)
ID_ZONED_TEMP_SETPOINT	float	温度
ID_ZONED_TEMP_ACTUAL	float	实际温度
ID_ZONED_FAN_SPEED_SETPOINT	int	风速
ID_ZONED_FAN_SPEED_RPM	int	风扇转速
ID_ZONED_FAN_DIRECTION_AVAILABLE	vector	可用风向
ID_ZONED_FAN_DIRECTION	int	风向
ID_ZONED_SEAT_TEMP	int	座椅温度
ID_ZONED_AC_ON	bool	AC 开关
ID_ZONED_AUTOMATIC_MODE_ON	bool	自动空调
ID_ZONED_AIR_RECIRCULATION_ON	bool	空调循环
ID_ZONED_MAX_AC_ON	bool	强力空调
ID_ZONED_DUAL_ZONE_ON	bool	多区域空调
ID_ZONED_MAX_DEFROST_ON	bool	强力除霜
ID_ZONED_HVAC_POWER_ON	bool	空调系统开关
ID_ZONED_HVAC_AUTO_RECIRC_ON	bool	自动空气循环
ID_WINDOW_DEFROSTER_ON	bool	车窗除霜

从使用上来说，CarHvacManager 要相对复杂一些，因为不同车型的配置会有比较大的差异，可用的属性会不同，即使是相同的属性，有些功能是区分区域的(如前后排空调)，不同区域的配置也可能会不一样，因此在实现相关源码的时候会涉及比较多的逻辑判断，这增加了源码上的复杂性。下面以设置空调温度作为例子说明如何使用 CarHvacManager 提供的 API。

示例：设置空调温度

在权限上，CarHvacManager 中大部分属性[①]所关联的权限都是 CONTROL_CAR_

① ID_OUTSIDE_AIR_TEMP 属性，所要求的权限为 CAR_EXTERIOR_ENVIRONMENT。

CLIMATE,因此为了成功调节空调温度,首先需要在 AndroidManifest.xml 中声明相关权限:

```
<uses-permission android:name="android.car.permission.CONTROL_CAR_CLIMATE" />
```

需要说明的是,该权限的保护级别是“signature|privileged”。因此,想要获得此权限,需要以系统应用[①]的身份运行。

通过下面的源码来看如何获取和设置空调温度。在这段源码中,除了设置温度以外,还包含了如何判断特定属性是否支持,以及如何获取当前状态的逻辑。

示例源码的功能是获取前排主驾(按中国的驾驶习惯为左侧)的温度后将其升高一度。很简单,但却涉及很多的逻辑判断。首先,要判断车辆是否支持该属性。然后,获取支持的区域和变化范围。在设置该属性前,还需要判断当前属性是否可用,设置的区域是否支持。

注意,当前属性是否可用与是否支持该属性是两个不同的概念。例如,虽然当前车辆支持空调功能,但在车辆熄火的状态下,空调设置可能暂时不可用。

```
// 获取当前前排左侧温度,并将温度升高一度
void incTemperature() {
    // 获取支持的属性列表
    List<CarPropertyConfig> carPropertyConfigs =
            mHvacManager.getPropertyList();
    int zone = VehicleAreaSeat.SEAT_ROW_1_LEFT;
    boolean supported = false;
    float max = 0f; // 最大值
    float min = 0f; // 最小值
    for (CarPropertyConfig prop : carPropertyConfigs) {
        if (prop.getPropertyId() == ID_ZONED_TEMP_SETPOINT) {
            for (int areaId : propConfig.getAreaIds()) {
                if ((zone & areaId) == zone) {
                    // 获取正确的位置值
                    zone = areaId;
                    supported = true;
                }
            }
            max = (Float) prop.getMaxValue();
            min = (Float) prop.getMinValue();
            break;
        }
    }
    if (supported
            && mHvacManager.isPropertyAvailable(
                ID_ZONED_TEMP_SETPOINT, zone)) {
        // 获取当前温度
        float current =
                mHvacManager.getFloatProperty(ID_ZONED_TEMP_SETPOINT, zone);
```

① 这里的系统应用是指拥有系统签名或者安装在 priv-app 目录下的应用。

```
            ++current;
            if (max > current && min < current) {
                // 设置新的温度值
                mHvacManager.setFloatProperty(ID_ZONED_TEMP_SETPOINT, zone,
                                    current);
            }
        }
    }
```

需要格外注意的是，在示例源码中以 VehicleAreaSeat.SEAT_ROW_1_LEFT 作为区域参数来设置前排左侧的空调温度。但是由于车型配置的原因及硬件抽象层具体实现的不同，实际支持的区域值可能是复合的。例如，可用区域中包含值 VehicleAreaSeat.SEAT_ROW_1_LEFT | VehicleAreaSeat.SEAT_ROW_2_LEFT | VehicleAreaSeat.SEAT_ROW_2_CENTER。这种情况下，虽然不支持单一调整 VehicleAreaSeat.SEAT_ROW_1_LEFT 区域的温度，但是在实现的时候可以考虑使用复合区域，因此，需要通过“位”操作来获取正确的区域 ID。

这也是笔者认为未来 Android Automotive OS 中还需要进一步完善或者说规范化的部分，目前尚未有明确的规范来要求制造商在实现多区域功能时如何定义支持的区域值。当然，空调相关功能仅提供给系统应用使用，各制造商的系统应用可较为方便地根据自身产品的特点做出调整。

可以看到，CarHvacManager 在使用的时候主要是要做好逻辑上的判断，如果疏忽了一些条件就有可能导致调用过程中发生异常，这是使用 Car API 控制分区域属性时候特别要注意的。在示例源码中，为了能更直观地说明使用方式，将相关判断都包含在了一个方法里，在实际开发过程中，读者可以将各个判断封装在单独的方法中进行复用。

CarHvacManager 同样提供了方法用以监听属性的变化。注册监听器的方法如下：

```
mCarHvacManager.registerCallback(new CarHvacEventCallback() {
 @Override
 public void onChangeEvent(CarPropertyValue value) {

 }

 @Override
 public void onErrorEvent(@PropertyId int propertyId, int area) {

 }
});
```

通过 registerCallback 方法可以在属性的值发生变化时获得回调。与 CarSensorManager 中不同的是，CarHvacManager 的监听器并不仅仅监听单个属性的变化，而是同时监听了 CarHvacManager 相关的所有属性的变化情况，如果要知道具体是哪个属性发生了变化就需要通过回调方法中 CarPropertyValue 参数进行进一步的判断。在使用完成后不要忘记移除监听器，以避免发生内存泄漏。

在介绍 CarHvacManager 的过程中,涉及了 CarPropertyValue、CarPropertyConfig 等用来定义管理车辆属性的相关类,在后文介绍 VehicleHAL 相关内容的时候会进一步具体介绍这些类的作用及实现,在此,读者可以先通过源码示例熟悉它们的用法。

5.4 CarCabinManager——座舱服务

CarCabinManager 提供的是座舱内相关功能的 API,包括座椅、安全带、车窗等。它在用法上和 CarHvacManager 类似,同样的 CarCabinManager 也是系统级别的,只有拥有系统权限的应用才可以使用。

