

behavior 策略代码阅读

入口 main.py

创建黑板和角色类



复制代码

```
bb = DBlackboard()
cls = find_behavior(bb.param.role)
# cls = Striker(*)
behavior = cls(bb)
```

这边要注意



复制代码

```
def find_behavior(name):
    try:
        cls = getattr(role, name) # 从 role 模块中获取名为 "Striker" 的类
        return cls
    except AttributeError:
        return None
```

cls = role.Striker() 这个是返回的Striker类，不是对象

然后behavior = cls(bb) 创建了striker(bb)这个对象，所以behavior是striker的实例



复制代码

```
class Striker(Role):
    """
    A striker is actively pursuing the ball.
    """

    def __init__(self, bb):
        super(Striker, self).__init__(bb)
        root = DynamicGuardSelector([(Parallel(PrepareIMU(bb),
                                                InitialParticles(bb)),
                                           LowerBoardReconnected()),
                                     (Sequence(
                                         Wait(1),
                                         EntryScanField(bb),
                                     ), IMUInitialized()),
                                     # (Parallel(GoToInitPos(bb),
FindBall(bb)),
```

```

#   BallSeen()),
#   (ScanFieldNew(bb),
MotionReStable()),

(Strike(bb), None)])

self.add_child(root)

```

然后现在可以梳理一下

behavior是striker的实例，striker继承自role，role继承自BehaviorTree

进循环



复制代码

```

bb.reset()
behavior.step()
bb.publish()

```

behavior.step()

step()函数是BehaviorTree类里面的，没有被overload，BehaviorTree里面的step()是这样的



复制代码

```

def step(self):
    # 判断1: 根任务是否正在运行?
    if self.root_task.get_status() == Status.RUNNING:
        # 情况A: 正在运行 → 继续执行
        self.root_task.run() # ← 跳到执行任务
    else:
        # 情况B: 未运行 → 启动任务
        self.root_task.set_control(self)
        self.root_task.on_start()

        # 判断2: 守卫条件是否通过?
        if self.root_task.check_guard():
            self.root_task.run() # ← 跳到执行任务
        else:
            self.root_task.failure()

```

然后他这里调用了很多root_task这个成员变量

这个成员变量哪来的呢?

我们看Striker的定义，Striker里面没写root_task这个东西

✓ 复制代码

```
class Striker(Role):
    """
    A striker is actively pursuing the ball.
    """

    def __init__(self, bb):
        super(Striker, self).__init__(bb)
        root = DynamicGuardSelector([(Parallel(PrepareIMU(bb),
                                                InitialParticles(bb)),
                                                LowerBoardReconnected()),
                                     (Sequence(
                                         Wait(1),
                                         EntryScanField(bb),
                                     ), IMUInitialized()),
                                     # (Parallel(GoToInitPos(bb),
FindBall(bb)),
                                     # BallSeen()),
                                     # (ScanFieldNew(bb),
MotionReStable()),
                                     (Strike(bb), None)])

        self.add_child(root)
```

但是里面有add_child(root)

✓ 复制代码

```
def add_child(self, child):
    if self.root_task is not None:
        raise RuntimeError('A behavior tree cannot have more than
one root task')
    child.set_control(self)
    self.root_task = child
    return 0
```

add_child里面有写self.root_task = child，这边的child是形参，实参是上面的root
也就是DynamicGuardSelector的实例

这里我们宕开一笔，讲讲这个DynamicGuardSelector，DGS是一个branch节点，枝干节点，它会执行第一个Guard为True的子任务，执行成功后重置。比如这里



复制代码

```
[(Parallel(PrepareIMU(bb),
           InitialParticles(bb)),
  LowerBoardReconnected()),
 (Sequence(
   Wait(1),
   EntryScanField(bb),
 ), IMUInitialized()),
 (Strike(bb), None)]
```

这边是一个列表，每一个元素是一个(行为，触发条件)的元组，比如这边的LowerBoardReconnected(), IMUInitialized(), None都是触发条件，None就是默认true

有必要这里讲一下DynamicGuardSelector的继承关系

DynamicGuardSelector继承自Branch，Branch继承自Task

ok 那我们就回去看



复制代码

```
def step(self):
    # 判断1：根任务是否正在运行？
    if self.root_task.get_status() == Status.RUNNING:
        # 情况A：正在运行 → 继续执行
        self.root_task.run() # ← 跳到执行任务
    else:
        # 情况B：未运行 → 启动任务
        self.root_task.set_control(self)
        self.root_task.on_start()

    # 判断2：守卫条件是否通过？
    if self.root_task.check_guard():
        self.root_task.run() # ← 跳到执行任务
    else:
        self.root_task.failure()
```

这个里面的调用

```
1 self.root_task.get_status()
```

这个写在task里面，直接return self._status

其中 _status有下面这些情况，是一个enum

[复制代码](#)

```
class Status(Enum):
    """The enumeration of the values that a task's status can have."""
    # Means that the task has never run or has been reset.
    FRESH = 0
    # Means that the task needs to run again.
    RUNNING = 1
    # Means that the task returned a failure result.
    FAILED = 2
    # Means that the task returned a success result.
    SUCCEEDED = 3
    # Means that the task has been terminated by an ancestor.
    CANCELLED = 4
```

```
2 self.root_task.set_control(self)
```

[复制代码](#)

```
def set_control(self, control):
    """
    This method will set a task as this task's control (parent).

    :param control: the parent task
    :type control: Task
    """
    self._control = control
    self._tree = control.get_tree()
```

但是注意，调用的时候的self是striker的self，所以control是DynamicGuardSelector的self，self.root_task._control = BehaviorTree 实例，get_tree()是返回了striker._tree

[复制代码](#)

```
def get_tree(self):
    """
    Returns the behavior tree this task belongs to.

    :return: the behavior tree
    :rtype: Task
    """
    if self._tree is None and \
```

```
        self._control is not None and \
        self._control.get_tree() is not None:
    self._tree = self._control.get_tree()
    return self._tree
```

3 self.root_task.on_start() 什么都不做, pass

4 self.root_task.check_guard()



