# 一、Spring概述

## spring是什么？

1. 轻量级的

零配置编程、API使用简单

1. 面向Bean

只需要编写非常普通的Bean

1. 松耦合

充分利用AOP思想

1. 王能胶

与主流框架无缝集成

1. 设计模式

将java中经典的设计模式运用得淋漓尽致

# 二、一切从Bean开始

## Spring设计伊始，为什么这么设计

Spring是为解决企业级应用开发的复杂性而设计，他可以做很多事。但归根结底支撑Spring的仅仅是少许的基本理念，而所有地这些的基本概念都能可以追溯到一个最根本的使命：简化开发。

而Spring则立志于全方面的简化java开发。对此，他主要采取了4个关键策略：

1. 基于POJO的轻量级的最小侵入性编程
2. 通过依赖注入和面向接口松耦合
3. 基于切面的惯性进行声明式编程
4. 通过切面和模板减少样板式代码

而他主要是通过：面向Bean、依赖注入以及面向切面这三种方式来达成的。

## 把一个代码公共使用有几种方法：

1. 导成jar包
2. 继承

## 面向Bean（控制反转）

Spring是面向Bean的编程（Bean Oriented Programming， BOP），Bean是Spring中才是真正的主角。Bean在Spring中作用就像Object对OOP的意义一样，Spring中没有Bean也就没有Spring存在的意义。Spring提供了IOC容器通过配置文件或者注解的方式来管理对象之间的依赖关系。

控制反转（其中最常见的方式叫做依赖注入（Dependency Injection， DI），还有一种方法叫“依赖查找”（Dependency Lookup， DL），他在C++、java、php以及.Net中都运用。在最早的Spring中是包含有依赖注入方法和依赖查询的，但因为依赖查询使用频率过低，不久就被Spring移除了，所以在Spring中控制反转也被称为依赖注入），他的基本概念是：不创建对象，但是描述创建他们的方式。在代码中不直接与对象和服务器连接，但在配置文件中描述哪一个组件哪一项服务。容器（在Spring框架中是IOC容器）负责将这些联系在一起。

在典型的IOC场景中，容器创建了所有对象，并设置必要的属性将他们链接在一起，决定什么时间调用方法。

## 依赖注入

Spring设计的核心org.springframework.beans包（架构核心是org.springframework.core包），它的设计目标是与javaBean组件一起使用。这个包通常不是由用户直接使用，而是服务器将其用作其他多数功能的底层中介。下一个最高级抽象是BeanFactory接口，它是工厂设计模式的实现，允许通过名称创建和检索对象。BeanFactory也可以管理对象之间的关系。

BeanFactory支持两个对象模型。

1. 单例：模型提供了具有特定名称的对象的共享实例，可以在查询是对其进行检索。Singleton是默认的也是最常用的对象模型。对于无状态服务对象很理想。
2. 原型：模型确保每次检索都会创建单独的对象。在每个用户都需要自己的对象时，原型模型最适合。

bean工厂的概念是Spring作为IOC容器的基础。IOC则将处理事情的责任从应用程序代码移动在框架。

实现依赖注入（赋值 把 = 右边的东西赋值给左边）：

@autowrite InterfaceA a; //自动把它的实现类注入进来

@Resource(“aaa”) A b; //IOC容器中类的id为aaa对象自动注入到这里（可以区分父子类的）

@autowrite A a; //根据类型自动注入

Spring的注入方式：

1. setter方法
2. 构造方法
3. 强制赋值

依赖链中的所有对象，都会在ioc容器里面初始化

## 面向切面

面向切面编程，即AOP，是一种编程思想，它允许程序员对横切关注点或横切典型的职责分界线的行为（例如日志和事物管理）进行模块化。AOP的核心构造是方面，它将那些影响多个类的行为封装到可重用的模块中。

AOP和IOC是补充性的技术，他们运行模块化方式解决企业应用程序开发中的复杂问题。在典型的面相对象开发方式中，可能要将日志记录语句放在所有方法和java类中才能实现日志功能。在AOP方式中，可以反过来将日志服务模块化，并以声明的方式将他们应用到需要日志的组件上。当然，优势就是java类不需要知道日志服务的存在，也不需要考虑相关的代码。所以，用Spring AOP编写的应用程序代码是松耦合的。

