Lab2

0. 封面

- **实验题目**：实验2 小车单颜色智能搬运
- **院系**：软件学院
- **设计者**：

学号	姓名
202100300063	李彦浩
202100300340	黄幸兒
202100300078	李世会
202100161049	徐芃恺

- **指导教师**：李新

- **实验时间**：2023-11-21(18:00-20:00)

1. 实验内容

1.1. 实验目标

阅读小车前后左右移动、旋转、识别颜色等功能相关开发文档，开发一个能识别指定颜色积木，并能将积木搬运到指定位置的小车控制程序。

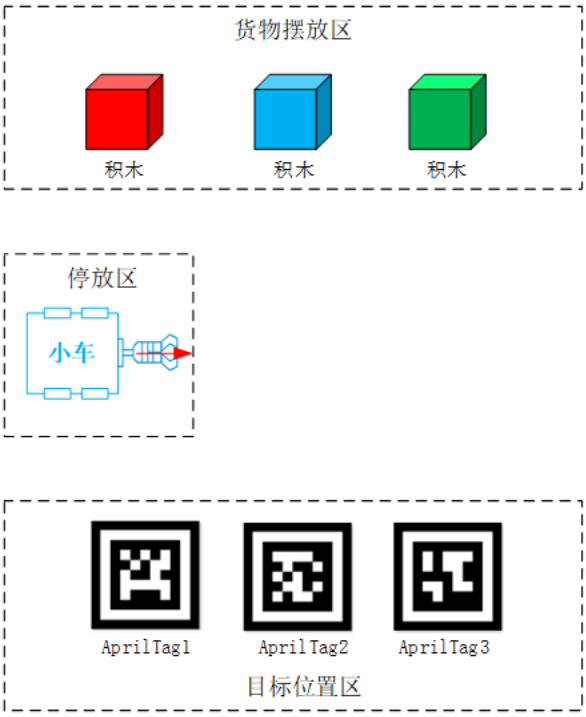


图 实验场地示意图

1.2. 实验要求

1. 启动小车程序，将小车从停放区移动到货物摆放区;
2. 在货物摆放区，搜索指定颜色的积木（通过用户指令指定），并夹取该积木
3. 小车夹住该积木，搬运到目标位置区，将红色积木放置于目标位置区域内（目标位置区域用黑色记号笔在地面上标出）
4. 程序开始后，不允许人为移动小车，验证程序的正确性和是否满足时间要求。

2. 需求分析

2.1. 功能性需求分析

1.需要实现启动小车时，小车从停放区行驶到货物摆放区

2.需要实现在货物摆放区，搜索指定颜色的积木（通过用户指令指定的），并夹取积木

3.需要实现小车夹取积木后，搬运到目标放置区，并放置到指定区域（通过识别AprilTag来放置）

2.2. 非功能性需求分析

1.代码可扩展性强，可以根据需求扩展相关功能

2.代码可维护性强，耦合度低，便于维护

3.代码复用性高，控制小车的不同行为动作的代码按模块进行封装，用到时直接调用

4.代码可读性好，采用有意义的变量名、注释和代码结构来提升可读性

5.代码节能性好：代码在运行时消耗小车较少的电量

3. 实验前置知识：ROS系统

HiWonder基于ROS系统进行机器人开发。在实验开始前，我们需要大致了解ROS系统。

3.1. 模块化分工

3.1.1. node

最小的进程单元。一个软件包里有多个可执行文件，一个可执行文件在运行执行之后会变成一个进程，进程在ROS中就叫做node节点。

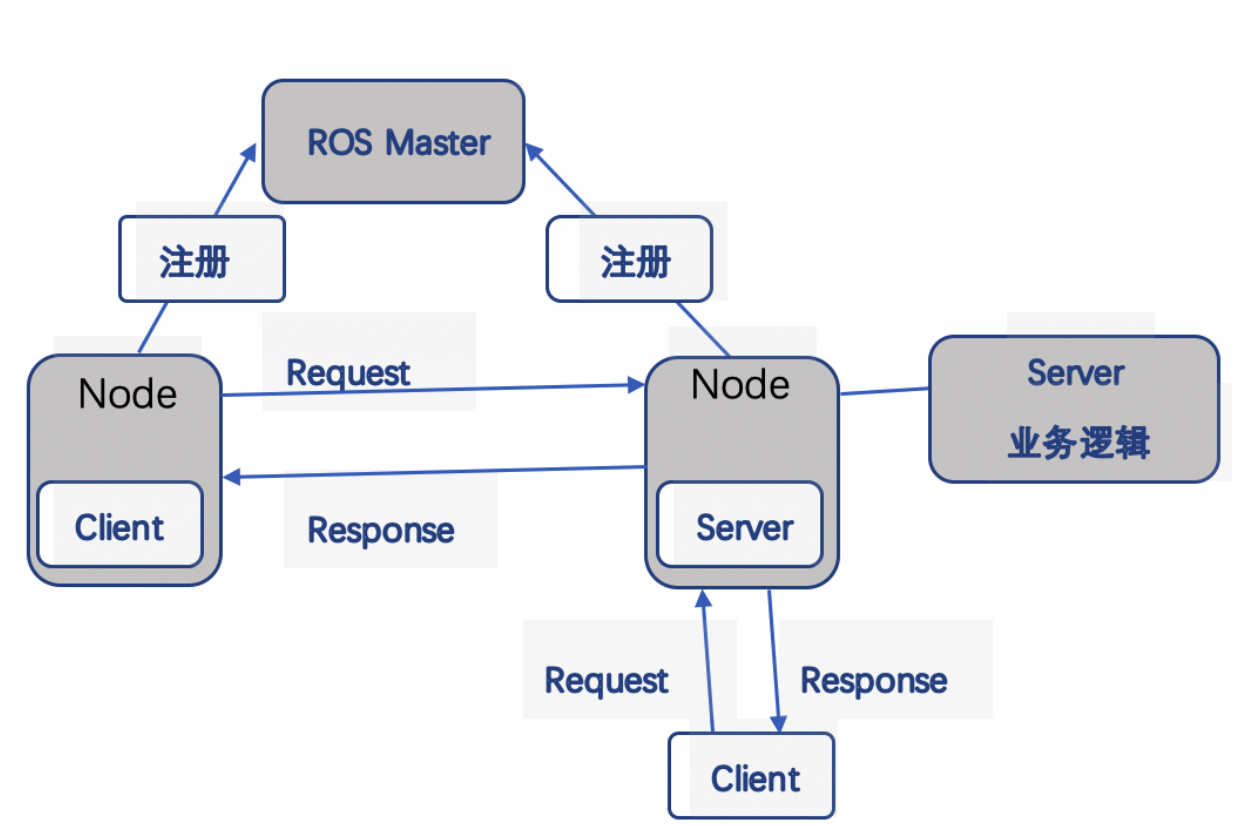
从程序来看，node就是一个可执行文件（C++或Python脚本）被执行；

功能上来说，一个node负责一个独立的功能，很多个node完成机器人的各种操作任务（如一个node控制底盘轮子，一个node驱动摄像头获取图像，一个node驱动激光雷达，一个node传感器信息进行路径规划等）。

