设计模式的分类

创建型： 在创建对象的同时隐藏创建逻辑，不使用 new 直接实例化对象，程序在**判断需要创建哪些对象时更灵活。**包括工厂/抽象工厂/单例/ 建造者/原型模式。

结构型： 通过类和接口间的继承和引用实现创建复杂结构的对象。包括适配器/桥接模式/过滤器/组合/装饰器/外观/享元/代理模式。

行为型： 通过类之间不同通信方式实现不同行为。包括责任链/命名/解释器/迭代器/中介者/备忘录/观察者/状态/策略/模板/访问者模式。

软件设计原则：

设计原则名称	简单定义
开闭原则	对扩展开放，对修改关闭
单一职责原则	一个类只负责一个功能领域中的相应职责
里氏替换原则	所有引用基类的地方必须能透明地使用其子类的对象
依赖倒置原则	依赖于抽象，不能依赖于具体实现
接口隔离原则	类之间的依赖关系应该建立在最小的接口上
合成/聚合复用原则	尽量使用合成/聚合，而不是通过继承达到复用的目的
迪米特法则	一个软件实体应当尽可能少的与其他实体发生相互作用

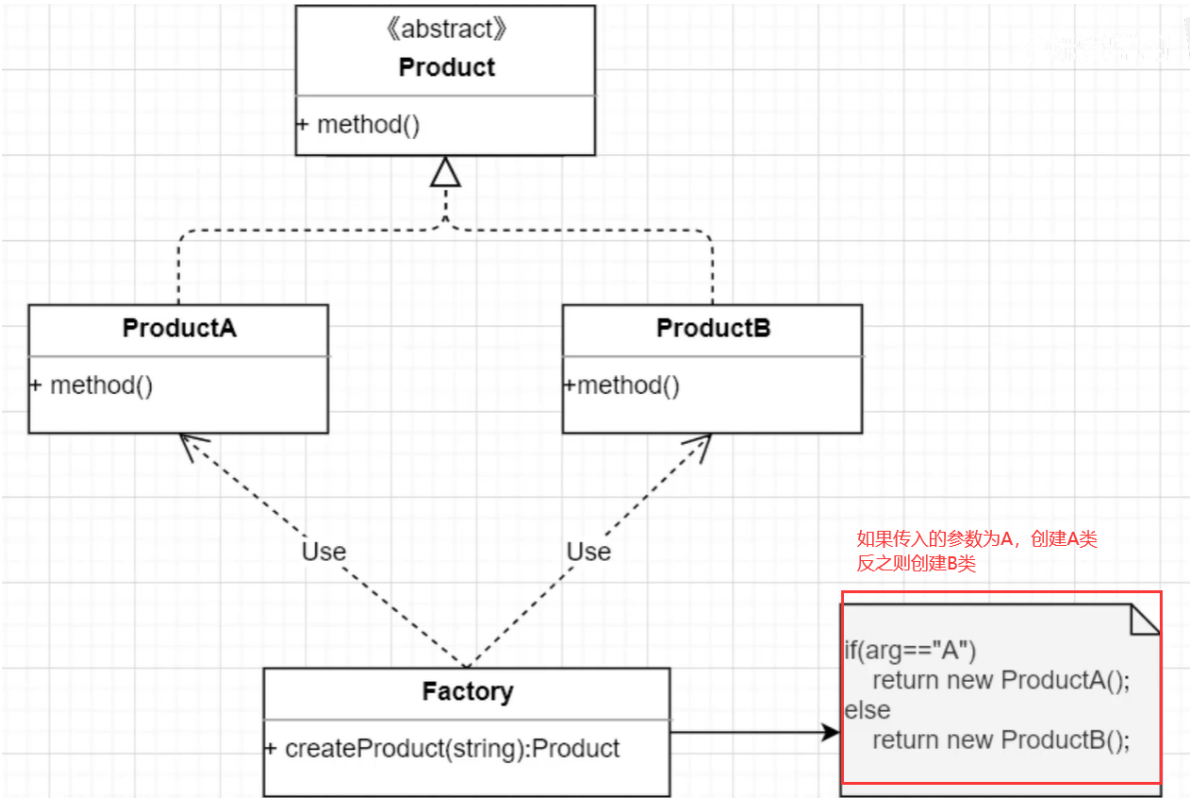
最近看到一个很好的形容，比如说一个厂的一个流水线正在生产拖拉机，现在又来了一个新订单要生产口罩，那么你有两种选择，你是把拖拉机那个流水线改成生产口罩的流水线呢，还是再开一条生产线生产口罩呢?

答案当然是再开一条，这里面的原理就是开闭原则

工厂模式

说一说简单工厂模式

简单工厂模式指由一个工厂对象来创建实例，客户端不需要关注创建逻辑，只需要提供传入工厂的参数就行了，可以根据参数的不同返回创建不同类的实例。简单工厂模式**专门定义一个类**来负责创建其他类的实例,被创建的实例通常都具有共同的父类。



优点：

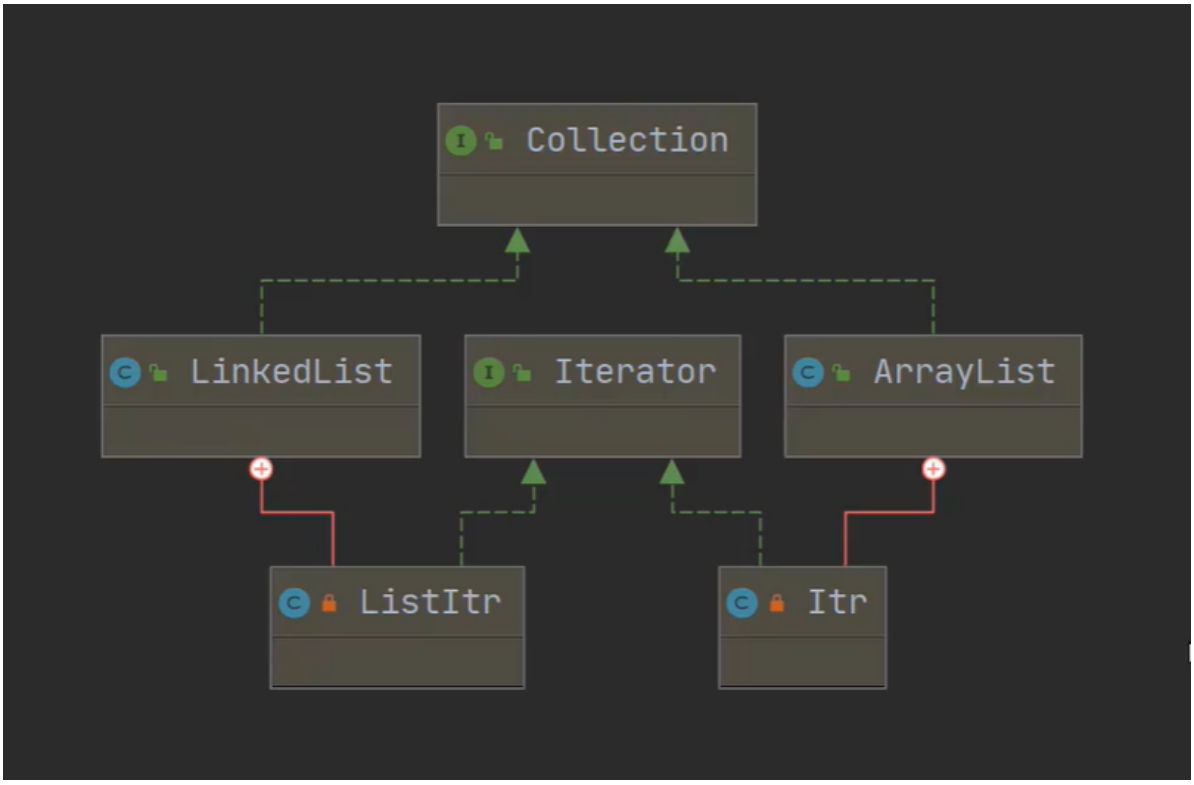
优点在于实现对象的创建和使用分离，创建呢，完全交给专门的工厂类负责，客户端程序员不关心怎么创建，只关心怎么使用

