

星露谷物语 - 技术实现文档

文档信息

- 项目名称: 星露谷关守
- (StardewFarm)引擎: Cocos2d-x 3.17
- 编程语言: C++11/14
- 文档版本: v1.0
- 最后更新: 2025-12-27

目录

1. 架构概览
2. 核心系统实现
3. 游戏场景系统
4. 农耕系统
5. 矿洞探险系统
6. 战斗系统
7. 背包与物品系统
8. 存档系统
9. 技能系统
10. NPC与对话系统
11. 钓鱼系统
12. 天气系统
13. 时间系统

1. 架构概览

1.1 整体架构设计

游戏采用分层架构设计：

表现层 (Presentation)
GameScene, MenuScene, HouseScene
MineScene, BeachScene

↓ 调用

业务逻辑层 (Business Logic)
FarmManager, InventoryManager
SkillManager, SaveManager
MiningManager, TimeManager

↓ 操作

数据层 (Data)
SaveData, ItemSlot, FarmTile
SkillData, MonsterData

1.2 核心设计模式

1.2.1 单例模式 (Singleton Pattern)

用于全局唯一的管理器类：

实现位置: [Classes/SaveManager.h](#)

```
class SaveManager {
public:
    static SaveManager* getInstance() {
        if (!instance_) {
            instance_ = new SaveManager();
        }
        return instance_;
    }

    static void destroyInstance() {
        if (instance_) {
            delete instance_;
            instance_ = nullptr;
        }
    }
}

private:
    SaveManager() = default;
    static SaveManager* instance_;
};
```

应用的类:

- **SaveManager** - 存档管理
- **InventoryManager** - 背包管理
- **SkillManager** - 技能管理
- **TimeManager** - 时间管理

1.2.2 观察者模式 (Observer Pattern)

用于事件通知系统：

实现位置: [Classes/Observer.h](#) (新增模板类)

```
// 事件定义
struct HealthChangedEvent {
    int oldHealth;
    int newHealth;
};

// 被观察者
class Player : public Observable<HealthChangedEvent> {
public:
    void takeDamage(int damage) {
        int oldHp = hp_;
        hp_ -= damage;
        notifyObservers({oldHp, hp_}); // 通知所有观察者
    }
};

// 观察者
class HealthBar : public IObservable<HealthChangedEvent> {
public:
    void onNotify(const HealthChangedEvent& event) override {
        updateDisplay(event.newHealth);
    }
};
```

应用场景:

- 玩家血量变化 → 更新UI
- 物品添加 → 刷新背包显示
- 技能升级 → 显示升级特效

1.2.3 工厂模式 (Factory Pattern)

用于创建复杂对象：

实现位置: [Classes/Monster.cpp](#)

```
Monster* Monster::create() {
    Monster* ret = new (std::nothrow) Monster();
    if (ret && ret->init()) {
        ret->autorelease();
        return ret;
    }
    CC_SAFE_DELETE(ret);
    return nullptr;
}
```

特点:

- 使用 `std::nothrow` 避免异常
- 使用 `autorelease()` 自动内存管理
- 安全的空指针检查

1.3 类继承关系图

```
cocos2d::Scene
└── GameScene (主游戏场景)
└── MenuScene (主菜单场景)
└── HouseScene (房屋场景)
└── MineScene (矿洞场景)
└── BeachScene (海滩场景)

cocos2d::Sprite
└── Player (玩家角色)
└── Monster (怪物基类)
    └── Slime (史莱姆)
    └── Zombie (僵尸)
└── Npc (NPC角色)

cocos2d::Node
└── InventoryManager (背包管理)
└── FarmManager (农场管理)
└── SkillManager (技能管理)
└── DialogueBox (对话框)
└── InventoryUI (背包界面)
└── ElevatorUI (电梯界面)
└── StorageChest (储物箱)
    └── ShippingBin (运输箱)

cocos2d::Layer
└── MapLayer (地图层基类)
    └── MineLayer (矿洞地图层)
└── InventoryUI (背包界面)
└── FishingLayer (钓鱼层)
```

2. 核心系统实现

2.1 应用程序入口

实现位置: [Classes/AppDelegate.cpp](#)

2.1.1 初始化流程

```
bool AppDelegate::applicationDidFinishLaunching() {
    // 1. 创建Director单例
```

```
auto director = Director::getInstance();
auto glview = director->getOpenGLView();

// 2. 创建OpenGL视图
if(!glview) {
    glview = GLViewImpl::create("StardewFarm");
    director->setOpenGLView(glview);
}

// 3. 设置设计分辨率
glview->setDesignResolutionSize(1280, 720,
                                  ResolutionPolicy::SHOW_ALL);

// 4. 设置帧率
director->setAnimationInterval(1.0f / 60);

// 5. 创建并运行第一个场景
auto scene = MenuScene::createScene();
director->runWithScene(scene);

return true;
}
```

关键参数说明:

- 设计分辨率: 1280x720 (16:9比例)
- 适配策略: SHOW_ALL (保持宽高比，可能有黑边)
- 帧率: 60 FPS

2.1.2 生命周期管理

```
void AppDelegate::applicationDidEnterBackground() {
    Director::getInstance()->stopAnimation();
    // 暂停背景音乐
}

void AppDelegate::applicationWillEnterForeground() {
    Director::getInstance()->startAnimation();
    // 恢复背景音乐
}
```

2.2 场景管理系统

2.2.1 场景切换机制

实现位置: [Classes/GameScene.cpp](#)

```
void GameScene::returnToMenu() {
    // 1. 保存游戏数据
    SaveManager::getInstance()->autoSave();

    // 2. 清理当前场景资源
    this->cleanup();

    // 3. 创建新场景
    auto menuScene = MenuScene::createScene();

    // 4. 使用淡入淡出过渡
    auto transition = TransitionFade::create(1.0f, menuScene);

    // 5. 切换场景
    Director::getInstance()->replaceScene(transition);
}
```

场景切换类型:

- `replaceScene()` - 替换场景 (释放旧场景)
- `pushScene()` - 推入场景 (保留旧场景)
- `popScene()` - 弹出场景 (返回上一场景)

过渡动画:

- `TransitionFade` - 淡入淡出
- `TransitionSlideInR` - 从右侧滑入
- `TransitionCrossFade` - 交叉淡化

2.2.2 场景初始化标准流程

```
bool GameScene::init() {
    if (!Scene::init()) {
        return false;
    }

    // 1. 初始化成员变量
    initVariables();

    // 2. 加载地图
    initMap();

    // 3. 创建玩家
    initPlayer();

    // 4. 设置摄像机
    initCamera();

    // 5. 初始化UI
    initUI();
}
```

```
// 6. 注册输入监听
initInputListeners();

// 7. 启动更新循环
this->scheduleUpdate();

return true;
}
```

2.3 输入系统

2.3.1 键盘事件处理

实现位置: [Classes/GameScene.cpp](#)

```
void GameScene::initInputListeners() {
    auto keyListener = EventListenerKeyboard::create();

    // 按键按下回调
    keyListener->onKeyPressed = [this](EventKeyboard::KeyCode keyCode, Event*
event) {
        switch(keyCode) {
            case EventKeyboard::KeyCode::KEY_W:
                player_->setMovingUp(true);
                break;
            case EventKeyboard::KeyCode::KEY_A:
                player_->setMovingLeft(true);
                break;
            case EventKeyboard::KeyCode::KEY_S:
                player_->setMovingDown(true);
                break;
            case EventKeyboard::KeyCode::KEY_D:
                player_->setMovingRight(true);
                break;
            case EventKeyboard::KeyCode::KEY_J:
                handleActionKey();
                break;
            case EventKeyboard::KeyCode::KEY_B:
                toggleInventory();
                break;
            // ... 其他按键
        }
    };

    // 按键释放回调
    keyListener->onKeyReleased = [this](EventKeyboard::KeyCode keyCode, Event*
event) {
        // 停止移动
        switch(keyCode) {
            case EventKeyboard::KeyCode::KEY_W:
```

```

        player_->setMovingUp(false);
        break;
    // ... 其他按键
}
};

// 注册监听器
_eventDispatcher->addEventWithSceneGraphPriority(keyListener, this);
}

```

按键映射表:

按键	KeyCode	功能	实现方法
W/A/S/D	KEY_W/A/S/D	移动	Player::setMoving*()
J	KEY_J	通用动作	handleActionKey()
K	KEY_K	特殊交互	handleSpecialKey()
B	KEY_B	背包	toggleInventory()
E	KEY_E	技能树	toggleSkillTree()
M	KEY_M	电梯	openElevator()
SPACE	KEY_SPACE	开箱/对话	handleSpaceKey()
ESC	KEY_ESCAPE	菜单	returnToMenu()
1-0	KEY_1~0	切换物品	selectToolbarItem()

2.3.2 鼠标事件处理

实现位置: [Classes/GameScene.cpp](#)

```

void GameScene::initMouseListener() {
    auto mouseListener = EventListenerMouse::create();

    mouseListener->onMouseDown = [this](Event* event) {
        EventMouse* mouseEvent = static_cast<EventMouse*>(event);
        Vec2 clickPos = Vec2(mouseEvent->getCursorX(), mouseEvent->getCursorY());

        // 转换为世界坐标
        Vec2 worldPos = this->convertToWorldSpace(clickPos);

        // 检查是否点击NPC
        for (auto* npc : npcs_) {
            if (npc->getBoundingBox().containsPoint(worldPos)) {
                startNpcInteraction(npc);
                return;
            }
        }
    };
}

```

```
// 钓鱼逻辑
if (currentToolbarItem_ == ItemType::ITEM_FishingRod) {
    handleFishing(worldPos);
}
};

_eventDispatcher->addEventListenWithSceneGraphPriority(mouseListener, this);
```

坐标系统转换:

屏幕坐标 (Screen) → 视图坐标 (View) → 世界坐标 (World) → 节点坐标 (Node)

2.4 游戏主循环

实现位置: [Classes/GameScene.cpp](#)

```
void GameScene::update(float delta) {
    // 1. 更新摄像机 (跟随玩家)
    updateCamera();

    // 2. 更新玩家状态
    if (player_) {
        player_->update(delta);
    }

    // 3. 更新怪物AI
    for (auto* monster : monsters_) {
        if (monster) {
            monster->update(delta);
        }
    }

    // 4. 更新农场状态
    if (farmManager_) {
        farmManager_->update(delta);
    }

    // 5. 更新时间系统
    TimeManager::getInstance()->update(delta);

    // 6. 碰撞检测
    checkCollisions();

    // 7. 清理死亡对象
    cleanupDeadEntities();
}
```

更新频率: 60次/秒 (delta ≈ 0.0167秒)

3. 游戏场景系统

3.1 GameScene - 主游戏场景

文件位置: [Classes/GameScene.h](#), [Classes/GameScene.cpp](#)