复制代码

```
class Task(object):
    def check_guard(self):
        # 判断: 是否有守卫?
        if self._guard is None:
            return True # 没有守卫 → 直接通过

        # 递归检查守卫的守卫
        if not self._guard.check_guard():
            return False

        # 执行守卫任务

self._guard.set_control(self._control.get_tree().guard_evaluator)
self._guard.on_start()
self._guard.run()

# 判断守卫结果
if self._guard.get_status() == Status.SUCCEEDED:
    return True
elif self._guard.get_status() == Status.FAILED:
    return False
```

5 self.root_task.run()

这个是在DynamicGuardSelector里面被重写了



复制代码

```
def run(self):
    # 步骤1: 初始化要运行的子节点索引
    self._child_to_run = None

    # 步骤2: 遍历所有子节点, 找第一个守卫通过的
    for idx, child in enumerate(self._children):
        if child.check_guard(): # ← 调用子节点的 check_guard()
```

```

        self._child_to_run = idx
        self.append_buffer(self._child_to_run)
        break # 找到第一个就停止

# 步骤3: 优先级抢占检查 (第一次运行时跳过)
if self.check_buffer() and self._running_child is not None:
    # ... 优先级切换逻辑

# 步骤4: 判断是否找到可运行的子节点
if self._child_to_run is None:
    self.failure() # 没找到 → 失败
else:
    # 步骤5: 判断是否首次运行
    if self._running_child is None:
        # 首次运行 → 初始化
        self._running_child = self._child_to_run
        running_child = self.get_child(self._running_child)
        running_child.set_control(self)
        running_child.on_start()
    else:

        # 继续运行
        running_child = self.get_child(self._running_child)

# 步骤6: 执行选中的子节点
running_child.run() # ← 递归调用子节点的 run()

```

这边有点乱了，但是感觉，整个step都没什么用，那关键是这个striker的逻辑。我们说，这个striker除了重连或者启动imu之外，都在运行这个(Strike(bb), None)，那我们重点是这个Strike(bb)

∨
📄 复制代码

```

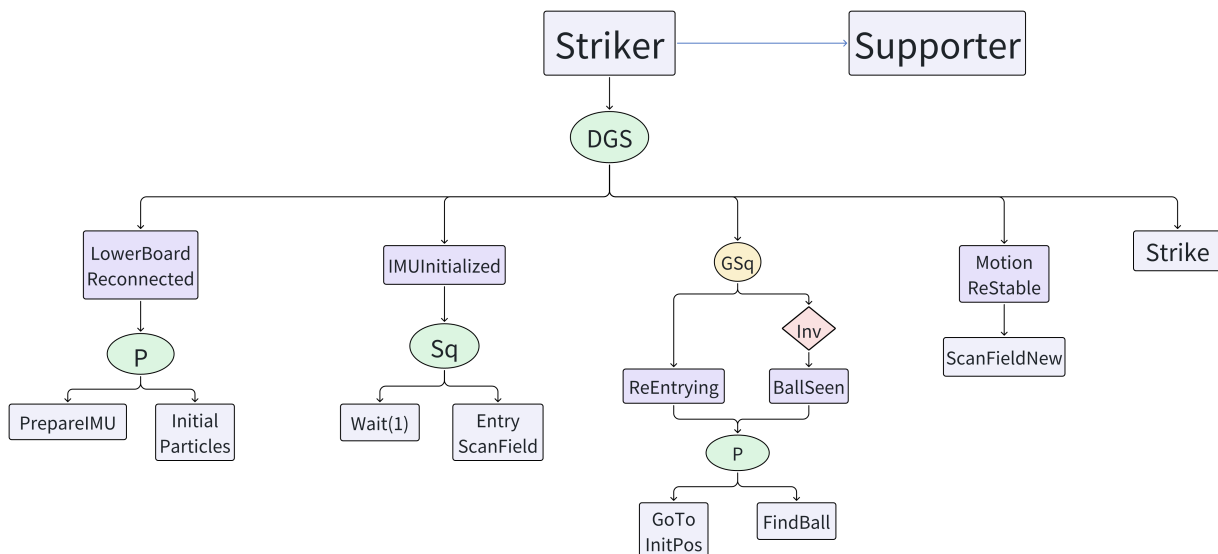
class Strike(Parallel):

    def __init__(self, bb):
        args = (StrikerBrain(bb),
                DynamicGuardSelector([
                    (Attack(bb),
                     GuardSelector(BallSeen(), TeamBallSeen(),
                                     Kicking()))),
                # (FindLastBall(bb), BallLostRecent()),
                # (Parallel(GoToInitPos(bb), FindBall(bb)),
                #  GuardSequence(ReEntrying(), Invert(BallSeen()))),
                (SeekBall(bb), None),

