第一届
低空经济智能飞行管理挑战赛 性能赛
SDK 使用说明（1.3）

2023.9.22

1．概述

mtuav_sdk 用于接入美团无人机竞赛系统。它能够模拟无人机规划调度系统，进而控制无人机执行各种任务，其主要功能包括：登陆系统、加载地图、获取无人机参数、货物以及障碍物信息，下发航线，执行飞行计划等。

2. 系统配置

2.1 硬件要求

- **最低配置**

  CPU：四核 Intel 或 AMD 移动处理器（Intel i5-8550U ，主频 1.8 GHz 或 AMD Ryzen 5 3500U，主频 2.1 GHz）

  GPU：Intel(R) UHD Graphics

  内存：8 GB（本机 Ubuntu）/ 16 GB（虚拟机）

  存储：100 GB（强烈推荐 SSD）

- **推荐配置**

  CPU：六核 Intel 或八核 AMD 移动处理器或更好（Intel i7-10750H，主频 5.00 GHz 或 AMD R7-5800H，主频 3.20 GHz）

  GPU：专用 Nvidia 或 AMD GPU（Nvidia GTX 1060 或 AMD RX 480，以及更高）

  内存：16 GB（本机 Ubuntu），不推荐虚拟机

  存储：500 GB（强烈推荐 SSD）

- **对于本地算法开发，配置越高越好**

2.2 系统要求

Ubuntu 18.04 及以上，安装 cmake，配置 C++ 编译环境等。

正式比赛时，参数者需要使用 SDK 连接我们比赛云端系统进行比赛任务。而为了方便参赛者的开发和调试，我们也把云端系统打包成了一个独立的单机版环境（Docker Image）。具体获取及使用方法请见 6.5。

3．SDK 及地图简介

- 以 sdk 包 mtuav_sdk.7z 为例，解压后目录结构如下：

```
mu@mu-System-Product-Name:~/workspace/sdk$ tree
.
├── api
│   ├── mtuav_sdk_export.h
│   ├── mtuav_sdk.h
│   ├── mtuav_sdk_logging.h
│   ├── mtuav_sdk_map.h
│   ├── mtuav_sdk_planner.h
│   └── mtuav_sdk_types.h
├── CMakeLists.txt
├── example
│   └── sdk_test_main.cpp
├── libs
│   ├── libmtuav_sdk.go.so
│   └── libmtuav_sdk.so
├── map
│   ├── competition_map.bin
│   └── test_map.bin
└── visualization
    └── test
        ├── meta_data.pb.txt
        └── voxel.map.txt

6 directories, 14 files
```

- **api**：提供了 SDK 头文件，包括 log 系统，航线规划器，地图以及所需数据结构，在使用时只需要 include "mtuav_sdk.h"。
- **example**：一个简单的开发 demo，供参考。
- **libs**：使用 SDK 所需的共享库。
- **map**：存放了比赛所需的地图文件，其中 competition_map.bin 是比赛用地图，开启在线系统后使用此地图，test_map.bin 是调试用地图，仅供单机版使用。mtuav::Map 可以加载此地图。
- **visualization**：test 目录下使用测试地图的可视化文件。启动单机版，在浏览器中打开可视化网页后，选择加载此文件。

- 导入 SDK 的 CMakeList 示例如下

```
CMAKE_MINIMUM_REQUIRED(VERSION 3.8)

PROJECT("mtuav-competition")

set(CMAKE_C_FLAGS "${CMAKE_C_FLAGS} -fvisibility=hidden -pipe -fPIC -pthread")

set(CMAKE_CXX_FLAGS "${CMAKE_C_FLAGS} -std=c++11\
    -fthreadsafe-statics -fvisibility-inlines-hidden")

INCLUDE_DIRECTORIES(api/)

LINK_DIRECTORIES(libs/)

ADD_EXECUTABLE(mtuav_sdk_example example/sdk_test_main.cpp)
TARGET_LINK_LIBRARIES(mtuav_sdk_example mtuav_sdk)
```

4. SDK 中的数据结构

开发所需的数据结构均定义在 mtuav_sdk_types.h 头文件中，以下为主要的数据结构

4.1 飞行规划：FlightPlan

```
typedef struct FlightPlan {
  FlightPlanType flight_plan_type;      // 飞行模式
  FlightPurpose flight_purpose;         // 飞行目的
  std::string flight_id;                // 标识航线，一个 id 只能执行一次，用于现在同一条航线的多次规划
  uint64_t takeoff_timestamp;           // 航线起飞的时间（UTC+8 时间），单位是毫秒
  std::vector<Segment> segments;        // 每条航线上的飞行片段
  std::vector<int> target_cargo_ids;    // 指定装载/卸载的货物 id，注意：装载失败，卸载到错误位置不会有通知
} FlightPlan;
```