3.1.1 场景组成

```
class GameScene : public cocos2d::Scene {
private:
    // 核心组件
    Player* player_;                                // 玩家角色
    MapLayer* mapLayer_;                             // 地图层
    FarmManager* farmManager_;                      // 农场管理器
    InventoryManager* inventoryMgr_;                // 背包管理器

    // UI组件
    InventoryUI* inventoryUI_;                     // 背包界面
    SkillTreeUI* skillTreeUI_;                      // 技能树界面
    MarketUI* marketUI_;                           // 市场界面
    EnergyBar* energyBar_;                          // 能量条
    std::vector<Sprite*> toolbarSlots_;           // 工具栏图标

    // 场景对象
    std::vector<Npc*> npcs_;                      // NPC列表
    std::vector<StorageChest*> chests_;             // 储物箱列表
    ShippingBin* shippingBin_;                      // 运输箱

    // 摄像机
    Camera* followCamera_;                         // 跟随摄像机
};

};
```

3.1.2 地图加载机制

实现位置: [Classes/GameScene.cpp](#)

```
void GameScene::initMap() {
    // 1. 创建地图层
    mapLayer_ = MapLayer::create();
    if (!mapLayer_) {
        CCLOG("Error: Failed to create MapLayer");
        return;
    }

    // 2. 加载TMX地图文件
    bool success = mapLayer_->loadTMXMap("map/farm.tmx");
```

```

if (!success) {
    CCLOG("Error: Failed to load farm.tmx");
    return;
}

// 3. 添加到场景 ( Z-order = -1 · 在最底层 )
this->addChild(mapLayer_, -1);

// 4. 获取地图尺寸
Size mapSize = mapLayer_->getMapSize();
mapSizeInPixels_ = Size(
    mapSize.width * mapLayer_->getTileSize().width,
    mapSize.height * mapLayer_->getTileSize().height
);
}

```

TMX地图结构 (farm.tmx):

```

<map version="1.0" orientation="orthogonal" width="50" height="50" tilewidth="32"
tileheight="32">
    <tileset firstgid="1" name="tileset2" tilewidth="32" tileheight="32">
        <image source="tileset2.png" width="512" height="512"/>
    </tileset>

    <!-- 地面层 -->
    <layer id="4" name="Ground" width="50" height="50">
        <data encoding="csv">
            1,2,3,4,5,...
        </data>
    </layer>

    <!-- 装饰层 -->
    <layer id="5" name="Tree" width="50" height="50">
        <data encoding="csv">
            0,0,100,0,0,...
        </data>
    </layer>

    <!-- 碰撞层 ( 不可见 ) -->
    <layer id="2" name="Collision" width="50" height="50" opacity="0">
        <data encoding="csv">
            0,0,1,1,0,...
        </data>
    </layer>
</map>

```

地图层级 (Z-order):

100: UI层 (背包、对话框)
10: 玩家和NPC
5: 装饰物 (树木)
0: 实体对象 (箱子)
-1: 地图底层

3.1.3 玩家初始化

实现位置: [Classes/GameScene.cpp](#)

```
void GameScene::initPlayer() {
    // 1. 创建玩家精灵
    player_ = Player::create("characters/player.png");
    if (!player_) {
        CCLOG("Error: Failed to create player");
        return;
    }

    // 2. 设置初始位置
    Vec2 spawnPos = Vec2(400, 300);

    // 从存档加载位置
    if (SaveManager::getInstance()->hasSaveFile()) {
        SaveData data;
        if (SaveManager::getInstance()->loadGame(data)) {
            spawnPos = data.playerPosition;
        }
    }

    player_->setPosition(spawnPos);

    // 3. 设置Z-order ( 高于地图 · 低于UI )
    this->addChild(player_, 10);

    // 4. 绑定地图层 ( 用于碰撞检测 )
    player_->setMapLayer(mapLayer_);

    // 5. 绑定农场管理器 ( 用于农耕操作 )
    player_->setFarmManager(farmManager_);

    // 6. 绑定背包管理器 ( 用于物品操作 )
    player_->setInventoryManager(inventoryMgr_);
}
```

3.1.4 摄像机跟随

实现位置: [Classes/GameScene.cpp](#)

```

void GameScene::updateCamera() {
    if (!player_ || !followCamera_) return;

    // 1. 获取玩家位置
    Vec2 playerPos = player_->getPosition();

    // 2. 计算目标摄像机位置
    Vec3 targetPos = Vec3(playerPos.x, playerPos.y, 0);

    // 3. 限制摄像机边界 (防止看到地图外)
    Size visibleSize = Director::getInstance()->getVisibleSize();
    float halfWidth = visibleSize.width / 2;
    float halfHeight = visibleSize.height / 2;

    // 左边界
    if (targetPos.x < halfWidth) {
        targetPos.x = halfWidth;
    }
    // 右边界
    if (targetPos.x > mapSizeInPixels_.width - halfWidth) {
        targetPos.x = mapSizeInPixels_.width - halfWidth;
    }
    // 下边界
    if (targetPos.y < halfHeight) {
        targetPos.y = halfHeight;
    }
    // 上边界
    if (targetPos.y > mapSizeInPixels_.height - halfHeight) {
        targetPos.y = mapSizeInPixels_.height - halfHeight;
    }

    // 4. 平滑移动摄像机 (插值)
    Vec3 currentPos = followCamera_->getPosition3D();
    float smoothFactor = 0.1f; // 平滑系数
    Vec3 newPos = currentPos + (targetPos - currentPos) * smoothFactor;

    // 5. 设置摄像机位置
    followCamera_->setPosition3D(newPos);
}

```

摄像机类型:

- 正交摄像机 (OrthographicCamera) - 用于2D游戏
- 透视摄像机 (PerspectiveCamera) - 用于3D游戏

平滑跟随原理:

新位置 = 当前位置 + (目标位置 - 当前位置) × 平滑系数

- 平滑系数越大，跟随越快

- 平滑系数越小，跟随越平滑

3.2 MineScene - 矿洞场景

文件位置: [Classes/MineScene.h](#), [Classes/MineScene.cpp](#)

3.2.1 楼层管理

```
class MineScene : public cocos2d::Scene {
private:
    int currentFloor_;                                // 当前楼层
    MineLayer* mineLayer_;                            // 矿洞地图层
    Player* player_;                                 // 玩家
    std::vector<Monster*> monsters_;                // 怪物列表
    std::vector<Sprite*> ores_;                     // 矿石列表
    TreasureChest* treasureChest_;                  // 宝箱

public:
    void setFloor(int floor) {
        currentFloor_ = floor;
        regenerateFloor();
    }
};
```

3.2.2 楼层生成算法

实现位置: [Classes/MineScene.cpp](#)

```
void MineScene::regenerateFloor() {
    // 1. 清理旧对象
    cleanupFloor();

    // 2. 重新加载地图
    std::string mapFile = "map/mine" + std::to_string(currentFloor_) + ".tmx";
    mineLayer_->loadTMXMap(mapFile);

    // 3. 生成怪物 ( 数量随楼层增加 )
    int monsterCount = 3 + currentFloor_ / 2;
    for (int i = 0; i < monsterCount; ++i) {
        spawnMonster();
    }

    // 4. 生成矿石 ( 随机位置 )
    int oreCount = 5 + cocos2d::random(0, 5);
    for (int i = 0; i < oreCount; ++i) {
        spawnOre();
    }

    // 5. 生成宝箱 ( 概率性 )
    if (cocos2d::random(0.0f, 1.0f) < 0.3f) {
```

```

        spawnTreasureChest();
    }

    // 6. 生成许愿池 (仅在5的倍数层)
    if (currentFloor_ % 5 == 0 && cocos2d::random(0.0f, 1.0f) < 0.5f) {
        spawnWishingWell();
    }
}

```

怪物生成逻辑:

```

void MineScene::spawnMonster() {
    // 1. 随机选择怪物类型
    bool isSlime = (cocos2d::random(0, 1) == 0);

    Monster* monster = nullptr;
    if (isSlime) {
        monster = Slime::create();
    } else {
        monster = Zombie::create();
    }

    // 2. 随机位置 (避开碰撞层)
    Vec2 spawnPos;
    do {
        spawnPos = Vec2(
            cocos2d::random(100.0f, 700.0f),
            cocos2d::random(100.0f, 500.0f)
        );
    } while (mineLayer_->isCollisionAt(spawnPos));

    monster->setPosition(spawnPos);

    // 3. 根据楼层调整属性
    float floorMultiplier = 1.0f + (currentFloor_ - 1) * 0.2f;
    monster->setHealthMultiplier(floorMultiplier);
    monster->setDamageMultiplier(floorMultiplier);

    // 4. 添加到场景
    this->addChild(monster, 5);
    monsters_.push_back(monster);
}

```

难度曲线:

楼层1: 血量×1.0, 攻击×1.0
 楼层2: 血量×1.2, 攻击×1.2
 楼层3: 血量×1.4, 攻击×1.4

楼层4：血量×1.6， 攻击×1.6
 楼层5：血量×1.8， 攻击×1.8

4. 农耕系统

文件位置: [Classes/FarmManager.h](#), [Classes/FarmManager.cpp](#)

4.1 农田数据结构

```
// 农田图块状态
struct FarmTile {
    bool isTilled; // 是否已耕种
    bool isWatered; // 是否已浇水
    bool hasPlant; // 是否有作物
    int cropId; // 作物ID
    int stage; // 作物生长阶段
    int progressDays; // 生长天数
    cocos2d::Sprite* sprite; // 作物精灵
};

class FarmManager : public cocos2d::Node {
private:
    std::vector<FarmTile> tiles_; // 农田图块数组
    std::unordered_map<int, CropDef> crops_; // 作物定义表
    Size mapSizeTiles_; // 地图尺寸(图块数)
    Size tileSize_; // 图块尺寸(像素)
};
```

4.2 作物定义系统

实现位置: [Classes/FarmManager.cpp](#)

```
// 作物定义结构
struct CropDef {
    std::string name; // 作物名称
    int growthStages; // 生长阶段数
    std::vector<int> stageDurations; // 每阶段天数
    int basePrice; // 基础价格
    std::vector<std::string> sprites; // 各阶段精灵图
};

void FarmManager::initCropDefinitions() {
    // 萝卜 (Radish)
    crops_[CROP_RADISH] = {
        "Radish", // 名称
        3, // 3个生长阶段
        {2, 2, 2}, // 每阶段2天·共6天成熟
        50, // 基础售价50金币
    };
}
```

```

    {
        "crops/radish_stage1.png",
        "crops/radish_stage2.png",
        "crops/radish_stage3.png"
    }
};

// 土豆 (Potato)
crops_[CROP_POTATO] = {
    "Potato",
    4,                                     // 4个生长阶段
    {2, 2, 2, 2},                           // 每阶段2天 · 共8天成熟
    80,
{
    "crops/potato_stage1.png",
    "crops/potato_stage2.png",
    "crops/potato_stage3.png",
    "crops/potato_stage4.png"
}
};

// 玉米 (Corn)
crops_[CROP_CORN] = {
    "Corn",
    5,                                     // 5个生长阶段
    {2, 2, 2, 2, 2},                         // 每阶段2天 · 共10天成熟
    120,
{
    "crops/corn_stage1.png",
    "crops/corn_stage2.png",
    "crops/corn_stage3.png",
    "crops/corn_stage4.png",
    "crops/corn_stage5.png"
}
};
}
}

