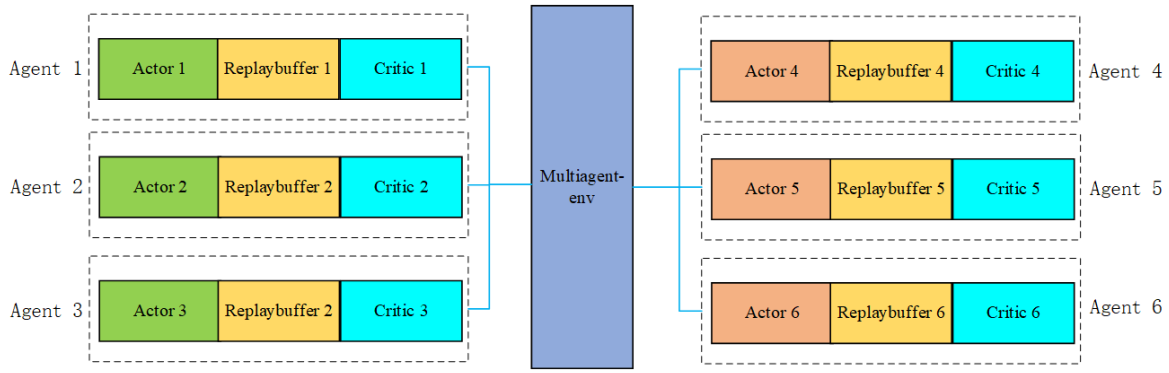
MADDPG Compitetion 3v3 模块化设计方案

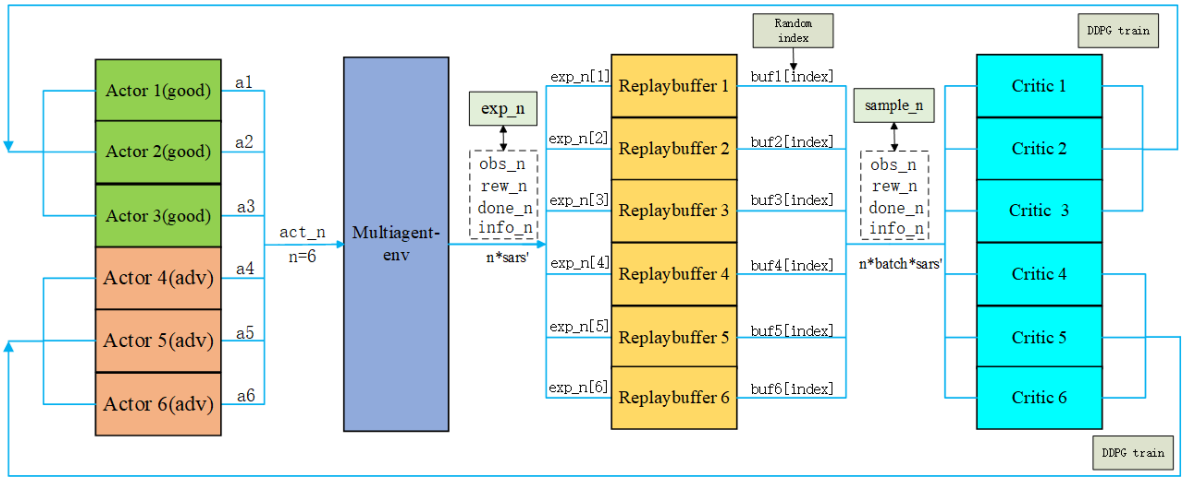
算法设计框架

基准代码的数据结构图：



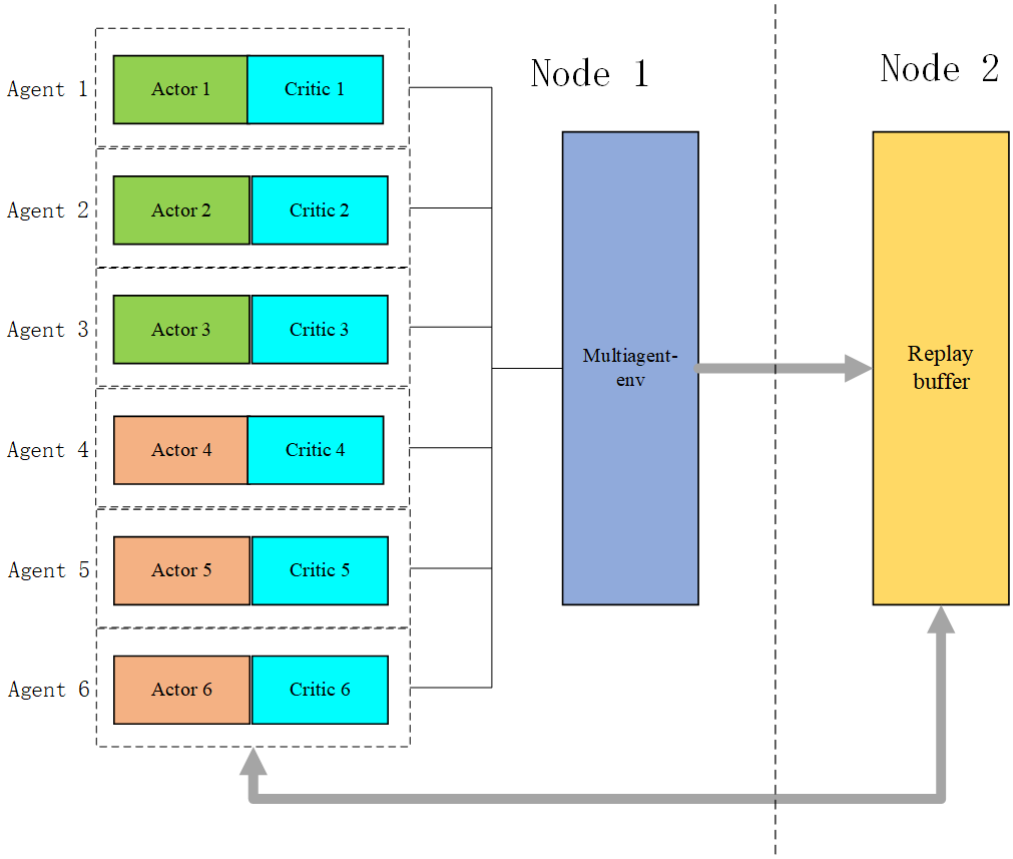
- 每个Agent包含了各自独立的Actor、Replaybuffer和Critic
- 每个Agent独立运行自己的DDPG算法对网络进行更新
- 多Agent与共享的Multiagent-env进行交互

原基准代码的数据流向图：



在基准代码中数据流向是：1、各自的Actor接受到env环境生成的obs，分别生成a[i]；2、将各自生成的a[i]拼接成act_n送到env后生成包含了obs_n,rew_n,done_n,info_n的experience；3、将experience拆开分别送入各个agent的replay