```

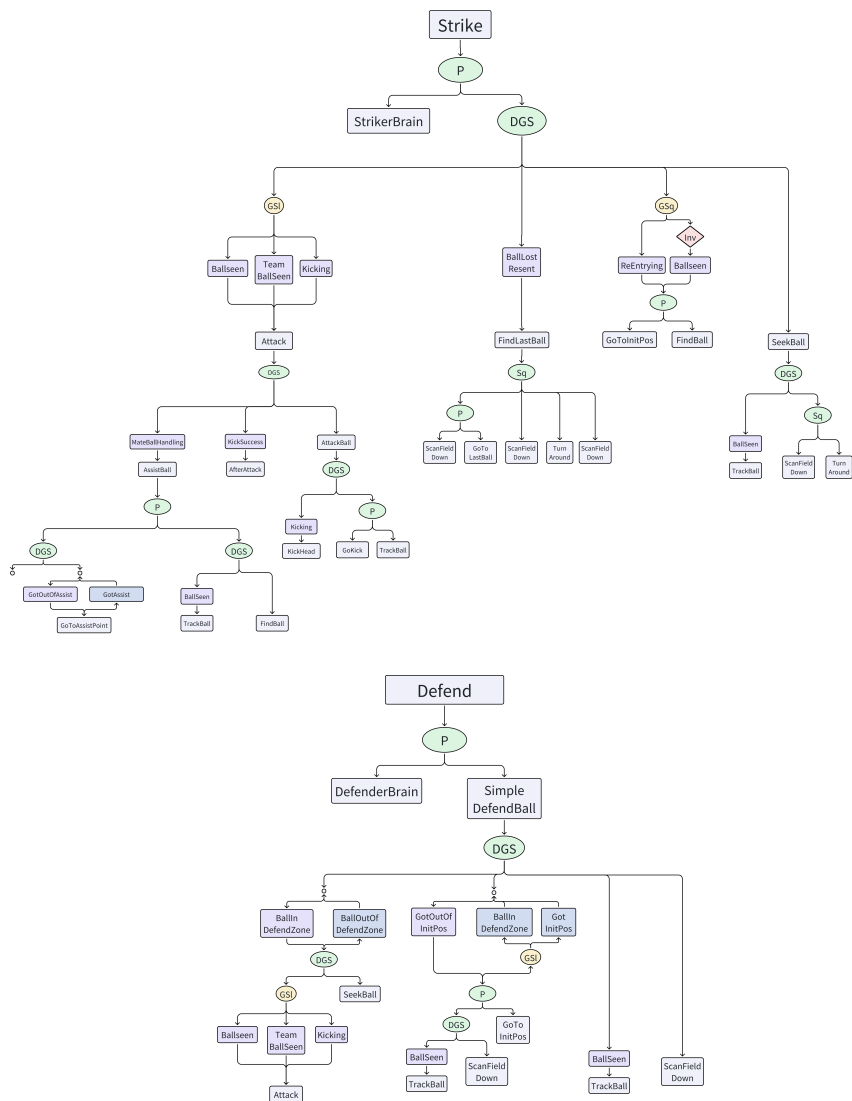
```
    ]))  
    super(Strike, self).__init__(*args)
```

可见，这个Strike也是一个行为树，



Strike继承自Parallel

左子树是StrikerBrain，右子树是又是一个DGS



我们先看左子树，StrikerBrain继承自skill，skill继承自leaf，leaf继承自task，这个子树在不停的运行execute()



复制代码

```
def execute(self):
    x = self.bb.vision_info.robot_pos.x
    y = self.bb.vision_info.robot_pos.y
    ballx = self.bb.vision_info.ball_global.x
    bally = self.bb.vision_info.ball_global.y

    if self.goal_timer.elapsed() > self.goal_time_out:
        self.use_goal_target = False
        self.goals_mid = []
        self.goal_timer.restart()
    my_goal = None
    if self.bb.vision_info.see_goal:
        if self.bb.param.attack_right:
            my_goal = self.bb.vision_info.goal_global.x < 0
```

```

        else:
            my_goal = self.bb.vision_info.goal_global.x > 0

            # if self.bb.vision_info.see_goal and not my_goal:
            #     goal =
VecPos.from_vector3(self.bb.vision_info.goal_global)
            #     self.goals_mid.append(goal)
            #     if len(self.goals_mid) > self.num_to_determine:
            #         self.target = VecPos()
            #         for pt in self.goals_mid:
            #             self.target.x += pt.x
            #             self.target.y += pt.y
            #         self.target.x /= len(self.goals_mid)
            #         self.target.y /= len(self.goals_mid)
            #         self.bb.attack_target = self.target
            #         # Shoot into the goal deeper
            #         self.target.x += self.target.x / abs(self.target.x) *
100
            #         self.goals_mid = []
            #         self.use_goal_target = True
            #         self.goal_timer.restart()
            # elif not self.use_goal_target:

            if self.bb.target_used:

                # set target.x
                if fabs(ballx) < 250 or fabs(bally) < 100:
                    if self.bb.param.attack_right:
                        self.target.x = 550
                    else:
                        self.target.x = -550
                elif fabs(ballx) > 300 and fabs(bally) > 150:
                    if self.bb.param.attack_right:
                        self.target.x = 450
                    else:
                        self.target.x = -450
                else:
                    if fabs(x) > fabs(self.target.x):
                        self.target.x += self.target.x / abs(self.target.x)
* 70

                # set target.y
                if fabs(ballx) > 300 and fabs(bally) > 100:
                    # print("target used: set y to magic_y")
                    # self.target.y = -bally / abs(bally) * self.magic_y

```

```

        self.target.y = bally / abs(bally) * self.magic_y
    elif fabs(ballx) < 250:
        if bally > 70:
            self.target.y = self.magic_y
        elif bally < -70:
            self.target.y = -self.magic_y

        # if fabs(bally) > self.bb.param.goal_width / 2:
        #     self.target.x -= 100
        # elif fabs(ballx) > 300:
        #     pass
        # if fabs(bally) > self.bb.param.goal_width / 2:
        #     self.target.x += 100
    # if fabs(ballx) > 300:

        # if fabs(ballx) > 200 and fabs(y) < 70 and
fabs(self.target.y) == 30:
        #     self.target.y *= 2
        self.bb.target_used = False

self.bb.target_used = True
self.bb.attack_target = self.target

# enable kicking
self.bb.kick_enabled = True

# Don't kick straight when kickoff
if self.bb.gc_info.kickoff and
self.bb.timer_gc_playing.elapsed() < 15:
    if self.bb.param.attack_right:
        self.bb.attack_target = VecPos(550, 320 *
self.random_side)
    else:
        self.bb.attack_target = VecPos(-550, 320 *
self.random_side)