```

4.3 核心农耕操作

4.3.1 耕地 (Tilling)

实现位置: [Classes/FarmManager.cpp](#)

```

ActionResult FarmManager::tillTile(const Vec2& tileCoord) {
    // 1. 坐标转换：世界坐标 → 图块坐标
    int tileX = static_cast<int>(tileCoord.x / tileSize_.width);
    int tileY = static_cast<int>(tileCoord.y / tileSize_.height);

    // 2. 边界检查
    if (tileX < 0 || tileX >= mapSizeTiles_.width ||
        tileY < 0 || tileY >= mapSizeTiles_.height) {

```

```

        return {false, "Out of bounds"};
    }

    // 3. 获取图块索引
    int index = tileY * static_cast<int>(mapSizeTiles_.width) + tileX;

    // 4. 检查是否已耕种
    if (tiles_[index].isTilled) {
        return {false, "Already tilled"};
    }

    // 5. 检查是否有碰撞 (树木、建筑物等)
    if (mapLayer_->isCollisionAt(tileCoord)) {
        return {false, "Cannot till here"};
    }

    // 6. 更新图块状态
    tiles_[index].isTilled = true;

    // 7. 更新视觉表现 (深色土地)
    updateTileVisuals(index);

    return {true, "Tilled successfully"};
}

```

图块索引计算:

索引 = Y坐标 × 地图宽度 + X坐标

示例：50×50地图，坐标(10, 5)

索引 = 5 × 50 + 10 = 260

4.3.2 种植 (Planting)

实现位置: [Classes/FarmManager.cpp](#)

```

ActionResult FarmManager::plantSeed(const Vec2& tileCoord, int cropId) {
    // 1. 坐标转换
    int index = getTileIndex(tileCoord);

    // 2. 检查是否已耕种
    if (!tiles_[index].isTilled) {
        return {false, "Must till first"};
    }

    // 3. 检查是否已有作物
    if (tiles_[index].hasPlant) {
        return {false, "Already planted"};
    }
}

```

```

// 4. 检查作物定义是否存在
if (crops_.find(cropId) == crops_.end()) {
    return {false, "Invalid crop ID"};
}

// 5. 更新图块状态
tiles_[index].hasPlant = true;
tiles_[index].cropId = cropId;
tiles_[index].stage = 0;
tiles_[index].progressDays = 0;

// 6. 创建作物精灵
const CropDef& crop = crops_[cropId];
auto sprite = Sprite::create(crop.sprites[0]);
sprite->setPosition(tileCoord);
this->addChild(sprite, 1);
tiles_[index].sprite = sprite;

// 7. 从背包移除种子
InventoryManager::getInstance()->removeItem(
    static_cast<ItemType>(ITEM_SEED_BASE + cropId), 1
);

return {true, "Planted " + crop.name};
}

```

4.3.3 浇水 (Watering)

实现位置: [Classes/FarmManager.cpp](#)

```

ActionResult FarmManager::waterTile(const Vec2& tileCoord) {
    int index = getTileIndex(tileCoord);

    // 1. 检查是否有作物
    if (!tiles_[index].hasPlant) {
        return {false, "No plant to water"};
    }

    // 2. 检查是否已浇水
    if (tiles_[index].isWatered) {
        return {false, "Already watered today"};
    }

    // 3. 更新浇水状态
    tiles_[index].isWatered = true;

    // 4. 视觉反馈 (深色湿润土地)
    updateTileVisuals(index);

    // 5. 播放水滴音效
}

```

```
    AudioEngine::play2d("audio/water.mp3");

    return {true, "Watered"};
}
```

4.3.4 收获 (Harvesting)

实现位置: [Classes/FarmManager.cpp](#)

```
ActionResult FarmManager::harvestCrop(const Vec2& tileCoord) {
    int index = getTileIndex(tileCoord);

    // 1. 检查是否有作物
    if (!tiles_[index].hasPlant) {
        return {false, "No crop to harvest"};
    }

    // 2. 获取作物定义
    const CropDef& crop = crops_[tiles_[index].cropId];

    // 3. 检查是否成熟
    if (tiles_[index].stage < crop.growthStages - 1) {
        return {false, "Crop not ready yet"};
    }

    // 4. 计算收获数量 (基础1个 + 技能加成)
    int harvestCount = 1;
    int agricultureLevel = SkillManager::getInstance()-
>getLevel(SkillType::Agriculture);

    // 10%概率每技能等级额外收获1个
    if (cocos2d::random(0.0f, 1.0f) < agricultureLevel * 0.1f) {
        harvestCount++;
    }

    // 5. 添加到背包
    ItemType itemType = static_cast<ItemType>(ITEM_CROP_BASE +
tiles_[index].cropId);
    InventoryManager::getInstance()->addItem(itemType, harvestCount);

    // 6. 增加技能经验
    SkillManager::getInstance()->addExperience(SkillType::Agriculture, 10);

    // 7. 清理图块
    if (tiles_[index].sprite) {
        tiles_[index].sprite->removeFromParent();
        tiles_[index].sprite = nullptr;
    }

    tiles_[index].hasPlant = false;
    tiles_[index].cropId = 0;
```

```
    tiles_[index].stage = 0;
    tiles_[index].progressDays = 0;

    // 8. 播放收获音效
    AudioEngine::play2d("audio/harvest.mp3");

    return {true, "Harvested " + std::to_string(harvestCount) + " " + crop.name};
}
```

4.4 作物生长系统

4.4.1 每日更新逻辑

实现位置: [Classes/FarmManager.cpp](#)

```
void FarmManager::onNewDay() {
    for (size_t i = 0; i < tiles_.size(); ++i) {
        FarmTile& tile = tiles_[i];

        // 只处理有作物的图块
        if (!tile.hasPlant) continue;

        // 1. 检查浇水状态
        if (!tile.isWatered) {
            // 未浇水，作物不生长
            CCLOG("Tile %zu: Crop not watered, no growth", i);
            continue;
        }

        // 2. 增加生长天数
        tile.progressDays++;

        // 3. 获取作物定义
        const CropDef& crop = crops_[tile.cropId];

        // 4. 计算应该在的阶段
        int cumulativeDays = 0;
        int targetStage = 0;

        for (int stage = 0; stage < crop.growthStages; ++stage) {
            cumulativeDays += crop.stageDurations[stage];
            if (tile.progressDays < cumulativeDays) {
                targetStage = stage;
                break;
            }
            targetStage = stage + 1;
        }

        // 确保不超过最大阶段
        if (targetStage >= crop.growthStages) {
            targetStage = crop.growthStages - 1;
        }
    }
}
```

```

    }

    // 5. 更新阶段
    if (tile.stage != targetStage) {
        tile.stage = targetStage;

        // 6. 更新精灵
        if (tile.sprite && targetStage < crop.sprites.size()) {
            tile.sprite->setTexture(crop.sprites[targetStage]);
        }

        CCLOG("Tile %zu: Crop grew to stage %d", i, targetStage);
    }

    // 7. 重置浇水状态
    tile.isWatered = false;
}

}

```

生长阶段计算示例 (萝卜 - 每阶段2天):

```

天数0: 阶段0 (种子)
天数1: 阶段0
天数2: 阶段1 (小苗)
天数3: 阶段1
天数4: 阶段2 (成熟)
天数5: 阶段2

```

4.4.2 雨天自动浇水

实现位置: [Classes/FarmManager.cpp](#)

```

void FarmManager::handleRainDay() {
    int wateredCount = 0;

    for (FarmTile& tile : tiles_) {
        if (tile.hasPlant && !tile.isWatered) {
            tile.isWatered = true;
            wateredCount++;
        }
    }

    CCLOG("Rain watered %d tiles automatically", wateredCount);
}

```

5. 矿洞探险系统

5.1 矿洞层级管理

文件位置: [Classes/MiningManager.h](#), [Classes/MiningManager.cpp](#)

5.1.1 矿石生成系统

```
enum class OreType {
    COPPER,      // 铜矿 (1-5层)
    SILVER,       // 银矿 (3-8层)
    GOLD,         // 金矿 (5-10层)
    DIAMOND       // 钻石 (8+层)
};

struct OreSpawn {
    OreType type;
    Vec2 position;
    int health;           // 需要挖掘次数
    Sprite* sprite;
};
```

矿石分布算法:

```
void MiningManager::spawnOres(int floor) {
    // 1. 根据楼层确定矿石类型概率
    std::vector<std::pair<OreType, float>> probabilities;

    if (floor <= 5) {
        probabilities = {
            {OreType::COPPER, 0.7f},
            {OreType::SILVER, 0.3f}
        };
    } else if (floor <= 10) {
        probabilities = {
            {OreType::COPPER, 0.4f},
            {OreType::SILVER, 0.4f},
            {OreType::GOLD, 0.2f}
        };
    } else {
        probabilities = {
            {OreType::SILVER, 0.3f},
            {OreType::GOLD, 0.4f},
            {OreType::DIAMOND, 0.3f}
        };
    }

    // 2. 生成矿石
    int oreCount = 5 + cocos2d::random(0, 5);
    for (int i = 0; i < oreCount; ++i) {
        // 随机选择类型
        OreType type = selectOreByProbability(probabilities);
```

```
// 随机位置
Vec2 pos = getRandomFloorPosition();

// 创建矿石
createOre(type, pos);
}

}
```

5.1.2 挖矿机制

实现位置: [Classes/MiningManager.cpp](#)

```
bool MiningManager::mineOre(const Vec2& position) {
    // 1. 查找范围内的矿石
    OreSpawn* targetOre = nullptr;
    float minDistance = 50.0f;

    for (auto& ore : ores_) {
        float distance = position.distance(ore.position);
        if (distance < minDistance) {
            targetOre = &ore;
            break;
        }
    }

    if (!targetOre) return false;

    // 2. 减少矿石血量
    targetOre->health--;

    // 3. 播放挖矿动画
    playMiningAnimation(targetOre->sprite);

    // 4. 检查是否挖完
    if (targetOre->health <= 0) {
        // 4.1 添加矿石到背包
        ItemType itemType = getOreItemType(targetOre->type);
        int count = 1 + cocos2d::random(0, 2); // 1-3个
        InventoryManager::getInstance()->addItem(itemType, count);

        // 4.2 增加挖矿经验
        int exp = getOreExperience(targetOre->type);
        SkillManager::getInstance()->addExperience(SkillType::Mining, exp);

        // 4.3 移除矿石
        targetOre->sprite->removeFromParent();
        ores_.erase(std::remove_if(ores_.begin(), ores_.end(),
            [targetOre](const OreSpawn& ore) {
                return &ore == targetOre;
            }), ores_.end());
    }
}
```

```

        return true;
    }

    return false;
}

