规则系统

某些类型的游戏包括复杂的规则集。例如，一款回合制战斗的角色扮演游戏可能包括规则，当对立角色进入同一空间时会发生什么：谁在由此产生的斗争中占上风？什么时候，如果有的话，任何一个角色都会有机会攻击对方？当其他角色试图攻击或与其中一名战斗人员互动时会发生什么？这些规则集可能会变得如此复杂，以至于编程语言中的条件逻辑语句变得笨重。

一些最有趣的游戏类型包括具有*紧急行为的系统*——在简单规则下，简单实体之间的交互导致整个系统中的有趣模式。例如，在动作游戏中，单个敌人角色的目标可能取决于角色的生命水平、该敌人最近多久看到玩家角色、附近有多少其他敌人角色、玩家击败敌人的频率以及其他变量。这些因素加在一起可能导致游戏玩法的现实变化，一些敌人成群结队地攻击玩家，另一些人则逃跑。

在GameplayKit中，*规则系统*解决了这两个问题。通过提供将游戏逻辑的某些元素视为数据的抽象，规则系统可以帮助您将游戏分解为功能性、可重用、可扩展的部分。通过结合模糊逻辑，规则系统可以将决策视为连续变量，而不是离散状态，甚至导致规则的简单组合的复杂行为。

设计规则系统

GameplayKit包括构建规则系统的两大类：`GKRule`，代表基于外部状态的具体决定，以及`GKRuleSystem`，根据状态数据评估一组规则以确定一组事实。

规则做出基本决定

`GKRule`对象有两个部分：*谓词*和*动作*。规则的谓词是决策部分——它评估一组状态信息并返回布尔结果。规则的操作是代码，其执行仅在谓词产生真实结果时才会触发。GameplayKit总是在规则系统中评估规则。通常，规则的谓词测试规则系统维护的状态数据，其操作要么改变系统的状态，要么断言事实——这是规则系统输出的一部分。

因为规则评估系统的状态并修改系统的属性，所以规则不需要本地状态数据。这种设计允许您将单个规则设计为具有定义良好行为的功能单元，并在任何规则系统中重用它们。

规则系统使用规则和状态来陈述事实

`GKRuleSystem`对象有三个关键部分：规则议程、状态数据和事实。

- **The agenda.** You combine a set of rules by adding them to the `agenda` of a rule system object (with the `addRule:` or `addRulesFromArray:` method). By default, a system evaluates rules in the order they were added to its agenda—to make sure some rules are always evaluated before others, change the `salience` property of those rules.
- **State information.**The rule system’s `state` dictionary contains information rules can be tested against. This dictionary can reference anything useful to your set of rules, be that simple data like strings and numbers or custom classes from your game model. How you write rules determines the structure of the data those rules will use.
- **事实。**一个*事实*代表了从系统中规则评估中得出的结论，可以是任何类型的对象——通常，简单的数据对象，如字符串就足够了，但也可以使用游戏模型中的自定义对象。一个事实的*会员级别*是决定其在系统事实集中存在的数字。1.0级的事实包括在内，零级的事实不包括。当规则系统评估其规则时，规则操作可以*断言*事实，将其添加到系统中，或*撤回*事实，将其从系统中删除。

然而，会员等级不必简单地是一个二进制状态——这一事实可以在0.0到1.0之间有任何等级。通过允许事实在等级上有所不同，您可以使用规则系统来定义*模糊逻辑*，其中规则可以得出具有可变强度、真实值或置信度的结论。当一条规则断言或撤回一个事实时，它会增加或减少该事实的等级。

Your game can then use the relative strengths of various facts to create gameplay effects. For example, whether an enemy moves to attack the player might be based on a “vulnerable” fact, whose grade is based on several rules. One rule might increase the grade if the player is nearby, another might reduce the grade if the player is in full health, and so on. You can also build rule systems in which some rules assert facts, and other rules examine the grades of those facts in order to assert or retract further conclusions. You might even use the grades of facts in the system to vary the influence of corresponding factors in the game—for example, you might tie the goal weights collected in a `GKBehavior` object to the grades of corresponding facts.