AOP的功能完全集成了Spring事物管理、日志和其他各种特性的上下文中。

AOP的核心思想（解耦）：先要把一个整体，给拆了，分别开发，等到发布的时候，在组装到一起运行。专人做专事

AOP关心的不是结果而是过程

Authentication权限认证

Logging日志

Transctions Manager 事物

lazy Loading 懒加载

Context Process 上下文处理

Error Handler 错误跟踪（异常捕获机制）

Cache 缓存

## Spring思想总结

### AOP

#### 应用场景

1. Aspect Oriented Programming（面向切面编程）
2. 找出多个类中有一定规律的代码，开发时拆开，运行时再合并。
3. 面向切面编程，即面向规则编程。

#### 一句话归纳

解耦，专人做专事

### OOP

#### 应用场景

1. Object Oriented Programming（面向对象编程）
2. 归纳总结生活中一起事务

#### 一句话归纳

封装、继承、多态

### BOP

#### 应用场景

1. Bean Oriented Programming（面向Bean编程）

2、面向Bean（普通的java类）设计程序

#### 一句话归纳

一切从Bean开始

### IOC

#### 应用场景

1. Inversion of Control（控制反转）
2. 将new对象的动作交给Spring管理，并由Spring保存已创建的对象（IOC容器）

#### 一句话归纳

转交控制权（即控制权反转），IOC值负责new

### DI/DL

#### 应用场景

1. Dependency Injection（依赖注入）/Dependency Lookup（依赖查找）
2. 依赖注入、依赖查询，Spring不仅保存自己创建的对象，而且保存对象与对象之间的关系
3. 注入即赋值，主要三种方式构造方法、set方法、直接赋值

#### 一句话归纳

先理清关系在赋值，DI负责理清关系和赋值

# 俯瞰Spring架构设计

## 系统框架

spring总共大约有20个模块，由1300多个不同文件构成。而这些组件被分别整合在核心容器（Core Container）、AOP（Aspect Oriented Programming）和设备支持（Instrmentation）、数据访问及集成（Data Access/Integeration）、Web、报文发送（Messaging）、Test，6个模块集合中。

**Web**

**Data Accrss/Integration**

**ORM**

**JDBC**

**Servlet**

**WebSocket**

**JMS**

**OXM**

**Portlet**

**Web**

**Transactions**

**AOP**

**Instrumentation**

**Messaging**

**Aspects**

**Core Container**

**Context**

**SpEL**

**Core**

**Beans**

**Test**

JDBC 使用的是模板模式

ORM（RowMapper）使用的是策略模式

Transactions（事务管理）使用的是代理模式

WebSocket（WS）http

AOP（config，切面编程，Cglib）使用的是代理模式

Beans（BOP）

Core核心实现逻辑

context IOC容器

SpEL EL表达式

做web编程的jsp现阶段被什么取代了：

FreeMark

Velocity

纯html + ajax

组成Spring框架的每个模块集合或者模块都可以单独存在，也可以一个或多个模块联合实现。每个模块的组成和功能如下：

1.核心容器：由spring-beans、spring-core、spring-context和spring-expression（Spring Expression Language， SpEl）4个模块组成

spring-beans和spring-core模块时

## 常用的设计模式

### 代理模式

1. 事情必须做，而自己又没有时间做或者不想做；
2. 持有被代理对象的引用。

静态代理、动态代理

JDK动态代理

CGLib动态代理

cglib.jar（全称Code Generation Library代码生产库）

asm.jar（全称assembly，装配）

jdk的动态代理是通过接口来进行强制转换的（即生成后的代理对象，可以强制转换为接口）

CGLib的动态代理是通过继承来实现的，通过生成一个被代理对象的子类，然后重写父类的方法（生成以后的对象，可以强制转换为被代理对象，也就是自己写的类。即把子类引用赋值给父类）。CGLib不用为其传递对象引用，因为在CGLib内部帮助实现了new一个被代理对象的子类。此子类重写了父类的所有方法，我们改变子类对象的某些属性，是可以间接的操作父类的属性的。

