C++ 面向对象编程

从三大特性到设计模式

核心内容

- 封装：数据隐藏与访问控制
- 继承：代码复用与类层次设计
- 多态：虚函数与动态绑定
- 构造与析构机制
- 抽象类与接口设计
- 组合 vs 继承
- 面向对象设计原则
- 实战：游戏开发场景应用

理论与实践 · 设计思想 · 场景应用

作者：Aweo

2025 年 10 月

Contents

1 C++ 面向对象编程 3
1.1 封装（Encapsulation） 3
1.1.1 什么是封装？ 3
1.1.2 访问控制 3
1.1.3 游戏场景：物品系统 4
1.1.4 封装的最佳实践 6
1.2 继承（Inheritance） 7
1.2.1 什么是继承？ 7
1.2.2 游戏场景：角色系统 7
1.2.3 继承中的构造和析构 10
1.2.4 多重继承（谨慎使用） 11
1.3 多态（Polymorphism） 12
1.3.1 什么是多态？ 12
1.3.2 虚函数机制 12
1.3.3 虚函数表（vtable）原理 14
1.3.4 游戏场景：AI 系统 15
1.4 抽象类与接口 18
1.4.1 抽象类 18
1.4.2 接口设计模式 20
1.5 组合 vs 继承 21
1.5.1 何时使用继承？ 21
1.5.2 何时使用组合？ 22
1.5.3 游戏场景：组件系统（推荐） 22
1.6 面向对象设计原则 25
1.6.1 SOLID 原则 25
1.7 总结与最佳实践 29

1 C++ 面向对象编程

C++的三大特性：封装、继承、多态，是面向对象编程的核心。

1.1 封装（Encapsulation）

1.1.1 什么是封装？

定义：将数据和操作数据的方法绑定在一起，隐藏对象的内部实现细节，只暴露必要的接口。

核心目的：

1. 数据保护：防止外部直接访问和修改内部数据
2. 降低耦合：外部代码不依赖内部实现
3. 提高可维护性：修改内部实现不影响外部代码

1.1.2 访问控制

```C++
class Player {
private:
    // 私有成员：只能在类内部访问
    int health;
    int maxHealth;
    float posX, posY;

protected:
    // 保护成员：类内部和派生类可访问
    int level;
    float experience;

public:
    // 公有成员：任何地方都可访问
    Player(int maxHp) : health(maxHp), maxHealth(maxHp),
                        posX(0), posY(0), level(1), experience(0) {}

    // 公有接口：提供安全的数据访问
    int getHealth() const { return health; }

    void takeDamage(int damage) {
        health -= damage;
        if (health < 0) health = 0;  // 确保健康值不为负
    }

    void heal(int amount) {
        health += amount;
        if (health > maxHealth) health = maxHealth;  // 不超过最大值
    }

    void moveTo(float x, float y) {
        posX = x;
        posY = y;
    }

    void getPosition(float& x, float& y) const {
        x = posX;
        y = posY;
    }
};

```

```
// 使用
Player player(100);
// player.health = -50;  // ✗ 错误！health是私有的
player.takeDamage(30);   // ✓ 通过公有接口安全地修改
std::cout << player.getHealth();  // 70
```

1.1.3 游戏场景：物品系统

C++

```
// 物品基类
class Item {
private:
    int itemId;
    std::string name;
    int stackSize;      // 当前堆叠数量
    int maxStackSize;   // 最大堆叠数量

public:
    Item(int id, const std::string& name, int maxStack = 1)
        : itemId(id), name(name), stackSize(1), maxStackSize(maxStack) {}

    // 尝试添加物品到堆叠
    bool addToStack(int amount) {
        if (stackSize + amount <= maxStackSize) {
            stackSize += amount;
            return true;
        }
        return false;
    }

    // 从堆叠中移除
    bool removeFromStack(int amount) {
        if (stackSize >= amount) {
            stackSize -= amount;
            return true;
        }
        return false;
    }

    // 只读访问
    int getId() const { return itemId; }
    const std::string& getName() const { return name; }
    int getStackSize() const { return stackSize; }
    int getMaxStackSize() const { return maxStackSize; }

    // 虚函数：子类可重写
    virtual void use() = 0;
    virtual ~Item() = default;
};

// 消耗品
class Consumable : public Item {
private:
    int healAmount;

public:
```

```
48      Consumable(int id, const std::string& name, int heal)
49          : Item(id, name, 99), healAmount(heal) {}  // 消耗品最多堆叠99个
50
51      void use() override {
52          std::cout << "使用 " << getName() << "，恢复 " << healAmount << " 生命值\n";
53          removeFromStack(1);
54      }
55  };
56
57  // 装备
58  class Equipment : public Item {
59  private:
60      int attackBonus;
61      int defenseBonus;
62      bool equipped;
63
64  public:
65      Equipment(int id, const std::string& name, int atk, int def)
66          : Item(id, name, 1),  // 装备不可堆叠
67            attackBonus(atk), defenseBonus(def), equipped(false) {}
68
69      void use() override {
70          equipped = !equipped;
71          std::cout << (equipped ? "装备了 " : "卸下了 ") << getName() << "\n";
72      }
73
74      bool isEquipped() const { return equipped; }
75      int getAttackBonus() const { return attackBonus; }
76      int getDefenseBonus() const { return defenseBonus; }
77  };
78
79  // 背包系统
80  class Inventory {
81  private:
82      std::vector<std::unique_ptr<Item>> items;
83      int maxSlots;
84
85  public:
86      Inventory(int slots = 20) : maxSlots(slots) {}
87
88      bool addItem(std::unique_ptr<Item> item) {
89          // 先尝试堆叠到已有物品
90          for (auto& existingItem : items) {
91              if (existingItem->getId() == item->getId()) {
92                  if (existingItem->addToStack(item->getStackSize())) {
93                      return true;
94                  }
95              }
96          }
97
98          // 无法堆叠，检查是否有空位
99          if (items.size() < maxSlots) {
100             items.push_back(std::move(item));
101             return true;
```

```
        }

        return false;  // 背包已满
    }

    void listItems() const {
        std::cout << "背包物品 (" << items.size() << "/" << maxSlots << "):\n";
        for (const auto& item : items) {
            std::cout << "- " << item->getName();
            if (item->getMaxStackSize() > 1) {
                std::cout << " x" << item->getStackSize();
            }
            std::cout << "\n";
        }
    }
};
```

