状态机

在几乎所有的游戏中，游戏相关的逻辑高度依赖于游戏的当前状态。例如，动画代码可能会根据玩家角色当前是行走、跳跃还是站立而改变。敌人移动代码可能取决于您模拟的情报对敌方角色做出的高级决定，例如是追逐弱势玩家还是逃离强大的玩家。即使游戏的哪些部分应该随时运行每帧更新代码，也可能取决于游戏是否正在播放、暂停，还是在菜单或剪切场景中。

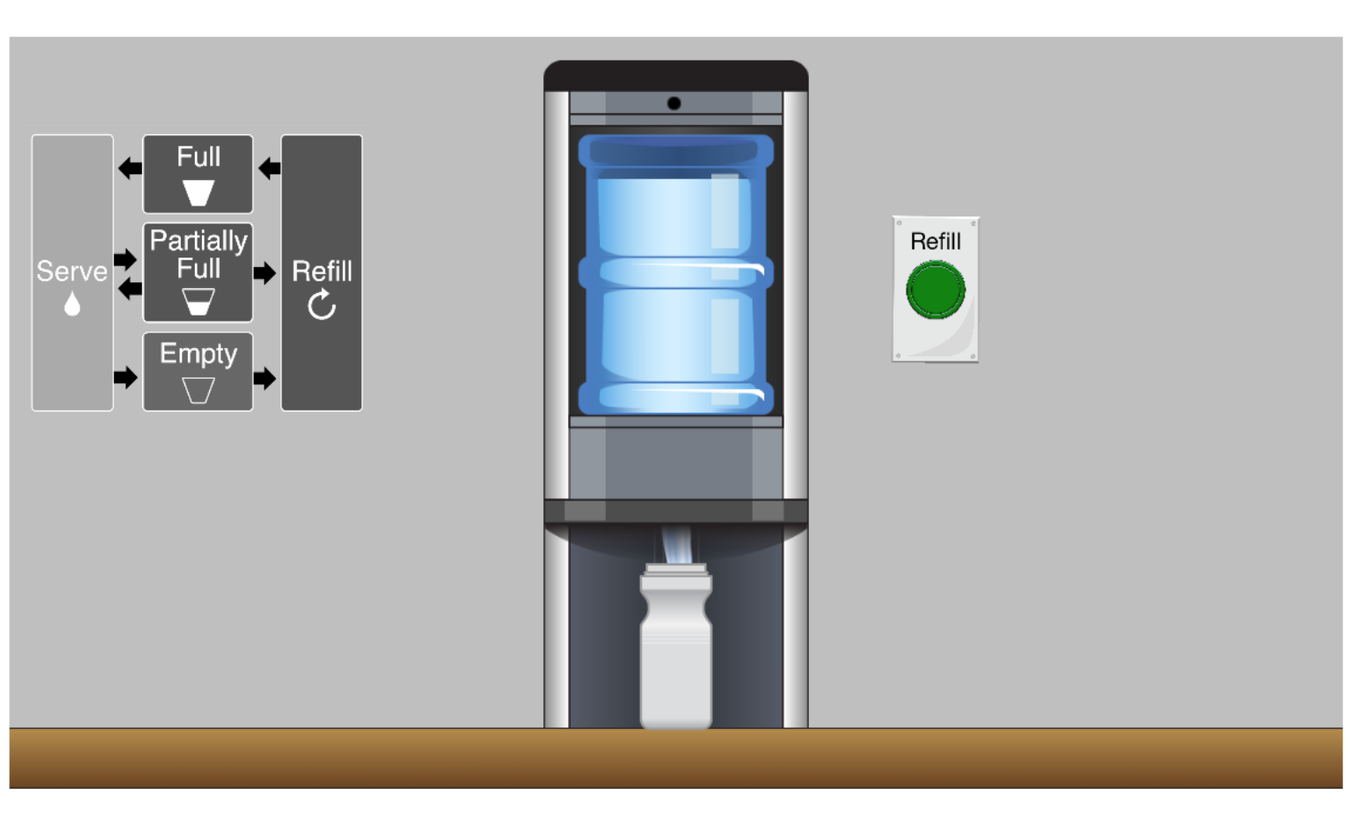
当您开始构建游戏时，很容易将所有与状态相关的代码放在一个地方——例如，在SpriteKit游戏的每帧更新方法中。然而，随着游戏的发展和复杂性，这种单一方法可能会变得难以维护或进一步扩展。