```

StrikerBrain的init会初始化self.target和bb.attack_target



复制代码

```

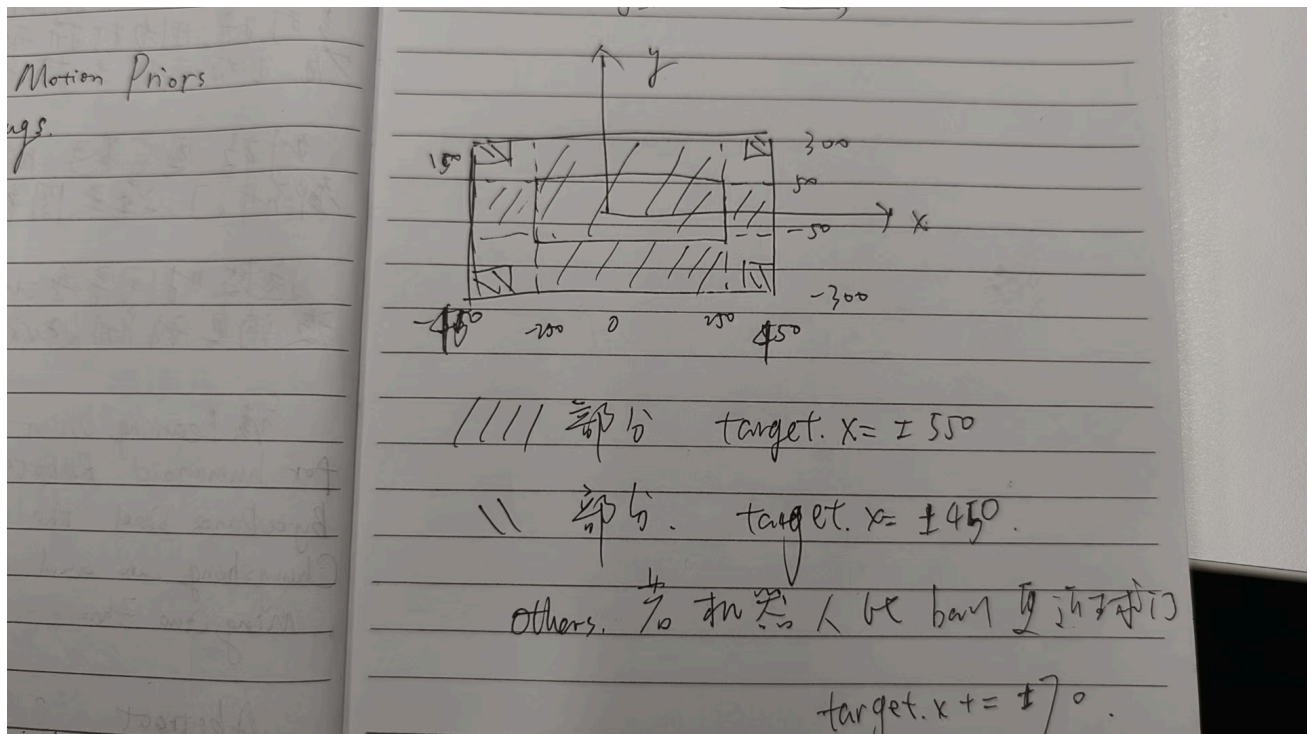
self.magic_y = 30
if bb.param.attack_right:
    self.target = VecPos(550, self.magic_y)
else:
    self.target = VecPos(-550, self.magic_y)

```

```
bb.attack target = self.target
```

其中这个magic_y是定义的一个魔数，用来射歪一点，让对方更难防守（）

然后execute里面首先是讲了target.x和target.y的生成逻辑，有一系列复杂的逻辑



well，看实机没看出来这部分（）

以上StrikerBrain的部分只是把得到的target写进blackboard，计算与执行解耦。执行的部分是右子树，也就是以下部分：

✓ 复制代码

```
DynamicGuardSelector([
    (Attack(bb),
      GuardSelector(BallSeen(), TeamBallSeen(), Kicking()))),
    # (FindLastBall(bb), BallLostRecent()),
    # (Parallel(GoToInitPos(bb), FindBall(bb)),
    #   GuardSequence(ReEntering(), Invert(BallSeen()))),
    (SeekBall(bb), None),
```

这又是一个DGS，DGS包括两个子树

✓ 复制代码

```
(Attack(bb), GuardSelector(BallSeen(), TeamBallSeen(), Kicking()))
```

和

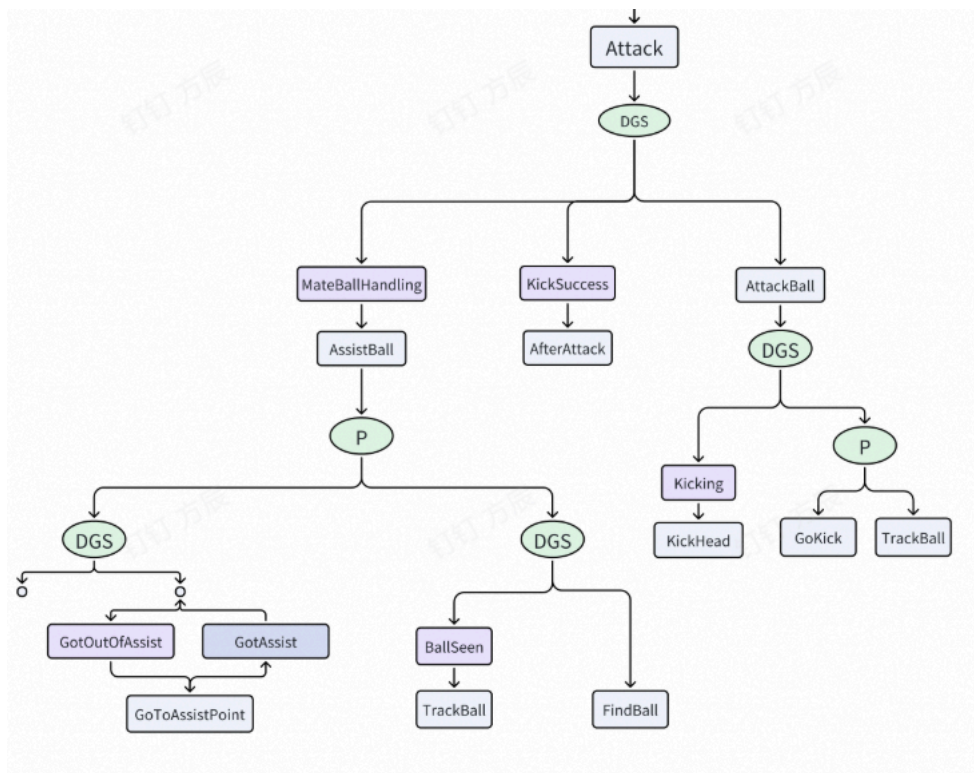
```
(SeekBall(bb), None)
```

这个也很好读懂，Attack(bb)的执行条件是GuardSelector(BallSeen(), TeamBallSeen(), Kicking())

