一、Fishing

1. 类概述

FishingLayer 是一个管理钓鱼活动的类。它继承自 Layer，用于在游戏界面中显示一个钓鱼过程，包括进度条、按钮和钓鱼成功或失败后的处理逻辑。

该类主要负责：

显示钓鱼的进度条。

监听钓鱼按钮点击事件并启动钓鱼过程。

在钓鱼过程中实时更新进度条。

钓鱼成功或失败后处理相应的逻辑，如将钓到的鱼添加到背包中。

2. 成员变量

m\_progressBar (ProgressTimer\*): 显示钓鱼进度的进度条。

m\_infoLabel (Label\*): 用于显示信息的标签（当前代码中未使用）。

m\_startButton (MenuItemImage\*): 钓鱼按钮，点击后触发钓鱼过程。

m\_isFishingInProgress (bool): 钓鱼是否正在进行的标志。

m\_fishingProgress (float): 钓鱼过程的当前进度，值从 0 到 100。

3. 构造函数与析构函数

构造函数 FishingLayer()：初始化成员变量，设置钓鱼状态为未开始（m\_isFishingInProgress = false）和进度条为 0。

析构函数 ~FishingLayer()：无特殊逻辑，销毁时自动清理资源。

4. 成员函数

4.1 FishingLayer\* FishingLayer::createLayer()

创建并返回一个 FishingLayer 实例。成功初始化后，返回该实例并设置为自动释放，否则返回 nullptr。

4.2 bool FishingLayer::init()

初始化钓鱼层，包括：

创建进度条 (ProgressTimer) 并设置其样式为水平条。

创建钓鱼按钮（MenuItemImage），并设置点击事件回调 onFishingButtonClicked。

将按钮添加到菜单中，并将其添加到当前层（this）中。

4.3 void FishingLayer::onFishingButtonClicked(Ref\* sender)

点击钓鱼按钮时触发该函数：

如果钓鱼未进行中（m\_isFishingInProgress == false），则调用 startFishing() 开始钓鱼过程。

4.4 void FishingLayer::startFishing()

启动钓鱼过程，设置初始进度为 0，并通过定时器每 0.1 秒调用一次 updateProgressBar 来更新进度条。

4.5 void FishingLayer::updateProgressBar(float dt)

定时器调用的函数，用于更新钓鱼进度条：

每次增加 2%，更新进度条的百分比。

当进度达到 100% 时，停止定时器（unschedule），并调用 onFishCaught() 表示钓鱼成功。

4.6 void FishingLayer::onFishCaught()

钓鱼成功时调用该函数：

停止钓鱼（m\_isFishingInProgress = false）。

创建一个钓到的鱼（示例为“sunfish”）。

将钓到的鱼添加到背包中（pack1->itemAdd(item0, 1)）。

删除进度条和钓鱼按钮，清理界面。

4.7 void FishingLayer::onFishMissed()

钓鱼失败时调用该函数：

停止钓鱼（m\_isFishingInProgress = false）。

删除进度条和钓鱼按钮，清理界面。

5. 外部依赖与交互

backPack: 该类与外部背包管理类（backPack）有关，背包变量 pack1 用来存储钓到的物品。

Fish: Fish 类被用来创建鱼类物品，这里假设 Fish 类有一个 create() 方法来生成鱼对象。

ProgressTimer 和 MenuItemImage: 这些是 Cocos2d-x 引擎的 UI 元素，分别用于显示进度条和按钮。

6. 钓鱼流程

当用户点击钓鱼按钮时，程序检查是否有钓鱼正在进行。如果没有，开始钓鱼，进度从 0 开始。

每 0.1 秒，更新进度条，直到进度达到 100%，表示钓鱼完成。

若钓鱼成功，生成鱼类对象并添加到背包，同时清理界面上的钓鱼进度条和按钮。

若钓鱼失败，则清理界面上的进度条和按钮。

7. 总结

FishingLayer 类实现了一个简单的钓鱼小游戏功能，涉及进度条的更新、钓鱼过程的管理以及钓鱼成功或失败的处理。其设计简洁，适合嵌入到较大项目中实现钓鱼玩法。