相反，它可以帮助正式定义游戏中的不同状态，以及确定允许哪些状态之间转换的规则。这些形式定义被称为*状态机*。然后，您可以将相关状态与任何代码关联，这些代码确定您的游戏在特定状态下对每帧更新所做的操作，何时过渡到另一种状态，以及转换可能导致哪些次要操作。通过使用状态机组织代码，您可以更容易地推理游戏中的复杂行为。

示例：分配器

在本章深入了解状态机的详细信息之前，您可以通过下载示例代码项目*Dispenser: GameplayKit State Machine Basics*快速体验它们如何简化您的游戏设计。这个简单的游戏模拟了一个饮水机，如图4-1所示。正如游戏显示的图表所示，分配器一次只能处于一种状态：空、满、部分满、上菜或重新充。使用状态机可以轻松形式化和执行此行为，并帮助您组织针对每种状态的游戏逻辑部分。

图4-1 点胶机样本代码项目在行动



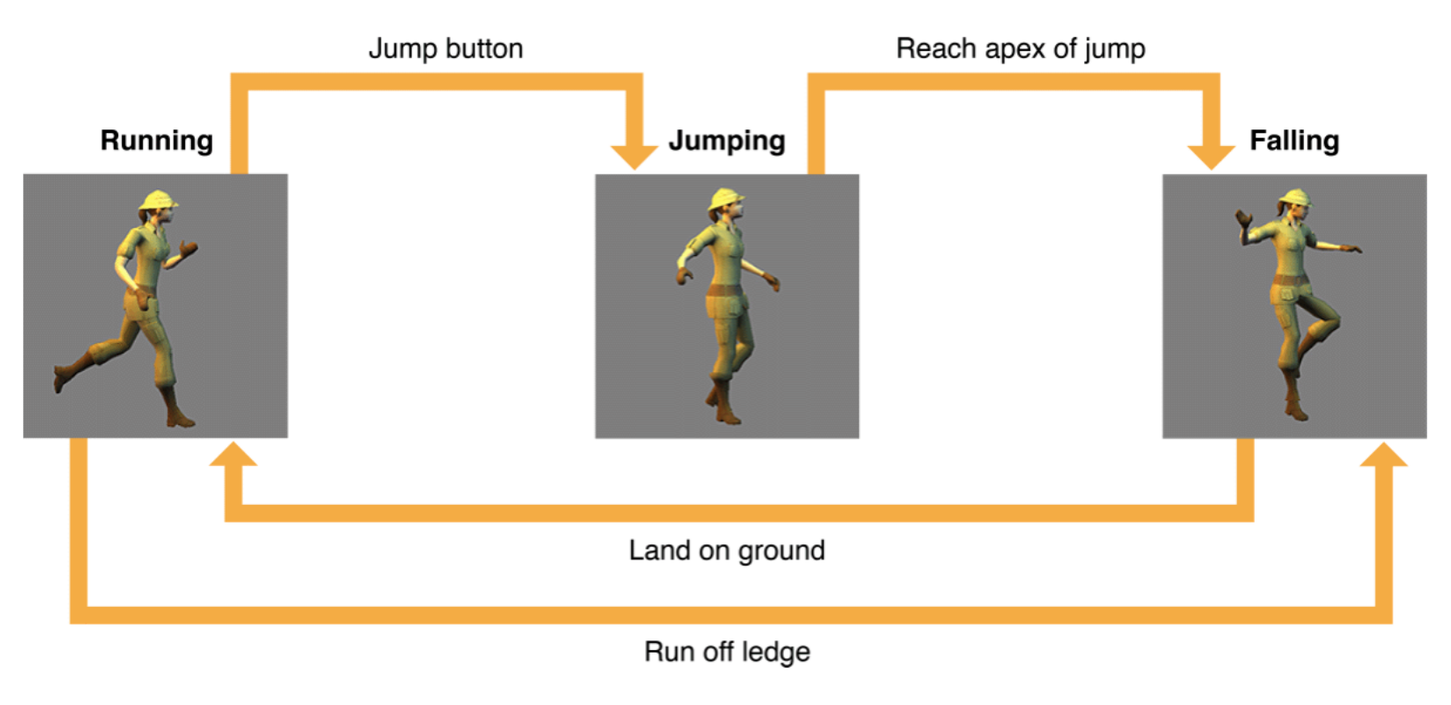
使用状态机进行设计

状态机可以应用于游戏中涉及状态依赖行为的任何部分。以下示例介绍了几种不同状态机的设计。

动画状态机器

考虑一款2D“无尽的奔跑”游戏 — — 在这种风格的游戏中，玩家角色自动运行，玩家必须按下跳跃按钮才能通过障碍物。跑步和跳跃有单独的动画，在跑步或跳跃时，玩家的位置需要更新。这种设计可以表示为具有三种状态的状态机，如图4-2所示。

