

エージェントの作り方

目次

1. エージェント作成前の準備
2. BasePlayerの作成
3. 各役職プレイヤーの作成（村人の場合）
4. 会話の取得方法
5. 発話の生成方法
6. 占い、霊能情報の取得
7. ゲームの流れ

1. エージェント作成前の準備

「Artificial Intelligence based Werewolf（<http://www.aiwolf.org/>）」のページにアクセスし、「資料 > 人狼知能サーバ」から人狼知能サーバの最新版をダウンロードする（2014/07/04時点ではver0.1.7）。



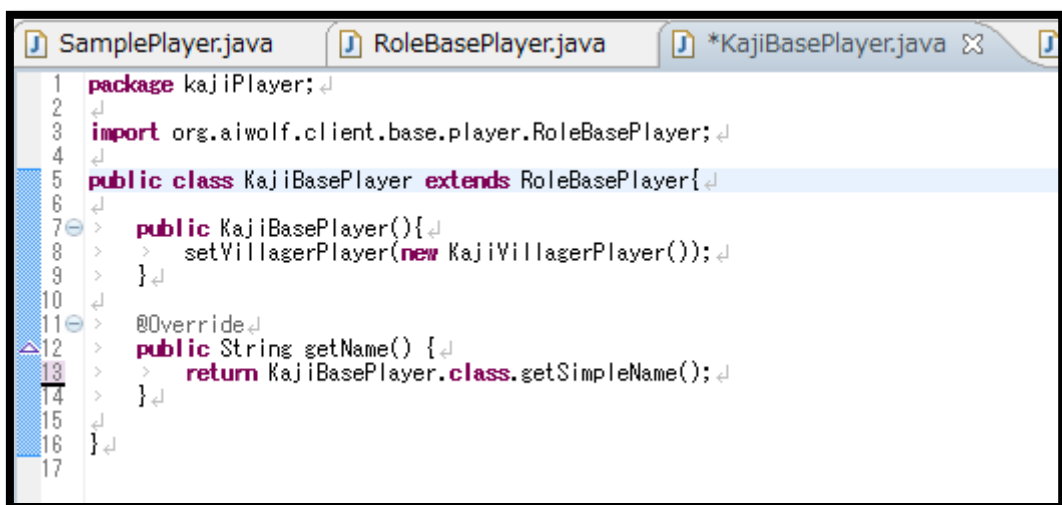
人狼知能サーバのダウンロード

ダウンロードしたファイル内の「jarsファイル」を開くと人狼エージェントを作成するのに必要な4つのjarファイルが入っている。eclipseでエージェントを作成する場合にはエージェントを作成するプロジェクトのビルドパスにこれらのjarファイルを加える。また、「docsファイル」には各クラスのjavadocが入っている。

2. BasePlayerの作成

人狼ゲームでは役職によって戦略が異なることと、役職ごとに実装すべきメソッドが異なることから、基本的には各役職ごとのエージェントを作成することになる。org.aiwolf.client.base.player.RoleBasePlayerは、ゲーム開始時にサーバから役職を与えられると、事前にセットしたその役職のエージェントでプレイするようになっているため、このクラスを継承したエージェントを作成すると良い。各役職のエージェントは、コンストラクタ内で「setOOOPlayer(セットしたいPlayerのインスタンス);」という様にセットする。(下図参照)

例として、KajiBasePlayerクラスを作成する。ここでは役職が村人の時にKajiVillagerPlayerクラスを起動するプレイヤーとしている。他の役職のエージェントも同時にセットすることは可能であり、セットしていない役職のプレイヤーについては、org.aiwolf.client.base.smpパッケージ内の各役職のサンプルプレイヤーが初期状態としてセットされている。また、getName()メソッドをオーバーライドしなければならないが、これはエージェントを示す名前なので好きな名前を返すようにすれば良い。



```
1 package kajiPlayer;
2
3 import org.aiwolf.client.base.player.RoleBasePlayer;
4
5 public class KajiBasePlayer extends RoleBasePlayer{
6
7     public KajiBasePlayer(){
8         > setVillagerPlayer(new KajiVillagerPlayer());
9     }
10
11     @Override
12     public String getName() {
13         > return KajiBasePlayer.class.getSimpleName();
14     }
15
16 }
17
```