AOP的切面具体实现是由代理模式实现

理解：中介、媒人、经纪人

特点：

1. 执行者、被代理人
2. 对于被代理人来说，这件事情是一定要做的，但是自己又不想做或者没有时间做，找代理。
3. 需要获取到被代理的人个人资料。

代理模式关注的是过程

总结：动态代理做了一件字节码重组的操作，使用ClassLoader将新生成的字节码加载到JVM中。

动态代理可以做一件什么事情：可以在每一个方法调用之前加一些代码，在方法调用之后再加一些代码。

AOP：事务代理（声明式事务、哪个方法需要加事务，哪个方法不需要加事务）、

日志监听

service方法

开启一个事务（open）

事务的执行（是由我们自己的代码完成的）

监听到是否有异常，可能需要根据异常的类型来决定这个事务是否要回滚还是继续提交（commit/rollback）

事务关闭（close）

#### 代码实现代理模式：

##### jdk实现方法：

使用jdk的动态代理方法一定要实现一个接口，因为jdk的动态代理返回的是一个实现这个接口的类。

jdk代理模式，实现原理：

1.拿到被代理对象的引用，然后获取他的接口

2.jdk代理重新生产一个类，同时实现我们给的代理对象所实现的接口

3.把被代理对象的引用也拿到了

4.重新动态生产一个class字节码

5.然后编译

package com.proxy.jdk;

public interface Person {

void findLove();

public String getSex();

public void setSex(String sex);

public String getName();

public void setName(String name);

}

package com.proxy.jdk;

public class ZhangSan implements Person {

private String sex = "女";

private String name = "张三";

@Override

public void findLove() {

System.out.println("高富帅");

System.out.println("180cm");

System.out.println("90kg");

}

public String getSex() {

return sex;

}

public void setSex(String sex) {

this.sex = sex;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

}

package com.proxy.jdk;

import java.lang.reflect.InvocationHandler;

import java.lang.reflect.Method;

import java.lang.reflect.Proxy;

public class Meipo implements InvocationHandler {

private Person target;

//获取被代理的资料

public Object getInstance(Person target) throws Exception{

this.target = target;

Class clazz = target.getClass();

return Proxy.newProxyInstance(clazz.getClassLoader(), clazz.getInterfaces(), this);

}

@Override

public Object invoke(Object proxy, Method method, Object[] args)

throws Throwable {

System.out.println("我是媒婆，你的性别是：" + this.target.getSex() + "得给你找个异性才行");

System.out.println("开始进行海选");

System.out.println("----------------");

this.target.findLove();

System.out.println("----------------");

System.out.println("如果合适的话，就准备办事");

return null;

}

}

package com.proxy.jdk;

public class Test {

public static void main(String[] args) {

try {

Person obj = (Person)new Meipo().getInstance(new ZhangSan());

obj.findLove();

} catch (Exception e) {

e.printStackTrace();

}

}

}

jdk实现的代码原理：

1. 生成源代码
2. 将生成的源代码输出到磁盘，保存.java文件
3. 编译源代码，并且生成.class文件

使用jdk动态代理专有的编译器：

avaCompiler compiler = ToolProvider.getSystemJavaCompiler();

StandardJavaFileManager manager = compiler.getStandardFileManager(null, null, null);

Iterable iterable = manager.getJavaFileObjects(files);

CompilationTask task = compiler.getTask(null, manager, null, null, null, iterable);

task.call();

manager.close();

1. 将class文件中的内容，动态加载到JVM中来

这个阶段就是将class类使用加载器加载到JVM当中

1. 返回被代理的代理对象

##### CGLib实现方法

使用CGLib.jar的时候需要asm.jar的支持

实现原理：

新建一个被代理对象的子类，然后返回赋值给这个父类（即被代理的类）。同样是做了字节码重组这样一件事

CGLib的使用代码：

package com.proxy.cglib;

public class ZhangSan {

private String sex = "女";

private String name = "张三";

public void findLove() {

System.out.println("高富帅");

System.out.println("180cm");

System.out.println("90kg");

}

public String getSex() {

return sex;

}

public void setSex(String sex) {

this.sex = sex;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

}

package com.proxy.cglib;

import java.lang.reflect.Method;

import net.sf.cglib.proxy.Enhancer;

import net.sf.cglib.proxy.MethodInterceptor;

import net.sf.cglib.proxy.MethodProxy;

public class CGMeipo implements MethodInterceptor {

public Object getInstance(class clazz) throws Exception{

Enhancer enhancer = new Enhancer();

//把父类设置为被代理的类

enhancer.setSuperclass(clazz);

enhancer.setCallback(this);

return enhancer.create();

}

@Override

public Object intercept(Object obj, Method method, Object[] args,

MethodProxy proxy) throws Throwable {

System.out.println("我是媒婆，你的性别是：" + "得给你找个异性才行");

