软件开发技术指南

软件系统综述

NX 登录账号与密码

默认管理员账户：nx

默认管理员账户密码：jetson

关键环境版本

JetPack 版本：5.1.2（Ubuntu 版本：20.04，CUDA 版本：11.4.315）

Python 版本：3.8

OpenCV 版本：4.9.0（CUDA 加速，含 opencv_contrib）

PyTorch 版本：2.1.0（TorchVision 版本：0.16.0）

ROS 版本：ROS Noetic

YoloV5 版本：6.2

远程连接方法

- SSH
  a. 首先 NX 连接显示器，启动，连接 WiFi。
  b. 打开一个控制台，执行指令 `ifconfig`，找到 `wlan0` 网卡的 IP 地址，通常是 192.168.x.xxx 格式的。
  c. 控制 NX 的设备与 NX 连接同一个 WiFi，打开一个控制台，执行 `ssh nx@192.168.x.xxx`。
  d. 首次登录会询问是否保存 NX 的密钥，输入 `yes` 并回车，然后输入 nx 的密码 `jetson`。
  e. SSH 连接成功后，可以合理使用 tmux 来将控制台窗口扩展为多个。

TL;DR 示例代码快速运行指南

准备工作

刷好系统之后，执行下面几步操作：

1. 命令行运行 `sudo usermod -aG dialout $USER`（永久解决/dev/ttyACM0 没权限的问题）
2. 命令行进入 ~/catkin_ws 目录，运行 `catkin_make`（编译工作空间）

运行示例程序

- 定点飞行示例(tutorial_basic 中的 `run_vel_control.launch`)，本示例一定要 ssh 远程操作
    a. 把 NX 调整到电源供电状态，飞机装上桨叶，该连接的线连接好。
    b. NX 连上 Wi-Fi，命令行运行 `ip addr` 查看本机 ip
    c. 把飞机放在空旷位置，长按 GPS 上的 `SWITCH` 按钮解锁。
    d. 电脑连上同一个 Wi-Fi，命令行上依次运行下列

```
Plain Text
ssh nx@NX 的 ip
source ～/catkin_ws/devel/setup.bash
roslaunch tutorial_basic run_vel_control.launch
```

    e. 飞机会以慢慢起飞，朝给定点(默认是正前方一米，高一米处)飞行，靠近定点后悬停，悬停 `3` 秒后慢慢降落

- 投放装置舵机控制示例(tutorial_catapult 中的 `catapult_driver`)，本示例可以 ssh 操作或者 NX 连接鼠标键盘屏幕操作，在 ssh 和非 ssh 上所输命令相同。如果是使用 px4 控制舵机，可能需要 off board 和 arm 之后才能发出控制信号。
    a. 启动一个命令行或 ssh 到 NX 上，运行下列命令

```
Plain Text
source ～/catkin_ws/devel/setup.bash
roslaunch tutorial_catapult catapult_driver.launch
```

    b. 启动另一个命令行，运行下列命令

```
Plain Text
#打开前仓
rostopic pub /servo/front std_msgs/Bool "data: true"
#关闭前仓
```

```
rostopic pub /servo/front std_msgs/Bool "data: false"
#打开后仓 1
rostopic pub /servo/back_1 std_msgs/Bool "data: true"
#关闭后仓 1
rostopic pub /servo/back_1 std_msgs/Bool "data: false"
#打开后仓 2
rostopic pub /servo/back_2 std_msgs/Bool "data: true"
#关闭后仓 2
rostopic pub /servo/back_2 std_msgs/Bool "data: false"
```

- 下视相机图像获取示例(tutorial_vision 中的 `simple_camera_driver.launch`)，需要连接屏幕

a. 启动一个命令行，运行下列命令

```
Plain Text
source ～/catkin_ws/devel/setup.bash
roslaunch tutorial_vision simple_camera_driver.launch
```

b. 启动另一个命令行，运行 `rqt_image_view image:=/camera/image_raw`，该命令会启动一个窗口，如果没有图像，在左上角下拉框换一下选项

示例代码详解

综述

示例代码位于`~/catkin_ws/src/tutorials` 目录下，由以下功能包构成：

- `tutorial_basic`：无人机起飞、降落和定点飞行三个最基本的功能。
- `tutorial_navigation`：无人机定位、导航前置节点和示例节点。
- `tutorial_vision`：摄像机驱动和视觉相关节点。
- `tutorial_catapult`：投放装置相关。
- `tutorial_all`：完整比赛任务实现（不包含钻圈）。

在以上功能包中，均有名为 `launch` 的文件夹，其内部为一些封装了一些节点的启动文件，要使用这些 `launch` 文件，只需要编译并 `source ~/catkin_ws/devel/setup.bash` 后，使用以下指令即可调用。

```
Plain Text
```

```
roslaunch <功能包名> <launch 文件名>.launch
```

要使无人机完成自主飞行任务，至少应当启动：

- mavros：这是用于上位机与飞控通信的节点，使用 `roslaunch tutorial_basic mavros.launch` 可启动，请注意：该步骤可能会报错提示没有操作“/dev/ttyACM0”的权限（见执行问题(2)），使用 `sudo chmod 777 /dev/ttyACM0` 指令可以解决该问题。
- MID360 雷达驱动：得到用于无人机定位的点云信息，若无人机定位失败时尝试自主飞行，GPS 会发出警报并拒绝解锁。使用 `roslaunch livox_ros_driver2 msg_MID360.launch` 指令可以启动雷达驱动。
- 定位算法：用于以 MID360 得到的点云信息进行定位，系统中自带 FAST_LIO 定位算法，使用 `roslaunch fast_lio mapping_mid360.launch rviz:=false` 指令可以启动雷达驱动。MID360 雷达驱动和 FAST_LIO 定位算法可以使用 `roslaunch tutorial_navigation mid360.launch` 一次性全部启动。
- 策略节点：对无人机飞行速度进行控制的任务节点，所有示例功能包中以 `run_` 开头的 launch 文件都会启动 mavros、MID360 雷达驱动、定位算法和特定任务的策略节点。

以下分别介绍每个功能包内的 launch 文件、节点和节点的实现。

tutorial_basic

launch 文件

- mavros.launch: 启动 mavros，如果提示 Permission Denied，需要使用 `sudo chmod 777 /dev/ttyACM0` 指令。
- run_takeoff.launch: 启动起飞/降落节点，节点说明见后文。此节点为了便于读者理解，在飞机悬停过程中没有使用 PID 固定位置，因此不太适合实物飞行，更适合仿真执行。
    - 参数：
        - run_mavros: 是否同时启动 mavros，默认为 `true`。
        - run_fast_lio: 是否同时启动 MID360 驱动和 FAST_LIO 定位算法，默认为 `true`。
- run_vel_control.launch: 启动定点飞行节点，节点说明见后文。
    - 参数：

- hover_x: 目标悬停点 x 坐标，默认为 `1`。
- hover_y: 目标悬停点 y 坐标，默认为 `0`。
- hover_z: 目标悬停点 z 坐标，默认为 `1`。
- run_mavros: 是否同时启动 mavros，默认为 `true`。
- run_fast_lio: 是否同时启动 MID360 驱动和 FAST_LIO 定位算法，默认为 `true`。

节点

- takeoff_node: 以`(0.0, 0.0, 0.4)`速度起飞，当海拔高于`1.0`时悬停，悬停过程中始终采用`(0.0, 0.0, 0.0)`的速度，悬停`3`秒后以`(0.0, 0.0, -0.2)`的速度降落，当海拔低于`0.1`时切换到`AUTO.LAND`模式（电机停转）。**请注意如果飞机重心不居中或飞控参数不准，长时间采用`(0.0, 0.0, 0.0)`的速度飞行可能会使飞机向一个方向偏，可能会有安全风险，此节点是为了便于用户阅读源码，学习通过 ROS 控制无人机行为而编写的，不适用于实物飞行。**
  - 代码解析:
    - 第一部分（从主函数开始，19~25 行）：

```C++
ros::init(argc, argv, "takeoff_node");
ros::NodeHandle nh;
ros::Subscriber state_sub =
nh.subscribe<mavros_msgs::State>("mavros/state", 1,
state_cb);
ros::Subscriber local_pose_sub =
nh.subscribe<geometry_msgs::PoseStamped>("mavros/local_
position/pose", 1, pose_cb);
ros::Publisher vel_pub =
nh.advertise<geometry_msgs::TwistStamped>("mavros/setpo
int_velocity/cmd_vel", 1);
ros::ServiceClient arming_client =
nh.serviceClient<mavros_msgs::CommandBool>("mavros/cmd/
arming");
ros::ServiceClient set_mode_client =
nh.serviceClient<mavros_msgs::SetMode>("mavros/set_mode
");
```

`ros::init`函数用于初始化一个 ROS 节点，只有调用了该函数以后，程序才可以使用 ROS 的各种 API，其中第三个参数`"takeoff_node"`是节点的名称，ROS 要求整个分布式系统中不能出现两个名称一样的节

点，节点在被 launch 文件引用时，节点名称会被 `node` 标签的 `name` 属性的值重写。

`ros::NodeHandle` 类是 C++版本的 ROS 用于操作节点的句柄，订阅、发布话题，创建、调用服务都需要使用该类的实例。

`ros::NodeHandle::subscribe` 方法用于订阅一个话题，该方法需要一个模板参数 `T`（代码中<>内的东西，在本段代码中分别为 `mavros_msgs::State` 和 `geometry_msgs::PoseStamped`），该模板参数用于指定话题内的消息类型。该方法的第一个参数就是话题的路径，在控制台中使用 `rostopic list` 指令可以看到当前 ROS 系统中都有哪些话题至少被一个节点订阅或发布。

`ros::NodeHandle::subscribe` 方法的第三个参数是一个函数，该函数称为**回调函数**。回调函数必须有且仅有一个 `const T::ConstPtr &` 类型的形参且无返回值，其中 `T` 是刚刚模板参数中给出的消息类型，`T` 的成员 `ConstPtr` 指的是 `T` 的常量指针，它在定义消息类型时就已经由 catkin 自动生成，`const &`是 C++的常引用语法。在 C++版的 ROS 中，某一节点订阅某一话题，且该话题中出现了新的消息时，只要发生了以下两种情况之一，回调函数就会被调用：

1. `ros::spinOnce` 函数被调用。该函数被调用时，节点消息队列中的所有消息都会作为实参传入回调函数中。
2. `ros::spin` 函数被调用。节点调用该函数后，线程会被阻塞，直到节点被 Ctrl+C 终止之前，后续的程序都不会继续进行，其功能类似于

```C++
while (ros::ok()) ros::spinOnce();
```

在本段代码中，订阅的两个话题分别是 `mavros/state` 和 `mavros/local_position/pose`。前者由 mavros 发布，其内容为 mavros 与飞控的连接情况、飞控的飞行模式、是否解锁等；后者也由 mavros 发布，其内容为飞机的实时定位信息。两个订阅的回调函数如下（8~16 行）：

```C++
mavros_msgs::State current_state;
void state_cb(const mavros_msgs::State::ConstPtr &msg)
{
    current_state = *msg;
}

geometry_msgs::PoseStamped current_pose;
```

```
void pose_cb(const geometry_msgs::PoseStamped::ConstPtr
&msg) {
    current_pose = *msg;
}
```

其功能均为将消息内容直接保存在全局变量中。

以下写法是不能正常工作的：

```
C++
mavros_msgs::State *current_state_ptr;
void state_cb(const mavros_msgs::State::ConstPtr &msg)
{
    current_state_ptr = msg;
}
```

这是因为一旦消息处理完毕后，消息就会被回收掉，其指针就会成为野指针，不再能用于访问消息的内容。在回调函数中保存消息内容必须采取复制消息的方法，但同时应当注意，占用内存较大的消息（如`sensor_msgs/Image`）的复制过程可能会相对耗时，产生性能问题，此类消息应当考虑在回调函数内就进行处理。

细心的读者可能会思考一个问题，如果在两次`spinOnce`函数调用期间，订阅的话题中出现了两条新的消息，那么在下一次`spinOnce`调用时，回调函数会被调用一次还是两次？这就与`subscribe`方法的第二个参数有关，该参数指定了节点在订阅此消息时，预留一个多大的缓存区用于保存暂未被处理的消息，若在两次处理之间，该话题内的新消息数量超过了缓存区的大小，缓存区中最旧的消息就会被丢弃，替换为本条最新的消息。在本段代码中，暂存队列大小均为 1，这意味着如果两次`spinOnce`内有两条新消息，那么更旧的那条消息会被丢弃，最终每次`spinOnce`时，两个回调函数都只会被调用一次。

在示例代码的后续部分中，`subscribe`方法的返回值并未被用到，那是否意味着其返回值并不需要使用一个`ros::Subscriber`类型的变量存储起来呢，在 Python 版本的 ROS 中是可以的，但在 C++版本的 ROS 中是不行的。这是因为如果废弃`subscribe`方法的返回值，该返回值会在函数返回后立即被析构，而`Subscriber`类型在析构时，会取消对话题的订阅，就无法达到订阅话题的目的了。

`ros::NodeHandle::advertise`方法用于创建一个发布某话题的句柄，与`subscribe`方法一样，该方法也需要一个模板参数，话题路径和队列大小。后续可以使用`ros::Publisher`类的`publish`方法向话题中发送新消息。本段代码向`mavros/setpoint_velocity/cmd_vel`话题

发送新消息，该话题由 mavros 订阅，用于控制无人机飞行的速度，Pixhawk 系列飞控要求，为了保证上位机和飞控的通信正常，确保飞行安全，该话题每秒必须被发布超过 20 条新消息。

`ros::NodeHandle::serviceClient` 方法用于创建一个调用某服务的句柄，在 ROS 中，服务是一种阻塞式通讯方法，当客户端节点调用一个服务端节点的服务时，客户端节点会被阻塞，等待服务端处理并回复一个调用结果。该方法需要一个服务类型模板参数和一个服务路径，在控制台中使用 `rosservice list` 可以看到当前 ROS 系统中都有哪些服务。

- 第二部分（28~32 行）：

```C++
ros::Rate rate(20.0);
while (ros::ok() && !current_state.connected) {
    ros::spinOnce();
    rate.sleep();
}
```

本段代码首先声明了一个 `ros::Rate` 变量，它是用于控制无限循环代码的执行频率的，其功能类似于 `ros::Duration`，`20.0` 是指若将 `rate.sleep();`放在一个死循环中，该循环每秒会执行 20 次。

接下来是一个 while 循环，循环条件是 `ros::ok()` 且`!current_state.connected`，`ros::ok()`的返回结果表示该节点是否已经被用户使用 Ctrl+C 关闭，正常情况下该函数的返回结果都是 `true`，在 ROS 程序中，使用 `ros::ok()`替换死循环中的 `true` 条件，是一种被建议的做法。`current_state` 是 `mavros/state` 话题内的最新消息，其 `connected` 成员表示 mavros 是否成功与飞控建立连接。因此本段代码的含义是“等待 mavros 与飞控的连接”，需要注意函数体内必须调用 `spinOnce` 函数，否则即使 `mavros/state` 话题内有新消息，回调函数也不会执行，全局变量 `current_state` 将永远不会更新，而 `rate.sleep();`语句是可以删除的。

- 第三部分（34~111 行）外层

```C++
int fsm_state = 0;
ros::Time last_srv_request = ros::Time::now();
while (ros::ok()) {
    geometry_msgs::TwistStamped twistStamped;
    switch (fsm_state) {
```

```
        ...
    }
    vel_pub.publish(twistStamped);
    ros::spinOnce();
    rate.sleep();
}
```