缺点：

缺点就很明显了，简单工厂类不够灵活，如果我们新增一个产品的话就要修改工厂类的判断逻辑，如果产品很多的话，这个判断逻辑，想想看，好多if，特别的复杂

工厂方法模式了解吗

举个例子 JDK里面有个Collection集合 这里面抽象产品就是Iterator，ArrayList和LinkedList里面都有Iterator的具体实现



工厂方法模式和**简单工厂模式中工厂负责生产所有产品**相比，工厂方法模式将生成具体产品的任务分发给具体的产品工厂。

也就是定义一个抽象工厂，其定义了产品的生产接口，但不负责具体的产品，将生产任务交给不同的子类工厂。这样不用通过指定类型来创建对象了。

工厂模式就是富士康(提供场地人员制作方法 比如 生产 运输 组装) 富士康里面有苹果的生产线 有华为的生产线 这些生产线来进行具体的生产，组装，运输。
创建一个接口，然后再根据不同的需求实现这个接口
比如 List list = new ArrarList();
List 是接口(富士康) ArrayList(苹果生产线)是具体实现

抽象工厂模式了解吗?

抽象工厂和工厂模式的区别在于，工厂模式只生产一大类对象，而抽象工厂可以生产好几大类对象，就是所谓1和n的区别。抽象工厂模式存在灵活性差的问题，变更时会违反开闭原则。

抽象工厂模式是工厂模式的一个扩展，如果抽象工厂模式只针对一类产品，就会退化成工厂模式

抽象工厂模式通过在 AbstarctFactory 中增加创建产品的接口，并在具体子工厂中实现新加产品的创建，当然前提是子工厂支持生产该产品。否则继承的这个接口可以什么也不干。

```
//抽象工厂
interface AbstractFactory{
    Phone createPhone(string param);
    Mask createMask(String param);
}
//具体工厂
class superFactory implements AbstractFactory{
    aoverride
    public Phone createPhone(String param) {
        return new iPhone();
    }
    @Override
    public Mask createHask(String param) {
        return new N95(;
    }
}
//产品大类--手机
interface Phone{ }
class iPhone implements Phone{ }
//产品大类--口罩
interface Mask{ }
class N95 implements Mask{ }
```

单例模式

什么是单例模式？单例模式的特点是什么？

单例模式属于创建型模式，**一个单例类在任何情况下都只存在一个实例**， 构造方法必须是**私有**的、由自己创建一个静态变量存储实例，对外提供一个静态公有方法获取实例。

优点是内存中只有一个实例，减少了开销，尤其是频繁创建和销毁实例的情况下。并且可以避免对资源的多重占用，如果一个类需要访问一些共享的资源，例如数据库连接或者文件句柄，那么使用单例模式可以确保只有一个实例能够访问这些资源，从而避免资源冲突和竞争的问题。

单例模式的常见写法有哪些？

线程安全 饿汉式

类一创建就创建对象，这种方法常用，但是容易产生垃圾对象，浪费内存空间。

优点：线程安全，没有加锁，执行效率较高

缺点：不是懒加载（使用的时候再创建对象），在类加载时就初始化，浪费内存空间

饿汉式基于**类加载机制**避免了多线程问题，但是如果类被不同的类加载器加载就会创建不同的实例对象

```
package com.design.singleton;

/**
 * 饿汉单例模式
```

```
 */
public class EagerSingleton {
    //饿汉模式在类被加载时，静态变量 eagerSingleton 就会被初始化，
    //此时类的私有构造函数会被调用，单例类的唯一实例会被创建
    private static final EagerSingleton eagerSingleton = new EagerSingleton();

    private EagerSingleton(){
    }

    public static EagerSingleton getInstance(){
        return eagerSingleton;
    }
}
```

调用

```
public class TestMain {
    public static void main(String[] args) {
        EagerSingleton s = EagerSingleton.getInstance();
    }
}
```

除了直接new，还可以用静态代码块实例化对象实现饿汉式

```
package com.design.singleton;

/**
 * 静态块饿汉单例模式
 */
public class StaticEagerSingleton {

    private static  StaticEagerSingleton staticEagerSingleton;

    static {
        staticEagerSingleton = new StaticEagerSingleton();
    }

    private StaticEagerSingleton(){
    }

    public static StaticEagerSingleton getInstance(){
        return staticEagerSingleton;
    }
}
```

线程不安全 懒汉式

这种方式在单线程下使用没有问题，但是多线程下没法保证单例，会出现问题哦

优点：懒加载（用到了再创建）

缺点：线程不安全

```
public class Singleton {
```

```
    // 1、私有化构造方法
    private Singleton(){ }
    // 2、定义一个静态变量指向自己类型
    private static Singleton instance;
    // 3、对外提供一个公共的方法获取实例
    public static Singleton getInstance() {
        // 判断为 null 的时候再创建对象
        if (instance == null) {
            //当多线程的时候，可能有多个线程到这里
            instance = new Singleton();
        }
        return instance;
    }
}
```

线程安全 懒汉式

保证线程安全那肯定得加锁咯，问题是加在哪里，这里是一个比较大的部分，当然了，结论就是你锁住的代码块越少越好（也就是足够细粒度的锁）。

这里先演示加在方法上， 存在的问题是：每一次调用 getInstance 获取实例时都需要加锁和释放锁，这样是非常影响性能的

优点：懒加载，线程安全

缺点：效率较低

```
public class Singleton {
    // 1、私有化构造方法
    private Singleton(){ }
    // 2、定义一个静态变量指向自己类型
    private static Singleton instance;
    // 3、对外提供一个公共的方法获取实例
    public synchronized static Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}
```