1.1.4 封装的最佳实践

C++

```
// ✓ 好的封装：不变性保证
class Rectangle {
private:
    double width;
    double height;

public:
    Rectangle(double w, double h) : width(w), height(h) {
        if (w <= 0 || h <= 0) {
            throw std::invalid_argument("宽和高必须为正数");
        }
    }

    void setWidth(double w) {
        if (w <= 0) throw std::invalid_argument("宽必须为正数");
        width = w;
    }

    void setHeight(double h) {
        if (h <= 0) throw std::invalid_argument("高必须为正数");
        height = h;
    }

    double getArea() const { return width * height; }
};

// ✗ 差的封装：暴露内部数据
class BadRectangle {
public:
    double width, height;  // 直接公开，无法保证数据有效性
};

BadRectangle rect;
rect.width = -5;  // 可以设置非法值！
```

1.2 继承（Inheritance）

1.2.1 什么是继承？

定义：一个类（派生类）可以继承另一个类（基类）的成员，实现代码复用和类层次结构。

继承类型：

- public 继承：is-a 关系（派生类是基类的一种）
- protected 继承：受保护实现
- private 继承：实现继承（不推荐，建议用组合）

1.2.2 游戏场景：角色系统

```cpp
// 基类：游戏实体
class GameObject {
protected:
    int id;
    std::string name;
    float x, y;
    bool active;

public:
    GameObject(int id, const std::string& name)
        : id(id), name(name), x(0), y(0), active(true) {}

    virtual ~GameObject() = default;

    // 每帧更新
    virtual void update(float deltaTime) = 0;

    // 渲染
    virtual void render() const = 0;

    // 通用功能
    void setPosition(float newX, float newY) {
        x = newX;
        y = newY;
    }

    void setActive(bool isActive) { active = isActive; }
    bool isActive() const { return active; }

    int getId() const { return id; }
    const std::string& getName() const { return name; }
};

// 派生类：可移动实体
class MovableObject : public GameObject {
protected:
    float velocityX, velocityY;
    float speed;

public:
    MovableObject(int id, const std::string& name, float speed)
        : GameObject(id, name), velocityX(0), velocityY(0), speed(speed) {}

    void setVelocity(float vx, float vy) {
```

```
45          velocityX = vx;
46          velocityY = vy;
47      }
48
49      void update(float deltaTime) override {
50          if (active) {
51              x += velocityX * speed * deltaTime;
52              y += velocityY * speed * deltaTime;
53          }
54      }
55  };
56
57  // 战斗实体
58  class CombatEntity : public MovableObject {
59  protected:
60      int health;
61      int maxHealth;
62      int attackPower;
63      int defense;
64
65  public:
66      CombatEntity(int id, const std::string& name, float speed, int maxHp, int atk, int
    def)
67          : MovableObject(id, name, speed),
68            health(maxHp), maxHealth(maxHp), attackPower(atk), defense(def) {}
69
70      virtual void takeDamage(int damage) {
71          int actualDamage = std::max(0, damage - defense);
72          health -= actualDamage;
73          if (health < 0) health = 0;
74
75          if (health == 0) {
76              onDeath();
77          }
78      }
79
80      virtual void attack(CombatEntity* target) {
81          if (target && isAlive()) {
82              std::cout << name << " 攻击 " << target->getName() << "\n";
83              target->takeDamage(attackPower);
84          }
85      }
86
87      virtual void onDeath() {
88          std::cout << name << " 已死亡\n";
89          setActive(false);
90      }
91
92      bool isAlive() const { return health > 0; }
93      int getHealth() const { return health; }
94  };
95
96  // 玩家角色
97  class Player : public CombatEntity {
98  private:
```

```
99      int level;
100     int experience;
101     int experienceToNextLevel;
102     Inventory inventory;
103
104 public:
105     Player(const std::string& name)
106         : CombatEntity(1, name, 100.0f, 100, 10, 5),
107           level(1), experience(0), experienceToNextLevel(100),
108           inventory(20) {}
109
110     void gainExperience(int exp) {
111         experience += exp;
112         std::cout << name << " 获得 " << exp << " 经验值\n";
113
114         while (experience >= experienceToNextLevel) {
115             levelUp();
116         }
117     }
118
119     void levelUp() {
120         level++;
121         experience -= experienceToNextLevel;
122         experienceToNextLevel = level * 100;
123
124         // 属性提升
125         maxHealth += 20;
126         health = maxHealth;
127         attackPower += 5;
128         defense += 2;
129
130         std::cout << name << " 升级到 " << level << " 级！\n";
131     }
132
133     void render() const override {
134         std::cout << "[玩家] " << name << " Lv." << level
135                   << " HP:" << health << "/" << maxHealth
136                   << " 位置:(" << x << "," << y << ")\n";
137     }
138
139     Inventory& getInventory() { return inventory; }
140 };
141
142 // 敌人
143 class Enemy : public CombatEntity {
144 private:
145     int expReward;
146
147 public:
148     Enemy(int id, const std::string& name, int maxHp, int atk, int def, int exp)
149         : CombatEntity(id, name, 50.0f, maxHp, atk, def), expReward(exp) {}
150
151     void onDeath() override {
152         CombatEntity::onDeath();
```

```
153         std::cout << "击败 " << name << "，获得 " << expReward << " 经验值\n";
154     }
155 
156     int getExpReward() const { return expReward; }
157 
158     void render() const override {
159         std::cout << "[敌人] " << name << " HP:" << health << "/" << maxHealth
160                   << " 位置:(" << x << "," << y << ")\n";
161     }
162 };
163 
164 // NPC
165 class NPC : public GameObject {
166 private:
167     std::string dialogue;
168 
169 public:
170     NPC(int id, const std::string& name, const std::string& dialogue)
171         : GameObject(id, name), dialogue(dialogue) {}
172 
173     void talk() {
174         std::cout << name << ": \"" << dialogue << "\"\n";
175     }
176 
177     void update(float deltaTime) override {
178         // NPC通常不需要更新逻辑
179     }
180 
181     void render() const override {
182         std::cout << "[NPC] " << name << " 位置:(" << x << "," << y << ")\n";
183     }
184 };
```