关于 FlightPlanType：

```
enum FlightPlanType {
  // 航点模式，飞机按照离散的航点进行飞行
  // 每两个航点构成一个行段，飞机都会经历一次加速和减速的过程，切加减速都以加大加速度来完成
  // 其中起飞和降落为独立阶段，第一个和最后一个航段为起飞和降落航段，必须垂直与地面向上或向下
  // 在飞行计划的执行过程中，也可以更改飞行计划，执行流程类似
  // 飞机执行前会检测飞行计划，如果明显超出飞行能力，会规划失败
  PLAN_WAY_POINTS = 0,
  // 轨迹模式，规划者需要计算稠密的轨迹点，指定每个轨迹点位置、时间、速度以及加速度
  // 轨迹模式下飞机飞行效率更高，更高更快完成任务，因此获得奖励也更高
  // 轨迹模式下，起飞和降落阶段也需要通过轨迹来规划，仅要求垂直起降
  PLAN_TRAJECTORIES = 1,
};
```

关于 FlightPurpose：

```
enum FlightPurpose {
  // 无特定目标，可用于调拨飞机
  FLIGHT_COMMON = 0,
  // 目标装载货物的航线，此航线飞机落地后，会自动装载地面货物当指定货舱
  FLIGHT_TAKE_CARGOS = 1,
  // 目标配送货物的航线，此航线飞机落地后，会自动卸载指定货舱的货物
  FLIGHT_DELIVER_CARGOS = 2,
  // 目标换电航线，若降落点为换电站，此航线飞机落地后，会自动更换电池
  FLIGHT_EXCHANGE_BATTERY = 3,
};
```

关于 Segment：

```
typedef struct Segment {
  Vec3 position;            // 目标位置
  double time_ms;           // 在 time_ms 飞行到目标位置，此时间是相对于起飞时间
  Vec3 v;                   // 到达目标位置时的速度，轨迹专用
  Vec3 a;                   // 到达目标位置时的加速度，轨迹专用
  SegmentType seg_type;     // 此片段是那种类型，起飞、飞行、降落；要求起飞和降落必须垂直
} Segment;
```

4.2 任务信息：TaskInfo

```
typedef struct TaskInfo {
  int32_t task_id;                      // 任务编号
  std::string map_id;                   // 地图编号
  std::vector<DroneInfo> drones;        // 无人机信息集合，包括每架无人机的编号、初始位置、性能限制等
  bool battery_consuming;               // 是否有电池消耗
  bool has_obstacles;                   // 是否有障碍物
  std::vector<Vec3> battery_stations; // 换电站的位置信息
  std::vector<Vec3> landing_positions;// 降落位置
} TaskInfo;
```

4.3 障碍物信息：ObstacleInfo

飞机当前探测到的动态障碍物，探测距离 60 米；
若飞机和动态障碍物距离小于 5 米，会被认为相撞；
相撞后，飞机被标记为 CRASH，炸机类型为 DRONE_CRASH_COLLIDE_DYNAMIC_OBSTACLE(5)
相撞后，障碍物本身不会消失，后续会继续出现（避免对子情况）

```
enum ObstacleType {
  OBSTACLE_DRONE = 1,  // 比较规则的运动，速度和加速度都可能很快
  OBSTACLE_BIRD = 2,   // 不规则运动（转向频繁），速度不会太快，加速度可能较大
  OBSTACLE_KITE = 3,   // 规则运动，速度和加速度都较小
}

typedef struct ObstacleInfo {
  Vec3 position;               // 障碍物位置信息
  Vec3 velocity;               // 障碍物速度信息
  double radius;               // 碰撞半径，现在默认都是 5
  ObstacleType obstacle_type; // 障碍物类型，当前支持飞机、飞鸟、风筝
  std::string id;              // 提供了障碍物 ID，方便跟踪障碍物
} ObstacleInfo;
```

4.4 无人机状态信息：DroneStatus

```
typedef struct DroneStatus {
  std::string drone_id;      // 无人机编号
  uint64_t timestamp;        // 当前时间戳
  Status status;             // 当前状态：等待、起飞中、平飞中、悬停中、降落中、坠机
```

```
  Vec3 position;              // 当前位置
  double height;              // 距地面的高度
  std::vector<int> delivering_cargo_ids;        // 所挂载餐箱编号的集合
  float battery;                                // 当前电量
  std::vector<ObstacleInfo> detected_obstacles;// 当前已知障碍物信息
  DroneCrashType crash_type;                    // 坠机类型
} DroneStatus;
```

