Java Spring 框架思想

~~此文为手打总结，部分附加注释理解，仅供个人学习~~

> 原文地址

主要运用的设计模式

框架可能不具备迁移性，但设计模式可以。Spring 中充分运用了下面这些设计模式：

- **工厂模式（Factory）**：所谓工厂就是屏蔽实例化（new）的细节，调用处无需关心实例化对象需要的环境参数，提升可维护性。Spring 的 BeanFactory 创建 bean 对象就是工厂模式的体现。**对应 js 的构造函数**
- **代理模式（Proxy）**：允许通过代理对象访问目标对象。Spring 实现 AOP 就是通过动态代理模式。**对应 js 的 proxy**
- **单例模式（Singleton）**：单实例确保一个类的全局唯一性。Spring 的 bean 默认都是单例。**对应 js 的 window 对象就是常见的单例模式**
- **包装器模式（Decorator）**：更常见的叫法应该是**装饰器模式**，将几个不同方法通用部分抽象出来，调用时通过包装器内部引导到不同的实现。比如 Spring 连接多种数据库就使用了包装器模式简化。**对应 ECMA 迟迟未进正式规范的 decorator**
- **观察者模式（Observer）**：最常见的，也就是事件机制，Spring 中可以通过 ApplicationEvent 实践观察者模式。**对应 js 中各种事件监听**
- **适配器模式（Adapter）**：通过适配器将接口转换为另一个格式的接口。Spring AOP 的增强和通知就使用了适配器模式。**对应前端请求库 axios 里面的 transformRequest 方法**
- **模板方法模式（Abstract Method）**：父类先定义一些函数，这些函数之间存在调用关联，将某些设定为抽象函数等待子类继承时去重写。Spring 的 jdbcTemplate、hibernateTemplate 等数据库操作类使用了模版方法模式。**对应 es6 中的 abstract class**

生态全家桶

- **Spring Boot**：简化了 Spring 应用的配置，约定大于配置
- **Spring Data**：数据操作和访问的工具集，比如支持 jdbc、redis 等数据源的操作
- **Spring Cloud**：微服务解决方案，基于 Spring Boot，继承了服务发现、配置管理、消息总线、负载均衡、断路器、数据监控等各种服务治理能力
- **Spring Security**：支持一些安全模型比如单点登录、令牌中继、令牌交换等
- **Spring MVC**：MVC WEB 框架

IOC（Inverse Of Control）控制反转

Spring 的核心，任何对象的调用都离不开 IOC。

假设有三个类：Country、Province、City，最大的类别是国家，其次是省、城市，国家类需要调用省类，省类需要调用城市类：

```
public class Country {
  private Province province;
  public Country(){
    this.province = new Province()
  }
}

public class Province {
  private City city;
  public Province(){
    this.city = new City()
  }
}

public class City {
  public City(){
    // ...
  }
}
```

假设来了一个需求，City 实例化时需增加人口（people）参数，就要改动所有类代码：

```
public class Country {
  private Province province;
  public Country(int people){
    this.province = new Province(people)
  }
}

public class Province {
  private City city;
  public Province(int people){
    this.city = new City(people)
  }
}

public class City {
  public City(int people){
    // ...
  }
}
```

那么在真实业务场景中，一个底层类可能被数以千计的类使用，这么改显然难以维护。IOC 就是为了解决这个问题，它使得可以只改动 City 的代码，而不用改动其他类的代码：

```
public class Country {
  private Province province;
  public Country(Province province){
    this.province = province
  }
}

public class Province {
  private City city;
  public Province(City city){
    this.city = city
  }
}

public class City {
  public City(int people){
    // ...
  }
}
```

可以看到，增加 people 属性只需要改动 city 类。然而这样做也是有成本的，就是类实例化步骤会稍微繁琐一些：

```
City city = new City(1000);
Province province = new Province(city);
Country country = new Country(province);
```

这就是所谓的控制反转，由 Country 依赖 Province 变成了类依赖框架（上面的实例化代码）注入。

然而手动维护这种初始化依赖是繁琐的，Spring 提供了 bean 容器自动做这件事，简单理解相当于把依赖进行配置化提升并自动中心化管理，只需要利用装饰器 Autowired 就可以自动注入依赖：

```
@Component
public class Country {
  @Autowired
  private Province province;
}
@Component
public class Province {
  @Autowired
  public City city;
}
@Component
public class City {
}
```

实际上这种自动分析并实例化的手段，不仅比手写方便，还能解决循环依赖的问题。在实际场景中，两个类相互调用是很常见的，假设现在有 A、B 类相互依赖：

```
@Component
public class A {
    @Autowired
    private B b;
}
@Component
public class B {
    @Autowired
    public A a;
}
```

那么假设想获取 A 实例，会经历这样一个过程：

获取 A 实例 -> 实例化不完整 A -> 检测到注入 B -> 实例化不完整 B -> 检测到注入 A -> 注入不完整 A -> 得到完整 B -> 得到完整 A -> 返回 A 实例

