一、如何通过游戏原型抽象类

1. **找名词**
2. **找动词和形容词**
3. **理关系**
4. **再分组**

1、找名词

目的：提炼出【类】

通过游戏原型，在游戏设计蓝图中，把所有出现的**具体事物**和**抽象概念**都列出来。这些名词是“类”的主要候选者。

在我们的游戏中：

> “一个**史莱姆角色**在**沙滩**上移动，它可以在水边**钓鱼**。钓鱼时会进入一个**QTE状态**。游戏开始时有一个**主菜单**，可以开始**游戏**或查看**帮助**。”

从这段描述中，我们可以找出以下名词：

- **具体事物**
  - 史莱姆（操作角色） → `Player` 类
  - 沙滩 → `Background` 类
- **抽象概念**
  - **QTE状态** → `QTEManager` 类 (管理QTE逻辑)
  - **主菜单** → `MenuState` 类 (主菜单界面”)
  - **帮助** → `HelpState` 类 (帮助界面)
  - **游戏** → `Game` 类

游戏设计的第一步，是分析游戏原型中的核心元素。游戏中的“名词”，无论是像“玩家”这样的实体，还是像“主菜单”这样的概念，都可以被直接映射为程序中的“类”。依此，可以逐个提炼、设计出 `Player` 类、`MenuState` 类等所有需要用到的类。

2、找动词和形容词

目的：明确【方法】和【属性】

确定了“有哪些事物”（类）之后，下一步是描述这些事物“能做什么”（方法）和“拥有什么特征”（属性）。

- **动词** → 对应类的**方法**，代表对象的行为。
- **形容词** → 对应类的**属性**，代表对象的状态和特征。

以 `Player` 类为例：

- **找到动词 (行为):**

- 角色可以**移动** → `Player::move(direction)`
- 角色可以**钓鱼** → `Player::startFishing()`
- 角色可以**播放动画** → `Player::updateAnimation()`

- **找到形容词/特征 (状态):**
  - 角色的**位置** → `sf::Vector2f m_position`
  - 角色的**移动速度** → `float m_speed`
  - 角色的**动画状态**（静止，移动还是钓鱼） → `Animation m_currentAnimation`
  - 角色钓到的**鱼的数量** → `Inventory` 类

通过这一步，明确类里应该有什么，能做什么。

以上可以设计出一个最简单的 Player.hpp:

```
#include <SFML/Graphics.hpp>
#include "Inventory.hpp" // Inventory 类，存储鱼的数量

class Player {
public:
    Player();
    void handleInput();                          // 处理输入
    void update(sf::Time deltaTime);             // 更新状态
    void draw(sf::RenderTarget& target);         // 绘制

    void move(const sf::Vector2f& direction);
    void startFishing();

private:
    void updateAnimation();
    sf::Sprite m_sprite;
    sf::Texture m_texture;
    sf::Vector2f m_position;
    float m_speed;
    PlayerState m_state;
    Inventory m_inventory;
};
```

3、理关系

目的：构建【类结构】

类间有三种关系：**组合、聚合与继承**。

我们在游戏设计中组织类间关系时，主要用到两种关系：**继承** 和 **组合**。

> ● **继承：** 表示一个类是另一个类的特殊化。例如，PlayState、MenuState、HelpState都“是一种”State。这是状态机模式的基础，通过一个共同的基类State指针，就可以管理所有不同的游戏状态。

> ● **组合：** 表示一个类包含另一个类的实例，并且两者的生命周期紧密绑定。如果容器对象被销毁，其包含的对象也会被销毁。例如，PlayState“拥有一个”Player对象。当PlayState结束时，该状态下的Player实例也随之消亡。这通常通过将成员对象直接声明在类中来实现：Player m_player;。
>
> ● **聚合：** 表示一个类使用另一个类的实例，但并不拥有它。两者的生命周期是独立的。例如，PlayState“有一个”指向ResourceManager的引用。PlayState需要使用资源管理器来加载纹理，但它不负责创建或销毁资源管理器。

4、再分组

目的： 创造【管理器类】进行优化

除了游戏的核心逻辑，还会有一些“公共服务”的需求，它们不属于任何一个特定的游戏对象。

- 比如：
  - 谁来加载图片和字体资源？如果`Player`和`Background`都自己加载，会造成重复和浪费。
  - 谁来播放背景音乐和音效?
  - 谁来负责存档和读档的复杂文件操作?
- **解决方案：** 创建专门的**管理器类**。
- 依据：**单一职责原则**。
- 例如：
  - 管理资源加载 → `ResourceManager` 类
  - 管理音频播放 → `AudioManager` 类
  - 管理存档读写 → `SaveManager` 类

`ResourceManager` 专门负责所有资源的加载和缓存，这样 `Player` 类就不需要关心文件读写的细节，只需向`ResourceManager` 请求自己需要的纹理即可。

二、示例教程的类的设计

（一）类的设计

1、核心框架类

（1） Game

- 创建并管理 `sf::RenderWindow`
- 驱动主循环（事件 → 更新 → 渲染）
- 持有 `StateMachine`、`ResourceManager`、`ServiceLocator`（用于音频管理）

（2） StateMachine

- 管理当前 `State` 实例

- 提供 `changeState()` 接口切换状态
- 转发事件、更新和绘制调用给当前状态

（3） State（抽象基类）

- 定义纯虚接口： `onEnter()` / `onExit()` （状态切换前后调用） `handleEvent(const sf::Event&)` `update(sf::Time)` `draw(sf::RenderTarget&)`

2、各个“界面”（状态）类

（1） SplashState（启动界面）

- 播放 Logo 或加载动画
- 资源初始化完成后，切换至 `MenuState`

（2） MenuState（主菜单界面）

- 渲染“开始游戏”、“退出”等选项
- 处理键盘/鼠标输入，切换至相应状态

（3） PlayState（主游戏界面）

- 持有并渲染 `Background`、`PaddlePlayer`、`PaddleAI`、`Ball`
- 控制玩家移动、AI逻辑、物理更新
- 检测球与挡板、边界碰撞，播放音效
- 管理游戏胜负逻辑与状态切换