本段代码的主体部分是一个 `while` 循环嵌套一个 `switch` 语句，这种设计模式称为**状态机模型**，在这种设计模式中，无人机时刻处在某种“状态”下，这里的状态可以理解为完成特定任务的某个阶段，如起飞、悬停、降落都可以理解为一个阶段，在这一阶段下，该状态的代码会被反复执行。当无人机处在某状态下，且满足特定的条件时，无人机会转换其状态到另一个状态，以上代码中 `fsm_state` 是用来存储无人机当前状态的整数，在 switch 语句内修改该整数，无人机就会在下一次主循环中切换到另一个状态。

`last_srv_request` 是一个用来计时的变量，在“悬停 x 秒”等策略中有所应用。另外注意到，在 while 循环内，switch 语句外，还声明并发布了一条控制无人机速度的消息，如此设计是因为 Pixhawk 系列飞控要求每秒至少发 20 次速度控制消息，这意味着采用 20.0 的 Rate，每次主循环都必须发布一个速度消息，哪怕该速度消息要求无人机的全方向速度均为 0。在 switch 语句中通过修改该 `geometry_msgs::TwistStamped` 对象的成员，可以控制无人机飞行的速度。

示例代码中的所有策略节点均采用状态机模型设计，故之后的代码详解只对策略的各状态进行详解，不再解释状态机模型本身。

- 切换 OFFBOARD 状态（fsm_state = 0）

```C++
if (current_state.mode == "OFFBOARD") {
    fsm_state = 1;  // goto before armed state
} else {
    if (ros::Time::now() - last_srv_request >
ros::Duration(1.0)) {
        mavros_msgs::SetMode offb_set_mode;
        offb_set_mode.request.custom_mode = "OFFBOARD";
        if (set_mode_client.call(offb_set_mode) &&
offb_set_mode.response.mode_sent) {
            ROS_INFO("Offboard enabled");
        } else {
            ROS_WARN("Failed to enable offboard");
        }
```

```
        last_srv_request = ros::Time::now();
    }
}
```

该状态的主体结构为一个“if-else”，其中 if 语句的条件满足时，状态机切换到状态 1（解锁状态），该条件称为**状态切换条件**，如果 if 语句的条件不满足，则 else 内的代码将被执行，它表示的是状态机处在该状态时，无人机的行为。

状态切换条件：若飞行模式成功切换到 OFFBOARD，切换到解锁状态（源 40~42，此处 1~3 行）。

状态行为：每`1.0`秒（源 43、51，此处 4、12 行）调用一次`set_mode_client`服务以切换到 OFFBOARD 飞行状态（源 44~46，此处 5~7 行），若调用成功，输出`"Offboard enabled"`（源 47，此处 8 行），否则输出`"Failed to enable offboard"`（源 49，此处 10 行）。

如果策略节点在此状态下一直输出`"Offboard enabled"`或一直输出`"Failed to enable offboard"`，无法跳转到解锁状态，请检查：

1. mavros 成功与飞控进行连接，使用`rostopic echo /mavros/state`指令检查。
2. 飞控参数正确。

- 解锁状态（fsm_state = 1）

```
C++
if (current_state.armed) {
    fsm_state = 2;  // goto takeoff state
} else {
    if (ros::Time::now() - last_srv_request >
ros::Duration(1.0)) {
        mavros_msgs::CommandBool arm_cmd;
        arm_cmd.request.value = true;
        if (arming_client.call(arm_cmd) &&
arm_cmd.response.success) {
            ROS_INFO("Vehicle armed");
        } else {
            ROS_WARN("Failed to arm vehicle");
        }
        last_srv_request = ros::Time::now();
    }
}
```

状态切换条件：若无人机成功解锁，切换到起飞状态（源 56~58 此处 1~3 行）。

状态行为：每 `1.0` 秒（源 59、67，此处 4、12 行）调用一次 `arming_client` 服务（源 60~62，此处 5~7 行），若调用成功，输出 `"Vehicle armed"`（源 63，此处 8 行），否则输出`"Failed to arm vehicle"`（源 65，此处 10 行）。

如果策略节点在此状态下一直输出`"Vehicle armed"`或一直输出 `"Failed to arm vehicle"`，无法跳转到起飞状态，请检查：

1. 飞控已进行校准。
2. 定位信息被正确输入到 mavros，通过检查 `rostopic echo /mavros/vision_pose/pose` 指令是否有输出来判断定位是否工作正常。
3. 飞控其它参数配置正确，重点检查与定位有关的参数是否设置为“使用视觉定位”，GPS 安全检查是否已经关闭。

- 起飞状态（fsm_state = 2）

```C++
if (current_pose.pose.position.z > 1.0) {
    fsm_state = 3;  // goto hover state
    last_srv_request = ros::Time::now();
} else {
    twistStamped.twist.linear.x = 0.0;
    twistStamped.twist.linear.y = 0.0;
    twistStamped.twist.linear.z = 0.4;
}
```

状态转换条件：若无人机海拔大于 `1.0`，切换到悬停状态（源 72~75，此处 1~4 行）。重置 `last_srv_request` 的值是因为下一状态需要计算时间。

状态行为：设置无人机的飞行速度为`(0.0, 0.0, 0.4)`（源 76~78 行，此处 5~7 行）。在 ROS 中，消息类型均为结构体，声明时其成员均会按照默认值（int 为 0，double 为 0.0，bool 为 false 等等）初始化，因此此处 `twistStamped.twist.linear.x` 和 `twistStamped.twist.linear.y` 不做赋值操作也是可以的。需要注意的是，实际调试无人机飞行时，初始的起飞速度不宜过快，宜采用能使无人机起飞的最小速度，以防止因为各方面配置疏忽造成的安全问题，在确定无人机能够按预期工作时，再逐渐加快无人机起飞速度。

起飞过程中无人机可能因为中心不居中等问题向一侧轻微偏移，如果需

要水平位置更加稳定的起飞，可以参考 tutorial_basic/vel_control_node 部分的 PID 控制方法和 tutorial_vision 部分关于霍夫圆检测的方法，实现起飞过程中对准起降平台的功能

- 悬停状态（fsm_state = 3）

```
C++
if (ros::Time::now() - last_srv_request >
ros::Duration(3.0)) {
    fsm_state = 4;  // goto land state
} else {
    twistStamped.twist.linear.x = 0.0;
    twistStamped.twist.linear.y = 0.0;
    twistStamped.twist.linear.z = 0.0;
}
```

状态转换条件：若状态机已进入悬停状态超过 `3.0` 秒，切换到降落状态（源 74、82~84，此处起飞状态第 3，悬停状态第 1~3 行）。

状态行为：设置无人机的飞行速度为`(0.0, 0.0, 0.0)`（源 85~87 行，此处 4~6 行）。

- 降落状态（fsm_state = 4）

```
C++
if (current_state.mode == "AUTO.LAND") {
    fsm_state = -1;  // goto do nothing state
} else if (current_pose.pose.position.z < 0.1) {
    if (ros::Time::now() - last_srv_request >
ros::Duration(0.5)) {
        mavros_msgs::SetMode land_set_mode;
        land_set_mode.request.custom_mode =
"AUTO.LAND";
        set_mode_client.call(land_set_mode);
        last_srv_request = ros::Time::now();
    }
} else {
    twistStamped.twist.linear.z = -0.2;
}
```

状态转换条件：若飞行状态已切换到 AUTO.LAND（源 91，此处 1 行），切换到空状态（源 92，此处 2 行）。空状态就是不做任何事，等待用户手动关闭节点的状态。

状态行为：若无人机的海拔小于 `0.1`（源 93，此处 3 行），每 `0.5` 秒

（源 94，此处 4 行）调用 `set_mode_client` 服务尝试切换到 AUTO.LAND 飞行状态（源 95~98，此处 5~8 行）。否则设置无人机的飞行速度为`(0.0, 0.0, -0.2)`（源 101 行，此处 11 行）。

需要注意，无人机初次降落的速度也不宜过快。

- vel_control_node: 以`(0.0, 0.0, 0.4)`速度起飞，当海拔高于`1.0`时朝给定点飞行，当距离给定点距离小于`0.3`时悬停，悬停过程中采用 PID 控制的方法使无人机被精确固定到目标点上，悬停`3`秒后以`(0.0, 0.0, -0.2)`的速度降落，当海拔低于`0.1`时切换到`AUTO.LAND`模式。
  - 参数:
    - hover_x: 悬停目标点 x 坐标，默认为`0.0`。
    - hover_y: 悬停目标点 y 坐标，默认为`0.0`。
    - hover_z: 悬停目标点 z 坐标，默认为`1.0`。
  - 代码解析:
    - 计算两点间距离的工具函数（13~22 行）

```C++
double getLengthBetweenPoints(geometry_msgs::Point a,
geometry_msgs::Point b,
                              double *out_err_x =
nullptr, double *out_err_y = nullptr, double *out_err_z
= nullptr) {
    double err_x = a.x - b.x;
    double err_y = a.y - b.y;
    double err_z = a.z - b.z;
    if (out_err_x != nullptr) *out_err_x = err_x;
    if (out_err_y != nullptr) *out_err_y = err_y;
    if (out_err_z != nullptr) *out_err_z = err_z;
    return sqrt(err_x * err_x + err_y * err_y + err_z *
err_z);
}
```

该函数传入两个`geometry_msgs::Point`类型的点 A 和 B，计算 A 和 B 每个分量的差值（源 15~17，此处 3~5 行），若函数调用者需要这些分量差值（表现为`out_err_x`、`out_err_y`和`out_err_z`参数不为`nullptr`），将计算结果通过传入指针的方式输出，最后函数返回使用勾股定理计算出来的两点距离（源 21，此处 9 行）。

    - 节点初始化，话题订阅，创建话题发布句柄和服务调用句柄，等待 FCU 连接部分

见 `tutorial_basic/takeoff_node` 代码解析第一部分，此处只修改话题 `mavros/local_position/pose` 的回调函数。

```
C++
geometry_msgs::PoseStamped current_pose;
geometry_msgs::Vector3 current_rpy;
void pose_cb(const geometry_msgs::PoseStamped::ConstPtr
&msg) {
    current_pose = *msg;
    tf::Quaternion quaternion;
    tf::quaternionMsgToTF(msg->pose.orientation,
quaternion);
    tf::Matrix3x3(quaternion).getRPY(current_rpy.x,
current_rpy.y, current_rpy.z);
}
```

以上修改版本的回调函数添加了“缓存无人机实时欧拉角”的功能，欧拉角与四元数一样，都表示无人机的旋转信息。在收到定位信息后，回调函数先使用 `tf::quaternionMsgToTF` 函数将 `geometry_msgs::Quaternion` 类型的四元数转换为 `tf::Quaternion` 类型的四元数，在这一过程中，数据实质上没有发生变化。接着使用 `tf::Matrix3x3` 函数将四元数转为了旋转矩阵，关于旋转矩阵的概念，读者可以自行学习计算机图形学相关知识了解。最后再使用 `getRPY` 函数得到无人机的滚转（roll）、俯仰（pitch）、偏航（yaw）三方向欧拉角。

- 参数读取部分（102~106 行）

```
C++
geometry_msgs::Point hover_target;
ros::NodeHandle param_nh("~");
hover_target.x = param_nh.param("hover_x", 0.);
hover_target.y = param_nh.param("hover_y", 0.);
hover_target.z = param_nh.param("hover_z", 1.);
```

`ros::NodeHandle::param` 方法可以用于在节点启动时，读取运行需要的参数，此处的参数可以在 launch 文件中以如下方式传入节点：

```
XML
<param name='外部参数' value='外部参数的值'/>
<node pkg='xxx' type='xxx' name='xxx'>
    <param name='参数 1' value='参数 1 的值'/>
    <param name='参数 2' value='参数 2 的值'/>
```

```
    ...
</node>
```

可以注意到此处使用了一个新的`ros::NodeHandle`，且构造时传入了一个字符串类型的参数`"~"`，它表示这个`ros::NodeHandle`采用节点名作为访问所有话题/服务/参数的前缀。按上述示例 launch 传入参数时，使用`rosparam list`指令可以看到，参数 1 和参数 2 实际上被放在了`/节点名/参数 1`和`/节点名/参数 2`的位置，因此直接使用一个无参的`ros::NodeHandle`是找不到这两个参数的。而外部参数被放在了`/外部参数`的位置，它可以使用无参`ros::NodeHandle`获取到。

此处传入的三个参数表示飞机的目标悬停位置。

- 切换 OFFBOARD 状态（fsm_state = 0）

  详见`tutorial_basic/takeoff_node`代码解析的切换 OFFBOARD 状态章节。

- 解锁状态（fsm_state = 1）

  详见`tutorial_basic/takeoff_node`代码解析的解锁状态章节。

- 起飞状态（fsm_state = 2）

  详见`tutorial_basic/takeoff_node`代码解析的起飞状态章节。

- 移动和悬停状态（fsm_state = 3）

```C++
if (getLengthBetweenPoints(hover_target,
current_pose.pose.position) > 0.3) {
    last_srv_request = ros::Time::now();
}
if (ros::Time::now() - last_srv_request >
ros::Duration(3.0)) {
    fsm_state = 4;  // goto land state
} else {  // PID control
    twist.twist = get_pid_vel(hover_target);
}
```

状态转换条件：当“无人机距离目标点的距离小于`0.3`”这一条件（源 161，此处 1 行）连续满足`3`秒（源 162、164，此处 2、4 行），切换到降落状态（源 165，此处 5 行）。这里也可以理解为，切换到降落状态需要计时器（`last_srv_request`）计时`3`秒，而只要无人机距离目标点的距离大于`0.3`，计数器的读数就会重置。

状态行为：使用 PID 算法计算飞机飞往`hover_target`点应具有的速度

（源 167，此处 7 行）。

PID 控制算法是一种广泛应用于工业和其他领域的控制算法，它基于某受控量的测量值 u(t)和目标值 r(t)之间偏差 e(t)的比例 Ke(t)，微分 Ke'(t)和积分 K∫e(t)dt，计算受控量的输出，使得受控量能尽可能快、尽可能准、尽可能稳的固定在目标值上的算法，其公式如下所示：

$$u(t) = K_p e(t) + K_i \int_0^t e(t)dt + K_d \frac{de(t)}{dt}$$

由于计算机只能处理离散值，在实际编程中，我们总是保存上一刻偏差的值，用这一刻偏差减上一刻偏差，除以每刻之间的时间估算偏差的导数，即

$$\frac{de(t)}{dt} \approx \frac{e(t_k) - e(t_{k-1})}{\Delta t}$$

而偏差的积分取每一刻偏差的和，乘每刻之间的时间来估算，即

$$\int_0^t e(t)dt \approx \Delta t \sum_{i=0}^{k} e(t_i)$$

`get_pid_vel` 函数声明在源代码 43 行，定义如下：

```C++
geometry_msgs::Point last_err;
geometry_msgs::Point err_sum;
double last_yaw_err = 0.;
double yaw_err_sum = 0.;
ros::Time last_pid_control_time;
geometry_msgs::Twist get_pid_vel(geometry_msgs::Point
target) {
    ros::Time currentStamp = current_pose.header.stamp;
    ros::Duration dt = currentStamp -
last_pid_control_time;
    if (dt.toSec() > 0.2) {
        err_sum.x = 0.;
        err_sum.y = 0.;
        err_sum.z = 0.;
        yaw_err_sum = 0.;
    }

    geometry_msgs::Point err;
    double absErr = getLengthBetweenPoints(target,
current_pose.pose.position, &err.x, &err.y, &err.z);

    double y_err = 0. - current_rpy.z;
    double dy_err = (y_err - last_yaw_err) /
```

```
dt.toSec();

    geometry_msgs::Twist ret;
    ret.angular.z = VEL_P * y_err + VEL_I * yaw_err_sum
+ VEL_D * dy_err;

    if (absErr > 0.8) {
        ret.linear.x = err.x * 0.8 / absErr;
        ret.linear.y = err.y * 0.8 / absErr;
        ret.linear.z = err.z * 0.8 / absErr;

        err_sum.x = .0;
        err_sum.y = .0;
        err_sum.z = .0;
    } else {
        geometry_msgs::Point d_err;
        d_err.x = (err.x - last_err.x) / dt.toSec();
        d_err.y = (err.y - last_err.y) / dt.toSec();
        d_err.z = (err.z - last_err.z) / dt.toSec();

        ret.linear.x = VEL_P * err.x + VEL_I *
err_sum.x + VEL_D * d_err.x;
        ret.linear.y = VEL_P * err.y + VEL_I *
err_sum.y + VEL_D * d_err.y;
        ret.linear.z = VEL_P * err.z + VEL_I *
err_sum.z + VEL_D * d_err.z;

        err_sum.x += err.x * dt.toSec();
        err_sum.y += err.y * dt.toSec();
        err_sum.z += err.z * dt.toSec();
    }

    last_err = err;
    last_yaw_err = y_err;
    yaw_err_sum += y_err * dt.toSec();
    last_pid_control_time = currentStamp;
    return ret;
}
```

