1 Spring 框架核心源码

1、使用 Spring 框架

2、反射机制

IoC 控制反转 Inverse of Control 创建对象的权限，Java 程序中需要用到的对象不再由程序员自己创建，而是交给 IoC 容器来创建。

1.1 IoC 核心思想

1、pom.xml

```
<dependencies>
    <!-- 引入 Servlet 依赖 -->
    <dependency>
        <groupId>javax.servlet</groupId>
        <artifactId>javax.servlet-
api</artifactId>
        <version>4.0.1</version>
    </dependency>
</dependencies>

<!-- 设置 Maven 的JDK版本，默认是5，需要手动改到8
-->
<build>
    <plugins>
```

```
        <plugin>

 <groupId>org.apache.maven.plugins</groupId
>
            <artifactId>maven-compiler-
plugin</artifactId>
            <version>3.8.1</version>
            <configuration>
                <source>1.8</source>
                <target>1.8</target>
                <encoding>UTF-8</encoding>
            </configuration>
        </plugin>
    </plugins>
</build>

<packaging>war</packaging>
```

2、创建 Servlet

```
package com.southwind.servlet;

import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import
javax.servlet.http.HttpServletRequest;
import
javax.servlet.http.HttpServletResponse;
import java.io.IOException;
```

```
@WebServlet("/hello")
public class HelloServlet extends
HttpServlet {

    @Override
    protected void doGet(HttpServletRequest
req, HttpServletResponse resp) throws
ServletException, IOException {
        resp.getWriter().write("Spring");
    }
}
```

3、部署到 Tomcat

4、Servlet、Service、Dao

当需求发生变更的时候，可能需要频繁修改 Java 代码，效率很低，如何解决?

静态工厂

```
package com.southwind.factory;

import com.southwind.dao.HelloDao;
import com.southwind.dao.impl.HelloDaoImpl;

public class BeanFactory {
    public static HelloDao getDao(){
        return new HelloDaoImpl();
    }
}
```

```
private HelloDao helloDao =
BeanFactory.getDao();
```

上述的方式并不能解决我们的问题，需求发生改变的时候，仍然需要修改代码，怎么做到

不改 Java 代码，就可以实现实现类的切换呢?

外部配置文件的方式

将具体的实现类写到配置文件中，Java 程序只需要读取配置文件即可。

XML、YAML、Properties、JSON

1、定义外部配置文件

```
helloDao=com.southwind.dao.impl.HelloDaoImp
l
```

2、Java 程序读取这个配置文件

```
package com.southwind.factory;

import com.southwind.dao.HelloDao;
import com.southwind.dao.impl.HelloDaoImpl;
import
com.southwind.dao.impl.HelloDaoImpl2;

import java.io.IOException;
```

```
import
java.lang.reflect.InvocationTargetException
;
import java.util.Properties;

public class BeanFactory {

    private static Properties properties;

    static {
        properties = new Properties();
        try {

 properties.load(BeanFactory.class.getClass
Loader().getResourceAsStream("factory.prope
rties"));
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    public static Object getDao(){
        String value =
properties.getProperty("helloDao");
        //反射机制创建对象
        try {
            Class clazz =
Class.forName(value);
```

```
            Object object =
clazz.getConstructor(null).newInstance(null
);
            return object;
        } catch (ClassNotFoundException e)
{
            e.printStackTrace();
        } catch (InstantiationException e)
{
            e.printStackTrace();
        } catch (IllegalAccessException e)
{
            e.printStackTrace();
        } catch (InvocationTargetException
e) {
            e.printStackTrace();
        } catch (NoSuchMethodException e) {
            e.printStackTrace();
        }
        return null;
    }
}
```

3、修改 Service

```
private HelloDao helloDao = (HelloDao)
BeanFactory.getDao();
```