System.out.println("开始进行海选");

System.out.println("----------------");

proxy.invokeSuper(obj, args);

System.out.println("----------------");

System.out.println("如果合适的话，就准备办事");

return null;

}

}

package com.proxy.cglib;

public class Test {

public static void main(String[] args) {

try {

ZhangSan obj =

(ZhangSan)new CGMeipo().getInstance(ZhangSan.class);

obj.findLove();

} catch (Exception e) {

e.printStackTrace();

}

}

}

CGLib的代码实现原理：

第一步、生成源代码

1. 编译成class文件
2. 加载到JVM中，并返回被代理对象

使用CGLib时很容易出现一个错误，那就是死循环。

死循环产生原因：

在intercept方法中使用了invoke方法了，而不是invokeSuper。而在invoke中也调用了intercept方法，这就产生了死循环。

### 工厂模式

#### 特点：

1. 隐藏复杂的逻辑过程，只关心结果。

简单工厂、工厂方法、抽象工厂

代码在工厂模式中实现统一管理、专业化管理。

#### spring中使用工厂模式的地方

spring中的工厂模式是生成Bean

在Spring中的工厂模式是BeanFactory（生产Bean的）

生产的Bean的种类：

单例的Bean

被代理过的Bean

最元素的Bean（原型）

List类型的Bean

作用于不同的Bean

Spring当中一个方法就解决了，即getBean()方法

#### 简单工厂模式代码：

package com.factory;

//汽车接口，汽车需要满足一定的标准

public interface Car {

//规定汽车的牌子

String getName();

}

package com.factory;

public class Audi implements Car {

@Override

public String getName() {

return "Audi";

}

}

package com.factory;

public class Bmw implements Car {

@Override

public String getName() {

return "Bmw";

}

}

package com.factory;

public class Benz implements Car {

@Override

public String getName() {

return "Benz";

}

}

package com.factory;

public class SimpleFactory {

//实现统一管理、专业化管理，如果没有工厂模式，小作坊，没有执行标准的

public Car getCar(String name){

if("Bmw".equals(name))

return new Bmw();

else if("Benz".equals(name))

return new Benz();

else if("Audi".equals(name))

return new Audi();

else

return null;

}

}

package com.factory;

public class SimpleFactoryTest {

public static void main(String[] args) {

//这边就是消费者

Car car = new SimpleFactory().getCar("Bmw");

if(car != null)

System.out.println(car.getName());

}

}

#### 工厂方法实现代码

package com.factory;

//汽车接口，汽车需要满足一定的标准

public interface Car {

//规定汽车的牌子

String getName();

}

package com.factory;

public class Audi implements Car {

@Override

public String getName() {

return "Audi";

}

}

package com.factory;

public class Bmw implements Car {

@Override

public String getName() {

return "Bmw";

}

}

package com.factory;

public class Benz implements Car {

@Override

public String getName() {

return "Benz";

}

}

package com.factory.func;

import com.factory.Car;

//工厂的接口就定义了所有工厂的执行标准

public interface Factory {

//符合汽车的一系列标准

Car getCar();

}

package com.factory.func;

import com.factory.Audi;

import com.factory.Car;

public class AudiFactory implements Factory {

@Override

public Car getCar() {

return new Audi();

}

}

package com.factory.func;

import com.factory.Benz;

import com.factory.Car;

public class BenzFactory implements Factory {

@Override

public Car getCar() {

return new Benz();

}

}

package com.factory.func;

import com.factory.Bmw;

import com.factory.Car;

public class BmwFactory implements Factory {

@Override

public Car getCar() {

return new Bmw();

}

}

package com.factory.func;

public class FactoryTest {

public static void main(String[] args) {

//工厂方法模式

Factory factory = new AudiFactory();

System.out.println(factory.getCar().getName());

Factory factory1 = new BmwFactory();

System.out.println(factory1.getCar().getName());

}

}

#### 抽象工厂实现代码：

package com.factory;

//汽车接口，汽车需要满足一定的标准

public interface Car {

//规定汽车的牌子

String getName();

}

package com.factory;

public class Audi implements Car {

@Override

public String getName() {

return "Audi";

}

}

package com.factory;

public class Bmw implements Car {

@Override

public String getName() {

return "Bmw";

