Spring的诞生史

一、知史可以明鉴

我们学习技术的时代赶上了最好的时代，跳过了很多前人经常踩的坑，前人在踩坑的过程中总结了很多经验和教训，而新时代的我们只是继承了前人的经验和教训，而忽略了这些采坑的过程，以至于我们面对很多新技术都不知道他是什么?他为什么存在?他为什么可以解决这个问题?更不知道如何掌握其原理!云里雾里一头雾水!

交流群的很多小伙伴，常常私聊我让我推荐一下学习SSM框架的视频和资料，我首先会打开他的资料卡看一下他的年龄，如果超过了他这个年龄应有的水平，我就会问他JSP+Servlet学了吗?很多小伙伴的回答是简单的学了一下，然后，我会给他一个关于JSP+Servlet的实战项目，顺便给他们找一些SSM的项目，并且建议他们首先看这个JSP+Servlet的实战项目。

更有甚者，学了基础之后就开始学习Spring Boot的，当问他们Spring Boot是什么的时候，大致也可以回答出来“约定大于配置”，“用起来很简单”，但是在细究其原理，也是吱吱呜呜，一知半解!如果我们没有经历过Spring最开始繁琐的配置、然后一步步精简，根本体会不到为什么会有Spring Boot这个东西!

不先学习常见的设计模式直接看Spring、MyBatis等源码，简直就是一个找虐的过程!不掌握Servlet原理、基本的Tomcat容器技术上来就看Spring MVC源码同样也是一个打击自信心的好地方!

学习是一个循序渐进的过程，不能急于求成，但也不能过分钻牛角尖!不能再一个技术上停滞不前，也不能如”蜻蜓点水”一般了了掠过!同样，如果你还没有掌握好Servlet和简单的设计模式我建议你先去查阅相关的资料进行系统的学习。

我也相信很多图书或视频等资料都忽略了讲述为什么会有Spring的过程，要么是简单概括并且痛斥EJB的各种弊端，要么就是只字不提，这是一种对读者很不负责任的表现，知史可以明鉴!因此，在进一步学习Spring核心原理之前，我们有必要介绍一下整个Web发展的简单历史，一步步引出为什么会有Spring!

二、Web发展简史

老一辈的软件开发人员一般经历了从Model1到Model2，然后到后来的三层模型，最后到现在的Spring Boot。如果从Model1到Model2说起到我们现在使用的

Spring Boot为整个时间轴的话，大致可以分为4个阶段：

(1)初级阶段：使用Model1/Model2/三层模模型进行开发;

(2)中级阶段：使用EJB进行分布式应用开发，忍受重量级框架带来的种种麻烦;

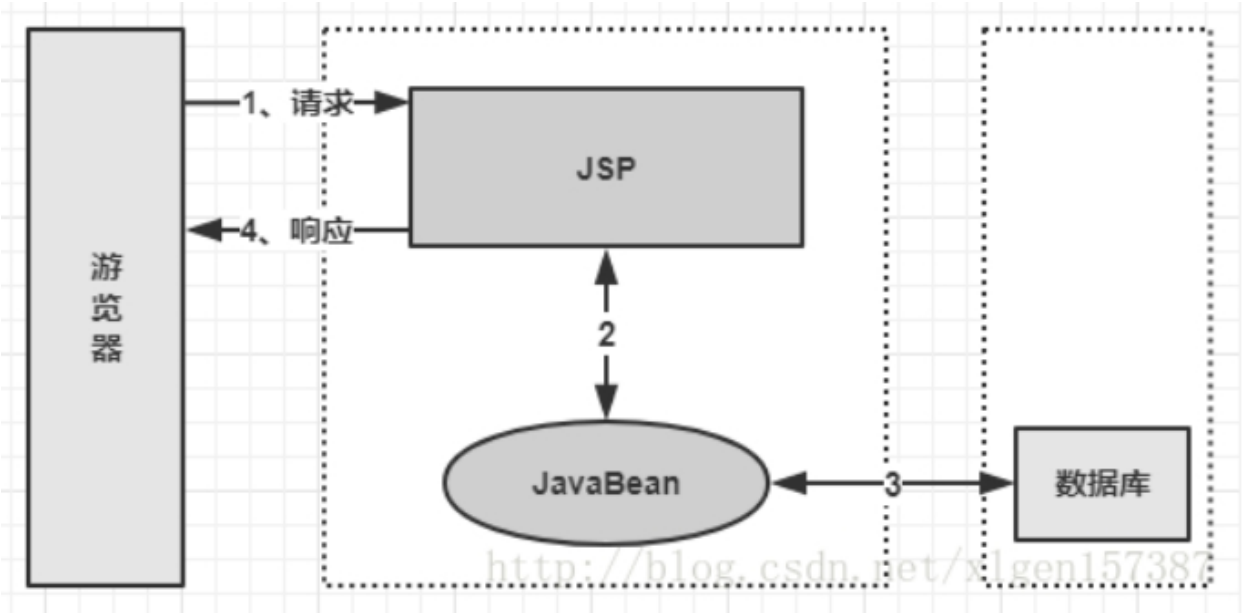
(3)高级阶段：使用Spring春天带给我们的美好，但是还要忍受很多繁琐的配置;