关于 status：

```
enum Status {
  // 等待航线
  READY = 0,
  // 起飞中
  TAKING_OFF = 1,
  // 平飞中
  FLYING = 2,
  // 悬停中
  HOVERING = 3,
  // 降落中
  LANDING = 4,
  // 坠机
  CRASH = 5,
  // 收到航线，待飞
  READY_TO_FLY = 6,
};
```

关于 DroneCrashType：

```
enum DroneCrashType {
  DRONE_CRASH_NONE = 0,
  DRONE_CRASH_LOW_BATTERY = 1,
  DRONE_CRASH_COLLIDE_OBSTACLE = 2,
  DRONE_CRASH_COLLIDE_DRONE = 3,
  DRONE_CRASH_LANDED_ON_ILLEGAL_POSITION = 4,
  DRONE_CRASH_COLLIDE_DYNAMIC_OBSTACLE = 5,
};
```

4.5 无人机信息：DroneInfo

```
typedef struct DroneInfo {
  std::string drone_id;
  Vec3 initial_pos;
  DroneLimits drone_limits;
} DroneInfo;
```

关于 DroneLimits：

```
// 无人机的性能限制
typedef struct DroneLimits {
  // 最大速度
  double max_fly_speed_v;
  double max_fly_speed_h;
  // 最大飞行加速度
  double max_fly_acc_v;
  double max_fly_acc_h;
  // 最大载重
  double max_weight;
  // 最大货物数
  double max_cargo_slots;
  // 最小飞行高度，起降阶段不限制
  double min_fly_height;
  // 最大飞行高度
  double max_fly_height;
  // 最大飞行时间
  double max_flight_seconds;
} DroneLimits;
```

4.6 货物信息：CargoInfo

```
typedef struct CargoInfo {
  // 货物 ID
  int id;
  // 货物状态
  CargoStatus status;
  // 位置
  Vec3 position;
  // 重量
  double weight;
  // 价值
  double award;
  // 需要运送的位置
  Vec3 target_position;
  // 距离用户期望送达还剩多少秒，<0 表示已经超时；
  int expected_seconds_left;
  // 距离最晚送达时间还剩多少秒，<0 表示已经超时；这个之后送达会有惩罚（如果时间不够，可以选择放弃此单）
  int latest_seconds_left;
} CargoInfo;
```

5. SDK 接口说明

5.1 PlannerAgent 类

提供登录、启停任务、下发航线等功能

5.1.1 PlannerAgent

```
/**
* 构造函数，传入比赛服务器的 IP 地址和 port 号("ip:port")，地图信息以及日志路径，初始化 PlannerAgent 对象。
*/
explicit PlannerAgent(std::string server_addr, std::shared_ptr<Map> map, std::string log_path = "");
```

5.1.2 Login

```
/**
* 登录比赛服务器，一个用户只能登录一次
*/
Response Login(std::string username, std::string password);
```

5.1.4 GetTaskCount

```
/**
* 获取当前任务数 n。
*/
int GetTaskCount();
```

5.1.5 QueryTask

```
/**
* 查看任务信息
* @param task_index 任务索引[0,n)
* @return 返回任务信息
*/
std::unique_ptr<TaskInfo> QueryTask(int task_index);
```

5.1.6 StartTask

```
/**
* 启动指定的任务
* @param task_index 任务索引[0, n)，一个人同一时间只能启动一个任务
* 断线后，1 分钟内，重新连接可以继续该任务
* 服务端收到请求后，如果当前执行队列已满，会返回失败
* 如果返回成功，服务端会开始创建相关资源。因此从返回成功，到收到第一个 TaskStatus 会有一段时间间隔
*/
Response StartTask(int task_index);
```

5.1.7 StopTask

```
/**
* 主动关闭当前的任务，当前任务被废弃。
*/
void StopTask();
```

5.1.8 ValidateFlightPlan

```
/**
* 校验航线是否合法，仅仅根据无人机能力进行本地验证，即使本地验证通过，不代表下发航线后一定能执行。执行还依赖与当前无人机的状态和位置等信息
* @param drone_limits 无人机限制条件
```

```
* @param flight_plan 无人机航线
* @return 根据返回值判断航线规划是否成功
*/
Response ValidateFlightPlan(const DroneLimits& drone_limits,
                            const FlightPlan& flight_plan);
```

5.1.9 DronePlanFlight

```
/**
* 给指定的无人机下发航线
* @param drone_id 无人机编号
* @param flight_plan 无人机航线
* @return 根据返回值判断航线规划是否成功
*/
Response DronePlanFlight(const std::string& drone_id, const FlightPlan& flight_plan);
```