}

}

package com.factory;

public class Benz implements Car {

@Override

public String getName() {

return "Benz";

}

}

package com.factory.abstr;

import com.factory.Audi;

import com.factory.Car;

public class AudiFactory extends AbstractFactory {

@Override

public Car getCar() {

return new Audi();

}

}

package com.factory.abstr;

import com.factory.Benz;

import com.factory.Car;

public class BenzFactory extends AbstractFactory {

@Override

public Car getCar() {

return new Benz();

}

}

package com.factory.abstr;

import com.factory.Bmw;

import com.factory.Car;

public class BmwFactory extends AbstractFactory {

@Override

public Car getCar() {

return new Bmw();

}

}

package com.factory.abstr;

import com.factory.Car;

public abstract class AbstractFactory {

public abstract Car getCar();

public Car getCar(String name){

if("Bmw".equals(name))

return new BmwFactory().getCar();

else if("Benz".equals(name))

return new BenzFactory().getCar();

else if("Audi".equals(name))

return new AudiFactory().getCar();

else

return null;

}

}

package com.factory.abstr;

import com.factory.Car;

public class DefultFactory extends AbstractFactory {

private AudiFactory d = new AudiFactory();

@Override

public Car getCar() {

return d.getCar();

}

}

package com.factory.abstr;

public class AbstractFactoryTest {

public static void main(String[] args) {

DefultFactory factory = new DefultFactory();

System.out.println(factory.getCar("Bmw").getName());

}

}

### 单例模式

#### 什么是单例模式：

1. 保证从系统启动到系统停止，全过程只会产生一个实例。

#### 为什么要有单例模式：

哪些情况下用到了单例模式？

穷举法：

配置文件：如果不是单例（两个配置文件中的内容一样的，则有一个是浪费的，如果是不一 样的，我们就不知道以哪个为准了）

直接上级领导：

#### 特点：

1. 保证从系统启动到系统终止，全过程只会产生一个实例。
2. 当我们在应用中遇到功能性冲突的时候，需要使用单例模式。

#### 常用单例模式写法

1.饿汉式(线程安全，调用效率高，但是不能延时加载)：

public class ImageLoader{

     private static ImageLoader instance = new ImageLoader;

     private ImageLoader(){}

     public static ImageLoader getInstance(){

          return instance;

      }

}

一上来就把单例对象创建出来了，要用的时候直接返回即可，这种可以说是单例模式中最简单的一种实现方式。但是问题也比较明显。单例在还没有使用到的时候，初始化就已经完成了。也就是说，如果程序从头到位都没用使用这个单例的话，单例的对象还是会创建。这就造成了不必要的资源浪费。所以不推荐这种实现方式。

2.懒汉式(线程安全，调用效率不高，但是能延时加载)：

public class SingletonDemo2 {

//类初始化时，不初始化这个对象(延时加载，真正用的时候再创建)

private static SingletonDemo2 instance;

//构造器私有化

private SingletonDemo2(){}

//方法同步，调用效率低

public static synchronized SingletonDemo2 getInstance(){

if(instance==null){

instance=new SingletonDemo2();

}

return instance;

}

}

饿汉式也顾名思义，就是这个汉子比较懒，一开始的时候什么也不做，知道要使用的时候采取创建实例的对象。看起来还不错，只有在使用实例的时候，我们才回去创建对象。但是细心的同学可能发现了，我们在获取实例的方法上加了锁，避免多线程引发的创建多个单例的情况。多线程的问题是避免了，但也造成了整体性能的下降，每次使用单例对象，都需要锁判断，降低了整体性能。很明显，懒汉式也不是我们所要追求的目标。

3.Double CheckLock实现单例：DCL也就是双重锁判断机制（由于JVM底层模型原因，偶尔会出问题，不建议使用）：

public class ImageLoader{

private static ImageLoader instance；

 private ImageLoader(){}

public static ImageLoader getInstance(){

if(instance == null){

synchronized （ImageLoader.class）{

if(instance == null){

instance = new ImageLoader();

}

  }

 }

return instance;