双重检查锁（DCL， 即 double-checked locking）

```
public class Singleton {
    // 1、私有化构造方法
    private Singleton() {
    }
    // 2、定义一个静态变量,这里注意是volatile的
    private volatile static Singleton instance;
    // 3、对外提供一个公共的方法获取实例
    public static Singleton getInstance() {
        // 第一重检查是否为 null，这里是为了之后不加锁进行的过滤
        if (instance == null) {
            // 使用 synchronized 加锁。为了防止多线程
            synchronized (Singleton.class) {//因为是静态方法，只能锁住当前类
```

```
                //第二重检查是否为null
                if(instance==null){
                    //new关键字创建对象不是原子操作
                    instance=new Singleton();
                }
            }
        }
        return instance;
    }
}
```

优点：懒加载，线程安全，效率较高

缺点：实现较复杂

这里的双重检查是指两次非空判断，锁指的是 synchronized 加锁，为什么要进行双重判断，其实很简单，第一重判断，如果实例已经存在，那么就不再需要进行同步操作，而是直接返回这个实例，如果没有创建，才会进入同步块，同步块的目的与之前相同，目的是为了防止有多个线程同时调用时，导致生成多个实例，有了同步块，每次只能有一个线程调用访问同步块内容，当第一个抢到锁的调用获取了实例之后，这个实例就会被创建，之后的所有调用都不会进入同步块，直接在第一重判断就返回了单例。

关于内部的第二重空判断的作用，当多个线程一起到达锁位置时，进行锁竞争，其中一个线程获取锁，如果是第一次进入则为 null，会进行单例对象的创建，完成后释放锁，其他线程获取锁后就会被空判断拦截，直接返回已创建的单例对象。

其中最关键的一个点就是 volatile 关键字的使用，双重检查锁中使用 volatile 的两个重要特性：可见性、禁止指令重排序

这里为什么要使用 volatile ？

这是因为 new 关键字创建对象不是原子操作，创建一个对象会经历下面的步骤：

\1. 在堆内存开辟内存空间

\2. 调用构造方法，初始化对象

\3. 引用变量指向堆内存空间

对应字节码指令如下：

为了提高性能，编译器和处理器常常会在不影响代码执行结果的情况下，对既定的代码执行顺序进行指令重排序

所以经过指令重排序之后，创建对象的执行顺序可能为 1 2 3 或者 1 3 2，因此当某个线程在乱序运行 1 3 2 指令的时候，引用变量指向堆内存空间，这个对象不为 null，但是没有初始化，其他线程有可能这个时候进入了 getInstance 的第一个 if(instance == null) 判断不为 nulll，**导致错误使用了没有初始化的非 null 实例，这样的话就会出现异常，这个就是著名的 DCL 失效问题。**

当我们在引用变量上面添加 volatile 关键字以后，会通过在创建对象指令的前后添加内存屏障（读、屏障）来禁止指令重排序，就可以避免这个问题，而且对 volatile 修饰的变量的修改对其他任何线程都是可见的。

静态内部类

```
public class Singleton {
    // 1、私有化构造方法
    private Singleton() {
    }
    // 2、对外提供获取实例的公共方法
    public static Singleton getInstance() {
        return InnerClass.INSTANCE;
    }
    // 定义静态内部类
    private static class InnerClass{
        private final static Singleton INSTANCE = new Singleton();
    }
}
```

优点：懒加载，线程安全，效率较高，实现简单

1)这种方式采用了类装载的机制来保证初始化实例时只有一个线程。

2)静态内部类方式在Singleton类被装载时并不会立即实例化，而是在需要实例化时，调用getInstance方法，于会装载SingletonInstance类，从而完成Singleton的实例化。

3)类的静态属性只会在第一次加载类的时候初始化，所以在这里，JVM帮助我们保证了线程的安全性，在类进行初始化时，别的线程是无法进入的。

下面是JavaGuide里面的，我没看太懂

静态内部类单例是如何实现懒加载的呢？首先，我们先了解下类的加载时机。

虚拟机规范要求**有且只有 5 种情况**必须立即对类进行初始化：

1. 遇到 new 、 getstatic 、 putstatic 、 invokestatic 这 4 条字节码指令 时。生成这 4 条指令最常见的 Java 代码场景是：

- - 使用 new 关键字实例化对象的时候
  - 读取或设置一个类的静态字段（final 修饰除外，被 final 修饰的静态字段是常量，已在编译期把结果放入常量池）的时候
  - 以及调用一个类的静态方法的时候

\2. 使用 java.lang.reflect 包方法对类进行反射调用的时候。

\3. 当初始化一个类的时候，如果发现其父类还没有进行过初始化，则需要先触发其父类的初始化。

\4. 当虚拟机启动时，用户需要指定一个要执行的主类（包含 main()的那个类），虚拟机会先初始化这个主类。

\5. 当使用 JDK 1.7 的动态语言支持时，如果一个 java.lang.invoke.MethodHandle 实例最后的解析结果是 REF_getStatic 、 REF_putStatic 、 REF_invokeStatic 的方法句柄，则需要先触发这个方法句柄所对应的类的初始化。

枚举单例

```
public enum Singleton {
    INSTANCE;
    public void doSomething(String str) {
        System.out.println(str);
    }
}
```