3.1.2. Master

节点管理器master，负责调配、管理node，在整个网络通信架构里相当于管理中心，管理着各个node。

node首先要在master进行注册，master将node拉入整个ROS程序中，node与node之间的通信首先由master牵线，之后才可以两两点对点通信。

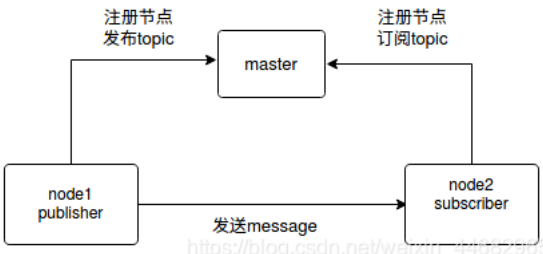


3.2. 通信方式

3.2.1. Topic

遵循发布-订阅模式（观察者模式）

1. publisher节点和subscriber节点到master进行注册
2. publisher发布topic
3. subscriber在master指挥下订阅改topic
4. publisher和subscriber建立单向通信



topic通信属于异步通信，
publisher每发布一个消息之后就会执行下一个动作，不关心消息之后会如何被处理，subscriber只管接受topic，不关心是由谁发送，

publisher和subscriber不存在协同工作，为异步通信。
topic可以同时有多个subscribers，也可以同时有多个publishers

示例

摄像头画面的发布、处理、显示：摄像头拍摄程序是一个node（记作node1），当node1运行启动之后，它作为一个Publisher就开始发布topic，叫做/camera_rgb，是rgb颜色信息，node2是图像处理程序,它订阅了/camera_rgb这个topic，经过master的介绍，node2就能建立和摄像头节点（node1）的连接。node3与node2相同。

通讯源码详解

3.2.1.1. 什么是msg?

msg指.msg文件(short for Message)，这是一种用于ROS系统中的通讯方式。一个.msg文件定义了通讯参数的类型、字段名。

例如：

```
# SetVelocity.msg
float64 velocity
float64 direction
float64 angular
```

也就是说，如果发布-订阅者之间协商好，使用这个消息格式，那么发布者发出一条消息后，接收者收到的消息格式就如同上面一样，字段名也是一样的，我个人就理解成web中的json。

3.2.1.2. 怎样使用msg通讯？

举个例子：

在小车运动的应用中（Lab1写了），我们想要通过发布-订阅模式来发布一条底盘速度的控制信息，例如：

```
set_velocity.publish(0,0,0) # 停止移动
```

那么订阅者怎么接收到呢？

3.2.1.2.1. 从注册开始

首先我们要搞清楚谁是发布者谁是订阅者。现在有两个node:

1. 底盘控制模块chassis_control_node
2. 小车移动模块car_move_node

好了，谁是发布者？谁是订阅者？

显然，前者订阅者，后者发布者。为什么？我们不妨认为发布者是发出命令的一方，订阅者是执行命令的一方，那么以上两个模块谁发出了命令？小车移动模块，它发布命令让底盘按照它的意愿更改速度。那么底盘控制模块就是订阅者。

如何注册呢？rospy提供了函数：

```
set_velocity = rospy.Publisher('/chassis_control/set_velocity', SetVelocity, queue_size=1)
```

1. `'/chassis_control/set_velocity'`：这是一个字符串，表示你要发布消息到的主题名称。在这个案例中，它是一个控制车辆速度的主题。当你的节点发布一个`SetVelocity`类型的消息到这个主题时，所有订阅这个主题的节点都会收到这个消息。
2. `SetVelocity`：这是一个消息类型。在ROS中，消息是在节点之间传递的数据结构。
3. `queue_size=1`：这是一个队列大小参数。在ROS中，发布者和订阅者之间的通信是异步的。这意味着，当你发布一个消息时，它不会立即被所有的订阅者收到。相反，它会被放入一个队列中，然后由ROS在适当的时候分发出去。这个队列的大小就是由`queue_size`参数定义的。在这个例子中，队列的大小是1，这意味着每个消息都会立即被传递给所有的订阅者，而不会等待其他消息被处理。

也就是小车移动模块发布了一个Topic，所有订阅这个Topic都将接收到它之后发布的信息。（当然一个模块可以注册多个Topic)

现在来看底盘控制模块

```
set_velocity_sub = rospy.Subscriber('/chassis_control/set_velocity', SetVelocity, Set_Velocity)
```

1. `'/chassis_control/set_velocity'`：这是一个字符串，表示你要订阅的主题名称。在这个案例中，它是一个控制车辆速度的主题。当有人（例如另一个节点）发布消息到这个主题时，你的订阅者就会收到这个消息。
2. `SetVelocity`：这是一个消息类型。在ROS中，消息是在节点之间传递的数据结构。
3. `Set_Velocity`：这是一个回调函数，当你的订阅者收到一个消息时，这个函数将被调用。这意味着，每当有人发布一个`SetVelocity`类型的消息到`/chassis_control/set_velocity`主题时，这个函数就会被执行。这个函数可能包含处理速度设置的代码，例如更新当前的速度或者执行一些动作。

ok，那么SetVelocity类型是什么？答案在/chassis_control/msg/SetVelocity.msg中

```
# SetVelocity.msg
float64 velocity
float64 direction
float64 angular
```

然而，在chassis_control_mode中，依赖的模块有：

```
from chassis_control.msg import *
```

也就是SetVelocity.msg映射为了一种消息类型。

3.2.1.2.2. 发布一条消息

在我们调用:

```
set_velocity.publish(0,0,0) # 停止移动
```

时，订阅了/chassis_control/set_velocity话题的chassis_control_node将收到这条消息，怎么收到的呢？

```
def get_subscribers(self, event):
        """
        Return an iterator for the subscribers for an event.
        :param event: The event to return subscribers for.
        """
        return iter(self._subscribers.get(event, ()))

def publish(self, event, *args, **kwargs):
        """
        Publish a event and return a list of values returned by its
        subscribers.

        :param event: The event to publish.
        :param args: The positional arguments to pass to the event's
                     subscribers.
        :param kwargs: The keyword arguments to pass to the event's
                       subscribers.
        """
        result = []
        for subscriber in self.get_subscribers(event):
            try:
                value = subscriber(event, *args, **kwargs)
            except Exception:
                logger.exception('Exception during event publication')
                value = None
            result.append(value)
        logger.debug('publish %s: args = %s, kwargs = %s, result = %s',
                     event, args, kwargs, result)
        return result
```

首先先获取订阅了这个Topic的所有订阅者，也就是get_subscribers(event)做的事情，随后逐一向他们发送新消息，对应：

```
value = subscriber(event, *args, **kwargs)
```

将返回值拼接起来，作为结果给publisher。

3.2.1.2.3. 接收消息触发回调函数

在订阅方接到消息后，它的回调函数将生效:

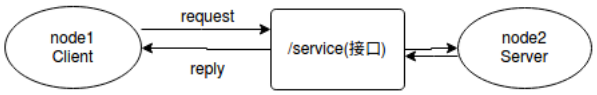
```
def Set_Velocity(msg):
    global th

    velocity = msg.velocity
    direction = msg.direction
    angular = round(msg.angular,2)

    if th is None: # 通过子线程去控制缓变速
        th = Thread(target=slow_velocity,args=(velocity, direction, angular))
        th.setDaemon(True)
        th.start()
```

此处的msg为：

```
{
  velocity:0,
  direction:0,
  angular:0
}
```

3.2.2. Service

service通信是双向的，接收消息同时有反馈。Service包括请求方（Client），应答方/服务提供方（server）。Client发送一个request，server处理，反馈一个reply，/Service类似于Server提供的一个服务接口，一般用string类型来定义service名称（类似于topic）。



service是同步通信方式，Client发布request后会原地等待reply，直到server处理完请求并反馈reply，Client接受到reply才会继续执行，是一种阻塞状态的通信，请求-应答通信模型只有请求时才执行服务，没有频繁的消息传递。

通讯源码详解

3.2.2.1. 什么是srv?