3、管理器类

（1） ResourceManager

- 加载并缓存纹理、字体、音效等资源
- 提供 `getTexture()`、`getFont()` 等接口，避免重复加载

（2） ServiceLocator

- 提供全局访问音频接口（`IAudioProvider` 实现）
- 解耦音频调用与具体实现
- 通过 `getAudio()` 获取音频服务实例

4、游戏元素类

（1） Background

- 加载并绘制场景背景

- 支持窗口大小适应

（2） PaddlePlayer

- 加载玩家挡板精灵
- 处理输入、控制移动、碰撞逻辑

（3） PaddleAI

- AI控制的挡板逻辑
- 简单跟踪小球位置，实现对战效果

（4） Ball

- 小球对象，控制物理运动与反弹
- 与挡板、窗口边界检测碰撞
- 播放碰撞音效，更新游戏逻辑

5、音频系统接口

（1） IAudioProvider（接口）

- 定义音效、音乐播放标准接口
- 具体实现由外部注入，方便替换

（2） SFMLSoundProvider

- 使用 SFML 音效模块实现 `IAudioProvider`
- 播放背景音乐与音效，支持音量控制

（3） NullSoundProvider

- 空对象模式，禁用音频功能
- 保证即使不使用音效，游戏逻辑不受影响

（二）类之间的关系

1. 继承关系

- `State`（抽象基类） ： `SplashState`、`MenuState`、`PlayState` 均继承自 `State`，统一状态切换接口。
- `IAudioProvider`（接口） ： `SFMLSoundProvider`、`FModSoundProvider`、`NullSoundProvider` 实现该接口，提供多样化音频功能选择。

2. 组合关系

组合方	拥有的对象或模块	说明
`Game`	`StateMachine`	控制当前游戏状态
	`ResourceManager`	统一资源管理
	`ServiceLocator`	全局音频接口提供
`StateMachine`	当前 `State` 实例	具体界面状态（Splash/Menu/Play）
`PlayState`	`Background`、`PaddlePlayer`、`PaddleAI`、`Ball`	游戏内实体对象

3. 依赖关系

依赖方	被依赖方	具体用途
`Game`、`State`	`ServiceLocator::getAudio()`	获取音频接口，播放音乐、音效
各 `State`	`ResourceManager`	加载背景、字体、按钮、实体资源
`Ball`、`Paddle`	`Collision` 工具类	碰撞检测逻辑
`PlayState`	`ResourceManager`、`Collision`	渲染资源、物理交互处理
`MenuState`	`ResourceManager`、`ServiceLocator`	加载菜单按钮资源，切换背景音乐

4. 隐含设计关系

- `ServiceLocator` 实际上是**单例模式**或**全局访问点**，隐藏了具体 `IAudioProvider` 实现，弱化模块间耦合。
- `ResourceManager` 典型**缓存管理**，避免纹理、音效重复加载，提升性能。
- `Collision` 设计为**纯工具类**，不需要实例化，方便跨模块调用。

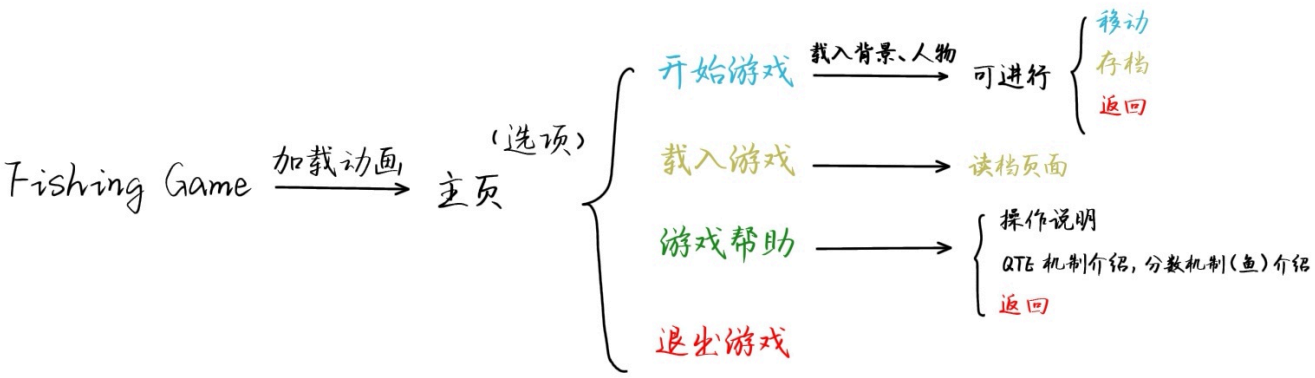
（三）整体结构图（树状逻辑）

```
Game
├── StateMachine
│   └── 当前State（Splash/Menu/Play）
│       └── PlayState
│           ├── Background
│           ├── PaddlePlayer
│           ├── PaddleAI
│           └── Ball
├── ResourceManager
├── ServiceLocator
│   └── IAudioProvider
│       ├── SFMLSoundProvider
│       └── NullSoundProvider
```

三、我们游戏的类的设计

根据我们游戏的功能框架，再加以补充、优化，可以设计出以下六大板块的类：

Fishlime 功能框架展示



按下可以切换连续钓鱼方式
① 按一次 space 钓一次鱼
② 按 space 进入钓鱼状态，再按一次 space 退出钓鱼状态

awsd 上下左右 移动 到水边 → 按 space 开始钓鱼 → 随机进行3次qte
全部成功：背包中鱼数量增加
失败一次：退出钓鱼，或重新钓鱼，由连续钓鱼方式决定

存档 → 在读档界面中选择一个方框，将目前背包中各种鱼的数量以及当前史莱姆所处的位置保存进去。

返回 → 通过按钮或 ESC 返回到菜单

音效、特效
- 背景音乐，不同页面不同背景音乐
- 环境音效：水声，风声，钓鱼中鱼翻腾声等
- 交互音效：QTE 成功或失败音效，成功钓鱼提示音
- 角色动画：史莱姆的待机、移动、钓鱼均有动画

