海达数云操控服务软件设计文档

作者	时间	版本
唐嘉豪	**2022-01-19**	**2.0.0**

目录

1 文档内容概要及服务软件设计概要介绍

1.1 文档大纲

本文档编写基于 2022-1 月 MMS 操控软件服务版本，旨在梳理服务架构，明确软件重要流程及类封装设计。对于新接触代码员工，为其供一个代码架构梳理，以达到快速熟悉服务代码架构，理解相关类设计以及知晓服务中各个工作流程的效果，并且在需要开发新功能时能迅速定位方向，减轻开发压力。对于较为熟悉原有服务架构的同事，针对 2.0 版本新增内容，进行介绍，针对原有功能，提供指引。最后，在今后软件的设计上，互相探讨标准。

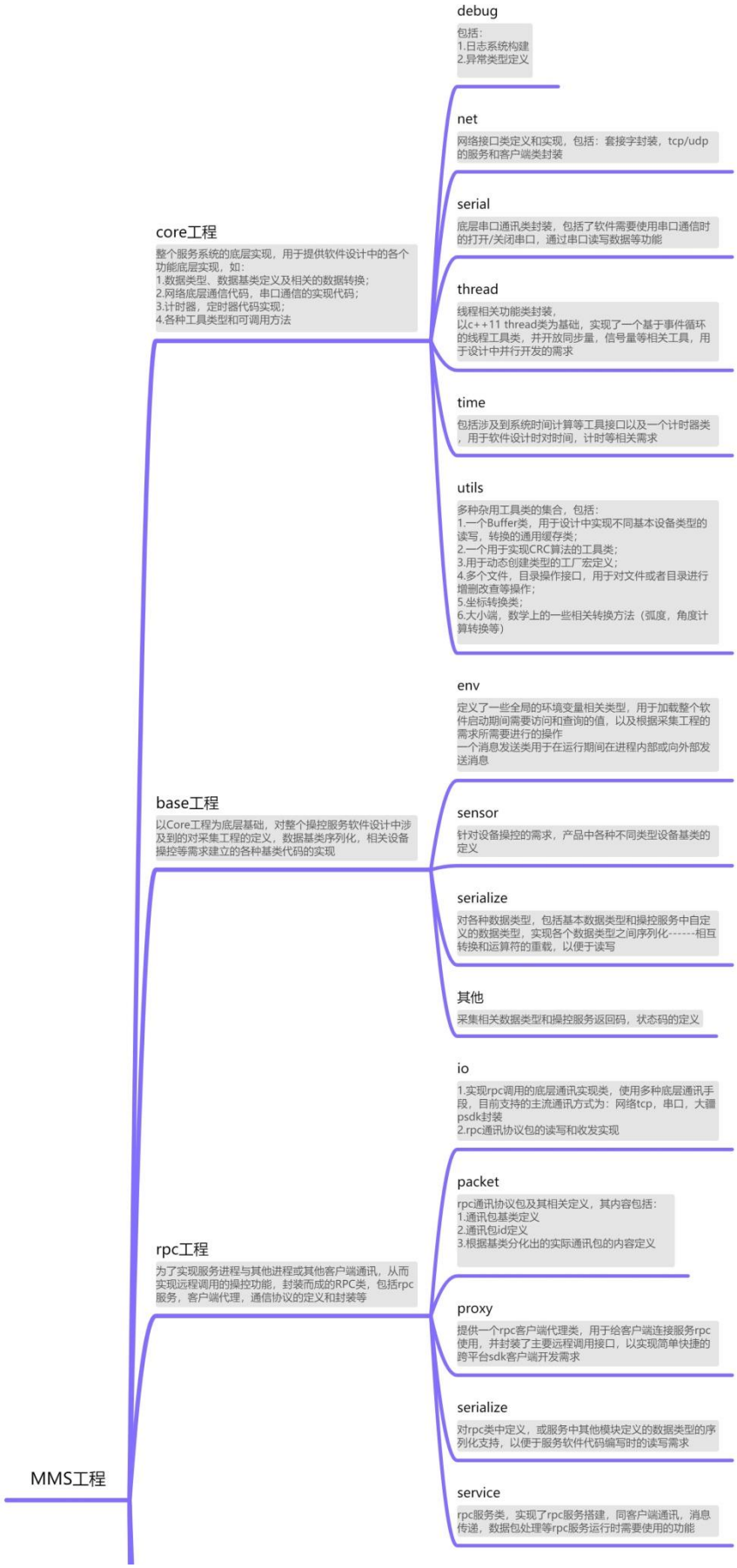
1.2 服务软件设计概要

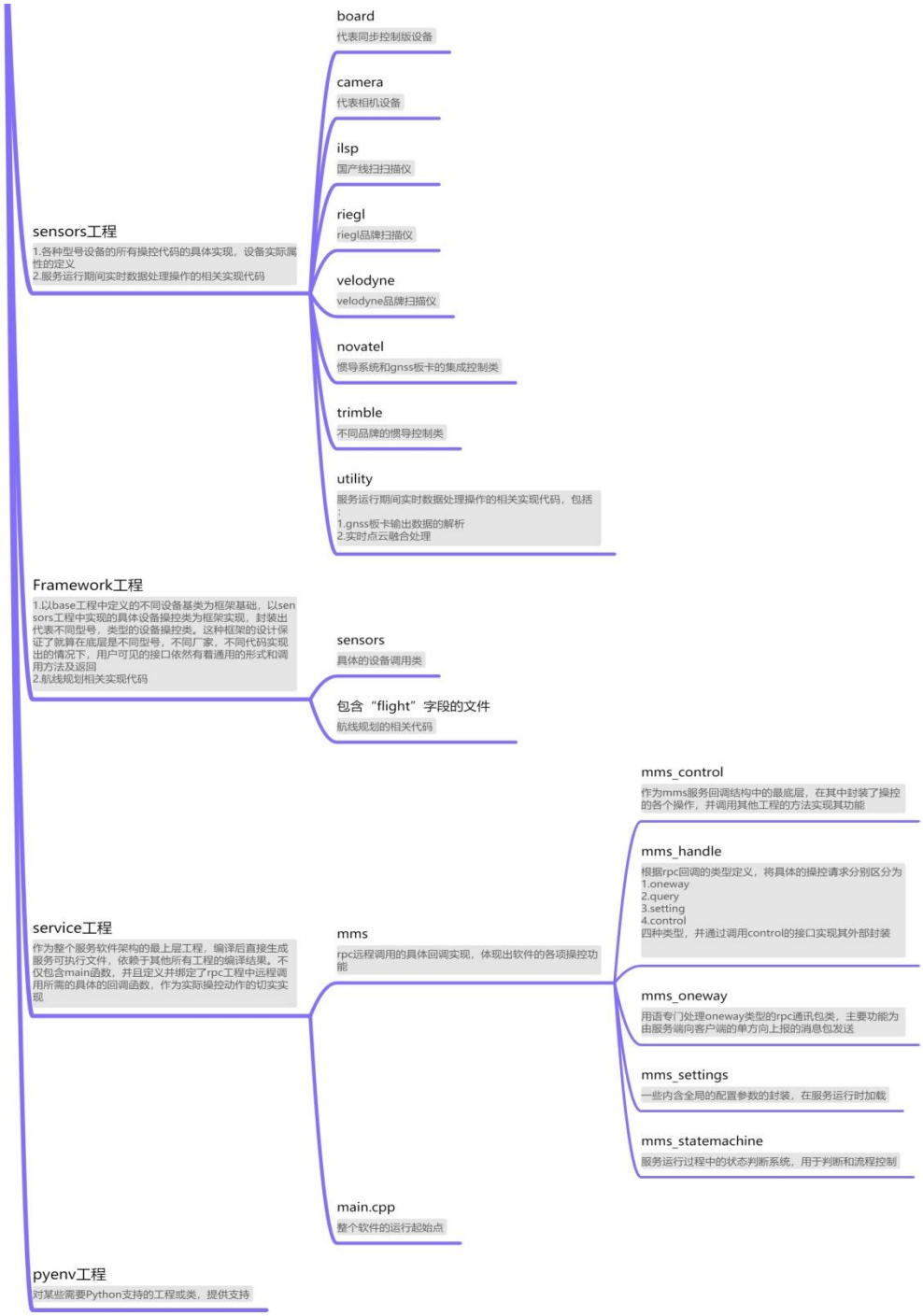
当前 MMS 服务底层设计中采用多层嵌套封装，架构采取多模块分工合作的方式。整体由多个子工程构成，各个子工程又通过统一的配置达到整合编译的效果。底层支持上，整个代码以 Qt 工程为基底，支持跨平台编译。