二、crop

1. 类概述

Crop 类表示一个游戏中的作物。它管理作物的状态（如种子、生长中、可收获），根据作物的生长时间更新状态，并且处理与作物相关的精灵（图像）更新。此外，Crop 类提供了触摸检测功能，用于判断玩家是否与作物发生交互。

2. 成员变量

name (std::string): 作物的名称，如“cauliflower”或“parsnip”。

growthTime (float): 作物从生长到成熟所需的时间。

currentTime (float): 当前作物生长的时间（单位：秒）。

state (State): 当前作物的状态，表示作物是否是种子、生长中，还是可以收获。

sprite (Sprite\*): 作物对应的精灵（图像），展示作物的当前状态。

3. 状态枚举（State）

State 枚举表示作物的状态：

SEED: 作物处于种子状态，尚未开始生长。

GROWING: 作物正在生长中，尚未成熟。

HARVESTABLE: 作物已经成熟，可以收获。

4. 构造函数与析构函数

4.1 Crop::Crop(const std::string& cropName, float growthTime, const std::string& spriteFile)

构造函数，初始化作物的名称、成长时间和初始状态为 SEED，并根据传入的文件名创建对应的精灵对象。若精灵创建成功，加入当前节点作为子节点。

参数：

cropName: 作物的名称（例如，“cauliflower”）。

growthTime: 作物需要的成长时间。

spriteFile: 作物的初始图像文件名。

4.2 ~Crop()

析构函数，C++ 默认提供，未在代码中显式实现。

5. 成员函数

5.1 Crop\* Crop::create(const std::string& cropName, float growthTime, const std::string& spriteFile)

创建并返回一个新的 Crop 对象。该函数使用 new 分配内存并调用构造函数进行初始化。如果创建和初始化成功，将对象自动加入释放池；如果失败，释放资源并返回 nullptr。

参数：

cropName: 作物的名称。

growthTime: 作物的成长时间。

spriteFile: 初始精灵文件名。

返回值：返回创建的 Crop 对象，若创建失败返回 nullptr。

5.2 void Crop::update(float deltaTime)

更新作物的状态。该函数每秒调用一次，更新作物的生长时间，并根据当前状态执行以下操作：

SEED: 如果当前时间达到 60 秒，改变作物状态为 GROWING，并更新作物的纹理。

GROWING: 如果生长时间已达到设定的 growthTime，则将状态更新为 HARVESTABLE，并更新纹理。

参数：

deltaTime: 每帧的时间增量，用于更新当前时间。

5.3 Crop::State Crop::getState() const

返回当前作物的状态（State 枚举）。

返回值：作物的状态（State::SEED、State::GROWING 或 State::HARVESTABLE）。

5.4 const std::string& Crop::getName() const

返回作物的名称。

返回值：作物的名称（例如，“cauliflower”）。

5.5 Sprite\* Crop::getSprite() const

返回作物的精灵对象。

返回值：作物的精灵对象指针。

5.6 void Crop::changeSpriteTexture(const std::string& newSpriteFile)