用规则系统构建游戏

GameplayKit提供了三种创建`GKRule`对象的方法。您选择的取决于您希望如何将规则构建纳入开发过程。

- The `ruleWithPredicate:assertingFact:grade:` and `ruleWithPredicate:retractingFact:grade:` methods create rules whose predicate is decided by an `NSPredicate` object, and whose action is hard-coded to assert or retract a specific fact. Because the `NSPredicate` class can describe conditional logic that accesses data through key paths, and because the rule’s action is fixed at its creation, this option produces rules that can be easily archived. This design lends itself well to data-driven workflows where game designers can tweak a game’s rule systems without writing code. (You can even use the `NSRuleEditor` class in OS X to support editing rules at run time.) However, these rules assert or retract facts with a constant grade, so they don’t work well for fuzzy logic systems involving facts whose grade is continuously variable.
- The `ruleWithBlockPredicate:action:` factory method creates a rule from two blocks: a predicate block to evaluate the rule and return a Boolean result, and an action block to invoke when the predicate is true. This option allows you to quickly and easily create rules in code, including fuzzy logic rules. However, the resulting rules are not archivable—they can exist only in the in-memory object graph of your game.
- 您可以子类`GKRule`创建自己的规则类，实现`evaluatePredicateWithSystem:`方法，代码来决定规则的谓词，以及使用代码执行操作的`performActionWithSystem:`方法。或者，您可以子类`GKNSPredicateRule`，它使用`NSPredicate`对象来决定规则的谓词，因此您只需要编写自己的代码来执行规则的操作。由于规则的关键属性在您的子类中实现，因此此选项的灵活性和可重用性取决于您如何设计这些类。

设计规则后，构建和使用规则系统需要四个基本步骤：

1. 在初始化期间，创建一个`GKRuleSystem`对象并填充其规则列表。
2. Set the rule system’s `state` dictionary to contain whatever game information the rules should act upon.

   根据您如何构建此信息，您可能会在初始化期间设置一次状态，以包含其内部状态在游戏过程中发生变化的对象。或者，您可以将状态设置为包含静态对象，并在需要评估规则系统时更新状态以总结游戏中的当前条件。
3. To evaluate rules, call the rule system’s `evaluate` method.

   If you periodically evaluate the same rule system, call its `reset` method before evaluating to clear the results of prior evaluations.
4. 在评估该系统后，检查其一组断言的事实，以得出结论并选择行动方案。

以下部分讨论规则系统在两个不同游戏中的示例使用。

游戏特效的基本规则系统

有关规则系统运行的简单示例，请参阅迷宫示例代码项目。在这个游戏中（几个经典街机游戏的变体，实体和组件、状态机器和寻路章节中讨论了其他功能），一个简单的规则系统决定了敌人角色正常状态的行为。清单8-1显示了建立和使用该规则系统的相关部分。

> 注
>
> 本节讨论示例代码项目迷宫：GameplayKit入门的功能。下载它，在Xcode中跟随。

清单8-1 A最小规则系统

```
- (instancetype)initWithGame:(AAPLGame *)game entity:(AAPLEntity *)entity {
    self = [super initWithGame:game entity:entity];
    if (self) {
        // 1
        _ruleSystem = [[GKRuleSystem alloc] init];
        NSPredicate *playerFar = [NSPredicate predicateWithFormat:@"$distanceToPlayer.floatValue >= 10.0"];
        [_ruleSystem addRule:[GKRule ruleWithPredicate:playerFar assertingFact:@"hunt" grade:1.0]];
        NSPredicate *playerNear = [NSPredicate predicateWithFormat:@"$distanceToPlayer.floatValue < 10.0"];
        [_ruleSystem addRule:[GKRule ruleWithPredicate:playerNear retractingFact:@"hunt" grade:1.0]];
    }
    return self;
}


- (void)updateWithDeltaTime:(NSTimeInterval)seconds {
    // 2
    NSUInteger distanceToPlayer = [self pathToPlayer].count;
    self.ruleSystem.state[@"distanceToPlayer"] = @(distanceToPlayer);

    // 3
    [self.ruleSystem reset];
    [self.ruleSystem evaluate];

    // 4
    self.hunting = ([self.ruleSystem gradeForFact:@"hunt"] > 0.0);
    if (self.hunting) {
        [self startFollowingPath:[self pathToPlayer]];
    } else {
        [self startFollowingPath:[self pathToNode:self.scatterTarget]];
    }
```

在本示例中，使用规则系统有四个步骤：

1. `initWithGame:entity:`方法创建一个`GKRuleSystem`实例，并使用`GKRule`对象填充它。规则系统将被重用，因此此示例将其保留在属性中。
2. The per-frame `updateWithDeltaTime:` method uses the rule system to determine what an enemy character will do next. First, it calculates the distance from the enemy character to the player character, and stores that information in the rule system’s `state` dictionary.
3. 接下来，此方法重置规则系统（将先前计算的规则返回到系统议程，并清除过去对更新方法的调用中断言的任何事实），并使用新更新的状态信息评估规则系统。
4. Finally, the update method uses the `gradeForFact:` method to test whether the rule system asserted a fact, and based on that information, chooses a behavior for the enemy character.

