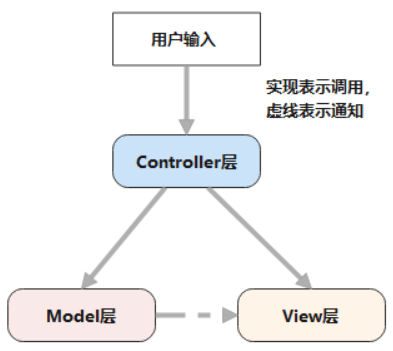
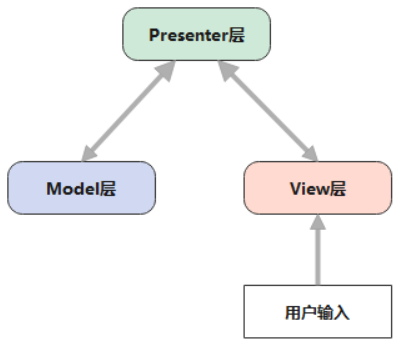
目的是增强代码的可重用性、可扩充性、可维护性，最终的目的是实现代码的高内聚和低耦合。

**MVC、MVP**

MVC：Model：模型、View：视图、Controller：控制。



MVP：Model：模型、View：视图、Presenter：表示。



基于MVC来实现背包系统：

Model——定义物品结构，保存物品数据。

View——控制背包显示方式。

Controller——背包内物品摆放，操作逻辑，拖拽事件处理。

**1. 面向对象设计的原则**

**1.1 开放封闭原则（OCP）**

软件实体（类、模块、函数等）应对扩展开放，但对修改封闭。

抽象是关键。

**1.2 单一职责原则（SRP）**

一个类应该有且只有一个变化的原因（职责）。

如何区分一个职责还是两个职责：看职责中是否可以再分离出两个没有任何交集的职责。

**1.3 里氏替换原则（LSP）**

使用基类对象指针或引用的函数必须能够在不了解衍生类的条件下使用衍生类的对象。即所有的衍生类必须符合使用者所期待的基类的行为。（是实现OCP原则的重要方式）

例如长方形和正方形，若正方形继承长方形，则在修改高度height时宽度width也会变，这不符合基类意图，违背了里氏替换原则。

**1.4 接口分离原则（ISP）**

客户类不应依赖他不需要的接口。即一个接口实现一个单独的功能。

**1.5 依赖倒置原则（DIP）**

A. 高层模块不应该依赖于低层模块，二者都应该依赖于抽象。

B. 抽象不应该依赖于具体实现细节，而具体实现细节应该依赖于抽象。

要针对接口编程，不要针对实现编程。

**1.6 最少知识原则/迪米特法则（LKP）**

只和自己直接的朋友交谈。对象应尽可能地避免调用由另一个方法返回的对象的方法

**2. 创建型模式**

对象实例化的模式，创建型模式用于解耦对象的实例化过程。

**\*2.1 单例模式**

饿汉式：声明时创建Instance，static构造函数创建。

class Singleton {

public:

static Singleton\* getInstance();

static void deleteInstance();

private:

Singleton();

~Singleton();

Singleton(const Singleton& singleton);

Singleton& operator=(const Singleton& singleton);

static Singleton\* m\_singleton;

};

Singleton\* Singleton::m\_singleton = new Singleton;

Singleton\* Singleton::getInstance() {

return m\_singleton;

}

void Singleton::deleteInstance() {

if (m\_singleton) {

delete m\_singleton;

m\_singleton = nullptr;

}

}

懒汉式：第一次使用对象时加载Instance，非线程安全/锁的性能消耗。

class Singleton {

public:

static Singleton\* getInstance();

static void deleteInstance();

private:

Singleton();

~Singleton();

Singleton(const Singleton& singleton);

Singleton& operator=(const Singleton& singleton);

static Singleton\* m\_singleton;

static std::mutex m\_mutex;

};

Singleton\* Singleton::m\_singleton = nullptr;

std::mutex Singleton::m\_mutex;

Singleton\* Singleton::getInstance() {

if (m\_singleton == nullptr) {

std::unique\_lock<std::mutex> lock(m\_mutex);

if (m\_singleton == nullptr) {

m\_singleton = new Singleton;

}

}

return m\_singleton;

}

void Singleton::deleteInstance() {

std::unique\_lock<std::mutex> lock(m\_mutex);

if (m\_singleton) {

delete m\_singleton;

m\_singleton = nullptr;

}

}

Unity中：可以通过继承的方式实现或通过添加静态方法/属性的方式实现。在Awake中初始化，若希望其它脚本在Awake中获取到，可叠加懒汉式（安全版本：懒汉基础上，在Awake中，若已存在销毁自身以及在OnDestroy中将Instance置空）。

**\*2.2 工厂方法模式**

定义一个用于创建对象的接口，让子类决定实例化哪一个类（构造函数），每个工厂定义一个工厂方法来创造产品，当产品过多时工厂泛滥。（生成某种种类的敌人，不在乎该敌人怎么生成）

