WrightEagle 底层代码介绍

柏爱俊

中国科学技术大学
计算机科学与技术学院
多智能体系统实验室

2010 年 9 月 26 日

主要内容

主要内容

信息 — 决策的基础

- Client 实时直接从 Server 处获得各种原始信息，这些信息经处理和不同程度的抽象后得到一系列可供决策使用的数据，可以说信息是决策的基础。不同的信息更新和组织策略会导致不同的决策模式。
- 比如，如果将关于“自己”的概念绑定到不同层次的决策数据里面，就很难方便地实现计算队友和对手的决策 (WE2008 及以前的版本)。
- 下面首先介绍一下 WrightEagleBASE-2.0.1 中对信息的更新和组织策略。

信息 — 决策的基础

- Client 实时直接从 Server 处获得各种原始信息，这些信息经处理和不同程度的抽象后得到一系列可供决策使用的数据，可以说信息是决策的基础。不同的信息更新和组织策略会导致不同的决策模式。
- 比如，如果将关于“自己”的概念绑定到不同层次的决策数据里面，就很难方便地实现计算队友和对手的决策 (WE2008 及以前的版本)。
- 下面首先介绍一下 WrightEagleBASE-2.0.1 中对信息的更新和组织策略。

信息 — 决策的基础

- Client 实时直接从 Server 处获得各种原始信息，这些信息经处理和不同程度的抽象后得到一系列可供决策使用的数据，可以说信息是决策的基础。不同的信息更新和组织策略会导致不同的决策模式。
- 比如，如果将关于“自己”的概念绑定到不同层次的决策数据里面，就很难方便地实现计算队友和对手的决策 (WE2008 及以前的版本)。
- 下面首先介绍一下 WrightEagleBASE-2.0.1 中对信息的更新和组织策略。

Parser & Observer

- Parser 主要负责解析 Server 发送过来的“字符串”，提取出 Client 的感知 (Perception)，并直接转存到 Observer 里面。同时，Parser 还负责跟 Server 取得连接，并发送初始化信息。在 WrightEagle-BASE 里面，Parser 以一个单独的线程运行，除非 Server 发来信息，否则一直处于等待状态。
- Observer 从 Parser 获得的感知主要包括：视觉、听觉和自身感知等信息。Observer 是 Parser 线程和主线程的共享数据，需要做好互斥保护。
- Parser 和 Observer 涉及的信息是相对信息且过于低级，不跟高层决策打交道，要供决策使用还需进一步处理。
- Parser.{h, cpp}, Observer.{h, cpp}

Parser & Observer

- Parser 主要负责解析 Server 发送过来的“字符串”，提取出 Client 的感知 (Perception)，并直接转存到 Observer 里面。同时，Parser 还负责跟 Server 取得连接，并发送初始化信息。在 WrightEagle-BASE 里面，Parser 以一个单独的线程运行，除非 Server 发来信息，否则一直处于等待状态。
- Observer 从 Parser 获得的感知主要包括：视觉、听觉和自身感知等信息。Observer 是 Parser 线程和主线程的共享数据，需要做好互斥保护。
- Parser 和 Observer 涉及的信息是相对信息且过于低级，不跟高层决策打交道，要供决策使用还需进一步处理。
- Parser.{h, cpp}, Observer.{h, cpp}

Parser & Observer

- Parser 主要负责解析 Server 发送过来的“字符串”，提取出 Client 的感知 (Perception)，并直接转存到 Observer 里面。同时，Parser 还负责跟 Server 取得连接，并发送初始化信息。在 WrightEagle-BASE 里面，Parser 以一个单独的线程运行，除非 Server 发来信息，否则一直处于等待状态。
- Observer 从 Parser 获得的感知主要包括：视觉、听觉和自身感知等信息。Observer 是 Parser 线程和主线程的共享数据，需要做好互斥保护。
- Parser 和 Observer 涉及的信息是相对信息且过于低级，不跟高层决策打交道，要供决策使用还需进一步处理。
- Parser.{h, cpp}, Observer.{h, cpp}

Parser & Observer

- Parser 主要负责解析 Server 发送过来的“字符串”，提取出 Client 的感知 (Perception)，并直接转存到 Observer 里面。同时，Parser 还负责跟 Server 取得连接，并发送初始化信息。在 WrightEagle-BASE 里面，Parser 以一个单独的线程运行，除非 Server 发来信息，否则一直处于等待状态。
- Observer 从 Parser 获得的感知主要包括：视觉、听觉和自身感知等信息。Observer 是 Parser 线程和主线程的共享数据，需要做好互斥保护。
- Parser 和 Observer 涉及的信息是相对信息且过于低级，不跟高层决策打交道，要供决策使用还需进一步处理。
- Parser.{h, cpp}, Observer.{h, cpp}

WorldState

- 将从 Observer 得到的相对信息换算成全局信息，并利用历史信息、短期预测等多种途径进行计算，更新得到 WorldState。WorldState 存储了包括球、球员在内场上所有对象的基本状态信息 (BallState & PlayerState)。
- 从 WorldState 可以得到关于 Side 的信息，但得不到自己的 Unum。这样设计的原因是为了保持其“客观性”，一份 WorldState 无需任何改动就可以供所有的“队友”决策使用，镜像对称后得到的新 WorldState 可以供所有“队手”决策使用。
- BallState.h, PlayerState.{h, cpp}, WorldState.{h, cpp}

WorldState

- 将从 Observer 得到的相对信息换算成全局信息，并利用历史信息、短期预测等多种途径进行计算，更新得到 WorldState。WorldState 存储了包括球、球员在内场上所有对象的基本状态信息 (BallState & PlayerState)。
- 从 WorldState 可以得到关于 Side 的信息，但得不到自己的 Unum。这样设计的原因是为了保持其“客观性”，一份 WorldState 无需任何改动就可以供所有的“队友”决策使用，镜像对称后得到的新 WorldState 可以供所有“队手”决策使用。
- BallState.h, PlayerState.{h, cpp}, WorldState.{h, cpp}

WorldState

- 将从 Observer 得到的相对信息换算成全局信息，并利用历史信息、短期预测等多种途径进行计算，更新得到 WorldState。WorldState 存储了包括球、球员在内场上所有对象的基本状态信息 (BallState & PlayerState)。
- 从 WorldState 可以得到关于 Side 的信息，但得不到自己的 Unum。这样设计的原因是为了保持其“客观性”，一份 WorldState 无需任何改动就可以供所有的“队友”决策使用，镜像对称后得到的新 WorldState 可以供所有“队手”决策使用。
- BallState.h, PlayerState.{h, cpp}, WorldState.{h, cpp}

InfoState

- 对 WorldState 里的信息进一步计算和抽象得到 InfoState，包括：
  - 关于各种相对位置、角度信息的 PositionInfo
  - 关于截球信息的 InterceptInfo
- InfoState 也是客观信息，仅通过 WorldState 更新。
- InfoState.{h, cpp}, PositionInfo.{h, cpp}, InterceptInfo.{h, cpp}

InfoState

- 对 WorldState 里的信息进一步计算和抽象得到 InfoState，包括：
  - 关于各种相对位置、角度信息的 PositionInfo
  - 关于截球信息的 InterceptInfo
- InfoState 也是客观信息，仅通过 WorldState 更新。
- InfoState.{h, cpp}, PositionInfo.{h, cpp}, InterceptInfo.{h, cpp}

InfoState

- 对 WorldState 里的信息进一步计算和抽象得到 InfoState，包括：
  - 关于各种相对位置、角度信息的 PositionInfo
  - 关于截球信息的 InterceptInfo
- InfoState 也是客观信息，仅通过 WorldState 更新。
- InfoState.{h, cpp}, PositionInfo.{h, cpp}, InterceptInfo.{h, cpp}