首先计算当前时刻与上一时刻之间的时间差（第 7~8 行），若时间差大于 `0.2` 秒（第 9 行），说明无人机已经有一段时间没有进行 PID 控制了，这可能是因为无人机刚刚进入 PID 控制的状态，因此这时应当考虑清空偏差的积分（第 10~13 行）。

接下来，代码对无人机三方向（17 行）和偏航（19 行）的偏差进行了

计算，其中 19 行的`0.`表示的是无人机的目标偏航角，`0.`表示朝向起飞时的正前方，也就是说，我们的这段代码会使飞机始终朝向正前方飞行。如果用户希望在 PID 控制过程中使无人机转向其它偏航角，则需要自行修改相关代码。请务必注意当无人机不朝向正前方时，全局坐标系 map 和本地坐标系 base_link 的 x 轴和 y 轴朝向会变得不再一致，为了将 PID 控制得到的全局坐标系下的速度转换到本地坐标系，需要使用类似如下的公司进行坐标转换（θ表示逆时针旋转角度）：

$$y_1 = x_1 cos\theta - x_2 sin\theta y_2 = x_1 sin\theta + x_2 cos\theta$$

在这之后我们得到了偏航误差的导数（第 20 行），并最终计算出无人机的目标偏航速度（第 23 行）。

第 25~46 行都是在计算无人机的线速度，如果偏差大于`0.8`米（第 25 行），那么我们认为无人机距离目标位置过于遥远，此时为了限制无人机的最大飞行速度以保证安全，不适合直接使用 PID 控制，故将偏差直接作为速度，同时将偏差积分归零。如果偏差小于`0.8`米，我们就使用 PID 控制对无人机位置进行校准。PID 控制的三个系数`VEL_P`、`VEL_I`、`VEL_D`的定义位于源文件第 9~11 行：

```C++
#define VEL_P 1.0
#define VEL_I 0.0
#define VEL_D 0.0
```

可以看到代码只使用了 P 控制，如果用户想要追求更加精确的控制效果，需要自行调整以上三个参数。

最后，将上次偏差更新为当前偏差，将当前偏差加进偏差积分中，并更新上次 PID 控制时间（第 48~51 行）。

- 降落状态（fsm_state = 4）

  详见`tutorial_basic/takeoff_node`代码解析的降落状态章节。

tutorial_navigation

launch 文件

- mid360.launch: 启动 MID360 雷达驱动和 FAST_LIO 定位算法，并构建 tf 树。
  - 代码解析：

```XML
<?xml version="1.0"?>
<launch>
```

```
    <include file="$(find
livox_ros_driver2)/launch_ROS1/msg_MID360.launch"/>
    <include file="$(find
fast_lio)/launch/mapping_mid360.launch">
        <arg name="rviz" value="false"/>
    </include>
    <node pkg="tutorial_navigation"
type="odom_forward_node" name="odom_forward_node"/>
    <node pkg="tf" type="static_transform_publisher"
name="tf_body_to_base_link_broadcaster" args="0 0 0 0 0 0
1 body base_link 100" />
    <node pkg="tf" type="static_transform_publisher"
name="tf_odom_to_camera_init_broadcaster" args="0 0 0 0 0
0 1 odom camera_init 100" />
    <node pkg="tf" type="static_transform_publisher"
name="tf_map_to_odom_broadcaster" args="0 0 0 0 0 0 1 map
odom 100" />
</launch>
```

本 launch 文件第 3~6 行是启动 MID360 雷达驱动和 FAST_LIO 定位算法的部分，没什么好说的。需要注意的是第 7~10 行，其中第 7 行启动了 `odom_forward_node` 节点，该节点的功能是将 FAST_LIO 的定位结果转发到`/mavros/vision_pose/pose` 话题中，只有将定位消息转发到了这一话题中，飞控才能收到定位消息，无人机才能正常起飞，否则飞控会认为定位失败，拒绝解锁。

第 8~10 行主要用于构建 tf 树，需要这些节点的原因在于 move_base 进行导航时，会尝试读取 `map->base_link` 和 `odom->base_link` 变换，而 FAST_LIO 帮我们构建的是 `camera_init->body` 变换，所以需要构建 `map->odom`，`odom->camera_init` 的恒等变换和 `body->base_link` 的恒等变换。

- move_base.launch: 启动 move_base。
  - 代码解析:

```XML
<?xml version="1.0"?>
<launch>
    <node pkg="move_base" type="move_base" respawn="false"
name="move_base" output="screen">
        <rosparam file="$(find
tutorial_navigation)/param/costmap_common_params.yaml"
command="load" ns="local_costmap"/>
```

```
        <rosparam file="$(find
tutorial_navigation)/param/costmap_common_params.yaml"
command="load" ns="global_costmap"/>
        <rosparam file="$(find
tutorial_navigation)/param/global_costmap_params.yaml"
command="load" ns="global_costmap"/>
        <rosparam file="$(find
tutorial_navigation)/param/local_costmap_params.yaml"
command="load" ns="local_costmap"/>
        <rosparam file="$(find
tutorial_navigation)/param/global_planner_params.yaml"
command="load" ns="GlobalPlanner"/>
        <rosparam file="$(find
tutorial_navigation)/param/dwa_local_planner_params.yaml"
command="load" ns="DWAPlannerROS"/>
        <rosparam file="$(find
tutorial_navigation)/param/move_base_params.yaml"
command="load"/>

        <remap from="odom" to="/Odometry"/>
    </node>
</launch>
```

本 launch 文件第 3~10 行用于加载一系列 move_base 的参数，请注意在无人机起飞并进行导航任务之前，**这些参数必须由用户自行调整至合适值**，如果用户想更换本地规划执行器，需要修改第 9 行的参数位置。

move_base 需要从`/odom`话题中读取里程计读数，但 FAST_LIO 定位算法输出的里程计读数在`/Odometry`中，因此第 12 行对 move_base 读取里程计读数的位置进行了 remap。

- run_navigation.launch: 启动导航飞行节点，节点说明见后文。**请注意运行此节点之前务必测试并调整导航参数**。
  - 参数:
    - target_x: 导航目标位置的 x 坐标，默认为`3`。
    - target_y: 导航目标位置的 y 坐标，默认为`0`。
    - target_z: 导航目标位置的 z 坐标，默认为`1`。
    - run_mavros: 是否同时启动 mavros，默认为`true`。
    - run_move_base: 是否同时启动 move_base，默认为`true`。
    - run_fast_lio: 是否同时启动 MID360 驱动和 FAST_LIO 定位算法，默认为`true`。

节点

- gps_tf_forward_node: 将`/mavros/local_position/pose`话题内的消息发布到TF 变换上，仅用作仿真调试，实物飞行时没有用到。
  - 参数:
    - parent:TF 变换的父节点，默认为`odom`。
    - child: TF 变换的子节点，默认为`base_link`。
  - 代码解析:

    本节点实现较为简单，原理是订阅`/mavros/local_position/pose`话题，在回调函数中将消息内的数据通过`tf::TransformBroadcaster::sendTransform`发布到 TF 树中，在此不过多赘述。在本节点中使用了 Boost 库的`boost::bind`函数，它是一种能将具有多个参数的函数，赋予部分参数值，变成一个新的函数的函数，其原理类似下面这个`getPlusFunc`函数。

    ```
    C++
    int sum(int a, int b) {
        return a + b;
    }

    std::function<int(int)> getPlusFunc(int b) {
        return [=](int a) -> int { return sum(a, b); }  // 这个语法叫 lambda 表达式
    }

    int main() {
        auto plus_5_func = getPlusFunc(5);
        std::cout << plus_5_func(2) << std::endl;  // Print: 7
    }
    ```

    但由于`ros::NodeHandle::subscribe`不支持传入 lambda 表达式作为回调函数，所以只能使用`boost::bind`。

- odom_forward_node: 将`/Odometry`话题内来自 FAST_LIO 的定位数据转发到`/mavros/vision_pose/pose`话题中，只有将定位消息转发到了这一话题中，飞控才能收到定位消息，无人机才能正常起飞，否则飞控会认为定位失败，拒绝解锁。

- navigation_node: 以`(0.0, 0.0, 0.4)`速度起飞，当海拔高于设定值时向给定点导航，当距离给定点距离小于`0.3`时终止导航，改为基于 PID 控制的悬停，悬停`1`秒后以`(0.0, 0.0, -0.2)`的速度降落，当海拔低于`0.1`时切换到`AUTO.LAND`模式。**请注意运行此节点之前务必测试并调整导航参数**。

- 参数:
  - target_x: 导航目标点的 x 坐标，默认为 `0`。
  - target_y: 导航目标点的 y 坐标，默认为 `0`。
  - target_z: 导航平面高度（飞机由起飞转为导航的高度），默认为 `1`。
- 代码解析:
  - 节点初始化，话题订阅，创建话题发布句柄和服务调用句柄，等待 FCU 连接部分

    见 `tutorial_basic/takeoff_node` 代码解析第一部分。本节点额外订阅或发布了以下三个话题。

    ```
    C++
    ros::Subscriber move_base_cmd_sub =
    nh.subscribe<geometry_msgs::Twist>("cmd_vel", 1,
    move_base_cmd_vel_cb);
    ros::Publisher goal_pub =
    nh.advertise<geometry_msgs::PoseStamped>("move_base_sim
    ple/goal", 1);
    ros::Publisher cancel_pub =
    nh.advertise<actionlib_msgs::GoalID>("move_base/cancel"
    , 1);
    ```

    其中`"move_base_simple/goal"`话题是用来向 move_base 发起导航指令的，只有向这个话题中发送消息，move_base 才会运行并输出目标速度。

    move_base 收到导航指令并运行导航算法后，会将无人机需要飞行的速度发送到`"cmd_vel"`话题，该话题会重新被该节点订阅，回调函数如下

    ```
    C++
    geometry_msgs::Twist move_base_twist;
    void move_base_cmd_vel_cb(const
    geometry_msgs::Twist::ConstPtr &msg) {
        move_base_twist = *msg;
    }
    ```

    在状态机内，该速度会被重新到 `/mavros/setpoint_velocity/cmd_vel`，那么能不能使用 remap 让 move_base 直接将速度发送到 mavros 呢，答案是不能，这是因为 move_base 发送的速度一方面仅包括 x 和 y 两个方向的速度，另一方面还包括偏航速度，我们既需要让飞机的高度固定在给定值，还需要让飞

机不受到 move_base 的偏航速度的影响，保持向前。

`"move_base/cancel"`话题是用来停止 move_base 的工作的，考虑到 move_base 的速度会发送给本节点而不是直接发送给 mavros，其实导航结束后可以不使用本话题停止导航，使用本话题更多是出于一种性能考虑。

- 参数读取部分

  详见 `tutorial_basic/vel_control_node` 代码解析的参数读取部分章节。

- 切换 OFFBOARD 状态（fsm_state = 0）

  详见 `tutorial_basic/takeoff_node` 代码解析的切换 OFFBOARD 状态章节。

- 解锁状态（fsm_state = 1）

  详见 `tutorial_basic/takeoff_node` 代码解析的解锁状态章节。

- 起飞状态（fsm_state = 2）

```C++
if (current_pose.pose.position.z > nav_target.z) {
    fsm_state = 3;  // goto navigation state
    geometry_msgs::PoseStamped move_base_msg;
    move_base_msg.header.frame_id = "map";
    move_base_msg.pose.position = nav_target;
    move_base_msg.pose.orientation.w = -1.0;
    goal_pub.publish(move_base_msg);
} else {
    twist.twist.linear.z = 0.4;
}
```

  详见 `tutorial_basic/takeoff_node` 代码解析的起飞状态章节。在本节点中，状态机跳出起飞状态的临界海拔取决于 `target_z` 参数（代码中的 `nav_target.z`）。同时在切出起飞状态时，发布导航任务（源 769~173 行，此处 3~7 行）。

- 导航状态（fsm_state = 3）

```C++
if (getLengthBetweenPoints(nav_target,
current_pose.pose.position) < 0.3) {
    fsm_state = 4;  // goto hover state
    actionlib_msgs::GoalID cancel_msg;
```

```
    cancel_pub.publish(cancel_msg);
    last_srv_request = ros::Time::now();
} else {
    twist.twist = move_base_twist;
    twist.twist.linear.z = std::max(-0.5, std::min(0.5,
nav_target.z - current_pose.pose.position.z));
    twist.twist.angular.z = std::max(-1.57,
std::min(1.57, -current_rpy.z));
}
```

状态转换条件：当飞机到导航目标点的距离小于 `0.3` 时（源 179 行，此处 1 行），跳转到悬停状态（源 180、183 行，此处 2、5 行），并终止导航任务（源 181~182 行，此处 3~4 行）。

状态行为：转发来自 move_base 的速度，并为该速度附加 z 方向和偏航速度（源 185~187 行，此处 7~9 行），此处为了简略，z 方向和偏航方向速度使用了一个简单的 P 控制，如果用户需要更精准的控制，可以自行将其替换为基于 PID 的控制。

- 悬停状态（fsm_state = 4）

  详见 `tutorial_basic/vel_control_node` 代码解析的移动和悬停状态章节。在本节点中悬停时间为 `1.0` 秒。

- 降落状态（fsm_state = 5）

  详见 `tutorial_basic/takeoff_node` 代码解析的降落状态章节。

tutorial_vision

launch 文件

- simple_camera_driver.launch: 启动下视相机驱动，下视相机画面会以 `sensor_msgs/Image` 类型的消息的形式发布到`"camera/image_raw"`话题中，可以使用 `rqt_image_view image:=/camera/image_raw` 命令观看相机画面。
  - 参数:
    - camera_id: 使用 `cv2.VideoCapture` 打开摄像机时提供的参数，只连接一个摄像头时一般就是 `0`，默认为 `0`。
    - frame_id: 发布的 `sensor_msgs/Image` 消息的 header 中，frame_id 的值，通常没什么用，默认为 `mono_camera`。
    - rate: 读取摄像头并将其发送到话题中的频率，不宜过大，默认为 `10`。
- circle_detector.launch: 启动霍夫圆识别节点，霍夫圆识别节点将会从 `"camera/image_raw"`话题中读取相机图像，并将识别结果以