CarCabinManager 中的属性

CarCabinManager 中的属性都和座舱内的硬件设备相关,如车门、后视镜、座椅等。与这些设备相关的属性又根据其特点进行了细分,对于可以移动、调节的设备而言,会有不同方向之分。表 5-3 列出了 CarCabinManager 中所包含的属性,以及属性所对应的主要设备和功能。

表 5-3 CarCabinManager 属性列表

属　　性	类　　型	功　　能
ID_DOOR_POS	int	车门
ID_DOOR_MOVE	int	
ID_DOOR_LOCK	bool	
ID_MIRROR_Z_POS	int	后视镜
ID_MIRROR_Z_MOVE	int	
ID_MIRROR_Y_POS	int	
ID_MIRROR_Y_MOVE	int	
ID_MIRROR_LOCK	bool	
ID_MIRROR_FOLD	bool	
ID_SEAT_MEMORY_SELECT	int	座椅记忆
ID_SEAT_MEMORY_SET	int	
ID_SEAT_BELT_BUCKLED	bool	安全带
ID_SEAT_BELT_HEIGHT_POS	int	
ID_SEAT_BELT_HEIGHT_MOVE	int	
ID_SEAT_FORE_AFT_POS	int	座椅前后位置
ID_SEAT_FORE_AFT_MOVE	int	
ID_SEAT_BACKREST_ANGLE_1_POS	int	座椅靠背
ID_SEAT_BACKREST_ANGLE_1_MOVE	int	
ID_SEAT_BACKREST_ANGLE_2_POS	int	
ID_SEAT_BACKREST_ANGLE_2_MOVE	int	
ID_SEAT_HEIGHT_POS	int	座椅高度
ID_SEAT_HEIGHT_MOVE	int	
ID_SEAT_DEPTH_POS	int	座椅深度
ID_SEAT_DEPTH_MOVE	int	

续表

属　　性	类　　型	功　　能
ID_SEAT_TILT_POS	int	座椅倾角
ID_SEAT_TILT_MOVE	int	
ID_SEAT_LUMBAR_FORE_AFT_POS	int	腰托
ID_SEAT_LUMBAR_FORE_AFT_MOVE	int	
ID_SEAT_LUMBAR_SIDE_SUPPORT_POS	int	
ID_SEAT_LUMBAR_SIDE_SUPPORT_MOVE	int	
ID_SEAT_HEADREST_HEIGHT_POS	int	头枕
ID_SEAT_HEADREST_HEIGHT_MOVE	int	
ID_SEAT_HEADREST_ANGLE_POS	int	
ID_SEAT_HEADREST_ANGLE_MOVE	int	
ID_SEAT_HEADREST_FORE_AFT_POS	int	
ID_SEAT_HEADREST_FORE_AFT_MOVE	int	
ID_WINDOW_POS	int	车窗
ID_WINDOW_MOVE	int	
ID_WINDOW_LOCK	bool	

可以看到 CarCabinManager 中最主要的是和座椅相关的属性，同时还有车窗、后视镜相关的功能。大部分功能同时会有位置(Position)和移动(Move)两个属性，其中位置属性主要是设置具体的位置值，而移动则是该设备的移动方向。

CarCabinManager 丰富了车内设施的控制功能，通过它提供的 API，开发者可以为驾驶者提供很多个性化的功能，如座椅调节、车窗调节。如果说通过 CarHvacManager 控制空调，能让驾驶者感受到车内的环境的变化，那么通过 CarCabinManager 控制车内设施，就能让驾驶者实实在在看到车内的智能化。

在用法上，CarCabinManager 和 CarHvacManager 非常相似，同样可以获取或设置属性的值，并对属性变化进行监听。API 也没什么两样。所以本节就不详细介绍 CarCabinManager 是如何使用的了。相关的权限主要有以下几个：

- android. car. permission. CONTROL_CAR_WINDOWS。
- android. car. permission. CONTROL_CAR_SEATS。
- android. car. permission. CONTROL_CAR_MIRRORS。

5.5 CarVendorExtensionManager——制造商拓展服务

前面在介绍 CarSensorManager 的时候曾提到过，CarSensorManager 中的传感器类型还比较有限，车辆上还有很多的传感器并没有出现在 CarSensorManager 中。同样的，CarHvacManager 或者 CarCabinManager 是否就将车上相关的功能都覆盖全面了呢？市场上的汽车品牌种类繁多，每款车型的功能又不相同。汽车制造商们也在不断推陈出新，推出一些属于品牌特有的功能来吸引消费者的目光。要想将所有的功能都定义为标准的属性是非常困难的。对此，Android Automotive OS 的做法是，除了定义了目前市场上绝大多数车型都适用的属性外，同样允许制造商根据自己所拥有的其他功能进行拓展。这就是本节介

绍的 CarVendorExtensionManager 主要作用,它让制造商可以扩展 VehicleHAL 中已经定义的属性,加入额外的功能。

1. CarVendorExtensionManager 的用法

通过以下方式获取 CarVendorExtensionManager 对象的实例:

```
mVendorManager = (CarVendorExtensionManager)
            mCarApi.getCarManager(Car.VENDOR_EXTENSION_SERVICE);
```

要使用 CarVendorExtensionManager 需要申请以下权限:

```
<uses-permission android:name="android.car.permission.CAR_VENDOR_EXTENSION" />
```

该权限同样是系统级别的,普通的第三方应用无法使用。

对于制造商自定义的属性,目前,Android Automotive OS 中统一使用"android.car.permission.CAR_VENDOR_EXTENSION"进行权限的控制。尚不支持制造商自定义不同的权限进行管理,在这点上似乎欠缺了一些灵活性。希望谷歌在未来继续完善,在自定义属性的权限上能有更细颗粒度的权限管理方式,根据其功能的特点、类别,定义不同的权限。

2. 获取/设置属性

在属性的设置和获取上,CarVendorExtensionManager 与 CarHvacManager、CarCabinManager 的使用方法区别并不大。但由于是拓展的属性,属性的类型是不确定的,所以在调用 setProperty 和 getProperty 时需要传入属性的类型。假设自定义了 CUSTOM_ZONED_FLOAT_PROP_ID 这一属性,且值为浮点类型,如果该属性是多区域的,还需要传入区域值,获取和设置该属性的方法如下:

```
mVendorManager.setProperty(
      Float.class,
      CUSTOM_ZONED_FLOAT_PROP_ID,
      VehicleAreaSeat.ROW_1_RIGHT,
      value);

float actualValue = mVendorManager.getProperty(
      Float.class, CUSTOM_ZONED_FLOAT_PROP_ID,
      VehicleAreaSeat.ROW_1_RIGHT);
```

像 CarVendorExtensionManager 这样设定和获取属性的方式,在后面介绍 CarPropertyManager 的时候还会遇到。实际上,控制车辆硬件相关功能都离不开 setProperty 和 getProperty 这两个方法。