InfoState

- 对 WorldState 里的信息进一步计算和抽象得到 InfoState，包括：
  - 关于各种相对位置、角度信息的 PositionInfo
  - 关于截球信息的 InterceptInfo
- InfoState 也是客观信息，仅通过 WorldState 更新。
- InfoState.{h, cpp}, PositionInfo.{h, cpp}, InterceptInfo.{h, cpp}

InfoState

- 对 WorldState 里的信息进一步计算和抽象得到 InfoState，包括：
  - 关于各种相对位置、角度信息的 PositionInfo
  - 关于截球信息的 InterceptInfo
- InfoState 也是客观信息，仅通过 WorldState 更新。
- InfoState.{h, cpp}, PositionInfo.{h, cpp}, InterceptInfo.{h, cpp}

DecisionData

- 特指比 InfoState 更抽象的上层决策数据，包括：
  - 进攻和防守使用的 Strategy
  - 仅供防守使用的 Analyser
- DecisionData 属于主观信息，通过 WorldState、InfoState 和自身历史进行更新。
- 决策要使用的一些历史相关信息存储在 DecisionData 里面，并以不同的形式进行更新。
- DecisionData.{h, cpp}, Strategy.{h, cpp}, Analyser.{h, cpp}

DecisionData

- 特指比 InfoState 更抽象的上层决策数据，包括：
  - 进攻和防守使用的 Strategy
  - 仅供防守使用的 Analyser
- DecisionData 属于主观信息，通过 WorldState、InfoState 和自身历史进行更新。
- 决策要使用的一些历史相关信息存储在 DecisionData 里面，并以不同的形式进行更新。
- DecisionData.{h, cpp}, Strategy.{h, cpp}, Analyser.{h, cpp}

DecisionData

- 特指比 InfoState 更抽象的上层决策数据，包括：
  - 进攻和防守使用的 Strategy
  - 仅供防守使用的 Analyser
- DecisionData 属于主观信息，通过 WorldState、InfoState 和自身历史进行更新。
- 决策要使用的一些历史相关信息存储在 DecisionData 里面，并以不同的形式进行更新。
- DecisionData.{h, cpp}, Strategy.{h, cpp}, Analyser.{h, cpp}

DecisionData

- 特指比 InfoState 更抽象的上层决策数据，包括：
  - 进攻和防守使用的 Strategy
  - 仅供防守使用的 Analyser
- DecisionData 属于主观信息，通过 WorldState、InfoState 和自身历史进行更新。
- 决策要使用的一些历史相关信息存储在 DecisionData 里面，并以不同的形式进行更新。
- DecisionData.{h, cpp}, Strategy.{h, cpp}, Analyser.{h, cpp}

DecisionData

- 特指比 InfoState 更抽象的上层决策数据，包括：
  - 进攻和防守使用的 Strategy
  - 仅供防守使用的 Analyser
- DecisionData 属于主观信息，通过 WorldState、InfoState 和自身历史进行更新。
- 决策要使用的一些历史相关信息存储在 DecisionData 里面，并以不同的形式进行更新。
- DecisionData.{h, cpp}, Strategy.{h, cpp}, Analyser.{h, cpp}

DecisionData

- 特指比 InfoState 更抽象的上层决策数据，包括：
  - 进攻和防守使用的 Strategy
  - 仅供防守使用的 Analyser
- DecisionData 属于主观信息，通过 WorldState、InfoState 和自身历史进行更新。
- 决策要使用的一些历史相关信息存储在 DecisionData 里面，并以不同的形式进行更新。
- DecisionData.{h, cpp}, Strategy.{h, cpp}, Analyser.{h, cpp}

主要内容

Agent — 决策的中心

- WrightEagleBASE 里所有行为的决策都是围绕 Agent 进行的，可认为是面向 Agent 的决策 (AOP)。
- Agent 完整封装了供决策使用的数据，并提供了动作接口：
  - 不同形式存在的 WorldState, InfoState, DecisionData
  - 阵型系统 Formation
  - 原子动作命令接口 ActionEffector

Agent — 决策的中心

- WrightEagleBASE 里所有行为的决策都是围绕 Agent 进行的，可认为是面向 Agent 的决策 (AOP)。
- Agent 完整封装了供决策使用的数据，并提供了动作接口：
  - 不同形式存在的 WorldState, InfoState, DecisionData
  - 阵型系统 Formation
  - 原子动作命令接口 ActionEffector

Agent — 决策的中心

- WrightEagleBASE 里所有行为的决策都是围绕 Agent 进行的，可认为是面向 Agent 的决策 (AOP)。
- Agent 完整封装了供决策使用的数据，并提供了动作接口：
  - 不同形式存在的 WorldState, InfoState, DecisionData
  - 阵型系统 Formation
  - 原子动作命令接口 ActionEffector

Agent — 决策的中心

- WrightEagleBASE 里所有行为的决策都是围绕 Agent 进行的，可认为是面向 Agent 的决策 (AOP)。
- Agent 完整封装了供决策使用的数据，并提供了动作接口：
  - 不同形式存在的 WorldState, InfoState, DecisionData
  - 阵型系统 Formation
  - 原子动作命令接口 ActionEffector

Agent — 决策的中心

- WrightEagleBASE 里所有行为的决策都是围绕 Agent 进行的，可认为是面向 Agent 的决策 (AOP)。
- Agent 完整封装了供决策使用的数据，并提供了动作接口：
  - 不同形式存在的 WorldState, InfoState, DecisionData
  - 阵型系统 Formation
  - 原子动作命令接口 ActionEffector

Agent — 决策的中心 (cont.)

- Agent 可以“克隆”出新的 Agent 实例 (仅 Unum 可能不同)，用来计算场上其他队友、对手的决策。
- 全局存在唯一一个 Agent 代表真实的 Client 自己，使用这个 Agent 进行决策期间，可以在任何地方、任何时候“克隆”出新的 Agent，“克隆”出的 Agent 同样具有“克隆”能力。
- Agent.{h, cpp}, Formation.{h, cpp}, ActionEffector.{h, cpp}, ...

Agent — 决策的中心 (cont.)

- Agent 可以“克隆”出新的 Agent 实例 (仅 Unum 可能不同)，用来计算场上其他队友、对手的决策。
- 全局存在唯一一个 Agent 代表真实的 Client 自己，使用这个 Agent 进行决策期间，可以在任何地方、任何时候“克隆”出新的 Agent，“克隆”出的 Agent 同样具有“克隆”能力。
- Agent.{h, cpp}, Formation.{h, cpp}, ActionEffector.{h, cpp}, ...

Agent — 决策的中心 (cont.)

- Agent 可以“克隆”出新的 Agent 实例 (仅 Unum 可能不同)，用来计算场上其他队友、对手的决策。
- 全局存在唯一一个 Agent 代表真实的 Client 自己，使用这个 Agent 进行决策期间，可以在任何地方、任何时候“克隆”出新的 Agent，“克隆”出的 Agent 同样具有“克隆”能力。
- Agent.{h, cpp}, Formation.{h, cpp}, ActionEffector.{h, cpp}, ...

