Java 的设计模式

作者：Hefery

原文链接：https://ld246.com/article/1642959926230

来源网站：链滴

设计模式

软件设计

UML

UML基础

定义：UML，Unified Modeling Language 统一建模语言
特点：用于说明、可视化、构建和编写一个正在开发的面向对象的、软件密集系统的制品的开放方法
分类：

- 结构式：强调的是系统式的建模

静态图(**类图**、对象图、包图)、实现图(组件图、部署图)、剖面图、符合结构图
- 行为式：强调系统模型中触发的事件

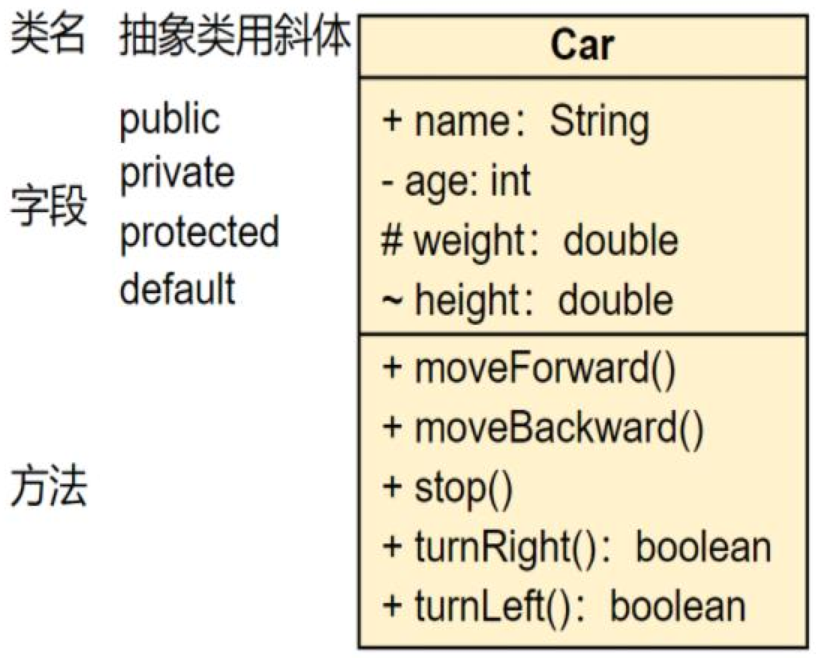
活动图、状态图、**用例图**
- 交互式：强调系统模型中资料流程

通信图、交互概述图、**时序图**、时间图

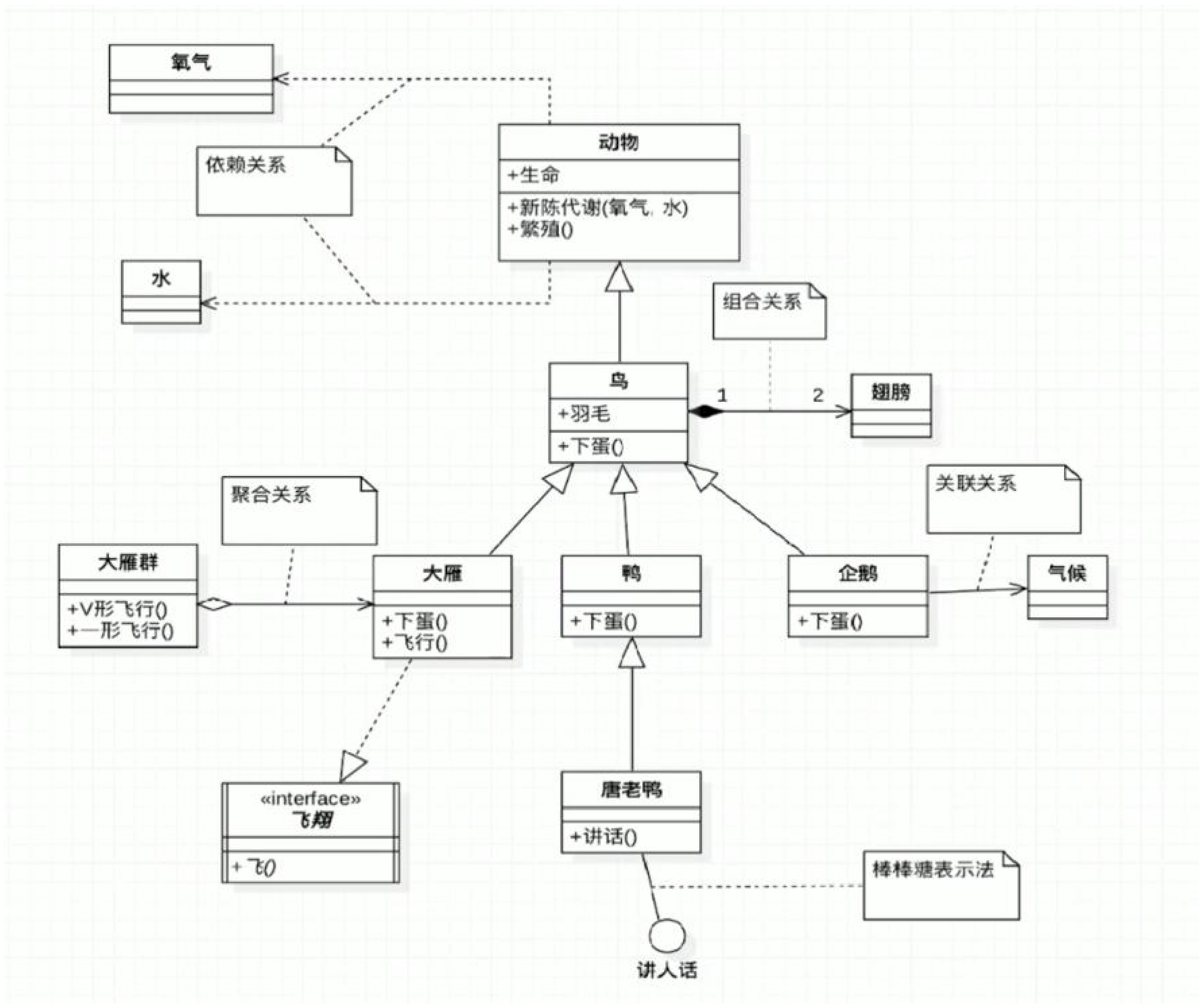
UML类图

定义：Class Diagram
作用：表示类、接口、实例等之间相互的静态关系
特点：

- 箭头：
  - 方向：子类 -> 父类（定义子类时需要通过extends关键字指定父类，子类一定是知道父类定义，但父类并不知道子类的定义，只有知道对方信息才能指向）
  - 空心三角箭头：继承或实现
  - 空心菱形：聚合 大雁群->大雁 （has a）整体和局部的关系，两者有着独自的生命周期
  - 实心菱形：组合 鸟->翅膀 （contains a）整体和局部的关系，两者有着相同的生命周期
- 线条：
  - 实线：继承父类，关联（表示一个类对象和另一个类对象有关联，通常是一个类中有另一个类象做为属性）
  - 虚线：实现接口，依赖（表示一种使用关系，一个类需要借助另一个类来实现功能，一般是一类使用另一个类做为参数使用，或作为返回值）



类名:
    抽象类：斜体
    接口：<接口名>
类属性：[+ | - | # | ~] 属性名:属性类型
    +：public
    -：private
    #：protected
    ~；default
    下划线：static
类行为：[+ | - | # | ~] 行为名[:属性类型]
    +：public
    -：private
    #：protected
    ~；default
    下划线：static



依赖关系：
　氧气 、水 <-- 动物：虚线-尖括号箭头，氧气 、水作为动物新陈代谢方法的入参
关联关系：
　气候 <-- 企鹅：实线-尖括号箭头，两个类之间存在关联，通常是一个类(气候)作为参数存在于另个类(企鹅)中

继承关系：
　动物 <-- 鸟 <-- 大雁、鸭、企鹅：实线-空心三角箭头
实现关系：
　飞翔 <-- 大雁：虚线-空心三角箭头

组合关系：
　翅膀 <-- 鸟：实线-实心菱形箭头，contains a关系
聚合关系：
　大雁 <-- 大雁群：实线-空心菱形，has a关系

UML时序图

作用：对象之间交互，对象按时间顺序排列

元素：

- 对象Actor：
- 生命线Lifeline：
- 控制焦点Focus of control：
- 消息Message：

特点：

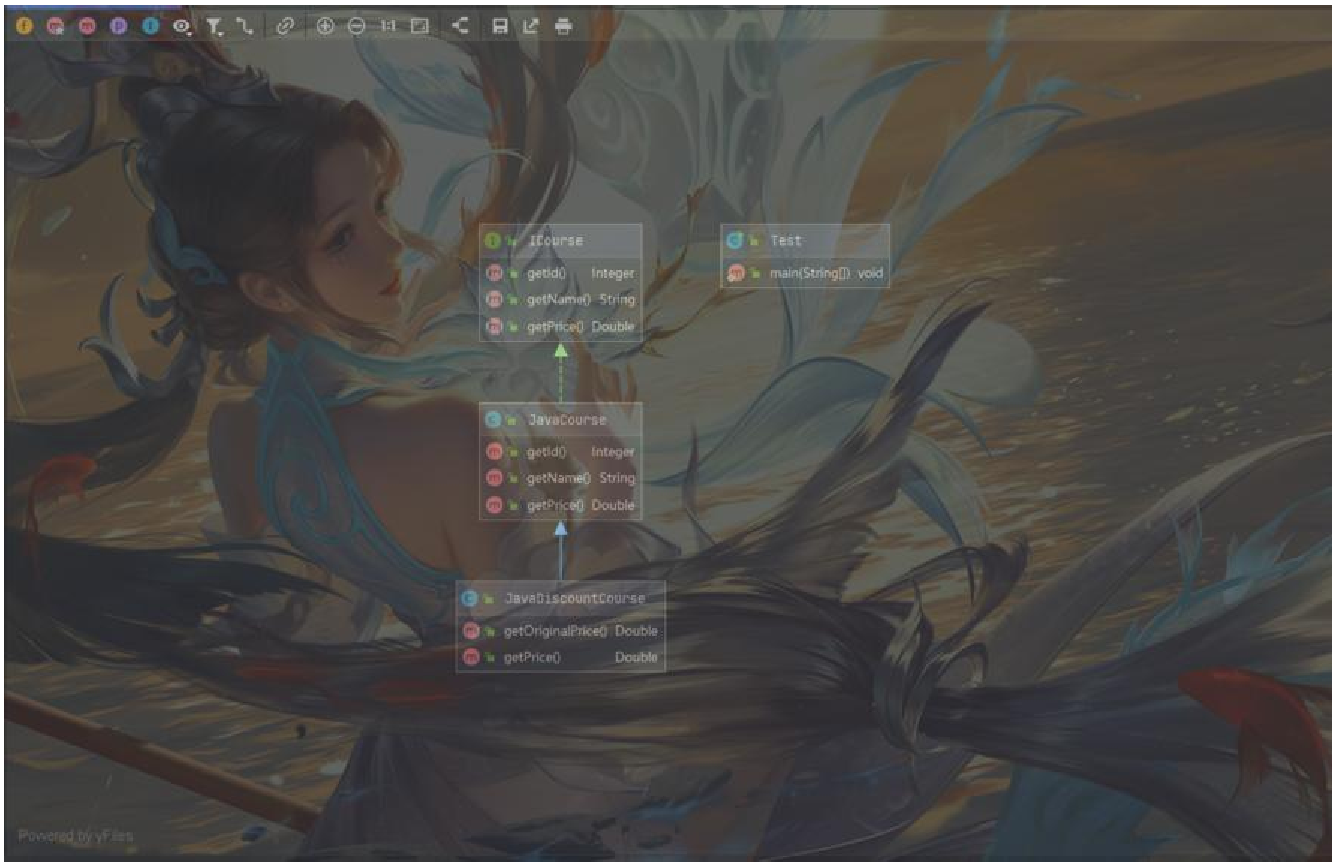
- 箭头：
  - 实线实心箭头：同步调用
  - 实线空心箭头：异步调用
  - 虚线空心箭头：返回

UML用例图

设计原则

开闭原则

- 定义：一个软件实体如类、模块、函数应该对拓展开放，对修改关闭
- 核心：用抽象拓展框架（ICourse-JavaCourse），用实现拓展细节(JavaCourse-JavaDiscountCourse)
- 优点：提高软件系统的可复用性和可维护性



ICourse接口有一个JavaCpurse实现，如果需要拓展一个折扣价，就再创建一个JavaDiscountCours
类去实现拓展
需要使用折扣价时，就new JavaDiscountCourse，该对象会有JavaCourse可用属性和方法，还有折价

依赖倒置

- 定义：高层模块不应该依赖低层模块，二者都应该依赖其抽象
- 核心：

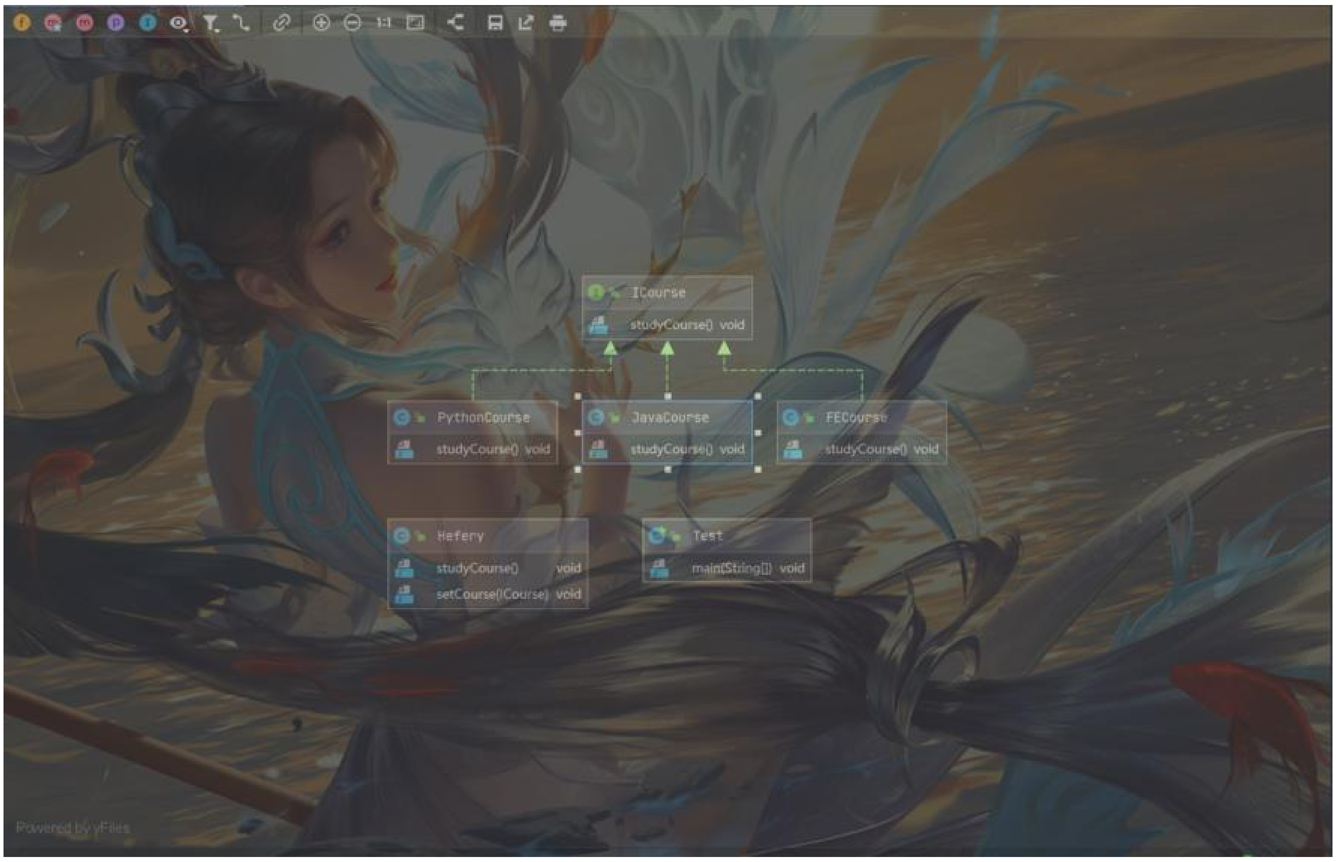
抽象不应该依赖细节

细节应该依赖抽象

针对接口编程（ICourse-JavaCourse、PythonCourse），不要针对实现编程（ICourse-Hefery）

- 优点：

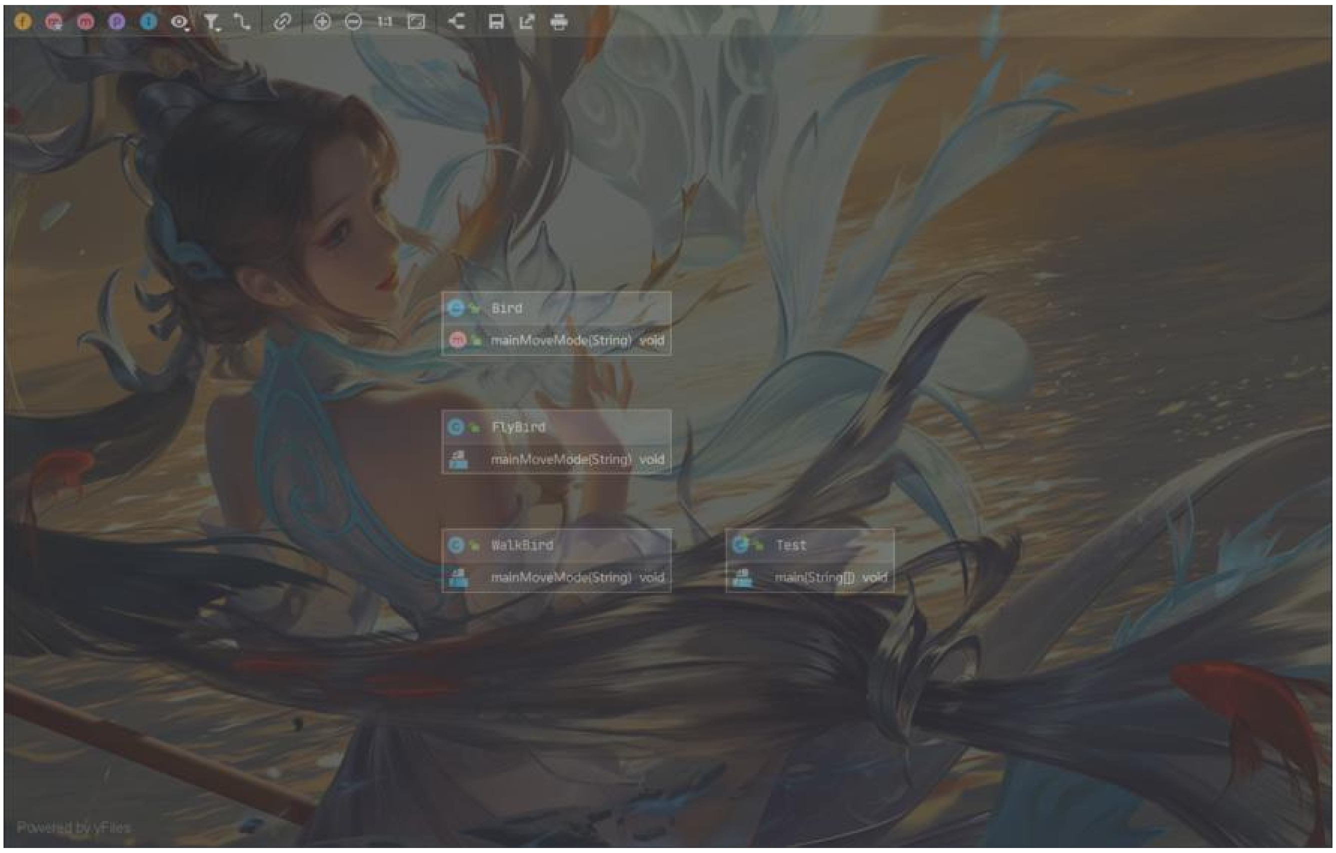
可以减少类间的耦合性、提高系统稳定性，提高代码可读性和可维护性，可降低修改程序所造成的风险



高层（Hefery）不依赖低层（ICourse），针对接口ICourse编程

单一职责

- 定义：不要存在多于一个导致类变更的原因
- 核心：一个类/接口/方法只负责一项职责
- 优点：降低类的复杂度、提高类的可读性，提高系统的可维护性、降低变更引起的风险


Bird
mainMoveMode(String) void
FlyBird
mainMoveMode(String) void
WalkBird
mainMoveMode(String) void
Test
main(String[]) void
Powered by yFiles

