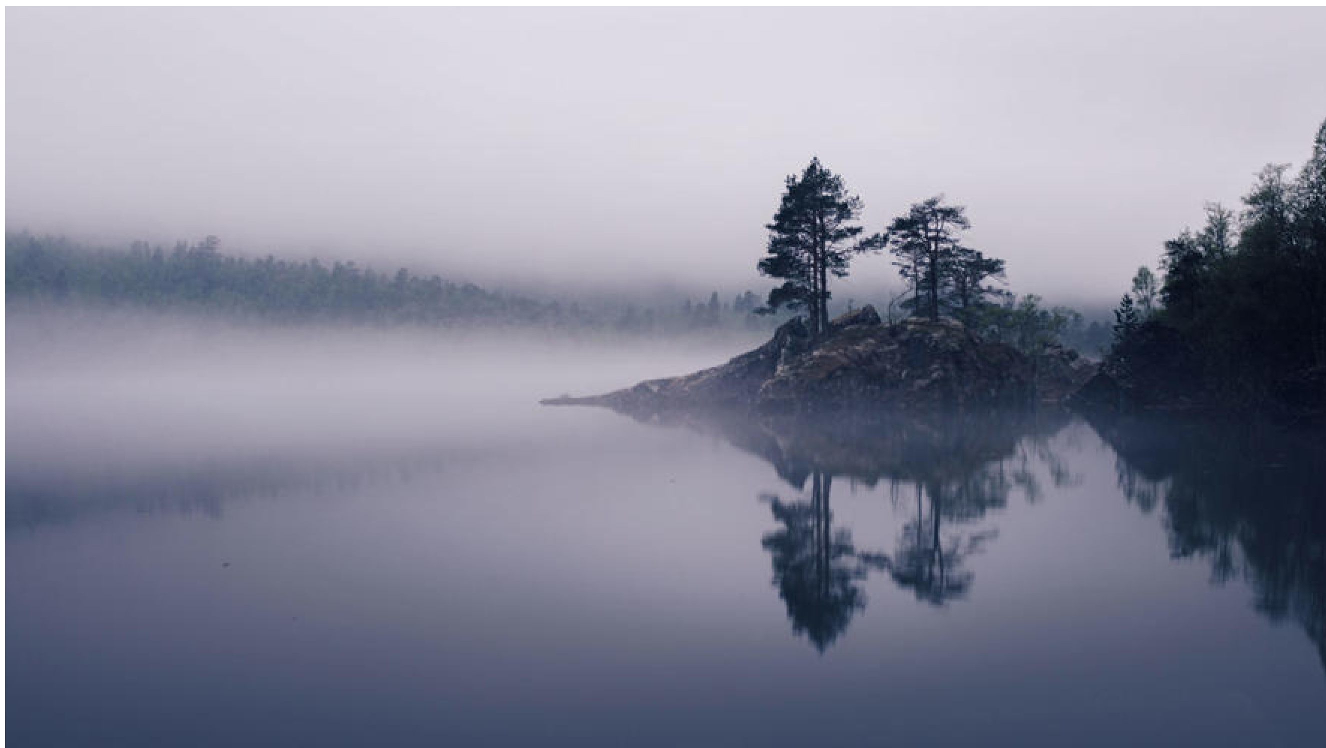
设计模式（装饰者模式）- 扩展原有功能

作者：Pleuvoir

原文链接：https://ld246.com/article/1522494348924

来源网站：链滴

许可协议：署名-相同方式共享 4.0 国际 (CC BY-SA 4.0)



装饰器模式（Decorator Pattern）允许向一个现有的对象添加新的功能，同时又不改变其结构。这类型的设计模式属于结构型模式，它是作为现有的类的一个包装。这种模式创建了一个装饰类，用来装原有的类，并在保持类方法签名完整性的前提下，提供了额外的功能。

怎么做?

1. 新建一个类，该类耦合了需要装饰的类，将实现传递进来，保证原有功能的完整性。为了表明该类某接口的包装类，所以一般声明为abstract类，即使该类并无抽象方法。
2. 新建一个需要扩充功能的类，继承上面新建的抽象类，增加新扩展的功能即可。

代码示例：

接口的抽象包装类

```
/**
 * {@link org.springframework.core.env.Environment} 接口的抽象包装类。
 * @author pleuvoir
 */
public abstract class AbstractEnvironmentWrap implements Environment {

    protected Environment environment;

    public AbstractEnvironmentWrap(Environment environment) {
        this.environment = environment;
    }
```

```
@Override
public boolean containsProperty(String key) {
    return environment.containsProperty(key);
}

@Override
public String getProperty(String key) {
    return environment.getProperty(key);
}

@Override
public String getProperty(String key, String defaultValue) {
    return environment.getProperty(key, defaultValue);
}

@Override
public <T> T getProperty(String key, Class<T> targetType) {
    return environment.getProperty(key, targetType);
}

@Override
public <T> T getProperty(String key, Class<T> targetType, T defaultValue) {
    return environment.getProperty(key, targetType, defaultValue);
}

@Override
public <T> Class<T> getPropertyAsClass(String key, Class<T> targetType) {
    return environment.getPropertyAsClass(key, targetType);
}

@Override
public String getRequiredProperty(String key) throws IllegalStateException {
    return environment.getRequiredProperty(key);
}

@Override
public <T> T getRequiredProperty(String key, Class<T> targetType)
        throws IllegalStateException {
    return environment.getRequiredProperty(key, targetType);
}

@Override
public String resolvePlaceholders(String text) {
    return environment.resolvePlaceholders(text);
}

@Override
public String resolveRequiredPlaceholders(String text)
        throws IllegalArgumentException {
    return environment.resolveRequiredPlaceholders(text);
}

@Override
```

```
    public String[] getActiveProfiles() {
        return environment.getActiveProfiles();
    }

    @Override
    public String[] getDefaultProfiles() {
        return environment.getDefaultProfiles();
    }

    @Override
    public boolean acceptsProfiles(String... profiles) {
        return environment.acceptsProfiles(profiles);
    }

}
```