buffer中；4、生成一个batch size的random index，从replay buffer里根据random index抽取sample_n；5、将抽取的sample_n复制n份，分别送入critic中；6、从Critic到Actor传递损失梯度，分别对两个网络进行DDPG模式的训练。

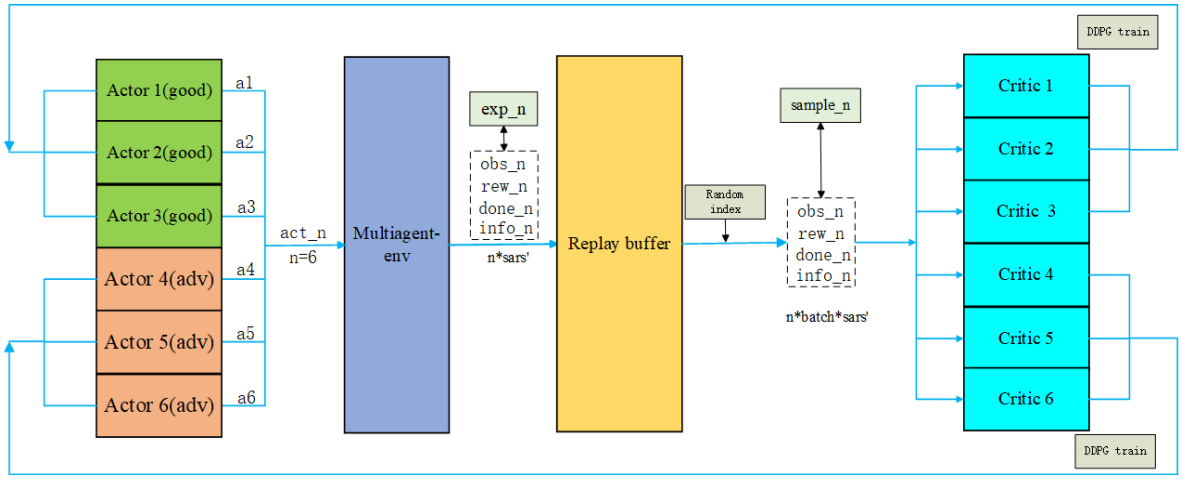
分离Replay buffer数据结构图



在原有的Agent类里面将Actor和Critic分别进行封装，将Replay buffer类的实例化从Agent类里面分离出来，env仍然使用共享结构。

单机版本Node2中的Replay buffer放置在单机内存区域，由Agents直接读取内存获取sampls；多机版本时，图中粗灰线部分表示需要用到节点间的通信。