Spring IoC 中的 bean 是单例

```
package com.southwind.factory;
```

```
import com.southwind.dao.HelloDao;
import com.southwind.dao.impl.HelloDaoImpl;
import
com.southwind.dao.impl.HelloDaoImpl2;

import java.io.IOException;
import
java.lang.reflect.InvocationTargetException
;
import java.util.HashMap;
import java.util.Map;
import java.util.Properties;

public class BeanFactory {

    private static Properties properties;
    private static Map<String,Object> cache
= new HashMap<>();

    static {
        properties = new Properties();
        try {

 properties.load(BeanFactory.class.getClass
Loader().getResourceAsStream("factory.prope
rties"));
        } catch (IOException e) {
            e.printStackTrace();
        }
```

```
    }

    public static Object getDao(String
beanName){
        //先判断缓存中是否存在bean
        if(!cache.containsKey(beanName)){
            synchronized
(BeanFactory.class){

 if(!cache.containsKey(beanName)){
                    //将bean存入缓存
                    //反射机制创建对象
                    try {
                        String value =
properties.getProperty(beanName);
                        Class clazz =
Class.forName(value);
                        Object object =
clazz.getConstructor(null).newInstance(null
);
                        cache.put(beanName,
object);
                    } catch
(ClassNotFoundException e) {

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

 e.printStackTrace();
```

```
                    } catch
(IllegalAccessException e) {

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

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

 e.printStackTrace();
                    }
                }
            }
        }
        return cache.get(beanName);
    }
}
```

```
1、private HelloDao helloDao = new
HelloDaoImpl();

2、private HelloDao helloDao =  (HelloDao)
BeanFactory.getDao("helloDao");
```

1、强依赖/紧耦合，编译之后无法修改，没有扩展性。

2、弱依赖/松耦合，编译之后仍然可以修改，让程序具有更好的扩展性。

自己放弃了创建对象的权限，将创建对象的权限交给了BeanFactory，这种将控制权交给别人的思想，就是控制反转 IoC。

1.2 Spring IoC 的使用

XML 和注解，XML 已经被淘汰了，目前主流的是基于注解的方式，Spring Boot 就是基于注解的方式。

```
package com.southwind.spring.entity;

import lombok.Data;
import
org.springframework.beans.factory.annotatio
n.Value;
import
org.springframework.stereotype.Component;

@Data
@Component("myOrder")
public class Order {
    @Value("xxx123")
    private String orderId;
    @Value("1000.0")
    private Float price;
}
```