```

经验值表:

矿石	挖掘次数	经验值	掉落物品
铜矿	3	5	铜矿石 1-3个
银矿	4	10	银矿石 1-3个
金矿	5	20	金矿石 1-3个
钻石	8	50	钻石 1-2个

5.2 许愿池系统

实现位置: [Classes/MineScene.cpp](#)

```

void MineScene::handleWishingWell() {
    // 1. 检查玩家是否持有物品
    ItemType currentItem = player_->getCurrentToolbarItem();
    if (currentItem == ItemType::ITEM_NONE) {
        showMessage("You need an item to make a wish");
        return;
    }

    // 2. 从背包移除物品
    InventoryManager::getInstance()->removeItem(currentItem, 1);

    // 3. 播放许愿动画
    playWishingAnimation();

    // 4. 随机奖励
    float roll = cocos2d::random(0.0f, 1.0f);

    if (roll < 0.4f) {
        // 40% 概率：金币奖励
        int goldAmount = 100 + cocos2d::random(0, 200);
        InventoryManager::getInstance()->addMoney(goldAmount);
        showMessage("Received " + std::to_string(goldAmount) + " gold!");

    } else if (roll < 0.7f) {
        // 30% 概率：生命恢复
        int healAmount = 20 + cocos2d::random(0, 30);
        player_->heal(healAmount);
        showMessage("Healed " + std::to_string(healAmount) + " HP!");
    }
}

```

```

} else if (roll < 0.9f) {
    // 20% 概率：铁剑
    InventoryManager::getInstance()->addItem(ItemType::ITEM_IronSword, 1);
    showMessage("Received an Iron Sword!");

} else if (roll < 0.97f) {
    // 7% 概率：金剑
    InventoryManager::getInstance()->addItem(ItemType::ITEM_GoldSword, 1);
    showMessage("Received a Golden Sword!");

} else {
    // 3% 概率：钻石剑
    InventoryManager::getInstance()->addItem(ItemType::ITEM_DiamondSword, 1);
    showMessage("JACKPOT! Received a Diamond Sword!");
}
}

```

奖励概率分布:

金币 (100-300): 40%
 生命恢复 (20-50): 30%
 铁剑: 20%
 金剑: 7%
 钻石剑: 3%

6. 战斗系统

6.1 怪物AI系统

文件位置: [Classes/Monster.h](#), [Classes/Monster.cpp](#)

6.1.1 怪物基类

```

class Monster : public cocos2d::Sprite {
protected:
    // 基础属性
    int hp_;                                // 当前血量
    int maxHp_;                             // 最大血量
    float attackPower_;                      // 攻击力
    float moveSpeed_;                        // 移动速度
    float attackRange_;                      // 攻击范围
    float detectionRange_;                  // 侦测范围

    // AI状态
    enum class AISState {
        IDLE,                               // 空闲
        PATROL,                             // 巡逻
        CHASE,                              // 追击
    };
}

```

```
    ATTACK          // 攻击
};

AIState aiState_;

// 目标
Player* targetPlayer_;

// 动画
cocos2d::Animate* walkAnimation_;
cocos2d::Animate* attackAnimation_;


public:
    virtual void update(float delta) override;
    virtual void takeDamage(int damage);
    virtual void onDeath();

protected:
    virtual void updateAI(float delta);
    virtual void initStats() = 0;           // 纯虚函数
    virtual void initDisplay() = 0;         // 纯虚函数
};
```

6.1.2 AI更新逻辑

实现位置: [Classes/Monster.cpp](#)

```
void Monster::update(float delta) {
    if (isDead()) return;

    // 1. 更新AI
    updateAI(delta);

    // 2. 更新移动
    updateMovement(delta);

    // 3. 更新动画
    updateAnimation(delta);
}

void Monster::updateAI(float delta) {
    if (!targetPlayer_) return;

    // 1. 计算与玩家距离
    float distance = this->getPosition().distance(targetPlayer_->getPosition());

    // 2. 状态机
    switch (aiState_) {
        case AIState::IDLE:
            // 2.1 检测玩家
            if (distance < detectionRange_) {
                aiState_ = AIState::CHASE;
```

```

        CCLLOG("Monster detected player!");
    }
    break;

case AIState::CHASE:
    // 2.2 追击玩家
    if (distance > detectionRange_) {
        // 超出侦测范围，返回空闲
        aiState_ = AIState::IDLE;
    } else if (distance < attackRange_) {
        // 进入攻击范围
        aiState_ = AIState::ATTACK;
        attackTimer_ = 0.0f;
    } else {
        // 向玩家移动
        Vec2 direction = targetPlayer_->getPosition() - this->getPosition();
        direction.normalize();

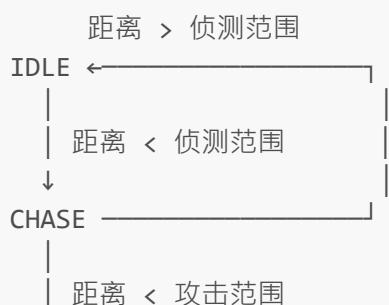
        Vec2 newPos = this->getPosition() + direction * moveSpeed_ * delta;
        this->setPosition(newPos);
    }
    break;

case AIState::ATTACK:
    // 2.3 攻击玩家
    attackTimer_ += delta;

    if (distance > attackRange_) {
        // 玩家逃离攻击范围
        aiState_ = AIState::CHASE;
    } else if (attackTimer_ >= attackCooldown_) {
        // 执行攻击
        performAttack();
        attackTimer_ = 0.0f;
    }
    break;
}
}

```

AI状态转换图:



↓
ATTACK

6.1.3 史莱姆实现

实现位置: [Classes/Slime.cpp](#)

```
void Slime::initStats() {
    maxHp_ = 50;
    hp_ = maxHp_;
    attackPower_ = 10.0f;
    moveSpeed_ = 100.0f;           // 移动快
    attackRange_ = 50.0f;
    detectionRange_ = 300.0f;
    attackCooldown_ = 1.5f;        // 攻击快
}

void Slime::initDisplay() {
    // 创建简单的精灵
    this->setTexture("monsters/slime.png");
    this->setScale(0.8f);

    // 添加血条
    createHealthBar();
}
```

6.1.4 僵尸实现

实现位置: [Classes/Zombie.cpp](#)

```
void Zombie::initStats() {
    maxHp_ = 100;                  // 血量高
    hp_ = maxHp_;
    attackPower_ = 20.0f;           // 攻击高
    moveSpeed_ = 50.0f;             // 移动慢
    attackRange_ = 60.0f;
    detectionRange_ = 250.0f;
    attackCooldown_ = 2.0f;         // 攻击慢
}
```

怪物属性对比:

属性	史莱姆	僵尸
血量	50	100
攻击力	10	20

属性	史莱姆	僵尸
移动速度	100	50
攻击冷却	1.5秒	2.0秒
特点	快速灵活	强力坦克

6.2 战斗伤害计算

6.2.1 玩家攻击怪物

实现位置: [Classes/GameScene.cpp](#)

```
void GameScene::handlePlayerAttack() {
    // 1. 获取玩家朝向
    Vec2 facingDirection = player_->getFacingDirection();
    Vec2 attackPos = player_->getPosition() + facingDirection * 50;

    // 2. 检测攻击范围内的怪物
    for (auto* monster : monsters_) {
        if (!monster || monster->isDead()) continue;

        float distance = attackPos.distance(monster->getPosition());
        if (distance < 60.0f) {
            // 3. 计算伤害
            int damage = calculatePlayerDamage();

            // 4. 应用伤害
            monster->takeDamage(damage);

            // 5. 击退效果
            Vec2 knockback = monster->getPosition() - player_->getPosition();
            knockback.normalize();
            knockback *= 100.0f;
            monster->applyKnockback(knockback);

            // 6. 播放受击动画
            monster->playHitAnimation();

            // 7. 检查死亡
            if (monster->isDead()) {
                handleMonsterDeath(monster);
            }

            break; // 一次只攻击一个怪物
        }
    }

    int GameScene::calculatePlayerDamage() {
        // 基础伤害
    }
}
```

```

int baseDamage = 10;

// 武器加成
ItemType weapon = player_->getCurrentWeapon();
switch (weapon) {
    case ItemType::ITEM_WoodenSword:
        baseDamage = 15;
        break;
    case ItemType::ITEM_IronSword:
        baseDamage = 25;
        break;
    case ItemType::ITEM_GoldSword:
        baseDamage = 40;
        break;
    case ItemType::ITEM_DiamondSword:
        baseDamage = 60;
        break;
}
}

// 随机浮动 ±20%
float multiplier = 0.8f + cocos2d::random(0.0f, 0.4f);

return static_cast<int>(baseDamage * multiplier);
}

```

武器伤害表:

武器	基础伤害	获取方式
空手	10	初始
木剑	15	矿洞1层掉落
铁剑	25	许愿池/矿洞3层
金剑	40	许愿池稀有
钻石剑	60	许愿池极稀有

6.2.2 怪物攻击玩家

实现位置: [Classes/Monster.cpp](#)

```

void Monster::performAttack() {
    if (!targetPlayer_) return;

    // 1. 检查距离
    float distance = this->getPosition().distance(targetPlayer_->getPosition());
    if (distance > attackRange_) return;

    // 2. 计算伤害 ( 带随机浮动 )
    float multiplier = 0.8f + cocos2d::random(0.0f, 0.4f);
}

```

```
int damage = static_cast<int>(attackPower_ * multiplier);

// 3. 应用伤害
targetPlayer_->takeDamage(damage);

// 4. 播放攻击动画
playAttackAnimation();

// 5. 播放音效
AudioEngine::play2d("audio/monster_attack.mp3");
}
```

6.2.3 无敌帧系统

实现位置: [Classes/Player.cpp](#)

```
void Player::takeDamage(int damage) {
    // 1. 检查无敌状态
    if (isInvulnerable_) {
        return;
    }

    // 2. 扣除血量
    hp_ -= damage;
    if (hp_ < 0) hp_ = 0;

    // 3. 更新UI
    updateHealthDisplay();

    // 4. 播放受击音效
    AudioEngine::play2d("audio/player_hit.mp3");

    // 5. 受击闪烁动画
    playHitFlashAnimation();

    // 6. 启动无敌帧 (1秒)
    isInvulnerable_ = true;
    this->scheduleOnce([this](float) {
        isInvulnerable_ = false;
    }, 1.0f, "invulnerable_timer");

    // 7. 检查死亡
    if (hp_ <= 0) {
        onDeath();
    }
}
```

无敌帧作用:

- 防止连续受击

- 给玩家反应时间
 - 持续时间: 1秒
-

7. 背包与物品系统

文件位置: [Classes/InventoryManager.h](#), [Classes/InventoryManager.cpp](#)

7.1 物品数据结构

```
// 物品类型枚举
enum class ItemType {
    ITEM_NONE = 0,

    // 工具
    ITEM_Hoe = 1,
    ITEM_WateringCan = 2,
    ITEM_Scythe = 3,
    ITEM_Axe = 4,
    ITEM_FishingRod = 5,

    // 种子
    ITEM_RadishSeed = 6,
    ITEM_PotatoSeed = 7,
    ITEM_CornSeed = 8,

    // 作物
    ITEM_Radish = 20,
    ITEM_Potato = 21,
    ITEM_Corn = 22,

    // 武器
    ITEM_WoodenSword = 40,
    ITEM_IronSword = 41,
    ITEM_GoldSword = 42,
    ITEM_DiamondSword = 43,

    // 矿石
    ITEM_CopperOre = 60,
    ITEM_SilverOre = 61,
    ITEM_GoldOre = 62,
    ITEM_Diamond = 63,

    // 鱼类
    ITEM_Anchovy = 80,
    ITEM_Carp = 81,
    ITEM_Eel = 82,
    // ... 更多鱼类
};

// 物品槽结构
struct ItemSlot {
```

```

ItemType type;           // 物品类型
int count;               // 数量

bool isEmpty() const {
    return type == ItemType::ITEM_NONE || count <= 0;
}

};

```

7.2 背包管理器

```

class InventoryManager : public cocos2d::Node {
private:
    std::vector<ItemSlot> slots_;   // 物品槽数组
    int money_;                      // 金币
    int maxSlots_;                  // 最大格子数

    static InventoryManager* s_instance; // 单例实例

public:
    // 单例访问
    static InventoryManager* getInstance();
    static void destroyInstance();

    // 物品操作
    bool addItem(ItemType type, int count);
    bool removeItem(ItemType type, int count);
    bool hasItem(ItemType type, int count = 1) const;

    // 槽位操作
    const ItemSlot& getSlot(int slotIndex) const;
    void setSlot(int slotIndex, ItemType type, int count);
    void swapSlots(int index1, int index2);

    // 金币操作
    void addMoney(int amount);
    bool spendMoney(int amount);
    int getMoney() const { return money_; }
};


```

7.3 物品添加逻辑

实现位置: [Classes/InventoryManager.cpp](#)

```

bool InventoryManager::addItem(ItemType type, int count) {
    if (type == ItemType::ITEM_NONE || count <= 0) {
        return false;
    }

    // 1. 检查是否可堆叠

```

```

if (isStackable(type)) {
    // 1.1 查找现有相同物品
    for (auto& slot : slots_) {
        if (slot.type == type) {
            slot.count += count;
            CCLOG("Added %d %s to existing stack (now %d)",
                  count, getItemName(type).c_str(), slot.count);
            return true;
        }
    }
}

// 2. 查找空槽位
for (auto& slot : slots_) {
    if (slot.isEmpty()) {
        slot.type = type;
        slot.count = count;
        CCLOG("Added %d %s to new slot", count, getItemName(type).c_str());
        return true;
    }
}

// 3. 背包已满
CCLOG("Inventory full! Cannot add %s", getItemName(type).c_str());
return false;
}

```

堆叠规则:

- 可堆叠: 种子、作物、矿石、鱼类
- 不可堆叠: 工具、武器

7.4 背包UI

文件位置: [Classes/InventoryUI.h](#), [Classes/InventoryUI.cpp](#)

7.4.1 UI布局

```

void InventoryUI::initUI() {
    // 1. 创建背景面板
    auto bg = Sprite::create("ui/inventory_bg.png");
    bg->setPosition(visibleSize / 2);
    this->addChild(bg);

    // 2. 创建格子 (5行 × 8列 = 40格)
    const int rows = 5;
    const int cols = 8;
    const float slotSize = 60.0f;
    const float padding = 5.0f;

    Vec2 startPos = Vec2(100, 500);

```

```

for (int row = 0; row < rows; ++row) {
    for (int col = 0; col < cols; ++col) {
        int index = row * cols + col;

        // 创建格子背景
        auto slotBg = Sprite::create("ui/slot_bg.png");
        Vec2 pos = startPos + Vec2(
            col * (slotSize + padding),
            -row * (slotSize + padding)
        );
        slotBg->setPosition(pos);
        this->addChild(slotBg);

        // 创建物品图标
        auto icon = Sprite::create();
        icon->setPosition(pos);
        this->addChild(icon);
        slotIcons_[index] = icon;

        // 创建数量标签
        auto countLabel = Label::createWithSystemFont("", "Arial", 18);
        countLabel->setPosition(pos + Vec2(15, -15));
        this->addChild(countLabel);
        countLabels_[index] = countLabel;
    }
}

// 3. 创建金币显示
moneyLabel_ = Label::createWithSystemFont("", "Arial", 24);
moneyLabel_->setPosition(Vec2(visibleSize.width / 2, 100));
this->addChild(moneyLabel_);
}

```

7.4.2 物品交换逻辑

实现位置: [Classes/InventoryUI.cpp](#)

```

void InventoryUI::handleSlotClick(int slotIndex) {
    // 1. 第一次点击 - 选中槽位
    if (selectedSlotIndex_ == -1) {
        selectedSlotIndex_ = slotIndex;
        highlightSlot(slotIndex);
        return;
    }

    // 2. 第二次点击同一个槽位 - 取消选择
    if (selectedSlotIndex_ == slotIndex) {
        selectedSlotIndex_ = -1;
        clearHighlight();
        return;
    }
}

```

```

    }

    // 3. 第二次点击不同槽位 - 交换物品
    InventoryManager::getInstance()->swapSlots(selectedSlotIndex_, slotIndex);

    // 4. 更新显示
    updateSlotDisplay(selectedSlotIndex_);
    updateSlotDisplay(slotIndex);

    // 5. 清除选择
    selectedSlotIndex_ = -1;
    clearHighlight();
}

}

```

7.5 物品信息系统

实现位置: [Classes/InventoryManager.cpp](#)

```

std::string InventoryManager::getItemName(ItemType type) const {
    static const std::unordered_map<ItemType, std::string> names = {
        {ItemType::ITEM_Hoe, "Hoe"},
        {ItemType::ITEM_WateringCan, "Watering Can"},
        {ItemType::ITEM_Scythe, "Scythe"},
        {ItemType::ITEM_Axe, "Axe"},
        {ItemType::ITEM_FishingRod, "Fishing Rod"},

        {ItemType::ITEM_RadishSeed, "Radish Seed"},
        {ItemType::ITEM_PotatoSeed, "Potato Seed"},
        {ItemType::ITEM_CornSeed, "Corn Seed"},

        {ItemType::ITEM_Radish, "Radish"},
        {ItemType::ITEM_Potato, "Potato"},
        {ItemType::ITEM_Corn, "Corn"},

        {ItemType::ITEM_WoodenSword, "Wooden Sword"},
        {ItemType::ITEM_IronSword, "Iron Sword"},
        {ItemType::ITEM_GoldSword, "Gold Sword"},
        {ItemType::ITEM_DiamondSword, "Diamond Sword"},

        {ItemType::ITEM_CopperOre, "Copper Ore"},
        {ItemType::ITEM_SilverOre, "Silver Ore"},
        {ItemType::ITEM_GoldOre, "Gold Ore"},
        {ItemType::ITEM_Diamond, "Diamond"},

        // ... 更多物品
    };

    auto it = names.find(type);
    if (it != names.end()) {
        return it->second;
    }
    return "Unknown Item";
}

```

```
}

std::string InventoryManager::getItemSpritePath(ItemType type) const {
    return "items/" + getItemName(type) + ".png";
}
```

8. 存档系统

文件位置: [Classes/SaveManager.h](#), [Classes/SaveManager.cpp](#)

8.1 存档数据结构

```
struct SaveData {
    // 玩家数据
    cocos2d::Vec2 playerPosition;
    int playerHp;
    int playerMaxHp;
    int playerEnergy;

    // 背包数据
    struct InventoryData {
        struct ItemSlotData {
            int type;          // ItemType转为int
            int count;
        };
        std::vector<ItemSlotData> slots;
        int money;
    } inventory;

    // 农场数据
    struct FarmTileData {
        int x, y;           // 图块坐标
        bool isTilled;
        bool isWatered;
        bool hasPlant;
        int cropId;
        int stage;
        int progressDays;
    };
    std::vector<FarmTileData> farmTiles;

    // 技能数据
    struct SkillData {
        int type;          // SkillType转为int
        int level;
        int experience;
    };
    std::vector<SkillData> skills;

    // 游戏进度
}
```

```
    int dayCount;
    int currentFloor;      // 矿洞楼层
};
```

8.2 JSON序列化

实现位置: [Classes/SaveManager.cpp](#)

```
rapidjson::Document SaveManager::serializeToJson(const SaveData& data) {
    using namespace rapidjson;
    Document doc;
    doc.SetObject();
    auto& allocator = doc.GetAllocator();

    // 1. 玩家位置
    Value playerPos(kObjectType);
    playerPos.AddMember("x", data.playerPosition.x, allocator);
    playerPos.AddMember("y", data.playerPosition.y, allocator);
    doc.AddMember("playerPosition", playerPos, allocator);

    // 2. 玩家状态
    doc.AddMember("playerHp", data.playerHp, allocator);
    doc.AddMember("playerMaxHp", data.playerMaxHp, allocator);
    doc.AddMember("playerEnergy", data.playerEnergy, allocator);

    // 3. 背包数据
    Value inventory(kObjectType);

    // 3.1 物品槽
    Value slotsArray(kArrayType);
    for (const auto& slot : data.inventory.slots) {
        Value slotObj(kObjectType);
        slotObj.AddMember("type", slot.type, allocator);
        slotObj.AddMember("count", slot.count, allocator);
        slotsArray.PushBack(slotObj, allocator);
    }
    inventory.AddMember("slots", slotsArray, allocator);

    // 3.2 金币
    inventory.AddMember("money", data.inventory.money, allocator);
    doc.AddMember("inventory", inventory, allocator);

    // 4. 农田数据
    Value tilesArray(kArrayType);
    for (const auto& tile : data.farmTiles) {
        // 只保存有作物的图块
        if (!tile.hasPlant) continue;

        Value tileObj(kObjectType);
        tileObj.AddMember("x", tile.x, allocator);
        tileObj.AddMember("y", tile.y, allocator);
```

```

        tileObj.AddMember("cropId", tile.cropId, allocator);
        tileObj.AddMember("stage", tile.stage, allocator);
        tileObj.AddMember("progressDays", tile.progressDays, allocator);
        tileObj.AddMember("isWatered", tile.isWatered, allocator);
        tilesArray.PushBack(tileObj, allocator);
    }
    doc.AddMember("farmTiles", tilesArray, allocator);

    // 5. 技能数据
    Value skillsArray(kArrayType);
    for (const auto& skill : data.skills) {
        Value skillObj(kObjectType);
        skillObj.AddMember("type", skill.type, allocator);
        skillObj.AddMember("level", skill.level, allocator);
        skillObj.AddMember("experience", skill.experience, allocator);
        skillsArray.PushBack(skillObj, allocator);
    }
    doc.AddMember("skills", skillsArray, allocator);

    // 6. 游戏进度
    doc.AddMember("dayCount", data.dayCount, allocator);
    doc.AddMember("currentFloor", data.currentFloor, allocator);

    return doc;
}