2 服务软件整体架构梳理

2.1 服务架构示意

服务以“mms”工程作为主工程，其下通过树形结构下分为“core”，“base”，“sensors”，“rpc”,“Framework”,“service”这 6 个主要子工程，并配套以“ctrl”,“ipservice”,“serialproxy”，“pyenv”这 4 个工具性质工程。各个子工程依次有不同分工，并包含众多内容，如下图所示。


MMS工程
core工程
整个服务系统的底层实现，用于提供软件设计中的各个功能底层实现，如：
1.数据类型、数据基类定义及相关的数据转换；
2.网络底层通信代码，串口通信的实现代码；
3.计时器，定时器代码实现；
4.各种工具类型和可调用方法
debug
包括：
1.日志系统构建
2.异常类型定义
net
网络接口类定义和实现，包括：套接字封装，tcp/udp的服务和客户端类封装
serial
底层串口通讯类封装，包括了软件需要使用串口通信时的打开/关闭串口，通过串口读写数据等功能
thread
线程相关功能类封装，
以c++11 thread类为基础，实现了一个基于事件循环的线程工具类，并开放同步量，信号量等相关工具，用于设计中并行开发的需求
time
包括涉及到系统时间计算等工具接口以及一个计时器类，用于软件设计时对时间，计时等相关需求
utils
多种杂用工具类的集合，包括：
1.一个Buffer类，用于设计中实现不同基本设备类型的读写，转换的通用缓存类；
2.一个用于实现CRC算法的工具类；
3.用于动态创建类型的工厂宏定义；
4.多个文件，目录操作接口，用于对文件或者目录进行增删改查等操作；
5.坐标转换类；
6.大小端，数学上的一些相关转换方法（弧度，角度计算转换等）
base工程
以Core工程为底层基础，对整个操控服务软件设计中涉及到的对采集工程的定义，数据基类序列化，相关设备操控等需求建立的各种基类代码的实现
env
定义了一些全局的环境变量相关类型，用于加载整个软件启动期间需要访问和查询的值，以及根据采集工程的需求所需要进行的操作
一个消息发送类用于在运行期间在进程内部或向外部发送消息
sensor
针对设备操控的需求，产品中各种不同类型设备基类的定义
serialize
对各种数据类型，包括基本数据类型和操控服务中自定义的数据类型，实现各个数据类型之间序列化------相互转换和运算符的重载，以便于读写
其他
采集相关数据类型和操控服务返回码，状态码的定义
rpc工程
为了实现服务进程与其他进程或其他客户端通讯，从而实现远程调用的操控功能，封装而成的RPC类，包括rpc服务，客户端代理，通信协议的定义和封装等
io
1.实现rpc调用的底层通讯实现类，使用多种底层通讯手段，目前支持的主流通讯方式为：网络tcp，串口，大疆psdk封装
2.rpc通讯协议包的读写和收发实现
packet
rpc通讯协议包及其相关定义，其内容包括：
1.通讯包基类定义
2.通讯包id定义
3.根据基类分化出的实际通讯包的内容定义
proxy
提供一个rpc客户端代理类，用于给客户端连接服务rpc使用，并封装了主要远程调用接口，以实现简单快捷的跨平台sdk客户端开发需求
serialize
对rpc类中定义，或服务中其他模块定义的数据类型的序列化支持，以便于服务软件代码编写时的读写需求
service
rpc服务类，实现了rpc服务搭建，同客户端通讯，消息传递，数据包处理等rpc服务运行时需要使用的功能



2.2 工程配置及项目第三方库依赖系统介绍

除了多工程分工协作外，整个 mms 系统采用统一的工程配置来对编译时的工程代码和第三方库依赖做出定义，其整体配置依托于 Qt 工程配置，即：以.pro 文件为各个工程的主要配置文件，辅佐以多个 pri 文件作为整体需求和依赖配置。

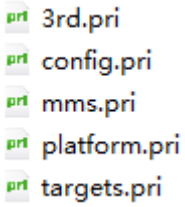
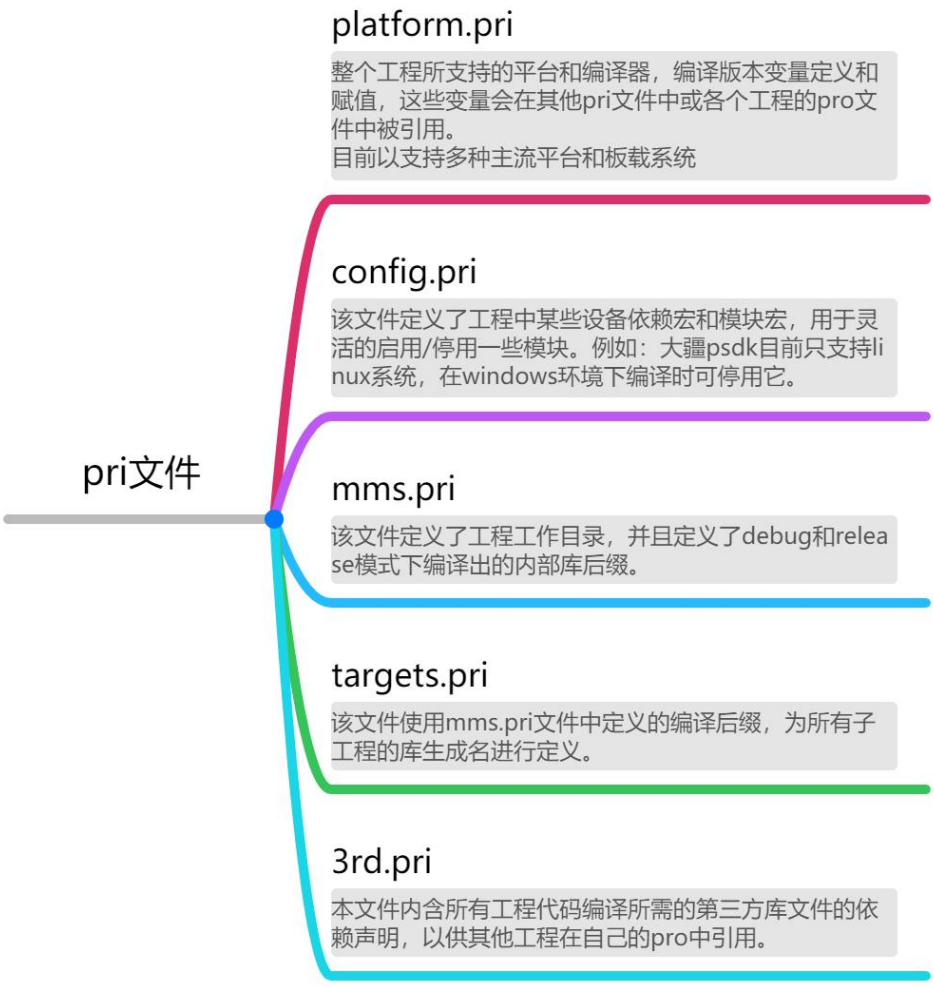


图 2-1　pri 工程配置文件

它们的内容和关系如下图所示：



3 重要工程流程介绍及服务软件设计标准

3.1 RPC 工程

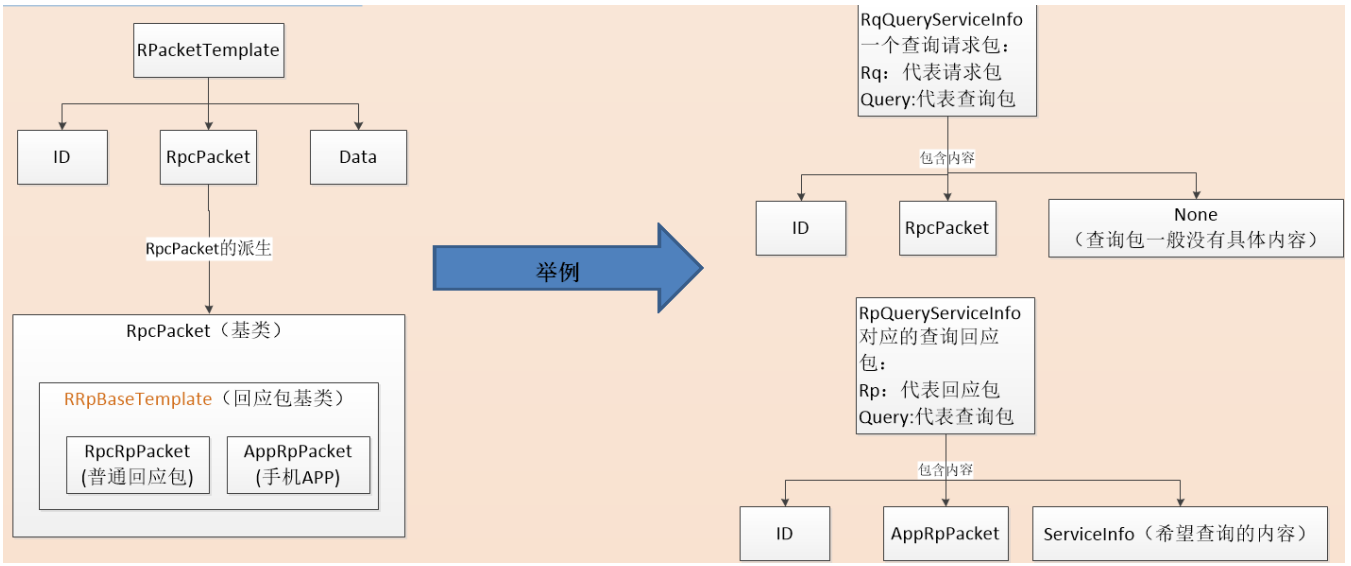
作为服务和其他进程、其他设备通讯的基础，rpc 工程提供了一个可远程调用的服务 service，不同的封装类和多个组件，便于软件设计人员方便地构造一个服务器端。