改变作物精灵的纹理文件。根据不同的作物状态（例如，从种子到生长中的阶段），更新精灵的显示图片。

参数：

newSpriteFile: 新的纹理文件名。

注意：

如果加载新的纹理失败，CCLOG 会输出加载失败的错误信息。

5.7 bool Crop::isTouched(const cocos2d::Vec2& touchPosition) const

检测玩家是否点击了作物的区域。通过获取精灵的边界框（boundingBox），判断触摸点是否在精灵区域内。

参数：

touchPosition: 玩家触摸的位置（屏幕坐标）。

返回值：

如果触摸位置在作物的精灵范围内，返回 true，否则返回 false。

6. 作物生长逻辑

种子状态 (State::SEED): 作物刚开始种植，每秒增加 currentTime。

如果 currentTime 达到 60 秒，切换到生长中状态。

同时根据作物的名称，切换相应的纹理（例如，花椰菜变为“cauliflower\_growing.png”）。

生长中状态 (State::GROWING): 作物正在生长，currentTime 继续增加。

如果 currentTime 达到设定的 growthTime，切换到可收获状态。

根据作物名称，更新对应的成熟图像。

可收获状态 (State::HARVESTABLE): 作物已经成熟，可以收获，纹理变为“可收获”的状态。

7. 总结

Crop 类是一个管理作物生命周期（从种子到成熟收获）的类，功能包括：

根据生长时间自动更新作物的状态。

在不同的状态下更新作物的图像。

检测玩家是否点击作物。

三、PlantingSystem

1. 类概述

PlantingSystem 类负责管理作物的种植、更新和收获功能。它处理作物的生命周期，提供种植新作物的功能，并且通过触摸事件管理收获过程。作物的状态（如生长、成熟、收获）会实时更新。玩家可以点击作物来收获它们，并将收获的物品添加到背包中。

该类通过与 Crop 类的交互来实现作物的管理，同时通过触摸事件检测玩家的交互。

2. 成员变量

crops (std::vector<Crop\*>): 存储场景中所有作物的容器。

3. 成员函数

3.1 bool PlantingSystem::init()

初始化函数，完成以下操作：

调用 Node::init() 来确保该节点能够正常工作。

设定每帧调用 update() 函数来更新作物的状态。

创建并注册一个触摸事件监听器，用于处理玩家的点击操作。当玩家触摸屏幕时，尝试收获触摸位置上的作物。

返回值：

如果初始化成功，返回 true。

否则返回 false。

3.2 bool PlantingSystem::plantCrop(const std::string& cropName, float growthTime, const std::string& spriteFile, const cocos2d::Vec2& position)

种植新作物，完成以下操作：

使用 Crop::create() 创建作物对象。

设置作物的位置并将其添加到场景中。

将作物添加到 crops 容器中以进行管理。

参数：

cropName: 作物名称（例如，“cauliflower”）。

growthTime: 作物生长所需的时间。

spriteFile: 作物的精灵文件名。

position: 作物在场景中的位置。

返回值：

如果种植成功，返回 true。

否则返回 false。

3.3 void PlantingSystem::update(float deltaTime)

更新所有作物的状态。该函数每帧调用，迭代 crops 容器中的所有作物，并调用每个作物的 update() 方法更新其生长状态。

参数：

deltaTime: 每帧的时间增量，用于更新作物的生长时间。

3.4 void PlantingSystem::harvestCrop(cocos2d::Vec2 touchPosition)

收获作物的函数。遍历所有作物，检查其是否处于 HARVESTABLE（可收获）状态，并检查玩家的触摸位置是否在作物的边界框内。如果满足条件，则收获作物并将其添加到背包中，随后从场景中移除作物。

参数：

touchPosition: 玩家触摸的位置，检查是否点击到某个作物。

处理流程：

遍历 crops 容器，检查每个作物的状态。

如果作物处于可收获状态且触摸位置在作物的范围内，进行收获。

收获后，从场景中移除作物，并从 crops 容器中删除该作物。

4. 触摸事件

onTouchBegan: 当玩家点击屏幕时触发，获取触摸位置并调用 harvestCrop() 尝试收获作物。如果收获成功，将会将该作物从场景中移除。

harvestCrop(): 遍历 crops 容器，检查每个作物是否可以被收获（即 HARVESTABLE 状态），如果玩家点击了该作物，则从背包中添加相应物品并删除作物。