图4-2 A动画状态机

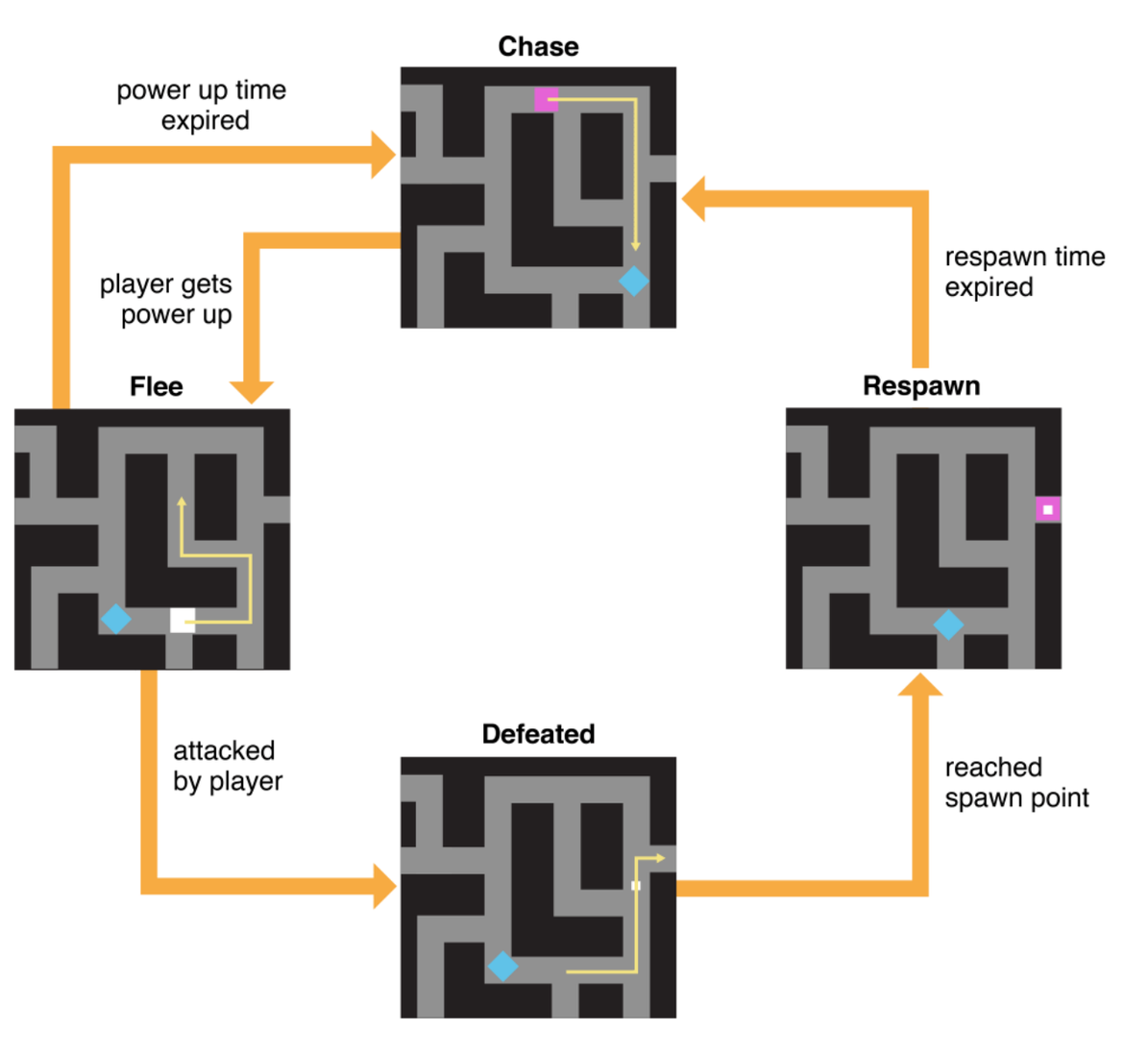


- **跑步。**处于此状态时，循环运行动画。如果玩家按下跳转按钮，则切换到跳转状态。如果玩家从窗台上跑下来，请切换到坠落状态。
- **跳跃。**进入此状态后，播放音效，启动一次跳跃动画。在这种状态下，将角色移动一段短距离（每帧），随着重力减速。当上升速度达到零时，切换到下降状态。
- **坠落。**进入此状态后，开始一次掉落动画。在这种状态下，将角色向下移动一段短距离（每帧）。到达地面后，切换到运行状态，播放一次性着陆动画和音效。

敌行为状态机器

考虑一款街机动作游戏，其中包含追逐玩家的敌人角色。玩家偶尔可以获得一种能量，使敌人暂时容易受到攻击，而失败的敌人在重新出现之前会离开游戏一段时间。每个敌人角色可以使用自己的状态机实例，状态如下，如图4-3所示：