Fishlime功能框架图

1. 核心框架类

这些类构成了整个游戏的骨架，驱动着程序的生命周期。

- **`Game`**
  - **分类:** 核心框架类
  - **职责:**
    - 创建并管理主窗口 (`sf::RenderWindow`)。
    - 持有并初始化所有全局管理器 (`StateMachine`, `ResourceManager` 等)。
    - 驱动游戏的主循环 (捕获事件、调用更新、执行渲染)。
- **`StateMachine`**
  - **分类:** 核心框架类
  - **职责:**
    - 使用**栈 (Stack)** 结构来管理一个或多个 `State` 对象。
    - 提供 `pushState`, `popState`, `changeState` 接口来控制游戏流程。
    - 将事件、更新、渲染的调用**委托**给栈顶的(或所有)状态。

2. 抽象基类

这些类是实现多态和框架扩展性的基石。它们定义了“契约”，但不提供具体实现。

- **`State`**
  - **分类:** 抽象基类
  - **职责:**
    - 定义所有游戏状态（界面）的通用接口，如 `handleEvent()`, `update()`, `draw()`。
- **`Item`**
  - **分类:** 抽象基类
  - **职责:**
    - 定义所有道具的通用接口和属性，如 `getName()`, `getDescription()`, `use()`。

3. 状态派生类

这些类继承自 `State`，是游戏中每一个具体界面的实现。

- **`SplashState`**
- **`MenuState`**
- **`PlayState`**
- **`HelpState`**
- **`LoadState`**
- **`SaveState`**

- **分类:** 具体派生类
- **职责:** (各自负责) 启动、主菜单、主游戏、帮助、读档、存档界面的所有逻辑、数据和渲染。

4. 游戏元素与组件类

这些是构成游戏世界的核心“实体”以及它们的功能“零件”。

- **`Player`**
  - **分类:** 游戏核心元素
  - **职责:**
    - 代表玩家角色，处理玩家的移动逻辑。
    - 持有 `Animator` 来管理自身动画。
    - 持有 `Inventory` 来管理道具。
- **`Background`**
  - **分类:** 游戏元素
  - **职责:**
    - 显示和管理游戏背景（可能是滚动的、多层的）。
- **`HUD`** (Heads-Up Display)
  - **分类:** 游戏元素/UI集合
  - **职责:**
    - 显示游戏中的实时信息，如分数、时间、QTE提示等。
- **`Animation`**
  - **分类:** 功能组件类
  - **职责:**
    - 封装**一条**动画序列的数据（帧、时长）。
- **`Animator`**
  - **分类:** 功能组件类
  - **职责:**
    - 管理一个对象所拥有的**所有** `Animation`，并提供 `play()` 接口来切换动画。

5. 道具系统类

这是一个完整的子系统，用于处理游戏中的所有道具。

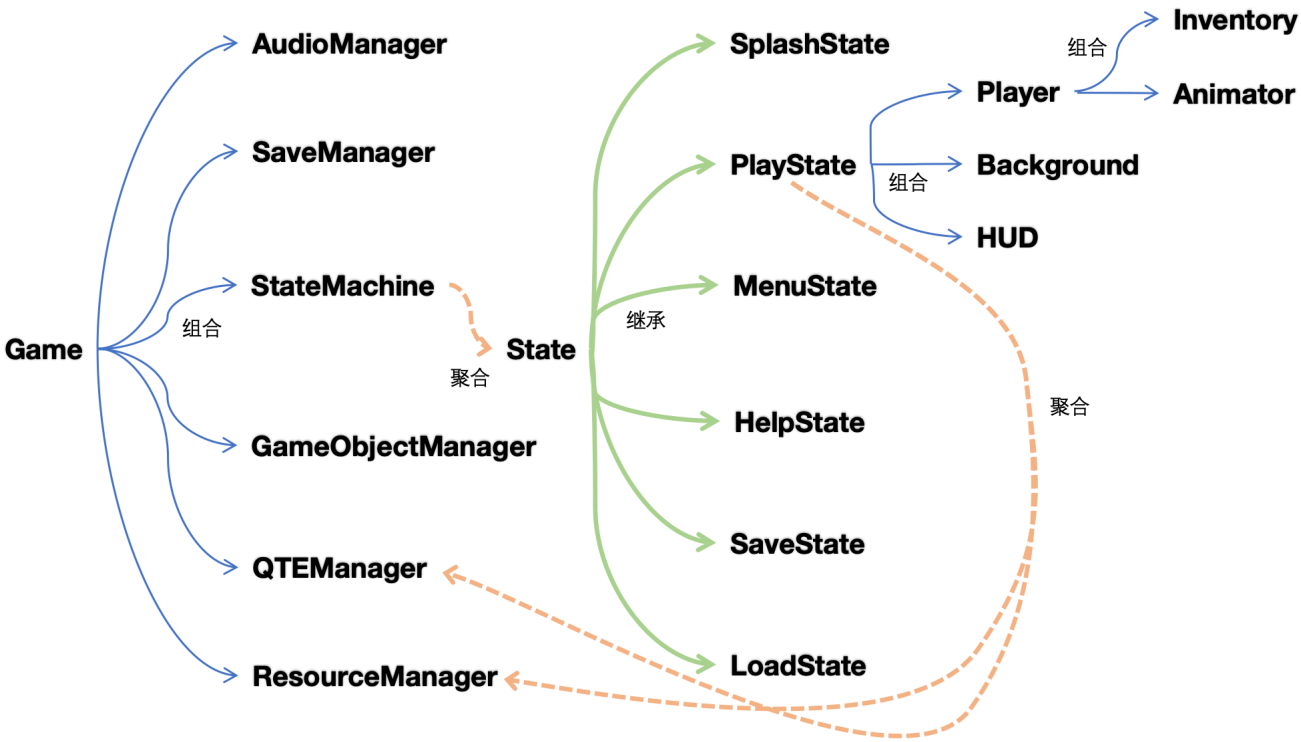
- **`Inventory`**
  - **分类:** 功能组件类
  - **职责:**