GuardSelector是依次检查里面的条件，有一个为True就返回True，否则返回False

那么就是如果机器人看到球、队友看到球或者正在踢球，就都执行Attack(bb)，否则就执行SeekBall(bb)

OK，我们来看Attack(bb)



Attack总体是一个DGS



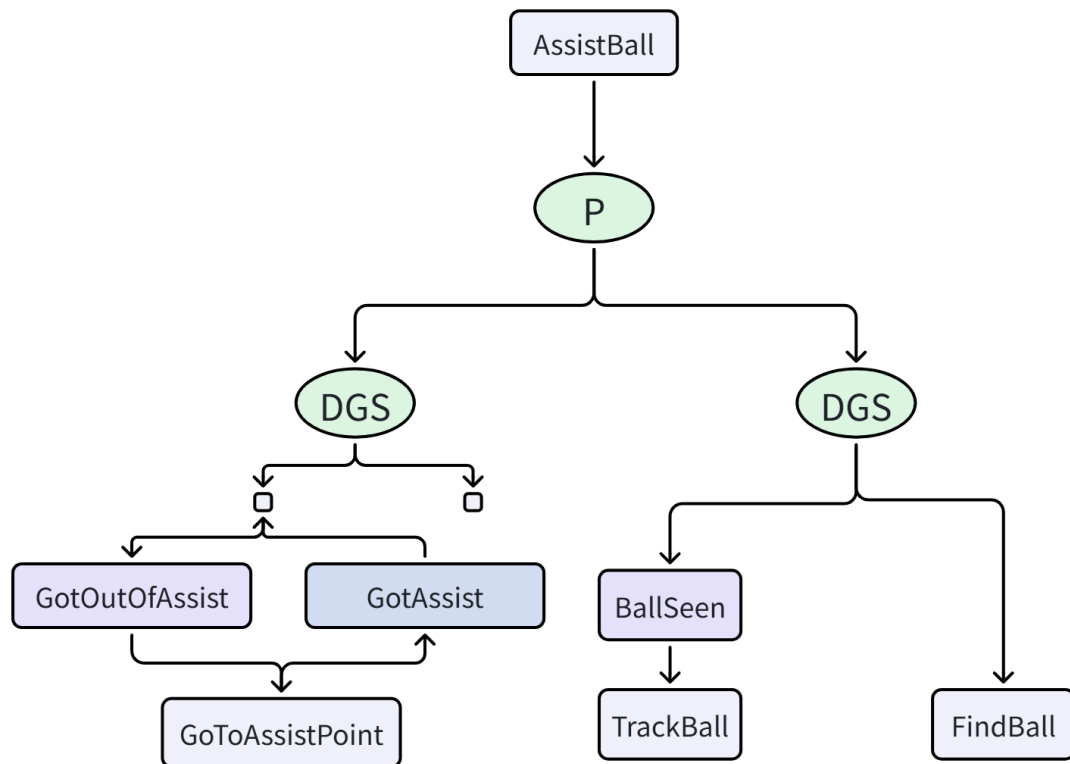
复制代码

```
(AssistBall(bb), MateBallHandling()),  
(AfterAttack(bb), KickSuccess()),  
(AttackBall(bb), None)
```

其中有三个分支

1 AssistBall

触发条件：MateBallHandling() → bb.mate_ball_handling == True



AssistBall (Parallel)

```

├── DynamicGuardSelector
│   ├── (GoToAssistPoint(bb), GotOutOfAssist(), GotAssist()) ← 移动到辅助点
│   └── (Do(bb), None, None)
└── DynamicGuardSelector
    ├── (TrackBall(bb), BallSeen()) ← 看到球：头追踪
    └── (FindBall(bb), None) ← 没看到：找球
  
```

这里需要特别注意，之前我们说DGS的元素是元组，第一个是行为，第二个是判断条件（守卫），但是这里我们发现这便是三元组，因为我们这里写了`use_exit=True`，启动了退出条件，现在即使Guard变成了False，也不会直接退出，而是会等到退出条件变成True才会退出

这下我们就能看懂了了

特别要注意，这里的`Do(bb)`，始终在执行`crouch()`，也就是`self.action_generator.crouch()`，也就是`return BodyCommand(0, 0, 0, BodyCommand.CROUCH)`

这个CROUCH对应的是步态0，也就是啥也不动，而不是其对应的英文“蹲下”，历史遗留问题



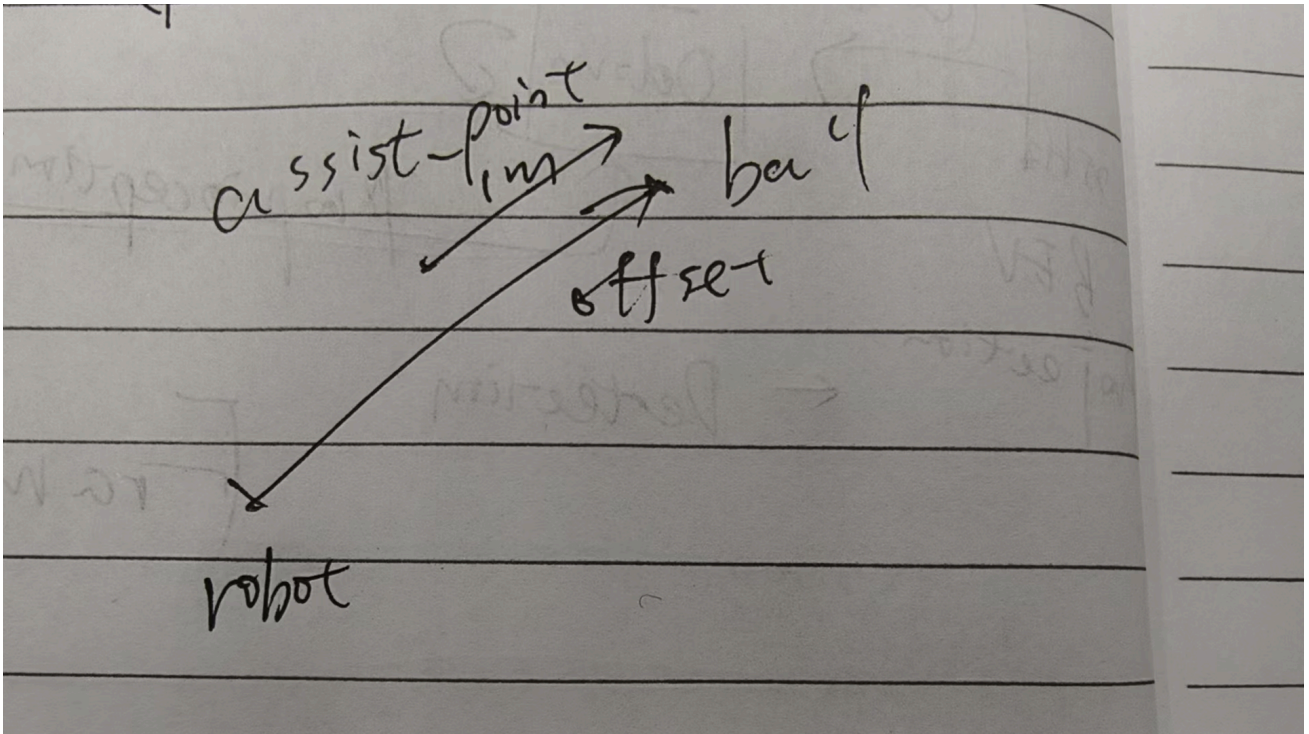
复制代码

```
case 0: // 站着
```

```
/* code */
parameters.stp.vel_cmd = {0,0,0};
parameters.stp.gait_type = 0;
break;
```