接口隔离

- 定义：用多个专门的接口，而不使用单一的总接口，客户端不应该依赖它不需要的接口
- 核心：

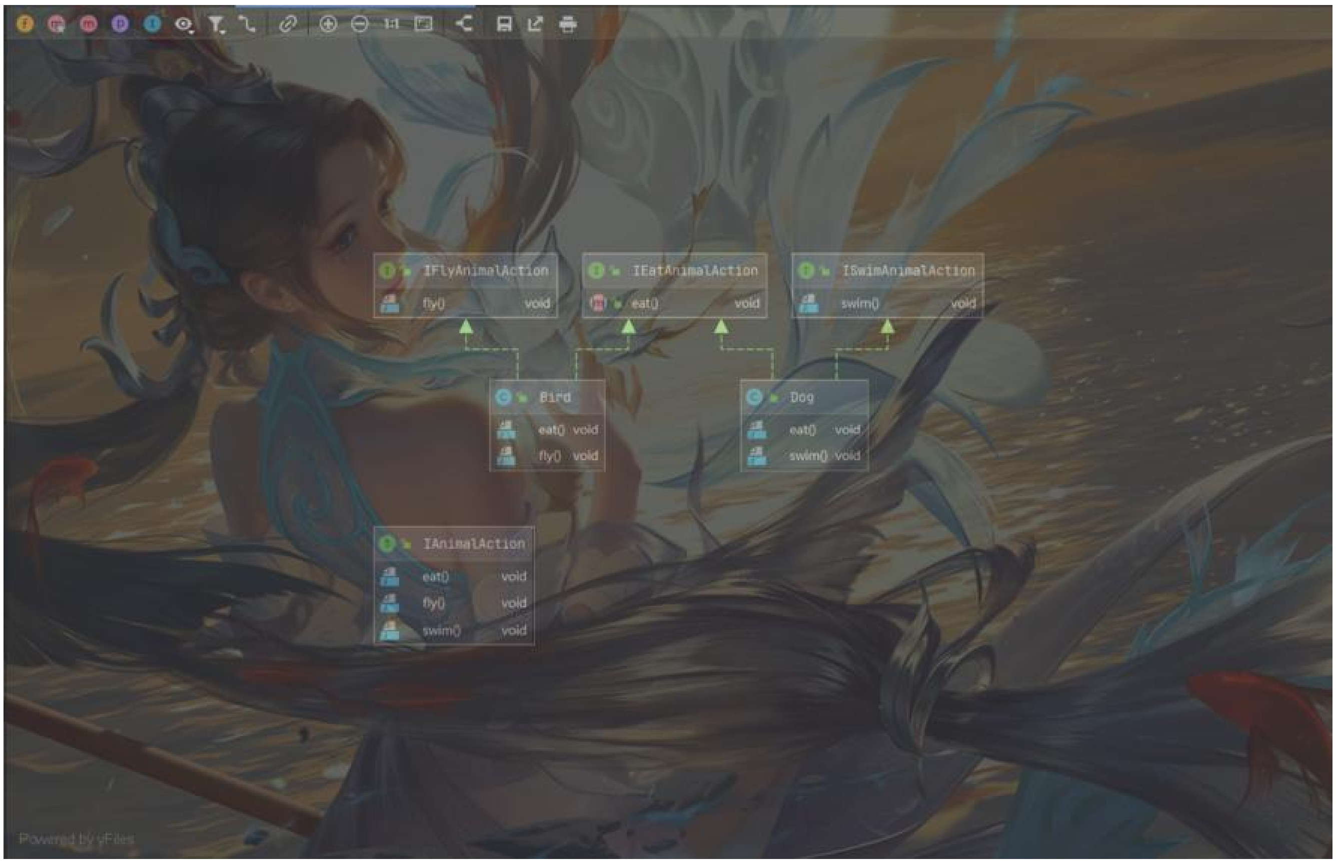
一个类对一个类的依赖应该建立在最小的接口上

建立单一接口，不要建立庞大臃肿的接口

尽量细化接口，接口中的方法尽量少

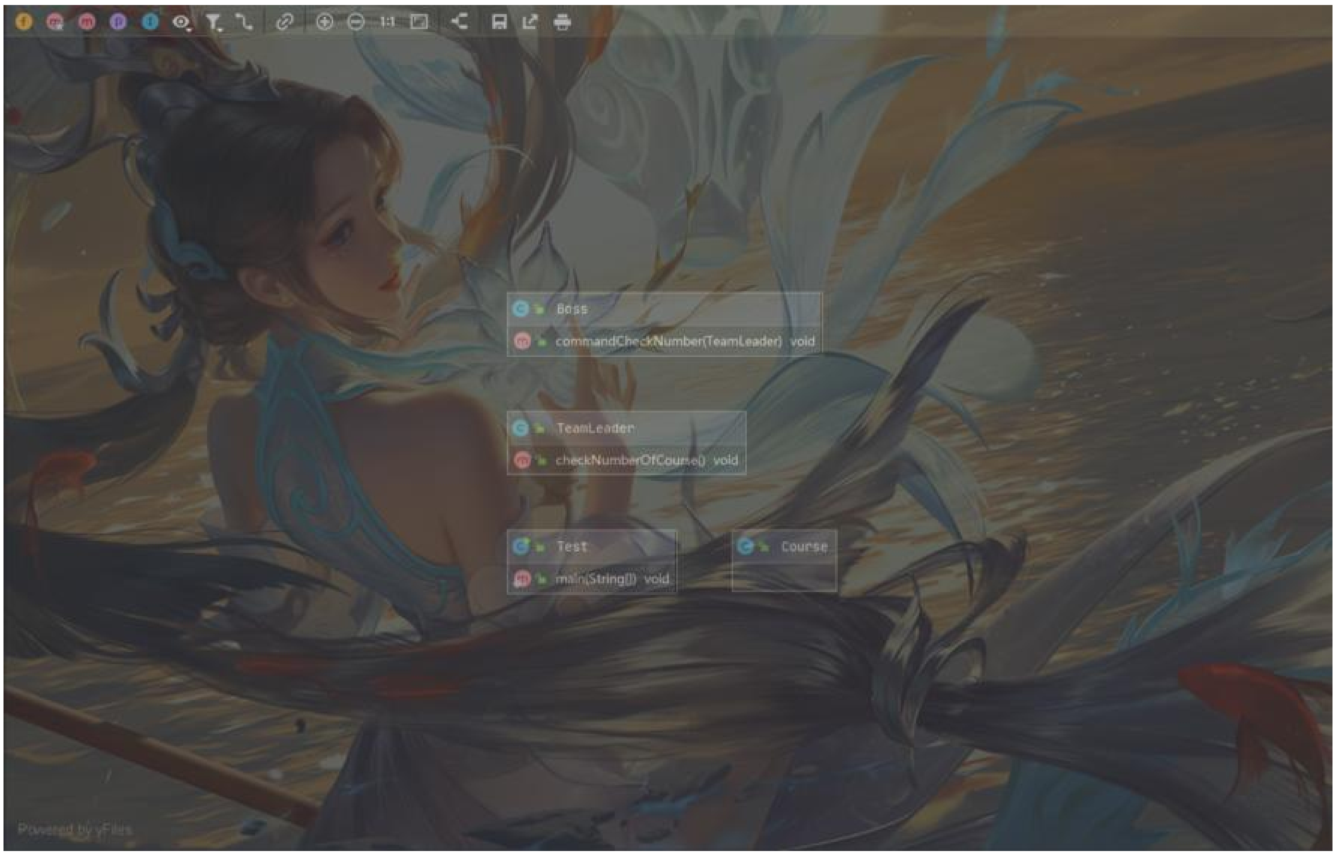
注意适度原则，一定要适度

- 优点：符合我们常说的高内聚低耦合的设计思想从而使得类具有很好的可读性、可扩展性和可维护性



迪米特原则

- 定义：一个对象应该对其他对象保持最少的了解。又叫最少知道原则
- 核心：尽量降低类与类之间的耦合
- 优点：降低类之间的耦合


Boss
commandCheckNumber(TeamLeader) void
TeamLeader
checkNumberOfCourse() void
Test
main(String[]) void
Course
Powered by yFiles

原文链接：Java 的设计模式

Boss想要检查数量，只需要向TeamLeader下达指令，由TeamLeader完成

里氏替换

- 定义：子类应该可以完全替换父类。也就是说在使用继承时，只扩展新功能，而不要破坏父类原有功能
- 核心：
  - 子类必须实现父类的抽象方法，但不得重写（覆盖）父类的非抽象（已实现）方法
  - 子类中可以增加自己特有的方法
  - 当子类覆盖或实现父类的方法时，方法的前置条件（即方法的形参）要比父类方法的输入参数宽松
  - 当子类的方法实现父类的抽象方法时，方法的后置条件（即方法的返回值）要比父类更严格
- 优点：规范继承时子类的一些书写规则，保持父类方法不被覆盖

合成复用

创建型模式

工厂方法模式

简单工厂

基础解析

- 定义：由一个工厂对象决定创建出哪一种产品类的实例
- 类型：创建型，但不属于GOF23种设计模式
- 场景：

工厂类负责创建的对象比较少

客户端（应用层）只知道传入工厂类的参数，对于如何创建对象（逻辑）不关心

- 优点：

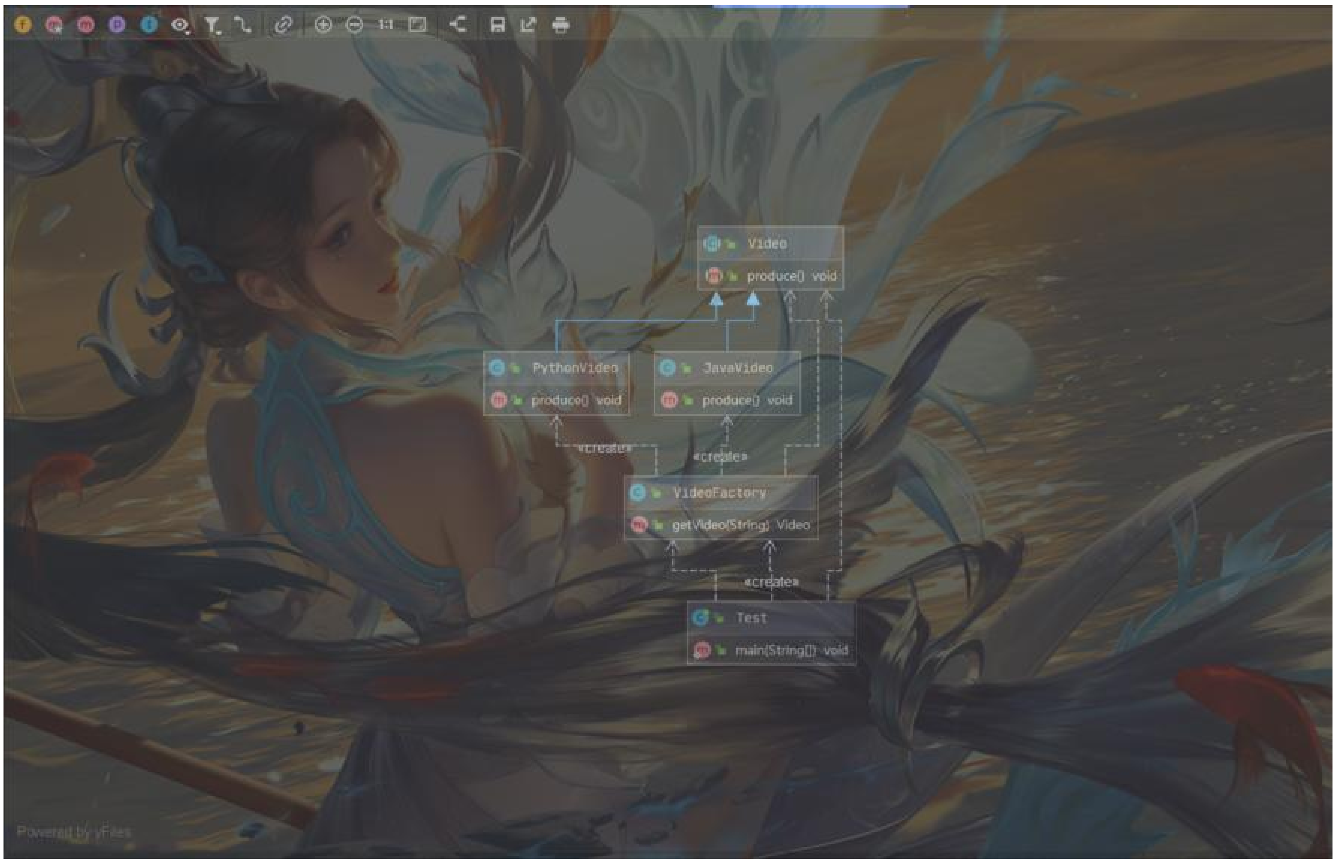
只需要传入一个正确的参数，就可以获取你所需要的对象而无须知道其创建细节

- 缺点：

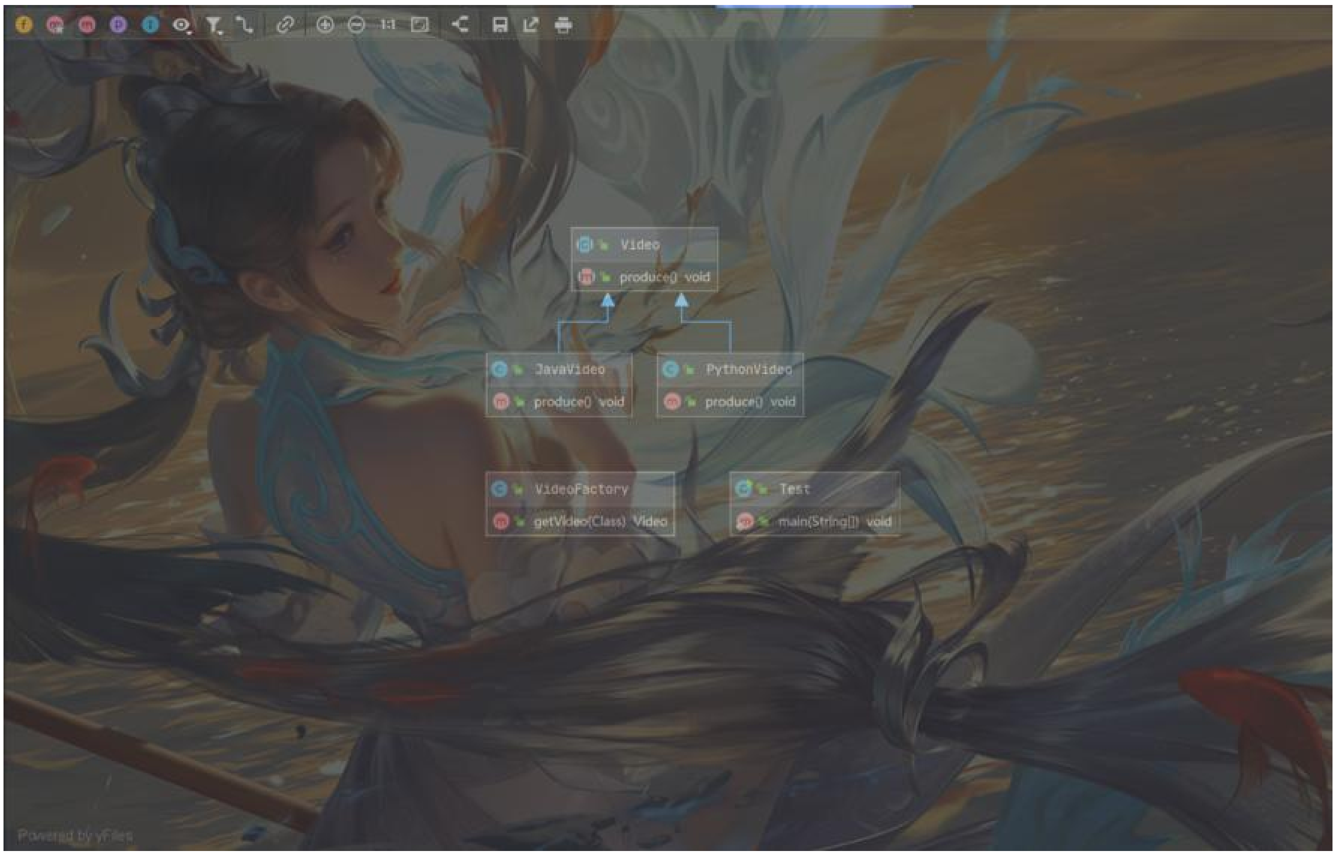
生产过程需要修改时，也要来修改此工厂，这个类不止一个引起修改的原因，违背了单一职责原则

工厂类的职责相对过重，增加新的产品需要修改工厂类的判断逻辑，违背开闭原则

场景解析



应用层Test不直接new XXXVideo，而是通过一个VideoFactory获取应用层需求，底层XXXVideo构所需类

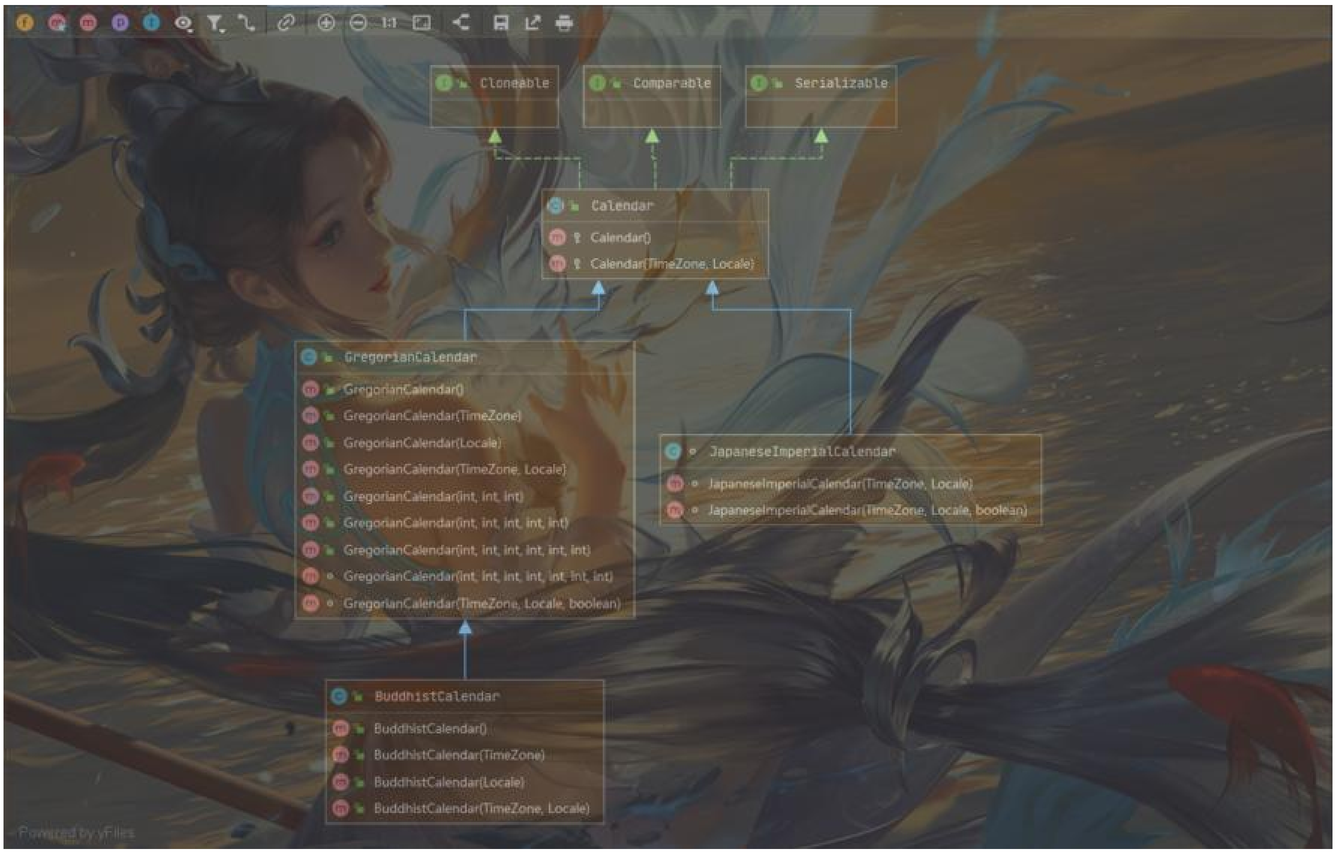


抽象类Video由JavaVide和PythonVideo实现器produce方法

VideoFactory有getVideo方法传入的Class参数去拿到是使用JavaVide

源码解析

```
public static Calendar getInstance(Locale aLocale) {
        return createCalendar(TimeZone.getDefault(), aLocale);
}

private static Calendar createCalendar(TimeZone zone, Locale aLocale) {
        CalendarProvider provider =
            LocaleProviderAdapter.getAdapter(CalendarProvider.class, aLocale)
                                 .getCalendarProvider();
        if (provider != null) {
            try {
                return provider.getInstance(zone, aLocale);
            } catch (IllegalArgumentException iae) {
                // fall back to the default instantiation
            }
        }

        Calendar cal = null;

        if (aLocale.hasExtensions()) {
            String caltype = aLocale.getUnicodeLocaleType("ca");
            if (caltype != null) {
                switch (caltype) {
                case "buddhist":
                cal = new BuddhistCalendar(zone, aLocale);
                    break;
                case "japanese":
                    cal = new JapaneseImperialCalendar(zone, aLocale);
                    break;
                case "gregory":
                    cal = new GregorianCalendar(zone, aLocale);
                    break;
                }
            }
        }
        if (cal == null) {
            // If no known calendar type is explicitly specified,
            // perform the traditional way to create a Calendar:
            // create a BuddhistCalendar for th_TH locale,
            // a JapaneseImperialCalendar for ja_JP_JP locale, or
            // a GregorianCalendar for any other locales.
            // NOTE: The language, country and variant strings are interned.
            if (aLocale.getLanguage() == "th" && aLocale.getCountry() == "TH") {
                cal = new BuddhistCalendar(zone, aLocale);
            } else if (aLocale.getVariant() == "JP" && aLocale.getLanguage() == "ja"
                       && aLocale.getCountry() == "JP") {
                cal = new JapaneseImperialCalendar(zone, aLocale);
            } else {
                cal = new GregorianCalendar(zone, aLocale);
            }
        }
        return cal;
    }
```



Cloneable
Comparable
Serializable
Calendar
Calendar()
Calendar(TimeZone, Locale)
GregorianCalendar
GregorianCalendar()
GregorianCalendar(TimeZone)
GregorianCalendar(Locale)
GregorianCalendar(TimeZone, Locale)
GregorianCalendar(int, int, int)
GregorianCalendar(int, int, int, int, int)
GregorianCalendar(int, int, int, int, int, int)
GregorianCalendar(int, int, int, int, int, int, int)
GregorianCalendar(TimeZone, Locale, boolean)
JapaneseImperialCalendar
JapaneseImperialCalendar(TimeZone, Locale)
JapaneseImperialCalendar(TimeZone, Locale, boolean)
BuddhistCalendar
BuddhistCalendar()
BuddhistCalendar(TimeZone)
BuddhistCalendar(Locale)
BuddhistCalendar(TimeZone, Locale)
Powered by yFiles