    - 作为 `Player` 的一个组件，负责存储和管理一个 `Item` 对象的集合。
- **`FishItem`**
    - **分类:** 道具派生类
    - **职责:**
        - 继承自 `Item`，代表“鱼”这种具体道具，拥有重量、稀有度等特有属性。

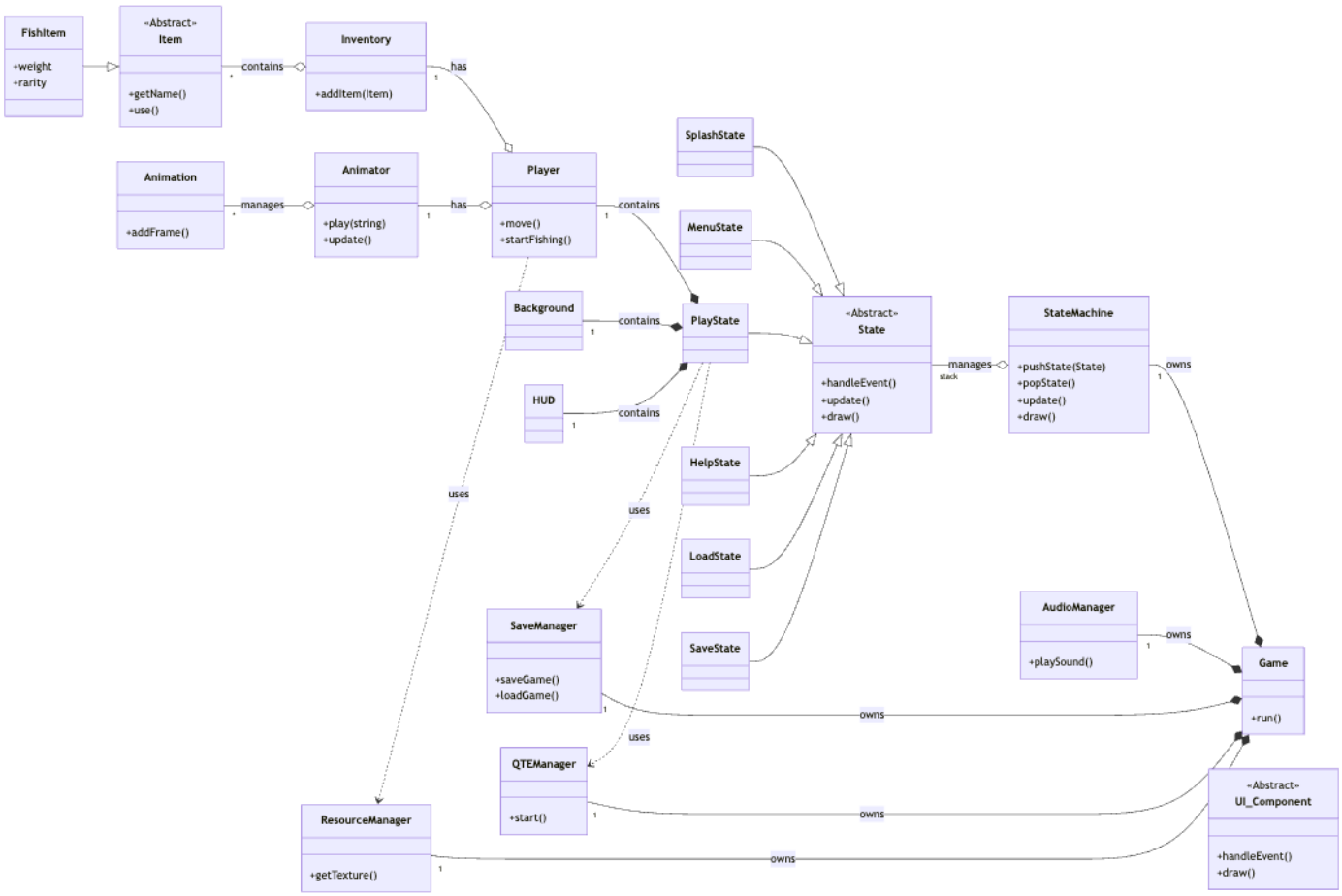
6. 管理器类

这些是全局性的“服务”类，遵循单一职责原则，为游戏的其他部分提供专门的支持。

- **`ResourceManager`**
- **`AudioManager`**
- **`SaveManager`**
- **`QTEManager`**
- **`GameObjectManager`**
    - **分类:** 管理器类
    - **职责:** (各自负责)
        - **`ResourceManager`**: 缓存并提供纹理、字体等资源。
        - **`AudioManager`**: 播放背景音乐和音效。
        - **`SaveManager`**: 处理游戏的存档与读档。
        - **`QTEManager`**: 封装和管理所有QTE玩法的复杂逻辑。
        - **`GameObjectManager`**: 统一创建、更新、渲染、销毁游戏对象。



类的架构图



类图

四、游戏设计的SOLID原则

SOLID分别是五个面向对象设计基本原则的首字母缩写：

- **单一职责原则(Single Responsibility Principle - SRP)**
- **开放-封闭原则 (Open-Closed Principle - OCP)**
- **里氏替换原则 (Liskov Substitution Principle - LSP)**
- **接口隔离原则 (Interface Segregation Principle - ISP)**
- **依赖倒置原则 (Dependency Inversion Principle - DIP)**

1. **单一职责原则**（SRP）

   定义：一个类应该只有一个引起它变化的原因。换言之，一个类应该只承担一项职责。

   遵循SRP，我们设计游戏时，我们将不同职责拆分到不同的类中，如：

● Player：核心职责是作为数据容器和状态协调者。它存储玩家的位置、速度、当前状态（如moving，fishing），并持有所需组件的引用。

● InputHandler：唯一职责是监听原始输入设备（键盘、鼠标），并将其转换为游戏内的抽象指令（如move等）。

● Animator：唯一职责是管理动画数据（帧、时长）并根据指令播放特定动画。

● Inventory：唯一职责是管理玩家获得的物品（如鱼）。

通过这种分解，Player类会非常稳定。可以修改InputHandler来调整键位，而无需触碰Player的核心逻辑。可以调整AnimatedSprite的动画数据，同样不影响Player其他部分代码。