图4-3 A敌行为状态机



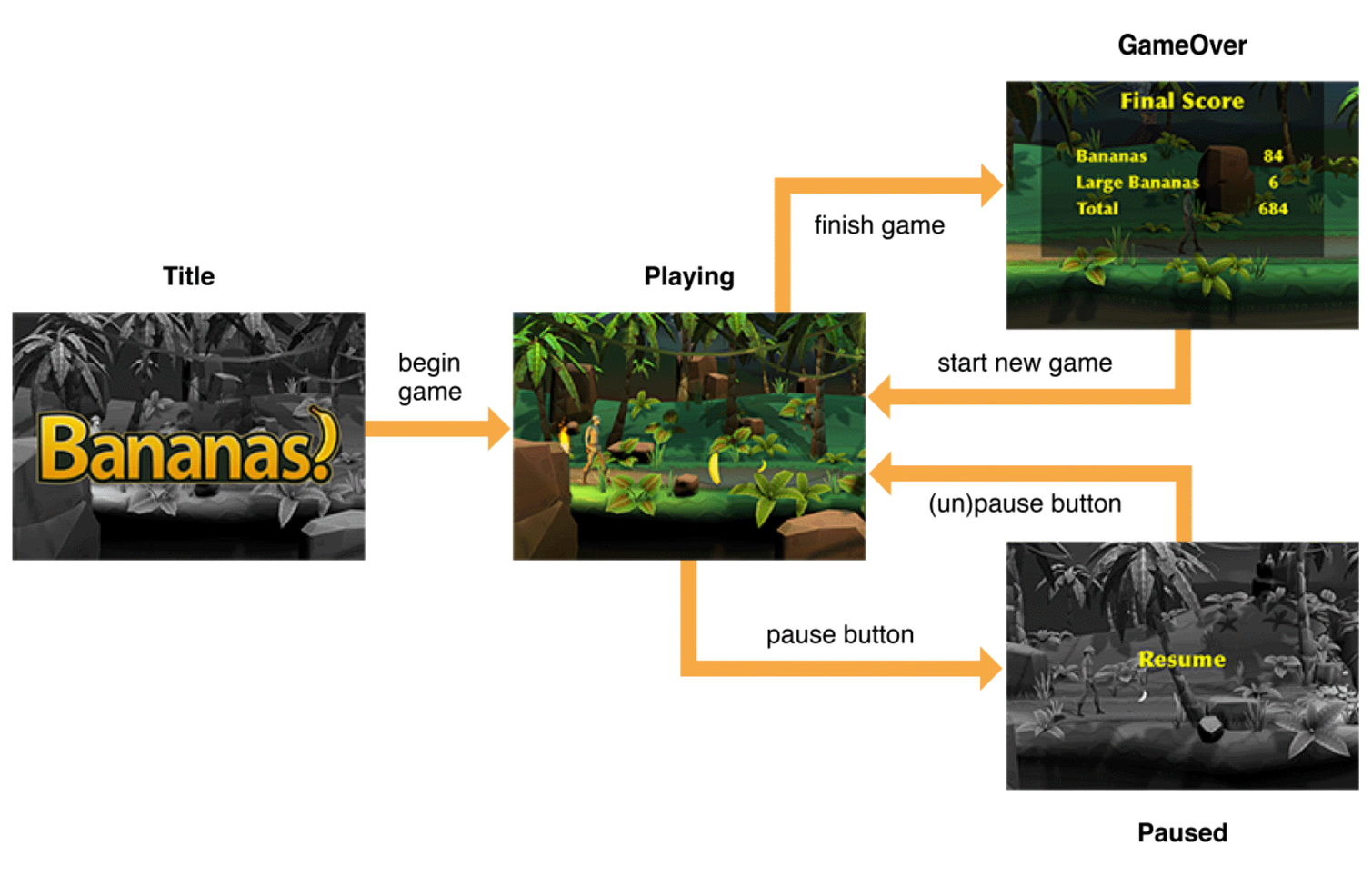
- **追逐。**进入此状态后，显示敌人的正常外观。在这种状态下，改变每帧的位置以追击玩家。如果玩家获得电源，请切换到逃跑状态。
- **逃跑。**进入此状态后，显示一个易损的外观。在这种状态下，更改每帧上的位置，以避免玩家。一段时间后，在这个状态下，回到大通状态。如果受到玩家攻击，请切换到失败状态。
- **失败。**进入此状态后，显示动画并增加玩家的分数。在这种状态下，将敌人的剩余部分移到中心位置，稍后将在那里重新出现。到达该位置后，切换到重生状态。
- **重生。**在这种状态下，只需跟踪进入后花费的时间。一段时间后，回到大通状态。