`tutorial_vision/CircleDetectResult` 类型的消息发送到 `"/circle_detect_result"`话题中，同时该节点还支持绘制霍夫圆识别结果，将绘制好的识别结果发送到`"/circle_detect_result_img"`话题中。

- 参数:
  - run_driver: 是否同时启动下视相机驱动，默认为 `false`。
  - dp: 霍夫圆识别算法的参数，默认为 `1`。
  - minDist: 霍夫圆识别算法的参数，默认为 `20`。
  - param1: 霍夫圆识别算法的参数，默认为 `50`。
  - param2: 霍夫圆识别算法的参数，默认为 `100`。
  - minRadius: 霍夫圆识别算法的参数，默认为 `70`。
  - maxRadius: 霍夫圆识别算法的参数，默认为 `500`。
  - pubImg: 是否发布图像形式的识别结果，这将有利于调试，但可能造成额外的性能损失，默认为 `true`。

- qr_detector.launch: 启动二维码识别节点，二维码识别节点将会从 `"camera/image_raw"`话题中读取相机图像，并将识别结果以 `tutorial_vision/StringStamped` 类型的消息发送到`"/qr_detect_result"`话题中。
  - 参数:
    - run_driver: 是否同时启动下视相机驱动，默认为 `true`。
- yolov5.launch: 启动 YoloV5 识别节点，YoloV5 识别节点将会从 `"camera/image_raw"`话题中读取相机图像，并将识别结果以 `tutorial_vision/StringStamped` 类型的消息发送到`"/yolo_detect"`话题中。
  - 参数:
    - run_driver: 是否同时启动下视相机驱动，默认为 `false`。
    - yolov5_root: YoloV5 的安装位置，本代飞机已为用户安装在 `~/Libraries/yolov5`，请注意使用示例代码提供的模型需要 yolov5 的版本为 `6.2`，不能使用最新的 `7.0`，默认值`$(env HOME)/Libraries/yolov5`。
    - weights_path: 模型文件的位置，默认值为`$(find tutorial_vision)/model/example.pt`。请注意该模型为直接使用 cifar100 源数据集训练的模型，用在实景画面识别任务中可能存在一些问题，详见后文节点介绍部分。
    - device: 执行识别算法的设备，有效值有空（自动选择）、`cpu` 和 `cuda:0`，默认值为 `cuda:0`。

- confidence: 置信度，有效值范围为 `0.0~1.0`，只有识别结果中正确率最高的结果的正确率大于此值时，节点才会将识别结果发送到话题中，否则节点不会输出任何信息，为 `0.0` 时，每帧画面都会有一个识别结果，默认为 `0.85`。

- run_accurately_land.launch: 启动精准降落节点，节点说明见后文。**请注意运行此节点之前务必测试并调整霍夫圆识别参数**。
  - 参数:
    - parking_x: 无人机开始执行精准降落任务的 x 坐标，默认为 `1`。
    - parking_y: 无人机开始执行精准降落任务的 y 坐标，默认为 `0`。
    - parking_z: 无人机开始执行精准降落任务的 z 坐标，默认为 `1`。
    - run_mavros: 是否同时启动 mavros，默认为 `true`。
    - run_driver: 是否同时启动下视相机驱动，默认为 `true`。
    - run_detector: 是否同时启动霍夫圆检测节点，默认为 `true`。
    - run_fast_lio: 是否同时启动 MID360 驱动和 FAST_LIO 定位算法，默认为 `true`。
    - dp: 霍夫圆识别算法的参数，仅当 `run_detector` 为 `true` 时有效，默认为 `1`。
    - minDist: 霍夫圆识别算法的参数，仅当 `run_detector` 为 `true` 时有效，默认为 `20`。
    - param1: 霍夫圆识别算法的参数，仅当 `run_detector` 为 `true` 时有效，默认为 `50`。
    - param2: 霍夫圆识别算法的参数，仅当 `run_detector` 为 `true` 时有效，默认为 `100`。
    - minRadius: 霍夫圆识别算法的参数，仅当 `run_detector` 为 `true` 时有效，默认为 `70`。
    - maxRadius: 霍夫圆识别算法的参数，仅当 `run_detector` 为 `true` 时有效，默认为 `500`。
- run_qr_detect.launch: 启动二维码识别任务节点，节点说明见后文。
  - 参数:
    - qr_x: 无人机进行二维码识别的 x 坐标，默认为 `1`。
    - qr_y: 无人机进行二维码识别的 y 坐标，默认为 `0`。
    - qr_z: 无人机进行二维码识别的 z 坐标，默认为 `1`。

        - run_mavros: 是否同时启动 mavros，默认为 `true`。
        - run_driver: 是否同时启动下视相机驱动，默认为 `true`。
        - run_detector: 是否同时启动二维码识别节点，默认为 `true`。
        - run_fast_lio: 是否同时启动 MID360 驱动和 FAST_LIO 定位算法，默认为 `true`。
- run_yolo_detect.launch: 启动 YoloV5 识别任务节点，节点说明见后文。
    - 参数:
        - sign_x: 无人机进行目标检测的 x 坐标，默认为 `1`。
        - sign_y: 无人机进行目标检测的 y 坐标，默认为 `0`。
        - sign_z: 无人机进行目标检测的 z 坐标，默认为 `1`。
        - run_mavros: 是否同时启动 mavros，默认为 `true`。
        - run_driver: 是否同时启动下视相机驱动，默认为 `true`。
        - run_detector: 是否同时启动 YoloV5 识别节点，默认为 `true`。
        - run_fast_lio: 是否同时启动 MID360 驱动和 FAST_LIO 定位算法，默认为 `true`。
        - yolov5_root: YoloV5 的安装位置，仅当 `run_detector` 为 `true` 时有效，本代飞机已为用户安装在`~/Libraries/yolov5`，请注意使用示例代码提供的模型需要 yolov5 的版本为 `6.2`，不能使用最新的 `7.0`，默认值`$(env HOME)/Libraries/yolov5`。
        - weights_path: 模型文件的位置，仅当 `run_detector` 为 `true` 时有效，默认值为`$(find tutorial_vision)/model/example.pt`。请注意该模型为直接使用 cifar100 源数据集训练的模型，用在实景画面识别任务中可能存在一些问题，详见后文节点介绍部分。
        - device: 执行识别算法的设备，仅当 `run_detector` 为 `true` 时有效，有效值有空（自动选择）、`cpu` 和 `cuda:0`，默认值为 `cuda:0`。
        - confidence: 置信度，仅当 `run_detector` 为 `true` 时有效，有效值范围为 `0.0~1.0`，只有识别结果中正确率最高的结果的正确率大于此值时，节点才会将识别结果发送到话题中，否则节点不会输出任何信息，为 `0.0` 时，每帧画面都会有一个识别结果，默认为 `0.85`。

消息类型

- CircleInfo: 霍夫圆检测的单个结果
    - center_x (float64): 该霍夫圆在画面中的 x 像素坐标（不一定为整数）

- center_y (float64): 该霍夫圆在画面中的 y 像素坐标（不一定为整数）
- radius (float64): 该霍夫圆的像素半径（不一定为整数）

- CircleDetectResult: 单次霍夫圆检测的全部结果
  - header (std_msgs/Header): 消息头部
    - seq (uint32): 表示本条消息是该话题中的第几条消息的 ID
    - stamp (time): 总是等于输入图像的 `header.stamp`
    - frame_id (string): 总是等于输入图像的 `header.frame_id`
  - height (uint32): 输入图像的高
  - width (uint32): 输入图像的宽
  - circles (tutorial_vision/CircleInfo[]): 识别结果集，如果本次检测没有检测到霍夫圆，话题中将不会产生新消息，保证该字段至少有一个元素。
- StringStamped: 单词二维码识别和 YoloV5 目标检测的结果
  - header (std_msgs/Header): 消息头部
    - seq (uint32): 表示本条消息是该话题中的第几条消息的 ID
    - stamp (time): 总是等于输入图像的 `header.stamp`
    - frame_id (string): 总是等于输入图像的 `header.frame_id`
  - data (string[]):

    对于二维码识别任务，输入图像中有几个二维码，就有几个二维码的识别结果，如果输入图像中没有二维码，该字段为空。

    对于目标检测任务，该字段长度必为 1，表示目标检测结果，如果检测结果不满足置信度要求，输出话题`"/yolo_detect"`中将不会有新消息产生。

节点

- simple_camera_driver.py: 使用 `cv2.VideoCapture` 方法打开下视摄像机，用 cv_bridge 将其转换为 `sensor_msgs/Image` 类型的消息，并发布到话题中。
  - 参数:
    - camera_id: 使用 `cv2.VideoCapture` 打开摄像机时提供的参数，只连接一个摄像头时一般就是 `0`，默认为 `0`。
    - frame_id: 发布的 `sensor_msgs/Image` 消息的 header 中，frame_id 的值，通常没什么用，默认为 `mono_camera`。
    - rate: 读取摄像头并将其发送到话题中的频率，不宜过大，默认为 `10`。

- 代码解析:

```Python
def main(argv):
    rospy.init_node("simple_camera_driver", argv=argv)
    camera_id = rospy.get_param("~camera_id", 0)
    frame_id = rospy.get_param("~frame_id", "mono_camera")
    rate_duration = rospy.get_param("~rate", 10)

    image_pub = rospy.Publisher("camera/image_raw", Image,
queue_size=1)
    capture = cv2.VideoCapture(camera_id)
    bridge = CvBridge()
    rate = rospy.Rate(rate_duration)
    while not rospy.is_shutdown():
        ret, frame = capture.read()
        if ret:
            try:
                image_msg = bridge.cv2_to_imgmsg(frame,
"bgr8")
                image_msg.header.stamp = rospy.Time.now()
                image_msg.header.frame_id = frame_id
                image_pub.publish(image_msg)
            except CvBridgeError as e:
                rospy.logerr(e)
        rate.sleep()


if __name__ == '__main__':
    main(sys.argv)
```

在 Python 中，`if __name__ == '__main__'`指代的是只有直接运行脚本才会执行的分支，但在全局的 if 分支中声明的变量是全局变量，可能会与函数内的局部变量冲突，所以我们在`if __name__ == '__main__'`中直接调用自定义的`main`方法，这样 Python 代码的架构就与 C++类似了。

在 Python 中，初始化一个 ROS 节点使用的方法是`rospy.init_node`，它与 C++的`ros::init`方法的功能是一样的。使用`rospy.get_param`方法可以获取参数，与 C++中的`ros::NodeHandle::param`方法一致，注意到参数名字符串的开头有一个“~”，这在 Python 版本的 ROS 中就是命名空间的意思，和 NodeHandle 构造方法的参数是一样的。

`rospy.Publisher`可以创建一个话题发布句柄，原 C++中用来表示消息类型的模板参数被放到了第二个形参上。

除此之外，Python 版本的 ROS 中还用 `rospy.Rate` 和 `not rospy.is_shutdown()`替换了 C++中的 `ros::Rate` 和 `ros::ok()`，Python 版本的 ROS 节点的主循环中没有使用 `spinOnce`，这是因为 Python 版本的 ROS 是在一个新线程里执行回调函数的，因此不需要 `spinOnce` 操作也可以收到并处理消息，而 `rospy.spin()`的作用也仅剩下阻塞当前线程。

使用 `cv2.VideoCapture` 函数可以打开一个摄像机，之后使用 `cv2.VideoCapture.read` 方法可以从摄像机中读取一帧画面，该方法有两个返回值 `ret` 和 `frame`，其中 `ret` 表示本次读取是否成功，为 `True` 表示成功，而 `frame` 是以 `numpy.ndarray` 类型存储的图像数据。

cv_bridge 是一个能将 OpenCV 的图像数据和 ROS 的 `sensor_msgs/Image` 类型的消息互相转化的库，在本节点中用到了 `CvBridge.cv2_to_imgmsg` 方法来将 OpenCV 图像数据（`numpy.ndarray` 类型）转换为 ROS 消息，在之后的识别节点中还用到了 `CvBridge.imgmsg_to_cv2` 来将 ROS 消息转回 OpenCV 图像数据。`"bgr8"`参数是用来说明 OpenCV 图像数据中数据存储的方式的，`"bgr8"`指的是每个像素先存储蓝色分量，再存储绿色分量，最后存储红色分量，每个分量占用 8 个字节，是 OpenCV 处理图像时最常见的格式。

- hough_circle_detector.py: 订阅`"camera/image_raw"`话题，将读取到的图片进行霍夫圆检测，将检测结果以 `tutorial_vision/CircleDetectResult` 类型发布到`"circle_detect_result"`话题。**请注意运行此节点之前务必测试并调整霍夫圆识别参数**。
  - 参数:
    - dp: 霍夫圆识别算法的参数，默认为 `1`。
    - minDist: 霍夫圆识别算法的参数，默认为 `20`。
    - param1: 霍夫圆识别算法的参数，默认为 `50`。
    - param2: 霍夫圆识别算法的参数，默认为 `30`。
    - minRadius: 霍夫圆识别算法的参数，默认为 `0`。
    - maxRadius: 霍夫圆识别算法的参数，默认为 `0`。
  - 代码解析:
    - 接口部分

```Python
rospy.Subscriber("camera/image_raw", Image,
    lambda msg: image_cb(msg, hough_param, bridge,
circle_pub, img_pub)
```

```
)
```

`rospy.Subscriber` 方法用于订阅一个话题，`lambda x: y` 这种语法叫 lambda 表达式，其中 x 是形参，y 是一条语句，在这里相当于声明了一个只有一个形参 msg 的函数作为回调函数。

- 识别部分

本节点直接调用 OpenCV 的内置函数 `cv2.HoughCircles` 进行霍夫圆检测，用户可以自行查阅有关资料了解关于霍夫圆检测的实现细节。

- qr_detector.py: 订阅`"camera/image_raw"`话题，将读取到的图片进行二维码识别，将识别结果以 `tutorial_vision/StringStamped` 类型发布到`"qr_detect_result"`话题。
  - 代码解析:

二维码识别基于 pyzbar 实现

- yolov5_predict.py: 订阅`"camera/image_raw"`话题，将读取到的图片进行 YoloV5 目标检测，将检测结果以 `tutorial_vision/StringStamped` 类型发布到`"yolo_detect"`话题。

示例代码附带的模型（example.pt）为直接使用 cifar100 数据集原始文件训练的模型，在实物识别任务中效果可能不理想，解决这一问题有以下两种方案:

a. 使用传统计算机视觉方法，提取投放靶中央的图片，再进行识别。这种方法对选手的计算机视觉技术能力有所要求，是赛方建议的解决方案。

b. 拍摄投放靶实物照片重新训练模型。这种方法对技术能力的要求较低，但需要自行制作大量数据集，比较麻烦。

  - 参数:

    - yolov5_root: YoloV5 的安装位置，本代飞机已为用户安装在 `~/Libraries/yolov5`，请注意使用示例代码提供的模型需要 yolov5 的版本为 `6.2`，不能使用最新的 `7.0`。
    - weights_path: 模型文件的位置。
    - device: 执行识别算法的设备，有效值有空（自动选择）、`cpu` 和 `cuda:0`，默认值为空。
    - confidence: 置信度，有效值范围为 `0.0~1.0`，只有识别结果中正确率最高的结果的正确率大于此值时，节点才会将识别结果发送到话题中，否则节点不会输出任何信息，为 `0.0` 时，每帧画面都会有一个识别结果，默认为 `0.85`。

