

Java高级工程师

面向对象终篇

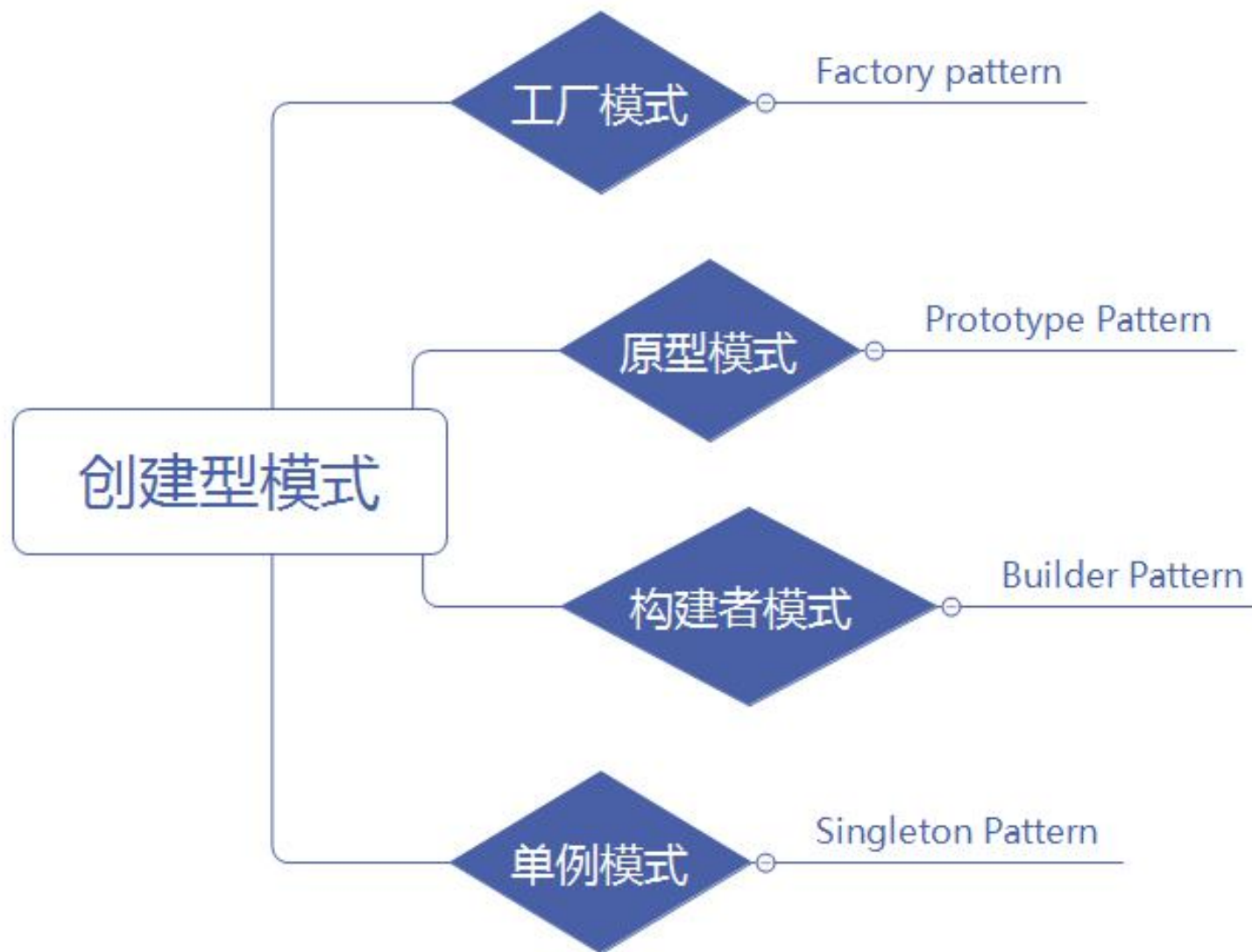
前序回顾

设计模式代表了软件设计的最佳实践

有哪些设计模式

设计模式按使用场景划分

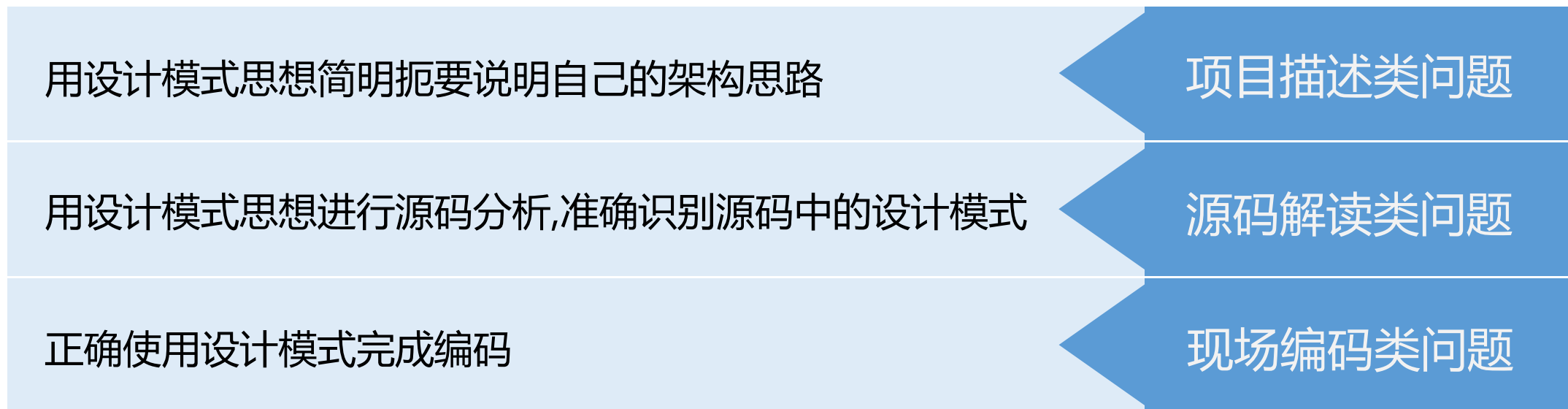
三大类







主要体现在以下过程:



题目:你都知道哪些设计模式

频度:中

难度:低

通过率:高

Note:回答8个以上就算通过

题目:请你描述一下你做xx项目的过程

频度:高

难度:低

通过率:低

背景<30%,实战60%,反思/复盘10%

解题思路

这个项目的需求背景是建设一个线上商铺的卡券系统 (发红包)...

经过系统分析,我发现卡券的形式很多,但基本功能是一致的,因此我选择用抽象工厂模式的思想来做系统架构...

采用spring可以很好的管理这些卡券对象

现在反思,上架卡券的流程当时因为时间太紧固化太严重,如果现在让我重新做一次,我会在里面引入责任链模式实现...,这样我的架构就可以扩展,还降低了很多的迭代成本

01

背景介绍

这个项目的需求背景是建设一个线上商铺的卡券系统 (发红包)...

02

设计模式识别、引出架构

经过系统分析,我发现卡券的形式很多,但基本功能是一致的,因此我选择用抽象工厂模式的思想来做系统架构...