下面的“使用状态机构建游戏”部分以及迷宫示例代码项目进一步说明了这种状态机。

一款用于游戏UI的状态机

在几乎任何游戏中，用户界面取决于游戏的状态，反之亦然。例如，暂停游戏可能会显示菜单屏幕，当游戏暂停时，正常的游戏代码不应生效。您可以使用一组状态来控制游戏的整体用户界面，如下图4-4所示：

图4-4 A游戏UI状态机



- **标题屏幕。**该应用程序以此状态打开。进入此状态后，显示标题屏幕。在这种状态下，为标题屏幕添加动画。用户输入后切换到播放状态。
- **玩。**在这种状态下，调用每帧更新逻辑进行游戏。在适当的用户输入时切换到暂停状态。
- **暂停。**进入此状态后，在游戏屏幕上应用视觉效果，以指示游戏暂停。退出状态时的效果去掉。（在此状态下无需暂停游戏操作，因为游戏逻辑仅在播放状态下调用。）在适当的用户输入时，切换回播放状态。
- **游戏结束。**进入此状态时，显示一个总结玩家操作和得分的用户界面，并响应操作该用户界面中任何交互元素的事件。在适当的用户输入后切换到播放状态（启动新游戏）。

这种状态机设计可以很容易地扩展到许多游戏。例如，额外的菜单状态可以处理一系列复杂的菜单，或者过场状态可以运行场景动画，但不处理输入。

用状态机构建游戏

In GameplayKit, a state machine is an instance of the `GKStateMachine` class. For each state, you define the actions that occur while in that state, or when transitioning into or out of that state, by creating a custom subclass of `GKState`. At any one time, a state machine has exactly one current state. When you perform per-frame update logic for your game objects (for example, from within the `update:` method of a SpriteKit scene or the `renderer:updateAtTime:` method of a SceneKit render delegate), call the `updateWithDeltaTime:` method of the state machine, and it in turn calls the same method on its current state object. When your game logic requires a change in state, call the state machine’s `enterState:` method to choose a new state.