工厂方法

基础解析

- 定义：定义一个创建对象的接口，规定每个产品都有一个专属工厂
- 类型：创建型
- 场景

创建对象需要大量重复的代码

客户端（应用层）不依赖产品类实例如何被创建、实现等细节

一个类通过其子类来指定创建哪个对象

- 优点

用户只需要关心所需产品对应的工厂，无须关心创建细节

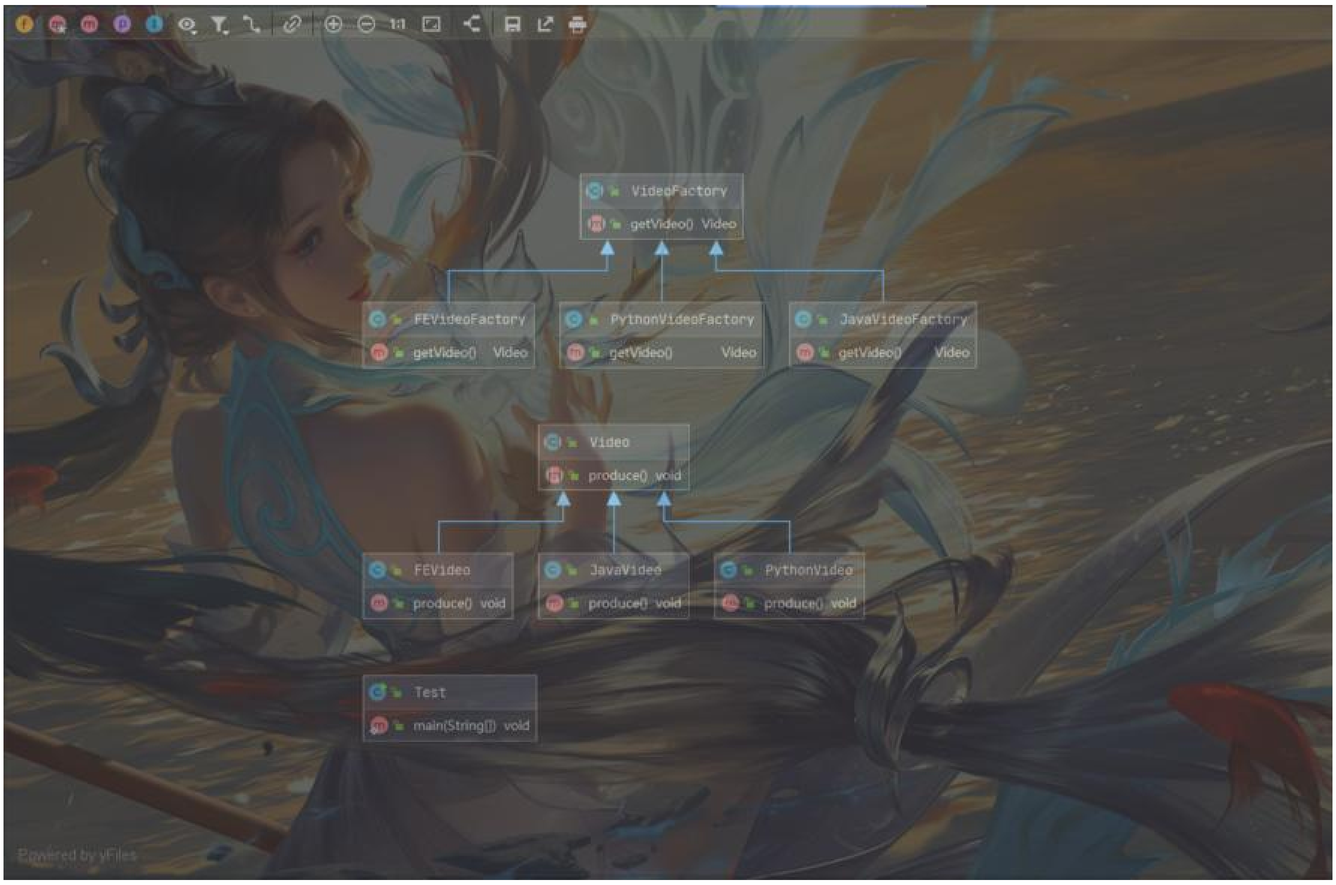
加入新产品符合开闭原则，提高可扩展性

- 缺点

类的个数容易过多，增加复杂度

增加了系统的抽象性和理解难度

场景解析



XXFactoryVideo --> XXVideo --> Video

产品：Video　具体产品：XXVideo
工厂：Factory　具体工厂：XXVideoFactory

源码解析

```
// 工厂
// Collection == VideoFactory
public interface Collection<E> extends Iterable<E> {
    ......
    // 工厂方法
    Iterator<E> iterator();
    ......
}

// 具体的实现工厂
// ArrayList == XXFactoryVideo
public class ArrayList<E> extends AbstractList<E> implements List<E>, RandomAccess, Clone
ble, java.io.Serializable {
    public Iterator<E> iterator() {
        // 生产具体产品
            return new Itr();
        }
}

// 具体产品
// Iterator == Video
// Itr == XXVideo
private class Itr implements Iterator<E> {
        int cursor;       // index of next element to return
        int lastRet = -1; // index of last element returned; -1 if no such
        int expectedModCount = modCount;

        Itr() {}

        public boolean hasNext() {
            return cursor != size;
        }

        @SuppressWarnings("unchecked")
        public E next() {
            checkForComodification();
            int i = cursor;
            if (i >= size)
                throw new NoSuchElementException();
            Object[] elementData = ArrayList.this.elementData;
            if (i >= elementData.length)
                throw new ConcurrentModificationException();
            cursor = i + 1;
            return (E) elementData[lastRet = i];
        }

        public void remove() {
            if (lastRet < 0)
                throw new IllegalStateException();
            checkForComodification();

            try {
                ArrayList.this.remove(lastRet);
                cursor = lastRet;
                lastRet = -1;
                expectedModCount = modCount;
            } catch (IndexOutOfBoundsException ex) {
                throw new ConcurrentModificationException();
            }
        }

        @Override
        @SuppressWarnings("unchecked")
        public void forEachRemaining(Consumer<? super E> consumer) {
            Objects.requireNonNull(consumer);
            final int size = ArrayList.this.size;
            int i = cursor;
            if (i >= size) {
                return;
            }
            final Object[] elementData = ArrayList.this.elementData;
            if (i >= elementData.length) {
                throw new ConcurrentModificationException();
            }
            while (i != size && modCount == expectedModCount) {
```

```
            consumer.accept((E) elementData[i++]);
        }
        // update once at end of iteration to reduce heap write traffic
        cursor = i;
        lastRet = i - 1;
        checkForComodification();
    }

    final void checkForComodification() {
        if (modCount != expectedModCount)
            throw new ConcurrentModificationException();
    }
}
```

抽象工厂模式

基础解析

- 定义：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，无须指定它们具体的类
- 类型：创建型
- 适用场景

客户端（应用层）不依赖于产品类实例如何被创建、实现等细节

强调一系列相关的产品对象（属于同一产品族）一起使用创建对象需要大量重复的代码

提供一个产品类的库，所有的产品以同样的接口出现，从而使客户端不依赖于具体实现

- 优点：

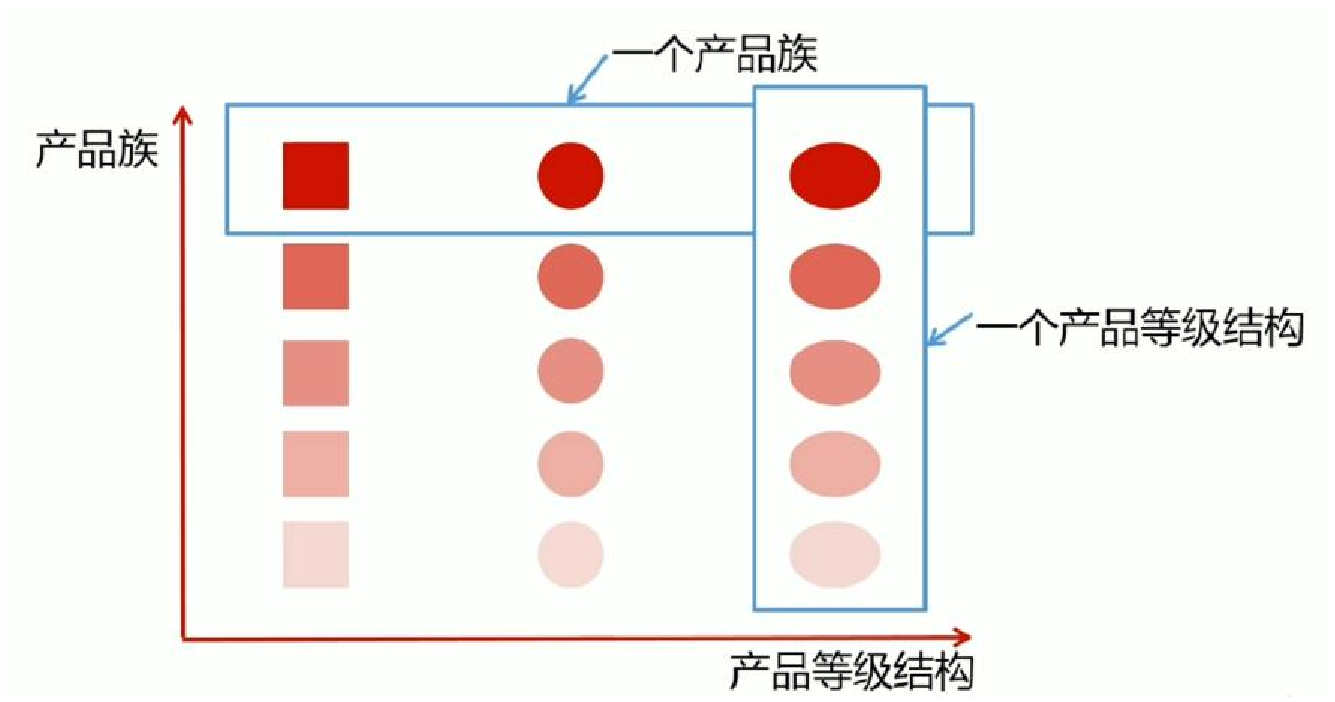
具体产品在应用层代码隔离，无须关心创建细节

将一个系列的产品族统一到一起创建

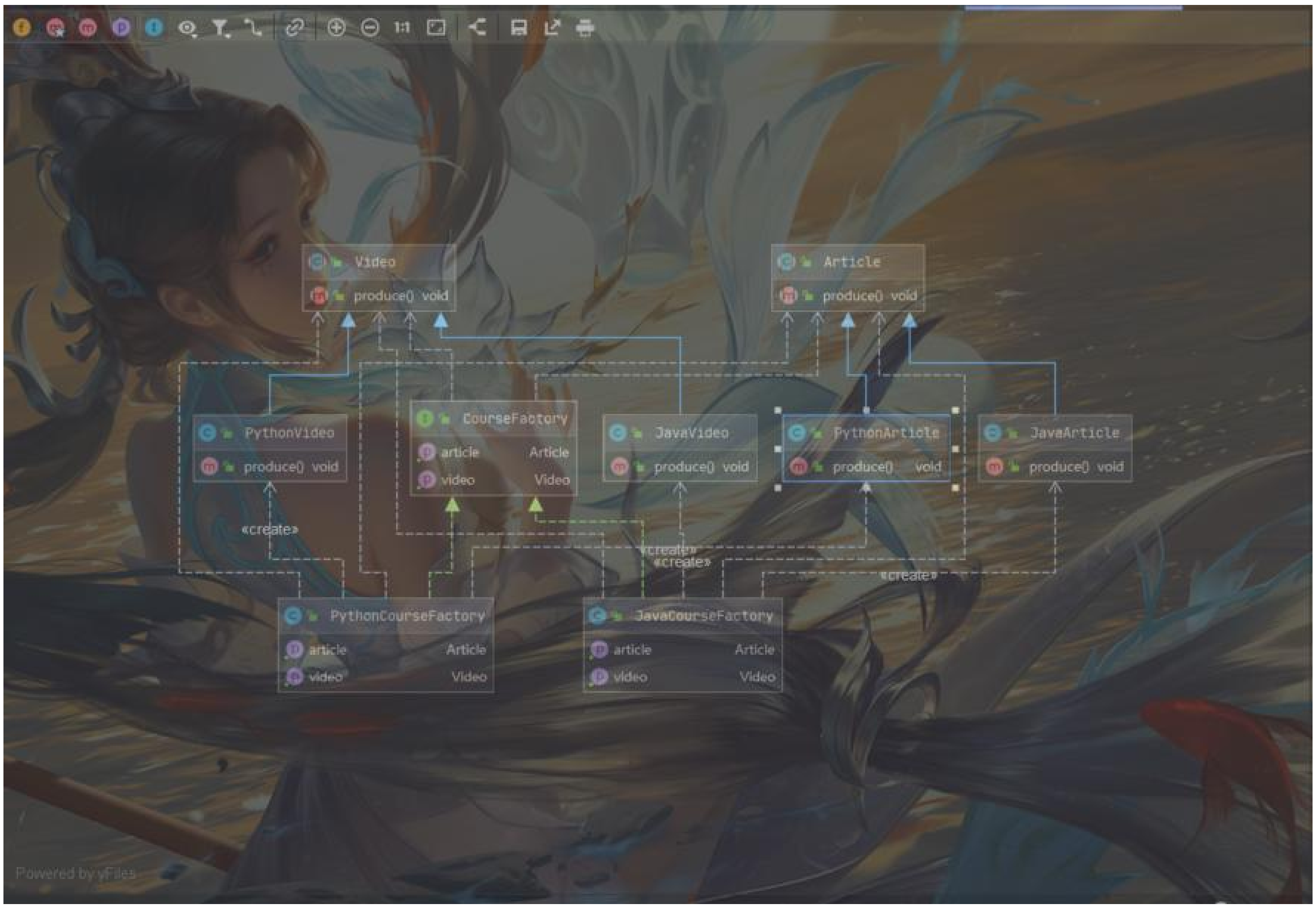
- 缺点：

规定了所有可能被创建的产品集合，产品族中扩展新的产品困难，需要修改抽象工厂的接口

增加了系统的抽象性和理解难度



场景解析



产品族：CourseFactory
产品：Video + Article
具体产品：XXVideo + XXArticle

源码解析

```
public interface Connection  extends Wrapper, AutoCloseable {
    // 同一产品族
    // 产品：ConnectionImpl
    Statement createStatement(int resultSetType, int resultSetConcurrency) throws SQLExcepti
n;
    PreparedStatement prepareStatement(String sql, int resultSetType, int resultSetConcurrenc
) throws SQLException;
}


public interface Statement extends Wrapper, AutoCloseable {
    // 同一产品族
    ResultSet executeQuery(String sql) throws SQLException;
    int executeUpdate(String sql) throws SQLException;
}

public interface SqlSessionFactory {
    // 同一产品族
    // 产品：DefaultSqlSessionFactory
    SqlSession XXX;
    Configuration  getConfiguration();
}
```

建造者模式

基础解析

- 定义：将一个复杂对象的构建和它的表示分离，使得同样的构建过程可以创建不同的表示
- 核心：用户只需指定需要建造的类型就可以得到想要的对象，建造得过程和细节无需了解
- 类型：创建型
- 适用场景：如果一个对象有非常复杂的内部结构（很多属性），想把复杂对象的创建和使用分离
- 优点：

封装性好，创建和使用分离

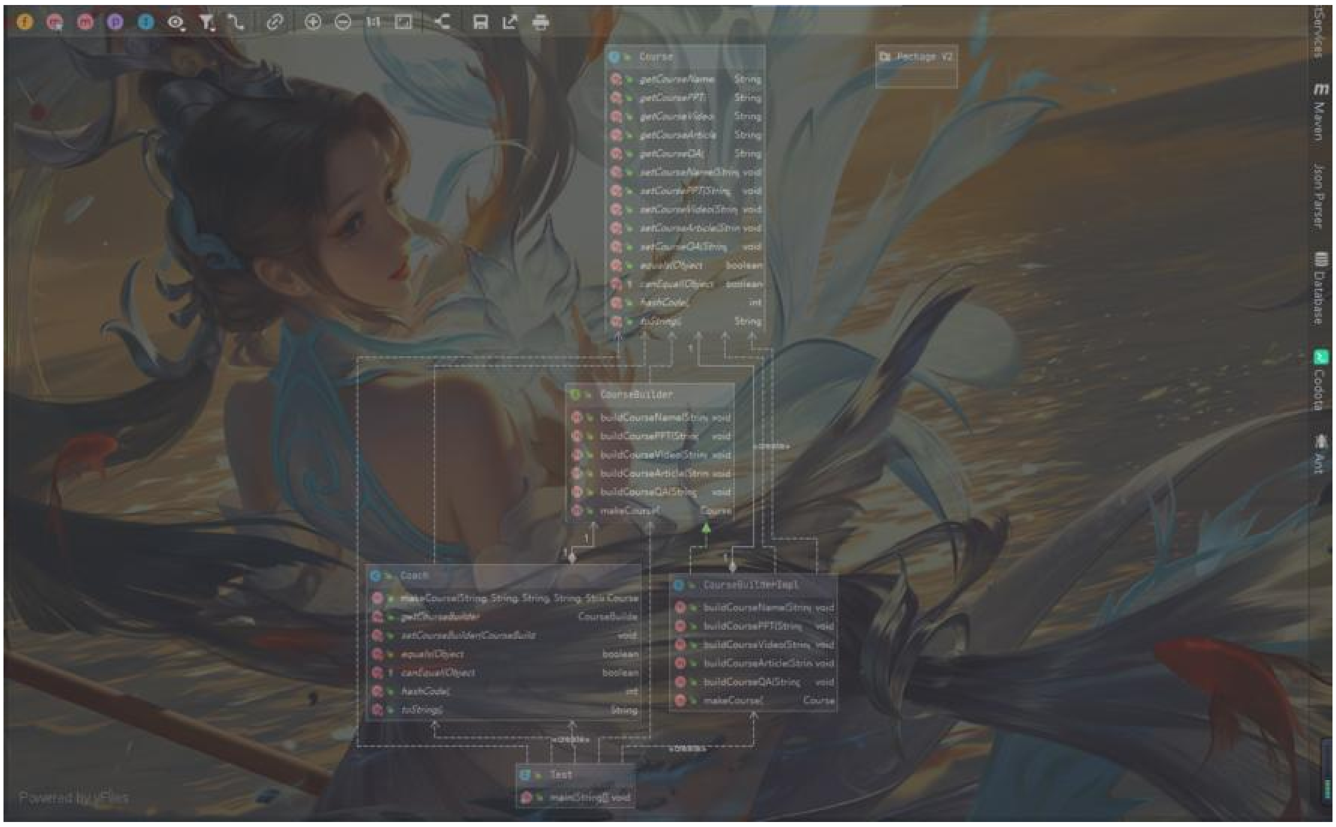
扩展性好，建造类之间相互独立，实现一定程度上解耦

- 缺点：

产生多余的Builder对象

产品内部发生变化，建造者都要修改，成本较大

场景解析


Course
getCourseName() String
getCoursePPT() String
getCourseVideo() String
getCourseArticle() String
getCourseQA() String
setCourseName(String) void
setCoursePPT(String) void
setCourseVideo(String) void
setCourseArticle(String) void
setCourseQA(String) void
equals(Object) boolean
canEqual(Object) boolean
hashCode() int
toString() String
Package V2
CourseBuilder
buildCourseName(String) void
buildCoursePPT(String) void
buildCourseVideo(String) void
buildCourseArticle(String) void
buildCourseQA(String) void
makeCourse() Course
«create»
Coach
makeCourse(String, String, String, String, String) Course
getCourseBuilder() CourseBuilder
setCourseBuilder(CourseBuilder) void
equals(Object) boolean
canEqual(Object) boolean
hashCode() int
toString() String
CourseBuilderImpl
buildCourseName(String) void
buildCoursePPT(String) void
buildCourseVideo(String) void
buildCourseArticle(String) void
buildCourseQA(String) void
makeCourse() Course
«create»
«create»
Test
main(String[]) void
Powered by yFiles
Services
Maven
Json Parser
Database
Codota
Ant