优点：简单，高效，线程安全，可以避免通过反射破坏枚举单例

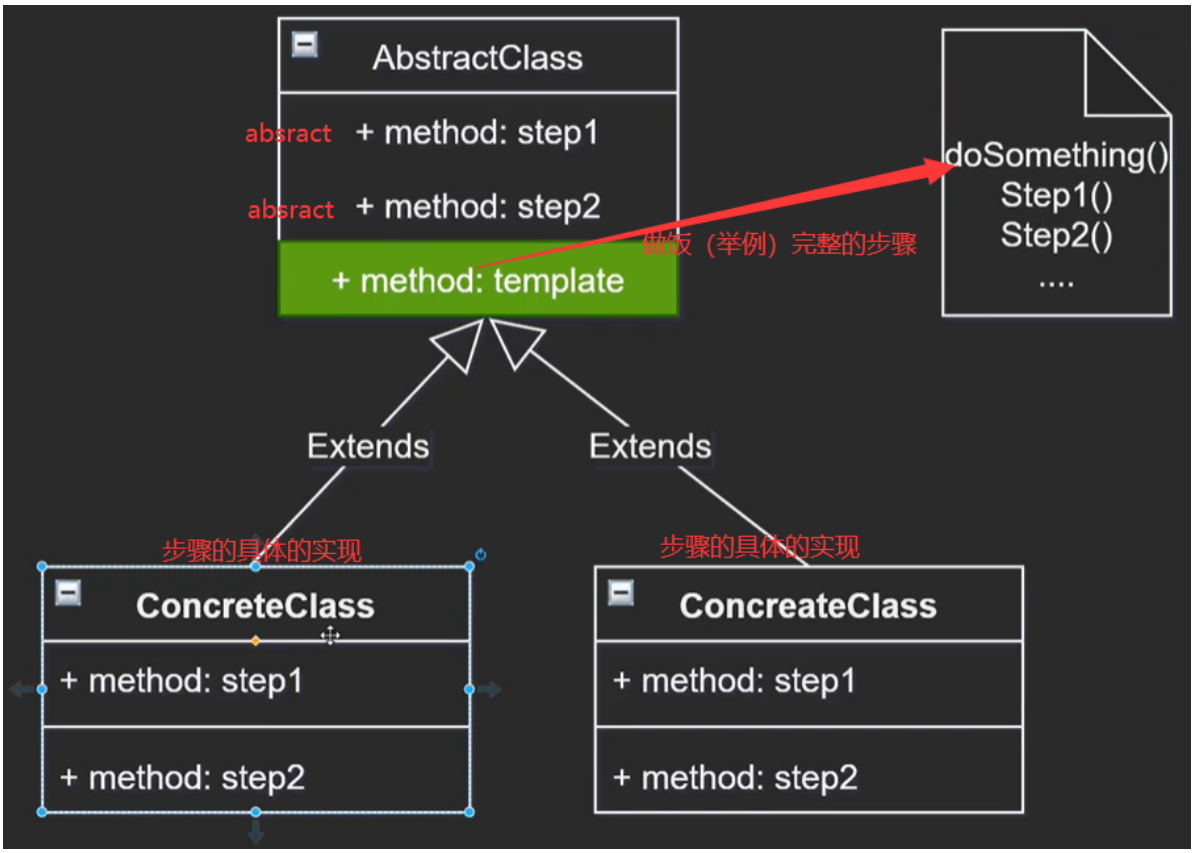
并且枚举类里面的变量都是static final 的，这意味着什么，意味着只有你用到这个类的时候，这些实例它才会加载，不会浪费空间

```
//可以用以下代码来调用
Singleton singleton=Singleton.INSTANCE;
```

模板方法模式

模板方法模式（Template Method Pattern），又叫模板模式(Template Pattern)，在一个抽象类公开定义了执行它的方法的模板。它的子类可以按需要重写方法实现，但调用将以抽象类中定义的方式进行。

简单说，模板方法模式，定义一个操作中的算法的骨架，而将一些步骤延迟到子类中，使得子类可以不改变一个算法的结构，就可以重定义该算法的某些特定步骤，这种类型的设计模式属于行为型模式。



就是把步骤给你定义好了，具体每一步具体干什么，你可以自己去定义

```
public class TemplateMethodPattern {
    public static void main(string[] args) {
        Cooking cooking = new CookingFood();
        cooking.cook();
    }
}
abstract class Cooking{
    protected abstract void step1(;protected abstract void step2();
    public void cook(){
        system.out.println("做饭开始");
        step1();
        step2();
        system.out.println("做饭结束");
    }
}
class CookingFood extends cooking{
    @override
    protected void step1() {
        system.out.println("放鸡蛋和西红柿");
    }
    @override
    protected void step2() {
        system.out.println("少放盐多放味精");
    }
}
```

适配器模式

适配器模式了解吗

所谓适配器模式就是将一个类的接口，转换成客户期望的另一个接口。它可以让原本两个不兼容的接口能够无缝完成对接。 就比如我们手机直接冲交流电肯定不行，我们需要一个充电器，把我们家里的插座的220V交流电转换成直流电，才能给我们手机充电。

在这个例子中：

- 适配器可以是一个转接头或者一个可以将充电器的电压输出转换为手机所需电压的设备。
- 适配者是充电器，主要的功能还是转接头的
- 目标类是手机

作为中间件的适配器将目标类和适配者解耦，增加了类的透明性和可复用性。

适配器模式的优缺点
优点：

1. 提高了类的复用；
2. 组合若干关联对象形成对外提供统一服务的接口；
3. 扩展性、灵活性好。

缺点：

1. 过多使用适配模式容易造成代码功能和逻辑意义的混淆。
2. 部分语言对继承的限制，可能至多只能适配一个适配者类

代理模式（proxy pattern）

什么是代理模式

代理模式的本质是一个**中间件**，相当于一个传话/办事的人，主要目的是解耦合**服务提供者**和**使用者。**使用者通过**代理**间接访问服务提供者。这种模式便于服务提供者的封装和服务提供者对于使用者的控制（访问权限、访问频率、访问方式），他是一种结构模式。

代理对象可以扮演服务对象的门卫或过滤器的角色，可以过滤和控制对服务对象的访问请求。例如，代理对象可以验证客户端的身份、检查访问权限、限制对服务对象的访问频率等等，以确保服务对象的安全和稳定。

此外，代理模式还可以对服务对象的访问进行优化。代理对象可以缓存服务对象的结果，避免重复执行相同的操作。代理对象还可以延迟服务对象的创建和初始化，以避免在不需要服务对象时浪费系统资源。

因此，代理模式可以帮助服务提供者更好地控制服务对象的访问，确保服务对象的安全和稳定，并优化系统性能。

静态代理

静态代理中，我们对目标对象的每个方法的增强都是手动完成的（*后面会具体演示代码*），非常不灵活（*比如接口一旦新增加方法，目标对象和代理对象都要进行修改*）且麻烦(*需要对每个目标类都单独写一个代理类*）。 实际应用场景非常非常少，日常开发几乎看不到使用静态代理的场景。

上面我们是从实现和应用角度来说的静态代理，从 JVM 层面来说， **静态代理在编译时就将接口、实现类、代理类这些都变成了一个个实际的 class 文件。**

静态代理实现步骤:

1. 定义一个接口及其实现类;
2. 创建一个代理类同样实现这个接口
3. **将目标对象注入进代理类**，然后在代理类的对应方法调用目标类中的对应方法。这样的话，我们就可以通过代理类屏蔽对目标对象的访问，并且可以在目标方法执行前后做一些自己想做的事情。

下面通过代码展示!

```
1.定义发送短信的接口
public interface SmsService {
    String send(String message);
}
2.实现发送短信的接口
public class SmsServiceImpl implements SmsService {
    public String send(String message) {
        System.out.println("send message:" + message);
        return message;
    }
}
3.创建代理类并同样实现发送短信的接口
public class SmsProxy implements SmsService {

    private final SmsService smsService;

    public SmsProxy(SmsService smsService) {
        this.smsService = smsService;
    }

    @Override
```

```
    public String send(String message) {
        //调用方法之前，我们可以添加自己的操作
        System.out.println("before method send()");
        smsService.send(message);
        //调用方法之后，我们同样可以添加自己的操作
        System.out.println("after method send()");
        return null;
    }
}
4.实际使用
public class Main {
    public static void main(String[] args) {
        SmsService smsService = new SmsServiceImpl();
        SmsProxy smsProxy = new SmsProxy(smsService);
        smsProxy.send("java");
    }
}

运行上述代码之后，控制台打印出：before method send()
send message:java
after method send()
可以输出结果看出，我们已经增加了 SmsServiceImpl 的send()方法。
```