3.1.1 Rpc 通信包类型：RpcPacket

服务中 RPC 通信包分为：

请求包，回应包，和上报包三种。其中，请求和回应包互相对应。上报包可看做无回应的特殊请求包，用来服务/客户端单方向地向对端发送信息包。

在代码实现上，服务中通过模板类的方式定义一个个包，每个包享有自己独立的 id 和包的具体内容，并且含有一个 Packet 类型来判断该包是三种中的哪一种。



3.1.2 RpcPacketId 和 RpcClientId 的定义和内容

RpcPacketId 为服务中自定义的类，通过一个 32 位无符号 int 成员来定义了整个操控服务 rpc 通讯包的类型，其中二进制表示的不同 bit 数的含义如下：

Type：

分为 ONEWAY（上报），REQUEST（请求），RESPONSE（回应）三种。用来表明包的主要类型。

Layer：

分为 RPC （普通 rpc 包）,APP（同手机端通讯） 两种。用来表明包是普通 rpc 通讯包还是在手机操控端使用的通讯包。

Sublayer：

根据 Layer 的不同，有更详细包类型的区分。

Subid：
包定义时的 ID，每个包绑定一个固有 id。
Reserve：
保留字段，当前未使用。

RpcPacket ID				
3bit	3bit	6bit	6bit	16bit
Type	Layer	Sublayer	Reserve	subid

RpcClient ID 也为自定义的类，通过一个 16 位 int 来定义了 MMS 通讯中的 Client 类型。其中每位的代表意义如右图，其中：
Type：
分为 MONITOR（监视），NORMAL（普通），SUPER（超级客户）三种。用来表明包的主要类型。

Subid：
包定义时的 ID，每个已开通的客户端连接绑定一个固有 id。

RpcClient ID	
4bit	12bit
Type	Subid

3.1.3 建立一个 RpcService 的重要组件：RpcClient，RpcServer,RpcClientManager 和 RpcWorker

欲建立一个 rpc 服务端，需要通过实现 RpcService 类来实现。而 RpcService 是由四大组件组成的，即 **RpcClient 类**，**RpcServer 类**，**RpcClientManager 类**和 **RpcWorker 类**，它们各自作为 rpcservice 实现不同功能的重要成员，在一个 Rpc 服务的生命周期里共同发挥不同的作用。

3.1.3.1 RPC 服务器中的服务端建立类—RpcServer

RpcServer 类，作用是在需要时建立一个 rpc 服务，步骤类似于建立一个 tcp 连接的服务端，即在独立的线程中打开通讯通道，监听，等待客户端连接。每当接收到客户端来的连接

包（RpcConnect）后，调用 RpcClientManager 类建立一个新的 RpcClient 对象，并开启其读写线程。自身进入下一轮循环等待其他客户端连接。

3.1.3.2 RPC 服务器中与客户端通信收发类—RpcClient

掌管 RPC 服务中与每个已连接的客户端的通信。RpcService 可供多个远程客户端连接，就是归在一个 RpcClientManager 和多个 RpcClient 的共同作用下实现的。

RpcClient 在独立的线程循环中负责服务和这个客户端建立连接后的收发包。

3.1.3.3 RPC 服务的消息处理和转发回调—RpcWorker/AppWorker

RpcWorker 类实现了消息回调机制和事件处理机制，目的是为了跨模块地实现 RPC 调用具体功能。在前文中介绍到：操控服务是多模块分工合作来实现各种功能的。为了实现客户端远程控制服务对具体的物理设备实现多种多样的操控功能，而又不过度增加 rpc 工程模块的耦合性，采取了 RpcWorker 和 AppWoler 的设计。它们继承于 thread 线程类（在 core 工程中定义），将服务和客户端之间双向发送的通讯包抽象化为事件，并通过事件循环处理线程处理服务待发给客户端的数据包或者从客户端接收到的数据包。

对于从客户端（RpcClient）接收到的包，按照定义好的规则交给其他回调函数处理。通常这些回调函数都是定义在其他模块里的，并在整个 RpcService 的对象初始化时绑定（在本操控服务中，定义于 Service 工程的 mms 目录里）。

对于将要发给客户端的包，把这些包继续分发给 RpcClient 处理。（别忘了！3.1.3.2 中介绍到，具体一个包如何通过底层传输通道在服务和客户端之间传输时 RpcClient 的工作）

注：可以简单地把这两个 worker 类理解为服务/客户端之间的中转站。信息以通信数据包为载体，在 worker 中被简单分类后中转处理。同时，AppWorker 是 RpcWorker 的特化，用于和 APP 操控进行通信。