下面我们详细讲GoToAssistPoint(bb), GotOutOfAssist(), GotAssist()

GoToAssistPoint(bb) 计算辅助点并且走过去



这里面有一些危险的操作

1 比如没对assist_point进行检查是否在场外

2 这个assist_point的意义多大值得思考，走到机器人和球连线靠近机器人方向的1m的地方，而且没考虑持球队友？

这边的代码需要有人专门负责去改。

！改用bhuman代码之后，有新的计算方法计算assist_point

`dest = self.bb.best_pos`

`dest.z = degree_between(robot_pos, ball_global)`

那么我们来看这个bb.best_pos是怎么计算出来的

[复制代码](#)

```
@property
def best_pos(self):
    field_length = self.param.field_length
    field_width = self.param.field_width

    voronoi_region = self.voronoi_region
    sample_points = sample_points_in_voronoi(voronoi_region,
num_points=20)

    position_rater = PositionRater(self.param.role)

    best_score = -Inf
    best_point = None
    #计算采样点中的最佳位置与分数
    ball_pos =
DBlackboard.vector3_to_tuple(self.vision_info.ball_global)
    for point in sample_points:
        score = position_rater.posRating(point, ball_pos,
self.online_mates_pos,
                                (-field_length/2,
field_length/2, -field_width/2, field_width/2))
        if score > best_score:
            best_point = point
            best_score = score

    # 当前位置与分数
    current_score =
position_rater.posRating(self.vision_info.robot_pos,
self.vision_info.ball_global, self.online_mates_pos,
                                (-field_length/2,
field_length/2, -field_width/2, field_width/2))
    current_pos =
DBlackboard.vector3_to_tuple(self.vision_info.robot_pos)

    if best_point is not None:
        best_x, best_y = best_point
        if current_pos is not None:
            current_x, current_y = current_pos
            distance = sqrt((current_x - best_x) ** 2 + (current_y
- best_y) ** 2)
        else:
```

```

        distance = float("inf")

        score_improvement = best_score - current_score
        #计算出的最佳位置显著高于当前位置且距离大于阈值，则更新最佳位置，
        否则保持当前位置
        self.score_improvement_threshold = 0.1    #评分提升阈值
        self.distance_threshold = 20.0          # 距离阈值
        if (score_improvement/best_score) >
self.score_improvement_threshold and distance > self.distance_threshold:
            best_point = best_point

        else:
            best_point = current_pos
    else:
        best_point = current_pos    #best_point是一个元组

        best_pos = VecPos.from_list(best_point)    # convert to VecPos (元
        组也可以使用from_list这个方法)

    return best_pos

```

首先是获得voronoi_region，每次获取voronoi_region就会调用一次

复制代码

```

@property
def voronoi_region(self):
    #队友位置
    teammates_pos = dict(self.online_mates_pos) #get_voronoi_regions函数
    要求输入的team_info是dict格式

    my_x = float(self.vision_info.robot_pos.x)
    my_y = float(self.vision_info.robot_pos.y)
    # 将自己 (self.id) 加入到字典中
    # 这样传给算法的字典就包含了 {1: (x,y), 2: (x,y)...}
    teammates_pos[self.id] = (my_x + 0.01*self.id, my_y + 0.01*self.id)
    #场地信息
    field_length = self.param.field_length
    field_width = self.param.field_width
    field_bounds = ((-field_length/2, -field_width/2), (field_length/2,
    field_width/2))
    #计算自己所在的voronoi区域
    voronoi_region = get_voronoi_region(self.id, teammates_pos,
    field_bounds)
    return voronoi_region

```

把自己的id，把整个队伍的teammates_pos（包括自己略微位移之后的），还有场地信息传入get_voronoi_regions函数（这其实可以写死，省一点运算时间）

这个get_voronoi_regions有一点复杂，但简单说就是取每两个机器人的中垂线，分出一些空间，然后把整个场地分成每个机器人的部分。

在所在的voronoi_region里面采样20个点，然后根据role进行打分

```
position_rater = PositionRater(self.param.role)
```

PositionRater逻辑

1 根据role设置参数

∨ 复制代码

```
"striker": {
    'ball_weight': 1.8,
    'goal_weight': 1, # 增加球门权重（球门变宽，射门机会增加）
    'teammate_weight': -0.6,
    'boundary_weight': -0.8,
    'offensive_bias': 1.4, # 增加进攻倾向
    'shooting_angle_weight': 1.3, # 增加射门角度权重
    'sigma_ball': 200,
    'sigma_teammate': 100,
    'sigma_boundary': 80,
    'preferred_ball_distance': 80,
    'wide_goal_bonus': 1.2, # 宽球门奖励系数
},
```