(4)骨灰级阶段：使用Spring Boot，畅享“预定大于配置”带给我们的种种乐趣!

三、Web发展初级阶段

1、Model1开发模式：

Model1的开发模式是：JSP+JavaBean的模式，它的核心是Jsp页面，在这个页面中，Jsp页面负责整合页面和JavaBean(业务逻辑)，而且渲染页面，它的基本流程如下：



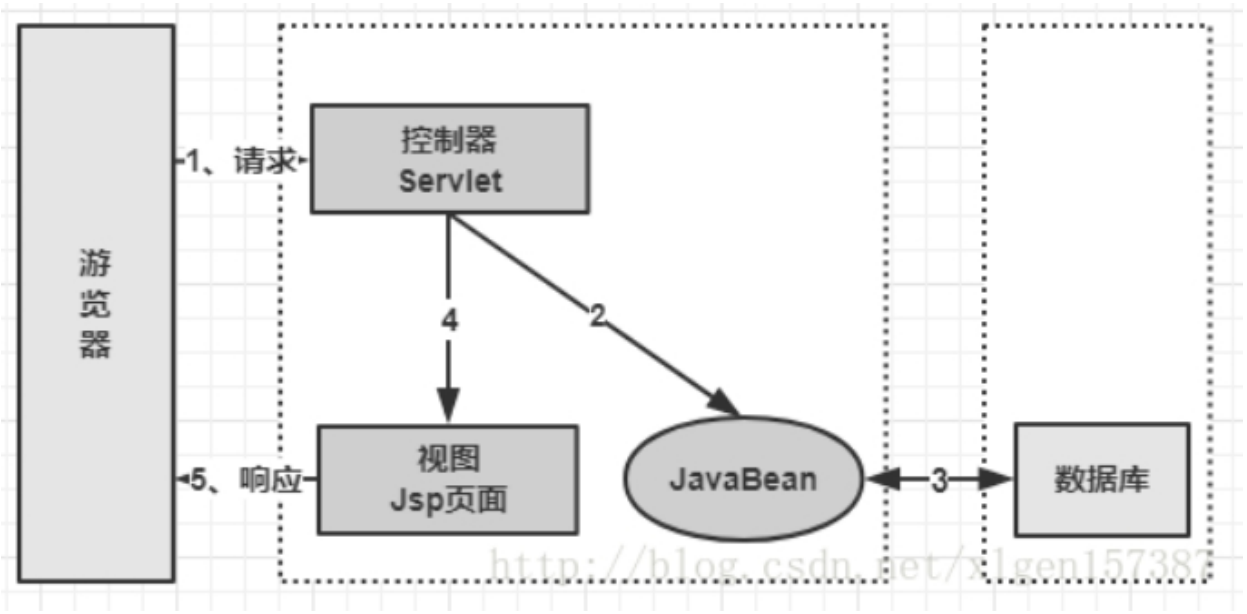
相信很多小伙伴在刚学习Web的时候，肯定使用到了Model1开发模式，也就是我们的业务代码、持久化代码直接写在Jsp页面里边，使用Jsp直接处理Web浏览器的请求，并使用JavaBean处理业务逻辑。

利用我们现在熟悉的MVC模型的思想去看，虽然编写代码十分容易，但Jsp混淆了MVC模型中的视图层和控制层，高度耦合的结果是Jsp代码十分复杂，后期维护困难!