3. 监听属性变化

CarVendorExtensionManager 同样可以通过注册回调接口的方式来监听属性值发生的变化。方式十分简单,实现 CarVendorExtensionCallback 就可以了,方式如下:

```
mVendorManager.registerCallback(new CarVendorExtensionCallback() {
 @Override
```

```
public void onChangeEvent(CarPropertyValue carPropertyValue) {
  if (carPropertyValue != null
    && carPropertyValue.getPropertyId() == CUSTOM_ZONED_FLOAT_PROP_ID) {
    float newValue = (Float) carPropertyValue.getValue();
   }
 }

 @Override
 public void onErrorEvent(int propertyId, int zone) {

 }
});
```

需要注意的是，尽管注册的是 CarVendorExtensionCallback，但是该回调方法不仅会收到拓展属性相关的变化事件，对于其他属性的变化事件（如空调、传感器）也有可能被传递过来。因此，在 onChangeEvent 方法中需要做好相关的判断，确保该次改变事件是所需要的。

有了 CarVendorExtensionManager，CarService 一下子拥有了拓展属性的能力。让原来看上去有限的功能，变得可以无比丰富，至少理论上是这样，当然实际上还依赖于制造商的实现。但无论如何这大大增加了可拓展性，制造商们不用为了一个新功能苦苦等待谷歌的更新，或是自己重新设计一套方案，而是只要定义增加新的属性就可以了。对于需要使用自定义属性的应用，只需要知道确切的 Id 和类型。

关于如何增加自定义属性，以及属性的定义规则会在后文介绍 CarPropertyManager 及 VehicleHAL 时再详细展开。

通过前文的介绍，读者已经知道可以使用 CarInfoManager 获取车辆相关的信息；CarSensorManager 来获取车辆传感器数据；CarHvacManager 控制空调相关的功能；CarCabinManager 控制车舱内设施；以及 CarVendorExtensionManager 提供属性拓展的能力。不同的属性分散在这些 Manager 中，让开发者可以更加清晰地找到对应的 API，并且对权限进行更加细致的划分。那么有没有可能对所有属性进行统一的管理呢？在介绍 CarPropertyService 开始的时候提到过，以上这些不同的 Manager 的背后都有 CarPropertyManager 的身影，它管理着所有的属性，下面就来看看 CarPropertyManager 到底做了什么，以及是如何实现的。

5.6 CarPropertyManager——车辆属性 API

本节介绍汽车属性中最主要的一个 Manager-CarPropertyManager。在 Android 9 中 CarPropertyManager 还是隐藏（hide）接口，所以不会在公开的 SDK 中出现，但是它十分重要。而在 Android 10 中，CarPropertyManager 变成了车辆属性的主要 API，并允许任何运行在 Android Automotive OS 上的应用进行调用。初看 CarPropertyManager 会觉得很熟悉，它的方法包括（但不限于）以下这些：

```
boolean registerListener(CarPropertyEventCallback callback, int prop, float rate)
boolean isPropertyAvailable(int propId, int area)
```

```
boolean getBooleanProperty(int prop, int area)
float getFloatProperty(int prop, int area)
int getIntProperty(int prop, int area)
int[] getIntArrayProperty(int prop, int area)
<E> CarPropertyValue<E> getProperty(Class<E> clazz, int propId, int area)
<E> CarPropertyValue<E> getProperty(int propId, int area)
<E> void setProperty(Class<E> clazz, int propId, int area, E val)
void setBooleanProperty(int prop, int area, boolean val)
void setFloatProperty(int prop, int area, float val)
void setIntProperty(int prop, int area, int val)
```

看到这些方法,就会发现和 CarHvacManager、CarVendorExtensionManager 等服务中的方法定义很类似。在使用方法上和之前提到的几个服务也是一样的。其实,无论是 CarInfoManager,还是 CarSensorManager 或 CarHvacManager,它们的功能都可以直接通过 CarPropertyManager 来完成。

1. CarPropertyManager 的用法

熟悉了 CarHvacManager、CarVendorExtensionManager 等几个相关服务的用法之后,在 CarPropertyManager 的使用上,相信读者对相关方法已经很了解了。这里再做一些简单的补充。

关于属性的获取,在 CarPropertyManager 中除了 getProperty 方法之外,还有像 getBooleanProperty、getIntProperty 这样明确属性类型的获取方法。其实这些方法只是对于 getProperty 方法的封装,以 getIntProperty 为例,它的实现是这样的:

```
[CarPropertyManager.java]

 public int getIntProperty(int prop, int area) {
    CarPropertyValue<Integer> carProp = getProperty(Integer.class, prop, area);
    return carProp != null ? carProp.getValue() : 0;
 }
```

看上去在明确知道属性类型的情况下,getBooleanProperty、getIntProperty 等方法在使用上更加简洁。但是在这里,依然推荐开发者们使用 getProperty 来获取相应的属性值,因为 getProperty 方法返回的是 CarPropertyValue 对象,其不仅包含属性值,还包含属性的状态,而 getIntProperty 等方法在属性不可用的情况下,返回的是默认值,这在有的时候会导致读取的数据不正确。

下面以 NIGHT_MODE(昼夜模式)属性为例,说明使用 getProperty 方法的好处。

```
CarPopertyValue<Boolean> val = mCarPropertyManager.
                    getProperty(Boolean.class, NIGHT_MODE, 0);
if (val == null || val.getStatus() != CarPropertyValue.STATUS_AVAILABLE) {
  // 该属性不支持或不可用,使用当前时间判断昼夜情况
} else if (val.getValue()) {
  // 黑夜模式
} else {
  // 白昼模式
}
```

从这段源码中，可以很直观地看到使用 getProperty 方法，与 getBooleanProperty 方法获取昼夜状态相比，可以更准确地判断当前属性的状态，并在属性不支持或不可用时，使用更合理的处理逻辑。因此，虽然 getProperty 方法会增加源码的数量，但在大部分情况下，依然推荐大家使用该方式获取属性。

在设置属性值方面，CarPropertyManager 同样提供了 setProperty 及明确类型的 setIntProperty、setBooleanProperty 等方法。同样的，明确类型的 setIntProperty、setBooleanProperty、setFloatProperty 方法是 setProperty 方法的简单包装。

```
[CarPropertyManager.java]

  public void setBooleanProperty(int prop, int area, boolean val) {
    setProperty(Boolean.class, prop, area, val);
  }
```

开发者根据需要调用相关 set 方法即可，用法十分简单。

在注册监听属性变化方面，CarPropertyManager 提供更细颗粒度的监听方法，registerListener[①] 方法可以监听单个属性值的变化，开发者可以通过在注册监听器时传入属性 ID 指定监听器所对应的属性。同时，可以指定数据上报的频率，与 5.2 节介绍的一样，该频率与属性类型和其他监听器的频率有关，并不能保证数据会以传入的期望频率进行上报。监听属性的方式，可参考以下源码：

```
CarPropertyManager.CarPropertyEventListener mCarPropertyEventListener =
new CarPropertyManager.CarPropertyEventListener() {

  @Override
  public void onChangeEvent(CarPropertyValue value) {

  }

  @Override
  public void onErrorEvent(int propId, int zone) {

  }
};

mCarPropertyManager.registerListener(mCarPropertyEventListener,
VehiclePropertyIds.PERF_VEHICLE_SPEED, /* rate = */ 5);
```