- accurately_land_node: 以`(0.0, 0.0, 0.4)`速度起飞，当海拔高于设定值时朝

给定点飞行，当距离给定点距离小于`0.3`时 PID 悬停，悬停`1.5`秒后检查下方的降落平台，以`(0.0, 0.0, -0.2)`的速度尽可能精确的降落到降落平台的圆内，当海拔低于`0.1`时切换到`AUTO.LAND`模式。

- 参数:
  - parking_x: 降落点的 x 坐标，默认为`0`。
  - parking_y: 降落点的 y 坐标，默认为`0`。
  - parking_z: 向降落点移动时的高度（飞机由起飞转为平移时的高度和开始降落时的高度），默认为`1`。
- 代码解析:
  - 节点初始化，话题订阅，创建话题发布句柄和服务调用句柄，等待 FCU 连接部分

    见`tutorial_basic/takeoff_node`代码解析第一部分。本节点额外订阅了来自霍夫圆检测节点的话题`"circle_detect_result"`。
  - 参数读取部分

    详见`tutorial_basic/vel_control_node`代码解析的参数读取部分章节。
  - 切换 OFFBOARD 状态（fsm_state = 0）

    详见`tutorial_basic/takeoff_node`代码解析的切换 OFFBOARD 状态章节。
  - 解锁状态（fsm_state = 1）

    详见`tutorial_basic/takeoff_node`代码解析的解锁状态章节。
  - 起飞状态（fsm_state = 2）

    详见`tutorial_basic/takeoff_node`代码解析的起飞状态章节。在本节点中，状态机跳出起飞状态的临界海拔取决于`parking_z`参数（代码中的`parking_target.z`）。
  - 移动和悬停状态（fsm_state = 3）

    详见`tutorial_basic/vel_control_node`代码解析的移动和悬停状态章节。
  - 等待识别圆状态（fsm_state = 4）

```
C++
if (ros::Time::now() -
circle_detect_result.header.stamp < ros::Duration(0.5)
```

```
&& circle_detect_result.circles.size() > 0) {
    fsm_state = 5;  // goto land state
} else {  // PID control
    twist.twist = get_pid_vel(parking_target);
}
```

状态转换条件：当最新一条关于霍夫圆检测节点检测结果的消息是在 `0.5` 秒内发出的（源 220，此处 1 行），且本条消息包含有效的霍夫圆（源 221，此处 1 行）时，切换到降落状态（源 222，此处 2 行）。

状态行为：PID 悬停在降落开始点（源 224，此处 4 行）。

该状态只是为了保证起降平台被成功识别，一般情况下此状态在首次切入时就会因为状态转换条件满足而切走。

- 降落状态（fsm_state = 5）

```C++
if (current_state.mode == "AUTO.LAND") {
    fsm_state = -1;  // goto do nothing state
} else if (current_pose.pose.position.z < 0.1) {
    if (ros::Time::now() - last_srv_request >
ros::Duration(0.5)) {
        mavros_msgs::SetMode land_set_mode;
        land_set_mode.request.custom_mode =
"AUTO.LAND";
        set_mode_client.call(land_set_mode);
        last_srv_request = ros::Time::now();
    }
} else if (circle_detect_result.circles.size() > 0) {
    geometry_msgs::Point err;
    err.x = circle_detect_result.width / 2.0 -
circle_detect_result.circles[0].center_x;
    err.y = circle_detect_result.height / 2.0 -
circle_detect_result.circles[0].center_y;
    twist.twist = get_pix_pid_vel(err);
    twist.twist.linear.z = -0.2;
} else {
    twist.twist.linear.z = -0.2;
}
```

状态转换条件：若飞行状态已切换到 AUTO.LAND（源 228，此处 1 行），切换到空状态（源 229，此处 2 行）。

状态行为：若无人机的海拔小于 `0.1`（源 230，此处 3 行），每 `0.5` 秒（源 231，此处 4 行）调用 `set_mode_client` 服务尝试切换到

AUTO.LAND 飞行状态（源 232~235，此处 5~8 行）；否则，若当前下视相机视野内有霍夫圆（源 237，此处 10 行），对准霍夫圆的圆心进行 x 和 y 方向的 PID 控制，并以`-0.2`的 z 方向速度降落（源 238~242，此处 11~15 行）；否则，直接以`(0.0, 0.0, -0.2)`的速度降落（源 244，此处 17 行）（这种情况多发生在飞机已经非常靠近起降平台，即将降落成功时）。

`get_pix_pid_vel`函数的定义与`get_pid_vel`函数非常相似，只有一点需要注意，即以下关于 x 和 y 方向速度赋值的代码（源 127~128 行）：

```C++
ret.linear.x = PIX_VEL_P * err.y + PIX_VEL_I *
pix_err_sum.y + PIX_VEL_D * d_err.y;
ret.linear.y = PIX_VEL_P * err.x + PIX_VEL_I *
pix_err_sum.x + PIX_VEL_D * d_err.x;
```

可以看到 x 方向的速度是以-y 方向的偏差赋值的，y 方向的速度是以-x 方向的偏差赋值的，这是因为当下视相机的正上方指向飞机的正前方时，在像素坐标系下，x 增大的方向是行方向，即无人机的右方向，即 y 减小的方向，而 y 增大的方向是列方向，即无人机的后方向，即 x 减小的方向，在这里实际上进行了一个坐标系转换。如果用户需要在降落过程中引入对偏航的控制，则还需要进一步修改此变换关系。

另外，像素偏差是以像素而不是米作为单位的，因此**`get_pix_pid_vel`PID 参数项远小于`get_pid_vel`**，在示例代码中，`PIX_VEL_P`的默认值为`0.001`，**请注意千万不要将像素偏差的 PID 参数调的过于大，否则可能会有安全问题。**

- qr_detect_node: 以`(0.0, 0.0, 0.4)`速度起飞，当海拔高于设定值时朝给定点飞行，当距离给定点距离小于`0.3`时 PID 悬停，悬停期间扫描下方的二维码，并在控制台中打印扫描结果，扫描成功后悬停`1.5`秒，之后回到起飞点处，并以`(0.0, 0.0, -0.2)`的速度降落。
  - 参数:
    - qr_x: 执行二维码扫描任务的 x 坐标，默认为`0`。
    - qr_y: 执行二维码扫描任务的 y 坐标，默认为`0`。
    - qr_z: 向执行任务点移动时的高度（飞机由起飞转为平移时的高度和回到起飞点时的高度），默认为`1`。
  - 代码解析:
    - 节点初始化，话题订阅，创建话题发布句柄和服务调用句柄，等待 FCU 连

接部分

见 `tutorial_basic/takeoff_node` 代码解析第一部分。本节点额外订阅了来自二维码识别节点的话题`"qr_detect_result"`。

- 参数读取部分

  详见 `tutorial_basic/vel_control_node` 代码解析的参数读取部分章节。

- 切换 OFFBOARD 状态（fsm_state = 0）

  详见 `tutorial_basic/takeoff_node` 代码解析的切换 OFFBOARD 状态章节。

- 解锁状态（fsm_state = 1）

  详见 `tutorial_basic/takeoff_node` 代码解析的解锁状态章节。

- 起飞状态（fsm_state = 2）

  详见 `tutorial_basic/takeoff_node` 代码解析的起飞状态章节。在本节点中，状态机跳出起飞状态的临界海拔取决于 `qr_z` 参数（代码中的 `qr_target.z`）。

- 移动到任务点状态（fsm_state = 3）

- 悬停状态（fsm_state = 5）

- 返回出生点状态（fsm_state = 6）

  以上三个状态详见 `tutorial_basic/vel_control_node` 代码解析的移动和悬停状态章节。

- 扫描二维码状态（fsm_state = 4）

```C++
if (ros::Time::now() - qr_result.header.stamp <
ros::Duration(1.0) && qr_result.data.size() > 0) {
    std::cout << "QR Detected: " << qr_result.data[0]
<< std::endl;
    fsm_state = 5;  // goto hover state
    last_srv_request = ros::Time::now();
} else {
    twist.twist = get_pid_vel(qr_target);
}
```

  状态转换条件：当最新一条关于二维码识别节点检测结果的消息是在 `1.0` 秒内发出的，且本条消息包含有效的二维码识别结果（源 184，此处 1 行）时，输出识别结果（源 185，此处 2 行），并切换到降落状态

（源 186~187，此处 3~4 行）。

状态行为：PID 悬停在扫描点（源 189，此处 6 行）。

- 降落状态（fsm_state = 7）

  详见 `tutorial_basic/takeoff_node` 代码解析的降落状态章节。

- yolo_detect_node: 以`(0.0, 0.0, 0.4)`速度起飞，当海拔高于设定值时朝给定点飞行，当距离给定点距离小于 `0.3` 时 PID 悬停，悬停期间进行下方图片的目标检测，并在控制台中打印物体类别，扫描成功后悬停 `1.5` 秒，之后回到起飞点处，并以`(0.0, 0.0, -0.2)`的速度降落。
  - 参数:
    - sign_x: 执行目标检测任务的 x 坐标，默认为 `0`。
    - sign_y: 执行目标检测任务的 y 坐标，默认为 `0`。
    - sign_z: 向执行任务点移动时的高度（飞机由起飞转为平移时的高度和回到起飞点时的高度），默认为 `1`。
  - 代码解析:
    - 节点初始化，话题订阅，创建话题发布句柄和服务调用句柄，等待 FCU 连接部分

      见 `tutorial_basic/takeoff_node` 代码解析第一部分。本节点额外订阅了来自二维码识别节点的话题`"qr_detect_result"`。

    - 参数读取部分

      详见 `tutorial_basic/vel_control_node` 代码解析的参数读取部分章节。

    - 切换 OFFBOARD 状态（fsm_state = 0）

      详见 `tutorial_basic/takeoff_node` 代码解析的切换 OFFBOARD 状态章节。

    - 解锁状态（fsm_state = 1）

      详见 `tutorial_basic/takeoff_node` 代码解析的解锁状态章节。

    - 起飞状态（fsm_state = 2）

      详见 `tutorial_basic/takeoff_node` 代码解析的起飞状态章节。在本节点中，状态机跳出起飞状态的临界海拔取决于 `qr_z` 参数（代码中的 `qr_target.z`）。

    - 移动到任务点状态（fsm_state = 3）

    - 悬停状态（fsm_state = 5）

- 返回出生点状态（fsm_state = 6）

  以上三个状态详见 `tutorial_basic/vel_control_node` 代码解析的移动和悬停状态章节。

- 目标检测状态（fsm_state = 4）

```C++
if (ros::Time::now() - sign_result.header.stamp < ros::Duration(3.0) && sign_result.data.size() > 0) {
    std::cout << "Yolo Detected: " << sign_result.data[0] << std::endl;
    fsm_state = 5;  // goto hover state
    last_srv_request = ros::Time::now();
} else {
    twist.twist = get_pid_vel(sign_target);
}
```

  状态转换条件：当最新一条关于 YoloV5 目标检测节点检测结果的消息是在 `3.0` 秒内发出的，且本条消息包含有效的目标检测结果（源 184，此处 1 行）时，输出识别结果（源 185，此处 2 行），并切换到降落状态（源 186~187，此处 3~4 行）。

  状态行为：PID 悬停在检测点（源 189，此处 6 行）。

  与二维码识别不一样的是，由于目标检测任务的失败率较高，无人机可能会因为始终没有成功检测出物品类别而卡在这个状态。

- 降落状态（fsm_state = 7）

  详见 `tutorial_basic/takeoff_node` 代码解析的降落状态章节。

tutorial_catapult

launch 文件

- catapult_driver.launch: 启动投放装置驱动，启动后通过`"servo/front"`、`"servo/back_1"`、`"servo/back_2"`三个 `std_msgs/Bool` 类型的话题控制投放装置，发送 `data` 为 `true` 消息时对应仓位打开。例:

```Bash
rostopic pub /servo/front std_msgs/Bool "data: true"
```

- run_item_deliver.launch: 启动投放任务节点，节点说明见后文
  - 参数:

- deliver_x: 投放点 x 坐标，默认为 `1`。
- deliver_y: 投放点 y 坐标，默认为 `0`。
- deliver_z: 投放高度，默认为 `1`。
- run_mavros: 是否同时启动 mavros，默认为 `true`。
- run_driver: 是否同时启动投放装置驱动，默认为 `true`。
- run_fast_lio: 是否同时启动 MID360 驱动和 FAST_LIO 定位算法，默认为 `true`。

节点

- actuator_driver: 通过飞控的 I/O PWM OUT 口控制投放装置的驱动
  - 参数:
    - front_servo_pin: 投放装置三个仓位中，控制最大的那个仓位的舵机的通道是几号，如果向 `servo/front` 发送消息时，打开/关闭的仓位不是该仓位，考虑修改该参数，有效取值有 `0`、`1`、`2`，默认为 `0`。
    - back_servo_1_pin: 当飞机朝前，俯视飞机时，控制位于右下角的仓位的舵机的通道是几号，如果向 `servo/back_1` 发送消息时，打开/关闭的仓位不是该仓位，考虑修改该参数，有效取值有 `0`、`1`、`2`，默认为 `1`。
    - back_servo_2_pin: 当飞机朝前，俯视飞机时，控制位于左下角的仓位的舵机的通道是几号，如果向 `servo/back_2` 发送消息时，打开/关闭的仓位不是该仓位，考虑修改该参数，有效取值有 `0`、`1`、`2`，默认为 `2`。
    - front_servo_open_dir: 控制**最大**仓位舵机时，向 mavros 发送值为几的控制信号能使仓位打开，如果控制时该仓位的打开/关闭状态与预期相反，考虑反转该参数的符号，有效值`-1`、`1`，默认为`-1`。
    - back_servo_1_open_dir: 控制**右侧小**仓位舵机时，向 mavros 发送值为几的控制信号能使仓位打开，如果控制时该仓位的打开/关闭状态与预期相反，考虑反转该参数的符号，有效值`-1`、`1`，默认为`-1`。
    - back_servo_1_open_dir: 控制**左侧小**仓位舵机时，向 mavros 发送值为几的控制信号能使仓位打开，如果控制时该仓位的打开/关闭状态与预期相反，考虑反转该参数的符号，有效值`-1`、`1`，默认为 `1`。

    如果舵机打开/关闭时，舵机的旋转角度不合适，应当修改飞控参数。
  - 代码解析:

```C++
#include <ros/ros.h>
```

```
#include <std_msgs/Bool.h>
#include <mavros_msgs/ActuatorControl.h>

void control_cb(const std_msgs::Bool::ConstPtr &msg,
                const ros::Publisher& mix_pub,
                int pin,
                int open_dir,
                mavros_msgs::ActuatorControl *control_msg)
{
    control_msg->controls[pin] = msg->data ? open_dir : -
open_dir;
    mix_pub.publish(*control_msg);
}

int main(int argc, char** argv) {
    ros::init(argc, argv, "actuator_driver");
    ros::NodeHandle nh;

    ros::NodeHandle param_nh("~");
    int front_servo_pin =
param_nh.param("front_servo_pin", 0);
    int back_servo_1_pin =
param_nh.param("back_servo_1_pin", 1);
    int back_servo_2_pin =
param_nh.param("back_servo_2_pin", 2);
    int front_servo_open_dir =
param_nh.param("front_servo_open_dir", -1);
    int back_servo_1_open_dir =
param_nh.param("back_servo_1_open_dir", -1);
    int back_servo_2_open_dir =
param_nh.param("back_servo_2_open_dir", 1);

    ros::Publisher claw_pub_mix =
nh.advertise<mavros_msgs::ActuatorControl>("/mavros/actuat
or_control", 1);
    mavros_msgs::ActuatorControl control_msg;
    control_msg.group_mix = 2;
    control_msg.controls[front_servo_pin] = -
front_servo_open_dir;
    control_msg.controls[back_servo_1_pin] = -
back_servo_1_open_dir;
    control_msg.controls[back_servo_2_pin] = -
back_servo_2_open_dir;
    claw_pub_mix.publish(control_msg);
```

```
    ros::Subscriber control_sub_l =
nh.subscribe<std_msgs::Bool>("servo/front", 1,
boost::bind(&control_cb, _1, claw_pub_mix,
front_servo_pin, front_servo_open_dir, &control_msg));
    ros::Subscriber control_sub_1 =
nh.subscribe<std_msgs::Bool>("servo/back_1", 1,
boost::bind(&control_cb, _1, claw_pub_mix,
back_servo_1_pin, back_servo_1_open_dir, &control_msg));
    ros::Subscriber control_sub_2 =
nh.subscribe<std_msgs::Bool>("servo/back_2", 1,
boost::bind(&control_cb, _1, claw_pub_mix,
back_servo_2_pin, back_servo_2_open_dir, &control_msg));
    ros::spin();
    return 0;
}
```