03

主流框架选择

采用spring可以很好的管理这些卡券对象

04

反思加复盘

现在反思,上架卡券的流程当时因为时间太紧固化太严重,如果现在让我重新做一次,我会在里面引入责任链模式实现... ,这样我的架构就可以扩展,还降低了很多的迭代成本

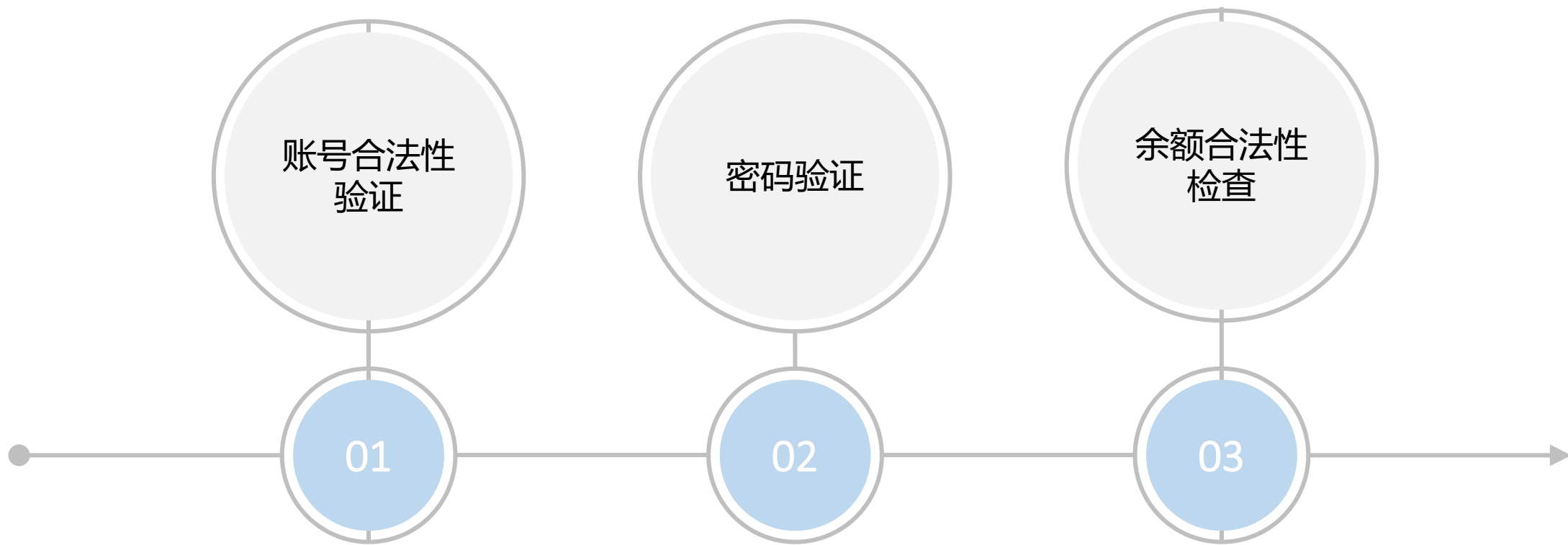
什么是责任链模式

很多对象由每一个对象对其下家的引用而连接起来形成一条链。请求在这个链上传递，直到链上的某一个对象决定处理此请求。

从面试题引入的知识点

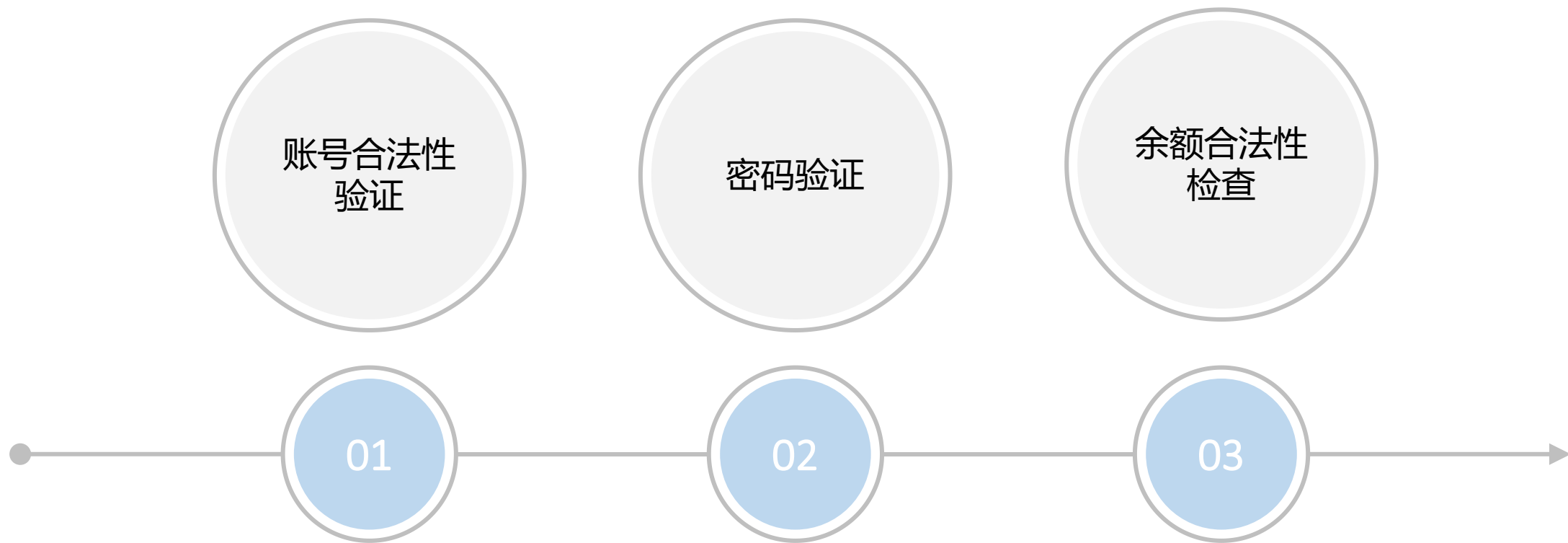
什么是责任链模式

想象你到银行ATM机上取款,银行的后台做了什么?



这 就 是 一 条 责 任 链

- 想象你到银行ATM机上取款,银行的后台做了什么?



这就是一条责任链，少验一个都不行

- 自顶向下,先做个壳

```
public class Main{

    public static void main(String[] args){
        // 实例化一个 Bank 对象
        Bank bank = new Bank();

        // 实例化标准请求对象
        BankRequest request = new BankRequest();
        BankResponse response = new BankResponse();

        // 赋值、调用
        request.setAccountNumber("ABCDEF");
        request.setAccountPassword("ABCDEF");
    }
}
```

- 把几个校验类写了——记住先抽接口把架构收个口

```
public interface AcctCheck{  
  
    // 通用 check 方法  
    void check(BankRequest request, BankResponse response);  
  
}
```

- 逐个实现

```
public class AccNoCheck implements AcctCheck{  
    @Override  
    public void check(BankRequest request, BankResponse response){  
        System.out.println("账户合法性检查");  
    }  
}
```

```
public class PasswdCheck implements AcctCheck{  
    @Override  
    public void check(BankRequest request, BankResponse response){  
        System.out.println("密码匹配检查");  
    }  
}
```

```
public class AvailCheck implements AcctCheck{  
    @Override  
    public void check(BankRequest request, BankResponse response){  
        System.out.println("可用余额检查");  
    }  
}
```