ActiveBehavior

- WrightEagleBASE 里面规划跟执行是分开进行的：规划由各 BehaviorPlanner 进行，执行由对应的 BehaviorExecuter 进行，这样可以每周期同时进行多个行为的规划但只选择具有最好收益的行为去执行。
- ActiveBehavior 是联系 BehaviorPlanner 和 BehaviorExecuter 的桥梁：
  - BehaviorPlanner 规划的结果表现为一个 ActiveBehavior。
  - 一个合法的 ActiveBehavior 是可“执行”的，执行过程调用相应的 BehaviorExecuter 进行。
- ActiveBehavior 里面存储了行为执行时可能用到的所有数据，这些数据在行为规划时填充。
- BehaviorBase.{h, cpp}

ActiveBehavior

- WrightEagleBASE 里面规划跟执行是分开进行的：规划由各 BehaviorPlanner 进行，执行由对应的 BehaviorExecuter 进行，这样可以每周期同时进行多个行为的规划但只选择具有最好收益的行为去执行。
- ActiveBehavior 是联系 BehaviorPlanner 和 BehaviorExecuter 的桥梁：
  - BehaviorPlanner 规划的结果表现为一个 ActiveBehavior。
  - 一个合法的 ActiveBehavior 是可“执行”的，执行过程调用相应的 BehaviorExecuter 进行。
- ActiveBehavior 里面存储了行为执行时可能用到的所有数据，这些数据在行为规划时填充。
- BehaviorBase.{h, cpp}

ActiveBehavior

- WrightEagleBASE 里面规划跟执行是分开进行的：规划由各 BehaviorPlanner 进行，执行由对应的 BehaviorExecuter 进行，这样可以每周期同时进行多个行为的规划但只选择具有最好收益的行为去执行。
- ActiveBehavior 是联系 BehaviorPlanner 和 BehaviorExecuter 的桥梁：
  - BehaviorPlanner 规划的结果表现为一个 ActiveBehavior。
  - 一个合法的 ActiveBehavior 是可“执行”的，执行过程调用相应的 BehaviorExecuter 进行。
- ActiveBehavior 里面存储了行为执行时可能用到的所有数据，这些数据在行为规划时填充。
- BehaviorBase.{h, cpp}

ActiveBehavior

- WrightEagleBASE 里面规划跟执行是分开进行的：规划由各 BehaviorPlanner 进行，执行由对应的 BehaviorExecuter 进行，这样可以每周期同时进行多个行为的规划但只选择具有最好收益的行为去执行。
- ActiveBehavior 是联系 BehaviorPlanner 和 BehaviorExecuter 的桥梁：
  - BehaviorPlanner 规划的结果表现为一个 ActiveBehavior。
  - 一个合法的 ActiveBehavior 是可“执行”的，执行过程调用相应的 BehaviorExecuter 进行。
- ActiveBehavior 里面存储了行为执行时可能用到的所有数据，这些数据在行为规划时填充。
- BehaviorBase.{h, cpp}

ActiveBehavior

- WrightEagleBASE 里面规划跟执行是分开进行的：规划由各 BehaviorPlanner 进行，执行由对应的 BehaviorExecuter 进行，这样可以每周期同时进行多个行为的规划但只选择具有最好收益的行为去执行。
- ActiveBehavior 是联系 BehaviorPlanner 和 BehaviorExecuter 的桥梁：
  - BehaviorPlanner 规划的结果表现为一个 ActiveBehavior。
  - 一个合法的 ActiveBehavior 是可“执行”的，执行过程调用相应的 BehaviorExecuter 进行。
- ActiveBehavior 里面存储了行为执行时可能用到的所有数据，这些数据在行为规划时填充。
- BehaviorBase.{h, cpp}

ActiveBehavior

- WrightEagleBASE 里面规划跟执行是分开进行的：规划由各 BehaviorPlanner 进行，执行由对应的 BehaviorExecuter 进行，这样可以每周期同时进行多个行为的规划但只选择具有最好收益的行为去执行。
- ActiveBehavior 是联系 BehaviorPlanner 和 BehaviorExecuter 的桥梁：
  - BehaviorPlanner 规划的结果表现为一个 ActiveBehavior。
  - 一个合法的 ActiveBehavior 是可“执行”的，执行过程调用相应的 BehaviorExecuter 进行。
- ActiveBehavior 里面存储了行为执行时可能用到的所有数据，这些数据在行为规划时填充。
- BehaviorBase.{h, cpp}

BehaviorPlanner

- 所有行为的规划过程在对应的 BehaviorPlanner 里面进行，规划过程不改变 Agent 里面的客观数据。
- 规划结果一定是一个填充好执行时所需数据的 ActiveBehavior，经上层选择后，交给对应的 BehaviorExecuter 执行。
- 并不是所有的 BehaviorPlanner 都有相应的 BehaviorExecuter，有些比较抽象的 BehaviorPlanner 是不需要执行的。
- 不同的 BehaviorPlanner 可以有适合自己“风格”的规划过程，互相不受限制且没有“影响”。
- Behavior∗.{h, cpp}

BehaviorPlanner

- 所有行为的规划过程在对应的 BehaviorPlanner 里面进行，规划过程不改变 Agent 里面的客观数据。
- 规划结果一定是一个填充好执行时所需数据的 ActiveBehavior，经上层选择后，交给对应的 BehaviorExecuter 执行。
- 并不是所有的 BehaviorPlanner 都有相应的 BehaviorExecuter，有些比较抽象的 BehaviorPlanner 是不需要执行的。
- 不同的 BehaviorPlanner 可以有适合自己“风格”的规划过程，互相不受限制且没有“影响”。
- Behavior∗.{h, cpp}

BehaviorPlanner

- 所有行为的规划过程在对应的 BehaviorPlanner 里面进行，规划过程不改变 Agent 里面的客观数据。
- 规划结果一定是一个填充好执行时所需数据的 ActiveBehavior，经上层选择后，交给对应的 BehaviorExecuter 执行。
- 并不是所有的 BehaviorPlanner 都有相应的 BehaviorExecuter，有些比较抽象的 BehaviorPlanner 是不需要执行的。
- 不同的 BehaviorPlanner 可以有适合自己“风格”的规划过程，互相不受限制且没有“影响”。
- Behavior∗.{h, cpp}

BehaviorPlanner

- 所有行为的规划过程在对应的 BehaviorPlanner 里面进行，规划过程不改变 Agent 里面的客观数据。
- 规划结果一定是一个填充好执行时所需数据的 ActiveBehavior，经上层选择后，交给对应的 BehaviorExecuter 执行。
- 并不是所有的 BehaviorPlanner 都有相应的 BehaviorExecuter，有些比较抽象的 BehaviorPlanner 是不需要执行的。
- 不同的 BehaviorPlanner 可以有适合自己“风格”的规划过程，互相不受限制且没有“影响”。
- Behavior∗.{h, cpp}

BehaviorPlanner

- 所有行为的规划过程在对应的 BehaviorPlanner 里面进行，规划过程不改变 Agent 里面的客观数据。
- 规划结果一定是一个填充好执行时所需数据的 ActiveBehavior，经上层选择后，交给对应的 BehaviorExecuter 执行。
- 并不是所有的 BehaviorPlanner 都有相应的 BehaviorExecuter，有些比较抽象的 BehaviorPlanner 是不需要执行的。
- 不同的 BehaviorPlanner 可以有适合自己“风格”的规划过程，互相不受限制且没有“影响”。
- Behavior$*$.{h, cpp}

BehaviorExectuer

- 每个需要执行的 BehaviorPlanner 都有一个对应的 BehaviorExecuter，负责实际执行这个行为。
- 执行过程最终都是调用 ActionEffector 提供的接口进行的。
- 不同的 BehaviorExecuter 对执行时数据的需求是不同的，只能用来执行对应的 ActionEffector。
- Behavior∗.{h, cpp}

BehaviorExectuer

- 每个需要执行的 BehaviorPlanner 都有一个对应的 BehaviorExecuter，负责实际执行这个行为。
- 执行过程最终都是调用 ActionEffector 提供的接口进行的。
- 不同的 BehaviorExecuter 对执行时数据的需求是不同的，只能用来执行对应的 ActionEffector。
- Behavior*.{h, cpp}

BehaviorExectuer

- 每个需要执行的 BehaviorPlanner 都有一个对应的 BehaviorExecuter，负责实际执行这个行为。
- 执行过程最终都是调用 ActionEffector 提供的接口进行的。
- 不同的 BehaviorExecuter 对执行时数据的需求是不同的，只能用来执行对应的 ActionEffector。
- Behavior∗.{h, cpp}