上位机控制飞控从 I/O PWM OUT 输出舵机控制 PWM 信号，需要向 `"/mavros/actuator_control"`话题发送 `mavros_msgs::ActuatorControl` 类型的消息（26 行）。该消息有两个最关键的成员 `group_mix` 和 `controls`，其中 `group_mix` 表示混控组，应当为 `2`（28 行），`controls` 是通过 I/O PWM OUT 输出的 8 个 PWM 信号，投放装置只有三个舵机，所以这里只用到了前三个通道，索引为 3~7 的通道没有被使用。

- actuator_driver（GPIO 版本）：通过 GPIO 控制投放装置的驱动
  - 参数：
    - front_servo_pin: 投放装置三个仓位中，控制最大的那个仓位的舵机的 GPIO 口（使用 BOARD 模式的编号），如果向 `servo/front` 发送消息时，打开/关闭的仓位不是该仓位，考虑修改该参数。对于 Jetson NX，有效取值有 `15`、`32`、`33`，默认为 `15`。
    - back_servo_1_pin: 当飞机朝前，俯视飞机时，控制位于右下角的仓位的舵机的 GPIO 口（使用 BOARD 模式的编号），如果向 `servo/back_1` 发送消息时，打开/关闭的仓位不是该仓位，考虑修改该参数。对于 Jetson NX，有效取值有 `15`、`32`、`33`，默认为 `32`。
    - back_servo_2_pin: 当飞机朝前，俯视飞机时，控制位于左下角的仓位的舵机的 GPIO 口（使用 BOARD 模式的编号），如果向 `servo/back_2` 发送消息时，打开/关闭的仓位不是该仓位，考虑修改该参数。对于 Jetson NX，有效取值有 `15`、`32`、`33`，默认为 `33`。
    - front_servo_open_width: **打开最大**仓位舵机的 PWM 信号的占空比，如果打

开该仓位时的舵机角度不合适，考虑修改该参数，有效值取值范围为 `0.0~100.0`，默认为 `9.0`。

- front_servo_close_width: **关闭最大**仓位舵机的 PWM 信号的占空比，如果关闭该仓位时的舵机角度不合适，考虑修改该参数，有效值取值范围为 `0.0~100.0`，默认为 `12.0`。
- back_servo_1_open_width: **打开右侧小**仓位舵机的 PWM 信号的占空比，如果打开该仓位时的舵机角度不合适，考虑修改该参数，有效值取值范围为 `0.0~100.0`，默认为 `6.0`。
- back_servo_1_close_width: **关闭右侧小**仓位舵机的 PWM 信号的占空比，如果关闭该仓位时的舵机角度不合适，考虑修改该参数，有效值取值范围为 `0.0~100.0`，默认为 `8.5`。
- back_servo_2_open_width: **打开左侧小**仓位舵机的 PWM 信号的占空比，如果打开该仓位时的舵机角度不合适，考虑修改该参数，有效值取值范围为 `0.0~100.0`，默认为 `8.5`。
- back_servo_2_close_width: **关闭左侧小**仓位舵机的 PWM 信号的占空比，如果关闭该仓位时的舵机角度不合适，考虑修改该参数，有效值取值范围为 `0.0~100.0`，默认为 `6.0`。

- 代码解析：

```C++
#include <ros/ros.h>
#include <std_msgs/Bool.h>
#include <JetsonGPIO.h>

#define SERVO_FREQ 50

void control_cb(const std_msgs::Bool::ConstPtr &msg,
GPIO::PWM *pwm, double open_width, double close_width) {
    pwm->ChangeDutyCycle(msg->data ? open_width :
close_width);
}

int main(int argc, char** argv) {
    ros::init(argc, argv, "actuator_driver");
    ros::NodeHandle param_nh("~");
    int front_servo_pin =
param_nh.param("front_servo_pin", 15);
    int back_servo_1_pin =
param_nh.param("back_servo_1_pin", 32);
```

```
    int back_servo_2_pin =
param_nh.param("back_servo_2_pin", 33);
    double front_servo_open_width =
param_nh.param("front_servo_open_width", 9.0);
    double front_servo_close_width =
param_nh.param("front_servo_close_width", 12.0);
    double back_servo_1_open_width =
param_nh.param("back_servo_1_open_width", 6.0);
    double back_servo_1_close_width =
param_nh.param("back_servo_1_close_width", 8.5);
    double back_servo_2_open_width =
param_nh.param("back_servo_2_open_width", 8.5);
    double back_servo_2_close_width =
param_nh.param("back_servo_2_close_width", 6.0);
    GPIO::setmode(GPIO::BOARD);
    GPIO::setup(front_servo_pin, GPIO::OUT);
    GPIO::setup(back_servo_1_pin, GPIO::OUT);
    GPIO::setup(back_servo_2_pin, GPIO::OUT);
    auto servo_1 = GPIO::PWM(front_servo_pin, SERVO_FREQ);
    auto servo_1 = GPIO::PWM(back_servo_1_pin,
SERVO_FREQ);
    auto servo_2 = GPIO::PWM(back_servo_2_pin,
SERVO_FREQ);
    servo_1.start(front_servo_close_width);
    servo_1.start(back_servo_1_close_width);
    servo_2.start(back_servo_2_close_width);

    ros::NodeHandle nh;
    ros::Subscriber control_sub_1 =
nh.subscribe<std_msgs::Bool>("servo/front", 1,
boost::bind(&control_cb, _1, &servo_1,
front_servo_open_width, front_servo_close_width));
    ros::Subscriber control_sub_1 =
nh.subscribe<std_msgs::Bool>("servo/back_1", 1,
boost::bind(&control_cb, _1, &servo_1,
back_servo_1_open_width, back_servo_1_close_width));
    ros::Subscriber control_sub_2 =
nh.subscribe<std_msgs::Bool>("servo/back_2", 1,
boost::bind(&control_cb, _1, &servo_2,
back_servo_2_open_width, back_servo_2_close_width));
    ros::spin();

    servo_1.stop();
    servo_1.stop();
```

```
    servo_2.stop();
    GPIO::cleanup();
    return 0;
}
```

基于 JetsonGPIO 库输出 PWM 信号来控制舵机，使用 `GPIO::PWM` 创建 PWM 控制句柄时，第二个参数 `SERVO_FREQ` 是 PWM 信号的频率，固定为 50。

- item_deliver_node: 以`(0.0, 0.0, 0.4)`速度起飞，当海拔高于设定值时朝给定点飞行，当距离给定点距离小于 `0.3` 时 PID 悬停，悬停期间打开投放装置，投放完成后悬停 `1.5` 秒，之后回到起飞点处，并以`(0.0, 0.0, -0.2)`的速度降落。
  - 参数：
    - deliver_x: 投放点的 x 坐标，默认为 `0`。
    - deliver_y: 投放点的 y 坐标，默认为 `0`。
    - deliver_z: 飞机执行投放任务的高度（飞机由起飞转为平移时的高度和回到起飞点时的高度），默认为 `1`。
  - 代码解析：
    - 节点初始化，话题订阅，创建话题发布句柄和服务调用句柄，等待 FCU 连接部分

      见 `tutorial_basic/takeoff_node` 代码解析第一部分。本节点额外发布了用于控制投放装置大仓位舵机的话题`"servo/front"`，可以在 launch 文件中使用 remap 标签来使本节点改为打开其它仓位。
    - 参数读取部分

      详见 `tutorial_basic/vel_control_node` 代码解析的参数读取部分章节。
    - 切换 OFFBOARD 状态（fsm_state = 0）

      详见 `tutorial_basic/takeoff_node` 代码解析的切换 OFFBOARD 状态章节。
    - 解锁状态（fsm_state = 1）

      详见 `tutorial_basic/takeoff_node` 代码解析的解锁状态章节。
    - 起飞状态（fsm_state = 2）

      详见 `tutorial_basic/takeoff_node` 代码解析的起飞状态章节。在本节点中，状态机跳出起飞状态的临界海拔取决于 `qr_z` 参数（代码中的 `qr_target.z`）。

- 移动到任务点状态（fsm_state = 3）
- 悬停状态（fsm_state = 5）
- 返回出生点状态（fsm_state = 6）

  以上三个状态详见 `tutorial_basic/vel_control_node` 代码解析的移动和悬停状态章节。

- 投放状态（fsm_state = 4）

```C++
{
    std_msgs::Bool catapult_msg;
    catapult_msg.data = true;
    catapult_pub.publish(catapult_msg);
    twist.twist = get_pid_vel(deliver_target);
    fsm_state = 5;  // goto hover state
    last_srv_request = ros::Time::now();
}
```

  本状态的代码使用一对大括号括起来（此处 1、8 行），这是因为本状态声明了 `catapult_msg` 变量，而 switch 语句在 case 后如果不 break，还会继续执行下一个 case，如果任由变量在 case 中声明，每个变量的作用域是模糊不清的，因此这样的写法是非法的。而使用大扩号就能将 `catapult_msg` 变量的作用域限制在这个 case 分支内。

  本状态只进行三件事：

  1. 向投放装置驱动发送开启仓位消息（源 180~182，此处 2~4 行）。
  2. 继续使用 PID 控制无人机速度（源 183 行，此处 5 行）。
  3. 切换到悬停状态（源 184~185 行，此处 6~7 行）。

  因此每次进入时总是只执行一次就切换到悬停状态。

- 降落状态（fsm_state = 7）

  详见 `tutorial_basic/takeoff_node` 代码解析的降落状态章节。

tutorial_all

launch 文件

- run_craic_task.launch: 启动完整比赛任务（不钻圈）策略节点，节点说明见后文。**请注意运行前务必测试并调整位于 `tutorial_navigation/param` 的 move_base 导航参数和位于 `tutorial_all/param` 的两个霍夫圆识别参数。**

- 参数:
  - working_altitude: 无人机的工作高度，默认为 `1.0`。
  - yolo_weights_path: YoloV5 目标检测模型的位置，仅当 `run_yolo_detector` 为 `true` 时有效，默认为`$(find tutorial_vision)/model/example.pt`。
  - pubImg: 霍夫圆检测节点是否将检测结果以图像形式输出到话题，其中投放点检测结果会发布到`"deliver_detect_result_img"`，起降点检测结果会发布到`"parking_detect_result_img"`，默认为 `false`。
  - run_all_dependence: 是否运行所有依赖节点，默认为 `true`。
  - run_mavros: 是否运行 mavros，默认为`"$(arg run_all_dependence)`。
  - run_move_base: 是否运行 move_base，默认为`"$(arg run_all_dependence)`。
  - run_fast_lio: 是否运行 MID360 雷达驱动和 FAST_LIO 定位算法，默认为`"$(arg run_all_dependence)`。
  - run_catapult_driver: 是否运行投放装置驱动，默认为`"$(arg run_all_dependence)`。
  - run_camera_driver: 是否运行下视相机驱动，默认为`"$(arg run_all_dependence)`。
  - run_deliver_detector: 是否运行投放点霍夫圆检测节点，默认为`"$(arg run_all_dependence)`。
  - run_parking_detector: 是否运行起降点霍夫圆检测节点，默认为`"$(arg run_all_dependence)`。
  - run_qr_detector: 是否运行二维码识别节点，默认为`"$(arg run_all_dependence)`。
  - run_yolo_detector: 是否运行 YoloV5 目标检测节点，默认为`"$(arg run_all_dependence)`。
- 代码解析:

注意到 27~38 行:

```XML
<node pkg="tutorial_vision"
type="hough_circle_detector.py" name="deliver_detector"
if="$(arg run_deliver_detector)">
    <rosparam file="$(find
```

```
tutorial_all)/param/deliver_detect_params.yaml"
command="load"/>
    <param name="pubImg" value="$(arg pubImg)"/>
    <remap from="circle_detect_result"
to="deliver_detect_result"/>
    <remap from="circle_detect_result_img"
to="deliver_detect_result_img"/>
</node>
<node pkg="tutorial_vision"
type="hough_circle_detector.py" name="parking_detector"
if="$(arg run_parking_detector)">
    <rosparam file="$(find
tutorial_all)/param/parking_detect_params.yaml"
command="load"/>
    <param name="pubImg" value="$(arg pubImg)"/>
    <remap from="circle_detect_result"
to="parking_detect_result"/>
    <remap from="circle_detect_result_img"
to="parking_detect_result_img"/>
</node>
```

这两个 node 标签分别用来启动投放点霍夫圆检测节点和起降点霍夫圆检测节点，它们都是 `hough_circle_detector.py` 这个程序（通过 `type` 属性得到），但由于 ROS 要求任意两个 ROS 节点进程不能使用同一个节点名，所以这里令它们的 `name` 标签取不同值，launch 文件中的 `name` 标签会覆盖调用 `ros::init` 或 `rospy.init_node` 时提供的节点名。

`rosparam` 标签用于从 yaml 文件中读取参数，该特性在启动 move_base 时已经用到。

节点

- run_craic_task_node: 完成除钻圈外，所有比赛任务的节点，由于钻圈的难点在于调参，所以该节点不提供钻圈的具体策略。
    - 参数:
        - working_altitude: 无人机工作高度，默认为 `1.0`。
    - 代码解析:
        - 节点初始化，话题订阅，创建话题发布句柄和服务调用句柄，等待 FCU 连接部分

            见 `tutorial_basic/takeoff_node` 代码解析第一部分。本节点额外订阅和发布的话题见以上所有策略节点的代码解析，话题订阅章节。与前

述节点不一样的是，投放装置控制话题的发布句柄被放在了一个 `vector` 内：

```
C++
std::vector<ros::Publisher> catapult_pubs = {
    nh.advertise<std_msgs::Bool>("servo/back_1", 1),
    nh.advertise<std_msgs::Bool>("servo/front", 1),
    nh.advertise<std_msgs::Bool>("servo/back_2", 1)
};
```