各役職のセット方法

3. 各役職プレイヤーの作成（村人の場合）

では、実際に各役職のエージェントを実装してみる。ここではKajiBasePlayerが村人のエージェントとしてセットしていたKajiVillagerクラスを実装する。村人の場合、占いや襲撃のメソッドを実装する必要はなく、実装しなければならないメソッドは、

- update(GameInfo)
- initialize()
- dayStart()
- talk()
- vote()
- finish()

の6つである(それぞれのメソッドが呼ばれるタイミングは下記の「7.ゲームの流れ」を参照のこと)。`org.aiwolf.client.base.player.AbstractVillagerPlayer`クラスはこれら以外のメソッドはエラーを返し、また`update(GameInfo)`メソッドと`initialize()`メソッドを簡単に実装してある抽象クラスである。`KajiVillager`クラスではこのクラスを継承する。

➤ `update(GameInfo)`と`initialize()`の実装内容

`AbstractVillagerPlayer` クラスは全役職において必要であろうメソッドを組み込んだ`AbstractPlayer`クラスを継承している。このクラスでは`initialize()`時に下図のフィールドの`me`(自分を表すAgentインスタンス)、`myRole`(自分の役職)フィールドをセットし、`update(GameInfo)`が呼ばれる際に、`gameInfoMap`(各日のGameInfo)、`day`(ゲーム内での現在の日にち)を更新する。

```
11 public abstract class AbstractPlayer implements Player{
12     <
13     > //Index:day, content:GameInfo Mapで
14     > Map<Integer, GameInfo> gameInfoMap = new HashMap<Integer, GameInfo>();
15     <
16     > int day;
17     <
18     > Agent me;
19     <
20     > Role myRole;
21     <
```

AbstractPlayerのフィールド

`day`, `me`, `myRole`の値についてはそれぞれ、`getDay()`, `getMe()`, `getMyRole()`というメソッドで取得でき、`gameInfoMap`は

- `getGameInfoMap()` : Map全体を取得
 - `getGameInfo(int)` : 指定した日にちのgameInfoを取得
 - `getLatestDayGameInfo` : 現在の日にちのgameInfoを取得
- というメソッドで取得できる。

`AbstractVillagerPlayer`を継承し(下図)、実装したい戦略に沿って必要なメソッドを実装する。

```

1 package KajiPlayer;
2
3 import org.aiwolf.client.base.player.AbstractVillagerPlayer;
4 import org.aiwolf.common.data.Agent;
5
6 public class KajiVillager extends AbstractVillagerPlayer{
7
8     @Override
9     public void dayStart() {
10         // TODO 自動生成されたメソッド・スタブ
11     }
12
13     @Override
14     public String talk() {
15         // TODO 自動生成されたメソッド・スタブ
16         return null;
17     }
18
19     @Override
20     public Agent vote() {
21         // TODO 自動生成されたメソッド・スタブ
22         return null;
23     }
24
25     @Override
26     public void finish() {
27         // TODO 自動生成されたメソッド・スタブ
28     }
29
30 }
31
32
33

```

Abstract○○Playerを継承したプレイヤー

4. 会話の取得方法

update(GameInfo)で取得するGameInfoにはその日の会話のログも含まれており、GameInfo.getTalkList()でList<Talk>を取得できる。Talkクラスで取得できる内容は次の通り。