- 把逻辑实现了

```
public class Bank{  
    public void fetchCash(BankRequest request, BankResponse response){  
  
        // 把检查对象通通实例化  
        AccNoCheck acctNoCheck = new AccNoCheck();  
        AvailCheck availCheck = new AvailCheck();  
        PasswdCheck passwdCheck = new PasswdCheck();  
  
        // 顺序调一次  
        acctNoCheck.check(request, response);  
        availCheck.check(request, response);  
        passwdCheck.check(request, response);  
    }  
}
```

- 调用一下看看

```
/Library/Java/JavaVirtualMachines/jdk1.8.0_201.jdk/Contents/Home/bin/java ...
```

账号合法性检查

可用余额检查

密码匹配检查

```
Process finished with exit code 0|
```

达到预期

然而:好像有哪里不对...


```
public class Bank{  
    public void fetchCash(BankRequest request, BankResponse response){  
  
        // 把检查对象通通实例化  
        AccNoCheck acctNoCheck = new AccNoCheck();  
        AvailCheck availCheck = new AvailCheck();  
        PasswdCheck passwdCheck = new PasswdCheck();  
  
        // 顺序调一次  
        acctNoCheck.check(request, response);  
        availCheck.check(request, response);  
        passwdCheck.check(request, response);  
    }  
}
```

一个取现的方法为什么要有那么多的检查逻辑?

```
public class Bank{  
    public void fetchCash(BankRequest request, BankResponse response){  
  
        // 把检查对象通通实例化  
        AccNoCheck acctNoCheck = new AcctNoCheck();  
        AvailCheck availCheck = new AvailCheck();  
        PasswdCheck passwdCheck = new PasswdCheck();  
  
        // 顺序调一次  
        acctNoCheck.check(request, response);  
        availCheck.check(request, response);  
        passwdCheck.check(request, response);  
    }  
}
```

如果有一天要增加一个检查项,这里是不是要修改了?

接口抽象

可用调用通用的底层能力了

```
public interface AcctCheck{  
  
    // 通用 check 方法  
    void check(BankRequest request, BankResponse response);  
  
    // 增加一个方法, 得到 next object  
    AcctCheck nextCheck();  
  
}
```

```
public class CheckHead implements AcctCheck{

    private AcctCheck nextObj;

    @Override
    public void check(BankRequest request, BankResponse response){
        nextObj = null;
    }

    @Override
    public AcctCheck nextCheck(){
        return nextObj;
    }

    // setter and getter

}
```

同样的,将其他的check类也改成一样的形式

思考:怎么串成链?

- Head对象指定第一个对象

```
@Override  
public AcctCheck nextCheck() { return new AcctNoCheck(); }
```

- 后面的check对象同理

```
@Override  
public AcctCheck nextCheck() { return new PassCheck(); }
```


- 最后一个对象null封口

```
@Override  
public AcctCheck nextCheck() { return null; }
```

```
public class Bank{  
    public void fetchCash(BankRequest request, BankResponse response){  
        // 把链头实例化  
        CheckHead checkHead = new CheckHead();  
  
        // 遍历链表  
        AcctCheck currCheck = checkHead;  
        while((currCheck = currCheck.nextCheck) != null){  
            currCheck.check(request, response);  
        }  
    }  
}
```

- 一个接口
- 一条链表
- 一个迭代器——平等调用

```
public class GoodsDomain{  
    private GoodLocation location;  
    private SellingPrice sellingPrice;  
    private FeedbackStrategy feedbackStrategy;  
}
```

```
public class AssembleGoodLocation{
    public GoodLocation buildGoodLocation(){
        System.out.println("Build GoodsLocation");
        return new GoodLocation();
    }
}

public class AssembleSellingPrice{
    public SellingPrice buildSellingPrice(){
        System.out.println("Build SellingPrice");
        return new SellingPrice();
    }
}

public class AssembleFeedbackStrategy{

    public FeedbackStratege buildFeedbackStrategy(){
        System.out.println("Build FeedbackStrategy");
        return new FeedbackStrategy();
    }

}
```

- 商品的上架是一个领域模型(简单的说,就是对象的转换过程),我们用责任链模式实现它

```
public class Shop{

    private String name;
    private String privity;
    private String city;
    private String street;

    private ShopDao shopDao;

    public Shop modify(Shop newShop){
        return shopDao.update(newShop);
    }

    public void delete(String shopId){
        shopDao.delete(shopId);
    }

    public Goods upload(GoodsRequest request){
        return coverRequestToGoods(request);
    }

    private Goods coverRequestToGoods(GoodsRequest request){
        return new Goods();
    }

}
```

- 商品的上架是一个领域模型(简单的说,就是对象的转换过程),我们用责任链模式实现它

```
public class Shop{  
  
    private String name;  
    private String privity;  
    private String city;  
    private String street;  
  
    private ShopDao shopDao;  
  
    public Shop modify(Shop newShop){  
        return shopDao.update(newShop);  
    }  
  
    public void delete(String shopId){  
        shopDao.delete(shopId);  
    }  
  
    public Goods upload(GoodsRequest request){  
        return coverRequestToGoods(request);  
    }  
  
    private Goods coverRequestToGoods(GoodsRequest request){  
        return new Goods();  
    }  
}
```

电商平台内部针对商品的领域对象非常复杂,我们在这里实现 GoodLocation, SellingPrice, FeedbackStrategy 对象,当然,这些对象都作为 GoodsDomain 中的属性存在

```
public interface Assemble{  
    void assembleItem(GoodsDomain domain);  
}
```

```
public class AssembleFeedbackStrategy implements Assemble{  
  
    public FeedbackStrategy buildFeedbackStrategy(){  
        System.out.println("Build FeedbackStrategy");  
        return new FeedbackStrategy();  
    }  
  
    @Override  
    public void assembleItem(GoodsDomain domain){  
        domain.setFeedBackStrategy(buildFeedbackStrategy())  
    }  
  
}
```



```
public interface Assemble{  
    void assembleItem(GoodsDomain domain);  
}
```

