课程目标

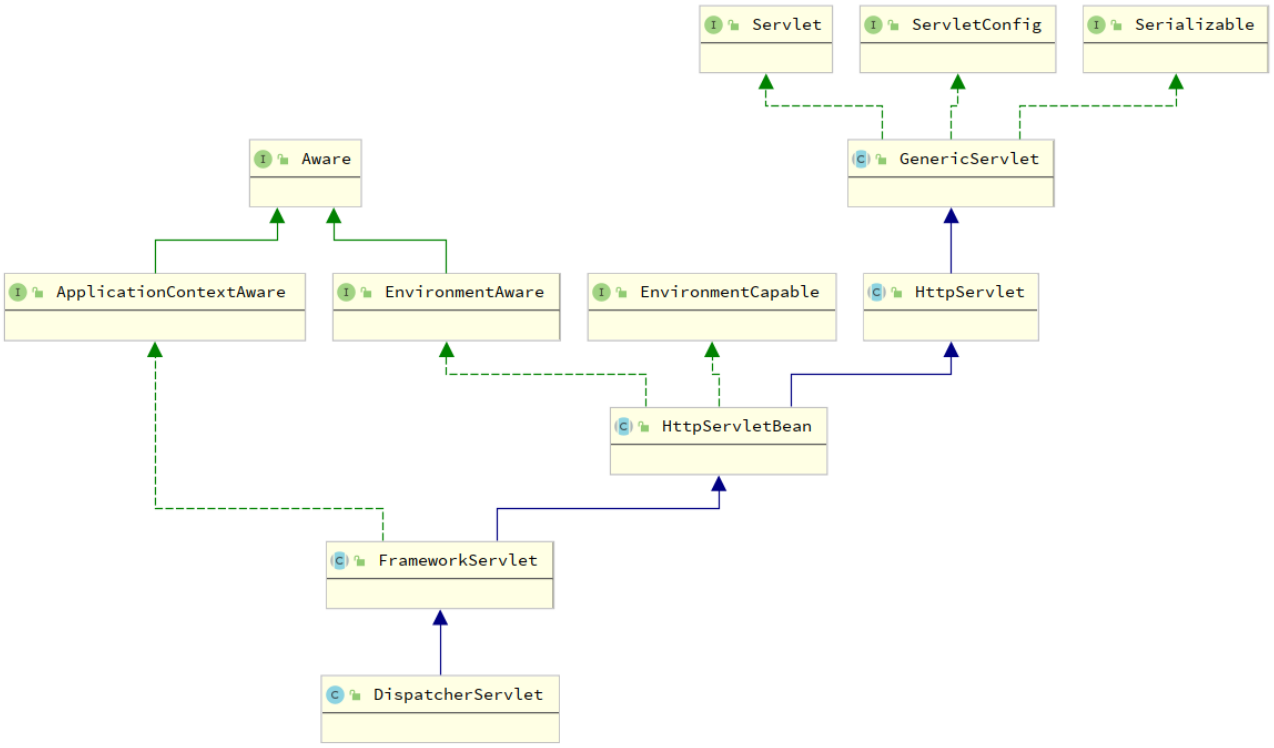
1、高仿真手写SpringIOC和DI部分。
2、用30个类搭建基本框架，满足核心功能。

内容定位

在完全掌握Spring 系统结构、实现原理，在理解设计模式的基础上，自己动手写一个高仿真版本的Spring框架，以达到透彻理解Spring的目的，感受作者创作意图。

1. 从Servlet到ApplicationContext

在300行代码提炼Spring 设计精华的课程中我们已经了解 SpringMVC的入口是DispatcherSerlvet，我们实现了DispatcherServlet的init()方法。在init()方法中完成了IOC容器的初始化。而在我们使用Spring的经验中，我们见得最多的是ApplicationContext，似乎 Spring托管的所有实例Bean都可以通过调用getBean()方法来获得。那么ApplicationContext又是从何而来的呢？从Spring 源码中我们可以看到，DispatcherServlet的类图如下：



