链滴

手写 Web 框架之实现 IOC/DI

作者：z875479694h
原文链接：https://ld246.com/article/1625939132749
来源网站：链滴

介绍

尝试写一个自己的Web框架，先模仿，再探索走出自己的道路，最终超越。框架用的多了，但是写框是第一次，而且模仿的还是Java生态中的基石——Spring。先了解其思路，一步一步去实现，或许性不如正主，但是可以一点点靠近。学习然后模仿是通往大牛道路的第一步。

一.注解

1. Autowired注解是自动注入，作用是将实例注入到带有@Autowired注解的变量中。
2. Component注解是组件，作用是框架会扫描所有带@Component的class并进行实例化。
3. ComponentSacn注解是扫描路径，作用是设置框架要扫描的包路径。
4. Scope注解是单例/原型标识，作用是不加默认是单例Bean，加了@Scope("prototype")后将是原Bean（多例模式）。

1.1 Autowired

```
package xyz.hcworld.jubilant.annotation;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

/**
 * @ClassName: Autowired
 * @Author: 张红尘
 * @Date: 2021-06-15
 * @Version:  1.0
```

```
 */
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.METHOD, ElementType.FIELD})
@Documented
public @interface Autowired {

    /**
     * @return 组件名
     */
    String value() default "";

}
```

1.2 Component

```
package xyz.hcworld.jubilant.annotation;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

/**
 * 注册Bean组件
 *
 * @ClassName: Component
 * @Author: 张红尘
 * @Date: 2021-06-13
 * @Version:  1.0
 */
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
@Documented
public @interface Component {
    /**
     * @return 组件名
     */
    String value() default "";

}
```

1.3 ComponentSacn

```
package xyz.hcworld.jubilant.annotation;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

/**
 * 扫描配置文件注解
 *
```

```
 * @ClassName: ComponentSacn
 * @Author: 张红尘
 * @Date: 2021-06-13
 * @Version:  1.0
 */
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
@Documented
public @interface ComponentScan {
    /**
     * @return 扫描路径
     */
    String value() default "";
}
```

1.4 Scope

```
package xyz.hcworld.jubilant.annotation;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

/**
 * 单例Bean/原型Bean标识
 * @ClassName: Scope
 * @Author: 张红尘
 * @Date: 2021-06-15
 * @Version:  1.0
 */
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
@Documented
public @interface Scope {

    String value();
}
```

二、Bean操作

1. BeanDefinition类是建模对象，通过建模对象创建实例。

2. BeanNameAware接口是bean名的Aware回调接口，继承接口可以获取bean的benaName。

3. BeanPostProcessor接口是初始化前后的增强接口，继承接口可以对bean进行动态增强，可以使用DK增强或者CGLib增强。（可以通过这个接口实现AOP）。

4. InitializingBean接口是应用级的初始化接口，继承该接口后框架实例化Bean后会对Bean进行初始操作。

2.1 BeanDefinition

```
package xyz.hcworld.jubilant.beans;
```

```java
/**
 * bean的配置信息
 * @ClassName: BeanDefinition
 * @Author: 张红尘
 * @Date: 2021-06-15
 * @Version:  1.0
 */
public class BeanDefinition {
    /**
     * 类的类型
     */
    private Class<?> clazz;
    /**
     * 作用域 是否是懒加载
     */
    private String scope;


    public Class<?> getClazz() {
        return clazz;
    }

    public void setClazz(Class<?> clazz) {
        this.clazz = clazz;
    }

    public String getScope() {
        return scope;
    }

    public void setScope(String scope) {
        this.scope = scope;
    }
}
```

2.2 BeanNameAware

```java
package xyz.hcworld.jubilant.beans;

/**
 * bean名的Aware回调接口
 * @ClassName: BeanNameAware
 * @Author: 张红尘
 * @Date: 2021-06-16
 * @Version:  1.0
 */
public interface BeanNameAware {
    /**
     * 名字
     * @param name
     */
    void setBeanName(String name);
```

```
}
```