迷宫示例代码项目（已见实体和组件一章）实现了几个经典街机游戏的变体。本作使用上面总结的单独实例状态机（参见敌方行为状态机）来驱动几个敌人角色。通常，敌人会追逐玩家，但当玩家获得能量时，他们就会逃跑。被击败后，他们回到重生点，然后在短时间内再次出现。

> 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。

定义国家及其行为

Each state in the state machine is a subclass of `GKState` containing custom code that implements state-specific behavior. This state machine uses four state classes: `AAPLEnemyChaseState`, `AAPLEnemyFleeState`, `AAPLEnemyDefeatedState`, and `AAPLEnemyRespawnState`. All four state classes make use of general information about the game world, so all four inherit from an `AAPLEnemyState` class that defines properties and a common initializer used by all state classes in the game. Listing 4-1 summarizes the definitions of these classes.

清单4-1 敌人国家定义

```
@interface AAPLEnemyState : GKState
@property (weak) AAPLGame *game;
@property AAPLEntity *entity;
- (instancetype)initWithGame:(AAPLGame *)game entity:(AAPLEntity *)entity;
// ...
@end

@interface AAPLEnemyChaseState : AAPLEnemyState
@end

@interface AAPLEnemyFleeState : AAPLEnemyState
@end

@interface AAPLEnemyDefeatedState : AAPLEnemyState
@property GKGridGraphNode *respawnPosition;
@end

@interface AAPLEnemyRespawnState : AAPLEnemyState
@end
```

Most of these state classes need no additional public properties—all information they need about the game comes from their reference to the main `AAPLGame` object. This reference is weak, because the state objects are owned by state machines, which in turn are owned by entities, each of which is owned by the `Game` object.

然后，每个状态类通过重写`GKState`的回车、退出和更新方法来定义特定于状态的行为。例如，清单4-2总结了Flee状态的实现。

上市4-2 恩美大通国家实施

```
- (BOOL)isValidNextState:(Class __nonnull)stateClass {
    return stateClass == [AAPLEnemyChaseState class] ||
        stateClass == [AAPLEnemyDefeatedState class];
}
- (void)didEnterWithPreviousState:(__nullable GKState *)previousState {
    AAPLSpriteComponent *component = (AAPLSpriteComponent *)[self.entity componentForClass:[AAPLSpriteComponent class]];
    [component useFleeAppearance];

    // Choose a target location to flee towards.
    // ...
}

- (void)updateWithDeltaTime:(NSTimeInterval)seconds {
    // If the enemy has reached its target, choose a new target.
    // ...

    // Flee towards the current target point.
    [self startFollowingPath:[self pathToNode:self.target]];
}
```

The state machine calls a state object’s `didEnterWithPreviousState:` method when that state becomes the machine’s current state. In the Flee state, this method uses the game’s `SpriteComponent` class to change the appearance of the enemy character. (See the Entities and Components chapter for discussion of this class.) This method also chooses a random location in the game level for the enemy to flee toward.

`updateWithDeltaTime:`方法为每帧动画调用（最终，来自显示游戏的SpriteKit场景的`update:`方法）。在这种方法中，Flee状态不断重新计算通往目标位置的路线，并设置沿该路线移动的敌人角色。（有关此方法实现的更深入讨论，请参阅寻路一章。）

您可以通过让每个类覆盖`isValidNextState:`方法来强制状态类中的先决条件或不变量。在本游戏中，重生状态仅适用于被击败状态，而不是追逐或逃跑状态，才是有效的下一个状态。因此，`EnemyRespawnState`类中的代码可以安全地假设失败状态的任何副作用已经发生。

创建和驱动状态机

After you define `GKState` subclasses, you can use them to create a state machine. In the Maze game, an `AAPLIntelligenceComponent` object manages a state machine for each enemy character. (To learn more about the component-based architecture in this game, see the Entities and Components chapter.) Setting up a state machine is simply a matter of creating and configuring an instance of each state class, then creating a `GKStateMachine` instance that uses those objects, as shown in Listing 4-3.