DispatcherServlet继承了FrameworkServlet，FrameworkServlet继承了 HttpServletBean，HttpServletBean 继承了HttpServlet。在HttpServletBean的init()方法中调用了FrameworkServlet的initServletBean()方法，在initServletBean()方法中初始化WebApplicationContext实例。在initServletBean()方法中调用了DispatcherServlet 重写的onRefresh()方法。在DispatcherServlet的onRefresh()方法中又调用了initStrategies()方法，初始化SpringMVC的九大组件。

其实，上面复杂的调用关系，我们可以简单的得出一个结论：就是在Servlet的init()方法中初始化了IOC容器和SpringMVC所依赖的九大组件。

2. 项目环境搭建

2.1. application.properties配置

还是先从application.properties文件开始，用application.properties来代替application.xml，具体配置如下：

```
# 托管的类扫描包路径
scanPackage=cn.sitedev.demo
```

2.2. pom.xml配置

接下来看pom.xml的配置，主要关注jar依赖：

```
```

```
    <dependencies>
        <dependency>
            <groupId>javax.servlet</groupId>
            <artifactId>servlet-api</artifactId>
            <version>2.4</version>
        </dependency>

    </dependencies>
```

2.3. web.xml配置

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xmlns="http://java.sun.com/xml/ns/j2ee" xmlns:javaee="http://java.sun.com/xml/
         xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
         xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee http://java.sun.com/xml/ns
         version="2.4">
    <display-name>My Web Application</display-name>
    <servlet>
        <servlet-name>mvc</servlet-name>
        <servlet-class>cn.sitedev.spring.framework.webmvc.servlet.MyDispatcherServlet</
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <param-value>application.properties</param-value>
        </init-param>

        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>mvc</servlet-name>
        <url-pattern>/*</url-pattern>
    </servlet-mapping>
</web-app>
```

2.4. MyDispatcherServlet实现

```
package cn.sitedev.spring.framework.webmvc.servlet;
```

```
2
3  import javax.servlet.ServletException;
4  import javax.servlet.http.HttpServlet;
5  import javax.servlet.http.HttpServletRequest;
6  import javax.servlet.http.HttpServletResponse;
7  import java.io.IOException;
8
9  public class MyDispatcherServlet extends HttpServlet {
10     @Override
11     protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws Servl
12         this.doPost(req, resp);
13     }
14
15     @Override
16     protected void doPost(HttpServletRequest req, HttpServletResponse resp) throws Serv
17     }
18 }
```

3. IOC顶层结构设计

3.1. annotation(自定义配置)模块

Annotation的代码实现我们还是沿用mini版本的不变，复制过来便可。

@MyService 注解

```
1  package cn.sitedev.spring.framework.annotation;
2
3  import java.lang.annotation.*;
4
5  @Target(ElementType.TYPE)
6  @Retention(RetentionPolicy.RUNTIME)
7  @Documented
8  public @interface MyService {
9      String value() default "";
10 }
```

@MyAutowired 注解

```
```

```java
package cn.sitedev.spring.framework.annotation;

import java.lang.annotation.*;

@Target(ElementType.FIELD)
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface MyAutowired {
    String value() default "";
}
```

@MyController 注解

```java
package cn.sitedev.spring.framework.annotation;

import java.lang.annotation.*;

@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface MyController {
    String value() default "";
}
```

@MyRequestMapping 注解

```java
package cn.sitedev.spring.framework.annotation;

import java.lang.annotation.*;

@Target({ElementType.TYPE, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface MyRequestMapping {
    String value() default "";
}
```

@MyRequestParam 注解

```java
package cn.sitedev.spring.framework.annotation;

import java.lang.annotation.*;

@Target(ElementType.PARAMETER)
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface MyRequestParam {
    String value() default "";
}
```

3.2. beans(配置封装)模块

MyBeanDefinition

```java
package cn.sitedev.spring.framework.beans.config;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

@Data
@AllArgsConstructor
@NoArgsConstructor
public class MyBeanDefinition {
    private String beanClassName;
    private String factoryBeanName;
}
```

MyBeanWrapper

```java
package cn.sitedev.spring.framework.beans;

import lombok.Getter;

@Getter
public class MyBeanWrapper {
    private Object wrappedInstance;
```

```
    private Class<?> wrappedClass;

    public MyBeanWrapper(Object wrappedInstance) {
        this.wrappedInstance = wrappedInstance;
        this.wrappedClass = wrappedInstance.getClass();
    }
}
```

3.3. context(IOC容器)模块

MyApplicationContext