2.3 BeanPostProcessor

```
/**
 * 后置处理器（bean初始化前后的增强处理）
 *
 * @ClassName: BeanPostProcessor
 * @Author: 张红尘
 * @Date: 2021-06-16
 * @Version:  1.0
 */
public interface BeanPostProcessor {
    /**
     * bean初始化前调用
     *
     * @param bean     bean本身
     * @param beanName bean的名字
     * @return
     */
    default Object postProcessBeforeInitialization(Object bean, String beanName) {
        return bean;
    }

    /**
     * bean初始化后调用
     *
     * @param bean     bean本身
     * @param beanName bean的名字
     * @return
     */
    default Object postProcessAfterInitialization(Object bean, String beanName) {
        return bean;
    }
}
```

2.4 InitializingBean

```
package xyz.hcworld.jubilant.beans;

/**
 * 初始化机制
 * @ClassName: Initializingbean
 * @Author: 张红尘
 * @Date: 2021-06-16
 * @Version:  1.0
 */
public interface InitializingBean {
    /**
     * 初始化
     * @throws Exception
     */
    void afterPropertiesSet() throws Exception;
```

```
}
```

三.容器类

1. Search类是搜索class的类，分别对jar与非jar两种情况的class进行扫描。

2. JubilantApplicationContext是核心容器类，ioc是整个流程都在当前容器进行。流程如下：获取有class的路径->扫描所有带@Component的class->生成所有带@Component的class的BeanDefinition对象-->存入BeanDefinitionMap-->遍历BeanDefinitionMap生成所有单例Bean存入单例池。

3.1 扫描class模块

```
package xyz.hcworld.jubilant.core;

import java.io.File;
import java.io.IOException;
import java.net.URL;

import java.util.ArrayList;
import java.util.Enumeration;
import java.util.List;
import java.util.jar.JarEntry;
import java.util.jar.JarFile;

/**
 * @ClassName: Search
 * @Author: 张红尘
 * @Date: 2021-06-18
 * @Version:  1.0
 */
public class Search {


    /**
     * 获取某包下（包括该包的所有子包）所有类
     *
     * @param packageName 包名
     * @return 类的完整名称
     */
    List<String> getClassName(String packageName) {
        ClassLoader loader = Thread.currentThread().getContextClassLoader();
        URL url = loader.getResource(packageName.replace(".", "/"));
        if (url == null) {
            return new ArrayList<>();
        }
        return "file".equals(url.getProtocol())?
                getClassNameByFile(url.getPath(), null, packageName.replace(".", System.getPropert
("file.separator")))
                :getClassNameByJar(url.getPath());
    }

    /**
     * 从项目文件获取某包下所有类
```

```
     *
     * @param filePath  文件路径
     * @param className 类名集合
     * @return 类的完整名称
     */
    private static List<String> getClassNameByFile(String filePath, List<String> className, Stri
g packageName) {
        List<String> myClassName = new ArrayList<>();
        File file = new File(filePath);
        File[] childFiles = file.listFiles();
        if (childFiles == null) {
            return myClassName;
        }
        for (File childFile : childFiles) {
            if (childFile.isDirectory()) {
                myClassName.addAll(getClassNameByFile(childFile.getPath(), myClassName, packa
eName));
                continue;
            }
            String childFilePath = childFile.getPath();
            if (childFilePath.endsWith(".class")) {
                //截取路径
                childFilePath = childFilePath.substring(childFilePath.indexOf(packageName), childFil
Path.lastIndexOf("."));
                //将反斜杠转成.
                myClassName.add(childFilePath.replace(System.getProperty("file.separator"), "."));
            }
        }
        return myClassName;
    }

    /**
     * 从jar获取某包下所有类
     *
     * @param jarPath jar文件路径
     * @return 类的完整名称
     */
    private static List<String> getClassNameByJar(String jarPath) {
        List<String> myClassName = new ArrayList<>();
        try (JarFile jarFile = new JarFile(System.getProperty("user.dir") + System.getProperty("file.
eparator") + System.getProperty("java.class.path"))) {
            Enumeration<JarEntry> entries = jarFile.entries();
            while (entries.hasMoreElements()) {
                JarEntry jarEntry = entries.nextElement();
                String entryName = jarEntry.getName();
                if (entryName.startsWith(jarPath) && entryName.endsWith(".class")) {
                    entryName = entryName.replace("/", ".").substring(0, entryName.lastIndexOf(".cla
s"));
                    myClassName.add(entryName);
                }
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
```

```
        return myClassName;
    }
}
```