2. CarPropertyManager 的相关类

本节介绍与 CarPropertyManager 紧密相关的一些类。

前文已经提到过像 CarInfoManager、CarHvacManager、CarSensorManager 都是通过 CarPropertyManager 实现其功能的。在其他几个 Manager 初始化的时候，都会创建属于自己的 CarProperyManager 对象。这几个 Manager 拥有了 CarPropertyManager 以后用来做什么

① 从 Android 10 开始，为了与其他 Manager 的监听方法命名保持一致，CarPropertyManager 的监听方法重命名为 registerCallback，但在用法上没有变化。

了呢？通过 CarInfoManager 中的 getFuelCapacity 方法为例就能看出一些端倪。源码如下：

```
[CarInfoManager.java]

  public float getFuelCapacity() {
    CarPropertyValue<Float> carProp = mCarPropertyMgr.getProperty(Float.class,
        BASIC_INFO_FUEL_CAPACITY, 0);
    return carProp != null ? carProp.getValue() : 0f;
  }
```

再来看看 CarHvacManager 中的 isPropertyAvailable 方法：

```
[CarHvacManager.java]

  public boolean isPropertyAvailable(@PropertyId int propertyId, int area) {
    return mCarPropertyMgr.isPropertyAvailable(propertyId, area);
  }
```

原来这背后其实就是对 CarPropertyManager 的调用。其他几个 Manager 并没有做什么具体的事情，只是把任务交给了 CarPropertyManager 再去执行。无论是 CarHvacManager 还是 CarInfoManager 只是 CarPropertyManager 的代理或者说是适配器。

那 Android Automotive OS 为什么要使用这样的设计？笔者认为，主要原因还是因为车辆属性繁多，对功能进行分类能让开发者使用的时候更加清晰，更容易进行功能的区分。当然，各个不同的 Manager 也会让源码变得更加复杂，增加维护的难度，特别是随着属性的增加，要将属性进行归类就会更加麻烦。所以如果是作为系统应用的开发者，对各个属性有了比较全面的了解后，完全可以通过 CarPropertyManager 来实现对所有属性的控制，这样反而能让源码显得更简洁。

除了和 CarPropertyManager 相关的这几个 Manager 之外，在之前的例子中，还出现了如 VehiclePropertyIds、CarPropertyValue、CarPropertyConfig 等相关的辅助类，由于种类繁多，有必要在这里梳理一下各个辅助类的作用。

(1) VehiclePropertyIds，CarPropertyManager 都是通过属性 ID 来对应具体的功能的，不同功能对应不同的 ID，VehiclePropertyIds 中列出了所有在 VehicleHAL 中定义的功能属性，是 Android Automotive OS 官方定义的属性集合。

(2) VehicleAreaDoor，许多功能点都分多个区域，在设置、获取相应属性时，需要传入区域参数，VehicleAreaDoor 定义了与车门相关的区域值，在使用与车门相关的属性时配套使用。

(3) VehicleAreaMirror，与 VehicleAreaDoor 类似，多区域定义，后视镜区域值。

(4) VehicleAreaSeat，多区域定义，座位区域值。

(5) VehicleAreaWheel，多区域定义，车胎区域值。

(6) VehicleAreaWindow，多区域定义，车窗区域值。

(7) VehicleAreaType，区域类型是用以区分一个属性所对应的位置的。对于非多区域属性，往往使用 VEHICLE_AREA_TYPE_GLOBAL 作为其区域 ID。每个区域属性都必须使用预定义的区域类型，即车门、车窗、座椅、轮胎、后视镜中的一个。每种区域类型都有一组在区域类型的枚举中定义的位标记，也就是前文中使用的像 SEAT_ROW_1_LEFT 这

样具体的区域值。

(8) VehicleLightState,灯光状态,开、关、日间。

(9) VehicleLightSwitch,灯光切换,开、关、日间、自动。

(10) VehicleOilLevel,油量状态。

以上这些辅助类中,都定义了相关的静态变量,同时,这些值都与 VehicleHAL 的相关定义一一对应,在 Car API 中将其再次定义是为了方便上层应用使用。

除了以上几个定义静态变量的辅助类以外,还会经常用到 CarPropertyConfig 和 CarPropertyValue 这两个模板类。CarPropertyConfig 和 CarPropertyValue 非常有用,通过前者能获取到一个属性的静态参数,如取值范围、类型、支持的区域等;通过后者能获取一个属性的值和状态。

这里举两个简单的例子。

(1) 通过 CarPropertyManager 获取当前车辆支持的属性(注意,需要拥有对应属性的权限才能获取)。

```
List<CarPropertyConfig> properties = mCarPropertyManager.getPropertyList();
```

CarPropertyConfig 对象的成员变量如表 5-4 所示。

表 5-4 CarPropertyConfig 成员变量列表

类 型	变 量 名	说 明
int	mAccess	该属性是否可读可写(0 不可读不可写; 1 可读; 2 可写; 3 可读写)
int	mAreaType	区域类型,与 VehicleAreaType 对应
int	mChangeMode	变化类型(0 该属性值始终不变; 1 发生变化时通知; 2 以一定频率持续通知当前值)
ArrayList<Integer>	mConfigArray	额外的配置属性
String	mConfigString	额外的配置信息
float	mMaxSampleRate	最大频率(仅对持续上报属性有效)
float	mMinSampleRate	最小频率(仅对持续上报属性有效)
int	mPropertyId	属性 ID
SparseArray<AreaConfig<T>>	mSupportedAreas	区域属性,包含该区域的取值范围
Class<T>	mType	属性的类型

开发者可以通过以上成员变量对应的 get 方法获取具体的值。对于区域属性来说,CarPropertyConfig 中还封装了额外的方法方便开发者获取特定区域的取值,AreaConfig 类的 getMinValue、getMaxValue 方法可以返回某一属性特定区域的取值范围。

(2) 通过 CarPropertyManager 获取当前的车速。

```
CarPropertyValue<Float> value = mCarPropertyManager.getProperty(Float.class,
    VehiclePropertyIds.PERF_VEHICLE_SPEED,
    VehicleAreaType.VEHICLE_AREA_TYPE_GLOBAL);
```

CarPropertyValue 对象的成员变量如表 5-5 所示。

表 5-5 CarPropertyValue 成员变量列表

类 型	变 量 名	说 明
int	mPropertyId	属性 ID
int	mAreaId	区域 ID
int	mStatus	状态(0 可用；1 不可用；2 错误)
long	mTimestamp	时间戳(单位：纳秒)
T	mValue	当前值

虽然看上去很简单,但实际使用过程中会涉及较多的判断,开发者可以进一步对属性进行封装管理,并总结一些有用的实践。CarPropertyConfig 和 CarPropertyValue 这两个类同样和 VehicleHAL 中的定义的结构体相关联,CarService 会将从 HAL 层获取的 VehiclePropConfig 和 VehiclePropValue[①] 分别转换为 CarPropertyConfig 和 CarPropertyValue。

3. 进一步了解 CarPropertyManager

通过前文的介绍,读者应该已经了解了 CarPropertyManager 的重要性。因此有必要进一步了解 CarPropertyManager 背后的原理,更全面地掌握它。