5.2.0 DroneHover

```
/**
* 通知指定的无人机在当前位置悬停
* @param task_index drone_id 无人机编号
* return 根据返回值判断去人机是否悬停成功
*/
Response DroneHover(const std::string& drone_id);
```

5.2 Map 类

加载地图

5.2.1 CreateMapFromFile

```
static std::unique_ptr<Map> CreateMapFromFile(std::string map_path);
```

5.2.2 Range

```
// 获取地图的范围，单位都是米
virtual void Range(float* min_x, float* max_x, float* min_y, float* max_y, float* min_z, float*
max_z) = 0;
```

5.2.3 Query

```
// 查询给定位置的体素信息
virtual const Voxel* Query(float x, float y, float z) = 0;
```

5.3 Log 类

打印日志

5.3.1 MLog

```
void MLog(const char* filename, int line, const char* format, ...);
```

6. 使用示例

6.1 加载地图示例

```
auto map  = mtuav::Map::CreateMapFromFile("[full path to map file]");
```

6.2 初始化以及使用日志

```
FLAGS_alsologtostderr = true;           //除了日志文件之外是否需要标准输出
FLAGS_colorlogtostderr = true;          //标准输出带颜色
FLAGS_logbufsecs = 0;                   //设置可以缓冲日志的最大秒数，0 指实时输出
FLAGS_max_log_size = 100;               //日志文件大小(单位：MB)
FLAGS_stop_logging_if_full_disk = true; //磁盘满时是否记录到磁盘
google::InitGoogleLogging("uav_champ_example");
google::SetLogDestination(google::GLOG_INFO,
"/home/user_name/work/uav_championship_algorithm_example/log/");
......
LOG(INFO) << "Login successfully";
LOG(INFO) << "QueryTask failed., task index: " << task_idx;
......
```

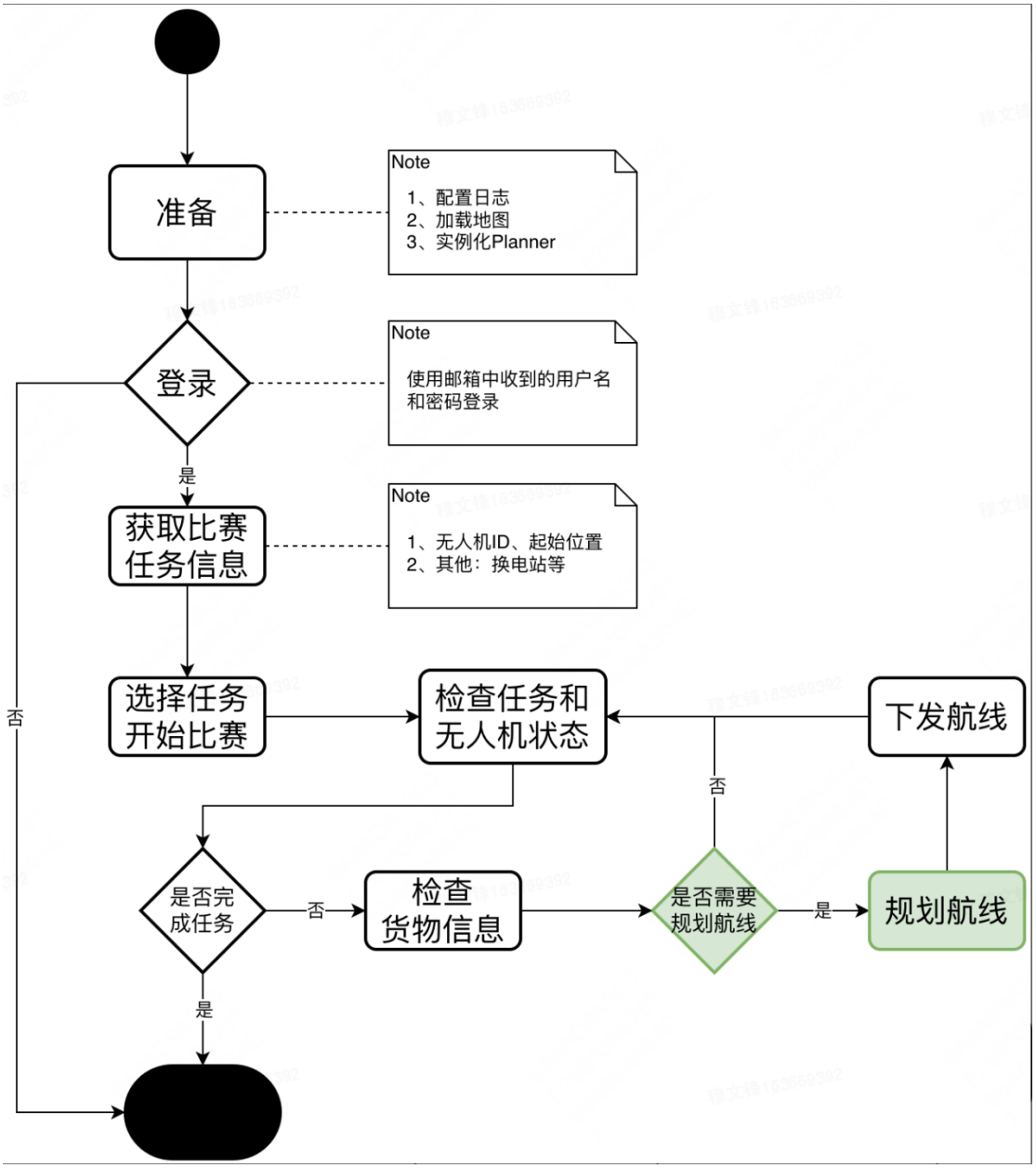
6.3 自定义航线规划器示例

```
#include "mtuav_sdk.h"

class Planner : public mtuav::PlannerAgent {
public:
  Planner() : mtuav::PlannerAgent("[server-address:port]") {}
protected:
  /* 以下纯虚函数需要由开发者自行根据需求实现*/

  // 连接断开时的处理函数
  void OnSdkError(std::string error_msg) override {
    std::cout << error_msg;
  }

  // 任务状态通知函数，每秒执行一次或者无人机、餐箱状态更新时也会触发
  void OnTaskStatus(int task_id, std::vector<mtuav::DroneStatus> status,
                    std::map<int, mtuav::CargoInfo> cargos) override {
    std::cout << "OnTaskStatus, id: " << task_id
              << ", status: " << status.size()
              << ", cargos: " << cargos.size()
              << std::endl;
  }

  // 任务完成或者发生异常时触发次函数调用，但通过 StopTask 主动关闭任务不会调用该函数
  void OnTaskDone(int task_id, bool done, float grade) override {
    std::cout << "OnTaskDone" << std::endl;
  }
```

```
};
```