其实 Spring 仅支持单例模式下非构造器的循环依赖，这是因为其内部有一套机制，让 bean 在初始化阶段先提前持有对方引用地址，这样就可以同时实例化两个对象了。

除了方便之外，IOC 配合 Spring 容器概念还可以使获取实例时不用关心一个类实例化需要哪些参数，只需要直接申明获取即可，这样在类的数量特别多，尤其是大量代码不是你写的情况下，不需要阅读类源码也可以轻松获取实例，实在是大大提升了可维护性。

说到这就提到了 Bean 容器，在 Spring 概念中，Bean 容器是对 class 的加强，如果说 Class 定义了类的基本含义，那 Bean 就是对类进行使用拓展，告诉应该如何实例化与使用这个类。

举个例子，比如利用注解描述的这段 Bean 类：

```
@Configuration
public class CityConfig {
  @Scope("prototype")
  @Lazy
  @Bean(initMethod = "init", destroyMethod = "destroy")
  public City city() {
    return new City()
  }
}
```

可以看到，额外描述了是否延迟加载，是否单例，初始化与析构函数分别是什么等等。

下面给出一个从 Bean 获取实例的例子，采用比较古老的 xml 配置方式：

```
public interface City {
  Int getPeople();
}
```

```
public class CityImpl implements City {
  public Int getPeople() {
    return 1000;
  }
}
```

接下来用 xml 描述这个 bean：

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

  <bean id="city" class="xxx.CityImpl"/>
</beans>
```

bean 支持的属性还有很多，就不一一列举了，总之 id 是一个可选的唯一标志，接下来可以通过 id 访问到 city 的实例。

```
public class App {
  public static void main(String[] args) {
    ApplicationContext context = new ClassPathXmlApplicationContext("classpath:application.xml");

    // 从 context 中读取 Bean，而不 new City()
    City city = context.getBean(City.class);

    System.out.println(city.getPeople());
  }
}
```

AOP（Aspect Oriented Program）面向切面编程

AOP 是为了解决主要业务逻辑与次要业务逻辑之间耦合问题的。

主要业务逻辑比如登陆、数据获取、查询等；次要业务逻辑比如性能监控、异常处理等等，次要业务逻辑往往有：不重要、和业务关联度低、贯穿多处业务逻辑的特性，如果没有好的设计模式，只能在业务代码里将主要逻辑与次要逻辑混合起来。但 AOP 可以做到主要、次要业务逻辑隔离。

使用 AOP 就是在定义在哪些地方（类、方法）切入，在什么地方切入（方法前、后、前后）以及做什么。

比如说，想在某个方法前后分别执行两个函数计算执行时间，下面是主要业务逻辑：

```
@Component("work")
public class Work {
  public void do() {
    System.out.println("执行业务逻辑");
  }
}
```

再定义切面方法：

```
@Component
@Aspect
class Broker {
  @Before("execution(* xxx.Work.do())")
  public void before(){
    // 记录开始时间
  }

  @After("execution(* xxx.Work.do())")
  public void after(){
    // 计算时间
  }
}
```

再通过 xml 定义扫描下这两个 Bean，就可以在运行 `work.do()` 之前执行 `before()`，之后执行 `after()`。

还可以完全覆盖原函数，利用 `joinPoint.proceed()` 可以执行原函数：

```
@Component
@Aspect
class Broker {
  @Around("execution(* xxx.Work.do())")
  public void around(ProceedingJoinPoint joinPoint) {
    // 记录开始时间

    try {
      joinPoint.proceed();
    } catch (Throwable throwable) {
      throwable.printStackTrace();
    }

    // 计算时间
  }
}
```

关于表达式 `execution(* xxx.Work.do())` 是用正则的方式匹配，* 表示任意返回类型的方法，后面就不用解释了。

可以看到，可以在不修改原方法的基础上，在其执行前后增加自定义业务逻辑，或者监控其报错，非常适合做次要业务逻辑，且由于不与主要业务逻辑代码耦合，保证了代码的简洁，且次要业务逻辑不容易遗漏。

总结

IOC 特别适合描述业务模型，后端天然需要这一套，然而随着前端越做越重，如果某个业务场景下需要将部分业务逻辑放到前端，也是非常推荐使用 IOC 设计模式来做，**这是后端沉淀了近 20 年的经验，没有必要再另辟蹊径**。

AOP 对前端有帮助但没有那么大，因为前端业务逻辑较为分散，如果要进行切面编程，往往用 window 事件监听来做会更彻底，可能这都是前端没有流行 AOP 的原因。当然前端约定大于配置的趋势下，比如打点或监控都集成到框架内部，往往也做到了业务代码无感，剩下的业务代码也就没有 AOP 的需求。

最后，Spring 的低侵入式设计，使得业务代码不用关心框架，让业务代码能够快速在不同框架间切换，这不仅方便了业务开发者，更使得 Spring 走向成功，这是前端还需要追赶的。