3.2 容器化

```
package xyz.hcworld.jubilant.core;

import xyz.hcworld.jubilant.beans.BeanDefinition;
import xyz.hcworld.jubilant.beans.BeanNameAware;
import xyz.hcworld.jubilant.beans.BeanPostProcessor;
import xyz.hcworld.jubilant.beans.InitializingBean;
import xyz.hcworld.jubilant.annotation.Autowired;
import xyz.hcworld.jubilant.annotation.Component;
import xyz.hcworld.jubilant.annotation.ComponentScan;
import xyz.hcworld.jubilant.annotation.Scope;

import java.io.File;
import java.lang.reflect.Field;
import java.lang.reflect.InvocationTargetException;
import java.util.*;
import java.util.concurrent.ConcurrentHashMap;

/**
 * 容器类
 *
 * @ClassName: WinterApplicationContext
 * @Author: 张红尘
 * @Date: 2021-06-13
 * @Version:  1.0
 */
public class JubilantApplicationContext {
    /**
     * 配置文件
     */
    private Class configClass;

    /**
     * 单例池，保存单例对象
     */
    private final ConcurrentHashMap<String, Object> singletonObjects = new ConcurrentHas
Map<>();
    /**
     * 所有Bean的配置数据的配置池
     */
    private final ConcurrentHashMap<String, BeanDefinition> beanDefinitionMap = new Conc
rrentHashMap<>();
    /**
     * 初始化bean的类
     */
    private final List<BeanPostProcessor> beanPostProcessorList = new ArrayList<>();

    /**
     * 构造方法
```