熟悉 Android 的读者应该知道,在 Android 中往往一个 Manager 会对应一个 Service,例如 ActivityManager 对应着 ActivityManagerService；PackageManager 对应着 PackageManagerService。它们运行在不同的进程中,通过 Binder 这一 IPC 机制进行通信。同样地,与 CarPropertyManager 相对应的是 CarPropertyService 这一服务。

同时,读者如果对车辆电子电器的架构有一定了解,那么应该知道,各个具体的功能往往有对应的电子控制单元(Electronic Control Unit,ECU)进行控制。例如,控制座椅位置的命令,最终需要发送到负责控制座椅的 ECU 中才能使座椅移动。

因此,一次控制命令的调用过程大致如图 5-1 所示。

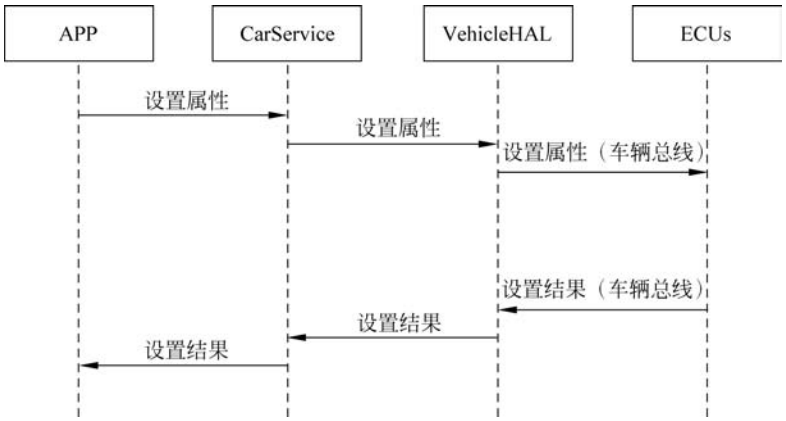

图 5-1 属性设置调用过程

上层的应用通过 Car API 提供的接口进行设置,最终通过车辆总线将命令发送给对应的 ECU,ECU 返回结果给 CarService,并通过回调通知相关应用。这一过程中的

① VehiclePropConfig 和 VehiclePropValue 定义在/hardware/interfaces/automotive/vehicle/2.0/types.hal 文件中。

VehicleHAL 非常重要，它指的是制造商实现的硬件抽象层服务，实现了 Android Automotive OS 定义的相关硬件抽象层接口。由于不同汽车制造商与 ECU 的通信方式、标准、数据格式都不尽相同，所以需要对其进行抽象，统一接口，而具体逻辑则由汽车制造商自己实现。关于 VehicleHAL 的具体内容，会在后面的章节再展开。

下面通过源码，进一步了解 CarPropertyManager 中的具体实现，建议读者在阅读的同时打开源码文件进行查看。

以下是 CarPropertyManager 的构造函数实现：

```
[CarPropertyManager.java]

  public CarPropertyManager(IBinder service, Handler handler, boolean dbg,String tag) {
    ...
    mService = ICarProperty.Stub.asInterface(service);
    ...
  }
```

在 CarPropertyManager 的构造函数中，获得了 ICarProperty 的远程对象，通过该远程对象就可以调用 CarPropertyService。关于 Binder 机制的具体实现及 AIDL 的调用过程，在此就不做展开了[①]。

再来看看 CarPropertyManager 的 setProperty 的调用过程：

```
[CarPropertyManager.java]

  public <E> void setProperty(Class<E> clazz, int propId, int area, E val) {
    ...
    try {
      mService.setProperty(new CarPropertyValue<>(propId, area, val));
    } catch (RemoteException e) {
      throw e.rethrowFromSystemServer();
    }
  }
```

通过 mService 对象，调用 CarPropertyService 中对应的方法：

```
[CarPropertyService.java]

    public void setProperty(CarPropertyValue prop) {
     ...
     ICarImpl.assertPermission(mContext, mHal.getWritePermission(propId));
     mHal.setProperty(prop);
    }
```

这里出现了一个新的对象 mHal，它是 PropertyHalService 对象的实例。

① AIDL 相关内容请参考：https://developer.android.com/guide/components/aidl。

```
[PropertyHalService.java]

    public void setProperty(CarPropertyValue prop) {
      ...
      try {
        mVehicleHal.set(halProp);
      } catch (PropertyTimeoutException e) {
        Log.e(CarLog.TAG_PROPERTY, "set, property not ready 0x"
                                + toHexString(halPropId), e);
        throw new RuntimeException(e);
      }
    }
```

接着调用 VehicleHal 的 set 方法,虽然对象命名叫 VehicleHal,但该 VehicleHal 对象依然是在 CarService 进程中定义并创建的对象。还没有看到对 HAL 层的 Binder 调用。接着往下:

```
[VehicleHal.java]

   protected void set(VehiclePropValue propValue)
                          throws PropertyTimeoutException {
     mHalClient.setValue(propValue);
   }
```

VehicleHal 中的 set 方法只是进一步调用了 HalClient 对象的 setValue 方法。

```
[HalClient.java]

    public void setValue(VehiclePropValue propValue)
                           throws PropertyTimeoutException {
     int status = invokeRetriable(() -> {
       try {
         return mVehicle.set(propValue);
       } catch (RemoteException e) {
         Log.e(CarLog.TAG_HAL, "Failed to set value", e);
         return StatusCode.TRY_AGAIN;
       }
     }, WAIT_CAP_FOR_RETRIABLE_RESULT_MS, SLEEP_BETWEEN_RETRIABLE_INVOKES_MS);
     ...
   }
```

在 HalClient 的 setValue 方法中,终于发现这个 mVehicle 对象是一个 HIDL 调用的远程对象,通过它,实际上调用的是抽象层 VehicleHal 的实现。这里涉及了 Android 8.0 引入的 HIDL 机制,不详细展开了[①]。随着 HIDL 机制的引入,VehicleHal 运行在独立的进程中,由设备制造商或汽车制造商进行实现。

① HIDL 相关内容请参考:https://source.android.com/devices/architecture/hidl。

以上就是一次完整的设置属性值的调用过程，可以看到设置命令最终将发送给制造商实现的 VehicleHal 进程，并由 VehicleHal 最终完成该次调用。

再来追踪 VehicleHal 中的事件的传递过程。

通过设置的流程，可以发现发起 HIDL 调用的远程对象是被 HalClient 对象所持有的，与 VehicleHal 的直接交互是在 HalClient 中完成的。因此事件的向上传递也是从 HalClient 开始的。在收到上报的事件之前，上层应用首先要注册相应的监听方法。

当应用调用 CarPropertyManager 的 registerListener 方法时，其会调用 CarPropertyService 的 registerListener 方法。

```
[CarPropertyService.java]

    public void registerListener(int propId, float rate,
                                 ICarPropertyEventListener listener) {
     ...

     IBinder listenerBinder = listener.asBinder();

     synchronized (mLock) {
        Client client = mClientMap.get(listenerBinder);
        if (client == null) {
            client = new Client(listener);
        }
        client.addProperty(propId, rate);
        List<Client> clients = mPropIdClientMap.get(propId);
        if (clients == null) {
            clients = new CopyOnWriteArrayList<Client>();
            mPropIdClientMap.put(propId, clients);
        }
        if (!clients.contains(client)) {
            clients.add(client);
        }
        if (!mListenerIsSet) {
            mHal.setListener(this);
        }
        if (rate > mHal.getSampleRate(propId)) {
            mHal.subscribeProperty(propId, rate);
        }
     }

     ...
    }
```