2、Model2开发模式：

Model1虽然在一定程度上解耦了，但JSP依旧即要负责页面控制，又要负责逻辑处理，职责不单一!此时Model2应运而生，使得各个部分各司其职，Model2是基于MVC模式的。

Model2的开发模式是：Jsp+Servlet+JavaBean的模式，它和Model1不同的是，增加了Servlet，将调用页面数据，调用业务逻辑等工作放到了Servlet中处理，从而减轻了Jsp的工作负担!它的基本流程如下：



Model2开发模式将Servlet的概念引入架构体系中，使用它来分配视图层Jsp的显示页面，同时调用模型层的JavaBean来控制业务逻辑。

3、Model1和Model2的区别：

Model1：简单，适合小型项目的开发，但是Jsp的职责过于繁重，职责分工不明确。在后期的维护工作中，必将为此付出代价!

Model2：相对于Model1来说，职责分工更为明确，在Model1的基础上，抽取了Servlet层，体现了一个分层的思想，适合大型的项目开发!(当时的评判标准是适合大型项目开发的，现在看起来已经过时了!)

Model2看起来已经尽善尽美了，尽管如此，他还不能称之为一个比较完善的MVC设计模式!

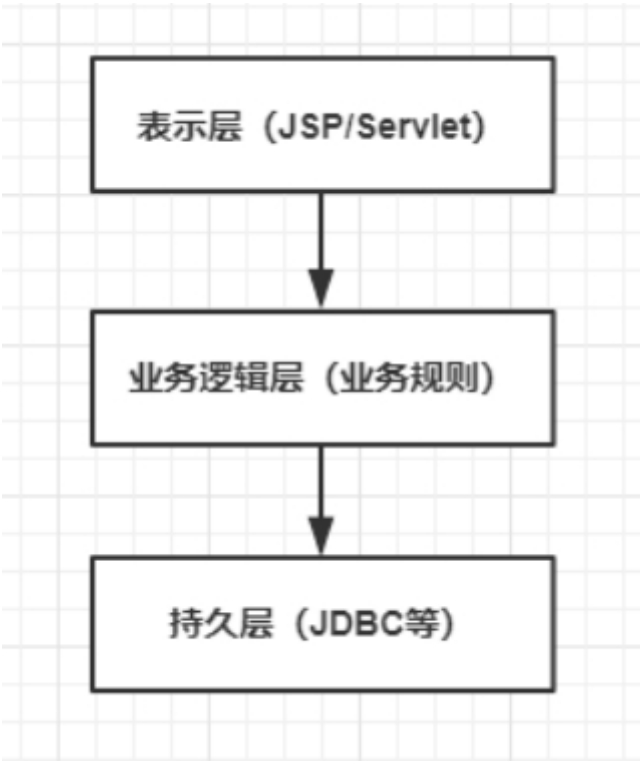
4、Model1和Model2与三层的对比：

在Model2中，我们将Servlet抽取出单独的一层，和Jsp协作完成用户数据交互的工作，也就是表示层。那么作为三层结构来说，又做了什么样的改进呢?三层则是在此基础上，将JavaBean再一次进行分割：业务逻辑、数据持久化，三层如下：

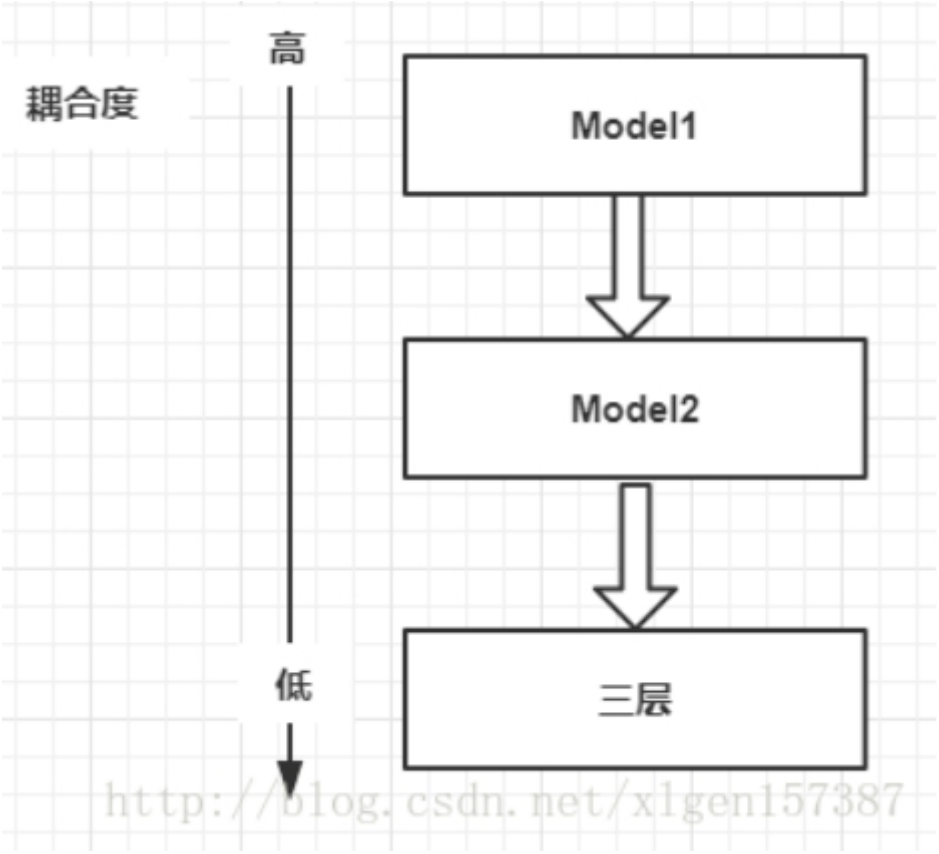
(1)表示层，JSP/Servlet;