每个组装器分别类似这样改造一下

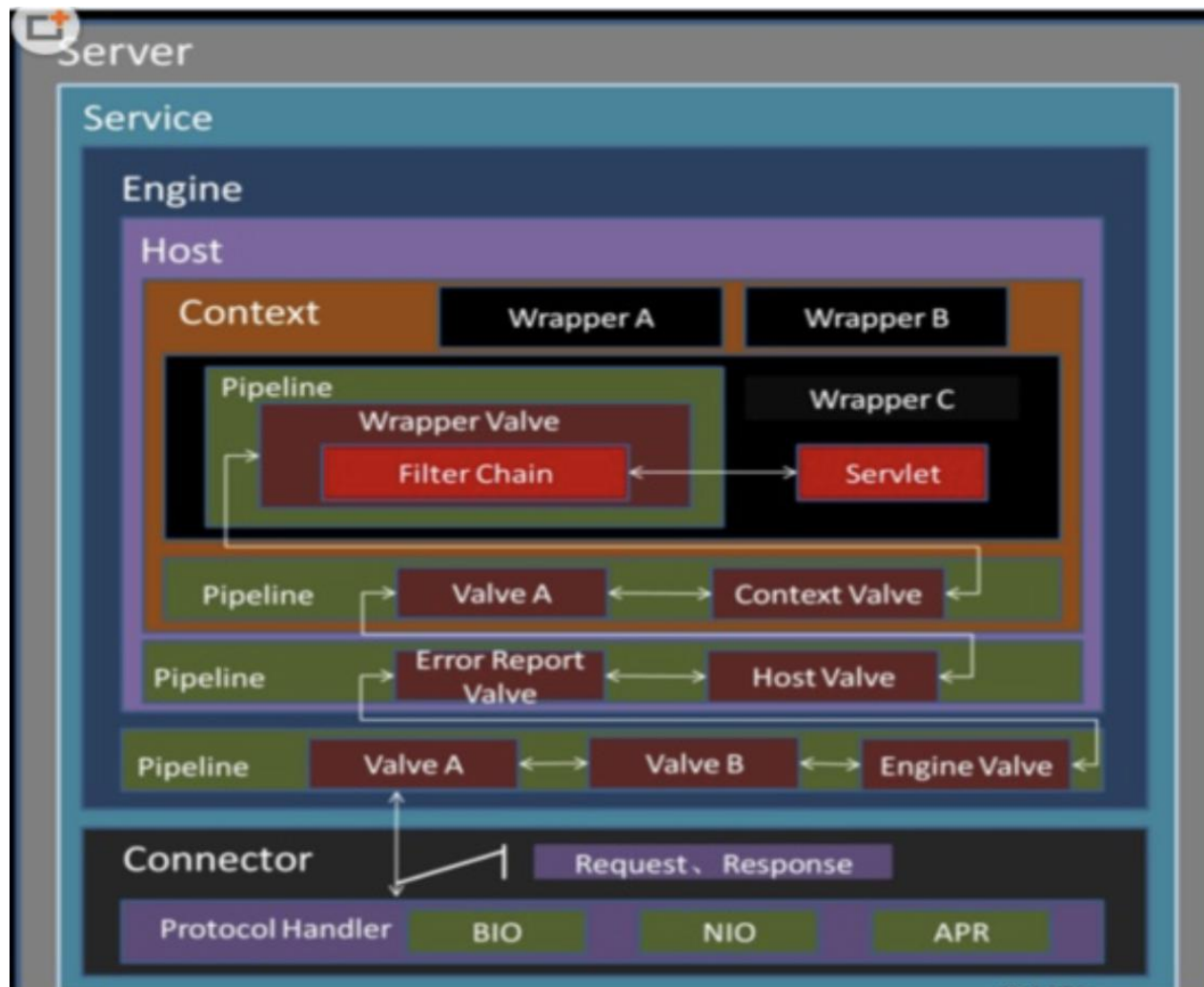
```
public class AssembleFeedbackStrategy implements Assemble{  
  
    public FeedbackStrategy buildFeedbackStrategy(){  
        System.out.println("Build FeedbackStrategy");  
        return new FeedbackStrategy();  
    }  
  
    @Override  
    public void assembleItem(GoodsDomain domain){  
        domain.setFeedBackStrategy(buildFeedbackStrategy())  
    }  
  
}
```

- 责任链模式在面试题中属于高级问题,也是在工程中改善我们架构的重要手段
- 问题: 你还知道其他使用责任链模式的场景吗?
- 频度: 高
- 难度: 中
- 通过率: 中

- 答案:
- 在其他场景中,最经典的使用之一就是在tomcat的管道(pipeline)中
- Tips: 什么是tomcat?
- tomcat 服务器是一个免费的开放源代码的Web 应用服务器
-
- java web应用,jsp,都可以跑在tomcat上,是工程实践中运用最广泛的 web应用服务器
-

- Tips:什么是tomcat的管道?
- 请求开始调用容器, 从容器的一系列处理到调用对应的Servlet之间, 会调用管道机制处理一系列的请求。其实类似于如下图所示:





Tomcat PipeLine中的基础阀门

- StandardEngineValve
- StandardHostValve
- StandardContextValve
- StandardWrapperValve

StandardEngineValve

调用时它会获取请求对应的主机host对象

StandardHostValve

获取请求对应的上下文context对象

StandardContextValve

上下文基础阀门先会判断是否访问了禁止目录WEB-INF或META-INF，接着获取请求对应的wrapper对象，再向客户端发送通知报文“HTTP/1.1 100 Continue”，最后调用wrapper对象中管道的第一个阀门

StandardWrapperValve

包装器基础阀门负责统计请求次数、统计处理时间、分配Servlet内存、执行servlet过滤器、调用Servlet的service方法、释放Servlet内存。

关键点

所有的Valve都是使用责任链模式进行组织的

StandardEngineValve 继承于 ValveBase

```
final class StandardEngineValve extends ValveBase {  
    private static final StringManager sm = StringManager.getManager("org.apache.catalina.core");  
  
    public StandardEngineValve() {  
        super( asyncSupported: true);  
    }  
}
```

ValveBase实现Valve接口

- 符合之前说过的特征:一个接口

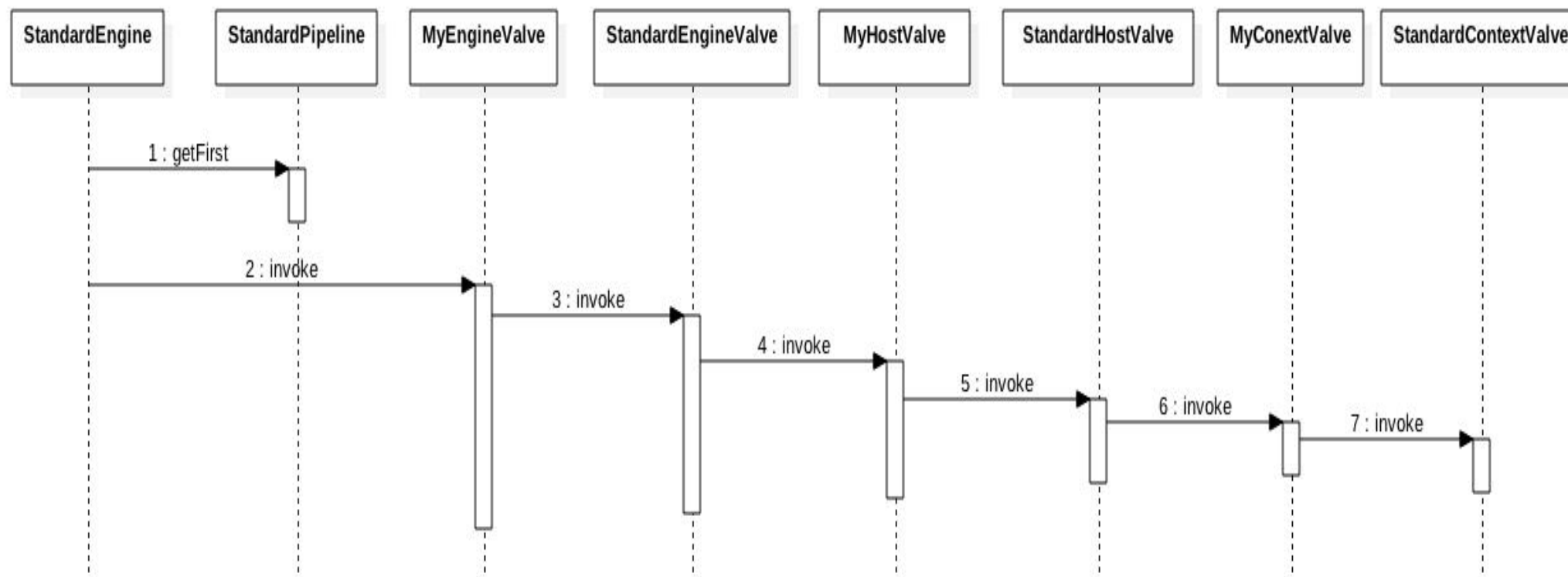
```
public abstract class ValveBase extends LifecycleMBeanBase implements Contained, Valve {  
    protected static final StringManager sm = StringManager.getManager(ValveBase.class);  
    protected boolean asyncSupported;  
    protected Container container;  
    protected Log containerLog;  
    protected Valve next;
```