6.4 业务流程示例

业务流程创建示意图，其中绿色框为需要实现的规划算法逻辑



示例代码：

```
#include <glog/logging.h>
#include <signal.h>
#include <unistd.h>
#include <chrono>
#include <iostream>
#include <thread>
#include "algorihtm.h"
#include "current_game_info.h"
#include "mtuav_sdk.h"
#include "planner.h"
```

```cpp
using namespace mtuav::algorithm;
using namespace mtuav;

// 初始化算法类静态成员变量
int64_t Algorithm::flightplan_num = 0;
bool task_stop = false;

void sigint_handler(int sig) {
    if (sig == SIGINT) {
        // ctrl+c 退出时执行的代码
        std::cout << "ctrl+c pressed!" << std::endl;
        task_stop = true;
    }
}

int main(int argc, const char* argv[]) {
    signal(SIGINT, sigint_handler);
    FLAGS_alsologtostderr = true;   //除了日志文件之外是否需要标准输出
    FLAGS_colorlogtostderr = true;  //标准输出带颜色
    FLAGS_logbufsecs = 0;           //设置可以缓冲日志的最大秒数，0 指实时输出
    FLAGS_max_log_size = 100;       //日志文件大小(单位：MB)
    FLAGS_stop_logging_if_full_disk = true;  //磁盘满时是否记录到磁盘
    google::InitGoogleLogging("uav_champ_example");
    // 配置本地 log 路径
    google::SetLogDestination(google::GLOG_INFO,
                              "[LOG-DIR]");
    // 配置本地路径读取地图信息
    auto map = mtuav::Map::CreateMapFromFile("[SDK-PATH]/map/test_map.bin");
    // 声明一个 planner 指针
    std::shared_ptr<Planner> planner = std::make_shared<Planner>(map);
    // LOG 打印是否成功读取地图
    if (map == nullptr) {
        LOG(INFO) << "Read map failed. ";
        return -1;
    } else {
        LOG(INFO) << "Read map successfully.";
    }
    mtuav::Response r =
        planner->Login("801f0ff5-5359-4c3e-99d4-f05d7eb47423", "e57aab02cf1f7433d7bf385748376164");
    if (r.success == false) {
        LOG(INFO) << "Login failed, msg: " << r.msg;
        return -1;
    } else {
        LOG(INFO) << "Login successfully";
    }

    // std::this_thread::sleep_for(std::chrono::milliseconds(3000));
    int task_num = planner->GetTaskCount();
```

```
    LOG(INFO) << "Task num: " << task_num;
    // TODO 选手指定比赛任务索引
    int task_idx = 3;
    // 获取比赛任务指针
    auto task = planner->QueryTask(task_idx);
    if (task == nullptr) {
        LOG(INFO) << "QueryTask failed., task index: " << task_idx;
        return -1;
    } else {
        LOG(INFO) << "QueryTask successfully, task index: " << task_idx
                  << ", task id: " << task->task_id;
    }

    // 声明比赛动态信息获取类（用于获取无人机实时状态，订单实时状态）
    std::shared_ptr<DynamicGameInfo> dynamic_info = DynamicGameInfo::getDynamicGameInfoPtr();
    // 设置任务结束标识符为 false
    dynamic_info->set_task_stop_flag(false);
    LOG(INFO) << "An instance of class DynamicGameInfo is created. task stop flag: "
              << std::boolalpha << dynamic_info->get_task_stop_flag();

    // TODO 选手需要按照自己的设计，声明算法类
    std::shared_ptr<myAlgorithm> alg = std::make_shared<myAlgorithm>();
    // 将地图指针传入算法实例
    alg->set_map_info(map);
    // 将任务指针传入算法实例
    alg->set_task_info(std::move(task));
    // 将 planner 指针传入算法实例
    alg->set_planner(planner);
    LOG(INFO) << "An instance of contestant's algorihtm class is created. ";

    // 启动对应的比赛任务
    auto r2 = planner->StartTask(task_idx);
    if (r2.success == false) {
        LOG(INFO) << "Start task failed, msg: " << r2.msg;
        return -1;
    } else {
        LOG(INFO) << "Start task successfully, task index: " << task_idx;
    }
    while (!dynamic_info->get_task_stop_flag()) {
        if (task_stop == true) {
            planner->StopTask();
            LOG(INFO) << " Stop task by ctrl+c ";
            break;
        }

        LOG(INFO) << "Soving the problem using the the algorithm designed by contestants. ";
        // 调用算法类求解前，先更获取最新的动态信息
        alg->update_dynamic_info();
        LOG(INFO) << "The latest dynamic info has been fetched. ";
```