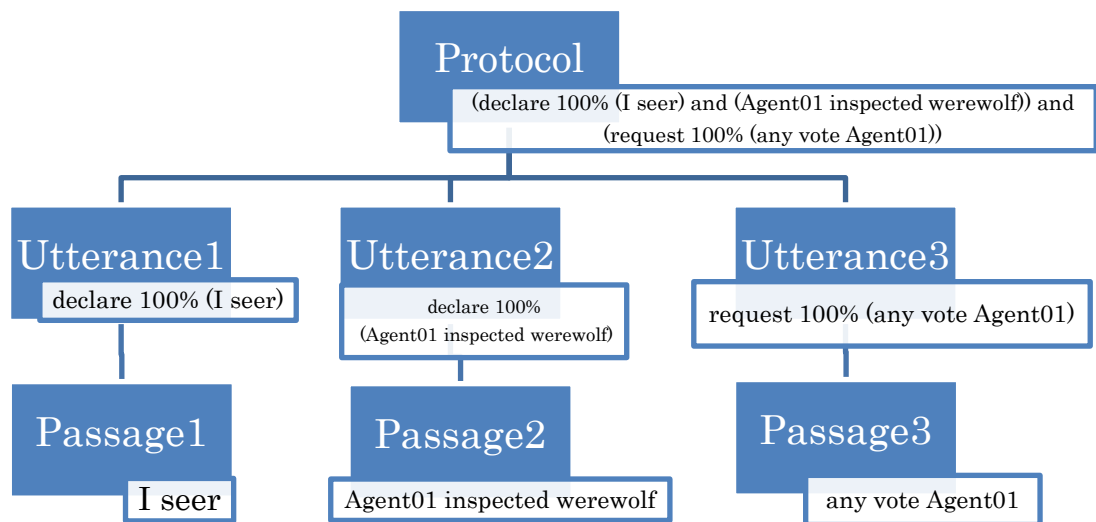
```

int day = talk.getDay(); // 発話された日にちの取得
int idx = talk.getIdx(); // 発話のID(その日の何番目の発話か)を取得
Agent agent = talk.getAgent(); // 発話者の取得
String content = talk.getContent(); // 発話の内容の取得

```

Talkクラスで取得できる値

contentには人狼プロトコルで書かれた文章が入っているため、中身を理解するためにはパースする必要がある。org.aiwolf.client.libパッケージにあるProtocolクラス、Utteranceクラス、Passageクラスは発話のパースのためのクラスであり、例えば、**「私は占い師であり、Agent01は人狼であった。みんなAgent01に投票してほしい。」**という発話は以下のような構造で表される。



Protocol, Utterance, Passageで表される発話の構造

すなわち,

- **Passage** : 「自分 = 占い師」や「みんながAgent01に投票する」といった事象のみを表す文(基礎文)
- **Utterance** : 基礎文の内容を宣言(declare)するのか, 要求(request)するのかといった情報を付加した文(発話文)
- **Protocol** : 発話文の集合で, 発話全体を表す.

というように説明できる (詳しくは, <http://www.aiwolf.org/>内の人狼プロトコル参照). 実際にこれらのクラスを使って, Passage2の「誰が占われ, その結果が何であったか」という情報を取得すると次のようになる.

```

String content = talk.getContent();
Protocol protocol = new Protocol(content);
Utterance utterance2 = protocol.getUtterances().get(1);
Passage passage2 = utterance2.getPassage();

// 占われた対象の取得
Agent target = passage2.getSubject();

// 占い結果の取得
Species species = passage2.getAttribution();
  
```

発話から情報を取得する方法

ProtocolクラスのコンストラクタにString型の発話を引数として入れると, 自動的にUtteranceやPassageが作られる. Protocol.getUtterances()でList<Utterance>を取得することができ, Utterance.getPassage()で基礎文を取得できる. この例ではPassage.getSubject()で占われた対象を取得し, Passage.getAttribution()で占いの結果を取得しているが, その他の情報を取得するメソッドについてはjavadoc参照.