1.2.3 继承中的构造和析构

C++

```
1  class Base {
2  public:
3      Base() { std::cout << "Base构造\n"; }
4      virtual ~Base() { std::cout << "Base析构\n"; }
5  };
6 
7  class Derived : public Base {
8  private:
9      int* data;
10 
11 public:
12     Derived() : Base() {  // 先调用基类构造
13         data = new int[100];
14         std::cout << "Derived构造\n";
15     }
16 
17     ~Derived() override {
18         delete[] data;
19         std::cout << "Derived析构\n";
20         // 然后自动调用Base析构
```

```
    }
};

// 使用
{
    Derived d;
    // 输出：
    // Base构造
    // Derived构造
}
// 作用域结束：
// Derived析构
// Base析构
```

1.2.4 多重继承（谨慎使用）

C++

```
// 接口类（纯虚函数）
class IRenderable {
public:
    virtual void render() const = 0;
    virtual ~IRenderable() = default;
};

class IUpdatable {
public:
    virtual void update(float deltaTime) = 0;
    virtual ~IUpdatable() = default;
};

class ICollidable {
public:
    virtual bool checkCollision(const ICollidable* other) const = 0;
    virtual ~ICollidable() = default;
};

// 游戏实体实现多个接口
class Bullet : public IRenderable, public IUpdatable, public ICollidable {
private:
    float x, y;
    float vx, vy;
    float radius;

public:
    Bullet(float x, float y, float vx, float vy)
        : x(x), y(y), vx(vx), vy(vy), radius(5.0f) {}

    void render() const override {
        std::cout << "渲染子弹于 (" << x << "," << y << ")\n";
    }

    void update(float deltaTime) override {
        x += vx * deltaTime;
        y += vy * deltaTime;
    }

```

```
    bool checkCollision(const ICollidable* other) const override {
        // 简化的碰撞检测
        return false;
    }
};
```