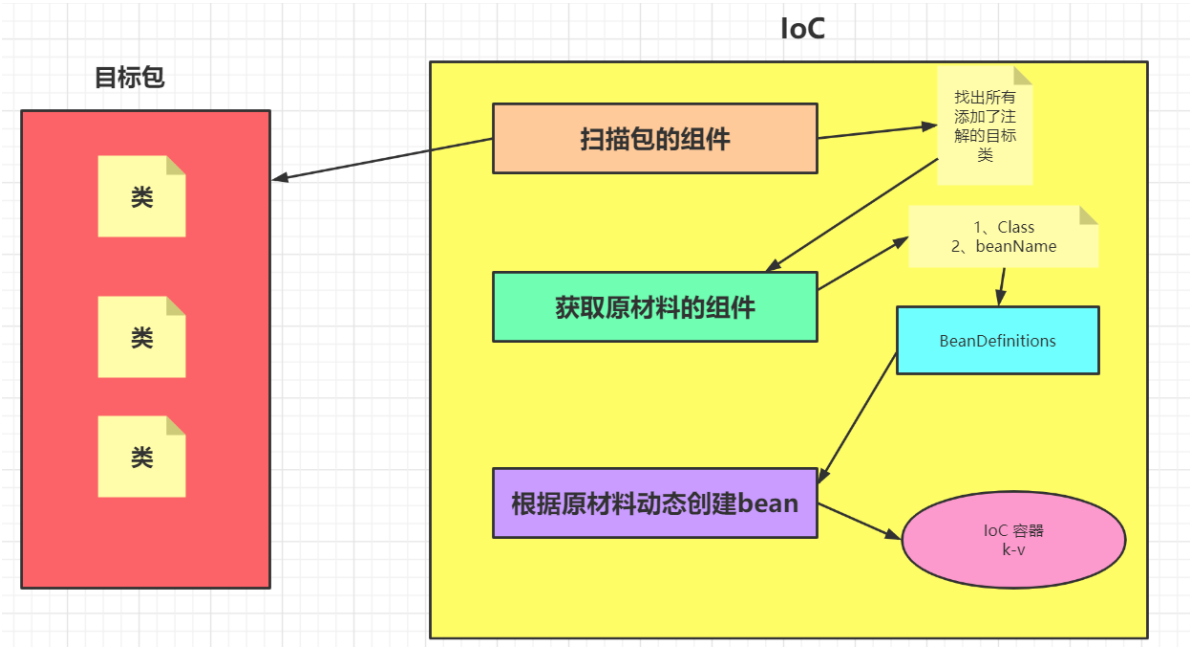
```java
package com.southwind.spring.entity;

import lombok.Data;
import
org.springframework.beans.factory.annotatio
n.Autowired;
import
org.springframework.beans.factory.annotatio
n.Qualifier;
import
org.springframework.beans.factory.annotatio
n.Value;
import
org.springframework.stereotype.Component;

@Data
@Component
public class Account {
    @Value("1")
    private Integer id;
    @Value("张三")
    private String name;
    @Value("22")
    private Integer age;
    @Autowired
    @Qualifier("order")
    private Order myOrder;
}
```

```java
package com.southwind.spring.test;
```

```
import
org.springframework.context.ApplicationCont
ext;
import
org.springframework.context.annotation.Anno
tationConfigApplicationContext;
import
org.springframework.context.support.ClassPa
thXmlApplicationContext;

public class Test {
    public static void main(String[] args)
{
        //加载IoC容器
        ApplicationContext
applicationContext = new
AnnotationConfigApplicationContext("com.sou
thwind.spring.entity");
        String[] beanDefinitionNames =
applicationContext.getBeanDefinitionNames()
;

 System.out.println(applicationContext.getB
eanDefinitionCount());
        for (String beanDefinitionName :
beanDefinitionNames) {

 System.out.println(beanDefinitionName);
```

```
 System.out.println(applicationContext.getB
ean(beanDefinitionName));
            }
        }
}
```

1.3 IoC 基于注解的执行原理



手写代码的思路：

1、自定义一个MyAnnotationConfigApplicationContext，构造器中传入要扫描的包。

2、获取这个包下的所有类。

3、遍历这些类，找出添加了 @Component 注解的类，获取它的 Class 和对应的 beanName，封装成一个 BeanDefinition，存入集合 Set，这个机会就是 IoC 自动装载的原材料。

4、遍历 Set 集合，通过反射机制创建对象，同时检测属性有没有添加 @Value 注解，如果有还需要给属性赋值，再将这些动态创建的对象以 k-v 的形式存入缓存区。

5、提供 getBean 等方法，通过 beanName 取出对应的 bean 即可。

代码实现

```
package com.southwind.myspring;

import lombok.AllArgsConstructor;
import lombok.Data;

@Data
@AllArgsConstructor
public class BeanDefinition {
    private String beanName;
    private Class beanClass;
}
```

```java
package com.southwind.myspring;

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

@Target(ElementType.FIELD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Autowired {
}
```

```java
package com.southwind.myspring;

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

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

```java
package com.southwind.myspring;

import java.lang.reflect.Field;
```

```
import
java.lang.reflect.InvocationTargetException
;
import java.lang.reflect.Method;
import java.util.*;

public class
MyAnnotationConfigApplicationContext {

    private Map<String,Object> ioc = new
HashMap<>();
    private List<String> beanNames = new
ArrayList<>();

    public
MyAnnotationConfigApplicationContext(String
pack) {
        //遍历包，找到目标类(原材料)
        Set<BeanDefinition> beanDefinitions
= findBeanDefinitions(pack);
        //根据原材料创建bean
        createObject(beanDefinitions);
        //自动装载
        autowireObject(beanDefinitions);
    }