```

生成的JSON示例:

```
{
    "playerPosition": {"x": 400.0, "y": 300.0},
    "playerHp": 80,
    "playerMaxHp": 100,
    "playerEnergy": 100,
    "inventory": {
        "slots": [
            {"type": 1, "count": 1},
            {"type": 6, "count": 10},
            {"type": 20, "count": 5}
        ],
        "money": 1500
    },
    "farmTiles": [
        {
            "x": 10, "y": 5,
            "cropId": 1,
            "stage": 2,
            "progressDays": 4,
            "isWatered": true
        }
    ],
    "skills": [
        {"type": 0, "level": 3, "experience": 250},

```

```

        {"type": 1, "level": 2, "experience": 100}
    ],
    "dayCount": 7,
    "currentFloor": 3
}

```

8.3 反序列化

实现位置: [Classes/SaveManager.cpp](#)

```

bool SaveManager::deserializeFromJson(const rapidjson::Document& doc, SaveData&
data) {
    try {
        // 1. 加载玩家位置
        if (doc.HasMember("playerPosition") && doc["playerPosition"].IsObject()) {
            const auto& pos = doc["playerPosition"];
            data.playerPosition.x = pos["x"].GetFloat();
            data.playerPosition.y = pos["y"].GetFloat();
        }

        // 2. 加载玩家状态
        if (doc.HasMember("playerHp")) {
            data.playerHp = doc["playerHp"].GetInt();
        }
        if (doc.HasMember("playerMaxHp")) {
            data.playerMaxHp = doc["playerMaxHp"].GetInt();
        }
        if (doc.HasMember("playerEnergy")) {
            data.playerEnergy = doc["playerEnergy"].GetInt();
        }

        // 3. 加载背包数据
        if (doc.HasMember("inventory") && doc["inventory"].IsObject()) {
            const auto& inv = doc["inventory"];

            // 3.1 加载物品槽
            data.inventory.slots.clear();
            if (inv.HasMember("slots") && inv["slots"].IsArray()) {
                const auto& slotsArray = inv["slots"].GetArray();
                for (rapidjson::SizeType i = 0; i < slotsArray.Size(); ++i) {
                    const auto& slotObj = slotsArray[i];
                    SaveData::InventoryData::ItemSlotData slot;
                    slot.type = slotObj["type"].GetInt();
                    slot.count = slotObj["count"].GetInt();
                    data.inventory.slots.push_back(slot);
                }
            }

            // 3.2 加载金币
            if (inv.HasMember("money")) {
                data.inventory.money = inv["money"].GetInt();
            }
        }
    }
}

```

```
        }

    }

    // 4. 加载农田数据
    data.farmTiles.clear();
    if (doc.HasMember("farmTiles") && doc["farmTiles"].IsArray()) {
        const auto& tilesArray = doc["farmTiles"].GetArray();
        for (rapidjson::SizeType i = 0; i < tilesArray.Size(); ++i) {
            const auto& tileObj = tilesArray[i];
            SaveData::FarmTileData tile;
            tile.x = tileObj["x"].GetInt();
            tile.y = tileObj["y"].GetInt();
            tile.isTilled = true;
            tile.hasPlant = true;
            tile.cropId = tileObj["cropId"].GetInt();
            tile.stage = tileObj["stage"].GetInt();
            tile.progressDays = tileObj["progressDays"].GetInt();
            tile.isWatered = tileObj["isWatered"].GetBool();
            data.farmTiles.push_back(tile);
        }
    }

    // 5. 加载技能数据
    data.skills.clear();
    if (doc.HasMember("skills") && doc["skills"].IsArray()) {
        const auto& skillsArray = doc["skills"].GetArray();
        for (rapidjson::SizeType i = 0; i < skillsArray.Size(); ++i) {
            const auto& skillObj = skillsArray[i];
            SaveData::SkillData skill;
            skill.type = skillObj["type"].GetInt();
            skill.level = skillObj["level"].GetInt();
            skill.experience = skillObj["experience"].GetInt();
            data.skills.push_back(skill);
        }
    }

    // 6. 加载游戏进度
    if (doc.HasMember("dayCount")) {
        data.dayCount = doc["dayCount"].GetInt();
    }
    if (doc.HasMember("currentFloor")) {
        data.currentFloor = doc["currentFloor"].GetInt();
    }

    return true;

} catch (const std::exception& e) {
    CCLOG("Error: Exception during deserialization: %s", e.what());
    return false;
} catch (...) {
    CCLOG("Error: Unknown exception during save deserialization");
    return false;
}
}
```

8.4 文件I/O操作

实现位置: [Classes/SaveManager.cpp](#)

```
bool SaveManager::saveGame(const SaveData& data) {
    try {
        std::string path = getSaveFilePath();
        CCLOG("Saving game to: %s", path.c_str());

        // 1. 序列化为JSON
        rapidjson::Document doc = serializeToJson(data);

        // 2. 转换为字符串 ( 带格式化 )
        rapidjson::StringBuffer buffer;
        rapidjson::PrettyWriter<rapidjson::StringBuffer> writer(buffer);
        doc.Accept(writer);

        std::string jsonStr = buffer.GetString();

        // 3. 写入文件
        FILE* file = fopen(path.c_str(), "w");
        if (!file) {
            CCLOG("Error: Failed to open save file for writing: %s",
path.c_str());
            return false;
        }

        size_t written = fwrite(jsonStr.c_str(), 1, jsonStr.length(), file);
        fclose(file);

        if (written != jsonStr.length()) {
            CCLOG("Error: Failed to write complete save data (wrote %zu of %zu
bytes)",
written, jsonStr.length());
            return false;
        }

        CCLOG("Game saved successfully!");
        return true;
    } catch (const std::exception& e) {
        CCLOG("Error: Exception during save operation: %s", e.what());
        return false;
    } catch (...) {
        CCLOG("Error: Unknown exception during save operation");
        return false;
    }
}

bool SaveManager::loadGame(SaveData& data) {
    try {
```

```
    std::string path = getSaveFilePath();
    CCLOG("Loading game from: %s", path.c_str());

    // 1. 检查文件是否存在
    if (!FileUtils::getInstance()->isFileExist(path)) {
        CCLOG("Error: Save file does not exist");
        return false;
    }

    // 2. 读取文件内容
    std::string jsonStr = FileUtils::getInstance()->getStringFromFile(path);
    if (jsonStr.empty()) {
        CCLOG("Error: Failed to read save file or file is empty");
        return false;
    }

    // 3. 解析JSON
    rapidjson::Document doc;
    doc.Parse(jsonStr.c_str());

    if (doc.HasParseError()) {
        CCLOG("Error: Failed to parse JSON at offset %zu: %d",
              doc.GetErrorOffset(), doc.GetParseError());
        return false;
    }

    // 4. 反序列化
    if (!deserializeFromJson(doc, data)) {
        CCLOG("Error: Failed to deserialize save data");
        return false;
    }

    CCLOG("Game loaded successfully!");
    return true;

} catch (const std::exception& e) {
    CCLOG("Error: Exception during load operation: %s", e.what());
    return false;
} catch (...) {
    CCLOG("Error: Unknown exception during load operation");
    return false;
}
}
```

存档文件路径:

- Windows: C:/Users/<用户名>/AppData/Roaming/StardewFarm/save.json
- macOS: ~/Library/Application Support/StardewFarm/save.json
- Linux: ~/.config/StardewFarm/save.json

8.5 自动存档机制

实现位置: [Classes/GameScene.cpp](#)

```
void GameScene::autoSave() {
    CCLOG("Auto-saving game...");

    // 1. 收集游戏数据
    SaveData data;

    // 1.1 玩家数据
    data.playerPosition = player_->getPosition();
    data.playerHp = player_->getHp();
    data.playerMaxHp = player_->getMaxHp();
    data.playerEnergy = player_->getEnergy();

    // 1.2 背包数据
    InventoryManager* inv = InventoryManager::getInstance();
    for (int i = 0; i < inv->getSlotCount(); ++i) {
        const auto& slot = inv->getSlot(i);
        data.inventory.slots.push_back({
            static_cast<int>(slot.type),
            slot.count
        });
    }
    data.inventory.money = inv->getMoney();

    // 1.3 农田数据
    data.farmTiles = farmManager_->getAllTiles();

    // 1.4 技能数据
    data.skills = SkillManager::getInstance()->getAllSkillsData();

    // 1.5 游戏进度
    data.dayCount = TimeManager::getInstance()->getDayCount();

    // 2. 保存到文件
    SaveManager::getInstance()->saveGame(data);
}
```

自动存档触发时机:

1. 进入矿洞前
2. 每天睡觉时
3. 退出游戏时
4. 场景切换时

9. 技能系统

文件位置: [Classes/SkillManager.h](#), [Classes/SkillManager.cpp](#)

9.1 技能数据结构

```

enum class SkillType {
    Agriculture,      // 农业
    Mining,           // 采矿
    Fishing,          // 钓鱼
    Cooking,          // 烹饪
    Count             // 总数 ( 用于数组大小 )
};

struct SkillData {
    std::string name;        // 技能名称
    int level;               // 当前等级
    int currentExp;          // 当前经验
    int expToNextLevel;      // 升级所需经验
    int maxLevel;            // 最大等级
};

class SkillManager : public cocos2d::Node {
private:
    std::array<SkillData, static_cast<size_t>(SkillType::Count)> skills_;

    using LevelUpCallback = std::function<void(SkillType, int)>;
    LevelUpCallback onLevelUp_;
};

```

9.2 经验值系统

实现位置: [Classes/SkillManager.cpp](#)

```

void SkillManager::addExperience(SkillType type, int exp) {
    // 1. 获取技能数据
    auto& skill = skills_[toIndex(type)];

    // 2. 增加经验
    skill.currentExp += exp;

    CCLOG("Added %d exp to %s (now %d/%d)",
          exp, skill.name.c_str(), skill.currentExp, skill.expToNextLevel);

    // 3. 检查升级
    while (skill.currentExp >= skill.expToNextLevel && skill.level <
skill.maxLevel) {
        levelUp(type);
    }
}

void SkillManager::levelUp(SkillType type) {
    auto& skill = skills_[toIndex(type)];

    // 1. 升级
    skill.level++;
}