1.3 多态（Polymorphism）

1.3.1 什么是多态？

定义：同一操作作用于不同对象，产生不同的行为。

实现方式：

1. 编译时多态：函数重载、模板
2. 运行时多态：虚函数、动态绑定

1.3.2 虚函数机制

```cpp
// 技能系统
class Skill {
protected:
    std::string name;
    int manaCost;
    float cooldown;
    float currentCooldown;

public:
    Skill(const std::string& name, int mana, float cd)
        : name(name), manaCost(mana), cooldown(cd), currentCooldown(0) {}

    virtual ~Skill() = default;

    // 纯虚函数：子类必须实现
    virtual void execute(CombatEntity* caster, CombatEntity* target) = 0;

    // 虚函数：子类可选择重写
    virtual void update(float deltaTime) {
        if (currentCooldown > 0) {
            currentCooldown -= deltaTime;
        }
    }

    bool isReady() const { return currentCooldown <= 0; }

    void startCooldown() { currentCooldown = cooldown; }

    const std::string& getName() const { return name; }
    int getManaCost() const { return manaCost; }
};

// 攻击技能
class AttackSkill : public Skill {
private:
    int damageMultiplier;

public:
    AttackSkill(const std::string& name, int mana, float cd, int dmgMul)
```

```
40          : Skill(name, mana, cd), damageMultiplier(dmgMul) {}
41
42      void execute(CombatEntity* caster, CombatEntity* target) override {
43          if (!isReady() || !caster || !target) return;
44
45          int damage = caster->getAttackPower() * damageMultiplier;
46          std::cout << caster->getName() << " 使用 " << name
47                    << " 对 " << target->getName() << " 造成 " << damage << " 伤害\n";
48          target->takeDamage(damage);
49          startCooldown();
50      }
51  };
52
53  // 治疗技能
54  class HealSkill : public Skill {
55  private:
56      int healAmount;
57
58  public:
59      HealSkill(const std::string& name, int mana, float cd, int heal)
60          : Skill(name, mana, cd), healAmount(heal) {}
61
62      void execute(CombatEntity* caster, CombatEntity* target) override {
63          if (!isReady() || !target) return;
64
65          std::cout << caster->getName() << " 使用 " << name
66                    << " 治疗 " << target->getName() << " " << healAmount << " 生命值\n";
67          // 假设CombatEntity有heal方法
68          startCooldown();
69      }
70  };
71
72  // 范围攻击技能
73  class AOESkill : public Skill {
74  private:
75      int damage;
76      float range;
77
78  public:
79      AOESkill(const std::string& name, int mana, float cd, int dmg, float r)
80          : Skill(name, mana, cd), damage(dmg), range(r) {}
81
82      void execute(CombatEntity* caster, CombatEntity* target) override {
83          if (!isReady() || !caster) return;
84
85          std::cout << caster->getName() << " 使用范围技能 " << name << "\n";
86          // 对范围内所有敌人造成伤害
87          startCooldown();
88      }
89  };
90
91  // 技能管理器
92  class SkillManager {
93  private:
```

```
94      std::vector<std::unique_ptr<Skill>> skills;
95
96  public:
97      void addSkill(std::unique_ptr<Skill> skill) {
98          skills.push_back(std::move(skill));
99      }
100
101     void useSkill(size_t index, CombatEntity* caster, CombatEntity* target) {
102         if (index < skills.size()) {
103             skills[index]->execute(caster, target);
104         }
105     }
106
107     void updateAll(float deltaTime) {
108         for (auto& skill : skills) {
109             skill->update(deltaTime);
110         }
111     }
112
113     void listSkills() const {
114         std::cout << "技能列表:\n";
115         for (size_t i = 0; i < skills.size(); ++i) {
116             std::cout << i << ". " << skills[i]->getName()
117                       << " (魔法消耗: " << skills[i]->getManaCost() << ")"
118                       << (skills[i]->isReady() ? " [就绪]" : " [冷却中]") << "\n";
119         }
120     }
121 };
```

1.3.3 虚函数表（vtable）原理

C++

```
1   class Animal {
2   public:
3       virtual void makeSound() { std::cout << "Animal sound\n"; }
4       virtual void move() { std::cout << "Animal moves\n"; }
5       virtual ~Animal() = default;
6   };
7
8   class Dog : public Animal {
9   public:
10      void makeSound() override { std::cout << "Woof!\n"; }
11      void move() override { std::cout << "Dog runs\n"; }
12  };
13
14  class Cat : public Animal {
15  public:
16      void makeSound() override { std::cout << "Meow!\n"; }
17      // 不重写move，使用基类实现
18  };
19
20  // 内存布局
21  /*
22  Animal对象:
23  ┌──────────┬──────┐
24  │  vptr    │ data │
```

```
└──────────┴────────┘
     |
     └──> Animal的vtable:
          ┌──────────────────────┐
          | &Animal::makeSound   |
          | &Animal::move        |
          | &Animal::~Animal     |
          └──────────────────────┘

Dog对象:
┌──────────┬────────┐
|  vptr    | data   |
└──────────┴────────┘
     |
     └──> Dog的vtable:
          ┌──────────────────────┐
          | &Dog::makeSound      |  // 重写
          | &Dog::move           |  // 重写
          | &Dog::~Dog           |
          └──────────────────────┘
*/

// 多态调用
void playWithAnimal(Animal* animal) {
    animal->makeSound();  // 运行时决定调用哪个版本
    animal->move();
}

Dog dog;
Cat cat;
playWithAnimal(&dog);  // 输出：Woof! Dog runs
playWithAnimal(&cat);  // 输出：Meow! Animal moves
```