Service使用.srv文件来进行通讯（srv is short for service,对标Topic的.msg文件），内容形如：

```
string data
---
bool success
string message
```

前者代表请求报文，后者代表响应报文。

是有srv嵌套msg的语法糖的，但是没有msg嵌套srv的（也嵌套不进去）

3.2.2.2. 如何注册

3.2.2.2.1. 注册server

在rospy中，创建一个server的的语句形如：

```
set_target_srv = rospy.Service('/visual_patrol/set_target', SetTarget, set_target)
```

1. `'/visual_patro/set_target'` :Service的Topic
2. `SetTarget` :请求响应类型为SetTarget类型，也就是：

```
string data
---
bool success
string message
```

3. `set_target` :回调函数，接到请求后执行该函数。我们可以看一下这个回调函数干了什么

```
def set_target(msg):
    global lock
    global target_en
    global target_color

    rospy.loginfo("%s", msg)
    with lock:
        target_color = msg.data
        led = Led()
        led.index = 0
        led.rgb.r = range_rgb[target_color][2]
        led.rgb.g = range_rgb[target_color][1]
        led.rgb.b = range_rgb[target_color][0]
        rgb_pub.publish(led)
        led.index = 1
        rgb_pub.publish(led)
        rospy.sleep(0.1)
        visual_running = rospy.ServiceProxy('/visual_processing/set_running', SetParam)
        visual_running('line',target_color)
        rospy.sleep(0.1)
        target_en = True

    return [True, 'set_target']
```

忽略逻辑，看他返回了什么

```
    return [True, 'set_target']
```

正好和SetTarget的响应格式相符（bool,string)。

这个例子还有一个很重要的地方，不过我们放在后面再说。

3.2.2.2.2. 注册client

在rospy中，创建一个client的的语句形如：

```
visual_running = rospy.ServiceProxy('/visual_processing/set_running', SetParam)
visual_running('line',target_color)
```

通过一个服务代理，请求对应的服务节点。

1. `'/visual_processing/set_running'` ：请求的接口名称
2. `SetParam` :请求类型

现在只知道请求类型是SetParam了，怎么设置实参呢？

```
visual_running('line',target_color)
```

我们可以看一下SetParam中的请求参数格式是什么

```
string type
string color
---
bool success
string messagev
```

两个string，而target_color也是个string，一会儿举例子的时候会说的。

3.2.2.2.3. 嵌套请求

3.2.2.2.1节里面提到了一个很重要的回调函数：

```
def set_target(msg):
    global lock
    global target_en
    global target_color

    rospy.loginfo("%s", msg)
    with lock:
        target_color = msg.data
        led = Led()
        led.index = 0
        led.rgb.r = range_rgb[target_color][2]
        led.rgb.g = range_rgb[target_color][1]
        led.rgb.b = range_rgb[target_color][0]
        rgb_pub.publish(led)
        led.index = 1
        rgb_pub.publish(led)
        rospy.sleep(0.1)
        visual_running = rospy.ServiceProxy('/visual_processing/set_running', SetParam)
        visual_running('line',target_color)
        rospy.sleep(0.1)
        target_en = True

    return [True, 'set_target']
```

他之所以重要是因为这一行：

```
visual_running = rospy.ServiceProxy('/visual_processing/set_running', SetParam)
```

嵌套请求了。其实这有点像分布式系统之间的各模块调用。

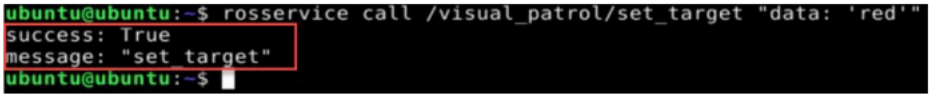
3.2.2.3. 命令行发起请求

之前3.2.2.2.3节提到的请求方式是通过编码来进行请求的，ROS系统还提供了一种方式：通过命令行发起请求。

这其实在`teach→1. 玩法实操→1.1. 追踪玩法实操`中提到过怎么使用命令行来请求一个node向外提供的服务。重温一下：

2) 启动玩法后我们还需要设置参数即选择巡线的颜色，这里以追踪红色为例，输入指令“**rosservice call /visual_patrol/set_target "data: 'red'"**”。

注意：巡线绿色和蓝色可在"data: 'red '"内 red 替换为 green 或 blue。(严格区分大小写)




```
rosservice call request_url **argv
```

1. `request_url`:请求路径
2. `argv`:参数列表

set_target这个接口的请求参数为：

```
string data
---
bool success
string message
```

也就是单个的string。这里相当于是

```
{
  data:'red'
}
```

随后就会执行对应的回调：

```
def set_target(msg):
    global lock
    global target_en
    global target_color

    rospy.loginfo("%s", msg)
    with lock:
        target_color = msg.data
        led = Led()
        led.index = 0
        led.rgb.r = range_rgb[target_color][2]
        led.rgb.g = range_rgb[target_color][1]
        led.rgb.b = range_rgb[target_color][0]
        rgb_pub.publish(led)
        led.index = 1
        rgb_pub.publish(led)
        rospy.sleep(0.1)
        visual_running = rospy.ServiceProxy('/visual_processing/set_running', SetParam)
        visual_running('line',target_color)
        rospy.sleep(0.1)
        target_en = True

    return [True, 'set_target']
```