动态代理

JDK动态代理

介绍：

在 Java 动态代理机制中 InvocationHandler 接口和 Proxy 类是核心。

Proxy 类中使用频率最高的方法是：newProxyInstance() ，这个方法主要用来生成一个代理对象。

```
    public static Object newProxyInstance(ClassLoader loader,
                                           Class<?>[] interfaces,
                                           InvocationHandler h)
        throws IllegalArgumentException
    {
        ......
    }
```

这个方法一共有 3 个参数：

1. **loader** :类加载器，用于加载代理对象。
2. **interfaces** : 被代理类实现的一些接口；
3. **h** : 实现了 InvocationHandler 接口的对象；

要实现动态代理的话，还必须需要实现InvocationHandler 来自定义处理逻辑。 当我们的动态代理对象调用一个方法时，这个方法的调用就会被转发到实现InvocationHandler 接口类的 invoke 方法来调用。

```
public interface InvocationHandler {

    /**
     * 当你使用代理对象调用方法的时候实际会调用到这个方法
     */
    public Object invoke(Object proxy, Method method, Object[] args)
        throws Throwable;
}
```

invoke() 方法有下面三个参数：

1. **proxy** :动态生成的代理类
2. **method** : 与代理类对象调用的方法相对应
3. **args** : 当前 method 方法的参数

也就是说：**你通过**Proxy** 类的 newProxyInstance() 创建的代理对象在调用方法的时候，实际会调用到实现**InvocationHandler** 接口的类的 invoke()**方法。** 你可以在 invoke() 方法中自定义处理逻辑，比如在方法执行前后做什么事情。

JDK 动态代理类使用步骤

1. 定义一个接口及其实现类；
2. 自定义 InvocationHandler 并重写invoke方法，在 invoke 方法中我们会调用原生方法（被代理类的方法）并自定义一些处理逻辑；
3. 通过 Proxy.newProxyInstance(ClassLoader loader,Class<?>[] interfaces,InvocationHandler h) 方法创建代理对象；

```
1.定义发送短信的接口
public interface SmsService {
    String send(String message);
}
2.实现发送短信的接口
public class SmsServiceImpl implements SmsService {
    public String send(String message) {
        System.out.println("send message:" + message);
        return message;
    }
}
3.定义一个 JDK 动态代理类
public class DebugInvocationHandler implements InvocationHandler {
    /**
     * 代理类中的真实对象
     */
    private final Object target;

    public DebugInvocationHandler(Object target) {
        this.target = target;
    }
    public Object invoke(Object proxy, Method method, Object[] args) throws
InvocationTargetException, IllegalAccessException {
        //调用方法之前，我们可以添加自己的操作
        System.out.println("before method " + method.getName());
        Object result = method.invoke(target, args);
```

```
        //调用方法之后，我们同样可以添加自己的操作
        System.out.println("after method " + method.getName());
        return result;
    }
}
```

invoke() 方法：当我们的动态代理对象调用原生方法的时候，最终实际上调用到的是 invoke() 方法，然后 invoke() 方法代替我们去调用了被代理对象的原生方法。

```
4.获取代理对象的工厂类
public class JdkProxyFactory {
    public static Object getProxy(Object target) {
        return Proxy.newProxyInstance(
                target.getClass().getClassLoader(), // 目标类的类加载
                target.getClass().getInterfaces(),  // 代理需要实现的接口，可指定多个
                new DebugInvocationHandler(target)   // 代理对象对应的自定义
InvocationHandler
        );
    }
}
```

getProxy() ：主要通过Proxy.newProxyInstance（）方法获取某个类的代理对象

```
5.实际使用
SmsService smsService = (SmsService) JdkProxyFactory.getProxy(new
SmsServiceImpl());
smsService.send("java");
```

运行上述代码之后，控制台打印出：

```
before method send
send message:java
after method send
```

CGLIB动态代理

介绍

JDK 动态代理有一个最致命的问题是其只能代理实现了接口的类。为了解决这个问题，我们可以用 CGLIB 动态代理机制来避免。

CGLIB（Code Generation Library）是一个基于ASM的字节码生成库，它允许我们在运行时对字节码进行修改和动态生成。CGLIB通过继承方式实现代理，很多框架都使用到了CGLIB，比如Spring中的AOP模块：如果目标对象实现了接口，则默认采用JDK动态代理，否则则采用CGLIB动态代理

在 CGLIB 动态代理机制中 MethodInterceptor 接口和 Enhancer 类是核心。

你需要自定义 MethodInterceptor 并重写 intercept 方法，intercept 用于拦截增强被代理类的方法。

```
public interface MethodInterceptor extends Callback{
    // 拦截被代理类中的方法
    public Object intercept(Object obj,
                            java.lang.reflect.Method method,
                            Object[] args,
                            MethodProxy proxy) throws Throwable;
}
```

1. **obj** : 被代理的对象（需要增强的对象）
2. **method** : 被拦截的方法（需要增强的方法）
3. **args** : 方法入参
4. **proxy** : 用于调用原始方法

你可以通过 Enhancer类来动态获取被代理类，**当代理类调用方法的时候，实际调用的是 MethodInterceptor 中的 intercept 方法。**

使用步骤

- 定义一个类;
- 自定义 MethodInterceptor 并重写 intercept 方法，intercept 用于拦截增强被代理类的方法，和 JDK 动态代理中的 invoke 方法类似;
- 通过 Enhancer 类的 create()创建代理类;

代码演示

导入依赖

```
<dependency>
  <groupId>cglib</groupId>
  <artifactId>cglib</artifactId>
  <version>3.3.0</version>
</dependency>
1.实现一个使用阿里云发送短信的类
public class AliSmsService {
    public String send(String message) {
        System.out.println("send message:" + message);
        return message;
    }
}
2.自定义 MethodInterceptor（方法拦截器）
public class DebugMethodInterceptor implements MethodInterceptor {
    /**
     * @param o           被代理的对象（需要增强的对象）
     * @param method      被拦截的方法（需要增强的方法）
     * @param args        方法入参
     * @param methodProxy 用于调用原始方法
     */
    @Override
    public Object intercept(Object o, Method method, Object[] args, MethodProxy
methodProxy) throws Throwable {
        //调用方法之前，我们可以添加自己的操作
        System.out.println("before method " + method.getName());
        Object object = methodProxy.invokeSuper(o, args);
        //调用方法之后，我们同样可以添加自己的操作
        System.out.println("after method " + method.getName());
        return object;
    }
}
3.获取代理类
public class CglibProxyFactory {
    public static Object getProxy(Class<?> clazz) {
        // 创建动态代理增强类
        Enhancer enhancer = new Enhancer();
```

```
        // 设置类加载器
        enhancer.setClassLoader(clazz.getClassLoader());
        // 设置被代理类
        enhancer.setSuperclass(clazz);
        // 设置方法拦截器
        enhancer.setCallback(new DebugMethodInterceptor());
        // 创建代理类
        return enhancer.create();
    }
}
4.实际使用
AliSmsService aliSmsService = (AliSmsService)
CglibProxyFactory.getProxy(AliSmsService.class);
aliSmsService.send("java");
运行上述代码之后，控制台打印出：
before method send
send message:java
after method send
```