清单4-3 定义状态机

```
@implementation AAPLIntelligenceComponent

- (instancetype)initWithGame:(AAPLGame *)game enemy:(AAPLEntity *)enemy startingPosition:(GKGridGraphNode *)origin {
    self = [super init];
    if (self) {
        AAPLEnemyChaseState *chase = [[AAPLEnemyChaseState alloc] initWithGame:game entity:enemy];
        AAPLEnemyFleeState *flee = [[AAPLEnemyFleeState alloc] initWithGame:game entity:enemy];
        AAPLEnemyDefeatedState *defeated = [[AAPLEnemyDefeatedState alloc] initWithGame:game entity:enemy];
        defeated.respawnPosition = origin;
        AAPLEnemyRespawnState *respawn = [[AAPLEnemyRespawnState alloc] initWithGame:game entity:enemy];

        _stateMachine = [GKStateMachine stateMachineWithStates:@[chase, flee, defeated, respawn]];
        [_stateMachine enterState:[AAPLEnemyChaseState class]];
    }
    return self;
}

- (void)updateWithDeltaTime:(NSTimeInterval)seconds {
    [self.stateMachine updateWithDeltaTime:seconds];
}

@end
```

然后，游戏为游戏中的每个敌人角色创建一个`AAPLIntelligenceComponent`类的实例。为了支持在每个状态类中运行每帧更新逻辑，`AAPLIntelligenceComponent`类使用其`updateWithDeltaTime:`方法在其状态机上调用相应的方法。反过来，状态机调用任何状态对象是其当前状态的更新方法——因此，例如，蔡斯状态的每帧更新逻辑仅在蔡斯状态处于活动状态时运行。

To run the state machine through its sequence of states, call its `enterState:` method whenever a state change is appropriate in your game. In this example, each enemy character’s state machine starts in the Chase state. The Maze game changes states in several places:

- The main `AAPLGame` object keeps track of whether the player has the power to defeat enemy characters. When the player gains or loses this power, the `setHasPowerup:` method changes all enemies’ current state to the Flee or Chase state, respectively, as shown below:

```
- (void)setHasPowerup:(BOOL)hasPowerup {
    static const NSTimeInterval powerupDuration = 10;
    if (hasPowerup != _hasPowerup) {
        // Choose the Flee or Chase state depending on whether the player has a powerup.
        Class nextState;
        if (!_hasPowerup) {
            nextState = [AAPLEnemyFleeState class];
        } else {
            nextState = [AAPLEnemyChaseState class];
        }
        // Make all enemies enter the chosen state.
        for (AAPLIntelligenceComponent *component in self.intelligenceSystem) {
            [component.stateMachine enterState:nextState];
        }
        self.powerupTimeRemaining = powerupDuration;
    }
    _hasPowerup = hasPowerup;
}
```

- AAPLGame对象还处理SpripriteKit报告的物理联系人。当玩家角色与敌人角色碰撞时，游戏首先使用该敌人的当前状态来确定他们互动的结果。如果敌人处于逃跑状态，玩家将击败敌人，该敌人将过渡到失败状态。
- `AAPLEnemyDefeatedState`类将战败的敌人带回起点，然后切换到重生状态。为了完成这些任务，失败状态找到通往起始位置的路径，然后使用游戏的`AAPLSpriteComponent`类生成SKAction序列。这个序列沿着路径移动敌人角色，在所有移动动作完成后——即角色在路径的末尾——它切换到重生状态。

  有关寻路和本游戏的`AAPLSpriteComponent`类的详细信息，请参阅寻路和实体和组件章节。
- The `AAPLEnemyRespawnState` class handles the wait time for each defeated enemy before it returns to play. This state class uses its `updateWithDeltaTime:` method to test an elapsed time property and switch to the Chase state accordingly:

```
- (void)updateWithDeltaTime:(NSTimeInterval)seconds {
    self.timeRemaining -= seconds;
    if (self.timeRemaining < 0) {
        [self.stateMachine enterState:[AAPLEnemyChaseState class]];
    }
}
```

◀ 实体和组件　　　　Minmax 策略师 ▶