4. 实验设计

4.1. 总体流程控制

我们用自顶向下的思路去思考如何实现。首先抛开诸如识别色块/夹取动作参数计算等实现细节，单单考虑小车的运动流程。

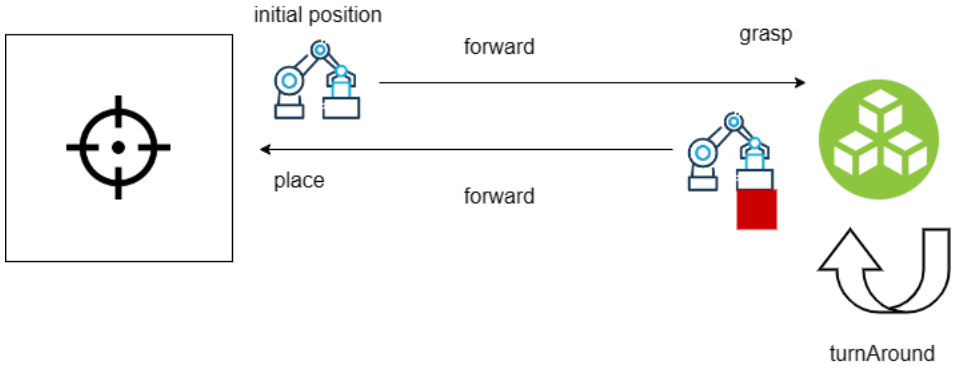
根据实验要求，可以归纳为四类：

动作类型	动作描述
forward	小车笔直向前走一段距离
turnAround	小车180°转身
grasp	小车识别色块并夹取
place	小车选择放置位置并放置物体

根据实验目标将这四类动作有序组合起来，可以得到：

1. forward：小车从初始位置笔直走向货物放置区
2. grasp：小车识别指定色块并夹取色块
3. turnAround：小车180°转身
4. forward：小车从货物放置区笔直返回目标位置区（小车初始位置与目标位置区大致相同）
5. place：小车选择目标位置区域作为放置位置，并将物体放下

流程图如下：



由于每次任务的流程总是一样的，因此我们可以用一个数组来记录整个流程，并编写驱动表作流程控制。

```
steps = ['forward','grasp','turnAround','forward','place']
cur_step = 0 # cur_step 意味着当前执行到哪一步了。从第零步开始
```

4.2. forward实现

小车向前走的实现分两种：巡线与非巡线。

1. 巡线：小车跟踪指定颜色的电工胶跑道抵达目的地
2. 非巡线：小车移动固定距离抵达目的地

这里我们选择了适用性较窄但稳定性更好的非巡线模式。也就是向底盘发布一个固定速度，并让主控休眠一段时间，在这段时间中，小车一直按照该固定速度前进。

```
if steps[cur_step] == 'forward':
    # 85cm = 480 unit thus 5 unit for 1 cm
    # run 5s,thus velocity should be 85
    set_velocity.publish(85.0,90.0,0.0)
    rospy.sleep(10.42)
    # stop
    set_velocity.publish(0,0,0)
```

```
    # next step
    flag=True
    cur_step += 1
```

其中,flag代表可以切换到下一步。85.0的麦轮线速度与10.42秒的主控休眠时间是根据实际场地调整得到的。

4.3. grasp实现

grasp需要拆解为更细粒度的动作：

1. 识别目标物块位置
2. 伸出机械臂至物块上方
3. 放下机械爪
4. 机械爪合拢，夹住物块
5. 机械臂复位

4.3.1. 识别目标色块：订阅识别服务

这属于CV的内容。也即特征提取、图像掩膜、腐蚀膨胀、寻找轮廓等传统机器视觉手段。而这一段在官方源码visual_processing_node中已经提供了：

```
def color_detect(img, color):
    global pub_time
    global publish_en
    global color_range_list

    rospy.loginfo('color = %s',color)

    if color == 'None':
        return img

    msg = Result()
    area_max = 0
    area_max_contour = 0
    img_copy = img.copy()
    img_h, img_w = img.shape[:2]
    frame_resize = cv2.resize(img_copy, size_m, interpolation=cv2.INTER_NEAREST)
    frame_lab = cv2.cvtColor(frame_resize, cv2.COLOR_BGR2LAB)  # 将图像转换到LAB空间

    if color in color_range_list:
        color_range = color_range_list[color]
        frame_mask = cv2.inRange(frame_lab, tuple(color_range['min']), tuple(color_range['max']))  # 对原图像和掩模进行位运算
        eroded = cv2.erode(frame_mask, cv2.getStructuringElement(cv2.MORPH_RECT, (2, 2)))          # 腐蚀
        dilated = cv2.dilate(eroded, cv2.getStructuringElement(cv2.MORPH_RECT, (2, 2)))            # 膨胀
        contours = cv2.findContours(dilated, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_NONE)[-2]         # 找出轮廓
        area_max_contour, area_max = getAreaMaxContour(contours)                                   # 找出最大轮廓

        if area_max > 100:  # 有找到最大面积
            (centerx, centery), radius = cv2.minEnclosingCircle(area_max_contour)  # 获取最小外接圆
            msg.center_x = int(Misc.map(centerx, 0, size_m[0], 0, img_w))
            msg.center_y = int(Misc.map(centery, 0, size_m[1], 0, img_h))
            msg.data = int(Misc.map(radius, 0, size_m[0], 0, img_w))
            cv2.circle(img, (msg.center_x, msg.center_y), msg.data+5, range_rgb[color], 2)
            publish_en = True

        if publish_en:
            if (time.time()-pub_time) >= 0.06:
                result_pub.publish(msg)  # 发布结果
                pub_time = time.time()

            if msg.data == 0:
                publish_en = False
                result_pub.publish(msg)

    return img
```

因此我们需要做的就是调用这个模块。这就涉及到之前提过的ros通信架构。注意到上述源码中的：

```
result_pub.publish(msg)
```

很显然，该node采用了发布订阅模式，因此我们就要先找到该模式对应的topic：

```
result_pub = rospy.Publisher('/visual_processing/result', Result, queue_size=1)
```

然后订阅该topic，定义处理返回消息的回调函数

```
result_sub = rospy.Subscriber('/visual_processing/result', Result, run)
```

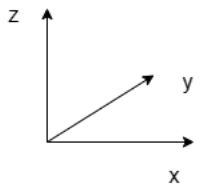
但这样还不够，这两步仅仅允许我们接受对应服务的返回消息，但请求对应服务的代码还未编写。这里我们想要在抵达货物放置区后开始识别指定色块，因此在grasp中我们要请求对应服务。而请求代理应在执行初始化时就声明：

```
visual_running = rospy.ServiceProxy('/visual_processing/set_running', SetParam)
```

```
elif steps[cur_step] == 'grasp':
    if flag:
        x_dis=500
        y_dis=0.15
        flag=False
        visual_running('color', target_color)
```

其中x_dis是机械臂横轴角度，范围0-1000，y_dis是机械臂纵轴长度，范围≥0。target_color是用户通过命令行自定义的识别颜色。

这里要稍微提一下小车遵循的坐标系，如下图：



该坐标系为视口坐标系，y轴朝向即车头朝向。

4.3.2. 识别目标色块：处理目标参数

在4.3.1节中我们请求了识别色块的对应接口，接下来就是处理接口返回的消息。在之前我们已经定义了回调函数run。这个函数在抽象层次上，我把它定义为一个代理，也即，目标参数消息返回之后，run函数负责将该参数保存到全局变量中，随后小车行为的函数move去访问这些全局变量，拿到结果。

```
with lock:
        if steps[cur_step] == 'grasp' and not arm_move:
            color_center_x = msg.center_x
            color_center_y = msg.center_y
```