Test --> Coach --> CourseBuilder --> CourseBuilderImpl --> Course

源码解析

```
public final class StringBuilder extends AbstractStringBuilder implements java.io.Serializable,
harSequence {
    @Override
    public StringBuilder append(String str) {
        super.append(str);
        return this;
    }

    @Override
    public StringBuilder append(boolean b) {
        super.append(b);
        return this;
    }

    @Override
    public StringBuilder append(char c) {
        super.append(c);
        return this;
    }

    @Override
    public StringBuilder append(int i) {
        super.append(i);
        return this;
    }
    @Override
    public StringBuilder append(long lng) {
        super.append(lng);
        return this;
    }

    @Override
    public StringBuilder append(float f) {
        super.append(f);
        return this;
    }

    @Override
    public StringBuilder append(double d) {
        super.append(d);
        return this;
    }
}

public final class StringBuffer extends AbstractStringBuilder implements java.io.Serializable, C
arSequence {
    @Override
    public synchronized StringBuffer append(Object obj) {
        toStringCache = null;
        super.append(String.valueOf(obj));
        return this;
    }

    @Override
    public synchronized StringBuffer append(String str) {
        toStringCache = null;
        super.append(str);
        return this;
    }
}

public class SqlSessionFactoryBuilder {
    public SqlSessionFactoryBuilder() {
    }

    public SqlSessionFactory build(Reader reader) {
        return this.build((Reader)reader, (String)null, (Properties)null);
    }

    public SqlSessionFactory build(Reader reader, String environment) {
        return this.build((Reader)reader, environment, (Properties)null);
    }

    public SqlSessionFactory build(Reader reader, Properties properties) {
        return this.build((Reader)reader, (String)null, properties);
    }

    public SqlSessionFactory build(Reader reader, String environment, Properties properties) {
        SqlSessionFactory var5;
        try {
            XMLConfigBuilder parser = new XMLConfigBuilder(reader, environment, properties);
            var5 = this.build(parser.parse());
        } catch (Exception var14) {
            throw ExceptionFactory.wrapException("Error building SqlSession.", var14);
        } finally {
            ErrorContext.instance().reset();

            try {
                reader.close();
            } catch (IOException var13) {
            }

        }

        return var5;
    }

    public SqlSessionFactory build(InputStream inputStream) {
        return this.build((InputStream)inputStream, (String)null, (Properties)null);
    }

    public SqlSessionFactory build(InputStream inputStream, String environment) {
        return this.build((InputStream)inputStream, environment, (Properties)null);
    }

    public SqlSessionFactory build(InputStream inputStream, Properties properties) {
        return this.build((InputStream)inputStream, (String)null, properties);
    }

    public SqlSessionFactory build(InputStream inputStream, String environment, Properties pr
perties) {
        SqlSessionFactory var5;
        try {
            XMLConfigBuilder parser = new XMLConfigBuilder(inputStream, environment, propert
es);
            var5 = this.build(parser.parse());
        } catch (Exception var14) {
            throw ExceptionFactory.wrapException("Error building SqlSession.", var14);
        } finally {
            ErrorContext.instance().reset();

            try {
                inputStream.close();
            } catch (IOException var13) {
            }

        }

        return var5;
    }

    public SqlSessionFactory build(Configuration config) {
        return new DefaultSqlSessionFactory(config);
    }
}


// BeanDefinitionBuilder
public final class BeanDefinitionBuilder {
    public static BeanDefinitionBuilder genericBeanDefinition(String beanClassName) {
        BeanDefinitionBuilder builder = new BeanDefinitionBuilder(new GenericBeanDefinition())

        builder.beanDefinition.setBeanClassName(beanClassName);
        return builder;
    }

    public static BeanDefinitionBuilder genericBeanDefinition(Class<?> beanClass) {
        BeanDefinitionBuilder builder = new BeanDefinitionBuilder(new GenericBeanDefinition())

        builder.beanDefinition.setBeanClass(beanClass);
        return builder;
    }

    public static <T> BeanDefinitionBuilder genericBeanDefinition(Class<T> beanClass, Supplie
<T> instanceSupplier) {
        BeanDefinitionBuilder builder = new BeanDefinitionBuilder(new GenericBeanDefinition())

        builder.beanDefinition.setBeanClass(beanClass);
        builder.beanDefinition.setInstanceSupplier(instanceSupplier);
        return builder;
    }
}

public abstract class ImmutableSet<E> extends ImmutableCollection<E> implements Set<E>


    public static <E> ImmutableSet<E> copyOf(Collection<? extends E> elements) {
    if (elements instanceof ImmutableSet && !(elements instanceof ImmutableSortedSet)) {
          @SuppressWarnings("unchecked") // all supported methods are covariant
           ImmutableSet<E> set = (ImmutableSet<E>) elements;
           if (!set.isPartialView()) {
             return set;
           }
        } else if (elements instanceof EnumSet) {
           return copyOfEnumSet((EnumSet) elements);
        }
        Object[] array = elements.toArray();
        return construct(array.length, array);
    }

    public static <E> ImmutableSet<E> copyOf(Iterable<? extends E> elements) {
        return (elements instanceof Collection)
            ? copyOf((Collection<? extends E>) elements)
            : copyOf(elements.iterator());
    }
    public static class Builder<E> extends ImmutableCollection.ArrayBasedBuilder<E> {
    public Builder<E> add(E element) {
            super.add(element);
            return this;
        }
        public Builder<E> add(E... elements) {
            super.add(elements);
            return this;
        }
    public ImmutableSet<E> build() {
            ImmutableSet<E> result = construct(size, contents);
            // construct has the side effect of deduping contents, so we update size
            // accordingly.
            size = result.size();
            return result;
        }
    }

}

public class Test {

    public static void main(String[] args) {
        Set<String> set = ImmutableSet.<String>builder().add("aaa").add("bbb").add("ccc").buil
();
        System.out.println(set);
    }

}
```

单例模式

基础解析

- 定义：保证一个类仅有一个实例，并提供一个全局访问点
- 类型：创建型
- 适用场景：任何情况下都只有一个实例
- 优点：在内存中只有一个实例，减少内存开销；避免对资源的多重占用；设置全局访问点，严格控访问
- 缺点：没有接口，拓展困难
- 核心：私有构造器；线程安全；延迟加载；序列化和反序列化安全；反射

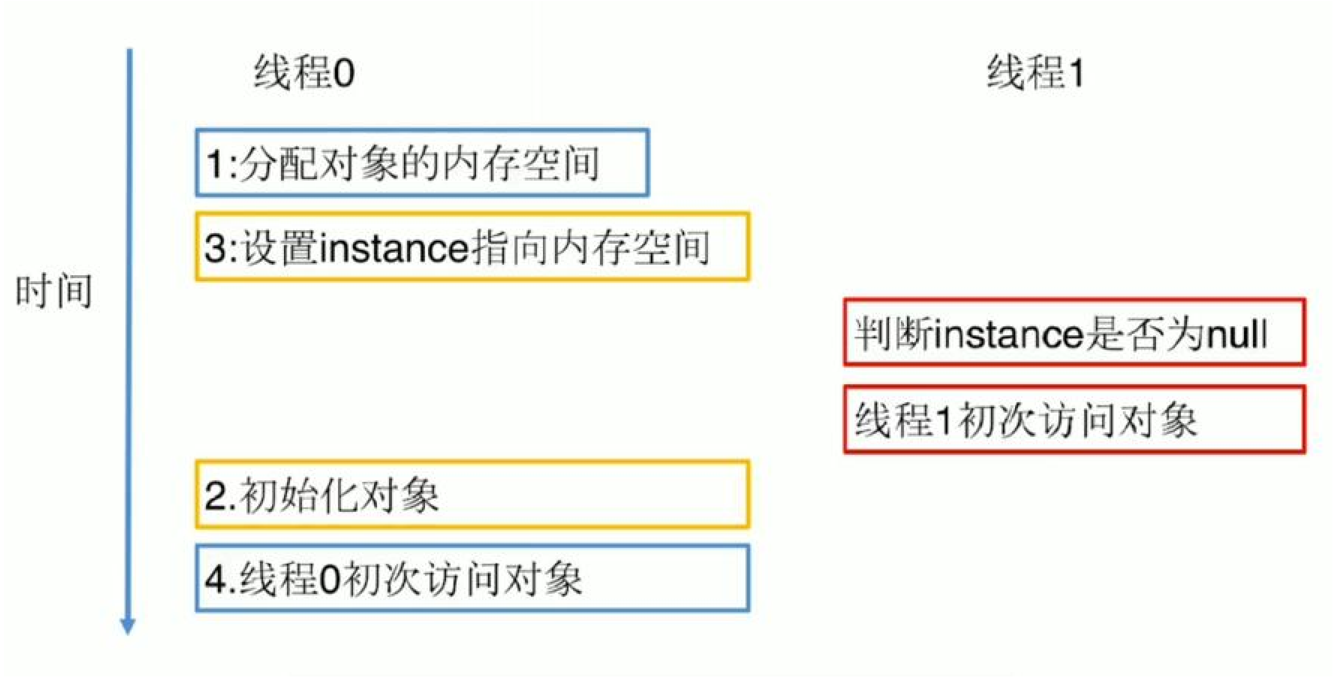
场景解析

单例—懒汉式

```
public class LazySingleton {

    private static LazySingleton lazySingleton = null;
    private LazySingleton() {

    }

    /**
     * 多线程情况下，可能出现创建多个"单例"
     * @return
     */
    /*public static LazySingleton getInstance() {
        if (lazySingleton == null) {
            lazySingleton = new LazySingleton();
        }
        return lazySingleton;
    }*/

    /**
     * 使用 同步锁synchronized，直接修饰static方法——锁类
     * 缺点：消耗资源
     * @return
     */
    public synchronized static LazySingleton getInstance() {
        if (lazySingleton == null) {
            lazySingleton = new LazySingleton();
        }
        return lazySingleton;
    }

}
```

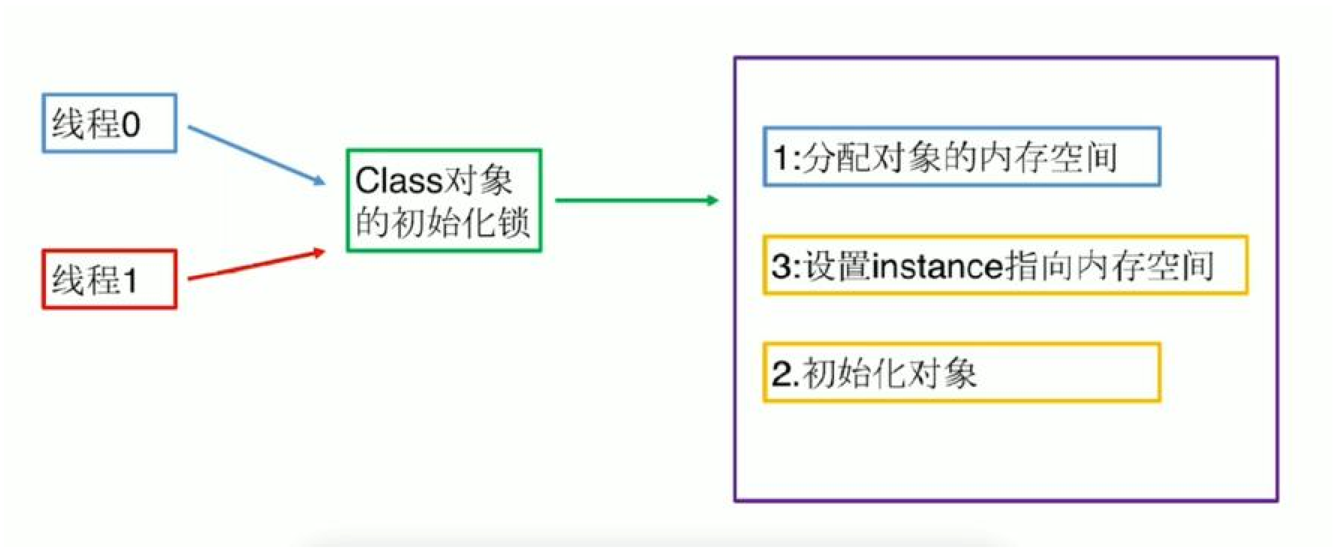
单例—懒汉式（避免指令重排序）

```
public class LazyDoubleCheckSingleton {

    // volatile避免指令重排序
    private volatile static LazyDoubleCheckSingleton lazyDoubleCheckSingleton = null;

    private LazyDoubleCheckSingleton() {

    }

    public static LazyDoubleCheckSingleton getInstance() {
        if (lazyDoubleCheckSingleton == null) {
            synchronized (LazyDoubleCheckSingleton.class) {
                if (lazyDoubleCheckSingleton == null) {
                    lazyDoubleCheckSingleton = new LazyDoubleCheckSingleton();
                    // 重排序：可能导致当前线程 return 前一线程为初始化的单例对象
                    // 1.给对象分配内存
                    // 3.设置 lazyDoubleCheckSingleton 指向该分配的内存
                    // 2.初始化对象
                    // 解决方案：
                    //      1. volatile避免指令重排序
                    //      2. 后一线程对指令重排序不可见
                }
            }
        }
        return lazyDoubleCheckSingleton;
    }

}
```



单例—懒汉式（静态内部类），基于类初始化的延迟加载解决方案

```
public class StaticInnerClassSingleton {

    private StaticInnerClassSingleton() {

    }

    private static class InnerClass {
        private static StaticInnerClassSingleton staticInnerClassSingleton = new StaticInnerClassS
ngleton();
    }

    public static StaticInnerClassSingleton getInstance() {
        return InnerClass.staticInnerClassSingleton;
    }

}
```



单例—饿汉式

```
public class HungrySingleton {

    // 类加载就初始化
    private final static HungrySingleton hungrySingleton;

    static {
        hungrySingleton = new HungrySingleton();
    }

    private HungrySingleton() {

    }

    public static HungrySingleton getInstance() {
        return hungrySingleton;
    }
}
```

懒汉式和饿汉式区别

单例—序列化和反序列化

序列化和反序列化会破坏单例，即序列化和反序列化后的两个对象不一致

```
public class HungrySingleton implements Serializable {

    // 类加载就初始化
    private final static HungrySingleton hungrySingleton;

    static {
        hungrySingleton = new HungrySingleton();
    }

    private HungrySingleton() {
    }

    public static HungrySingleton getInstance() {
        return hungrySingleton;
    }

    public Object readResolve() {
        return hungrySingleton;
    }
}
```

单例构造器禁止反射调用：在单例的构造器加防御代码

```
// 类加载就创建单例对象
public class HungrySingleton implements Serializable {

    // 类加载就初始化
    private final static HungrySingleton hungrySingleton;

    static {
        hungrySingleton = new HungrySingleton();
    }

    private HungrySingleton() {
        if (hungrySingleton != null) {
            throw new RuntimeException("单例构造器禁止反射调用");
        }
    }

    public static HungrySingleton getInstance() {
        return hungrySingleton;
    }

    public Object readResolve() {
        return hungrySingleton;
    }
}

public class StaticInnerClassSingleton {

    private StaticInnerClassSingleton() {
        if (InnerClass.staticInnerClassSingleton != null) {
            throw new RuntimeException("单例构造器禁止反射调用");
        }
    }

    private static class InnerClass {
        private static StaticInnerClassSingleton staticInnerClassSingleton = new StaticInnerClassS
ngleton();
    }

    public static StaticInnerClassSingleton getInstance() {
        return InnerClass.staticInnerClassSingleton;
    }
}
```

单例—反射攻击

反射也会破坏单例

```
public class LazySingleton {

    private static LazySingleton lazySingleton = null;

    private static boolean flag = true;

    private LazySingleton() {
        if (flag) {
            flag = false;
        } else {
            throw new RuntimeException("单例构造器禁止反射调用");
        }
    }

    /**
     * 使用 同步锁synchronized，直接修饰static方法——锁类
     * 缺点：消耗资源
     * @return
     */
    public synchronized static LazySingleton getInstance() {
        if (lazySingleton == null) {
            lazySingleton = new LazySingleton();
        }
        return lazySingleton;
    }

}

public class Test {

    public static void main(String[] args) throws IOException, ClassNotFoundException {
        Class objectClass = LazySingleton.class;
        Constructor constructor = objectClass.getDeclaredConstructor();
        constructor.setAccessible(true);

        LazySingleton instance = LazySingleton.getInstance();
        // 通过反射，还是可以修改flag的值，破坏单例
        /*Field flag = objectClass.getDeclaredField("flag");
        flag.setAccessible(true);
        flag.set(instance, true);*/
        LazySingleton newInstance = (LazySingleton) constructor.newInstance();

        System.out.println(instance);
        System.out.println(newInstance);
        System.out.println(instance == newInstance);
    }
}
```

使用枚举避免反射攻击

```
public enum  EnumInstance {

    INSTANCE {
        protected void printTestCase() {
            System.out.println("Hefery Print Test");
        }
    };

    protected abstract void printTestCase();

    private Object data;

    public Object getData() {
        return data;
    }

    public void setData(Object data) {
        this.data = data;
    }

    public static EnumInstance getInstance() {
        return INSTANCE;
    }
}


public class Test {

    public static void main(String[] args) throws IOException, ClassNotFoundException, NoSuc
MethodException, IllegalAccessException, InvocationTargetException, InstantiationException,
NoSuchFieldException {
    // 使用Enum
        /*EnumInstance instance = EnumInstance.getInstance();
        instance.setData(new Object());

        ObjectOutputStream objectOutputStream = new ObjectOutputStream(new FileOutputSt
eam("singleton_file"));
        objectOutputStream.writeObject(instance);

        File file = new File("singleton_file");
        ObjectInputStream objectInputStream = new java.io.ObjectInputStream(new FileInputStr
am(file));

        EnumInstance newInstance = (EnumInstance) objectInputStream.readObject();

        System.out.println(instance.getData());
        System.out.println(newInstance.getData());
        System.out.println(instance.getData() == newInstance.getData());*/

        Class objectClass = EnumInstance.class;
        Constructor constructor = objectClass.getDeclaredConstructor(String.class, int.class);
        constructor.setAccessible(true);

        EnumInstance instance = (EnumInstance) constructor.newInstance("Hefery", 666);
        instance.printTestCase();
        // 通过反射，还是可以修改flag的值，破坏单例
        Field flag = objectClass.getDeclaredField("flag");
        flag.setAccessible(true);
        flag.set(instance, true);
        EnumInstance newInstance = (EnumInstance) constructor.newInstance();

        System.out.println(instance);
        System.out.println(newInstance);
        System.out.println(instance == newInstance);
    }

}
```

容器单例

```
public class ContainerSingleton {

    private static Map<String, Object> singletonMap = new HashMap<String, Object>();

    private ContainerSingleton() {

    }

    public static void putInstance(String key, Object instance) {
        if (StringUtils.isNotBlank(key) && instance != null) {
            if (!singletonMap.containsKey(key)) {
                singletonMap.put(key, instance);
            }
        }
    }

    public static Object getInstance(String key) {
        return singletonMap.get(key);
    }

}
```

使用HashMap容器还是有线程不安全的隐患

ThreadLocal线程单例

不能保证全局唯一，但是可以保证线程唯一

```
public class ThreadLocalInstance {

    private static final ThreadLocal<ThreadLocalInstance> threadLocalInstance = new ThreadL
cal<ThreadLocalInstance>() {
        @Override
        protected ThreadLocalInstance initialValue() {
            return new ThreadLocalInstance();
        }
    };

    private ThreadLocalInstance() {

    }

    public static ThreadLocalInstance getInstance() {
        return threadLocalInstance.get();
    }

}
```

破坏单例的情况

源码解析

```
// 饿汉式
public class Runtime {
    private static Runtime currentRuntime = new Runtime();

    public static Runtime getRuntime() {
        return currentRuntime;
    }

}

// 容器单例
public class Desktop {
    public static synchronized Desktop getDesktop(){
        if (GraphicsEnvironment.isHeadless()) throw new HeadlessException();
        if (!Desktop.isDesktopSupported()) {
            throw new UnsupportedOperationException("Desktop API is not " +
                                                    "supported on the current platform");
        }

        sun.awt.AppContext context = sun.awt.AppContext.getAppContext();
        Desktop desktop = (Desktop)context.get(Desktop.class);

        if (desktop == null) {
            desktop = new Desktop();
            context.put(Desktop.class, desktop);
        }

        return desktop;
    }
}

//
public abstract class AbstractFactoryBean<T> implements FactoryBean<T>, BeanClassLoader
ware, BeanFactoryAware, InitializingBean, DisposableBean {
    public final T getObject() throws Exception {
        if (this.isSingleton()) {
            return this.initialized ? this.singletonInstance : this.getEarlySingletonInstance();
        } else {
            return this.createInstance();
        }
    }

    private T getEarlySingletonInstance() throws Exception {
        Class<?>[] ifcs = this.getEarlySingletonInterfaces();
        if (ifcs == null) {
            throw new FactoryBeanNotInitializedException(this.getClass().getName() + " does not
upport circular references");
        } else {
            if (this.earlySingletonInstance == null) {
                this.earlySingletonInstance = Proxy.newProxyInstance(this.beanClassLoader, ifcs, ne
 AbstractFactoryBean.EarlySingletonInvocationHandler());
            }

            return this.earlySingletonInstance;
        }
    }


}

// 线程单例
public class ErrorContext {
    private static final ThreadLocal<ErrorContext> LOCAL = new ThreadLocal();

    private ErrorContext() {
    }

    public static ErrorContext instance() {
        ErrorContext context = (ErrorContext)LOCAL.get();
        if (context == null) {
            context = new ErrorContext();
            LOCAL.set(context);
        }
        return context;
    }
}
```

原型模式

基础解析

- 定义：原型实例指定创建对象的种类，并且通过拷贝这些原型创建新的对象
- 核心：不需要知道任何创建细节，不调用构造函数
- 类型：创建型
- 适用场景：

类初始化消耗较多资源

new一个对象需要非常繁琐的过程（数据准备、访问权限）

构造函数比较复杂

循环体生产大量对象

● 优点：

原型模式性能比直接new一个对象性能高

简化创建过程

● 缺点：

必须配置克隆方法

对克隆复杂对象或对克隆出的对象进行复杂改造时，容易引入风险

深拷贝、浅拷贝要运用得到

● 拓展：

深克隆

浅克隆

场景解析

```
public class Mail implements Cloneable {

    private String name;
    private String emailAdress;
    private String content;

    public Mail() {
        System.out.println("Mail Class Constructor");
    }

    @Override
    protected Object clone() throws CloneNotSupportedException {
        System.out.println("Clone Mail Object");
        return super.clone();
    }

}

public class MailUtils {

    public static void sendMail(Mail mail) {
        String outputContent = "向{0}同学，邮件地址：{1}，邮件内容：{2}，发送邮件成功";
        System.out.println(MessageFormat.format(outputContent, mail.getName(), mail.getEmai
Adress(), mail.getContent()));
    }

    public static void saveOriginMailRecord(Mail mail) {
        System.out.println("存储originMail记录， originMail： " + mail.getContent());
    }
}

public class Test {

    public static void main(String[] args) throws CloneNotSupportedException {
        Mail mail = new Mail();
        mail.setContent("初始化模板");

        for (int i = 0; i < 10; i++) {
            Mail mailTemp = (Mail) mail.clone();
            mailTemp.setName("姓名： " + i);
            mailTemp.setEmailAdress("Emil地址： " + i + "@126.com");
            mailTemp.setContent("恭喜中奖");
            MailUtils.sendMail(mailTemp);
        }

        MailUtils.saveOriginMailRecord(mail);
    }

}
```

克隆会破坏单例，为了避免这种情况，类不实现 Cloneable 接口，或者 Override clone()方法时直接回getInstance()