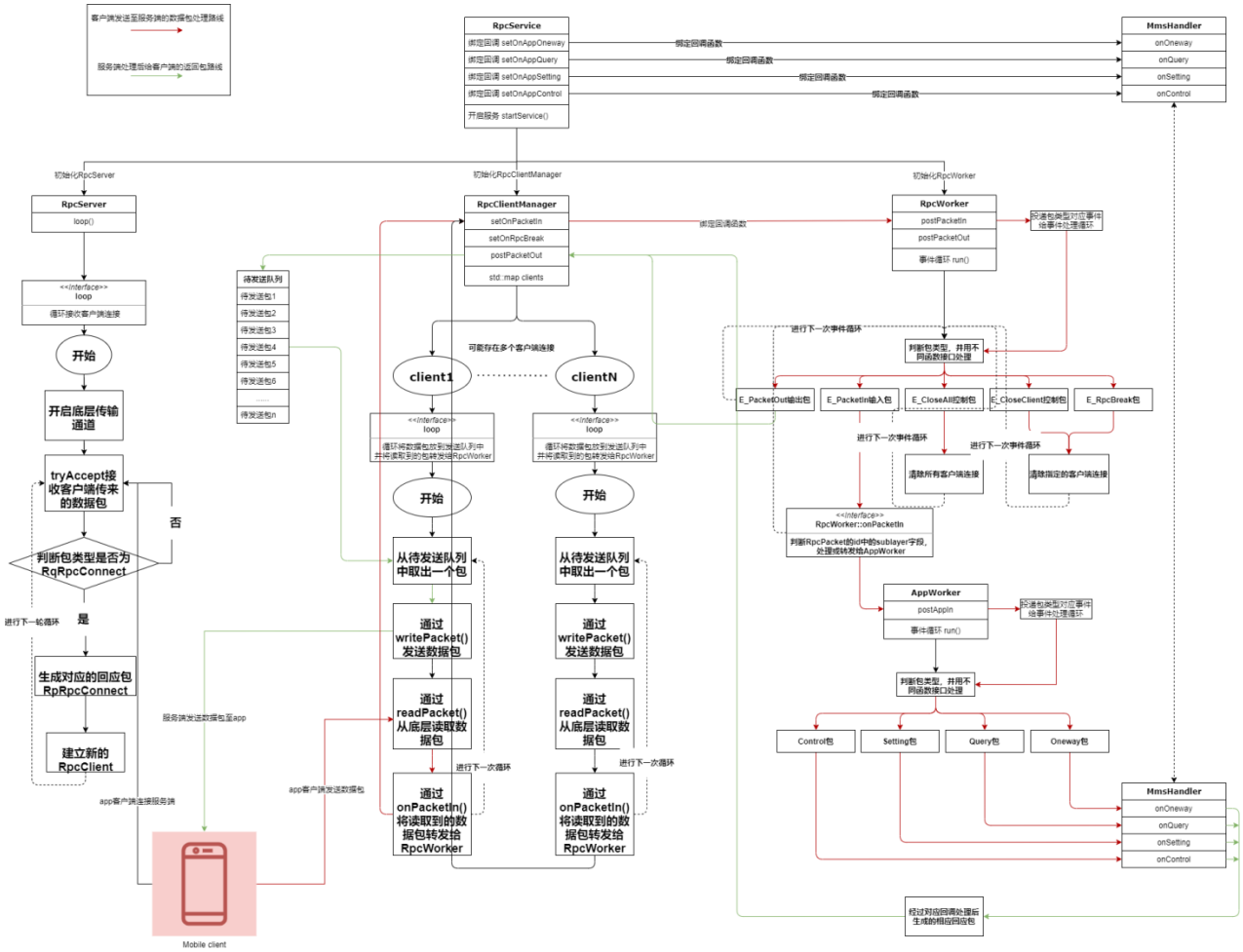
3.1.3.4RPC 服务的多客户端管理类—RpcClientManager

RpcClientManager 类用于实现多客户端连接的需求。通过绑定各个客户端接收/发送数据包相关回调，实现服务运行中的和各个客户端的数据包收发。

一个明显的例子就是，3.1.3.3 节中提到的 worker 的中转功能，其实是要通过 RpcClientManager 类，在多个客户端同时连接时才能知道数据包要发给哪个客户端，或者知道数据包是来自于哪个客户端。

3.1.4 Rpc 通信读写包逻辑流程

整个操控服务的远程调用流程和逻辑复杂，涉及到 rpc 工程和 service 工程的多个类，我们可以以通信数据包为单位，将客户端发送包和服务回应包在整体流程的走向为路线，绘制出流程图如下。其中，红色线标记着从客户端发来的包在服务里的层层处理路线，绿色线表示服务生成的对应包是如何经过服务返回给客户端的，它们便共同组成了一次完整的数据包传输链路。



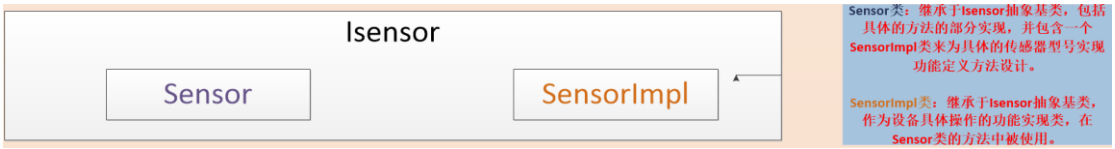
3.2 Sensors，base，Framework 工程中的具体设备类型设计

3.2.1 具体设备类型设计概要

Sensor 相关类在 MMS 服务中作为设备类使用。实现软件层面上和硬件设备通信以及对硬件设备进行操控的功能。

要设计出一个对应于现实中的具体设备类，操控服务代码采用多级继承配合实现类复合使用的设计方式。即：一个具体的设备 A 封装类根据现实中的类型，继承于相应的 Sensor 基类，其基类规范了这一种类型的行为。并以复合的方式将行为的实现----对应的 SensorImpl 类作为成员，以供接口调用。

注：sensor 基类和 SensorImpl 基类于 **base 工程**的 sensor 目录中定义并实现，具体到特定设备 A 的 SensorImplA 于 sensors 工程中定义并实现，最后，具体的设备 A 封装类 SensorA 则在 **Framework 工程**中的 sensors 目录中定义实现。