}

}

可以看到，在获取单例对象的时候，我们先进行了两为空判断，并且在第二次判断前加了锁，这就让程序变得更加优秀，在使用的时候，只会前几次获取单例对象的时候会进行锁判断，一旦单例对象创建完成，锁的任务也就完成了，在懒汉式的基础上，提高了性能。DCL是使用最多的单例实现方式，能够在使用的时候才进行单例对象的初始化创建，并且能够在绝大多数情况下保证对象的唯一性的正确性。请注意，是绝大多数情况下，也就是说，这种模式也不能完全保证单例的对象的完美实现，但是，就一般情况下，这种模式都能满足需求。俗话说，学无止境，接下来我们就来看看单例模式的终极实现版本。

4.静态内部类实现模式（线程安全，调用效率高，可以延时加载）

public class SingletonDemo3 {

private static class SingletonClassInstance{

private static final SingletonDemo3 instance=new SingletonDemo3();

}

private SingletonDemo3(){}

public static SingletonDemo3 getInstance(){

return SingletonClassInstance.instance;

}

}

这里面使用了final修饰是为了防止内部误操作，代理模式，CGLib的代理模式下，前面或后面代码也许会修改这个instance变量

可以发现这种方式，并未加锁，因为第一次加载ImageLoader类时，并不会实例化单例对象，只有第一次调用getInstance()方法时会导致虚拟机加载InnerInstance类，这种  
方式不仅能保证对象的单一性，还避免加锁带来的性能问题，又启动了延迟加载的优化，所以这就是单例模式的终极实现版本，也是推荐使用的方式。

5.枚举类（线程安全，调用效率高，不能延时加载，可以天然的防止反射和反序列化调用）

public enum SingletonDemo4 {

//枚举元素本身就是单例

INSTANCE;

//添加自己需要的操作

public void singletonOperation(){

}

}

如何选用：

-单例对象 占用资源少，不需要延时加载，枚举 好于 饿汉

-单例对象 占用资源多，需要延时加载，静态内部类 好于 懒汉式

### 委派模式

#### spring中使用委派模式的地方

spring的IOC容器中，有一个Register的东西（为了告诉我们的容器，在这个类被初始化的过程中，需要做很多不同逻辑处理，需要实现多个任务执行者，分别实现各自的功能）

保证结果的多样性，对于用户来是只有一种方法

#### 什么是委派模式

两个角色：受托人和委托人（社会上是平等的关系）

公司里面：项目经理，普通员工（法律上是平等的，工作的关系，各自的职责会不一样）

项目经理（委托人）：主要职责是安排任务

普通员工（受托人）：执行任务

#### 特点：

1. 类似于中介的功能（委托机制）
2. 持有被委托人的引用
3. 不关心过程，只关心结果

要和代理模式区分开

#### 为什么要用委派模式：

主要目的就是隐藏具体实现逻辑

#### 委派模式的代码

package com.delegate;

public interface IExector {

//具体执行，在公司中，员工执行任务，规定在一周之内必须完成

void doing();

}

package com.delegate;

//想法，用代码来描述这种想法

public class ExectorA implements IExector {

@Override

public void doing() {

System.out.println("员工A开始执行");

}

}

package com.delegate;

//想法，用代码来描述这种想法

public class ExectorB implements IExector {

@Override

public void doing() {

System.out.println("员工B开始执行");

}

}

package com.delegate;

public class Dispatcher implements IExector {

IExector exector;

Dispatcher(IExector exector){

this.exector = exector;

}

@Override

public void doing() {

this.exector.doing();

}

}

package com.delegate;

public class DispatcherTest {

public static void main(String[] args) {

Dispatcher dispatcher = new Dispatcher(new ExectorA());

dispatcher.doing();

}

}

### 策略模式

1. 过程不同，但结果一样

策略模式的代码实现：

List<Long> numbers = new ArrayList<Long>();

Collections.sort(numbers, new Comparator<Long>(){

@Override

//返回值是固定的

//0、-1、1

public int compare(Long o1, Long o2) {

return 0;

//这里由用户自己实现

}

});

#### spring中使用策略模式的地方

回调处理

### 原型模式

1、过程相同，但结果不一样。或者叫模板模式（提高开发效率）

#### 深拷贝和浅拷贝的概念

浅拷贝：能够直接拷贝其实际内容的数据类型。只支持9种（java的八大基本数据类型和String），其他对象类型的都只会复制引用，实际用的是同一块地址

深拷贝：基于字节码的拷贝

clone方法就是浅拷贝：

深拷贝的代码为：

public Object clone(){

//深度克隆

ByteArrayOutputStream bos = null;

ObjectOutputStream oos = null;