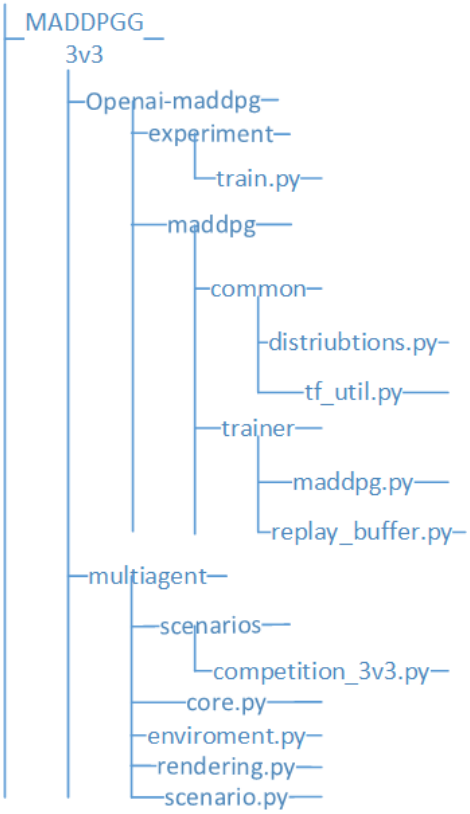
分离Replay buffer数据流向图



数据流向图大致与基准代码保持一致，不同点在于：

1. 从Multiagent-env交互生成的experience不再进行拆分，直接送到Replay buffer进行存储；
2. 从Replay buffer采样是也不需要对多个agent的样本进行重组，直接用random index采样得到sample_n并复制送入Agent各自的Critic中。

基准代码文件结构图



train.py

函数：make_env(scenario_name, arglist, benchmark=False)

功能：产生一个多agents的模拟环境env。scenario_name模拟环境名称，本程序中为"competition_3v3"，输入的初始化参数arglist在函数中并未用到，可以作为功能扩展备用。

返回：一个MultiAgentEnv的实例，MultiAgentEnv对象在environment.py中定义，由多个独立的agent env组成，每个agent分别对应3v3中的一个点

函数：get_trainers(env, num_adversaries, obs_shape_n, arglist)

功能：从env中读取agent数量，再根据env中的动作空间action_sapce和观察空间形状obs_space_n创建MADDPG的agent训练实例，训练实例对象由maddpg/trainer/maddpg.py文件定义。

返回：一个multiagent的训练实例对象列表trainers

函数：adversary_leave_screen(env)，green_leave_screen(env)

功能：判断双方agent是否越界，可以综合写在一个函数里is_leave_screen(env, agent_category)，根据参数agent_category选择计算good或者是adversary是否越界。

返回：判断是否越界的bool值

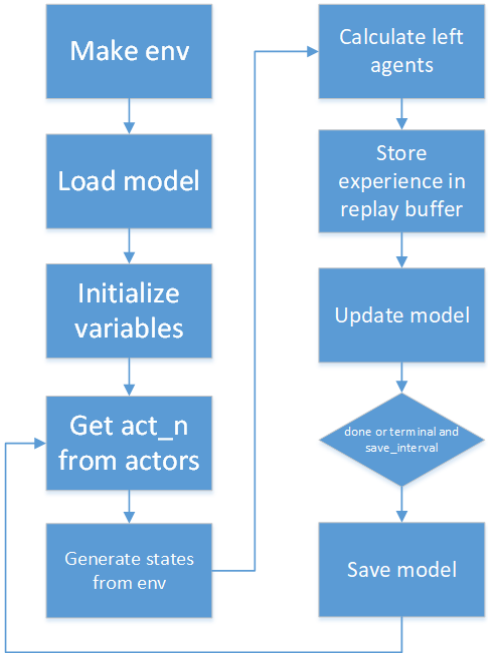
函数：adversary_all_die(env)

功能：判断所有的adversary节点是否都被消灭，可以同is_leave_screen(env, agent_category)函数一起放入到competition_3v3.py中，集成到env的类方法中，方便移植和使用。

返回：判断adversary节点是否都被消灭的bool值

函数：train(arglist)