这里要注意，arm_move是机械臂是否在运动的标志位。即，机械臂并没有在运动时，我们才会更新目标色块位置信息。试想一下如果不这么做会怎样：

1. 机械臂根据目标色块位置信息运动
2. 由于机械臂在运动，目标色块位置信息（相对位置信息）改变
3. 由于目标色块位置信息改变，机械臂继续运动

这样就陷入了无尽的死循环，因此这个标志位是很有必要的！

随后在move函数的grasp驱动项中，我们就可以使用这个目标位置信息参数了。也就是让小车将机械臂移动到该参数表示的位置。

4.3.3. 伸出机械臂至物块上方：PID算法

首先介绍一下使用的算法：PID算法。

PID算法（Proportional-Integral-Derivative Algorithm）是一种经典的控制算法，广泛应用于工业自动化领域。它基于对系统误差的比例、积分和微分进行控制，以实现精确的动态控制。PID包含三方面的控制，即比例控制（P）、积分控制（I）和微分控制（D）

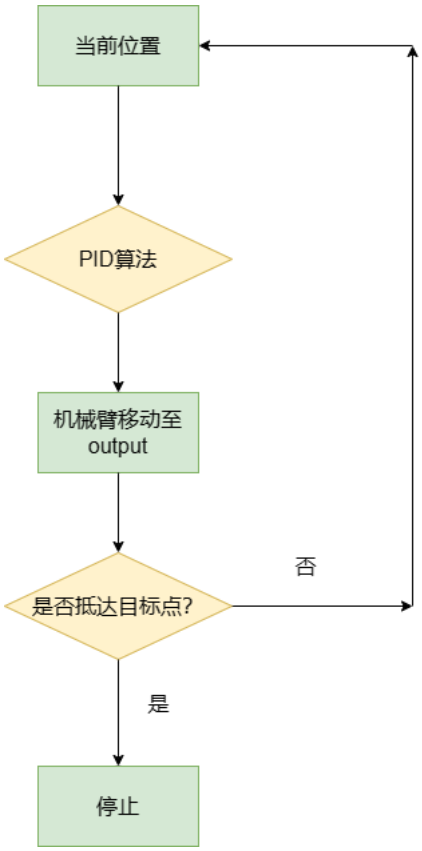
- 比例控制（Proportional Control）：比例控制是PID算法的基本组成部分。它根据当前系统的误差（偏差）与设定值之间的差异，生成一个与误差成比例的控制输出。比例控制的作用是根据误差的大小来提供快速而稳定的响应，使得控制器输出与误差成正比。比例增益参数Kp决定了比例控制的灵敏度，即控制器输出的变化程度。
- 积分控制（Integral Control）：积分控制是为了消除系统的静态误差而引入的。它对误差的累积进行控制，并生成一个与误差累积值成比例的控制输出。积分控制的作用是在长时间内消除系统的偏差，使得系统能够精确地跟踪设定值。积分增益参数Ki决定了积分控制的影响程度，即控制器输出与误差累积值的关系。
- 微分控制（Derivative Control）：微分控制是为了预测系统的未来变化趋势而引入的。它根据当前误差的变化率生成一个与变化率成比例的控制输出。微分控制的作用是提供对误差变化率的快速响应，使得系统能够更好地调节自身的行为。微分增益参数Kd决定了微分控制的灵敏度，即控制器输出与误差变化率的关系。

PID控制器根据上述比例、积分和微分控制的组合生成最终的控制输出。具体而言，PID控制器的输出值由以下公式计算：

```
OUT = Kp * Ek + Ki * Sk + Kd * Dk
```

要注意的是，PID算法不是一个一蹴而就的算法。每次的结果将作为下一次计算的初始值，对于当前我们的场景就是，机械臂根据PID算法的计算结果移动一段距离，但并没有直接到达目标点，而是重新再次计算，然后继续移动一段距离，直到抵达目标点为止。

流程图如下：



在官方给出的源码中，已经实现好了PID算法，因此我们只需要直接调用即可。

```
diff_x = abs(color_center_x - centreX)
diff_y = abs(color_center_y - centreY)

if diff_x < 10:
    color_x_pid.SetPoint = color_center_x
else:
    color_x_pid.SetPoint = centreX
color_x_pid.update(color_center_x)
dx = color_x_pid.output
rospy.loginfo("dx: %f", dx)
x_dis += int(dx)
x_dis = 200 if x_dis < 200 else x_dis
x_dis = 800 if x_dis > 800 else x_dis

if diff_y < 10:
    color_y_pid.SetPoint = color_center_y
else:
    color_y_pid.SetPoint = centreY
color_y_pid.update(color_center_y)
dy = color_y_pid.output
rospy.loginfo("dy: %f", dy)
y_dis += dy
y_dis = 0.12 if y_dis < 0.12 else y_dis
y_dis = 0.28 if y_dis > 0.28 else y_dis
```

1. diff_x/diff_y：物块与屏幕中心点的偏移量，作为pid算法的入参
2. dx/dy：pid算法给出的迭代偏移量

x_dis与y_dis加上这两段偏移量，可得当前位置。

那么怎么判断是否已经抵达目标点了呢？简单来说，如果最终位置逐渐收敛到某一个坐标点，我们就可以认为机械臂已经到达了目标点了，换句话说，只要当前位置和最终位置偏差足够小，这个迭代过程就应该判停。

```
if abs(dx) < 2 and abs(dy) < 0.003:
    num += 1
    if num == 10:
        num = 0
        # rospy.loginfo("ready to grasp")
        offset_y = Misc.map(target[2], -180, -150, -0.03, 0.03)
        arm_move = True  # ready to grasp
else:
    num = 0
```

这里采用计数法，对于全局变量num，初始值为0。每当机械臂距离目标点的偏移量收缩在某个范围内（对于x，<2；对于y，<0.003），那么可以认为这次的位置是合法的目标位置，如果连续10次都稳定在了该范围内，我们就认为机械臂当前通过了稳定性测试，可以继续下一步（下抓并夹取）了。

在这里需要注意，当前位置是需要不断更新的，以便下一轮迭代，因此在获取到dx/dy这组偏移量之后，要实时地移动机械臂来更新当前位置。

那么机械臂舵机的参数如何设置呢？这部分涉及到逆运动学求解，即，给定目标点参数，求解机械臂各舵机的参数。

这一部分官方源码已经为我们封装好了，我们只需要直接调用即可

```
target = ik.setPitchRanges((0, round(y_dis-offset_y, 4), -0.08), -180, -180, 0) #arm down
if target:
    servo_data = target[1]
    rospy.loginfo("sevo_data[3]: %d,servo_data[4]: %d,servo_d ata[5]: %d,x_dis: %d",servo_data['servo3'],servo_data['servo4'],servo_data['s
    bus_servo_control.set_servos(joints_pub, 1000, ((3, servo_data['servo3']), (4, servo_data['servo4']),
                                                    (5, servo_data['servo5']), (6, x_dis)))
```