5. 作物收获与背包

在 harvestCrop() 中：

判断作物的名称（如“cauliflower”或“parsnip”），并根据作物名称创建相应的物品（如 Fruit 对象）。

将收获的物品添加到背包（pack1->itemAdd()）。

作物被收获后，从场景中移除并从 crops 容器中删除。

6. 收获过程的核心逻辑

在触摸事件中，根据触摸位置检查玩家是否点击了某个作物。

如果该作物可以收获（HARVESTABLE 状态），执行以下操作：

根据作物的名称创建相应的物品（如“cauliflower”对应 Fruit::create("cauliflower")）。

将该物品添加到背包中。

从场景中移除作物，删除 crops 容器中的该作物。

7. 总结

PlantingSystem 类提供了一个完整的作物管理和收获系统。它能够：

管理多个作物的种植和生长。

每帧更新作物的生长状态。

监听玩家的触摸操作，允许玩家点击并收获成熟的作物。

将收获的作物添加到背包中，并从场景中移除。

四、PlantingSystem

1. 类概述

PlantingSystem 类负责管理作物的种植、更新和收获功能。它处理作物的生命周期，提供种植新作物的功能，并且通过触摸事件管理收获过程。作物的状态（如生长、成熟、收获）会实时更新。玩家可以点击作物来收获它们，并将收获的物品添加到背包中。

该类通过与 Crop 类的交互来实现作物的管理，同时通过触摸事件检测玩家的交互。

2. 成员变量

crops (std::vector<Crop\*>): 存储场景中所有作物的容器。

3. 成员函数

3.1 bool PlantingSystem::init()

初始化函数，完成以下操作：

调用 Node::init() 来确保该节点能够正常工作。

设定每帧调用 update() 函数来更新作物的状态。

创建并注册一个触摸事件监听器，用于处理玩家的点击操作。当玩家触摸屏幕时，尝试收获触摸位置上的作物。

返回值：

如果初始化成功，返回 true。

否则返回 false。

3.2 bool PlantingSystem::plantCrop(const std::string& cropName, float growthTime, const std::string& spriteFile, const cocos2d::Vec2& position)

种植新作物，完成以下操作：

使用 Crop::create() 创建作物对象。

设置作物的位置并将其添加到场景中。

将作物添加到 crops 容器中以进行管理。

参数：

cropName: 作物名称（例如，“cauliflower”）。

growthTime: 作物生长所需的时间。

spriteFile: 作物的精灵文件名。

position: 作物在场景中的位置。

返回值：

如果种植成功，返回 true。

否则返回 false。

3.3 void PlantingSystem::update(float deltaTime)

更新所有作物的状态。该函数每帧调用，迭代 crops 容器中的所有作物，并调用每个作物的 update() 方法更新其生长状态。

参数：

deltaTime: 每帧的时间增量，用于更新作物的生长时间。

3.4 void PlantingSystem::harvestCrop(cocos2d::Vec2 touchPosition)

收获作物的函数。遍历所有作物，检查其是否处于 HARVESTABLE（可收获）状态，并检查玩家的触摸位置是否在作物的边界框内。如果满足条件，则收获作物并将其添加到背包中，随后从场景中移除作物。

参数：

touchPosition: 玩家触摸的位置，检查是否点击到某个作物。

处理流程：

遍历 crops 容器，检查每个作物的状态。

如果作物处于可收获状态且触摸位置在作物的范围内，进行收获。

收获后，从场景中移除作物，并从 crops 容器中删除该作物。

4. 触摸事件

onTouchBegan: 当玩家点击屏幕时触发，获取触摸位置并调用 harvestCrop() 尝试收获作物。如果收获成功，将会将该作物从场景中移除。

harvestCrop(): 遍历 crops 容器，检查每个作物是否可以被收获（即 HARVESTABLE 状态），如果玩家点击了该作物，则从背包中添加相应物品并删除作物。