BehaviorExectuer

- 每个需要执行的 BehaviorPlanner 都有一个对应的 BehaviorExecuter，负责实际执行这个行为。
- 执行过程最终都是调用 ActionEffector 提供的接口进行的。
- 不同的 BehaviorExecuter 对执行时数据的需求是不同的，只能用来执行对应的 ActionEffector。
- Behavior$*$.{h, cpp}

Agent 的“克隆”计算

- 有时候需要预测队友的反应，因为队友跟自己使用的是相同的代码，最好的办法就是准备一份可供“队友”决策时使用的完备的数据，并调用相应决策代码，观察决策结果 — 这通过“克隆”出队友 Agent 实现。
- 相应的，有时需要预测对手的反应，缺失精确对手模型的情况下，根据“英雄所见略同”原理，可以认为“对手”可能做出自己在那种情况下类似的决策 — 这通过“克隆”出对手 Agent 实现。

Agent 的“克隆”计算

- 有时候需要预测队友的反应，因为队友跟自己使用的是相同的代码，最好的办法就是准备一份可供“队友”决策时使用的完备的数据，并调用相应决策代码，观察决策结果 — 这通过“克隆”出队友 Agent 实现。
- 相应的，有时需要预测对手的反应，缺失精确对手模型的情况下，根据“英雄所见略同”原理，可以认为“对手”可能做出自己在那种情况下类似的决策 — 这通过“克隆”出对手 Agent 实现。

Agent 的“克隆”计算 (cont.)

- 为了可以进行“克隆”计算，Agent 提供的支持有：
  - 显式区分客观和主观信息，客观信息在“克隆”计算时重用。
  - 维护供对手 Agent 决策使用的的 WorldState，通过镜像得到。
- “克隆”计算前往往需要对 WorldState 作少量修改，并且在计算后恢复 WorldState，这是通过 WorldStateSetter 机制实现的 (用法比较 ugly，这是为性能考虑做出的牺牲)。
- Agent.{h, cpp}, WorldState.h:360

Agent 的“克隆”计算 (cont.)

- 为了可以进行“克隆”计算，Agent 提供的支持有：
  - 显式区分客观和主观信息，客观信息在“克隆”计算时重用。
  - 维护供对手 Agent 决策使用的的 WorldState，通过镜像得到。
- “克隆”计算前往往需要对 WorldState 作少量修改，并且在计算后恢复 WorldState，这是通过 WorldStateSetter 机制实现的 (用法比较 ugly，这是为性能考虑做出的牺牲)。
- Agent.{h, cpp}, WorldState.h:360

Agent 的“克隆”计算 (cont.)

- 为了可以进行“克隆”计算，Agent 提供的支持有：
  - 显式区分客观和主观信息，客观信息在“克隆”计算时重用。
  - 维护供对手 Agent 决策使用的的 WorldState，通过镜像得到。
- “克隆”计算前往往需要对 WorldState 作少量修改，并且在计算后恢复 WorldState，这是通过 WorldStateSetter 机制实现的 (用法比较 ugly，这是为性能考虑做出的牺牲)。
- Agent.{h, cpp}, WorldState.h:360

Agent 的“克隆”计算 (cont.)

- 为了可以进行“克隆”计算，Agent 提供的支持有：
  - 显式区分客观和主观信息，客观信息在“克隆”计算时重用。
  - 维护供对手 Agent 决策使用的的 WorldState，通过镜像得到。
- “克隆”计算前往往需要对 WorldState 作少量修改，并且在计算后恢复 WorldState，这是通过 WorldStateSetter 机制实现的 (用法比较 ugly，这是为性能考虑做出的牺牲)。
- Agent.{h, cpp}, WorldState.h:360

Agent 的“克隆”计算 (cont.)

- 为了可以进行“克隆”计算，Agent 提供的支持有：
  - 显式区分客观和主观信息，客观信息在“克隆”计算时重用。
  - 维护供对手 Agent 决策使用的的 WorldState，通过镜像得到。
- “克隆”计算前往往需要对 WorldState 作少量修改，并且在计算后恢复 WorldState，这是通过 WorldStateSetter 机制实现的 (用法比较 ugly，这是为性能考虑做出的牺牲)。
- Agent.{h, cpp}, WorldState.h:360

DecisionTree

- 行为决策的最上层，调用 BehaviorPlanner、筛选 ActiveBehavior，并完成执行。
- 应该具有某种树状结构，可以计算多“步”，不过出于性能的考虑，目前只计算了一“步”。
- 同时设置了历史 ActiveBehavior 信息，供行为保持用。
- DecisionTree.{h, cpp}

DecisionTree

- 行为决策的最上层，调用 BehaviorPlanner、筛选 ActiveBehavior，并完成执行。
- 应该具有某种树状结构，可以计算多“步”，不过出于性能的考虑，目前只计算了一“步”。
- 同时设置了历史 ActiveBehavior 信息，供行为保持用。
- DecisionTree.{h, cpp}

DecisionTree

- 行为决策的最上层，调用 BehaviorPlanner、筛选 ActiveBehavior，并完成执行。
- 应该具有某种树状结构，可以计算多“步”，不过出于性能的考虑，目前只计算了一“步”。
- 同时设置了历史 ActiveBehavior 信息，供行为保持用。
- DecisionTree.{h, cpp}

DecisionTree

- 行为决策的最上层，调用 BehaviorPlanner、筛选 ActiveBehavior，并完成执行。
- 应该具有某种树状结构，可以计算多“步”，不过出于性能的考虑，目前只计算了一“步”。
- 同时设置了历史 ActiveBehavior 信息，供行为保持用。
- DecisionTree.{h, cpp}

主要内容

目录

- conf/ — player.conf、server.conf 等配置文件
- data/ — 离线计算的数据文件
- formations/ — 阵型文件
- src/ — C++ 源码
- Logfiles/ — 所有 Log 存在此处
- Debug/ — Debug 版本 Makefile
- Release/ — Release 版本 Makefile

目录

- conf/ — player.conf、server.conf 等配置文件
- data/ — 离线计算的数据文件
- formations/ — 阵型文件
- src/ — C++ 源码
- Logfiles/ — 所有 Log 存在此处
- Debug/ — Debug 版本 Makefile
- Release/ — Release 版本 Makefile

目录

- conf/ — player.conf、server.conf 等配置文件
- data/ — 离线计算的数据文件
- formations/ — 阵型文件
- src/ — C++ 源码
- Logfiles/ — 所有 Log 存在此处
- Debug/ — Debug 版本 Makefile
- Release/ — Release 版本 Makefile

目录

- conf/ — player.conf、server.conf 等配置文件
- data/ — 离线计算的数据文件
- formations/ — 阵型文件
- src/ — C++ 源码
- Logfiles/ — 所有 Log 存在此处
- Debug/ — Debug 版本 Makefile
- Release/ — Release 版本 Makefile

目录

- conf/ — player.conf、server.conf 等配置文件
- data/ — 离线计算的数据文件
- formations/ — 阵型文件
- src/ — C++ 源码
- Logfiles/ — 所有 Log 存在此处
- Debug/ — Debug 版本 Makefile
- Release/ — Release 版本 Makefile

目录

- conf/ — player.conf、server.conf 等配置文件
- data/ — 离线计算的数据文件
- formations/ — 阵型文件
- src/ — C++ 源码
- Logfiles/ — 所有 Log 存在此处
- Debug/ — Debug 版本 Makefile
- Release/ — Release 版本 Makefile

目录

- conf/ — player.conf、server.conf 等配置文件
- data/ — 离线计算的数据文件
- formations/ — 阵型文件
- src/ — C++ 源码
- Logfiles/ — 所有 Log 存在此处
- Debug/ — Debug 版本 Makefile
- Release/ — Release 版本 Makefile