静态代理和动态代理的区别

\1. 灵活性： 动态代理更加灵活，不需要必须实现接口，可以直接代理实现类，并且可以不需要针对每个目标类都创建一个代理类（也就是说，你也可以用这个Factory逻辑给其他你需要代理的目标类创建对象）。另外，静态代理中，接口一旦新增加方法，目标对象和代理对象都要进行修改，这是非常麻烦的！

\2. JVM 层面： 静态代理在编译时就将接口、实现类、代理类这些都变成了一个个实际的 class 文件。而动态代理是在运行时动态生成类字节码，并加载到 JVM 中的。

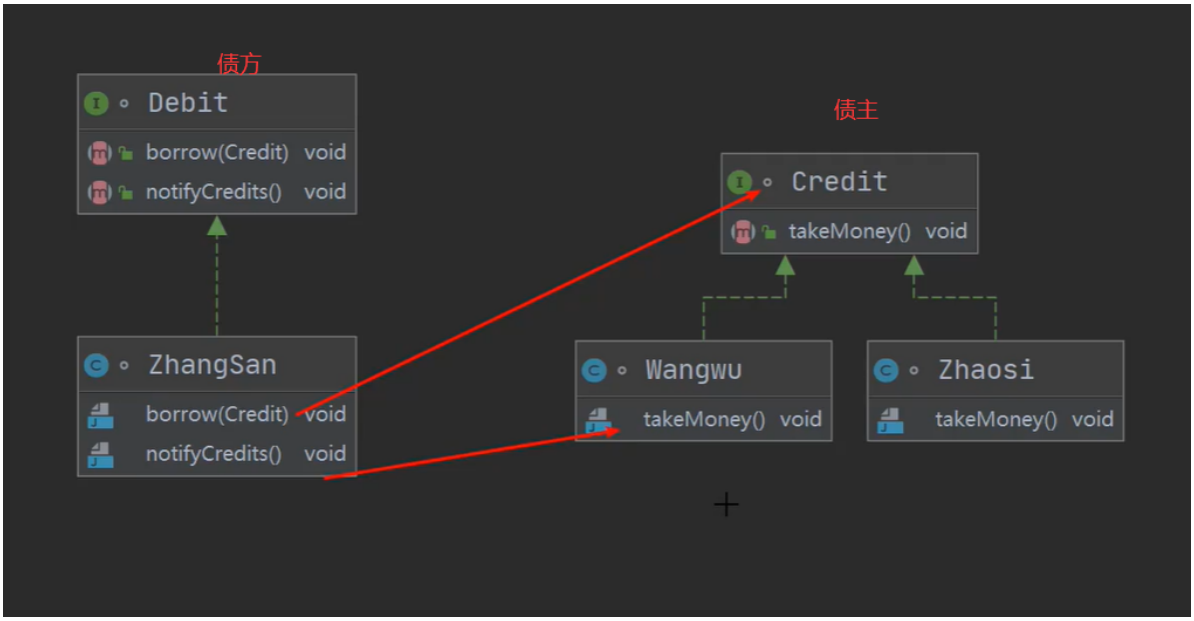
静态代理是在编译时就已经创建好代理类，这个代理类与被代理类实现了相同的接口，代理类中的方法实现调用了被代理类中的相应方法，并在调用前后进行了一些额外的操作。静态代理的好处在于编写简单，但是缺点是只能代理一个固定的接口或类，如果需要代理的类或接口过多，代理类的数量会增多，代码会变得冗余。

动态代理是在运行时通过反射等机制生成代理类，代理类的生成是动态的，可以在代理类中增加通用的逻辑，比如日志记录、性能统计等。可以代理任意接口或类，不需要为每个被代理类都创建一个代理类。在动态代理中，代理类不需要实现被代理类的接口，而是通过实现 InvocationHandler 接口，重写 invoke() 方法，实现代理逻辑。动态代理的好处在于可以提高代码复用率和灵活性，但是缺点在于性能较静态代理差，因为需要在运行时动态生成代理类和调用代理方法。

观察者模式

也称之为发布-订阅模式

可以想象成张三去找王五和李四借钱，那王五和李四就是观察者，看张三是否有钱没有，问张三要钱



说一说观察者模式

观察者模式主要用于处理对象间的一对多的关系，是一种对象行为模式。

该模式的实际应用场景比较容易确认，当一个对象状态发生变化时，所有该对象的关注者均能收到状态变化通知，以进行相应的处理。

观察者模式的优缺点

优点：

\1. 被观察者和观察者之间是抽象耦合的；

\2. 耦合度较低，两者之间的关联仅仅在于消息的通知；

\3. 被观察者无需关心他的观察者；

\4. 支持广播通信；

缺点：

\1. 观察者只知道被观察对象发生了变化，但不知变化的过程和缘由；

\2. 观察者同时也可能是被观察者，消息传递的链路可能会过长，完成所有通知花费时间较多；

\3. 如果观察者和被观察者之间产生循环依赖，或者消息传递链路形成闭环，会导致无限循环；

你的项目是怎么用的观察者模式?

在支付场景下，用户购买一件商品，当支付成功之后三方会回调自身，在这个时候系统可能会有很多需要执行的逻辑（如：更新订单状态，发送邮件通知，赠送礼品...），这些逻辑之间并没有强耦合，因此天然适合使用观察者模式去实现这些功能，当有更多的操作时，只需要添加新的观察者就能实现，完美实现了对修改关闭，对扩展开放的开闭原则。

在这里，第三方系统可以看作是观察者，因为它需要观察支付成功这一事件的发生，并且在事件发生后执行一些与该事件相关的操作。另外，可以将支付系统视为主题或被观察者，因为它会通知观察者支付成功这一事件的发生。（对于为什么系统可以称之为观察者和被观察者呢，可以想象把用户和系统的交互想成对话的形式，有说话者也有倾听者，一个说完话就变成了倾听者）

装饰器模式

什么是装饰器模式

其实继承本质也是装饰器模式，因为在原有的类上进行了包裹和封装

装饰器模式主要对现有的类对象进行包裹和封装，以期望在不改变类对象及其类定义的情况下，为对象添加额外功能。是一种对象结构型模式。需要注意的是，该过程是通过调用被包裹之后的对象完成功能添加的，而不是直接修改现有对象的行为，相当于增加了中间层。

```
interface Robot{
    void doSomething(;
}
class FirstRobot implements Robot{
    aoverride
    public void doSomething( {
        system.out.println("对话");
        System.out.printin("唱歌");
    }
}
abstract class RobotDecorator implements Robot{
    private Robot robot;
    public RobotDecorator(Robot robot){
        this.robot = robot;
    }
    @Override
    public void doSomething() {
        robot.doSomething();
        public void doHoreThing(){
            robot.doSomething() ;
            system.out.println("跳舞、拖地");
        }
    }
}
```