```
package cn.sitedev.spring.framework.context;

import cn.sitedev.spring.framework.beans.MyBeanWrapper;
import cn.sitedev.spring.framework.beans.config.MyBeanDefinition;
import cn.sitedev.spring.framework.beans.support.MyBeanDefinitionReader;

import java.util.List;
import java.util.Map;
import java.util.Properties;
import java.util.concurrent.ConcurrentHashMap;

/**
 * 按之前源码分析的套路，IOC，DI，MVC，AOP
 */
public class MyApplicationContext {
    // 存储注册信息的BeanDefinition
    protected final Map<String, MyBeanDefinition> beanDefinitionMap = new ConcurrentHas
    private String[] configLocations;
    private MyBeanDefinitionReader reader;
    // 单例的IOC容器缓存
    private Map<String, Object> factoryBeanObjectCache = new ConcurrentHashMap<>();
    // 通用的IOC容器
    private Map<String, MyBeanWrapper> factoryBeanInstanceCache = new ConcurrentHashMap

    public MyApplicationContext(String... configLocations) {
        this.configLocations = configLocations;
        try {
            // 1. 定位，定位配置文件
            this.reader = new MyBeanDefinitionReader(this.configLocations);
```

```
30
31              // 2. 加载配置文件，扫描相关的类，把它们封装成BeanDefinition
32              List<MyBeanDefinition> beanDefinitions = reader.loadBeanDefinitions();
33
34              // 3. 注册，把配置信息放到容器中(伪IOC容器)
35              doRegisterBeanDefinition(beanDefinitions);
36
37              // 4. 完成自动依赖注入
38              doAutowired();
39
40          } catch (Exception e) {
41              e.printStackTrace();
42          }
43      }
44
45      // 只处理非延时加载的情况
46      private void doAutowired() {
47          for (Map.Entry<String, MyBeanDefinition> beanDefinitionEntry :
48                  this.beanDefinitionMap.entrySet()) {
49              String beanName = beanDefinitionEntry.getKey();
50              try {
51                  getBean(beanName);
52              } catch (Exception e) {
53                  e.printStackTrace();
54              }
55          }
56      }
57
58      private void doRegisterBeanDefinition(List<MyBeanDefinition> beanDefinitions) throw
59          for (MyBeanDefinition beanDefinition : beanDefinitions) {
60              if (this.beanDefinitionMap.containsKey(beanDefinition.getFactoryBeanName())
61                  throw new Exception("The \"" + beanDefinition.getFactoryBeanName() + "\
62                          "exists");
63              }
64              this.beanDefinitionMap.put(beanDefinition.getBeanClassName(), beanDefinitio
65              this.beanDefinitionMap.put(beanDefinition.getFactoryBeanName(), beanDefinit
66          }
67          // 到这里为止，容器初始化完毕
68      }
69
70      private Object getBean(Class<?> beanClass) throws Exception {
71          return getBean(beanClass.getName());
72      }
```

```
73
74      // 依赖注入，从这里开始
75      public Object getBean(String beanName) throws Exception {
76          return null;
77      }
78
79      public String[] getBeanDefinitionNames() {
80          return this.beanDefinitionMap.keySet().toArray(new String[this.beanDefinitionMa
81      }
82
83      public int getBeanDefinitionCount() {
84          return this.beanDefinitionMap.size();
85      }
86
87      public Properties getConfig() {
88          return this.reader.getConfig();
89      }
90  }
```

MyBeanDefinitionReader

```
1  package cn.sitedev.spring.framework.beans.support;
2
3  import cn.sitedev.spring.framework.beans.config.MyBeanDefinition;
4
5  import java.io.File;
6  import java.io.IOException;
7  import java.io.InputStream;
8  import java.net.URL;
9  import java.util.ArrayList;
10 import java.util.List;
11 import java.util.Properties;
12
13 public class MyBeanDefinitionReader {
14     private List<String> registryBeanClasses = new ArrayList<>();
15     private Properties config = new Properties();
16     // 固定配置文件中的key，相对于xml的规范
17     private static final String SCAN_PACKAGE = "scanPackage";
18
19     public MyBeanDefinitionReader(String... locations) {
20         // 通过URL定位找到其所对应的文件，然后转换为文件流
```