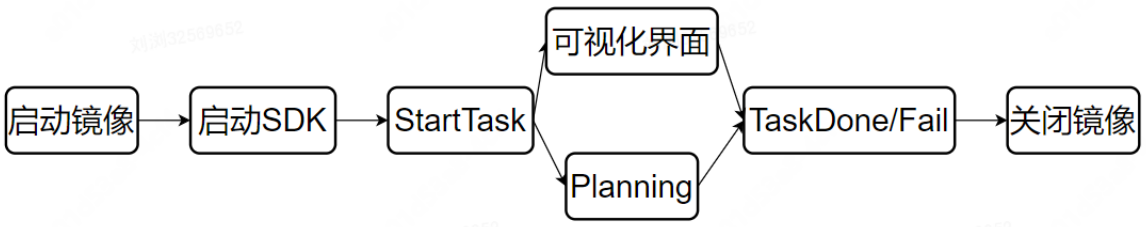
```
        // 调用算法求解函数，solve 函数内内部输出飞行计划,返回值为下次调用算法求解间隔（毫秒）
        int64_t sleep_time_ms = alg->solve();
        LOG(INFO) << "Algorithm calculation completed, the next call interval is " << sleep_time_ms
                  << " ms.";
        // 选手可自行控制算法的调用间隔
        std::this_thread::sleep_for(std::chrono::milliseconds(sleep_time_ms));
    }

    sleep(1);
    planner->StopTask();
    google::ShutdownGoogleLogging();
    return 0;
}
```

* 附：调用 StartTask 需要注意，一个用户只允许启动一个任务，即调用一次 StartTask 方法。

```
// 启动对应的比赛任务
auto r2 = planner->StartTask(task_idx);
if (r2.success == false) {
    LOG(INFO) << "Start task failed, msg: " << r2.msg;
    return -1;
} else {
    LOG(INFO) << "Start task successfully, task index: " << task_idx;
}
```