(2)业务逻辑层：业务规则;

(3)持久化层：主要包装持久化的逻辑 ;



各个的耦合性如下图：



Model1、Model2、三层是在解耦的基础上一步步进化而来，通过解耦我们可以进行进一步的抽象，以应对现实需求的变动。

四、Web发展中级阶段、高级阶段和骨灰级阶段

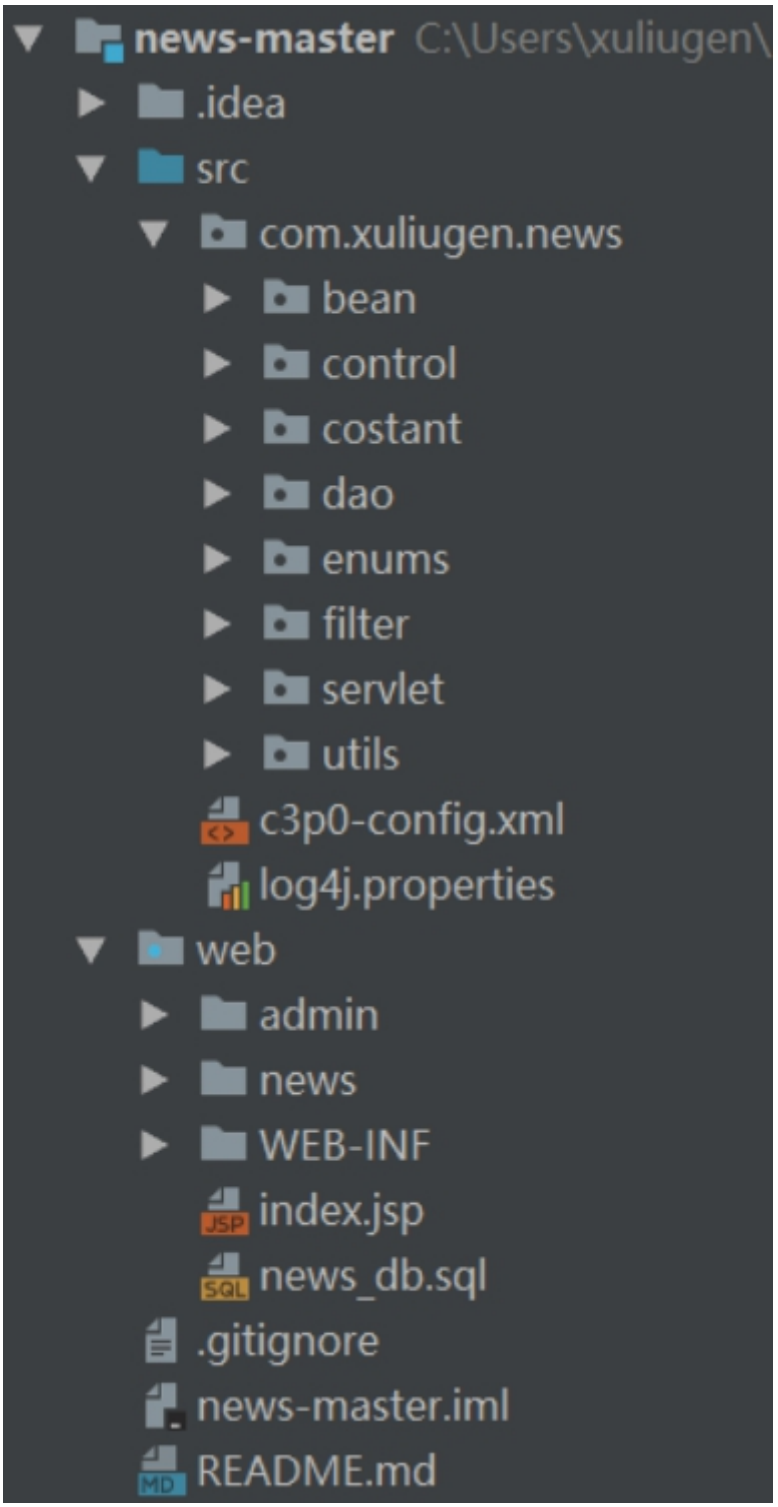
这一小节似乎有点应付，对于中级阶段，因为我没有用过EJB，在这里不敢妄加评论，以免误导大家。但是相信每一位接触过Spring的小伙伴，都应该知道Rod Johnson在2002年编写的《Expert One-to-One J2EE Design and Development》一书，Rod 在本书中对J2EE正统框架臃肿、低效、脱离现实的种种学院派做法提出了质疑，并以此书为指导思想，编写了interface21框架，也就是后来的Spring。