文件

- dbg，dd — 动态调试工具
- genlog，showlog — SightLog 相关工具
- memcheck — 检查内存错误工具
- initrc — 以上工具共用脚本
- dynamicdebug.txt — 动态调试用
- start.sh — 球队上场脚本
- Makefile — Makefile

文件

- dbg，dd — 动态调试工具
- genlog，showlog — SightLog 相关工具
- memcheck — 检查内存错误工具
- initrc — 以上工具共用脚本
- dynamicdebug.txt — 动态调试用
- start.sh — 球队上场脚本
- Makefile — Makefile

文件

- dbg，dd — 动态调试工具
- genlog，showlog — SightLog 相关工具
- memcheck — 检查内存错误工具
- initrc — 以上工具共用脚本
- dynamicdebug.txt — 动态调试用
- start.sh — 球队上场脚本
- Makefile — Makefile

文件

- dbg，dd — 动态调试工具
- genlog，showlog — SightLog 相关工具
- memcheck — 检查内存错误工具
- initrc — 以上工具共用脚本
- dynamicdebug.txt — 动态调试用
- start.sh — 球队上场脚本
- Makefile — Makefile

文件

- dbg，dd — 动态调试工具
- genlog，showlog — SightLog 相关工具
- memcheck — 检查内存错误工具
- initrc — 以上工具共用脚本
- dynamicdebug.txt — 动态调试用
- start.sh — 球队上场脚本
- Makefile — Makefile

文件

- dbg，dd — 动态调试工具
- genlog，showlog — SightLog 相关工具
- memcheck — 检查内存错误工具
- initrc — 以上工具共用脚本
- dynamicdebug.txt — 动态调试用
- start.sh — 球队上场脚本
- Makefile — Makefile

文件

- `dbg, dd` — 动态调试工具
- `genlog, showlog` — SightLog 相关工具
- `memcheck` — 检查内存错误工具
- `initrc` — 以上工具共用脚本
- `dynamicdebug.txt` — 动态调试用
- `start.sh` — 球队上场脚本
- `Makefile` — Makefile

一些源文件

- Types.{h, cpp} — 基本数据类型、宏定义
- Geometry.{h, cpp} — 几何相关计算
- Utilities.{h, cpp} — 功能函数、数据结构
- Dasher.{h, cpp} — 行动相关
- Kicker.{h, cpp} — 踢球相关
- Tackler.{h, cpp} — 铲球相关
- Behavior*.{h, cpp} — 各种行为规划、执行
- CommunicationSystem.{h, cpp} — 通信决策子系统
- VisualSystem.{h, cpp} — 视觉决策子系统
- ... — ...

一些源文件

- Types.{h, cpp} — 基本数据类型、宏定义
- Geometry.{h, cpp} — 几何相关计算
- Utilities.{h, cpp} — 功能函数、数据结构
- Dasher.{h, cpp} — 行动相关
- Kicker.{h, cpp} — 踢球相关
- Tackler.{h, cpp} — 铲球相关
- Behavior*.{h, cpp} — 各种行为规划、执行
- CommunicationSystem.{h, cpp} — 通信决策子系统
- VisualSystem.{h, cpp} — 视觉决策子系统
- ... — ...

一些源文件

- Types.{h, cpp} — 基本数据类型、宏定义
- Geometry.{h, cpp} — 几何相关计算
- Utilities.{h, cpp} — 功能函数、数据结构
- Dasher.{h, cpp} — 行动相关
- Kicker.{h, cpp} — 踢球相关
- Tackler.{h, cpp} — 铲球相关
- Behavior*.{h, cpp} — 各种行为规划、执行
- CommunicationSystem.{h, cpp} — 通信决策子系统
- VisualSystem.{h, cpp} — 视觉决策子系统
- ... — ...

一些源文件

- Types.{h, cpp} — 基本数据类型、宏定义
- Geometry.{h, cpp} — 几何相关计算
- Utilities.{h, cpp} — 功能函数、数据结构
- Dasher.{h, cpp} — 行动相关
- Kicker.{h, cpp} — 踢球相关
- Tackler.{h, cpp} — 铲球相关
- Behavior*.{h, cpp} — 各种行为规划、执行
- CommunicationSystem.{h, cpp} — 通信决策子系统
- VisualSystem.{h, cpp} — 视觉决策子系统
- ... — ...

一些源文件

- Types.{h, cpp} — 基本数据类型、宏定义
- Geometry.{h, cpp} — 几何相关计算
- Utilities.{h, cpp} — 功能函数、数据结构
- Dasher.{h, cpp} — 行动相关
- Kicker.{h, cpp} — 踢球相关
- Tackler.{h, cpp} — 铲球相关
- Behavior*.{h, cpp} — 各种行为规划、执行
- CommunicationSystem.{h, cpp} — 通信决策子系统
- VisualSystem.{h, cpp} — 视觉决策子系统
- ... — ...

一些源文件

- Types.{h, cpp} — 基本数据类型、宏定义
- Geometry.{h, cpp} — 几何相关计算
- Utilities.{h, cpp} — 功能函数、数据结构
- Dasher.{h, cpp} — 行动相关
- Kicker.{h, cpp} — 踢球相关
- Tackler.{h, cpp} — 铲球相关
- Behavior*.{h, cpp} — 各种行为规划、执行
- CommunicationSystem.{h, cpp} — 通信决策子系统
- VisualSystem.{h, cpp} — 视觉决策子系统
- ... — ...

一些源文件

- Types.{h, cpp} — 基本数据类型、宏定义
- Geometry.{h, cpp} — 几何相关计算
- Utilities.{h, cpp} — 功能函数、数据结构
- Dasher.{h, cpp} — 行动相关
- Kicker.{h, cpp} — 踢球相关
- Tackler.{h, cpp} — 铲球相关
- Behavior*.{h, cpp} — 各种行为规划、执行
- CommunicationSystem.{h, cpp} — 通信决策子系统
- VisualSystem.{h, cpp} — 视觉决策子系统
- ... — ...

一些源文件

- Types.{h, cpp} — 基本数据类型、宏定义
- Geometry.{h, cpp} — 几何相关计算
- Utilities.{h, cpp} — 功能函数、数据结构
- Dasher.{h, cpp} — 行动相关
- Kicker.{h, cpp} — 踢球相关
- Tackler.{h, cpp} — 铲球相关
- Behavior*.{h, cpp} — 各种行为规划、执行
- CommunicationSystem.{h, cpp} — 通信决策子系统
- VisualSystem.{h, cpp} — 视觉决策子系统
- ... — ...

一些源文件

- Types.{h, cpp} — 基本数据类型、宏定义
- Geometry.{h, cpp} — 几何相关计算
- Utilities.{h, cpp} — 功能函数、数据结构
- Dasher.{h, cpp} — 行动相关
- Kicker.{h, cpp} — 踢球相关
- Tackler.{h, cpp} — 铲球相关
- Behavior*.{h, cpp} — 各种行为规划、执行
- CommunicationSystem.{h, cpp} — 通信决策子系统
- VisualSystem.{h, cpp} — 视觉决策子系统
- ... — ...

一些源文件

- Types.{h, cpp} — 基本数据类型、宏定义
- Geometry.{h, cpp} — 几何相关计算
- Utilities.{h, cpp} — 功能函数、数据结构
- Dasher.{h, cpp} — 行动相关
- Kicker.{h, cpp} — 踢球相关
- Tackler.{h, cpp} — 铲球相关
- Behavior*.{h, cpp} — 各种行为规划、执行
- CommunicationSystem.{h, cpp} — 通信决策子系统
- VisualSystem.{h, cpp} — 视觉决策子系统
- … — …

一些继承关系

- BaseState
  - StaticState
  - MobileState
    - BallState
    - PlayerState
- BehaviorAttackData
  - BehaviorDefenseData
- Client
  - Player
  - Coach

一些继承关系

- BaseState
  - StaticState
  - MobileState
    - BallState
    - PlayerState
- BehaviorAttackData
  - BehaviorDefenseData
- Client
  - Player
  - Coach

一些继承关系

- BaseState
  - StaticState
  - MobileState
    - BallState
    - PlayerState
- BehaviorAttackData
  - BehaviorDefenseData
- Client
  - Player
  - Coach

一些继承关系 (cont.)