源码解析

```
public class Object {
    protected native Object clone() throws CloneNotSupportedException;
}

public interface Cloneable {
}

public class ArrayList<E> extends AbstractList<E>
        implements List<E>, RandomAccess, Cloneable, java.io.Serializable
{
    public Object clone() {
        try {
            ArrayList<?> v = (ArrayList<?>) super.clone();
            v.elementData = Arrays.copyOf(elementData, size);
            v.modCount = 0;
            return v;
        } catch (CloneNotSupportedException e) {
            // this shouldn't happen, since we are Cloneable
            throw new InternalError(e);
        }
    }
}

public class HashMap<K,V> extends AbstractMap<K,V>
    implements Map<K,V>, Cloneable, Serializable {
    @SuppressWarnings("unchecked")
    @Override
    public Object clone() {
        HashMap<K,V> result;
        try {
            result = (HashMap<K,V>)super.clone();
        } catch (CloneNotSupportedException e) {
            // this shouldn't happen, since we are Cloneable
            throw new InternalError(e);
        }
        result.reinitialize();
        result.putMapEntries(this, false);
        return result;
    }
}
```

结构型模式

适配器模式

基础解析

场景解析

源码解析

装饰者模式

基础解析

● 定义：在不改变原有对象的基础上，将功能附加到对象上，提供了比继承更有弹性的替代方案（拓原有对象的功能）

● 类型：结构型

● 适用场景：

拓展一个类的功能，或给一个类添加附加职责

动态给对象添加功能，这些功能可以动态撤销

● 优点：

继承的有力补充，比继承灵活，不改变原有对象情况下，给一个对象拓展功能

通过使用不同装饰类以及这些装饰的排列组合，可以实现不同的效果

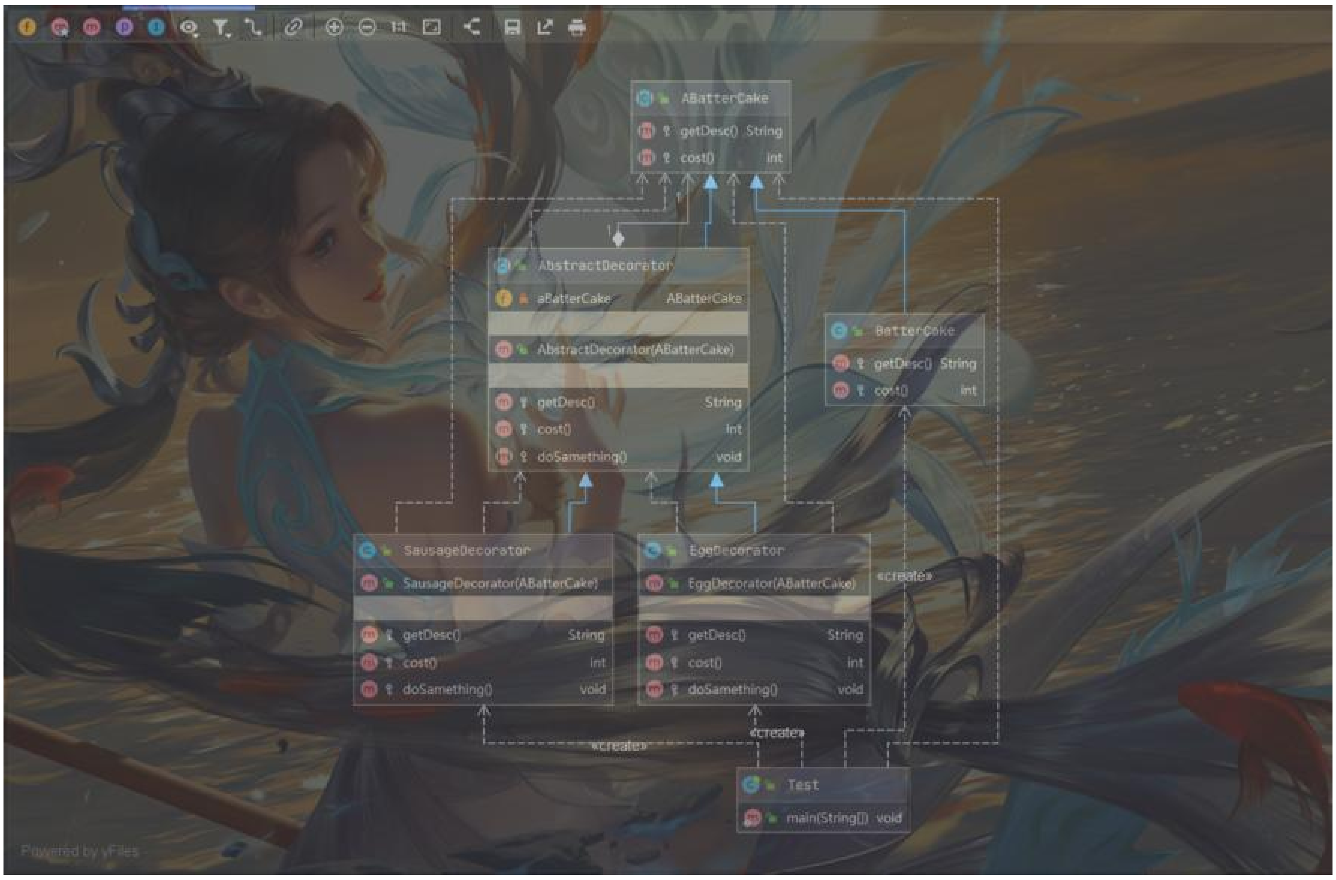
符合开闭原则

● 缺点：

会出现更多代码、类，增加程序复杂性

动态装饰、多层装饰时会更加复杂

场景解析

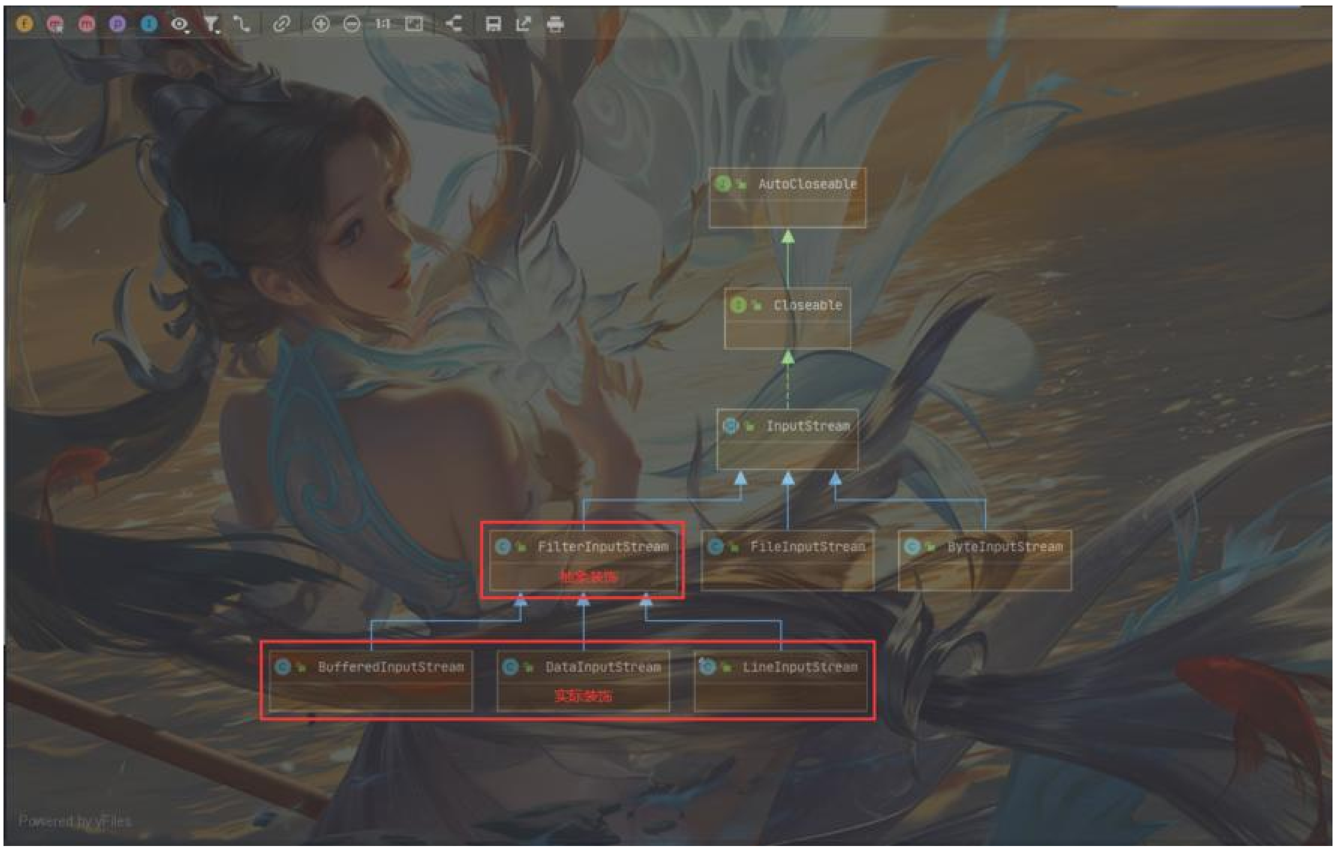


要被装饰的实体对象：BatterCake

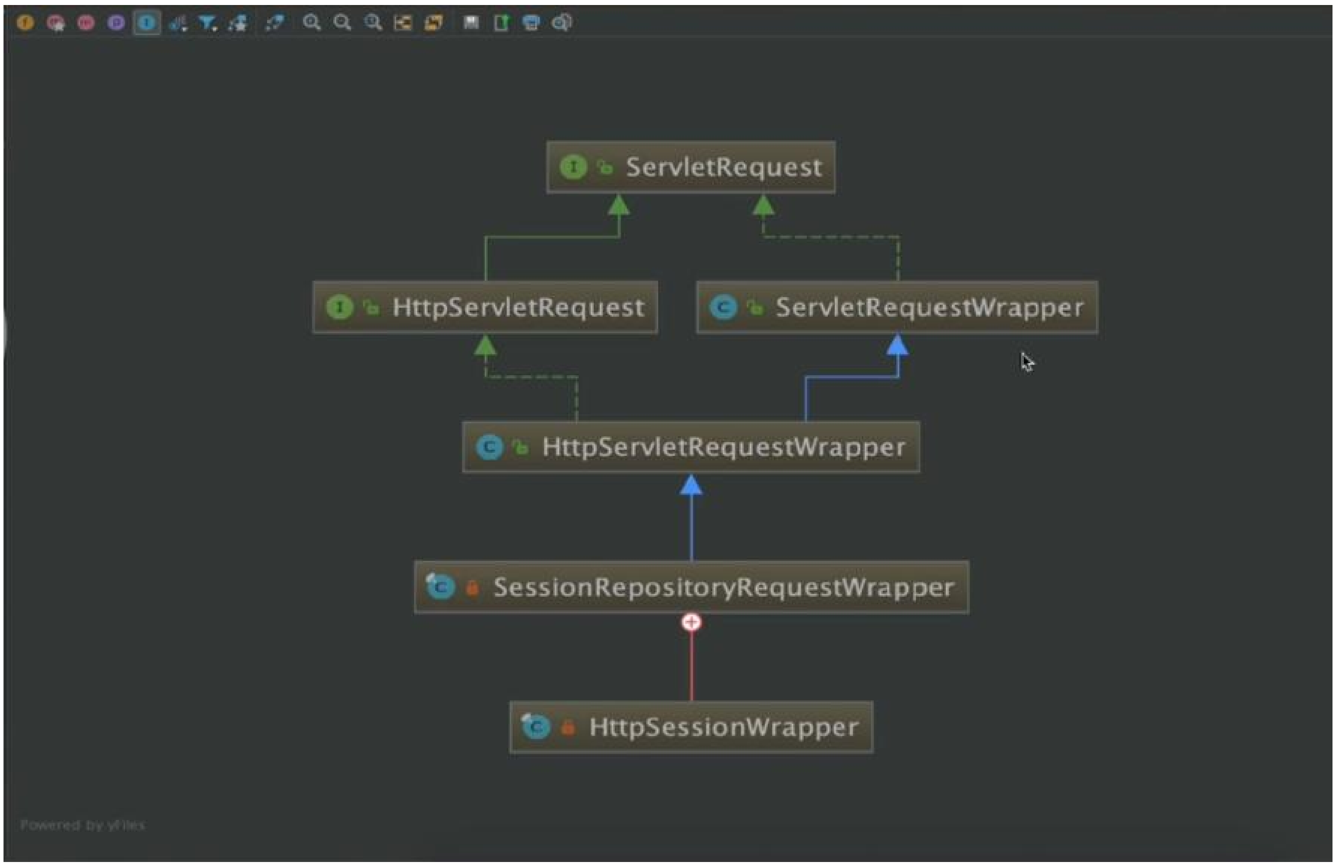
抽象的装饰角色：AbstractDecorator
实际装饰的角色：EggDecorator、SausageDecorator

```
public class Test {
    public static void main(String[] args) {
        ABatterCake aBatterCake;
        // 原始煎饼
        aBatterCake = new BatterCake();
        // 加蛋
        aBatterCake = new EggDecorator(aBatterCake);
        aBatterCake = new EggDecorator(aBatterCake);
        // 加香肠
        aBatterCake = new SausageDecorator(aBatterCake);

        System.out.println(aBatterCake.getDesc() + ",售价：" + aBatterCake.cost());
    }
}
```

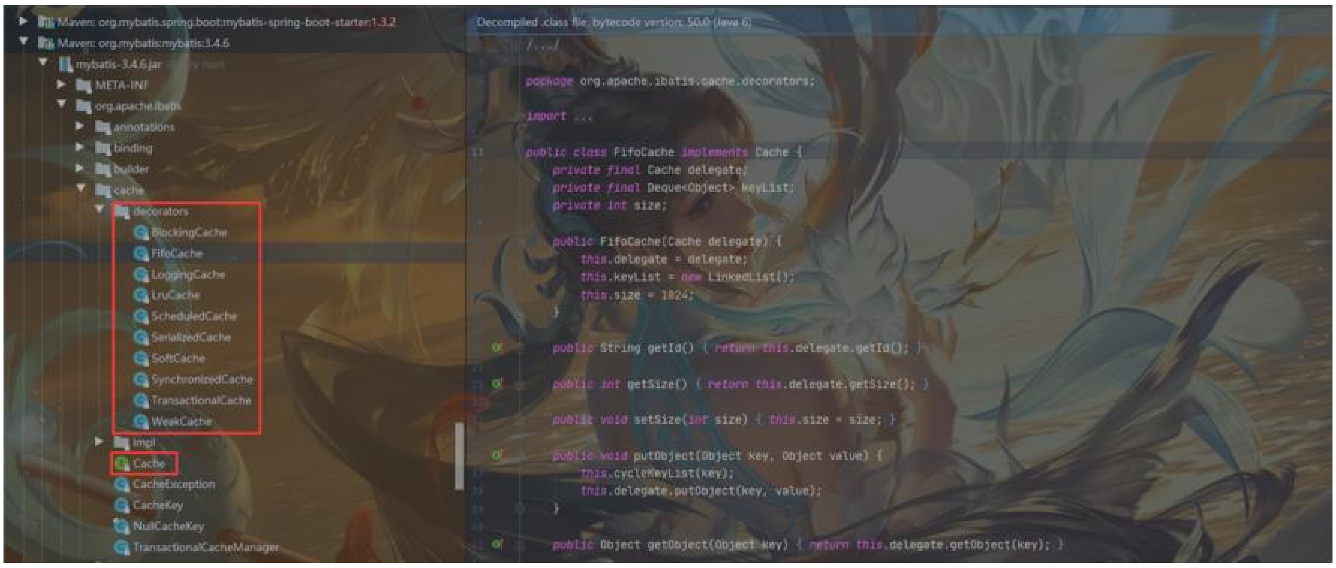
源码解析



```
public abstract class Reader implements Readable, Closeable {......}

public class BufferedReader extends Reader {
    private Reader in;
    public BufferedReader(Reader in, int sz) {
        super(in);
        if (sz <= 0)
            throw new IllegalArgumentException("Buffer size <= 0");
        this.in = in;
        cb = new char[sz];
        nextChar = nChars = 0;
    }
}
```



```
public class TransactionAwareCacheDecorator implements Cache {
    private final Cache targetCache;

    public TransactionAwareCacheDecorator(Cache targetCache) {
        Assert.notNull(targetCache, "Target Cache must not be null");
        this.targetCache = targetCache;
    }
}
```



```
▶ Maven: org.mybatis.spring.boot:mybatis-spring-boot-starter:1.3.2
▼ Maven: org.mybatis:mybatis:3.4.6
  ▼ mybatis-3.4.6.jar
    ▶ META-INF
    ▼ org.apache.ibatis
      ▶ annotations
      ▶ binding
      ▶ builder
      ▼ cache
        ▼ decorators
            BlockingCache
            FifoCache
            LoggingCache
            LruCache
            ScheduledCache
            SerializedCache
            SoftCache
            SynchronizedCache
            TransactionalCache
            WeakCache
        ▶ impl
          Cache
          CacheException
          CacheKey
          NullCacheKey
          TransactionalCacheManager
```

```
Decompiled .class file, bytecode version: 50.0 (Java 6)
/.../

package org.apache.ibatis.cache.decorators;

import ...

public class FifoCache implements Cache {
    private final Cache delegate;
    private final Deque<Object> keyList;
    private int size;

    public FifoCache(Cache delegate) {
        this.delegate = delegate;
        this.keyList = new LinkedList();
        this.size = 1024;
    }

    public String getId() { return this.delegate.getId(); }

    public int getSize() { return this.delegate.getSize(); }

    public void setSize(int size) { this.size = size; }

    public void putObject(Object key, Object value) {
        this.cycleKeyList(key);
        this.delegate.putObject(key, value);
    }

    public Object getObject(Object key) { return this.delegate.getObject(key); }
```

代理模式

基础解析

- 定义：为其他对象提供一种代理，以控制对这个对象的访问
- 核心：代理对象在客户端和目标对象之间起到中介的作用
- 类型：结构型
- 适用场景：

保护目标对象

增强目标对象

- 优点：

代理模式能将代理对象与真实被调用的目标对象分离

一定程度上降低了系统的耦合度，扩展性好

保护目标对象

增强目标对象

- 缺点：

代理模式会造成系统设计中类的数目增加

在客户端和目标对象增加一个代理对象，会造成请求处理速度变慢

增加系统的复杂度

- 扩展：

静态代理

动态代理

CGLib代理

- Spring代理选择-扩展

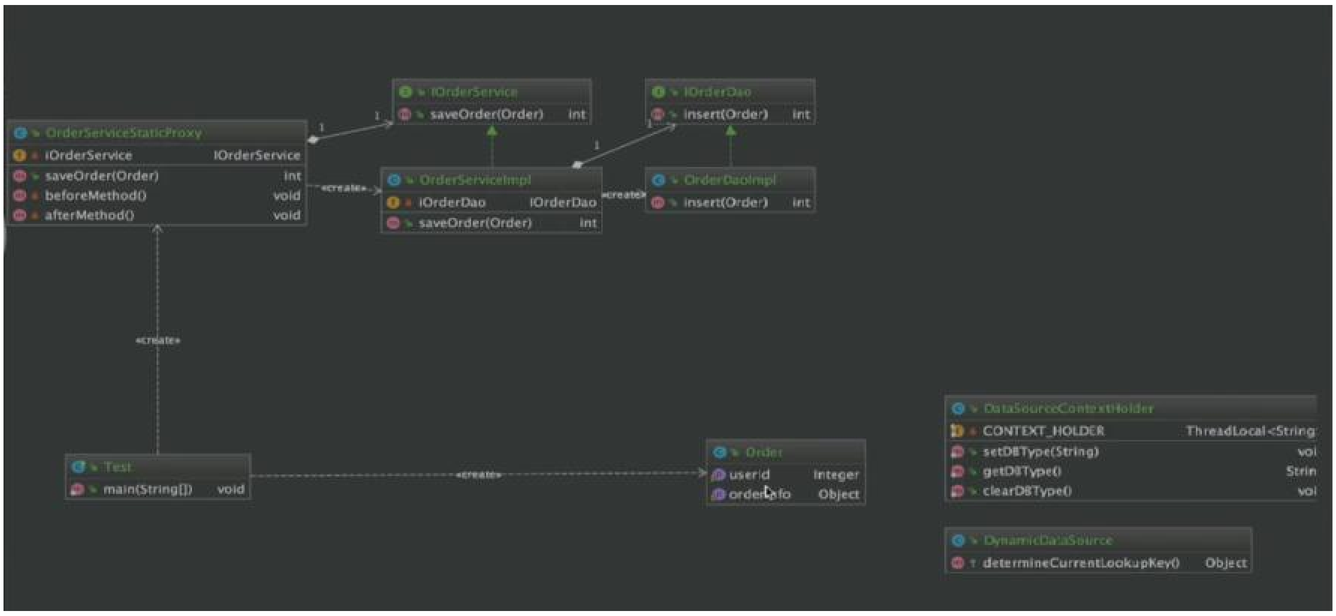
当Bean有实现接口时，Spring就会用JDK的动态代理

当Bean没有实现接口时，Spring使用CGlib

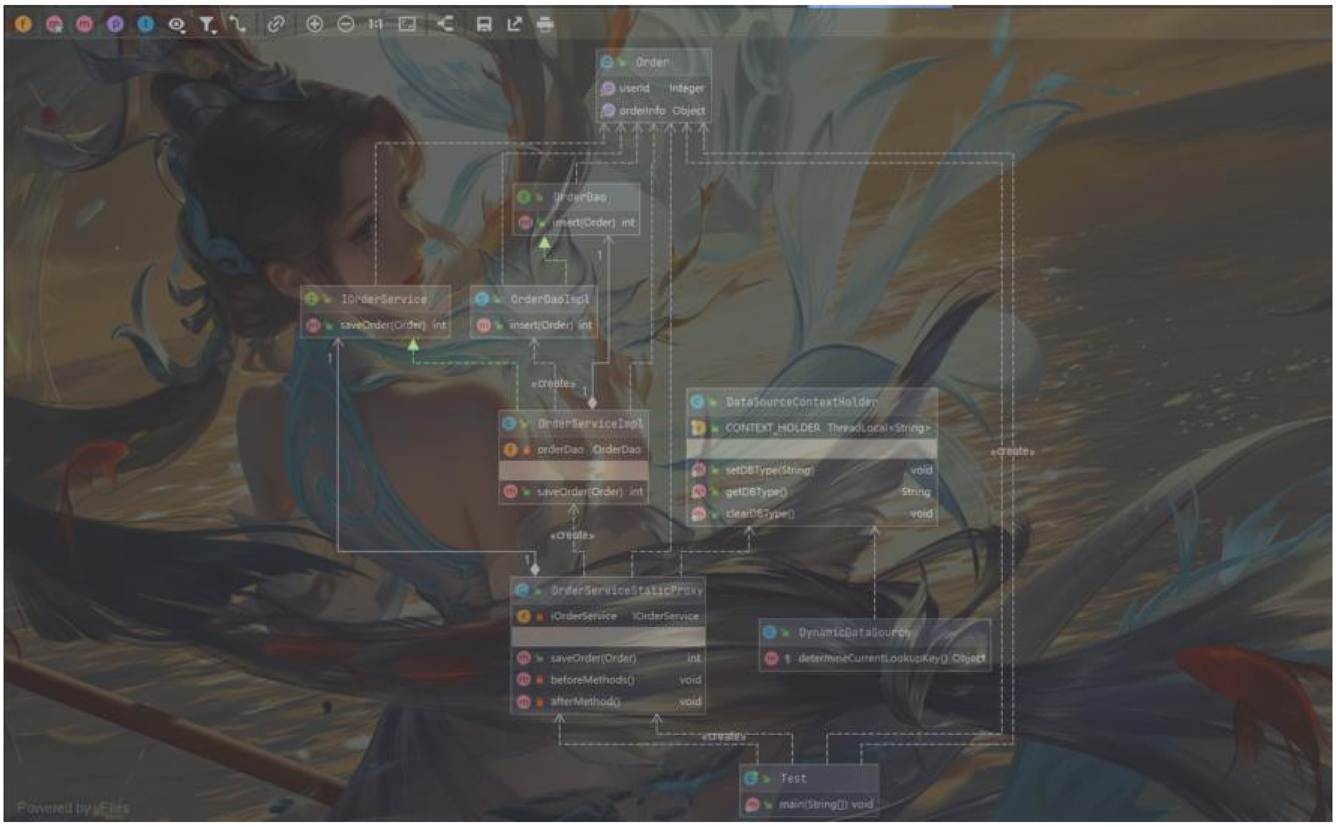
可以强制使用Cglib：在spring配置中加入<aop:aspectj-autoproxy proxy-target-class="true"/>）