对于高级阶段和骨灰级阶段是我们后期一系列文章的重点，本篇只作为一个阶段划分，不做过多的解释，因此让我们重新回到Web发展的初级阶段。

对EJB有兴趣的可以参考文章：http://www.uml.org.cn/j2ee/2009112011.asp

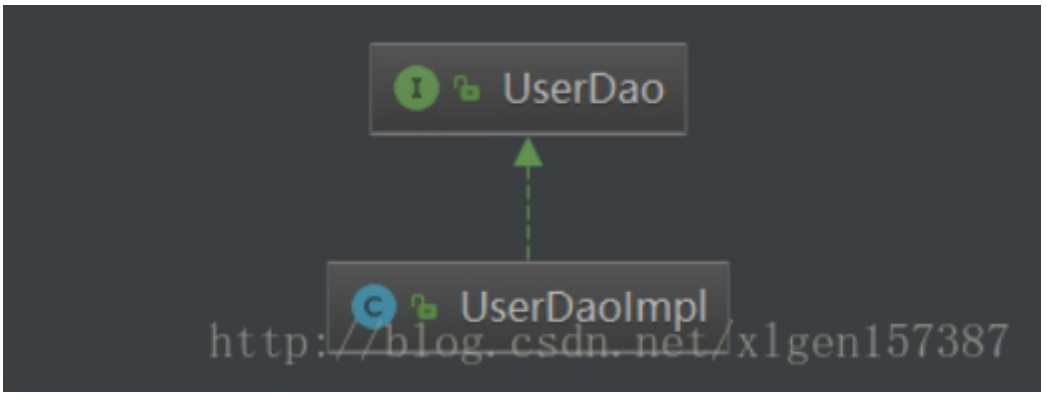
五、Web发展初级阶段存在的问题

经历过初级阶段的小伙伴肯定看得懂下边的一个项目结构，一个简单的MVC三层结构，使用JSP+Servlet+MySQL+JDBC技术，面向接口编程：



1、面向接口编程的实例化对象

以用户管理模块为例，有一个UserDao接口，有一个接口的实现类UserDaoImpl，如下：



由于是面向接口编程，因此我们在每次使用UserDao的时候，都要进行实例化一次，实例化代码如下：

```
UserDao userDao = new UserDaoImpl();
```

我们在每次使用UserDao的时候都需要进行实例化，当然不仅仅有UserDao需要进行实例化，还有很多需要进行实例化的，举例如下：

```
public class UserControllerImpl implements UserController {

    public boolean register(User user) {
        UserDao userDao = new UserDaoImpl();
        return userDao.register(user);
    }

    public User login(String username, String password) {
        UserDao userDao = new UserDaoImpl();
        return userDao.login(username, password);
    }
```


可以看出，每一个方法中都需要进行实例化我们需要用到的接口的实现类，这就会存在大量的实例化对象，并且他们的生命周期可能就是从方法的调用开始到方法的调用结束为止，加大了GC回收的压力!