这么做的好处是我们可以直接使用“已投放货物数量”变量（后文的 `posted_object`）来确定下一次投放时应该打开哪一个仓位，减少代码的冗余。

- 参数读取部分

  详见 `tutorial_basic/vel_control_node` 代码解析的参数读取部分章节。

- 状态机外部

```
C++
geometry_msgs::Point qr_position;
qr_position.x = 1.8;
qr_position.y = 0.0;
qr_position.z = working_altitude;
geometry_msgs::Point deliver_position[4];
deliver_position[0].x = 1.8;
deliver_position[0].y = 1.6;
deliver_position[0].z = working_altitude;
deliver_position[1].x = 3.6;
deliver_position[1].y = 1.6;
deliver_position[1].z = working_altitude;
deliver_position[2].x = 3.6;
deliver_position[2].y = -1.6;
deliver_position[2].z = working_altitude;
deliver_position[3].x = 1.8;
deliver_position[3].y = -1.6;
deliver_position[3].z = working_altitude;
geometry_msgs::Point special_deliver_position;
special_deliver_position.x = 6.0;
special_deliver_position.y = 1.0;
special_deliver_position.z = working_altitude;
geometry_msgs::Point left_land_position,
right_land_position;
```

```
left_land_position.x = 0.0;
left_land_position.y = 1.6;
left_land_position.z = working_altitude;
right_land_position.x = 0.0;
right_land_position.y = -1.6;
right_land_position.z = working_altitude;

int fsm_state = 0;
int checking_deliver_point = 0;
int posted_object = 0;
std::unordered_set<std::string> post_target;
std::string land_target;
ros::Time last_srv_request = ros::Time::now();
```

1~28 行定义了一些根据规则得到的点位。

状态机外部声明了一些变量，`checking_deliver_point` 指代的是当前正在检查的投放点，因为投放点总共有 4 个，所以该变量的有效取值有 `0`、`1`、`2` 和 `3`。`posted_object` 指代的是已经投放出去的货物的数量，由于货物有 3 个，所以该变量的有效值有 `0`、`1` 和 `2`。`checking_deliver_point` 可以放在 `deliver_position` 的索引中用来控制无人机前往哪一个投放点，`posted_object` 可以放在 `catapult_pubs` 的索引中用来控制下一次投放打开的仓位。

`post_target` 是识别二维码后，用来存放投放目标的容器，不会使用 STL 容器也不要紧，只需要知道 `std::unordered_set` 是一个可以无序存放任意个不重复的元素，并且可以用 `count` 方法在 `O(1)`时间内判断某元素是否位于这个容器中的“数组”就可以了，之后的`<std::string>` 就是之前介绍 `ros::NodeHandle::subscribe` 时提到的模板语法，表示该容器中元素的类型。

`land_target` 也在识别二维码后赋值，表示无人机最终的降落位置（left 或 right）。

- 切换 OFFBOARD 状态（fsm_state = 0）

  详见 `tutorial_basic/takeoff_node` 代码解析的切换 OFFBOARD 状态章节。

- 解锁状态（fsm_state = 1）

  详见 `tutorial_basic/takeoff_node` 代码解析的解锁状态章节。

- 起飞状态（fsm_state = 2）

  详见 `tutorial_basic/takeoff_node` 代码解析的起飞状态章节。在本

节点中，状态机跳出起飞状态的临界海拔取决于`working_altitude`参数（代码中的`working_altitude`变量）。

- 识别二维码状态（fsm_state = 3）

```
C++
if (ros::Time::now() - last_srv_request >
ros::Duration(1.0) &&
        qr_result.header.stamp > last_srv_request &&
        qr_result.data.size() > 0) {
    std::cout << "\033[34mQR Detect: " <<
qr_result.data[0] << "\033[0m" << std::endl;
    analyseQrMessage(qr_result.data[0], post_target,
land_target);
    fsm_state = 4;  // goto check deliver point state
    std::cout << "\033[32mReached Check Deliver Point
State.\033[0m" << std::endl;
    last_srv_request = ros::Time::now();
} else if (ros::Time::now() - last_srv_request >
ros::Duration(5000.0)) {  // Timeout, detect failed
    land_target = "left";
    fsm_state = 7;  // goto navigate to special sign
state
    std::cout << "\033[32mReached Navigate to Special
Sign State.\033[0m" << std::endl;
} else {
    if
(getLengthBetweenPoints(current_pose.pose.position,
qr_position) > 0.3) {
        last_srv_request = ros::Time::now();
    }
    twist.twist = get_pid_vel(qr_position);
}
```

状态转换条件:

1. 当满足以下所有条件:
    a. “无人机距离二维码扫描点的距离小于`0.3`”这一条件（源 314~316，此处 14~16 行）连续满足超过`1`秒（源 301，此处 1 行）。
    b. 最近一条二维码识别消息发布于条件 1-a 开始计时以后（源 302，此处 2 行）。
    c. 二维码识别消息中存在有效的结果（源 303，此处 3 行）。

则处理二维码识别结果（源 305，此处 5 行），并切换到检查投放点状态（源 306，此处 6 行）。

2. 当条件 1-a 已经满足超过 `5000.0` 秒（源 309，此处 9 行），设降落点为 left（源 310，此处 10 行），并切换到导航到特殊投放靶状态（源 311，此处 11 行）。这一分支是用来保证即使意外发生，无人机依然能完成后续任务，不至于因超时而无分的，正式比赛时可以减小等待时间到一合理值。

状态行为：PID 控制无人机固定在二维码扫描点（源 317，此处 17 行）。

二维码识别结果处理函数的定义如下所示：

```
C++
void analyseQrMessage(const std::string &str,
std::unordered_set<std::string> &post_target,
std::string &land_target) {
    std::vector<std::string> split;
    boost::split(split, str, boost::is_any_of(","),
boost::token_compress_on);
    post_target.insert(split[0]);
    post_target.insert(split[1]);
    land_target = split[2];
}
```

原理就是使用 `boost::split` 函数在英文逗号位置分割开二维码识别结果字符串，然后把分成的三个子字符串放在正确的位置上。

- 检查投放点状态（fsm_state = 4）

```
C++
if (ros::Time::now() - last_srv_request >
ros::Duration(1.0) &&
        sign_result.header.stamp > last_srv_request &&
        sign_result.data.size() > 0) {  // detect
succeeded
    if (post_target.count(sign_result.data[0])) {  //
this sign is required to be delivered
        fsm_state = 5;  // goto deliver state
        std::cout << "\033[32mReached Deliver
State.\033[0m" << std::endl;
        last_srv_request = ros::Time::now();
    } else {  // this sign is not required to be
delivered
```

```
        checking_deliver_point++;
        if (checking_deliver_point >= 4) {
            fsm_state = 7;  // goto navigate to special
sign state
            std::cout << "\033[32mReached Navigate to
Special Sign State.\033[0m" << std::endl;
        }
        last_srv_request = ros::Time::now();
    }
} else if (ros::Time::now() - last_srv_request >
ros::Duration(5000.0)) {  // Timeout, detect failed
    checking_deliver_point++;
    if (checking_deliver_point >= 4) {
        fsm_state = 7;  // goto navigate to special
sign state
        std::cout << "\033[32mReached Navigate to
Special Sign State.\033[0m" << std::endl;
    }
    last_srv_request = ros::Time::now();
} else {
    if
(getLengthBetweenPoints(current_pose.pose.position,
deliver_position[checking_deliver_point]) > 0.3) {
        last_srv_request = ros::Time::now();
    }
    twist.twist =
get_pid_vel(deliver_position[checking_deliver_point]);
}
```

状态转换条件:

1. 当满足以下所有条件:

   a. “无人机距离当前投放点的距离小于`0.3`”这一条件（源 344~346，此处 24~26 行）连续满足超过`1`秒（源 321，此处 1 行）。

   b. 最近一条投放点识别消息发布于条件 1-a 开始计时以后（源 322，此处 2 行）。

   c. 投放点识别消息中存在有效的结果（源 323，此处 3 行）。

   则判断:

   i. 如果当前投放点是二维码要求投放的投放点（源 324，此处 4 行），则跳转到投放状态（源 325、327，此处 5、7 行）

ii. 否则：如果当前投放点已经是最后一个投放点（源 330，此处 10 行），跳转到导航到特殊投放靶状态（源 331，此处 11 行），反之检查下一个投放点（源 329，此处 9 行）。

2. 当条件 1-a 已经满足超过 `5000.0` 秒（源 336，此处 16 行），认为该投放点不是二维码要求的投放点，执行上述 1-ii 策略（源 337~342，此处 17~22 行）。这一分支是用来保证即使无人机难以识别投放点上的物体，无人机依然能完成后续任务，不至于因超时而无分的，正式比赛时可以减小等待时间到一合理值。

状态行为：PID 控制无人机固定在二维码扫描点（源 347，此处 27 行）。

- 投放状态（fsm_state = 5）

```C++
if (ros::Time::now() - last_srv_request > ros::Duration(4.0)) {  // release catapult
    std_msgs::Bool catapult_msg;
    catapult_msg.data = true;
    catapult_pubs[posted_object].publish(catapult_msg);
    fsm_state = 6;  // goto wait for object drop state
    std::cout << "\033[32mReached Wait for Object Drop State.\033[0m" << std::endl;
    last_srv_request = ros::Time::now();
} else {
    if (deliver_detect_result.header.stamp > last_srv_request - ros::Duration(1.0)) {
        geometry_msgs::Point err;
        err.x = deliver_detect_result.width / 2.0 - deliver_detect_result.circles[0].center_x;
        err.y = deliver_detect_result.height / 2.0 - deliver_detect_result.circles[0].center_y;
        twist.twist = get_pix_pid_vel(err);
    }
    twist.twist.linear.z = std::max(-0.5, std::min(0.5, working_altitude - current_pose.pose.position.z));
}
```

状态转换条件：当状态机进入该状态超过 `4.0` 秒（源 327、351，此处检查投放点状态 7，投放状态 1 行），打开当前仓位（源 352~354，此处 2~4 行），跳转到投放后悬停状态（源 355、357，此处 5、7 行）。

状态行为：

1. 如果当前收到最新的霍夫圆检测结果是在进入该状态前 `1` 秒内，

或进入该状态之后才收到的（源 359，此处 9 行）（这是为了保证 `deliver_detect_result` 变量内存储的确实是当前飞机正下方投放点的识别结果，而不是之前已经经过的投放点的识别结果），则 x 和 y 方向速度按下方投放点的霍夫圆圆心做 PID 控制（源 360~363，此处 10~13 行），否则 x 和 y 方向速度不做控制。

2. z 方向速度以 `working_altitude` 为目标做 P 控制（源 365，此处 15 行）（偷懒，追求精度可换成 PID 控制）。

`get_pix_pid_vel` 函数的定义见 `tutorial_vision` 包内 `accurately_land_node` 节点。

- 投放后悬停状态（fsm_state = 6）

```C++
if (ros::Time::now() - last_srv_request > ros::Duration(1.0)) {
    posted_object++;
    checking_deliver_point++;
    // if 2 blocks are both delivered or 4 deliver point are all checked,
    //     goto navigate to special sign state,
    // else go back to check deliver point state to check the next.
    if (posted_object == 2 || checking_deliver_point == 4) {
        std::cout << "\033[32mReached Navigate to Special Sign State.\033[0m" << std::endl;
        fsm_state = 7;
    } else {
        std::cout << "\033[32mReached Check Deliver Point State.\033[0m" << std::endl;
        fsm_state = 4;
        last_srv_request = ros::Time::now();
    }
}
```

状态转换条件：当状态机进入该状态超过 `1.0` 秒（源 357、369，此处投放状态 7，投放后悬停状态 1 行），`posted_object` 和 `checking_deliver_point` 均自增 1（源 370~371，此处 2~3 行）（这将使得下一次切换到检查投放点状态时，无人机会飞往下一个投放点，且下一次无人机进行投放时，会打开下一个仓位）。然后如果满足以下两个条件之一（源 375，此处 7 行）：

1. `posted_object == 2`，这意味着只剩下一个需要投放到特殊靶

的货物，剩下的投放点都是不满足二维码要求的。

2. `checking_deliver_point == 4`，这意味着 4 个投放点都已经检查一遍了，如果此时 `posted_object == 2` 不成立，则一定是之前出现了误识别，但到这一步也只能放弃该分值了。

那么跳转到导航到特殊靶状态（源 377，此处 9 行），反之跳转到检查投放点状态（源 380~381 行，此处 12~13 行），继续检查下一个投放点。

状态行为：什么也不做，不使用 PID 控制是为了防止实际投放位置和理论投放位置偏差太大，切换到该状态的瞬间使用 PID 控制会导致无人机拨动货物，造成投放位置偏离投放点圆心。

- 导航到特殊靶状态（fsm_state = 7）

```C++
{
    geometry_msgs::PoseStamped move_base_msg;
    move_base_msg.header.frame_id = "map";
    move_base_msg.pose.position =
special_deliver_position;
    move_base_msg.pose.orientation.w = -1.0;
    goal_pub.publish(move_base_msg);
    fsm_state = 8;  // goto wait for navigation mission
state
    std::cout << "\033[32mReached Wait for Navigation
Mission State.\033[0m" << std::endl;
}
```

本状态只发布导航任务（源 387~391，此处 2~6 行），之后直接跳转到等待导航任务完成状态（源 392，此处 7 行）。关于发布导航任务的细节，详见 `tutorial_navigation` 包内 `navigation_node` 节点。

- 等待导航任务完成状态（fsm_state = 8）

详见 `tutorial_navigation/navigation_node` 代码解析的导航状态章节。

- 投放特殊靶状态（fsm_state = 9）

该状态的行为与前文投放状态（fsm_state = 5）的行为类似，这里不过多赘述。

- 投放特殊靶后悬停状态（fsm_state = 10）

该状态的行为与前文投放后悬停状态（fsm_state = 6）的行为类似，这

里不过多赘述。唯一的区别在于该状态不再需要操作`posted_object`和`checking_deliver_point`两个变量（因为这两个变量在后续都没有使用到，此时已失去意义），且离开状态时根据`land_target`的值，发布导航命令和选择跳转到等待抵达左降落点状态还是等待抵达右降落点状态。

- 等待抵达左降落点状态（fsm_state = 11）
- 等待抵达右降落点状态（fsm_state = 12）

  详见`tutorial_navigation/navigation_node`代码解析的导航状态章节，两状态的区别在于无人机的导航目标点不同。

- 精准降落状态（fsm_state = 13）

```C++
if (current_pose.pose.position.z < 0.1) {
    fsm_state = 100;  // goto land state
    std::cout << "\033[32mReached Land State.\033[0m"
<< std::endl;
} else if (parking_detect_result.header.stamp >
last_srv_request &&
parking_detect_result.circles.size() > 0) {
    geometry_msgs::Point err;
    err.x = parking_detect_result.width / 2.0 -
parking_detect_result.circles[0].center_x;
    err.y = parking_detect_result.height / 2.0 -
parking_detect_result.circles[0].center_y;
    twist.twist = get_pix_pid_vel(err);
    twist.twist.linear.z = -0.1;
}
```

  状态转换条件：当无人机的高度小于`0.1`时（源 471，此处 1 行），跳转到降落状态（源 472，此处 2 行）。

  状态行为：若当前下视相机视野内有霍夫圆（既判断消息发出时间也判断消息有效性）（源 474，此处 4 行），对准霍夫圆的圆心进行 x 和 y 方向的 PID 控制，并以`-0.1`的 z 方向速度降落（源 475~479，此处 5~9 行）。

  相比`tutorial_vision/accurately_land_node`，将电机停转逻辑放在了一个单独的状态里。同时删除了看不见圆时的固定速度下降逻辑，降落过程中如果霍夫圆短暂消失，策略依然会使用最近一次霍夫圆的位置进行 PID 控制。

- 降落状态（fsm_state = 100）

详见`tutorial_basic/takeoff_node`代码解析的降落状态章节。

- 空状态（default）

```
C++
if (fsm_state != -1) {
    ROS_FATAL("FATAL ERROR: FSM reaches an invalid
state: %d, emergency landing.", fsm_state);
    fsm_state = 100;
}
```

