物理攻击类道具模块

物理攻击类道具有火、冰和火箭三种。在吃掉物理攻击类道具后，会获得三次使用相应道具机会，但是不同的道具攻击策略和效果是不同的。

图x

例如，火道具如上图左所示。在玩家吃掉道具后，会获得三次喷火机会，每次火会在相应方向三个格施法，如上图右所示。受到火焰伤害的玩家会掉一点血。冰道具和火道具是完全相同的，只是道具形状和动画效果不同。

图x

火箭炮道具如上图所示。和火道具相比，火箭道具是发射一枚火箭，一直直线飞行直到碰撞到玩家或墙壁。受到火箭伤害的玩家会掉一点血。

从设计的角度，这三种道具只在使用策略上有所不同。对于玩家来说，只需要发出攻击指令，即可使用相应道具。可以使用策略模式，将将可变的部分从程序中抽象出来分离成算法接口，在该接口下分别封装一系列算法实现。

图x

该模块使用策略模式的类图如上图所示。抽象策略接口（Strategy）定义了一个公共接口，各种不同的算法以不同的方式实现这个接口；具体策略（Concrete Strategy）类实现了具体的冰、火、火箭的攻击策略，提

供具体的算法实现；环境（Context）类是此处的Hero类，持有一个策略类的引用，最终给客户端调用。接口定义如下：

```
public interface ISkill {
    void useSkill();
}
```

具体的策略（以火道具为例）类如下：

```
public class FireSkill implements ISkill {
    @Override
    void useSkill()
    {
            //实现向指定方向距离为3的格子释放火焰
    }
}
```

玩家控制的角色继承自虚基类英雄Hero类，在里面对当前英雄道具（技能）进行设定。

```
public abstract class Hero {
    //技能属性
    private ISkill iskill;
    //技能的setter
    public void setIskill(ISkill iskill) { this.iskill = iskill; }
    //技能使用
    public void attack() { iskill.useSkill(); }
}
```

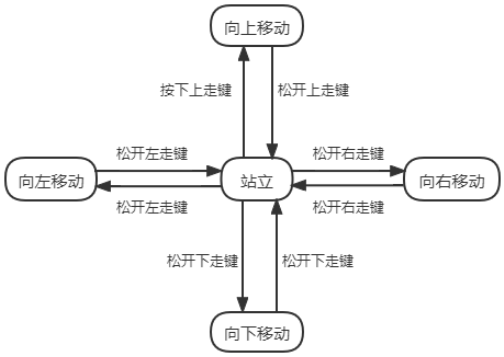
这样就完成了采用策略模式对攻击行为进行多种方式的实现。后续如果采用更多的攻击方法，或者新增更多涉及物理攻击的道具，都可以继承自ISkill接口，易于扩展、理解和切换。同时具有更好的代码复用性，使用组合而不是继承让架构更加灵活。

人物行走模块

在玩家角色行走的过程中，对应有多种运动状态。

图x

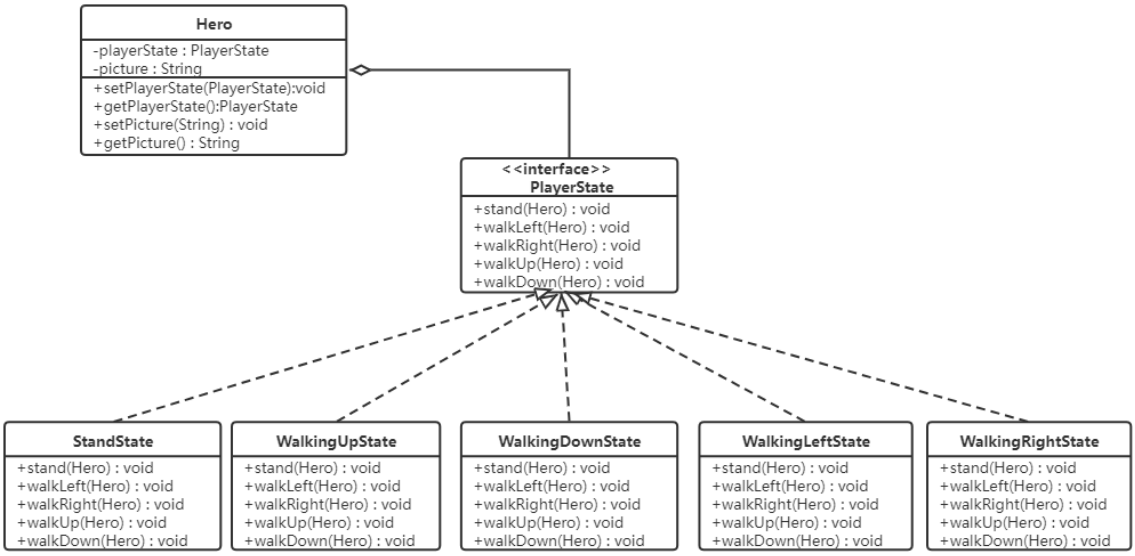
以上图黄色人物为例，它对应了多种状态：，向下移动，向左移动，向右移动，站立。这个玩家在移动上对应不同的状态，而他的行为也取决于当前的状态，状态图如下图所示。