5. 作物收获与背包

在 harvestCrop() 中：

判断作物的名称（如“cauliflower”或“parsnip”），并根据作物名称创建相应的物品（如 Fruit 对象）。

将收获的物品添加到背包（pack1->itemAdd()）。

作物被收获后，从场景中移除并从 crops 容器中删除。

6. 收获过程的核心逻辑

在触摸事件中，根据触摸位置检查玩家是否点击了某个作物。

如果该作物可以收获（HARVESTABLE 状态），执行以下操作：

根据作物的名称创建相应的物品（如“cauliflower”对应 Fruit::create("cauliflower")）。

将该物品添加到背包中。

从场景中移除作物，删除 crops 容器中的该作物。

7. 总结

PlantingSystem 类提供了一个完整的作物管理和收获系统。它能够：

管理多个作物的种植和生长。

每帧更新作物的生长状态。

监听玩家的触摸操作，允许玩家点击并收获成熟的作物。

将收获的作物添加到背包中，并从场景中移除。

此类适用于农业类游戏或模拟类游戏，能够动态地模拟作物的生长、收获以及资源管理。通过触摸事件与作物进行交互，可以实现游戏内的收获系统和背包管理。

五、Timer

1. 类概述

Timer1 类是一个用于模拟时间流逝和季节更替的计时器。它追踪经过的天数，并在每30天后自动更换季节。该类使用 std::chrono 库来获取和计算经过的时间，并根据经过的天数进行季节切换。

2. 成员变量

days\_elapsed (int): 记录从计时器开始以来经过的天数。初始值为 0。

season (std::string): 当前的季节。初始值为 "Spring"（春季）。

start\_time (std::chrono::steady\_clock::time\_point): 记录计时器开始时间，使用 steady\_clock 来保持精确的时间测量，避免由于系统时间变动而造成的误差。

3. 成员函数

3.1 Timer1::Timer1()

构造函数，用于初始化计时器。它将 days\_elapsed 设置为 0，season 设置为 "Spring"，并且记录当前时间为计时器的开始时间。

初始化值：

days\_elapsed = 0

season = "Spring"

start\_time = 当前时间（std::chrono::steady\_clock::now()）

3.2 int Timer1::getDaysElapsed() const

返回自计时器开始以来经过的天数。

返回值：返回 days\_elapsed，表示已经过去的天数。

3.3 std::string Timer1::getSeason() const

返回当前的季节。

返回值：返回季节的字符串值（例如，"Spring"、"Summer"、"Fall"、"Winter"）。

3.4 void Timer1::update()

更新计时器，计算已过天数并根据规则检查季节是否更替。每经过30天，季节会发生变化。

处理逻辑：

获取当前的时间（now），并计算从 start\_time 开始的经过时间。

通过 elapsed\_seconds.count() 获得自计时器开始以来经过的总秒数，将秒数转换为天数（1天 = 1440秒）。

如果经过的天数发生变化（new\_days\_elapsed > days\_elapsed），更新 days\_elapsed。

每经过30天，检查并更换季节：

从 "Spring"（春季）更换到 "Summer"（夏季）。

从 "Summer"（夏季）更换到 "Fall"（秋季）。

从 "Fall"（秋季）更换到 "Winter"（冬季）。

从 "Winter"（冬季）重新回到 "Spring"（春季）。

更换季节后，重新设置 start\_time 为当前时间，开始新的计时周期。

注意：

每30天更换季节，季节的循环顺序为：春季 → 夏季 → 秋季 → 冬季 → 春季。

4. 时间计算与季节更替

时间计算：std::chrono::steady\_clock::now() 被用于获取当前时间，与 start\_time 进行比较以计算经过的秒数。将秒数除以 1440（即每秒 60 秒 × 60 分钟 × 24 小时），从而得到过去的天数。

季节更替：

每经过30天，update() 方法检查并切换季节。季节的顺序是固定的：Spring -> Summer -> Fall -> Winter -> Spring。

一旦季节更替完成，重新记录计时器的开始时间 start\_time，以便下一个季节循环。

5. 总结

Timer1 类提供了一个简单的时间流逝管理和季节切换机制。它根据经过的天数更新当前的季节，并在每30天后自动进行季节更替。

六、WheelGame

1. 类概述

WheelGame 类实现了一个转盘游戏的逻辑。玩家点击旋转按钮时，转盘开始旋转，并在减速后停止。游戏的目标是根据转盘最终停留的位置决定玩家的奖励或惩罚（例如增加或减少一定的金额）。转盘旋转的速度逐渐减缓，最终停在某个角度位置，判断玩家是否成功。