这段代码显示了规则系统的最小使用——事实上，它做任何用简单的`if`语句无法实现的事情。然而，通过扩展此示例，您可以轻松地为游戏添加复杂行为。例如：

- 添加更多状态信息，例如该敌人角色与对方敌人角色之间的距离，游戏开始以来的时间，或玩家完成关卡的进展。
- 添加更多测试附加状态信息的规则。添加更多代表所添加规则结论的事实，并根据系统所主张的事实组合为敌人角色选择一种行为。
- 将设置每个规则测试条件的`NSPredicate`定义从源代码中移到数据文件中，以便您或团队的其他成员可以修改游戏的行为，而无需使用Xcode从源代码重新编译游戏。
- Alternatively, use the `ruleWithBlockPredicate:action:` method or create custom `GKRule` subclasses to make rules that assert facts with variable grade, and use the fuzzy logic operators `minimumGradeForFacts:` and `maximumGradeForFacts:` to choose a behavior based on which combination of facts the system has asserted with the highest confidence.

一种用于紧急行为的模糊逻辑规则系统

DemoBots示例代码项目使用上述几个选项来创建具有有趣行为的字符。在这个游戏的每个关卡中，有几个“任务机器人”角色，每个角色都可以处于“好”或“坏”状态。游戏的目的是让玩家（“调试机器人”角色）找到每个坏机器人并“调试”它，将机器人更改为良好状态。好机器人只是在关卡中遵循预先定义的路径，但坏机器人可以以几种方式中的任何一种——它可能会试图猎杀和攻击玩家角色，寻找好机器人并把它们变成坏人，或者在关卡中巡逻，寻找要制造麻烦。

> 注
>
> 本章讨论示例代码项目*DemoBots：使用SpriteKit和GameplayKit构建跨平台游戏*的功能。下载它，在Xcode中跟随。

本游戏使用规则系统来决定坏机器人随时采取哪些行动。与前一个例子不同，这个规则系统使用模糊逻辑——这个系统中的规则断言具有可变等级的事实，在评估其规则系统后，机器人比较几个事实的等级，以决定采取哪种行动。

本系统中的每个规则都使用有关游戏状态的摘要信息的快照（请参阅示例代码项目中的`TaskBotSnapshot`类），以便多个规则类可以使用此类信息，而无需重新计算。游戏仅在评估规则系统时生成快照，并通过定期评估规则系统，而不是在每个帧上评估规则系统，从而最大限度地减少不必要的计算。

为了构建这个系统，DemoBots定义了一组`GKRule`子类，每个子类执行一个简单的计算来确定相关事实的等级。例如，清单8-2显示了用于确定当前级别中坏机器人数量是否较低的规则类。

清单8-2 一个模糊逻辑规则的例子

```
class BadTaskBotPercentageLowRule: FuzzyTaskBotRule {
    override func grade() -> Float {
        return max(0.0, 1.0 - 3.0 * snapshot.badBotPercentage)
    }
    init() { super.init(fact: .BadTaskBotPercentageLow) }
}
```

当关卡包含零坏机器人时，这条规则会提供最高等级，并随着关卡中坏机器人与总机器人的比例增加而降低等级。当一个级别的三分之一或更多机器人是坏的时，规则的等级为零。

在DemoBots中，清单8-3中以缩写形式显示的超类为评估这些规则和用每条规则计算的等级断言事实提供了一个共同的基础设施。

模糊逻辑规则常见超类8-3 A清单

```
class FuzzyTaskBotRule: GKRule {
    var snapshot: EntitySnapshot!
    func grade() -> Float { return 0.0 }
    let fact: Fact

    init(fact: Fact) {
        self.fact = fact
        super.init()
        // Set the salience so that 'fuzzy' rules will evaluate first.
        salience = Int.max
    }

    override func evaluatePredicateWithSystem(system: GKRuleSystem) -> Bool {
        snapshot = system.state["snapshot"] as! EntitySnapshot
        if grade() >= 0.0 {
            return true
        }
        return false
    }

    override func performActionWithSystem(system: GKRuleSystem) {
        system.assertFact(fact.rawValue, grade: grade())
    }
}
```