功能：train.py中最重要的函数，完成模型训练的功能。其具体结构如下图所示：



由于原来的代码是面向过程的，上述框图展现的子功能都集中在train的函数里面，不利用功能的扩展和进行分布式训练使用。因此，计划将train函数中的各个子功能按上述结构图进行划分，分别对各个子功能进行函数的封装，以增强代码的可读性和可扩展性。

函数：load_model(arglist, load_dir=None)

功能：利用maddpg/common/tf_util.py中的initialize()h函数对模型参数进行初始化，若load_dir参数非空，则用load_state函数进行模型的加载。

函数：initialize_variables(arglist)

功能：初始化训练参数：train_step, start_time, episode_step, episode_rewards， agent_rewards. 将env相关的参数初始化放入competition_3v3的类里面进行初始化， 包括red_win， red_leave， green_win， green_leave， agent_death_index。

返回：初始化参数的列表

模块Get act_n from actors, Generate states from env, Calculate left agents等三个模块的功能由env类中加入相应的方法给出。

函数：replay_buffer.add_experience(obs_n, action_n, rew_n, new_obs_n, done_n, terminal)

功能：将obs_n, action_n, rew_n, new_obs_n, done_n, terminal添加到共享的replay buffer中，这一函数的功能由原MADDPGAgentTrainer拆分出来的replaybuffer类提供。原MADDPGAgentTrainer将replay_buffer与actor和critic封装在一个类里面，将其拆分出来，replaybuffer仍然由maddpg/trainer/replay_buffer.py中的类定义，不同点是这里的replay buffer存储的是n个agnets的状态向量，另外在这里加入与actor及critic的通信接口。

返回：存储成功标识bool值

函数：update_model(trainers)

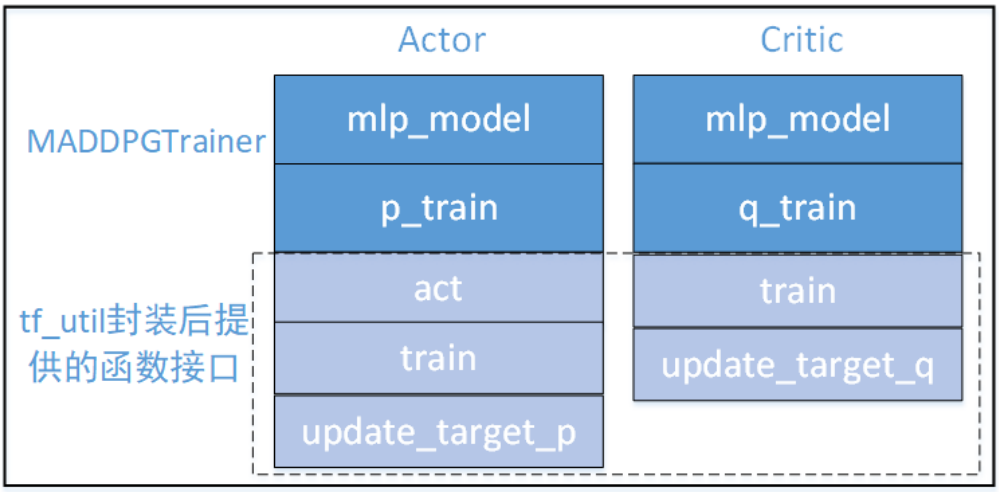
功能：传入agents对象的列表trainers，原代码trainer包含的actor和critic都在trainer的类里，修改后将actor和critic分离出来，再另外定义trainer的类，调用分离后的actor和critic的类，在做update_model操作时通过trainer逐一调用每个critic的preupdate()和update()方法更新网络参数，完成一次step的训练。

函数：save_model()

功能：保存网络模型。仍然用封装在tf_util中的函数save_state(fname)完成模型的保存，里面对tf模型的读取、加载和保存等基本功能都实现了很好的封装，有增强代码的复用性和正确性。

maddpg.py

maddpg仍然是完成算法训练的类，修改后与原代码的不同点是按照不同功能分成多个模块：



mlp_model：定义一个具有三层全连接网络的类，返回网络前向传播的tensor，原代码将其定义在train.py里面，修改后将其移动到maddpg.py里

p_train：定义了Actor类中生成action，训练网络和更新目标网络的方法，通过tf_util封装后分别提供act、train和update_target_p的函数接口，将封装后的函数接口作为Actor的类方法，提供外部调用。

q_train：定义Critic中训练网络和更新网络的方法，与p_train类似，同样通过tf_util进行封装。