2

∨ 复制代码

```
total_rating =
(ball_potential + goal_potential +
teammate_potential + boundary_potential + role_potential)
```

评分函数由5个势场叠加

- ball_potential 球吸引，距离球越近分越高
- goal_potential 球门吸引

- teammate_potential 队友排斥
- boundary_potential 边界排斥
- role_potential 射门角度和进攻位置的加成

有疑问！

这个为什么是离球越近越好，不会直接撞上去吗？

assist_point也是这个给出的，但是这个assist没有单独的逻辑啊，所有striker都是这样的逻辑

下面找20个采样点中分数最高的，然后和当前位置比决定要不要移动

移动条件是

分数提升超过 10% ($\text{score_improvement} / \text{best_score} > 0.1$)

距离超过 20cm

2 AfterAttack

这个分支是为了避免不停地射

(AfterAttack(bb), KickSuccess())

KickSuccess是

class KickSuccess(ConditionLeaf):

def condition(self):

return 0 <= self.get_bb().timer_kick_success.elapsed() <= 1

timer_kick_success 在黑板里这样更新



复制代码

```
if self.motion_info.status == MotionInfo.KICKING:
    self.kicking = True
else:
    if self.kicking:
        # 刚从 KICKING 变成非
        KICKING
        self.timer_kick_success.restart()
        # ← 踢完的瞬间重置计时器
    self.kicking = False
```

初始值 `Timer(left_shift=9999)`, `elapsed` 一开始就是 9999, 所以 `KickSuccess` 默认为 `False`。

触发时机: 踢球动作结束的瞬间, `timer_kick_success` 重置, `elapsed` 从 0 开始计时, 1 秒内 `KickSuccess` 为 `True`

`AfterAttack`就简单了, 并行站着不动和`FindBall(bb, look_down=True)`

这个`look_down`需要注意, 有可能这里会有问题

!!!

目前代码里, `bb.kicking`一直是`false`, 从来没更新过, 这是一个feature (bushi)

!!!

3 AttackBall

首先会先`set_state('ball_handling')`写入`behavior_info`, 声明自己在控球

底下有一个`announce_ball_handling()` 这个是空函数, 现在什么都不做

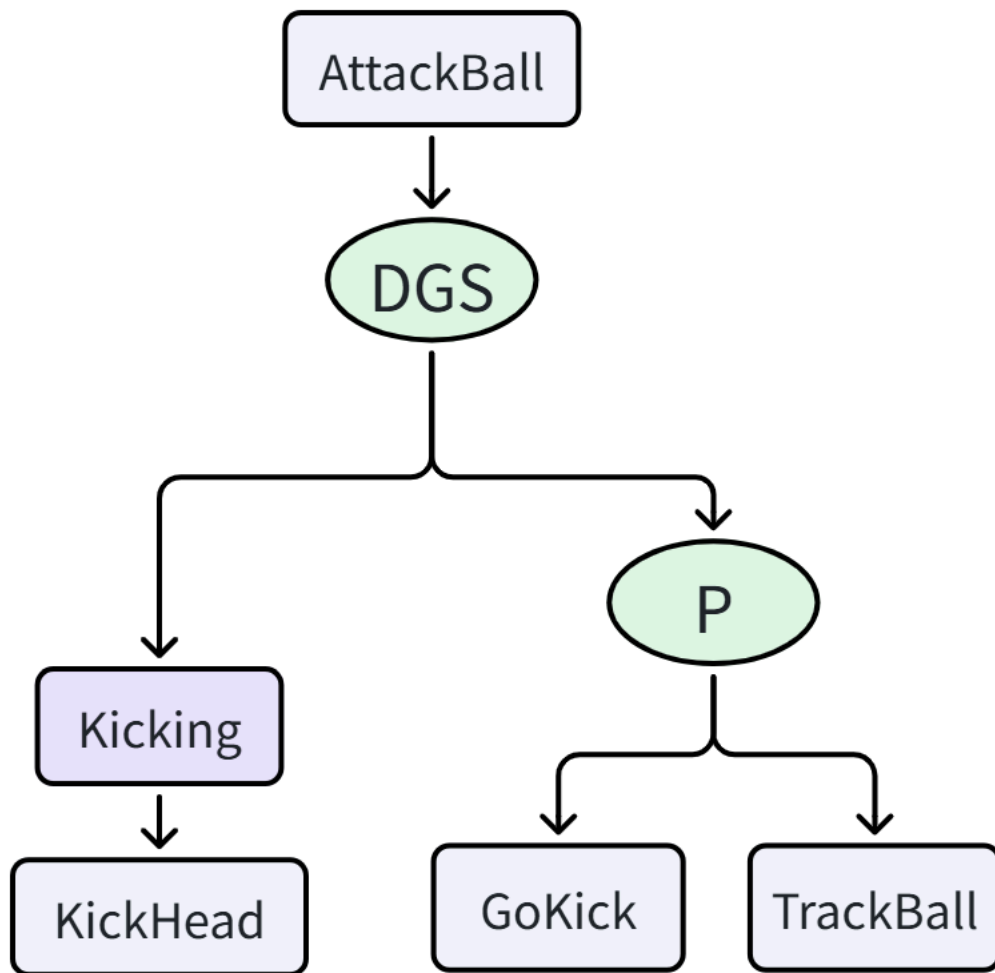
∨

复制代码

```
class AfterAttack(Parallel):

    def __init__(self, bb):
        args = (Do(bb, 'crouch'), FindBall(bb, look_down=True))
        super(AfterAttack, self).__init__(*args)

    def run(self):
        # Release ball handling after kicking ball
        self.get_bb().set_state('searching')
        super(AfterAttack, self).run()
```



注意哈，因为我们之前提到了，kicking一直是false，所以KickHead就一直没发生过

别看了，直接看P(GoKick, TrackBall)