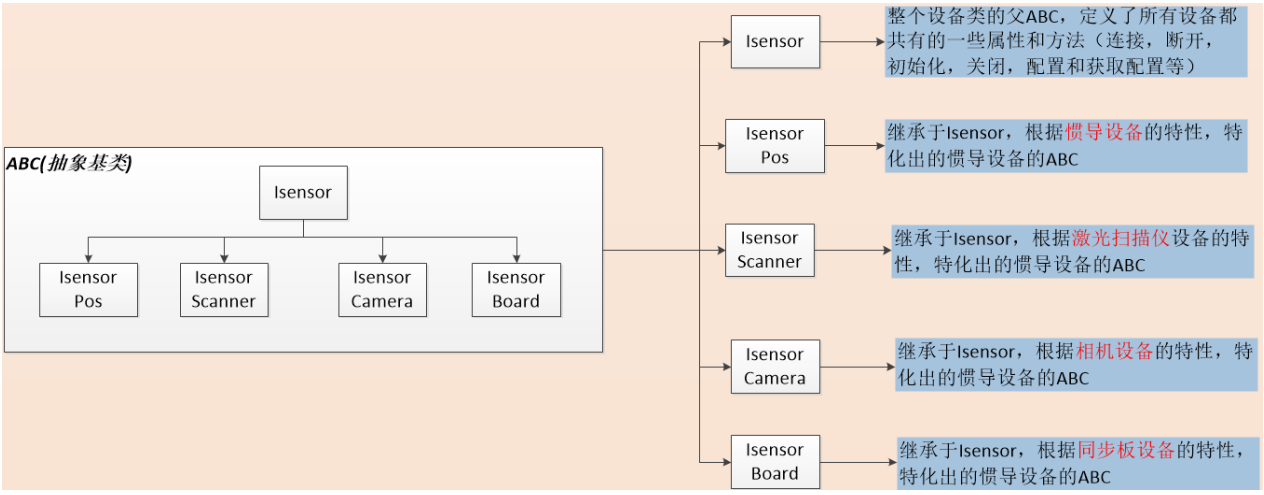


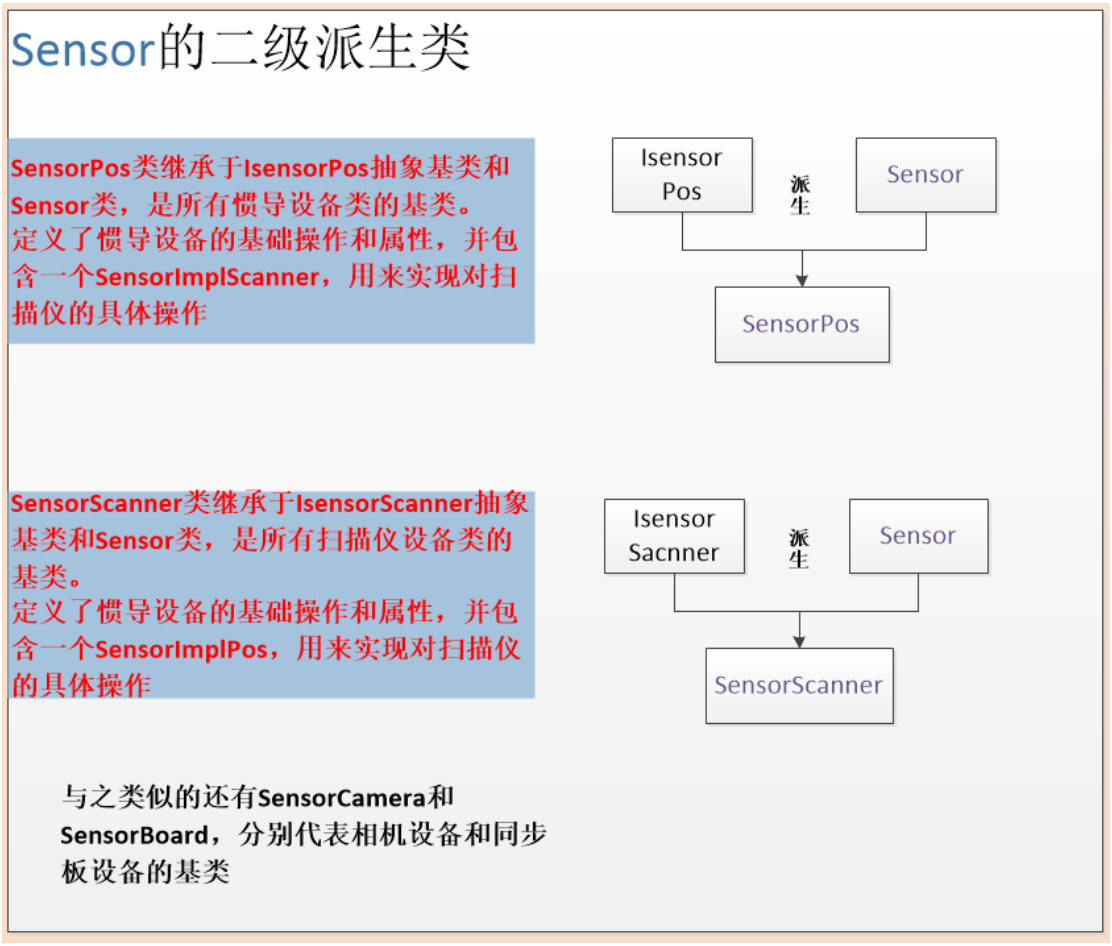
3.2.2 Sensor（设备类型）基类封装思路

作为一个具体设备类的组件之一，在代码设计上，采用了三层继承封装。

首先，MMS 服务采取 ABC（抽象基类）的方式来设计**设备抽象基类 ISensor**，它定义了作为一个硬件设备所必须拥有的常用行为和属性（如连接 connect，断开连接 disconnect，初始化 initial，关机 shutdown 等）。再将一个固定的 id 属性和它绑定，成为不同设备唯一的标识，这就是设备类的二级派生：**Sensor 基类**。同时根据不同实际设备类型（惯导，扫描仪，同步板，相机等）创建相应的子类，作为第二级派生中规定设备行为的**基类 IsensorPos，IsensorScanner，IsensorCamera,Isensorboard**。最后，采用多重继承的方法，将 **Sensor 基类**（内含每个设备不同的标识）与**对应的 Isensor**（规定了不同类型的物理设备所规定的行为，如：惯导类设备有获取位置，卫星等数据的接口而扫描仪类设备没有）结合，派生出第三级派生类：SensorBoard，SensorCamera，SensorScanner 和 SensorCamera。

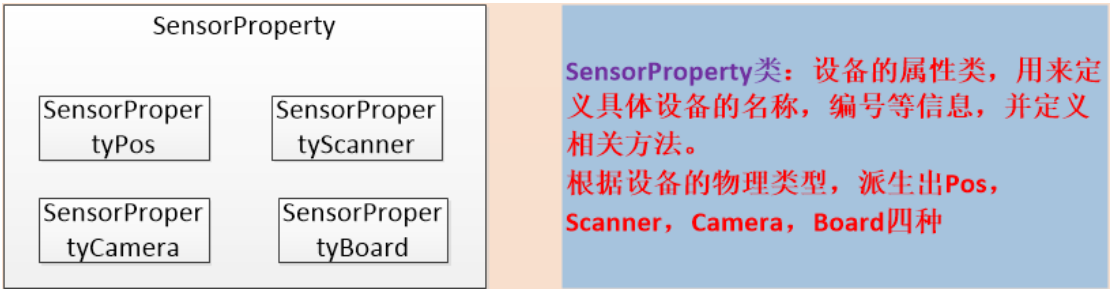
注：各层基类均在 base 工程中的 sensor.h 中声明。





3.2.3 SensorProperty（设备属性类）基类封装思路

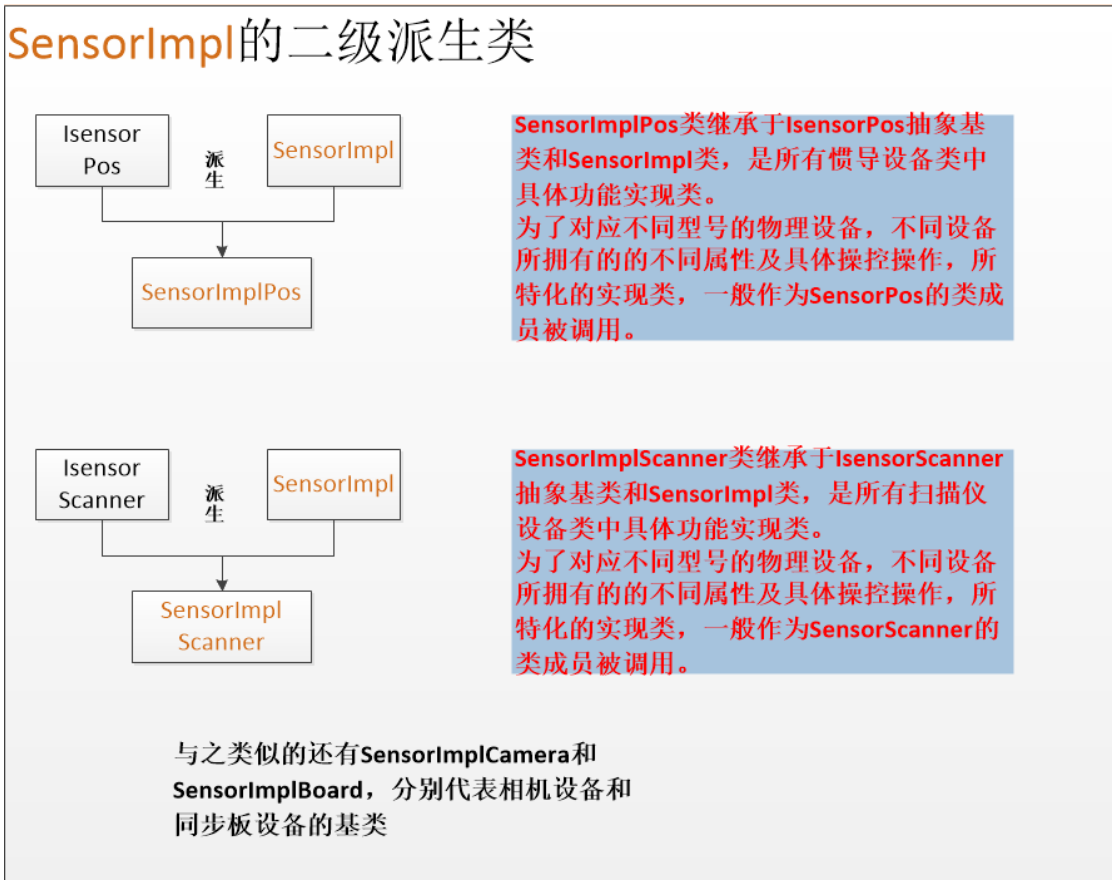
作为具体设备类的组件之一，SensorProperty 基类承载了设备的具体属性。设计架构上，同 sensor 基类相同，采用两级派生，最上层为 **SensorProperty 类，**它的内容包含所有物理设备都会拥有的“名称”，“序列号”，“设备 id”等属性。再由其派生出子类 PropertyPos，PropertyScanner，PropertyCamera 和 PropertyBoard 二级基类。



3.2.4 SensorImpl（设备功能实现）基类封装思路

作为一个具体设备类的另一个组成组件，SensorImpl 类实现了 sensor 基类中规定的所有设备行为。设计架构上，与各种 sensor 基类类似，采用三层派生架构：其中 Isensor 作为最顶层的抽象基类，规定了待实现的行为接口。然后派生出第二级 SensorImpl，最后根据不同物理设备派生出 SensorImplPos 等实际功能实现类。并且拥有一个具体的属性类（由 **SensorProperty 类**或其二级子类派生出的特化类型）。