```

```

// 2. 扣除升级所需经验
skill.currentExp -= skill.expToNextLevel;

// 3. 计算下一级所需经验 (递增公式)
skill.expToNextLevel = calculateExpForLevel(skill.level + 1);

CCLOG("LEVEL UP! %s reached level %d!", skill.name.c_str(), skill.level);

// 4. 触发升级回调
if (onLevelUp_) {
    onLevelUp_(type, skill.level);
}

// 5. 播放升级特效
playLevelUpEffect(type);
}

int SkillManager::calculateExpForLevel(int level) {
    // 经验公式 : baseExp × level × level
    const int baseExp = 50;
    return baseExp * level * level;
}

```

升级经验表:

等级	所需总经验	本级所需
1→2	200	200
2→3	650	450
3→4	1450	800
4→5	2700	1250
5→6	4450	1750

公式: 经验 = 50 × level²

9.3 技能加成效果

实现位置: [Classes/SkillManager.cpp](#)

```

// 农业技能加成
int SkillManager::getExtraHarvestChance(int agricultureLevel) {
    // 每级增加10%额外收获几率
    return agricultureLevel * 10;
}

// 采矿技能加成
float SkillManager::getReducedMiningStamina(int miningLevel) {

```

```

    // 每级减少5%体力消耗
    return 1.0f - (miningLevel * 0.05f);
}

// 钓鱼技能加成
float SkillManager::getFishingDifficulty(int fishingLevel) {
    // 每级降低8%难度
    return 1.0f - (fishingLevel * 0.08f);
}

int SkillManager::getFishPriceBonus(int fishingLevel) {
    // 每级增加5%鱼价格
    return fishingLevel * 5;
}

```

技能效果总览:

技能	等级1	等级3	等级5
农业 - 额外收获	10%	30%	50%
采矿 - 体力减免	5%	15%	25%
钓鱼 - 难度降低	8%	24%	40%
钓鱼 - 价格加成	5%	15%	25%

9.4 技能树UI

文件位置: [Classes/SkillTreeUI.h](#), [Classes/SkillTreeUI.cpp](#)

```

void SkillTreeUI::initUI() {
    // 1. 背景
    auto bg = LayerColor::create(Color4B(30, 30, 40, 230));
    this->addChild(bg);

    // 2. 标题
    auto title = Label::createWithSystemFont("Skills", "Arial", 48);
    title->setPosition(Vec2(visibleSize.width / 2, visibleSize.height - 50));
    this->addChild(title);

    // 3. 技能列表
    const float yStart = 500;
    const float ySpacing = 100;

    for (int i = 0; i < static_cast<int>(SkillType::Count); ++i) {
        SkillType type = static_cast<SkillType>(i);

        // 3.1 技能名称
        std::string name = SkillManager::getInstance()->getSkillName(type);
        auto nameLabel = Label::createWithSystemFont(name, "Arial", 32);
        nameLabel->setPosition(Vec2(200, yStart - i * ySpacing));
    }
}

```

```

    this->addChild(nameLabel);

    // 3.2 等级显示
    int level = SkillManager::getInstance()->getLevel(type);
    auto levelLabel = Label::createWithSystemFont(
        "Lv " + std::to_string(level),
        "Arial", 28
    );
    levelLabel->setPosition(Vec2(400, yStart - i * ySpacing));
    this->addChild(levelLabel);
    skillLevelLabels_[i] = levelLabel;

    // 3.3 经验条
    auto expBar = createProgressBar(300, 20);
    expBar->setPosition(Vec2(650, yStart - i * ySpacing));
    this->addChild(expBar);
    skillExpBars_[i] = expBar;

    // 3.4 经验文本
    int currentExp = SkillManager::getInstance()->getCurrentExp(type);
    int maxExp = SkillManager::getInstance()->getExpToNextLevel(type);
    auto expLabel = Label::createWithSystemFont(
        std::to_string(currentExp) + " / " + std::to_string(maxExp),
        "Arial", 20
    );
    expLabel->setPosition(Vec2(650, yStart - i * ySpacing - 30));
    this->addChild(expLabel);
    skillExpLabels_[i] = expLabel;
}

}

```

10. NPC与对话系统

文件位置: [Classes/Npc.h](#), [Classes/Npc.cpp](#), [Classes/DialogueBox.h](#), [Classes/DialogueBox.cpp](#)

10.1 NPC类结构

```

enum class NpcType {
    Villager, // 村民
    Merchant // 商人
};

class Npc : public cocos2d::Sprite {
private:
    std::string name_; // NPC名称
    std::vector<std::string> dialogues_; // 对话列表
    NpcType type_; // NPC类型

public:
    static Npc* create(const std::string& name,

```

```

        const std::string& spriteFile,
        NpcType type = NpcType::Villager);

std::string getNpcName() const { return name_; }
std::string getDialogue() const; // 随机对话
const std::vector<std::string>& getDialogues() const { return dialogues_; }
std::string getPortraitFile() const;

bool isMerchant() const { return type_ == NpcType::Merchant; }
};


```

10.2 对话框系统

```

class DialogueBox : public cocos2d::Node {
private:
    Npc* npc_; // 当前NPC
    cocos2d::Label* dialogue_label_; // 对话文本
    cocos2d::Sprite* background_; // 背景
    cocos2d::Sprite* portrait_; // 头像
    bool is_visible_; // 是否显示
    bool waiting_for_choice_; // 等待选择

    // 选择UI
    cocos2d::Node* choice_node_;
    cocos2d::Label* option1_label_;
    cocos2d::Label* option2_label_;

    using ChoiceCallback = std::function<void(int)>;
    ChoiceCallback choice_callback_;

public:
    void showDialogue(const std::string& text);
    void showChoices(const std::string& option1,
                     const std::string& option2,
                     const ChoiceCallback& callback);
    void closeDialogue();
};


```

10.3 对话显示逻辑

实现位置: [Classes/DialogueBox.cpp](#)

```

void DialogueBox::showDialogue(const std::string& text) {
    // 1. 隐藏选择UI
    waiting_for_choice_ = false;
    choice_node_->setVisible(false);

    // 2. 显示对话框
    this->setVisible(true);
}


```

```
is_visible_ = true;

// 3. 设置对话文本
dialogue_label_->setString(text);

// 4. 文字打字机效果 ( 可选 )
// playTypewriterEffect(text);
}

void DialogueBox::showChoices(const std::string& option1,
                               const std::string& option2,
                               const ChoiceCallback& callback) {
    // 1. 设置选项文本
    option1_label_->setString(option1);
    option2_label_->setString(option2);
    choice_callback_ = callback;

    // 2. 显示选择UI
    waiting_for_choice_ = true;
    is_visible_ = true;
    this->setVisible(true);
    choice_node_->setVisible(true);
}
```

10.4 商人交易系统

实现位置: [Classes/GameScene.cpp](#)

```
void GameScene::startMerchantInteraction(Npc* merchant) {
    // 1. 显示欢迎对话
    dialogueBox_->setNpc(merchant);
    dialogueBox_->showDialogue("Welcome! What would you like to do?");

    // 2. 显示选择 ( 购买/出售 )
    dialogueBox_->showChoices("Buy", "Sell", [this](int choice) {
        if (choice == 0) {
            // 打开购买界面
            openMerchantShop();
        } else {
            // 打开出售界面
            openSellInterface();
        }
        dialogueBox_->closeDialogue();
    });
}

void GameScene::openMerchantShop() {
    // 商店物品列表
    struct ShopItem {
        ItemType type;
        int price;
```

```

        int stock;           // -1表示无限
    };

    std::vector<ShopItem> shopItems = {
        {ItemType::ITEM_RadishSeed, 20, -1},
        {ItemType::ITEM_PotatoSeed, 30, -1},
        {ItemType::ITEM_CornSeed, 50, -1},
        {ItemType::ITEM_WoodenSword, 100, 5},
        {ItemType::ITEM_IronSword, 500, 3},
    };

    // 打开商店UI
    marketUI_->showShop(shopItems);
}

```

11. 钓鱼系统

文件位置: [Classes/FishingLayer.h](#), [Classes/FishingLayer.cpp](#)

11.1 钓鱼机制

```

class FishingLayer : public cocos2d::Layer {
private:
    enum class FishingState {
        IDLE,           // 空闲
        CASTING,        // 抛竿
        WAITING,        // 等待咬钩
        HOOKING,        // 提竿
        MINIGAME,       // 小游戏
        CAUGHT          // 成功
    };

    FishingState state_;

    // 小游戏组件
    Sprite* fishIcon_;           // 鱼图标
    Sprite* sliderBar_;          // 滑块
    float fishPosition_;         // 鱼位置 (0-1)
    float sliderPosition_;        // 滑块位置 (0-1)
    float sliderVelocity_;       // 滑块速度
    float progress_;             // 捕获进度 (0-1)

    // 鱼类数据
    Fish* currentFish_;          // 当前鱼
    float difficulty_;            // 难度系数
};

```

11.2 抛竿逻辑

实现位置: [Classes/FishingLayer.cpp](#)

```
void FishingLayer::handleCast(const Vec2& targetPos) {
    // 1. 计算抛竿距离
    Vec2 playerPos = player_->getPosition();
    float distance = playerPos.distance(targetPos);

    // 2. 限制最大距离
    const float maxDistance = 300.0f;
    if (distance > maxDistance) {
        distance = maxDistance;
    }

    // 3. 播放抛竿动画
    playCastAnimation(targetPos, distance);

    // 4. 切换状态
    state_ = FishingState::WAITING;

    // 5. 开始等待咬钩 ( 随机时间 2-5秒 )
    float waitTime = 2.0f + cocos2d::random(0.0f, 3.0f);
    this->scheduleOnce([this](float) {
        onFishBite();
    }, waitTime, "fish_bite");
}

void FishingLayer::onFishBite() {
    // 1. 显示感叹号
    showExclamationMark();

    // 2. 播放音效
    AudioEngine::play2d("audio/fish_bite.mp3");

    // 3. 切换状态
    state_ = FishingState::HOOKING;

    // 4. 等待玩家反应 ( 1秒内点击 )
    hookingTimer_ = 0.0f;
    hookingTimeout_ = 1.0f;
}
```