2. **开放-封闭原则** （OCP）

   定义：软件实体（类、模块、函数等）应该对扩展开放，对修改封闭 。

该原则的核心在于通过抽象来应对变化。我们游戏的核心玩法是QTE。如果在QTEManager中使用一个巨大的switch语句来处理不同类型的QTE，那么每当要添加一种新的QTE玩法时，都必须修改这个switch语句，这违反了OCP。

根据OCP原则进行优化：

1. 定义一个抽象的IQteStrategy接口，包含update()、handleInput()、getStatus()等方法。
2. 为每种QTE玩法创建一个具体的策略类，如ProgressBarStrategy、RotatingPointerStrategy，它们都实现IQteStrategy接口。
3. QTEManager持有一个指向当前IQteStrategy的指针。当需要开始一个QTE时，只需实例化一个具体的策略对象并交给QTEManager即可。

如此一来，当需要添加新的QTE玩法时，只需创建一个新的策略类，而QTEManager的代码完全不需要改动，完美实现了“对扩展开放，对修改封闭”。

3. **里氏替换原则** （LSP）

定义：所有引用基类的地方必须能透明地使用其子类的对象，而不会引发错误。简单来说，子类必须能够完全替代其父类。

此原则主要约束继承关系。在StateMachine的设计中，StateMachine持有一个基类指针State* m_currentState，它通过这个指针调用update()、draw()等虚函数。LSP要求任何State的子类（MenuState, PlayState等）都必须忠实地实现基类的所有接口，并且其行为符合基类的约定。例如，如果某个子类的update()方法导致了游戏循环的崩溃或意外行为，那么它就违反了LSP。此原则确保了多态能够安全、正确地工作。

4. **接口隔离原则**（ISP）

定义：客户端不应该被强迫依赖于它们不使用的方法。即，应使用多个小的、专门的接口，而不是一个大的、臃肿的接口。

假设我们有一个通用的IGameObject接口，包含了update(), draw(), handleInput(), onCollision()等所有可能的方法。对于一个纯粹的背景装饰物（如云），它只需要被draw()，但却被强制要求实现handleInput()和onCollision()等空方法。

根据ISP原则进行优化：

将大接口拆分为多个小接口：

- IUpdateable { virtual void update(sf::Time dt) = 0; }
- IDrawable { virtual void draw(sf::RenderTarget& target) = 0; }
- IInputHandler { virtual void handleInput(...) = 0; }

一个Player类可以同时实现这三个接口，而背景中的云朵类只需实现IDrawable接口。这使得类的设计更加清晰和精炼。

5. **依赖倒置原则**（DIP）

定义：高层模块不应该依赖于低层模块，两者都应该依赖于抽象。抽象不应该依赖于细节，细节应该依赖于抽象 。

高层模块（如PlayState）负责游戏逻辑，低层模块（如ResourceManager）负责具体实现（如从磁盘加载文件）。一个常见的错误是让PlayState直接#include "ResourceManager.h"并调用其实例。这造成了高层对低层的直接依赖。如果ResourceManager的实现方式改变（例如，从本地文件加载改为从网络加载），PlayState也可能需要修改。

优化方式：依赖注入

DIP的实现核心是“倒置”依赖关系。

1. **定义抽象**：创建一个IResourceManager接口，其中声明loadTexture()等纯虚函数。
2. **依赖抽象**：PlayState依赖于IResourceManager接口，而不是具体的ResourceManager类。
3. **实现细节**：具体的ResourceManager类实现IResourceManager接口。
4. **注入依赖**：PlayState不自己创建ResourceManager实例。而是在其构造函数中接收一个IResourceManager&的引用。这个实例由更高层的模块（如Game类）创建，并在创建PlayState时“注入”进去。

这种方法彻底解除了PlayState和ResourceManager之间的耦合，极大地提高了代码的模块化程度和可测试性。我们可以轻松地为PlayState提供一个“模拟的”资源管理器来进行单元测试，而无需真正地读写磁盘。这是避免使用全局单例（Singleton）模式（一种常见的反模式）的现代、专业方法。

五、如何管理类、类之间的关系

理想结构：高内聚，低耦合

设计思想：分层委托、集中管理

1. 以核心流程管理机制为例：`Game` -> `StateMachine` -> `State`

（1）`Game` 类："顶层管理"

- **如何管理类**
  - `Game` 类是我们整个应用的顶层管理者（CEO），它的管理职责非常明确：它**创建并拥有**所有全局性的、唯一的"部门经理"，比如`StateMachine`（流程总监）、`ResourceManager`（物资部经理）和`AudioManager`（宣传部经理）。它负责这些核心模块的整个生命周期。
- **类之间的关系**
  - `Game` 类与 `StateMachine` 等管理器的关系是组合。这意味着 `StateMachine` 是 `Game` 的固有组成部分，`Game` 对象被创建时，`StateMachine` 也被创建。这种强所属关系确保了核心模块的稳定存在。

（2） `StateMachine` 类："运营总监"

- **如何管理类**
  - `StateMachine` 是负责具体"项目流程"的"运营总监"。它不执行具体业务，但它**集中管理**着所有具体业务的"项目经理"——也就是我们的`State`派生类（如`PlayState`, `MenuState`）。它通过一个栈结构来管理这些状态，决定了哪个状态应该被创建、哪个应该被销毁、哪个是当前活跃的状态，方便处理游戏暂停、菜单切换等流程。
- **类之间的关系**
  - `StateMachine`与`State`基类的关系是聚合和依赖。它持有一个`State`的指针集合，并依赖`State`定义的统一接口（如`update`, `draw`）来下达指令。
  - 它通过这个统一接口与所有`State`的派生类进行交互，而无需知道当前到底是`PlayState`还是`HelpState`。这正是面向对象中多态的强大之处，也是我们架构解耦的关键。