注：相关内容在 base 工程的 sensor_impl 文件中声明



3.2.5 设计一个具体的设备类

步骤一：

进行非代码层级的现实分析。该设备 A 是惯导设备，激光扫描仪，同步板设备还是相机设备？

步骤二：

假设设备 A 是其中一种设备，假设为 Scanner。

步骤三：

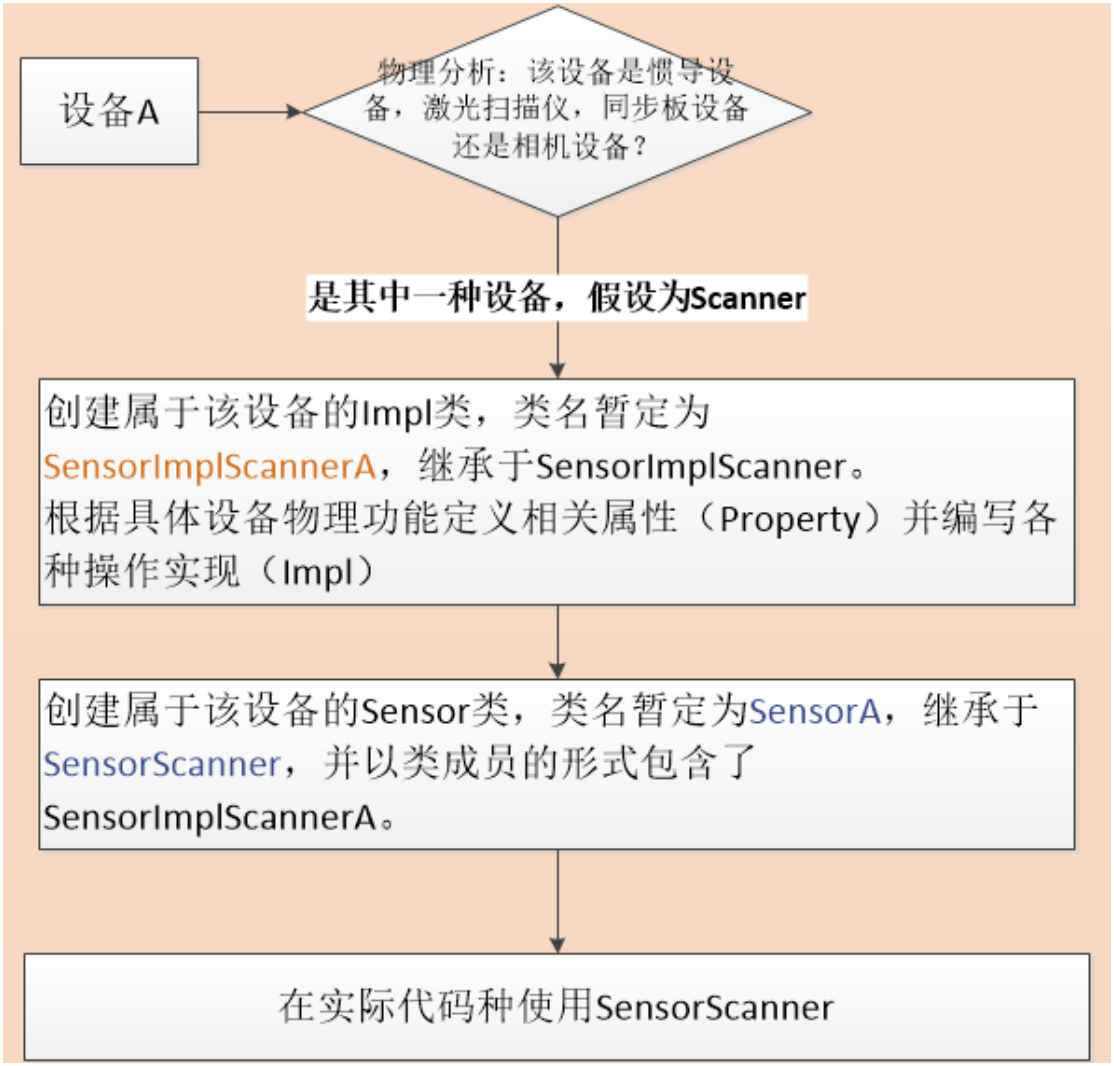
1.创建属于该设备的 Impl 类，类名暂定为 SensorImplScannerA，继承于 SensorImplScanner。

根据具体设备物理功能定义相关属性（Property）并编写各种操作实现（Impl）。该步骤中定义的代码应放置于 sensors 工程中的对应目录中。

2.创建属于该设备的 Sensor 类，类名暂定为 SensorA，继承于 SensorScanner，并以类成员的形式包含了 SensorImplScannerA。并根据需求重写继承于基类的虚函数。该步骤定义的代码应置于 Framework 工程中的对应目录中。

步骤四：

于其他工程需要对该设备 A 进行操控时使用 Framework 工程中封装的 SensorA 类。



3.3 航线规划设计及流程介绍

3.3.1 服务航线规划设计概要

操控服务涉及到的航线规划相关功能由位置预测算法模型和航线相关功能设计两部分组成。

前者由一个多级继承的类模板 Prediciton 负责，定义于 Framework 工程的

positionprediction 文件中。它的主要功能是根据从惯导设备获取的当前位置，航向角度，速度等参数判断机载设备在飞行时的出/入航线和曝光点预测的任务。

后者通过多个服务自定义类来实现了操控加载航线，移除航线执行航线，获取航线实时状态等具体的功能。这些类定义于 Framework 工程的 flight_excutor，flight_excutor_manager，flight_manager 等文件中。

另外，在 rpc 工程的 serialize 目录中的 dt_app_control 文件里，定义了航线规划功能所用到的自定义数据类型和它们在服务和客户端通信时的底层读写方式。

3.3.2 位置预测算法模型类—Prediciton

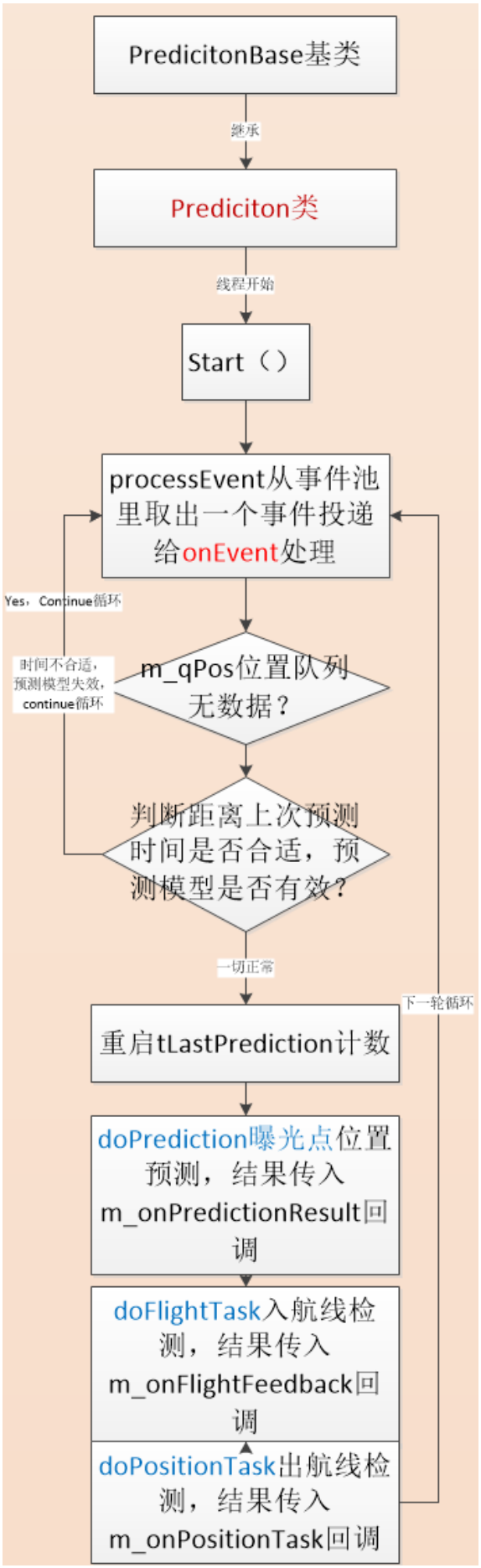
Prediction 类，继承于 Thread 类。以事件循环的方式在线程中处理航线相关的一个个事件。

重要类成员主要包括：

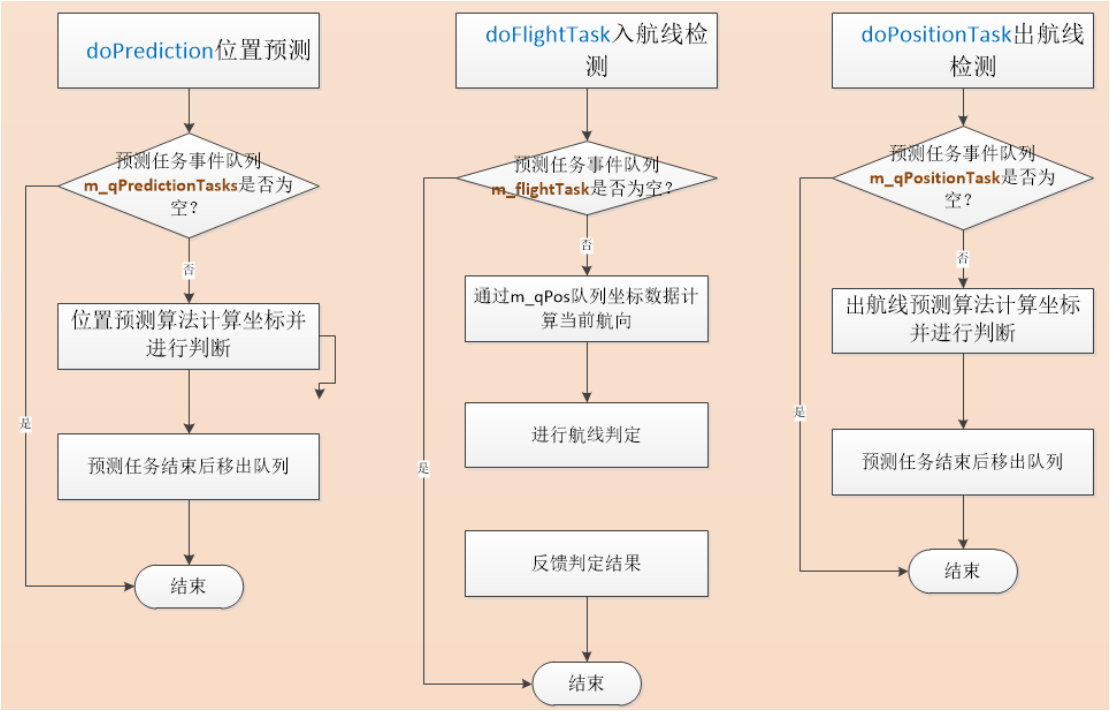
1.当前航线位置，航线状态判定的函数指针（用来后续封装中回调函数），

2.航线执行过程中不断变化的数据集合（以队列或数组的形式保存，在出入航线等算法中进行判断当前航线执行的情况）

它的判断流程如下图：


PredicitonBase基类
继承
Prediciton类
线程开始
Start（）
processEvent从事件池里取出一个事件投递给onEvent处理
Yes，Continue循环
m_qPos位置队列无数据？
时间不合适，预测模型失效，continue循环
判断距离上次预测时间是否合适，预测模型是否有效？
一切正常
下一轮循环
重启tLastPrediction计数
doPrediction曝光点位置预测，结果传入m_onPredictionResult回调
doFlightTask入航线检测，结果传入m_onFlightFeedback回调
doPositionTask出航线检测，结果传入m_onPositionTask回调