- Updatable
  - InfoStateBase
    - InterceptInfo
    - PositionInfo
  - DecisionData
    - Strategy
    - Analyser
- FormationTacticBase
  - FormationTacticKickOffPosition
  - ...

一些继承关系 (cont.)

- Updatable
  - InfoStateBase
    - InterceptInfo
    - PositionInfo
  - DecisionData
    - Strategy
    - Analyser
- FormationTacticBase
  - FormationTacticKickOffPosition
  - ...

一些继承关系 (cont.)

- BehaviorAttackData
  - BehaviorPlannerBase<BehaviorAttackData>
    - BehaviorAttackPlanner
    - BehaviorShootPlanner
    - BehaviorPassPlanner
    - BehaviorDribblePlanner
    - ...
- BehaviorAttackData
- BehaviorExecutable
  - BehaviorExecuterBase<BehaviorAttackData>
    - BehaviorShootExecuter
    - BehaviorPassExecuter
    - BehaviorDribbleExecuter
    - ...

一些继承关系 (cont.)

- BehaviorAttackData
  - BehaviorPlannerBase<BehaviorAttackData>
    - BehaviorAttackPlanner
    - BehaviorShootPlanner
    - BehaviorPassPlanner
    - BehaviorDribblePlanner
    - ...
- BehaviorAttackData
- BehaviorExecutable
  - BehaviorExecuterBase<BehaviorAttackData>
    - BehaviorShootExecuter
    - BehaviorPassExecuter
    - BehaviorDribbleExecuter
    - ...

一些继承关系 (cont.)

- BehaviorDefenseData
  - BehaviorPlannerBase<BehaviorDefenseData>
    - BehaviorDefensePlanner
    - BehaviorBlockPlanner
    - BehaviorMarkPlanner
    - BehaviorFormationPlanner
    - ...
- BehaviorDefenseData
- BehaviorExecutable
  - BehaviorExecuterBase<BehaviorDefenseData>
    - BehaviorBlockExecuter
    - BehaviorMarkExecuter
    - BehaviorFormationExecuter
    - ...

一些继承关系 (cont.)

- BehaviorDefenseData
  - BehaviorPlannerBase<BehaviorDefenseData>
    - BehaviorDefensePlanner
    - BehaviorBlockPlanner
    - BehaviorMarkPlanner
    - BehaviorFormationPlanner
    - ...
- BehaviorDefenseData
- BehaviorExecutable
  - BehaviorExecuterBase<BehaviorDefenseData>
    - BehaviorBlockExecuter
    - BehaviorMarkExecuter
    - BehaviorFormationExecuter
    - ...

Player 运行流程

- Client::RunNormal(···)
  - Client::SendOptionToServer(···)
  - Client::MainLoop(···)
    - Observer::WaitForNewInfo(···)
    - Player::Run(···)
    - Observer::SetCommandSend(···)
- Player::Run(···)
  - WorldModel::Update(···)
  - DecisionTree::Decision(···)
  - VisualSystem::Decision(···)
  - CommunicateSystem::Decision(···)

Player 运行流程

- Client::RunNormal(···)
  - Client::SendOptionToServer(···)
  - Client::MainLoop(···)
    - Observer::WaitForNewInfo(···)
    - Player::Run(···)
    - Observer::SetCommandSend(···)
- Player::Run(···)
  - WorldModel::Update(···)
  - DecisionTree::Decision(···)
  - VisualSystem::Decision(···)
  - CommunicateSystem::Decision(···)

Player 决策流程

- DecisionTree::Decision(···)
  - DecisionTree::Search(···)
    - Bheavior∗Planner::Plan(···)
    - Bheavior∗Planner::Plan(···)
    - ···
  - ActiveBehavior::Execute(···)
    - Behavior∗Executer::Execute(···)
- Bheavior∗Planner::Plan(···)
  - Bheavior∗Planner::Plan(···)
  - Bheavior∗Planner::Plan(···)
  - ···

Player 决策流程

- DecisionTree::Decision(···)
  - DecisionTree::Search(···)
    - Bheavior∗Planner::Plan(···)
    - Bheavior∗Planner::Plan(···)
    - ···
  - ActiveBehavior::Execute(···)
    - Behavior∗Executer::Execute(···)
- Bheavior∗Planner::Plan(···)
  - Bheavior∗Planner::Plan(···)
  - Bheavior∗Planner::Plan(···)
  - ···

主要内容

PlayerParam & ServerParam

- PlayerParam & ServerParam 都继承于 ParamEngine，存储所有“参数”信息。
- 可以通过 Server 发来的信息、配置文件和命令行参数来更新。
- ParamEngine.{h, cpp}, PlayerParam.{h, cpp}, ServerParam.{h, cpp}

PlayerParam & ServerParam

- PlayerParam & ServerParam 都继承于 ParamEngine，存储所有“参数”信息。
- 可以通过 Server 发来的信息、配置文件和命令行参数来更新。
- ParamEngine.{h, cpp}, PlayerParam.{h, cpp}, ServerParam.{h, cpp}

PlayerParam & ServerParam

- PlayerParam & ServerParam 都继承于 ParamEngine，存储所有“参数”信息。
- 可以通过 Server 发来的信息、配置文件和命令行参数来更新。
- ParamEngine.{h, cpp}, PlayerParam.{h, cpp}, ServerParam.{h, cpp}

Dasher

- GetBall(···) — 在指定周期或以“最快”方式截球
- GoToPoint(···) — 按指定方式“最快”跑动到某一目标点
- CycleNeedToPoint(···) — 计算跑动某一点所需的整数周期
- RealCycleNeedToPoint(···) — 计算跑动某一点所需的实数周期
- Dasher.{h, cpp}

Dasher

- GetBall(···) — 在指定周期或以“最快”方式截球
- GoToPoint(···) — 按指定方式“最快”跑动到某一目标点
- CycleNeedToPoint(···) — 计算跑动某一点所需的整数周期
- RealCycleNeedToPoint(···) — 计算跑动某一点所需的实数周期
- Dasher.{h, cpp}

Dasher

- GetBall(···) — 在指定周期或以“最快”方式截球
- GoToPoint(···) — 按指定方式“最快”跑动到某一目标点
- CycleNeedToPoint(···) — 计算跑动某一点所需的整数周期
- RealCycleNeedToPoint(···) — 计算跑动某一点所需的实数周期
- Dasher.{h, cpp}

Dasher

- GetBall(···) — 在指定周期或以“最快”方式截球
- GoToPoint(···) — 按指定方式“最快”跑动到某一目标点
- CycleNeedToPoint(···) — 计算跑动某一点所需的整数周期
- RealCycleNeedToPoint(···) — 计算跑动某一点所需的实数周期
- Dasher.{h, cpp}

Dasher

- GetBall(···) — 在指定周期或以“最快”方式截球
- GoToPoint(···) — 按指定方式“最快”跑动到某一目标点
- CycleNeedToPoint(···) — 计算跑动某一点所需的整数周期
- RealCycleNeedToPoint(···) — 计算跑动某一点所需的实数周期
- Dasher.{h, cpp}

Kicker

- KickBall(···) — 按指定方式踢球
- GetMaxSpeed(···) — 得到指定方向出球能球得到的最大速率
- GetStopBallAction(···) — 得到停球动作
- GetAccelerateBallAction(···) — 将球加速到指定的目标速度
- GetKickBallToAngleAction(···) — 将球踢向指定方向
- Kicker.{h, cpp}