上述源码进一步调用 PropertyHalService 的 subscribeProperty 方法，中间会再经过 VehicleHal.java 的调用，最后调用 HalClient 中的 subscribe 方法，调用的路径和设置的流程是一样的。在此省略一些中间过程，直接来看 HalClient 的 subscribe 方法的实现：

```
[HalClient.java]

  public void subscribe(SubscribeOptions... options) throws RemoteException {
    mVehicle.subscribe(mInternalCallback,
                new ArrayList<>(Arrays.asList(options)));
  }
```

这里将 mInternalCallback 对象传递给了 Hal 层。mInternalCallback 对象的实例是继承了 IVehicleCallback.Stub 的 HIDL 桩对象,实现如下:

```
[HalClient.java]

  private static class VehicleCallback extends IVehicleCallback.Stub {
    private Handler mHandler;

    VehicleCallback(Handler handler) {
     mHandler = handler;
    }

    @Override
    public void onPropertyEvent(ArrayList<VehiclePropValue> propValues) {
     mHandler
      .sendMessage(Message.obtain(mHandler,
                        CallbackHandler.MSG_ON_PROPERTY_EVENT,
                        propValues));
    }

    @Override
    public void onPropertySet(VehiclePropValue propValue) {
     mHandler
      .sendMessage(Message.obtain(mHandler,
                        CallbackHandler.MSG_ON_PROPERTY_SET,
                        propValue));
    }

    @Override
    public void onPropertySetError(int errorCode, int propId, int areaId) {
     mHandler.sendMessage(Message.obtain(mHandler,
      CallbackHandler.MSG_ON_SET_ERROR,
      new PropertySetError(errorCode, propId, areaId)));
    }
  }
```

通过 IVehicleCallback,制造商实现的 VehicleHal 进程就可以将事件传递给 CarService 进程了。有兴趣的读者可以进一步追踪当 IVehicleCallback 的 onPropertyEvent 方法被调用后,事件又是如何传递给应用注册的监听器的。

上文出现了一些新的对象,为了便于理解,整理上述类之间的关系,见图 5-2。

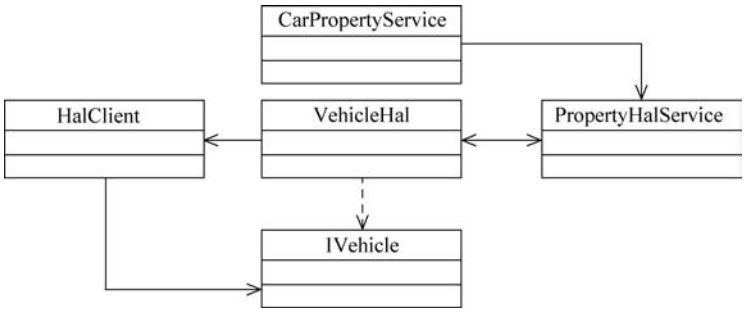

图 5-2　CarPropertyService 关联类图

在 CarService 中虽然有 VehicleHal 类，但该类并不直接调用 HIDL 方法，而是进一步调用 HalClient 封装的方法，HalClient 才是真正与 HAL 层打交道的类。同时，PropertyHalService 处理与 CarPropertyService 相关的业务逻辑，在后面的内容中，还会接触 InputHalService、PowerHalService，VehicleHal 会将不同的事件分发给对应的类进行处理。

总体来说，通过 CarPropertyService 的层层调用，最后通过 IVehicle 与实现了车辆硬件抽象层的进程通信。VehicleHal 进程由各个厂家进行实现，再进一步将消息发送给关联的 ECU，实现控制功能。

4. 车辆属性的权限控制

上文分析了 CarPropertyService 设置属性和监听属性变化的具体实现。同时，介绍了相关的实现类。在上层应用访问属性的时候，需要申请相应的权限。那么这些权限是通过什么方法管理的呢？

下面再介绍 PropertyHalServiceIds 这个类，它作为辅助类，规定了各个属性的访问权限。读者可以先打开手中的源码，找到 PropertyHalServiceIds 源文件。

PropertyHalServiceIds 对象是在 PropertyHalService 的构造函数中创建的。

```
[PropertyHalService.java]

  public PropertyHalService(VehicleHal vehicleHal) {
    mPropIds = new PropertyHalServiceIds();
    …
  }
```

在 PropertyHalServiceIds 的构造函数中将各个权限与属性进行了映射。

```
[PropertyHalServiceIds.java]

 public PropertyHalServiceIds() {
  mProps = new SparseArray<>();

  …
  mProps.put(VehicleProperty.MIRROR_Z_MOVE, new Pair<>(
       Car.PERMISSION_CONTROL_CAR_MIRRORS,
       Car.PERMISSION_CONTROL_CAR_MIRRORS));
```

```
    mProps.put(VehicleProperty.MIRROR_Y_POS, new Pair<>(
            Car.PERMISSION_CONTROL_CAR_MIRRORS,
            Car.PERMISSION_CONTROL_CAR_MIRRORS));

    ...
    mProps.put(VehicleProperty.HVAC_MAX_AC_ON, new Pair<>(
            Car.PERMISSION_CONTROL_CAR_CLIMATE,
            Car.PERMISSION_CONTROL_CAR_CLIMATE));
     mProps.put(VehicleProperty.HVAC_MAX_DEFROST_ON, new Pair<>(
            Car.PERMISSION_CONTROL_CAR_CLIMATE,
            Car.PERMISSION_CONTROL_CAR_CLIMATE));
    ...

    mProps.put(VehicleProperty.PERF_VEHICLE_SPEED, new Pair<>(
            Car.PERMISSION_SPEED,
            Car.PERMISSION_SPEED));
    mProps.put(VehicleProperty.ENGINE_COOLANT_TEMP, new Pair<>(
            Car.PERMISSION_CAR_ENGINE_DETAILED,
            Car.PERMISSION_CAR_ENGINE_DETAILED));

    ...
  }
```

属性对应的权限可以分为读权限和写权限,如果该属性是只读的,那么它的写权限就为空。对于制造商拓展属性,则统一使用“android.car.permission.CAR_VENDOR_EXTENSION”权限进行管理。

有了权限与属性的对应关系,控制权限就变得容易了,在调用相应接口时进行判断就可以了,这里以 CarPropertyService 的 getProperty 方法为例,看一下 CarService 如何判断调用者是否符合权限要求的。

```
[CarPropertyService.java]

  @Override
  public CarPropertyValue getProperty(int prop, int zone) {
     ...
     ICarImpl.assertPermission(mContext, mHal.getReadPermission(prop));
     return mHal.getProperty(prop, zone);
  }
```