1.3.4 游戏场景：AI 系统

C++

```cpp
// AI行为基类
class AIBehavior {
protected:
    std::string behaviorName;

public:
    AIBehavior(const std::string& name) : behaviorName(name) {}
    virtual ~AIBehavior() = default;

    // 执行行为
    virtual void execute(Enemy* entity, Player* target, float deltaTime) = 0;

    // 判断是否可以执行该行为
    virtual bool canExecute(Enemy* entity, Player* target) const = 0;

    const std::string& getName() const { return behaviorName; }
};

// 追逐行为
class ChaseBehavior : public AIBehavior {
```

```
21  private:
22      float chaseRange;
23
24  public:
25      ChaseBehavior(float range = 200.0f)
26          : AIBehavior("Chase"), chaseRange(range) {}
27
28      bool canExecute(Enemy* entity, Player* target) const override {
29          if (!entity || !target) return false;
30
31          float dx = target->getX() - entity->getX();
32          float dy = target->getY() - entity->getY();
33          float distance = std::sqrt(dx*dx + dy*dy);
34
35          return distance < chaseRange;
36      }
37
38      void execute(Enemy* entity, Player* target, float deltaTime) override {
39          float dx = target->getX() - entity->getX();
40          float dy = target->getY() - entity->getY();
41          float distance = std::sqrt(dx*dx + dy*dy);
42
43          if (distance > 0) {
44              float vx = dx / distance;
45              float vy = dy / distance;
46              entity->setVelocity(vx, vy);
47          }
48      }
49  };
50
51  // 攻击行为
52  class AttackBehavior : public AIBehavior {
53  private:
54      float attackRange;
55      float attackCooldown;
56      float currentCooldown;
57
58  public:
59      AttackBehavior(float range = 30.0f, float cooldown = 1.0f)
60          : AIBehavior("Attack"), attackRange(range),
61            attackCooldown(cooldown), currentCooldown(0) {}
62
63      bool canExecute(Enemy* entity, Player* target) const override {
64          if (!entity || !target || currentCooldown > 0) return false;
65
66          float dx = target->getX() - entity->getX();
67          float dy = target->getY() - entity->getY();
68          float distance = std::sqrt(dx*dx + dy*dy);
69
70          return distance <= attackRange;
71      }
72
73      void execute(Enemy* entity, Player* target, float deltaTime) override {
74          if (currentCooldown > 0) {
75              currentCooldown -= deltaTime;
```

```
76              return;
77          }
78
79          entity->attack(target);
80          currentCooldown = attackCooldown;
81      }
82  };
83
84  // 巡逻行为
85  class PatrolBehavior : public AIBehavior {
86  private:
87      std::vector<std::pair<float, float>> waypoints;
88      size_t currentWaypoint;
89      float waypointReachDistance;
90
91  public:
92      PatrolBehavior(const std::vector<std::pair<float, float>>& points)
93          : AIBehavior("Patrol"), waypoints(points), currentWaypoint(0),
94            waypointReachDistance(10.0f) {}
95
96      bool canExecute(Enemy* entity, Player* target) const override {
97          return !waypoints.empty();
98      }
99
100     void execute(Enemy* entity, Player* target, float deltaTime) override {
101         if (waypoints.empty()) return;
102
103         auto& [targetX, targetY] = waypoints[currentWaypoint];
104         float dx = targetX - entity->getX();
105         float dy = targetY - entity->getY();
106         float distance = std::sqrt(dx*dx + dy*dy);
107
108         if (distance < waypointReachDistance) {
109             // 到达当前巡逻点，前往下一个
110             currentWaypoint = (currentWaypoint + 1) % waypoints.size();
111         } else {
112             // 移动向巡逻点
113             float vx = dx / distance;
114             float vy = dy / distance;
115             entity->setVelocity(vx * 0.5f, vy * 0.5f);  // 巡逻速度较慢
116         }
117     }
118 };
119
120 // AI控制器
121 class AIController {
122 private:
123     std::vector<std::unique_ptr<AIBehavior>> behaviors;
124     AIBehavior* currentBehavior;
125
126 public:
127     AIController() : currentBehavior(nullptr) {}
128
129     void addBehavior(std::unique_ptr<AIBehavior> behavior) {
130         behaviors.push_back(std::move(behavior));
```

```
    }

    void update(Enemy* entity, Player* target, float deltaTime) {
        // 选择优先级最高且可执行的行为
        for (auto& behavior : behaviors) {
            if (behavior->canExecute(entity, target)) {
                if (currentBehavior != behavior.get()) {
                    currentBehavior = behavior.get();
                    std::cout << entity->getName() << " 切换行为: "
                              << currentBehavior->getName() << "\n";
                }
                currentBehavior->execute(entity, target, deltaTime);
                return;
            }
        }

        // 没有可执行的行为，停止移动
        entity->setVelocity(0, 0);
        currentBehavior = nullptr;
    }
};

// 使用示例
void setupEnemy() {
    Enemy enemy(1, "哥布林", 50, 5, 2, 25);

    AIController ai;
    ai.addBehavior(std::make_unique<AttackBehavior>(30.0f, 1.0f));  // 优先级最高
    ai.addBehavior(std::make_unique<ChaseBehavior>(200.0f));
    ai.addBehavior(std::make_unique<PatrolBehavior>(
        std::vector<std::pair<float, float>>{{0,0}, {100,0}, {100,100}, {0,100}}
    ));

    // 游戏循环中
    // ai.update(&enemy, &player, deltaTime);
}
```