**2.3 抽象工厂模式**

提供一个创建一系列相关或相互依赖对象的接口，而无需指定它们的具体类。需要将产品进行分类。（各汽车厂生产燃油车与新能源车）

**2.4 建造者模式**

封装一个复杂对象的创建过程，并可以按步骤构造。Director、Builder、Product。（房子打地基、砌墙、屋顶）

**2.5 原型模式**

通过复制现有的实例来创建新的实例。（重写拷贝构造函数/创造新的拷贝函数）

**3. 结构型模式**

把类或对象结合在一起形成一个更大的结构。

**\*3.1 装饰器模式**

动态的给对象添加新的功能。（各个组件添加，煎饼果子加鸡蛋香肠）

**3.2 代理模式**

为其它对象提供一个代理以便控制这个对象的访问。（核心功能封装）

静态代理：在运行前代理类就已经存在。

动态代理：在运行过程中产生代理类。

**3.3 桥接模式**

将抽象部分和它的实现部分分离，使它们都可以独立的变化。（按键映射指定功能）

**\*3.4 适配器模式**

将一个类的方法接口转换成客户希望的另一个接口。（排序的多种方法实现）

**3.5 组合模式**

将对象组合成树形结构以表示“部分-整体”的层次结构。（学校包含学院，学院包含专业）

**3.6 外观模式**

对外提供一个统一的方法，来访问子系统中的一群接口。（吃饭：包括迎宾接待、厨师做饭、服务员上菜、清洁工收拾洗碗、基金：用户只和基金打交道，实际为基金经理人与股票等投资品打交道）

**3.7 享元模式**

通过共享技术来有效的支持大量细粒度的对象。（string常量池）

**4. 行为型模式**

类和对象如何交互，及划分责任和算法。

**4.1 策略模式**

定义一系列算法，把他们封装起来，并且使它们可以相互替换。（父类abstract方法，不同子类重写不同，打折策略不同收费不同）

**4.2 模板模式**

定义一个算法结构，而将一些步骤延迟到子类实现。（例如做饭，准备原料的步骤相同，子类可以分别煎或炒）

**4.3 命令模式**

将命令请求封装为一个对象，使得可以用不同的请求来进行参数化。（将灯的开关、电视机的开关操作封装成对象，并绑定到控制器上，控制器可根据指令进行操作）

**4.4 迭代器模式**

一种遍历访问聚合对象中各个元素的方法，不暴露该对象的内部结构。（C++的STL）

**\*4.5 观察者模式**

对象间的一对多的依赖关系，当一个对象状态发生改变时，所有依赖它的对象都会得到通知并自动更新。（发布订阅模式）（C#中的委托、店铺到货时店主通知所有顾客而不是顾客定期询问）

**4.6 仲裁/中介者模式**

用一个中介对象来封装一系列的对象交互。（聊天室中转站，发送方发送消息给聊天室，接收方从聊天室接收消息）

**4.7 备忘录模式**

在不破坏封装的前提下，保持对象的内部状态。（保存某一时间的状态，便于后续读取或恢复）

**4.8 解释器模式**

给定一个语言，定义它的文法的一种表示，并定义一个解释器。（字符串解码）

**\*4.9 状态模式**

允许一个对象在其对象内部状态改变时改变它的行为。（类似状态机）

**4.10 责任链模式**

将请求的发送者和接收者解耦，使的多个对象都有处理这个请求的机会。（发送者只和底层接受者交流，若其处理不了则依次交给上级）

**4.11 访问者模式**

不改变数据结构的前提下，增加作用于一组对象元素的新功能。（对象继承访问接口，传递自身供访问）

**5. 游戏编程模式**

**5.1 子类沙盒模式**

使用基类提供的操作集合来定义子类中的行为。（一个技能除自己的特性外，具有多个共同特征如播放声音、粒子效果等，写在父类中）

**5.2 类型对象模式**

通过创建一个类来支持新类型的灵活创建，其每个实例都代表一个不同对象类型。（先创建种类，再根据种类创建对象）

**5.3 组件模式/ ECS系统**

允许一个单一的实体跨越多个不同域而不会导致耦合。

ECS：Entity Component System 实体组件系统

Entity是实例，作为承载组件的载体，也是框架中维护对象的实体。

Component只包含数据，具备这个组件便具有这个功能。

System作为逻辑维护，维护对应的组件执行相关操作。

**5.4 事件队列模式**

对消息或事件的发送与受理进行时间上的解耦。（Services.EventSystem）

**5.5 游戏循环模式**

实现游戏运行过程中对用户输入处理和时间处理的解耦。（Unity内置了Update的游戏循环模式）

**5.6 服务定位器模式**

提供服务的全局接入点，而不必让用户和实现它的具体类耦合。（Services.ServiceLocator，可以通过单例实现，Unity引擎把这个模式和组件模式结合并用在了GetComponent中）

**5.7 数据局部性模式**

通过合理组织数据利用CPU的缓存机制来加快内存访问速度。（基于大数组存储保证数据连续性）

**5.8 脏标记模式**

将工作推迟到必要时进行以避免不必要的工作，减轻计算开销和同步开销。（渲染时若为dirty再渲染）

**5.9 对象池模式**

使用固定的对象池重用对象，取代单独地分配和释放对象，以此来达到提升性能和优化内存使用的目的。