Kicker

- KickBall(···) — 按指定方式踢球
- GetMaxSpeed(···) — 得到指定方向出球能球得到的最大速率
- GetStopBallAction(···) — 得到停球动作
- GetAccelerateBallAction(···) — 将球加速到指定的目标速度
- GetKickBallToAngleAction(···) — 将球踢向指定方向
- Kicker.{h, cpp}

Kicker

- KickBall(···) — 按指定方式踢球
- GetMaxSpeed(···) — 得到指定方向出球能球得到的最大速率
- GetStopBallAction(···) — 得到停球动作
- GetAccelerateBallAction(···) — 将球加速到指定的目标速度
- GetKickBallToAngleAction(···) — 将球踢向指定方向
- Kicker.{h, cpp}

Kicker

- KickBall(···) — 按指定方式踢球
- GetMaxSpeed(···) — 得到指定方向出球能球得到的最大速率
- GetStopBallAction(···) — 得到停球动作
- GetAccelerateBallAction(···) — 将球加速到指定的目标速度
- GetKickBallToAngleAction(···) — 将球踢向指定方向
- Kicker.{h, cpp}

Kicker

- KickBall(···) — 按指定方式踢球
- GetMaxSpeed(···) — 得到指定方向出球能球得到的最大速率
- GetStopBallAction(···) — 得到停球动作
- GetAccelerateBallAction(···) — 将球加速到指定的目标速度
- GetKickBallToAngleAction(···) — 将球踢向指定方向
- Kicker.{h, cpp}

Kicker

- KickBall(···) — 按指定方式踢球
- GetMaxSpeed(···) — 得到指定方向出球能球得到的最大速率
- GetStopBallAction(···) — 得到停球动作
- GetAccelerateBallAction(···) — 将球加速到指定的目标速度
- GetKickBallToAngleAction(···) — 将球踢向指定方向
- Kicker.{h, cpp}

Tackler

- TackleStopBall(···) — 铲停球
- CanTackleStopBall(···) — 是否铲停球
- TackleToDir(···) — 将球铲到指定方向
- CanTackleToDir(···) — 是否可以将球铲到指定方向
- GetBallVelAfterTackleToDir(···) — 得到将球铲到指定方向后的出球速度
- Tackler.{h, cpp}

Tackler

- TackleStopBall(···) — 铲停球
- CanTackleStopBall(···) — 是否铲停球
- TackleToDir(···) — 将球铲到指定方向
- CanTackleToDir(···) — 是否可以将球铲到指定方向
- GetBallVelAfterTackleToDir(···) — 得到将球铲到指定方向后的出球速度
- Tackler.{h, cpp}

Tackler

- TackleStopBall(···) — 铲停球
- CanTackleStopBall(···) — 是否铲停球
- TackleToDir(···) — 将球铲到指定方向
- CanTackleToDir(···) — 是否可以将球铲到指定方向
- GetBallVelAfterTackleToDir(···) — 得到将球铲到指定方向后的出球速度
- Tackler.{h, cpp}

Tackler

- TackleStopBall(···) — 铲停球
- CanTackleStopBall(···) — 是否铲停球
- TackleToDir(···) — 将球铲到指定方向
- CanTackleToDir(···) — 是否可以将球铲到指定方向
- GetBallVelAfterTackleToDir(···) — 得到将球铲到指定方向后的出球速度
- Tackler.{h, cpp}

Tackler

- TackleStopBall(···) — 铲停球
- CanTackleStopBall(···) — 是否铲停球
- TackleToDir(···) — 将球铲到指定方向
- CanTackleToDir(···) — 是否可以将球铲到指定方向
- GetBallVelAfterTackleToDir(···) — 得到将球铲到指定方向后的出球速度
- Tackler.{h, cpp}

Tackler

- TackleStopBall(···) — 铲停球
- CanTackleStopBall(···) — 是否铲停球
- TackleToDir(···) — 将球铲到指定方向
- CanTackleToDir(···) — 是否可以将球铲到指定方向
- GetBallVelAfterTackleToDir(···) — 得到将球铲到指定方向后的出球速度
- Tackler.{h, cpp}

VisualSystem

- RaiseBall(···) — 按指定权值注意球
- RaisePlayer(···) — 按指定权值注意球员
- SetForceSeeBall(···) — 设置强制看球 (要能看到才会去看)
- SetForceSeePlayer(···) — 设置强制看某人
- SetCritical(···) — 设置是否强制使用窄视角
- ForbidDecision(···) — 禁止视觉决策
- SetCanTurn(···) — 设置视觉决策时是否考虑转身动作
- ChangeViewWidth(···) — 高层决定改变视角
- VisualSystem.{h, cpp}

VisualSystem

- RaiseBall(···) — 按指定权值注意球
- RaisePlayer(···) — 按指定权值注意球员
- SetForceSeeBall(···) — 设置强制看球 (要能看到才会去看)
- SetForceSeePlayer(···) — 设置强制看某人
- SetCritical(···) — 设置是否强制使用窄视角
- ForbidDecision(···) — 禁止视觉决策
- SetCanTurn(···) — 设置视觉决策时是否考虑转身动作
- ChangeViewWidth(···) — 高层决定改变视角
- VisualSystem.{h, cpp}

VisualSystem

- RaiseBall(···) — 按指定权值注意球
- RaisePlayer(···) — 按指定权值注意球员
- SetForceSeeBall(···) — 设置强制看球 (要能看到才会去看)
- SetForceSeePlayer(···) — 设置强制看某人
- SetCritical(···) — 设置是否强制使用窄视角
- ForbidDecision(···) — 禁止视觉决策
- SetCanTurn(···) — 设置视觉决策时是否考虑转身动作
- ChangeViewWidth(···) — 高层决定改变视角
- VisualSystem.{h, cpp}

VisualSystem

- RaiseBall(···) — 按指定权值注意球
- RaisePlayer(···) — 按指定权值注意球员
- SetForceSeeBall(···) — 设置强制看球 (要能看到才会去看)
- SetForceSeePlayer(···) — 设置强制看某人
- SetCritical(···) — 设置是否强制使用窄视角
- ForbidDecision(···) — 禁止视觉决策
- SetCanTurn(···) — 设置视觉决策时是否考虑转身动作
- ChangeViewWidth(···) — 高层决定改变视角
- VisualSystem.{h, cpp}

VisualSystem

- RaiseBall(···) — 按指定权值注意球
- RaisePlayer(···) — 按指定权值注意球员
- SetForceSeeBall(···) — 设置强制看球 (要能看到才会去看)
- SetForceSeePlayer(···) — 设置强制看某人
- SetCritical(···) — 设置是否强制使用窄视角
- ForbidDecision(···) — 禁止视觉决策
- SetCanTurn(···) — 设置视觉决策时是否考虑转身动作
- ChangeViewWidth(···) — 高层决定改变视角
- VisualSystem.{h, cpp}

VisualSystem

- RaiseBall(···) — 按指定权值注意球
- RaisePlayer(···) — 按指定权值注意球员
- SetForceSeeBall(···) — 设置强制看球 (要能看到才会去看)
- SetForceSeePlayer(···) — 设置强制看某人
- SetCritical(···) — 设置是否强制使用窄视角
- ForbidDecision(···) — 禁止视觉决策
- SetCanTurn(···) — 设置视觉决策时是否考虑转身动作
- ChangeViewWidth(···) — 高层决定改变视角
- VisualSystem.{h, cpp}

VisualSystem

- RaiseBall(···) — 按指定权值注意球
- RaisePlayer(···) — 按指定权值注意球员
- SetForceSeeBall(···) — 设置强制看球 (要能看到才会去看)
- SetForceSeePlayer(···) — 设置强制看某人
- SetCritical(···) — 设置是否强制使用窄视角
- ForbidDecision(···) — 禁止视觉决策
- SetCanTurn(···) — 设置视觉决策时是否考虑转身动作
- ChangeViewWidth(···) — 高层决定改变视角
- VisualSystem.{h, cpp}