    public void
autowireObject(Set<BeanDefinition>
beanDefinitions){
```

```
        Iterator<BeanDefinition> iterator =
beanDefinitions.iterator();
        while (iterator.hasNext()) {
            BeanDefinition beanDefinition =
iterator.next();
            Class clazz =
beanDefinition.getBeanClass();
            Field[] declaredFields =
clazz.getDeclaredFields();
            for (Field declaredField :
declaredFields) {
                Autowired annotation =
declaredField.getAnnotation(Autowired.class
);
                if(annotation!=null){
                    Qualifier qualifier =
declaredField.getAnnotation(Qualifier.class
);
                    if(qualifier!=null){
                        //byName
                        try {
                            String beanName
= qualifier.value();
                            Object bean =
getBean(beanName);
                            String
fieldName = declaredField.getName();
                            String
methodName = "set"+fieldName.substring(0,
1).toUpperCase()+fieldName.substring(1);
```

```
                                Method method =
clazz.getMethod(methodName,
declaredField.getType());
                                Object object =
getBean(beanDefinition.getBeanName());

 method.invoke(object, bean);
                            } catch
(NoSuchMethodException e) {

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

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

 e.printStackTrace();
                            }
                        }else{
                            //byType
                        }
                    }
                }
            }
        }

    public Object getBean(String beanName){
        return ioc.get(beanName);
```

```
    }

    public String[]
getBeanDefinitionNames(){
        return beanNames.toArray(new
String[0]);
    }

    public Integer getBeanDefinitionCount()
{
        return beanNames.size();
    }

    public void
createObject(Set<BeanDefinition>
beanDefinitions){
        Iterator<BeanDefinition> iterator =
beanDefinitions.iterator();
        while (iterator.hasNext()) {
            BeanDefinition beanDefinition =
iterator.next();
            Class clazz =
beanDefinition.getBeanClass();
            String beanName =
beanDefinition.getBeanName();
            try {
                //创建的对象
                Object object =
clazz.getConstructor().newInstance();
                //完成属性的赋值
```

```
                Field[] declaredFields =
clazz.getDeclaredFields();
                for (Field declaredField :
declaredFields) {
                    Value valueAnnotation =
declaredField.getAnnotation(Value.class);

 if(valueAnnotation!=null){
                        String value =
valueAnnotation.value();
                        String fieldName =
declaredField.getName();
                        String methodName =
"set"+fieldName.substring(0,
1).toUpperCase()+fieldName.substring(1);
                        Method method =
clazz.getMethod(methodName,declaredField.ge
tType());
                        //完成数据类型转换
                        Object val = null;
                        switch
(declaredField.getType().getName()){
                            case
"java.lang.Integer":
                                val =
Integer.parseInt(value);
                                break;
                            case
"java.lang.String":
```

```
                                    val =
value;
                                    break;
                                case
"java.lang.Float":
                                    val =
Float.parseFloat(value);
                                    break;
                            }

 method.invoke(object, val);
                        }
                    }
                    //存入缓存
                    ioc.put(beanName, object);
                } catch (InstantiationException
e) {
                    e.printStackTrace();
                } catch (IllegalAccessException
e) {
                    e.printStackTrace();
                } catch
(InvocationTargetException e) {
                    e.printStackTrace();
                } catch (NoSuchMethodException
e) {
                    e.printStackTrace();
                }
            }
        }
```

```
    public Set<BeanDefinition>
findBeanDefinitions(String pack){
        //1、获取包下的所有类
        Set<Class<?>> classes =
MyTools.getClasses(pack);
        Iterator<Class<?>> iterator =
classes.iterator();
        Set<BeanDefinition> beanDefinitions
= new HashSet<>();
        while (iterator.hasNext()) {
            //2、遍历这些类，找到添加了注解的类
            Class<?> clazz =
iterator.next();
            Component componentAnnotation =
clazz.getAnnotation(Component.class);
            if(componentAnnotation!=null){
                //获取Component注解的值
                String beanName =
componentAnnotation.value();
                if("".equals(beanName)){
                    //获取类名首字母小写
                    String className =
clazz.getName().replaceAll(clazz.getPackage
().getName() + ".", "");
                    beanName =
className.substring(0,
1).toLowerCase()+className.substring(1);
                }
```