考虑到有选手直接修改本文件用作自己队伍的比赛代码，空状态相比之前几个节点，增加了当状态机进入不存在的状态时，紧急迫降的功能。降落状态会在降落完成后将`fsm_state`设为`-1`，如果`fsm_state`的值为一不存在的状态且不是`-1`，那么大概率是编码错误导致，此时为了安全考量，切换到降落状态。

常见问题及解决方案

1. 非编译、非执行问题

(1)

Q：用户使用`nvcc -V`提示“找不到 nvcc”

A：首先询问用户是否更换 zsh 等其它 shell，如果确已更换，告知其在 shell 的初始化文件中添加 cuda 环境变量，以 zsh 为例，在`~/.zshrc`中添加

```
Bash
export PATH=/usr/local/cuda/bin:$PATH
export LD_LIBRARY_PATH=/usr/local/cuda/lib64:$LD_LIBRARY_PATH
```

如果用户没有更换 shell，首先检查`~/.bashrc`中是否含有上述语句，如果不含有，令其添加后再尝试；如果含有，参考《软件系统配置方法简述》第四步安装 nvidia-jetpack。

---

(2)

Q：用户称在使用 pip 安装某些包时出现了错误。

A：如果是网络问题，尝试使用 pip 国内镜像源解决，如果非网络问题，尝试执行

`python -m pip install --upgrade pip && pip install --upgrade setuptools` 后再安装。

---

(3)

Q：用户想要进入 eMMC 内安装的系统

A：使用下列指令可重启并进入 eMMC 内的系统

```
Bash
sudo mount /dev/mmcblk0p1 /mnt
sudo mv /mnt/etc/setssdroot.conf /mnt/etc/setssdroot.conf.bak
sudo reboot
```

在 eMMC 内的系统上使用下列逆操作指令可重启并返回 SSD 内的系统

```
Bash
sudo mv /etc/setssdroot.conf.bak /etc/setssdroot.conf
sudo reboot
```

---

(4)

Q：用户单买了一个投放装置，询问如何驱动投放装置。

A：有通过 GPIO 驱动和通过 PX6C 驱动两种方式。

- 通过 GPIO 驱动（不适用 Nano，这是因为 Nano 只有两个 GPIO 口支持 PWM，无法控制三个舵机，但 Nano 仍可以基于 I2C 与 PCA9685 通信，由 PCA9685 控制舵机。考虑到产品不自带 PCA9685，故认为用户如果自备了 PCA9685，那么就应该具备使用 PCA9685 的相关知识，所以以下不提供关于 PCA9685 的使用细节。）

1. 使用 Jetson-IO 启用 PWM 功能：在控制台中执行：

```
Bash
sudo /opt/nvidia/jetson-io/jetson-io.py
```

按键盘方向键依次选中 `Configure Jetson 40pin Header > Configure header pins manually` 选项并回车。

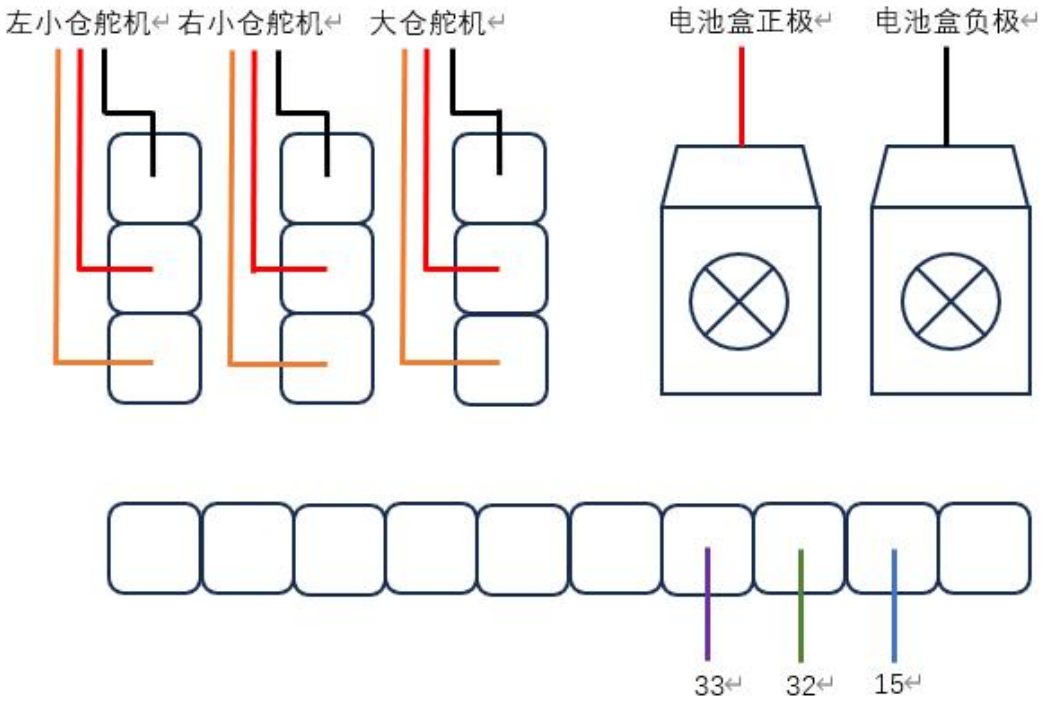
然后选择三个以 `pwm` 开头的选项，按回车键使前方`[ ]`内出现`*`号，然后选中 `Back` 并回车。

最后，选中 `Save pin changes > Save and Reboot to configure pins` 选项并回车。

2. 安装 JetsonGPIO，执行：

```Bash
git clone https://github.com/pjueon/JetsonGPIO.git
mkdir JetsonGPIO/build
cd JetsonGPIO/build
cmake ..
make -j
sudo make install
```

3. 接线，按如下方式连接舵机、电池盒、投放装置上的分线板和上位机（15、32、33 三个数字是 BOARD 模式下三个 GPIO 口的编号），然后在电池盒中安装三节 7 号电池。



4. 开启上位机，编译 **GPIO 版本**的 `tutorial_catapult` 包，使用以下指令启动投放装置驱动。

```Bash
roslaunch tutorial_catapult catapult_driver.launch
```

之后向`/servo/front`、`/servo/back_1`、`/servo/back_2`三个话题中发送`std_msgs/Bool`类型的消息就能控制投放装置了，其中`data`为`true`表示打开对应投放仓，为`false`表示关闭对应投放仓。

- 通过 PX6C 驱动

1. 使用 QGroundControl 调整飞控与 PWM 输出相关的参数（这一版参数可以找姚

明锐要到）。

2. 按《通过 GPIO 驱动》章节中第三步的方法接线，但 10-pin 口不再与 GPIO 对应引脚相连，而是找一条 10-pin 的 GH1.25 转杜邦线转接线，连接 PX6C 上的 I/O PWM OUT 和投放装置分线板，以整排杜邦线在连接时不发生扭转为正确方向。

3. 开启上位机，编译**默认版本**的 `tutorial_catapult` 包，使用以下指令启动投放装置驱动。

```
Bash
roslaunch tutorial_catapult catapult_driver.launch
```

之后**首先需要使用代码起飞飞机（即 OFFBOARD+ARM，否则即使发送话题消息，PX6C 也不会输出 PWM 控制信号）**，再向`/servo/front`、`/servo/back_1`、`/servo/back_2` 三个话题中发送 `std_msgs/Bool` 类型的消息就能控制投放装置了，其中 `data` 为 `true` 表示打开对应投放仓，为 `false` 表示关闭对应投放仓。

---

(5)

Q：没有使用本代飞机系统镜像的用户在尝试使用 GPIO 控制舵机时，发现调用 `GPIO::PWM::ChangeDutyCycle` 函数后，对应 GPIO 口并没有输出 PWM 信号。

A：令其使用 Jetson-IO 启用相应 GPIO 的 PWM 功能。另外，如果用户的上位机不是 NX 而是 Jetson Orin 等其它系列，提醒其修改 `tutorial_catapult/src/actuator_driver.cpp` 内的 GPIO 端口号，如果用户使用的是 Jetson Nano，告知其 Jetson Nano 只有两个 GPIO 口支持 PWM 输出，无法控制三个舵机，让其使用飞控控制方法或者使用 PCA9685。

2. 编译问题

(1)

Q：用户在使用 `cmake` 或 `catkin_make` 时，出现了含义下列内容的报错。

```
Plain Text
Could not find a package configuration file provided by "xxx" with
any of
  the following names:

    xxxConfig.cmake
    xxx-config.cmake
```

A：首先检查依赖是否存在但 CMake 没有找到，确认：

a. xxx 已在 CMakeLists.txt 中被 `find_package`，注意如果 `find_package` 不

加 `REQUIRED`，则失败时不会立即终止编译，而是会发出警告。

b. xxx 已在 package.xml 中被 `depend`。

c. CMakeLists.txt 中对目标进行了正确的 `target_link_libraries` 和 `add_dependencies` 操作。

如果以上条件均满足，说明用户缺少编译需要的依赖，如果 xxx 是 ros 功能包，修复方法同常见问题及解决方案(2)。

如果 xxx 不是 ros 功能包，首先可以尝试以下指令安装依赖：

```
Bash
sudo apt install xxx
sudo apt install libxxx
sudo apt install libxxx-dev  # 是这条指令的概率最大
pip install xxx
pip install pyxxx
```

如果 apt 和 pip 找不到上述可安装依赖，则可能需要编码编译安装，需要用户在 github 等开源托管平台上找到相应依赖的源码，克隆至主目录并编译安装，安装时应优先参考仓库 README 的指导。

大部分基于 CMake 构建的仓库都可以使用以下方法编译安装，以仓库名为 xxx 为例，在 xxx 仓库文件夹内执行：

```
Bash
mkdir build
cd build
cmake ..
make -j
sudo make install
```

---

(2)

Q：用户在使用 `cmake` 或 `catkin_make` 时，`#include` 语句报出了类似下面的错误：

```
Plain Text
fatal error: xxx.h: No such file or directory
    x | #include <xxx.h>
      |          ^~~~~
```

A：这是因为`#include` 时，编译器没有找到对应的头文件，首先询问该头文件是用户自己编写的头文件，自己编写的消息类型自动生成的头文件，还是所调用库带有的头文件或者所调用 ros 库带有的消息类型。

- 如果是自己编写的头文件

  确认：

1. 头文件所在目录已被包含在 `include_directories` 内，如使用 `catkin_create_pkg` 创建的功能包自动创建的 CMakeLists.txt 内的 `include_directories` 通常是这样编写的：

```CMake
include_directories(
    ${catkin_INCLUDE_DIRS}
    # include
)
```

   那么此处 `include` 被注释就可能是导致问题的原因。

2. 引用路径正确，如 `include_directories` 内加入了 `include`，头文件放在了 `include/example_pkg/example.h`，而代码中写的是`#include "example.h"`，那么这就是一种引用路径错误，应将代码改为`#include "example_pkg/example.h"`。

- 如果是自己编写的消息类型

  确认：

1. 重新 `catkin_make` 编译一次，问题依旧。
2. 定义消息类型的功能包已被编译，且所在工作空间被正确 source（没有发生非编译、非执行问题(2)所描述的 source 顺序问题）。
3. 定义消息类型的功能包的 CMakeLists.txt 编写正确，重点检查是否先后调用了 `add_message_files`、`generate_messages` 和 `catkin_package`，`find_package` 是否含有 `message_generation`。
4. 调用消息类型的功能包的 `find_package` 内含有定义消息类型的功能包，`include_directories` 内含有`${catkin_INCLUDE_DIRS}`。

- 如果是所调用库带有的头文件

  确认：

1. 被调用库已被成功 `find_package`，注意如下写法的 `find_package` 在失败时不会报错，而仅是给出警告：

```CMake
find_package(<库名>)
```

想要在 `find_package` 失败时立即停止编译并报错，正确写法是

```CMake
find_package(<库名> REQUIRED)
```

2. `include_directories` 内含有库的头文件目录，常见写法是`${<库名>_INCLUDE_DIRS}`，如`${OpenCV_INCLUDE_DIRS}`。
3. 被调用库正确安装，不存在版本兼容问题。

- 如果是所调用 ros 库带有的消息类型

确认：

1. 调用消息类型的功能包的 `find_package` 内含有所调用消息类型的功能包，`include_directories` 内含有`${catkin_INCLUDE_DIRS}`。
2. 所调用 ros 库使用 `apt` 正确安装，或使用源码正确编译，且所在工作空间被正确 source（没有发生非编译、非执行问题(2)所描述的 source 顺序问题）。

3. 执行问题

(1)

Q：用户在 `roslaunch` 和 `rosrun` 时提示找不到功能包，报错如下：

```Plain Text
rosrun:
[rospack] Error: package 'xxx' not found

roslaunch:
RLException: [xxx.launch] is neither a launch file in package
[xxx] nor is [xxx] a launch file name
The traceback for the exception was written to the log file
```

A：首先询问 xxx 包是否是用户自己的功能包，如果是，询问用户是否已经使用 catkin_make 编译（即使使用 python，刚创建的功能包也需要编译一次），如果已编译，询问是否在`.bashrc` 中添加了 source 语句，如果没有添加提醒其添加或手动 source；如果添加了，且 source 后当前工作空间的节点可以使用，但其它工作空间的节点使用不了了，告知其工作空间的 source 有顺序，必须“后创建的先 source，先创建的后 source”，可以通过解决 source 语句的顺序或者选择在每个 source 语句后添加一个`--extend` 参数解决。

如果 xxx 包不是用户自己的功能包，首先尝试使用如下指令安装功能包：

```
Bash
sudo apt install ros-noetic-xxx
```

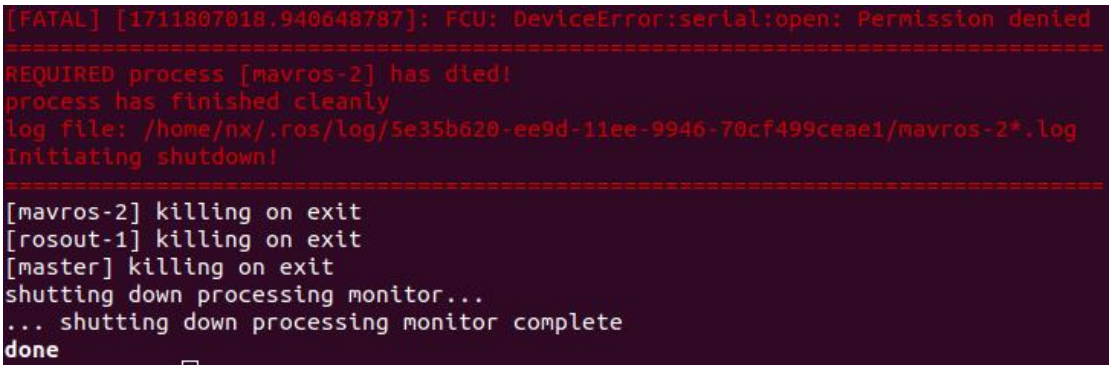
其中包名 xxx 的下划线在该指令中要换成短横线，比如用户尝试运行的功能包为“realsense2_camera”，则需要使用 apt 安装的软件名为“ros-noetic-realsense2-camera”。

如果 apt 找不到上述名称的包，则需要用户自行去 github 等开源托管平台上找到该功能包，自行克隆到 `catkin_ws/src` 内并使用 `catkin_make` 编译。

---

(2)

Q：用户在使用 `roslaunch tutorial_basic mavros.launch` 启动 mavros 时遇到了以下报错:



A：两种解决方法

1. 每次先执行 `sudo chmod 777 /dev/ttyACM0`，再启动 mavros。
2. 一劳永逸，执行 `sudo usermod -aG dialout $USER`，之后启动 mavros 就不存在这个问题了。