```
        InputStream inputStream =
                this.getClass().getClassLoader().getResourceAsStream(locations[0].repla
                        "classpath:", ""));
        try {
            config.load(inputStream);
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            if (inputStream != null) {
                try {
                    inputStream.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
        // 扫描指定包下的文件/文件夹
        doScanner(config.getProperty(SCAN_PACKAGE));
    }

    private void doScanner(String scanPackage) {
        // 转换为文件路径，实际上就是把.替换为/
        URL url = this.getClass().getClassLoader().getResource("/" + scanPackage.replac
                , "/"));
        File classPath = new File(url.getFile());
        for (File file : classPath.listFiles()) {
            if (file.isDirectory()) {
                doScanner(scanPackage + "." + file.getName());
            } else {
                if (!file.getName().endsWith(".class")) {
                    continue;
                }
                String className = (scanPackage + "." + file.getName()).replace(".class
                registryBeanClasses.add(className);
            }
        }
    }

    public Properties getConfig() {
        return this.config;
    }

    // 把配置文件中扫描到的所有的配置信息转换成MyBeanDefinition对象,以便之后IOC操作方便
```

```
64      public List<MyBeanDefinition> loadBeanDefinitions() {
65          List<MyBeanDefinition> result = new ArrayList<>();
66          try {
67              for (String className : registryBeanClasses) {
68                  Class<?> beanClass = Class.forName(className);
69                  // 如果是一个接口，是不能被实例化的
70                  // 用它的实现类来实例化
71                  if (beanClass.isInterface()) {
72                      continue;
73                  }
74
75                  // beanName有三种情况：
76                  // 1. 默认是类名首字母小写
77                  // 2. 自定义名字
78                  // 3. 接口注入
79                  result.add(doCreateBeanDefinition(toLowerFirstCase(beanClass.getSimpleN
80                          beanClass.getName()));
81
82                  Class<?>[] interfaces = beanClass.getInterfaces();
83                  for (Class<?> i : interfaces) {
84                      // 如果是多个实现类，只能覆盖
85                      // 为什么？因为Spring没有那么智能
86                      // 这个时候，可以自定义名字
87                      result.add(doCreateBeanDefinition(i.getName(), beanClass.getName())
88                  }
89              }
90          } catch (ClassNotFoundException e) {
91              e.printStackTrace();
92          }
93          return result;
94      }
95
96      // 把每一个配置信息解析成一个BeanDefinition
97      private MyBeanDefinition doCreateBeanDefinition(String factoryBeanName, String bear
98          MyBeanDefinition beanDefinition = new MyBeanDefinition(beanClassName, factoryBe
99          return beanDefinition;
100     }
101
102     // 如果类名本身是小写字母，确实会出问题
103     // 但是要说明的是，这个方法是自己用的，是private类型的
104     // 传值也是自己传，类也遵循了驼峰命名法
105     // 默认传入的值，存在首字母小写的情况，但不可能出现非字母的情况
106     private String toLowerFirstCase(String simpleName) {
```

```
        char[] chars = simpleName.toCharArray();
        // 之所以加，是因为大小写字母的ASCII码相差32
        // 而且大写字母的ASCII码要小于小写字母的ASCII码
        // 在Java中,对char做数学运算，实际上就是对ASCII码做数学运算
        chars[0] += 32;
        return String.valueOf(chars);
    }
}
```

4. 完成DI依赖注入功能

在之前的源码分析中，我们已经了解到，依赖注入的入口是从getBean()方法开始的，前面的IOC手写部分基本流程已通。先在MyApplicationContext中定义好IOC容器，一个是MyBeanWrapper，一个是单例对象缓存

```
package cn.sitedev.spring.framework.context;

import cn.sitedev.spring.framework.beans.MyBeanWrapper;
import cn.sitedev.spring.framework.beans.config.MyBeanDefinition;
import cn.sitedev.spring.framework.beans.support.MyBeanDefinitionReader;

import java.util.List;
import java.util.Map;
import java.util.Properties;
import java.util.concurrent.ConcurrentHashMap;