```
                //3、将这些类封装成
BeanDefinition，装载到集合中
                beanDefinitions.add(new
BeanDefinition(beanName, clazz));
                beanNames.add(beanName);
            }
        }
        return beanDefinitions;
    }
}
```

```
package com.southwind.myspring;

import java.io.File;
import java.io.FileFilter;
import java.io.IOException;
import java.net.JarURLConnection;
import java.net.URL;
import java.net.URLDecoder;
import java.util.Enumeration;
import java.util.LinkedHashSet;
import java.util.Set;
import java.util.jar.JarEntry;
import java.util.jar.JarFile;

public class MyTools {

    public static Set<Class<?>>
getClasses(String pack) {
```

```
        // 第一个class类的集合
        Set<Class<?>> classes = new
LinkedHashSet<Class<?>>();
        // 是否循环迭代
        boolean recursive = true;
        // 获取包的名字 并进行替换
        String packageName = pack;
        String packageDirName =
packageName.replace('.', '/');
        // 定义一个枚举的集合 并进行循环来处理这个
目录下的things
        Enumeration<URL> dirs;
        try {
            dirs =
Thread.currentThread().getContextClassLoade
r().getResources(packageDirName);
            // 循环迭代下去
            while (dirs.hasMoreElements())
{
                // 获取下一个元素
                URL url =
dirs.nextElement();
                // 得到协议的名称
                String protocol =
url.getProtocol();
                // 如果是以文件的形式保存在服务
器上
                if
("file".equals(protocol)) {
                    // 获取包的物理路径
```

```
                    String filePath =
URLDecoder.decode(url.getFile(), "UTF-8");
                    // 以文件的方式扫描整个包下
的文件 并添加到集合中

 findClassesInPackageByFile(packageName,
filePath, recursive, classes);
                } else if
("jar".equals(protocol)) {
                    // 如果是jar包文件
                    // 定义一个JarFile
                    System.out.println("jar
类型的扫描");
                    JarFile jar;
                    try {
                        // 获取jar
                        jar =
((JarURLConnection)
url.openConnection()).getJarFile();
                        // 从此jar包 得到一个
枚举类

 Enumeration<JarEntry> entries =
jar.entries();

 findClassesInPackageByJar(packageName,
entries, packageDirName, recursive,
classes);
                    } catch (IOException e)
{
```

```
                                // log.error("在扫描
用户定义视图时从jar包获取文件出错");

 e.printStackTrace();
                            }
                        }
                    }
                } catch (IOException e) {
                    e.printStackTrace();
                }
                return classes;
            }

            private static void
findClassesInPackageByJar(String
packageName, Enumeration<JarEntry> entries,
String packageDirName, final boolean
recursive, Set<Class<?>> classes) {
                // 同样的进行循环迭代
                while (entries.hasMoreElements()) {
                    // 获取jar里的一个实体 可以是目录 和
一些jar包里的其他文件 如META-INF等文件
                    JarEntry entry =
entries.nextElement();
                    String name = entry.getName();
                    // 如果是以/开头的
                    if (name.charAt(0) == '/') {
                        // 获取后面的字符串
                        name = name.substring(1);
                    }
```