11.3 钓鱼小游戏

实现位置: [Classes/FishingLayer.cpp](#)

```
void FishingLayer::startMinigame() {
    // 1. 选择鱼类
    currentFish_ = selectRandomFish();

    // 2. 设置难度
```

```
difficulty_ = currentFish_->getDifficulty();

// 技能加成
int fishingLevel = SkillManager::getInstance()->getLevel(SkillType::Fishing);
difficulty_ *= SkillManager::getInstance()-
>getFishingDifficulty(fishingLevel);

// 3. 初始化小游戏
fishPosition_ = 0.5f;
sliderPosition_ = 0.5f;
sliderVelocity_ = 0.0f;
progress_ = 0.0f;

// 4. 显示UI
showMinigameUI();

// 5. 切换状态
state_ = FishingState::MINIGAME;
}

void FishingLayer::updateMinigame(float delta) {
    // 1. 更新鱼位置 (随机移动)
    float fishSpeed = 0.5f * difficulty_;
    float randomMove = (cocos2d::random(-1.0f, 1.0f)) * fishSpeed * delta;
    fishPosition_ += randomMove;
    fishPosition_ = clampf(fishPosition_, 0.0f, 1.0f);

    // 2. 更新滑块位置
    if (isMousePressed_) {
        // 按住上升
        sliderVelocity_ = 1.5f;
    } else {
        // 松开下降
        sliderVelocity_ = -2.0f;
    }

    sliderPosition_ += sliderVelocity_ * delta;
    sliderPosition_ = clampf(sliderPosition_, 0.0f, 1.0f);

    // 3. 检查是否捕获
    const float catchRadius = 0.1f;
    if (abs(fishPosition_ - sliderPosition_) < catchRadius) {
        // 在范围内 · 增加进度
        progress_ += 0.5f * delta;
    } else {
        // 在范围外 · 减少进度
        progress_ -= 0.3f * delta;
    }

    progress_ = clampf(progress_, 0.0f, 1.0f);

    // 4. 检查成功或失败
    if (progress_ >= 1.0f) {
        onCatchSuccess();
    }
}
```

```
    } else if (progress_ <= 0.0f) {
        onCatchFailed();
    }

    // 5. 更新UI
    updateMinigameUI();
}
```

小游戏难度因素:

- 鱼的移动速度
- 捕获区域大小
- 滑块灵敏度
- 技能等级加成

11.4 鱼类系统

文件位置: [Classes/Fish.h](#)

```
class Fish {
protected:
    std::string name_;
    int basePrice_;
    float difficulty_;           // 0.5 - 2.0
    float rarity_;              // 0.0 - 1.0

public:
    virtual ~Fish() = default;

    virtual std::string getName() const { return name_; }
    virtual int getPrice() const { return basePrice_; }
    virtual float getDifficulty() const { return difficulty_; }
    virtual float getRarity() const { return rarity_; }
    virtual std::string getSpritePath() const = 0;
};

// 具体鱼类
class AnchovyFish : public Fish {
public:
    AnchovyFish() {
        name_ = "Anchovy";
        basePrice_ = 30;
        difficulty_ = 0.6f;
        rarity_ = 0.8f;
    }
    std::string getSpritePath() const override { return "fish/anchovy.png"; }
};

class SturgeonFish : public Fish {
public:
    SturgeonFish() {
```

```

        name_ = "Sturgeon";
        basePrice_ = 200;
        difficulty_ = 1.8f;
        rarity_ = 0.05f;
    }
    std::string getSpritePath() const override { return "fish/sturgeon.png"; }
};
```

鱼类数据表:

鱼类	价格	难度	稀有度	出现位置
凤尾鱼	30	0.6	80%	海滩
鲤鱼	50	0.7	60%	海滩
鳗鱼	80	1.0	40%	海滩
比目鱼	100	1.2	30%	海滩
鲈鱼	120	1.3	25%	海滩
河豚	150	1.5	15%	海滩
虹鳟鱼	180	1.6	10%	海滩
鲟鱼	200	1.8	5%	海滩

12. 天气系统

文件位置: [Classes/WeatherManager.h](#), [Classes/WeatherManager.cpp](#)

12.1 天气类型

```

enum class WeatherType {
    SUNNY,           // 晴天
    RAINY,           // 雨天
    SNOWY            // 雪天
};

class WeatherManager : public cocos2d::Node {
private:
    WeatherType currentWeather_;
    WeatherType nextDayWeather_;

    // 天气预报 (7天)
    std::array<WeatherType, 7> forecast_;

    // 视觉效果
    ParticleSystem* weatherParticles_;

public:
```

```
void updateWeather();
void generateForecast();
WeatherType getCurrentWeather() const { return currentWeather_; }
const std::array<WeatherType, 7>& getForecast() const { return forecast_; }
};
```

12.2 天气生成算法

实现位置: [Classes/WeatherManager.cpp](#)

```
void WeatherManager::generateForecast() {
    // 1. 获取当前季节 ( 简化 : 根据天数判断 )
    int day = TimeManager::getInstance()->getDayCount();
    Season season = getSeason(day);

    // 2. 根据季节设置天气概率
    std::vector<std::pair<WeatherType, float>> probabilities;

    switch (season) {
        case Season::SPRING:
            probabilities = {
                {WeatherType::SUNNY, 0.6f},
                {WeatherType::RAINY, 0.4f},
                {WeatherType::SNOWY, 0.0f}
            };
            break;

        case Season::SUMMER:
            probabilities = {
                {WeatherType::SUNNY, 0.8f},
                {WeatherType::RAINY, 0.2f},
                {WeatherType::SNOWY, 0.0f}
            };
            break;

        case Season::AUTUMN:
            probabilities = {
                {WeatherType::SUNNY, 0.5f},
                {WeatherType::RAINY, 0.5f},
                {WeatherType::SNOWY, 0.0f}
            };
            break;

        case Season::WINTER:
            probabilities = {
                {WeatherType::SUNNY, 0.3f},
                {WeatherType::RAINY, 0.2f},
                {WeatherType::SNOWY, 0.5f}
            };
            break;
    }
}
```

```
// 3. 生成7天预报
for (int i = 0; i < 7; ++i) {
    forecast_[i] = selectWeatherByProbability(probabilities);
}
}
```

12.3 天气效果

实现位置: [Classes/WeatherManager.cpp](#)

```
void WeatherManager::applyWeatherEffects() {
    switch (currentWeather_) {
        case WeatherType::SUNNY:
            // 无特殊效果
            break;

        case WeatherType::RAINY:
            // 1. 自动浇水
            FarmManager::getInstance()->handleRainDay();

            // 2. 雨滴粒子效果
            showRainParticles();

            // 3. 降低视野亮度
            adjustLighting(0.8f);
            break;

        case WeatherType::SNOWY:
            // 1. 雪花粒子效果
            showSnowParticles();

            // 2. 大幅提高物价
            adjustMarketPrices(1.5f);

            // 3. 降低视野亮度
            adjustLighting(0.9f);
            break;
    }
}

void WeatherManager::showRainParticles() {
    // 使用粒子系统创建雨滴效果
    auto rain = ParticleRain::create();
    rain->setPosition(Vec2(visibleSize.width / 2, visibleSize.height));
    rain->setLife(5.0f);
    rain->setLifeVar(0.5f);
    rain->setGravity(Vec2(0, -1000));
    rain->setSpeed(500);
    rain->setSpeedVar(100);
    this->addChild(rain, 100);
```

```
    weatherParticles_ = rain;
}
```

天气对游戏的影响:

天气	农田	物价	视觉
晴天	无	正常	明亮
雨天	自动浇水	正常	稍暗 + 雨滴
雪天	无	+50%	稍暗 + 雪花

13. 时间系统

文件位置: [Classes/TimeManager.h](#), [Classes/TimeManager.cpp](#)

13.1 时间数据结构

```
class TimeManager {
private:
    int currentHour_;           // 当前小时 (0-23)
    int currentMinute_;         // 当前分钟 (0-59)
    int dayCount_;              // 游戏天数

    float timeScale_;           // 时间流速 (1.0 = 正常)
    float accumulator_;          // 累加器

    static TimeManager* instance_;

public:
    static TimeManager* getInstance();

    void update(float delta);
    void advanceTime(int minutes);
    void skipToNextDay();

    int getHour() const { return currentHour_; }
    int getMinute() const { return currentMinute_; }
    int getDayCount() const { return dayCount_; }

    std::string getTimeString() const;
};
```

13.2 时间流逝逻辑

实现位置: [Classes/TimeManager.cpp](#)

```
void TimeManager::update(float delta) {
    // 1. 累加真实时间
    accumulator_ += delta * timeScale_;

    // 2. 每秒=游戏内10分钟
    const float gameMinutesPerSecond = 10.0f;

    if (accumulator_ >= 1.0f) {
        accumulator_ -= 1.0f;

        // 3. 推进游戏时间
        advanceTime(static_cast<int>(gameMinutesPerSecond));
    }
}

void TimeManager::advanceTime(int minutes) {
    // 1. 增加分钟
    currentMinute_ += minutes;

    // 2. 处理进位
    while (currentMinute_ >= 60) {
        currentMinute_ -= 60;
        currentHour_++;
    }

    // 3. 处理小时进位
    while (currentHour_ >= 24) {
        currentHour_ -= 24;
        onNewDay();
    }

    // 4. 检查强制睡觉时间 (0点)
    if (currentHour_ == 0 && currentMinute_ == 0) {
        forceSleep();
    }
}

void TimeManager::onNewDay() {
    dayCount_++;

    CCLOG("== NEW DAY %d ==", dayCount_);

    // 1. 通知农场管理器
    FarmManager::getInstance()->onNewDay();

    // 2. 更新天气
    WeatherManager::getInstance()->updateWeather();

    // 3. 恢复玩家状态
    Player::getInstance()->restoreEnergy();

    // 4. 刷新商店库存
    MarketManager::getInstance()->refreshStock();
```

```
// 5. 自动存档  
SaveManager::getInstance()->autoSave();  
}
```

时间流速：

真实1秒 = 游戏10分钟
真实6秒 = 游戏1小时
真实2.4分钟 = 游戏1天 (6:00 - 0:00)

总结

本技术文档详细讲解了星露谷守游戏的核心系统实现：

核心特性

1. 架构设计 - 分层架构、设计模式、继承关系
2. 场景系统 - 场景管理、地图加载、摄像机跟随
3. 农耕系统 - 耕地、种植、浇水、收获、作物生长
4. 矿洞系统 - 楼层生成、矿石分布、许愿池
5. 战斗系统 - 怪物AI、伤害计算、无敌帧
6. 背包系统 - 物品管理、UI交互、堆叠规则
7. 存档系统 - JSON序列化、文件I/O、自动存档
8. 技能系统 - 经验值、升级、加成效果
9. **NPC系统** - 对话、商人交易
10. 钓鱼系统 - 抛竿、小游戏、鱼类
11. 天气系统 - 天气生成、视觉效果、游戏影响
12. 时间系统 - 时间流逝、每日更新

技术亮点

- 使用Cocos2d-x 3.17游戏引擎
- C++11/14现代特性 (auto, lambda, nullptr等)
- 设计模式应用 (单例、观察者、工厂)
- 完整的存档系统 (JSON格式)
- 丰富的游戏机制 (农耕、战斗、钓鱼)
- TMX地图格式支持
- 粒子系统视觉效果

代码质量

-  详细的中文注释
-  清晰的代码结构
-  模块化设计
-  异常处理机制

-  完善的数据结构
-

文档版本: v1.0 最后更新: 2025-12-27 总页数: 约100页 代码行数: 约15,000行