/**
 * 按之前源码分析的套路，IOC，DI，MVC，AOP
 */
public class MyApplicationContext {
    // 存储注册信息的BeanDefinition
    protected final Map<String, MyBeanDefinition> beanDefinitionMap = new ConcurrentHas
    private String[] configLocations;
    private MyBeanDefinitionReader reader;
    // 单例的IOC容器缓存
    private Map<String, Object> factoryBeanObjectCache = new ConcurrentHashMap<>();
    // 通用的IOC容器
    private Map<String, MyBeanWrapper> factoryBeanInstanceCache = new ConcurrentHashMap
    ...
```

4.1. 从getBean()开始

下面，我们从完善 getBean()方法开始：

```
    private Object getBean(Class<?> beanClass) throws Exception {
        return getBean(beanClass.getName());
    }

    // 依赖注入，从这里开始，通过读取BeanDefinition中的信息
    // 然后，通过反射机制创建一个实例并返回
    // Spring的做法是，不会把最原始的对象放进去，会用一个BeanWrapper来进行一次包装
    // 装饰器模式：
    // 1. 保留原来的OOP关系
    // 2. 我们需要对它进行扩展，增强(为了以后AOP打基础)
    public Object getBean(String beanName) throws Exception {
        // 1. 读取配置信息
        MyBeanDefinition beanDefinition = this.beanDefinitionMap.get(beanName);

        // 2. 实例化
        Object instance = instantiateBean(beanName, beanDefinition);

        // 3. 把这个对象封装到BeanWrapper中
        MyBeanWrapper beanWrapper = new MyBeanWrapper(instance);

        // 4. 把BeanWrapper存到IOC容器中
        // class A { B b; }
        // class B { A a; }
        // 先有鸡还是先有蛋的问题，一个方法是搞不定的，要分两次
        // 5. 拿到BeanWrapper之后，把BeanWrapper保存到IOC容器中去
        this.factoryBeanInstanceCache.put(beanName, beanWrapper);

        // 6. 执行依赖注入
        populateBean(beanName, new MyBeanDefinition(), beanWrapper);

        return this.factoryBeanInstanceCache.get(beanName).getWrappedInstance();

    }

    private void populateBean(String beanName, MyBeanDefinition beanDefinition,
                              MyBeanWrapper beanWrapper) {
        Object instance = beanWrapper.getWrappedInstance();

```

```
        Class<?> clazz = beanWrapper.getWrappedClass();
        // 判断只有加了注解的类，才执行依赖注入
        if (!(clazz.isAnnotationPresent(MyController.class) || clazz.isAnnotationPresen
            return;
        }

        // 获得所有的fields
        Field[] fields = clazz.getDeclaredFields();
        for (Field field : fields) {
            if (!field.isAnnotationPresent(MyAutowired.class)) {
                continue;
            }

            MyAutowired autowired = field.getAnnotation(MyAutowired.class);

            String autowiredBeanName = autowired.value().trim();

            if ("".equals(autowiredBeanName)) {
                autowiredBeanName = field.getType().getName();
            }

            // 强制访问
            field.setAccessible(true);

            try {
                if (this.factoryBeanInstanceCache.get(autowiredBeanName) == null) {
                    continue;
                }
                field.set(instance,
                        this.factoryBeanInstanceCache.get(autowiredBeanName).getWrapped
            } catch (IllegalAccessException e) {
                e.printStackTrace();
            }
        }

    }

    private Object instantiateBean(String beanName, MyBeanDefinition beanDefinition) {
        // 1. 拿到要实例化的对象的类名
        String className = beanDefinition.getBeanClassName();

        // 2. 反射实例化，得到一个对象
        Object instance = null;
```

```
82          try {
83              // 假设默认就是单例，细节暂且不考虑，先把主线拉通
84              if (this.factoryBeanObjectCache.containsKey(className)) {
85                  instance = this.factoryBeanObjectCache.get(className);
86              } else {
87                  Class<?> clazz = Class.forName(className);
88                  instance = clazz.newInstance();
89                  this.factoryBeanObjectCache.put(className, instance);
90                  this.factoryBeanObjectCache.put(beanDefinition.getFactoryBeanName(), in
91              }
92          } catch (Exception e) {
93              e.printStackTrace();
94          }
95          return instance;
96
97      }
```

至此，DI部分就完成了。