ICarImpl 的 assetPermission 方法会判断调用者是否拥有权限或本次调用是否是本应用自身发起的,如果不是,就抛出异常。而 PropertyHalServiceIds 的 getReadPermission 方法会返回对应属性的读权限。具体源码如下:

```
[ICarImpl.java]

  public static void assertPermission(Context context, String permission) {
   if (context.checkCallingOrSelfPermission(permission)
```

```
                        != PackageManager.PERMISSION_GRANTED) {
      throw new SecurityException("requires " + permission);
    }
   }

[PropertyHalServiceIds.java]

   @Nullable
   public String getReadPermission(int propId) {
    Pair<String, String> p = mProps.get(propId);
    if (p != null) {
     // Property ID exists. Return read permission.
     if (p.first == null) {
       Log.e(TAG, "propId is not available for reading : 0x"
                          + toHexString(propId));
     }
     return p.first;
    } else {
     return null;
    }
   }
```

权限的控制对于车辆属性来说很重要，尤其是汽车对于安全有更高的要求。因此目前Car API涉及的权限级别都比较高。

Android中的权限通过AndroidManifest.xml文件进行定义。与其他定义在Android框架中且以“android”为包名（frameworks/base/core/res/AndroidManifest.xml文件中定义的权限）的权限不同，Car API相关的权限单独定义在CarService的AndroidManifest.xml中（packages/services/Car/service/AndroidManifest.xml），并以“android.car”为包名。不过，对于需要申请相关权限的应用而言，处理CarService中定义的权限与处理Android中其他权限的方式并无区别，开发者需要按照Android的权限规范[①]申请相关权限。

5.7 了解 VehicleHAL

本节介绍Android Automotive OS中对于VehicleHAL，即车辆硬件抽象层的定义。

前文中多次提到了VehicleHAL，当提起VehicleHAL的时候，它可能包含以下含义。

（1）由Android Automotive OS定义的硬件抽象层接口，包括车辆属性和方法；

（2）由制造商根据硬件抽象层定义所实现的服务进程；

（3）在CarService中，由Java实现的VehicleHal辅助类。

而在本节中，VehicleHAL主要指代的是第一种含义。而制造商提供的VehicleHal服务进程是业务逻辑主要的实现者，非常重要，但只要满足了硬件抽象层所定义的方法和行为，各制造商的具体实现可能很不一样，没有统一的范式，因此在本书中不过多展开分析。

① Android权限概览：https://developer.android.com/guide/topics/permissions/overview。

VehicleHAL 是连通 CarService 与制造商实现的车辆控制服务进程的桥梁,其中包括种类繁多的车辆属性。那么,各种各样的车辆属性是如何被定义的?有什么规则可循?要找到这些问题的答案,就需要更深入地了解 VehicleHAL。本节通过源码进一步分析 VehicleHAL。

说到 VehicleHAL,就需要先简单解释 HAL 层的概念。HAL 层即硬件抽象层(Hardware Abstraction Layer)的缩写。按照谷歌官方的说法[①],HAL 可定义一个标准接口以供硬件供应商实现,这可让 Android 忽略较低级别的驱动程序实现。

根据上面的定义,就能知道 VehicleHAL 是车辆硬件抽象层。它的作用是定义了标准的接口,让 CarService 可以忽略各个汽车制造商的具体实现。换句话说,CarService 调用 VehicleHAL 定义的接口,而制造商们负责实现这些接口。

下面来看 VehicleHAL 的具体内容。

VehicleHAL 的源码位于 hardware/interfaces/automotive/vehicle/2.0/路径下(截至本书完成时,最新的 VehicleHAL 版本为 2.0)。主要的文件只有三个:

- IVehicle.hal。
- IVehicleCallback.hal。
- types.hal。

这三个文件的定义方式、语法都遵循了 Android 8.0 中提出的 HIDL 的定义规范[②],这里不做展开,有兴趣的读者可以阅读谷歌官网上的相关资料。

其中,IVehicle.hal、IVehicleCallback.hal 的定义都很简单,这两个文件中定义的是具体的类和方法。例如,在 IVehicle 中定义了获取属性值的 get 方法、订阅属性的 subscribe 方法;IVehicleCallback 中定义了属性变化时的回调方法 onPropertyEvent,这些方法在之前介绍 CarPropertyService 的内容时有所提及。方法并不多,是因为 CarService 与 VehicleHAL 的主要实现是基于"属性"的,因此用于定义具体数据结构和属性值的 types.hal 就显得格外重要了。

1. 车辆属性

车辆相关属性值的定义都在 types.hal 文件中,具体来看 types.hal 中对各个属性的定义格式。以 PERF_VEHICLE_SPEED 属性为例,它代表了车速信号,具体定义如下:

```
[hardware/interfaces/automotive/vehicle/2.0/types.hal]

  PERF_VEHICLE_SPEED = (
    0x0207
    | VehiclePropertyGroup:SYSTEM
    | VehiclePropertyType:FLOAT
    | VehicleArea:GLOBAL)
```

其中,VehiclePropertyGroup、VehiclePropertyType、VehicleArea 这几个枚举类型的定义同样在该文件中可以找到。这里的 VehiclePropertyGroup:SYSTEM 等于 0x10000000,

① 具体请参考:https://source.android.com/devices/architecture/hal。

② 具体请参考:https://source.android.com/devices/architecture/hidl/code-style。

VehiclePropertyType：FLOAT 等于 0x00600000，VehicleArea：GLOBAL 等于 0x01000000。因此该属性的值就是 0x11600207。如果查看 CarSensorManager 中车速属性的定义，会发现该值和 CarSensorManager 中的 SENSOR_TYPE_CAR_SPEED 属性的值一模一样，这就是 CarSensorManager 中的车速属性值的原始定义。

VehicleHAL 定义了标准的属性名称和值（为了保持版本的兼容性，这些属性值的定义基本不会发生变化）。制造商在实现汽车服务的时候会通过定义好的属性值区分具体的功能，同时 CarService 中也是通过这些属性来控制具体功能的。

仔细观察属性值的定义可以发现各个属性的定义并不是随意为之的，而是有它的规则。每个属性都是通过不同的掩码组合而来，因此每个属性的不同位有各自的含义。还是以 PERF_VEHICLE_SPEED 属性为例，具体情况如图 5-3 所示。

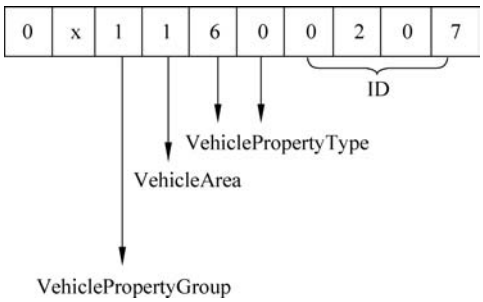

图 5-3　属性各个位的含义

这样通过属性的值，使用者就能知道属性的组别、区域和类型了。其他的属性需要遵循相同的规则。CarPropertyManager 中涉及的属性虽然众多，但都是根据这样的规则来定义的。