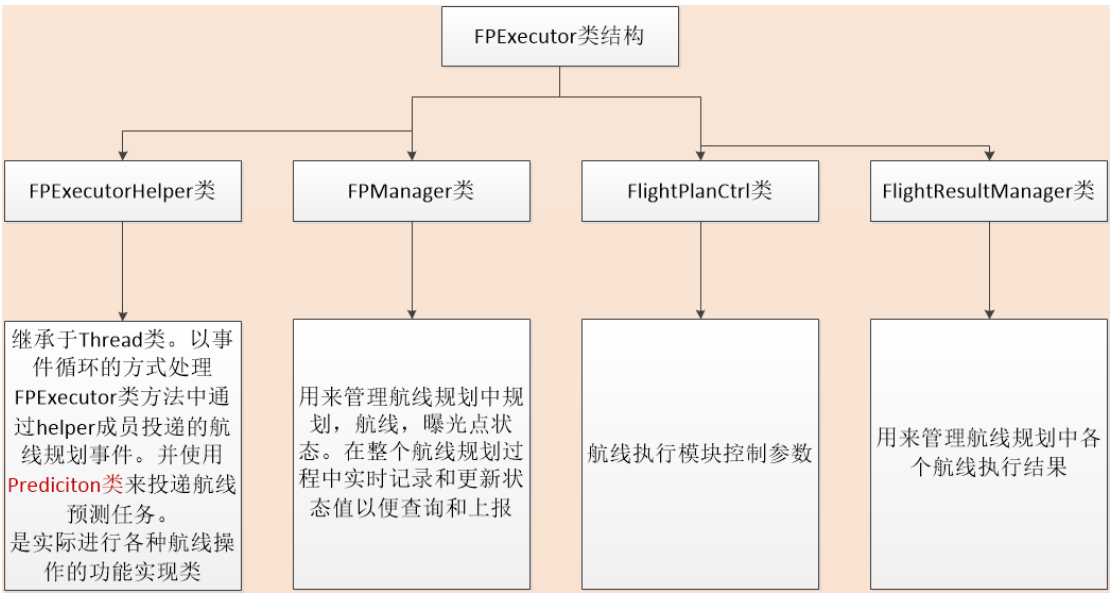
其中，doPrediction()，doFlightTask()，doPositionTask()三个主要的预测任务分别流程如下：



3.3.3 航线功能实现类—FPExecutor

FPExcutor 类：

航线规划接口类的外部封装，提供给 MMS 的 RPC 服务中回调使用的航线规划相关操作接口。内部包含 FPExecutorHelper 类，FPManager 类，FlightPlanCtrl 类和 FlightResultManager 类用于细化各个实际功能。其各块的主要工作和整体结构如下：



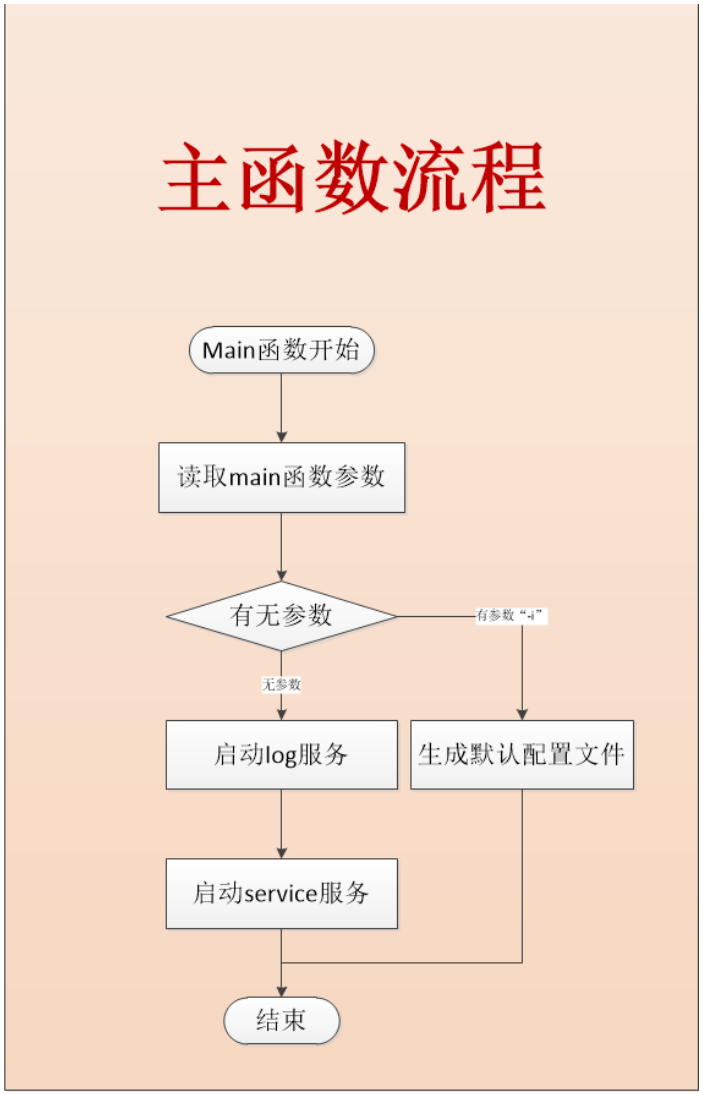
3.4 Service 工程

3.4.1 Service 工程设计概要

service 工程作为整个服务工程的出口，也是主函数 main 所在的工程。在其中的 mms 目录中定义了所有操控可供外部远程调用的接口和接口的实现，它们一方面是通过调用其他不同模块工程中定义的各个类型和设计结构（如 Framework 工程中定义的设备操控类，base 工程中定义的各种工程管理类等），另一方面是和 rpc 工程中的 RpcService 回调所绑定，实现客户端对服务的远程调用（可参考 3.1.3 中 RpcService 各模块的介绍理解）。