2. 成员变量

wheel (Sprite\*): 转盘的精灵对象，表示转盘的图像。

isSpinning (bool): 标志位，表示转盘是否正在旋转。

rotationSpeed (float): 转盘的旋转速度，用于控制旋转的快慢。

totalRotation (float): 转盘旋转的总角度，用于计算转盘的最终停止角度。

rest (float): 上一次转盘停止时的角度，记录下来用于本次旋转时的初始角度。全局变量。

randomEngine (std::mt19937): 随机数生成引擎，用于生成随机数。

distribution (std::uniform\_int\_distribution<>): 随机数分布，用于生成随机的整数（决定随机角度）。

spinButton (MenuItemImage\*): 旋转按钮，用于触发转盘的旋转。

3. 成员函数

3.1 WheelGame\* WheelGame::create()

静态工厂函数，用于创建 WheelGame 类的实例。

返回值：返回 WheelGame 实例指针。如果创建成功，则返回实例；否则返回 nullptr。

3.2 bool WheelGame::init()

初始化函数，设置游戏的初始状态，包括转盘精灵的创建、随机数生成器的初始化、旋转按钮的配置等。

返回值：如果初始化成功，返回 true；否则返回 false。

主要步骤：

获取当前屏幕的大小（Director::getInstance()->getVisibleSize()）。

创建转盘精灵，并将其定位在屏幕中央。

初始化随机数生成器，准备生成随机数。

创建旋转按钮，设置按钮的正常和点击状态图像。

将按钮添加到场景中。

3.3 void WheelGame::onSpinButtonClicked(Ref\* sender)

按钮点击事件处理函数，当玩家点击“旋转”按钮时调用。如果转盘没有正在旋转，调用 spinWheel() 开始旋转。

参数：sender 为事件发送者，一般为按钮本身。

3.4 void WheelGame::spinWheel()

启动转盘旋转的函数。设置转盘的初始旋转速度，并开始调度每帧更新转盘的旋转。

主要操作：

设置初始旋转速度：rotationSpeed = 50.0f + rest，即在每次旋转时，加入上一次旋转的角度。

更新 totalRotation 为上一次的角度 rest。

调度每帧更新 updateWheel() 函数，控制转盘旋转。

3.5 void WheelGame::updateWheel(float dt)

每帧调用的更新函数，负责更新转盘的旋转和处理转盘的减速逻辑。

参数：dt 为每帧的时间增量。

主要操作：

更新转盘的旋转角度：wheel->setRotation(wheel->getRotation() + rotationSpeed)。

更新 totalRotation，累加当前的旋转角度。

渐渐减小旋转速度：rotationSpeed \*= 0.95f（每帧减速5%）。

使用随机数生成器生成一个随机数并累加到 totalRotation，使旋转过程更具不可预测性。

当 rotationSpeed 小于某个阈值（0.1f）时，停止旋转：

使用 fmod(totalRotation, 360.0f) 获取最终的旋转角度（0到360度之间）。

判断转盘最终停留的位置：

如果最终角度在 90 到 270 度之间，判定为成功（moneyChange(100, 0)，增加100货币）。

否则判定为失败（moneyChange(100, 1)，扣除100货币）。

更新 rest 为本次旋转停止时的角度，用于下次旋转的初始角度。

4. 转盘旋转与游戏规则

旋转速度：转盘的初始旋转速度为 50.0f + rest，即每次旋转时会加入上次旋转的角度。旋转速度每帧都在逐渐减小，直到转盘停止。

随机数的使用：通过随机数生成器，在每一帧的旋转中增加一个小的随机值，使转盘的旋转过程更具不确定性。

胜负判定：转盘停止时，根据最终角度判断是否成功。成功的标准是转盘停留在 90 到 270 度之间。

如果在成功区间（90°-270°），调用 moneyChange(100, 0)，增加100货币。

如果不在成功区间，则调用 moneyChange(100, 1)，扣除100货币。

5. 总结

WheelGame 类实现了一个基于转盘的游戏逻辑。玩家点击旋转按钮后，转盘开始旋转，旋转速度逐渐减慢，最终停留在某个角度。根据转盘的停止角度，决定玩家是否获得奖励或惩罚。游戏中的旋转速度和旋转过程充满随机性，增加了游戏的趣味性。