下面对不同位的枚举类型做进一步分析。

VehiclePropertyGroup 主要用于区分该属性是 AOSP 定义的还是制造商自定义的，SYSTEM 意味着该值是 Android Automotive OS 的标准定义，任何使用 Android Automotive OS 的制造商都需要遵循一样的定义值；而 VENDOR 意味着是制造商自定义的车辆属性，这些属性对应的功能应该不存在于当前标准的 VehicleHAL 属性列表中，如 VehicleHAL 中已有定义的功能属性，则不应该再重复定义。VehiclePropertyGroup 的具体定义如下：

```
[hardware/interfaces/automotive/vehicle/2.0/types.hal]

enum VehiclePropertyGroup : int32_t {
    SYSTEM      = 0x10000000,
    VENDOR      = 0x20000000,
    MASK        = 0xf0000000,
};
```

而 VehiclePropertyType 则定义了该属性的类型，目前 VehiclePropertyType 主要支持的是一些基本类型，如整型、浮点型、字符串型等，虽然在定义中有 MIXED 这样的复合类型定义，但是在 CarPropertyService 中并不支持复合类型，其主要原因是为了确保数据的通用性，CarPropertyService 本身都是由不同属性、信号所驱动的，单个属性或者说信号的含义是单一的，这样的好处是维持了不同属性间的颗粒度大致相同。当然，有时候就需要制造商在增加定义时做比较细的划分了。除此之外，也可以考虑使用字节数组的方式传递一些复杂类型，并进行序列化和反序列化。VehiclePropertyType 的具体定义如下：

```
[hardware/interfaces/automotive/vehicle/2.0/types.hal]

enum VehiclePropertyType : int32_t {
    STRING        = 0x00100000,
    BOOLEAN       = 0x00200000,
    INT32         = 0x00400000,
    INT32_VEC     = 0x00410000,
    INT64         = 0x00500000,
    INT64_VEC     = 0x00510000,
    FLOAT         = 0x00600000,
    FLOAT_VEC     = 0x00610000,
    BYTES         = 0x00700000,

    MIXED         = 0x00e00000,

    MASK          = 0x00ff0000
};
```

VehicleArea 则定义了属性所对应的区域值,如车窗、座椅等。用以明确该属性在车辆上具体的物理位置,具体定义如下:

```
[hardware/interfaces/automotive/vehicle/2.0/types.hal]

enum VehicleArea : int32_t {
    GLOBAL       = 0x01000000,
    WINDOW       = 0x03000000,
    MIRROR       = 0x04000000,
    SEAT         = 0x05000000,
    DOOR         = 0x06000000,
    WHEEL        = 0x07000000,
    MASK         = 0x0f000000,
};
```

以上就是车辆属性的具体定义方式,Android Automotive OS 对于车辆硬件抽象层的定义还是非常简练的。主要体现在:在接口方法上只定义了几个方法,而不同功能是通过属性 ID 来进行区分的。这样就可以避免重复定义很多作用类似的 set/get 接口。当然,定义虽简单,但是实现起来可就未必了,需要支持如此多的属性。那么就需要汽车制造商在硬件抽象层好好实现相关的功能了。

2. 自定义属性

在之前介绍 CarVendorExtensionManager 时提到制造商可根据需要自定义特有的属性。这部分内容就通过具体的例子说明该如何增加自定义属性。通过前面的介绍,相信读者已经了解了车辆属性定义的具体规则。自定义属性并不复杂,只需按照规则增加新的属性就可以了。

首先,根据 HIDL 定义的规范,在自定义属性前需要创建新的 types.hal 文件。假设现有一款支持“车辆隐身”功能的车型,该功能支持打开和关闭,打开时车就会隐身。这是某品牌特有的功能,在标准的车辆属性中是没有现成的属性可以使用的,但开发者又期望可以通

过 Car API 对该功能进行控制。制造商可以通过自定义属性实现这一需求，该属性可能的定义如下：

```
package vendor.hardware.vehiclevendorextension@1.0;

import android.hardware.automotive.vehicle@2.0;

 enum VehicleProperty: android.hardware.automotive.vehicle@2.0::VehicleProperty {

  /**
   * 车辆隐身
   * @change_mode VehiclePropertyChangeMode:ON_CHANGE
   * @access VehiclePropertyAccess:READ_WRITE
   */
  INVISIBILITY_MODE = (
   0x0001
   | VehiclePropertyGroup:VENDOR
   | VehiclePropertyType:BOOLEAN
   | VehicleArea:GLOBAL),
 }
```

由于是拓展属性，在声明了新定义的属性枚举名称及包名（此处取名为 vendor.hardware.vehiclevendorextension@1.0）的同时，可以在继承原来标准 VehicleProperty 列表的基础上进行，因此加上 android.hardware.automotive.vehicle@2.0::VehicleProperty。这里有几点是需要注意的。

（1）低四位用以区分不同属性，可以从 1 开始，随着属性的新增顺序增加。

（2）由于是制造商自定义的属性，因此类别必须是 VehiclePropertyGroup:VENDOR，用以区别于 AOSP 中的属性（VehiclePropertyGroup:SYSTEM）。

（3）VehiclePropertyType 代表属性值的类型，包括 string、boolean、int32 等，可根据实际属性的类型进行自定义。

（4）VehicleArea 代表区域类型，包含 global（全局）、window（车窗）、mirror（反光镜）、seat（座椅）、door（门）、wheel（车轮）。除了 global，其他区域有更加细分的区域定义，如 VehicleAreaWindow、VehicleAreaMirror 等。但在定义属性时，只需指定至 VehicleArea 就可以了。

3. VehicleHAL 服务进程的实现

到此为止，读者已经了解了 Android Automotive OS 中 VehicleHAL 的主要定义了。尽管还有很多在 types.hal 中定义的数据类型没有被提及，但相信读者已经对最主要的车辆属性及硬件抽象接口是如何定义的有了一个大致的了解。其他在 VehicleHAL 中定义的属性，如输入事件、电源状态等，在后文中还会进一步介绍。

有了车辆硬件抽象层的定义，那么对于制造商而言最重要的工作就是实现一个 VehicleHAL 的服务，为 CarService 提供支持。

在这一部分的实现上，各个制造商的实现各有不同。Android 在 AOSP 的源码中提供了默认的参考实现。位于 hardware/interfaces/automotive/vehicle/2.0/default 路径下。有兴趣的读者参考其实现。尽管其中并不包含与车辆总线交互这样实际的业务逻辑，但是

默认实现中对于 VehicleProperty 队列的管理、消息订阅的管理上都提供了一些值得参考的实践。

制造商该如何具体实现 VehicleHal 服务的具体细节在此就不再展开了,笔者在这里也是抛砖引玉,希望有兴趣或从事相关开发工作的读者可以继续深入学习,结合制造商自身的软件架构特点实现一个高性能且稳定的汽车服务。

5.8 小　　结

本章主要介绍了 CarPropertyManager 的相关功能和实现。并分别介绍了与其相关联的其他几个服务 CarInfoManager、CarHvacManager、CarSensorManager、CarCabinManager 和 CarVendorExtensionManager。

同时介绍了 VehicleHAL,及其定义的规则。

相信通过本章的阅读,读者们能对 Android Automotive OS 中与车辆硬件相关部分的使用和实现有一个比较清晰的了解。需要注意的是,在编写本书时,Android Automotive OS 尚处在起步阶段,车辆属性还有待丰富,谷歌也会不断重构相关的模块。但是基本的架构设计相信是不会变化的,希望未来会有更多的功能加入并开放出来。