场景解析

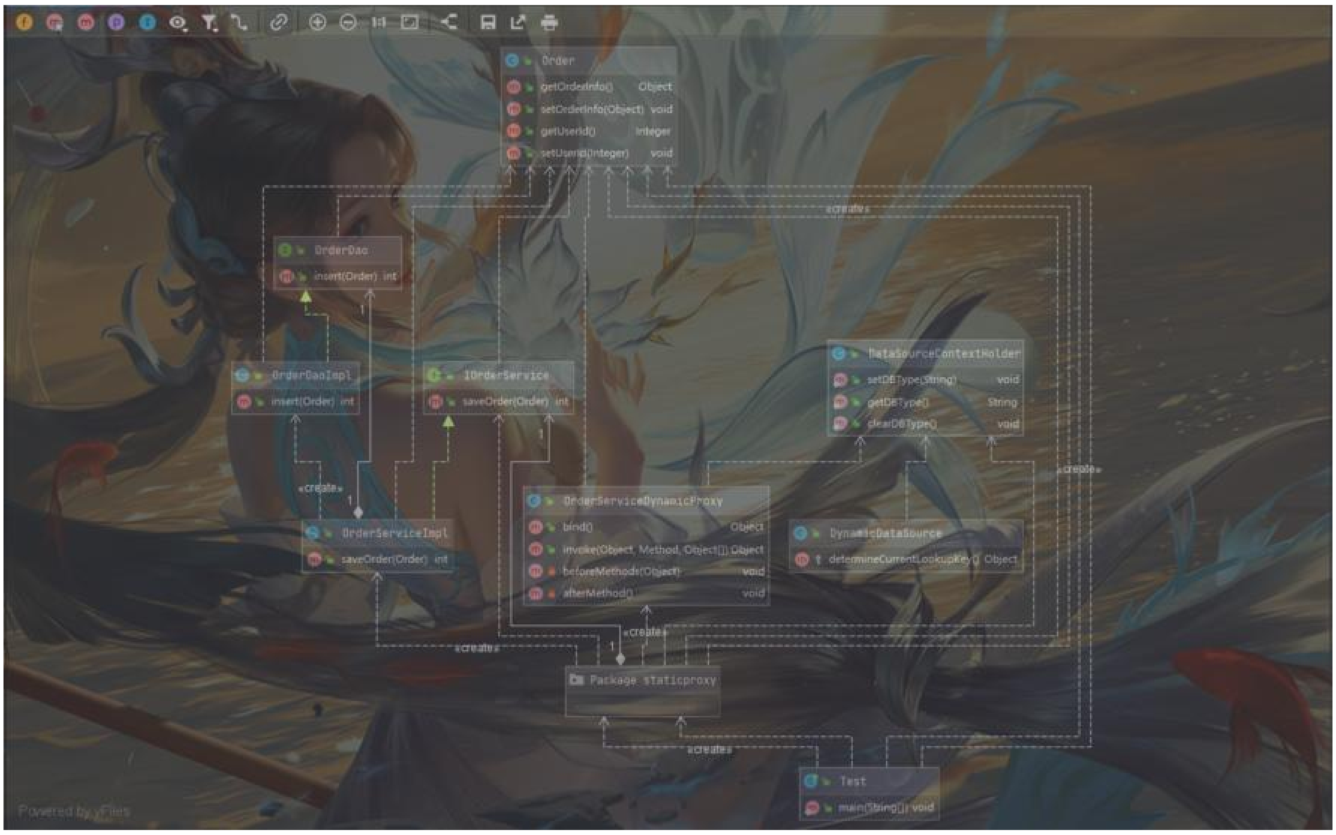
静态代理



原文链接：Java 的设计模式



动态代理



源码解析

```
public class Proxy implements java.io.Serializable {
```

原文链接：Java 的设计模式

```
@CallerSensitive
public static InvocationHandler getInvocationHandler(Object proxy)
    throws IllegalArgumentException
{
    /*
     * Verify that the object is actually a proxy instance.
     */
    if (!isProxyClass(proxy.getClass())) {
        throw new IllegalArgumentException("not a proxy instance");
    }

    final Proxy p = (Proxy) proxy;
    final InvocationHandler ih = p.h;
    if (System.getSecurityManager() != null) {
        Class<?> ihClass = ih.getClass();
        Class<?> caller = Reflection.getCallerClass();
        if (ReflectUtil.needsPackageAccessCheck(caller.getClassLoader(),
                                                ihClass.getClassLoader()))
        {
            ReflectUtil.checkPackageAccess(ihClass);
        }
    }

    return ih;
}

@CallerSensitive
public static Object newProxyInstance(ClassLoader loader,
                                      Class<?>[] interfaces,
                                      InvocationHandler h)
    throws IllegalArgumentException
{
    Objects.requireNonNull(h);

    final Class<?>[] intfs = interfaces.clone();
    final SecurityManager sm = System.getSecurityManager();
    if (sm != null) {
        checkProxyAccess(Reflection.getCallerClass(), loader, intfs);
    }

    /*
     * Look up or generate the designated proxy class.
     */
    Class<?> cl = getProxyClass0(loader, intfs);

    /*
     * Invoke its constructor with the designated invocation handler.
     */
    try {
        if (sm != null) {
            checkNewProxyPermission(Reflection.getCallerClass(), cl);
        }

        final Constructor<?> cons = cl.getConstructor(constructorParams);
```

```
            final InvocationHandler ih = h;
            if (!Modifier.isPublic(cl.getModifiers())) {
                AccessController.doPrivileged(new PrivilegedAction<Void>() {
                    public Void run() {
                        cons.setAccessible(true);
                        return null;
                    }
                });
            }
            return cons.newInstance(new Object[]{h});
        } catch (IllegalAccessException|InstantiationException e) {
            throw new InternalError(e.toString(), e);
        } catch (InvocationTargetException e) {
            Throwable t = e.getCause();
            if (t instanceof RuntimeException) {
                throw (RuntimeException) t;
            } else {
                throw new InternalError(t.toString(), t);
            }
        } catch (NoSuchMethodException e) {
            throw new InternalError(e.toString(), e);
        }
    }
}

JdkDynamicAopProxy
CglibAopProxy
MapperProxyFactory
```

外观模式

基础解析

- 定义：提供一个统一的接口，用来访问子系统中的一系列接口
- 核心：外观模式定义一个高层接口，让子系统更容易使用
- 类型：结构性
- 适用场景：

子系统越来越复杂，增加外观模式提供简单的调用接口

构建多层系统结构，利用外观对象作为每层的入口，简化层间调用

- 优点：

简化调用过程，无需深入了解子系统，防止带来风险

减少系统依赖，松散耦合

更好地划分访问层次

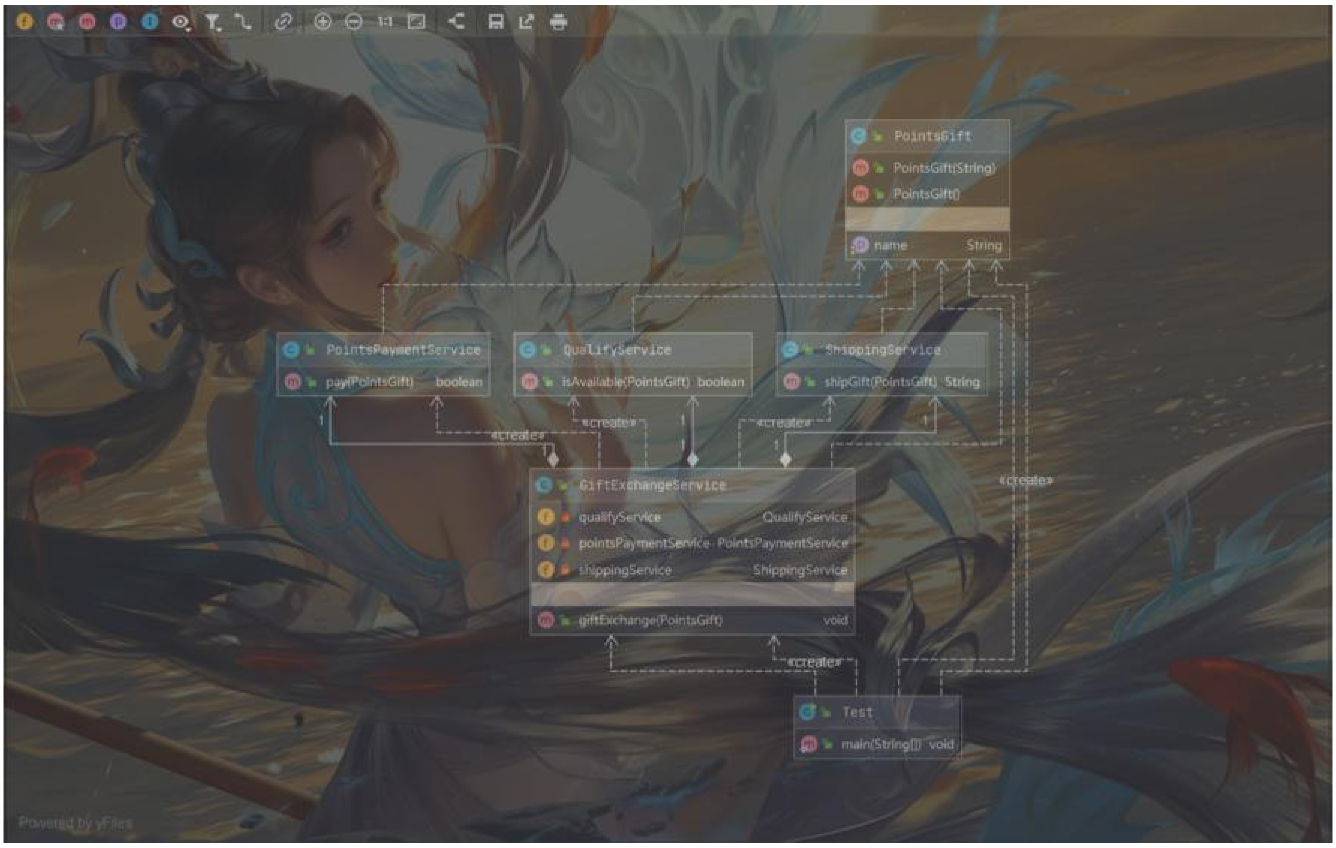
符合迪米特法则

- 缺点：

增加子系统、拓展子系统容易引入风险

不符合开闭原则

场景解析



Test应用层只需调用 GiftExchangeService.giftExchange 方法即可实现积分兑换礼物的功能需求
实际上，外观接口包含了
  资格审查 QualifyService.isAvailable
  积分支付 PointsPaymentService.pay
  物流派送 ShippingService.shipGift
应用层不直接与这些子系统交互，而是通过 GiftExchangeService.giftExchange 接口调用实现

源码解析

```
// JdbcUtils封装
public abstract class JdbcUtils {
    public static final int TYPE_UNKNOWN = -2147483648;
    private static final Log logger = LogFactory.getLog(JdbcUtils.class);

    public JdbcUtils() {
    }

    public static void closeConnection(@Nullable Connection con) {
        if (con != null) {
            try {
                con.close();
            } catch (SQLException var2) {
                logger.debug("Could not close JDBC Connection", var2);
            } catch (Throwable var3) {
                logger.debug("Unexpected exception on closing JDBC Connection", var3);
            }
        }
```

原文链接：Java 的设计模式

```
    }

    public static void closeStatement(@Nullable Statement stmt) {
        if (stmt != null) {
            try {
                stmt.close();
            } catch (SQLException var2) {
                logger.trace("Could not close JDBC Statement", var2);
            } catch (Throwable var3) {
                logger.trace("Unexpected exception on closing JDBC Statement", var3);
            }
        }

    }

    public static void closeResultSet(@Nullable ResultSet rs) {
        if (rs != null) {
            try {
                rs.close();
            } catch (SQLException var2) {
                logger.trace("Could not close JDBC ResultSet", var2);
            } catch (Throwable var3) {
                logger.trace("Unexpected exception on closing JDBC ResultSet", var3);
            }
        }

    }

    @Nullable
    public static Object getResultSetValue(ResultSet rs, int index, @Nullable Class<?> required
ype) throws SQLException {
        if (requiredType == null) {
            return getResultSetValue(rs, index);
        } else if (String.class == requiredType) {
            return rs.getString(index);
        } else {
            Object value;
            if (Boolean.TYPE != requiredType && Boolean.class != requiredType) {
                if (Byte.TYPE != requiredType && Byte.class != requiredType) {
                    if (Short.TYPE != requiredType && Short.class != requiredType) {
                        if (Integer.TYPE != requiredType && Integer.class != requiredType) {
                            if (Long.TYPE != requiredType && Long.class != requiredType) {
                                if (Float.TYPE != requiredType && Float.class != requiredType) {
                                    if (Double.TYPE != requiredType && Double.class != requiredType &&
Number.class != requiredType) {
                                        if (BigDecimal.class == requiredType) {
                                            return rs.getBigDecimal(index);
                                        }

                                        if (Date.class == requiredType) {
                                            return rs.getDate(index);
                                        }
                                        if (Time.class == requiredType) {
                                            return rs.getTime(index);
                                        }

                                        if (Timestamp.class != requiredType && java.util.Date.class != requir
dType) {
                                            if (byte[].class == requiredType) {
                                                return rs.getBytes(index);
                                            }

                                            if (Blob.class == requiredType) {
                                                return rs.getBlob(index);
                                            }

                                            if (Clob.class == requiredType) {
                                                return rs.getClob(index);
                                            }

                                            if (requiredType.isEnum()) {
                                                Object obj = rs.getObject(index);
                                                if (obj instanceof String) {
                                                    return obj;
                                                }

                                                if (obj instanceof Number) {
                                                    return NumberUtils.convertNumberToTargetClass((Number)o
j, Integer.class);
                                                }

                                                return rs.getString(index);
                                            }

                                            try {
                                                return rs.getObject(index, requiredType);
                                            } catch (AbstractMethodError var5) {
                                                logger.debug("JDBC driver does not implement JDBC 4.1 'getOb
ect(int, Class)' method", var5);
                                            } catch (SQLFeatureNotSupportedException var6) {
                                                logger.debug("JDBC driver does not support JDBC 4.1 'getObjec
(int, Class)' method", var6);
                                            } catch (SQLException var7) {
                                                logger.debug("JDBC driver has limited support for JDBC 4.1 'ge
Object(int, Class)' method", var7);
                                            }

                                            String typeName = requiredType.getSimpleName();
                                            if ("LocalDate".equals(typeName)) {
                                                return rs.getDate(index);
                                            }

                                            if ("LocalTime".equals(typeName)) {
                                                return rs.getTime(index);
                                            }
                                        if ("LocalDateTime".equals(typeName)) {
                                            return rs.getTimestamp(index);
                                        }

                                        return getResultSetValue(rs, index);
                                    }

                                    return rs.getTimestamp(index);
                                }

                                value = rs.getDouble(index);
                            } else {
                                value = rs.getFloat(index);
                            }
                        } else {
                            value = rs.getLong(index);
                        }
                    } else {
                        value = rs.getInt(index);
                    }
                } else {
                    value = rs.getShort(index);
                }
            } else {
                value = rs.getByte(index);
            }
        } else {
            value = rs.getBoolean(index);
        }

        return rs.wasNull() ? null : value;
    }
}

@Nullable
public static Object getResultSetValue(ResultSet rs, int index) throws SQLException {
    Object obj = rs.getObject(index);
    String className = null;
    if (obj != null) {
        className = obj.getClass().getName();
    }

    if (obj instanceof Blob) {
        Blob blob = (Blob)obj;
        obj = blob.getBytes(1L, (int)blob.length());
    } else if (obj instanceof Clob) {
        Clob clob = (Clob)obj;
        obj = clob.getSubString(1L, (int)clob.length());
    } else if (!"oracle.sql.TIMESTAMP".equals(className) && !"oracle.sql.TIMESTAMPTZ".equ
ls(className)) {
        if (className != null && className.startsWith("oracle.sql.DATE")) {
            String metaDataClassName = rs.getMetaData().getColumnClassName(index);
            if (!"java.sql.Timestamp".equals(metaDataClassName) && !"oracle.sql.TIMESTAMP".
quals(metaDataClassName)) {
```

```
                obj = rs.getDate(index);
            } else {
                obj = rs.getTimestamp(index);
            }
        } else if (obj instanceof Date && "java.sql.Timestamp".equals(rs.getMetaData().getCol
mnClassName(index))) {
            obj = rs.getTimestamp(index);
        }
    } else {
        obj = rs.getTimestamp(index);
    }

    return obj;
}

public static Object extractDatabaseMetaData(DataSource dataSource, DatabaseMetaData
allback action) throws MetaDataAccessException {
    Connection con = null;

    Object var4;
    try {
        con = DataSourceUtils.getConnection(dataSource);
        DatabaseMetaData metaData = con.getMetaData();
        if (metaData == null) {
            throw new MetaDataAccessException("DatabaseMetaData returned by Connection [
 + con + "] was null");
        }

        var4 = action.processMetaData(metaData);
    } catch (CannotGetJdbcConnectionException var10) {
        throw new MetaDataAccessException("Could not get Connection for extracting meta-
ata", var10);
    } catch (SQLException var11) {
        throw new MetaDataAccessException("Error while extracting DatabaseMetaData", var
1);
    } catch (AbstractMethodError var12) {
        throw new MetaDataAccessException("JDBC DatabaseMetaData method not impleme
ted by JDBC driver - upgrade your driver", var12);
    } finally {
        DataSourceUtils.releaseConnection(con, dataSource);
    }

    return var4;
}

public static <T> T extractDatabaseMetaData(DataSource dataSource, String metaDataMet
odName) throws MetaDataAccessException {
    return extractDatabaseMetaData(dataSource, (dbmd) -> {
        try {
            return DatabaseMetaData.class.getMethod(metaDataMethodName).invoke(dbmd);
        } catch (NoSuchMethodException var3) {
            throw new MetaDataAccessException("No method named '" + metaDataMethodN
me + "' found on DatabaseMetaData instance [" + dbmd + "]", var3);
        } catch (IllegalAccessException var4) {
            throw new MetaDataAccessException("Could not access DatabaseMetaData method
'" + metaDataMethodName + "'", var4);
        } catch (InvocationTargetException var5) {
            if (var5.getTargetException() instanceof SQLException) {
                throw (SQLException)var5.getTargetException();
            } else {
                throw new MetaDataAccessException("Invocation of DatabaseMetaData method '
 + metaDataMethodName + "' failed", var5);
            }
        }
    });
}

public static boolean supportsBatchUpdates(Connection con) {
    try {
        DatabaseMetaData dbmd = con.getMetaData();
        if (dbmd != null) {
            if (dbmd.supportsBatchUpdates()) {
                logger.debug("JDBC driver supports batch updates");
                return true;
            }

            logger.debug("JDBC driver does not support batch updates");
        }
    } catch (SQLException var2) {
        logger.debug("JDBC driver 'supportsBatchUpdates' method threw exception", var2);
    }

    return false;
}

@Nullable
public static String commonDatabaseName(@Nullable String source) {
    String name = source;
    if (source != null && source.startsWith("DB2")) {
        name = "DB2";
    } else if ("Sybase SQL Server".equals(source) || "Adaptive Server Enterprise".equals(source)
|| "ASE".equals(source) || "sql server".equalsIgnoreCase(source)) {
        name = "Sybase";
    }

    return name;
}

public static boolean isNumeric(int sqlType) {
    return -7 == sqlType || -5 == sqlType || 3 == sqlType || 8 == sqlType || 6 == sqlType || 4
== sqlType || 2 == sqlType || 7 == sqlType || 5 == sqlType || -6 == sqlType;
}

public static String lookupColumnName(ResultSetMetaData resultSetMetaData, int column
ndex) throws SQLException {
    String name = resultSetMetaData.getColumnLabel(columnIndex);
    if (!StringUtils.hasLength(name)) {
        name = resultSetMetaData.getColumnName(columnIndex);
    }

    return name;
}

public static String convertUnderscoreNameToPropertyName(@Nullable String name) {
    StringBuilder result = new StringBuilder();
    boolean nextIsUpper = false;
    if (name != null && name.length() > 0) {
        if (name.length() > 1 && name.charAt(1) == '_') {
            result.append(Character.toUpperCase(name.charAt(0)));
        } else {
            result.append(Character.toLowerCase(name.charAt(0)));
        }

        for(int i = 1; i < name.length(); ++i) {
            char c = name.charAt(i);
            if (c == '_') {
                nextIsUpper = true;
            } else if (nextIsUpper) {
                result.append(Character.toUpperCase(c));
                nextIsUpper = false;
            } else {
                result.append(Character.toLowerCase(c));
            }
        }
    }

    return result.toString();
}
}
```