（3） `State` 派生类 (如`PlayState`)：具体的"项目经理"

- **如何管理类**
  - 每一个`State`派生类，比如`PlayState`，就是一位具体的"项目经理"。它负责管理自己项目内部的所有"团队成员"，也就是这个场景所特有的游戏元素。例如，`PlayState`会创建并管理它自己的`Player`对象、`Background`对象和`HUD`对象。
- **类之间的关系**
  1. `PlayState`与`State`的关系是继承。`PlayState`是一种`State`，这使得它可以被`StateMachine`统一管理。
  2. `PlayState`与`Player`、`Background`等的关系是组合。`Player`是`PlayState`这个项目中的一员，当`PlayState`这个项目结束时，这个`Player`实例也随之结束。

六、优化设计方法

为了确保我们的代码在未来功能增加时，不会变得混乱和难以维护，应该优化设计方法，以使得：

1. **代码清晰，各司其职：** 每个类的功能都一目了然，可读性极高。
2. **维护轻松，精准定位：** 想修改存档逻辑？我们只需要去`SaveManager`。想改QTE难度？我们只需要去`QTEManager`。定位问题非常快。
3. **高度复用，方便协作：** `ResourceManager`这样的类可以被轻松地拿到下一个游戏里复用。团队也可以分工，不同成员负责不同的管理器，并行开发，互不干扰。

如前所述，**单一职责原则**是面向对象设计的一项重要基本原则。基于这个原则可以提出优化方法：

通过单一职责原则，可以概括出四个字：各司其职。

原则定义： 一个类，应该只有一个引起它变化的原因。通俗讲，一个类只做一件事，并把它做好。

例如：`ResourceManager`等管理器类的设计思想：

- **`ResourceManager`**
  - **唯一职责：** 加载、缓存和提供资源（纹理、字体等）。
  - `Player`类和`Background`类都不需要关心图片文件在哪、如何从硬盘读取。它们只需要跟`ResourceManager`说：“我要‘player.png’这个纹理”，然后就能拿到。如果未来想优化资源加载方式（比如从单个文件改为打包文件），只需要修改`ResourceManager`这一个类，其他所有使用资源的地方都不需要改动。这一个重要思想叫做**解耦**。
- **`QTEManager`**
  - **唯一职责：** 封装所有关于QTE玩法的复杂逻辑。
  - `PlayState` 作为“总管”，不需要知道QTE内部的判定方式。它只需要在钓鱼时对`QTEManager`说：“开始QTE！”，然后等待`QTEManager`告诉它一个结果：“成功了”或“失败了”。这让`PlayState`的代码保持简洁，只关注于“何时触发QTE”和“QTE结束了怎么办”这两个高层逻辑。
- **`AudioManager`** 、 **`SaveManager`等** 同理。

七、问题清单

1. 游戏的主循环应该如何设计?（基于消息（事件）的程序运行机制）

在基于消息（事件）驱动的程序架构下，游戏主循环主要由三个连续、反复执行的阶段组成：事件处理、逻辑更新和图形渲染。

1. **事件处理**
   - 不断从操作系统或窗口管理器获取用户输入、窗口消息（如键盘、鼠标、窗口关闭、大小改变等）。
   - 将每条消息分发给相应的子系统，保证所有交互都能被及时响应且不会遗漏。
   - 在此阶段，不会执行游戏世界的物理或动画更新，只做状态收集和命令派发。

2. **游戏逻辑更新**
    - 根据上一步收集到的输入状态以及游戏自身状态，推进游戏世界的发展。
    - 包括角色运动、碰撞检测、AI 决策、动画进度、物理模拟等。
    - 通常要计算自上次更新以来的时间增量（delta time，代码中常以dt代替），用于驱动角色移动和动画播放。
3. **图形渲染**
    - 清除上一帧的画面缓存，然后按照一定的层次顺序绘制场景中的精灵、背景、UI 等。
    - 渲染操作只负责“画出当前状态”，绝不在此阶段修改游戏逻辑或物理数据。
    - 最后将渲染结果推送到屏幕（通常是一帧显示操作），完成一次完整的画面刷新。

结合以上三个阶段，可以得到一个典型的主循环结构：

> **循环直到窗口关闭**
>
> 1. 拉取并分发所有事件
> 2. 根据时间增量更新游戏逻辑
> 3. 清屏→绘制→交换缓冲→显示

更进一步的，为了确保游戏逻辑的稳定性和渲染的流畅性，需要设计一个良好的主循环，可以使用以下方式来优化一个主循环：

1. 架构游戏循环：为稳定逻辑设计的固定时间步长

以下是一个基础的SFML主循环：

```
while (window.isOpen()) {
  //... poll events...
  window.clear();
  //... draw something...
  window.display();
}
```

这个循环存在一个典型问题：它的执行速度完全依赖于计算机的性能。在快的机器上，循环每秒执行上千次，游戏逻辑（如玩家移动）会快得无法控制；在慢的机器上则相反。这对于需要精确物理和动画同步的游戏是致命的。

解决方案：固定时间步长（Fixed Timestep）循环

专业的游戏循环会将**逻辑更新（Update）与渲染（Render）**分离。逻辑更新以一个固定的频率执行（例如，每秒60次），而渲染则可以尽可能快地执行。这确保了无论帧率如何波动，游戏世界的行为都是一致和可预测的。

示例：

```
void Game::run() {
  sf::Clock clock;
  sf::Time timeSinceLastUpdate = sf::Time::Zero;
  const sf::Time TimePerFrame = sf::seconds(1.f / 60.f); // 逻辑更新频率为60Hz
```

```
5
6     while (m_window.isOpen()) {
7       sf::Time elapsedTime = clock.restart();
8       timeSinceLastUpdate += elapsedTime;
9
10      while (timeSinceLastUpdate > TimePerFrame) {
11        timeSinceLastUpdate -= TimePerFrame;
12        processEvents(); // 处理输入事件
13        update(TimePerFrame); // 以固定的时间步长更新逻辑
14      }
15
16      render(); // 渲染画面
17    }
18  }
```