有了这个基础设施，构建一个规则系统就是创建每个规则类的实例的简单问题。然后，在提供快照并评估规则系统后，`TaskBot`类的实例将系统所断言的任何事实的级别与最小和最大函数进行比较，如清单8-4所示。

列举8-4 从一个规则系统中得出的结论

```
// 1
let huntPlayerBotRaw = [
    ruleSystem.minimumGradeForFacts([Fact.BadTaskBotPercentageHigh.rawValue, Fact.PlayerBotNear.rawValue]),
    ruleSystem.minimumGradeForFacts([Fact.BadTaskBotPercentageMedium.rawValue, Fact.PlayerBotNear.rawValue]),
    ruleSystem.minimumGradeForFacts([Fact.BadTaskBotPercentageHigh.rawValue, Fact.PlayerBotFar.rawValue]),
    ruleSystem.minimumGradeForFacts([Fact.BadTaskBotPercentageHigh.rawValue, Fact.PlayerBotMedium.rawValue, Fact.GoodTaskBotMedium.rawValue]),
]
let huntPlayerBot = huntPlayerBotRaw.reduce(0.0, combine: max)
let huntTaskBotRaw = [
    ruleSystem.minimumGradeForFacts([Fact.BadTaskBotPercentageLow.rawValue, Fact.GoodTaskBotNear.rawValue]),
    ruleSystem.minimumGradeForFacts([Fact.BadTaskBotPercentageMedium.rawValue, Fact.GoodTaskBotNear.rawValue]),
    ruleSystem.minimumGradeForFacts([Fact.BadTaskBotPercentageLow.rawValue, Fact.PlayerBotMedium.rawValue, Fact.GoodTaskBotMedium.rawValue]),
    ruleSystem.minimumGradeForFacts([Fact.BadTaskBotPercentageMedium.rawValue, Fact.PlayerBotFar.rawValue, Fact.GoodTaskBotMedium.rawValue]),
    ruleSystem.minimumGradeForFacts([Fact.BadTaskBotPercentageLow.rawValue, Fact.PlayerBotFar.rawValue, Fact.GoodTaskBotFar.rawValue])
]
let huntTaskBot = huntTaskBotRaw.reduce(0.0, combine: max)

// 2
if huntPlayerBot >= huntTaskBot && huntPlayerBot > 0.0 {
    guard let playerBotAgent = state.playerBotTarget?.target.agent else { return }
    mandate = .HuntAgent(playerBotAgent)
} else if huntTaskBot > huntPlayerBot {
    mandate = .HuntAgent(state.nearestGoodTaskBotTarget!.target.agent)
} else {
    switch mandate {
    case .FollowBadPatrolPath:
        break
    default:
        let closestPointOnBadPath = closestPointOnPath(badPathPoints)
        mandate = .ReturnToPositionOnPath(float2(closestPointOnBadPath))
    }
}
```

此列表中的代码执行两项主要任务（由上面编号注释表示）：

1. 比较系统中事实的等级以得出结论。

   In fuzzy logic, the minimum function (in the example above, the `minimumGradeForFacts:` call) and the maximum function (the `reduce(_:combine:)` call) behave similarly to the AND and OR operators of Boolean logic. Thus, the `huntPlayerBotRaw` and `huntPlayerBot` calculations are similar to the following pseudocode:

   `huntPlayerBot = (PlayerBotNear AND BadTaskBotPercentageHigh) OR (PlayerBotNear AND BadTaskBotPercentageMedium) OR ...`

   However, unlike in Boolean logic, each element in this calculation is a grade—a level of confidence in an assertion—so the resulting `huntPlayerBot` value is also a grade. You can think of this as a probability: “The likelihood that I should hunt for the player is…”
2. 利用结论来选择行动方案。

   This set of `if` statements sets the `TaskBot` object’s `mandate` property to a value that determines its objective for the next few moments of gameplay. If the `huntPlayerBot` score is nonzero and greater than the `huntTaskBot` score, the bot moves to attack the player character; if the `huntTaskBot` score is greater, the bot moves to the nearest good bot to turn it bad; and if both scores are zero, the bot simply patrols its route.

   This objective holds until the next time the `TaskBot` object evaluates its rule system. In the meantime, per-frame update methods in other game components use the `mandate` value to drive the bot’s actions. Depending on the bot’s mandate, the `GKAgent` object responsible for the bot’s movement gets a set of goals that cause the bot to find and follow a path toward a target, or a different set of goals that cause the bot to patrol around the game level.

◀ 代理人、目标和行为