ByteArrayInputStream bis = null;

ObjectInputStream ois = null;

try {

//return super.clone();//默认浅克隆，只克隆八大基本数据类型和String

//序列化

bos = new ByteArrayOutputStream();

oos = new ObjectOutputStream(bos);

oos.writeObject(this);

//反序列化

bis = new ByteArrayInputStream(bos.toByteArray());

ois = new ObjectInputStream(bis);

TheGreatestSage copy = (TheGreatestSage)ois.readObject();

copy.birthday = new Date();

return copy;

} catch (Exception e) {

e.printStackTrace();

return null;

}finally{

try {

bos.close();

oos.close();

bis.close();

ois.close();

} catch (IOException e) {

e.printStackTrace();

}

}

}

#### 为什么使用原型结构

主要是负责拷贝那些复杂的数据结构的数据

Fast Json

### 模板模式

1. 执行流程一样，但中间有些步骤不同

模板模式的代码：

package com.template;

//冲饮料(拿出去卖钱了)

public abstract class Bevegrage {

//不能被重写

public final void create(){

//1、把水烧开

boilWater();

//2、把杯子准备好、原材料放到杯中

pourInCup();

//3、用水冲泡

brew();

//4、添加辅料

addCoundiments();

}

public abstract void pourInCup();

public abstract void addCoundiments();

public void brew(){

System.out.println("将开水放入杯中进行冲泡");

};

public void boilWater(){

System.out.println("烧开水，烧到100度可以起锅了");

}

}

package com.template;

public class Tea extends Bevegrage{

//原材料放到杯中

public void pourInCup() {

System.out.println("将茶叶放入杯中");

}

//房辅料

public void addCoundiments() {

System.out.println("添加蜂蜜");

}

}

package com.template;

public class Coffee extends Bevegrage{

//原材料放到杯中

public void pourInCup() {

System.out.println("将咖啡倒入杯中");

}

//房辅料

public void addCoundiments() {

System.out.println("添加牛奶和糖");

}

}

package com.template;

public class TestTemplate {

public static void main(String[] args) {

// Coffee coffee = new Coffee();

// coffee.create();

Tea tea = new Tea();

tea.create();

}

}

#### spring中使用模板模式的地方

SpringJDBC

是java规范，各个数据库厂商自己去实现

1、加载驱动类DriverManager

2、建立连接

3、创建语句集(标准语句集、预处理语句集)(语句集？ MySQL、Oracle、SQLServer、Access)

4、执行语句集

5、结果集ResultSet 游标

ORM(?)

## 设计模式思想总结

### 代理模式（Proxy）

#### 应用场景（特点）

1. 两个角色：执行者、被代理人
2. 对于被代理人来说，这件事情是一定要做的，但是我自己又不想做或者没有时间做，找代理
3. 需要获取到被代理的人个人资料

#### 一句话归纳

办事要求人，所以找代理

### 工厂模式（Factory）

#### 应用场景（特点）

1. 对调用者隐藏复杂的逻辑处理过程，调用者只关心执行结果。
2. 工厂要对结果负责，保证生产出符合规范的产品

#### 一句话归纳

只对结果负责，不要三无产品

### 单例模式（Singleton）

#### 应用场景（特点）

1. 保证从系统启动到系统终止，全过程只会产生一个实例
2. 当我们在应用中遇到功能性冲突的时候，需要使用单例模式

#### 一句话归纳

保证独一无二

### 委派模式（Delegate）

#### 应用场景（特点）

1. 两个参与角色，委托人和被委托人
2. 委托人和被委托人在权利上完全平等（即实现同一个接口）
3. 委托人持有被委托人的引用
4. 不关心过程，只关心结果

#### 一句话归纳

干活是你的（普通员工），功劳是我的（项目经理）

### 策略模式（Strategy）

#### 应用场景（特点）

1. 最终执行结果是固定的
2. 执行过程和执行逻辑不一样

#### 一句话归纳

我行我素，达到目的就行

### 原型模式（Prototype）

#### 应用场景（特点）

1. 首先有一个原型
2. 数据内容相同，但对象实例不同（完全两个个体）

#### 一句话归纳

拔一根猴毛，吹出千万个

### 模板模式（Template）

#### 应用场景（特点）

1. 执行流程固定，但中间有些步骤有细微差别。
2. 可实现批量生产

#### 一句话归纳

流程标准化，原料自己加