VisualSystem

- RaiseBall(···) — 按指定权值注意球
- RaisePlayer(···) — 按指定权值注意球员
- SetForceSeeBall(···) — 设置强制看球 (要能看到才会去看)
- SetForceSeePlayer(···) — 设置强制看某人
- SetCritical(···) — 设置是否强制使用窄视角
- ForbidDecision(···) — 禁止视觉决策
- SetCanTurn(···) — 设置视觉决策时是否考虑转身动作
- ChangeViewWidth(···) — 高层决定改变视角
- VisualSystem.{h, cpp}

VisualSystem

- RaiseBall(···) — 按指定权值注意球
- RaisePlayer(···) — 按指定权值注意球员
- SetForceSeeBall(···) — 设置强制看球 (要能看到才会去看)
- SetForceSeePlayer(···) — 设置强制看某人
- SetCritical(···) — 设置是否强制使用窄视角
- ForbidDecision(···) — 禁止视觉决策
- SetCanTurn(···) — 设置视觉决策时是否考虑转身动作
- ChangeViewWidth(···) — 高层决定改变视角
- VisualSystem.{h, cpp}

CommunicateSystem

- SendBallStatus(···) — 广播球状态信息
- SendTeammateStatus(···) — 广播队友状态信息
- SendOpponentStatus(···) — 广播对手状态信息
- ParseReceivedTeammateMsg(···) — 解析听到的队友“喊话”
- CommunicateSystem.{h, cpp}

CommunicateSystem

- SendBallStatus(···) — 广播球状态信息
- SendTeammateStatus(···) — 广播队友状态信息
- SendOpponentStatus(···) — 广播对手状态信息
- ParseReceivedTeammateMsg(···) — 解析听到的队友“喊话”
- CommunicateSystem.{h, cpp}

CommunicateSystem

- SendBallStatus(···) — 广播球状态信息
- SendTeammateStatus(···) — 广播队友状态信息
- SendOpponentStatus(···) — 广播对手状态信息
- ParseReceivedTeammateMsg(···) — 解析听到的队友“喊话”
- CommunicateSystem.{h, cpp}

CommunicateSystem

- SendBallStatus(···) — 广播球状态信息
- SendTeammateStatus(···) — 广播队友状态信息
- SendOpponentStatus(···) — 广播对手状态信息
- ParseReceivedTeammateMsg(···) — 解析听到的队友“喊话”
- CommunicateSystem.{h, cpp}

CommunicateSystem

- SendBallStatus(···) — 广播球状态信息
- SendTeammateStatus(···) — 广播队友状态信息
- SendOpponentStatus(···) — 广播对手状态信息
- ParseReceivedTeammateMsg(···) — 解析听到的队友“喊话”
- CommunicateSystem.{h, cpp}

DynamicDebug

- 正常比赛时，Client 是实时运行的，无法直接使用 gdb 调试。让 Client 在运行时记录所有从 Server 处收到的“字符串”到文件 — MsgLog。通过读取 MsgLog，模拟从 Server 收到“字符串”，让 Client 按“步进”方式运行。从而可以使用 gdb 进行调试，这就是动态调试 (DynamicDebug)。
- 软件依赖：gdb, cgdb

DynamicDebug

- 正常比赛时，Client 是实时运行的，无法直接使用 gdb 调试。让 Client 在运行时记录所有从 Server 处收到的“字符串”到文件 — MsgLog。通过读取 MsgLog，模拟从 Server 收到“字符串”，让 Client 按“步进”方式运行。从而可以使用 gdb 进行调试，这就是动态调试 (DynamicDebug)。
- 软件依赖：gdb, cgdb

DynamicDebug 操作流程

1. 打开 save_server_message 选项，正常跑比赛，MsgLog 记录在 Logfiles/目录内；
2. ./dd 球员号码，按照“步进”方式运行 Client；
3. ./dbg，使用 gdb attach 到 Client 进程上，进行调试。

DynamicDebug 操作流程

1. 打开 `save_server_message` 选项，正常跑比赛，MsgLog 记录在 `Logfiles/`目录内；
2. `./dd` 球员号码，按照“步进”方式运行 Client；
3. `./dbg`，使用 gdb attach 到 Client 进程上，进行调试。

DynamicDebug 操作流程

1. 打开 `save_server_message` 选项，正常跑比赛，MsgLog 记录在 `Logfiles/`目录内；
2. `./dd` 球员号码，按照“步进”方式运行 Client；
3. `./dbg`，使用 gdb attach 到 Client 进程上，进行调试。

Logger

记录 Client 运行期间的 Log，方便掌握 Client 的决策过程。

- TextLogger — 以文本形式记录 Log，可以使用各种文本编辑工具查看
- SightLogger — 以 rcg 格式记录 Log，可通过 rcsslogplayer 可视化显示
  - 如果记录了 MsgLog，则可以通过 `./genlog` 球员号码产生指定球员的 SightLog，并且可以通过 `./showlog` 调用 rcsslogplayer 显示相应的 SightLog。

Logger

记录 Client 运行期间的 Log，方便掌握 Client 的决策过程。

- TextLogger — 以文本形式记录 Log，可以使用各种文本编辑工具查看
- SightLogger — 以 rcg 格式记录 Log，可通过 rcsslogplayer 可视化显示
  - 如果记录了 MsgLog，则可以通过 `./genlog` 球员号码产生指定球员的 SightLog，并且可以通过 `./showlog` 调用 rcsslogplayer 显示相应的 SightLog。

Logger

记录 Client 运行期间的 Log，方便掌握 Client 的决策过程。

- TextLogger — 以文本形式记录 Log，可以使用各种文本编辑工具查看
- SightLogger — 以 rcg 格式记录 Log，可通过 rcsslogplayer 可视化显示
  - 如果记录了 MsgLog，则可以通过 `./genlog` 球员号码产生指定球员的 SightLog，并且可以通过 `./showlog` 调用 rcsslogplayer 显示相应的 SightLog。

TimeTest

Client 每周期决策时间有限，仅为 100ms，有时需要查看某一函数或某处代码时间开销到底怎么样，这时候可以使用 TimeTest 工具。使用方法：

1. 在合适的位置增加代码：`TIMETEST(测试名称)`；
2. 正常跑比赛；
3. 时间开销信息记录在 `Test` 目录内。

TimeTest

Client 每周期决策时间有限，仅为 100ms，有时需要查看某一函数或某处代码时间开销到底怎么样，这时候可以使用 TimeTest 工具。使用方法：

1. 在合适的位置增加代码：`TIMETEST(测试名称)`；
2. 正常跑比赛；
3. 时间开销信息记录在 `Test` 目录内。

TimeTest

Client 每周期决策时间有限，仅为 100ms，有时需要查看某一函数或某处代码时间开销到底怎么样，这时候可以使用 TimeTest 工具。使用方法：

1. 在合适的位置增加代码：`TIMETEST(测试名称)`；
2. 正常跑比赛；
3. 时间开销信息记录在 `Test` 目录内。

Plotter

可以实时或在动调时调用 Gnuplot 可视化显示数据，依赖软件 Gnuplot，使用方法：

1. 写好绘图函数，确保会以适当的方式被调用；
2. 打开 `use_plotter` 选项，运行 Client。

Plotter

可以实时或在动调时调用 Gnuplot 可视化显示数据，依赖软件 Gnuplot，使用方法：

1. 写好绘图函数，确保会以适当的方式被调用；
2. 打开 `use_plotter` 选项，运行 Client。

参考资料

- 蓝鹰仿真 2D 机器人足球队，*WrightEagleBASE-2.0.1*，http://www.wrighteagle.org/2D

谢谢大家!

Q & A