1.4 抽象类与接口

1.4.1 抽象类

定义：包含至少一个纯虚函数的类，不能实例化，用作基类定义接口。

```
// 武器系统
class Weapon {
protected:
    std::string name;
    int damage;
    float range;

public:
    Weapon(const std::string& name, int dmg, float rng)
        : name(name), damage(dmg), range(rng) {}

    virtual ~Weapon() = default;
```

```
14      // 纯虚函数：攻击方法
15      virtual void attack(const std::string& targetName) = 0;
16
17      // 纯虚函数：特殊能力
18      virtual void specialAbility() = 0;
19
20      // 普通虚函数：可以有默认实现
21      virtual void display() const {
22          std::cout << "武器: " << name << " 伤害:" << damage
23                    << " 范围:" << range << "\n";
24      }
25
26      int getDamage() const { return damage; }
27      float getRange() const { return range; }
28  };
29
30  // Weapon weapon;  // ✗ 错误！不能实例化抽象类
31
32  // 近战武器
33  class MeleeWeapon : public Weapon {
34  protected:
35      float attackSpeed;
36
37  public:
38      MeleeWeapon(const std::string& name, int dmg, float speed)
39          : Weapon(name, dmg, 2.0f), attackSpeed(speed) {}
40
41      void attack(const std::string& targetName) override {
42          std::cout << "用 " << name << " 近战攻击 " << targetName
43                    << "，造成 " << damage << " 伤害\n";
44      }
45  };
46
47  // 剑
48  class Sword : public MeleeWeapon {
49  public:
50      Sword() : MeleeWeapon("铁剑", 25, 1.0f) {}
51
52      void specialAbility() override {
53          std::cout << "剑技：旋风斩！范围伤害 " << damage * 1.5 << "\n";
54      }
55  };
56
57  // 锤子
58  class Hammer : public MeleeWeapon {
59  public:
60      Hammer() : MeleeWeapon("战锤", 40, 0.7f) {}
61
62      void specialAbility() override {
63          std::cout << "重击：震地！眩晕敌人\n";
64      }
65  };
66
67  // 远程武器
```

```cpp
class RangedWeapon : public Weapon {
protected:
    int ammo;
    int maxAmmo;

public:
    RangedWeapon(const std::string& name, int dmg, float rng, int maxAmmo)
        : Weapon(name, dmg, rng), ammo(maxAmmo), maxAmmo(maxAmmo) {}

    void attack(const std::string& targetName) override {
        if (ammo > 0) {
            std::cout << "用 " << name << " 远程攻击 " << targetName
                      << "，造成 " << damage << " 伤害\n";
            ammo--;
        } else {
            std::cout << "弹药耗尽！需要装填\n";
        }
    }

    void reload() {
        ammo = maxAmmo;
        std::cout << name << " 已装填\n";
    }
};

// 弓
class Bow : public RangedWeapon {
public:
    Bow() : RangedWeapon("长弓", 20, 50.0f, 30) {}

    void specialAbility() override {
        if (ammo >= 3) {
            std::cout << "多重箭！发射3支箭\n";
            ammo -= 3;
        }
    }
};

// 使用多态
void testWeapon(Weapon* weapon) {
    weapon->display();
    weapon->attack("哥布林");
    weapon->specialAbility();
}

Sword sword;
Bow bow;
testWeapon(&sword);
testWeapon(&bow);
```

1.4.2 接口设计模式

```cpp
// 保存/加载接口
class ISaveable {
public:
```

```
    virtual void save(std::ostream& out) const = 0;
    virtual void load(std::istream& in) = 0;
    virtual ~ISaveable() = default;
};

// 玩家数据实现保存接口
class PlayerData : public ISaveable {
private:
    std::string playerName;
    int level;
    int health;
    float x, y;

public:
    void save(std::ostream& out) const override {
        out << playerName << "\n"
            << level << "\n"
            << health << "\n"
            << x << " " << y << "\n";
    }

    void load(std::istream& in) override {
        std::getline(in, playerName);
        in >> level >> health >> x >> y;
        in.ignore();  // 忽略换行符
    }
};

// 存档管理器
class SaveManager {
public:
    static void saveGame(const std::string& filename, const ISaveable& data) {
        std::ofstream file(filename);
        if (file.is_open()) {
            data.save(file);
            std::cout << "游戏已保存到 " << filename << "\n";
        }
    }

    static void loadGame(const std::string& filename, ISaveable& data) {
        std::ifstream file(filename);
        if (file.is_open()) {
            data.load(file);
            std::cout << "游戏已从 " << filename << " 加载\n";
        }
    }
};
```

1.5 组合 vs 继承

1.5.1 何时使用继承？

Is-A 关系：派生类是基类的一种特化

```
// ✓ 好的继承：明确的is-a关系
class Vehicle { };
```

```
class Car : public Vehicle { };  // Car is a Vehicle

class Animal { };
class Dog : public Animal { };   // Dog is an Animal
```

1.5.2 何时使用组合？

Has-A 关系：对象包含另一个对象

```C++
// ✓ 好的组合：has-a关系
class Engine {
public:
    void start() { std::cout << "引擎启动\n"; }
    void stop() { std::cout << "引擎停止\n"; }
};

class Car {
private:
    Engine engine;  // Car has an Engine

public:
    void start() {
        engine.start();
        std::cout << "汽车启动\n";
    }
};
```

1.5.3 游戏场景：组件系统（推荐）

```C++
// 组件基类
class Component {
protected:
    bool enabled;

public:
    Component() : enabled(true) {}
    virtual ~Component() = default;

    virtual void update(float deltaTime) = 0;

    void setEnabled(bool e) { enabled = e; }
    bool isEnabled() const { return enabled; }
};

// 具体组件
class TransformComponent : public Component {
public:
    float x, y;
    float rotation;
    float scaleX, scaleY;

    TransformComponent()
        : x(0), y(0), rotation(0), scaleX(1), scaleY(1) {}

    void update(float deltaTime) override {
        // 变换更新逻辑
```

```
    }

    void setPosition(float newX, float newY) {
        x = newX;
        y = newY;
    }
};

class SpriteComponent : public Component {
private:
    std::string texturePath;
    int width, height;

public:
    SpriteComponent(const std::string& path, int w, int h)
        : texturePath(path), width(w), height(h) {}

    void update(float deltaTime) override {
        // 精灵更新逻辑
    }

    void render(float x, float y) const {
        std::cout << "渲染精灵 " << texturePath
                  << " 于 (" << x << "," << y << ")\n";
    }
};

class PhysicsComponent : public Component {
private:
    float velocityX, velocityY;
    float mass;
    bool useGravity;

public:
    PhysicsComponent(float m = 1.0f, bool gravity = true)
        : velocityX(0), velocityY(0), mass(m), useGravity(gravity) {}

    void update(float deltaTime) override {
        if (useGravity) {
            velocityY += 9.8f * deltaTime;  // 重力加速度
        }
    }

    void setVelocity(float vx, float vy) {
        velocityX = vx;
        velocityY = vy;
    }

    float getVelocityX() const { return velocityX; }
    float getVelocityY() const { return velocityY; }
};

class HealthComponent : public Component {
private:
    int currentHealth;
```

```
83      int maxHealth;
84
85  public:
86      HealthComponent(int maxHp)
87          : currentHealth(maxHp), maxHealth(maxHp) {}
88
89      void update(float deltaTime) override {
90          // 生命值相关更新（如自动回复）
91      }
92
93      void takeDamage(int damage) {
94          currentHealth -= damage;
95          if (currentHealth < 0) currentHealth = 0;
96      }
97
98      void heal(int amount) {
99          currentHealth += amount;
100         if (currentHealth > maxHealth) currentHealth = maxHealth;
101     }
102
103     bool isAlive() const { return currentHealth > 0; }
104     int getHealth() const { return currentHealth; }
105 };
106
107 // 游戏实体：组合多个组件
108 class Entity {
109 private:
110     std::string name;
111     std::map<std::string, std::unique_ptr<Component>> components;
112
113 public:
114     Entity(const std::string& name) : name(name) {}
115
116     template<typename T, typename... Args>
117     T* addComponent(const std::string& compName, Args&&... args) {
118         auto comp = std::make_unique<T>(std::forward<Args>(args)...);
119         T* ptr = comp.get();
120         components[compName] = std::move(comp);
121         return ptr;
122     }
123
124     template<typename T>
125     T* getComponent(const std::string& compName) {
126         auto it = components.find(compName);
127         if (it != components.end()) {
128             return dynamic_cast<T*>(it->second.get());
129         }
130         return nullptr;
131     }
132
133     void update(float deltaTime) {
134         for (auto& [name, comp] : components) {
135             if (comp->isEnabled()) {
136                 comp->update(deltaTime);
137             }
```

```
        }
    }

    const std::string& getName() const { return name; }
};

// 使用示例：创建一个玩家实体
Entity* createPlayer() {
    auto player = new Entity("Player");

    // 添加变换组件
    auto transform = player->addComponent<TransformComponent>("transform");
    transform->setPosition(100, 100);

    // 添加精灵组件
    player->addComponent<SpriteComponent>("sprite", "player.png", 32, 32);

    // 添加物理组件
    auto physics = player->addComponent<PhysicsComponent>("physics", 1.0f, true);
    physics->setVelocity(50, 0);

    // 添加生命值组件
    player->addComponent<HealthComponent>("health", 100);

    return player;
}

// 游戏循环
void gameLoop() {
    Entity* player = createPlayer();
    float deltaTime = 0.016f;  // 约60 FPS

    // 更新所有组件
    player->update(deltaTime);

    // 访问特定组件
    auto health = player->getComponent<HealthComponent>("health");
    if (health) {
        health->takeDamage(10);
        std::cout << "玩家生命值: " << health->getHealth() << "\n";
    }

    auto transform = player->getComponent<TransformComponent>("transform");
    auto sprite = player->getComponent<SpriteComponent>("sprite");
    if (transform && sprite) {
        sprite->render(transform->x, transform->y);
    }

    delete player;
}
```

1.6 面向对象设计原则

1.6.1 SOLID 原则

1. 单一职责原则（Single Responsibility Principle）

一个类应该只有一个引起它变化的原因。

```cpp
// ✗ 违反SRP：一个类做太多事
class BadPlayer {
    void move() { }
    void attack() { }
    void saveToDatabase() { }  // 不应该在这里！
    void renderGraphics() { }  // 不应该在这里！
};

// ✓ 遵循SRP：职责分离
class Player {
    void move() { }
    void attack() { }
};

class PlayerRepository {
    void save(const Player& player) { }
    void load(Player& player) { }
};

class PlayerRenderer {
    void render(const Player& player) { }
};
```

2. 开闭原则（Open-Closed Principle）

对扩展开放，对修改关闭。

```cpp
// ✓ 通过继承扩展，不修改原有代码
class DamageCalculator {
public:
    virtual int calculate(int baseDamage) const {
        return baseDamage;
    }
    virtual ~DamageCalculator() = default;
};

class CriticalDamageCalculator : public DamageCalculator {
private:
    float critChance;
    float critMultiplier;

public:
    CriticalDamageCalculator(float chance, float multiplier)
        : critChance(chance), critMultiplier(multiplier) {}

    int calculate(int baseDamage) const override {
        if (rand() % 100 < critChance * 100) {
            return baseDamage * critMultiplier;
        }
        return baseDamage;
    }
};
```

3. 里氏替换原则（Liskov Substitution Principle）

子类对象应该能够替换父类对象。

```cpp
// ✓ 正确的LSP
class Bird {
public:
    virtual void eat() { std::cout << "鸟在吃\n"; }
    virtual ~Bird() = default;
};

class Sparrow : public Bird {
public:
    void eat() override { std::cout << "麻雀在吃虫子\n"; }
};

void feedBird(Bird* bird) {
    bird->eat();  // 任何Bird的子类都可以
}

// ✗ 违反LSP的经典例子
class BirdBad {
public:
    virtual void fly() { std::cout << "飞\n"; }
};

class Penguin : public BirdBad {
public:
    void fly() override {
        throw std::logic_error("企鹅不会飞！");  // 违反LSP！
    }
};
```

4. 接口隔离原则（Interface Segregation Principle）

不应该强迫客户端依赖它不使用的接口。

```cpp
// ✗ 违反ISP：庞大的接口
class IWorker {
public:
    virtual void work() = 0;
    virtual void eat() = 0;
    virtual void sleep() = 0;
};

// Robot实现IWorker，但不需要eat和sleep

// ✓ 遵循ISP：接口分离
class IWorkable {
public:
    virtual void work() = 0;
    virtual ~IWorkable() = default;
};

class IFeedable {
public:
    virtual void eat() = 0;
    virtual ~IFeedable() = default;
```

```cpp
};

class ISleepable {
public:
    virtual void sleep() = 0;
    virtual ~ISleepable() = default;
};

class Human : public IWorkable, public IFeedable, public ISleepable {
public:
    void work() override { std::cout << "工作\n"; }
    void eat() override { std::cout << "吃饭\n"; }
    void sleep() override { std::cout << "睡觉\n"; }
};

class Robot : public IWorkable {
public:
    void work() override { std::cout << "工作\n"; }
    // 不需要实现eat和sleep
};
```

5. 依赖倒置原则（Dependency Inversion Principle）

高层模块不应该依赖低层模块，两者都应该依赖抽象。

C++

```cpp
// ✗ 违反DIP：直接依赖具体类
class BadGame {
    Sword sword;  // 依赖具体武器

public:
    void attack() {
        sword.attack("敌人");
    }
};

// ✓ 遵循DIP：依赖抽象
class GoodGame {
private:
    std::unique_ptr<Weapon> weapon;  // 依赖抽象接口

public:
    void setWeapon(std::unique_ptr<Weapon> w) {
        weapon = std::move(w);
    }

    void attack(const std::string& target) {
        if (weapon) {
            weapon->attack(target);
        }
    }
};

// 使用
GoodGame game;
game.setWeapon(std::make_unique<Sword>());
```

```
31  game.attack("哥布林");
32
33  game.setWeapon(std::make_unique<Bow>());
34  game.attack("龙");
```

1.7 总结与最佳实践

封装：

- 优先使用 private，必要时用 protected，谨慎使用 public
- 提供 getter/setter 保护数据完整性
- 隐藏实现细节，暴露稳定接口

继承：

- 用于“is-a”关系
- 避免深层继承（一般不超过 3 层）
- 基类析构函数声明为 virtual
- 考虑使用 final 防止继承

多态：

- 通过虚函数实现运行时多态
- 纯虚函数定义接口契约
- 优先使用接口（抽象类）而非具体类

组合优于继承：

- 优先考虑组件系统
- 更灵活，耦合度更低
- 更容易测试和维护

设计原则：

- 遵循 SOLID 原则
- 保持高内聚、低耦合
- 面向接口编程
- 优先使用组合而非继承