链表结构出来了

```
public interface Valve {  
    /**  
     * @return the next Valve in the pipeline containing this Valve, if any.  
     */  
    Valve getNext();  
  
    /**  
     * Set the next Valve in the pipeline containing this Valve.  
     * @param valve The new next valve, or <code>null</code> if none  
     */  
}
```

```
/**
 * Start {@link Valve}s in this pipeline and implement the requirements
 * of {@link LifecycleBase#startInternal()}.
 *
 * @exception LifecycleException if this component detects a fatal error
 * that prevents this component from being used
 */
@Override
protected synchronized void startInternal() throws LifecycleException{
    //Start the Valves in our pipeline (including the basic), if any
    Valve current = first;
    if (current == null) {
        current = basic;
    }
    while (current != null){
        if (current instanceof Lifecycle)
            ((Lifecycle) current).start();
        current = current.getNext();
    }
    setState(LifecycleState. STARTING) ;
}
```

熟悉的实现



扩展

除了tomcat, netty, mina也有相同的特征

责任链模式在不同平台间的领域对象转换/构造中也很适用

- 商品信息这个领域对象有三个属性:
- 产地
- 售价
- 营销策略

```
public class GoodsDomain{  
    private GoodLocation location;  
    private SellingPrice sellingPrice;  
    private FeedbackStrategy feedbackStrategy;  
}
```

构建领域对象是多少电商平台必备的工作

```
public interface Assemble{  
    void assembleItem(GoodsDomain domain);  
}
```

```
public class AssembleFeedbackStrategy implements Assemble{

    public FeedbackStrategy buildFeedbackStrategy(){
        System.out.println("Build FeedbackStrategy");
        return new FeedbackStrategy();
    }

    @Override
    public void assembleItem(GoodsDomain domain){
        domain.setFeedBackStrategy(buildFeedbackStrategy())
    }
}
```

```
public class AssembleGoodLocation implements Assemble{  
    public GoodLocation buildGoodLocation(){  
        System.out.println("Build GoodsLocation");  
        return new GoodLocation();  
    }  
  
    @Override  
    public void assembleItem(GoodsDomain domain){  
        domain.setFeedBackStrategy(buildGoodLocation())  
    }  
}
```

```
public class AssembleSellingPrice implements Assemble{  
    public SellingPrice buildSellingPrice(){  
        System.out.println("Build SellingPrice");  
        return new SellingPrice();  
    }  
  
    @Override  
    public void assembleItem(GoodsDomain domain){  
        domain.setFeedBackStrategy(buildSellingPrice())  
    }  
}
```

```
public interface AssembleHandle{  
    // 指向下一组构建器  
    Assemble getNext();  
    void addItem(Assemble assemble);  
    void invokePipeline(GoodsDomain domain);  
}
```



```
public class AssembleHandle implements AssembleHandle{  
    // 抽象构建器接口的定义  
    private Assemble firstAssemble;  
    private Assemble currAssemble;  
    private LinkedList<Assemble> pipeline = new LinkedList<>();  
}
```

```
/**
 * 指向下一组构建器
 */
@Override
public Assemble getNext(){
    // 防止初始化异常
    if (firstAssemble == null){
        return null;
    }
    // 初次运行
    if (currAssemble == null){
        currAssemble = firstAssemble;
        return firstAssemble;
    }
    // 最后一个元素
    if (pipeline.getLast().equals(currAssemble)){
        return null;
    }
    int index = pipeline.indexOf(currAssemble);

    this.currAssemble = pipeline.get(++index);
    return this.currAssemble;
}
```

```
@Override  
public void addItem(Assemble assemble){  
    pipeline.add(assemble);  
    if(firstAssemble == null){  
        firstAssemble = assemble;  
    }  
}
```

```
public static AssembleHandle getPipLineHandleInstance(){
    AssembleHandle handle = new AssembleHandleImpl();
    handle.addItem(new AssembleSellingPrice());
    handle.addItem(new AssembleGoodsLocation());
    handle.addItem(new AssembleFeedbackStrategy());

    handle.setFirstAssemble(handle.getFirstAssemble());
    handle.setCurrAssemble(null);
    return handle;
}
```

```
@Override
public void invokePipeLine(GoodsDomain domain){
    Assemble assemble = null;
    while((assemble = this.getNext()) != null){
        assemble.assembleItem(domain);
    }
}
```

开卷考

请手写一个单例模式的实现

频度:高

难度:低

通过率:低

单例模式很好实现

但是永远写不对,被刷也不知道为什么.

单例模式

顾名思义就是只创建一个实例

单例模式

如你的电脑上,鼠标就是一个单例(不会出现两个光标)

单例模式

银行里,利率就是一个单例

在我们之前的代码中实现一个单例模式

- 目标:用单例模式保证有且只有一个Bank对象在上下文中存在

```
public static void main(String[] args){  
    // 实例化一个 Bank 对象  
    Bank bank = new Bank();  
}
```

```
public static void main(String[] args){  
    // 初始化  
    Map<String, Bank> objMap = new HashMap<>();  
    Bank bank;  
  
    // 单例模式, 实例化一个 Bank 对象  
    if(!objMap.containsKey("bank")){  
        bank = new Bank();  
        // 将单例放入 map 中, 保证其他请求不会重复调用  
        objMap.put("bank", bank);  
    }else{  
        bank = objMap.get("bank");  
    }  
}
```