在使用的时候

```
//看着是不是很像输入输出流那个写法，JDK里面也大量使用了装饰器模式
new RobotDecorator(new FirstRobot()).doSomething();
```

讲讲装饰器模式的应用场景

- 如果希望在不修改代码的情况下使用该对象，且在运行的时候为该对象新增额外的功能，可以用这个模式
- 如果继承来扩展对象的行为不可行或者难以实现，可以用这个模式。
- 有的类使用了final禁止继承，来限制对某个类的进一步扩展。扩展这个类的唯一方法是用装饰器模式：用封装器对其进行封装。

装饰器模式和代理模式有什么异同

它们的主要目的是在不修改原有代码的情况下增强类的功能，但是它们的实现方式和使用场景有所不同。

异同点如下：

相同点：

- 装饰器模式和代理模式都使用了组合的方式，在装饰器模式中，被装饰对象和装饰器都实现了同一个接口，代理模式中，代理类和被代理类也实现了同一个接口。
- 两种模式都能够动态地增强对象的功能。

不同点：

- 装饰器模式的目的是为了在不修改原有对象的前提下，动态地给对象增加新的行为。通常用于为对象增加一些辅助性的功能，如缓存、日志等。
- 代理模式的目的是为了控制对对象的访问。代理类可以代替被代理类进行一些处理，如权限验证、性能统计等。代理模式的使用场景比较广泛，常用于远程代理、虚拟代理、缓存代理等。
- 装饰器模式中，装饰器和被装饰对象实现了相同的接口，装饰器对被装饰对象进行了透明化处理，使用者不需要知道有装饰器的存在。而代理模式中，代理类实现了被代理类的接口，并持有一个被代理类的对象，使用者需要知道代理类的存在，并通过代理类访问被代理类。

责任链模式

什么是责任链模式

责任链模式（Chain of Responsibility Pattern）是一种行为型设计模式，用于将请求的发送者和接收者解耦，并使多个对象都有机会处理请求。其基本思想是：将多个处理对象串成一条链，请求沿着这条链传递，直到有一个对象处理它为止。它最原始的裸体结构： switch-case 语句。

讲讲责任链模式的应用场景

当程序需要使用不同方式处理不同种类请求， 而且请求类型和顺序预先未知时， 可以使用责任链模式。 该模式能将多个处理者连接成一条链。 接收到请求后， 它会 “询问” 每个处理者是否能够对其进行处理。 这样所有处理者都有机会来处理请求。

当必须按顺序执行多个处理者时（比如，一个订单的处理流程可能需要经过多个处理阶段，每个阶段都需要执行一些操作，例如检查订单、确认支付、发货、确认收货等等，这些处理阶段之间的顺序是固定的，必须按照一定的顺序逐个执行，才能完成订单的处理。）， 可以使用该模式。

责任链模式的优缺点

责任链模式的优点如下：

1. 降低耦合度：责任链模式将请求的发送者和接收者解耦，请求发送者无需知道具体的处理者是谁，只需要知道它们在责任链上的顺序即可。
2. 灵活性增加：责任链模式可以根据需要动态地增加或删除处理者，从而灵活地调整请求的处理流程。
3. 可扩展性：责任链模式可以通过继承和实现来扩展新的处理者，而无需修改已有的代码，从而使系统具有更好的扩展性和维护性。
4. 可维护性：责任链模式使代码的结构更清晰，易于理解和维护。

责任链模式的缺点如下：

1. 处理时间不确定：由于请求的处理是在责任链上依次传递的，因此处理时间不确定，可能会出现某个处理者处理时间过长，导致整个请求的处理时间过长。
2. 可能出现死循环：如果责任链中的某个处理者在处理请求时出现错误，并且不向后传递请求，那么请求可能会在责任链中一直循环下去，无法被处理。
3. 需要事先确定责任链的顺序：责任链模式需要事先确定处理者的顺序，如果顺序不当，可能会导致请求无法被处理。

策略模式

什么是策略模式?

看名字，策略模式的侧重点是策略，也就是怎么做这件事，比如我去洗衣服，有干洗和水系，这就是不同的策略。 策略模式（Strategy Pattern）属于对象的行为模式。其用意是针对一**组**算法，将每一个算法封装到具有共同接口的独立的类中，从而使得它们可以 相互替换。策略模式使得算法可以在不影响到客户端的情况下发生变化。 其主要目的是通过定义相似的算法（这个相似的算法，可以理解为，如何进行处理这个问题），替换 if else 语句写法，并且可以随时相互替换。

JDK也有用到策略模式，比如说ThreadPoolExecutor中，你可以指定拒绝策略

策略模式有什么好处?

定义了一系列封装了算法、行为的对象，他们可以相互替换

1. 提高代码复用性：策略模式将一组算法封装成一个独立的类，并使得这些算法可以互相替换，因此可以在不修改原有代码的情况下重复使用已有的算法。
2. 提高代码可维护性：由于策略模式将不同的算法封装成独立的类，因此可以很方便地增加、修改或删除算法，从而提高代码的可维护性。
3. 提高代码扩展性：由于策略模式将算法封装成独立的类，因此可以很方便地增加新的算法实现，从而提高代码的扩展性。
4. 提高代码灵活性：由于策略模式允许客户端动态地选择算法实现，因此可以很灵活地适应不同的业务需求，从而提高代码的灵活性。

Spring 使用了哪些设计模式?

- 工厂设计模式 : Spring 使用工厂模式通过 BeanFactory 、 ApplicationContext 创建 bean 对象。
- 代理设计模式 : Spring AOP 功能的实现。
- 适配器模式 :Spring AOP 的增强或通知(Advice)使用到了适配器模式、 spring MVC 中也是用到了适配器模式适配成为 Controller（也就是比如我们有一个OrderContrller，这个类可以不加@Controller注解，但是需要通过实现接口来成为一个控制器，这是两种成为控制器的办法） ，但是为什么是适配呢？ 因为他这个类本来是个普通类，需要实现这个接口才能被SpringMVC识别成为一个控制器，这个接口就相当于适配器的作用，将现有的类适配成符合 Spring MVC 框架要求的控制器。
- 单例设计模式 : Spring 中的 Bean 默认都是单例的（默认是懒加载）但是单例Bean默认是应用程序启动时就创建好了，保证它在整个应用程序的生命周期都是可用的，并且避免了在运行时创建它所需要的开销和避免了潜在的并发问题。
- 模板方法模式 : Spring 中 jdbcTemplate 、 hibernateTemplate 等以 Template 结尾的对数据库操作的类，它们就使用到了模板模式。
- 装饰器设计模式 : 我们的项目需要连接多个数据库，而且不同的客户在每次访问中根据需要会去访问不同的数据库。这种模式让我们可以根据客户的需求能够动态切换不同的数据源。
- 观察者模式: Spring 事件驱动模型（也就是监听..）就是观察者模式很经典的一个应用。

JDK 使用了哪些设计模式?