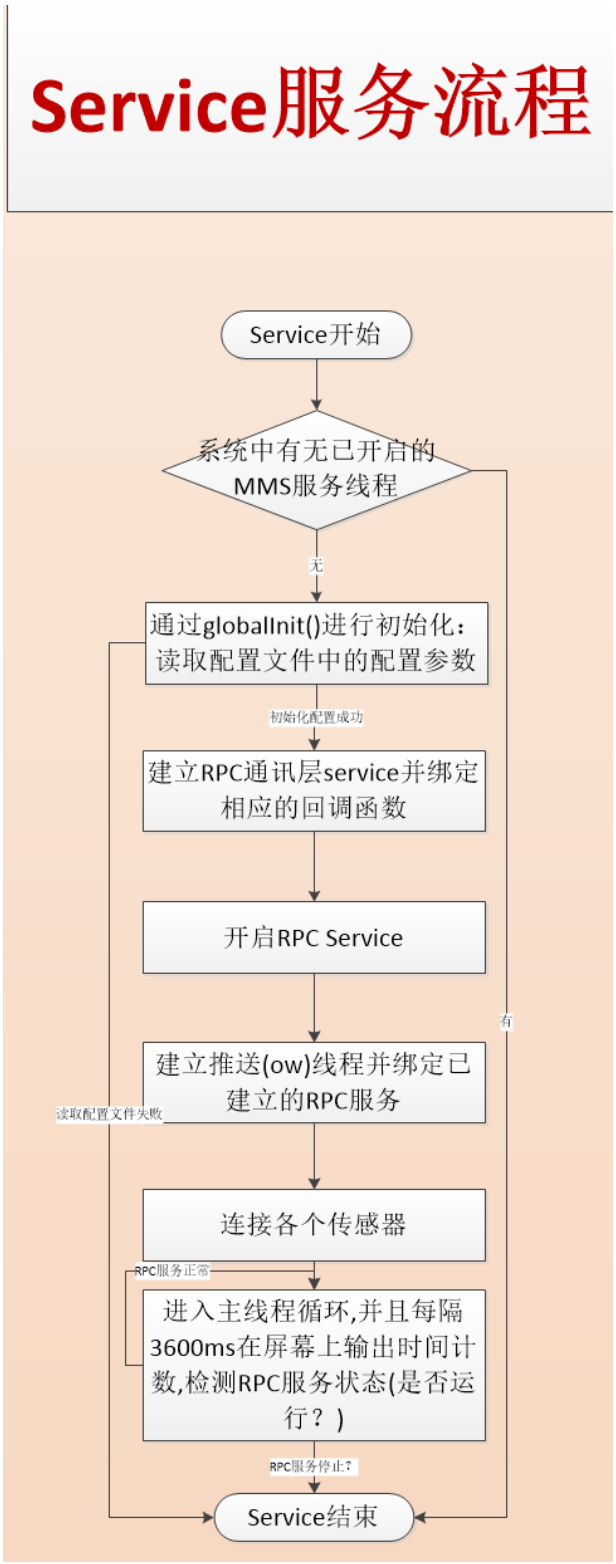
3.4.2 服务 main 函数流程

main 函数中通过判断命令行参数来确定启动服务或是实现其他辅助功能，流程如下:



而其中 service 服务流程如下：

Service服务流程



3.5 操控服务异常码--MmsResult_ID 类设计

3.5.1 服务异常码设计介绍

服务 2.0 及以后版本加强对服务操作中的返回码要求，以便于帮助研发人员快速定位问题出现原因和地点。

MmsResult_ID 类用于描述服务中各种接口和对设备操控的返回，并根据错误情况的类型定义了不同异常码值。

MmsResult_ID 类的主要内容为一个 16 位无符号的 int 成员，其各个字段的含义如下图：

MmsResult_ID		
4bit	4bit	8bit
type	reserve	resultid

其中，type 为异常类型，当前有 3 种类型。Reserve 为保留字段，以供后续开放新功能需求使用。Resultid 为具体的异常码值，范围为 0-255。

注：具体异常代码定义可参考《MMS 操控服务异常代码定义》文档。

4 一个示例：“如何快速熟悉服务代码并开始参与软件功能编写？”

4.1 第一步：明确开发需求，找准主要对应工程模块

根据具体的开发需求，将开发工作分类。例如：

如果当前的需求是针对一款全新设备开发对应的操控功能，就应该以 Sensors 工程，Framework 工程为主要开发工程，在其中以服务 3.2 中的规则编写对应功能代码。

如果当前需求需要对底层代码功能进行修改，应以 core 工程，base 工程为主要开发工程。

如果当前需求集中在客户端和服务之间通讯相关，应以 rpc 工程和 service 工程为主要开发工程。

4.2 第二步：新增功能类型，尽量避免修改原有定义类

由于目前服务代码设计中慢慢偏向多扩展，多兼容的开发模式，当涉及到新功能，新设备定义时应当在设计时考虑高复用性，低耦合性的功能开发和接口设计。同时，如果涉及到同方向，不同类别的功能模块开发，应避免在原有已定义完善的类型中修改，而是定义与之同级的新类型，并尽量统一接口。

4.3 第三步：依靠原有层级设计架构，进行新的需求功能设计

当前服务代码多处采用自上而下多层级的设计模式，一方面是为了降低代码耦合性，另一方面也是减轻开发同类型类似功能时的开发周期。当开发需求为已有功能的同级别，同性质工作时，依赖已有的层级架构进行设计，在对细节代码较为陌生的情况下，是一个快速开发和熟悉的方法之一。

5 服务编码规范及相关注释需求

5.1 服务中代码注释规范

为了提高代码可读性和可移植性，结合《海达数云 C++注释规范 V1.0》文档内容，定义如下规范：

一、针对自定义类型所添加的功能介绍注释

```
//-------------------------------------------------------
// EOutOfBoundary
//-------------------------------------------------------
/**
 * @brief The EOutOfBoundary class一般在数据越界时被抛出
 * @name :EOutOfBoundary异常类
 * @arg info：该异常的描述字符串
 * @arg idx ：越界时的位置计数值
 * @arg len ：数据存储数组的长度
 * @author :TJH
 */
class EOutOfBoundary : public std::exception
{
public:
    EOutOfBoundary() _E_NOEXCEPT;
    EOutOfBoundary(const char* info) _E_NOEXCEPT;
    EOutOfBoundary(const char* info, size_t idx, size_t len) _E_NOEXCEPT;
    EOutOfBoundary(const EOutOfBoundary& other) _E_NOEXCEPT;
    EOutOfBoundary& operator=(const EOutOfBoundary& other) _E_NOEXCEPT;

    virtual const char* what() const _E_NOEXCEPT;
```

对于自定义类型，应紧跟在类定义的上方，必须采用多行注释详细描述，注释描述标准参考如上图，应至少有以下几项内容：

@class ：类名

@brief：简要介绍，包括设计用途和简要使用方法等

对于涉及复杂的类型，还可以添加：

@details：详细介绍，

@ingroup：模块名，

@code：在注释中开始说明一段代码，

@note：一些提示信息，描述一些可能需要注意的问题等字段。

有关这些字段的具体内容和规范，可参考《海达数云 C++注释规范 V1.0》文档 3.2 节内容。

二、类文件针对函数接口的相关注释

```
/**
 * @brief Logger::isOpened 判断log文件是否打开
 * @return true:log文件打开 false：log文件未打开
 */
bool isOpened() const;
/**
 * @brief Logger::elapsedFromLastLog 获取距离上次log记录的时间
 * @return 距离上次log记录的时间间隔
 */
i64 elapsedFromLastLog() const; // ms
/**
 * @name:initLogger
 * @brief Logger::initLogger 初始化Logger
 * @param prefix：log文件标志名
 */
void initLogger(const char* prefix);
```

```
/**
 * @brief read：根据偏移量读取分包传输的rpc包
 * @param buf:读缓冲
 * @param len:缓冲区长度
 * @param MinPacketLen：缓冲区偏移量，含义为RPC包的包头长度
 * @return 读取的字节数
 */
u32 read(u8* buf, u32 len, u32 MinPacketLen);
```

对于定义的函数接口，函数定义注释与函数实现注释必须保持一致，为简便注释内容，一般直接将函数定义注释拷贝粘贴在函数实现上方，内容保持完全一致即可。其内容包括

@brief：该函数实现的功能

@param：函数所需参数，若同时包含输入和输出，可用[IN]/[OUT]在每个参数介绍中指出。

@return：函数的返回值，以及其含义。

三、针对代码中重要算法和处理流程涉及到复杂的逻辑判断时的介绍注释

```
//在读包前先将所有待发送包发送
if (!_qOutPackets.empty())
{
    //控制每次发送包的间隔，减小发送的瞬时带宽需求
    if(etSend.elapsed() < 100)
    {
        continue;
    }
    // pop a packet
    wpacket = popPacketOut();
    if (!wpacket) {
        continue;
    }

    // send the packet
    if(wpacket->packetId().isAppOneway())
    {
        /*对于oneway推送包，有特殊的发送逻辑来判断是否发送该包：
         * 1.app客户端初始化未完成前，停止oneway包的发送，
         * 2.当收到APP客户端发送的控制请求包时，在对应的回应包发送回APP前，停止oneway
         * 3.其他情况下，oneway包正常发送
         */
```

```
/*
 *实行点云赋色步骤:
 * 1.相机参数设置,包括:
 * ------设置相机内参路径
 * ------设置相机杆臂值文件路径
 * ------设置相机与载体坐标系的杆臂值
 *
 * 2.设置影相文件夹路径
 *
 * 3.开启线程运行process_pcd_colorize,处理点云
 *
 * 4.将获取的照片文件名和syn信息加入处理队列(影像文件需按1.jpg,2.jpg，3.jpg...格式输入
 */

// 处理赋色过程
void process_pcd_colorize();

// 添加syn记录
void addSynRecord(const string& syn_str);

// 添加影像数据
//  image_name影像名称不带路径
void addImageData(const string& image_name);
```

涉及重要流程，或逻辑复杂处理过程时，应在开始处或重要判断处上方添加相关注释，说明判断方法，理清流程步骤。