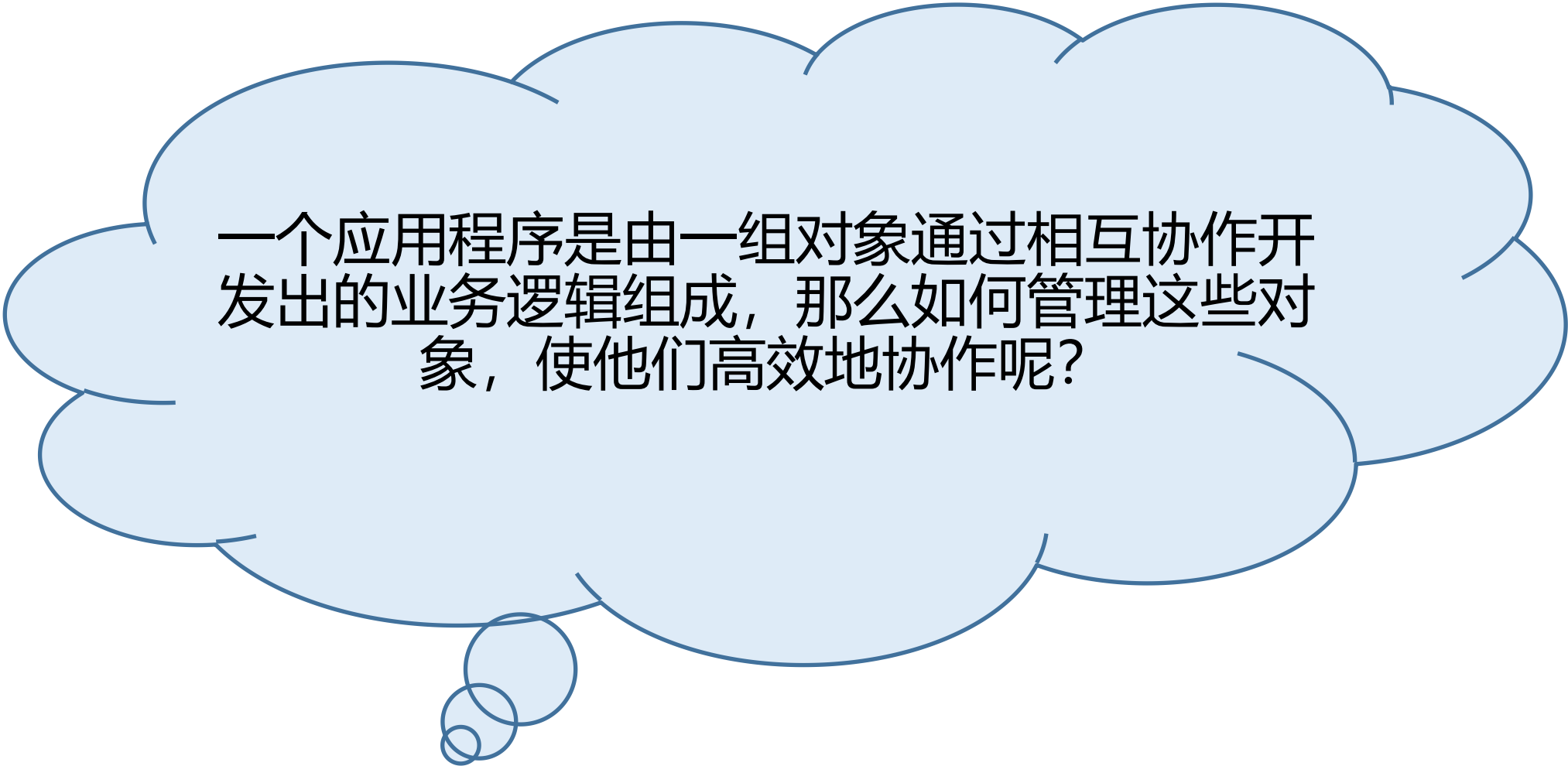
该答案基本不得分

两大缺陷

没有使用工厂模式
没有考虑线程安全问题

提到单例模式不得不关注到 spring

什么是spring?



一个应用程序是由一组对象通过相互协作开发出的业务逻辑组成，那么如何管理这些对象，使他们高效地协作呢？

spring来负责控制对象的生命周期和对象间的关系。所有的类都会在spring容器中登记，告诉spring你是个什么东西，你需要什么东西，然后spring会在系统运行到适当的时候，把你想要的东西主动给你，同时也把你交给其他需要你的东西。

- 引入spring的依赖，直接引入context即可，因为会将其依赖的所有包全部引入。

```
<dependency>  
    <groupId>org.springframework</groupId>  
    <artifactId>spring-context</artifactId>  
    <version>4.3.18.RELEASE</version>  
</dependency>
```

```
package com.ioc.demo1;

public interface UserService {
    public void sayHello();
}
```

```
package com.ioc.demo1;
public class UserServiceImpl implements UserService {
    public void sayHello() {
        System.out.println("Hello Spring");
    }
}
```

- 在resources中创建xml配置文件application-context.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">

    <!-- UserService的创建权交给spring -->
    <bean id="userService" class="com.ioc.demol.UserServiceImpl"></bean>

</beans>
```

- 在程序中读取spring的配置文件，通过spring框架获得bean，完成相应的操作

```
@Test
/**
 * spring的方式实现
 */
public void demo2() {
    // spring工厂
    ApplicationContext applicationContext = new ClassPathXmlApplicationContext("application-context.xml")
    // 通过工厂获取类对象
    UserService userService = (UserService) applicationContext.getBean("userService");
    userService.sayHello();
}
```

- 面试题:
- 你读过Spring的源码吗?
- Spring 创建的bean默认遵循什么设计模式?
- 请简单描述Spring创建bean的过程

三个问题其实是一个题

Spring原理和源码面试

难度:高

频度:高

通过率:低

答题指引

从 bean的创建开始,spring 说到底是个bean管理的框架

答题指引

从单例模式入手,这是Spring创建bean的默认方式

```
// Create bean instance.  
if (mbd.isSingleton()) {  
    sharedInstance = getSingleton(beanName, () -> {  
        try {  
            return createBean(beanName, mbd, args);  
        }  
        catch (BeansException ex) {  
            // Explicitly remove instance from singleton cache: It might have been put there  
            // eagerly by the creation process, to allow for circular reference resolution.  
            // Also remove any beans that received a temporary reference to the bean.  
            destroySingleton(beanName);  
            throw ex;  
        }  
    });  
    bean = getObjectForBeanInstance(sharedInstance, name, beanName, mbd);  
}
```

逻辑就埋在这个function里

```
synchronized (this.singletonObjects) {  
    Object singletonObject = this.singletonObjects.get(beanName);  
    if (singletonObject == null) {  
        if (this.singletonsCurrentlyInDestruction) {  
            throw new BeanCreationNotAllowedException(beanName,  
                "Singleton bean creation not allowed while singletons of this factory are in destruction " +  
                "(Do not request a bean from a BeanFactory in a destroy method implementation!)");  
        }  
        if (logger.isDebugEnabled()) {  
            logger.debug("Creating shared instance of singleton bean '" + beanName + "'");  
        }  
        beforeSingletonCreation(beanName);  
        boolean newSingleton = false;  
        boolean recordSuppressedExceptions = (this.suppressedExceptions == null);  
        if (recordSuppressedExceptions) {  
            this.suppressedExceptions = new LinkedHashSet<>();  
        }  
        try {  
            singletonObject = singletonFactory.getObject();  
            newSingleton = true;  
        }  
        catch (IllegalStateException ex) {
```

```
synchronized (this.singletonObjects) {  
    Object singletonObject = this.singletonObjects.get(beanName);  
    if (singletonObject == null) {  
        if (this.singletonsCurrentlyInDestruction) {  
            throw new BeanCreationNotAllowedException(beanName,  
                "Singleton bean creation not allowed while singletons of this factory are in destruction " +  
                "(Do not request a bean from a BeanFactory in a destroy method implementation!)");  
        }  
        if (logger.isDebugEnabled()) {  
            logger.debug("Creating shared instance of singleton bean '" + beanName + "'");  
        }  
        beforeSingletonCreation(beanName);  
        boolean newSingleton = false;  
        boolean recordSuppressedExceptions = (this.suppressedExceptions == null);  
        if (recordSuppressedExceptions) {  
            this.suppressedExceptions = new LinkedHashSet<>();  
        }  
        try {  
            singletonObject = singletonFactory.getObject();  
            newSingleton = true;  
        }  
        catch (IllegalStateException ex) {
```

先从一个注册列表中按**beanName**取

取不到说明上下文没有单例,走创建单例的逻辑