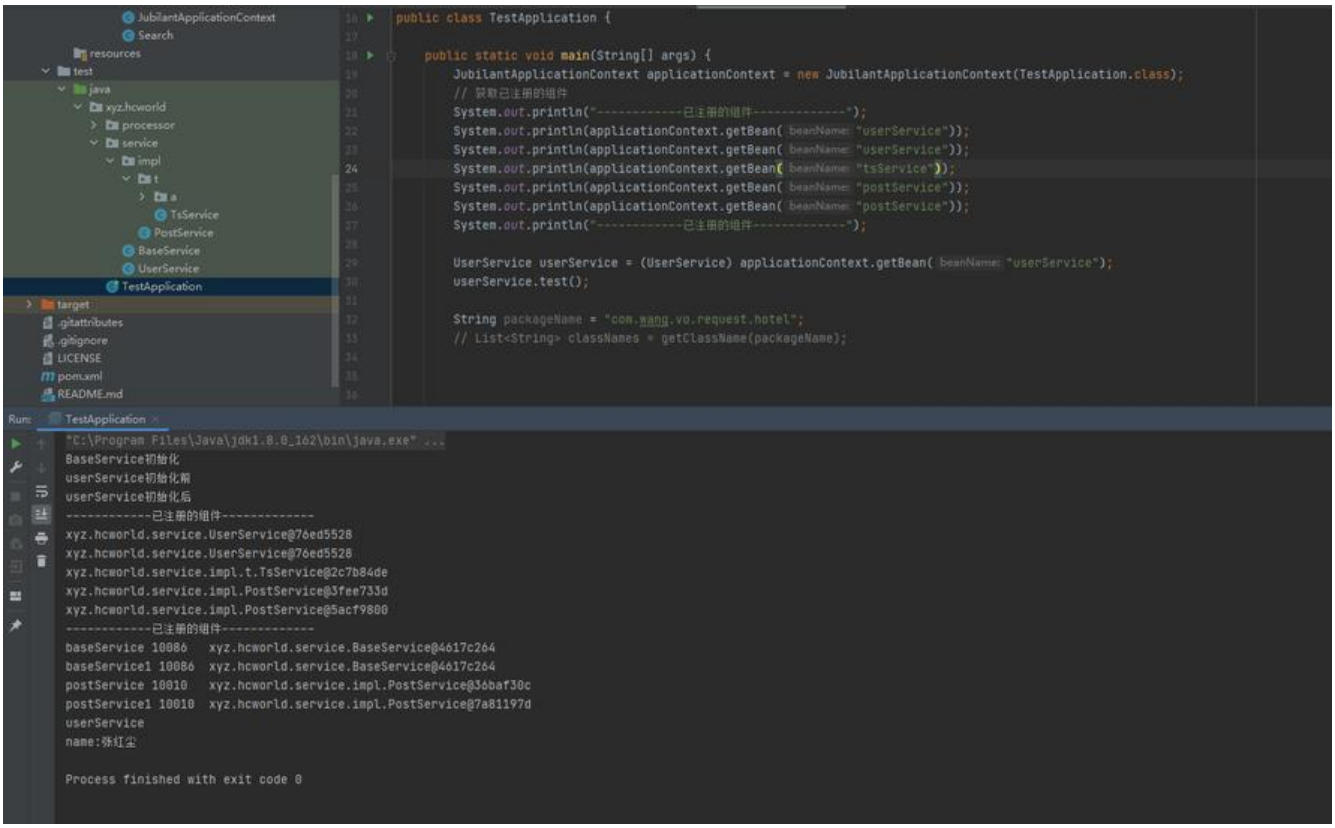
```
     *
     * @param configClass 配置类
     */
    public JubilantApplicationContext(Class<?> configClass) {
        this.configClass = configClass;
    // 判断是否有ComponentScan注解，如果没有就结束不往下走
    if (!configClass.isAnnotationPresent(ComponentScan.class)) {
            return;
    }
    // 解析配置类
    // 对ComponentScan注解解析 -->扫描路径 -->扫描-->生成BeanDefinition对象-->存入BeanD
finitionMap
    scan(configClass);
        // Aspect类先实例化
    for (Map.Entry<String, BeanDefinition> entry : beanDefinitionMap.entrySet()) {
            if (!entry.getValue().getClazz().isAnnotationPresent(Aspect.class)) {
                continue;
            }
            Object bean = createBean(entry.getKey(), entry.getValue());
            singletonObjects.put(entry.getKey(), bean);
    }

        for (Map.Entry<String, BeanDefinition> entry : beanDefinitionMap.entrySet()) {
            // 创建所有的单例bean，已经实例化的就不需要实例化了
            if ("singleton".equals(entry.getValue().getScope()) && !singletonObjects.containsKey(e
try.getKey())) {
                Object bean = createBean(entry.getKey(), entry.getValue());
                singletonObjects.put(entry.getKey(), bean);
            }
        }

    }

    /**
     * 扫描
     *
     * @param configClass
     */
    private void scan(Class<?> configClass) {

        ComponentScan componentScanAnnotation = configClass.getDeclaredAnnotation(Com
onentScan.class);
        // 扫描路径(不填路径默认获取Application文件所在的包) xyz.hcworld
        String path = componentScanAnnotation.value().isEmpty() ? configClass.getPackage().ge
Name() : componentScanAnnotation.value();
        Search search = new Search();
        //获取文件夹或者jar下的所有类名
        List<String> classNames = search.getClassName(path);
        if (classNames == null) {
            return;
        }
        ClassLoader classLoader = JubilantApplicationContext.class.getClassLoader();
        try {
            for (String className : classNames) {
```

```
                Class<?> clazz = classLoader.loadClass(className);
                // 如果没有Component代表这不是一个bean结束本次循环
                if (!clazz.isAnnotationPresent(Component.class)) {
                    continue;
                }
                // 创建bean对象
                // 解析类，判断当前Bean是单例还是原型（prototype）Bean 生成BeanDefinition对象

                // 统一处理方式：BeanDefinition
                Component componentAnnotation = clazz.getDeclaredAnnotation(Component.clas
);
                // 获取Bean的名字
                String beanName = componentAnnotation.value();
                //如果没有自定义beanName则以类名（首字母小写）为beanName
                if (beanName.isEmpty()) {
                    String[] name = className.split("\\.");
                    StringBuilder nameBuffer = new StringBuilder(name[name.length - 1]);
                    beanName = nameBuffer.replace(0, 1, Character.toString(nameBuffer.charAt(0)).t
LowerCase()).toString();
                }

                // bean的配置信息
                BeanDefinition beanDefinition = new BeanDefinition();
                beanDefinition.setClazz(clazz);
                // 判断Scope是否存在,存在则是自定义配置，不存在则为单例
                beanDefinition.setScope(
                        clazz.isAnnotationPresent(Scope.class) ?
                                clazz.getDeclaredAnnotation(Scope.class).value() : "singleton");
                // 存进配置池
                beanDefinitionMap.put(beanName, beanDefinition);
                // 将实现BeanPostProcessor（初始化bean）接口的类存到初始化配置池中
                if (BeanPostProcessor.class.isAssignableFrom(clazz)) {
                    BeanPostProcessor beanPostProcessor = (BeanPostProcessor) createBean(beanN
me, beanDefinitionMap.get(beanName));
                    beanPostProcessorList.add(beanPostProcessor);
                }

            }
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
    }