桥接模式

基础解析

- 定义：将抽象部分与它的具体实现部分分离，使它们都可以独立地变化
- 核心：通过组合的方式建立两个类之间联系，而不是继承
- 类型：结构型
- 适用场景：

抽象和具体实现之间增加更多的灵活性

一个类存在两个（或多个）独立变化的维度，且这两个（或多个）维度都需要独立进行扩展

不希望使用继承，或因为多层继承导致系统类的个数剧增

- 优点：

分离抽象部分及其具体实现部分

提高了系统的可扩展性

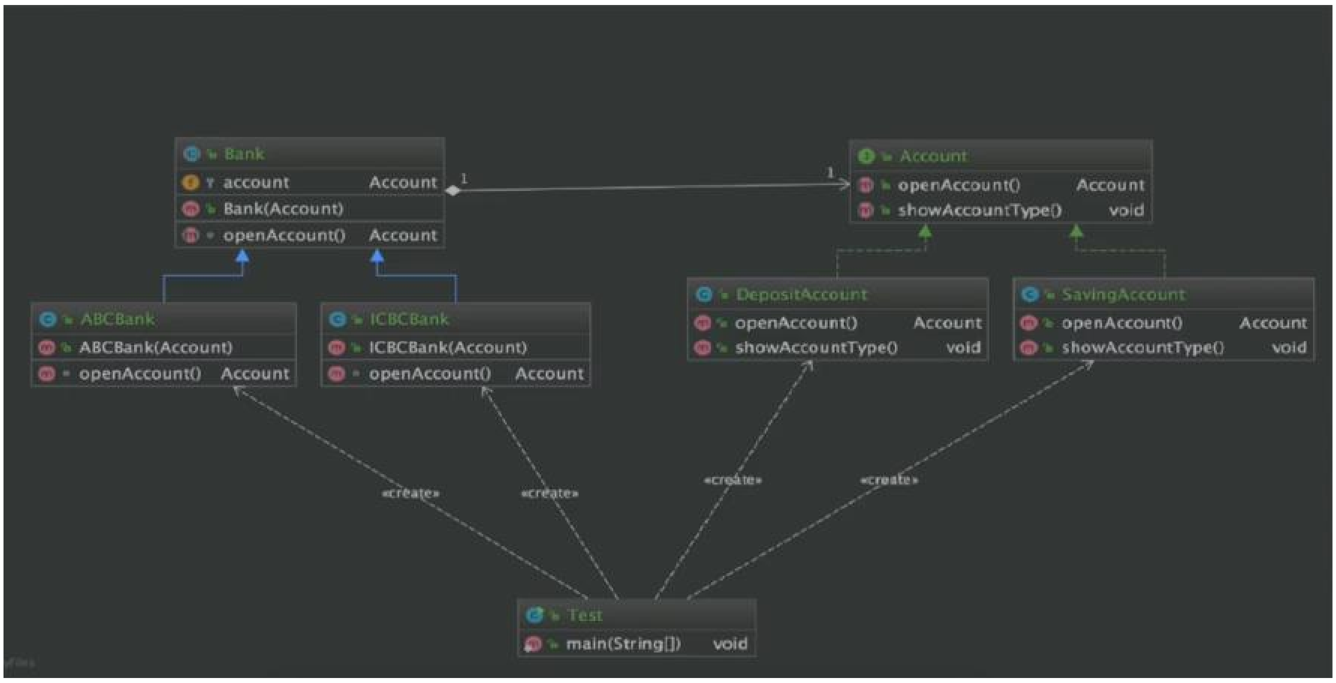
符合开闭原则、合成复用原则

- 缺点：

增加了系统的理解与设计难度

需要正确地识别出系统中两个独立变化的维度

场景解析



```
// 账号
public interface Account {
    Account openAccount();
    void showAccountType();
}

// 活期账号
public class SavingAccount implements Account {
    @Override
    public Account openAccount() {
        System.out.println("打开活期账号");
        return new SavingAccount();
    }

    @Override
    public void showAccountType() {
        System.out.println("这是一个活期账户");
    }
}

// 定期账号
public class DepositAccount implements Account {
    @Override
    public Account openAccount() {
        System.out.println("打开定期账号");
        return new DepositAccount();
    }
    @Override
    public void showAccountType() {
        System.out.println("这是一个定期账户");
    }
}

// 银行
public abstract class Bank {
    protected Account account;

    public Bank(Account account) {
        this.account = account;
    }

    abstract Account openAccount();
}

// 农行
public class ABCBank extends Bank {
    public ABCBank(Account account) {
        super(account);
    }

    @Override
    Account openAccount() {
        System.out.println("打开中国农业银行账号");
        account.openAccount();
        return account;
    }
}

// 工商
public class ICBCBank extends Bank {
    public ICBCBank(Account account) {
        super(account);
    }

    @Override
    Account openAccount() {
        System.out.println("打开中国工商银行账号");
        account.openAccount();
        return account;
    }
}

// 测试类
public class Test {
    public static void main(String[] args) {
        Bank icbcBank = new ICBCBank(new DepositAccount());
        Account icbcAccount = icbcBank.openAccount();
        icbcAccount.showAccountType();

        System.out.println("======================");
        Bank abcBank = new ABCBank(new SavingAccount());
        Account abcAccount = abcBank.openAccount();
        abcAccount.showAccountType();
    }
}
```

源码解析

```
public interface Driver {
    Connection connect(String url, java.util.Properties info) throws SQLException;
    boolean acceptsURL(String url) throws SQLException;
    DriverPropertyInfo[] getPropertyInfo(String url, java.util.Properties info) throws SQLExcepti
n;
}

public class DriverManager {
    // List of registered JDBC drivers
    private final static CopyOnWriteArrayList<DriverInfo> registeredDrivers = new CopyOnWri
eArrayList<>();

    public static synchronized void registerDriver(java.sql.Driver driver, DriverAction da)
        throws SQLException {
        /* Register the driver if it has not already been added to our list */
        if(driver != null) {
            registeredDrivers.addIfAbsent(new DriverInfo(driver, da));
        } else {
            // This is for compatibility with the original DriverManager
            throw new NullPointerException();
        }
        println("registerDriver: " + driver);
    }

    @CallerSensitive
    public static Connection getConnection(String url,
        String user, String password) throws SQLException {
        java.util.Properties info = new java.util.Properties();
        if (user != null) {
            info.put("user", user);
        }
        if (password != null) {
            info.put("password", password);
        }
        return (getConnection(url, info, Reflection.getCallerClass()));
    }
}

class DriverInfo {

    final Driver driver;
    DriverAction da;
    DriverInfo(Driver driver, DriverAction action) {
        this.driver = driver;
        da = action;
    }
    @Override
    public boolean equals(Object other) {
        return (other instanceof DriverInfo)
                && this.driver == ((DriverInfo) other).driver;
    }

    @Override
    public int hashCode() {
        return driver.hashCode();
    }

    @Override
    public String toString() {
        return ("driver[className="  + driver + "]");
    }

    DriverAction action() {
        return da;
    }
}

public interface Connection  extends Wrapper, AutoCloseable {
    PreparedStatement prepareStatement(String sql) throws SQLException;
    void commit() throws SQLException;
    void rollback() throws SQLException;
    void close() throws SQLException;
}
```

组合模式

基础解析

● 定义：将对象组合成树形结构以表示“部分-整体”的层次结构（多个对象组合成一个对象）

● 核心：组合模式使客户端对单个对象和组合对象保持一致的处理方法

● 类型：结构型

● 适用场景：

希望客户端可以忽略组合对象与单个对象的差异

处理一个树形结构

● 优点：

清楚地定义分层次的复杂对象，表示对象的全部或部分层次

让客户端忽略层次的差异，方便对整个层次结构进行控制

简化客户端代码

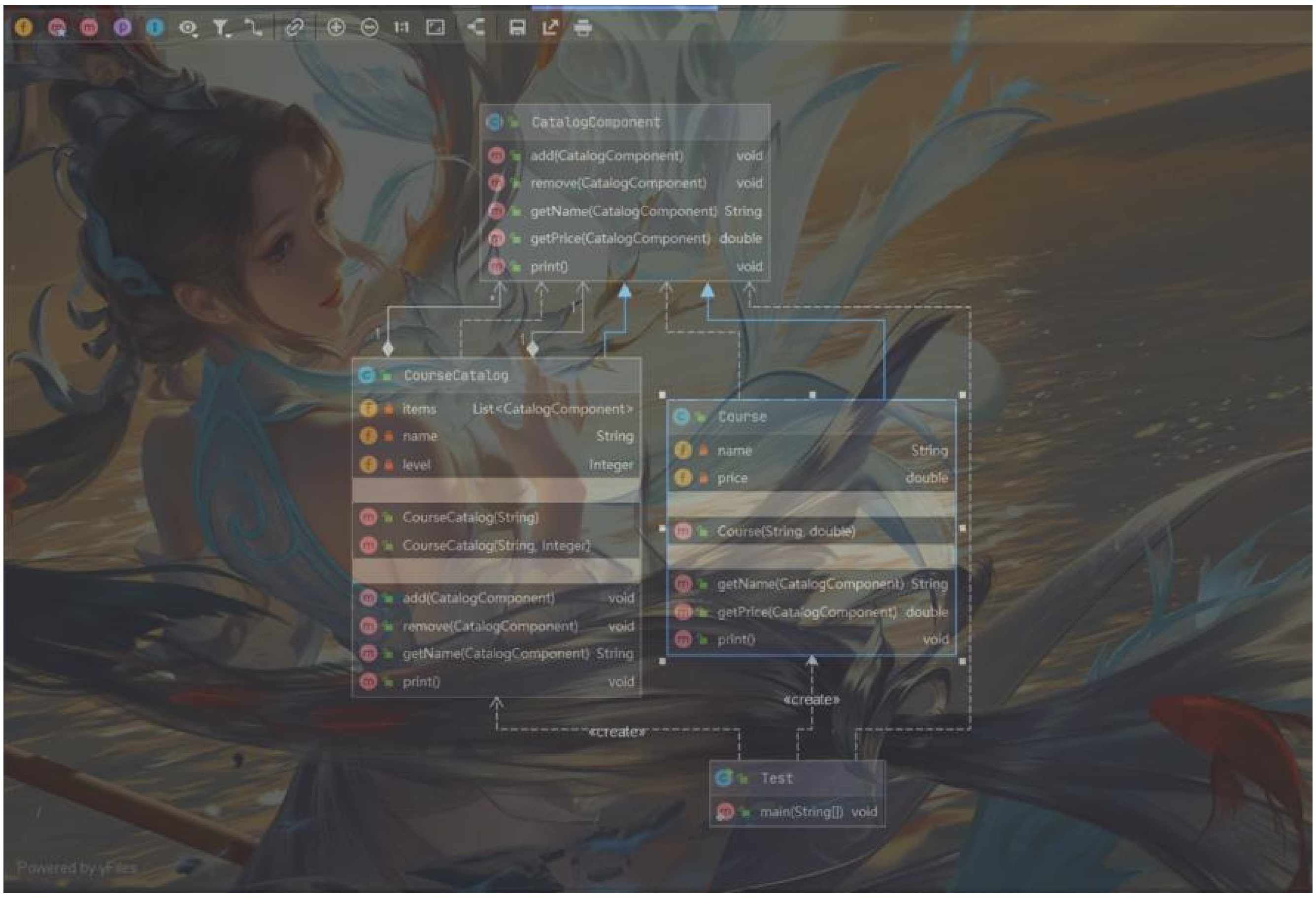
符合开闭原则

● 缺点：

限制类型时会比较复杂

使设计变得更加抽象

场景解析



CatalogComponent
add(CatalogComponent) void
remove(CatalogComponent) void
getName(CatalogComponent) String
getPrice(CatalogComponent) double
print() void
CourseCatalog
items List<CatalogComponent>
name String
level Integer
CourseCatalog(String)
CourseCatalog(String, Integer)
add(CatalogComponent) void
remove(CatalogComponent) void
getName(CatalogComponent) String
print() void
Course
name String
price double
Course(String, double)
getName(CatalogComponent) String
getPrice(CatalogComponent) double
print() void
«create»
«create»
Test
main(String[]) void
Powered by yFiles

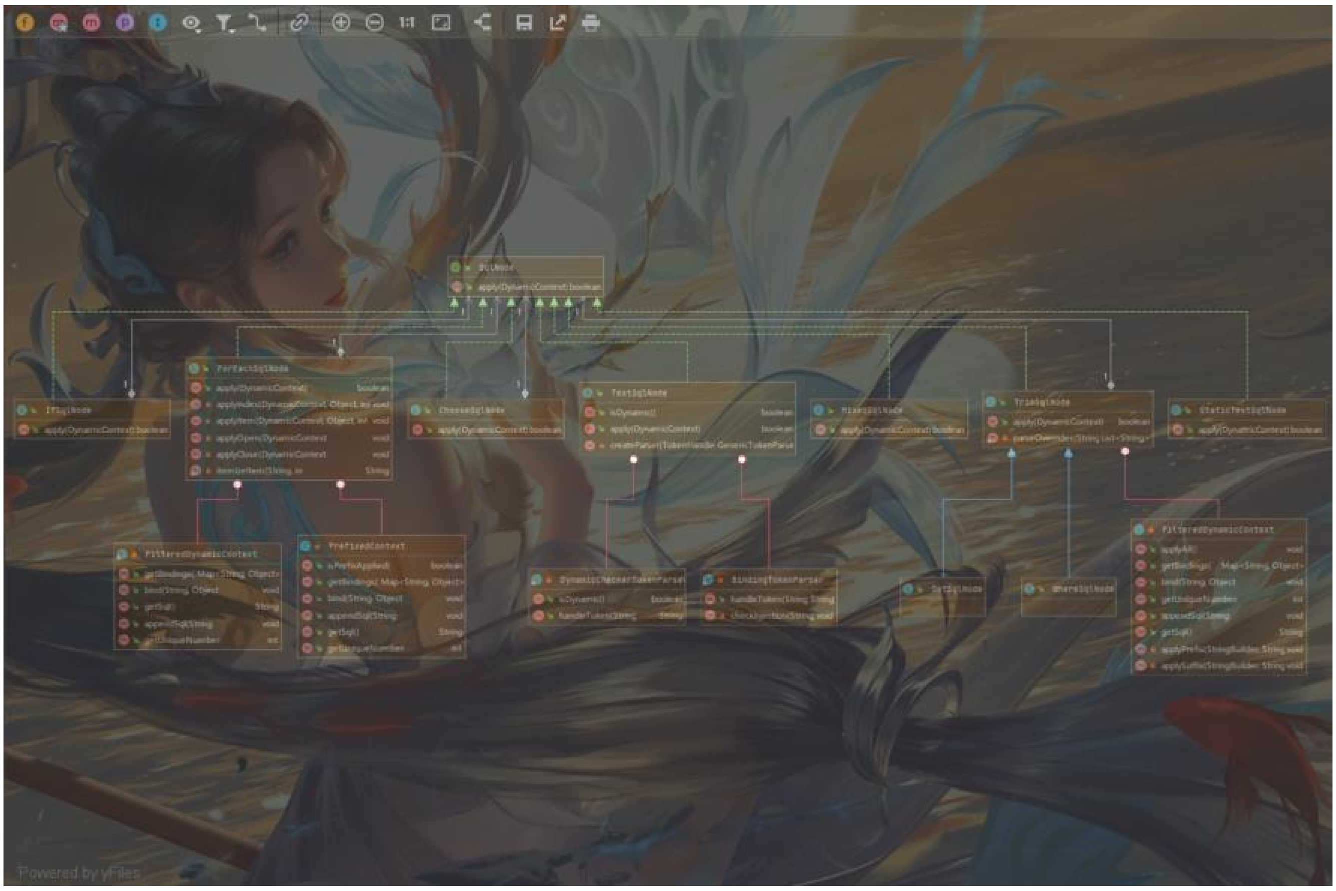
课程目录下可以有课程Course和课程子目录CourseCatalog

```
// 目录组件
public abstract class CatalogComponent {

    public void add(CatalogComponent catalogComponent) {
        throw new UnsupportedOperationException("不支持添加操作");
    }

    public void remove(CatalogComponent catalogComponent) {
        throw new UnsupportedOperationException("不支持删除操作");
    }

    public String getName(CatalogComponent catalogComponent) {
        throw new UnsupportedOperationException("不支持获取名称操作");
    }

    public double getPrice(CatalogComponent catalogComponent) {
        throw new UnsupportedOperationException("不支持获取价格操作");
    }

    public void print() {
        throw new UnsupportedOperationException("不支持打印操作");
    }
```

```
}

// 课程
public class Course extends CatalogComponent {

    private String name;
    private double price;

    public Course(String name, double price) {
        this.name = name;
        this.price = price;
    }

    @Override
    public String getName(CatalogComponent catalogComponent) {
        return this.name;
    }

    @Override
    public double getPrice(CatalogComponent catalogComponent) {
        return this.price;
    }

    @Override
    public void print() {
        System.out.println("Course name： " + name + " Price： " + price);
    }
}

// 课程目录
public class CourseCatalog extends CatalogComponent {

    private List<CatalogComponent> items = new ArrayList<CatalogComponent>();

    private String name;
    private Integer level;

    public CourseCatalog(String name) {
        this.name = name;
    }

    public CourseCatalog(String name, Integer level) {
        this.name = name;
        this.level = level;
    }

    @Override
    public void add(CatalogComponent catalogComponent) {
        items.add(catalogComponent);
    }

    @Override
    public void remove(CatalogComponent catalogComponent) {
        items.remove(catalogComponent);
    }

    @Override
    public String getName(CatalogComponent catalogComponent) {
        return this.name;
    }

    @Override
    public void print() {
        System.out.println("目录名称：" + this.name);
        for (CatalogComponent catalogComponent: items) {
            if (this.level != null) {
                for (int i = 0; i < this.level; i++) {
                    System.out.print("  ");
                }
            }
            catalogComponent.print();
        }
    }

}

// 测试类
public class Test {
    public static void main(String[] args) {
        CatalogComponent linuxCourse = new Course("Linux课程", 11);
        CatalogComponent windowsCourse = new Course("Windows课程", 12);

        CatalogComponent javaCourseCatalog = new CourseCatalog("Java课程目录", 2);
        CatalogComponent mallCourse1 = new Course("电商课程1", 122);
        CatalogComponent mallCourse2 = new Course("电商课程2", 132);
        CatalogComponent designCourse = new Course("Java设计模式", 142);
        javaCourseCatalog.add(mallCourse1);
        javaCourseCatalog.add(mallCourse2);
        javaCourseCatalog.add(designCourse);

        CatalogComponent imoocMainCourseCatalog = new CourseCatalog("慕课网课程主目录",
);
        imoocMainCourseCatalog.add(linuxCourse);
        imoocMainCourseCatalog.add(windowsCourse);
        imoocMainCourseCatalog.add(javaCourseCatalog);

        imoocMainCourseCatalog.print();
    }
}

// 测试结果
目录名称：慕课网课程主目录
  Course name：Linux课程 Price：11.0
  Course name：Windows课程 Price：12.0
  目录名称：Java课程目录
    Course name：电商课程1 Price：122.0
    Course name：电商课程2 Price：132.0
    Course name：Java设计模式 Price：142.0
```

源码解析

```
package java.awt;
public class Container extends Component {
    public Component add(Component comp) {
        addImpl(comp, null, -1);
        return comp;
    }
}

public class HashMap<K,V> extends AbstractMap<K,V>
    implements Map<K,V>, Cloneable, Serializable {
    public void putAll(Map<? extends K, ? extends V> m) {
        putMapEntries(m, true);
    }

    final void putMapEntries(Map<? extends K, ? extends V> m, boolean evict) {
        int s = m.size();
        if (s > 0) {
            if (table == null) { // pre-size
                float ft = ((float)s / loadFactor) + 1.0F;
                int t = ((ft < (float)MAXIMUM_CAPACITY) ?
                         (int)ft : MAXIMUM_CAPACITY);
                if (t > threshold)
                    threshold = tableSizeFor(t);
            }
            else if (s > threshold)
                resize();
            for (Map.Entry<? extends K, ? extends V> e : m.entrySet()) {
                K key = e.getKey();
                V value = e.getValue();
                putVal(hash(key), key, value, false, evict);
            }
        }
    }
}


package org.apache.ibatis.scripting.xmltags;

public interface SqlNode {
    boolean apply(DynamicContext var1);
}
```