```
synchronized (this.singletonObjects) {  
    Object singletonObject = this.singletonObjects.get(beanName);  
    if (singletonObject == null) {  
        if (this.singletonObjectsCurrentlyInDestruction) {  
            throw new BeanCreationNotAllowedException(beanName,  
                "Singleton bean creation not allowed while singletons of this factory are in destruction " +  
                "(Do not request a bean from a BeanFactory in a destroy method implementation!)");  
        }  
        if (logger.isDebugEnabled()) {  
            logger.debug("Creating shared instance of singleton bean '" + beanName + "'");  
        }  
        beforeSingletonCreation(beanName);  
        boolean newSingleton = false;  
        boolean recordSuppressedExceptions = (this.suppressedExceptions == null);  
        if (recordSuppressedExceptions) {  
            this.suppressedExceptions = new LinkedHashSet<>();  
        }  
        try {  
            singletonObject = singletonFactory.getObject();  
            newSingleton = true;  
        }  
        catch (IllegalStateException ex) {
```

先从一个注册列表中按beanName
取
取不到说明上下文没有单例,走创建
单例的逻辑

这里就在创建单例bean

Spring中单例的实现总结

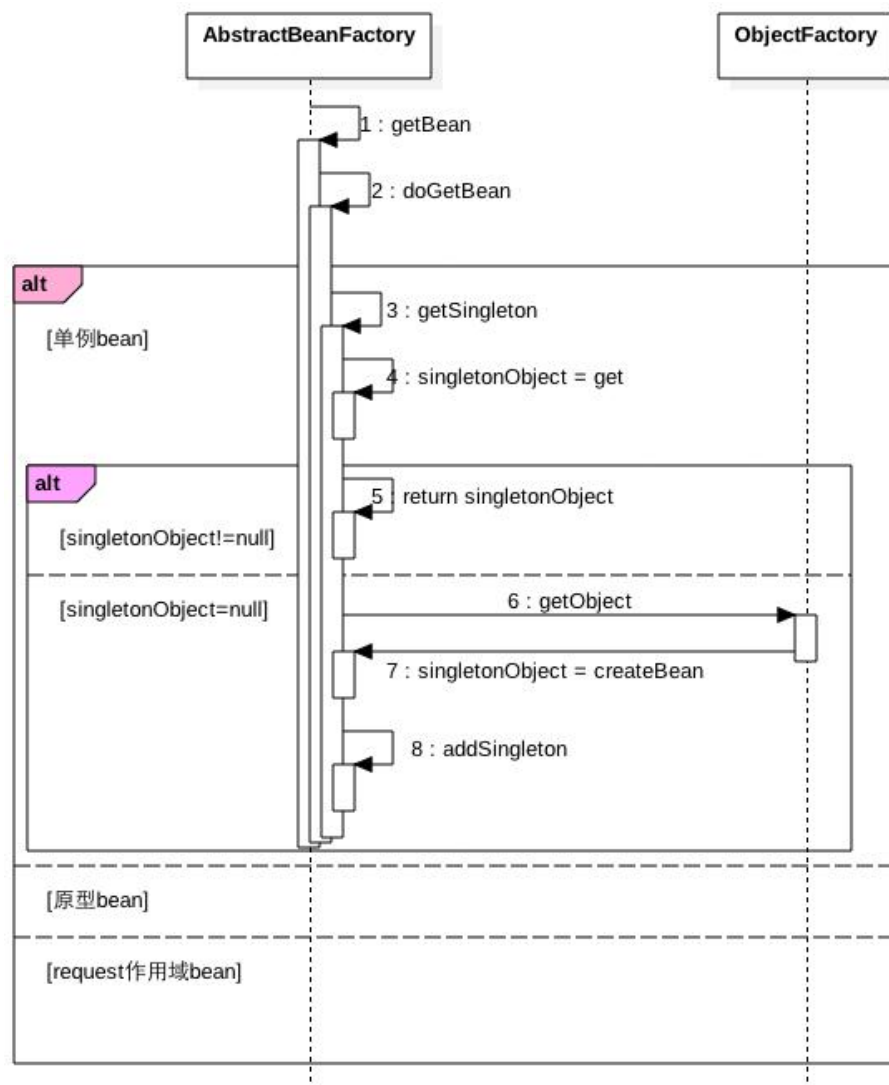
- 要有一个全局确认待实现的对象是否已经在作用域中存在的能力
- 源码实现:
 - */** Cache of singleton objects: bean name to bean instance. */*
private final Map<String, Object> singletonObjects = new ConcurrentHashMap<>(256);
 - synchronized (this.singletonObjects) {
 Object oldObject = this.singletonObjects.get(beanName);

单例的实现总结

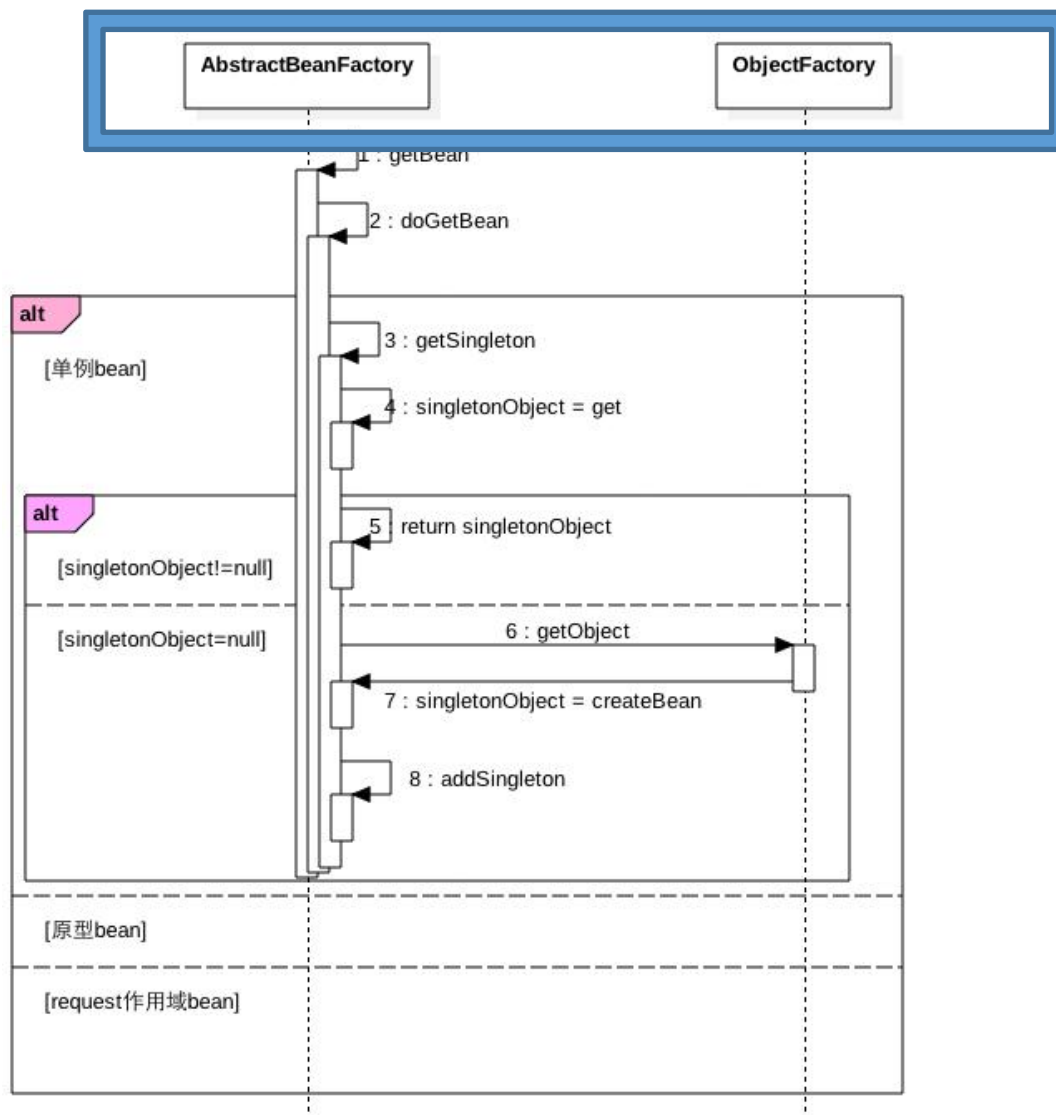
- 关键点:
- ConcurrentHashMap
- Synchronized
- Note:单例模式要注意运行背景,在多线程背景下必须采用线程安全的方式实现.
- 不少候选人吐槽:单例这么简单的设计模式为什么判我不对,一定有黑幕...
- 多读经典源码,才能明白自己认知的差距在哪里

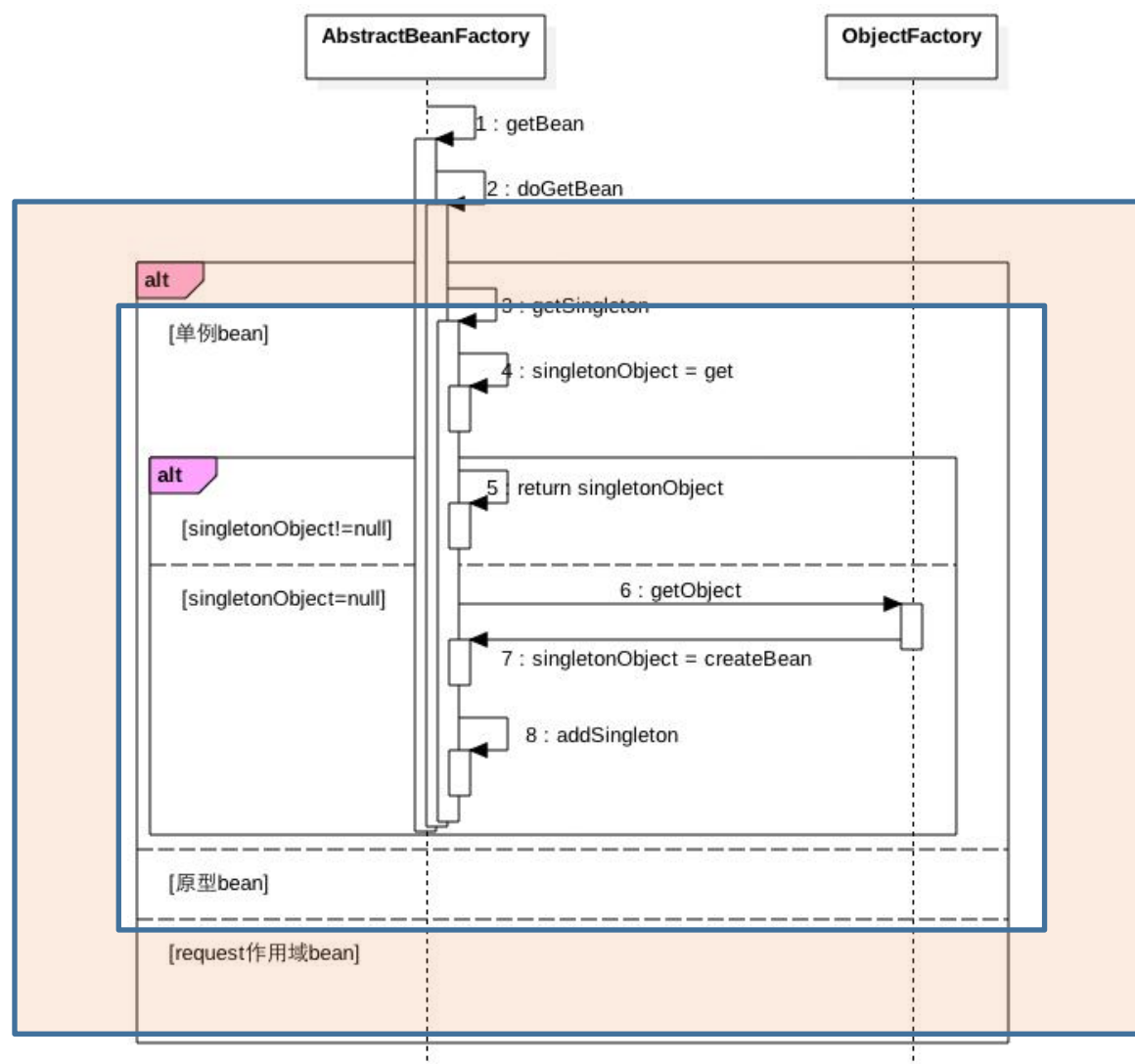
一句话总结

Spring内部是通过一个ConcurrentMap来管理单个bean的。获取bean时候会先看看singletonObjects中是否有，有则直接返回，没有则创建后放入。



工厂模式是基础





单例模式

单例模式对责任链模式也有很好的支持

加入工厂模式,先把工厂的能力做一个接口抽象