    /**
     * 根据配置创建出一个bean对象（反射）
     *
     * @param beanDefinition 自定义配置
     * @return bean对象
     */
    public Object createBean(String beanName, BeanDefinition beanDefinition) {

        Class<?> clazz = beanDefinition.getClazz();
        try {
            Object instance = clazz.getDeclaredConstructor().newInstance();
```

```
            // 依赖注入
            for (Field declaredField : clazz.getDeclaredFields()) {
                // 没有Autowired就不注入
                if (!declaredField.isAnnotationPresent(Autowired.class)) {
                    continue;
                }
                String[] name = declaredField.getType().getName().split("\\.");
                StringBuilder nameBuffer = new StringBuilder(name[name.length - 1]);
                nameBuffer.replace(0, 1, Character.toString(nameBuffer.charAt(0)).toLowerCase());
                // 通过反射注入属性
                Object bean = getBean(nameBuffer.toString());
                if (bean == null) {
                    throw new NullPointerException();
                }
                // 当变量为private时需要忽略访问修饰符的检查
                declaredField.setAccessible(true);
                declaredField.set(instance, bean);
            }
            // Aware回调
            if (instance instanceof BeanNameAware) {
                ((BeanNameAware) instance).setBeanName(beanName);
            }
            //初始化前的增强
            for (BeanPostProcessor beanPostProcessor : beanPostProcessorList) {
                instance = beanPostProcessor.postProcessBeforeInitialization(instance, beanName);
            }

            // 初始化
            if (instance instanceof InitializingBean) {
                ((InitializingBean) instance).afterPropertiesSet();
            }
            // BeanPostProcessor 外部扩展机制（Bean的前后置处理）
            //初始化后的增强
            for (BeanPostProcessor beanPostProcessor : beanPostProcessorList) {
                instance = beanPostProcessor.postProcessAfterInitialization(instance, beanName);
            }

            return instance;
        } catch (IllegalAccessException | InvocationTargetException | NoSuchMethodException e)

            e.printStackTrace();
        } catch (Exception e) {
            e.printStackTrace();
        }
        return null;
    }


    /**
     * 获取Bean对象
     *
     * @param beanName bean名字
     * @return bean对象
     */
```

```
    public Object getBean(String beanName) {
        // 判断bean是否存在,不存在抛出npe异常
        if (!beanDefinitionMap.containsKey(beanName)) {
            throw new NullPointerException();
        }
        BeanDefinition beanDefinition = beanDefinitionMap.get(beanName);
        // 判断是否是单例,是返回单例对象，不是创建原型对象
        return "singleton".equals(beanDefinition.getScope()) ?
                singletonObjects.get(beanName) : createBean(beanName, beanDefinition);
    }

    /**
     * 获取文件制定包下的所有class文件（不管有多少层）
     *
     * @param file      目录
     * @param filePaths 临时存储
     * @return 所有class文件的绝对路径
     */
    private Set<File> getFilePath(File file, Set<File> filePaths) {
        File[] files = file.listFiles();
        if (files == null || files.length == 0) {
            return new HashSet<>();
        }
        for (File file1 : files) {
            if (file1.isDirectory()) {
                //递归调用
                getFilePath(file1, filePaths);
                continue;
            }
            //保存文件路径到集合中
            filePaths.add(file1);
        }
        return filePaths;
    }
}
```

四、结果



未来发展

展望

* 第一步，通过模仿Spring实现IOC
* 第二步，通过模仿Spring实现AOP
* 第三步，实现MVC框架

说明

ioc 的代码是跟着B站的某教程敲的。但是修改了许多不合理的部分，如扫描class部分，原教程打包无法运行，因为获取不了绝对路径。修改了既能打包jar后运行又能在ide中运行。以及原本丧心病狂的i-else-for嵌套改成了防御性的if判断，去除了else。

先模仿在摸索自己的道路。doge

不足

笔者水平有限也非大厂的大牛，如有不对之处请指正。以及部分硬编码问题纯粹是笔者懒，还没来得改，自行修改即可。

地址

GitHub:https://github.com/z875479694h/jubilant

Gitee:https://gitee.com/hcworld/jubilant