発話の取得に関して, 1日の発話ログを取り出し, それぞれの発話のカテゴリー

(COMINGOUTやRESULT等)ごとに処理を行うサンプルコードを記載する。

```
List<Talk> talkList = getLatestDayGameInfo().getTalkList();  
for(Talk talk: talkList){  
    > //TalkからProtocolの生成  
    > Protocol protocol = new Protocol(talk.getContent());  
    > for(Utterance utterance: protocol.getUtterances()){  
    >     > Passage passage = utterance.getPassage();  
    >     > switch (passage.getCategory()) {  
    >     > case COMINGOUT:  
    >     >     > //カミングアウトの発話の場合の処理  
    >     >     > break;  
    >     > case ESTIMATE:  
    >     >     > //推測に関する発話の場合の処理  
    >     >     > case RESULT:  
    >     >     >     > if(passage.getVerb() == Verb.inspected){  
    >     >     >     >     > //占い結果の発話の場合の処理  
    >     >     >     >     > }else if(passage.getVerb() == Verb.medium_telled){  
    >     >     >     >     > //霊能結果の発話の場合の処理  
    >     >     >     >     > }  
    >     >     > default:  
    >     >     >     > break;  
    >     > }  
    > }  
}
```

カテゴリーごとに処理を行うサンプルコード

5. 発話の生成方法

次に自分が発話を行う方法の説明をする。先に述べたとおり、このゲームは人狼プロトコルを用いて進行するが、org.aiwolf.client.lib.TemplateTalkFactoryクラスを用いれば人狼プロトコルを知らずとも発話の生成が可能である。現段階で生成可能な発話は7種類。

- ① estimate(Agent agent, State state) : 「agentはstateだと思う。」
- ② comingout(Agent agent, State role) : 「agentがroleだとカミングアウトする。」
- ③ inspected(Agent agent, Species species) : 「agentを占った結果, speciesだった。」
- ④ medium_telled(Agent agent, Species species) : 「霊能の結果, agentはspeciesだった。」
- ⑤ guarded(Agent agent) : 「agentを護衛した。」
- ⑥ skip() : 様子見(発話することは無いが, まだ会話は終わらせたくない)。
- ⑦ over() : もう発話しなくて良い。

これらのメソッドでUtteranceクラスを生成できる。サーバに発話を送る時にはString型で送らなければならないため、以下のような手順でString型に変換する。

[1] TemplateTalkFactoryクラスでUtteranceを作成(複数の発話なら複数個)

[2] ProtocolクラスのコンストラクタにUtterance, またはList<Utterance>を引数として入れてProtocolクラスのインスタンスを作成

[3] Protocol.getText()でString型に変換してサーバに送る.

例えば, 占い師が自分の役職をカミングアウトする場合は次のように発話を生成する.

```
//カミングアウトの発話のサンプルコード
Agent me = getMe();//自分(Agent型)の取得
Role role = getMyRole();//自分の役職の取得
Utterance utterance = TemplateTalkFactory.comingout(me, role);
Protocol protocol = new Protocol(utterance);
String talkContent = protocol.getText();//サーバに返すべき値
```

カミングアウトの発話のサンプルコード

6. 占い, 霊能情報の取得

占い師(または霊能者)でプレイする場合, update(GameInfo)で得られる情報の中に, 占い結果(または霊能結果)が入っており, GameInfo.getDivineResult() (霊能結果は, GameInfo.getMediumResult) で取得できる. 占い結果と霊能結果はJudgeクラスで表されており, 下図の様に各情報を取得できる.

```
//占った日にち
int day = judge.getDay();
//占いをを行ったプレイヤー
Agent agent = judge.getAgent();
//占われたプレイヤー
Agent target = judge.getTarget();
//占い結果
Species result = judge.getResult();
```

Judgeクラスから取得できる情報

例えば, 霊能者が自分の役職をカミングアウトし, 今日の霊能結果を報告する場合は次のように発話を生成する.

```
Agent me = getMe();//自分(Agent型)の取得
Role role = getMyRole();//自分の役職の取得
Judge judge = getLatestDayGameInfo().getMediumResult();

List<Utterance> utterances = new ArrayList<Utterance>();

//カミングアウトの発話文
utterances.add(TemplateTalkFactory.comingout(me, role));

//霊能結果報告の発話文
utterances.add(TemplateTalkFactory.medium_telled(judge.getTarget(), judge.getResult()));

//発話の生成
Protocol protocol = new Protocol(utterances);

String talkContent = protocol.getText();//サーバに返すべき値
```

COと霊能結果報告の発話のサンプルコード

7. ゲームの流れ

Ver0.1.5における人狼サーバでは下記の流れでゲームを行う。