2、使用单例模式的一次改进

了解设计模式的可能会想到使用单例模式的方式来解决这个问题，以此来避免大量重复的创建对象，但是我们还要考虑到众多的这种对象的创建都需要改成单例模式的话，是一个耗时耗力的操作。

对于这个系统来说，如果都把这种面向接口的对象实现类转换为单例模式的方式的话，大概也要写十几个或者上百个这种单例模式代码，而对于一个单例模式的写法来说，往往是模板式的代码，以静态内部类的方式实现代理模式如下：

```
public class UserDaoSingletonImpl implements UserDao {

    private static class SingletonHolder {
        private static final UserDaoSingletonImpl INSTANCE = new UserDaoSingletonImpl();
    }

    private UserDaoSingletonImpl() {

    }

    public static final UserDaoSingletonImpl getInstance() {
        return SingletonHolder.INSTANCE;
    }

    @Override
    public boolean register(User user) {
        return false;
    }

    @Override
    public User login(String username, String password) {
        return null;
    }
```

单利模式的模板代码

具体的业务代码

可以看出，这种方式有两个问题：

(1)业务代码与单例模式的模板代码放在一个类里，耦合性较高;

(2)大量重复的单例模式的模板代码;

从上述可以看出，使用的单例模式虽然从性能上有所提高，但是却加重了我们的开发成本。因此只会小规模的使用，例如我们操作JDBC的Utils对象等。

3、我们开发中遇到的痛点

从上述代码的演进过程我们可以看得出来，我们即需要一个单例的对象来避免系统中大量重复对象的创建和销毁，又不想因为使用单例模式造成大量重复无用的模板代码和代码的耦合!

(突然想到一个段子，想和大家分享一下：产品经理在给甲方汇报方案的时候说了两种方案：一种是实用的，一种是美观的，问甲方希望选择哪一种?甲方说：有没有即实用又美观的!)

4、我们还能怎么做

作为学院派的书生来说，我们可能会联想到“数据库连接池”,我们在获取数据库连接的时候会从这个池子中拿到一个连接的，假设这个数据库连接池很特殊，有且只能有N个数据库连接，并且每一个连接对象都不同(假设)，那么这个不就相当于每一个连接都是单例的了吗?既可以避免大量对象的创建，也可以实现不会出现大量重复性的模板代码。

因此，这里应该有一个大胆的想法，我们是否可以建立一个池子，将我们的接口实现类对象放入到这个池子中，我们在使用的时候直接从这个池子里边取就行了!

5、这个池子

如果我们要创建这个池子，首先要确定需要把哪些对象放进这个池子，通过怎样的方式放进去，放进去之后如何进行管理，如何进行获取，池子中的每一个对象的生命周期是怎么样的等等这些东西都是我们需要考虑到的!

6、恭喜你

如果你已经了解了上述Web演进的过程，以及我们想要创建的这个池子，那么恭喜你!你已经打开了Spring核心原理的大门了!

上述我们想要创建的池子其实就是Spring容器的雏形，将接口实现类的对象放进池子进行管理的过程其实也是Spring IOC依赖注入、控制反转的雏形!

Spring的依赖注入/控制反转就是从我们的配置文件或注解中的得到我们需要进行注入到Spring容器的实现类的信息，Spring IOC通过这些配置信息创建一个个单例的对象并放入Spring容器中，Spring容器可以看做是一个集合保存着我们的这些对象。

7、小总结

上文中主要从一个切入点探讨了一下为什么有Spring，以及介绍了一下Spring IOC和Spring容器的基本雏形概念，当然还可以从其他方面进行切入。这里没有进一步探讨AOP的概念，对于新入门的小伙伴来说，这个确实有必要讨论一下，也决定在后续文章中由浅入深的探讨一下，而对于老手来说，其实我上边写的基本上是浪费大家时间的!

六、总结

从历史的角度来说，不同时期的大革命在爆发之前，都会有一个蓄谋已久的“导火线”!Spring的出现，同样顺应了历史发展潮流，正式由于那个时期J2EE

开发标准的种种弊端造就了Spring的出现!即使不是Spring，同样也会有其他类似的产品出现，只不过历史选择了Spring，Spring顺应了历史!没有切肤之痛，是不会体会到Spring带给我们的乐趣与快感!

同样的，每个时代都会有每一个时代的问题，Spring也是!正如十年前我们的计算机可能带不动一款游戏，今天我们的计算机也有可能带不动一款如今的游戏，同样十年后的计算机也会有一款他带不动的游戏出现!以一种发展的眼光去看Spring，就可以很好的理解Spring Boot是以一种什么样的角色出现在我们的面前了!

时代选择了Spring，同样Spring也被这个时代所选择着!你我只有不停的进步，不停地学习才能跟上这个时代!