5.11 障碍物融合模块传感器数据故障诊断设计

1. 需求及背景介绍

a. 目前实车测试发现，IO负载较高的时候障碍物融合模块的故障诊断存在数据异常的情况，经过分析大概率是融合模块和故障诊断模块使用同一个线程导致资源不足引起数据丢失，导致故障管理模块误报传感器数据丢失的问题；

b. 目前的传感器融合数据源故障诊断的方式是传感器数据触发型的，每个传感器数据到达都会判断所有传感器的延时、自己的频率和主传感器的时间差，其中存在较多的字符串的运算，目前认为会影响传感器融合的性能和资源。

2. 方案设计

a. 主要叙述

i. 障碍物融合模块数据的故障管理独立于目前的融合模块，定频执行（100ms）；

ii. 单相机跟踪数据、雷达预处理数据和任务管理调度融合的数据，使用回调函数订阅topic的形式保存数据；

iii. 目前故障管理主要包含3个核对功能，单数据的最大最小频率核对、单数据的最大延时确认和单数据与主传感器的最大延时确认，此数据均和目前方案保持一致，核对顺序为最大延时、最大最小频率和主传感器的最大延时，一个核对失败不再核对后面的；

iv. 设计一个故障管理类和一个pavaro的组件，组建中有一个故障管理类的对象；

v. 只核对任务管理模块下发的需要核对的传感器数据。

a. 故障管理类的设计伪代码

</> C++ | 收起

```cpp
// 保存每一种传感器最新时间戳、计算频率的时间戳和计算频率的个数
struct SensorInfo {
    SensorInfo() {}
    SensorInfo(
        const uint32_t& latest_timestamp,
        const uint32_t& frequency_start_timestamp,
        const uint32_t& frequency_counts) :
            _latest_timestamp(latest_timestamp),
            _frequency_start_timestamp(frequency_start_timestamp),
            _frequency_counts(frequency_counts) {}
    uint32_t _latest_timestamp{0};
    uint32_t _frequency_start_timestamp{0};
    uint32_t _frequency_counts{0};
```

```
};

// 保存传感器核对的阈值
struct SensorMonitorParams {
    SensorMonitorParams() {}
    SensorMonitorParams(
        const float max_freq_threshold,
        const float min_freq_threshold,
        const float max_delay_threshold) :
            _max_freq_threshold(max_freq_threshold),
            _min_freq_threshold(min_freq_threshold),
            _max_delay_threshold(max_delay_threshold) {}

    float _max_freq_threshold{25.0};  // 最大频率阈值
    float _min_freq_threshold{15.0};  // 最小频率阈值
    float _max_delay_threshold{0.5};  // 最大延时阈值
};

class EmDatasouceFaultMonitor {
   public:
        // 0. 初始化所有私有成员变量
        void init() {}
        // 1.设置传感器数据信息，主要用于接收传感器数据后设置sensor_id和
timestamp;
        void set_sensor_data(sensor_id, timestamp) {
            // 1. 在map1中根据sensor_id 和timestamp存储不同传感器的最新数据
            set_sensor_id_2_timestamp(sensor_id, timestamp);
            // 2. 在map2中根据sensor_id 增加传感器数据的个数，用于计算频率
            set_sensor_id_2_counts(sensor_id);
        }

        // 2. 根据em_task任务设置每种传感器主要阈值
        void set_thres(params) {
            // 设置核对时候使用的阈值
        }

        // 3. 主要的执行核对程序
        void process_monitor() {
            // a. 核对每个传感器最新数据和当前时间差
            if(!check_sensor_delay()) {
                return;
            }
            // b.核对传感器数据的频率
```

```
56            if (!check_frequency()) {
57                return;
58            }
59            // c.核对和主传感器数据的时间差
60            if (!check_main_sensor_timediff()) {
61                return;
62            }
63            // d. 核对定位数据是否有效
64            if (!check_local_valid()) {
65                return;
66            }
67        }
68
69    private:
70        // 1. 核对传感器数据延时
71        bool check_sensor_delay() {
72            // 循环所有sensor_id_2_timestamp，分别计算当前时间和每个传感器最新数据的时间差，和阈值对比，大于报故障
73            }
74        // 2. 核对传感器数据的频率
75        bool check_sensor_frequency() {
76            // 循环获取频率开始的时间和当前时间的差值，大于1s，计算频率，如果不满足阈值条件报故障，重新设置当前时间为频率开始时间，传感器数据设置为0
77
78        }
79        // 3. 核对和主传感器数据的时间差
80        bool check_main_sensor_timediff() {
81            // 根据设置的主传感器名称获取最新主传感器的时间戳，然后循环所有传感器数据，计算差值是否满足，不满足报故障；
82        }
83        // 4. 核对定位丢失时间
84        bool check_location_valid() {
85            // 直接调用em接口核对定位数据是否异常，丢失数据不更新有效时间，同时判断丢失时间是否超过阈值，超过报故障，重新赋值
86        }
87
88        // 5. 设置传感器数据最新时间戳
89        void set_sensor_id_2_timestamp(){
90        }
91
92        // 6. 设置传感器数据频率计算的开始时间
93        void set_sensor_id_2_frequency_timestamp(){
94        }
95
```

```
        // 7. 设置传感器数据频率计算的传感器个数
        void set_sensor_id_2_counts(){
        }

        // 私有成员
        std::map<string, SensorInfo> _sensor_id_2_sensor_info_map;
        std::map<string, SensorMonitorParams> _sensor_id_2_thresh_map;
        float _location_lost_times;
        float _location_lost_times_threshold;
}
```

b. 独立的故障核对component的伪代码

</> C++ | 收起

```
class EmFaultMonitorComponent : public
::apollo::os::computing::TimerComponent(100000)
public:
    bool init() {
        // 1. 注册任务管理调度em的topic的回掉函数
        em_task_func();
       // 2. 设置雷达数据
        radar_callback();
        // 3. 设置相机数据
        camera_callback();
    }
    bool proc() {
        if (!_inited) {
            return;
        }
        _em_monitor.process_monitor();
    }

     // 1. 注册任务管理调度em的topic的回掉函数
    void em_task_func(){
        // 重置所有参数
        _em_monitor.reset();
        // 初始化所有参数
        _em_monitor.init();
        // 根据任务管理参数设置阈值
        _em_monitor.set_thres(max_frequency, min_frequency,
max_timediff);
        _inited = true;
    }
```

```
    // 设置雷达数据
    void radar_callback() {
        // 1. 设置传感器数据信息
        _em_monitor.set_sensor_data()
    }

    // 设置相机数据
    void camera_callback(){
        // 1. 设置传感器数据信息
        _em_monitor.set_sensor_data();
    }
private:
    // 收到任务管理之后设置为true
    bool _inited{false};
    EmDatasouceFaultMonitor _em_monitor;
```

3. 代码修改

评审：IDG-VOYAH-10183, optimize em fusion fault monitor

4. 仿真测试

5. 实车测试