其中target数组中的1号索引的内容就是各舵机求解结果，该结果本身也是个数组。我们只需要直接将其值通过set_servos函数设置到各舵机中即可。

之后的两步：

1. 机械爪闭合夹住物块

2. 机械臂复位

比较简单：

```
bus_servo_control.set_servos(joints_pub, 500, ((1, 450),)) # close claw
rospy.sleep(0.8)
bus_servo_control.set_servos(joints_pub, 1500, ((1, 450), (2, 500), (3, 80), (4, 825), (5, 625), (6, 500))) # 机械臂抬起来
rospy.sleep(1.5)
arm_move = False
# next step
cur_step += 1
```

4.3. turnAround实现

小车转身在实验一中也早已实现了，这里简单贴一下代码：

```
elif steps[cur_step] == 'turnAround':
      # -0.55 * 2 = -0.1 = 90deg, thus -1.1 * 2 = -2.2 = 180 deg
      set_velocity.publish(0.0,90.0,-0.55)
      rospy.sleep(4.0)
      # stop
      set_velocity.publish(0.0,0.0,0.0)
      rospy.sleep(1.0)
      # next step
      cur_step += 1
```

4.4. place实现

首先我们需要考虑一个问题：如何确定放置位置？

一种比较朴素的想法是，直接固定几个参数，让机械臂运动。但我们要考虑到，车每次返回目标区域时的位置并没有想象中那么”精准“。我们想让小车”笔直“、”180°转身“。但可能由于环境的影响，并没有办法做到100%的准确，因此可能导致固定参数下的放置可能会因为车运动时的”不精准“而放歪（比如放到框外）。

另外一个思路是，给小车一个参照物，让小车把物块放到参照物旁边，这样只要参照物在目标区域内，物块也会被摆放到框内。

后面一种思路就是我们采用的思路。这里我们选择了april_tag，一个很像二维码的标签，来作为参照物：

AprilTag1

把这个摆到目标区域内，之后小车放置时就把这个标签当作参照物。

为了识别这个标签，我们就要调用官方源码中封装的函数：

```
elif steps[cur_step] == 'place':
    if flag:
        x_dis=500
        y_dis=0.15
        flag=False
        visual_running('apriltag', '')
```

随后还是使用PID算法迭代处理，这一部分和上述代码很类似，不再赘述了。

4.5. 夹取稳定性

在先前的4.3节中，我们介绍了夹取的大致流程。但现场调试时我们发现，有时夹取并不稳定，并没有准确的夹上物块。对于这个问题，我提供两个解决思路。

4.5.1. 放缓速度（从预防的角度）

这也是我们组最终的解决方案，简洁有效。

在调试时，我们通过小车摄像头传回的图像，发现当小车锁定色块并移动时，往往由于机械臂移动太快而短暂的丢失目标，这样色块识别的模块就以为色块不在屏幕中，导致返回的参数偏差很大。机械臂依照这个偏差很大的参数做PID，很容易完全偏离色块位置，这样就再也检索不到色块，导致卡死。

在设置舵机运动的函数set_servos中，有这么一个参数：

```
bus_servo_control.set_servos(joints_pub, 20, ((3, servo_data['servo3']), (4, servo_data['servo4']),
                                                                          (5, servo_data['servo5']), (6, x_dis)))
```

其中20代表20ms，也就是机械臂运动的时间。这个时间太短了，导致上述的bug。于是我们将它改为了1000ms，放缓速度，提升夹取稳定性。最终成功跑出来了，证明了这个方案是可行的。

4.5.2. 差错检查（从弥补的角度）

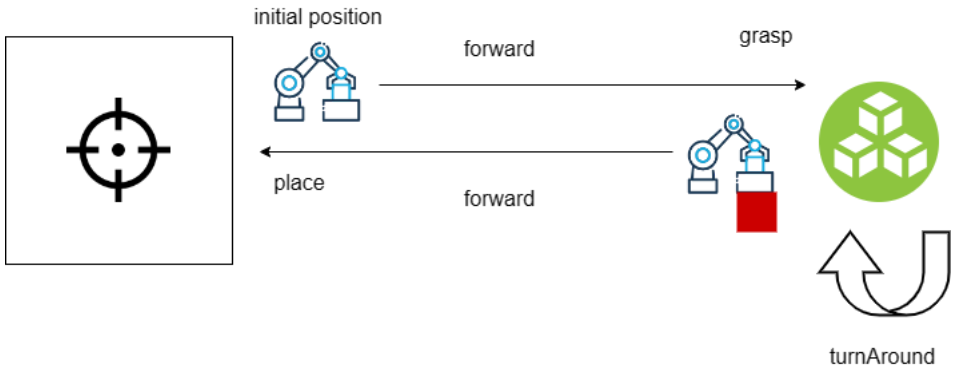
由于我们组夹取的稳定性还是不错的，最终这个解决方案就只停留在我的设想中，没有真正使用。现在假设小车执行完grasp这一步，接下来一步就是转身。注意，此时我们还没有取消颜色追踪服务，这个服务直到place时才被标签识别服务替换。

1. 如果小车没抓取到色块，则转身后，由于色块不在屏幕中，颜色追踪服务将返回一个错误的参数。
2. 如果小车抓取到色块，转身后，色块正被机械爪夹着，仍处于屏幕中，颜色追踪服务将返回一个正确的参数。