享元模式

基础解析

- 定义：提供了减少对象数量从而改善应用所需对象的对象结构的方式
- 核心：运用共享技术有效地支持大量细粒度的对象
- 类型：结构型
- 适用场景：

常应用于系统底层的开发，以便解决系统的性能问题

系统有相似对象、需要缓冲池

- 优点：

减少对象的创建，降低内存中对象的数量，降低系统的内存，提高效率

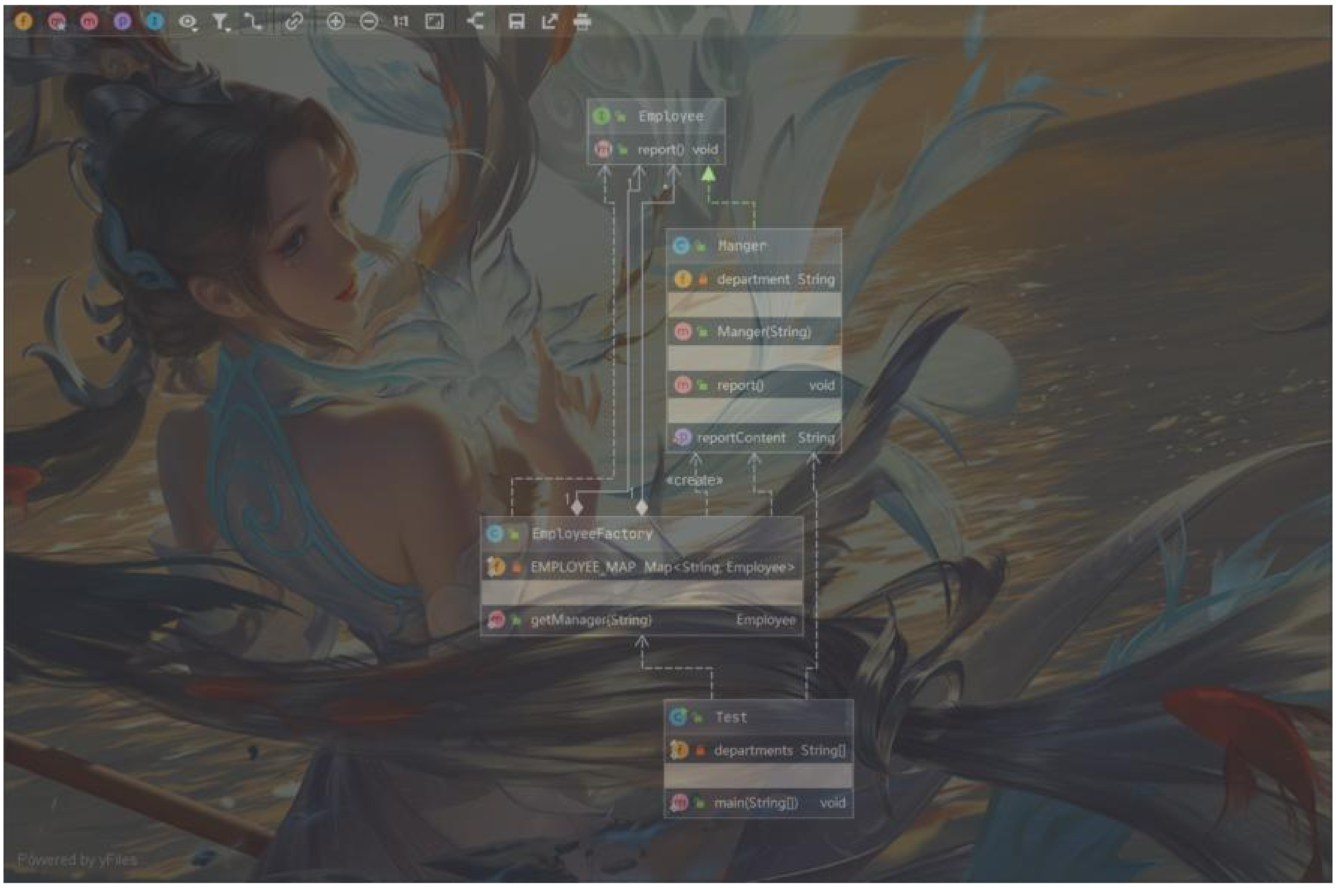
减少内存之外的其他资源占用

- 缺点：

关注内/外部状态、关注线程安全问题

使系统、程序的逻辑复杂化

场景解析



```
public interface Employee {
    void report();
}

public class Manger implements Employee {
    @Override
    public void report() {
        System.out.println(reportContent);
    }

    private String department;
    private String reportContent;

    public Manger(String department) {
        this.department = department;
    }

    public void setReportContent(String reportContent) {
        this.reportContent = reportContent;
    }
}

public class EmployeeFactory {

    private static final Map<String, Employee> EMPLOYEE_MAP = new HashMap<String, Empl
yee>();

    public static Employee getManager(String department) {
        Manger manger = (Manger) EMPLOYEE_MAP.get(department);
        if (manger == null ) {
            manger = new Manger(department);
            System.out.print("创建部门经理" + department);
            String reportContent = department + "部门汇报：主要内容是......";
            manger.setReportContent(reportContent);
            System.out.println(" 创建报告： " + reportContent);
            EMPLOYEE_MAP.put(department, manger);
        }
        return manger;
    }

}

public class Test {
    private static final String departments[] = {"RD", "QA", "PM", "BD"};

    public static void main(String[] args) {
        for (int i = 0; i < 10; i++) {
            // 选一个部门经理
            String department = departments[(int) (Math.random() * departments.length)];
            Manger manger = (Manger) EmployeeFactory.getManager(department);
        }
    }
}
```

源码解析

```
public final class Integer extends Number implements Comparable<Integer> {
    public static Integer valueOf(int i) {
        if (i >= IntegerCache.low && i <= IntegerCache.high)
            return IntegerCache.cache[i + (-IntegerCache.low)];
        return new Integer(i);
    }

    private static class IntegerCache {
         static final int low = -128;
        static final int high;
        static final Integer cache[];

        static {
            // high value may be configured by property
            int h = 127;
            String integerCacheHighPropValue =
                sun.misc.VM.getSavedProperty("java.lang.Integer.IntegerCache.high");
            if (integerCacheHighPropValue != null) {
                try {
                    int i = parseInt(integerCacheHighPropValue);
                    i = Math.max(i, 127);
                    // Maximum array size is Integer.MAX_VALUE
                    h = Math.min(i, Integer.MAX_VALUE - (-low) -1);
                } catch( NumberFormatException nfe) {
                    // If the property cannot be parsed into an int, ignore it.
                }
            }
            high = h;
        }
    }

public class Test {
    public static void main(String[] args) {
        Integer a = Integer.valueOf(100);
        Integer b = 100;
        Integer c = Integer.valueOf(1000);
        Integer d = 1000;
        System.out.println(a == b);  // true
        System.out.println(c == d);  // false
        // 值在 -128-127 间就从Catch中取，否则就new
    }
}
```

行为型模式

策略模式

观察者模式

责任链模式

备忘录模式

模板方法模式

基础解析

- 定义：定义了一个算法的骨架，并允许子类为一个或多个步骤提供实现
- 核心：模板方法使得子类可以在不改变算法结构的情况下，重新定义算法的某些步骤
- 类型：行为型
- 适用场景

一次性实现一个算法的不变的部分，并将可变的行为留给子类来实现

各子类中公共的行为被提取出来并集中到一个公共父类中，从而避免代码重复

- 优点：

提高复用性

提高扩展性

符合开闭原则

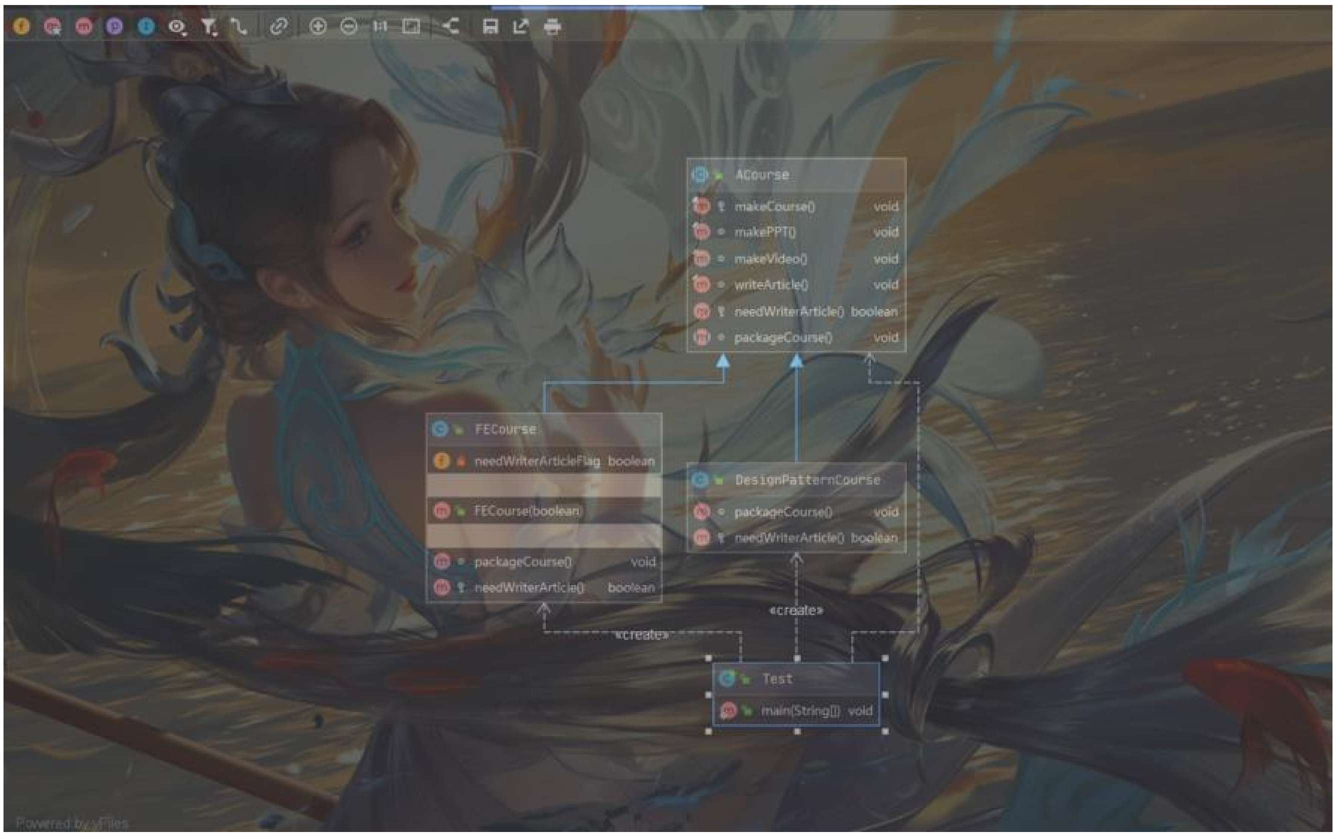
- 缺点：

类数目增加

增加了系统实现的复杂度

继承关系自身缺点，如果父类添加新的抽象方法，所有子类都要改一遍

场景解析



```
// 课程抽象
public abstract class ACourse {
    // 制作课程
    protected final void makeCourse() {
        this.makePPT();
        this.makeVideo();
        if (needWriterArticle()) {
            this.writeArticle();
        }
        this.packageCourse();
    }

    // 制作PPT
    final void makePPT() {
        System.out.println("制作课程PPT");
    }

    // 制作视频
    final void makeVideo() {
        System.out.println("制作课程视频");
    }

    // 编写手记
    final void writeArticle() {
        System.out.println("编写手记");
    }

    // 钩子方法:是否需要编写手记
    protected boolean needWriterArticle() {
        return false;
    }

    abstract void packageCourse();
}

// 课程实现类 设计模式
public class DesignPatternCourse extends ACourse {
    @Override
    void packageCourse() {
        System.out.println("提供Java课程源代码");
    }

    @Override
    protected boolean needWriterArticle() {
        return true;
    }
}

// 课程实现类 前端
public class FECourse extends ACourse {

    private boolean needWriterArticleFlag = false;

    @Override
    void packageCourse() {
        System.out.println("提供前端课程源代码");
        System.out.println("提供前端课程图片等多媒体素材");
    }

    public FECourse(boolean needWriterArticleFlag) {
        this.needWriterArticleFlag = needWriterArticleFlag;
    }

    @Override
    protected boolean needWriterArticle() {
        return this.needWriterArticleFlag;
    }

}

// 测试
public class Test {
    public static void main(String[] args) {
        System.out.println("后端设计模式课程 start");
        ACourse designPatternCourse = new DesignPatternCourse();
        designPatternCourse.makeCourse();
        System.out.println("后端设计模式课程  end");

        System.out.println("====================================");
        System.out.println("前端设计模式课程 start");
        ACourse feCourse = new FECourse(false);
        feCourse.makeCourse();
        System.out.println("前端设计模式课程  end");
    }
}
```

源码解析

```
// AbstractList、AbstractSet、AbstractMap
public abstract class AbstractList<E> extends AbstractCollection<E> implements List<E> {
    public boolean addAll(int index, Collection<? extends E> c) {
        rangeCheckForAdd(index);
        boolean modified = false;
        for (E e : c) {
            add(index++, e);
            modified = true;
        }
        return modified;
    }

    abstract public E get(int index);
}

public class ArrayList<E> extends AbstractList<E>
        implements List<E>, RandomAccess, Cloneable, java.io.Serializable
{
    public E get(int index) {
        rangeCheck(index);

        return elementData(index);
    }
}
```

迭代器模式

基础解析

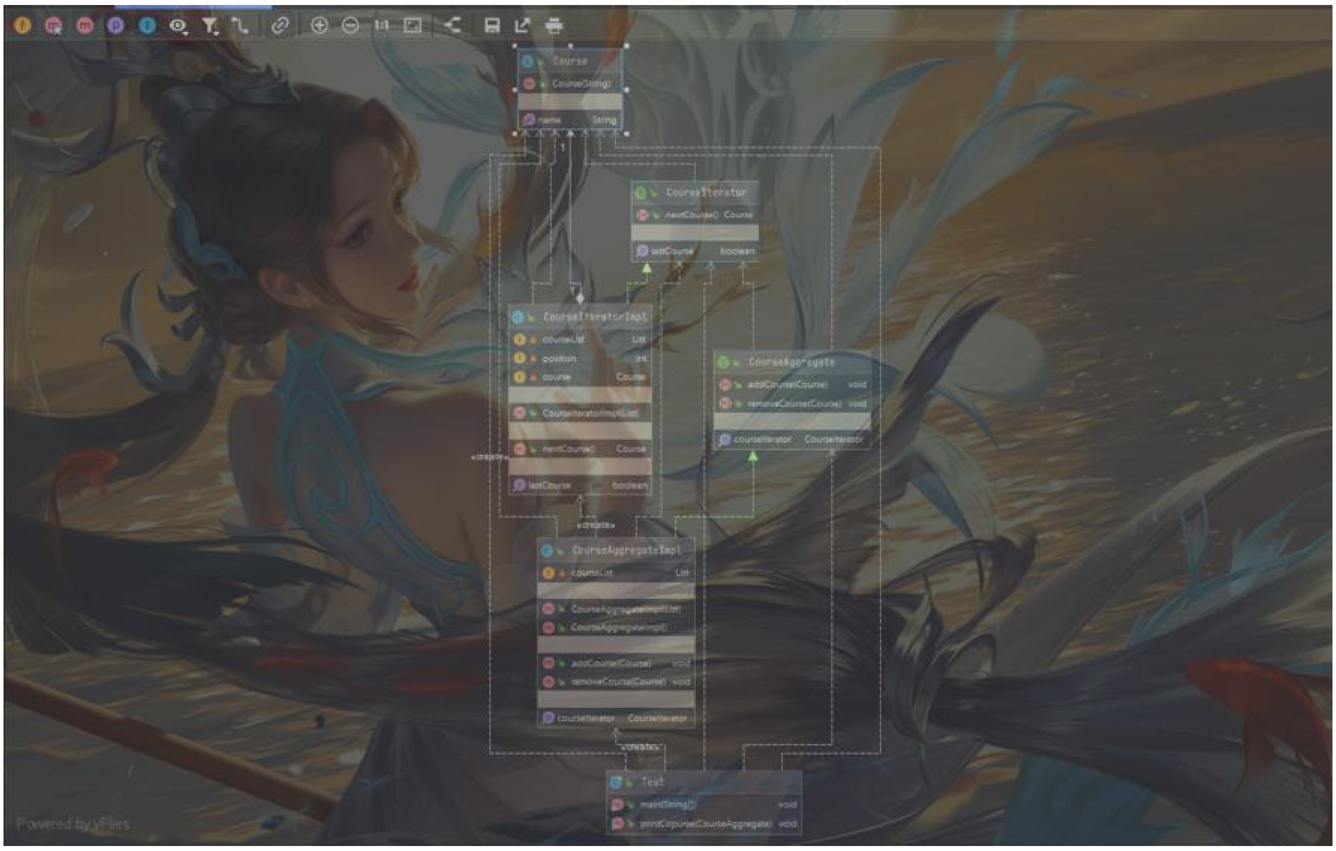
- 定义：提供一种方法，顺序访问一个集合对象中的各个元素，而又不暴露该对象的内部表示
- 类型：行为型
- 适用场景

访问一个集合对象的内容而无需暴露它的内部表示

为遍历不同的集合结构提供一个统一的接口

- 优点：分离了集合对象的遍历行为
- 缺点：类的个数成对增加

场景解析



```
// 课程
public class Course {
    private String name;

    public Course(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}

public interface CourseAggregate {
    void addCourse(Course course);
    void removeCourse(Course course);

    CourseIterator getCourseIterator();
}
public class CourseAggregateImpl implements CourseAggregate {
    private List courseList;

    public CourseAggregateImpl(List courseList) {
        this.courseList = courseList;
    }
    public CourseAggregateImpl() {
        this.courseList = new ArrayList();
    }

    @Override
    public void addCourse(Course course) {
        courseList.add(course);
    }

    @Override
    public void removeCourse(Course course) {
        courseList.remove(course);
    }

    @Override
    public CourseIterator getCourseIterator() {
        return new CourseIteratorImpl(courseList);
    }
}

public interface CourseIterator {
    Course nextCourse();
    boolean isLastCourse();
}
public class CourseIteratorImpl implements CourseIterator {
    private List courseList;
    private int position;
    private Course course;

    public CourseIteratorImpl(List courseList) {
        this.courseList = courseList;
    }

    @Override
    public Course nextCourse() {
        System.out.println("返回课程，位置是：" + position);
        course = (Course) courseList.get(position);
        position++;
        return course;
    }

    @Override
    public boolean isLastCourse() {
        if (position < courseList.size()) {
            return false;
        }
        return true;
    }
}

// 测试
public class Test {
    public static void main(String[] args) {
        Course course1 = new Course("电商课程1");
        Course course2 = new Course("电商课程2");
        Course course3 = new Course("Java设计模式");
        Course course4 = new Course("Python");
        Course course5 = new Course("算法");
        Course course6 = new Course("前端");
        Course course7 = new Course("Linux");

        CourseAggregate courseAggregate = new CourseAggregateImpl();
        courseAggregate.addCourse(course1);
        courseAggregate.addCourse(course2);
        courseAggregate.addCourse(course3);
        courseAggregate.addCourse(course4);
        courseAggregate.addCourse(course5);
        courseAggregate.addCourse(course6);
        courseAggregate.addCourse(course7);

        System.out.println("课程列表================================");
        printCopurse(courseAggregate);
        courseAggregate.removeCourse(course4);
        courseAggregate.removeCourse(course5);
        System.out.println("删除操作后课程列表=====================");
        printCopurse(courseAggregate);
    }

    public static void printCopurse(CourseAggregate courseAggregate) {
        CourseIterator courseIterator = courseAggregate.getCourseIterator();
        while (!courseIterator.isLastCourse()) {
            Course course = courseIterator.nextCourse();
            System.out.println(course.getName());
        }
    }
}

// 测试结果
课程列表================================
返回课程，位置是：0
电商课程1
返回课程，位置是：1
电商课程2
返回课程，位置是：2
Java设计模式
返回课程，位置是：3
Python
返回课程，位置是：4
算法
返回课程，位置是：5
前端
返回课程，位置是：6
Linux
删除操作后课程列表=====================
返回课程，位置是：0
电商课程1
返回课程，位置是：1
电商课程2
```

```
返回课程，位置是：2
Java设计模式
返回课程，位置是：3
前端
返回课程，位置是：4
Linux
```

源码解析

```
public interface Iterator<E> {
    boolean hasNext();
    default void remove() {
        throw new UnsupportedOperationException("remove");
    }
}

private class Itr implements Iterator<E> {
    public boolean hasNext() {
            return cursor != size();
        }

        public E next() {
            checkForComodification();
            try {
                int i = cursor;
                E next = get(i);
                lastRet = i;
                cursor = i + 1;
                return next;
            } catch (IndexOutOfBoundsException e) {
                checkForComodification();
                throw new NoSuchElementException();
            }
        }

        public void remove() {
            if (lastRet < 0)
                throw new IllegalStateException();
            checkForComodification();

            try {
                AbstractList.this.remove(lastRet);
                if (lastRet < cursor)
                    cursor--;
                lastRet = -1;
                expectedModCount = modCount;
            } catch (IndexOutOfBoundsException e) {
                throw new ConcurrentModificationException();
            }
        }
}

private class ListItr extends Itr implements ListIterator<E> {
        ListItr(int index) {
```

```
    cursor = index;
}

public boolean hasPrevious() {
    return cursor != 0;
}

public E previous() {
    checkForComodification();
    try {
        int i = cursor - 1;
        E previous = get(i);
        lastRet = cursor = i;
        return previous;
    } catch (IndexOutOfBoundsException e) {
        checkForComodification();
        throw new NoSuchElementException();
    }
}

public int nextIndex() {
    return cursor;
}

public int previousIndex() {
    return cursor-1;
}

public void set(E e) {
    if (lastRet < 0)
        throw new IllegalStateException();
    checkForComodification();

    try {
        AbstractList.this.set(lastRet, e);
        expectedModCount = modCount;
    } catch (IndexOutOfBoundsException ex) {
        throw new ConcurrentModificationException();
    }
}

public void add(E e) {
    checkForComodification();

    try {
        int i = cursor;
        AbstractList.this.add(i, e);
        lastRet = -1;
        cursor = i + 1;
        expectedModCount = modCount;
    } catch (IndexOutOfBoundsException ex) {
        throw new ConcurrentModificationException();
    }
}
```

```
    }
```

中介者模式

命令模式

访问者模式

解释器模式

状态模式

原文链接：Java 的设计模式