在软件工程中，设计模式（design pattern）是对软件设计中普遍存在（反复出现）的各种问题，所提出的解决方案。以下是整理的几个在 JDK 库中常用的几个设计模式。

桥接模式

这个模式将抽象和抽象操作的实现进行了解耦，这样使得抽象和实现可以独立地变化。

GOF 在提出桥梁模式的时候指出，桥梁模式的用意是"将抽象化(Abstraction)与实现化(Implementation)脱耦，使得二者可以独立地变化"。这句话有三个关键词，也就是抽象化、实现化和脱耦。

在 Java 应用中，对于桥接模式有一个非常典型的例子，就是应用程序使用JDBC 驱动程序进行开发的方式。所谓驱动程序，指的是按照预先约定好的接口来操作计算机系统或者是外围设备的程序。

适配器模式

用来把一个接口转化成另一个接口。使得原本由于接口不兼容而不能一起工作的那些类可以在一起工作。

组合模式

又叫做部分-整体模式，使得客户端看来单个对象和对象的组合是同等的。

换句话说，某个类型的方法同时也接受自身类型作为参数。

java.util.Arrays#asList()

java.io.InputStreamReader(InputStream)

java.io.OutputStreamWriter(OutputStream)

java.util.Map#putAll(Map)

java.util.List#addAll(Collection)

java.util.Set#addAll(Collection)

装饰者模式

动态的给一个对象附加额外的功能，这也是子类的一种替代方式。可以看到，在创建一个类型的时候，同时也传入同一类型的对象。这在 JDK 里随处可见，你会发现它无处不在，所以下面这个列表只是一小部分。

享元模式

使用缓存来加速大量小对象的访问时间。

代理模式

代理模式是用一个简单的对象来代替一个复杂的或者创建耗时的对象。

java.io.BufferedInputStream(InputStream)

java.io.DataInputStream(InputStream)

java.io.BufferedOutputStream(OutputStream)

java.util.zip.ZipOutputStream(OutputStream)

java.util.Collections#checkedList|Map|Set|SortedSet|Sort

edMap

java.lang.Integer#valueOf(int)

java.lang.Boolean#valueOf(boolean)

java.lang.Byte#valueOf(byte)

java.lang.Character#valueOf(char)

抽象工厂模式

抽象工厂模式提供了一个协议来生成一系列的相关或者独立的对象，而不用指定具体对象的类型。它使得应用程序能够和使用的框架的具体实现进行解耦。这在 JDK 或者许多框架比如 Spring 中都随处可见。它们也很容易识别，一个创建新对象的方法，返回的却是接口或者抽象类的，就是抽象工厂模式了。

建造者模式

定义了一个新的类来构建另一个类的实例，以简化复杂对象的创建。建造

模式通常也使用方法链接来实现。

java.lang.reflect.Proxy

RMI

java.util.Calendar#getInstance()

java.util.Arrays#asList()

java.util.ResourceBundle#getBundle()

java.sql.DriverManager#getConnection()

java.sql.Connection#createStatement()

java.sql.Statement#executeQuery()

java.text.NumberFormat#getInstance()

javax.xml.transform.TransformerFactory#newInstance()

工厂方法就是一个返回具体对象的方法。

原型模式

使得类的实例能够生成自身的拷贝。如果创建一个对象的实例非常复杂且耗时，就可以使用这种模式，而不重新创建一个新的实例，你可以拷贝一个对象并直接修改它。

java.lang.StringBuilder#append()

java.lang.StringBuffer#append()

java.sql.PreparedStatement

javax.swing.GroupLayout.Group#addComponent()

java.lang.Proxy#newProxyInstance()

java.lang.Object#toString()

java.lang.Class#newInstance()

java.lang.reflect.Array#newInstance()

java.lang.reflect.Constructor#newInstance()

java.lang.Boolean#valueOf(String)

java.lang.Class#forName()

单例模式

用来确保类只有一个实例。Joshua Bloch 在 Effetive Java 中建议到，还有一种方法就是使用枚举。

责任链模式

通过把请求从一个对象传递到链条中下一个对象的方式，直到请求被处理完毕，以实现对象间的解耦。

命令模式

将操作封装到对象内，以便存储，传递和返回。

java.lang.Object#clone()

java.lang.Cloneable

java.lang.Runtime#getRuntime()

java.awt.Toolkit#getDefaultToolkit()

java.awt.GraphicsEnvironment#getLocalGraphicsEnvironment()

java.awt.Desktop#getDesktop()

java.util.logging.Logger#log()

javax.servlet.Filter#doFilter()

解释器模式

这个模式通常定义了一个语言的语法，然后解析相应语法的语句。

迭代器模式

提供一个一致的方法来顺序访问集合中的对象，这个方法与底层的集合的具体实现无关。

中介者模式

通过使用一个中间对象来进行消息分发以及减少类之间的直接依赖。

java.lang.Runnable

javax.swing.Action

java.util.Pattern

java.text.Normalizer

java.text.Format

java.util.Iterator

java.util.Enumeration

备忘录模式

生成对象状态的一个快照，以便对象可以恢复原始状态而不用暴露自身的内容。 Date 对象通过自身内部的一个 long 值来实现备忘录模式。

空对象模式

这个模式通过一个无意义的对象来代替没有对象这个状态。它使得你不用额外对空对象进行处理。

观察者模式

它使得一个对象可以灵活的将消息发送给感兴趣的对象。

java.util.Timer

java.util.concurrent.Executor#execute()

java.util.concurrent.ExecutorService#submit()

java.lang.reflect.Method#invoke()

java.util.Date

java.io.Serializable

java.util.Collections#emptyList()

java.util.Collections#emptyMap()

java.util.Collections#emptySet()

状态模式

通过改变对象内部的状态，使得你可以在运行时动态改变一个对象的行为。

策略模式

使用这个模式来将一组算法封装成一系列对象。通过传递这些对象可以灵

活的改变程序的功能。

模板方法模式

java.util.EventListener

javax.servlet.http.HttpSessionBindingListener

javax.servlet.http.HttpSessionAttributeListener

javax.faces.event.PhaseListener

java.util.Iterator

javax.faces.lifecycle.LifeCycle#execute()

java.util.Comparator#compare()

javax.servlet.http.HttpServlet

javax.servlet.Filter#doFilter()

让子类可以重写方法的一部分，而不是整个重写，你可以控制子类需要重写那些操作。

访问者模式

提供一个方便的可维护的方式来操作一组对象。它使得你在不改变操作的对象前提下，可以修改或者扩展对象的行为。