也就是说，在转身后，我们可以通过能否检测到色块这一事实，来确认我们是否真的抓到了色块。如果没有抓到，重新转身、夹取。

当然这个方案需要改造一部分上述提到的代码，尤其是turnAround驱动项中的。

5. 流程图



6. 完整工程

主要代码如下，其他部分见附录：

```
#!/usr/bin/python3
# coding=utf8
# Date:2022/05/30import sys
import sys
import cv2
import time
import math
import rospy
import numpy as np
from threading import RLock, Timer, Thread

from std_srvs.srv import *
from std_msgs.msg import *
from sensor_msgs.msg import Image
```

```
from sensor.msg import Led
from chassis_control.msg import *
from visual_processing.msg import Result
from visual_processing.srv import SetParam
from intelligent_transport.srv import SetTarget
from hiwonder_servo_msgs.msg import MultiRawIdPosDur

from armpi_pro import PID
from armpi_pro import Misc
from armpi_pro import bus_servo_control
from kinematics import ik_transform
lock = RLock()
ik = ik_transform.ArmIK()

steps = ['forward','grasp','turnAround','forward','place']
cur_step = 0
arm_move = False
target_color = 'None'
__isRunning = False

# color for LED
range_rgb = {
    'red': (0, 0, 255),
    'blue': (255, 0, 0),
    'green': (0, 255, 0),
}

x_dis = 500 # 6th steering engine initial angle
y_dis = 0.15 # a magic number for ik solution
color_center_x = 0
color_center_y = 0
apriltag_center_x = 0
apriltag_center_y = 0
centreX = 320
centreY = 410
offset_y = 0
object_angle = 0
angle = 0
flag=False

# cube tracking
color_x_pid = PID.PID(P=0.06, I=0, D=0)
color_y_pid = PID.PID(P=0.00003, I=0, D=0)
apriltag_x_pid = PID.PID(P=0.06, I=0, D=0)    # pid初始化
apriltag_y_pid = PID.PID(P=0.00003, I=0, D=0)
result_sub = None
heartbeat_timer = None

num = 0
def run(msg):
    global lock
    global arm_move
    global steps,cur_step
    global color_center_x,color_center_y
    global apriltag_center_x,apriltag_center_y,object_angle

    # rospy.loginfo("center_x = %f,center_y = %f",center_x,center_y)

    # rospy.loginfo("cur_step: %d",cur_step)
    with lock:
        if steps[cur_step] == 'grasp' and not arm_move:
            color_center_x = msg.center_x
            color_center_y = msg.center_y
        elif steps[cur_step] == 'place' and not arm_move:
            apriltag_center_x = msg.center_x
            apriltag_center_y = msg.center_y
            object_angle = msg.angle

def move():
    global  cur_step,steps
    global x_dis, y_dis
    global arm_move,flag
    global offset_y,object_angle,angle
    global __isRunning
    global target_color
    global num
    global color_center_x,color_center_y
```

```
    global apriltag_center_x,apriltag_center_y

    # rospy.loginfo('target_color = %s',target_color)

    while __isRunning:
        #rospy.loginfo("cur_step: %d",cur_step)
        # forward to the target
        if steps[cur_step] == 'forward':
            # 85cm = 480 unit thus 5 unit for 1 cm
            # run 5s,thus velocity should be 85
            set_velocity.publish(85.0,90.0,0.0)
            rospy.sleep(10.42)
            # stop
            set_velocity.publish(0,0,0)
            # next step
            flag=True
            cur_step += 1
        elif steps[cur_step] == 'grasp':
            if flag:
                x_dis=500
                y_dis=0.15
                flag=False
                visual_running('color', target_color)

            diff_x = abs(color_center_x - centreX)
            diff_y = abs(color_center_y - centreY)

            if diff_x < 10:
                color_x_pid.SetPoint = color_center_x
            else:
                color_x_pid.SetPoint = centreX
            color_x_pid.update(color_center_x)
            dx = color_x_pid.output
            rospy.loginfo("dx: %f", dx)
            x_dis += int(dx)
            x_dis = 200 if x_dis < 200 else x_dis
            x_dis = 800 if x_dis > 800 else x_dis

            if diff_y < 10:
                color_y_pid.SetPoint = color_center_y
            else:
                color_y_pid.SetPoint = centreY
            color_y_pid.update(color_center_y)
            dy = color_y_pid.output
            rospy.loginfo("dy: %f", dy)
            y_dis += dy
            y_dis = 0.12 if y_dis < 0.12 else y_dis
            y_dis = 0.28 if y_dis > 0.28 else y_dis

            target = ik.setPitchRanges((0, round(y_dis, 4), 0.03), -180, -180, 0)

            if target:
                servo_data = target[1]
                rospy.loginfo("sevo_data[3]: %d,servo_data[4]: %d,servo_data[5]: %d,x_dis: %d ", servo_data['servo3'],
                              servo_data['servo4'], servo_data['servo5'], x_dis)
                bus_servo_control.set_servos(joints_pub, 20, ((3, servo_data['servo3']), (4, servo_data['servo4']),
                                                              (5, servo_data['servo5']), (6, x_dis)))
                rospy.sleep(0.02)
            if abs(dx) < 2 and abs(dy) < 0.003:
                num += 1
                if num == 10:
                    num = 0
                    # rospy.loginfo("ready to grasp")
                    offset_y = Misc.map(target[2], -180, -150, -0.03, 0.03)
                    arm_move = True  # ready to grasp
            else:
                num = 0

            if arm_move:
                buzzer_pub.publish(0.1)
                bus_servo_control.set_servos(joints_pub, 500, ((1, 20),)) # open claw
                rospy.sleep(0.5)
                print(offset_y)
                target = ik.setPitchRanges((0, round(y_dis-offset_y, 4), -0.08), -180, -180, 0) #arm down
                if target:
                    servo_data = target[1]
                    rospy.loginfo("sevo_data[3]: %d,servo_data[4]: %d,servo_d ata[5]: %d,x_dis: %d",servo_data['servo3'],servo_data['servo4
                    bus_servo_control.set_servos(joints_pub, 1000, ((3, servo_data['servo3']), (4, servo_data['servo4']),
```

```
                                                            (5, servo_data['servo5']), (6, x_dis)))
            rospy.sleep(1.5)
            bus_servo_control.set_servos(joints_pub, 500, ((1, 450),)) # close claw
            rospy.sleep(0.8)
            bus_servo_control.set_servos(joints_pub, 1500, ((1, 450), (2, 500), (3, 80), (4, 825), (5, 625), (6, 500))) # 机械臂抬起来
            rospy.sleep(1.5)
            arm_move = False
            # next step
            cur_step += 1
    elif steps[cur_step] == 'turnAround':
        # -0.55 * 2 = -0.1 = 90deg, thus -1.1 * 2 = -2.2 = 180 deg
        set_velocity.publish(0.0,90.0,-0.55)
        rospy.sleep(4.0)
        # stop
        set_velocity.publish(0.0,0.0,0.0)
        rospy.sleep(1.0)
        # next step
        cur_step += 1
    elif steps[cur_step] == 'place':
        if flag:
            x_dis=500
            y_dis=0.15
            flag=False
            visual_running('apriltag', '')

        diff_x = abs(apriltag_center_x - centreX)
        diff_y = abs(apriltag_center_y - centreY)

        if diff_x < 10:
            apriltag_x_pid.SetPoint = apriltag_center_x
        else:
            apriltag_x_pid.SetPoint = centreX
        apriltag_x_pid.update(apriltag_center_x)
        dx = apriltag_x_pid.output
        rospy.loginfo("dx: %f", dx)
        x_dis += int(dx)
        x_dis = 200 if x_dis < 200 else x_dis
        x_dis = 800 if x_dis > 800 else x_dis

        if diff_y < 10:
            apriltag_y_pid.SetPoint = apriltag_center_y
        else:
            apriltag_y_pid.SetPoint = centreY
        apriltag_y_pid.update(apriltag_center_y)
        dy = apriltag_y_pid.output