6.5 比赛平台单机调试版

比赛系统提供单机版，供参赛者参考、学习、调试使用。单机版以 docker 镜像提供，此镜像使用流程如下：



➢ 获取方式如下：

```
# 此镜像只能在 linux 系统中使用，提前安装好 docker
# 拉取镜像，可以不定时拉取，若有更新我们会在群里通知
docker pull marcobright2023/mtuav-competition:standalone-final
# 启动单机版系统，内置了一个测试任务，注意：每次启动只能支持一个人执行一次比赛任务的执行
# 其中<host dir>电脑本地目录，用来保存单机版产生的日志
# 启动后，127.0.0.1:50051 是单机版的接入地址，把这个地址传入 PlannerAgent 的构造函数
# 单机版默认使用测试地图，路径：[sdk_dir]/map/test_map.bin
# SDK 接入且启动任务（StartTask）后，
# 在浏览器输入 http://127.0.0.1:8888，可以打开一个可视化窗口，这里可以观察飞机和航线的状态
```

```
# 注意：此可视化比较费计算资源，建议使用另外的电脑观看，地址需要更新为 http://[docker-ip]:8888
docker run -id -p 8888:8888 -p 50051:50051 -v <host dir>:/mt-log marcobright2023/mtuav-competition:standalone-final start
```

```
# 注意不要把 libmtuav_sdk.go.so 和 libmtuav_sdk.so 放入系统目录
# 通过`ldd [你的可执行程序]`可以查看具体链接的 so，如果是/lib/、/lib64/等系统目录请删除复制进去的 so
# 通过下面命令，指定链接路径，需要使用绝对路径
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/abs_path_to[sdk/libs]/   # 导出链接地址，注意使用绝对路径

# run planner agent with 127.0.0.1:50051
# ./planner_agent
```

- 单机版可视化监控使用说明如下
  - **启动网址：**http://127.0.0.1:8888

    在启动竞赛 docker 之后，使用 SDK 接入，StartTask 成功后，本机可以访问此网址
    如果其他机器，使用 http://[ubuntu-ip]:8888，其他 ubuntu-ip 为启动此镜像的主机 IP
    注意：SDK 再 StartTask 后不要退出，退出后服务会关闭任务，同时也会关闭可视化程序。
  - **打开地图**

    界面左侧分为地图和无人机
    地图界面：

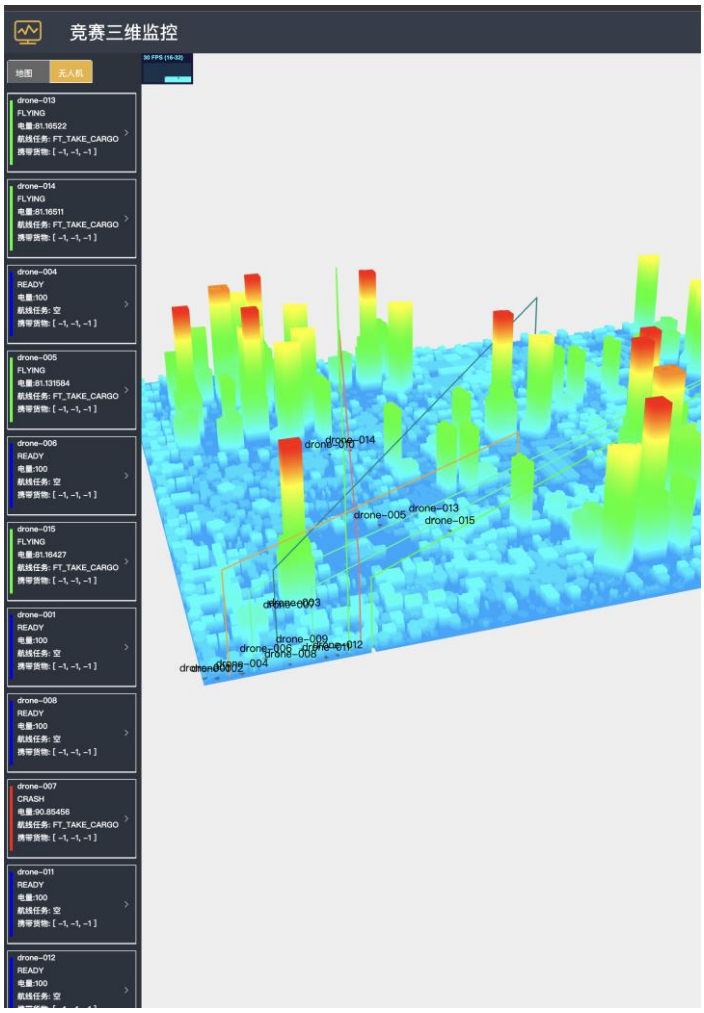
    点击打开文件，一定要选择全部地图文件，然后打开。系统会读取数据文件并显示；
    如果电脑性能不好，显示地图可能会造成卡顿。

  - **无人机：**

显示无人机的基本信息：名称、状态、任务类型、电量、携带货物、坐标信息。
点击某个无人机，右面的三维界面则会跳到该无人机的位置。

- **三维显示界面：**

显示三维地图
显示仿真机和航线信息。
三维操作：左键：平移；右键：旋转



6.6 连接在线比赛系统

在线比赛系统地址：sim.race.meituan.com:8090，使用方法如下（planner.h）：

```
explicit Planner(std::shared_ptr<mtuav::Map> map)
    : mtuav::PlannerAgent("sim.race.meituan.com:8090", std::move(map)) {}
```