在这个循环中，update(TimePerFrame)会以恒定的$\frac{1}{60}$秒为间隔被调用，保证了物理和游戏逻辑的稳定性。如果渲染速度过慢，update会连续执行多次以“追赶”逝去的时间，保证游戏逻辑不掉队。

2. 实现状态模式

这是通过状态设计模式，可以使用状态机来管理不同游戏界面（如主菜单、游戏、帮助界面）。该模式包含三个核心角色：

• **Context（上下文）**：在我们的项目中，是StateMachine类。它维护一个指向当前状态的引用，并将所有与状态相关的请求委托给当前状态对象处理。

• **State（状态接口）**：一个抽象基类，定义了所有具体状态都必须实现的接口，如handleEvent(), update(), draw()。

• **ConcreteState（具体状态）**：实现State接口的具体类，如MenuState, PlayState。每个类封装了特定状态下的行为。

3. 高级状态管理：状态栈

一个简单的状态机在切换状态时（例如从PlayState切换到MenuState），通常会销毁旧状态并创建新状态。这意味着当从游戏回到菜单时，PlayState的所有信息（玩家位置、玩家状态、道具情况等）都丢失了，再次回到游戏只能通过读档，或者开始新游戏，无法使用“继续游戏”回到刚刚正在进行的游戏。

一个更强大、更灵活的解决方案是使用**状态栈（State Stack）**。

StateMachine不再只持有一个当前状态，而是持有一个状态的栈（例如std::vector< std::unique_ptr< State > >）。

• **Push**: 当需要进入一个新状态时，不是替换当前状态，而是将新状态push到栈顶。

• **Pop**: 当需要离开当前状态时，将栈顶的状态pop。

• **Update/Draw**: StateMachine只对栈顶的状态进行更新和绘制。但也可以选择性地绘制栈中下层的状态，以实现菜单覆盖在游戏画面之上的效果。

通过状态栈，还可以实现游戏进程的暂停等，暂停后可以回到上次游戏状态，而不是重新开始一轮新的游戏。

2. 键盘和鼠标事件如何处理？（用户输入）

键盘和鼠标的输入通常分为两种处理方式：事件式处理和实时状态查询。

（1）事件式处理

在 `while (window.pollEvent(event))` 循环里接收并分发系统发来的每一条输入消息。

- **键盘事件**
  - 当某个按键被按下时触发，既可以检测当前按下（Press）的是什么按键，也可以检查是否伴有修饰键。
  - 同时，还可以检测是否有按键释放（Released）。
- **鼠标事件**
  - 可以检测鼠标按键按下（左/右/中键），并且给出位置。
  - 可以检测鼠标按键释放。
  - 可检测鼠标在窗口内的移动情况，并给出当前位置。
  - 可以检测滚轮滚动（滚动增量，垂直或水平滚轮）。

优点：

- **无丢失**：每次按下/释放、移动、滚轮都有对应事件。
- **低开销**：只在状态改变时产生消息。
- **适合触发型操作**：按下F进入钓鱼状态，qte时快速输入以进行判定等。

（2）实时状态查询

在 Update 阶段，通过 SFML 提供的静态函数可以直接查询设备当前状态：

- **方法**
  - `sf::Keyboard::isKeyPressed(sf::Keyboard::Key::keyCode)` 可以检测键盘某个按键是否按下，SFML还有检测某个按钮是否被按下，鼠标指针当前在哪个位置的函数等等。

优点：

- **适合持续动作**：角色移动、镜头平移等需要“持续按住”的场景。
- **代码简洁**：在每帧直接检测，不用管理按下/抬起事件的时间点。

以上两种方法的使用，可以参考以下方法：

1. 离散触发交给事件式，例如“按下空格键开始跳跃”、“单击按钮”；
2. 持续检测交给实时查询，例如“按住方向键持续移动”、“按住鼠标右键拖动视角”；
3. 统一分发：在事件循环里把按下/抬起放入一个输入管理器，在更新阶段读取管理器里的状态，让游戏逻辑层面只关心“当前哪些按键被按下”或“是否在这一帧发生了点击”；

3. 如何显示对象？（刷新机制）

基于 SFML 的渲染管线，“显示对象”时会把画的所有精灵、形状、文字等一次性地提交给渲染窗口，然后由底层完成真正的像素输出。该过程的核心是**双缓冲+刷新**机制，可以归纳为以下三步：

1. **清屏（Clear）**
    - 在每一帧开始时，调用 `window.clear()`（或带颜色参数的 `window.clear(sf::Color::Black)`）把“后备缓冲区”清空，准备好干净的画布。
    - 这一步只操作内存中的一块画布（还没有draw和display），不会马上影响到屏幕。
2. **提交绘制命令（Draw）**
    - 把所有要显示的对象依次通过 `window.draw(obj)` 提交给后备缓冲区。
    - `obj` 可以是精灵，可以是形状等等。
    - 每次 `draw` 调用只是把顶点、纹理和变换矩阵等信息写入渲染队列，不立即绘制到屏幕上。
3. **交换并呈现（Display / Swap Buffers）**
    - 调用 `window.display()`，底层会把“后备缓冲区”与“前端缓冲区”做一次交换。
    - 前端缓冲区即当前屏幕所见，交换后你刚刚在后备缓冲区绘制的内容就被显示出来；后备缓冲区则成为新的画布，用于下一帧的清屏和绘制。

4. 对象如何管理？（游戏对象管理类）

在游戏设计中，所有游戏对象通常使用一个游戏对象管理类（`GameObjectManager`）来统一管理，他可以让游戏对象的创建、更新、渲染以及销毁都在 `GameObjectManager` 内部自动进行，主循环与具体对象逻辑完全分离，使得代码清晰又易扩展。

但同样的，在小型游戏的设计中，也可以选择每个 State 手动管理对象，在每个具体的 `State`（如 `PlayState`）中，把需要的对象作为成员或一个容器来维护，使得代码更直观。

若采用 `GameObjectManager` 统一维护，可以遵循以下设计要点：

（1）抽象基类（GameObject）

- 定义统一接口

```
class GameObject {
public:
    virtual ~GameObject() = default;
    virtual void update(sf::Time dt) = 0;
    virtual void draw(sf::RenderTarget& target) const = 0;
};
```

- 所有具体对象（如 `Background`、`Player`）都继承自它，保证管理类能够用多态方式批量调用 。

（2）存储与生命周期管理

- 容器：内部用 `std::vector<std::unique_ptr<GameObject>>` 保存所有活跃对象。
- 智能指针：通过 `unique_ptr` 或 `shared_ptr` 管理对象生命周期，避免手动 `new/delete` 漏网。
- 延迟删除：如果在遍历中需要删除对象，可先打一个标记，遍历完再统一擦除，以免迭代器失效。

（3）添加与移除接口

```cpp
class GameObjectManager {
public:
    void add(std::unique_ptr<GameObject> obj);
    void remove(const GameObject* obj);    // 标记删除
    void clear();                          // 一次性清除所有
    // 其他
};
```

- **add()**：游戏启动或状态切换时注册新对象。
- **remove()**：对象销毁。
- **clear()**：通常在状态退出（onExit）时调用，确保旧状态的对象不残留。

（4）批量更新与绘制

在主循环或对应的 State 中，仅需两行：

```cpp
manager.updateAll(deltaTime);    // 统一调用所有对象的 update()
manager.drawAll(window);         // 统一调用所有对象的 draw()
```

- `updateAll` 内部遍历容器，按加入顺序依次调用每个对象的 `update(dt)`；
- `drawAll` 则在 `window.clear()` 与 `window.display()` 之间，按顺序调用每个对象的 `draw(target)`。
  这样就把“事件—更新—渲染”的主循环与具体对象逻辑解耦了 。

（5）与状态机（StateMachine）结合

- 每个 `State`（如 `PlayState`）持有一个自己的 `GameObjectManager` 实例：

```cpp
class PlayState : public State {
    GameObjectManager objects;
    // …
    void onEnter() override {
        objects.add(std::make_unique<Background>(…));
        objects.add(std::make_unique<Player>(…));
        // …
    }
    void handleEvent(const sf::Event& ev) override { /* 可以转发给对象 */ }
    void update(sf::Time dt) override { objects.updateAll(dt); }
    void draw(sf::RenderTarget& rt) override { objects.drawAll(rt); }
};
```

- 状态切换时，旧 State 调用 `objects.clear()`，新 State 在 `onEnter()` 再注册自己需要的对象。

5. 类的设计步骤，和基本设计原则（方法、属性和关系）

（1）类的设计步骤

在“一、如何通过游戏原型抽象类”章节中，我们已经详细讨论过这个问题，这里仅对设计流程做一个总结，具体设计方法请阅读上文。

1. **需求分析**
    - 明确系统功能、用例和场景，理清“谁要做什么”。
    - 列出系统中的主要实体（概念、组件、模块）。
2. **职责划分**
    - 将系统拆分成多个相对独立的 “职责单元”。
    - 每个职责单元对应一个或一组类。
3. **定义类接口**
    - 为每个类确定对外可见的方法（`public` / `protected` 接口）。
    - 保持接口最小化，只暴露必要的操作。
4. **确定属性与状态**
    - 罗列每个类需要保存的核心数据（成员变量）。
    - 注意用合适的访问权限（一般私有 `private`，必要时提供 `getter/setter`）。
5. **设计类之间关系**
    - 确定继承、组合、聚合、依赖等关系。
    - 优先组合/聚合，谨慎使用继承；保持类的复用与可扩展。
6. **细化方法实现**
    - 编写方法内部逻辑，遵循单一职责，方法粒度适中。
    - 方法要小、专一，可单元测试。
7. **绘制类图**
    - UML 类图或类似草图，用于可视化展示属性、方法及依赖关系。
    - 示例可在“三、我们游戏的类的设计”章节末尾查看。
8. **迭代重构**
    - 随着需求演进，不断重构，保持高内聚低耦合。
    - 及时抽象公共部分，拆分臃肿类。

（2）基本设计原则

在“四、游戏设计的SOLID原则”章节中，我们也已经详细讨论过这个问题，接下来仅对几个原则进行简要介绍，并且引申出一些其他的，可能用到、但没那么重要的原则。

1. 单一职责原则（SRP）

> 一个类只做一件事、且把这件事做到极致。

- 方法：类内方法都围绕同一职责，不要“绑架”多余功能。
- 属性：只保存与该职责高度相关的数据。

2. 开放-封闭原则（OCP）

> 对扩展开放，对修改关闭。

- 方法：新增行为时，通过继承或组合扩展；避免直接修改已有代码。
- 属性：为可能变化的数据留好扩展点（如使用 `enum`、策略模式等）。

3. 里氏替换原则（LSP）

> 子类必须能够替换它们的父类。

- 继承关系：子类重写方法时不要改变父类语义；方法的前置条件不能比父类更严格，后置条件不能更松散。

4. 接口隔离原则（ISP）

> 不强迫客户端依赖它不需要的接口。

- 方法：将大接口拆分成多个小接口；让类只实现它关心的那一部分。
- 属性：仅包含接口所需成员，避免“胖”接口。

5. 依赖倒置原则（DIP）

> 高层模块不依赖低层模块，二者都应依赖抽象；抽象不依赖细节，细节依赖抽象。

- 关系：通过接口或抽象类来约束依赖，使用依赖注入（构造函数、Setter）降低耦合。

6. 合成复用原则（CARP）

> 优先使用组合/聚合，而不是继承。

- 关系：把可复用功能封装成独立组件，通过成员变量持有并调用。

7. 最小知识原则（Law of Demeter）

> 一个对象应当对其他对象有最少的了解。

- 关系：对象只调用自己直接持有的对象的方法，避免链式调用 `a->b()->c()->…`。

8. 高内聚低耦合

- 高内聚：类内部的属性和方法紧密相关，围绕一个核心职责。
- 低耦合：类与类之间依赖松散，通过抽象和接口进行交互。