        rospy.loginfo("dy: %f", dy)
        y_dis += dy
        y_dis = 0.12 if y_dis < 0.12 else y_dis
        y_dis = 0.28 if y_dis > 0.28 else y_dis

        target = ik.setPitchRanges((0, round(y_dis, 4), 0.0), -180, -180, 0)
        if target:
            servo_data = target[1]
            rospy.loginfo("sevo_data[3]: %d,servo_data[4]: %d,servo_data[5]: %d,x_dis: %d ", servo_data['servo3'],
                          servo_data['servo4'], servo_data['servo5'], x_dis)
            bus_servo_control.set_servos(joints_pub, 20, ((3, servo_data['servo3']), (4, servo_data['servo4']),
                                                          (5, servo_data['servo5']), (6, x_dis)))
            rospy.sleep(0.02)

        arm_move = True

        if arm_move:
            angle_pul = Misc.map(angle, 0, 80, 500, 800)
            bus_servo_control.set_servos(joints_pub, 500, ((2, angle_pul),))
            rospy.sleep(0.5)
            target = ik.setPitchRanges((0, round(y_dis + offset_y, 4), -0.08), -180, -180, 0) #arm down
            if target:
                servo_data = target[1]
                bus_servo_control.set_servos(joints_pub, 1000, ((3, servo_data['servo3']), (4, servo_data['servo4']),
                                                                (5, servo_data['servo5']), (6, x_dis)))
            rospy.sleep(1.0)
            buzzer_pub.publish(0.1)
            bus_servo_control.set_servos(joints_pub, 500, ((1, 120),)) # open claw
            rospy.sleep(0.5)

            # 机械臂复位
            bus_servo_control.set_servos(joints_pub, 1500, ((1, 120), (2, 500), (3, 80), (4, 825), (5, 625), (6, 500))) # 机械臂抬起来
            rospy.sleep(1.5)
```

```
                # the end
                break
    reset()

def init():
    rospy.loginfo('Lab2 Init')
    initMove()
    reset()

def initMove(delay=True):
    with lock:
        bus_servo_control.set_servos(joints_pub, 1500, ((1, 75), (2, 500), (3, 80), (4, 825), (5, 625), (6, 500)))
    if delay:
        rospy.sleep(2)

def reset():
    global x_dis,y_dis
    global cur_step
    global target_color
    global arm_move,flag
    global color_center_x,color_center_y
    global apriltag_center_x,apriltag_center_y

    with lock:
        cur_step = 0
        target_color = 'None'
        arm_move = False
        flag = False
        color_x_pid.clear()
        color_y_pid.clear()
        apriltag_x_pid.clear()
        apriltag_y_pid.clear()
        off_rgb()
        x_dis = 500
        y_dis = 0.15
        color_center_x = 0
        color_center_y = 0
        apriltag_center_x = 0
        apriltag_center_y = 0
        set_velocity.publish(0, 90, 0)

def off_rgb():
    led = Led()
    led.index = 0
    led.rgb.r = 0
    led.rgb.g = 0
    led.rgb.b = 0
    rgb_pub.publish(led)
    led.index = 1
    rgb_pub.publish(led)

def set_rgb(color):
    global lock
    with lock:
        led = Led()
        led.index = 0
        led.rgb.r = range_rgb[color][2]
        led.rgb.g = range_rgb[color][1]
        led.rgb.b = range_rgb[color][0]
        rgb_pub.publish(led)
        rospy.sleep(0.05)
        led.index = 1
        rgb_pub.publish(led)
        rospy.sleep(0.05)

def set_running(msg):
    if msg.data:
        start_running()
    else:
        stop_running()

    return [True, 'set_running']

def set_target(msg):
    global lock
    global target_color
```

```
    rospy.loginfo('%s',msg)
    with lock:
        target_color = msg.data
        led = Led()
        led.index = 0
        led.rgb.r = range_rgb[target_color][2]
        led.rgb.g = range_rgb[target_color][1]
        led.rgb.b = range_rgb[target_color][0]
        rgb_pub.publish(led)
        led.index = 1
        rgb_pub.publish(led)
        rospy.sleep(0.1)
    return [True, 'set_target']

def stop_running():
    global lock
    global __isRunning

    rospy.loginfo('stop running Lab2')
    with lock:
        __isRunning = False
        reset()
        initMove(delay=False)
        #set_velocity.publish(0, 0, 0)
        rospy.ServiceProxy('/visual_processing/set_running', SetParam)()

def start_running():
    global lock
    global __isRunning

    rospy.loginfo('start running Lab2')
    with lock:
        #init()
        __isRunning = True
        rospy.sleep(0.1)
        th = Thread(target=move)
        th.setDaemon(True)
        th.start()

# enter service
def enter_func(msg):
    global lock
    global result_sub

    rospy.loginfo('enter object tracking')
    init()
    with lock:
        if result_sub is None:
            # wake up visual_processing_node
            rospy.ServiceProxy('/visual_processing/enter', Trigger)()
            # subscribe the result from the detection module of visual_processing_node
            result_sub = rospy.Subscriber('/visual_processing/result', Result, run)

    return [True, 'enter']

# exit service
def exit_func(msg):
    global lock
    global result_sub
    global __isRunning
    global heartbeat_timer

    rospy.loginfo('exit Lab2')
    with lock:
        __isRunning = False
        rospy.ServiceProxy('/visual_processing/exit', Trigger)()
        reset()
        try:
            if result_sub is not None:
                result_sub.unregister()
                result_sub = None
            if heartbeat_timer is not None:
                heartbeat_timer.cancel()
                heartbeat_timer = None
        except BaseException as e:
            rospy.loginfo('%s', e)

    return [True, 'exit']
```

```
# heartbeat connection service
def heartbeat_srv_cb(msg):
    global heartbeat_timer

    if isinstance(heartbeat_timer, Timer):
        heartbeat_timer.cancel()
    if msg.data:
        heartbeat_timer = Timer(5, rospy.ServiceProxy('/Lab2/exit', Trigger))
        heartbeat_timer.start()
    rsp = SetBoolResponse()
    rsp.success = msg.data

    return rsp


if __name__ == '__main__':
    # init node
    rospy.init_node('Lab2', log_level=rospy.DEBUG)
    # Steering Engine
    joints_pub = rospy.Publisher('/servo_controllers/port_id_1/multi_id_pos_dur', MultiRawIdPosDur, queue_size=1)
    # ServiceProxy
    visual_running = rospy.ServiceProxy('/visual_processing/set_running', SetParam)
    # Services
    enter_srv = rospy.Service('/Lab2/enter', Trigger, enter_func)
    running_srv = rospy.Service('/Lab2/set_running', SetBool, set_running)
    set_target_srv = rospy.Service('/Lab2/set_target', SetTarget, set_target)
    exit_srv = rospy.Service('/Lab2/exit', Trigger, exit_func)
    heartbeat_srv = rospy.Service('/Lab2/heartbeat', SetBool, heartbeat_srv_cb)
    # Publishers
    # Chassis Control
    set_velocity = rospy.Publisher('/chassis_control/set_velocity', SetVelocity, queue_size=1)
    set_translation = rospy.Publisher('/chassis_control/set_translation', SetTranslation, queue_size=1)
    # Buzzer Control
    buzzer_pub = rospy.Publisher('/sensor/buzzer', Float32, queue_size=1)
    # RGB LED Control
    rgb_pub = rospy.Publisher('/sensor/rgb_led', Led, queue_size=1)
    rospy.sleep(0.5)

    try:
        rospy.spin()
    except KeyboardInterrupt:
        rospy.loginfo('Shutting down')
```

7. 如何复现我们的成果

仅提供命令行方式。请注意，小车与物块摆放的参数请参照2023年秋季学期SDU软件学院实验楼501的实验环境。如果不同，请自行调整代码中的参数。

7.1. 启动服务节点

```
rosrun Lab2 Lab2_node.py
```

7.2. 进入服务

```
rosservice call /Lab2/enter "{}"
```

7.3. 设定目标颜色

```
rosservice call /Lab2/set_target "data: 'blue'"
```

7.4. 开始运行

```
rosservice call /Lab2/set_running "data: true"
```