登录权限：用户名和密码会通过邮件发送给参赛者（注意保护好用户名和密码，若需要修改请联系赛事组织人员），启动 SDK 后，调用 mtuav::PlannerAgent::Login 进行登录，检测 Response 是否登录成功：

```
// 登录比赛服务器
Response Login(std::string username, std::string password);
```

登录限制：每个用户名只能登录一次，后续登录会失败（若暂时不跑任务，尽量不要连接在线系统）

系统限制：最多支持同时支持 25 个比赛任务，若系统容量已满，启动任务会返回失败。(若长时间无法启动，请联系赛事组织人员安排扩容)

任务限制：

测试阶段，每个任务限时 1 小时，从启动任务开始计时，到 1 小时自动停止并计算当前得分

注意事项：

不同于单机镜像，在线系统需要使用[SDK-PATH]/map/competition_map.bin（地图尺寸更大）

同时，在线比赛系统没有可视化窗口，飞机状态只能通过代码来判断

规划时，要给飞机电量留余量，极限规划可能引发坠机（电量<1%时随机坠机）

建议：

先使用单机镜像调试代码（地图和场景相对简单），然后连接在线系统进行测试

强大的算力对算法会有些帮助，但不是决定性的。充分利用已知信息，提前最好预计算；充分利用多核计算能力；任何时候都优先考虑飞行安全。

7. 常见问题解答

7.1 SDK 只有 C++版本的吗？

是的，由于无人机规划是一个时效性高且算力需求高的系统，因此希望大家能用 C++来实现算法。

同样，比赛的目的是希望参赛者能够实现一套云端的无人机规划调度系统，规划算法是比赛的主要内容，而系统能力也是比赛内容之一。

7.2 比赛需要远程接入比赛系统，如果网络延时很大怎么办？

先使用单机版调试好算法实现，然后尽量在网络情况好的时候接入比赛系统。

国外的参赛同学，可以考虑注册国内腾讯云、阿里云的服务器。

当然，如有问题，也可以联系我们。

7.3 飞机可以降落/停靠的位置有哪些？

飞机的起始位置、货物出现的位置、任务提供的换电站、飞机停留点

7.4 一次下发所有航线，还是得分段下发航线

为了模拟真实运营，每次只能下发一架飞机的一条航线。多架飞机的航线下发需要调用多次下发函数。一架飞机的后续航线，需要根据飞机状态确认下发时机。

7.5 如何知道在哪里取餐，以及送往何处？

需要开发者从 CargoInfo 中获取餐箱当前位置以及需要送达的地点。具体来说就是会从任务状态通知函数 OnTaskStatus 接收到通知。

7.6 下发航线后，判断航线是否有效的标准

系统收到航线后，会根据当前的限制条件对航线做预先验证，验证失败则不会执行航线。若指定飞机当前为坠毁状态，是无效规划；若当前飞机正在航线中，或处于悬停状态，则会跳过新航线的起飞阶段，执行后续航线。

7.7 如何换电？

下发一条 FlightPurpose 是 FLIGHT_EXCHANGE_BATTERY 的航线，且终点必须停在换电站，否则无法换电。注意，如果在换电站上停留的时间超过固定时长（60 秒），那么惩罚也会随着超过的时长而加大。

7.8 装载货物与卸载货物的条件

装载货物：航线目的为 FLIGHT_TAKE_CARGOS，航线终点必须停止在货物起始位置，否则无法装货。

卸载货物：航线目的为 FLIGHT_DELIVER_CARGOS，航线终点必须停止在地面，降落完成后自动卸载货物；卸载货物不会判断货物的目的地是否正确。

7.9 如何判断无人机相撞？

本次比赛统一定义无人机的安全间距为 10 米，如果两架无人机最短距离小于 10 米，会被判定为相撞，飞机状态转换为 CRASHED。注意：降落到距离很近的位置也会判定为相撞。

7.10 创意赛的仿真接口是怎样的？

由于初赛阶段不开发创意赛的接口，初赛完成后后，我们会更新 SDK，给创意赛的参赛者单独发送。

创意赛的方式主要为：参赛者提供飞机位置、换电站位置、停留点位置等，也可以指定货物出现位置和概率，进而完成航线规划。同时需要根据收集到的信息完成数据统计，为自己的创意补充数据佐证。

7.11 系统时间需要同步吗？

开始比赛前检查下主机的时区设置，要求为 Beijing 时间（UTC+8）；同时做一次时间同步 (sudo ntpdate -b -p 5 -u cn.ntp.org.cn)

8. 联系方式与支持

邮箱：liuyuhan05@meituan.com

技术问题也可在比赛微信群中交流。