图x

可以看出，这个对象的行为取决于它当前的动作，例如初始状态下人物是向下站立状态，那么它可以向上下左右四个方向进行移动，对应的动画为头朝下。

类的设计如下图所示，五种状态（向上移动，向下移动，向左移动，向右移动，站立）实现了接口PlayerState类，在不同的状态下，执行的函数逻辑会有所不同。在Hero类中持有一个状态类变量playerState，记录当前的状态，picture变量用来记录当前的角色的动画画面为本地哪图片（分别对应头朝上、头朝下、头朝左、头朝右）。在需要修改状态的时候，需要将Hero的引用传入，实现状态的修改。



图x

此时Hero类的内容如下所示：

```
public class Hero {
    //五种状态
    public static final PlayerState STAND = new StandState();
    public static final PlayerState WALKLEFT = new WalkingLeftState();
    public static final PlayerState WALKRIGHT = new WalkingRightState();
    public static final PlayerState WALKUP = new WalkingUpState();
    public static final PlayerState WALKDOWN = new WalkingDownState();

    //移动状态，初始为站立状态
    private PlayerState playerState = STAND;
    //对应头朝上、头朝下、头朝左、头朝右的图片路径
```

```
    private String picture;
    //状态的getter和setter
    public void setPlayerState(PlayerState playerState);
    public PlayerState getPlayerState();
    //图片路径的getter和setter
    public void setPicture(String file);
    public String getPicture();
}
```

PlayerState接口定义如下：

```
public interface PlayerState
{
    void stand(Hero hero);
    void walkLeft(Hero hero);
    void walkRight(Hero hero);
    void walkUp(Hero hero);
    void walkDown(Hero hero);
}
```

具体的状态类，以向站立为例，如果键盘持续按下。

```
public class StandState implements PlayerState{
    void stand(Hero hero)
    {
        //no nothing
    }
    void walkLeft(Hero hero)
    {
        if(keyboard press A)
        {
            hero.setPlayerState(Hero.STAND);
            hero.setPicture("/HeadLeft.jpg");
        }
    }
    void walkUp(Hero hero)
    {
        if(keyboard press W)
        {
            hero.setPlayerState(Hero.WALKUP);
            hero.setPicture("/HeadUp.jpg");
        }
    }
    void walkDown(Hero hero)
    {
        if(keyboard press S)
        {
            hero.setPlayerState(Hero.WALKDOWN);
```

```
            hero.setPicture("/HeadDown.jpg");
        }
    }
    void walkRight(Hero hero)
    {
        if(keyboard press D)
        {
            hero.setPlayerState(Hero.WALKRIGHT);
            hero.setPicture("/HeadRight.jpg");
        }
    }
}
```

这样就完成了行走状态的设计。采用状态模式将实现细节封装在各个不同的状态类中，状态转换交给状态类自己去实现，外部无需关心。去除了大量的判读逻辑，代码可读性更好了。今后也可以实现状态的新增，而无需对环境类（Hero）做出修改。