需要扩充功能的实现类

```
/**
 * {@link org.springframework.core.env.Environment} 包装类，提供了更为简便的数据获取方法
 * @author pleuvoir
 *
 */
public class EnvironmentWrap extends AbstractEnvironmentWrap {

    public EnvironmentWrap(Environment environment) {
        super(environment);
    }

    /**
     * 获取Integer类型的数据，若数据格式不可转换为Integer类型，将抛出异常{@link NumberFor
atException}
     * @param key
     * @return
     */
    public Integer getInteger(String key) {
        String val = super.getProperty(key);
        return Integer.valueOf(val);
    }

    /**
     * 获取Integer类型的数据，并提供默认的值，若数据格式不可转换为Integer类型或者为null时，
会返回默认值
     * @param key
     * @param defaultVal
     * @return
     */
    public Integer getInteger(String key, Integer defaultVal) {
        Integer i = null;
        try {
            i = getInteger(key);
        } catch (NumberFormatException e) {
        }
        if (i == null) {
```

```
            i = defaultVal;
        }
        return i;
    }

    /**
     * 获取Long类型的数据，若数据格式不可转换为Long类型，将抛出异常{@link NumberFormatEx
eption}
     * @param key
     * @return
     */
    public Long getLong(String key) {
        String val = super.getProperty(key);
        return Long.valueOf(val);
    }

    /**
     * 获取Long类型的数据，并提供默认的值，若数据格式不可转换为Long类型或者为null时，将会
回默认值
     * @param key
     * @param defaultVal
     * @return
     */
    public Long getLong(String key, Long defaultVal) {
        Long i = null;
        try {
            i = getLong(key);
        } catch (NumberFormatException e) {
        }
        if (i == null) {
            i = defaultVal;
        }
        return i;
    }

    /**
     * 获取Double类型的数据，若数据格式不可转换为Double类型，将抛出异常{@link NumberFor
atException}
     * @param key
     * @return
     */
    public Double getDouble(String key) {
        String val = super.getProperty(key);
        return Double.valueOf(val);
    }

    /**
     * 获取Double类型的数据，并提供默认的值，若数据格式不可转换为Double类型或者为null时，
会返回默认值
     * @param key
     * @param defaultVal
     * @return
     */
    public Double getDouble(String key, Double defaultVal) {
```

```
        Double i = null;
        try {
            i = getDouble(key);
        } catch (NumberFormatException e) {
        }
        if (i == null) {
            i = defaultVal;
        }
        return i;
    }

    /**
     * 获取BigDecimal类型的数据，若数据格式不可转换为BigDecimal类型，将抛出异常
     * {@link NumberFormatException}
     * @param key
     * @return
     */
    public BigDecimal getBigDecimal(String key) {
        String val = super.getProperty(key);
        return new BigDecimal(val);
    }

    /**
     * 获取BigDecimal类型的数据，并提供默认的值，若数据格式不可转换为BigDecimal类型或者为n
ll时，将会返回默认值
     * @param key
     * @param defaultVal
     * @return
     */
    public BigDecimal getBigDecimal(String key, BigDecimal defaultVal) {
        BigDecimal i = null;
        try {
            i = getBigDecimal(key);
        } catch (NumberFormatException e) {
        }
        if (i == null) {
            i = defaultVal;
        }
        return i;
    }

    /**
     * 获取Boolean类型的数据，若数据格式不可转换为Boolean类型，将返回false
     * @param key
     * @return
     */
    public Boolean getBoolean(String key) {
        String val = super.getProperty(key);
        return Boolean.valueOf(val);
    }

    /**
     * 获取String类型的数据
     * @param key
```

```
     * @return
     */
    public String getString(String key) {
        return super.getProperty(key);
    }

}
```

实际上除了扩展原有功能外，还可以修改原有方法的实现，在原有方法前后增加新的执行方法，这种用方法有点代理模式的感觉。

建议：如果只是扩展新功能（提供新方法），类的后缀使用Wrap，如果对原有方法进行了修改则使用ecorator。

结论

在不修改原有类的基础上进行扩展功能，可代替继承。特别适合对一些jar包或者不能修改源代码功能扩展。如果多装饰几次，就会变得无比的麻烦，可参考java IO的实现方式。

原文链接：设计模式（装饰者模式）- 扩展原有功能