TrackBall大家都很熟悉，不必再多介绍

GoKick



复制代码

```
class GoKick(Skill):
    """
    Go to the ball and kick.
    """

    def __init__(self, bb):
        super(GoKick, self).__init__(bb)
        self.kick_timer = Timer()

    def on_start(self):
        self.kick_timer = Timer()
        self.bb.kicking = False
```

```

        self.get_bb().target_used = True

    def on_end(self):
        super(GoKick, self).on_end()

    def execute(self):
        ball_global =
VecPos.from_vector3(self.bb.vision_info.ball_global)
        # Kick direction(global) of ball
        target = (self.bb.attack_target - ball_global).slope()
        enable_dribble = False
        robot_pos = VecPos.from_vector3(self.bb.vision_info.robot_pos)
        if self.bb.param.pos_role == 'striker':
            if self.bb.param.attack_right:
                enable_dribble = ball_global.x < -150
            else:
                enable_dribble = ball_global.x >= 150
        else:
            enable_dribble = False
        # here
        # self.bb.action_cmd.bodyCmd =
self.action_generator.dribble(ball_global.x, ball_global.y, target)

        if enable_dribble:
            self.bb.action_cmd.bodyCmd =
self.action_generator.dribble(ball_global.x, ball_global.y, target)
        else:
            self.bb.action_cmd.bodyCmd =
self.action_generator.kick_ball(ball_global.x, ball_global.y, target)

        return Status.RUNNING

```

这边是先从StrikerBrain里面拿到要攻击的目标点，然后根据这个目标点和球的位置计算出踢球的角度，然后根据目前球的位置，选择dribble（盘带）或者kick（踢球）

∨
📄 复制代码

```

if enable_dribble:
    self.bb.action_cmd.bodyCmd =
self.action_generator.dribble(ball_global.x, ball_global.y, target)
else:
    self.bb.action_cmd.bodyCmd =
self.action_generator.kick_ball(ball_global.x, ball_global.y, target)

```

现在是离球门近就是dribble，离球门远就是kick

比如dribble，具体体现为

[复制代码](#)

```
self.bb.action_cmd.bodyCmd = self.action_generator.dribble(  
    ball_global.x, ball_global.y, target)
```

然后发出的就是BodyCommand(ball_global.x, ball_global.y, target, DRIBBLE=3)

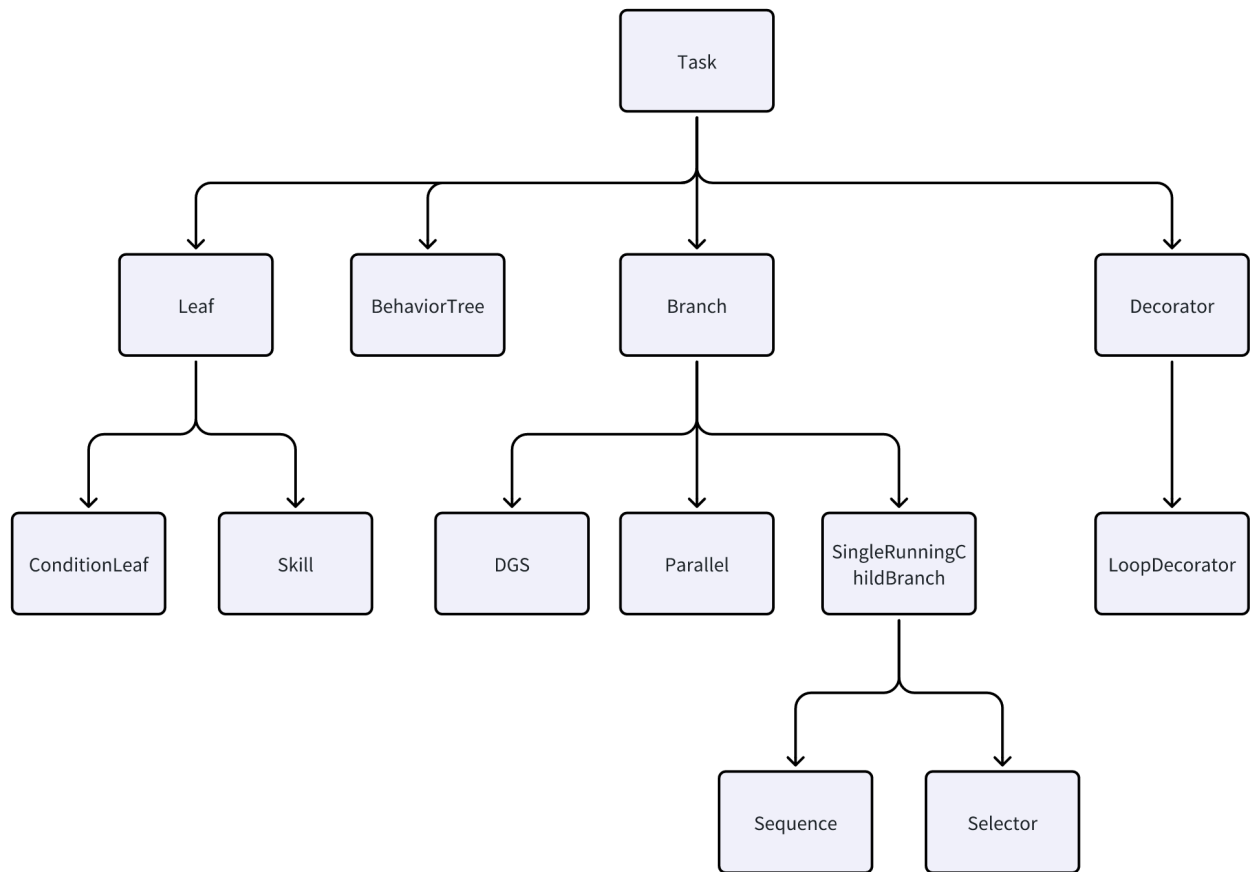
最后

```
bb.publish()    # 把 action_cmd 发出去
```

最后的最后，还必须再看最后一个东西

我们前面都是以Striker举例的，我们还没看GCStriker

继承关系一览



p.s.

1 python八股回忆

`super(Child, self).init(args)`

这个是Python中调用父类构造函数

2 行为树八股速查

guard 守卫 就是一个条件检查

行为树图例

任务（Task）是行为树的基本单位，行为树的任意子树（包括整棵树和任一叶子节点）都是一个任务。（可参考上面的“继承关系”）