```
public interface IBankFactory{  
    Bank createSingletonFactory();  
}
```

```
public class BankFactory implements IBankFactory {
    // 静态化一个存储 Bank 示例的 map
    public static ConcurrentMap<String, Bank> objMap;

    // 一个静态方法，取工厂示例，降低复杂度
    public static BankFactory getDefaultFactory(){
        return new BankFactory();
    }

    @Override
    public Bank createSingletonFactory(){
        // 全局上下文，初始化
        if(objMap == null){
            objMap = new ConcurrentHashMap<String, Bank>();
        }
        if(objMap.containsKey("bank")){
            return objMap.get("bank");
        }else{
            return new Bank();
        }
    }
}
```

```
public class Main{
    public static void main(String[] args){
        // 单例工厂中取 Bank 对象
        Bank bank = BankFactory.getDefaultFactory.createSingletonFactory();
        // 实例化标准请求对象
        BankRequest request = new BankRequest();
        BankResponse response = new BankResponse();

        // 赋值、调用
        request.setAccountNumber("ABCDEFGH");
        request.setAccountPassword("123456");

        bank.fetchCash(request, response);
    }
}
```

总结

单例模式保证对象单例的逻辑适合让工厂模式来“吃掉”它
这样对提升架构清晰度有好处

课程总结

本次课程两个意图:

课程总结

一、领会设计模式在经典代码中的意义

课程总结

二、培养读源码的习惯和技巧:

从设计模式入手定位代码

分析设计模式特点定位代码间的关联关系

课程总结

三、深度理解源码,让源码面试类的话题有话说,有深度: