Bot

From Botzone Wiki

Contents

何为Bot

Bot是Botzone上用户提交的程序，具有一定的人工智能，可以根据已有的游戏规则跟其他的Bot或人类玩家进行对抗。

运行环境

Botzone的评测机均为运行Ubuntu 16.04的x86-64架构虚拟机，且仅提供一个可用CPU核。

目前平台可以运行多线程程序，但是由于运行在单核环境下，因此并不会带来性能收益。

Bot可以读写用户存储空间下的文件。详情请在Botzone上点击头像菜单，选择“管理存储空间”进行了解。

支持的语言

Bot可以用以下语言编写：

- C/C++（平台上编译时会定义 **`_BOTZONE_ONLINE`** 宏）
- Java
- JavaScript
- C# (Mono)
- python2
- python3
- Pascal

针对 Python 多文件上传的提示

请参考以下链接：

http://blog.ablepear.com/2012/10/bundling-python-files-into-stand-alone.html

(如果无法访问，请参考BundleMultiPython)

简单来说，就是把python文件打包成zip，并在zip根目录额外包含一个`__main__.py`作为入口点，然后上传整个zip作为python源码。

注意：数据文件请不要打包，而是使用账户菜单里的“用户存储空间”功能上传，然后在程序中使用'data'路径访问。

针对 C++ 多文件上传的提示

请使用该工具：https://github.com/vinniefalco/Amalgamate。

该工具会将若干C++源代码文件合并成一个文件。

提供的运行库

- C/C++: 提供 JSONCPP（`#include "jsoncpp/json.h"`）、 nlohmann/json (https://github.com/nlohmann/json)（`#include "nlohmann/json.hpp"`）、 Eigen (http://eigen.tuxfamily.org/index.php?title=Main_Page)（`#include "Eigen/xx"`）
  - “带很多库”版的编译器里，提供 tensorflow_cc、libtorch (1.5.0) 库，以及 libboost 和 libopenblas，不过要注意如果选择了这个编译器，那么编译用到的 JSONCPP 会是最新版，和 Botzone 提供的稍有不同。
- Java: 提供 JSON.simple
- C#: 提供 Newtonsoft.Json (http://www.newtonsoft.com/json)
- python(2/3): 均提供 numpy, scipy, CPU 下的 TensorFlow、theano、pytorch(0.4.0，python 3.6 除外) 和 mxnet(0.12.0), 以及 keras(2.1.6)、lasagne、scikit-image 和 h5py
  - python 3.6 不支持 theano 和 lasagne
  - python 3.6 的 pytorch 版本是 1.8.0
  - python 3.6 的 mxnet 版本是 1.4.0

如果有更新库或者增加库的需求，请到讨论区发帖提醒管理员。

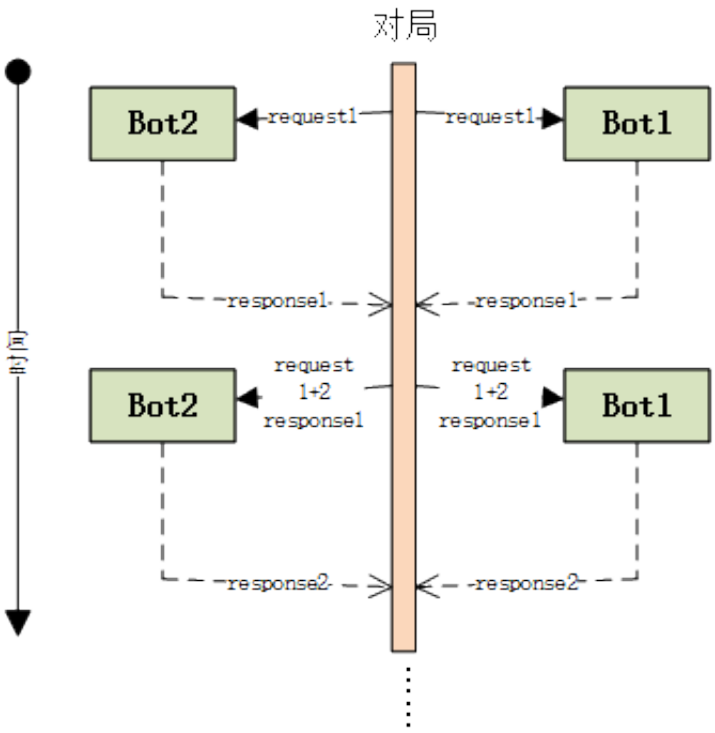
交互

Bot的每次生命周期均为一次输入、一次输出，可以参见右图，每次输入会包括该Bot以往跟平台的所有交互内容。交互格式为单行JSON或简化交互。

新增：现在已经支持“长时运行”模式，可以通过输出特定指令来保持Bot的持续运行，减少冷启动的开销。详见#长时运行。

因此，Bot的程序运行流程应当是：

- 启动程序
- 从平台获取以往的交互内容和最新输入
- 根据获得的交互内容（request + response）进行计算
  - 根据以往自己的输入（request）输出（response）恢复局面到最新状态
  - 根据本次输入（request）给出本次的决策输出
- 输出结果以及保存信息
- 关闭程序



资源和时间限制

如果没有特别指出，每次运行，平台都要求程序在 1秒 内结束、使用的内存在 256 MB 内。

每场对局，每个 Bot 的第一回合的时间限制会放宽到原来的两倍。

#长时运行模式下，除了程序刚刚启动的回合，其他回合的时间限制请参看提交 Bot 的页面。

由于不同语言运行效率有别，我们对不同语言的时限也有不同的调整。

- C/C++：1倍
- Java：3倍
- C#：6倍
- JavaScript：2倍
- python：6倍
- Pascal：1倍

交互格式

为了照顾初学者，交互形式有两种：JSON交互和简化交互。

提示：为了限制对局 Log 的大小，data、globaldata 域作为对局的中间变量，在对局结束后将不会被存入 Log 中。

如果希望能进行调试，请使用 debug 域，该域会在对局结束后仍保留在 Log 中。

JSON交互

使用这种交互，可能需要在自己的程序中增加JSON处理相关的库，具体请参见下方样例程序的说明。

Bot 得到的输入

实际上Bot得到的输入是单行紧凑的JSON。

```
{
   "requests" : [
      "Judge request in Turn 1", // 第 1 回合 Bot 从平台获取的信息（request），具体格式依游戏而定
      "Judge request in Turn 2", // 第 2 回合 Bot 从平台获取的信息（request），具体格式依游戏而定
      ...
   ],
   "responses" : [
      "Bot response in Turn 1", // 第 1 回合 Bot 输出的信息（response），具体格式依游戏而定
      "Bot response in Turn 2", // 第 2 回合 Bot 输出的信息（response），具体格式依游戏而定
      ...
   ],
```

```
    "data" : "saved data", // 上回合 Bot 保存的信息，最大长度为100KB【注意不会保留在 Log 中】
    "globaldata" : "globally saved data", // 来自上次对局的、Bot 全局保存的信息，最大长度为100KB【注意不会保留在 Log 中
    "time_limit" : "", // 时间限制
    "memory_limit" : "" // 内存限制
}
```

Bot 应该给出的输出

Bot应当给出的是单行紧凑的JSON。

```
{
    "response" : "response msg", // Bot 此回合的输出信息（response）
    "debug" : "debug info", // 调试信息，将被写入log，最大长度为1KB
    "data" : "saved data" // Bot 此回合的保存信息，将在下回合输入【注意不会保留在 Log 中】
    "globaldata" : "globally saved data" // Bot 的全局保存信息，将会在下回合输入，对局结束后也会保留，下次对局可以继续利用
}
```

简化交互

使用这种交互，只需使用标准输入输出按行操作即可。

Bot 得到的输入

1. 你的 Bot 首先会收到一行，其中只有一个数字 **n**，表示目前是第 n 回合（从 1 开始）。
2. 接下来，是 2 * n - 1 条信息，这些信息由 Bot 从平台获取的信息（request）与 Bot 以前每个回合输出的信息（response）交替组成。
    - 从 1 开始计数，(1 <= i < n)
        - 第 2 * i - 1 条信息为第 i 回合 Bot **从平台获取的信息（request）**，共 n - 1 条
        - 第 2 * i 条信息为 Bot 在第 i 回合**输出的信息（response）**，共 n - 1 条
        - 最后一条，即第 2 * n - 1 条信息，为当前回合 Bot **从平台获取的新信息（request）**
    - 每条信息可能是 1 行，也可能是多行，具体格式依游戏而定。
    - 你的 Bot 需要根据以上信息，推演出当前局面，并给出当前局面下的最好决策。
3. 接下来是**data**，一行 Bot 上回合保存的信息，最大长度为100KB【注意不会保留在 Log 中】。
4. 接下来是**globaldata**，**一行或多行** Bot 上回合或上次对局保存的全局信息，最大长度为100KB【注意不会保留在 Log 中】。
    - 可以认为 data 行之后的所有内容都是 globaldata，直到文件结尾。

样例输入

```
3
第一回合游戏信息（request），即 Bot 当时得到的输入
第一回合 Bot 的输出（response）
第二回合游戏信息
第二回合 Bot 的输出
第三回合游戏信息
Bot 上回合保存了这句话作为data!
Bot 上次运行程序保存了这句话作为globaldata!
```

Bot 应该给出的输出

你的 Bot 应当输出四段数据，以换行符隔开。

1. 首先是本回合你的 Bot 做出的决策，请按照游戏要求输出。一定只占一行。
2. 接下来是debug，一行用于回放时候进行调试的信息，最大长度为1KB【注意会保留在 Log 中】。
3. 接下来是data，一行 Bot 本回合想保存的信息，将在下回合输入，最大长度为100KB【注意不会保留在 Log 中】。
4. 接下来是globaldata，一行或多行 Bot 想保存的全局信息，将会在下回合输入，对局结束后也会保留，最大长度为100KB【注意不会保留在 Log 中】。

样例输出

```
本回合 Bot 的输出
Bot 这回合想保存这句话作为debug，到时候回放看!
Bot 这回合保存了这句话作为data，下回合想用!
Bot 这次运行程序保存了这句话作为globaldata，以后想用!
```

JSON交互的样例程序

以下给出C++、C#、Java、python和JavaScript的JSON交互样例程序：

C++

本地编译的方式请查看 JSONCPP。

```cpp
#include <iostream>
#include <string>
#include <sstream>
#include "jsoncpp/json.h" // C++编译时默认包含此库

using namespace std;
```

```cpp
int main()
{
    // 读入JSON
    string str;
    getline(cin, str);
    Json::Reader reader;
    Json::Value input;
    reader.parse(str, input);

    // 分析自己收到的输入和自己过往的输出，并恢复状态
    string data = input["data"].asString(); // 该对局中，以前该Bot运行时存储的信息
    int turnID = input["responses"].size();
    for (int i = 0; i < turnID; i++)
    {
        istringstream in(input["requests"][i].asString()),
            out(input["responses"][i].asString());

        // 根据这些输入输出逐渐恢复状态到当前回合
    }

    // 看看自己本回合输入
    istringstream in(input["requests"][turnID].asString());

    // 做出决策存为myAction

    // 输出决策JSON
    Json::Value ret;
    ret["response"] = myAction;
    ret["data"] = myData; // 可以存储一些前述的信息，在整个对局中使用
    Json::FastWriter writer;
    cout << writer.write(ret) << endl;
    return 0;
}
```

C#

本地编译的方式请查看 Newtonsoft.Json (http://www.newtonsoft.com/json)。

```cpp
using System;
using Newtonsoft.Json;

struct InputData
{
    public dynamic[] // 类型请根据具体游戏决定
        requests, // 从平台获取的信息集合
        responses; // 自己曾经输出的信息集合
    public string
        data, // 该对局中，上回合该Bot运行时存储的信息
        globaldata; // 来自上次对局的、Bot全局保存的信息
    public int time_limit, memory_limit;
}

struct OutputData
{
    public dynamic response; // 此回合的输出信息
    public string
        debug, // 调试信息，将被写入log
        data, // 此回合的保存信息，将在下回合输入
        globaldata; // Bot的全局保存信息，将会在下回合输入，对局结束后也会保留，下次对局可以继续利用
}

class Program
{
```

```
    static void Main(string[] args) // 请保证文件中只有一个类定义了Main方法
    {
        // 将输入解析为结构体
        var input = JsonConvert.DeserializeObject<InputData>(
            Console.ReadLine()
        );

        // 分析自己收到的输入和自己过往的输出，并恢复状态
        int turnID = input.responses.Length;
        for (int i = 0; i < turnID; i++)
        {
            dynamic inputOfThatTurn = input.requests[i], // 当时的输入
                outputOfThatTurn = input.responses[i]; // 当时的输出

            // 根据这些输入输出逐渐恢复状态到当前回合
        }

        // 看看自己本回合输入
        dynamic inputOfCurrentTurn = input.requests[turnID];

        // 做出决策存为myAction

        // 输出决策JSON
        OutputData output = new OutputData();
        output.response = myAction;
        output.debug = "hhh";
        Console.WriteLine(
            JsonConvert.SerializeObject(output)
        );
    }
}
```

JavaScript

```
// 初始化标准输入流
var readline = require('readline');
process.stdin.resume();
process.stdin.setEncoding('utf8');

var rl = readline.createInterface({
    input: process.stdin
});

rl.on('line', function (line) {
    // 解析读入的JSON
    var input = JSON.parse(line);
    var data = input.data; // 该对局中，以前该Bot运行时存储的信息

    // 分析自己收到的输入和自己过往的输出，并恢复状态
    for (var i = 0; i < input.responses.length; i++) {
        var myInput = input.requests[i], myOutput = input.responses[i];
        // 根据这些输入输出逐渐恢复状态到当前回合
    }

    // 看看自己本回合输入
    var currInput = input.requests[input.requests.length - 1];

    var myAction = {}, myData = {};

    // 作出决策并输出
    process.stdout.write(JSON.stringify({
        response: myAction,
        data: myData // 可以存储一些前述的信息，在整个对局中使用
    }));
```

```
    // 退出程序
    process.exit(0);
});
```

python

仅提供python3的样例。

```
import json

# 解析读入的JSON
full_input = json.loads(input())
if "data" in full_input:
    my_data = full_input["data"]; # 该对局中，上回合该Bot运行时存储的信息
else:
    my_data = None

# 分析自己收到的输入和自己过往的输出，并恢复状态
all_requests = full_input["requests"]
all_responses = full_input["responses"]
for i in range(len(all_responses)):
    myInput = all_requests[i] # i回合我的输入
    myOutput = all_responses[i] # i回合我的输出
    # TODO: 根据规则，处理这些输入输出，从而逐渐恢复状态到当前回合
    pass

# 看看自己最新一回合输入
curr_input = all_requests[-1]

# TODO: 作出决策并输出
my_action = { "x": 1, "y": 1 }

print(json.dumps({
    "response": my_action,
    "data": my_data # 可以存储一些前述的信息，在该对局下回合中使用，可以是dict或者字符串
}))
```

Java

本地编译的方式请查看 JSON.simple。

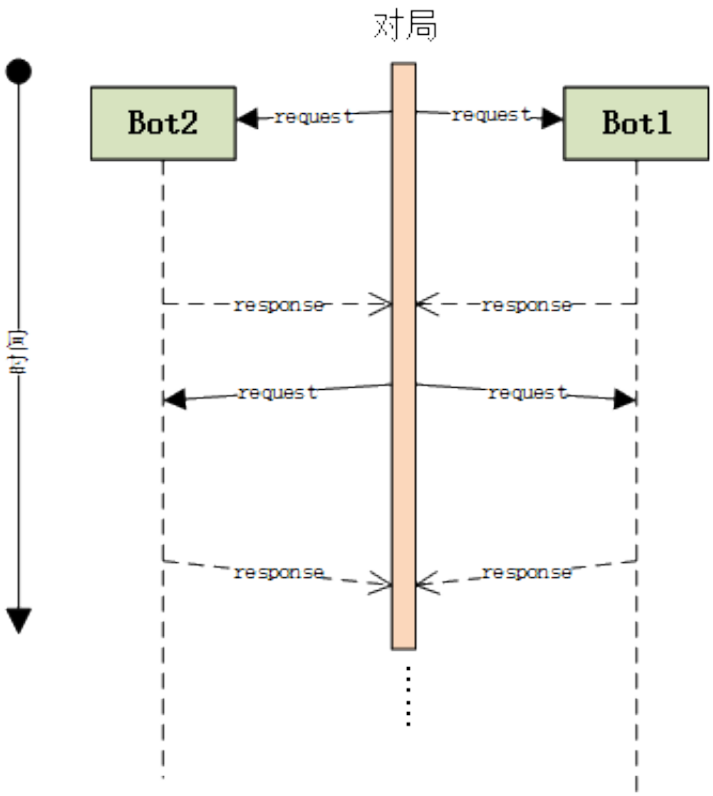
```
import java.util.*;
import org.json.simple.JSONValue; // Java 库中自动包含此库

class Main { // 注意！！类名""必须""为 Main，且不要有package语句，但上传的 java 代码文件名可以任意
    static class SomeClass {
        // 如果要定义类，请使用 static inner class
    }
    public static void main(String[] args) {
        String input = new Scanner(System.in).nextLine();
        Map<String, List> inputJSON = (Map) JSONValue.parse(input);
        // 下面的 TYPE 为单条 response / request 类型，有较大可能为 Map<String, Long> 或 String
        List<TYPE> requests = inputJSON.get("requests");
        List<TYPE> responses = inputJSON.get("responses");
        for (TYPE rec : requests) {
            // 处理一下
        }
        for (TYPE rec : responses) {
            // 再处理一下
        }
        // 这边运算得出一个 output，注意类型也为 TYPE
```

```
        Map<String, TYPE> outputJSON = new HashMap();
        outputJSON.put("response", output);
        System.out.print(JSONValue.toJSONString(outputJSON));
    }
}
```

长时运行

如果希望Bot在回合结束后不必退出，从而避免每次重启Bot造成的额外开销（比如神经网络的初始化过程），则可以选择此种交互方式，模式请参见右图。

该模式尚在测试中，可能会有非预期的问题，敬请留意，如果遇到问题请联系管理员或者在讨论区发帖，谢谢支持！



启用方法

要开启长时运行模式，需要至少先使用正常的交互方式完成一个回合。（可以认为该回合的输入是只有一个request的输入，且是有限行的）

在该回合结束时，**先按前文所述的正常交互方式输出**，再通过标准输出方式输出如下内容的一行（注意前后均应有换行符）：

```
>>>BOTZONE_REQUEST_KEEP_RUNNING<<<
```

建议在输出后清空标准输出缓冲区（如`fflush(stdout);`、`cout << flush;`、`sys.stdout.flush()`等）。

输出该行，意味着本回合的输入结束。

启用后会发生什么

在Botzone收到如上输入后，会认为本回合Bot已经完成输出，因此会将Bot强制休眠（发送SIGSTOP信号到进程组），在下一回合轮到自己的时候再唤醒（发送SIGCONT到进程组）。

休眠Bot的过程由于延迟，很可能不是在输出那行内容以后立刻进行，而是在Bot等待下回合输入的过程中进行。**需要卡时的话，请注意在input()或getline()等操作后再开始卡时。**

开启该模式后：

- **下一回合收到的输入将只有一回合的游戏信息（request），不再有历史数据（之前回合的request和response）和data等信息**
- debug、data、globaldata功能将失效（请勿在使用简单IO时，向globaldata中存储不定多行的数据）
- Bot的输出格式无须变化
- 在回合之间还未轮到自己时，如果Bot在占用CPU（比如多线程提前计算一些信息），这部分计算所使用的CPU时间会计入下回合的时间，因此请不要这么做

需要连续运行多个回合时

如果需要维持该模式，此后每一回合输出response之后，都需要输出上述的一行命令。

如果在开启该模式后，某一回合输出完成后没有输出上述的一行，那么平台就不知道Bot的输出是否结束，因此超时。

如果你的需求十分特殊，希望在连续运行多个回合后关闭程序，然后再启动，那么你需要在希望关闭程序的回合自行了断（在输出决策response后，不输出上述命令，而是直接退出）。这样，下一回合你的程序会像传统运行方式一样，重新启动。

调试

开启该模式后，Botzone提供的对局Log里的“玩家输入”会做出相应变化，可以直接从平台复制进行调试。

不过要注意，该调试方法会假设你的**Bot的输出和平台上完全一致**（即**相同局面下平台上和本地调试的决策均相同**），如果你的Bot有随机成分，那么将无法正确进行调试。

为了回溯到某一个回合，你的Bot调试时可能需要重新计算之前回合的动作，因此可能会演算很久。