```
                // 如果前半部分和定义的包名相同
                if
(name.startsWith(packageDirName)) {
                    int idx =
name.lastIndexOf('/');
                    // 如果以"/"结尾 是一个包
                    if (idx != -1) {
                        // 获取包名 把"/"替换成"."
                        packageName =
name.substring(0, idx).replace('/', '.');
                    }
                    // 如果可以迭代下去 并且是一个包
                    if ((idx != -1) ||
recursive) {
                        // 如果是一个.class文件 而
且不是目录
                        if
(name.endsWith(".class") &&
!entry.isDirectory()) {
                            // 去掉后面的".class"
获取真正的类名
                            String className =
name.substring(packageName.length() + 1,
name.length() - 6);
                            try {
                                // 添加到classes

 classes.add(Class.forName(packageName +
'.' + className));
```

```
                            } catch
(ClassNotFoundException e) {
                                // .error("添加
用户自定义视图类错误 找不到此类的.class文件");

 e.printStackTrace();
                            }
                        }
                    }
                }
            }
        }

    private static void
findClassesInPackageByFile(String
packageName, String packagePath, final
boolean recursive, Set<Class<?>> classes) {
        // 获取此包的目录 建立一个File
        File dir = new File(packagePath);
        // 如果不存在或者 也不是目录就直接返回
        if (!dir.exists() ||
!dir.isDirectory()) {
            // log.warn("用户定义包名 " +
packageName + " 下没有任何文件");
            return;
        }
        // 如果存在 就获取包下的所有文件 包括目录
        File[] dirfiles = dir.listFiles(new
FileFilter() {
```

```
            // 自定义过滤规则 如果可以循环(包含子目录) 或则是以.class结尾的文件(编译好的java类文件)
            @Override
            public boolean accept(File file) {
                return (recursive && file.isDirectory()) || (file.getName().endsWith(".class"));
            }
        });
        // 循环所有文件
        for (File file : dirfiles) {
            // 如果是目录 则继续扫描
            if (file.isDirectory()) {

 findClassesInPackageByFile(packageName + "." + file.getName(), file.getAbsolutePath(), recursive, classes);
            } else {
                // 如果是java类文件 去掉后面的.class 只留下类名
                String className = file.getName().substring(0, file.getName().length() - 6);
                try {
                    // 添加到集合中去
```

```
                        //
classes.add(Class.forName(packageName + '.'
+
                        // className));
                        // 经过回复同学的提醒，这里
用forName有一些不好，会触发static方法，没有使用
classLoader的load干净

 classes.add(Thread.currentThread().getCont
extClassLoader().loadClass(packageName +
'.' + className));
                    } catch
(ClassNotFoundException e) {
                        // log.error("添加用户自
定义视图类错误 找不到此类的.class文件");
                        e.printStackTrace();
                    }
                }
            }
        }

}
```

```java
package com.southwind.myspring;

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

@Target(ElementType.FIELD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Qualifier {
    String value();
}
```

```java
package com.southwind.myspring;

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

@Target(ElementType.FIELD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Value {
    String value();
}
```

```java
package com.southwind.myspring.entity;
```

```java
import com.southwind.myspring.Component;
import com.southwind.myspring.Value;
import lombok.Data;

@Data
@Component("myOrder")
public class Order {
    @Value("xxx123")
    private String orderId;
    @Value("1000.5")
    private Float price;
}
```

```java
package com.southwind.myspring.entity;

import com.southwind.myspring.Autowired;
import com.southwind.myspring.Component;
import com.southwind.myspring.Qualifier;
import com.southwind.myspring.Value;
import lombok.Data;

@Data
@Component
public class Account {
    @Value("1")
    private Integer id;
    @Value("张三")
    private String name;
    @Value("22")
    private Integer age;
```

```
    @Autowired
    @Qualifier("myOrder")
    private Order order;
}
```

```
package com.southwind.myspring;

public class Test {
    public static void main(String[] args)
{

 MyAnnotationConfigApplicationContext
applicationContext = new
MyAnnotationConfigApplicationContext("com.s
outhwind.myspring.entity");

 System.out.println(applicationContext.getB
eanDefinitionCount());
        String[] beanDefinitionNames =
applicationContext.getBeanDefinitionNames()
;
        for (String beanDefinitionName :
beanDefinitionNames) {

 System.out.println(beanDefinitionName);

 System.out.println(applicationContext.getB
ean(beanDefinitionName));
        }
    }
}
```